



BIBLIOTECĂ
de
MATEMATICĂ-INFORMATICĂ

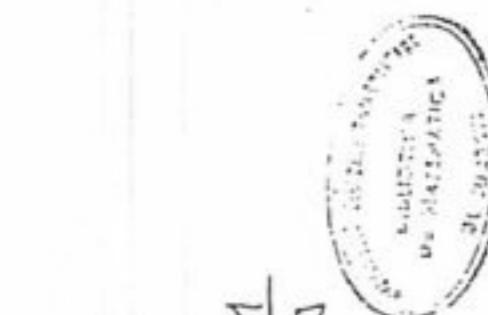
Cota 4 27642
Inventar 6229654

4 27642

RODICA CETERCHI

**STRUCTURI DE DATE ȘI ALGORITMI
ASPECTE MATEMATICE ȘI APLICAȚII**

**Partea I
STRUCTURI DE DATE ELEMENTARE**



editura universității din bucurești*

Cuprins

Referenți științifici: Prof. dr. George GEORGESCU
Prof. dr. Alexandru MATEESCU
Prof. dr. Ioan TOMESCU

© Editura Universității din București
Sos. Panduri 90-92, București - 76235; Tel./Fax: 410.23.84
E-mail: editura@unibuc.ro
Internet: www.editura.unibuc.ro

Biblioteca de Matematică



1100 022 9657

Descrierea CIP a Bibliotecii Naționale a României
CETERCHI, RODICA

Structuri de date: aspecte matematice și
aplicații / Rodica Ceterchi - București, Editura
Universității din București, 2001

p. ; cm.

Bibliogr.

ISBN 973-575-594-7

004

Introducere în studiul structurilor de date	5
I. Structuri lineare	9
1. Structuri lineare în alocare statică	11
2. Structuri lineare în alocare dinamică. Liste simplu înlățuite	21
3. Alte tipuri de liste. Aplicații ale listelor înlățuite	33
4. Structuri lineare cu restricții la intrare/ieșire: stive și cozi	39
II. Structuri arborescente	49
1. Arbori	51
2. Arbori binari	58
3. Arbori binari - similaritate și echivalentă	66
4. Arbori binari de căutare	73
5. Arbori binari de căutare echibrăți AVL	89
III. Sortări interne	99
1. Sortarea prin inserție directă	101
2. Sortarea prin selecție directă	105
3. Sortarea prin interschimbare directă	108
④ Sortarea prin inserție cu micsorarea incrementului (sortarea Shell)	114
⑤ Sortarea prin selecție folosind structuri arborescente (sortarea cu ansamblu)	116
⑥ Sortarea prin interschimbare folosind partiții (QuickSort)	125
⑦ Interclasarea a două siruri ordonate. Sortarea prin interclasare	135
IV. Arbori binari stricți. Aplicații	141
⑧ Arbori binari stricți. Definiție. Proprietăți	143
2. Limita inferioară a algoritmilor de sortare bazați pe comparații între chei	155
3. Arbori binari stricți cu ponderi. Algoritmul lui Huffman	157
4. Aplicații la codificare. Coduri Huffman	161
5. Interclasarea optimală a mai multor siruri	165
Teste grilă	167
Bibliografie	177

Introducere în studiul structurilor de date

Structurile de date au apărut ca și materie distinctă în informatică în curriculum ACM încă din 1968. O formulă semnificativă care le plasează în contextul programării este dată de titlul unei celebre cărți a lui N. Wirth "Algorithms + Data Structures = Programs". Într-adevăr, ecuația "algoritmi + structuri de date = programe" pune în evidență faptul că orice problemă de programare presupune, pe lângă găsirea unuia sau mai multor algoritmi, alegerea unei structuri de date adecvate pentru menținerea datelor problemei. Iar de multe ori, găsirea algoritmului și a structurii adecvate merg mână în mână, după cum vom vedea.

Mulțimile de date pe care le manipulează programele sunt în general colecții *finite*, de date *omogene*, adică date de același tip. Tipul unui element al mulțimii poate fi foarte complex, dar, în general, un element va fi identificat printr-o valoare unică numită *cheie*. O structură de date menține o asemenea colecție finită într-o mulțime cu o anumită organizare internă - structura. Organizarea internă a datelor va influența, și în același timp depinde de *operațiile* ce trebuie efectuate pe elementele mulțimii, de aceea orice structură de date vine însotită de o listă de operații ce se efectuează pe ea și de algoritmi care implementează aceste operații.

Dăm în continuare o listă a operațiilor de bază pe structuri de date, împreună cu o descriere a lor.

Traversarea unei structuri este operația care accesează fiecare element o singură dată în vederea procesării; pentru procesarea unui element în cadrul traversării se mai folosește denumirea de *vizitare* a elementului respectiv.

Căutarea este operația care caută un element cu cheie dată într-o structură și se termină cu sau fără succes. Ea constă dintr-o traversare, eventual incompletă, a structurii, în care vizitarea fiecărui element revine la a-l compara cu cel căutat. Dacă elementul căutat există în structură, nu se mai continuă traversarea și locul primei apariții este returnat, căutarea încheindu-se cu succes. Dacă elementul căutat nu există în structură traversarea va trebui să fie completă pentru a putea decide aceasta, și căutarea se va termina fără succes.

Inserarea unui nou element într-o structură dată este operația care adaugă un element nou, de același tip, unei colecții finite de elemente reprezentate structurat. "Adăugarea" se face astfel încât noua structură, cu un element în plus, să fie de același tip cu structura inițială.

Lucrarea de față conține material prezentat în prima parte a cursului de "Structuri de date și algoritmi" pe care autoarea îl fine la Facultatea de Matematică a Universității din București, în cadrul Colegiului de Informatică și al secției Matematică-Informatică. Ea se adresează în primul rând acestor studenți, dar nu numai lor, lucrarea putând fi utilă tuturor celor care doresc să-și îmbogățească sau să-și sistematizeze cunoștințele din domeniul.

Se așteaptă din partea cititorului să posede cunoștințe generale de informatică, în special programare (metode de programare, noțiunea de algoritm și cea de complexitate) și cunoașterea cel puțin a limbajului Pascal.

Autoarea mulțumește pe această cale colaboratorilor pe care i-a avut de-a lungul anilor și care și-au adus contribuția la buna desfășurare a cursului. Mulțumește de asemenea acestor studenți ai anului II serie 2000-2001 care au contribuit la tehnoredactarea lucrării, precum și Editurii Universității din București pentru sprijinul acordat în vederea publicării.

Stergerea sau extragerea unui element dintr-o structură este operația inversă a inserării. Ea "scoate" un element dintr-o mulțime structurată dată, refăcând structura inițială pe elementele rămase. Elementul extras poate fi folosit pentru diverse procesări, printre care și alte operații pe structuri. Există două aspecte ale acestei operații ce trebuie luate în considerare: aspectul de "extragere", când accentul cade pe elementul extras din structură, și aspectul de stergere, când accentul cade pe elementele rămase și pe refacerea tipului inițial de structură pe ele. Oricum am denumi operația, trebuie să ne ocupăm de ambele aspecte de mai sus.

Operațiile de inserare și stergere sunt foarte importante pentru reprezentarea mulțimilor care au un caracter *dinamic* în timp, mulțimi care își schimbă frecvent componența. În problemele care manipulează mulțimi dinamice, trebuie să le reprezentăm pe acestea cu ajutorul unor structuri de date pe care operațiile de inserare și stergere să se efectueze rapid și necostisitor. Atât inserarea cât și stergerea unui element sunt precedate în general de operații de căutare a locului inserării în (respectiv stergerii din) structură, deci performanțele acestor operații depind de performanța operației de căutare. La rândul ei, căutarea depinde de operația de traversare.

Acste patru tipuri de operații de bază pentru menținerea mulțimilor dinamice, sunt însoțite și de operații auxiliare (dar absolut necesare), precum *initializarea unei structuri* și *crearea unei structuri*. În general, crearea unei structuri de un anume tip se face în doi pași:

- (1) *initializarea structurii cu structura vidă*
- (2) un ciclu repetitiv de lungime variabilă în care
 - (a) se ia câte un element dintr-un fișier de intrare;
 - (b) pentru fiecare asemenea element se apelează o procedură ce implementează operația de *inserare*.

Pentru toate structurile de date discutate în această lucrare vom codifica modul în care se reprezintă structura vidă, iar operațiile de creare vor fi ca mai sus, bazate pe initializări, urmate de inserări repetitive.

Structurile elementare de date discutate în primele două capitole, cele lineare și cele arborescente, vor avea implementate pe ele toate operațiile de mai sus.

Combinarea a două structuri este o altă operație importantă. Ea produce din două structuri de același tip o alta, de același tip, ce conține elementele reunite ale primelor două. Vom vedea în capitolul III secțiunea 7 un prim exemplu de operație de combinare, interclasarea a două siruri ordonate. În particular, și inserarea poate fi privită ca și caz

particular de combinare, una din structurile ce sunt date de intrare pentru combinare având un singur element.

Sortarea este o altă operație extrem de importantă pentru multe aplicații. Ea constă în ordonarea totală a elementelor unei mulțimi. Trebuie tratat separat cazul mulțimilor statice de al celor dinamice.

Pentru cazul mulțimilor *statice* (acele mulțimi care nu-și schimbă componența în timp) ne putem pune problema sortării elementelor ei o singură dată. În plus, ne putem pune problema sortării *in situ*, adică "pe loc", folosind aceleași locații ca și datele de intrare (mulțimea nesortată), căci mulțimea nu-și schimbă componența. Capitolul III se ocupă de studiul algoritmilor de sortare *in situ* și bazați pe comparații între chei.

Dacă avem de a face cu mulțimi dinamice pe care se pune și problema sortării, atunci atunci vom lucra cu tipuri speciale de structuri, în care relațiile între chei joacă și ele un rol în organizarea structurii. Menționăm structurile lineare ordonate, arborii binari de căutare, arborii parțiali ordonați, și.a., structuri pe care vom avea algoritmi specifici de inserare și stergere, și operații ce pot produce la cerere o versiune sortată a mulțimii.

O altă operație importantă pe structuri este operația de *extragere a minimului* (respectiv maximului) dintr-o structură. Ea poate fi privită ca un caz particular de extragere, dintr-o structură ordonată, deci are legătură și cu sortarea. Vom vedea că există structuri care optimizează performanța acestei operații, făcând extragerea propriu-zisă în timp $O(1)$ și refacerea structurii pe datele rămase în timp $O(\log n)$; este vorba despre structura de ansamblu studiată în capitolul III, secțiunea 5.

În cele ce urmează vom înțelege prin structură de date de un anume tip nu numai organizarea internă a datelor și modul de acces la ele, ci și *gestionarea structurii*, adică operațiile care se execută pe respectiva structură, cu algoritmi specifici ce implementează aceste operații. De exemplu, un arbore binar de căutare va fi structura caracterizată nu numai prin definiția de la începutul secțiunii 4 a capitolului II, ci și prin algoritmi specifici de căutare, inserare și stergere prezentați acolo. Analog, structurile de stivă și de coadă prezentate în capitolul I secțiunea 4 sunt structuri lineare cu operații specifice de inserare și stergere, și pe care nu vom face traversări și căutări (decât folosind eventual în mod repetat operația de stergere).

Structurile de date se împart în două mari categorii: cele *lineare*, pe care le vom prezenta în capitolul I, și cele *nelineare* - arborii și grafurile. Dintre acestea din urmă prezentăm pe scurt doar arborii, în capitolul II, cu accent special pe arborii binari. Studiul arborilor binari se

continuă în capitolul IV cu clasa arborilor binari stricți. Proprietățile demonstrate aici pentru arbori binari stricți se transpun imediat în proprietăți ale arborilor binari oarecare.

Pentru reprezentarea unei structuri pentru o problemă de programare, care se finalizează printr-un program într-un anume limbaj de programare, avem nevoie de tipuri de date specifice, bazate pe tipurile predefinite ale limbajului. Mai avem nevoie de asemenea să ne punem problema alocării spațiului pentru structură, modurile de alocare depinzând din nou de facilitățile oferite de limbaj. Indiferent de acestea însă, ne vom pune problema de a lucra cu *spațiu în alocare statică* și cu *spațiu în alocare dinamică*, într-un mod pe care îl vrem suficient de general pentru a putea utiliza structurile și implementa algoritmii în cât mai multe limbaje de programare.

Algoritmii prezentați în lucrare sunt descriși atât în limbaj natural, cât și ca proceduri în pseudo-cod (mai rar chiar cod Pascal). Pseudo-codul folosit este extrem de asemănător Pascal-ului, fiind ușor de transformat în cod. El pune în evidență modul în care se *structurează* respectivul algoritm, structurare pe care vrem să o regăsim intactă chiar dacă schimbăm modul de reprezentare al datelor (trecând de la alocare statică la alocare dinamică, de exemplu) sau dacă trecem la alt limbaj de programare. Cu alte cuvinte, am vrut să facem materialul utilizabil și celor care vor să programeze în alte limbaje decât Pascal.

Câteva caracteristici ale pseudo-codului folosit: nu declarăm tipul și modul de transmitere al variabilelor pentru proceduri; nu declarăm variabilele locale; eliminăm secvențele *begin...end* preferând să marcăm corpul unei instrucțiuni cu secvențe de tipul *while...endwhile*, *if...endif*, *if...else...endif*, *for...endfor*, etc. Vor exista și secvențe repetitive pentru care ieșirea nu se face la unul din capete, pentru care am folosit notația *do...repeat*, dar care se pot transforma în cicluri uzuale și implementa în cod cu ajutorul unor variabile suplimentare. Sperăm că, această scriere facilitează implementarea în Pascal, pe care o considerăm un exercițiu simplu dar util.

Un ultim cuvânt despre performanța algoritmilor prezentați în lucrare. Adevararea unei structuri la un tip de problemă depinde de performanța operațiilor ce se efectuează pe ea. Am analizat performanța algoritmilor prezentați în funcție de două caracteristici: *spațiul utilizat* și *timpul de rulare*, iar acesta din urmă este evaluat în funcție de operațiile de bază ale algoritmului. Presupunem cunoscută notația $O(\cdot)$ pentru complexitate.

Capitolul I

Structuri lineare

Cel mai simplu tip de structură pentru reprezentarea unei colecții finite de date este structura lineară. Ea presupune existența unei ordini totale pe mulțimea datelor și un anumit tip de accesare a lor, accesarea secvențială: știm care este primul element al colecției și, din fiecare element, putem să-l accesăm pe următorul.

Pentru structurile lineare, ca și pentru celealte tipuri de structuri, avem două posibilități de alocare de spațiu: alocarea statică și cea dinamică. Vom prezenta deci structuri lineare în alocare statică (vectori) și structuri lineare în alocare dinamică (liste înțălnițite), împreună cu operațiile de bază pe ele: traversarea, căutarea, inserarea și ștergerea. Să nu uităm însă că mulțimile cu care lucrăm au în general un caracter dinamic; componența mulțimii se schimbă frecvent prin operații de inserări și ștergeri, deci chiar lucrul cu un tip de dată cunoscut, vectorul, va fi de data aceasta diferit, cind vectorul nu reprezintă decât un mod particular de alocare de spațiu pentru o structură lineară.

Vom analiza complexitatea operațiilor, în special a operațiiei de căutare, care precede și operațiile de inserare/ștergere. Vom stabili raportul dintre avantajele și dezavantajele unui tip de alocare față de celălalt în funcție de operațiile care se efectuează cel mai frecvent.

În ultimul paragraf vom studia tipuri particulare de structuri lineare, structurile lineare cu restricții la intrare/ieșire, adică acele structuri lineare în care inserările și ștergerile se fac numai la anumite capete prestabilite. Stivele și cozile sunt exemple clasice de asemenea structuri, cu aplicații deosebit de numeroase, atât la programare, cât și pentru sistemele de operare la gestionarea resurselor.

1.1. Structuri lineare în alocare statică

Numele structură lineară în alocare statică, o colecție finită de date omogene (date de aceeași tip) cu următoarele caracteristici:

(a) Datele sunt indexate după o mulțime finită, total ordonată, de n întregi consecutivi (numită mulțimea indicilor);

(b) Datele ocupă locații succesive în memorie.

O asemenea structură mai poartă numele de vector și orice limbaj de programare de nivel înalt are tipuri de date specifice pentru lucru cu vectori. În Pascal este tipul *array*. Cu ajutorul lui vom defini tipul *vector*, cu

$$\text{type vector} = \text{array}[l..Max] \text{ of } <\text{tip_de_bază}>.$$

O variabilă A de tip vector (declarată cu *var A: vector*) va ocupa în memorie Max locații succesive de dimensiune w , unde w este lungimea în cuvinte a tipului *<tip_de_bază>* al componentelor. Caracteristica (b) din definiția de mai sus se reflectă asupra modului în care se lucrează cu datele de tip vector. Pentru variabila A de mai sus, calculatorul menține adresa primului element al lui A , numită și adresa de bază, să o notăm *Base(A)*. Vom nota $\text{Loc}(A[k])$ = adresa elementului $A[k]$. Pentru accesarea componentei $A[k]$ se calculează adresa ei cu formula

$$\text{Loc}(A[k]) = \text{Base}(A) + w(k-l)$$

dacă l este primul indice al lui A , sau, pentru $A[LB..UB]$ cu formula

$$\text{Loc}(A[k]) = \text{Base}(A) + w(k-LB).$$

Pentru orice valoare a indicelui k , timpul de calcul al formulei este același. Mai mult, dat fiind un indice k , un program poate accesa $A[k]$ fără a inspecta celelalte componente ale lui A . Vom spune că timpul de acces la oricare componentă $A[k]$ este $O(1)$, și acesta este unul din mariile avantaje ale structurii.

Să presupunem acum că avem n elemente depuse în n locații succesive ale unui vector A ca mai sus. Evident, n trebuie să fie mai mic decât Max . Presupunem de asemenea că ele ocupă locații indexate de la l la n . Să vedem cum se implementează operațiile de traversare, inserare, stergere și căutare pe o asemenea structură.

Traversarea unei structuri lineare în alocare statică

Operația de traversare este extrem de simplă pe structurile lineare.

```
procedure Traversare(A, l, n)
k := 1; {initializarea indicelui pentru traversare}
```

12

```

while k <= n do {test pentru nedepășirea structurii}
    vizitează A[k];
    k := k+1; {trecem la componenta următoare}
endwhile
endproc

```

Evident, putem folosi avantajele instrucțiunii *for* și atunci procedura de traversare devine:

```

procedure Traversare1(A, l, n)
for i := 1 to n do
    vizitează A[i];
endfor
endproc

```

Inserarea și ștergerea unui element al unei structuri lineare

Pentru operația de inserare a unui nou element, *Elem* în structura menținută în $A[l \dots n]$ trebuie să avem în primul rând spațiu, adică $n < Max$. Dacă condiția e îndeplinită, avem $Max - n$ locații libere în care putem să inserăm tot atâtca elemente. Inserarea la sfârșitul structurii, pe componenta $n-1$, nu pune probleme. Dar dacă vrem să inserăm într-o poziție k din vector, aflată între l și n , atunci va trebui să mutăm toate componentele de la k la n căte bă locație la dreapta pentru a face loc nouui element.

```

procedure Insert(A, l, n, k, Elem)
{inserează în structura lineară A[l .. n], pe poziția k, valoarea lui Elem}
{mută pe rând elementele de la A[n] până la A[k] căte o locație la dreapta}
    i := n;
    while i >= k do
        A[i+1] := A[i];
        i := i-1;
    endwhile
    {inserarea propriu-zisă}
    A[k] := Elem;
    {crește dimensiunea structurii}
    n := n+1;
endproc

```

Dacă vrem să facem inserarea cu procedura de mai sus, atunci se presupune că, înainte de apelul procedurii am făcut următoarele două operații strict necesare: (1) un test pentru a verifica dacă mai există locații libere, $n < Max$, deci inserarea e posibilă; (2) un test pentru a verifica dacă k este o poziție validă de inserare (pentru a păstra contiguitatea datelor), adică $l \leq k \leq n+1$.

Operația de ștergere sau de extragere a unei valori din vector este inversă inserării. Ea produce o variabilă X în care se găsește elementul extras și o nouă structură lineară obținută prin reorganizarea elementelor rămase. Evident, nu putem extrage decât dintr-o structură ce conține cel puțin un element, deci pentru care $n \geq 1$. Din nou, dacă extragerea s-ar face numai la capătul din dreapta, adică dacă s-ar extrage doar $A[n]$ ea nu ar pune probleme. Dacă se extrage însă de pe o componentă k oarecare, atunci "reorganizarea" elementelor rămase implică din nou un sir de mutări astfel încât ele să rămână contigue.

```

procedure Delete(A, l, n, k, X)
{extrage în X valoarea A[k] și reface vectorul}
{extragerea propriu-zisă}
X := A[k];
{refacerea structurii de vector}
for i := k to n-1 do
    A[i] := A[i+1];
endfor
{scade dimensiunea structurii}
n := n-1;
endproc

```

Dacă facem ștergerea cu procedura de mai sus, atunci putem presupune că, înainte de apelul ei, am făcut următoarele două operații strict necesare: (1) am verificat că structura are cel puțin un element, $n \geq 1$, deci ștergerea are sens; (2) am verificat că k este o poziție validă de ștergere, adică $l \leq k \leq n+1$.

În general, operațiile de inserare/ștergere într-o structură dată, în cazul de față într-o structură lineară în alocare statică, se fac pe poziții oarecare în interiorul structurii, poziții care trebuie căutate și găsite cu ajutorul unor traversări, eventual incomplete, guvermate de o anumită condiție, în funcție de natura problemei. Cu alte cuvinte, în multe

probleme reale, poziția k pentru inserare/ștergere nu este cunoscută dinainte, deci nu va fi transmisă ca parametru, ci trebuie găsită, cu o traversare guvernată de o condiție specifică problemei și, evident, de condiția de nedepășire a structurii.

Traversarea se poate face pe acest tip de structură lineară în ambele sensuri: de la stânga la dreapta, adică în sensul crescător al indicilor sau de la dreapta la stânga, adică în sensul descrescător al indicilor.

Traversarea de la stânga la dreapta se face în felul următor:

```
i:=1
while (i<=n) and {condiția de inserare/ștergere nu e îndeplinită} do
    i:=i+1
endwhile
```

Traversarea de la dreapta la stânga se face în felul următor:

```
i:=n
while (i>=1) and {condiția de inserare/ștergere nu e îndeplinită} do
    i:=i-1
endwhile
```

Ambele tipuri de traversări sunt guvernate de: o condiție de nedepășire a structurii, apoi de o condiție a cărei formulare depinde de natura problemei. În funcție de această a doua condiție, la terminarea traversării putem decide dacă locul inserării sau ștergerii a fost găsit și, dacă da, care este acesta, și scriem în continuare codul care implementează inserarea sau ștergerea propriu-zisă, cu mutările aferente, și cu modificarea dimensiunii.

Să observăm că operațiile de inserare și ștergere dintr-un vector sunt costisoare în termeni de *mutări* de componente. Putem să evaluăm numărul de mutări M .

Fie p_i = probabilitatea evenimentului de a insera o valoare nouă pe componenta i , $i \in [1..n]$. La inserarea pe poziția i trebuie să mutăm $n-i+1$ componente. Numărul mediu de mutări la inserare va fi deci:

$$M = \sum_{i=1}^n p_i (n - i + 1).$$

Dacă inserarea se face cu probabilități egale pe orice poziție, adică $p_1 = \dots = p_n = \frac{1}{n}$, atunci:

$$M = \frac{1}{n} (n + (n-1) + \dots + 1) = \frac{1}{n} * \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

Analog, fie p_i = probabilitatea evenimentului de a șterge componenta $A[i]$, $i \in [1..n]$. La ștergerea lui $A[i]$ trebuie să mutăm $n-i$ componente. Numărul mediu de mutări la ștergere va fi deci:

$$M = \sum_{i=1}^n p_i (n - i).$$

Dacă ștergem cu probabilități egale orice componentă, atunci:

$$M = \frac{1}{n} ((n-1) + (n-2) + \dots + 1) = \frac{1}{n} * \frac{n(n-1)}{2} = \frac{n-1}{2}.$$

Observăm că, atât la inserarea cât și la ștergere trebuie să mutăm în medie cam jumătate din componente, $n/2$. Există și cazul cel mai nefavorabil, când inserăm sau ștergem prima componentă, deci facem numărul maxim de mutări $O(n)$. Din acest motiv, structura de vector nu este recomandată pentru mulțimi cu caracter dinamic, mulțimi care-și schimbă frecvent componența prin inserări și ștergeri.

Căutarea într-o structură lineară în alocare secvențială

O operație elementară pe orice structură de date este operația de căutare a unei valori de date, *Val*. În cazul structurilor lineare avem căutarea lineară care este în esență o traversare, eventual incompletă a structurii, în care vizitarea revine la a compara elementul vizitat cu valoarea căutată. Ea se termină cu succes dacă am găsit valoarea *Val* și returnează indicele pe care am găsit-o, și fără succes dacă termină de traversat structura fără să găsească pe *Val*.

procedure SearchLin (A, 1, n, Val, Loc)

{caută linear valoarea *Val* în $A[1..n]$ și returnează $Loc = 0$ dacă nu o găsește, și o valoare $Loc \in [1..n]$ dacă o găsește pe componenta $A[Loc]$ }

```
Loc := 0
i := 1;
while (i <= n) and (A[i] <= Val) do
    i := i+1
endwhile
if i <= n then
    Loc := i
endif
endproc {SearchLin}
```

Observăm că ciclul repetitiv *while* în care se face traversarea eventual incompletă a structurii este guvernat de două condiții: condiția de nedepășire a structurii, $i \leq n$ și condiția de a nu fi găsit încă pe *Val*, $A[i] \neq Val$. Se poate elimina condiția de nedepășire a structurii, reducând în felul acesta numărul de comparații la jumătate, prin *tehnica componentei marcaj*. Se adaugă structurii o locație în plus la capătul unde se termină traversarea, deci pe componenta $n+1$ dacă traversăm de la stânga la dreapta ca mai sus. (Dacă traversăm de la dreapta la stânga se adaugă o componentă $A[0]$.)

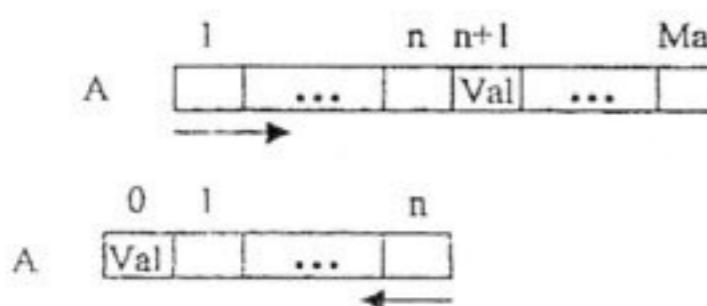


Fig. 1.1.1. Componente marcaj pentru traversarea cu căutare.

Se pune valoarea căutată pe componenta marcaj. În felul acesta căutarea se va termina întotdeauna cu succes, valoarea *Val* fiind găsită pe componenta marcaj dacă nu apare în restul structurii. Găsirea ei pe componenta marcaj codifică situația "căutare fără succes".

```
procedure SearchLin1 (A, l, n, Val, Loc)
  A[n+1] := Val {introducem Val pe componenta marcaj}
  Loc := l
  while A[Loc] <> Val do
    Loc := Loc + 1
  endwhile
  if Loc = n+1 then
    "Căutare fără succes"
  else
    "Am găsit pe componenta Loc"
  endif
endproc; {SearchLin1}
```

Aceeași tehnică a componentei marcaj poate fi folosită și la traversările făcute în scopul căutării locului unei operații de inserare/ștergere.

Să analizăm acum complexitatea căutării lineare în termeni de elemente accesate, care este același cu numărul de comparații de la ciclul *while*. În cazul cel mai nefavorabil, când *Val* nu se găseste în structură trebuie să traversăm toată structura, deci accesăm $n+1$ componente (și pe cea marcaj). $C = n+1$.

Să analizăm acum cazul mediu. Fie p_i = probabilitatea ca $Val = A[i]$, $i \in [l..n]$ și q = probabilitatea ca *Val* să nu se găsească în $A[l..n]$. Avem

$$\sum_{i=1}^n p_i + q = 1.$$

Dacă $A[i] = Val$ (prima apariție a lui *Val* este pe componenta $A[i]$) atunci facem i comparații, deci numărul mediu de comparații va fi:

$$C = \sum_{i=1}^n p_i i - q(n+1).$$

În particular, dacă *Val* se găsește precis în vector, deci $q=0$, și se găsește cu probabilitate egală pe oricare din componente, adică $p_1 = \dots = p_n = 1/n$ avem:

$$C = \frac{1}{n} (1 \cdot 1 + 2 \cdot 2 + \dots + n) = \frac{1}{n} * \frac{n(n+1)}{2} = \frac{n+1}{2}.$$

Deci, în cazul căutării cu succes, numărul mediu de comparații este $(n+1)/2$, adică jumătate din dimensiunea vectorului. În ceea ce privește clasa de complexitate, algoritmul de căutare lineară are complexitatea $O(n)$.

Exerciții

1. Să se scrie o procedură de inserare într-o structură lineară în alocare statică cu elemente ordonate total crescător. Să se folosească această procedură de inserare pentru a crea un vector ordonat crescător.
2. Să se scrie un program care conține:
 - O procedură de inserare a unei valori date pe o poziție dată într-o structură lineară în alocare statică.
 - O procedură de ștergere a unei valori date de pe o poziție dată dintr-o structură lineară în alocare statică.
 - O procedură de creare a unui vector care apelează în mod repetat procedura de inserare de la punctul a) pentru fiecare pereche (valoare, poziție) citită.
 - O procedură de afișare a vectorului.

- e) O interfață a programului principal prin care utilizatorul să poată apela procedurile de mai sus.

Căutarea liniară într-un vector sortat

Dacă elementele unei structuri lineare au și o organizare "internă", de exemplu dacă sunt sortate crescător, adică avem

$$A[1] \leq A[2] \leq \dots \leq A[n],$$

atunci această informație în plus poate fi folosită pentru a îmbunătăți performanța operației de căutare. (Despre modalitățile de a sorta crescător un vector vom vorbi într-un capitol separat)

Este suficient să traversăm elementele strict mai mici decât Val , la primul indice i cu $A[i] \geq Val$ avem, fie egalitate, deci căutare cu succes, fie inegalitate strictă, după care continuarea parcurgerii nu mai are sens căci Val nu se găsește în A .

```
procedure SearchLinOrd (A, 1, n, Val, Loc)
    Loc:= 0
    i:= 1
    while (i <= n) and (A[i] < Val) do
        i:=i+1
    endwhile
    if i <= n then
        if A[i] = Val then
            {căutare cu succes}
            Loc:=i
        else {A[i] > Val}
            {căutare fără succes}
        endif
    else
        {căutare fără succes}
    endif
endproc{ SearchLinOrd}
```

Dar, unul din avantajele majore ale structurii de vector, și anume posibilitatea accesării în timp $O(1)$ a oricărei componente, ne permite să îmbunătățim și mai mult performanța căutării pe un vector ordonat.

Căutarea binară

Fie $A[1..n]$ un vector cu elementele ordonate crescător,

$$A[1] \leq A[2] \leq \dots \leq A[n].$$

Algoritmul de căutare binară este un exemplu clasic de metodă Divide et Impera. La fiecare pas problema se „împarte” în două sub-probleme de dimensiuni mai mici: comparând pe Val cu valoarea mediană a vectorului, dacă Val este mai mică decât aceasta, atunci are sens să continuăm căutarea ei doar pe subvectorul din stânga, iar dacă este mai mare, atunci continuăm căutarea pe subvectorul din dreapta.

Algoritmul de căutare binară are următoarea structură:

- (1) Se începe cu segmentul definit de indicii $Left := 1$ și $Right := n$
- (2) Pentru fiecare subvector $A[Left..Right]$ se repetă:
 - (a) Se calculează mijlocul segmentului $Mid := (Left + Right) \text{ div } 2$
 - (b) Se compară Val cu $A[Mid]$:
 - dacă $Val = A[Mid]$ căutarea se termină cu succes;
 - dacă $Val < A[Mid]$ se reia pasul (2) pe $[Left..Mid-1]$;
 - dacă $Val > A[Mid]$ se reia pasul (2) pe $[Mid+1..Right]$.

Următoarea procedură în pseudo-cod implementează acest algoritm:

```
procedure SearchBin(A, 1, n, Val, Loc)
    Left:= 1;
    Right:= n;
    Mid:=(Left + Right) div 2;
    Loc:= 0
    while (Left <= Right) and (Val <> A[Mid]) do
        if Val < A[Mid] then {se continuă pe subintervalul din stânga}
            Right:= Mid-1
        else {Val > A[Mid]} {se continuă pe subintervalul din dreapta}
            Left:= Mid+1
        endif
        Mid:= (Left + Right) div 2
    endwhile
    if A[Mid] = Val then
        Loc:= Mid {căutare cu succes}
    else
        Loc:= 0 {căutare fără succes}
    endif
endproc{ SearchBin}
```

Fie $C(n)$ numărul de comparații pe care îl necesită căutarea binară pe un vector cu n componente. După fiecare comparație, dimensiunea segmentului pe care căutăm se va reduce la jumătate. Dacă după $C(n)$ comparații am încheiat căutarea, atunci avem relația

$$2^{C(n)} \geq n$$

de unde

$$C(n) = \lfloor \log_2 n \rfloor + 1.$$

Deci, complexitatea căutării binare este de ordinul $O(\log_2 n)$, ceea ce reprezintă o îmbunătățire substanțială față de $O(n)$, performanța căutării liniare.

Să ne reamintim însă că doi au fost factorii care ne-au permis aplicarea acestui algoritm:

- faptul că elementele erau sortate crescător, iar operația de sortare în sine poate fi costisitoare;

- faptul că structura lineară era în alocare statică, deci permite acces în timp $O(1)$ la jumătatea unui segment. Am văzut că o asemenea structură nu este potrivită pentru un set de date pe care avem și operații frecvente de inserări și ștergeri.

Abia structura de arbore binar de căutare echilibrat AVL ne va permite să împăcăm simultan cele două cerințe: posibilitatea efectuării necostisitoare a inserărilor și ștergerilor cu performanțe de ordin $O(\log_2 n)$ pentru operația de căutare.

1.2. Structuri liniare în alocare dinamică. Liste simplu înlățuite.

Am văzut că, dacă folosim alocarea statică pentru reprezentarea unei mulțimi cu o structură lineară, ne izbim de două neajunsuri majore, și anume:

- finitudinea spațiului pre-alocat structurii: Max va fi numărul maxim de elemente pe care va putea să-l conțină lista în alocare statică $A[1..Max]$.

- necesitatea efectuării de operații de mutare la inserări și ștergeri, din cauză că elementele structurii ocupă adrese succesive din RAM.

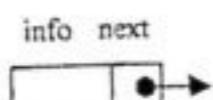
Ideal ar fi ca, la fiecare operație de inserare să putem face alocare de spațiu pentru noul element și să-l inserăm în structură cu un număr minim de operații și, invers, la ștergere să putem reface, din nou cu număr minim de operații, structura lineară pe elementele rămase, iar spațiul eliberat de componenta ce a fost șters să fie dealocat.

Limbajele procedurale precum Pascal sau C ne pun la dispoziție facilități de alocare dinamică de spațiu, precum și variabile de tip adresă (pointeri) pentru lucru cu variabile dinamice. Cel mai simplu tip de structură lineară care folosește facilitățile oferite de alocarea dinamică de spațiu este lista simplu înlățuită pe care o prezentăm în continuare.

Liste simplu înlățuite

O listă simplu înlățuită este o structură de date aranjate în ordine lineară. Spre deosebire de vectori, unde ordinea lineară a datelor este determinată de ordinea lineară pe mulțimea indicilor și datele ocupă locații successive în memorie, într-o listă simplu înlățuită ordinea lineară este determinată de existența către unui pointer, în fiecare nod, către nodul "următor" al structurii, nodurile având adrese oarecare în memorie. Accesul la întreaga structură se face având adresa primului nod.

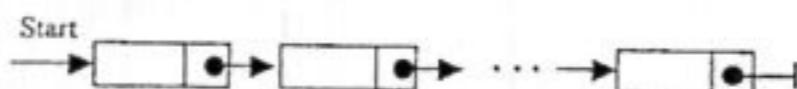
Un nod al unei liste simplu înlățuite va conține deci: (1) un set de câmpuri pe care se reprezintă un element al mulțimii; de obicei vom identifica elementul cu valoarea de pe un singur câmp, numit câmp cheie; în algoritmii care urmează putem presupune că elementul ocupă un singur câmp, *info*; (2) un pointer către nodul următor, *next*. De exemplu, în Pascal putem folosi următoarele declarații de tip pentru o listă de întregi.



```
type pnod = ^ nod;
nod = record
    info: integer;
    next: pnod
end;
```

Tipul *nod* va fi pentru nodurile listei, iar tipul *pnod* pentru adrese, adică variabile de tip pointer către noduri.

Lista în întregime va fi dată de un pointer către primul nod (variabila *Start* de tip *pnod*). Câmpul de adresă al ultimului nod va fi setat la *nil*. Lista *Start* este vidă dacă nu conține nici un element, situație care se codifică prin *Start = nil*.



Pentru structura de listă putem da și o definiție recursivă: O listă *L* de un anume tip de bază este:

- (1) fie lista vidă ($L = \emptyset$);
- (2) fie este nevidă, și atunci conține un nod numit *capul* listei, urmat de o altă listă de același tip de bază, unde prin "tip de bază" ne referim la tipul de date de pe câmpul *info*.

Observație: Liste simplu înlățuite se pot reprezenta și în alocare statică. Câmpurile *info* ocupă anumite locații ale unui vector, iar câmpurile *next* asociate vor conține indicele elementului următor. Această reprezentare se numește *reprezentarea cu cursori* a listei.

	<i>I</i>	<i>2</i>	<i>k</i>	<i>n</i>
<i>info</i>		<i>Y</i>	<i>X</i>	<i>Z</i>
<i>next</i>	<i>n</i>	2		0

Fig.1.2.1. Reprezentarea cu cursori a listei (*X,Y,Z*); lista începe la indicele *k*.

Traversarea unei liste simplu înlățuite

Spre deosebire de structurile lineare în alocare statică unde aveam posibilitatea traversării structurii "în ambele sensuri", la liste simplu înlățuite nu putem parurge structura decât într-un singur sens, accesând

primul nod, *Start* și, din fiecare nod curent *p* accesind nodul următor cu ajutorul adresei *p^.next*.

Structura procedurii de traversare este aceeași ca și la structuri lineare în alocare statică, decât că rolul indicelui curent este acum preluat de un pointer curent.

```
procedure Trav_Lista (Start)
p:= Start; {initializarea pointerului curent pentru traversare}
while p<>nil do {test pentru nedepășirea structurii}
    {vizitează nodul p^}
    p:=p^.next {trecem la componenta următoare}
endwhile
endproc{ Trav_Lista}
```

Exemple de prelucrări simple care implică traversări sunt afișarea datelor din noduri la consolă sau aflarea numărului de noduri al listei.

Căutarea într-o listă simplu înlățuită

În lista *Start* se căută o valoare dată *Val*. Mai precis se căută un nod, *Loc*, cu proprietatea *Loc^.info=Val*. Dacă un astfel de nod nu există se returnează *Loc=nil*.

```
procedure Search_List (Start, Val, Loc)
Loc:=Start;
while (Loc<>nil) and (Loc^.info<>Val) do
    Loc:=Loc^.next
endwhile
if Loc<>nil then
    {căutare cu succes}
else
    {Loc=nil} {căutare fără succes}
endif
endproc{Search_List}
```

Evident, variabila *Loc* va trebui să fie transmisă pentru a returna rezultatul căutării și în afara procedurii.

Putem scrie altfel procedura de căutare, punând în evidență traversarea, eventual incompletă, unde "procesarea" este comparația cu *Val*.

```

procedure Search_List_1 (Start, Val, Loc)
p:=Start
while p<>nil do {test de nedepășirea structurii}
    if p^.info=Val then
        Loc:=p
        exit {căutare cu succes cu ieșire din procedură}
    else p:=p^.next
    endif
endwhile
Loc:=nil {căutare fără succes}
endproc{Search_List_1}

```

Analiza complexității căutării pe o listă simplă înlățuită este asemănătoare celei de la structuri lineare în alocare statică. Ea este de ordinul $O(n)$ unde n este numărul de noduri al listei. Abia la listele înlățuite circulare cu nod marcat vom putea reduce la jumătate numărul de comparații eliminând testul de nedepășire al structurii (ceea ce nu ne scoate din această clasă de complexitate).

Inserarea unui nod într-o listă simplu înlățuită

La inserarea unui nou element într-o listă simplu înlățuită va trebui să facem (a) operația de creare a unui nod nou, având pe câmpul *info* să valoarea elementului de inserat, și (b) operația de legare a nodului nou în listă.

(a) Operația de creare a nodului nou este cea care face alocare dinamică de spațiu. Crearea unui nod nou, pentru o listă de întregi, având o valoare dată x întreagă, se face cu secvența de instrucțiuni :

```

new(Nou) {alocarea de spațiu pentru noul nod}
Nou^.info:=x {setarea câmpului info la valoarea dorită}
Nou^.next:=nil {setarea câmpului de legătură la nil}

```

Există posibilitatea depășirii "dimensiunii" și la structurile lineare în alocare dinamică. Dacă programul nu mai are locații libere din care să aloce spațiu pentru variabila dinamică p^{\dagger} , atunci $new(p)$ va returna $p=nil$. În principiu, după invocarea lui $new(p)$, p trebuie testat să nu fie *nil*.

Dacă dorim să facem crearea nodului nou separat și dacă natura problemei o cere (de exemplu dacă inserarea se face obligatoriu) atunci

putem modulariza secvența de creare, de exemplu cu ajutorul unei funcții:

```

function Nou(x:integer):pnod;
new(Nou)
Nou^.info:=x
Nou^.next:=nil
endfunction

```

Funcția *Nou* crează un nod nou cu valoare dată x pe câmpul *info* și returnează pointerul *Nou* către acest nod. Pointerul către nodul nou creat va fi transmis ca parametru unei proceduri de inserare și va fi folosit pentru operația de inserare propriu-zisă. În anumite tipuri de probleme însă este mai practic ca operația de creare să nu se facă separat, ci abia când a sosit momentul inserării în listă.

(b) Operația de inserare propriu-zisă într-o listă simplu înlățuită este operația care "leagă" nodul nou creat de celealte noduri din listă, într-un loc anume în listă, care trebuie determinat în funcție de natura problemei. Odată determinat locul inserării, legarea noului nod la listă se face simplu realocând valorile a doi pointeri (sau schimbând două legături).

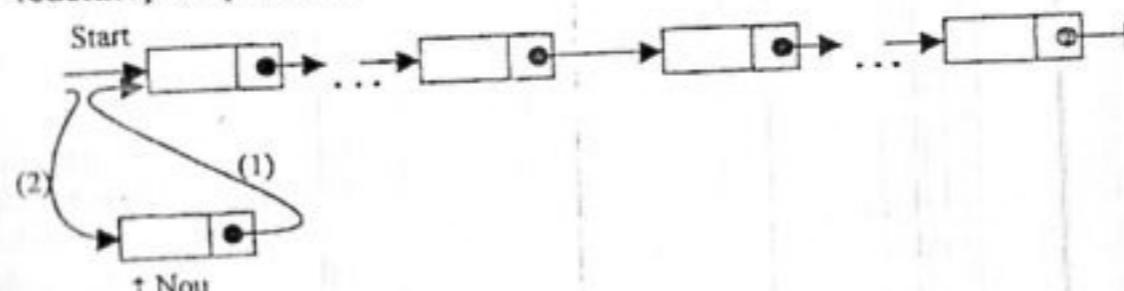
Cazul inserării în capul listei trebuie depistat și tratat separat de către procedura de inserare, deoarece în acest caz se schimbă adresa primului nod al listei. Legarea în capul listei *Start* a unui nod nou, creat deja cu funcția *Nou* ca mai sus, se face cu secvența:

```

Nou^.next:=Start (1)
Start:=Nou (2)

```

După cum se vede și din figură, prima instrucțiune realizează legătura la dreapta (1), iar a doua legătura la stânga (2) – în cazul acesta ea redefineste capul listei.



În general, determinarea locului inserării în listă se face cu o traversare eventual incompletă a listei, cu un pointer curent p .

26

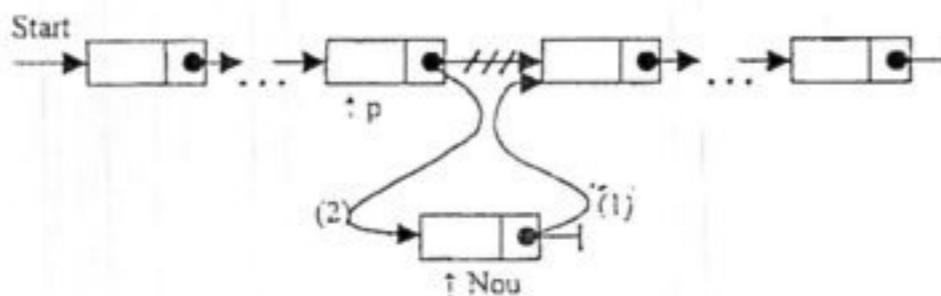
Traversarea va fi guvernată de o condiție de neterminare a structurii și o condiție legată de natura locului de inserat. În funcție de acestă a două condiție, traversarea se oprește în una din următoarele două situații :

- (b1) Nodul pe care s-a oprit p este cel după care urmează să se facă inserarea;
- (b2) Nodul pe care s-a oprit p este cel înaintea căruia trebuie să se facă inserarea.

(b1) Dacă pointerul curent al traversării p se oprește înaintea locului de inserat, atunci inserarea se face cu secvența de instrucțuni

```
Nou↑.next:=p↑.next (1)
p↑.next:=Nou (2)
```

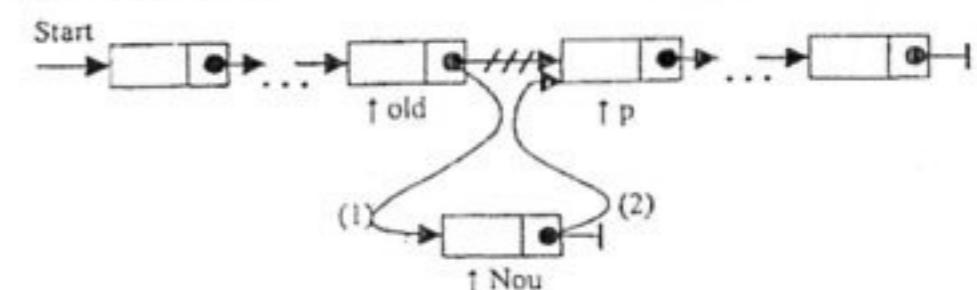
care realizează noile legături la dreapta (1) și la stânga (2), după cum este ilustrat și în figură.



Un exemplu de inserare în care traversarea se poate opri înainte de locul inserării este inserarea unei valori întregi x într-o listă de întregi, pe al k -lea nod al listei, unde k este întreg. Pentru a găsi locul inserării se va face o traversare cu contorizarea nodurilor și ne oprim cu p pe al $(k-1)$ -lea nod al listei (dacă există) inserând noul nod după $p↑$. Dacă lista nu are cel puțin $k-1$ noduri, nu se face inserarea, iar inserarea cu $k=1$ este inserarea în capul listei.

