

## 8. Arbori

### 8.1. Arboari generalizați

#### 8.1.1. Definiții

- În definirea noțiunii de arbore se pornește de la noțiunea de **vector**.
  - Fie **V** o mulțime având elementele  $a_1, a_2, \dots, a_n$ .
  - Pe mulțimea **V** se poate defini o astfel numită "relație de precedență" în felul următor: se spune că  $a_i$  precede pe  $a_j$  dacă  $i < j$ . Aceasta se notează:  $a_i \prec a_j$ .
  - Se poate verifica ușor că relația astfel definită are următoarele proprietăți, valabile pentru structura vector:

(1) Oricare ar fi  $a \in V$  avem  $a \not\prec a$ . (S-a notat cu  $\not\prec$  relația "nu precede");

(antireflexivitate)

(2) Dacă  $a \prec b$  și  $b \prec c$  atunci  $a \prec c$  (tranzitivitate); [ 8.1.1.a ]

(3) Oricare ar fi  $a \in V$  și  $b \in V$ , dacă  $a \neq b$  atunci avem fie  $a \prec b$  fie  $b \prec a$ .

- Din proprietățile (1) și (2) rezultă că relația de precedență nu determină în **V** "bucle închise", adică nu există nici o secvență de elemente care se preced două câte două și în care ultimul element este același cu primul, cum ar fi de exemplu  $a \prec b \prec c \prec d \prec a$ .
- Proprietatea (3) precizează că relația de precedență este definită pentru **oricare** două elemente  $a$  și  $b$  ale lui **V**, cu singura condiție ca  $a \neq b$ .
- Fie **V** o mulțime finită peste elementele căreia s-a definit o relație de precedență, stabilind referitor la fiecare pereche de elemente, care dintre ele îl precede pe celalălt.
  - Dacă această relație posedă proprietățile [8.1.1.a], atunci ea **imprimă** peste mulțimea **V** o **structură vector** (fig.8.1.1.a).

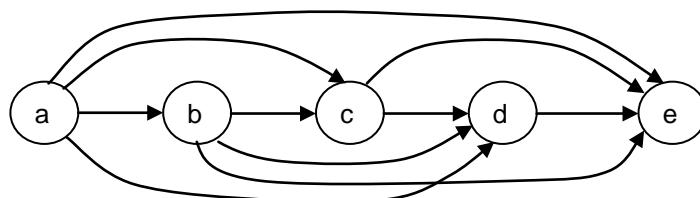
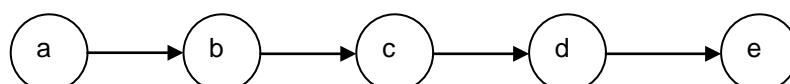


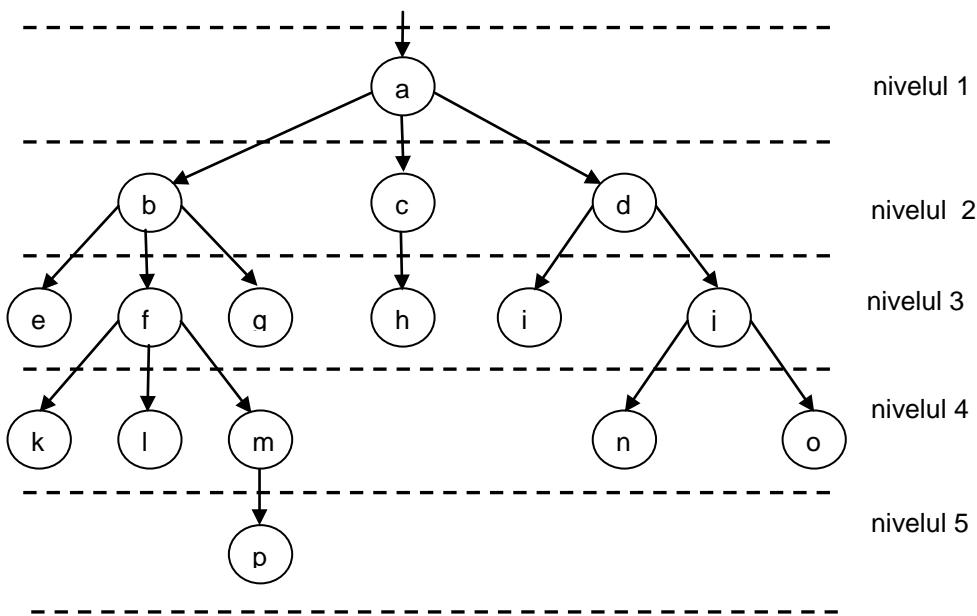
Fig. 8.1.1.a. Structură vector

- În figura 8.1.1.b apare o altă reprezentare intuitivă a unei structuri **vector**. Săgețile din figură indică relația "succesor".



### Fig.8.1.1.b. Relația succesor

- Relația ”**succesor**” se definește cu ajutorul relației de **precedență** după cum urmează:
  - Dacă între elementele  $a$  și  $b$  ale lui  $V$  este valabilă relația  $a \prec b$  și nu există nici un  $c \in V$  astfel ca  $a \prec c \prec b$  atunci se zice că  $b$  este **succesorul** lui  $a$ .
- Se observă că relația "succesor" (mulțimea săgeților din figura 8.1.1.b.), **precizează** relația "precedență" fără a fi însă **identică** cu ea.
  - Spre exemplu, există relația  $b \prec e$  (prin tranzitivitate), dar nici o săgeată nu conectează pe  $b$  cu  $e$ .
- În figura 8.1.1.c apare o așa numită **structură arbore** care se definește prin **generalizarea** structurii **vector**.



**Fig. 8.1.1.c. Structură arbore**

- Astfel, dacă în cazul **vectorului**, **toate elementele** cu excepția ultimului au **exact un succesor**, la **structura de arbore** se admite ca **fiecare element** să aibă un **număr oarecare de succesi**ri, inclusiv zero, cu **restrictia** ca două elemente distincte să **nu** aibă același succesor.
- Relația **succesor** definește o relație de **precedență** pe structura arbore. Astfel, din figura avem  $b \prec p$ ,  $d \prec n$ , etc.
- **Relația de precedență** definită pe structura arbore se bucură de proprietățile (1) și (2) de la [8.1.1.a] dar **nu** satisface proprietatea (3).
  - Într-adevăr în figura 8.1.1.c, pentru elementele  $b$  și  $c$  **nu** este valabilă nici una din relațiile  $b \prec c$  sau  $c \prec b$ , la fel pentru elementele  $d$  și  $k$ .
  - Prin urmare, relația de precedență este definită numai pentru **o parte** a perechilor de elemente ale arborelui, cu alte cuvinte este o **relație parțială**.
- În general, o **structură arbore** se definește ca o mulțime  $A$  de  $n \leq 0$  noduri de același tip, peste care s-a definit o relație de precedență având proprietățile (1) și (2) de la [8.1.1.a] precum și următoarele două proprietăți [8.1.1.b]:
  -

- (3) Oricare ar fi  $b, c \in A$ , astfel încât  $b \prec c$  și  $c \prec b$ , dacă  $b \prec d$  și  $c \prec e$  atunci  $d \neq e$ .
    - Cu alte cuvinte, două elemente oarecare între care **nu** există relația de precedență **nu** pot avea același succesor.
- [ 8 . 1 . 1 . b ]
- (4) Dacă **A nu** este vidă ( $n > 0$ ) atunci există un element numit **rădăcină**, care precede toate celelalte elemente.

- Pentru structura arbore se poate formula și o altă definiție echivalentă cu cea de mai sus.
- Prin **arbore**, se înțelege o mulțime de  $n \geq 0$  noduri de același tip, care dacă **nu** este vidă, atunci are un anumit nod numit **rădăcină**, iar restul nodurilor formează **un număr finit de arbori**, doi căte doi **disjuncți**.
- Se constată că atât această definiție, cât și structura pe care o definește, sunt **recursive**, lucru deosebit de important deoarece permite prelucrarea simplă a unei astfel de structuri cu ajutorul unor **algoritmi recursivi**.
- În continuare se vor defini câteva **noțiuni** referitoare la arbori.
- Prin **subarborii** unui arbore, se înțeleg arborii în care se descompune acesta prin îndepărțarea rădăcinii.
  - Spre exemplu arborele din figura 8.1.1.c, după îndepărțarea rădăcinii  $a$ , se descompune în trei subarbori având rădăcinile respectiv  $b, c$  și  $d$ .
- Oricare nod al unui arbore este rădăcina unui **arbore parțial**.
  - Spre exemplu în aceeași figură,  $f$  este rădăcina arborelui parțial format din nodurile  $f, k, l, m$  și  $p$ .
- Un arbore parțial **nu** este întotdeauna **subarbore** pentru arborele complet, în schimb orice **subarbore** este în același timp și **arbore parțial**.
- Într-o structură de arbore, succesorul unui nod se mai numește și **"fiul"** sau **"urmașul"** său.
- Dacă un nod are unul sau mai mulți fii, atunci el se numește **"tatăl"** sau **"părintele"** acestora.
- Dacă un nod are mai mulți fii, aceștia se numesc **"frați"** între ei.
  - Spre exemplu în fig. 8.1.1.c nodul  $b$  este tatăl lui  $e, f$  și  $g$  care sunt frați între ei și sunt în același timp fiiii lui  $b$ .
- Se observă că într-o structură arbore, **arborele parțial** determinat de orice nod diferit de rădăcină, este **subarbore** pentru **arborele parțial** determinat de tatăl său.

- Astfel  $f$  este tatăl lui  $m$ , iar arborele parțial determinat de  $m$  este subarbore pentru arborele parțial determinat de  $f$ .
- Într-o structură arbore se definesc **niveluri** în felul următor: rădăcina formează nivelul 1, fiile ei formează nivelul 2 și în general fiile tuturor nodurilor nivelului  $n$  formează nivelul  $n+1$  (fig.8.1.1.c).
- Nivelul maxim al nodurilor unui arbore se numește **înălțimea** arborelui.
- Numărul fiilor unui nod definește **gradul nodului** respectiv.
- Un nod de grad zero se numește **terminal (frunză)**, iar un nod de grad diferit de zero, nod **intern**.
- **Gradul** maxim al nodurilor unui arbore se numește **gradul arborelui**.
  - Arboarele din figura 8.1.1.c are înălțimea 5, nodul  $d$  este de grad 2, nodul  $h$  este terminal,  $f$  este un nod intern iar gradul arborelui este 3.
- Dacă  $n_1, n_2, \dots, n_k$  este o **secvență** de noduri aparținând unui arbore, astfel încât  $n_i$  este părintele lui  $n_{i+1}$  pentru  $1 \leq i < k$ , atunci această secvență se numește "**drum**" sau "**cale**" de la nodul  $n_1$  la nodul  $n_k$ .
- **Lungimea** unui **drum** este un întreg având valoarea egală cu numărul de ramuri (săgeți) care trebuie traversate pentru a ajunge de la rădăcină la nodul respectiv.
- Rădăcina are lungimea drumului egală cu 1, fiile ei au lungimea drumului egală cu 2 și în general un nod situat pe nivelul  $i$  are lungimea drumului  $i$ .
  - Spre exemplu, în figura 8.1.1.c, lungimea drumului la nodul  $c$  este 2 iar la nodul  $p$  este 5.
- Dacă există un drum de la nodul  $a$  la nodul  $b$ , atunci nodul  $a$  se numește **strămoș** sau **ancestor** al lui  $b$ , iar nodul  $b$  **descendent** sau **urmaș** al lui  $a$ .
  - Spre exemplu în aceeași figură, strămoșii lui  $f$  sunt  $f, b$  și  $a$  iar descendenții săi  $f, k, l, m$  și  $p$ .
- Conform celor deja precizate **tatăl** unui nod este **strămoșul său direct (predecesor)** iar **fiul** unui nod este **descendentul său direct (succesor)**.
- Un strămos respectiv un descendental unui nod, altul decât nodul însuși, se numește **strămoș propriu** respectiv **descendent propriu**.
- Într-un arbore, **rădăcina** este **singurul nod** fără **nici** un strămoș propriu.
- Un nod care **nu** are descendenți proprii se numește **nod terminal (frunză)**.
- **Înălțimea** unui nod într-un arbore este **lungimea celui mai lung drum** de la **nodul respectiv** la un **nod terminal**.

- Pornind de la această definiție, **înălțimea** unui arbore se poate defini și ca fiind egală cu **înălțimea nodului rădăcină**.
- **Adâncimea** unui nod este egală cu lungimea **drumului unic** de la rădăcină la acel nod.
- În practică se mai definește și **lungimea drumului unui arbore** numită și **lungimea drumului intern**, ca fiind egală cu suma lungimilor drumurilor corespunzătoare tuturor nodurilor arborelui.
- Formal, **lungimea medie a drumului intern al unui arbore**, notată cu  $P_I$  se definește cu ajutorul formulei [8.1.1.c].

---


$$P_I = \frac{1}{n} \sum_{i=1}^h n_i * i \quad \text{unde } n = \text{numărul total de noduri}$$

$i = \text{nivelul nodului}$  [8.1.1.c]  
 $n_i = \text{numărul de noduri pe nivelul } i$   
 $h = \text{înălțimea arborelui}$

---

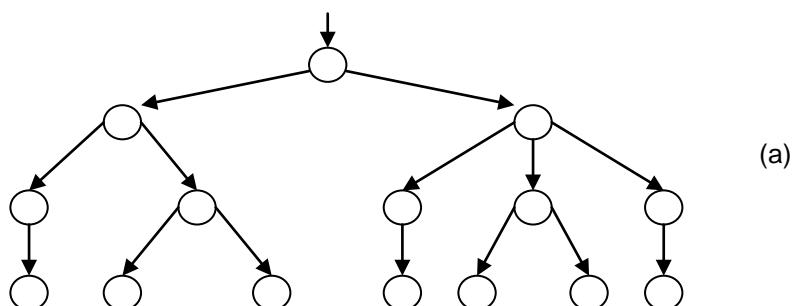
- În scopul definirii **lungimii drumului extern**, ori de câte ori se întâlnește în arborele inițial un **subarbore vid**, acesta se înlocuiește cu un **nod special**.
- În structura care se obține după această activitate, toate nodurile originale vor fi de același **grad** (gradul arborelui).
- Se precizează că **nodurile speciale** care au înlocuit ramurile absente **nu** au descendenți, prin definiție.
  - Spre exemplu, arborele (a) din figura 8.1.1.d, extins cu noduri speciale reprezentate prin patrate, ia forma (b)..
- **Lungimea drumului extern** se definește ca fiind suma lungimilor drumurilor la toate nodurile speciale.
- Dacă numărul nodurilor speciale la nivelul  $i$  este  $m_i$ , atunci **lungimea medie a drumului extern**  $P_E$  este cea definită de formula [8.1.1.d].

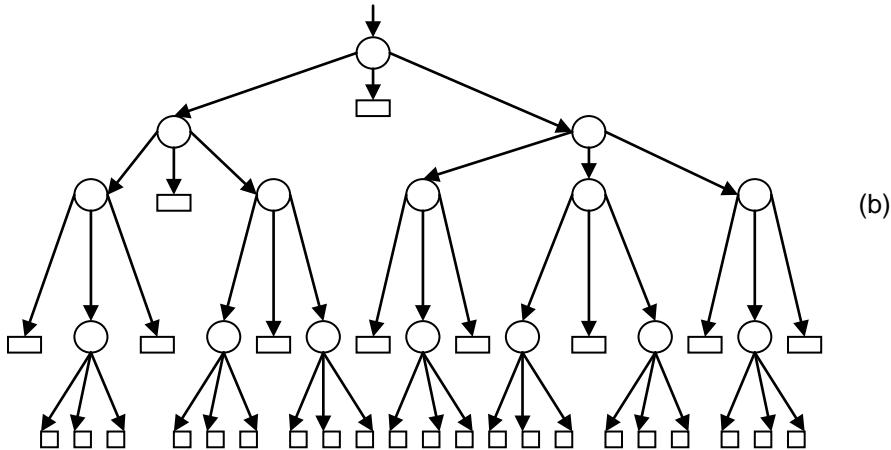
---


$$P_E = \frac{1}{m} \sum_{i=1}^{h+1} m_i * i \quad \text{unde } m = \text{numărul total de noduri speciale}$$

$i = \text{nivelul nodului special}$  [8.1.1.d]  
 $n_i = \text{numărul de noduri speciale pe nivelul } i$   
 $h = \text{înălțimea arborelui}$

---





**Fig. 8.1.1.d.** Arbore ternar extins cu noduri speciale

### 8.1.2. Tipul de date abstract arbore generalizat

- La fel ca și în cazul altor tipuri de structuri de date, este dificil de stabilit un set de operatori care să fie valabil pentru toate aplicațiile posibile ale **structurilor arbore**.
- Cu toate acestea, din mulțimea operatorilor posibili se recomandă pentru **TDA Arbore generalizat** forma prezentată în secvența [8.1.2.a].

#### TDA Arbore generalizat

**Modelul matematic:** arbore definit în sens matematic.

Elementele arborelui aparțin unui aceluiași tip, numit și *tip de bază*.

#### Notății:

*TipNod* - tipul unui nod al arborelui;

*TipArbore* - tipul arbore;

*TipCheie* - tipul cheii unui nod;

*TipNod N*;

*TipArbore A*;

*TipCheie v*;

[8.1.2.a]

#### Operatori:

1. *TipNod Tata(TipNod N, TipArbore A)*  
- operator care returnează tatăl (părintele) nodului *N* din arborele *A*. Dacă *N* este chiar rădăcina lui *A* se returnează "nodul vid"
2. *TipNod PrimulFiu(TipNod N, TipArbore A)*  
- operator care returnează cel mai din stânga fiu al nodului *N* din arborele *A*. Dacă *N* este un nod terminal, operatorul returnează "nodul vid".
3. *TipNod FrateDreapta(TipNod N, TipArbore A)*

- operator care returnează nodul care este fratele dreapta "imediat" al nodului  $N$  din arborele  $A$ . Acest nod se definește ca fiind nodul  $M$  care are același părinte  $T$  ca și  $N$  și care în reprezentarea arborelui apare imediat în dreapta lui  $N$  în ordonarea de la stânga la dreapta a fiilor lui  $T$ .
4. ***TipCheie Cheie***(*TipNod N, TipArbore A*)
    - operator care returnează cheia nodului  $N$  din arborele  $A$ .
  5. ***TipArbore Creaza<sub>i</sub>***(*TipCheie v, TipArbore A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>i</sub>*)
    - este o familie de operatori care sunt reprezentanți pentru fiecare din valorile lui  $i=0,1,2,..$ . Funcția ***Creaza<sub>i</sub>*** generează un nod nou  $R$ , care are cheia  $v$  și căruia îi asociază  $i$  fii. Aceștia sunt respectiv subarborei  $A_1, A_2, ..., A_i$ . În final se generează de fapt un arbore nou având rădăcina  $R$ . Dacă  $i=0$ , atunci  $R$  este în același timp rădăcina și nod terminal.
  6. ***TipNod Radacina***(*TipArbore A*)
    - operator care returnează nodul care este rădăcina arborelui  $A$  sau "nodul vid" dacă  $A$  este un arbore vid.
  7. ***TipArbore Initializeaza***(*TipArbore A*)
    - crează arborele  $A$  vid.
  8. ***Insereaza***(*TipNod N, TipArbore A, TipNod T*)
    - operator care realizează inserția nodului  $N$  ca fiu al nodului  $T$  în arborele  $A$ . În particular se poate preciza și poziția nodului în lista de fii ai tatălui  $T$  (prin convenție se acceptă sintagma "cel mai din dreapta fiu").
  9. ***Suprima***(*TipNod N, TipArbore A*)
    - operator care realizează suprimarea nodului  $N$  și a descendenților săi din arborele  $A$ . Suprimarea se poate defini diferențiat funcție de aplicația în care este utilizată structura de date în cauză.
  10. ***Inordine***(*TipArbore A, TipSubprogram ProcesareNod(...)*)
    - operator care parcurge nodurile arborelui  $A$  în "inordine" și aplică fiecărui nod subprogramul de prelucrare *ProcesareNod*.
  11. ***Preordine***(*TipArbore A, TipSubprogram ProcesareNod(...)*)
    - operator care parcurge nodurile arborelui  $A$  în "preordine" și aplică fiecărui nod subprogramul de prelucrare *ProcesareNod*.
  12. ***Postordine***(*TipArbore A, TipSubprogram ProcesareNod(...)*)
    - operator care parcurge nodurile arborelui  $A$  în "postordine" și aplică fiecărui nod subprogramul de prelucrare *ProcesareNod*.

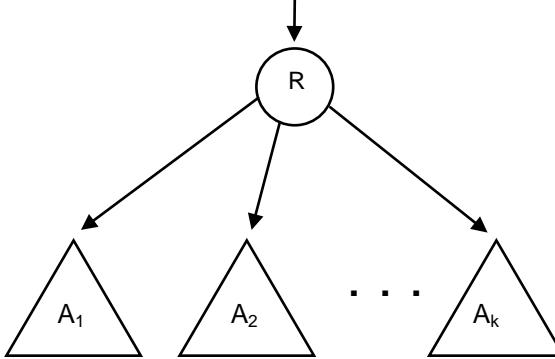
- Structura **arbore generalizat** este **importantă** deoarece apare frecvent în **practică**, spre exemplu arborii familiali, sau structura unei cărți defalcată pe capitole, secțiuni, paragrafe și subparagrafe.
- Din punctul de vedere al reprezentării lor în memorie, **arborii generalizați** au marea dezavantaj că **au noduri de grade diferite**, ceea ce conduce la **structuri neuniforme** de date sau la **structuri uniforme parțial utilizate**.

### 8.1.3. Traversarea arborilor generalizați

- Una din activitățile fundamentale care se execută asupra unei structuri arbore este **traversarea** acesteia.
- Ca și în cazul listelor liniare, prin **traversarea** unui arbore se înțelege execuția unei anumite operații asupra tuturor nodurilor arborelui.
- În timpul traversării, nodurile sunt **vizitate** într-o anumită **ordine**, astfel încât ele pot fi considerate ca și cum ar fi integrate într-o listă liniară.
- Descrierea și înțelegerea celor mai mulți algoritmi este mult ușurată dacă în cursul prelucrării se poate preciza **elementul următor** al structurii arbore, respectiv se poate **liniariza** structura arbore.
- În principiu tehniciile de traversare a arborilor generalizați se încadrează în **două** mari categorii:
  - (1) Tehnici bazate pe **căutarea în adâncime** (“**depth-first search**”)
  - (2) Tehnici bazate pe **căutarea prin cuprindere** (“**breadth-first search**”).
- Aceste tehnici își au sorgintea în traversarea structurilor de date **graf**, dar prin particularizare sunt aplicabile și altor categorii de structuri de date, respectiv structurii de date **arbore generalizat**.

#### 8.1.3.1. Traversarea arborilor generalizați prin tehnici bazate pe căutarea în adâncime: preordine, inordine și postordine

- **Principiul căutării în adâncime (depth-first search)** presupune:
  - (1) Parcurgerea “**în adâncime**” a structurii, prin îndepărțare de punctul de pornire, până când acest lucru **nu** mai este posibil.
  - (2) În acest moment se **revine** pe drumul parcurs până la proximul punct care permite o nouă posibilitate de înaintare în adâncime.
  - (3) Procesul **continuă** în aceeași manieră până când structura de date este parcursă în întregime.

- Există trei moduri de **traversare (liniarizare)** care se referă la o structură de date arbore generalizat, bazate pe căutarea în adâncime și anume:
  - **(1) Traversarea în preordine**
  - **(2) Traversarea în inordine**
  - **(3) Traversarea în postordine**
- Cele trei modalități de traversare, se **definesc** de regulă în manieră **recursivă**, asemeni structurii arbore și anume, **ordonarea** unui arbore se definește presupunând că **subarborii** săi s-au ordonat **deja**.
  - Cum subarborii au noduri strict mai puține decât arborele complet, rezultă că, aplicând recursiv de un număr suficient de ori definiția, se ajunge la ordonarea unui arbore vid, care este evidentă.
- Cele trei moduri de traversare ale unui arbore generalizat **A** se definesc recursiv după cum urmează:
  - Dacă arborele **A** este **nul**, atunci ordonarea lui **A** în preordine, inordine și postordine se reduce la **lista vidă**.
  - Dacă **A** se reduce la **un singur nod**, atunci nodul însuși, reprezintă traversarea lui **A**, în oricare din cele trei moduri.
  - Pentru restul cazurilor, fie arborele **A** cu rădăcina **R** și cu subarborii **A<sub>1</sub>, A<sub>2</sub>, …, A<sub>k</sub>**. (fig. 8.1.3.1.a):
 

**Fig.8.1.3.1.a.** Structură de arbore generalizat

1<sup>o</sup>. Traversarea în **preordine** a arborelui **A** presupune:

- Traversarea rădăcinii **R**
- Traversarea în **preordine** a lui **A<sub>1</sub>**
- Traversarea în **preordine** a lui **A<sub>2</sub>, A<sub>3</sub>** și aşa mai departe până la **A<sub>k</sub>** inclusiv.

2<sup>o</sup>. Traversarea în **inordine** a arborelui **A** presupune:

- Traversarea în **inordine** a subarborelui **A<sub>1</sub>**
- Traversarea nodului **R**

- Traversările în **inordine** ale subarborilor  $\mathbf{A}_2, \mathbf{A}_3, \dots, \mathbf{A}_k$ .

3º. Traversarea în **postordine** a arborelui  $\mathbf{A}$  constă în:

- Traversarea în **postordine** a subarborilor  $\mathbf{A}_1, \mathbf{A}_2, \dots, \mathbf{A}_k$
- Traversarea nodului  $\mathbf{R}$ .

- **Structurile de principiu** ale procedurilor care realizează aceste traversări apar în secvența [8.1.3.1.a] iar un exemplu de **implementare generică** în secvența [8.1.3.1.b].

---

```
/*Traversarea arborelui generalizat - modalități principiale*/

Preordine(A) : R, Preordine(A1), Preordine(A2), ..., Preordine(Ak).
Inordine(A) : Inordine(A1), R, Inordine(A2), ..., Inordine(Ak). [8.1.3.1.a]
Postordine(A) : Postordine(A1), Postordine(A2), ..., Postordine(Ak), R.
```

---

```
/*Traversarea în Preordine a arborelui generalizat - varianta pseudocod*/
```

```
subprogram Preordine(TipNod r)
    *listea(r);
    pentru (fiecare fiu f al lui r, (dacă există vreunul),
        în ordine de la stânga spre dreapta) execută
        Preordine(f);
    □
```

---

```
/*Traversarea în Inordine a arborelui generalizat - varianta pseudocod*/
```

```
subprogram Inordine(TipNod r)
    dacă (r este nod terminal) atunci
        *listea(r);
    altfel [8.1.3.1.b]
        Inordine(cel mai din stânga fiu al lui r);
        *listea(r);
        pentru (fiecare fiu f al lui r, cu excepția celui
            mai din stânga, în ordine de la stânga spre
            dreapta) execută
            Inordine(f);
        □
    □
```

---

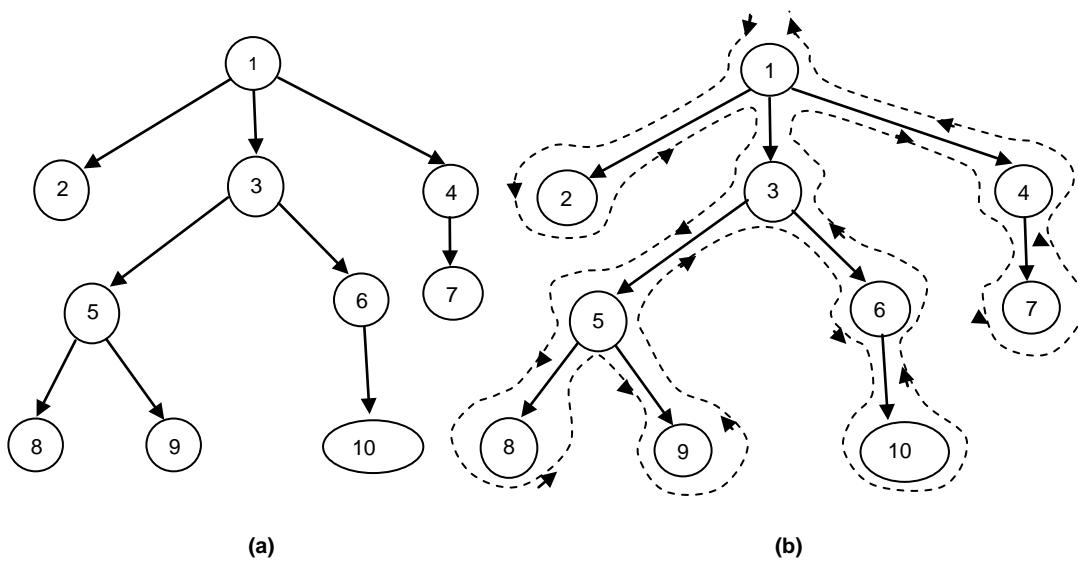
```
/*Traversarea în Postordine a arborelui generalizat - varianta pseudocod*/
```

```
subprogram Postordine(TipNod r)
    pentru fiecare fiu f al lui r, (dacă există vreunul),
        în ordine de la stânga spre dreapta execută
        Postordine(f);
    □
    *listea(r);
```

---

- Spre exemplu pentru arborele din figura 8.1.3.1.b (a), traversările anterior definite, conduc la următoarele secvențe de noduri:

- preordine: 1,2,3,5,8,9,6,10,4,7.
- postordine: 2,8,9,5,10,6,3,7,4,1.
- inordine: 2,1,8,5,9,3,10,6,7,4.



**Fig.8.1.3.1.b.** Traversarea unui arbore generalizat

- O metodă practică simplă de parcurgere a unui **arbore** este următoarea:
  - Dându-se o **structură arbore generalizat**, se imaginează parcurgerea acesteia în sens trigonometric pozitiv, rămânând cât mai aproape posibil de arbore (fig.8.1.3.1.b (b)).
  - Pentru **preordine**, nodurile se listează de **prima** dată când sunt întâlnite: 1, 2, 3, 5, 8, 9, 6, 10, 4, 7;
  - Pentru **postordine** nodurile se listează **ultima** dată când sunt întâlnite, respectiv când sensul de parcurgere este spre părinții lor: 2, 8, 9, 5, 10, 6, 3, 7, 4, 1;
  - Pentru **inordine**, un **nod terminal** se listează când este întâlnit **prima** oară, iar un **nod interior** când este întâlnit **a doua oară**: 2, 1, 8, 5, 9, 3, 10, 6, 7, 4.

□-----  
**Exemplul 8.1.3.1.a.** Implementarea traversării în **preordine** a unui **arbore** utilizând operatorii definiți în cadrul **TDA Arbore generalizat**, varianta recursivă.

- Procedura din secvența următoare realizează tipărirea în **preordine** a cheilor nodurilor arborelui generalizat **A**
- Procedura este apelată prin următorul apel:  
`TraversarePreordine(Radacina(A))`.

- Se presupune că nodurile arborelui sunt de tip `tip_nod`.

```
-----  
void TraversarePreordine (tip_nod r);  
  
/*Listeaza cheile descendenților lui r în preordine*/  
/*Implementare bazată pe setul de operatori TDA Arbore  
generalizat*/  
  
{ tip_nod f;  
    scrie(Cheie(r,a));  
    f= PrimulFiu(r,a);  
    while(!f==0)  
    {  
        TraversarePreordine (f);  
        f= FrateDreapta(f,a)  
    }  
} /*TraversarePreordine*/  
-----
```

[8.1.3.1.c]

□-----□

**Exemplul 8.1.3.1.b.** Implementarea traversării în preordine a unui **arbore** utilizând operatorii definiți în cadrul **TDA Arbore generalizat**, varianta nerecursivă.

- Se presupune că nodurile arborelui sunt de tip `tip_nod`.
- În cadrul variantei nerecursive se utilizează **stiva** `s`, care conține elemente de `tip_nod`.
- Când se ajunge la prelucrarea nodului `n`, stiva va conține memorat **drumul** de la rădăcină la nodul `n`, cu rădăcina la baza stivei și nodul `n` în vârf.
- Procedura care apare în secvența [8.1.3.1.d], are **două moduri de acțiune**.
  - (1) În **primul mod**, se descinde de-a lungul celui mai stâng drum neexplorat din cadrul arborelui, tipărand (**prelucrând**) nodurile întâlnite pe drum și **introducându-le** în stivă, până se ajunge la un nod terminal.
  - (2) Se trece în cel de-al **doilea mod** de operare care presupune revenirea pe drumul memorat în stivă, **eliminând** pe rând nodurile, până la întâlnirea primului nod care are frate dreapta.
  - (3) Se **revine** în primul mod și reîncepe descenderea cu acest frate încă neexplorat.
  - (4) Se continuă alternarea modurilor de parcurgere până la parcurgerea integrală a arborelui.
- Procedura începe în modul unu și se termină când stiva devine vidă.

```
-----  
void TraversarePreordineNerecursiva(tip_nod a);  
  
/*Implementare nerecursivă bazată pe structura stivă  
Se utilizează operatorii TDA Arbore generalizat, TDA Stivă*/  
  
{ tip_nod m;  
    tip_stiva s;
```

```

boolean gata;

Initializează(s); /*initializare stivă*/
m= Radacina(a); /*stabilire nod de pornire - rădăcina*/
gata= false;
while(!gata)
    if( !m==0 ) THEN /*primul mod de parcursere*/
    {
        *scrive(Cheie(m,a)); /*prelucare nod*/
        Push(m,s);           /*introducere nod în stivă*/
        m= PrimulFiu(m,a);  /*explorează fiul stâng*/
    }
    else
        [8.1.3.1.d]
        if Stivid(s) /*terminare parcursere*/
            gata= true;
        else /*modul al doilea de parcursere*/
        {
            m= FrateDreapta(VarfSt(s),a); /*explorează
                                              frate dreapta*/
            Pop(s); /*extragere nod din stivă*/
        }
    } /*TraversarePreordineNerecursiva*/

```

---

- Traversările în preordine și postordine ale unui arbore sunt utile în obținerea de **informații anterioare (ancestrale)** referitoare la noduri.
  - Astfel, se consideră ordinea liniară a nodurilor unui arbore generalizat parcurs în postordine.
  - Se definește **POSTORDINE(n)** ca fiind poziția exprimată ca număr întreg, a nodului n în această ordonare.
  - **POSTORDINE(n)** se numește “**număr postordine al lui n**”.
  - Se definește de asemenea **DESCENDENȚI(n)** ca fiind egal cu numărul descendenților proprii ai nodului n.
    - Spre exemplu în fig.8.1.3.1.b, numerele postordine ale nodurilor "9", "6" și "1" sunt respectiv 3,6 și 10.
- Numerele **postordine** asignate nodurilor se bucură de următoarea **proprietate**:
  - Într-un arbore generalizat, nodurile oricărui subarbore având rădăcina n sunt numerotate **consecutiv** de la (**POSTORDINE(n)-DESCENDENȚI(n)**) la **POSTORDINE(n)**.
  - Astfel, pentru a verifica dacă un nod x este **descendentul** unui nod y, este suficient să se verifice condiția [8.1.3.1.e]:
 

---

 POSTORDINE(y)- DESCENDENȚI(y)<POSTORDINE(x)<POSTORDINE(y)
 [8.1.3.1.e]
 

---

- O proprietate similară se poate defini și pentru ordonarea în **preordine**.

### 8.1.3.2. Traversarea arborilor generalizați prin tehnica căutării prin cuprindere

- Tehnica căutării prin **cuprindere** derivă tot din traversarea **grafurilor**, dar ea este utilizată, e adevarat mai rar, și la traversarea arborilor [Ko86].
- Se mai numește și traversare pe niveluri (“**level traversal**”) [We94] și este utilizată cu precădere în reprezentarea grafică a arborilor.
- **Principiul** acestei tehnici este simplu: nodurile nivelului  $i+1$  al structurii arbore sunt vizitate **doar** după ce au fost vizitate toate nodurile nivelului  $i$  ( $0 < i < h$ , unde  $h$  este înălțimea arborelui).
- Pentru implementarea acestei tehnici de parcurgere a arborilor generalizați, se utilizează drept structură de date suport, **structura coadă**.

```
-----  
subprogram TraversarePrinCuprindere1(tip_nod r)  
  
/*Varianta pseudocod bazată pe TDA Coadă*/  
  
tip_coadă coada;  
tip_nod y;  
Initializeaza(coada);  
daca (r nu este nodul vid) atunci  
    Adauga(r,coada); /*procesul de amorsare*/  
    cât timp NOT vid(coada)execută [8.1.3.2.a]  
        r=Cap(coada); Scoate(coada); /*scoate nodul din coadă*/  
        *listea(r); /*prelucrare nod*/  
        pentru fiecare fiu y al lui r, (dacă există vreunul),  
            în ordine de la stânga la dreapta execută  
            Adauga(y,coada); /*adaugă nodul în coadă*/  
    □  
□  
-----
```

**Exemplul 8.1.3.2.** Implementarea traversării primării prin cuprindere a unui arbore utilizând operatorii definiți în cadrul **TDA Arbore generalizat** și **TDA Coadă** este ilustrată în secvența [8.1.3.2.b]. Nodurile vizitate sunt afișate.

```
-----  
void TraversarePrinCuprindere2(tip_nod r);  
  
/*Implementare bazată pe TDA Arbore generalizat, TDA Coada*/  
  
{    tip_coadă coada;  
    tip_nod f;  
  
    Initializeaza(coada);  
    Adauga(r,coada);  
    while (!vid(coada))  
    {  
        r=Cap(coada); Scoate(coada);  
        *scrive(Cheie(r));  
        f=PrimulFiu(r);  
    }  
-----
```

```

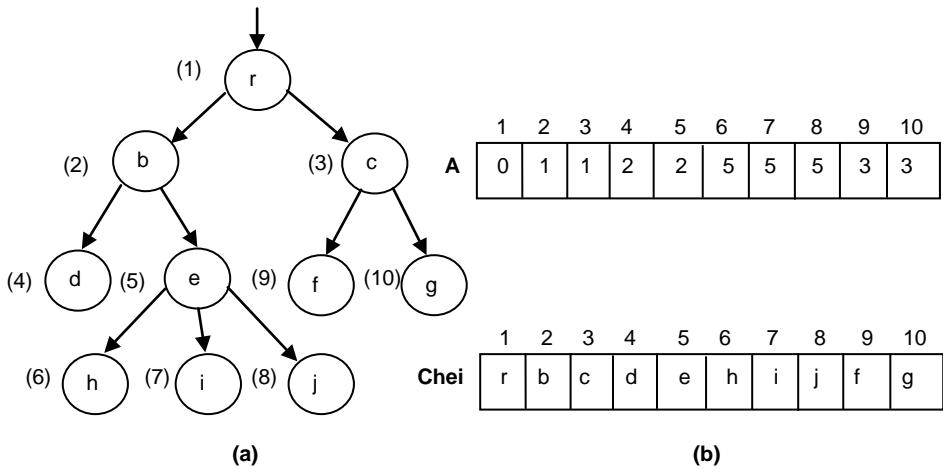
if ( !f==NIL )
{
    Adauga( f , coada );
    f=FrateDreapta( f )
    while ( !f==NIL )
    {
        Adauga( f , coada );
        f=FrateDreapta( f )
    }
}
} /*TraversarePrinCuprindere*/
-----
```

## 8.1.4. Tehnici de implementare a TDA arbore generalizat

- În cadrul acestui subcapitol se vor prezenta câteva din **implementările** posibile ale **structurii arbore generalizat**, corelate cu aprecieri referitoare la capacitatea acestora de a suporta operatorii definiți în cadrul **TDA Arbore generalizat**.

### 8.1.4.1. Implementarea arborilor generalizați cu ajutorul tablourilor

- Se bazează pe următoarea **tehnologie**:
  - Fie A un arbore generalizat cu n noduri.
  - Se numerotează nodurile arborelui A de la 1 la n.
  - Se asociază nodurilor arborelui elementele unui tablou A, astfel încât nodului i al arborelui îi corespunde locația A[ i ] din tablou.
  - Cea mai simplă manieră de reprezentare a unui arbore, care permite implementarea operației **Tata**, este cea conform căreia fiecare intrare A[ i ] din tablou memorează un indicator (cursor) la **părintele** nodului i.
  - Rădăcina se poate distinge prin valoarea zero a cursorului.
  - Cu alte cuvinte, dacă A[ i ]=j atunci nodul j este **părintele** nodului i iar dacă A[ i ]=0, atunci nodul i este **rădăcina** arborelui.
- Acest mod de reprezentare, face uz de proprietatea arborilor care stipulează că orice nod are **un singur părinte**, motiv pentru care se numește și "indicator spre părinte".
- Găsirea părintelui unui nod se face într-un interval **constant** de timp **O(1)** iar parcurgerea arborelui în direcția nod – rădăcină, se realizează într-un interval de timp proporțional cu adâncimea nodului.



**Fig. 8.1.4.1.a.** Reprezentarea arborilor generalizați cu ajutorul tablourilor (varianta “indicator spre părinte”)

- Această reprezentare poate implementa simplu și operatorul **Cheie** dacă se adaugă un alt tablou **Chei**, astfel încât **Chei[i]** este cheia nodului **i**, sau dacă elementele tabloului **A** se definesc de tip articol având câmpurile **cheie** și **cursor**.
  - În figura 8.1.4.1.a apare o astfel de reprezentare (b) a aborelui generalizat (a).
  - Reprezentarea "indicator spre părinte" are însă **dezavantajul** implementării dificile a operatorilor referitor la fii.
    - Astfel, unui nod dat **n**, **i** se determină cu dificultate fiii sau înălțimea.
    - În plus, reprezentarea "indicator spre părinte", **nu** permite specificarea ordinii fiilor unui nod, astfel încât operatorii **PrimulFiu** și **FrateDreapta** **nu** sunt bine precizați.
    - Pentru a da acuratețe reprezentării, se poate impune o **ordine artificială** a nodurilor în tablou, respectând următoarele convenții:
      - (a) - numerotarea fiilor unui nod se face numai după ce nodul a fost numărat;
      - (b) - numerele fiilor cresc de la stânga spre dreapta. Nu este necesar ca fiii să ocupe poziții adiacente în tablou.
  - În accepțiunea respectării acestor convenții, în secvența [8.1.4.1.a] apare redactată funcția **FrateDreapta**.
  - Tipurile de date presupuse sunt cele precizate în antetul procedurii. Se presupune că **nodul vid** este reprezentat prin zero.

```
/*Implementarea Arborilor generalizați cu ajutorul  
tablourilor - varianta "Indicator spre părinte"*/
```

```
typedef int tip_nod;  
typedef tip_nod tip_arbore[MaxNod];
```

```

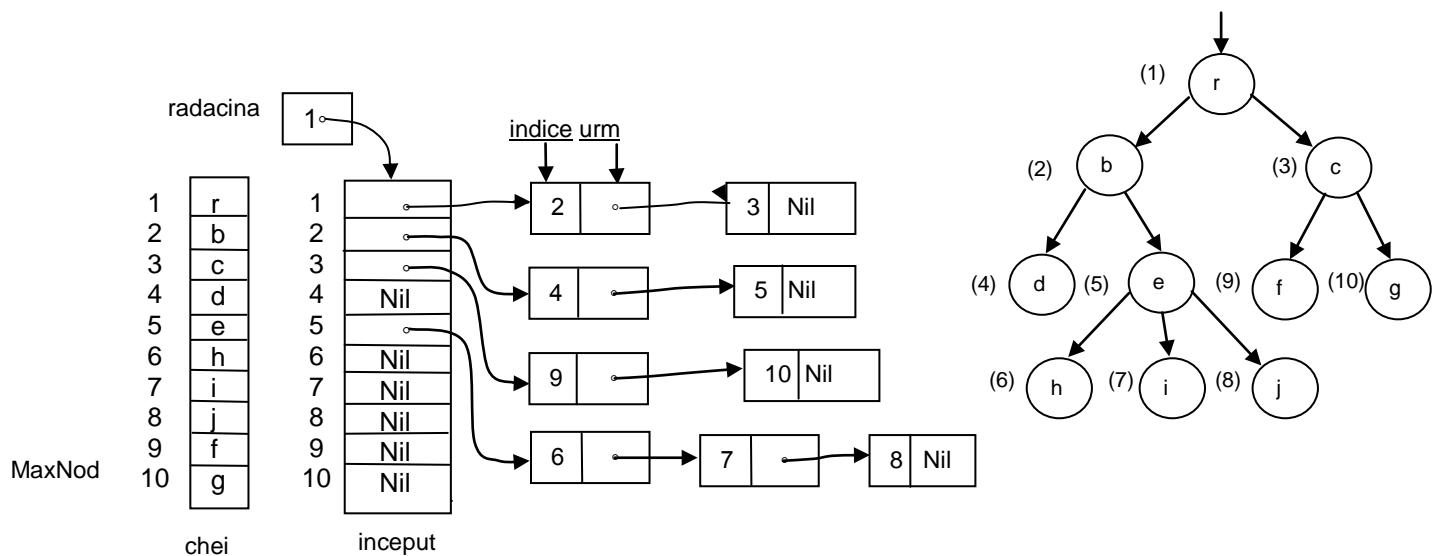
/*Exemplu de implementare a operatorului FrateDreapta*/

tip_nod FrateDreapta(tip_nod n, tip_arbore a)
{
    tip_nod i,rezultat,parinte;

    parinte=a[n];                                /*[8.1.4.1.a]*/
    rezultat=0;
    i=n;
    do
    {
        i++;
        if (a[i]==parinte) rezultat=i;
    }
    while(!rezultat==0) || (i=MaxNod))
    return rezultat;
}  /*FrateDreapta*/
-----
```

### 8.1.4.2. Implementarea arborilor generalizați cu ajutorul listelor

- O manieră importantă și utilă de implementare a arborilor generalizați este aceea de a crea pentru **fiecare nod** al arborelui **o listă** a fiilor săi.
- Datorită faptului că numărul fiilor poate fi **variabil**, o variantă potrivită de implementare o reprezintă utilizarea **listelor înlántuite**.
- În fig.8.1.4.2.a se sugerează maniera în care se poate implementa arborele din figura 8.1.4.1.a.(a).
  - Se utilizează un **tablou** (**inceput**) care conține câte o locație pentru fiecare nod al arborelui.
  - Fiecare element al tabloului **inceput** este o referință la o **listă înlántuită simplă**, ale cărei elemente sunt nodurile fi fi ai nodului corespunzător intrării, adică elementele listei indicate de **inceput** [*i*] sunt fiile nodului *i*.



**Fig.8.1.4.2.a.** Reprezentarea arborilor generalizați cu ajutorul listelor înlántuite

- În continuare se prezintă două posibilități de implementare.
  - Pentru prima se sugerează tipurile de date prevăzute în secvența [8.1.4.2.a]. Este vorba despre o implementare bazată pe liste înlățuite.
    - Se presupune că rădăcina arborelui este memorată în câmpul specific **radacina**.
    - Nodul vid se reprezintă prin valoarea NIL.
    - În aceste condiții în secvența [8.1.4.2.b] apare implementarea operatorului **PrimulFiu**.

---

/\*Reprezentarea arborilor generalizați utilizând liste înlățuite simple implementate cu ajutorul pointerilor\*/

```

int MaxNod=...; /*număr maxim noduri*/

typedef struct nod_lista { /*structura nod listă*/
    int indice;
    struct nod_lista * urm;
} tip_nod;
typedef struct tip_arbore { /*structura arbore*/
    tip_nod * inceput[MaxNod];
    tip_cheie chei[MaxNod];
    int radacina;
} tip_arbore

```

[ 8.1.4.2.a ]

---

/\*Exemplu de implementare al operatorului **PrimulFiu**  
{se utilizează TDA Listă, varianta restrânsă\*/

```

tip_nod PrimulFiu(tip_nod n, tip_arbore a)

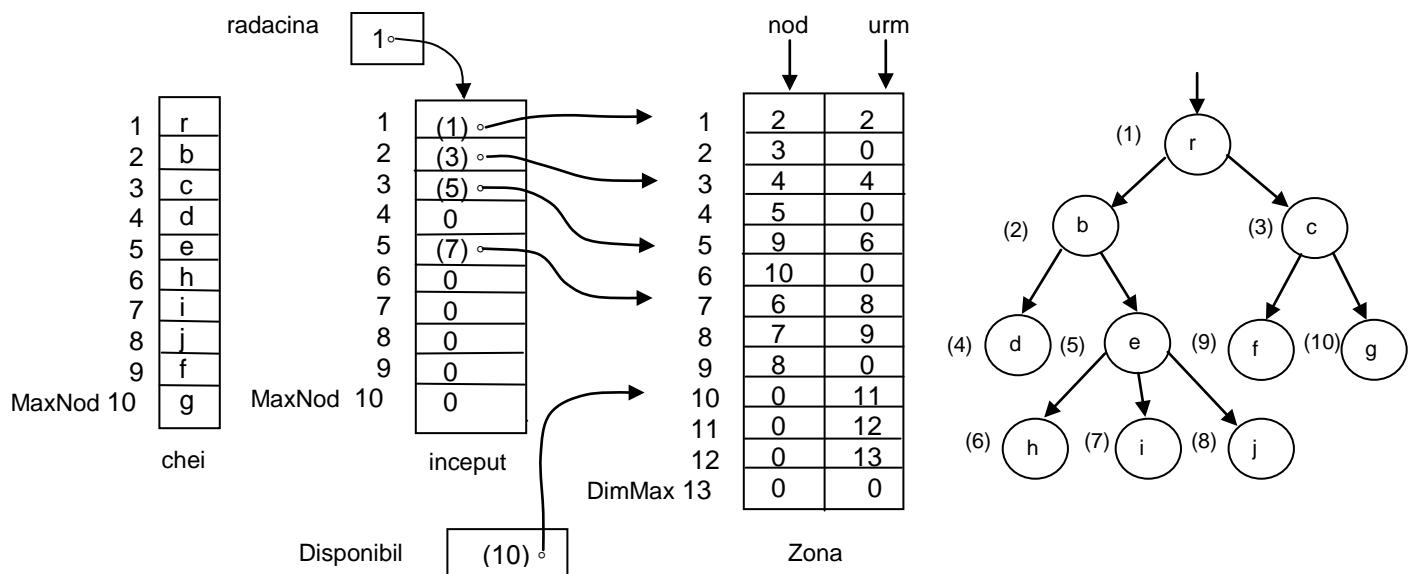
{ tip_nod * lista, rezultat;
  lista=a.inceput[n];
  if Fin(lista) /*n este un nod terminal*/
    rezultat=0;
  else
    rezultat=Furnizeaza(Primul(lista),lista);
} /*PrimulFiu*/

```

[ 8.1.4.2.b ]

- 
- Cea de-a două modalitate de implementare se bazează pe **implementarea listelor** cu ajutorul **cursorilor** (Vol.1 &6.3.3).
    - În această reprezentare atât listele ca atare cât și înlățuirile din cadrul lor sunt întregi utilizați ca și cursori într-o tabelă de elemente (Zona).
    - În fig.8.1.4.2.b se prezintă maniera în care se poate implementa arborele generalizat din figura 8.1.4.1.a.(a).

- Se utilizează și în acest caz tabloul `inceput` care conține referințe la listele înălțuite memorate în tabloul `Zona`, liste ale căror elemente sunt nodurile fiilor ai nodului corespunzător intrării.



**Fig.8.1.4.2.b.** Reprezentarea arborilor generalizați cu ajutorul listelor înălțuite implementate cu ajutorul cursorilor

- În secvența [8.1.4.2.c] apar structurile de date aferente acestei implementări, iar în secvențele [8.1.4.2.d,e] implementările operatorilor **PrimulFiu** respectiv **Tata**.
- După cum se remarcă, implementarea operatorului **Tata** se realizează mai dificil, încrucișat trebuie realizată o baleere a tuturor listelor de fiți în vederea stabilirii cărei liste îi aparține nodul furnizat ca parametru.

---

/\*Reprezentarea arborilor generalizați utilizând liste  
înlățuite implementate cu ajutorul cursorilor\*/

```

int MaxNod=...; /*număr maxim noduri*/
int DimMax=...; /*dimensiune tablou Zona*/

typedef int tip_cursor;
typedef int tip_nod;

typedef struct tip_arbore {           /*structura arbore*/
    tip_cursor inceput[MaxNod];
    tip_cheie chei[MaxNod];           /**[8.1.4.2.c]*/
    tip_nod radacina;
} tip_arbore;

typedef struct element_zona {        /*structura element Zona*/
    tip_nod nod;
    tip_cursor urm;
} element_zona

element_zona Zona[DimMax];           /*definire tablou Zona*/

```

```

tip_cursorDisponibil; /*cursor inceput lista disponibil*/
tip_arbore R;
-----
/*Exemplu de implementare al operatorului PrimulFiu*/

tip_nod PrimulFiu(tip_nod n, tip_arbore a)
{
    tip_cursor c; {cursor la inceputul listei fiilor lui n}
    tip_nod rezultat;

    c=a.inceput[n];
    if (c=0) {n este nod terminal} /*[8.1.4.2.d]*/
        rezultat=0;
    else
        rezultat=Zona[c].nod;
    return rezultat;
} /*PrimulFiu*/
-----
/*Exemplu de implementare al operatorului Tata*/

tip_nod Tata(tip_nod n, tip_arbore a)
{
    tip_nod p; /*parcurge tății posibili ai lui n*/
    tip_cursor c; /*parcurge fiile lui p*/
    tip_nod rezultat;

    rezultat=0; p= -1; /*[8.1.4.2.e]*/
    do {
        p=p+1; c=a.inceput[p]; /*lista fiilor lui p*/
        while ((c!=0) && (rezultat=0)) /*verifică dacă n este
            printre fiile lui p*/
        {
            if (Zona[c].nod=n) rezultat=p;
            else c=Zona[c].urm;
        }
        while((rezultat!=0) || (p=MaxNod-1));
    return rezultat;
} /*Tata*/

```

### 8.1.4.3. Implementarea structurii arbore generalizat pe baza relațiilor "primul-fiu" și "frate-dreapta"

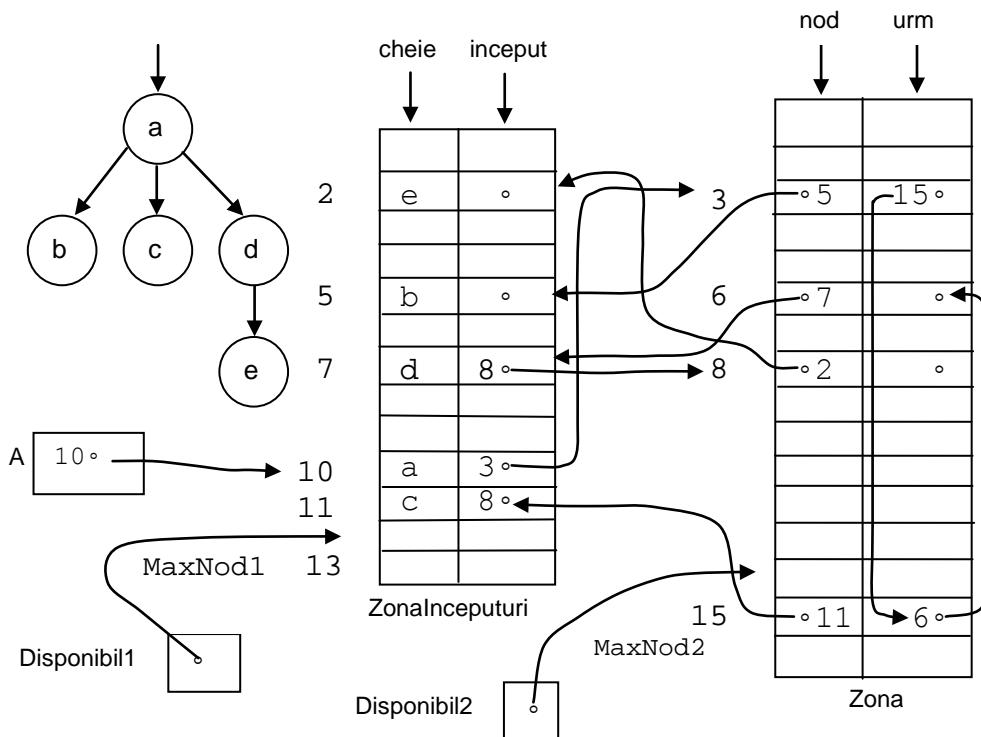
- Implementările structurilor arbori generalizați, descrise până în prezent, printre alte **dezavantaje**, îl au și pe acela de a **nu** permite implementarea simplă a operatorului **Creaza** și deci de a **nu** permite dezvoltarea facilă a unor **structuri complexe** pornind de la **structuri simple**.
- Această dificultate provine din faptul că, în timp ce, **spre exemplu** în reprezentarea bazată pe **cursori, liste fiile** partajează o singură zonă comună (Zona), fiecare arbore are propriul său tablou de inceputuri (inceput).
- Astfel, pentru a implementa spre **exemplu** operatorul **TipArbore Creaza<sub>2</sub>**(**TipCheie v, TipArbore A<sub>1</sub>,A<sub>2</sub>**) este necesar ca arborii A<sub>1</sub> și A<sub>2</sub> să fie recopiate într-un al treilea arbore, al cărui tablou de inceputuri conține tablourile celor

doi arbori, completat cu locația corespunzătoare lui v indicând o listă de fii care conține rădăcinile lui  $A_1$  și  $A_2$ .

- Pentru a depăși această dificultate, pornind de la structura de date implementată în paragraful anterior cu ajutorul cursorilor, se propun două variante de rezolvare.

- **Varianta 1**

- (1) Tabloul inceput se înlocuiește cu un tablou care cuprinde începuturile listelor de fii pentru **toate nodurile** corespunzătoare tuturor arborilor luați în considerare (ZonaIncepuri din fig.8.1.4.3.a).
  - Elementele acestui tablou sunt articole cu două câmpuri: cheie și inceput.
  - Câmpul cheie identifică un nod al arborelui iar inceput este un cursor care indică lista filor săi memorată în tabloul Zona.
  - Acest cursor materializează de fapt relația “**primul-fiu**”.
- (2) Nodurile **nu** mai sunt numerotate în ordinea  $1, 2, \dots, n$  (ca în tabela inceput) ci ele sunt reprezentate printr-un indice arbitrar în ZonaIncepuri.
  - Din acest motiv, este normal ca în cadrul tabloului Zona, câmpul nod să **nu** mai indice direct "numărul" unui nod, ci el să aibă valoarea unui cursor în ZonaIncepuri, valoare care indică **poziția** acelui nod.
  - Câmpul urm din cadrul același tablou realizează înlănțuirea nodurilor fii și materializează relația “**frate-dreapta**”.
- În această reprezentare TipAbore este de fapt un cursor în ZonaIncepuri.
- În figura 8.1.4.3.a este prezentată structura de date aferentă arborelui generalizat reprezentat în colțul stânga sus al figurii.



**Fig.8.1.4.3.a.** Reprezentarea unui arbore generalizat cu ajutorul relațiilor “primul-fiu” și “frate-dreapta” (Varianta 1)

- Cheile nodurilor sunt a,b,c,d și e cărora li s-au atribuit prin mecanismul de alocare pozițiile 10,5,11,7 respectiv 2 în tabloul ZonaIncepaturi.
- Prinț-un mecanism similar au fost atribuite și locațiile tabloului Zona.
- În acest scop, cele două tablouri dispun de **liste de disponibili specifice** (Disponibil1 respectiv Disponibil2) care înlățuite locațiile libere și din care se face alocarea spațiului necesar nodurilor.
- Evident, în aceleasi liste sunt înlanțuite nodurile disponibilizate.
- Din punct de vedere formal, structurile de date aferente acestei reprezentări apar în secvența [8.1.4.3.a].

---

```
/*Reprezentarea arborilor generalizați bazată pe relațiile
primul-fiu și frate-dreapta (Varianta 1) C*/
```

```
int MaxNod1=...; /*dimensiune tablou ZonaIncepaturi*/
int MaxNod2=...; /*dimensiune tablou Zona*/

typedef int tip_cursor;
typedef int tip_nod;
typedef tip_cursor tip_arbore;

typedef struct linie_tablou_incepaturi {
    tip_cheie cheie;
    tip_cursor inceput;
} linie_tablou_incepaturi; /*[8.1.4.3.a]*/

typedef struct linie_tablou_zona {
    tip_nod nod;
    tip_cursor urm;
} linie_tablou_zona;

linie_tablou_incepaturi ZonaIncepaturi[MaxNod1]; /*definire
                                                    tablou ZonaIncepaturi*/
linie_tablou_zona Zona[MaxNod2]; /*definire tablou Zona*/

tip_cursor Disponibil1; /*cursor inceput lista disponibili a
                           tabloului ZonaIncepaturi*/
tip_cursor Disponibil2; /*cursor inceput lista disponibili a
                           tabloului Zona*/
tip_arbore R;
```

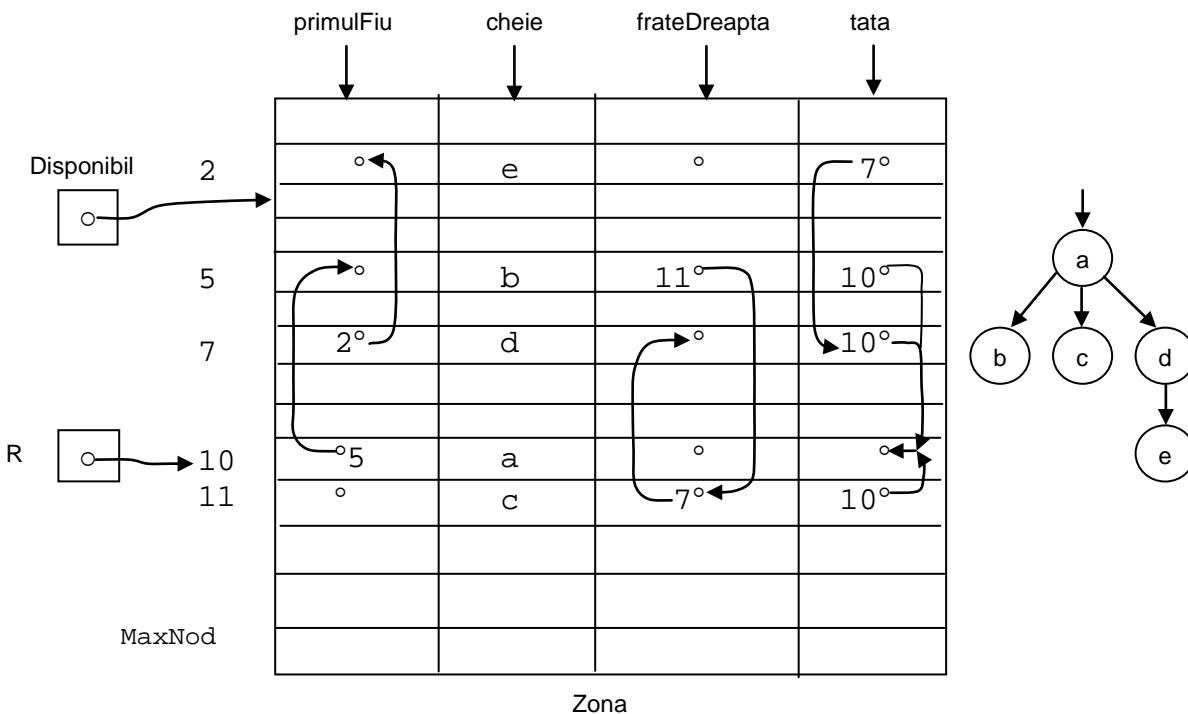
---

- Structura de date poate fi utilizată în implementarea eficientă a operatorului **Creaza** încrât permite **combinarea** simplă a structurilor de arbori.
- În cadrul acestei structuri, câmpul urm al tabloului Zona, precizează chiar fratele dreapta al nodului în cauză.

- De asemenea, fiind dat un nod al cărui cursor în ZonaIncepaturi este  $i$ , câmpul ZonaIncepaturi[i].inceput indică cursorul în spațiul Zona al primului fiu al lui  $i$ . Fie  $j$  valoarea acestui cursor. În continuare cheia primului fiu se poate afla simplu știind că Zona[j].nod reprezintă cursorul acestui nod în ZonaIncepaturi.

### • Varianta 2

- Pornind de la reprezentarea bazată pe cursori, pentru a simplifica și mai mult lucrurile se poate renunța la tabloul **inceput**.
- Astfel, în continuare un nod **nu** va mai fi precizat prin cursorul său în tabloul **inceput** ci prin indexul celulei pe care el o ocupă în tabloul **Zona**.
- Pentru identificarea unui nod, în tabloul **Zona** se adaugă câmpul **cheie**.
- În aceste condiții, câmpul **urm** devine **frateDreapta** și va indica direct fratele dreapta al nodului respectiv.
- Pentru a se păstra informația referitoare la primul fiu, **Zona** se prelungeste cu un câmp auxiliar numit **primulFiu**.



**Fig.8.1.4.3.b.** Reprezentarea unui arbore generalizat cu ajutorul relațiilor “primul-fiu” și “frate-dreapta” (Varianta 2)

- O structură de date încadrată în **TipArbore**, este desemnată în aceste condiții printr-un **cursor** în tabloul **Zona**, cursor care indică nodul rădăcină al arborelui.
- Acest mod de reprezentare este însă impropriu implementării operatorului **Tata**, care presupune baleerea integrală a tabloului **Zona**.

- Pentru a implementa în mod eficient și acest operator se adaugă un al 4-lea câmp (tata), care indică direct părintele nodului în cauză.
- În fig.8.1.4.3.b, apare reprezentarea arborelui generalizat din dreapta figurii, iar în secvența [8.1.4.3.b] apare definiția formală a structurii de date corespunzătoare acestui mod de implementare al arborilor generalizați.

---

/\*Reprezentarea arborilor generalizați bazată pe relațiile primul-fiu și frate-dreapta (Varianta 2) C\*/

```

int MaxNod=...;

typedef int tip_cursor;
typedef tip_cursor tip_arbore;

typedef struct linie_tablou {      /*structura linie tablou*/
    tip_cursor primulFiu;
    tip_cheie cheie;
    tip_cursor frateDreapta;          /*[8.1.4.3.b]*/
    tip_cursor tata;
} linie_tablou;

linie_tablou Zona[MaxNod];      /*definire tablou Zona*/
tip_cursor Disponibil; /*cursor inceput lista disponibili*/
tip_arbore R;

```

---

- În secvența [8.1.4.3.c] este prezentat un exemplu de implementare a operatorului **Creaza<sub>2</sub>**, pornind de la reprezentarea propusă.
- Se presupune că există o listă a liberilor în tabloul Zona, indicată prin cursorul Disponibil, în cadrul căreia elementele sunt înlăntuite prin intermediul câmpului frateDreapta.
- Lista Disponibil este utilizată pentru alocarea/eliberarea dinamică de către utilizator a nodurilor în tabloul Zona.

---

/\*Exemplu de implementare al operatorului Creaza2\*/

```

tip_arbore Creaza2(tip_cheie v, tip_arbore t1,t2)
{
    tip_cursor temp; /*păstrează indexul rădăcinii noului
                        arbore*/
    temp=Disponibil; /*alocare nod nou*/
    Disponibil=Zona[Disponibil].frateDreapta;
    Zona[temp].primulFiu=t1;           [8.1.4.3.c]
    Zona[temp].cheie=v;
    Zona[temp].frateDreapta=0;   Zona[temp].tata=0;
    Zona[t1].frateDreapta=t2;   Zona[t1].tata=temp;
    Zona[t2].frateDreapta=0;   Zona[t2].tata=temp;
    return temp;
} /*Creaza2*/

```

---

## 8.2. Arbori binari

### 8.2.1. Definiții

- Prin arbore binar se înțelege o mulțime de  $n \geq 0$  noduri care dacă nu este vidă, conține un anumit nod numit **rădăcină**, iar restul nodurilor formează doi arbori binari disjuncti numiți: **subarborele stâng** respectiv **subarborele drept**.
- Ca și exemple pot fi considerați arborii binari reprezentați în figura 8.2.1.a.

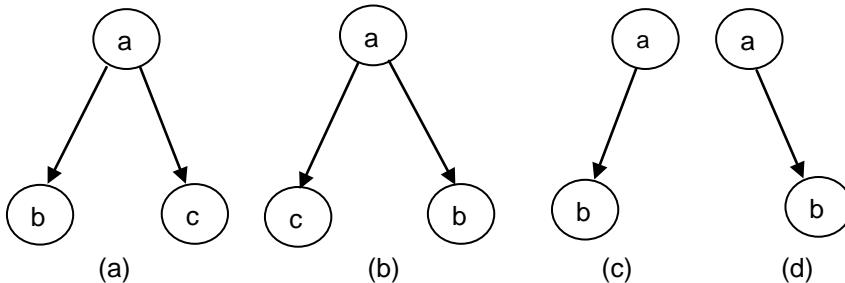


Fig.8.2.1.a. Structuri de arbori binari

- Structurile (a) și (b) din fig.8.2.1.a. deși conțin noduri identice, reprezintă arbori binari **diferiți** deoarece se face o distincție netă între subarborele stâng și subarborele drept al unui arbore binar.
- Acest lucru este pus și mai pregnant în evidență de structurile (c) și (d) din fig.8.2.1.a. care de asemenea reprezintă arbori **diferiți**.
  - O expresie aritmetică obișnuită poate fi reprezentată cu ajutorul unui arbore binar întrucât operatorii aritmetici sunt operatori binari..
  - Spre exemplu, pentru expresia  $(a+b/c)*(d-e*f)$ , arboarele binar corespunzător care se mai numește și arbore de parcursere (“parse tree”), apare în figura 8.2.1.b.

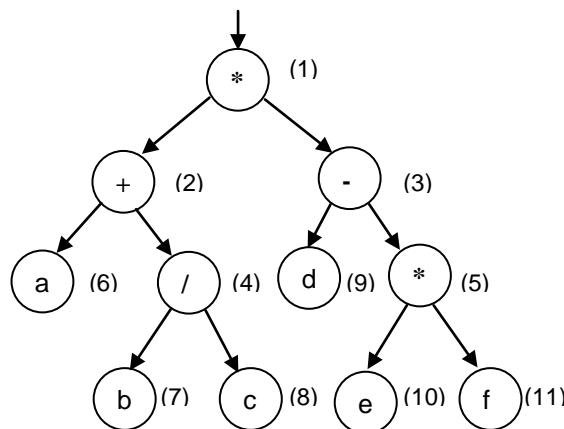
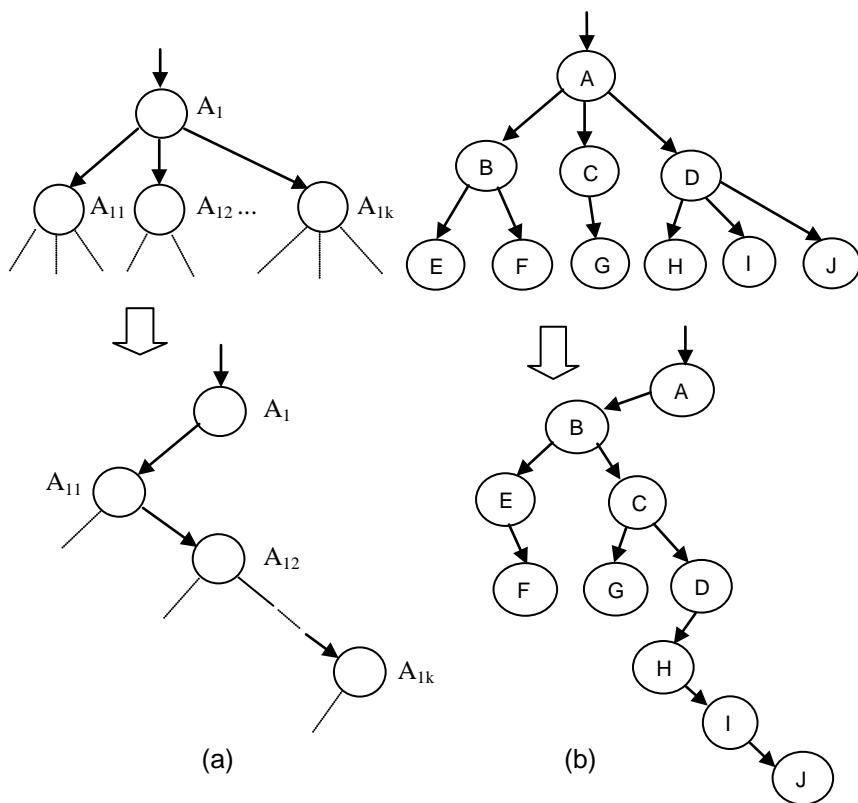


Fig.8.2.1.b. Arbore binar asociat unei expresii aritmetice

- Structura **arbore binar** este deosebit de importantă deoarece:
  - (1) Este simplu de **reprezentat**
  - (2) Este ușor de **prelucrat**, bucurându-se de o serie de proprietăți specifice
  - (3) **Orice structură arbore**, poate fi transformată într-o structură **arbore binar**.

### 8.2.2. Tehnica transformării unei structuri arbore generalizat într-o structură arbore binar.

- Fie un **arbore generalizat** oarecare  $A$ , care are rădăcina  $A_1$  și subarborii  $A_{11}, A_{12} \dots, A_{1k}$ .
- **Transformarea** acestuia într-un **arbore binar** se realizează după cum urmează:
  - (1) Se ia  $A_1$  drept rădăcină a arborelui binar.
  - (2) Se face subarborele  $A_{11}$  fiul său stâng.
  - (3) Fiecare subarbore  $A_{1i}$  se face fiul drept al lui  $A_{1,i-1}$  pentru  $2 \leq i \leq k$ .
  - (4) Se continuă în aceeași manieră transformând după același algoritm fiecare din subarborii rezultați, până la parcurgerea integrală a arborelui inițial.
- Grafic, această tehnică apare reprezentată în figura 8.2.2.a, (a)-cazul general și (b)-un exemplu.



### 8.2.3. TDA Arbore binar

- Tipul de date abstract **arbore binar**, ca de altfel orice TDA presupune:
  - (1) Definirea **modelului matematic** asociat structurii arbore binar.
  - (2) Definirea **setului de operatori** care gestionează structura.
- Ca și în cadrul altor TDA-uri, este greu de stabilit un set de operatori care să ofere satisfacție în toate clasele de aplicații posibile.
- În cadrul cursului de față se propune pentru **TDA Arbore binar** forma prezentată în [8.2.3.a].

---

#### TDA Arbore binar

**Modelul matematic:** o structură de noduri. Toate nodurile sunt de același tip. Fiecare nod este rădacina unui subarbore și este alcătuit din trei câmpuri: element, indicator stâng și indicator drept. Indicatorii precizează subarborele stâng respectiv subarborele drept al subarborelui în cauza. În cadrul câmpului element poate exista un câmp cheie care identifică nodul.

#### **Notății:**

*TipNod* - tipul asociat nodurilor arborelui

*TipArboreBinar* - tipul arbore binar

*TipIndicatorNod* - indicator la nod

*TipArboreBinar t,s,d;*

*TipIndicatorNod r,n;*

*TipNod w;*

*boolean b;*

[ 8 . 2 . 3 . a ]

#### **Operatori:**

1. **CreazaArboreVid**(*TipArboreBinar \* t*) - procedură care crează arborele vid *t*;
2. *boolean ArboreVid*(*TipArboreBinar t*) - funcție care returnează valoarea adevărat dacă arborele *t* este vid și fals în caz contrar.
3. *TipIndicatorNod Radacina*(*TipIndicatorNod n*) - returnează indicatorul rădăcinii arborelui binar căruia îi aparține nodul *n*;
4. *TipIndicatorNod Parinte*(*TipIndicatorNod n, TipArboreBinar t*) - returnează indicatorul părintelui(tatălui) nodului *n* aparținând arborelui *t*;

5. *TipIndicatorNod FiuStanga*(*TipIndicatorNod n, TipArboreBinar t*) - returnează indicatorul fiului stâng al nodului *n* aparținând arborelui *t*;
  6. *TipIndicatorNod FiuDreapta*(*TipIndicatorNod n, TipArboreBinar t*) - returnează indicatorul fiului drept al nodului *n* aparținând arborelui *t*;
  7. ***SuprimaSub***(*TipIndicatorNod n, TipArboreBinar t*); - suprimă subarborele a cărui radacină este nodul *n*, aparținând arborelui binar *t*;
  8. ***InlocuiesteSub***(*TipIndicatorNod n, TipArboreBinar r,t*) - inserează arborele binar *r* în *t*, cu rădacina lui *r* localizată în poziția nodului *n*, înlocuind subarborele indicat de *n*. Operatorul se utilizează de regulă când *n* este o frunză a lui *t*, caz în care poate fi asimilat cu operatorul *adaugă*;
  9. *TipArboreBinar Creaza2*(*TipArboreBinar s,d, TipIndicatorNod r*) - crează un arbore binar nou care are nodul *r* pe post de rădăcină și pe *s* și *d* drept subarbore stâng respectiv subarbore drept
  10. *TipNod Furnizeaza*(*TipIndicatorNod n, TipArboreBinar t*) - returnează conținutul nodului indicat de *n* din arborele binar *t*. Se presupune că *n* există în *t*;
  11. ***Actualizeaza***(*TipIndicatorNod n, TipNod w, TipArboreBinar t*); - înlocuiește conținutul nodului indicat de *n* din *t* cu valoarea *w*. Se presupune că *n* există.
  12. *TipIndicatorNod Cauta*(*TipNod w, TipArboreBinar t*) - returnează indicatorul nodului aparținând arborelui binar *t*, al cărui conținut este *w*;
  13. ***Preordine***(*TipArboreBinar t, Vizitare(listaArgumente)*) - operator care realizează parcurgerea în preordine a arborelui binar *t* aplicând fiecărui nod, procedura *Vizitare*;
  14. ***Inordine***(*TipArboreBinar t, Vizitare(listaArgumente)*) - operator care realizează parcurgerea în inordine a arborelui binar *t* aplicând fiecărui nod, procedura *Vizitare*;
  15. ***Postordine***(*TipArboreBinar t, Vizitare(listaArgumente)*) - operator care realizează parcurgerea în postordine a arborelui binar *t* aplicând fiecărui nod, procedura *Vizitare*.
- 

#### 8.2.4. Tehnici de implementare a arborilor binari

- În aplicațiile curente se utilizează **două** modalități de implementare a structurii arboare binar:

- (1) Implementarea bazată pe **tablouri**, (mai rar utilizată).
- (2) Implementarea bazată pe **pointeri**.

#### 8.2.4.1. Implementarea arborilor binari cu ajutorul structurii tablou

- Pentru implementarea unui arbore binar se poate utiliza o structură **tablou liniar** definit după cum urmează [8.2.4.1.a]:

---

```
/*Implementarea unui arbore binar cu ajutorul unei structuri
tablou liniar*/
```

```
typedef struct element {
    char op;
    int stang,drept; /*[8.2.4.1.a]*/
} tip_element

typedef tip_element tip_arbore[MaxNod];

tip_arbore t;
```

---

- În prealabil fiecărui nod al arborelui i se asociază în mod **aleator** o intrare în tabloul liniar.
- În principiu, acesta este o implementare bazată pe **cursori**.
- În fig.8.2.4.1.a apare o astfel de reprezentare a unui arbore binar.

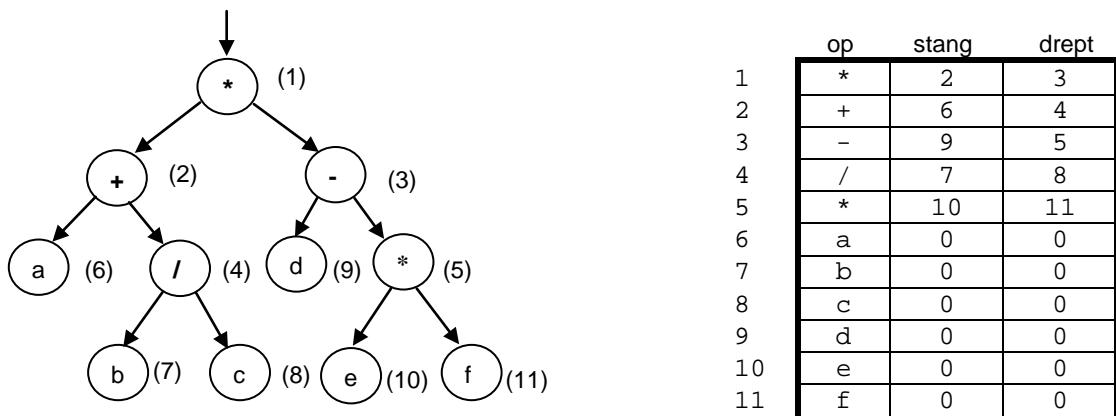


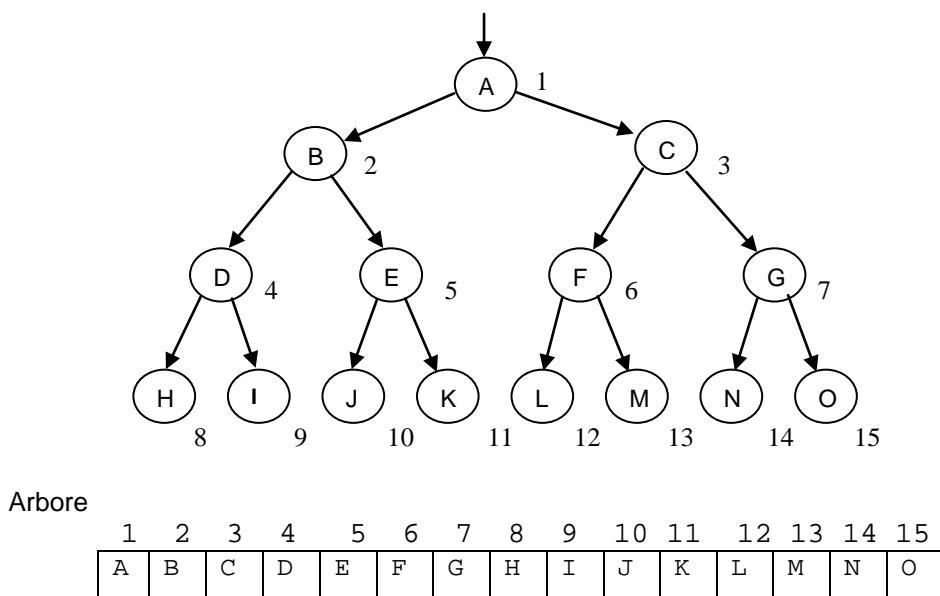
Fig.8.2.4.1.a. Reprezentarea unui arbore binar cu ajutorul unui tablou liniar

- Un alt mod de reprezentare cu ajutorul structurilor tablou se bazează pe următoarele două **leme**.
  - **Lema 1.** Numărul maxim de noduri al nivelului  $i$  al unui arbore binar este  $2^{i-1}$ .
  - Ca atare, numărul maxim de noduri al unui arbore de înălțime  $h$  este [8.2.4.1.b].
-

$$NrMax = \sum_{i=1}^h 2^{i-1} = 2^h - 1 \quad \text{pentru } h > 0$$

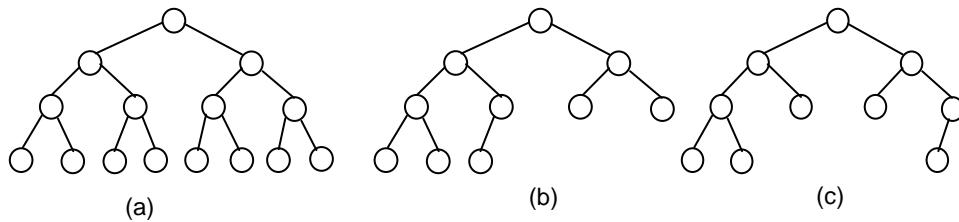
[ 8 . 2 . 4 . 1 . b ]

- Arborele binar de înălțime  $h$  care are exact  $2^h - 1$  noduri se numește **arbore binar plin de înălțime  $h$** .
- Se **numerotează** secvențial nodurile unui arbore binar plin de înălțime  $h$ , începând cu nodul situat pe **nivelul 1** și continuând numărătoarea nivel cu nivel de **sus în jos**, și în cadrul fiecărui nivel, de la **stânga la dreapta**.
- Pornind de la această numerotare se poate realiza o **implementare elegantă** a structurii arbore binar, **asociind** fiecărui nod al arborelui, locația corespunzătoare numărului său într-un **tablou liniar** (fig.8.2.4.1.b.).



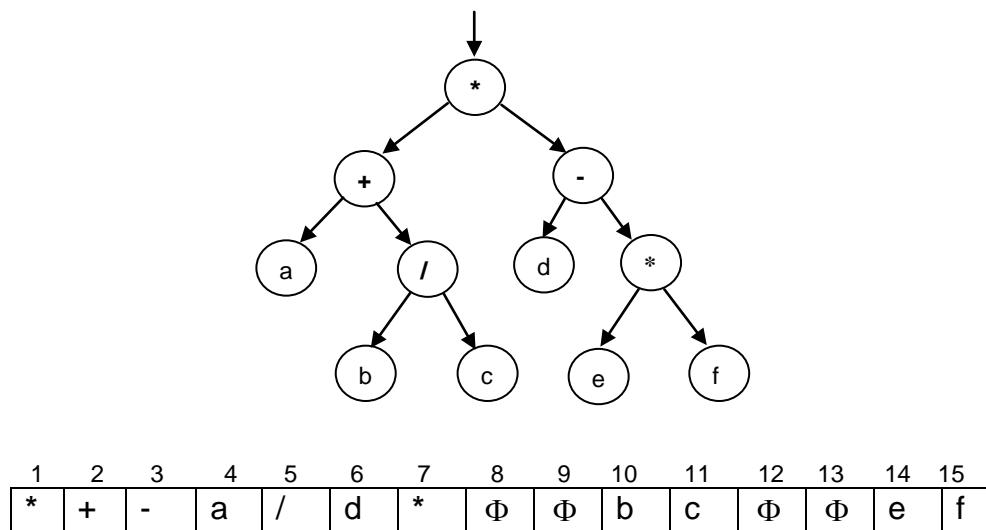
**Fig.8.2.4.1.b.** Arbore binar plin și reprezentarea sa cu ajutorul unui tablou liniar

- Un arbore binar cu  $n$  noduri și de înălțime  $h$  se numește **arbore binar complet** dacă nodurile sale corespund nodurilor care sunt numerotate de 1 la  $n$  într-un arbore binar plin de înălțime  $h$ .
- O consecință a acestei definiții este aceea că într-un **arbore binar complet**, nodurile terminale apar pe **cel mult două niveluri adiacente**.
- Asemenea arborelui binar plin și arborele complet poate fi memorat într-un **tablou liniar** de dimensiune  $n$ .
- Diferența dintre un **arbore binar complet** și un **arbore binar plin**, ambele de înălțime  $h$ , este aceea că primul are un număr  $n$  de noduri unde  $n < 2^h - 1$ , iar cel de-al doilea are **exact**  $2^h - 1$  noduri.
- Ambii arbori sunt de fapt **arbori binari de înălțime minimă** care se bucură de proprietatea că nodurile lor terminale sunt distribuite pe **cel mult două niveluri** ale arborelui (fig.8.2.4.1.c)



**Fig.8.2.4.1.c.** Arboare binare: plin (a), complet (b), de înălțime minimă (echilibrat) (c)

- Lema următoare precizează maniera deosebit de simplă în care se poate determina părintele, fiul stâng și fiul drept al unui nod precizat, fără memorarea explicită a nici unui fel de informație de legătură.
  - Lema 2.** Dacă se reprezintă un **arbore binar complet** într-o manieră conformă cu cele anterior precizate, atunci pentru orice nod având indicele  $i, 1 \leq i \leq n$  sunt valabile relațiile:
    - $\text{Parinte}(i)$  este nodul având indicele  $\lfloor i/2 \rfloor$  dacă  $i \neq 1$ . Dacă  $i=1$  nodul indicat de  $i$  este nodul rădăcină care nu are părinte.
    - $\text{FiuStanga}(i)$  este nodul având indicele  $2*i$  dacă  $2*i \leq n$ . Dacă  $2*i > n$ , atunci nodul  $i$  nu are fiu stâng.
    - $\text{FiuDreapta}(i)$  este nodul având indicele  $2*i+1$  dacă  $2*i+1 \leq n$ . Dacă  $2*i+1 > n$ , atunci nodul  $i$  nu are fiu drept.
  - Acest mod de implementare poate fi în mod evident utilizat pentru **orice structură arbore binar**, în marea majoritate a cazurilor rezultând însă o utilizare **ineficientă** a zonei de memorie alocate tabloului.
  - Spre exemplu în fig.8.2.4.1.d apare reprezentarea arborelui binar din fig.8.2.4.1.a. În acest caz se fac următoarele precizări:
    - h este **cea mai mică înățime de arbore binar plin** care "cuprinde" arborele în cauză;
    - Se numerotează și **nodurile lipsă** ale arborelui de reprezentat, ele fiind înlocuite cu nodul vid  $\Phi$  în cadrul reprezentării.



**Fig.8.2.4.1.d.** Reprezentarea unui arbore binar oarecare

- Se remarcă similitudinea dintre acest mod de reprezentare și reprezentarea **ansamblelor** (Vol.1 &3.2.5).
- Această manieră de reprezentare a arborilor binari este bună pentru **arborii compleți** și este în general **ineficientă** pentru toți ceilalți.
- De asemenea, ea suferă și de **neajunsul** specific reprezentărilor bazate pe **tablouri liniare** și anume, dimensiunea fixă a zonei alocate.
- **Concluzionând**, referitor la această manieră de implementare a arborilor binari se pot face următoarele precizări:
  - **Avantaje:** Simplitate, absența legăturilor; parcurgerea simplă a arborelui în ambele sensuri.
  - **Dezavantaje:** Dimensiunea limitată, implementarea relativ complicată a modificărilor (inserții, suprimări), utilizare ineficientă în cazul arborilor rari.
  - **Se recomandă:** În cazul arborilor binari cu o dinamică redusă a modificărilor, în care se realizează multe parcurgeri sau căutări.

#### **8.2.4.2. Implementarea arborilor binari cu ajutorul pointerilor**

- Maniera cea mai naturală și cea mai flexibilă de reprezentare a **arborilor binari** este cea bazată pe **TDA Indicator**.
- Acest mod de reprezentare are două **avantaje**:
  - Pe de-o parte înlătură **limitările** reprezentării secvențiale bazate pe structura tablou.
  - Pe de altă parte face uz de proprietățile **recursive** ale structurii arbore, implementarea realizându-se cu ajutorul **structurilor de date dinamice**.
- Un arbore binar poate fi descris cu ajutorul unei structuri de date dinamice în varianta C (secvența [8.2.4.2.a]) respectiv în varianta Pascal (secvența [8.2.4.2.b]).

---

**/\*Implementarea arborilor binari cu ajutorul pointerilor\*/**

```

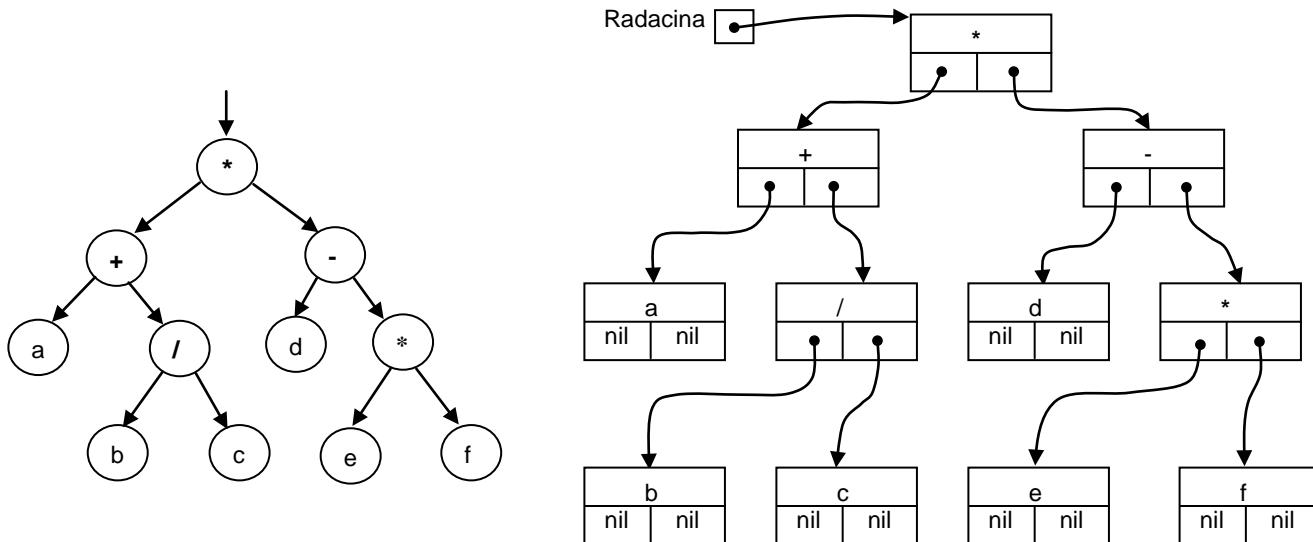
typedef struct tip_nod
{
    /*diferite câmpuri*/
    char cheie;
    struct tip_nod* stang;           /*[8.2.4.2.a]*/
    struct tip_nod* drept;
} tip_nod;

typedef tip_nod * ref_tip_nod;

```

---

- În fig.8.2.4.2.a se poate urmări reprezentarea **arborelui binar** din figura 8.2.4.1.a bazată pe definiția din secvența [8.2.4.2.a].
- Este ușor de văzut că practic **orice arbore** poate fi reprezentat în acest mod.
- În cadrul cursului de față, această modalitate de reprezentare va fi utilizată în mod preponderent.



**Fig.8.2.4.2.a.** Arbore binar reprezentat cu ajutorul pointerilor

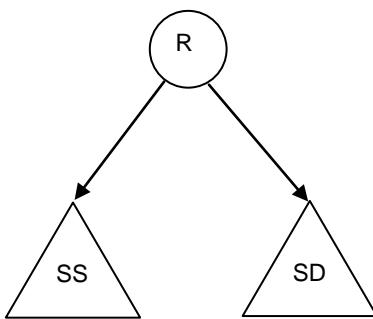
### 8.2.5. Traversarea arborilor binari

- Referitor la **tehniciile de traversare a arborilor binari** se face precizarea că ele derivă direct prin **particularizare** din tehniciile de traversare a **arborilor generalizați**.
- În consecință și în acest caz se disting tehniciile bazate pe **căutarea în adâncime** (preordine, inordine și postordine) precum și tehnica **căutării prin cuprindere**.

#### 8.2.5.1. Traversarea arborilor binari prin tehnici bazate pe căutarea în adâncime

- Traversarea în adâncime (**depth first**), aşa cum s-a precizat și în cadrul arborilor generalizați, presupune **parcursarea în adâncime** a arborelui binar atât de departe cât se poate plecând de la rădăcină.
- Când s-a ajuns la o frunză, **se revine** pe drumul parcurs la proximul nod care are fii neexplorați și parcursarea se reia în aceeași manieră până la traversarea integrală a structurii.
- Schema este bună dar presupune **memorarea** drumului parcurs, ca atare necesită fie o **implementare recursivă** fie utilizarea unei **stive**.
- **Traversarea în adâncime** are trei variante: **preordine, inordine și postordine**.

- Considerările avute în vedere la traversarea arborilor generalizați rămân valabile și pentru arborii binari.
- Astfel, referindu-ne la arboarele binar din fig.8.2.5.1.a și considerând pe R drept rădacină, iar pe SS și SD drept subarborele său stâng, respectiv subarborele său drept, atunci cele **trei moduri de parcursare** sunt evidențiate de modelele recursive din secvența [8.2.5.1.a].



**Fig.8.2.5.1.a.** Model de arbore binar

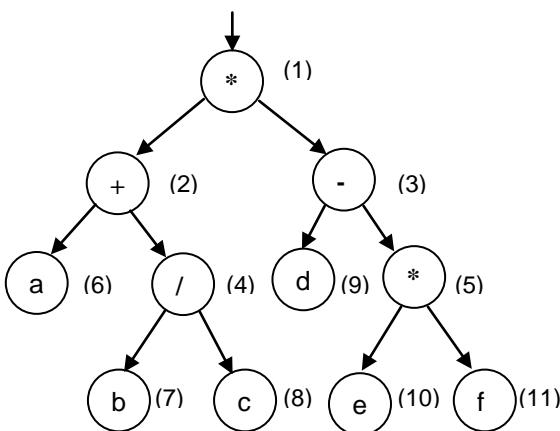
**Modele recursive pentru traversarea arborilor binari prin tehnica căutării în adâncime**

```

Preordine(A) : R, Preordine(SS), Preordine(SD)
Inordine(A)  : Inordine(SS), R, Inordine(SD)           [ 8 . 2 . 5 . 1 . a ]
Postordine(A): Postordine(SS), Postordine(SD), R
  
```

---

- Traversarea unei structuri de date este de fapt o **ordonare liniară** a componentelor sale în baza unui anumit **protocol**.
- Spre **exemplu**, traversând arboarele care memorează **expresia aritmetică**  $(a+b/c)*(d-e*f)$  din fig.8.2.1.b și înregistrând caracterul corespunzător pe măsură ce sunt vizitate nodurile arborului, se obțin următoarele ordonări:
  - Preordine: **\*+a/bc-d\*ef**
  - Inordine: **a+b/c\*d-e\*f**
  - Postordine: **abc/+def\*-\***



**Fig.8.2.1.b.** Arbore binar corespunzător unei expresii aritmetice (reluare)

- Se precizează faptul că ultima ordonare este cunoscută în matematică sub denumirea de **notație poloneză (postfix)**.
- Prin scrierea unei expresii aritmetice în **notație poloneză** se înțelege scrierea operatorului **după** cei doi operanzi, în loc de a-l scrie între ei, ca în notația algebraică obișnuită (**infix**).
- Astfel spre exemplu:

<b>a+b</b>	devine	<b>ab+</b>
<b>a*b+c</b>	devine	<b>ab*c+</b>
<b>a*(b+c)</b>	devine	<b>abc+*</b>
<b>a-b/c</b>	devine	<b>abc/-</b>
<b>(a-b)/c</b>	devine	<b>ab-c/</b>

- Este ușor de observat că **forma poloneză** a unei expresii reprezentate printr-un arbore binar se obține **parcurgând** arborele în **postordine**.
- În mod similar, în matematică se definește **notație poloneză inversă (prefix)** în care operatorul se scrie **în fața** celor doi operanzi. De exemplu expresia **a+b** se scrie **+ab**.
- Se observă de asemenea că **notația prefix** corespunde **traversării** în **preordine** a arborelui expresiei.
- O proprietate interesantă a notației poloneze este aceea că în ambele forme ea păstrează **semnificația** expresiei aritmetice **fără** a utiliza **paranteze**.
- Cu privire la ordonarea în **inordine**, se face precizarea că ea corespunde notației obișnuite a expresiei aritmetice dar cu **omiterea parantezelor**, motiv pentru care semnificația matematică a expresiei se pierde, cu toate că arborele binar corespunzător păstrează în structura sa această semnificație (vezi fig.8.2.1.b).
- În continuare, cele trei metode de traversare vor fi concretizate în trei **proceduri recursive** în care:
  - **r** este o **variabilă** de tip pointer care indică rădăcina arborelui.
  - **Viziteaza**(**r^**) reprezintă **operația** care trebuie executată asupra fiecărui nod.
- Considerând pentru structura **arbore binar** definiția din secvența [8.2.5.1.b], structura de principiu a celor trei metode de traversare este prezentată în [8.2.5.1.c].

---

**/\*Traversarea arborilor binari\*/**

```

typedef struct tip_nod /*definire structură arbore binar*/
{
    /*diferite câmpuri*/
    char cheie;
    struct tip_nod* stang;      /*[8.2.5.1.b]*/
    struct tip_nod* drept;
} tip_nod;
```

```

typedef tip_nod* ref_tip_nod;
-----
void Preordine(ref_tip_nod r) /*traversare in preordine*/
{
    if (r!=NULL)
    {
        Viziteaza(*r);
        Preordine((*r).stang); /* Preordine(r->stang); */
        Preordine((*r).drept); /* Preordine(r->drept); */
    }
} /*Preordine*/

void Inordine(ref_tip_nod r) /*traversare in inordine*/
{
    if (r!=NULL)
    {
        Inordine((*r).stang); /*[8.2.5.1.c]*/
        Viziteaza(*r);
        Inordine((*r).drept);
    }
} /*Inordine*/

void Postordine(ref_tip_nod r) /*traversare in postordine*/
{
    if (r!=NULL)
    {
        Postordine((*r).stang);
        Postordine((*r).drept);
        Viziteaza(*r);
    }
} /*Postordine*/
-----
```

### 8.2.5.2. Traversarea arborilor binari prin tehnica căutării prin cuprindere

- Traversarea prin **cuprindere** (**breadth-first**) presupune traversarea tuturor nodurilor de pe un nivel al structurii arborelui, după care se trece la nivelul următor parcurgându-se nodurile acestui nivel și.a.m.d până la epuizarea tuturor nivelurilor arborelui.
- În cazul arborilor binari, rămâne valabilă **schema de principiu** a parcurgerii prin cuprindere bazată pe utilizarea unei structuri de date **coadă**, prezentată pentru arborii generalizați (secvența [8.1.3.2.a]), cu precizarea că nodurile pot avea cel mult **două fiicări**.
- În continuare se prezintă o **altă variantă** de parcurgere care permite efectiv **evidențierea nivelurilor arborelui binar**, variantă care utilizează în tandem **două structuri coadă** (secvența [8.2.5.2.a]).
  - La parcurgerea nodurilor unui nivel al arborelui binar, noduri care sunt memorate într-o coadă din cozi, fiile acestora se adaugă celelalte cozi.
  - La terminarea unui nivel (golirea cozii curente parcurse), cozile sunt comutate și procesul se reia până la parcurgerea integrală a arborelui binar, adică golirea ambelor cozii.

**Traversarea prin cuprindere unui arbore binar cu evidențierea nivelurilor acestuia - varianta pseudocod**

```
subprogram TraversarePrinCuprindereArboreBinar(tip_nod
radacina)

/*Se utilizeaza TDA Arbore binar si TDA Coada*/
{
    tip_coada coada1,coada2;

    r=radacina;
    Initializeaza(coada1); Initializeaza(coada2);
    daca (r nu este nodul vid) atunci
        Adauga(r,coada1); /*procesul de amorsare*/
    repetă
        cât timp (NOT Vid(coada1))execută
            r<-Cap(coada1); Scoate(coada1);
            *listea(r);
            dacă (FiuStanga(r) nu este nodul vid) atunci
                Adauga(FiuStanga(r),coada2);
            dacă (FiuDreapta(r) nu este nodul vid) atunci
                Adauga(FiuDreapta(r),coada2);
        □
        cât timp (NOT Vid(coada2))execută [ 8.2.5.2.a ]
            r<-Cap(coada2); Scoate(coada2);
            *listea(r);
            dacă (FiuStanga(r) nu este nodul vid) atunci
                Adauga(FiuStanga(r),coada1);
            dacă (FiuDreapta(r) nu este nodul vid) atunci
                Adauga(FiuDreapta(r),coada1);
        □
    până când (Vid(coada1) AND Vid(coada2))
}
```

### 8.2.6. Arbori binari cu legături ("Threaded Trees")

- Implementările bazate pe **indicatori** (cursori sau pointeri) permit parcurgerea simplă în manieră **descendentă** a unui arbore în (variantă recursivă sau iterativă) dar fac practic relativ dificilă **parcurgerea ascendentă**.
- În principiu această problemă poate fi rezolvată, memorând în fiecare moment al parcurgerii pe lângă **nodul curent și părintele acestuia**:
  - **Dezavantaj:** se realizează un **consum mai ridicat de memorie**.
  - **Avantaj:** se reduce gradul de complexitate al algoritmului de parcurgere.
- În acest caz, pentru a **reduce** consumul de memorie poate fi avantajos să se modifice în mod convenabil structura de date, acceptând o creștere corespunzătoare a complexității algoritmilor de prelucrare.
- O **primă soluție** pentru a realiza parcurgerea ascendentă simplă a unui arbore este aceea de a **memora** pentru fiecare nod **în parte** legătura la **părintele său**, rezolvând această problemă în maniera listelor dublu **înlănțuite**.

- Dacă nodul are o structură suficient de complexă, un câmp de înlățuire suplimentar **nu** conduce la un consum excesiv de memorie [8.2.6.a].

---

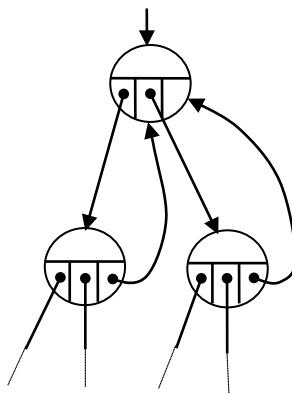
/\*Arborei binari cu trei indicatori - structura de date\*/

```
typedef struct tip_nod
{
    tip_info info;
    struct tip_nod * stang; /*[8.2.6.a]*/
    struct tip_nod * drept;
    struct tip_nod * parinte;
} tip_nod;

typedef tip_nod * ref_tip_nod;
```

---

- Reprezentarea grafică a acestei soluții apare în figura 8.2.6.a.



**Fig.8.2.6.a.** Arbore binar cu legături

- O altă posibilitate larg utilizată în implementarea arborilor binari se bazează pe observația că o mare parte a **nodurilor** unei structuri arbore au câmpurile indicator nule.
- Aceste câmpuri pot fi **modificate** astfel încât ele să indice noduri în arbore care **preced** sau **succed** nodul în cauză într-o anumită ordine de traversare.
  - Spre exemplu, referindu-ne la traversarea în **inordine**, se poate conveni ca în fiecare nod, în locul **înlățuirii vide pe dreapta** să apară legătura la **nodul următor** la parcursarea în **inordine** iar în locul **înlățuirii vide pe stânga** să apară legătura la **nodul anterior** în aceeași parcursare.
  - Întrucât câmpurile indicator au fost **suprâncărcate** ("overloaded") este necesar ca în structura nodului să fie efectuate modificări care să evidențieze natura acestor câmpuri: **pointeri** (înlățuri) sau **legături** ("threads").
  - Se precizează faptul că **pointerii** indică fii ai nodului în cauză, iar **legăturile** predecesori sau succesorii la parcursarea în inordine.
- O astfel de structură arbore se numește **arbore cu legături** ("threaded tree").

- Un arbore cu legături poate fi:

- (1) **Parțial** (dacă numai **unul** din câmpurile unui nod este folosit ca legătură).
- (2) **Plin** ("fully threaded tree") dacă se utilizează ambele câmpuri.

### 8.2.6.1. Arbori binari cu legături plini ("Fully Threaded Trees")

- O posibilitate de implementare a unui **arbore cu legături plin** este cea precizată în secvența [8.2.6.1.a].

---

**/\*Arbori binari cu legături plini (Fully Threaded Trees)\*/**

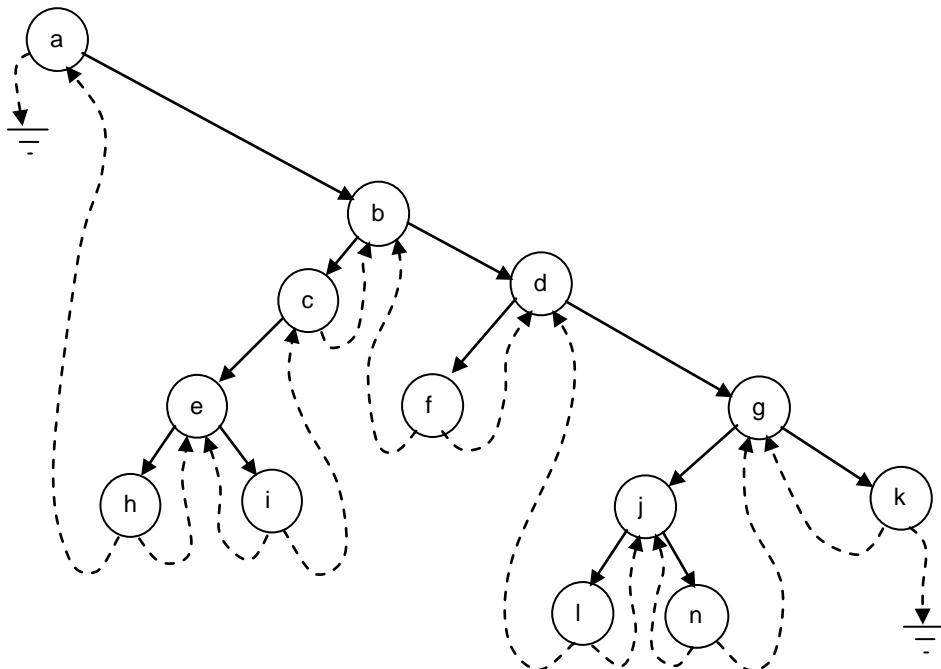
```
typedef tip_nod_legaturi * ref_tip_nod;

typedef struct tip_nod_legaturi
{
    tip_info info;
    boolean legatura_stg, legatura_dr;
    ref_tip_nod stang;                                /*[8.2.6.1.a]*/
    ref_tip_nod drept;
} tip_nod_legaturi;

typedef ref_tip_nod tip_arbore_binar_leg;
```

---

- Câmpurile booleene legatura\_stg respectiv legatura\_dr au valoarea **adevărat** dacă câmpurile indicator corespunzătoare sunt legături respectiv au valoarea **fals** când acestea sunt înlănuiri.
- Un **exemplu** de astfel de implementare apare în figura 8.2.6.1.a. **Legăturile** sunt marcate cu linii **întrerupte** iar **pointerii** cu linii **continue**.



**Fig.8.2.6.1.a.** Arbore cu legături plin

- Ținând cont de faptul că parcurgerea în inordine a arborelui din figura 8.2.6.1.a. este:  
**a,h,e,i,c,b,f,d,l,j,n,g,k**
- Se observă că:
  - Pentru nodurile terminale, legăturile precizează **predecesorul** (legătura pe stânga) respectiv **succesorul** (legătura pe drepta) nodului în cauză la parcurgerea în inordine, spre exemplu e respectiv c pentru nodul i .
  - Cel mai din **stânga nod** al fiecărui **subarbore drept** indică prin legătura sa pe stânga **părintele** arborelui (h pe a sau l pe d), după cum același lucru este valabil și pentru cel mai din **dreapta nod** al fiecărui **subarbore stâng** (n pe g).
  - **Cel mai din stânga și cel mai din dreapta** nod al structurii au legătura pe stânga respectiv pe dreapta **vidă** (nodurile a și k ).
- Se menționează faptul că **parcurgerea în inordine** poate fi evidențiată practic simplu realizând **proiecția pe abscisă** a nodurilor arborelui.
- Un **prim avantaj** al acestui mod de reprezentare este acela că permite **traversarea în inordine** a arborelui binar în **manieră iterativă**.
  - Într-adevăr, în această traversare **primul** nod vizitat este cel mai din stânga nod al subarborelui stâng.
  - După vizitarea acestui nod, urmează **părintele** acestuia la care se ajunge direct urmând legătura pe dreapta a nodului respectiv.
  - După vizitarea părintelui, se trece la parcurgerea în aceeași manieră a **subarborelui drept**.
  - Se continuă în aceeași manieră până la vizitarea **celui mai din dreapta** nod al întregii structuri, nod care este singurul care are legătura pe dreapta vidă.
- Procedura care ilustrează această traversare este **InorderIterativ** și apare în secvența [8.2.6.1.b].

---

**/\*Traversarea unui arbore cu legături - varianta iterativă\*/**

```
void InorderIterativ (ref_tip_nod n);
/*Traversează arborele binar cu legături în inordine începând
cu nodul n*/
{
    while (n != NULL) /*până la atingerea celui mai din dreapta
nod*/
    {
        while ((!n->legatura_stg) && (n->stang!=NULL))
            n=n->stang; /*avansare pe stânga cât este posibil*/
        viziteaza(n); /*vizitare cel mai din stânga nod*/
        while (n->legatura_dr) /*urcare în arbore în inordine*/
        {
            if (n->legatura_stg)
                n=n->legatura_stg;
            else
                n=n->legatura_dr;
        }
    }
}
```

```

        n=n->drept;
        viziteaza(n);
    } /*while*/
    n=n->drept; /*se trece la subarborele drept*/
} /*while*/
} {InorderIterativ}

```

---

- Un al doilea avantaj al acestui nod de traversare este acela că traversarea poate demara cu **oricare** nod al arborelui, lucru care este practic imposibil în varianta recursivă.
- În această reprezentare **operatorii** definiți de TDA arbore binar se implementează într-o **manieră similară** cu reprezentarea bazată pe **pointeri**.
- **Diferența** apare la inserția sau suprimarea unui subarbore.
  - Se consideră spre exemplu operatorul **InlocuiesteSub**(*n,r,t*) unde *n* este de TipIndicatorNod, iar *r* și *t* de TipArboreBinar.
  - La inițiativa acestui operator, **subarborele** având rădăcina *r* trebuie plasat în locul precizat de indicatorul *n*.
  - Pentru simplificarea lucrurilor, se va ignora posibilitatea de a **reutiliza** spațiul alocat subarborelui indicat de *n* precum și eventualele supraîncărări (“aliasing”).
  - **Insetria** subarborelui indicat de *r* presupune de fapt poziționarea **legăturii** pe **dreapta a celui mai din dreapta nod** al subarborelui *r* astfel încât să indice poziția precizată de nodul similar din subarborele *n* și în mod analog pentru **cel mai din stânga nod** al subarborelui *r*, după cum rezultă din figura 8.2.6.1.b.
- În secvența [8.2.6.1.c] apare implementarea operatorului **InlocuiesteSub**.

---

#### **/\*Operatorul Inlocuieste Subarbore - varianta C\*/**

```

void InlocuiesteSub(ref_tip_nod n, tip_arbore_binar_leg r, t)
{
    ref_tip_nod cel_mai_stg,cel_mai_dr; /*noduri extreme în
                                             subarborele n*/
    ref_tip_nod r_stg,r_dr; /*noduri extreme în subarborele r*/
    /*caută nodurile extreme în arborele original n*/

    /*determinare nod extrem stânga*/
    cel_mai_stg=n;
    do
        cel_mai_stg= cel_mai_stg->stang;
    while ((cel_mai_stg->legatura_stg) ||
           (cel_mai_stg->stang == NULL));

    /*determinare nod extrem dreapta*/
    cel_mai_dr=n;
    do
        cel_mai_dr= cel_mai_dr->drept;
    while ((cel_mai_dr->legatura_dr) ||
           (cel_mai_dr->drept == NULL));

```

---

```

while ((cel_mai_dr->legatura_dr) ||  

        (cel_mai_dr->drept == NULL));  
  

/*înlocuiește nodul n cu nodul r*/  

n=r;  
  

/*caută și modifică legăturile extreme ale lui r*/  
  

/*modificare legătură extremă pe stânga*/  

r_stg=n;  

while (r_stg->stang!=NULL)  

    r_stg=rStg->stang;  

r_stg->stang= cel_mai_stg;  

r_stg->legatura_stg=TRUE;  
  

/*modificare legătură extremă pe dreapta*/  

r_dr=n;  

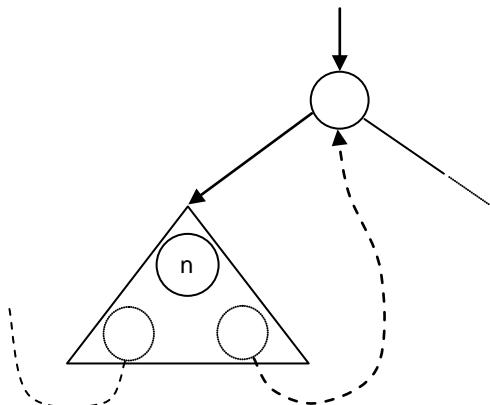
while (r_dr->drept!=NULL)  

    r_dr=r_dr->drept;  

r_dr->drept= cel_mai_dr;  

r_dr->legatura_dr=TRUE;  
  

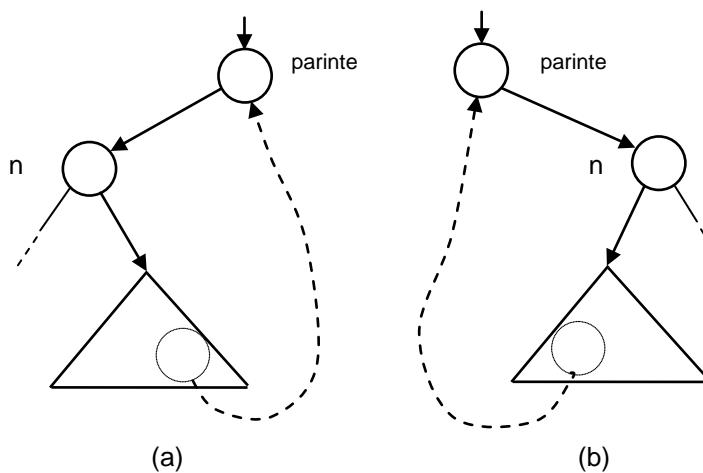
} /*InlocuiesteSub*/
-----
```



**Fig.8.2.6.1.b.** Implementarea operatorului InlocuiesteSub

- **Revenind** la problema de la care s-a pornit și anume **aflarea părintelui unui nod precizat**, arborii cu legături pot fi utilizați în acest scop.
  - Astfel considerând nodul n, pentru a afla **părintele** său se procedează după cum urmează:
  - În primul rând trebuie să se determine dacă nodul n este fiul **stâng** sau **drept** al părintelui său.
  - Presupunând că el este fiul **stâng**, atunci subarborele drept al lui n se află situat între nodul n și părintele acestuia la parcurgerea în **inordine**.
  - În consecință **cel mai din dreapta** descendent al lui n precede imediat părintele nodului n, deci urmând legătura sa pe dreapta se ajunge la nodul căutat (fig.8.2.6.1.c.(a)).

- Problema se rezolvă într-o manieră similară dacă n este **fiul dreapt** al tatălui său (fig.8.2.6.1.c.(b)).



**Fig.8.2.6.1.c.** Stabilirea părintelui unui nod

- Cu alte cuvinte:
    - Dacă  $n$  este un **fiu stâng**, atunci părintele său poate fi găsit urmând înlăncările pe **dreapta** cât de departe este posibil și parcurgând prima legătură pe dreapta.
    - Dacă  $n$  este un **fiu dreapta**, atunci părintele său poate fi găsit parcurgând înlăncările pe **stânga** cât de departe se poate și apoi urmând prima legătură pe stânga.
    - Desigur, încrucișat inițial **nu** se cunoaște dacă  $n$  este fiu stâng sau fiu drept al părintelui său, este necesară parcurgerea ambelor posibilități
    - **Varianta corectă** se determină verificând dacă nodul la care s-a ajuns are încrucișare pe  $n$  ca fiu stânga respectiv dreapta.

### 8.2.6.2. Arbori binari cu legături partiale

- După cum s-a precizat, în cazul arborilor binari este posibil să fie utilizat **numai unul** din câmpurile de legătură (cel stâng sau cel drept) funcție de sensul parcurgerii arborelui, rezultând astfel un **arbore binar cu legături partiale**.
  - În cazul parcurgerii în **inordine** este suficientă utilizarea câmpului **drept**.
  - Un astfel de arbore apare în fig.8.2.6.2.a, iar structura de date asociată lui în secvența [8.2.6.2.a]

/\*Arbore binari cu legături parțiale\*/

```
typedef tip nod legaturi * ref tip nod;
```

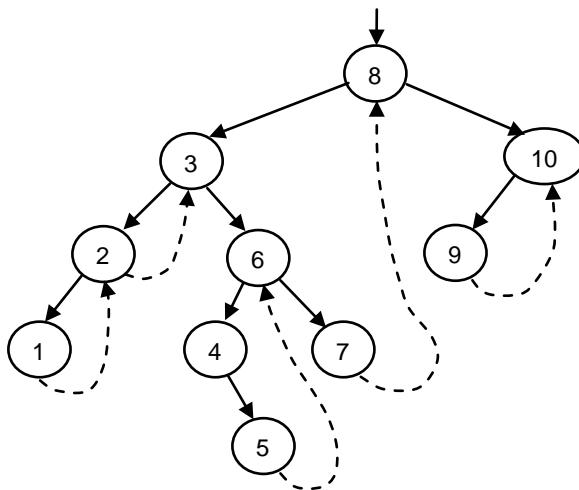
```
typedef struct tip_nod_leg_par  
{  
    tip element element;
```

```

    ref_tip_nod stang;
    ref_tip_nod drept;
    boolean legatura;
} tip_nod_leg_par;

typedef ref_tip_nod tip_arbore_binar_leg_par;
-----
```

- Câmpul legatura se referă la câmpul drept.
- Astfel dacă p este de tip ref\_tip\_nod și p->legatura= FALSE atunci p->drept indică subarborele drept al nodului p, altfel p->drept indică legătura nodului p (reprezentată punctat în fig.8.2.6.2.a).



**Fig.8.2.6.2.a.** Arbore binar cu legături parțiale

- Această reprezentare suportă **implementarea** simplă (eventual cu anumite restricții) a majorității operatorilor definiți pe **TDA arbore binar**.
- Implementarea TDA arbore binar, pornind de la aceste considerente, poate fi considerată un **exercițiu** incitant și în același timp util.

## 8.2.7 Aplicații ale arborilor binari

- Dintre aplicațiile specifice arborilor binari se prezintă câteva considerate mai reprezentative:
  - Construcția unui arbore binar de **înălțime minină**
  - Generarea unui arbore binar pornind de la specificarea sa în **preordine**
  - Construcția unui **arbore de parcursere** pornind de la forma postfix a expresiei asociate

### 8.2.7.1. Construcția și reprezentarea grafică a unui arbore binar de înălțime minimă

- Se presupune că arborele binar ce trebuie construit conține noduri încadrate în tipul clasic definit în secvența [8.2.4.2.b], în care cheile nodurilor sunt  $n$  **numere** care se **citesc** de la tastatură.
- Restricția care se impune este aceea, ca arborele construit să aibă **înălțimea minimă**.
- Pentru aceasta este necesar ca **fiecărui nivel** al structurii arborelui care se construiește să-i fie alocate **numărul maxim** de noduri posibile, cu excepția nivelului de bază.
- Acest lucru poate fi realizat prin **distribuirea** în mod egal a nodurilor care se introduc în structură pe **stânga** respectiv pe **dreapta** fiecărui nod deja introdus.
- **Regula** după care se asigură distribuția egală pentru un număr cunoscut de noduri  $n$  poate fi formulată simplu în **termeni recursivi**:

- 1º Se ia un nod drept rădăcină.
- 2º Se generează subarborele stâng cu  $ns = n \text{ DIV } 2$  noduri.
- 3º Se generează subarborele drept cu  $nd = n - ns - 1$  noduri.

- Acest algoritm permite construcția simplă a unui arbore binar de înălțime minimă.
- Un **arbore binar** este considerat **perfect echilibrat** dacă pentru fiecare nod al său, numărul de noduri al subarborelui stâng diferă de cel al subarborelui drept cu **cel mult** o unitate.
- În mod evident că **arborele de înălțime minimă** construit de către acest algoritm este **unul perfect echilibrat**.
- Implementarea algoritmului **varianta C** apare în secvența [8.2.7.1.a].
  - Cheile nodurilor se introduc de la tastatură.
  - Prima valoare introdusă este numărul total de noduri  $n$ .
  - Sarcina construcției efective a arborelui binar revine funcției **Arbore** care:
    - Primește ca parametru de intrare numărul de noduri  $n$ .
    - Determină numărul de noduri pentru cei doi subarbore.
    - Citește cheia nodului rădăcină.
    - Construiește în manieră recursivă arborele.
    - Returnează referința la arborele binar construit.

---

```
//Construcția unui arbore binar de înălțime minimă
```

```
#include "stdafx.h"
#include <stdlib.h>

typedef struct nod
{
    int cheie;
    struct nod* stang;
```

```

        struct nod* drept;
    } NOD;

int n;
NOD* radacina;

NOD* Arbore(int n)
{
    //construcție arbore perfect echilibrat cu N noduri
    NOD* NodNou;
    int x, ns, nd;

    if (n==0) //arborele vid
        return NULL;

    ns= n/2;      //determinare număr noduri subarbore stâng
    nd= n-ns-1;  //determinare număr noduri subarbore drept
    scanf_s("%d", &x); //citire cheie rădăcină

    //alocare memorie nod rădăcină arbore
    if ((NodNou = (NOD *)malloc(sizeof(NOD))) == NULL)
    {
        printf("Eroare la malloc");
        return NULL;
    }
    //completare conținut nod rădăcină
    NodNou->cheie = x;
    NodNou->stang = Arbore(ns); //completare înlănțuire
    NodNou->drept = Arbore(nd); //completare înlănțuire
    return NodNou;
} //Arbore

void Afiseaza_Arbore(NOD* t, int h)
{
    //afișează arborele t
    int i;
    if (t != NULL)
    {
        Afiseaza_Arbore(t->stang, h - 5);
        for (i = 0; i<h; i++) printf(" ");
        printf("%d\r\n", t->cheie);
        Afiseaza_Arbore(t->drept, h - 5);
    } //if
} //Afiseaza_Arbore

int main(int argc, char** argv)
{
    printf("n=");
    scanf_s("%d", &n); //numărul total de noduri
    radacina = Arbore(n); //construcție arbore
    Afiseaza_Arbore(radacina, 50);
    return 0;
} //main
-----
```

- Funcția **Afiseaza\_Arbore** realizează **reprezentarea grafică** a unei structuri de arbore binar răsturnat (rotit cu 90° în sensul acelor de ceasornic), conform următoarei **specificații**:

1º Pe fiecare rând se afișează **exact un nod**

2º Nodurile sunt ordonate de sus în jos în **inordine**

3º Dacă un nod se afișează pe un rând precedat de  $h$  blancuri, atunci fiile săi (dacă există) se afișează precedați de  $h-d$  blancuri.  $d$  este o variabilă a cărei valoare este stabilită de către programator și precizează **distanța** dintre nivelurile arborelui măsurată în caractere "spațiu"

4º Pentru **rădăcină**, se alege o valoare corespunzătoare a lui  $h$ , ținând cont că arborele se dezvoltă spre stânga.

- Această modalitate de reprezentare grafică a arborilor binari este una dintre cele mai **simple și imediate**, ea recomandându-se în mod deosebit în aplicațiile didactice.
- Trebuie subliniat faptul că simplitatea și transparența acestui program rezultă din utilizarea **recursivității**.
  - Aceasta reprezintă un **argument** în plus adus afirmației, că **algoritmii recursivi** reprezintă modalitatea cea mai potrivită de prelucrare a unor **structuri de date** care la rândul lor sunt definite **recursiv**.
  - Acest lucru este demonstrat atât de către funcția care **construiește** arborele cât și de către procedura de **afișare** a acestuia, procedură care este un exemplu clasic de parcursere în inordine a unui arbore binar.

#### 8.2.7.2. Generarea unui arbore binar pornind de la specificarea sa în preordine

- Obiectivul **exemplului** de față, îl constituie generarea unui arbore binar cu formă dirijată.
- Specificarea arborelui se face în **preordine**, enumerând nodurile (reprezentate în cazul de față printr-un singur caracter) începând cu **rădăcina** urmată mai întâi de **subarborele stâng** apoi de cel **drept**.

