

```

cin>>c->nr;
c->ad=0;
c->as=u;
if (u) u->ad=c;
else p=c;
u=c;
}

void pop(nod *p)
{
    if (!p) cout<<"coada este vida";
    else {
        cout<<p->nr;
        c=p; p=p->ad;
        p->as=0;
        delete c;
    }
}
main(){
    push(u);
    push(u);
    pop(p);
    pop(p);
    pop(p);
}

```

## GRAFURI NEORIENTATE

### 2.1. ASPECTE TEORETICE

#### 2.1.1. Noțiunea de graf neorientat

**Definiție.** Se numește **graf neorientat** o pereche ordonată de mulțimi notată

$G=(V, M)$  unde:

$V$  : este o mulțime **finită și nevidă**, ale cărei elemente se numesc **noduri** sau **vârfuri**;

$M$  : este o mulțime, de **perechi neordonate de elemente distincte din  $V$** , ale cărei elemente se numesc **muchii**.

♦ Exemplu de graf neorientat:

$G=(V, M)$  unde:  $V=\{1,2,3,4\}$   
 $M=\{\{1,2\}, \{2,3\}, \{1,4\}\}$

Demonstratie:

Perechea  $G$  este graf neorientat deoarece respectă definiția prezentată mai sus, adică:

$V$  este finită și nevidă;

$M$  este o mulțime de perechi neordonate (submulțimi cu două elemente) de elemente din  $V$ .

În continuare, vom nota submulțimea  $\{x,y\}$ , care reprezintă o muchie, cu  $[x,y]$  (într-un graf neorientat muchia  $[x,y]$  este aceeași cu muchia  $[y,x]$ ). În baza celor spuse anterior, graful prezentat în exemplul de mai sus se reprezintă textual astfel:

$G=(V, M)$  unde:  $V=\{1,2,3,4\}$   
 $M=\{[1,2], [2,3], [1,4]\}$

În teoria grafurilor neorientate, se întâlnesc frecvent noțiunile:

– **extremitățile unei muchii**

• fiind dată muchia  $m=[x,y]$ , se numesc extremități ale sale **nodurile  $x$  și  $y$** ;

– **vârfuri adiacente**

• dacă într-un graf există muchia  $m=[x,y]$ , se spune despre nodurile  $x$  și  $y$  că sunt **adiacente**;

– **incidentă**

• dacă  $m_1$  și  $m_2$  sunt două muchii ale aceluiași graf, se numește **incidente** dacă au **o extremitate comună**.

Exemplu:  $m_1=[x,y]$  și  $m_2=[y,z]$  sunt incidente

- dacă  $m=[x,y]$  este o muchie într-un graf, se spune despre ea și nodul  $x$ , sau nodul  $y$ , că sunt incidente.

Reprezentarea unui graf neorientat admite două forme, și anume:

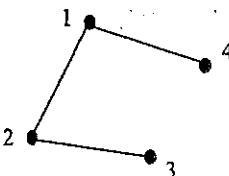
- reprezentare textuală: așa cum s-a reprezentat graful din exemplul anterior;
- reprezentare grafică: muchiile sunt reprezentate prin linii, iar nodurile prin puncte.

♦ Exemplu de graf neorientat reprezentat textual:

$$G=(V, M) \text{ unde: } V=\{1,2,3,4\}$$

$$M=\{[1,2], [2,3], [1,4]\}$$

♦ Exemplu de graf neorientat reprezentat grafic (este graful de la exemplul anterior):



## 2.1.2. Notiunea de graf parțial

**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Se numește **graf parțial**, al grafului  $G$ , **graful neorientat**  $G_i=(V_i, M_i)$  unde  $M_i \subseteq M$ .

♦ Atenție! Citind cu atenție definiția de mai sus, tragem concluzia:

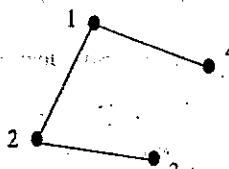
Un **graf parțial** al unui graf neorientat  $G=(V, M)$  are aceeași multime de vârfuri ca și  $G$  iar multimea muchiilor este o submultime a lui  $M$  sau chiar  $M$ .

♦ Exemplu: Fie graful neorientat:

$$G=(V, M) \text{ unde: } V=\{1,2,3,4\}$$

$$M=\{[1,2], [1,4], [2,3]\}$$

reprezentat grafic astfel:



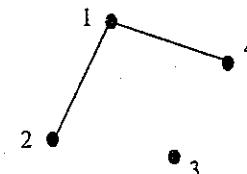
## GRAFURI NEORIENTATE

1. Un exemplu de **graf parțial** al grafului  $G$  este graful neorientat:

$$G_i=(V, M_i) \text{ unde: } V=\{1,2,3,4\}$$

$$M_i=\{\} \quad (\text{s-a eliminat muchia } [2,3])$$

reprezentat grafic astfel:

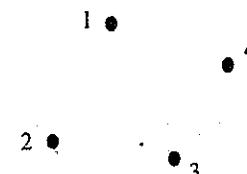


2. Un exemplu de **graf parțial** al grafului  $G$  este graful neorientat:

$$G_i=(V, M_i) \text{ unde: } V=\{1,2,3,4\}$$

$$M_i=\emptyset \quad (\text{s-au eliminat toate muchiile})$$

reprezentat grafic astfel:



**Observație.** Fie  $G=(V, M)$  un graf neorientat. Un **graf parțial**, al grafului  $G$ , se obține păstrând vârfurile și eliminând eventual niște muchii (se pot elimina și toate muchiile, sau chiar nici una).

## 2.1.3. Notiunea de subgraf

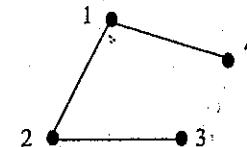
**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Se numește **subgraf** al grafului  $G$ , graful neorientat  $G_i=(V_i, M_i)$  unde  $V_i \subseteq V$  iar  $M_i$  conține toate muchiile din  $M$  care au extremități în  $V_i$ .

♦ Exemplu: Fie graful neorientat:

$$G=(V, M) \text{ unde: } V=\{1,2,3,4\}$$

$$M=\{[1,2], [2,3], [1,4]\}$$

reprezentat grafic astfel:

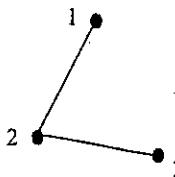


1. Un exemplu de subgraf al grafului G este graful neorientat:

$$G_1 = (V_1, M_1) \text{ unde: } V_1 = \{1, 2, 3\} \quad \{s-a \text{ sters nodul } 4\}$$

$$M_1 = \{[1, 2], [2, 3]\} \quad \{s-a \text{ eliminat muchia } [1, 4]\}$$

reprezentat grafic astfel:

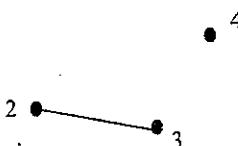


2. Un exemplu de subgraf al grafului G este graful neorientat:

$$G_1 = (V_1, M_1) \text{ unde: } V_1 = \{2, 3, 4\} \quad \{s-a \text{ eliminat nodul } 1\}$$

$$M_1 = \{[2, 3]\} \quad \{s-au \text{ eliminat muchiile } [1, 4], [1, 2]\}$$

reprezentat grafic astfel:



**Observație.** Fie  $G=(V, M)$  un graf neorientat. Un subgraf, al grafului G, se obține ștergând anumite vârfuri și odată cu acestea și muchiile care le admit ca extremitate.

## 2.1.4. Gradul unui vârf

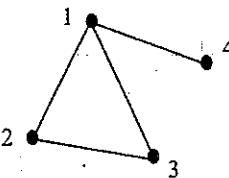
**Definiție.** Fie  $G=(V, M)$  un graf neorientat și x un nod al său. Se numește grad al nodului x, numărul muchiilor incidente cu x, notat  $d(x)$ .

♦ **Exemplu:** Fie graful neorientat:

$$G=(V, M) \text{ unde: } V=\{1, 2, 3, 4\}$$

$$M=\{[1, 2], [2, 3], [1, 4], [1, 3]\}$$

reprezentat grafic astfel:



Gradul nodului 1 este  $d(1)$  și  $d(1)=3$  (în graf sunt trei muchii incidente cu 1)

Gradul nodului 2 este  $d(2)$  și  $d(2)=2$  (în graf sunt două muchii incidente cu 2)

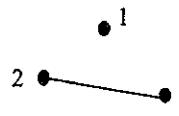
Gradul nodului 3 este  $d(3)$  și  $d(3)=2$  (în graf sunt două muchii incidente cu 3)

Gradul nodului 4 este  $d(4)$  și  $d(4)=1$  (în graf este o singură muchie incidentă cu 4)

**Observații.** 1. Dacă gradul unui vârf este 0, vârful respectiv se numește vârf izolat.

2. Dacă gradul unui vârf este 1, vârful respectiv se numește vârf terminal.

În graful care admite reprezentarea grafică:



deoarece  $d(1)=0$ , vârful 1 se numește vârf izolat, și

deoarece  $d(2)=d(3)=1$ , vârfurile 2 și 3 se numesc vârfuri terminale.

**Propoziție.** În graful neorientat  $G=(V, M)$ , în care  $V=\{x_1, x_2, \dots, x_n\}$  și sunt m muchii,

$$\text{se verifică egalitatea: } \sum_{i=1}^n d(x_i) = 2m$$

**Demonstratie:**  $\sum_{i=1}^n d(x_i) = 2m$

Muchia  $[x, y]$  contribuie cu o unitate la gradul lui x și cu o unitate la gradul lui y, deci, cu două unități la suma din enunț. Cum în total sunt m muchii, rezultă că suma gradelor este  $2m$ .

Având în vedere faptul că suma gradelor vârfurilor dintr-un graf este un număr par ( $2m$ ), a apărut corolarul prezentat mai jos.

**Corolar.** În orice graf neorientat,  $G=(V, M)$ , există un număr par de vârfuri de grad impar.

**Demonstratie:**

Demostrarea ține cont de propoziția de mai sus, adică de faptul că într-un graf neorientat suma S a tuturor gradelor nodurilor este un număr par (dublul numărului muchiilor).

Fie  $S_1$  suma gradelor vârfurilor de grad par (este un număr par, ca sumă de numere pare) și  $S_2$  suma gradelor vârfurilor de grad impar.

Cum  $S=S_1+S_2$ , rezultă că  $S_2=S-S_1$ , deci un număr par (ca diferență de numere pare).

## 2.1.5. Graf complet

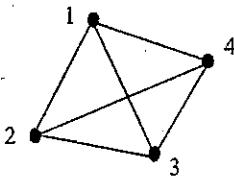
**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Graful G se numește graf complet, dacă oricare două vârfuri distincte ale sale sunt adiacente.

♦ Exemplu de graf neorientat complet:

$$G = (V, M) \text{ unde: } V = \{1, 2, 3, 4\}$$

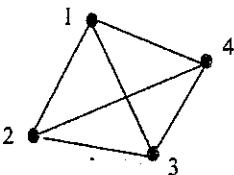
$$M = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}\}$$

Reprezentarea sa grafică este:



- Observații.**
1. Într-un graf complet cu  $n$  vârfuri gradul fiecărui vârf este  $n-1$ , deoarece fiecare vârf este legăt prin muchii de toate celelalte vârfuri.
  2. Graful complet cu  $n$  vârfuri se notează cu  $K_n$ .

În particular, graful:



se notează  $K_4$  (este un graf complet cu 4 vârfuri).

**Propoziție.** Într-un graf complet cu  $n$  vârfuri, notat  $K_n$ , există  $\frac{n(n-1)}{2}$  muchii.

**Demonstrație:**

Din fiecare vârf  $x_i$  pleacă  $n-1$  muchii, deci  $d(x_i) = n-1$ , pentru orice  $i = 1..n$ .

Cum, folosind propoziția prezentată în secțiunea gradul unui vârf,

$$2m = d(x_1) + d(x_2) + \dots + d(x_n) \Rightarrow 2m = (n-1) + (n-1) + \dots + (n-1)$$

$$\Rightarrow 2m = n(n-1) \Rightarrow m = \frac{n(n-1)}{2}$$

## 2.1.6. Graf bipartit

**Definiție.** Fie  $G = (V, M)$  un graf neorientat. Graful  $G$  se numește **graf bipartit**, dacă există două multimi nevide  $V_1$  și  $V_2$  cu proprietățile:

$$- V_1 \cup V_2 = V$$

$$- V_1 \cap V_2 = \emptyset$$

orice muchie a lui  $G$  are o extremitate în  $V_1$  și pe cealaltă în  $V_2$ .

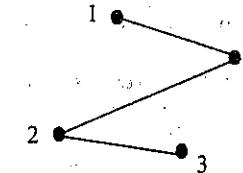
## GRAFURI NEORIENTATE

♦ Exemplu de graf neorientat bipartit:

$$G = (V, M) \text{ unde: } V = \{1, 2, 3, 4\}$$

$$M = \{\{1, 3\}, \{2, 3\}, \{2, 4\}\}$$

Reprezentarea sa grafică este:



**Demonstrație:**

Graful de mai sus este bipartit deoarece respectă întocmai definiția grafului bipartit, adică există două multimi  $V_1 = \{1, 2\}$  și  $V_2 = \{3, 4\}$  astfel încât:

$$- V_1 \cup V_2 = V$$

$$- V_1 \cap V_2 = \emptyset$$

orice muchie din  $G$  are o extremitate în  $V_1$  și pe cealaltă în  $V_2$ .

**Observație.** A demonstra că un graf nu este bipartit înseamnă a demonstra că nu pot fi construite cele două multimi de care se vorbește în definiția grafului bipartit.

## 2.1.7. Graf bipartit complet

**Definiție.** Fie  $G = (V, M)$  un graf bipartit. Graful  $G$  se numește **graf bipartit complet** dacă:

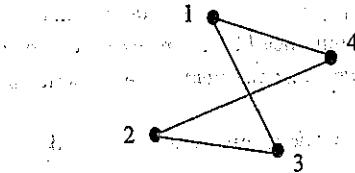
pentru orice  $x$  din  $V_1$  și orice  $y$  din  $V_2$  există în  $G$  muchia  $[x, y]$ .

♦ Exemplu de graf neorientat bipartit complet:

$$G = (V, M) \text{ unde: } V = \{1, 2, 3, 4\}$$

$$M = \{\{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}\}$$

Reprezentarea sa grafică este:



**Demonstrație:**

Graful de mai sus este, mai întâi, bipartit deoarece respectă întocmai definiția grafului bipartit, adică există două multimi  $V_1 = \{1, 2\}$  și  $V_2 = \{3, 4\}$  astfel încât:

$$- V_1 \cup V_2 = V$$

- $V_1 \cap V_2 = \emptyset$
- orice muchie din  $G$  are o extremitate în  $V_1$  și pe celalătă în  $V_2$ .

Cum, în plus, pentru orice  $x$  din  $V_1$  și orice  $y$  din  $V_2$  există în  $G$  muchia  $[x,y]$ , rezultă, conform definiției, că graful  $G$  este bipartit complet.

**Observație.** A demonstra că un graf este bipartit complet înseamnă a demonstra:

- că este bipartit;
- pentru orice  $x$  din  $V_1$  și orice  $y$  din  $V_2$  există în  $G$  muchia  $[x,y]$ .

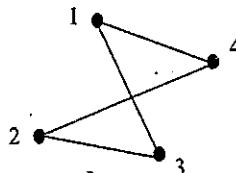
**Observație.** Într-un graf bipartit complet în care  $V_1$  are  $p$  elemente și  $V_2$  are  $q$  elemente există  $pq$  muchii.

**Demonstrație:**

Fiecare vârf  $x_i$  din  $V_1$  este legat de toate vârfurile aflate în  $V_2$ , altfel spus, există  $q$  muchii care-l admit ca extremitate pe  $x_i$ . Cum în  $V_1$  sunt  $p$  elemente, adică  $i=1\dots p$ , înseamnă că elementele multimii  $V_1$  sunt legate prin  $pq$  muchii de elementele multimii  $V_2$ .

**Observație.** Graful bipartit complet în care  $V_1$  are  $p$  elemente și  $V_2$  are  $q$  elemente se notează cu  $K_{p,q}$ .

În particular, graful:



se notează  $K_{2,2}$  (este un graf bipartit complet cu  $|V_1|=2$  și  $|V_2|=2$ ,  $V_1$  și  $V_2$  din definiție).

### 2.1.8. Reprezentarea grafurilor neorientate

Fie  $G=(V, M)$  un graf neorientat, unde  $V=\{x_1, x_2, \dots, x_n\}$  și  $M=\{m_1, m_2, \dots, m_p\}$ .

Deoarece între mulțimea  $\{x_1, x_2, \dots, x_n\}$  și mulțimea  $\{1, 2, \dots, n\}$  există o bijectie,  $x_i \leftrightarrow i$ , putem presupune, fără a restrânge generalitatea, mai ales pentru a ușura scrierea, că  $V=\{1, 2, \dots, n\}$ .

În baza celor spuse mai sus, mai departe, în loc de  $x_i$  vom scrie  $i$  și în loc de muchia  $[x_i, x_j]$  vom scrie  $[i, j]$ .

Pentru a putea prelucra un graf neorientat cu ajutorul unui program, trebuie mai întâi să fie reprezentat în programul respectiv.

Pentru a reprezenta un graf, într-un program, există mai multe modalități folosind diverse structuri de date, dintre acestea, în continuare, vom prezenta:

- reprezentarea unui graf prin matricea de adiacență;
- reprezentarea unui graf prin liste de adiacență;
- reprezentarea unui graf prin sirul muchiilor.

### Matricea de adiacență

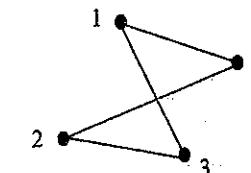
Fie  $G=(V, M)$  un graf neorientat cu  $n$  vârfuri ( $V=\{1, 2, \dots, n\}$ ) și  $m$  muchii.

**Matricea de adiacență**, asociată grafului  $G$ , este o matrice patratică de ordinul  $n$ , cu elementele definite astfel:

$$a_{ij} = \begin{cases} 1, & \text{daca } [i, j] \in M \\ 0, & \text{daca } [i, j] \notin M \end{cases}$$

(altfel spus,  $a_{ij}=1$  dacă există muchie între  $i$  și  $j$  și  $a_{ij}=0$  dacă nu există muchie între  $i$  și  $j$ )

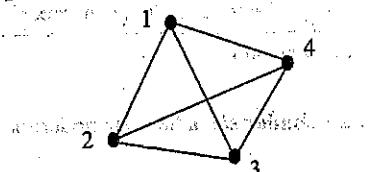
◆ **Exemplul 1.** Fie graful reprezentat grafic ca în figura de mai jos:



Matricea de adiacență, asociată grafului, este:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix}$$

◆ **Exemplul 2.** Fie graful reprezentat grafic ca în figura de mai jos:



Matricea de adiacență, asociată grafului, este:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

◆ Comentarii:

- Matricea de adiacență este o matrice pătratică, de ordin  $n$ , și simetrică față de diagonală principală (adică  $a[i][j]=a[j][i]$ ).

Sevențele de citire a matricei de adiacență, sunt:

```
int a[100][100];
```

```
cout<<"n="; cin>>n;
for (i=1;i<=n-1;i++)
    for (j=i+1;j<=n;j++)
    {
        cin>>a[i][j];
        a[j][i]=a[i][j];
    }
```

sau:

```
cin>>n; cin>>m;
for (i=1;i<=n;i++)
    for (j=1;j<=n;j++)
        a[i][j]=0;
for (i=1;i<=m;i++)
{
    cout<<"dati extremitatile muchiei "<<i; cin>>x>>y;
    a[x][y]=1; a[y][x]=1;
}
```

- Matricea de adiacență are toate elementele de pe diagonală principală egale cu 0.
- Numeărul elementelor egale cu 1 de pe linia  $i$  (sau coloana  $i$ ) este egal cu gradul vârfului  $i$ . Dacă vârful  $i$  este un vârf izolat pe linia  $i$  (și coloana  $i$ ) nu sunt elemente egale cu 1.

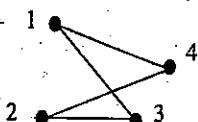
### Liste de adiacență

Fie  $G=(V, M)$  un graf neorientat cu  $n$  vârfuri ( $V=\{1, 2, \dots, n\}$ ) și  $m$  muchii.

Reprezentarea grafului  $G$  prin liste de adiacență constă în:

- precizarea numărului de vârfuri,  $n$ ,
- pentru fiecare vârf  $i$ , se precizează lista  $L_i$  a vecinilor săi, adică lista nodurilor adiacente cu nodul  $i$ .

◆ Exemplul 1. Fie graful reprezentat grafic ca în figura de mai jos:



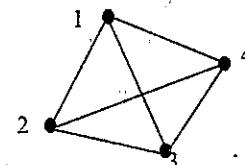
### GRAFURI NEORIENTATE

Reprezentarea sa prin liste de adiacențe presupune:

- precizarea numărului de vârfuri  $n$ ,  $n=4$ ;
- precizarea listei vecinilor lui  $i$ , pentru  $i=1..n$ , astfel:

Vârful $i$	Lista vecinilor lui $i$
1	3, 4
2	3, 4
3	1, 2
4	1, 2

◆ Exemplul 2. Fie graful reprezentat grafic ca în figura de mai jos:



Reprezentarea sa prin liste de adiacențe presupune:

- precizarea numărului de vârfuri  $n$ ,  $n=4$ ;
- precizarea listei vecinilor lui  $i$ , pentru  $i=1..n$ , astfel:

Vârful $i$	Lista vecinilor lui $i$
1	2, 3, 4
2	1, 3, 4
3	1, 2, 4
4	1, 2, 3

◆ Comentarii:

Acest mod de reprezentare se poate implementa astfel:

- Se folosește un tablou bidimensional  $T$ , caracterizat astfel:
  - are  $n+2m$  coloane;
  - $T_{1,i}=i$  pentru  $i=1..n$ ;
  - Pentru  $i=1..n$   $T_{2,i}=k$ , dacă  $T_{1,k}$  este primul nod din lista vecinilor lui  $i$ ;  
 $T_{2,i}=0$ , dacă nodul  $i$  este izolat;
  - Dacă  $T_{2,i}=u$ , adică  $u$  este un nod din lista vecinilor lui  $i$ , atunci  
 $T_{2,j}=0$ , dacă  $u$  este ultimul nod din lista vecinilor lui  $i$ ;  
 $T_{2,j}=j+1$ , dacă  $u$  nu este ultimul nod din lista vecinilor lui  $i$ .

◆ Exemplu de completare a tabloului pentru graful de la exemplul 1.

Prima etapă. Se numerotează coloanele  $(1..n+2m)$  și se trec vâfurile.

1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4								

A doua etapă. Se trec în tabel vecinii lui 1, începând de la coloana 5.

1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	3	4						
5				6	0						

$T_{2,1}=5$ , pentru că primul vecin (3) al lui 1 s-a trecut la coloana 5 ( $T_{1,5}=3$ ).

$T_{2,5}=6$ , pentru că următorul vecin (4) al lui 1 s-a trecut la coloana 6 ( $T_{1,6}=4$ ).

$T_{2,6}=0$ , pentru că vecinul  $T_{1,6}$  (4) al lui 1 este ultimul din listă.

A treia etapă. Se trec în tabel vecinii lui 2, începând de la coloana 7.

1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	3	4	3	4				
5	7			6	0	8	0				

$T_{2,2}=7$ , pentru că primul vecin (3) al lui 2 s-a trecut la coloana 7 ( $T_{1,7}=3$ ).

$T_{2,7}=8$ , pentru că următorul vecin (4) al lui 2 s-a trecut la coloana 8 ( $T_{1,8}=4$ ).

$T_{2,8}=0$ , pentru că vecinul  $T_{1,8}$  (4) al lui 2 este ultimul din listă.

A patra etapă. Se trec în tabel vecinii lui 3, începând de la coloana 9.

1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	3	4	3	4	1	2		
5	7	9		6	0	8	0	10	0		

$T_{2,3}=9$ , pentru că primul vecin (1) al lui 3 s-a trecut la coloana 9 ( $T_{1,9}=1$ ).

$T_{2,9}=10$ , pentru că următorul vecin (2) al lui 3 s-a trecut la coloana 10 ( $T_{1,10}=2$ ).

$T_{2,10}=0$ , pentru că vecinul  $T_{1,10}$  (2) al lui 3 este ultimul din listă.

Ultima etapă. Se trec în tabel vecinii lui 4, începând de la coloana 11.

1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	3	4	3	4	1	2	1	2
5	7	9	11	6	0	8	0	10	0	12	0

$T_{2,4}=11$ , pentru că primul vecin (1) al lui 4 s-a trecut la coloana 11 ( $T_{1,11}=1$ ).

$T_{2,11}=12$ , pentru că următorul vecin (2) al lui 4 s-a trecut la coloana 12 ( $T_{1,12}=2$ ).

$T_{2,12}=0$ , pentru că vecinul  $T_{1,12}$  (2) al lui 4 este ultimul din listă.

## GRAFURI NEORIENTATE

2. Se folosește un tablou unidimensional, cu numele cap, și un tablou bidimensional, cu numele L (care reține listele de vecini pentru fiecare nod), caracterizate astfel:

Tabloul cap:

- are  $n$  componente;
- $cap_i=c$ , dacă primul nod din lista vecinilor lui  $i$  este trecut în tabloul L la coloana  $c$ , adică  $L_{i,c}$  este primul vecin al lui  $i$ ,
- și  $cap_i=0$ , dacă nodul  $i$  este izolat

Tabloul L:

- are  $2m$  componente;
- dacă  $k$  este un vecin al nodului  $i$ , atunci:

$L_{1,p}=k$  și  $L_{2,p}=0$ , dacă  $k$  este ultimul vecin din listă, sau  
 $L_{1,p}=k$  și  $L_{2,p}=p+1$  dacă  $k$  nu este ultimul vecin din listă  
( $p$  este coloana la care s-a ajuns în tabloul L).

♦ Exemplu de completare a tablourilor cap și L, pentru graful de la exemplul 1.

Tabloul cap

1	2	3	4
1	3	5	7

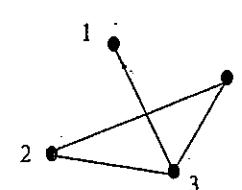
Tabloul L

1	2	3	4	5	6	7	8
3	4	3	4	1	2	1	2
2	0	4	0	6	0	8	0

3. Se folosește un tablou bidimensional, cu numele L, caracterizat astfel:

- are  $n$  linii;
- pe linia  $i$  se trec vecinii nodului  $i$ .

♦ Exemplu de completare a tabloului L, pentru graful:



**Tabloul L**

3		
3	4	
1	2	4
2	3	

♦ Implementarea în limbajul C/C++, a ideii prezentate mai sus, se realizează conform secvenței de program prezentată mai jos.

```
int L[20][20];
int nr_vec[20];

cout<<"n=";
cin>>n;
for (i=1;i<=n;i++)
{
    cout<<"Dati numărul vecinilor nodului "<<i; cin>>nr_vec[i];
    for (j=1;j<=nr_vec[i];j++)
        cin>>L[i][j];
}
```

♦ Construirea matricei de adiacență când se cunoaște L (listele vecinilor fiecărui nod).

```
for (i=1;i<=n;i++)
{
    for (j=1;j<=nr_vec[i];j++)
        a[i][L[i][j]]=1;
}
```

♦ Construirea tabloului L (listele vecinilor nodurilor) când se cunoaște matricea de adiacență.

```
for (i=1;i<=n;i++)
{
    k=0;
    for (j=1;j<=n;j++)
        if (a[i][j]==1) {
            k=k+1;
            L[i][k]=j;
        }
}
```

4. Se folosește un tablou unidimensional, cu numele L, caracterizat astfel:

– componente sale sunt de tip referință;

– are n componente;

– L<sub>i</sub> pointează spre începutul listei vecinilor nodului i.

**Șirul muchiilor**

Fie G=(V, M) un graf neorientat cu n vârfuri (V={1,2, ..., n}) și m muchii.

Reprezentarea grafului G constă în precizarea numărului n de noduri și numărului m de muchii, precum și în precizarea extremităților pentru fiecare muchie în parte.

**Comentarii:**

Acest mod de reprezentare se implementează astfel:

1. Se dă numărul n de noduri, numărul m de muchii și extremitățile fiecărei muchii, care sunt trecute în vectorii e1 și e2 astfel:

extremitățile primei muchii sunt e1[1] și e2[1];

extremitățile celei de-a două muchii sunt e1[2] și e2[2];

deci M={[e1[1],e2[1]], [e1[2],e2[2]], ..., [e1[m],e2[m]]}.

♦ Secvența C/C++ corespunzătoare este:

```
int e1[100], e2[100];
int n, m, i;
cout<<"n=";
cin>>n;
cout<<"m=";
cin>>m;
for (i=1;i<=m;i++)
{
    cout<<"Dati extremitatile muchiei cu numarul "<<i;
    cin>>e1[i]>>e2[i];
}
```

♦ Construirea matricei de adiacență, când se cunoaște șirul muchiilor ca mai sus.

```
cout<<"n=";
cin>>n;
for (i=1;i<=m;i++)
{
    a[e1[i]][e2[i]]=1;
    a[e2[i]][e1[i]]=1;
}
```

◆ Construirea sirului muchiilor, ca mai sus, cînd se dă matricea de adiacență.

```

k=0;
for (i=1;i<=n-1;i++)
    for (j=i+1;j<=n;j++)
        if (a[i][j]==1)
        {
            k=k+1;
            e1[k]=i;
            e2[k]=j;
        }
m=k;

```

2. Se folosește un tablou unidimensional, cu numele  $e$ , caracterizat astfel:

- componentele sale sunt de tip structură;
- are  $m$  componente;
- $e_i$  reprezintă muchia  $i$ .

Pentru implementare este nevoie de:

```

typedef struct{
    int x;
    int y;
}muchie;
muchie e[20];

```

Accesul la muchia  $i$  se face:  $e[i].x \dots e[i].y$

◆ Secvența C/C++ corespunzătoare este:

```

cout<<"n="; cin>>n;
cout<<"m="; cin>>m;
for (i=1;i<=m;i++)
{
    cout<<"Dati extremitatile muchiei cu numarul "<<i;
    cin>>e[i].x>>e[i].y;
}

```

◆ Construirea matricei de adiacență, cînd se cunoaște sirul muchiilor ca mai sus.

```

cout<<"n="; cin>>n;
for (i=1;i<=m;i++)

```

```

    {
        a[e[i].x][e[i].y]=1;
        a[e[i].y][e[i].x]=1;
    }

```

◆ Construirea sirului muchiilor, ca mai sus, cînd se dă matricea de adiacență.

```

k=0;
for (i=1;i<=n-1;i++)
    for (j=i+1;j<=n;j++)
        if (a[i][j]==1) {
            k=k+1;
            e[k].x=i;
            e[k].y=j;
        }
m=k;

```

## 2.1.9. Parcursarea grafurilor

Fie  $G=(V, M)$  graf neorientat, unde  $V=\{x_1, x_2, \dots, x_n\}$  și  $M=\{m_1, m_2, \dots, m_p\}$ .

Prin parcursarea grafului  $G$  se înțelege vizitarea, într-un mod sistematic, a tuturor nodurilor, plecând de la un nod  $p_1$  (nod de plecare) mergând pe muchii incidente două câte două.

Un graf poate fi parcurs în următoarele două moduri:

- în lățime (BF = Breadth First)
- în adâncime (DF = Depth First)

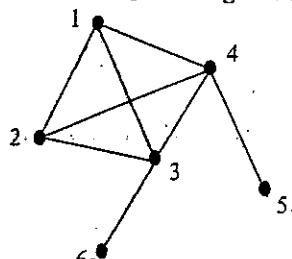
### Parcursarea în lățime (BF – breadth first)

Fie  $G=(V, M)$  un graf neorientat cu  $n$  vîrfuri ( $V=\{1, 2, \dots, n\}$ ) și  $m$  muchii.

Algoritmul de parcursare a grafului în lățime, folosind o coadă, este:

- inițial toate nodurile se consideră nevizitate;
- se citește nodul de plecare  $p_1$ , care se consideră acum vizitat, și se trece în coadă pe prima poziție;
- se trec în coadă toate nodurile nevizitate pînă în prezent și sunt adiacente cu nodul de plecare (odată cu trecerea lor în coadă se marchiază ca fiind vizitate);
- se trece la următorul element din coadă, care ia rolul nodului de plecare, și se reia pasul anterior;
- algoritmul se termină după ce sunt parcuse toate elementele din coadă.

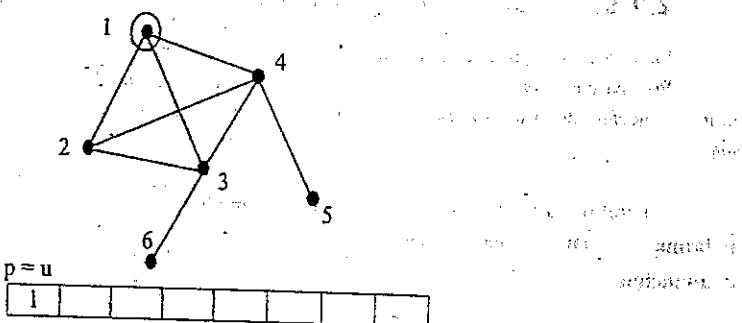
- ♦ Exemplul 1. Fie graful reprezentat grafic ca în figura de mai jos:



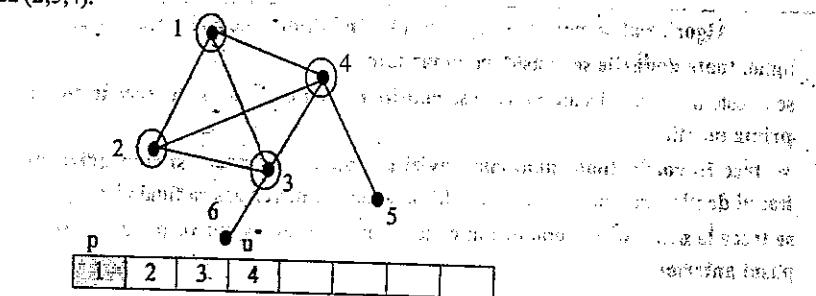
**Observație.** În continuare, un nod se consideră vizitat când este încercuit: și cu  $p$  și  $u$  notăm indicele primului, respectiv ultimului element din coadă.

Parcurgerea în lățime, a grafului de mai sus, presupune parcurgerea etapelor:

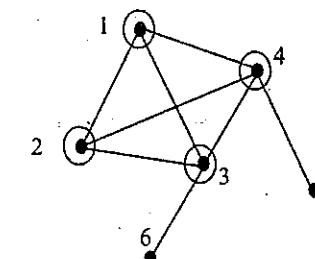
- ♦ La început, nici un nod nu este vizitat (graful arată ca în figura inițială, adică nici un nod nu este încercuit).
- ♦ Se pleacă de la nodul 1, care se trece în coadă pe prima poziție și se marchiază ca fiind vizitat.



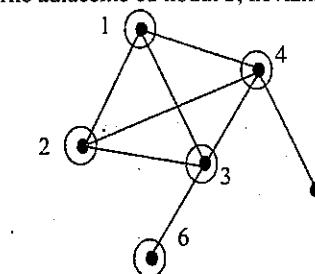
- ♦ Se vizitează și se trec în coadă toate nodurile adiacente cu nodul 1, nevizitate, încă (2,3,4).



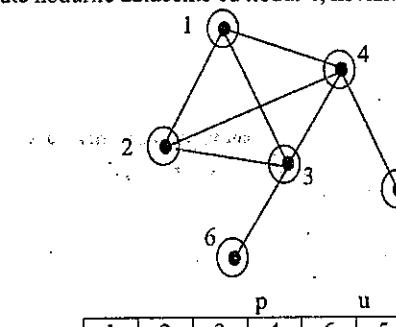
- ♦ Se trece la următorul element din coadă; acesta este 2. Se vizitează și se trec în coadă toate nodurile adiacente cu nodul 2, nevizitate încă (nu este nici un nod care să verifice condițiile).



- ♦ Se trece la următorul element din coadă; acesta este 3. Se vizitează și se trec în coadă toate nodurile adiacente cu nodul 3, nevizitate încă (6).



- ♦ Se trece la următorul element din coadă; acesta este 4. Se vizitează și se trec în coadă toate nodurile adiacente cu nodul 4, nevizitate încă (5).



- ◆ Se trece la următorul element din coadă; acesta este 6. Se vizitează și se trec în coadă toate nodurile adiacente cu nodul 6, nevizitate încă (nu este nici un nod care să verifice condițiile).

p	u
1	2

- ◆ Se trece la următorul element din coadă; acesta este 5. Se vizitează și se trec în coadă toate nodurile adiacente cu nodul 5, nevizitate încă (nu este nici un nod care să verifice condițiile).

p=u
1

◆ Algoritmul se încheie aici (nu mai sunt noduri).

Deci, parcurgerea în lățime a grafului este:

1 2 3 4 6 5

În continuare, vom prezenta programele C/C++ care implementează algoritmul prezentat mai sus.

### Programul 1 (abordare nerecursivă)

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

int a[20][20];
int coada[20], viz[20];
int i, n, el, j, p, u, pl, m, x, y;

main()
{
    clrscr();
    cout<<"n="; cin>>n;
    cout<<"m="; cin>>m;
    for (i=1;i<=m;i++)
    {
        cout<<"x y ="; cin>>x>>y;
        a[x][y]=1; a[y][x]=1;
    }
    for (i=1;i<=n;i++)
        viz[i]=0;
    cout<<"dati nodul de plecare :"; cin>>pl;
    viz[pl]=1;
    coada[1]=pl;
    p=1;
    u=1;
```

**citirea grafului**

**secvență care precizează  
că nodurile sunt nevizitate**

```
while (p<=u) {
    el=coada[p];
    for (j= 1;j<=n;j++)
        if ((a[el][j]==1) && (viz[j]==0))
    {
        u=u+1;
        coada[u]=j;
        viz[j]=1;
    }
    p=p+1;
}
for (i=1;i<=u;i++) cout<<coada[i]<< " ";
```

### Programul 2 (abordare recursivă)

#### Comentarii la funcția `parc_latime`:

- are un parametru formal, i, care reprezintă poziția curentă la care s-a ajuns în coadă;
- procedează astfel:
  - se parcurg nodurile grafului, cu j
  - dacă j este adiacent cu nodul curent din coadă și j este nevizitat
  - se adaugă la coadă
  - și se marchiază ca fiind vizitat;
  - dacă mai sunt elemente în coadă se trece la următorul și se reaplăzează funcția

```
#include <conio.h>
#include <iostream.h>
#include <stdio.h>
int a[20][20];
int coada[20], viz[20];
int i, n, j, u, pl, m, x, y;

void parc_latime(int i)
{
    int j;
    for (j=1;j<=n;j++)
        if ((a[coada[i]][j]==1) && (viz[j]==0)) {
            u=u+1;
            coada[u]=j;
            viz[j]=1;
        }
    if (i<=u) parc_latime(i+1);
}
```

```

main(){
    clrscr();
    cout<<"n="; cin>>n;
    cout<<"m="; cin>>m;
    for (i=1;i<=m;i++)
    {
        cout<<x y "; cin>>x>>y;
        a[x][y]=1;a[y][x]=1;
    }
    for (i=1;i<=n;i++)
        viz[i]=0;
    cout<<"dati nodul de plecare :"; cin>>pl;
    viz[pl]=1;
    coada[1]=pl;
    u=1;
    parc_latime(1);
    for (i=1;i<=u;i++)
        cout<<coada[i]<<" ";
    getch();
}

```

**citirea grafului**

secvența care precizează că nodurile sunt nevizitate

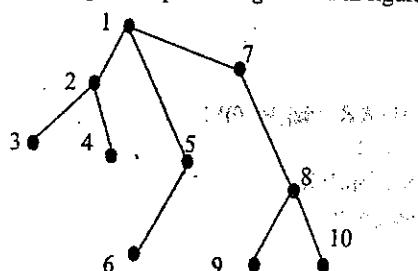
**Parcursarea în adâncime (DF-depth first)**

Fie  $G=(V, M)$  un graf neorientat cu  $n$  vârfuri ( $V=\{1, 2, \dots, n\}$ ) și  $m$  muchii.

**Algoritmul recursiv de parcursare a grafului în adâncime este implementat în funcția `parc_adancime`, caracterizată astfel:**

- are un parametru formal (nodul curent, asupra căruia se aplică);
- procedează astfel:
  - afisează nodul asupra căruia se aplică și-l marchiază ca fiind vizitat;
  - pentru fiecare vecin nevizitat de-al nodului curent
    - se autoapelează asupra sa;

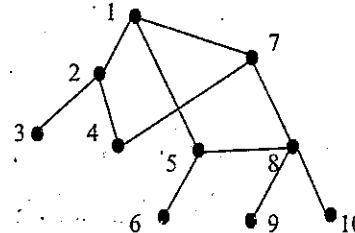
◆ **Exemplul 1.** Fie graful reprezentat grafic ca în figura de mai jos:



Aplicarea algoritmului de parcursare în adâncime, asupra grafului de mai sus, plecând de la primul nod, conduce la afișarea următoarei secvențe:

1 2 3 4 5 6 7 8 9 10

◆ **Exemplul 2.** Fie graful reprezentat grafic ca în figura de mai jos:



Aplicarea algoritmului de parcursare în adâncime, asupra grafului de mai sus, plecând de la primul nod, conduce la afișarea următoarei secvențe:

1 2 3 4 7 8 5 6 9 10

```

#include <conio.h>
#include <iostream.h>
#include <stdio.h>
int a[20][20];
int viz[20];
int i, n, j, pl, m, x, y;
void parc_adancime(int pl)
{
    int j;
    cout<<pl<<" "; viz[pl]=1;
    for (j=1;j<=n;j++)
        if ((a[pl][j]==1) && (viz[j]==0))
            parc_adancime(j);
}

```

```

main()
{
    cout<<"n m "; cin>>n>>m;
    for (i=1;i<=m;i++)
    {
        cout<<x y "; cin>>x>>y;
        a[x][y]=1;a[y][x]=1;
    }
    for (i=1;i<=n;i++)
        viz[i]=0;
    cout<<"dati nodul de plecare :"; cin>>pl;
    parc_adancime(pl);
}

```

## 2.1.10. Conexitate

În această secțiune, vor fi prezentate noțiunile:

- lanț
- ciclu
- graf conex
- componentă conexă

### Lanț

**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Se numește lanț, în graful  $G$ , o succesiune de noduri, notată  $L = [x_{i_1}, x_{i_2}, \dots, x_{i_k}]$ , cu proprietatea că oricare două noduri consecutive sunt adiacente, altfel spus  $[x_{i_1}, x_{i_2}], \dots, [x_{i_{k-1}}, x_{i_k}] \in M$

Se întâlnesc noțiunile:

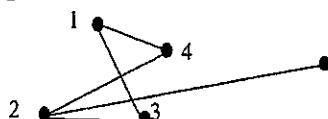
- extremitățile lanțului
  - fiind dat lanțul  $L = [x_{i_1}, x_{i_2}, \dots, x_{i_k}]$ , se numesc extremități ale sale nodurile  $x_{i_1}$  și  $x_{i_k}$  ( $x_{i_1}$  – extremitate inițială;  $x_{i_k}$  – extremitate finală);
- lungimea lanțului
  - fiind dat lanțul  $L = [x_{i_1}, x_{i_2}, \dots, x_{i_k}]$ , prin lungimea sa se înțelege numărul de muchii care apar în cadrul lui

#### ♦ Exemplu de lanț:

Fie graful  $G=(V, M)$  unde:  $V=\{1,2,3,4,5\}$

$$M=\{\{1,3\}, \{1,4\}, \{2,3\}, \{2,4\}, \{2,5\}\}$$

cu reprezentarea grafică astfel:



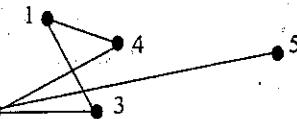
$L1=[1, 3, 2, 4]$  este în graful  $G$  lanț cu lungimea 3 și extremitățile 1 și 4.

$L2=[5, 2, 4, 1, 3, 2]$  este în graful  $G$  lanț cu lungimea 5 și extremitățile 5 și 2.

♦ Atenție! Dacă  $L = [x_{i_1}, x_{i_2}, \dots, x_{i_k}]$  este lanț în graful  $G$ , atunci și  $L1 = [x_{i_k}, \dots, x_{i_2}, x_{i_1}]$  este lanț în graful  $G$ .

**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Se numește lanț elementar, în graful  $G$ , lanțul  $L = [x_{i_1}, x_{i_2}, \dots, x_{i_k}]$  cu proprietatea că oricare două noduri ale sale sunt distincte (altfel spus: printr-un nod nu se trece decât o singură dată).

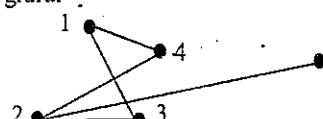
#### ♦ Exemplu: În graful



lanțul  $L1=[1, 3, 2, 4]$  este lanț elementar.

**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Se numește ciclu neelementar în graful  $G$  lanțul  $L = [x_{i_1}, x_{i_2}, \dots, x_{i_k}]$  cu proprietatea că nodurile sale nu sunt distincte două căte două (altfel spus: prin anumite noduri se trece de mai multe ori).

#### ♦ Exemplu: În graful

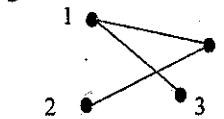


lanțul  $L2=[5, 2, 4, 1, 3, 2]$  este lanț neelementar (prin 2 s-a trecut de două ori).

### Ciclu

**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Se numește ciclu, în graful  $G$ , lanțul  $C = [x_{i_1}, x_{i_2}, \dots, x_{i_k}]$  cu proprietatea că  $x_{i_1} = x_{i_k}$  și are muchiile diferite două căte două.

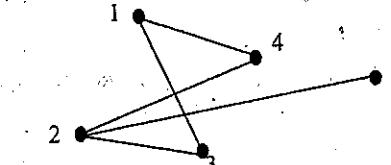
#### ♦ Exemplu: În graful



lanțul  $C=[1, 3, 2, 4, 1]$  este ciclu.

**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Se numește ciclu elementar, în graful  $G$ , un ciclu cu proprietatea că oricare două noduri ale sale, cu excepția primului și a ultimului, sunt distincte.

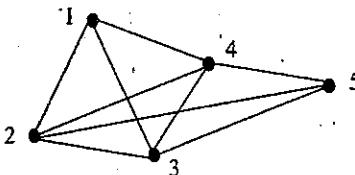
#### ♦ Exemplu: În graful



ciclul  $C=[1, 3, 2, 4, 1]$  este ciclu elementar.

**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Se numește ciclu neelementar, în graful  $G$ , un ciclu cu proprietatea că nodurile sale, cu excepția primului și a ultimului, nu sunt distințe.

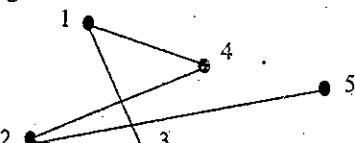
♦ **Exemplu:** În graful



ciclul  $C=[5, 3, 4, 1, 2, 4, 5]$  este ciclu neelementar (prin 4 s-a trecut de două ori).

**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Două cicluri  $C_1$  și  $C_2$  sunt egale, dacă muchiile lor induc același graf parțial al subgrafului generat de vârfurile ce aparțin lui  $C_1$ , respectiv lui  $C_2$ .

♦ **Exemplu:** În graful



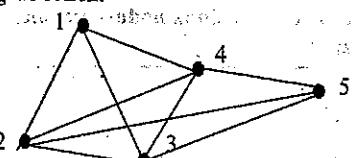
ciclul  $C_1=[1, 3, 2, 4, 1]$  este egal cu ciclul  $C_2=[3, 2, 4, 1, 3]$ .

**Observație.** Un ciclu se numește *par*, dacă lungimea sa este un număr par și se numește *impar*, dacă lungimea sa este un număr impar.

### Graf conex

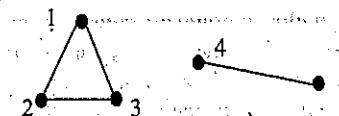
**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Graful  $G$  se numește conex dacă pentru oricare două vârfuri  $x$  și  $y$ ,  $x \neq y$ , există un lanț în  $G$  de la  $x$  la  $y$ .

♦ **Exemplu de graf conex:**



Graful este conex, deoarece oricare ar fi vârfurile  $x$  și  $y$ ,  $x \neq y$ , există un lanț în  $G$  care să le lege.

♦ **Exemplu de graf care nu este conex:**



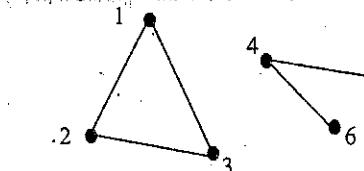
Graful nu este conex, deoarece există două vârfuri, cum ar fi 1 și 4, pentru care nu există nici un lanț în graf care să le lege.

### Componentă conexă

**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Se numește componentă conexă, un graf neorientat  $G_1=(V_1, M_1)$  care verifică următoarele condiții:

- este subgraf al grafului  $G$ ;
- este conex;
- nu există nici un lanț în  $G$  care să lege un nod din  $V_1$  cu un nod din  $V-V_1$ .

♦ **Exemplu:** Fie graful  $G=(V, M) : V=\{1,2,3,4,5,6\}, M=\{\{1,2\}, \{1,3\}, \{2,3\}, \{4,5\}, \{4,6\}\}$



Pentru graful de mai sus, graful  $G_1=(V_1, M_1)$  unde:  $V_1=\{4,5,6\}$  și  $M_1=\{\{4,5\}, \{4,6\}\}$

este componentă conexă, deoarece:

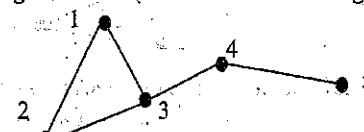
- este subgraf al grafului  $G$ ;
- este conex;
- nu există nici un lanț în  $G$  care să lege un nod din  $V_1$  cu un nod din  $V-V_1=\{1,2,3\}$ .

La fel se poate spune și despre graful  $G_2=(V_2, M_2)$  unde:  $V_2=\{1,2,3\}$  și  $M_2=\{\{1,2\}, \{1,3\}, \{2,3\}\}$

În concluzie, graful, din figura de mai sus, este format din două componente conexe.

**Observație.** Fie  $G=(V, M)$  un graf neorientat. Graful  $G$  este conex dacă și numai dacă este format dintr-o singură componentă conexă.

♦ **Exemplu de graf conex (este format dintr-o singură componentă conexă):**



**Observatie.** Fie  $G=(V, M)$  un graf neorientat. Pentru a verifica dacă graful este format din una sau mai multe componente conexe se procedează astfel:

- se parcurge graful, prin una din metodele de parcurgere;
- dacă după parcurgere mai există în graf noduri nevizitate, atunci graful este format din mai multe componente conexe, altfel este format dintr-o singură componentă conexă, adică graful este conex.