(b2) În cazul în care traversarea se oprește cu pointerul curent p pe nodul înainte de care trebuie să inserăm, dificultatea constă în faptul că, neputând accesa nodul precedent, nu putem seta toate legăturile necesare. Dificultatea acesta se depășește făcând traversarea cu doi pointeri succesivi în loc de unul singur. Dacă p este pointerul care conduce traversarea, fie old un alt pointer, care îl urmează pe p rămnând întotdeauna un pas în urma lui pe nodul precedent. În cazul acesta, la terminarea traversării locul inserării este între nodurile $old↑$ și $p↑$, deci

legarea se face după old ca și cum am fi în cazul (b1), după cum ilustrează și figura :



Următoarea procedură în pseudo-cod presupune nodul creat deja cu funcția Nou și îl inserează înainte de primul nod din listă cu valoare dată Val pe câmpul $info$.

```
procedure Insert1 (Start, Nou, Val)
{inserarea lui Nou↑ înainte de primul p pentru care pt.info=Val}
old:=nil
p:=Start
{traversarea incompletă}
while(p<>nil) and (p.info<>Val) do
    old:= p;
    p:= p↑.next
endwhile
{inserarea, dacă este cazul}
if p = nil then {n-am găsit locația, eventual nu inserez}
else {inserarea între old și p}
    { (1) legătura la stânga}
    if old = nil then {inserăm în capul listei}
        Start:= Nou
    else {inserare după old}
        old↑.next:= Nou
    endif
    { (2) legătura la dreapta }
    Nou↑.next:= p
endif
endproc{Insert1}
```

Aplicație a procedurii de inserare la crearea unei liste simplu înălțuite ordonate

Programul inițializează lista cu lista vidă și conține un ciclu repetitiv care citește câte un întreg de la consolă și îl inseră la locul lui în lista ordonată apelând procedura *InsertListaOrd*.

```
Start:= nil {initializarea listei cu lista vidă }
while not eof do
    read(x)
    InsertListaOrd(Start,x)
endwhile
```

Procedura de inserare într-o listă ordonată este un exemplu clasic în care suntem în cazul (b2).

```
procedure InsertListaOrd(Start, Val)
{ inserarea valorii Val într-o listă de întregi cu câmpurile info ordonate crescător }
{ căutarea locului inserării în listă }
p:= Start;
old:= nil; {initializările pentru parcurgere}
while (p <> nil) and (p^.info <= Val) do
    old:= p
    p:= p^.next
endwhile
{inserarea propriu-zisă}
new(Nou)
Nou^.info:=Val
{legătura la dreapta}
Nou^.next:=p
{legătura la stânga}
if old = nil then {inserare în capul listei}
    Start:= Nou
else {inserare după old}
    old^.next:= Nou
endif
endproc{InsertListaOrd}
```

Procedura acoperă și cazul inserării pe ultimul loc ($p = \text{nil}$) precum și cazul inserării primului nod în lista $Start = \text{nil}$.

Observație: O altă soluție care s-ar putea adopta în cazul (b2) în care traversarea se termină cu p pe un nod înaintea căruia trebuie să inserăm, soluție aplicabilă numai în cazul în care $p < \text{nil}$, ar fi următoarea: alocăm spațiu pentru un nod nou, copiem în el valorile din câmpurile lui p^{\dagger} , valoarea de inserat o punem pe câmpul *info* al lui p , apoi inserăm noul nod după p .

Exerciții

1. La traversarea cu doi pointeri în scopul inserării, am putea face avansul pointerilor cu următoarea secvență:

```
p:=p^.next
old:=old^.next ?
```

Să se justifice răspunsul.

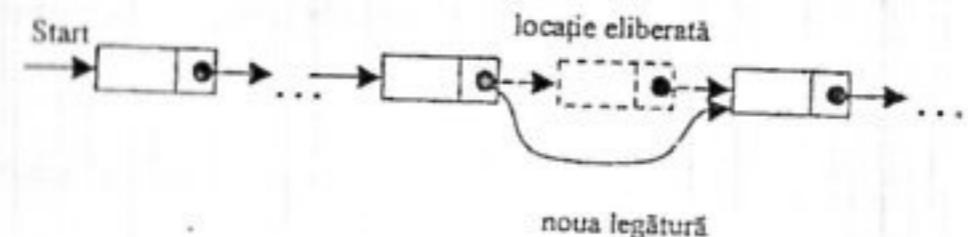
2. Să se scrie în pseudo-cod, apoi în cod, o procedură *Insert(Start, x, k)* care inseră un nod nou cu valoarea x pe câmpul *info* ca al k -lea nod al listei *Start*, dacă este posibil (k este întreg).

3. Să se scrie o versiune recursivă a procedurii de inserare într-o listă ordonată.

4. Să se scrie, în pseudo-cod și cod, o procedură de inserare în care suntem în cazul (b2), dar facem traversarea cu un singur pointer curent aplicând ideea Observației de la sfârșitul secțiunii.

Stergerea unui nod dintr-o listă simplu înălțuită

În cazul stergerii unui nod dintr-o listă simplu înălțuită trebuie să: (a) refacem structura de listă simplă înălțuită pe nodurile rămasă și (b) să ne ocupăm eventual de dealocarea de spațiu pentru nodul sters (spunem "eventual" pentru că, în anumite aplicații, nodul extras dintr-o listă poate fi folosit la alte operații, de exemplu se poate cere inserarea lui în altă listă, deci problema dealocării de spațiu nu se pune).



30

(b) Dealocarea de spațiu se face în felul următor. Fie *temp* o variabilă de tip pointer către nodul de șters din listă. Se refac structura de listă pe nodurile rămase (resetând niște adrese după cum vom vedea în cele ce urmează). Se "exploatează" valoarea extrasă, adică se procesează câmpul *info* al nodului *temp* șters din listă. După ce ne-am asigurat că nu mai avem nevoie să procesăm nici unul din câmpurile lui *temp* se invocă *dispose(temp)*.

Efectul este acela de a returna locațiile de memorie în care am avut valorile din *temp* în lista locațiilor libere.

(a) Ca și în cazul inserării, poziția nodului de șters se găsește în urma unei traversări eventual incomplete a listei cu un pointer curent *p*, traversare guvernată de condiția de nedepășire a structurii și de încă o condiție specifică problemei. Această a doua condiție face ca traversarea să se oprească într-o din următoarele două situații :

(a1) ștergerea se face după nodul *p* cu care am terminat traversarea, adică trebuie șters nodul *p*.*next*.

În cazul acesta traversarea se face cu un singur pointer. Presupunând că nodul ce trebuie șters există efectiv, adică *p*.*next* \neq nil facem :

```
temp:=p.next {salvăm adresa nodului în variabila temp}
p.next:=p.next.next {refacerea structurii de listă a nodurilor rămase}
with temp do
    {utilizăm valorile din nodul extras}
endwith
dispose(temp) {dealocarea de spațiu}
```

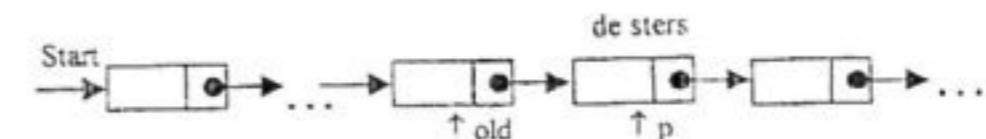
Observăm că refacerea structurii de listă se face cu schimbarea valorii unui singur pointer.

O procedură care face extragerea și transmite adresa nodului extras în afara ei este următoarea:

```
procedure Del_1 (Start,temp)
p:= Start
while (p $\neq$ nil) and{condiția de oprire este false} do
p:=p.next
endwhile
if p $\neq$ nil then
    {traversarea s-a terminat; trebuie șters nodul p}
    {salvăm în temp adresa nodului extras}
    temp:= p
    {refacerea structurii de listă pe nodurile rămase}
    if old = nil then {cazul ștergerii primului nod al listei}
        Start:=p.next
    else
        old.next:= p.next
    endif
endif
endproc { Del_1 }
```

```
{refacerea structurii de listă pe nodurile rămase}
p.next:= p.next.next
endif
end proc {Del_1}
```

(a2) pointerul curent *p* termină traversarea exact pe nodul ce trebuie șters. În cazul acesta vom face, ca și la inserare, traversarea cu doi pointeri curenți succesiivi, *old* cu un pas în urma lui *p*, astfel încât să putem șterge nodul de după *old* plasându-ne în cazul (a1)



O procedură care șterge un nod pe care se oprește traversarea, transmînd adresa nodului extras în afara ei este următoarea:

```
procedure Del_2 (Start,temp)
p:= Start
old:= nil
while (p $\neq$ nil) and{condiția de oprire este false} do
    old:= p
    p:= p.next
endwhile
if p $\neq$ nil then {traversarea s-a terminat; trebuie șters nodul p}
    {salvăm în temp adresa nodului extras}
    temp:= p
    {refacerea structurii de listă pe nodurile rămase}
    if old = nil then {cazul ștergerii primului nod al listei}
        Start:=p.next
    else
        old.next:= p.next
    endif
endif
endproc { Del_2 }
```

Să observăm că trebuie tratat separat cazul ștergerii primului nod al listei și, deoarece variabila *Start* își poate schimba valoarea, ea va trebui să fie transmisă prin adresă. Tot prin adresă trebuie transmisă și variabila *temp* ce conține nodul extras.

Să mai observăm că înainte de a apela o procedură de extragere dintr-o listă trebuie să ne asigurăm că acesta nu este vidă și că deci extragerea are sens.

Observație : În cazul în care trebuie șters nodul p pe care se termină traversarea, și acesta are succesor, adică $p \uparrow .next \neq nil$, atunci se mai poate aplica o metodă care nu implică traversarea cu doi pointeri curenți, și anume: după ce se salvează câmpurile lui $p \uparrow$ de care am avea eventuală nevoie, se mută în ele câmpurile corespunzătoare ale nodului următor, $p \uparrow .next$, apoi se elimină $p \uparrow .next$ din listă.

Exerciții

1. Să se scrie o procedură care extrage valori date dintr-o listă simplu înlățuită ordonată (refăcând structura de listă ordonată pe nodurile rămase).
2. Să se scrie o versiune recursivă a ștergerii unui nod cu valoare dată dintr-o listă simplu înlățuită ordonată.
3. Să se scrie o versiune recursivă a ștergerii unui nod cu valoare dată dintr-o listă simplu înlățuită oarecare.
4. Să se scrie o procedură de ștergere în care suntem în cazul (a2), dar facem traversarea cu un singur pointer și aplicăm ideea Observației de la sfârșitul secțiunii.
5. Să se scrie un program care să compare între ele două liste simplu înlățuite.
6. Să se scrie un program care să realizeze inversarea legăturilor într-o listă simplu înlățuită.
7. Să se scrie un program care să șteargă repetițiile dintr-o listă simplu înlățuită.

1.3. Alte tipuri de liste. Aplicații ale listelor înlățuite.

Liste cu nod marcat

O listă cu nod marcat este o listă simplu înlățuită care conține un nod de tip special, numit nod marcat, ca și prim nod al listei. Câmpul *info* al acestui nod nu conține în general valori, cu anumite excepții după cum vom vedea.

O listă, *Start*, cu nod marcat, va conține în variabila *Start* adresa acestui nod, iar lista efectivă, în care în fiecare nod avem reprezentat un element al mulțimii de date, va fi $Start \uparrow .next$.

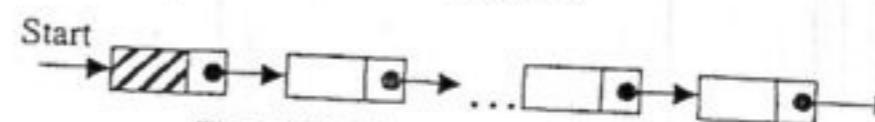


Fig. 1.3.1. Listă cu nod marcat.

O listă cu nod marcat vidă va conține doar nodul marcat.



Fig. 1.3.2. Listă vidă cu nod marcat.

Inițializarea unei asemenea liste cu lista vidă se va face cu secvența:

<code>new(Start)</code>	<code>{alocare de spațiu pentru nodul marcat}</code>
<code>Start \uparrow .next := nil</code>	<code>{inițializarea listei efective cu nil}</code>

Avantajul listelor cu nod marcat față de cele fără marcat apare la operațiile de inserare și ștergere, când cazul inserării în capul listei nu mai trebuie tratat separat în cod, căci variabila *Start* nu își schimbă valoarea. În primul rând, inițializarea celor doi pointeri succesivi pentru parcurgere se va face cu

```
old := Start
p := Start \uparrow .next
```

Observăm că *old* nu mai are valoarea *nil* nici când suntem în cazul inserării în capul listei, deci legătura la stânga la inserare se face (indiferent de locul inserării) cu *old*.*next* := *Nou*.

Liste circulare

O listă simplu înlățuită circulară este o listă simplu înlățuită în care câmpul adresă al ultimului nod, în loc să fie setat la *nil*, este setat la adresa primului nod al listei.

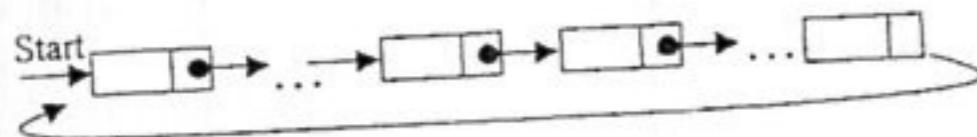


Fig.1.3.3. Lista circulară.

Lista circulară este deosebit de utilă pentru aplicațiile în care este nevoie să facem parcurgeri repetitive ale listei. În loc să tratăm parcurgerile repetitive separat, le putem trata ca pe o singură parcurgere până când atingem scopul propus de aplicație. Evident, pe un asemenea tip de listă, testul de nedepășire al structurii nu va mai fi de tipul $p \neq \text{nil}$.

Liste circulare cu nod marcat

Putem combina avantajele celor două tipuri de modificări prezentate mai sus, lucrând cu liste circulare cu nod marcat.

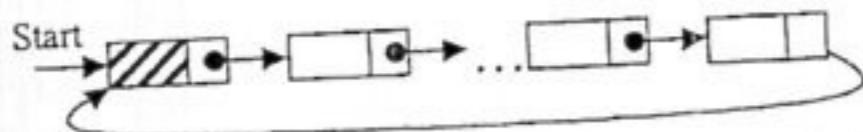


Fig.1.3.4. Lista circulară cu nod marcat.

Lista conține nodul marcat și, în plus, este circulară.
Lista vidă circulară cu nod marcat va conține doar nodul marcat, iar câmpul său *next* va fi setat la *Start*.



Fig.1.3.5. Lista vidă circulară cu nod marcat.

Pe un asemenea tip de listă putem folosi nodul marcat la fel cum foloseam componenta marcat la căutarea lineară pe structurile lineare în alocare statică. Se introduce valoarea căutată *Val* pe câmpul *info* al nodului marcat cu *Start*.*info* := *Val*. Apoi se începe căutarea în lista *Start*.*next*. Cum lista este circulară, în cazul negăsirii lui *Val* în lista efectivă, parcurgerea ajunge pe nodul marcat, unde îl găsește pe *Val*, situație care codifică căutarea fără succes. Fie *Loc* pointerul returnat de operația de căutare pe o asemenea listă. Dacă *Loc* = *Start* căutarea este cu succes, dacă *Loc* ≠ *Start* căutarea este fără succes.

Liste dublu înlățuite

Listele dublu înlățuite sunt liste ale căror noduri au cîmpuri de legătură atât către următorul nod al listei, cât și către precedentul. Presupunem un nod al unei asemenea liste ca o variabilă de tip *nod*, unde avem declarațiile de tip

```
type pnod = ^ nod;
      nod = record
        info: integer;
        next, prev: pnod
      end;
```



Fig.1.3.6. Nod pentru lista dublu înlățuită.

La listele de acest tip, la procedurile de inserare și ștergere, traversarea în scopul găsirii locului se poate face cu un singur pointer *p*, chiar dacă suntem în cazurile în care traversarea se oprește după locul inserării sau chiar pe nodul de sters, căci avem acces la nodul precedent prin intermediul variabilei *p*.*pred*. Un exemplu simplu pentru care acest

tip de listă ar fi indicat este o listă pe care se cer frecvent parcurgeri în ambele sensuri.

Evident, și acest tip de listă se poate combina cu celelalte tipuri obținând astfel liste dublu înăntărită cu nod marcat, circulară, etc.

Când alegem pentru o problemă dată acest tip de listă trebuie să ţinem cont de faptul că adresele către nodurile precedente ocupă și ele spațiu în plus și trebuie să cîntărim dacă avantajele compensează acest dezavantaj.

Exerciții:

- Să se arate care sunt avantajele folosirii unei liste cu nod marcat în cazul operației de ștergere. Să se scrie pseudocodul pentru procedura de ștergere pentru acest tip de listă.
- Dacă vrem să facem o singură parcurgere a unei liste circulare, care este testul de nedepășire a structurii?
- Rescrieți algoritmul de căutare a unei valori date *Val* într-o listă simplu înăntărită pentru o listă circulară cu nod marcat.
- Putem adapta tehnica componentei marcat de la căutarea pe structuri liniare în alocare statică pentru listele simplu înăntărite fără să folosim versiunea circulară a lor?
- Să se scrie procedurile de inserare și de ștergere de nod dintr-o listă dublu înăntărită. Să se aplique la crearea unei liste dublu înăntărite ordonate, precum și la evoluția ei dinamică (se cer de la consolă inserări și ștergeri).

Aplicații ale listelor înăntărite

Dăm în continuare câteva exemple de probleme în care reprezentarea datelor într-o structură de listă înăntărită este indicată, fiind de exemplu mai avantajoasă decât cea de vector.

1) Reprezentarea vectorilor rari

Numim vectori rari acei vectori care au foarte multe componente nule. În general, pentru a implementa operații pe mulțimi de vectori de dimensiune dată (adunarea, înmulțirea a doi vectori componentă cu componentă, etc.) ei sunt reprezentați în alocare statică cu ajutorul tipului *array*. Această reprezentare face ca programele ce implementează operațiile de mai sus să fie extrem de simplu de scris.

Probleme apar când vectorii au dimensiuni foarte mari și, în plus, cea mai mare parte a componentelor lor sunt nule. Pentru un vector de dimensiune 1000, cu numai două valori nenule (pe componente de indice 17 și 912 de exemplu) și cu toate celelalte componente nule, reprezentarea cu o variabilă *A* de tip *array [1..1000] of integer* este ineficientă din punct de vedere al spațiului, căci 998 de locații vor conține valoarea 0. O reprezentare mai eficientă este aceea cu ajutorul unei liste simplu înăntărite, cu doar două noduri, în care să menținem valorile nenule, împreună cu indicii pe care ele apar, organizată ca listă ordonată după cîmpul indicilor.

2) Reprezentarea polinoamelor rare

O problemă asemănătoare este cea a reprezentării polinoamelor rare, polinoame care au foarte mulți coeficienți nuli. În general, polinoamele se reprezintă în alocare statică folosind vectori în care menținem coeficienții.

Ca și la vectorii rari, o reprezentare mai eficientă din punct de vedere al spațiului ar fi sub forma unei liste simplu înăntărite (ordonate după valorile exponentilor) în care menținem în fiecare nod informație despre monoamele cu coeficienți nenuli.

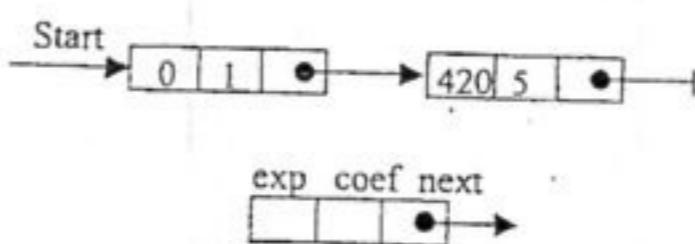


Fig.1.3.7. Reprezentarea polinoamelor rare.
Polinomul $P(X)=5X^{420}+1$

3) Reprezentarea matricilor rare

Matricile rare sunt matricile cu foarte multe componente nule, deci reprezentarea lor în alocare statică conduce la risipă de spațiu. Putem folosi posibilitățile alocării dinamice în felul următor: să menținem fiecare linie într-o listă simplu înăntărită ce conține doar elemente nenule și, analog, pentru coloane. Fiecare element nenul a_{ij} al matricii va fi un nod care face parte din două liste: lista liniei *i* și lista

coloanei j . Pentru întreaga matrice vom avea astfel un set de liste ce mențin liniile și un set de liste ce mențin coloanele. Pentru o matrice de dimensiune 1000×1000 aceasta înseamnă să ocupăm, în loc de 10^6 locații în alocarea statică, doar 2×1000 locații ce conțin pointerii către listele linii și listele coloane, plus locațiile ocupate de componentele nenule. Pentru implementarea mai eficientă a algoritmilor de adunare și înmulțire de matrici este util să facem listele linii și listele coloane liste circulare și cu nod marcat.

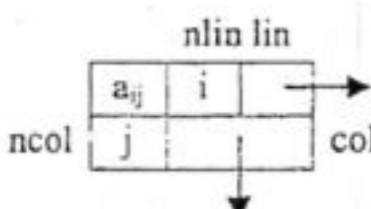


Fig.1.3.8. Reprezentarea matricilor rare. Un nod.

4) Reprezentarea numerelor mari

Numeralele mari sunt numere care depășesc posibilitățile reprezentării în calculator. Pentru a le reprezenta și pentru a face operații pe ele putem folosi structuri de liste simplu înălțuite, reprezentând fiecare cifră (sau câte un grup de cifre) pe câte un nod al listei.

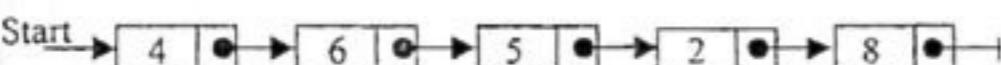


Fig.1.3.9. Reprezentarea numerelor mari.

Întregul 82564 se reprezintă cu lista de mai sus.

Exerciții

1. Să se scrie un program care să facă adunarea și produsul a două polinoame reprezentate ca liste simplu înălțuite.
2. Să se scrie un program care să realizeze adunarea și înmulțirea numerelor mari.
3. Să se scrie un program pentru adunarea, respectiv produsul scalar a doi vectori rari.
4. Să se scrie un program pentru adunarea, respectiv înmulțirea a două matrici rare.

1.4. Structuri lineare cu restricții la intrare/ieșire: stive și cozi

Vom numi structuri lineare cu restricții la intrare și/sau ieșire acele structuri lineare pentru care operațiile de inserare de elemente noi (intrare) și/sau operațiile de extragere de elemente (ieșire) se pot face numai la unul din cele două capete ale structurii. Cele mai importante structuri din această categorie sunt stiva, în care inserările și ștergerile se fac la același capăt, și coada, în care inserările se fac la un capăt iar ștergerile la celălalt.

ACESTE tipuri de structuri sunt caracterizate de operațiile de inserare și ștergere. Nu vom avea operații de traversare sau de căutare, chiar dacă pentru reprezentarea lor folosim liste în alocare statică sau dinamică, iar pe liste în general facem traversări și căutări. Inspectarea unui element dintr-o stivă sau dintr-o coadă se face numai la extragerea respectivului element, iar pentru a putea face acest lucru elementul trebuie "adus" în poziția de unde extragerea este posibilă, eventual prin extragerea altor elemente.

Potem folosi pentru structurile de tip stivă și coadă atât alocare statică cât și dinamică. Vom menține eventual informație în plus asupra locului unde se fac inserările și ștergerile.

Stiva

Stiva este o structură lineară în care inserările și ștergerile se fac doar la un singur capăt numit *vârful* stivei (sau *baza* stivei de către alți autori). Vom folosi pentru acest capăt numele de variabilă *Top*.

Se mai folosește pentru stivă și denumirea de structură de tip LIFO, inițialele de la "Last In First Out" - în traducere "ultimul intrat, primul ieșit" - care arată modul specific de funcționare al acestei structuri: ultimul element introdus va fi și primul extras din stivă.

Pentru operația de inserare într-o stivă s-a incetătenit denumirea *Push*. O procedură *Push(Stack, Val)* inseră valoarea *Val* în stiva *Stack* producând o nouă stivă ce are în vârf valoarea *Val*. Încercarea de inserare într-o stivă plină conduce la supradepășire (Overflow).

Pentru operația de extragere dintr-o stivă s-a incetătenit denumirea *Pop*. *Pop(Stack, X)* extrage din stiva *Stack* o valoare pe care o depune în variabila *X* și produce o stivă care are un element mai puțin. Evident, este inutil să încercăm să extragem elemente dintr-o stivă goală, încercare ce poartă denumirea de subdepășire (Underflow).

Stiva în alocare statică

Vom folosi componentele unui vector pentru a menține o stivă, fie acesta vectorul $Stack[1..Max]$. Convenim să umplem stiva în sensul crescător al indicilor. Atunci stiva este identificată prin vectorul $Stack$ și indicele Top al ultimului element introdus (în același timp, primul ce poate fi extras). Locațiile lui $Stack$ de la $Top+1$ la Max sunt libere, deci putem insera în ele. Vom codifica stiva vidă prin $Top=0$, iar stiva plină prin $Top=Max$.

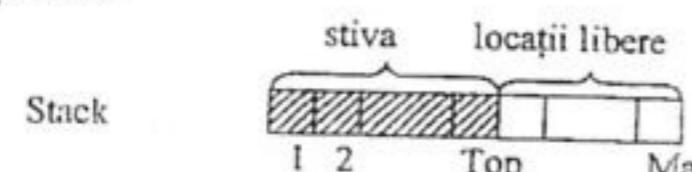


Fig. 1.4.1. Stiva $Stack$ în alocare statică.

O procedură de inserare a valorii Val în stiva dată prin $Stack$ și Top , procedură care face testul de supradepășire în corpul ei, arată în felul următor:

```
procedure Push(Stack, Top, Val)
  if Top=Max then
    Overflow
  else Top:=Top+1
    Stack[Top]:=Val
endproc
```

Analog, avem procedura de extragere în variabila X a unei valori din stivă:

```
procedure Pop(Stack, Top, X)
  if Top=0 then
    Underflow
  else X:=Stack[Top]
    Top:=Top-1
endproc
```

Putem scrie procedurile de inserare și ștergere astfel încât testele de supradepășire, respectiv subdepășire să se facă în afara lor, în procedura apelantă. Dacă testele se fac în interiorul procedurilor, să nu uităm că rezultatul lor trebuie transmis procedurilor apelante.

Stiva în alocare dinamică

În alocare dinamică stiva va fi reprezentată printr-o listă simplu înălțuită Top . Pointerul Top către capul listei reprezintă totalitatea stivei cât și locul pentru inserări și ștergeri. Stiva vidă va fi reprezentată printr-o listă vidă, $Top=nil$.

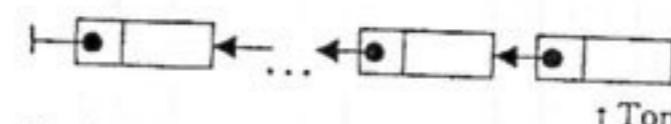


Fig. 1.4.2. Stiva Top în alocare dinamică.

Operațiile de inserare și ștergere se implementează după cum urmează:

```
procedure Push(Top, Val)
  new(Temp)
  if Temp=nil then
    Overflow
  Else Temp.info := Val
    Temp.next := Top
    Top := Temp
endproc
```

```
procedure Pop(Top, X)
  if Top=nil then
    Underflow
  else X:=Top.info
    Temp:=Top
    Top:=Top.next
    dispose(Temp)
endproc.
```

Coada

Coada este o structură lineară în care inserările se fac la un capăt, numit *spatele* sau *sfârșitul cozii*, capăt pe care-l vom numi *Rear*, iar ștergerile la celălalt capăt, numit *față* sau *începutul cozii*, și pentru care vom folosi denumirea de *Front*.

Pentru coadă se mai folosește și denumirea de listă FIFO, de la "First In First Out"- în traducere "primul intrat, primul ieșit" - denumire care arată modul specific de gestionare: clementele sunt extrase și procesate în ordinea intrării.

Și pentru această structură putem avea situațiile de supradepășire - încercarea de a insera un element într-o coadă plină, și subdepășire - încercarea de a extrage un element dintr-o coadă goală.

Coada în alocare statică

În alocare statică o coadă va fi dată printr-un vector *Queue* $[1..Max]$ reprezentând totalitatea spațiului disponibil pentru coadă, împreună cu doi indicii, *Front* pentru începutul cozii, capătul din care se fac ștergeri, și *Rear* pentru sfârșitul cozii, capătul în care se fac inserări.

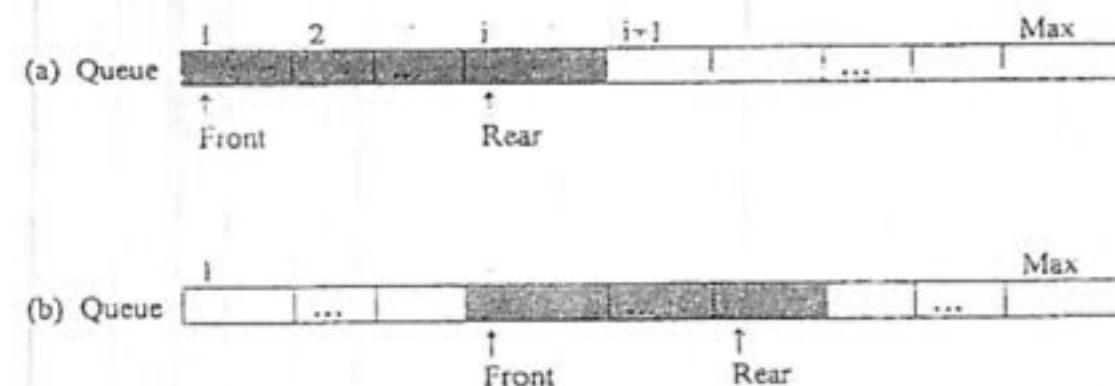


Fig.1.4.3. Coada în alocare statică. În fig. (a) este reprezentată o coadă în care s-au făcut numai inserări (s-au inserat i elemente).

În fig. (b) avem o coadă în care s-au făcut și inserări și ștergeri.

Inserarea unei valori noi *Val* în coada de mai sus se face cu:

Rear:=*Rear*+1

iar ștergerea unei valori cu:

```
X:=Queue[Front]
Front:=Front+1
```

Astfel, prin inserări și ștergeri repetitive, întreg conținutul cozii avansează spre capătul din dreapta al vectorului. Când avem *Rear*=*Max* nu mai putem face inserări. Am putea găsi o metodă de a folosi locațiile din vectorul *Queue* care s-au eliberat în urma ștergerilor, *Queue*[*1..Front-1*]. De exemplu, să considerăm locația *Queue*[1] ca urmând lui *Queue*[*Max*] și să folosim pentru inserare codul:

```
if Rear=Max then
    Rear:=1
else
    Rear:=Rear+1
endif
```

Dar aceasta este același lucru cu a aduna 1 la *Rear* în aritmetică modulo *Max*, adică

Rear:=*Rear* mod *Max*-1

Avem în felul acesta o coadă circulară, ca în figura 1.4.4 în care locația de indice 1 urmează celei de indice *Max* și locațiile ce se eliberează în urma ștergerilor sunt reutilizate pentru inserări.

Coada vidă va fi codificată cu *Front*=*Rear*=0. Coada plină (pe versiunea circulară) va fi codificată cu *Rear*+1=*Front* (mod *Max*).

Un alt caz ce trebuie tratat separat este cel al cozii cu un singur element, când indicii *Rear* și *Front* sunt egali și diferiți de 0, *Rear*=*Front*≠0.

Procedura *Insert*(*Queue*, *Front*, *Rear*, *Val*) inserează valoarea *Val* în coada circulară dată prin vectorul *Queue* și indicii *Front* și *Rear*.

```
procedure Insert(Queue, Front, Rear, Val);
if (Rear mod Max+1=Front) then
    {Overflow}
else {inserare}
    {dacă coada era vidă se modifică și indicele Front}
```

```

if Front=0 then
    Front:=1
endif
Rear:=Rear mod Max+1
Queue[Rear]:=Val
endif
endproc

```

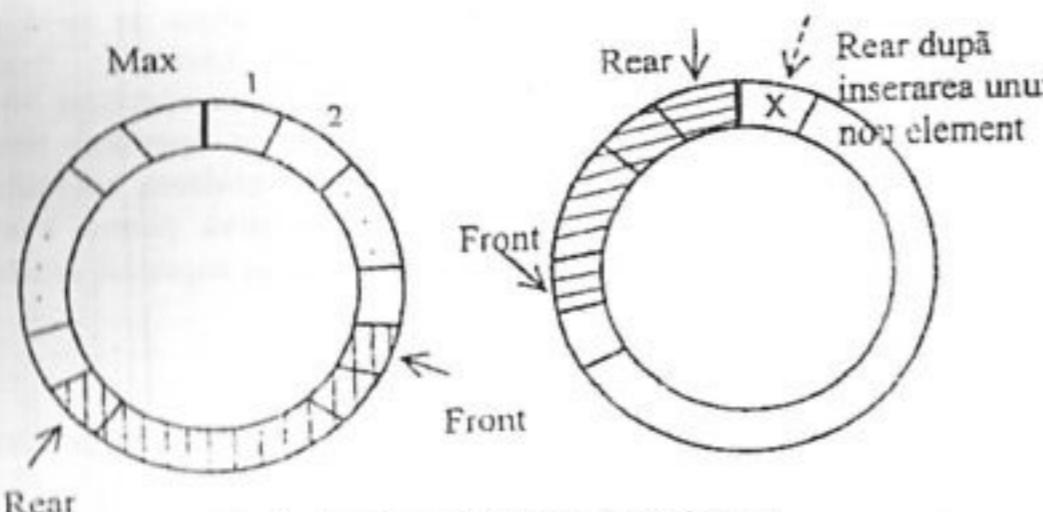


Fig. 1.4.4. Coadă circulară în alocare statică. Inserarea unui nou element.

Următoarea procedură extrage o valoare din coadă în variabila X.

```

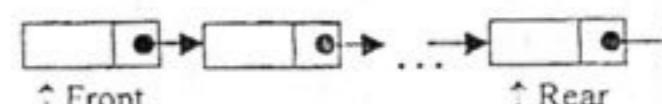
procedure Delete (Queue, Front, Rear, X);
if Front=0 then {Underflow}
else {extragere}
    X:=Queue[Front]
    {refacerea cozii}
    if Front=Rear then
        {dacă coada are un singur element}
        Front:=0;
        Rear:=0;
    else Front:=Front mod Max+1
    endif
endif
endproc

```

Coada în alocare dinamică

În alocare dinamică coada va fi reprezentată printr-o listă simplu înlanțuită dată printr-un pointer *Front* către primul nod al listei, care va fi în același timp capătul din care se fac ștergerile, împreună cu un pointer *Rear* pe care-l vom menține tot timpul pe ultimul nod al listei, acesta fiind capătul în care se fac inserările.

Coadă vidă va fi dată de *Front=Rear=nil*, iar pentru coada cu un singur element vom avea *Front=Rear=nil*.



Procedurile de inserare și ștergere din coadă arată în felul următor:

```

procedure Insert(Front, Rear, Val)
new(p)
if p=nil then
    {Overflow}
else with p↑do
    info:=Val
    next:=nil
endwith
if Rear=nil then {coada era vidă}
    Front:=p
else
    Rear↑.next:=p
endif
Rear:=p
endif
endproc

```

```

procedure Delete (Front, Rear, X);
if Front = nil then
    {Underflow}
else

```

46

{extragerea valonii cu eliberarea spațiului care a fost ocupat de nodul
Front}

```

X:= Front^.info
p:= Front
Front:= Front^.next
dispose(p);
if Front = nil then
    {coada avea un singur element iar acum e vidă}
    Rear:= nil
endif
endif
endproc

```

Cazuri particulare de cozi

Se numește DEQUE (de la *Double Ended Queue*) o structură lineară în care inserările și ștergerile se pot face la oricare din cele două capete, dar în nici un alt loc din coadă.

În anumite tipuri de aplicații sau în modelarea anumitor probleme pot apărea structuri de cozi cu restricții de tipul: inserările se pot face la un singur capăt și extragerile la amândouă.

Un alt tip important de coadă este *coada cu priorități*. Este o coadă în care elementele au, pe lângă cheie și o prioritate. Vom presupune că cea mai înaltă prioritate este 1, urmată de 2, și așa mai departe. Ordinea lineară este dată de regulile:

- elementele cu aceeași prioritate sunt extrase (și procesate) în ordinea intrării;
- toate elementele cu prioritate i se află înaintea celor cu prioritate $i+1$ (și deci vor fi extrase înaintea lor).

Extragerile se fac dintr-un singur capăt. Ca să se poată aplica regulile de mai sus la extragere, inserarea unui nou element cu prioritate i se va face la sfârșitul listei ce conține toate elementele cu prioritate i .

Coada cu priorități, în reprezentarea ei ca listă, poate fi concepută ca fiind compusă din mai multe cozi concatenate: coada elementelor cu prioritate 1, urmată de cea a elementelor cu prioritate 2, etc. Deși extragerile se fac în timp $O(1)$, inserările se fac în timp $O(n)$. Abia în capitolul 3 secțiunea 5 vom vedea o reprezentare arborescentă a cozilor cu priorități, în care inserările se fac în timp $O(\log n)$.

Aplicații ale structurii de stivă și coadă

În primul rând aceste tipuri de structuri sunt folosite de sistemele de operare. Reamintim doar structura de stivă pe care o implică utilizarea recursivității, iar diverse tipuri de cozi sunt utilizate de sistemele multi-tasking, multi-user, etc.

Aplicațiile la programare sunt extrem de numeroase. Cea mai simplă aplicație a structurii de stivă este inversarea ordinii unui set de elemente date. O clasă întreagă de aplicații folosesc structura de stivă pentru evaluarea corectitudinii unor expresii, de exemplu: evaluarea corectitudinii parantetizării expresiilor aritmetice cu operatori binari în notație infix; evaluarea expresiilor aritmetice cu operatori binari în notație postfix (scriere poloneză inversă), cu validarea corectitudinii. Analizoarele sintactice folosesc structura de stivă pentru evaluarea corectitudinii codului. Alte aplicații vom vedea în capitolul următor, la traversarea structurilor neliniare.

Exerciții

1. Să se scrie un program care inversează ordinea unui sir de caractere citite de la o consolă, folosind o stivă.
2. Să se scrie un program care citește de la consolă o expresie aritmetică cu operatori binari în notație infix, cu paranteze, și care evaluatează corectitudinea parantetizării. (Programul utilizează o stivă și face evaluarea pe măsura ce citește expresia, deci cu o singură parcurgere a datelor de intrare)
3. Să se scrie un program care are ca date de intrare un sir de caractere ce reprezintă o expresie aritmetică cu operatori binari (+, *, /) și operanți întregi, în notație postfix. Folosind structura de stivă, cu o singură parcurgere a expresiei, să se evaluateze expresia (sau să se decida incorectitudinea ei).
4. Să se scrie proceduri de extragere și inserare de elementele într-o coadă cu priorități reprezentată ca listă simplu înlățuită.
5. Se dau trei stive: S_{In} pentru datele de intrare, care conține în ordine crescătoare întregii $\{1, 2, \dots, n\}$, S_{Out} pentru ieșire și S_{Aux} o stivă auxiliară.
 - a) Pentru $n=4$, puteți să imaginați un mod de utilizare al stivelor astfel încât dacă în S_{In} avem configurația 1234 în S_{Out} să obținem configurația 3412? Dar configurația 1324?
 - b) Putem să folosim cele trei stive pentru a genera pe rând în S_{Out} toate permutările de n elemente, având în S_{In} configurația 1234?

Capitolul II

Structuri arborescente

Structurile arborescente sunt cel mai simplu tip de structuri neliniare. Începem în acest capitol un studiu al lor, care nu se vrea și nici nu poate fi exhaustiv în cadrul restrâns al acestei lucrări.

Prezentăm la început noțiunea de arbore oarecare, sau k -arbore, discutăm despre posibilitatea reprezentării lor cu diverse tipuri de legături, introducem pe scurt câteva tipuri particulare de arbori ce vor fi folosiți ulterior și prezentăm terminologia uzuală pentru lucru cu arbori.

Apoi ne concentrăm asupra unei clase speciale importante de arbori, arborii binari. Prezentăm pentru aceștia, în detaliu, tipurile speciale de traversări. Dăm o caracterizare matematică precisă noțiunilor de similaritate și echivalență, noțiuni care acoperă ideile intuitive de "formă" a unui arbore binar, modelând asemănarea și egalitatea pe mulțimea arborilor binari.

O clasă particulară de arbori binari este prezentată în secțiunea 4, arborii binari de căutare. După cum arată și numele, aceste structuri sunt optime (în anumite cazuri) pentru operația de căutare. Prezentăm algoritmii specifici de inserare și ștergere, construcția prin inserări repetitive și, evident, operația de căutare, împreună cu analiza complexității lor.

Pentru a aduce performanța operației de căutare de la $O(n)$ cât era la structurile lineare la $O(\log_2 n)$ prezentăm în ultima secțiune noțiunea de arbore binar de căutare echilibrat AVL.

2.1. Arbori

O primă definiție a structurii de arbore o putem da pornind de la o structură mai complexă, cea de graf.

Definiție. Se numește *graf* $G = (X, V)$ o pereche formată din două mulțimi, mulțimea X a nodurilor sau vîrfurilor grafului, și mulțimea V a muchiilor grafului, unde o muchie $v \in V$ este o pereche ordonată de noduri $v = (x, y)$, $x, y \in X$.

Un graf neorientat este un graf în care perechea (x, y) se identifică cu perechea (y, x) . Un graf fără cicluri este un graf în care, pornind de la un vîrf dat nu putem ajunge din nou la el folosind muchii.

Definiție. Se numește *arbore* un graf $H = (X, V)$ care este neorientat, conex, fără cicluri, cu un nod precizat numit rădăcină. Pentru orice vîrf $x \in X$, există un număr finit de vîrfuri $x_1, \dots, x_n \in X$ asociate lui x , numite descendenți direcți (sau fi) lui x .

Putem da și o definiție recursivă a structurii de arbore: O structură de arbore, T , de un anume tip de bază este

- (1) fie o structură vidă (adică $T = \emptyset$);
- (2) fie este nevidă, deci conține un nod de tipul de bază, pe care-l vom numi rădăcină și îl vom nota $\text{root}(T)$, plus un număr finit de structuri disjuncte de arbori de același tip, T_1, T_2, \dots, T_k , numiți subarborii lui T (sau fi ai $\text{root}(T)$).

Cuvintele "tip de bază" se referă la informație de un anume tip reținută în nod (eticheta nodului) și la numărul maxim de descendenți ai fiecărui nod. Dacă numărul maxim de descendenți ai fiecărui nod este un întreg k , atunci spunem că avem un k -arbore.

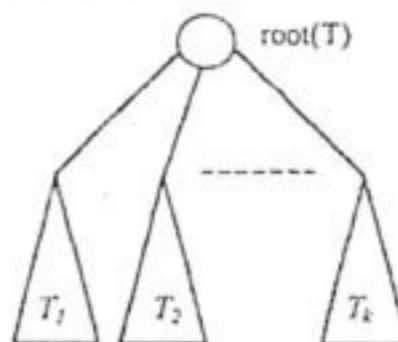


Fig.2.1.1. Un k -arbore T , cu nodul rădăcină $\text{root}(T)$ și fi săi, T_1, T_2, \dots, T_k .

Reamintim definiția recursivă a unei liste: O listă L de un anume tip de bază este:

- (1) fie lista vidă ($L = \emptyset$);
- (2) fie este nevidă, și atunci conține un nod numit capul listei, urmat de o altă listă de același tip de bază.

Observăm că listele apar ca și cazuri particulare de arbori, în care fiecare nod are un singur descendenter și pot fi considerate arbori degenerați.

Pentru reprezentarea arborilor, cea mai frecventă este reprezentarea ca graf, în care relația tată - fiu dintre un nod și descendenții lui direcți este reprezentată ca muchie sau arc.

Se mai poate folosi și reprezentarea ca expresie parantetizată, fiecare vârf fiind urmat de lista fiilor săi între paranteze.

O structură arborescentă poate fi folosită pentru a codifica o multitudine de relații ierarhice; dăm ca exemplu doar relația de incluziune între mulțimi: dacă X este inclus în Y atunci reprezentăm X ca fiu al lui Y . Universul ce cuprinde toate mulțimile va fi rădăcina arborelui.

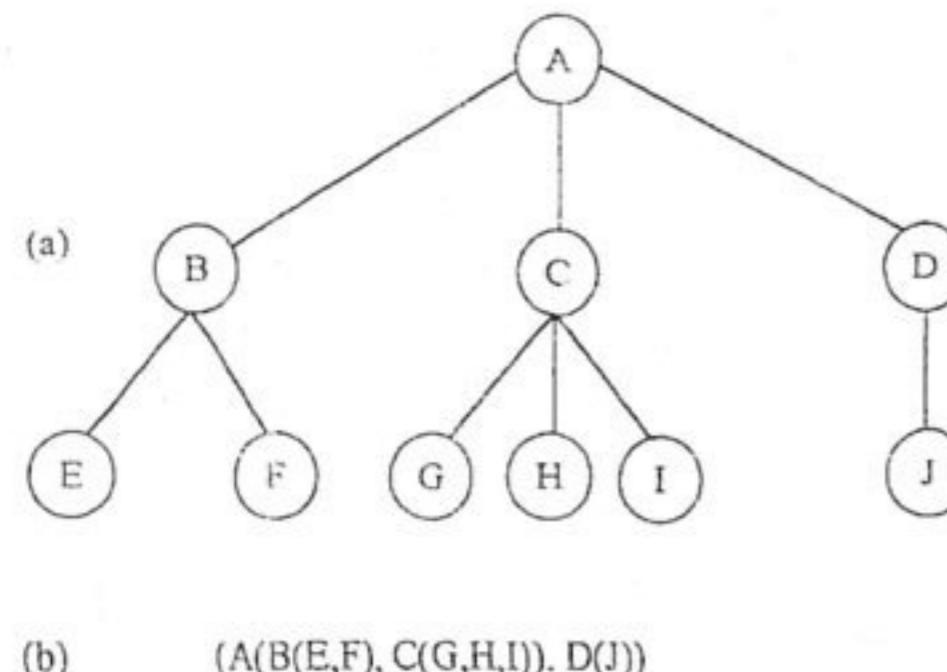


Fig.2.1.2.(a) Un 3-arbore reprezentat ca graf.

(b) Același 3-arbore reprezentat ca listă.

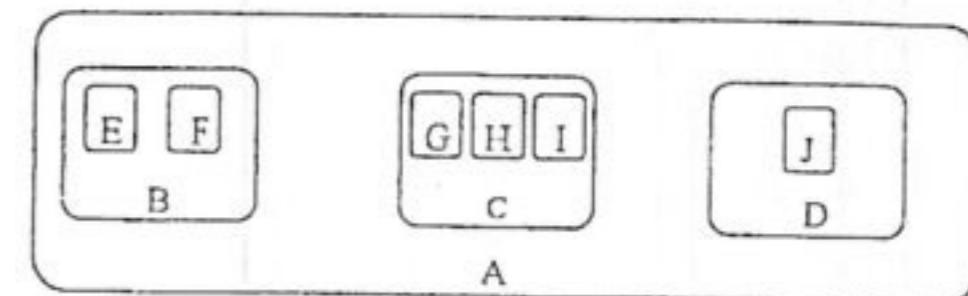


Fig.2.1.3. Relații de incluziune modelate cu 3-arbori

Vom face distincție între *arborii ordonați*, arbori în care există o relație de ordine între descendenții fiecărui nod, și cei *neordonatați*, în care descendenții formează o mulțime neordonată.