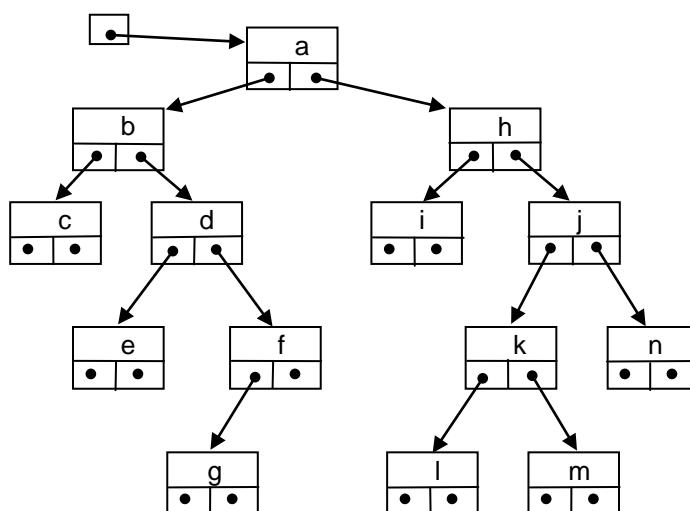


Fig.8.2.7.2.a. Structură arbore binar

- Astfel arborelui specificat în forma:

**abc..de..fg...hi..jkl..m...n..**

unde punctul semnifică **arborele vid**, îi corespunde reprezentarea din figura 8.2.7.2.a.

- Procedura care construiește o astfel de structură pornind de la specificarea arborelui în preordine precizată printr-un sir de caractere apare în [8.2.7.2.a]

- Se precizează că sunt valabile structurile de date precizate în secvența [8.2.4.2.a].

```
-----  

/*Generarea unui AB pornind de la specificarea sa in  

preordine*/  

/*Generarea unui AB pornind de la specificarea sa in  

preordine*/  

typedef struct tip_nod  

{  

    /*diferite campuri*/  

    char cheie;  

    struct tip_nod* stang;  

    struct tip_nod* drept;  

} tip_nod;  

typedef tip_nod * ref_tip_nod;  

/*Procedura de creare a AB*/  

void Creare(ref_tip_nod * p);  

{  

    char ch;  

    scanf("%c", &ch);  

    if (ch!='.')                                /*[8.2.7.2.a]*/  

    {  

        (*p)=(tip_nod *)malloc(sizeof(tip_nod)); /*completare  

                                                inlantuire*/  

        (*p)->cheie=ch;      /*asignare cheie nod curent*/  

        Creare(&(*p)->stang);   /*creare subarbore stang*/  

        Creare(&(*p)->drept);   /*creare subarbore drept*/  

    } /*if*/  

    else  

        (*p)=NULL;  

}  

/*Creare*/  

-----
```

### 8.2.7.3. Construcția unui arbore de parcurgere ("Parse Tree")

- După cum s-a precizat în paragraful 8.2.1. oricărei expresii aritmetice i se poate asocia un arbore binar numit arbore de parcurgere ("parse tree").
- În paragraful de față se pune problema de a **construi** un astfel de arbore binar asociat unei **expresii aritmetice**.

- Algoritmul precizat de secvența [8.2.7.3.a] realizează acest lucru pornind de la notația **postfix** a expresiei.

- Se presupune ca forma **postfix** a expresiei aritmetice apare ca un sir de caractere care este citit element cu element **de la dreapta la stânga** de către procedură [De89].
- 

```

subprogram ConstrArbParcurgere(ref_tip_nod * p)

/*Construcția unui arbore de parcurgere pentru o expresie
aritmetică pornind de la forma postfix a expresiei. Varianta
pseudocod. Expresia se parcurge de la dreapta la stânga*/

    tip_element t;
    citeste_element(t); /*citește elementul curent și
                           avansează la elementul următor de la dreapta la
                           stânga*/
    daca (t<>inceput)
        /*generează un nod nou indicat de p;
         *plasează pe t în noul nod;
         daca (t este un operator) [8.2.7.3.a]
            /*construcție recursivă*/
            ConstrArbParcurgere(&(*p)->drept);
            ConstrArbParcurgere(&(*p)->stang)
            □ /*daca*/
        altfel
            /*elementele non operator sunt întotdeauna
             frunze*/
            *se face subarborele drept al lui p, vid;
            *se face subarborele stang al lui p, vid
            □ /*altfel*/
        □ /*daca*/
    /*ConstrArbParcurgere*/

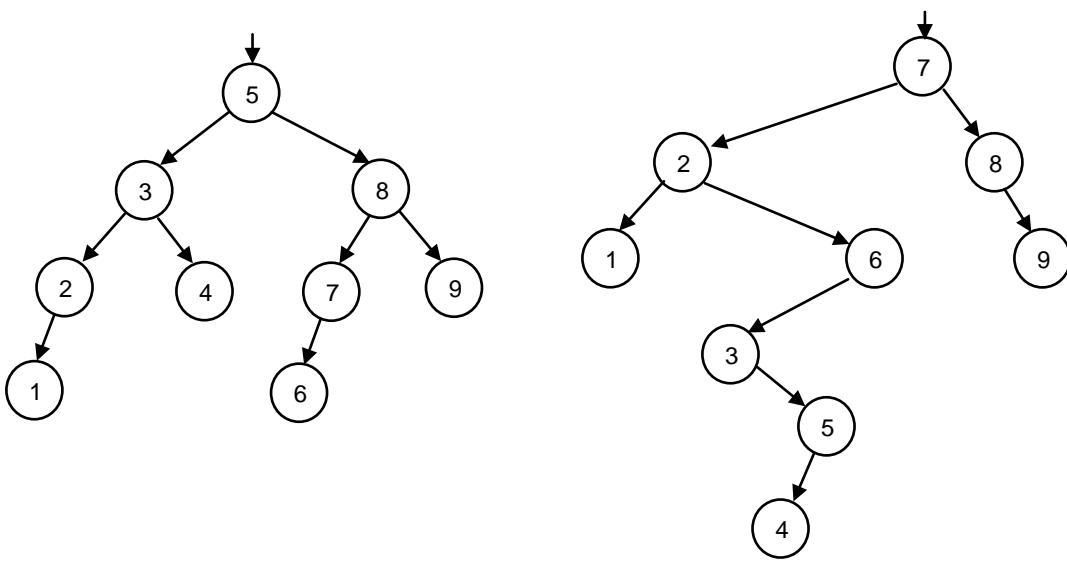
```

---

## 8.3. Arbori binari ordonați

### 8.3.1. Definiții

- Structura **arbore binar** poate fi utilizată pentru a reprezenta în mod convenabil o mulțime de elemente, în care elementele se regăsesc după o **cheie unică**.
  - Se **presupune** că avem o mulțime de  $n$  noduri definite ca articole, fiecare având câte o cheie care este număr întreg.
  - Dacă cele  $n$  articole se **organizează** într-o structură **listă liniară**, căutarea unei chei necesită în medie  $n/2$  comparații.
  - După cum se va vedea în continuare, **organizarea** celor  $n$  articole într-o **structură arbore binar convenabilă**, reduce numărul de căutări la maximum  $\log_2 n$ .
  - Acest lucru devine posibil utilizând structura **arbore binar ordonat**.
- Prin **arbore binar ordonat** se înțelege un **arbore binar** care are proprietatea că, parcurgând nodurile sale **în inordine**, secvența cheilor este **monoton crescătoare**.
- Un **arbore binar ordonat** se bucură de următoarea **proprietate**:
  - Dacă  $n$  este un **nod oarecare** al arborelui, având cheia  $c$ , **atunci**:
    - **Toate** nodurile din **subarborele stâng** a lui  $n$  au cheile mai **mici** sau egale cu  $c$ .
    - **Toate** nodurile din **subarborele drept** al lui  $n$  au chei mai **mari** sau egale cu  $c$ .
- De aici rezultă un procedeu foarte simplu de **căutare**:
  - Începând cu rădăcina, se trece la fiul **stâng** sau la fiul **drept**, după cum cheia căutată este mai **mică** sau mai **mare** decât cea a nodului curent.
- Numărul **comparațiilor de chei** efectuate în cadrul acestui procedeu este cel mult egal cu **înălțimea arborelui**.
- Din acest motiv acești arbori sunt cunoscuți și sub denumirea de **arbori binari de căutare** (“**Binary Search Trees**”).
- În general înălțimea unui arbore **nu** este determinată de **numărul** nodurilor sale.
  - Spre exemplu cu cele 9 noduri precizate în fig.8.3.1.a se poate construi atât arborele ordonat (a) de înălțime 4 cât și arborele ordonat (b) de înălțime 6.



**Fig.8.3.1.a.** Arbori binari ordonați de diferite înălțimi

- Este simplu de observat că un arbore are înălțimea **minimă** dacă **fiecare** nivel al său conține **numărul maxim de noduri**, cu excepția posibilă a ultimului nivel.
- Deoarece **numărul maxim de noduri** al nivelului  $i$  este  $2^{i-1}$ , rezultă că **înălțimea minimă** a unui arbore binar cu  $n$  noduri este:

$$h_{\min} = \lceil \log_2(n+1) \rceil$$

- Prin aceasta se justifică și afirmația că o **căutare** într-un **arbore binar ordonat** necesită aproximativ  **$\log_2 n$**  comparații de chei.
- Se precizează însă, că această afirmație este valabilă în **ipoteza** că nodurile s-au organizat într-o **structură arbore binar ordonat de înălțime minimă**.
- Dacă această condiție **nu** este satisfăcută, **eficiența** procesului de căutare poate fi mult redusă, în cazul cel mai defavorabil arborele degenerând într-o structură **listă liniară**.
- Aceasta se întâmplă când subarborele drept (sau stâng) al tuturor nodurilor este **vid**, caz în care înălțimea arborelui devine egală cu  $n$ , iar căutarea **nu** este mai eficientă decât căutarea într-o **listă liniară** ( $O(n)$ ).

### 8.3.2. Tipul de date abstract arbore binar ordonat

- Într-o manieră similară celei în care au fost definite **tipurile de date abstracte** pe parcursul acestui curs, și în cazul **arborilor binari ordonați** se poate defini un astfel de **TDA**.
- Acesta presupune desigur:
  - (1) Definirea **modelului matematic** asociat.
  - (2) Precizarea **setului de operatori**.

- Ca și pentru celelalte structuri studiate și în acest caz este greu de definit un set de operatori **general** valabil.
  - Din mulțimea seturilor posibile se propune setul prezentat în [8.3.2.a].
- 

### **TDA Arbore Binar Ordinat (ABO)**

**Modelul matematic:** este un arbore binar, fiecare nod având asociată o cheie specifică. Pentru fiecare nod al arborelui este valabilă următoarea proprietate: cheia nodului respectiv este mai mare decât cheia oricărui nod al subarborelui său stâng și mai mică decât cheia oricărui nod al subarborelui său drept.

#### **Notatii:**

*TipCheie* - tipul cheii asociate structurii nodului  
*TipElement* - tipul asociat structurii unui nod  
*RefTipNod* - referința la un nod al structurii  
*TipABO* - tipul arbore binar ordinat  
*TipABO b;*  
*TipCheie x,k;*  
*TipElement e;*  
*p: RefTipNod;*

[8.3.2.a]

#### **Operatori:**

1. **Creaza**(*TipABO b*) - procedură care crează arborele binar vid *b*;
  2. **RefTipNod Cauta**(*TipCheie x, TipABO b*) - operator funcție care caută în arborele *b* un nod având cheia identică cu *x* returnând referința la nodul în cauză respectiv indicatorul vid dacă un astfel de nod nu există;
  3. **Actualizeaza**(*TipElement e, TipABO b*) - caută nodul din arborele *b* care are aceeași cheie cu nodul *e* și îi modifică conținutul memorând pe *e* în acest nod. Dacă un astfel de nod nu există, operatorul nu realizează nici o acțiune;
  4. **Insereaza**(*TipElement e, TipABO b*) - inserează elementul *e* în arborele *b* astfel încât acesta să rămână un ABO;
  5. **SuprimaMin**(*TipABO b, TipElement e*) - extrage nodul cu cheia minimă din arborele cu rădacia *b* și îl returnează în *e*. În urma suprimării arborele *b* rămâne un ABO;
  6. **Suprima**(*TipCheie x, TipABO b*) - suprimă nodul cu cheia *x* din arborele *b*, astfel încât arborele să rămână ordinat. Dacă nu există un astfel de nod, procedura nu realizează nimic.
-

### 8.3.3. Tehnici de căutare în arbori binari ordonați

- Fie  $b$  o referință care indică rădăcina unui **arbore binar ordonat**, ale cărui noduri au structura definită în [8.3.3.a] și
- Fie  $x$  un număr întreg dat.
- În aceste condiții funcția **Caută**( $x, b$ ) precizată în secvența [8.3.3.b] execută căutarea aceluui nod aparținând arborelui  $b$  care are cheia egală cu  $x$ .
  - Căutarea se realizează în conformitate cu procedeul descris în paragraful anterior.
  - Funcția **Caută** returnează valoarea NIL dacă **nu** găsește nici un nod cu cheia  $x$ , altminteri valoarea ei este egală cu pointerul care indică acest nod.

---

```
/*Structura de date Arbore Binar Ordinat*/
```

```
typedef struct tip_nod
{
    //diferite campuri
    char cheie;
    struct tip_nod* stang;      /*[8.3.3.a]*/
    struct tip_nod* drept;
};

typedef tip_nod * ref_tip_nod;

/*Căutare în ABO (Varianta iterativă) - varianta pseudocod*/
ref_tip_nod Cauta(tip_cheie x, ref_tip_nod b)

/*caută iterativ în arborele binar b nodul cu cheia x și
returnează referința la acest nod, sau NULL daca nu îl
găsește*/

    boolean gasit=false;
    cat_timp(b<>NULL) AND (NOT gasit)           [8.3.3.b]
        daca(b->cheie=x) gasit=true;
        altfel
            daca(x<b->cheie)
                b=b->stang;
            altfel
                b=b->drept;
        □
    returneaza b;
/*cauta*/
```

---

```
/*Căutare în ABO (Varianta iterativă) - implementare C*/
```

```
ref_tip_nod Cauta(tip_cheie x, ref_tip_nod b)

{
    boolean gasit;
```

```

gasit=false;
while ((b!=NULL) && (!gasit))                                /*[8.3.3.b]*/
{
    if (b->cheie==x)
        gasit=true;
    else
        if (x<b->cheie)
            b=b->stang;
        else
            b=b->drept;
}
return b;
} {Cauta}
-----
```

- Același proces de căutare poate fi implementat și în **variantă recursivă** ținând cont de faptul că **arborele binar** este definit ca și o **structură de date recursivă**.
- **Varianta recursivă** a căutării apare în secvența [8.3.3.c].
  - Se face însă precizarea că această implementare este **mai puțin performantă** deoarece principală căutarea în arborii binari ordonați este o operație **pur sevențială** care **nu** necesită memorarea drumului parcurs.

```

/*Căutare în ABO (Varianta recursivă pseudocod)*/

ref_tip_nod CautaRecursiv(tip_cheie x, ref_tip_nod b)

/*caută recursiv în arborele binar b nodul cu cheia x și
returnează referința la acest nod, sau NULL daca nu îl
găsește*/

daca (b==NULL) returneaza b;
altfel
    daca (x<b->cheie) b=CautaRecursiv(x,b->stang);
    altfel                                                 /*8.3.3.c*/
        daca (x>b->cheie) b=CautaRecursiv(x,b->drept);
        altfel
            returneaza b;
/*CautaRecursiv*/
```

---

```

/*Căutare în ABO (Varianta recursivă)- implementare C*/
```

```

ref_tip_nod CautaRecursiv(tip_cheie x, ref_tip_nod b)

/*caută recursiv în arborele binar b nodul cu cheia x și
returnează referința la acest nod, sau NULL daca nu îl
găsește*/

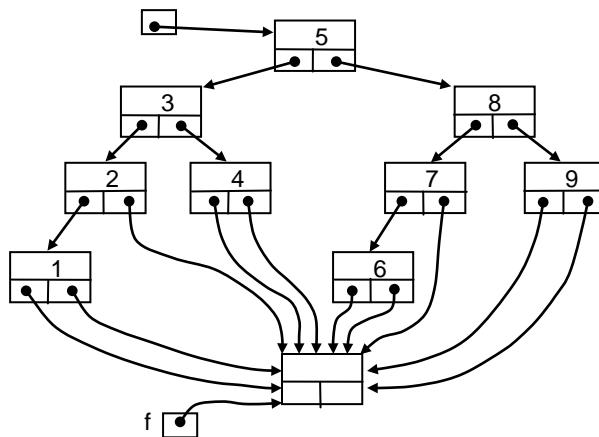
{
    if (b==NULL) THEN
        return b;
    else
        if (x<b->cheie)
            b=CautaRecursiv(x,b->stang);      /*[8.3.3.c]*/
```

```

    else
        if (x>b->cheie)
            b=CautaRecursiv(x,b->drept);
        else
            return b;
} {CautaRecursiv}
-----
```

- Tehnica de căutare poate fi simplificată dacă se aplică **metoda fanionului**.

- În cazul arborilor această metodă presupune completarea structurii cu un **nod fictiv**, nodul fanion, care este indicat de un pointer notat cu f.
- Se modifică în continuare structura arborelui, **modificând** toate referințele egale cu NIL astfel încât să-l indice pe f.
- Spre exemplu, arboarele binar ordonat din figura 8.3.1.a. stânga va avea structura din figura 8.3.3.a. Procedura de căutare propriu-zisă apare în secvența 8.3.3.d.



**Fig.8.3.3.a.** Arbore binar ordonat modificat

```

/*Căutare in ABO utilizând tehnica fanionului*/

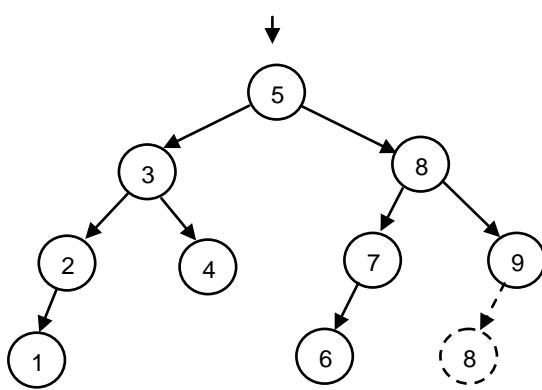
ref_tip_nod Cauta1(tip_cheie x, ref_tip_nod b)
{
    f->cheie=x; /*ref_tip_nod f este parametru global*/
    while (b->cheie!=x)
        if (x<b->cheie)
            b=b->stang; /*8.3.3.d*/
        else
            b=b->drept;
    return b;
} /*Cauta1*/
-----
```

- Înainte de demararea procesului de căutare propriu-zisă, se asignează cheia fanionului f cu x.
- În procesul căutării, nodul cu cheia x se găsește acum cu certitudine.

- Dacă acest nod este fanionul  $f$  atunci în arborele inițial **nu** există un nod cu cheia  $x$ .
- În caz contrar, nodul găsit este cel căutat.
- Se observă însă că structura din fig.8.3.3.a **nu** mai este **din punct de vedere formal un arbore**, dar ea se poate utiliza în reprezentarea unei **structuri arbore** în procesul de căutare.
- Se remarcă **simplificarea** condiției în instrucțiunea **while**, element care conferă o performanță superioară acestei metode.

#### **8.3.4. Inserția nodurilor în ABO. Crearea arborilor binari ordonați**

- În cadrul acestui paragraf se tratează:
  - (1) **Inserția nodurilor într-un arbore binar ordonat.**
  - (2) Problema **construcției unui arbore binar ordonat**, pornind de la o mulțime dată de noduri.
- Procesul de creare al unui ABO constă în **inserția** câte unui nod într-un arbore binar ordonat care inițial este vid.
  - Problema care se pune este aceea de a executa inserția de o asemenea manieră încât arborele să rămână **ordonat** și după adăugarea noului nod.
  - Aceasta se realizează **traversând** arborele începând cu rădăcina și selectând fiul **stâng** sau fiul **drept**, după cum cheia de inserat este mai **mică** sau mai **mare** decât cheia nodului parcurs.
  - Aceasta proces se **repetă** până când se ajunge la un pointer NULL.
  - În continuare inserția se realizează modificând acest pointer astfel încât să indice noul nod.
- Se precizează că inserția noului nod **trebuie** realizată chiar dacă arborele conține deja un nod cu cheia egală cu cea nouă.
  - În acest caz, dacă se ajunge la un nod cu cheia egală cu cea de inserat, se procedează ca și cum aceasta din urmă ar fi **mai mare**, deci se trece la fiul **drept** al nodului curent.
  - În felul acesta la parcurgerea în **inordine** a arborelui binar ordonat se obține o metodă de **sortare stabilă** (Vol.1 &3.1).
- În fig.8.3.4.a se prezintă inserția unei noi chei cu numărul 8 în structura existentă de arbore ordonat.
  - La parcurgerea în inordine a acestui arbore, se observă că cele două chei egale sunt parcurse în ordinea în care au fost inserate.



**Fig.8.3.4.a.** Inserția unui nod nou cu o cheie existentă

- În continuare se prezintă o **procedură recursivă** pentru inserția unui nod într-un **arbore binar ordonat**, astfel încât acesta să rămână ordonat.
- Se precizează că inițial, arborele poate fi vid.
- Structura arbore la care se vor face referiri este cea precizată în secvența [8.3.3.a].
- Procedura **Insereaza** realizează inserția unui nod cu cheia  $x$  într-un arbore binar ordonat [8.3.4.a].
  - Se precizează că  $x$  este un număr întreg reprezentând cheia nodului de inserat și  $b$  un pointer care indică rădăcina arborelui

---

```

/*Inserția unui nod într-un arbore binar ordonat (Varianta pseudocod)*/

void insereaza(tip_cheie x, ref_tip_nod b)

/*inserează nodul x în arborele binar ordonat cu radăcina b*/
daca(b!=NULL)
    daca(x<b->cheie) /*parcuregere arbore binar*/
        insereaza(x,b->stang);
    altfel
        insereaza(x,b->drept);
    altfel /*b este NULL s-a găsit locul de inserție*/
        /*insertie nod nou*/
        b=aloca_memorie(tip_nod); /*alocare nod nou în b*/
        b->cheie=x; b->stang=NULL; b->drept=NULL;
    □
/*insereaza1*/

```

---

- În secvența [8.3.4.b] se prezintă un fragment de **program principal** care utilizează procedura de mai sus în vederea creării unui arbore binar ordonat.
  - Se presupune că:
    - Toate cheile sunt diferite de zero.
    - Cheile se citesc de la dispozitivul de intrare.

- Secvența de chei se încheie cu o cheie fictivă egală cu zero pe post de terminator.

---

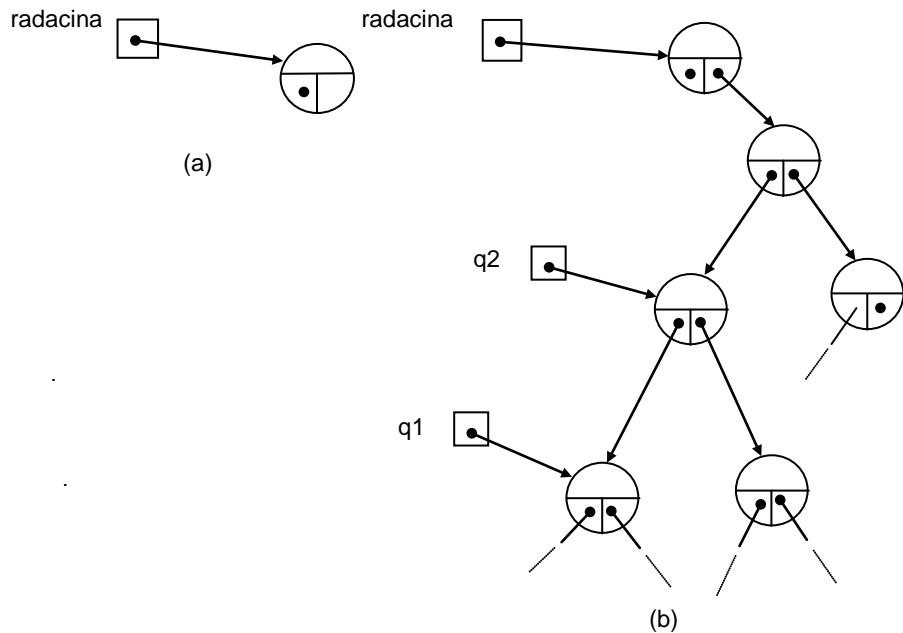
```
/*Construcția unui arbore binar ordonat - varianta
pseudocod*/
```

```
ref_tip_nod radacina;
tip_cheie c;

radacina=NULL;
citereste(c);                                /*8.3.4.b*/
cat_timp (c!=0)
|   insereaza(c,radacina);
|   citereste(c);
|   □
```

---

- Desigur, în crearea arborilor binari ordonați se poate utiliza **metoda fanionului** prezentată în paragraful anterior, caz în care trebuie realizate unele **modificări** în codul procedurii **Inserează**.
- În continuare se descrie o **variantă nerecursivă** a procedurii **Inserează**.
  - În cadrul acestei variante se disting două părți și anume:
    - (1) **Parcurgerea** arborelui pentru găsirea locului unde trebuie inserat noul nod.
    - (2) **Insetția** propriu-zisă.
  - Prima parte se implementează cu ajutorul a **două pointeri**  $q_1$  și  $q_2$ , urmând un algoritm similar celui utilizat la liste (tehnica celor doi pointeri - Vol.1 &6.4.2).
    - Cei doi pointeri indică mereu două noduri "**consecutive**" ale arborelui:
      - $q_2^{\wedge}$  este nodul curent (initial rădăcina).
      - $q_1^{\wedge}$  este fiul sau stâng sau fiul drept, după cum  $x$ , cheia care se caută, este mai mică respectiv mai mare decât cheia nodului curent indicat de  $q_2$ .
    - Pointerii avansează din nod în nod de-a lungul arborelui, până când pointerul  $q_1$  devine NULL, moment în care se realizează inserția propriu-zisă.
    - Se precizează că este nevoie și de o variabilă întreagă  $d$ , pentru a preciza dacă nodul nou trebuie inserat ca fiu stâng sau ca fiu drept al lui  $q_2^{\wedge}$ .
      - Această variabilă se asignează în timpul parcurgerii arborelui și se testează în cadrul inserției propriu-zise [Wi76].
  - Spre deosebire de **varianta recursivă** în care traseul parcurs este **memorat implicit** de către mecanismul de implementare al recursivității cu ajutorul unei **stive**, în acest caz **nu** este nevoie de stivă întrucât **nu** trebuie să se revină în arbore decât cu un singur nivel (pentru a realiza înlățuirea), motiv pentru care sunt suficienți **două pointeri consecutivi** (fig.8.3.4.b (b)).



**Fig.8.3.4.b.** Arbori binari ordonați. Tehnica celor doi pointeri

- Procedura care realizează inserția unui nod într-un **arbore binar ordonat** în manieră nerecursivă apare în secvența [8.3.4.c].

---

**/\*Insertția în ABO (Implementare nerecursivă) – varianta pseudocod\*/**

```

subprogram InsereazaNerecursiv(tip_cheie x, ref_tip_nod b);

/*inserează un nod cu cheia x în arborele binar ordonat b –
varianta iterativă*/

ref_tip_nod q1, q2;
int d;

q2=b;
q1=(*q2).drept;           /*q1=q2->drept;*/
d=1;
/*parcursere arbore binar*/
cat_timp(q1!=NULL)
  q2=q1;
  daca(x<(*q1).cheie)
    |   q1=(*q1).stang;
    |   d=-1;
    |
  altfel
    |   q1=(*q1).drept;          /*8.3.4.c*/
    |   d=1;
    |
  □ /*terminare parcursere arbore binar*/
/*inserție nod nou pe poziția lui q1*/
q1=aloca_memorie(tip_nod); /*creare nod nou*/
(*q1).cheie=x; (*q1).stang=NULL; (*q1).drept=NULL;

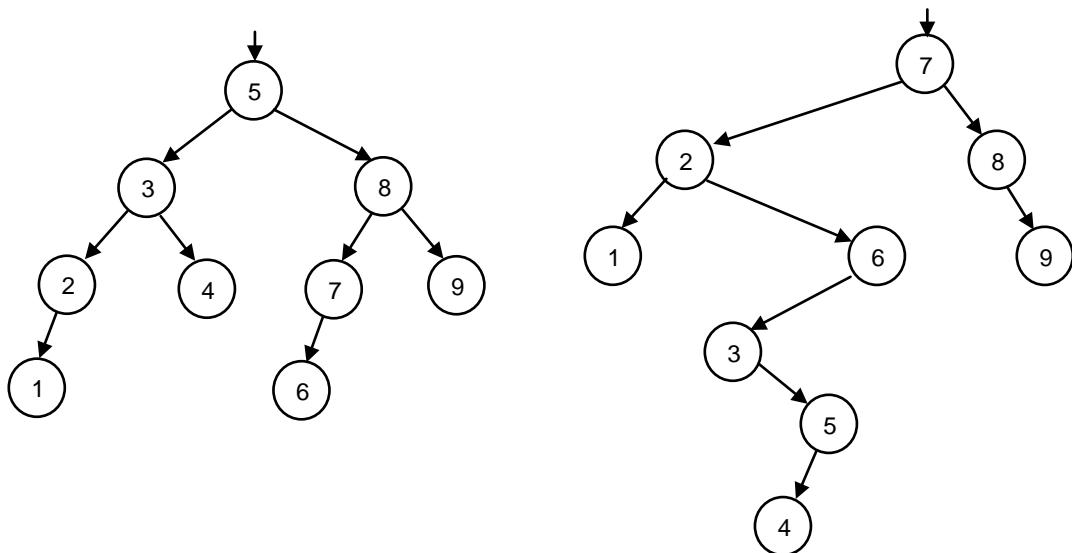
/*completare înlățuire părinte*/
daca (d<0)           /*daca (x<q2^cheie) ...*/

```

```

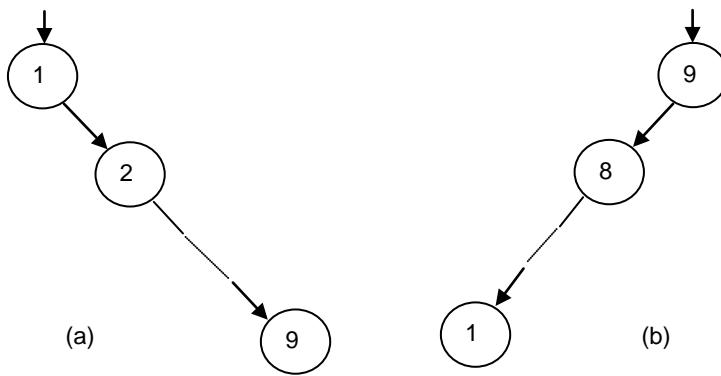
(*q2).stang=q1; /*este fiu stâng*/
altfel
    (*q2).drept=q1; /*este fiu drept*/
/*InsereazaNerecursiv*/
-----
```

- Este ușor de văzut că această procedură funcționează corect **numai** dacă arborele are **cel puțin un nod**.
- Din acest motiv în implementarea structurii arborelui se utilizează **tehnica nodului fictiv**.
- Astfel inițial arborele va conține un **nod fictiv** a cărui înlățuire pe dreapta indică primul **nod efectiv** al arborelui.
  - În această accepțiune **arborele binar vid** arată ca și în figura 8.3.4.b.(a).
  - Drept consecință cei doi pointeri vor putea fi poziționați în mod corespunzător chiar și pentru arborele vid: q2 indică nodul fictiv iar q1 este NULL.
- De asemenea se face precizarea că se poate renunța la variabila d.
  - Faptul că noul nod trebuie inserat ca fiu stâng sau ca fiu drept al lui q2 se poate stabili comparând cheia lui q2 cu cheia de inserat x.
  - Acest procedeu este sugerat ca și comentariu în secvența [8.3.4.c.]
- Cu privire la crearea arborilor binari ordonați se poate menționa faptul că **înălțimea** arborilor obținuți prin procedurile prezentate, depinde de **ordinea** în care se furnizează inițial cheile.
  - Astfel, dacă spre exemplu secvența cheilor inițiale este 5, 3, 8, 2, 4, 7, 9, 1, 6, atunci se obține arborele din figura 8.3.1.a stânga, având o înălțime minimă.
  - Dacă aceleași chei se furnizează în ordinea 7, 2, 8, 1, 6, 9, 3, 5, 4 atunci rezultă arborele mai puțin avantajos din aceeași figura dreapta.



**Fig.8.3.1.a.** Arboare binari ordonați (reluare)

- În cazul cel mai **defavorabil**, arborele poate degenera în **listă liniară**, lucru care se întâmplă în cazul în care cheile sunt furnizate în vederea inserției în **secvență ordonată crescător** respectiv **descrescător** (fig.8.3.4.c.(a),(b)).

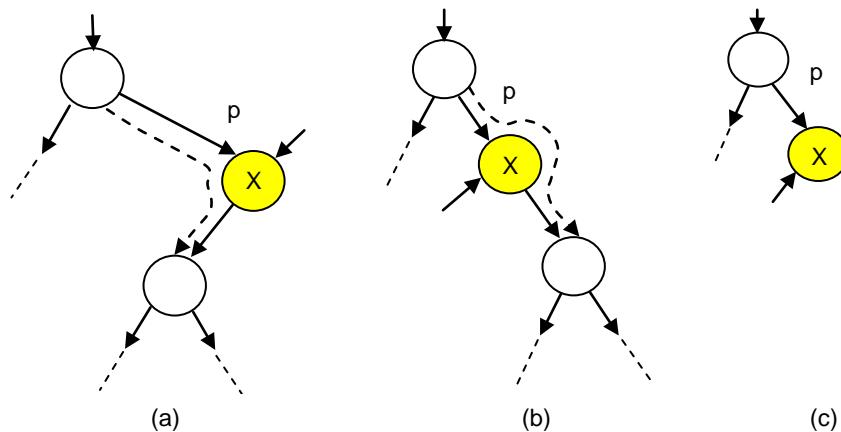


**Fig.8.4.3.c.** Arbori binari ordonați degenerați în liste liniare.

- Este evident faptul că în astfel de situații performanța căutării scade catastrofal fiind practic egală cu cea a căutării într-o listă **liniară ordonată**.
- Din fericire, probabilitatea ca să apară astfel de situații este destul de redusă, fenomen ce va fi analizat mai târziu în cadrul acestui capitol.

### 8.3.5. Suprimarea nodurilor în arbori binari ordonați

- Se consideră o structură **arbore binar ordonat** și o cheie precizată  $x$ .
- Se cere să se **suprime** din structura arbore nodul având cheia  $x$ .
  - Pentru aceasta, în prealabil se **caută** dacă există un nod cu o astfel de cheie.
  - Dacă **nu**, suprimarea s-a încheiat și se emite eventual un mesaj de eroare.
  - În caz contrar se execută suprimarea propriu-zisă, de o asemenea manieră încât arborele să rămână **ordonat** și după terminarea ei.
- Se disting două cazuri, după cum nodul care trebuie suprimit are:
  - (1) **Cel mult un fiu**.
  - (2) **Doi fii**.
- (1) **Primul caz** se rezolvă conform figurii 8.3.5 (a,b,c) în care se prezintă cele trei variante posibile.



### Fig.8.3.5.a. Suprimarea unui nod într-un ABO. Cazul 1.

- **Regula generală** care se deduce în acest caz este următoarea:
  - Fie  $p$  câmpul referintă aparținând **tatălui** nodului  $x$ , referință care indică nodul  $x$ .
  - Valoarea lui  $p$  se **modifică** astfel încât acesta să indice **unicul fiu** al lui  $x$  (dacă un astfel de fiu există - fig. 8.3.5.a (a),(b)) sau în caz contrar  $p$  devine NULL (fig.8.3.5.a (c)).
- Fragmentul de cod care apare în continuare ilustrează acest procedeu [8.3.5.a].

```
-----  
/*Suprimarea unui nod într-un ABO. Cazul 1: nodul de suprimit  
are un singur sau niciun fiu - varianta pseudocod*/
```

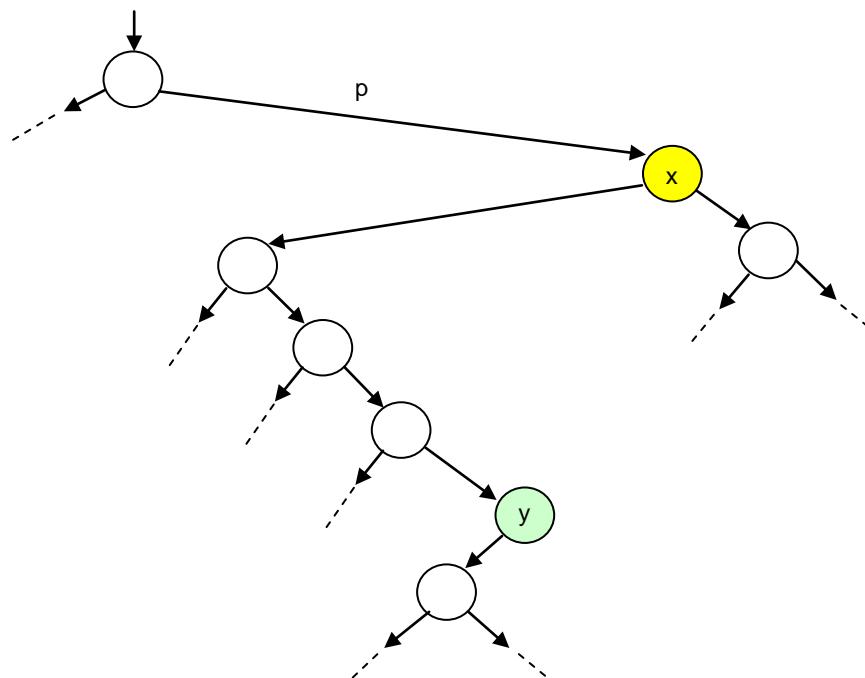
```
q=p; /*p indică nodul de suprimit*/  
daca(q->drept=NULL)  
    p=q->stang;  
    altfel  
        daca(q->stang=NULL)  
            p=q->drept;  
-----
```

- Ca **exemplu**, se prezintă în continuare implementarea operatorului **SuprimaMin** care suprimă și în același timp returnează **cel mai mic element** al unui arbore binar ordonat (secvența [8.3.5.b]).
  - **Cel mai mic** element al unui arbore binar ordonat, este **cel mai din stânga nod** al arborelui, nod la care se ajunge înaintând mereu spre stânga pornind de la rădacină.
  - Primul nod care **nu** are înlățuire spre stanga (**nu** are fiu stâng) este nodul căutat.
  - Suprimarea lui este imediată în baza procedeului precizat mai sus.

```
-----  
/*Operatorul SuprimaMin în ABO - varianta pseudocod*/
```

```
ref_tip_nod SuprimaMin(ref_tip_nod b)  
  
    ref_tip_nod temp;  
  
    daca(b!=NULL)  
        daca(b->stang!=NULL) [8.3.5.b]  
            temp=SuprimaMin(b->stang);  
            altfel /*b->stang este null*/  
                temp=b;  
                b=b->drept; /*suprimare*/  
                return temp;  
                □  
/*SuprimaMin*/  
-----
```

- Într-o manieră similară se poate implementa operatorul **SuprimaMax**.
  - Aceasta realizează suprimarea celui mai mare nod al arborelui, care este evident nodul situat cel mai la **dreapta** în arbore.
- (2) **Cel de-al doilea caz**, în care nodul cu cheia  $x$  are doi fii se rezolvă astfel:
  - (1) Se caută **predecesorul** nodului de suprimit  $x$  în ordonarea în **inordine** a arborelui.
    - Fie acesta  $y$ . Se demonstrează că nodul  $y$  există și că el **nu** are fiu drept.
  - (2) Se modifică nodul  $x$ , asignând toate câmpurile sale, cu excepția câmpurilor **stâng** și **drept** cu câmpurile corespunzătoare ale lui  $y$ .
    - În acest moment în structura arbore, nodul  $y$  se găsește în dublu exemplar: în locul său inițial și în locul fostului nod  $x$ .
  - (3) Se suprimă nodul  $y$  inițial, conform fragmentului [8.3.5.a] deoarece nodul nu are fiu drept.
- Cu privire la nodul  $y$ , se poate demonstra că el se detectează după următoarea **metodă**:
  - Se construiește o secvență de noduri care începe cu fiul **stâng** al lui  $x$ , după care se alege drept succesor al fiecărui nod, fiul său **drept**.
  - Primul nod al secvenței care **nu** are fiu drept este  $y$  (fig.8.3.5.b).
    - Este de fapt **cel mai mare nod** al subarborelui stâng al arborelui binar care are rădăcina  $x$ .



**Fig. 8.3.5.b.** Suprimarea unui nod într-un ABO. Cazul 2.

- Procedura care realizează **suprimarea** unui nod într-o **structură arbore binarordonat** apare în secvența [8.3.5.c].

- Procedura locală **SuprimaPred**, caută **predecesorul în inordine** al nodului x, realizând suprimarea acestuia conform metodei descrise (are cel mult un fiu).
- După cum se observă, procedura **SuprimaPred** se utilizează numai în situația în care nodul x are doi fii.

---

**/\*Suprimarea unui nod într-un ABO. Cazul 2: nodul de suprimat are doi fii - varianta pseudocod\*/**

```

ref_tip_nod q; /*global*/

void SuprimaPred(ref_tip_nod r)
    /*caută și suprimă predecesorul nodului indicat de r*/
    daca(r->drept!=NULL)
        SuprimaPred(r->drept);
    altfel /*
        | q->cheie=r->cheie; /*mută conținutul lui r în q*/
        | q=r; /*salvare adresa nod r*/
        | r=r->stang; /*suprimă nodul r*/
        |
    /*SuprimaPred*/

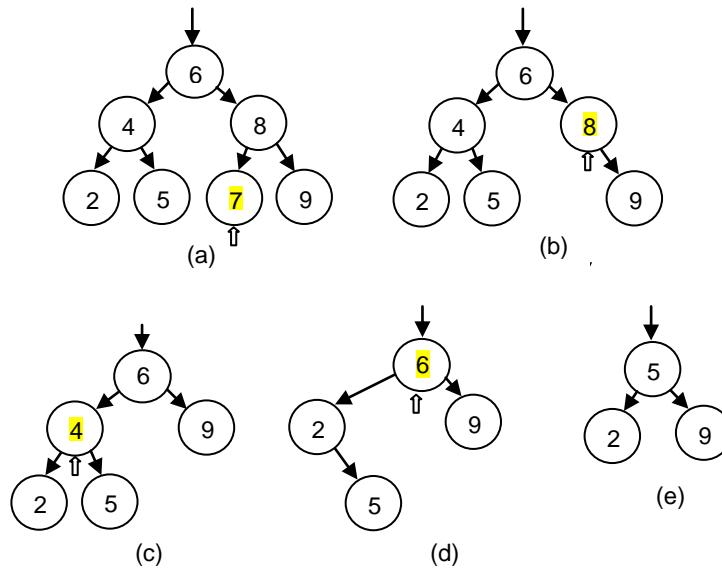
void Suprima(tip_cheie x, ref_tip_nod b)
    /*caută și suprimă nodul cu cheia x din arborele binar
    ordonat b*/
    daca(b=NULL) /*nodul nu a fost gasit*/
        afiseaza('nodul nu se gasescete'); [8.3.5.c]
    altfel
        daca(x<b->cheie) /*cautare cheie de suprimat*/
            Suprima(x,b->stang);
        altfel
            daca(x>b->cheie)
                Suprima(x,b^.drept);
            altfel /*nodul s-a gasit*/
                q=b;
                daca(q->drept=NULL)
                    /*nodul nu are fiu drept*/
                    b=q->stang; /*suprimare*/
                altfel
                    daca q->stang=NULL
                        /*nodul nu are fiu stang*/
                        b=q^.drept; /*suprimare*/
                    altfel /*nodul are 2 fiu*/
                        SuprimaPred(q->stang);
                |
            |
    /*Suprima*/

```

---

- Procedura **SuprimaPred**:
  - Găsește pointerul r care indică nodul având cea mai mare cheie, dintre cele subarborelui stâng, al arborelui care are drept rădăcină nodul cu cheia x.

- Înlocuiește câmpurile nodului cu cheia  $x$ , indicat de pointerul  $q$ , cu câmpurile nodului indicat de  $r$  (cu excepția înlățuirilor).
- Suprimă nodul indicat de  $r$ , acesta din urmă având un singur fiu (sau niciunul).
- Pentru a ilustra comportarea acestei proceduri în fig.8.3.5.c se prezintă:
  - O structură arbore binar ordonat (a).
  - Din care se suprimă în mod succesiv nodurile având cheile 7, 8, 4, și 6 (fig.8.3.5.c (b-e)).



**Fig. 8.3.5.c.** Suprimarea nodurilor într-o structură arbore binar ordonat

- Există și o altă soluție de a rezolva suprimarea în cazul în care nodul  $x$  are doi fiu și anume:
  - (1) Se caută **succesorul** nodului  $x$  în ordonarea în inordine a cheilor arborelui. Se demonstrează că el există și ca **nu** are fiu stâng.
  - (2) Pentru suprimare se procedează analog ca și în cazul anterior, cu deosebirea că totul se realizează **simetric** (în oglindă).
  - În acest caz, de fapt se caută nodul cu cea mai mică cheie a subarborelui drept al arborelui care-l are pe  $x$  drept rădăcină.
- Pentru o mai bună înțelegere a celor prezentate se reamintește o **proprietate** a arborilor binari ordonați:
  - Proiecția pe abscisă a nodurilor unui arbore binar, conduce la ordonarea lor în **inordine**.
  - În cazul **arborilor binari ordonați** se obține de fapt **secvența ordonată crescător** a cheilor arborelui.

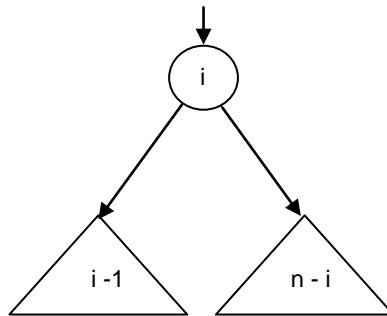
### 8.3.6. Analiza căutării în arbori binari ordonați

- În general, în activitatea de programare se manifestă o anumită suspiciune față de căutarea și inserția nodurilor într-o structură **arbore binar ordonat**.
- Această suspiciune este motivată de faptul că programatorul în general **nu** are controlul creșterii arborelui și ca atare **nu** poate anticipa cu suficientă precizie forma acestuia.
  - După cum s-a precizat, efortul de căutare al unei chei variază între  $O(\log_2 n)$  pentru **arborele binar perfect echilibrat** (de înălțime minimă) și  $O(n)$  pentru **arborele binar ordonat degenerat într-o listă liniară**.
  - Cele două situații reprezintă extremele situațiilor reale iar probabilitatea ca ele să apară este în general redusă [Wi76].
- **Cazul general** care va fi analizat în continuare, este următorul:
  - Se consideră  $n$  chei.
  - Cele  $n$  chei pot fi permutate în  $n!$  moduri.
  - Întrucât forma unui **arbore binar ordonat** depinde de ordinea în care sunt inserate cheile în arbore, **fiecare permutare** conduce la un arbore binar distinct.
  - Se cere să se determine **lungimea medie  $a_n$  a drumului de căutare**, corespunzător tuturor celor  $n!$  arbori binari cu  $n$  noduri, care pot fi generați pornind de la cele  $n!$  permutări ale celor  $n$  chei originale.
  - Se consideră că cele  $n$  chei sunt **dinamice** având valorile  $1, 2, \dots, n$  și se presupune că sosesc în ordine aleatoare cu o **distribuție normală** a **probabilității** de apariție.
  - În acest context **lungimea medie a drumului de căutare** într-un **arbore binar** cu  $n$  noduri, notată  $a_n$ , se definește ca fiind o **sumă** de  $n$  termeni, fiecare termen referindu-se la un nod al arborelui și fiind egal cu **produsul** dintre **nivelul nodului** respectiv (adică **lungimea drumului** la nodul respectiv) și **probabilitatea** sa de acces.
  - Dacă se presupune că **toate nodurile** sunt **în mod egal căutate** (au probabilitatea de acces  $1/n$ ), atunci formal **lungimea medie a drumului de căutare  $a_n$**  apare în [8.3.6.a] unde  $p_i$  este lungimea drumului la nodul  $i$  (nivelul nodului  $i$ ).

$$a_n = \frac{1}{n} \sum_{i=1}^n p_i$$

[8.3.6.a]

- La crearea arborelui, probabilitatea ca cheia  $i$  să devină **prima cheie** și deci implicit rădăcina arborelui este  $1/n$ .
- În consecință, subarborele stâng va conține  $i-1$  noduri iar subarborele drept  $n-i$  noduri (fig.8.3.6.a).



**Fig.8.3.6.a.** Distribuția numerică a cheilor într-un ABO

- Fie  $a_{i-1}$  lungimea medie a drumului de căutare pentru subarborele stâng, respectiv  $a_{n-i}$  lungimea medie a drumului de căutare pentru subarborele drept, presupunând din nou că toate permutările celor  $n-1$  chei rămase sunt egal posibile.
- În aceste condiții în arborele din figura 8.3.6.a, nodurile pot fi împărțite în trei clase:
  1. Cele  $i-1$  noduri ale **subarborelui stâng**, care au lungimea medie a drumului egală cu  $a_{i-1}+1$ .
  2. **Rădăcina** care are lungimea drumului 1.
  3. Cele  $n-i$  noduri ale **subarborelui drept**, cu lungimea medie a drumului egală cu  $a_{n-i}+1$ .
- Astfel lungimea medie a drumului de căutare  $a_n^{(i)}$  pentru situația în care  $i$  este cheia nodului rădăcină poate fi exprimată ca o sumă de trei termeni [8.3.6.b].

$$a_n^{(i)} = (a_{i-1} + 1) \frac{i-1}{n} + 1 * \frac{1}{n} + (a_{n-i} + 1) \frac{n-i}{n} \quad [8.3.6.b]$$

- În consecință,  $a_n$  poate fi considerat ca și **media aritmetică** a termenilor  $a_n^{(i)}$  pentru toți  $i=1, 2, \dots, n$ , respectiv pentru toți cei  $n$  arbori care au respectiv cheile  $1, 2, \dots, n$  drept **rădăcină** [8.3.6.c].

$$\begin{aligned} a_n &= \frac{1}{n} \sum_{i=1}^n a_n^{(i)} = \frac{1}{n} \sum_{i=1}^n \left[ (a_{i-1} + 1) \frac{i-1}{n} + \frac{1}{n} + (a_{n-i} + 1) \frac{n-i}{n} \right] = \\ &= 1 + \frac{1}{n^2} \sum_{i=1}^n [(i-1)a_{i-1} + (n-i)a_{n-i}] = 1 + \frac{2}{n^2} \sum_{i=1}^n (i-1)a_{i-1} = 1 + \frac{2}{n^2} \sum_{i=1}^{n-2} i * a_i \quad [8.3.6.c] \end{aligned}$$

- Ecuația [8.3.6.c] este o **relație de recurență** pentru  $a_n$  prezentată în forma  $a_n = f_1(a_1, a_2, \dots, a_{n-1})$ .

- Din această formă se poate deduce o **relație de recurență** de forma  $a_n = f_2(a_{n-1})$ .
- În acest scop, din formula de mai sus rezultă direct [8.3.6.d] respectiv [8.3.6.e].

$$a_n = 1 + \frac{2}{n^2} \sum_{i=1}^{n-1} i * a_i = 1 + \frac{2}{n^2} (n-1)a_{n-1} + \frac{2}{n^2} \sum_{i=1}^{n-2} i * a_i \quad [8.3.6.d]$$

$$a_{n-1} = 1 + \frac{2}{(n-1)^2} \sum_{i=1}^{n-2} i * a_i \quad [8.3.6.e]$$

- Înmulțind relația [8.3.6.e] cu  $((n-1)/n)^2$  se obține [8.3.6.f]:

$$\frac{2}{n^2} \sum_{i=1}^{n-2} i * a_i = \frac{(n-1)^2}{n^2} (a_{n-1} - 1) \quad [8.3.6.f]$$

- Înlocuind pe [8.3.6.f] în [8.3.6.d] rezultă forma dorită a relației de recurență [8.3.6.g].

$$a_n = \frac{1}{n^2} ((n^2 - 1)a_{n-1} + 2n - 1) \quad [8.3.6.g]$$

- Pe de altă parte,  $a_n$  poate fi exprimat într-o **formă nerecursivă** utilizând termenii **funcției armonice**  $H$  [8.3.6.h] după cum se prezintă în relația [8.3.6.i]

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \quad [8.3.6.h]$$

$$a_n = 2 \frac{n+1}{n} H_n - 3 \quad [8.3.6.i]$$

- Se poate verifica faptul că relația [8.3.6.i] verifică relația recursivă [8.3.6.g].
- Dar valoarea aproximativă a lui  $H_n$  poate fi determinată în baza formulei lui **Euler** [8.3.6.j].

$$H_n = \gamma + \ln(n) + \frac{1}{12n^2} + \dots \quad [8.3.6.j]$$

unde  $\gamma \approx 0.577$  este constanta lui Euler. Înlocuind această valoare în formula [8.3.6.i] rezultă [8.3.6.k].

$$a_n \approx 2[\ln(n) + \gamma] = 2\ln(n) - c$$

[8.3.6.k]

- Reamintim că această valoare calculată  $a_n$  reprezintă **lungimea medie a drumului de căutare** pentru **toți arborii binari ordonați** care pot fi construiți pornind de la  $n$  chei, adică  $n!$  arbori.
- Întrucât **lungimea medie a drumului de căutare** într-un **arbore binar perfect echilibrat** cu  $n$  chei este [8.3.6.l]:

$$a'_n = \log_2(n) - 1$$

[8.3.6.1]

- Dacă calculăm **la limită** valoarea raportului dintre cele două lungimi de drumuri, neglijând termenii constanți care pentru valori mari ale lui  $n$  devin neglijabili, obținem relația finală [8.3.6.m].

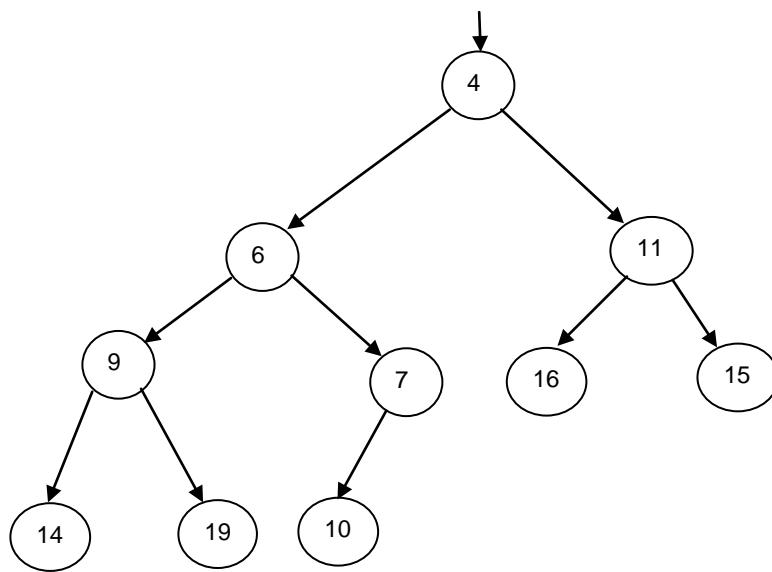
$$\lim_{n \rightarrow \infty} \frac{a_n}{a'_n} = \frac{2\ln(n)}{\log_2(n)} = \frac{2\ln(n)}{\frac{\ln(n)}{\ln 2}} = 2\ln 2 \approx 1.386$$

[8.3.6.m]

- **Concluzia** este că înlocuind **arborele binar perfect echilibrat** cu un **arbore binar ordonat oarecare**, efortul de căutare crește în medie cu 39 %.
  - Desigur, creșterea acestui efort poate fi mult mai mare, dacă arborele binar ordonat oarecare este nefavorabil, spre exemplu degenerat într-o listă, dar această situație are o probabilitate foarte mică de a se realiza.
  - Cele 39 % impun practic limita efortului adițional de calcul care poate fi cheltuit în mod profitabil pentru reorganizarea structurii după inserarea cheilor.
  - În acest sens un rol esențial îl joacă **raportul** dintre numărul de **accese la noduri** (căutări) și **numărul de inserții** realizate în arbore.
    - Cu cât acest raport este mai mare cu atât reorganizarea structurii este mai justificată.
  - În general valoarea 39 % este suficient de redusă pentru ca în majoritatea aplicațiilor să se recurgă la tehnici directe de inserare și să nu se facă uz de reorganizare decât în situații deosebite.

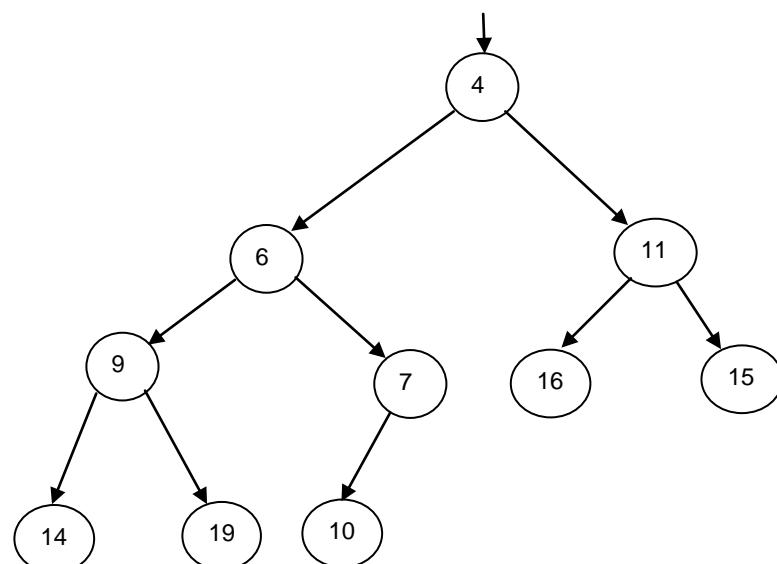
### 8.3.7. Arboi binari parțial ordonați

- O structură arbore binar aparte o reprezintă structura **arbore binar parțial ordonat**.
- Caracteristica esențială a unui astfel de arbore este aceea că cheia oricărui nod **nu** este mai mare (mică) decât cheile fililor săi.



**Fig.8.3.7.a.** Arbore binar parțial ordonat

- Un exemplu de astfel de arbore apare în figura 8.3.7.a.
- Deoarece un arbore binar parțial ordonat este de fapt un arbore binar, se poate realiza o reprezentare eficientă a sa cu ajutorul unui tablou liniar aplicând tehnica specificată la paragraful 8.2.4.1.
- Această reprezentare este cunoscută și sub numele de **ansamblu** (heap) și a fost definită în partea I (sortare prin metoda ansamblelor - Vol.1 &3.2.5.)
- Spre exemplu arborele binar parțial ordonat din figura 8.3.7.a apare reprezentat ca un ansamblu în figura 8.3.7.b.



**Fig.8.3.7.b.** Reprezentarea unui arbore binar parțial ordonat ca un ansamblu

- Structura ansamblu permite implementarea eficientă și foarte elegantă atât a unor metode de sortare (**sortarea prin metoda ansamblelor** - Vol.1 &3.2.5) cât și a unor structuri de date derivate din liste (**cozi bazate pe prioritate** - Vol.1 &6.5.5.3).
- Se reamintește de asemenea faptul că aceasta structură a fost extinsă cu un set specific de operatori într-un **tip de date abstract ansamblu** (Vol.1 &6.5.5.5).

### 8.3.8. Aplicații ale arborilor binari ordonați

#### 8.3.8.1. Problema concordanței

- În cadrul acestui paragraf se propune reluarea **problemei concordanței** prezentată în Vol.1 și rezolvarea ei cu ajutorul structurilor de date **arbore binar ordonat**.
- Se reamintește că **problema concordanței** constă de fapt în **determinarea frecvențelor de apariție** ale cuvintelor unui text.
- **Problema** se formulează astfel:
  - Se consideră un text format dintr-o succesiune de cuvinte.
  - Se parurge textul și se delimitizează cuvintele.
  - Pentru fiecare cuvânt se verifică dacă **este** sau **nu la prima apariție**.
  - În caz **afirmativ**, cuvântul se înregistrează și contorul asociat se inițializează pe valoarea 1.
  - În caz **negativ** se incrementează contorul asociat cuvântului, contor care memorează numărul de apariții.
  - În final se dispune de **lista** (ordonată) a **tuturor cuvintelor** și de **numărul de apariții** ale fiecărui.
- În acest scop, nodurile reprezentând cuvintele sunt organizate într-o **structură arbore binar ordonat**, pornind de la un arbore vid.
- Procesul de creare a structurii arbore binar ordonat se desfășoară după cum urmează:
  - Se citește un nou cuvânt și se caută în arbore.
    - Dacă **nu** se găsește atunci cuvântul se inserează.
    - Dacă cuvântul se găsește, atunci se incrementează contorul de apariții al cuvântului respectiv.
  - Procesul continuă până la epuizarea tuturor cuvintelor textului analizat.
- Se presupune că un nod al structurii **arbore binar ordonat** are structura precizată în secvența [8.3.8.1.a].

---

**/\*Problema concordanței. Implementare bazată pe arbori binari ordonați structuri de date\*/**

```
typedef struct cuvant {
    int cheie;                                     /*[8.3.8.1.a]*/
    int contor;
    struct cuvant * stang;
    struct cuvant * drept;
```

```
 } tip_nod;
```

- Fie radacina o variabilă pointer care indică rădăcina arborelui binar ordonat.
- Programul care rezolvă problema concordanței apare în secvența [8.3.8.1.b].

---

/\*PROGRAM Concordanta - varianta C\*/

```
#include "stdafx.h"
#include <stdlib.h>

/*definire structura nod arbore*/
typedef struct cuvant {
    int cheie;
    int contor;
    struct cuvant * stang;
    struct cuvant * drept;
} tip_nod;

typedef struct cuvant * ref_tip_nod;

ref_tip_nod radacina; /*arborele cuvintelor*/
int cuv;

void Imprarbore(ref_tip_nod r)
/*afișază arborile cu radacina r parcurgîndu-l în inordine*/
{
    if (r != NULL)
    {
        Imprarbore(r->stang);
        printf("cheia %d, are frecventa %d \n", r->cheie,
               r->contor);
        Imprarbore(r->drept);
    }
} /*Imprarbore*/

void Cauta(int x, ref_tip_nod * p)
/*caută cuvântul x în arborele p; dacă nu îl găsește îl inserează; dacă
îl găsește îi incrementează contorul*/
{
    if ((*p) == NULL) /*cuvântul nu există, deci inserție*/
    {
        (*p) = (cuvant*)malloc(sizeof(struct cuvant));
        (*p)->cheie = x; (*p)->contor = 1;
        (*p)->stang = NULL; (*p)->drept = NULL;
    }
    else
        if (x<(*p)->cheie)
            Cauta(x, &(*p)->stang);
        else
            if (x>(*p)->cheie)
                Cauta(x, &(*p)->drept);
            else /*cuvânt găsit, incrementare contor*/
                (*p)->contor = (*p)->contor + 1;
} /*Cauta*/

int main()
{
    ref_tip_nod radacina;
    int cuv;
    radacina = NULL;
```

```

printf("cheie = ");
scanf_s("%d", &cuv);
while (cuv != 0)
{
    Cauta(cuv, &radacina);
    printf("cheie = ");
    scanf_s("%d", &cuv);
}
Imprarbore(radacina);
}

```

---

- Pentru simplificare se presupune ca **textul** analizat constă dintr-o succesiune de **numere întregi** care modelează cuvintele textului, iar cifra 0 este utilizată ca **terminator**.
- Textul se introduce prin furnizarea numerelor de la tastatură.
- Procedura **Cauta** realizează următoarele:
  - (1) Caută cheia cuv în arborele indicat de pointerul **radacina**.
  - (2) Dacă **nu** o găsește, inserează cheia în arbore.
  - (3) Dacă găsește cheia incrementează contorul corespunzător.
- După cum se observă, această procedură este o **combinație** a căutării și creeării arborilor binari ordonați.
- Metoda de **parcursere** a arborelui este cea prezentată la căutarea în arbori binari ordonați varianta recursivă (&8.3.3)
- Dacă **nu** se găsește nici un nod cu cheia cuv atunci are loc inserția similară celei utilizate în cadrul procedurii **Insereaza** definită la inserția în arbori binari ordonați varianta 1 (&8.3.4).
- Procedura recursivă **Imprarbore** parcurge nodurile arborelui în **inordine** afișându-le unele sub altele, fără a reflecta însă și structura arborelui, element care diferențiază această procedură de cea prezentată în secvența [8.2.7.1.a].
- În continuare se prezintă o **a doua variantă**, de data aceasta **nerecursivă** a procedurii **Cauta** bazată pe **varianta nerecursivă** a procedurii de inserție ([8.3.4.c]).
  - Parcurgerea arborelui se face cu ajutorul a doi pointeri q1 și q2 după cum s-a prezentat în paragraful 8.3.4.
  - Codul aferent acestei proceduri apare în [8.3.8.1.c] în forma procedurii **CautaNerecursiv**.

---

**/\*Problema concordanței. Implementare bazată pe arbori binari ordonați - varianta iterativă pseudocod\*/**

```

subprogram CautaNerecursiv(int x, ref_tip_nod radacina)

/*inserează un nod cu cheia x în arborele binar ordonat b -
varianta iterativă*/

ref_tip_nod q1,q2;
int d;

q2=b;
q1=(*q2).drept;           /*q1=q2->drept;*/

```

```

d=1;
cat_timp((q1!=NULL)&&(d!=0)) /*căutare în arbore binar*/
    q2=q1;
    daca(x<(*q1).cheie)
        q1=(*q1).stang;
        d=-1; /*q1 este fiu stâng*/
    □
    altfel
        daca(x>(*q1).cheie)
            q1=(*q1).drept; [8.3.4.c]
            d=1; /*q1 este fiu drept*/
        □
        altfel
            d=0; /*cheia s-a găsit în arbore*/
    □ /*terminare căutare*/

daca(d=0) /*cheia s-a găsit în arbore*/
    (*q1).numar=(*q1).numar+1 /*incrementare contor*/
    altfel /* inserție nod nou pe poziția lui q1*/
        q1= aloca_memorie(tip_nod); /*creare nod nou*/
        (*q1).cheie=x; (*q1).numar=1;
        (*q1).stang=NULL; (*q1).drept=NULL;
        /*completare înlățuire părinte*/
        daca d<0           /*daca (x<q2^cheie) ...*/
            (*q2).stang=q1; /*q1 este fiu stâng*/
        altfel
            (*q2).drept=q1; /*q1 este fiu drept*/
        □
/*CautaNerecursiv*/

```

- Condiția necesară ca această procedură să lucreze corect este ca arborele să **aibă cel puțin** un nod, motiv pentru care în implementarea structurii arborelui s-a utilizat tehnica **nodului fictiv**.
- De asemenea, pentru funcționarea corectă a programului **Concordanta** cu procedura **CautaNerecursiv**, initializarea **radacina=NULL** din programul principal indicată cu [\*] în secvența [8.3.8.1.b], se înlocuiește cu secvența [8.3.8.1.d].

```

radaciana= (cuvant*)malloc(sizeof(struct cuvant));
radacina->drept=NULL; [8.3.8.1.d]

```

- În consecință **radacina** indică **un nod fictiv** în cadrul căruia se asignează numai câmpul **drept**.
- Rădăcina efectivă a arborelui apare ca fiu drept al acestui nod. În mod evident trebuie reformulat apelul procedurii de căutare din programul principal.

### 8.3.8.2. Generator de referințe încrucișate

- În cadrul acestui paragraf se prezintă un program pentru construirea unui **index de referințe încrucișate** ("cross-reference index") referitor la un text dat.
- Specificația programului este următoarea:

- Programul citește un **text** din fișierul de intrare.
- Selectionează și memorează **toate cuvintele** textului precum și **numerele rândurilor** în care acestea au fost întâlnite.
- La terminarea parcurgerii textului, programul furnizează o **listă** conținând **toate cuvintele ordonate alfabetic**.
- Pentru fiecare cuvânt în parte, se furnizează numerele liniilor de text în care el apare (**tabela de referințe încrucișate**).
- Metoda utilizată este aceea de a construi un **arbore binar ordonat** în care cheia (identificatorul) fiecărui nod este un cuvânt al textului.
- Un astfel de arbore care este ordonat în ordinea alfabetica a cuvintelor se mai numește și **arbore lexicografic**.
- În afara cheii, fiecare nod al arborelui conține un **pointer** la o **listă liniară** care păstrează numerele rândurilor în care a fost întâlnit respectivul identificator.
- Pentru a accelera procesul de inserție al unui nod nou în această listă, conform celor precizate la studiul listelor înlántuite, (Vol.1 &6.3.2.1), este indicat a se păstra o **referință la sfârșitul listei**, referință care apare tot ca și un câmp distinct al nodului arborelui.
- Se observă astfel că în cadrul acestui exemplu se utilizează structuri de date combinate: arbori și liste. Avem de fapt de-a face cu **un arbore de liste**.
- Programul complet varianta C apare în secvența [8.3.8.2.a] și el constă din două faze:
  - (1) **Faza de parcurgere a textului**, de identificare a cuvintelor și de creare a **arborelui lexicografic**.
  - (2) **Faza de afișare** a tabelei de referințe încrucișate.
- Pentru urmărirea mai ușoară a programului se fac următoarele precizări:
- Prin **cuvânt** al textului se înțelege orice secvență de litere sau cifre care începe cu o literă. Cuvântul se termină la întâlnirea primului caracter diferit de literă sau cifră.
- Funcția **cauta** are rolul de a căuta cuvântul curent în structura **arbore binar ordonat**.
  - Dacă cuvântul **nu** este găsit el se inserează în arbore.
    - Se precizează că la inserția unui nod nou în arbore, se crează și primul nod al listei liniare atașate.
    - Dacă cuvântul este găsit, se adaugă un nou nod listei liniare atașate conținând numărul liniei curente.
    - Funcția returnează un pointer la nodul găsit respectiv nou inserat.
- Funcția **listearaz\_cuvant** realizează afișarea cuvântului asociat unui nod al arborelui lexicografic, urmat de afișarea listei care memorează liniile în care el apare.
- Funcția **parurge\_arbore** realizează **traversarea în inordine a arborelui** și afișează nodurile întâlnite obținându-se astfel **tabelul ordonat de referințe încrucișate**. Ea face uz de funcția **listearaz\_cuvant**.
- Programul principal este implementat de funcția **parurge\_fisier**.
  - Această funcție parurge textul sursă, delimiteză cuvintele și pentru fiecare cuvânt apeleză funcția **caută** care are rolul de a completa, respectiv de a modifica structura arborelui indicat de variabila **radacina**.

- La terminarea parcurgerii textului se afișează **tabela de referințe încrucișate** (funcția **parcurge\_arbore**) după care se dezafectează spațiul de memorie alocat prin distrugerea arborelui lexicografic.

```
-----  

/* Generator de referințe încrucișate*/  

#include <stdio.h>  

#include <malloc.h>  

#include <string.h>  

#define LMAXCUVANT 64  

typedef struct rand  

{  

    int nr;  

    rand *urm;  

} rand;  

typedef struct cuvant  

{  

    char cheie[LMAXCUVANT];  

    rand *prim, *ultim;  

    cuvant *stang, *drept;  

} cuvant;  

cuvant* cauta(cuvant *pc,char *cheie,int nr_linie)  

{  

    if(pc==NULL) /*creare nod terminal în arborele binar*/  

    {  

        rand *pr2;  

        pr2=(rand*)malloc(sizeof(rand));  

        pr2->nr=nr_linie;  

        pr2->urm=NULL;  

        cuvant *pc2;  

        pc2=(cuvant*)malloc(sizeof(cuvant));  

        strcpy(pc2->cheie,cheie);  

        pc2->prim=pc2->ultim=pr2;  

        pc2->stang=pc2->drept=NULL;  

        return pc2;  

    }  

    else  

    {  

        if(strcmp(pc->cheie,cheie)>0) /* căutare cuvânt în  

                                              subarborele stâng */  

        {  

            pc->stang=cauta(pc->stang,cheie,nr_linie);  

            return pc;  

        }  

        else  

        if(strcmp(pc->cheie,cheie)<0) /* căutare cuvânt în  

                                              subarborele drept */  

        {  

            pc->drept=cauta(pc->drept,cheie,nr_linie);  

            return pc;  

        }  

        else /* cuvântul a fost găsit și se va insera un
    }
```

```

        nod în lista înlățuită */
    {
        rand *pr;
        pr=(rand*)malloc(sizeof(rand));
        pr->nr=nr_linie;
        pr->urm=NULL;
        pc->ultim->urm=pr;
        pc->ultim=pr;
        return pc;
    }
}

void listeaza_cuvant(cuvant *pc) /* listare cuvânt și linii
apariție */
{
    rand *pr;
    printf("%s\t",pc->cheie);
    for(pr=pc->prim;pr!=NULL;pr=pr->urm)
        printf("%d ",pr->nr);
    printf("\n");
}

void parcurge_arbore(cuvant *pc) /* parcurgere arbore în
inordine */
{
    if(pc!=NULL)
    {
        parcurge_arbore(pc->stang);
        listeaza_cuvant(pc);
        parcurge_arbore(pc->drept);
    }
}

cuvant* distruge_arbore(cuvant *pc) /* eliberare memorie
arbore */
{
    if(pc!=NULL)
    {
        pc->stang=distruge_arbore(pc->stang);
        pc->drept=distruge_arbore(pc->drept);

        /* se șterge lista simplu înlățuită din nod */
        rand *pr,*pr2;
        pr=pc->prim;
        while(pr!=NULL)
        {
            pr2=pr;
            pr=pr->urm;
            free(pr2);
        }
        pc->prim=pc->ultim=NULL;

        free(pc);
    }

    return NULL;
}

```

```

void parurge_fisier(char *nume)
{
    FILE *f=NULL;
    cuvant *radacina=NULL;
    char cuv[LMAXCUVANT],c;
    int i=0;
    int n=1;

    if((f=fopen(nume,"rt"))==NULL)
    {
        printf("Eroare la citire fisier...\n");
        return;
    }

    while(!feof(f))
    {
        fscanf(f,"%c",&c);
        if((((c>='a')&&(c<='z'))||( (c>='A')&&(c<='Z')) )
            /* citire identificatori */
        {

            while((((c>='a')&&(c<='z'))||( (c>='A')&&(c<='Z')) )&&(i<
                LMAXCUVANT)&&( !feof(f)))
            {
                cuv[i]=c;
                i++;
                fscanf(f,"%c",&c);
            }

            cuv[i]=0;
            radacina=cauta(radacina,cuv,n);
            i=0;
        }

        if((c>='0')&&(c<='9')) /* citire constante
                                         numerice */
        {
            while((((c>='0')&&(c<='9'))&&( !feof(f)))
            {
                fscanf(f,"%c",&c);
            }
        }

        if(c==10) /* citire rand nou */
        {
            n++; /* incrementare nr. linie */
        }
    }
    fclose(f);

    parurge_arbore(radacina);
    radacina=distrug_arbore(radacina);
}

void main(int argc, char* argv[])
{
    if(argc>1)

```

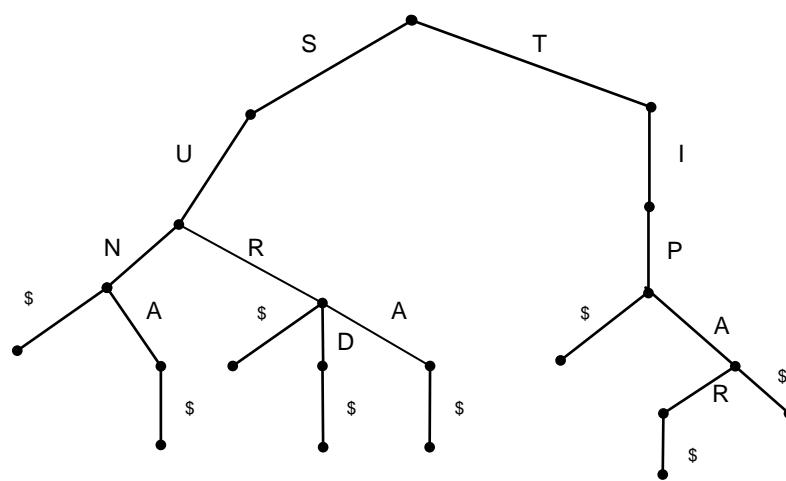
```
{  
    parcurge_fisier(argv[1]);  
}  
-----
```

- Desigur, programul de față abordează la nivel schematic partea de analiză a textului și de evidențiere a cuvintelor.
- Pornind de la această formă simplificată se poate concepe o abordare mai complexă a construcției tabelii de referințe încrucișate care ține cont și de alte aspecte cum ar fi delimitatorii sau caracterele speciale.

## 8.4. Arbori de regăsire (“Trie Trees”)

### 8.4.1. Definire

- Arborii de regăsire sunt structuri de date speciale care pot fi utilizate în reprezentarea **mulțimilor de caractere**.
- De asemenea cu ajutorul lor pot fi reprezentate tipuri de date care sunt **șiruri de obiecte** de orice tip sau **șiruri de numere**.
- În literatura de specialitate **arborii de regăsire** sunt cunoscuți sub denumirea de structuri **trie**, cuvânt derivat din cuvântul “**retrieval**” (regăsire).
- Un **arbore de regăsire** permite implementarea simplă a operatorilor definiți asupra unei structuri de date **mulțime** ale cărei elemente sunt **șiruri de caractere** (cuvinte).
- Este vorba în principiu despre operatorii:
  - *Inserează.*
  - *Suprimă.*
  - *Apartine.*
  - *Initializează.*
  - *Afișează.*
- Ultimul operator realizează afișarea tuturor membrilor (cuvintelor) mulțimii.
- Utilizarea arborilor de regăsire este eficientă atunci când există **mai multe cuvinte** care **încep cu aceeași secvență de caractere**, adică atunci când numărul de **prefixe distințe** al tuturor cuvintelor din mulțime este mult mai redus decât numărul total de cuvinte.
- Într-un **arbore de regăsire** fiecare **drum** de la **rădăcină** spre un **nod terminal** corespunde unui **cuvânt al mulțimii** care este reprezentată de arbore.
  - Astfel **nodurile** arborelui de regăsire corespund **prefixelor** cuvintelor mulțimii.
  - Pentru delimitarea cuvintelor se folosește un **caracter special de sfârșit** (simbolul `\$`).
- În figura 8.4.1.a apare un arbore de regăsire reprezentând o mulțime formată din cuvintele SUN, SUNA, SUR, SURD, SURA, TIP, TIPA, TIPAR.
  - Rădăcina corespunde sirului vid, iar cei doi fii ai săi corespund prefixelor S și T.
  - Parcurgând drumurile arborelui rezultă celealte cuvinte precizate.



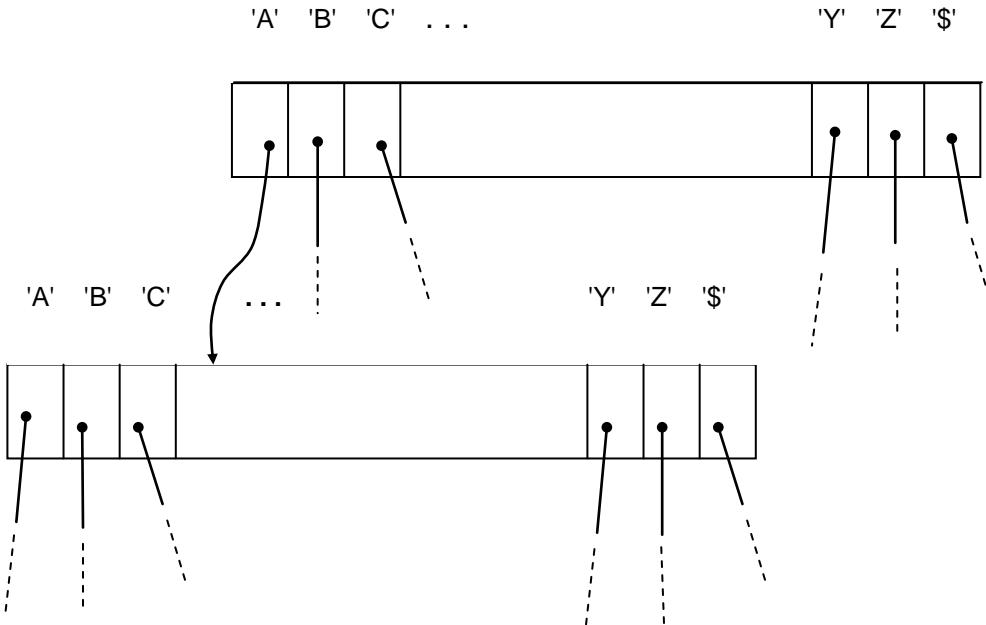
SUN, SUNA, SUR, SURD, SURA, TIP, TIPA, TIPAR

**Fig.8.4.1.a.** Arbore de regăsire

- În legătură cu **arborii de regăsire** se pot face următoarele observații:
  - (1) Fiecare nod al arborelui poate avea cel mult 29 de fii (câte unul pentru fiecare literă a alfabetului, plus caracterul `\$').
  - (2) Cele mai multe dintre noduri vor avea mult mai puțini fii decât 29.
  - (3) Un nod la care se ajunge printr-un ram etichetat cu caracterul `\$', **nu** poate avea nici un fiu și care atare poate fi eventual omis din structură.

#### 8.4.2. Structura de date "Nod arbore de regăsire"

- Un nod al unui arbore de regăsire poate fi privit ca o **structură asociere** al cărei **domeniu** este mulțimea  $\{A, B, C, \dots, Z, \$\}$  și al cărei **codomeniu** este o **mulțime de valori** aparținând tipului ReferintăNodArboreDeRegăsire.
- Cu alte cuvinte o **asociere** definită pe mulțimea **caracterelor** cu valori **în mulțimea pointerilor la noduri** (fig.8.4.2.a).



**Fig.8.4.2.a.** Structura de date NodArboreDeRegăsire

- Un **arbore de regăsire** poate fi identificat printr-un pointer la **rădăcina** sa (care este de fapt un nod).
- În consecință rezultă că **structurile de date abstracte** “ArboreDeRegăsire” și “NodArboreDeRegăsire” pot fi încadrate într-o **aceeași** structură de date, deși **operatorii specifi** care se aplică fiecăreia dintre ele sunt substanțial **diferiți**.
- Presupunând spre exemplu că:
  - $p$  este un pointer la o structură NodArboreDeRegăsire care este încadrat în tipul RefNodArboreDeRegasire.
  - $Nod$  o instanță a structurii NodArboreDeRegăsire
  - $c$  este o valoare de tip caracter,
- Asupra structurii NodArboreDeRegăsire se definesc următorii operatori:
  1. **Initializează** (*NodArboreDeRegasire Nod*) – operator care face ca *Nod* să indice asocierea vidă.
  2. **Atribuie** (*NodArboreDeRegasire Nod, char c, RefNodArboreDeRegasire p*) – operator care asociază caracterului *c* din nodul *Nod* referința *p*. După cum se observă, *p* este o referință la un nod al arborelui.

- 3. **RefNodArboreDeRegasire ValoareNod**  
 $(NodArboreDeRegasire Nod, char c)$  - operator care returnează referința asociată caracterului  $c$  din  $Nod$ .
  
- 4. **NodNou** ( $NodArboreDeRegasire Nod, char c$ ) - operator care face ca valoarea lui  $Nod$  pentru caracterul  $c$  să fie un pointer la un nod nou inițializat pe asocierea nulă.

#### 8.4.2.1. Implementare bazată pe tablouri a structurii **Nod arbore de regăsire**

- O implementare simplă a **nodurilor unui arbore de regăsire** este cea bazată pe un **tablou de pointeri la noduri**, al cărui **set de indici** este format din mulțimea caracterelor  $A, B, C, \dots, Z, \$$ .
- Se pot defini următoarele **structuri de date** [8.4.2.1.a].

---

```
/*Structura NodArboreDeRegasire - Implementare bazată pe
tablouri*/
```

```
int Dim_Nod=29;
typedef ref_nod_arb_regăsire * nod_arb_regăsire;
/*[8.4.2.1.a]*/
typedef ref_nod_arb_regăsire nod_arb_regăsire[Dim_Nod];
```

---

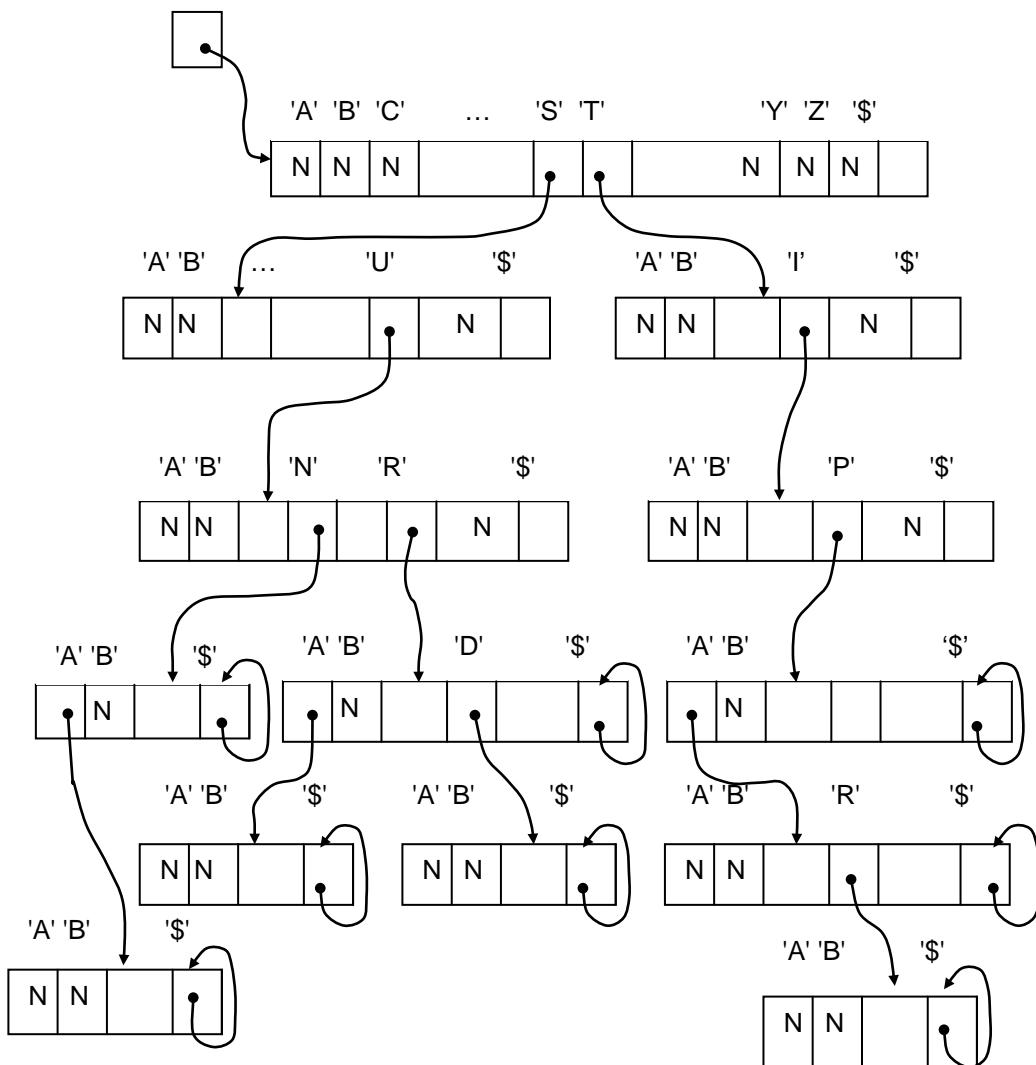
```
{Structura NodArboreDeRegasire - Implementare bazată pe
tablouri - varianta PASCAL}
```

```
TYPE RefNodArboreDeRegăsire=^NodArboreDeRegăsire;
    Caractere=( 'A', 'B', 'C', ..., 'Z' ,'$' );
/*[8.4.2.1.a]*/
    NodArboreDeRegăsire = ARRAY [Caractere] OF
        RefNodArboreDeRegăsire;
```

---

- Varianta pseudocod a implementării operatorilor definiți pe structura **NodArboreDeRegăsire** apare în secvența [8.4.2.1.b].
- Pentru a **nu supraîncărca** structura cu noduri frunză care sunt fiii corespunzători caracterului '\$' se adoptă următoarea **convenție**:
  - (1)  $Nod['\$']$  are valoarea null.
    - În acest caz nodul **nu** are fiu corespunzând caracterului '\$'.
  - (2)  $Nod['\$']$  are valoarea **pointerului** care indică nodul însuși.
    - În acest caz se presupune că nodul are un astfel de fiu, care de fapt **nu** se crează niciodată.

- O astfel de situație precizează **sfârșitul** unui **cuvânt** pe **nivelul anterior** al structurii arborelui de regăsire, respectiv la părintele nodului respectiv (fig.8.4.2.1.a).



**Fig.8.4.2.1.a.** Reprezentarea structurii ArboreDeRegasire

---

/\*Implementarea bazată pe tablouri a operatorilor  
TDA NodArboreDeRegăsire - varianta pseudocod\*/

```

Subprogram Initializeaza(nod_arb_regăsire Nod)
    CHAR c;
    pentru c='A' pana la '$'
        Nod[c]= NULL;
    /*Initializeaza*/
    
Subprogram Atribuie(nod_arb_regăsire Nod, CHAR c,
    ref_nod_arb_regăsire p)
    Nod[c]= p;                                [ 8.4.2.1.b ]
/*Atribuie*/
    
ref_nod_arb_regăsire ValoareNod(nod_arb_regăsire Nod, CHAR c)
    returneaza Nod[c];
/*ValoareNod*/
    
Subprogram NodNou (nod_arb_regăsire Nod, CHAR c);

```

```
Nod[c] = aloca_memorie(nod_arb_regasire);
Initializeaza(Nod[c]);
/*NodNou*/
```

---

#### 8.4.2.2. Implementare bazată pe liste înlățuite a structurii **Nod arbore de regăsire**

- Implementarea nodurilor arborilor de regăsire cu ajutorul tablourilor este **ineficientă** deoarece:
    - (1) Se rezervă **în fiecare nod** loc pentru pointeri la **toate literele** alfabetului.
    - (2) Se ajunge astfel ca **volumul de memorie** ocupat de structură să **depășească** cu mult lungimea totală a cuvintelor mulțimii.
  - Sunt posibile însă și alte reprezentări.
  - Pornind de la observația că un nod al unui arbore de regăsire este de fapt o **asociere**, în principiu poate fi utilizată **orice reprezentare** a unei astfel de **structuri asociere**.
  - În cazul de față, deoarece pentru un nod **domeniul** asocierei poate conține un număr variabil de elemente, apare ca deosebit de potrivită reprezentarea bazată pe **liste înlățuite**.
  - Astfel **asocierea** corespunzătoare unui **nod al unui arbore de regăsire** poate fi reprezentată ca o **listă înlățuită** de caractere pentru care valoarea asociată este un pointer diferit de NULL [8.4.2.2.a].
- 

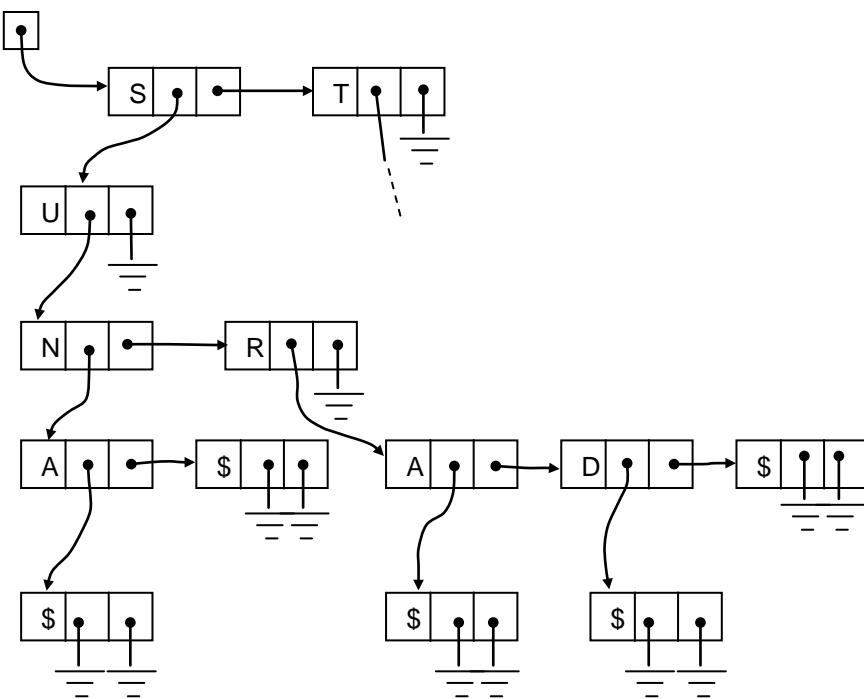
```
/*Structura NodArboreDeRegasire - Implementare bazată pe
liste înlățuite - varianta C*/
typedef Nod_lista * ref_nod_lista;

typedef struct Nod_lista
{
    char domeniu; /*[8.4.2.2.a]*/
    ref_nod_lista valoare; /*indică prima celulă
                           a listei nodului fiu*/
    ref_nod_lista urmator; /*indică următoarea
                           celulă a listei nodului curent*/
} Nod_lista;

typedef ref_nod_lista nod_arb_regasire;
```

---

- În figura 8.4.2.2.a apare reprezentată în această manieră o porțiune a arborelui de regăsire din figura 8.4.1.a.



**Fig.8.4.2.2.a.** Arbore de regăsire implementat cu ajutorul listelor

- În aceste condiții, operatorii **Initializeaza**, **Atribuie**, **ValoareNod** și **NodNou** definiți pentru structura **NodArboreDeRegăsire** se pot implementa cu ușurință.

### 8.4.3. Structura de date Arbore de regăsire

- Pornind de la cele anterior precizate, în continuare se poate defini **structura de date ArboreDeRegăsire**:

```
typedef ref_nod_arb_regasire tip_arb_regasire; /*[8.4.3.a]*/
```

- Pe această structură de date se pot implementa operatorii specifici: **Inițializare**, **Inserează**, **Apartine**, **Suprimă** și **Afișează**.
- Pentru exemplificare se prezintă implementarea operatorului **Inserează** utilizând drept suport, operatorii definiți pe structura **nod\_arbore\_regasire** implementată la rândul ei cu ajutorul **tablourilor**.
  - Se presupune că în prealabil este definit un **tip\_cuvânt** reprezentat printr-un **tablou de caractere**.
  - Acest tablou de caractere conține un singur cuvânt care se termină cu caracterul '\$'.
- În acest context, procedura **Inserează(tip\_cuvânt x, tip\_arb\_regasire cuvinte)** care inserează cuvântul **x** în mulțimea cuvinte reprezentată printr-un arbore de regăsire, apare în secvența [8.4.3.b].

```
/*Implementarea operatorului Insereaza pentru arbori de
```

**regăsire implementați cu ajutorul tablourilor - varianta pseudocod\*/**

```
typedef char sir[] tip_cuvânt; /*cuvântul de inserat*/

Subprogram Inserează (tip_cuvant x, tip_arb_regasire
cuvintele)
/*inserează cuvântul x în arborele de regăsire cuvinte*/

    int i; /*precizează poziția caracterului curent în
            cuvântul x*/
    tip_arb_regasire t; /*utilizat pentru a parcurge
            nodurile arborelui de regăsire corespunzator
            prefixelor lui x*/
    i=0;                                /*[8.4.3.b]*/
    t=cuvintele; /*rădăcina arbore*/
    cât_timp (x[i]!='$')
        daca (ValoareNod (*t, x[i])== NULL) NodNou(*t, x[i]);
            /*dacă nodul curent nu are fiu pentru caracterul
            x[i], atunci se creează unul*/
            t=ValoareNod (*t,x[i]); /*se trece la fiul lui t
            pentru caracterul x [i] (chiar dacă a fost
            creat recent)*/
            i=i+1; /*se avansează la caracterul următor în
            cuvântul x*/
        □ /*cât_timp*/
        /*s-a ajuns la caracterul '$' în cuvantul x*/
        Atribuie(*t,'$',t) /*se face o buclă pentru '$' pentru a
            marca un nod terminal*/
    /*Insereaza*/
-----
```

- Se presupune că cuvântul de inserat **nu** se găsește în structură.
- **Procedura** funcționează după cum urmează:
  - i este cursorul de caractere în tabloul x.
  - t este un pointer utilizat în parcurgerea nodurilor arborelui de regăsire.
  - Se parcurge cuvântul x caracter cu caracter în bucla **cât\_timp** până la întâlnirea caracterului '\$'.
  - Pentru fiecare caracter  $x[i]$  diferit de caracterul '\$' întâlnit, se verifică dacă nodul curent are un fiu pentru acest caracter. Dacă **nu** are, se crează un astfel de fiu.
  - Se trece pe nivelul următor al arborelui la fiul caracterului  $x[i]$ . Acest lucru se întâmplă chiar dacă nodul a fost creat în iterația curentă.
  - Se avansează la caracterul următor în cuvântul x și se reia bucla de prelucrare.
  - Când se ajunge la sfârșitul cuvântului x, ( $x[i] = '$'$ ), parcurgerea arborelui de regăsire a avansat până la nivelul următor ultimului caracter introdus.
  - În acest moment se realizează înlățuirea  $\text{Nod}['$'] = t$ , adică se realizează înlățuirea nodului cu el însuși pentru a marca **sfârșitul** cuvântului introdus.

- Implementarea operatorilor **Inițializare**, **Apartine**, **Suprimă** și **Afișează** se recomandă ca și exercițiu.
- Într-o manieră similară se poate realiza implementarea acelorași operatori utilizând de această dată implementarea bazată pe liste înlănuite a arborilor de regăsire.

#### 8.4.4. Evaluarea performanței structurii Arbore de regăsire

- În cele ce urmează ne propunem să realizăm o **analiză comparată** din punctul de vedere al **timpului** și al **volumului** de memorie necesar pentru **reprezentarea mulțimilor**.
  - Se pornește de la următoarele considerente:
    - Se reprezintă  $n$  cuvinte.
    - Cuvintele au  $p$  prefixe diferite.
    - Lungimea totală a tuturor cuvintelor este de  $m$  caractere.
  - Analiza comparată se va realiza utilizând în acest scop structurile **tabelă de dispersie** respectiv **arbore de regăsire**.
    - În cele ce urmează se consideră **pointerii** implementați pe 4 octeți.
  - Probabil că cel mai eficient mod de a memora cuvintele, din punct de vedere al spațiului ocupat, mod care permite implementarea simplă a operatorilor **Inserează** și **Suprimă** este **tabela de dispersie deschisă** bazată pe **înlătuirea directă** (Vol.1 & 7.3.3.1).
  - Deoarece cuvintele au lungime variabilă, nodurile listelor asociate tabelei de dispersie **nu** vor conține chiar cuvintele, ci fiecare nod va conține două referințe:
    - Un pointer care înlănuie într-o **listă înlănuitură** nodurile ale căror **indici primari** sunt identici.
    - Un idicator care indică începutul cuvântului într-o **zonă de cuvinte**.
  - Cuvintele vor fi memorate în **zona de cuvinte**, ca și un tablou de caractere, sfârșitul fiecarui cuvânt fiind indicat de caracterul '**\$**'.
  - Spre exemplu cuvintele SUN, SUNA, SURD vor fi memorate astfel:
- |                                     |
|-------------------------------------|
| 0 1 2 3 4 5 6 7 8 9 10 11 12 13 ... |
| S U N \$ S U N A \$ S U R D \$ ...  |
- Pointerii la cele trei cuvinte sunt de fapt **cursori** la pozițiile 0, 4 și 9 ale tabelului de cuvinte, poziții care marchează începutul cuvintelor respective.
  - **Spațiul de memorie** necesar pentru implementarea bazată pe structura **tabelă de dispersie deschisă** poate fi determinat pe baza calculului următor:

- a) Sunt necesari  $8n$  octeți pentru **nodurile listelor asociate tabelei** (avem  $n$  cuvinte fiecare necesitând câte doi pointeri a 4 octeți).
  - b) Sunt necesari  $m+n$  octeți pentru cele  $n$  **cuvinte** memorate în **zona de cuvinte** cu lungimea totală  $m$  și caracterele lor de sfârșit;
  - c) Sunt necesari  $4d$  octeți pentru **tabela de dispersie** propriu-zisă, unde  $d$  este dimensiunea tabelei (un număr prim).
- Însumând aceste valori, în **total** pentru reprezentarea bazată pe structura **tabela de dispersie deschisă** rezultă un necesar de  **$9n+m+4d$**  octeți.
  - În cazul **arborelui de regăsire** ale cărui noduri sunt implementate ca și **liste înlántuite** sunt necesare  $p+n$  noduri, unde:
    - $p$  reprezintă numărul de noduri destinate **prefixelor** (câte unul pentru fiecare prefix).
    - $n$  reprezintă numărul de noduri destinate **sfârșiturilor de cuvinte** (câte unul pentru fiecare cuvânt).
  - Fiecare **nod** conține un caracter și doi pointeri necesitând 9 octeți de memorie.
  - În total reprezentarea bazată pe structura **arbore de regăsire** necesită  **$9n+9p$**  octeți.
  - Se compară necesarul de  $9n+m+4d$  octeți pentru tabela de dispersie cu  $9n+9p$  octeți, necesarul pentru arborele de regăsire
    - De fapt trebuie comparate valorile  $m+4d$  și  $9p$ .
    - De regulă în aplicații care implementează dicționare de mari dimensiuni, raportul  $m/p$  este de obicei mai mic ca 3, deci  $m < 3p$ .
  - În consecință, concluzia este că **tabela de dispersie** va utiliza mai puțin spațiu.
  - Din punctul de vedere al **timpului de execuție**:
    - În cazul **arborelui de regăsire** realizarea operațiilor **Inserează**, **Suprimă** și **Apartine** consumă un **timp** proporțional cu lungimea cuvântului implicat.
    - În cazul **tabelului de dispersie**, calculul valorii funcției de dispersie consumă un interval de timp de calcul comparabil cu realizarea operației **Apartine** într-un arbore de regăsire.
    - Deoarece calculul valorii funcției de dispersie **nu** include timpul necesar rezolvării problemelor de **coliziune**, sau realizării efective a inserției, a suprimării sau testului de apartenență, este de așteptat ca **arborii de regăsire** să fie **considerabil mai rapizi** decât **tabelele de dispersie**.
  - Un alt avantaj al **arborelui de regăsire** este acela că **suportă** realizarea eficientă a operației **Min**, obiectiv dificil de realizat în cazul **tabelei de dispersie**.

## 8.5. Arbori binari echilibrați. Arbori AVL

### 8.5.1. Definirea arborilor echilibrați AVL

- Din **analiza** căutării în **arbori binari ordonați** prezentată în &8.3.6. rezultă în mod evident că o procedură de inserare care **restaurează** structura arbore astfel încât ea să fie **tot timpul perfect echilibrată** nu este viabilă, deoarece activitatea de restructurare este foarte **complexă**.
- Cu toate acestea sunt posibile anumite abordări mai realiste, dacă termenul "**echilibrat**" este definit într-o manieră **mai puțin strictă**.
- Astfel de criterii de echilibrare "**imperfectă**" pot conduce la **tehnici** mai simple de **reorganizare a structurilor arbori binari ordonați**, al căror cost deteriorează într-o măsură redusă **performanța medie de căutare**.
- Una dintre aceste definiții ale **echilibrării arborilor binari ordonați** este cea propusă de **Adelson, Velskii și Landis** în 1962 și care are următorul enunț:
  - Un **arbore binar ordonat** este **echilibrat** dacă și numai dacă pentru oricare nod al arborelui, înălțimile celor doi subarborei diferă cu cel mult 1.
  - Arboare care satisfac acest criteriu se numesc "**arbori AVL**" după numele inventatorilor.
  - În cele ce urmează, acești arbori vor fi denumiți "**arbori echilibrați**".
    - Se atrage atenția asupra faptului că **arborii perfect echilibrați** sunt de asemenea **arbori AVL**.
  - Această definiție are câteva **avantaje**:
    - (1) Este foarte **simplă**.
    - (2) Conduce la o **procedură** viabilă de re-echilibrare.
    - (3) Asigură o **lungime medie a drumului de căutare** practic identică cu cea a unui **arbore perfect echilibrat**.
  - În acest context se vor studia următorii operatori definiți în cadrul structurii **arbore echilibrat**:
    - 1º **Căutarea** unui nod cu o cheie dată.
    - 2º **Inserția** unui nod cu o cheie dată.
    - 3º **Suprimarea** unui nod cu o cheie dată.
  - Toți acești operatori necesită un **efort de calcul** de ordinul  $O(\log_2 n)$ , unde  $n$  este numărul nodurilor structurii, chiar în **cel mai defavorabil caz**.
  - Acest lucru este o consecință directă a teoremei demonstrată de **Adelson-Velskii și Landis**, care garantează că:

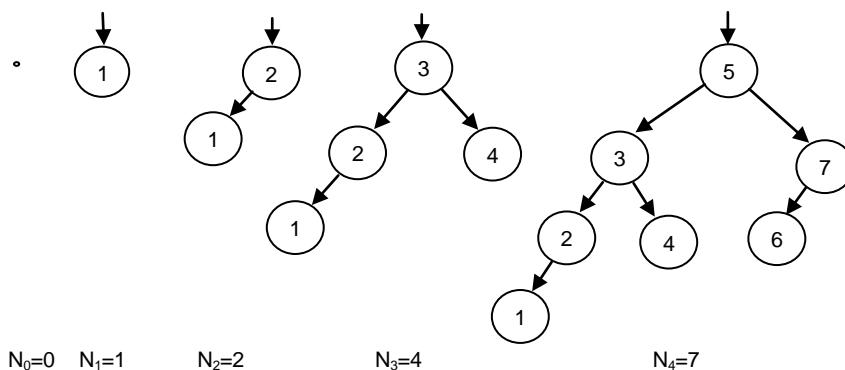
- Un arbore echilibrat **nu** va fi niciodată cu mai mult de **45 %** mai înalt decât omologul său perfect echilibrat, **indiferent** de numărul de noduri pe care-l conține.
- Dacă se notează cu  $h_o(n)$  înălțimea unui **arbore echilibrat cu n noduri**, atunci este valabilă relația [8.5.1.a].

$$\log(n+1) < h_o(n) < 1.4404 * \log(n+2) - 0.328 \quad [8.5.1.a]$$

- Optimul este atins de arborii echilibrați având un număr de noduri  $n=2^k-1$ .

### 8.5.2. Arbori Fibonacci

- Se consideră următoarea problemă:
  - Se dă o anumită **înălțime**  $h$  și se cere să se construiască **arborele echilibrat AVL** care conține **numărul minim de noduri** pentru înălțimea dată.
  - Ca și în cazul arborelui având pe  $h$  minim, valoarea dorită a lui  $h$  poate fi atinsă numai pentru **anumite** valori specifice ale lui  $n$ .
  - Se notează **arborele de înălțime  $h$**  cu număr minim de noduri cu  $A_h$ .
  - În acest caz,  $A_0$  este arborele vid iar  $A_1$  este un arbore cu un singur nod.
  - Pentru a construi arborele  $A_h$  pentru  $h>1$ , se va porni de la rădăcină, căreia îi se vor ataşa doi subarbore care le rândul lor vor avea un număr minim de noduri. Acești arbori sunt de asemenea de tip A.
  - Evident unul din subarborei trebuie să aibă înălțimea  $h-1$ , iar celuilalt îi este permisă o înălțime cu 1 mai mică deci ( $h-2$ ).
  - În figura 8.5.2.a sunt reprezentări arbori de înălțime 2, 3 și 4.



**Fig.8.5.2.a.** Arboare Fibonacci de înălțime 2 , 3 și 4

- Deoarece principiul lor de alcătuire seamănă foarte mult cu numerele lui Fibonacci, adică fiecare arbore curent se construiește din cei doi anteriori, ei se numesc **arbori Fibonacci**.

- **Arborii Fibonacci** se definesc **formal** după cum urmează:
  1. Arborele vid  $A_0$  este arborele Fibonacci de înălțime 0.
  2. Arborele cu un singur nod  $A_1$  este arborele Fibonacci de înălțime 1.
  3. Dacă  $A_{h-1}$  și  $A_{h-2}$  sunt arbori Fibonacci de înălțime  $h-1$  respectiv  $h-2$ , atunci  $A_h = \langle A_{h-1}, r, A_{h-2} \rangle$  este arborele Fibonacci de înălțime  $h$ , având rădacina  $r$ .
  4. Nici un alt arbore **nu** este un arbore Fibonacci.
- Numărul de noduri ale lui  $A_h$  este definit de următoarea relație de recurență [8.5.2.a], care justifică de fapt denumirea de **arbori Fibonacci**.

---


$$\begin{aligned}N_0 &= 0, \quad N_1 = 1 \\N_h &= N_{h-1} + 1 + N_{h-2}\end{aligned}$$


---

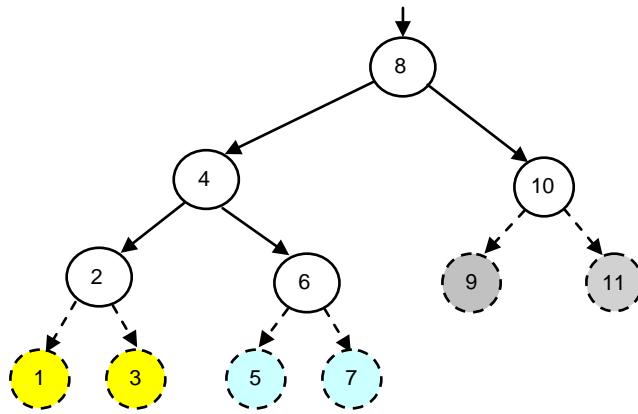
[ 8 . 5 . 2 . a ]

- Numerele  $N_i$  sunt numerele de noduri valabile pentru **cazul cel mai defavorabil**, respectiv când se atinge limita superioară a lui  $h$  în relația [8.5.1.a].
- **Arborii Fibonacci** sunt cazurile cele mai **dezavantajoase** de **arbori AVL**.

### 8.5.3. Inserția nodurilor în arbori echilibrați AVL

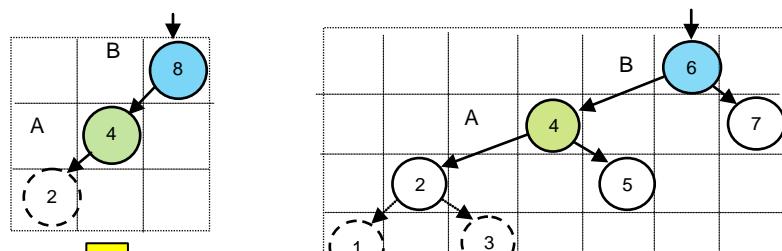
- Se dă un **arbore AVL** având rădăcina  $R$  și pe  $S$  de înălțime  $h_S$  și pe  $D$  de înălțime  $h_D$ , pe post de subarbori stâng respectiv drept.
  - Se cere să se insereze un nod nou în acest arbore.
- În cazul **inserției** se pot distinge trei cazuri.
  - Se presupune că nodul nou se inserează în **subarborele stâng S**, determinând creșterea cu 1 a înălțimii acestuia.
    - (1)  $h_S = h_D$  : în urma inserției  $S$  și  $D$  devin de înălțimi inegale, fără însă a viola criteriul echilibrului.
    - (2)  $h_S < h_D$  : în urma inserției  $S$  și  $D$  devin de înălțimi egale, echilibrul fiind îmbunătățit.
    - (3)  $h_S > h_D$  : criteriul echilibrului este violat și arborele trebuie **reechilibrat**.
- Astfel, în arborele echilibrat din figura 8.5.3.a:
  - Nodurile 9 sau 11 pot fi inserate **fără** reechilibrare.

- Inserția unuia din nodurile 1 , 3 , 5 sau 7 necesită însă **reechilibrarea** arborelui.

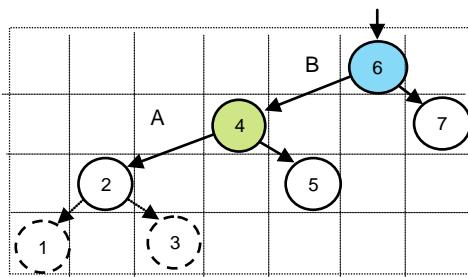


**Fig.8.5.3.a.** Arbore echilibrat AVL

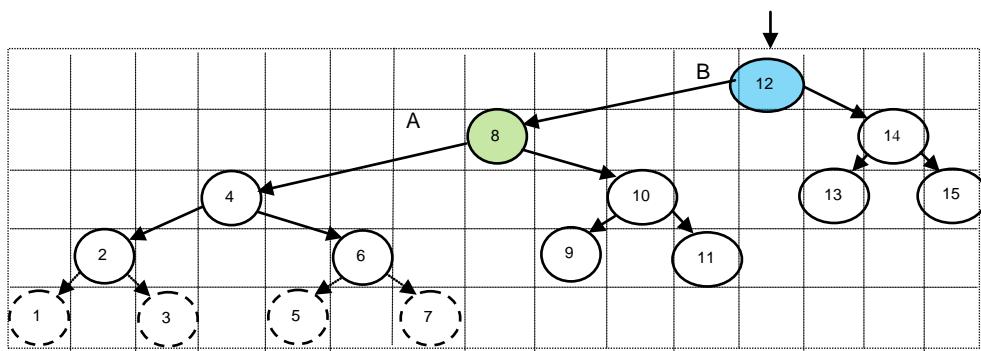
- O analiză atentă a situațiilor posibile care rezultă în urma inserției evidențiază faptul că există numai **două configurații** care necesită tratamente speciale.
- Celelalte configurații pot fi reduse la aceste două situații din considerente de **simetrie**.
- **Prima situație** se referă la inserția nodurilor 1 sau 3 în arborele reprezentat cu linie continuă în figura 8.5.3.a.
- Cea de-a două situație se referă la inserția nodurilor 5 sau 7 în arborele din figura 8.5.3.a.
- Cele două situații sunt prezentate în figurile 8.5.3.b și 8.5.3.c, fiecare în câte trei ipostaze (a), (b) și (c) care evoluează de la simplu la complicat.
  - Cele două situații sunt denumite cazul "**1 Stânga**" respectiv cazul "**2 Stânga**".
  - Ambele cazuri presupun creșterea **subarborelui stâng** S, ca atare reprezintă un **caz stânga**.
    - **Cazul 1 Stânga** presupune creșterea **subarborelui stâng** al **subarborelui stâng** al arborelui în cauză .
    - **Cazul 2 Stânga** presupune creșterea **subarborelui drept** al **subarborelui stâng** al arborelui în cauză.
- Elementele adăugate prin inserție apar cu linie punctată.



(a)

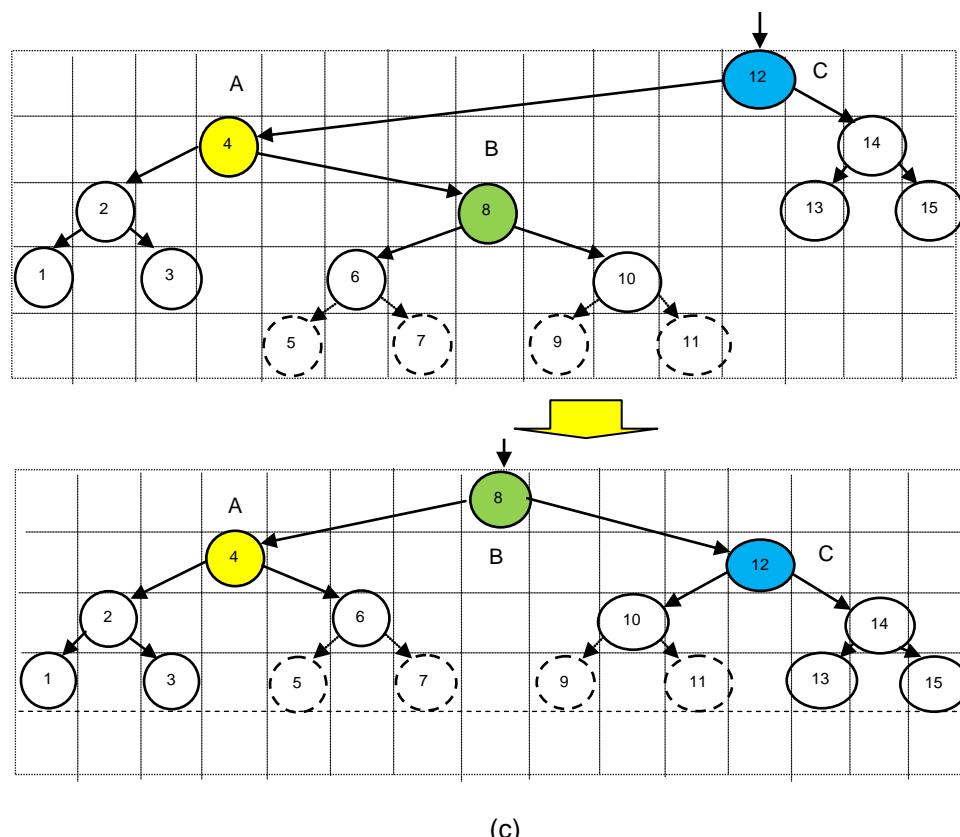
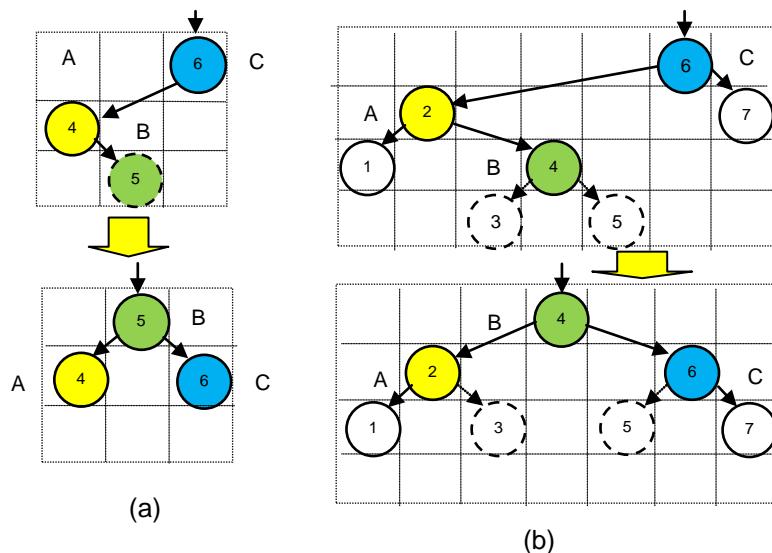


(b)



(c)

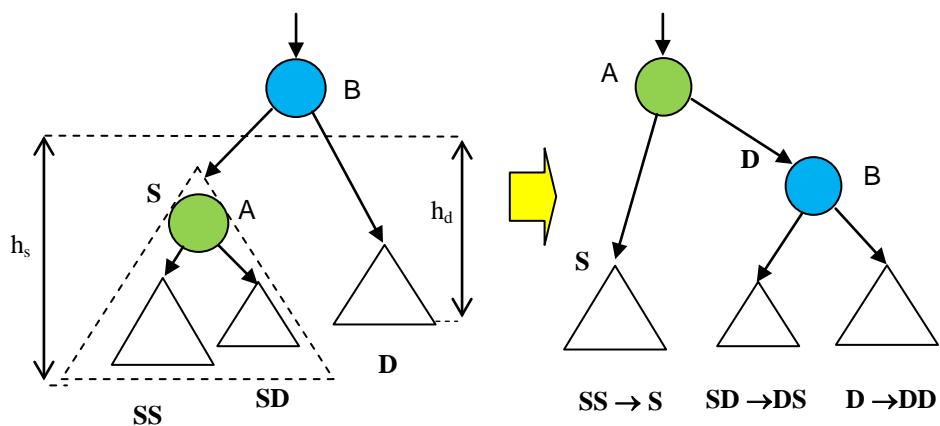
**Fig.8.5.3.b.** Echilibrarea arborilor AVL. Cazul 1Stânga



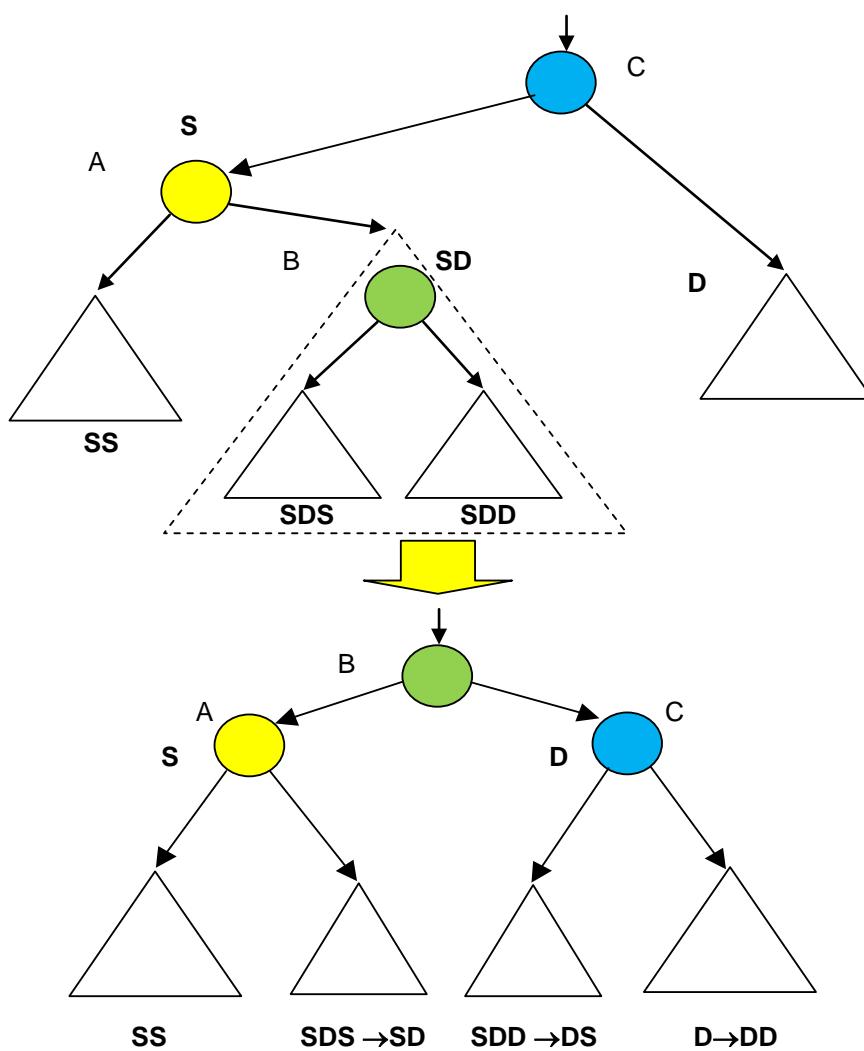
**Fig.8.5.3.c. Echilibrarea arborilor AVL, Cazul 2 Stânga**

- Prin **transformări simple**, structurile arbore se reechilibrează.
    - În **cazul 1 Stânga** este vorba despre o rotație simplă a două noduri respectiv A și B.
    - În **cazul 2 Stânga** este vorba despre o rotație dublă în care sunt implicate trei noduri: A, B și C.
      - Se subliniază faptul că arborii AVL fiind **arbori ordonați**, singurele mișcări permise ale nodurilor sunt cele pe **verticală**.

- Pozițiile relative ale proiecțiilor pe orizontală ale nodurilor aparținând unui arbore AVL, trebuie să rămână nemodificate.
- Sinteză acestor cazuri precum și modul sintetic în care se realizează procesul de echilibrare pentru cazurile pe stânga sunt prezentate în figurile 8.5.3.d și 8.5.3.e.



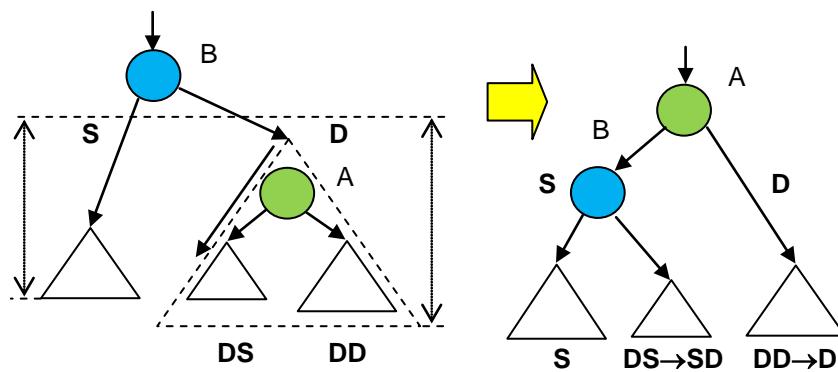
**Fig.8.5.3.d.** Echilibrarea arborilor AVL. **Cazul 1 Stânga.** Schema generală



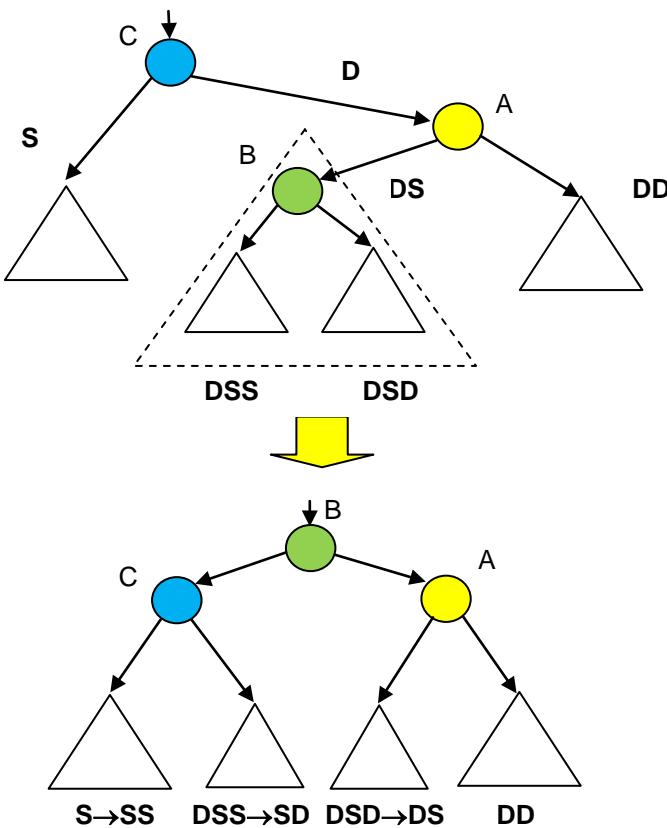
**Fig.8.5.3.e.** Echilibrarea arborilor AVL. **Cazul 2 Stânga.** Schema generală

- Aceleasi scheme sintetice de data aceasta pentru cazurile pe **dreapta** apar in figurile 8.5.3.f respectiv 8.5.3.g.

- Este vorba despre cazurile **1** respectiv **2 Dreapta**.



**Fig.8.5.3.f.** Echilibrarea arborilor AVL. **Cazul 1 Dreapta.** Schema generală



**Fig.8.5.3.g.** Echilibrarea arborilor AVL. **Cazul 2 Dreapta.** Schema generală

- De această dată a crescut **subarborele drept** al arborelui original și e necesară reechilibrarea.
- În oglindă cu cazurile pe **stânga** și aici distingem:
  - **Cazul 1 Dreapta** care presupune creșterea **subarborelui drept** al **subarborelui drept** al arborelui original.