```
#include <conio.h>
#include <iostream.h>
#include <stdio.h>
int a[20][20];
int viz[20];
int i, n, j, pl, m, x, y, ok;

void parc_adancime(int pl)
{
    int j;
    viz[pl]=1;
    for (j=1;j<=n;j++)
        if ((a[pl][j]==1) && (viz[j]==0))
            parc_adancime(j);
}
main()
{
    cout<<"n m "; cin>>n>>m;
    for (i=1;i<=m;i++)
    {
        cout<<"x y "; cin>>x>>y;
        a[x][y]=1; a[y][x]=1;
    }
    for (i=1;i<=n;i++)
        viz[i]=0;

    cout<<"dati nodul de plecare ?"; cin>>pl;
    parc_adancime(pl);
    ok=0;
    for (j=1;j<=n;j++)
        if (viz[j]==0) ok=1;
    if (ok) cout<<"graful este format din mai multe componente conexe";
    else cout<<"graful este conex";
    getch();
}
```

nodul curent nu se mai afișează ca la problema de parcurgere

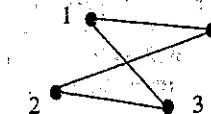
### 2.1.11. Grafuri Hamiltoniene

**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Se numește lant hamiltonian, în graful  $G$ , un lanț elementar care conține toate vârfurile grafului  $G$ .

◆ **Exemplu de lanț hamiltonian:**

Fie graful  $G=(V, M)$  unde:  $V=\{1,2,3,4\}$ ,  $M=\{\{1,3\}, \{1,4\}, \{2,3\}, \{2,4\}\}$

Reprezentarea sa grafică este:



Lanțul  $L=[1, 3, 2, 4]$  este, în graful  $G$ , lanț hamiltonian.

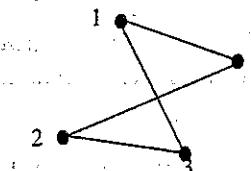
**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Se numește ciclu hamiltonian, în graful  $G$ , un ciclu elementar care conține toate vârfurile grafului  $G$ .

◆ **Exemplu de ciclu hamiltonian:**

Fie graful  $G=(V, M)$  unde:  $V=\{1,2,3,4\}$ ,

$M=\{\{1,3\}, \{1,4\}, \{2,3\}, \{2,4\}\}$

Reprezentarea sa grafică este:



Ciclul  $C=[1, 3, 2, 4, 1]$  este, în graful  $G$ , ciclu hamiltonian.

**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Graful  $G$  este hamiltonian dacă conține cel puțin un ciclu hamiltonian.

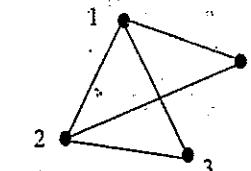
◆ **Exemplu de graf hamiltonian:**

Graful  $G=(V, M)$  unde:

$V=\{1,2,3,4\}$

$M=\{\{1,2\}, \{1,3\}, \{1,4\}, \{2,3\}, \{2,4\}\}$

cu reprezentarea grafică:



este hamiltonian, deoarece conține cel puțin un ciclu hamiltonian; ciclul  $C=[1, 3, 2, 4, 1]$  este, în graful  $G$ , ciclu hamiltonian.

**Observație.** Fie  $G=(V, M)$  un graf neorientat. Ca în graful  $G$  să existe un ciclu hamiltonian, trebuie ca el să aibă cel puțin trei vârfuri.

**Teoremă.** Graful complet  $K_n$  este graf hamiltonian.

**Demonstrație:**

Orice succesiune  $x_1, x_2, \dots, x_n, x_1$  de  $n+1$  noduri distincte (exceptie fac primul și ultimul) am alege, poate fi privită ca un ciclu hamiltonian, deci graful  $K_n$  este hamiltonian.

**Teoremă.** Fie  $G=(V, M)$  un graf neorientat. Dacă are  $n \geq 3$  noduri și gradul fiecărui vârf  $x$  verifică relația  $d(x) \geq n/2$ , atunci  $G$  este hamiltonian.

#### ◆ Problema determinări tuturor ciclurilor hamiltoniene dintr-un graf neorientat.

**Enunț:** Fie  $G=(V, M)$  un graf neorientat, cu  $n$  vârfuri. Să se determine toate ciclurile hamiltoniene din graful  $G$ .

**Rezolvare:** Problema se rezolvă folosind metoda Backtracking.

Pentru rezolvare se vor folosi:

$k$ : variabilă întreagă, care reprezintă la al cătelea nod s-a ajuns

(al doilea, al treilea...)

$x$ : vector cu componente întregi, cu proprietatea:

$x_k$  : reprezintă al  $k$ -lea nod din ciclu.

**Observație.** Pentru a evita parcurgerea unui drum de două ori se va recurge la strategia de a atribui lui  $x_1$  valoarea 1, adică toate ciclurile să plece de la primul nod, din acest motiv  $x_k \in \{2, \dots, n\}$ , pentru  $k \in \{2, \dots, n\}$ .

În concluzie, a rezolva problema înseamnă a determina vectorii:

$$x = (x_1, x_2, \dots, x_n), \text{ unde } x_1 = 1 \text{ și } x_k \in \{2, \dots, n\}, k \in \{2, \dots, n\};$$

Tabla va arăta astfel:

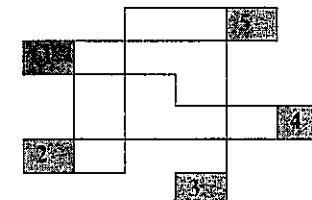
	n					
k	2	3	...	n-1	n	
	2	3	...	n-1	n	
	...	...	...	...	...	...
	2	3	...	n-1	n	
	2	3	...	n-1	n	
2	1					

2. La finalul ciclului  $x_k$  trebuie să se verifice că  $x_k = x_1$ .

Pentru reprezentarea grafului în program se va folosi matricea de adiacență, definită astfel:

- $a_{ij}=1$ , dacă există muchie între nodurile  $i$  și  $j$ ;
- $a_{ij}=0$ , dacă nu există muchie între nodurile  $i$  și  $j$ .

**Exemplu:** Pentru graful de mai jos



matricea de adiacență se definește astfel:

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

#### ◆ Concluzii:

1. Între nodurile  $k$  și  $i$ , există muchie dacă  $a_{ki}=1$  (și  $a_{ik}=1$ ), deci între nodurile  $x_k$  și  $x_i$  există muchie dacă  $a[x_k][x_i]=1$  (și  $a[x_i][x_k]=1$ ).
2. Nodul  $x_k$  trebuie să fie diferit de nodul  $x_i$ , pentru  $i=1 \dots k-1$ .
3. Nodul  $x_n$  trebuie să fie legat prin muchie de nodul  $x_1$ , adică  $a[x_n][x_1]=1$ .

#### ◆ Comentarii la procedura Valid:

Trebuie verificat că:

1. există muchie între nodurile  $x_{k-1}$  și  $x_k$ , adică trebuie verificat că  $a[x_{k-1}][x_k]=1$ ;
2. nodul  $x_k$  este diferit de toate nodurile prin care s-a trecut, adică:

  - $x_k \neq x_i$  pentru  $i=1 \dots k-1$ .

3. dacă s-a ajuns la al  $n$ -lea nod din ciclu, trebuie să existe muchie între acesta și primul nod, adică: dacă  $k=n$  atunci  $a[x_k][x_1]=1$ .

```
#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int sir[20];
sir x;
int i, k, n;
int as, ev;
int a[20][20];
```

```

void succ(sir x, int k, int &as)
{
    if (x[k]<n) {
        as=1;
        x[k]=x[k]+1;
    }
    else as=0;
}

void valid( sir x, int k, int &ev)
{
    ev=1;
    if (a[x[k-1]][x[k]]==0)           dacă nu există drum între  $x_{k-1}$  și  $x_k$ 
        ev=0;
    else {
        for (i=1;i=k-1;i++)
            if (x[i]==x[k])
                ev=0;
        if ((k==n) && (a[x[n]][x[1]]==0))
            ev=0;
    }
}

void afis(sir x, int k)
{
int i;
for (i=1;i<=k;i++)
    cout<<x[i]<<" ";
cout<<x[1]<<" "; cout<<endl;
}

main()
{
cout<<"n="; cin>>n;
for (i=1;i<=n-1;i++)
    for (j=i+1;j<=n;j++)
    {
        cin>>a[i][j];
        a[j][i]=a[i][j];
    }
}

```

```

x[1]=1;
k=2;

x[k]=1;           priviți tabla
while (k>1){
    do{
        succ(x,k,as);
        if (as)
            valid(x,k,ev);
    }while (as && !ev);

    if (as)
        if (k==n)
            afis(x,k);
        else {
            k=k+1;
            x[k]=1;
        }
    else k=k-1;
}
getche();
}

```

### 2.1.12. Grafuri Euleriene

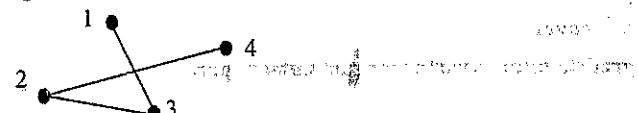
**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Se numește lant eulerian, în graful  $G$ , un lant care conține toate muchiile grafului  $G$ , fiecare muchie apărând în lant o singură dată.

♦ Exemplu de lanț eulerian:

Fie graful  $G=(V, M)$  unde:  $V=\{1,2,3,4\}$

$M=\{[1,3],[2,3],[2,4]\}$

Reprezentarea sa grafică este:



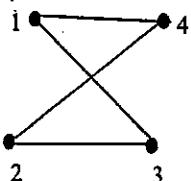
Lanțul  $L=[1, 3, 2, 4]$  este, în graful  $G$ , lanț eulerian.

**Definiție.** Fie  $G=(V, M)$  un graf neorientat. Se numește ciclu eulerian, în graful  $G$ , un ciclu care conține toate muchiile grafului  $G$ , fiecare muchie apărând în ciclu o singură dată.

♦ Exemplu de ciclu eulerian:

Fie graful  $G=(V, M)$  unde:  $V=\{1,2,3,4\}$ ,  $M=\{[1,3], [1,4], [2,3], [2,4]\}$

Reprezentarea sa grafică este:



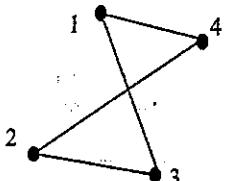
Ciclul  $C=[1, 3, 2, 4, 1]$  este, în graful  $G$ , ciclu eulerian.

**Teoremă.** Fie  $G=(V, M)$  un graf neorientat. Graful  $G$ , fără vârfuri izolate, este eulerian dacă și numai dacă este conex și gradele tuturor vâfurilor sale sunt numere pare.

♦ Exemplul 1. Graful  $G=(V, M)$ , unde:

$V=\{1,2,3,4\}$ ,  $M=\{[1,3], [1,4], [2,3], [2,4]\}$ ,

cu reprezentarea grafică:



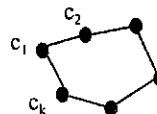
este graf eulerian, deoarece verifică condițiile teoremei anterioare, adică:

- nu are vârfuri izolate;
- este conex;
- gradele tuturor vâfurilor sunt numere pare.

**Definiție.** Un graf care conține cel puțin un ciclu eulerian se numește graf eulerian.

În continuare, se prezintă algoritmul de determinare a unui ciclu eulerian într-un graf neorientat.

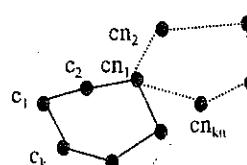
Pas1. Se determină ciclul  $C=(c_1, c_2, \dots, c_k, c_1)$ , unde  $c_1=1$ .



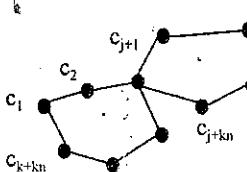
Pas2. Se caută, printre nodurile ciclului determinat la pasul anterior, un nod pentru care mai există muchii incidente cu el neluate încă. Fie acesta  $c_j$ ; construim ciclul

$C_n=(cn_1, cn_2, \dots, cn_{kn}, cn_1)$  unde  $cn_i=c_j$

$cn$  – ciclu nou



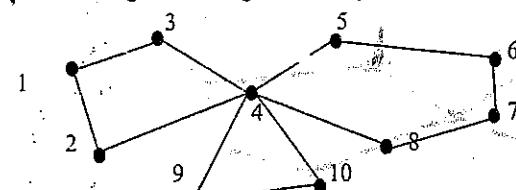
Pas3. Ciclul determinat la pasul 2 se "concatenează" la ciclul  $C$ , obținându-se astfel ciclul  $C=(c_1, c_2, \dots, c_j, c_{j+1}, \dots, c_{j+kn-1}, \dots, c_{k+kn}, c_1)$  figurat mai jos:



Pas4. Dacă nu sau ales toate muchiile, se reia pasul 2.

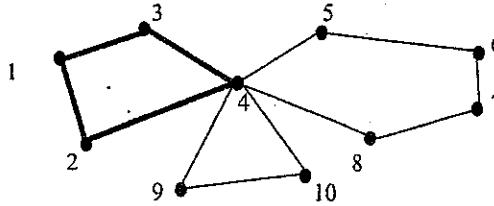
Pentru a înțelege mai bine algoritmul prezentat mai sus, urmăriți exercițiul de mai jos.

♦ Exercițiu: Pentru graful din figura de mai jos, să se determine un ciclu eulerian.

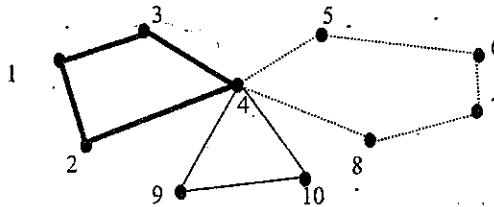


**Rezolvare:**

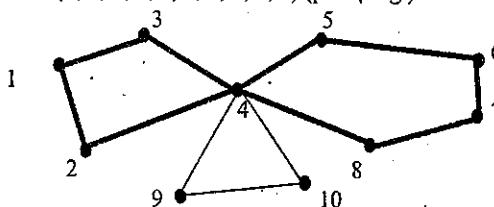
**Pas1.** Se determină un ciclu plecând de la nodul 1; fie acesta:  $C=(1, 3, 4, 2, 1)$  (priviți figura).



**Pas2.** Se caută, printre nodurile ciclului determinat la pasul anterior, un nod pentru care mai există muchii incidente cu el neluate încă; fie acesta nodul 4. Construim ciclul  $C_n$ , plecând de la acest nod:  $C_n=(4, 5, 6, 7, 8, 4)$  (priviți fig.).

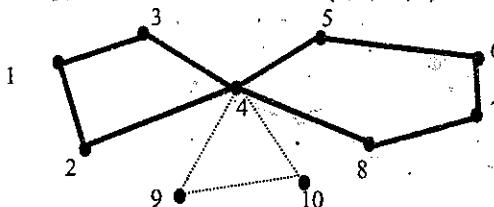


**Pas3.** Ciclul determinat la pasul 2 se "concatenează" la ciclul  $C$ , obținându-se astfel ciclul:  $C=(1, 3, 4, 5, 6, 7, 8, 4, 2, 1)$  (priviți fig.).



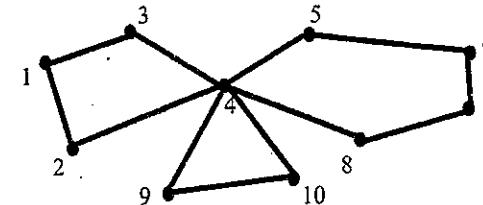
**Pas4.** Deoarece mai sunt muchii neluate încă, se reia pasul 2.

**Pas2'.** Se caută, printre nodurile ciclului determinat până în prezent, un nod pentru care mai există muchii incidente cu el neluate încă; acesta este nodul 4. Construim ciclul  $C_n$ , plecând de la acest nod:  $C_n=(4, 9, 10, 4)$ .



**Pas3'.** Ciclul determinat la pasul 2' se "concatenează" la ciclul  $C$ , obținându-se astfel

ciclul:  $C=(1, 3, 4, 9, 10, 4, 5, 6, 7, 8, 4, 2, 1)$ .



În continuare, se prezintă programul de determinare a unui ciclu eulerian într-un graf neorientat.

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

typedef int mat[20][20];
typedef int sir[20];

mat a;
sir viz, c, cn, gr;
int i, k, j, n, kn, poz, m, eulerian, x, y;

int grad(int i){
    int s, j;
    s=0;
    for(j=1;j<=n;j++)
        s=s+a[i][j];
    return s;
}

int varf_isolate(){
    int i, ok;
    ok=0;
    for(i=1;i<=n;i++)
        if (grad(i)==0) ok=1;
    return ok;
}

int grade_pare(){
    int i, ok;
    ok=1;
    for(i=1;i<=n;i++)
        if (grad(i)%2!=0) ok=0;
}
```

```

    return ok;
void adancime(int i){
    int j;
    viz[i]=1;
    for (j=1;j<=n;j++)
        if ((viz[j]==0) && (a[i][j]==1)) adancime(j);
}
int conex(){
    int i, ok;
    for (i=1;i<=n;i++)
        viz[i]=0;
    adancime(1);
    ok=1;
    for (i=1;i<=n;i++)
        if (viz[i]==0) ok=0;
    return ok;
}
main(){
clrscr();
    cout<<"m="; cin>>m;
    cout<<"n="; cin>>n;
    for (i=1;i<=m;i++){
        cout<<"x y"; cin>>x>>y;
        a[x][y]=1;
        a[y][x]=1;
    }

    for (i=1;i<=n;i++)
        gr[i]=grad(i);
    eulerian=(!varf_izolate()) && grade_pare() && conex();

    if (!eulerian) cout<<"graful nu este eulerian";
    else{
        c[1]=1;
        k=i;
        do{
            for (j=1;j<=n;j++)
                if (a[c[k]][j]==1){
                    k=k+1;
                    c[k]=j;
                }
        } while (c[k]!=1);
    }
}

```

```

    gr[c[k]]=gr[c[k]]-1;
    gr[c[k-1]]=gr[c[k-1]]-1;
    a[c[k-1]][c[k]]=0;
    a[c[k]][c[k-1]]=0;
    break;
}
} while (c[k]!=1);

while (k-1<m){
    for (j=1;j<=k-1;j++)
        if (gr[c[j]]>0) {
            cn[1]=c[j];
            poz=j;
            break;
        }
    kn=1;
    do{
        for (j=1;j<=n;j++)
            if (a[cn[kn]][j]==1){
                kn=kn+1;
                cn[kn]=j;
                gr[cn[kn]]=gr[cn[kn]]-1;
                gr[cn[kn-1]]=gr[cn[kn-1]]-1;
                a[cn[kn-1]][cn[kn]]=0;
                a[cn[kn]][cn[kn-1]]=0;
                break;
            }
    } while (cn[kn]!=cn[1]);

    for (j=k;j>=poz;j--)
        c[j+kn-1]=c[j];
    for (j=1;j<=kn-1;j++)
        c[poz+j]=cn[j+1];
    k=k+kn-1;
}

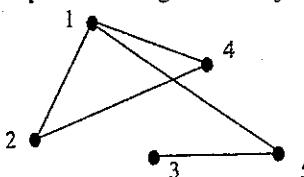
for (i=1;i<=k;i++)
    cout<<c[i]<<" ";
getche();
}

```

## 2.2. PROBLEME PROPUSE

### 2.2.1. Noțiunea de graf neorientat

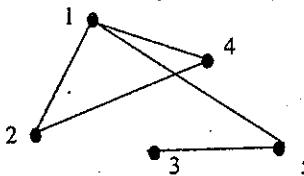
- Să se precizeze dacă rețelei de circulație din orașul dumneavoastră i se poate asocia un graf neorientat; în caz afirmativ, să se definească graful corespunzător.
- Având la dispoziție un grup de  $n$  persoane,  $n \in N^*$ , să se precizeze dacă i se poate asocia un graf neorientat; în caz afirmativ, să se definească graful corespunzător.
- Având la dispoziție o hartă cu  $n$  țării,  $n \in N^*$ , să se precizeze dacă i se poate asocia un graf neorientat; în caz afirmativ, să se definească graful corespunzător.
- Având la dispoziție toate stelele, să se precizeze dacă li se poate asocia un graf neorientat; justificați răspunsul;
- Pentru graful reprezentat în figura de mai jos



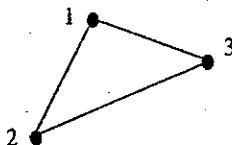
- precizați mulțimea nodurilor;
- precizați mulțimea muchiilor;
- dăți exemple de noduri adiacente;
- pentru fiecare muchie precizați extremitățile sale;
- dăți exemple de muchii incidente.

### 2.2.2. Noțiunea de graf parțial

- Să se determine două grafuri parțiale ale grafului de mai jos:



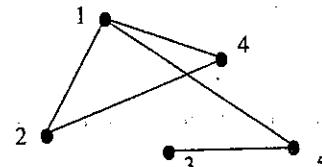
- Să se determine toate grafurile parțiale ale grafului de mai jos:



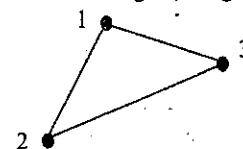
- Fie  $G$  un graf neorientat, cu  $n$  vârfuri și  $m$  muchii. Să se determine numărul grafurilor parțiale ale grafului  $G$ .

### 2.2.3. Noțiunea de subgraf

- Să se determine două subgrafuri ale grafului de mai jos:



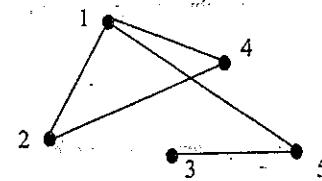
- Să se determine toate subgrafurile grafului de mai jos:



- Fie  $G$  un graf neorientat, cu  $n$  vârfuri și  $m$  muchii. Să se determine numărul subgrafurilor grafului  $G$ .

### 2.2.4. Gradul unui vârf

- Înălțind dat graful de mai jos, să se determine pentru fiecare vârf în parte gradul său; să se precizeze vârfurile terminale și vârfurile izolate.



- Să se demonstreze că orice graf  $G$ , cu  $n \geq 2$  noduri, conține cel puțin două vârfuri care au același grad.

- Să se verifice dacă există grafuri cu 5 noduri pentru care:

a) sirul gradelor vârfurilor sale este:

$$1, 2, 3, 0, 5$$

b) sirul gradelor vârfurilor sale este:

$$1, 2, 3, 4, 1$$

- Fie graful  $G$ , cu  $n$  vârfuri și  $m$  muchii, astfel încât să fie îndeplinită relația:

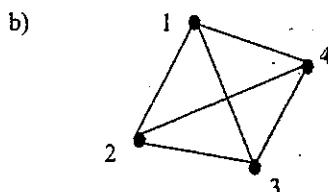
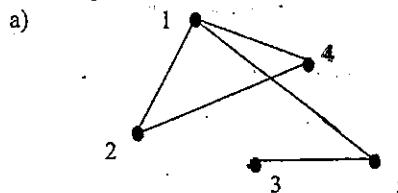
$$m > \frac{(n-1)(n-2)}{2}$$

Să se demonstreze că  $G$  nu are vârfuri izolate.

5. Fie  $G$  un graf neorientat, cu  $n$  vârfuri și  $m$  muchii, reprezentat prin matricea de adiacență. Să se realizeze programe, în C/C++, care:
- afișează gradele tuturor vâfurilor;
  - afișează vâfurile de grad par;
  - afișează vâfurile izolate;
  - afișează vâfurile terminale;
  - verifică dacă graful are vâfurile izolate;
  - verifică dacă graful are vâfurile terminale;
  - verifică dacă graful are vâfurile interioare (nu sunt nici izolate nici terminale);
  - verifică dacă graful are toate vâfurile izolate;
  - verifică dacă graful are toate vâfurile interioare (nu sunt nici izolate nici terminale);
  - afișează gradul unui vârf dat;
  - afișează vecinii unui nod dat, vf;
  - verifică dacă un vârf dat este terminal, izolat sau interior;
  - afișează gradul cel mai mare și toate vâfurile care au acest grad;
  - afișează frecvența vâfurilor:  
izolate : n1  
terminale : n2  
interioare : n3
  - fiind dat sirul  $g_1, \dots, g_m$ , să se verifice dacă poate reprezenta sirul gradelor vâfurilor în această ordine;
  - fiind dat sirul  $g_1, \dots, g_m$ , să se verifice dacă poate reprezenta sirul gradelor vâfurilor (nu neapărat în această ordine).

## 2.2.5. Graf complet

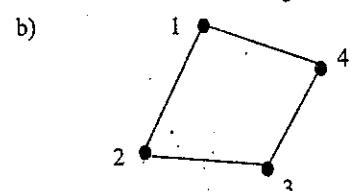
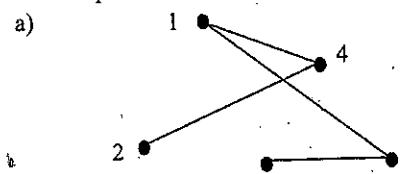
1. Fiind date grafurile de mai jos, să se precizeze care dintre ele este complet și să se justifice răspunsul.



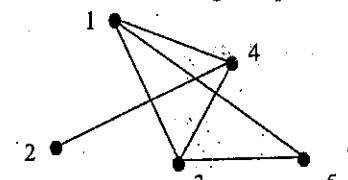
- Pentru grafurile  $K_3$  și  $K_4$ :
  - să se precizeze gradul fiecărui vârf;
  - să se precizeze numărul de muchii;
  - să se realizeze o reprezentare grafică.
- Fie graful  $G$ , cu  $n$  vârfuri, dat prin matricea de adiacență. Să se realizeze un subprogram care precizează dacă graful este complet, astfel:
  - facând o analiză asupra nodurilor;
  - facând o analiză asupra muchiilor.
- Fie graful  $G$ , cu  $n$  vârfuri, dat prin matricea de adiacență. Să se realizeze subprograme care precizează:
  - câte muchii mai trebuie adăugate pentru a deveni complet;
  - între ce noduri mai trebuie adăugate muchii astfel încât graful să devină complet.

## 2.2.6. Graf bipartit

1. Fiind date grafurile de mai jos, să se precizeze care dintre ele este bipartit și să se justifice răspunsul.



2. Ce muchie trebuie eliminată din graful prezentat mai jos astfel încât să devină bipartit?

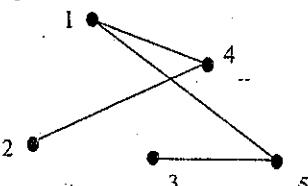


- Fie graful bipartit  $G$ , fără vârfuri izolate, dat prin matricea de adiacență. Să se realizeze un program care determină multimiile  $V_1$  și  $V_2$  despre care se vorbește în definiție.
- Fie graful  $G$  cu  $n$  vârfuri, dat prin matricea de adiacență. Să se realizeze un program care precizează dacă graful este bipartit.

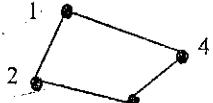
## 2.2.7. Graf bipartit complet

1. Fiind date grafurile de mai jos, să se precizeze care dintre ele este bipartit complet și să se justifice răspunsul.

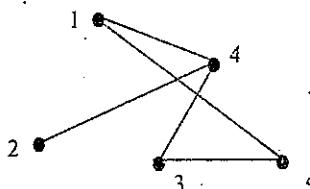
a)



b)



2. Ce muchie trebuie adăugată în graful prezentat mai jos astfel încât să devină bipartit complet:

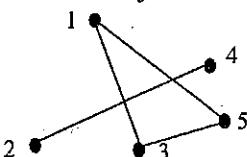


3. Fie graful  $G$ , cu  $n$  vârfuri reprezentate  $\{1, \dots, n\}$ . Presupunând că graful este bipartit complet, astfel încât  $|V_1| = p$ , cu  $p < n$  și  $V_1 = \{1, \dots, p\}$ , să se construiască matricea de adiacență.
4. Fie graful  $G$ , cu  $n$  vârfuri reprezentate  $\{1, \dots, n\}$ . Presupunând că graful este bipartit complet și că  $V_1$  este formată din nodurile reprezentate prin numere pare, să se construiască matricea de adiacență.
5. Să se determine numărul total de grafuri bipartit complete cu  $n$  vârfuri date.

## 2.2.8. Reprezentarea grafurilor

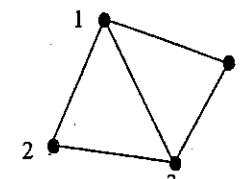
1. Fiind date grafurile de mai jos:

a)



## GRAFURI NEORIENTATE

b)



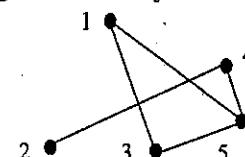
să se precizeze pentru fiecare în parte:

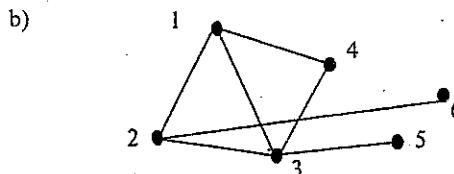
1. matricea de adiacență
2. listele de adiacență
3. sirul muchiilor
2. Fiind dată o matrice pătratică de ordin  $n$ , să se precizeze dacă poate fi considerată matricea de adiacență a unui graf neorientat cu  $n$  vârfuri.
3. Să se determine numărul total de grafuri neorientate care au vârfurile  $\{1, \dots, n\}$ .
4. Să se realizeze un program care generează matricele de adiacență ale tuturor grafurilor neorientate cu vârfurile  $\{1, \dots, n\}$ .
5. Să se realizeze un program în C/C++ care, fiind date matricele de adiacență  $A_1$  și  $A_2$  de dimensiune  $n$ , verifică dacă matricea  $A_2$  poate reprezenta un graf parțial al grafului reprezentat de matricea  $A_1$ .
6. Să se realizeze un program în C/C++ care să verifice dacă un graf este pentru alt graf subgraf.
7. Să se realizeze un program care generează toate grafurile bipartite complete cu  $n$  vârfuri.
8. Fie  $G$  un graf neorientat, cu  $n$  vârfuri și cu muchiile  $m_1, m_2, \dots, m_p$ . Să se realizeze un program, în C/C++, care determină toate grafurile parțiale ale lui  $G$ .
9. Fie  $G$  un graf neorientat, cu vârfurile  $1, 2, \dots, n$ , dat prin matricea de adiacență. Să se realizeze un program în C/C++ care determină toate subgrafurile lui  $G$ .
10. Fiind dat un grup de persoane, reprezentate prin numere de la 1 la  $n$ , și precizându-se relațiile de simpatie astfel: pentru fiecare persoană  $i$ , se citesc numerele de ordine ale persoanelor pe care aceasta le simpatizează (numerele se dau pe aceeași linie separate prin spații), să se precizeze dacă în grupul de persoane amintit, toate simpatiile sunt reciproce.
11. Să se realizeze un program care permite desenarea unui graf neorientat cu  $n$  vârfuri și  $m$  muchii, cunoscându-se lista muchiilor.

## 2.2.9. Parcursarea grafurilor

1. Fiind date grafurile de mai jos:

a)





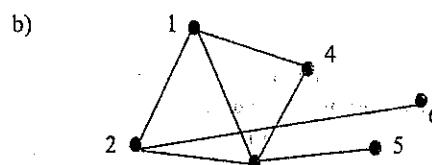
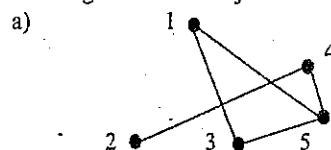
să se precizeze, pentru fiecare în parte, lista nodurilor obținută în urma parcurgerii:

- în lățime;
- în adâncime.

2. Fie  $G$  un graf, cu  $n$  noduri și  $m$  muchii. Precizându-se un nod, de exemplu nodul 1, să se realizeze un program care afișează toate nodurile accesibile din acest nod.

## 2.2.10. Conexitate

1. Fiind date grafurile de mai jos:



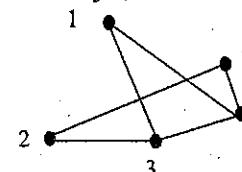
să se precizeze pentru fiecare în parte: un lanț, un lanț de lungime 4, un lanț elementar, un ciclu, un ciclu elementar, două cicluri egale.

2. Fie  $G$  un graf neorientat, cu  $n$  noduri și  $m$  muchii. Precizându-se două noduri, np (nodul de plecare) și ns (nodul de sosire), să se determine toate lanțurile elementare care le admit ca extremități.
3. Fiind dat un graf neorientat, cu  $n$  noduri și  $m$  muchii, să se determine toate lanțurile elementare care au cea mai mare lungime.
4. Să se realizeze un program care, fiind dat un graf neorientat, verifică dacă conține un ciclu de lungime 4.
5. Să se realizeze un program care, fiind dat un graf neorientat, afișează toate ciclurile elementare de lungime  $p$ , plecând de la nodul 1.
6. Să se realizeze un program care, fiind dat un graf neorientat, determină câte componente conexe are.
7. Să se realizeze un program care, fiind dat un graf neorientat, determină toate componentele conexe ale sale.

8. Să se realizeze un program care, fiind dat un graf neorientat, determină toate perechile de vârfuri între care există cel puțin un lanț.
9. Speologii au cercetat n culoare subterane, pentru a stabili dacă aparțin același peșteri. Prin tehnici specifice de curenți de aer și de colorare a cursurilor de apă, a fost demonstrată existența unor canale de legătură între mai multe culoare. Precizându-se perechile de culoare între care au fost stabilite legături, să se afle dacă sistemul de culoare aparține unei singure peșteri.
10. Într-un grup de  $n$  persoane, se precizează perechi de persoane care se consideră prietene. Folosind principiul că "prietenul prietenului meu mi-ești prieten", să se determine grupurile cu un număr maxim de persoane între care se pot stabili relații de prietenie, directe sau induse.

## 2.2.11. Cicluri Hamiltoniene

1. Pentru graful de mai jos, să se dea exemplu de un lanț și de un ciclu hamiltonian.



2. Să se realizeze un program care pentru un graf dat verifică dacă satisfac condițiile teoremei prezentate în secțiunea 1.2.11.
3. Să se arate că numărul ciclurilor hamiltoniene ale grafului  $K_n$ , cu  $n \geq 3$ , este  $\frac{(n-1)!}{2}$ .
4. Să se arate că numărul ciclurilor elementare ale grafului  $K_n$ , cu  $n \geq 3$ , este  $\frac{1}{2} \sum_{k=3}^n n(n-1)\dots(n-k+1)$ .
5. Să se realizeze un program care pentru un graf dat verifică dacă este hamiltonian.
6. La curtea regelui Artur s-au adunat  $2n$  cavaleri și fiecare dintre ei are printre cei prezenti cel mult  $n-1$  dușmani. Arătați că Merlin, consilierul lui Artur, poate să-i așeze pe cavaleri, la o masă rotundă, în așa fel încât nici unul dintre ei să nu stea alături de vreun dușman de-al său.
7. Se consideră  $n$  persoane. Fiecare are printre cei prezenti cel mult  $n/2$  dușmani. Să se determine toate posibilitățile de așezare a acestora la o masă rotundă astfel încât nici unul dintre ei să nu stea lângă un dușman al său.
8. La un cenacu literar sunt invitați un număr de  $n$  elevi, identificați cu  $1 \dots n$ , de la Licee, identificate  $1 \dots L$ . Să se determine toate modalitățile de așezare a acestora la o masă rotundă astfel încât să nu fie doi elevi de la același liceu vecini.

## 2.3. INDICAȚII ȘI RĂSPUNSURI

### 2.3.1. Noțiunea de graf neorientat

#### Problema 1

Răspunsul este afirmativ (Da). Definim graful neorientat asociat rețelei astfel:

- mulțimea nodurilor este mulțimea intersecțiilor dintre străzi și a capetelor de străzi (la ieșirea din oraș); este mulțime finită și nevidă;
- mulțimea muchiilor este mulțimea bucătilor de stradă dintre două intersecții sau dintre o intersecție și un capăt de stradă (la ieșirea din oraș).

#### Problema 2

Răspunsul este afirmativ (Da). Definim graful neorientat asociat astfel:

- mulțimea nodurilor este mulțimea persoanelor (este mulțime finită și nevidă);
- muchia dintre nodurile  $x$  și  $y$  se definește ca fiind reprezentarea ideii "persoanele  $x$  și  $y$  se cunosc" (pot exista nenumărate definiții; dați și altele).

#### Problema 3

Răspunsul este afirmativ (Da). Definim graful neorientat asociat astfel:

- mulțimea nodurilor este mulțimea țărilor (este mulțime finită și nevidă);
- muchia dintre nodurile  $x$  și  $y$  se definește ca fiind reprezentarea ideii "din țara  $x$  se poate ajunge în țara  $y$ , cu avionul" (pot exista nenumărate definiții).

#### Problema 4

Dacă admitem că există o infinitate de stele: Răspunsul este negativ (Nu), deoarece mulțimea nodurilor trebuie să fie, conform definiției, finită și nevidă.

Dacă admitem că stelele sunt într-un număr finit: Răspunsul este pozitiv (Da) și putem defini graful neorientat asociat lor astfel:

- mulțimea nodurilor este mulțimea stelelor (este mulțime finită și nevidă);
- muchia între nodurile  $x$  și  $y$  se definește ca fiind reprezentarea ideii "de pe steaua  $x$  se poate vedea steaua  $y$ " (pot exista nenumărate definiții).

#### Problema 5

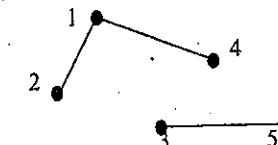
- $V=\{1, 2, 3, 4, 5\}$ ;
- $M=\{[1,2], [1,4], [1,5], [2,4], [3,5]\}$ ;
- Nodul 1 este adiacent cu nodul 4; nodul 3 este adiacent cu nodul 5;
- $[1,2] : 1 \text{ și } 2; [1,4] : 1 \text{ și } 4; [1,5] : 1 \text{ și } 5; [2,4] : 2 \text{ și } 4; [3,5] : 3 \text{ și } 5$ ;
- $[1,2]$  și  $[1,4]; [2,4]$  și  $[1,4]; \dots$

### 2.3.2. Noțiunea de graf parțial

#### Problema 1

Cele două grafuri parțiale vor fi reprezentate prin desen în figurile de mai jos:

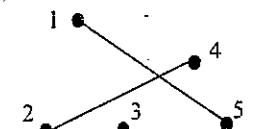
Primul graf parțial (se elimină muchiile  $[1,5], [2,4]$ )



$$V_1 = \{1, 2, 3, 4, 5\}$$

$$M_1 = \{[1,2], [1,4], [3,5]\}$$

Al doilea graf parțial (se elimină muchiile  $[1,2], [1,4], [3,5]$ )



$$V_1 = \{1, 2, 3, 4, 5\}$$

$$M_1 = \{[1,5], [2,4]\}$$

#### Problema 2

Grafurile parțiale, care se obțin plecând de la graful din enunț, sunt în număr de:

- 1 dacă se elimină 0 muchii  $(C_3^0)$
- 3 dacă se elimină o muchie  $(C_3^1)$
- 3 dacă se elimină două muchii  $(C_3^2)$
- 1 dacă se elimină toate muchiile  $(C_3^3)$

deci, în total,  $1+3+3+1=8$  grafuri parțiale.

#### Problema 3

Având în vedere că: "un graf parțial al grafului  $G$  se obține prin eliminarea unui număr oarecare de muchii", putem scrie:

Numărul total al grafurilor parțiale ale grafului  $G$ , este suma dintre numărul grafurilor parțiale care se obțin:

- |   |             |
|---|-------------|
| prin eliminarea unui număr de 0 muchii:     | $C_m^0$     |
| prin eliminarea unui număr de 1 muchie:     | $C_m^1$     |
| prin eliminarea unui număr de 2 muchii:     | $C_m^2$     |
| .....                                       | .....       |
| prin eliminarea unui număr de $m-1$ muchii: | $C_m^{m-1}$ |
| prin eliminarea unui număr de $m$ muchii:   | $C_m^m$     |

prin adunare se obține:  $C_m^0 + C_m^1 + C_m^2 + \dots + C_m^{m-1} + C_m^m = 2^m$

**Observatie.** Suma de mai sus se calculează astfel:

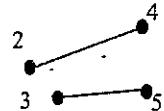
$$\text{în relația } C_m^0 + C_m^1 x^1 + C_m^2 x^2 + \dots + C_m^{m-1} x^{m-1} + C_m^m x^m = (1+x)^m \\ \text{se înlocuiește } x \text{ cu valoarea } 1$$

### 2.3.3. Noțiunea de subgraf

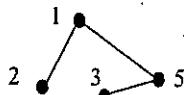
#### Problema 1

Cele două subgrafuri vor fi reprezentate prin desen în figurile de mai jos:

**Primul subgraf** (se elimină nodul 1 (odată cu el și muchiile incidente [1,2], [1,4], [1,5]))



**Al doilea subgraf** (se elimină nodul 4 (odată cu el și muchiile incidente [1,4], [2,4]))



#### Problema 2

Subgrafurile care se obțin plecând de la graful din enunț sunt în număr de:

- 1 dacă se elimină 0 noduri  $(C_3^0)$
- 3 dacă se elimină 1 nod  $(C_3^1)$
- 3 dacă se elimină 2 noduri  $(C_3^2)$

deci, în total  $1+3+3=7$  subgrafuri.