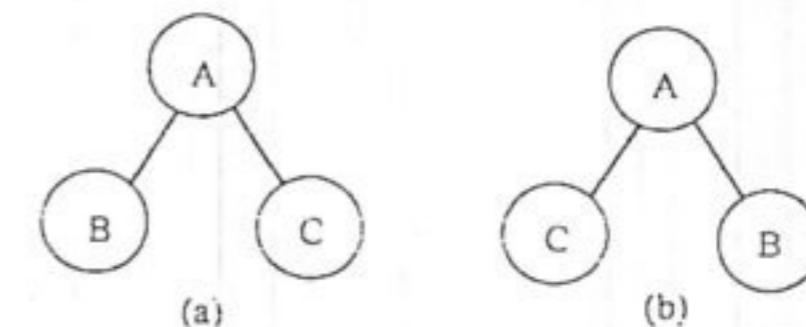


Fig.2.1.4. Arborii (a) și (b) sunt identici ca arbori ne-ordonatați, dar distinți ca arbori ordonați.

Terminologia pe care o vom folosi este următoarea:

Vom numi gradul arborelui, întregul k care reprezintă numărul maxim de fiu ai unui nod.

Fiecare nod al arborelui îi vom asocia un *nivel* în felul următor:

(a) rădăcina se află la nivelul 0

(b) dacă un nod se află la nivelul i atunci fiile săi sunt la nivelul $i+1$

Numim *înălțime* (sau *adâncime*) a unui arbore nivelul maxim al nodurilor sale.

Se numește *terminal* sau *frunză* un nod fără descendenți.

Se numește *nod interior* orice nod care nu e terminal.

Un 2-arbore ordonat se numește *arbore binar*. Arborii binari ocupă un loc important în clasa structurilor arborescente. Pentru cei doi fii ai unui arbore binar s-a întărit denumirea de fiu stâng, respectiv fiu drept.

Tipuri de legături pentru reprezentarea arborilor

Tipurile de legături folosite pentru reprezentarea arborilor depind de mai mulți factori: ce tip de mulțime de date reprezintă, ce arbore, ce relații trebuie puse în evidență, ce operații sunt efectuate mai frecvent, etc.

Reprezentarea unui arbore cu legătură *tată - fiu*, însemnă, în principiu, că din fiecare nod, avem acces la oricare dintre fii. Ea este potrivită pentru arbori ordonați și pentru care accesul la noduri se face "de sus în jos".

Cu legături de tip *tată* (fiecare nod "știe" cine este tatăl său) se pot reprezenta arbori neordonati. Este un tip de legătură frecvent pentru probleme în care nodurile arborelui reprezintă elementele unei mulțimi și suntem interesați să facem operații pe mulțimi: teste de apartenență și reuniiuni de mulțimi.

Un alt tip de reprezentare este cu tip de legătură *fiu - frațe*. Aceasta înseamnă că pentru fiecare nod al arborelui avem acces la primul său fiu și la frații săi. (Nodurile de pe același nivel se numesc frați și îi organizăm ca listă.)

Putem trece în mod canonnic de la un k -arbore T (reprezentat cu legătură *tată - fii*) la un arbore binar care se va numi *arbore binar canonico asociat lui T* transformând "primul fiu" în fiu stâng și legătura la frații în fiu drept.

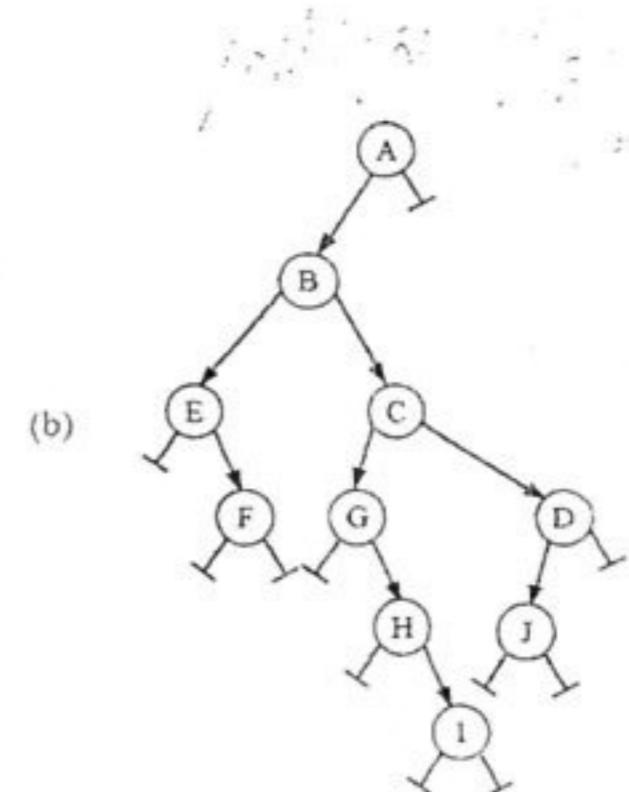
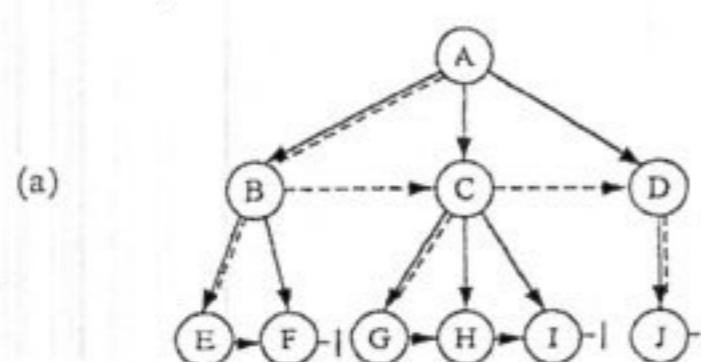


Fig 2.1.5. Figura (b) reprezintă arborele binar asociat canonico 3-arborelui din figura (a). S-au folosit legăturile primul fiu și frațe (reprezentate prin săgeți punctate în figura (a)). Legătura "primul fiu" din arborele inițial devine legătură de tip fiu stâng, iar legătura "frațe" devine legătură de tip "fiu drept" în arborele binar asociat canonico.

Prin trecerea la arborele binar canonico asociat putem transfera probleme de k -arbori la arbori binari.

Arbori compleți pe niveluri

În multe probleme ne interesează să avem înălțime minimă pentru un k -arbore pentru număr total de noduri n fixat. Un tip de arbore care îndeplinește acest deziderat este *arborele complet pe niveluri* - un k -arbore care are toate nivelurile pline, ceea ce cu excepția ultimului nivel, unde nodurile apar cel mai în stânga.

Înălțimea unui asemenea arbore este egală cu $\lceil \log_k n \rceil$.

Un 2-arbore ordonat se numește *arbore binar*. Arborii binari ocupă un loc important în clasa structurilor arborescente. Pentru cei doi fii ai unui arbore binar s-a întărit denumirea de fiu stâng, respectiv fiu drept.

Tipuri de legături pentru reprezentarea arborilor

Tipurile de legături folosite pentru reprezentarea arborilor depind de mai mulți factori: ce tip de mulțime de date reprezintă, ce arbore, ce relații trebuie puse în evidență, ce operații sunt efectuate mai frecvent, etc.

Reprezentarea unui arbore cu legătură *tată - fiu*, însemnă, în principiu, că din fiecare nod, avem acces la oricare dintre fii. Ea este potrivită pentru arbori ordonați și pentru care accesul la noduri se face "de sus în jos".

Cu legături de tip *tată* (fiecare nod "știe" cine este tatăl său) se pot reprezenta arbori neordonați. Este un tip de legătură frecvent pentru probleme în care nodurile arborelui reprezintă elementele unei mulțimi și suntem interesați să facem operații pe mulțimi: teste de apartenență și reuniiuni de mulțimi.

Un alt tip de reprezentare este cu tip de legătură *fiu - frate*. Aceasta înseamnă că pentru fiecare nod al arborelui avem acces la primul său fiu și la frații săi. (Nodurile de pe același nivel se numesc frați și îi organizăm ca listă.)

Putem trece în mod canonico de la un k -arbore T (reprezentat cu legătură *tată - fiu*) la un arbore binar care se va numi *arbore binar canonico asociat lui T* transformând "primul fiu" în fiu stâng și legătura la frați în fiu drept.

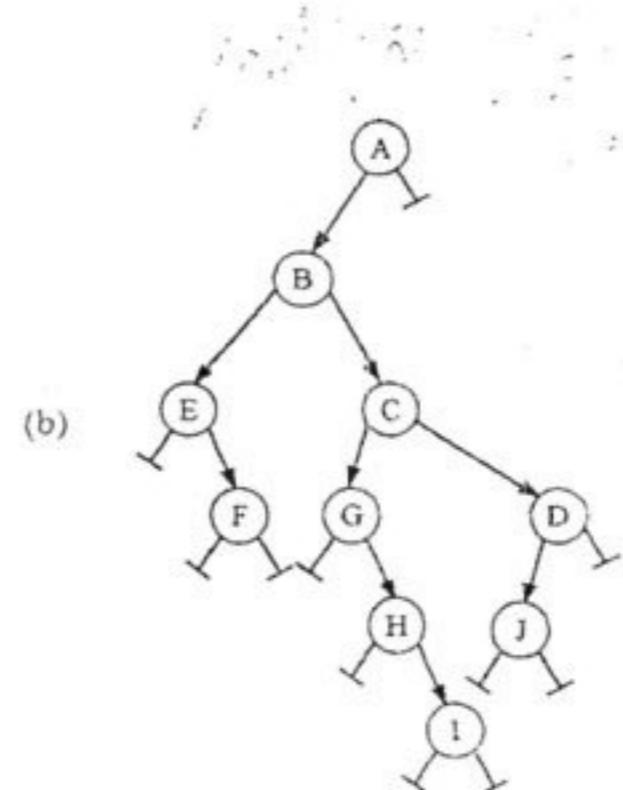
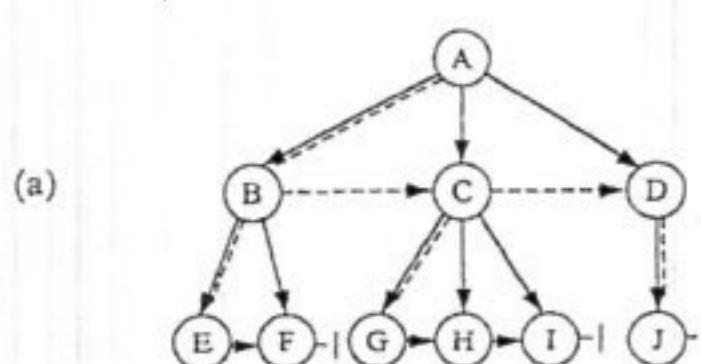


Fig 2.1.5. Figura (b) reprezintă arborele binar asociat canonico 3-arborelui din figura (a). S-au folosit legăturile primul fiu și frate (reprezentate prin săgeți punctate în figura (a)). Legătura "primul fiu" din arborele inițial devine legătură de tip fiu stâng, iar legătura "frate" devine legătură de tip "fiu drept" în arborele binar asociat canonico.

Prin trecerea la arborele binar canonico asociat putem transfera probleme de k -arbori la arbori binari.

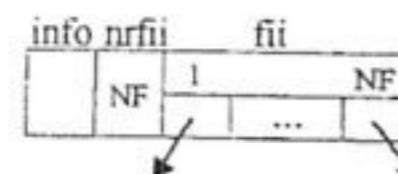
Arbori compleți pe niveluri

În multe probleme ne interesează să avem înălțime minimă pentru un k -arbore pentru număr total de noduri n fixat. Un tip de arbore care îndeplinește acest deziderat este *arborele complet pe niveluri* - un k -arbore care are toate nivelurile pline, eventual cu excepția ultimului nivel, unde nodurile apar cel mai în stânga.

Înălțimea unui asemenea arbore este egală cu $\lceil \log_k n \rceil$.

O reprezentare în alocare dinamică a unui arbore oarecare

Pentru o reprezentare în alocare dinamică a unui arbore, în care suntem interesați să facem parcurgeri de la rădăcină în jos, am putea folosi următorul tip de nod:



Fiecare nod are trei câmpuri:

info: pentru informația din noduri

nrffii: valoarea *NF*, întreagă, reprezintă numărul de fiu ai nodului

fii: vector de dimensiune *NF*, cu componente pointeri, astfel încât, *fii[J]* este pointer către fiul *J* al nodului, iar *J* = 1, 2, ..., *NF*.

Traversarea unui arbore oarecare

Traversarea este accesarea sau vizitarea fiecărui nod al structurii arborescente, o singură dată.

Algoritmul de traversare are următoarea structură:

1. Se pornește de la rădăcină.

2. La fiecare nod curent:

(a) se procesează *info*

(b) se introduc într-o structură ajutătoare fiile nodului curent în vederea procesării ulterioare.

3. Se extrage din structura ajutătoare un alt nod și se reia de la punctul 2. Ca structură ajutătoare putem folosi una din structurile lineare pe care le cunoaștem, stivă sau coadă.

Dacă folosim o coadă obținem *traversarea în lățime* (*breadth first*) a arborelui.

Dacă folosim o stivă obținem *traversarea în adâncime* (*depth first*) a arborelui.

Procedurile de traversare au următoarea structură:

procedure *TravLat*(Root)

{traversarea în lățime a arborelui Root, folosind o coadă Queue și procedurile *QINSERT* (Queue, ·) pentru inserare, respectiv *QDELETE*(Queue, ·) pentru ștergere din coadă}

Queue $\leftarrow \emptyset$ {initializarea cozii}

if Root \neq nil then

QINSERT(Queue, Root)

endif

while Queue $\neq \emptyset$ do

QDELETE(Queue, NodCrt)

 {procesează NodCrt.info}

 for K := 1 to NodCrt.info.nrffii do

QINSERT(Queue, NodCrt.info.fii[K])

 endfor

endwhile

endproc

procedure *TravAd*(Root)

{traversarea în adâncime a unui arbore, Root, folosind o stivă STACK, cu procedurile *PUSH* (STACK, ·) și *POP* (STACK, ·) pentru introducere, respectiv extragere din stivă}

STACK $\leftarrow \emptyset$ {initializarea cozii}

if Root \neq nil then

PUSH(STACK, Root)

endif

while STACK $\neq \emptyset$ do

POP(STACK, NodCrt)

 {procesează NodCrt.info}

 for K := NodCrt.info.nrffii down to 1 do

PUSH(STACK, NodCrt.info.fii[K])

 endfor

endwhile

endproc

Exerciții

1. Să se scrie un program care are ca dată de intrare un *k*-arbore și să se creeze arborele binar canonic asociat lui.

2. Să se scrie un program care are ca date de intrare un arbore binar considerat canonic asociat unui *k*-arbore și care:

(a) il determină pe *k*

(b) construiește *k*-arborele.

3. Să se implementeze operațiile de traversare ale unui *k*-arbore (*k* fixat).

58

2.2. Arbori binari

Definiție. Un *arbore binar* este un 2-arbore ordonat. Pentru cei doi fiți ai unui arbore binar s-a încreșterea de denumirea de fiu stâng, respectiv fiu drept.

Definiția recursivă a arborelui binar: Un arbore binar T este:

(1) fie un arbore vid ($T = \emptyset$).

(2) fie e nevid, și atunci conține un nod numit rădăcină, împreună cu doi subarbore binari disjuncți numiți subarborele stâng, respectiv subarborele drept.

Potem folosi structura de arbore binar pentru a reprezenta de exemplu:

1. arbori genealogici (pedigree): fiecare nod va reprezenta o persoană, iar fiul stâng - tatăl, fiul drept - mama respectiv persoane;
2. turnee: fiecare nod reprezintă cîștigătorul unui meci, iar cei doi fiți reprezintă participanții la respectivul meci;
3. expresii aritmetice cu operatori binari.

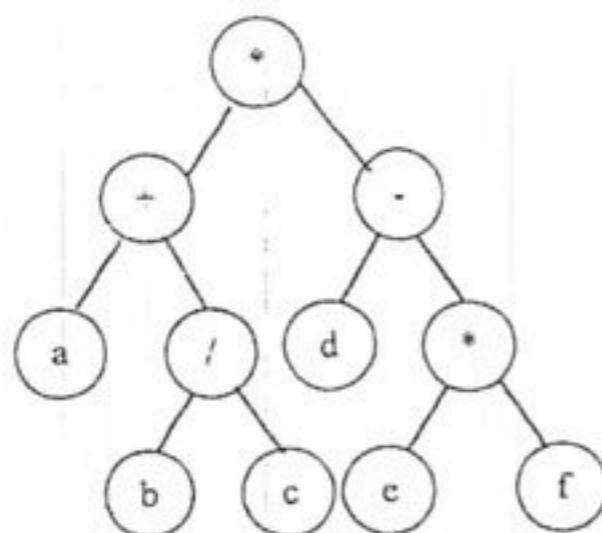
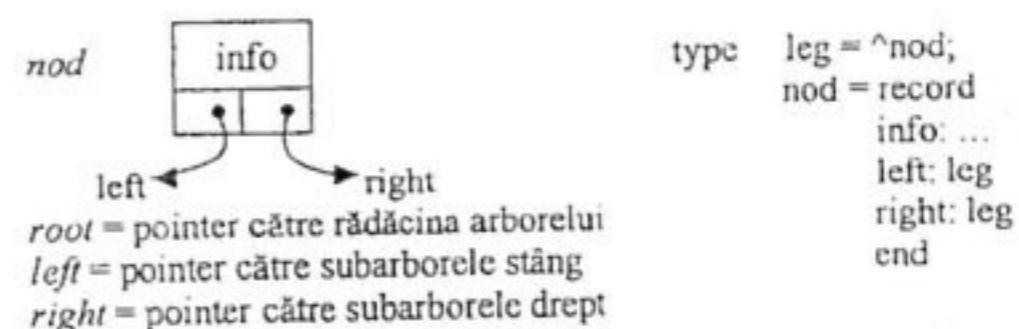


Fig. 2.2.1. Arborele binar ce reprezintă expresia aritmetică $(a + (b / c)) * (d - (e * f))$.

O reprezentare dinamică a arborilor binari în Pascal o putem face cu următoarele tipuri de date:



Un subarbore vid e marcat prin valoarea *nil* a pointerului *root* către rădăcina lui.

Traversări de arbori binari

Pe lângă traversarea în lățime (care se face folosind o coadă) există trei tipuri specifice de traversări în adâncime ale arborilor binari:

- 1) în Preordine (RSD) (Rădăcină Stânga Dreapta)
- 2) în Inordine (SRD)
- 3) în Postordine (SDR)

Prezentăm întâi versiunile iterative ale acestor traversări. Toate trei folosesc un pointer curent *p*, cu care se începe traversarea de la rădăcină (*p* := *Root*), și o stivă pentru a ține minte nodurile ce rămân de procesat.

Procedure PreOrdine(Root){RSD}

```

Stack ← Ø {initializarea stivei Stack cu stiva Ø}
p := Root {initializare pointerului curent}
while p ≠ nil do
  (A) repeat
    1) {procesează p.info}
    2) if p.right ≠ nil then
        PUSH (Stack, p.right)
      endif
    3) p := p.left
    until p = nil
  (B) if Stack ≠ Ø then
    POP (Stack, p)
    endif
  endwhile
endproc
  
```

- (A) Pentru fiecare nod se repetă pașii 1-3:
- 1) se procesează nodul;
 - 2) dacă are fiu drept nevid, se introduce fiul drept în stivă;
 - 3) se merge un pas la stânga și se reia de la 1).
- (B) Când nu mai pot merge la stânga
- 4) se extrage din stivă un nod și se reia de la (A).

Procedure InOrdine(Root); {SRD}

```

Stack ← Ø
p := Root
do
  while p ≠ nil do
    {introduc fiecare nod în stivă și mă duc la stânga, câte un pas
     până nu mai am unde}
    PUSH (Stack, p)
    p := p↑.left
  endwhile
  repeat
    if Stack ≠ Ø then
      {extrag câte un nod din stivă și-l procesez}
      POP(Stack, p)
      {procesează p↑.info}
    else return {ieșire din do..repeat}
    endif
    until p↑.right ≠ nil {până la primul care are fiu drept}
    p := p↑.right {mă duc un pas la dreapta și reiau ciclul do..repeat}
  repeat
endproc

```

procedure PostOrdine(Root) {SDR}

```

Stack ← Ø
p := Root
do
  (1) while p ≠ nil and p↑.left ≠ nil do
    PUSH (Stack, p)
    p := p↑.left
  endwhile
  (2) while p↑.right = nil do
    repeat
      {procesează p↑.info}

```

```

if Stack ≠ Ø then
  Old := p
  POP (Stack, p)
else return {ieșire din do..repeat}
endif
  until Old = p↑.left
endwhile
(3) PUSH (Stack, p)
(4) p := p↑.right
repeat
endproc

```

La *PostOrdine* apare o dificultate în plus față de celelalte două tipuri: când extrag un nod din stivă, n-am voie să-l procesez decât dacă am terminat procesarea fiilor, stâng și drept, dacă există. Pentru asta am nevoie de încă un pointer curent, *Old*, spre nodul procesat anterior, care este fie fiul stâng (*Old* = *p*↑.left), fie fiul drept (*Old* = *p*↑.right) al nodului curent *p*.

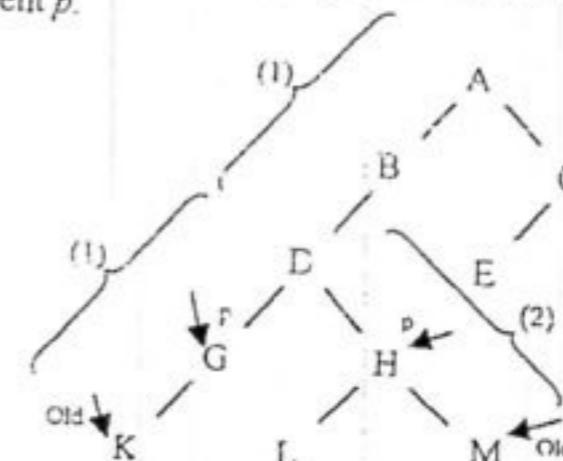
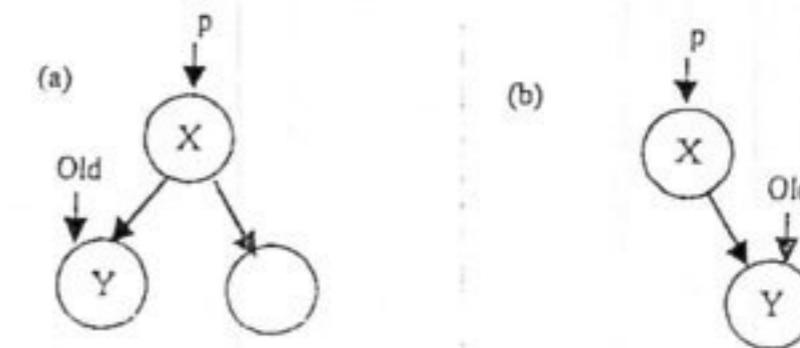


Fig.2.2.2. Traversarea în post ordine.

Cele două situații posibile sunt:



62

- (a) Dacă *Old* îmi arată că vin de pe o ramură stângă, atunci, dacă nodul curent are fiu drept, îl introducem la loc în stivă (3), mergem un pas la dreapta (4), și se reia ciclul *do .. repeat*.
- (b) Dacă *Old* îmi arată că vin de pe o ramură dreaptă, atunci trebuie să procesăm nodul *p* extras din stivă (se reia *repeat ... until* de la (2)) Ciclul (2) constă din două structuri repetitive una în alta, *while* controlat de *p*[†].right = nil și *repeat ... until* *Old* = *p*[†].left.
- Dacă sunt în situația (a) (vin din stânga) *Old* = *p*[†].left este adevărat, deci se ieșe din *repeat* și controlul se transferă lui *while*. Deci, pe ramuri de tip (1) (vezi Fig. 2.2.2.) procesez fiecare nod extras din stivă pentru că fiul lui stâng e procesat iar fiu drept nu are. Dacă sunt în situația (b) (vin din dreapta) *Old* = *p*[†].left este falsă, deci se reia secvența din interiorul lui *repeat*. Pe ramuri de tip (2) (vezi Fig. 2.2.2.) procesez fiecare nod extras din stivă pentru că tocmai am terminat procesarea fiului drept.

Procedurile recursive de traversare a arborilor binari au o formă foarte simplă și le prezentăm în continuare.

```
procedure Preord (p: leg);
begin
  if p < nil then
    begin
      Procesează (p)
      Preord (p†.left)
      Preord (p†.right)
    end
  end
```

```
procedure Inord (p: leg);
begin
  if p < nil then
    begin
      Inord (p†.left);
      Procesează (p);
      Inord (p†.right)
    end
  end
```

```
procedure Postord (p: leg);
begin
  if p < nil then
    begin
      Postord (p†.left);
      Postord (p†.right);
      Procesează (p)
    end
  end;
```

Arborei binari compleți pe niveluri

Un arbore binar complet pe niveluri este un arbore binar care are toate nivelurile pline, eventual cu excepția ultimului nivel, unde nodurile apar cel mai în stânga.

Înălțimea unui asemenea arbore este egală cu $\lceil \log_2 n \rceil$.

Pe lângă faptul că are înălțime minimă, avantajul acestui tip de arbore este că el se poate reprezenta neechivoc în alocare statică cu ajutorul unui vector ce menține cheile în ordinea dată de traversarea în lățime.

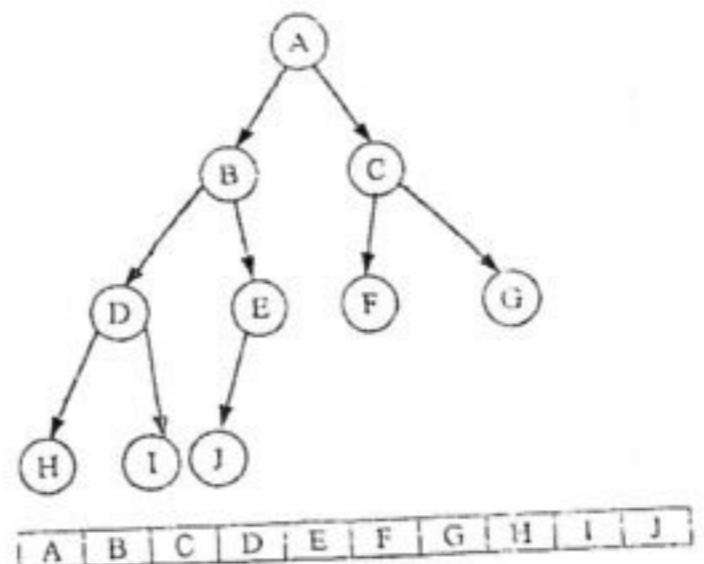


Fig. 2.2.3. Arbore binar complet pe niveluri, reprezentat ca vector.

Pe această reprezentare statică obținem ușor legăturile fiu stâng, fiu drept și tată cu formulele:

- A[k div 2] este tatăl lui A[k]
- A[2k] este fiul stâng a lui A[k]
- A[2k+1] este fiul drept a lui A[k].

Arborei binari însăilați

Subarborii vizibili ai unui arbore binar (legăturile de tip *nil*) pot fi utilizati pentru a facilita efectuarea unor operații frecvente pe arbori. Dacă de exemplu vrem să facilităm operația de traversare în inordine, am putea face *însăilarea în inordine* a unui arbore binar, în felul următor: toți pointerii *nil* se reseteză către nodurile

- succesoare în inordine, dacă sunt legături de tip fiu drept,
- predecesoare în inordine, dacă sunt legături de tip fiu stâng.

Va trebui să menținem în fiecare nod un câmp marcat în plus, care să ne spună dacă legătura stângă și/sau dreaptă sunt legături în arbore sau legături de însăilare. Însăilarea revine practic la suprapunerea unei structuri de listă simplu înlănțuită peste structura de arbore binar. Analog se poate face însăilarea în preordine.

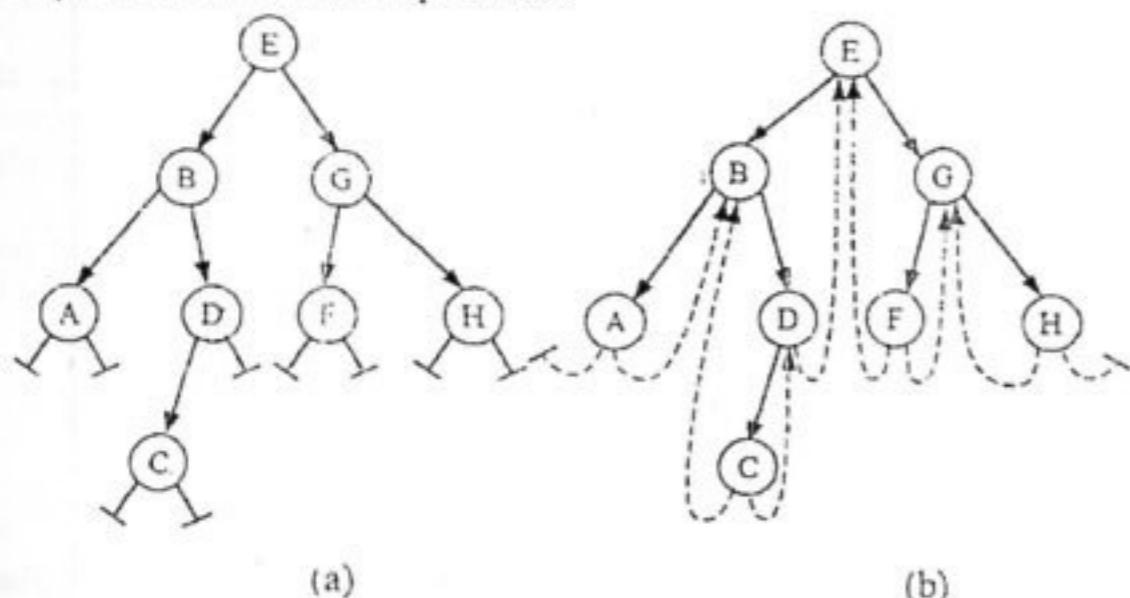


Fig. 2.2.4. Arbore binar însăilit în inordine (b): Pointerii care aveau valoarea *nil* în (a) au fost setați în felul următor: legăturile de tip *left* (fiu stâng) către nodul predecesor în inordine, iar legăturile de tip *right* (fiu drept) către nodul succesor în inordine.

Exerciții

1. După modelul de la arborele binar, să se scrie cum se exprimă legăturile fiu stâng, fiu drept și tată pentru un 3-arbore complet pe niveluri. Dar pentru un *k*-arbore?
2. Scrieți un program care primește ca date de intrare un arbore binar și produce arborele binar însăilit în inordine. Scrieți apoi o procedură care utilizează legăturile de însăilare la traversare.
3. Scrieți o procedură recursivă care afișează un arbore binar la consolă, pe niveluri.
4. Scrieți o procedură care numără frunzele unui arbore binar.
5. Transformați în programe Pascal procedurile iterative cu cele trei tipuri de traversări în adâncime ale unui arbore binar.
6. Comparați din punct de vedere al *spațiului* utilizat versiunile iterative cu cele recursive ale algoritmilor de traversare în adâncime ale unui arbore binar.

2.3. Arbori binari – Similaritate și echivalență

Ne ocupăm în acest paragraf cu formularea unor definiții precise și cu caracterizarea matematică a unor noțiuni intuitive pentru arbori binari, precum noțiunea ca doi arbori să aibă “aceeași formă” (similaritatea) și noțiunea să aibă “aceeași formă și același conținut” (echivalența).

Informal, doi arbori binari T și T' se numesc *similari* dacă “au aceeași formă”. Mai precis, putem spune că există o bijecție între nodurile lui T și cele lui T' astfel încât dacă nodurile u_1 și u_2 aparținând lui T le corespund nodurile u'_1 și u'_2 aparținând lui T' , atunci dacă u_1 aparține lui $stg(u_1)$ vom avea și u'_1 aparține lui $stg(u'_1)$, iar dacă dacă u_2 aparține lui $dr(u_2)$ vom avea și u'_2 aparține lui $dr(u'_2)$. Cu alte cuvinte, relațiile tată-fiu dintre perechi de noduri din T se păstrează prin bijecție în arborele T' .

Doi arbori binari sunt *echivalenți* dacă sunt similari și, în plus, nodurile ce se corespund prin bijecție conțin aceleași informații.

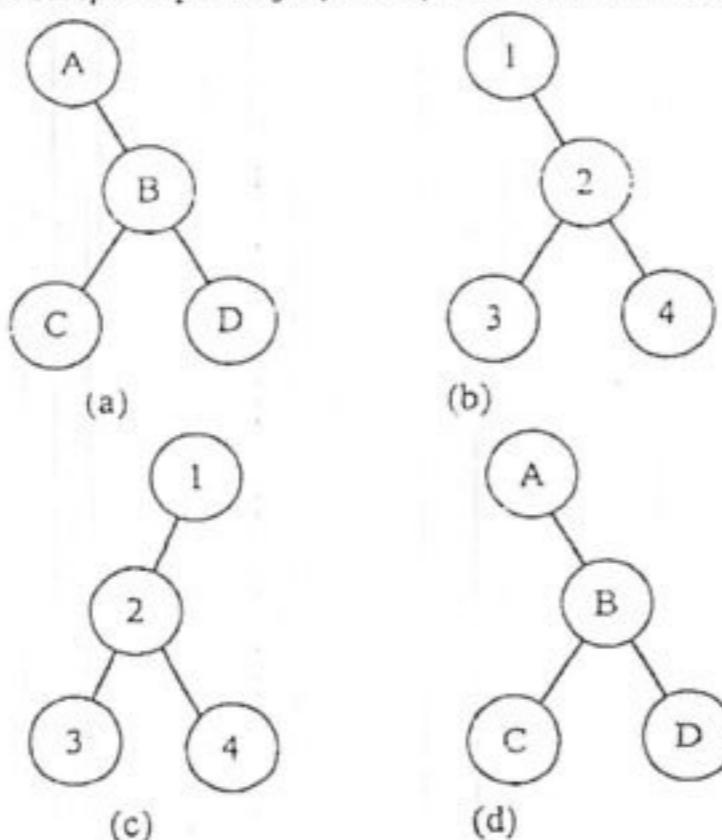


Fig.2.3.1. Arborii (a) și (b) sunt similari, (b) și (d) sunt similari, (b) este diferit de (c), (a) și (d) sunt echivalenți.

Putem să dăm alte definiții ale relațiilor de similaritate și echivalență, bazate pe definiția recursivă a arborelui binar, pe care o reamintim:

Definiție T se numește *arbore binar* dacă:

(a) fie T este vid ($T = \emptyset$)

(b) fie T este nevid ($T \neq \emptyset$) și deci conține un nod numit rădăcină, notat $root(T)$, împreună cu doi arbori binari $stg(T)$ și $dr(T)$, numiți fiul stâng, respectiv fiul drept al lui $root(T)$.

Pe baza ei putem defini formal similaritatea și echivalența după cum urmează.

Definiție Doi arbori binari T și T' se numesc *similari*, și notăm aceasta cu $T \approx T'$, dacă:

(a) fie sunt ambii vizi, adică $T = \emptyset$ și $T' = \emptyset$;

(b) fie sunt ambii nevizi, adică $T \neq \emptyset$ și $T' \neq \emptyset$ și atunci avem $stg(T) \approx stg(T')$ (fiul stâng ai rădăcinilor sunt similari) și $dr(T) \approx dr(T')$ (fiul drept ai rădăcinilor sunt similari)

Definiție Doi arbori binari T și T' se numesc *echivalenți*, și notăm aceasta cu $T \equiv T'$, dacă:

(a) fie sunt ambii vizi, adică $T = \emptyset$ și $T' = \emptyset$;

(b) fie sunt ambii nevizi, adică $T \neq \emptyset$ și $T' \neq \emptyset$ și atunci avem $info(root(T)) = info(root(T'))$ și $stg(T) \equiv stg(T')$ și $dr(T) \equiv dr(T')$

Fie $T = u_1 \ u_2 \ \dots \ u_n$ un arbore binar dat prin lista nodurilor sale la parcurgerea în preordine (RSD). Fiecare nod u_i îi vom asocia trei valori: $info(u_i)$ reprezentând informația din nodul u_i și $l(u_i), r(u_i)$ valorile funcțiilor l și r în nodul u_i .

Funcțiile l (de la left) și r (de la right) sunt funcții marcate pentru existența fiilor stângi, respectiv drepti, definite într-un nod u după cum urmează:

$$l(u) := \begin{cases} 1, & \text{dacă } u \text{ are fiu stâng nevid (adică } stg(u) \neq \emptyset) \\ 0, & \text{dacă } u \text{ nu are fiu stâng nevid (adică } stg(u) = \emptyset) \end{cases}$$

Analog,

$$r(u) := \begin{cases} 1, & \text{dacă } dr(u) \neq \emptyset \\ 0, & \text{dacă } dr(u) = \emptyset \end{cases}$$

Atunci arborele $T = u_1 \ u_2 \ \dots \ u_n$ va fi caracterizat prin valorile celor trei vectori cu n componente, vectorii $info$, l și r (ultimii doi având ca valori posibile doar pe 0 și 1).

info	1	2	...	i	...	n
l				info(u _i)		
r				l(u _i)		
r				r(u _i)		

Ex : (a)	Ex : (b)																											
<table border="1"> <tr> <th>info</th><th>A</th><th>B</th><th>C</th><th>D</th></tr> <tr> <th>l</th><td>0</td><td>1</td><td>0</td><td>0</td></tr> <tr> <th>r</th><td>1</td><td>1</td><td>0</td><td>0</td></tr> </table>	info	A	B	C	D	l	0	1	0	0	r	1	1	0	0	<table border="1"> <tr> <td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr> <td>0</td><td>1</td><td>0</td><td>0</td></tr> <tr> <td>1</td><td>1</td><td>0</td><td>0</td></tr> </table>	1	2	3	4	0	1	0	0	1	1	0	0
info	A	B	C	D																								
l	0	1	0	0																								
r	1	1	0	0																								
1	2	3	4																									
0	1	0	0																									
1	1	0	0																									

Fig.2.3.2. Valorile vectorilor $info$, l și r pentru arborii (a) și (b) din Fig.2.3.1.

Să definim tot pe nodurile unui arbore binar funcția f prin $f(u) := l(u) + r(u) - 1$.

Să observăm că

$$f(u) = \begin{cases} 1, & \text{dacă } l(u) = r(u) = 1, \text{ adică } u \text{ are doi fiu} \\ 0, & \text{dacă } \begin{cases} l(u) = 1 \text{ și } r(u) = 0 \\ l(u) = 0 \text{ și } r(u) = 1 \end{cases} \text{ sau} \\ & \text{adică } u \text{ are un singur fiu} \\ -1, & \text{dacă } l(u) = r(u) = 0, \text{ adică } u \text{ nu are fiu} \end{cases}$$

Funcția f definită mai sus, asociată cu parcurgerea în preordine a unui arbore binar ne dă extremitate de multă informație despre arbore după cum vom vedea din rezultatul care urmează.

Lemă Fie $T = u_1, u_2, \dots, u_n$ un arbore binar parcurs în preordine.

Atunci următoarele relații sunt adevărate:

(i) $f(u_1) + \dots + f(u_k) \geq 0$, pentru orice $k \in [1, n]$

(ii) $f(u_1) + \dots + f(u_n) = -1$

(iii) Dacă $f(u_1) = 1$ atunci există $\min\{k / f(u_1) + \dots + f(u_k) = 0\} = n_1 + 1$, unde n_1 este numărul de noduri din fiul stâng al lui T .

Demonstrație O vom face prin inducție după n .

$n=1$. T are un singur nod $T = u_1$ deci $l(u_1) = r(u_1) = 0$, deci $f(u_1) = -1$ (din cele trei relații în acest caz nu rămâne decât relația (ii) care se reduce la $f(u_1) = -1$ pe care tocmai am demonstrat-o adevărată).

Fie $n > 1$. Ipoteza de inducție este că presupunem lema adevărată pentru orice arbore binar cu m noduri, unde $m < n$.

Fie acum un n oarecare și $T = u_1, u_2, \dots, u_n$ un arbore binar parcurs în preordine cu n noduri. Avem doar două valori posibile pentru $f(u_1)$, $f(u_1) \in \{0, 1\}$ (căci dacă $f(u_1) = -1$ atunci rezultă $n=1$, ceea ce contrazice ipoteza $n > 1$).

Să analizăm pe rând cele două cazuri:

(a) $f(u_1) = 0$. Rezultă că u_1 rădăcina lui T are un singur fiu nevid deci avem

- fie (a1) : $stg(T) = \emptyset$ și $dr(T) \neq \emptyset$ ceea ce este echivalent cu $l(u_1) = 0$ și $r(u_1) = 1$
- fie (a2) : $stg(T) \neq \emptyset$ și $dr(T) = \emptyset$ ceea ce este echivalent cu $l(u_1) = 1$ și $r(u_1) = 0$

Raționamentul este același pentru oricare din cazurile (a1) sau (a2).

Presupunem că suntem în cazul (a1), adică singurul fiu nevid al lui T este cel drept. Atunci $dr(T) = u_2, u_3, \dots, u_n$ reprezintă parcurgerea lui în preordine, $dr(T)$ are $n-1$ noduri și îi putem aplica ipoteza de inducție obținând relațiile (i) și (ii) pentru $dr(T)$:

- (i) $f(u_2) + \dots + f(u_n) \geq 0$, pentru orice $k \in [2, n]$
- (ii) $f(u_2) + \dots + f(u_n) = -1$

La aceste relații putem aduna $f(u_1)$ care are valoarea 0 și adăugăm și relația $f(u_1) = 0$ obținând:

relația (i) pentru T :

$$f(u_1) = 0 \\ f(u_1) + f(u_2) + \dots + f(u_n) = f(u_2) + \dots + f(u_n) \geq 0, \text{ pentru orice } k \in [2, n] \\ \text{relația (ii) pentru } T :$$

$$f(u_1) + f(u_2) + \dots + f(u_n) = f(u_2) + \dots + f(u_n) = -1.$$

și am încheiat demonstrația în cazul (a).

(b) $f(u_1) = 1$. Rezultă că T are ambii fiu nevizi. Dacă n_1 este numărul de noduri din fiul stâng al lui T , atunci nodurile de la u_2 la u_{n_1+1} sunt nodurile lui $stg(T)$ parcurs în preordine, iar de la u_{n_1+2} la u_n sunt nodurile lui $dr(T)$ parcurs în preordine. Cu alte cuvinte, din parcurgerea în preordine a lui T obținem parcurgerile în preordine ale fiilor săi : $stg(T) = u_2, \dots, u_{n_1+1}$ și $dr(T) = u_{n_1+2}, \dots, u_n$.

Subarborilor $stg(T)$ și $dr(T)$ le putem aplica ipoteza de inducție.

Să calculăm acum sumele parțiale din enunțul lemei pentru arborele T și să le evaluăm.

$$k=1 \quad f(u_1) = 1 > 0$$

$$k \in [2, n] \quad f(u_1) + f(u_2) + \dots + f(u_k) = 1 + f(u_2) + \dots + f(u_k) > 0$$

pentru că $f(u_2) + \dots + f(u_k) \geq 0$ din relația (i) pentru $stg(T)$

$$k = n_l + 1 \quad f(u_1) + f(u_2) + \dots + f(u_{n_l+1}) = 1 + f(u_2) + \dots + f(u_{n_l+1}) = 1 - 1 = 0$$

pentru că $f(u_2) + \dots + f(u_{n_l+1}) = -1$ din relația (ii) pentru $stg(T)$.

(cu acesta am demonstrat relația (iii) pentru T).

$$k \in [n_l + 2, n-1]$$

$$f(u_1) + f(u_2) + \dots + f(u_{n_l+1}) + f(u_{n_l+2}) + \dots + f(u_k) = f(u_{n_l+2}) + \dots + f(u_k) \geq 0$$

pentru că este relație de tip (i) pentru $dr(T)$

(cu aceasta am încheiat demonstrarea relațiilor (i) pentru T).

$$k = n \quad f(u_1) + f(u_2) + \dots + f(u_{n_l+1}) + f(u_{n_l+2}) + \dots + f(u_n) = f(u_{n_l+2}) + \dots + f(u_n) = -1$$

pentru că este relația (ii) pentru $dr(T)$

(cu acesta am demonstrat și relația (ii) pentru T).
q.e.d.

Teorema de caracterizare a similarității și echivalenței arborilor binari

Teoremă Fie $T = u_1, u_2, \dots, u_n$ și $T' = u'_1, u'_2, \dots, u'_{n'}$ doi arbori binari dați în parcurgerile lor în preordine. Următoarele afirmații sunt adevărate:

(a) T și T' sunt similari ($T \approx T'$) dacă și numai dacă avem relațiile

$$n = n'$$

(S) $I(u_i) = I(u'_i)$ pentru orice i aparținând intervalului $[1, n]$
 $r(u_i) = r(u'_i)$ pentru orice i aparținând intervalului $[1, n]$

(b) T și T' sunt echivalenți ($T \equiv T'$) dacă și numai dacă avem relațiile

$$n = n'$$

(E) $I(u_i) = I(u'_i)$ pentru orice i aparținând intervalului $[1, n]$
 $r(u_i) = r(u'_i)$ pentru orice i aparținând intervalului $[1, n]$
 $info(u_i) = info(u'_i)$ pentru orice i aparținând intervalului $[1, n]$

Demonstrație Este evident că e suficient să demonstrăm punctul (a), (b) urmând imediat din definiția relației de echivalență.

Vom face demonstrația prin inducție după n numărul de noduri ale lui T .

$n = 0$ Dacă T și T' sunt similari, cum $T = \emptyset$ vom avea și $T' = \emptyset$, $n = n'$, deci $n = n'$ (singura relație din setul (S) care este de demonstrat în acest caz).

Pentru implicația în sens contrar, din $n = 0$ și $n = n'$ rezultă $n' = 0$, deci atât T cât și T' sunt vizi, deci similari (primul caz al definiției).

$n > 0$ Ipoteza de inducție: presupunem, pentru arbori cu m noduri, $m < n$, că, dacă sunt similari, atunci avem relațiile (S) pentru nodurile lor.

Fie acum T și T' similari, T cu n noduri $n > 0$. Aceasta înseamnă că $T \neq \emptyset$, deci și $T' \neq \emptyset$, și în plus avem:

$$(b1) \ stg(T) \approx stg(T')$$

$$(b2) \ dr(T) \approx dr(T')$$

Fie n_l și n_r numărul de noduri din $stg(T)$ respectiv $dr(T)$, și, analog n'_l și n'_r , numărul de noduri din $stg(T')$ respectiv $dr(T')$. Avem:

$$n = 1 + n_l + n_r \text{ și } n' = 1 + n'_l + n'_r.$$

Din relația de similaritate (b1), aplicând ipoteza de inducție (căci $stg(T)$ are mai puțin de n noduri) obținem relațiile:

$$n_l = n'_l$$

$$I(u_i) = I(u'_i)$$

pentru orice i aparținând intervalului $[2, n_l - 1]$

$$r(u_i) = r(u'_i)$$

pentru orice i aparținând intervalului $[2, n_l - 1]$

Din relația de similaritate (b2), aplicând ipoteza de inducție obținem relațiile:

$$n_r = n'_r$$

$$I(u_i) = I(u'_i)$$

pentru orice i aparținând intervalului $[n_l + 2, n]$

$$r(u_i) = r(u'_i)$$

pentru orice i aparținând intervalului $[n_l + 2, n]$

Punând cap la cap aceste două seturi de relații obținem pe de o parte $n = n'$, pe de altă parte toate egalitățile dorite între funcțiile marcaj, mai puțin cele pentru primul nod u_1 .