- **Cazul 2 Dreapta** care presupune creșterea **subarborelui stâng** al **subarborelui drept** al arborelui original.
- Și în acest caz reechilibrarea se rezolvă prin **una** respectiv **două rotații** ale nodurilor A și B, respectiv ale nodurilor A, B și C.
- **Principal**, procesul de **echilibrare** împreună cu modalitatea efectivă de **restructurare** apar pentru fiecare din cele două cazuri în figurile mai sus precizate.
- Un **algoritm** pentru inserție și reechilibrare depinde în **mod critic** de maniera în care este memorată informația referitoare la **situația echilibrului arborelui**.
- (1) O soluție posibilă este aceea care exploatează faptul că această informație este conținută în **mod implicit** în structura arborelui.
  - În acest caz factorul de echilibru al unui nod afectat de inserție, trebuie determinat de fiecare dată prin parcurgerea arborelui, lucru care conduce la o **regie excesivă**.
- (2) O altă soluție este aceea prin care se atribuie **fiecărui nod** al arborelui un **factor explicit de echilibru**.
  - **Factorul de echilibru** se referă la subarborele a cărui **rădăcină** o constituie nodul în cauză.
  - **Factorul de echilibru** al unui **nod**, va fi interpretat, prin definiție, ca și diferența dintre înălțimea **subarborelui său drept** și înălțimea **subarborelui său stâng**.
- În acest caz structura unui nod devine [8.5.3.a]:

---

```
/*Structura unui nod al unui arbore AVL - varianta C*/

typedef struct Nod_AVL
{
    int cheie;
    int contor;
    struct nod* stang; /*[8.5.3.a]*/
    struct nod* drept;
    int ech; /*ia valorile -1 sau 0 sau +1*/
} nod;
```

---

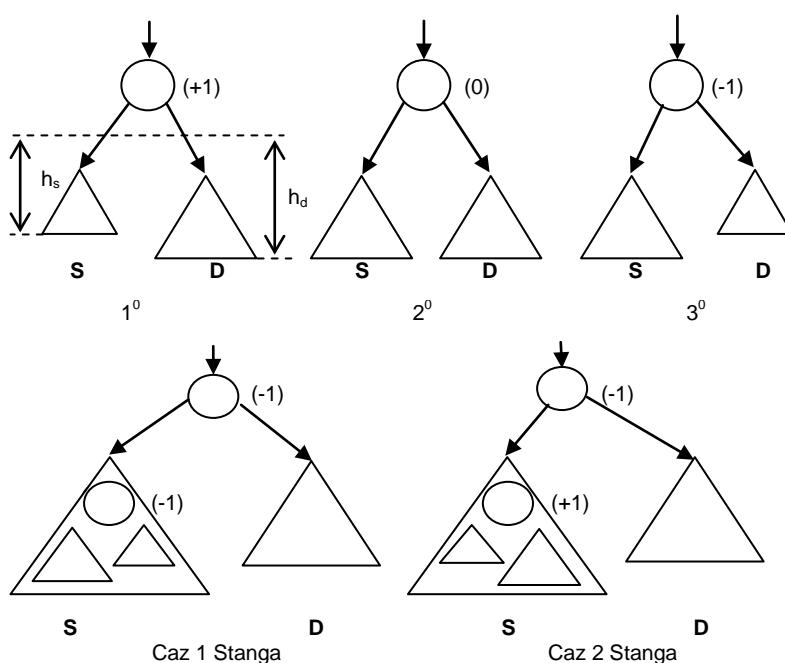
```
typedef Nod_AVL* ref_tip_nod_AVL;
```

---

```
{Structura unui nod al unui arbore AVL - varianta PASCAL}
```

```
TYPE TipRef=^TipNod
      TipNod=RECORD
          cheie:integer; [8.5.3.a]
          contor:integer;
          stang,drept:TipRef;
          ech:(-1,0,+1)
      END;
```

- Pornind de la **structura nod** definită în secvența [8.5.3.a], **inserția** unui nod se desfășoară în trei etape:
  - (1) Se parcurge arborele binar, pentru a verifica dacă nu cumva cheia există deja.
  - (2) Se înserează noul nod și se initializează factorul său de echilibru pe valoarea zero.
  - (3) Se revine pe drumul de căutare și se verifică factorul de echilibru pentru fiecare nod întâlnit, procedându-se la echilibrare acolo unde este cazul.
- Această metodă necesită unele verificări redundante:
  - Odată echilibrul stabilit, **nu** mai este necesară verificarea factorului de echilibru pentru strămoșii nodului.
- Cu toate acestea se va face totuși uz de ea, deoarece:
  - (1) Este ușor de înțeles.
  - (2) Se poate implementa printr-o **extindere a procedurilor recursive de căutare și inserție a nodurilor în arbori binari ordonați**, descrise în & 8.3.4.
- Aceste proceduri care includ operația de căutare a unui nod, datorită formulării lor **recursive**, asigură în manieră implicită "**revenirea de-a lungul drumului de căutare**".
- **Informația** care trebuie transmisă la **revenirea** din fiecare pas este cea referitoare la **modificarea înălțimii subarborelui** în care s-a făcut inserția.
  - Din acest motiv, în **lista de parametri ai procedurii** se introduce parametrul variabil boolean  $h$ , a cărui valoare "adevărat" semnifică **creșterea înălțimii subarborelui**.
  - Se presupune că procedura de inserție revine din **subarborele stâng** la un nod  $p^*$  (vezi fig.8.5.3.h), cu indicația că **înălțimea subarborelui stâng** a crescut.



**Fig.8.5.3.h.** Inserția în arbori AVL. Cazul Stânga. Schema generală

- Se pot distinge **trei situații** referitoare la înălțimea subarborelui **înaintea** respectiv **după** realizarea inserției:
  - (1)  $h_S < h_D$ , deci  $p^.ech = +1$ ; După inserție factorul de echilibru devine  $p^.ech = 0$ , ca atare inechilibrul anterior referitor la nodul  $p$  a fost rezolvat.
  - (2)  $h_S = h_D$ , deci  $p^.ech = 0$ ; După inserție factorul de echilibru devine  $p^.ech = -1$ , în consecință greutatea este acum înclinată spre stânga, dar arborele rămâne echilibrat în sensul AVL.
  - (3)  $h_S > h_D$ , deci  $p^.ech = -1$ ; ca atare este necesară **reechilibrarea arborelui**.
- În cazul  $3^0$ , **inspectarea** factorului de echilibru al **rădăcinii subarborelui stâng** ( $p1^.ech$ ) conduce la stabilirea cazului **1 Stânga** sau cazul **2 Stânga**.
  - (1) Dacă acest nod are la rândul său înălțimea subarborelui său stâng mai mare ca cea a celui drept, adică factorul de echilibru egal cu  $(-1)$ , suntem în cazul **1 Stânga**.
  - (2) Dacă factorul de echilibru al acestui nod este egal cu  $(+1)$  suntem în cazul **2 Stânga** (fig. 8.5.3.h).
  - (3) În această situație **nu** poate apărea un subarbore stâng a cărui rădăcină are un factor de echilibru nul [Wi76].
- **Operația de reechilibrare** constă dintr-o secvență de reasignări de pointeri.
  - De fapt pointerii sunt schimbați ciclic, rezultând fie o **rotație simplă** a două noduri, fie o **rotație dublă** a trei noduri implicate.
  - În plus, pe lângă rotirea pointerilor, **factorii de echilibru** respectivi sunt reajustați.
- Procedura care realizează acest lucru apare în secvența [8.5.3.b]. Principiul de lucru este cel ilustrat în figura 8.5.3.h.

**/\*Insertția unui nod într-un arbore echilibrat AVL -varianta pseudocod\*/**

```
Subprogram InsertEchilibrat(int x, ref_tip_nod_AVL p,
boolean *h)
/*insereză nodul cu cheia x în arborele echilibrat indicat de p și returnează h=TRUE dacă arborele își modifică înălțimea. Dacă nodul există deja în arbore îi incrementează contorul asociat*/

    ref_tip_nod_AVL p1,p2;
    *h=FALSE;
    dacă(p==NULL) /*cuvântul nu e arbore*/
        /*se crează și se insereză un nod nou*/
        p=aloca_memorie(tip_nod_AVL); /*creare nod nou*/
        p->cheie=x; p->contor=1;
        p->stang=NULL; p->drept=NULL; p->ech=0; *h=TRUE;
        □ /*daca*/
    altfel
        daca(x<p->cheie) /*parcuregere subarbore stâng*/
            InsertEchilibrat(x,p->stang,&h);
            daca(*h) /*ramura stângă a crescut în înălțime*/
                daca(p->ech==1) /*cazul ech=1*/
                    p->ech=0; *h=FALSE;
                    revenire;
                    □ /*daca*/
                [8.5.3.b]
                daca(p->ech==0) /*cazul ech=0*/
                    p->ech=-1;
                    revenire;
                    □ /*daca*/
                daca(p->ech== -1) /*cazul ech=-1 reechilibrare*/
                    p1=p->stang;
                    daca(p1->ech== -1) /*cazul 1 stânga*/
                        p->stang=p1->drept;
                        p1->drept=p;
                        p->ech=0; p=p1;
                        □ /*daca*/
                    altfel /*cazul 2 stânga*/
                        p2=p1->drept;
                        p1->drept=p2->stang;
                        p2->stang=p1;
                        p->stang=p2->drept;
                        p2->drept=p;
                        daca(p2->ech== -1)
                            p->ech=+1;
                        altfel
                            p->ech=0;
                        daca(p2->ech== +1)
                            p1->ech=-1;
                        altfel
                            p1->ech=0;
                        p=p2;
                        □ /*altfel*/
                    p->ech=0; *h=FALSE; revenire;
                    □ /*daca*/
                □ /*daca*/
            □ /*daca*/
        □ /*daca*/
```

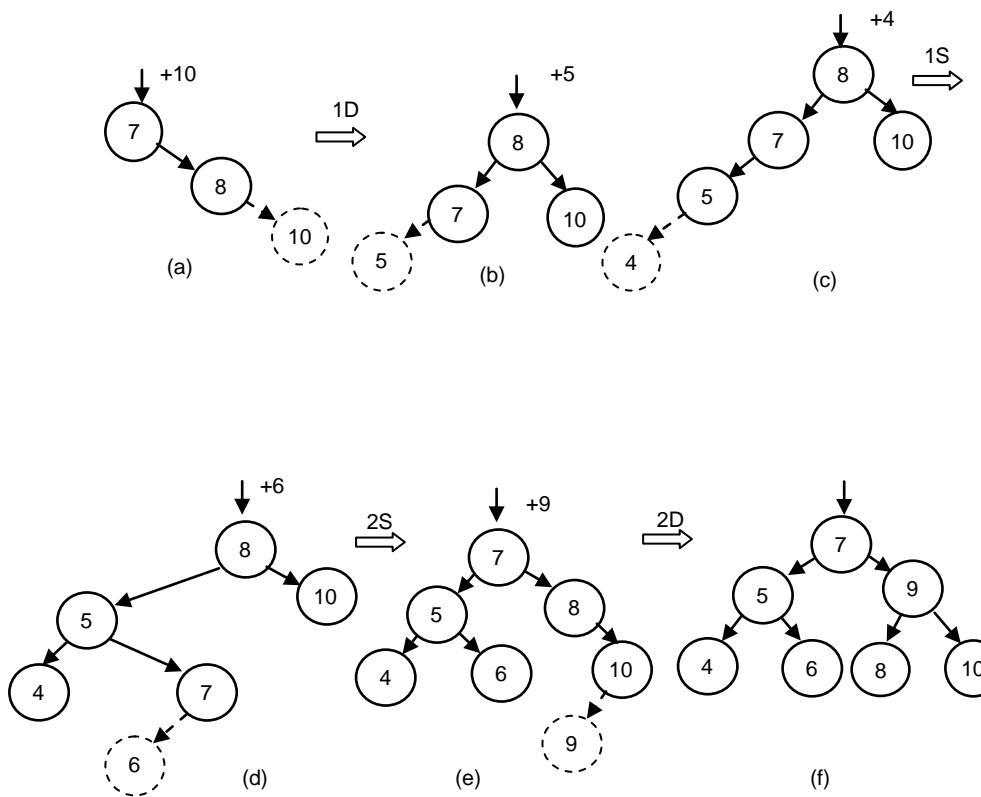
[8.5.3.b]

```

altfel
    daca(x>p->cheie) /*parcure subarbore drept*/
        InsertEchilibrat(x,p->drept,&h);
        daca(*h)/*ramura dreapta a crescut in
            inaltime*/
            daca(p->ech== -1) /*cazul ech=-1*/
                p->ech=0; *h=FALSE
                revenire;
                □ /*daca*/
            daca(p->ech== 0) /*cazul ech=-0*/
                p->ech+=1;
                revenire;
                □ /*daca*/
            daca(p->ech== 1) /*cazul ech=1*/
                /*reechilibrare dreapta*/
                p1=p->drept;
                daca(p1->ech== +1) THEN
                    /*cazul 1 dreapta*/
                    p->drept=p1->stang;
                    p1->stang=p;
                    p->ech=0; p=p1;
                    □ /*daca*/
                altfel
                    /*cazul 2 dreapta*/
                    p2=p1->stang;
                    p1->stang=p2->drept;
                    p2->drept=p1;
                    p->drept=p2->stang;
                    p2->stang=p;
                    daca(p2->ech== +1)
                        p->ech=-1;
                    altfel
                        p->ech=0;
                    daca(p2^.ech== -1)
                        p1->ech=+1;
                    altfel
                        p1->^.ech=0;
                        p=p2; [8.5.3.b]
                        □ /*altfel*/
                    p->ech=0; *h=FALSE; revenire;
                    □ /*daca*/
                    □ /*daca*/
                □ /*daca*/
            altfel
                /*s-a gasit cheia, incrementare contor*/
                p->contor=p->contor+1;
                revenire;
                □ /*altfel*/
/*InsertEchilibrat*/
-----
```

- Procedura **InsertEchilibrat** funcționează după cum urmează:
  - (1) Inițial se parcurge arborele indicat de referință  $p$  pe stânga respectiv pe dreapta după valoarea cheii  $x$  care se caută. Parcurgerea se realizează prin apeluri recursive ale procedurii **InsertEchilibrat**.
  - (2) Dacă se ajunge la o referință  $p=NULL$  are loc inserția, cu modificarea lui  $h=TRUE$  specificând astfel că înălțimea subarborelui a crescut.

- (3) După o astfel de inserție se revine din apelul recursiv și se verifică echilibrul nodului curent realizându-se eventual echilibrarea pe **stânga** (dacă se revine din stânga) sau pe **dreapta** (dacă se revine din dreapta).
- (4) Dacă se găsește o cheie egală cu  $x$  se incrementează contorul nodului în cauză.
- (5) Cu privire la variabila  $h$  se fac următoarele precizări:
  - Inserția îl poziționează pe  $h=TRUE$ .
  - Revenirile prin noduri cu factorul de echilibru 0 nu îl modifică pe  $h$ .
  - Reechilibrarea îl poziționează pe  $h=FALSE$ .
- Pentru exemplificare se consideră succesiunea de inserții într-un arbore AVL precizată în figura 8.5.3.i.



**Fig.8.3.5.i.** Inserții succesive într-un arbore echilibrat AVL.

- Se consideră arborele echilibrat AVL (a).
- Inserția cheii 10 conduce la un arbore dezechilibrat (cazul 1 Dreapta), a cărui echilibrare perfectă se realizează printr-o rotație simplă dreapta, fig.8.5.3.i (b).
- Inserțiile nodurilor 5 și 4 conduc ladezechilibrarea subarborelui cu rădăcina 7. Echilibrarea sa se realizează printr-o rotație simplă (cazul 1 Stânga) (d).
- Inserția în continuare a cheii 6 produce din noudezechilibrarea arborelui, a cărui echilibrare se realizează printr-o rotație dublă stânga rezultând arborele (e) (cazul 2 Stânga).

- În sfârșit, inserția nodului 9 conduce la cazul 2 Dreapta, care necesită în vedere echilibrării arborelui cu rădăcina 8 o rotație dublă care conduce la arborele echilibrat AVL (f).
- În legătură cu **performanțele inserției într-un arbore echilibrat AVL** se ridică două probleme:
  1. Dacă toate cele  $n!$  permutări de  $n$  chei apar cu **probabilitate egală**, care este **înălțimea probabilă** a arborelui echilibrat care se construiește?
  2. Care este **probabilitatea** ca o inserție să necesite **reechilibrarea** arborelui?
- Analiza matematică a acestui complicat algoritm este încă o problemă parțial rezolvată.
- Teste empirice ale înălțimii arborilor generați de algoritmul [8.5.3.b.] conduc la valoarea  $h = \log(n) + c$ , unde  $c$  este o constantă mică ( $c \approx 0.25$ ).
- Aceasta înseamnă că în practică, **arborii echilibrați AVL**, se comportă **la fel de bine** ca și **arborii perfect echilibrați**, fiind însă mai ușor de realizat.
- **Testele empirice** sugerează de asemenea că în medie, **reechilibrarea** este necesară aproximativ la fiecare **două inserții**.
  - Atât rotațiile simple cât și cele duble sunt echiprobabile.
- Complexitatea operației de reechilibrare sugerează faptul că arborii echilibrați trebuie utilizati de regulă când operațiile de căutare a informației sunt mult mai frecvente decât cele de inserare.

#### **8.5.4. Suprimarea nodurilor în arbori echilibrați AVL**

- În cazul arborilor echilibrați AVL, **suprimarea** este o operație mai **complicată** decât inserția.
- În principiu însă, operația de **reechilibrare** rămâne aceeași, reducându-se la una sau două **rotații** la stânga sau la dreapta.
- **Tehnica** care stă la baza suprimării nodurilor în arbori echilibrați AVL este similară celei utilizate în cazul **arborilor binari ordonați** prezentată în &8.3.5.
  - Cazul evident este cel în care, nodul care se suprimă este un **nod terminal** sau are **un singur descendenter**.
  - Dacă nodul de suprimat are însă **doi descendenți**, el va fi înlocuit cu cel mai din dreapta nod al subarborelui său stâng (predecesorul său la parcurgerea în inordine).
  - Ca și în cazul inserției, se utilizează **variabila booleană h** a cărei valoare adevărat semnifică **reducerea înălțimii subarborelui**.
  - **Reechilibrarea** se execută **numai** când **h** este adevărat.
  - Variabila **h** se poziționează pe adevărat după suprimarea unui nod al structurii, sau dacă reechilibrarea însăși reduce înălțimea subarborelui.
  - Tehnica suprimării nodurilor din arbori echilibrați AVL este materializată de procedura **SuprimEchilibrat** secvența [8.5.4.a]

---

/\***Suprimarea unui nod într-un arbore echilibrat AVL - varianta pseudocod\*/**

```

tip_ref_nod_AVL q; /*variabila globala*/
boolean h; /*variabila globala*/

Subprogram Echilibrul(ref_tip_nod_AVL p, boolean *h)
/*echilibrează subarborele stâng după o suprimare*/

    ref_tip_nod_AVL p1,p2;
    int e1,e2;

    /*h=adevărat, ramura stânga a devenit mai mică*/
    dacă (p->ech== -1)
        p->ech=0; /*cazul ech=-1*/
        revenire;
        □ /*dacă*/
    dacă (p->ech== 0) /*cazul ech=0*/
        p->ech+=1; *h=FALSE;
        revenire;
        □ /*dacă*/
    dacă (p->ech== +1) /*reechilibrare subarbore stang*/
        p1=p->drept; e1=p1->ech;
        dacă (e1>=0)
            /*cazul 1 dreapta - rotație simplă*/
            p->drept=p1->stang; p1->stang=p;
            dacă (e1==0)
                p->ech+=1; p1->ech=-1;
                *h=FALSE;
                □ /*dacă*/
            altfel
                p->ech=0;
                p1->ech=0;
                □ /*altfel*/
        p=p1;
        □ /*dacă*/
    altfel
        /*cazul 2 dreapta - rotație dublă*/
        p2=p1->stang; e2=p2->ech;
        p1->stang=p2^.drept; p2->drept=p1;
        p->drept=p2->stang;
        p2->stang=p;
        dacă (e2== +1)
            p->ech=-1;
            altfel
                p->ech=0;
        dacă (e2== -1)
            p1->ech+=1;
            altfel
                p1->ech=0;
            p=p2; p2->ech=0;
            □ /*altfel*/
        revenire; /*revenire cazul reeechilibrare stânga*/
        □ /*dacă*/
/*Echilibrul*/

```

**Subprogram Echilibru2**(tip\_ref\_nod\_AVL p, boolean \*h);  
/\*echilibrează subarborele drept după o suprimare\*/

tip\_ref\_nod\_AVL p1,p2;  
int e1,e2;

[8.5.4.a]

```

/*h=adevarat, ramura dreapta a devenit mai mică*/
dacă(p->ech==+1)
| p->ech=0; /*cazul ech=+1*/
| revenire;
| | /*dacă*/
dacă (p->ech==0) /*cazul ech=0*/
| p->ech=-1; *h=FALSE
| revenire;
| | /*dacă*/
dacă (p->ech==−1) /*reechilibrare subarbore drept*/
| p1=p->stang; e1=p1^.ech;
| dacă(e1<=0)
| | /*cazul 1 stânga - rotație simplă*/
| | p->stang=p1->drept; p1->drept=p;
| | dacă(e1==0)
| | | p->ech=−1; p1->ech=+1;
| | | *h=FALSE;
| | | | /*dacă*/
| | | altfel
| | | | p->ech=0;
| | | | p1->ech=0;
| | | | | /*altfel*/
| | | p=p1
| | | | /*dacă*/
| | | altfel
| | | | /*cazul 2 stânga - rotație dublă*/
| | | | p2=p1->drept; e2=p2->ech;
| | | | p1->drept=p2->stang; p2->stang=p1;
| | | | p->stang=p2->drept;
| | | | p2->drept=p;
| | | dacă(e2==−1)
| | | | p->ech=+1;
| | | | altfel
| | | | | p->ech=0;
| | | | dacă(e2==+1)
| | | | | p1->ech=−1;
| | | | | altfel
| | | | | | p1->ech=0;
| | | | | p=p2; p2->ech=0;
| | | | | | /*altfel*/
| | | | revenire; /*revenire cazul reeechilibrare dreapta*/
| | | | | /*dacă*/
| | | | /*Echilibru2*/

```

```

Subprogram Suprima(tip_ref_nod_AVL r, boolean *h)
/*suprimă cel mai din dreapta nod al arborelui indicat de r
 și mută conținutul său în nodul q*/
*h=FALSE;
dacă(r->drept!=NULL)
| Suprima(r->drept,&h);
| dacă(*h) Echilibru2(r,&h);
| | /*dacă*/
altfel
| | q->cheie=r->cheie;
| | q->contor=r->contor;
| | r=r->stang; *h=TRUE;
| | | /*altfel*/

```

[8.5.4.a]

```

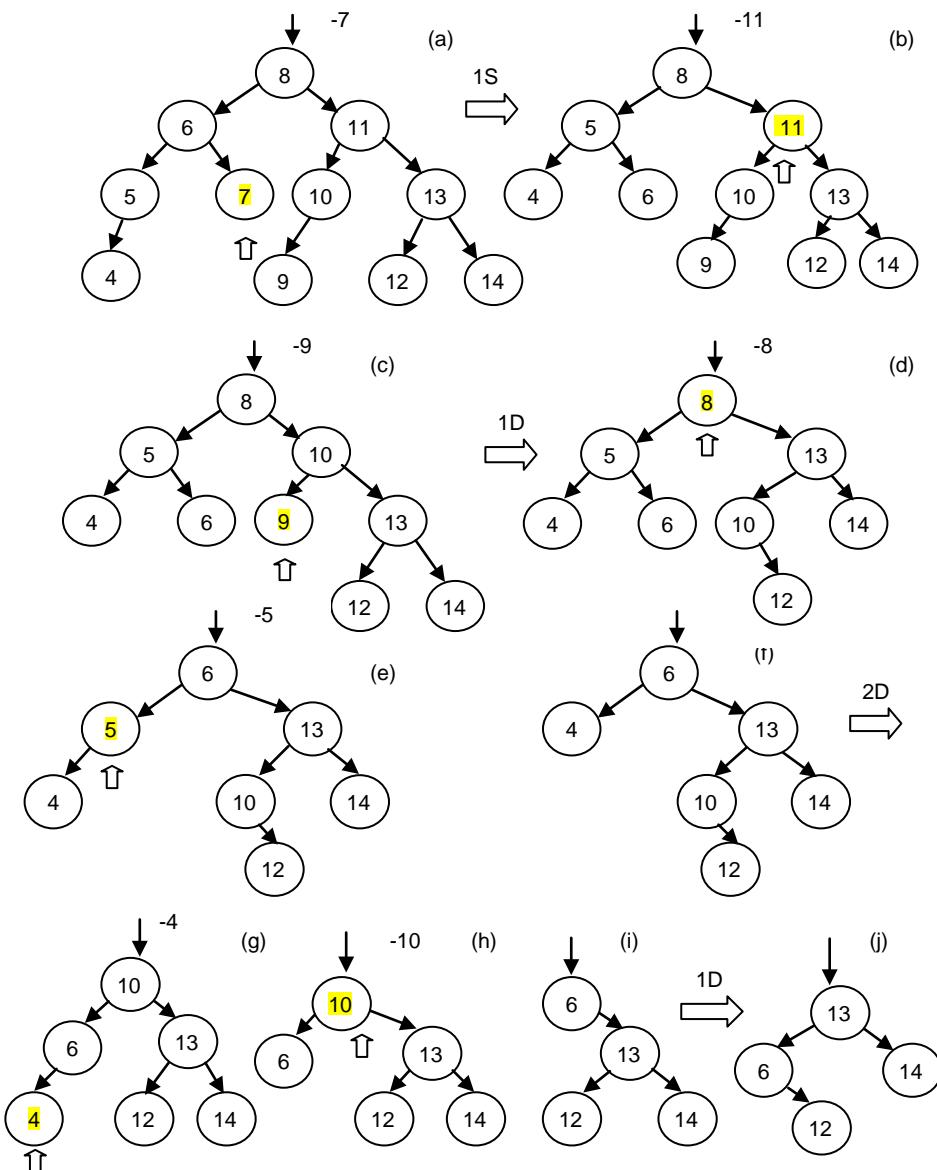
revenire;
/*Suprima*/

Subprogram SuprimaEchilibrat(int x, tip_ref_nod_AVL p,
boolean *h)
/*caută și suprimă nodul cu cheia x din arborele referit de
p. Dacă arborele își modifică înălțimea se returnează
h=TRUE*/
daca(p==NULL)
| *scrie('cheia nu e în arbore'); *h=FALSE;
| □ /*dacă*/
altfel
daca(x<p->cheie)
| SuprimaEchilibrat(x,p->stang,&h);
| daca(*h) Echilibru1(p,&h); /*echilibrare stânga*/
| □ /*dacă*/
altfel
daca(x>p->cheie)
| SuprimaEchilibrat(x,p->drept,&h);
| daca(*h) Echilibru2(p,&h); /*echilibrare
dreapta*/
| □ /*dacă*/
altfel
| /*nod gasit, suprimare nod p*/
q=p;
daca(q->drept==NULL) /*[8.5.4.a]*/
| p=q->stang; *h=TRUE;
| □ /*dacă*/
altfel
daca(q->stang==NULL)
| p=q->drept; *h=TRUE;
| □ /*dacă*/
altfel /*suprimare predecesor*/
Suprima(q^.stang,&h);
daca(*h) Echilibru1(p,&h);
□ /*altfel*/
/*elibereaza_memoria(q)*/
□ /*altfel*/
revenire;
/*SuprimaEchilibrat*/

```

- 
- În cadrul procedurii **SuprimaEchilibrat** se definesc trei proceduri:
    - (1) **Echilibru1** care se aplică când **subarborele stâng** s-a redus din înălțime.
    - (2) **Echilibru2** care se aplică când **subarborele drept** s-a redus din înălțime.
    - (3) **Suprima** – are rolul procedurii **Supred** la arbori binari ordonați:
      - (3.1) Găsește și înlocuiește nodul de suprimit cu predecesorul său.
      - (3.2) Suprimă predecesorul.

- (3.3) În plus procedura **Suprima** realizează eventualele reechilibrări la revenirea recursivă pe drumul parcurs în arbore, apelând când este cazul, procedura **Echilibru2** (revine din dreapta întotdeauna).
- Mersul procedurii **SuprimEchilibrat** este normal:
  - (1) Se parurge recursiv arborele AVL pentru căutarea cheii de suprimat, prin apeluri ale procedurii **SuprimaEchilibrat** pe stânga sau pe dreapta după cum cheia care se caută e mai mică respectiv mai mare decât cea a nodului curent.
  - (2) Când se găsește cheia ea se suprimă exact ca și la arborii binari ordonați:
    - Cazul 1 fiu: se rezolvă prin suprimare directă.
    - Cazul 2 fiu: se apelează procedura **Suprima** descrisă mai sus.
  - (3) Este important de reamintit faptul că după fiecare revenire dintr-un apel recursiv se verifică valoarea lui  $h$  și dacă este necesar se apelează procedura corespunzătoare de echilibrare.
- Modul de lucru al procedurii, este prezentat în figura 8.5.4.a.



**Fig.8.5.4.a.** Suprimări succesive în arbori echilibrați AVL

- Dându-se arborele binar echilibrat (a), se suprimă în mod succesiv nodurile având cheile  $7, 11, 9, 8, 5, 4$  și  $10$ , rezultând arborii (b)...(j).
  - Suprimarea cheii  $7$  este simplă însă conduce la subarborele dezechilibrat cu rădăcina  $6$ . Reechilibrarea acestuia presupune o rotație simplă (cazul 1 stânga).
  - Suprimarea nodului  $11$  nu ridică probleme.
  - Reechilibrarea devine din nou necesară după suprimarea nodului  $9$ ; de data aceasta, subarborele având rădăcina  $10$ , este reechilibrat printr-o rotație simplă dreapta (cazul 1 dreapta).
  - Suprimarea cheii  $8$  este imediată.
  - Deși nodul  $5$  are un singur descendant, suprimarea sa presupune o reechilibrare mai complicată bazată pe o dublă rotație (cazul 2 dreapta).
  - Ultimul caz, cel al suprimării nodului cu cheia  $10$  presupune înainte de reechilibrare, înlocuirea acestuia cu cel mai din dreapta element al arborelui său stâng (nodul cu cheia  $6$ ).
- În cazul **arborilor binari echilibrați**, suprimarea unui nod se realizează în cel mai defavorabil caz cu performanța  $O(\log_2 n)$ .
- Diferența esențială dintre **inserție** și **suprimare** în cazul arborilor echilibrați AVL este următoarea:
  - În urma unei **inserții**, reechilibrarea se realizează **o singură dată** prin una sau două rotații (a două sau trei noduri).
  - În cel mai defavorabil caz, **suprimarea** poate necesita o **reechilibrare a fiecărui nod** situat pe drumul de căutare.
    - Spre exemplu, în cazul arborelui **Fibonacci**, suprimarea nodului său cel mai din dreapta, necesită numărul maxim de rotații, acesta fiind cazul cel mai **defavorabil** de suprimare dintr-un arbore echilibrat.
- În realitate, testele experimentale indică faptul suprinzător că:
  - (1) În cazul **inserției** reechilibrarea devine necesară aproximativ la fiecare **a doua** inserție.
  - (2) În cazul **suprimării** reechilibrarea devine necesară aproximativ la fiecare **a 5-a** suprimare.

## 8.6. Arbori binari optimi

### 8.6.1. Definirea arborilor binari optimi

- Până în acest moment, organizarea arborilor binari ordonați s-a bazat pe presupunerea că frecvențele de acces sunt **egale** pentru toate nodurile structurii, cu alte cuvinte toate cheile au **aceeași** probabilitate de apariție.
- Această presupunere este motivată de faptul că de regulă **lipsesc** informații referitoare la **distribuția acceselor** la cheile structurii.
- Există însă situații în care se pot cunoaște **probabilitățile de acces ale diferitelor chei**.
- Un exemplu în acest sens îl constituie activitatea prin care un **compilator** stabilește dacă un anumit **identificator** este sau nu cuvânt **cheie** (rezervat).
  - **Măsurători statistice** efectuate pe loturi semnificative de programe permit obținerea de informații relativ exacte referitoare la **frecvențele de apariție** și în consecință la **frecvențele de acces la cuvintele cheie** ale unui limbaj.
- Structura **arbore binar ordonat** corespunzătoare unor astfel de **cuvinte cheie** este **constantă** (fixă) încât cuvintele cheie **nu** se modifică (**nu** se inserează și **nu** se suprimă).
- Se notează cu  $p_i$  **probabilitatea de acces** la nodul  $i$  care are cheia  $k_i$  și aparține unei structuri arbore binar.  $p_i$  se definește conform formulei [8.6.1.a]:

$$P\{x = k_i\} = p_i; \quad \sum_{i=1}^n p_i = 1 \quad [8.6.1.a]$$

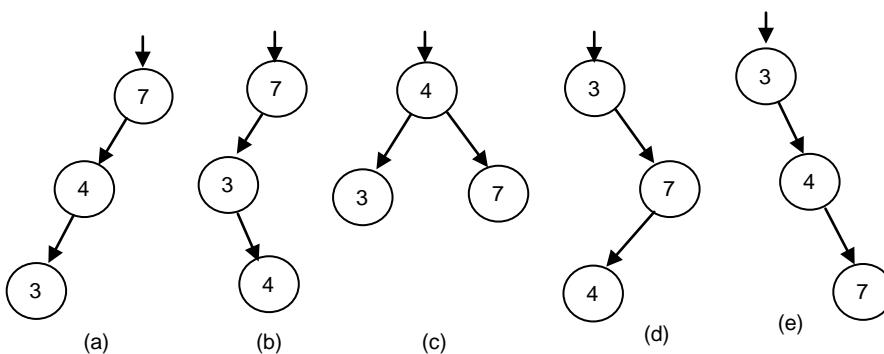
- **Problema** care se pune se referă la **organizarea** de o asemenea manieră a **arborelui binar ordonat**, încât **numărul total al pașilor de căutare** contorizat pentru un număr suficient de încercări, să devină **minim**.
- În acest scop, **lungimea drumului de căutare**, definit în &8.1.1 se modifică prin atribuirea unei **ponderi** (greutăți) fiecărui nod.
  - Nodurile la care accesul se face mai frecvent devin noduri cu **pondere mai mare**, cele vizitate mai rar, noduri cu **pondere mai mică**.
  - **Ponderea** unui nod se asimilează cu **probabilitatea de acces** la acel nod.
- Se definește astfel noțiunea de **drum ponderat** asociat unui **arbore binar ordonat**.

- Lungimea  $P_I$  a drumului ponderat asociat unui **arbore binar** este suma lungimilor tuturor drumurilor de la rădăcină la fiecare nod al arborelui, fiecare drum fiind **ponderat** cu probabilitatea de acces la nodul respectiv [8.6.1.b].

$$P_I = \sum_{i=1}^n p_i * h_i$$

[ 8 . 6 . 1 . b ]

- Se precizează că  $h_i$  este **nivelul** nodului  $i$ .
- În continuare se urmărește **minimizarea lungimii drumului ponderat** pentru o **distribuție de probabilități data**.
  - Spre exemplu se consideră setul de chei 3,4,7 având probabilitățile de acces  $p_1=1/7$ ,  $p_2=2/7$  și  $p_3=4/7$ .
  - Aceste chei pot fi aranjate ca și arbori binari ordonați în 6 moduri dintre care 5 sunt diferite (fig.8.6.1.a).



**Fig.8.6.1.a.** Arbori binari cu trei noduri

- Lungimile drumurilor ponderate calculate conform relației [8.6.1.b] sunt [8.6.1.c]:

$$P_I^{(a)} = \frac{1}{7}(1*3 + 2*2 + 4*1) = \frac{11}{7}$$

$$P_I^{(b)} = \frac{1}{7}(1*2 + 2*3 + 4*1) = \frac{12}{7}$$

$$P_I^{(c)} = \frac{1}{7}(1*2 + 2*1 + 4*2) = \frac{12}{7}$$

[ 8 . 6 . 1 . c ]

$$P_I^{(d)} = \frac{1}{7}(1*1 + 2*3 + 4*2) = \frac{15}{7}$$

$$P_I^{(e)} = \frac{1}{7}(1*1 + 2*2 + 4*3) = \frac{17}{7}$$

- După cum se observă, în acest exemplu **aranjamentul optim nu** este cel corespunzător arborelui binar perfect echilibrat, ci arborelui degenerat în listă liniară (a).

- Exemplul anterior referitor la activitatea **compilatorului** poate fi reluat într-un **caz mai general** și anume:
  - Cuvintele preluate din textul sursă **nu** sunt întotdeauna **cuvinte cheie**.
    - De fapt acest lucru este mai degrabă o excepție.
    - Faptul că un **cuvânt oarecare**  $k_i$  nu este un **cuvânt rezervat**, respectiv cheia sa **nu** se găsește în **arborele binar al cuvintelor cheie**, poate fi considerat drept un acces **ipotecic** la un nod "special" inserat între **cheia** imediat mai **mică** și cea imediat mai **mare**.
    - Acestui **nod special** i se asociază o **lungime a drumului extern**.
    - Se presupune că se cunosc **probabilitățile de apariție**  $q_i$  ale unor astfel de cuvinte  $x$  care se situează cu valoarea între două chei  $k_i$  și  $k_{i+1}$ .
      - Structura **arborelui binar ordonat** poate fi considerabil **modificată**, prin luarea în considerare și a **căutărilor nereușite**, respectiv a căutărilor care **nu** se referă la **cuvintele cheie** ale limbajului.
      - În acest caz, **lungimea drumului ponderat total**  $P$  pentru un astfel de arbore se definește cu ajutorul formulei [8.6.1.d].
 

---


$$P = \sum_{i=1}^n p_i * h_i + \sum_{j=0}^m q_j * h'_j$$

unde

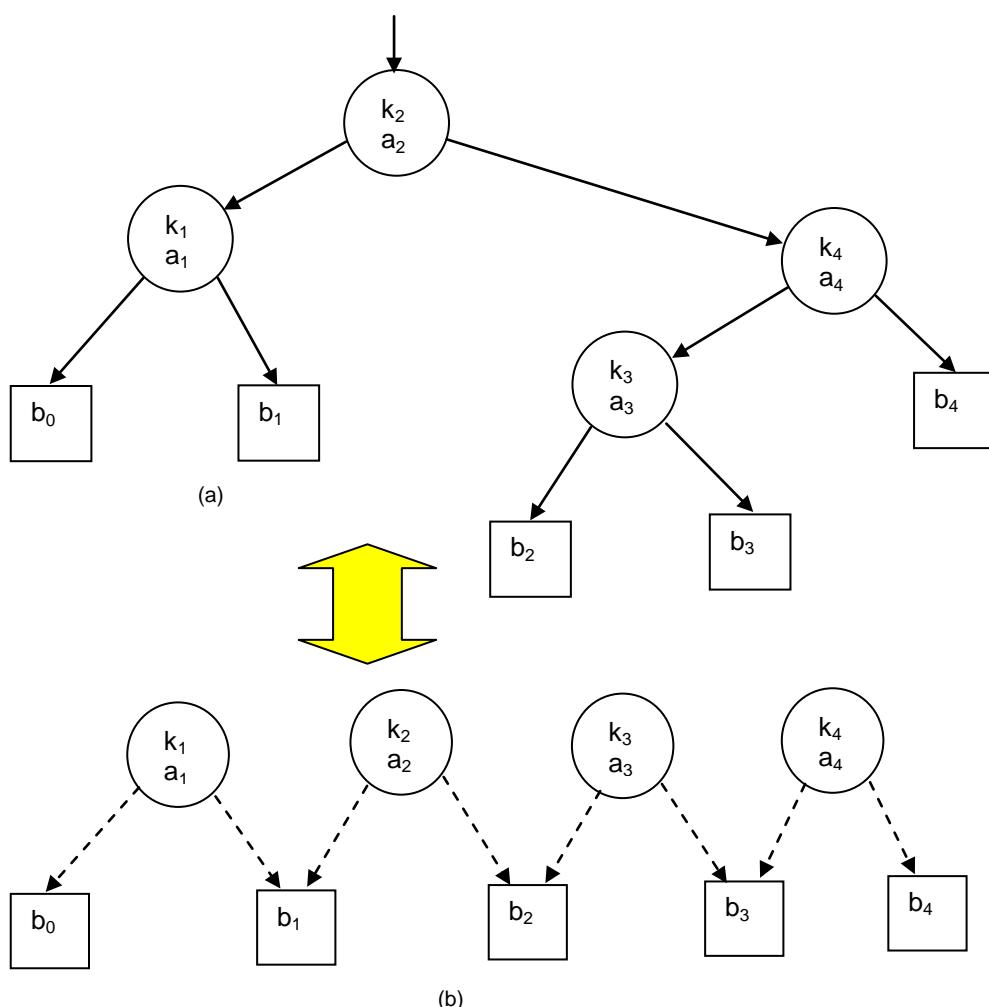
$$\sum_{i=1}^n p_i + \sum_{j=0}^m q_j = 1$$


---
- Se face precizarea că:
  - $h_i$  este nivelul (adâncimea) **nodului intern**  $i$ .
  - $h'_j$  este nivelul (adâncimea) **nodului extern**  $j$ .
  - $n$  este numărul de **noduri interne**.
  - $m$  este numărul de **noduri externe**.
- Lungimea **drumului ponderat total** se mai numește și **cost al arborelui binar ordonat**, întrucât ea reprezintă o măsură a **efortului așteptat** a fi depus în procesul de căutare.
- Se consideră **toate** structurile **arbore binar ordonat** care pot fi construite pornind de la setul de chei  $k_i$ , având probabilitățile asociate  $p_i$  și  $q_j$ .
- Se numește **arbore binar optim**, arborele a cărui structură conduce la un **cost minim**.

- În activitatea practică, **probabilitățile** pot fi substituite cu **contoare de frecvență**, care memorează **numărul de acces** la un nod.

- Astfel, se notează cu:

- $a_i$  = numărul care precizează de câte ori  $x$  este egal cu cheia  $k_i$ .
- $b_j$  = numărul care precizează de câte ori  $x$  este cuprins între cheile  $k_j$  și  $k_{j+1}$ .
- Prin **convenție** se consideră că:
  - $b_0$  este numărul care precizează de câte ori  $x$  este găsit ca fiind mai mic decât  $k_1$ , adică cea mai mică cheie.
  - $b_n$  precizează de câte ori  $x$  a fost găsit ca fiind mai mare decât  $k_n$ , adică cea mai mare cheie (fig.8.6.1.b).



**Fig.8.6.1.b.** Arbore binar optim cu patru chei cu frecvențe de acces asociate (a), reprezentarea schematică a arborelui  $A_{04}$  (b)

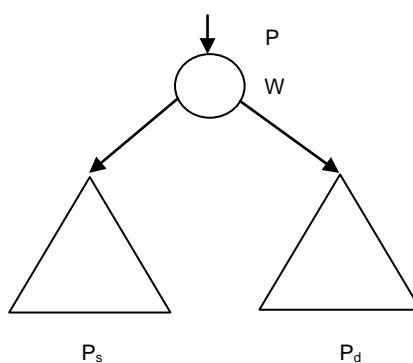
- Pentru calculul **lungimii drumului ponderat total**  $P$  se va utiliza formula [8.6.1.e]:

$$P = \sum_{i=1}^n a_i * h_i + \sum_{j=0}^n b_j * h_j \quad [8.6.1.e]$$

- Astfel pe lângă faptul că se utilizează **frecvențe** în loc de **probabilități**, apare și avantajul exprimării lui P prin **valori întregi**.

### 8.6.2. Construcția arborilor binari optimi

- Construcția arborilor binari optimi **nu** este o treabă simplă.
- Luând în considerare faptul că numărul **configurațiilor posibile** de arbori cu n noduri crește **exponențial** cu valoarea lui n, aflarea **arborelui binar optim** pentru valori mari ale lui n pare foarte complicată.
- **Arborii binari optimi** se bucură însă de **proprietatea** foarte importantă că:
  - **Toți subarborii** unui **arbore binar optim** sunt de asemenea **optimi**.
- Această proprietate sugerează un **algoritm** care **construiește** în mod **sistemtic** arbori binari optimi din ce în ce mai mari, pornind de la nodurile individuale care sunt considerate drept cei mai mici subarbori.
- Arborii cresc astfel de la frunze spre rădăcină, conform metodei "**bottom-up**" ("de jos în sus").
- **Ecuația** care este cheia acestui algoritm este [8.6.1.f].
  - Se notează cu P lungimea drumului ponderat total al arborelui.
  - Se notează cu  $P_s$  respectiv  $P_d$  lungimile drumurilor ponderate corespunzătoare subarborilor stâng respectiv drept ai rădăcinii (fig.8.6.1.c).



**Fig. 8.6.1.c.** Arbore binar optim

- P poate fi calculat ca fiind egal cu suma dintre  $P_s$ ,  $P_d$  și numărul W care precizează de câte ori se trece prin ramul inițial al rădăcinii.
- W se numește **ponderea arborelui binar optim** și are valoarea egală cu **numărul total de căutări** efectuate în arbore, adică **suma** frecvențelor  $a_i$  și  $b_j$  pentru toate nodurile cuprinse în arbore [8.6.1.f], [8.6.1.g].

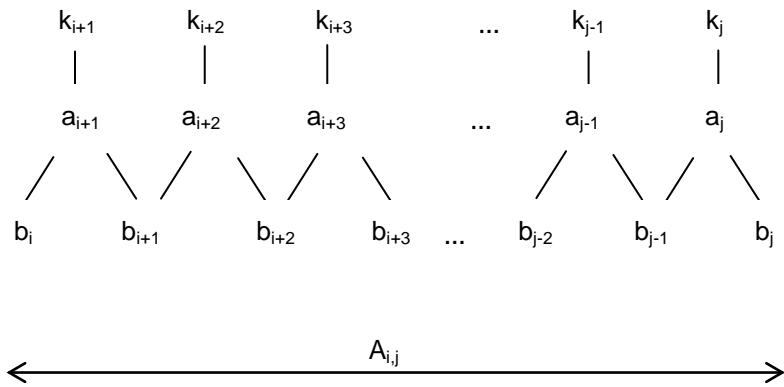
$$P = P_s + W + P_d$$

[ 8 . 6 . 1 . f ]

$$W = \sum_{i=1}^n a_i + \sum_{j=0}^n b_j$$

[ 8 . 6 . 1 . g ]

- **Media lungimii drumului ponderat** corespunzător acestui arbore este  $P/W$ .
- În continuare se va preciza **modul de notare** al **ponderilor și lungimilor drumurilor** corespunzătoare tuturor subarborilor care constau dintr-un număr de chei adiacente.
  - Fie  $A_{i,j}$  **arborele binar optim** reprezentat schematic în fig.8.6.1.d.
  - După cum se observă,  $A_{i,j}$  este definit prin:
    - Nodurile cu cheile adiacente  $k_{i+1}, k_{i+2}, \dots, k_j$
    - Frecvențele de acces la chei  $a_{i+1}, a_{i+2}, \dots, a_j$
    - Frecvențele de acces interchei  $b_i, b_{i+1}, \dots, b_j$



**Fig.8.6.1.d.** Reprezentarea schematică a arborelui binar optim  $A_{i,j}$

- Se face precizarea că în cadrul structurii arborelui, **proiecția** cheilor pe axa absciselor produce **secvența ordonată crescător** a acestora, indiferent de aranjamentul lor.
- Această proprietate derivă imediat din observația ca **arborii binari optimi** sunt de fapt la origine **arbori binari ordonați**.
- Se notează cu  $w_{i,j}$  **ponderea** și cu  $p_{i,j}$  **lungimea drumului total** al subarborelui binar optim  $A_{i,j}$ .
  - Conform celor deja precizate,  $w_{i,j}$  este suma frecvențelor  $a_i$  și  $b_j$  [8.6.1.g].
  - Atât  $w_{i,j}$  cât și  $p_{i,j}$  sunt definiți pentru  $0 \leq i \leq j \leq n$ .

- **Formulele recurente** de pentru calcul valorilor lui  $w_{ij}$  respectiv  $p_{ij}$  apar în [8.6.1.h] respectiv [8.6.1.i].

$$w_{ii} = b_i \quad \text{pentru } 0 \leq i \leq n \quad [8.6.1.h]$$

$$w_{ij} = w_{i,j-1} + a_j + b_j \quad \text{pentru } 0 \leq i < j \leq n$$

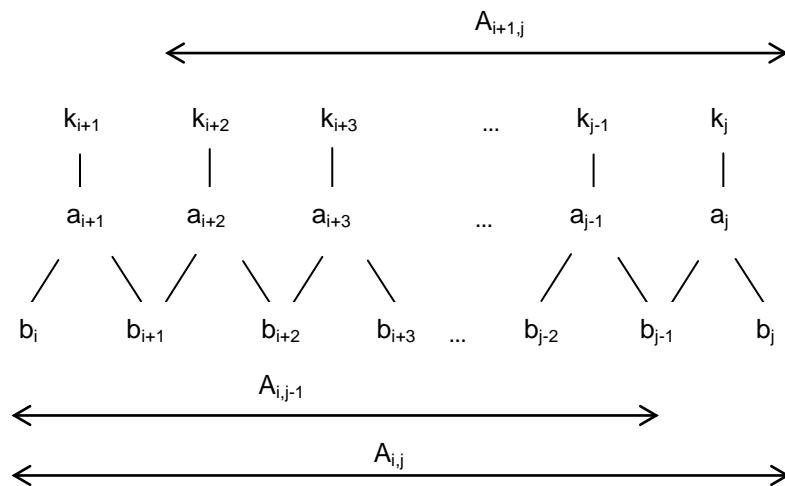
$$p_{ii} = w_{ii} = b_i \quad \text{pentru } 0 \leq i \leq n \quad [8.6.1.i]$$

$$p_{ij} = w_{ij} + \min(p_{i,k-1} + p_{kj}) \quad \text{pentru } 0 \leq i < j \leq n$$

- Ultima ecuație rezultă imediat din relația [8.6.1.f] și din **definiția optimalității**, adică:
  - **Arborele binar optim** are drept **rădăcină** acel nod având cheia cu indicele  $k$ , pentru care suma  $P_s$  și  $P_d$  este **minimă**.
  - De aici apare și **ideea** care poate conduce la **construcția arborelui optim**  $A_{ij}$  și anume:
    - Trebuie căutată între toate cheile cuprinse între  $k_{i+1}$  și  $k_j$  acea cheie cu indicele  $k$  care conduce la un arbore cu  $p_{ij}$  minim.
    - Stau la dispoziție de fapt  $j-i$  posibilități de a alege rădacina unui astfel de arbore.
    - Deoarece există  $(1/2)n^2$  valori ale lui  $p_{ij}$  pentru cele  $0 < j-i \leq n$  cazuri unde  $i \leq j$  (jumătatea superioară a matricei  $P$  care memorează lungimile drumurilor), operația de minimizare va necesita aproximativ  $(1/6)n^3$  operații [Kn76].
    - Aceasta înseamnă că un **arbore binar optim** poate fi determinat în  $O(n^3)$  unități de timp utilizând  $O(n^2)$  locații de memorie.
  - **Knuth** propune **reducerea timpului de execuție** cu un factor proporțional cu  $n$ , element care face posibilă utilizarea practică a acestui algoritm.
  - Se notează cu  $R$  mulțimea indicilor  $k$  ai nodurilor pentru care relația [8.6.1.i] atinge valoarea minimă.
  - Mulțimea  $R$  este structurată ca și o matrice denumită în continuare matricea  $R$ .
    - Un element oarecare  $r_{ij}$  unde ( $i < j$ ) al mulțimii  $R$  specifică de fapt indicele  $k$  al rădăcinii arborelui optim  $A_{ij}$ .
    - Evident în matricea  $R$  există câte o locație pentru **rădăcina** fiecărui arbore binar optim care poate fi construit plecând de la cheile date  $k_i$  și frecvențele de acces  $a_i$  respectiv  $b_j$ .
    - Căutarea lui  $r_{ij}$  care în mod normal trebuie efectuată pe domeniul  $j-i$  se poate reduce la un interval mult mai mic.

- **Observația** lui Knuth care permite acest lucru este următoarea:

- Se presupune că  $r_{ij}$  este o rădăcină a arborelui optim  $A_{ij}$ .
- Se subliniază faptul că  $r_{ij}$  este de fapt un indice de cheie a cărui valoare este cuprinsă între  $i$  și  $j$ .
- Este evident că rădăcina  $r_{ij}$  este cuprinsă:
  - Între rădăcina arborelui rezultat prin suprimarea celui mai din **dreapta** nod al arborelui  $A_{ij}$  adică arborele  $A_{i,j-1}$ .
  - Și rădăcina arborelui rezultat din suprimarea celui mai din **stânga** nod al său adică  $A_{i+1,j}$  aşa cum rezultă din figura 8.6.1.e.



**Fig.8.6.1.e.** Observația Knuth. Reprezentarea schematică a arborilor binari optimi  $A_{ij}$ ,  $A_{i,j-1}$ ,  $A_{i+1,j}$

- Această observație este o **consecință** a faptului că:
  - a) Adăugarea unui nod la **dreapta** arborelui produce (eventual) deplasarea spre **dreapta** a rădăcinii arborelui optim.
  - b) Suprimarea celui mai din **stânga** nod, produce (eventual) deplasarea tot spre **dreapta** a rădăcinii arborelui optim.
- Acest lucru se exprimă formal cu ajutorul relației [8.6.1.j]:

$$r_{i,j-1} \leq r_{ij} \leq r_{i+1,j}$$

[ 8 . 6 . 1 . j ]

- Această observație permite **limitarea procesului de căutare**:

- Soluțiile pentru  $r_{ij}$  trebuie căutate **numai** în domeniul  $r_{i,j-1}, r_{i+1,j}$ .

- Rezultă astfel un număr  $O(n^2)$  de pași elementari pentru construcția unui arbore binar optim.
- Cu alte cuvinte, pentru a determina **rădăcina arborelui binar optim**  $A_{ij}$ :
  - (1) Se verifică **toate** perechile de arbori  $A_{i,k-1}$  și  $A_{k,j}$  pentru  $k \in [r_{i,j-1}, r_{i+1,j}]$ .
  - (2) Se alege acel indice  $k$  pentru care suma  $p_{i,k-1} + p_{k,j}$  este minimă conform relației [8.6.1.i].
- În continuare se trece la descrierea **algoritmului de construcție** al arborelui binar optim  $A_{on}$  care conține  $n$  chei.
- Se reamintesc următoarele definiții care se referă la **arborele binar optim**  $A_{ij}$  constând din nodurile având cheile  $k_{i+1}, \dots, k_j$ :

$a_i$  : frecvența căutărilor cheii  $k_i$ .

$b_j$  : frecvența căutărilor unui argument  $x$  cuprins între cheile  $k_j$  și  $k_{j+1}$ .

$w_{ij}$  : ponderea arborelui binar optim  $A_{ij}$ .

$p_{ij}$  : lungimea drumului ponderat al arborelui binar optim  $A_{ij}$ .

$r_{ij}$  : indicele cheii rădăcinii arborelui binar optim  $A_{ij}$ .

- Se declară următoarele structuri de date [8.6.1.k]:

---

#### {Arborei optimi: Structuri specifice de date - C}

```

int a[Numar_Noduri]; /*tablou frecvențe chei*/ [8.6.1.k]
int b[Numar_Noduri]; /*tablou frecvențe interchei*/
int p[Numar_Noduri, Numar_Noduri]; /*matrice lungimi drumuri
                                      ponderate*/
int w[Numar_Noduri, Numar_Noduri]; /*matrice ponderi arbori
                                      optimi*/
int r[Numar_Noduri, Numar_Noduri]; /*matrice radacini arbori
                                      optimi*/

```

---

#### {Arborei optimi: Structuri specifice de date - PASCAL}

```

TYPE TipIndex=0..n;
VAR a: ARRAY[1..n] OF integer; [8.6.1.k]
    b: ARRAY[TipIndex] OF integer;
    p,w: ARRAY[TipIndex,TipIndex] OF integer;
    r: ARRAY[TipIndex,TipIndex] OF TipIndex;

```

---

- Se presupune că ponderile  $w_{ij}$  au fost calculate în mod direct utilizându-se valorile pentru frecvențele  $a$  și  $b$ , pe baza relațiilor [8.6.1.h] și memorate în matricea  $W$ .
- **Procedura de construcție:**

- Utilizează matricea W drept argument (dată de intrare).
- Construiește gradual matricea R, care memorează rădăcinile subarborilor binari optimi.
- Construiește simultan cu R, matricea P utilizată în procesul de construcție. Matricea P poate fi considerată un rezultat intermedian.
- Procedeul folosit este următorul:
  - (1) Se pornește cu cei mai mici subarbori posibili, respectiv cei care **nu** conțin nici un nod, adică arborii  $A_{ii}$  care au lățimea 0.
  - (2) Se construiesc succesiv subarbori cu lățimi din ce în ce mai mari: 1,2,3, etc.

• Se notează cu  $h$  lățimea  $j-i$  a subarborelui  $A_{ij}$ .

• Pentru toți arborii care au lățimea  $h=0$ ,  $p_{ii}$  poate fi determinat direct din relația [8.6.1.i] conform secvenței [8.6.1.l].

---

**/\*Construcția arborilor optimi de lățime  $h=0$ \*/**

```
pentru i=0 la n
  p[i,i]=w[i,i]; /*=b[i]*/

```

[ 8 . 6 . 1 . l ]

---

- În cazul lui  $h=1$ , avem de-a face cu arbori cu un singur nod, nod care în mod evident este și rădăcina arborelui.
- $i$  precizează limita **stânga** pentru index iar  $j$  limita **dreapta** în arborele considerat  $A_{ij}$  (secvența [8.6.1.m]).

---

**/\*Construcția arborilor optimi de lățime  $h=1$ \*/**

```
pentru i=0 la n-1
  j=i+1;
  | p[i,j]=p[i,i]+p[j,j]+w[i,j];
  | r[i,j]=j;
  | □

```

[ 8 . 6 . 1 . m ]

---

- Pentru cazurile în care lățimea  $h$  este mai mare ca 1, se utilizează o **secvență repetitivă** cu  $h$  luând valori succesive cuprinse între 2 și  $n$ .
- Cazul  $h=n$  presupune construcția arborelui  $A_{on}$ .
- În fiecare caz, lungimea drumului minim  $p_{ij}$  și indexul  $r_{ij}=k$  asociat rădăcinii se caută printr-o instrucție de ciclare simplă cu indicele  $k$  luând valori în intervalul  $[r_{i,j-1}, r_{i+1,j}]$  furnizat de relația [8.6.1.j].
- Secvența de construcție a arborilor binari optimi cu lățimea mai mare ca 1 apare în [8.6.1.n].

### {Construcția arborilor optimi de lățime h>1}

```

pentru (h=2 la n)
    pentru (i=0 la n-h) [8.6.1.n]
        j=i+h;
        *găsește acel indice m care conduce la o valoare
        minimă al lui min. min se determină cu relația
            min = minim(p[i,m-1]+p[m,j]), dintre toți indicii
            m care satisfac relația r[i,j-1]<=m<=r[i+1,j];
        p[i,j]=min+w[i,j];
        r[i,j]=m;
    □

```

- Detaliile rafinării acestei secvențe apar în programul **ReprezentareArbore** procedura **ArbOpt**, secvența [8.8.c].
- **Lungimea medie a drumului** pentru arborele  $A_{on}$  este furnizată de raportul  $p_{on}/w_{on}$  iar rădâcina sa este nodul având indicele  $r_{on}$ .
- În forma din secvența [8.6.1.n], algoritmul de construcție al arborelui binar optim  $A_{on}$  necesită un **efort de calcul** de ordinul  $O(n^2)$  și un **volum de memorie** de același ordin.
  - Aceste valori **nu** sunt în general acceptabile pentru valori foarte mari ale lui  $n$ .
- **Hu și Tacker** au dezvoltat un alt algoritm care necesită numai  $O(n)$  locații de memorie și  $O(n * \log n)$  efort de calcul.
  - Acest algoritm ia însă în considerare **numai** cazurile în care frecvențele de acces la chei sunt nule ( $a_i=0$ ), adică sunt înregistrate numai accesele interchei [Kn76].
- Un alt algoritm cu performanțe similare a fost descris de **Walker și Gotlieb**.
  - Acest algoritm care construiește un arbore **aproape optim**, poate fi implementat utilizînd **principii euristice**.
    - **Idea de bază** este următoarea:
      - (1) Se consideră toate nodurile (adevărate și speciale) ale structurii arbore, ponderate de frecvențele (probabilitățile) lor de acces, ca fiind distribuite pe o scară liniară.
      - (2) Se caută nodul care este cel mai apropiat de "centrul de greutate".
    - Acest nod se numește "**centroid**" și **indexul** său se calculează cu formula [8.6.1.o], valoarea obținută rotunjindu-se la cel mai apropiat întreg.

$$\frac{1}{w} \left( \sum_{i=1}^n i * a_i + \sum_{j=0}^n j * b_j \right)$$

[8.6.1.o]

- Dacă toate nodurile au aceeași pondere, atunci în mod evident rădăcina arborelui optim coincide cu **centroidul**.
- În marea majoritate a restului cazurilor, rădăcina se găsește în vecinătatea apropiată a centroidului, utilizându-se în acest sens o căutare limitată a unui optim local.
- (3) În continuare procedura este aplicată celor 2 arbori care au rezultat, apoi celor 4 și.m.d.
- (4) După ce s-a ajuns la arbori suficienți de mici se poate aplica metoda exactă descrisă anterior.
- Metoda care rezultă, conduce la arbori destul de avantajoși (în 2-3% din cazuri chiar la soluția optimă) necesitând  $O(n)$  unități de spațiu de memorare și  $O(n * \log n)$  unități de timp pentru execuție.

## 8.7. Arbori Huffman

### 8.7.1. Coduri prefix. Algoritmul lui Huffmann.

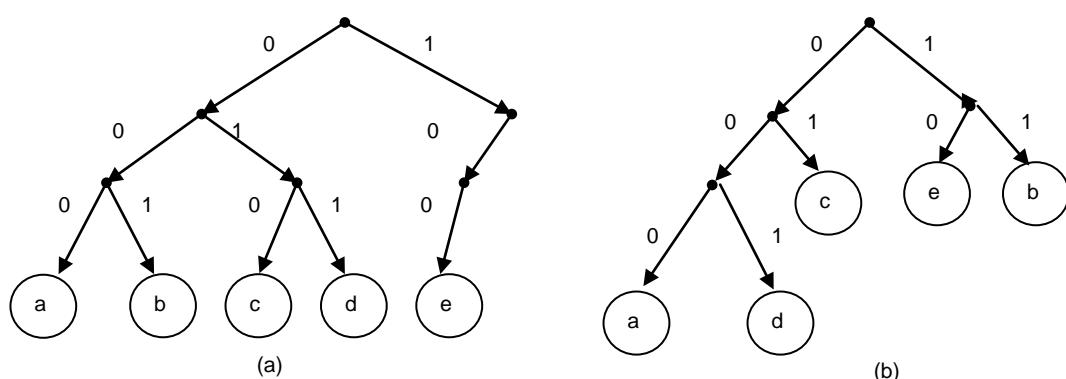
- Un exemplu de utilizare al **arborilor binari cu ponderi** ca structuri de date îl reprezintă **codurile Huffmann**.
- Se presupune că se prelucrează **mesaje** care constau din **secvențe de caractere**.
- În fiecare mesaj, caracterele sunt **independente** și pot apărea în **orice poziție** a mesajului.
- Se presupune, de asemenea, că se cunoaște **probabilitatea de apariție** a fiecărui caracter, probabilitate care **nu** depinde de poziția caracterului în cadrul mesajului.
  - Spre exemplu, se consideră mesaje formate din 5 caractere **a, b, c, d, e** care apar respectiv cu probabilitățile: 0.12, 0.4, 0.15, 0.08, 0.25.
  - Se dorește **a se codifica fiecare caracter** printr-o secvență de cifre binare "0" și "1", astfel încât codul unui caracter, să **nu** fie **prefix** pentru codul nici unui alt caracter.
  - Această proprietate numită și "**proprietatea de prefix**" permite **decodificarea** unui mesaj format din caractere "0" și "1" prin ștergerea repetată a prefixelor sirului care sunt coduri de caractere.
    - Spre exemplu, în fig.8.7.1.a se prezintă două codificări posibile ale simbolurilor anterior precizate.
    - Pentru codul 1, algoritmul de decodificare este simplu: se separă grupe de câte 3 biți care se decodifică conform tabelei (evident codurile 101, 110 și 111 nu există).
    - Astfel, mesajul **001010011** reprezintă sirul original bcd.

Simbol	Probabilitate	Cod 1	Cod 2
a	0.12	000	000
b	0.40	001	11
c	0.15	010	01
d	0.08	011	001
e	0.25	100	10

Fig.8.7.1.a. Codificare binară

- Se poate ușor verifica faptul că și codul 2 are **proprietatea de prefix**.
- Diferența față de cazul precedent este aceea că separarea grupelor de cifre nu se poate face dintr-o dată deoarece lungimea grupelor este variabilă (2 sau 3).
  - Astfel secvența **1101001** reprezintă tot secvența bcd.
- **Problema** care se pune, se **formulează** în felul următor:

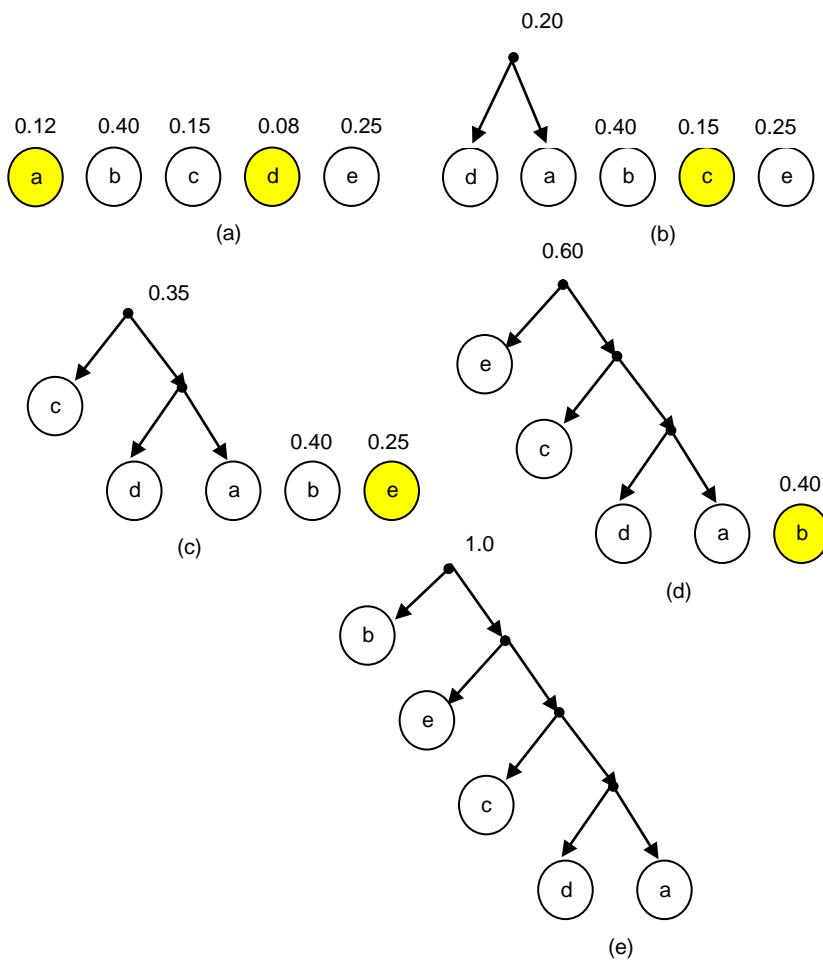
- Se dă un **set de caractere** precum și **probabilitățile lor de apariție**.
- Se cere să se construiască un **cod** cu **proprietatea de prefix**, astfel încât lungimea medie a codului pentru un caracter să fie **minimă**.
- Este evident faptul că un astfel de cod va conduce la **mesaje de lungime minimă**.
- Una din tehniciile de afilare a **codului de prefix optim** este **algoritmul lui Huffmann**.
- Conform algoritmului lui **Huffmann**:
  - (1) Se selectează două caractere a și b care au probabilitatea cea mai scăzută de apariție.
  - (2) Se înlocuiesc cele două caractere cu un singur caracter imaginar (spre exemplu x), a cărui probabilitate de apariție este suma probabilităților lui a și b.
  - (3) Aplicând iterativ această procedură, se obține **codul prefix optimal** pentru un set de caractere.
  - (4) Codul pentru setul original de caractere se obține utilizând codul lui x la întâlnirea caracterului a sau b, căruia i se adaugă un "0" pentru a, respectiv un "1" pentru b.
- În această manieră, un **cod prefix** poate fi reprezentat printr-un **arbore binar**, dacă **nodurile terminale** ale arborelui li se asociază caracterele originale ale alfabetului, iar ramurilor arborelui ponderile 0 sau 1.
- **Codul unui caracter** poate fi asimilat cu un drum în **arborele binar** respectiv.
- Toate drumurile pornesc de la rădăcină și se finalizează cu un nod terminal.
- Se consideră că trecerea la **fiul stâng** al unui nod adaugă un "0" codului, iar trecerea la **fiul drept**, se adaugă un "1".
- **Sevența de cifre binare** rezultată, reprezintă **codul** caracterului respectiv.
- În figura 8.7.1.b apare reprezentarea în forma de **arbore binar** a codului 1 (a) respectiv a codului 2 (b) din figura 8.7.1.a.



**Fig.8.7.1.b.** Coduri prefix în reprezentare de arbore binar

### 8.7.2. Arbori Huffmann. Implementarea algoritmului lui Huffmann

- În implementarea algoritmului lui **Huffmann** se utilizează o colecție de **arbori binari speciali** care se bucură de următoarele caracteristici:
  - **Nodurile terminale** sunt caractere.
  - **Rădăcina** oricărui arbore are asociată o valoare care reprezintă suma probabilităților de apariție a caracterelor corespunzătoare tuturor nodurilor terminale ale arborelui respectiv.
    - Această sumă se numește **greutate** sau **pondere** a arborelui.
- Inițial **fiecare caracter** este un arbore format dintr-un singur nod.
- La terminarea algoritmului rezultă **un singur arbore** ale căruia noduri terminale conțin toate caracterele alfabetului.
- În acest arbore, **drumul** de la rădăcină la orice nod terminal reprezintă codul caracterului asociat nodului, conform schemei stânga egal "0", dreapta egal "1".
  - Astfel de arbori se numesc **arbori Huffmann**.
- **Construcția unui arbore Huffmann** se realizează după cum urmează:
  - (1) În fiecare pas, algoritmul lui Huffmann selectează din colecție doi arbori cu cea mai mică greutate.
  - (2) Cei doi arbori sunt combinații într-unul singur, a cărui greutate este egală cu suma greutăților celor doi arbori.
    - Combinarea arborilor se realizează, generând o nouă rădăcină care are drept fii rădăcinile arborilor în cauză (ordinea lor **nu contează**).
  - (3) Continuând în aceeași manieră, în final se obține arboarele, care pentru probabilitățile date, reprezintă codul cu lungimea medie minimă.
- În fig.8.7.2.a apare reprezentarea grafică a construcției unui astfel de arbore Huffmann.



**Fig.8.7.2.a.** Construcția unui arbore Huffmann

- În continuare se descriu **structurile de date specifice**.
- Pentru reprezentarea arborilor și implementarea algoritmului lui Huffmann se utilizează trei tablouri: arbore, alfabet și zona (secvență [8.7.2.a]).

---

**/\*Arborei Huffmann Structuri de date - varianta C\*/**

```

typedef struct linie_arbore {
    int fiu_stang;      /*cursor în arbore*/
    int fiu_drept;     /*cursor în arbore*/
    int parinte;        /*cursor în arbore*/
} linie_arbore;

linie_arbore arbore[MaxNod1]; /*tabloul arbore*/
int ultim_nod;                /*cursor în arbore*/

typedef struct linie_alfabet {
    char simbol;           [8.7.2.a]
    float probabilitate;
    int terminal;          /*cursor în arbore*/
} linie_alfabet;

linie_alfabet alfabet[MaxNod2]; /*tabloul alfabet*/

typedef struct linie_zona {

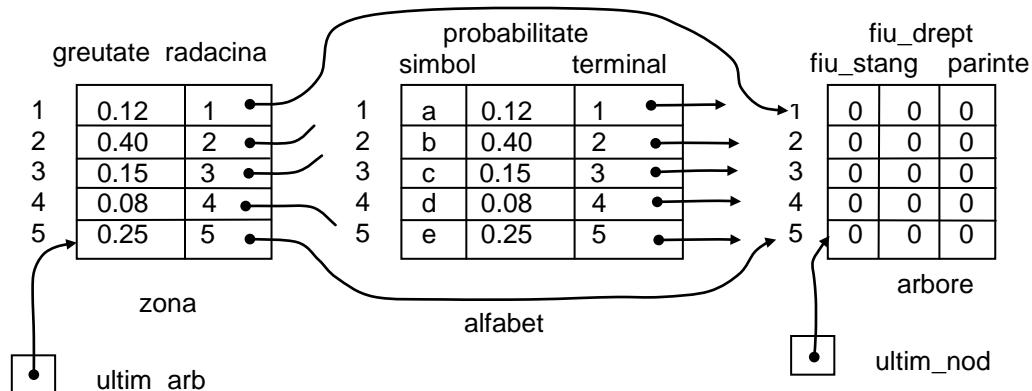
```

```

        float greutate;
        int radacina;      /*cursor în arbore*/
    } linie_zona;
linie_zona zona[MaxNod3];      /*tabloul zona*/
int ultim_arb;                 /*cursor în zona*/
-----

```

- Tabloul arbore, memorează **structurile arborilor**
  - Fiecare element al tabloului arbore se referă la un nod al unui arbore pentru care se păstrează cursorii la fiul său stâng, la cel drept și la părintele său.
  - Variabila de tip indice (cursor) ultim\_nod, indică ultimul element ocupat al tabloului.
  - Câmpul parinte a fost prevăzut pentru a facilita parcurgerea drumului de la un nod terminal spre rădăcină în vederea determinării codului unui
- **Simbolurile și probabilitățile lor de apariție** se înregistrează în tabloul alfabet.
  - Pentru fiecare simbol se prevede cursorul care indică nodul terminal asociat din tabloul arbore.
- Se mai utilizează și tabloul zona pentru precizarea **colecției de arbori Huffmann**.
  - Fiecare înregistrare a tabloului zona se referă la un arbore și cuprinde greutatea arborelui și un cursor la rădăcina sa din tabloul arbore.
  - Și în acest caz se utilizează variabila de tip indice (cursor) ultimArb care indică ultimul element ocupat al tabloului zona.
- Valorile inițiale ale câmpurilor celor trei tablouri pentru exemplul anterior apar în figura 8.7.2.b.



**Fig.8.7.2.b.** Valori inițiale pentru structurile de date utilizate

- Se presupune că înregistrările de la 1 la ultim\_arb din zona și cele de la 1 la ultim\_nod din arbore sunt integral ocupate.
- În secvențele care urmează sunt omise comparațiile între valorile cursorilor și limitele maxime ale tablourilor utilizate.
- O primă formă pseudocod a algoritmului lui Huffmann apare în secvența [8.7.2.b].

```
/*Algoritmul lui Huffmann - varianta pseudocod*/
```

```
cat timp (există mai mult de un arbore în zona) [8.7.2.b]
    i=indexul în zona al arborelui cu greutatea minimă;
    j=indexul în zona al arborelui cu greutatea minimă
        următoare;
    *crează în tabloul arbore un nod nou care are ca fiu
        stâng pe zona[i].radacina și ca fiu drept pe
        zona[j].radacina;
    *înlocuiește arborele i din zona cu arborele
        a cărui radacină este nodul nou creat și a cărui
        greutate este zona[i].greutate+zona[j].greutate;
    *suprimă arborele j din zona;
    □
```

- Pentru **rafinarea** acestei forme se descriu:

- (1) Procedura **GreutateMinima** care determină indicii i și j din tabloul zona ai arborilor cu cele mai mici greutăți [8.7.2.c].

```
/*Determinarea arborilor minimi - varianta pseudocod*/
```

```
Subprogram GreutateMinima(int min,int min1)
```

```
/*poziționează pe min și min1 pe arborii cei mai ușori din
   zona. Se presupune ca în zona există cel puțin 2 arbori*/
int i; /*indice cautare*/
daca(zona[1].greutate<=zona[2].greutate)
    min=1;
    min1=2;
    □
altfel
    min=2;
    min1=1;
    □
pentru(i=3 la ultim_arb)
    daca(zona[i].gretate<zona[min].greutate)
        min1=min;
        min=i;
        □
    altfel
        daca(zona[i].greutate<zona[min1].greutate)
            min1=i;
    □
/*GreutateMinima*/
```

[8.7.2.c]

- (2) Functia **Creeaza** care generează un nou arbore in tabloul arbore [8.7.2.d].

```
/*Generarea unui nou arbore in tabloul arbore - varianta C*/
```

```
int Creeaza(int arb_stang,int arb_drept) [8.7.2.d]
    ultim_nod= ultim_nod+1; /*arborele nou creat este
                               arbore[ultim_nod]*/
    arbore[ultim_nod].fiu_stang=zona[arb_stang].radacina;
```

```

arbore[ultim_nod].fiu_drept=zona[arb_drept].radacina;
arbore[ultim_nod].parinte=0;
arbore[zona[arb_stang].radacina].parinte=ultim_nod;
arbore[zona[arb_drept].radacina].parinte=ultim_nod;
returneaza ultim_nod;
/*Creeaza*/
-----
```

- Procedura **Huffman** care apare în secvența [8.7.2.e]

- Procedura **Huffman** nu are parametri de intrare sau de ieșire, ea operând asupra unor structuri de date globale.

```
/*Construcția unui arbore Huffman - varianta pseudocod*/
```

**Subprogram Huffman;**

```

int i,j; /*indicatori pentru arborii cei mai mici*/
int radnou; /*rădaciana arborelui nou creat*/

cat timp(ultim_arb>1)
  GreutateMinima(i,j); [8.7.2.e]
  radnou=Creeaza(i,j);
  /*se înlocuiește arborele i din zona cu arborele
   a cărui rădacina este radnou*/
  zona[i].greutate=zona[i].greutate+zona[j].greutate;
  zona[i].radacina=radnou;
  /*se suprimă arborele j din zona înlocuindu-l cu
   ultimul arbore*/
  zona[j].greutate=zona[ultim_arb].greutate;
  zona[j].radacina=zona[ultim_arb].radacina;
  ultim_arb = ultim_arb-1 /*actualizare ultim_arb*/
  □
/*Huffman*/
```

- După terminarea execuției procedurii **Huffman**, **codul pentru fiecare simbol** se determină după cum urmează:

- (1) Se caută simbolul în tabloul alfabet.
  - Câmpul terminal al înregistrării corespunzătoare simbolului, este cursorul înregistrării din tabloul arbore care corespunde nodului terminal asociat simbolului.
- (2) În continuare, în mod repetat, se determină **părintele** p al nodului curent n, până se ajunge la rădăcina arborelui.
  - În acest scop se utilizează câmpul **parinte** a cărui valoare este un cursor tot în tabloul arbore.
- (3) Pentru fiecare părinte p, se verifică dacă nodul curent n este fiul său din stânga sau cel din dreapta, memorându-se un "0" respectiv un "1".
- (4) **Secvența finală de cifre binare** rezultată reprezintă **codul simbolului** în **ordine inversă**.

## 8.8. Reprezentarea grafică a structurilor arbore

- În cadrul acestui paragraf se abordează problema **reprezentării grafice** a unei **structuri arbore** pe ecranul unui display în **mod caracter**.
  - Maniera de afişare este **secvențială** și **discretă**.
    - Se afișează siruri de caractere numai de la stânga la dreapta și de sus în jos, rând după rând (mod ecran, 25 de rânduri, fiecare a câte 80 de caractere).
  - În acest scop:
    - (1) Într-o primă etapă se generează **structura topologică a arborelui** care se dorește a fi reprezentată.
    - (2) Într-o a doua etapă această structură se transformă într-una reprezentabilă prin afişare în maniera mai sus precizată, determinând coordonatele efective ale nodurilor și ale conexiunilor care le unesc în aşezarea lor pe ecran.
  - Pentru prima etapă, soluția cea mai potrivită de **generare a unei structuri arbore** este cea bazată pe un **algoritm recursiv**.
  - Se va utiliza drept suport al reprezentării **structura arbore binar optim** definită în &8.6.
  - În acest acop se redactează funcția **Arbore(i, j:index):TipRef** (secvența [8.8.c]),
    - Funcția **Arbore**:
      - (1) Pornește de la matricea **r** care conține rădăcinile subarborelor binari optimi.
      - (2) Generează **structura de date** corespunzătoare arborelui optim **A<sub>ij</sub>**.
      - (3) Returnează referința la rădăcina **arborelui binar optim** construit.
        - Parametrii **i** și **j** sunt cei care precizează indicii limită ai nodurilor **arborelui binar optim** a cărui rădăcină este nodul având cheia **k** memorată în **r[i, j]**.
    - Se definesc următoarele **tipuri de date** [8.8.a].

---

{Reprezentarea grafică a structurilor arbore. Structuri de date}

```
TYPE TipRef=^TipNod;
TipNod=RECORD
    cheie:string;
    poz:pozLin;
    stang,drept,leg:TipRef
END;
```

---

- Câmpurile **poz** și **leg** sunt prevăzute pentru scopuri care vor fi discutate ulterior.

- După cum s-a precizat, funcția **Arbore** are drept punct de pornire matricea  $r$  care memorează **indicii cheilor radăcinilor subarborelor binari optimi**.
- **Ideea** algoritmului de generare este următoarea:
  - Pentru arborele binar având rădăcina  $k$  (valoarea  $r[i, j]$ ), funcția **Arbore** generează **recursiv** subarborele său stâng având drept rădăcină cheia  $k_s$  (valoarea  $r[i, k-1]$ ) respectiv subarborele drept având drept rădăcină cheia  $k_d$  (valoarea  $r[k, j]$ ).
  - Funcția realizează de fapt o **traversare în inordine** a nodurilor arborelui binar optim care se construiește, în spate a arborelui  $A_{on}$ .
    - În consecință cheile nodurilor vor fi parcuse în **ordine alfabetică**.
  - Funcția **Arbore** contorizează numărul nodurilor generate în variabila globală  $k$ .
    - Cel de-al  $k$ -lea nod generat, este atribuit celei de-a  $k$ -a chei.
  - Deoarece **numărul total de chei** este cunoscut și **cheile sunt ordonate alfabetic**, coordonata orizontală  $poz$  a fiecarei chei în cadrul liniei de ecran pe care va fi afișată, se poate determina simplu înmulțind pe  $k$  cu valoarea unui **factor de scară** (dimensiunea în caractere a rândului/număr total de chei).
    - Această coordonată se memorează în câmpul  $poz$  al fiecărui nod pe măsură ce nodurile sunt create (vezi secvența [8.8.a]).
  - Se precizează de asemenea faptul că:
    - Cuvintele cheie sunt de tip **șir de caractere** (string).
    - Cuvintele cheie sunt memorate în tabloul predefinit de caractere **chei**, în ordine alfabetică, indicele de intrare în tabel fiind chiar valoarea  $k$ .
- Afisarea arborelui este realizată de către procedura **AfiseazaArbore** (secvența [8.8.c]).
- Procedura **AfiseazaArbore** utilizează drept **intrări**:
  - Setul de  $n$  cuvinte cheie memorate în tabloul **chei**.
  - Structura arbore binar optim generată de funcția **Arbore(0, n)**.
- În etapa întâi, se generează **structura preliminară a arborelui de reprezentat** (secvența [8.8.b]) în care:
  - (1) Coordonata orizontală a fiecărui nod (cheie) a arborelui de afișat este înregistrată în câmpul  $poz$ .
  - (2) Coordonata verticală se va determina **implicit** în momentul afișării, funcție de nivelul nodului în cadrul structurii arborelui.

---

**{Generarea unei structuri de arbore binar optim}**

---

`k:=0; radacina:=Arbore(0, n);`

[ 8 . 8 . b ]

---

- În continuare se poate aborda etapa a doua și anume, **afișarea arborelui** pe ecranul monitorului.
- Acest lucru se realizează plecând de la rădăcină în jos, prin prelucrarea în fiecare pas a unui rând (nivel) de noduri al arborelui.
  - Se realizează de fapt o parcurgere **prin cuprindere** a arborelui realizată prin **metoda celor două cozi**.
- Pentru a realiza accesul la nodurile unui rând (nivel) al structurii de arbore, se utilizează câmpul **leg** precizat în secvența [8.8.a].
  - Nodurile care trebuie afișate în rândul curent sunt înălțuite prin câmpul **leg** în lista **current**. Lista **current** este **prima coadă** utilizată în traversare.
  - Pe parcursul prelucrării nodurilor listei **current**, se identifică descendenții fiecărui nod și se alcătuiește cu aceștia o a doua listă numită **urm**, înălțuirile realizându-se tot prin câmpul **leg**. Lista **urm** este cea de-a **două coadă**.
  - Când se trece la nivelul următor coada (lista) **urm** devine coada **current** și se initializează noua coadă (listă) **urm**.
  - Așa cum s-a precizat, listele **current** respectiv **urm** sunt de fapt **structuri de date coadă**, iar **parcurgerea prin cuprindere** este realizată în baza tehnicii prezentate la &8.2.5.2.
- **Detaliile** de implementare ale algoritmului de afișare apar în secvența [8.8.c].
- Se impun următoarele precizări:
  1. Lista **urm** care conține nodurile nivelui următor al structurii de arbore, se generează prin tehnica inserției în față, nodul cel mai din stânga devenind astfel ultimul nod al listei.
  2. În vederea parcurgerii, lista trebuie **inversată**, activitate care se realizează în momentul în care lista **urm** devine lista **current**.
  3. Pentru fiecare cheie (nod) din lista **current** care urmează a fi afișată, se determină și se afișează **conexiunile** pe stânga și pe dreapta în forma unor segmente orizontale.
  4. Variabilele **u1, u2, u3** și **u4** precizează pozițiile de început și de sfârșit ale segmentelor din stânga, respectiv din dreapta unui nod, reprezentate ca o succesiune de caractere "linie orizontală".
  5. Afișarea nodurilor din linia **current** este precedată de afișarea pentru fiecare nod a unei linii verticale formate din trei segmente care marchează legătura dintre niveluri.
- Programul **ReprezentareArbore** [8.8.c] este destinat prelucrării unor texte sursă Pascal.
- În cadrul structurii programului se definesc următoarele proceduri și funcții:
  - Funcția **DrumArbEch(i, j : index) : integer;**
    - Generează în manieră recursivă matricea **r** corespunzătoare **arborelui perfect echilibrat** care poate fi construit utilizând cele **n** chei date și returnează **lungimea drumului** acestui arbore.
    - Arborele este precizat prin indicii **i** și **j** ai nodurilor sale extreme.
    - Se utilizează următorul **procedeu**: întrucât tabloul **chei** este un tablou ordonat, pentru fiecare apel al funcției, se alege pe post de rădăcină a arborelui **current**, indicele **k** al **cheii mediane** a intervalului delimitat de

indicii  $i$  și  $j$ , indice care se memorează în  $r[i, j]$ .

- În continuare, se determină lungimea drumului prin apelul recursiv al funcției **DrumArbEch** pentru subarborele stâng, respectiv pentru cel drept, după care se adaugă ponderea  $w[i, j]$  conform formulei [8.6.1.f].
- Procedura **ArbOpt** - construiește **arborele binar optim** pornind de la distribuția  $w$  a ponderilor nodurilor.
  - De fapt procedura completează matricea  $r$  cu indicii cheilor care reprezintă rădăcinile subarborelor optimi și matricea  $p$  cu lungimile corespunzătoare ale drumurilor asociate.
- Procedura **AfiseazaArbore** realizează afișarea efectivă a structurii arborelui și are drept parametri de intrare indicii  $i$  și  $j$  care delimită arborele de afișat.
  - În cadrul procedurii **AfiseazaArbore** se definește funcția **Arbore**, utilizată la generarea structurii arborelui ce urmează a fi afișat pornind de la matricea  $r$  corespunzătoare.
  - Ambele proceduri au fost descrise anterior.
- Programul principal.
- Mersul **programului principal** este următorul.
  - (1) În prima parte a programului principal:
    - Se inițializează tabelul cuvintelor cheie și contoarele memorate în tablourile  $a$  și  $b$ .
    - Se citește textul sursă (un program Pascal) de la dispozitivul de intrare.
    - Pe măsură ce este citit textul, sunt recunoscuți identificatorii și cuvintele cheie, actualizându-se contoarele de frecvență  $a_i$  și  $b_j$  (bucla **repeat**).
    - $a_i$  se referă la cuvintele cheie  $k_i$  iar  $b_j$  la identificatorii situați între  $k_j$  și  $k_{j+1}$ .
    - În continuare, pentru fiecare cuvânt cheie se afișează frecvențele de acces  $b_{i-1}$  și  $a_i$ .
    - Se afișează de asemenea o statistică a acestora, respectiv suma frecvențelor de acces pentru  $a_i$  și  $b_j$ .
  - (2) În cea de-a doua parte a programului:
    - Pornind de la frecvențele de acces se calculează matricea  $w$  a ponderilor.
  - (3) În cea de-a treia parte a programului:
    - Se apelează funcția **DrumArbEch** care construiește matricea  $r$  memorând rădăcinile subarborelor corespunzătoare **arborelui binar perfect echilibrat** cu limitele 0 și  $n$ , căruia îi calculează și lungimea drumului. Este vorba despre arborele binar perfect echilibrat al **cuvintelor cheie**.
    - După aceasta se tipărește lungimea medie a drumului ponderat și se afișează **arborele binar perfect echilibrat** prin apelul procedurii **AfiseazaArbore**.
  - (4) În cea de-a patra parte a programului:
    - Se apelează procedura **ArbOpt** care generează **arborele binar optim** pornind de la matricea  $w$ .

- Se calculează și se afișează pentru acest arbore lungimea medie a drumului după care este afișat arborele prin apelul procedurii **AfiseazaArbore**.
- (5) În final:
  - Se recalculează matricea  $w$  luând în considerare de această dată numai frecvențele de acces la chei  $a_i$ , cu alte cuvinte frecvențele  $b_j$  se consideră egale cu 0.
  - Se apelează din nou procedura **ArbOpt** pentru a determina arborele binar optim pentru această situație.
  - Se afișează alura arborelui astfel determinat prin apelul procedurii de afișare **AfiseazaArbore**.
- Experimentele arată că de regulă în acest context:
  - (1) Arborele **perfect echilibrat** nu poate fi considerat nici pe departe optim.
  - (2) Frecvențele identificatorilor care nu sunt cuvinte cheie influențează în mod decisiv structura arborelui optim care se construiește.

---

**PROGRAM ReprezentareArbore;**

```

CONST n=31; {numar chei}
           lch=10; { lungime maximă cheie}

TYPE index=0..n;
       alfa=string[lch];

VAR ch:char;
      k1,k2:integer;
      id:alfa; {identificator sau cheie} [8.8.c]
      chei:ARRAY[1..n] OF alfa;
      i,j,k:integer;
      a:ARRAY[1..n] OF integer;
      b:ARRAY[index] OF integer;
      p,w:ARRAY[index,index] OF integer;
      r:ARRAY[index,index] OF index;
      suma,sumb:integer;
      litere:SET OF char;
      cifre:SET OF char;

FUNCTION DrumArbEch(i,j:index):integer;
{Generează în manieră recursivă matricea r corespunzătoare
arborelui perfect echilibrat care poate fi construit
utilizând cele n chei date și returnează lungimea drumului
acestui arbore}

VAR k:integer;
BEGIN
  k:=(i+j+1) DIV 2;
  r[i,j]:=k; {k este cheia mediană}
  IF i>=j THEN
    DrumArbEch:=b[k]
  ELSE
    DrumArbEch:= DrumArbEch(i,k-1)+ DrumArbEch(k,j)
               +w[i,j]
  END; {DrumArbEch}

```

```

PROCEDURE ArbOpt;
{Construiește arborele binar optim pornind de la matricea w a
ponderilor nodurilor. De fapt se generează matricea r cu
indicii cheilor care reprezintă rădăcinile subarborelor
optimi și matricea p cu lungimile corespunzătoare ale
drumurilor asociate}

VAR x,min:integer;
i,j,k,h,m:index;
BEGIN {intrare:w; iesire:p,r}
{construcție arbori de lățime nulă(h=0)}
FOR i:=0 TO n DO
p[i,i]:=w[i,i];
{construcție arbori de lățime arbore=1 (h=1)}
FOR i:=0 TO n-1 DO
BEGIN
j:=i+1;
p[i,j]:=p[i,i]+p[j,j]+w[i,j];
r[i,j]:=j
END;
{construcție arbori de lățime arbore>1}
FOR h:=2 TO n DO {h=lățimea arborelui considerat}
FOR i:=0 TO n-h DO {i=indexul stâng al arborelui}
BEGIN {j=indexul sau drept}
j:=i+h;
m:=r[i,j-1]; min:=p[i,m-1]+p[m,j];
FOR k:=m+1 TO r[i+1,j] DO
BEGIN
x:=p[i,k-1]+p[k,j];
IF x<min THEN
BEGIN
m:=k;
min:=x
END
END;
p[i,j]:=min+w[i,j]; {pondere arbore optim}
r[i,j]:=m {rădăcină arbore optim}
END
END; {ArbOpt}

PROCEDURE AfiseazaArbore;
{Realizează afişarea efectivă a structurii arborelui și are
drept parametri de intrare indicii i și j care delimitizează
arborele de afișat}

CONST ll=80; {lățime linie de afișat}

TYPE ref=^nod;
pozLin=0..ll;
nod=RECORD
cheie:alfa;
poz:pozLin;
sting,drept,leg:ref
END;

VAR radacina,curent,urm:ref;
q,q1,q2:ref;

```

```

i,k:integer;
u,u1,u2,u3,u4:pozLin;

FUNCTION Arbore(i,j:index):ref;
{Generează structura de date corespunzătoare arborelui
optim. Pornește de la matricea r care conține rădăcinile
subarborilor binari optimi și returnează referința la
rădăcina arborelui optim construit}

VAR p:ref;
BEGIN
  IF i=j THEN p:=NIL
  ELSE
    BEGIN
      new(p);
      p^.sting:=Arbore(i,r[i,j]-1);
      p^.poz:=(((l1-lch)*k) DIV (n-1)) + (lch DIV 2);
      k:=k+1;
      p^.cheie:=chei [r[i,j]];
      p^.drept:=Arbore(r[i,j],j)
    END;
    Arbore:=p
  END; {Arbore}

BEGIN {AfiseazaArbore}
  k:=0; radacina:=Arbore(0,n);
  curent:=radacina;
  radacina^.leg:=NIL;
  urm:=NIL;
  WHILE curent<>NIL DO
    BEGIN
      {se afișează liniile verticale de legătură între
      niveluri pentru toate cuvintele din linia curentă}
      FOR i:=1 TO 3 DO
        BEGIN
          u:=0; q:=curent;
          REPEAT
            u1:=q^.poz;
            REPEAT
              Write(' '); u:=u+1
            UNTIL u=u1;
            Write('I'); u:=u+1; q:=q^.leg
            UNTIL q= NIL;
            WriteLn
        END;
      {se afișează linia curentă; se determină descendenții
      nodurilor din lista curent și se formează lista
      rândului următor urm}
      q:=curent; u:=0;
      REPEAT
        i:=lch;
        WHILE q^.cheie[i]=' ' DO i:=i-1; {lungime cheie}
        u2:=q^.poz-((i-1) DIV 2); u3:=u2+i;
        q1:=q^.sting; q2:=q^.drept;
        IF q1=NIL THEN
          u1:=u2
        ELSE
          BEGIN

```

```

        u1:=q1^.poz; q1^.leg:=urm; urm:=q1
    END;
IF q2=NIL THEN
    u4:=u3
ELSE
    BEGIN
        u4:=q2^.poz+1; q2^.leg:=urm; urm:=q2
    END;
    i:=0;
WHILE u<u1 DO BEGIN Write(' '); u:=u+1 END;
WHILE u<u2 DO BEGIN Write('-'); u:=u+1 END;
WHILE u<u3 DO
    BEGIN
        i:=i+1; Write(q^.cheie[i]); u:=u+1
    END;
WHILE u<u4 DO BEGIN Write('-'); u:=u+1 END;
    q:=q^.leg
UNTIL q=NIL;
WriteLn;
{se inversează lista urm și se face curentă}
curent:=NIL;
WHILE urm<>NIL DO
    BEGIN
        q:=urm; urm:=q^.leg;
        q^.leg:=curent; curent:=q
    END
END {WHILE}
END; {AfiseazaArbore}

```

[ 8.8.c ]

```

BEGIN {Programul principal - ReprezentareArbore}
{se initializează static tabela de chei}
chei[ 1]:='ARRAY' ; chei[ 2]:='BEGIN' ;
chei[ 3]:='CASE' ; chei[ 4]:='CONST' ;
chei[ 5]:='DIV' ; chei[ 6]:='DOWNTO' ;
chei[ 7]:='DO' ; chei[ 8]:='ELSE' ;
chei[ 9]:='END' ; chei[10]:='FILE' ;
chei[11]:='FOR' ; chei[12]:='FUNCTION' ;
chei[13]:='GOTO' ; chei[14]:='IF' ;
chei[15]:='IN' ; chei[16]:='LABEL' ;
chei[17]:='MOD' ; chei[18]:='NIL' ;
chei[19]:='OF' ; chei[20]:='PROCEDURE' ;
chei[21]:='PROGRAM' ; chei[22]:='RECORD' ;
chei[23]:='REPEAT' ; chei[24]:='SET' ;
chei[25]:='THEN' ; chei[26]:='TO' ;
chei[27]:='TYPE' ; chei[28]:='UNTIL' ;
chei[29]:='VAR' ; chei[30]:='WHILE' ;
chei[31]:='WITH' ;
FOR i:=1 TO n DO {se initializează contoarele a și b}
    BEGIN
        a[i]:=0; b[i]:=0
    END;
b[0]:=0; k2:=lch;
litere:=['a'..'z', 'A'..'Z'];
cifre:=['0'..'9'];
{se balează textul de intrare, se identifică cheile și
  identifierii și se determină a și b}
REPEAT
    Read(ch);

```

```

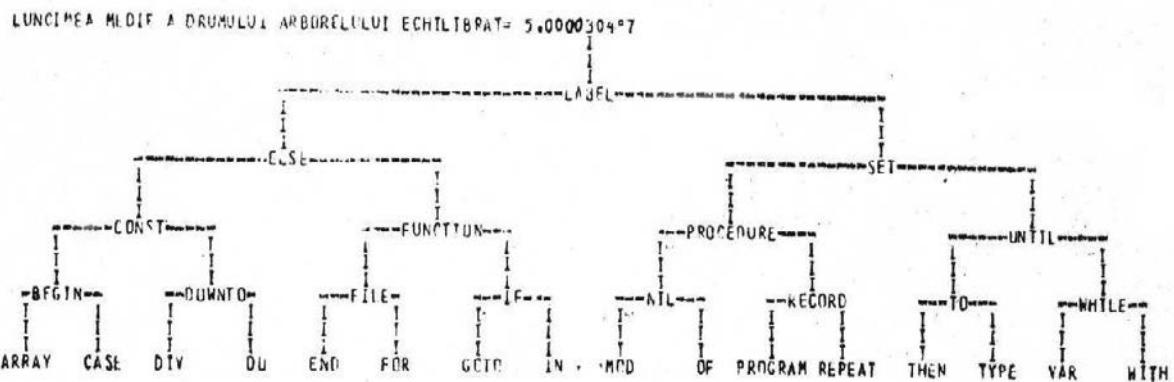
IF ch IN litere THEN
    BEGIN {identificator sau cheie}
        k1:=0;
        REPEAT
            IF k1<lch THEN
                BEGIN
                    k1:=k1+1; id[k1]:=ch
                END;
                Read(ch)
            UNTIL NOT((ch IN litere) OR (ch IN cifre));
            IF k1>=k2 THEN
                k2:=k1
            ELSE
                REPEAT
                    id[k2]:=' '; k2:=k2-1
                UNTIL k2=k1;
                i:=1; j:=n;
            REPEAT
                k:=(i+j) DIV 2;
                IF chei [k]<=id THEN i:=k+1;
                IF chei [k]>=id THEN j:=k-1
            UNTIL i>j;
            IF chei [k]=id THEN
                a[k]:=a[k]+1
            ELSE
                BEGIN
                    k:=(i+j) DIV 2; b[k]:=b[k]+1
                END
            END
        ELSE
            IF ch=''' '' THEN
                REPEAT
                    Read(ch)
                UNTIL ch=''' [ 8.8.c ]
            ELSE
                IF ch='{' THEN
                    REPEAT
                        Read(ch)
                    UNTIL ch='}'
                UNTIL ch='$'; {caracter sfârșit text sursă}
{pentru fiecare cuvânt cheie se afișează frecvențele a și b}
WriteLn('Cuvintele cheie și frecvențele lor de
        apariție');
suma:=0; sumb:=b[0];
FOR i:=1 TO n DO
    BEGIN
        suma:=suma+a[i]; sumb:=sumb+b[i];
        WriteLn('      ',b[i-1], '      ',a[i], '      ',chei[i],
                chei[i,1],chei[i,lch])
    END;
{se afișează suma frecvențelor de acces pentru a și b}
WriteLn('      ',b[n]);
WriteLn('      ----- -----');
WriteLn('      ',sumb,'      ',suma);
{se calculează matricea w din a și b}
FOR i:=1 TO n DO
    BEGIN

```

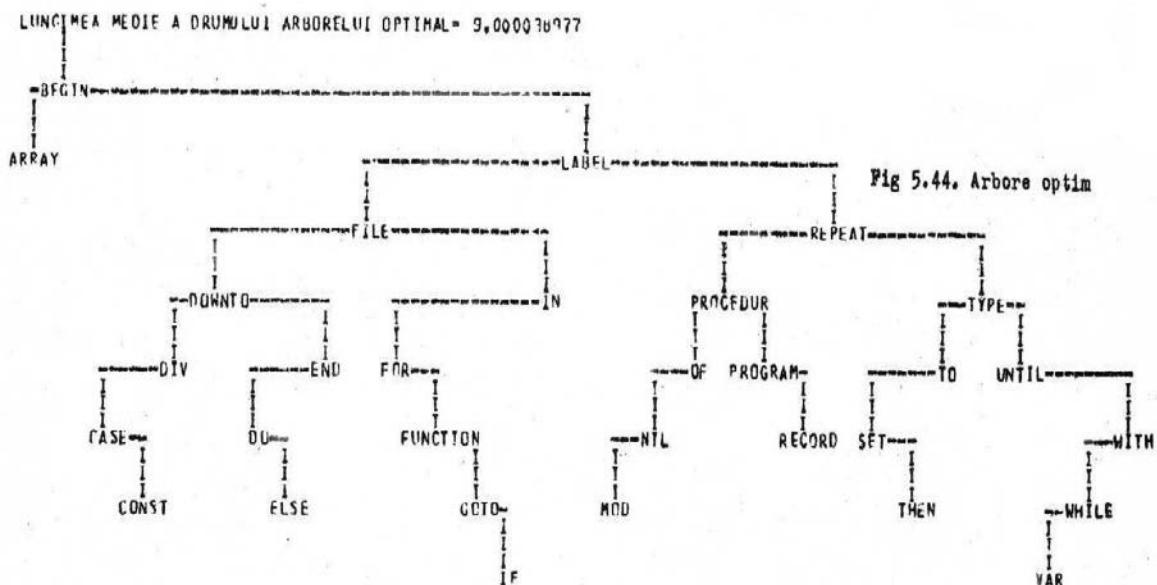
```

w[i,i]:=b[i];
FOR j:=i+1 TO n DO
    w[i,j]:=w[i,j-1]+a[j]+b[j]
END;
{se construiește și se afisează arborele perfect
echilibrat cu limitele 0 și n}
WriteLn;WriteLn;
WriteLn('    lungimea medie a drumului arborelului
    echilibrat=',conv(DrumArbEch(0,n))/conv(w[0,n]));
AfiseazaArbore;
{se construiește și se afisează arborele optim cu limitele
    0 și n}
Arbopt;
WriteLn;WriteLn;
WriteLn('    lungimea medie a drumului arborelui
    Optim =',conv(p[0,n])/conv(w[0,n]));
AfiseazaArbore;
{se recalculează w considerând numai cuvintele-cheie,
    adică făcând b=0}
FOR i:=0 TO n DO
    BEGIN
        w[i,i]:=0;
        FOR j:=i+1 TO n DO
            w[i,j]:=w[i,j-1]+a[j]
    END;
{se construiește și se afisează arborele optim cu limitele
    0 și n care nu conține decât cuvinte cheie}
Arbopt; [8.8.c]
WriteLn;WriteLn;
WriteLn('    arborele optim considerând numai
    cuvintele-cheie');
AfiseazaArbore
END. {ReprezentareArbore}
-----
```

- Rezultatele execuției acestui program apar în figurile următoare după cum urmează.
  - În figura 8.8.a apare reprezentarea grafică a **arborelui binar perfect echilibrat** corespunzător **cuvintelor cheie**.
  - În figura 8.8.b apare reprezentarea grafică a **arborelui binar optim** pentru **cuvintele cheie și identificatorii din programul sursă** [8.8.c].
  - În figura 8.8.c apare reprezentarea grafică a același **arbore binar optim** considerând **numai cuvintele cheie**.



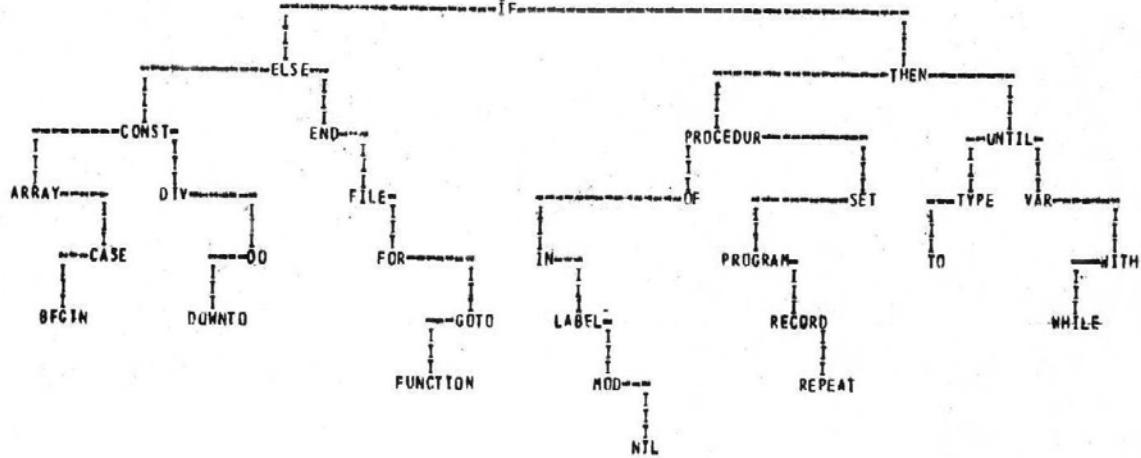
**Fig.8.8.a.** Reprezentarea grafică a arborelui binar perfect echilibrat corespunzător cuvintelor cheie



**Fig 5.44.** Arbore optim

**Fig.8.8.b.** Reprezentarea grafică a arborelui binar optim

ARBORELE OPTIM CONSIDERIND NUMAI CUVINTELE-CHEIE



**Fig.8.8.c.** Reprezentarea grafică a arborelui binar optim considerând numai cuvintele cheie

## 8.9. Arbori multicăi

### 8.9.1. Generalități

- Până în prezent au fost studiate cu predilecție structuri arbore în care fiecare nod avea **cel mult** doi descendenți.
- Desigur, acest lucru este pe deplin justificat dacă spre exemplu, se dorește să se reprezinte **descendența unei persoane** din punctul de vedere al strămoșilor,
  - În acest caz, fiecărei persoane i se asociază cei doi părinți ai săi.
- Dacă problema se abordează însă punctul de vedere al **urmașilor**, atunci o familie poate să aibă mai mult de doi copii, rezultând astfel noduri cu mai multe ramuri (gradul arborelui este mai mare ca 2).
  - Structurile care conțin astfel de noduri se numesc, după cum s-a mai precizat, **arbori generalizați**.
- Astfel de structuri ridică însă unele **probleme** în implementare.
  - Spre **exemplu**, în situația anterioară, dacă se cunoaște **numărul de copii**, atunci referințele la aceștia pot fi memorate într-un **tablou**, care devine o componentă a nodului afectat persoanei respective.
  - Dacă **numărul de copii** variază în limite largi, aceasta poate conduce la utilizarea ineficientă a memoriei.
- O altă soluție, mai eficientă, este aceea de a crea cu **referințele la copii**, **o listă liniară** al cărei început se păstrează în nodul părinte.
- Această structură de date, poate fi și mai mult complicată prin introducerea unor **componente suplimentare** în nodul corespunzător unei persoane, pentru a putea reprezenta spre exemplu diferite **grade de rudenie**.
  - O astfel de structură poate prograda rapid spre o **bază de date relațională** care poate îngloba mai mulți arbori în ea.
  - **Algoritmii** care operează asupra unei astfel de structuri, depind în mod **intim** de **structurile de date** definite, precizarea unor reguli și tehnici cu caracter general în acest caz fiind lipsită de sens.
  - Este însă evident faptul că modalitățile de **reprezentare a arborilor** sugerate până în prezent **nu** corespund într-o manieră eficientă unor astfel de structuri.
- În acest context se pot utiliza **arborii multicăi**, care reprezintă o categorie specială de **arbori generalizați**. Specificația acestora este următoarea:

- (1) Este vorba despre construcția și exploatarea **arborilor de foarte mari dimensiuni**.
- (2) Arbori în care se fac frecvent **inserții și suprimări**.
- (3) Arbori pentru care **dimensiunile memoriei centrale** sunt **insuficiente** sau a căror memorare vreme îndelungată în memoria sistemului de calcul este prea **costisitoare**.
- (4) Arbori în care **operația de căutare** trebuie să fie cât mai performantă.
- În acest scop a fost dezvoltată o **tehnică specială de implementare a arborilor multicăi**.
  - Să presupune că nodurile unui arbore trebuie să fie memorate într-o **memorie secundară**, spre exemplu pe un **disc magnetic**.
  - **Structurile de date dinamice** definite în acest curs se pretează foarte bine și acestui scop:
    - Astfel, **pointerii** care de regulă indică **adrese de memorie** pot indica în acest caz **adrese de disc**.
    - Utilizând spre exemplu un **arbore binar echilibrat** cu  $10^6$  noduri, căutarea unei chei necesită aproximativ  $\log_2 10^6 \approx 20$  pași.
    - Deoarece în acest caz, fiecare pas necesită un **acces** la disc (care este lent) se impune cu necesitate **o altă organizare** pentru **reducerea numărului de accese**.
  - **Arborii multicăi** reprezintă o **soluție perfectă** a acestei probleme.
- Se pornește de la următoarea constatare:
  - Este cunoscut faptul că după realizarea **accesului** (de regulă mecanic) la un anumit element de pe disc (**pistă**) sunt ușor accesibile (electronic) un întreg grup de elemente (**sectoarele corespunzătoare**).
  - Aceasta **sugerează** faptul că:
    - Un arbore poate fi divizat în **subarbori**.
    - **Subarborii** pot fi memorati pe disc ca unități la care accesul se realizează foarte rapid.
    - Acești subarbori se numesc **pagini**.
  - Considerând că accesul la **fiecare pagină** presupune un acces disc, dacă spre exemplu se plasează 100 noduri pe o pagină, atunci căutarea în arborele cu  $10^6$  noduri presupune  $\log_{100} 10^6 = 3$  accese disc în loc de 20.
  - În situația în care arborele crește aleator, în cel mai **defavorabil caz** (când degenerază în lista liniară) numărul de accese poate ajunge însă la  $10^4$ .

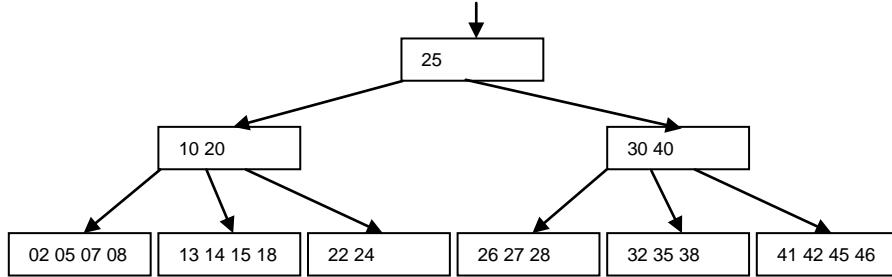
- Ca atare este evident faptul că și în cazul **arborilor multicăi**, trebuie avut în vedere un **mecanism de control al creșterii acestora**.
- Există mai multe variante de implementare a **arborilor multicăi**.
- Una dintre cele mai cunoscute modalități de implementare a arborilor multicăi o reprezintă **arborii-B**.

## 8.9.2. Arbori-B

### 8.9.2.1. Definire

- Din discuția asupra **mecanismului** de control al creșterii arborilor multicăi, **arborii perfect echilibrați** se exclud de la început din cauza costului ridicat al echilibrării.
- Un **criteriu** foarte potrivit în acest scop a fost postulat de **R.Bayer** în 1970 și anume:
  - Fiecare **pagină** a arborelui multicăi, cu excepția uneia, conține între  $n$  și  $2n$  noduri, unde  $n$  este o constantă dată.
    - Astfel într-un **arbore** cu  $N$  noduri, a cărui dimensiune maximă a unei pagini este cuprinsă între  $n$  și  $2n$  noduri, în cel mai rău caz, se fac  $\log_n N$  **accese la pagini** pentru a căuta o cheie precizată.
    - Factorul de **utilizare al memoriei** este de cel puțin 50%, deoarece orice pagină este cel puțin pe jumătate plină.
    - În plus schema preconizată presupune **algoritmi simpli** pentru **căutare, inserție și suprimare** în comparație cu alte metode.
- Structurile de date propuse de **Bayer** se numesc **arbori-B** iar  $n$  se numește **ordinul arborelui-B**.
- **Arborii-B** se bucură de următoarele **proprietăți**:
  - (1) Fiecare pagină a arborelui-B conține **cel mult**  $2n$  noduri (chei).
  - (2) Fiecare pagină, cu excepția paginii rădăcină conține **cel puțin**  $n$  noduri.
  - (3) Fiecare pagină este fie:
    - **O pagină terminală** - caz în care **nu** are descendenți.
    - **O pagina interioară** - caz în care are  $m+1$  descendenți unde  $m$  este numărul de chei din pagină ( $n \leq m \leq 2n$ ).
  - (4) Toate **paginiile terminale** sunt la același **nivel**.

- În figura 8.9.2.1.a apare reprezentat un arbore-B de ordinul 2 cu 3 niveluri.
  - Toate paginile conțin 2 , 3 sau 4 noduri cu excepția paginii rădăcină care conține unul singur.
  - Toate paginile terminale apar pe nivelul 3.

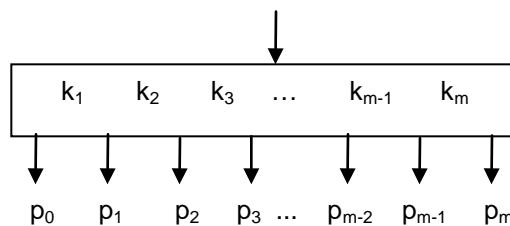


**Fig.8.9.2.1.a.** Arbore-B de ordinul 2

- Dacă această structură se **liniarizează** prin inserarea cheilor descendenților printre cheile strămoșilor lor, cheile nodurilor apar în **ordine crescătoare** de la stânga la dreapta.
- Această structurare reprezintă o **extensie naturală** a **structurii arbore binar ordonat** și ea stă la baza **metodei de căutare** ce va fi prezentată în continuare.

### 8.9.2.2. Căutarea cheilor în arbori-B

- Se consideră o **pagină** a unui arbore-B de forma prezentată în fig.8.9.2.2.a și o **cheie** dată  $x$ .
- Presupunând că pagina a fost transferată în memoria centrală a sistemului de calcul, pentru **căutarea cheii**  $x$  printre cheile  $k_1, \dots, k_m$  aparținând paginii, se poate utiliza o **metodă de căutare convențională**.
  - Se face următoarea **precizare** importantă: cheile  $k_i$  sunt **ordonate crescător** în cadrul paginii.



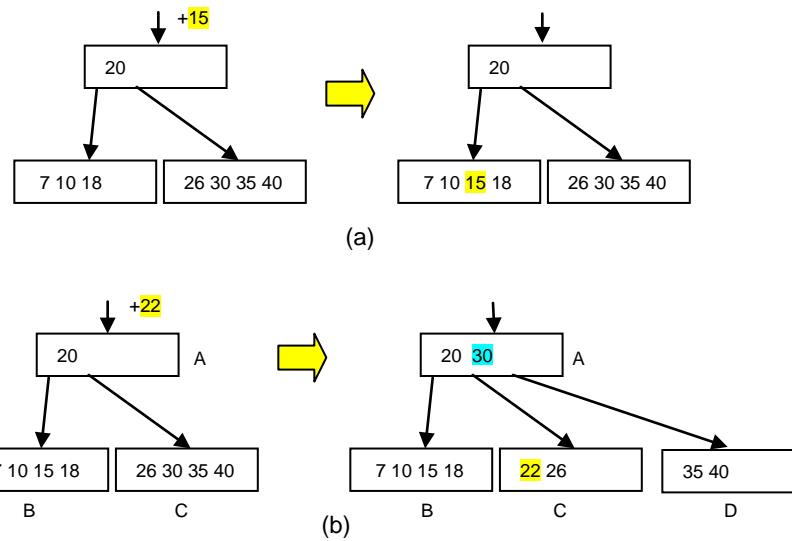
**Fig8.9.2.2.a.** Pagină cu  $m$  chei a unui arbore-B

- Astfel, dacă  $m$  este mare se poate utiliza **căutarea binară**, altfel, **căutarea liniară**.

- Trebuie subliniat faptul că **timpul de căutare** în memoria centrală este probabil **neglijabil** în comparație cu **timpul de transfer** al unei pagini din memoria secundară în cea primară.
- Dacă cheia  $x$  nu se găsește în **pagina curentă** este valabilă una din următoarele situații:
  - (1)  $k_i < x < k_{i+1}$ , pentru  $1 \leq i < m$ . Căutarea continuă în pagina  $p_i$ .
  - (2)  $k_m < x$ . Căutarea continuă în pagina  $p_m$ .
  - (3)  $x < k_1$ . Căutarea continuă în pagina  $p_0$ .
- Dacă **pointerul la pagina** desemnată de algoritmul de mai sus este **vid**, atunci **nu** există nici un nod cu cheia  $x$  și **căutarea este terminată**, adică s-a ajuns la baza arborelui într-o **pagină terminală**.

### 8.9.2.3. Inserția nodurilor în arbori B

- În ceea ce privește **inserția nodurilor** în arborii-B de ordinul  $n$ , aceasta se realizează implicit într-o **pagină terminală**.
  - Există în principiu două situații:
    - (1) Nodul trebuie inserat într-o pagină conținând  $m < 2n$  noduri.
      - În acest caz inserția se realizează simplu în pagina respectivă **inserând** cheia corespunzătoare la **locul potrivit** în secvența ordonată a cheilor.
        - Spre exemplu inserția cheii cu numărul 15 în arborele-B din fig.8.9.2.3.a (a).
    - (2) Nodul trebuie inserat într-o pagina care este **plină**, adică conține deja  $2n$  chei.
      - În acest caz structura arborelui se modifică conform exemplului prezentat în fig.8.9.2.3.a (b).
      - Astfel, spre exemplu inserția cheii cu numărul 22 în arborele-B se realizează în următorii pași:
        - (1) Se caută cheia 22 și se descoperă că ea lipsește, iar inserția în pagina destinație C este imposibilă deoarece aceasta este plină (conține  $2n$  chei).
        - (2) Pagina C se **scindează** în două prin alocarea unei noi pagini D.
        - (3) Cele  $2n+1$  chei ale paginii C sunt distribuite după cum urmează: primele  $n$  (cele mai mici) în pagina C, ultimele  $n$  (cele mai mari) în pagina D iar cheia mediană este translată pe nivelul inferior în pagina strâmoș A.



**Fig.8.9.2.3.a** Inserția nodurilor în arbori-B. Exemple

- Această schemă păstrează toate proprietățile caracteristice ale arborilor-B.
  - Se observă că paginile rezultate din scindare au exact  $n$  noduri.
- Desigur este posibil ca scindarea să se **propage** spre nivelurile superioare ale structurii, în cazul extrem până la rădăcină.
  - Aceasta este de fapt **singura** posibilitate ca un arbore-B să crească în înălțime.
- Maniera de creștere a unui astfel de arbore este **inedită**: el crește de la nodurile terminale spre rădăcină.
- În continuare se va dezvolta un **program** care materializează concepțele prezentate.
- Pornind de la proprietatea de **propagare a scindării paginii**, se consideră că formularea **recursivă** a algoritmului este cea mai convenabilă.
- Structura generală a programului este similară programului de inserție în **arbori echilibrați** (vezi &8.5.3).
- Pentru început se precizează structurile de date utilizate în implementarea arborilor-B [8.9.2.3.a].

---

**/\*Structură de date pentru arbori-B - varianta C\*/**

```

int n=...; /*n este ordinul arborelui B*/
int nn=2*n;

typedef struct nod {
    int cheie; /*cheie nod*/
    struct pagina * p; /*referința la pagina cu chei
                           mai mari*/
    int contor; /*contor chei*/
} NOD;
/*[8.9.2.3.a]*/

```

```

typedef struct pagina {
    int m; /*nr curent de elemente în pagină*/
    struct pagina* p0; /*referința la p0*/
    NOD elem[nn];
} PAGINA;

typedef PAGINA* refPagina;
-----
{Structură de date pentru arbori-B - varianta PASCAL}

CONST nn=2*n;

TYPE RefPagina=^pagina;
indice=0..nn;

nod=RECORD
    cheie:integer;
    p:RefPagina;
    contor:integer
END;                                [8.9.2.3.a]

pagina=RECORD
    m:indice; {nr curent de noduri în pagină}
    p0:RefPagina; {referință la prima pagină}
    elem:ARRAY[1..nn] OF nod
END;
-----
```

- Referitor la structura nod:
  - Câmpul **cheie** precizează cheia nodului respectiv.
  - Câmpul **p** indică pagina urmaș care conține **chei mai mari** decât cheia nodului în cauză.
  - Câmpul **contor** este utilizat ca **numărător de accese**.
- Referitor la structura **pagina**:
  - Fiecare pagină oferă spațiu pentru  $2n$  noduri.
  - Variabila **m** indică **numărul curent de noduri** memorate în pagina respectivă.
  - Tabloul **elem** memorează nodurile din pagina curentă în ordinea crescătoare a cheilor.
  - **p0** indică pagina urmaș cu chei mai mici decât cea mai mică cheie din pagină.
  - Deoarece  $m \geq n$ , (cu excepția rădăcinii), se garantează o utilizare a memoriei de cel puțin 50 %.

- În programul [8.9.2.3.d] apare **algoritmul de căutare și inserție** materializat de procedura **Cauta**.

- Structura de principiu a procedurii **Cauta** este asemănătoare cu cea a algoritmului de căutare în arbori binari, cu **excepția** faptului că decizia de ramificație în arbore **nu** e binară ci este cea specifică **arborilor-B** (&8.9.2.2).
- Căutarea în interiorul unei pagini este o **căutare binară** efectuată în tabloul elemente al paginii curente.
- Forma **pseudocod** a procedurii **Cauta** apare în secvența [8.9.2.3.b].

---

**/\*Schita de principiu a procedurii de căutare în arbori-B - varianta pseudocod\*/**

```
Subprogram Cauta(int x, refPagina a, boolean *h, NOD *v)

    NOD u;
    daca(a==null)
        /*x nu este în arbore*/
        *se crează nodul v;
        *i se atribuie cheia x;
        *se face h=TRUE indicând pasarea nodului v spre
          părintele său;
        □ /*daca*/
    altfel
        /*se caută x în pagina curentă a*/
        *căutare binară într-un tablou liniar;
        daca(găsit)
            *incrementează contorul de accese;
            altfel [8.9.2.3.b]
                Cauta(x, urmas, &h, &u);
                daca(*h) Insereaza; /*nodul u a fost pasat
                                         spre părintele său*/
                □ /*altfel*/
            □ /*altfel*/
    /*Cauta*/
```

---

**(Schita de principiu a procedurii de căutare în arbori-B - varianta PASCAL}**

```
PROCEDURE Cauta(x:integer; a:RefPagina; VAR h:boolean;
                    VAR v:nod);

VAR u:nod;

BEGIN
    IF a=NIL THEN
        BEGIN {x nu este în arbore}
            {*se creează un nod nou v}
            {*se atribuie cheia x nodului v și se pune h pe
              adevarat indicând pasarea nodului v spre rădăcina}
        END
    ELSE
        BEGIN {se caută x în pagina curentă a^}
            {*căutare binară într-un tablou liniar}
            IF gasit THEN
```

```

{ *incrementează contorul de accesă}
ELSE [8.9.2.3.b]
    BEGIN
        Cauta(x, urmas, h, u);
        IF h THEN Insereaza {nodul u care se pasează}
    END
END; {Cauta}
-----
```

- **Algoritmul de inserție** este formulat ca și o procedură aparte (procedura **Insereaza**) care este activată după ce procesul de căutare indică faptul că unul din noduri este **pasat spre pagina părinte**.
- Acest lucru este precizat de către valoarea "adevărat" a parametrului h returnat de procedura **Cauta**.
  - Dacă h este adevarat, parametrul u indică nodul care trebuie pasat paginii părinte, în direcția rădăcinii.
- Se precizează faptul că procesul de inserție începe într-o **pagină ipotetică**, de tip "**nod special**" situată virtual **sub nivelul terminal**.
  - Noul nod creat, este transmis via parametrul u paginii terminale pentru adevărata inserție.