**Atenție!** Toate nodurile nu se pot elimina pentru că s-ar obține un graf cu multimea vârfurilor vidă (acest lucru nu este permis de definiție).

#### Problema 3

Având în vedere că: un subgraf al grafului G se obține prin eliminarea unui număr oarecare de noduri (diferit de numărul total de noduri ale grafului), putem scrie:

Numărul total al subgrafurilor grafului G este suma dintre numărul subgrafurilor care se obțin:

prin eliminarea unui număr de 0 noduri:  $C_n^0$

prin eliminarea unui număr de 1 noduri:  $C_n^1$

prin eliminarea unui număr de 2 noduri:  $C_n^2$

.....

prin eliminarea unui număr de  $n-1$  noduri:  $C_n^{n-1}$

### GRAFURI NEORIENTATE

$$\text{prin adunare se obține: } C_n^0 + C_n^1 + C_n^2 + \dots + C_n^{n-1} = 2^n - 1$$

**Observatie.** Suma de mai sus se calculează astfel:

$$\text{în relația } C_n^0 + C_n^1 x^1 + C_n^2 x^2 + \dots + C_n^{n-1} x^{n-1} + C_n^n x^n = (1+x)^n$$

se înlocuiește  $x$  cu valoarea 1 și se obține:

$$C_n^0 + C_n^1 + C_n^2 + \dots + C_n^{n-1} + C_n^n = 2^n \text{ de unde rezultă:}$$

$$C_n^0 + C_n^1 + C_n^2 + \dots + C_n^{n-1} = 2^n - C_n^n = (2^n - 1)$$

### 2.3.4. Gradul unui vârf

#### Problema 1

- |           |           |           |                         |
|-----------|-----------|-----------|-------------------------|
| $d(1)=3;$ | $d(3)=1;$ | $d(5)=2;$ | $3$ este vârf terminal; |
| $d(2)=2;$ | $d(4)=2;$ | $d(6)=0;$ | $6$ este vârf izolat;   |

#### Problema 2

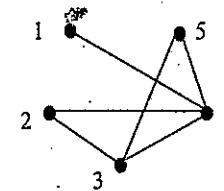
Fie  $V=\{x_1, x_2, \dots, x_n\}$ . Presupunem că graful nu conține două vârfuri care să aibă același grad, deci  $d(x_i) \neq d(x_j)$  pentru  $i \neq j$  și  $d(x_i) \in \{0, 1, \dots, n-1\}$  pentru  $i=1 \dots n$ . În concluzie, sirul gradelor vârfurilor coincide cu  $\{0, 1, \dots, n-1\}$ , făcând eventual abstracție de ordine (presupunem de exemplu:  $d(x_1)=0, d(x_2)=1, \dots, d(x_n)=n-1$ ). Dică, așa cum se vede și în exemplul prezentat între paranteze, există un vârf care are gradul 0, adică nu este legat de nici un vârf, și un vârf care are gradul  $n-1$ , adică este legat de toate celelalte, deci și de cel de gradul 0. **Contradicție**, deoarece cel de gradul 0 nu este legat de nici un vârf.

În concluzie, presupunerea făcută la începutul rezolvării este falsă.

#### Problema 3

- Categoric nu, deoarece dacă ar exista un astfel de graf ar conține un vârf care ar avea gradul 5 (adică ar fi legat de încă 5 vârfuri). Acest lucru nu se poate întâmpla deoarece fără el în graf mai sunt decât 4 vârfuri.

- Dacă un astfel de graf ar exista, l-am putea reprezenta astfel:



Nodul 3 a fost legat de nodul 5 (dar putea fi legat și de nodul 1); acest lucru nu este posibil deoarece astfel acesta ar căpăta gradul 2 și el practic îl are 1.

**Problema 4**

Presupunem că graful are un vârf izolat și toate celelalte noduri generează un subgraf complet (oricare două dintre ele sunt legăte printr-o muchie), adică graful ar avea  $\frac{(n-1)(n-2)}{2}$  muchii, oricum nu mai multe decât  $\frac{(n-1)(n-2)}{2}$ , așa cum se spune în enunț. Cum situația aleasă este cea mai convenabilă în acest sens, înseamnă că un astfel de graf nu poate exista.

**Problema 5**

**Observație.** A determina gradul vârfului  $i$ , înseamnă a determina numărul elementelor care au valoarea 1 și se găsesc pe linia  $i$  în matricea de adiacență.

Mai jos, este prezentată funcția care returnează gradul vârfului  $i$  (funcția constă în calcularea sumei elementelor de pe linia  $i$  din matricea de adiacență).

**Funcția care returnează gradul nodului  $i$** 

```
int grad( int i )
{
    int s, j;
    s=0;
    for (j=1;j<=n;j++)
        s=s+a[i][j];
    return s;
}
```

**problemă a**

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
typedef int mat[20][20];
mat a;
int i, j, n;

int grad( int i )
{
    int s, j;
    s=0;
    for (j=1;j<=n;j++)
        s=s+a[i][j];
    return s;
}
```

**GRAFURI NEORIENTATE**

```
main()
{
    clrscr();
    cout<<"n=";
    cin>>n;
    for (i=1;i<=n-1;i++)
        for (j=i+1;j<=n;j++)
        {
            cout<<"a["<<i<<","<<j<<"]=";
            cin>>a[i][j];
            a[j][i]=a[i][j];
        }
    for (i=1;i<=n;i++)
        cout<<"varful "<<i<<" are gradul"<< grad(i)<<endl;
    getch();
}
```

**Observație.** La problemele care urmează, nu prezentăm decât secvența prin care diferă de problema anterioară. Problemele se fac asemănător, adică:

- se scrie funcția **grad** (de determinare a gradului unui vârf);
- se citește matricea de adiacență;
- se scrie secvența .....

**Problema b**

```
for (i=1;i<=n;i++)
    if (grad(i) % 2==0)
        cout<<i<<" ";
```

**Problema c**

```
for (i=1;i<=n;i++)
    if (grad(i)==0)
        cout<<i<<" ";
```

**Problema d**

```
for (i=1;i<=n;i++)
    if (grad(i)==1)
        cout<<i<<" ";
```

**Problema e**

```
ok=0;
for (i=1;i<=n;i++)
    if (grad(i)==0) ok=1;
if (ok) cout<<"Da";
else cout<<"Nu";
```

**Problema f**

```
ok=0;
for (i=1;i<=n;i++)
    if (grad(i) =1)
        ok=1;
if (ok) cout<<"Da";
else cout<<"Nu";
```

**Problema h**

```
ok=1;
for (i=1;i<=n;i++)
    if (grad(i) !=0)
        ok=0;
if (ok) cout<<"Da";
else cout<<"Nu";
```

**Problema j**

```
cout<<"Dati varful : "; cin>>vf;
cout<<" varful "<<vf<<" are gradul "<<
grad(vf);
```

**Problema l**

```
cout<<"Dati varful : "; cin>>vf;
if (grad(vf)==0) cout<<"varful "<<vf<<" este izolat"<<endl;
else if (grad(vf)==1) cout<<"varful "<<vf<<" este terminal";
else cout<<"varful "<<vf<<" este interior";
```

**Problema m****Cum se procedează?**

- se determină gradul cel mai mare, în gr\_max (este o problemă simplă de determinare a maximului)
- se parcurg toate nodurile, cu i
  - dacă gradul vîrfului i este egal cu gr\_max  
se afișează nodul i

**Problema g**

```
ok=0;
for (i=1;i<=n;i++)
    if (grad(i) >1)
        ok=1;
if (ok) cout<<"Da";
else cout<<"Nu";
```

**Problema i**

```
ok=1;
for (i=1;i<=n;i++)
    if (grad(i) <=1)
        ok=0;
if (ok) cout<<"Da";
else cout<<"Nu";
```

**Problema k**

```
cout<<"Dati varful : "; cin>>vf;
for (j=1;j<=n;j++)
    if (a[vf][j]==1) cout<<j<<" ";
```

**GRAFURI NEORIENTATE**

```
gr_max=grad(1);
for (i=2;i<=n;i++)
    if (grad(i)>gr_max)
        gr_max=grad(i);

cout<<"Cel mai mare grad este "<< gr_max<<endl;
cout<<"si nodurile care au acest grad sunt"<<endl;
for (i=1;i<=n;i++)
    if (grad(i)==gr_max)
        cout<<i<<" ";
```

**Problema n**

```
n1=0; n2=0; n3=0;
for (i=1;i<=n;i++)
    if (grad(i)==0) n1=n1+1;
    else if (grad(i)==1) n2=n2+1;
    else n3=n3+1;
cout<<"In graful dat sunt : "<<endl;
cout<<"    "<<n1<<" varfuri izolate"<<endl;
cout<<"    "<<n2<<" varfuri terminale"<<endl;
cout<<"    "<<n3<<" varfuri interioare"<<endl;
```

**Problema o**

```
for (i=1;i<=n;i++)
    cin>>g[i];
ok=1;
for (i=1;i<=n;i++)
    if (grad(i)!=g[i])
        ok=0;
cout<<ok;
```

**Problema p****Cum se procedează?**

- se citește sirul g;
- se construiește sirul gradelor vîrfurilor, gr;
- se sortează crescător cele două siruri, folosind funcția:

```

void sort_crescator(sir x, int n)
{
    int i, ok, aux;
    do{
        ok=1;
        for (i=1;i<=n-1;i++)
            if (x[i]>x[i+1]) {
                aux= x[i];
                x[i]= x[i+1];
                x[i+1]=aux;
                ok=0;
            }
    }while (ok==0);
}

- se verifică dacă sunt identice;

for (i=1;i<=n;i++)
    cin>>g[i];
for (i=1;i<=n;i++)
    gr[i]=grad(i);

sort_crescator(g,n);
sort_crescator(gr,n);

ok=1;
for (i=1;i<=n;i++)
    if (gr[i]!=g[i])
        ok=0;
cout<<ok;

```

### 2.3.5. Graf complet

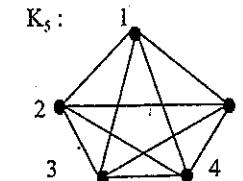
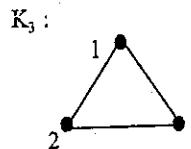
#### Problema 1

- a) Graful nu este complet, deoarece există două noduri între care nu există muchie (1 și 3).
- b) Graful este complet, deoarece oricare ar fi două vârfuri distincte există o muchie care le unește.

#### Problema 2

- a)  $K_3 : d(x)=3-1=2 \quad \forall x \in V$ ;       $K_3 : d(x)=5-1=4 \quad \forall x \in V$ ;
- b)  $K_3 : m=3(3-1)/2=3$ ;       $K_5 : m=5(5-1)/2=10$ ;

c)



#### Problema 3

##### problema a

**Observatie.** A verifica că un graf este complet, făcând o analiză asupra vârfurilor, înseamnă a verifica dacă toate vârfurile au gradul  $n-1$ .

```

#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int mat[20][20];
    mat a;
    int i, j, n;

int grad( int i )
{
    int s, j;
    s=0;
    for (j=1;j<=n;j++)
        s=s+a[i][j];
    return s;
}

int complet()
{
    int i, c;
    c=1;
    for (i=1;i<=n;i++)
        if (grad(i)!=n-1)
            c=0;
    return c;
}

main()
{
    clrscr();
    cout<<"n=";
    cin>>n;
}

```

```

for (i=1;i<=n-1;i++)
    for (j=i+1;j<=n;j++)
    {
        cout<<"a["<<i<<","<<j<<"]=";
        cin>>a[i][j];
        a[j][i]=a[i][j];
    }
cout<<complet();
getche(); }

```

**problemă b**

**Observație.** A verifica că un graf este complet, făcând o analiză asupra muchiilor, înseamnă a verifica dacă are  $n(n-1)/2$  muchii, altfel spus: trebuie verificat dacă toate elementele de deasupra diagonalei principale din matricea de adiacență sunt egale cu 1.

```

#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int mat[20][20];
mat a;
int i, j, n;

int complet(){
    int i, j, c;
    c=1;
    for (i=1;i<=n-1;i++)
        for (j=i+1;j<=n;j++)
            if (a[i][j]!=1)
                c=0;
    return c;
}

main(){
    clrscr();
    cout<<"n="; cin>>n;
    for (i=1;i<=n-1;i++)
        for (j=i+1;j<=n;j++)
    {
        cout<<"a["<<i<<","<<j<<"]=";
        cin>>a[i][j];
        a[j][i]=a[i][j];
    }
}

```

```

if (complet()) cout<<"da";
else cout<<"nu";
getche();
}

```

**Problema 4****problemă a**

**Observație.** Pentru a putea preciza câte muchii mai trebuie adăugate astfel încât graful să devină complet, trebuie să se numere elementele care au valoarea 0 în matricea de adiacență și se găsească deasupra diagonalei principale.

```

#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int mat[20][20];
mat a;
int i, j, n;

int nr_muchi(){
{
    int nr, i, j;
    nr=0;
    for (i=1;i<=n-1;i++)
        for (j=i+1;j<=n;j++)
            if (a[i][j]==0) nr=nr+1;
    return nr;
}

main(){
    clrscr();
    cout<<"n="; cin>>n;
    for (i=1;i<=n-1;i++)
        for (j=i+1;j<=n;j++)
    {
        cout<<"a["<<i<<","<<j<<"]=";
        cin>>a[i][j];
        a[j][i]=a[i][j];
    }
    cout<<"Mai trebuie adăugate "<<nr_muchi()<<" pentru a deveni complet";
    getche();
}

```

**problemă b**

**Observație.** Pentru a afișa nodurile din graf, între care trebuie să se adauge muchii, astfel încât acesta să devină complet, se procedează astfel:  
 – se parcurge matricea de adiacență deasupra diagonalei principale și acolo unde  $a_{ij}=0$ , adică nu există muchie între nodurile  $i$  și  $j$ , se afisează  $i$  și  $j$ .

```
#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int mat[20][20];
mat a;
int i, j, n;

void afisare_noduri()
{
    int i, j;
    for (i=1; i<=n-1; i++)
        for (j=i+1; j<=n; j++)
            if (a[i][j]==0)
                cout<<i<<"  "<<j<<endl;
}

main()
{
    clrscr();
    cout<<"n="; cin>>n;
    for (i=1; i<=n-1; i++)
        for (j=i+1; j<=n; j++)
    {
        cout<<"a["<<i<<","<<j<<"]=";
        cin>>a[i][j];
        a[j][i]=a[i][j];
    }
    afisare_noduri();
    getch();
}
```

**2.3.6. Graf bipartit****Problema 1**

- Graful este bipartit, deoarece respectă definiția;  $V_1=\{1, 2, 3\}$ ,  $V_2=\{4, 5\}$ .
- Graful este bipartit, deoarece respectă definiția;  $V_1=\{1, 3\}$ ,  $V_2=\{2, 4\}$ .

**GRAFURI NEORIENTATE****Problema 2**

Pentru a obține un graf bipartit, trebuie eliminată muchia  $[1, 3]$ ;  $V_1=\{1, 2, 3\}$ ,  $V_2=\{4, 5\}$ .

**Problema 3**

Se procedează astfel:

- la început  $V_1=[ ]$  și  $V_2=[ ]$
- se parcurge matricea de adiacență, linie cu linie, deasupra diagonalei principale dacă pe linia  $i$  se găsește elementul  $a[i][j]=1$ , atunci

dacă  $i$  aparține deja lui  $V_2$  atunci

$j$  se adaugă la multimea  $V_1$ :  $V_1:=V_1+[j]$

altfel

$i$  se adaugă la multimea  $V_1$ :  $V_1:=V_1+[i]$

$j$  se adaugă la multimea  $V_2$ :  $V_2:=V_2+[j]$

**Problema 4**

Se procedează astfel:

Programul se realizează asemănător cu cel de la problema anterioară, numai că, la final, în loc să se afișeze multimile  $V_1$  și  $V_2$  se verifică dacă intersecția lor este vidă.

dacă  $V_1 \cap V_2 = \emptyset$  atunci

graful este bipartit

altfel

graful nu este bipartit

**2.3.7. Graf bipartit complet****Problema 1**

- Graful este bipartit, deoarece respectă definiția,  $V_1=\{1, 2, 3\}$ ,  $V_2=\{4, 5\}$ , dar nu este complet, deoarece există noduri în  $V_1$  (ex. 2) nelegate de toate nodurile din  $V_2$  (ex. 5).
- Graful este bipartit deoarece respectă definiția,  $V_1=\{1, 3\}$ ,  $V_2=\{2, 4\}$ , dar cum toate nodurile din  $V_1$  sunt legate de toate nodurile din  $V_2$  înseamnă că este bipartit complet.

**Problema 2**

Pentru a obține un graf bipartit complet trebuie adăugată muchia  $[2, 5]$ ;  $V_1=\{1, 2, 3\}$ ,  $V_2=\{4, 5\}$ .

**Problema 3**

Se procedează astfel:

Pe liniile  $1..p$  din matricea de adiacență

se pune valoarea 1 pe coloanele  $p+1..n$ , deoarece toate elementele din  $V_1$  sunt legate de toate elementele din  $V_2$

(nu trebuie uitat că matricea de adiacență este simetrică față de diagonala principală)

```

for (i=1;i<=p;i++)
    for (j=p+1;j<=n;j++)
    {
        a[i][j]=1;
        a[j][i]=1;
    }

```

**Problema 4**

**Se procedează astfel:**

Este suficient să găsim construirea matricei deasupra diagonalei principale, pentru că ea este simetrică față de diagonala principală. Matricea se construiește astfel:

- pe liniile pare, de deasupra diagonalei principale,
- se pune valoarea 1 pe coloanele impare, deoarece toate elementele din V1 sunt legate de toate elementele din V2
- pe liniile impare, de deasupra diagonalei principale;
- se pune valoarea 1 pe coloanele pare, deoarece toate elementele din V2 sunt legate de toate elementele din V1

(nu trebuie uitat că matricea de adiacență este simetrică față de diagonala principală)

```

for (i=1;i<=n-1;i++)
    for (j=i+1;j<=n;j++)
        if ((i % 2==0) && (j % 2!=0))
        {
            a[i][j]=1;
            a[j][i]=1;
        }
    for (i=1;i<=n-1;i++)
        for (j=i+1;j<=n;j++)
            if ((i % 2!=0) && (j % 2==0))
            {
                a[i][j]=1;
                a[j][i]=1;
            }

```

**GRAFURI NEORIENTATE****Problema 5**

Un graf bipartit complet este unic determinat de o partiție a lui V în două submulțimi V1 și V2, disjuncte și nevide. A determina toate grafurile bipartit complete, înseamnă a determina în câte moduri se pot construi mulțimile V1 și V2. Pentru aceasta procedăm astfel:

În mulțimea V1 se pune nodul 1, pentru a nu repeta soluțiile

(mai sunt n-1 noduri nepuse).

**Partiția\_1** la V1 se adaugă 0 noduri (sunt  $C_{n-1}^0$  situații) iar la V2 restul

**Partiția\_2** la V1 se adaugă 1 noduri (sunt  $C_{n-1}^1$  situații) iar la V2 restul

**Partiția\_3** la V1 se adaugă 2 noduri (sunt  $C_{n-1}^2$  situații) iar la V2 restul

**Partiția\_n-1** la V1 se adaugă n-2 noduri (sunt  $C_{n-1}^{n-2}$  situații) iar la V2 restul

(altă partiție nu mai există, pentru că la V1 nu se pot adăuga încă n-1 noduri deoarece V2 ar fi vidă, în acest caz, și ar fi în contradicție cu definiția grafului bipartit).

În total sunt  $C_{n-1}^0 + C_{n-1}^1 + C_{n-1}^2 + \dots + C_{n-1}^{n-2}$  posibilități de construire. Această sumă se calculează astfel:

$$C_{n-1}^0 + C_{n-1}^1 + C_{n-1}^2 + \dots + C_{n-1}^{n-2} + C_{n-1}^{n-1} = 2^{n-1} \text{ de unde rezultă că:}$$

$$C_{n-1}^0 + C_{n-1}^1 + C_{n-1}^2 + \dots + C_{n-1}^{n-2} = 2^{n-1} - C_{n-1}^{n-1} = 2^{n-1} - 1.$$

**2.3.8. Reprezentarea grafurilor****Problema 1**

a)

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Vârful i	Lista vecinilor lui i
1	3, 5
2	4
3	1, 5
4	2
5	1, 3

M={[e1[1],e2[1]]=1,3, [e1[2],e2[2]]=1,5, [e1[3],e2[3]]=2,4, [e1[4],e2[4]]=3,5}

b)

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Vârful i	Lista vecinilor lui i
1	2, 3, 4
2	1, 3
3	1, 2, 4
4	1, 3

$$M = \{[e1[1], e2[1]] = [1, 2], [e1[2], e2[2]] = [1, 3], [e1[3], e2[3]] = [1, 4], [e1[4], e2[4]] = [2, 3], [e1[5], e2[5]] = [3, 4]\}$$

**Problema 2**

A verifică dacă matricea poate reprezenta matricea de adiacență a unui graf neorientat, trebuie verificate următoarele:

- matricea are pe diagonala principală numai elemente egale cu 0;
- matricea are deasupra diagonalei principale numai elemente de 0 și 1;
- matricea este simetrică față de diagonala principală.

```

ok1=1;
for (i=1;i<=n;i++)
    if (a[i][i]!=0)
        ok1=0;
ok2=1;
for (i=1;i<=n-1;i++)
    for (j=i+1;j<=n;j++)
        if (!(a[i][j]==0) || (a[i][j]==1))
            ok2=0;
ok3=1;
for (i=1;i<=n-1;i++)
    for (j=i+1;j<=n;j++)
        if (a[i][j] != a[j][i])
            ok3=0;
ok=ok1 && ok2 && ok3;
cout<<ok;

```

**Problema 3**

Un graf neorientat, cu  $n$  vârfuri, este unic determinat de o matrice de adiacență pătratică de ordinul  $n$ . Deci, a determina grafurile neorientate cu  $n$  vârfuri înseamnă a determina toate matricele pătratice de ordinul  $n$ , caracterizate astfel:

- au numai elemente de 0 și/sau 1;
- pe diagonala principală au numai elemente de 0;
- sunt simetrice față de diagonala principală.

Citind cu atenție caracteristicile matricei de adiacență, constatăm cu ușurință că pentru a determina o astfel de matrice este suficient să-i determinăm elementele de deasupra diagonalei principale, deoarece matricea are deasupra diagonalei principale  $(n^2-n)/2$  elemente, problema se reduce la a determina toți vectori de lungime  $(n^2-n)/2$  cu elemente de 0 și 1; acești vectori sunt în număr de  $2^{(n^2-n)/2}$  (este vorba de determinarea submulțimilor unei mulțimi cu  $(n^2-n)/2$  elemente).

**Problema 4**

Problema se reduce la a determina toți vectori de lungime  $(n^2-n)/2$  cu elemente de 0 și/sau 1; (vezi explicațiile de la problema anterioară). Acest lucru se face folosind metoda Backtracking, astfel:

- se generează pe rând toți vectorii de lungime  $p=(n^2-n)/2$  cu elemente de 0 și/sau 1;
- pentru fiecare vector generat, se construiește matricea de adiacență corespunzătoare, astfel:

- pe diagonala principală se pune 0;
- deasupra diagonalei principale se pun elementele vectorului, astfel:

```

ks=1;
for (i=1;i<=n-1;i++)
    for (j=i+1;j<=n;j++)

```

```
{
```

```
    a[i][j]=x[ks];

```

```
    ks=ks+1;

```

```
    a[j][i]=a[i][j];
}
}
```

- elementele de sub diagonala principală trebuie puse astfel încât matricea să fie simetrică față de diagonala principală.

În concluzie, trebuie generate elementele mulțimii  $\{(x_1, x_2, \dots, x_p) | x_k \in \{0, 1\}, k=1, \dots, p\}$ .

$x=(x_1, x_2, \dots, x_p)$  unde  $x_k \in \{0, 1\}$ ;

$k \in \{1, \dots, p\}$ ;

$p=(n^2-n)/2$

```

#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int sir[100];
sir x;
int i, k, n, p;
int as, ev;
int a[100][100];

void succ(sir x, int k, int &as)
{
    if (x[k]<1) {
        as=1;
        x[k]=x[k]+1;
    }
    else as=0;
}
void valid( int &ev)
{
    ev=1;
}

void afis(sir x)
{
    int i, j, ks;
    ks=1;
    for (i=1;i<=n-1;i++)
        for (j=i+1;j<=n;j++)
    {
        a[i][j]=x[ks];
        ks=ks+1;
        a[j][i]=a[i][j];
    }
}

```

generarea matricei de adiacență,  
plecând de la vectorul generat

```

for (i=1;i<=n;i++)
{
    for (j=1;j<=n;j++)
        cout<<a[i][j]<<" ";
    cout<<endl<<endl;
}
cout<<endl<<endl;

main()
{
    cout<<"n="; cin>>n;
    p=(n*n-n)/2;
    k=1;
    x[k]=-1;
    while (k>0) {
        do{
            succ(x,k,as);
            if (as)
                valid(ev);
        }while (as && !ev);
        if (as)
            if (k==p) afis(x);
            else {
                k=k+1;
                x[k]=-1;
            }
        else k=k-1;
    }
}

```

### afisarea matricei de adiacență

priviți tabla

### Problema 5

Matricea de adiacență A2 reprezintă un graf parțial al grafului reprezentat de matricea de adiacență A1 dacă acolo unde elementele matricei A1 sunt 0 și elementele corespunzătoare din matricea A2 sunt 0; în rest nu contează, adică, dacă elementele lui A1 sunt 1 elementele corespunzătoare din A2 pot fi 0 sau 1 (acest lucru se exprimă astfel:  $A2[i][j] \leq A1[i][j]$ ). Este suficient să facem verificarea pentru elementele de deasupra diagonalei principale.

```

ok=1;
for (i=1;i<=n-1;i++)
    for (j=i+1;j<=n;j++)
        if (!(A2[i][j] <= A1[i][j]))
            ok=0;

```

```
if (ok) cout<<"A2 reprezinta un graf partial al grafului reprezentat de A1";
else cout<<"A2 nu reprezinta un graf partial al grafului reprezentat de A1";
```

**Problema 7**

Un graf bipartit complet este unic determinat de o partiție a lui  $V$  în două submulțimi  $V_1$  și  $V_2$ , disjuncte și nevide. A determina toate grafurile bipartit complete, înseamnă a determina toate modurile în care se pot construi mulțimile  $V_1$  și  $V_2$ , din elementele  $\{1 \dots n\}$ . Acest lucru se realizează astfel:

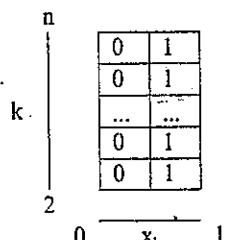
În mulțimea  $V_1$  se pune nodul 1, pentru a nu repeta soluțiile (mai sunt  $n-1$  noduri nepuse). Problema revine la a determina toate posibilitățile de a plasa vârfurile  $2 \dots n$  în prima sau în a doua mulțime; orice astfel de plasare convine cu excepția plasării tuturor vârfurilor în prima mulțime.

Pentru a rezolva această problemă vom folosi metoda Backtracking, astfel:

Se generează toți vectorii:

$x = (x_1, x_2, \dots, x_n)$  cu  $x_1=1$  și  $x_k \in \{0, 1\}$  pentru  $k \in \{2, \dots, n\}$ , fără ca toate elementele să fie egale cu 1, cu următoarea semnificație:

dacă  $x_i=1$ , atunci i face parte din  $V_1$   
altfel i face parte din  $V_2$



```
#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int sir[100];
sir x;
int i, k, n;
int as, ev;

void succ(sir x, int k, int &as)
{
    if (x[k]<1) {
        as=1;
        x[k]=x[k]+1;
    }
}
```

```
else as=0;
}

void valid( int &ev)
{
    ev=1;
}

void afis(sir x, int k)
{
    int i, s;
    s=0;
    for (i=1;i<=k;i++)
        s=s+x[i];
    if (s!=n)
    {
        cout<<"V1 : ";
        for (i=1;i<=n;i++)
            if (x[i]==1) cout<<i<<" ";
        cout<<endl;
        cout<<"V2 : ";
        for (i=1;i<=n;i++)
            if (x[i]==0) cout<<i<<" ";
        cout<<endl<<endl;
    }
}

main(){
    clrscr();
    cout<<"n=";
    cin>>n;
    x[1]=1;
    k=2;
    x[k]=-1;
    while (k>1)
    {
        do{
            succ(x,k,as);
            if (as)
                valid(ev);
        }while (as && !ev);

        if (as)
            if (k==n) afis(x,k);
    }
}
```

se verifică dacă toate componentele  
au valoarea 1

din V1 fac parte numai elementele i  
pentru care  $x_i=1$

din V2 fac parte numai elementele i  
pentru care  $x_i=0$

priviți tabla

```

        else {
            k=k+1;
            x[k]=-1;
        }
    else k=k-1;
}
getche();
}

```

**Problema 8**

După cum se vede în enunțul problemei, graful se dă prin precizarea numărului de noduri și a șirului muchiilor sale  $m_1 \dots m_p$ . Plecând de la faptul că un graf parțial al său se obține prin eliminarea unui număr oarecare de muchii, păstrând aceeași mulțime de vârfuri, problema dată se reduce la generarea submulțimilor mulțimii  $\{m_1 \dots m_p\}$ . Acest lucru îl vom realiza folosind metoda Backtracking, astfel:

- se generază submulțimile mulțimii  $\{m_1 \dots m_p\}$
- pentru fiecare astfel de submulțime generată, construim matricea de adiacență și o afișăm.

Se generează toți vectorii:

$x=(x_1, x_2, \dots, x_p)$  cu  $x_k \in \{0, 1\}$  pentru  $k \in \{1, \dots, p\}$  cu următoarea semnificație:

dacă  $x_i=0$ , atunci  $m_i$  se elimină din mulțimea  $\{m_1 \dots m_p\}$

altfel  $m_i$  nu se elimină din mulțimea  $\{m_1 \dots m_p\}$

k	0	1
	0	1
	...	...
	0	1
	0	1

```

#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int sir[100];
typedef struct{
    int x, y;
}muchie;

```

```

sir x;
int a[100][100];
int i, k, n, p, nr, j;
int as, ev;
muchie m[100];

```

```
void succ(sir x, int k, int &as)
```

```
{
    if (x[k]<1) {
        as=1;
        x[k]=x[k]+1;
    }
    else as=0;
}
```

```
void valid(int &ev){
```

```
    ev=1;
}
```

```
void afis(sir x, int k){
```

```
    int i;
```

```
    for (i=1;i<=k;i++)
```

```
        if (x[i]==1) {
            a[m[i].x][m[i].y]=1;
            a[m[i].y][m[i].x]=1;
        }
    }
```

```
nr=nr+1;
```

```
cout<<"matricea de adiacenta al celui de-al "<<nr<<-lea graf parțial"<<endl;
```

```
for (i=1;i<=n;i++)
```

```
{
```

```
    for (j=1;j<=n;j++)
        cout<<a[i][j]<<" ";
    cout<<endl;
}
```

**construirea matricei de adiacență**

**afișarea matricei de adiacență**

```

main(){
clrscr();
nr=0;
cout<<"n="; cin>>n;

cout<<"p="; cin>>p;
for (i=1;i<=p;i++)
    cin>>m[i].x>>m[i].y;

k=1;
x[k]=-1;
while (k>0)
{
    do{
        succ(x,k,as);
        if (as)
            valid(ev);
    }while (as && !ev);
    if (as)
        if (k==p) afis(x,k);
        else {
            k=k+1;
            x[k]=-1;
        }
    else k=k-1;
}
getche();
}

```

priviti tabla

### Problema 9

Tinând cont de faptul că un subgraf al unui graf se obține prin eliminarea unui număr de noduri, și a multor incidente cu aceste noduri, tragem următoarea concluzie:

Problema se reduce la a genera toate submulțimile multimii  $\{1, \dots, n\}$  (fără mulțimea vidă), iar pentru fiecare submulțime generată se afișează decât muchii ale grafului care au ambele extremități printre elementele submulțimii.

Rezolvarea problemei se face folosind metoda Backtracking, și constă în a genera elementele multimii:

$((x_1, x_2, \dots, x_n) \mid x_k \in \{0,1\}, k=1, \dots, n, \text{nu toate nule})$   
(dacă  $x_k=1$ , înseamnă că nodul  $k$  face parte din submulțime, altfel nu)

$x=(x_1, x_2, \dots, x_n)$  unde  $x_k \in \{0,1\}$ ;

$k \in \{1, \dots, n\};$

$n$ $k$ $1$	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td></tr> <tr><td>...</td><td>...</td></tr> <tr><td>0</td><td>1</td></tr> <tr><td>0</td><td>1</td></tr> </table>	0	1	0	1	...	...	0	1	0	1	$0 \quad x_k \quad 1$
0	1											
0	1											
...	...											
0	1											
0	1											

#include <conio.h>

#include <iostream.h>

#include <stdio.h>

typedef int sir[100];

typedef int mat[20][20];

int e1, e2;

int i, j, n, m, ns, k;

mat a;

sir x;

int as, ev;

void succ(sir x, int k, int &as)

{

if (x[k]<1) {

as=1;

x[k]=x[k]+1;

}

else as=0;

}

```

void afis(sir x, int k){
int i;
ns=ns+1;                                ns este numărul soluției
cout<<"subgraful cu numarul "<< ns<<"are nodurile "; curente
cout<<endl;
for (i=1;i<=k;i++)
    if (x[i]==1)                         se afișează nodurile
        cout<<i<<" ";
cout<<endl;
cout<<"si muchiile ";
for (i=1;i<=k-1;i++)
    for (j=i+1;j<=k;j++)
        if ((x[i]==1) && (x[j]==1) && (a[i][j]==1))      se afișează muchiile
            cout<<" ("<<i<<","<<j<<") "<<endl;
cout<<endl<<endl;
}

void valid(sir x, int k, int &ev){
int i, s;
ev=1;
if (k==n){                                aici se elimină soluția
    s=0;                                  cu toate componentele
    for (i=1;i<=k;i++)                  egale cu 0
        s=s+x[i];
    if (s==0) ev=0;
}
}

main()
clrscr();
cout<<"n="; cin>>n;
cout<<"m="; cin>>m;
for (i=1;i<=m;i++)
{
    cout<<"e1 e2 "; cin>>e1>>e2;
    a[e1][e2]=1;
    a[e2][e1]=1;
}
ns=0;
x[1]=-1;
k=1;

```

```

while (k>0){
    do{
        succ(x,k,as);
        if (as)
            valid(x,k,ev);
    }while (as && !ev);
    if (as)
        if (k==n) afis(x,k);
        else {
            k=k+1;
            x[k]=-1;
        }
    else k=k-1;
}
getche();
}

```

**Problema 10**

Problema se reduce la:

- a ne imagina un graf neorientat,
- nodurile grafului sunt persoanele iar muchiile sunt reprezentarea relațiilor de simpatie între persoane,
- care este reprezentat prin liste de adiacență (citite de la tastatură)
- a construi matricea de adiacență corespunzătoare (plecând de la liste de adiacență);
- a verifica proprietatea de simetrie față de diagonala principală, adică:  $a[i][j]=a[j][i]$ , pentru  $1 \leq i, j \leq n$ .

```

#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int mat[20][20];
int i, j, n, vec;
mat a;
int ok;

main(){
clrscr();
cout<<"n="; cin>>n;
for (i=1;i<=n;i++)
{
    cout<<"Pentru "<<i<<endl;
    cout<<"dati numarul vecinilor";
}

```

se construiește matricea de adiacență plecând de la liste de adiacență

```

    cin>>vec;
    for (k=1;k<=vec;k++)
    {
        cin>>j;
        a[i][j]=1;
    }
}

ok=1;
for (i=1;i<=n-1;i++)
    for (j=i+1;j<=n;j++)
        if (a[i][j]!=a[j][i]){
            ok =0;
            cout<<"nepotrivire intre "<<i<<" si "<<j;
        }
if (ok) cout<<"toate simpatiile sunt reciproce";
getche();

```

se verifică proprietatea de simetrie

### 2.3.9. Parcurgerea grafurilor

#### Problema 1

Listele nodurilor obținute în urma parcurgerii în lățime:

- a) 1, 3, 5, 4, 2;
- b) 1, 2, 3, 4, 6, 5;

Listele nodurilor obținute în urma parcurgerii în adâncime:

- c) 1, 3, 5, 4, 2;
- d) 1, 2, 3, 4, 5, 6;

#### Problema 2

Această problemă este, până la urmă, o problemă de parcursere a unui graf. Pentru a determina nodurile care sunt accesibile din nodul 1, se procedează astfel:

- la început, nici un nod nu se consideră vizitat;
- se vizitează nodul 1;
- se aplică una dintre procedurile de parcursere, de exemplu în adâncime, plecând de la nodul 1;
- se afișează toate nodurile care au fost vizitate, în urma apelului procedurii de care am vorbit (aceste noduri sunt, de fapt, nodurile la care se poate ajunge plecând de la nodul 1).

```

#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int sir[100];
typedef int mat[20][20];

```

```

int i, n, pl, m, x, y;
mat a; sir viz;
void parc_adancime(int pl){
int j;
viz[pl]=1;
for (j=1;j<=n;j++)
    if ((a[pl][j]==1) && (viz[j]==0))
        parc_adancime(j);
}
main(){
    clrscr();
    cout<<"n m "; cin>>n>>m;
    for (i=1;i<=m;i++){
        cout<<"x y "; cin>>x>>y;
        a[x][y]=1; a[y][x]=1;
    }
    for (i=1;i<=n;i++)
        viz[i]=0;
    pl=1;
    parc_adancime(pl);
    cout<<" nodurile care sunt accesibile din nodul 1 sunt ";
    for (i=1;i<=n;i++)
        if (viz[i]==1) cout<<i<<" ";
    getche();
}

```

### 2.3.10. Conexitate

#### Problema 1

- a) L1=[1, 3, 5] lanț;
- L2=[1, 3, 5, 4, 2] lanț de lungime 4;

Oricare dintre lanțurile de mai sus sunt elementare.

C1=[1, 3, 5, 1] ciclu;

Ciclul de mai sus este elementar.

C1=[1, 3, 5, 1] este egal cu C2=[3, 5, 1, 3];

- b) L1=[1, 2, 3, 1, 4] lanț;
- L2=[4, 1, 2, 3, 5] lanț de lungime 4;

Lanțul de mai sus este elementar.

C1=[1, 2, 3, 1] ciclu;

Ciclul de mai sus este elementar.

C1=[1, 2, 3, 4, 1] este egal cu C2=[4, 3, 2, 1, 4];

**Problema 2**

Problema se rezolvă folosind metoda Backtracking, și se reduce la a determina sirurile de forma  $x_1, \dots, x_p$  unde  $x_1=np$  și  $x_p=ns$  (deci, construcția unei soluții se oprește atunci când o componentă a sa primește valoarea ns), cu componente distincte, astfel încât între  $x_i$  și  $x_{i+1}$  să existe muchie în graful dat.

În concluzie, soluția este  $x=(x_1, x_2, \dots, x_p)$ , unde  $x_1=np$ ,  $x_p=ns$ ,  $x_k \in \{1, \dots, n\}$   $k \in \{2, \dots, p-1\}$ ;  $x_i \neq x_j$ , pentru  $i \neq j$ , și  $a[x_{k-1}][x_k]=1$

**◆ Comentarii la funcția Valid:**

Trebuie verificat că:

- există muchie între nodurile  $x_{k-1}$  și  $x_k$ , adică trebuie verificat că  $a[x_{k-1}][x_k]=1$ ;
- nodul  $x_k$  este diferit de toate nodurile prin care s-a trecut, adică:

$$x_k \neq x_i \text{ pentru } i=1 \dots k-1.$$

```
#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int sir[100];
typedef int mat[20][20];
int e1, e2;
int i, j, n, m, ns, np, k;
mat a;
sir x;
int as, ev;

void succ(sir x, int k, int &as)
{
    if (x[k] < n) {
        as = 1;
        x[k] = x[k] + 1;
    }
    else as = 0;
}

void afis(sir x, int k)
{
    int i;
    for (i = 1; i <= k; i++)
        cout << x[i] << " ";
    cout << endl;
}

```

**GRAFURI NEORIENTATE**

```
void valid(sir x, int k, int &ev){
    int i;
    ev = 1;
    if (a[x[k-1]][x[k]] == 0) ev = 0;
    else
        for (i = 1; i <= k - 1; i++)
            if (x[k] == x[i]) ev = 0;
}

main(){
    clrscr();
    cout << "n="; cin >> n;
    cout << "m="; cin >> m;
    for (i = 1; i <= m; i++)
    {
        cout << "e1 e2 "; cin >> e1 >> e2;
        a[e1][e2] = 1;
        a[e2][e1] = 1;
    }
    cout << "np="; cin >> np;
    cout << "ns="; cin >> ns;
    x[1] = np;
    k = 2; x[k] = 0;
    while (k > 1)
    {
        do{
            succ(x, k, as);
            if (as)
                valid(x, k, ev);
        }while (as && !ev);
        if (as) {
            if (k <= n)
                if (x[k] == ns)
                    afis(x, k);
                else{
                    k = k + 1;
                    x[k] = 0;
                }
            else k = k - 1;
        }
        getch();
    }
}

```

trebuie să existe muchie între  
 $x_{k-1}$  și  $x_k$

**Problema 3**

Problema se rezolvă folosind metoda Backtracking, astfel:

- se generează aranjamentele multimi  $\{1, \dots, n\}$  în vectorul  $x = (x_1, x_2, \dots, x_n)$  unde  $n! = n \dots 2$  (deci, se începe cu generarea soluțiilor cu lungimea cea mai mare);
- dintr vectorii generați, se aleg cei care verifică proprietatea: între  $x_i$  și  $x_{i+1}$  există muchie și se afișează cei cu lungimea cea mai mare (acest lucru este ilustrat în funcția Afis).

**◆ Comentarii la funcția Afis:**

- la primul apel al funcției ( $ns=1$ ) se păstrează lungimea vectorului soluție ( $lung=k$ ) pentru că aceasta este cea mai mare;
- la următoarele apeluri nu se ia în calcul decât vectorii cu lungimea egală cu lung.

```
#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int sir[100];
typedef int mat[20][20];
int e1, e2;
int i, j, n, n1, m, k, ns, lung;
mat a;
sir x;
int as, ev;
void succ(sir x, int k, int &as){
    if (x[k] < n) {
        as = 1;
        x[k] = x[k] + 1;
    } else as = 0;
}
void afis(sir x, int k){
    int i;
    ns = ns + 1;
    if (ns == 1) lung = k;
    if (k == lung){
        for (i = 1; i <= k; i++)
            cout << x[i] << " ";
        cout << endl;
    }
}
```

```
void valid(sir x, int k, int &ev)
{
    int i;
    ev = 1;
    if (a[x[k-1]][x[k]] == 0) ev = 0;
    else
        for (i = 1; i <= k - 1; i++)
            if (x[k] == x[i]) ev = 0;
}

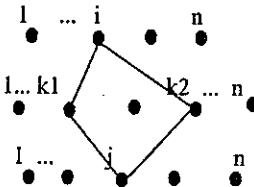
main()
{
    clrscr();
    cout << "n="; cin >> n;
    cout << "m="; cin >> m;
    for (i = 1; i <= m; i++)
    {
        cout << "e1 e2 "; cin >> e1 >> e2;
        a[e1][e2] = 1;
        a[e2][e1] = 1;
    }
    ns = 0;
    for (n1 = n; n1 >= 2; n1--)
    {
        x[1] = 0;
        k = 1;
        while (k > 0)
        {
            do{
                succ(x, k, as);
                if (as)
                    valid(x, k, ev);
            } while (as && !ev);
            if (as)
                if (k == n1) afis(x, k);
                else {
                    k = k + 1;
                    x[k] = 0;
                }
            else k = k - 1;
        }
        getche();
    }
}
```

**Problema 4**

Dacă ar exista un ciclu de lungime 4 ar fi de forma: i, k1, j, k2, i. Pentru a demonstra existența unui astfel de ciclu, procedăm astfel:

- încercăm să arătăm că pentru o pereche de noduri (i, j) există alte două noduri,

k1 și k2 diferite de acestea și legate de ele prin muchii ( $a[i,k1]=1$ ,  $a[j,k1]=1$  și  $a[i,k2]=1$ ,  $a[j,k2]=1$ )



```
ok=0;
for (i=1;i<=n-1;i++)
    for (j=i+1;j<=n;j++)
    {
        nr=0;
        for (k=1;k<=n;k++)
            if ((i!=k) && (j!=k) && (a[i][k]==1) && (a[j][k]==1))
                nr=nr+1;
        if (nr>=2)
            ok=1;
    }
```

**Problema 5**

Problema se rezolvă folosind metoda Backtracking, și se reduce la a determina sirurile de formă  $x_1, \dots, x_p$  unde:

- $x_i=1$ ;
- componente sunt distincte;
- între  $x_i$  și  $x_{i+1}$ ,  $i=1 \dots p-1$ , să existe muchie în graful dat;
- $x_1$  și  $x_p$  sunt unite printr-o muchie.

În concluzie, soluția este  $x=(x_1, x_2, \dots, x_p)$ , unde  $x_i=1$ ,  $x_i \in \{2, \dots, n\}$   $k \in \{2, \dots, p\}$ ;

$x_i \neq x_j$  pentru  $i \neq j$ ,  $a[x_i, x_j]=1$  pentru  $i=2..p$ , și  $a[x_1, x_p]=1$

```
#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int sir[100];
typedef int mat[20][20];
int e1, e2;
```

```
int i, j, n, m, p, k;
mat a;
sir x;
int as, ev;

void succ(sir x, int k, int &as)
```

```
{}
if (x[k]<n) {
    as=1;
    x[k]=x[k]+1;
}
else as=0;
}

void afis(sir x, int k)
{
int i;
for (i=1;i<=k;i++)
    cout<<x[i]<<" ";
cout<<x[1]<<" ";
cout<<endl;
}

void valid(sir x, int k, int &ev)
{
int i;
ev=1;
if (a[x[k-1]][x[k]]==0) ev=0;
else {
    for (i=1;i<=k-1;i++)
        if (x[k]==x[i])
            ev=0;
    if ((k==p) && (a[x[k]][x[1]]==0))
        ev=0;
}}
```

```
main()
{
clrscr();
cout<<" n="; cin>>n;
cout<<"m="; cin>>m;
```

```

for (i=1;i<=m;i++)
{
    cout<<"e1 e2 "; cin>>e1>>e2;
    a[e1][e2]=1;
    a[e2][e1]=1;
}
cout<<"p="; cin>>p;
x[1]=1;
k=2;
x[k]=1;
while (k>1)
{
    do{
        succ(x,k,as);
        if (as)
            valid(x,k,ev);
    } while (as && !ev);
    if (as)
        if (k==p)
            afis(x,k);
        else {
            k=k+1;
            x[k]=1;
        }
    else k=k-1;
}
getche();
}

```

**Problema 6**

Până la urmă, este o problemă de parcurgere a unui graf și se rezolvă astfel:

- se citește matricea de adiacență;
- **toate nodurile se consideră inițial nevizitate;**
- la început, sunt 0 componente conexe;
- se parcurg nodurile grafului
  - dacă se găsește un nod nevizitat încă
    - înseamnă că s-a mai găsit o componentă conexă
    - și se vizitează toate nodurile la care se poate ajunge plecând de la acest nod (pentru aceasta este nevoie de una dintre procedurile de parcurgere a unui graf (în adâncime sau în lățime))

```

#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int sir[100];
typedef int mat[20][20];
mat a;
sir viz;
int i, n, m, x, y, nr,j;
void parc_adancime(int pl)
{
    int j;
    viz[pl]=1;
    for (j=1;j<=n;j++)
        if ((a[pl][j]==1) && (viz[j]==0))
            parc_adancime(j);
}
main()
{
clrscr();
cout<<"n="; cin>>n;
cout<<"m="; cin>>m;
for (i=1;i<=m;i++)
{
    cout<<"x y"; cin>>x>>y;
    a[x][y]=1;
    a[y][x]=1;
}
for (i=1;i<=n;i++)
    viz[i]=0;
nr=0;
for (i=1;i<=n;i++)
    if (viz[i]==0) {
        nr=nr+1;
        parc_adancime(i);
    }
cout<<"are "<<nr<<" componente conexe";
getche();
}

```

**Problema 7**

Se procedează astfel:

- se citește matricea de adjacență;
  - **toate nodurile** se consideră inițial nevizitate;
  - se parcurge nodurile grafului
    - dacă se găsește un nod nevizitat încă,
    - se vizitează și se afișează;
    - se vizitează și se afișează toate nodurile la care se poate ajunge plecând de la acest nod
- (acest lucru se realizează printr-un apel al procedurii **parc\_adancime**).