Să demonstrăm că $I(u_1) = I(u'_1)$. Dacă $I(u_1) = 0$, atunci $n_l = 0$, deci $n'_l = 0$, deci $I(u'_1) = 0$, și avem egalitatea dorită. Dacă $I(u_1) = 1$ atunci $n_l \neq 0$, dar $n'_l = n_l$, deci $n'_l \neq 0$, deci $I(u'_1) = 1$, și din nou avem egalitatea dorită. Analog se demonstrează că $r(u_1) = r(u'_1)$, ceea ce încheie demonstrația unei implicații.

Pentru implicația în sens invers să presupunem că ipoteză de inducție că, dacă relațiile de tip (S) sunt adevărate atunci arborii T și T' sunt similari, unde T are m noduri, iar $m < n$.

Să presupunem acum că avem arborii T și T' ca în enunț, T cu n noduri, și relațiile (S) pentru ei sunt adevărate.

Este suficient să demonstrăm că avem $n_l = n'_l$. Într-adevăr dacă această relație e adevărată, atunci rezultă ușor din $n = n'$ că $n_r = n'_r$.

Apoi, egalitățile funcțiilor marcaj se împart în două grupe: pentru indicii $i \in [2, n_l + 1]$, împreună cu $n_l = n'_l$, conduc, conform ipotezei de inducție la $stg(T) \approx stg(T')$; egalitățile pentru indicii $i \in [n_l + 2, n]$, împreună cu $n_r = n'_r$, conduc, tot prin aplicarea ipotezei de inducție la concluzia $dr(T) \approx dr(T')$. De unde, aplicând definiția recursivă a similarității rezultă $T \approx T'$.

Să demonstrăm egalitatea $n_l = n'_l$. Avem trei cazuri:

- (1) $l(u_l) = 0$. Atunci și $l(u'_l) = 0$, iar $n_l = 0$ și $n'_l = 0$, de unde rezultă $n_l = n'_l$;
- (2) $l(u_l) = l$ și $r(u_l) = 0$. Dar atunci și $r(u'_l) = 0$, deci $n_l = 0$ și $n'_l = 0$, de unde $n_l = n'_l$, deci $n_l = n'_l$.
- (3) $l(u_l) = l$ și $r(u_l) = l$. Deci $f(u_l) = l$. Conform relației (iii) din Lemă există $\min\{k | f(u_1) + \dots + f(u_k) = 0\} = n_l + 1$.

Pe de altă parte pentru u'_l , avem $l(u'_l) = l$ și $r(u'_l) = l$, deci $f(u'_l) = l$, deci conform relației (iii) din Lemă există $\min\{k | f(u'_1) + \dots + f(u'_{l+1}) = 0\} = n'_l + 1$. Cum funcțiile marcaj l și r sunt egale pe nodurile corespunzătoare din T și T' vom avea $f(u_i) = f(u'_i)$ pentru orice $i \in [1, n]$, deci sumele parțiale pe cei doi arbori coincid, deci va coincide și indicele pe care se anulează prima oară, adică vom avea $n_l + 1 = n'_l + 1$, de unde $n_l = n'_l$. Q.E.D.

Teorema ne spune că informația completă despre structura unui arbore binar este cuprinsă în vectorii $info[1..n]$, $l[1..n]$ și $r[1..n]$ menționați la începutul secțiunii. Cu alte cuvinte, din valorile acestor vectori se poate reconstrui arborele. Mai mult, avem chiar o "metodă" de construcție, care se poate transforma ușor în algoritm: calculăm $f(u)$ pentru fiecare nod, apoi sumele parțiale pe rând, la prima care se anulează am aflat ce noduri conține subarborele stâng, etc.

Dacă am avea doar sirul câmpurilor $info$ de la parcurgerea arborelui binar în preordine, informația nu ar fi suficientă pentru a reconstituî structura arborescentă. Lucrurile se schimbă însă dacă am avea și lista câmpurilor $info$ ale parcurgerii în inordine, cu condiția ca arborele să nu conțină chei multiple.

Exerciții

1. Scrieți un program care are ca date de intrare vectorii $info$, l și r și construiește arborele binar corespunzător.
2. Scrieți un program care are ca date de intrare un arbore binar și construiește vectorii $info$, l și r .
3. Demonstrați că din parcurgerile în preordine și inordine ale unui arbore binar se poate reconstituî structura arborescentă.
4. Scrieți un program care are ca date de intrare parcurgerile în preordine și inordine ale unui arbore binar și construiește arborele binar corespunzător.

2.4. Arbori binari de căutare

Arborii binari sunt frecvent folosiți la reprezentarea unei mulțimi de date ale cărei elemente sunt identificate printr-o cheie unică. Dacă această cheie ia valori de un tip total ordonat, de exemplu întreg sau caracter, este posibil să "îmbogățim" structura arborescentă cu relații de ordine între chei.

Dacă am reprezenta aceleași date printr-o listă, atunci o "îmbogățire" naturală a structurii de listă bazată pe ordinea dintre chei ar fi lista ordonată. După cum am văzut, pe o listă ordonată se îmbunătățește performanța operației de căutare a unei chei. Ea va fi în cazul mediu de două ori mai mare pe o listă ordonată decât pe o listă neordonată.

Ne punem întrebarea dacă în cazul arborilor, în particular în cazul arborilor binari, există un analog al listei ordonate. Un tip particular de arbore binar, numit arbore binar de căutare, este o structură concepută special pentru a îmbunătății operația de căutare a unei chei.

Definiție. Un arbore binar ale căruia chei iau valori de un tip total ordonat se numește *arbore binar de căutare* dacă cheia fiecărui nod este mai mare decât orice cheie din fiul său stâng și mai mică decât orice cheie din fiul său drept. Formal, într-un arbore binar de căutare, pentru orice nod u al său avem relațiile:

- (1) $info[u] > info[v]$, pentru orice $v \in left[u]$
- (2) $info[u] < info[w]$, pentru orice $w \in right[u]$.

Să observăm că ar fi suficient să impunem existența acestor relații de ordine între un nod și descendenții săi direcți. Cu alte cuvinte, dacă T este un arbore binar, cu chei de un tip total ordonat, și cu proprietatea că pentru orice nod u al său avem relațiile :

- (1') $info[u] > info[root(left[u])]$
- (2') $info[u] < info[root(right[u])]$

atunci relațiile (1) și (2) sunt adevărate și T este un arbore binar de căutare.

Să mai observăm că, din pricina inegalităților stricte de mai sus, într-un arbore binar de căutare aşa cum a fost definit până acum nu pot exista chei multiple; cu alte cuvinte, nu pot exista două noduri care să aibă aceeași valoare a cîmpului $info$. Vom numi un asemenea arbore un arbore binar de căutare *strict*. Problema cheilor multiple se poate rezolva în mai multe moduri. Dacă cheile nu sunt însotite de alte date semnificative, atunci am putea contoriza pur și simplu în fiecare nod numărul de aparitii

74

ale respectivei chei. Dacă însă cheile sunt însojite de alte date semnificative și dorim să reprezentăm efectiv cheile multiple în arbore ca noduri distincte, atunci putem folosi una din cele două structuri de arbore binar de căutare *nestrict*.

Numim arbore binar de căutare *nestrict la stânga* un arbore binar T cu proprietatea că în fiecare nod u al său avem relațiile:

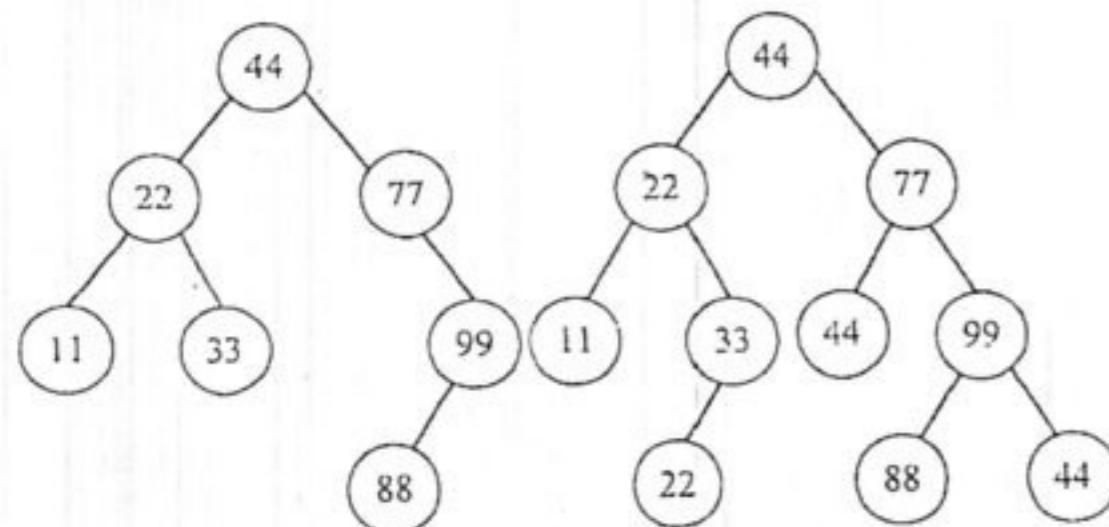
(3) $\text{info}[u] \geq \text{info}[v]$, pentru orice $v \in \text{left}[u]$

(4) $\text{info}[u] < \text{info}[w]$, pentru orice $w \in \text{right}[u]$.

Analog, se definește noțiunea de arbore binar de căutare *nestrict la dreapta*, cu relațiile:

(5) $\text{info}[u] > \text{info}[v]$, pentru orice $v \in \text{left}[u]$

(6) $\text{info}[u] \leq \text{info}[w]$, pentru orice $w \in \text{right}[u]$.



(a) Arbore binar de căutare strict (b) Arbore binar de căutare nestrict la dreapta. Cheile 22 și 44 sunt chei multiple.

Fig.2.4.1. Exemple de arbori binari de căutare

În sfârșit, o altă proprietate importantă a arborilor binari de căutare este aceea că parcurgerea în inordine (SRD) a unui asemenea arbore produce o listă ordonată crescător a cheilor. De aceea structura de arbore binar de căutare este potrivită pentru seturi de date pe care, pe lângă inserări și ștergeri, se cere destul de frecvent ordonarea totală a cheilor.

Căutarea într-un arbore binar de căutare

Într-un arbore binar de căutare operația de căutare a unei chei se desfășoară în felul următor: se pornește de la rădăcină în jos pe un drum care este determinat în fiecare nod prin alegerea ramurii stângi sau ramurii drepte în funcție de rezultatul comparării valorii căutate cu cheia nodului respectiv.

Să presupunem că avem arboarele binar de căutare reprezentat în Pascal cu ajutorul tipurilor de date :

```
type pnod = ^ nod;
nod = record
  info:integer;
  left, right : pnod;
end;
```

O funcție Pascal care implementează operația de căutare a unei valori Val , într-un arbore binar de căutare dat prin pointerul $Root$ către rădăcina lui este următoarea:

```
function Loc (Val: integer; Root: pnod): pnod;
var found: boolean;
begin
  found:= false;
  while (Root <> nil) and not found do
    begin
      if Root^.info = Val then
        found = true;
      else if Root^.info > Val then
        Root:= Root^.left
      else
        Root:= Root^.right
    end;
  Loc:= Root
end. {function Loc}
```

Funcția $Loc (Val, Root)$ va returna valoarea *nil* dacă cheia Val nu se găsește printre nodurile arborelui sau un pointer către nodul în care s-a găsit Val în cazul căutării cu succes. (Să observăm că, în acest din

urmă caz programul returnează prima apariție a valorii căutate.) Căutarea iterativă ce se desfășoară în ciclul *while* este asemănătoare căutării de la liste înlăncuite. Ea este guvernată de două condiții: condiția de neterminare a structurii (pointerul *Root* cu care facem parcurgerea să fie diferit de *nil*) și condiția de a nu fi găsit încă valoarea căutată, formulată cu ajutorul variabilei booleene *found*. Să ne reamintim faptul că la structuri lineare aveam tehnica componentei marcaj care ne permitea reducerea la jumătate a numărului de comparații la căutarea iterativă. O tehnică similară se poate aplica și în cazul arborilor. Putem completa un arbore binar de căutare cu un nod marcaj la sfârșit, un nod la care să fie legate toate ramurile arborelui. Completarea se poate face pur și simplu alocând spațiu pentru acest nou nod, *mEnd* și săcând toți pointerii *nil* ai arborelui egali cu *mEnd*.

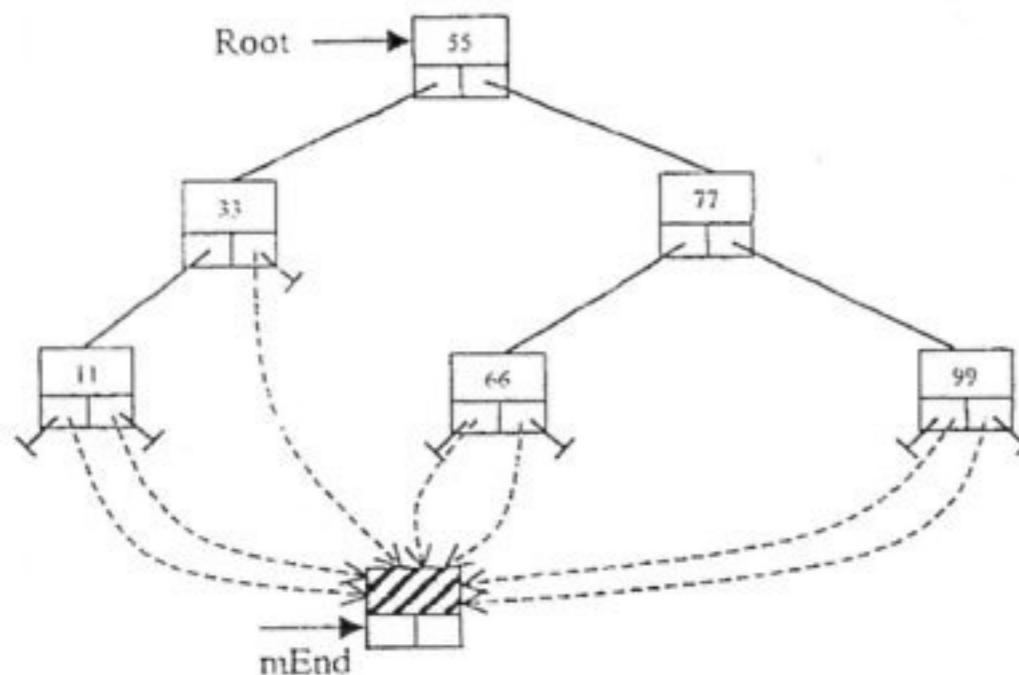


Fig.2.4.2. Arbore binar de căutare completat cu nod marcaj la sfîrșit

La căutare, se pune valoarea căutată în nodul marcaj. Procedura modificată de căutare arată în felul următor:

```
function Loc (Val: integer; Root: pnod): pnod;
begin
```

```
    mEnd^.info := Val; {se pune valoarea căutată în nodul marcaj}
    while Root^.info <> Val do
```

```
        if Val < Root^.info then Root := Root^.left
        else Root := Root^.right;
        Loc := Root
    end.
```

Loc va returna nodul marcaj *mEnd* în loc de valoarea *nil* la căutarea fără succes.

Exerciții

- Este corectă folosirea pointerului *Root* la parcurgerea cu căutare în function *Loc*? Justificați răspunsul.
- Modificați procedura de căutare a unei valori într-un arbore binar de căutare nestrict la dreapta astfel încât ea să returneze toate aparițiile unei chei.
- Scrieți o versiune recursivă a căutării unei chei într-un arbore binar de căutare.
- Scrieți un program care să completeze un arbore binar de căutare cu un nod marcaj la sfîrșit.

Căutare cu inserare într-un arbore binar de căutare

Ca pentru toate structurile dinamice, operația de inserare a unei chei într-un arbore binar de căutare este o operație de bază, cu ajutorul căreia se face construcția structurii. Inserarea unei chei într-un arbore binar de căutare trebuie precedată de o căutare a locului pentru inserare, ca și în cazul inserării într-o listă ordonată, astfel încât structura rezultată să fie tot un arbore binar de căutare.

Căutarea cu inserare trebuie să rezolve și problema cheilor multiple. În cele ce urmează vom rezolva această problemă prin contorizarea aparițiilor. Mai precis, vom căuta o valoare *Val* într-un arbore binar de căutare. Dacă *Val* se găsește deja în arbore vom incrementa un câmp suplimentar *contor* menținut în fiecare nod. Dacă *Val* nu se găsește în arbore ea va fi inserată ca un nod nou la capătul drumului de căutare (care este un sudarbore vid).

Presupunem date următoarele tipuri de date pentru reprezentarea nodurilor arborelui și a pointerilor către ele:

```
type pnod = ^nod
        nod = record
            info: integer;
```

```

        contor: integer;
        left, right: pnod;
    end

Procedura SearchIns (x, p) este o procedură recursivă care caută valoarea x în arborele binar de căutare de rădăcină p și o inserează dacă nu o găsește, iar dacă o găsește incrementează câmpul contor al nodului respectiv.

procedure SearchIns (x: integer; var p: pnod);
begin
    if p= nil then {x nu a fost găsit și va fi inserat}
        begin
            new (p)
            with p do begin {completarea câmpurilor noului nod}
                info:= x
                contor:= 1
                left:= nil
                right:= nil
            end
        end
    else {p≠ nil}
        if x<p.info then
            SearchIns (x, p.left)
        else if x>p.info then
            SearchIns (x, p.right)
        else {x a fost găsit și se incrementează contorul}
            p.contor:= p.contor + 1
endproc {SearchIns}
```

Procedura poate fi folosită de un program de construcție a unui arbore binar de căutare prin inserări repetitive de chei citite dintr-un fișier, a cărui secvență principală este:

```

begin
{program construcție arbore binar de căutare prin inserări repetitive}
Root:= nil {initializarea arborelui cu arborele vid}
while not eof do {dacă fișierul este consola}
    begin
        read (x)
        SearchIns (x, Root)
    end
end.
```

După cum am văzut căutarea se simplifică prin folosirea unui nod marcat la sfârșitul arborelui, *mEnd*. Dacă folosim un asemenea nod, procedura de căutare cu inserare devine:

```

procedure SearchIns1 (x: integer; var p: pnod);
begin
    if x<p.info then SearchIns1 (x, p.left )
    else if x>p.info then SearchIns1 (x, p.right)
    else if p=mEnd then
        p.contor= p.contor+1
    else {p= mEnd, deci trebuie să inserăm}
        begin
            new(p);
            with p do begin
                info:= x
                contor:= 1
                left:= mEnd
                right:= mEnd
            end
        end
end; {SearchIns1}
```

Vom da în continuare o versiune iterativă a căutării cu inserare. Ca și în cazul inserării iterative într-o listă ordonată, căutarea trebuie făcută parcurgând arborele cu doi pointeri, *p1* către nodul curent și *p2* tot timpul cu un pas în urma lui *p1*, căci inserarea propriu-zisă va fi făcută când *p1* devine *nil*, caz în care *p2* va fi folosit pentru a lega noul nod la restul arborelui.

O variabilă suplimentară, *d*, va fi folosită pentru a codifica direcția în care plecăm dintr-un nod pentru a găsi locul inserării. Vom codifica cu *d*=-1, dacă am plecat la stânga și cu *d*=+1, dacă am plecat la dreapta. Astfel valoarea lui *d* ne va spune dacă noul nod inserat apare ca fiu stâng (*d* = -1) sau drept (*d* = +1) al lui *p2*. Deoarece lucrăm tot pe varianta care contorizează cheile multiple putem folosi valoarea 0 a variabilei *d* pentru a codifica faptul că valoarea *x* de inserat se găsește deja în arbore.

```

procedure SearchInsIterativ (x: integer; root: pnod);
var p1, p2: pnod; d: integer;
begin {initializarea pointerilor pentru parcurgere}
p2:= nil;
p1:= root;
while (p1<>nil) and (d<>0) do
  if x< p1^.info then
    p2:= p1
    p1:= p1^.left
    d:= -1
  else if x> p1^.info then
    p2:= p1
    p1:= p1^.right
    d:= 1
  else {x= p1^.info}
    d:= 0
  if p1<>nil then
    {d=0 și am găsit x în arborele root, deci trebuie incrementat contorul}
    p1^.contor:= p1^.contor+1
  else {p1=nil și facem inserarea}
    begin
      new(p1)
      with p1^ do begin
        info:= x
        contor:= 1
        left:= nil
        right:= nil
      end
      {legarea noului nod la tata}
      if p2=nil then {cazul inserării într-un arbore vid}
        root:= p1
      else if d<0 then
        p2^.left:= p1
      else p2^.right:= p1
    end
  end
endproc {procedura SearchInsIterativ}

```

Să observăm că trebuie să tratăm separat cazul inserării primului nod în arbore.

if p2=nil then root:= p1.

Puteam scăpa de tratarea separată a acestui caz lucrând, ca și în cazul listelor, cu arbori cu nod marcat la început. Pointerul *root* va indica către acest nod marcat, iar fiul lui drept (de exemplu) va conține un pointer către arborele propriu-zis. Inițializarea unui astfel de arbore în programul de creare prin inserări repetate se va face prin următoarea secvență:

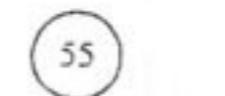
```

new (Root);
Root^.right := nil;

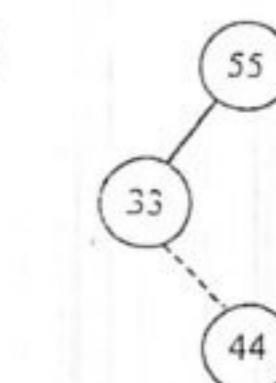
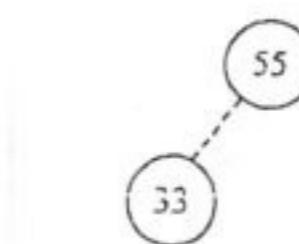
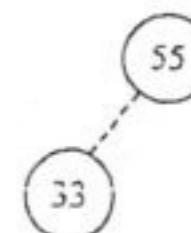
```

La versiunea iterativă a căutării cu inserare se va modifica inițializarea parcurgerii și de asemenea se va simplifica legarea la tată a noului nod.

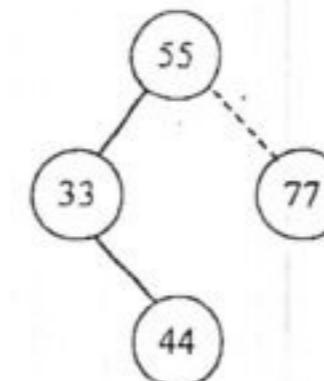
Ilustrăm în cele ce urmează crearea unui arbore binar de căutare prin inserări repetate:



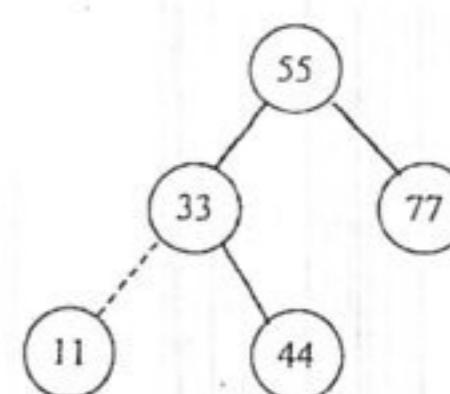
(a) Se inserează 55



(c) Se inserează 44



(d) Se inserează 77



(e) Se inserează 11

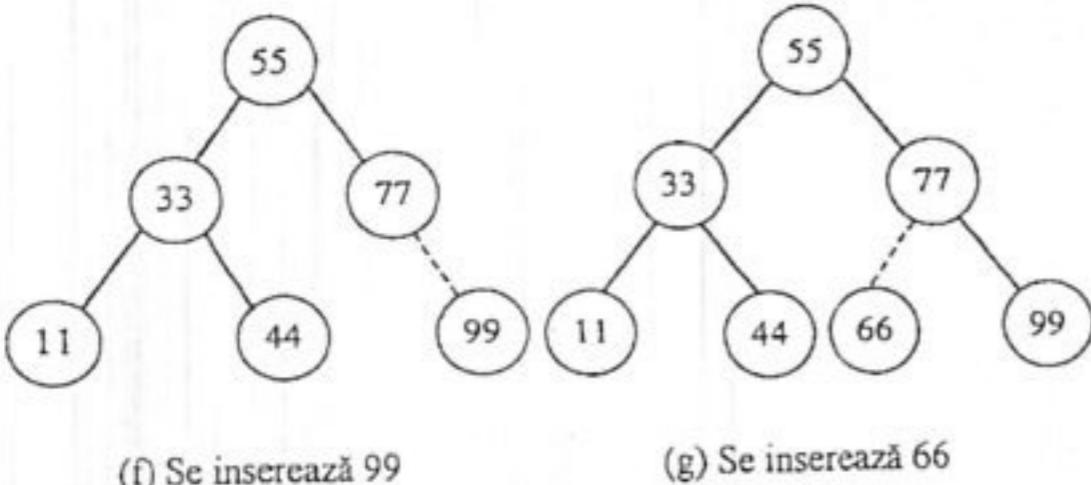


Fig.2.4.3. Arborele binar de căutare rezultat din citirea și inserarea pe rînd a cheilor : 55, 33, 44, 77, 11, 99, 66

Exercitii

- Să se scrie versiunea iterativă a căutării cu inserare într-un arbore binar de căutare cu nod marcat la început.
 - Să se modifice procedurile de căutare cu inserare astfel încât rezultatul să fie un arbore binar de căutare nestrict la dreapta. Cu alte cuvinte cheile multiple sunt inserate efectiv (chiar dacă găsim o cheie cu valoarea x în arbore continuăm să căutăm primul loc gol din subarborele drept și o inserăm efectiv acolo), iar câmpul contor dispare.
 - Scripteți o procedură de căutare într-un arbore binar de căutare nestrict la dreapta, care să returneze un pointer către ultima apariție a unei chei (prin ultima apariție a unei chei multiple se înțelege ultima cheie cu această valoare care a fost inserată)
 - Dați o secvență de date de intrare care citite și inserate pe rând cu algoritmul de căutare și inserare, să conducă la arborele binar de căutare din figura 2.4.1.

Alte operații pe un arbore binar de căutare

Reamintim că un arbore binar de căutare parcurs în inordine ne dă o listă ordonată crescător a cheilor sale. De aceea arborele binar de căutare este o structură de date potrivită și pentru sortare.

Alte operații care se implementează destul de ușor pe el sunt operația de extragere a minimului și cea simetrică, de extragere a maximului. Dacă vrem să găsim nodul cu cheia cea mai mică dintr-un

arbore binar de căutare trebuie să urmărească mai lung drum de la rădăcină plecând mereu la stânga în fiecare nod. Similar, maximul se va găsi în "extremitatea dreaptă" a arborelui.

Următoarea funcție returnează un pointer către nodul care conține cea mai mică cheie din arborele binar de căutare *root*.

```

function min (root: pnod): pnod;
    var p: pnod;
begin
    p := root;
    while ( p <> nil) and (p^.left <> nil) do
        p:=p^.left;
    min:=p
end;

```

Dacă rădăcina *root* nu are fiu stâng, funcția va returna *min=root*, căci în rădăcină se găsește cea mai mică cheie din arbore. Funcția de găsire a maximului este similară, decât că face parcurgerea mergând mereu la dreapta.

Exerciti

1. Scrieți o funcție care returnează un pointer către nodul cu valoarea maximă.
 2. Scrieți o versiune recursivă a găsirii minimului (maximului) dintr-un arbore binar de căutare.
 3. Funcționează algoritmii de mai sus pe un arbore binar de căutare nesortat?

Stergerea unui nod dintr-un arbore binar de căutare

Ne ocupăm acum de problema inversă inserării, problema ștergerii unui nod cu cheie dată, x , dintr-un arbore binar de căutare. Dacă nodul cu cheia x nu are nici un fiu nevid sau are un singur fiu nevid, problema e simplă: singurul fiu nevid (dacă există) se leagă la tatăl nodului cu cheia x , ca fiu stâng dacă x a fost fiu stâng sau ca fiu drept dacă x a fost fiu drept. Dificultăți apar când avem de șters un nod care are ambii fiu nevizibili. O soluție care afectează cel mai puțin forma generală a arborelui ar fi să păstrăm nodul efectiv și să înlocuim doar valoarea x cu o altă valoare, tot

din arbore, care să poată juca, ca și x , rolul de separator între valorile din subarborele stâng și cele din subarborele drept al lui x .

În continuare, prezentăm o versiune recursivă a procedurii de ștergere a unui nod dintr-un arbore binar de căutare.

```

procedure Delete (x: integer; var p: leg);
var q: leg;
procedure Del (var r: leg);
begin
    if r.right <> nil then
        Del (r.right)
    else begin
        q.info := r.info
        q.contor := r.contor
        q := r
        r := r.left
    end
end; {proc. Del}
begin
    if p = nil then writeln ('Nu am găsit')
    else if x < p.info then
        Delete (x, p.left)
    else if x > p.info then
        Delete (x, p.right)
    else begin {ștergere p}
        q := p
        if q.right = nil then
            p := q.left
        else if q.left = nil then
            p := q.right
        else Del (q.left)
    end
end

```

O variantă iterativă a căutării cu ștergere a unui nod dintr-un arbore binar de căutare este implementată de procedura *SearchDel*. Ea utilizează procedurile *Delete1* și *Delete2* care șterg un nod cu un fiu, respectiv cu doi fii.

```

procedure Search Del (x: integer; root: leg );
var p1, p2, falseroot: leg;
{p1, p2 pointeri curenti, falseroot pentru nod fals înainte de rădăcina }

```

```

procedure Delete1(var p: leg); {Șterge un nod p cu cel mult un succesor}
begin

```

```

    if p.left = nil then
        p := p.right
    else p := p.left
end; {Delete 1}

```

```

procedure Delete 2 (var p: leg ); {șterge un nod p cu doi succesi}
{caută predecesorul în inordine al lui p.info mergând un pas la stânga,
apoi la dreapta cât se poate. Parcurgerea se face cu r și q = tată r }
var q, r: leg; { d1 = -1 <=> r = q.left }
d1: integer; { d1 = 1 <=> r = q.right }
begin

```

```

(a)   q := p
      r := p.left
      d1 := -1
      while r.right <> nil do
          begin
              q := r
              r := r.right;
              d1 := 1
          end

```

```

(b)   p.info := r.info; {Se copiază în p valorile din r}
      p.contor = r.contor;

```

```

(c) {Se leagă de tată, q, subarborele stâng al lui r }
      if d1 < 0 then

```

```

          q.left := r.left
      else q.right := r.left
end; {Delete 2}

```

```

begin {Search Del}

```

```

    new( falseroot );
    falseroot.right := root ; {adăugăm nod marcaj}

```

```

p1 := root;
p2 := falseroot;
d := 1
found := false
while (p1 ≠ nil) and not found do
begin
    p2 := p1
    if x < p1↑.info then begin
        p2 := p1
        p1 := p1↑.left
        d := -1
    end
    else if x > p1↑.info then begin
        p2 := p1
        p1 := p1↑.right
        d := 1
    end
    else found := true
end;
if not found then
    "Nu am găsit"
else {found = true și trebuie să șterg nodul p1↑}
begin
    if (p1↑.left=nil) or (p1↑.right = nil) then
        Delete1 (p1) {ștergere caz 1}
    else Delete 2 (p1); {ștergere caz 2}
    {legarea noului nod p1↑ de tatăl său p2↑}
    if d > 0 then
        p2↑.right := p1
    else p2↑.left := p1
end
end; {procedure SearchDel}

```

Cazul 1: Nodul de șters, $p1^\uparrow$, are cel mult un fiu: "Noul" nod $p1^\uparrow$ returnat de $Delete1$, va fi acest fiu (chiar dacă este nil).

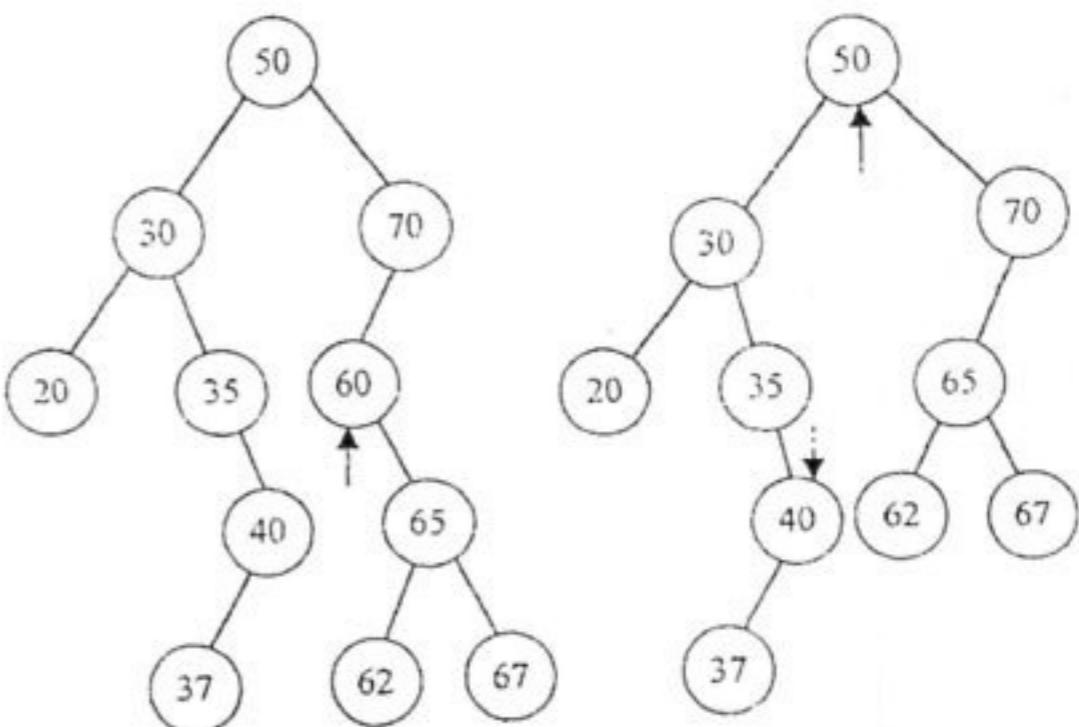
Cazul 2: Nodul de șters $p1^\uparrow$, are doi fiu: procedura $Delete2$

(a) caută predecesorul în inordine al lui $p1^\uparrow.info$ adică merge un pas la stânga, apoi la dreapta cât poate cu doi pointeri curenți r și q tatăl lui r , și $d1$ pentru codificat dacă fiul e stâng sau drept.

(b) copiază în toate câmpurile lui $p1^\uparrow$, cu excepția câmpurilor pointeri către fiu, valorile din r^\uparrow .

(c) leagă pe tatăl lui r^\uparrow , q^\uparrow , de fiul stâng al lui r^\uparrow (chiar dacă e nil), în funcție de valoarea lui $d1$.

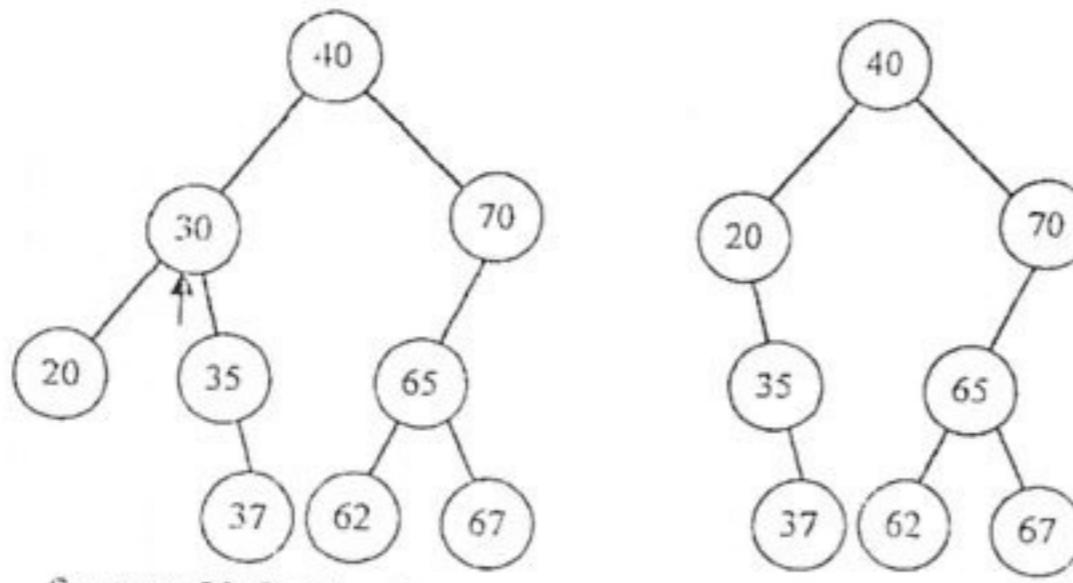
La ieșirea din $Delete1$ sau $Delete2$, nodul $p1^\uparrow$ este "noul" nod obținut prin înlocuirea celui vechi după caz. El trebuie legat de tatăl său, $p2^\uparrow$, în funcție de valoarea lui d .



Se șterge 60. Unicul său fiu devine noul fiu stâng al lui 70.

Se șterge 50.
(a) Se caută predecesorul în inordine al lui 50 în arbore. El este 40.

(b) Se șterge predecesorul, care are cel mult un fiu și urcă 40 ca și cheie separatoare în locul lui 50.



Se șterge 30. Se înlocuiește cu predecesorul în inordine, 20.

Arborele rezultat în urma ștergerii pe rând a cheilor 60, 50, 30.

Fig. 2.4.4. Ștergeri de noduri dintr-un arbore binar de căutare.

Complexitatea operațiilor pe un arbore binar de căutare

Operațiile de inserare și ștergere de noduri într-un arbore binar de căutare depind în mod esențial de operația de căutare. Căutarea revine la parcurgerea, eventual incompletă, a unei ramuri, de la rădăcină până la un nod interior în cazul căutării cu succes, sau până la primul fiu vid întâlnit în cazul căutării fără succes (și al inserării). Performanța căutării depinde de lungimea ramurilor pe care se caută; media ei va fi dată de lungimea medie a ramurilor, iar dimensiunea maximă de lungimea celor mai lungi ramuri, adică de adâncimea arborelui. Forma arborelui, deci și adâncimea depind, cu algoritmi dați, de ordinea introducerii cheilor și putem avea cazul cel mai nefavorabil, în care adâncimea arborelui este n , deci performanța operației de căutare este de același ordin ca cea de la structurile lineare, și anume $O(n)$.

În capitolul IV vom estima lungimea medie a drumului de la rădăcină până la o frunză și adâncimea unui arbore binar. Anticipând puțin, avem o limită inferioară pentru adâncime de ordinul lui $\log_2 n$, ceea ce înseamnă că performanța operației de căutare nu poate coborâ sub ea. Ne punem problema dacă putem atinge această limită optimă.

2.5. Arboare binari de căutare echilibrați AVL

Procedeul clasic de construire a arborelui binar de căutare ne dă un arbore a cărui formă depinde foarte mult de ordinea în care sunt furnizate valorile nodurilor. În cazul cel mai general nu obținem un arbore de înălțime minimă.

Cazul cel mai favorabil, în care obținem înălțime minimă, este cel în care ni se furnizează pe rând mijloacele intervalelor (subintervalelor) vectorului sortat.

Cazul cel mai nefavorabil este cel în care valorile vin în ordine crescătoare (sau descrescătoare), caz în care arborele binar de căutare obținut este degenerat (e chiar o listă înlanțuită, cu legăturile date de fișe drepti, cei stângi fiind toți nil (dacă valorile vin în ordine crescătoare)).

Problemă: cum modificăm algoritmul de construcție astfel încât să obținem înălțime minimă pentru arbore, pentru a îmbunătăți timpul de căutare?

Să observăm că la inserarea unui nou element crește cu 1 înălțimea subarborelui în care s-a făcut inserția. Ne propunem, pentru noua metodă de construcție, următorul criteriu: diferența dintre înălțimile fiului stâng și cel drept să nu depășească pe 1.

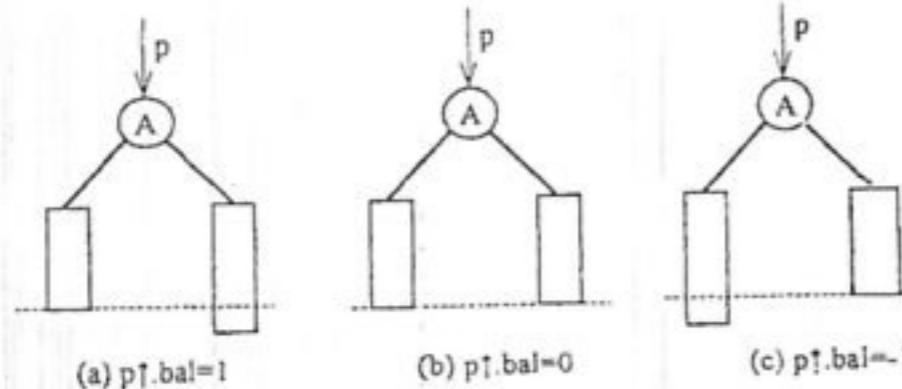
Definiție. Se numește *arbore binar de căutare echilibrat AVL* (Adelson-Velskii-Landis) un arbore care în fiecare nod are proprietatea că înălțimile subarborelor stâng și drept diferă cu cel mult 1.

Pentru un nod dat, fie h_l și h_r înălțimile subarborelui stâng, respectiv drept. Avem trei situații posibile în acest nod, codificate cu valorile variabilei $bal = h_r - h_l$, pe care o numim *factor de echilibru*, în felul următor:

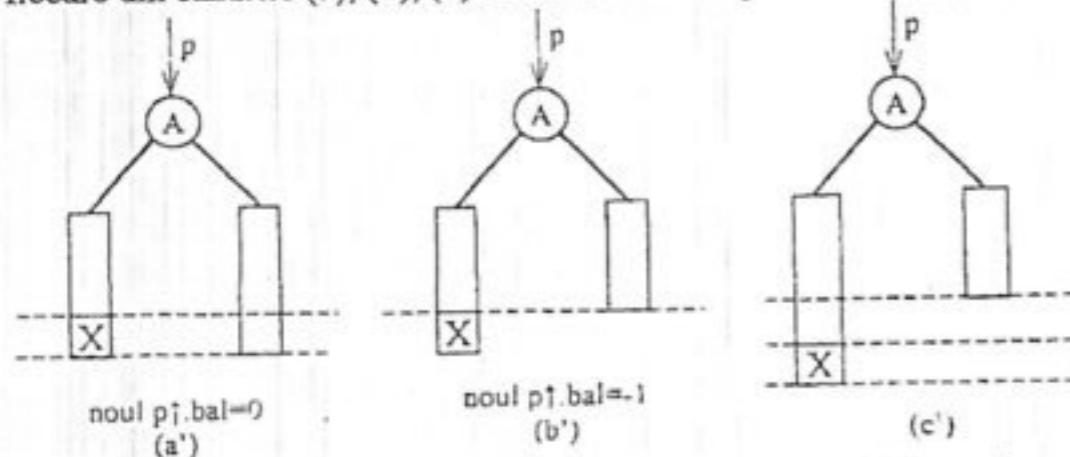
$$bal = \begin{cases} 1, & h_l = h_r - 1 \\ 0, & h_l = h_r \\ -1, & h_l = h_r + 1 \end{cases}$$

Informația despre valoarea factorului de echilibru în fiecare nod p al unui arbore o vom scrie într-un nou câmp al lui p , câmpul $bal: -1..1$.

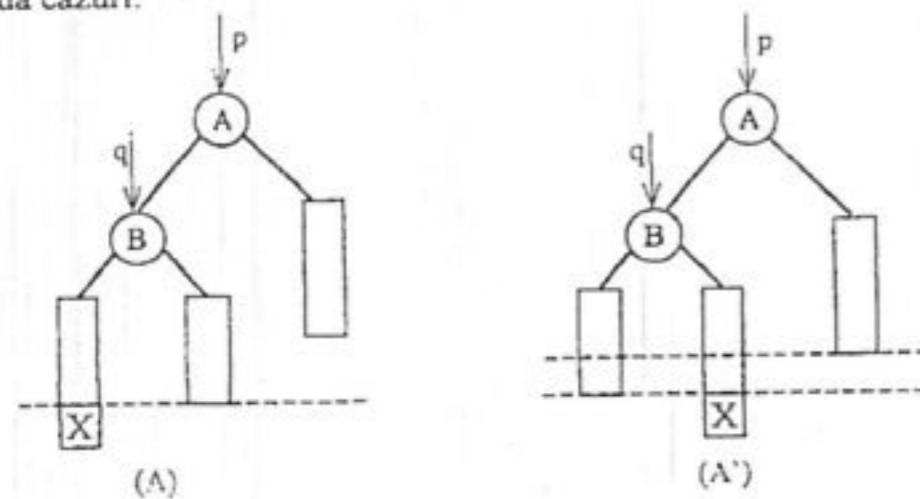
Dăm în continuare o prezentare a modului în care, cu ajutorul unor operații suplimentare numite *reechilibrări*, vom putea face inserarea de noduri noi într-un arbore binar de căutare păstrând în același timp proprietatea ca arborele să fie "echilibrat" în fiecare nod, adică variabila bal să rămână în domeniul $[-1..1]$. Cele trei situații de la care plecăm sunt



Presupunând că inserăm acum un nou element în subarborele stâng, fiecare din cazurile (a), (b), (c) se transformă respectiv în (a'), (b'), (c'):

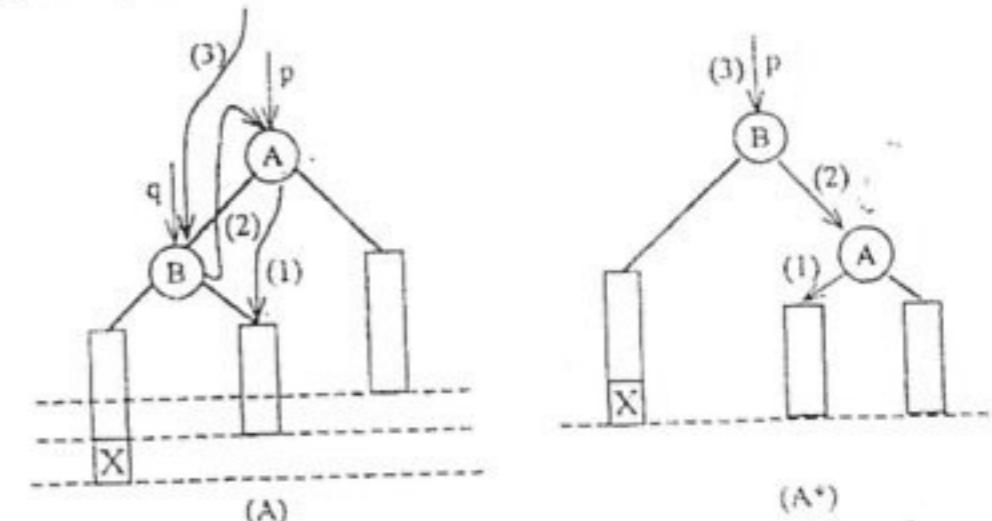


Observăm că în cazurile (a') și (b') proprietatea de echilibrare în nodul $p \uparrow$ s-a păstrat, schimbându-se doar valoarea factorului de echilibru. În cazul (c') însă, arborele de rădăcină $p \uparrow$ nu mai este echilibrat, și trebuie să-l reechilibrem. Pentru acesta, să ne uităm mai în detaliu pe subarborele stâng unde s-a produsdezechilibrarea. Avem următoarele două cazuri:



Am desfășurat subarborele stâng, presupunem că are rădăcina B, și fie $q := p \uparrow.left$.

