1. ALGORITMI DE SORTARE ŞI CĂUTARE

1.1.SORTARE

Problema sortării este următoarea.

Fie M o mulțime de elemente şi o relație de ordine pe această mulțime (în cazurile practice, în general, mulțimea R, eventual submulțimi ale acesteia). Fie $A = \{a_1, ..., a_n\} \subset M$ o submulțime. Să se ordoneze – crescător sau descrescător – elementele lui A.

Problema sortării poate fi formulată sub o formă mai generală:

Se dă o mulțime de articole (înregistrări) conținând acelaşi număr şi tip de informații. Fie structura (articolul) generic $R = \{R_1, ..., R_p\}$. Dintre aceste informații ne interesează, spre exemplu, informația r_i , $i \in \{1, ..., p\}$ pe care o vom numi *cheie* (vom nota mai departe cheile cu litera *k*). Dacă cheile articolelor sunt comparabile ca valoare, atunci să se aranjeze articolele din mulțime aşa încât cheile lor să fie în ordine crescătoare, spre exemplu. Vom obține ordinea:

$$R_1, R_2, ..., R_n \quad \text{astfel încât:} \quad k_1 \le k_2 \le ..... \le k_n$$

În continuare vom considera o mulțime de chei $\{k_1, ..., k_n\}$ pe care le vom ordona crescător prin mai multe metode.

1.1.1. Sortarea prin enumerare

Fiecare cheie este comparată cu toate celelalte numărându-se câte sunt mai mici decât cheia curentă. În felul acesta se găseşte poziția pe care trebuie trecută fiecare cheie. Cheile sortate se vor obține într-un tablou S.

procedure Sortnum $(n,\ k; \text{var}\ S)$

$k\ [1...n]$: tabloul cheilor de sortat

$S\ [1...n]$: tabloul cheilor sortate

$POZ\ [1...n]$: tabloul de numărare

$POZ\ [i]$: numărul de chei mai mici decât cheia $k\ [i]$, $i = \overline{1,\ n}$

```
for i=1 to n do POZ[i] ← 0 next i;
for j=2 to n do
    for i=1 to j-1 do
```

if $k\ [i] < k\ [j]$ **then** $POZ\ [j] = POZ\ [j] + 1$

else $POZ\ [i\ ] = POZ\ [i\ ] + 1;$

end if

next i

next j

for *i=1* **to** *n* **do**

$S\ [POZ\ [i] + 1] \leftarrow k\ [i];$

end_proc.

1.1.2. Sortare prin inserție directă

Înainte de a examina cheia k_j, $j = \overline{2, n}$ considerăm $k_1 < k_2 < ... < k_{j-1}$ şi vom insera pe k_j printre cheile $k_1 ... k_{j-2}$ în locul care îi revine. Vom compara pe rând k_j cu $k_{j-1}, k_{j-2}, ..., k_j$ până când ajungem la prima cheie k_i, $i = \overline{1, j - 1}$ astfel încât $k_i < k_j$. În acest caz vom insera pe k_j pe poziția $i + 1$. Operația de inserare va presupune şi o deplasare spre dreapta a cheilor. Deplasările la dreapta vor fi executate în acelaşi timp cu comparațiile.

procedure Insertdir (n, var, k)

$k\ [1...n\ ]$: tabloul cheilor; la sfârşit vor fi în ordine crescătoare.

R: valoare pentru cheia curentă.

for *j=2* **to** *n* **do**

$R \leftarrow k\ [j\ ]$; $i \leftarrow j - 1$; continuă ← true;

while $(i > 0)$ and (continuă) do {Se compară R cu k_i, $i = j-1, ..., 1$}

if $R \geq k\ [i\ ]$ **then**

continuă ← false {S-a găsit poziția $(i+1)$ pentru R}

else

$k\ [i + 1] \leftarrow k\ [i\ ];$

$i \leftarrow i - 1;$

end if

end while

$k\ [i + 1] \leftarrow R$;{Inserează R pe poziția $i + 1$; dacă $i = 0$ atunci R este cea mai mică cheie până la pasul curent}.

next j

end_proc

1.1.3. Sortare prin interschimbare

1.1.3.1. Metoda bulelor

Se compară, la primul pas, k_1 cu k_2, k_2 cu $k_3,\ldots,k_{n-1}$ cu k_n şi se realizează interschimbări unde este necesar. Se constată că cheile cu valori mari tind să se deplaseze spre dreapta. După prima trecere, pe poziția n se va plasa cheia cu valoarea cea mai mare. La pasul următor comparațiile se vor face de la k_1 cu k_2 până la k_{n-2} cu k_{n-1}. Aceşti paşi se execută până când se constată că nu mai sunt interschimbări.