```
#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int sir[100];
typedef int mat[20][20];
mat a;
sir viz;
int i, n, m, x, y;
int nr;

void parc_adancime(int pl)
{
    int j;
    viz[pl]=1; cout<<pl<<" ";
    for (j=1;j<=n;j++)
        if((a[pl][j]==1) && (viz[j]==0))
            parc_adancime(j);
}

main()
{
    clrscr();
    cout<<"n="; cin>>n;
    cout<<"m="; cin>>m;
    for (i=1;i<=m;i++)
    {
        cout<<" x y"; cin>>x>>y;
        a[x][y]=1;
        a[y][x]=1;
    }
}
```

```
for (i=1;i<=n;i++)
    viz[i]=0;
nr=0;
for (i=1;i<=n;i++)
    if (viz[i]==0) {
        nr=nr+1;
        cout<< " componenta conexă cu numarul "<<nr << " are
nodurile"<<endl;
        parc_adancime(i);
        cout<<endl<<endl;
    }
getche();
}
```

**Problema 8**

Problema se reduce, până la urmă, la a afișa toate perechile de forma (i1, i2), unde i1 și i2 sunt noduri din aceeași componentă conexă. Pentru aceasta:

- se generează **toate componentele conexe**;
- pentru fiecare componentă conexă generată, se afișează **toate perechile ordonate de noduri** (i1,i2), care se pot forma cu elementele sale.

**Observație.** Pentru a nu încurca nodurile care fac parte din componente conexe diferite, astfel încât să afișăm de două ori aceleași noduri, procedăm astfel:

- după ce se afișează perechile formate din nodurile unei componente conexe, se măresc cu 1 toate elementele din vectorul **viz**, care corespund nodurilor componentei afișate, pentru a deveni 2, astfel încât să nu fie confundate cu nodurile componentei conexe care se va determina, pentru care vectorul **viz** va avea valoarea 1.

```
#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int sir[100];
typedef int mat[20][20];
mat a;
sir viz;
int i, n, m, x, y, i1, i2;

void parc_adancime(int pl)
{
    int j;
    viz[pl]=1;
    for (j=1;j<=n;j++)
        if((a[pl][j]==1) && (viz[j]==0))
            parc_adancime(j);
}
```

```

        if ((a[pl][j]==1) && (viz[j]==0))
            parc_adancime(j);
    }

main()
{
    clrscr();
    cout<<"n="; cin>>n;
    cout<<"m="; cin>>m;
    for (i=1;i<=m;i++)
    {
        cout<<"x y"; cin>>x>>y;
        a[x][y]=1;
        a[y][x]=1;
    }
    for (i=1;i<=n;i++)
        viz[i]=0;
    for (i=1;i<=n;i++)
        if (viz[i]==0)
    {
        parc_adancime(i);
        for (i1=1;i1<=n-1;i1++)
            for (i2=i1+1;i2<=n;i2++)
                if ((viz[i1]==1) && (viz[i2]==1))
                    cout<<"(" <<i1 << ", " <<i2 << ")";
        for (i1=1;i1<=n;i1++)
            if (viz[i1]==1) viz[i1]=viz[i1]+1;
        cout<<endl<<endl;
    }
    getche();
}

```

**Problema 9**

Sistemului de culoare i se pune în corespondență un graf neorientat, astfel:

- nodurile grafului sunt culoarele;
- muchiile grafului sunt legăturile între culoare.

A verifică dacă toate culoarele fac parte din aceeași peșteră, înseamnă a verifica dacă graful asociat este conex (adică este format dintr-o singură componentă conexă):

- la început, **nodurile sunt nevizitate**;
- se aplică funcția de parcurgere a grafului, plecând de la un nod oarecare;
- dacă, după parcurgere, în graf mai sunt noduri nevizitate, nu este conex;

Vezi problema rezolvată din secțiunea 2.1.10

```

cout<<"dati nodul de plecare : "; cin>>pl;
parc_adancime(pl);
ok=0;
for (i=1;i<=n;i++)
    if (viz[i]==0) ok=1;
if (ok) cout<<"sunt mai multe peșteri";
else cout<<"este o singura peștera";

```

**Problema 10**

Grupului de persoane, despre care se vorbește în enunțul problemei, i se pune în corespondență un graf neorientat, astfel:

- nodurile grafului sunt persoanele;
- muchiile grafului sunt precizările idei că două persoane sunt prietene (în mod direct).

A determină grupurile de persoane; cu un număr maxim de membri, între care se pot stabili prietenii, direkte sau indirekte, se reduce la a determina componentele conexe cu cele mai multe noduri (este o problemă de maxim). Acest lucru se realizează astfel:

- pe măsură ce se determină componenta conexă, cu numărul nr,
- se determină numărul de noduri ale sale în comp[nr]
- și i se păstrează nodurile în mulțimea varfuri[nr]
- dacă comp[nr] > max (max – cel mai mare număr înregistrat până în prezent) devine max
- se afișează nodurile tuturor componentelor conexe care au max noduri.

**Atenție!** Funcția **parc\_adancime** are un parametru în plus (nr), pe care îl folosește în scopul de a diferenția elementele diferitelor componente conexe, astfel:

- la fiecare apel, elementele care au fost vizitate sunt marcate cu nr (viz[i]=nr).

```

#include <conio.h>
#include <iostream.h>
#include <stdio.h>
typedef int sir[100];
typedef int mat[20][20];
mat a;
sir viz, comp;
int i, n, m, x, y, max, j, nr;
mat varfuri;

```

```

void parc_adancime(int pl, int nr){
    int j;
    viz[pl]=nr;
    for (j=1;j<=n;j++)
        if ((a[pl][j]==1) && (viz[j]==0))
            parc_adancime(j,nr);
}
main(){
    clrscr();
    cout<<"n="; cin>>n;
    cout<<"m="; cin>>m;
    for (i=1;i<=m;i++){
        cout<<"x y"; cin>>x>>y;
        a[x][y]=1;
        a[y][x]=1;
    }
    for (i=1;i<=n;i++)
        viz[i]=0;
    max=0; nr=0;
    for (i=1;i<=n;i++)
        if (viz[i]==0) {
            nr=nr+1;
            parc_adancime(i,nr);
            comp[nr]=0;
            for (j=1;j<=n;j++)
                if (viz[j]==nr){
                    comp[nr]=comp[nr]+1;
                    varfuri[nr][comp[nr]]=j;
                }
            if (comp[nr]>max)
                max=comp[nr];
        }
    for (i=1;i<=nr;i++)
        if (comp[i]==max){
            cout<<i<<"....";
            for (j=1;j<=comp[i];j++)
                cout<<varfuri[i][j]<<" ";
            cout<<" ";
        }
}

```

### 2.3.11. Cicluri Hamiltoniene

#### Problema 1

$L=[1, 3, 5, 4, 2]$  : este lanț hamiltonian;

$C=[5, 4, 2, 3, 1, 5]$  : este ciclu hamiltonian;

#### Problema 2

Dacă  $n \geq 3$  și  $d(x) \geq n/2$  pentru orice nod  $x$ , atunci graful este hamiltonian.

.....

int grad(int i)

{

    int s, j;

    s=0;

    for (j=1;j<=n;j++)

        s=s+a[i][j];

    return s;

}

.....

ok1=(n>=3);

ok2=1;

for (i=1;i<=n;i++)

    if (! (grad(i)>=n/2))

        ok2=0;

ok=ok1 && ok2;

if (ok) cout<<"se verifică condițiile teoremei";

else cout<<"nu se verifică condițiile teoremei";

#### Problema 3

Demonstrăm acest lucru folosind metoda inducției matematice. Pentru a ușura scrierea, vom nota cu  $H(n)$  numărul ciclurilor hamiltoniene ale grafului  $K_n$ ; în baza acestei notări, enunțul problemei cere să demonstrează că:  $H(n) = \frac{(n-1)!}{2} \quad \forall n \geq 3$

##### I. Verificare:

Pentru  $n=3$   $H(3)=H(3)=\frac{(3-1)!}{2}=\frac{2!}{2}=1$  (adevărat, pentru că într-un graf complet cu trei vârfuri este un singur ciclu hamiltonian)

##### II. Demonstrația $P(n) \Rightarrow P(n+1)$

$$P(n) : \quad H(n) = \frac{(n-1)!}{2}$$

$$P(n+1) : \quad H(n+1) = \frac{(n+1-1)!}{2}$$

Altfel spus, dacă se știe că numărul ciclurilor hamiltoniene în graful  $K_n$  este  $(n-1)!/2$ , să se demonstreze că numărul ciclurilor hamiltoniene în  $K_{n+1}$  este  $n!/2$ .

Având la bază faptul că din fiecare ciclu hamiltonian din  $K_n$  se pot obține  $n$  cicluri hamiltoniene distincte în  $K_{n+1}$ , prin intercalarea nodului  $n+1$  în toate cele  $n$  locuri posibile (între două noduri succesive; ex:  $x_1 \ x_2 \ \dots \ x_n \ x_1$  (între  $x_1 \ x_2, x_2 \ x_3, \dots$ )), putem spune că în  $K_{n+1}$  sunt  $n$  ori numărul ciclurilor din  $K_n$  cicluri hamiltoniene, ceea ce ne permite să scriem:

$$H(n+1) = n \cdot H(n) = n \cdot \frac{(n-1)!}{2} = \frac{n!}{2} = \frac{(n+1-1)!}{2}$$

#### Problema 4

Având la bază:

- 1) fiecare ciclu elementar este un ciclu hamiltonian în subgraful complet induș de vârfurile sale și invers, fiecare ciclu hamiltonian dintr-un subgraf cu  $k$  vârfuri este ciclu elementar în  $G$ .
- 2) într-un graf complet cu  $k$  vârfuri sunt  $\frac{(k-1)!}{2}$  cicluri hamiltoniene;
- 3) într-un graf cu  $n$  vârfuri pot exista  $C_n^k$  subgrafuri cu  $k$  noduri;

tragem concluzia:

**Intr-un graf cu  $n$  vârfuri, numărul ciclurilor elementare este egal cu:**

$$\begin{aligned} C_n^3 \cdot \frac{(3-1)!}{2} + C_n^4 \cdot \frac{(4-1)!}{2} + \dots + C_n^n \cdot \frac{(n-1)!}{2} &= \sum_{k=3}^n C_n^k \cdot \frac{(k-1)!}{2} = \sum_{k=3}^n \frac{n!}{k!(n-k)!} \cdot \frac{(k-1)!}{2} = \\ &= \sum_{k=3}^n \frac{n!}{k!(n-k)!} \cdot \frac{(k-1)!}{2} = \sum_{k=3}^n \frac{n!}{(k-1)!k \cdot (n-k)!} \cdot \frac{(k-1)!}{2} = \frac{1}{2} \sum_{k=3}^n \frac{n(n-1)\dots(n-k+1)}{k} \end{aligned}$$

#### Problema 5

Problema se face la fel ca și problema determinării ciclurilor hamiltoniene, pe care am prezentat-o în secțiunea 2.1.11., numai că, în situația de față, va trebui ca în loc să se afișeze ciclurile să se dea răspunsul da sau nu. Acest lucru se poate face astfel:

Se folosește o variabilă booleană `ok`, care la începutul programului are valoarea false (se presupune că nu există cicluri), iar în procedura de afișare să primească valoarea true (adică s-a găsit un ciclu hamiltonian; eventual, la primul apel al procedurii `Afis` să se renunțe la generarea ciclurilor care ar mai putea fi găsite) și să se afișeze înainte de terminarea programului.

**Altfel:**

Se procedează la fel ca la problema 2 (adică, se verifică dacă sunt satisfăcute condițiile teoremei prezentată în secțiunea 2.1.11.).

#### Problema 6

Asociem grupului de persoane, despre care se vorbește în enunțul problemei, un graf neorientat definit astfel: nodurile grafului reprezintă persoanele iar muchia între nodurile  $i$  și  $j$  reprezintă precizarea ideii că persoanele  $i$  și  $j$  sunt prietene, adică în relație de nedușmănie. Din ipoteza că fiecare cavaler are cel mult  $(n-1)$  dușmani, rezultă că pentru fiecare cavaler există cel puțin  $2n-(n-1)=(n+1)\geq 2n/2$  cavaleri cu care acesta se află în relație de nedușmănie, adică de prietenie. Înținând cont de cele spuse mai sus, putem trage concluzia că în graful asociat grupului de persoane sunt verificate condițiile teoremei prezentată în secțiunea 2.1.11. conform căreia în graful respectiv există cel puțin un ciclu hamiltonian. Ciclului hamiltonian îi asociem o așezare a cavalerilor la masa rotundă în condițiile cerute.

## ARBORI

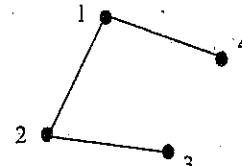
### 3.1. ASPECTE TEORETICE

#### 3.1.1. Noțiunea de arbore

**Definiție.** Se numește arbore un graf conex și fără cicluri.

◆ **Exemplu de arbore:**

Graful  $G=(V, M)$  unde  $V=\{1,2,3,4\}$  și  $M=\{\{1,2\}, [2,3], [1,4]\}$ , a cărui reprezentare grafică este figurată mai jos, este arbore.



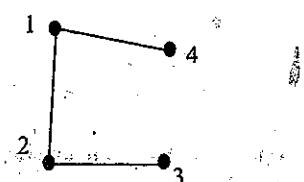
**Demonstrație:**

Graful prezentat mai sus este arbore, deoarece este conex și fără cicluri.

**Teoremă.** Fie  $G=(V,M)$  un graf. Următoarele afirmații sunt echivalente:

- $G$  este arbore.
- $G$  este un graf conex, minimal cu această proprietate (dacă se elimină orice muchie, se obține un graf neconex).
- $G$  este fără cicluri, maximal cu această proprietate (dacă se adaugă o muchie, se obține un graf care are cel puțin un ciclu).

◆ **Exemplu:** Pentru graful din figura de mai jos, se verifică foarte ușor echivalența afirmațiilor din teorema.



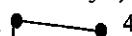
**Demonstrație:**

Graful este arbore (conex și fără cicluri).

Graful este conex, minimal cu această proprietate, deoarece orice muchie s-ar elimina graful nu ar mai fi conex (urmăriți figurile de mai jos)



s-a eliminat muchia [1,2]

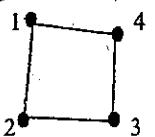


s-a eliminat muchia [2,3]

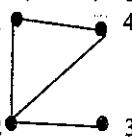


s-a eliminat muchia [1,4]

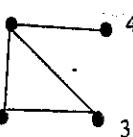
Graful este fără cicluri, maximal cu această proprietate, deoarece orice muchie s-ar adăuga graful ar conține cel puțin un ciclu (urmăriți figurile de mai jos)



s-a adăugat muchia [4,3]



s-a adăugat muchia [2,4]



s-a adăugat muchia [1,3]

**Teoremă.** Orice arbore, cu  $n \geq 2$  vârfuri, conține cel puțin două vârfuri terminale.

#### Demonstrație:

Presupunem că există un arbore A, cu  $n \geq 2$  vârfuri, care conține un singur vârf terminal; fie acesta  $x_1$ .

Alegem în A cel mai lung lanț elementar,adică lanțul care conține numărul de muchii maxim posibil, care-l admite pe  $x_1$  ca extremitate; fie acesta  $L = [x_1, x_2, \dots, x_{p-1}, x_p]$ . Deoarece  $x_1$  este singurul vârf terminal, înseamnă că  $x_p$  are gradul  $\geq 2$ , deci pe lângă nodul  $x_{p-1}$  mai este legat de cel puțin un alt nod, fie acesta y. Există două situații:

- y nu aparține lanțului L, și atunci lanțul nu ar fi cel mai mare deoarece s-ar mai putea prelungi cu muchia  $[x_p, y]$ . Contradicție, deoarece am presupus că L este cel mai lung.
- y aparține lanțului L, și atunci am putea scoate în evidență ciclul  $[x_p, y, \dots, x_{p-1}, x_p]$ .

Contradicție, deoarece A este arbore și, deci, nu conține cicluri.

Pentru a ușura înțelegerea, urmăriți figurile:



Contradicția provine din faptul că am presupus că A conține un singur vârf terminal. În concluzie, A are cel puțin două vârfuri terminale.

**Teoremă.** Orice arbore cu n vârfuri are  $n-1$  muchii.

#### Demonstrație:

Demonstrația teoremei se realizează recurgând la principiul inducției matematice.

Fie A un arbore cu n vârfuri și A(n) numărul muchiilor acestuia. Enunțul teoremei cere să se demonstreze că  $A(n)=n-1$ .

#### I. Verificare

Dacă  $n=1 \Rightarrow A(n)=A(1)=1-1=0$  Adevarat, deoarece arborele fiind format dintr-un singur nod nu are nici o muchie.

#### II. Demonstrația $P(n) \Rightarrow P(n+1)$

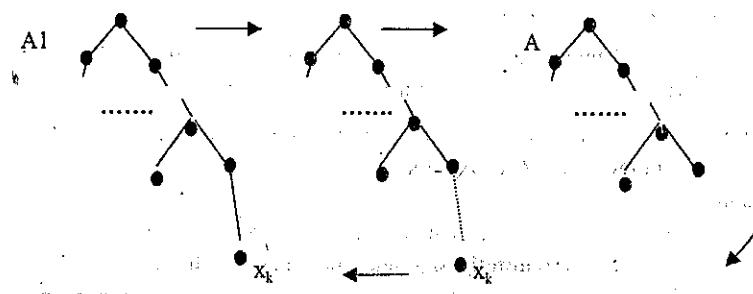
$P(n) : A(n)=n-1$

$P(n+1) : A(n+1)=(n+1)-1$

Cu alte cuvinte, dacă se știe că un arbore cu n vârfuri are  $n-1$  muchii, trebuie să se demonstreze că un arbore cu  $n+1$  vârfuri are  $n$  muchii.

Fie A1 arborele cu  $n+1$  vârfuri, și  $x_k$  un nod terminal al său. Dacă se elimină nodul  $x_k$  și muchia incidentă cu acesta se obține un arbore A cu n vârfuri care are, conform ipotezei,  $n-1$  muchii.

Pentru a ușura înțelegerea urmăriți figurile în sensul dat de săgeți:



Tinând cont de faptul că A1 se obține adăugând la A nodul și muchia eliminată tragem concluzia:

Arboarele A1, cu  $n+1$  noduri, are cu 1 mai multe muchii decât A, care are  $n-1$  muchii, deci A1 are  $1+(n-1)$  muchii, adică  $n$  muchii.

**Definiție.** Fie  $G=(V,M)$  un graf. Graful G se numește aciclic dacă nu conține cicluri.

**Definiție.** Fie  $G=(V,M)$  un graf. Graful G se numește pădure dacă nu conține cicluri.

**În continuare, prezentăm algoritmul prin care se verifică dacă un graf conține sau nu cicluri.**

Ideea generală a algoritmului este:

Se încearcă reconstituirea multimii nodurilor, plecând de la extremitățile primei muchii și adăugând noi vârfuri adiacente cu vârfuri adăugate deja. Dacă în tentativa de căutare a unor noi noduri se găsește un nod din multime care este adiacent cu un alt nod adăugat deja la multime, se deduce că a fost găsit un ciclu și algoritmul se oprește, altfel dacă nu se găsește nici un nod adiacent cu vreunul din multime se adaugă la multime extremitățile unei muchii neutilizate încă și se reia căutarea.

Elementele principale, cu care se lucrează, sunt:

- $u_1, u_2, \dots, u_m$  : sirul muchiilor;
- $vf\_luate$  : multime în care se adaugă pe rând nodurile ( $vf\_luate =$  vârfuri luate);
- $m\_neluate$  : multimea indicilor muchiilor ale căror extremități nu au fost introduse încă în multimea  $vf\_luate$  ( $m\_neluate =$  muchii neluate).

Descrierea detaliată, în cuvinte, a algoritmului:

Pas1. Se introduc în  $vf\_luate$  extremitățile primei muchii, adică se introduc  $u_1, x$  și  $u_1, y$ , iar în  $m\_neluate$  se introduc indicii muchiilor rămase, adică  $2, \dots, m$ ;

Pas2. Printre muchiile neluate încă, se caută o muchie care are o extremitate în  $vf\_luate$ .

Dacă nu se găsește o astfel de muchie,

atunci

se alege o muchie la întâmplare care nu a fost luată încă, și extremitățile sale sunt adăugate la  $vf\_luate$ , iar indicele său, de exemplu  $j$ , este eliminat din  $m\_neluate$ ; se reia pasul 2

altfel

dacă ambele extremități, ale muchiei găsite, fac parte din  $vf\_luate$ , atunci

se trage concluzia: **a fost găsit un ciclu** și se renunță la algoritru

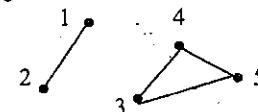
altfel

se introduce în  $vf\_luate$  cealaltă extremitate a muchiei; se elimină din  $m\_neluate$  indicele său (al muchiei găsite); se reia pasul 2;

pas 3. Dacă se ajunge la acest pas, adică multimea muchiilor a fost epuizată, se afișează mesajul: **Graful nu conține cicluri**

♦ Exemplu: Fiind dat graful  $G=(V,M)$  unde  $V=\{1,2,3,4,5\}$  și  $M=\{u_1, u_2, u_3, u_4\}=\{[1,2], [3,4], [3,5], [4,5]\}$

cu reprezentarea grafică:



să se verifice dacă are sau nu cicluri, aplicând algoritmul prezentat mai sus.

Demonstrație:

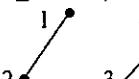
Pas1.  $vf\_luate=\{1, 2\}$  extremitățile primei muchii,  $u_1$ , indicii muchiilor neluate în calcul, până acum.



Pas2. Printre indicii 2, 3, 4, ai muchiilor neluate în calcul, se caută o muchie care are o extremitate în  $vf\_luate=\{1, 2\}$ .

Deoarece această muchie nu se găsește,

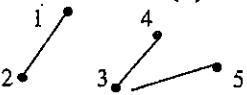
se alege la întâmplare o muchie, de exemplu  $u_2$ , și extremitățile sale, 3 și 4, se adaugă la  $vf\_luate$ , deci  $vf\_luate=\{1, 2, 3, 4\}$ , iar indicele său, 2, este eliminat din  $m\_neluate$ , adică  $m\_neluate=\{3, 4\}$ .



Pas2'. Printre indicii 3, 4, ai muchiilor neluate în calcul, se caută o muchie care are o extremitate în  $vf\_luate=\{1, 2, 3, 4\}$ .

Se găsește, de exemplu,  $u_3=[3, 5]$ ,

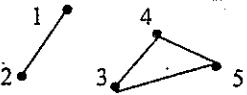
și extremitatea sa 5, care nu se găsește în  $vf\_luate$ , se adaugă la  $vf\_luate$ , deci  $vf\_luate=\{1, 2, 3, 4, 5\}$ , iar indicele său, 3, este eliminat din  $m\_neluate$ , adică  $m\_neluate=\{4\}$ .



Pas2''. Printre indicii, ... 4, ai muchiilor neluate în calcul, se caută o muchie care are o extremitate în  $vf\_luate=\{1, 2, 3, 4, 5\}$ .

Se găsește, de exemplu,  $u_4=[4, 5]$ ,

care, deoarece are ambele extremități în multimea  $vf\_luate$ , înseamnă că s-a găsit un ciclu ( $C=[4, 3, 5, 4]$ ). Algoritmul se oprește aici.



În concluzie, graful dat nu este aciclic (el conține un ciclu).

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

typedef int mat[20][20];
typedef struct{
    int x,y;
}muchie;
typedef muchie sir[30];
typedef int vec[30];

int i, j, n, m, x, y, aciclic, gasit, nm,gasitj, gasitx, gasity, i0;
sir u;
vec vf_luate, m_neluate;

main(){clrscr();
    cout<<"n="; cin>>n;
    cout<<"m="; cin>>m;
    for (i=1;i<=m;i++){
        cout<<i<<" x y ";
        cin>>x>>y;
        u[i].x=x;
        u[i].y=y;
    }
    vf_luate[1]=u[1].x;
    vf_luate[2]=u[1].y;
    nvf=2;
    nm=0;
    for (j=2;j<=m;j++){
        nm=nm+1;
        m_neluate[nm]=j;
    }
    aciclic=1;
    while (aciclic && (nm!=0))
    {
        gasit=0;
        for (j=2;j<=m;j++)
        {
            gasitj=0;
            for (i=1;i<=nm;i++)
            {
                if (m_neluate[i]==j) { gasitj=1;i0=i; }
                if (gasitj){}
                    x=u[j].x; y=u[j].y;
                    gasitx=0;
                    for (i=1;i<=nvf;i++)
                        if (vf_luate[i]==x) gasitx=1;
                    gasity=0;
                    for (i=1;i<=nvf;i++)
                        if (vf_luate[i]==y) gasity=1;
                    if (gasitx || gasity){
                        gasit=1;
                        break;
                    }
            }
            if (!gasit){
                for (j=2;j<=mj;j++)
                {
                    gasitj=0;
                    for (i=1;i<=nm;i++)
                        if (m_neluate[i]==j) { gasitj=1;i0=i; }
                    if (gasitj){
                        nvf++;
                        vf_luate[nvf]=u[j].x;
                        nvf++;
                        vf_luate[nvf]=u[j].y;
                        for (i=i0;i<=nm-1;i++)
                            m_neluate[i]=m_neluate[i+1];
                        nm=nm-1;
                        break;
                    }
                }
            }
            else
            {
                if (gasitx && gasity) aciclic=0;
                else{
                    if (gasitx)
                        nvf++;
                    vf_luate[nvf]=y;
                }
            }
        }
    }
}
```

```
for (i=1;i<=nm;i++)
    if (m_neluate[i]==j) { gasitj=1;i0=i; }
if (gasitj){
    x=u[j].x; y=u[j].y;
    gasitx=0;
    for (i=1;i<=nvf;i++)
        if (vf_luate[i]==x) gasitx=1;
    gasity=0;
    for (i=1;i<=nvf;i++)
        if (vf_luate[i]==y) gasity=1;
    if (gasitx || gasity){
        gasit=1;
        break;
    }
}
if (!gasit){
    for (j=2;j<=mj;j++)
    {
        gasitj=0;
        for (i=1;i<=nm;i++)
            if (m_neluate[i]==j) { gasitj=1;i0=i; }
        if (gasitj){
            nvf++;
            vf_luate[nvf]=u[j].x;
            nvf++;
            vf_luate[nvf]=u[j].y;
            for (i=i0;i<=nm-1;i++)
                m_neluate[i]=m_neluate[i+1];
            nm=nm-1;
            break;
        }
    }
}
else
{
    if (gasitx && gasity) aciclic=0;
    else{
        if (gasitx)
            nvf++;
        vf_luate[nvf]=y;
    }
}
else{
    nvf++;
}
```

```

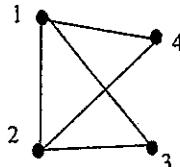
    vf_luate[nvf]=x;
}
for (i=i0;i<=nm-1;i++)
    m_neluate[i]=m_neluate[i+1];
nm=nm-1;
}
if (aciclic) cout<<"este aciclic";
else cout<<"nu este aciclic";
getche();
}

```

### 3.1.2. Arborele parțial de cost minim

**Definiție.** Fie  $G=(V,M)$  un graf. Se numește arbore parțial al lui  $G$ , un graf parțial al său, care, în plus, este și arbore.

◆ **Exemplu:** Dacă din graful:



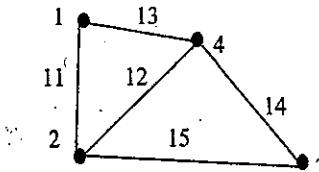
se elimină muchiile  $[2,4]$  și  $[1,3]$ , se obține un graf parțial, care este și arbore (conex și fără cicluri), deci se obține un arbore parțial.

**Propozitie.** Fie  $G=(V,M)$  un graf. Graful  $G$  conține un arbore parțial dacă și numai dacă este conex.

Fie graful  $G=(V, M)$ , conex, și funcția  $\text{cost} : M \rightarrow \mathbb{R}_+$ , atașată acestui graf, care asociază fiecărei muchii  $m$  numărul real și pozitiv  $\text{cost}(m)$ .

**Definiție.** Fie  $G=(V, M)$  un graf conex și  $H=(V, M_1)$  un arbore parțial al său. Definim costul arborelui parțial  $H$  ca fiind suma costurilor muchiilor sale.

◆ **Exemplu:** Fie graful din figura de mai jos (costul fiecărei muchii fiind scris pe ea).



Considerând arborele parțial al său,  $H$ , care se obține eliminând muchiile  $[2,4]$  și  $[3,4]$ , putem calcula costul său, conform definiției de mai sus, astfel:

$$\text{cost}(H) = \text{cost}([1,2]) + \text{cost}([1,4]) + \text{cost}([2,3]) = 11 + 13 + 15 = 39$$

Una dintre cele mai importante probleme care se pun, când este vorba de arborele parțial de cost minim, este:

Fiind dat graful  $G$ , să se determine un arbore parțial de cost minim al său.

Un corespondent practic, al problemei de mai sus, ar fi următoarea problemă:

Fiind date  $n$  orașe și costul conectării anumitor perechi de orașe, să se aleagă acele muchii care asigură existența unui drum între oricare două orașe astfel încât costul total să fie minim.

În continuare, prezentăm algoritmul prin care se determină un arbore parțial de cost minim al unui graf.

Elementele principale, cu care se lucrază, sunt:

– vectorul  $Ap$ , cu atâta componente câte noduri are graful, cu următoarea semnificație:

$Ap[i]$  : reține 0, dacă nodul  $i$  aparține arborelui deja construit;  
reține  $j$ , dacă nodul  $i$  nu aparține arborelui deja construit și muchia de cost minim care unește  $p$  și  $i$  cu unul din nodurile arborelui deja construit este  $[i,j]$

– vectorul  $Tata$ , cu atâta componente câte noduri are graful, cu următoarea semnificație:

$Tata[i]$  : reține părintele nodului  $i$ , din arborele deja construit;

Descrierea detaliată, în cuvinte, a algoritmului:

Pas1. Se citește numărul nodurilor  $n$ , matricea costurilor  $\text{cost}$  și nodul de plecare  $pl$ .

Se completează elementele vectorului  $Ap$ , astfel:  $Ap[pl]=0$  și

$$Ap[i]=pl, \text{ pentru } i=1..n, i \neq pl$$

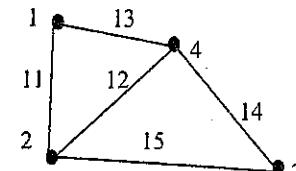
Pas2. Se alege muchia de cost minim  $[i,j]$  (dintre cele punctate) care are o extremitate într-unul din nodurile arborelui deja construit, iar cealaltă extremitate într-un nod care nu aparține arborelui (altfel spus,  $Ap[i]=0$  și  $Ap[j]>0$ );

Se actualizează vectorul  $Tata$ , pentru nodul  $j$ , astfel:  $Tata[j]=Ap[j]$ ;

Se actualizează vectorul  $Ap$ , pentru nodul  $j$ .

pas 3. Dacă nu au fost alese  $n-1$  muchii, se revine la pasul 2

◆ **Exemplu:** Fie graful din figura de mai jos (costul fiecărei muchii fiind scris pe ea).



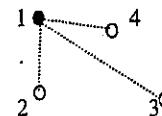
**Pas1.** Se pleacă de la nodul 1, deci  $pl=1$ ;

Tata[1]=0

Costul=0 (costul total al arborelui deja construit)

Ap[1]=0 (pentru că nodul 1 este adăugat la arbore)

Deci Ap=(Ap[1], Ap[2], Ap[3], Ap[4])=(0, 1, 1, 1)



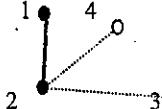
**Pas2.** Se alege muchia de cost minim, care are o extremitate în nodurile alese deja, {1}, și cealaltă într-un nod neales încă, {2, 3, 4}; se găsește [1,2];

Tata[2]=1

Costul=Costul+cost[1,2]=0+11=11

Ap[2]=0 (deoarece nodul 2 este ales)

Deci Ap=(Ap[1], Ap[2], Ap[3], Ap[4])=(0, 0, 2, 2)



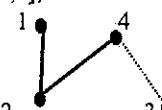
**Pas2.** Se alege muchia de cost minim, care are o extremitate în nodurile alese deja, {1, 2}, și cealaltă într-un nod neales încă, {3, 4}; se găsește [2,4];

Tata[4]=2

Costul=Costul+cost[2,4]=11+12=23

Ap[4]=0 (deoarece nodul 4 este ales)

Deci Ap=(Ap[1], Ap[2], Ap[3], Ap[4])=(0, 0, 4, 0)



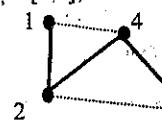
**Pas2.** Se alege muchia de cost minim, care are o extremitate în nodurile alese deja, {1,2,4}, și cealaltă într-un nod neales încă, {3}; se găsește [4,3];

Tata[3]=4

Costul=Costul+cost[4,3]=23+14=37

Ap[3]=0 (deoarece nodul 3 este ales)

Deci Ap=(Ap[1], Ap[2], Ap[3], Ap[4])=(0, 0, 0, 0)



Final

În final, avem: Tata=(Tata[1], Tata[2], Tata[3], Tata[4])=(0,1,4,2)

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

```
#include <conio.h>
```

```
#include <stdio.h>
```

```
typedef int mat[20][20];
```

```
typedef int sir[20];
```

```
mat cost;
```

```
sir ap, tata;
```

```
int i, k, j, n, pl;
```

```
double min, costul;
```

```
main()
{
    clrscr();
    cout<<"n="; cin>>n;
    for (i=1;i<=n-1;i++)
        for (j=i+1;j<=n;j++)
    {
        cout<<"cost["<<i<<","<<j<<"]=";
        cin>>cost[i][j];
        cost[j][i]=cost[i][j];
    }
}
```

```
costul=0;
cout<<"nodul de plecare : "; cin>>pl;
ap[pl]=0;
```

```
for (i=1;i<=n;i++)
    if (i!=pl) ap[i]=pl;
```

```
for (k=1;k<=n-1;k++)
    {

```

```
        min=100000;
        for (i=1;i<=n;i++)
            if (ap[i]!=0)

```

```
                if (cost[i][ap[i]]<min)

```

```
                    {
                        min=cost[i][ap[i]];
                        j=i;
                    }

```

```
                    tata[j]=ap[j];

```

```
                    costul=costul+cost[ap[i]][i];

```

```
                    ap[j]=0;

```

```
                    for (i=1;i<=n;i++)

```

```
                        if ((ap[i]!=0) && (cost[i][ap[i]]>cost[i][j]))

```

```
                            ap[i]=j;

```

```
                    }

```

```
                    cout<<"costul este : "<<costul<<endl;

```

```
                    for (i=1;i<=n;i++)

```

```
                        cout<<tata[i]<<" ";

```

```
                    getche();
                }

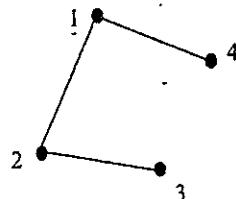
```

### 3.1.3. Arbori binari

**Definiție.** Se numește arborescență un arbore caracterizat astfel:

- are un vârf special numit rădăcină;
- celelalte noduri pot fi grupate în  $p \geq 0$  mulțimi disjuncte,  $V_1, V_2, \dots, V_p$  astfel încât fiecare dintre aceste mulțimi să conțină un nod adjacents cu rădăcina iar subgrafurile generate de acestea să fie la rândul lor arborescente.

◆ **Exemplu.** Graful din figura de mai jos este o arborescență.



**Demonstrație:**

Graful este arbore. Conține un nod special numit rădăcină, fie acesta nodul 1.

Celelalte noduri se pot împărți în mulțimile disjuncte  $V_1 = \{2, 3\}$  și  $V_2 = \{4\}$  astfel încât în mulțimea  $V_1$  să existe nodul 2 adjacents cu rădăcina, iar în mulțimea  $V_2$  să existe nodul 4 adjacents cu rădăcina, și subgrafurile generate de  $V_1$  și  $V_2$  să fie la rândul lor arborescente.

**Observații.** 1. Dacă o arborescență este formată dintr-un singur nod spunem că este formată doar din nodul rădăcină.

2. Dacă ordinea relativă a arborescențelor, generate de  $V_1, V_2, \dots, V_p$  din definiție, are importanță, arborescența se numește arbore ordonat.

În reprezentarea grafică a unei arborescențe, nodurile se desenează pe nivele, astfel:

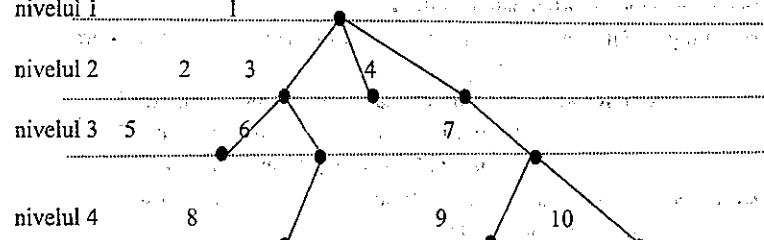
pe primul nivel : se desenează rădăcina;

pe al doilea nivel : se desenează nodurile adjacente cu rădăcina;

pe al treilea nivel : se desenează nodurile adjacente cu noduri de pe nivelul al doilea, și nu se află pe nivelul 1;

pe al k-lea nivel : se desenează nodurile adjacente cu noduri de pe nivelul k-1, și nu se află pe nivelul k-2;

◆ **Exemplu** de arborescență reprezentată grafic, conform regulii prezentate mai sus:



Terminologia utilizată, în structurile arborescente, folosește cuvinte ca: *descendent*, *tată*, *fiu*, *fraté*, *nepot*, *văr*, *unchi*, *străbunic*... Pentru a înțelege semnificația lor urmăriți figura de mai sus și comentariile de mai jos:

nodul 1 este rădăcina arborescenței;

tată pentru nodurile 2, 3, 4;

bunic pentru nodurile 5, 6, 7; străbunic pentru nodurile 8, 9, 10;

descendent (succesor) al nodului 1;

fiu pentru nodul 1;

frate pentru nodurile 3, 6;

tată pentru nodurile 5, 6;

bunic pentru nodul 8; unchi pentru nodul 7;

nepot pentru nodul 1;

fiu pentru nodul 2;

frate pentru nodul 5;

tată pentru nodul 8; văr pentru nodul 7;

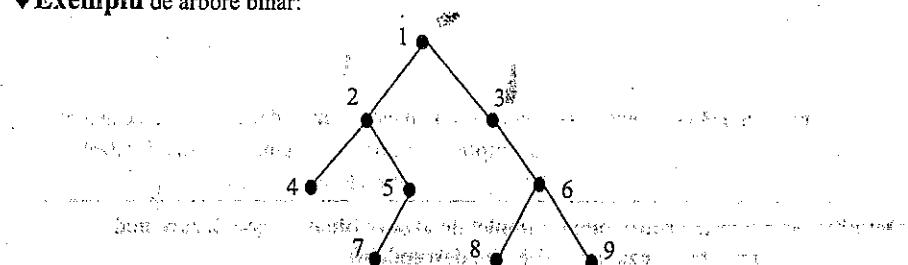
străneput pentru nodul 1;

nepot pentru nodul 2;

fiu pentru nodul 6;

**Definiție.** Se numește arbore binar, o mulțime finită de noduri care este fie vidă, fie un arbore ordonat în care fiecare nod are cel mult doi descendenți (succesori).

◆ **Exemplu** de arbore binar:



Teoriile care tratează arborii binari folosesc în plus față de cele care se referă la structurile arborescente în general, următoarele noțiuni:

- **succesor stâng**: pentru un nod, se numește succesor stâng acel succesor care este figurat în stânga sa.

**Exemplu:** pentru nodul 1, succesorul stâng este nodul 2; pentru nodul 5, succesorul stâng este nodul 7; pentru nodul 7, succesorul stâng nu există.

- **succesor drept**: pentru un nod, se numește succesor drept acel succesor care este figurat în dreapta sa.

**Exemplu:** pentru nodul 1, succesorul drept este nodul 3; pentru nodul 5, succesorul drept nu există.

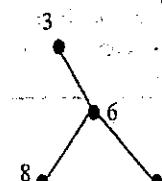
- **subarbore stâng**: pentru un nod, se numește subarbore stâng subarborele care se obține suprimând muchia care-l leagă pe acesta de succesorul său stâng; dacă succesorul stâng nu există, se spune că subarborele stâng este vid.

**Exemplu.** Pentru nodul 1 subarborele stâng este:



- **subarbore drept**: pentru un nod, se numește subarbore drept subarborele care se obține suprimând muchia care-l leagă pe acesta de succesorul său drept; dacă succesorul drept nu există, se spune că subarborele drept este vid.

**Exemplu.** Pentru nodul 1 subarborele drept este:

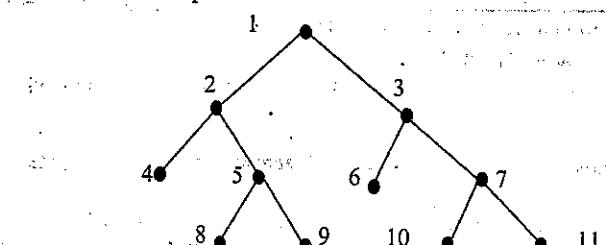


- **nod frunză (sau terminal)**: un nod se numește frunză dacă nu are descendenți.

**Exemplu.** În arborele prezentat mai sus, frunzele sunt nodurile 4, 7, 8 și 9.

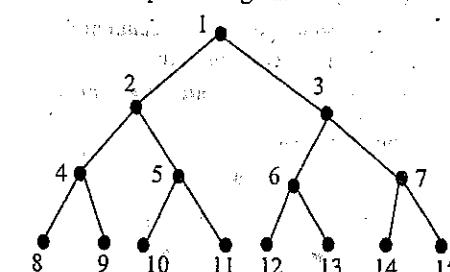
**Definiție.** Se numește arbore binar complet un arbore binar în care fiecare nod, care nu este frunză, are exact doi descendenți.

#### ♦ Exemplu de arbore binar complet:



**Propoziție.** Un arbore binar complet care are  $p$  noduri terminale, toate situate pe același nivel, are în total  $2p-1$  noduri.

#### ♦ Exemplu: Fie arborele binar complet din figura:



Deoarece are  $p=8$  noduri terminale, toate situate pe același nivel, are, conform propoziției de mai sus,  $2p-1=2\cdot8-1=15$  noduri.

### 3.1.4. Reprezentarea arborilor binari

Dacă avem în vedere faptul că un arbore binar este un arbore, care înainte de toate este un graf, putem spune că printre metodele de reprezentare a arborilor binari se numără și metodele de reprezentare a grafurilor, cum ar fi:

- reprezentarea prin matricea de adiacență;
- reprezentarea prin listele de adiacență;
- reprezentarea prin sirul muchiilor;

Deoarece aceste metode au fost deja prezentate, destul de detaliat în capitolul 2 al acestei cărți, în continuare vom prezenta decât modalitățile de reprezentare specifice arborilor binari. Acestea sunt:

- reprezentarea standard:

1. cu ajutorul vectorilor
2. folosind alocarea dinamică

- reprezentarea cu ajutorul legăturii Tata

## Reprezentarea standard

Acet mod de reprezentare se realizează astfel:

- se precizează nodul rădăcină;
- pentru fiecare nod se precizează **descendentul stâng, descendental drept și informația**.