Cazul (A): Putem reechilibra arborele, făcând din B rădăcina, pe A fiu drept al lui B, iar fiul drept al lui B să devină fiu stâng al lui A, reasignând niște valori unor pointeri, adică resetând legăturile (1),(2),(3):



Secvența de instrucțiuni care realizează trecerea de la (A) la (A*), trecere care se numește *rotație SS* (Stânga-Stânga) este:

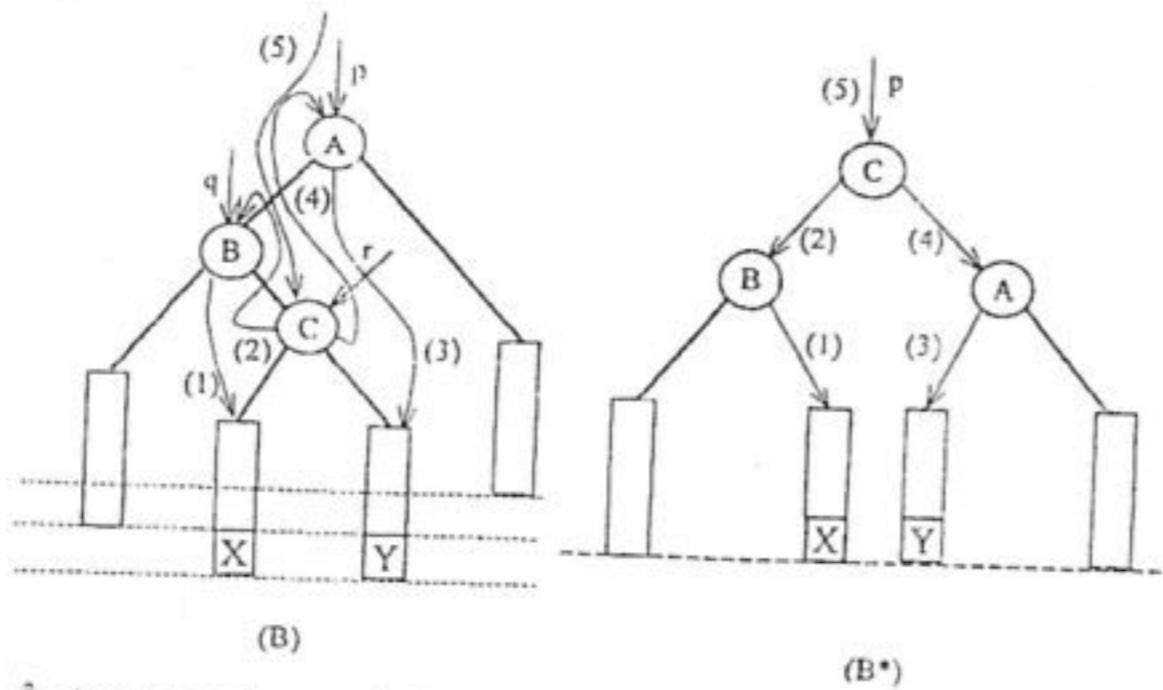
- (1) $p \uparrow.left := q \uparrow.right$
- (2) $q \uparrow.right := p$
{înainte de reasignarea lui p calculăm noul factor de echilibru în A}
 $p \uparrow.bal := 0$
- (3) $p := q$

secvență urmată de calcularea factorului de echilibru pentru noul arbore:
 $p \uparrow.bal = 0$.

Cazul (A'): Nu putem folosi același procedeu ca la (A), deoarece am obținut un arbore neechilibrat. Desfășurând subarborele drept al lui $q \uparrow$, obținem situația din figura (B), unde noul nod inserat este fie Y, fie X, iar cazul (A') devine

Cazul (B): Reechilibram făcând pe C rădăcină, cu fiu stâng B, fiu drept A, iar vechiul fiu stâng al lui C devine noul fiu drept al lui B, vechiul fiu drept al lui C devine noul fiu stâng al lui A. Reasignările de pointeri care realizează noile legături (1),(2),(3),(4) sunt:

- (1) $q \uparrow.right := r \uparrow.left$
- (2) $r \uparrow.left := q$
- (3) $p \uparrow.left := r \uparrow.right$
- (4) $r \uparrow.right := p$



Înainte de reasignarea lui p către noua rădăcină trebuie să calculăm noi factori de echilibru în nodurile A ($p \uparrow$) și B ($q \uparrow$) în funcție de ce anume s-a inserat: nodul X sau nodul Y , cu secvența:

```
if  $r \uparrow .bal = -1$  then {s-a inserat  $X$ }
begin  $q \uparrow .bal := 0$ ;
       $p \uparrow .bal := 1$ 
end
```

```
else { $r \uparrow .bal = 1$ , adică s-a inserat  $Y$ }
begin  $q \uparrow .bal := -1$ ;
       $p \uparrow .bal := 0$ 
end;
```

(5) $p := r$ (reasignarea pointerului către rădăcină)

Urmează apoi calculul factorului de echilibru pentru arborele din figura (B*) reechilibrat

$p \uparrow .bal := 0$

Trecerea de la arborele de tip (B) la (B*) poartă numele de *rotație SD* (Stânga-Dreapta).

Cele spuse mai sus se aplică și pentru cazul în care se inserează un nod nou pe subarborele drept al unui arbore de rădăcină $p \uparrow$. Vom avea atunci încă două cazuri în care trebuie să facem reechilibrarea, simetricele cazurilor (A) și (B), care se tratează analog. Simetricul cazului (A) va conduce la *rotație DD*, iar al cazului (B) la *rotație DS*.

Procedura recursivă $Search(x, p)$ care caută și eventual inserează un nod nou x în arborele de rădăcină $p \uparrow$ se va modifica. În lista de parametri mai apare o variabilă booleană, h , ce se transmite procedurii apelatoare, cu semnificația:

$h = \begin{cases} \text{true}, & \text{dacă s-a modificat înălțimea arborelui de rădăcină } p \uparrow \\ \text{false}, & \text{în caz contrar.} \end{cases}$

Observăm că în ambele cazuri de reechilibrare, înălțimea arborelui reechilibrat este egală cu înălțimea arborelui dinainte de inserția care a stricat echilibrul, deci după reechilibrări h trebuie să ia valoarea false. Modificarea procedurii $Search$ se face în felul următor: după fiecare apel al ei, se testează h , iar dacă $h = \text{true}$, avem de tratat separat cazurile pentru cele trei valori ale lui $p \uparrow .bal$.

Pentru nodurile unui arbore AVL vom folosi următoarele definiții de tip:

```
type leg =  $\uparrow$ nod;
nod = record
  info:integer;
  left, right:leg;
  bal:-1..1
end;
```

Procedura recursivă de inserare cu reechilibrare este prezentată în continuare. Ea este o modificare a procedurii recursive de căutare cu inserare $Search/ins$ prezentată în secțiunea 2.4 pentru arborele binar de căutare. Modificările apar la întoarcerea din apelurile recursive care fac inserări în subarborele stâng, respectiv în cel drept.

```
procedure Search (x:integer; var p:leg; var h:boolean);
var q,r:leg;
begin
  if  $p = \text{nil}$  then {inserare nod}
    begin new(p);
      with  $p \uparrow$  do begin info:=x;
        left:=nil;
        right:=nil;
        bal:=0 {apare acum}
      end
      h:=true;
```

```

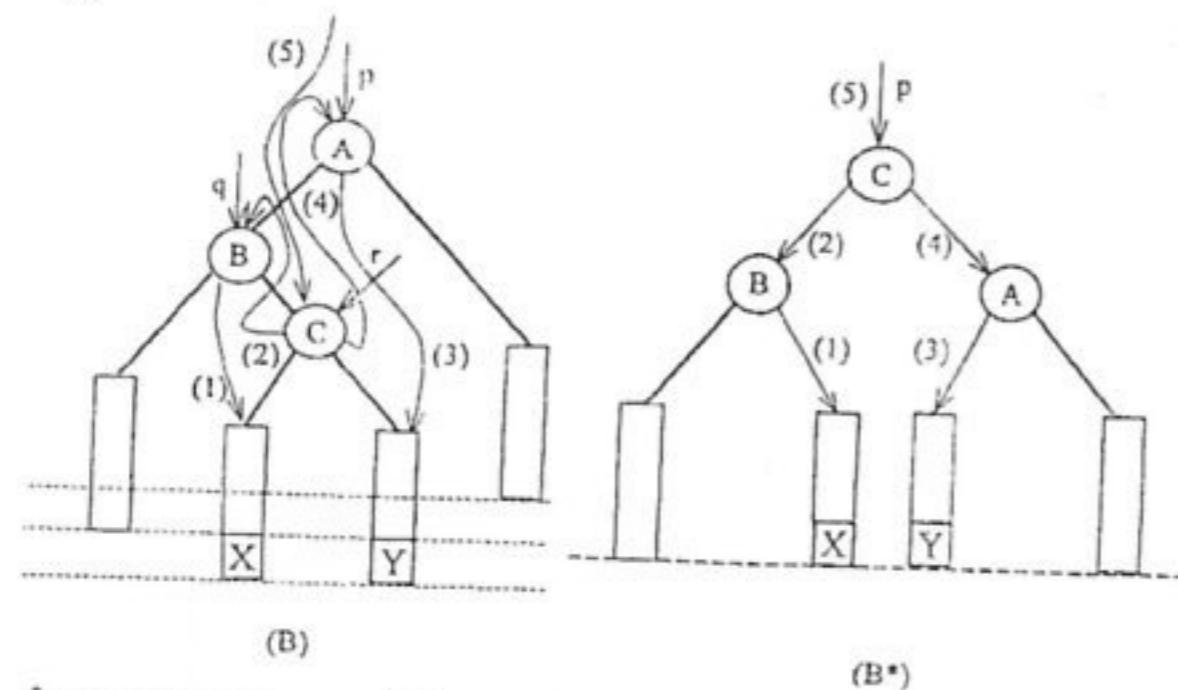
        end
    else
        if x < p↑.info then
            begin
                Search (x, p↑.left, h);
                {de aici incepe modificarea față de SearchIns}
                if h then {a crescut înălțimea ramurii stângi}
                    case p↑.bal of
                        1: {acum cele două ramuri ale lui p↑ sunt egale}
                            begin p↑.bal:=0;
                                h:=false;
                            end
                        0: {ramura din stânga e mai lungă}
                            p↑.bal:=-1 {h rămâne true}
                        -1: {reechilibrăm}
                            begin
                                q:=p↑.left;
                                if q↑.bal = -1 then {cazul (A) rotație SS}
                                    begin p↑.left:=q↑.right; {1}
                                        q↑.right:=p; {2}
                                        p↑.bal:=0;
                                        p:=q {3}
                                    end
                                else {cazul (B) rotație SD}
                                    begin r:=q↑.right;
                                        q↑.right:=r↑.left; {1}
                                        r↑.left:=q; {2}
                                        p↑.left:=r↑.right; {3}
                                        r↑.right:=p; {4}
                                        if r↑.bal = -1 then
                                            {s-a inserat X pe r↑.left}
                                            begin q↑.bal:=0; p↑.bal:=1; end
                                        else {s-a inserat Y pe r↑.right}
                                            begin q↑.bal:=-1; p↑.bal:=0; end;
                                            p:=r {5}
                                    end;
                                p↑.bal:=0; h:=false
                            end;
                        end { case p↑.bal=-1}
                    end{case};
                end {inserarea cu reechilibrare pe ramura stângă}

```

```

else {x>=p↑.info} {inserarea cu reechilibrare pe ramura stângă}
if x>p↑.info then
begin
    Search (x, p↑.right, h);
    if h then {a crescut înălțimea ramurii drepte}
        case p↑.bal of
            -1: {cele două ramuri ale lui p↑ au devenit egale}
                begin p↑.bal:=0;
                    h:=false;
                end
            0: {ramura dreaptă e mai lungă}
                p↑.bal:=1 {h rămâne true}
            1: {reechilibrăm}
                begin
                    q:=p↑.right;
                    if q↑.bal=1 then {cazul simetric lui A}
                        begin {rotație DD}
                            p↑.right:=q↑.left;
                            q↑.left:=p;
                            p↑.bal:=0;
                            p:=q
                        end
                    else begin {rotație DS}
                        r:=q↑.left;
                        q↑.left:=r↑.right;
                        r↑.right:=q;
                        p↑.right:=r↑.left;
                        r↑.left:=p;
                        if r↑.bal = +1 then begin
                            p↑.bal:=-1; q↑.bal:=0 end
                        else {r↑.bal = -1} begin
                            q↑.bal:=-1; p↑.bal:=0 end;
                            p:=r
                        end
                    end;
                    p↑.bal = 0; h:=false;
                end;
            end{case}
        end
    else {x = p↑.info și incrementăm contorul}

```



Înainte de reasignarea lui p către noua rădăcină trebuie să calculăm noi factori de echilibru în nodurile A ($p \uparrow$) și B ($q \uparrow$) în funcție de ce anume s-a inserat: nodul X sau nodul Y , cu secvența:

```
if  $r \uparrow .bal = -1$  then {s-a inserat  $X$ }
begin  $q \uparrow .bal := 0$ ;
       $p \uparrow .bal := 1$ 
end
```

```
else { $r \uparrow .bal = 1$ , adică s-a inserat  $Y$ }
begin  $q \uparrow .bal := -1$ ;
       $p \uparrow .bal := 0$ 
end;
```

(5) $p := r$ {reasignarea pointerului către rădăcină}

Urmează apoi calculul factorului de echilibru pentru arborele din figura (B*) reechilibrat

$p \uparrow .bal := 0$

Trecerea de la arborele de tip (B) la (B*) poartă numele de *rotatie SD* (Stânga-Dreapta).

Cele spuse mai sus se aplică și pentru cazul în care se inserază un nod nou pe subarborele drept al unui arbore de rădăcină $p \uparrow$. Vom avea atunci încă două cazuri în care trebuie să facem reechilibrarea, simetricele cazurilor (A) și (B), care se tratează analog. Simetricul cazului (A) va conduce la *rotatie DD*, iar al cazului (B) la *rotatie DS*.

Procedura recursivă $Search(x, p)$ care caută și eventual inserază un nod nou x în arborele de rădăcină $p \uparrow$ se va modifica. În lista de parametri mai apare o variabilă booleană, h , ce se transmite procedurii apelatoare, cu semnificația:

$$h = \begin{cases} \text{true}, & \text{dacă s-a modificat înălțimea arborelui de rădăcină } p \uparrow \\ \text{false}, & \text{în caz contrar.} \end{cases}$$

Observăm că în ambele cazuri de reechilibrare, înălțimea arborelui reechilibrat este egală cu înălțimea arborelui dinainte de inserția care a stricat echilibrul, deci după reechilibrări h trebuie să ia valoarea false. Modificarea procedurii $Search$ se face în felul următor: după fiecare apel al ei, se testează h , iar dacă $h = \text{true}$, avem de tratat separat cazurile pentru cele trei valori ale lui $p \uparrow .bal$.

Pentru nodurile unui arbore AVL vom folosi următoarele definiții de tip:

```
type leg = ↑nod;
nod = record
  info:integer;
  left, right:leg;
  bal:-1..1
end;
```

Procedura recursivă de inserare cu reechilibrare este prezentată în continuare. Ea este o modificare a procedurii recursive de căutare cu inserare $SearchIns$ prezentată în secțiunea 2.4 pentru arborele binar de căutare. Modificările apar la întoarcerea din apelurile recursive care fac inserări în subarborele stâng, respectiv în cel drept.

```
procedure Search (x:integer; var p:leg; var h:boolean);
var q,r:leg;
begin
  if  $p = \text{nil}$  then {inserare nod}
  begin new(p);
    with p do begin info:=x;
              left:=nil;
              right:=nil;
              bal:=0 {apare acum}
            end
    h:=true;
```

```

        end
    else
        if x < p↑.info then
            begin
                Search (x, p↑.left, h);
                {de aici incepe modificarea față de SearchIns}
                if h then {a crescut înălțimea ramurii stângi}
                    case p↑.bal of
                        1: {acum cele două ramuri ale lui p↑ sunt egale}
                            begin p↑.bal:=0;
                                h:=false;
                            end
                        0: {ramura din stânga e mai lungă}
                            p↑.bal:=-1 {h rămâne true}
                        -1: {reechilibrăm}
                            begin
                                q:=p↑.left;
                                if q↑.bal = -1 then {cazul (A) rotație SS}
                                    begin p↑.left:=q↑.right; {1}
                                        q↑.right:=p; {2}
                                        p↑.bal:=0;
                                        p:=q {3}
                                    end
                                else {cazul (B) rotație SD}
                                    begin r:=q↑.right;
                                        q↑.right:=r↑.left; {1}
                                        r↑.left:=q; {2}
                                        p↑.left:=r↑.right; {3}
                                        r↑.right:=p; {4}
                                        if r↑.bal = -1 then
                                            {s-a inserat X pe r↑.left}
                                            begin q↑.bal:=0; p↑.bal:=1; end
                                        else {s-a inserat Y pe r↑.right}
                                            begin q↑.bal:=-1; p↑.bal:=0; end;
                                        p:=r {5}
                                    end;
                                p↑.bal:=0; h:=false
                            end {case p↑.bal=-1}
                        end{case}
                    end {inserarea cu reechilibrare pe ramura stângă}
    
```

```

else {x>=p↑.info} {inserarea cu reechilibrare pe ramura stângă}
if x>p↑.info then
begin
    Search (x, p↑.right, h);
    if h then {a crescut înălțimea ramurii drepte}
        case p↑.bal of
            -1: {cele două ramuri ale lui p↑ au devenit egale}
                begin p↑.bal:=0;
                    h:=false;
                end
            0: {ramura dreaptă e mai lungă}
                p↑.bal:=1 {h rămâne true}
            1: {reechilibrăm}
                begin
                    q:=p↑.right;
                    if q↑.bal=1 then {cazul simetric lui A}
                        begin {rotație DD}
                            p↑.right:=q↑.left;
                            q↑.left:=p;
                            p↑.bal:=0;
                            p:=q
                        end
                    else begin {rotație DS}
                        r:=q↑.left;
                        q↑.left:=r↑.right;
                        r↑.right:=q;
                        p↑.right:=r↑.left;
                        r↑.left:=p;
                        if r↑.bal = +1 then begin
                            p↑.bal:=-1; q↑.bal:=0 end
                        else {r↑.bal = -1} begin
                            q↑.bal:=-1; p↑.bal:=0 end;
                            p:=r
                        end
                    end;
                    p↑.bal = 0; h:=false;
                end;
            end{case}
        end
    else {x = p↑.info și incrementăm contorul}

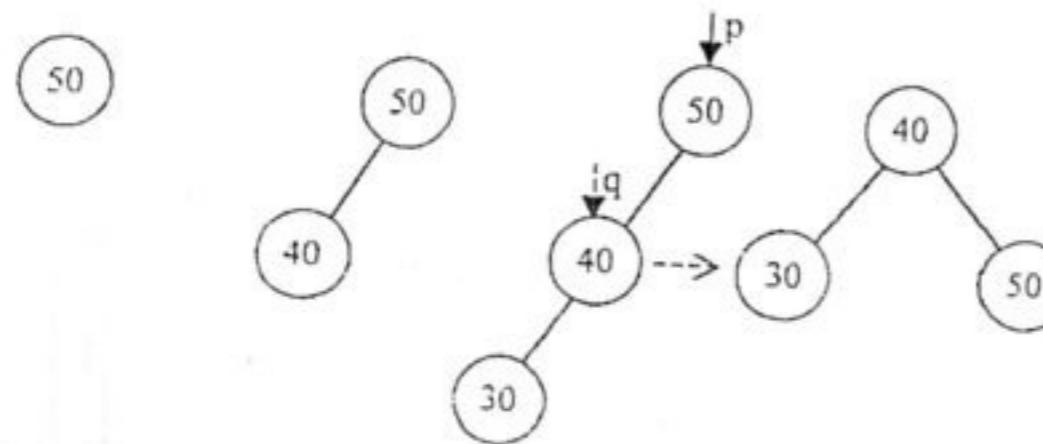
```

```

begin
  p^.contor:=p^.contor + 1; h:=false
end{Search}

```

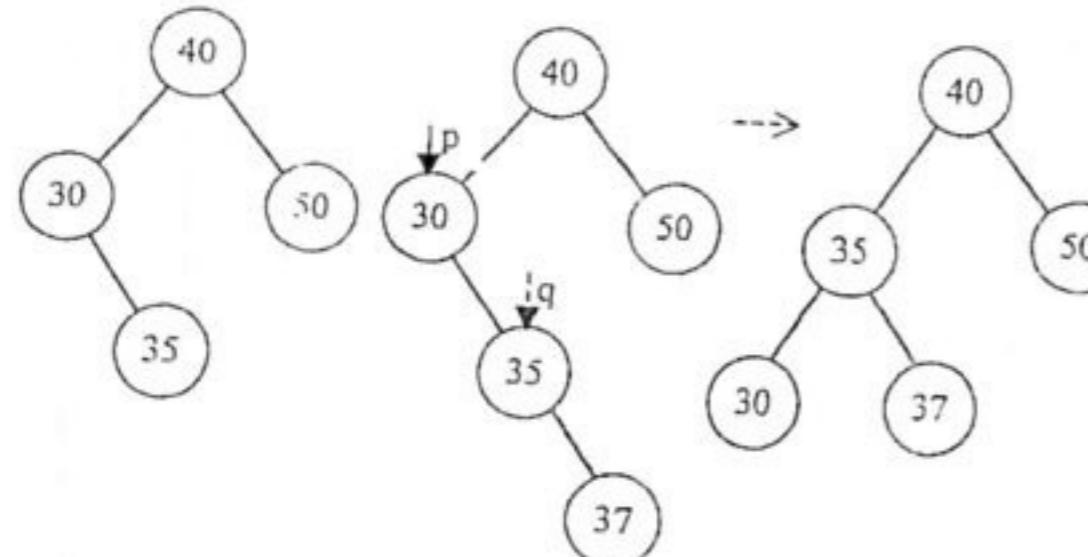
Ilustrăm în continuare construcția unui arbore binar de căutare echilibrat AVL prin inserări repetitive de chei. Vom inseră pe rând, cu reechilibrare dacă e cazul, cheile: 50, 40, 30, 35, 37, 39, 45, 55, 42.



Se inserează 50

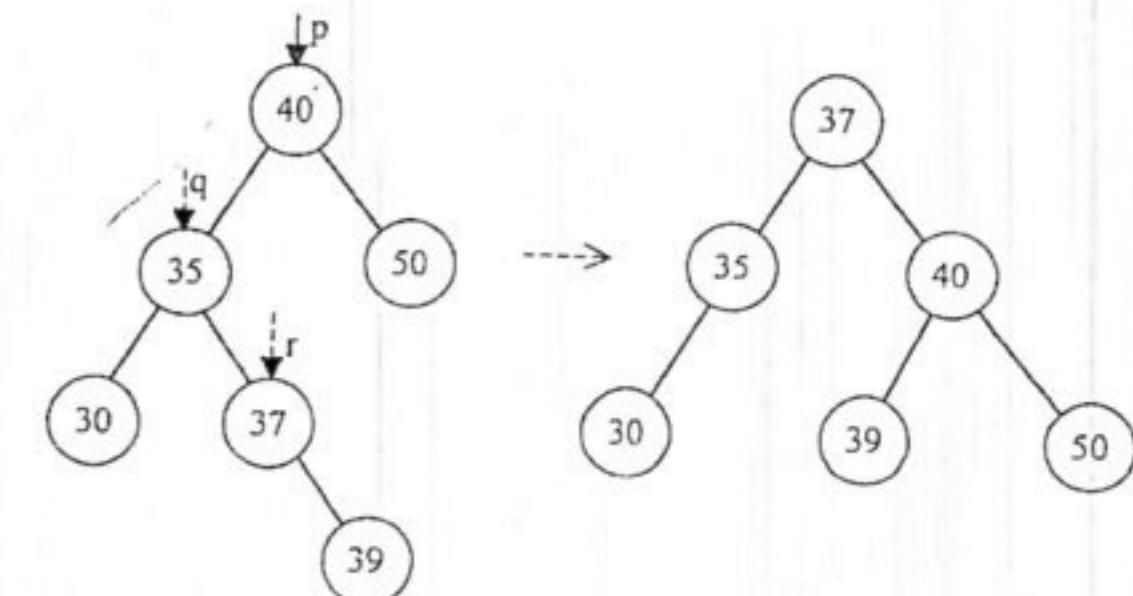
Se inserează 40

Se inserează 30. E nevoie de
reechilibrare în 50. Rotație SS

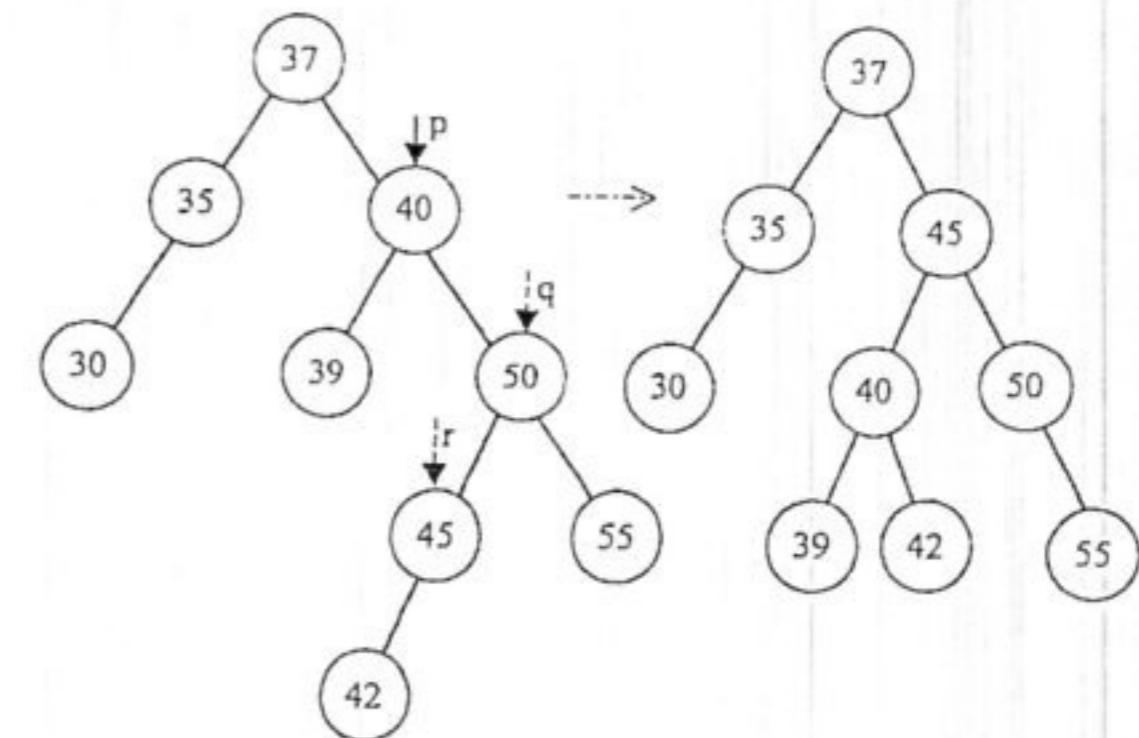


Se inserează 35

Se inserează 37. E nevoie de reechilibrare
în 30. Rotație DD.



Se inserează 39. E nevoie de reechilibrare
în 40. Rotație SD.



Se inserează 45 și 55. Apoi se inserează 42. E nevoie de
reechilibrare în 40. Rotație DS.

Capitolul III Sortări interne

Ne ocupăm în acest capitol de prezentarea unor algoritmi de sortare internă. Cu excepția algoritmului de sortare prin interclasare, toți algoritmii de sortare de vectori din acest capitol au două caracteristici comune esențiale care îi grupează într-o clasă:

(1) sortările se fac *in situ*, adică în același locații în care a fost dat vectorul inițial. Din acest motiv algoritmul folosește un număr minim de locații suplimentare (câte una pentru cei trei algoritmi direcți și pentru sortarea cu ansamble, și ceva mai multe pentru sortarea Shell și pentru Quick Sort)

(2) algoritmii se bazează pe comparații între chei, $a[i] < a[j]$.

Printre algoritmii care nu fac parte din această clasă se află algoritmul *Bucket Sort* (care stă la baza sortării lexicografice), versiunea lui mai simplă, sortarea prin numărare, sortarea prin interclasare, s.a.

Prezentăm întâi trei algoritmi *direcți* de sortare: prin inserție, prin selecția minimului (sau maximului) și prin interschimbare. Analizăm asemănările și deosebirile dintre ei și le evaluăm complexitatea în funcție de numărul de comparații și numărul de mutări. Toți trei intră în clasa de complexitate $O(n^2)$. Prezentăm și câteva îmbunătățiri ale lor, dar îmbunătățiri care nu ne scot din această clasă de complexitate.

Următorii trei algoritmi prezentați sunt mai performanți. Fiecare dintre ei se bazează pe unul dintre algoritmii direcți. Sortarea Shell aplică ideea inserției ajungând la performanța $O(n^{1.5})$. Sortarea cu ansamblu se bazează pe selecția minimului/maximului folosind pentru aceasta o structură arborescentă (arborele binar parțial ordonat și complet pe niveluri - ansamblul) de pe care extragerea minimului se efectuează în timp $O(1)$. Algoritmul HeapSort ajunge astfel la performanța $O(n \log_2 n)$. Sortarea rapidă (QuickSort) se bazează pe interschimbări și este de tip Divide et Impera împărțind tot timpul vectorul de sortat în subvectori care pot fi sortați independent. Cu prețul unui spațiu în plus, ajunge și el în cazul mediu la performanța $O(n \log_2 n)$.

Ultima secțiune prezintă operația de interclasare a doi vectori ordonați. Este primul exemplu de operație de combinare a două structuri. Spre deosebire de algoritmii din clasa precedentă, sortarea bazată pe interclasare folosește spațiu suplimentar de dimensiunea spațiului pentru datele de intrare, iar operațiile de bază sunt mutările. Materialul pregătește secțiunea 5 a capitolului următor.

3.1. Sortarea prin inserție directă

La primul pas iterativ al algoritmului se ia componenta $A[2]$ și se inserează la locul ei în vectorul sortat de lungime 1, $A[1]$, producând un vector sortat de lungime 2.

În general, la pasul iterativ i al algoritmului, vectorul este împărțit în două: bucata $A[1], \dots, A[i]$ este sortată crescător și se numește *destinație*, iar $A[i+1], \dots, A[n]$ este încă nesortată și se va numi *sursă*. Se ia prima componentă din sursă, $A[i+1]$, și se caută să se insereze la locul ei în destinație. Căutarea locului lui $A[i+1]$ în destinație se face parcurgând destinația de la dreapta la stânga și mutând pe rând câte o poziție la dreapta componentele care sunt mai mari decât valoarea de inserat, până când găsim locul valorii $x = A[i+1]$ și o inserăm. După executarea acestei operații elementare numită *pasă*, dimensiunea destinației crește cu 1, iar a sursei scade cu 1. În felul acesta, după executarea a $n-1$ pași iterativi, întregul vector $A[1..n]$ va fi sortat.

Algoritmul se scrie în felul următor:

```

procedure InsDir (A)
for i:= 2 to n do
    x:= A[i];
    {se caută locul valorii x în destinație}
    j:= i- 1;
    while (j > 0) and (x < A[j]) do
        A[j+1]:= A[j]
        j:= j- 1
    endwhile
    {inserarea lui x la locul lui}
    A[j+1]:= x
endfor
endproc.

```

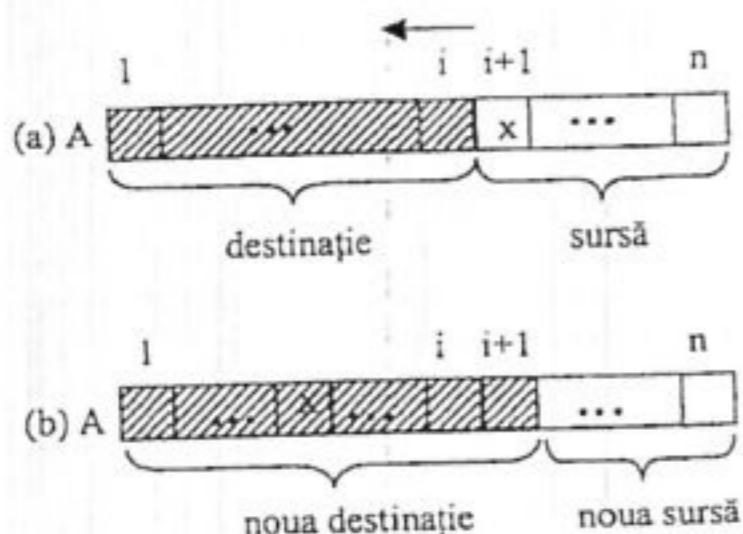


Fig.3.1.1. La pasul i al algoritmului prin inserare directă, pornim cu o destinație de lungime i . Se inserează x de pe prima componentă a sursei la locul lui în destinație, crescând dimensiunea destinației cu 1.

Ne reamintim de la căutarea secvențială că se poate elibera testul de nedepășire al dimensiunii prin introducerea unei componente marcată la sfârșit și plasarea valorii căutate pe această componentă marcată. Vom introduce componenta $A[0]$ pe care o vom folosi drept componentă marcată pentru pasele ce parcurg destinația. Programul modificat arată în felul următor:

```

procedure InsDir1(A)
for i:= 2 to n do
  {se introduce valoarea în componentă marcată}
  A[0]:= A[i];
  {se căută locul valorii în destinație}
  j:= i-1;
  while (A[0] < A[j]) do
    A[j+1]:= A[j]
    j:= j-1
  endwhile
  A[j+1]:= A[0]
endfor
endproc.

```

Cu utilizarea unei componente marcate, numărul de comparații dintr-o pasă de inserare s-a redus la jumătate.

Să mai observăm că această metodă de sortare este *stabilă*: cheile egale apar în vectorul sortat în aceeași ordine în care erau în vectorul inițial. Stabilitatea unei metode de sortare este o proprietate importantă mai ales când sortăm repetat după chei multiple.

Complexitatea sortării prin inserare directă

Vom estima complexitatea acestui algoritm direct de sortare internă, ca și pe algoritmilor direcți ce urmează, în funcție de numărul de comparații, C , și numărul de mutări de componente, M , estimări (dacă este posibil) la fiecare pas iterativ, căci comparațiile și mutările sunt operațiile de bază efectuate de acești algoritmi.

Să treacem acum la analiza sortării prin inserare directă. La fiecare pas iterativ i (cu $i=1, \dots, n-1$) al algoritmului, când se inseră componenta $A[i-1]$ la locul ei în destinație să notăm cu C_i numărul de comparații efectuat (presupunem că lucrăm pe variante cu componentă marcată, deci avem căte o singură comparație ce controlează ciclul *while* care caută locul inserării efectuând și mutări). C_i poate atinge o valoare maximă, $i+1$, dacă valoarea de inserat este cea mai mică și va veni pe prima componentă a sursei; poate atinge o valoare minimă, și anume 1, dacă valoarea de inserat este cea mai mare și poziția ei va fi ultima din noua sursă, iar valoarea medie a lui C_i va fi $i/2$.

Dacă notăm cu M_i numărul de mutări efectuat de algoritm la pasul iterativ i , avem relația $M_i = C_i + 1$ între mutări și comparații. De fapt, mutările din interiorul ciclului *while* sunt în număr de $C_i - 1$, la care se adaugă cele două mutări din afara lui, plasarea valorii de inserat pe componentă marcată la începutul programului și plasarea ei la locul final la sfârșitul programului.

Tinând cont de toate acestea, putem estima numărul minim, maxim și mediu de comparații și mutări pe algoritm în ansamblu (avem $n-1$ pași iterativi) cu formulele:

$$C_{\min} = 1 + 1 + \dots + 1 = n-1 \\ \text{pas 1 pas 2 pas } n-1$$

Numărul minim total de comparații se atinge, dacă se atinge minimul la fiecare pas, lucru care se întâmplă dacă la fiecare pas valoarea de inserat este cea mai mare, deci când vectorul este deja ordonat crescător. Avem și:

$$M_{\min} = 2 \div 2 + \dots + 2 = 2(n-1)$$

pas 1 pas 2 pas n-1

pentru număr minim de mutări.

Numărul maxim de comparații și mutări pe ansamblul algoritmului se atinge dacă avem numărul maxim de comparații la fiecare pas iterativ, deci în situația în care, la fiecare pas, valoarea de inserat este cea mai mică, deci vectorul era sortat inițial descrescător.

Însumând pentru cei $n-1$ pași iterativi avem:

$$C_{\max} = 2 + 3 + \dots + n = \frac{n(n+1)}{2} - 1$$

$$M_{\max} = 3 + 4 + \dots + (n+1) = \frac{n(n+1)}{2} - 1 + (n-1) = \frac{n^2 + 3n - 4}{2}$$

Pentru numărul mediu de comparații și mutări avem:

$$C_{\text{mediu}} = \frac{1}{2}(2 + 3 + \dots + n) = \frac{1}{2}C_{\max} = \frac{n(n+1)}{4} - \frac{1}{2} = \frac{n^2 + n - 2}{4}$$

$$M_{\text{mediu}} = C_{\text{mediu}} + 2(n-1) = \frac{n^2 + n - 2}{4} + 2(n-1) = \frac{n^2 + 9n - 10}{4}$$

3.2. Sortarea prin selecție directă

Operația pe care se bazează acest algoritm este cea de căutare liniară a minimului dintr-un vector, urmată de plasarea acestui minim la locul lui, adică pe prima componentă. Este și ceea ce face algoritmul la primul pas iterativ.

În general, la pasul iterativ i al algoritmului vectorul este împărțit în doi subvectori: *destinația* $A[1..i-1]$ ce conține minime puse la locul lor de pașii anteriori, și *sursa* $A[i..n]$, pe care se efectuează o pasă, adică se caută secvențial minimul din subvectorul sursă și apoi se interschimbă cu componenta $A[i]$. În felul acesta dimensiunea destinației crește cu 1, iar a sursei scade corespunzător. După $n-1$ pași iterativi vectorul A va fi sortat crescător.

Un program care implementează acest algoritm este de exemplu următorul:

```
procedure SelDir(A)
for i := 1 to n-1 do
    {căutarea secvențială a minimului în A[i..n] }
    k := i; min := A[i]; {initializarea minimului}
    for j := i+1 to n do
        if A[j] < min then {se schimbă minimul local}
            k := j; min := A[j]
    endif
    endfor
    {schimbarea minimului cu A[i] }
    A[k] := A[i]
    A[i] := min
endfor
endproc
```

$\left\{ \begin{array}{l} i = 1 \\ j = i+1 \end{array} \right. \quad \left\{ \begin{array}{l} i = i+1 \\ j = n \end{array} \right. \quad \left\{ \begin{array}{l} k = i \\ min = A[i] \end{array} \right. \quad \left\{ \begin{array}{l} k = j \\ min = A[j] \end{array} \right. \quad \left\{ \begin{array}{l} A[k] = A[j] \\ A[i] = min \end{array} \right.$

Evident, se poate aplica același procedeu pentru selectarea prin căutare secvențială a maximului, urmată de plasarea maximului la locul său final în vector, caz în care destinația se construiește la capătul din dreapta al vectorului.

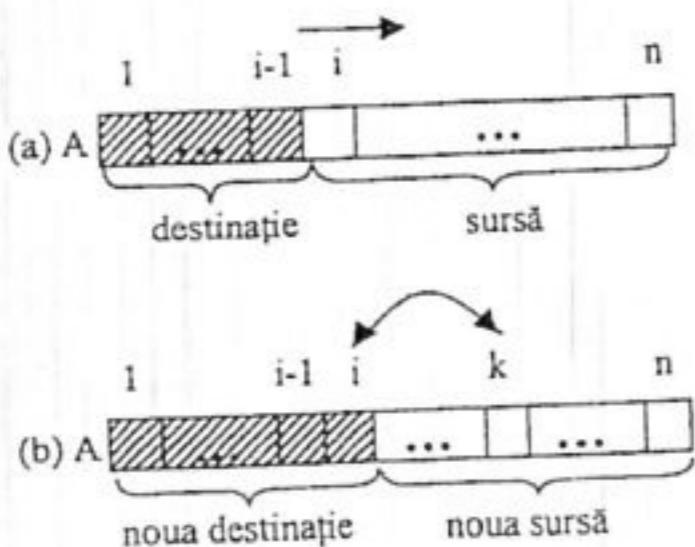


Fig. 3.2.1. Sortarea prin selecție directă a minimului. La pasul i se parcurge de la stânga la dreapta sursă, căutând minimul (a). După ce acesta s-a găsit pe componenta k se interschimbă $A[k]$ cu $A[i]$.

Sortarea prin selecție directă a minimului poate fi considerată într-un anume sens inversul sortării prin inserție directă. La fiecare pas, inserția directă ia un element din sursă și inspectează o parte din elementele din destinație, pe când selecția directă inspectează toate elementele din sursă și plasează unul în destinație.

Lungimea parcurgerii pe care o face inserția directă depinde de valoarea de inserat; ea va fi maximă, deci toate elementele destinației vor fi inspectate, dacă valoarea de inserat este mai mică decât toate valorile din destinație. În schimb, lungimea parcurgerii pe care o face selecția directă pe sursă este mereu aceeași, egală cu dimensiunea sursei. Nu avem nici o posibilitate de a evita aceasta, deoarece, până nu terminăm de parcurs sursa nu știm dacă minimul local este și minim global pentru sursă.

Complexitatea sortării prin selecție directă

La pasul iterativ i al algoritmului se caută secvențial minimul pe subvectorul $A[i..n]$, efectuându-se comparații cu toate elementele acestui subvector, comparații în număr de $n-i$. Să observăm că numărul de comparații este independent de ordinea inițială a componentelor, deci nu

sens, sortarea prin selecție directă se comportă mai puțin natural decât cea prin inserție directă. Însumând pentru cei $n-1$ pași iterativi ai algoritmului obținem numărul total de comparații:

$$C = (n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2}$$

Nu avem aceeași situație în cazul mutărilor, al căror număr depinde de ordinea componentelor. Pe de o parte, la fiecare pas iterativ i se fac cel puțin trei mutări (cele din afara ciclului *for* interior): $\min := A[i]$ la început, iar la sfârșit $A[k] := A[i]$ și $A[i] := \min$. Acesta este numărul minim de mutări și se atinge dacă în interiorul ciclului *for* nu se fac alte mutări, cu alte cuvinte dacă la parcurgerea subvectorului $A[i..n]$ nu se întâlnesc alte minime locale. Acest lucru se întâmplă când cheile inițiale sunt ordonate crescător, și deci $A[i]$ este minim pentru subvectorul $A[i..n]$, la fiecare pas i . Însumând pentru cei $n-1$ pași iterativi obținem:

$$M_{\min} = 3 + 3 + \dots + 3 = 3(n-1).$$

Cu cât întâlnim mai multe minime locale în $A[i..n]$ cu atât mai multe mutări va face algoritmul. Cazul cel mai nefavorabil este cel în care fiecare componentă întâlnită este noul minim local și se efectuează mutări, lucru care se întâmplă dacă cheile sunt date inițial în ordine descrescătoare. Atunci, numărul maxim de mutări la pasul i va fi și dat de $n-i-1-2=n-i-3$. Însumând pentru cei $n-1$ pași iterativi obținem:

$$\begin{aligned} M_{\max} &= [(n-1) + 3] + [(n-2) + 3] + \dots + [1 + 3] = \\ &= (n-1) + (n-2) + \dots + 1 + 3(n-1) = \frac{n(n-1)}{2} + 3(n-1). \end{aligned}$$

Pentru număr mediu de mutări dăm fără demonstrație următoarea estimare

$$M_{\text{mediu}} = n(\ln \gamma + e)$$

unde γ este constanta lui Euler ($\gamma = 0,577\dots$).

3.3. Sortarea prin interschimbare directă

Pentru această metodă de sortare directă operația de bază (o pasă) este următoarea: parcurgem vectorul de la dreapta la stânga comparând două elemente succesive $A[i-1]$ și $A[i]$; dacă $A[i-1] \leq A[i]$ le lăsăm pe loc, dacă $A[i-1] > A[i]$ le interschimbăm. După efectuarea unei asemenea pase pe întregul vectorul $A[1..n]$ vom avea pe componenta $A[1]$ minimul din vector. De ce? Pentru că minimul din vector e întâlnit la un moment dat de parcursere și interschimbat cu vecinul său; din acest motiv intră și în următoarea comparație, e din nou interschimbat, și așa mai departe. În felul acesta pasa îl impinge până la locul lui final, pe componenta $A[1]$. Putem acum relua operația pe $A[2..n]$ și, în urma efectuării unei pase, minimul din acest subvector va fi plasat pe componenta $A[2]$.

În general, la pasul iterativ i al algoritmului avem subvectorul $A[1..i-1]$ gata sortat și conținând cele mai mici $i-1$ valori din A . Acest subvector se va numi *destinație*. Efectuăm o pasă de la dreapta la stânga pe *sursă* $A[i..n]$, iar rezultatul va fi împingerea minimului din acest vector pe poziția $A[i]$. Crește în felul acesta dimensiunea destinației cu 1, iar cea a sursei scade corespunzător. După $n-1$ pași iterativi întregul vector este sortat. Algoritmul se mai numește sortare cu bule (Bubble Sort), denumirea fiind inspirată de analogia dintre bulele unei băuturi gazoase care se ridică la suprafață și elementele minime ale vectorului care "urcă" la locul lor.

```
procedure InterschDir(A)
for j:=2 to n do
    for i:=n downto j do
        if A[i-1] > A[i] then
            interschimbă (A[i-1], A[i])
        endif
    endfor
endfor
endproc.
```

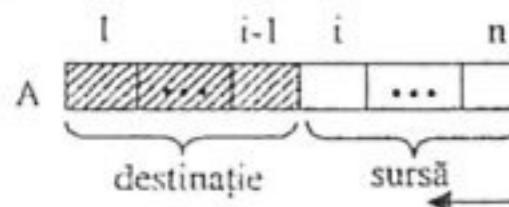


Fig. 3.3.1. După $i-1$ pași iterativi destinația are dimensiune $i-1$. Pasele se fac pe sursă de la dreapta la stânga.

Să observăm că, dacă facem pase de la stânga la dreapta la acest algoritm, atunci am pune maxime la locul lor final în vector în loc de minime. Cu alte cuvinte am construi destinația spre celălat capăt al vectorului.