procedure Interschimb (n, var, k)

$k\ [1...n]$: tabloul cheilor

Max: indice în tabloul k

$Max \leftarrow n$:{indică poziția pe care va fi aşezată cea mai mare cheie dintre cele care nu şi-au găsit locul}.

repeat

$t \leftarrow 0;$

for *i=1* **to**, $Max - 1$ **do**

if $k\ [i] > k\ [i]$ **then**

$k\ [i] \leftrightarrow k\ [i+1];$

$t \leftarrow i;$

end if

next i

$Max \leftarrow t;$

until $Max = 0;$

end Sort interschimb.

1.1.3.2. Metoda sortării rapide

Pentru cheile $k_1,\ldots,k_n$ se porneşte cu doi indici, inițial $i = 1$ şi $j = n$ comparându-se k_i cu k_j. Dacă nu este necesar interschimbul ($k_i > k_j$) se majorează j cu $1\ (j \leftarrow j - 1)$ repetându-se procesul prin micşorarea lui j. Dacă apare un interschimb ($k_i > k_j$) se măreşte i cu $1\ (i \leftarrow i + 1)$ repetându-se procesul de

comparare, mărind pe i până la apariția unui nou interschimb. apoi se micşorează din nou j continuându-se procesul de *"ardere a lumânării la ambele capete"* până când $i = j$.

Vom folosi metoda generală *"Divide et impera"*. Presupunem că pentru doi indici $1 \le 1 < r \le n$ vom aranja secvența de chei $\{k_1, \ldots, k_r\}$ astfel încât cheia k_1 va fi pusă pe o poziție $P \in \{1, 1+1, \ldots, r-1, r\}$ şi $\begin{cases} \forall\ i \in \{1, 1+1, \ldots, p-1\}\ k_i \le k_p \\ \forall\ j \in \{p+1, \ldots, r\}\ k_p \le k_j \end{cases}$. Acest proces va putea continua pentru secvențele de chei $\{k_1, \ldots, k_{p-1}\}$ şi $\{k_{p+1}, \ldots, k_r\}$ independent una de alta.

procedure Poziționare $(n, 1, r;\ \text{var}\ k,\ p)$

$k\ [1 \ldots n]$: tabloul cheilor

$l,\ r$: indici în tablou cu semnificația de mai sus.

p: indice cu semnificația de mai sus.

$i \leftarrow 1;\ j \leftarrow r;\ inc\_i \leftarrow 0;\ dec\ j \leftarrow 1;$

while $i < j$ **do**

if $k\ [i] > k\ [j]$ **then** $k\ [i] > k\ [j];\ inc\ i \leftarrow dec\ j;$

end if

$i \leftarrow inc\ i + 1;\ j \leftarrow j\ dec\ j;$

end while

$p \leftarrow i;$

end_proc.

Algoritmul de sortare rapidă porneşte cu perechea $(1,\ r) = (1,\ n)$ încercând poziționare $(n,\ 1,\ r,\ k,\ p)$. Va continua procesul cu una din perechile $(1,\ p-1)$ sau $(p+1,\ r)$ cealaltă fiind introdusă într-o stivă S. O intrare în stiva S, de forma $(a,\ b)$ reprezintă o cerere de sortare a subsecvenței $\{k_a, \ldots, k_b\}$ la un anumit timp, în viitor. Procesul de sortare poate continua doar dacă perechea curentă $(1,\ r)$ are proprietatea $l < r$. Când $l = r$ procesul va continua cu perechea $(1,\ r) = (a,\ b)$, unde $(a,\ b)$ este prima pereche din vârful stivei. Când stiva devine vidă, procesul de sortare este complet încheiat.

procedure Quicksort $(n,\ \text{var}\ k)$

$k\ [1 \ldots n]$: tabloul cheilor

S: stiva menționată mai sus. Poate avea cel mult $\log_2 n$ intrări.