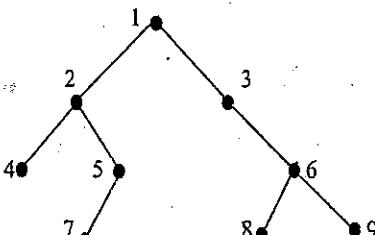
Implementarea acestui mod de reprezentare, în cadrul unui program, se face:

### 1. folosind vectorii

**Observație.** În acest caz, se folosesc vectorii **st, dr și info**, care au atâtatea componente câte noduri are arborele, și variabila **rad** cu următoarele semnificații:

- st[i]** : reține **descendentul stâng** al nodului **i**, dacă acesta există, și reține **0**, dacă nodul **i** nu are **descendent stâng**;
- dr[i]** : reține **descendentul drept** al nodului **i**, dacă acesta există, și reține **0**, dacă nodul **i** nu are **descendent drept**;
- info[i]** : reține **informația asociată** nodului **i**;
- rad** : reține nodul care reprezintă **rădăcina arborelui**.

#### ♦ Exemplul 1. În cazul arborelui binar



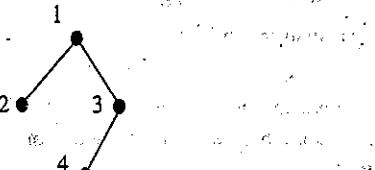
avem: **rad=1,**

$$\text{st} = (\text{st}_1, \text{st}_2, \text{st}_3, \text{st}_4, \text{st}_5, \text{st}_6, \text{st}_7, \text{st}_8, \text{st}_9) = (2, 4, 0, 0, 7, 8, 0, 0, 0),$$

$$\text{dr} = (\text{dr}_1, \text{dr}_2, \text{dr}_3, \text{dr}_4, \text{dr}_5, \text{dr}_6, \text{dr}_7, \text{dr}_8, \text{dr}_9) = (3, 5, 6, 0, 0, 9, 0, 0, 0),$$

componentele vectorului **info** se completează în funcție de cerințe.

#### ♦ Exemplul 2. În cazul arborelui binar



avem: **rad=1,**

$$\text{st} = (\text{st}_1, \text{st}_2, \text{st}_3, \text{st}_4) = (2, 0, 4, 0),$$

$$\text{dr} = (\text{dr}_1, \text{dr}_2, \text{dr}_3, \text{dr}_4) = (3, 0, 0, 0),$$

componentele vectorului **info** se completează în funcție de cerințe.

## ARBORI

În cazul acestui mod de reprezentare, secțiunile programului, care prelucrează un arbore binar cu nodurile **1..n**, sunt:

#### ♦ Secțiunea declarațiilor

```
typedef int sir[30];
sir st, dr, info;
int rad, n, i;
```

#### ♦ Secțiunea de citire a datelor de intrare

```
cout<<"Dati numarul de noduri ale arborelui n= "; cin>>n;
cout<<"Dati nodul radacina rad= "; cin>>rad;
for (i=1;i<=n;i++)
{
    cout<<"pentru nodul "<<i<<" dati : ";
    cout<<"descendentul stang : "; cin>>st[i];
    cout<<"descendentul drept : "; cin>>dr[i];
    cout<<"informatia : "; cin>>info[i];
}
```

### 2. folosind alocarea dinamică

**Observație.** În acest caz, fiecare nod este reprezentat în **Heap** de o înregistrare cu următoarea structură:

```
struct nod{
```

```
    int info;
    nod *st, *dr;
};
```

În cazul acestui mod de reprezentare, secțiunile programului, care prelucrează un arbore binar, sunt:

#### ♦ Secțiunea declarațiilor

```
struct nod{
    int info;
    nod *st, *dr;
};
nod *rad;
```

#### ♦ Secțiunea de citire a datelor de intrare

Această secțiune conține decât apelul funcției **creare**, prezentată mai jos, deoarece **programul va reține doar adresa rădăcinii**.

```
creare(rad);
```

```
void creare(nod *rad)
```

```
{
    int val;           (val = valoare)
```

```

cout<<"val="; cin>>val;
if (val!=0) {
    rad=new nod;
    rad->info=val;
    creare(rad->st);
    creare(rad->dr);
}
else rad=0;
}

```

**Exemple:** 1. Pentru arborele figurat în exemplul 1, de mai sus, nodurile se introduc în ordinea: 1, 2, 4, 0, 0, 5, 7, 0, 0, 0, 3, 0, 6, 8, 0, 0, 9, 0, 0

2. Pentru arborele figurat în exemplul 2, de mai sus, nodurile se introduc în ordinea: 1, 2, 0, 0, 3, 4, 0, 0, 0

### Reprezentarea cu ajutorul legăturii tata

Acest mod de reprezentare se realizează astfel:

- pentru fiecare nod, se precizează părintele său;
- pentru fiecare nod, se precizează ce fel de descendenter este, stâng sau drept, pentru părintele său.

**Observație.** Implementarea acestui mod de reprezentare, în cadrul unui program, se face folosind vectorii tata și desc (desc=descendent), care au atâtea componente câte noduri are arborele, cu următoarele semnificații:

- |                |   |
|----------------|---|
| <b>tata[i]</b> | : reține părintele nodului i, dacă acesta există,<br>și reține 0, dacă nodul i nu are părinte (în cazul rădăcinii);   |
| <b>desc[i]</b> | : reține valoarea -1, dacă nodul i este descendenter stâng;<br>reține valoarea 1, dacă nodul i este descendenter drept;<br>reține valoarea 0, dacă nodul i este rădăcina arborelui; |

**Exemple:** 1. Pentru arborele figurat în exemplul 1, de mai sus, avem:

$$\begin{aligned}
 \text{tata} &= (\text{tata}_1, \text{tata}_2, \text{tata}_3, \text{tata}_4, \text{tata}_5, \text{tata}_6, \text{tata}_7, \text{tata}_8, \text{tata}_9) \\
 &= (0, 1, 1, 2, 2, 3, 5, 6, 6) \\
 \text{desc} &= (\text{desc}_1, \text{desc}_2, \text{desc}_3, \text{desc}_4, \text{desc}_5, \text{desc}_6, \text{desc}_7, \text{desc}_8, \text{desc}_9) \\
 &= (0, -1, 1, -1, 1, 1, -1, -1, 1)
 \end{aligned}$$

2. Pentru arborele figurat în exemplul 2, de mai sus, avem:

$$\begin{aligned}
 \text{tata} &= (\text{tata}_1, \text{tata}_2, \text{tata}_3, \text{tata}_4) = (0, 1, 1, 3) \\
 \text{desc} &= (\text{desc}_1, \text{desc}_2, \text{desc}_3, \text{desc}_4) = (0, -1, 1, -1)
 \end{aligned}$$

### 3.1.5. Parcursarea arborilor binari

Prin parcursarea arborilor binari se înțelege examinarea în mod sistematic a nodurilor sale astfel încât fiecare nod să fie atins o singură dată.

Există trei modalități, specifice arborilor binari, de parcursere:

- parcursarea în preordine (RSD : rădăcina-stânga-dreapta)
- parcursarea în inordine (SRD : stânga-rădăcina-dreapta)
- parcursarea în postordine (SDR : stânga-dreapta-rădăcina)

*În continuare, vom prezenta o descriere, recursivă, pentru fiecare dintre aceste metode.*

**Parcursarea în preordine**, a arborelui cu rădăcina i, presupune parcursarea etapelor:

- se vizitează rădăcina (nodul i);
- dacă există, se parcurge în preordine subarborele stâng (are rădăcina st[i]);
- dacă există, se parcurge în preordine subarborele drept (are rădăcina dr[i]);

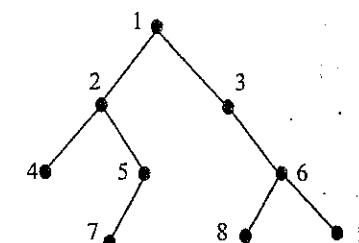
**Parcursarea în inordine**, a arborelui cu rădăcina i, presupune parcursarea etapelor:

- dacă există, se parcurge în inordine subarborele stâng (are rădăcina st[i]);
- se vizitează rădăcina (nodul i);
- dacă există, se parcurge în inordine subarborele drept (are rădăcina dr[i]);

**Parcursarea în postordine**, a arborelui cu rădăcina i, presupune parcursarea etapelor:

- dacă există, se parcurge în postordine subarborele stâng (are rădăcina st[i]);
- dacă există, se parcurge în postordine subarborele drept (are rădăcina dr[i]);
- se vizitează rădăcina (nodul i);

♦ **Exemplul 1.** Fie arborele binar:



Lista nodurilor în urma parcurgerii

în preordine este : 1, 2, 4, 5, 7, 3, 6, 8, 9

în inordine este : 4, 2, 7, 5, 1, 3, 8, 6, 9

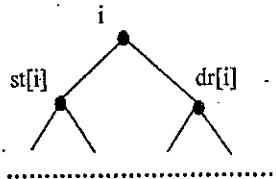
în postordine este : 4, 7, 5, 2, 8, 9, 6, 3, 1

*În continuare, prezintăm programul care, fiind dat un arbore binar cu nodurile 1,...,n și rădăcina 1, afișează listele nodurilor obținute în urma parcumerilor în preordine, în inordine și în postordine.*

**Comentarii:** La execuția programului de mai jos, se realizează următoarele:

- se citește un arbore;
- se afișează lista nodurilor obținută în urma parcurgerii în preordine (prin apelul funcției **preordine**);
- se afișează lista nodurilor obținută în urma parcurgerii în inordine (prin apelul funcției **inordine**);
- se afișează lista nodurilor obținută în urma parcurgerii în postordine (prin apelul funcției **postordine**).

Toate funcțiile, **preordine**, **inordine** și **postordine**, sunt realizate conform comentariilor care au fost prezentate mai sus, atunci când au fost descrise, și au un parametru *i* care reprezintă rădâcina subarborelui ce urmează să fie parcurs cu ajutorul lor.



```

#include<iostream.h>
#include<conio.h>
#include<stdio.h>
typedef int sir[30];
sir st, dr, info;
int rad, n, i;
void preordine( int i )
{
    cout<<info[i]<<" ";
    if (st[i]!=0) preordine(st[i]);
    if (dr[i]!=0) preordine(dr[i]);
}
void inordine( int i )
{
    if (st[i]!=0) inordine(st[i]);
    cout<<info[i]<<" ";
    if (dr[i]!=0) inordine(dr[i]);
}
void postordine( int i )
{
    if (st[i]!=0) postordine(st[i]);
    if (dr[i]!=0) postordine(dr[i]);
    cout<<info[i]<<" ";
}
  
```

```

main()
{
    clrscr();
    cout<<"Dati numarul de noduri ale arborelui n="; cin>>n;
    cout<<"Dati nodul radacina rad="; cin>>rad;
    for (i=1;i<=n;i++)
    {
        cout<<"pentru nodul "<<i<<" dati : "<<endl;
        cout<<"descendentul stang : "; cin>>st[i];
        cout<<"descendentul drept : "; cin>>dr[i];
        cout<<"informația : "; cin>>info[i];
    }
    cout<<" preordine : "<<endl; preordine(rad);
    cout<<" inordine : "<<endl; inordine(rad);
    cout<<" postordine : "<<endl; postordine(rad);
    cout<<endl;
    getch();
}
  
```

Programul de mai jos face același lucru ca și cel de sus, numai că folosește alocarea dinamică. În cazul acesta, vor apărea niște diferențe la scriere, cum ar fi: în loc de *int* își folosește *nod\** *i* (pointer), în loc de *st[i]* se folosește *i->st*, în loc de *dr[i]* se folosește *i->dr* și în loc de *info[i]* se folosește *i->info*.

```

#include<iostream.h>
#include<conio.h>
#include<stdio.h>
struct nod{
    int info;
    nod *st, *dr;
};
nod *rad;
void creare(nod* rad)
{
    int val; (val=valoare)
    cout<<"val="; cin>>val;
    if (val!=0) {
        rad=new nod;
        rad->info=val;
        creare(rad->st);
        creare(rad->dr);
    }
}
  
```

creează arborele cu radacina *rad*

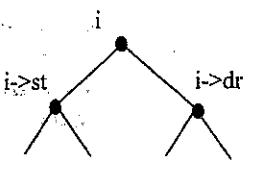
se citește o valoare (informația *rad*)  
dacă este diferită de 0,  
se alocă spațiu pentru *rad*  
se încarcă informația citită  
se creează subarborele stâng  
se creează subarborele drept

```

    else
        rad=0;
}
void preordine( nod *i)
{
    cout<<i->info<<" ";
    if (i->st!=0) preordine(i->st);
    if (i->dr!=0) preordine(i->dr);
}
void inordine( nod *i)
{
    if (i->st!=0) inordine(i->st);
    cout<<i->info<<" ";
    if (i->dr!=0) inordine(i->dr);
}
void postordine(nod *i)
{
    if (i->st!=0) postordine(i->st);
    if (i->dr!=0) postordine(i->dr);
    cout<<i->info<<" ";
}
main(){
    clrscr();
    creare(rad);
    cout<<"parcursarea in preordine"<<endl;
    preordine(rad);
    cout<<endl;
    cout<<"parcursarea in inordine"<<endl;
    inordine(rad);
    cout<<endl;
    cout<<"parcursarea in postordine"<<endl;
    postordine(rad);
    cout<<endl;
    getch();
}

```

Așa arată arborele cu rădăcina în i, prelucrat de cele trei funcții, în cazul alocării dinamice.



♦ Atenție! Funcțiile de parcursere ale arborilor binari (preordine, inordine, postordine) mai pot fi implementate și ca mai jos.

### Implementare statică

```

void preordine( int i)
{
    if (i!=0) {
        cout<<info[i]<<" ";
        preordine(st[i]);
        preordine(dr[i]);
    }
}
void inordine( int i)
{
    if (i!=0) {
        inordine(st[i]);
        cout<<info[i]<<" ";
        inordine(dr[i]);
    }
}
void postordine( int i)
{
    if (i!=0) {
        postordine(st[i]);
        postordine(dr[i]);
        cout<<info[i]<<" ";
    }
}

```

### Implementare dinamică

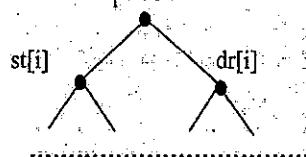
```

void preordine( nod *i)
{
    if (i!=0) {
        cout<<i->info<<" ";
        preordine(i->st);
        preordine(i->dr);
    }
}
void inordine( nod *i)
{
    if (i!=0) {
        inordine(i->st);
        cout<<i->info<<" ";
        inordine(i->dr);
    }
}
void postordine(nod *i)
{
    if (i!=0) {
        postordine(i->st);
        postordine(i->dr);
        cout<<i->info<<" ";
    }
}

```

### Arborele este reprezentat folosind vectorii st, dr și info.

**Observație.** În continuare, vom prezenta schema care stă la baza realizării funcțiilor de prelucrare, sau analiză, a tuturor nodurilor unui arbore binar, cu rădăcina și reprezentarea grafică:



atunci când analiza, sau prelucrarea, nodului i nu influențează prelucrarea sau analiza nodurilor din subarborele stâng sau subarborele drept.

Funcția are numele ...dorit și un parametru formal, i, de tip int, care reprezintă rădăcina subarborelui prelucrat, și procedează astfel:

- dacă se verifică condițiile referitoare la nodul i,  
atunci .....  
altfel .....
- dacă există subarborele stâng, (de cele mai multe ori se prelucrează direct).  
atunci .....  
altfel .....
- dacă există subarborele drept, (de cele mai multe ori se prelucrează direct)  
atunci .....  
altfel .....

void prelucrare( int i )

```
{
    if (..... nodul i verifică....)
        .....
    else .....  

        if (st[i]!=0)      (asta înseamnă "dacă subarborele stâng există")
            .....
        else .....  

        if (dr[i]!=0)     (asta înseamnă "dacă subarborele drept există")
            .....
        else .....  

}
```

**Observații.** 1. Funcțiile realizate, după schema de mai sus, sunt valabile atunci când arborele este reprezentat folosind vectorii st, dr și info.

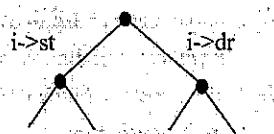
2. Ramurile else, de la if-urile din schemă, pot lipsi în unele situații.

3. if (..... nodul i verifică....) se poate traduce în:

- |                               |   |
|-------------------------------|---|
| if (i % 2 == 0)               | : atunci când trebuie prelucrate nodurile cu numere de ordine pare; |
| if (info[i] % 2 == 0)         | : atunci când trebuie prelucrate nodurile cu informații pare;       |
| if (st[i]==0)                 | : atunci când trebuie prelucrate nodurile fără descendenter stâng;  |
| if (dr[i]==0)                 | : atunci când trebuie prelucrate nodurile fără descendenter drept;  |
| if ((st[i]!=0) && (dr[i]!=0)) | : atunci când trebuie prelucrate nodurile cu doi descendenti;       |

### Arborele este reprezentat folosind alocare dinamică.

**Observație.** În continuare, vom prezenta schema care stă la baza realizării funcțiilor de prelucrare, sau analiză, a tuturor nodurilor unui arbore binar, cu rădăcina și reprezentarea grafică:



atunci când analiza, sau prelucrarea, nodului i nu influențează prelucrarea, sau analiza, nodurilor din subarborele stâng sau subarborele drept.

Funcția are numele ...dorit și un parametru formal, i de tip nod\* (pointer), care reprezintă rădăcina subarborelui prelucrat, și procedează astfel:

- dacă se verifică condițiile referitoare la nodul i,  
atunci .....  
altfel .....
- dacă există subarborele stâng, (de cele mai multe ori se prelucrează direct)  
atunci .....  
altfel .....

```

    - dacă există subarborele drept, (de cele mai multe ori se prelucră direct)
      atunci .....  

      altfel .....  

void prelucrare(nod *i)  

{  

    if(..... nodul i verifică....)  

        .....  

    else .....  

    if(i->st!=0) (asta înseamnă "dacă subarborele stâng există")  

        .....  

    else .....  

    if(i->dr!=0) (asta înseamnă "dacă subarborele drept există")  

        .....  

    else .....  

}
  
```

**Observații.** 1. Funcțiile realizate, după schema de mai sus, sunt valabile atunci când arborele este reprezentat folosind **alocare dinamică**.

2. Ramurile **else**, de la **if**-urile din schema, pot lipsi în unele situații.

3. **if(..... nodul i verifică....)** se poate traduce în:

- if (i->notat %2==0) : atunci când trebuie prelucrate nodurile cu numere de ordine pare;
- if (i->info % 2==0) : atunci când trebuie prelucrate nodurile cu informații pare;
- if (i->st==0) : atunci când trebuie prelucrate nodurile fără descendenter stâng;
- if (i->dr==0) : atunci când trebuie prelucrate nodurile fără descendenter drept;
- if ((i->st!=0) && (i->dr!=0)) : atunci când trebuie prelucrate nodurile cu doi descendenti;

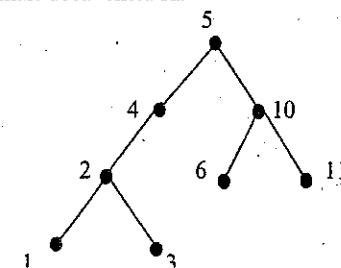
### 3.1.6. Arboare binar de căutare

Fie  $A=(V, U)$  un arbore binar în care fiecare nod conține cel puțin un câmp informațional de un tip ordinal (un tip pe care este definită o relație de ordine). În acest caz, câmpul se poate numi **câmp cheie**, iar valorile inscrise în acest câmp se numesc **chei**.

**Definiție.** Se numește **arbore binar de căutare** un arbore binar în care, pentru fiecare nod, se verifică următoarele:

- orice cheie asociată unui nod din subarborele stâng este mai mică decât cheia asociată nodului;
- orice cheie asociată unui nod din subarborele drept este mai mare decât cheia asociată nodului.

♦ **Exemplu:** Arborele binar figurat mai jos (pentru care presupunem valorile cheilor ca fiind valorile scrisă lângă nodurile sale) este **arbore binar de căutare**, deoarece, pentru orice nod, cheile nodurilor din subarborele stâng sunt mai mici decât cheia sa, iar cheile nodurilor din subarborele drept sunt mai mari decât cheia sa.



**Operațiile principale, care se pot efectua asupra unui arbore binar de căutare, sunt:**

- **crearea** arborelui;
- **adăugarea** unui nod la arbore;
- **căutarea** unui nod în arbore;
- **listarea** informațiilor nodurilor arborelui;
- **stergerea** unui nod al arborelui.

**Atenție!** Pentru toate funcțiile prezentate mai jos, considerăm arborele reprezentat folosind **alocarea dinamică**.

**Structura nodurilor poate fi:**

```

struct nod{
    int cheie;
    ..... alte câmpuri informaționale
    nod *st, *dr;
};
  
```

## Crearea și adăugarea

Crearea unui arbore binar de căutare se face aplicând de un număr de ori operația de adăugare. De aceea, în continuare, vom prezenta decât funcția de adăugare, a unui nou nod la un arbore binar de căutare.

**Observație:** Presupunem că dorim să adăugăm un nod cu cheia val (val=valoare), la un arbore binar de căutare care are rădăcina c. Acest lucru se realizează folosind funcția Adauga caracterizată astfel:

are doi parametrii:

c : reprezintă rădăcina arborelui în care se face adăugarea;

val : reprezintă cheia nodului care se dorește adăugat;

și procedează astfel:

- dacă nodul c există ( $c \neq 0$ ):

- se compară cheia nodului c cu cheia nodului care se adaugă la arbore (val);

- dacă cheia nodului c este egală cu cheia val,

- se afișează mesajul "Acest nod a mai fost adăugat";

- dacă cheia nodului c este mai mică decât val,

- se reia procesul de adăugare a nodului în subarborele drept  
(se aplică funcția Adauga asupra subarborelui cu rădăcina c->dr);

- dacă cheia nodului c este mai mare decât val,

- se reia procesul de adăugare a nodului în subarborele stâng;  
(se aplică funcția Adauga asupra subarborelui cu rădăcina c->st);

altfel

- se alocă spațiu pentru noul nod, c;

- se încarcă informațiile;

- se stabilesc legăturile cu descendenții săi, adică:

c->st=0;

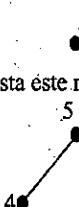
c->dr=0;

♦ **Exemplu:** Să se creeze un arbore binar de căutare cu cheile:

5, 4, 10, 2, 1, 3, în care rădăcina are cheia 5.

**Rezolvare:**

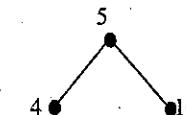
Se pornește de la arborele binar de căutare vid. Se adaugă nodul cu cheia 5 (la început se adaugă rădăcina)



Se adaugă nodul 4; cum acesta este mai mic decât 5, se adaugă în subarborele stâng.

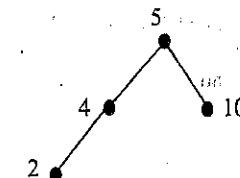


Se adaugă nodul 10; cum acesta este mai mare decât 5, se adaugă în subarborele drept.



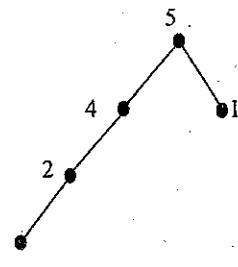
Se adaugă nodul 2; cum acesta este mai mic decât 5, se adaugă în subarborele stâng.

Deoarece este mai mic decât 4 se adaugă în subarborele său stâng.



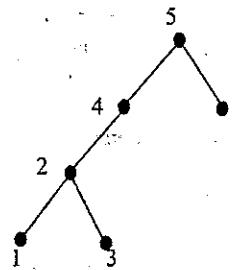
Se adaugă nodul 1; cum acesta este mai mic decât 5, se adaugă în subarborele stâng.

Deoarece este mai mic decât 4, se adaugă în subarborele său stâng. Deoarece este mai mic decât 2, se adaugă în subarborele său stâng.



Se adaugă nodul 3; cum acesta este mai mic decât 5, se adaugă în subarborele stâng.

Deoarece este mai mic decât 4, se adaugă în subarborele său stâng. Deoarece este mai mare decât 2, se adaugă în subarborele său drept.



În continuare, prezintăm funcția de adăugare a nodului cu cheia val la arborele binar de căutare cu rădăcina c.

```

void adauga(nod *c, int val)
{
    if (c!=0)
        if (c->cheie==val)
            cout<<"nodul exista deja";
        else
            if (c->cheie<val)
                adauga(c->dr, val);
            else
                adauga(c->st, val);
    else {
        c=new nod;
        c->cheie=val;
        .....
        c->st=0;
        c->dr=0;
    }
}

```

**Observație.** Crearea unui arbore binar de căutare s-ar putea face, folosind funcția de mai sus, astfel:

```

rad=0;
do{
    cout<<"val= "; cin>>val;
    if (val!=1)
        adauga(rad, val);
}while (val!= -1);

```

## Căutarea

**Observație.** Presupunem că dorim să căutăm un nod cu cheia val (val=valoare), într-un arbore binar de căutare care are rădăcina c. Acest lucru se realizează folosind funcția Cauta caracterizată astfel:

*are doi parametrii:*

- c : reprezintă rădăcina arborelui în care se face căutarea;
  - val : cheia nodului care se căută;
- și procedează astfel:*
- dacă nodul c există (c!=0):
    - se compară cheia nodului c cu cheia nodului care se căută (val)
    - dacă cheia nodului c este egală cu cheia val,
      - se afisează mesajul "Acum nod are informatiile ...";

## ARBORI

- dacă cheia nodului c este mai mică decât val,
    - se reia procesul de căutare a nodului în subarborele drept
      - (se aplică funcția Cauta asupra subarborelui cu rădăcina c->dr);
  - dacă cheia nodului c este mai mare decât val,
    - se reia procesul de căutare a nodului în subarborele stâng
      - (se aplică funcția Cauta asupra subarborelui cu rădăcina c->st);
- altfel
- se afisează mesajul "nodul căutat nu există";

*În continuare, prezentăm funcția de căutare a nodului cu cheia val în arborele binar de căutare cu rădăcina c.*

```

void cauta(nod *c, int val)
{
    if (c!=0) if (c->cheie==val)
        cout<<".....";
    else if (c->cheie<val) cauta(c->dr, val);
    else cauta(c->st, val);
    else cout<<"nodul cautat nu există";
}

```

## Comentarii.

1. Odată creat un arbore de căutare, permite regăsirea mai rapidă a informației decât dacă ar fi memorat secesional, deoarece odată analizată informația atașată unui nod, în caz de inegalitate se trece ori în subarborele stâng (caz în care se renunță la parcurgerea subarborelui drept) ori în subarborele drept (caz în care se renunță la parcurgerea subarborelui stâng).
2. Există și situația în care scade eficiența de căutare; acest lucru se întâmplă atunci când cheile sunt introduse în ordine crescătoare sau descrescătoare. În acest caz, arborele degeneră într-o listă liniară.

## Listarea

Dacă avem în vedere faptul că un arbore binar de căutare este un arbore binar, care până la urmă este un graf, putem trage concluzia că listarea arborelui binar de căutare se poate face:

- folosind metodele de parcurgere ale grafurilor: în lățime și în adâncime;
- folosind metodele de parcurgere specifice arborilor binari:
  - în preordine, în inordine și în postordine.

Deoarece aceste metode au fost prezentate deja, în capitolele anterioare, acum ne rezumăm decât la a face o trimiterie la ele.

**Observație.** Parcurgând în inordine nodurile unui arbore binar de căutare, ale cărui chei sunt numere întregi, obținem lista cheilor aranjate în ordine crescătoare.

## Ștergerea

**Observație.** Presupunem că dorim să ștergem un nod cu cheia val (val=valorare), dintr-un arbore binar de căutare care are rădăcina c. Acest lucru se realizează folosind funcția Sterge caracterizată astfel:

are doi parametri:

c : care reprezintă rădăcina arborelui din care se face ștergerea;

val : care reprezintă cheia nodului care se dorește sters;

și procedează astfel:

1. Dacă nodul c există (c!=0):

1.1. se compară cheia nodului c cu cheia nodului care se dorește sters;

se pot întâlni situațiile:

a) dacă cheia nodului c este egală cu cheia val,

1) dacă nici subarborele stâng nici cel drept nu există

– se face ștergerea având grija ca la părintele lui să înlocuim adresa către el cu nil;

2) dacă subarborele stâng nu există,

– se face ștergerea având grija ca la părintele lui să înlocuim adresa către el cu adresa subarborelui său drept;

3) dacă subarborele drept nu există,

– se face ștergerea având grija ca la părintele lui să înlocuim adresa către el cu adresa subarborelui său stâng;

4) dacă subarborele stâng și cel drept există,

(în acest caz, nodul se va sterge, după cum se va vedea, numai logic)

– se determină cel mai din dreapta nod al subarborelui stâng corespunzător nodului care urmează să fie sters (acesta este nodul care, efectiv, se sterge fizic);

– cheia și toate celelalte informații utile, conținute de nodul care va fi sters fizic, se mută în nodul care va fi sters logic;

– subarborele stâng al nodului care se va sterge fizic se leagă:

– în stânga nodului care se sterge logic (dacă nodul care se sterge fizic este descendent direct al nodului care se sterge logic);

– în dreapta tatălui nodului care se sterge fizic (altfel);

– se sterge fizic nodul care a fost identificat ca fiind cel mai în dreapta în subarborele stâng al nodului cu cheia val.

b) dacă cheia nodului c este mai mică decât cheia val,

se reia procesul de ștergere, a nodului, în subarborele drept

(se aplică funcția Sterge asupra subarborelui cu rădăcina c->dr).

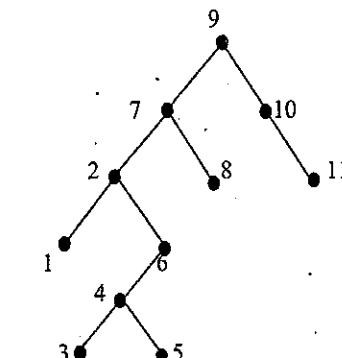
c) dacă cheia nodului c este mai mare decât cheia val,  
se reia procesul de ștergere, a nodului, în subarborele stâng;  
(se aplică funcția Sterge asupra subarborelui cu rădăcina c->st);

2. Dacă nodul c nu există (c==0):

– se afișează un mesaj prin care se precizează că nodul, care se dorește sters, nu există în arbore.

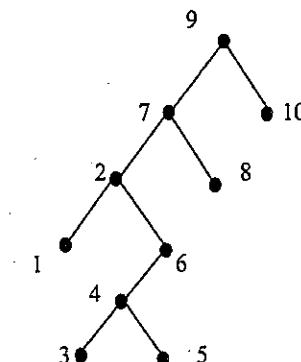
♦ **Exemplu:** Fiind dat arborele binar de căutare, din figura de mai jos, să se aplice algoritmul de ștergere, prezentat mai sus, pentru:

1. ștergerea nodului 11;
2. ștergerea nodului 10;
3. ștergerea nodului 7.



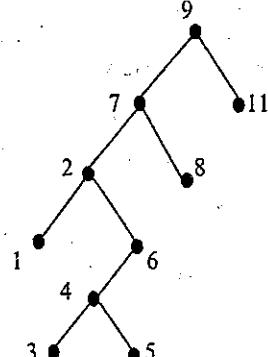
**Rezolvare:**

1.



la părintele lui 11, 10, se înlocuiește adresa către el cu nil

2.

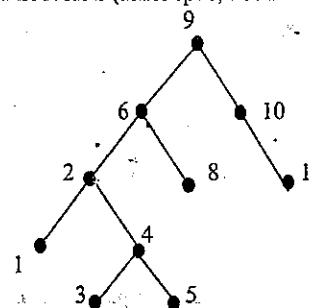


deoarece subarborele stâng, al nodului care se șterge, nu există, se înlocuieste adresa tatălui său către el cu adresa subarborelui său drept

3. Pentru a șterge nodul 7, care are doi descendenți, se procedează astfel:

- se determină cel mai din dreapta nod, al subarborelui său stâng; acesta este nodul 6;
- deoarece nodul 6 nu este descendente direct al nodului pe care dorim să-l ștergem (7), procedăm astfel:

- informația conținută de nodul 6 se mută în nodul 7;
- subarborele stâng al nodului 6 se leagă în dreapta tatălui său, adică în dreapta nodului 2 (altfel spus, nodul 4 trece pe poziția nodului 6);



*În continuare, prezentăm funcția de ștergere a nodului cu cheia val din arborele binar de căutare cu rădăcina c.*

```

void sterge(nod *c, int val)
{
    nod *urm;
    if(c!=0)
        if(c->cheie==val)
            {
                if((c->st==0) && (c->dr==0))
                    {
                        delete c;
                        c=0;
                    }
                else
                    if(c->st!=0)
                        if(c->dr!=0)
                            {
                                urm=c->dr;
                                delete c;
                                c=urm;
                            }
                        else
                            if(c->st!=0)
                                urm=c->st;
                            else
                                urm=c->dr;
                            c=urm;
                    }
                else
                    if(c->cheie<val) sterge(c->dr, val);
                    else sterge(c->st, val);
            }
        else cout<<"valoarea nu se află în arbore";
    }
  
```

```

else
    if(c->st==0)
        {
            urm=c->dr;
            delete c;
            c=urm;
        }
    else
        if(c->dr==0)
            {
                urm=c->st;
                delete c;
                c=urm;
            }
        else
            celmaidindreapta(c,c->st);
    }
else
    if(c->cheie<val) sterge(c->dr, val);
    else sterge(c->st, val);
else
    cout<<"valoarea nu se află în arbore";
}
  
```

Funcția `Sterge` folosește funcția `celmaidindreapta` care determină, referitor la nodul care trebuie șters, atunci când acesta are doi descendenți, cel mai din dreapta nod al subarborelui său stâng.

```

void celmaidindreapta(nod *c, nod *d)
{
    nod *urm;
    if(d->dr!=0)
        celmaidindreapta(c, d->dr);
    else
        {
            c->cheie=d->cheie;
            urm=d;
            d=d->st;
            delete urm;
        }
}
  
```

♦ Structura generală a programului care folosește funcțiile de mai sus ar putea fi:

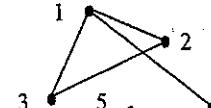
```
#include<conio.h>
#include<stdio.h>
#include<iostream.h>
struct nod{
    int cheie;
    ..... alte câmpuri informaționale
    nod *st, *dr;
};

nod *rad;
- se scrie funcția de adăugare : Adauga
- se scrie funcția de listare : Preordine
- se scrie funcția : Celmaiindreapta
- se scrie funcția de stergere : Sterge
- se scrie programul principal (funcția main()), astfel:
main(){ clrscr();
    rad=0;
    do{
        cout<<"val="; cin>>val;
        if (val!=1) adauga(rad, val);
    }while (val!=1);
    cout<<"Operatiile pe care le puteti face sunt :"<<endl;
    cout<<"1. Adaugare "<<endl;
    cout<<"2. Listare "<<endl;
    cout<<"3. Cautare "<<endl;
    cout<<"4. Stergere "<<endl;
    cout<<"dati optiunea dorita : "; cin>>op;
    switch (op)
    {
        case 1 : {
            cout<<"Dati cheia nodului pe care-l adaugati";
            cin>>val; adauga(rad, val); break;
        }
        case 2 : {
            cout<<"Lista nodurilor ";
            preordine(rad); break;
        }
        case 3 : {
            cout<<"Dati cheia nodului pe care-l cautati";
            cin>>val; cauta(rad, val); break;
        }
        case 4 : {
            cout<<"Dati cheia nodului pe care-l stergeti";
            cin>>val; sterge(rad, val); break;
        }
    }
}
```

## 3.2. PROBLEME PROPUSE

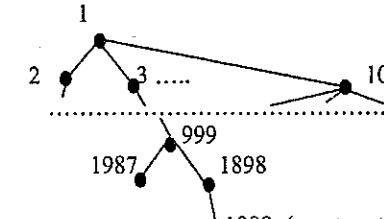
### 3.2.1. Noțiunea de arbore

1. Pentru graful reprezentat în figura:



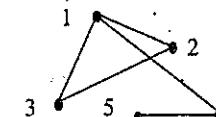
să se eliminate o muchie astfel încât acesta să devină arbore.

2. Câte muchii are arborele din figura:



1989 (acesta este ultimul nod)

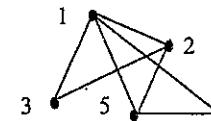
3. Pentru graful din figura de mai jos, să se aplică algoritmul de verificare a existenței ciclurilor prezentat în secțiunea 3.1.1.



4. Fieind dat un graf, să se realizeze un program cu ajutorul căruia se poate verifica dacă graful este sau nu arbore.

### 3.2.2. Arbore parțial de cost minim

1. Pentru graful reprezentat în figura:



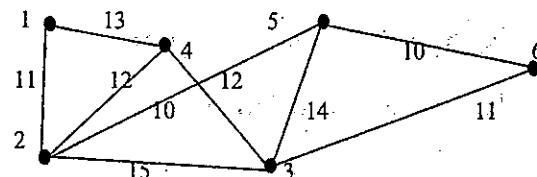
să se dea exemplu de un arbore parțial.

2. Pentru graful reprezentat în figura:



să se precizeze dacă conține sau nu un arbore parțial, și de ce?

3. Fie graful din figura de mai jos (costul fiecărei muchii fiind scris pe ea).



Să se determine un arbore parțial de cost minim, aplicând algoritmul prezentat în secțiunea 3.1.2.

### 3.2.3. Arbori binari

#### 3.2.3.1. Afisări

Fiind dat un arbore binar cu nodurile noteate 1,...,n, și rădăcina 1, în care fiecare nod are ca informație o valoare întreagă, să se realizeze subprograme, în C/C++, care afișează:

1. informațiile conținute de noduri;
2. informațiile conținute de nodurile cu numere de ordine pare;
3. nodurile care rețin ca informații numere pare;
4. nodurile cu doi descendenți;
5. nodurile care au decât descendantul stâng;
6. nodurile care au decât descendantul drept;
7. informațiile nodurilor care au decât descendantul stâng;
8. informațiile nodurilor care au decât descendantul drept;
9. nodurile terminale;
10. informațiile nodurilor terminale;
11. descendenții nodului **nod**;
12. nodurile care au ca tată pe nodul **nod**;
13. descendantul stâng al nodului **nod**;
14. descendantul drept al nodului **nod**;
15. tatăl nodului **nod**;
16. nivelul pe care se află nodul **nod**;
17. nodurile de pe nivelul **p**;
18. informațiile nodurilor de pe nivelul **p**;
19. nodurile a căror informație este egală cu valoarea **val**;
20. nodurile a căror informație este egală cu valoarea **val**, și se găsesc pe nivelul **p**;
21. nodurile a căror informație este egală cu valoarea **val**, și sunt terminale.

#### 3.2.3.2. Numărare

Fiind dat un arbore binar cu nodurile noteate 1,...,n, și rădăcina 1, în care fiecare nod are ca informație o valoare întreagă, să se realizeze subprograme, în C/C++, care calculează:

1. numărul nodurilor;
2. numărul nodurilor cu numere de ordine pare;
3. numărul nodurilor care rețin informații pare;
4. numărul descendenților nodului **nod**;
5. numărul nodurilor terminale;
6. numărul nodurilor care au doi descendenți;
7. numărul nodurilor care au decât descendant stâng;
8. numărul de nivele;
9. numărul nodurilor de pe nivelul **p**.

#### 3.2.3.3. Sume

Fiind dat un arbore binar cu nodurile noteate 1,...,n, și rădăcina 1, în care fiecare nod are ca informație o valoare întreagă, să se realizeze subprograme, în C/C++, care calculează:

1. suma informațiilor nodurilor;
2. suma informațiilor nodurilor cu numere de ordine pare;
3. suma informațiilor nodurilor terminale;
4. suma informațiilor nodurilor care au decât descendant stâng;
5. suma informațiilor nodurilor de pe nivelul **p**.

#### 3.2.3.4. Produse

Fiind dat un arbore binar cu nodurile noteate 1,...,n, și rădăcina 1, în care fiecare nod are ca informație o valoare întreagă, să se realizeze subprograme, în C/C++, care calculează:

1. produsul informațiilor nodurilor;
2. produsul informațiilor nodurilor cu numere de ordine pare;
3. produsul informațiilor nodurilor terminale;
4. produsul informațiilor nodurilor care au decât descendant stâng;
5. produsul informațiilor nodurilor de pe nivelul **p**.

#### 3.2.3.5. Verificări

Fiind dat un arbore binar cu nodurile noteate 1,...,n, și rădăcina 1, în care fiecare nod are ca informație o valoare întreagă, să se realizeze subprograme, în C/C++, care:

1. verifică dacă toate nodurile conțin informații numere pare;
2. verifică dacă toate nodurile de pe nivelul **p** conțin informații numere pare;
3. verifică dacă toate nodurile terminale conțin informații numere pare;
4. verifică dacă conține cel puțin un nod a cărui informație este un număr par;
5. verifică dacă conține cel puțin un nod, pe nivelul **p**, a cărui informație este pară;
6. verifică dacă conține cel puțin un nod terminal a cărui informație este un număr par;
7. verifică dacă nu conține noduri a căror informație este un număr par;

8. verifică dacă nu conține noduri pe nivelul p a căror informație este un număr par;
9. verifică dacă nu conține noduri terminale a căror informație este un număr par;
10. verifică dacă arborele este complet;
11. verifică dacă conține noduri a căror informație este egală cu val;
12. verifică dacă conține noduri, pe nivelul p, a căror informație este egală cu val;
13. verifică dacă conține noduri terminale a căror informație este egală cu val.

### 3.2.3.6. Maxime (minime)

Fiind dat un arbore binar cu nodurile noteate 1,...,n, și rădăcina 1, în care fiecare nod are ca informație o valoare întreagă pozitivă, să se realizeze subprograme care determină:

1. maximul (minimul) informațiilor conținute de noduri;
2. maximul (minimul) informațiilor nodurilor cu numere de ordine pare;
3. maximul (minimul) informațiilor pare.

### 3.2.4. Arbori binari de căutare

1. Să se deseneze arborii binari de căutare ale căror noduri au drept chei valorile:  
a) 8, 5, 7, 12, 4, 9, 3 (8: cheia rădăcinii); b) 12, 4, 8, 19, 54, 31, 6 (12: cheia rădăcinii).
2. Fiind dat sirul  $x_1, x_2, \dots, x_n$ , să se afișeze valorile componentelor sale în ordine crescătoare (folosind un arbore binar de căutare).
3. Fiind dat un arbore binar de căutare, să se realizeze subprograme care returnează:  
a) valoarea celei mai mari chei; b) valoarea celei mai mici chei.
4. Să se realizeze un program de gestiune a unei biblioteci, folosind un arbore binar de căutare (nodurile arborelui rețin informații ca: cod (cheia), nume autor, titlu, editura iar programul permite operații ca: căutarea unui scriitor după codul său, adăugarea unei noi achiziții, listarea tuturor cărților, stergerea unui nod după un cod dat...).

### 3.2.5. Probleme diverse

1. Fie  $G=(V,M)$  un graf. Să se demonstreze că următoarele afirmații sunt echivalente:
  - a)  $G$  este arbore;
  - b)  $G$  este conex, minimal cu această proprietate;
  - c)  $G$  este fără cicluri, maximal cu această proprietate.
2. Fie  $G$  un graf, cu  $n \geq 3$  vârfuri. Să se demonstreze că următoarele afirmații sunt echivalente:
  - a)  $G$  este arbore;
  - b)  $G$  este conex și are  $n-1$  muchii;
  - c)  $G$  este fără cicluri și are  $n-1$  muchii.
3. Să se demonstreze că un graf  $G=(V,M)$  conține un arbore parțial dacă și numai dacă este conex.
4. Să se demonstreze că un graf  $G=(V,M)$  cu  $n$  vârfuri și cel puțin  $n$  muchii conține cel puțin un ciclu.
5. Să se demonstreze că un arbore binar complet care are  $n$  noduri terminale, toate situate pe același nivel, are în total  $2n-1$  noduri.

## 3.3. INDICAȚII ȘI RĂSPUNSURI

### 3.3.1. Notiunea de arbore

#### Problema 1

Eliminând muchia [2,3] (sau [1,2] sau [1,3]) graful devine un graf conex și fără cicluri, deci devine arbore.

#### Problema 2

Conform teoremei:

"Un arbore cu  $n$  noduri are  $n-1$  muchii"

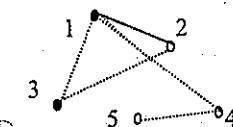
graful din problemă are  $1989-1=1988$  muchii.

#### Problema 3

$G=(V,M)$  unde  $V=\{1,2,3,4,5\}$  și

$M=\{u_1, u_2, u_3, u_4, u_5\}=\{[1,2], [1,3], [1,4], [2,3], [4,5]\}$

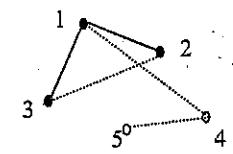
- Pas1.  $vf\_luate=\{1, 2\}$  extremitățile primei muchii,  $u_1$ ;  
 $m\_neluate=\{2, 3, 4, 5\}$  indicii muchiilor neluate în calcul, până în prezent;



- Pas2. Printre indicii 2, 3, 4, 5 ai muchiilor neluate în calcul, se caută o muchie care are o extremitate în  $vf\_luate=\{1, 2\}$ .

Se găsește, de exemplu,  $u_2=[1,3]$ ,

și extremitatea sa, 3, care nu se găsește în  $vf\_luate$  se adaugă la  $vf\_luate$ , deci  $vf\_luate=\{1, 2, 3\}$ , iar indicele său, 2, este eliminat din  $m\_neluate$ , adică  $m\_neluate=\{3, 4, 5\}$ .



- Pas2'. Printre indicii 3, 4, 5 ai muchiilor neluate în calcul, se caută o muchie care are o extremitate în  $vf\_luate=\{1, 2, 3\}$ .