**/\*Schița de principiu a inserției nodurilor în arbori-B - varianta pseudocod\*/**

#### **Subprogram Insereaza**

```

daca((numărul de noduri m al paginii a)<nn)
    *se inserează nodul u în pagina a la locul potrivit
    și se face h=fals;
altfel [8.9.2.3.c]
    *se crează o nouă pagină b;
    *se inserează cheia u la locul potrivit între
    cheile paginii a;
    *se redistribuie cheile paginii a, primele n
    pe a, ultimele n pe b;
    *se pasează nodul v=u conținând cheia mediană spre
    nivelul inferior;
    *h rămâne poziționat pe valoarea true;
    □ /*altfel*/
/*Insereaza*/
-----
```

**{Schița de principiu a inserției nodurilor în arbori-B - varianta PASCAL}**

#### **PROCEDURE Insereaza**

```

BEGIN
    IF (numărul de noduri a^.m al paginii a^)<2n THEN
        *se inserează nodul u în pagina a^ la locul potrivit
        și se face h=fals
-----
```

```

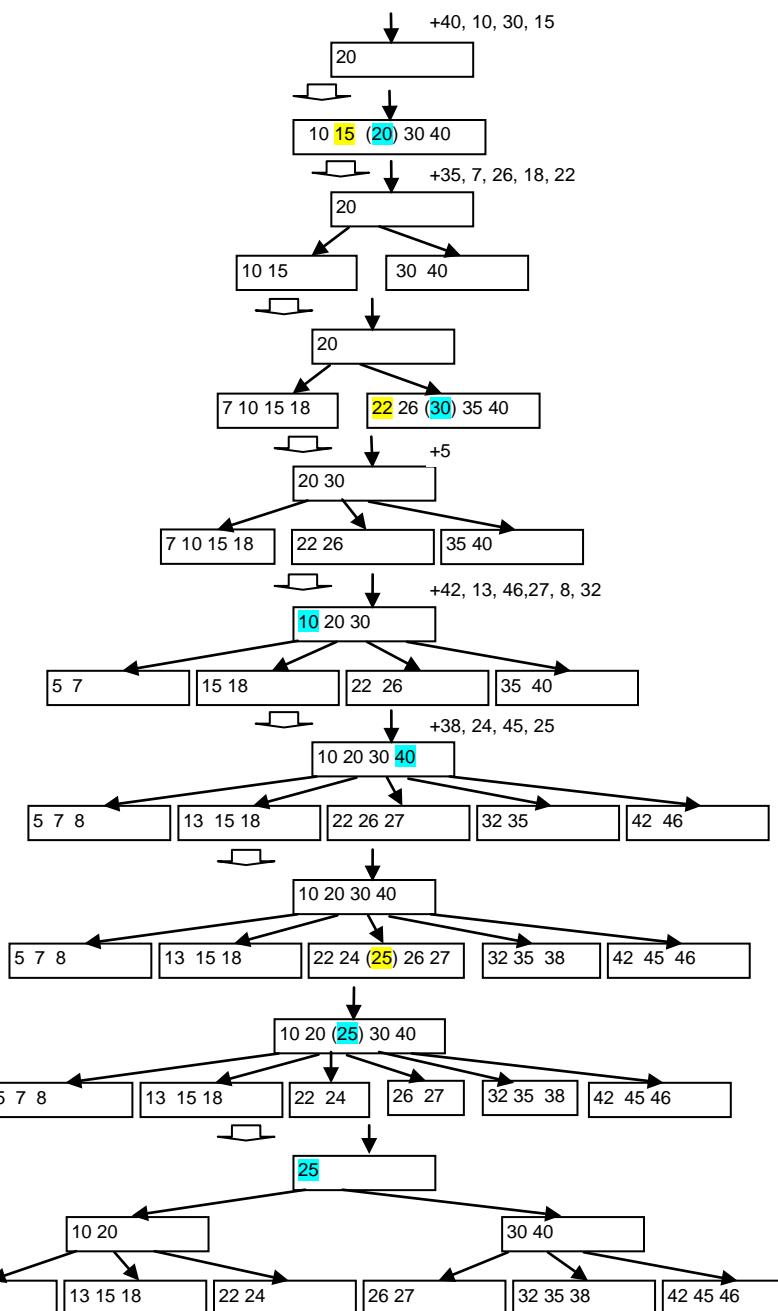
ELSE [8.9.2.3.c]
    BEGIN
        * se creează o nouă pagină b^;
        * se redistribuie cheile paginii a^, primele n pe
          a^, ultimele n pe b^;
        * se pasează nodul v=u conținând cheia mediană
          spre nivelul superior;
        * h rămâne poziționat pe valoarea true
    END
END; {Insereaza}

```

---

- Dacă parametrul  $h$  devine adevărat după apelul procedurii **Caută** din **programul principal**, acesta indică necesitatea **scindării** paginii **rădăcină**.
- Deoarece **pagina rădăcină** are un rol **special**, acest proces trebuie programat separat.
  - El constă de fapt din alocarea unei **noi pagini rădăcină** și inserarea unui singur nod transmis prin parametrul  $u$ .
- Implementarea inserției în arbori-B apare în programul **ArboriB**, secvența [8.9.2.4.a] din paragraful următor, procedurile **Caută** respectiv **Insereaza**.
- În legătură cu **inserția nodurilor** în arbori-B se fac următoarele observații:
  - (1) Deoarece paginile arborelui sunt alocate în memoria secundară, este necesar un mecanism pentru realizarea **transferului paginii curente** în memoria primară.
  - (2) Întrucât fiecare activare a procedurii **Caută** implică o alocare de pagină în memoria principală, vor fi necesare cel mult  $k = \log_n N$  apeluri recursive, unde  $n$  este ordinul arborelui-B iar  $N$  numărul total de noduri.
  - (3) Prin urmare dacă arborele conține  $N$  noduri, în memoria principală trebuie să încapă **cel puțin**  $k = \log_n N$  pagini.
  - (4) Acesta este unul din factorii care limitează **dimensiunea**  $2^n$  a paginii.
- În cadrul secvenței [8.9.2.4.a] instrucția **WITH** rezolvă aceste aspecte.
  - (1) În primul rând, indică faptul cunoscut că referirile sunt relative la pagina  $a$ .
  - (2) În al doilea rând, deoarece paginile sunt alocate în memoria secundară, instrucția **WITH** presupune și realizarea **transferului paginii vizate** în memoria primară.
- De fapt în memorie trebuie să existe mai mult de  $k$  pagini din cauza scindărilor care apar.
  - O consecință a acestei maniere de lucru este faptul că **pagina rădăcină** trebuie să fie **tot timpul** în memoria principală, ea fiind punctul de pornire al tuturor activităților.

- Un alt **avantaj** al structurii de date de tip arbore-B se referă la **actualizarea simplă și eficientă**, în mod secvențial a întregii structuri.
  - În acest caz, fiecare pagină este adusă în memorie exact odată.
- Se observă de asemenea faptul că **arborii-B** cresc relativ greu în înălțime, inserția unei noi pagini respectiv adăugarea unui nou nivel se realizează după inserția unui număr semnificativ de chei.
- În figura 8.9.2.3.b apare urma execuției programului la inserarea următoarei succesiuni de chei: 20; 40, 10, 30, 15; 35, 7, 26, 18, 22; 5; 42, 13, 46, 27, 8, 32; 38, 24, 45, 25;
- Punctul și virgula precizează momentele la care au avut loc alocări de pagini.
- Inserția ultimei chei (25) cauzează două scindări și alocarea a 3 noi pagini.

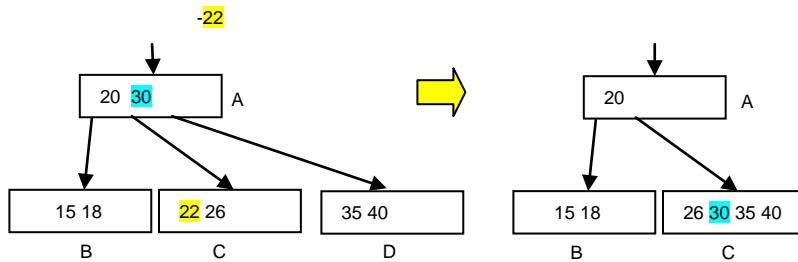


**Fig.8.9.3.2.b.** Construcția unui arbore-B de ordinul 2

#### **8.9.2.4. Suprimarea nodurilor în arbori-B**

- Principal, suprimarea nodurilor în arborii-B, este o operație simplă, la nivel de detaliu însă ea devine complicată.
- Se disting două situații:
  - (1) Nodul se găsește într-o **pagini terminală**, caz în care suprimarea este imediată.
  - (2) Nodul se găsește într-o **pagini internă**.
    - În acest caz nodul în cauză trebuie **înlocuit** cu unul dintre cele două noduri **adiacente**, care sunt în pagini terminale și prin urmare pot fi ușor suprimate.
    - De regulă înlocuirea se realizează cu **predecesorul** nodului respectiv.
- **Căutarea cheii adiacente** este similară celei utilizate la suprimarea nodurilor într-un **arbore binar ordonat** (vezi &8.3.5).
  - (1) Se înaintează spre **pagina terminală P** de-a lungul celor mai din **dreapta** pointeri ai **subarborelui stâng** al cheii de suprimat.
  - (2) Se înlocuiește nodul de suprimat cu **cel mai din dreapta nod** al lui P .
  - (3) Se reduce dimensiunea lui P cu 1.
- **Reducerea dimensiunii paginii** trebuie să fie urmată de **verificarea** numărului m de noduri din pagină.
  - Dacă  $m < n$  apare fenomenul numit "**subdepășire**" care este indicat de valoarea adevărat a lui h.
  - În acest caz soluția de rezolvare este aceea de a "**împrumuta**" un nod de la paginile vecine.
    - Întrucât această operație presupune aducerea unei pagini vecine (Q) în memoria principală - o operație relativ costisitoare - se preferă exploatarea la maxim a acestei situații prin împrumutarea mai multor noduri.
    - Astfel, în mod uzual, nodurile se distribuie în mod egal în paginile P și Q, proces numit **echilibrare**.

- În situația în care **nu** poate fi împrumutat nici un nod din pagina Q - aceasta având dimensiunea minimă n, paginile P și Q care împreună au  $2n-1$  noduri se **contopesc** într-una singură.
  - Acest lucru presupune extragerea nodului **median** din **pagina părinte** a lui P și Q, gruparea tuturor nodurilor într-una din pagini, adăugarea nodului median și ștergerea celeilalte pagini.
  - Acesta este procesul invers scindării paginii, proces care poate fi urmărit în figura 8.9.2.3.b, la suprimarea cheii cu numărul 22.



**Fig.8.9.2.3.b** Suprimarea nodurilor în arbori-B

- Din nou, extragerea cheii din mijloc din pagina strămoș, poate determina **subdepășirea**, situație care poate fi rezolvată fie prin echilibrare fie prin contopire.
- În caz extrem, procesul de contopire se poate propaga până la **rădăcină**.
- Dacă rădăcina este redusă la dimensiunea 0, ea dispare cauzând **reducerea înălțimii** arborelui-B.
- Aceasta este de fapt **singura cale de reducere** a dimensiunii unui arbore-B.
- În figura 8.9.2.4.a se prezintă evoluția unui arbore-B, rezultată din suprimarea următoarei sevențe de chei: 45, 25, 24; 38, 32; 8, 27, 46, 13, 42; 5, 22, 18, 26; 7, 35, 15.
  - Și în acest caz punctul și virgula precizează momentele la care sunt eliberate pagini.

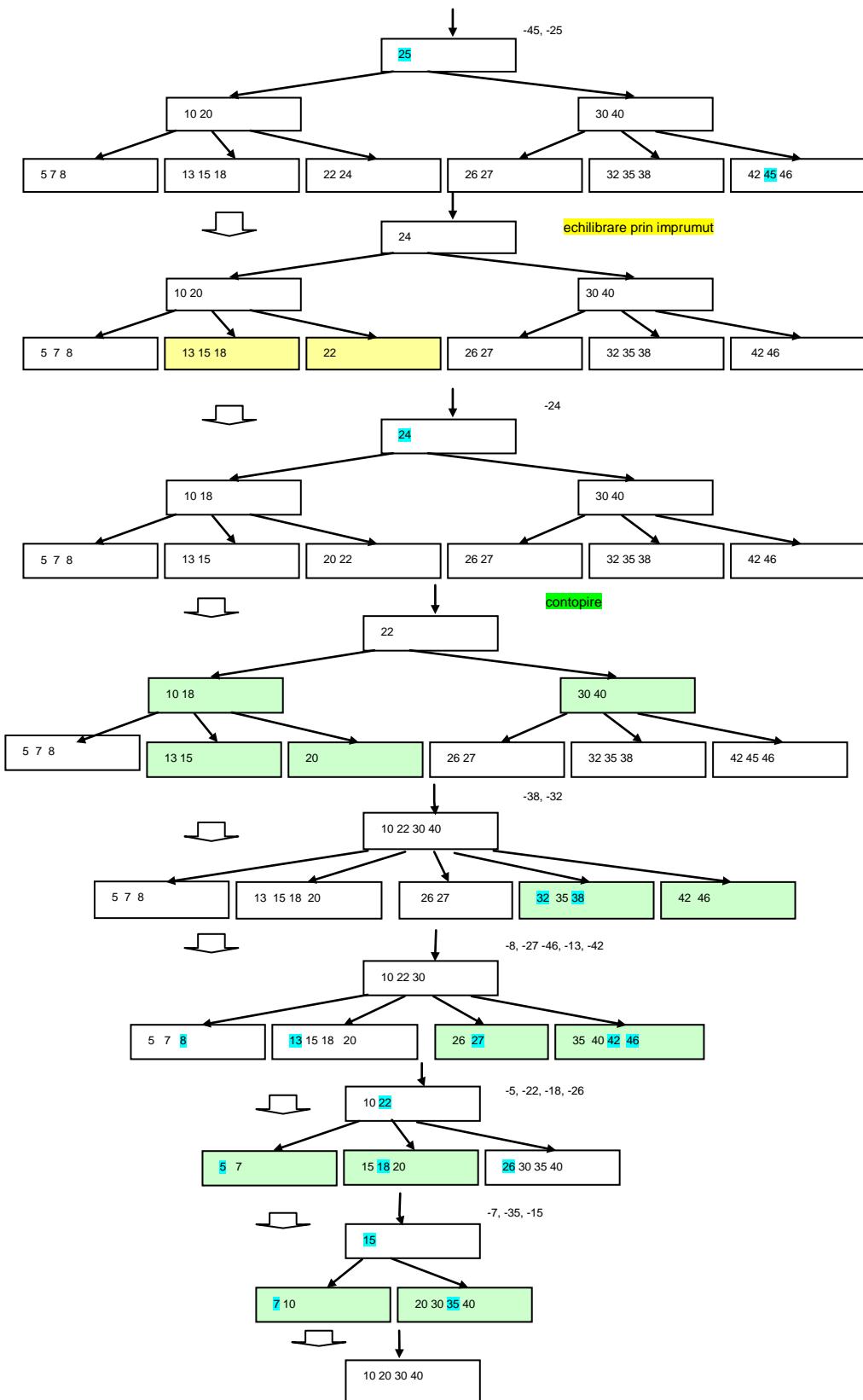


Fig.8.9.4.2.a. Suprimarea nodurilor în arbori-B

- Algoritmul de suprimare este inclus în programul din secvența [8.9.2.4.a] conceput ca și exemplu de aplicație pentru utilizarea arborilor-B.

- Referitor la **suprimare**, în cadrul aplicației sunt dezvoltate mai multe proceduri și anume.
- (1) Procedura **Suprima**:
  - Realizează parcurgerea arborelui-B căutând cheia de suprimat în manieră recursivă.
  - Dacă găsește cheia într-o pagină **terminală**, o suprimă, mută cheile mai mari cu o poziție spre stânga și îl asignează pe  $h=m < n$ , semnalând dacă este cazul **subdepășirea**.
  - În toate situațiile se verifică valoarea lui  $h$  și se apelează procedura **Subdepasire** dacă  $h=TRUE$ .
- (2) Procedura **Supr**:
  - Caută în manieră recursivă, **predecesorul** cheii de suprimat, substituie cheia de suprimat cu predecesorul găsit, îl suprimă și dacă este cazul semnalează subdepășire.
  - La revenirea din fiecare apel recursiv, verifică pe  $h$  și apelează dacă este cazul procedura **Subdepasire**.
- (3) Procedura **Subdepasire**:
  - Rezolvă problema echilibrării respectiv a contopirii paginilor adiacente funcție de situația concretă (pagina din dreapta respectiv cea din stânga paginii subdepășite indicate de referința  $a$ ).
- (4) Procedura **TipArb**:
  - Parcurge arborele B și afișează într-o manieră specifică structura acestuia.

---

**{Program arbori-B Varianta PASCAL}**

```

PROGRAM ArboriB;

{Căutare, inserție și suprimare în arbori B}

CONST n=2; nn=4; {dimensiune pagina}

TYPE refPagina=^pagina;

      TipNod=RECORD
        cheie:integer;
        p:refPagina; {referință la pagina cu chei mai mari}
        contor:integer
      END;

      pagina =RECORD
        m:0..nn;           {m= numărul de noduri}
        p0:refPagina;    {referință la pagina zero}
        e: ARRAY[1..nn] OF TipNod
      END;

```

```

END;

VAR radacina,q:refPagina; x:integer; [8.9.2.4.a]
      h:boolean; u:nod;

PROCEDURE Cauta(x:integer; VAR a:refPagina; VAR h:boolean;
                  VAR v:nod);

{Caută cheia x în arborele B de rădăcina a. Dacă o
găsește incrementează contorul, altfel inserează un nod
nou cu cheia x și contor:=1. Dacă un nod trebuie pasat
spre un nivel interior, el este atribuit lui v. h:=TRUE
semnifică că arborele a devenit mai înalt}

VAR k,s,d:integer; q:refPagina; u:nod;

PROCEDURE Insereaza;
  VAR i:integer; b:refPagina;
  BEGIN {inserează pe u în dreapta lui a^.elem[d] }
    WITH a^ DO
      BEGIN
        IF m<nn THEN
          BEGIN
            m:=m+1; h:=false;
            FOR i:=m DOWNTO d+2 DO e[i]:=e[i-1];
            e[d+1]:=u
          END
        ELSE
          BEGIN {pagina a^ e plină; scindează pagina și
                    atribuie nodul median lui v}
            new(b); {se creează o pagină nouă b}
            IF d<=n THEN
              BEGIN
                IF d=n THEN{inserează pe u} [8.9.2.4.a]
                  v:=u
                ELSE
                  BEGIN
                    v:=e[n];
                    FOR i:=n DOWNTO d+2 DO
                      e[i]:=e[i-1];
                      e[d+1]:=u
                    END;
                FOR i:=1 TO n DO
                  b^.e[i]:=a^.e[i+n]
                END
              ELSE
                BEGIN
                  d:=d-n; v:=e[n+1];
                  FOR i:=1 TO d-1 DO
                    b^.e[i]:=a^.e[i+n+1];
                    b^.e[d]:=u;
                  FOR i:=d+1 TO n DO
                    b^.e[i]:=a^.e[i+n];
                  END;
                m:=n; b^.m:=n; b^.p0:=v.p; v.p:=b
              END
            END {WITH}
          END; {Insereaza}

```

```

BEGIN {caută cheia x în pagina a^; h:=false}
  IF a=NIL THEN
    BEGIN {nodul cu cheia x nu este în arbore}
      h:=true;
      WITH v DO {se creează noul nod v}
        BEGIN
          cheie:=x; contor:=1; p:=NIL
        END
    END
  ELSE
    WITH a^ DO
      BEGIN
        s:=1; d:=m; {căutare binară}
        REPEAT
          k:=(s+d) DIV 2;
          IF x<=e[k].cheie THEN d:=k-1;
          IF x>=e[k].cheie THEN s:=k+1;
        UNTIL d<s;
        IF s-d>1 THEN {găsit, incrementare contor}
          BEGIN
            e[k].contor:=e[k].contor+1;
            h:=false
          END
        ELSE
          BEGIN {nodul nu e în această pagină}
            IF d=0 THEN
              q:=p0
            ELSE
              q:=e[d].p;
            Cauta(x,q,h,u);
            IF h THEN Insereaza
          END
        END
      END;
    END; {Cauta}

```

**PROCEDURE Suprima**(x:integer; **VAR** a:refPagina;  
**VAR** h:boolean);

{Caută și suprimă nodul cu cheia x din arborele-B având  
rădăcina a. Dacă o pagină devine subdimensionată se  
încearcă fie echilibrarea cu o pagină adiacentă (dacă este  
posibil), fie contopirea. h:=TRUE semnifică că pagina a  
este subdimensionată}

```

VAR i,k,s,d:integer; q:refPagina;

PROCEDURE Subdepasire(VAR c,a:refPagina; VAR s1:integer;  

VAR h:boolean);
{a=pagina subdepăsată; c=pagina strămoș}
VAR b:refPagina; i,k,mb,mc:integer;
BEGIN
  mc:=c^.m; {h=true; a^.m=n-1}
  IF s1<mc THEN
    BEGIN {b este pagina din dreapta lui a}
      s1:=s1+1;
      b:=c^.e[s1].p; mb:=b^.m; k:=(mb-n+1) DIV 2;
      {k=nr. de noduri disponibile în pagina

```

```

    adiacentă b}
    a^.e[n]:=c^.e[s1]; a^.e[n].p:=b^.p0;
    IF k>0 THEN
        BEGIN {mută k noduri din b în a }
            FOR i:=1 TO k-1 DO a^.e[i+n]:=b^.e[i];
            c^.e[s1]:=b^.e[k]; c^.e[s1].p:=b;
            b^.p0:=b^.e[k].p; mb:=mb-k;
            FOR i:=1 TO mb DO b^.e[i]:=b^.e[i+k];
            b^.m:=mb; a^.m:=n-1+k; h:=false
        END
    ELSE
        BEGIN {contopește paginile a și b}
            FOR i:=1 TO n DO a^.e[i+n]:=b^.e[i];
            FOR i:=s1 TO mc-1 DO c^.e[i]:=c^.e[i+1];
            a^.m:=nn; c^.m:=mc-1 {Dispose(b)}
        END
    END
ELSE
    BEGIN {b este pagina din stânga lui a}
        IF s1=1 THEN b:=c^.p0 ELSE b:=c^.e[s1-1].p;
        mb:=b^.m+1; k:=(mb-n) DIV 2;
        IF k>0 THEN
            BEGIN {mută k noduri din pagina b în a}
                FOR i:=n-1 DOWNTO 1 DO
                    a^.e[i+k]:=a^.e[i];
                    a^.e[k]:=c^.e[s1]; a^.e[k].p:=a^.p0;
                    mb:=mb-k;
                FOR i:=k-1 DOWNTO 1 DO
                    a^.e[i]:=b^.e[i+mb];
                    a^.p0:=b^.e[mb].p;
                    c^.e[s1]:=b^.e[mb]; c^.e[s1].p:=a;
                    b^.m:=mb-1; a^.m:=n-1+k; h:=false;
            END
        ELSE
            BEGIN {contopire pagini a și b}
                b^.e[mb]:=c^.e[s1]; b^.e[mb].p:=a^.p0;
                FOR i:=1 TO n-1 DO b^.e[i+mb]:=a^.e[i];
                b^.m:=nn; c^.m:=mc-1 {Dispose(a)}
            END
        END
    END;
{Subdepasire}

```

```

PROCEDURE Supr(VAR p:refPagina; VAR h:boolean);
VAR q:refPagina; {a și k variabile globale}
BEGIN
    WITH p^ DO
        BEGIN
            q:=e[m].p;
            IF q<>NIL THEN
                BEGIN
                    Supr(q,h);
                    IF h THEN Subdepasire(p,q,m,h)
                END
            ELSE
                BEGIN
                    p^.e[m].p:=a^.e[k].p; a^.e[k]:=p^.e[m];
                    m:=m-1; h:=m<n
                END
        END;
{Subdepasire}

```

```

END
END; {Supr}

BEGIN {Suprima}
  IF a=NIL THEN
    BEGIN
      WriteLn('    cheia nu exista in arbore');
      h:=false
    END
  ELSE
    WITH a^ DO
      BEGIN {căutare binară}
        s:=1; d:=m;
        REPEAT
          k:=(s+d) DIV 2;
          IF x<=e[k].cheie THEN d:=k-1;
          IF x>=e[k].cheie THEN s:=k+1;
        UNTIL s>d;
        IF d=0 THEN
          q:=p0
        ELSE
          q:=e[d].p;
        IF s-d>1 THEN
          BEGIN {găsit; se sterge (suprimă) e[k]}
            IF q=NIL THEN
              BEGIN {a este o pagină terminală}
                m:=m-1; h:=m<n;
                FOR i:=k TO m DO e[i]:=e[i+1];
              END
            ELSE
              BEGIN {a nu este o pagină terminală}
                Supr(q,h);
                IF h THEN Subdepasire(a,q,d,h)
              END
            END
          ELSE {cautare pe pagina urmatoare}
            BEGIN
              Suprima(x,q,h);
              IF h THEN Subdepasire(a,q,d,h)
            END
          END
        END
      END; {Suprima}

PROCEDURE TipArb(p:refPagina; l:integer);
  VAR i:integer;
  BEGIN [8.9.2.4.a]
    IF p<>NIL THEN
      WITH p^ DO
        BEGIN
          FOR i:=1 TO l DO Write(' ');
          FOR i:=1 TO m DO Write(e[i].cheie);
          WriteLn;
          TipArb(p0,l+1);
          FOR i:=1 TO m DO TipArb(e[i].p,l+1)
        END
      END; {TipArb}

BEGIN {programul principal}

```

```

radacina:=NIL; Read(x);
WHILE x<>0 DO
    BEGIN
        WriteLn('cauta cheia ',x);
        Cauta(x,radacina,h,u);
        IF h THEN
            BEGIN {inserează noua pagină de bază}
                q:=radacina; new(radacina);
                WITH radacina^ DO
                    BEGIN
                        m:=1; p0:=q; e[1]:=u
                    END;
                END;
                TipArb(radacina,1); Read(x)
            END;
        Read(x);
        WHILE x<>0 DO
            BEGIN
                WriteLn('sterge cheia ',x);
                Suprima(x,radacina,h);
                IF h THEN
                    BEGIN {pagina de bază s-a redus ca dimensiune}
                        IF radacina^.m=0 THEN
                            BEGIN
                                q:=radacina; radacina:=q^.p0
                                {Dispose(q)}
                            END
                        END;
                    TipArb(radacina,1); Read(x)
                END
            END.

```

---

- Analiza performanței arborilor-B pune în evidență o puternică **dependență** a dimensiunii  $n$  a paginii, față de caracteristicile memoriei și de cele ale sistemului de calcul.
- Ca și o observație se poate menționa faptul că la inserția nodurilor în arborii-B, **scindarea** paginii poate fi amânată, încercând în prealabil **echilibrarea** paginilor alăturate.
- Diferite variante de **arbori B** sunt discutate de către Knuth [Kn76].

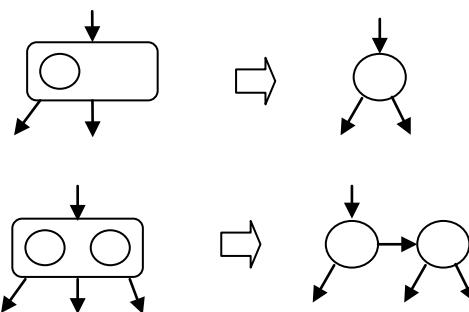
### 8.9.3. Arboare-B binari

- O categorie aparte de arbori-B o reprezintă cei de ordinul 1 ( $n=1$ ), care se mai numesc "**arbori-B binari**" sau "**arbori BB**".
- Întrucât paginile unui astfel de arbore conțin **unul** sau **două noduri**, utilizarea lor în contextul reprezentării unor masive de date în memoria secundară a sistemului de calcul este în afara discuției din cauza risipei de memorie.
  - Aproximativ 50 % din pagini vor conține un singur element.

- În continuare, studiul acestei categorii de arbori se va aborda în accepțiunea **rezidenței lor integrale** în memoria centrală.

- Conform **definiției** arborilor-B:

- (1) Toate **paginiile terminale** ale unui arbore-B binar apar la **același nivel**.
- (2) **Paginiile interioare** conțin 1 sau 2 chei și în consecință 2 respectiv 3 descendenți.
- Din acest motiv **arborii-B binari** se mai numesc și "**arbori 2-3**".
- Pentru reprezentarea paginilor unui astfel de arbore se pot utiliza **tablouri liniare** sau **pointeri**.
  - Din considerente de ocupare a memoriei se preferă **pointerii**.
- Fiecare **pagina** a unui arbore 2-3 este de fapt o listă înlățuită de 1 sau 2 noduri.
- Deoarece **fiecare pagina** are cel mult trei descendenți, există posibilitatea **combinării** pointerilor ce indică **descendenții** cu cei ce realizează **înlățuirea** în cadrul listei, după cum rezultă din figura 8.9.3.a.



**Fig.8.9.3.a.** Reprezentarea arborilor-B binari

- Astfel noțiunea de **pagina** dispare, nodurile apărând într-o structură asemănătoare unui arbore binar obișnuit.
- Totuși trebuie precizat faptul că în acest caz, se definesc **două categorii de pointeri**:
  - Pointerii care se referă la **descendenți** (cei verticali).
  - Pointerii care se referă la **frați** (cei orizontali).
- Deoarece **numai** pointerul pe dreapta poate fi orizontal, este suficientă o variabilă booleană indicator (`orizontala`) în cadrul unui nod care să precizeze acest fapt.
- În aceste condiții se pot utiliza următoarele structuri de date [8.9.3.a].

---

```
/*Structură de date pentru reprezentarea unui nod al unui
arbore-B binar - varianta C*/
```

```
typedef struct tip_nod
```

```

    /*diferite campuri*/
    char cheie;
    struct tip_nod* stang;
    struct tip_nod* drept;
    boolean orizontal;
} NOD_BB;

```

[8.3.3.a]

---

```

{Structură de date pentru reprezentarea unui nod al unui
arbore-B binar - varianta Pascal}

```

```

TYPE RefNod=^Nod

```

```

Nod=RECORD

```

```

    cheie:integer;

```

```

    {alte câmpuri}

```

```

    stang,drept:RefNod;

```

```

    orizontal:boolean

```

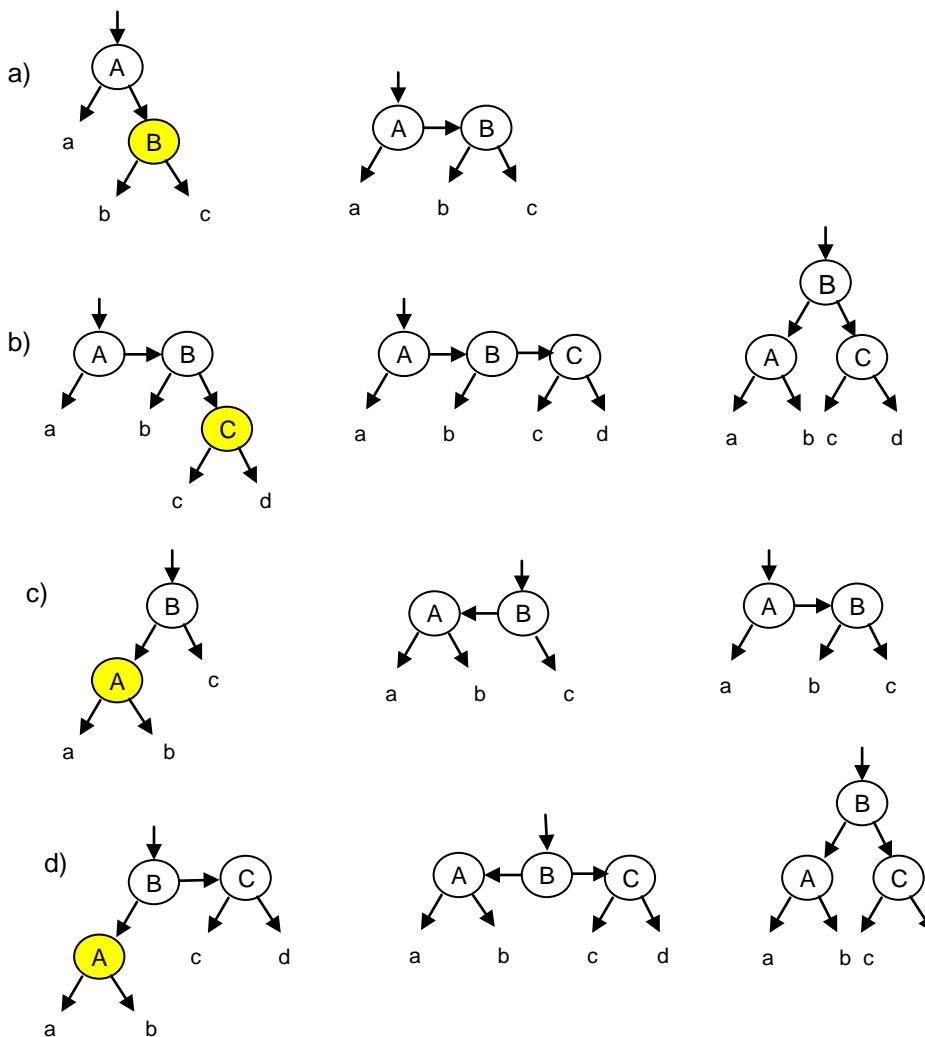
```

END;

```

[8.9.3.a]

- Acest mod de organizare asigură o **lungime maximă a drumului de căutare** în arbore  $p = 2 \lceil \log_2 N \rceil$ .



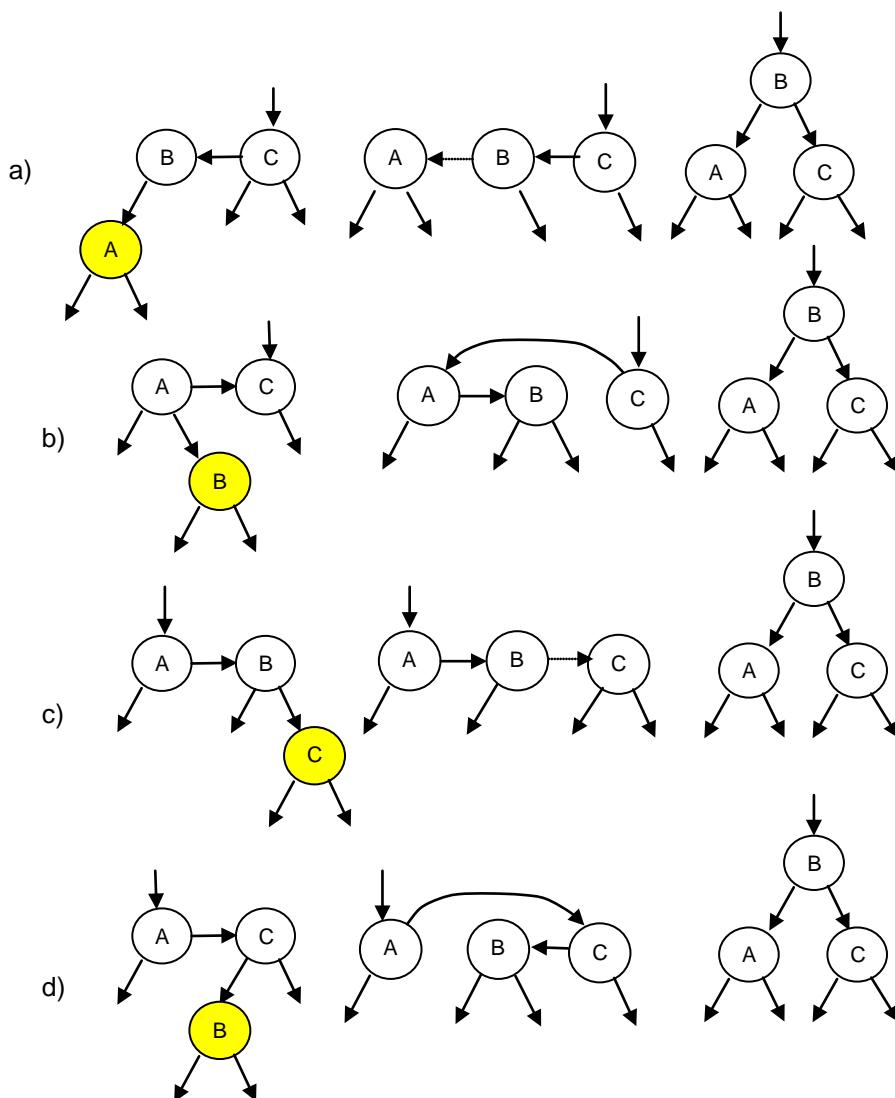
### **Fig.8.9.3.b.** Inserția nodurilor în arbori-B binari

- În ceea ce privește **inserția** unui nod nou într-un arbore-B binar, se disting patru situații posibile, după cum crește **subarborele** sau **stâng** sau **subarborele** sau **drept** (fig.8.9.3.b).
  - Se reamintește faptul că arborii-B cresc spre rădăcină și că toate nodurile lor terminale sunt la același nivel.
  - (a) Cazul (a). Se referă la situația în care **subarborele drept** al nodului A crește, iar **a** este singura cheie din pagina (ipotetică) corespunzătoare.
    - Nodul B care se adaugă va deveni fratele drept al lui A, pointerul drept vertical al celui din urmă devenind orizontal.
  - (b) Cazul (b). Simpla modificare a tipului pointerului **nu** este posibilă dacă A are frate drept.
    - În această situație se obține o pagină cu 3 noduri, care se scindează.
    - Nodul B din mijloc este trecut în sus pe nivelul superior.
  - (c) Cazul (c). Dacă subarborele stâng al unui nod B a crescut în înălțime și B este singur în pagină (cazul (c)), atunci A poate fi adus în aceeași pagină.
    - A devine frate stânga al lui B, element care contravine definiției, însă un pointer stânga nu poate fi orizontal.
    - Se modifică rădăcina arborelui astfel încât să-l indice pe A, iar pointerul orizontal dreapta al lui A îl va indica pe B.
  - (d) Cazul (d). Dacă însă B are frate drept, inserția lui A face necesară scindarea paginii. B este transmis în sus pe nivelul superior.
    - A și C devin descendenții nodului B.
    - Se precizează faptul că în cadrul **procesului de căutare**, **nu** se face nici o diferență între pointerii orizontali și cei verticali.

#### **8.9.3.1. Arbori-B binari simetrici**

- Se observă însă în figura 8.9.3.b o **diferențiere** referitoare la tratarea diferită a cazurilor de creștere a subarborelui drept (cazul (a)) respectiv a celui stâng (cazul (c)), element care conferă structurii de arbore-B binar un **caracter asimetric**.
  - Evitarea acestei asimetrii a condus la conceptul de **arborii-B binari simetrici** (arborei **BBS**).
- **Arborii-B binari simetrici** sunt arbori-B în care atât pointerul stâng cât și cel drept pot fi de tip "orizontal" sau "vertical".

- Din acest motiv, fiecare nod necesită în acest caz două variabile booleene pentru precizarea naturii celor doi pointeri respectiv OS și OD.
- În medie **procesul de căutare** în astfel de arbori este **mai eficient**, în schimb algoritmii de inserție și suprimare sunt mai complicați.
- În continuare se abordează în detaliu doar studiul **inserției** nodurilor în arbori BBS, pentru care se disting patru cazuri (fig.8.9.3.1.a).
  - Simetria este evidentă.



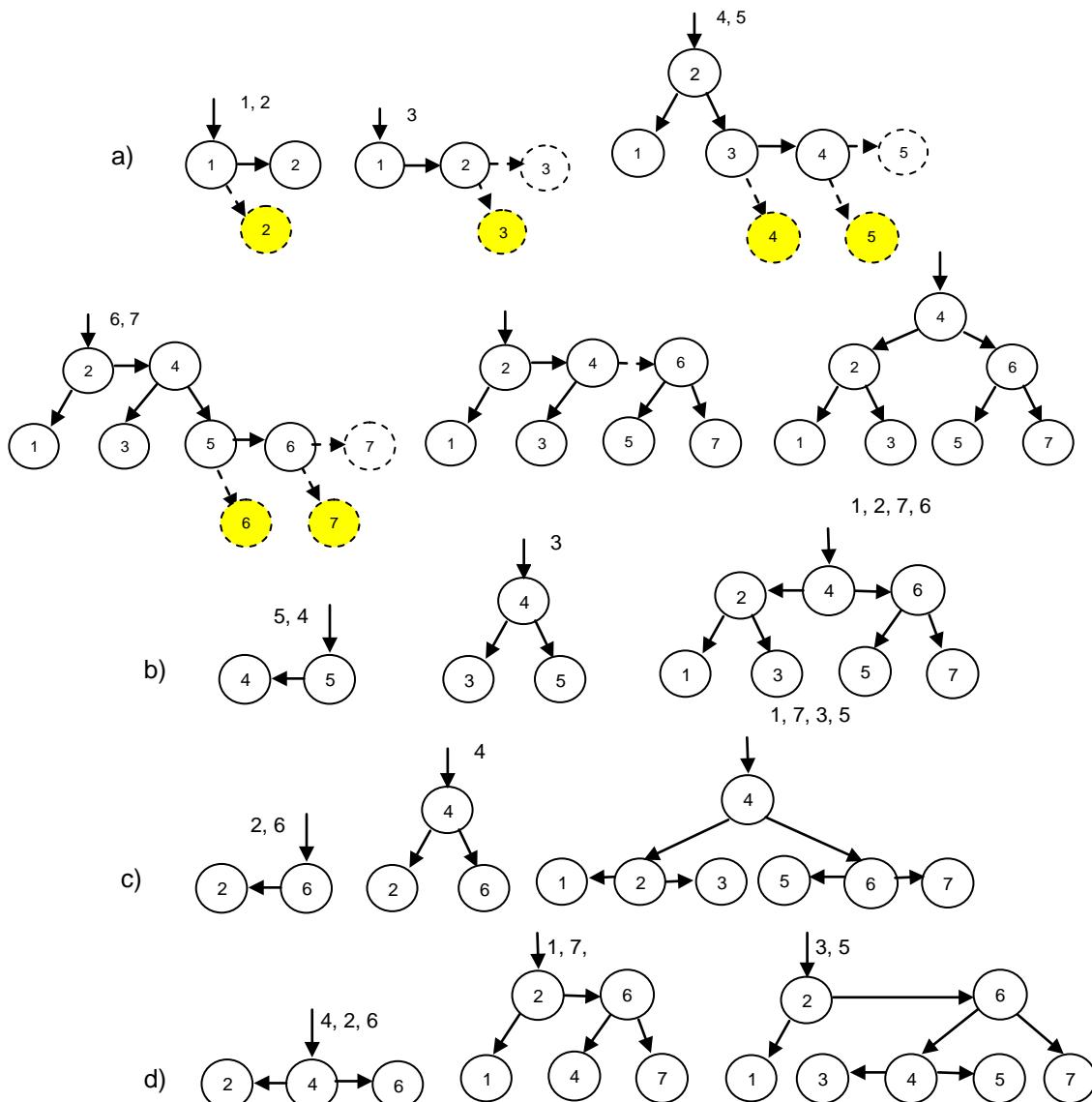
**Fig.8.9.3.1.a.** Inserția nodurilor în arbori-B binari simetrici. (a) Cazul 1 Stânga, (b) Cazul 2 Stânga, (c) Cazul 1 Dreapta, (d) Cazul 2 Dreapta.

- Se precizează faptul că, situația în care crește un subarbore a unui nod A fără frate, se tratează simplu:
  - Nodul rădăcină al subansamblului respectiv devine fratele lui A.
  - Această situație **nu** apare reprezentată în figură.

- Cazurile prezentate în figura 8.9.3.1.a conduc la **depășirea paginii** și în consecință la **scindare**.
  - Pentru fiecare din aceste cazuri din figura 8.9.3.1.a:
    - Coloana din dreapta precizează **situația inițială**.
    - Coloana din mijloc precizează **situația intermediară** rezultată în urma ridicării nodului în "pagină", urmată de creșterea numărului de subarbori.
    - Coloana din dreapta **situația finală** rezultată în urma rearanjării structurii (scindării).
  - În continuare, **se renunță la conceptul de pagină** care de altfel a stat la baza arborilor-B.
  - Ceea ce interesează în continuare este **limitarea lungimii maxime a drumului de căutare** la  $2 \log_2 N$ .
    - Pentru aceasta este necesar ca pe **nici un drum** să nu apară **două pointeri orizontali succesivi**.
    - Această restricție **nu** interzice însă existența unor noduri având **ambii pointeri orizontali** (unul spre stânga, celălalt spre dreapta).
  - Cu aceste precizări, un **arbore B binar simetric** se definește ca fiind structura arbore care satisface următoarele proprietăți:
    - (1) Fiecare nod conține o cheie și cel mult doi pointeri de subarbori.
    - (2) Fiecare pointer este fie orizontal, fie vertical.
    - (3) Nu există doi pointeri orizontali succesivi în nici un drum de căutare.
    - (4) Toate nodurile terminale apar la același nivel.
  - Din această definiție rezultă că cel mai lung drum de căutare **nu** poate depăși dublul înălțimii arborelui ( $2 \log_2 N$ ).
  - În figura 8.9.3.1.b se prezintă maniera de dezvoltare a arborilor BBS.
    - Secvențele de chei care se inserează corespunzător celor patru situații prezentate în figură sunt următoarele:
      - (a) 1, 2 ; 3 ; 4, 5, 6 ; 7 ;
      - (b) 5, 4 ; 3 ; 1, 2, 7, 6 ;
      - (c) 6, 2 ; 4 ; 1, 7, 3, 5 ;
      - (d) 4, 2, 6 ; 1, 7 ; 3, 5 ;
    - Punctul și virgula precizează momentul realizării reprezentării.
    - Prima dintre situații (a) este prezentată mai detaliat, pentru înțelegerea mai facilă a procedurii de reechilibrare.

- Se remarcă două **caracteristici fundamentale**:

- (1) Toate **nodurile terminale** apar la același nivel.
- (2) **Echilibrarea** se realizează dacă apar doi pointeri orizontali succesivi indicând același sens.



**Fig.8.9.3.1.b.** Dezvoltarea arborilor-B binari simetrici.

- În continuare se prezintă structurile de date utilizate la definirea unui nod al unui **arbore-B binar simetric** (secvența [8.9.3.1.a] precum și algoritmul care realizează construcția unui arbore-B binar simetric (secvența [8.9.3.1.b]).

---

```
/*Structură de date pentru reprezentarea unui nod al unui
arbore-B binar simetric - varianta C*/
```

```
typedef struct tip_nod
{
    /*diferite campuri*/
    char cheie;
    int contor;
```

```

struct tip_nod* stang;           /*[8.9.3.1.a]*/
struct tip_nod* drept;
boolean os,od;
} NOD_BBS;

typedef tip_nod * ref_tip_nod_BBS;
-----
{Structura de date nod arbore-B binar simetric - varianta
PASCAL}

TYPE nod=RECORD
    cheie: integer;
    contor: integer;
    stang,drept: refNod;
    os,od: boolean
END;
-----
{Construcția unui arbore nod arbore-B binar simetric -
Varianta PASCAL}

PROCEDURE Cauta(x:integer; VAR p:refNod; VAR h:integer);

{Caută cheia x într-un arbore-B binar simetric. Dacă o
găsește îi incrementează contorul asociat. Dacă nu o găsește
crează un nod nou, inserează cheia și funcție de situație
restructurează arborele}

VAR p1,p2:refNod;
BEGIN
    IF p=NIL THEN
        BEGIN {cheia nu e în arbore; se inserează}
            new(p); h:=2;
            WITH p^ DO
                BEGIN
                    cheie:=x; contor:=1; stang:=NIL;
                    drept:=NIL; os:=false; od:=false
                END
            END
        ELSE
            IF x<p^.cheie THEN {parcursere subarbore stâng}
                BEGIN
                    Cauta(x,p^.stang,h);
                    IF h<>0 THEN
                        IF p^.os THEN
                            BEGIN
                                p1:=p^.stang; h:=2; pq.os:=false;
                                IF p1^.os THEN
                                    BEGIN {Cazul 1 Stânga}
                                        p^.stang:=p1^.drept;
                                        p1^.drept:=p; p1^.os:=false;
                                        p:=p1
                                    END
                                ELSE
                                    IF p1^.od THEN
                                        BEGIN {Cazul 2 Stânga}
                                            p2:=p1^.drept;
                                            p1^.od:=false;
                                            p1^.drept:=p2^.stang;
                                        END
                                END
                            END
                        END
                    END
                END
            END
        END
    END

```

```

        p2^.stang:=p1;
        p^.stang:=p2^.drept;
        p2^.drept:=p;
        p:=p2
    END
END
ELSE
BEGIN
    m:=h-1;
    IF h<>0 THEN p^.os:=true
END [ 8.9.3.1.b ]
END
ELSE
IF x>p^.cheie THEN {parcurgere subarbore drept}
BEGIN
    Cauta(x,p^.drept,h);
    IF h<>0 THEN
        IF p^.od THEN
            BEGIN
                p1:=p^.drept; h:=2;
                p^.od:=false
                IF p1^.od THEN
                    BEGIN {Cazul 1 Dreapta}
                        p^.drept:=p1^.stang;
                        p1^.stang:=p;
                        p1^.od:=false;
                        p:=p1
                    END
                ELSE
                    IF p1^.os THEN
                        BEGIN {Cazul 2 Dreapta}
                            p2:=p1^.sting;
                            p1^.os:=false;
                            p1^.stang:=p2^.drept;
                            p2^.drept:=p1;
                            p^.drept:=p2^.stang;
                            p2^.stang:=p; p:=p2
                        END
                    END
                ELSE
                    BEGIN
                        h:=h-1;
                        IF h<>0 THEN p^.od:=true;
                    END
                END
            ELSE {cheia există în arbore}
                BEGIN
                    p^.contor:=p^.contor+1; h:=0
                END
            END;
        {Cauta}
-----
```

- Procedura recursivă **Cauta** se bazează pe schema clasică a inserției în arbori binari ordonați (vezi &8.3.4.).

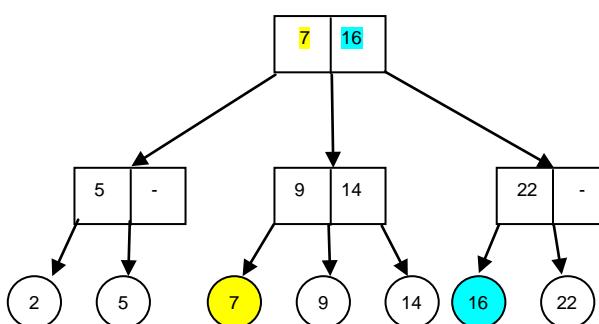
- Parametrul  $h$ , care corespunde parametrului cu același nume din programul de căutare în arbori-B, are rolul de a indica faptul că arborele având rădăcina  $p$  s-a **modificat**.
- Reprezentarea "paginilor" ca și "liste înlățuite" are drept consecință traversarea unei "pagini" în unul sau două apeluri ale procedurii de căutare.
- Din acest motiv trebuie să se facă distincție între:
  - (1) Un subarbore (indicat printr-un **pointer vertical**) care a crescut.
  - (2) Un nod frate (indicat printr-un **pointer orizontal**) care a mai obținut un frate și care necesită astfel scindarea paginii.
- Problema se rezolvă introducând trei valori pentru variabila  $h$ :
  - (1)  $h=0$  - subarborele  $p$  nu determină modificarea structurii arborelui.
  - (2)  $h=1$  - nodul  $p$  a obținut un frate.
  - (3)  $h=2$  - subarborele  $p$  a crescut în înălțime.
- Se observă că rotirea pointerelor este foarte asemănătoare cu cea implementată de algoritmul referitor la **arborii echilibrați** (&8.5.3).
- Se poate demonstra faptul că **arborii AVL** echilibrați sunt de fapt **un subset** al **arborilor BBS**.
  - Rezultă, în consecință că drumul mediu al arborilor BBS este mai lung decât cel corespunzător clasei AVL.
  - Pe de altă parte însă, rearanjarea nodurilor este necesară mai puțin frecvent, pentru clasa BBS.
- În concluzie:
  - (1) **Arborii echilibrați AVL** sunt de preferat în acele aplicații în care operația de căutare a cheilor este mai frecventă decât inserția sau suprimarea.
  - (2) Dacă raportul acestor operații este moderat, se preferă **arborii BBS**.
  - (3) Desigur, limite în acest sens sunt greu de precizat, deoarece pe lângă raportul dintre frecvența căutărilor și frecvența modificărilor structurale, intervin cu o pondere foarte semnificativă detaliile de implementare.

#### 8.9.4. Arbori 2-3

##### 8.9.4.1. Definire

- Arborii-B binari se încadrează în categoria arborilor-B, în particular fiind definiți drept "**arbori B de ordinul 1**".
- Conform **definiției**:
  - Toate **nodurile terminale** ale unui astfel de arbore apar la același nivel.
  - **Nodurile interioare** conțin unul sau două elemente și în consecință au doi respectiv trei descendenți.
- Din acest motiv **arborii-B binari** se mai numesc și **arbori 2-3**.

- În paragraful anterior (&8.9.3) au fost prezentate unele modalități de implementare a arborilor 2-3 bazate pe structuri de **arbori binari modificați**.
- În cadrul paragrafului de față va fi prezentată o altă **modalitate de reprezentare** [AH85].
  - Ca și în alte situații se presupune că **arborele** va reprezenta o **mulțime de elemente** peste care este definită o **relație de ordonare** notată cu “<”.
  - Elementele mulțimii sunt plasate în **nodurile terminale** care sunt situate toate **pe același nivel** al arborelui.
  - Dacă un element  $x$  se găsește la stânga unui element  $y$  atunci este valabilă relația  $x < y$ .
  - În fiecare **nod interior** al structurii arbore 2-3:
    - (1) Pe **prima poziție** va fi memorată **cheia celui mai mic descendant terminal** al celui **de-al doilea fiu** al nodului.
    - (2) Dacă nodul conține și un **al treilea fiu**, atunci pe **poziția următoare** în nod se înregistrează **cheia celui mai mic descendant terminal** al celui **de-al treilea fiu**.
    - Se precizează că prin **cel mai mic descendant terminal** se înțelege elementul **terminal** cu **cheia cea mai mică** în contextul considerat.
  - Astfel în figura 8.9.4.1.a. luând în considerare nodul rădăcină  $(7, 16)$  se observă că:
    - (1) Prima cheie memorată în acest nod este cel mai mic descendant terminal al celui de-al doilea fiu al său care este nodul  $(9, 14)$  adică cheia 7.
    - (2) Pe a doua poziție în cadrul nodului rădăcină apare valoarea celui mai mic descendant terminal al celui de-al treilea fiu al său care este nodul  $(22, -)$ , adică cheia 16.
  - Se observă că, principal se obține o structură asemănătoare cu structura de arbore-B reprezentată cu ajutorul paginilor (& 8.9.2).



**Fig.8.9.4.1.a.** Arbore 2-3

- Un arbore 2-3 cu  $h$  niveluri (de înălțime  $h$ ), conține între  $2^{h-1}$  și  $3^{h-1}$  noduri.
- Cu alte cuvinte, arborele 2-3 care reprezintă o mulțime de  $n$  elemente are înălțimea cuprinsă între  $1 + \log_3 n$  și  $1 + \log_2 n$ .

- Astfel, **lungimea maximă** a drumului de căutare pentru un nod terminal al arborelui este  $O(\log_2 n)$ .
- **Căutarea unei chei într-un arbore 2-3** se realizează cu un efort  $O(\log_2 n)$  prin particularizarea căutării într-un **arbore-B**.
  - Astfel căutarea unei chei  $x$  se realizează simplu parcurgând arborele de la rădăcină spre nodurile terminale.
  - În **fiecare** nod parcurs care conține cheile  $(a, -)$  sau  $(a, b)$  se compară  $x$  cu  $a$ .
    - (1) Dacă  $x < a$  se trece la **primul fiu** al nodului.
    - (2) Dacă  $x \geq a$  și nodul are numai 2 fii se trece la **fiul al doilea**.
    - (3) Dacă nodul în cauză are 3 fii se compară  $x$  cu  $b$  și dacă  $x < b$  se trece la **fiul al doilea** al nodului, altfel se trece la **fiul al treilea**.
  - Elementul  $x$  există în arbore dacă și numai dacă există un nod terminal cu cheia  $x$ .
  - Căutarea se poate opri la depistarea unor egalități de genul  $x=a$  sau  $x=b$  în cadrul **nodurilor interioare** dacă se urmărește numai **apartenența**, dar trebuie să continue până la depistarea nodului terminal dacă se dorează prelucrarea acestuia.
- Implementarea acestei reprezentări se poate realiza în limbajul PASCAL utilizând structura de tip articol cu variante (secvența [8.9.4.1.a]).

---

#### {Implementarea arborilor-B binari (2-3)}

```

type TipElement = record
    cheie: REAL;
    {alte câmpuri utile}
end;

FelNod = (terminal,interior);

RefNodArbore2-3 = ^TipNodArbore2-3;           [8.9.4.1.a]

TipNodArbore2-3 = record
  case fel: FelNod of
    terminal: (element: TipElement);
    interior: (cheieStg,cheieDr: REAL;
                fiuUnu,fiuDoi,fiuTrei:
                RefNodArbore2-3)
  end {case}
end; {record}

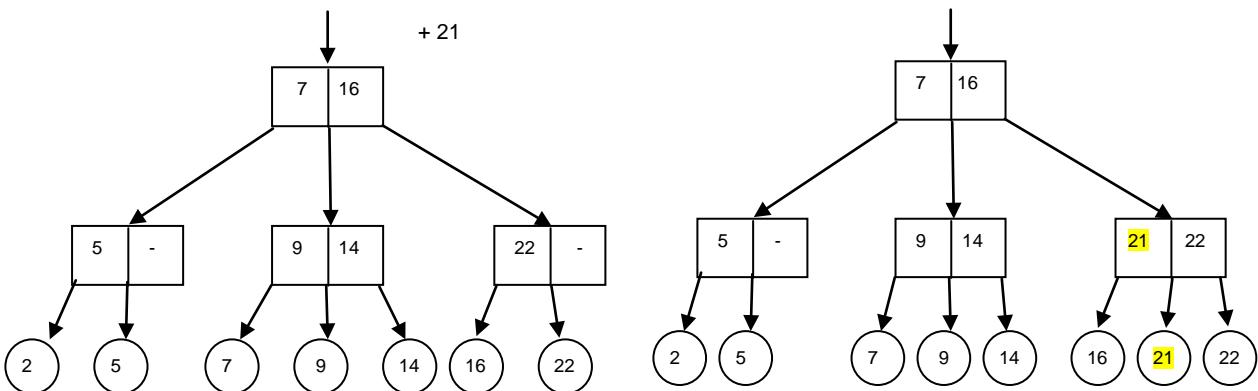
```

---

#### 8.9.4.2. Inserția nodurilor în arbori 2-3

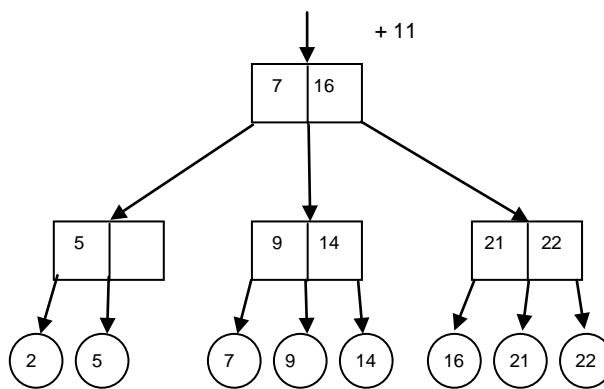
- Inserția unui nod nou într-un arbore 2-3 se realizează principal ca și inserția într-un arbore-B.
- Inserția unui nod cu cheia  $x$  începe cu căutarea cheii  $x$  în arbore.

- Dacă se ajunge la un nod aparținând nivelului situat chiar deasupra nivelului terminal și se constată ca fii acestuia nu îl conțin pe x, se procedează la inserție.
- Sunt posibile următoarele situații:
  - (1) Dacă nodul respectiv are numai **două fii**, se face x cel de-a treilea fiu al nodului, plasându-l la locul potrivit și reajustând structura nodului astfel încât acesta să reflecte noua situație.
    - În figura 8.9.4.2.a este prezentată inserția nodului cu cheia 21 în structura de arbore din figura 8.9.4.1.a, inserție realizată în maniera mai sus precizată.

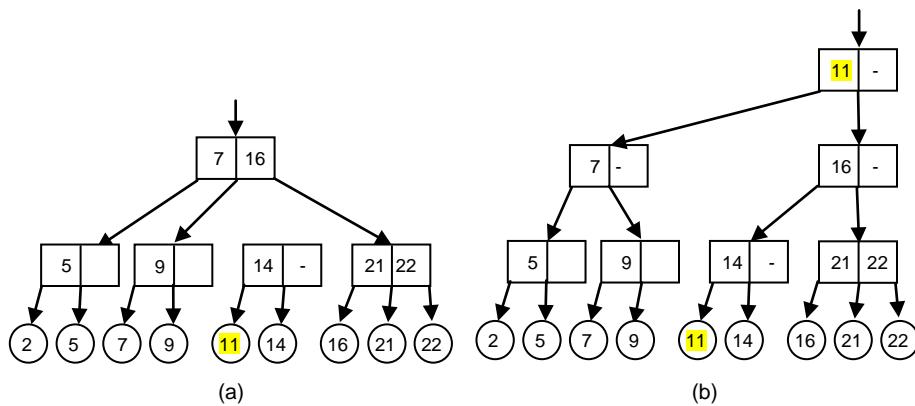


**Fig.8.9.4.2.a.** Inserția unui nod nou într-un arbore 2-3

- (2) Dacă nodul considerat are **trei fii** atunci adăugarea celui de-al patrulea fiu fiind imposibilă, nodul se **scindează** în două noduri, iar cei patru fii sunt redistribuiți:
  - Cei doi fii cu chei mai mici, fostului nod.
  - Ceilalți doi fii cu chei mai mari, nodului nou apărut prin scindare.
- În continuare nodul nou apărut trebuie inserat în structura arbore.
- În urma acestui proces **scindarea** se poate propaga spre nivelurile superioare ale structurii, în cazul extrem până la rădăcină.
  - Aceasta este de fapt singura posibilitate ca un arbore 2-3 să crească în înălțime.
- În figura 8.9.4.2.b se prezintă inserția nodului cu cheia 11 în arborele 2-3 din fig. 8.9.4.2.a.
  - După cum se observă, inserția se realizează în două etape.
  - (1) În prima etapă are loc scindarea nodului (9, 14) în două noduri (9, -) respectiv (14, -) cu repartizarea corespunzătoare a nodurilor terminale (fig. 8.9.4.2.b (a)).
  - (2) În etapa următoare se inserează nodul nou creat în structura de arbore, lucru care necesită o nouă scindare, de data aceasta a nodului rădăcină și crearea unei noi rădăcini (fig. 8.9.4.2.b (b)).



**Fig.8.9.4.2.a.** Inserția unui nod nou într-un arbore 2-3 (reluare)



**Fig.8.9.4.2.b.** Inserție într-un arbore binar 2-3

- Deși principial inserția se realizează simplu, implementarea sa efectivă presupune o serie de **detalii**.
- În cele ce urmează se vor prezenta două proceduri:
  - (1) Procedura **Insertie** care se apelează pentru rădăcină.
  - (2) Procedura **Insertiel** care se apelează pentru celelalte noduri ale structurii și care parcurge arborele în manieră recursivă.
- Se presupune că arborele 2-3 în care se realizează inserția **nu** este nici arbore vid nici arbore cu un singur nod intern.
  - În aceste două cazuri inserția se realizează direct de către procedura **Insertie**.
- Procedura **Insertiel** returnează **pointerul la nodul nou creat** (dacă se crează un astfel de nod) și **cheia celui mai mic element** al subarborelui determinat de acest nod.
  - Acest lucru este realizat cu ajutorul parametrilor **Pnou** respectiv **Mic** care se atribuie în situația în care se returnează un nod nou.
- Structura de principiu a procedurii **Insertiel** apare în secvența [8.9.4.2.a] iar implementarea detaliată în secvența [8.9.4.2.b].
- Se precizează faptul că în secvența [8.9.4.2.a] din motive de reducere a complexității, o serie de detalii ale procesului de inserție, care apar de altfel în secvența [8.9.4.2.b] sunt omise.

```

/*Insertia în arborii 2-3*/

/*Valabilă pentru toate nodurile cu excepția nodului
rădacină*/
/*Schița de principiu a algoritmului - varianta pseudocod*/



Subprogram Insertie1(RefNodArbore2-3 Nod, TipElement X;
                         var RefNodArbore2-3 Pnou:, var REAL Mic);
    /* X - element care urmează a fi inserat în
       subarborele Nod*/
    /* Pnou - pointer la nodul nou creat la dreapta lui Nod
       (dacă este cazul)*/
    /* Mic - cheia celui mai mic element al subarborelui
       indicat de Pnou */

RefNodArbore2-3 Pprim,W;
REAL MicPrim;

Pnou= NULL;
daca (Nod este terminal)
    daca X nu este elementul lui Nod
        /*creează un nod nou terminal indicat de Pnou;
         *atribuie pe X noului nod;
         Mic= X.cheie;
         □ /*daca*/
         □ /*daca*/
altfel [8.9.4.2.a]
    /*Nod este un nod interior*/
    *fie W acel fiu al lui Nod, căruia îi va apartine X;
    Insertie1(W,X,Pprim,MicPrim);
    daca (Pprim<>NULL) /*a fost creat un nod nou*/
        *inserează pointerul Pprim printre fiile lui
          Nod chiar în dreapta lui W;
        daca Nod are 4 fiile
            /*creează un nod nou indicat de Pnou;
             *asociază acestui nod fiile trei și patru
               ai lui Nod;
             *actualizează valorile cheieStg și
               cheieDr în Nod și în noul nod;
             *asignează pe Mic cu cea mai mică
               cheie aparținând fiilor nouului nod
            □ /*daca*/
            □ /*daca*/
        □ /*altfel*/
    revenire;
/*Insertie1*/
-----  

{Insertia în arborii 2-3}

{Valabilă pentru toate nodurile cu excepția nodului
radacină}
{Schița de principiu a algoritmului - varianta PASCAL like}

procedure Insertie1(Nod: RefNodArbore2-3; X: TipElement;
                         var Pnou: RefNodArbore2-3; var Mic: REAL);
    {X - element care urmează a fi inserat în
      subarborele Nod
      Pnou - pointer la nodul nou creat la dreapta lui Nod
      Mic - cheia celui mai mic element al subarborelui
            indicat de Pnou}

```

(dacă este cazul)

Mic - cheia celui mai mic element al subarborelui indicat de Pnou}

```

var Pprim,W: RefNodArbore2-3; MicPrim: REAL;

begin
  Pnou= nil;
  if Nod este terminal then
    begin
      if X nu este elementul lui Nod then
        begin
          *creează un nod nou terminal indicat de Pnou;
          *atribuie pe X noului nod;
          Mic:= X.cheie
        end
      end
    else [8.9.4.2.a]
      begin {Nod este un nod interior}
        *fie W acel fiu al lui Nod, căruia îi va
        apartine X;
        Insertiel(W,X,Pprim,MicPrim);
        if Pprim <> nil then {a fost creat un nod nou}
          begin
            *inserează pointerul Pprim printre fiii lui
            Nod chiar în dreapta lui W;
            if Nod are 4 fii then
              begin
                *creează un nod nou indicat de Pnou;
                *asociază acestui nod fiii trei și patru
                  ai lui Nod;
                *actualizează valorile cheieStg și
                  cheieDr în Nod și în noul nod;
                *asignează pe Mic cu cea mai mică
                  cheie aparținând fiilor noului nod
              end
            end
          end
        end
      end; {Insertiel}
-----
```

### {Insertia normală în arbori 2-3 - varianta PASCAL}

{Valabilă pentru toate nodurile cu excepția nodului rădacină}

```

procedure Insertiel(Nod: RefNodArbore2-3;X: TipElement;
                      var Pnou: RefNodArbore2-3; var Mic: REAL);

var Pprim: RefNodArbore2-3;
  MicPrim: REAL;
  Fiu: 1..3; {indică care dintre fiii nodului este
              selectat în apelul recursiv}
  W: RefNodArbore2-3; {pointer la fiu}

begin
  Pnou := nil;
  if Nod^.fel=terminal then {Nod este un nod terminal}
    begin
```

```

if Nod^.element.cheie<>X.cheie then
    begin {se creează un nod nou care-l conține pe
        X și se returnează pointerul acestui nod}
        NEW(Pnou);
        Pnou^.fel:= terminal;
        if Nod^.element.cheie<X.cheie then
            begin {X se plasează în noul nod care este
                situat la dreapta nodului curent,
                fiind mai mare}
                Pnou^.element:= X; Mic:= X.cheie
            end
        else
            begin {X se plasează în stânga elementului
                din nodul curent prin interschimbare}
                Pnou^.element:= Nod^.element;
                Nod^.element:= X;
                Mic:= Pnou^.element.cheie
            end
        end
    end [8.9.4.2.b]
end
else
    begin {Nod este un nod interior}
        {se selectează fiul corespunzător W pentru
        parcurgere}
        if X.cheie<Nod^.cheieStg then
            begin
                Fiu:= 1; W:=Nod^.fiuUnu
            end
        else
            if(Nod^.fiuTrei=nil)or
                (X.cheie<Nod^.cheieDr) then
                    begin {X aparține celui de-al doilea
                        subarbore}
                        Fiu:= 2; W:= Nod^.fiuDoi
                    end
                else
                    begin {X aparține celui de-al treilea
                        subarbore}
                        Fiu:= 3; W:= Nod^.fiuTrei
                    end;
    Insertiel(W,X,Pprim,MicPrim);
    if Pprim<>nil then
        {trebuie inserat noul fiu al lui Nod}
        if Nod^.fiuTrei=nil then
            {Nod are numai doi fii; noul nod se inserează
            la locul potrivit}
            if Fiu=2 then {Fiu=2}
                begin
                    Nod^.fiuTrei:= Pprim;
                    Nod^.cheieDr:= MicPrim
                end
            else
                begin {Fiu=1}
                    Nod^.fiuTrei:= Nod^.fiuDoi;
                    Nod^.cheieDr:= Nod^.cheieStg;
                    Nod^.fiuDoi:= Pprim;
                    Nod^.cheieStg:= MicPrim
                end

```

```

else [8.9.4.2.b]
    begin {Nod are trei fii}
        NEW(Pnou); {se creaza un nod nou}
        Pnou^.fel := interior;
        if Fiu=3 then
            begin {Pprim și fiul trei devin fiii
                nouului nod}
                Pnou^.fiuUnu:= Nod^.fiuTrei;
                Pnou^.fiuDoi:= Pprim;
                Pnou^.fiuTrei:= nil;
                Pnou^.cheieStg:= MicPrim; {cheieDr
                    este nedefinită pentru Pnou}
                Mic:= Nod^.cheieDr;
                Nod^.fiuTrei:= nil
            end
        else
            begin {Fiu<=2; Se mută fiul trei al lui
                Nod în Pnou}
                Pnou^.fiu_2:= Nod^.fiuTrei;
                Pnou^.cheieStg:= Nod^.cheieDr;
                Pnou^.fiuTrei:= nil;
                Nod^.fiuTrei:= nil
            end;
        if Fiu=2 then
            begin {Pprim devine primul fiu al lui
                Pnou}
                Pnou^.fiuUnu:= Pprim; [8.9.4.2.b]
                Mic:= MicPrim
            end;
        if Fiu=1 then
            begin {fiul doi al lui Nod se mută în
                Pnou; Pprim devine fiul doi al
                lui Nod}
                Pnou^.fiuUnu:= Nod^.fiuDoi;
                Mic:= Nod^.cheieStg;
                Nod^.fiuDoi:= Pprim;
                Nod^.cheieStg:= MicPrim
            end
        end
    end
end; {Insertiel}
-----
```

- În continuare se trece la redactarea procedurii **Insertie** care apelează procedura **Insertiel**.
- Dacă **Insertiel** returnează un nod nou, atunci **Insertie** trebuie să creeze o nouă rădăcină.
  - Codul procedurii apare în secvența [8.9.4.2.c].

---

**{Insertia rădăcinii arborilor 2-3 - varianta PASCAL}**

```

procedure Insertie(X: TipElement; var M: RefNodArbore2-3);
    var Pprim: RefNodArbore2-3; {pointer la nodul nou
                                returnat de Insertiel}
    MicPrim: REAL; {valoarea Mic a subarborelui
                    determinat de Pprim}
```

```

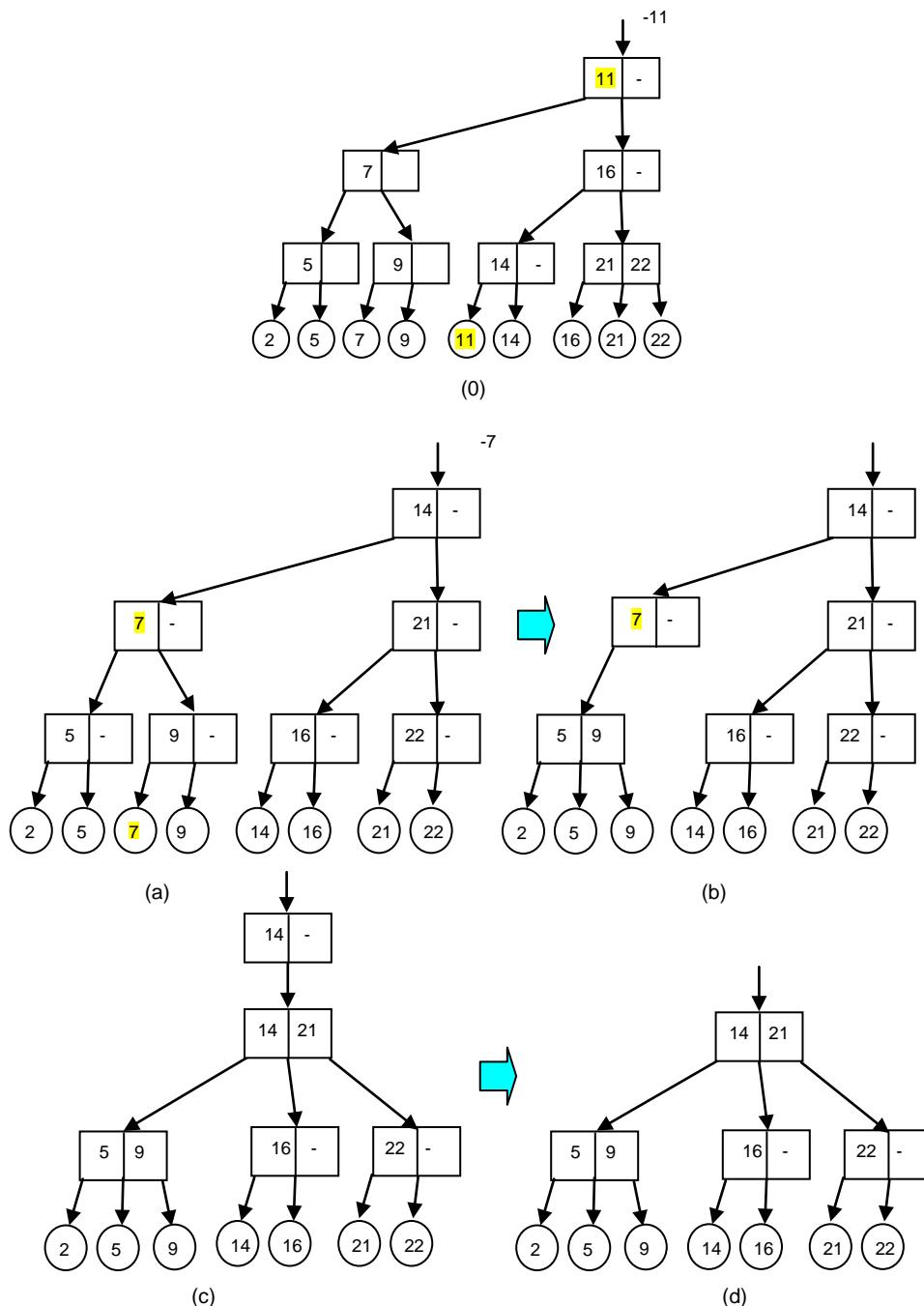
Temp: RefNodArbore2-3; {utilizat pentru memorarea
                      temporară a valorii pointerului M}

begin
  if M este arbore vid sau arbore cu un singur nod then
    *se realizează direct inserția;
  else
    begin
      Insertie1(M,X,Pprim,MicPrim);
      if Pprim <> nil then
        begin {se creează o nouă rădăcină; fiile
                  rădăcinii sunt cei indicați de M
                  respectiv Pprim}
        Temp:= M;                               [8.9.4.2.c]
        NEW(M);
        M^.fel:= interior;
        M^.fiuUnu:= Temp;
        M^.fiuDoi:= Pprim;
        M^.cheieStg:= MicPrim;
        M^.fiuTrei:= nil
      end
    end
  end; {Insertie}
-----
```

#### 8.9.4.3. Suprimarea nodurilor în arbori 2-3

- Suprimarea nodurilor în arbori 2-3 se realizează într-o manieră similară suprimării în arbori-B.
- Se presupune că se dă un arbore 2-3 și o cheie  $x$  și se cere să se suprime nodul având cheia egală cu  $x$ .
- În prealabil se caută nodul cu cheia precizată după care se trece la suprimarea propriu-zisă.
- Astfel, fie  $n$  părintele nodului de suprimat și fie  $p$  părintele lui  $n$  (bunicul nodului de suprimat). Sunt posibile mai multe situații:
  - (1) Dacă  $n$  (părintele nodului de suprimat) are **trei fii** suprimarea se realizează simplu prin reorganizarea nodului indicat de  $n$ .
  - (2) Dacă  $n$  (părintele nodului de suprimat) are **doi fii** în urma suprimării  $n$  rămâne cu un singur fiu, adică apare fenomenul cunoscut sub numele de **"subdepășire"**, arboarele 2-3 trebuie reorganizat situație în care există trei cazuri:
    - (a) Dacă  $n$  este chiar nodul **rădăcină**, după suprimare, singurul său fiu va deveni noua rădăcină.
    - (b) Dacă  $p$  (bunicul nodului de suprimat) mai are un fiu situat la stânga sau la dreapta lui  $n$  și acest fiu are la rândul său 3 fii, în cadrul unui proces denumit **"echilibrare"** se va realiza **"împrumutul"** unui fiu astfel încât  $n$  să aibă doi fii.
    - (c) Dacă însă fiul sau fiile lui  $p$ , adiacenți lui  $n$  au la rândul lor numai câte doi fii, echilibrarea **nu** se mai poate realiza și în consecință se transferă singurul fiu al lui  $n$  unui frate adiacent și se suprimă  $n$ .

- Dacă în urma acestui proces denumit “**cotopire**” p rămâne cu un singur fiu, se repetă procedura în mod recursiv substituind pe n cu p, trecând astfel pe nivelul superior al structurii arborelui.
- În caz extrem acest proces se poate propaga până la rădăcină, aceasta fiind practic singura cale de reducere a înălțimii arborelui.
- Spre exemplu se presupune că se dorește suprimarea nodului cu cheia 11 din arborele din figura 8.9.4.3.a.(0).
  - Dacă nodul terminal 11 este suprimit, părintele său (14, -) rămâne cu un singur fiu.
  - În această situație se poate realiza **împrumutul** de la fratele din dreapta al nodului (14, -) care are trei fii.
  - Astfel este împrumutat nodul 16 și în urma reactualizării cheilor rezultă arborele din figura 8.9.4.3.a.(a).



### Fig.8.9.4.3.a. Suprimarea nodurilor într-un arbore 2-3

- În continuare se procedează la suprimarea nodului cu cheia 7 din arborele din figura 8.9.4.3.(a).
  - În consecință părintele său (9, -) rămâne cu un singur fiu.
  - Întrucât nu se poate realiza împrumutul deoarece unicul frate al nodului în cauză, adică cel situat în stânga sa, are numai doi fii, se procedează la **contopire** prin transferarea cheii 9 și suprimarea nodului (9, -).
  - În urma acestui proces rezultă arborele (b) din aceeași figură.
  - După cum se observă nodul interior (7, -) are un singur fiu iar fratele său, nodul (21, -) are numai doi fii, în consecință se aplică din nou **contopirea** cu reactualizarea cheilor, obținându-se un nod cu trei fii (fig.8.9.4.3.(c)).
  - Acum rădăcina are un singur fiu, deci ea poate fi suprimată și înlocuită cu acest fiu (fig.8.9.4.3(d)).
- Din cele prezentate rezultă faptul că prelucrarea arborilor 2-3 necesită manipularea frecventă a valorilor memorate în nodurile interioare.
- Valorile necesare în procesul de prelucrare pot fi determinate în două moduri:
  - (1) Parcurgând arboarele pentru fiecare valoare.
  - (2) Reținând în timpul prelucrării arborelui **cea mai mică valoare** a **descendenților fiecărui nod** situat pe drumul de la rădăcină spre nodul de suprimat.
    - Această informație poate fi determinată simplu pentru fiecare nod parcurs în cadrul unui **algoritm recursiv** de **suprimare**.
- În cele ce urmează se va prezenta structura de principiu a unei funcții **Suprima1** (secvența (8.9.4.3.a)) care primește ca parametri un pointer la un nod de arbore 2-3 (Nod) și o cheie x.
  - Sarcina funcției **Suprima1** este aceea de a suprima nodul terminal cu cheia x dintre descendenții nodului precizat, dacă există un astfel de nod.
  - Pentru a se atinge acest obiectiv, în cadrul funcției se realizează o **parcurgere recursivă** a structurii arborelui în vederea determinării nodului terminal x, după care se revine pe drumul parcurs restructurând după necesități arboarele.
  - După fiecare apel, funcția **Suprima1** returnează **valoarea adevărat** dacă după restructurare Nod rămâne cu **un singur fiu** respectiv returnează **valoarea fals** dacă Nod rămâne cu doi sau trei fii.

---

{**Suprimarea în arbori 2-3**}

{Schița de principiu a algoritmului – varianta PASCAL like}

```
function Suprima1(Nod: RefNodArbore2-3; X: TipElement):  
    BOOLEAN;  
var NumaiUnul: BOOLEAN; {memorează valoarea returnată  
                           de un apel al funcției Suprima1}  
begin
```

```

Suprimal:= FALSE;
if fiii lui nod sunt terminali then [8.9.4.3.a]
  begin
    if X este printre acești terminali then
      begin
        *extragă pe X;
        *deplasează fiii lui Nod situați în dreapta
          nodului X cu o poziție spre stânga;
        if Nod are acum un singur fiu then
          Suprimal:= TRUE
        end
      end
    else
      begin {nodul nu are fii terminali el situându-se pe
        un nivel interior}
        *determină acel fiu al lui Nod, care l-ar putea
          avea pe X ca descendant: fie acesta W;
        NumaiUnul:= Suprimal(W,X); {W are valoarea
          Nod^.fiuUnu, Nod^.fiuDoi sau Nod^.fiuTrei
          după cum este cazul}
        if NumaiUnul then [8.9.4.3.a]
          begin {restructurare}
            if W este primul fiu al lui Nod then
              if Y (cel de-al doilea fiu al lui Nod) are
                trei copii then
                  *împrumută primul fiu al lui Y și
                    transferă-l drept al doilea fiu al lui W
              else
                begin {Y are doi fii, deci contopire}
                  *transferă fiul lui W, drept prim fiu
                    al lui Y;
                  *extragă pe W dintre fiii lui Nod;
                  if Nod are acum un singur fiu then
                    Suprimal:= TRUE
                end;
            if W este cel de-al doilea fiu
              al lui Nod then
              if Y,primul fiu al lui Nod,are
                trei fii then
                  *împrumută cel de-al treilea fiu al lui
                    Y și transferă-l drept primul fiu al
                    lui W
              else {Y are doi fii}
                if Z,cel de-al treilea fiu al lui
                  Nod,există și are trei copii then
                    *împrumută primul fiu al lui Z și
                      transferă-l drept al doilea fiu al
                      lui W
              else
                begin{nici un alt fiu al lui Nod,nu
                  are trei fii}
                  *transferă fiul lui W drept al
                    treilea fiu al lui Y;
                  *extragă pe W dintre fiii lui Nod;
                  if Nod are acum un singur fiu then
                    Suprimal:= TRUE
                end;
            if W este cel de-al treilea fiu al

```

```

        lui Nod then
if Y, cel de-al doilea fiu al lui Nod, are
        trei fii then
            *împrumută cel de-al treilea fiu al lui
            Y și transferă-l drept primul fiu al
            lui W
        else [8.9.4.3.a]
            begin {Y are doi fii}
                *transferă fiul lui W drept cel de-al
                treilea fiu al lui Y;
                *extrage pe W dintre fii lui Nod
            end {în acest caz lui Nod îi rămân cu
                siguranță doi fii}
        end
    end
end; {Suprima1}
-----
```

- Codul detaliat al funcției **Suprima1** este propus drept exercițiu.
- Un alt exercițiu propus este acela de a redacta procedura **Suprima**(m, x) care:
  - Verifică și rezolvă direct cazurile speciale în care arborele m este vid sau conține un singur element.
  - Dacă arborele conține mai multe noduri, apelează funcția **Suprima1**.
  - Dacă **Suprima1** returnează adevărat, procedura **Suprima**(m, x) suprimă rădăcina (nodul idicat de m) și face pe m să indice pe fiul unic al rădăcinii.

## 9. Structura de date mulțime

### 9.1. Introducere

- O altă structură de date considerată uneori fundamentală altelei avansată, este **structura mulțime**, care se definește generic astfel:

```
-----  
TYPE TipMultime = SET OF TipDeBaza; [9.1.a]  
-----
```

- Valorile posibile ale unei variabile  $x$  a tipului TipMultime, sunt mulțimi de elemente ale lui TipDeBaza.
- Vom numi **mulțime de bază** mulțimea tuturor elementelor lui TipDeBaza.
- În aceste condiții, **mulțimea tuturor submulțimilor** de elemente ale lui TipDeBaza formează **puterea mulțimii de bază**.
- Tipul TipMultime are ca domeniu de valori **puterea mulțimii de bază** asociată lui TipDeBaza.
- Cu alte cuvinte fiind dată mulțimea de bază, prin **mulțime** vom înțelege **orice submulțime** a acesteia, inclusiv mulțimea vidă.
  - Spre exemplu dacă se alege drept mulțime de bază  $\{a, b, c\}$ , atunci se pot utiliza următoarele opt submulțimi drept constante ale tipului mulțime asociat tipului de bază [9.1.b].

```
-----  
TYPE TipMultime = SET OF (a,b,c); [9.1.b]  
[]; [a]; [b]; [c]; [a,b]; [a,c]; [b,c]; [a,b,c];  
-----
```

- Cardinalitatea unui tip mulțime este:  
$$\text{Card}(\text{TipMultime}) = 2^{\text{Card}(\text{TipDeBaza})}$$
  - Această formulă poate fi dedusă simplu din faptul că fiecare dintre elementele lui TipDeBaza (al căror număr este egal cu cardinalitatea lui TipDeBaza), poate fi reprezentat printr-o singură valoare "absent" sau "present" și că toate elementele sunt independente unele față de altele.
- În manieră clasica peste un tip mulțime se definesc următoarele legi de compozitie internă, valabile pentru două mulțimi de același tip: **atribuire, reunire, scădere, intersecție, negare**.
- De asemenea se mai definesc și operatorii relaționali care conduc la valori booleene: **egalitate, inegalitate, incluziune, incluziune inversă**.
- În sfârșit, dacă  $m$  este instanță a lui TipDeBaza și  $m$  este o variabilă de tip mulțime

având asociat același tip de bază, atunci este definită și **relația de apartenență** a lui e la m.

- Capitolul de față își propune:
  - (1) Să extindă setul de operatorii clasici și asupra altor categorii de mulțimi cu caracter mai deosebit.
  - (2) Să studieze și să prezinte câteva dintre posibilitățile de implementare ale acestui tip de date abstract.
- Deși în matematică **noțiunea de mulțime** nu se definește fiind considerată o **noțiune primară**, în cadrul cursului de față vom înțelege prin **mulțime** o colecție de elemente.
  - Fiecare element al unei mulțimi poate fi la rândul său o mulțime sau un element primitiv numit **atom**.
  - Toate elementele unei mulțimi sunt diferite, adică o mulțime **nu conține două copii** ale aceluiași element.
  - Atunci când sunt utilizări în proiectarea algoritmilor și a structurilor de date, **atomii** sunt de regulă întregi, caractere, sau siruri de caractere.
  - În orice mulțime toate elementele sunt de același **tip**.
  - De multe ori, se consideră că atomii sunt **ordonăți liniar** printr-o relație de precedență. [Cr87].

## 9.2. Tipul de date abstract mulțime

- Considerând mulțimea un **tip de date abstract**, asupra ei pot fi imaginate diferite tipuri de operații derive din operațiile clasice definite asupra mulțimilor.
- În continuare pentru aplicațiile avute în vedere se consideră urmatorul **set de operatori** [AH85].

---

### Tipul de date abstract Mulțime

**Modelul matematic:** Mulțime definită în sens matematic.

**Notății:**

[9.2.a]

<i>TipMultime</i>	- tipul de date abstract mulțime
<i>TipElement</i>	- tipul de date asociat elementelor mulțimii (tipul de bază)
<i>A, B, C</i>	- mulțimi încadrate în <i>TipMultime</i>
<i>x</i>	- valoare (obiect) de <i>TipElement</i>
<i>b</i>	- valoare booleană

**Operatori:**

1. **Reuniune**(*TipMultime A, TipMultime B, TipMultime C*) - operator care primește ca date de intrare multimile *A* și *B* și atribuie rezultatul  $A \cup B$  variabilei multime *C*.
2. **Intersectie**(*TipMultime A, TipMultime B, TipMultime C*) - operator care primește ca date de intrare multimile *A* și *B* și atribuie rezultatul  $A \cap B$  variabilei multime *C*.
3. **Diferenta**(*TipMultime A, TipMultime B, TipMultime C*) - operator care primește ca date de intrare multimile *A* și *B* și atribuie rezultatul  $A - B$  variabilei multime *C*.
4. **Uniune**(*TipMultime A, TipMultime B, TipMultime C*) - definește operatorul uniune adică reuniunea multimilor disjuncte. Cu alte cuvinte operatorul atribuie variabilei multime *C* valoarea  $A \cup B$ . *C* nu este definită dacă  $A \cap B \neq \emptyset$ , adică dacă multimile *A* și *B* nu sunt disjuncte.
5. **boolean Apartine**(*TipElement x, TipMultime A*) - operator care primește ca parametri de intrare elementul *x* al cărui tip este tipul de bază al multimii *A* și multimea *A*, și returnează o valoare booleană - adevărat sau fals - după cum *x* aparține sau nu multimii *A*.
6. **Vid**(*TipMultime A*) - operator care atribuie multimii *A*, multimea vidă.
7. **Adauga**(*TipElement x, TipMultime A*) - unde *A* este o variabilă de tip multime iar *x* un element al cărui tip este identic cu tipul elementelor lui *A*. Operatorul face din *x* un element al lui *A* adică noua valoare a lui *A* este  $A \cup \{x\}$ . Dacă *x* este deja membru al multimii *A*, operatorul nu-l modifică pe *A*.
8. **Suprima**(*TipElement x, TipMultime A*) - operator care extrage atomul *x* din *A*, adică *A* este înlocuit cu  $A - \{x\}$ . Dacă *x* nu aparține multimii originale *A*, operatorul nu modifică valoarea lui *A*.
9. **Atribuie**(*TipMultime A, TipMultime B*) - operator care face ca valoarea variabilei multime *A* să fie egală cu valoarea variabilei multimi *B*, adică îl atribuie pe *B* lui *A* ( $A=B$ ).
10. **TipElement Min**(*TipMultime A*) -operator care returnează cel mai mic element al multimii *A*. În mod similar **Max**(*A*) returnează cel mai mare element al multimii *A*. Aceste operații pot fi aplicate numai multimilor ale căror elemente pot fi ordonate liniar printr-o relație de precedență.

11. boolean **Egal**(*TipMultime A, TipMultime B*) - operator care returnează valoarea adevărată dacă și numai dacă mulțimile *A* și *B* sunt egale.
12. *TipMultime Caută(TipElement x)* - operator care operează asupra unei colecții de mulțimi distincte. **Caută(x)** returnează mulțimea (unică) căreia îi aparține elementul *x*.
- 

## 9.3. Implementarea structurii mulțime utilizând structuri de date fundamentale

### 9.3.1. Implementarea structuri mulțime cu ajutorul vectorilor binari

- O implementare performantă a unei structuri de date abstrakte **mulțime** depinde:
    - De **operațiile** care vor fi realizate asupra structurii.
    - De **dimensiunea mulțimii** (cardinalitatea acesteia).
  - Când **mulțimile** cu care se lucrează sunt **submulțimi** ale unei “mulțimi de bază” de mici dimensiuni ale cărei elemente sunt întregii  $0, 1, 2, \dots, n-1$ , cu un *n* precizat, atunci în implementare se poate utiliza **vectorul binar** (“bit-vector”) sau **tabloul boolean**.
  - O astfel de mulțime poate fi reprezentată printr-un **vector binar** pe baza **funcției sale caracteristice**.
    - **Funcția caracteristică** definită pe mulțimea indicilor vectorului cu valori în boolean, precizează că cel de-al *i*-lea bit al vectorului este adevărat (are valoarea 1) dacă *i* este un element al mulțimii [Cr87].
  - Referitor la această implementare se pot face următoarele precizări:
    - (1) Operațiile **Apartine**, **Adaugă** și **Suprimă** pot fi realizate într-un interval constant de timp, adresând direct bitul corespunzător.
    - (2) Operațiile **Reuniune**, **Intersecție**, și **Diferență** pot fi realizate într-un timp proporțional cu dimensiunea mulțimii de bază (cardinalitatea acesteia).
    - (3) Dacă această cardinalitate este suficient de redusă, astfel încât vectorul binar se suprapune ca dimensiune peste un cuvânt de calculator, atunci operațiile **Reuniune**, **Intersecție**, și **Diferență** pot fi realizate printr-o singură operație logică cablată.
  - În limbajul PASCAL, anumite mulțimi de dimensiuni reduse pot fi generate cu ajutorul constructorului **SET**. Dimensiunea maximă a unei astfel de mulțimi depinde de compilatorul utilizat.
  - În general, pentru aplicații care utilizează mulțimi care sunt submulțimi ale unei mulțimi de bază universale  $0, 1, 2, \dots, n-1$ , cu *n* depinzând de aplicație, utilizând **vectorul binar** drept suport se poate defini următorul **tip mulțime** [9.3.1.a].
-

```

/*Definirea unui TipMultime utilizând vectorul binar -
varianta C*/
int NumarElemente=n;

typedef boolean TipMultime[NumarElemente]; /*[9.3.1.a]*/

```

TipMultime A;

---

{Definirea unui TipMultime utilizând vectorul binar -
varianta PASCAL}

```

type TipMultime=array[0..n-1] of boolean; [9.3.1.a]

```

**var** A: TipMultime;

---

- Dacă A este o variabilă a tipului TipMultime, atunci A[i] este adevărat dacă și numai dacă i aparține mulțimii A.

- Operația **Reuniune** poate fi implementată ca și în secvența [9.3.1.b].
- 

/\*Implementarea operatorului Reuniune (Performanță O(n)) -
Varianta pseudocod\*/

```

subprogram Reuniune(TipMultime A, TipMultime B, TipMultime
C);

```

```

    int i;
    pentru (i=0 la n-1) [9.3.1.b]
        C[i]= A[i] || B[i];

```

---

{Implementarea operatorului Reuniune (Performanță O(n)) -
Varianta PASCAL}

```

procedure Reuniune(A,B: TipMultime; var C: TipMultime);

```

```

    var i: integer;
    begin
        for i:= 0 to n-1 do [9.3.1.b]
            C[i]:= A[i] or B[i]
    end;

```

---

- Operatorii **Intersectie** și **Diferenta** rezultă imediat înlocuind operația “**or**” (“||”) cu “**and**” (“&&”) respectiv “**and not**” (“&& !”).

- Într-o manieră similară, se pot implementa și ceilalți operatori precizați în &9.2

- Operatorii **Uniune** și **Caută** nu au sens în acest context.

- **Vectorii binari** pot fi utilizați în implementarea mulțimilor și în situația în care elementele acestora **nu** sunt întregi consecutivi.

- În acest caz trebuie stabilită o **corespondență** între **elementele mulțimii** și o **submulțime convenabilă** a numerelor **întregi**.
  - Pentru acest scop poate fi utilizată o structură de date de tip **asociere** care permite stabilirea corespondenței în ambele sensuri (Vol.1 &6.8).
  - **Recomandare:** de regulă corespondența “întregi – elemente ale mulțimii” poate fi cel mai eficient implementată cu ajutorul unui **tablou A**, unde  $A[i]$  este elementul corespunzător întregului  $i$ .

### 9.3.2. Implementarea structurii mulțime cu ajutorul listelor înlățuite

- Structura de date mulțime poate fi implementată și cu ajutorul unei **liste înlățuite**, unde nodurile listei sunt elemente ale mulțimii.
- Reprezentarea bazată pe liste are unele **avantaje** față de reprezentarea bazată pe vectori binari:
  - (1) Utilizează **spațiul de memorie** strict necesar mulțimii și nu spațiul corespunzător “mulțimii de bază”.
  - (2) Este mai **generală** întrucât poate manipula mulțimi care nu sunt supuse nici unei constrângeri.
- Pentru exemplificare se prezintă implementarea operatorului **Intersecție** în cazul mulțimilor reprezentate prin **liste înlățuite simple**.
  - Dacă listele **nu** sunt ordonate, trebuie verificată pentru fiecare element al lui  $L_1$  concordanța cu fiecare element al lui  $L_2$  parcurgând de fiecare dată integral pe  $L_2$ , proces a cărui regie este  $O(n^2)$ .
  - Dacă mulțimea de bază este **ordonată liniar**, atunci ea poate fi reprezentată printr-o listă ordonată.
    - Avantajul unei astfel de implementări este acela că **nu** este necesară parcurgerea întregii liste pentru a determina dacă un element aparține sau nu acesteia.
  - Un element aparține **intersecției** a două liste  $L_1$  și  $L_2$  dacă și numai dacă se găsește în ambele liste.
  - Întrucât cele două liste sunt **ordonate**, pentru fiecare element  $e$  al lui  $L_1$  se parcurge  $L_2$  până la găsirea:
    - (1) Unui element identic, caz în care elementul se adaugă intersecției.
    - (2) Unui element mai mare, caz în care se trece la elementul următor al lui  $L_1$ .
  - Fiecare dintre cele două liste este parcursă cu ajutorul unui **indicator de poziție** specific.
  - Cu alte cuvinte, **intersecția a două mulțimi** reprezentate prin listele ordonate  $L_1$  și  $L_2$  poate fi determinată într-o **singură parcurgere secvențială** a celor două liste, avansând întotdeauna indicatorul de poziție corespunzător listei care conține cel mai mic element curent.
  - Procedura care implementează această tehnică apare în secvența [9.3.2.b].

- Mulțimile avute în vedere sunt implementate ca și liste înlățuite ale căror noduri aparțin tipului nod definit ca și în secvența [9.3.2.a]:

```

-----/*Definirea elementelor mulțimii reprezentate prin liste
înlățuite ordonate -Structuri de date - varianta C*/
-----typedef TipNod * ref_nod;

typedef struct TipNod
{
    TipElement element; /*[9.3.2.a]*/
    ref_nod urm;
} TipNod;

ref_nod incepA, incepB, incepC;
-----/* Determinarea intersecției a două mulțimi - varianta
pseudocod*/

procedure Intersecție(ref_nod incepA, ref_nod incepB,
ref_nod incepC)

/*Determină intersecția mulțimilor A și B reprezentate ca și
liste înlățuite ordonate indicate de pointerii incepA și
incepB. Intersecția se memorează în lista C indicată de
incepC*/

    ref_nod indicA,indicB,indicC; /*pointeri la nodurile
curente în A și B, respectiv la ultimul nod adăugat în C*/
    incepC=aloca_memorie(TipNod); /*creează lista C*/
    indicA=incepA;
    indicB=incepB;
    indicC=incepC;
    cat timp ((indicA != null) && (indicB != null))
        /*compară elementele curente ale listelor A și B*/
        daca(indicA^.element==indicB^.element)
            {se adaugă elementul intersecției}
            indicC^.urm=aloca_memorie(TipNod);
            indicC=indicC^.urm; /*[9.3.2.b]*/
            indicC^.element=indicA^.element;
            indicA=indicA^.urm;
            indicB=indicB^.urm;
        altfel /*elemente diferite*/
            daca(indicA^.element<indicB^.element)
                indicA=indicA^.urm;
            altfel
                indicB=indicB^.urm;
        cat timp /*cat timp*/
            indicC^.urm=null;
    /*Intersecție*/
-----
```

{Definirea elementelor mulțimii reprezentate prin liste  
înlăntuite ordonate - structuri de date - varianta PASCAL}

```
type RefNod=^TipNod;
  TipNod=record
    element: TipElement;
    urm: RefNod
  end; [9.3.2.a]

-----/* Determinarea intersecției a două mulțimi - varianta
PASCAL*/-----
```

**procedure Intersecție**(incepA,incepB: RefNod; **var** incepC:  
RefNod);  
{Determină intersecția mulțimilor A și B reprezentate ca și  
liste înlăntuite ordonate indicate de pointerii incepA și  
incepB. Intersecția se memorează în lista C indicată de  
incepC}

**var** indicA,indicB,indicC: RefNod;  
{pointeri la nodurile curente în A și B, respectiv la  
ultimul nod adăugat în C}

**begin**  
 new(incepC); {creează lista C}  
 indicA := incepA;  
 indicB := incepB;  
 indicC := incepC;  
 **while** (indicA <> nil) **and** (indicB <> nil) **do**  
 **begin**{compară elementele curente  
 ale listelor A și B}  
 **if** indicA^.element = indicB^.element **then**  
 **begin** {se adaugă elementul intersecției}  
 new(indicC^.urm); [9.3.2.b]  
 indicC := indicC^.urm;  
 indicC^.element := indicA^.element;  
 indicA := indicA^.urm;  
 indicB := indicB^.urm  
 **end**  
 **else** {elemente diferite}  
 **if** indicA^.element < indicB^.element **then**  
 indicA := indicA^.urm  
 **else**  
 indicB := indicB^.urm  
 **end**;  
 indicC^.urm := **nil**  
 **end**; {Intersecție}

- În cadrul procedurii **Intersecție**, A și B sunt listele corespunzătoare submulțimilor operanți, iar C lista corespunzătoare mulțimii intersecție care se crează.
- Fiecare listă are un pointer specific: indicA, indicB respectiv indicC.
- Se poziționează pointerii pe începutul listelor specifice.

- În continuare într-o buclă (instructiunea **while**), se compară elementele curente ale listelor A și B.
  - În caz de egalitate, se trece elementul în C și se avansează ambii pointeri în A și B.
  - În caz de inegalitate se avansează pointerul din lista care conține cel mai mic element curent.
- În rutina din secvența [9.3.2.b], se presupune că **TipElement** este un tip ordonat, ale cărui constante se pot compara cu operatorul “>”.
  - Dacă acest lucru nu este valabil, trebuie implementată o funcție care stabilește relația de precedență dintre două elemente.
- Ca și exercițiul se poate realiza implementarea procedurii **Intersecție** utilizând operațiile primitive definite asupra listelor.
- Operațiile **Reuniune** și **Diferență** pot fi implementate prin proceduri foarte apropriate ca formă de procedura **Intersecție**.
  - Pentru **Reuniune** este necesar ca toate elementele din A și B să fie trecute în ordine în C.
    - (1) Procedeul este identic cu cel aplicat la **Intersecție** pentru elementele egale.
    - (2) Pentru elementele diferite, se trece în lista C cel mai mic element.
    - (3) Când se ajunge la sfârșitul uneia din liste (A sau B) trebuie să fie trecute în C restul elementelor aparținând listei neterminate.
  - Pentru operatorul **Diferență**, nu se adaugă la C elemente găsite egale.
    - (1) Se adaugă la C elementul curent al listei A când este găsit mai mic decât elementul curent al listei B.
    - (2) Se adaugă la C elementele care au mai rămas în lista A, când B a fost parcurs integral.
- În secvența [9.3.2.c] apare un exemplu de implementare a structurii mulțime cu ajutorul listelor înlăncuite respectiv operatorii **Adaugare**, **Afisare**, **Intersecție**, **Reuniune** și **Diferență**.

---

//Exemplu de implementare a structurii mulțime utilizând structura listă înlăncuită ordonată

```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```

//structuri de date
typedef int TipElement;

typedef struct TipNod{

    TipElement element;
    struct TipNod *urm;

} TipNod;

typedef TipNod *ref_nod;

ref_nod incepA, incepB, incepC, incepD, incepE, incepF;
//pot fi si locale

ref_nod Adauga(ref_nod incep, TipElement elem)
//functie utilizata pentru a adauga cuvinte elemente in
multimea incep. Creaza o lista ordonata

{
    ref_nod aux, p;

    p = (TipNod*)malloc(sizeof(TipNod));
    p->element = elem;
    p->urm = NULL;

    if (incep == NULL) //lista vida
        incep = p;
    else
    {
        aux = incep;
        if (aux->urm == NULL) //lista cu un element
        {
            if (aux->element < p->element) aux->urm = p;
            else
                if (p->element < aux->element)
                {
                    p->urm = aux;
                    aux->urm = NULL;
                    incep = p;
                }
        }
        else //lista cu mai multe elemente
        {
            while ((aux->urm != NULL) && (aux->element <=
                p->element))
                aux = aux->urm;
            if (aux->element > p->element) //insertie in fata
            {
                *p = *aux;
                aux->urm = p;
                aux->element = elem;
            }
            else
                aux->urm = p; //insertie in spate
        }
    }
}

```

```

        }
        return incep;
    }
//Adaugare

void Afisare(ref_nod incep)
{
    ref_nod aux;
    aux = incep;

    while (aux != NULL)
    {
        printf("%d ", aux->element);
        aux = aux->urm;
    }

    printf("\n");
}
//Afisare

ref_nod Intersectie(ref_nod incepA, ref_nod incepB, ref_nod
incepC)
//intersectia multimilor incepA si incepB. Rezultatul in
incepC care este returnat
{
    ref_nod indicA, indicB, indicC;
    incepC = (TipNod*)malloc(sizeof(TipNod));

    indicA = incepA;
    indicB = incepB;
    indicC = incepC;

    while ((indicA != NULL) && (indicB != NULL))
    {
        if (indicA->element == indicB->element)
        {
            indicC->urm =
(TipNod*)malloc(sizeof(TipNod));
            indicC = indicC->urm;
            indicC->element = indicA->element;
            indicA = indicA->urm;
            indicB = indicB->urm;
        }
        else
        if (indicA->element<indicB->element)
            indicA = indicA->urm;
        else
            indicB = indicB->urm;
    }

    indicC->urm = NULL;
    return incepC->urm;
}
//Intersectie

```

```

ref_nod Reuniune(ref_nod incepA, ref_nod incepB, ref_nod
incepC)
//reuniunea multimilor incepA si incepB. Rezultatul in
incepC care este returnat

{
    ref_nod indicA, indicB, indicC;
    incepC = (TipNod*)malloc(sizeof(TipNod));

    indicA = incepA;
    indicB = incepB;
    indicC = incepC;

    while ((indicA != NULL) && (indicB != NULL))
    {
        indicC->urm = (TipNod*)malloc(sizeof(TipNod));
        indicC = indicC->urm;

        if (indicA->element == indicB->element)
        {
            indicC->element = indicA->element;
            indicA = indicA->urm;
            indicB = indicB->urm;
        }
        else
        {
            if (indicA->element<indicB->element)
            {
                indicC->element = indicA->element;
                indicA = indicA->urm;
            }
            else
            {
                indicC->element = indicB->element;
                indicB = indicB->urm;
            }
        }
    }

    if ((indicA != NULL) || (indicB != NULL))
    {
        indicC->urm = (TipNod*)malloc(sizeof(TipNod));
        indicC = indicC->urm;

        while (indicA != NULL)
        {
            indicC->element = indicA->element;
            indicA = indicA->urm;
        }

        while (indicB != NULL)
        {
            indicC->element = indicB->element;
            indicB = indicB->urm;
        }
    }
}

```

```

        indicC->urm = NULL;
        return incepC->urm;
    }
//Reuniune

ref_nod Diferenta(ref_nod incepA, ref_nod incepB, ref_nod
incepC)
//diferenta multimilor incepA si incepB. Rezultatul in
incepC care este returnat

{
    ref_nod indicA, indicB, indicC;
    incepC = (TipNod*)malloc(sizeof(TipNod));

    indicA = incepA;
    indicB = incepB;
    indicC = incepC;

    while ((indicA != NULL) && (indicB != NULL))
    {
        if (indicA->element == indicB->element)
        {
            indicA = indicA->urm;
            indicB = indicB->urm;
        }
        else
        if (indicA->element<indicB->element)
        {
            indicC->urm =
(TipNod*)malloc(sizeof(TipNod));
            indicC = indicC->urm;
            indicC->element = indicA->element;
            indicA = indicA->urm;
        }
        else
            indicB = indicB->urm;
    }

    while (indicA != NULL)
    {
        indicC->urm = (TipNod*)malloc(sizeof(TipNod));
        indicC = indicC->urm;
        indicC->element = indicA->element;
        indicA = indicA->urm;
    }

    indicC->urm = NULL;
    return incepC->urm;
}
//Diferenta
-----
```

- Operația **Atribuie**(*A,B*) presupune copierea listei *A* în lista *B*.

- Trebuie subliniat faptul că **nu** este suficient ca acest lucru să fie implementat prin simpla modificare a pointerului care-l indică pe B dându-i pur și simplu valoarea pointerului care-l indică pe A, ci **trebuie realizată efectiv copierea**.
- Dacă nu se procedează în această manieră, modificări ulterioare ale unei mulțimi se resfrâng în mod nedorit și asupra celeilalte (conceptul "aliasing").
- Operatorul ***Min*** returnează primul element al listei.
- Operatorii ***Suprimă*** și ***Caută*** pot fi implementați prin căutarea nodului implicat în operație, după una din tehniciile precizate la studiul listelor, iar în cazul lui ***Suprimă***, ștergând nodul găsit.
- În ceea ce privește operația ***Adaugă***, aceasta este o aplicație derivată din listele ordonate, care a fost precizată în primul volum al lucrării, cap.6 [Cr00].

## 9.4. Structuri de date derivate din structura mulțime

### 9.4.1. Structura dicționar

- În unele aplicații care utilizează structuri de date mulțime nu sunt necesare operații complexe cum ar fi reuniunea sau intersecția.
  - De regulă, este necesară păstrarea evidenței unei mulțimi “curente” de obiecte în care periodic se realizează **adăugiri**, **extrageri** sau teste de **apartenență**.
- Se numește **dicționar** o structură de date abstractă derivată din structura mulțime, pe care sunt definiți operatorii:
  - ( 1 ) ***Adaugă***.
  - ( 2 ) ***Suprimă***.
  - ( 3 ) ***Apartine***.
  - ( 4 ) ***Vid*** - care permite inițializarea unei astfel de structuri de date.
- În continuare vor fi discutate câteva posibilități de implementare a **structurii de date dicționar**.
  - Un dicționar poate fi implementat cu ajutorul unei structuri **listă** atât în varianta **ordonată** cât și în varianta **neordonată**.
  - O altă implementare posibilă a dicționarului o reprezintă **vectorul binar**.
    - Aceast lucru este realizabil dacă elementele mulțimii sunt numere întregi în domeniul  $0, 1, 2, \dots, N-1$ , cu un  $N$  precizat, sau pot fi puse în corespondență cu un astfel de domeniu de numere întregi.
  - În continuare se prezintă mai în detaliu **alte două modalități** de implementare a **structurii dicționar**.

#### 9.4.1.1. Implementarea structurii dicționar cu ajutorul tablourilor

## liniare

- Un dicționar poate fi implementat cu ajutorul unui **tablou** prevăzut cu un indicator la ultima intrare utilizată.
  - Această implementare este viabilă dacă se presupune că dicționarul **nu** va conține mai multe elemente decât dimensiunea maximă a tabloului.
  - Față de listele înlántuite, această implementare care are **avantajul** simplității, are două **dezavantaje**:
    - (1) Dicționarul nu poate crește în mod arbitrar.
    - (2) Spațiul de memorie este utilizat ineficient.
- În secvența [9.4.1.1.a] apare un model al acestei reprezentări.

---

```
/*Implementarea structurii dicționar cu ajutorul structurii
tablou -varianta C*/
```

```
int DimMax = {o valoare potrivita};

typedef TipDictionar
{
    int ultim;
    TipElement continut[DimMax];
} TipDictionar;

void Vid(TipDictionar * A) /*Performanța O(1)*/
{
    (*A).ultim= -1;
} /*Vid*/

boolean Apartine(TipElement x, TipDictionar * A)
{
    int i;                      /*Performanța O(n)*/
    for (i=0;i<(*A).ultim;i++)
        if ((*A).continut[i]==x) return true;
    return false;
} /*Apartine*/                  /*[9.4.1.1.a]*/

void Adauga(TipElement x, TipDictionar * A)
{
    if (!Apartine(x,&A))      /*Performanța O(n)*/
        if ((*A).ultim<(DimMax-1))
        {
            (*A).ultim=(*A).ultim+1;
            (*A).continut[(*A).ultim]=x;
        }
    else
        *eroare('structura este plina');
} /*Adauga*/

void Suprima(TipElement x, TipDictionar * A)
{
```

```

int i;                                /*Performanță O(n)*/

if ((*A).ultim>=0)
{
    i= 0;
    while (((*A).continut[i]<>x)&&(i<(*A).ultim))
        i++;
    if ((*A).continut[i]==x)
    {
        (*A).continut[i]=(*A).continut[(*A).ultim];
        (*A).ultim=A.ultim-1;
    } /*if*/
} /*if*/
} /*Suprima*/

```

---

**{Implementarea structurii dicționar cu ajutorul structurii tablou - varianta PASCAL}**

```

const DimMax = {o valoare potrivita};

type TipDictionar = record
    ultim: integer;
    continut: array[1..DimMax] of TipElement
end;

procedure Vid(var A: TipDictionar); {Performanță O(1)}
begin
    A.ultim := 0;
end;{Vid}

function Apartine(x: TipElement; var A: TipDictionar): boolean;
var i: integer;                      /*Performanță O(n)*/
begin
    for i := 1 to A.ultim do
        if A.continut[i] = x then return(true);
    return(false)
end;{Apartine}                      [9.4.1.1.a]

procedure Adauga(x: TipElement; var A: TipDictionar);
begin
    if not Apartine(x,A) then          {Performanță O(n)}
        if A.ultim < DimMax then
            begin
                A.ultim := A.ultim + 1;
                A.continut[A.ultim] := x
            end
        else
            eroare('structura este plina')
    end;{Adauga}

procedure Suprima(x: TipElement; var A: TipDictionar);
var i: integer;
begin                                {Performanță O(n)}
    if A.ultim > 0 then
        begin

```

```

i := 1;
while (A.continut[i] <> x) and (i < A.ultim) do
    i := i + 1;
if A.continut[i] = x then
    begin
        A.continut[i] = A.continut[A.ultim];
        A.ultim := A.ultim - 1
    end
end
end; {Suprima}

```

---

- Implementarea intersecției și reuniunii cu ajutorul tablourilor liniare este relativ **dificilă** motiv pentru care **nu** a fost abordată problema reprezentării mulțimilor în general cu ajutorul tablourilor.
- Cu toate acestea, întrucât există metode eficiente de sortare a tablourilor, procedurile descrise în secvența care se referă la structura dicționar pot fi considerate drept punct de plecare într-o posibilă implementare a structurilor multime cu ajutorul **tablourilor liniare**.

#### **9.4.1.2. Implementarea structurii dicționar prin tehnica dispersiei**

- După cum s-a precizat în paragraful anterior, implementarea **dicționarului** bazată pe **tablouri liniare** necesită în medie  $O(n)$  pași pentru execuția operațiilor **Adaugă**, **Suprimă** sau **Apartine** într-un dicționar cu  $n$  elemente.
- O regie similară se obține și pentru implementarea bazată pe **liste înlănuite**.
- Implementarea bazată pe **vectori binari**, consumă un interval constant de timp pentru a executa oricare din operațiile anterior precizate, cu restricția însă că dimensiunea mulțimii de bază este limitată la o dimensiune impusă de arhitectura hardware.
- O altă tehnică larg utilizată în implementarea structurilor dicționar este **tehnica dispersiei** (Vol.1 &7.3).
  - **Tehnica dispersiei** necesită în medie un timp constant pe operație.
  - Nu impune nici o restricție referitoare la cardinalitatea sau tipul elementelor mulțimii de bază.
  - În cel mai rău caz, această tehnică necesită  $O(n)$  pași adică identic cu implementările bazate pe tablouri sau liste. În practică însă, se ajunge foarte rar în această situație.
- După cum s-a precizat în Vol.1, se pot lua în considerare două variante ale tehnicii dispersiei:
  - (1) **Dispersia deschisă** în care situațiile de coliziune se tratează prin înlănuire directă și care permite ca mulțimea să fie memorată într-un spațiu practic nelimitat, (deci cardinalitatea mulțimii de bază **nu** este limitată).

- (2) **Dispersia închisă** bazată pe metoda adresării deschise liniare sau adresării deschise patratice în care se utilizează un spațiu fix de memorie și în consecință mulțimea de bază va fi limitată ca dimensiune.
- Întrucât principiile tehnicii dispersiei au fost pe larg discutate în referința mai sus precizată, în paragraful de față se vor prezenta două **exemple** care materializează cele două variante.
  - În ambele exemple se va utiliza o **funcție de dispersie** clasică notată cu  $h(x)$  unde  $x$  este de tip sir de caractere (string).
- În secvența [9.4.1.2.a] se observă structurile de date utilizate în implementarea **dicționarului** prin tehnica **dispersiei deschise**.
  - **Dicționarul** este de fapt un **tablou de pointeri**, fiecare indicând o lista înălțuită cu noduri de tip cuvânt.
  - Fiecare listă cuprinde acele elemente  $x$  ale dicționarului pentru care funcția  $h(x)$  furnizează aceeași valoare (clasa de elemente), situația de coliziune fiind rezolvată în acest caz prin **metoda înlățuirii directe**.
  - Tehnica utilizată în implementarea listelor face necesară tratarea separată a primului cuvânt din listă, element care se observă în cadrul procedurii **Suprimă**.

---

/\*Implementarea structurii dicționar prin tehnica dispersiei deschise - varianta C\*/

```

#include <stdio.h>
#include <stdlib.h>
#include<stdbool.h>
#include<string.h>

#define P=131

typedef char* TipElement; //pointer la string

typedef int TipIndice;

typedef struct TipCuvant
{
    TipElement element;
    struct TipCuvant *urm;
} TipCuvant;

typedef TipCuvant *RefCuvant;

typedef struct TipDictionar
{
    RefCuvant dictionar[P];
} TipDictionar;

TipIndice h(TipElement x)
{
    int i,suma;

```

```

    suma=0;
    int nr=strlen(x);
    for(i=0;i<nr;i++)
        suma=suma+(int)x[i];
    return suma%P;
}

void Vid(TipDictionar *A)
{
    int i;

    for(i=0;i<=P-1;i++)
        A->dictionar[i]=NULL;
}

bool Apartine(TipElement x, TipDictionar *A)
{
    RefCuvant curent;

    curent=A->dictionar[h(x)];
    while(curent!=NULL)
    {
        if(strcmp(curent->element,x)==0) return true;
        else (curent=curent->urm);
    }
    return false;
}

void Adauga(TipElement x, TipDictionar *A)           // [9.4.1.2.a]
{
    int l;
    RefCuvant vechi;

    if((Apartine(x,A))==false)
    {
        l=h(x);
        vechi=A->dictionar[l];
        A->dictionar[l]=(RefCuvant)malloc(sizeof(TipCuvant));
        // alocare memorie pentru campul element (cuvantul propriu-zis)
        A->dictionar[l]->element=
            (char*)malloc(strlen(x)+1)*sizeof(char));
        strcpy(A->dictionar[l]->element,x);
        A->dictionar[l]->urm=vechi;
    }
}

void Suprima(TipElement x, TipDictionar *A){

    RefCuvant curent,aux;
    int l;

    l=h(x);
    if(A->dictionar[l]!=NULL)
    {
        if(strcmp(A->dictionar[l]->element,x)==0)
        {
            aux=A->dictionar[l];
            A->dictionar[l]=A->dictionar[l]->urm;
            free(aux); // elibereare memorie
        }
    }
}

```

```

        }

    else
    {
        curent=A->dictionar[1];

        while(curent->urm!=NULL)
        {
            if(strcmp(curent->urm->element,x)==0)
            {
                aux=curent->urm;
                curent->urm=curent->urm->urm;
                free(aux); // eliberare memorie
            }
            else curent=curent->urm;
        }
    }
}

```

```

void Afisare(TipDictionar *A)
{
    for(i=0; i<P; i++)
    {
        printf("Hash %d: ", i);
        while(A->dictionar[i]!=NULL)
        {
            printf("%s ",A->dictionar[i]->element);
            A->dictionar[i]=A->dictionar[i]->urm;
        }
        printf("\n");
    }
}

```

---

**{Implementarea structurii dicționar prin tehnica dispersiei deschise - varianta PASCAL}**

```
const P={o valoare convenabilă, de regulă număr prim}
```

```

type TipElement = array[1..12] of char;
    TipIndice = 0..(P-1);
    RefCuvant: ^TipCuvant;
    TipCuvant = record
        element: TipElement;
        urm: RefCuvant
    end;
    TipDictionar = array[TipIndice] of RefCuvant;

function h(x: TipElement): TipIndice;
    var i,suma: integer;
begin
    suma := 0;
    for i := 1 to 12 do
        suma := suma + ord(x[i]);
    h := suma mod P
end; {h}

```

```

procedure Vid(var A: TipDictionar); [9.4.1.2.a]
  var i: integer;
begin
  for i := 1 to P-1 do
    A[i] := nil
end; {Vid}

function Apartine(x: TipElement; var A: TipDictionar): boolean;
var curent: RefCuvant;
begin
  curent := A[h(x)];
  while curent <> nil do
    if curent^.element = x then
      return(true)
    else
      curent := curent^.urm;
  return(false)
end; {Apartine}

procedure Adauga(x: TipElement; var A: TipDictionar);
var l: integer;
  vechi: RefCuvant;
begin
  if not Apartine(x,A) then
    begin
      l := h(x);
      vechi := A[l]; {inserție în față}
      new(A[l]);
      A[l]^ .element := x;
      A[l]^ .urm := vechi
    end
  end; {Adauga}

procedure Suprima(x: TipElement; var A: TipDictionar); [9.4.1.2.a]
var curent: RefCuvant;
  l: integer;
begin
  l := h(x);
  if A[l] <> nil then
    begin
      if A[l]^ .element = x then {x este pe prima
                                  poziție}
        A[l] := A[l]^ .urm {se scoate x din listă}
      else
        begin
          curent := A[l];
          {se aplica tehnica lookahead}
          while curent^.urm <> nil do
            if curent^.urm^.element = x then
              begin {scoate pe x din listă}
                curent^.urm := curent^.urm^.urm;
                return
              end
            else {x nu a fost încă găsit}
              curent := curent^.urm
        end
    end
  end;

```

```
    end  
  end  
end; {Suprima}
```

---

- În secvența [9.4.1.2.b] apare implementarea **dicționarului** prin tehnica **dispersiei închise**.
  - Situațiile de coliziune se tratează prin **metoda adresării deschise liniare**.
  - Structurile de date utilizate sunt cele precizate în secvența [9.4.1.2.b], iar funcția  $h(x)$  este cea utilizată în exemplul anterior.
  - Prin **convenție**, s-a utilizat un sir de 12 caractere underscore pentru valoarea **liber** și un sir de 12 asteriscuri pentru valoarea **sters**, presupunându-se că nici un element nu poate lua aceste valori.
- Au fost definite două funcții de căutare.
  - (1) Funcția **Caută** parurge tabelul A conform **metodei de adresare deschisă liniară** până:
    - (a) Îl găsește pe  $x$ .
    - (b) Găsește o locație neocupată (**liber**).
    - (c) A parcurs circular tabloul și nu l-a găsit pe  $x$ .
    - (d) În toate cazurile **Caută** returnează **indicele** din tablou la care s-a oprit căutarea indiferent de motiv.
  - (2) Funcția **Caută1** este asemănătoare lui **Caută** cu singura deosebire că ea extinde procesul de căutare și la locații marcate cu **sters**.
    - **Caută1** se folosește numai în operatorul **Adaugă** pentru a găsi prima locație disponibilă.

---

```
/*Implementarea structurii dicționar prin tehnica dispersiei  
închise - Varianta C*/
```

```
#include <stdio.h>  
#include <stdlib.h>  
#include<stdbool.h>  
#include<string.h>  
  
#define P 131  
  
const char *liber = "_____";  
const char *sters= "*****";  
  
typedef char *TipElement;  
  
typedef TipElement TipDictionar[P];  
  
int h(TipElement x)
```

```

    {
        int i,suma;

        suma=0;
        int nr=strlen(x);
        for(i=0;i<nr;i++)
            suma=suma+(int)x[i];
        return suma%P;
    }

void Vid(TipDictionar A)
{
    int i;

    for(i=0;i<P;i++)
        A[i]=(char*)malloc(sizeof(char));
    for(i=0;i<P;i++)
        strcpy(A[i],liber);
}

void Adauga(TipElement x, TipDictionar A)
{
    int l;

    if(strcmp(A[Cauta(x,A)],x)!=0) {
        l=Cauta1(x,A);
        if( (strcmp(A[l],liber)==0) || (strcmp(A[l],sters)==0) )
            strcpy(A[l],x);
        else
            printf("\n Tabela e plina\n");
    }
}

bool Apartine(TipElement x, TipDictionar A)
{
    if(A[Cauta(x,A))==x)
        return true;
    else
        return false;
}

int Cauta( TipElement x, TipDictionar A)
{
    int initial,i;

    initial= h(x);
    i= 0;
    while (((i<P) && strcmp(A[(initial+i) % P],x)!=0)
           &&(strcmp(A[(initial+i) % P],liber)!=0))
        i= i+1;
    return ((initial+i) % P);
}

int Cauta1( TipElement x, TipDictionar A)
{
    //verifica si pozitiile sterse
    int initial,i;
}

```

```

        initial= h(x);
        i= 0;
        while (((i<P) && strcmp(A[(initial+i)% P],x)!=0)
               && strcmp(A[(initial+i)% P],liber)!=0 &&
               strcmp(A[(initial+i)% P],sters)!=0)
            i= i+1;

        return ((initial+i) % P);
    }

void Suprima(TipElement x, TipDictionar A)
{
    int i;

    i=Cauta(x,A);
    if(strcmp(A[i],x)==0)
        strcpy(A[i],sters);
}

void Afisare(TipDictionar A)
{
    int i;

    for(i=0;i<P;i++)
        printf("Codul Hash :%d -> %s\n",h(A[i]),A[i]);
}

-----
{Implementarea structurii dicționar prin tehnica dispersiei
închise - Varianta PASCAL}

const P = {o valoare potrivită, de regulă număr prim};
    liber = ' ' ; {12 blancuri}
    sters = '*****' ; {12 asteriscuri}

type TipElement = array[1..12] of char;
    TipDictionar = array[0..P-1] of TipElement;

procedure Vid(var A: TipDictionar);
    var i: integer;
    begin
        for i := 0 to P-1 do
            A[i] := liber
    end; {Vid}

function Cauta(x: TipElement; A: TipDictionar): integer;
    var initial,i: integer;
    begin [9.4.1.2.b]
        initial := h(x);
        i := 0;
        while (i<P) and (A[(initial+i) mod P]<>x) and
               (A[(initial+i) mod P]<>liber) do
            i := i+1;
        Cauta := (initial+i) mod P
    end; {Cauta}

```

```

function Cauta1(x: TipElement; A: TipDictionar): integer;
{similară funcției Cauta dar în plus returnează și o intrare
ștearsă}

procedure Adauga(x: TipElement; var A: TipDictionar);
var l: integer;
begin
    if A[Cauta(x,A)] <> x then
        begin
            l := Cauta1(x,A);
            if (A[l] = liber) or (A[l] = sters) then
                A[l] := x
            else
                eroare('tabela este plină')
        end
    end; {Adauga}

function Apartine(x: TipElement; A: TipDictionar): boolean;
begin
    if A[Cauta(x,A)] = x then
        Apartine := true
    Else [9.4.1.2.b]
        Apartine := false
    end; {Apartine}

procedure Suprima(x: TipElement; var A: TipDictionar);
var i: integer;
begin
    i := Cauta(x,A);
    if A[i] = x then A[i]:= sters
end; {Suprima}

```

---

## 9.4.2. Structuri de date complexe bazate pe structura mulțime

### 9.4.2.1. Relația bazată pe corespondențe multiple

- Se consideră o mulțime de studenți și o mulțime de concursuri profesionale.
- Un exemplu de relație bazată pe **corespondențe multiple** (“**many-many relationship**” [AH85]) îl reprezintă relația dintre studenți și concursurile profesionale la care aceștia participă.
  - Aceasta este o relație bazată pe corespondențe multiple deoarece:
    - (1) La un concurs pot participa mai mulți studenți.
    - (2) Un student poate participa la mai multe concursuri profesionale.
    - (3) În timp, listele de participanți se pot modifica, întrucât se pot înscrie noi studenți, unii se pot retrage sau pot trece la alte discipline.
- Problemele care se pun în legătură cu această relație bazată pe corespondențe multiple se referă la a și la un moment dat:

- (1) **Care sunt studenții** care participă la o anumită disciplină de concurs.
- (2) **La ce concurs** participă un anumit student.