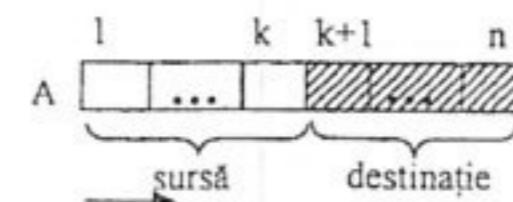


Fig. 3.3.2. Algoritmul de sortare prin interschimbare directă, cu pase de la stânga la dreapta, după $n-k$ pași iterativi.

Să observăm că, prin faptul că face pase pe sursă și nu pe destinație, cât și prin faptul că pune un minim (sau maxim) la locul lui, algoritmul seamănă cu cel de sortare prin selecție directă a minimului (respectiv a maximului). Însă, spre deosebire de acesta din urmă, care se mulțumește să plaseze minimul și lasă pe loc toate celelalte componente, cu excepția celui cu care minimul este interschimbat, algoritmul de interschimbare directă face și alte modificări în structura sursei, deplasând și maxime, câte o componentă, spre locul lor final. Această structură diferită a algoritmului ne permitem să-i aducem îmbunătățiri.

Una din îmbunătățiri se referă la posibilitatea reducerii numărului de pași iterativi, iar cealaltă la scurtarea lungimii paselor, ambele imposibil de efectuat la sortarea prin selecție directă. Le discutăm în continuare.

Observația pe care se bazează prima îmbunătățire a algoritmului este următoarea: dacă la o pasă nu se face nici o interschimbare, atunci sursa este sortată și, cum și destinația este, înseamnă că întregul vector A este sortat și este inutil să mai reluăm pasele. Structura algoritmului se schimbă. Vom menține o variabilă suplimentară, Sch , în care contorizăm interschimbările efective de la o pasă. Se trece la pasă următoare numai dacă la precedenta Sch a fost diferit de 0. Ciclul respectiv exterior *for* din programul anterior devine acum un *while* care depinde și de această variabilă.

```
procedure InterschDir1(A)
j:=2; {initializarea capătului din stânga al sursei}
Sch:= 1; {ca să intrăm în corpul while la prima pasă}
while (j ≤ n) and (Sch<>0) do
    Sch:= 0
```

```

for i:=n downto j do
    if A[i-1] > A[i] then
        interschimbă (A[i-1], A[i])
        Sch:=Sch+ 1
    endif
endfor
j:= j+ 1
endwhile
endproc

```

Să observăm că, la această versiune, ciclul *for* interior a rămas aproape neschimbat (cu excepția contorizării interschimbărilor). În orice caz, a rămas de aceeași lungime, fixă, care este lungimea sursei $A[j..n]$, și ea scade doar cu 1 de la un pas iterativ la altul.

O altă idee este să ținem minte, nu numai faptul că s-au făcut efectiv interschimbări, dar și locul unde s-a efectuat ultima interschimbare. Din acest loc, până la destinație, avem de-a face cu o bucată sortată, deci e suficient să reluăm pasă următoare de la n și până aici. Dacă ultima interschimbare s-a făcut între $A[k-1]$ și $A[k]$, atunci sursa pentru pasul următor va fi $A[k+1..n]$. Se schimbă deci și lungimea surselor. Codul care implementează și această modificare este:

```

procedure InterschDir2(A)
j:=2; {initializarea capătului din stg. al sursei }
while (j ≤ n ) do
    i:= n;
    while (i ≥ j ) do
        if A[i-1]> A[i] then
            interschimbă (A[i-1], A[i] )
            k:= i {ținem minte locul interschimbări}
        endif
        i:= i- 1
    endwhile
    j:= k+1; {noul capăt din stânga al sursei}
endwhile
endproc

```

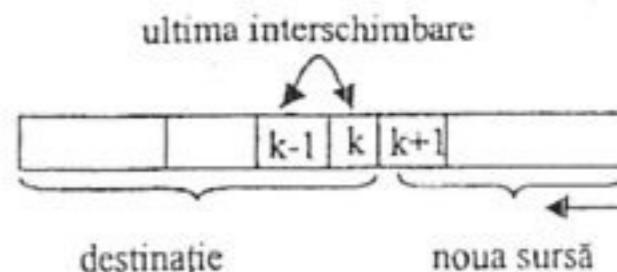


Fig. 3.3.3. Dimensiunea noii surse în funcție de locul ultimei interschimbări.

După cum observam mai devreme, pasele de la dreapta la stânga duc valorile minime la locul lor, iar valorile maxime avansează cu o poziție spre locul lor final. Pasele în sens invers, de la stânga la dreapta împing valori maxime la locul final și fac ca minimele să avanzeze cu o poziție. O idee este să alternăm direcția paselor. În felul acesta destinația se construiește la ambele extremități ale vectorului, din minime și maxime puse la locul lor. Sursa va fi $A[left..right]$ și, evident, vom ține cont și de îmbunătățirea precedentă care micșorează sursa în funcție de locul unde s-a făcut ultima interschimbare.

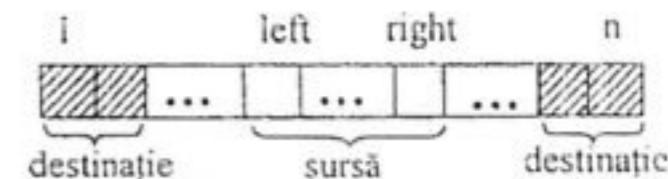


Fig. 3.3.4. La algoritmul ShakeSort, cu direcția alternantă a paselor, destinația se construiește la ambele capete ale vectorului.

```

procedure ShakeSort(A)
left:= 2; right:= n;{initializarea sursei}
k:= n;
repeat {atâtă timp cât avem sursă, adică left ≤ right}
    {pasă de la dreapta la stânga}
    for i:= right downto left do
        if A[i-1]> A[i] then
            interschimbă (A[i-1], A[i]);
            k:= i
        endif
    endfor
    left:= k+1; {redefinirea extremității stângi a sursei}

```

```

    {pasă de la stânga la dreapta}
    for i:= left to right do
        if A[i-1]> A[i] then
            interschimbă (A[i-1], A[i]);
            k:= i
        endif
    endfor
    right:= k-1; {redefinirea extreamei drepte a sursei}
until (left> right)
endproc.

```

Complexitatea sortării prin interschimbare directă

Ca și la sortarea prin selecție directă (și spre deosebire de inserția directă), numărul de comparații pe care-l face algoritmul la fiecare pas iterativ i este determinat de lungimea subvectorului sursă $A[i..n]$ pe care se face o pasă și este independent de ordinea cheilor. La pasul i algoritmul face $C_i = n-i$ comparații. Însumând pentru cei $n-1$ pași iterativi obținem:

$$C = (n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2}$$

În ceea ce privește numărul de mutări, acesta depinde de ordinea inițială a valorilor din vectori. Mutările sunt date de interschimbările între două componente alăturate, care se fac dacă relația de ordine dintre ele nu este cea dorită. Numărul minim de mutări la un pas iterativ este zero și se atinge dacă nu se face nici o interschimbare, lucru care se întâmplă dacă subvectorul sursă la acel pas iterativ este total ordonat. Dacă vectorul era inițial sortat crescător, atunci numărul minim de mutări se atinge la fiecare pas iterativ și (sumând pe 0 de $n-1$ ori) obținem numărul minim total de mutări

$$M_{min}=0.$$

Pe de altă parte, numărul maxim de mutări la pasul i se obține dacă se efectuează câte o interschimbare la fiecare comparație, lucru care se întâmplă dacă (făcând pase de la dreapta la stânga) primul element care intră în comparații, $A[i..n]$, este minimul de pe sursă $A[i..n]$ și locul lui final este pe componenta $A[i]$. Numărul maxim de mutări la pasul i va fi dat de $(n-i)*3$. Însumând după cei $n-1$ pași iterativi obținem:

$$M_{max} = (n-1)*3 + (n-2)*3 + \dots + 1*3 = \frac{3n(n-1)}{2}$$

Numărul mediu de mutări la pasul i se atinge dacă doar pentru jumătate din numărul de comparații facem interschimbări, deci va fi dat de formula $\lceil (n-1)/2 \rceil * 3$. Însumând pentru cei $n-1$ pași iterativi obținem:

$$M_{mediu} = \frac{(n-1)}{2} * 3 + \frac{(n-2)}{2} * 3 + \dots + \frac{1}{2} * 3 = \frac{3n(n-1)}{4}$$

Să observăm că la versiunea îmbunătățită ShakeSort, care schimbă și direcția paselor, se atinge un număr minim de comparații, $C_{min} = n-1$ în cazul în care vectorul este gata sortat crescător.

```

procedure ShellSort (A,h);
    for m:= 1 to t do
        k:= h[m]; {stabilim incrementul}
        s:= -k
        for i:= k+1 to n do
            x:= A[i];
            j:= i-k;
            if s = 0 then s:= -k
            endif
            s:= s+1
            A[s]:= x
            while x < A[j] do
                A[j+k]:= A[j]
                j:= j-k
            endwhile
            A[j+k]:= x
        endfor
    endfor
endproc

```

Knuth recomandă incremenți obținuți cu formulele recursive:

$$h_{k+1} = 3h_k + 1 \text{ și } h_1 = 1, t = [\log_3 n] - 1, \text{ avem } 1, 4, 13, 40, \dots$$

$$h_{k+1} = 2h_k + 1 \text{ și } h_1 = 1, t = [\log_2 n] - 1, \text{ avem } 1, 3, 15, 31, \dots$$

Există o estimare a complexității acestui algoritm care-l plasează în clasa $O(n^{1.2})$ din punct de vedere al numărului de comparații, deci al timpului. Din punct de vedere al spațiului, am văzut că el necesită $h[1]$ locații suplimentare pentru componentele marcaj, cu ajutorul cărora eliminăm teste de nedepășire a dimensiunii, teste care ar dubla numărul de comparații.

3.4. Sortarea prin inserție cu mășorarea incrementului (Sortare Shell)

În 1959 D.L. Shell a propus un algoritm de sortare bazat pe metoda prin inserție directă, algoritm cu o performanță îmbunătățită deoarece face comparații între chei mai distanțate din vector. Algoritmul sortează mai întâi prin inserție subvectori obținuți din vectorul inițial prin extragerea componentelor aflate la o distanță h una de cealaltă, distanță care se numește increment. Repetând procedeul pentru incremenți din ce în ce mai mici și, în final, pentru incrementul 1, se obține vectorul sortat.

Mai precis, se consideră un sir descrescător de numere naturale, numite incremenți dintre care ultimul, h_t , este 1:

$$h_1 > h_2 > \dots > h_t > h_{t+1} > \dots > h_l = 1.$$

Algoritmul are următoarea structură:

- (1) Se pornește cu incrementul h_1 .
- (2) La pasul iterativ m se consideră incrementul h_m . Se sortează prin inserție directă subvectorii obținuți din vectorul inițial luând elemente aflate la distanța h_m , adică subvectorii:

$$A[1], A[h_m+1], A[2h_m+1], \dots$$

$$A[2], A[h_m+2], \dots$$

⋮

$$A[h_m], A[2h_m], \dots$$

Apoi se reia pasul (2) pentru incrementul h_{m+1} .

Deoarece ultimul increment este $h_t=1$, ultimul pas iterativ t se reduce la sortarea prin inserție directă, deci vectorul va fi sortat. Considerăm incremenții introdusi în componentele unui vector $h[1..t]$. Vom face sortările prin inserție directă folosind componente marcaj. Cum la pasul 1 vom avea de sortat $h[1]$ subvectori, vom avea de alocat spațiu în plus de dimensiune $h[1]$.

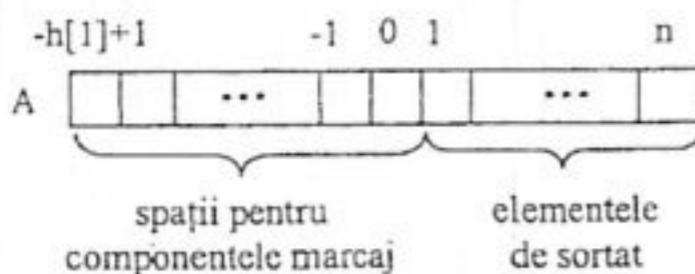


Fig. 3.4.1. Alocare de spațiu la sortarea Shell.

3.5. Sortarea prin selecție folosind structuri arborescente (sortarea cu ansamblu)

Reamintim că la sortarea prin selecție directă aveam de selectat la fiecare pas i , minimul din vectorul $A[i..n]$. Căutarea minimului se făcea secvențial, deci numărul de comparații era tot timpul maxim, independent de ordinea inițială a cheilor și egal cu lungimea vectorului - 1.

Sortarea prin selecție s-ar putea îmbunătăți dacă am avea o structură de date de pe care extragerea minimului (respectiv a maximului) să se facă rapid, dacă se poate chiar *optim*. Încă nu știm ce înseamnă optim, dar vom vedea că va fi de ordinul $O(\log n)$. Introducem în această secțiune o structură arborescentă care optimizează operația de extragere a maximului (sau minimului), și anume arboarele parțial ordonat și complet - ansamblu.

Arbore parțial ordonați și ansamble

Definiție. Se numește *arbore parțial max-ordonat* un arbore binar cu chei de un tip total ordonat și cu proprietatea că în oricare nod u al său avem relațiile:

$$\begin{aligned} \text{info}[u] &> \text{info}[\text{root}(\text{left}[u])], \text{ dacă } \text{left}[u] \text{ este nevid} \\ \text{info}[u] &> \text{info}[\text{root}(\text{right}[u])], \text{ dacă } \text{right}[u] \text{ este nevid.} \end{aligned}$$

Se vede imediat că putem caracteriza un arbore parțial max-ordonat și în modul următor: pentru orice nod u avem relațiile adevărate

$$\begin{aligned} \text{info}[u] &> \text{info}[v], \text{ pentru oricare nod } v \in \text{left}[u] \\ \text{info}[u] &> \text{info}[w], \text{ pentru oricare nod } w \in \text{right}[u]. \end{aligned}$$

O consecință imediată a definiției de mai sus este că într-un arbore parțial max-ordonat elementul cu cheia maximă se va afla în rădăcină.

Se poate da o definiție analoagă noțiunii de *arbore parțial min-ordonat*, și într-un asemenea arbore elementul cu cheia minimă se va afla în rădăcină.

Reamintim din capitolul II că se numește *arbore (binar) complet pe niveluri* un arbore (binar) care are toate nivelurile pline, eventual cu

excepția ultimului nivel, unde toate elementele sunt aliniate cel mai la stânga.

Arborii binari complezi pe niveluri au avantajul că pot fi reprezentați neambiguu cu ajutorul unui vector în care sunt menținute cheile arborelui date în ordinea parcurgerii lui în lățime: pe prima componentă rădăcina, apoi elementele de pe nivelul următor în ordine de la stânga la dreapta, și.a.m.d. Această reprezentare ne permite să accesăm din fiecare nod al arborelui atât fiul stâng și cel drept, cât și tatăl. Dacă A este vectorul care menține un asemenea arbore, atunci legăturile menționate sunt date de formulele simple:

$$\begin{aligned} A[k] &\text{ are ca fiu stâng pe } A[2*k] \\ A[k] &\text{ are ca fiu drept pe } A[2*k + 1] \\ \text{tatăl lui } A[k] &\text{ este } A[k \div 2]. \end{aligned}$$

De vreme ce avem și legături de tip "tată" însemnă că vom putea face, pe lângă drumuri de la rădăcină în jos în arbore și drumuri de la un nod către rădăcina arborelui, lucru pe care îl vom exploata în cele ce urmează. În plus, rădăcina se află pe componenta $A[1]$.

Să combinăm acum cele două proprietăți ale arborilor binari prezentate mai sus.

Definiție. Se numește *ansamblu* (mai precis *max-ansamblu*) un arbore binar max-ordonat și complet pe niveluri, reprezentat ca vector. (Denumirea în limba engleză este *Heap*.)

Analog se poate da definiția noțiunii de *min-ansamblu*.

În cele ce urmează ne vom ocupa de studiul structurii de max-ansamblu, cu operațiile specifice lui: inserarea, pe care se bazează și operația de creare, și mai ales operația de extragere a maximului numită și decapitare, deoarece, maximul aflându-se în rădăcină, extragerea lui revine la extragerea rădăcinii.

Inserarea într-un ansamblu

Algoritmul de inserare al unui nod nou într-un ansamblu are următoarea structură simplă:

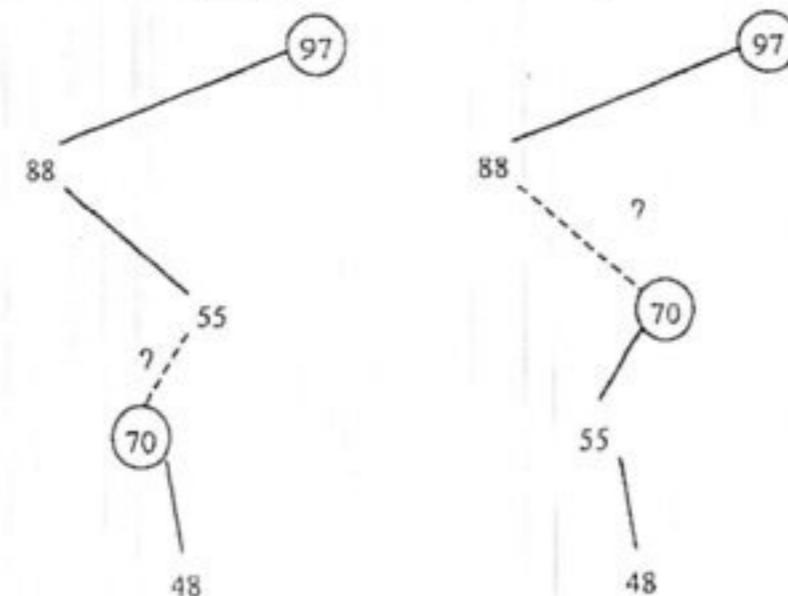
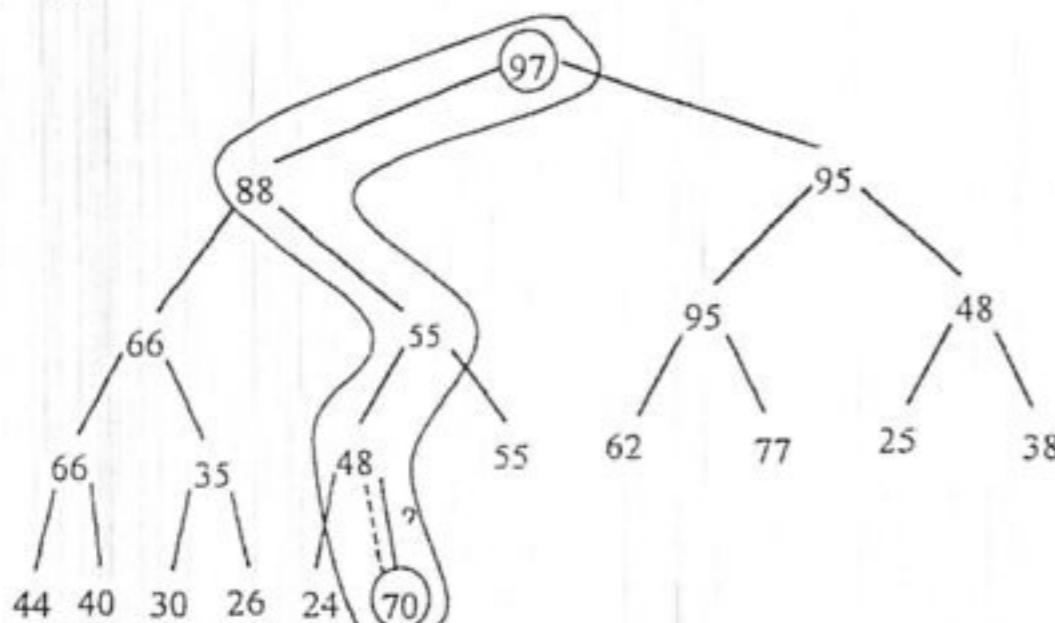
(1) Se pune nodul de inserat (*nod*) pe ultimul nivel al arborelui, aliniat cel mai la stânga (în felul acesta arborele rămâne complet);

(2) Se repetă (eventual până la rădăcină) comparația între $\text{info}[\text{nod}]$ și $\text{info}[\text{tata}[\text{nod}]]$

(2a) dacă $\text{info}[\text{nod}] < \text{info}[\text{tata}[\text{nod}]]$ atunci am terminat, căci am găsit locul lui *nod* în ansamblu (noua cheie nu violează condiția de arbore max-ordonat)

(2b) dacă nu, interschimb *nod* cu *tata[nod]* și reluăm de la (2a). După cum se vede, inserarea implică o parcurgere, eventual incompletă, a unui drum de la ultimul nivel până la rădăcină, în care numărul de comparații va fi proporțional cu lungimea drumului. Numărul de mutări poate fi făcut aproape egal cu numărul de comparații făcând inserarea propriu-zisă a cheii abia după ce i-am găsit locul final.

Ilustrăm în figurile următoare inserarea unei chei cu valoarea 70 într-un ansamblu, punând în evidență doar ramura pe care cheia își cauță locul. La comparația cu cheia 88 nu se mai continuă drumul în sus, cheia 70 și-a găsit locul final.



O procedură care implementează algoritmul de inserare al valorii *Val* în ansamblul *Ans[1..n]* este prezentată în continuare. Ea parcurge cu un indice curent *p* o ramură în sus, căutând locul lui *Val*.

```

procedure InsHeap(Ans, n, Val)
{în ansamblul Ans[1..n] se inserează Val}

n:=n+1 {crește dimensiunea ansamblului cu 1}
p:=n {p = indice pentru nodul curent; se inițializează}
while p>1 do {cât timp nu am ajuns la rădăcină}
    Tata:=p div 2
    if Val<=Ans[Tata] then
        Ans[p]:=Val {se inserează} {cazul (a)}
        exit {am terminat}
    else
        Ans[p]:=Ans[Tata] {se coboară tatăl în locul fiului}
        p:=Tata {nodul curent se reactualizează}
    endif
endwhile
Ans[1]:=Val {inserarea în rădăcină} {cazul (b)}
endproc

```

Observație: Inserarea propriu-zisă a valorii *Val* se face doar după ce i-am găsit locul, fie într-un nod interior (cazul (a)), fie în rădăcină (cazul (b)).

Construirea unui ansamblu. Asamblarea.

Operația de construcție a unui ansamblu se face, ca toate operațiile pe mulțimi dinamice discutate până acum, prin inserări repetitive.

- dacă avem cel puțin o cheie, atunci ea se va insera în rădăcină (căci un arbore cu un nod este ansamblu)
- pentru fiecare valoare nouă se apelează algoritmul de inserare *InsHeap*

O operație specifică este construirea unui ansamblu din inserări repetitive de chei care se află deja în locațiile unui vector. Ea se numește *asamblare* și este implementată de algoritmul următor:

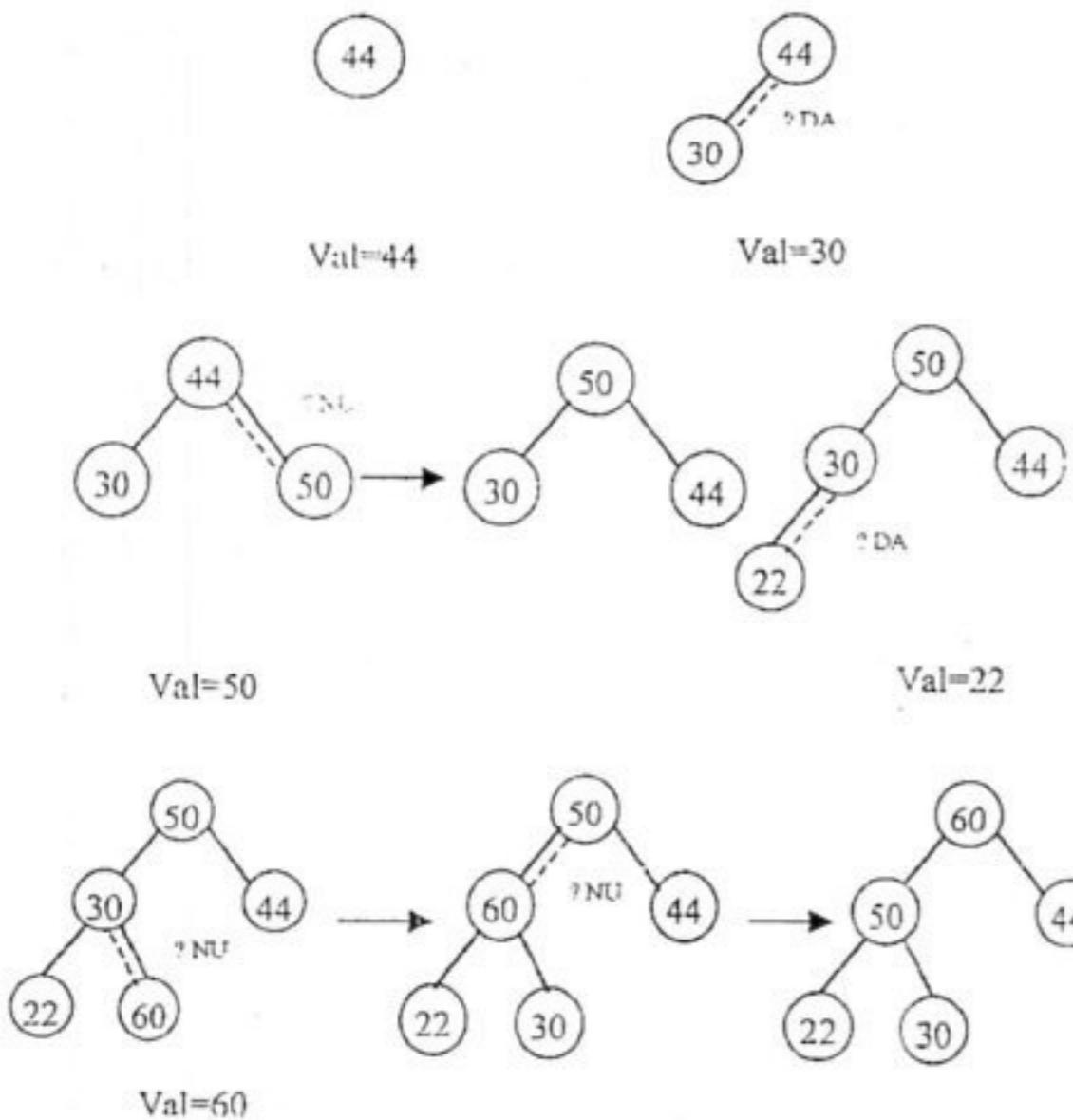
- *A[1]* este ansamblu
- la fiecare pas iterativ *j* se apelează procedura de inserare în ansamblul *A[1..j]* a valorii *A[j+1]*, pentru *j* = 1, ..., *n*-1.

```

procedure Asamblare (A, n)
    for j:=1 to n-1 do
        InsHeap (A, j, A[j+1])
    endfor
endproc

```

Ilustrăm în continuare construirea unui ansamblu din următoarele chei citite și inserate pe rând: 44, 30, 50, 22, 60. Ultima valoare inserată 60 trebuie să urce până la rădăcină.



Extragerea maximului sau decapitarea unui ansamblu

Evident, pe lîngă extragerea propriu-zisă, ne interesează refacerea structurii de ansamblu. Algoritmul de extragere are structura:

- (1) Se extrage valoarea din rădăcină (în vederea unei procesări)
- (2) Se înlocuiește rădăcina cu ultimul nod (arborele binar rămâne complet, dar eventual nu mai e max-ordonat)
- (3) Se "coboară" noua rădăcină la locul ei prin comparații cu cel mai mare dintre fii.

```
procedure DelHeap (Ans, n, Val)
```

```

Val:=Ans[1] {extragerea rădăcinii}
{Last=ultimul element din ansamblu ce trebuie inserat în rădăcină, apoi
trebuie să căutăm locul lui}
Last:=Ans[n]
n:=n-1 {scade dimensiunea ansamblului}
{căutăm locul lui Last în ansamblu, cu p indice nod curent, iar l, r indici
pentru fiu stâng, respectiv drept}
p:=1; l:=2; r:=3 {initializarea indicilor curenți de la rădăcină}
while (r<=n) do {test de nedepășire a dimensiunii ansamblului}
    if (Last>=Ans[l]) and (Last>=Ans[r]) then
        Ans[p]:=Last {inserarea}
        exit {am terminat}
    endif
    if Ans[l]>=Ans[r] then {continuăm pe ramura stângă}
        Ans[p]:=Ans[l]
        p:=l {reactualizarea lui p}
    else {continuăm pe ramura dreaptă}
        Ans[p]:=Ans[r]
        p:=r {reactualizarea lui p}
    endif
    l:=2*p; r:=l+1; {reactualizarea filor lui p}
endwhile
if (l=n) and (Last<Ans[l]) then
    Ans[p]:=Ans[l]
    p:=l
endif
Ans[p]:=Last {inserarea propriu-zisă a lui Last la locul lui}
endproc {DelHeap}

```

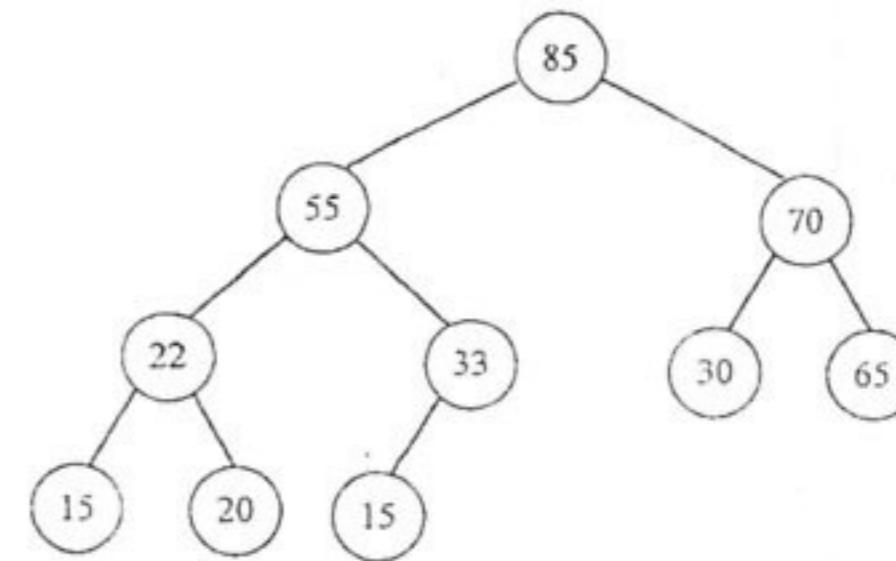
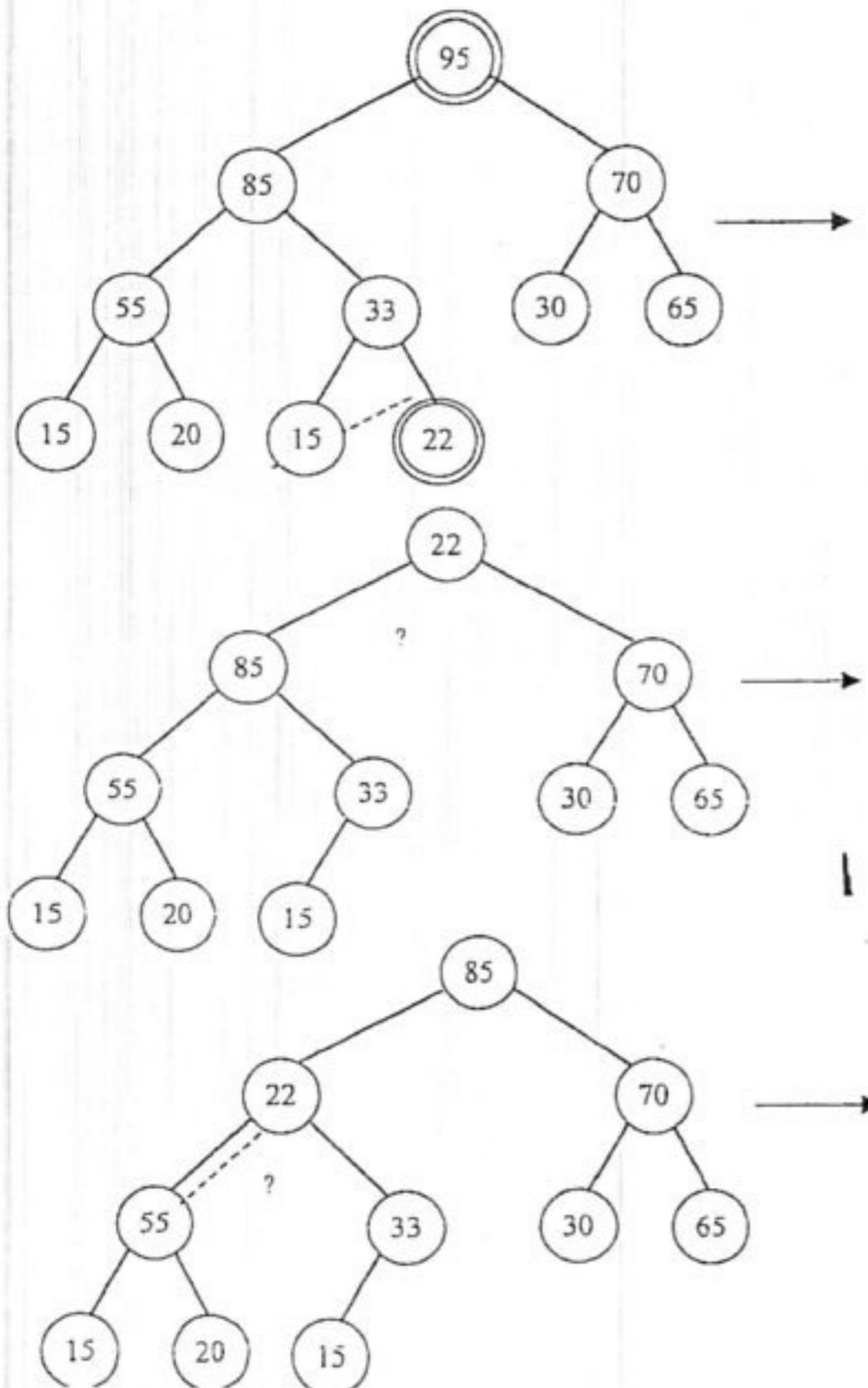


Fig.3.5.1. Decapitarea unui ansamblu. Se extrage maximul 95 din rădăcină și se plasează ultima valoare, 22, în locul lui. Se coboară 22 la locul final în ansamblu prin comparații cu cel mai mare dintre fiți.

Complexitatea operațiilor într-un ansamblu

Atât operația de inserare, cât și operația de decapitare implică parcurgerea cu comparații a unei ramuri din arbore, deci fac, chiar în cazul cel mai nefavorabil, un număr de comparații egal cu adâncimea arborelui. Pentru un arbore binar complet pe niveluri cu n noduri adâncimea este $\lceil \log_2 n \rceil$. Deci complexitatea algoritmului de inserare în ansamblu și al celui de extragere de maxim cu reasamblare este de ordinul $O(\log_2 n)$.

Sortarea cu ansamble

Se poate folosi structura de ansamblu (împreună cu operațiile pe ansamblu) pentru a scrie un algoritm de sortare internă extrem de performant, *HeapSort*. Structura lui este următoarea:

- (pasul 0) Se asamblează vectorul $A[1..n]$. Maximul va plasat pe componenta $A[1]$.
- (pasul 1) Se decapitează ansamblul $A[1..n]$, cu reasamblarea lui $A[1..n-1]$ și se pune maximul extras pe componenta $A[n]$.

(pasul j) Se decapitează ansamblul $A[1..n-j+1]$, cu reasamblarea lui $A[1..n-j]$ și se pune maximul extras pe componenta $A[n-j+1]$.

După $n-1$ pași iterativi vectorul A este sortat.

Procedura care implementează acest algoritm se scrie foarte simplu în felul următor:

```
procedure HeapSort (A, n) {sortarea cu ansamble}
{sortăm vectorul A[1..n]}
    Asamblare (A, n)
    while n>1 do
        DelHeap (A, n, Val)
        A[n+1]:=Val
    endwhile
endproc
```

Complexitatea algoritmului de sortare cu ansamble

Pentru fiecare din cele n componente ale vectorului avem de făcut o operație de reasamblare al cărei cost în funcție de numărul de comparații este de ordinul lui $\log_2 n$. Deci costul total al algoritmului este de ordinul $O(n \log_2 n)$, performanță care se atinge atât în cazul mediu cât și în cazul cel mai nefavorabil.

În ceea ce privește spațiul, algoritmul nu necesită decât o locație în plus, exact ca și algoritmii direcți de sortare.

Exerciții

- Definiția dată max-ansamblului nu permite existența cheilor multiple. Dați o nouă definiție care să permită aceasta.
- Este sortarea cu ansamble o metodă stabilă de sortare? Justificați răspunsul.
- Scrieți un algoritm de sortare bazat pe structura de min-ansamblu.

3.6. Sortarea prin interschimbare folosind partiții (QuickSort)

Vom discuta acum un algoritm de sortare bazat pe operația de interschimbare. Am dorit să îmbunătățim performanța sortării prin interschimbare directă, comparând și interschimbând între ele, dacă e cazul, componente aflate la distanțe mai mari în vector. Algoritmul care urmează i se datorează lui C. A. R. Hoare și poartă numele de sortare rapidă (Quick Sort).

Operația de bază a acestui algoritm se numește operația de *partitionare* a unui vector $A[1..n]$ cu ajutorul unei valori numită *pivot* și este descrisă în continuare.

Algoritmul de partitionare al vectorului $A[1..n]$

- Se alege o valoare x din vectorul $A[1..n]$
- (a) Se parcurge de la stânga la dreapta vectorul lăsând pe loc componentele cu proprietatea $A[i] < x$, până la primul indice i pentru care $A[i] \geq x$.
- (b) Se parcurge de la dreapta la stânga vectorul, lăsând pe loc componentele cu proprietatea $A[j] > x$, până la primul indice j pentru care $A[j] \leq x$.
- Dacă $i < j$ atunci se interschimbă $A[i]$ cu $A[j]$ și avansează indicii pentru parcurgeri ($i:=i+1; j:=j-1$).

Se repetă pașii 2 și 3 până când cele două parcurgeri se întâlnesc, adică $i = j$.

În urma aplicării algoritmului de mai sus vectorul A este acum împărțit în doi subvectori: unul conține valori mai mici decât valoarea pivot x , iar celălalt doar valori mai mari decât x . Mai precis, avem:

$$A[k] \leq x, \text{ pentru orice } k \in [1..i-1]$$

$$A[k] \geq x, \text{ pentru orice } k \in [j+1..n]$$

și, de asemenea, și relațiile care rezultă din cele precedente:

$$A[k] = x, \text{ pentru orice } k \in [j+1..i-1].$$

Presupunem dată o procedură care interschimbă două valori date, u și v :

```
procedure Interschimbă(u,v);
```

```
    var w:{același tip ca u și v}{locația suplimentară}
```

```
begin
```

```
    w:=u; u:=v; v:=w;
```

```
end
```

126

Procedura de partitie care implementeaza algoritmul de partitie descris mai sus pe subvectorul $A[Left..Right]$ este urmatoarea:

```

procedure Partition(Left, Right: indici);
{ iLeft=indicele curent pentru parcurgerea (2a)}
{ iRight=indicele curent pentru parcurgerea (2b)}
{ initializarea indicilor curenți pentru parcurgeri}
iLeft:=Left; iRight:=Right;
{alegerea pivotului în poziție mediană}
x:=A[(Left+Right)div 2];
{partitia}
repeat
(2a) while A[iLeft] < x do
    iLeft:= iLeft+1;
    endwhile
(2b) while A[iRight] > x do
    iRight:= iRight - 1;
    endwhile
(3) if iLeft <= iRight then
    Interschimbă(A[iLeft], A[iRight]);
    iLeft:= iLeft+1;
    iRight:= iRight-1;
endif
until (iLeft > iRight)
endproc{Partition}
```

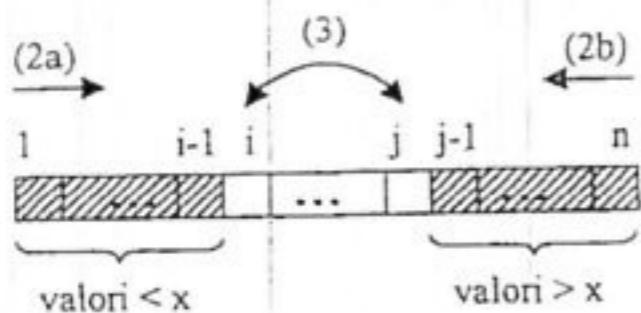


Fig.3.6.1. Procedura de partitie.

Deoarece valoarea pivot x a actionat ca un separator, putem acum să sortăm *in situ* în mod independent cei doi subvectori obținuți prin partitie, căci valorile lor nu se mai amestecă, iar concatenarea

rezultatelor va duce la obținerea vectorului A sortat. Ce metodă să aplicăm pe subvectorii obținuți prin partitie? Putem aplica fiecărui procedura de partitie, apoi încă odată subvectorilor rezultati, și aşa mai departe, până ajungem la subvectori de lungime 1 care sunt gata sortați. Aceasta este ideea algoritmului *QuickSort*, care folosește partitioarea pentru a sorta.

Vom avea mai multe versiuni ale algoritmului de sortare rapidă în funcție de alegerea pivotului și în funcție de metoda de prelucrare a subvectorilor rezultați din partitii. În ceea ce privește alegerea valorii pivot, putem opta între: valoarea aflată pe componenta mediană, o valoare aleasă în mod aleator de pe una din componente, sau o valoare aflată la una din extremitățile vectorului. Prezentăm deocamdată algoritmul bazat pe alegerea valorii mediene ca pivot. Schimbări semnificative în procedura de partitie apar doar la alegerea pivotului la o extremitate.

In ceea ce privește modul de prelucrare al subvectorilor obținuți din partitii avem de optat între: (a) a folosi recursia, făcând căte un apel pentru fiecare subvector rezultat; (b) o versiune iterativă, în care să menținem într-o stivă capetele subvectorilor ce rămân de procesat; (c) combinații între prelucrări iterative și prelucrări cu apeluri recursive. Alegerea unui tip de prelucrare sau al altuia depinde de considerente legate de spațiu, căci stiva, fie cea gestionată explicit în program, cât și cea implicită gestionată de recursie ocupă spațiu suplimentar, o caracteristică a acestui tip de sortare ce trebuie luată în considerare.

Dăm în continuare o procedură *QuickSortRec* care implementează algoritmul de sortare rapidă. Ea utilizează procedura recursivă *QSortRec* care face partitie pe subvectorul $A[Left..Right]$ și apoi folosește apeluri recursive pentru prelucrarea subintervalelor rezultante. În corpul principal al procedurii *QuickSortRec* avem un singur apel, *QSortRec(l,n)*.

```

procedure QuickSortRec(A);
procedure QSortRec(Left, Right: indici)
{partitia (codul din corpul procedurii Partition)}
i:=Left; j:=Right;
x:=A[(Left+Right)div 2];
repeat
    while A[i] < x do i:= i+1;
    while A[j] > x do j:= j - 1;
    if i <= j then
        Interschimbă(A[i], A[j]);
    endif
    i:= i+1;
    j:= j-1;
endrepeat
endproc{QuickSortRec}
```

```

        i:= i+1;
        j:= j-1;
    endif
until (i > j);
{apeluri recursive pentru subvectorii rezultați}
if Left < j then QSortRec(Left, j);
if i < Right then QSortRec(i, Right);
endproc{QSortRec}
begin {corpul principal al lui QuickSortRec(A)}
    QSortRec(1,n)
endproc{ QuickSortRec }

```

Dăm în continuare, schematic, structura unei versiuni iterative a algoritmului de sortare rapidă. Presupunem dată o stivă *Stack*, capabilă să mențină perechi de capete de intervale, împreună cu procedurile *Push(Stack, l, r)*, care introduce perechea *(l, r)* în stiva *Stack* și *Pop(Stack, l, r)*, care extrage din stiva *Stack* perechea de valori *(l, r)*. Presupunem de asemenea că procedura de partitie este modularizată, adică, *Partition(Left, Right, iLeft, iRight)* face partitia pe subvectorul *A[Left..Right]* și returnează capetele subvectorilor ce rămân de procesat în variabilele *iLeft* și *iRight*.

```

procedure QuickSortIter(A);
    Push(Stack, 1,n); {initializarea stivei}
    repeat
        Pop(Stack, Left, Right);
        Partition(Left, Right, iLeft, iRight);
        {introducem în stivă capetele intervalelor ce rămân de prelucrat}
        if Left < iRight then
            Push(Stack, Left, iRight);
        endif
        if iLeft < Right then
            Push(Stack, iLeft, Right);
        endif
    until Stack = Ø
endproc {QuickSortIter}

```

Să observăm că la această versiune efectuăm multe operații în plus pe stivă: introducerea în stivă la sfârșitul ciclului *repeat*, unrată imediat, la reluarea ciclului, de extrageri din stivă. Am putea elibera aceste operații

în plus, prin introducerea a încă unui ciclu repetitiv, care să reia la prelucrare directă, o parte din intervale. Noua versiune iterativă, care are printre avantaje și pe acela al unei dimensiuni mai scăzute a stivei, arată în felul următor:

```

procedure QuickSortIter1(A);
    Push(Stack, 1, n);
    repeat
        Pop(Stack, Left, Right);
        repeat
            {noul ciclu repetitiv în care se aplică partitia pe A[Left..Right]}
            Partition(Left, Right, i, j);
            {introducem în stiva Stack intervalele din dreapta}
            if i < Right then
                Push(Stack, i, Right);
            endif
            {redefinim extrema dreaptă a intervalului din stânga care este
            reluat la prelucrare iterativă}
            Right := j;
        until Left >= Right
    until Stack = Ø
endproc {QuickSortIter1}

```

Exerciții

1. Să se scrie în cod o procedură *Partition(Left, Right, i, j)* care să poată fi apelată de procedurile *QsortRec* (în versiunea din text este dat codul pentru partitie), *QuickSortIter* și *QuickSortIter1*.
2. Să se modifice procedura *QuickSortIter1* astfel încât intervalele cele mai lungi să fie depuse în stivă, iar cele mai scurte să fie reluate la prelucrare directă.
3. Să se scrie în cod procedurile *QuickSortIter* și *QuickSortIter1*, nemodularizate, cu stiva *Stack* în alocare statică reprezentată printr-un vector cu componente de tip

```

type interval = record
    left, right: indici;
end

```

O altă alegere a pivotului pentru sortarea rapidă

Până acum am ales ca pivot valoarea mediană din vectorul pe care facem partitura. Ce se întâmplă dacă alegem o valoare situată la una din extremități?

Să presupunem că, pentru partitiorarea vectorului $A[Left..Right]$ alegem $x = A[Left]$. Atunci pasul (2a) din algoritmul de partitiorare (parcugerea de la stânga la dreapta a vectorului până la primul indice i pentru care $A[i] \geq x$) nu mai are sens, căci acest indice este chiar $Left$. Executăm (2b), adică parcugerea de la dreapta la stânga vectorul până la primul indice j pentru care $A[j] \leq x$.