$S \Leftarrow \Phi$; $l \leftarrow 1$; $r \leftarrow n$; continuă ← true;

repeat

while $l < r$ **do**; Se plasează în stivă secvența cea mai lungă

Poziționare (n, l, r, k, p);

if $r - p \geq p - 1$ **then** $\{(p + 1, r) \Rightarrow S; r \leftarrow p - 1\}$

else $\{(1, p - 1) \Rightarrow S; l \leftarrow p + 1\}$

end if

end while

if $S = \Phi$ **then** continuă ← false

else $(l, r) \Leftarrow S$;

end if

until (not continuă);

end_proc;

1.2.CAUTARE

Problema căutării este următoarea: presupunând că s-au memorat $n \in N$ înregistrări, se cere să se localizeze o anumită înregistrare; în cazul când aceasta nu este găsită se va răspunde printr-un mesaj corespunzător. Ca şi în cazul sortării se presupune că fiecare înregistrare posedă un câmp special denumit cheia înregistrării, iar căutarea făcându-se în funcție de această cheie. Practic, dacă W este cheia căutată ea trebuie localizată printre cheile $k_1, ..., k_n$. În continuare vom ignora acțiunile care se execută după ce cheia k a fost găsită.

1.2.1. Căutarea secvențială

Pentru $i = \overline{1, n}$ se testează identitatea $W = k_i$. În caz de egalitate algoritmul se termină cu succes. Altfel, când $i = n + 1$, algoritmul se termină cu insucces. Remarcăm faptul că în acest caz decizia are două variante de continuare.

procedure Caut_secvența_1(W, k; găsit)

W: cheia căutată

$k\ [1...n]$: tabloul cheilor

găsit: variabila booleană

găsit ← false; $i \leftarrow 1$;
while ($i \le n$ and not găsit) **do**
if $W = k_i$ **then** găsit ← true
else $i \leftarrow i + 1$
end if
end while
end_proc.

O variantă mai rapidă de căutare secvențială este cea în care, printre cheile $k_1, ..., k_n$, se introduce o cheie fictivă $k_{n+1} = W$.

procedure Caut_secvența_2 (W, k; găsit)
găsit ← false; $i \leftarrow 1$; $k_{n+1} \leftarrow W$;
while $W \neq k_i$ **do**
$i \leftarrow i + 1$;
end while
if $i \le n$ **then** găsit ← true;
end if
end_proc.

Căutarea secvențială într-un tabel ordonat de chei

Vom presupune că înregistrările $R_1, ..., R_n$ sunt astfel memorate încât $k_1 < k_2 < ... < k_n$. Pentru a găsi înregistrarea cu cheia W vom introduce în tabloul de chei o cheie $k_{n+1} = \infty$, $\left(k_{n+1} > k_i,\ i = \overline{1,\ n}\right)$ şi $W < \infty$ (eventual $k_{n+1} = \max\{k_n, W\}$).

procedure Caut_secvența_3 (W, k; găsit)
găsit ← false; $i \leftarrow 1$; $k_{n+1} \leftarrow \infty$;
while $W > k_i$ **do**
$i \leftarrow i + 1$;
end while
if $W = k_i$ **then** găsit ← true;
end if
end_proc.

1.2.2. Căutarea prin comparație de chei

Acest tip de căutări presupune o ordonare liniară a cheilor (exemplu: ordonare lexicografică pentru chei de tip şir de caractere sau numerică pentru chei numerice). În continuare vom considera ordonarea numerică crescătoare a cheilor: $k_1 < k_2 < ... < k_n$. În cazul acestor tipuri de căutări decizia se ia între trei variante. Se compară cheia W cu cheia k_i, $i = \overline{1, n}$ şi căutarea continuă după una din variantele: $W < k_i$, $W = k_i$, $W > k_i$.

1.2.2.1. Căutarea binară

Se compară cheia căutată W cu cheia care se găseşte la mijlocul tabelului de chei $k_1 < k_2 < ... < k_n$. Ca rezultat al acestei comparaţii se indică în care jumătate de tabel se continuă căutarea. Se determină noul mijloc şi se repetă comparaţia. După cel mult $\log_2 n$ comparaţii se va găsi cheia căutată sau se va constata că lipseşte din tabel.

procedure Căutare_binară (W, k; găsit)
  $k\ [1...n]$: tabelul cheilor; $k_1 < k_2 < ... < k_n$
  găsit ← false; $i \leftarrow 1$; $u \leftarrow n$;
  while $1 \le u$ and (not găsit) do
    $i \leftarrow \left[\frac{(1 + u)}{2} \right]$;
    if $W < k_i$ **then** $u \leftarrow i - 1$
    else if $W > k_i$ **then** $l \leftarrow i + 1$
      else găsit ← true;
      end if
    end if
  end while
end_proc