#### 9.4.2.2. Implementare bazată pe structura tablou a relației bazate pe corespondențe multiple

- Structura de date cea mai simplă care acoperă aceste cerințe este un **tablou** cu două dimensiuni **INSCRIERE**, sugerat de figura 9.4.2.2.a, unde valoarea adevarat reprezintă **înscriș** (marcat cu x) iar 0 valoarea fals reprezintă **neînscriș**.

	Matematica	Programare	Mecanica
Agache			x
Anghel	x	x	x
Arianu			
Ban		x	
Banu	x		
Berinde	x	x	
Cartu		x	
Cerbu	x		x

INSCRIERE

**Fig. 9.4.2.2.a.** Exemplu de relație bazată pe corespondențe multiple implementată cu ajutorul unei structuri tablou

- Pentru a **înscrie** un student la un concurs profesional este necesară crearea în prealabil a două **asocieri**.
  - (1) O **asociere AS**, eventual implementată prin tehnica dispersiei, care translatează **numele studentului** într-un **indice** aparținând unei dimensiuni a tabloului.
  - (2) O a doua **asociere AC** care translatează **numele concursului** într-un **indice** al celei de-a doua dimensiuni a tabloului.
- În această situație:
  - (1) **Înscrierea** studentului s la concursul c se realizează simplu [9.4.2.2.a].

---

INSCRIERE [AS(s), AC(c)] = true [9.4.2.2.a]

---

- (2) **Retragerea** studentului s de la cursul c se implementează exact la fel atribuind însă constanta booleană false.
- (3) Pentru a afla **concurșurile** la care s-a înscriș **un student** cu numele s se parurge linia AS(s) a tabloului INSCRIERE.

- (4) Pentru a afla **studenții** înscriși la **concursul** C se parcurge coloana AC (C) a tabloului **INSCRIERE**.
- Această abordare conduce la o implementare simplă și performantă, în schimb consumul de mare de memorie este mare și gradul de utilizare redus.
- Presupunând că în universitate există în total aproximativ 6000 de studenți care pot participa la 20 de concursuri diferite, avem nevoie de un tablou care ocupă 120.000 de elemente.
- Deoarece mai puțin de 20% din numărul studenților participă de fapt la astfel de concursuri, marea majoritate la o singură disciplină, rezultă că structura tablou este foarte slab utilizată (sub 6%).
- O astfel de matrice se numește **rară** ("sparse"), parcurgerea ei presupunând un interval considerabil de timp și în același timp, memorarea ei, o mare risipă de memorie.
- Ca atare se investighează și alte posibilități de implementare.

#### **9.4.2.3. Implementarea relației bazate pe corespondențe multiple cu ajutorul mulțimilor. Variante de implementare**

- O metodă mai bună de a rezolva această problemă, este aceea de a implementa relația bazată pe corespondențe multiple ca și o **colecție de mulțimi**.
- Două dintre aceste mulțimi sunt S reprezentând mulțimea tuturor **studenților** și C reprezentând mulțimea tuturor **concursurilor**.
  - Fiecare element al lui S este o structură TipStudent1.
  - Fiecare element al lui C este o structură TipConcurs1.

---

**{Implementarea relației bazate pe corespondențe multiple cu ajutorul mulțimilor Varianta 1}**

```
type TipStudent1=record
    marca: integer;
    numeStudent: string[20]
  end;
```

```
TipConcurs1=record                                     [ 9 . 4 . 2 . 3 . a ]
    dataC: TipData;
    numeConcurs: string[15]
  end;
```

```
TipInscriere1=record
    student: TipStudent1;
    concurs: TipConcurs1
  end;
```

- Pentru a implementa relația dorită este necesară o a treia mulțime  $I$  care implementează **înscrierea**.
  - Elementele mulțimii  $I$  sunt structuri `TipInscriere1`, care realizează **asocierea** student-concurs.
  - În mulțimea  $I$  există câte un element pentru fiecare locație marcată cu  $x$  în structura tablou `INSCRIERE` prezentată în figura 9.4.2.2.a.
- În plus pentru a rezolva **problema corespondențelor multiple** este necesară precizarea unor **mulțimi suplimentare**. Este vorba despre:
  - (1) Mulțimile  $C_s$  – câte o mulțime pentru fiecare student  $s$ , mulțime care include concursurile la care acesta participă.
  - (2) Mulțimile  $S_c$  – câte o mulțime pentru fiecare concurs  $c$ , incluzând mulțimea studenților înscriși la concursul respectiv.
- Astfel de mulțimi ridică însă probleme de implementare din cauza:
  - Numărului mare de mulțimi  $C_s$ .
  - Numărului mare de elemente din mulțimea  $S_c$ .
  - Naturii diferite a elementelor celor două tipuri de mulțimi: structuri `TipStudent1` respectiv `TipConcurs1`.
- Se pot concepe mai multe **variante de implementare** a mulțimilor  $C_s$  respectiv  $S_c$ .
  - (1) O **primă variantă**, numită **varianta 0**, constă în implementarea mulțimilor  $C_s$  și  $S_c$  ca și **mulțimi de indicatori** la structuri `TipConcurs1` respectiv la structuri `TipStudent1`.
  - (2) O altă **variantă** care economisește spațiu și în același timp permite precizarea rapidă a relațiilor studenți-concursuri este următoarea.
    - Atât mulțimile  $C_s$  cât și mulțimile  $S_c$  sunt concepute unitar ca fiind alcătuite din structuri `TipInscriere1`, fiecare structură precizând studentul  $s$  și concursul  $c$  la care acesta s-a înscris [9.4.2.3.a].
    - Formal, definirea acestor mulțimi este cea precizată în [9.4.2.3.b].

$$C_s = \{ (s, c) \mid s \text{ s-a înscris la concursul } c, s=ct \} \quad [9.4.2.3.b]$$

$$S_c = \{ (s, c) \mid s \text{ s-a înscris la concursul } c, c=ct \}$$

- Referitor la secvența [9.4.2.3.b] se face precizarea că  $(s, c)$  e de fapt un articol de `TipInscriere1`.

- În concluzie, deși au componente de aceeași tip, cele două tipuri de mulțimi se **diferențiază** prin aceea că într-o mulțime  $C_S$ ,  $s$  este constant, iar într-o mulțime  $S_C$ ,  $c$  este constant.
- Pentru implementarea acestor mulțimi aferente relațiilor bazate pe corespondențe multiple se pot utiliza în mod avantajos **structurile de date multilistă**.
  - Se reamintește că o **multilistă** este o colecție de noduri dintre care unele au mai mult decât un pointer și pot face parte simultan din mai multe liste.
  - Pentru fiecare tip de nod aparținând unei structuri multilistă este important să se precizeze cu claritate numele și semnificația pointerilor implicați, nodurile putând差别 ca structură.
- Pornind de la această abordare denumită **varianta 1** de implementare, se rafinează **varianta 2** (secvența [9.4.2.3.c]) în care:
  - (1) Structura **TipInscriere1** se modifică în **TipInscriere2**, care constă din două câmpuri indicator:
    - Câmpul **concursUrm** indicând elementul următor de tip înscriere din mulțimea  $C_S$  căreia îi aparține.
    - Câmpul **studUrm** indicând elementul următor de tip înscriere din mulțimea  $S_C$  corespunzătoare [9.4.2.3.c].
  - (2) Structurii **TipStudent1** i se atașează un indicator care precizează primul concurs la care s-a înscris studentul în cauză, adică prima structură de tip **TipInscriere2** din mulțimea  $C_S$  respectivă și devine structura **TipStudent2**.
  - (3) Structurii **TipConcurs1** i se atașează un indicator care precizează primul student care s-a înscris la concursul în cauză, adică prima structură de tip **TipInscriere2** din mulțimea  $S_C$  asociată și devine structura **TipConcurs2**.

---

**{Implementarea relației bazate pe corespondențe multiple  
Varianta 2}**

```
type RefTipInscriere2 = ^TipInscriere2;

TipStudent2=record
    marca: integer;
    numeStudent: string[20];
    concurs: RefTipInscriere2
end;

TipConcurs2=record [9.4.2.3.c]
    dataC: TipData;
    numeConcurs: string[15];
    student: RefTipInscriere2
end;
```

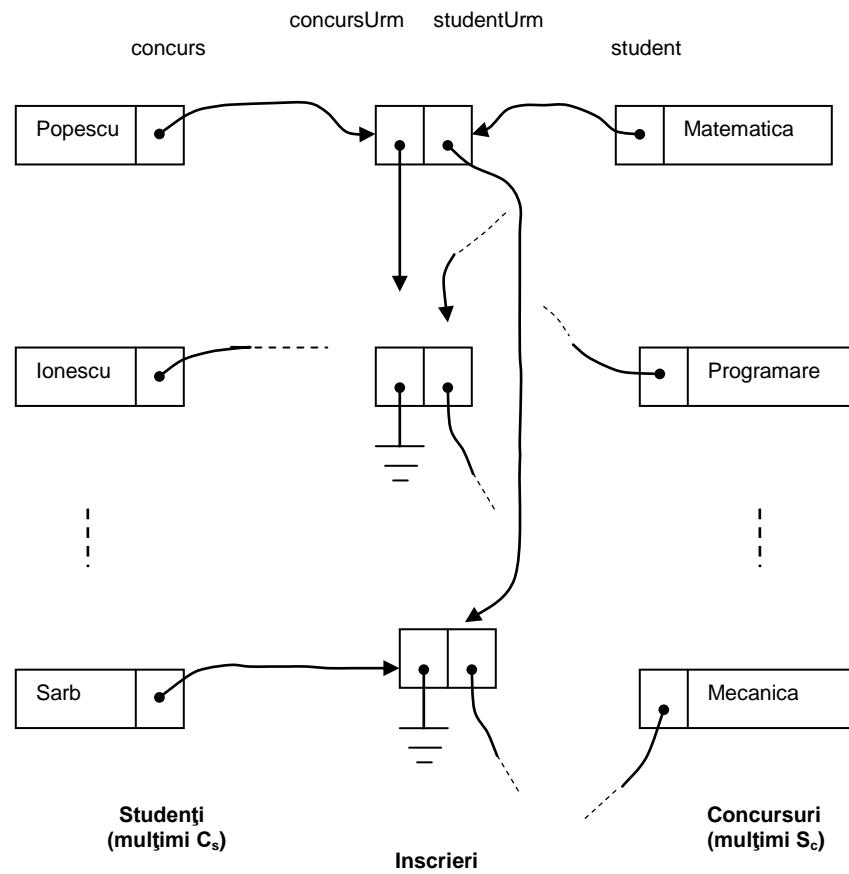
```

TipInscriere2=record
    concursUrm: RefTipInscriere2;
    studentUrm: RefTipInscriere2
end;

```

---

- Se constată faptul că un articol de TipInscriere2 aparține în același timp la două liste înlănuite distințe (fig.9.4.2.3.a).



**Fig.9.4.2.3.a.** Implementarea relației bazate pe corespondențe multiple, varianta 2

- Fiecare structură **TipStudent2**, este începutul unei liste care materializează mulțimea  $C_s$  corespunzătoare, adică mulțimea concursurilor la care s-a înscris studentul în cauză.
- Fiecare structură **TipConcurs2**, este începutul unei liste care materializează mulțimea  $S_c$  corespunzătoare, adică mulțimea studenților care participă la concursul în cauză.
- Se reamintește faptul că, atât mulțimile  $S_c$  cât și mulțimile  $C_s$  sunt formate din structuri **Inscriere2**.

- De fapt o structură TipInscriere2 nu indică în mod explicit nici **studentul** nici **concursul** la care se referă.
  - Această informație rezultă în mod implicit din **lista** în care este înlățuită structura respectivă.
- În continuare, structurile TipStudent2, respectiv structurile TipConcurs2 se vor numi **proprietari** ai structurilor TipInscriere2 care aparțin listelor pe care le inițiază.
- Astfel pentru a se preciza la ce concursuri participă un anumit student s:
  - Trebuie parcuse structurile TipInscriere2 din mulțimea  $C_s$  (pointerul concursUrm) pornind de la structura TipStudent **proprietar**.
  - Pentru fiecare element parcurs, trebuie determinată structura TipConcurs **proprietară**.
- Se observă că structura de date propusă **varianta 2** nu poate soluționa simplu această cerință.
- Pentru soluționarea acestei cerințe se pot adăuga structurii înscriere încă doi **indicatori** unul pentru structura proprietar de TipStudent, celălalt pentru structura proprietar de TipConcurs, după cum se prezintă în **varianta 3** de implementare, secvența [9.4.2.3.d].

---

**{Implementarea relației bazate pe corespondențe multiple  
Varianta 3}**

```

type RefTipInscriere3 = ^TipInscriere3;

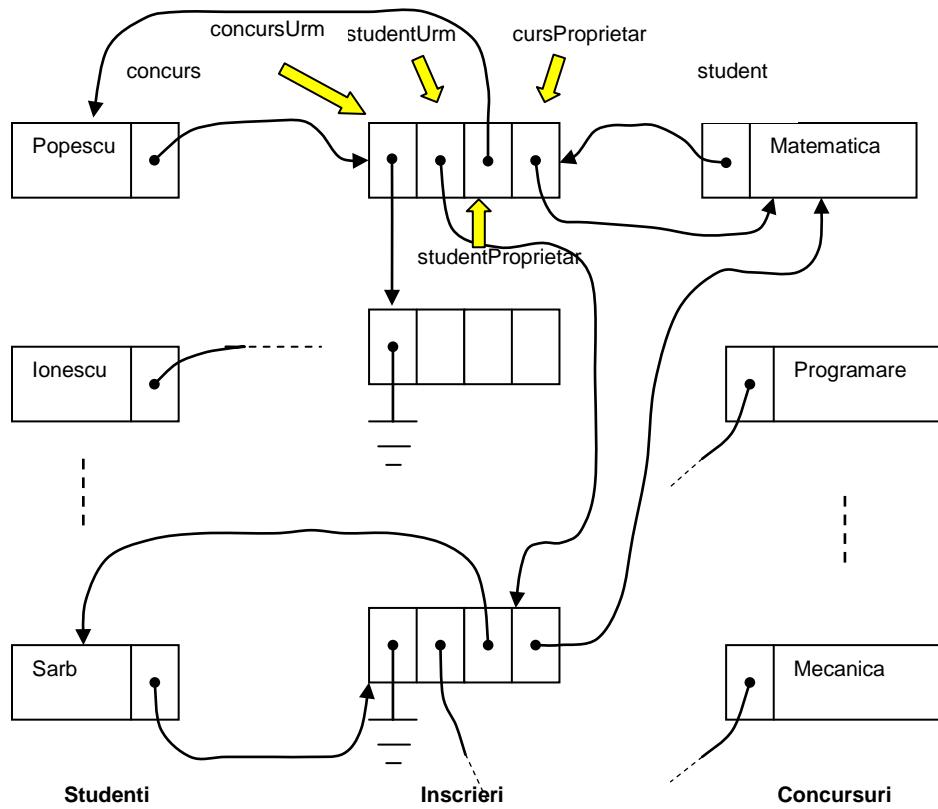
TipStudent3=record
  marca: integer;
  numeStudent: string[20];
  concurs: RefTipInscriere3
end;[ 9 . 4 . 2 . 3 . d ]

TipConcurs3=record
  dataC: TipData;
  numeConcurs: string[15];
  student: RefTipInscriere3
end;[ 9 . 4 . 2 . 3 . d ]

TipInscriere3=record
  concursUrm: RefTipInscriere3;
  studentUrm: RefTipInscriere3;
  studentProprietar: RefTipStudent3;
  concursProprietar: RefTipConcurs3
end;
```

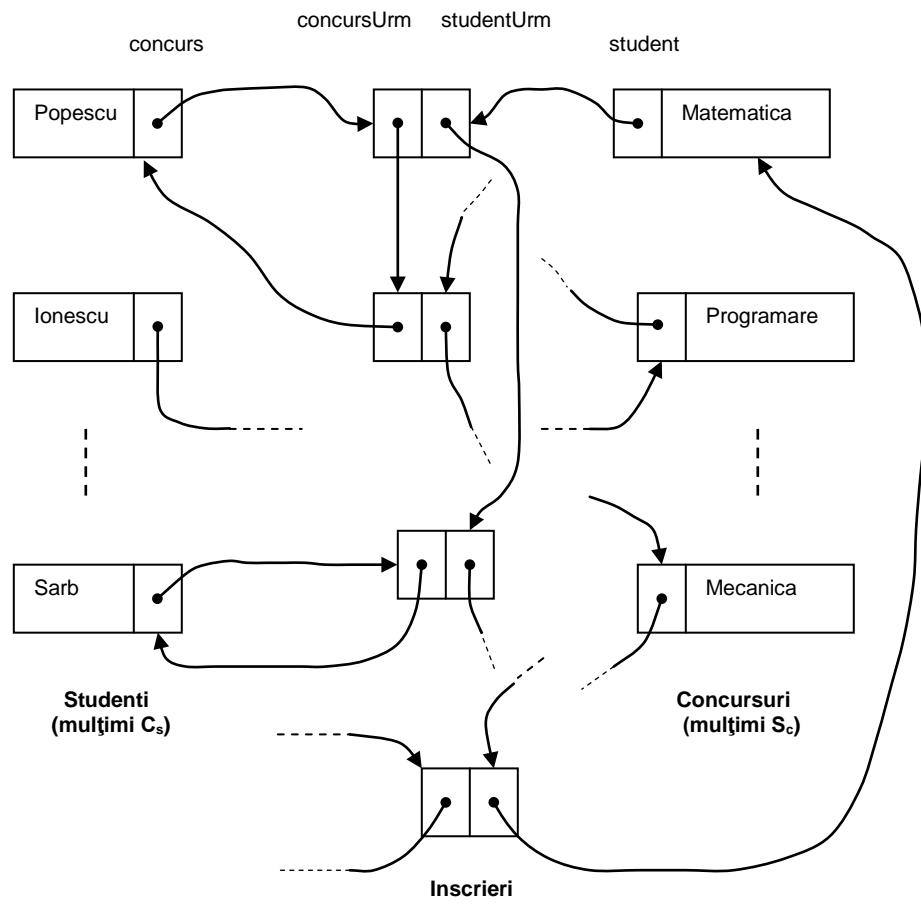
---

- Varianta 3 de implementare apare reprezentată grafic în figura 9.4.2.3.b.



**Fig.94.2.3.b.** Implementarea relației bazate pe corespondențe multiple, varianta 3

- O astfel de structură de articol înscriere este **cea mai eficientă** din punct de vedere al performanței în contextul avut în vedere.
  - Are însă dezavantajul că ocupă multă memorie.
- Se poate salva o cantitate substanțială de memorie, cu prețul sporirii rezonabile a timpului de acces utilizând următoarea metodă:
  - Se elimină ultimii doi pointeri din structura articoului **Inscriere3**.
  - Se plasează la sfârșitul fiecărei liste  $S_C$  un pointer la **concursul proprietar**
  - Se plasează la sfârșitul fiecărei liste  $C_S$  un pointer la **studentul proprietar**.
  - Astfel fiecare structură de **TipStudent** sau **TipConcurs** devine parte a unei **liste circulare** care include înregistrările a căror proprietar este.
- Acest lucru apare ilustrat în fig.9.4.2.3.c drept **varianta 4** de implementare.



**Fig.94.2.3.c.** Implementarea relației bazate pe corespondențe multiple, varianta 4

- Dacă se dorește spre **exemplu** să se afle mulțimea studenților înscriși la concursul de matematică, se procedează după cum urmează:
  - (1) Se depistează mai întâi structura de **TipConcurs Matematica** a mulțimii concursurilor.
    - Modul în care se realizează acest lucru depinde de maniera de implementare a mulțimii CONCURSURI, spre exemplu ca lista înălțuită sau ca tabelă de dispersie.
  - (2) Pornind de la pointerul cuprins în articolul **Matematica** se ajunge la prima structură **TipInscriere** din lista sa circulară.
  - (3) Pentru a afla studentul **proprietar** al acestei structuri de **TipInscriere** se urmărește câmpul **concursUrm** al structurii până când se găsește o structură **TipStudent**.
  - (4) Pentru a afla restul studenților înscriși la matematică, se înaintează pe înălțuirea indicată de câmpul **studentUrm** începând cu prima structură **TipInscriere** și pentru fiecare structură parcursă se aplică procedeul precizat anterior de determinare a studentului proprietar.

- (5) În final, urmând înlănțuirea indicată de câmpul studUrm se ajunge din nou la structura Matematica și astfel lista urmată s-a închis.
- Operațiile aferente listării studenților participanți la concursul de Matematica, respectiv determinării studentului proprietar al unui articol de TipInscriere sunt descrise formal în secvențele [9.4.2.3.e] respectiv [9.4.2.3.f].

---

**{Listare studenți înscriși la concursul de matematică}**

```
pentru fiecare articol de TipInscriere din lista Sc
    indicata de articolul Matematica execută
```

```
begin [9.4.2.3.e]
    *se atribuie lui s numele studentului proprietar
        al articolului de TipInscriere curent;
    afiseaza(s)
end;
```

---

{\*se atribuie lui s numele studentului proprietar al  
articolului de TipInscriere}

```
atribuie p <- referința la articolul de TipInscriere
    curent;
```

```
repeta [9.4.2.3.f]
    atribuie p <- p^.concursUrm
    pana cand p este o referinta la un articol de
        TipStudent;
    atribuie s <- numeStudent din articolul de TipStudent
        indicat de p}
```

---

- Pentru a implementa o astfel de structură de tip multilistă în limbajul PASCAL se poate defini un singur tip de **articol cu variante** pentru cazurile student, concurs și înscriere (secvența [9.4.2.3.g]).
- Acest lucru este necesar deoarece câmpurile concursUrm respectiv studUrm pot indica articole de tipuri diferite.
- O posibilă implementare PASCAL a listării studenților care participă la un concurs precizat, apar în forma procedurii ListareStudenti(numeC:TipC)) în secvența [9.4.2.3.g].
- În aceeași secvență apar precizate și structurile de date aferente.
- Se face precizarea că această abordare poate fi utilizată drept model pentru o implementare C.

---

**{Implementarea relației bazate pe corespondențe multiple  
Varianta 4}**

```
type TipS = string[20];
    TipC = string[15];
```

```

TipFel = (student,concurs,inscriere);
RefArticol = ^TipArticol;

TipArticol = record
  case fel: TipFel of
    student: (numeStudent: TipS;
               marca: integer;
               primulConcurs: RefArticol);
    concurs: (numeConcurs: TipC;
               dataC: TipData;
               primulStudent: RefArticol);
    inscriere: (concursUrm,studUrm: RefArticol)
  end;

```

#### {Listarea studenților participanți la un concurs precizat}

```

procedure ListareStudenti(numeC: TipC);           [9.4.2.3.g]
  var c,e,p: RefArticol;
  begin
    c:=pointer la articolul concurs pentru care
    c^.numeConcurs = numeC;  {depinde de implementare}
    e:=c^.primulStudent;
    while e^.fel = inscriere do
      begin
        p:=e;
        repeat
          p:=p^.concursUrm
        until p^.fel = student;
        write(p^.numeStudent);
        e:=e^.studUrm
      end
    end; {ListareStudenti}
-----
```

- Exemplul de implementare al unei relații bazate pe corespondențe multiple prezentat în cadrul acestui paragraf, ilustrează faptul că **metoda de dezvoltare graduală pas cu pas** ("stepwise refinement") specifică **dezvoltării algoritmilor**, poate fi utilizată cu succes și în cazul **dezvoltării de structuri de date optimizate**.
- Rezultatul este acela că, de cele mai mult ori structurile rezultate ajung să nu mai semene practic deloc cu modelul real pe care îl abstractizează.

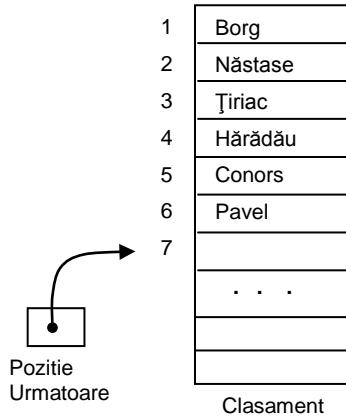
#### 9.4.2.4. Structuri de date combinate

- De cele mai multe ori implementarea reprezentării structurilor abstrakte mulțime sau asociere ridică probleme deosebite.
  - Alegând o anumită reprezentare, anumite operații se implementează simplu și performant altele în schimb necesită o regie ridicată.
  - De fapt ca și în alte domenii, **nu** există soluție care să aibă numai avantaje, respectiv o structură de date care să implementeze simplu și în același timp performant toți operatorii.

- În acest caz, o posibilitate de rezolvare a problemei constă în utilizarea a **două sau mai multe structuri de date diferite** pentru a reprezenta **o aceeași structură de date abstractă**.
- Se presupune spre exemplu că se dorește menținerea **unui clasament al jucătorilor de tenis**, în care fiecare jucător are poziția sa unică.
- **Specificația de definire a clasamentului** este următoarea:
  - (1) Jucătorii ocupă **pozițiile** în clasament în ordine valorică descrescătoare.
  - (2) Un nou jucător este **adăugat** la sfârșitul clasamentului, deci pe poziția cu numărul cel mai mare.
  - (3) Un jucător poate **provoca** la joc jucătorul situat pe poziția anterioară poziției sale.
  - (4) Dacă câștigă meciul, cei doi jucători își **interschimbă** pozițiile în clasament.
- Această situație poate fi reprezentată cu ajutorul unei **structuri de date abstracte** pentru care **modelul** utilizat este o **asociere** între **nume de jucători** (reprezentate ca siruri de caractere) și **pozițiile acestora în clasament** (întregi).
- **Operatorii** pe care îi suportă această structură de date sunt [9.4.2.4.a]:
  - **Operatori definiți pentru structura Clasament**

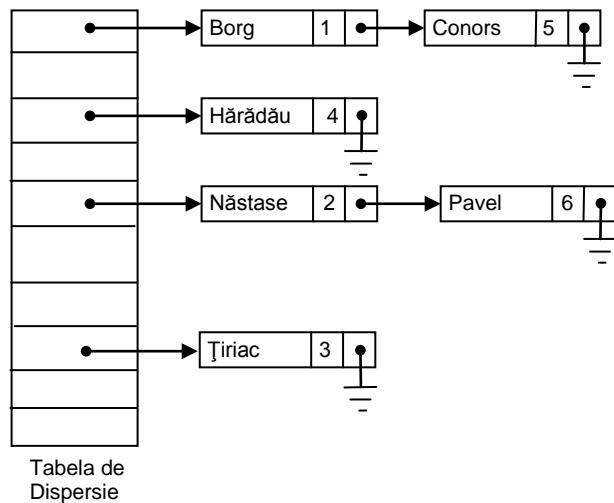
- 
- {Operatori definiți pentru structura Clasament}**
1. **Adauga( nume )** - adaugă numele unui jucător pe poziția cu numărul cel mai mare din clasament.
  2. **poziție Pozitie( nume )** - returnează poziția în tablou a jucatorului precizat ca parametru.
- [ 9.4.2.4.a ]
3. **nume Provoaca( nume )** - funcție care returnează numele jucătorului de pe poziția  $i-1$ , dacă poziția jucătorului precizat ca parametru este  $i$ ,  $i > 1$ .
  4. **Interschimba( pozitie )** - interschimbă în clasament numele jucătorilor situați pe pozițiile  $i$  și  $i-1$ , dacă poziția precizată ca parametru este  $i$ ,  $i > 1$ .
- 

- În acest context se poate observa faptul că primii trei operatori au drept parametru un nume de jucător, în timp ce ultimul operator are drept parametru poziția jucătorului care a lansat provocarea.
- (1) Clasamentul poate fi reprezentat spre exemplu printr-un **tablou Clasament**, unde **Clasament[ i ]** conține numele jucătorului situat pe poziția  $i$  în clasament (fig.9.4.2.4.a).



**Fig.9.4.2.4.a.** Implementarea clasamentului utilizând structura tablou

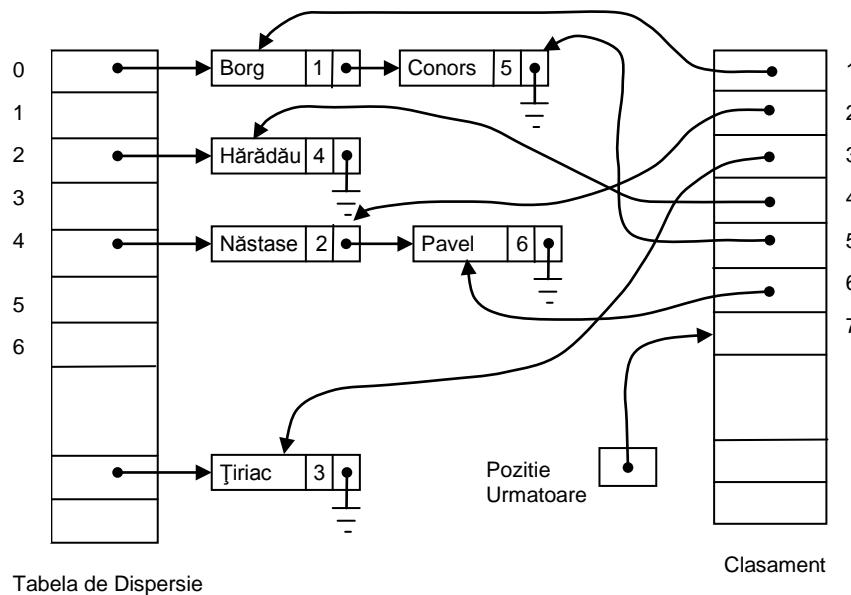
- Se mai utilizează variabila `PozitieUrmatoare` care precizează prima poziție neocupată în cadrul clasamentului.
  - Cu ajutorul acestei variabile adăugarea unui nou jucător în clasament se face într-un număr constant și reduși de pași.
- Operatorul `Interschimba(pozitie)` se implementează simplu cu performanța  $O(1)$ .
- În schimb operatorii `Provoaca(nume)` și `Pozitie(nume)` necesită o căutare în tabloul `Clasament` ceea ce necesită un efort de ordinul  $O(n)$  unde  $n$  este numărul de jucători participanți.
- (2) Pe de altă parte, implementarea clasamentului poate fi realizată utilizând o **tabelă de dispersie deschisă** care reprezintă asocierea dintre nume și poziții (fig.9.4.2.4.b).



**Fig.9.4.2.4.b.** Implementarea clasamentului utilizând o tabelă de dispersie

- Se presupune că numărul de intrări în această tabelă este apropiat de numărul de jucători.
- Operatorul `Adauga(nume)` consumă în medie  $O(1)$  unități de tip.

- Operatorul **Pozitie( nume )** consumă de asemenea în medie  $O(1)$  unități de timp.
- **Provoaca( nume )** consumă  $O(1)$  unități de tip pentru căutarea numelui, dar necesită în schimb  $O(n)$  unități de tip pentru a găsi jucătorul situat pe poziția anterioară, întrucât trebuie parcursă întreaga tabelă de dispersie.
- **Interschimba( pozitie )** necesită de asemenea  $O(n)$  unității de timp pentru a găsi pozițiile  $i$  și  $i-1$ .
- (3) Se presupune în continuare că se combină cele două structuri de date, după cum rezultă din figura 9.4.2.4.c..



**Fig.9.4.2.4.c.** Implementarea clasamentului utilizând structuri de date combinate

- Celulele Tabelei de Dispersie vor conține informații referitoare la numele jucatorului și la poziția sa în tabloul Clasament.
- Tabloul Clasament va conține la Clasament[i] un pointer la celula corespunzătoare jucătorului din poziția i în Tabela de Dispersie.
- În acest nou context :
  - Un nume de jucator nou se adaugă inserându-l în tabela de dispersie în  $O(1)$  unități de timp în medie și de asemenea plasând pointerul celulei noi create în tabelul Clasament în poziția marcată de cursorul Pozitie Urmatoare, într-un interval redus constant de timp ( $O(1)$ ).
  - Operatorul **Pozitie** necesită de asemenea un efort de ordinul  $O(1)$  rezultat din accesul prin tabela de dispersie la jucătorul cu numele precizat și aflarea poziției în clasament memorată în celula aferentă.
  - Pentru a implementa funcția **Provoaca**, se caută numele jucătorului în tabela de dispersie, în medie în  $O(1)$  unități de timp, se obține rangul i al jucătorului

căutat și urmând indicația pointerului Clasament [ i-1 ] se ajunge la celula conținând numele jucătorului provocat.

- Consultarea lui Clasament [ i-1 ] consumă un interval constant de timp, căutarea în tabela de dispersie în medie O(1) unități de timp, deci în medie **Provoaca** necesită O(1) unități de timp.
- **Interschimba** (*pozitie*) consumă un interval constant de timp pentru a găsi celulele jucătorilor în tabloul Clasament situați pe *pozitie* respectiv *pozitie-1*, pentru a interschimba pozițiile în celulele corespunzătoare din Tabela de Dispersie și pentru a interschimba pointerii la cele două celule în tabloul Clasament.
  - Astfel, și operatorul **Interschimba** necesită un interval constant de timp chiar în cazul cel mai defavorabil.
- Acest exemplu, fără a fi deosebit de reprezentativ, are o **valoare intrinsecă deosebită**, el evidențиind în **mod limpede și clar** o **metodă conceptuală simplă**, care poate conduce la performanțe spectaculoase în ceea ce privește sporirea eficienței operatorilor care acționează asupra unei structuri de date, desigur într-un context specific.
- **Dezavantajul** este de asemenea evident: necesitatea multiplicării structurii în două sau mai multe reprezentări diferite care sunt prelucrate simultan.

## 9.5. Implementarea structuri mulțime cu ajutorul structurilor de date avansate

- În cadrul acestui paragraf vor fi precizate câteva modalități mai eficiente de implementare a structurii de date mulțime.
- Aceste modalități, mai complexe în principiu, se pretează implementării mulțimilor de mari dimensiuni și sunt bazate pe diferite categorii de arbori cum ar fi **arborii binari ordonați**, **arborii de căutare** și **arborii echilibrați**.

### 9.5.1. Implementarea structuri mulțime cu ajutorul arborilor binari ordonați

- **Arborii binari ordonați** pot fi utilizati în reprezentarea acelor mulțimi peste care este definită o **relație de ordonare** precizată de regulă prin operatorul "<".
  - Această modalitate de reprezentare este utilă în cazul mulțimilor ale căror elemente aparțin unui **univers extins**, care face practic imposibilă utilizarea elementelor mulțimii în calitate de indici direcți într-un tablou.
  - Un exemplu în acest sens îl constituie mulțimea identificatorilor posibili ai unui program.
- În implementarea bazată pe **arbori binari ordonați**, operatorii **Insereaza**, **Suprima**, **Apartine** și **Min** pot fi implementați fiecare în O(log<sub>2</sub>n) pași în medie, pentru o mulțime de cardinalitate n, element care rezultă din maniera în care este concepută, reprezentată și exploataată o astfel de structură.

- Structura **arbore binar ordonat** a fost prezentată detaliat în capitolul 8, paragraful &8.3.
  - Se reamintește că **proprietatea fundamentală** a acestui tip de arbore este aceea că toate elementele memorate în subarborele stâng al oricărui nod  $x$  sunt mai mici decât elementul memorat în  $x$  și toate elementele memorate în subarborele drept, sunt mai mari ca elementul memorat în  $x$ .
  - Această proprietate, definiție pentru **arborii binari ordonați** este valabilă pentru orice nod al unui astfel de arbore inclusiv pentru rădăcină.
- Se presupune spre exemplu, că reprezentarea unei mulțimi dicționar se realizează cu ajutorul unei structuri **arbore binar ordonat**, ca cea prezentată în secvența [9.5.1.a].
  - După cum se observă tipul **mulțime** se declară ca și un pointer la un nod. Acest nod este rădăcina arborelui binar ordonat care reprezintă mulțimea.

---

**/\* Reprezentarea mulțimilor cu ajutorul arborilor binari ordonați \*/**

```
typedef struct tip_nod
{
    tip_element element;
    struct tip_nod* stang;           /*[9.5.1.a]*/
    struct tip_nod* drept;
};

typedef tip_nod * ref_tip_nod;

typedef ref_tip_nod tip_multime;
```

---

- În acest context, implementarea operației **Apartine** este simplă:
  - Pentru a determina apartenența lui  $x$  la mulțime se execută o căutare în arborele binar asociat bazată pe tehnici clasice (&8.3.3).
  - O implementare recursivă a unei astfel de tehnici apare în funcția **Apartine**( $x, A$ ) din secvența [9.5.1.b].

---

**/\*Implementarea operatorului Apartine\*/**

```
boolean Apartine(tip_element x, tip_multime A)

/*caută recursiv în arborele binar A elementul x și
returnează true dacă x aparține lui A sau false în caz
contrar*/

{
```

```

        return false;
    else
        if (x==A->element)
            return true;
        else
            if (x<A->element)
                b=Apartine(x,b->stang); /*[9.5.1.b]*/
            else
                if (x>A->element)
                    b=Apartine(x,A->drept);
    } {Apartine}

```

---

- Operatorul ***Inserează***(*x*,*A*) care adaugă un element nou mulțimii este de asemenea simplu de implementat aplicând tehnicele de creare a arborilor binari ordonați.
    - O implementare posibilă apare prezentată în secvența [9.5.1.c]..
- 

```

/*Implementarea operatorului Insereaza - varianta
pseudocod*/

void Insereaza(tip_cheie x, tip_multime A)

/*inserează elementul x în arborele binar ordonat A*/

daca (A==NULL)
    /*insertie element x*/
    A=aloca_memorie(tip_multime); /*alocare nod nou*/
    A->element=x; A->stang=NULL; A->drept=NULL;
    □
  altfel                  /*[9.5.1.c]*/
    daca (x<A->element) /*parcursere arbore binar*/
        Insereaza(x,A->stang);
    daca
        Insereaza(x,A->drept);
    else
        return; /*daca x==A^.element, x aparține
                  deja mulțimii și nu se face nimic*/
/*Insereaza*/

```

---

- După cum se observă, dacă elementul de inserat *x*, aparține deja mulțimii, procedura ***Inserează*** nu face nimic.
- **Suprimarea** unui nod dintr-o structură arbore binar ordonat ridică unele probleme presupunând tratarea distinctă a cazurilor în care nodul de suprimit are:
  - (1) Un fiu sau niciunul.
  - (2) Doi fii.
- Prima situație nu ridică probleme, cea de-a doua însă necesită o prelucrare specială.

- O metodă de rezolvare a acestei situații o reprezintă înlocuirea nodului de suprimat cu **precedesorul său direct** la ordonarea în **inordine** și suprimarea nodului corespunzător precedesorului care nu are fiu drept, metodă prezentată în &8.3.5.
- O metodă similară, presupune înlocuirea nodului de suprimat cu **succesorul său direct** la ordonarea în **inordine** a cheilor și **suprimarea succesorului** care este un nod fără fiu stâng.
  - Aflarea **succesorului** se reduce de fapt la aflarea celui mai mic nod (cel mai din stânga) al **subarborelui drept** al arborelui care are drept rădăcină nodul de suprimat.
  - În acest scop se dezvoltă funcția **SupriMin** care returnează elementul minim al unui arbore suprimând nodul care-i corespunde (secvența [9.5.1.d] ).
  - În continuare bazat pe această metodă, se prezintă procedura **Suprimă** care suprimă un element precizat **x** din arborele **A** (secvența [9.5.1.e] ).
  - Se observă clar tratarea celor trei situații: nici un fiu, un fiu, sau doi fii.
    - În ultima situație se înlocuiește nodul cu successorul său direct determinat de funcția **SupriMin(A^.drept)**, adică cu nodul având cheia cea mai mică aparținând **subarborelui drept** al nodului în cauză.

---

**/\*Implementarea operatorului SupriMin - varianta pseudocod\*/**

```

tip_element SupriMin(tip_multime A)
/*returnează și suprimă cel mai mic element din A*/

tip_element x;
daca (A->stang==NULL)
    /*A indică cel mai mic element*/
    x=A->element;                         /*[9.5.1.d]*/
    A=A->drept;   /*suprimare nod*/
    return x;
    □

altfel
    x=SupriMin(A->stang);
/*SupriMin*/

```

---

**/\*Implementarea operatorului Suprimă - varianta pseudocod\*/**

```

void Suprimă(tip_element x, tip_multime A);

daca (A!=NULL)
    daca (x<A->element)
        Suprimă(x,A->stang);
    altfel
        daca (x>A->element)                  /*[9.5.1.e]*/
            Suprimă(x,A->drept);
        altfel /*pointerul A indică nodul x*/

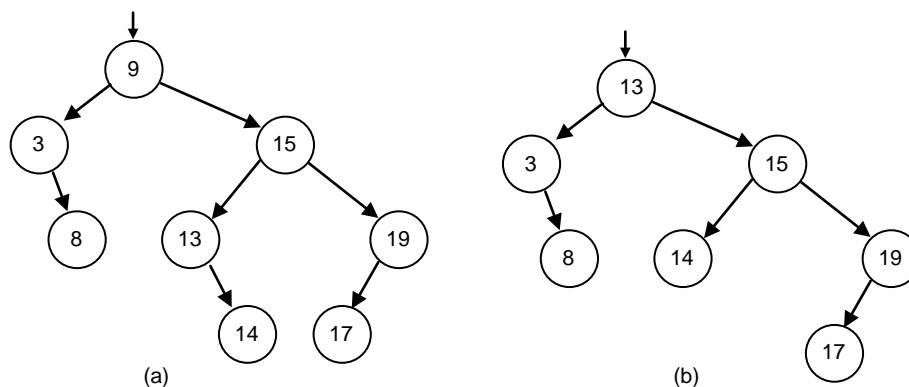
```

```

daca (A->stang==NULL) && (A->drept==NULL)
    A=NULL; /*ambii fii lipsesc*/
    altfel
        daca (A->stang==NULL)
            A=A->drept;
        altfel
            daca (A->drept==NULL)
                A=A->stang;
            altfel /*x are ambii fii*/
                A->element=Suprimin(A->drept);
/*Suprima*/

```

- În figura 9.5.1.a se prezintă un exemplu de arbore binar (a) din care s-a suprimat cheia 9 conform metodei prezentate (b).



**Fig.9.5.1.a.** Suprimarea unui nod dintr-o structură de arbore binar ordonat

#### **9.5.1.1. Considerații referitoare la performanțele implementării structurii multime cu ajutorul arborilor binari ordonați**

- Este ușor de remarcat că dacă arborele binar cu  $n$  noduri care reprezintă **structura multime** este un **arbore binar complet** (toate nodurile cu excepția celor de pe ultimul nivel au câte doi fiți), atunci operațiile **Insereaza**, **Suprima**, **Apartine** și **Min** necesită cel mult  $O(\log_2 n)$  pași, întrucât în cel mai rău caz aceste operații necesită parcurgerea tuturor nivelurilor arborelui.
  - De regulă, însă, arborii binari ordonați a căror formă depinde de ordinea de inserare a cheilor, nu sunt arbori binari compleți.
    - În cel mai defavorabil caz, un astfel de arbore poate degenera într-o structură **listă liniară** în care operațiile de mai sus necesită  $O(n)$  pași, pentru un arbore cu  $n$  noduri.
  - Cu alte cuvinte în realitate, operatorii **Insereaza**, **Suprima**, **Apartine** și **Min** implementați cu ajutorul **structurii arbore binar ordonat** necesită un efort de calcul cuprins între  $O(\log_2 n)$  și  $O(n)$ .
  - Analiza detaliată a căutării în arborii binari ordonați prezentată în §8.3.6 demonstrează însă că în medie, arborii binari ordonați reali sunt cu 39% mai înalți ca și arborii binari compleți.

- Concluziile prezentate în cadrul respectivei analize rămân valabile și pentru situația de față.
- Astfel se poate concluziona că **testarea apartenenței** unui element la o mulțime, **inserția** unui nou element într-o mulțime, **suprimarea** unui element al unei mulțimi și **căutarea** elementului minim al unei mulțimi necesită în medie  $O(\log_2 n)$  pași în implementarea mulțimilor bazată pe **arbori binari ordonați**.
- În comparație cu implementările bazate pe **alte tipuri de structuri de date** se pot face următoarele observații.
  - (1) Implementarea bazată pe **tabele de dispersie** a structurii dicționar necesită în medie un timp constant pentru operațiile definite.
    - Deși această performanță este superioară cele obținute în implementările bazate pe arbori binari ordonați, o **tabelă de dispersie** necesită  $O(n)$  pași pentru implementarea operației **Min**.
    - Astfel, dacă operația **Min** se utilizează des, atunci **arborii binari ordonați** sunt mai potrivici, dacă **Min** nu este necesară se preferă **tehnica dispersiei**.
  - (2) Referitor la implementarea structurii dicționar, structura **arbore binar ordonat** poate fi comparată și cu structura **arbore binar parțial ordonat** (ansamblu) utilizat în implementarea cozilor bazate pe prioritate.
    - Într-un **arbore binar parțial ordonat** (ansamblu) cu  $n$  elemente sunt necesari  $O(\log_2 n)$  pași pentru operatorii **Insereaza** și **Extrage** (suprimă minimul) nu numai în medie ci chiar în cel mai defavorabil caz, astfel din acest punct de vedere **arborii binari parțial ordonați** sunt de preferat.
    - Totuși **arborii binari ordonați** permit implementarea la fel de performantă a operațiilor **Suprima**, **Min** cât și a combinației **SupriMin**, în timp ce **arborii parțial ordonați** permit numai ultimele două operații.
    - În plus, **Apartine** necesită  $O(n)$  pași într-un **arbore binar parțial ordonat** respectiv numai  $O(\log_2 n)$  pași în **arborii binari ordonați**.
    - Astfel, dacă **arborii binari parțial ordonați** sunt foarte potrivici în implementarea cozilor bazate pe prioritate, ei **nu** pot fi utilizați cu aceeași eficiență ca și **arborii binari ordonați** dacă este necesară implementarea unui set extins de operatori (ca și în cazul mulțimilor).

### 9.5.2. Implementarea structurii mulțime cu ajutorul arborilor de regăsire

- După cum s-a mai precizat, **arborii de regăsire** sunt structuri de date speciale care pot fi utilizate în reprezentarea mulțimilor cu deosebire a celorale ale căror elemente sunt caractere.
- De asemenea cu ajutorul lor pot fi reprezentate tipuri de date care sunt **șiruri de obiecte** de orice tip sau șiruri de numere.

- În literatura de specialitate această categorie de arbori de regăsire sunt cunoscuți sub denumirea de structuri de tip **trie** cuvânt derivat din cuvântul “**“retrieval”**” (**regăsire**) (paragraful &8.4).
- Așa cum s-a precizat în paragraful sus amintit, un **arbore de regăsire** permite implementarea simplă a operatorilor definiți asupra unei **structuri de date mulțime** ale cărei elemente sunt **șiruri de caractere** (cuvinte).
  - Este vorba în principiu despre operatorii ***Inserează***, ***Suprimă***, ***Inițializează*** și ***Afișează***.
  - Ultimul operator realizează afișarea tuturor membrilor (cuvintelor) mulțimii.
- Utilizarea arborilor de regăsire este eficientă atunci când există mai multe cuvinte care încep cu aceeași secvență de litere, adică atunci când **numărul de prefixe distințe** al tuturor cuvintelor din mulțime este mult mai redus decât **numărul total de cuvinte**.
- În paragraful 8.4 se prezintă la nivel de detaliu structura de date **Nod ArboreDeRegasire** utilizată ca suport de reprezentare pentru structura **Arbore DeRegăsire**.
  - În acest context s-au prezentat două modalități de implementare una bazată pe **tablouri** și o a doua bazată pe **liste liniare**.
  - Pentru exemplificare s-a prezentat implementarea operatorului ***Insereaza***, iar în finalul paragrafului menționat s-a realizat o analiză comparată a performanțelor structurii **ArboreDeRegăsire** respectiv a tehnicii dispersiei în contextul utilizării lor la implementarea mulțimilor de cuvinte.
  - Analiza realizată a evidențiat faptul că **arborii de regăsire** pot fi mai performanți decât **tablele de dispersie**, ei constituind o posibilitate preferențială de implementare eficientă a **structurii dicționar**.

### 9.5.3. Implementarea structurii mulțime cu ajutorul arborilor binari ordonați echilibrați

- Este cunoscut faptul că performanța prelucrării unor **structuri de date arbore** este proporțională cu înălțimea acestora.
- În situația în care un arbore cu  $n$  noduri este un **arbore binar plin, complet** sau în general **de înălțime minimă**, performanța prelucrării obține în cel mai defavorabil caz, valoarea  $O(\log_2 n)$ .
- Întrucât în procesul de exploatare, înălțimea arborilor evoluează în mod aleator și ea este în medie cu 39 % mai mare ca și a arborilor de înălțime minimă (&8.3.6), au fost dezvoltate structuri speciale de arbori a căror înălțime evoluează în mod controlat și este menținută în jurul înălțimii minime.
- În această categorie se încadrează **arborii binari ordonați echilibrați** pentru care efortul de prelucrare este în cel mai defavorabil caz  $O(\log_2 n)$ , în medie mai redus.
- Din păcate complexitatea algoritmilor care prelucrază astfel de arbori **nu** justifică întotdeauna utilizarea lor.

- În această categorie se includ **arborii AVL**, **arborii binari optimi**, **arborii B**, **arborii 2-3** etc.
- Fiecare dintre aceste **structuri arbore** a fost prezentată la nivel de detaliu din punctul de vedere al proprietăților specifice, al modalităților de implementare și al performanțelor aferente.
- Ca atare, ori care dintre aceste **structuri arbore** poate fi utilizată pentru a implementa în mod performant, într-un context specific, **structura de date mulțime**.

## 9.6. Mulțimi pe care sunt definiți operatorii UNIUNE și CAUTĂ

- În continuare se propune modelarea următorului **scenariu**:
  - Se presupune că există **o colecție de obiecte unice** fiecare dintre ele aparținând unei anumite **mulțimi**.
  - Se **combină** aceste mulțimi într-o ordine oarecare prin operații **Uniune** și din timp în timp se cere să se precizeze **căreia dintre mulțimi** îi aparține un **obiect specificat**.
- Această scenariu poate fi soluționat utilizând mulțimi pe care sunt definite operațiile **Uniune** și **Caută**.
- Se reamintește definirea celor doi operatori:
  - Operatorul **Uniune** ( $A, B, C$ ) atribuie lui  $C$  mulțimea rezultată din reuniunea mulțimilor  $A$  și  $B$  care sunt **mulțimi disjuncte** (nu au elemente în comun) (vezi §9.2).
    - Operatorul **Uniune** nu este definit dacă  $A$  și  $B$  nu sunt disjuncte, element care diferențiază această operație de reuniune normală a mulțimilor.
  - **Caută** ( $x$ ) este o funcție care returnează numele (unic) al mulțimii căreia îi aparține obiectul  $x$ .
    - Dacă  $x$  apare în mai multe mulțimi sau în niciuna, operatorul **Caută** nu este definit (vezi §9.2)..

### 9.6.1. Implementarea bazată pe tablouri a mulțimii pe care sunt definiți operatorii UNIUNE și CAUTĂ

- Se consideră o structură abstractă de date **MulțimeDeSubmulțimi** constând dintr-o **mulțime de submulțimi disjuncte** care se vor numi **componente**, peste care sunt definiți următorii **operatori** [9.6.1.a]:
 

---

1. **Uniune**(*TipNumeSubmulțime A, TipNumeSubmulțime B*) - realizează uniunea componentelor *A* și *B* denumind rezultatul uniunii *A* sau *B*, în mod arbitrar. [9.6.1.a]
  2. *TipNumeSubmulțime Caută*(*TipElementSubmulțime x*) - este o funcție care returnează numele componentei căreia îi aparține elementul *x*.
  3. **Initializeaza**(*TipNumeSubmulțime A, TipElementSubmulțime x*) - creează o submulțime componentă numită *A*, care conține numai elementul *x*.
- 

- Pentru a realiza o implementare rezonabilă a structurii *MultimeDeSubmultimi* este necesar să se observe că tipul **mulțime de submulțimi** include alte două tipuri:
    - (1) Tipul corespunzător **numelui submulțimilor componente** - *TipNumeSubmultime*.
    - (2) Tipul corespunzător **elementelor submulțimilor** - *TipElementSubmultime*.
  - În cele ce urmează, se vor utiliza **întregi** pentru a preciza atât **numele submulțimilor componente** cât și **elementele** acestora.
    - Dacă *n* este numărul total de elemente, atunci elementele submulțimilor aparțin **domeniului**  $0..n-1$ .
    - Pentru implementare, este important ca tipul elementelor să fie un tip **subdomeniu**, deoarece el va fi utilizat ca indice într-un **tablou** care materializează corespondența element-submulțime.
      - Elementele acestui tablou sunt **nume de submulțimi**.
      - Astfel dacă *M* este o variabilă încadrată în tipul *MultimeDeSubmulțimi*, atunci  $M[i]=j$  precizează că **elementul i** aparține **submulțimii cu numele j**.
    - Tipul corespunzător numelor submulțimilor componente în principiu **nu** este supus restricțiilor, însă constantele sale sunt elemente de tablou și nu indici.
    - Trebuie subliniat faptul că dacă tipul elementelor submulțimilor este altul decât tipul subdomeniu, trebuie definită o **asociere**, eventual printr-o tabelă de dispersie, care să realizeze o corespondență de genul element-indice de tablou.
  - Cunoscând în avans numărul total de elemente, se poate defini drept suport al reprezentării mulțimilor pe care sunt definiți operatorii **Uniune** și **Caută** structura de date din secvența [9.6.1.a] sau varianta generalizată a acesteia din secvența [9.6.1.b]
- 

/\*Mulțimi pe care sunt definiți operatorii UNIUNE și CAUTA

### **Implementare bazata pe tablouri varianta C\*/**

```
#define N {număr total de elemente}; [9.6.1.a]

typedef int Multime_De_Submultimi[N];

Multime_De_Submultimi M;
-----  

{Mulțimi pe care sunt definiți operatorii UNIUNE și CAUTA  

Implementare bazata pe tablouri - varianta PASCAL}

CONST n = {număr total de elemente}; [9.6.1.a]

type MultimeDeSubmultimi = array[1..n] OF INTEGER;
-----  

{Mulțimi pe care sunt definiți operatorii UNIUNE și CAUTA  

Implementare bazata pe tablouri Varianta generalizată  

PASCAL}

type MultimeDeSubmultimi = array[SubdomeniuElemente] OF
    TipNumeSubmultime; [9.6.1.b]
var M: MultimeDeSubmultimi;
-----
```

- În continuare, se presupune că se declară variabila M de tip MultimeDeSubmultimi, cu precizarea că M[x] memorează numele submulțimii căreia îi aparține la momentul considerat elementul x.
- Operatorii **Uniune**, **Cauta** și **Initializeaza** sunt simplu de implementat.

- Spre exemplu implementarea operatorului **Uniune** apare în secvența [9.6.1.c].

```
/*Implementarea operatorului Uniune*/

subprogram Uniune(TipNumeSubmultime A, TipNumeSubmultime B,
                    MultimeDeSubmultimi M)

    int x;
    pentru (x=0 la n-1)
        daca (M[x]==B) [9.6.1.c]
            M[x]= A;
    /*Uniune*/
-----
```

- Într-o manieră similară, **Initializeaza**(A, x) atribuie lui M[x] valoarea A.
- **Cauta**(x) returnează valoarea lui M[x].
- Performanța raportată la timp a acestei implementări poate fi simplu apreciată. Astfel:

- Execuția lui ***Uniune*** consumă  $O(n)$  unități de timp.
- ***Caută(x)*** și ***Initializeaza(A,x)*** consumă intervale constante de timp de execuție ( $O(1)$ ).

### 9.6.2. Implementarea bazată pe liste înlăntuite a mulțimii pe care sunt definiți operatorii **UNIUNE** și **CAUTA**

- Se pornește de la observația că în cel mai defavorabil caz pentru a introduce  $n$  elemente definite ca mai sus într-o singură submulțime, sunt necesare  $n-1$  execuții ale operatorului ***Uniune***, primul element al fiecarei mulțimi fiind introdus printr-o operație ***Initializeaza***.
  - Utilizând algoritmul din secvența anterioară [9.6.1.c], cele  $n-1$  execuții vor consuma  $O(n^2)$  unități de timp.
- O cale de creștere a vitezei de execuție a operatorului ***Uniune*** este aceea de a lega într-o listă înlăntuită toți membrii unei submulțimi.
  - Astfel, în loc de a parurge toți membrii ambelor submulțimi la realizarea uniunii A cu B, se va parurge numai lista elementelor lui B care vor fi introduse în mulțimea A.
- În medie această soluție salvează timp dar și aici poate apărea o situație defavorabilă.
  - Dacă se presupune că la cea de-a  $i$ -a execuție a operatorului se execută ***Uniune(A,B)*** unde A este o submulțime de dimensiune 1, B este o submulțime de dimensiune  $i$ , iar rezultatul va fi mulțimea A, această operație presupune  $O(i)$  unități de timp.
  - Cel mai defavorabil caz constă dintr-o secvență de  $n-1$  astfel de uniuni, în care de fiecare dată se adaugă unei submulțimi cu **un element**, o altă submulțime care după fiecare execuție crește cu un element.
  - Timpul necesar execuției unei astfel de secvențe este precizat în [9.6.2.a] adică un efort de ordinul  $O(n^2)$ .

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

[9.6.2.a]

- Pentru a evita această situație defavorabilă, implementarea operatorului ***Uniune*** trebuie să realizeze întotdeauna mutarea elementelor submulțimii cu un număr mai mic de elemente în cea cu un număr mai mare.

- Prin urmare, de fiecare dată când un element este adăugat unei submulțimi prin operația ***Uniune***, el va aparține unei submulțimi de cel puțin două ori mai mari ca și cea din care provine.
- Astfel, dacă există inițial  $n$  submulțimi fiecare cu câte un element, nici unul din cele  $n$  elemente nu își poate schimba submulțimea căreia îi aparține de mai mult de  $1 + \log_2 n$  ori.
- Întrucât timpul necesar execuției noii versiuni a operației ***Uniune*** este proporțional cu numărul de elemente care își schimba numele submulțimii căreia îi aparțin, rezultă că **numărul total** de astfel de schimbări este cel mult  $n^*(1 + \log_2 n)$ .
- Drept consecință, toate operațiile de tip ***Uniune*** necesare pentru crearea unei mulțimi cu  $n$  elemente, vor necesita maximum  $O(n \log_2 n)$  unități de timp.
- Pentru implementarea acestei soluții se propun următoarele **structuri de date**:
  - (1) Tabloul **inceputuri** care conține câte o intrare pentru fiecare submulțime. Fiecare element al acestui tablou este un articol având următoarea structură:
    1. Câmpul **contor** care precizează numărul de elemente ale submulțimii.
    2. Câmpul **primulElement** de tip index în tabloul **nume**, câmp care indică primul element al listei membrilor submulțimii.
  - (2) Tabloul **nume** păstrează membrii submulțimilor ca liste înlăntuite prin intermediul cursorilor. Fiecare element aparținând unei submulțimi îi corespunde o intrare în tablou cu următoarea structură:
    1. Câmpul **numeSubmulțime** care păstrează numele submulțimii căreia îi aparține elementul respectiv.
    2. Câmpul de înlătuire **urm** de tip index în tabloul **nume**, care precizează următorul membru al submulțimii respective. Indexul corespunzător lui **NULL** se notează cu 0 (zero).
- În cazul special în care atât numele submulțimilor cât și elementelor lor aparțin subdomeniului  $1..n$ , pentru implementarea mai sus descrisă se pot defini structurile de date din secvența [9.6.2.a].

---

/\*Mulțimi pe care sunt definiți operatorii Uniune și Caută  
Implementare bazată pe liste înlăntuite implementate cu  
ajutorul cursorilor - structuri de date - varianta C\*/

```
#define N {număr total de elemente};

typedef struct tip_multime
{
    int contor; /*numărul curent de elemente al mulțimii*/
```

```

        int primulElem; /*cursor la primul element în
    } multime;           tabloul nume*/
}

typedef struct tip_element
{
    int numeSubmultime; /*numele submulțimii cărei îi
                           aparține elementul*/
    int urm; /*cursor la elementul următor în tabloul
               nume*/
} element;                                     [9.6.2.a]

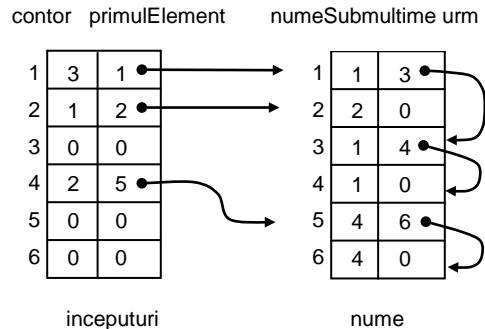
typedef struct MultimeDeSubmultimi
{
    tip_multime inceputuri[N]; /*tablou conținând
                                incepurile listelor corespunzătoare submulțimilor*/
    tip_element nume[N]; /*tablou care precizează pentru
                           fiecare element submulțimea căreia îi aparține;
                           elementele aparținând aceleiași submulțimi sunt
                           înlántuite*/
} multime_de_submultimi;

-----{Multimi pe care sunt definiți operatorii Uniune și Caută
      Implementare bazată pe liste înlántuite implementate cu
      ajutorul cursorilor - structuri de date - varianta PASCAL}

type TipNumeSubmultime = 1..n;
TipElementSubmultime = 1..n;

MultimeDeSubmultimi = record {mulțime de submulțimi}
    inceputuri: array[1..n] OF
        {tablou conținând incepurile listelor
         corespunzătoare submulțimilor}
    record {început de listă}
        contor: 0..n; {număr curent de elemente}
        primulElem: 0..n {cursor în tabloul nume}
    end;
    nume: array[1..n] OF                               [9.6.2.a]
        {tablou care precizează pentru fiecare element
         submulțimea căreia îi aparține; elementele
         aparținând aceleiași submulțimi sunt
         înlántuite}
    record {componenta tablou}
        numeSubmultime: TipNumeSubmultime;
        urm: 0..n {cursor în tabloul Nume}
    end
end; {MultimeDeSubmultimi}
-----
```

- În figura 9.6.2.a. apare prezentat un exemplu de structură de date de tip **mulțime de submulțimi** unde submulțimea 1 este {1,3,4}, submulțimea 2 este {2} iar submulțimea 4 este {5,6}, exemplu bazat pe modelul de structură de date propus.



**Fig.9.6.2.a.** Implementarea structurii mulțime de submulțimi cu ajutorul listelor înlăncuite bazate pe cursori

- Procedurile **Initializare**, **Uniune** și **Caută** apar în secvența [9.6.2.b].

```

/*Mulțimi pe care sunt definiți operatorii Uniune și Caută
Implementarea operatorilor Initializare, Uniune și Caută
- varianta C*/

```

```

typedef int Tip_Nume_Submultime;
typedef int Tip_Element_Submultime;

void Initializare(Tip_Nume_Submultime A,
                    Tip_Element_Submultime x, MultimeDeSubmultimi * M)
/*initializează pe A ca o submulțime care-l conține numai pe
 x*/
{
    (*M).nume[x].numeSubmultime=A;
    (*M).nume[x].urm=0; /*sfârșitul listei elementelor
                          lui A*/
    (*M).inceputuri[A].contor=1;
    (*M).inceputuri[A].primulElem=x;
} /*Initializarea*/

void Uniune (Tip_Nume_Submultime A,
                Tip_Nume_Submultime B, MultimeDeSubmultimi * M)

/*realizează uniunea lui A și B denumind submulțimea
rezultată în mod arbitrar A sau B*/
{
    int i; /*utilizat în găsirea sfârșitului celei mai
            scurte liste*/

    if ((*M).inceputuri[A].contor>
          (*M).inceputuri[B].contor)
    { /*A este cea mai mare mulțime, deci se adaugă
       B la A*/
        i=(*M).inceputuri[B].primulElem;
        do /*se parcurge mulțimea B modificând pentru
              fiecare element al său, numele submulțimii
              din B în A*/
        {

```

```

        (*M).nume[i].numeSubmultime=A;
        i=(*M).nume[i].urm;                                /*[9.6.2.b]*/
    } while ((*M).nume[i].urm==0);
/*înlănțuie lista A la sfârșitul lui B și modifică
   numele listei B în A. Se șterge lista B*/
    (*M).nume[i].numeSubmultime=A; /*i indică ultimul
                                   element al lui B*/
    (*M).nume[i].urm=(*M).inceputuri[A].primulElem;
    (*M).inceputuri[A].primulElem=
        (*M).inceputuri[B].primulElem;
    (*M).inceputuri[A].contor=(*M).inceputuri[A].contor
        +(*M).inceputuri[B].contor;
    (*M).inceputuri[B].contor=0;
    (*M).inceputuri[B].primulElem=0 /*sterge multimea
                                    B*/
}
/*if*/
else /*B este cel puțin tot așa de mare ca A, se
      adaugă A la B*/
{
    *secvență similară cu cea de mai sus,
      interschimbând însă pe B cu A;
}
/*else*/
} /*Uniune*/

```

```

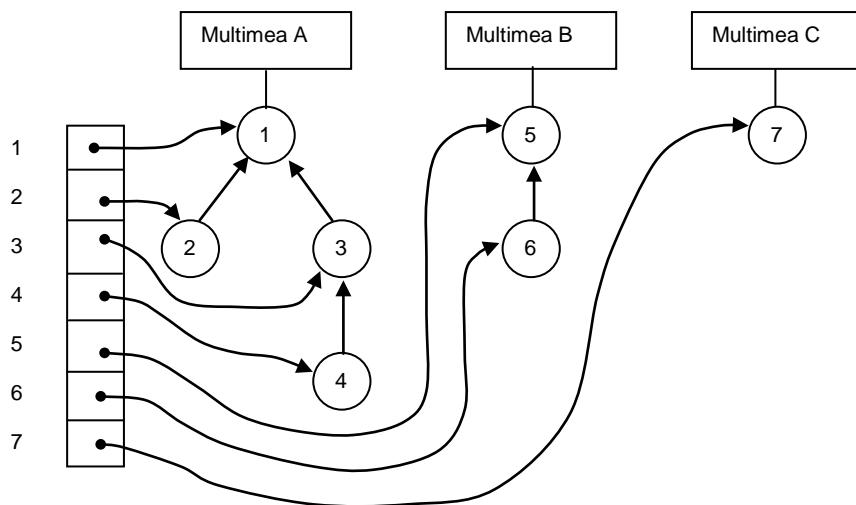
Tip_Nume_Submultime Cauta(Tip_Element_Submultime x,
                           MultimeDeSubmultimi * M)
/*returneaza numele submultimii căreia îi aparține x*/
{
    return((*M).nume[x].numeSubmultime)
} /*Cauta*/
-----
```

- Referitor la implementarea operatorului **Uniune** se fac următoarele precizări.
  - Inițial se verifică care dintre mulțimi conține mai puține elemente și se adaugă această submulțime celeilalte.
  - Prima parte a algoritmului tratează situația în care B are mai puține elemente ca și A iar partea a doua situația inversă.
    - În cazul în care A are mai multe elemente decât B, tuturor elementelor mulțimii B li se schimbă numele în tabloul nume , în A (bucla **do - while**).
    - Se înlănțuie lista A la sfârșitul listei B modificate.
    - Se modifică începutul listei A astfel încât să indice începutul listei B modificate, iar contorul corespunzător să indice suma contoarelor celor două liste.
    - În final se șterge lista B din tabloul **inceputuri** .

- Cazul următor se tratează identic cu deosebirea că se interschimbă A cu B.

### 9.6.3. Implementare bazată pe arbori a mulțimii pe care sunt definiți operatorii UNIUNE și CAUTĂ

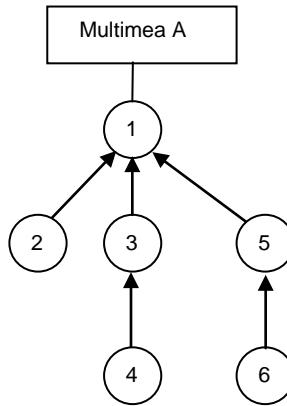
- O altă metodă de implementare a mulțimilor peste care sunt definiți operatorii **Uniune** și **Caută** se bazează pe **structura arbore** implementată în varianta **indicator spre părinte** (vezi &8.4.1.4).
- Descrierea acestei metode se va realiza numai la **nivel de principiu**.
  - Astfel se presupune că nodurile structurii arbore conțin elementele unei mulțimi sau eventual referiri la aceste elemente prin intermediul unei asocieri.
  - Fiecare nod, cu excepția rădăcinii conține un **indicator spre părintele** său.
  - Rădăcina păstrează **numele mulțimii**.
- Asocierea dintre numele mulțimilor și rădăcinile arborilor care memorează mulțimile, permite accesul simplu la o anumită mulțime în cazul realizării operației **Uniune**.
  - În fig. 9.6.3.a sunt prezentate mulțimile  $A = \{1, 2, 3, 4\}$ ,  $B = \{5, 6\}$  și  $C = \{7\}$  reprezentate în acest mod.
  - Dreptunghiurile care conțin numele mulțimilor nu sunt considerate noduri separate, ci se presupune că fac parte din structura nodului rădăcină.



**Fig.9.6.3.a.** Mulțime de submulțimi reprezentată ca și o colecție de arbori

- Pentru a afla **numele mulțimii** care conține un anumit element x:

- (1) Pe baza asocierii, se determină nodul din arbore care îl conține pe  $x$ .
- (2) Urmând drumul de la nod spre rădăcină se află numele mulțimii care conține elementul.
- Pentru a realiza **uniunea a două mulțimi** este suficient că rădăcina arborelui corespunzător uneia dintre mulțimi să devină fiul rădăcinii arborelui corespunzător celeilalte.
- Astfel unind mulțimea A și B din fig. 9.6.3.a se obține mulțimea din fig. 9.6.3.b.



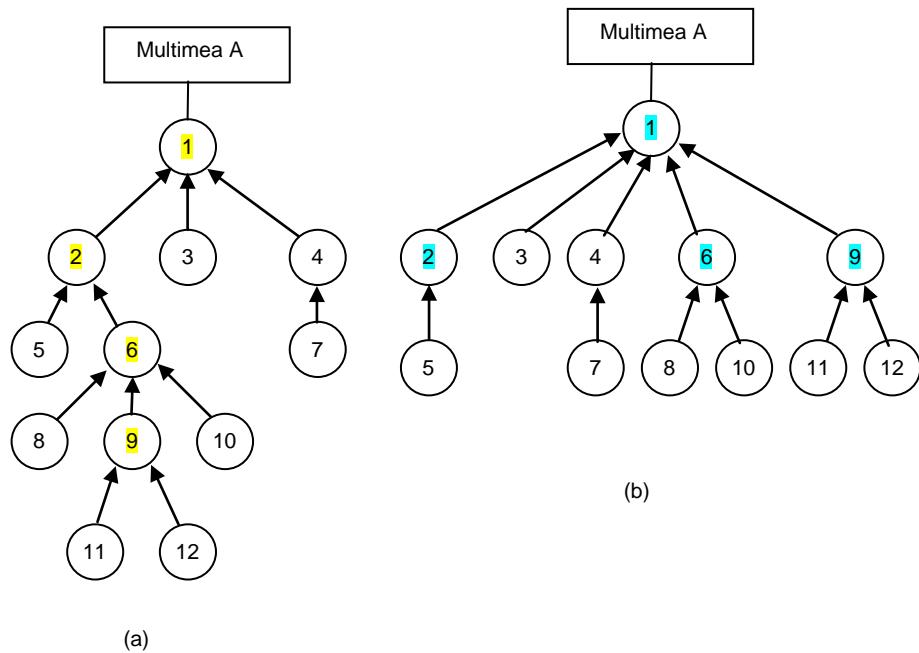
**Fig.9.6.3.b.** Uniunea mulțimilor A și B

- În cazul cel mai **defavorabil** rezultă un arbore degenerat într-o listă liniară cu  $n$  noduri.
  - În această situație se ajunge executând în mod repetat operația **Uniune** între o mulțime conținând un singur element și o altă submulțime generată în aceeași manieră.
  - Execuția operației **Căuta** asupra tuturor elementelor necesită în acest caz  $O(n^2)$  unități de timp.
  - Întrucât operația **Uniune** se execută în  $O(1)$  unități de timp, performanța de ansamblu a acestei implementări depinde direct de numărul de operații de căutare efectuate.
- O cale de **creștere a acestei performanțe** este aceea de a face cât mai eficientă operația de căutare.
  - În acest scop fiecărei rădăcini i se adaugă un **câmp suplimentar** care precizează **numărul de elemente ale mulțimii**.
  - Când se realizează uniunea a două mulțimi se va proceda astfel încât rădăcina arborelui cu număr mai mic de elemente să devină fiul rădăcinii arborelui cu mai multe elemente.

- Astfel de fiecare dată când un nod este mutat printr-o operație **uniune** într-un nou arbore se întâmplă două lucruri:
  - (1) Distanța de la nod la rădăcină crește cu o unitate
  - (2) Un nod al mulțimii cu mai puține elemente devine membru al unei mulțimi care este cel puțin de două ori mai mare decât cea din care provine.
- Prin urmare, dacă numărul total de elemente este  $n$ , nici un nod nu poate fi mutat de mai mult de  $\log_2 n$  ori.
  - Rezultă că distanța de la un nod al structurii la rădăcină nu poate depăși, niciodată  $\log_2 n$ .
  - În consecință, performanța operației de căutare a unui nod devine  $O(\log_2 n)$ , iar performanța totală corespunzătoare căutării tuturor nodurilor  $O(n \log_2 n)$ .

#### **9.6.3.1 Comprimarea drumului.**

- O altă idee care poate contribui la creșterea performanței implementării mulțimilor pe care sunt definiți operatorii **uniune** și **Caută** este **comprimarea drumului** ("path compression") [AH85].
  - Conform acestei metode în timpul execuției operației de căutare, când se parcurge drumul de la nod spre rădăcină, fiecare nod întâlnit de-a lungul acestui parcurs se face fiu al rădăcinii.
  - Acest lucru se poate realiza în două trecheri:
    - (1) La prima trecere se află rădăcina.
    - (2) La a doua trecere se reparcurge drumul făcând din fiecare nod un fiu al rădăcinii.
  - În figura 9.6.3.1.a (b) se prezintă structura modificată a arborelui (a), obținută în urma execuției operației **Caută** pentru elementul 9.
  - După cum se observă nodurile 1 și 2 nu sunt afectate deoarece nodul 1 este chiar rădăcina iar 2 este fiul acestuia.



**Fig.9.6.3.1.a.** Exemplu de comprimare a drumului

- Comprimarea drumului **nu** afectează performanța operatorului **Uniune** în schimb îmbunătășește progresiv performanța operatorului **Caută**, prin scurtarea drumului parcurs, cu un efort relativ redus.
- Analiza performanței medii a operației de căutare, atunci când se utilizează comprimarea drumului este o operație foarte dificilă.
  - În cazul în care arborele este degenerat în lista liniară **Caută** poate necesita  $O(n)$  unități de timp, însă comprimarea drumului poate modifica în timp structura arborelui astfel încât căutarea tuturor elementelor mulțimii, respectiv toate cele  $n$  operații **Caută** să necesite în total un efort de calcul de ordinul  $O(n)$ .
- Algoritmul care utilizează atât comprimarea drumului, cât și adăugarea arborelui mai mic celui mai mare, este în mod asymptotic, **cea mai eficientă metodă** cunoscută de implementare a mulțimilor peste care sunt definite operațiile **Uniune** și **Caută**.

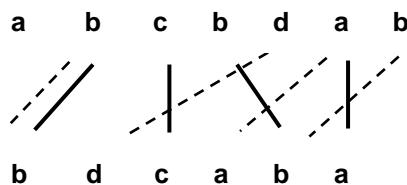
## 9.7. Mulțimi pe care sunt definiți operatorii UNIUNE, CAUTĂ și PARTIȚIONARE

- Fie  $M$  o mulțime ale cărei elemente sunt ordonate de o relație " $<$ ".
- Peste această mulțime se consideră definiții operatorii **Uniune** și **Caută** precizați anterior precum și operatorul **Partitionare**.
- Operatorul **Partitionare**  $(M, M_1, M_2, x)$  îl divide pe  $M$  în două mulțimi  $M_1 = \{e \mid e \in M \text{ și } e < x\}$  respectiv  $M_2 = \{e \mid e \in M \text{ și } e \geq x\}$ .
  - Valoarea lui  $M$  după diviziune este **nedefinită**, mai puțin în cazul în care ea este una din mulțimile  $M_1$  sau  $M_2$ .

- Cu alte cuvinte operatorul **Partitionare** separă mulțimea M în două submulțimi:
  - (1) Prima mulțime  $M_1$  conține toate elementele lui M **mai mici** decât elementul x furnizat ca parametru
  - (2) A doua mulțime  $M_2$  conține toate elementele lui M cu valori **mai mari sau egale** cu x.

### 9.7.1. Problema celei mai lungi subsecvențe comune (LCS)

- În cele mai multe situații **operația de partitioare a unei mulțimi** rezidă în compararea fiecărui element al mulțimii cu o valoare fixă x.
  - În continuare se va aborda o astfel de problemă.
- Se numește **subsecvență** a unei secvențe x, un sir format din 0 sau mai multe elemente, **nu neapărat contigue**, extras din x.
  - Fiind date două secvențe x și y se numește **cea mai lungă subsecvență comună** a celor două secvențe ("longest common subsequence" – LCS), cea mai lungă secvență care este în același timp subsecvență atât pentru x cât și pentru y.
- Spre exemplu, un LCS al secvențelor  $a, b, c, b, d, a, b$  și  $b, d, c, a, b, a$  este subsecvența  $b, c, b, a$  obținută după cum rezultă din figura 9.7.1.a.
  - Mai există și alte LCS-uri, de lungime 4, spre exemplu  $b, d, a, b$  dar nu există nici o subsecvență comună de lungime 5.



**Fig.9.7.1.a.** Exemplu de cea mai lungă subsecvență comună.

- În **UNIX** există comanda DIFF care compară fișierele linie cu linie și află cea mai lungă subsecvență comună, unde o **linie a fișierului** este considerată drept un element al subsecvenței.
  - Premiza de la care pornește comanda DIFF este următoarea: se presupune că liniile care nu apar în LCS sunt liniile inserate, șterse sau modificate care diferențiază cele două fișiere.

- Dacă spre exemplu cele două fișiere sunt două versiuni ale unui același program, DIFF va găsi cu mare probabilitate modificările efectuate.
- Există mai multe soluții cu caracter de generalitate care rezolvă **problema LCS** de regulă în  $O(n^2)$  pași pentru o subsecvență de lungime n.
- Comanda DIFF utilizează o **strategie** diferită care este foarte eficientă când fișierele nu au prea multe repetări ale aceleleași linii.
- Algoritmul utilizat de DIFF face uz de o implementare eficientă a mulțimilor pe care sunt definiți operatorii **Uniune**, **Caută** și **Partitionare** și se execută în  $O(p \log_2 n)$  unități de timp unde:
  - n este numărul maxim de linii ale fișierului.
  - p este numărul de perechi de poziții, câte una din fiecare fișier, care conțin aceeași linie.
- Spre exemplu p în figura 9.7.1.a are valoarea  $4+6+1+1=12$  respectiv:
  - Cei doi de a din fiecare sir contribuie cu  $2 \times 2 = 4$  perechi
  - b contribuie cu  $3 \times 2 = 6$  perechi
  - c și d cu câte  $1 \times 1 = 1$  pereche.
- În cel mai rău caz p poate fi  $n^2$  iar algoritmul va consuma  $O(n^2 \log_2 n)$  unități de timp.
  - Acest lucru se întâmplă atunci când A este identic cu B și toate elementele sale sunt identice.
  - În practică însă p este apropiat de n rezultând o eficiență de ordinul  $O(n \log_2 n)$ .

### 9.7.2. Determinarea LCS utilizând mulțimi pe care sunt definiți operatorii UNIUNE, CAUTĂ și PARTIȚIONARE

- Fie  $A=a_1a_2\dots a_n$  și  $B=b_1b_2\dots b_m$  două secvențe pentru care se dorește găsirea secvenței LCS.
- Algoritmul pentru determinarea LCS constă în **trei** pași.
  - (1) **Pasul 1.** Pentru fiecare simbol a aparținând secvenței A, se identifică poziția sa în cadrul secvenței.
    - În acest scop se definește structura  $\text{Poziții}(a) = \{ i \mid a_i = a \}$ .

- Se face precizarea că această corespondență **nu** este biunivocă, un același element a poate să apară pe mai multe poziții în cadrul secvenței A.
- După cum se observă, Poziții constă din mai multe **mulțimi de indici**, câte una pentru fiecare element distinct al secvenței A.
- Aceste mulțimi conțin pozițiile în cadrul secvenței ale respectivului element exprimate ca indici.
- Înregistrarea mulțimii pozițiilor se poate realiza în două moduri:
  - a) Prin intermediul unei **asociieri** între simboluri și începuturile listelor care conțin pozițiile;
  - b) Prin intermediul unei **tabele de dispersie** deschise.
- În ambele situații Poziții se poate determina în medie cu un efort de calcul de ordinul  $O(n)$  **pași**, unde prin "pas" se înțelege timpul necesar prelucrării unui simbol (prin dispersie, asociere sau prin comparație cu un altul).
  - Acest timp poate fi o **constantă** dacă simbolurile sunt caractere sau întregi dar tot atât de bine în situația în care simbolurile lui A și B sunt **linii de text**, atunci timpul necesar prelucrării unui simbol depinde de lungimea medie a liniilor textului.
- (2) **Pasul 2.** După ce s-au determinat mulțimile de indici ale structurii Poziții, pe rând, pentru fiecare simbol (element) care apare în secvența A, se poate trece la determinarea LCS în raport cu secvența B.
- În cele ce urmează se va preciza modul în care se găsește **lungimea lui LCS**, aflarea secvenței propriu-zise fiind recomandată ca exercițiu.
  - Se consideră secvența curentă A ca fiind delimitată de elementele  $a_1 \dots a_i$ .
  - Pasul 2 al algoritmului ia în considerare secvențe  $b_1 \dots b_j$  ale secvenței B pentru  $j=1, 2, \dots, m$ .
  - Considerând o secvență  $b_1 \dots b_j$ , este necesar să fie determinată pentru fiecare indice  $i$  cuprins între 0 și  $n$ , LCS-ul pentru secvențele  $a_1 \dots a_i$  și  $b_1 \dots b_j$ .
  - Valorile indicilor  $i$  procesați se grupează în mulțimile  $M_k$ , pentru  $k=0, 1, 2, \dots, n$ , unde o mulțime  $M_k$  constă din toate valorile indicelui  $i$  pentru care LCS-ul secvențelor  $a_1 \dots a_i$  și  $b_1 \dots b_j$  are lungimea  $k$ .
  - Astfel o mulțime  $M_k$  va fi formată întotdeauna dintr-un set de indici întregi **consecutivi**, iar indicii mulțimii  $M_{k+1}$  sunt **mai mari** decât cei ai lui  $M_k$  pentru toți  $k$ .

- Spre exemplu, considerând secvențele din figura 9.7.1.a, în figura 9.7.2.a se prezintă modul de determinare al mulțimilor  $M_k$  pentru  $j=5$ .

- Idea de bază** este aceea conform căreia, se iau pe rând elementele secvenței A, începând cu secvența vidă și se adaugă pe rând elementul următor al secvenței A.
- În fiecare astfel de pas se determină numărul de coincidențe dintre secvența A și secvența B cu  $j=5$ .
- Pentru început se încearcă găsirea coincidențelor între 0 elemente ale secvenței A ( $i=0$ ) și cele 5 elemente b, d, c, a, b ale secvenței B.
  - Se găsește evident un LCS de lungime 0, ca atare indicele 0 este adăugat mulțimii  $M_0$  (fig.9.7.2.a.(a)).
- Considerând în continuare pe  $i=1$ , deci primul element al secvenței A și cele 5 elemente ale secvenței B se găsește un LCS de lungime 1 (fig.9.7.2.a.(b)), deci indicele 1 aparține mulțimii  $M_1$ .
- Trecând la elementul următor al lui A și luând în calcul primele două elemente ale lui A, în același context se găsește un LCS de lungime 2 (fig.9.7.2.a.(c)).
- Procedând în același mod se obțin în final următoarele mulțimi de indici:  $M_0=\{0\}$ ,  $M_1=\{1\}$ ,  $M_2=\{2,3\}$ ,  $M_3=\{4,5,6\}$ ,  $M_4=\{7\}$  (fig.9.7.2.a. (a)... (h)).

i, j	1 2 3 4 5
i=0	
j=5	b d c a b

(a)

i, j	1 2 3 4 5
i=1	a
j=5	b d c a b

(b)

i, j	1 2 3 4 5
i=2	a b
j=5	b d c a b

(c)

i, j	1 2 3 4 5
i=3	a b c
j=5	b d c a b

(d)

i, j	1 2 3 4 5
i=4	a b c b
j=5	b d c a b

(e)

i, j	1 2 3 4 5
i=5	a b c b d
j=5	b d c a b

(f)

i, j	1 2 3 4 5 6
i=6	a b c b d a
j=5	b d c a b

(g)

i, j	1 2 3 4 5 6 7
i=7	a b c b d a b
j=5	b d c a b

(h)

**Fig.9.7.2.a.** Exemplu de determinare a mulțimilor  $M_k$

- (3) În **pasul 3** al algoritmului:
  - Se presupune că s-au determinat mulțimile  $M_k$  pentru poziția  $j-1$  a celei de-a doua secvențe, cu alte cuvinte pentru o secvență  $B$  de lungime  $j-1$ .
  - Se investighează în continuare maniera în care se modifică conținutul mulțimilor  $M_k$  atunci când **se adaugă** secvenței  $B$  elementul situat pe poziția următoare  $j$ .
  - În acest scop se consideră **Pozitii $(b_j)$** , adică pozițiile în care elementul adăugat  $b_j$  apare în secvența  $A$ .
  - Pentru fiecare indice  $r$  aparținând lui **Pozitii $(b_j)$**  se verifică dacă se poate îmbunătăți lungimea vreunui LCS existent, adăugând potrivirea dintre  $a_r$  și  $b_j$  LCS-ului deja determinat pentru secvențele  $a_1 \dots a_{r-1}$  și  $b_1 \dots b_{j-1}$ .
  - Dacă atât  $r-1$  ( $r$ -ul anterior) cât și  $r$  ( $r$ -ul curent) aparțin unei aceleiasi mulțimi  $M_k$ , atunci toate elementele  $s \geq r$  din  $M_k$ , aparțin de fapt mulțimii  $M_{k+1}$  pentru  $b_j$  considerat.
    - Acest lucru este valabil pe baza observației că celor  $k$  potriviri existente între elementele secvențelor  $a_1 \dots a_{r-1}$  și  $b_1 \dots b_{j-1}$  li se mai adaugă o potrivire și anume cea dintre  $a_r$  și  $b_j$ .
- **În consecință**, trecând de la elementul  $b_{j-1}$  la elementul  $b_j$ , adică adăugând secvenței  $B$  elementul următor, mulțimile  $M_k$  și  $M_{k+1}$  se pot modifica pe baza următorului algoritm:
  - (1) Se determină mulțimea de indicii  $r$  aparținând lui **Pozitii $(b_j)$** . Pentru fiecare indice  $r$ :
    - (2) Se execută operatorul **Caută $(r)$**  pentru a determina mulțimea  $M_k$  căreia îi aparține  $r$ .
    - (3) Se execută operatorul **Caută $(r-1)$** .
      - Dacă **Caută $(r-1)$**  nu este mulțimea  $M_k$ , potrivirea dintre  $b_j$  și  $a_r$  nu conduce la nici un beneficiu. În consecință se sar pașii 4 și 5 ai algoritmului, iar  $M_{k+1}$  nu se modifică.
    - (4) Dacă **Caută $(r-1)$**  este  $M_k$ , se execută operatorul **Partitionare $(M_k, M_k, M'_k, r)$**  pentru a determina acele elemente ale lui  $M_k$  care sunt mai mari sau egale cu  $r$  și pentru a construi cu ele mulțimea  $M'_k$ .
    - (5) Se execută operatorul **Uniune $(M'_k, M_{k+1}, M_{k+1})$**  prin care se mută elementele din mulțime  $M'_k$  în mulțimea  $M_{k+1}$ .

(6) Ciclul se reia de la (2) pentru fiecare valoare a lui  $r$  din mulțimea Pozitii ( $b_j$ ).

- **OBS:** Se subliniază faptul că indicii  $r$  ai lui Pozitii ( $b_j$ ) trebuie procesați începând cu **cea mai mare valoare**, în **ordine descrescătoare**.

- Pentru a motiva acest lucru se consideră spre exemplu situația în care elementele 7 și 9 aparțin mulțimii Pozitii ( $b_j$ ) și înaintea extinderii secvenței  $B$  cu elementul  $b_j$ , mulțimea  $M_3 = \{6, 7, 8, 9\}$  și mulțimea  $M_4 = \{10, 11\}$ .
- Dacă se tratează elementele 7 și 9 în această ordine, (Cazul 1, fig.9.7.2.b.(a)) procesând mai întâi elementul 7, se partionează  $M_3$  în  $M'_3 = \{6\}$  și  $M''_3 = \{7, 8, 9\}$  ceea ce conduce la  $M_4 = \{7, 8, 9, 10, 11\}$ .
- În continuare procesând elementul 9,  $M_4$  se partionează în  $M'_4 = \{7, 8\}$  și  $M''_4 = \{9, 10, 11\}$ , după care 9, 10 și 11 se unesc în  $M_5$ .
- Astfel în **mod paradoxal** indicele 9 a fost mutat din  $M_3$  în  $M_5$ , adăugând un singur element celei de-a doua secvențe, lucru care este imposibil.
- Eroarea provine din faptul că s-au considerat în mod eronat potrivirile dintre  $b_j$  și  $a_7$  respectiv dintre  $b_j$  și  $a_9$  în această ordine, creându-se astfel un LCS imaginari de lungime 5.
- Pentru a evita această situație verificările trebuie realizate considerând întâi potrivirea  $b_j$  și  $a_9$  și apoi potrivirea  $b_j$  și  $a_7$  (Cazul 2 fig.9.7.2.b.(b)).

$$M_3 = \{6, 7, 8, 9\}; M_4 = \{10, 11\}; \\ \text{Pozitii } (b_j) = \{7, 9\};$$

**Cazul 1.** Se prelucreză întâi 7 apoi 9

- I. Cauta (7) =  $M_3$   
Cauta (6) =  $M_3$
- II. Partitionare ( $M_3, M_3, M'_3, 7$ )  
 $\Rightarrow M_3 = \{6\}; M'_3 = \{7, 8, 9\};$
- III. Uniune ( $M'_3, M_4, M_4$ )  
 $\Rightarrow M_4 = \{7, 8, 9, 10, 11\};$   
 $r-1 \quad r$
- IV. Cauta (9) =  $M_4$   
Cauta (8) =  $M_4$
- V. Partitionare ( $M_4, M_4, M'_4, 9$ )  
 $\Rightarrow M_4 = \{7, 8\}; M'_4 = \{9, 10, 11\};$
- VI. Uniune ( $M'_4, M_5, M_5$ )  
 $\Rightarrow M_5 = \{9, 10, 11\};$   
**absurd**

$$M_3 = \{6, 7, 8, 9\}; M_4 = \{10, 11\}; \\ \text{Pozitii } (b_j) = \{7, 9\};$$

**Cazul 2.** Se prelucreză întâi 9 apoi 7

- I. Cauta (9) =  $M_3$   
Cauta (8) =  $M_3$
- II. Partitionare ( $M_3, M_3, M'_3, 9$ )  
 $\Rightarrow M_3 = \{6, 7, 8\}; M'_3 = \{9\};$
- III. Uniune ( $M'_3, M_4, M_4$ )  
 $\Rightarrow M_4 = \{9, 10, 11\}$
- IV. Cauta (7) =  $M_3$   
Cauta (6) =  $M_3$
- V. Partitionare ( $M_3, M_3, M'_3, 7$ )  
 $\Rightarrow M_3 = \{6\}; M'_3 = \{7, 8\};$
- VI. Uniune ( $M'_3, M_4, M_4$ )  
 $\Rightarrow M_4 = \{7, 8, 9, 10, 11\};$   
**corect**

(a)

(b)

**Fig.9.7.2.b.** Exemple de execuție incorectă (a), respectiv corectă (b) a algoritmului propus

- În secvența [9.7.2.a] apare o schiță a algoritmului care calculează și actualizează mulțimile  $M_k$  în timpul parcurgerii celei de-a doua secvențe.
  - Se presupune că structura **Pozitii** corespunzătoare tuturor elementelor secvenței  $A$  a fost construită în prealabil.
- Pentru a determina lungimea LCS, este suficient ca la sfârșitul prelucrării să se execute o operație **Cauta(n)**,  $n$  fiind indicele ultimului caracter al secvenței  $A$ .
- După cum s-a mai precizat, determinarea efectivă a LCS-ului este sugerată ca exercițiu.

---

/\*Determinarea lungimii LCS\*/

```

[1]  *initializează  $M_0 = \{0, 1, 2, \dots, n\}$ ;
[2]  *initializează  $M_i$  pe multimea vidă  $[i=1, n]$ ;
[3]  pentru ( $j=1$  la  $m$ ) /*determină  $M_k$  pentru poziția  $j$ /
[4]    pentru ( $r$  in Pozitii( $b_j$ ), începând cu cea mai mare
         valoare)
[5]      k=Cauta(r);
[6]      daca (k==Cauta(r-1))
[7]        /*r nu este cel mai mic din  $M_k$ */
[8]        Partitionare( $M_k, M_k, M_k', r$ );
        Uniune( $M_k', M_{k+1}, M_{k+1}$ );
        □ /*daca*/
        □ /*pentru*/
/*LCS*/

```

---

### 9.7.3. Analiza performanței algoritmului LCS

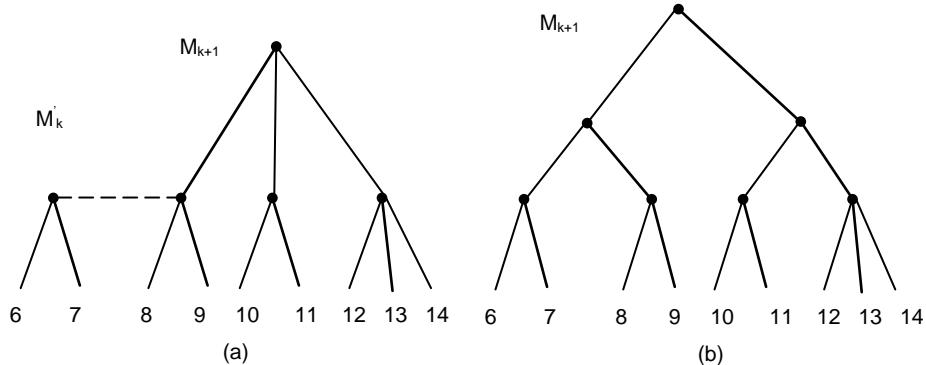
- După cum s-a precizat, algoritmul din secvența [9.7.2.a] este performant și util dacă **nu** există prea multe potriviri între cele două secvențe.
- Măsura **numărului de potriviri** este  $p$  cu mențiunea că structura **Pozitii** este calculată pentru secvența  $A$  ([9.7.3.a]).

---


$$p = \sum_{j=1}^m \text{card}(\text{Pozitii}(b_j)) \quad [9.7.3.a]$$

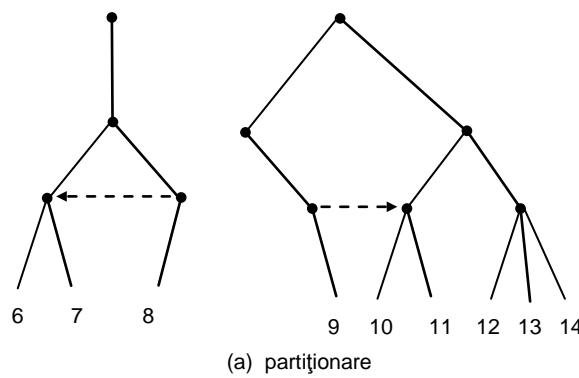

---

- În formula [9.7.3.a]  $\text{card}(\text{POZITII}(b_j))$  este cardinalitatea mulțimii  $\text{Pozitii}(b_j)$  iar  $m$  este lungimea secvenței  $B$ .
  - Cu alte cuvinte  $p$  este **suma numărului pozitilor** din secvența  $A$  care se potrivesc cu  $b_j$ , pentru toți  $j=1, m$ .
  - Este de așteptat ca valoarea medie a lui  $p$  să fie de același ordin cu  $m$  și  $n$  care sunt lungimile celor două secvențe (fișiere).
- O structură de date eficientă pentru implementarea mulțimilor  $M_k$  este **arborele 2-3** (&8.9.4).
  - Utilizând o astfel de structură, **initializarea** mulțimilor  $M_k$  (secvența [9.7.2.a], liniile [1] și [2]) necesită  $O(n)$  pași.
  - Operația **Caută** necesită un **tablou suplimentar** care păstrează **asocierea** dintre elemente și nodurile terminale corespunzătoare lor, precum și completarea structurii arbore 2-3 cu **pointeri** de tipul "indicator spre părinte".
  - **Numele mulțimii** ( $k$  pentru  $M_k$ ) poate fi păstrat în rădăcină, astfel încât operația **Caută** se execută urmând drumul de la nod spre rădăcină în  $O(\log_2 n)$  pași.
  - În consecință, toate execuțiile liniilor [5] și [6] din cadrul algoritmului necesită un efort de ordinul  $O(p \log_2 n)$ , deoarece fiecare linie este executată exact odată pentru fiecare potrivire.
- Operatorul **Uniune** din linia [8] are proprietatea specială că fiecare membru al lui  $M'_{k'}$  este mai mic decât orice membru al lui  $M_{k+1}$ , proprietate avantajoasă pentru implementarea bazată pe arbori 2-3.
  - Operația **Uniune** începe prin plasarea arborelui corespunzător mulțimii  $M'_{k'}$  la stânga celui corespunzător mulțimii  $M_{k+1}$ . Pot apărea trei situații:
    - (1) Dacă ambii arbori sunt de aceeași înălțime, se creează o nouă rădăcină care are drept fiu rădăcinile celor doi arbori.
    - (2) Dacă  $M'_{k'}$  este mai scurt, se inserează rădăcina acestui arbore ca și cel mai din stânga fiu al celui mai din stânga nod al lui  $M_{k+1}$  situat la nivelul corespunzător. Dacă acest nod are patru fiu se procedează la restructurare ca și în cazul inserției în arbori 2-3 (fig.9.7.3.a (a), (b)).
    - (3) Dacă  $M_{k+1}$  este mai scurt, rădăcina lui se face cel mai din dreapta fiu al celui mai din dreapta nod al lui  $M'_{k'}$  situat la nivelul corespunzător după care, dacă este necesar se restructurează și se redenumește arboarele.

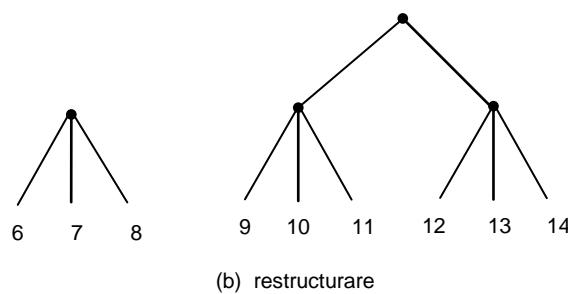


**Fig.9.7.3.a.** Exemplu de execuție a operatorului **Unione**

- Operatorul **Partitionare** pentru  $x$ , (linia [7]) necesită parcurgerea drumului de la nodul terminal  $x$  spre rădăcină, duplicând fiecare nod interior situat de-a lungul drumului și atașând câte o copie a fiecărui nod celor doi arbori rezultați.
  - Nodurile fără urmași sunt eliminate.
  - Nodurile cu un singur fiu sunt suprimate, nodul fiu fiind inserat după caz în arborele respectiv la nivelul corespunzător.
  - Dacă este necesar se procedează la restructurarea arborelui.
  - Această situație se poate urmări în figura 9.7.3.b, unde s-a realizat partiționarea în raport cu valoarea 9 a arborelui din figura 9.7.3.a (b).



(a) partiționare



(b) restructurare

**Fig.9.7.3.b.** Exemplu de execuție a operatorului **Partitionare**

- Analizând un număr mare de cazuri, s-a observat că partitōnarea și reorganizarea arborilor 2-3 de “jos în sus” necesită un efort de calcul de ordinul  $O(\log_2 n)$ .
  - Astfel, timpul total necesar execuției liniilor [7] și [8] din secvența [9.7.2.a] este proporțional cu  $O(p \log_2 n)$  și în consecință întregul algoritm necesită în medie  $O(p \log_2 n)$  pași.
- Mai trebuie adăugat timpul necesar calculului operatorului **Pozitii(a)**.
  - După cum s-a mai precizat, dacă elementele lui  $a$  sunt de mari dimensiuni, acest calcul poate dura mai mult decât oricare altă parte a algoritmului.
  - Dacă elementele pot fi comparate într-un singur pas, atunci sortarea sirului  $a_1, a_2, \dots, a_n$ , de fapt sortarea obiectelor  $(i, a_i)$  după câmpul  $a_i$ , se poate realiza cu un efort proporțional cu  $O(n \log_2 n)$ , după care **Pozitii(a)** poate fi completat pornind de la lista sortată, în  $O(n)$  pași.
- Astfel lungimea secvenței LCS poate fi calculată cu un efort de ordinul  $O((\max(n,p)) \log_2 n)$  și întrucăt de regulă  $p \geq n$ , rezultă  $O(p \log_2 n)$ .

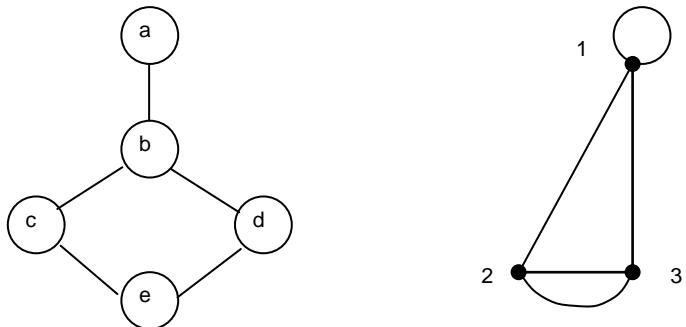
## 10. Structura de date graf

- În problemele care apar în programare, matematică, inginerie în general și în multe alte domenii, apare adeseori necesitatea reprezentării unor **relații arbitrară** între diferite obiecte, respectiv a **interconexiunilor** dintre acestea.
- Spre exemplu, dându-se **traseele aeriene ale unui stat** se cere să se precizeze drumul optim dintre două orașe.
  - Criteriul de optimalitate poate fi spre exemplu timpul sau prețul, drumul optim putând să difere pentru cele două situații.
- **Circuitele electrice** sunt alte exemple evidente în care interconexiunile dintre obiecte joacă un rol central.
  - Piese (tranzistoare, rezistențe, condensatoare) sunt interconectate prin fire electrice.
  - Astfel de circuite pot fi reprezentate și prelucrate de către un sistem de calcul în scopul rezolvării unor probleme simple cum ar fi: “*Sunt toate piesele date conectate în același circuit?*” sau a unor probleme mai complicate cum ar fi: “*Este funcțional un anumit circuit electric?*”.
- Un al treilea exemplu îl reprezintă **planificarea activităților**, în care obiectele sunt task-uri (activități, procese) iar interconexiunile precizează care dintre activități trebuie finalizate înaintea altora.
  - Întrebarea la care trebuie să ofere un răspuns este: “*Când trebuie planificată fiecare activitate?*”.
- **Structurile de date** care pot modela în mod natural situații de natură celor mai sus prezentate sunt cele derivate din **conceptul matematic** cunoscut sub denumirea de **graf**.
- **Teoria grafurilor** este o ramură majoră a matematicii combinatorii care în timp a fost și este încă intens studiată.
  - Multe din proprietățile importante și utile ale grafurilor au fost demonstreate, altele cu un grad sporit de dificultate își așteaptă încă rezolvarea.
- În cadrul capitolului de față vor fi prezentate doar câteva din proprietățile fundamentale ale grafurilor în scopul înțelegерii algoritmilor fundamentali de prelucrare a structurilor de date graf.
  - Ca și în multe alte domenii, studiul algoritmic al grafurilor respectiv al structurilor de date graf, este de dată relativ recentă astfel încât alături de algoritmii fundamentali cunoscuți de mai multă vreme, mulți dintre algoritmii de mare interes au fost descoperiți în ultimii ani [Se88].

### 10.1. Definiții

- Un graf, în cea mai largă acceptiune a termenului, poate fi definit ca fiind o colecție de **noduri și arce**.

- Un **nod** este un **obiect** care poate avea un nume și eventual alte proprietăți asociate.
- Un **arc** este o **conexiune** neorientată între două noduri.
- Notând cu  $N$  mulțimea nodurilor și cu  $A$  mulțimea arcelor, un graf  $G$  poate fi precizat formal prin enunțul  $G = (N, A)$ .
  - În figura 10.1.a. (a),(b) apar două exemple de grafuri.



**Fig.10.1.a.** Exemple de grafuri

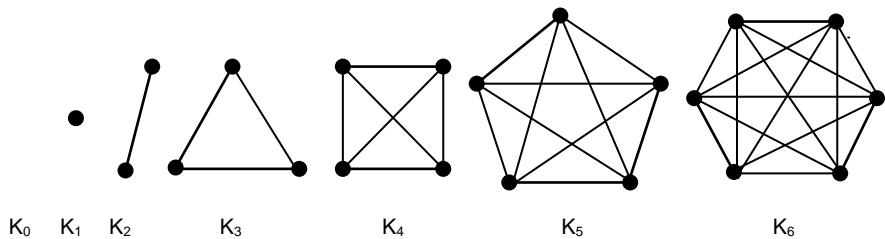
- **Ordinul** unui graf este numărul de noduri pe care acesta le conține și se notează cu  $|G|$ .
- **Arcele** definesc o **relație de incidentă** între perechile de noduri.
- Două noduri conectate printr-un arc se numesc **adiacente**, altfel ele sunt **independente**.
- Dacă  $a$  este un **arc** care leagă nodurile  $x$  și  $y$  ( $a \sim (x, y)$ ), atunci  $a$  este **incident** cu  $x, y$ .
  - Singura proprietate presupusă pentru această relație este **simetria** [10.1.a].

---

$(x, y) \in A \Rightarrow (y, x) \in A$  unde  $A$  este mulțimea arcelor. [10.1.a]

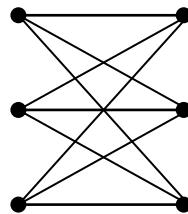
---

- Un graf poate fi definit astfel încât să aibă un arc  $a \sim (x, x)$ .
  - Un astfel de arc se numește **bucă** ("loop").
  - Dacă relația de incidentă este **reflexivă**, atunci fiecare nod conține o astfel de buclă.
- Există și posibilitatea ca să existe mai multe arce care conectează aceeași pereche de noduri. Într-un astfel de caz se spune că cele două noduri sunt conectate printr-un **arc multiplu**.
  - În figura 10.1.a (b) este reprezentat un graf cu buclă și arc multiplu.
- Grafurile în care nu sunt acceptate arce multiple se numesc **grafuri simple**.
- Numărul de arce incidente unui nod reprezintă **gradul** nodului respectiv.
- Se numește **graf regulat** acel graf în care toate nodurile sunt de același grad.
- Se numește **graf complet de ordinul**  $n$  și se notează cu  $K_n$ , acel graf în care fiecare pereche de noduri este adiacentă.
  - În figura 10.1.b. apar reprezentate grafurile complete până la ordinul 6.



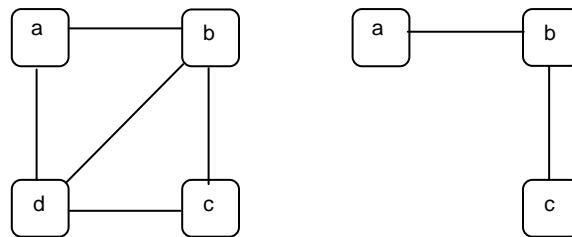
**Fig.10.1.b.** Exemple de grafuri complete

- Un **graf** se numește **planar** dacă el poate fi astfel reprezentat într-un plan, încât oricare două arce ale sale se intersectează numai în noduri [GG78].
- O teoremă demonstrată de Kuratowski precizează că orice **graf neplanar** conține cel puțin unul din următoarele grafuri de bază:
  - (1) **5-graful complet** ( $K_5$ ), sau
  - (2) **Graful utilitar** în care există două mulțimi de câte trei noduri, fiecare nod fiind conectat cu toate nodurile din cealaltă mulțime.



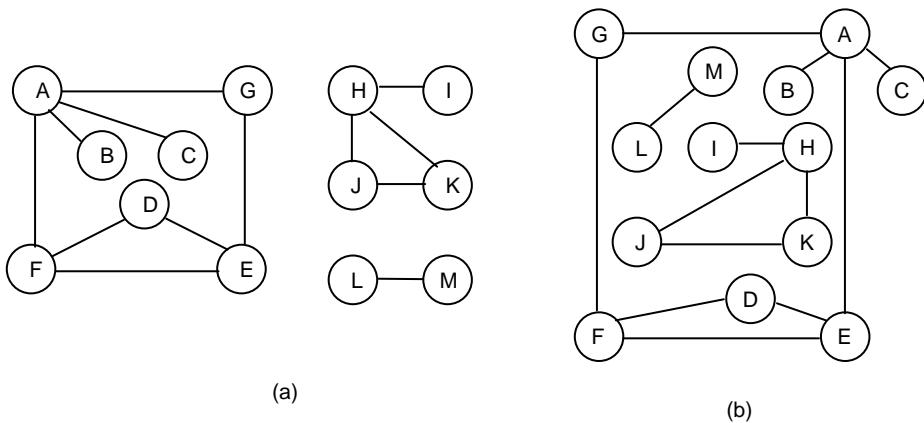
**Fig.10.1.c.** Graf utilitar

- Un **graf** se numește **bipartit** dacă nodurile sale pot fi partaționate în două mulțimi distințe  $N_1$  și  $N_2$  astfel încât orice arc al său conectează un nod din  $N_1$  cu un nod din  $N_2$ .
  - Spre exemplu **graful utilitar** este în același timp un **graf bipartit**.
- Fie  $G = (N, A)$  un graf cu mulțimea nodurilor  $N$  și cu mulțimea arcelor  $A$ . Un **subgraf** al lui  $G$  este **graful  $G' = (N', A')$**  unde:
  - (1)  $N'$  este o submulțime a lui  $N$ .
  - (2)  $A'$  constă din arce  $(x, y)$  ale lui  $A$ , astfel încât  $x$  și  $y$  aparțin lui  $N'$ .
- În figura 10.1.d apare un exemplu de **graf** (a) și un **subgraf** al său (b).



**Fig.10.1 d.** Graf și un subgraf al său

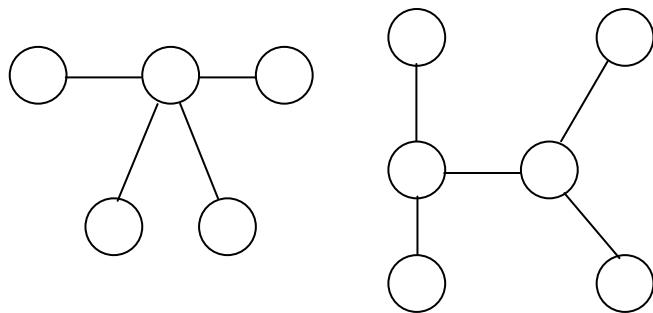
- Dacă mulțimea de arce  $A'$  conține **toate arcele**  $(x, y)$  ale lui  $A$  pentru care atât  $x$  cât și  $y$  sunt în  $N'$ , atunci  $G'$  se numește **subgraf induș** al lui  $G$  [AH85].
  - Un **graf** poate fi **reprezentat** în **manieră grafică** marcând nodurile sale și trasând linii care materializează arcele.
    - Reprezentarea unui graf nu este unică. Spre exemplu fig.10.1.e (a) respectiv (b) reprezintă unul și același graf.
  - În același timp însă, un **graf** poate fi conceput ca și un **tip de date abstract**, independent de o anumită reprezentare.
    - Un graf, poate fi definit spre exemplu precizând doar **mulțimea nodurilor** și **mulțimea arcelor** sale.



**Fig.10.1.e.** Reprezentări echivalente ale unui graf

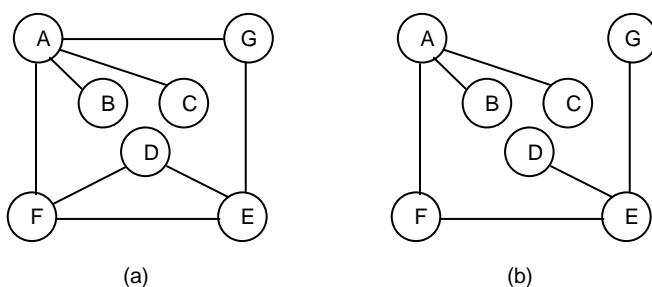
- În anumite aplicații, cum ar fi exemplul cu traseele aeriene, poziția nodurilor (orașelor) este precizată fizic prin amplasarea lor pe harta reală a statului, rearanjarea structurii fiind lipsită de sens.
  - În alte aplicații însă, cum ar fi planificarea activităților, sunt importante nodurile și arcele ca atare independent de dispunerea lor geometrică.
    - În cadrul capitolului de față vor fi abordați algoritmi generali, care prelucrează colecții de noduri și arce, făcând abstracție de dispunerea lor geometrică, cu alte cuvinte făcând abstracție de **topologia** grafurilor.
  - Se numește **drum** (“path”) de la nodul  $x$  la nodul  $y$ , aparținând unui graf, o **secvență de noduri**  $n_1, n_2, \dots, n_j$  în care nodurile succesive sunt conectate prin arce aparținând grafului.
    - **Lungimea** unui drum este egală cu numărul de arce care compun drumul.
    - La limită, un singur nod precizează un drum la el însuși de lungime zero.
  - Un **drum** se numește **simplu** dacă toate nodurile sale, exceptând eventual primul și ultimul sunt distințe.
  - Un **ciclu (buclă)** este un drum simplu de lungime cel puțin 1, care începe și se sfârșește în același nod.
  - Dacă există un drum de la nodul  $x$  la nodul  $y$  se spune că acel drum **conectează** cele două noduri, respectiv nodurile  $x$  și  $y$  sunt **conectate**.
  - Un **graf** se numește **conex**, dacă de la fiecare nod al său există un drum spre oricare alt nod al grafului, respectiv dacă oricare pereche de noduri aparținând grafului este conectată.

- Intuitiv, dacă nodurile se consideră obiecte fizice, iar conexiunile fire care le leagă, atunci un **graf conex** rămâne unitar, indiferent de care nod ar fi “suspendat în aer”.
- Un graf care nu este conex este format din **componente conexe**.
  - Spre exemplu, graful din fig.10.1.a este format din trei componente conexe.
- O **componentă conexă** a unui graf G este de fapt un **subgraf induș maximal conectat** al său [AH 85].
- Un **graf** se numește **ciclic** dacă conține cel puțin un ciclu.
  - Un **ciclu** care include **toate arcele grafului** o singură dată se numește **ciclu eulerian (hamiltonian)**.
  - Este ușor de observat că un asemenea ciclu există numai dacă graful este conex și gradul fiecărui nod este par.
- Un **graf conex aciclic** se mai numește și **arbore liber**.
  - În fig.10.1.f apare un graf constând din două componente conexe în care fiecare componentă conexă este un arbore liber.
  - Se remarcă în acest sens, observația ca **arborii** sunt de fapt cazuri particulare ale **grafurilor**.
  - Un grup de arbori neconectați formează o **pădure** (“**forest**”).



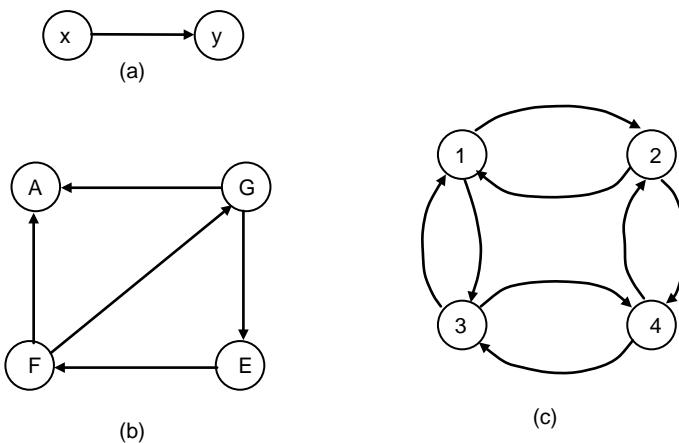
**Fig.10.1.f.** Graf aciclic format din două componente conexe

- Un **arbore de acoperire** (“**spanning tree**”) al unui graf, este un **subgraf** care conține toate nodurile grafului inițial, dar dintre conexiuni numai atâtea câte sunt necesare formării unui arbore.
  - Se face precizarea că termenul de “**acoperire**” în acest context are sensul termenului “**cuprindere**”.
- În figura 10.1.g este prezentat un graf (a) și un arbore de acoperire al grafului (b).



**Fig.10.1.g.** Graf și un arbore de acoperire al grafului

- Un **arbore liber** poate fi transformat într-un **arbore ordinar** dacă se “suspendă” arboarele de un nod considerat drept **rădăcină** și se orientează arcele spre rădăcină.
- **Arborii liberi** au două proprietăți importante:
  - (1) Orice **arbore liber** cu  $n$  noduri conține exact  $n-1$  arce (câte un arc la fiecare nod, mai puțin rădăcina).
  - (2) Dacă unui **arbore liber** i se adaugă un arc el devine obligatoriu un **graf ciclic**.
- De aici rezultă două **consecințe** importante și anume:
  - (1) Un graf cu  $n$  noduri și mai puțin de  $n-1$  arce nu poate fi conex.
  - (2) Pot exista grafuri cu  $n$  noduri și  $n-1$  arce care nu sunt arbori liberi. (Spre exemplu dacă au mai multe componente conexe).
- Notând cu  $n$  numărul de **noduri** ale unui graf și cu  $a$  numărul de **arce**, atunci  $a$  poate lua orice valoare între 0 și  $(1/2)n(n-1)$ .
  - Graful care conține toate arcele posibile este **graful complet** de ordinul  $n$  ( $K_n$ ).
  - Graful care are relativ puține arce (spre exemplu  $a < n \log_2 n$ ) se numește **graf rar** (“sparse”).
  - Graful cu un număr de arce apropiat de graful complet se numește **graf dens**.
- Dependența fundamentală a **topologiei unui graf** de doi parametri ( $n$  și  $a$ ), face ca studiul comparativ al algoritmilor utilizati în prelucrarea grafurilor să devină mai complicat din cauza posibilităților multiple care pot să apară.
  - Astfel, presupunând că un algoritm de prelucrare a unui graf necesită un efort de calcul de ordinul  $O(n^2)$  în timp ce un alt algoritm care rezolvă aceeași problemă necesită un efort de ordinul  $O((n+a)\log_2 n)$  pași, atunci, în cazul unui graf cu  $n$  noduri și  $a$  arce, este de preferat primul algoritm dacă graful este dens, respectiv al doilea dacă graful este rar.
- Grafurile prezentate până în prezent se numesc și **grafuri neorientate** și ele reprezintă cea mai simplă categorie de grafuri.
- Prin asocierea de informații suplimentare nodurilor și arcelor, se pot obține categorii de grafuri mai complicate.
- Astfel, într-un **graf ponderat** (“**weighted graph**”), fiecărui arc i se asociază o valoare (de regulă pozitivă) numită **pondere** care poate reprezenta spre exemplu o distanță sau un cost.
- În cadrul **grafurilor orientate** (“**directed graphs**”), arcele sunt orientate, având un sens precizat, de la  $x$  la  $y$  spre exemplu.
  - În acest caz  $x$  se numește coada sau sursa arcului iar  $y$  vârful sau destinația sa.
  - Pentru reprezentarea arcelor orientate se utilizează săgeți sau segmente direcționate (fig.10.1.h. (a),(b),(c)).
- **Grafurile orientate ponderate** se mai numesc și **rețele** (“**networks**”).
- Informațiile suplimentare referitoare la noduri și arce nuanțează și în același timp complică manipularea grafurilor care le conțin.



**Fig.10.1.h.** Grafuri orientate

## 10.2. Tipul de date abstract graf

- Pentru a defini **tipul de date abstract** (TDA) **graf** este necesară:
  - (1) Precizarea **modelului matematic** care conturează conceptul de graf prezentat în paragraful anterior.
  - (2) Precizarea **setului de operatori** definiți pe acest model.
- În cele ce urmează se prezintă două variante de definire a unui **TDA graf**, una extinsă și alta mai restrânsă.

### 10.2.1. TDA graf. Varianta 1 (Shiflet)

- În cadrul variantei Shiflet [Sh90] un **graf** este considerat ca și o structură de noduri și arce.
  - Fiecare nod are o cheie care identifică în mod univoc nodul.
  - Modelul matematic, notațiile utilizate și setul de operatori preconizat pentru această variantă apar în [10.2.1.a]:

---

**TDA Graf**      (Varianta 1 - Shiflet)      [10.2.1.a]

**Modelul matematic:** graful definit în sens matematic.

**Notății:**

<i>TipGraf</i>	- tipul de date abstract <b>graf</b> ;
<i>TipElement</i>	- tipul asociat portiunii element a nodului
<i>TipCheie</i>	- tipul asociat portiunii cheie a unui element
<i>TipInfo</i>	- tipul corespunzător portiunii de informație a unui element
<i>TipIndicNod</i>	- tip referință la structura unui nod
<i>TipIndicArc</i>	- tip referință la structura unui arc

**Operatori:**

1. ***InitGraf***(*TipGraf g*) - procedură care creează graful vid *g*;
2. *boolean* ***GrafVid***(*TipGraf g*) - operator care returnează **true** dacă graful este vid respectiv **false** în caz contrar;
3. *boolean* ***GrafPlin***(*TipGraf g*) - operator boolean care returnează **true** dacă graful este plin. Se precizează faptul că această funcție este legată direct de maniera de implementare a grafului. Valoarea ei adevărată presupune faptul că zona de memorie alocată structurii a fost epuizată și în consecință nu se mai pot adăuga noi noduri.
4. *TipCheie* ***CheieElemGraf***(*TipGraf g, TipElement e*) - operator care returnează cheia elementului *e* aparținând grafului *g*.
5. *boolean* ***CautaCheieGraf***(*TipGraf g, TipCheie k*) - operator boolean care returnează valoarea adevărat dacă cheia *k* este găsită în graful *g*.
6. ***IndicaNod***(*TipGraf g, TipCheie k, TipIndicNod\* indicNod*) - operator care face ca *IndicNod* să indice acel nod din *g* care are cheia *k*, presupunând că un astfel de nod există.
7. ***IndicaArc***(*TipGraf g, TipCheie k1, TipCheie k2, TipIndicArc\* indicArc*) - operator care face ca *indicArc* să indice arcul care conectează nodurile cu cheile *k1* și *k2* din graful *g*, presupunând că arcul există. În caz contrar *indicArc* ia valoarea indicatorului vid.
8. *boolean* ***ArcVid***(*TipGraf g, TipIndicArc indicArc*) - operator boolean care returnează valoarea adevărat dacă arcul indicat de *indicArc* este vid.
9. ***InserNod***(*TipGraf\* g, TipElement e*) - operator care înserează un nod *e* în graful *g* ca un nod izolat (fără conexiuni). Se presupune că înaintea inserției în *g* nu există nici un nod care are cheia identică cu cheia lui *e*.
10. ***InserArc***(*TipGraf\* g, TipCheie k1, TipCheie k2*) - operator care înserează în *g* un arc incident nodurilor având cheile *k1* și *k2*. Se presupune că cele două noduri există și că arcul respectiv nu există înaintea inserției.
11. ***SuprimNod***(*TipGraf\* g, TipIndicNod indicNod*) - operator care suprimă din *g* nodul precizat de *indicNod*, împreună cu toate arcele incidente. Se presupune că înaintea suprimării, un astfel de nod există.
12. ***SuprimArc***(*TipGraf\* g, TipIndicArc indicArc*) - operator care suprimă din *g*, arcul precizat de *indicArc*. Se

presupune că înaintea suprimării un astfel de arc există.