Pasul (3) devine: dacă $Left < j$, atunci interschimbă $A[Left]$ cu $A[j]$. Observăm că valoarea pivot $x = A[Left]$ a participat la interschimbare, și se află acum plasată în extremitatea dreaptă a subvectorului $A[Left..j]$. Reluăm acum parcugerea de tip (2a) de la stânga la dreapta. Parcugerea de tip (2b) nu mai are sens. La sfârșit interschimbă. Observăm că valoarea pivot participă din nou la interschimbare. Procedeul continuă până când cele două parcugeri se întâlnesc, adică atât timp cât mai există componente în vector care nu au fost comparate cu pivotul. La sfârșit obținem valoarea pivot x plasată la locul ei final în vector. Fie loc indicele lui A ce va conține pe x . Avem atunci:

$$A[k] \leq x \text{ pentru orice } k \in [1..loc-1]$$

$$A[k] \geq x \text{ pentru orice } k \in [loc-1..n]$$

deci subintervalele ce trebuie procesate în continuare sunt $A[1..loc-1]$ și $A[loc-1..n]$.

Dăm mai jos noua procedură de partitura. La parcugeri ca va lăsa pe loc valorile egale cu pivotul. Indicele loc ține minte tot timpul componenta pe care se află valoarea pivot, iar la sfârșit o transmite procedurii apelante pentru a putea fi calculate noile capete ale subvectorilor rezultați.

```
procedure Partition2(Left, Right: indice, var loc: indice)
{loc este indicele pe care se va plasa în final valoarea  $x = A[Left]$ }
{initializarea indicilor  $i$  și  $j$  pentru parcugerile de la stânga la dreapta,
respectiv de la dreapta la stânga}
i := Left; j := Right;
loc := Left {initializarea indicelui loc}
while i < j do
    {parcugerea de la dreapta la stânga, urmată de interschimbare}
```

```
while ( $A[loc] \leq A[j]$ ) and ( $j \neq loc$ ) do
    j := j - 1
end while
if  $A[loc] > A[j]$  then
    Interschimbă( $A[loc], A[j]$ )
    loc := j
end if
{parcugerea de la stânga la dreapta, urmată de interschimbare}
while ( $A[i] \leq A[loc]$ ) and ( $i \neq loc$ ) do
    i := i + 1
end while
if  $A[i] > A[loc]$  then
    Interschimbă( $A[loc], A[i]$ )
    loc := i
end if
end while
end {Partition2}
```

O versiune recursivă a sortării rapide ce folosește procedura *Partition2*, analoagă lui *QSortRec*, este următoarea:

Procedure *QSortRec2*(Left, Right)

```
Partition2(Left, Right, loc);
if Left < (loc-1) then QSortRec2(Left, loc-1);
if (loc+1) < Right then QSortRec2(loc+1, Right)
endproc {QSortRec2}
```

Pentru a sorta vectorul A se face apelul *QSortRec2*(1, n).

Exerciții

1. Să se scrie versiunile iterative ale sortării rapide ce folosesc procedura *Partition2*.

Complexitatea sortării rapide

Vom evalua complexitatea acestui algoritm în funcție de timpul de rulare care depinde în mod esențial de numărul de comparații, și în funcție de necesitățile de spațiu în plus.

Pentru fiecare alegere a unei valori pivot x facem la procedura de partitiorare un număr de n comparații. Numărul acesta rămâne constant la

fiecare pasă a algoritmului. Mai departe performanța lui depinde de numărul de sub-intervale pe care le generează partitia.

Dacă fiecare partitie înjumătășește intervalul pe care se face (cazul cel mai favorabil), atunci vom avea un număr total de $\log_2 n$ subintervale de procesat, deci numărul total de comparații este de ordinul lui $O(n \log_2 n)$. Tot aceasta este performanța și în cazul mediu.

Puteam avea și cazul cel mai nefavorabil, când fiecare pivot ales x este maximul sau minimul din subinterval. Atunci, la fiecare partitie se generează un subinterval de lungime 1 și unul de lungime $n-1$, deci avem un număr total de n intervale de prelucrat. Numărul total de comparații va fi în acest caz de ordinul lui $O(n^2)$, exact ca și performanța algoritmilor direcți.

O altă caracteristică a acestui algoritm este faptul că necesită spațiu în plus pentru menținerea subintervalelor. Atât versiunile iterative, care gestionează explicit stiva ce menține intervalele, cât și cele recursive, au nevoie de acest spațiu suplimentar.

Aplicație a procedurii de partitie la găsirea medianei

Mediana unui vector este acea valoare dintre componente sale care este mai mare decât jumătatea din componente și mai mică decât cealaltă jumătate. Dacă am sortat întregul vector, mediana s-ar găsi la mijlocul vectorului. Vrem să găsim însă această valoare fără a sorta întregul vector.

Problema se generalizează la găsirea valorii a k -a dintr-un vector, cea care este mai mare decât $k-1$ componente și mai mică decât $n-k$ componente, pe scurt, cea care ar ocupa locația $A[k]$ în vectorul sortat, fără a sorta întregul vector.

C. A. R. Hoare a propus un algoritm care se bazează pe procedura de partitie a sortării rapide. Se aplică procedura de partitie pe întregul vector, deci $Left=1$ și $Right=n$, cu valoarea pivot $x = A[k]$. Indicii i și j cu care se termină partitia au proprietățile:

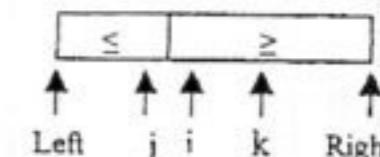
$$i > j \text{ (cele două parcurgeri s-au întâlnit)}$$

$$A[s] \leq x, \text{ pentru orice } s < i$$

$$A[s] \geq x, \text{ pentru orice } s > j.$$

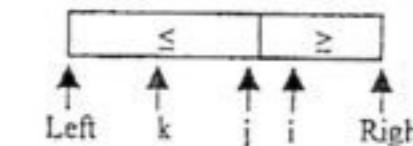
Aveam trei cazuri posibile pentru indicele k în raport cu indicii i și j :

(1) $j < i < k$



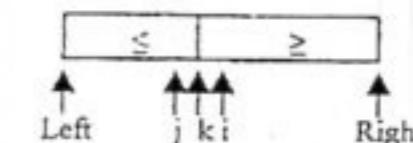
Limita unde se termină cele două partitii este în stânga lui k (din cauză că valoarea pivot x a fost mai mică decât valoarea căutată). Se reia partitie pe subvectorul $A[i..Right]$.

(2) $k < j < i$



Cazul simetric lui (1). Limita unde se termină partitiiile este în dreapta lui k . Se reia partitie pe subvectorul $A[Left..j]$.

(3) $j < k < i$



Valoarea de pe componenta k este cea căutată, deoarece acum în stânga avem $k-1$ valori mai mici decât ea, iar în dreapta $n-k$ valori mai mari.

Procedura de partitie se reia pe subintervalele ce conțin indicele k până când se ajunge în cazul (3). Procedura *Find* implementează algoritmul descris mai sus, presupunând că procedura *Partition* ($Left$, $Right$, i , j) este complet modularizată și funcționează ca cea descrisă la sortarea rapidă.

```

procedure Find (A: vector, k: indice);
    Left:=1; Right:= n;
    while Left< Right do
        x:= A[k];
        Partition (Left, Right, i, j);
        if j< k then Left:= i; {se continuă pe subvectorul din dreapta}
        if k< i then Right:= j; {se continuă pe subvectorul din stânga}
    endwhile
endproc {Find}

```

O modificare a algoritmului de mai sus se poate face în felul următor: în loc să aşteptăm ca cele două parcurgeri să se întâlnească ($i>j$), să vedem când una din parcurgeri depăşeşte indicele k și să reluăm partitia pe subintervalul corespunzător cu noua valoare pivot $x=A/k$. Mai precis:

1. dacă $k \leq i < j$ reluăm partitia pe $A[Left..j]$ cu noul pivot;
2. dacă $i < j \leq k$ reluăm partitia pe $A[i..Right]$ cu noul pivot;

Procedura *Find2* implementează această idee.

```

procedure Find2 (A: vector, k: indice)
    Left:= 1; Right:= n;
    while Left< Right do
        x:= A[k]
        i:= Left; j:= Right
        while (i<= k) and (k<= j) do
            while A[i]< x do i:= i+1;
            while x< A[j] do j:= j-1;
            if i<=j then
                Interschimbă (A[i], A[j]);
                i:= i+1; j:= j-1
            endif
        endwhile
        if (k< i) and (k> j) then Right:= j;
        if (j< k) and (k> i) then Left:= i;
    endwhile
endproc {Find2}

```

3.7. Interclasarea a două siruri ordonate. Sortarea prin interclasare.

Interclasarea a două siruri sortate

Se dau două siruri ordonate crescător $A[1..dimA]$ și $B[1..dimB]$. Ne punem problema să construim sirul $C[1..dimA + dimB]$ ordonat crescător ce conține toate elementele lui A și B . Este un prim exemplu de operație de combinare a două structuri: din două structuri de același tip (în cazul acesta structuri liniare și ordonate) producem o alta de același tip, care este "reuniunea" elementelor lor.

Algoritmul de interclasare are următoarea structură simplă: Se parcurg simultan sirurile A , B și C . La fiecare pas se compară cele două componente curente din A și B , iar cea mai mică dintre ele este mutată în C . Când s-a terminat una din surse, A sau B , componentele rămasă se adaugă la C (ce poartă denumirea de destinație). Procedura *Merging* ce-l implementează este dată în continuare:

```

procedure Merging(dimA, dimB, A, B, C)
    iA := 1; iB := 1; iC := 1; {initializarea indicilor pentru parcurgeri}
    while (iA ≤ dimA) and (iB ≤ dimB) do
        if A[iA] < B[iB] then
            C[iC] := A[iA]
            iA := iA + 1
        else C[iC] := B[iB]
            iB := iB + 1
        endif
        iC := iC + 1
    endwhile
    if iA > dimA then {s-a epuizat sursa A}
        {mută în C componentele rămasă în B}
        for k := 0 to dimB - iB do C[iC + k] := B[iB + k]
    else {s-a epuizat sursa B}
        {mută în C componentele rămasă în A}
        for k := 0 to dimA - iA do C[iC + k] := A[iA + k]
    endif
endproc

```

Complexitatea algoritmului de interclasare: timp și spațiu

Estimăm complexitatea întâi în funcție de numărul de comparații. La fiecare comparație se mută în C o valoare, care nu va mai fi examinată ulterior de algoritm. Pentru a completa toate cele $\dim A + \dim B$ locații ale destinației C , vom face deci cel puțin $\dim A + \dim B - 1$ comparații (ultima componentă se mută în C , fără a mai fi comparată).

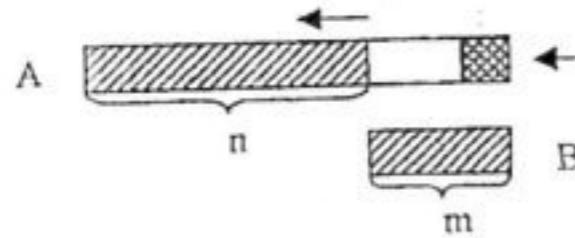
În felul acesta, numărul maxim de comparații pe care îl face algoritmul de interclasare este $C_{\max} = \dim A + \dim B - 1$. Este cazul cel mai nefavorabil, care se atinge dacă ultimele componente ale surselor trebuie să fie comparate între ele, deci dacă $A[\dim A]$ și $B[\dim B]$ vor fi cele mai mari elemente din C .

Există și cazul cel mai favorabil, cu număr minim de comparații $C_{\min} = \min(\dim A, \dim B)$, caz care se atinge când vectorul sursă de dimensiune mai mică are toate componentele mai mici decât cele din a doua sursă. Dacă de exemplu $\dim B \leq \dim A$ și $B[\dim B] \leq A[1]$, atunci se fac doar $\dim B$ comparații, ($\dim B = \min(\dim A, \dim B)$), iar toate componente din A se mută în C fără comparații.

În ceea ce privește numărul de mutări, el este tot timpul constant și egal cu $M = \dim A + \dim B$.

O caracteristică importantă a acestui algoritm, care-l placează într-o clasă distinctă față de toți ceilalți algoritmi discuți în acest capitol este faptul că are nevoie de spațiu în plus, $C[1.. \dim A + \dim B]$, egal cu dimensiune cu spațiul necesar datelor de intrare. Cu alte cuvinte, față de ceilalți algoritmi, spațiul utilizat este dublu.

Algoritmul se poate îmbunătăți ca să folosească doar $\min(\dim A, \dim B)$ locații în plus. Presupunem $\dim A = n$, $\dim B = m$ și $n \geq m$. Alocăm pentru A un număr de $n + m$ locații, sursa din A ordonată crescător de la $A[n]$ la $A[1]$, și punem rezultatul interclasării în capătul liber al lui A (deoarece A are loc pentru cele m componente ale lui B). La mutări de elemente din A , se mai eliberează locații. Destinația este A , ordonată de la $A[n+m]$ la $A[1]$.



Sortarea prin interclasare

Potem folosi interclasarea pentru a sorta un vector. Algoritmul are următoarea structură:

- (1) Se pleacă cu subvectori de lungime 1 ai lui A (care sunt sortați). Interclasând căte doi asemenea subvectori obținem A împărțit în subvectori de lungime 2 ordonați crescător.
- (2) A este împărțit în subvectori de lungime l ordonați crescător. Se interclasează căte doi, obținând subvectori de lungime $2 \cdot l$ ordonați crescător (această operatie se numește pasă și este implementată de procedura *MergePass*). Se reia pasul (2) până când avem un singur subvector.

Algoritmul este implementat de procedura *MergeSort* prezentată în continuare. Ea gestionează ca spațiu doi vectori A și B de aceeași lungime, $\dim A$, în modul următor: la pasul (1) A este sursă iar B destinație pentru interclasare. La pasul următor, B este sursă și A destinație și.m.d.

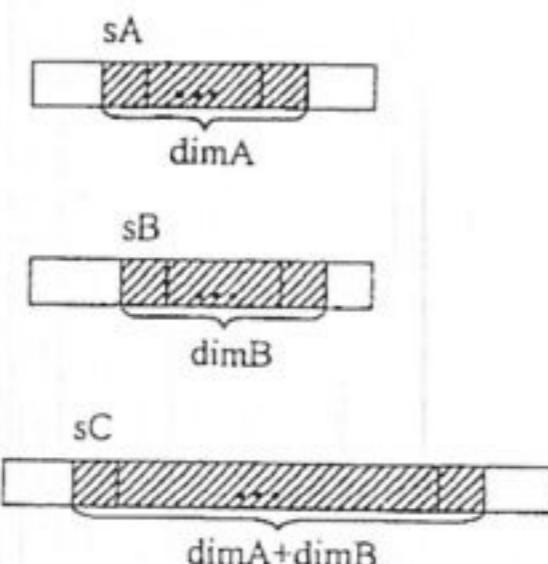
```

procedure Merge (dimA, dimB, dimC, sA, sB, sC, A, B, C)
{interclasează (exact ca și Merging) subvectorul  $A[sA, \dots, sA + \dim A - 1]$ 
cu subvectorul  $B[sB, \dots, sB + \dim B - 1]$  și pune rezultatul pe subvectorul
 $C[sC, \dots, sC + \dim A + \dim B - 1]$ .}
begin
  iA := sA - 1; {initializarea indicelui curent al lui A}
  iB := sB - 1; {initializarea indicelui curent al lui B}
  iC := sC - 1; {initializarea indicelui curent al lui C}
  while (iA < dimA + sA - 1) and (iB < dimB + sB - 1) do
    begin
      iC := iC + 1;
      if A[iA + 1] < B[iB + 1] then
        begin
          iA := iA + 1;
          C[iC] := A[iA];
        end
      else
        begin
          iB := iB + 1;
          C[iC] := B[iB];
        end
    end;
end;
```

```

if iA > dimA + sA - 1 then {s-a terminat vectorul A}
    {completăm pe C cu B}
    for k := 1 to dimB + sB - iB - 1 do
        C[iC + k] := B[iB + k]
    else {s-a terminat vectorul B}
        {completăm pe C cu A}
        for k := 1 to dimA + sA - iA - 1 do
            C[iC + k] := A[iA + k]
    end; {procedure Merge}

```



```

procedure MergePass (n, l, A, B);
{Se dă vectorul A, de dimensiune n, împărțit în subvectori de vectori de lungime l (eventual cu excepția ultimului, care poate fi mai scurt) sortați. Fiecare pereche de subvectori este ordonată prin interclasare folosind procedura Merge. Rezultatul se depune în B.}
begin
    q := n / (l shl 1); {q = numărul perechilor de subvectori întregi}
    s := (l * q) shl 1; {s = l * q * 2 = numărul de elemente în cele q perechi}
    r := n - s; {r = numărul de elemente rămase după interclasarea celor q perechi}
    {interclasarea celor q perechi de subvectori întregi}
    for i := 1 to q do
        begin

```

```

sA := ((i - 1) * l) shl 1 + 1; {indicele de la care începe al i-lea vector de interclasat}
    Merge (l, l, sA, sA + 1, sA, A, A, B)
end;
{ce facem cu restul componentelor?}
if r <= l then {cazul în care mi-a rămas un singur subvector}
    for i := l to r do
        B[s + i] := A[s + i]
    else {cazul în care mi-au rămas 2 subvectori, ultimul incomplet}
        Merge (l, r - l, s + 1, s + 1 + 1, s + 1, A, A, B)
end; {procedure MergePass}

```

```

procedure MergeSort (dimA, A);
begin
    l := 1;
    while l < dimA do
        begin
            MergePass(dimA, l, A, B);
            MergePass(dimA, l * 2, B, A);
            l := l * 4
        end;
    end; {procedure MergeSort}

```

Complexitatea sortării prin interclasare

Dacă la sortarea unui vector cu n componente facem k pași iterativi până ajungem să-l sortăm în întregime, înscamnă că între k și n avem relația $2^{k-1} < n \leq 2^k$, de unde rezultă $k = \lceil \log_2 n \rceil$. Deoarece la un pas iterativ (care constă din câte o pasă pe subvectorii surse) facem aproximativ $n/2$ comparații, rezultă că algoritmul intră în clasa de complexitate $O(n \log_2 n)$. Să nu uităm însă că el dublează dimensiunea spațiului de intrare.

Exerciții:

1. Să se scrie o versiune recursivă a sortării prin interclasare.
2. Pe vectorul A pe care vrem să-l sortăm prin interclasare putem depista subvectori de lungime mai mare decât 1 gata ordonați crescător și am putea începe prin a-i interclasă pe aceștia. Să se formuleze un algoritm care sortează un vector interclasând subșirurile maximale ordonate și să se scrie procedura asociată lui.

Capitolul IV

Arbore binari stricți. Aplicații.

Studiem în acest capitol o clasă particulară de arbori binari, și anume arborii binari stricți. Ei sunt arbori binari cu proprietatea că fiecare nod are exact fie *zero*, fie *două* fi. Din cauză că orice arbore binar se poate extinde printr-un procedeu canonico la un arbore binar strict, clasa acestora din urmă se dovedește a nu fi totuși foarte "particulară" în raport cu cea a arborilor binari, deci rezultatele obținute pentru arborii binari stricți sunt valabile și pentru arbori binari.

În prima secțiune dăm definiția, urmată de exemple. Demonstrăm în continuare proprietăți importante ale lor, care exprimă relații între numărul de noduri și adâncime, și între lungimi (internă, externă și medie) și numărul de noduri. Toate aceste relații sunt aplicabile arborilor binari.

În secțiunea a doua folosim aceste proprietăți pentru a stabili un rezultat extrem de important, și anume că limita inferioară a complexității algoritmilor de sortare internă bazați pe comparații între chei este aproximativă de $n \log_2 n$. Rezultatul completează prezentarea clasei acestor algoritmi din capitolul precedent.

Introducem apoi noțiunea de arbore binar strict cu ponderi și noțiunea asociată de lungime externă ponderată. Prezentăm algoritmul lui Huffman de construcție al unui arbore binar strict cu lungime externă ponderată minimă.

În ultimele două secțiuni prezentăm aplicații practice ale acestui algoritm: aplicații la codificare, care ne conduc la generarea de coduri binare de lungime variabilă, coduri optime pentru că minimizează lungimea mesajelor și aplicații la interclasarea mai multor șiruri, care minimizează numărul de mutări.

4.1. Arbori binari stricți. Definiție. Proprietăți.

Într-un arbore binar oarecare un nod poate avea zero, unul sau doi fiu.

Definiție: Un *arbore binar strict* este un arbore binar în care fiecare nod are fie nici un fiu, fie exact doi fiu.

Cu alte cuvinte, într-un asemenea arbore spre deosebire de unul binar oarecare nu vor exista noduri cu un singur fiu.

Nodurile cu doi copii se vor numi *noduri interne*, iar cele fără copii se vor numi *noduri externe* sau *frunze*. Ele pot fi de alt tip decât nodurile interne.

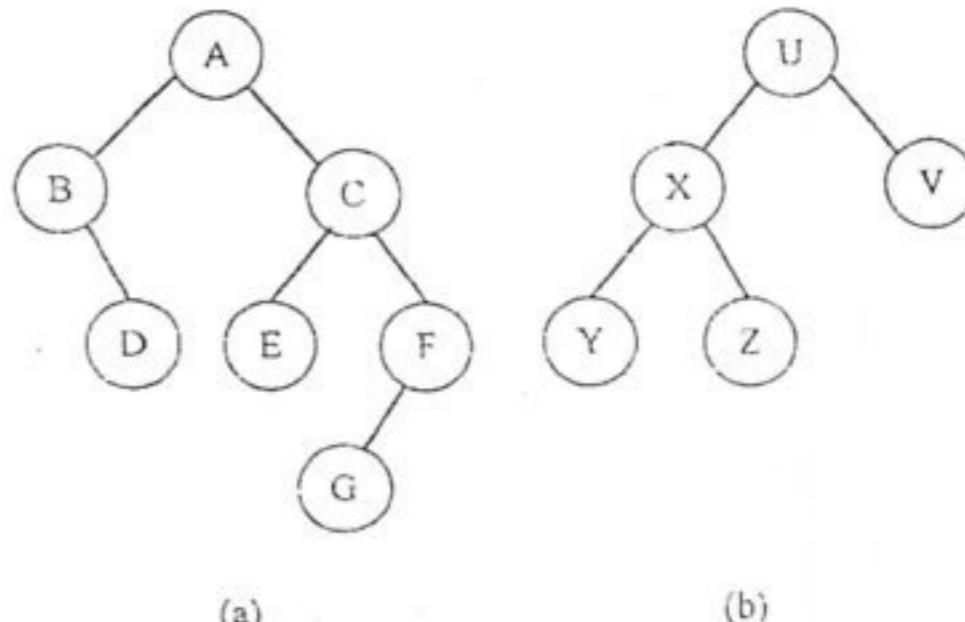


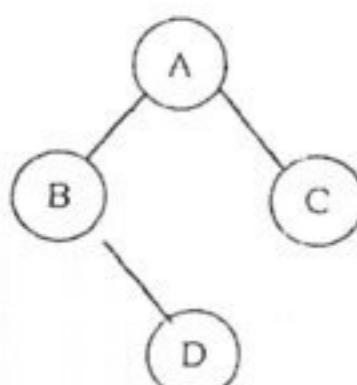
Fig.4.1.1. (a) Un arbore binar nestrikt. (b) Arbore binar strict.

În mulțimea arborilor binari, clasa arborilor binari stricți joacă un rol important. Unul din motive este acela că orice arbore binar se poate extinde (completa) în mod canonic până la un arbore binar strict.

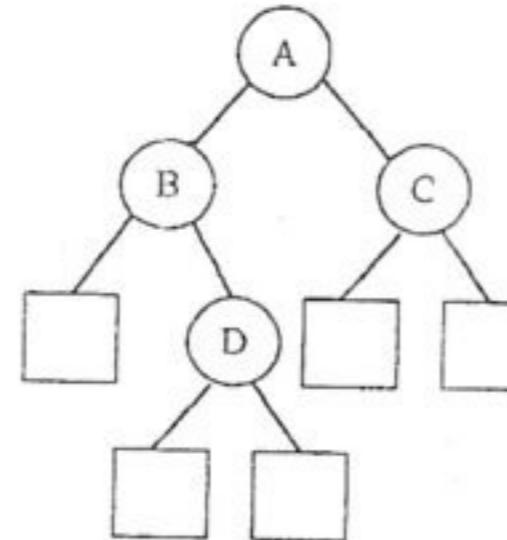
Completarea canonica a unui arbore binar oarecare T la unul strict se face în felul următor: se înlocuiesc fiecare fiu vid al nodurilor din T cu un nod de tip special. Nodurile arborelui inițial T vor deveni toate noduri interne, iar cele adăugate canonice vor fi frunze.

Convenim să reprezentăm grafic cu cercuri nodurile interne și cu dreptunghiuri nodurile externe.

Ilustrăm mai jos procedeul completării canonice al arborilor binari:

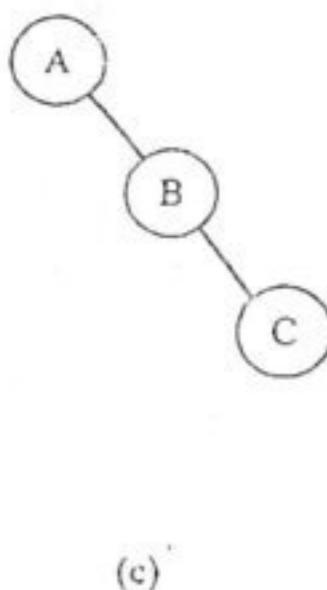


(a)

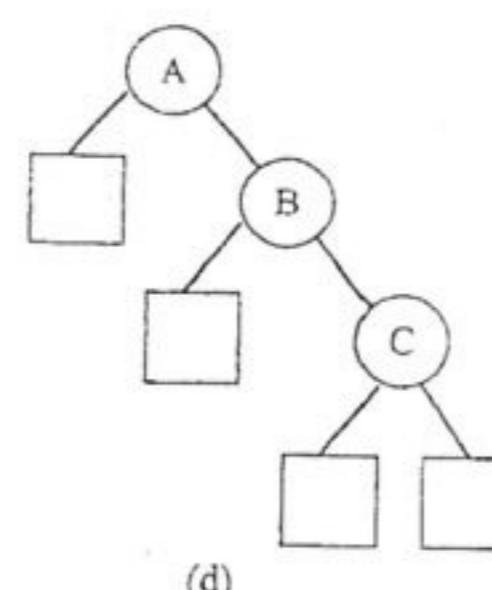


(b)

Fig. 4.1.2. Un arbore binar (a) și completatul său canonic la un arbore binar strict (b).



(c)



(d)

Fig. 4.1.3. Un arbore binar (c) și completatul său canonic (d).

După cum vom vedea din exemplele de mai jos, distingerea celor două tipuri de noduri - interioare sau frunze - nu este importantă doar pentru structura de arbore, ci și pentru tipul de informație pe care îl menținem în fiecare nod.

Exemple de aplicații ale structurii de arbore binar strict

(a) Reprezentarea expresiilor aritmetice cu operatori binari

Vrem să reprezentăm expresii aritmetice ce conțin operatori binari, (de exemplu $\{+, -, *, /\}$ operatorii de adunare, scădere, înmulțire și împărțire) și operanzi (fie constante, de tip întreg sau real, fie variabile, deci elemente de tip caracter peste un anume alfabet).

Vom reprezenta operanzi în frunze, iar operatorii în noduri interioare.

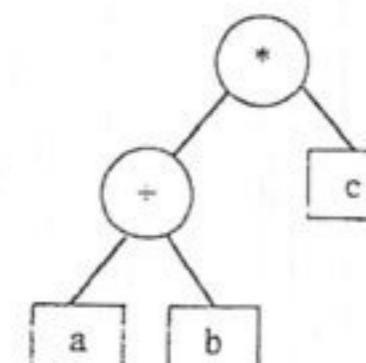


Fig. 4.1.4. Expresia $(a+b)*c$ ca arbore binar strict.

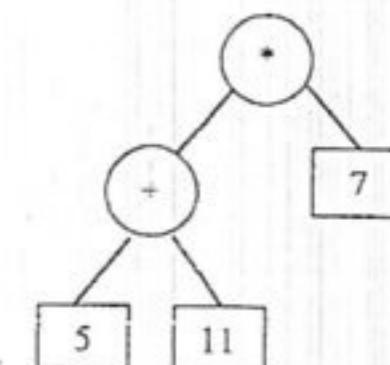


Fig. 4.1.5. Expresia $(5+11)*7$ ca arbore binar strict.

(b) Reprezentarea procedurilor de decizie

În mare, o procedură de decizie se bazează pe teste în urma cărora se decide asupra unor acțiuni. Dacă testele sunt cu rezultat binar (de tip Da/Nu) le putem reprezenta în nodurile interioare ale unui arbore binar strict, iar acțiunile în frunze.

(c) Reprezentarea algoritmilor

Un algoritm, ca și o procedură de decizie descrisă la (b) constă din efectuarea unor teste cu rezultat binar, și din efectuarea unor acțiuni.

Pentru ordonarea mulțimii $\{a_1, a_2, a_3\}$ pe bază de comparații între chei vom avea arborele:

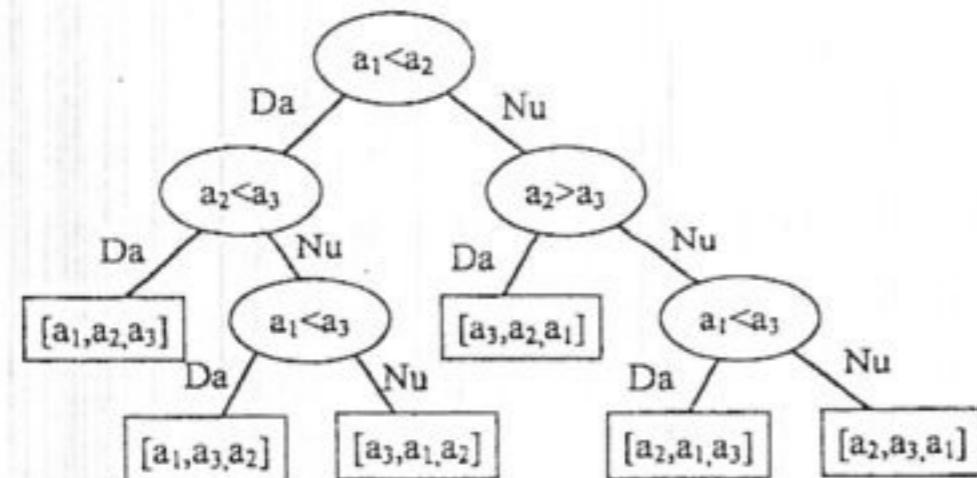


Fig. 4.1.6 Arbore binar strict asociat sortării mulțimii $\{a_1, a_2, a_3\}$.

(d) Aplicații la codificarea binară

Vrem să codificăm un alfabet, de exemplu $V = \{a, b, c, d, e\}$, cu ajutorul caracterelor 0 și 1 din scrierea binară. Reprezentăm în frunze fiecare din caracterele lui V și construim un arbore binar strict cu aceste cinci frunze. Acest lucru poate fi făcut în mai multe moduri, după cum ilustrăm mai jos:

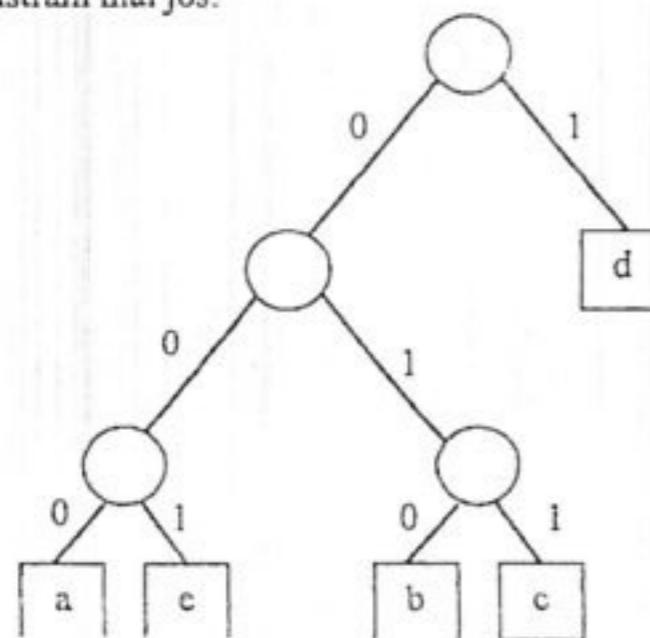


Fig. 4.1.7

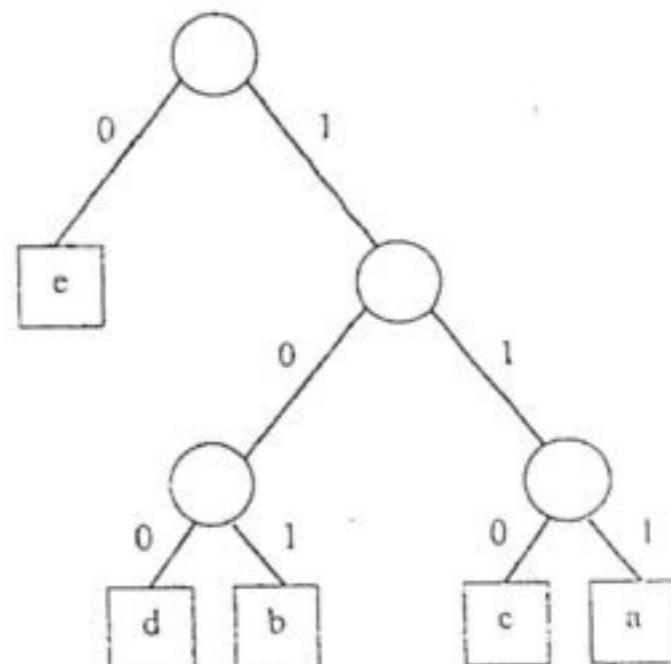


Fig. 4.1.8

Presupunem că fiul stâng are eticheta 0, iar cel drept 1. De la rădăcină până la fiecare frunză avem un drum unic, care va fi identificat printr-un sir compus numai din 0 și 1. Acest sir va fi codificarea binară a caracterului din frunză. De exemplu, codurile asociate arborilor din figurile de mai sus vor fi respectiv:

a	000
b	010
c	011
d	1
e	001

a	111
b	101
c	110
d	100
e	0

Fig. 4.1.9 Codurile binare peste alfabetul $\{a, b, c, d, e\}$, asociate arborilor din fig. 4.1.7, respectiv fig. 4.1.8.

Proprietăți

Datorită structurii sale deosebite, într-un arbore binar strict avem niște proprietăți speciale exprimate prin relații între diverse caracteristici ale sale, proprietăți pe care le vom prezenta și demonstra în continuare.

Notăm cu N_E numărul nodurilor externe și cu N_I numărul nodurilor interne ale unui arbore binar strict. Avem următorul rezultat.

Propoziția 1. Într-un arbore binar strict, numărul nodurilor externe și al celor interne sunt legate prin relația:

$$N_E = N_I + 1.$$

Demonstrație. Vom număra în două moduri diferențial numărul total de arce ale unui asemenea arbore. Pe de o parte, ținând cont de faptul că din fiecare nod intern pleacă două arce, putem exprima numărul total de arce ca fiind $2 * N_I$. Pe de altă parte, ținând cont de faptul că în fiecare nod al arborelui, cu excepția rădăcinii, intră căte un arc, putem exprima numărul total de arce ca fiind $N_I + N_E - 1$. Egalând cele două expresii, obținem

$$2N_I = N_I + N_E - 1,$$

din care rezultă, prin calcule

$$N_E = 2N_I - N_I - 1 = N_I + 1,$$

ceea ce ne conduce exact la relația din enunț. q.e.d.

Vom numi *lungimea externă* a unui arbore binar strict suma lungimilor drumurilor de la rădăcină până la fiecare nod extern. Drumul de la rădăcină la frunză se măsoară în număr de arce. Notând cu E mulțimea frunzelor, avem deci:

$$L_E = \sum_{x \in E} l(r, x)$$

unde r este rădăcina iar cu $l(r, x)$ am notat lungimea drumului de la r la nodul x .

Similar, se introduce noțiunea de *lungimea internă* a unui arbore binar strict, ca fiind suma lungimilor drumurilor de la rădăcină la toate nodurile interioare. Notând cu I mulțimea nodurilor interioare, avem formula care dă lungimea internă

$$L_I = \sum_{y \in I} l(r, y).$$

O relație importantă între cele două tipuri de lungimi și numărul de noduri ale unui arbore binar strict este dată de următorul rezultat:

Propoziția 2. Într-un arbore binar strict este adevărată relația

$$L_E = L_I + 2N_I.$$

Demonstrație. Folosim inducția matematică, după $n =$ numărul total de noduri ale unui arbore binar strict. (Stim că n nu poate lua orice valoare din mulțimea numerelor naturale.)

(a) $n = 1$

Cel mai simplu tip de arbore binar strict este cel cu un singur nod, deci pentru care $n = 1$. Acest nod va fi o frunză și pentru acest arbore avem următoarele valori ale caracteristicilor sale:



Arbore binar strict cu
un nod

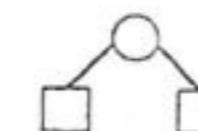
$$\begin{array}{l} N_E = 1 \\ N_I = 0 \end{array}$$

$$\begin{array}{l} L_E = 0 \\ L_I = 0 \end{array}$$

Înlocuite în formula din enunț ele dau o relație adevărată, deci pentru $n = 1$ afirmația este demonstrată.

(b) $n = 3$

Să ilustrăm și cazul celui mai simplu tip de arbore binar strict care are și un nod intern: acesta va fi unicul nod intern, va fi rădăcină și va avea doi copii frunze (de unde $n = 3$). Caracteristicile lui sunt:



Arbore binar strict cu
trei noduri

$$\begin{array}{l} N_E = 2 \\ N_I = 1 \end{array}$$

$$\begin{array}{l} L_E = 2 \\ L_I = 0 \end{array}$$

Aceste valori, înlocuite în formula din enunț dau o relație adevărată, deci am demonstrat afirmația și pentru $n = 3$.

(c) Presupunem că ipoteza de inducție că relația $L_E = L_I + 2N_I$ este adevărată pentru orice arbore binar strict care are un număr total de noduri mai mic decât un număr natural m dat.

Fie un arbore binar strict T cu numărul total de noduri n , $n \geq m$. Arboarele T este compus dintr-un nod rădăcină, intern, și fișii săi stâng și drept, T^s și T^d , care sunt la rândul lor subarbore binari stricți. Notând cu N_I^s , N_E^s , L_I^s , L_E^s caracteristicile lui T^s și cu N_I^d , N_E^d , L_I^d , L_E^d caracteristicile lui T^d , putem exprima caracteristicile lui T în funcție de caracteristicile fiilor săi prin următoarele relații:

- (1) $N_I = N_I^s + N_I^d + 1$
- (2) $N_E = N_E^s + N_E^d$
- (3) $L_E = L_E^s + N_E^s + L_E^d + N_E^d$
- (4) $L_I = L_I^s + N_I^s + L_I^d + N_I^d$

Ultimele două relații rezultă din faptul că drumul în T până la un nod x , de orice tip, x aparținând de exemplu lui T^s , este egal cu drumul de la rădăcina lui T^s până la x , la care se adaugă arcul de la rădăcina lui T la rădăcina lui T^s .

Subarboreii T^s și T^d au un număr total de noduri strict mai mic decât T , deci lor le putem aplica ipoteza de inducție și avem relațiile adevărate:

- (5) $L_E^s = L_I^s + 2N_I^s$
- (6) $L_E^d = L_I^d + 2N_I^d$

Calculăm acum lungimea externă a lui T , plecând de la relația (3) și aplicând întâi (5) și (6). Avem

$$L_E = L_E^s + N_E^s + L_E^d + N_E^d = L_I^s + N_I^s + N_E^s + L_I^d + N_I^d + N_E^d - N_I^s - N_I^d = (L_I^s + N_I^s + L_I^d + N_I^d) + (2N_I^s - 1) + (2N_I^d - 1)$$

unde, pentru ultima egalitate, am grupat termenii și am aplicat relația stabilită în propoziția 1 arborilor T^s și T^d . Înțând cont de expresia din prima paranteză, putem aplica (4) și avem deci în continuare:

$$L_E = L_I + 2(N_I^s + N_I^d + 1) = L_I + 2N_I$$

unde, pentru ultima egalitate am aplicat relația (1).

Aceasta încheie demonstrația prin inducție a formulei din enunț. q.e.d.

Propoziția 3. Într-un arbore binar strict de adâncime d avem următoarea inegalitate

$$N_E \leq 2^d.$$

Demonstrație. Inegalitatea din enunț revine la demonstrarea următoarelor afirmații:

- (a) Dintre toți arborii binari stricți de adâncime dată, d , cel cu număr maxim de frunze este cel care are toate frunzele la ultimul nivel (adică d).
- (b) Un arbore cu toate frunzele la nivelul d are exact 2^d frunze, adică $N_E = 2^d$.

Afirmația (a) este imediată.

Demonstrăm (b) prin inducție după valorile lui d . Pentru $d = 0$ avem cazul arborelui cu un singur nod, de tip frunză și relația se verifică.

Pentru $d = 1$ avem un arbore binar strict cu un singur nod interior, rădăcina și două frunze și din nou relația se verifică.

Presupunem că pentru $d = k$, numărul de frunze (aflate toate la nivelul k) este $N_E = 2^k$.

Putem construi dintr-un asemenea arbore, în mod foarte simplu și direct, un arbore binar strict care are adâncimea $d = k + 1$ și proprietatea că toate frunzele sunt la nivelul $k + 1$. Se înlocuiește fiecare frunză de la nivelul k , cu un nod interior, iar acestora li se atachează căte două frunze, procedeu care produce un arbore cu de două ori mai multe frunze decât precedentul. Deci, pentru arboarele de adâncimea $k + 1$ și toate frunzele la acest nivel avem $N_E = 2 * 2^k = 2^{k+1}$, ceea ce încheie demonstrația. q.e.d.

Un corolar extrem de util al rezultatului precedent este următorul:

Corolar. Într-un arbore binar strict de adâncimea d avem inegalitatea

$$d \geq \lceil \log_2 N_E \rceil.$$

Demonstrație. Imediată din relația stabilită în propoziția anterioară,

$$N_E \leq 2^d$$

căreia îi aplicăm funcția crescătoare \log_2 , iar apoi funcția parte întreagă superioară și obținem

$$\lceil \log_2 N_E \rceil \leq d$$

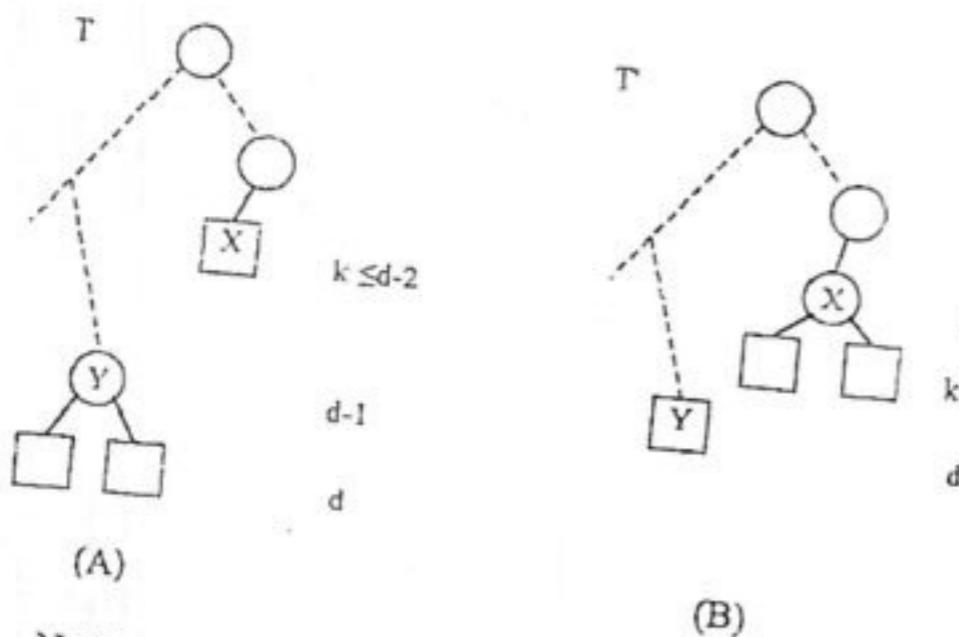
care e chiar relația din enunț. q.e.d.

Stabilim în continuare niște rezultate care ne vor permite să estimăm lungimea externă minimă și medie a unui arbore binar strict.

Propoziția 4. Dintre toți arborii binari stricți cu același număr de frunze, fixat, N_E , au lungime externă minimă aceia cu proprietatea că frunzele lor sunt repartizate pe cel mult două niveluri adiacente.

Demonstratie. Notăm ca și până acum cu d adâncimea unui arbore. Vom considera un arbore binar strict care nu are proprietatea din enunț și vom arăta că îl putem modifica, transformându-l într-un arbore binar strict cu același număr de frunze, dar cu lungime externă strict mai mică și cu frunzele repartizate pe niveluri mai apropiate.

Fie T un asemenea arbore, ilustrat în figura (A), care are frunze și la un nivel $k \leq d-2$; notăm cu X o asemenea frunză și cu Y un nod interior aflat la nivelul $d-1$, deci care va avea doi copii frunze la nivelul d . Îi facem lui T următoarele modificări: înlocuim nodul interior Y cu o frunză, marcată tot cu Y , iar frunza X o înlocuim cu un nod interior, marcat tot cu X , căruia îi atașăm drept copii două frunze ce se vor afla la nivelul $k+1$. Obținem în felul acesta tot un arbore binar strict, să-l notăm T' , ilustrat în figura (B).



Notăm cu L_E lungimea externă a lui T și cu L'_E lungimea externă a lui T' . Legătura dintre L'_E , L_E , d și k se exprimă prin relația

$$L'_E = L_E - (2d + k) + [2(k + 1)] + (d - 1)$$

care reflectă modificările făcute asupra lui T pentru a-l obține pe T' .

Calculând, obținem:

$$L'_E - L_E = -2d - k + 2k + 2 + d - 1 = -d + k + 1 = -(d - k - 1).$$

Dar, din $k \leq d-2$ rezultă $1 \leq d-k-1$, deci $d-k-1 > 0$ și deci $L'_E - L_E < 0$, ceea ce înseamnă că lungimea externă a lui T' este strict mai mică decât cea a lui T . În plus, este evident că modificările aduse lui T nu au afectat numărul total de frunze, care rămâne N_E .

Construcția de mai sus demonstrează că ne putem „plimba” în familia arborilor binari stricți cu același număr N_E de frunze, păstrându-l, și în același timp, micșorând strict lungimea externă. Putem face această „plimbare”, aplicând în mod repetat construcția, atât timp cât relația $d - k \geq 2$ este satisfăcută și n-o înai putem aplica (cu același efect, micșorarea strictă a lungimii) pentru $d - k = 1$, adică pentru valori ale lui k ce satisfac relația $k = d - 1$. Cu alte cuvinte, trebuie să ne oprim la penultimul nivel al arborelui, ceea ce demonstrează afirmația din enunț, că lungimea externă se minimizează pentru acei arbori ai căror frunze sunt repartizate pe cel mult două niveluri adiacente. q.e.d.

Propoziția 5. Lungimea externă minimă a unui arbore binar strict cu l frunze este dată de formula

$$L_E^{\min} = l \lfloor \lg l \rfloor - 2(l - 2^{\lfloor \lg l \rfloor})$$

Demonstratie. Avem de tratat două cazuri pentru care se atinge lungimea externă minimă.