***Exemplu*:** căutarea cheii *k* = 653

k = 61, 87, 154, 426, 512, [[610, [653]], 703, 897, 908]

④

653

Căutarea binară poate fi exemplificată prin intermediul unui *arbore binar de căutare*. Nodurile unui astfel de arbore pot fi înregistrările $R_1, ..., R_n$ sau cheile

asociate lor. Pentru uşurinţă vom considera nodurile arborelui binar etichetate cu cheile $k_1, \ldots, k_n$ $(k_1 < k_2 < \ldots < k_n)$.

Arborele binar corespunzător unei căutări binare se construieşte astfel:

– rădăcina arborelui va fi etichetată cu $K_{\left\lceil \frac{n}{2} \right\rceil}$;

– subarborele stâng va avea $\left\lceil \frac{n}{2} \right\rceil - 1$ noduri etichetate cu $k_1, \ldots, k_{\left\lceil \frac{n}{2} \right\rceil - 1}$;

– subarborele drept va avea $\left\lceil \frac{n}{2} \right\rceil$ noduri etichetate cu $k_{\left\lceil \frac{n}{2} \right\rceil + 1}, \ldots, k_n$;

– pentru cele două subintervale (submulţimi) de chei se procedează la fel, determinându-se mijlocul ca fiind rădăcina subarborelui respectiv.

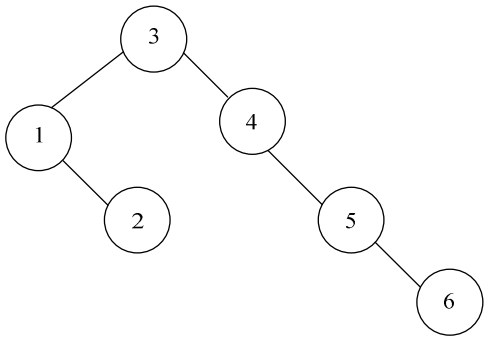
***Observaţie*:** Se poate folosi unul din algoritmii de inserare într-un arbore binar.

Căutarea într-un astfel de arbore se realizează cu secvenţa de instrucţiuni corespunzătoare desprinse din algoritmul de căutare şi inserare prezentat la arbori binari.

Un arbore binar de căutare, creat în acest fel, are următoarea proprietate: pentru orice nod, numărul de niveluri din subarborele stâng şi subarborele drept diferă cu cel mult o unitate.

Un arbore cu o astfel de proprietate poartă numele de *arbore echilibrat*. În felul acesta, numărul de căutări în subarborele stâng respectiv drept va diferi cu cel mult o căutare.

***Exemplu*:** $k = \{1, 2, 3, 4, 5, 6\}$



Căutarea cheii 2 necesită 3 comparaţii

Căutarea cheii 6 necesită 4 comparaţii

Căutarea cheii 1.5 necesită 3 comparaţii

Căutarea cheii 4.5 necesită 3 comparaţii

Pentru cele ce vor urma, vom transforma arborele binar de căutare într-un *arbore binar strict*.

***Definiție*:** *Un arbore binar este strict dacă orice nod neterminal are doi descendenți.*

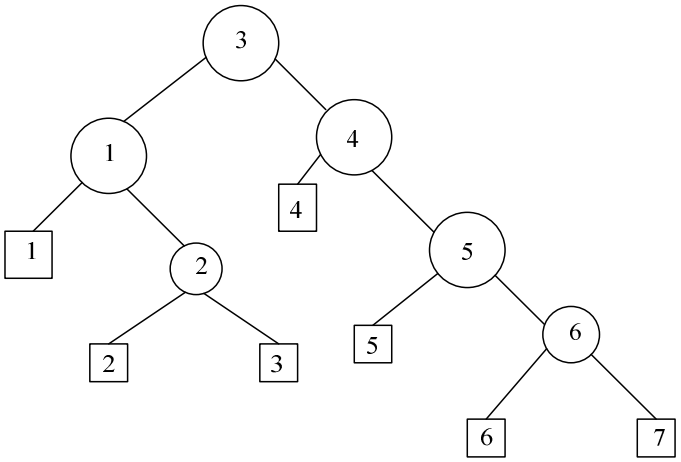
Arborele binar strict îl vom obține în felul următor:

– pentru orice nod k_i care nu are descendent stâng vom adăuga un astfel de descendent pe care îl vom nota cu k_j (nod *"pătratic"*);

– pentru orice nod k_i care nu are descendent drept vom adăuga un astfel de descendent pe care îl vom nota cu k_{j+1};

Nodurile din arborele binar original le vom nota cu k_i, $i = \overline{1, n}$ (noduri *"circulare"*).