13. **ActualizNod**(*TipGraf\** *g*, *TipIndicNod* *indicNod*, *TipInfo* *x*) - operator care plasează valoarea lui *x* în portiunea "informație" a nodului indicat de *indicNod* din graful *g*. Se presupune că *indicNod* precizează un nod al grafului.
14. *TipElement* **FurnizeazaNod**(*TipGraf* *g*, *TipIndicNod* *indicNod*) - operator care returnează valoarea elementului memorat în nodul indicat de *indicNod* în graful *g*.
15. **TraversGraf**(*TipGraf* *g*, *Vizită* (*ListăArgumente*)) - operator care realizează traversarea grafului *g*, executând pentru fiecare element al acestuia procedura *Vizită*(*ListăArgumente*), unde *Vizită* este o procedură specificată de utilizator iar *ListăArgumente* este lista de parametri a acesteia.
- 

### 10.2.2. TDA graf. Varianta 2 (Decker)

- În viziunea lui Rick Decker [De89], **tipul de date abstract graf** constă:
    - (1) Dintr-o mulțime **Pozitii** care materializează mulțimea nodurilor grafului.
    - (2) O mulțime **Atomii**, care materializează conținuturile nodurilor grafului.
    - (3) O relație **R** astfel încât  $pRq$  (unde  $p, q$  aparțin mulțimii **Pozitii**) precizează existența unui arc care conectează nodurile  $p$  și  $q$ .
  - Drept urmare un **graf** poate fi definit drept o **tripletă**  $(P, R, f)$  unde:
    - $P$  este o **submulțime** finită a mulțimii **Pozitie**.
    - $f$  este o **funcție** definită pe  $P$  cu valori în mulțimea **Atomii**.
    - $R$  este **relația simetrică** în  $P$  (și ireflexivă dacă nu se permit bucle), definită anterior.
  - De regulă mulțimea pozițiilor (nodurilor) unui graf  $g$  se precizează prin notația  $N(g)$ .
  - Deși nu este absolut necesar, se poate defini mulțimea **Arce** sau  $A(g)$  ca fiind mulțimea tuturor mulțimilor formate din două elemente  $\{p, q\}$  pentru care  $pRq$ .
  - În acest context, **TDA graf** este definit după cum urmează [10.2.2.a].
- 

**TDA Graf (Varianta 2 - Decker)**

[10.2.2.a]

**Modelul matematic:** graful definit în sensul precizat în paragraful &10.2.2.

**Notatii:**

- TipGraf* - tipul de date abstract graf  
*TipPozitie* - tipul asociat nodurilor grafului  
*TipAtom* - tipul asociat portiunii de informații a

**Operatori:**

1. ***Crează***(*TipGraf g*) - operator care crează graful vid *g*.
  2. ***boolean Adiacent***(*TipPozitie p, TipPozitie q, TipGraf g*)  
- operator care returnează valoarea **true** dacă și numai dacă  $pRq$ , adică dacă în graf există un arc de la nodul *p* la nodul *q*.
  3. ***Modifică***(*TipAtom a, TipPozitie p, TipGraf g*) - modifică atomul asociat poziției *p* făcând  $f(p)=a$ . Cu alte cuvinte, zonei de date a nodului indicat de  $p \in N(g)$  i se conferă valoarea *a*.
  4. ***TipAtom Furnizează***(*TipPozitie p, TipGraf g*) - operator care returnează valoarea  $f(p)$ , cu alte cuvinte zona de date a nodului indicat de  $p \in N(g)$ .
  5. ***SuprimNod***(*TipPozitie p, TipGraf g*) - operator care suprimă nodul indicat de *p* din *g* împreună cu toate arcele incidente. Se consideră valabilă precondiția  $p \in N(g)$ .
  6. ***SuprimArc***(*TipArc e, TipGraf g*) - operator care extrage pe *e* din mulțimea arcelor lui *g*. Se consideră valabilă precondiția  $e \in A(g)$ .
  7. ***InserNod***(*TipPozitie p, TipGraf g*) - operator care adaugă pe *p* mulțimii pozițiilor lui *g* fără a-l modifica pe *R*. Cu alte cuvinte, nodul indicat de *p* se adaugă la *g*, fără nici un arc de conexiune cu un alt nod al grafului. Se presupune valabilă precondiția  $p \notin N(g)$ .
  8. ***InserArc***(*TipArc e, TipGraf g*) - operator care include pe *e* în mulțimea arcelor lui *g* modificând relația structurală *R*. Această operație presupune inițial valabile următoarele afirmații:
    - a) dacă  $e=\{p,q\}$  atunci  $p,q \in N(g)$  și
    - b)  $e \notin A(g)$ .
- 

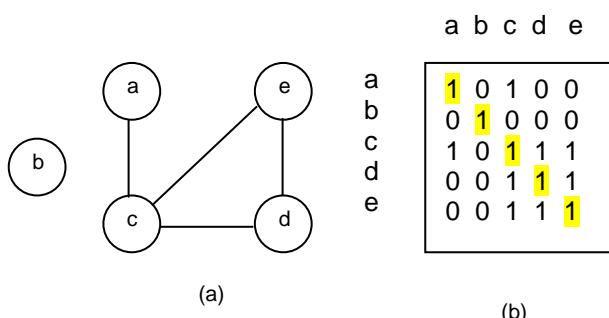
### 10.3. Tehnici de implementare a tipului de date abstract graf

- În vederea prelucrării grafurilor concepute ca și tipuri de date abstracte (TDA) cu ajutorul sistemelor de calcul, este necesară la primul rând stabilirea **modului lor de reprezentare**.
- Această activitate constă de fapt din desemnarea unei structuri de date concrete care să materializeze în situația respectivă tipul de date abstract graf.
- În cadrul paragrafului de față se prezintă mai multe posibilități, alegerea depinzând, ca și pentru marea majoritate a tipurilor de date deja studiate:
  - (1) De natura grafurilor de implementat.
  - (2) De natura și frecvența operațiilor care se execută asupra lor.

- În esență se cunosc **două modalități majore** de implementare a grafurilor: una bazată pe **matrici de adiacențe** iar celalată bazată pe **structuri de adiacențe**.

### 10.3.1. Implementarea grafurilor cu ajutorul matricilor de adiacențe

- Cel mai direct mod de reprezentare al unui tip de date abstract graf îl constituie **matricea de adiacențe** (“**adjacency matrix**”).
- Dacă se consideră graful  $G=(N,A)$  cu mulțimea nodurilor  $N=\{1, 2, \dots, n\}$ , atunci **matricea de adiacențe** asociată grafului  $G$ , este o matrice  $A[n,n]$  de elemente booleene, unde  $A[x,y]$  este adevărat dacă și numai dacă în graf există un arc de la nodul  $x$  la nodul  $y$ .
  - Adesea elementelor booleene ale matricii sunt înlocuite cu întregii 1 (adevărat), respectiv 0 (fals).
- Primul pas în reprezentarea unui graf printr-o matrice de adiacențe constă în stabilirea unei **corespondențe** între **numele nodurilor** și **mulțimea indicilor matricei**.
  - Această corespondență poate fi realizată:
    - (1) În **mod implicit** prin alegerea corespunzătoare a tipului de bază al mulțimii  $N$ .
    - (2) În **mod explicit** prin precizarea unei **asocieri definite** pe **mulțimea nodurilor** cu valori în **mulțimea indicilor matricei**.
- În cazul **corespondenței implicate** cel mai simplu mod de implementare constă în “a denumi” nodurile cu **numere întregi** care coincid cu indicii de acces în **matricea de adiacențe**.
  - Numele nodurilor pot fi de asemenea litere consecutive ale alfabetului sau în cazul limbajului Pascal, constante ale unui tip enumerare definit de utilizator, în ambele situații existând posibilitatea conversiei directe a tipurilor respective în tipul întreg prin funcții specifice de limbaj.
- În cazul **corespondenței explicite**, pentru implementarea asocierii pot fi utilizate:
  - Tehnici specifice simple cum ar fi cele bazate pe tablouri sau liste.
  - Tehnici mai avansate bazate spre exemplu pe arbori binari sau pe metoda dispersiei.
- Pentru o urmărire facilă a algoritmilor, în cadrul capitolului de față, se va utiliza o **metodă implicită** conform căreia nodurile vor avea numele format dintr-o singură literă.
- În figura 10.3.1.a. apare reprezentarea bazată pe matrice de adiacențe (b) a grafului (a).



### **Fig.10.3.1.a.** Graf (a) reprezentat prin matrice de adiacențe (b).

- Se observă faptul că reprezentarea are un **caracter simetric** întrucât fiind vorba despre un **graf neorientat**, arcul care conectează nodul  $x$  cu nodul  $y$  este reprezentat prin **două valori** în matrice:  $A[x, y]$  respectiv  $A[y, x]$ .
  - În astfel de situații, deoarece matricea de adiacențe este simetrică, ea poate fi memorată pe jumătate, element care pe lângă avantaje evidente are și dezavantaje.
    - Pe de-o parte **nu** toate limbajele de programare sunt propice unei astfel de implementări.
    - Pe de altă parte **algoritmii** care prelucrează astfel de matrici sunt mai complicați decât cei care prelucrează matrici integrale.
- În prelucrarea grafurilor se poate face presupunerea că un nod este conectat cu el însuși, element care se reflectă în valoarea “adevărat” memorată în toate elementele situate pe diagonala principală a matricei de adiacențe.
  - Acest lucru nu este însă obligatoriu și poate fi reconsiderat de la caz la caz.
- În continuare se prezintă două studii de caz pentru implementarea TDA graf cu ajutorul matricilor de adiacențe.

#### **10.3.1.1. Studiu de caz 1.**

- După cum s-a precizat, un graf este definit prin **două mulțimi: mulțimea nodurilor și mulțimea arcelor** sale.
- În vederea prelucrării, un astfel de graf trebuie furnizat drept dată de intrare algoritmului care realizează această activitate.
- În acest scop, este necesar a se preciza modul în care se vor introduce în memoria sistemului de calcul elementele celor două mulțimi.
- (1) O posibilitate în acest sens o reprezintă **citirea directă**, ca dată de intrare a **matricii de adiacențe**, metodă care nu convine în cazul matricilor rare.
- (2) O altă posibilitate o reprezintă următoarea:
  - În prima etapă se citesc **numele nodurilor** în vederea asocierii acestora cu indicii matricei de adiacențe.
  - În etapa următoare, se citesc **perechile de nume de noduri** care definesc **arce** în cadrul grafului.
    - Pornind de la aceste perechi se generează **matricea de adiacențe**.
  - Se face precizarea că prima etapă poate să lipsească dacă în implementarea asocierii se utilizează o **metodă implicită**.
- În secvența [10.3.1.1.a] apare un exemplu de program pentru **crearea** unei matrice de adiacențe.
  - Corespondența nume-nod-indice este realizată implicit prin funcția **index(n)**, care are drept parametru numele nodului și returnează indicele acestuia.
  - Din acest motiv, prima etapă se reduce la citirea valorilor  $N$  și  $A$  care reprezintă numărul de noduri, respectiv numărul de arce ale grafului.

- Ordinea în care se furnizează perechile de noduri în etapa a doua nu este relevantă, încărcând matricea de adiacențe nu este în nici un mod influențată de această ordine.

---

**/\*Cazul 1. Implementarea TDA Graf utilizând matrici de adiacențe - varianta pseudocod\*/**

```

int maxN = 50; /* N = numărul maxim de noduri*/
int j,x,y;
int N; /* N = numărul curent de noduri*/
int A; /* A = numărul curent de arce*/

Tip_Nod n1,n2;
boolean graf[maxN,maxN]; /* matricea de adiacențe */

*citeste(N,A); [10.3.1.1.a]
/*initializare matrice de adiacențe*/
pentru (x=1 la N)
  pentru (y=1 la N)
    graf[x,y]= false;
/*initializare diagonală principală a MA*/
pentru (x=1 la N)
  graf[x,x]= true;
/*construcție matrice de adiacențe pentru graf*/
pentru (j=1 la A)
  *citeste(n1,n2);
  x=index(n1); y=index(n2);
  graf[x,y]=true;
  graf[y,x]=true
  □ /*pentru*/
/*Creare matrice de adiacențe*/

```

---

- După cum se observă în cadrul secvenței, tipul variabilelor n1 și n2 este TipNod care nu este precizat.
- De asemenea nu este precizat nici codul aferent funcției **index**, acestea depinzând direct de maniera de reprezentare a nodurilor.
  - Spre exemplu n1 și n2 pot fi de tip caracter (int) iar funcția **index** o expresie de forma n1-`a`.

### 10.3.1.2. Studiu de caz 2

- Studiul de caz 2 prezintă o metodă mai elaborată de implementare a unui TDA graf.
- Reprezentarea presupune definirea tipurilor și structurilor de date în conformitate cu secvența [10.3.1.2.a].

---

**/\*Cazul 2. Implementarea TDA Graf utilizând matrici de adiacențe - varianta C\*/**

```

const Numar_Noduri = ....;
typedef Tip_Cheie ....;

```

```

typedef Tip_Info ....;

/*definire tip structură element (nod)*/
typedef struct Element
{
    Tip_Cheie Cheie;
    Tip_Info Info;
} Tip_Element;

/*definire tip tablou elemente*/
typedef Tip_Element Tip_Tablu_Elemente[Numar_Noduri];

/*definire tip matrice de adiacențe*/
typedef boolean Tip_Matr_Adj[Numar_Noduri,Numar_Noduri];

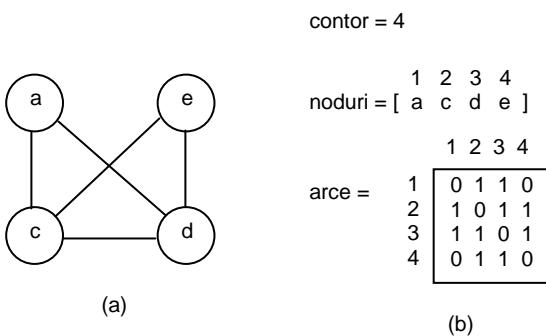
/*definire tip structură graf*/
typedef struct Graf
{
    int contor;                                     /*[10.3.1.2.a]*/
    Tip_Tablu_Elemente noduri;
    Tip_Matr_Adj arce;
} Tip_Graf

/*definire structură arc*/
typedef struct Arc
{
    int linie;
    int coloana;
} Tip_Arc

Tip_Graf g;
Tip_Cheie k,k1,k2;
Tip_Element e;
int indicNod;
Tip_Arc indicArc;
-----

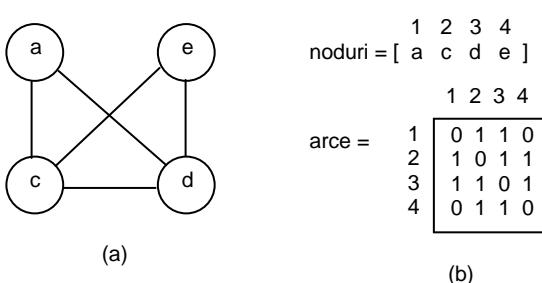
```

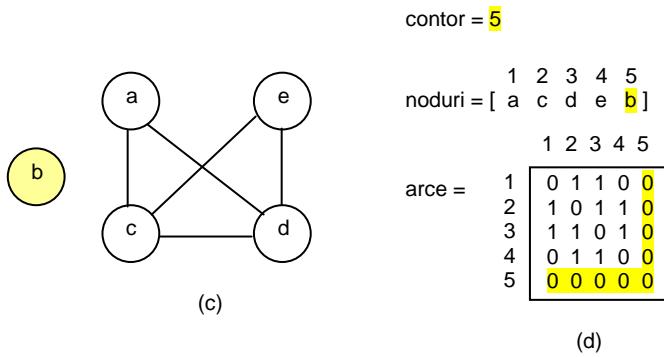
- În accepțiunea acestei reprezentări, **graful** din fig. 10.3.1.2.a.(a) va fi implementat prin următoarele elemente:
  - (1) contor – care precizează numărul de noduri.
  - (2) noduri – tabloul care păstrează nodurile propriu-zise.
  - (3) arce – matricea de adiacențe a grafului (fig.10.3.1.2.a.(b)).



**Fig.10.3.1.2.a.** Reprezentarea elaborată a unui graf utilizând matrice de adiacențe

- În anumite situații, pentru simplificare, nodurile nu conțin alte informații în afara cheii, caz în care *Tip\_Element* = *Tip\_Cheie*.
  - Alteori nodurile nu conțin nici un fel de informații (nici măcar cheia) situație în care interesează numai **numele nodurilor** în vederea identificării lor în cadrul reprezentării.
- În continuare se fac unele **considerații** referitoare la implementarea în acest context a setului de operatori extins (Varianta 1, (Shiflet)).
  - (1) Operatorii *InitGraf*, *GrafVid*, *GrafPlin*, împreună cu *InserNod* și *SuprimNod* se referă în regim de consultare sau modificare la contorul care păstrează **numărul nodurilor grafului**: *g*.*contor*.
  - (2) Informația conținută de tabloul noduri poate fi **ordonată** sau **neordonată**.
    - Dacă tabloul noduri este **ordonat**, localizarea unei chei în cadrul operatorilor *CautăCheieGraf* sau *IndicaNod* se poate realiza prin tehnica **căutării binare**.
    - Dacă tabloul noduri este **neordonat** localizarea unei chei se poate realiza prin tehnica **căutării liniare**.
    - Operatorul *CautăCheieGraf* indică numai dacă cheia este prezentă sau nu în graf.
    - Operatorul *IndicaNod*(*TipGraf* *g*, *TipCheie* *k*, *TipIndicNod* \* *indicNod*) asignează lui *IndicNod* indexul nodului din graful *g*, care are cheia egală cu *k*.
    - Operatorul *IndicaArc*(*TipGraf* *g*, *TipCheie* *k1*, *TipCheie* *k2*, *TipArc* \* *indicArc*) se comportă într-o manieră similară, returnând indexul nodului *k1* în variabila de ieșire *indicArc.linie* și indexul nodului *k2* în *indicArc.coloană*.
  - (3) **Inserția** unui nod depinde de asemenea de maniera de organizare a datelor în cadrul tabloului noduri.
    - (a) Dacă noduri este un **tablou neordonat**, se incrementează *g*.*contor* și se memorează nodul de inserat în *g*.*noduri*[*g*.*contor*].
      - După cum rezultă din fig.10.3.1.2.b, care reprezintă inserția nodului *b* în graful (a), nodul nou introdus este izolat, adică în matricea de adiacențe se introduce valoarea fals pe linia *g*.*contor* și pe coloana *g*.*contor* (d).





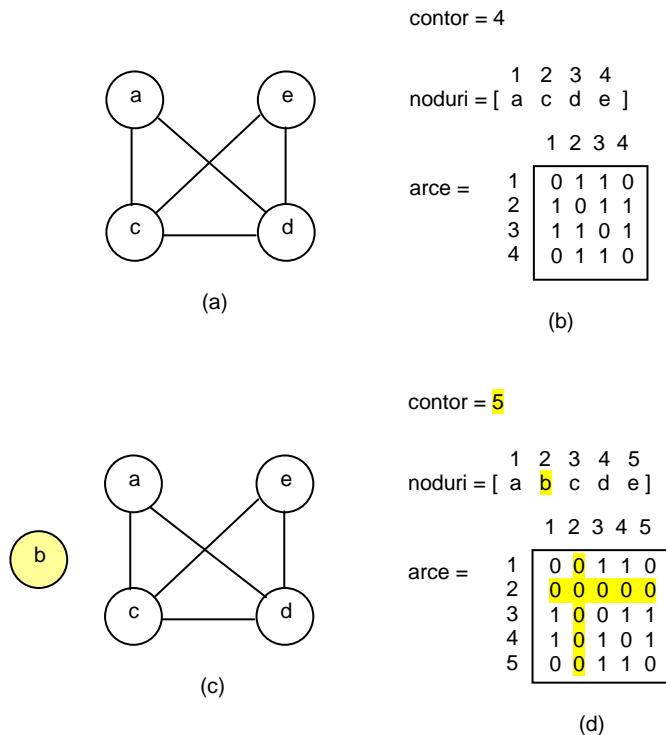
**Fig.10.3.1.2.b.** Inserția unui nod într-un graf (varianta tablou noduri neordonat)

- Procedura efectivă de inserție a unui nod nou în acest context apare în secvența [10.3.1.2.b].

```
/*Inserția unui nod. (Tabloul noduri neordonat) - varianta pseudocod*/
```

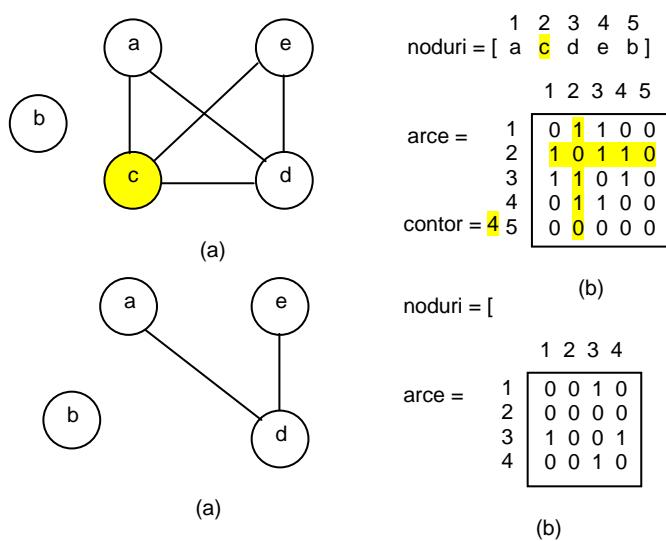
```
procedure InserNod(Tip_Graf* g, Tip_Element e)
    int i, j;
    /*[10.3.1.2.b]*/
    /*incrementare contor noduri*/
    (*g).contor= (*g).contor+1;
    /*se plasează nodul nou*/
    (*g).noduri[(*g).contor]= e;
    /*se initializează matricea de adiacențe pentru nodul nou*/
    pentru (i=1 la (*g).contor) /*initializare coloană nod*/
        (*g).arce[i,(*g).contor]= false;
    pentru (j=1 la (*g).contor) /*initializare linie nod*/
        (*g).arce[(*g).contor,j]= false;
/*InserNod*/
```

- (b) Dacă în tabloul noduri informațiile sunt **ordonate**, atunci:
  - În prealabil trebuie determinat locul în care se va realiza inserția.
  - Se mută elementele tabloului `g.noduri` pentru a crea loc noului nod.
  - Se mută liniile și coloanele matricei de adiacențe `g.arce` pentru a crea loc liniei și coloanei corespunzătoare noului nod.
  - În final se realizează inserția propriu-zisă prin completarea tabloului noduri și a matricei de adiacențe.



**Fig.10.3.1.2.c.** Inserția unui nod într-un graf (varianta tablou noduri ordonat)

- (4) Într-o manieră similară, la **suprimarea** nodului cu indexul `indicNod`, trebuie efectuate mișcări în tablourile `g.noduri` și `g.arce`.
  - Se presupune că se dorește suprimarea nodului `c` din structura graf din figura 10.3.1.2.b.
  - (a) Dacă tabloul `noduri` este **neordonat**.
    - În vederea suprimării, `indicNod` are valoarea 2 precizând nodul `c`, iar suprimarea propriu-zisă presupune ștergerea nodului din `g.Noduri` și modificarea matricei de adiacențe prin excluderea arcelor conexe nodului `c`.
    - Suprimarea nodului `c` se poate realiza prin mutarea ultimului element al tabloului `noduri` în locul său.
    - Pentru păstrarea corectitudinii reprezentării, este necesară ștergerea arcelor conexe lui `c` din matricea de adiacențe.
      - Pentru aceasta se copiază **linia** și **coloana** corespunzătoare ultimului nod din matricea `g.arce`, adică nodul `b`, peste linia și coloana nodului care a fost suprimat.
    - În final se decrementează variabila `g.contor`.
    - Rezultatul suprimării apare în figura 10.3.1.2.d.



**Fig.10.3.1.2.d.** Suprimarea unui nod dintr-o structură graf (varianta tablou noduri neordonat)

- Procedura care implementează aceste activități se numește **SuprimNod** și apare în secvența [10.3.1.2.c].

---

**/\*Suprimarea unui nod. (Tabloul noduri neordonat) - varianta pseudocod\*/**

```

procedure SuprimNod(TipGraf * g, int indicNod)

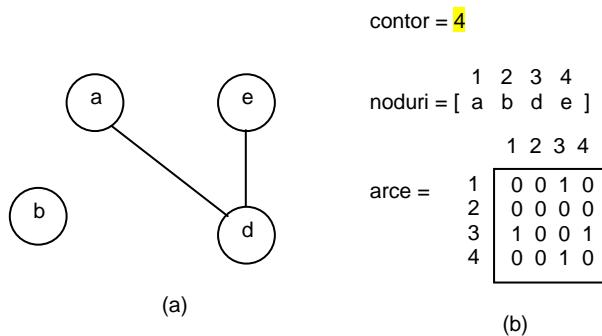
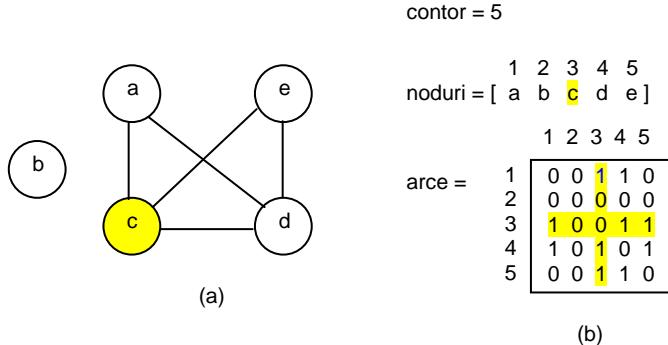
int i, j;

/*actualizare tablou de noduri*/           /*[10.3.1.2.c]*/
/*mutarea ultimului nod în tabloul de noduri*/
(*g).noduri[indicNod]= (*g).noduri[(*g).contor];
/*actualizare matrice de adiacențe*/
pentru (j=1 la (*g).contor) /*mutare linie în matricea
arce*/
    (*g).arce[indicNod,j]= (*g).arce[(*g).contor,j];
pentru (i=1 la (*g).contor) /*mutare coloană în matricea
arce*/
    (*g).arce[i,indicNod]= (*g).arce[i,(*g).contor];
/*decrementare contor noduri*/
    (*g).contor= (*g).contor-1;
/*SuprimNod*/

```

---

- (b) Dacă tabloul noduri este **ordonat**, atunci **suprimarea** presupune :
  - Mutarea cu o poziție a tuturor nodurilor care au indexul mai mare ca indicNod din tabloul noduri .
  - Mutarea liniilor aflate sub linia lui c cu o poziție în sus.
  - Mutarea coloanelor situate la dreapta coloanei lui c cu o poziție spre stânga (fig. 10.3.1.2.e).



**Fig.10.3.1.2.e.** Suprimarea unui nod dintr-o structură graf (varianta tablou noduri ordonat)

- După cum se observă **suprimarea unui nod** presupune implicit și **suprimarea arcelor** conexe lui.
- (5) Există însă posibilitatea de a **șterge arce** fără a modifica mulțimea nodurilor.
  - În acest scop se utilizează procedura **SuprimArc**(TipGraf\* g, TipArc indicArc) secvența [10.3.1.2.d].
    - Datorită simetriei reprezentării stergerea unui arc presupune două modificări în matricea de adiacențe.

---

**{Suprimarea unui arc - varianta pseudocod}**

```

procedure SuprimArc(Tip_Graf * g, Tip_Arc indicArc)
    /*[10.3.1.2.d]*/
/*suprimă arcul (linie, coloană) din matricea de adiacențe*/
(*g).arce[indicArc.linie, indicArc.coloana]= false;
(*g).arce[indicArc.coloana, indicArc.linie]= false;
/*SuprimArc*/

```

---

- În concluzie în studiul de caz 2, **crearea unei structuri de date pentru un TDA graf** presupune două etape:
  - (1) Precizarea nodurilor grafului, implementată printr-o suită de apeluri ale procedurii **InserNod** (câte un apel pentru fiecare nod al grafului).
  - (2) Conectarea nodurilor grafului, implementată printr-o suită de apeluri ale procedurii **InserArc** (câte un apel pentru fiecare arc al grafului).

- În general reprezentarea bazată pe **matrice de adiacențe** este eficientă în cazul **grafurilor dense**.
  - Din punctul de vedere al spațiului de memorie necesar reprezentării, matricea de adiacențe necesită  $n^2$  locații de memorie pentru un graf cu  $n$  noduri.
  - În plus mai sunt necesare locații de memorie pentru memorarea informațiilor aferente celor  $n$  noduri.
  - Crearea grafului necesită un efort proporțional cu  $O(n)$  pentru noduri și aproximativ  $O(n^2)$  pași pentru arce, mai precis  $O(a)$ .
- În consecință, de regulă, utilizarea reprezentării bazate pe **matrice de adiacențe** conduce la algoritmi care necesită un efort de calcul de ordinul  $O(n^2)$ .

### 10.3.2. Implementarea grafurilor cu ajutorul structurilor de adiacențe

- O altă manieră de reprezentare a TDA graf o **constituie structurile de adiacențe** ("adjacency-structures").
  - În cadrul acestei reprezentări, fiecărui nod al grafului i se asociază o **listă de adiacențe** în care sunt înlănuite toate nodurile cu care acesta este conectat.
- În continuare se prezintă două studii de caz pentru implementarea grafurilor cu ajutorul structurilor de adiacență.

#### 10.3.2.1. Studiu de caz 1

- Implementarea structurii de adiacențe se bazează în cazul 1 de studiu pe **liste înlănuite simple**.
  - Începuturile listelor de adiacențe sunt păstrate într-un tablou **stradăj indexat** prin intermediul nodurilor.
  - Inițial în acest tablou se introduc înlănuiri vide, urmând ca inserțiile în liste să fie de tipul "la începutul listei".
  - Adăugarea unui arc care conectează nodul  $x$  cu nodul  $y$  în cadrul acestui mod de reprezentare presupune în cazul grafurilor neorientate:
    - (1) Inserția nodului  $x$  în lista de adiacențe a lui  $y$ .
    - (2) Inserția nodului  $y$  în lista de adiacențe a nodului  $x$ .
- Un exemplu de program care construiește o astfel de structură apare în secvența [10.3.2.1.a].

**/\*Cazul 1. Construcția unui graf utilizând structuri de adiacențe implementate cu ajutorul listelor înlănuite simple - varianta pseudocod C-like\*/**

```
const maxN = 100;

typedef struct Tip_Nod* Ref_Tip_Nod;

/*structura unui nod*/
typedef struct Tip_Nod
{
    int nume;
```

```

    Ref_Tip_Nod urm;
} Tip_Nod;

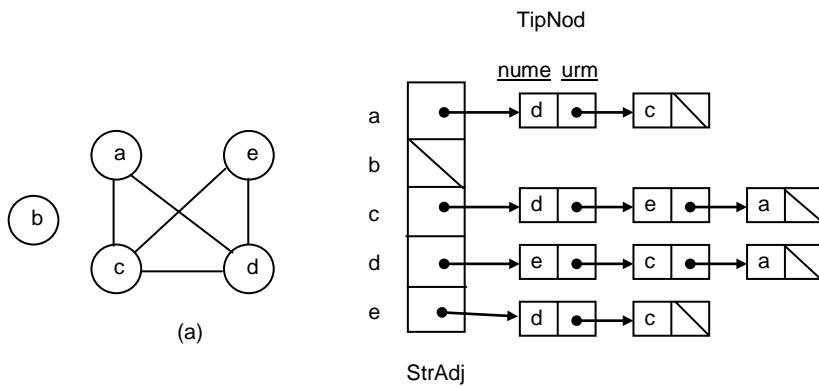
/*structura de adiacențe*/
Ref_Tip_Nod StrAdj[maxN];

int j,x,y;
int N; /*N= numărul de noduri ale grafului*/
int A; /*A= numărul de arce ale grafului*/
Ref_Tip_Nod u,v;

*citeste(N,A); /*N= număr noduri; A= număr arce*/
/*initializare structură de adiacențe*/
pentru (j=0 la N-1)
    StrAdj[j]= null;
/*construcție graf*/
pentru (j=1 la A) /*adăugare arce*/
    *citere(n1,n2); /*[10.3.2.1.a]*/
    x= index(n1); y= index(n2);
    *aloca(u); /*aloca memorie pentru nodul u*/
    u->nume= x; u->urm= StrAdj[y];
    StrAdj[y]= u; /* inserție în față a lui x în lista lui y*/
    *aloca(v); /*aloca memorie pentru nodul v*/
    v->nume= y; v->urm= StrAdj[x];
    StrAdj[x]= v /* inserție în față a lui y în lista lui x*/
    /*pentru*/
/*construcție graf utilizând structuri de adiacențe*/

```

- În figura 10.3.2.1.a se prezintă reprezentarea grafică a structurii construite pornind de la graful (a) din aceeași figură.
- Se face precizarea că datele de intrare (arcele) au fost furnizate în următoarea ordine:  $(a,c)$ ,  $(a,d)$ ,  $(c,e)$ ,  $(c,d)$  și  $(d,e)$ .



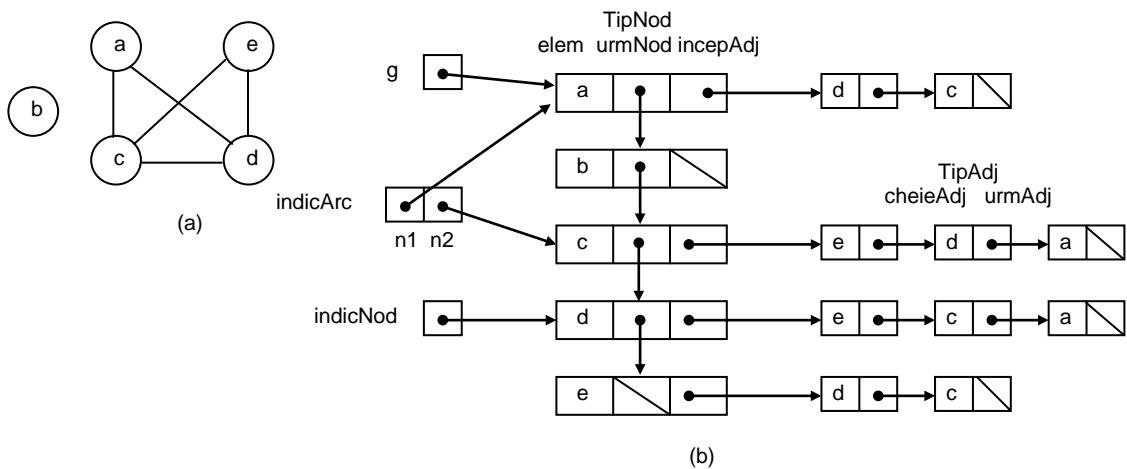
**Fig.10.3.2.1.a.** Graf și structura sa de adiacențe

- Se observă că un arc oarecare  $(x,y)$  este evidențiat în două locuri în cadrul structurii, atât în lista de adiacențe a lui  $x$ , cât și în cea a lui  $y$ .
  - Acum mod redundant de evidențiere își dovedește utilitatea în situația în care se cere să se determine într-o manieră eficientă care sunt nodurile conectate la un anumit nod  $x$ .
- Pentru acest mod de reprezentare, contează ordinea în care sunt prezentate arcele respectiv perechile de noduri, la intrare.

- Astfel, un același graf poate fi reprezentat ca structură de adiacențe în moduri diferite.
- Ordinea în care apar arcele în lista de adiacențe, afectează la rândul ei ordinea în care sunt prelucrate arcele de către algoritm.
  - Funcție de natura algoritmilor utilizați în prelucrare această ordine poate să influențeze sau nu rezultatul prelucrării.

### 10.3.2.2. Studiu de caz 2

- În acest cazul 2 de studiu, implementarea structurilor de adiacențe se bazează pe **structuri multilistă**.
  - Astfel, o **structură de adiacențe** este de fapt o **listă înlănțuită** a nodurilor grafului.
  - Pentru fiecare nod al acestei liste se păstrează o **listă a arcelor**, adică o listă înlănțuită a cheilor nodurilor adiacente.
  - Cu alte cuvinte, o **structură de adiacențe** în acest context este o **listă de liste**.
  - În consecință, fiecare nod al listei nodurilor va conține două înlănțuiriri, una indicând nodul următor, cealaltă, lista nodurilor adiacente.
- În figura 10.3.2.2.a apare structura de adiacențe (b) a grafului (a).



**Fig.10.3.2.2.a.** Reprezentarea unui graf ca și o structură de adiacențe utilizând structuri multilistă

- Implementarea C a structurii multilistă apare în secvența [10.3.2.2.a].

---

*/\*Cazul 2. Implementarea grafurilor utilizând structuri de adiacențe implementate cu ajutorul structurilor multilistă - varianta C\*/*

```

typedef ... Tip_Cheie;
typedef ... Tip_Info;

/*definire tip structura element*/
typedef struct Element
{

```

```

        Tip_Cheie cheie;
        Tip_Info info;
    } Tip_Element

/*definire referință la structura nod al listei de
adiacente*/
typedef struct* Adj Ref_Tip_Adj;

/*definire tip structura nod al listei de adiacențe*/
typedef struct Adj
{
    Tip_Cheie cheie_Adj;
    Ref_Tip_Adj urm_Adj;
} Tip_Adj;

/*definire referință la structura nod al listei nodurilor*/
typedef struct * Nod Ref_Tip_Nod;

/*definire tip structura graf*/
typedef Ref_Tip_Nod Tip_Graf;

/*definire tip structura nod al listei nodurilor*/
typedef struct Nod
{
    Tip_Element elem;
    Ref_Tip_Nod urm_Nod;
    Ref_Tip_Adj incep_Adj;           /*[10.3.2.2.a]*/
} Tip_Nod

/*definire tip structura arc*/
typedef struct Arc
{
    Ref_Tip_Nod n1,n2;
} Tip_Arc

TipGraf g;
Ref_Tip_Nod indic_Nod;
Tip_Arc indic_Arc;
Tip_Cheie k,k1,k2;
Tip_Element e;
-----
```

- Se face precizarea că valorile aferente nodurilor sunt păstrate integral în lista de noduri, în lista de arce apărând numai cheile.
  - Este posibil ca câmpul info să lipsească și deci Tip\_Element= Tip\_Cheie.
- În figura 10.3.2.2.a apare reprezentarea unei structuri de adiacențe implementate cu multiliste cu precizarea câmpurilor aferente.
  - De asemenea sunt prezentate ca exemplu variabilele g, indicNod și indicArc, evidențiindu-se structura fiecareia.
- În cadrul acestei structuri de date:
  - Operatorii **CautăCheieGraf**, **IndicăNod** și **IndicăArc** aparținând setului de operatori extins (varianta 1) utilizează tehnica **căutării liniare** în liste înlăntuite pentru determinarea unui nod a cărui cheie este cunoscută.

- **Insetția unui nod** nou se realizează simplu la începutul listei nodurilor.
- Operatorul ***InserArc***(*Tip\_Graf* \* *g*, *Tip\_Cheie* *k*<sub>1</sub>, *Tip\_Cheie* *k*<sub>2</sub>) presupune inserția lui *k*<sub>1</sub> în lista de adiacențe a lui *k*<sub>2</sub> și reciproc.
  - Și în acest caz inserția se realizează cel mai simplu la începutul listei.
- **Suprimarea unui arc** precizat spre exemplu de indicatorul ***indicArc*** presupune extragerea a două noduri din două liste de adiacențe diferite.
  - Astfel în figura 10.3.2.2.a, variabila *indicArc* conține doi pointeri *n*<sub>1</sub> și *n*<sub>2</sub>, care indică cele două noduri conectate din lista de noduri.
  - În vederea suprimării arcului care le conectează este necesar ca fiecare nod în parte să fie suprimit din lista de adiacențe a celuilalt.
  - În cazul ilustrat, pentru a suprima arcul (a,c)=(c,a) se scoate a din lista lui c, respectiv c din lista lui a.
- Procedura care realizează **suprimarea unui arc** în această manieră a apare în secvența [10.3.2.2.b].
  - Se face precizarea că procedura ***SuprimArc*** este redactată în termenii setului de operatori aplicabili obiectelor aparținând lui **TDA Listă** [Vol 1. &6.2.1].

---

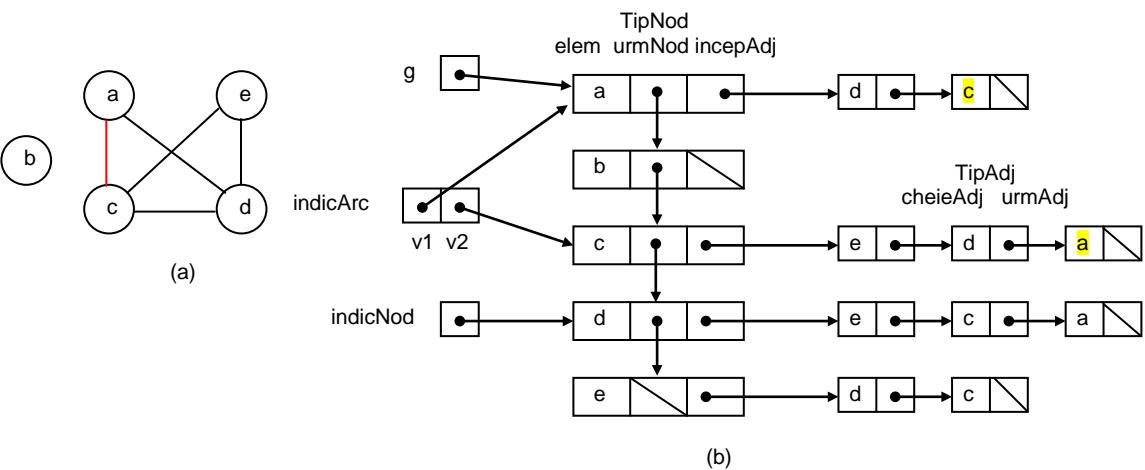
/\*Suprimarea unui arc. În implementare se utilizează operatorii definiți pentru TDA Lista înlățuită simplă - varianta C-like\*/

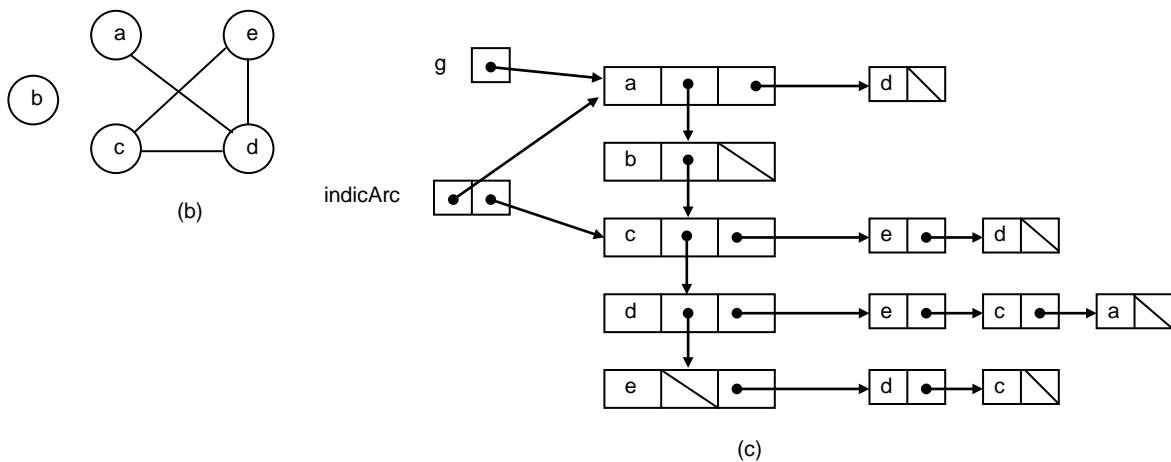
```

procedure SuprimArc(Tip_Graf g, Tip_Arc indic_Arc)
  Ref_Tip_Adj ik1,ik2;                                /*10.3.2.2.b*/
  /*caută cheia nodului n1 în lista de adiacențe a lui n2*/
  ik1= Cauta(indic_Arc.n1->elem.cheie,
  /*caută cheia nodului n2 în lista de adiacențe a lui n1*/
  ik2= Cauta(indic_Arc.n2->elem.cheie,
  /*suprimă nodul n1 din lista de adiacențe a lui n2*/
  Suprima(ik1,indic_Arc.n2->incep_Adj);
  /*suprimă nodul n2 din lista de adiacențe a lui n1*/
  Suprima(ik2,indic_Arc.n1->incep_Adj);
/*SuprimArc*/

```

---





**Fig.10.3.2.2.b.** Structură de adiacențe (c) după suprimarea arcului  $(a, c)$ . Graful original (a), graful după suprimare (b).

- În figura 10.3.2.2.b apare structura de adiacențe (c) aferentă grafului (b) după suprimarea arcului ( $a, c$ ) din graful original (a).
  - **Suprimarea unui nod** dintr-o structură graf, presupune nu numai suprimarea propriu-zisă a nodului respectiv ci în plus suprimarea tuturor arcelor incidente acestui nod.
    - În acest scop, se determină cheia  $k1$  a nodului de suprimat.
    - În continuare, atât timp cât mai există elemente în lista de adiacente a lui  $k1$  se realizează următoarea secvență de operații:
      - (1) Se determină cheia  $k2$  a primului element din lista de adiacențe a lui  $k1$ .
      - (2) Se suprimă arcul  $(k1, k2)$  suprimând pe  $k2$  din lista lui  $k1$  și pe  $k1$  din lista lui  $k2$ .
      - (3) Se reia procesul de la (1).
    - În final se suprimă nodul  $k1$  din lista nodurilor grafului.
  - Procedura care realizează suprimarea unui nod apare în secvența [10.3.2.2.c]
    - Se face de asemenea precizarea că procedura **SuprimNod** este redactată în termenii setului de operatori aplicabili obiectelor aparținând lui **TDA Listă** [Vol 1. &6.2.1].

```
/*Suprimarea unui nod. În implementare se utilizează  
operatorii definiți pentru TDA Listă înlanțuită simplă -  
varianta pseudocod*/
```

```

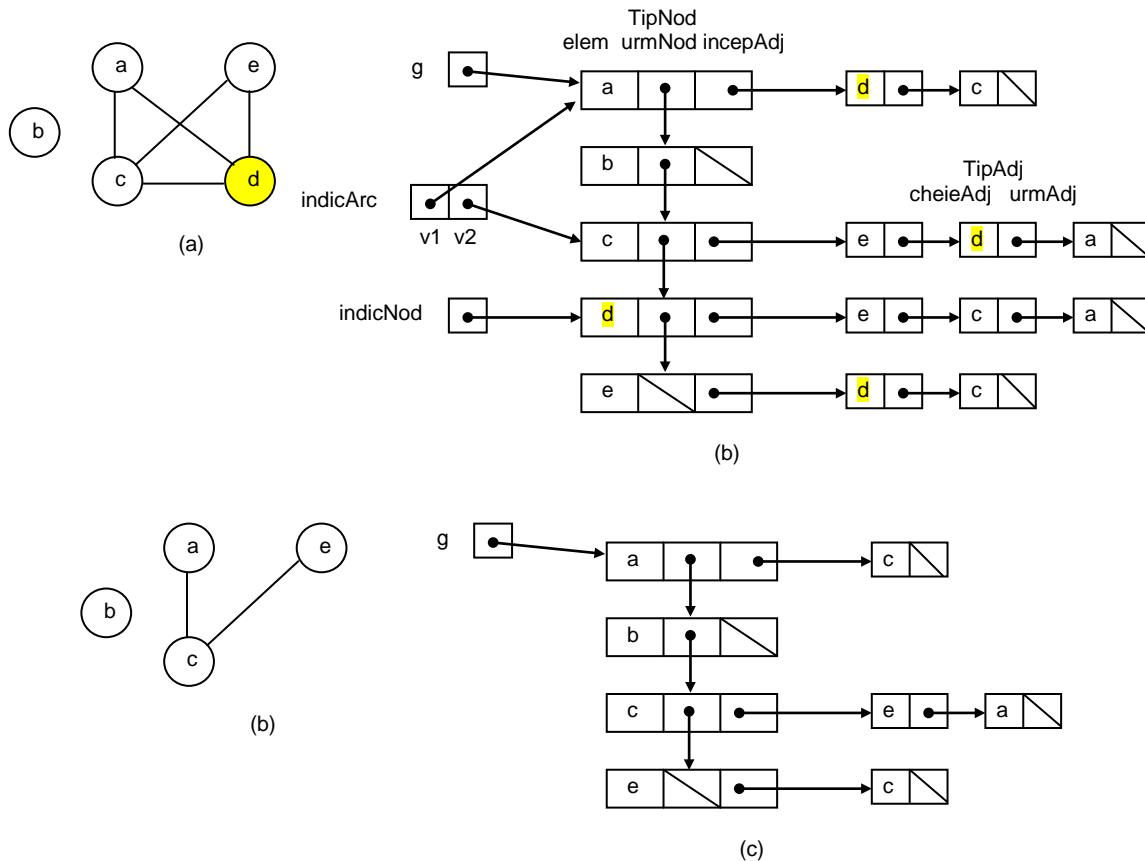
procedure SuprimNod(Tip_Graf g, Ref_Tip_Nod indic_Nod)

    Tip_Cheie k1,k2;
    Ref_Tip_Nod  indic_Nod2;
    Ref_Tip_Adj  ik1,curent;           /*[10.3.2.2.c]*/
    curent= indic_Nod->incep_Adj; /*curent indică lista de
                                    suprimit*/
```

```

        adiacențe a lui k1*/
cat timp (not Fin(current))
    k2= (Primul(current))->cheieAdj; /*k2 indică primul
                                         element al listei k1*/
    /*suprimă pe k2 din lista lui k1*/
    Suprima(Primul(current),current);
    /*caută nodul k2 în graf*/
    indicNod2= Cauta(k2,g);
    /*caută pe k1 în lista lui k2*/
    ik1= Cauta(k1,indicNod2->incepAdj);
    /*suprimă pe k1 din lista lui k2*/
    Suprima(ik1,indicNod2->incepAdj);
    □ /*cat timp*/
    Suprima(indicNod,g) /*suprimă nodul k1 din graf*/
/*SuprimNod*/
-----
```

- În fig.10.3.2.2.c apare reprezentată structura de adiacențe (c) a grafului (b) rezultată în urma suprimării nodului d din graful original (a).

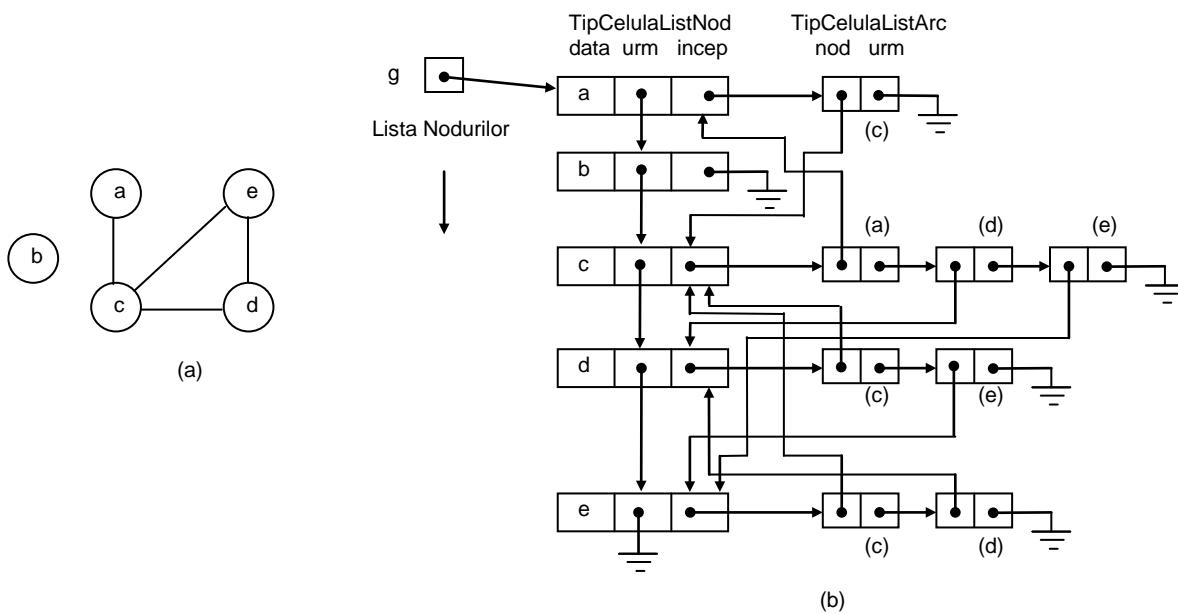


**Fig.10.3.2.2.c.** Structura de adiacențe (c) a grafului (b) rezultată în urma suprimării nodului d din graful original (a).

### 10.3.2.3. Studiu de caz 3

- Cel de al treilea studiu de caz prezintă o altă variantă de implementare a structurilor de adiacență bazată pe **structuri multilistă**.

- Exemplul se referă la varianta Deker de reprezentare a **TDA Graf** [De89].
- Modificarea față de varianta anterioară constă în faptul că în **listele de adiacențe** ale nodurilor, cheile nodurilor adiacente sunt înlocuite cu **pointeri** la nodurile corespunzătoare din **lista nodurilor**.
- Structura de adiacențe, conformă acestei reprezentări a grafului (a) din fig.10.3.2.3.a apare în aceeași figura (b).
- Deși modificarea efectuată conduce la o reprezentare grafică mai complicată, implementarea operatorilor definiți asupra unui **TDA graf** devine mai simplă.
  - Programatorul **nu** trebuie să fie timorat de mulțimea înlățuirilor prezente în structura multilistă, deoarece menținerea evidenței acestora revine programului.
  - De fapt, la nivel de implementare lucrurile se simplifică.



**Fig.10.3.2.3.a.** Structură de adiacențe pentru un graf

- În continuare se prezintă în accepțiunea acestei reprezentări, implementarea setului de operatori restrâns definit asupra **TDA graf** (Varianta 2 (Decker)).
  - **Structurile de date** utilizate în reprezentare apar în secvența [10.3.2.3.a].
  - **Codul** aferent operatorilor apare în secvența [10.3.2.3.b].
  - Se face precizarea că în concordanță cu maniera de definire a tipurilor de date, din lista de parametri a fost omis parametrul **g:TipGraf**, prezența sa nefiind considerată necesară.

---

{Cazul 3. Implementarea grafurilor utilizând structuri de adiacențe implementate cu ajutorul structurilor multilistă. Se utilizează TDA Graf (Varianta 2 (Decker)) - implementare PASCAL}

```
type
  RefTipPozitie = ^TipCelulaListNod;
  RefListArc = ^TipCelulaListArc;
```

```

TipCelulaListNod = record {celulă în lista nodurilor}
    data : TipAtom; {informația
                      apartinând nodului}
    urm : RefTipPozitie; {următoarea
                           celulă în lista nodurilor}
    incep: RefListArc {referință la
                        lista de noduri adiacente}
end;

TipCelulaListArc = record {celulă în lista de adiacențe}
    data: ...; {informația atașată
                  arcului}
    nod : RefTipPozitie; {nodul
                           destinație}
    urm : RefListArc {înlănțuirea în
                       lista de adiacențe}
end;

TipArc = record {structură arc}
    nod1,nod2: RefTipPozitie
end;

TipGraf = RefTipPozitie; {structură graf}
-----

procedure Creaza(var g: TipGraf);
begin
    g:= nil
end; {Creaza}

function Adiacent(p,q: RefTipPozitie): boolean;
{Primeste pointerii p și q la lista de noduri a lui g. Caută
 în lista de arce a lui p, celula care indica nodul q}

var gata: boolean; {specifică terminarea traversării listei
                     de adiacențe}
    curent: RefListArc; {pointer la celula curentă a listei
                          de adiacențe }

begin {Adiacent}
    Adiacent:= false;
    curent:= p^.incep; {începutul listei de adiacențe a lui
                         p}
    gata:= false;
    while not gata do
        if curent=nil then
            gata:= true
        else
            if curent^.nod = q then [10.3.2.3.b]
                begin
                    gata:= true;
                    Adiacent:= true
                end
            else
                curent:= curent^.urm
    end; {Adiacent}

procedure Modifica(a: TipAtom; var p: RefTipPozitie);
begin

```

```

p^.data := a
end; {Modifica}

function Furnizeaza(p: RefTipPozitie ): TipAtom;
begin
  Furnizeaza := p^.data
end; {Furnizeaza}

procedure SuprimNod(p: RefTipPozitie; var g: TipGraf);
var NodCurent: RefTipPozitie;

procedure SuprimaCel(n: RefTipPozitie; var început:
                      RefListArc);
{procedură recursivă care suprimă din lista indicată de
 "început", celula având câmpul nod egal cu n}
var temp: RefListArc;

begin
  if început<>nil then
    if început^.nod = n then
      begin
        temp:= început;
        început:= început^.urm;
        DISPOSE(temp)
      end
    else
      SuprimaCel(n,început^.urm)
  end; {SuprimaCel}

procedure StergeList(var început: RefListArc);
{suprimă recursiv toate elementele listei indicate de
 început}
begin
  if început<>nil then
    begin
      StergeList(început^.urm);
      DISPOSE(început)
    end
  end; {StergeList}

begin {SuprimNod}
  NodCurent:= g;
  WHILE NodCurent<>nil do {parcurge lista de noduri a
                                grafului}
    begin
      if p<>NodCurent then
        SuprimaCel(p,NodCurent^.incep); {pentru fiecare
                                         nod al grafului, parcurge lista sa de adiacențe
                                         și suprimă nodul p}
        NodCurent:= NodCurent^.urm           [10.3.2.3.b]
      end
      StergeList(p^.incep); {șterge lista de adiacențe a
                             nodului indicat de p}
      SuprimaCel(g,p) {suprimă nodul p}
  end; {SuprimNod}

procedure InserNod(var p: RefTipPozitie; var g: TipGraf);
begin

```

```

New(p);
p^.urm:= g;      { inserție în față}
p^.incep:= nil;
g:= p
end; {InserNod}

procedure InserArc(e: TipArc; var g: TipGraf);
var temp: RefListArc;
begin
    if not Adiacent(e.nod1,e.nod2) then
        begin
            temp:= e.nod1^.incep; { inserție nod2 în lista nod1}
            New(e.nod1^.incep);
            e.nod1^.incep^.nod:= e.nod2;
            e.nod1^.incep^.urm:= temp;
            temp:= e.nod2^.incep; { inserție nod1 în lista nod2}
            New(e.nod2^.incep);
            e.nod2^.incep^.nod:= e.nod1;
            e.nod2^.incep^.urm:= temp
        end
    end; {InserArc}

procedure SuprimArc(e: TipArc; var g: TipGraf)

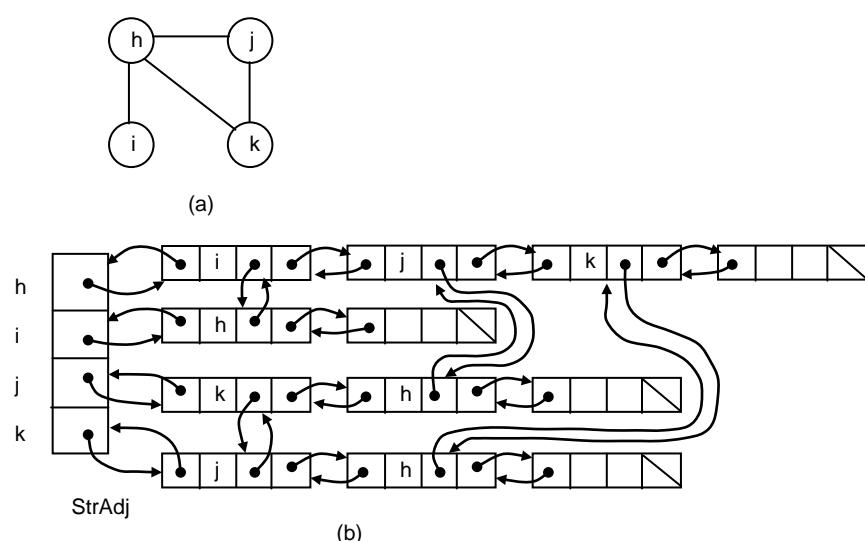
procedure SuprimaCel(n: RefTipPozitie; var inceput:
                      RefListArc);
{procedură recursivă care suprimă din lista indicată de
 "început", celula având câmpul nod egal cu n}
.
.
.
end; {SuprimaCel}                                {[10.3.2.3.b]}

begin {SuprimArc}
    if Adiacent(e.nod1,e.nod2) then
        begin
            SuprimaCel(e.nod2,e.nod1^.incep); {suprimare nod2
                                                din lista nod1}
            SuprimaCel(e.nod1,e.nod2^.incep)  {suprimare nod1
                                                din lista nod2}
        end
    end; {SuprimArc}
-----
```

### 10.3.3. Considerente referitoare la stabilirea modului de reprezentare a unui TDA graf.

- Din punct de vedere al eficienței, alegerea unei anumite reprezentări pentru o structură de date depinde esențial de **natura operațiilor** care se execută asupra ei.
  - Acest lucru este valabil și în cazul grafurilor.
- (1) Un prim aspect care poate fi luat în considerare în **alegerea reprezentării**, este cel legat de **dinamica grafului** avut în vedere.

- Dacă graful este puternic și imprevizibil **variabil în timp**, la baza reprezentării trebuie să stea o **structură dinamică**, altfel poate fi utilizată o **structură statică**.
- (2) Un al doilea aspect se referă la **frecvența** cu care se execută diferitele **operații** asupra grafului de reprezentat.
- De regulă, astfel de operații includ:
  - (a) **Identificarea** (căutarea) unui nod sau unui arc.
  - (b) **Inserția** unui nod sau unui arc.
  - (c) **Suprimarea** unui nod sau arc.
  - (d) **Traversarea** unui graf.
- (a) Referitor la **identificarea** sau **căutarea** unui nod sau arc al unui graf:
  - În cazul reprezentării bazate pe **matrice de adiacențe** **timpul de acces** la un element al matricei (grafului) **nu** depinde de dimensiunea acesteia.
  - În cazul reprezentării prin **structuri de adiacențe**, **timpul de acces** depinde de poziția în listă a elementului căutat, respectiv de ordinea de creare a structurii.
  - În consecință, reprezentarea bazată pe **matrici de adiacențe** este mai utilă în cazul algoritmilor în care se verifică frecvența prezență arcelor sau nodurilor într-un graf.
- (b) **Inserția** nodurilor se realizează simplu în ambele reprezentări.
- (c) În ceea ce privește **suprimarea** nodurilor (arcelor) ea se realizează simplu în cazul reprezentării bazate pe **matrice de adiacențe** și mai complicat în cazul **structurii de adiacențe**.
  - În ultimul caz, pentru a șterge un nod  $x$  și toate arcele conexe, nu este suficient să se anuleze lista sa de adiacențe, ci în plus, nodul  $x$  trebuie căutat și suprimit din listele de adiacențe ale tuturor nodurilor care apar în propria sa listă.
  - Această situație poate fi rezolvată într-o manieră eficientă, **înlănțuind** între ele **nodurile corespunzătoare** unui anumit **arc** și reprezentând structura de adiacențe prin **liste dublu înlănțuite** (fig.10.3.3).



**Fig. 10.3.3.** Exemplu de structură de adiacențe complexă

- În aceste condiții la suprimarea unui arc, nodurile corespunzătoare pot fi ușor înlăturate din listele de adiacențe parcurgând înlănțuirile.
- Este însă evident faptul că înlănțuirile suplimentar introduse, deși simplifică suprimarea, complică atât structura în ansamblul ei cât și modul ei de prelucrare (spre exemplu inserția).
- Din acest motiv, ele nu trebuie să introducă decât în situația în care sunt absolut necesare.
- (d) Traversarea grafurilor va fi abordată într-un paragraf ulterior. Ea depinde în manieră esențială de modul de reprezentare al structurii.
- (3) **Grafurile orientate** și cele **ponderate** pot fi reprezentate prin structuri similare.
  - (a) Spre exemplu, pentru **grafurile orientate** se pot utiliza aceleași reprezentări pentru structura graf, exceptând faptul că un **arc** este reprezentat doar **o singură dată** în structură.
  - Astfel arcul de la nodul  $x$  la nodul  $y$  este reprezentat prin valoarea adevărată în câmpul  $A[x, y]$  al **matricei de adiacențe**, sau prin apariția lui  $y$  în lista de adiacențe a lui  $x$  în reprezentarea bazată pe **structuri de adiacențe**.
  - De fapt un **graf neorientat** poate fi conceput ca și un **graf orientat** cu arce direcționate în ambele sensuri, între toate perechile de noduri conectate.
  - (b) În cazul **grafurilor ponderate**, se pot de asemenea utiliza aceleași reprezentări cu diferența că:
    - În **matricea de adiacențe** se memorează **ponderile arcelor** în locul valorilor booleene.
      - Se utilizează o **pondere inexistentă convenită** pentru a reprezenta valoarea ("false") adică absența conexiunii.
    - În nodurile **listelor de adiacențe** se includ **câmpuri suplimentare** pentru memorarea ponderii arcelor.
- (4) În unele cazuri, în vederea modelării unor situații mai complicate, sau pentru a reduce regia unor algoritmi complecși, nodurilor și arcelor unui graf li se pot asocia și alte **informații**.
  - Aceste informații pot fi păstrate în cadrul structurii propriu-zise care materializează graful sau de la caz la caz în **structuri conexe**.
- (5) În ceea ce privește **spațiul de memorie** necesar se pot face următoarele considerații.
  - Presupunând că numărul nodurilor grafului de reprezentat este  $n$  și numărul arcelor  $a$ .
  - În implementarea bazată pe tablouri, **matricea de adiacențe** împreună cu **tabloul nodurilor** necesită un spațiu de memorie proporțional cu  $n+n^2$ .
  - Pentru un graf cu multe noduri și arce relativ puține, risipa de spațiu este mare.

- În reprezentarea bazată pe **structuri înlățuite** fiecare nod apare odată în lista nodurilor și fiecare arc  $(x, y)$  consumă două noduri din listele de adiacențe, unul pentru  $x$  și altul pentru  $y$ .
  - Astfel în reprezentarea bazată pe **structuri de adiacențe**, spațiul de memorie necesar este proporțional cu  $n+2a$ .
  - Mai trebuie luat în considerare și faptul că înlățuirile ca atare consumă ele însele spațiu de memorie.
- Elementul hotărâtor în departajarea performanțelor celor două metode de reprezentare, din punctul de vedere al **spațiului de memorie** necesar îl reprezintă **numărul de arce**  $a$ .
  - După cum s-a precizat în §10.1, **numărul de arce**  $a$  poate lua valoari în domeniul  $[0, (1/2)n(n-1)]$ .
  - În consecință din valoarea maximă a numărului de arce  $a=(1/2)n(n-1)$  se deduce relația  $2a=n^2-n < n^2$ .
  - Rezultă că  $2a < n^2$ , deci reprezentarea înlățuită în principiu este mai eficientă ca cea matricială, din punctul de vedere considerat.
- După cum s-a mai precizat însă la o analiză mai aprofundată trebuie luat în considerare și **spațiul de memorie auxiliar** necesar depozitării informațiilor aferente nodurilor și mai ales înlățuirilor propriu-zise.

## 10.4. Tehnici fundamentale de traversare a grafurilor

- Rezolvarea eficientă a problemelor curente referitoare la grafuri, presupune de regulă, **traversarea** (vizitarea sau parcurgerea) nodurilor și arcelor acestora într-o manieră sistematică.
- În acest scop s-au dezvoltat două **tehnici fundamentale**, una bazată pe **căutarea în adâncime**, cealaltă bazată pe **căutarea prin cuprindere**.

### 10.4.1. Traversarea grafurilor prin tehnica căutării "în adâncime" ("Depth-First Search")

- **Căutarea în adâncime** este una dintre tehnicele fundamentale de traversare a grafurilor.
  - **Căutarea în adâncime** constituie nucleul în jurul căruia pot fi dezvoltăți numeroși algoritmi eficienți de prelucrare a grafurilor.
  - Spre exemplu, **traversarea în preordine a arborilor** își are originea în această tehnică.
- **Principiul căutării "în adâncime"** într-un graf  $G$  este următorul:
  - (1) Se marchează inițial toate nodurile grafului  $G$  cu marca "nevizitat".
  - (2) Căutarea debutează cu selecția unui nod  $n$  a lui  $G$  pe post de nod de pornire și cu marcarea acestuia cu "vizitat".
  - (3) În continuare, fiecare nod nevizitat adiacent lui  $n$  este "căutat" la rândul său, aplicând în mod recursiv același algoritm de "căutare în adâncime".
  - (4) Odată ce toate nodurile la care se poate ajunge pornind de la  $n$  au fost vizitate în maniera mai sus precizată, cercetarea lui  $n$  este terminată.

- (5) Dacă în graf au rămas noduri nevizitate, se selectează unul dintre ele drept nod nou de pornire și procesul se repetă până când toate nodurile grafului au fost vizitate.
- Această tehnică se numește căutare “în adâncime” (“**depth-first**”) deoarece parcurgerea grafului se realizează înaintând “în adâncime” pe o direcție aleasă atât timp cât acest lucru este posibil.
  - Spre **exemplu**, presupunând că  $x$  este **ultimul nod vizitat**, căutarea în adâncime selectează un arc neexplorat conectat la nodul  $x$ .
  - Fie  $y$  nodul corespunzător acestui arc.
    - Dacă nodul  $y$  a fost deja vizitat, se caută un alt arc neexplorat conectat la  $x$ .
    - Dacă  $y$  nu a fost vizitat anterior, el este marcat “vizitat” și se inițiază o nouă căutare începând cu nodul  $y$ .
  - În momentul în care se epuizează căutarea pe toate drumurile posibile pornind de la  $y$ , se revine la nodul  $x$ , (principiul recursivității) și se continuă în aceeași manieră selecția arcelor neexplorate ale acestui nod, până când sunt epuizate toate posibilitățile care derivă din  $x$ .
  - Se observă clar **tendința inițială de adâncire**, de îndepărțare față de sursă, urmată de o revenire pe măsura epuizării tuturor posibilităților de traversare.
- Se consideră:
  - O **structură graf** într-o reprezentare oarecare.
  - Un tablou **marc** ale cărui elemente corespund nodurilor grafului.
    - În tabloul **marc** se memorează faptul că un nod al grafului a fost sau nu vizitat.
- Schița de principiu a **algoritmului de căutare în adâncime** apare în secvența [10.4.1.a].

---

**/\*Căutarea "în adâncime". Schița de principiu a algoritmului. Varianta pseudocod\*/**

```

subprogram CautaInAdincime(Tip_Nod x)
  Tip_Nod y;

  marc[x] = vizitat;                                     /*[10.4.1.a]*/
  pentru (fiecare nod  $y$  adiacent lui  $x$ )
    daca (marc[y] este nevizitat)
      CautaInAdincime(y);
  /*CautaInAdincime*/

```

---

#### **10.4.1.1. Căutarea "în adâncime", varianta CLR**

- O variantă mai elaborată a **traversării în adâncime** este cea propusă de Cormen, Leiserson și Rivest [CLR92].
  - Conform acesteia, pe parcursul traversării, nodurile sunt **colorate** pentru a marca stările prin care trec.

- Din aceste motive **traversarea grafurilor** este cunoscută și sub denumirea de "colorare a grafurilor".
    - Culorile utilizate sunt alb, gri și negru.
  - Toate nodurile sunt inițial colorate în **alb**, în timpul traversării sunt colorate în **gri** iar la terminarea traversării sunt colorate în **negru**.
    - Un nod **alb** care este descoperit prima dată în traversare este colorat în **gri**.
    - Un nod este colorat în **negru** când toate nodurile adiacente lui au fost descoperite.
      - În consecință, nodurile gri și negre au fost deja întâlnite în procesul de traversare, ele marcând modul în care avansează traversarea.
    - Un nod colorat în **gri** poate avea și noduri adiacente albe.
      - Nodurile **gri** marchează frontieră între nodurile descoperite și cele nedescoperite și ele se păstrează de regulă în structura de date (stiva) asociată procesului de traversare.
  - În procesul de traversare se poate construi un **subgraf asociat traversării**, subgraf care include arcele parcuse în traversare și care este de fapt un **graf de precedențe**.
    - Acest subgraf este un **graf conex aciclic**, adică un **arbore**, care poate fi simplu reprezentat prin tehnica "**indicator spre părinte**".
  - **Tehnica de construcție a subgrafului** este următoarea:
    - Atunci când în procesul de căutare se ajunge de la nodul  $u$  la nodul  $v$ , nodul  $v$  nefiind vizitat încă, acest lucru se marchează în subgraful asociat  $s$  prin  $s[v] = u$ , adică  $u$  este părintele lui  $v$ .
    - **Graful de precedențe** este descris formal conform relațiilor [10.4.1.1.a].
- 

$$\begin{aligned} G_{\text{pred}} &= (N, A_{\text{pred}}) \\ A_{\text{pred}} &= \{ (s[v], v) : v \in N \text{ și } s[v] \neq \text{nil} \} \end{aligned} \quad [10.4.1.1.a]$$


---

- **Subgrafurile predecesorilor** asociate căutării în adâncime într-un graf precizat, formează o **pădure de arbori de căutare în adâncime**.
- Pe lângă crearea propriu-zisă a subgrafului predecesorilor fiecărui nod i se pot asocia două **mărci de timp** ("timestamps"):
  - (1)  $i[v]$  memorează **momentul descoperirii** nodului  $v$  (colorarea sa în **gri**).
  - (2)  $f[v]$  memorează **momentul terminării** explorării nodurilor adiacente lui  $v$  (colorarea sa în **negru**).
- **Mărcile de timp** sunt utilizate în mulți algoritmi referitori la grafuri și ele precizează în general comportamentul lor în timp.
  - În cazul de față, pentru simplitate, **timpul** este conceput ca un întreg care ia valori între 1 și  $2|N|$ , întrucât este incrementat cu 1 la fiecare **descoperire** respectiv **terminare** de examinare a fiecărui din cele  $|N|$  noduri ale grafului.
- Varianta de **căutare în adâncime** propusă de Cormen, Leserson și Rivest apare în secvența [10.4.1.1.b].

```
/*Cautarea "în adâncime". Schița de principiu a
algoritmului. Varianta 2 (Cormen, Leiserson, Rivest} -
format pseudocod*/
```

```
procedure TraversareInAdâncime(Tip_Graf G)
[1]   pentru (fiecare nod u∈N(G))
[2]       | culoare[u]=alb;
[3]       | sp[u]=null;
[4]       | | /*pentru*/
[4]   temp=0;
[5]   pentru (fiecare nod u∈N(G))
[6]       | daca (culoare[u] este alb)
[7]           | | CautareInAdâncime(u);
/*TraversareInAdâncime*/                                /*[10.4.1.1.b]*/

procedure CautareInAdâncime(Tip_Nod u)
[1]   culoare[u]=gri;
[2]   temp=temp+1; i[u]=temp;
[3]   pentru (fiecare nod v adiacent lui u)
[4]       | daca (culoare[v] este alb)
[5]           | | sp[v]=u;
[6]           | | | CautareInAdâncime(v)
[6]           | | | | /*daca*/
[7]   culoare[u]=negru;
[8]   temp=temp+1; f[u]=temp;
/*CautareInAdâncime*/
```

- **Analiza algoritmului.**

- Liniile 1-3 și 5-7 ale lui **TraversareInAdâncime** se execută pentru fiecare nod, deci necesită un timp proporțional cu  $O(N)$ , excluzând timpul necesar execuției apelurilor procedurii de căutare propriu-zise.
- Procedura **CăutareInAdâncime** este apelată exact odată pentru fiecare nod  $v \in N$ , deoarece ea este invocată doar pentru noduri albe și primul lucru pe care îl face este să coloreze respectivul nod în gri.
- Liniile 3-6 ale procedurii **CăutareInAdâncime** se execută pentru fiecare arc, deci într-un interval de timp proporțional cu  $|Adj[v]|$  unde este valabilă formula [10.4.1.1.c].

$$\sum_{v \in N} |Adj[v]| = O(A) \quad [10.4.1.1.c]$$

- În consecință rezultă că costul total al execuției liniilor 3-6 ale procedurii **CăutareInAdâncime** este  $O(A)$ .
- **Timpul total de execuție al traversării prin căutare în adâncime** este deci  $O(N+A)$ .
- În continuare se detaliază această tehnică de căutare pentru diferite modalități de implementare a grafurilor.

#### 10.4.1.2. Căutare "în adâncime" în grafuri reprezentate prin structuri de adiacențe

- Procedura care implementează **căutarea în adâncime** în grafuri reprezentate prin **structuri de adiacențe** apare în secvența [10.4.1.2.a].
  - Procedura **Traversare1** completează tabloul `marc[1..maxN]` pe măsură ce sunt traversate (vizitate) nodurile grafului.
  - Tabloul `marc` este poziționat inițial pe zero, astfel încât `marc[x]=0` indică faptul că nodul `x` nu a fost încă vizitat.
  - Pe parcursul traversării câmpul `marc` corespunzător unui nod `x` se completează în momentul începerei vizitării cu valoarea "id", valoare care se incrementează la fiecare nod vizitat și care indică faptul că nodul `x` este cel de-al `id`-lea vizitat.
  - Procedura de traversare utilizează procedura recursivă **CautaInAdâncime** care realizează vizitarea tuturor nodurilor aparținătoare acelei componente conexe a grafului, căreia îi aparține nodul furnizat ca parametru.
  - Se face precizarea că procedura **Traversare1** din secvența [10.4.1.2.a] se referă la reprezentarea TDA **graf** bazată pe **structuri de adiacență implementate cu ajutorul listelor înlántuite simple**.

---

```
/*Traversarea "în adâncime" a grafurilor reprezentate prin
structuri de adiacențe implementate cu ajutorul listelor
înlăntuite simple - varianta pseudocod*/
```

```
int marc[maxN]; /*tabloul marc*/
int id;

subprogram CautaInAdincime(int x);
    Ref_Tip_Nod t;

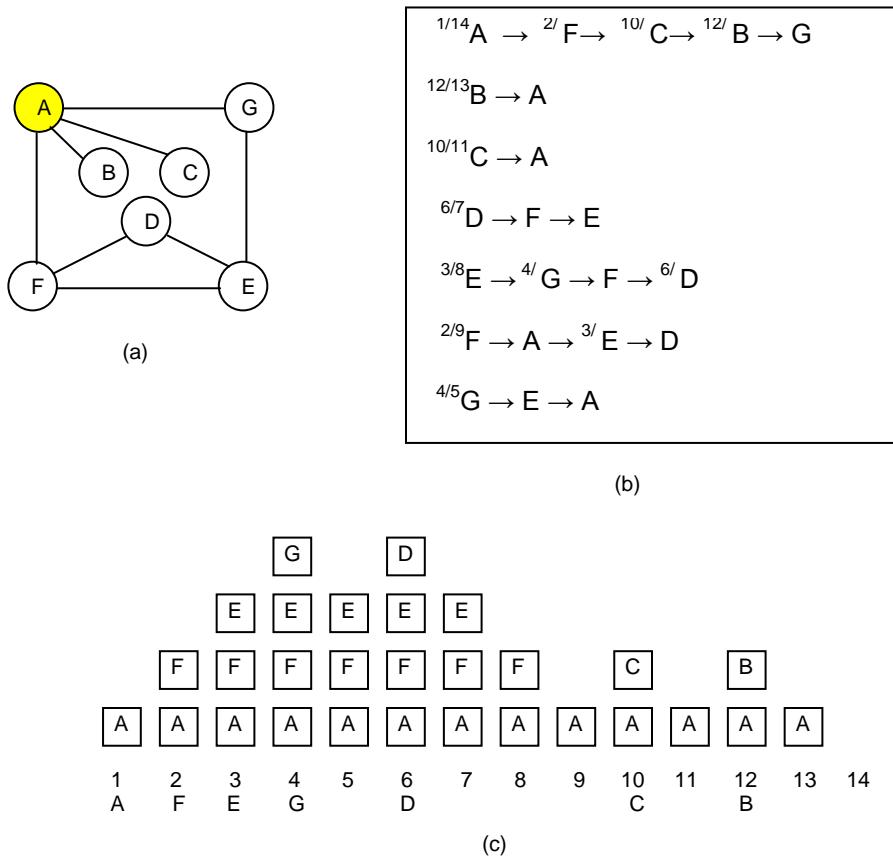
    id= id+1; marc[x]= id;
    *scrie(t->nume);
    t= Stradj[x];
    cat timp (t<>null)                                /*10.4.1.2.a*/
        daca (marc[t->nume]==0)
            CautaInAdincime(t->nume);
        t= t->urm
        □ /*cat timp*/
/*CautaInAdincime*/

subprogram Traversare1;
    int x;

    id= 0;
    pentru (x=1 la N)
        marc[x]= 0;
    pentru (x=1 la N)
        daca (marc[x]==0)
            CautaInAdincime(x);
            *scrie_rand_ecran (writeln);
        □ /*daca*/
/*Travesare1*/
```

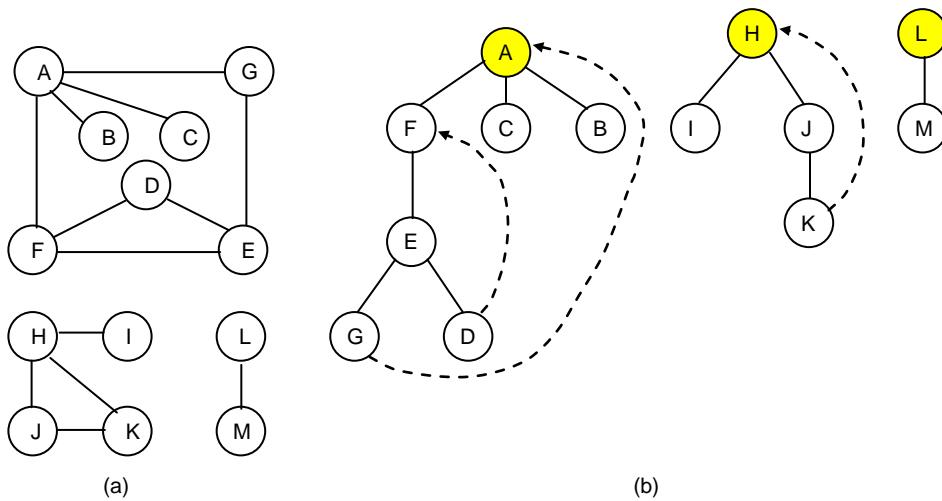
- **Vizitarea unui nod** presupune parcurgerea tuturor arcelor conexe lui, adică parcurgerea listei sale de adiacențe și verificarea pentru fiecare arc în parte, dacă el conduce la un nod care a fost sau nu vizitat.
  - În caz că nodul este nevizitat procedura se apelează recursiv pentru acel nod.
- Procedura **Traversare1**, parcurge tabloul **marc** și apelează procedura **CautăInAdâncime** pentru nodurile nevizitate, până la traversarea integrală a grafului.
  - Trebuie observat faptul că fiecare apel al procedurii **CautăInAdâncime** realizat din cadrul procedurii **Traversare1** asigură parcurgerea unei **componente conexe a grafului** și anume a componentei conexe care conține nodul selectat.
  - În figura 10.4.1.2.a apare **urma execuției** algoritmului de căutare în adâncime pentru graful (a) din figură.
    - **Structura de adiacențe** aferentă grafului apare în aceeași figură (b).
    - **Evoluția conținutului stivei** apare în în aceeași figură (c).
    - Se menționează faptul că nodurile structurii de adiacențe au atașat un **exponent fracționar** care precizează:
      - La **numărător** momentul la care **este descoperit nodul**, adică momentul la care este introdus în stivă (colorarea în gri).
      - La **numitor**, momentul **terminării explorării nodului**, adică momentul la care este scos din stivă (colorarea în negru).
  - Graful este **traversat** drept consecință a apelului **CautăInAdâncime(A)**, efectuat în ciclul **for** al procedurii **Traversare1**.
  - În continuare se prezintă la nivel de detaliu parcurgerea prin **căutare în adâncime** a grafului (a) din figura 10.4.1.2.a.
    - Nodul de pornire A este introdus în stivă la momentul 1.
    - Pentru nodul A se parcurge lista sa de adiacențe, primul arc traversat fiind AF, deoarece F este primul nod din lista de adiacențe a lui A.
    - În continuare se apelează procedura **CautăInAdâncime** pentru nodul F, în consecință nodul F este introdus în stivă la momentul 2 și se traversează arcul FA, A fiind primul nod din lista de adiacențe a lui F.
    - Întrucât nodul A a fost deja descoperit (intrarea sa în tabloul **marc** conține o valoare nenulă), se alege în continuare arcul FE, E fiind nodul următor în lista de adiacențe a lui F.
    - Nodul E se introduce în stivă la momentul 3.
    - Se traversează în continuare arcul EG (G se introduce în stivă la momentul 4) G fiind primul nod din lista de adiacențe a lui E.
    - În continuare se traversează arcul GE respectiv GA (nodurile E respectiv A fiind deja descoperite), moment în care se termină parcurgerea listei de adiacențe a lui G, fapt realizat la timpul 5.

- Se elimină G din stivă, se revine în lista nodului E și se continuă parcurgerea listei sale de adiacențe traversând arcele EF (nodul F a fost deja vizitat) și ED.
- Ca atare nodul D este descoperit la momentul 6 și în continuare vizita lui D presupune traversarea arcelor DE și DF care niciunul nu conduce la un nod nou.
- Terminarea parcurgerii listei de adiacențe a lui D are drept consecință finalizarea vizitării lui și scoaterea din stivă la momentul 7.
- Se revine în lista de adiacențe a lui E. Deoarece D a fost ultimul nod din lista de adiacențe a lui E, vizita lui E se încheie la momentul 8 și revine în nodul F a cărui vizitare se încheie prin parcurgerea arcului FD (D deja vizitat).
- Se elimină F din stivă la momentul 9, se revine în lista lui A și se găsește nodul C nevizitat încă.
- C se introduce în stivă la momentul 10, i se parurge lista care îl conține doar pe A deja vizitat și este extras din stivă la momentul 11.
- Se continuă parcurgerea listei de adiacențe a nodului A și se găsește nodul B care suferă un tratament similar lui C, la momentele 12 respectiv 13.
- În final se ajunge la sfârșitul listei lui A, A se scoate din stivă la momentul 14 și procesul de traversare se încheie.



**Fig.10.4.1.2.a.** Urma execuției algoritmului recursiv de căutare în adâncime

- Un alt mod de a urmări desfășurarea operației de căutare în adâncime este acela de a redesena graful traversat pornind de la **apelurile recursive** ale procedurii **CautăInAdâncime**, ca în figura 10.4.2.1.b.



**Fig.10.4.2.1.b.** Arborescences for depth-first search

- În figura 10.4.2.1.b:
  - O **linie continuă** indică faptul că nodul aflat la extremitatea sa inferioară, a fost găsit în procesul de căutare în lista de adiacențe a nodului aflat la extremitatea sa superioară și nefiind vizitat la momentul considerat, s-a realizat pentru el un apel recursiv al procedurii de căutare.
  - O **linie punctată** indică un nod descoperit în lista de adiacențe a nodului sursă pentru care apelul recursiv nu se realizează, deoarece nodul a fost deja vizitat sau este în curs de vizitare, adică este memorat în stiva asociată prelucrării.
    - Ca atare condiția din instrucția **if** a procedurii **CautăInAdâncime** nu este îndeplinită nodul fiind deja marcat cu vizitat în tabloul **marc** și în consecință pentru acest nod **nu** se realizează un apel recursiv al procedurii.
- Aplicând această metodă, pentru fiecare componentă conexă a unui graf se obține un **arbore de acoperire** ("spanning tree") numit și **arbore de căutare în adâncime** al componentei.
  - Traversarea în **preordine** a arborelui de căutare în adâncime, furnizează nodurile în ordinea în care sunt **prima dată** întâlnite în procesul de căutare.
  - Traversarea în **postordine** a arborelui de căutare în adâncime, furnizează nodurile în ordinea în care **cercetarea lor se încheie**.
- Este important de subliniat faptul că întrucât ramurile arborilor de acoperire, materializează arcele grafului, **multimea (pădurea) arborilor de căutare în adâncime** asociati unui graf reprezintă **o altă metodă de reprezentare grafică a grafului**.
- O proprietate esențială a **arborilor de căutare în adâncime** pentru grafuri neorientate este aceea că **liniile punctate** indică întotdeauna un **strămoș** al nodului în cauză.
- În orice moment al execuției algoritmului, nodurile grafului se împart în trei **clase**:

- (1) **Clasa I-a** conține nodurile pentru care procesul de vizitare s-a terminat (colorate în **negră**).
- (2) **Clasa II-a** conține nodurile care sunt în curs de vizitare (colorate în **gri**).
- (3) **Clasa III-a** conține nodurile la care nu s-a ajuns încă (colorate în **alb**).
- (1) În ceea ce privește **clasa I-a** de noduri, datorită modului de implementare a procedurii de căutare, nu va mai fi selectat nici un arc care indică vreun astfel de nod, motiv pentru care aceste arce **nu** se reprezintă în structura arbore.
- (2) În ceea ce privește **clasa a III-a** de noduri, aceasta cuprinde nodurile pentru care se vor realiza apeluri recursive și arcurile care conduc la ele vor fi marcate cu **linie continuă** în arbore.
- (3) Mai rămân nodurile din **clasa II-a**: acestea sunt nodurile care au apărut cu siguranță în drumul de la nodul curent la rădăcina arborelui. Ele sunt colorate în gri și sunt memorate în stiva asociată căutării.
  - Ca atare, orice arc procesat care indică vreunul din aceste noduri apare reprezentat cu **linie punctată** în **arborele de căutare în adâncime**.
- În concluzie:
  - Arcele marcate **continuu** în figura 10.4.1.2.b se numesc **arce de arbore**.
  - Arcele marcate **punctat** se numesc **arce de return**.
- Din punct de vedere **formal**, dacă  $x$  și  $y$  sunt noduri ale grafului ce urmează a fi traversat atunci:
  - (1) **Arcul de arbore** este acel arc  $(x, y)$  al grafului pentru care instanța de apel a procedurii recursive **CautăInAdâncime**( $x$ ) apelează instanța **CautăInAdâncime**( $y$ ).
    - La momentul apelului, nodul  $y$  aparține clasei a III-a el fiind colorat în **alb**, adică nevizitat.
  - (2) **Arcul de return**  $(x, y)$  este un arc al grafului întâlnit în procesul de vizitare al nodului  $x$ , adică întâlnit în lista lui de adiacențe și care conduce la un nod  $y$  aparținând clasei I-a sau a II-a.
    - Cu alte cuvinte  $y$  este un nod pentru care procesul de vizitare s-a terminat (colorat în **negră**) sau care se află în stiva asociată traversării (colorat în **gri**).
    - Arcul de return  $(x, y)$  indică de fapt un **nod strămoș** al nodului  $x$ .

#### **10.4.1.3. Căutare "în adâncime" în grafuri reprezentate prin matrici de adiacențe**

- Procedura care implementează **căutarea în adâncime** în grafuri reprezentate prin **matrici de adiacențe** apare în secvența [10.4.1.3.a].
  - Traversarea listei de adiacențe a unui nod din **structura de adiacențe**, se transformă în parcurgerea **liniei** corespunzătoare nodului din **matricea de adiacențe**, căutând valori adevărate (care marchează arce).

- Şi în acest caz, selecţia unui arc care conduce la un nod **nevizitat** este urmată de un **apel recursiv** al procedurii de căutare pentru nodul respectiv.
  - Datorită modului diferit de reprezentare a grafului, arcele conectate la noduri sunt examineate într-o altă ordine, motiv pentru care **arborii de căutare în adâncime** care alcătuiesc pădurea corespunzătoare grafului diferă ca formă.

*/\*Căutare "în adâncime" în grafuri reprezentate prin matrici de adiacențe - varianta pseudocod\*/*

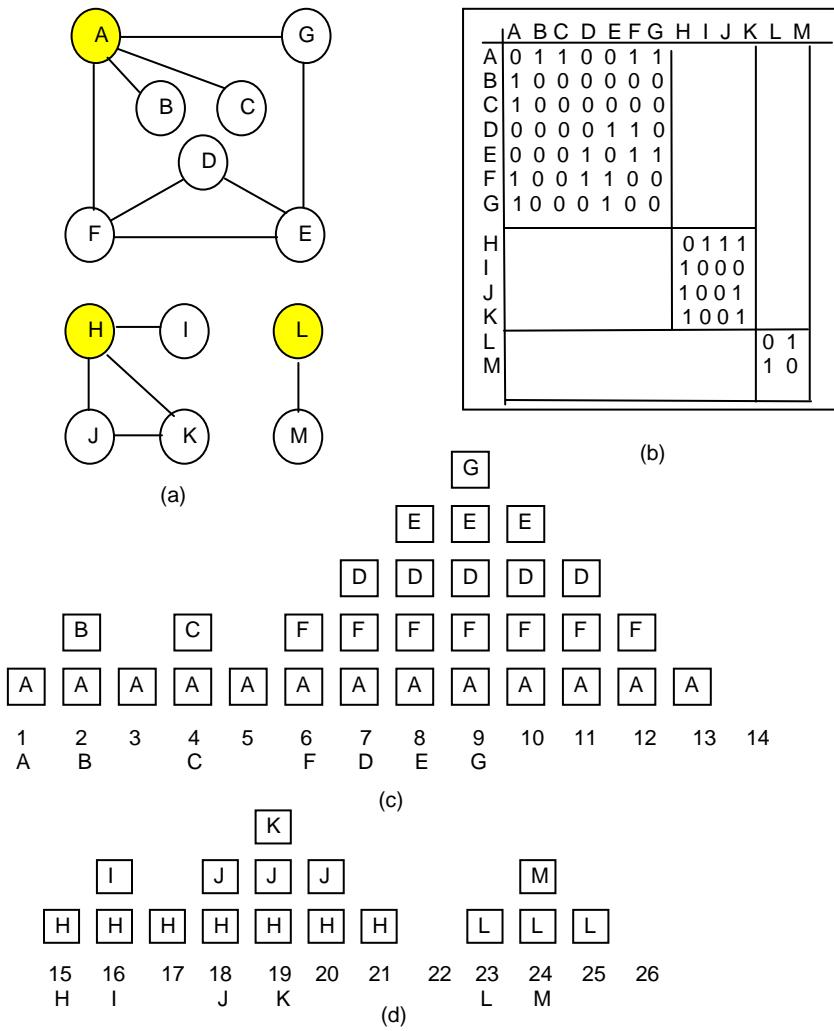
```

boolean A[numarNoduri]; /*matricea de adiacențe*/
int marc[numarNoduri]; /*tabloul marc pentru evidența
nodurilor*/
int id; /*contor de noduri*/

procedure CautaInAdincime1(int x);
    int t;

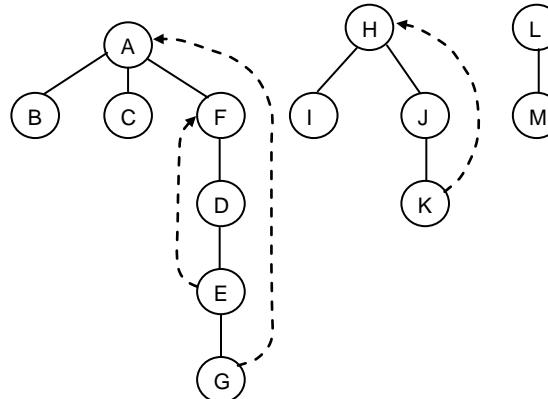
    id= id+1; marc[x]= id;
    *scrie(x); /*10.4.1.3.a*/
    pentru (t=1 la N)
        daca (A[x,t])
            daca (marc[t]==0)
                CautaInAdincime1(t);
/*CautaInAdincime1*/

```



**Fig.10.4.1.3.a.** Căutare în adâncime în grafuri reprezentate prin matrice de adiacențe

- Pentru graful din figura 10.4.1.3.a (a), a cărui matrice de adiacențe apare în aceeași figură (b), evoluția stivei asociate traversării apare în (c) și (d) iar arborii de căutare în adâncime corespunzători apar în fig. 10.4.1.3.b.



**Fig.10.4.1.3.b.** Pădure de arbori de căutare în adâncime în grafuri reprezentate prin matrice de adiacențe

- Se observă unele diferențe față de pădurea de arbori reprezentată în fig. 10.4.2.1.b. corespunzătoare aceluiași graf.
  - Prin aceasta se subliniază faptul că **o pădure de arbori de căutare în adâncime** nu este altceva decât o altă manieră de reprezentare a unui graf, a cărei alcătuire particulară depinde de:
    - (1) Metoda de traversare a grafului.
    - (2) Reprezentarea internă utilizată pentru graf.
  - Din punct de vedere al **eficienței**, căutarea în adâncime în grafuri reprezentate prin matrici de adiacențe necesită un timp proporțional cu  $O(n^2)$ .
    - Acest lucru este evident întrucât în procesul de traversare este verificat fiecare element al matricei de adiacențe.
  - Căutarea în adâncime rezolvă unele **probleme fundamentale** ale prelucrării grafurilor.
    - (1) Deoarece procedura de parcursare a unui graf se bazează pe traversarea pe rând a componentelor sale conexe, **numărul componentelor conexe** ale unui graf poate fi determinat simplu contorizând numărul de apeluri ale procedurii **CăutăInAdâncime** efectuat din ultima linie a procedurii **Traversare1**.
    - (2) Căutarea în adâncime permite verificarea simplă a **existenței ciclurilor** într-un graf.
      - Astfel, un graf conține un **ciclu**, dacă și numai dacă procedura **CăutăInAdâncime** descoperă o valoare diferită de zero în tabloul **marc**.
      - Această înseamnă că se parcurge un arc care conduce la un nod care a mai fost vizitat, deci graful conține un **ciclu**.
      - În cazul grafurilor **neorientate** trebuie însă să se țină cont de reprezentarea dublă a fiecărui arc, care poate produce confuzii.

- La reprezentarea grafurilor prin păduri de arbori de căutare, **liniile punctate** sunt acele care închid **ciclurile**.

#### 10.4.1.4. Căutare "în adâncime" nerecursivă

- Este cunoscut faptul că orice algoritm **recursiv** poate fi transformat într-un algoritm echivalent **iterativ**.
  - Tehnica realizării acestei transformări se bazează pe definirea și utilizarea de către programator a unei structuri de date **stivă** [Cr00].
- În secvența [10.4.1.4.a] apare echivalentul iterativ al algoritmului de căutare în adâncime bazat pe această tehnică.
- Graful se consideră reprezentat prin **structuri de adiacențe** implementate cu ajutorul **listelor înlățuite simple**.

---

*/\*Traversarea prin CA a grafurilor reprezentate prin SA implementate cu ajutorul listelor înlățuite simple - se utilizează TDA Stiva - Varianta iterativă\*/*

```

const maxN = 100;

typedef struct Nod * Ref_Tip_Nod;

/*structura unui nod*/
typedef struct Nod
{
    int nume;
    Ref_Tip_Nod urm;
} Tip_Nod;

/*structura de adiacențe*/
Ref_Tip_Nod StrAdj[maxN];

int id; /*contor noduri*/
int x; /*nod curent*/
int marc[maxN]; /*tablou evidență noduri*/

Tip_Stiva s; /*stiva*/

void CautaInAdincimeNerecursiv(int x)
{
    Ref_Tip_Nod t;

    push(x,s); /*amorsare proces*/
    do
    {
        x= varfSt(s); pop(s);
        id= id + 1; marc[x]= id; /*colorare în negru*/
        *scrie(x);
        t= Stradj[x];
        while (t<>null)
        {
            if (marc[t^.nume]==0) /*10.4.1.4.a*/
            {

```

```

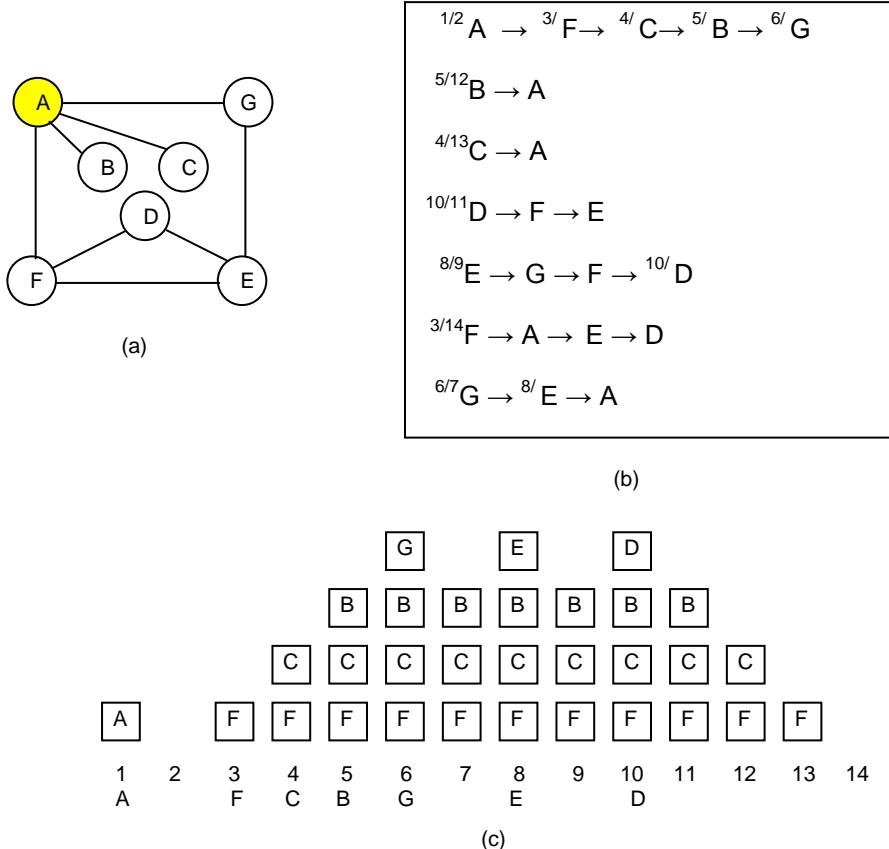
        push(t^.nume); /*nodul curent în stivă*/
        marc[t^.nume]=-1; /*colorare în gri*/
    }
    t= t^.urm;
}
while (stivid(s));
} /*CautăInAdincimeNerecursiv*/



void Traversare2
{
    id= 0; /*initializare contor noduri*/
    initializeaza(s); /*initializare stivă*/
    /*initializare - colorare în alb a tuturor nodurilor*/
    for (x=1;x<=N;x++)
        marc[x]= 0;
    /*traversare nerecursivă graf*/
    for (x=1;x<=N;x++)
        if (marc[x]==0)
        {
            CautăInAdincimeNerecursiv(x);
            *scrie_rand;
        };
} /*Traversare2*/
-----
```

- Nodurile care au fost atinse dar nu au fost încă cercetate sunt păstrate într-o structură de date stivă **s**.
  - Pentru gestionarea **stivei** sunt utilizati operatorii specifici definiți pe **TDA Stivă**: **initializează(s)**, **push(x,s)**, **pop(s)**, **stivid(s)** și **varfSt(s)** (Vol.1 &6.5.3.1).
- În timpul vizitării unui nod, pe măsura parcurgerii nodurilor sale adiacente, se introduc în stivă acele noduri care nu au fost încă vizitate și care nu sunt deja în stivă.
  - Introducerea în stivă a unui nod se marchează cu valoarea **-1** în tabloul **marc**, ceea ce echivalează cu colorarea lui în **gri**.
- În **varianta recursivă**, păstrarea evidenței nodurilor parțial vizitate este realizată în mod transparent pentru utilizator de către variabila locală **t** a procedurii recursive **CautăInAdâncime**.
- În **varianta iterativă** păstrarea evidenței nodurilor parțial vizitate se poate realiza memorând în stivă numele nodurilor parcuse de variabila **t**, respectiv valorile **t^.nume**.
- În plus, pentru a cunoaște cu precizie starea nodurilor, se utilizează tehnica de a marca în tabloul **marc** diferențele categorii de noduri cu **valori distințe**. Astfel:
  - (1) Se marchează cu valoarea **0** nodurile care **nu** au fost încă atinse (nodurile **albe**).
  - (2) Se marchează cu valoarea **-1** nodurile care au fost descoperite în procesul de vizitare și care au fost introduse în stivă (nodurile **gri**).
  - (3) Se marchează cu o valoare pozitivă corespunzătoare ordinii de parcurgere, nodurile care au fost extrase din stivă și pentru care vizitarea este în curs de desfașurare (nodurile **negre**).

- Se observă faptul că în acest caz, nodurile în curs de vizitare sunt marcate în negru, în momentul în care sunt extrase din stivă.
- Această metodă a fost aplicată în procedura **Traversare2** din secvența [10.4.1.4.a].
- În figura 10.4.1.4.a se prezintă modul în care se desfășoară **căutarea în adâncime în variantă nerecursivă** pentru graful din figură (a).
- Forma simplificată a reprezentării bazate pe structuri de adiacențe a grafului apare în (b), iar evoluția stivei asociate în (c).



**Fig.10.4.1.4.a.** Căutare în adâncime iterativă. Evoluția conținutului stivei.

- Presupunând că primul nod pentru care se apelează procedura **CautăÎnAdâncimeNerecursiv** este nodul A, acesta este introdus în stivă în cadrul procesului de amorsare a traversării, după care este imediat scos din stivă procedându-se la explorarea listei lui de adiacențe.
- În lista sa de adiacențe se descoperă nodurile F, C, B, și G care se introduc în stivă în această ordine.
- Terminându-se vizitarea listei nodului A, se revine și se extrage din stivă în vederea vizitării, ultimul nod introdus adică nodul G procedându-se la parcurgerea listei sale de adiacențe.
- Se ajunge astfel la nodul E nevizitat încă, care este și el introdus în stivă.
- Se parurge în continuare nodul A, deja vizitat, prezent în lista de adiacențe a nodului G, moment la care vizitarea listei lui G se încheie și se revine în bucla principală.

- Se trece la vizitarea nodului E care este extras din stivă.
- Procesul continuă în aceeași manieră, adică în lista de adiacențe a lui E este descoperit nodul D și a.m.d. până la golirea integrală a stivei de noduri.
- **Procesul integral** poate fi urmărit în figura 10.4.1.4.a (c), în care sunt marcate și momentele la care este introdus în stivă respectiv este extras din stivă fiecare nod.
  - Această informație este de altfel precizată și în reprezentarea schematică a structurii de adiacențe a grafului, unde spre exemplu notația  ${}^{3/14}F$  precizează faptul că nodul F este introdus în stivă la momentul 3 și este extras la momentul 14.
- Din implementare se observă că în cazul **variantei iterative**, ordinea de parcursere a nodurilor **nu este identică** cu cea rezultată din **varianta recursivă**.
  - Acest lucru poate fi însă evitat simplu, dacă în varianta iterativă, nodurile se introduc în stivă în ordinea inversă apariției lor în lista de adiacențe.
  - În mod inherent, vor rezulta și **arbori de căutare în adâncime** cu o structură specifică.
- De regulă însă, în marea majoritate a situațiilor reale, nu este necesar ca parcurserea să se raporteze strict la ordinea rezultată din modelul recursiv.
- De obicei, din punctul de vedere al scopului urmărit toate variantele de parcursere în adâncime sunt echivalente.

#### **10.4.1.5. Analiza căutării "în adâncime"**

- Se consideră un graf G cu  $a$  arce și  $n$  noduri implementat prin **structuri de adiacențe**.
- **Traversarea** sa în baza **tehnicii de căutare în adâncime** necesită un timp de calcul de ordinul  $O(n+a)$ .
  - Termenul  $n$  provine din parcurserea nodurilor și marcarea acestui fapt în tabloul `marc`.
  - Termenul  $a$  provine din parcurserea tuturor arcelor.
- Presupunând că  $n \ll a$ , timpul necesar căutării este  $O(a)$ .
  - Acest lucru rezultă din faptul că pentru nici un nod procedura de căutare **nu este** apelată mai mult decât odată, deoarece după primul apel **CăutăInAdâncime(x)**, `marc[x]` este asignat cu `id-ul` curent și în continuare nu se mai realizează nici un apel pentru nodul  $x$ .
  - În procesul de traversare sunt însă verificate **toate arcele grafului**.
  - În consecință timpul total petrecut cu parcurserea listelor de adiacențe este proporțional cu suma acestor liste deci este  $O(a)$ .

## 10.4.2. Traversarea grafurilor prin tehnica căutării "prin cuprindere" ("Breadth-First Search")

- O altă manieră sistematică de traversare a nodurilor unui graf o reprezintă **căutarea prin cuprindere** ("breadth first search").
- **Căutarea prin cuprindere** se bazează pe următoarea tehnică:
  - Pentru fiecare nod vizitat  $x$ , se caută în imediata sa vecinătate "**cuprinzând**" în vederea vizitării toate nodurile adiacente lui.
  - Pentru implementarea acestei tehnici de parcurs, în locul stivei din metoda de căutare anterioară, pentru reținerea nodurilor vizate se poate utiliza o **structură de date coadă**.
  - Schița de principiu a algoritmului de parcursare apare în secvența [10.4.2.a].
    - Se precizează faptul că s-a utilizat o structură de date **coadă** ( $Q$ ) asupra căreia acționează operatori specifici [Vol.1,&6.5.4.1, Cr00].

```
-----  
/*Căutare prin cuprindere. Schița de principiu. Varianta 1  
pseudocod - se utilizeaza TDA Coada*/
```

```
subprogram CautaPrinCuprindere(Tip_Nod x, Tip_Multime T)  
/*se parcurg toate nodurile adiacente lui x prin căutare  
prin cuprindere. Se construiește multimea T care include  
arborele de parcursare prin cuprindere*/  
  
Coada_De_Noduri Q;  
Tip_Nod x,y;  
int marc[maxN]; /*tabloul marc*/ /*[10.4.2.a]*/  
  
marc[x]= vizitat; /*marcheaza nodul x vizitat*/  
Adauga(x,Q); /*se introduce nodul de pornire în coadă*/  
cat timp (not vid(Q)) executa  
    x= Cap(Q); /*citeste în x nodul din capul cozii*/  
    Scoate(Q); /*scoate nodul din coadă*/  
    pentru (fiecare nod y adiacent lui x) executa  
        daca (marc[y] este nevizitat)  
            marc[y]= vizitat;  
            Adauga(y,Q); /*adaugă-l pe y în coadă*/  
            INSERTIE((x,y),T) /*inserează arcul (x,y) în  
                           arborele de parcursare asociat*/  
            □ /*daca*/  
            □ /*cat timp*/  
    /*CautaPrinCuprindere*/  
-----
```

- Algoritmul din secvența [10.4.2.a] inserează cu ajutorul operatorului **INSERTIE** arcele parcurse ale grafului într-o mulțime  $T$  care materializează traversarea și despre care se presupune că este inițial vidă.
- Se presupune de asemenea că tabloul  $marc$  este inițializat integral cu marca "nevizitat".
- Procedura lucrează pentru o singură componentă conexă.
  - Dacă graful **nu** este conex, procedura **CăutăPrinCuprindere** trebuie apelată pentru fiecare componentă conexă în parte.

- Se atrage atenția asupra faptului că în cazul căutării prin cuprindere, un nod trebuie marcat cu vizitat înaintea introducerii sale în coadă pentru a se evita plasarea sa de mai multe ori în această structură.

#### 10.4.2.1. Căutarea "prin cuprindere", varianta CLR

- **Căutarea prin cuprindere** este una dintre cele mai cunoscute metode de căutare, utilizate printre alții de către **Djikstra** și **Prim** în celebrii lor algoritmi [CLR92].
- Ca și în cazul căutării în adâncime, pentru a ține evidența procesului de căutare nodurile sunt colorate în alb, gri și negru.
- Toate nodurile sunt colorate inițial alb și ele devin mai târziu gri, apoi negre.
  - La prima descoperire a unui nod, acesta este colorat în gri.
  - La terminarea vizitării unui nod, acesta este colorat în negru.
  - Nodurile gri și negre sunt noduri deja descoperite în procesul de căutare, dar ele sunt diferențiate pentru a se asigura funcționarea corectă a căutării.
  - Nodurile gri care pot avea ca adiacenți și noduri albe, marchează frontieră dintre nodurile vizitate și cele nevizitate.
  - De fapt, nodul curent nu se colorează în negru când toate nodurile adiacente lui au fost deja vizitate.
- Își încasează căutările prin cuprindere se poate construi un **subgraf sp** al predecesorilor nodurilor vizitate.
  - Ori de câte ori un nod v este descoperit pentru prima oară în procesul de căutare la parcurgerea listei de adiacențe a nodului u, acest lucru se marchează prin  $sp[v] = u$ .
- După cum s-a mai precizat **subgraful predecesorilor** este de fapt un **arbore liber**, reprezentat printr-un tablou liniar în baza relației “indicator spre părinte”.
- Procedura **TraversarePrinCuprindere** din secvența [10.4.2.1.a] presupune graful  $G = (N, A)$  reprezentat prin **structuri de adiacențe**.
  - Culoarea curentă a fiecărui nod  $u \in N$  este memorată în tabloul `culoare[u]`.
  - Predecesorul nodului u, adică nodul în lista căruia a fost descoperit, este înregistrat în tabloul `sp[u]`.
  - Dacă u nu are predecesor (adică este nodul de pornire) se marchează acest lucru prin `sp[u] = null`.
- În cadrul procesului de **traversare prin cuprindere a grafului**, se poate calcula și **distanța** de la nodul de start la fiecare din nodurile grafului.
  - Distanța de la sursă la nodul curent u calculată de către algoritm este memorată în `d[u]`.
  - Unitatea de măsură a distanței este **numărul de arce traversate**.
- Algoritmul de traversare utilizează o **coadă FIFO** notată cu Q pentru a gestiona nodurile implicate în procesul de căutare, scop în care face uz de operatorii consacrați pentru această structură de date.

*/\*Căutarea prin cuprindere. Schița de principiu. Varianta 2  
pseudocod (Cormen, Leiserson, Rivest) - se utilizeaza TDA  
Coada\*/*

```
Tip_Coada Q; /*structura COADA*/  
  
CautaPrin Cuprindere(Tip_Graf G, Tip_Nod s)  
[1]   pentru (fiecare nod u ∈ N(G)-{s}) /*s este nodul de  
          start*/  
[2]     culoare[u]=alb;  
[3]     d[u]=∞;  
[4]     sp[u]=null;  
     □ /*pentru*/  
[5]     culoare[s]=gri; /*s este nodul de start*/  
[6]     d[s]=0;  
[7]     sp[s]=null;  
[8]     Initializeaza(Q); Adauga(s,Q);      /*10.4.2.1.a*/  
[9]     cat timp not vid(Q)do  
[10]       u=Cap(Q);  
[11]       pentru (fiecare v ∈ Adj[u])  
[12]         daca (culoare[v]=alb)  
[13]           culoare[v]=gri;  
[14]           d[v]=d[u]+1;  
[15]           sp[v]=u;  
[16]           Adauga(v,Q);  
           □ /*daca*/  
[17]         Scoate(Q);  
[18]         culoare[u]=negru;  
     □ /*cat timp*/  
-----
```

- **Funcționarea algoritmului.**

- Liniile 1- 4 inițializează structurile de date, adică:

- Toate nodurile u cu excepția nodului de start sunt marcate cu alb în tabloul culoare.
- Distanțele corespunzătoare tuturor nodurilor sunt setate pe  $\infty$  ( $d[u]=\infty$ ).
- Părintele fiecărui nod este inițializat cu nil ( $sp[u]=null$ ).
- Linia 5 marchează nodul s furnizat ca parametru al procedurii de căutare cu gri, el fiind considerat **sursa** (originea) procesului de căutare.
- Liniile 6 și 7 inițializează  $d[s]$  pe zero și  $sp[s]$  cu **null**.
- Linia 8 inițializează coada Q și îl adaugă pe s în coadă.
  - De altfel coada Q va conține numai noduri colorate în **gri**.
- Bucla principală a programului apare între liniile 9-18 și ea iterează atâtă vreme cât există noduri (gri) în coadă.
  - Nodurile gri din coadă sunt noduri descoperite în procesul de căutare în liste de adiacențe care nu au fost încă epuizate.
  - Linia 10 furnizează nodul gri u aflat în capul cozii Q.
  - Bucla **for** (liniile 11-16) parcurge fiecare nod v al listei de adiacențe a lui u.

- Dacă v este alb, el nu a fost încă descoperit, ca atare algoritmul îl descoperă executând liniile 13-16 adică:
  - Nodul v este colorat în gri.
  - Distanța  $d[v]$  este setată pe  $d[u] + 1$  indicînd creșterea acesteia cu o unitate în raport cu părintele nodului.
  - u este memorat ca și părinte al lui v.
  - În final v este adăugat în coada Q la sfârșitul acesteia.
- După ce toate nodurile listei de adiacențe a lui u au fost examineate, u este extras din coada Q și colorat în negru (liniile 17-18).
- **Analiza performanței.**
- Se analizează **timpul de execuție** al algoritmului pentru un graf  $G = (N, A)$ .
- După inițializare **nici un nod** nu mai este ulterior colorat în alb, ca atare testul din linia 12 asigură faptul că fiecare nod este adăugat cozii cel mult odată și este scos din coadă **cel mult odată**.
- Operațiile **Adauga** și **Scoate** din coadă consumă un timp  $O(1)$ , ca atare timpul total dedicat operării cozii Q este  $O(N)$ .
- Deoarece lista fiecărui nod este scanată integral înaintea scoaterii nodului din coadă, această operație se realizează cel mult odată pentru fiecare nod.
  - Întrucât suma lungimilor tuturor listelor de adiacență este  $O(A)$ , timpul total necesar pentru scanarea listelor de adiacențe este  $O(A)$ .
- Regia inițializării este  $O(N)$  deci timpul total de execuție al procedurii **CautăPrinCuprindere** este  $O(N+A)$ .
- În concluzie, căutarea prin cuprindere necesită un timp de execuție liniar cu numărul de noduri și cu mărimea listelor de adiacențe ale reprezentării grafului G.

#### **10.4.2.2. Căutare "prin cuprindere" în grafuri reprezentate prin structuri de adiacențe**

- În secvența [10.4.2.2.a] apare un exemplu de procedură care parcurge un graf în baza tehnicii de **parcurs prin cuprindere**, utilizând o **structură de date coadă**.
- Graful se consideră reprezentat cu ajutorul **structurilor de adiacențe** implementate cu **liste înlănuite simple**.

---

**{Traversarea prin cuprindere a grafurilor reprezentate prin SA implementate cu ajutorul listelor înlănuite simple - se utilizeaza TDA Coada - varianta pseudocod}**

```

int x;
int id; /*contor noduri*/
int mark[maxN]; /*tablou evidență noduri*/
Tip_Coada Q;

subprogram CautaPrinCuprindere(int x)
  Ref_Tip_Nod t;

```

```

Adauga(x,Q); /*amorsare proces de căutare*/
repetă
    x= Cap(Q); Scoate(Q);
    id= id+1; marc[x]= id; /*colorare în negru*/
    *scrie(x);
    t= Stradj[x];
    cat timp (t<>null)
        daca (marc[t->nume]=0) /*[10.4.2.2.a]*/
            | Adauga(t->nume,Q);
            | marc[t->nume]=-1 /*colorare în gri*/
            |   □ /*dacă*/
            |   t= t->urm;
            |   □ /*cât timp*/
    pana cand Vid(Q)
    □ /*repetă*/
/*CautaPrinCuprindere*/

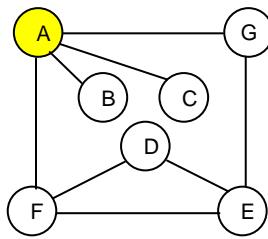
```

```
program ParcurgerePrinCuprindere;
```

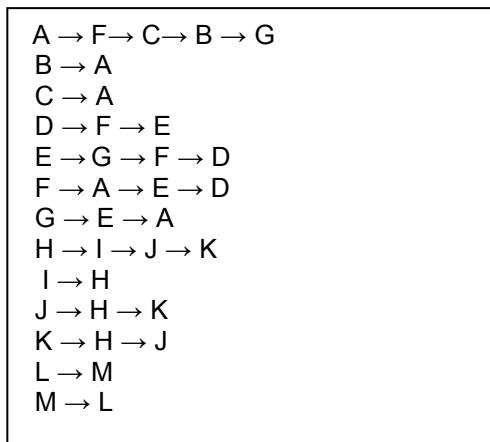
```

id=0;
Initializeaza(Q);
/*inițializare tablou de noduri*/
pentru (x= 1 la N)
    marc[x]= 0;
/*căutare prin cuprindere în graf*/
pentru (x= 1 la N)
    daca (marc[x] este 0)
        | CautaPrinCuprindere(x);
        | *scrie_rând (writeln);
        |   □ /*daca*/
/*ParcurgerePrinCuprindere*/
-----
```

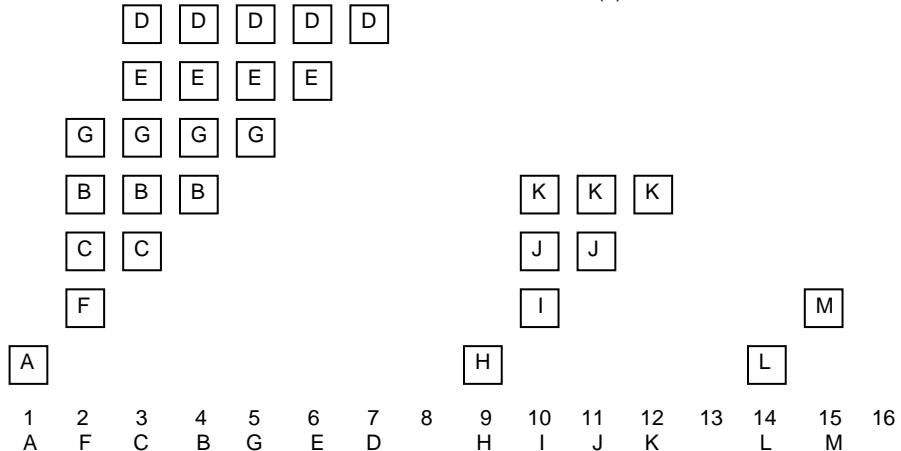
- În figura 10.4.4.2.a se prezintă un exemplu de traversare prin cuprindere a unui graf reprezentat prin structuri de adiacențe.



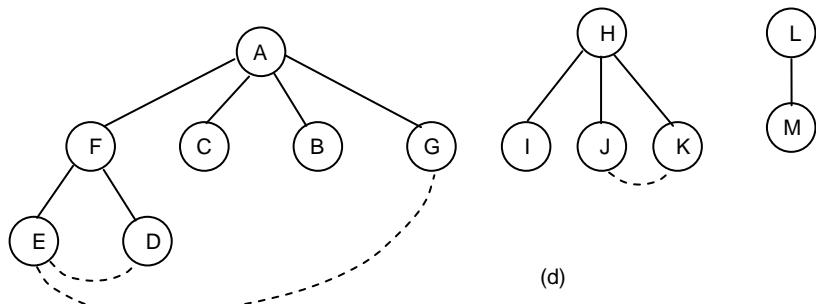
(a)



(b)



(c)



(d)

**Fig.10.4.2.2.a.** Traversarea prin cuprindere a unui graf

- Se consideră graful din figura 10.4.2.2.a (a) reprezentat prin **structura de adiacențe** din aceeași figură (b).
  - Ordinea de parcurgere a arcelor va fi următoarea: AF, AC, AB, AG, FA, FE, FD, CA, BA, GE, GA, DF, DE, EG, EF, ED, HI, EH, JH, HK, IH, JH, JK, KH, KJ, IM, MI
  - Evoluția conținutului cozii pe parcursul traversării și ordinea în care sunt traversate nodurile apar în fig. 10.4.2.2.a (c).
- În mod similar cu traversarea bazată pe căutarea în adâncime, se poate construi o **pădure de arbori de acoperire ("spanning trees")** specifică **căutării prin cuprindere**.
  - În acest caz, un arc  $(x, y)$  se consideră **ramură a arborelui de căutare prin cuprindere**, dacă în bucla **pentru** a secvenței [10.4.2.a], respectiv **căt timp** a secvenței [10.4.2.2.a] nodul  $y$ , respectiv  $t \rightarrow \text{urm}$ , este vizitat întâia dată venind dinspre nodul  $x$ .

- În cazul **căutării prin cuprindere** în grafuri neorientate, fiecare arc al grafului care **nu** este o ramură a arborelui de căutare prin cuprindere, este un **arc de trecere** care conectează două noduri dintre care niciunul **nu** este strămoșul celuilalt.
- Arborii de căutare aferenți parcurgerii grafului (a) din figura fig. 10.4.2.2.a apar în aceeași figură (d).
  - După cum s-a precizat, acești arbori cuprind acele arce care conduc întâia dată la un anumit nod.
- Utilizând arbori de căutare prin cuprindere, verificarea existenței **ciclurilor** în cadrul unui graf, poate fi realizată în  $O(n)$  unități de timp, indiferent de numărul de arce.
  - După cum s-a precizat în §10.1, un graf cu  $n$  noduri și  $n$  sau mai multe arce, trebuie să aibă cel puțin un **ciclu**.
  - Cu toate acestea și un graf cu  $n$  noduri și  $n-1$  sau mai puține arce poate avea cicluri, dacă conține două sau mai multe componente conexe.
- O modalitate sigură de a determina **ciclurile** unui graf este aceea de a-i construi **pădurea arborilor de căutare prin cuprindere**.
  - Fiecare **arc de trecere**, reprezentat punctat în figură, închide un **ciclu simplu** care cuprinde arcele arborelui care conectează cele două noduri, prin cel mai apropiat strămoș comun al lor.

#### 10.4.2.3. Analiza căutării "prin cuprindere"

- Din punctul de vedere al **timpului de execuție**, complexitatea algoritmului de traversare (căutare) prin cuprindere este aceeași ca și la căutarea în adâncime.
- Fiecare nod al grafului este plasat în coadă o singură dată, astfel corpul buclei **cat timp** (secvența [10.4.2.2.a]) se execută o singură dată pentru fiecare nod.
- Fiecare arc  $(x, y)$  este examinat de două ori, odată pentru  $x$  și odată pentru  $y$ .
- Astfel, dacă graful are  $n$  noduri și  $a$  arce, timpul de execuție al algoritmului de căutare prin cuprindere este  $O(\max(n, a))$  dacă utilizăm reprezentarea prin structuri de adiacență.
  - Deoarece în general  $a \geq n$ , de regulă se va considera timpul de execuție al căutării prin cuprindere  $O(a)$ , ca și în cazul căutării în adâncime.

#### 10.4.3. Comparație între tehnicele fundamentale de traversare a grafurilor

- **Traversarea** unui graf, indiferent de metoda utilizată, are principal un **caracter unitar**, particularizarea rezultând din **structura de date** utilizată drept suport în implementare.
- În ambele tehnici de traversare, nodurile pot fi divizate în trei **clase**:
  - (1) Clasa "**arbore**" – care cuprinde nodurile care au fost extrase din structura de date utilizată în traversare.
    - Sunt nodurile deja vizitate, adică cele colorate în **negru**;
  - (2) Clasa "**vecinătate**" – care cuprinde nodurile adiacente nodurilor traversate sau în curs de traversare.