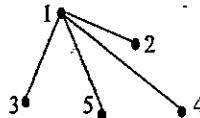
Se găsește, de exemplu,  $u_3=[2,3]$ ,

care, deoarece are ambele extremități în mulțimea  $vf\_luate$ , înseamnă că s-a găsit un ciclu ( $C=[1, 2, 3, 1]$ ). Algoritmul se oprește aici.

**Problema 4**

Pentru a demonstra că un graf este arbore, trebuie arătat că este conex și fără cicluri. În baza celor spuse, programul se realizează astfel:

- se citește graful; presupunem că este reprezentat prin sirul muchiilor  $u_1, u_2, \dots, u_m$ , dar în același timp se construiește și matricea de adiacență  $A$ ;
- se verifică dacă graful conține cicluri:  
pentru aceasta se procedează la fel cum s-a procedat în programul, prezentat în secțiunea 3.1.1., de verificare a existenței ciclurilor;
- se verifică dacă graful este conex:  
pentru aceasta se procedează la fel cum s-a procedat în programul, prezentat în secțiunea 2.1.10., de verificare a conexității unui graf.

**3.3.2. Arbore parțial de cost minim****Problema 1****Problema 2**

Conform teoremei:

"Un graf conține un arbore parțial dacă și numai dacă este conex" și graful din problemă admite un arbore parțial (deoarece este conex).

**Problema 3**

pas1. se pleacă de la nodul 1, deci  $pl=1$ ;

$$Tata[1]=0;$$

$Costul=0$  (costul total al arborelui deja construit);

$Ap[1]=0$  (pentru că nodul 1 a fost adăugat la arbore);

$$Ap=(Ap[1], Ap[2], Ap[3], Ap[4], Ap[5], Ap[6])=(0, 1, 1, 1, 1, 1);$$

Pas2. Se alege muchia de cost minim, care are o extremitate în nodurile alese deja, {1}, și cealaltă într-un nod neales încă, {2, 3, 4, 5, 6}; se găsește {1, 2};

$$Tata[2]=1;$$

$$Costul=Costul+cost[1,2]=0+11=11;$$

$Ap[2]=0$  (deoarece nodul 2 este ales);

$$Deci, Ap=(Ap[1], Ap[2], Ap[3], Ap[4], Ap[5], Ap[6])=(0, 0, 2, 2, 2, 1);$$

Pas2'. Se alege muchia de cost minim care are o extremitate în nodurile alese deja, {1, 2}, și cealaltă într-un nod neales încă, {3, 4, 5, 6}; se găsește {2, 3};

$$Tata[3]=2;$$

$$Costul=Costul+cost[2,3]=11+10=21;$$

$Ap[5]=0$  (deoarece nodul 5 este ales);

$$Deci, Ap=(Ap[1], Ap[2], Ap[3], Ap[4], Ap[5], Ap[6])=(0, 0, 5, 2, 0, 5);$$

Pas2''. Se alege muchia de cost minim, care are o extremitate în nodurile alese deja,

$$\{1, 2, 5\}, și cealaltă într-un nod neales încă, {3, 4, 6}; se găsește [5,6];$$

$$Tata[6]=5;$$

$$Costul=Costul+cost[5,6]=21+10=31;$$

$Ap[6]=0$  (deoarece nodul 6 este ales);

$$Deci, Ap=(Ap[1], Ap[2], Ap[3], Ap[4], Ap[5], Ap[6])=(0, 0, 6, 2, 0, 0);$$

Pas2'''. Se alege muchia de cost minim, care are o extremitate în nodurile alese deja,

$$\{1, 2, 5, 6\}, și cealaltă într-un nod neales încă, {3, 4}; se găsește [6,3];$$

$$Tata[3]=6;$$

$$Costul=Costul+cost[6,3]=31+11=42;$$

$Ap[3]=0$  (deoarece nodul 3 este ales);

$$Deci, Ap=(Ap[1], Ap[2], Ap[3], Ap[4], Ap[5], Ap[6])=(0, 0, 0, 2, 0, 0);$$

Pas2''''. Se alege muchia de cost minim, care are o extremitate în nodurile alese deja,

$$\{1, 2, 3, 5, 6\}, și cealaltă într-un nod neales încă, {4}; se găsește [2,4];$$

$$Tata[4]=2;$$

$$Costul=Costul+cost[4,2]=42+12=54;$$

$Ap[4]=0$  (deoarece nodul 4 este ales);

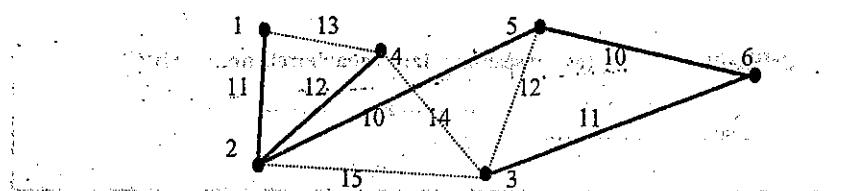
$$Deci, Ap=(Ap[1], Ap[2], Ap[3], Ap[4], Ap[5], Ap[6])=(0, 0, 0, 0, 0, 0);$$

În final, avem:

$$costul=54;$$

$$Tata=(Tata[1], Tata[2], Tata[3], Tata[4], Tata[5], Tata[6])=(0, 1, 6, 2, 2, 3).$$

Arborele parțial de cost minim, obținut în urma aplicării algoritmului prezentat în secțiunea 3.1.2., este:

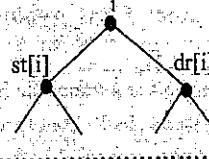


### 3.3.3. Arboare binari

#### 3.3.3.1. Afisări

Arboarele este reprezentat folosind vectorii st, dr și info.

**Observație:** În continuare, vom prezenta schema care să la baza realizării funcțiilor de prelucrare, sau analiză, a tuturor nodurilor unui arbor binar, cu rădăcina i și reprezentarea grafică:



atunci când analiza, sau prelucrarea, nodului i nu influențează prelucrarea sau analiza nodurilor din subarborele stâng sau subarborele drept.

Funcția are numele ...dorit și un parametru formal, i, de tip întreg, care reprezintă rădăcina subarborelui prelucrat, și procedează astfel:

- dacă se verifică condițiile referitoare la nodul i,  
atunci .....  
altfel .....
- dacă există subarborele stâng, (de cele mai multe ori se prelucrează)  
atunci .....  
altfel .....
- dacă există subarborele drept, (de cele mai multe ori se prelucrează)  
atunci .....  
altfel .....

```

void prelucrare( int i)
{
    if(.... nodul i verifică....)
        .....
    else .....
    if(st[i]==0)      (asta înseamnă "dacă subarborele stâng există")
        .....
    else .....
    if(dr[i]==0)     (asta înseamnă "dacă subarborele drept există")
        .....
    else .....
}
  
```

**Observații:** 1. Funcțiile, realizate după schema de mai sus, sunt valabile atunci când arboarele este reprezentat folosind vectorii st, dr și info.

2. Ramurile else, de la if-urile din schema, pot lipsi în unele situații.

3. if(.... nodul i verifică....) se poate traduce în:

if (i % 2==0) : atunci când trebuie prelucrate nodurile cu numere de ordine pare;

if (info[i] % 2==0) : atunci când trebuie prelucrate nodurile cu informații pare;

if (st[i]==0) : atunci când trebuie prelucrate nodurile fără descendenter stâng;

if (dr[i]==0) : atunci când trebuie prelucrate nodurile fără descendenter drept;

if ((st[i]==0) && (dr[i]==0)) : atunci când trebuie prelucrate nodurile cu doi descendenti;

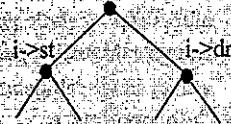
**Problema 2 rezolvată integral.**

```

#include<stdio.h>
#include<conio.h>
#include<iostream.h>
typedef int sir[30];
int rad, n, i;
sir info, st, dr;
void afis_2( int i)
{
    if (i % 2==0) cout<<info[i]<<" ";
    if (st[i]==0) afis_2(st[i]);
    if (dr[i]!=0) afis_2(dr[i]);
}
main()
{
    clrscr();
    cout<<"n="; cin>>n;
    for (i=1;i<=n;i++)
    {
        cout<<"informatiile despre nodul "<<i<<endl;
        cout<<info["<<i<<"]="; cin>>info[i];
        cout<<"st["<<i<<"]="; cin>>st[i];
        cout<<"dr["<<i<<"]="; cin>>dr[i];
    }
    cout<<"dati radacina : "; cin>>rad;
    afis_2(rad);
}
  
```

**Arborele este reprezentat folosind alocare dinamică.**

**Observație.** În continuare, vom prezenta schema care stă la baza realizării funcțiilor de prelucrare, sau analiză, a tuturor nodurilor unui arbore binar, cu rădăcina i și reprezentarea grafică:



atunci când analiza, sau prelucrarea, nodului i nu influențează prelucrarea sau analiza nodurilor din subarborele stâng sau subarborele drept.

Funcția are numele ...dorit și un parametru formal i de tip nod\* (pointer), care reprezintă rădăcina subarborelui prelucrat, și procedează astfel:

- dacă se verifică condițiile referitoare la nodul i,  
atunci .....  
altfel .....
- dacă există subarborele stâng, (de cele mai multe ori se prelucrează)  
atunci .....  
altfel .....
- dacă există subarborele drept, (de cele mai multe ori se prelucrează)  
atunci .....  
altfel .....

void prelucrare(nod \*i)

```

{
    if(.....nodul i verifică.....)
        .....
    else
        if(i->st!=0)      (asta înseamnă "dacă subarborele stâng există")
            .....
        else
            if(i->dr!=0)  (asta înseamnă "dacă subarborele drept există")
                .....
            else
}
  
```

**Observații.** 1. Funcțiile, realizate după schema de mai sus, sunt valabile atunci când arborele este reprezentat folosind alocare dinamică.

2. Ramurile else, de la if-urile din schemă, pot lipsi în unele situații.

- 3. if(..... nodul i verifică.....) se poate traduce în:
- if (i->notat % 2==0) : atunci când trebuie prelucrate nodurile cu numere de ordine pare;
- if (i->info % 2==0) : atunci când trebuie prelucrate nodurile cu informații pare;
- if (i->st==0) : atunci când trebuie prelucrate nodurile fără descendenter stâng;
- if (i->dr==0) : atunci când trebuie prelucrate nodurile fără descendenter drept;
- if ((i->st!=0) && (i->dr!=0)) : atunci când trebuie prelucrate nodurile cu doi descendenti;

**Problema 2 rezolvată integral.**

```

#include<stdio.h>
#include<conio.h>
#include<iostream.h>
struct nod {
    int notat;
    int info;
    nod *st, *dr;
};
nod *rad;
void creare(nod *rad)
{
    int val, nott;
    cout<<"notare="; cin>>nott;
    if (nott!=0) {
        cin>>val;
        rad=new nod;
        rad->info=val;
        rad->notat=nott;
        creare(rad->st);
        creare(rad->dr);
    }
    else
        rad=0;
}
  
```

crează arborele cu radacina rad

(val=valoare; nott=notare)  
se citește numărul de ordine  
dacă este diferit de 0,  
se citește informația  
se alocă spațiu pentru rad  
se încarcă informația citită  
și numărul de ordine  
se creează subarborele stâng  
se creează subarborele drept

altfel  
rad primește valoarea 0  
(se renunță la creare)

```

void afis_2(nod *i)
{
    if (i->notat % 2==0) cout<<i->info<<" ";
    if (i->st!=0) afis_2(i->st);
    if (i->dr!=0) afis_2(i->dr);
}

main()
{
    clrscr();
    creare(rad);
    afis_2(rad);
    getch();
}

```

**Problema 3**

```

void afis_3(int i)
{
    if (info[i] % 2==0) cout<<i<<" ";
    if (st[i]!=0) afis_3(st[i]);
    if (dr[i]!=0) afis_3(dr[i]);
}

```

**Problema 4**

```

void afis_4(int i)
{
    if ((st[i]!=0) && (dr[i]!=0)) cout<<i<<" ";
    if (st[i]!=0) afis_4(st[i]);
    if (dr[i]!=0) afis_4(dr[i]);
}

```

**Problema 5**

```

void afis_5(int i)
{
    if ((dr[i]==0) && (st[i]==0)) cout<<i<<" ";
    if (st[i]!=0) afis_5(st[i]);
    if (dr[i]!=0) afis_5(dr[i]);
}

```

**Problema 6**

```

void afis_6(int i)
{
    if ((dr[i]!=0) && (st[i]!=0)) cout<<i<<" ";
    if (st[i]!=0) afis_6(st[i]);
    if (dr[i]!=0) afis_6(dr[i]);
}

```

**Problema 7**

```

void afis_7(int i)
{
    if ((dr[i]==0) && (st[i]!=0)) cout<<info[i]<<" ";
    if (st[i]!=0) afis_7(st[i]);
    if (dr[i]!=0) afis_7(dr[i]);
}

```

**Problema 8**

```

void afis_8(int i)
{
    if ((dr[i]!=0) && (st[i]==0)) cout<<info[i]<<" ";
    if (st[i]!=0) afis_8(st[i]);
    if (dr[i]!=0) afis_8(dr[i]);
}

```

**Problema 9**

```

void afis_9(int i)
{
    if ((dr[i]==0) && (st[i]==0)) cout<<i<<" ";
    if (st[i]!=0) afis_9(st[i]);
    if (dr[i]!=0) afis_9(dr[i]);
}

```

**Problema 10**

```

void afis_10(int i)
{
    if ((dr[i]==0) && (st[i]==0)) cout<<info[i]<<" ";
    if (st[i]!=0) afis_10(st[i]);
    if (dr[i]!=0) afis_10(dr[i]);
}

```

**Problema 11**

```

void afis_11(int i)
{
    if (i==nod) {
        if (st[i]!=0) cout<<st[i]<<" ";
        else cout<<"stangul nu exista";
        if (dr[i]!=0) cout<<dr[i]<<" ";
        else cout<<" dreptul nu exista";
    }
    else {
        if (st[i]!=0) afis_11(st[i]);
        if (dr[i]!=0) afis_11(dr[i]);
    }
}

```

**Problema 12**

```
void afis_12(int i)
{
    if (i==nod) {
        if (st[i]!=0) cout<<st[i]<<" ";
        if (dr[i]!=0) cout<<dr[i]<<" ";
    }
    else {
        if (st[i]!=0) afis_12(st[i]);
        if (dr[i]!=0) afis_12(dr[i]);
    }
}
```

**Problema 13**

```
void afis_13(int i)
{
    if (i==nod) {
        if (st[i]!=0) cout<<st[i]<<" ";
        else cout<<"stangul nu exista ";
    }
    else {
        if (st[i]!=0) afis_13(st[i]);
        if (dr[i]!=0) afis_13(dr[i]);
    }
}
```

**Problema 14**

```
void afis_14(int i)
{
    if (i==nod) {
        if (dr[i]!=0) cout<<dr[i]<<" ";
        else cout<<"dreptul nu exista ";
    }
    else {
        if (st[i]!=0) afis_14(st[i]);
        if (dr[i]!=0) afis_14(dr[i]);
    }
}
```

**Problema 15**

```
void afis_15(int i)
{
    if ((st[i]==nod) || (dr[i]==nod))
        cout<<<<" ";
```

```
else {
    if (st[i]!=0) afis_15(st[i]);
    if (dr[i]!=0) afis_15(dr[i]);
}
```

**Problema 16**

**Comentarii:** Funcția are trei parametrii:

i : este rădăcina arborelui în care se face căutarea;  
nod : reprezintă nodul căutat;  
nivel : reprezintă nivelul la care s-a ajuns în arbore; acesta la apelul funcției este 1;  
și procedează astfel:  
dacă rădăcina arborelui este egală cu nodul căutat,  
atunci se afișează nivelul rădăcini (nivelul la care s-a ajuns)  
altfel  
dacă descendantul stâng există,  
se cauță în subarborele care-l admite ca rădăcină, dar nivelul este mărit cu 1;  
dacă descendantul drept există,  
se cauță în subarborele care-l admite ca rădăcină, dar nivelul este mărit cu 1.

void afis\_16(int i, int nod, int nivel)

```
{
    if (i==nod) cout<<nivel;
    else {
        if (st[i]!=0) afis_16(st[i], nod, nivel+1);
        if (dr[i]!=0) afis_16(dr[i], nod, nivel+1);
    }
}
```

Apel: afis\_20(rad, nod, 1);

**Problema 17**

**Comentarii:** Funcția are trei parametrii:

i : este rădăcina arborelui în care se face căutarea;  
nivel : reprezintă nivelul la care s-a ajuns în arbore; acesta la apelul funcției este 1;  
p : reprezintă nivelul care interesează, din arbore;  
și procedează astfel:

dacă nivelul curent este egal cu p,  
atunci se afișează nodul de pe acest nivel (se afișează rădăcina subarborelui)

altfel  
 dacă **descendentul stâng există**,  
 se caută în subarborele care-l admite ca rădăcină, dar nivelul  
 este mărit cu 1;  
 dacă **descendentul drept există**,  
 se caută în subarborele care-l admite ca rădăcină, dar nivelul  
 este mărit cu 1.

```
void afis_17(int i, int nivel, int p)
{
    if (nivel==p) cout<<i<<" ";
    else {
        if (st[i]!=0) afis_17(st[i], nivel+1, p);
        if (dr[i]!=0) afis_17(dr[i], nivel+1, p);
    }
}
```

Apel: afis\_17(rad, 1, p);

**Problema 18**

```
void afis_18(int i, int nivel, int p)
{
    if (nivel==p) cout<<info[i]<<" ";
    else {
        if (st[i]!=0) afis_18(st[i], nivel+1, p);
        if (dr[i]!=0) afis_18(dr[i], nivel+1, p);
    }
}
```

Apel: afis\_18(rad, 1, p);

**Problema 19**

```
void afis_19(int i)
{
    if (info[i]==val) cout<<i<<" ";
    if (st[i]!=0) afis_19(st[i]);
    if (dr[i]!=0) afis_19(dr[i]);
}
```

**Problema 20**

```
void afis_20(int i, int nivel, int p)
{
    if (nivel==p) {
        if (info[i]==val)
            cout<<i<<" ";
    }
    else {
        if (st[i]!=0) afis_20(st[i], nivel+1, p);
    }
}
```

```
    if (dr[i]!=0) afis_20(dr[i], nivel+1, p);
}
```

Apel: afis\_20(rad, 1, p);

**Problema 21**

```
void afis_21(int i)
{
    if ((st[i]==0) && (dr[i]==0)) {
        if (info[i]==val) cout<<i<<" ";
    }
    else {
        if (st[i]!=0) afis_21(st[i]);
        if (dr[i]!=0) afis_21(dr[i]);
    }
}
```

**3.3.3.2. Numărare**

**Se procedează astfel:**

- dacă subarborele stâng există, atunci se calculează numărul nodurilor sale (**ns**), altfel acesta este 0 (**ns=0**);
- dacă subarborele drept există, atunci se calculează numărul nodurilor sale (**nd**), altfel acesta este 0 (**nd=0**);
- **numărul nodurilor care verifică condiția este egal cu suma dintre:**  
     1 (care corespunde rădăcinii), **ns** și **nd**      dacă rădăcina verifică condiția, și  
     **ns** și **nd**      dacă rădăcina nu verifică condiția.

**Problema 1**

```
int numar_1(int i)
{
    int ns, nd;
    if (st[i]!=0) ns=numar_1(st[i]);
    else ns=0;
    if (dr[i]!=0) nd=numar_1(dr[i]);
    else nd=0;
    return (1+ns+nd);
}
```

**Problema 2**

```
int numar_2(int i)
{
    int ns, nd;
    if (st[i]!=0) ns=numar_2(st[i]);
    else ns=0;
    if (dr[i]!=0) nd=numar_2(dr[i]);
    else nd=0;
}
```

```

if (i % 2==0) return (1+ns+nd);
else return (ns+nd);
}

```

**rădăcina nu se numără, deoarece  
nu verifică condiția din enunț**

**Problema 3**

```

int numar_3(int i)
{
    int ns, nd;
    if (st[i]!=0) ns=numar_3(st[i]);
    else ns=0;
    if (dr[i]!=0) nd=numar_3(dr[i]);
    else nd=0;
    if (info[i] % 2==0) return (1+ns+nd);
    else return (ns+nd);
}

```

**rădăcina nu se numără,  
deoarece nu verifică condiția**

**Problema 4**

```

int numar_4(int i, int nod)
{
    int nr, ns, nd;
    if (i==nod) {
        nr=0;
        if (st[i]!=0) nr=nr+1;
        if (dr[i]!=0) nr=nr+1;
    }
    else nr=0;
    if (st[i]!=0) ns=numar_4(st[i],nod);
    else ns=0;
    if (dr[i]!=0) nd=numar_4(dr[i],nod);
    else nd=0;
    return (nr+ns+nd);
}

```

**dacă există, se numără desc. stâng  
dacă există, se numără desc. drept**

**Problema 5**

```

int numar_5(int i)
{
    int ns, nd;
    if ((st[i]==0) && (dr[i]==0))
        return 1;
    else {
        if (st[i]!=0) ns=numar_5(st[i]);
        else ns=0;
    }
}

```

```

if (dr[i]!=0) nd=numar_5(dr[i]);
else nd=0;
return ns+nd;
}

```

**Problema 6**

```

int numar_6(int i)
{
    int ns, nd;
    if (st[i]!=0) ns=numar_6(st[i]);
    else ns=0;
    if (dr[i]!=0) nd=numar_6(dr[i]);
    else nd=0;
    if ((st[i]!=0) && (dr[i]!=0))
        return (1+ns+nd);
    else return (ns+nd);
}

```

**Problema 7**

```

int numar_7(int i)
{
    int ns, nd;
    if (st[i]!=0) ns=numar_7(st[i]);
    else ns=0;
    if (dr[i]!=0) nd=numar_7(dr[i]);
    else nd=0;
    if ((st[i]!=0) && (dr[i]==0))
        return (1+ns+nd);
    else return (ns+nd);
}

```

**Problema 8**

**Se procedează astfel:**

- dacă subarborele stâng există, atunci se calculează numărul nivelelor sale (ns), altfel acesta este 0 (ns=0);
- dacă subarborele drept există, atunci se calculează numărul nivelelor sale (nd), altfel acesta este 0 (nd=0);
- numărul nivelelor din arbore este egal cu cel mai mare număr dintre ns și nd mărit cu 1 (1 corespunde nivelului rădăcinii).

```
int numar_8(int i)
```

```
{
    int ns, nd;
```

```

if (st[i]!=0) ns=numero_8(st[i]);
else ns=0;
if (dr[i]!=0) nd=numero_8(dr[i]);
else nd=0;
if (ns>nd) return (1+ns);
else return (1+nd);
}

```

**Problema 9**

**Comentarii:** Funcția are trei parametri:

**i** este rădăcina arborelui în care se face căutarea;  
**nivel** reprezintă nivelul la care s-a ajuns în arbore; acesta la apelul funcției este 1;

**p** reprezintă nivelul care interesează, din arbore;  
si procedează astfel:

- dacă nivelul curent este egal cu p,  
 atunci numărul este 1 (deoarece pe nivelul p se află rădăcina )
- altfel:
  - dacă **descendentul stâng există**,  
 - se numără în subarborele care-l admite ca rădăcină,  
 câte noduri sunt pe nivelul căutat (rezultatul este pus în ns);
  - dacă **descendentul drept există**,  
 - se numără în subarborele care-l admite ca rădăcină,  
 câte noduri sunt pe nivelul căutat (rezultatul este pus în nd);
  - numărul total de noduri de pe nivelul p este ns+nd.

```

int numero_9(int i, int nivel, int p)
{
    int ns, nd;
    if (nivel==p) return 1;
    else {
        if (st[i]!=0) ns=numero_9(st[i], nivel+1, p);
        else ns=0;
        if (dr[i]!=0) nd=numero_9(dr[i], nivel+1, p);
        else nd=0;
        return (ns+nd);
    }
}

```

**3.3.3.3. Sume**

**Observație.** Pentru a calcula suma informațiilor nodurilor, care verifică o condiție, dintr-un arbore binar, se procedează astfel:

- dacă subarborele stâng există, atunci se calculează suma (ss) pentru nodurile care verifică condiția și se găsesc în acest subarbore, altfel suma este 0 (ss=0);
- dacă subarborele drept există, atunci se calculează suma (sd) pentru nodurile care verifică condiția și se găsesc în acest subarbore, altfel suma este 0 (sd=0);
- suma totală este egală cu:  
 info[i]+ss+sd, dacă nodul i verifică condiția;  
 ss+sd, dacă nodul i nu verifică condiția.

**Problema 1**

```

int suma_1(int i)
{
    int ss, sd;
    if (st[i]!=0) ss=suma_1(st[i]);
    else ss=0;
    if (dr[i]!=0) sd=suma_1(dr[i]);
    else sd=0;
    return (info[i]+ss+sd);
}

```

**Problema 2**

```

int suma_2(int i)
{
    int ss, sd;
    if (st[i]!=0) ss=suma_2(st[i]);
    else ss=0;
    if (dr[i]!=0) sd=suma_2(dr[i]);
    else sd=0;
    if (i % 2==0) return (info[i]+ss+sd);
    else return (ss+sd);
}

```

**Problema 3**

```

int suma_3(int i)
{
    int ss, sd;
    if (st[i]!=0) ss=suma_3(st[i]);
    else ss=0;
    if (dr[i]!=0) sd=suma_3(dr[i]);
    else sd=0;
    if ((st[i]==0) && (dr[i]==0)) return (info[i]+ss+sd);
    else return (ss+sd);
}

```

**Problema 4**

```
int suma_4(int i)
{
    int ss, sd;
    if (st[i]!=0) ss=suma_4(st[i]);
    else ss=0;
    if (dr[i]!=0) sd=suma_4(dr[i]);
    else sd=0;
    if ((st[i]!=0) && (dr[i]==0)) return (info[i]+ss+sd);
    else return (ss+sd);
}
```

**Problema 5**

```
int suma_5(int i, int p, int nivel)
{
    int ss, sd;
    if (nivel==p) return(info[i]);
    else {
        if (st[i]!=0) ss=suma_5(st[i], p, nivel+1);
        else ss=0;
        if (dr[i]!=0) sd=suma_5(dr[i], p, nivel+1);
        else sd=0;
        return (ss+sd);
    }
}
```

Apel: suma=suma\_5(rad, p, 1);

**3.3.3.4. Produse**

**Observație.** Pentru a calcula produsul informațiilor nodurilor, care verifică o condiție, dintr-un arbore binar, se procedează astfel:

- dacă subarborele stâng există, atunci se calculează produsul (ps) pentru nodurile care verifică condiția și se găsesc în acest subarbore, altfel produsul este 1 (ps=1);
- dacă subarborele drept există, atunci se calculează produsul (pd) pentru nodurile care verifică condiția și se găsesc în acest subarbore, altfel produsul este 1 (pd=1);
- produsul total este egal cu:  
 info[i]\*ps\*pd, dacă nodul i verifică condiția;  
 ps\*pd, dacă nodul i nu verifică condiția.

**Problema 1**

```
int produs_1(int i)
{
    int ps, pd;
    if (st[i]!=0) ps=produs_1(st[i]);
    else ps=1;
```

```
if (dr[i]!=0) pd=produs_1(dr[i]);
else pd=1;
return (info[i]*ps*pd);
}
```

**Observație.** Celelalte funcții se realizează asemănător cu cele de la sume, cu următoarele modificări:

- funcția se va numi **produs\_i**, în loc de **suma\_i**;
- atunci când la sume rezultatele intermedii se adună la produse se **înmulțesc**.

**3.3.3.5. Verificări**

**Observație.** Pentru a verifica dacă **toate nodurile care ... ,** și se găsesc într-un arbore binar, verifică o condiție, se procedează astfel:

- se verifică dacă **toate nodurile care ... ,** și se găsesc în subarborele stâng verifică condiția (rezultatul este pus în **ok\_s**);
- se verifică dacă **toate nodurile care ... ,** și se găsesc în subarborele drept verifică condiția (rezultatul este pus în **ok\_d**);
- **răspunsul final este**
  - 1, dacă rădăcina, i, verifică condiția și **toate nodurile care ... ,** din subarborele stâng verifică condiția și **toate nodurile care ... ,** din subarborele drept verifică condiția;
  - 0, altfel.

*altfel spus:*

- **răspunsul final este**
  - 1, dacă **((nodul i ... verifică) și (ok\_s=1) și (ok\_d=1));**
  - 0, altfel.

**Problema 1**

```
int verifica_1(int i) {
    int ok_s, ok_d;
    if (st[i]!=0) ok_s=verifica_1(st[i]);
    else ok_s=1;
    if (dr[i]!=0) ok_d=verifica_1(dr[i]);
    else ok_d=1;
    if (info[i] % 2==0) return (ok_s && ok_d);
    else return 0;
}
```

**Problema 2**

```
int verifica_2(int i, int p, int nivel)
{
    int ok_s, ok_d;
    if (nivel==p) {
        if (info[i] % 2==0) return 1;
        else return 0;
    }
    else {
        if (st[i]!=0) ok_s=verifica_2(st[i], p, nivel+1);
        else ok_s=1;
        if (dr[i]!=0) ok_d=verifica_2(dr[i], p, nivel+1);
        else ok_d=1;
        return (ok_s && ok_d);
    }
}
```

**Problema 3**

```
int verifica_3(int i)
{
    int ok_s, ok_d;
    if ((st[i]==0) && (dr[i]==0)) {
        if (info[i] % 2==0) return 1;
        else return 0;
    }
    else {
        if (st[i]!=0) ok_s=verifica_3(st[i]);
        else ok_s=1;
        if (dr[i]!=0) ok_d=verifica_3(dr[i]);
        else ok_d=1;
        return (ok_s && ok_d);
    }
}
```

**Observație.** Pentru a verifica dacă cel puțin un nod care ...., dintr-un arbore binar, verifică o condiție, se procedează astfel:

– dacă rădăcina verifică condiția,

atunci răspunsul final este 1

altfel

– se verifică dacă cel puțin un nod care .... din subarborele stâng

verifică condiția (rezultatul este pus în ok\_s);

– se verifică dacă cel puțin un nod care .... din subarborele drept

verifică condiția (rezultatul este pus în ok\_d);

– răspunsul final este

– 1, dacă ((ok\_s=1) sau (ok\_d=1));

– 0, altfel.

**Problema 4**

```
int verifica_4(int i)
{
    int ok_s, ok_d;
    if (info[i] % 2==0) return 1;
    else {
        if (st[i]!=0) ok_s=verifica_4(st[i]);
        else ok_s=0;
        if (dr[i]!=0) ok_d=verifica_4(dr[i]);
        else ok_d=0;
        return (ok_s || ok_d);
    }
}
```

**Problema 5**

```
int verifica_5(int i, int p, int nivel)
{
    int ok_s, ok_d;
    if (nivel==p) {
        if (info[i] % 2==0) return 1;
        else return 0;
    }
    else {
        if (st[i]!=0) ok_s=verifica_5(st[i], p, nivel+1);
        else ok_s=0;
        if (dr[i]!=0) ok_d=verifica_5(dr[i], p, nivel+1);
        else ok_d=0;
        return (ok_s || ok_d);
    }
}
```

**Problema 6**

```
int verifica_6(int i)
{
    int ok_s, ok_d;
    if ((st[i]==0) && (dr[i]==0)) {
        if (info[i] % 2==0) return 1;
        else return 0;
    }
}
```

```

    else {
        if(st[i]!=0) ok_s=verifica_6(st[i]);
        else ok_s=0;
        if(dr[i]!=0) ok_d=verifica_6(dr[i]);
        else ok_d=0;
        return (ok_s || ok_d);
    }
}

```

**Observație.** Pentru a verifica dacă "toate nodurile care ... și se găsesc într-un arbore binar, nu verifică o condiție", se procedează ca la problemele în care se cere să se verifice dacă "toate nodurile care ... și se găsesc într-un arbore binar, verifică o condiție", numai că atunci când se face analiză unui element condiția este negată.

#### Problema 7

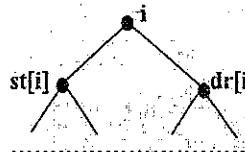
```

int verifica_7( int i)
{
    int ok_s, ok_d;
    if(st[i]!=0) ok_s=verifica_7(st[i]);
    else ok_s=1;
    if(dr[i]!=0) ok_d=verifica_7(dr[i]);
    else ok_d=1;
    if (!(info[i] % 2==0)) return (ok_s && ok_d);
    else return 0;
}

```

#### Problema 10

**Observație.** Pentru a verifica dacă un arbore binar, de forma,



este complet (adică toate nodurile neterminale au doi descendenti), se procedează astfel:

- dacă  $st[i]$  și  $dr[i]$  sunt 0, atunci înseamnă că arborele este format doar din nodul  $i$ , adică este complet, altfel dacă  $st[i]$  sau  $dr[i]$  este 0, atunci înseamnă că nodul  $i$ , neterminal, are decât un descendet, deci arborele nu este complet;
- altfel - se verifică dacă subarborele cu rădăcina  $st[i]$  este complet și rezultatul este pus în  $ok\_s$ ;
- se verifică dacă subarborele cu rădăcina  $dr[i]$  este complet și rezultatul este pus în  $ok\_d$ ;

- rezultatul final este:

1, dacă  $ok\_s$  și  $ok\_d$   
0, altfel:

```

int verifica_10(int i)
{
    int ok_s, ok_d;
    if ((st[i]==0) && (dr[i]==0)) return 1;
    else if ((st[i]==0) || (dr[i]==0)) return 0;
    else {
        ok_s=verifica_10(st[i]);
        ok_d=verifica_10(dr[i]);
        return (ok_s && ok_d);
    }
}

```

#### Problema 11

```

int verifica_11(int i, int val)
{
    int ok_s, ok_d;
    if (info[i]==val) return 1;
    else {
        if(st[i]!=0) ok_s=verifica_11(st[i],val);
        else ok_s=0;
        if (dr[i]!=0) ok_d=verifica_11(dr[i],val);
        else ok_d=0;
        return (ok_s || ok_d);
    }
}

```

#### Problema 12

```

int verifica_12(int i, int p, int nivel){
    int ok_s, ok_d;
    if (nivel==p) {
        if (info[i]==val) return 1;
        else return 0;
    }
    else {
        if(st[i]!=0) ok_s=verifica_12(st[i], p, nivel+1);
        else ok_s=0;
        if (dr[i]!=0) ok_d=verifica_12(dr[i], p, nivel+1);
        else ok_d=0;
        return (ok_s || ok_d);
    }
}

```

Apel: ok=verifica\_12(rad,p,1);

**Problema 13**

```
int verifica_13(int i){
    int ok_s, ok_d;
    if ((st[i]==0) && (dr[i]==0)) {
        if (info[i]==val) return 1;
        else return 0;
    }
    else {
        if (st[i]!=0) ok_s=verifica_13(st[i]);
        else ok_s=0;
        if (dr[i]!=0) ok_d=verifica_13(dr[i]);
        else ok_d=0;
        return (ok_s || ok_d)
    }
}
```

**3.3.3.6. Maxime****problemă 1**

```
int max_1(int i){
    int max_s, max_d, maxim;
    if (st[i]!=0) max_s=max_1(st[i]);
    else max_s=-1;
    if (dr[i]!=0) max_d=max_1(dr[i]);
    else max_d=-1;

    maxim=info[i];
    if (maxim<max_s) maxim=max_s;
    if (maxim<max_d) maxim=max_d;
    return maxim;
}
```

o valoare foarte mică

**problemă 2**

```
int max_2(int i){
    int max_s, max_d, maxim;
    if (st[i]!=0) max_s=max_2(st[i]);
    else max_s=-1;
    if (dr[i]!=0) max_d=max_2(dr[i]);
    else max_d=-1;

    if (i % 2==0) maxim=info[i];
    else maxim=-1;
    if (maxim<max_s) maxim=max_s;
    if (maxim<max_d) maxim=max_d;
    return maxim;
}
```

**problemă 3**

```
int max_3(int i)
{
    int max_s, max_d, maxim;
    if (st[i]!=0) max_s=max_3(st[i]);
    else max_s=-1;
    if (dr[i]!=0) max_d=max_3(dr[i]);
    else max_d=-1;
    if (info[i] % 2==0) maxim=info[i];
    else maxim=-1;
    if (maxim<max_s) maxim=max_s;
    if (maxim<max_d) maxim=max_d;
    return maxim;
}
```

**Observație.** Pentru a determina minimul elementelor, care verifică o condiție, dintr-un arbore binar, se procedează la fel ca la maxime, numai că în acest caz pentru determinarea minimelor din cei doi subarbore vom folosi notatiile **min\_s** și **min\_d** iar **minimul final este cel mai mic număr** dintre: **(min\_s)** și **(min\_d)** și **(informația nodului i, care se ia în calcul)**.

**3.3.4. Arbori binari de căutare****Problema 2**

Se procedează astfel:

- se citește sirul  $x_1, x_2, \dots, x_n$ ;
- toate elementele sirului, citit mai sus, se adaugă într-un arbore binar de căutare, astfel:
  - se pleacă de la arborele binar de căutare vid;
  - pentru  $i=1 \dots n$ 
    - se adaugă la arborele elementul  $x_i$  (pentru aceasta este nevoie de funcția **Adauga**, prezentată în secțiunea 3.1.6.)
- se afișează lista nodurilor, prin aplicarea funcției **Inordine**.

main()

clrscr();

```

cout<<"n="; cin>>n;
for (i=1;i<=n;i++)
    cin>>x[i];
rad=0;
for (i=1;i<=n;i++)
    adauga(rad, x[i]);
inordine(rad);
getche();
}

```

**Problema 3**

a)

**Observație.** Valoarea celei mai mari chei se află în cel mai din dreapta nod al subarborelui drept al rădăcinii. De aceea, pentru a o determina se procedează în felul următor:

Se construiește funcția cu numele max, caracterizată astfel:  
 – are un parametru, nod \*rad, care reprezintă rădăcina arborelui în care se face căutarea;  
 – returnează o valoare întreagă;  
 și procedează astfel:  
 dacă rădăcina nu are descendenter drept,  
 atunci înseamnă că ea este nodul cel mai din dreapta, și deci cea mai mare cheie din arbore este cheia sa;  
 altfel, cea mai mare cheie din arbore este cheia determinată pe același principiu din subarborele cu rădăcina descendental său drept (rad->dr).

```

int max( nod *rad)
{
    if (rad->dr==0) return rad->cheie;
    else
        return max(rad->dr);
}
b)
int min( nod *rad)
{
    if (rad->st==0) return rad->cheie;
    else
        return min(rad->st);
}

```

**Problema 4**

Problema se face într-un mod asemănător cu problema prezentată în secțiunea 3.1.6., numai că, în situația problemei de față, structura nodurilor este:

```

typedef char string[100];
struct nod{
    int cod;           (este câmpul cheie)
    string nume_autor;
    string titlu;
    string editura;
    nod *st, *dr;
};

```

iar referitor la funcțiile pe care le folosim, facem următoarele comentarii:

la adaugare : pentru nodul care se adaugă, se citesc toate câmpurile din structură;  
 la preordine : se afișează fie toate câmpurile, fie numai cele care interesează (după caz).

**3.3.5. Probleme diverse****Problema 1**

Pentru a demonstra echivalența cerută de problemă, este suficient să demonstreăm că:  $a \Leftrightarrow b$  și  $a \Leftrightarrow c$ , deoarece, din tranzitivitatea relației de echivalență, rezultă și  $b \Leftrightarrow c$ .

A demonstra  $a \Rightarrow b$  înseamnă a demonstra  $a \Rightarrow b$  și  $b \Rightarrow a$   
 $"a \Rightarrow b"$ .

Știind că "G este arbore", trebuie demonstrat că "G este conex, minimal cu această proprietate".

Proprietatea de conexitate este comună ipotezei și concluziei ( $\text{arbore} = \text{conex și fără cicluri}$ ), deci rămâne de demonstrat că "G este minimal în raport cu această proprietate".

Pentru aceasta procedăm astfel:

Presupunem că G nu este minimal în raport cu proprietatea de conexitate, adică, există în G două noduri x și y, astfel încât după eliminarea muchiei [x,y] graful obținut, fie acesta  $G_1$ , să fie tot conex. Altfel spus, să existe în  $G_1$  lanțul  $L=[x, x_1, \dots, y]$ , ca în figura:



Dacă adăugăm în  $G_1$  muchia eliminată [x,y], pentru a ne situa în G, se obține ciclul  $C=[x, x_1, \dots, y, x]$ . Contradicție, deoarece  $G_1$ , din ipoteză, este arbore și deci nu are cicluri. S-a ajuns la contradicție pentru că am presupus că G nu este minimal în raport cu proprietatea de conexitate.

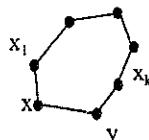
"b) $\Rightarrow$ a)".

Știind că "G este conex, minimal cu această proprietate", trebuie demonstrat că "G este arbore".

Proprietatea de conexitate este comună ipotezei și concluziei (arbore=conex și fără cicluri), deci rămâne de demonstrat că "G este fără cicluri".

Pentru aceasta procedăm astfel:

Presupunem că G conține un ciclu, fie acesta  $C=[x, x_1, \dots, x_k, y, x]$  ca în figura



Dacă eliminăm, în G, muchia [x,y], conform ipotezei: "G minimal în raport cu proprietatea de conexitate", am obține un graf, fie acesta  $G_1$ , care nu este conex, adică, în  $G_1$  nu există lanț de la x la y. Fals, pentru că, oricum, după eliminarea muchiei [x,y] ar exista în noul graf lanțul  $L=[x, x_1, \dots, x_k, y]$ . Greșala a provenit datorită faptului că am presupus că G conține un ciclu.

A demonstra  $a) \Leftrightarrow c)$  înseamnă a demonstra  $a) \Rightarrow c)$  și  $c) \Rightarrow a)$ .

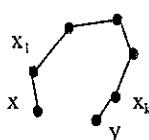
"a) $\Rightarrow$ c)".

Știind că "G este arbore", trebuie demonstrat că "G este fără cicluri, maximal cu această proprietate".

Proprietatea "G este fără cicluri", este comună ipotezei și concluziei (arbore=conex și fără cicluri), deci rămâne de demonstrat că "G este maximal în raport cu această proprietate".

Pentru aceasta procedăm astfel:

Alegem în G, la întâmplare, două noduri x și y neadiacente. Deoarece, conform ipotezei, G este arbore, adică conex, înseamnă că, în G, există lanțul  $L=[x, x_1, \dots, x_k, y]$ .



Dacă adăugăm muchia [x,y], se obține ciclul  $C=[x, x_1, \dots, x_k, y, x]$ , deci G este fără cicluri, maximal.

"c) $\Rightarrow$ a)".

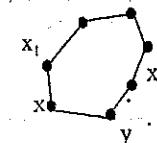
Știind că "G este fără cicluri, maximal cu această proprietate", trebuie demonstrat că "G este arbore".

Proprietatea "G este fără cicluri", este comună ipotezei și concluziei (arbore=conex și fără cicluri), deci rămâne de demonstrat că "G este conex".

Pentru aceasta procedăm astfel:

Presupunem că G nu este conex. Deci, există în G două noduri x și y neadiacente care nu sunt unite printr-un lanț. Dacă se adaugă muchia [x,y] se obține, conform ipotezei

"G este fără cicluri, maximal cu această proprietate", un graf, fie acesta  $G_1$ , în care se formează un ciclu  $C=[x, x_1, \dots, x_k, y, x]$



Dacă eliminăm, în  $G_1$ , muchia [x,y], ne plasăm în G și obținem  $L=[x, x_1, \dots, x_k, y]$ . Fals, Contradicția provine de la faptul că am presupus că G nu este conex.

**Problema 2**

Pentru a demonstra echivalența cerută de problemă, este suficient să demonstreăm că:  $a) \Leftrightarrow b)$  și  $a) \Leftrightarrow c)$ , deoarece, din tranzitivitatea relației de echivalență, rezultă și  $b) \Leftrightarrow c)$ .

A demonstra:  $a) \Leftrightarrow b)$  înseamnă a demonstra  $a) \Rightarrow b)$  și  $b) \Rightarrow a)$   
 $"a) \Rightarrow b)"$ .

Știind că "G este arbore" trebuie demonstrat că "G este conex și are  $n-1$  muchii".

Implicația se demonstrează făcând apel la definiția arborelui și la teorema: "Orice arbore cu  $n$  vârfuri are  $n-1$  muchii", pe care am prezentat-o deja în secțiunea 3.1.1.

" $b) \Rightarrow a)$ ".

Știind că "G este conex și are  $n-1$  muchii", trebuie demonstrat că "G este arbore".

Deoarece G este conex, rezultă, conform unei teoreme pe care am prezentat-o în secțiunea 3.1.2., că el conține un arbore parțial, fie acesta A. Deoarece A s-a obținut prin eliminarea unor muchii din G, înseamnă că are  $n$  noduri și cum este arbore are  $n-1$  muchii, conform unei teoreme pe care am prezentat-o deja în secțiunea 3.1.1.. În concluzie:  $A=G$ , deci G este arbore.

A demonstra  $a) \Leftrightarrow c)$  înseamnă a demonstra  $a) \Rightarrow c)$  și  $c) \Rightarrow a)$   
 $"a) \Rightarrow c)"$ .

Știind că "G este arbore", trebuie demonstrat că "G este fără cicluri și are  $n-1$  muchii".

Implicația se demonstrează făcând apel la definiția arborelui și la teorema: "Orice arbore cu  $n$  vârfuri are  $n-1$  muchii", pe care am prezentat-o în secțiunea 3.1.1.

" $c) \Rightarrow a)$ ".

Știind că "G este fără cicluri și are  $n-1$  muchii", trebuie demonstrat că "G este arbore".

Proprietatea "G este fără cicluri", este comună ipotezei și concluziei (arbore=conex și fără cicluri), deci rămâne de demonstrat că "G este conex".