(a) Cazul în care toate frunzele sunt la ultimul nivel, d . Atunci $l \approx 2^d$, deci $d = \lg l$ (unde \lg este notația pentru logaritmul în baza 2).

În cazul acesta, lungimea externă este dată de formula

$$L_E = l * d = l * \lg l,$$

Unde în partea dreaptă avem exact valoarea expresiei din enunț, deoarece

$$l - 2^{\lfloor \lg l \rfloor} = l - 2^{\lg l} = l - 2^d = 0.$$

(b) Dacă frunzele nu sunt toate la același nivel, arunci, conform Propoziției 4, ele sunt repartizate pe două niveluri adiacente, d și $d-1$. În acest caz l nu va mai fi putere a lui 2, ci vom avea $l < 2^d$, de unde rezultă

$$d = \lceil \lg l \rceil \text{ și } d-1 = \lfloor \lg l \rfloor.$$

Formula din enunț, ce trebuie demonstrată devine

$$L_E = l(d-1) + 2(l - 2^{d-1}).$$

Când calculăm L_E , ajungem la nivelul $d - 1$ și suma lungimilor drumurilor până acolo se exprimă prin $l(d - 1)$. Pentru nodurile de la acest nivel, care sunt frunze, calculul s-a terminat. Mai trebuie să estimăm acum numărul de frunze de la nivelul d , deoarece

$$L_E = l(d - 1) + \text{numărul de frunze de la nivelul } d.$$

Dar, numărul de frunze de la nivelul d este dublul numărului de noduri interioare de la nivelul $d - 1$, iar acesta din urmă este egal cu $(l - 2^{d-1})$. Înlocuind în expresia de mai sus, obținem

$$L_E = l(d - 1) + 2(l - 2^{d-1})$$

exact ceea ce trebuia demonstrat. q.e.d.

Următorul rezultat ne dă o margine inferioară pentru lungimea externă medie a unui arbore binar strict în funcție de numărul total de frunze.

Propoziția 6. Într-un arbore binar strict avem următoarea inegalitate:

$$L_E^{\text{medie}} \geq \lfloor \lg l \rfloor.$$

Demonstrație. Prin lungime medie înțelegem media raportată la numărul de frunze. Deoarece am estimat în Propoziția 5 lungimea minimă, putem estima acum media ei și obținem

$$L_E^{\text{min}}/l \geq \lfloor \lg l \rfloor - 2(l - 2^{d-1})/l.$$

Ultimul termen al sumei este un număr pozitiv, subunitar și avem următorul sir de inegalități

$$L_E^{\text{medie}} \geq L_E^{\text{min}}/l > \lfloor \lg l \rfloor.$$

(unde prima inegalitate este evidentă), ceea ce ne conduce la inegalitatea din enunț. q.e.d.

4.2. Limita inferioară a algoritmilor de sortare bazați pe comparații între chei

Complexitatea unui algoritm de sortare internă este o măsură a timpului de rulare al algoritmului în funcție de n , numărul elementelor vectorului de sortat. Algoritmi de sortare internă dintre care enumerăm:

- sortarea directă prin inserție
- sortarea directă prin selecția minimului (sau maximului)
- sortarea directă prin interschimbare
- sortarea rapidă (QuickSort)
- sortarea cu ansamblu (HeapSort)

efectuează comparații între chei, interschimbări de chei și operații de asignare de valori. Ultimele două operații le-am evaluat în funcție de numărul de mutări. Numărul mutărilor depinde în mod esențial de numărul de comparații, deci complexitatea algoritmilor se estimează în mod esențial în funcție de numărul de comparații.

Stim că primii trei algoritmi de mai sus au o complexitate de ordinul lui n^2 . Sunt algoritmi direcți, simpli, dar relativ neperformanți. Următorii doi, sortarea rapidă și sortarea cu ansamblu au performanțe mult mai bune și anume de ordinul lui $n \log n$.

Vom demonstra în continuare un rezultat care arată că nu putem coborî sub limita $n \log n$, cu alte cuvinte, că nu putem găsi algoritmi de sortare în clasa celor bazați pe comparații între chei, mai performanți.

Teoremă. Orice algoritm de sortare a n chei, algoritm bazat pe comparații între chei, va face cel puțin

- $\lceil \log_2 n! \rceil$ comparații în cazul cel mai nefavorabil;
- $\lfloor \log_2 n! \rfloor$ comparații în cazul mediu.

Demonstrație. Cheia demonstrării acestui rezultat este structura de arbore binar strict cu proprietățile sale.

Presupunem că lucrăm cu algoritmi pentru care datele de intrare sunt vectori de lungime n , (x_1, x_2, \dots, x_n) , cu componentele presupuse distincte. Reamintim că lucrăm cu algoritmi bazați pe comparații de tipul $x_i < x_j$. Unui asemenea algoritm îi se asociază un arbore de decizie în modul următor:

- o instrucție de ieșire va fi o frunză ce conține vectorul sortat;

- b) o comparație $x_i < x_j$ va fi un nod interior, al cărui fiu stâng este un arbore asociat instrucțiunilor ce se execută dacă $x_i < x_j$ este adevărată, iar fiul drept este arborele asociat instrucțiunilor ce se execută dacă $x_i < x_j$ este falsă.
- c) Putem presupune că acest arbore este binar strict, deoarece, dacă un nod interior are un singur fiu, putem înlocui nodul cu acest fiu, obținând arborele de decizie al unui algoritm cel puțin la fel de eficient ca cel inițial.

Numărul total al instrucțiunilor de ieșire și deci al frunzelor va fi numărul total de ordini posibile pe vectorul (x_1, \dots, x_n) , deci $n!$.

Pe un set particular de date de intrare, acțiunea algoritmului este echivalentă cu parcurgerea unui drum de la rădăcină până la frunza ce conține respectivul set de date ordonat. Numărul de comparații efectuate este egal cu numărul de noduri interioare pe acest drum, care este egal cu lungimea acestui drum măsurată în arce. (Vezi fig. 4.1.6 pentru $n=3$.)

Dacă vrem să estimăm numărul de comparații pe care îl face algoritmul în cazul mediu, aceasta revine la a estima numărul mediu de noduri interioare pe toate drumurile, cu alte cuvinte lungimea externă medie a arborelui binar strict de decizie asociat.

Dacă vrem să estimăm numărul de comparații în cazul cel mai nefavorabil, aceasta revine la a estima numărul de noduri interioare pe drumul cel mai lung până la o frunză.

Dacă vrem să estimăm numărul de comparații în cazul cel mai favorabil, aceasta revine la a estima numărul de noduri interioare pe drumul cel mai scurt la o frunză.

Reamintim că arborele nostru de decizie este un arbore binar strict cu $n!$ frunze.

Din Propoziția 6 avem că lungimea externă medie a unui astfel de arbore este mai mare decât $\lfloor \log_2 n! \rfloor$, deci în cazul mediu un algoritm de sortare bazat pe comparații face cel puțin $\lfloor \log_2 n! \rfloor$ comparații, ceea ce încheie demonstrația afirmației (b).

Pentru a demonstra afirmația (a), ne reamintim că drumurile cele mai lungi până la o frunză sunt așa pentru frunzele aflate la ultimul nivel, $d =$ adâncimea arborelui, iar Corolarul la Propoziția 3 stabilește că

$$d \geq \lceil \log_2 n! \rceil,$$

ceea ce încheie demonstrația afirmației (a). Q.E.D.

4.3. Arboare binari stricți cu ponderi. Algoritmul lui Huffman

Fie un arbore binar strict, T , cu mulțimea frunzelor $E = \{a_1, a_2, \dots, a_n\}$. El se va numi *cu ponderi* dacă există o funcție cu valori reale $w: E \rightarrow R$, cu alte cuvinte, dacă fiecare frunză a_i are asociat un număr real w_i , numit *ponderea ei*.

Reamintim că lungimea externă a lui T este

$$L_E = \sum_{i=1}^n l_i = \sum_{i=1}^n l(r, a_i),$$

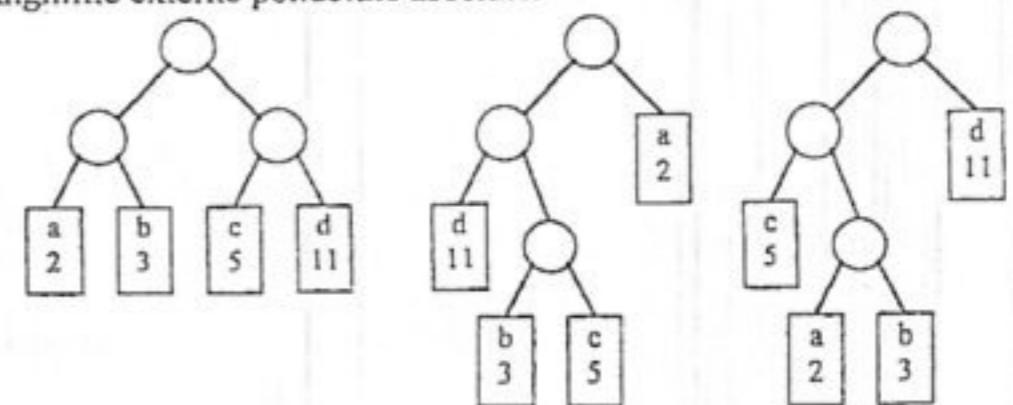
unde am notat $l_i = l(r, a_i)$ lungimea drumului de la rădăcina r a lui T , la frunza a_i . Dacă T este ponderat, putem considera *lungimea externă ponderată* a lui, un număr real exprimat prin

$$L_E^w = \sum_{i=1}^n w_i l_i = \sum_{i=1}^n w_i l(r, a_i).$$

Pentru arborii binari stricți fără ponderi, răspunsul la întrebarea pentru care din ei se minimizează lungimea externă a fost dat în Propoziția 4: la un număr fixat de frunze, lungimea externă se minimizează pentru aceia care au frunzele repartizate pe cel mult două niveluri adiacente.

Ne punem o întrebare similară pentru arborii binari stricți cu ponderi și anume: pentru un număr fixat de frunze, cu ponderi date, fixate și ele $\{(a_1, w_1), \dots, (a_n, w_n)\}$, pentru care arbori se atinge lungimea externă ponderată minimă?

Ilustrăm variația lungimii externe ponderate, pe un exemplu cu patru frunze cu ponderi: $\{(a, 2), (b, 3), (c, 5), (d, 11)\}$. Prezentăm în figura de mai jos trei arbori binari stricți ce conțin doar aceste frunze, cu lungimile externe ponderate asociate.



Observăm, de exemplu, că arboarele (A) care are frunzele toate pe același nivel și ar avea lungime externă minimă dacă nu ar avea ponderi, are acum o lungime externă ponderată $L = 42$, care este mai mare decât a arborelui (C).

Un răspuns constructiv la întrebarea pentru care arbori se atinge lungimea externă ponderată minimă este dat în paragraful următor.

Algoritmul lui Huffman

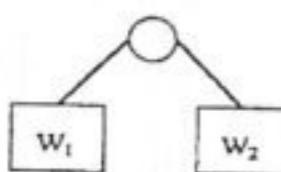
Problema de rezolvat este următoarea: date fiind n ponderi, noteate w_1, w_2, \dots, w_n să se găsească printre toți arborii binari stricti cu n frunze unul, nu neapărat unic, care să aibă cea mai mică lungime externă ponderată. O metodă de construire a unui asemenea arbore este cunoscută sub numele de *algoritmul lui Huffman*.

Algoritmul lui Huffman – o formulare recursivă

- pentru $n = 1$ soluția va fi arborele binar strict cu un singur nod de tip frunză, cu unica pondere w_1 . Lungimea lui externă ponderată va fi 0.
- Presupunem că din $n - 1$ ponderi date, putem construi un arbore binar strict cu lungime externă ponderată minimă.
- Fie acum n ponderi date, w_1, w_2, \dots, w_n și să presupunem că w_1 și w_2 sunt cele mai mici dintre aceste ponderi. Să considerăm acum sirul de $n - 1$ ponderi, $w_1 + w_2, w_3, \dots, w_n$. Conform punctului (b) (ipoteza de inducție) putem construi un arbore binar strict T' asociat acestui sir de ponderi

$$w_1 + w_2, w_3, \dots, w_n$$

care să fie soluție a problemei noastre, adică să aibă lungime externă ponderată minimă, L' . Să înlocuim acum în arborele T' frunza cu ponderea $w_1 + w_2$ cu un subarbore de forma



obținând în felul acesta un arbore binar strict T . Arborele T va fi soluție pentru ponderile $w_1, w_2, w_3, \dots, w_n$ căci lungimea lui externă ponderată va fi L , cu $L = L' + w_1 + w_2$, iar L' este minimă și w_1 și w_2 sunt și ele cele mai mici ponderi din sir.

Algoritmul lui Huffman – o formulare iterativă

Presupunem că cele n ponderi sunt aranjate în ordine descrescătoare $w_1 \geq w_2 \geq \dots \geq w_{n-1} \geq w_n$.

Pasul 1. Algoritmul formează n arbori binari stricti, fiecare de tip frunză, cu câte o pondere w_i asociată ci. Avem o pădure cu n arbori.

Pasul 2. Se „leagă” doi subarbori cu ponderi minime din pădure, cu ajutorul unui nod interior pentru care ei devin cei doi fii, iar acest arbore va fi arbore binar strict cu ponderea asociată egală cu suma ponderilor arborilor pe care i-am legat, pondere pe care o vom asocia nodului intern. Am redus în felul acesta cu 1 numărul de subarbori din pădure. Se reia pasul iterativ (2) pînă când obținem un singur arbore.

În general, la pasul iterativ k , algoritmul are $n - k + 1$ arbori binari stricti cu ponderi. „Leagă” doi arbori cu ponderile cele mai mici conform procedeului descris anterior, producând în felul acesta $n - k$ arbori binari stricti cu ponderi.

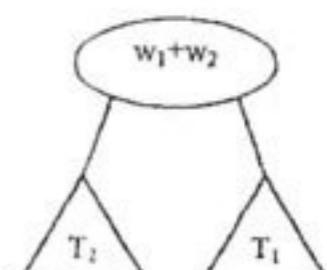
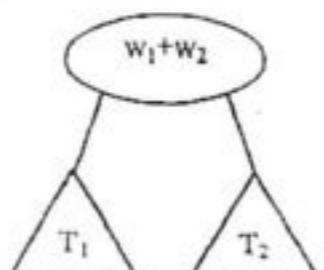
Algoritmul se termină la al $(n - 1)$ -lea pas iterativ, când produce un singur arbore, căci, din pădurea inițială de n arbori, la fiecare pas iterativ numărul de arbori descrește cu unul.

Un arbore binar strict obinut prin aplicarea algoritmului de mai sus se va numi *arbore Huffman* asociat ponderilor $\{w_1, w_2, \dots, w_n\}$.

Pentru simplificarea prezentării am ignorat în mod deliberat alte informații ce ar putea fi conținute în frunze. Aceste alte informații sunt vitale pentru diversele aplicații ale algoritmului, după cum vom vedea în secțiunile următoare, dar, pentru construirea efectivă a unui arbore Huffman, sunt suficiente ponderile.

Ne-unicitatea arborelui Huffman

Aplicarea algoritmului nu conduce în general la un arbore Huffman unic. Unul din motivele acestui fapt este că, la fiecare „legare” a doi arbori T_1 și T_2 , de ponderi minime, w_1 , respectiv w_2 , avem două posibilități distincte (T_1 devine fiu stâng și T_2 fiu drept sau viceversa), algoritmul nefiind influențat decât de ponderea $w_1 + w_2$ a arborelui rezultat.

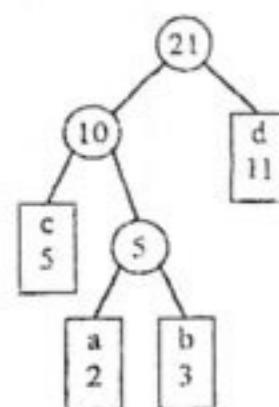
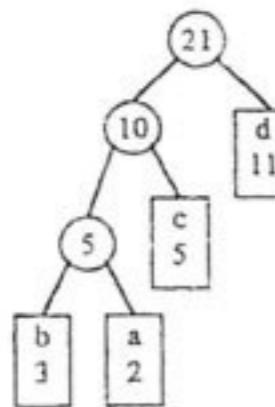
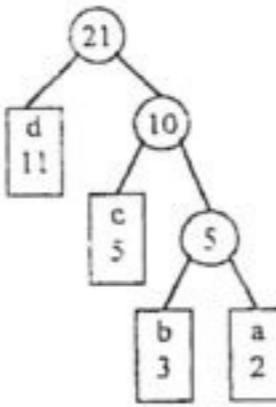


Această ambiguitate ar putea fi eliminată stabilind o convenție potrivit căreia, de exemplu, arborele cu ponderea cea mai mică se devină fiu stâng. Evident această convenție poate funcționa doar dacă cele două ponderi în discuție sunt distincte, adică $w_1 \neq w_2$.

Iar aceasta ar trebui să se întâmple la fiecare pas iterativ al algoritmului! Am pus astfel în evidență un al doilea motiv pentru neunicitatea arborelui Huffman și anume existența mai multor ponderi minime, egale între ele. Am pus, de asemenea, în evidență *cazul în care arborele Huffman este unic*, și anume dacă:

- ponderile inițiale au asemenea valori, încât la fiecare pas iterativ al algoritmului ponderile arborelor produși sunt distincte;
- se stabilește o convenție de „legare“ a celor doi arbori cu cele mai mici ponderi, de exemplu cel cu ponderea cea mai mică devine fiu stâng, iar convenția se respectă la fiecare pas iterativ.

Să ne reîntoarcem acum puțin la primul exemplu din acest capitol, care construia un arbore binar strict din mulțimea de frunze ponderate $\{(a, 2), (b, 3), (c, 5), (d, 11)\}$ și să observăm că arborele din figura (C) este un arbore Huffman, deci $L = 36$ este lungimea externă ponderată minimă. Ilustrăm mai jos și alți arbori Huffman asociati acestor ponderi:

(C) $L=36$ (D) $L=36$ (E) $L=36$

Exerciții

- Să se construiască arbori Huffman pentru următorul set de frunze cu ponderi $\{(a, 1), (b, 1), (c, 2), (d, 2), (e, 2), (f, 4)\}$ și să se calculeze lungimea externă ponderată minimă.
- Să se scrie în pseudo-cod și cod proceduri care construiesc un arbore Huffman dintr-un set dat de frunze cu ponderi (versiune iterativă și versiune recursivă).

4.4. Aplicații ale arborilor binari stricți la codificare. Coduri Huffman.

Fie V un alfabet finit. Notăm cu V^* monoidul liber generat de V , adică mulțimea “cuvintelor” (șiruri de caractere) formate peste alfabetul V , la care se adaugă cuvântul vid, λ , ce nu conține nici un caracter.

Cu notația de mai sus, prin $\{0, 1\}^*$ vom înțelege mulțimea șirurilor ce conțin doar caracterele 0 și 1 și o putem identifica cu mulțimea tuturor șirurilor de biți.

Numim *cod binar peste alfabetul V* o funcție injectivă $c : V \rightarrow \{0, 1\}^*$. Codul asociat unei litere $a \in V$ va fi un șir $c(a)$ ce conține doar caracterele 0 și 1. Proprietatea de injectivitate ne asigură că două litere distincte $a \neq b$ din V vor avea asociate coduri distincte, adică $c(a) \neq c(b)$.

O funcție $c : V \rightarrow \{0, 1\}^*$ ca mai sus se prelungește în mod natural la mulțimea cuvintelor peste V , adică la o funcție $\bar{c} : V^* \rightarrow \{0, 1\}^*$, prin formula

$$\bar{c}(a_1 a_2 \dots a_n) := c(a_1) c(a_2) \dots c(a_n).$$

Aceasta înseamnă: codul asociat cuvântului $a_1 a_2 \dots a_n$ se obține concatenând codurile fiecărei litere în parte, evident în ordinea în care ele apar în cuvânt. și funcția extinsă \bar{c} are proprietatea de injectivitate și o vom numi cod binar peste V^* .

În continuare, prin cod binar vom înțelege fie codul c peste V , fie extensia lui canonică \bar{c} la V^* , în funcție de context, și ne vom referi numai la astfel de coduri.

Spunem că un cod (binar) *are proprietatea prefix* dacă pentru oricare două litere $x, y \in V$, $c(x)$ nu este prefix al lui $c(y)$.

Arborii binari pot fi folosiți în mod natural la reprezentarea sau chiar la generarea codurilor binare cu proprietatea prefix. Etichetăm arcele arborelui binar cu convenția: un arc de tip fiu stâng are eticheta 0 iar un arc de tip fiu drept are eticheta 1. Punem caracterele din V în frunzele unui asemenea arbore iar codul $c : V \rightarrow \{0, 1\}^*$ asociat va fi dat de $c(a) =$ șirul de etichete al drumului (unic) de la rădăcina arborelui la frunza în care este reprezentat caracterul $a \in V$. Un asemenea cod este injectiv pentru că două litere distincte din V ocupă frunze diferite și este un cod cu proprietatea prefix pentru că nu reprezentăm caracter în nodurile interioare.

Spunem că un cod este *de lungime fixă* dacă toate literele din V se codifică cu siruri de biți de aceeași lungime, adică dacă:

$$|c(a)| = |c(b)| \text{ pentru orice } a, b \in V,$$

unde prin $|c(a)|$ notăm lungimea sirului de biți $c(a)$.

Un cod care nu are această proprietate se numește cod *de lungime variabilă*.

Problemele pe care ni le punem sunt:

(a) De construit un cod binar cu proprietatea prefix peste un alfabet dat V .

(b) Să codificăm cuvinte peste V adică: dat fiind orice cuvânt peste V , $a_1 a_2 \dots a_n \in V^*$, să producem codul asociat lui, din codurile literelor, folosind formula

$$\bar{c}(a_1 a_2 \dots a_n) = c(a_1) c(a_2) \dots c(a_n).$$

(c) Să decodificăm siruri de biți, adică: dat fiind orice cuvânt $u \in \{0, 1\}^*$, să decidem dacă există sau nu un cuvânt $x \in V^*$ al cărui cod este u și, în cazul afirmativ, să spunem care este acest x (unic) cu proprietatea $\bar{c}(x) = u$.

Cele trei probleme sunt legate între ele. În primul rând, în mod evident, codificarea și decodificarea sunt operații inverse una alteia. În al doilea rând, construcția codului trebuie făcută în așa fel încât să optimizeze anumite performanțe ale operațiilor de codificare și decodificare. După cum reiese din discuția despre legătura dintre arbori binari și codurile binare cu proprietatea prefix, problema construcției unui cod binar cu proprietatea prefix peste un alfabet dat, V , este echivalentă cu problemă construcției unui arbore binar strict pentru o mulțime dată de frunze, V .

Dacă vrem să generăm în felul acesta coduri de lungime fixă, va trebui să construim arboarele binar strict cu toate frunzele pe același nivel, și implicit, va trebui să adăugăm frunze care nu vor corespunde nici unui caracter din V dacă $|V| < 2^d$.

Pentru a genera coduri de lungime variabilă, nu e nevoie să impunem nici o restricție asupra nivelului la care se află frunzele și nici nu trebuie să adăugăm frunze în plus. Nivelul la care se află o frunză ne dă implicit lungimea codului asociat caracterului respectiv.

Potrivit dezideratului general, putem exploata intelligent această ultimă proprietate în felul următor.

Un deziderat general al oricărei probleme de codificare este ca lungimea mesajelor codificate să fie minimă. Acest deziderat este impus

de considerente practice, cum ar fi scurtarea timpului de transmisie fizică a mesajului codificat.

Unul din modurile de a minimiza lungimea mesajelor codificate este de a face lungimea codului fiecărui caracter invers proporțională cu frecvența lui de apariție. Cu alte cuvinte, codul trebuie să asocieze caracterelor cu probabilitate mare de apariție un cod cât mai scurt, iar celor mai puțin frecvente, coduri mai lungi.

Problema construcției unui cod binar peste V , care să aibă această proprietate, se reformulează în felul următor.

În primul rând, alfabetul de codificat, V , se dă împreună cu o repartizie de probabilitate pe el; cu alte cuvinte avem o funcție

$$w: V \rightarrow [0, 1]$$

cu proprietatea

$$\sum_{a \in V} w(a) = 1, a \in V.$$

Pentru fiecare $a \in V$, $w(a)$ reprezintă probabilitatea apariției caracterului a sau frecvența lui relativă în anumite texte.

Problema construcției unui arbore binar strict cu frunze din V devine în felul acesta o problemă de construcție a unui arbore binar strict *ponderat*, având ca frunze elementele lui V cu ponderile (probabilitățile) asociate, adică, mulțimea frunzelor va fi

$$E = \{(a, w(a)) : a \in V\}.$$

Unui asemenea set de frunze îl putem aplica algoritmul lui Huffman, construind din el un arbore binar strict (nu neapărat unic) cu lungime externă ponderată minimă.

Minimizarea lungimii externe ponderate va satisface dezideratul general de minimizare a lungimii mesajelor. În plus, din modul de funcționare al algoritmului se vede că acele caractere $a \in V$ cu probabilitate de apariție $w(a)$ mică sunt legate printr-o linie în arbore, deci vor fi la nivelurile mai joase și, în consecință, vor avea codurile cele mai lungi, pe când caracterele cu probabilitate mare de apariție vor fi legate mai târziu la arbore, deci vor apărea pe niveluri mai mici, iar codurile lor asociate vor fi în consecință mai scurte.

Un cod binar peste un alfabet V dat, împreună cu o probabilitate $w: V \rightarrow [0, 1]$, cod asociat unui arbore binar strict cu ponderi construit cu algoritmul lui Huffman se numește *cod Huffman*.

Cele trei probleme formulate în general pentru coduri devin în acest context:

(a) *Construcția codului*: revine la aplicarea algoritmului lui Huffman pentru construirea unui arbore binar strict cu lungime externă ponderată minimă pentru mulțimea de frunze ponderate

$$E = \{(a, w(a)) \mid a \in V\}.$$

(b) *Codificarea*. Fiecare caracter $a \in V$ i se asociază codul constând din sirul de etichete al arcelor ce compun drumul de la rădăcină la frunza asociată caracterului a .

(c) *Decodificarea* unui sir de biți revine la parcurgeri repetitive ale către unui drum în arborele lui Huffman, începând de la rădăcină, conform convenției: în cazul în care caracterul (bitul) curent este 0, parcurgerea continuă pe fiul stâng, dacă este 1, parcurgerea continuă pe fiul drept. De fiecare dată când ajungem într-o frunză, am terminat de decodificat un caracter și reluăm parcurgerea de la rădăcină pentru restul sirului de biți. Dacă o asemenea parcurgere se termină într-un nod interior, atunci sirul $u \in \{0, 1\}^*$ de decodificat nu este valid, adică nu există nici un cuvânt $x \in V^*$, astfel încât $c(x) = u$.

Exerciții

1. Să se scrie codurile binare pentru alfabetul $\{a, b, c, d\}$ asociate figurilor (A), (B), (C), (D), (E) din secțiunea 4.3. Care dintre ele sunt coduri Huffman?
2. Care este lungimea minimă în biți a cuvintelor unui cod binar de lungime fixă, care să fie capabil să codifice un alfabet cu n caractere?
3. Scrieți un algoritm care să construiască un arbore Huffman pentru un alfabet cu ponderi date, arbore reprezentat în aşa fel încât să poată fi folosit atât la codificare, cât și la decodificare. Scrieți proceduri care fac, la cerere, codificarea și decodificarea.

4.5. Interclasarea optimală a mai multor siruri

Se dau n ($n > 2$) siruri ordonate crescător, S_1, S_2, \dots, S_n , cu lungimile respective l_1, l_2, \dots, l_n . Vrem să le interclasăm pe toate, obținând un sir S ordonat crescător ce conține toate elementele sirurilor inițiale. Putem face aceasta interclasând pe rând căte două siruri (operație descrisă în capitolul 4 secțiunea 7). La fiecare pas se reduce cu 1 numărul sirurilor sursă, deci vom obține în cele din urmă un singur sir. Dar cu ce preț?

Am văzut că fiecare operație de interclasare este costisitoare în termeni de mutări: interclasarea lui S_i cu S_j (să notăm rezultatul cu $S_i \& S_j$) ne costă $l_i + l_j$ mutări; iar dacă rezultatul se interclasează cu S_k , adică facem operația $(S_i \& S_j) \& S_k$ costul total va fi $2*(l_i + l_j) - l_k$.

Costul total al interclasării a n siruri depinde de *strategia de interclasare* folosită, adică de ordinea în care interclasăm două căte două sirurile.

Să ilustrăm cu un exemplu. Fie sirurile S_1, S_2, S_3 de lungime 90, 40 și respectiv 10. Să considerăm următoarele două strategii de interclasare, cu costurile lor:

(1) $(S_1 \& S_2) \& S_3$ are costul total

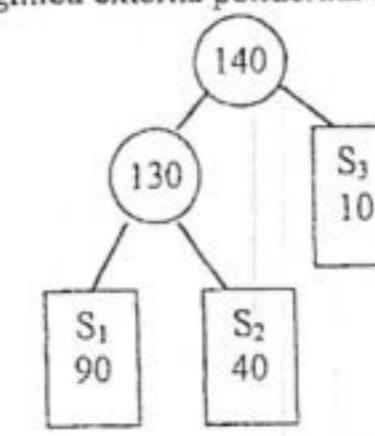
$$(90+40)+((90+40)+10)=(90+40)*2+10=270.$$

(2) $(S_2 \& S_3) \& S_1$ are costul total

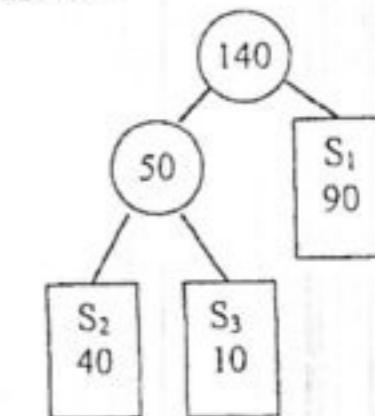
$$(40+10)+((40+10)+90)=(40+10)*2+90=190.$$

Deci a doua strategie este mult mai performantă decât prima.

Observăm că fiecare din strategiile de mai sus îi putem asocia un arbore binar strict cu ponderi, iar costul total al strategiei va fi lungimea externă ponderată a arborelui asociat.



$$L_E = 2*90 + 2*40 + 10 = 270$$



$$L_E = 2*40 + 2*10 + 90 = 190$$

În general, fiecărei strategii de interclasare a n şiruri i se asociază un arbore binar strict cu ponderi în care frunzele sunt asociate şirurilor S_i , $i=1, \dots, n$, iar ponderile sunt lungimile l_i .

Lungimea externă ponderată a arborelui asociat unei strategii va fi exact costul total în număr de mutări al respectivei strategii. Acest cost se minimizează dacă arborele asociat strategiei este arborele Huffman.

TESTUL I

✓ **Problema 1 [G1]**

Se dau două liste simplu înlățuite către care indică pointerii l_1 și l_2 având câmpul de legătură `next` și câmpul informației `info`.

Se dă următoarea secvență de instrucțiuni:

```
p1:=l1; p2:=l2;
while (p1<>nil) and (p2<>nil) and (p1^.info=p2^.info) do
begin p1:=p1^.next; p2:=p2^.next; end;
if ..... then
  write(' Listele sunt identice!')
else
  write(' Listele nu sunt identice');
```

Ce condiție trebuie pusă unde este spațiul liber astfel încât programul să decidă dacă listele sunt sau nu identice?

- [A] $(p1^.info=p2^.info)$
- [B] $(p1^.info=p2^.info)$ and $(p1^.next=p2^.next)$
- ✓ [C] $(p1=nil)$ and $(p2=nil)$
- [D] $(p1^.next=p2^.next)$
- [E] $(p1^.next^.info=p2^.next^.info)$

✓ **Problema 2 [G1]**

Câte interschimbări sunt făcute pentru a ordona **crescător** folosind algoritmul de sortare prin interschimbare directă un vector de n elemente ordonate **descrescător**.

- | | | | |
|----------------|-----------|--------------------------|------------------------|
| [A] n | [B] n^2 | ✓ [C] $\frac{n(n-1)}{2}$ | [D] $\frac{n(n+1)}{2}$ |
| [E] $\log_2 n$ | | | |

Problema 3 [G1]

Se dă secvența: 88 99 66 33 44 22.

Să se afle care este rezultatul aplicării procedurii de partitie din cadrul sortării rapide (QUICK SORT) având ca pivot valoarea 88 (de pe prima poziție).

168

- [A] 44; 22; 66; 33; 88; 99.
 [C] 22; 44; 66; 33; 88; 99.
 [E] 99; 44; 66; 33; 88; 22.

- [B] 22; 44; 33; 99; 66; 88.
 [D] 66; 88; 99; 44; 22; 33.

169

Problema 6 [G1]

Care din următoarele numere este încază cel mai corect adâncimea unui arbore binar complet pe niveluri cu n noduri? Justificați răspunsul.

- [A] n [B] $n \cdot \log_2 n$ [C] $\frac{n(n+1)}{2}$ [D] $\log_2 n$
 [E] n^2

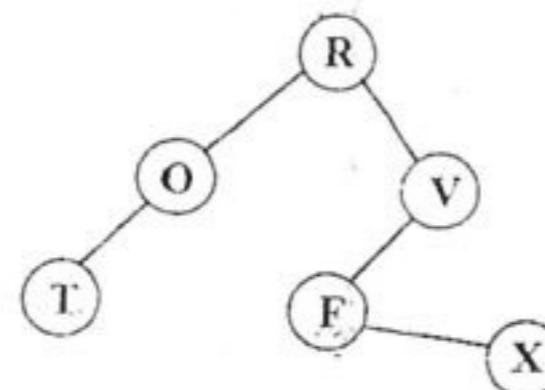
Problema 4 [G1]

Să se indice care din următorii vectori reprezintă un ansamblu. Să se specifică tipul lui (max-ansamblu sau min-ansamblu).

- [A] 40; 35; 33; 25; 23; 31; 30; 11.
 [B] 40; 33; 31; 25; 23; 30; 35; 11.
 [C] 11; 25; 22; 30; 33; 31; 35; 40.
 [D] 11; 22; 25; 35; 33; 30; 31; 40.
 [E] 11; 22; 30; 33; 35; 31; 25; 40.

Problema 5 [G1]

Se consideră următorul arbore binar:



Care sunt parcurgerile arborelui în preordine, inordine și postordine?

- / [A] pre: R O T V F X
 in: T O R F X V
 post: T O X F V R

- [B] pre: O R T V F X
 in: T O R F X V
 post: T O X F V R

- [C] pre: R O T V F X
 in: O T R F X V
 post: T O X F V R

- [D] pre: R O T V F X
 in: O T R F X V
 post: T O F X V R

- [E] pre: R O T V F X
 in: T O R F X V
 post: O T X F V R

Problema 7 [Gn]

Care din următorii vectori sunt obținuți în urma execuției a 3 (trei) pași iterativi din algoritm de sortare prin interschimbare (pentru variantele corecte precizați direcția paselor).

- [A] 11; 22; 55; 33; 77; 44; 66;
 55; 44; 77. [B] 22; 33; 44; 66;
 [C] 11; 77; 22; 33; 44; 55; 66.
 55; 44; 77. [D] 11; 22; 33; 66;
 [E] 22; 44; 33; 11; 55; 66; 77.

Problema 8 [Gn]

Se dă vectorul: 22 33 55 44 88 77 66.

Care din următoarele afirmații sunt adevărate?

- [A] Vectorul a fost obținut prin aplicarea a 2 (doi) pași iterativi ai algoritmului de sortare prin selecție directă a maximului.

- [B] Vectorul a fost obținut prin aplicarea a 2 (doi) pași iterativi ai algoritmului de sortare prin selecție directă a minimului.

- [C] Vectorul a fost obținut prin aplicarea a 3 (trei) pași iterativi ai algoritmului de sortare prin selecție directă a minimului.

- [D] Vectorul a fost obținut prin aplicarea a 2 (doi) pași iterativi ai algoritmului de sortare prin interschimbare directă.

- [E] Vectorul a fost obținut după aplicarea a 2 (doi) pași iterativi ai algoritmului de sortare prin inserție directă.

Problema 9 [Gn]

Fie o listă simplu înlățuită cu câmpul de legătură **next**. Se dă un pointer $p \neq \text{nil}$ către un nod al listei. Care din următoarele secvențe de instrucțiuni realizează adăugarea unui nod cu informația k după nodul p .

- [A] new(q);
 $q^.info:=k;$
 $q^.next:=p;$
 $p^.next:=q;$
- [C] new(q);
 $q^.info:=k;$
 $q^.next:=p^.next;$
 $p^.next:=q;$
- [E] new(q);
 $q^.next:=p^.next;$
 $q^.info:=k;$
 $p^.next:=q^.next;$

- [B] new(q);
 $q^.next:=p^.next;$
 $p^.next:=q;$
 $q^.info:=k;$
- [D] new(q);
 $q^.next:=k;$
 $q:=p^.next;$
 $p^.next:=q;$

Problema 13[D]

Să se construiască arborele binar corespunzător următoarelor parcurgeri:
inordine: B C E D R T S U
postordine: E D C B T U S R

Problema 14 [D]

Se consideră schema de codificare Hoffman binară peste alfabetul $A = \{A, B, C, D, E\}$. Probabilitățile de apariție ale literelor din A sunt:

Literă	Probabilitatea
A	28 %
B	9 %
C	21 %
D	10 %
E	32 %

- (a) Să se construiască arborele Hoffman asociat acestor date respectând regulile:
1. La compunerea a doi arbori cel cu probabilitatea mai mică va deveni arbore stâng.
2. La etichetarea arborelui se folosește cifra 0 pentru subarborele stâng și cifra 1 pentru cel drept.
(b) Să se decodifice (folosind arborele construit la punctul (a)) următoarea secvență de biți: 1001100011110

Problema 10 [D]

Se dau parcurgerile în preordine și inordine ale unui arbore binar:

preordine: B C D A F E

inordine: D C B A E F

Care este arborele binar corespunzător?

Problema 11 [D]

Se consideră un arbore binar de căutare, inițial vid. Arborele va avea în subarborele drept chei mai mari decât cheia din rădăcină. Se introduc în arbore următoarele chei unice în ordinea de mai jos (de la stânga la dreapta):

44 33 11 99 77 22 88

Care este arborele rezultat după introducerea cheilor?

Problema 12 [D]

Să se construiască un arbore binar echilibrat AVL din următoarele chei introduse pe rând. Să se ilustreze pașii intermediari, mai ales re-echilibrările.

37 19 30 25 28 32

Problema 15 [D]

Care sunt toate numerele naturale care pot reprezenta numărul total de noduri ale unui arbore binar strict? Justificați răspunsul.

Problema 16 [D]

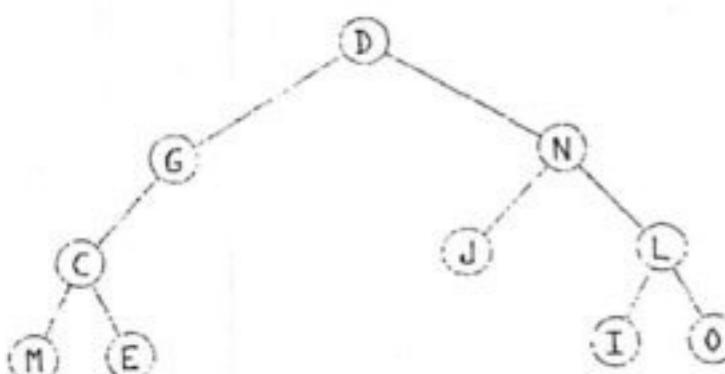
Să se construiască un min-ansamblu din următoarele chei inserate pe rând. Să se ilustreze pașii intermediari.

77 55 33 88 66 44

172

TESTUL II✓ **Problema II.1 [G1]**

Se consideră următorul arbore binar:



Care sunt parcurgerile arborelui în preordine, inordine și postordine?

- | | | |
|-----|-------------|---------------------|
| (A) | Preordine: | D G C M E N J L I O |
| | Inordine: | M C E D G J N I L O |
| | Postordine: | M E C G J I C L N D |
| (B) | Preordine: | D G M C E N J L I O |
| | Inordine: | M C E G D J N I L O |
| | Postordine: | M E C G J I O L N D |
| (C) | Preordine: | D G C M E N J L I O |
| | Inordine: | M C E G D J N I L O |
| | Postordine: | E M C G J I O L N D |
| (D) | Preordine: | D G C M E N J L I O |
| | Inordine: | M C E G D J N I L O |
| | Postordine: | M E C G J I O L N D |
| (E) | Preordine: | D G C M E N J L I O |
| | Inordine: | M C E G D J N I L O |
| | Postordine: | E M C G J I O L N D |

✓ **Problema II.2 [D]**

Se consideră un arbore binar de căutare, inițial vid. Arborele va avea în subarborele drept chei mai mari decât cheia din rădăcină. Se introduc în arbore următoarele chei unice în ordinea de mai jos (de la stânga la dreapta):

4 7 2 0 1 5 6 3

Care este arborele rezultat după introducerea cheilor?

✓ **Problema II.3 [D]**

Se dau parcurgerile în preordine și inordine ale unui arbore binar:

preordine: A C D B F H J I G

inordine: D C F B H A I G J

Care este arborele binar corespunzător?

✓ **Problema II.4 [D]**

Să se construiască un min-ansamblu din următoarele chei inserate pe rând. Să se ilustreze pașii intermediari.

66, 44, 22, 71, 60, 33

Problema II.5 [D]

Se dă o listă circulară având primul nod dat de pointerul *p* și o cheie *intreagă k*.

Să se scrie un program PASCAL care să adauge cheia *k* la listă (indiferent pe ce poziție) cu condiția ca *k* să nu fie informația niciunui nod din lista circulară.

✓ Problema II.6 [D]

Să se constuiască un arbore binar echilibrat AVL din următoarele chei introduse pe rând. Să se ilustreze pașii intermediari, mai ales re-echilibrările.

27, 9, 20, 15, 18, 17

✓ Problema II.7 [G1]

Se consideră schema de codificare Huffman binară peste alfabetul A = {C, N, X, A, H, U, F, T}. Probabilitățile de apariție ale literelor din alfabet sunt:

Literă	Probabilitatea
C	25%
N	15%
X	14%
A	3%
H	6%
U	17%
F	1%
T	19%

- Arborele binar Huffman se construiește respectând următoarele reguli:
- La compunerea a doi arbori cel cu probabilitatea mai mică va deveni subarbore stâng.
 - La etichetarea arborelui se folosește cifra 0 pentru subarborele stâng și cifra 1 pentru subarborele drept.

Se consideră următoarea secvență de litere din alfabetul A:

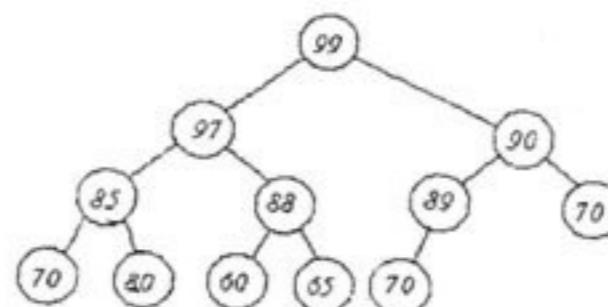
H N T N C F A H

Care este secvența codificată binară?

- [A] 010111000110100100001001111
- [B] 010111000110100100001010101
- ✓ [C] 0101110001101001000010010101
- [D] 01011100011011001000010010101
- [E] 0101110000111001000010010101

✓ Problema II.8 [D]

Se dă următorul max-ansamblu. Să se extragă, pe rând, două chei de valoare maximă, și să se ilustreze structura de ansamblu a cheilor rămase.



✓ Problema II.9 [G1]

Se consideră două stive S1 și S2 și o coadă C având ca elemente de informație literele alfabetului. Se definesc trei tipuri de operații:

- 'x' litera 'x' se introduce în stiva S1;
- 1 dacă stiva S1 este nevidă se scoate o literă din S1 și se introduce în stiva S2, altfel nu se execută nimic;
- 2 dacă stiva S2 este nevidă se scoate o literă din S2 și se introduce în coada C, altfel nu se execută nimic.

Atât stivele S1 și S2 cât și coada C sunt considerate de capacitate infinită, iar inițial sunt vide.

Se dă o secvență de operații. Să se determine conținutul cozii C după execuția operațiilor.

Coada C este reprezentată cu punctul de intrare în dreapta și cu punctul de ieșire în stânga.

Secvența de operații este:

2 Y B V 1 W 1 Q U L K 1 1 1 2 1 1 2 2 2 2 2

Care este conținutul cozii C după executarea operațiilor?

- [A] B V Y W Q U
- [B] Y W Q B U L
- [C] U Y B Q L K W
- [D] Y V B Q L U W
- [E] Y Q W V U K B

BIBLIOGRAFIE:

1. A. V. Aho, J. E. Hopcroft, J. D. Ullman: "Data Structures and Algorithms", Addison-Wesley Publ. Comp., 1983
2. S. Baase: "Computer Algorithms: Introduction to Design and Analysis", Addison-Wesley Publ. Comp., 1988
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest: "Introduction to Algorithms", The MIT Press, 1990
4. E. Horowitz, S. Sahni: "Fundamentals of Data Structures in Pascal", Computer Science Press, 1984
5. D. E. Knuth : "The Art of Computer Programming" vol. 1, "Fundamental Algorithms", Reading, Massachusetts, 1969; vol.3, "Sorting and Searching", Addison-Wesley, 1973" (sau versiunile în limba română: (1) "Tratat de programarea calculatoarelor", vol I "Algoritmi fundamentali", Editura Tehnică Bucureşti, 1974, și vol. III "Sortare și căutare", Ed. Tehn. Buc. 1976, (2) "Arta programării calculatoarelor", vol. I "Algoritmi fundamentali", TEORA, 2000, și vol. 3 "Sortare și căutare", TEORA, 2001)
6. L. Livovschi, H. Georgescu: "Sinteză și analiza algoritmilor", Ed. Științifică și Enciclopedică, Bucureşti, 1986
7. L. Nyhoff, S. Leestma: "Advanced programming in Pascal with Data Structures", MacMillan Publ. Comp. N.Y., 1988
8. I. Tomescu: "Data Structures", Editura Univ. din Bucuresti, 1997
9. N. Wirth: "Algorithms + Data Structures = Programs", Prentice Hall Inc., 1976

Erată

p. 20, r. 1: Se adaugă cuv. subliniat „...C(n) numărul maxim de comparații...”

p. 20, r. 7: Se înlocuiește „C(n) = ...” cu „C(n) \geq ...”

p. 94, rândurile 6–11 de jos se înlocuiesc cu:

```
if r.t.bal = 1 then q.t.bal := -1  
else q.t.bal := 0;  
if r.t.bal = -1 then p.t.bal := 1  
else p.t.bal := 0;
```

p. 95, rândurile 8–11 de jos se înlocuiesc cu:

```
if r.t.bal = -1 then q.t.bal := 1  
else q.t.bal := 0;  
if r.t.bal = 1 then p.t.bal := -1  
else p.t.bal := 0;
```

p. 107, r. 2 de jos: formula este „... $(\ln \gamma + c)$ ”

p. 110, r. 11 de jos: se adaugă ceea ce e subliniat „i := n; k := n;”