- Aceste noduri au fost luate în considerare dar nu au fost încă vizitate și se găsesc introduse în **structura de date** utilizată în traversare.
- Sunt nodurile colorate în **gri**;
- (3) Clasa "**neîntâlnite**" - care cuprinde nodurile la care nu s-a ajuns până la momentul considerat.
  - Sunt nodurile colorate în **alb**.
- Un **arbore de căutare** ia naștere conectând fiecare nod al grafului, cu nodul care a cauzat introducerea sa în **structura de date** utilizată în parcurgere.
- Pentru a parcurge în mod sistematic o **componentă conexă** a unui graf, deci pentru a implementa o procedură "parcurge":
  - (1) Se introduce un nod oarecare (nodul de pornire) al componentei, în clasa "**vecinătate**" și toate celelalte noduri în clasa "**neîntâlnite**".
  - (2) În continuare, până la vizitarea **tuturor nodurilor grafului** se aplică următorul procedeu:
    - Se selectează un nod – fie acesta  $x$  – din clasa "**vecinătate**" și se mută în clasa "**arbore**".
    - Se trec în clasa "**vecinătate**" toate nodurile din clasa "**neîntâlnite**" care sunt adiacente lui  $x$ .
    - Se reia procedeul de la (2).
- **Metodele de parcurgere a grafurilor** se diferențiază după maniera în care **sunt alese nodurile** care se trec din clasa "**vecinătate**" în clasa "**arbore**".
  - (1) La parcurgerea "**în adâncime**" se alege din vecinătate nodul **cel mai recent întâlnit** (ultimul întâlnit) ceea ce corespunde cu utilizarea unei **stive** pentru păstrarea nodurilor din clasa "**vecinătate**".
  - (2) La parcurgerea "**prin cuprindere**" se alege nodul **cel mai devreme întâlnit** (primul întâlnit) ceea ce presupune păstrarea într-o **coadă** a nodurilor din clasa "**vecinătate**".
  - (3) Se pot utiliza în acest scop și alte structuri de date, spre exemplu **cozi bazate pe prioritate** în cazul grafurilor ponderate.
- Contrastul dintre primele două metode de parcurgere este și mai evident în cazul **grafurilor de mari dimensiuni**.
  - Căutarea "**în adâncime**" se avântă în profunzime de-a lungul arcelor grafului memorând într-o stivă punctele de ramificație.
  - Căutarea "**prin cuprindere**" mătură prin extindere radială graful, memorând într-o coadă frontieră locurilor deja vizitate.
  - La căutarea "**în adâncime**" graful este explorat căutând noi noduri, cât mai departe de punctul de plecare și luând în considerare noduri mai apropiate numai în situația în care nu se poate înainta mai departe.
  - Căutarea "**prin cuprindere**" acoperă complet zona din jurul punctului de plecare mergând mai departe numai când tot ceea ce este în imediata sa apropiere a fost parcurs.
  - Este însă evident faptul că în ambele situații, ordinea efectivă de parcurgere a nodurilor depinde:

- (1) Pe de o parte de **structura de date** utilizată pentru implementarea grafului.
- (2) Pe altă parte de **ordinea** în care sunt introduse inițial nodurile în această structură.
- În afara diferențelor rezultate din manierele de operare ale celor două metode, se remarcă diferențe fundamentale în ceea ce privește **implementarea**.
  - Căutarea “**în adâncime**” poate fi simplu exprimată în manieră **recursivă** ea bazându-se pe o structură de date **stivă**.
  - Căutarea “**prin cuprindere**” admite o implementare simplă **iterativă** fiind bazată pe o structură de date **coadă**.
- Aceasta este încă un exemplu care evidențiază cu pregnanță legătura strânsă care există între o anume **structură de date** și **algoritmul** care o prelucreză.

## 10.5. Aplicații ale traversării grafurilor

- În cadrul paragrafului de față se prezintă câteva dintre aplicațiile tehnicielor de traversare a grafurilor.
- Se au în vedere în acest context:
  - Arborii de acoperire.
  - Tehnici de determinare a arborilor de acoperire.
  - Unele aspecte legate de conexitate.
  - Punctele de articulație ale unui graf.
  - Componentele biconexe ale unui graf.

### 10.5.1. Arbori de acoperire ("Spanning Trees"). Determinarea arborilor de acoperire

- Una din aplicațiile traversării grafurilor o reprezintă determinarea unui **arbore de acoperire** ("spanning tree") pentru un graf.
  - După cum s-a mai precizat, orice **graf conex aciclic** este un **arbore liber**.
- Un **arbore de acoperire** al unui graf  $G$ , este un **arbore liber** construit pornind de la anumite arce ale lui  $G$ , într-o astfel de manieră încât el să conțină toate nodurile lui  $G$ .
  - Un alt mod de a preciza **arborii de acoperire** este acela de a-i considera drept cea mai mică colecție de arce aparținând unui graf, care permite comunicația între ori care două noduri ale grafului (&10.1.).
- În prezentarea tehnicielor fundamentale de traversare a grafurilor s-a făcut precizarea că fiecare traversări a unui graf i se poate asocia un **arbore de acoperire**, respectiv o **pădure** ("forest") de astfel de arbori dacă graful conține mai multe componente conexe. Există de fapt, câte un **arbore** pentru fiecare **componentă**.
  - Arborii de acoperire sunt denumiți și **arbori de căutare în adâncime** respectiv **arbori de căutare prin cuprindere** în dependență de metoda de traversare utilizată în determinarea lor.

- În continuare se vor prezenta unele **tehnici de determinare** a arborilor de acoperire pentru ambele tipuri de traversări.
- Înainte de prezentarea efectivă a acestor tehnici, se subliniază faptul că o altă posibilitate de evidențiere a arborilor de acoperire o reprezintă construcția **subgrafului de precedență** sp prezentat în variantele Cormen, Leiserson și Rivest de traversare a grafurilor atât pentru **căutarea în adâncime** cât și pentru **căutarea prin cuprindere**.
  - După cum s-a precizat subgraful sp este un **arbore liber** reprezentat prin tehnica "indicator spre părinte" care materializează de fapt **arborele de acoperire** specific.

#### 10.5.1.1. Determinarea unui arbore de căutare "în adâncime"

- Aspectele teoretice ale acestei tehnici au fost precizate în &10.4.1.
- În continuare se prezintă **algoritmul** care determină un **arbore de căutare în adâncime** pornind de la traversarea grafurilor prin tehnica **căutării în adâncime**.
- **Tehnica** este simplă:
  - Ori de câte ori se vizitează un nod nevizitat încă, se marchează arcul care leagă ultimul nod curent de noul nod.
  - Toate arcele astfel marcate sunt **arce de arbore** ale arborelui de căutare în adâncime.
  - Restul arcelor, adică cele nemarcate, sunt **arce de return** care se trasează punctat în reprezentarea arborelui.
- Procedura care implementează această tehnică pentru căutarea în adâncime, se numește **ArboreDeAcoperireCA** și apare în secvența [10.5.1.1.a].
- Se fac următoarele precizări:
  - 1) Graful se consideră reprezentat prin **structuri de adiacențe** implementate cu ajutorul structurii de date multilistă **varianta Decker**, prezentată în studiul de caz 3 din paragraful &10.3.2.3.
  - 2) Pentru marcarea nodurilor vizitate, structura unui nod implementat de articolul **TipCelulăListNod** din secvența [10.3.2.3.a], se suplimentează cu câmpul **marc** de tip boolean. La inițializarea procedurii câmpul se pune pe "false" pentru toate nodurile grafului, urmând a deveni "true" în momentul vizitării.
  - 3) În vederea marcării arcelor **arborelui de căutare în adâncime**, structura unei celule aparținând unei liste de adiacențe, respectiv articolul **TipCelulăListArc** din aceeași secvență [10.3.2.3.a], se completează cu câmpul **marcArc** de tip boolean, a cărui valoare inițial falsă devine adevărată dacă **arcul este de arbore**, respectiv rămâne falsă dacă **arcul este de return**.
- Se face de asemenea precizarea că **arborele de căutare în adâncime** corespunzător unui graf **nu este unic**.
  - Astfel pentru un același graf pot fi obținuți diferiți arbori de căutare în adâncime funcție de:
    - (1) Modul și ordinea în care se crează structura de adiacențe aferentă.

- (2) Nodul cu care se începe parcurgerea.
- (3) Ordinea în care sunt parcuse nodurile adiacente.

---

{Determinarea unui arbore de acoperire minim pentru CA în grafuri reprezentate prin SA implementate cu multiliste (varianta Decker) - implementare PASCAL}

```

procedure ArboreDeAcoperireCA(g: TipGraf);

var p: RefTipPozitie;
    e: RefListArc;

procedure ConstructieACA(p: TipPozitie);

var arcCurent: RefListArc;
    q: TipPozitie;
begin
    p^.marc:= true; {marcarea cu vizitat a nodului curent}
    arcCurent:= p^.incep; {lista de adiacențe}
    while arcCurent <> nil do
        begin
            q:= arcCurent^.nod;
            if not q^.marc then [10.5.1.1.a]
                begin
                    arcCurent^.marcArb:= true;
                    {arcul arborelui CA poate fi afișat; de
                     asemenea poate fi marcată și cealaltă copie a
                     lui arcCurent, deoarece în această
                     implementare fiecare arc este reprezentat de
                     două ori}
                    ConstructieACA(q)
                end;
            arcCurent:= arcCurent^.urm
        end
    end; {ConstructieCA}

begin {ArboreDeAcoperireCA}
    p:= g; {p - pointer pentru lista nodurilor}
    while p <> nil do {marcarea cu nevizitat a nodurilor}
        begin
            p^.marc:= false;
            e:= p^.incep; {e-pointer pentru lista de adiacente}
            while e <> nil do {marcarea cu nevizitat a arcelor}
                begin
                    e^.marcArb:= false;
                    e:= e^.urm
                end;
            p:= p^.urm
        end;
    p:= g;
    ConstructieACA(p)
end; {ArboreDeAcoperireCA}

```

---

#### 10.5.1.2. Determinarea unui arbore de căutare „prin cuprindere”

- Într-o manieră asemănătoare, pornind de la algoritmul traversării prin cuprindere a unui graf se poate concepe procedura **ArboreDeAcoperireCC** care determină un **arbore de căutare prin cuprindere** (secvența [10.5.1.2.a]).
- Toate precizările făcute anterior rămânând valabile și pentru această procedură.
- În plus se face precizarea că pentru gestionarea **structurii de date coadă** au fost utilizați **operatori specifici** [Cr00].

---

{Determinarea unui arbore de acoperire minim pentru CC în grafuri reprezentate prin SA implementate cu multiliste (varianta Decker) - se utilizează TDA Coadă - implementare PASCAL}

```
procedure ArboreDeAcoperireCC(g: TipGraf);  
  
var C: TipCoada; {de poziții}  
    n,m,e: RefTipPozitie;  
    arcCurent: RefListArc;  
begin  
    n:= g; {n - pointer pentru lista nodurilor}  
    while n <> nil do {marcarea cu „nevizitat” a nodurilor}  
        begin  
            n^.marc:= false;  
            e:= n^.incep; {e-pointer pentru lista de adiacente}  
            while e <> nil do {marcarea cu nevizitat a arcelor}  
                begin  
                    e^.marcArb:= false;  
                    e:= e^.urm  
                end;  
            n:= n^.urm  
        end;  
    Initializeaza(C); {initializare coadă}  
    n:= g; {n este nodul de pornire}  
    Adauga(n,C); {amorsare parcurgere}  
    n^.marc:= true;  
    while not Vid(C) do  
        begin  
            n:= Cap(C);  
            Scoate(C);  
            arcCurent:= n^.incep; {[10.5.1.2.a]}  
            while arcCurent <> nil do  
                begin  
                    m:= arcCurent^.nod;  
                    if not m^.marc then  
                        begin  
                            Adauga(m,C);  
                            arcCurent^.marcArc:= true  
                            {se poate afișa arcul de arbore;  
                             se poate marca și cealaltă copie a lui  
                             arcCurent}  
                        end;  
                    arcCurent:= arcCurent^.urm  
                end  
        end  
end
```

```
end; {ArboreDeAcoperireCC}
```

---

## 10.5.2. Grafuri și conexiuni

- În cadrul grafurilor, noțiunea de **conexiune** joacă un rol central și ea este strâns legată de noțiunea de **arc**, respectiv de noțiunea de **drum**.
- În paragraful §10.1. se definesc pornind de la aceste elemente noțiunile de **graf conex**, respectiv de **componentă conexă** a unui graf.
  - (1) Se reamintește că un **graf conex** este acela în care pentru fiecare nod al său există un drum spre oricare alt nod al grafului.
  - (2) Un graf care **nu este conex** este format din  **componente conexe**.
- Deoarece în anumite situații prelucrarea grafurilor este simplificată dacă grafurile sunt partajate în componentele lor conexe, în continuare se vor prezenta unele **tehnici de determinare a componentelor conexe** ale unui graf.
- De asemenea sunt prezentate noțiunile de **graf biconex** și **punct de articulație** precum și tehnicele determinării acestora.

### 10.5.2.1. Determinarea componentelor conexe ale unui graf

- Oricare dintre **metodele de traversare** a grafurilor prezentate în paragraful anterior poate fi utilizată pentru determinarea componentelor conexe ale unui graf.
  - Acest lucru este posibil deoarece toate metodele de traversare se bazează pe aceeași **strategie generală** a vizitării tuturor nodurilor dintr-o componentă conexă începând de la o altă componentă.
- O manieră simplă de a vizualiza componente conexe este aceea de a modifica **procedura recursivă de traversare prin căutare în adâncime** spre exemplu aşa cum se sugerează în procedura **ComponenteConexe** secvența [10.5.2.1.a].

---

```
/*Determinarea componentelor conexe ale unui graf
reprezentat prin SA implementate cu ajutorul listelor
înlăntuite simple - varianta pseudocod*/

int id; /*contor noduri*/
int marc[maxN]; /*tablou evidență noduri*/

subprogram Componenta(int x);
/*parcurge componentă conexă căreia îi aparține nodul x*/
ref_tip_nod t;

id= id+1; marc[x]= id; /*marchează nodul vizitat*/
*scrie(t->nume);
t= StrAdj[x]; /*t - pointer în lista de adiacențe */
cât timp (t<>null)/*parcurge lista de adiacențe alui x*/
daca (marc[t->nume] este 0)
    Componenta(t->nume);
    t= t->urm;
    /*cât timp*/
```

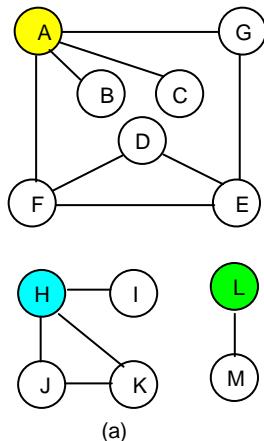
```

/*Componenta*/
/*10.5.2.1.a*/

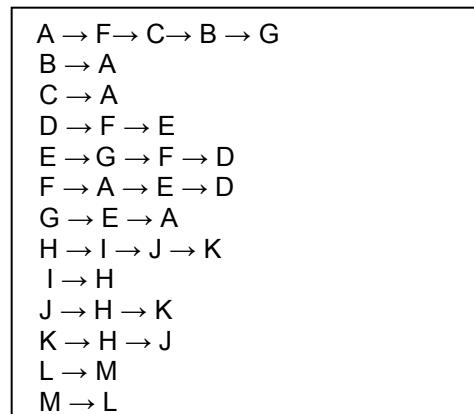
subprogram ComponenteConexe;
/*apeleză componentele conexe ale grafului*/
    int x;

    id= 0;
    pentru (x= 1 la N)
        marc[x]= 0;
    pentru (x= 1 la N)
        dacă marc[x]=0
            Componenta(x);
            /*scrie_rand (writeln); /*afisează componenta curentă*/
            □ /*dacă*/
/*ComponenteConexe*/
-----
```

- Alte variante ale procedurii de căutare în adâncime cum ar fi cea aplicată grafurilor reprezentate prin matrice de adiacențe sau cea nerecursivă, precum și căutarea prin cuprindere, modificate în aceeași manieră, vor evidenția aceleasi componente conexe, dar nodurile vor fi vizualizate în altă ordine.
- Funcție de natura prelucrărilor ulterioare ale grafului pot fi utilizate și alte metode.
- Astfel spre **exemplu**, se poate introduce tabloul **invmarc** (“inversul” tabloului **marc**) care se completează ori de câte ori se completează tabloul **marc**, respectiv când **marc[x]= id** se asignează și **invmarc[id]= x**.
  - În tabloul **invmarc** intrarea **id** conține indexul celui de-al **id**-lea nod vizitat. În consecință, nodurile aparținând aceleasi componente conexe sunt **contigue** adică ocupă poziții alăturate.
  - În acest tablou, indexul care precizează **o nouă componentă conexă**, este dat de valoarea lui **id** din momentul în care procedura **Componenta** este apelată din procedura **ComponenteConexe**.
  - Aceste valori pot fi memorate separat sau pot fi marcate în tabloul **invmarc**, spre exemplu primind valori **negative** (opuse).
- În fig.10.5.2.1.a (c) se prezintă valorile pe care le conțin aceste tablouri, în urma execuției procedurii **ComponenteConexe** asupra grafului (a), reprezentat prin structura de adiacențe din aceeași figură (b).



(a)



(b)

x	1	2	3	4	5	6	7	8	9	10	11	12	13
nume[x]	A	B	C	D	E	F	G	H	I	J	K	L	M
marc[x]	1	7	6	5	3	2	4	8	9	10	11	12	13
invmarc[x]	-1	6	5	7	4	3	2	-8	9	10	11	-12	13
ordine	A	F	E	G	D	C	B	H	I	J	K	L	M

(c)

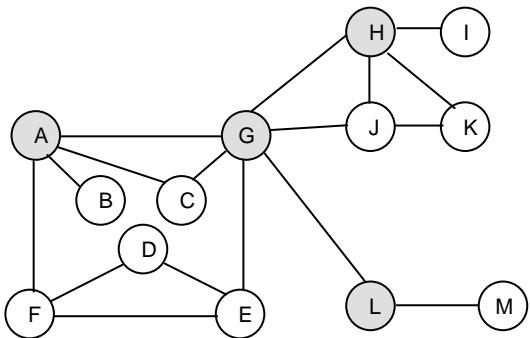
**Fig. 10.5.2.1.a.** Evidențierea componentelor conexe ale unui graf

- În activitatea practică, este deosebit de avantajoasă utilizarea unor astfel de tehnici de partajare a grafurilor în componente conexe în vederea prelucrării ulterioare a acestor componente în cadrul unor algoritmi complecși.
  - Astfel, algoritmii de complexitate mai ridicată sunt eliberați de detaliile prelucrării unor grafuri care nu sunt conexe și în consecință devin mai simpli.

## 10.5.2.2. Puncte de articulație și componente biconexe

- În anumite situații este util a se prevedea **mai mult decât un singur drum** între nodurile unui graf cu scopul de a rezolva posibile căderi ale unor puncte de contact (noduri).
  - Astfel, spre exemplu în rețeaua feroviară există mai multe posibilități de a ajunge într-un anumit loc.
  - De asemenea într-un circuit integrat liniile principale de comunicație sunt adesea dublate astfel încât circuitul rămâne încă în funcțiune dacă vreo componentă cade.
- Un **punct de articulație** al unui **graf conex** este un **nod**, care dacă este suprimat, **graful se rupe** în două sau mai multe bucăți.
- Un graf care **nu** conține **puncte de articulație** se numește **graf biconex**.
  - Într-un **graf biconex**, fiecare pereche de noduri este conectată prin cel puțin **două** drumuri distincte.
  - Un graf care **nu** este **biconex** se divide în **componente biconexe**, acestea fiind mulțimi de noduri mutual accesibile via două drumuri distincte.

- În fig. 10.5.2.2.a apare un graf conex care însă nu este biconex.
- **Punctele de articulație** ale acestui graf sunt:
  - A care leagă pe B de restul grafului.
  - H care leagă pe I de restul grafului.
  - L care leagă pe M de restul grafului.
  - G prin a cărui suprimare graful se divide în trei părți.
- În concluzie în cadrul grafului din figură există **șase componente biconexe**:
  - (1) Grupul de noduri  $\{A, C, G, D, E, F\}$
  - (2) Grupul de noduri  $\{G, J, H, K\}$
  - (3) Nodul individual B
  - (4) Nodul individual I
  - (5) Nodul individual L
  - (6) Nodul individual M.



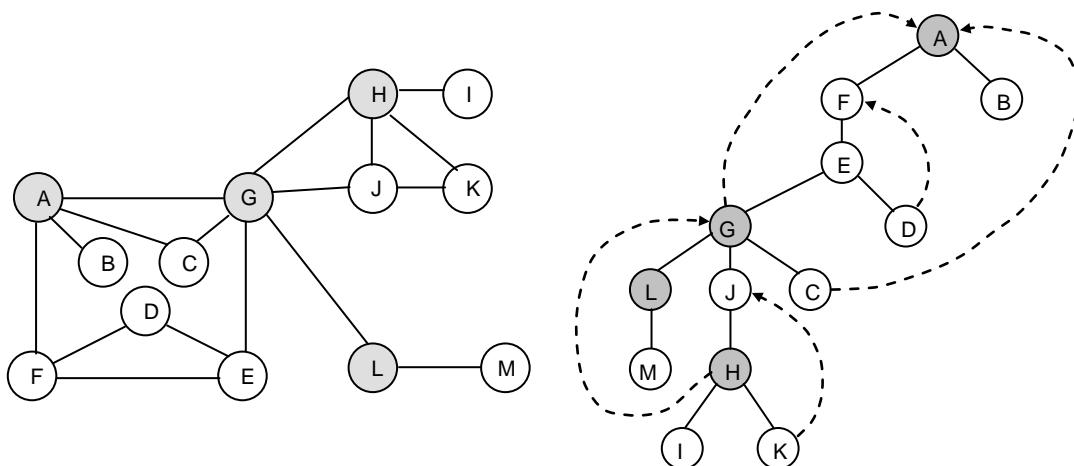
**Fig.10.5.2.2.a.** Graf care nu este biconex

- În acest context una din problemele care se ridică este aceea a determinării **punctelor de articulație** ale unui graf.
  - Se face precizarea că aceasta este una din mulțimea de probleme deosebit de importante referitoare la conexitatea grafurilor.
- Astfel, ca și un **exemplu de aplicație al conexității grafurilor** poate fi prezentată o rețea care este de fapt un graf în care nodurile comunică unele cu altele.
  - În legătură cu această rețea se ridică următoarea întrebare fundamentală: “Care este **capacitatea de supraviețuire** a rețelei atunci când unele dintre nodurile sale cad?”
  - Cu alte cuvinte în ce condiții, în urma căderii unor noduri rețeaua rămâne încă funcțională?
- Un **graf are conexitatea k** sau altfel spus are **numărul de conexitate egal cu k** dacă prin suprimarea a oricare **k-1** noduri ale sale graful rămâne conex [FK69].

- Spre exemplu, **un graf are conexitatea doi**, dacă și numai dacă **nu are puncte de articulație**, cu alte cuvinte, dacă și numai dacă este **biconex**.
- Cu cât numărul de conexitate al unui graf este mai mare, cu atât capacitatea de supraviețuire a grafului la căderea unora din nodurile sale este mai mare.

### 10.5.2.3. Determinarea punctelor de articulație ale unui graf

- În vederea determinării **punctelor de articulație** ale unui **graf conex**, poate fi utilizată, printr-o extensie simplă, **traversarea grafurilor prin tehnică căutării în adâncime**.
  - **Absența punctelor de articulație** precizează un **graf biconex**.
- Se consideră spre **exemplu**, graful din figura 10.5.2.2.a și un arbore de căutare în adâncime asociat, ca și cel din fig.10.5.2.3.a.



**Fig.10.5.2.3.a.** Arbore de căutare în adâncime pentru determinarea punctelor de articulație ale unui graf conex

- Se observă că suprimarea nodului **E** **nu** conduce la dezmembrarea grafului deoarece ambii fii ai acestuia, **G** respectiv **D** sunt conectați prin **arce de return** (linii punctate) cu noduri situate deasupra în arbore.
- Pe de altă parte suprimarea lui **G** conduce la scindarea grafului deoarece nu există astfel de alternative pentru nodurile **L** sau **J**.
- Un **nod** oarecare **x** al unui graf **nu este un punct de articulație** dacă **fiecare fiu** **y** al său are printre descendenți vreun nod conectat (printr-o linie punctată) cu un nod situat în arbore deasupra lui **x**, cu alte cuvinte, dacă există o conexiune alternativă de la **x** la **y**.
  - Această verificare **nu** este valabilă pentru **rădăcina** arborelui de căutare în adâncime deoarece nu există noduri situate “deasupra” acesteia.
  - În consecință, **rădăcina** este un **punct de articulație** dacă are doi sau mai mulți fii deoarece singurul drum care conectează fiii rădăcinii trece prin rădăcina însăși.
- Determinarea **punctelor de articulație** poate fi implementată pornind de la **căutarea în adâncime**, prin transformarea procedurii de căutare într-o **funcție** care returnează pentru nodul furnizat ca parametru, **cel mai înalt punct** din cadrul arborelui de

căutare întâlnit în timpul căutării, adică nodul cu cea mai mică valoare memorată în tabloul marc.

- Algoritmul implementat în forma funcției **Articulație** apare în secvența [10.5.2.3.a].
- 

```
/*Determinarea punctelor de articulație ale unui graf
reprezentat prin SA - varianta pseudocod*/

int function articulatie(int x)
/*returnează cel mai înalt nod din arborele de căutare care
poate fi întâlnit pornind de la x*/
Ref_Tip_Nod t;
int m,min;

id= id+1; marc[x]= id; min= id; /*marcare nod curent*/
t= StrAdj[x]; /*t - pointer în lista de adiacențe*/
cât timp (t<>null) /*prelucrare structură de adiacențe*/
    dacă (marc[t->nume]==0)                      /*10.5.2.3.a*/
        m= articulatie(t->nume);
        dacă (m<min)
            min= m;
        dacă (m>=marc[x])
            *scrive(x); /*x este punct de articulație*/
            /*dacă*/
        altfel
            dacă (marc[t->nume]<min)
                min= marc[t->nume];
            t= t->urm;
            /*cât timp*/
    returneaza min;
/*Articulatie*/
```

---

- Referitor la această funcție se fac următoarele precizări:
  - (1) La căutarea în adâncime într-un graf, valoarea lui `marc[x]` pentru orice nod `x` al grafului precizează **numărul de ordine** al nodului `x` în cadrul traversării.
    - Aceeași ordine rezultă și din traversarea în preordine a nodurilor arborelui de căutare în adâncime.
    - Cu cât nodul este traversat mai devreme, cu atât numărul său de ordine în `marc` este mai mic și cu atât poziția sa în cadrul arborelui de căutare este mai înaltă.
    - **Concluzia:** descendenții unui nod vor avea în tabloul `marc` numere mai mari decât nodul părinte și vor fi situați sub acesta în arborele de căutare.
  - (2) Pentru fiecare nod `x`, valoarea `min` returnată de funcția **Articulație** este **cel mai mic număr de ordine** întâlnit în cadrul traversării.
    - Acest număr poate corespunde chiar lui `x` sau oricărui nod `z` la care se poate ajunge pornind de la `x`, coborând zero sau mai multe arce ale

arborelui până la un descendant  $w$  ( $w$  poate fi chiar  $x$ ), iar apoi urmând un arc de return ( $w, z$ ).

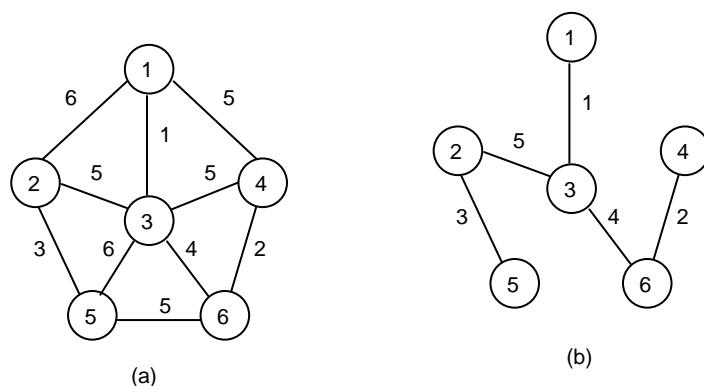
- Valoarea  $\min$  se calculează pentru toate nodurile  $x$ , traversând arborele în **postordine**.
  - Astfel, când se procesează nodul  $x$  valoarea  $\min$  a fost deja calculată pentru toți fișii  $y$  ai lui  $x$ .
- (3) Pentru un nod oarecare  $x$ , valoarea  $\min$  asociată este **cea mai mică valoare** dintre:
  - Numărul de ordine al lui  $x$ .
  - Numărul de ordine al oricărui nod  $z$  pentru care există în arborele de căutare în adâncime un arc de return ( $x, z$ ).
  - Valoarea lui  $\min$  pentru toți fișii  $y$  ai lui  $x$ .
- (4) **Punctele de articulație** se găsesc astfel:
  - **Rădăcina este punct de articulație** dacă și numai dacă are doi sau mai mulți fișii.
  - **Un nod**  $x$ , altul decât rădăcina este un **punct de articulație** dacă și numai dacă există vreun fiu  $y$  al lui  $x$  astfel încât valoarea  $\min$  pentru  $y$  este **mai mare sau egală** cu numărul de ordine al lui  $x$ .
    - În acest caz,  $x$  deconectează pe  $y$  și descendenții acestuia de restul grafului.
- (5) Dacă valoarea  $\min$  pentru **toți fișii**  $y$  ai lui  $x$  este **mai mică** decât numărul de ordine al lui  $x$ , atunci există cu siguranță un drum pe care se poate ajunge de la oricare din fișii  $y$  ai lui  $x$  înapoi la un strămoș propriu  $z$  a lui  $x$  (nodul care are numărul de ordine egal cu  $\min$  pentru  $y$ ).
  - În consecință suprimarea nodului  $x$  nu conduce la deconectarea nici unui fiu  $y$  al lui  $x$  sau a descendenților săi de restul grafului [AH85].
- Funcția **Articulație( $x$ )** determină de fapt în manieră recursivă **cel mai înalt punct al arborelui** care poate fi atins via un arc de return pentru **orice descendant** al nodului  $x$  furnizat ca parametru și utilizează această informație pentru a stabili dacă  $x$  este un punct de articulație.
  - După cum s-a precizat, aceasta presupune o simplă comparație între valoarea  $m$ , adică minimul determinat pentru nodul descendant curent și  $\text{marc}[x]$  adică numărul de ordine al lui  $x$ .
  - În plus, mai este necesar să se verifice dacă  $x$  nu este cumva **rădăcina** arborelui de căutare în adâncime, cu alte cuvinte, dacă  $x$  nu este cumva primul nod parcurs în apelul funcției **Articulație** pentru componenta conexă a grafului care conține nodul  $x$ .
    - Această verificare se face în afara funcției recursive, motiv pentru care ea nu este prezentă în secvența [10.5.2.3.a].
- Această secvență care de fapt afișează punctele de articulație, poate fi ușor extinsă pentru a realiza și alte prelucrări asupra **punctelor de articulație** sau a **componentelor biconexe**.
- Deoarece procedeul derivă din **traversarea grafurilor prin tehnica căutării în**

**adâncime**, efortul său de execuție este proporțional cu  $O(n+a)$  în reprezentarea grafurilor prin structuri de adiacențe, respectiv cu  $O(n^2)$  în reprezentarea bazată pe matrice de adiacențe,  $n$  reprezentând numărul de noduri, iar  $a$  numărul de arce al grafului.

## 11. Grafuri ponderate ("Weighted Graphs")

- Adeseori, modelarea problemelor practice presupune utilizarea unor grafuri în care arcelor li se asociază **ponderi** care pot fi greutăți, costuri, valori, etc.
- Astfel de grafuri se mai numesc și **grafuri ponderate** ("weighted graphs").
  - Spre exemplu, pe **harta traseelor aeriene** ale unei zone, arcele reprezintă rute de zbor iar ponderile distanțe, tempi sau prețuri.
  - Într-un **circuit electric** unde arcele reprezintă legături, lungimea sau caracteristicile fizice ale acestora sunt de regulă utilizate ca ponderi.
  - Într-o activitate de **planificare în execuție a taskurilor**, ponderea poate reprezenta fie timpul, fie costul execuției unui task, fie timpul de așteptare până la lansarea în execuție a taskului.
- Este evident faptul că într-un asemenea context apar în mod natural probleme legate de **minimizarea costurilor**.
- În cadrul acestui paragraf vor fi prezentate mai în detaliu două astfel de probleme referitoare la grafurile ponderate:
  - (1) Găsirea **drumului cu costul cel mai redus** care conectează toate punctele grafului.
  - (2) Găsirea **drumului cu costul cel mai redus** care leagă două puncte date.
- Prima problemă care este în mod evident utilă pentru grafuri reprezentând circuite electrice sau ceva analog, se numește **problema arborelui de acoperire minim** ("minimum spanning tree problem").
- Cea de-a doua problemă este utilă în grafurile reprezentând hărți de trasee (aeriene, feroviare, turistice) și se numește **problema drumului minim** ("shortest-path problem").
  - Aceste probleme sunt tipice pentru o largă categorie de aspecte ce apar în prelucrarea **grafurilor ponderate**.
  - Se impune o precizare.
  - De regulă algoritmii utilizați presupun parcurgerea grafului, motiv pentru care în mod intuitiv **ponderile** sunt asociate cu **distanțe**.
    - Se spune de obicei "cel mai apropiat nod" cu sensul de poziționare geografică a nodului.
    - De fapt acest mod de a concepe lucrurile este valabil în contextul **problemei drumului minim**.
  - În general, este însă foarte important să se avea în vedere faptul, că **nu** este absolut necesar ca ponderile să fie proporționale cu distanțele și că ele pot reprezenta orice altceva, ca spre exemplu **tempi, costuri sau valori**.

- În situațiile în care ponderile reprezintă într-adevăr distanțe, alți algoritmi cu caracter specific, pot fi mai potriviti decât cei care vor fi prezentati în continuare.
- În figura 11.a (a) apare o reprezentare grafică a unui **graf neorientat ponderat**.



**Fig.11.a** Reprezentarea unui graf ponderat și a unui arbore de acoperire minim asociat

- În ceea ce privește **reprezentarea structurii de date abstracte graf ponderat**, principal ea se reprezintă ca și grafurile normale cu următoarele **deosebiri**:
  - (1) În cazul reprezentării prin **matrice de adiacențe**, matricea va conține **ponderi** în locul valorilor booleene.
  - (2) În cazul reprezentării prin **structuri de adiacențe**, fiecărui element al listei i se adăugă un **câmp suplimentar** pentru memorarea **ponderii**.
- Se presupune faptul că **ponderile sunt toate pozitive**.
  - Există însă algoritmi mult mai complicați care pot trata și ponderi negative.
  - Astfel de algoritmi sunt utilizați mai rar în activitatea practică.

## 11.1. Arbori de acoperire minimi ("Minimum-Cost Spanning Trees")

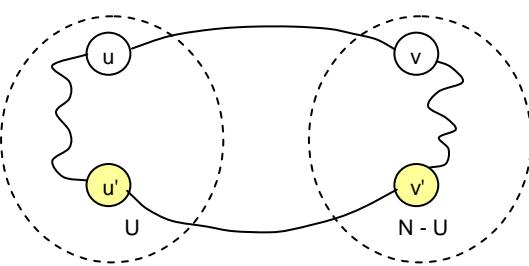
- Se presupune că  $G = (N, A)$  este un graf conex în care oricare arc  $(u, v)$  aparținând lui A are atașat un cost specific  $\text{cost}(u, v)$ .
- Un **arbore de acoperire** al lui G este un **arbore liber** care cuprinde toate nodurile din N (fig.11.a.(b)).
  - **Costul** unui arbore de acoperire este **suma costurilor** tuturor arcelor cuprinse în arbore.
- **Definiție:** Un **arbore de acoperire minim** al unui graf ponderat este o selecție minimă de arce care conectează toate nodurile grafului, astfel încât costul său este cel puțin la fel de mic ca și costul oricărui alt arbore de acoperire al grafului.
- O altă **definiție** este următoarea.
  - Dându-se **orice partitioare** a mulțimii nodurilor unui graf în două submulțimi, **arborele de acoperire minim** conține **arcele cu ponderea cea mai mică** care leagă cele două mulțimi.

**mai mică** care conectează un nod aparținând unei submulțimi cu un nod aparținând celeilalte [Se 88].

- Se face precizarea că un arbore de acoperire minim **nu** este în mod necesar **unic**.
- O aplicație tipică a arborilor de acoperire minimi o reprezintă proiectarea **rețelelor de comunicații**.
  - Nodurile grafului reprezintă orașele iar arcele, comunicațiile posibile dintre ele.
  - Costul asociat unui arc reprezintă de fapt costul selecției acelei legături a rețelei.
  - Un **arbore de acoperire minim** reprezintă o rețea care conectează cu un cost minim toate orașele.

### 11.1.1. Proprietatea arborilor de acoperire minimi

- Există mai multe moduri de a construi **arbori de acoperire minimi** asociați unui graf ponderat.
- Marea lor majoritate se bazează pe următoarea proprietate, denumită și **proprietatea arborilor de acoperire minimi**.
  - Fie  $G = (N, A)$  un **graf conex** și o **funcție de cost** definită pe arcele sale.
  - Fie  $U$  o submulțime proprie a **mulțimii de noduri**  $N$ .
  - Dacă  $(u, v)$  este **un arc cu costul cel mai scăzut** astfel încât  $u \in U$  iar  $v \in N - U$ , atunci există un **arbore de acoperire minim** care include arcul  $(u, v)$ .
- **Demonstrația** acestei aserțiuni nu este complicată și ea se realizează prin **metoda reducerii la absurd**.
  - Se presupune dimpotrivă că **nu** există un arbore de acoperire minim al lui  $G$  care include **arcul cu costul cel mai scăzut**  $(u, v)$ .
  - Fie  $T$  oricare **arbore de acoperire minim** al lui  $G$ .
  - Adăugarea arcului  $(u, v)$  arborelui  $T$  trebuie să conducă la apariția unui **ciclu**, deoarece  $T$  este un **arbore liber** și conform proprietății (b) dacă unui arbore liber i se adaugă un arc el devine un graf ciclic, adică va conține un ciclu (& 10.1.).
    - Acest ciclu include arcul  $(u, v)$ .
  - În consecință, în ciclul nou format, trebuie să existe un alt arc  $(u', v')$  al lui  $T$  astfel încât  $u' \in U$  și  $v' \in N - U$ , după cum rezultă din figura 11.1.1.a.
    - Dacă acest lucru **nu** ar fi adevărat, atunci în cadrul ciclului **nu** ar exista o altă posibilitate de a ajunge de la nodul  $v$  la nodul  $u$  decât reparcurgând arcul  $(u, v)$ .



**Fig.11.1.1.a.** Demonstrarea proprietății arborilor de acoperire minimi

- Suprimând arcul  $(u', v')$ , ciclul dispare și obținem arborele de acoperire  $T'$  al cărui cost **nu** este cu siguranță mai ridicat decât al lui  $T$ , deoarece s-a presupus inițial că  $\text{cost}(u, v) \leq \text{cost}(u', v')$ , adică  $(u, v)$  este un **arc cu costul cel mai scăzut**, după cum s-a precizat în condițiile inițiale.
- Astfel existența lui  $T'$  **contrazice** presupunerea inițială și anume că **nu** există un arbore de acoperire minim care să includă arcul  $(u, v)$  și în consecință proprietatea arborilor de acoperire minimi este demonstrată.

## 11.2. Determinarea arborilor de acoperire minimi

- Există mai multe metode de determinare a unui **arbore minim asociat unui graf ponderat**, metode care în general exploatează proprietatea anterior enunțată pentru acești arbori.
- Dintre acestea se remarcă cu deosebire:
  - (1) Algoritmul lui Prim.
  - (2) Metoda căutării “bazate pe prioritate”.
  - (3) Algoritmul lui Kruskal.

### 11.2.1. Algoritmul lui Prim

- Fie  $G$  **graful ponderat** pentru care se dorește determinarea unui **arbore de acoperire minim**.
  - Fie  $N = \{1, 2, 3, \dots, n\}$  mulțimea nodurilor grafului  $G$ .
  - Fie  $U$  o mulțime vidă de noduri ale grafului.
- **Algoritmul lui Prim**
  - Începe prin a introduce în mulțimea  $U$  nodul de pornire, să zicem nodul  $\{1\}$ .
  - În continuare, într-o manieră ciclică, este construit pas cu pas **arborele de acoperire minim**.
  - Astfel, în fiecare **pas** al algoritmului:
    - (1) Se selecteză arcul cu **cost minim**  $(u, v)$  care conectează mulțimea  $U$  cu mulțimea  $N - U$ .

- (2) Se adaugă acest arc arborelui de acoperire minim.
- (3) Se adaugă nodul v mulțimii U.
- Ciclul se repetă până când  $U=N$ .
- Schița **algoritmului lui Prim** apare în secvența [11.2.1.a].

---

/\*Construcția unui arbore de acoperire minim AAM al unui graf G\*/

```

procedure PRIM(Tip_Graf G; Multime_De_Arce AAM);
  /*construiește multimea AAM care conține arcele unui arbore de acoperire minim al grafului G*/

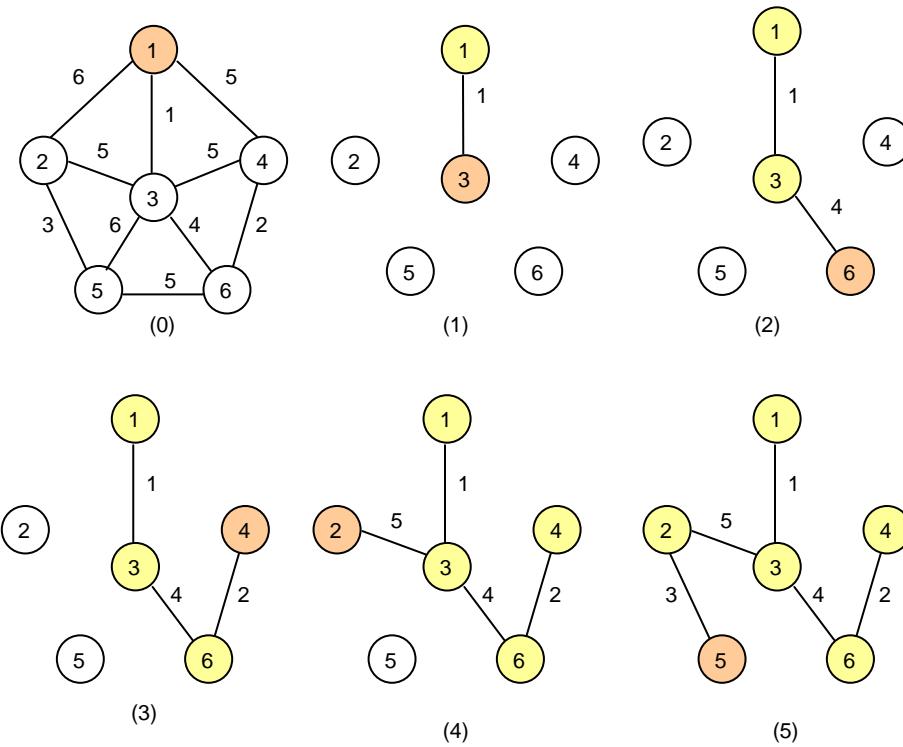
  Multime_De_Noduri U;
  Tip_Nod u,v;

  AAM= {multimea vidă};
  U= {nodul de pornire};
  căt timp (U<>N) execută          /*[11.2.1.a]*/
    fie (u,v) arcul cu costul cel mai redus care
      satisface condiția (u aparține lui U) && (v aparține
        lui N-U);
    AAM= AAM ∪ {(u,v)}; /*adaugă arcul (u,v) multimii AAM*/
    U= U ∪ {v};       /*adaugă nodul v multimii U*/
  /*căt timp*/
/*PRIM*/

```

---

- În figura 11.2.1.a apare reprezentată pas cu pas secvența de construcție a arborelui de acoperire minim pentru graful (0) din aceeași figură.



**Fig.11.2.1.a.** Construcția unui arbore de acoperire minim al unui graf pe baza algoritmului lui Prim

### 11.2.1.1. Exemplu de implementare a algoritmului lui Prim

- Se presupune un graf definit prin:
  - (1) Mulțimea nodurilor sale  $\{1, 2, 3, \dots, n\}$ .
  - (2) Matricea COST care memorează costurile arcelor sale.
- O modalitate simplă de a afla arcul cu costul cel mai redus care conectează mulțimile U și N-U, este aceea de a utiliza două tablouri.
  - Primul dintre tablouri, denumit APROPIAT memorează în locația APROPIAT[i] acel nod care este la momentul respectiv cel mai apropiat de nodul i și care aparține mulțimii N-U. Nodul i aparține mulțimii U.
    - Acest tablou materializează de fapt **mulțimea** N-U.
  - Cel de-al doilea tablou denumit COSTMIN, memorează în locația nodului i costul arcului ( $i, APROPIAT[i]$ ).
- **Algoritmul lui Prim** de construcție al **arborelui de acoperire minim** este următorul:
  - (1) Se inițializează U cu mulțimea vidă și se selectează un nod de pornire care se introduce în mulțimea U.
  - (2) Se inițializează corespunzător tablourile APROPIAT și COSTMIN.
  - (3) În fiecare pas al algoritmului se balează tabloul COSTMIN pentru a găsi acel nod, fie acesta k, aparținând mulțimii N-U, care este cel mai apropiat de nodurile mulțimii U, adică nodul pentru care valoarea COSTMIN[k] este minimă.
  - (4) Se adaugă nodul k mulțimii U și se tipărește arcul ( $k, APROPIAT[k]$ ) ca și aparținând **arborelui de acoperire minim**.
  - (5) În continuare se actualizează tablourile COSTMIN și APROPIAT luând în considerare faptul că nodul k a fost adăugat mulțimii U. În consecință:
    - (1) Pe de o parte apar noi posibilități de conectare între mulțimile U și N-U.
    - (2) Pe de altă parte costurile unor conexiuni existente se pot reduce prin introducerea noului nod k în mulțimea U.
  - (6) Algoritmul se reia de la pasul (3) până când toate nodurile au fost adăugate mulțimii U.
- O versiune C a algoritmului lui Prim apar în secvența [11.2.1.1.a].
- Se presupune că COST este un tablou de dimensiuni nxn, astfel încât COST[i, j] reprezintă costul arcului ( $i, j$ ).
  - Dacă arcul ( $i, j$ ) nu există, COST[i, j] are o **valoare mare specifică**. Vom nota această valoare cu simbolul  $\infty$ .

- Ori de câte ori se găsește un nod  $k$  pentru a fi introdus în mulțimea  $U$ , se face  $\text{COSTMIN}[k]$  egal cu valoarea “**infinit**”, pentru ca nodul respectiv să nu mai fie selectat.
    - Valoarea **infinit** reprezintă o valoare mare convenită, astfel încât acest nod să nu mai poată fi selectat în continuare spre a fi inclus în  $U$ . Vom nota această valoare cu  $\infty+$ .
    - Se face precizarea că valoarea **infinit** trebuie să fie **mai mare** decât costul oricărui arc al grafului, respectiv **mai mare** decât **costul** asociat **lipsei arcului**.
- 

### {Implementarea algoritmului lui Prim - varianta C}

```

procedure Prim(float COST[MaxNod,MaxNod], int n)

/*afișează arcele arborelui de acoperire minim pentru un
graf având nodurile {1,2,...,n} și matricea COST pentru
costurile arcelor*/

float COSTMIN[MaxNod];
int APROPIAT[MaxNod];
int i,j,k,min;

/*i și j sunt indici. În timpul parcurgerii tabloului
COSTMIN, k este indicele celui mai apropiat nod găsit pâna
la momentul curent, iar min=COSTMIN[k]*/
COSTMIN[1]= infinit+; /*nodul 1 este nod de start*/

/*initializează mulțimea U numai cu nodul 1*/
for (i=2;i<=n;i++)
{
  COSTMIN[i]=COST[1,i];
  APROPIAT[i]=1;
} /*for*/                                /*[11.2.1.1.a]*/

/*caută cel mai apropiat nod k din afara mulțimii U,
față de mulțimea U*/
for (i=2;i<=n;i++)
{
  min=COSTMIN[2];
  k=2;
  for (j=3;j<=n;j++)
    if (COSTMIN[j]<min)
    {
      min=COSTMIN[j];
      k=j; /*k este cel mai apropiat nod de mulțimea U*/
    } /*if*/
  *scrive(k,APROPIAT[k]); /*tipărește arcul minim*/
  COSTMIN[k]=infinity; /*k se adaugă lui U*/
  for (j=2;j<=n;j++) /*ajusteză costurile minime ale
                        nodurilor mulțimii U după includerea nodului k*/
    if(COST[k,j]<COSTMIN[j] && COSTMIN[j]<infinity)
    {
      COSTMIN[j]=COST[k,j];
      APROPIAT[j]=k;
    } /*if*/
}

```

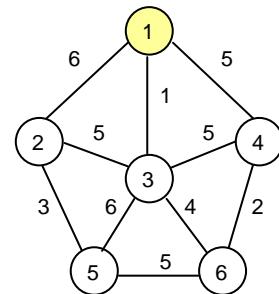
```

} /*for*/
/*Prim*/

```

- În figura 11.2.1.1.b apare urma execuției algoritmului lui Prim aplicat grafului din figura 11.2.1.1.a.(0).

(0) initializare <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Nod</th> <th>1</th> <th>2</th> <th><b>3</b></th> <th>4</th> <th>5</th> <th>6</th> </tr> </thead> <tbody> <tr> <td>Apropiat</td> <td>-</td> <td>1</td> <td>1</td> <td>1</td> <td>1</td> <td>1</td> </tr> <tr> <td>CostMin</td> <td><math>\infty +</math></td> <td>6</td> <td><b>1</b></td> <td>5</td> <td><math>\infty</math></td> <td><math>\infty</math></td> </tr> </tbody> </table> $U=\{1\}$ $N-U=\{2,3,4,5,6\}$	Nod	1	2	<b>3</b>	4	5	6	Apropiat	-	1	1	1	1	1	CostMin	$\infty +$	6	<b>1</b>	5	$\infty$	$\infty$	(1) se selecteaza nodul 3 <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Nod</th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> <th>5</th> <th>6</th> </tr> </thead> <tbody> <tr> <td>Apropiat</td> <td>-</td> <td>3</td> <td>-</td> <td>1</td> <td>3</td> <td>3</td> </tr> <tr> <td>CostMin</td> <td><math>\infty +</math></td> <td>5</td> <td><math>\infty +</math></td> <td>5</td> <td>6</td> <td><b>4</b></td> </tr> </tbody> </table> $U=\{1,3\}$ $N-U=\{2,4,5,6\}$	Nod	1	2	3	4	5	6	Apropiat	-	3	-	1	3	3	CostMin	$\infty +$	5	$\infty +$	5	6	<b>4</b>
Nod	1	2	<b>3</b>	4	5	6																																					
Apropiat	-	1	1	1	1	1																																					
CostMin	$\infty +$	6	<b>1</b>	5	$\infty$	$\infty$																																					
Nod	1	2	3	4	5	6																																					
Apropiat	-	3	-	1	3	3																																					
CostMin	$\infty +$	5	$\infty +$	5	6	<b>4</b>																																					
(2) se selecteaza nodul 6 <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Nod</th> <th>1</th> <th>2</th> <th>3</th> <th><b>4</b></th> <th>5</th> <th>6</th> </tr> </thead> <tbody> <tr> <td>Apropiat</td> <td>-</td> <td>3</td> <td>-</td> <td>6</td> <td>6</td> <td>-</td> </tr> <tr> <td>CostMin</td> <td><math>\infty +</math></td> <td>5</td> <td><math>\infty +</math></td> <td><b>2</b></td> <td>5</td> <td><math>\infty +</math></td> </tr> </tbody> </table> $U=\{1,3,6\}$ $N-U=\{2,4,5\}$	Nod	1	2	3	<b>4</b>	5	6	Apropiat	-	3	-	6	6	-	CostMin	$\infty +$	5	$\infty +$	<b>2</b>	5	$\infty +$	(3) se selecteaza nodul 4 <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Nod</th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> <th>5</th> <th>6</th> </tr> </thead> <tbody> <tr> <td>Apropiat</td> <td>-</td> <td>3</td> <td>-</td> <td>-</td> <td>6</td> <td>-</td> </tr> <tr> <td>CostMin</td> <td><math>\infty +</math></td> <td><b>5</b></td> <td><math>\infty +</math></td> <td><math>\infty +</math></td> <td>5</td> <td><math>\infty +</math></td> </tr> </tbody> </table> $U=\{1,3,6,4\}$ $N-U=\{2,5\}$	Nod	1	2	3	4	5	6	Apropiat	-	3	-	-	6	-	CostMin	$\infty +$	<b>5</b>	$\infty +$	$\infty +$	5	$\infty +$
Nod	1	2	3	<b>4</b>	5	6																																					
Apropiat	-	3	-	6	6	-																																					
CostMin	$\infty +$	5	$\infty +$	<b>2</b>	5	$\infty +$																																					
Nod	1	2	3	4	5	6																																					
Apropiat	-	3	-	-	6	-																																					
CostMin	$\infty +$	<b>5</b>	$\infty +$	$\infty +$	5	$\infty +$																																					
(4) se selecteaza nodul 2 <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Nod</th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> <th><b>5</b></th> <th>6</th> </tr> </thead> <tbody> <tr> <td>Apropiat</td> <td>-</td> <td>-</td> <td>-</td> <td>-</td> <td>2</td> <td>-</td> </tr> <tr> <td>CostMin</td> <td><math>\infty +</math></td> <td><math>\infty +</math></td> <td><math>\infty +</math></td> <td><math>\infty +</math></td> <td><b>3</b></td> <td><math>\infty +</math></td> </tr> </tbody> </table> $U=\{1,3,6,4,2\}$ $N-U=\{5\}$	Nod	1	2	3	4	<b>5</b>	6	Apropiat	-	-	-	-	2	-	CostMin	$\infty +$	$\infty +$	$\infty +$	$\infty +$	<b>3</b>	$\infty +$	(5) se selecteaza nodul 5 <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>Nod</th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> <th>5</th> <th>6</th> </tr> </thead> <tbody> <tr> <td>Apropiat</td> <td>-</td> <td>-</td> <td>-</td> <td>-</td> <td>-</td> <td>-</td> </tr> <tr> <td>CostMin</td> <td><math>\infty +</math></td> <td><math>\infty +</math></td> <td><math>\infty +</math></td> <td><math>\infty +</math></td> <td><math>\infty +</math></td> <td><math>\infty +</math></td> </tr> </tbody> </table> $U=\{1,3,6,4,2,5\}$ $N-U=\{\}$	Nod	1	2	3	4	5	6	Apropiat	-	-	-	-	-	-	CostMin	$\infty +$					
Nod	1	2	3	4	<b>5</b>	6																																					
Apropiat	-	-	-	-	2	-																																					
CostMin	$\infty +$	$\infty +$	$\infty +$	$\infty +$	<b>3</b>	$\infty +$																																					
Nod	1	2	3	4	5	6																																					
Apropiat	-	-	-	-	-	-																																					
CostMin	$\infty +$																																										



**Fig.11.2.1.1.b.** Exemplu de aplicare a algoritmului lui Prim

### 11.2.1.2. Analiza performanței algoritmului lui Prim

- Complexitatea** relativă la timpul de execuție a implementării prezentate a algoritmului lui Prim este  $O(n^2)$ , deoarece:
  - (1) Se fac  $n-1$  iterații în bucla exterioară **for**.
  - (2) Fiecare iterație necesită  $O(n)$  unități de timp datorită celor două bucle **for** interioare succesive, prima care determină arcul minim și cea de-a doua care ajustează costurile lui  $U$ .
- Pentru valori mari ale lui  $n$ , performanța algoritmului poate deveni nesatisfăcătoare.

### 11.2.2. Metoda căutării "bazate pe prioritate" ("Priority-First Search")

- După cum s-a precizat în &10.4.3, considerând nodurile unui graf divizate în trei clase, "arbore", "vecinătate" și "neîntâlnite", atunci metodele de traversare a grafului se diferențiază după maniera în care sunt alese nodurile care trec din clasa "vecinătate" în clasa "arbore".
  - Astfel la traversarea "în adâncime" se alege din vecinătate nodul **cel mai recent**

**întâlnit** (ultimul) ceea ce corespunde utilizării unei **stive** în păstrarea nodurilor clasei “vecinătate”.

- La traversarea “**prin cuprindere**” se alege nodul **cel mai devreme întâlnit** (primul) ceea ce corespunde unei **structuri de date coadă**.
- Determinarea **arborelui de acoperire minim** se poate asimila cu acea traversare a grafului în care se alege din clasa “vecinătate” nodul cu **prioritatea maximă**.
  - Aceste poate fi spre exemplu, acel nod la care conduce **arcul cu ponderea minimă**.
- **Structura de date** care poate fi asociată acestei metode este **coada bazată pe prioritate**.
- În secvența [11.2.2.a] se prezintă o metodă de determinare a **arborelui minim** bazată pe considerențele mai sus precizate.
  - Tehnica utilizată se mai numește și **căutare bazată pe prioritate** (“**priority first search**”) [Se 88].
- Referitor la această secvență se fac următoarele **precizări**:
  - Graful se va considera reprezentat printr-o **structură de adiacențe**, implementată cu ajutorul **listelor înlántuite simple** (&10.3.2, Caz 1).
  - **Structura nodului** listelor de adiacențe se completează cu câmpul “cost” utilizat pe post de **prioritate**.
    - În acest câmp se memorează costul arcului care conduce la nodul în cauză.
  - Procedura **Initializeaza** și funcțiile **Extragă** și **Vid** implementează operatorii respectivi în contextul **cozilor bazate pe prioritate**.
  - Funcția **boolean Actualizeaza(Coada\_Bazata\_Pe\_Prioritate q, Tip\_Nod k, Tip\_Prioritate p)** implementează un operator special referitor la coada bazată pe prioritate.
    - Operatorul verifică dacă nodul  $k$  precizat ca parametru apare în coada bazată pe prioritate  $q$ , cu o prioritate cel puțin egală cu prioritatea  $p$  precizată ca parametru și acționază după cum urmează:
      - (1) Dacă nodul  $k$  nu apare în coadă el este inserat cu prioritatea  $p$ .
      - (2) Dacă nodul  $k$  apare în coadă însă are un cost mai mare (adică o prioritate mai mică) decât prioritatea  $p$  precizată ca parametru, se realizează schimbarea priorității sale la valoarea  $p$ .
      - (3) Dacă nodul  $k$  apare în coadă însă are un cost mai mic (adică o prioritate mai mare) decât cea precizată ca parametru, operatorul nu face nimic.
      - (4) Dacă se produce vreo modificare (inserție sau modificare de prioritate) funcția **Actualizeaza** returnează valoarea “true”. Aceasta permite actualizarea corespunzătoare a tablourilor **marc** și **parinte**.

---

/\*Determinarea unui arbore de acoperire minim al unui graf  
prin metoda căutării bazate pe prioritate - varianta  
pseudocod - se utilizează TDA Coadă Bazată pe Prioritate\*/

```

/*structuri de date*/

typedef struct Tip_Nod* Ref_Tip_Nod;

typedef struct Tip_Nod /*structura unui nod al listei de
                        adiacențe*/
{
    int nume;
    int cost;
    Ref_Tip_Nod urm;
} Nod;

Ref_Tip_Nod StrAdj[MaxNod]; /*structura de adiacențe*/
int id,k;
int marc[MaxNod]; /*tablou pentru evidența nodurilor*/
int parinte[MaxNod]; /*arborele de acoperire minim*/
Coada_Bazata_Pe_Prioritate q;

void CautaPrioritar(int k);

/*construiește arborele de acoperire minim în varianta
indicator spre părinte (tabloul parinte) pornind de la nodul
k*/
Ref_Tip_Nod t;

[1] daca Actualizeaza(q,k,nevazut) parinte[k]=0;
     /*amorsare căutare, valabilă numai pentru rădăcină*/
[2] repeta
[3]   id=id+1;                                /*[11.2.2.a]*/
[4]   k=Extrage(q); /*k este nodul cel mai prioritar*/
[5]   marc[k]=-marc[k]; /*k trece în clasa "arbore"*/
[6]   daca (marc[k]==nevazut) marc[k]=0; /*pt. rădăcină*/
[7]   t=Stradj[k]; /*lista de adiacențe a lui k*/
[8]   cat timp (t<>null)
[9]     daca (marc[t^.nume]<0)/*nevizitat sau în coadă*/
[10]      daca Actualizeaza(q,t^.nume,t^.cost)
[11]        marc[t->nume]=-(t->cost); /*nodul t^.nume
                                         trece în clasa "vecinătate"*/
[12]        parinte[t->nume]=k; /*marcare părinte*/
[13]        /*daca*/
[14]        t=t->urm;
[15]      /*cat timp*/
[16] pana cand Vid(q);
[17]   /*repeta*/
/*CautaPrioritar*/

void ArboreMinim
/*programul principal pentru construcția unui arbore de
acoperire minim*/

id=0;
Initializeaza(q);
pentru (k=1 la N)
    marc[k]=-nevazut;
pentru (k=1 la N)
    daca (marc[k]==-nevazut)
        CautaPrioritar(k);

```

- Arborele de acoperire minim care în acest caz este un **arbore de căutare bazată pe prioritate** este păstrat în tabloul **parinte** în reprezentarea “indicator spre părinte”.
- Fiecare locație a tabloului **parinte** memorează părintele nodului în cauză respectiv nodul care a determinat mutarea nodului din clasa “vecinătate” în clasa “arbore”.
- Tabloul **mark** memorează starea nodurilor grafului.
  - Pentru fiecare nod  $k$  al arborelui,  $\text{marc}[k]$  memorează de fapt **prioritatea** nodului în cauză, respectiv **costul arcului** care-l leagă pe  $k$  de părintele său  $\text{parinte}[k]$ .
- Sunt valabile următoarele **convenții**:
  - (1) Nodurile din clasa “arbore” sunt marcate cu **valori pozitive** în tabloul **marc**.
  - (2) Nodurile din clasa "vecinătate" sunt marcate cu **valori negative** (linia [11]).
  - (3) Nodurile din clasa “neîntâlnite” sunt marcate cu “-nevăzut” și nu cu valoarea zero.
    - Se face precizarea că nevăzut reprezintă o valoare mare pozitivă.
- Se observă faptul că atâtă vreme cât nodurile se găsesc în **coada bazată pe prioritate** ele sunt marcate în tabloul **marc** cu **valoarea negativă** a costului (priorității).
- În momentul în care un nod este trecut în clasa "arbore" i se schimbă **semnul** valorii memorate în tabloul **marc** (linia [5]).

#### 11.2.2.1. Analiza performanței metoda căutării "bazate pe prioritate"

- Analiza algoritmului de **căutare bazată pe prioritate** în determinarea **arborelui de acoperire minim** conduce la performanța  $O((n+a) \log_2 n)$ .
- **Motivația** este următoarea:
  - În procesul construcției arborelui sunt parcurse toate nodurile și toate arcele grafului.
  - Fiecare **nod** conduce la o inserție și fiecare **arc** la o eventuală modificare de prioritate în cadrul cozii bazate pe prioritate utilizate.
  - Presupunând că implementarea cozii bazate pe prioritate s-a realizat cu ajutorul **ansamblelor**, atunci atât inserția cât și modificarea se realizează în  $O(\log_2 n)$  pași.
  - În consecință performanța totală va fi  $O((n+a) \log_2 n)$ .

#### 11.2.2.2. Considerații referitoare la metoda căutării "bazate pe prioritate"

- Metoda căutării “bazate pe prioritate” are un pronunțat caracter de **generalitate**.

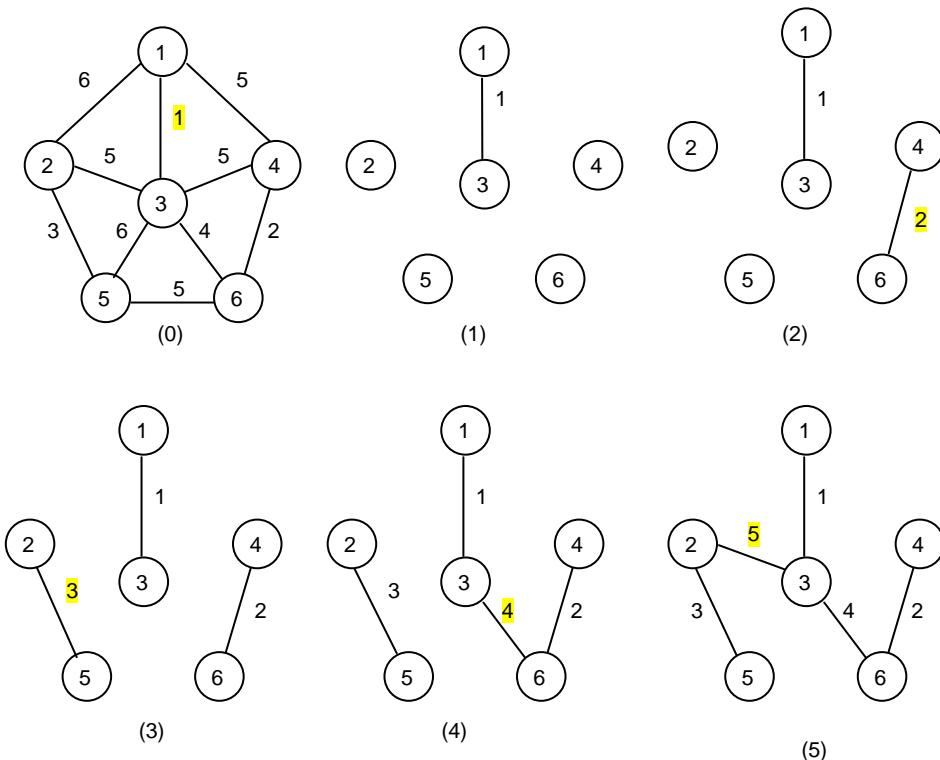
- Astfel, după cum se va vedea în continuare, pornind de la această metodă se poate dezvolta un algoritm care rezolvă **problema drumului minim**.
- De asemenea, pornind tot de la aceeași metodă va fi dezvoltat un algoritm care rezolvă aceleași probleme în cazul **grafurilor dense** cu un efort de calcul proporțional cu  $O(n^2)$ .
- Istorici lucrările au evoluat însă altfel.
  - În anul 1957 **Prim** publică algoritmul pentru determinarea **arborelui de acoperire minim**.
  - În anul 1959 **Dijkstra** publică algoritmul referitor la **determinarea drumului minim**.
- Pentru clarificare, s-a convenit ca:
  - (1) În cazul **grafurilor dense** cele două soluții să fie numite:
    - **Algoritmul lui Prim** pentru determinarea **arborelui de acoperire minim** al unui graf.
    - **Algoritmul lui Dijkstra** pentru determinarea **drumului minim** într-un graf.
  - (2) În cazul **grafurilor rare** algoritmul să poarte numele:
    - **Algoritm de căutare bazată pe prioritate**.
- După cum se va vedea, de fapt soluțiile propuse se întrepătrund și ele nu sunt decât cazuri particulare ale unui **algoritm generalizat de căutare bazat pe prioritate**.
- Este evident faptul că metoda **căutării "bazate pe prioritate"** este aplicabilă cu preponderență în cazul **grafurilor ponderate** considerând ponderile drept **priorități**.
- Cu toate acestea, această metodă poate fi aplicată și **grafurilor neponderate**.
- Într-un astfel de context, **căutarea bazată pe prioritate** poate **generaliza** tehniciile de traversare bazate pe **căutarea "în adâncime"** respectiv pe **căutarea "prin cuprindere"** prin atribuirea corespunzătoare de **priorități** nodurilor grafului.
  - Se reamintește faptul că **id** este o variabilă a cărei valoare se incrementează de la 1 la n pe măsură ce sunt procesate nodurile grafului în timpul execuției procedurii **CautaPrioritar** din secvența [11.2.2.a].
  - Variabila **id** poate fi utilizată în **atribuirea de priorități** nodurilor examineate, în baza convenției "**minim = prioritar**".
- Astfel, dacă în procesul de **căutare bazată pe prioritate**:
  - (1) Se consideră **prioritatea** unui nod egală cu **n-id** se obține **căutarea "în adâncime"**.
  - (2) Se consideră **prioritatea** unui nod egală chiar cu **id**, se obține **căutarea "prin cuprindere"**.
- În primul caz nodurile **nou întâlnite** au cea mai mare prioritate.
- În cel de-al doilea caz nodurile **cele mai vechi** adică cel mai devreme întâlnite, au cea mai mare prioritate.
- De fapt aceste atribuirile de priorități determină structura **coadă bazată pe prioritate** să se comporte ca o **stivă** respectiv ca și o **coadă normală**, structuri specifice celor două tipuri de parcurgeri.

### 11.2.3. Algoritmul lui Kruskal

- Fie graful conex  $G = (N, A)$  cu  $N = \{1, 2, \dots, n\}$  și cu funcția de cost  $c$  definită pe mulțimea arcelor  $A$ .
- O altă metodă de construcție a unui **arbore de acoperire minim**, este **algoritmul lui Kruskal**.
  - **Algoritmul lui Kruskal** pornește de la un graf  $T = (N, \Phi)$  care constă doar din cele  $n$  noduri ale grafului original  $G$ , dar care nu are nici un arc.
    - În această situație fiecare nod este de fapt o **componentă conexă** a grafului care constă chiar din nodul respectiv.
    - În continuare pornind de la mulțimea curentă a componentelor conexe, algoritmul selectează pe rând câte un **arc de cost minim** pe care îl adaugă componentelor conexe care cresc în dimensiune dar al căror număr se reduce.
    - În final rezultă o singură **componentă conexă** care este chiar **arborele de acoperire minim**.
  - Pentru a construi componente din ce în ce mai mari se examinează arcele din mulțimea  $A$  în **ordinea crescătoare a costului lor**.
    - Dacă arcul selectat conectează două noduri aparținând unor  **componente conexe distincte**, arcul respectiv este adăugat grafului  $T$ .
    - Dacă arcul selectat conectează două noduri aparținând unei **aceleeași componente conexe**, arcul este neglijat.
      - **Motivul:** introducerea sa ar conduce la apariția unui ciclu în respectiva componentă și în final la un ciclu în arborele de acoperire, lucru nepermis prin definiție.
    - Aplicând în manieră repetitivă acest procedeu, la momentul la care toate nodurile grafului aparțin unei singure componente conexe, algoritmul se termină și  $T$  reprezintă arborele de acoperire minim al grafului  $G$ .
  - Cu alte cuvinte algoritmul lui Kruskal pornește de la o **pădure** cu  $n$  arbori.
    - În fiecare din cei  $n-1$  pași, algoritmul combină doi arbori într-unul singur, utilizând ca legătură **arcul cu costul cel mai redus** curent.
    - Procedeul continuă până în momentul în care rămâne un singur arbore.
    - Aceasta este **arborele de acoperire minim**.
  - Spre exemplu considerând **graful ponderat** din fig.11.2.3.a (0)
    - În succesiunea (1) - (5) din cadrul aceleiasi figuri se prezintă maniera de determinare a unui arbore de acoperire minim al grafului în baza algoritmului lui Kruskal.
      - Ordinea în care se adaugă arcele rezultă din figură.
      - Inițial se adaugă arcele cu costurile 1, 2, 3 și 4, toate acceptate, întrucât nici unul dintre ele nu generează vreun ciclu.
      - Arcele (1, 4) și (3, 4) de cost 5 **nu** pot fi acceptate deoarece conectează noduri aparținând unei aceleleași componente (fig.11.2.3.a (d)) și conduc la

cicluri.

- În consecință se acceptă arcul (2, 3) de cost 5 care nu produce nici un ciclu încheind astfel construcția arborelui de acoperire minim.
- Ar putea fi selectat și arcul (5–6) de cost 5. Arborele de acoperire minim **nu** este unic.



**Fig.11.2.3.a.** Construcția unui arbore de acoperire minim pe baza algoritmului lui Kruskal

- Algoritmul lui Kruskal publicat în anul 1956 poate fi implementat în mai multe moduri.
- În continuare se prezintă un exemplu de implementare.

### 11.2.3.1. Exemplu de implementare a algoritmului lui Kruskal

- Algoritmul lui Kruskal poate fi implementat utilizând structura de date **mulțime de submulțimi** pe care sunt definiți operatorii **Uniune** și **Caută**, aşa cum apare ea prezentată în paragraful &9.6.
- Schița de principiu a unei astfel de implementări apare în secvența [11.2.3.1.a].

---

```
/*Implementarea algoritmului lui Kruskal - varianta
pseudocod C-like - se utilizează TDA Mulțime pe care sunt
definiți operatorii Uniune și Caută, TDA Coadă bazată pe
prioritate și TDA Mulțime*/
```

```
procedura KRUSKAL(Mulțime_De_Noduri N, Mulțime_De_Arce A,
Mulțime_De_Arce T)
```

```
    int ncomp; /*numărul curent de componente*/
    Coada_Bazata_Pe_Prioritate CPArce; /* coadă bazată pe
```

```

prioritate care păstrează arcele în vederea selecției
arcului minim*/
Mulțime_De_Submulțimi componente; /*mulțime de submulțimi
pe care sunt definiți operatorii Uniune și CautăComp*/
Tip_Nod x,y;
Tip_Arc a;
int compUrm; /*numele noii componente*/
int xComp,yComp; /*nume de componente*/

[1] Vid(T); /*face mulțimea T (arborele minim) vidă*/
/*initializează coada bazată pe prioritate pe coada vidă*/
[2] Initializeaza(CPArce);
[3] compUrm=0;
[4] nComp=(numărul de membri ai lui N);
/*initializează mulțimea de submulțimi, astfel încât
fiecare componentă să conțină un singur nod din N*/
[5] pentru (toate nodurile x aparținând lui N)
{
[6]     compUrm=compUrm+1; /*[11.2.3.1.a]*/
[7]     InitializeazaComp(compUrm,x,componente);
} /*pentru*/
/*initializează coada de priorități a arcelor*/
[8] pentru (toate arcele aparținând mulțimii A)
[9]     Insereaza(a,CPArce);
/*determină arborele de acoperire minim T*/
[10] cat timp ((nComp>1)&&(!Vid(CPArce)))
{
    /*cercetează arcul următor*/
[11]     a=Extrage(CPArce); /*se presupune că a=(x,y)*/
[12]     xComp=CautăComp(x,componente);
[13]     yComp=CautăComp(y,componente);
[14]     daca (xComp<>yComp)
    {
        /*a conectează două componente diferite*/
[15]         Uniune(xComp,yComp,componente);
[16]         nComp=nComp-1;
        /*adaugă arcul a arborelui minim reprezentat de
        mulțimea T*/
[17]         Adauga(a,T);
    } /*daca*/
} /*cat timp*/
/*KRUSKAL*/
-----
```

- Referitor la această implementare se fac următoarele **precizări**:
- (1) Mulțimea arcelor este structurată ca și o **coadă bazată pe priorități** denumită CParce.
  - În coada CParce se introduc inițial toate arcele grafului cu ajutorul operatorului **Insereaza** (cea de-a doua buclă **pentru**, liniile [ 8 ] și [ 9 ] din cadrul algoritmului).
  - Din coada CParce se extrage în fiecare pas al algoritmului cu ajutorul operatorului **Extrage**, arcul curent cu costul minim (linia [ 11 ]).
- (2) **Mulțimea componentelor conexe** este structurată ca și o **mulțime de submulțimi** denumită **componente** pe care sunt definiți următorii operatori specifici:

- (a) **Uniune**(componentă A, componentă B, Multime\_De\_Submulțimi C); – reunește componentele A și B ale mulțimii de submulțimi C. Rezultatul uniunii se va numi fie A fie B în mod arbitrar. Se observă diferența față de definiția utilizată în capitolul 9, unde C nu apare ca și parametru.
- (b) componentă **CautaComp**(nod x, Multime\_De\_Submulțimi C) – returnează numele acelei componente a lui C al cărei membru este nodul x. Operația va fi utilizată pentru a stabili dacă cele două noduri care determină un arc aparțin unei aceleasi componente conexe sau la componente diferite.
- (c) **InitializeazăComp**(componentă A, nod x, Multime\_De\_Submulțimi C) – crează componenta cu numele A a mulțimii de submulțimi C. Componenta A va conține numai nodul x.
- (3) **Arborele minim** este reprezentat ca și o mulțime de arce T care este structurată ca și un **TDA Mulțime clasic** peste care sunt definiți operatorii:
  - (a) **Vid**(Multime\_De\_Arce T) care crează mulțimea vidă T.
  - (b) **Adaugă**(Tip\_Arc a, Multime\_De\_Arce T) care adaugă arcul a mulțimii de arce T (linia [17]).
- **Algoritmul lui Kruskal** are ca rezultat construcția mulțimii T care cuprinde arcele care alcătuiesc **arborele de acoperire minim al grafului**.

### 11.2.3.2. Analiza performanței algoritmului lui Kruskal

- Analiza performanței **algoritmului lui Kruskal** se realizează în baza următoarelor constatări.
- Timpul de execuție al implementării **algoritmului lui Kruskal** este dependent de doi factori:
  - (1) Implementarea **cozii bazate pe prioritate**.
  - (2) Implementarea **mulțimii de submulțimi**.
- Astfel, dacă există a arce:
  - Sunt necesare  $O(a \log a)$  unități de timp pentru a insera toate arcele în coada bazată pe priorități (liniile [8] și [9]), dacă aceasta este implementată cu ajutorul **ansamblelor** sau al **arborilor binari echilibrați** spre exemplu.
- În fiecare iterație a buclei **cat timp** (linia [10]), determinarea arcului de cost minim (linia [11]) consumă în aceste condiții  $O(\log a)$  unități de timp.
  - În consecință dacă avem a arce **gestionarea cozii de priorități** consumă în cel mai rău caz  $O(a \log_2 a)$  unități de timp.
- Timpul total necesar execuției operațiilor **Uniune** (linia [15]) și **CautaComp** (liniile [12] și [13]) depinde de natura implementării structurii de date **mulțime de submulțimi**.
  - După cum s-a văzut în capitolul 9, există metode care obțin performanțe de ordinul  $O(a \log_2 a)$  pentru un total de a elemente.
- În **concluzie** algoritmul lui Kruskal poate fi implementat astfel încât să fie executat în  $O(a \log_2 a)$  unități de timp.

- **Cel mai defavorabil caz** este acela în care **graful nu este conex** și în consecință trebuie escăudate **toate arcele**.
- În situația în care **graful este conex, cel mai defavorabil caz** este acela în care arborele de acoperire minim constă din doi subarbore conecțiuni prin **cel mai lung arc al grafului**, arc care va fi determinat doar după examinarea tuturor arcelor din coada bazată pe prioritate.
- Pentru grafurile tipice, este de așteptat ca determinarea arborelui de acoperire minim (care conține  $n-1$  arce) să se termine înaintea examinării tuturor arcelor, respectiv înainte de a ajunge la cel mai lung arc.
  - Cu toate acestea și într-o astfel de situație timpul de execuție este proporțional cu  $a$ , deoarece construcția inițială a cozii bazate pe prioritate presupune examinarea (inserția) tuturor arcelor.

### 11.3. Drumul minim ("Shortest Path")

- Problema **drumului minim** se referă la a găsi acel drum într-un graf ponderat care conținează nodurile  $x$  și  $y$  aparținând grafului și care se bucură de proprietatea că **suma ponderilor arcelor** care-l compun este **minimă**.
- Chiar și în cazul **grafurilor neponderate**, situație în care **nu există ponderi**, problema prezintă încă interes.
  - Ea se referă la determinarea aceluiași drum care conține **numărul minim de arce** care conținează pe  $x$  și  $y$ .
  - În plus, se cunoaște deja un algoritm care rezolvă elegant această problemă, și anume **căutarea "prin cuprindere"**.
  - Se poate demonstra simplu prin **metoda inducției** că metoda **căutării "prin cuprindere"** care demarează cu nodul  $x$  va realiza în prima etapă vizitarea tuturor nodurilor la care se poate ajunge plecând de la  $x$  parcurgând un singur arc, apoi toate nodurile la care se poate ajunge de la  $x$  parcurgând două arce, și.a.m.d.
    - Practic, la un moment dat, înainte de a trece la un nod  $y$  care este conținut la nodul  $x$  prin  $k+1$  arce, vor fi vizitate în prealabil toate nodurile conținute prin  $k$  arce.
    - Astfel, în momentul în care s-a ajuns la nodul  $y$ , a fost determinat și **drumul minim**, deoarece nu există un alt drum mai scurt care conținează nodul  $x$  și  $y$ .
- În general, **drumul minim** poate implica oricare două noduri  $x, y$  aparținând unui graf.
- În acest context prezintă interes și **generalizarea** acestei probleme în sensul de a determina **drumurile minime care conțină un nod  $x$  și toate celelalte noduri ale grafului**.
- Această problemă este cunoscută sub denumirea de "**problema drumurilor minime cu origine unică**".
  - Ea poate fi rezolvată în contextul **grafurilor ponderate neorientate** pe baza unui algoritm cunoscut și anume parcurgând graful prin tehnica "**căutării bazate pe prioritate**".
  - O altă manieră de rezolvare a acestei probleme în contextul **grafurilor orientate**

va fi prezentată în capitolul următor.

- Dacă se desenează **drumurile minime** care conectează nodul  $x$  cu toate celelalte noduri ale grafului, cu siguranță nu se vor obține cicluri și în consecință rezultă un **arbore de acoperire corespunzător drumului minim**.
  - Fiecare nod al grafului conduce la un alt arbore de acoperire.
  - Astfel, pentru graful ponderat neorientat din figura 11.3.a. (a), arborii de acoperire corespunzători drumului minim, pentru nodurile A, G și M apar în aceeași figură, pozițiile (b), (c) respectiv (d).

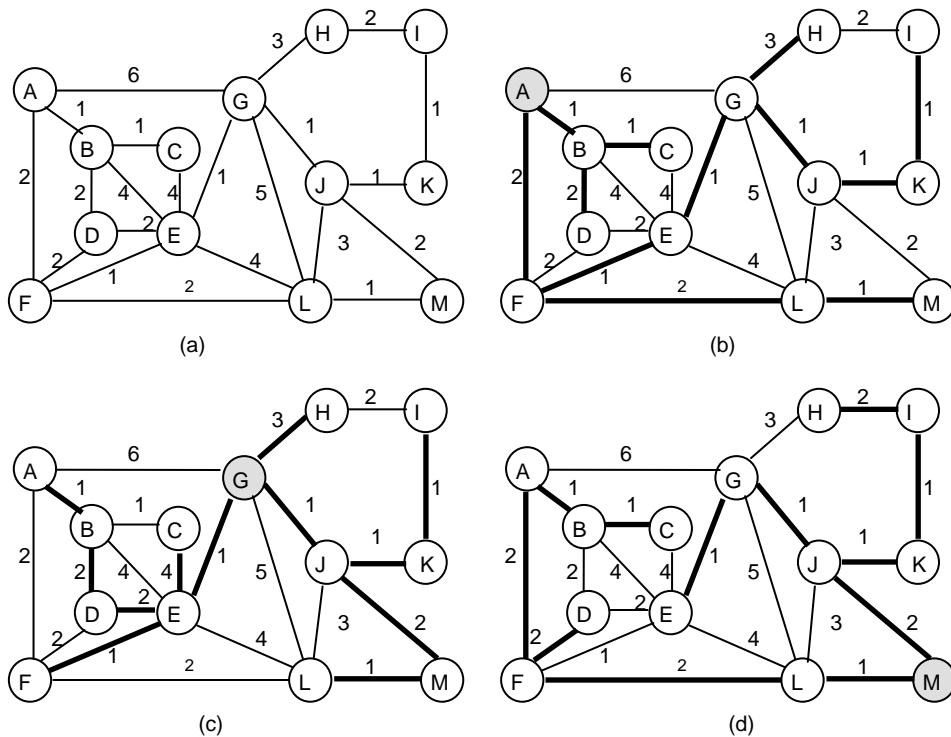


Fig.11.3.a. Arbori de acoperire corespunzători drumului minim

### 11.3.1. Determinarea drumurilor minime cu origine unică corespunzătoare unui nod al unui graf prin tehnica căutării "bazate pe prioritate"

- Problema determinării drumurilor minime corespunzătoare unui nod precizat al unui graf ponderat se reduce de fapt la determinarea **arborelui de acoperire corespunzător drumului minim** care are drept rădăcină **nodul** în cauză.
- Această problemă poate fi soluționată într-o manieră principal identică cu determinarea **arborelui de acoperire minim**.
- Astfel, **arborele de acoperire corespunzător drumului minim** pentru nodul  $x$ , se poate construi adăugând în fiecare pas, nodul din clasa “vecinătate” care este **cel mai “apropiat” nodului origine  $x$** .
  - Se reamintește faptul că procedura **ArboreMinim** (secvența [11.2.2.a]), care construiește **arborele de acoperire minim (de cost minim)**, adăuga la fiecare pas arborelui minim, acel nod din clasa “vecinătate” care era **cel mai “apropiat” arborelui** la momentul respectiv, adică nodul cu costul curent minim.

- În situația de față, pentru a determina nodul curent cel mai “**apropiat**” de nodul  $x$ , se utilizează tot tabloul  $marc$ .
- Pentru fiecare nod  $k$  al arborelui de acoperire,  $marc[k]$  memorează **distanța** de la nodul  $k$  la nodul  $x$ , parcurgând **drumul minim**, care de altfel este memorat de **arborele de acoperire**.
- În momentul în care nodul  $k$  este adăugat arborelui, se actualizează multimea “**vecinătate**” parcurgând lista de adiacențe a lui  $k$ .
  - Pentru fiecare nod  $t^{\wedge}.nume$  aparținând listei de adiacențe a lui  $k$ , cea mai scurtă distanță până la nodul  $x$ , trecând prin nodul  $k$ , este  $marc[k] + t^{\wedge}.cost$ .
  - Această remarcă conduce la implementarea imediată a algoritmului de determinare al **arborelui de acoperire corespunzător drumului minim**.
  - Pur și simplu se utilizează procedura **ArboreMinim** (secvența [11.2.2.a]) bazată pe parcurgerea grafului prin **tehnica căutării “bazate pe prioritate”** în care se consideră prioritatea fiecărui nod  $t$  examinat, ca fiind egală cu  $marc[k] + t^{\wedge}.cost$ .
    - Aceasta presupune modificarea liniilor [10] și [11] din procedura menționată după cum urmează.

```
-----[10] if Actualizeaza(q, t^{\wedge}.nume, mark[k]+t^{\wedge}.cost)
[11]     marc[t^{\wedge}.nume]==(mark[k]+t^{\wedge}.cost); /*nodul
           t^{\wedge}.nume trece în clasa "vecinătate"*/
-----
```

- Noua formă a procedurii redenumită **DrumMinim** apare în secvența [11.3.1.a].

```
-----/*Determinarea unui arbore de acoperire corespunzător
drumului minim al unui graf prin metoda căutării bazate pe
prioritate - varianta pseudocod - se utilizează TDA Coadă
Bazată pe Prioritate */
```

```
/*structuri de date*/
typedef struct Tip_Nod* Ref_Tip_Nod;

/*structura unui nod al listei de adiacențe*/
typedef struct Tip_Nod
{
    int nume;
    int cost;
    Ref_Tip_Nod urm;
} Nod;

Ref_Tip_Nod StrAdj[MaxNod]; /*structura de adiacențe*/
int id,k;
int marc[MaxNod]; /*tablou pentru evidența nodurilor*/
int parinte[MaxNod]; /*arborele de acoperirie minim*/
Coadă_Bazata_Pe_Prioritate q;
```

```
procedura CautaPrioritar(int k)
    Ref_Tip_Nod t;
```

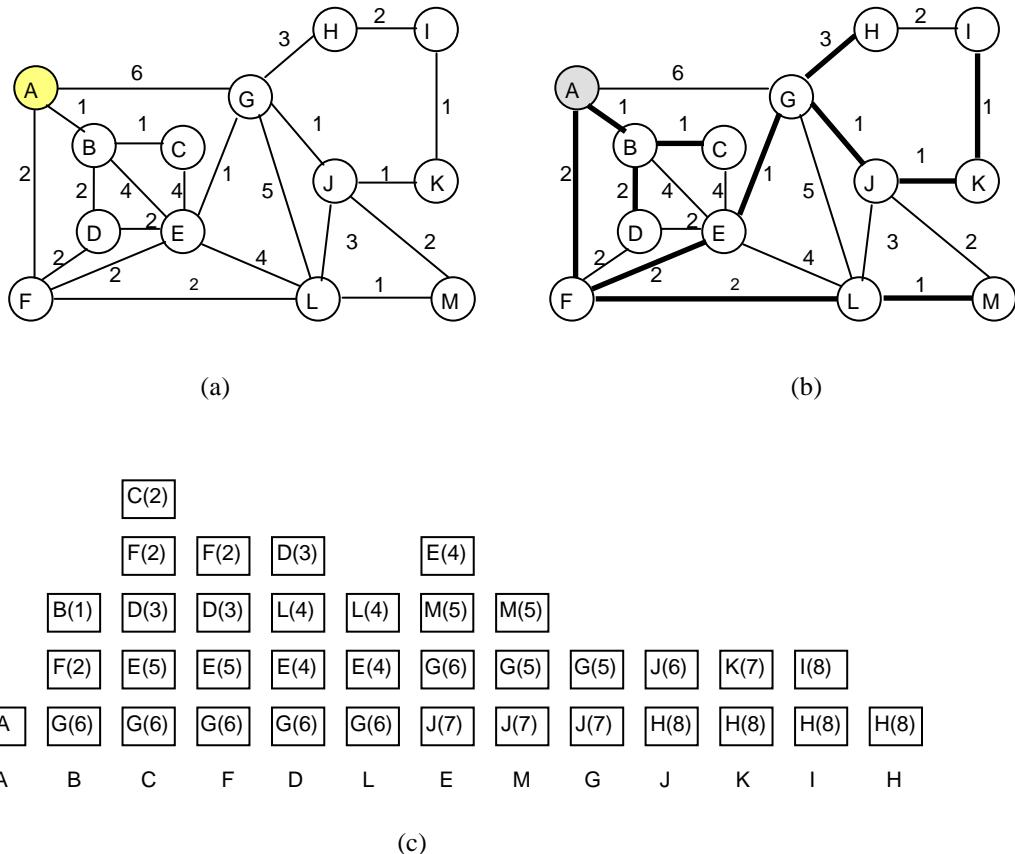
```

[1]  daca Actualizeaza(q,k,nevazut)
        parinte[k]=0; /*amorsare proces de căutare*/
[2]  repeta
[3]      id=id+1;
[4]      k=Extrage(q);
[5]      marc[k]=-marc[k]; /*k trece în clasa "arbore"*/
[6]      daca (marc[k]==nevazut)
[7]          marc[k]=0; /*nodul origine*/
[8]          t=StrAdj[k];
[9]          cat timp (t<>null)                  /*[11.3.1.a]*/
[10]         daca (marc[t^.nume]<0) /*nod nevizitat sau în
[11]             coadă*/
[12]             daca Actualizeaza(q,t^.nume,mark[k]+t^.cost)
[13]                 marc[t^.nume]=-(mark[k]+t^.cost); /*nodul
[14]                     t^.nume trece în clasa "vecinată"*/
[15]                     parinte[t^.nume]=k; /*înregistrare părinte*/
[16]                     /*daca*/
[17]                     t=t^.urm;
[18]
[19]         pana cand Vid(q);
[20]
/*CautaPrioritar*/

void DrumMinim;
/*programul principal pentru construcția unui arbore de
acoperire corespunzător drumului minim*/

id= 0;
Initializeaza(q);
pentru (k= 1 la N)
    marc[k]= -nevazut;
pentru (k= 1 la N)
    daca (marc[k]== -nevazut)
        CautaPrioritar(k);
/*DrumMinim*/
-----
```

- Utilizând procedura **DrumMinim** modificată pentru construcția arborelui de acoperire corespunzător drumului minim pentru graful din figura 11.3.1.a (a) considerând nodul A drept origine, se obține arborele de acoperire (b) iar evoluția conținutului cozii bazate pe prioritate apare în aceeași figură (c).
- Reprezentarea "indicator spre părinte" a arborelui construit apare în figura 11.3.1.b.
- Astfel, în graful menționat din figura 11.3.a. (a) spre exemplu, cel mai scurt drum de la nodul A la nodul H are lungimea totală 8 (memorată în `mark[8]` care este intrarea pentru nodul H) și pornește de la nodul A trece prin nodurile F, E, G și ajunge la nodul H (tabloul `nume(parinte[k])`).
  - Acest drum este efectiv memorat în tabloul `părinte` și poate fi reconstituit în sens invers urmărind traseul precizat de părintele fiecărui nod începând cu H (poziția 8) obținându-se succesiunea H, G, E, F, A.



**Fig.11.3.1.a.** Conținutul cozii bazate pe prioritate pentru construcția arborelui de acoperire corespunzător drumului minim

nume	A	B	C	D	E	F	G	H	I	J	K	L	M
k	1	2	3	4	5	6	7	8	9	10	11	12	13
nume(parinte[k])	0	A	B	B	F	A	E	G	K	G	I	F	L
marc[k]	0	1	2	3	4	2	5	8	6	7	4	5	

**Fig.11.3.1.b.** Reprezentarea arborelui de acoperire corespunzător drumului minim

- Această metodă de determinare a **drumurilor minime cu origine unică** corespunzătoare unui nod al unui graf, este aplicabilă **grafurilor rare** și ea se încadrează în performanță precizată deja pentru algoritmul **ArboreMinim**, respectiv  $O((n+a) \log n)$ .
- Algoritmul prezentat în acest paragraf este cunoscut și sub numele de **algoritmul lui Dijkstra** pentru determinarea drumului minim, algoritm care va fi reluat într-o manieră mai aprofundată în capitolul destinat grafurilor orientate.

## 11.4. Arbori de acoperire și drumuri minime în grafuri dense. Generalizarea algoritmului căutării bazate pe prioritare

- Este cunoscut faptul că în marea majoritate a aplicațiilor, în cazul **grafurilor dense**, reprezentarea cea mai eficientă este cea bazată pe **matrice de adiacențe**.
- Pentru un graf ponderat implementat printr-o matrice de adiacențe, o modalitate uzuală de a reprezenta **coada bazată pe prioritare**, este aceea de a utiliza în acest scop un **tablou neordonat** [Se88].
  - Această reprezentare permite obținerea unei performanțe de ordinul  $O(n^2)$  pentru orice algoritm de traversare al grafului prin tehnica **căutării bazate pe prioritare**.
  - Performanța este posibilă dacă se combină bucla de **actualizare a priorităților** cu bucla de **determinare a elementului minim**.
    - De fiecare dată când se extrage un nod din clasa “vecinătate”, se procesează toate nodurile adiacente actualizându-li-se, dacă este cazul, prioritățile și căutând **ponderea minimă**.
    - În consecință determinarea **arborilor de acoperire minimi** și a **drumurilor minime** în **grafuri dense** utilizând **căutarea “bazată pe prioritare”** se poate realiza într-un interval de timp proporțional cu  $O(n^2)$  ( $n$  numărul de noduri).
  - În secvența [11.4.a] apare **un exemplu generalizat** de implementare a **tehnicii de căutare “bazată pe prioritare”**.
    - Se precizează faptul că această procedură **CautPrioritar** a fost dezvoltată pornind de la procedura **ArboreMinim** (secvența [11.2.2.a]) cu următoarele particularități:
      - (1) Graful se consideră reprezentat prin **matricea de adiacențe A** care memorează ponderile arcelor, adică prioritățile nodurilor vizate.
      - (2) **Coada bazată pe priorități** se păstrează în tabloul **marc** iar operatorii definiți pe această structură se implementează **direct** în cadrul algoritmului.
      - (3) Ca și în cazul secvenței [11.2.2.a] semnul unui element aparținând tabloului precizează dacă nodul corespunzător (precizat prin indicele de intrare în tabloul **marc**) este în arbore dacă are semnul plus respectiv în coada bazată pe priorități, dacă are semnul minus.
        - Inițial, toate nodurile aparțin clasei “neîntâlnite” și ele se plasează în coada bazată pe priorități respectiv li se alocă valoarea “-nevăzut” în tabloul **marc**.
        - Pentru a modifica prioritatea unui nod, pur și simplu se introduce noua prioritățe în intrarea corespunzătoare nodului din tabloul **marc**.
        - Pentru a extrage nodul cu cea mai înaltă prioritate, se balează tabloul **marc** și se caută poziția care memorează cea mai mică valoare negativă, valoare care se complementează.
        - Nodul în cauză este extras astfel din coada bazată pe prioritate și introdus în arbore (părinte[t]=k).
      - (4) Se precizează faptul că valoarea “nevăzut” trebuie să fie cu ceva mai mică decât valoarea maximă întreagă reprezentabilă, deoarece un

număr cu 1 mai mare decât valoarea “nevăzut” este folosit ca și fanion pentru determinarea minimului, iar valoarea negativă a acestui număr trebuie să fie reprezentabilă.

- Tabloul marc se prelungeste spre stânga cu componenta  $marc[0]$  utilizată pe post de **fanion**.

---

**/\*Exemplu generalizat de implementare a tehnicii de căutare bazată pe prioritate” - varianta pseudocod\*/**

```

/*structuri de date*/
float A[DimMax,DimMax]; /*matricea de adiacențe (costuri)*/
float marc[DimMax]; /*tabloul marc - coada bazată pe
prioritate*/
int parinte[DimMax]; /*tabloul părinte - arborele de căutare
bazată pe prioritate*/
int N; /*numărul curent de noduri*/

procedura CautPrioritar;
    int k,min,t;

    /*inițializare structuri de date*/
    pentru (k=1 la N)
        marc[k]=-nevazut;                                /*[11.4.a]*/
        parinte[k]=0;
    □
        marc[0]=-(nevazut+1); /*fanion*/
        min=1;
    repeta
        k=min; marc[k]=-marc[k]; min=0; /*trece nodul k
                                         în clasa arbore*/
        daca marc[k]=nevazut
            marc[k]=0; /*nodul origine*/
        pentru (t=1 la N)
            daca (marc[t]<0)
                daca ((A[k,t]>>0) && (marc[t]<(-prioritate)))
                    marc[t]=-prioritate; /*nodul în coada BP*/
                    parinte[t]=k; /*înregistrare părinte*/
                □ /*daca*/
                daca (marc[t]>marc[min])
                    min=t;
                □ /*daca*/
            □ /*pentru*/
        pana cand (min=-0);
    □ /*repeta*/
/*CautPrioritar*/

```

---

- Caracterul de **generalitate** al acestui algoritm rezultă din interpretarea valorii “prioritate”:
  - (1) Dacă în matricea de adiacențe se memorează ponderi și dacă pe post de prioritate se utilizează valoarea  $A[k, t]$  se obține **algoritmul lui PRIM** de determinare a **arborelui de acoperire minim** (de cost minim).
  - (2) Dacă se utilizează ca și prioritate valoarea  $marc[k]+A[k, t]$  se obține **algoritmul lui Dijkstra** pentru **problema drumului minim**.

- Presupunând că se utilizează variabila `id` care memorează numărul nodului curent vizitat:
  - (3) Dacă pe post de prioritate se utilizează valoarea `n-id` (`n` este numărul de noduri), se obține **căutarea “în adâncime”**.
  - (4) Dacă pe post de prioritate se utilizează chiar `id` se obține **căutarea “prin cuprindere”**.
- De fapt procedurile **ArboreMinim** (secvența [11.2.2.a]) și **CautPrioritar** (secvența [11.4.a]) realizează în principiu același lucru cu următoarele diferențe:
  - (1) Procedura **ArboreMinim** se aplică mai eficient **grafurilor rare** (implementate prin structuri de adiacențe) în timp ce **CautPrioritar** este mai eficientă în cazul **grafurilor dense** (implementate ca matrici de adiacențe).
  - (2) **Coadă bazată pe prioritate** este implementată în primul caz printr-o metodă avansată (de exemplu structura ansamblu) iar în cel de-al doilea caz printr-un tablou neordonat.

#### **11.4.1. Analiza algoritmului generalizat de căutare bazată pe prioritate**

- Performanța algoritmului generalizat de căutare bazată pe prioritate este  $O(n^2)$ .
- Această afirmație rezultă imediat din inspecția secvenței [11.4.a].
  - De fiecare dată când este vizitat un nod, se realizează o trecere prin toate cele  $n$  intrări ale rândului corespunzător nodului din matricea de adiacențe cu scop dublu:
    - (1) De a actualiza ponderile tuturor nodurilor învecinate.
    - (2) De a găsi elementul cu prioritate maximă din coada bazată pe prioritate.
- Din punctul de vedere al numărului de arce, algoritmul este liniar, adică performanța sa este  $O(a)$ .

#### **11.5. Considerente referitoare la performanțele comparate ale algoritmilor de determinare a arborilor de acoperire minimi**

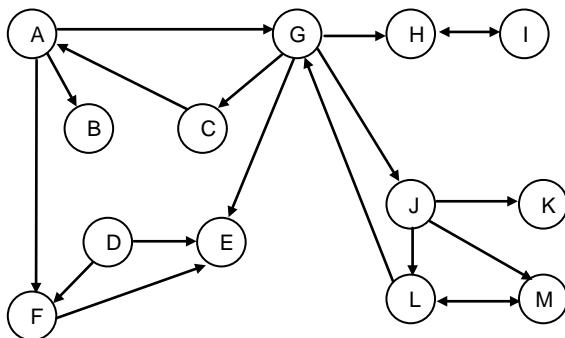
- În cadrul acestui capitol au fost dezvoltăți mai mulți algoritmi care rezolvă **problema arborelui de acoperire minim** și corelat cu aceasta și **problema drumului minim**.
- Este vorba despre:
  - Algoritmul lui Prim
  - Metoda căutării “bazate pe prioritate”
  - Algoritmul lui Kruskal.
- Fiecare din acești algoritmi este mai potrivit pentru o anumită categorie de grafuri.
- Astfel, referitor la cel mai defavorabil caz se pot preciza următoarele:
  - **Căutarea ”bazată pe prioritate”** obține în cel mai defavorabil caz o performanță de ordinul  $O((a+n) \log n)$ .

- Performanța **algoritmului lui Prim** în cel mai defavorabil caz este  $O(n^2)$ .
- Performanța **algoritmului lui Kruskal** în cel mai defavorabil caz este  $O(a \log a)$ .
- În practică însă **nu** este recomandabil ca selecția algoritmului utilizat să se realizeze după criteriul “**cel mai defavorabil caz**” din cel puțin două motive:
  - (1) Cazul cel mai defavorabil apare de regulă în practică cu o probabilitate redusă.
  - (2) Funcție de situația reală, algoritmii care în cazul cel mai defavorabil obțin performanțe mai slabe, pot conduce în exploatarea curentă la performanțe superioare algoritmilor care în situații extreme obțin performanțe mai bune.
- În final se pot formula următoarele **concluzii**:
  - (1) **Căutarea “bazată pe prioritate” și metoda lui Kruskal** rulează în intervale de timp proporționale cu **numărul de arce**, pentru marea majoritate a grafurilor care apar în practică.
    - În primul rând, deoarece multe din arcele cercetate **nu** necesită ajustarea cozii bazate pe priorități, ajustare care necesită  $\log_2 n$  pași.
    - În al doilea rând, deoarece cel mai lung arc al arborelui de acoperire minim este probabil suficient de scurt pentru ca până la determinarea lui să nu fie extrase prea multe arce din coada bazată pe priorități.
  - (2) Pentru **grafuri rare**, este de așteptat ca metoda **căutării “bazate pe prioritate”** să fie mai rapidă deoarece este de presupus că ea va gestiona cozii bazate pe prioritate scurte.
  - (3) **Algoritmul lui Prim**, rulează într-un timp proporțional cu **numărul de arce** în cazul **grafurilor dense**, **nu** însă și în cazul **grafurilor rare**.

## 12. Grafuri orientate

- După cum s-a mai precizat, **grafurile orientate** sunt grafuri în care arcele care conectează nodurile au un singur sens.
  - Această restricție face mult mai dificilă evidențierea și exploatarea diverselor proprietăți ale grafurilor de acest tip.
  - Prelucrarea unui astfel de graf este identică cu o călătorie cu mașina într-un oraș cu foarte multe străzi cu sens unic sau cu o călătorie cu avionul într-o țară în care liniile aeriene nu sunt dus-întors.
  - În astfel de situații a ajunge dintr-un loc în altul poate reprezenta o adevărată problemă.
- De multe ori arcele orientate reflectă anumite tipuri de **relații de precedență** în aplicația pe care o modeleză.
  - Spre exemplu un **graf orientat** poate fi utilizat ca model pentru o **linie de fabricație**:
    - Nodurile corespund diverselor operații care trebuie executate.
    - Arcele orientate ordonează temporal structura. Astfel, un arc de la nodul  $x$  la nodul  $y$  precizează faptul că operațiunea corespunzătoare nodului  $x$  trebuie executată **înaintea** celei corespunzătoare nodului  $y$ .
    - În acest context, problema care se pune este aceea de a executa toate aceste operații, fără a eluda niciuna dintre relațiile de precedență impuse.
- După cum s-a menționat în capitolul 10, **reprezentările grafurilor orientate** sunt simple extensii (de fapt simplificări) ale reprezentărilor grafurilor neorientate. Astfel:
  - În reprezentarea bazată pe **liste de adiacențe**, **fiecare arc apare o singură dată**: arcul de la nodul  $x$  la  $y$  este reprezentat prin prezența nodului  $y$  în lista de adiacențe a lui  $x$ .
  - În reprezentarea bazată pe **matrice de adiacențe**, arcul de la nodul  $x$  la  $y$  se marchează tot **o singură dată**, prin valoarea "adevărat" a elementului matricii de adiacențe situat în linia  $x$ , coloana  $y$  (nu și a elementului situat în linia  $y$ , coloana  $x$ ).
- În figura 12.1.a apare un exemplu de **graf orientat**
  - Graful constă din arcele AG, AB, CA, LM, JM, JL, JK, ED, DF, NI, FE, AF, GE, GC, HG, GJ, LG, IH, ML.
  - **Ordinea** în care nodurile apar la specificarea arcelor este **semnificativă**.
    - Notația AG precizând un arc care își are sursa în nodul A și destinația în nodul G.

- Este însă posibil ca între două noduri să existe două arce având sensuri opuse (exemplu HI și IH, respectiv LM și ML).



**Fig.12.1.a.** Exemplu de graf orientat.

- Pornind de la această precizare este ușor de observat faptul că un **graf neorientat** poate fi asimilat cu **graful orientat** identic ca și structură, care conține pentru fiecare arc al grafului neorientat două arce opuse.
  - În acest context unei dintre algoritmii dezvoltăți în acest capitol, pot fi considerați **generalizări** ale **algoritmilor** din capitolele anterioare.
- În cadrul acestui capitol vor fi prezentate unele dintre problemele specifice acestei categorii de grafuri precum și o serie de algoritmi consacrați care le soluționează.
- Este vorba despre:
  - (1) Problema drumurilor minime cu origine unică.
  - (2) Problema drumurilor minime corespunzătoare tuturor perechilor de noduri.
  - (3) Închiderea tranzitivă.
  - (4) Grafuri orientate aciclice.
  - (5) Componente conectate în grafuri orientate.
  - (6) Problema rețelelor de curgere.
  - (7) Problema potrivirilor.

## 12.1. Problema drumurilor minime cu origine unică ("Single-Source Shortest Path Problem")

- O primă problemă care se abordează în contextul grafurilor orientate este următoarea:
  - Se consideră un graf orientat  $G = (N, A)$  în care fiecare arc are **o pondere pozitivă** și pentru care se precizează un nod pe post de **"origine"**.
  - Se cere să se determine **costul** celui mai scurt drum de la origine la oricare alt nod al grafului.
    - Conform celor deja precizate, **costul** unui drum este **suma ponderilor arcelor** care formează drumul.

- Această problemă este cunoscută sub numele de **problema drumurilor minime cu origine unică** (“single source shortest path problem”).
- Desigur o abordare mai naturală, ar fi aceea care-și propune să determine **cel mai scurt drum de la o origine la o destinație precizată**.
  - Această problemă are același grad de dificultate ca și problema anterioară care de fapt o **generalizează**.
  - Astfel, pentru a rezolva această ultimă problemă, este suficient ca execuția algoritmului care determină drumurile minime cu origine unică să fie oprită în momentul în care se ajunge la destinația precizată.
- Exact ca și în cazul grafurilor ponderate neorientate, ponderea poate avea semnificația fizică a unui cost, a unei valori, a unei lungimi, a unui timp, etc.
- Intuitiv, graful  $G$  poate fi asimilat spre exemplu, cu o hartă a traseelor aeriene ale unui stat în care:
  - Fiecare nod reprezintă un oraș.
  - Fiecare arc  $(x, y)$  reprezintă o legătură aeriană de la orașul  $x$  la orașul  $y$ .
  - Ponderea unui arc  $(x, y)$  poate fi timpul de zbor sau prețul biletului.
- Desigur, ca model ar putea fi utilizat și un graf neorientat deoarece ponderea arcului  $(x, y)$  trebuie să fie în principiu aceeași cu cea a arcului  $(y, x)$ .
  - În realitate pe de o parte **nu** toate traseele aeriene sunt dus-întors, iar pe de altă parte dacă ele sunt dus-întors, timpul de zbor s-ar putea să difere în cele două sensuri.
  - În orice caz, considerând arcele  $(x, y)$  și  $(y, x)$  cu ponderi identice, aceasta nu conduce la o simplificare a rezolvării problemei.

### 12.1.1. Algoritmul lui Dijkstra

- Pentru a rezolva problema drumurilor minime cu origine unică, o manieră clasică de abordare o reprezintă utilizarea unui algoritm bazat pe tehnica “**greedy**” adesea cunoscut sub denumirea de “**algoritmul lui Dijkstra**”
  - Acest algoritm a fost publicat de către **E.W. Dijkstra** în anul 1959.
- Algoritmul se bazează pe o **structură de date mulțime**  $M$  care conține nodurile pentru care cea mai scurtă distanță la nodul origine este deja cunoscută.
  - Inițial  $M$  conține numai nodul **origine**.
  - În fiecare pas al execuției algoritmului, se adaugă mulțimii  $M$  un nod  $x$  care **nu** aparține încă mulțimii și a cărui distanță de la origine este cât mai scurtă posibil.
  - Presupunând că toate arcele au ponderi pozitive, întotdeauna se poate găsi un **drum minim** care leagă originea de nodul  $x$ , drum care trece **numai** prin nodurile conținute de  $M$ .
    - Un astfel de drum se numește “**drum special**”.
  - Pentru înregistrarea **lungimii drumurilor speciale** corespunzătoare nodurilor grafului se utilizează un tablou  $D$  care este actualizat în fiecare pas al algoritmului.

- În momentul în care  $M$  include toate nodurile grafului, toate drumurile sunt speciale și în consecință, tabloul  $D$  memorează cea mai scurtă distanță de la origine la fiecare nod al grafului.
  - Schița de principiu a **algoritmului lui Dijkstra** apare în secvența [12.1.1.a].
- 

**procedura** Dijkstra;

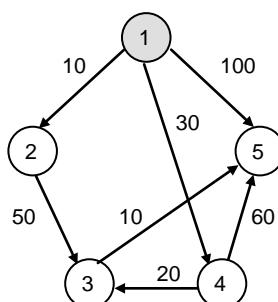
```

/*Determină costurile celor mai scurte drumuri care
conectează nodul 1, considerat drept origine, cu toate
celelalte noduri ale unui graf orientat. Distanțele se
pastrează în tabloul D*/
[1] M=[1]; /*nodul origine*/
    /*initializarea tabloului de distanțe D*/
[2] pentru (i=2 la n)
[3]   D[i]=COST[1,i];           /*[12.1.1.a]*/
    /*determinarea drumurilor minime*/
[4] pentru (i=1 la n-1)
[5]   *alege un nod x aparținând mulțimii N-M astfel
      încât D[x] să fie minim;
[6]   *adaugă pe x lui M;
    /*actualizare distanțe*/
[7]   pentru (fiecare nod y din mulțimea N-M)
[8]     D[y]= min(D[y], D[x]+COST[x,y])
    /*pentru*/
/*Dijkstra*/

```

---

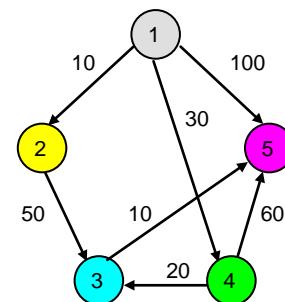
- Referitor la algoritmul lui **Dijkstra**, se presupune că:
  - Se dă un **graf orientat ponderat**  $G=(N, A)$  unde  $N=\{1, 2, 3, \dots, n\}$ .
  - Nodul 1 este considerat drept origine.
  - COST este un tablou cu două dimensiuni unde  $COST[i, j]$  reprezintă costul deplasării de la nodul  $i$  la nodul  $j$  pe arcul  $(i, j)$ .
    - Dacă arcul  $(i, j)$  nu există, se presupune că  $C[i, j]$  este  $\infty$ , respectiv are o valoare **mai mare decât orice cost**.
  - La fiecare iterație a algoritmului, tabloul  $D[i]$  conține lungimea drumului special minim curent de la origine la nodul  $i$ .
- Pentru exemplificare, se prezintă execuția algoritmului lui Dijkstra pentru graful orientat din figura 12.1.1.a.



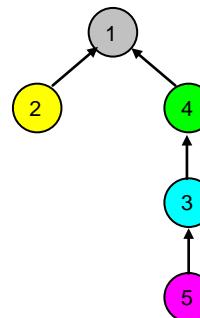
**Fig.12.1.1.a.** Graf orientat ponderat

- Inițial  $M = \{1\}$ ,  $D[2] = 10$ ,  $D[3] = \infty$ ,  $D[4] = 30$  și  $D[5] = 100$ .
- În prima iterație a buclei **pentru** din liniile [4]–[8], se selectează  $x=2$  ca fiind nodul de distanță minimă din  $D$ .
- În continuare se face  $D[3]=\min(\infty, 10+50)=60$ .
  - $D[4]$  și  $D[5]$  nu se modifică deoarece în ambele cazuri, drumul direct care conectează nodurile respective cu originea este mai scurt decât drumul care trece prin nodul 2.
- În continuare, modul în care se modifică tabloul  $D$  după fiecare iterație a buclei **pentru** mai sus precizate apare în figura 12.1.1.b.

Iterația	M	x	D[2]	D[3]	D[4]	D[5]
Initial	{1}	-	10	$\infty$	30	100
(1)	{1,2}	2	10	60	30	100
(2)	{1,2,4}	4	10	50	30	90
(3)	{1,2,4,3}	3	10	50	30	60
(4)	{1,2,4,3,5}	5	10	50	30	60



x	1	2	3	4	5
Parinte[x]	0	1	4	1	3



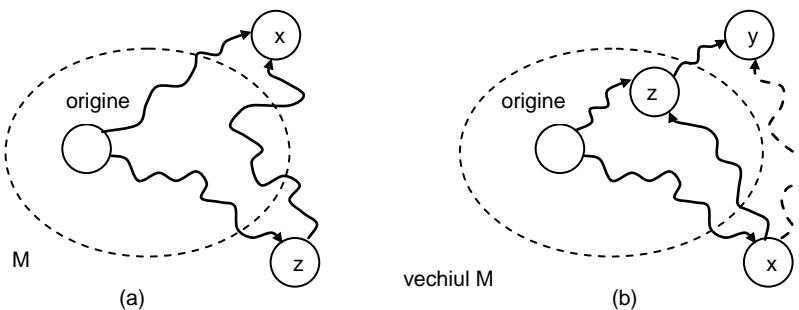
**Fig.12.1.1.b.** Determinarea drumurilor minime cu origine unică în baza algoritmului lui Dijkstra

- În vederea reconstrucției drumurilor minime de la origine la fiecare nod al grafului, se utilizează un alt tablou denumit **Părinte**.
  - În acest tablou,  $\text{Părinte}[x]$  memorează nodul care precede nodul  $x$  în cadrul drumului minim, adică memorează tatăl nodului  $x$ .
  - Tabloul **Părinte** se inițializează cu  $\text{Părinte}[i]=1$  pentru toate valorile lui  $i \neq 1$ .
  - Tabloul **Părinte** poate fi actualizat după linia [8] în procedura Dijkstra din secvența [12.1.1.a].
    - Astfel, dacă în linia [8],  $D[x]+\text{COST}[x,y] < D[y]$ , atunci se face  $\text{Părinte}[y]=x$ .

- După terminarea procedurii, drumul la fiecare nod poate fi reconstituit în sens invers mergând pe înlățuirile indicate de tabloul Părinte.
- Astfel, pentru graful orientat din fig. 12.1.1.a, tabloul Părinte conține valorile precizate în figura 12.1.1.b.
  - Pentru a găsi spre exemplu, drumul minim de la nodul 1 la nodul 5 al grafului, se determină predecesorii (părinții) în ordine inversă începând cu nodul 5.
  - Astfel se determină 3 ca predecesorul lui 5, 4 ca predecesor al lui 3, 1 ca predecesor al lui 4.
  - În consecință drumul cel mai scurt de la nodul 1 la nodul 5 este 1 , 4 , 3 , 5 iar lungimea sa 6 0 este memorată în D[ 5 ] .
- În aceeași figură apare reprezentat și **arborele de acoperire corespunzător drumurilor minime cu origine unică** atașat grafului din figura 12.1.1.a.

## 12.1.2. Demonstrarea funcționalității algoritmului lui Dijkstra

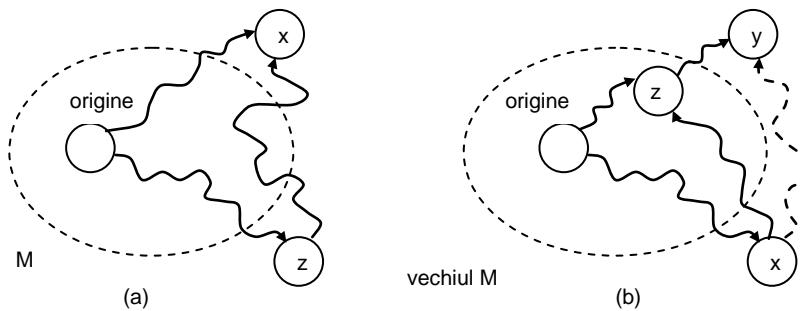
- Algoritmul lui Dijkstra este un exemplu de tehnică “**greedy**” în sensul că ceea ce apare local ca fiind cea mai bună activitate de realizat la un moment dat, reprezintă de fapt cel mai bun lucru de realizat la nivelul întregului la acel moment.
  - În acest caz, cel mai bun lucru de realizat la nivel local este acela de a determina distanța la acel nod  $x$  care este situat în afara mulțimii  $M$  și care are **cel mai scurt drum special** la momentul considerat.
- Pentru a demonstra că în această situație **nu** poate exista un alt drum nespecial mai scurt de la nodul  $x$  la origine decât drumul special care cuprinde **numai** noduri din  $M$ , se va utiliza **metoda reducerii la absurd**.
  - Se presupune că **există** un drum ipotetic nespecial, mai scurt decât drumul de la origine la nodul  $x$ , care întâi părăsește mulțimea  $M$ , atinge nodul  $z$ , iar apoi (probabil) intră șiiese din  $M$  de câteva ori înainte de a ajunge la  $x$  (fig.12.1.2.a. (a)).



**Fig.12.1.2.a.** Drum minim ipotetic și drum special minim imposibil

- Dacă acest drum este mai scurt decât drumul special care leagă originea de  $x$ , atunci segmentul inițial al drumului care leagă originea de  $z$ , este un drum special la  $z$ , mai scurt decât cel mai scurt drum special la  $x$ .
  - Trebuie subliniată în acest context importanța faptului că valorile ponderilor sunt **pozitive**.

- Fără această restricție, presupunerea avansată poate fi incorectă după cum întreaga funcționare a algoritmului lui Dijkstra poate fi incorectă.
- În acest caz însă, în linia [5] a algoritmului din secvența [12.1.1.a] ar fi fost ales nodul  $z$  și nu nodul  $x$ , deoarece  $D[z] < D[x]$  conform celor presupuse.
- Întrucât acest lucru **nu** s-a întâmplat, rezultă că **nu** există un astfel de nod  $z$  și că ipoteza formulată conduce la o contradicție.
- Pentru a completa demonstrația funcționalității algoritmului lui Dijkstra, mai trebuie verificat dacă în fiecare moment,  $D[y]$  reprezintă într-adevăr **cea mai scurtă distanță** a unui drum special la nodul  $y$ .
- Esența acestei argumentații rezidă în observația că atunci când se adaugă un nou nod  $x$  mulțimii  $M$  în linia [6] a secvenței [12.1.1.a], liniile [7] și [8] ajustează pe  $D$  luând în considerare posibilitatea de a exista în acel moment un drum special mai scurt spre  $y$  trecând prin nodul  $x$ .
  - Dacă există, acest drum trece prin vechiul  $M$ , ajunge la  $x$  și de aici imediat la  $y$ , și costul său,  $D[x] + COST[x, y]$ , va fi comparat cu  $D[y]$  în linia [8].
  - Dacă noul drum special este mai scurt,  $D[y]$  va fi redus în consecință.
  - Această verificare se realizează pentru fiecare nod  $y$  încă neselectat, adică aparținând mulțimii  $N - M$
- Singura **altă** posibilitate pentru un **drum mai scurt** apare în figura 12.1.2.a. (b), în care un astfel de drum înaintează spre  $x$ , revine în vechiul  $M$  la un membru oarecare  $z$  al acestuia, iar apoi atinge pe  $y$ .



**Fig.12.1.2.a.** Drum minim ipotetic și drum special minim imposibil (reluare)

- În realitate un astfel de drum **nu poate exista**.
- Deoarece  $z$  a fost plasat în  $M$  înaintea lui  $x$ , cel mai scurt dintre toate drumurile care conectează originea cu  $z$  trece obligatoriu prin vechiul  $M$ .
- Deci drumul la  $z$  trecând prin  $x$ , prezentat în figura 12.1.2.a. (b), **nu** este mai scurt decât drumul direct la  $z$  prin  $M$ .
- În consecință, lungimea drumului din figura 12.1.2.a. (b), de la origine la  $y$ , trecând prin  $x$  și  $z$  **nu este mai mică** decât vechea valoare  $D[y]$ , de vreme ce  $D[y]$  **nu a fost mai mare** decât lungimea celui mai scurt drum la  $z$  prin  $M$  și apoi direct la  $y$  la momentul selecției nodului  $z$ .

- Astfel D[y] **nu** va fi modificat în linia [8] a secvenței [12.1.1.a] de un drum trecând prin x și z (fig.12.1.2.a (b)), motiv pentru care lungimea unui astfel de drum **nu** este luată în considerare în cadrul algoritmului.

### 12.1.3. Analiza performanței algoritmului lui Dijkstra

- Se presupune că algoritmul din secvența [12.1.1.a] reluată mai jos, operează asupra unui graf orientat cu n noduri și a arce.

---

**procedura** Dijkstra;

```
/*Determină costurile celor mai scurte drumuri care
conectează nodul 1, considerat drept origine, cu toate
celelalte noduri ale unui graf orientat. Distanțele se
pastrează în tabloul D*/
```

```
[1] M=[1]; /*nodul origine*/
    /*initializarea tabloului de distanțe D*/
[2] pentru (i=2 la n)
[3]   D[i]=COST[1,i];                      /*[12.1.1.a]*/
    /*determinarea drumurilor minime*/
[4] pentru (i=1 la n-1)
[5]   | *alege un nod x aparținând mulțimii N-M astfel
   |   încât D[x] să fie minim;
[6]   | *adaugă pe x lui M;
   |   /*actualizare distanțe*/
[7]   | pentru (fiecare nod y din mulțimea N-M)
[8]     |   D[y]= min(D[y], D[x]+COST[x,y])
     |   □ /*pentru*/
/*Dijkstra*/
```

---

- (1) Dacă pentru reprezentarea grafului se utilizează o **matrice de adiacențe**, atunci bucla **pentru** din liniile [7] - [8] necesită un efort de calcul proporțional cu  $O(n)$ .
  - Întrucât această buclă este executată de  $n-1$  ori, (bucla **pentru** din linia [4]), timpul total de execuție va fi proporțional cu  $O(n^2)$ .
  - Este ușor de observat că restul algoritmului **nu** necesită tempi superiori acestei valori.
- (2) Dacă  $a$  este mult mai mic decât  $n^2$ , este mai indicat ca în reprezentarea grafului să se utilizeze **liste de adiacențe**, respectiv o **coadă bazată pe prioritate** implementată ca un ansamblu (arbore binar parțial ordonat), pentru a păstra distanțele la nodurile mulțimii N-M.
  - În aceste condiții, bucla **pentru** din liniile [7] - [8] poate fi implementată ca și o traversare a **listei de adiacențe** a lui x cu actualizarea distanțelor nodurilor din **coada bazată pe prioritate**.
  - În total, pe întreaga durată a execuției procedurii, se vor realiza  $a$  actualizări, fiecare cu un efort proporțional cu  $O(\log_2 n)$ , astfel încât timpul total consumat în liniile [7] - [8] va fi proporțional cu  $O(a \log_2 n)$  și nu cu  $O(n^2)$ .
  - În mod evident liniile [2] - [3] necesită asemenea liniilor [4] - [6] un efort de calcul proporțional cu  $O(n)$ .

- Utilizând pentru reprezentarea mulțimii N-M o **coadă bazată pe priorități**, linia [5] implementează operația de extragere a elementului cu prioritatea minimă din coadă, fiecare din cele  $n-1$  iterații ale acestei linii necesitând  $O(\log_2 n)$  unități de timp, în total  $O(n \log_2 n)$ .
- În consecință, **timpul total** consumat de această variantă a algoritmului lui Dijkstra este limitat la  $O(a \log_2 n)$  ( $a \geq n-1$ ), performanță care este considerabil mai bună decât  $O(n^2)$ .
- Se reamintește faptul că această performanță se poate obține numai dacă  $a$  este mult mai mic în raport cu  $n^2$ .

## 12.2 Problema drumurilor minime corespunzătoare tuturor perechilor de noduri ("All-Pairs Shortest Path Problem")

- Se presupune că există un **graf orientat ponderat** conținând timpii de zbor pentru anumite trasee aeriene care conectează orașe și că se dorește construcția unei tabele care furnizează **cei mai scurți timpi de zbor** între **oricare două orașe**.
  - Aceasta este o instanță a **problemei drumurilor minime corespunzătoare tuturor perechilor de noduri** ("all-pairs shortest paths problem").
- Pentru a formula problema în **termeni formali**, se presupune că se dă un **graf orientat ponderat**  $G = (N, A)$ , în care fiecare arc  $(x, y)$  are o pondere pozitivă  $COST[x, y]$ .
  - Rezolvarea **problemei drumurilor minime corespunzătoare tuturor perechilor de noduri** pentru graful  $G$ , presupune determinarea pentru fiecare pereche ordonată de noduri  $(x, y)$ , unde  $x, y \in N$ , a **lungimii drumului minim** care conectează nodul  $x$  cu nodul  $y$ .
- Această problemă poate fi rezolvată cu ajutorul **algoritmului lui Dijkstra**, considerând pe rând, fiecare nod al grafului  $G$  drept origine.
  - De fapt rezultatul execuției algoritmului lui Dijkstra este un **tablou** care conține distanțele de la origine la restul nodurilor grafului.
  - În cazul de față, deoarece este necesară determinarea distanțelor pentru toate perechile de noduri, avem nevoie de o **matrice de elemente** în care fiecare **rând** se poate determina în baza algoritmului lui Dijkstra.
- O rezolvare mai directă a **problemei drumurilor minime corespunzătoare tuturor perechilor de noduri ale unui graf**, este datorată **algoritmului lui R.W. Floyd** datând din anul 1962.

### 12.2.1. Algoritmul lui Floyd

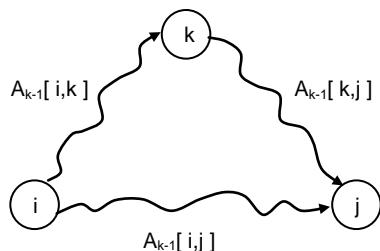
- Fie **graful orientat ponderat**  $G = (N, A)$  care are atașată matricea de ponderi  $COST$ .
  - Pentru conveniență, se va considera că nodurile mulțimii  $N$  sunt numerotate de la 1 la  $n$ .
- **Algoritmul lui Floyd** utilizează o matrice  $A$  de dimensiuni  $n \times n$  cu ajutorul căreia se determină **lungimile drumurilor minime** între toate perechile de noduri.
  - Inițial se face  $A[i, j] = COST[i, j]$  pentru toți  $i \neq j$ .