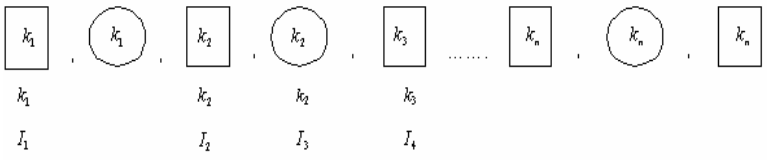
***Observație*:** Nodurile circulare pot fi numite *"noduri interne"* iar cele pătratice *"noduri externe"*.



În felul acesta căutarea cheii 1.5 se poate încheia cu răspunsul că ea se află în intervalul (1, 2), sau cheia 6.3 este în intervalul (6, 7).

În continuare, prin arbore binar de căutare ataşat mulțimii de chei $k_1 < k_2 < ... < k_n$ vom înțelege arborele binar strict corespunzător.

Parcurgerea în inordine a unui astfel de arbore va furniza secvența:



Cele $n + 1$ noduri terminale (pătratice), $k_1, ..., k_n$, corespund în inordine intervalelor $I_1, ..., I_{n+1}$:

$I_j = k_{j-1}, k_j, \ j = \overline{1, n+1}.$

Cunoaştem problema căutării cheii W în tabloul $(k_1, \ldots, k_n)$ folosind strategia reprezentabilă prin arbori binari (vezi cursul de arbori). Acest tip de căutări s-au executat (deşi nu am spus-o) în condițiile în care frecvențele de apariție ale valorilor $k_1, \ldots, k_n$ au fost considerate egale şi la fel frecvențele intervalelor $I_1, \ldots, I_{n+1}$. În continuare vom generaliza problema pentru cazul probabilităților oarecare date.

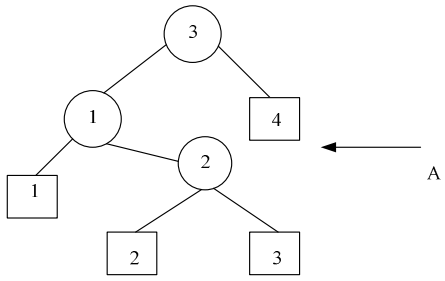
1.2.3. *Arbori binari de căutare optimi*

Presupunem că probabilitatea ca $W = k_j$ este $p_i, \ j = \overline{1, n}$. Probabilitatea ca $W \in I_j$ este $q_j, \ j = \overline{1, n+1}$ şi este satisfăcută relația $\sum_{j=1}^{n} p_j + \sum_{j=1}^{n+1} q_j = 1.$

Dorim să găsim un arbore binar de căutare, cu nodurile etichetate $k_1, \ldots, k_n$ aşa încât numărul de comparații din timpul căutării să fie minim. Practic, dacă notăm cu A arborele binar de căutare şi cu $\rho(A) = \sum_{j=1}^{n} p_j \cdot niv_j + \sum_{j=1}^{n+1} q_j \cdot (niv_j - 1)$ ne interesează să găsim acel arbore A pentru care valoarea $\rho(A)$ este minimă. În acest caz arborele binar de căutare, A, va fi optim.

Principiul care stă la baza rezolvării acestei probleme este următorul: *"toți subarborii unui arbore optim sunt optimi"*.

***Exemplu*:**



Dacă A este optim pentru ponderile: $(p_1, p_2, p_3; q_1, q_2, q_3, q_4)$, atunci subarborele din stânga rădăcinii circulare *3* este şi el optim pentru $(p_1, p_2; q_1, q_2, q_3)$.

Notăm cu $C_{(i,\ j)}$ costul unui subarbore optim determinat de secvența de noduri $\{k_1, k_{j+1}..., k_{j-1}\}$. Ponderile corespunzătoare acestui subarbore sunt: $(p_i, \ldots, p_{j-1}; q_i, \ldots, q_j)$.