Pentru aceasta procedăm astfel:

Presupunem că G nu este conex. Deci, el este format din mai multe componente conexe și presupunem că este format din  $m > 1$  componente conexe. Deoarece G nu conține cicluri, înseamnă că nici componentele sale conexe nu conțin cicluri, deci sunt

arbori cu, să presupunem,  $n_1, n_2, \dots, n_m$  noduri și, în cazul acesta,  $n_1-1, n_2-1, \dots, n_m-1$  muchii. Cum numărul muchiilor lui G este egal cu suma numărului muchiilor componentelor sale conexe, rezultă că G are  $(n_1-1)+(n_2-1)+\dots+(n_m-1)=n-m$  muchii. Deoarece  $m>1$ , avem  $n-m<1$ , ceea ce este fals. Contradicția a provenit din faptul că am presupus că G nu este conex.

#### Problema 3

" $\Rightarrow$ " Se știe că "G conține un arbore parțial" și trebuie să demonstrăm că "G este conex".

Fie A un arbore parțial al lui G. Cum A este arbore parțial, înseamnă că este conex și are tot atâtea noduri câte are și G. Dacă se adaugă la A muchiile care au fost eliminate din G, pentru al obține (pentru că până la urmă el este un graf parțial și se obține prin eliminarea unor muchii), proprietatea de conexitate se păstrează. Deci G este conex.

" $\Leftarrow$ " Se știe că "G este conex" și trebuie să demonstrăm că "G conține un arbore parțial".

Deoarece G este conex, se pot întâlni situațiile:

1. G nu conține cicluri. În acest caz, G este arbore și alegem ca arbore parțial graful însuși, deci A=G.
2. G conține cicluri. În acest caz, se procedează astfel:

Se alege un ciclu C și eliminând o muchie din cadrul său se obține un graf  $G_1$  care este conex. Dacă graful  $G_1$  nu conține cicluri înseamnă că el este arbore și atunci îl alegem ca arbore parțial, adică  $A=G_1$ , altfel, se reia procesul. Oricum, după un număr finit de pași obținem arborele parțial căutat.

#### Problema 4

Numărul maxim de muchii pentru un graf cu  $n$  noduri și fără cicluri este  $n-1$ . Proprietatea de a fi fără cicluri dispără atunci când graful are cel puțin  $n$  muchii.

#### Problema 5

În prima fază, demonstrăm prin inducție că "pe nivelul k sunt  $2^{k-1}$  noduri".

I Verificare:

Dacă  $k=1$ , sunt  $2^{k-1}=2^{1-1}=2^0=1$  noduri (adevărat).

II Demonstrația:  $P(k) \Rightarrow P(k+1)$

$P(k)$  : "Pe nivelul k sunt  $2^{k-1}$  noduri" (adevărat)

$P(k+1)$  : "Pe nivelul k+1 sunt  $2^k$  noduri"

Cu alte cuvinte, dacă pe nivelul k sunt  $2^{k-1}$  noduri, să se demonstreze că pe nivelul k+1 sunt  $2^{(k+1)-1}$  noduri.

Deoarece fiecare nod de pe nivelul k are doi descendenti pe nivelul k+1, înseamnă că pe nivelul k+1 se află 2 înmulțit cu numărul nodurilor de pe nivelul k noduri, adică  $2 \cdot 2^{k-1} = 2^k$  noduri (Am terminat demonstrația).

Presupunem că arborele are p nivele. Numărul nodurilor sale se determină adunând numărul nodurilor de pe fiecare nivel, de la 1 până la p, adică:

$$2^{1-1} + 2^{2-1} + \dots + 2^{p-1} = 1 + 2 + 2^2 + \dots + 2^{p-1} = 2^p - 1 = 2 \cdot 2^{p-1} - 1 = 2^n - 1$$

Am folosit faptul că pe ultimul nivel are  $n$  noduri, adică  $n=2^{p-1}$ .

## GRAFURI ORIENTATE

### 4.1. ASPECTE TEORETICE

#### 4.1.1. Noțiunea de graf orientat

**Definiție.** Se numește **graf orientat** o pereche ordonată de mulțimi notată  $G=(V, U)$ , unde:

$V$  : este o mulțime, **finită** și **nevidă**, ale cărei elemente se numesc **noduri** sau **vârfuri**;

$U$  : este o mulțime, de **perechi ordonate de elemente distincte** din  $V$ , ale cărei elemente se numesc **arce**.

#### ♦ Exemplu de graf orientat:

$G=(V, U)$  unde:  $V=\{1,2,3,4\}$   
 $U=\{(1,2), (2,3), (1,4)\}$

#### Demonstratie:

Perechea G este graf orientat deoarece respectă întocmai definiția prezentată mai sus, adică:

$V$  : este finită și nevidă;

$U$  : este o mulțime de perechi ordonate de elemente din  $V$ .

În continuare, vom nota submulțimea  $\{(x,y)\}$ , care reprezintă un arc, cu  $(x,y)$  (într-un graf orientat arcul  $(x,y)$  este diferit de arcul  $(y,x)$ ). În baza celor spuse anterior, graful prezentat în exemplul de mai sus se reprezintă textual astfel:

$G=(V, U)$  unde:  $V=\{1,2,3,4\}$   
 $U=\{(1,2), (2,3), (1,4)\}$

În teoria grafurilor orientate se întâlnesc frecvent noțiunile:

#### - extremitățile unui arc

- fiind dat arcul  $u=(x,y)$ , se numesc extremități ale sale **nodurile x și y**;  
 $x$  se numește **extremitate initială**;  
 $y$  se numește **extremitate finală**.

#### - vârfuri adiacente

- dacă într-un graf există arcul  $u=(x,y)$  (sau  $u=(y,x)$ , sau amândouă), se spune despre **nodurile x și y că sunt adiacente**;

## - incidentă

- dacă  $u_1$  și  $u_2$  sunt două arce ale aceluiași graf, se numesc incidente dacă au o extremitate comună.

Exemplu.  $u_1=(x,y)$  și  $u_2=(y,z)$  sunt incidente;

- dacă  $u_1=(x,y)$  este un arc într-un graf, se spune despre el și nodul  $x$ , sau nodul  $y$ , că sunt incidente.

Reprezentarea unui graf orientat admite două forme, și anume:

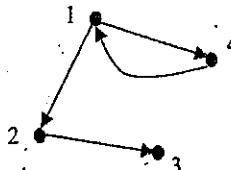
- reprezentare textuală: așa cum s-a reprezentat graful din exemplul anterior;
- reprezentare grafică: arcele sunt reprezentate prin săgeți orientate, iar nodurile prin puncte.

## ♦ Exemplu de graf orientat reprezentat textual:

$$G=(V, U) \text{ unde: } V=\{1, 2, 3, 4\}$$

$$U=\{(1,2), (2,3), (1,4), (4,1)\}$$

## ♦ Exemplu de graf orientat reprezentat grafic (este graful de la exemplul anterior):



## Alte definiții

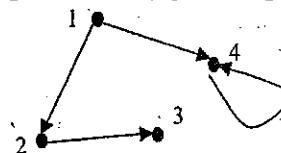
Definiție. Se numește graf orientat o pereche ordonată de multimi notată  $G=(V, U)$ , unde:

$V$  : este o mulțime, finită și nevidă, ale cărei elemente se numesc *noduri* sau *vârfuri*;

$U$  : este o mulțime, de *perechi ordonate de elemente din  $V$* , ale cărei elemente se numesc *arce*.

Această definiție diferă de prima definiție prin faptul că acum nu se mai spune despre *extremitățile unui arc* că trebuie să fie distincte. În baza acestei definiții, sunt permise și arce de genul:  $u=(x,x)$  unde  $x \in V$ ; aceste arce se numesc *bucle*.

## ♦ Exemplu de graf orientat (reprezentat grafic):



$$V=\{1, 2, 3, 4\}$$

$$U=\{(1,2), (2,3), (1,4), (4,1)\}$$

## GRAFURI ORIENTATE

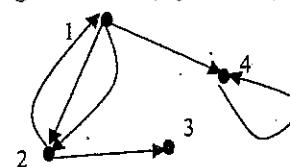
Definiție. Se numește graf orientat o pereche ordonată de multimi notată  $G=(V, U)$ , unde:

$V$  : este o mulțime, finită și nevidă, ale cărei elemente se numesc *noduri* sau *vârfuri*;

$U$  : este o familie de *perechi ordonate de elemente din  $V$* , numită *familia de arce*.

Această definiție diferă de cea anterioară prin faptul că acum nu numai că se admit bucle, dar se admit și mai multe arce identice.

## ♦ Exemplu de graf orientat (reprezentat grafic):



$$V=\{1, 2, 3, 4\}$$

$$U=\{(1,2), (1,2), (2,1), (1,4), (2,3), (4,4)\}$$

Observație. Dacă într-un graf orientat numărul arcelor identice nu depășește numărul  $p$ , atunci se numește  $p$ -graf.

## 4.1.2. Noțiunea de graf parțial

Definiție. Fie  $G=(V, U)$  un graf orientat. Se numește graf parțial, al grafului  $G$ , graful orientat  $G_1=(V, U_1)$  unde  $U_1 \subseteq U$ .

## ♦ Atenție! Citind cu atenție definiția, de mai sus, tragem concluzia:

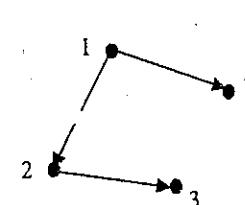
Un graf parțial, al unui graf orientat  $G=(V, U)$ , are aceeași mulțime de vârfuri ca și  $G$ , iar mulțimea arcelor este o submulțime a lui  $U$  sau chiar  $U$ .

## ♦ Exemplu: Fie graful orientat

$$G=(V, U) \text{ unde: } V=\{1, 2, 3, 4\}$$

$$U=\{(1,2), (1,4), (2,3)\}$$

reprezentat grafic astfel:

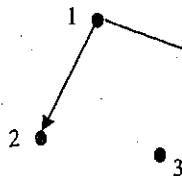


1. Un exemplu de graf parțial al grafului G este graful orientat:

$$G_1 = (V, U_1) \text{ unde: } V = \{1, 2, 3, 4\}$$

$$U_1 = \{(1,2), (1,4)\} \quad (\text{s-a eliminat arcul } (2,3))$$

reprezentat grafic astfel:

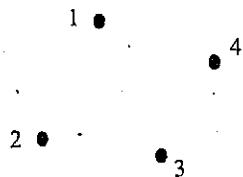


2. Un exemplu de graf parțial al grafului G este graful orientat:

$$G_1 = (V, U_1) \text{ unde: } V = \{1, 2, 3, 4\}$$

$$U_1 = \emptyset \quad (\text{s-au eliminat toate arcele})$$

reprezentat grafic astfel:



**Observație.** Fie  $G = (V, U)$  un graf orientat. Un graf parțial al grafului G, se obtine păstrând vârfurile și eliminând eventual niște arce (se pot elimina și toate arcele sau chiar nici unul).

### 4.1.3. Noțiunea de subgraf

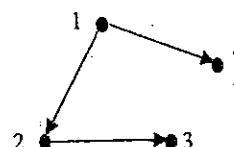
**Definiție.** Fie  $G = (V, U)$  un graf orientat. Se numește subgraf, al grafului G, graful orientat  $G_1 = (V_1, U_1)$  unde  $V_1 \subseteq V$  iar  $U_1$  conține toate arcele din U care au extremitățile în  $V_1$ .

♦ **Exemplu:** Fie graful orientat

$$G = (V, U) \text{ unde: } V = \{1, 2, 3, 4\}$$

$$U = \{(1,2), (2,3), (1,4)\}$$

reprezentat grafic astfel:



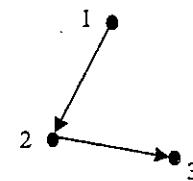
1. Un exemplu de subgraf al grafului G este graful orientat:

### GRAFURI ORIENTATE

$$G_1 = (V_1, U_1) \text{ unde: } V_1 = \{1, 2, 3\} \quad (\text{s-a sters nodul } 4)$$

$$U_1 = \{(1,2), (2,3)\} \quad (\text{s-a eliminat arcul } (1,4))$$

reprezentat grafic astfel:

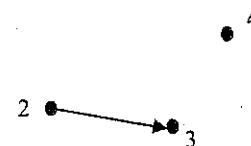


2. Un exemplu de subgraf al grafului G este graful orientat:

$$G_1 = (V_1, U_1) \text{ unde: } V_1 = \{2, 3, 4\} \quad (\text{s-a eliminat nodul } 1)$$

$$U_1 = \{(2,3)\} \quad (\text{s-au eliminat arcele } (1,4) \text{ și } (1,2))$$

reprezentat grafic astfel:



**Observație.** Fie  $G = (V, U)$  un graf orientat. Un subgraf, al grafului G, se obține sterând eventual anumite vârfuri și odată cu acestea și arcele care le admit ca extremitate (nu se pot sterge toate vârfurile deoarece s-ar obține un graf cu multimea vârfurilor vidă).

### 4.1.4. Gradul unui vârf

Având la bază ideea că "raportat la un vârf există arce care ies din acel vârf și arce care intră în acel vârf", au luat naștere următoarele noțiuni:

– grad exterior

– grad interior

care vor fi prezentate în continuare.

#### Grad exterior

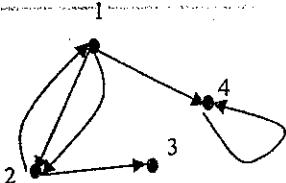
**Definiție.** Fie  $G = (V, U)$  un graf orientat și x un nod al său. Se numește grad exterior al nodului x, numărul arcelor de forma  $(x, y)$  (adică numărul arcelor care ies din x), notat  $d^+(x)$ .

♦ **Exemplu:** Fie graful orientat:

$$G = (V, U) \text{ unde: } V = \{1, 2, 3, 4\}$$

$$U = \{(1,2), (1,2), (2,1), (1,4), (2,3), (4,4)\}$$

reprezentat grafic astfel:



Gradul exterior al nodului 1 este  $d^+(1)=3$  (în graf, sunt trei arce care ies din 1).

Gradul exterior al nodului 2 este  $d^+(2)=2$  (în graf, sunt două arce care ies din 2).

Gradul exterior al nodului 3 este  $d^+(3)=0$  (în graf, nu sunt arce care ies din 3).

Gradul exterior al nodului 4 este  $d^+(4)=1$  (în graf, este un arc care ieșe din 4 (bucla)).

**Observații.** 1. Multimea succesorilor lui  $x$  se notează cu  $\Gamma^+(x)$  și se reprezintă astfel:

$$\Gamma^+(x) = \{y \in V / (x, y) \in U\}$$

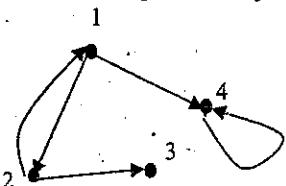
2. Multimea arcelor ce ies din  $x$  se notează cu  $\omega^+(x)$  și se reprezintă astfel:

$$\omega^+(x) = \{(x, y) \in U / y \in V\}$$

3. Legat de cardinalele multimilor  $\Gamma^+(x)$  și  $\omega^+(x)$ , putem scrie:

$$|\Gamma^+(x)| = |\omega^+(x)| = d^+(x)$$

Raportat la graful prezentat în figura de mai jos,



putem scrie:

$$\Gamma^+(1) = \{2, 4\}$$

$$\omega^+(1) = \{(1, 2), (1, 4)\}$$

$$|\Gamma^+(1)| = |\omega^+(1)| = d^+(1) = 2$$

$$\Gamma^+(2) = \{1, 3\}$$

$$\omega^+(2) = \{(2, 1), (2, 3)\}$$

$$|\Gamma^+(2)| = |\omega^+(2)| = d^+(2) = 2$$

$$\Gamma^+(3) = \emptyset$$

$$\omega^+(3) = \emptyset$$

$$|\Gamma^+(3)| = |\omega^+(3)| = d^+(3) = 0$$

$$\Gamma^+(4) = \{4\}$$

$$\omega^+(4) = \{(4, 4)\}$$

$$|\Gamma^+(4)| = |\omega^+(4)| = d^+(4) = 1$$

### Grad interior

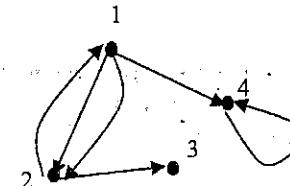
**Definiție.** Fie  $G=(V, U)$  un graf orientat și  $x$  un nod al său. Se numește **grad interior** al nodului  $x$ , numărul arcelor de formă  $(y, x)$  (adică numărul arcelor care intră în  $x$ ), notat  $d^-(x)$ .

♦ **Exemplu:** Fie graful orientat

$$G=(V, U) \text{ unde: } V=\{1, 2, 3, 4\}$$

$$U=\{(1, 2), (1, 2), (2, 1), (1, 4), (2, 3), (4, 4)\}$$

reprezentat grafic astfel:



Gradul interior al nodului 1 este  $d^-(1)=1$  (în graf, este un arc care intră în 1).

Gradul interior al nodului 2 este  $d^-(2)=2$  (în graf, sunt două arce care intră în 2).

Gradul interior al nodului 3 este  $d^-(3)=1$  (în graf, este un arc care intră în 3).

Gradul interior al nodului 4 este  $d^-(4)=2$  (în graf, sunt două arce care intră în 4).

**Observații.** 1. Multimea predecesorilor lui  $x$  se notează cu  $\Gamma^-(x)$  și se reprezintă astfel:

$$\Gamma^-(x) = \{y \in V / (y, x) \in U\}$$

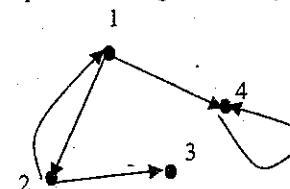
2. Multimea arcelor ce intră în  $x$  se notează cu  $\omega^-(x)$  și se reprezintă astfel:

$$\omega^-(x) = \{(y, x) \in U / y \in V\}$$

3. Legat de cardinalele multimilor  $\Gamma^-(x)$  și  $\omega^-(x)$ , putem scrie:

$$|\Gamma^-(x)| = |\omega^-(x)| = d^-(x)$$

Raportat la graful prezentat în figura de mai jos,



putem scrie:

$$\Gamma^-(1) = \{2\}$$

$$\omega^-(1) = \{(2, 1)\}$$

$$|\Gamma^-(1)| = |\omega^-(1)| = d^-(1) = 1$$

$$\Gamma^-(2) = \{1\}$$

$$\omega^-(2) = \{(1, 2)\}$$

$$|\Gamma^-(2)| = |\omega^-(2)| = d^-(2) = 1$$

$$\Gamma^-(3) = \{2\}$$

$$\omega^-(3) = \{(2, 3)\}$$

$$|\Gamma^-(3)| = |\omega^-(3)| = d^-(3) = 1$$

$$\Gamma^-(4) = \{1, 4\}$$

$$\omega^-(4) = \{(1, 4), (4, 4)\}$$

$$|\Gamma^-(4)| = |\omega^-(4)| = d^-(4) = 2$$

**Observație.** Un nod  $x$  se numește izolat dacă:

$$d^+(x) = d^-(x) = 0$$

**Propoziție.** În graful orientat  $G=(V, U)$ , în care  $V=\{x_1, x_2, \dots, x_n\}$  și sunt  $m$  arce,

$$\text{se verifică egalitatea: } \sum_{i=1}^n d^+(x_i) = \sum_{i=1}^n d^-(x_i) = m$$

### 4.1.5. Graf complet

**Definiție.** Fie  $G=(V, U)$  un graf orientat. Graful  $G$  se numește **graf complet** dacă oricare două vârfuri distincte ale sale sunt adjacente.

Două vârfuri  $x$  și  $y$  sunt adjacente dacă:

- între ele există arcul  $(x,y)$ , sau
- între ele există arcul  $(y,x)$ , sau
- între ele există arcele  $(x,y)$  și  $(y,x)$ .

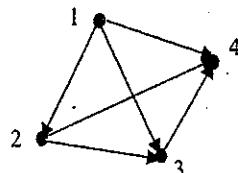
♦ **Exemplu** de graf orientat complet:

$G=(V, U)$  unde:

$$V=\{1,2,3,4\}$$

$$U=\{(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)\}$$

Reprezentarea sa grafică este:



**Propoziție.** Într-un graf complet cu  $n$  vârfuri, există între  $\frac{n(n-1)}{2}$  și  $n(n-1)$  arce.

( $U$  este privită ca o mulțime (prima definiție a grafului) nu ca o familie)

**Demonstrație:** Numărul cel mai mic de arce, când graful este complet, se obține atunci când nodurile sunt unite doar printr-un singur arc și se determină astfel:

Pentru fiecare vârf  $x_i$ , numărul arcelor care intră și ieșă este  $n-1$ , deci

$$d^+(x_i) + d^-(x_i) = n-1, \text{ pentru orice } i=1..n.$$

Însumând toate aceste relații, obținem:

$$\sum_{i=1}^n (d^+(x_i) + d^-(x_i)) = \sum_{i=1}^n (n-1) \Rightarrow \sum_{i=1}^n d^+(x_i) + \sum_{i=1}^n d^-(x_i) = n(n-1) \quad (1)$$

Tinând cont de faptul că:

$$\sum_{i=1}^n d^+(x_i) = \sum_{i=1}^n d^-(x_i) = m$$

relația (1) devine:

$$m + m = n(n-1) \Rightarrow m = \frac{n(n-1)}{2}$$

Numărul cel mai mare de arce, când graful este complet, se obține atunci când nodurile sunt unite prin două arce, adică pentru nodurile  $x$  și  $y$  există arcele  $(x,y)$  și  $(y,x)$ , și este egal cu

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

### GRAFURI ORIENTATE

### 4.1.6. Conexitate

În această secțiune, vor fi prezentate noțiunile:

- lanț
- drum
- circuit
- graf conex
- componentă conexă

#### Lanț

**Definiție.** Fie  $G=(V, U)$  un graf orientat. Se numește lanț, în graful  $G$ , o succesiune de arce, notată  $L = [u_{i_1}, u_{i_2}, \dots, u_{i_k}]$ , cu proprietatea că oricare două arce consecutive au o extremitate comună (nu are importanță orientarea arcelor).

Se întâlnesc noțiunile:

- **extremitățile lanțului**
  - fiind dat lanțul  $L = [u_{i_1}, u_{i_2}, \dots, u_{i_k}]$ , se numesc **extremități ale sale** extremitatea arcului  $u_{i_1}$  care nu este comună cu arcul  $u_{i_2}$ , și extremitatea arcului  $u_{i_k}$  care nu este comună cu arcul  $u_{i_{k-1}}$ ;
- **lungimea lanțului**
  - fiind dat lanțul  $L = [u_{i_1}, u_{i_2}, \dots, u_{i_k}]$ , prin lungimea sa se înțelege numărul de arce care apar în  $L$ ;

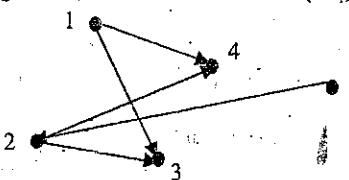
♦ **Exemplu de lanț:**

Fie graful  $G=(V, U)$ , unde:  $V=\{1,2,3,4,5\}$

$$U=\{(1,3), (1,4), (2,3), (2,4), (5,2)\}=$$

$$\{u_1, u_2, u_3, u_4, u_5\}$$

cu reprezentarea grafică astfel:



$L_1 = [u_1, u_3, u_5]$  este, în graful  $G$ , lanț cu lungimea 3 și extremitățile 1 și 5.

$L_2 = [u_1, u_2, u_4, u_5]$  este, în graful  $G$ , lanț cu lungimea 4 și extremitățile 3 și 5.

**Atenție!** Dacă  $L = [u_{i_1}, u_{i_2}, \dots, u_{i_k}]$  este lanț în graful  $G$ , atunci și

$L_1 = [u_{i_k}, u_{i_{k-1}}, \dots, u_{i_1}]$  este lanț în graful  $G$ .

## Drum

**Definiție.** Fie  $G=(V, U)$  un graf orientat. Se numește **drum**, în graful  $G$ , o succesiune de noduri, notată  $D = (x_{i_1}, x_{i_2}, \dots, x_{i_k})$ , cu proprietatea  $(x_{i_1}, x_{i_2}), \dots, (x_{i_{k-1}}, x_{i_k}) \in U$  (altfel spus,  $(x_{i_1}, x_{i_2}), \dots, (x_{i_{k-1}}, x_{i_k})$  sunt arce).

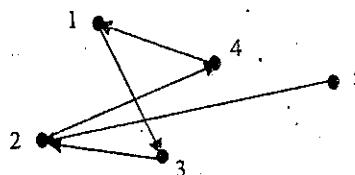
Se întâlnesc noțiunile:

- **extremitățile drumului**
  - fiind dat drumul  $D = (x_{i_1}, x_{i_2}, \dots, x_{i_k})$  se numesc extremități ale sale nodurile  $x_{i_1}$  și  $x_{i_k}$  ( $x_{i_1}$  – extremitate inițială;  $x_{i_k}$  – extremitate finală);
- **lungimea drumului**
  - fiind dat drumul  $D = (x_{i_1}, x_{i_2}, \dots, x_{i_k})$ , prin lungimea sa se înțelege numărul de arce care apar în cadrul său;

### ♦ Exemplu de drum:

Fie graful  $G=(V, U)$  unde:  $V=\{1,2,3,4,5\}$   
 $U=\{(1,3), (4,1), (3,2), (2,4), (5,2)\}$

cu reprezentarea grafică astfel:



$D1=(1, 3, 2)$  este, în graful  $G$ , drum cu lungimea 2 și extremitățile 1 și 2.

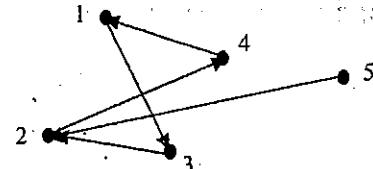
$D2=(4, 1, 3, 2)$  este, în graful  $G$ , drum cu lungimea 3 și extremitățile 4 și 2.

**Atenție!** Dacă  $D = (x_{i_1}, x_{i_2}, \dots, x_{i_k})$  este drum, în graful  $G$ , atunci nu neapărat și  $D1 = (x_{i_k}, x_{i_{k-1}}, \dots, x_{i_1})$  este drum, în graful  $G$ .

**Definiție.** Fie  $G=(V, U)$  un graf orientat. Se numește **drum elementar**, în graful  $G$ , drumul  $D = (x_{i_1}, x_{i_2}, \dots, x_{i_k})$  cu proprietatea că oricare două noduri ale sale sunt distincte (altfel spus, printr-un nod nu se trece decât o singură dată).

## GRAFURI ORIENTATE

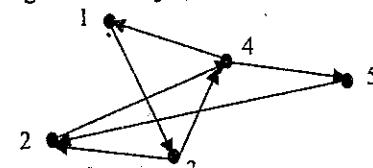
### ♦ Exemplu: În graful de mai jos,



drumul  $D1=(4, 1, 3, 2)$  este **drum elementar**,

**Definiție.** Fie  $G=(V, U)$  un graf orientat. Se numește **drum neelementar**, în graful  $G$ , drumul  $D = (x_{i_1}, x_{i_2}, \dots, x_{i_k})$  cu proprietatea că nodurile sale nu sunt distincte două către două (altfel spus, prin anumite noduri s-a trecut de mai multe ori).

### ♦ Exemplu: În graful de mai jos,

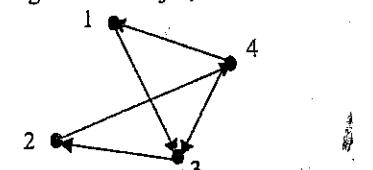


drumul  $D2=(4, 1, 3, 2, 4, 5, 2)$  este **drum neelementar** (prin 4 și 2 s-a trecut de două ori).

## Circuit

**Definiție.** Fie  $G=(V, U)$  un graf neorientat. Se numește **circuit**, în graful  $G$ , drumul  $D = (x_{i_1}, x_{i_2}, \dots, x_{i_k})$  cu proprietatea că  $x_{i_1} = x_{i_k}$  și are arcele cel compun diferite două către două (circuitul se notează în continuare cu  $C$ ).

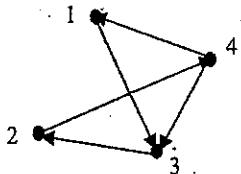
### ♦ Exemplu: În graful de mai jos,



drumul  $C=(1, 3, 2, 4, 1)$  este circuit.

**Definiție.** Fie  $G=(V, U)$  un graf orientat. Se numește **circuit elementar**, în graful  $G$ , un circuit cu proprietatea că oricare două noduri ale sale, cu excepția primului și a ultimului, sunt distincte.

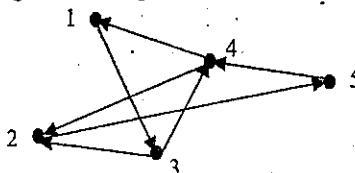
♦ Exemplu: În graful de mai jos,



circuitul  $C=(1, 3, 2, 4, 1)$  este circuit elementar.

**Definiție.** Fie  $G=(V, U)$  un graf orientat. Se numește circuit neelementar, în graful  $G$ , un circuit cu proprietatea că nodurile sale, cu excepția primului și a ultimului, nu sunt distințe.

♦ Exemplu: În graful de mai jos,

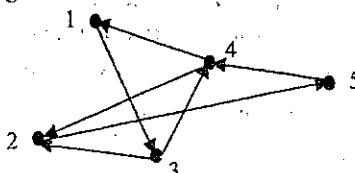


circuitul  $C=(1, 3, 4, 2, 5, 4, 1)$  este circuit neelementar (prin 4 s-a trecut de două ori).

### Graf conex

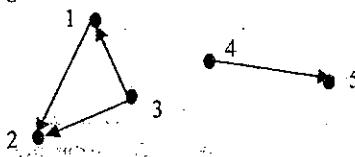
**Definiție.** Fie  $G=(V, U)$  un graf orientat. Graful  $G$  se numește conex dacă pentru oricare două vârfuri  $x$  și  $y$ ,  $x \neq y$ , există un lanț de extremități  $x$  și  $y$ .

♦ Exemplu de graf conex:



Graful este conex, deoarece oricare ar fi vârfurile  $x$  și  $y$ ,  $x \neq y$ , există un lanț în  $G$  care să le lege.

♦ Exemplu de graf care nu este conex:



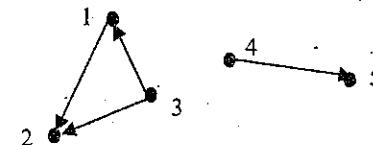
Graful nu este conex, deoarece există două vârfuri, cum ar fi 1 și 4, pentru care nu există nici un lanț în graf care să le lege.

### Componentă conexă

**Definiție.** Fie  $G=(V, U)$  un graf orientat. Se numește componentă conexă un graf orientat  $G_1=(V_1, U_1)$  care verifică următoarele condiții:

- este subgraf al grafului  $G$ ;
- este conex;
- nu există nici un lanț în  $G$  care să lege un nod din  $V_1$  cu un nod din  $V-V_1$ .

♦ Exemplu: Fie graful  $G=(V, U) : V=\{1,2,3,4,5\}$  și  $U=\{(1,2), (3,1), (3,2), (4,5)\}$



Pentru graful de mai sus, graful  $G_1=(V_1, U_1)$  unde:  $V_1=\{4,5\}$  și  $U_1=\{(4,5)\}$

este componentă conexă, deoarece:

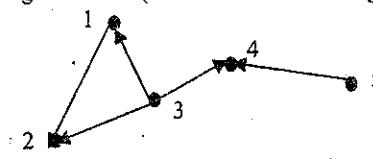
- este subgraf al grafului  $G$ ;
- este conex;
- nu există nici un lanț în  $G$  care să lege un nod din  $V_1$  cu un nod din  $V-V_1=\{1,2,3\}$ .

La fel, se poate spune și despre graful  $G_2=(V_2, U_2)$  unde:  $V_2=\{1,2,3\}$  și  $U_2=\{(1,2), (3,1), (3,2)\}$

În concluzie, graful din figura de mai sus este format din două componente conexe.

**Observație.** Fie  $G=(V, U)$  un graf orientat. Graful  $G$  este conex dacă și numai dacă este format dintr-o singură componentă conexă.

♦ Exemplu de graf conex (este format dintr-o singură componentă conexă):



### 4.1.7. Reprezentarea grafurilor orientate

Fie  $G=(V, U)$  un graf orientat, unde  $V=\{x_1, x_2, \dots, x_n\}$  și  $U=\{u_1, u_2, \dots, u_m\}$ .

Deoarece între mulțimea  $\{x_1, x_2, \dots, x_n\}$  și mulțimea  $\{1, 2, \dots, n\}$  există o bijectie,  $x_i \leftrightarrow i$ , putem presupune, fără a restrânge generalitatea, mai ales pentru a ușura scrierea, că  $V=\{1, 2, \dots, n\}$ .

În baza celor spuse mai sus, mai departe, în loc de  $x_i$  vom scrie  $i$ , și în loc de arcul

$(x_i, x_j)$  vom scrie  $(i, j)$ .

Pentru a putea prelucra un graf orientat cu ajutorul unui program, trebuie mai întâi să fie reprezentat în programul respectiv.

Pentru a reprezenta un graf orientat, într-un program, există mai multe modalități folosind diverse structuri de date; dintre acestea în continuare vom prezenta:

- reprezentarea prin matricea de adiacență;
- reprezentarea prin matricea vârfuri-arce;
- reprezentarea prin matricea drumurilor;
- reprezentarea prin listele de adiacență;
- reprezentarea prin sirul arcelor.

### Matricea de adiacență

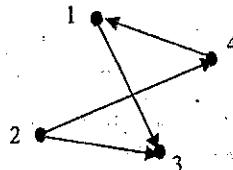
Fie  $G = (V, U)$  un graf orientat cu  $n$  vârfuri ( $V = \{1, 2, \dots, n\}$ ) și  $m$  arce.

Matricea de adiacență ( $A \in M_n(\{0, 1\})$ ), asociată grafului  $G$ , este o matrice pătratică de ordin  $n$ , cu elementele:

$$a_{i,j} = \begin{cases} 1, & \text{daca } (i, j) \in U \\ 0, & \text{daca } (i, j) \notin U \end{cases}$$

(altfel spus,  $a_{i,j}=1$ , dacă există arc între  $i$  și  $j$  și  $a_{i,j}=0$ , dacă nu există arc între  $i$  și  $j$ ).

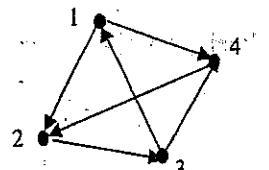
♦ Exemplul 1. Fie graful reprezentat ca în figura de mai jos:



Matricea de adiacență asociată grafului este:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

♦ Exemplul 2. Fie graful reprezentat ca în figura de mai jos:



Matricea de adiacență asociată grafului este:

### GRAFURI ORIENTATE

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

#### ◆ Comentarii:

1. Matricea de adiacență este o matrice pătratică, de ordin  $n$ , și nu este neapărat simetrică față de diagonala principală, așa cum este în cazul grafurilor neorientate.

#### ◆ Secvențele de citire a matricei de adiacență:

int a[100][100];

```

cout<<"n="; cin>>n;
for (i=1;i<=n;i++)
    for (j=1;j<=n;j++)
    {
        cout<<"a[" <<i<<"," <<j<<"]=";
        cin>>a[i][j];
    }
  
```

sau:

```

cout<<"n m "; cin>>n>>m;
for (i=1;i<=m;i++)
{
    cout<<"dati extremitatile arcului "<<i<< " "; cin>>x>>y;
    a[x][y]=1;
}
  
```

2. Matricea de adiacență are toate elementele de pe diagonala principală egale cu 0 (ne referim la definiția 1, când graful nu are bucle).

3. Numărul elementelor egale cu 1 de pe linia  $i$  este egal cu gradul exterior al vârfului  $i$ .

```

int gr_ext(int i)
{
    int j, s;
    s=0;
    for (j=1;j<=n;j++) s=s+a[i][j];
    return s;
}
  
```

4. Numărul elementelor egale cu 1 de pe coloana i este egal cu gradul interior al vârfului i.

```
int gr_int(int i)
{
    int j, s;
    s=0;
    for (j=1;j<=n;j++)
        s=s+a[j][i];
    return s;
}
```

5. Dacă vârful i este un vârf izolat, pe linia i și coloana i nu sunt elemente egale cu 1.

```
int vf_isolat(int i)
{
    return (gr_ext(i)==0) && (gr_int(i)==0);
}
```

### Matricea vârfuri–arce

Fie  $G=(V, U)$  un graf orientat cu n vârfuri ( $V=\{1, 2, \dots, n\}$ ) și m arce.

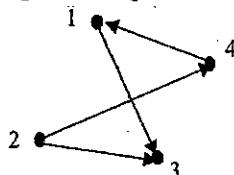
Matricea vârfuri–arce ( $B \in M_{nxm}(\{-1, 0, 1\})$ ), asociată grafului G, este o matrice cu n linii și m coloane, cu elementele:

$$b_{i,j} = \begin{cases} -1, & \text{daca } i \text{ este extremitate finală pentru arcul } u_j \\ 0, & \text{daca } i \text{ nu este extremitate pentru arcul } u_j \\ +1, & \text{daca } i \text{ este extremitate initială pentru arcul } u_j \end{cases}$$

- ♦ Exemplul 1. Fie graful  $G=(V, U)$  :  $V=\{1, 2, 3, 4\}$ ,  $U=\{(1,3), (2,3), (2,4), (4,1)\}=$

$=\{u_1, u_2, u_3, u_4\}$ ,

reprezentat ca în figura de mai jos:



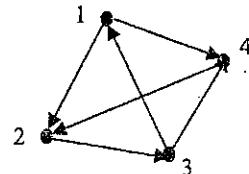
Matricea vârfuri–arce asociată grafului este:

### GRAFURI ORIENTATE

$$B = \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 1 & 0 \\ -1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 \end{pmatrix}$$

- ♦ Exemplul 2. Fie graful  $G$  :  $V=\{1, 2, 3, 4\}$ ,  $U=\{(1,2), (1,4), (2,3), (3,1), (3,4), (4,2)\}=$   
 $=\{u_1, u_2, u_3, u_4, u_5, u_6\}$

reprezentat ca în figura de mai jos:



Matricea vârfuri–arce asociată grafului este:

$$B = \begin{pmatrix} b_{11} & b_{12} & b_{13} & b_{14} & b_{15} & b_{16} \\ b_{21} & b_{22} & b_{23} & b_{24} & b_{25} & b_{26} \\ b_{31} & b_{32} & b_{33} & b_{34} & b_{35} & b_{36} \\ b_{41} & b_{42} & b_{43} & b_{44} & b_{45} & b_{46} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & -1 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & -1 & 1 & 1 & 0 \\ 0 & -1 & 0 & 0 & -1 & 1 \end{pmatrix}$$

### Comentarii:

1. Matricea vârfuri–arce nu este neapărat o matrice pătratică.

#### Secvențele de citire a matricei vârfuri–arce:

```
int b[100][100];
cout<<"n m ";
cin>>n>>m;
for (i=1;i<=n;i++)
    for (j=1;j<=m;j++)
        cout<<"b["<<i<<","<<j<<"]=";
        cin>>b[i][j];
    }
```

sau:

```
cout<<"n m ";
cin>>n>>m;
for (i=1;i<=m;i++)
{
    cout<<"dati extremitatile arcului "<<i<< " ";
    cin>>x>>y;
    b[x][i]=1;
    b[y][i]=-1;
}
```

2. Numărul elementelor egale cu 1 de pe linia  $i$  este egal cu gradul exterior al vârfului  $i$ .

```
int gr_ext_B( int i)
{
    int j, nr;
    nr=0;
    for (j=1;j<=m;j++)
        if (b[i][j]==1)
            nr=nr+1;
    return nr;
}
```

3. Numărul elementelor egale cu -1 de pe linia  $i$  este egal cu gradul interior al vârfului  $i$ .

```
int gr_int_B( int i)
{
    int j, nr;
    nr=0;
    for (j=1;j<=m;j++)
        if (b[i][j]==-1)
            nr=nr+1;
    return nr;
}
```

4. Dacă vârful  $i$  este un vârf izolat, pe linia  $i$  nu sunt elemente egale cu 1 sau -1.

```
int vf_isolat_B( int i)
{
    return (gr_ext_B(i)==0) && (gr_int_B(i)==0);
}
```

5. Pe fiecare coloană  $j$ , există un singur element egal cu 1 și un singur element egal cu -1.

Indicele liniei pe care se află 1 este extremitatea inițială a arcului  $u_j$

Indicele liniei pe care se află -1 este extremitatea finală a arcului  $u_j$

\*Construirea matricei de adiacență, când se cunoaște matricea vârfuri–arce.

- se parcurge matricea vârfuri–arce, de la prima până la ultima coloană, cu  $j$ 
  - pe coloana  $j$ , se depistează indicele liniei pe care se află 1, fie acesta **plus1**;
  - pe coloana  $j$ , se depistează indicele liniei pe care se află -1, fie acesta **minus1**;
  - în matricea de adiacență elementul  $a[plus1][minus1]$  se face 1.

```
for (j=1;j<=m;j++)
{
    for (i=1;i<=n;j++)
        if (b[i][j]==1) plus1=i;
        else if (b[i][j]==-1) minus1=i;
    a[plus1][minus1]=1;
}
```

\*Construirea matricei vârfuri–arce, când se cunoaște matricea de adiacență.

- se folosește variabila întreagă  $k$ , cu următorul rol:
  - reprezintă numărul arcului la care s-a ajuns (la al cătelea element  $a_{ij}=1$  s-a ajuns), care este practic indicele curent al coloanei la care s-a ajuns în matricea vârfuri–arce.

-  $k=0$ ;

- se parcurge matricea de adiacență, linie cu linie
  - dacă se găsește un element  $a_{ij}=1$ , atunci
    - se mărește  $k$  cu 1;
    - în coloana  $k$ , din matricea vârfuri–arce, se trece
      - pe linia  $i$  valoarea 1
      - pe linia  $j$  valoarea -1

```
k=0;
for (i=1;i<=n;i++)
    for (j=1;j<=n;j++)
        if (a[i][j]==1)
        {
            k=k+1;
            a[i][k]=1;
            a[j][k]=-1;
        }
```

### Matricea drumurilor

Fie  $G=(V, U)$  un graf orientat cu  $n$  vârfuri ( $V=\{1, 2, \dots, n\}$ ) și  $m$  arce.

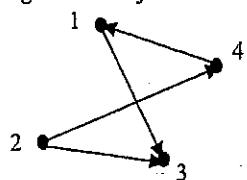
Matricea drumurilor ( $D \in M_n(\{0,1\})$ ), asociată grafului  $G$ , este o matrice cu  $n$  linii și  $n$  coloane, cu elementele:

$$d_{i,j} = \begin{cases} 0, & \text{daca nu exista drum in } G \text{ de la } i \text{ la } j \\ 1, & \text{daca exista drum in } G \text{ de la } i \text{ la } j \end{cases}$$

♦ **Exemplul 1.** Fie graful  $G=(V, U)$ :  $V=\{1, 2, 3, 4\}$ ,  $U=\{(1,3), (2,3), (2,4), (4,1)\}$

$$= \{u_1, u_2, u_3, u_4\},$$

reprezentat ca în figura de mai jos:

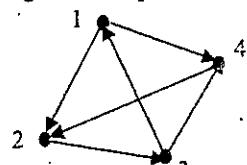


Matricea drumurilor asociată grafului este:

$$D = \begin{pmatrix} d_{11} & d_{12} & d_{13} & d_{14} \\ d_{21} & d_{22} & d_{23} & d_{24} \\ d_{31} & d_{32} & d_{33} & d_{34} \\ d_{41} & d_{42} & d_{43} & d_{44} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

♦ **Exemplul 2.** Fie graful  $G$ :  $V=\{1, 2, 3, 4\}$ ,  $U=\{(1,2), (1,4), (2,3), (3,1), (3,4), (4,2)\}=$   
 $= \{u_1, u_2, u_3, u_4, u_5, u_6\}$ ,

reprezentat ca în figura de mai jos:



Matricea drumurilor asociată grafului este:

$$D = \begin{pmatrix} d_{11} & d_{12} & d_{13} & d_{14} \\ d_{21} & d_{22} & d_{23} & d_{24} \\ d_{31} & d_{32} & d_{33} & d_{34} \\ d_{41} & d_{42} & d_{43} & d_{44} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

#### ◆ Comentarii:

1. Matricea drumurilor este o matrice pătratică.

În continuare, este prezentat în pseudocod algoritmul Roy-Warshall de determinare a matricei drumurilor plecând de la matricea de adiacență.

.....  
 pentru  $k=1 \dots n$   
     pentru  $i=1 \dots n$  ( $i \neq k$ )  
         pentru  $j=1 \dots n$  ( $j \neq k$ )  
             dacă  $a_{ij}=0$  atunci  
                  $a_{ij}=\min(a_{ik}, a_{kj})$   
 .....

Algoritmul constă într-un sir de transformări aplicate matricei de adiacență; de aceea, nu folosește o altă matrice, rezultatul final fiind practic matricea de adiacență modificată.

Dacă  $T_1, T_2, \dots, T_n$  ( $T_k$ ,  $k=1 \dots n$ ) este sirul transformărilor aplicate matricei  $A$ , cu  $T_k(A)=B$  atunci

$$b_{i,j} = \begin{cases} \max(a_{ij}, \min(a_{ik}, a_{kj})), & \text{daca } k \neq i \text{ si } k \neq j \\ a_{ij}, & \text{daca } k = i \text{ sau } k = j \end{cases}$$

2. Dacă în matricea drumurilor  $d_{ii}=1$ , înseamnă că există în graf un circuit de extremități  $i$ .
3. Dacă în matricea drumurilor linia  $i$  și coloana  $i$  au elementele egale cu 0, nodul  $i$  este un nod izolat.