- Dacă nu există niciun arc de la nodul  $i$  la nodul  $j$ , se presupune că  $\text{COST}[i, j] = \infty$ .
  - Elementele diagonalei principale a matricei  $A$  se pun toate pe 0.
  - **Algoritmul lui Floyd** execută în iterații asupra matricii  $A$ .
    - După cea de-a  $k$ -a iterație,  $A[i, j]$  va conține lungimea minimă a unui drum de la nodul  $i$  la nodul  $j$ , care **nu** trece prin nici un nod cu număr mai mare ca și  $k$ .
    - Cu alte cuvinte,  $i$  și  $j$  pot fi oricare două noduri ale grafului care delimităază începutul și sfârșitul drumului, dar orice nod intermediar care intră în alcătuirea drumului trebuie să aibă numărul mai mic sau cel mult egal cu  $k$ .
    - În cea de-a  $k$ -a iterație, se examinează nodul  $k$  și se utilizează următoarea **formulă** în calculul lui  $A$  (formula [12.2.1.a]):
- 

$$A_k[i, j] = \min \begin{cases} A_{k-1}[i, j] \\ A_{k-1}[i, k] + A_{k-1}[k, j] \end{cases} \quad [12.2.1.a]$$


---

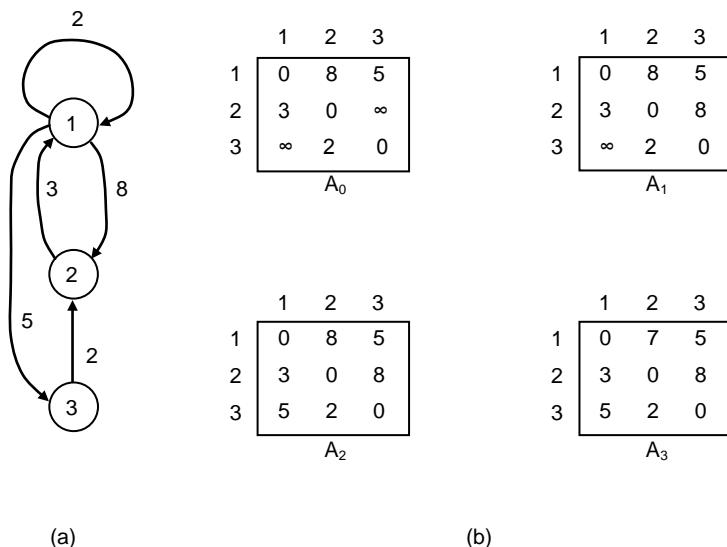
- Se face precizarea că indicele  $k$  semnifică **momentul de timp** la care se realizează calculul matricei  $A$  (în cadrul celei de-a  $k$ -a iterații) și **nu** faptul că ar exista  $n$  matrici  $A$ , fiind utilizat pentru o înțelegere mai facilă a funcționării algoritmului.
- Formula de mai sus are interpretarea simplă precizată în figura 12.2.1.a.



**Fig.12.2.1.a.** Selecția drumului minim de la nodul  $i$  la nodul  $j$  la momentul  $k$

- Pentru calculul lui  $A_k[i, j]$  la momentul curent, se compară  $A_{k-1}[i, j]$  - adică costul drumului de la  $i$  la  $j$  fără a trece prin nodul  $k$  sau oricare alt nod cu număr mai mare ca și  $k$ , la momentul anterior, cu  $A_{k-1}[i, k] + A_{k-1}[k, j]$  - adică costul drumului de la  $i$  la  $k$  însumat cu costul drumului de la  $k$  la  $j$  fără a trece prin nici un nod cu număr mai mare ca și  $k$ , tot la momentul anterior.
- Dacă drumul din urmă se dovedește a fi mai scurt, atunci el este atribuit valorii  $A_k[i, j]$ , altfel aceasta rămâne identică cu valoarea anterioară.

- Pentru graful ponderat din figura 12.2.1.b. (a), se prezintă în cadrul aceleiași figurii modul în care se modifică matricea  $A$  pornind de la conținutul ei inițial  $A_0$  și terminând cu conținutul său final  $A_3$ .



**Fig.12.2.1.b.** Calculul lungimii drumurilor minime corespunzătoare tuturor perechilor de noduri ale unui graf utilizând algoritmul lui Floyd

- Deoarece  $A_k[i, k] = A_{k-1}[i, k]$  și  $A_k[k, j] = A_{k-1}[k, j]$ , nici o intrare a matricii A în care intervine pe post de indice valoarea k, **nu** se modifică în timpul celei de-a k-a iterării.
    - Cu alte cuvinte, matricea A este **invariantă** în raport cu indicele k.
  - În consecință se poate utiliza **o singură copie a matricii A**.
  - Structura de principiu a procedurii care implementează **algoritmul lui Floyd** în baza considerentelor mai sus enunțate apare în secvența [12.2.1.a].

```

PROCEDURE Floyd(float A[n,n], float COST[n,n])

/*Procedura determină matricea drumurilor minime A pornind
de la matricea ponderilor COST*/

    int i,j,k;
    /*matricea A se initializează cu matricea COST*/
[1] pentru (i=1 la n)
[2]     pentru (j=1 la n)
[3]         A[i,j]=COST[i,j];                      /*[12.2.1.a]*/
        /*diagonala principală a lui A pe zero*/
[4]     pentru (i=1 la n)
[5]         A[i,i]=0;
        /*determinarea matricei lungimilor drumurilor minime A*/
[6]     pentru (k=1 la n)
[7]         pentru (i=1 la n)
[8]             pentru (j=1 la n)
[9]                 daca (A[i,k]+A[k,j]<A[i,j])
[10]                A[i,j]=A[i,k]+A[k,j];

```

```
/*Floyd*/
```

---

- Timpul de execuție al acestei proceduri este în mod clar proporțional cu  $O(n^3)$  deoarece la baza sa stă o buclă triplă încubată.
- Verificarea corectitudinii funcționării algoritmului se poate realiza simplu prin metoda inducției.
  - Astfel este ușor de demonstrat că după ce  $k$  trece prin bucla triplă **pentru**,  $A[i, j]$  memorează lungimea celui mai scurt drum de la nodul  $i$  la nodul  $j$  care în niciun caz nu trece prin un nod cu număr mai mare ca și  $k$  [AH85].

### 12.2.2. Comparație între algoritmul lui Floyd și algoritmul lui Dijkstra

- Pentru grafuri reprezentate prin **matrici de adiacențe**:
  - (1) Versiunea **algoritmului Dijkstra** determină drumurile minime specifice unui nod precizat cu performanță  $O(n^2)$ .
  - (2) **Algoritmul lui Floyd** determină toate drumurile minime cu performanță  $O(n^3)$ .
    - Constantele de proporționalitate reale depind de natura compilatorului, de sistemul de calcul și de implementarea propriu-zisă.
    - De fapt activitatea experimentală și măsurările reale reprezintă cele mai bune criterii de apreciere a performanțelor unui algoritm.
- În **condițiile** în care numărul de arce a este **mult mai redus** decât  $n^2$  și grafurile sunt reprezentate prin **structuri de adiacențe**:
  - (1) **Algoritmul lui Floyd** își păstrează performanță stabilită  $O(n^3)$ .
  - (2) Este de așteptat ca **algoritmul lui Dijkstra** să se comporte mai bine.
    - Astfel după cum s-a precizat, acest algoritm determină drumurile minime corespunzătoare unui nod precizat (origine) cu un efort de calcul proporțional cu  $O(a \log_2 n)$ .
    - În consecință utilizând această variantă de algoritm, problema drumurilor minime corespunzătoare tuturor perechilor de noduri se poate rezolva aplicând algoritmul lui Dijkstra tuturor nodurilor grafului cu performanță  $O(na \log_2 n)$ .
    - Se face din nou precizarea că această performanță se poate obține atunci când  $a << n^2$  respectiv în cazul **grafurilor rare de mari dimensiuni**.

### 12.2.3. Determinarea traseelor drumurilor minime

- Algoritmii dezvoltați în cadrul acestui paragraf determină de fapt **costurile drumurilor minime**.
    - De multe ori, în practică este însă foarte util a se cunoaște și **traseul** acestor drumuri minime.
  - Pentru a rezolva această problemă, în contextul **algoritmului lui Floyd** este necesară utilizarea unei alte matrici numite Drum.
    - În matricea Drum, o locație  $\text{Drum}[i, j]$  memorează indicele nodului  $k$  care conduce în cadrul algoritmului la cea mai mică valoare pentru  $A[i, j]$ .
    - Dacă  $\text{Drum}[i, j] = 0$  atunci cel mai scurt drum este cel direct de la nodul  $i$  la nodul  $j$ .
  - Varianta modificată a **algoritmului lui Floyd** care determină și matricea Drum apare în secvența [12.2.3.a].
- 

```

/*definirea matricii care memorează traseele drumurilor
minime*/
float Drum[n,n];

PROCEDURE Floyd(float A[n,n], float COST[n,n])

/*Procedura determină matricea A care memorează lungimile
drumurilor minime pornind de la matricea ponderilor COST. În
plus determină traseul drumurilor minime în matricea Drum*/

    int i,j,k;
    /*initializarea matricilor A și Drum*/
[1]  pentru (i=1 la n)
[2]    pentru (j=1 la n)
[3]      A[i,j]=COST[i,j];
[4]      | Drum[i,j]=0;
[5]      |   □                                     /*[12.2.3.a]*/
[6]      |   /*diagonala principală a lui A pe zero*/
[7]  pentru (i=1 la n)
[8]    A[i,i]= 0;
[9]    /*determinarea lungimii drumurilor minime în matricea A
       și a traseelor acestor drumuri în matricea Drum*/
[10]   pentru (k=1 la n)
[11]     pentru (i=1 la n)
[12]       dacă (A[i,k]+A[k,j]<A[i,j])
[13]         | A[i,j]=A[i,k]+A[k,j];
[14]         | Drum[i,j]=k;
[15]         |   □
/*Floyd*/

```

---

- Matricea Drum este de fapt o **colecție de arbori corespunzători drumurilor minime**, câte unul pentru fiecare nod al grafului original, considerat drept origine.
- Afisarea **traseului drumului minim** de la nodul  $i$  la nodul  $j$ , prin evidențierea nodurilor intermediare se poate realiza cu ajutorul procedurii recursive **Traseu** (secvența [12.2.3.b]).

---

```
PROCEDURE Traseu(int i,j)
```

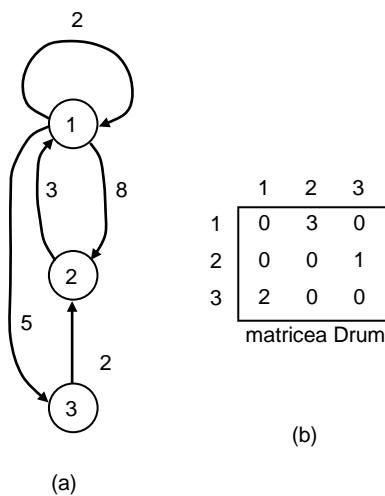
```
/*Afiseaza traseul drumului minim de la nodul i la nodul j*/

int k; /*[12.2.3.b]*/
k=Drum[i,j];
daca (k!=0)
  Traseu(i,k);
  *scrive(k);
  Traseu(k,j);
  □
/*Traseu*/


---


```

- Procedura **Traseu** aplicată unei matrici oarecare ar putea cicla la infinit.
  - Acest lucru **nu** se întâmplă în situația în care ea este aplicată matricii Drum, deoarece este imposibil ca un nod oarecare  $k$  să apară în drumul minim de la nodul  $i$  la nodul  $j$  și în același timp  $j$  să apară în drumul minim de la  $i$  la  $k$ .
  - Se reamintește din nou importanța crucială a valorilor **pozitive** a ponderilor arcelor grafului.
- În figura 12.2.3.a apare conținutul final al matricii Drum (b) pentru graful orientat din aceeași figura (a).



**Fig.12.2.3.a.** Matricea traseelor drumurilor minime (b) corespunzătoare grafului (a).

## 12.2.4. Aplicație. Determinarea centrului unui graf orientat ponderat

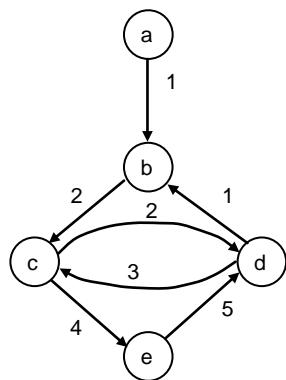
- Se presupune că se dă graful orientat ponderat  $G = (N, A)$  și se cere să se determine **cel mai central nod** al său adică **centrul grafului**.
- Pentru a defini termenul de “**cel mai central nod**” se definește pentru început noțiunea de **excentricitate a unui nod** aparținând unui graf ponderat.
  - **Excentricitatea** unui nod  $x \in N$  este precizată de formula [12.2.4.a]:

---


$$\text{Excentricitate}[x] = \max_{y \in N} \{\text{drumurile minime de la } x \text{ la } y\} \quad [12.2.4.a]$$


---

- Cu alte cuvinte, **excentricitatea** unui nod al unui graf ponderat este **valoarea maximă** dintre lungimile drumurilor minime de la nodul respectiv la toate celelalte noduri ale grafului.
- **Centrul unui graf**  $G$  este nodul a cărui excentricitate este **minimă**.
  - Spre exemplu pentru graful din figura 12.2.4.a. (a), valorile **excentricităților** nodurilor apar în tabloul (c) din aceeași figură.
  - Conform definiției anterioare, centrul grafului  $G$  este nodul  $d$ .
- Utilizând **algoritmul lui Floyd**, determinarea **centrului unui graf**  $G$  se poate realiza simplu.
- Presupunând că **ponderilor arcelor grafului** sunt memorate în matricea COST, se procedează astfel:
  - (1) Se determină cu ajutorul **procedurii Floyd** matricea drumurilor minime corespunzătoare tuturor perechilor de noduri (fig.12.2.4.a. (b)).
  - (2) Se determină **excentricitățile nodurilor**  $i$ ,  $(1 \leq i \leq N)$  găsind valoarea maximă de pe fiecare coloană  $i$  a matricei.
  - (3) Se caută nodul cu **excentricitatea minimă**. Acesta este centrul grafului  $G$ .
- În figura 12.2.4.a. (b) apare matricea valorilor drumurilor minime pentru graful din aceeași figură (a).
  - Determinarea nodului central al grafului este evidentă.



a	b	c	<b>d</b>	e	
a	0	1	3	5	7
b	$\infty$	0	2	4	6
c	$\infty$	3	0	2	4
d	$\infty$	1	3	0	7
e	$\infty$	6	8	5	0
max		6	8	<b>5</b>	7

(b)

nod	excentricitate
a	$\infty$
b	6
c	8
<b>d</b>	<b>5</b>
e	7

(c)

**Fig.12.2.4.a.** Determinarea centrului unui graf ponderat

### 12.3. Închiderea tranzitivă

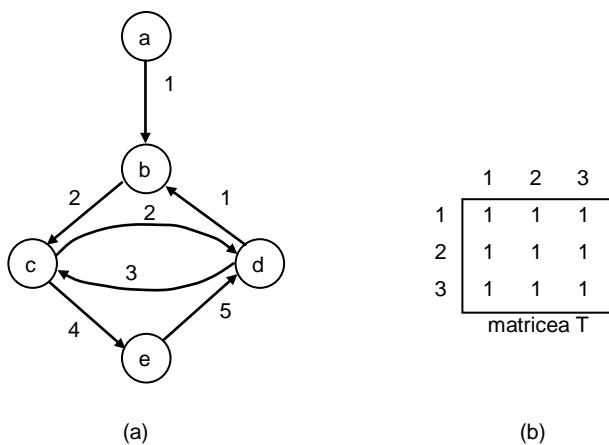
- În grafurile neorientate, nodurile la care se poate ajunge pornind de la un nod precizat, cu alte cuvinte conexiunile unui nod la grafului, pot fi determinate simplu aplicând **proprietățile de conectivitate** ale grafurilor.
  - Pur și simplu, toate nodurile la care se poate ajunge în procesul de căutare aparțin unei aceleiași **componente conexe a grafului**.
- Această observație este valabilă în principiu și în cazul **grafurilor orientate** cu precizarea însă că în această situație, rezolvarea este mai complexă și ea **nu** poate fi redusă la simpla determinare a componentelor conexe.
- O modalitate de a rezolva problemele de conectivitate în cazul **grafurilor orientate** este următoarea:
  - Se **completează** graful inițial cu arce pe baza următoarei metode: dacă în graful inițial se poate ajunge într-un mod oarecare de la nodul  $x$  la nodul  $y$  parcurgând arcele orientate ale grafului, atunci se adaugă grafului arcul  $(x, y)$ .
- Graful care se obține adăugând **toate arcele** de această natură se numește **închiderea tranzitivă** a grafului inițial.
  - Deoarece este de așteptat să fie adăugate un număr mare de arce, deci graful obținut să fie dens, se apreciază că pentru **închiderea tranzitivă**, cea mai potrivită metodă de reprezentare este cea bazată pe **matrice de adiacențe**.
- Odată determinată închiderea tranzitivă a unui graf orientat, răspunsul la întrebarea: “Există un drum în graful orientat de la nodul  $x$  la nodul  $y$ ? ” este imediat.
- **Algoritmul lui Floyd** poate fi specializat astfel încât să determine **închiderea tranzitivă** a unui graf  $G$ .
  - Algoritmul care rezultă se numește **algoritmul lui Warshall** și care deși a apărut tot în anul 1962 este anterior ca dată algoritmului lui Floyd.

#### 12.3.1. Algoritmul lui Warshall

- **Algoritmul lui Warshall** se bazează pe observația simplă că, dacă într-un graf orientat există o modalitate de a ajunge de la nodul  $i$  la nodul  $k$  și o modalitate de a ajunge de la nodul  $k$  la nodul  $j$ , parcurgând arce ale grafului, atunci cu siguranță există un drum care conectează nodul  $i$  cu nodul  $j$ .
  - Problema constă de fapt în a exploata de asemenea manieră această observație încât calculul să se realizeze la **o singură trecere prin matrice**.
- Acest lucru este posibil în baza următoarei interpretări sugerate de **Warshall**:
  - “Dacă există o posibilitate de a ajunge de la nodul  $i$  la nodul  $k$  utilizând numai noduri cu indici mai mici decât  $k$ , și o posibilitate de a ajunge de la

nodul  $k$  la nodul  $j$  în aceleași condiții, atunci există un drum de la nodul  $i$  la nodul  $j$  care străbate numai noduri care cu indicele mai mic ca și  $k+1$ ".

- În baza acestei interpretări, dându-se graful ordonat  $G$  și matricea sa de adiacențe  $A$ , **algoritmul lui Warshall** determină matricea  $T$  care reprezintă **închiderea tranzitivă** a grafului  $G$ .
  - În această matrice  $T[i, j] = \text{true}$ , dacă există posibilitatea de a ajunge de la nodul  $i$  la nodul  $j$ , altfel  $T[i, j] = \text{false}$ .
- Spre exemplu în figura 12.3.a. (b) apare închiderea tranzitivă în forma matricii  $T$  a grafului orientat din figura 12.3.a. (a).



(a)

(b)

**Fig.12.3.a.** Închiderea tranzitivă a unui graf orientat

- Închiderea tranzitivă** poate fi determinată aplicând o procedură similară procedurii **Floyd**, utilizând însă următoarea formulă de calcul în cea de-a  $k$ -a trecere prin matricea  $T$  (formula [12.3.a]):

---


$$T_k[i, j] = T_{k-1}[i, j] \text{ OR } (T_{k-1}[i, k] \text{ AND } T_{k-1}[k, j]) \quad [12.3.a]$$


---

- Această formulă precizează că există un drum de la nodul  $i$  la nodul  $j$  care nu trece printr-un nod cu număr mai mare ca și  $k$  dacă:
  - (1) Există deja un drum de la  $i$  la  $j$  care nu trece prin nici un nod cu număr mai mare decât  $k-1$ , sau
  - (2) Există un drum de la  $i$  la  $k$  care nu trece prin nici un nod cu număr mai mare decât  $k-1$  și există un drum de la  $k$  la  $j$ , care nu trece prin niciun nod cu număr mai mare decât  $k-1$ .
- Ca și în cazul algoritmului lui Floyd, sunt valabile formulele  $T_k[i, k] = T_{k-1}[i, k]$  și  $T_k[k, j] = T_{k-1}[k, j]$ , adică matricea  $T$  este **invariantă** în raport cu  $k$ , motiv pentru care în calcul se poate utiliza o singură instanță a matricii  $T$ .
- Structura de principiu a procedurii care implementează **algoritmul lui Warshall** apare în secvența [12.3.a].

---

**/\*Algoritmul lui Warshall - varianta C\*/**

```

void Warshall(Boolean A[n][n], Boolean T[n][n])

/*Procedura construiește în T închiderea tranzitivă a lui
A*/
{
    int i,j,k;

    /*initializarea matricei T*/
[1] for (i=0;i<n;i++)
[2]     for (j=0;j<n;j++)
[3]         T[i,j]=A[i,j];                                /*[12.3.a]*/

    /*determinarea matricei T*/
[4] for (k=0;k<n;k++)
[5]     for (i=0;i<n;i++)
[6]         for (j=0;j<n;j++)
[7]             if (T[i,j]==false)
[8]                 T[i,j]= T[i,k] && T[k,j];
}
/*Warshall*/
-----
```

### 12.3.2. Analiza performanței algoritmului lui Warshall

- În urma unei analize sumare a codului, performanța algoritmului rezultă imediat ca fiind egală cu  $O(n^3)$ .
- După alți autori, performanța poate fi stabilită și astfel:
  - Fiecare din cele  $n$  arce conduce la o iterare cu  $n$  pași în bucla **for** cea mai interioară.
  - În plus, sunt testate și eventual actualizate toate cele  $n^2$  locații ale matricii T.
  - Rezultă un timp de execuție proporțional cu  $O(an + n^2)$  [Se 88].

## 12.4. Traversarea grafurilor orientate

- Tehnicile fundamentate de traversare a grafurilor dezvoltate în cadrul capitolului 10, au un **caracter universal** și ele pot fi aplicate, cu particularizări specifice, oricărui tip de graf.
  - Astfel în cazul **grafurilor orientate**, în timpul traversării se ține cont efectiv de **orientarea arcelor** examineate, lucru care în contextul grafurilor neorientate **nu** este necesar.
  - Din acest motiv și arborii de acoperire rezultați în urma procesului de traversare a grafurilor orientate au o structură mai complicată.
- În cele ce urmează vor fi abordate unele aspecte legate de **traversarea grafurilor orientate** prin tehnica “**căutării în adâncime**”.
  - Opțiunea este motivată de faptul că această manieră de traversare a grafurilor orientate stă la baza rezolvării a numeroase probleme practice care se pun în legătură cu această categorie de grafuri.

## 12.4.1. Traversarea grafurilor orientate prin tehnica căutării "în adâncime"

- Principiul traversării în adâncime este deja cunoscut.
  - Se presupune că există un graf orientat  $G$  în care toate nodurile sunt marcate inițial cu "nevizitat".
  - Căutarea în adâncime acționează inițial selectând un nod  $x$  al lui  $G$  ca și nod de start, nod care este marcat cu "vizitat".
  - În continuare, fiecare nod nevizitat adiacent lui  $x$  este "căutat" pe rând, utilizând aceeași tehnică în manieră recursivă.
  - În momentul în care toate nodurile la care se poate ajunge pornind de la  $x$  au fost vizitate, traversarea este încheiată.
  - Dacă totuși mai rămân noduri nevizitate, se selectează unul dintre acestea drept următorul nod de start și se relansează căutarea recursivă.
  - Acest proces se repetă până când sunt parcurse toate nodurile grafului  $G$ .
- Pentru implementare se consideră că graful  $G$  care se dorește a fi traversat este reprezentat cu ajutorul **listelor de adiacențe**.
  - Se notează cu  $L(x)$  lista de adiacențe a nodului  $x$ .
  - De asemenea se mai utilizează tabloul  $marc$ , ale cărui elemente pot lua valorile "nevizitat" respectiv "vizitat", tablou care este utilizat în menținerea evidenței stării nodurilor grafului.
- Structura de principiu a procedurii recursive de căutare în adâncime apare în secvența [12.4.1.a].
  - Această procedură trebuie apelată însă dintr-un context mai general care asigură inițializarea tabloului  $marc$  și selectarea nodurilor încă nevizitate ale grafului (secvența [12.4.1.b]).

---

**/\*Căutare în adâncime - varianta pseudocod\*/**

```
procedura CautInAdâncime(tip_nod x)

/*varianta pseudocod CautInAdacime*/

tip_nod k;

[1] marc[x]= vizitat;
[2] Prelucrare(x);
[3] pentru (fiecare nod k din L(x))           /*[12.4.1.a]*/
[4]   daca (marc[k] este nevizitat)
[5]     CautInAdâncime(k);
/*CautInAdâncime*/



---


/*Programul principal*/

/*initializare tablou mark*/
pentru (x=1 la n)
  marc[x]= nevizitat;
/*determinare componente conexe*/
pentru (x= 1 la n)                                /*[12.4.1.b]*/

```

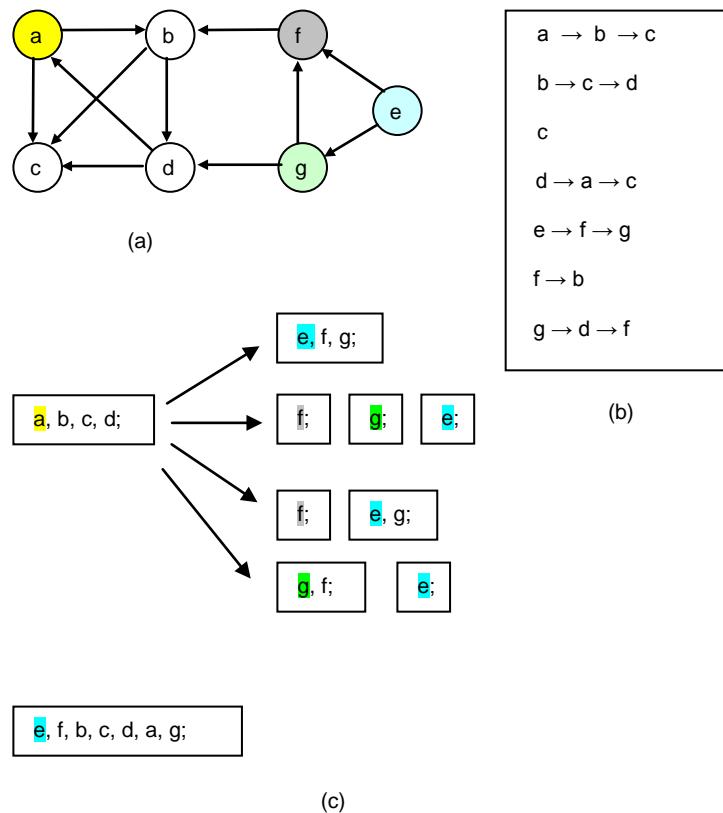
```

daca (marc[x] este nevizitat)
    CautInAdâncime(x);

```

---

- Se face precizarea, că procedura **CautInAdâncime** nu realizează nici o prelucrare efectivă a nodurilor vizitate, aceasta fiind sugerată generic prin apelativul **Prelucrare(x)** în linia [ 2 ] a secvenței [12.4.1.a].
- După cum s-a precizat, această tehnică stă la baza rezolvării mai multor probleme specifice grafurilor orientate.
  - În consecință, prelucrarea nodurilor vizitate va îmbrăca o formă specifică funcție de problema care se dorește a fi soluționată în baza acestei tehnici de parcursere a grafurilor orientate.
- **Performanța procedurii CautInAdâncime**, analizată în contextul parcurgerii unui graf orientat cu arce, cu  $n \leq a$ , în reprezentarea bazată pe liste de adiacențe este  $O(a)$ , după cum s-a evidențiat în capitolul 10.
- Pentru **exemplificarea** procesului de traversare a grafurilor orientate prin metoda căutării în adâncime:
  - Se presupune că procedura din secvența [12.4.1.b] este aplicată grafului orientat din figura 12.4.1.a. (a), considerând că nodul de pornire este nodul a ( $x=a$ ).
  - Graful se consideră reprezentat printr-o **structură de adiacențe** sugerată în aceeași figuru (b).



**Fig.12.4.1.a.** Traversarea unui graf orientat

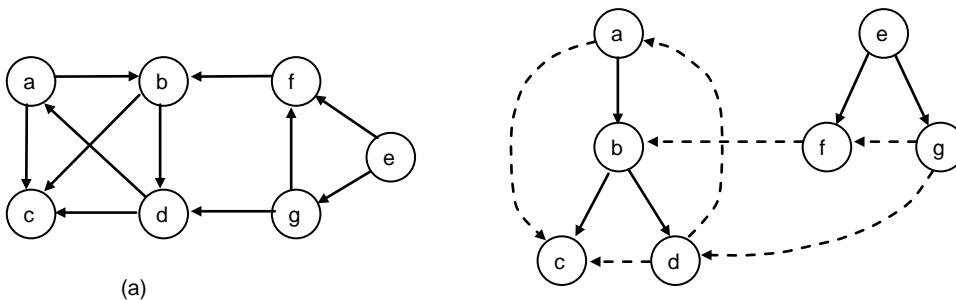
- Algoritmul marchează nodul a cu vizitat și selectează nodul b, primul din lista de adiacențe a lui a.

- Deoarece nodul  $b$  este nevizitat, căutarea continuă prin realizarea unei apel **CautInAdâncime**( $b$ ).
- În continuare se marchează  $b$  cu vizitat și se selectează primul nod din lista sa de adiacențe.
- Presupunând că nodul  $c$  apare înaintea lui  $d$  în această listă, aşa cum se prezintă în figură, se apelează **CautInAdâncime**( $c$ ).
- Deoarece lista de adiacențe a lui  $c$  este vidă, căutarea revine în lista lui  $b$  de unde se selectează nodul  $d$ .
- Se parurge în continuare lista de adiacențe a lui  $d$ .
- Întrucât nodurile  $a$  și  $c$  care apar în lista de adiacențe a lui  $d$  au fost deja vizitate, căutarea lui  $d$  se încheie și se revine în lista lui  $b$  apoi în cea a lui  $a$ , care sunt și ele epuizate.
- În acest moment apelul inițial **CautInAdâncime**( $a$ ) s-a terminat.
- Totuși, deoarece graful nu a fost încă parcurs în întregime întrucât nodurile  $e, f$  și  $g$  sunt încă nevizitate, se realizează un nou apel al procedurii **CautInAdâncime** spre exemplu pentru nodul  $e$ , apel care asigură parcurgerea integrală a grafului.
- Se face următoarea **observație**:
  - Dacă graful din fig.12.4.1.a. (a) ar fi un **graf neorientat**, el ar reprezenta o **singură componentă conexă** și ar putea fi parcurs integral, realizând un singur apel al procedurii **CautInAdâncime** din cadrul secvenței [12.4.1.b] **indiferent** de nodul care este selectat drept **punct de start**.
- În cazul **grafurilor orientate** situația se schimbă.
  - Numărul de apeluri nerecursive ale procedurii **CautInAdâncime**, realizat în cadrul secvenței [12.4.1.b] depinde în mod esențial de ordinea în care sunt selectate nodurile care reprezintă puncte de start.
    - Spre exemplu în cazul mai sus prezentat, deoarece s-a ales inițial nodul  $a$  și apoi nodul  $e$  drept puncte de pornire, graful apare constituit din două componente conexe ( $a, b, c, d$ ) și ( $e, f, g$ ).
    - Dacă în locul nodului  $e$ , la cel de-al doilea apel se selectează nodul  $g$  apoi  $e$ , rezultă trei componente conexe ( $a, b, c, d$ ), ( $g, f$ ) și ( $e$ ).
    - Dacă în locul nodului  $e$ , la cel de-al doilea apel se selectează nodurile  $f, g$  și  $e$  în această ordine rezultă 4 componente conexe.
    - Dacă în locul nodului  $e$ , la cel de-al doilea apel se selectează nodurile  $f, e$  și  $g$  în această ordine rezultă 3 componente conexe.
    - Dacă la primul apel, se alege nodul  $e$  drept nod de start, graful din figură conține o singură componentă conexă (fig.12.4.1.a. (c)).
  - Se putea însă alege inițial oricare alt nod drept punct de pornire a parcurgerii, iar ulterior oricare dintre nodurile rămase, fiecare situație conducând la o structură diferită de componente conexe ale grafului.
- Se poate concluziona că **numărul de componente conexe ale unui graf orientat** depinde în mod esențial de **ordinea în care sunt selectate nodurile** care reprezintă puncte de start ale componentelor.

- Această particularitate se reflectă direct în topologia **arborilor de acoperire** care rezultă în urma unor astfel de parcurgeri ale grafurilor orientate.

#### 12.4.2. Păduri de arbori de căutare în adâncime pentru grafuri orientate

- În procesul de traversare al unui graf orientat, anumite arce conduc la noduri nevizitate.
  - Arcele care conduc la noduri nevizitate se numesc “**arce de arbore**” și ele formează un **arbore** respectiv o **pădure de arbori de căutare în adâncime** pentru graful orientat dat.
- În figura 12.4.2.a apare o astfel de **pădure de arbori de căutare în adâncime** corespunzătoare grafului orientat din fig. 12.4.1.a. (a).



**Fig.12.4.2.a.** Pădure de arbori de căutare în adâncime pentru graful din fig. 12.4.1.a. (a)

- După cum se observă în această figură, în afara “**arcelor de arbore**” trasate cu linie continuă, mai apar și alte categorii specifice de arce rezultate din parcurgerea grafurilor orientate prin tehnica căutării în adâncime.
  - Este vorba despre:
    - (1) Arce de tip “**înapoi**” (“**back**”).
    - (2) Arce de tip “**înainte**” (“**forward**”).
    - (3) Arce numite de “**trecere**” (“**cross**”).
    - Toate aceste tipuri de arce apar trasate cu **linie îintreruptă** în cadrul figurii.
- Se reamintește faptul că în contextul **grafurilor neorientate** pentru **arborii de căutare în adâncime** se definesc numai două tipuri de arce: arce “**de arbore**” și arce “**de return**”.
- (1) În contextul **grafurilor orientate**, un arc ca și arcul  $(d, a)$  se numește **arc de tip “înapoi”** deoarece el conectează un nod cu unul din **strămoșii săi** din cadrul arborelui de căutare.
  - Se precizează faptul că un arc care conectează un nod cu el însuși este încadrat tot în această categorie.
- (2) Un arc care conectează un nod cu unul din **descendenții săi proprii** se numește **arc de tip “înainte”**.
  - În figura 12.4.2.a un astfel de arc este  $(a, c)$ .

- (3) Un arc de tip  $(d, c)$  sau  $(g, d)$  care conectează două noduri care nu sunt în relație **descendent** sau **strămoș** se numește **arc “de trecere”**.
  - Se observă că toate arcele de trecere din figura 12.4.2.a sunt orientate de la dreapta la stânga, pe baza presupunerii că:
    - (a) Nodurile fii nou descoperite sunt adăugate în arborii de căutare în adâncime de la **stânga la dreapta**, în ordinea în care sunt vizitate.
    - (b) Noii arbori sunt adăugați pădurii tot de la **stânga la dreapta**.
  - Această presupunere nu este întâmplătoare și ea este legată de observația formulată în paragraful anterior relativ la dependența structurii topologice a pădurii arborilor de căutare de ordinea de vizitare a nodurilor.
- **Problema** care se pune în continuare se referă la modul în care pot fi individualizate cele patru categorii de arce.
  - În primul rând este evident faptul că arcele “**de arbore**” sunt o categorie specială de arce, ele conducând la **noduri nevizitate** din cadrul grafului, motiv pentru care pot fi depistate cu ușurință.
  - Pentru a depista celelalte categorii de arce, care toate conduc la noduri care au fost deja vizitate, este necesar a se introduce o **numerotare a nodurilor** grafului orientat în ordinea în care acestea sunt selectate în procesul de **căutare în adâncime**.
    - Aceste numere pot fi memorate într-un tablou **OrdineInAdâncime** de valori întregi, tablou care conține o locație pentru fiecare nod al grafului.
    - Tabloul poate fi completat pe baza secvenței [12.4.2.a] care se introduce după linia [1] a procedurii **CautInAdâncime** din secvența [12.4.1.a].

---

```

/*completare procedura CautareInAdancime*/
contor= contor+1;
OrdineInAdâncime[x]=contor;           /*[12.4.2.a]*/

```

---

- Se precizează faptul că această manieră de numerotare a mai fost utilizată atât pentru grafuri, cât și pentru alte structuri de date ca de exemplu arborii, ca un element care cuantizează esența procesului de traversare a unei structuri de date.
  - Se reamintește de asemenea faptul că **esența procesului de traversare** constă în **transformarea structurii** supusă traversării într-o **structură listă liniară**.
- În consecință, în timpul traversării unui graf orientat, tuturor descendenților unui nod oarecare  $x$  li se vor atribui **numere** mai mari decât lui  $x$  în tabloul **OrdineInAdâncime**.
- În acest context este valabilă următoarea teoremă:  $y$  este un **descendent** al lui  $x$  dacă și numai dacă este satisfăcută relația [12.4.2.b].
 

---

---

```

OrdineInAdâncime[x] ≤ OrdineInAdâncime[y] ≤ OrdineInAdâncime[x] +
"numărul de descendenți ai lui x"           [12.4.2.b]

```

---

- În aceste condiții:
  - (1) Un **arc de tip “înainte”** conectează un nod cu un număr de ordine mai mic cu un nod având un număr de ordine mai mare.

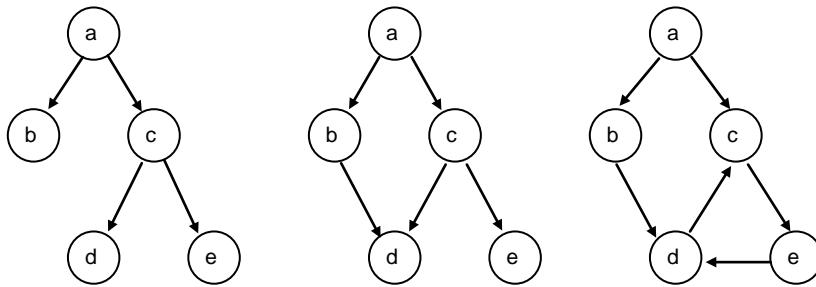
- (2) Un **arc de tip “înapoi”** conectează un nod cu număr mai mare cu un nod având numărul de ordine mai mic.
  - Se face precizarea că în ambele situații, cele două noduri trebuie să fie în **relația strămoș sau descendant**, respectiv pentru ele trebuie să fie satisfăcută **relația** [12.4.2.b].
- (3) **Arcele “de trecere”** conectează noduri cu numere de ordine mai mari cu noduri cu numere mai mici, care **nu** sunt în **relația strămoș sau descendant** deci pentru care relația [12.4.2.b] **nu** este valabilă.
- Pentru a demonstra această afirmație se procedează prin **reducere la absurd**.
  - Se presupune ca  $(x, y)$  este un **arc “de trecere”** și că  $\text{OrdineInAdâncime}[x] \leq \text{OrdineInAdâncime}[y]$ , adică  $x$  și  $y$  sunt în relația strămoș-descendant.
  - Deoarece numărul lui  $x$  este mai mic sau egal cu al lui  $y$ , rezultă că nodul  $x$  este vizitat înaintea lui  $y$ .
  - În consecință, orice nod vizitat în timpul scurs între primul apel al procedurii **CautInAdâncime**( $x$ ) și terminarea acestui apel, este un descendant al nodului  $x$  în procesul de căutare în adâncime.
  - Nodul  $y$  se află într-o astfel de situație.
    - Dacă  $y$  este un nod nevizitat în momentul explorării arcului  $(x, y)$ , atunci arcul  $(x, y)$  este un arc **“de arbore”**.
    - Altfel, nodul  $y$  a fost cu siguranță deja vizitat anterior momentului explorării arcului  $(x, y)$  și în consecință arcul  $(x, y)$  este un arc de tip **“înainte”**.
    - Întrucât concluzia la care s-a ajuns în ambele situații contrazice ipoteza inițială, rezultă că **nu** poate exista un arc de trecere  $(x, y)$  astfel încât  $\text{OrdineInAdâncime}[x] \leq \text{OrdineInAdâncime}[y]$ .
- În următoarele două secțiuni, se prezintă câteva dintre modalitățile de utilizare a traversării grafurilor orientate prin tehnica căutării în adâncime la rezolvarea diverselor probleme legate de acest tip de grafuri.

## 12.5. Grafuri orientate aciclice

- Un **graf orientat aciclic** este un graf orientat care **nu** conține cicluri.
- Apreciate în termenii relațiilor pe care le modeleză, **grafurile orientate aciclice** au un caracter de generalitate mai pronunțat decât **arborii** dar sunt mai puțin generale ca și **grafurile orientate**.
- Ca și **topologie**, astfel de grafuri pot conține multe cicluri, dacă **nu** se ia în considerare direcționarea arcelor.
  - Dacă însă se ia în considerare direcționarea arcelor un astfel de graf **nu** conține nici un ciclu.
- De fapt aceste grafuri pot fi considerate parțial **arbori**, parțial **grafuri orientate**, element care le conferă o serie de proprietăți speciale.
  - Spre exemplu, **pădurea de arbori de căutare în adâncime** asociată unui graf

orientat aciclic, **nu** conține arce de tip “înapoi”.

- În figura 12.5.a. apare un exemplu de arbore (a), un exemplu de graf orientat aciclic (b) și un exemplu de graf orientat cu un ciclu (c).



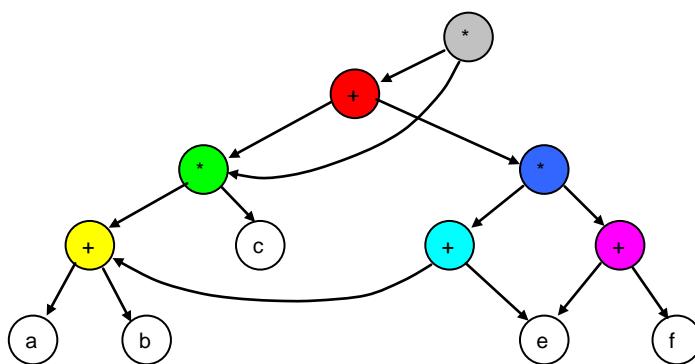
**Fig.12.5.a.** Tipuri de grafuri orientate

- Printre altele, **grafurile orientate aciclice** sunt utile în reprezentarea sintetică a structurii **expresiilor aritmetice** care conțin **subexpresii comune**.

- Spre exemplu figura 12.5.b. evidențiază un astfel de graf pentru expresia aritmetică:

$$((a+b)*c + ((a+b)+e) * (e+f)) * ((a+b)*c)$$

- Termenii  $a+b$  și  $(a+b)*c$  reprezintă **subexpresii comune partajate**, motiv pentru care sunt reprezentate prin noduri care sunt destinația mai multor arce orientate.

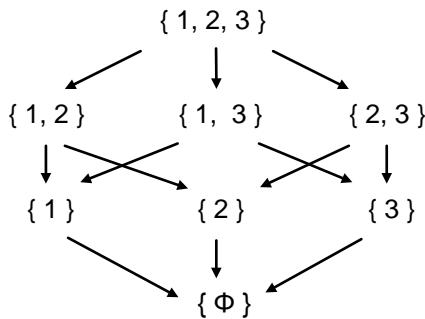


**Fig.12.5.b.** Graf orientat aciclic reprezentând o expresie aritmetică

- Grafurile orientate aciclice** pot fi utilizate cu succes în modelarea **relației de ordonare parțială**.

- Se reamintește faptul că o **relație de ordonare parțială R** pe o mulțime  $M$ , este o **relație binară** care satisface următoarele proprietăți:
  - (1)  $aR\alpha$  este falsă pentru  $\forall \alpha \in M$  (**nereflexivitate**).
  - (2) Pentru  $\forall a, b, c \in M$ , dacă  $aRb$  și  $bRc$  sunt adevărate, atunci  $aRc$  este adevărată (**tranzitivitate**).
  - (3) Dacă  $aRb$  este adevărată și  $bRa$  este adevărată, atunci  $a=b$  pentru  $\forall a, b \in M$  (**antisimetrie**).

- Două exemple naturale de relații de ordonare parțială sunt:
  - (1) Relația “**mai mic decât**” ( $<$ ) definită pe **mulțimea numerelor întregi**.
  - (2) Relația “**submulțime proprie a unei mulțimi**” ( $\subset$ ) definită pentru **mulțimi de elemente**.
- Spre exemplu fie **mulțimea**  $M = \{1, 2, 3\}$  și fie  $P(M)$  **puterea mulțimii**  $M$ , adică mulțimea tuturor submulțimilor proprii ale mulțimii  $M$ .
  - În acest context  $P(M) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$ .
  - Este simplu de verificat că relația “submulțime a mulțimii  $M$ ” ( $\subset$ ) este o **relație de ordonare** parțială pentru puterea mulțimii  $M$ .
- **Grafurile orientate aciclice** pot fi utilizate în **reprezentarea** unor astfel de relații.
- În acest, scop o **relație  $R$**  poate fi asimilată cu o **mulțime de perechi** (arce), care satisfac următoarea proprietate: perechea  $(a, b)$  aparține mulțimii  $R$  dacă  $aRb$  este adevărată.
- În aceste condiții, este valabilă următoarea **definiție**:
  - Dacă  $R$  este o **relație de ordonare parțială** pe o mulțime  $M$ , atunci graful  $G = (M, R)$  este un **graf orientat aciclic**.
- **Reciproc**:
  - Se presupune că  $G = (M, R)$  este un **graf orientat aciclic**.
  - Se presupune de asemenea că  $R^+$  este o **relație** definită prin afirmația  $aR^+b$  este adevărată dacă și numai dacă există un drum de lungime mai mare sau egală cu 1 care conectează nodul  $a$  cu nodul  $b$  în graful  $G$
  - În aceste condiții,  $R^+$  reprezintă o **relație de ordonare parțială** pe  $M$ .
    - $R^+$  este de fapt mulțimea relațiilor care materializează **închiderea tranzitivă** a lui  $R$ .
- În figura 12.5.c apare graful orientat aciclic  $G = (P(M), R)$  unde  $M = \{1, 2, 3\}$ .
  - Relația  $R^+$  se traduce prin **submulțime proprie** a puterii mulțimii  $M$ .



**Fig.12.5.c.** Graf orientat aciclic reprezentând submulțimile proprii ale unei mulțimi date

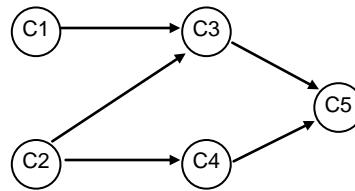
#### 12.5.1. Determinarea aciclității unui graf orientat

- Se consideră un graf orientat  $G = (N, A)$  și se cere să se stabilească dacă  $G$  este aciclic, cu alte cuvinte să se determine dacă  $G$  nu conține cicluri.
- Pentru a rezolva această problemă poate fi utilizată cu succes **tehnica căutării în adâncime**.
  - Astfel, dacă pe parcursul traversării grafului  $G$  prin **tehnica căutării în adâncime** se întâlnește cel puțin un arc de tip “**înapoi**”, în mod evident graful conține cel puțin un **ciclu**.
  - **Reciproc**, dacă un graf orientat conține cel puțin un **ciclu**, atunci în orice traversare a grafului prin tehnica căutării în adâncime va apărea cel puțin un arc de tip “**înapoi**”.
- Pentru a **demonstra** acest lucru, se presupune că  $G$  este **ciclic**.
  - Realizând o traversare prin căutare în adâncime în  $G$ , va exista cu siguranță în ciclu un nod  $x$ , al cărui număr de ordine la căutarea în adâncime este mai mic decât numărul corespunzător oricărui nod aparținând ciclului.
  - Se consideră un arc  $(y, x)$  din ciclul care-l conține pe  $x$ .
  - Deoarece  $y$  este în ciclu, el trebuie să fie un **descendent** al lui  $x$  în pădurea de arbori de căutare prin cuprindere.
  - Deci  $(y, x)$  **nu** poate fi un arc de “trecere”.
  - Deoarece numărul de ordine la căutarea în adâncime a lui  $y$  este mai mare decât numărul lui  $x$ ,  $(y, x)$  **nu** poate fi nici arc de tip “arbore” nici arc de tip “înainte”.
  - În consecință  $(y, x)$  **nu** poate fi decât un arc de tip “înapoi”.

### 12.5.2. Aplicație. Sortarea topologică

- **Sortarea topologică** a mai făcut obiectul unor prezentări pe parcursul acestei lucrări.
- În paragraful de față se prezintă o altă modalitate de soluționare a **sortării topologice** utilizând drept structuri de date suport **grafurile orientate aciclice**.
- Se reamintește faptul că un **proiect de mari dimensiuni**, este adesea divizat într-o colecție de activități (task-uri) mai mici, fiecare având un caracter mai mult sau mai puțin unitar.
  - În vederea finalizării proiectului, unele dintre activități trebuie realizate într-o anumită ordine specificată.
- Spre exemplu, o **programă universitară** poate conține o serie precizată de cursuri dintre care unele presupun în mod obligatoriu absolvirea în prealabil a altora (pre requisite).
  - O astfel de situație poate fi modelată simplu cu ajutorul **grafurilor orientate aciclice**.
  - Spre exemplu dacă cursul  $D$  este un curs care nu poate fi urmat decât după absolvirea cursului  $C$ , în graf apare un arc  $(C, D)$ .
- În figura 12.5.1.a apare un **graf orientat aciclic** care evidențiază intercondiționările de această natură impuse unui număr de 5 cursuri.

- Cursul C3 spre exemplu, presupune absolvirea în prealabil a cursurilor C1 și C2.



**Fig.12.5.1.a.** Graf orientat aciclic modelând intercondiționări

- **Sortarea topologică** realizează o astfel de **ordonare liniară** a activităților proiectului încât niciuna dintre intercondiționările impuse să **nu** fie încălcate.
  - Cu alte cuvinte, **sortarea topologică** este un **proces de ordonare liniară** a nodurilor unui **graf orientat aciclic**, astfel încât dacă există un arc de la nodul  $i$  la nodul  $j$ , atunci nodul  $i$  să apară înaintea nodului  $j$  în ordonarea liniară.
- Spre exemplu lista  $C1, C2, C3, C4, C5$  reprezintă o sortare topologică a grafului din figura 12.5.1.a.
- Sortarea topologică poate fi realizată simplu pornind de la algoritmul traversării unui graf prin **tehnica căutării în adâncime**.
  - Astfel, dacă în procedura **CautInAdâncime** din secvența [12.4.1.a] se adaugă linia [5] care conține o instrucțiune de tipărire a lui  $x$ , se obține procedura **SortTopologic** care afișează nodurile grafului aciclic **sortate topologic în ordine inversă** (secvența [12.5.1.a]).
  - Acest lucru se întâmplă deoarece procedura **SortTopologic** afișează un nod oarecare  $x$  al grafului aciclic după ce a terminat explorarea (afișarea) tuturor descendenților săi.

---

```

PROCEDURE SortTopologic(tip_nod x);

/*Afișează nodurile accesibile pornind de la nodul x, în
ordine topologică inversă/

tip_nod k;
                           /*[12.5.1.a]*/
[1]  marc[x]=vizitat;
[2]  pentru fiecare nod k din L(x)
[3]    daca (marc[k] este nevizitat)
[4]      SortTopologic(k);
[5]      *scrive(x);
/*SortTopologic*/

```

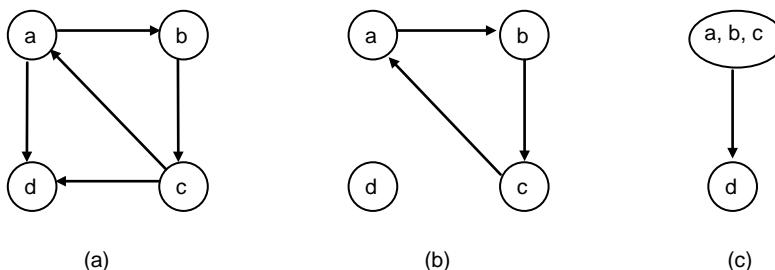
---

- **Precizare:** realizarea sortării topologice a unui graf orientat, este echivalentă cu realizarea unei sortări topologice de tip **SortTopologic** în graful obținut prin **inversarea sensului arcelor grafului** [Se 88].
- Este de asemenea evident faptul că, de regulă, ordonarea produsă de acest tip de sortare **nu** este unică.

- Tehnica utilizată este perfect funcțională deoarece la parcurgerea unui graf orientat aciclic **nu** apar arce de tip “înapoi”.
- Acest lucru se **demonstrează** după cum urmează:
  - Se consideră momentul la care căutarea în adâncime părăsește nodul  $x$ .
  - Toate arcele care apar în pădurea de arbori de căutare în adâncime asociată grafului și care provin din nodul  $x$ , sunt sau arce “de arbore”, sau arce de tip “înainte”, sau arce de “trecere”.
  - Toate aceste arce sunt însă direcționate spre noduri a căror vizită s-a încheiat și în consecință, evidențierea lor precede evidențierea lui  $x$  în cadrul ordinii care se construiește.

## 12.6. Componente puternic conectate

- O **componentă puternic conectată** a unui **graf orientat** este o submulțime de noduri ale grafului în care există un drum de la **oricare** nod al mulțimii la **oricare** alt nod aparținând aceleiași mulțimi.
- Fie  $G = (N, A)$  un **graf orientat**.
- Se partăzonează mulțimea  $N$  a nodurilor grafului  $G$  în **clase de echivalență**  $N_i$ , ( $1 \leq i \leq r$ ), pe baza următoarei **definiții a relației de echivalență**:
  - Nodurile  $x$  și  $y$  sunt **echivalente** dacă și numai dacă există un drum de la nodul  $x$  la nodul  $y$  și un drum de la nodul  $y$  la nodul  $x$ .
- Fie  $A_i$ , ( $1 \leq i \leq r$ ) mulțimea arcelor ale căror surse și destinații aparțin mulțimii  $N_i$ .
- Grafurile  $G_i = (N_i, A_i)$  se numesc **componentele puternic conectate** ale grafului  $G$ , sau mai simplu **componente puternice**.
- Un **graf orientat** care constă dintr-o singură componentă puternică se numește **puternic conectat**.
- În figura 12.6.a apare un graf orientat (a), care conține două componente puternice (b).



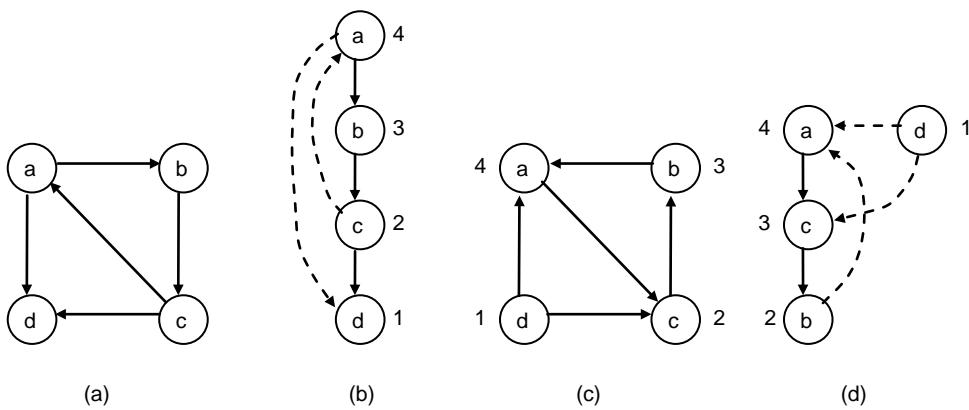
**Fig.12.6.a.** Componete puternic conectate și graful redus al unui graf orientat

- Se face precizarea că fiecare **nod** al unui graf orientat aparține unei **componente puternice** dar există **arce** care **nu** aparțin nici unei componente.
  - Astfel de arce se numesc **arce de “transfer”** și ele conectează noduri aparținând la **componente puternice** diferite.

- Considerând **componentele puternice** drept **noduri** și reprezentând interconexiunile dintre componentele puternice ale unui graf  $G$ , respectiv reprezentând **arcele de transfer**, se obține **graful redus** al grafului  $G$ .
  - Nodurile **grafului redus** sunt **componentele puternice** ale grafului orginal.
  - În **graful redus** apare un arc de la componenta puternică  $C_i$  la componenta puternică  $C_j$  dacă în graf original  $G$  există un arc ce leagă un nod aparținând lui  $C_i$  cu un nod aparținând lui  $C_j$ .
  - **Graful redus** este întotdeauna un **graf orientat aciclic**, deoarece dacă ar conține vreun ciclu, atunci toate componentele din acel ciclu aparțin unei aceleleași componente puternic conectate, element ce ar evidenția faptul că determinarea componentelor puternice pentru graf original s-a realizat defectuos.
- În figura 12.6.1. (c) apare graful redus al grafului orientat din fig. 12.6.1. (a).
- În continuare se vor prezenta doi algoritmi pentru determinarea **componentelor puternic conectate ale unui graf orientat**, ambii bazați pe **tehnica căutării în adâncime**.

### 12.6.1. Algoritmul lui Kosaraju-Sharir

- Algoritmul pentru determinarea **componentelor puternic conectate** ale unui **graf orientat**  $G$  a fost sugerat în anul 1978 de R.Kosaraju (nepublicat) și publicat în anul 1981 de către Sharir, motiv pentru care a fost denumit **algoritmul Kosaraju-Sharir**.
- **Algoritmul Kosaraju-Sharir** constă din următorii pași:
  - (1) Se realizează o **traversare prin căutare în adâncime** a grafului  $G$  și se **numerotează nodurile** în ordinea terminării apelurilor recursive corespunzătoare lor.
    - Acest lucru se poate realiza, spre exemplu, numerotând nodul  $x$  după linia [5] a procedurii **CautInAdancime** din secvența [12.4.1.a].
  - (2) Se construiește un nou **graf orientat**  $G_r$ , **inversând sensul** tuturor arcelor grafului original  $G$ .
  - (3) Se realizează o **traversare prin căutare în adâncime** în graf  $G_r$ , începând cu **nodul** care are **numărul cel mai mare** conform numerotării de la pasul 1.
  - (4) Dacă această traversare **nu** acoperă toate nodurile, se lansează următoarea traversare începând cu **nodul** care are **numărul cel mai mare** dintre nodurile neparcuse încă.
  - (5) Se continuă în aceeași manieră până la epuizarea tuturor nodurilor.
- Fiecare **arbore de căutare în adâncime** al **pădurii** care rezultă, este o **componentă puternică conectată** a grafului  $G$ .
- În figura 12.6.1.a este ilustrată aplicarea algoritmului Kosaraju-Sharir asupra grafului orientat (a).



**Fig.12.6.1.a.** Determinarea componentelor puternic conectate ale unui graf orientat

- Astfel, după traversarea grafului începând cu nodul  $a$  și continuând cu  $b$ , se obține arborele de căutare în adâncime și numerotarea nodurilor din fig. 12.6.1.a. (b).
- Inversând sensurile arcelor grafului original, rezultă graful  $G'$  (fig.12.6.1.a. (c)).
- În continuare, realizând o traversare prin căutare în adâncime a lui  $G'$  rezultă pădurea din aceeași figură (d).
- Într-adevăr, traversarea începe cu nodul  $a$ , considerat că rădăcină, deoarece  $a$  are cel mai mare număr.
  - Din  $a$  se ajunge la nodul  $c$  și la nodul  $b$ .
  - Următorul arbore de căutare în adâncime are rădăcina  $d$ , deoarece  $d$  este nodul cu numărul cel mai mare dintre cele rămase neparcurse.
- Fiecare arbore al acestei păduri formează o **componentă puternic conectată** a grafului orientat original  $G$ .
- S-a afirmat că nodurile unei componente puternic conectate a unui graf, corespund exact nodurilor unui arbore al pădurii arborilor de acoperire rezultate în urma celei de-a doua traversări a grafului, respectiv a traversării grafului inversat.
- Pentru a **demonstra** acest lucru se pornește de la următoarea observație:
  - Dacă  $x$  și  $y$  sunt noduri aparținând unei aceleleași componente puternice, atunci în  $G$  există un drum de la  $x$  la  $y$  și un drum de la  $y$  la  $x$ .
  - În consecință și în graful inversat  $G'$  există un drum de la  $x$  la  $y$  și un drum de la  $y$  la  $x$ .
- Se presupune că în traversarea lui  $G'$ , căutarea începe cu o rădăcină  $r$  și că se ajunge la  $x$  sau  $y$ .
  - Deoarece  $x$  și  $y$  sunt conectați în ambele sensuri, atât  $x$  cât și  $y$  vor aparține arborelui de căutare în adâncime cu rădăcina  $r$ .
- Presupunând în continuare că  $x$  și  $y$  aparțin unui același arbore de căutare în adâncime derivat din  $G'$ , trebuie demonstrat că cele două noduri aparțin și aceleleași componente puternic conectate.
  - Fie  $r$  rădăcina arborelui de căutare în adâncime în  $G'$  care-i conține pe  $x$  și pe  $y$ .
  - Deoarece  $x$  este un descendent al lui  $r$ , rezultă că în  $G'$  există un drum de la  $r$  la

$x$ . În consecință în  $G$  există un drum de la  $x$  la  $r$ .

- Trebuie demonstrat în continuare că în  $G$  există și un drum de la  $r$  la  $x$ .
- În construcția pădurii de căutare în adâncime corespunzătoare lui  $Gr$ , nodul  $x$  era nevizitat în momentul în care se iniția căutarea pentru  $r$ .
  - Acest lucru este motivat de faptul că  $r$  are un număr mai mare ca și  $x$ , deoarece în căutarea în adâncime a lui  $G$  apelul recursiv al lui  $x$  se termină înaintea apelului recursiv a lui  $r$ .
- Se pune următoarea întrebare: "Ar fi putut, în căutarea în adâncime a lui  $G$  ca vizitarea nodului  $x$  să fie demarată înainte de vizitarea nodului  $r$ ?"
  - Acest lucru este imposibil întrucât existența drumului în  $G$  de la  $x$  la  $r$  presupus anterior, ar implica faptul că vizitarea lui  $r$  să înceapă și să se termine înaintea terminării vizitării lui  $x$ .
- În concluzie, în traversarea lui  $G$ ,  $x$  este vizitat în timpul căutării lui  $r$ , astfel  $x$  este un descendent al lui  $r$  în arborele de căutare în adâncime al lui  $G$ .
  - Deci există un drum de la  $r$  la  $x$  în  $G$  și în consecință nodurile  $x$  și  $r$  aparțin unei aceleleași componente puternic conectate a grafului  $G$ .
- Printr-o argumentare identică se demonstrează că și  $r$  și  $y$  aparțin unei aceleleași componente puternic conectate.
  - Faptul că în ambele situații este vorba de una și aceeași componentă este certificat de existența drumului  $x \rightarrow r \rightarrow y$ , respectiv de existența drumului  $y \rightarrow r \rightarrow x$  [AH 85].

### 12.6.2. Algoritmul lui Tarjan

- O altă metodă ingenioasă de determinare a **componentelor puternic conectate** ale unui graf orientat a fost publicată de R.E. Tarjan în anul 1972.
- Metoda se bazează tot pe **tehnica căutării în adâncime** și o variantă de implementare a sa apare în secvența [12.6.2.a] în forma funcției **Tarjan**.
- În legătură cu această secvență se fac câteva precizări.
- (1) Graful se consideră reprezentat printr-o **structură de adiacențe** implementată cu ajutorul listelor înlăntuite simple.
- (2) Funcția **Tarjan** prezentată, utilizează variabila  $min$  pentru a determina **cel mai înalt nod** care poate fi atins prin intermediul unui arc de tip "înapoi", pentru orice descendent al nodului  $k$  pe care îl primește ca și parametru.
  - Acest lucru este realizat într-o manieră similară funcției care determină **componentele biconexe** ale unui graf (secvența [10.5.2.3.a], capitolul 10).

```
-----  
int Tarjan(int k)
```

```
/*Determinarea componentelor puternice ale unui graf*/
```

```
tip_legatura t;  
int m,min;
```

```

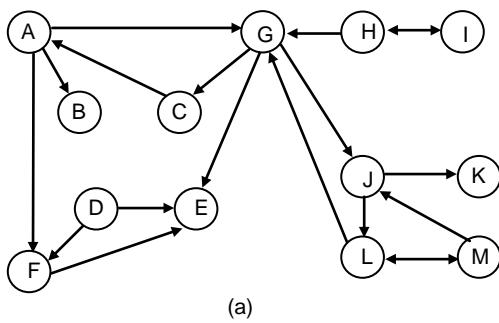
        id=id+1; marc[k]=id; /*marcare nod k*/
        Stiva[p]=k; /*introducere nod k în stiva*/
        p=p+1; /*avans indicator stiva*/
        t=Stradj[k]; /*inițiere parcurgere lista de adiacențe*/
cat timp (t<>NULL)
    daca (marc[t^.nume]==0)
        m=Tarjan(t^.nume);
    altfel
        m=marc[t^.nume];
    daca (m<min)                                /*[12.6.2.a]*/
        min=m;
        t=t^.urm;
    □ /*cat timp*/
daca (min==marc[k]) /*evidențiere componentă puternică*/
    repeta
        p=p-1;
        *scrive(Stiva[p]);
        marc[Stiva[p]]=N+1; /*N este numărul de noduri*/
        pana cand (Stiva[p]==k);
    □ /*repeta*/
        *scrive_rand;
    □ /*daca*/
return min;
/*Tarjan*/

```

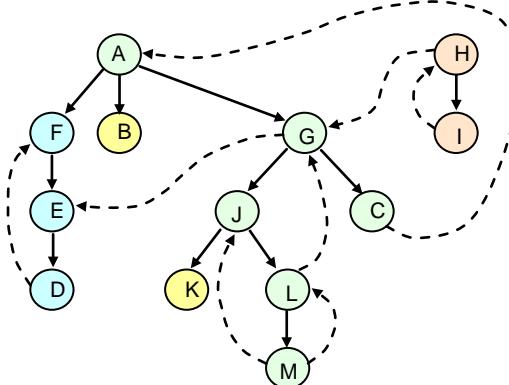
---

- În plus însă, valoarea `min` determinată, este utilizată și la evidențierea **componentelor puternic conectate**.
- În acest scop se utilizează **o stivă** implementată ca un tablou `Stiva` controlată de indicele `p`.
  - În această stivă se introduc numele (numerele) nodurilor pentru care este apelată funcția `Tarjan` imediat după intrarea în apel.
  - Nodurile care aparțin unei componente puternice, sunt afișate prin extragere din stivă, după vizitarea ultimului membru al componentei puternic conectate (bucla **repeta**).
- **Elementul esențial** al determinării este verificarea `min=marc[k]` efectuată la terminarea apelurilor recursive:
  - Dacă acest test conduce la valoarea de adevăr, se afișează toate nodurile din stivă până la nodul `k` inclusiv, întrucât ele aparțin aceleiași componente puternic conectate ca și nodul `k`.
- Acest algoritm poate fi însă adaptat să realizeze prelucrări mult mai sofisticate decât simpla afișare a componentelor puternic conectate.
- Fiind bazat pe tehnica căutării în adâncime, în grafuri reprezentate prin structuri de adiacențe, algoritmul lui Tarjan obține o **performanță** de ordinul  $O(n+a)$ .
- Demonstrația riguroasă a funcționalității algoritmului este în afara scopului cursului, însă vor fi schițate ideile care o fundamentează.
- **Metoda lui Tarjan** se bazează pe **două observații** care au fost subliniate și cu alte prilejuri.

- (1) Prima observație se referă la faptul că la terminarea apelului funcției **Tarjan** pentru un nod  $x$ , **nu** mai pot fi întâlnite alte noduri care să aparțină aceleleași componente puternic conectate ca și nodul respectiv deoarece toate nodurile la care se poate ajunge plecând de la  $x$  au fost deja parcuse.
- (2) A doua observație se referă la faptul că un arc de tip “înapoi” al arborelui de căutare în adâncime precizează un alt drum de la un nod la altul în cadrul arborelui și el de fapt leagă elementele componentei puternice.
- Exact ca și în cazul determinării **punctelor de articulație** ale unui graf și în acest caz se păstrează urma **celui mai înalt ascendent**, la care se poate ajunge via un arc de tip “înapoi”, pentru toți descendenții fiecărui nod.
- **Interpretarea rezultatului** algoritmului lui **Tarjan** este următoarea.
  - (1) Un nod  $x$  care nu are descendenți sau legături de tip înapoi în arborele de căutare în adâncime, determină o **componentă puternic conectată**.
  - (2) Dacă un nod  $x$  are un **descendent** în arborele de căutare în adâncime din care pornește un arc de tip “înapoi” spre  $x$  și nu are niciun descendant din care să pornească vreun arc de tip “înapoi” spre vreun nod situat **deasupra** lui  $x$  în arborele de căutare în adâncime, atunci nodul  $x$  împreună cu toți descendenții săi (cu excepția acelor noduri care satisfac situația (1), respectiv a nodurilor care satisfac situația (2) și a descendenților lor), determină o **componentă puternic conectată**.
  - (3) Fiecare descendant  $y$  al nodului  $x$  care nu satisfac nici una din cele două situații mai sus precizate, are descendenți care sunt sursa unor arce de tip “înapoi” care ajung la noduri mai înalte ca  $x$  în arborele de căutare în adâncime.
    - Pentru un astfel de nod, există un drum direct de la  $x$  la  $y$  rezultat din parcurgerea arborelui de căutare în adâncime.
    - Mai există însă un drum de la nodul  $y$  la nodul  $x$  care poate fi determinat coborând în arbore de la  $y$  în jos până la nodul descendant de la care printr-un arc de tip “înapoi” se ajunge la un strămoș al lui  $y$  și apoi continuând în aceeași manieră, se ajunge până la  $x$ .
    - Rezultă că nodul  $y$  aparține aceleleași **componente puternic conectate** ca și nodul  $x$ .
- Aceste situații pot fi urmărite în figura 12.6.2.a care reprezintă **pădurea de arbori de căutare în adâncime** (b) corespunzătoare grafului orientat (a).
  - Nodurile  $B$  și  $K$  satisfac prima situație, astfel încât se constituie ele însele în componente puternice ale grafului.
  - Nodurile  $F$  (reprezentând pe  $F, E$  și  $D$ ),  $H$  (reprezentând pe  $H$  și  $I$ ) și  $A$  (reprezentând pe  $A, G, J, L, M$  și  $C$ ) satisfac a doua situație.
    - Membrii componentei evidențiate de  $A$  au fost determinați după eliminarea nodurilor  $B, K, F$  și  $H$  și a descendenților lor care apar în componente puternice stabilite anterior.
    - Restul nodurilor se încadrează în situația a treia.



(a)



(b)

**Fig.12.6.2.a.** Graf orientat (a) și pădure de arbori de căutare în adâncime asociată (b)

- O subliniere foarte importantă este aceea ca odată un nod parcurs, el primește în tabloul **marc** o valoare mare ( $N+1$ ), astfel încât arcele de “trecere” spre aceste noduri sunt ignorate.

## 12.7. Rețele de curgere ("Network-Flow")

- Grafurile orientate ponderate sunt modele foarte utile pentru anumite tipuri de aplicații inclusiv pentru **modelarea fluxului materialelor** în cadrul unei **rețele interconectate**.
- Se poate considera spre exemplu o **rețea de conducte de petrol** de diferite dimensiuni interconectate într-o manieră complexă, cu comutatoare controlând direcția fluxului pe juncțiuni.
  - Se presupune în continuare că o astfel de rețea are o singură sursă (un câmp petrolifer) și o singură destinație (o mare rafinărie) la care în final sunt conectate toate conductele.
  - Întrebarea care se pune este următoarea: “Ce configurație de poziționare a comutatoarelor de control conduce la un **flux maxim** de la sursă la destinație?”.
  - Interacțiunile complexe care se manifestă la nivelul juncțiunilor fac ca **problema rețelelor de curgere ("network flow problem")** să nu fie una simplu soluționabilă.
  - Astfel de rețele sunt cunoscute și sub denumirea de **rețele de transport** [FK 69].
  - Același scenariu general, poate fi utilizat la:

- (1) Modelarea traficului rutier de-a lungul autostrăzilor unei rețele de drumuri
  - (2) Modelarea fluxului materialelor într-o întreprindere
  - (3) Modelarea fluxului datelor în rețelele de comunicații.

• Pentru rezolvarea acestei probleme au fost propuse mai multe soluții, care sunt mai mult sau mai puțin satisfăcătoare funcție de circumstanțe.

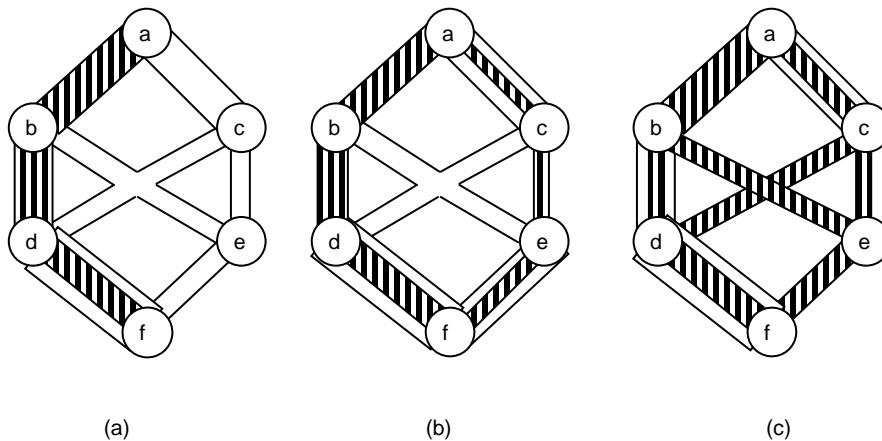
  - De fapt, această problemă este încă intens studiată și soluția cea mai bună nu a fost încă determinată.

• În cadrul capitolului de față se prezintă o soluție clasică a problemei rețelelor de curgere, strâns legată de algoritmii referitori la structura de date graf, dezvoltată în cadrul capitolelor anterioare.

  - În același timp acesta este un exemplu de rezolvare a unei probleme utilizând unele algoritmice deja consacrate.

### **12.7.1. Problema rețelelor de curgere**

- În fig. 12.7.1.a apare o reprezentare idealizată a unei mici rețele de conducte de petrol.

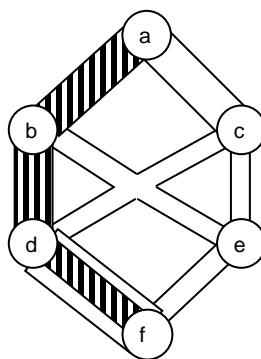


**Fig.12.7.1.a.** Diverse fluxuri printr-o rețea de curgere simplă

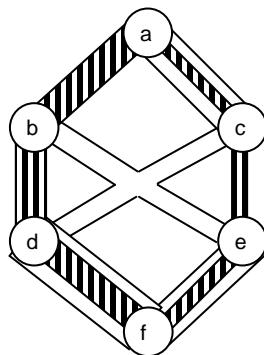
- Se fac următoarele specificații:
    - (1) Conductele sunt de **capacitate fixă**, proporțională cu grosimea lor și pot fi parcurse într-un singur sens (în figură de sus în jos).
    - (2) Comutatoarele distribuitoare situate la fiecare juncțiune dirijează **cantitatea de petrol** care se scurge pe fiecare dintre direcțiile posibile.
    - (3) Indiferent de poziționarea comutatoarelor, sistemul se găsește într-o **stare de echilibru** atunci când:
      - (3.1) Cantitatea de petrol care intră în sistem la partea sa superioară,

este egală cu cantitatea de petrol care părăsește sistemul la partea sa inferioară. Aceasta este cantitatea pentru care trebuie determinată **valoarea maximă**.

- (3.2) Cantitatea de petrol care intră în fiecare joncțiune este egală cu cantitatea care ieșe din joncțiune.
- (4) Atât fluxul curent prin conductă, cât și capacitatea sa maximă vor fi exprimate în valori întregi, spre exemplu litri pe secundă.
- La prima vedere **nu** pare evident faptul că poziționarea comutatoarelor afectează fluxul maxim.
- În figura 12.7.1.a se demonstrează că acest lucru este posibil.
- În situația (a), comutatoarele care controlează conductele ab și bd sunt deschise integral, astfel încât acestea funcționează la capacitatea maximă iar conducta df numai parțial.

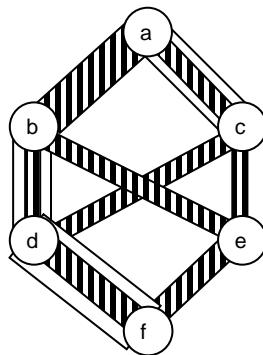


- În situația (b) se deschid înapoi plus conducta ac (parțial), ce (total) și ef (parțial).
  - Fluxul prin rețea este mai mare acum începând atât conducta bd cât și conducta ce funcționează la capacitatea maximă.
  - Acest flux ar putea fi mărit deschizând traseul acdf astfel încât conducta df să funcționeze și ea la capacitatea maximă.



- După cum se vede însă din situația (c) există o soluție și mai bună.

- Modificând comutatorul b și permitând în acest mod curgerea petrolului prin conducta be la capacitatea maximă a acesteia, se crează posibilitatea utilizării la capacitate maximă și a conductei cd.

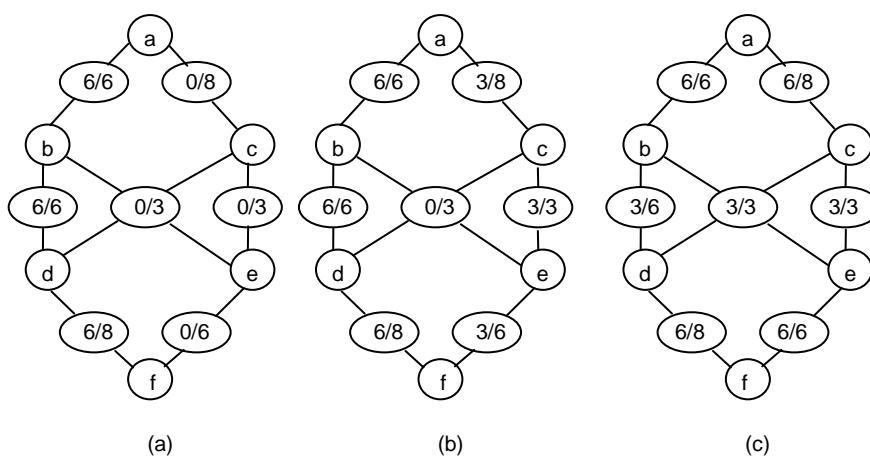


- Ca atare, după cum se observă, poziționarea comutatoarelor distribuitoare influențează în mod evident fluxul total prin rețea.
- Problema rețelelor de curgere** constă în a determina acea poziționare a comutatoarelor pentru orice rețea de acest tip, care conduce la **fluxul maxim**.
  - De asemenea trebuie dovedit faptul că nici o altă poziționare **nu** poate obține un flux mai mare.
- Această situație poate fi în mod evident modelată printr-un **graf orientat**.
- Astfel, o **rețea de curgere** se definește ca și un **graf orientat ponderat** a cărui mulțime a nodurilor conține în afara celor normale, două noduri speciale:
  - (1) Un nod sursă în care **nu** intră nici un arc
  - (2) Un nod destinație din care **nu** pleacă nici un arc.
- Ponderile arcelor se numesc **capacități** și se presupun pozitive.
- Fluxurile** (debitele) sunt definite printr-un alt set de ponderi asociate fiecărui arc.
  - Acstea ponderi sunt supuse la două restricții:
    - (1) Fluxul într-un arc este mai mic sau cel mult egal cu capacitatea arcului respectiv
    - (2) Fluxul la intrare în oricare nod este egal cu fluxul de la ieșirea nodului respectiv.
  - Valoarea **fluxului rețelei** este fluxul din nodul sursă sau nodul destinație al rețelei.
- Problema rețelelor de curgere** constă în determinarea **valorii maxime** a fluxului rețelei.
- Fiind modelate prin grafuri, rețelele pot fi reprezentate fie prin intermediul matricilor de adiacențe fie prin intermediul structurilor de adiacențe.

- În plus însă, în locul unei singure ponderi fiecărui arc îi sunt asociate **două ponderi** respectiv **capacitatea și fluxul**.
- Această cerință poate fi rezolvată prin definirea a **două câmpuri specifice** în structura nodului unei liste de adiacențe, sau prin utilizare a **două matrici** în reprezentarea bazată pe matrici de adiacențe.
- De asemenea în ambele reprezentări poate fi utilizată ca și alternativă o structură de tip articol cu două câmpuri corespunzătoare celor două ponderi.
- Chiar dacă rețelele sunt de fapt grafuri orientate, algoritmul care rezolvă problema rețelelor de curgere necesită traversarea grafului și în sens invers sensului arcelor, motiv pentru care în reprezentare se va utiliza un **graf neorientat**.
  - Astfel, pentru fiecare arc de la  $x$  la  $y$  aparținând rețelei, cu capacitatea  $c$  și fluxul  $f$ , se va păstra și imaginea arcului de la  $y$  la  $x$  având asociate ponderile  $-c$  și  $-f$ .
  - În acest context, în reprezentarea bazată pe **structuri de adiacențe**, este necesar să fie prevăzute legături care conectează nodurile aparținând celor două liste de adiacențe care reprezintă același arc, astfel încât modificarea fluxului într-una dintre reprezentări să poată fi imediat actualizată și în cealaltă.

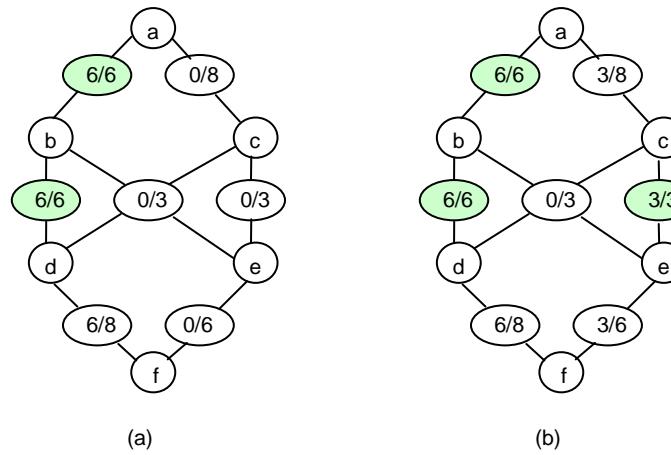
### 12.7.2 Metoda Ford-Fulkerson

- Metoda clasică de rezolvare a problemei rețelelor de curgere a fost dezvoltată de către L.R.Ford și D.R.Fulkerson în anul 1962.
  - Ei au furnizat o metodă care îmbunătățește orice flux legal prin rețea, evident cu excepția celui maxim.
- Practic se pornește de la **valoarea zero a fluxului** și se aplică metoda în mod repetat.
  - Atâtă vreme cât metoda poate fi aplicată, ea conduce la obținerea unor fluxuri crescătoare, iar când nu mai poate fi aplicată s-a obținut fluxul maxim.
- De fapt fluxul maxim din figura 12.7.1.a a fost obținut aplicând această metodă.
- În continuare se reexaminează aceeași situație în termenii grafurilor reprezentate în figura 12.7.2.a.
  - Pentru fiecare arc al grafului se indică într-o elipsă ponderile asociate în formatul **flux/capacitate** ( $f / c$ ).
  - Pentru simplificare, săgețile arcelor au fost omise, considerându-se că toate indică sensul “în jos”.



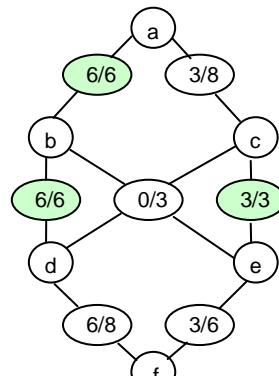
**Fig.12.7.2.a.** Determinarea fluxului maxim într-o rețea de curgere

- Se face însă precizarea că aplicabilitatea metodei care va fi dezvoltată în continuare **nu este restrânsă numai la grafuri în care toate săgețile indică aceeași direcție**.
  - Pentru prezentarea metodei însă, s-a adoptat o astfel de soluție deoarece aceasta este foarte intuitivă.
  - Cu această simplificare, practic fluxul într-o rețea de curgere poate fi exprimat natural în termenii unui lichid care străbate conductele rețelei.
- Se consideră **orice drum orientat** "în jos" prin rețea de la sursă la destinație.
  - Este evident faptul că pe orice drum fluxul poate fi crescut până la **capacitatea maximă a celei mai înguste conducte de pe traseu**, ajustând în mod corespunzător fluxul în toate arcele care compun drumul.
    - În figura 12.7.2.a a fost aplicată această regulă de-a lungul drumului abdf (situația (a)), respectiv și de-a lungul drumului acef (situația (b)).

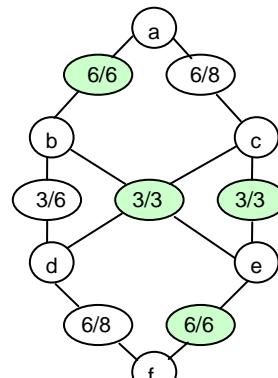


- În continuare, regula poate fi aplicată și altor drumuri, creând spre exemplu situația în care toate drumurile orientate din rețea au cel puțin un arc care funcționează la capacitatea maximă.
- Există însă și **o altă metodă** de creștere a fluxului.

- Se consideră **drumuri arbitrale** prin rețea care pot conține și arce orientate în sens invers (respectiv de la destinație spre sursă).
  - Fluxul poate fi crescut de-a lungul unui astfel de drum, crescând fluxul în arcele care au sensul de la sursă spre destinație și diminuând cu aceeași valoare, fluxul în arcele direcționate în sens invers.
  - Spre exemplu, după cum se observă în figura 12.7.2.a (c), fluxul prin rețea poate fi crescut cu 3 unități de-a lungul drumului  $a \rightarrow c \rightarrow d \rightarrow b \rightarrow e \rightarrow f$  față de situația (b).



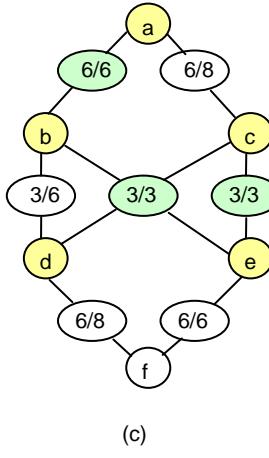
(b)



(c)

- Aceasta rezultă din suplimentarea fluxului prin ac și cd cu câte 3 unități la fiecare și prin reducerea fluxului cu 3 unități prin bd cu redirectarea acestuia pe traseul be și ef.
  - De fapt nu se pierde din fluxul de pe ramura df întrucât cele 3 unități care vedea dinspre bd vin acum pe ramura cd .
  - Pentru a simplifica terminologia:
    - Arcele direcționate de la **sursă** spre **destinație** se numesc arce de tip “înainte”.
    - Arcele orientate de la **destinație** spre **sursă** se numesc arce de tip “înapoi”.
  - Este important de subliniat faptul că valoarea cu care poate fi crescut fluxul este limitată de
    - (1) **Minimul** dintre capacitatele neutilizate ale arcelor de tip “înainte”
    - (2) **Minimul** fluxurilor în arcele de tip “înapoi” de pe traseu.
  - Cu alte cuvinte, în noul flux, cel puțin unul din arcele de tip înapoi situate pe traseu funcționează la capacitatea maximă sau cel puțin unul dintre arcele de tip “înapoi” situat de-a lungul traseului devine gol.
  - Ca atare fluxul **nu** mai poate fi crescut de-a lungul unui traseu care conține un arc “înainte” plin sau un arc “înapoi” gol.

- Observațiile formulate mai sus sugerează **principiul metodei** de creștere a fluxului într-o rețea de curgere.
  - Aceasta se referă la determinarea acelor drumuri care conțin arce “înainte” care **nu** sunt pline sau arce “înapoi” care **nu** sunt goale.
- **Elementul fundamental** al **metodei Ford-Fulkerson** constă în observația că dacă **fluxul în rețea este maxim**, în rețea **nu** există nici un astfel de drum.
  - Cu alte cuvinte, dacă **orice drum** de la sursă la destinație dintr-o rețea, conține un arc “înainte” plin sau un arc “înapoi” gol, atunci **fluxul în rețea este maxim**.
- Pentru a demonstra acest fapt, în primul rând, se inspectează graful și se determină **primul arc “înainte” plin** sau **primul arc “înapoi” gol** pe fiecare drum.
  - Mulțimea arcelor astfel determinate **taie** graful în două părți.
    - Spre exemplu din figura 12.7.2.a (c) arcele  $ab$ ,  $cd$  și  $ce$  realizează această tăietură.



- Pentru orice tăietură a rețelei se poate determina fluxul prin tăietură.
  - Acesta este de fapt **fluxul total** prin arcele tăieturii care merg de la sursă spre destinație.
  - În general, arcele pot traversa în ambele sensuri tăietura.
  - Pentru determinarea efectivă a fluxului se adună fluxurile care trec spre destinație și se scad fluxurile din arcele care traversează în sens invers tăietura.
    - În cazul prezentat, fluxul prin tăietură are valoarea 12 și este egal cu fluxul prin rețea.
- Rezultă că, ori de câte ori fluxul prin tăietură egalează fluxul total, pe de o parte fluxul este **maxim** iar pe de altă parte tăietura este **minimă**, ceea ce înseamnă că orice altă tăietură are fluxul cel puțin la fel de mare.
  - Aceasta este teorema ”**flux maxim – tăietură minimă**” numită și **teorema lui Ford-Fulkerson**.

- Obs: Prin **tăietură minimă** se înțelege capacitate neutilizată minimă.
  - Drept urmare:
    - (1) Fluxul nu poate fi oricât de mare - altfel tăietura ar putea fi și ea mai mică.
    - (2) Nu există tăieturi mai mici - altfel și fluxul ar putea de asemenea să aibă valori mai mari [Se 88].
  - **Teorema Ford-Fulkerson** se mai poate enunța și în următoarea formă:
    - Într-o rețea de curgere dată, **valoarea maximă** a unui **flux** este egală cu **capacitatea minimă** a unei **tăieturi** [FK 69].
- 12.7.3. Implementarea metodei Ford-Fulkerson. Căutarea în rețea**
- Metoda Ford-Fulkerson descrisă anterior poate fi rezumată după cum urmează:
    - (1) Se pornește cu flux zero peste tot și se crește fluxul de-a lungul oricărui drum de la sursă la destinație care **nu** conține **arce înainte pline** sau **arce înapoi goale**.
    - (2) Operațiunea continuă până când **nu** mai există nici un astfel de drum în rețea.
  - După cum se observă însă acesta **nu** este un algoritm în sensul strict al cuvântului deoarece metoda de determinare a drumurilor **nu** este specificată, practic putând fi selectată orice succesiune de drumuri.
    - Spre exemplu la baza selecției ar putea sta observația că cu cât drumul este mai lung, cu atât rețeaua va fi mai repede umplută, astfel încât sunt de preferat drumurile cele mai lungi.
    - Indiferent însă de strategia abordată, maniera de selectare a drumurilor are o influență decisivă asupra performanței metodei.
  - În anul 1972 **Edmonds și Karp** au demonstrat următoarea **proprietate**.
    - Dacă în metoda Ford-Fulkerson se utilizează **cel mai scurt drum curent disponibil** de la sursă la destinație, atunci numărul de pași necesari determinării fluxului maxim într-o rețea cu  $n$  noduri și  $a$  arce este mai mic decât produsul  $na$ .
      - Demonstrația acestei proprietăți depășește scopul cursului de față.
    - În consecință, o posibilitate de a aborda rezolvarea acestei probleme o reprezintă utilizarea **tehnicii de căutare “prin cuprindere”** pentru găsirea **drumurilor minime**.
    - Proprietatea anterior enunțată precizează o margine superioară pentru performanța acestui algoritm.
      - În realitate, pentru rețelele obișnuite sunt necesari mult mai puțini pași.
    - Pornind însă de la traversarea grafurilor prin **tehnica căutării bazate pe priorităte**

abordată în capitolul anterior se poate implementa și o altă metodă sugerată tot de Edmonds și Karp și anume:

- Găsirea aceluia drum din rețea care determină creșterea fluxului cu cea mai mare valoare posibilă.
- Această obiectiv poate fi atins simplu utilizând pentru **prioritate**, în cadrul metodei dezvoltate în secvența [11.4.a] din capitolul &11, o valoare corespunzătoare.

```
-----  
PROCEDURE CautPrioritar;  
VAR k,min,t: integer;  
  
BEGIN  
    FOR k:= 1 TO N DO  
        BEGIN  
            marc[k]:= -nevazut;                                [11.4.a]  
            parinte[k]:= 0  
        END;  
        marc[0]:= -(nevazut + 1);  
        min:= 1;  
        REPEAT  
            k:= min; marc[k]:= -marc[k]; min:= 0;  
            IF marc[k] = nevazut THEN marc[k]:= 0;  
            FOR t:= 1 TO N DO  
                IF marc[t] < 0 THEN  
                    BEGIN  
                        IF(A[k,t] <> 0) AND (marc[t] < -prioritate) THEN  
                            BEGIN  
                                marc[t]:= -prioritate;  
                                parinte[t]:= k  
                            END;  
                            IF marc[t] > marc[min] THEN min:= t  
                        END  
                    UNTIL min = 0  
    END; {CautPrioritar}  
-----
```

- Spre exemplu, pentru grafurile reprezentate prin matrice de adiacențe, **determinarea valorii priorității** se face pe baza următorului calcul [12.7.3.a]:

```
-----  
IF capacitate[k,t] > 0 THEN  
    prioritate:= capacitate[k,t] - flux[k,t];  
ELSE                                              [12.7.3.a]  
    prioritate:= -flux[k,t];  
IF prioritate > marc[k] THEN prioritate:= marc[k];  
-----
```

- În continuare, pentru a adapta procedura CautPrioritar din secvența [11.4.a] noului scop, trebuie operate unele modificări.
  - (1) Deoarece se dorește selecția nodului cu cea mai mare prioritate, este necesară:
    - (a) Fie modificarea mecanismului cozii bazate pe prioritate astfel încât

acesta să returneze maximul în locul minimului

- (b) Fie utilizarea pentru prioritate a expresiei maxint-1-prioritate (maxint fiind întregul maxim reprezentabil).
- (2) Este necesar a fi modificată procedura de căutare bazată pe prioritate astfel încât să primească drept date de intrare nodul sursă și nodul destinație.
  - În acest fel orice căutare pornește de la nodul sursă și se termină când se determină un drum până la nodul destinație.
  - Dacă **nu** se găsește un astfel de drum, arborele de căutare parțial care a fost determinat definește o **tăietură minimă** pentru rețea.
  - Dacă se găsește un astfel de drum, atunci fluxul prin rețea poate fi crescut în continuare.
- (3) Valoarea inițială din tabloul **marc** corespunzător nodului sursă trebuie poziționată pe **maxint** pentru a preciza faptul că practic de la sursă se poate obține orice flux.
  - În realitate această valoare este imediat restrânsă la capacitatea maximă a tuturor conductelor care părăsesc sursa.
- Implementând în procedura **CautPrioritar** (secvența [11.4.a]), modificările descrise anterior, determinarea fluxului maxim poate fi realizată conform secvenței [12.7.3.b].

---

#### **REPEAT**

```
CautPrioritar(1,N);
y:= N; x:= parinte[N];
WHILE x <> 0 DO
    BEGIN                                     [12.7.3.b]
        flux[x,y]:= flux[x,y] + marc[N];
        flux[y,x]:= -flux[x,y];
        y:= x; x:= parinte[y]
    END
UNTIL marc[N] = 1 - maxint;

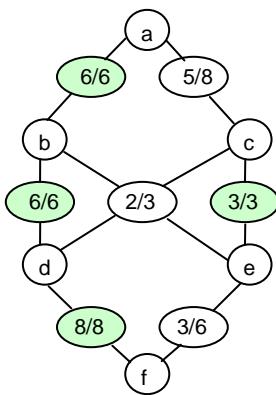

---


```

- În cadrul secvenței se presupune că rețeaua este reprezentată printr-o **matrice de adiacențe**.
  - Atâtă vreme cât **CautPrioritar** poate determina un drum care crește fluxul cu cantitatea maximă posibilă, se revine de-a lungul drumului utilizând tabloul **părinte** construit de procedura **CautPrioritar** și se crește fluxul după cum rezultă din secvența [12.7.3.b], bucla **WHILE**.
  - Dacă **N** (nodul destinație) nu poate fi atins după câteva apeluri ale procedurii **CautPrioritar**, atunci s-a determinat o **tăietură minimă** și algoritmul se termină.
  - După cum s-a observat din fig.12.7.2.a, algoritmul crește mai întâi fluxul de-a lungul

drumului abdf, apoi de-a lungul drumului acef și în cele din urmă de-a lungul lui acdbef.

- De notat faptul că traseul acdf nu este ales pentru al treilea pas deoarece el ar crește fluxul cu două unități și nu cu trei unități, după cum se realizează în pasul selectat.



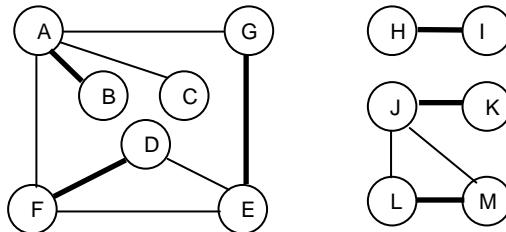
- Deși acest algoritm este simplu de implementat și relativ ușor de aplicat în practică pentru rețelele utilizate în mod curent, analiza sa este complicată.
  - În primul rând, CautPrioritar necesită un efort proporțional cu  $O(n^2)$  în cel mai defavorabil caz.
  - Ca și alternativă poate fi utilizat algoritmul ArboreMinim (secvența [11.2.2.a]), dezvoltat pentru reprezentarea grafurilor ca și structuri de adiacențe, care necesită un timp proporțional cu  $O((a+n) \log n)$  pentru fiecare iterație.
- Oricum este de așteptat că algoritmul din secvența [12.7.3.b] să dureze mai puțin decât procedura ArboreMinim (secvența [11.2.2.a]) deoarece spre deosebire de cea din urmă care determină toate drumurile care aparțin arborelui de acoperire minim, algoritmul de față se oprește la determinarea drumului care conduce la destinație.
- Problema care se pune în continuare se referă la numărul necesar de iterări.
- O marginie superioară pentru acest număr este furnizată de următoarea **proprietate** enunțată tot de Edmonds și Karp.
  - Dacă în metoda Ford-Fulkerson se folosește drumul care crește fluxul cu cantitatea maximă posibilă, atunci numărul de pași necesari determinării fluxului maxim într-o rețea este mai mic decât  $1 - 1 \log_{M/M-1} f^*$  unde  $f^*$  reprezintă costul fluxului și  $M$  numărul maxim de arce dintr-o tăietură a rețelei.
- Această proprietate a fost enunțată **nu** atât pentru precizarea limitei numărului de iterări cât mai ales pentru a evidenția complexitatea analizei.
- De fapt această problemă a fost intens studiată și au fost dezvoltăți algoritmi complecși care obțin performanțe mult mai bune decât algoritmul prezentat.
- Cu toate acestea algoritmul lui Edmonds și Karp este foarte greu de depășit în performanță pentru rețelele care apar în mod obișnuit în practică.

- Problema rețelelor de curgere (de transport) poate fi extinsă în diferite moduri și multe din variantele sale au fost studiate în detaliu datorită importanței pe care le prezintă pentru aplicațiile curente.
  - Spre exemplu **problema fluxului de mărfuri multiple** ("**multicommodity flux problem**") presupune introducerea în rețea a mai multor destinații și a mai multor tipuri de materiale care parcurg rețeaua.
- Aceste generalizări fac ca problema fluxului maxim să devină mult mai dificilă și rezolvarea ei presupune algoritmi mult mai complicați.
  - Spre exemplu, până la ora actuală **nu** se cunoaște pentru acest caz o teoremă analoagă teoremei flux maxim – tăietură minimă.
- Alte extensii ale problemei rețelelor de curgere se bazează pe următoarele abordări:
  - Luarea în considerare a restricțiilor de capacitate din noduri - ușor de rezolvat prin introducerea unor arce artificiale care tratează aceste capacitați
  - Utilizarea conductelor cu dublu sens - ușor de rezolvat prin înlocuirea arcului neorientat cu o pereche de arce orientate
  - Introducerea unei limite inferioare a fluxului în arce - dificil de rezolvat
- Dacă se ia în considerare faptul real că conductele presupun în afară de capacitați de transport și costuri atunci problema **costului minim** al unei rețele pentru un flux precizat, devine o problemă complicată de cercetare.

## 12.8. Problema potrivirilor ("Matching")

- O problemă care apare destul de frecvent în practică este problema **perechilor stabili**.
- Ea se referă la asociarea în perechi a unor elemente în conformitate cu preferințele acestora, evitând conflictele.
  - Spre exemplu, în S.U.A. a fost implementat un sistem destul de complicat de plasare a studenților din anii superiori de la universitățile de medicină în spitale pentru efectuarea stagiu lui.
    - Fiecare student prezintă o listă cu spitale ordonate în ordinea preferințelor și fiecare spital prezintă o listă cu studenți de asemenea ordonați în ordinea preferințelor.
    - Problema complicată care trebuie rezolvată este aceea de a asocia fiecărui student un loc într-un spital, într-o manieră corectă respectând toate preferințele formulate.

- Algoritmul care realizează această asociere este foarte sofisticat deoarece este de așteptat ca cei mai buni studenți să fie solicitați de mai multe spitale, iar spitalele cele mai bune să fie solicitate de mai mulți studenți.
- Este însă evident faptul că nu în toate situațiile preferințele reciproce pot fi satisfăcute prin asocieri corespunzătoare și în mod efectiv deși algoritmul găsește soluția optimă în condițiile date, în multe situații apar nemulțumiri.
- **Problema perechilor (relațiilor) stabile** este un caz special al **problemei potrivirilor**, care la rândul ei este o problemă fundamentală a **teoriei grafurilor**, problemă care a fost foarte intens studiată.
- Fiind dat un graf, o **potrivire** este o **submulțime a arcelor grafului**, selectată astfel astfel încât nici unul din nodurile conectate de arcele aparținând acestei submulțimi **nu** pare mai mult decât odată.
  - Aceasta înseamnă că nodurile conectate de aceste arce sunt asociate în **perechi**.
  - **Nu** este necesar să fie implicate toate nodurile.
  - Chiar dacă se încearcă ca potrivirea să acopere cât mai multe noduri posibile, diferitele posibilități de asociere conduc de regulă la numere diferite de noduri neutilizate.
- De un interes particular se bucură **potrivirea maximă** care conține numărul maxim posibil de arce respectiv numărul minim de noduri neutilizate.
  - Cazul ideal este acela al grafului cu  $2n$  noduri și  $n$  arce care conectează noduri care apar o singură dată.
- O **potrivire completă** este potrivirea în care fiecare nod al grafului este conectat prin arce aparținând potrivirii.
  - Este evident că orice potrivire completă este în același timp și maximă [AH 85].
- În figura 12.8.a se prezintă o potrivire maximă pentru graful reprezentat (mulțimea arcelor îngroșate).



**Fig.12.8.a.** Potrivire maximă pentru un graf

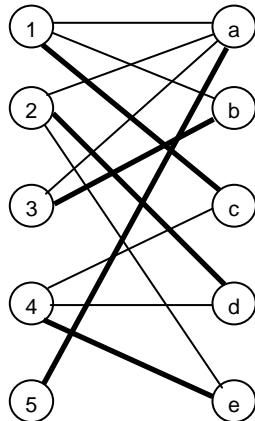
- Determinarea unei astfel de potriviri este însă o problemă dificilă, chiar în cazul grafului simplu din figură.

- Spre exemplu, o metodă de determinare a potrivirilor ar putea consta în selecția arcelor eligibile în ordinea în care acestea apar la traversarea grafului prin **tehnica căutării în adâncime**.
- În cazul grafului din figură această metodă conduce la selecția arcelor AF, EG, HI, JK și LM care reprezintă o potrivire dar **nu** cea maximă.
- Problema repartizării studenților la spitale pentru satisfacerea stagiului, descrisă anterior, poate fi modelată cu ajutorul unui graf în care studenții și spitalele corespund nodurilor grafului iar arcele reprezintă preferințele.
- Există posibilitatea de a atribui **ponderi** preferințelor (de exemplu valori cuprinse între 1 și 10), situație care prefigurează **problema potrivirilor ponderate**.
- **Problema potrivirilor ponderate** se poate formula astfel.
  - Dându-se un graf ponderat, problema potrivirilor ponderate constă în determinarea unei mulțimi de arce, astfel încât nici un nod nu apare mai mult decât o singură dată și suma ponderilor arcelor este maximă [SE 88].
  - Există și o altă variantă a acestei probleme în care se ține cont de **ordinea** în care sunt formulate preferințele, fără a mai fi necesară acordarea de ponderi.
- Problema potrivirilor suscită un interes sporit din partea matematicienilor și informaticienilor din cauza naturii sale intuitive și a largii aplicabilități.
- Soluționarea ei în cazul general care presupune utilizarea într-o manieră complexă a matematicii combinatoriale depășește cadrul prezentului volum.
- O variantă cunoscută a problemei potrivirilor ponderate o **reprezintă problema relațiilor stabile** (“**stable marriage problem**”), formulată de regulă în cadrul extrem de delicat al selecției partenerilor de viață.
  - În volumul 1 al cărții de față, § 4.5 s-a prezentat o modalitate de rezolvare a unei variante a acestei probleme, rezolvare care în contextul unei abordări recursive bazată pe tehnica backtracking, determină toate soluțiile posibile ale problemei.
  - Inconvenientele sunt legate de eficiența relativ scăzută datorată utilizării recursivității și contextul rigid al formulării problemei.
- În cadrul subcapitolului de față, vor fi dezvoltate alte două metode eficiente de determinare a **potrivirii maxime** în contextul simplificat al **grafurilor bipartite**.

### 12.8.1. Grafuri bipartite

- Așa cum s-a precizat, problema potrivirilor este o problemă reprezentativă pentru foarte multe situații care apar în practică.
- Alte situații în afara celor deja prezentate, specifice acestei probleme, sunt:
  - Repartizarea şomerilor pe posturile disponibile,

- Alcătuirea orarului prin repartizarea cursurilor pozițiilor orarului,
- Repartizarea membrilor parlamentului pe comisii, etc.
- Grafurile care modelează astfel de cazuri și numesc **grafuri bipartite**.
- După cum s-a mai precizat un **graf bipartit** este acel graf ale căruia noduri pot fi divizate în două mulțimi disjuncte și ale cărui arce conectează exclusiv noduri aparținând unui, uneia dintre mulțimi iar celălalt celei de-a doua mulțimi (Capitolul 10, &10.1).
  - În figura 12.8.1.a apare un exemplu de graf bipartit.

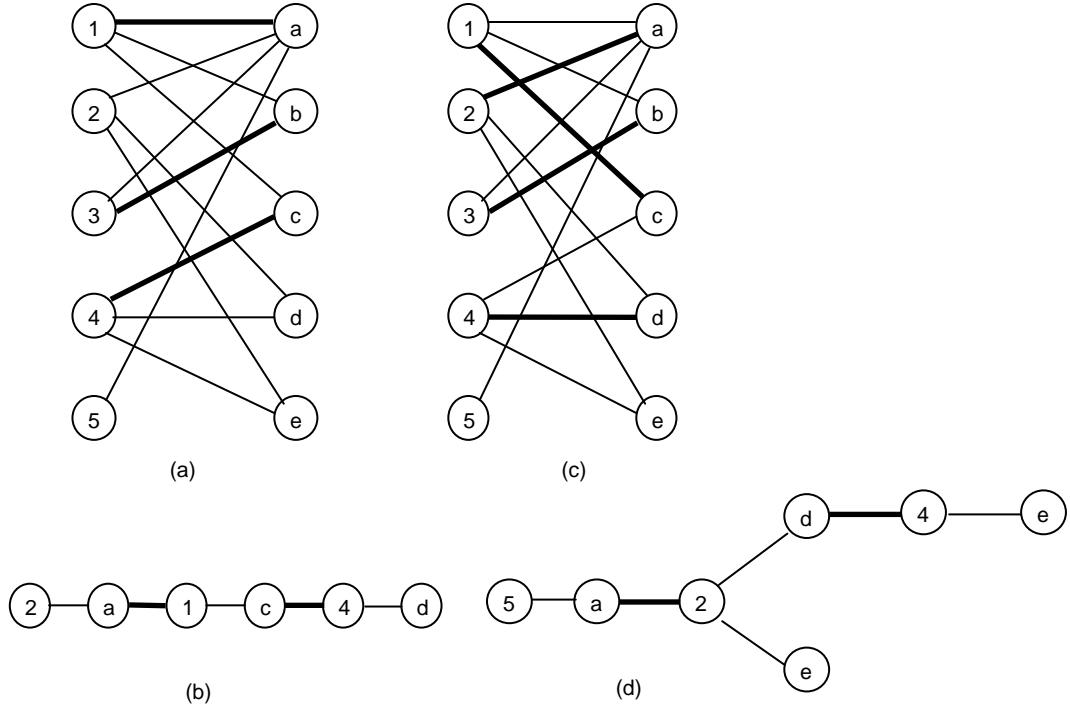


**Fig.12.8.1.a.** Graf bipartit și o potrivire maximă a sa

- În ceea ce privește reprezentarea grafurilor bipartite rămân valabile modalitățile de reprezentare clasice definite în cazul grafurilor normale, adică matricile de adiacențe respectiv structurile de adiacențe.
- Revenind la exemplul din figura 12.8.1.a se observă că s-au notat cu litere mici elementele unei mulțimi de noduri și cu cifre elementele celeilalte.
- În acest context, **problema potrivirii maxime** pentru grafurile bipartite poate fi formulată astfel.
  - Se cere să se determine cea mai mare submulțime a mulțimii perechilor literă-cifră care se bucură de proprietatea că oricare două perechi ale submulțimii **nu** conțin aceeași literă sau aceeași cifră.
  - Rezolvarea eficientă a problemei potrivirii maxime este o sarcină dificilă.
  - O metodă directă de rezolvare poate fi următoarea: se generează toate potrivirile posibile și se selectează cea care conține cel mai mare număr de arce.
    - Dezavantajul acestei metode este acela că timpul ei de execuție este o funcție exponențială în raport cu numărul de arce.
  - În continuare se prezintă două metode mai eficiente care rezolvă această problemă.

## 12.8.2. Determinarea potrivirii maxime prin tehnica drumurilor augmentate

- Unul din algoritmii eficienți de determinare a potrivirii maxime utilizează tehnica **drumurilor augmentate** (“**augmenting path**”).
- Fie  $P$  o potrivire în graful  $G$ .
- Un nod  $x$  este “**potrivit**”, adică constituie încă o pereche, dacă este extremitatea unui arc aparținând lui  $P$ .
- Un drum care conectează două noduri care nu au fost încă “potrivite” și în care arcele alternative aparțin lui  $P$ , se numește **drum augmentat** relativ la  $P$ .
- Se observă că un **drum augmentat** trebuie să fie de lungime impară și trebuie să înceapă și să se termine cu arce care nu aparțin lui  $P$ .
- Se subliniază de asemenea faptul că fiind dat un drum argumentat  $D$  se poate găsi întotdeauna o potrivire mai mare în graful  $G$ , extrăgând din  $P$  acele arce care sunt în  $D$  și apoi adăugând lui  $P$ , arcele din  $D$  care inițial nu au fost în  $P$ .
- Această nouă potrivire este  $P \Delta D$ , unde semnul  $\Delta$  reprezintă operatorul “sau exclusiv” aplicat mulțimilor.
- De fapt noua potrivire conține acele arce din care sunt ori în  $P$  ori în  $D$  dar nu în amândouă.
- În figura 12.8.2.a (a) apare **un graf bipartit** și o **potrivire**  $P$  constând din arcele îngroșate  $(1, a), (3, b)$  și  $(4, c)$ .
  - Drumul  $2, a, 1, c, 4, d$  din fig. 12.8.2.a (b) este un **drum augmentat** relativ la  $P$ .
  - În aceeași figură desenul (c) materializează potrivirea mai mare  $(1, c), (2, a), (3, b), (4, d)$  obținută prin extragerea din  $P$  a arcelor ce aparțin drumului augmentat  $(a, 1), (c, 4)$  și apoi adăugând lui  $P$  celelalte arce ale drumului  $(2, a), (1, c)$  și  $(4, d)$ .



**Fig.12.8.2.a.** Potriviri și drumuri augmentate într-un graf bipartit

- O observație extrem de importantă este următoarea:  $P$  este **potrivirea maximă**, dacă și numai dacă, **nu** mai există nici un drum augmentat relativ la  $P$ .
  - Se presupune că  $P$  și  $Q$  sunt două potriviri ale lui  $G$
  - Se presupune de asemenea că  $|P| < |Q|$  ( $|P|$  precizează numărul de arce din  $P$ ).
  - Pentru a vedea dacă  $P \Delta Q$  conține un drum augmentat relativ la  $P$ , se consideră graful  $G' = (N, P \Delta Q)$ .
  - Deoarece atât  $P$  cât și  $Q$  sunt ambele potriviri în  $G$ , fiecare nod aparținând lui  $N$  este o extremitate pentru cel mult un arc al lui  $P$  și o extremitate pentru cel mult un arc al lui  $Q$ .
  - Astfel, fiecare **componentă conexă** a lui  $G'$  formează un drum simplu (posibil un ciclu) care conține arce **alternând** între  $P$  și  $Q$ .
  - Fiecare drum care nu este un ciclu, este fie un drum augmentat relativ la  $P$  fie un drum augmentat relativ la  $Q$ , după cum conține mai multe arce din  $P$  respectiv din  $Q$ .
  - Fiecare ciclu are un număr egal de arce din  $P$  și  $Q$ .
  - Deoarece  $|P| < |Q|$ ,  $P \Delta Q$  are mai multe arce din  $Q$  decât din  $P$  și astfel conține cel puțin un **drum augmentat** relativ la  $P$  [AH 85].
- În continuare, se poate formula algoritmul care determină potrivirea maximă  $P$  pentru un

graf  $G = (N, A)$ .

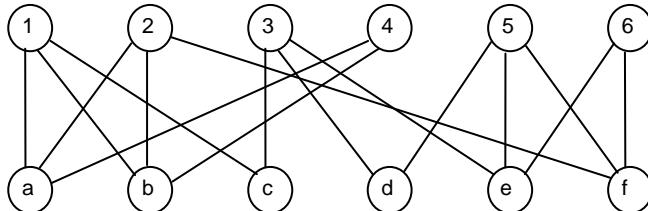
- (1) Se ponește cu  $P = \emptyset$ ;
  - (2) Se determină un drum augmentat  $D$  relativ la  $P$  și se înlocuiește  $P$  cu  $P \cup D$ ;
  - (3) Se repetă pasul (2) până când nu mai există nici un drum augmentat.
  - (4) În acest moment  $P$  reprezintă potrivirea maximă.
- Mai rămâne de precizat modul în care poate fi determinat un **drum augmentat** relativ la o potrivire  $P$ .
  - Acest lucru se va realiza în contextul unui **graf bipartit**  $G$ , ale cărui noduri sunt partaționate în două mulțimi  $N_1$  și  $N_2$ .
  - În continuare, utilizând o procedură similară “căutării prin cuprindere”, se construiește un graf al drumului augmentat pentru  $G$  relativ la potrivirea  $P$ , parcurgând succesiv nivelurile  $i = 0, 1, 2, \dots, k$ .
  - Nivelul 0 constă din toate nodurile “nepotrivite” ale lui  $N_1$ .
    - La **nivelul impar**  $i$  se adaugă noile noduri care sunt adiacente nodurilor nivelului  $i - 1$  prin arce care **nu aparțin potrivirii** și care vor fi adăugate lui  $P$ .
    - La **nivelul par**  $i$  se adaugă noile noduri care sunt adiacente nivelului  $i - 1$  prin arce care **aparțin potrivirii**  $P$ .
  - Se continuă construcția grafului drumului augmentat nivel cu nivel până când un nod “nepotrivit” este adăugat unui nivel impar sau până când nu mai poate fi adăugat nici un nod.
    - Dacă există un drum augmentat relativ la  $P$ , un nod “nepotrivit”  $x$  va fi adăugat în consecință unui nivel impar.
    - Drumul de la  $x$  la oricare nod al nivelului 0 este un **drum augmentat** relativ la  $P$ .
  - În figura 12.8.2.a (d) se prezintă un drum augmentat pentru graful (a), relativ la potrivirea (c).
    - În acest context  $N_1 = \{1, 2, 3, 4, 5\}$  iar  $N_2 = \{a, b, c, d, e\}$ .
    - La momentul considerat, la nivelul 0, mulțimea nodurilor “nepotrivite” ale lui  $N_1$  conține un singur element, nodul 5, cu care se pornește construcția grafului drumului augmentat.
    - La nivelul 1 se adaugă arcul care **nu aparține** potrivirii  $(5, a)$ .

- La nivelul 2 se adaugă arcul  $(a, 2)$  care **apartine** potrivirii.
- La nivelul 3 se poate adăuga fie arcul  $(2, d)$  fie arcul  $(2, e)$  nici unul dintre ele neapărținând potrivirii.
- Dacă se adaugă arcul  $(2, d)$ , se poate adăuga în continuare arcul  $(d, 4)$  la nivelul 4 (apartine potrivirii), după care se poate adăuga arcul  $(4, e)$  la nivelul 5 (nu aparține potrivirii)
- Deoarece nodul  $e$  este “nepotrivit” din punctul de vedere al grafului drumului augmentat, construcția acestui graf se termină în ambele cazuri după adăugarea nodului  $e$ .
- După cum se observă s-au construit două drumuri augmentate după cum se remarcă în figura 12.8.2.a (d)
- Ambele drumuri  $e, 4, d, 2, a, 5$  și  $e, 2, a, 5$  sunt **drumuri augmentate** relativ la potrivirea din fig. 12.8.2.a (c).
- Spre exemplu dacă se utilizează primul drum și se extrag din potrivirea  $P$  reprezentată la (c) arcele arcele din drum care aparțin lui  $P(4, d)$  și  $(2, d)$ , respectiv se adaugă arcele din drumul augmentat care nu aparțin lui  $P(e, 4), (d, 2), (a, 5)$  se obține potrivirea maximă din figura 12.8.1.a.
- Se presupune că  $G$  are  $n$  noduri și  $a$  arce.
- Construcția grafului drumului augmentat pentru o potrivire dată, necesită  $O(a)$  pași de execuție dacă se utilizează reprezentarea bazată pe structuri de adiacențe.
  - Deci fiecare drum augmentat nou necesită un efort de calcul proporțional cu  $O(a)$ .
- Pentru determinarea potrivirii maxime este necesară determinarea a cel mult  $n/2$  drumuri augmentate, deoarece fiecare dintre ele mărește potrivirea curentă cu cel puțin un arc.
  - În consecință, pentru grafuri bipartite, potrivirea maximă poate fi determinată în  $O(n*a)$  unități de timp de execuție.

### 12.8.3. Determinarea potrivirii maxime prin metoda căutării în rețele de curgere

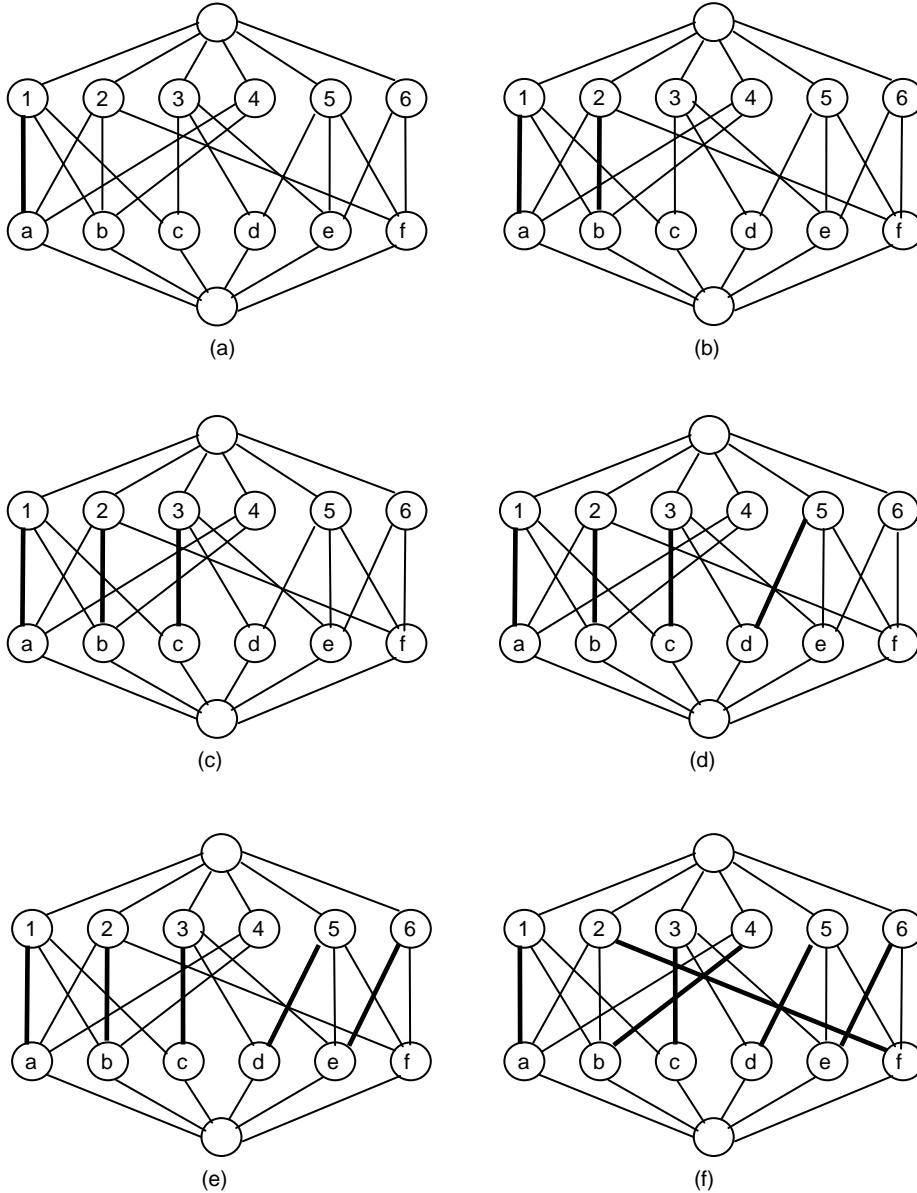
- O altă soluție eficientă pentru determinarea potrivirii maxime poate fi formulată după cum urmează.
- Pentru a rezolva o instanță particulară a problemei potrivirilor, se va construi o instanță corespunzătoare a **problemei rețelelor de curgere** a cărei soluție va fi determinată utilizând metoda prezentată în secțiunea anterioară.
- Utilizând această soluție, în continuare se poate rezolva și problema potrivirilor în același context.

- De fapt în această manieră, problema potrivirilor se reduce la problema rețelelor de curgere.
- Construcția rețelei este directă.
  - (1) Dându-se o instanță a unei potriviri pentru un graf bipartit, se construiește o instanță a **rețelei de curgere** adăugând un **nod sursă** din care pornesc arce spre toate nodurile uneia dintre mulțimile de noduri ale grafului bipartit.
  - (2) Se adaugă în continuare toate arcele grafului bipartit care conectează nodurile primei mulțimi cu nodurile aparținând celei de-a doua.
  - (3) La sfârșit se adaugă un nod destinație și un set de arce care conectează nodurile celei de-a doua mulțimi a grafului bipartit cu acest nod.
  - (4) Toate arcele care apar în rețea se consideră de capacitate unu.
- În figura 12.8.3.b se prezintă modul de construcție al rețelei de curgere pentru graful bipartit din fig. 12.8.3.a împreună cu pașii rezolvării problemei rețelei și implicit a determinării potrivirii maxime.



**Fig.2.8.3.a.** Graf bipartit

- Din reprezentare rezultă în mod evident partajarea nodurilor grafului bipartit, direcția fluxurilor precum și faptul că deoarece toate capacitatele sunt 1, fiecare drum prin rețea corespunde unui arc al potrivirii.
- Astfel în exemplul din figură, desenele (a), (b), (c) și (d) materializează primii patru pași care conduc la determinarea arcelor corepunzătoare unei potriviri parțiale: a1, b2, c3 și d5.



**Fig.12.8.3.b.** Utilizarea rețelelor de curgere în determinarea potrivirii maxime pentru un graf bipartit

- Pentru determinarea acestor drumuri se utilizează secvența [12.7.3.b].
- Fiecare apel al procedurii CautPrioritar determină fie găsirea unui drum care crește fluxul total cu o unitate fie termină căutarea.
- În pasul (e) toate drumurile directe prin rețea au fost selectate și algoritmul trebuie să utilizeze arce de tip "înapoi".
- Drumul găsit în acest pas este 4 , b , 2 , f .
  - Acest drum, în mod evident determină creșterea fluxului în rețea după cum s-a văzut în secțiunea anterioară.
- În contextul de față, un drum poate fi conceput ca și un set de pași prin care se crează o nouă potrivire parțială (utilizând unul sau mai multe arce) pornind de la potrivirea curentă.

- În situația reprezentată în desen (e), această construcție decurge în mod natural din traversarea arcelor din rețea în ordinea:
  - (1) “4b” ceea ce presupune adăugarea arcului (b , 4 ) potrivirii,
  - (2) “b2” ceea ce înseamnă extragerea arcului (b , 2 )
  - (3) “2f” ceea ce presupune adăugarea arcului ( f , 2 ) potrivirii.
- Astfel, după ce drumul e construit, se ajunge la potrivirea a1 , b4 , c3 , d5 , e6 , f2 .
- În mod echivalent fluxul prin rețea este produs de conductele pline care conectează nodurile precizate și este maxim în situația dată.
- Demonstrația faptului că potrivirea conține exact arcele care umplute la capacitate determină fluxul maxim prin rețea este imediată.
  - (1) În primul rând, rețeaua de curgere conduce întotdeauna la o potrivire legală element care decurge din următorul raționament:
    - Întrucât fiecare nod al rețelei are un arc de capacitate unu care îl traversează (intră sau ieșe din el), rezultă că prin fiecare nod poate trece o singură unitate de flux ceea ce implică faptul că fiecare nod va fi inclus o singură dată în potrivire.
    - (2) În al doilea rând, nici o potrivire nu poate avea mai multe arce, deoarece o astfel de potrivire ar conduce la un flux superior fluxului maxim determinat de algoritm.
- Astfel, pentru a determina potrivirea maximă pentru un graf bipartit, se transformă graful într-o rețea de curgere potrivită utilizării algoritmului dezvoltat în secțiunea precedentă.
- Este evident faptul că rețeaua care rezultă în acest caz este mai simplă decât în cazul grafurilor oarecare pentru care a fost conceput algoritmul.
  - Din acest motiv, algoritmul dezvoltat este oarecum mai eficient în acest caz particular.
- În consecință, luând în considerare performanțele algoritmului de căutare în rețele, potrivirea maximă pentru un graf bipartit poate fi determinată în:
  - $O(n^3)$  pași în cazul grafurilor dense reprezentate prin matrice de adiacențe
  - $O(n(n+a) \log n)$  pași în cazul grafurilor rare reprezentate prin structuri de adiacențe.
- Demonstrația este imediată:
  - Construcția preconizată asigură faptul că fiecare apel al procedurii CautăPrioritar adaugă cel puțin un arc potrivirii deci vor fi necesare cel mult  $n/2$  apeluri ale procedurii pe parcursul execuției algoritmului.

- Astfel timpul total necesitat de execuția algoritmului este proporțional cu produsul dintre factorul  $n$  și performanța algoritmului de căutare în rețele utilizat.