Fie suma acestor ponderi. Costul minim al unui arbore binar de rădăcină k_1, $i \le 1 \le j-1$ este $w(i,j) + C(i,l) + C(1+l, j)$, unde $C(i,l)$ şi $C(l+1, j)$ sunt costurile subarborelui stâng şi drept ai arborelui de rădăcină k_1. Atunci:

$$(*)\begin{cases} C(i,\ i) = 0 \\ C(i,\ j) = \min\limits_{i \le l \le j} \{ w\ (i,\ j) + C\ (i,1) + C\ (l+1,\ j) \} \\ \qquad\qquad pentru\ \ i < j \end{cases}$$

Fie $R\ (i,\ j)$ mulțimea tuturor mărimilor l pentru care în $(*)$ se atinge valoarea minimă; această mulțime specifică rădăcinile posibile ale arborilor optimi.

Relațiile $(*)$, cu valorile de pornire $C\ (i,\ i) = 0$, fac posibil să evaluăm $C\ (i,\ j)$ pentru $m = j - 1 = 1, 2, 3, \ldots, n$.

Vom aplica metoda programării dinamice ca în cazul problemei de înmulțire optimă a matricilor. Vom calcula $w_{i,\ j}$ şi $C_{i,\ j}$ succesiv pentru $m = j - 1 = 1, 2, 3, \ldots, n$. Vom reține în $C\ (i, j)$, $i < j$ valoarea 1 pentru care se realizează minimul relației $(*)$.

procedure Generare_arbore_optim $(n,\ p,\ q;$ var $C,\ w)$

$C\ [1\ \ldots\ n + 1,\ 1\ \ldots\ n + 1]$: matricea costurilor

$w\ [1\ \ldots\ n + 1,\ 1\ \ldots\ n + 1]$: matricea sumei ponderilor

$p\ [1 \ldots n]$: ponderile $p_j,\ j = \overline{1,\ n}$

$q\ [1 \ldots n]$: ponderile $q_j,\ j = \overline{1,\ n + 1}$

for *i=1* **to** *n* **do**

$C\ (i,\ i) \leftarrow 0;$

$w_{ii} \leftarrow q_{\mathrm{i}}$

next i

for *m=1* **to** *n* **do** $\{n = j - 1\ este\ dif.\}$

for *i=1* **to** *n+1-m* **do**

$j \leftarrow i + m;$

$w_{i\,j} \leftarrow w_{i,\,j-1} + p_{j-1} + q_j;$

$C(i,\ j) = \min\limits_{i \le l \le j} \{ w_{i\,j} + C\ (i,l) + C(l+1,\ j) \}$

$C\ (i,\ j) \leftarrow 1$, {unde *l* este valoarea $i \le l < j$ pentru care se indeplineşte acest minim}

next i

next m

end_proc.

Dacă $C\ (1,\ n+1)$ este costul pentru arborele binar optim de căutare, atunci rădăcina lui va fi k_1, unde $l = C\ (1,\ n+1)$. Subarborele stâng al lui k_1 va avea nodurile etichetate $k_1, \ldots, k_{l-1}$ iar rădăcina lui (descendent stâng direct al lui k_1) va fi $k_r = C\ (1,\ 1)$. Subarborele drept al lui k_1 va avea nodurile etichetate $k_{l+1}, \ldots, k_{n+1}$, iar rădăcina lui (descendent drept al lui k_1) va fi $k_t = C_{(n+1,\ l+1)}$, etc.

function Arbore $(n,\ C,\ i,\ j)$: Întoarce *P* referință la nodurile arborelui

$C\ [1 \ldots n+1,\ 1 \ldots n+1]$: matricea costurilor

$i,\ j$: indici pentru nodurile subarborelui (i<j).

$l \leftarrow C\ (j,\ i)$;

Alocă spațiu pentru P

$INFO\ (P) \leftarrow k_1;\ LLINK\ (P) \leftarrow RLINK\ (P) \leftarrow \Phi$;

if $C\ (i,\ l) \neq 0$ **then**

$LLINK\ (P) \leftarrow ARBORE\ (C,\ i,\ l)$;

end if

if $C\ (l+1,\ j) \neq 0$ **then**

$RLINK\ (P) \leftarrow ARBORE\ (C,\ l+1,\ j)$;

end if

return *P*;

end_proc

Apel: $ARBORE\ (l,\ n+1) \rightarrow RAD$; RAD: referință la rădăcina arborelui.