♦ Programul C/C++ de construire și afișare a matricei drumurilor.

```
#include <iostream.h>
#include <stdio.h>
#include <conio.h>
typedef int mat[30][30];
mat a;
int i, j, k, n, m, x, y;
main()
{
    cout<<"n="; cin>>n;
    cout<<"m="; cin>>m;
    for (i=1;i<=m;i++)
    {
        cout<<"arcul "<<i<<" ";
        cout<<"x y "; cin>>x>>y;
        a[x][y]=1;
    }
    for (k=1;k<=n;k++)
        for (i=1;i<=n;i++)
            for (j=1;j<=n;j++)
                if (a[i][j]==0)
                    a[i][j]=a[i][k]*a[k][j];
}
```

```
cout<<"matricea drumurilor este ";
for (i=1;i<=n;i++)
{
    for (j=1;j<=n;j++)
        cout<<a[i][j]<<" ";
    cout<<endl;
}
getche(); }
```

secvența de citire a matricei de adiacență

secvența de transformare a matricei de adiacență în matricea drumurilor

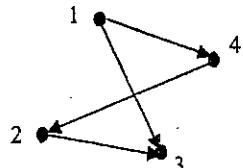
## Liste de adiacență

Fie  $G=(V, U)$  un graf orientat, cu  $n$  vârfuri ( $V=\{1, 2, \dots, n\}$ ) și  $m$  arce.

Reprezentarea grafului  $G$ , prin liste de adiacență, constă în:

- precizarea numărului de vârfuri  $n$ ;
- pentru fiecare vârf  $i$ , se precizează lista  $L_i$  a succesorilor săi, adică lista nodurilor care fac parte din multimea  $\Gamma^+(i)$ .

♦ Exemplul 1. Fie graful din figura de mai jos:

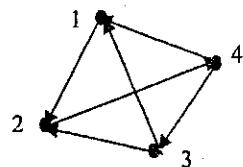


Reprezentarea sa, prin liste de adiacențe, presupune:

- precizarea numărului de vârfuri  $n$ ,  $n=4$ ;
- precizarea listei succesorilor lui  $i$ , pentru  $i=1..n$

Vârful $i$	Lista succesorilor lui $i$
1	3, 4
2	3
3	
4	2

♦ Exemplul 2. Fie graful din figura de mai jos:



Reprezentarea sa, prin liste de adiacențe, presupune:

- precizarea numărului de vârfuri  $n$ ,  $n=4$ ;
- precizarea listei vecinilor lui  $i$ , pentru  $i=1..n$

Vârful $i$	Lista vecinilor lui $i$
1	2
2	4
3	1, 2
4	1, 3

## GRAFURI ORIENTATE

### ◆ Comentarii:

Acest mod de reprezentare se poate implementa astfel:

1. Se folosește un tablou bidimensional, caracterizat astfel:

- are  $n+m$  coloane;
- $T_{i,i}=i$ , pentru  $i=1..n$ ;
- Pentru  $i=1..n$   $T_{2,i}=k$ , dacă  $T_{i,k}$  este primul nod din lista vecinilor lui  $i$ ;
- $T_{2,i}=0$ , dacă nodul  $i$  nu are succesiuni;
- Dacă  $T_{i,j}=u$ , adică  $u$  este un nod din lista vecinilor lui  $i$ , atunci:

$T_{2,j}=0$ , dacă  $u$  este ultimul nod din lista vecinilor lui  $i$ ;

$T_{2,j}=j+1$ , dacă  $u$  nu este ultimul nod din lista vecinilor lui  $i$ .

♦ Exemplu de completare a tabloului pentru graful de la exemplul 1.

Prima etapă. Se numerotează coloanele  $(1..n+m)$ , și se trec vârfurile.

1	2	3	4	5	6	7	8
1	2	3	4				

A doua etapă. Se trec în tabel vecinii lui 1, începând de la coloana 5.

1	2	3	4	5	6	7	8
1	2	3	4	3	4		
5				6	0		

$T_{2,1}=5$ , pentru că primul vecin (3) al lui 1 s-a trecut la coloana 5 ( $T_{1,5}=3$ );

$T_{2,5}=6$ , pentru că următorul vecin (4) al lui 1 s-a trecut la coloana 6 ( $T_{1,6}=4$ );

$T_{2,6}=0$ , pentru că vecinul  $T_{1,6}$  (4) al lui 1 este ultimul din listă.

A treia etapă. Se trec în tabel vecinii lui 2, începând de la coloana 7.

1	2	3	4	5	6	7	8
1	2	3	4	3	4	3	
5	7	0		6	0	0	

$T_{2,2}=7$ , pentru că primul vecin (3) al lui 2 s-a trecut la coloana 7 ( $T_{1,7}=3$ );

$T_{2,7}=0$ , pentru că vecinul  $T_{1,7}$  (3) al lui 2 este ultimul din listă.

A patra etapă. Se trec în tabel vecinii lui 3, începând de la coloana 8.

1	2	3	4	5	6	7	8
1	2	3	4	3	4	3	
5	7	0		6	0	0	

$T_{2,3}=0$ , pentru că 3 nu are succesiuni, deci lista sa este vidă.

**Ultima etapă.** Se trec în tabel vecinii lui 4, începând de la coloana 8 (aici s-a ajuns),

1	2	3	4	5	6	7	8
1	2	3	4	3	4	3	2
5	7	0	8	6	0	0	0

$T_{2,4}=8$ , pentru că primul vecin (2) al lui 4 s-a trecut la coloana 8 ( $T_{1,8}=2$ );  
 $T_{2,8}=0$ , pentru că vecinul  $T_{1,8}$  (2) al lui 4 este ultimul din listă.

- 2.** Se folosesc un tablou unidimensional, cu numele cap, și un tablou bidimensional, cu numele L (care reține listele succesorilor pentru fiecare nod), caracterizate astfel:

Tabloul cap:

- are n componente;
- $cap_i=c$ , dacă primul nod din lista vecinilor lui i este trecut în tabloul L la coloana c, adică  $L_{1,c}$  este primul vecin al lui i, și  $cap_i=0$ , dacă nodul i nu are succesi.

Tabloul L:

- are m componente;
- dacă k este un vecin al nodului i, atunci:  
 $L_{1,p}=k$  și  $L_{2,p}=0$ , dacă k este ultimul vecin din listă, sau  
 $L_{1,p}=k$  și  $L_{2,p}=p+1$ , dacă k nu este ultimul vecin din listă.  
(p este coloana la care s-a ajuns în tabloul L)

♦ Exemplu de completare a tablourilor cap și L, pentru graful de la exemplul 1

Tabloul cap:

1	2	3	4
1	3	0	4

Tabloul L:

1	2	3	4
3	4	3	2
2	0	0	0

- 3.** Se folosesc un tablou bidimensional, cu numele L, caracterizat astfel:

- are n linii;
- pe linia i, se trăc succesorii nodului i.

♦ Exemplu de completare a tabloului L, pentru graful:



Tableul L

3		
1	4	
2	.	
1	3	

♦ Implementarea în limbajul C/C++, a ideii prezentate mai sus, se realizează conform secvenței de program prezentată mai jos.

```
int L[20][20];
int nr_vec[20];

cout<<"n="; cin>>n;
for (i=1;i<=n;i++)
{
    cout<<"Dati numarul vecinilor nodului "<<i; cin>>nr_vec[i];
    for (j=1;j<=nr_vec[i];j++)
        cin>>L[i][j];
}
```

♦ Construirea matricei de adiacență, când se cunoaște L (listele vecinilor fiecărui nod).

```
for (i=1;i<=n;i++)
{
    for (j=1;j<=nr_vec[i];j++)
        a[i][L[i][j]]=1;
}
```

♦ Construirea tabloului L (listele vecinilor nodurilor), când se cunoaște matricea de adiacență

```
for (i=1;i<=n;i++)
{
    k=0;
    for (j=1;j<=n;j++)
        if (a[i][j]==1) {
            k=k+1;
            L[i][k]=j;
        }
}
```

4. Se folosește un tablou unidimensional, cu numele L, caracterizat astfel:

- componente sale sunt de tip referință;
- are n componente;
- L<sub>i</sub> pointează spre începutul listei succesorilor nodului i.

◆ Construirea matricei de adiacență, când se cunoaște L (listele vecinilor fiecărui nod).

```
for (i=1;i<=n;i++)
{
    c=L[i];
    while (c)
    {
        a[i][c>nod]=1;
        c=c->urm;
    }
}
```

### Şirul arcelor

Fie G=(V, U) un graf orientat, cu n vârfuri (V={1,2, ..., n}) și m arce.

Reprezentarea grafului G constă în precizarea numărului n de noduri și numărului m de arce precum și în precizarea extremităților pentru fiecare arc în parte.

◆ Comentarii:

Acest mod de reprezentare se implementează astfel:

1. Se dă numărul n de noduri și numărul m de arce, iar extremitățile fiecărui arc sunt trecute în vectorii e1 și e2, astfel:

- extremitățile primului arc sunt e1[1] și e2[1];
- extremitățile celui de-al doilea arc sunt e1[2] și e2[2];
- .....
- deci, U={(e1[1],e2[1]), (e1[2],e2[2]), ..., (e1[m],e2[m])}

◆ Secvența C/C++ corespunzătoare este:

```
int e1[100], e2[100];
int n, m, i;

cout<<"n="; cin>>n;
cout<<"m="; cin>>m;
for (i=1;i<=m;i++)
{
    cout<<n=>>n;
    cout<<m=>>m;
}
```

```
cout<<"Dati extremitatile arcului cu numarul "<<i<<" ";
cin>>e1[i]>>e2[i];
}
```

◆ Construirea matricei de adiacență, când se cunoaște șirul muchiilor ca mai sus.

```
cout<<"n="; cin>>n;
for (i=1;i<=m;i++)
    a[e1[i]][e2[i]]=1;
```

◆ Construirea șirului arcelor, ca mai sus, când se dă matricea de adiacență.

```
k=0;
for (i=1;i<=n;i++)
    for (j=1;j<=n;j++)
        if (a[i][j]==1) {
            k=k+1;
            e1[k]=i;
            e2[k]=j;
        }
m=k;
```

2. Se folosește un tablou unidimensional, cu numele u, caracterizat astfel:

- componente sale sunt de tip record;
- are m componente;
- u<sub>i</sub> reprezintă arcul i.

◆ Pentru implementare este nevoie de:

```
struct arc{
    int x;
    int y;
};
arc u[20];
```

Accesul la arcul i se face: u[i].x ..... u[i].y

◆ Secvența C/C++ corespunzătoare este:

```
cout<<"n="; cin>>n;
cout<<"m="; cin>>m;
for (i=1;i<=m;i++)
{
```

```

cout<<"Dati extremitatile arcului cu numarul "<<i<<" ";
cin>>u[i].x>>u[i].y;
}

```

◆ Construirea matricei de adiacență, când se cunoaște sirul arcelor ca mai sus:

```

cout<<"n="; cin>>n;
for (i=1;i<=n;i++)
    a[u[i].x][u[i].y]=1;

```

◆ Construirea sirului arcelor, ca mai sus, când se dă matricea de adiacență:

```

k=0;
for (i=1;i<=n;i++)
    for (j=1;j<=n;j++)
        if (a[i][j]==1) {
            k=k+1;
            u[k].x=i;
            u[k].y=j;
        }
m=k;

```

#### 4.1.8. Tare conexitate

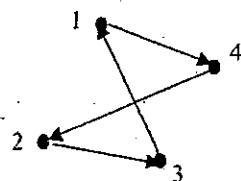
În această secțiune, vor fi prezentate noțiunile:

- graf tare conex
- componentă tare conexă
- algoritmul de descompunere a unui graf în componente tare conexe

##### Graf tare conex

**Definiție.** Fie  $G=(V, U)$  un graf orientat. Graful  $G$  se numește **tare conex**, dacă pentru oricare două vârfuri  $x$  și  $y$  există un drum în  $G$  de la  $x$  la  $y$  și un drum de la  $y$  la  $x$ .

◆ Exemplu de graf tare conex.



Graful este tare conex, deoarece, oricare ar fi vârfurile  $x$  și  $y$ , există un drum în  $G$  de la  $x$  la  $y$  și un drum de la  $y$  la  $x$ .

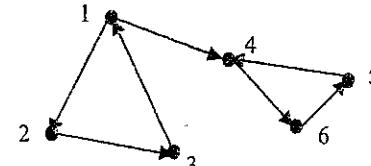
##### Componentă tare conexă

**Definiție.** Fie  $G=(V, U)$  un graf orientat. Se numește **componentă tare conexă**, un graf orientat  $G_i=(V_i, U_i)$  care verifică următoarele condiții:

- este subgraf al grafului  $G$ ;

- este tare conex;
- oricare are fi  $x \in V - V_i$ , subgrauflui lui  $G$  generat de  $V_i \cup \{x\}$  nu mai este tare conex.

◆ Exemplu. Fie graful orientat prezentat în figura de mai jos:



Acet graf are două componente tare conexe:

- subgrauflul generat de nodurile 1, 2, 3;
- subgrauflul generat de nodurile 4, 5, 6.

**Observație.** Fie  $G=(V, U)$  un graf orientat. Graful  $G$  este tare conex, dacă admite o singură componentă tare conexă.

##### Algoritmul de descompunere a unui graf în componente tare conexe

Algoritmul procedează astfel:

- la început, nu este depistată nici o componentă tare conexă ( $nc=0$ );
- deci, nici un nod nu face parte din vreo componentă tare conexă ( $luate=[ ]$ );
- se parcurg nodurile grafului, cu i,
- dacă i nu a fost introdus în nici o componentă tare conexă,
  - se mărește numărul componentelor tare conexe cu 1,
  - se construiește noua componentă tare conexă, astfel:
    - se intersectează predecesorii lui i cu succesorii săi, și se reunesc cu {i}.

Pentru implementarea acestui algoritm, în limbajul C/C++, cu ajutorul programului prezentat mai jos, s-au folosit:

**Funcțiile:**

**Succesori(i, S)**: care pune în sirul S toate nodurile j, din graf, cu proprietatea că există drum între i și ele.

**Predecesori(i, P)**: care pune în sirul P toate nodurile j, din graf, cu proprietatea că există drum între ele și i.

**Vectorul:**

**Comp**: ale cărui componente, care sunt siruri de elemente, vor reține, la final, componentele tare conexe;

**Variabilele:**

d : matricea drumurilor;

luate : un sir care reține toate nodurile care fac deja parte dintr-o componentă tare conexă;

nc : reprezintă numărul componentelor tare conexe depistate;

În program, au mai fost folosite și alte variabile dar deoarece rolul lor reiese foarte ușor din urmărirea programului nu-l mai comentăm.

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

typedef int mat[20][20];
typedef int sir[20];

mat d;
sir S, P, luate;
sir comp[50], ncomp;
int ok, nl, i, j, n, m, nc, ns, np;
```

```
void succesor(int i, sir S, int& ns)
{
    int j;
    ns=0;
    for (j=1;j<=n;j++)
        if (d[i][j]==1)
    {
        ns=ns+1;
        S[ns]=j;
    }
}
```

```
void predecesori(int i, sir P, int& np)
```

```
{
    int j;
    np=0;
    for (j=1;j<=n;j++)
        if (d[j][i]==1)
    {
        np=np+1;
        P[np]=j;
    }
}
```

```
void intersectie(sir S, int ns, sir P, int np, sir x, int& nx)
```

```
{
    int ok;
    int i, j;
    nx=0;
    for (i=1;i<=ns;i++)
    {
        ok=0;
        for (j=i;j<=np;j++)
            if (S[i]==P[j]) ok=1;
        if (ok==1)
        {
            nx++;
            x[nx]=S[i];
        }
    }
}
```

```
main()
```

```
{
    clrscr();
    cout<<"n="; cin>>n;
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
```

```

cout<<"d["<<i<<","<<j<<"] ";
cin>>d[i][j];
}

nc=0;
nl=0;
for (i=1;i<=n;i++)
{
    ok=0;
    for (j=1;j<=nl;j++)
        if (luate[j]==i) ok=1;

    if (ok==0)
    {
        nc++;
        succesori(i,S,ns);
        predecesori(i,P,np);
        intersectie(S,ns,P,np,comp[nc],nocomp[nc]);
    }

    for (j=1;j<=ncomp[nc];j++)
    {
        nl++;
        luate[nl]=comp[nc][j];
    }
}

for (i=1;i<=nc;i++)
{
    cout<<"componenta tare conexa cu numarul "<<i<<endl;
    for (j=1;j<=ncomp[i];j++)
        cout<<comp[i][j]<< " ";
    cout<<endl;
}

```

## 4.1.9. Drumuri minime și maxime

### Noțiuni generale

În această secțiune vor fi prezentate, așa cum sugerează și titlul, modurile de tratare a problemelor care fac parte din următoarele două mari clase de probleme:

- probleme în care se cere determinarea **drumurilor minime**, dintr-un graf;
- probleme în care se cere determinarea **drumurilor maxime**, dintr-un graf.

Problemele de **minim** (**maxim**) se pot enunța astfel:

1. Fiind dat graful  $G=(V,U)$ , cu matricea costurilor asociată  $C \in M_n(\mathbb{N})$ , să se determine **drumurile de lungime minimă** (**maximă**) între **oricare două vârfuri**.
2. Fiind dat graful  $G=(V,U)$ , cu matricea costurilor asociată  $C \in M_n(\mathbb{N})$ , să se determine **drumurile de lungime minimă** (**maximă**) între **vârfurile i și j**.
3. Fiind dat graful  $G=(V,U)$ , cu matricea costurilor asociată  $C \in M_n(\mathbb{N})$ , să se determine **drumurile de lungime minimă** (**maximă**) între **vârful i și toate celelalte vârfuri**.

În cazul **problemelor de minim**, fiind dat graful  $G=(V, U)$  î se asociază **matricea costurilor, forma 1**, definită astfel:

$C \in M_n(\mathbb{N})$ , unde:

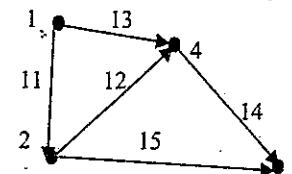
$$c_{i,j} = \begin{cases} \text{cost}, & \text{daca } i \text{ si } j \text{ există un arc cu costul cost} \\ 0, & \text{daca } i = j \\ \infty, & \text{daca } i \neq j \text{ si } (i, j) \notin U \end{cases}$$

În cazul **problemelor de maxim**, fiind dat graful  $G=(V, U)$  î se asociază **matricea costurilor, forma 2**, definită astfel:

$C \in M_n(\mathbb{N})$ , unde:

$$c_{i,j} = \begin{cases} \text{cost}, & \text{daca } i \text{ si } j \text{ există un arc cu costul cost} \\ 0, & \text{daca } i = j \\ -\infty, & \text{daca } i \neq j \text{ si } (i, j) \notin U \end{cases}$$

◆ **Exemplu:** Fiind dat graful din figura de mai jos (costul fiecărui arc fiind scris pe ea)



matricea costurilor se scrie în felul următor:

**Forma 1:**

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{pmatrix} = \begin{pmatrix} 0 & 11 & \infty & 13 \\ \infty & 0 & 15 & 12 \\ \infty & \infty & 0 & \infty \\ \infty & \infty & 14 & 0 \end{pmatrix}$$

**Forma 2:**

$$C = \begin{pmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{pmatrix} = \begin{pmatrix} 0 & 11 & -\infty & 13 \\ -\infty & 0 & 15 & 12 \\ -\infty & -\infty & 0 & -\infty \\ -\infty & -\infty & 14 & 0 \end{pmatrix}$$

**Observații.** 1. Matricea costurilor forma 1 diferă de matricea costurilor forma 2 prin faptul că în loc de  $\infty$  apare  $-\infty$ .

2. În program nu se poate scrie  $\infty$  sau  $-\infty$ , de aceea recomandăm ca atunci când trebuie folosite să se definească două constante foarte mari, ca de exemplu, pentru  $\infty$ : const p\_infinit = 1.e10;  
pentru  $-\infty$ : const m\_infinit = -1.e10;

În continuare, vor fi prezentate doi algoritmi care permit rezolvarea unor probleme de minim (maxim), și anume, vor fi prezentate:

algoritmul Roy-Floyd și algoritmul lui Dijkstra.

### Algoritmul Roy-Floyd

Acest algoritm se aplică în cazul problemelor în care se dă un graf  $G=(V, U)$ , care are matricea costurilor  $C$ , și se cere să se determine lungimea drumurilor minime, și în unele cazuri și nodurile care constituie drumurile respective, între oricare două noduri ale grafului.

**Observație.** Algoritmul are la bază următoarea idee:

"Dacă drumul minim de la nodul  $i$  la nodul  $j$  trece prin nodul  $k$ , atunci și drumul de nodul  $i$  la nodul  $k$ , precum și de la nodul  $k$  la nodul  $j$ , este minim."

și constă, de fapt, într-un sir de  $n$  transformări aplicate matricei costurilor  $C$ , astfel:

$$T_k(C) = B, B \in M_n(R), b_{i,j} = \min(c_{i,j}, c_{i,k} + c_{k,j}), \forall i, j \in \{1, \dots, n\}$$

care se poate implementa astfel:

```

for k=1 ... n
    for i=1 ... n
        for j=1 ... n
            if (c[i,j] < c[i,k]+c[k,j]) then
                c[i,j]:=c[i,k]+c[k,j];

```

**Descrierea detaliată a algoritmului prezentat mai sus:**

Pentru fiecare pereche de noduri  $(i, j)$ , unde  $i, j \in \{1, \dots, n\}$ , se procedează astfel:

se parcurge cu  $k$  toate nodurile grafului, diferite de  $i$  și  $j$ ,

pentru fiecare nod  $k$ , se execută:

dacă costul drumului între nodurile  $i$  și  $j$  este mai mic decât suma costurilor drumurilor între nodurile  $i$  și  $k$  și între nodurile  $k$  și  $j$ , atunci costul drumului initial de la  $i$  la  $j$  se va înlocui cu costul drumului  $i-k-j$ , evident, acest lucru făcându-se prin modificarea matricei costurilor.

**Observație.** Algoritmul prezentat în forma de mai sus, permite decât calcularea lungimii drumurilor minime între oricare două noduri ale grafului. Dacă se dorește și afișarea nodurilor care compun efectiv aceste drume, va trebui completat astfel:

Se folosește matricea  $D$ , pătratică de ordinul  $n$ , ale cărei elemente  $d_{i,j}$ , unde

$i, j \in \{1, \dots, n\}$ , sunt multumii cu următoarea semnificație:

$d_{i,j}$  este, în final, multimea nodurilor grafului care pot precede pe nodul  $j$  în drumul minim de la nodul  $i$  la nodul  $j$ .

Matricea  $D$  este inițializată, odată cu inițializarea matricei costurilor, astfel:

$$d_{i,j} = \begin{cases} \{i\}, & \text{dacă } i \neq j \text{ și } c_{i,j} < \infty \\ \emptyset, & \text{dacă } i = j \text{ sau } c_{i,j} = \infty \end{cases}$$

și transformată pe parcurs, odată cu transformarea matricei costurilor, după regulile:

dacă  $c_{i,j} < c_{i,k} + c_{k,j}$ , atunci  $d_{i,j}$  rămâne neschimbat;

dacă  $c_{i,j} = c_{i,k} + c_{k,j}$ , atunci la  $d_{i,j}$  se adaugă vârfurile din  $d_{k,j}$ ;

dacă  $c_{i,j} > c_{i,k} + c_{k,j}$ , atunci  $d_{i,j}$  se initializează cu  $d_{k,j}$ .

În continuare, se prezintă programul, în C/C++, care implementează algoritmul comentat anterior

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

```
#include <conio.h>
```

```
#include <stdio.h>
```

```
const p_inf=10000;
```

```
typedef int sir[20];
```

```
typedef int mat[20][20];
```

```
typedef sir matm[20][20];
```

```
mat c, nd;
```

```
matm d;
```

```
sir dr;
```

```
int i, k, j, n,m, ld,x,y,val;
```

```

void drum_de_la(int i,int j){
    int k, gasitk,t;
    if (i!=j){
        for (k=1;k<=n;k++)
        {
            gasitk=0;
            for (t=1;t<=nd[i][j];t++)
                if (d[i][j][t]==k) gasitk=1;
            if (gasitk==1){
                ld=ld+1;
                dr[ld]=k;
                drum_de_la(i,k);
                ld=ld-1;
            }
        }
    }
    else{
        for (k=ld;k>=1;k--)
            cout<<dr[k]<<" ";
        cout<<endl;
    }
}

void afis(){
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            if (c[i][j]==p_inf) cout<<"nu exista drum intre "<<i<<" si "<<j;
            else if (i!=j){
                cout<<"lung. drumului min de la "<<i<<" la "<<j<<" este
                    "<<c[i][j]<<endl;
                cout<<"iar drumarile sunt :"<<endl;
                ld=1;
                dr[ld]=j;
                drum_de_la(i,j);
            }
}

void reuneste(sir x,int nx, sir y, int ny, sir z, int& nz){
    int i, j, ok;
    nz=0;
    for (i=1;i<=nx;i++){
        nz++;

```

```

        z[nz]=x[i];
    }
    for (j=1;j<=ny;j++)
    {
        ok=0;
        for (i=1;i<=nx;i++)
            if (x[i]==y[j]) ok=1;
        if (!ok) {
            nz++;
            z[nz]=y[j];
        }
    }
}

void face_sirul(sir x, int nx, sir y, int& ny){
    int i;
    ny=0;
    for (i=1;i<=nx;i++){
        ny++;
        y[ny]=x[i];
    }
}

main(){
    clrscr();
    cout<<"n="; cin>>n;
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            if (i==j) c[i][j]=0;
            else c[i][j]=p_inf;
    cout<<"m="; cin>>m;
    for (i=1;i<=m;i++)
    {
        cout<<"x y val "; cin>>x>>y>>val;
        c[x][y]=val;
    }
    for (i=1;i<=n;i++)
    {
        for (j=1;j<=n;j++)
            cout<<c[i][j]<<" ";
        cout<<endl;
    }
}

```

```

for (i=1;i<=n;i++)
    for (j=1;j<=n;j++)
        if ((i!=j) && (c[i][j]<p_inf))
        {
            nd[i][j]=1;
            d[i][j][1]=i;
        }
        else nd[i][j]=0;

for (k=1;k<=n;k++)
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            if (c[i][j]==c[i][k]+c[k][j])
                reuneste(d[i][j],nd[i][j],d[k][j],nd[k][j],d[i][j],nd[i][j]);
            else if (c[i][j]>c[i][k]+c[k][j])
                c[i][j]=c[i][k]+c[k][j];
                face_sirul(d[k][j],nd[k][j],d[i][j],nd[i][j]);
        }

afis();
getche();
}

```

### Algoritmul lui Dijkstra

Acest algoritm se aplică în cazul problemelor în care se dă un graf  $G=(V, U)$ , care are matricea costurilor  $C$ , și se cere să se determine lungimea drumurilor minime (maxime), și în unele cazuri și nodurile care constituie drumurile respective, între un nod și oricare alt nod al grafului.

**Algoritmul folosește următoarele variabile:**

- n : reprezintă numărul de noduri ale grafului;
- c : reprezintă matricea costurilor asociată grafului;
- pl : reprezintă nodul de plecare;
- d : vector cu n componente caracterizat astfel:  
    d[i] reprezintă distanța de la nodul pl până la nodul i;
- p : vector cu n componente caracterizat astfel:  
    p[i] reprezintă nodul care precede pe i în drumul minim;  
și procedează astfel:  
- se citeste nodul de plecare pl;  
- se completează componentele vectorului d astfel  
    d[i]=c[pl][i], pentru i=1...n și i≠pl;

$d[pl]=0$

- se completează componentele vectorului p astfel:

pentru  $i=1..n$ ,  $p[i]=pl$ , dacă  $i\neq pl$  și  $c[pl][i]=\infty$ ;

$p[i]=0$ , altfel;

- se execută de  $n-2$  ori următoarele:

- se determină nodul  $j$ , dintre cele rămasse, pentru care  $d[j]$  este minim

- se recalculează distantele nodurilor rămasse și se stabilesc predecesorii lor, astfel:  
pentru nodul i:

- se calculează  $\min\{d[i], d[i]+c[j,i]\}$

- dacă  $\min\{d[i], d[i]+c[j,i]\}=d[i]$

atunci

$d[i]$  și  $p[i]$  rămân nemodificați

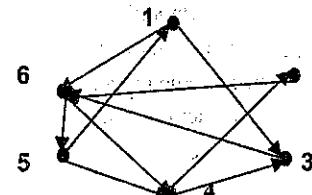
altfel

$d[i]=d[j]+c[j,i]$

$p[i]=j$

(unde  $j$  este nodul proaspăt căutat)

**Exemplu de aplicare a algoritmului asupra grafului:**



care are matricea costurilor:

$$C = \begin{pmatrix} 0 & \infty & 2 & \infty & \infty & 2 \\ \infty & 0 & \infty & \infty & \infty & 4 \\ \infty & \infty & 0 & \infty & \infty & 2 \\ \infty & 3 & 1 & 0 & \infty & \infty \\ 3 & \infty & \infty & 1 & 0 & \infty \\ \infty & \infty & \infty & 2 & 3 & 0 \end{pmatrix}$$

Rezolvare:

## Pasul 1

Se stabilește nodul de plecare:  $pl=1$ Se completează vectorii  $d$  și  $p$ , astfel:-  $d[i]=c[pl,i]$ , pentru  $i=1\dots n$ ;(în componentele vectorului  $d$  se trec elementele de pe prima linie din  $C$ )- pentru  $i=1\dots n$ ,  $p[i]=pl$ , dacă  $i \neq pl$  și  $c[pl,i] \neq \infty$ ; $p[i]=0$ , altfel;

	1	2	3	4	5	6
$d$	0	$\infty$	2	$\infty$	$\infty$	2
$p$	0	0	1	0	0	1

## Pasul 2

Dintre nodurile nealese încă,  $nealese=\{2,3,4,5,6\}$ , se alege nodul  $j$  pentru care $d_j=\min\{d_2, d_3, d_4, d_5, d_6\}$ ; deci, se alege  $j=3$ , pentru că  $d_3=\min\{\infty, 2, \infty, \infty, 2\}$ 

	1	2	3	4	5	6
$d$	0		2			
$p$	0		1			

Se completează componentele vectorilor  $d$  și  $p$ , pentru nodurile nealese, astfel:2 :  $\min(d_2, d_3+C_{3,2})=\min(\infty, 2+\infty)=\infty$  ( $d_2$  și  $p_2$  rămân nemodificate)4 :  $\min(d_4, d_3+C_{3,4})=\min(\infty, 2+\infty)=\infty$  ( $d_4$  și  $p_4$  rămân nemodificate)5 :  $\min(d_5, d_3+C_{3,5})=\min(\infty, 2+\infty)=\infty$  ( $d_5$  și  $p_5$  rămân nemodificate)6 :  $\min(d_6, d_3+C_{3,6})=\min(2, 2+2)=2$  ( $d_6$  și  $p_6$  rămân nemodificate)

	1	2	3	4	5	6
$d$	0	$\infty$	2	$\infty$	$\infty$	2
$p$	0	0	1	0	0	1

## Pasul 3

Dintre nodurile nealese încă,  $nealese=\{2,4,5,6\}$ , se alege nodul  $j$  pentru care $d_j=\min\{d_2, d_4, d_5, d_6\}$ ; deci, se alege  $j=6$ , pentru că  $d_6=\min\{\infty, \infty, \infty, 2\}$ 

	1	2	3	4	5	6
$d$	0		2			2
$p$	0		1			1

Se completează componentele vectorilor  $d$  și  $p$ , pentru nodurile nealese, astfel:2 :  $\min(d_2, d_6+C_{6,2})=\min(\infty, 2+\infty)=\infty$  ( $d_2$  și  $p_2$  rămân nemodificate)4 :  $\min(d_4, d_6+C_{6,4})=\min(\infty, 2+2)=4$  ( $d_4=4$  și  $p_4=6$ )5 :  $\min(d_5, d_6+C_{6,5})=\min(\infty, 2+3)=5$  ( $d_5=5$  și  $p_5=6$ )

## GRAFURI ORIENTATE

	1	2	3	4	5	6
$d$	0	$\infty$	2	4	5	2
$p$	0	0	1	6	6	1

## Pasul 4

Dintre nodurile nealese încă,  $nealese=\{2,4,5\}$ , se alege nodul  $j$  pentru care  $d_j=\min\{d_2, d_4, d_5\}$ ; deci, se alege  $j=4$ , pentru că  $d_4=\min\{\infty, 4, 5\}$ 

	1	2	3	4	5	6
$d$	0		2	4		2
$p$	0		1	6		1

Se completează componentele vectorilor  $d$  și  $p$ , pentru nodurile nealese, astfel:2 :  $\min(d_2, d_4+C_{4,2})=\min(\infty, 4+3)=7$  ( $d_2=7$  și  $p_2=4$ )5 :  $\min(d_5, d_4+C_{4,5})=\min(5, 4+\infty)=5$  ( $d_5$  și  $p_5$  rămân nemodificate)

	1	2	3	4	5	6
$d$	0	7	2	4	5	2
$p$	0	4	1	6	6	1

## Pasul 5

Dintre nodurile nealese încă,  $nealese=\{2,5\}$ , se alege nodul  $j$  pentru care $d_j=\min\{d_2, d_5\}$ ; deci, se alege  $j=5$ , pentru că  $d_5=\min\{\infty, 5\}$ 

	1	2	3	4	5	6
$d$	0		2	4	5	2
$p$	0		1	6	6	1

Se completează componentele vectorilor  $d$  și  $p$ , pentru nodurile nealese, astfel:2 :  $\min(d_2, d_5+C_{5,2})=\min(7, 5+\infty)=7$  ( $d_2$  și  $p_2$  rămân nemodificate)

	1	2	3	4	5	6
$d$	0	7	2	4	5	2
$p$	0	4	1	6	6	1

## Concluzii:

Drumurile minime de la nodul 1 la celelalte noduri sunt:

2 :  $p_2=4$ ,  $p_4=6$ ,  $p_6=1$ ; deci, de la 1 la 2 avem : 1, 6, 4, 23 :  $p_3=1$ ; deci, de la 1 la 3 avem : 1, 34 :  $p_4=6$ ,  $p_6=1$ ; deci, de la 1 la 4 avem : 1, 6, 45 :  $p_5=6$ ,  $p_6=1$ ; deci, de la 1 la 5 avem : 1, 6, 56 :  $p_6=1$ ; deci, de la 1 la 6 avem : 1, 6

În continuare, se prezintă programul, în C/C++, care implementează algoritmul comentat mai sus.

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

typedef int mat[20][20];
typedef int sir[20];
const p_inf=10000;
mat c;
sir d, p;
int i, k, j, n, pl, x, y, val, m, n_r, poz;
sir ramase;
double min;

void drum(int i){
    if (i!=0){
        drum(p[i]);
        cout<<i<<" ";
    }
}

main(){
    clrscr();
    cout<<"n="; cin>>n;
    for (i=1;i<=n;i++)
        for (j=1;j<=n;j++)
            if (i==j) c[i][j]=0;
            else c[i][j]=p_inf;

    cout<<"m="; cin>>m;
    for (i=1;i<=m;i++){
        cout<<"x y val" ; cin>>x>>y>>val;
        c[x][y]=val;
    }
    for (i=1;i<=n;i++){
        for (j=1;j<=n;j++)
            cout<<c[i][j]<<" ";
        cout<<endl;
    }
}
```

```
cout<<"nodul de plecare : "; cin>>pl;
n_r=0;
for (i=1;i<=n;i++)
    if (i!=pl) {
        n_r=n_r+1;
        ramase[n_r]=i;
    }

for (i=1;i<=n_r;i++)
    d[ramase[i]]=c[pl][ramase[i]];
for (i=1;i<=n;i++)
    if ((c[pl][i]!=p_inf) && (i!=pl)) p[i]=pl;
    else p[i]=0;

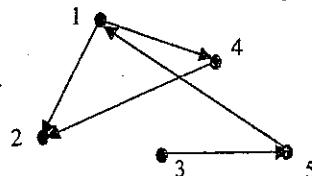
for (k=1;k<=n-2;k++){
    min=1e20;
    for (i=1;i<=n_r;i++)
        if (d[ramase[i]]<min)
            {
                min=d[ramase[i]];
                j=ramase[i];
                poz=i;
            }
    for (i=poz;i<=n_r-1;i++)
        ramase[i]=ramase[i+1];
    n_r=n_r-1;
    for (i=1;i<=n_r;i++)
        if (c[j][ramase[i]]!=p_inf){
            if (((d[ramase[i]]==p_inf) || (d[ramase[i]]>d[j]+c[j][ramase[i]])) ||
                (d[ramase[i]]==d[j]+c[j][ramase[i]]))
                d[ramase[i]]=d[j]+c[j][ramase[i]];
            p[ramase[i]]=j;
        }
}

for (i=1;i<=n;i++)
    if (i!=pl){
        if (p[i]==0) cout<<" nu exista ";
        else {
            drum(i);
            cout<<endl;
        }
    }
}}
```

## 4.2. PROBLEME PROPUSE

### 4.2.1. Noțiunea de graf orientat

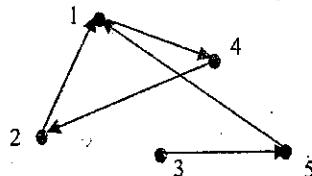
- Să se precizeze dacă rețelei de circulație din orașul dumneavoastră i se poate asocia un graf orientat; în caz afirmativ, să se definească graful corespunzător.
- Având la dispoziție un grup de  $n$  persoane,  $n \in N^*$ , să se precizeze dacă i se poate asocia un graf orientat; în caz afirmativ, să se definească graful corespunzător.
- Având la dispoziție o hartă cu  $n$  țării,  $n \in N^*$ , să se precizeze dacă i se poate asocia un graf orientat; în caz afirmativ, să se definească graful corespunzător.
- Având la dispoziție toate stelele, să se precizeze dacă li se poate asocia un graf orientat; justificați răspunsul.
- Pentru graful reprezentat în figura de mai jos



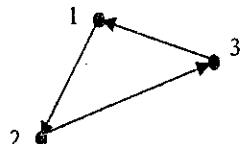
- precizați multimea nodurilor;
- precizați multimea arcelor;
- dăți exemple de noduri adiacente;
- pentru fiecare arc precizați extremitățile sale;
- dăți exemple de arce incidente.

### 4.2.2. Noțiunea de graf parțial

- Să se determine două grafuri parțiale ale grafului de mai jos:



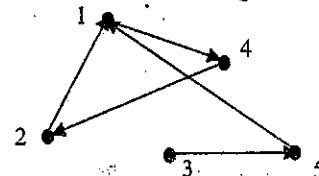
- Să se determine toate grafurile parțiale ale grafului de mai jos:



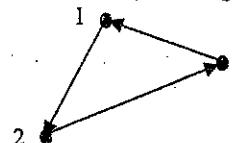
- Fie  $G$  un graf orientat, cu  $n$  vârfuri și  $m$  arce. Să se determine numărul grafurilor parțiale ale grafului  $G$ .

### 4.2.3. Noțiunea de subgraf

- Să se determine două subgrafuri ale grafului de mai jos:



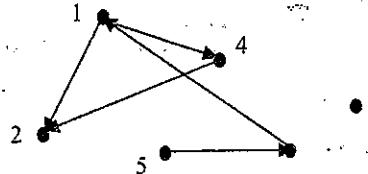
- Să se determine toate subgrafurile grafului de mai jos:



- Fie  $G$  un graf orientat, cu  $n$  vârfuri și  $m$  arce. Să se determine numărul subgrafurilor grafului  $G$ .

### 4.2.4. Gradul unui vârf

- Fiind dat graful de mai jos, să se determine pentru fiecare vârf  $x$ :  $d^+(x)$ ,  $d^-(x)$ ,  $\Gamma^+(x)$ ,  $\Gamma^-(x)$ ,  $\omega^+(x)$  și  $\omega^-(x)$ ; să se precizeze vârfurile izolate.



Un graf orientat cu proprietatea că între oricare două vârfuri distincte  $x$  și  $y$  există un arc și numai unul se numește graf turneu.

- Pentru un graf turneu cu  $n \geq 2$  vârfuri  $\{x_1, \dots, x_n\}$ , se notează  $r_i = d^-(x_i)$  și  $s_i = d^+(x_i)$ , pentru  $i=1 \dots n$ . Să se arate că:

a)  $r_i + s_i = n - 1$ ;

b)  $\sum_{i=1}^n r_i = \sum_{i=1}^n s_i = C_n^2$

c)  $\sum_{i=1}^n r_i^2 = \sum_{i=1}^n s_i^2$

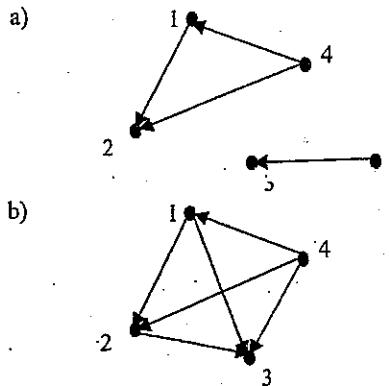
- Să se determine numărul grafurilor turneu cu  $n$  vârfuri.

- Fie  $G$  un graf orientat, cu  $n$  vârfuri și  $m$  arce, reprezentat prin matricea de adiacență. Să se realizeze programe, în Pascal, care:

- a) afișează gradele (exteroare și interioare) tuturor vârfurilor;
- b) afișează vârfurile de grad exterior par;
- c) afișează vârfurile izolate;
- d) afișează vârfurile terminale ( $d^+(x)=1$  și  $d^-(x)=0$  sau  $d^+(x)=0$  și  $d^-(x)=1$ );
- e) verifică dacă graful are vârfuri izolate;
- f) verifică dacă graful are vârfuri terminale;
- g) verifică dacă graful are vârfuri interioare (nu sunt nici izolate nici terminale);
- h) verifică dacă graful are toate vârfurile izolate;
- i) verifică dacă graful are toate vârfurile interioare (nu sunt nici izolate nici terminale);
- j) afișează gradul, exterior și interior, unui vârf dat;
- k) verifică dacă un vârf dat este terminal, izolat sau interior;
- l) afișează gradul exterior cel mai mare și toate vârfurile care au acest grad;
- m) afișează frecvența vârfurilor:
  - izolate : n1
  - terminale : n2
  - interioare : n3
- n) fiind dat sirul  $g_1, \dots, g_n$ , să se verifice dacă poate reprezenta sirul gradelor exterioare ale vârfurilor, în această ordine;
- o) fiind dat sirul  $g_1, \dots, g_n$ , să se verifice dacă poate reprezenta sirul gradelor exterioare ale vârfurilor (nu neapărat în această ordine).

#### 4.2.5. Graf complet

1. Fiind date grafurile de mai jos, să se precizeze care dintre ele este complet și să se justifice răspunsul.



3. Fie graful  $G$ , cu  $n$  vârfuri, dat prin matricea de adiacență. Să se realizeze un program care precizează dacă graful este complet.

## Capitolul 5

# PROGRAMAREA PE OBIECTE

## 5.1. ASPECTE TEORETICE

### Noțiuni teoretice despre obiecte

Programarea orientată pe obiecte, care în unele situații se mai întâlnește și sub denumirile: **programare orientată spre obiecte** sau **programare obiectuală** sau **OOP – Object Oriented Programming**, are la bază noțiunea de obiect.

În clasele anterioare, a fost studiată noțiunea de structură. După cum se știe, structura este un ansamblu de date de diverse tipuri standard sau definite de utilizator înainte de a fi folosite.

♦ **Exemplu:** Mai jos, este prezentată structura **fractie**, care are două câmpuri (date) de tipul int.

```
struct fractie{
    int nr;           (nr : numărător)
    int nm;          (nm : numitor)
};
```

Dacă la o structură oarecare, pe lângă date, se mai introduc în componența sa și subprograme (funcții), iar în sintaxa sa în loc de cuvântul cheie **struct** se scrie **class**, atunci se ajunge la un un tip abstract numit **class**. Având în vedere aceste amănunte, se poate, cu ușurință, imagina structura de definire a tipului **class**, pe care o prezentăm mai jos.

**Definiție.** **Class** este un concept superior celui de **structură**, care se prezintă sub forma unui ansamblu format din date și subprograme (funcții), definit conform următoarei structuri sintactice:

```
class{
    [private]
    date
    metode
    [protected]
    date
    metode
    [public]
    date
    metode
};
```

**Observații:** 1. Datele care intră în definiția unui tip abstract class se numesc date membre iar funcțiile din componentă sa se numesc funcții membre sau metode.

2. Clasa este un tip abstract iar obiectul este instantiere a clasei.

Public, protected și private, întâlniți în definiția de mai sus, sunt modificatori de acces, fiecare având rol până la întâlnirea altuia sau până la sfârșitul definiției. Aceștia au fost trecuți între paranteze drepte deoarece în definiția particulară a unui tip abstract class pot lipsi. Ei au următoarea semnificație:

- private** : interzice accesul la date și metode din afara obiectului; membrii din acest domeniu sunt accesibili numai funcțiilor membru ale clasei.
  - public** : permite accesul la date și metode din afara obiectului din orice loc din domeniul în care este vizibilă definiția clasei;
  - protected** : membrii din acest domeniu sunt accesibili numai funcțiilor membru ale clasei și funcțiilor membru ale claselor dervate din aceasta.
- În cazul în care nu apare nici un modificador, este interzis accesul.

♦ Exemplu: Mai jos, este prezentată clasa fractie, care are în componență să două date de tipul int și două funcții.

```
class fractie{
    public:
        int nr;
        int nm;
        void cit();
        void afis();
};
```

#### Comentarii:

Conform observației de mai sus, dacă luăm în calcul definiția clasei prezentată anterior, putem face următoarele precizări:

- datele nr și nm reprezintă pentru clasa "fractie" datele membre ale sale;
- funcțiile cit() și afis() reprezintă pentru clasa "fractie" metodele sale.

**Observații:** 1. Declararea unor obiecte se face întocmai ca declararea unor variabile.  
 2. Presupunând că avem la dispoziție obiectul "ob" care are în componență să datele data1 și data2, precum și funcțiile metoda1() și metoda2(), accesul la acestea se poate face astfel:  
 ob.data1, ob.data2, ob.metoda1() și ob.metoda2(),  
 deci, altfel spus, în scriere, la apel, numele obiectului este separat prin (punct) de numele datei sau metodelor la care se face referire.

#### ♦ Exemplu:

Fie:

```
class fractie{
    public:
        int nr;
        int nm;
        void cit();
        void afis();
};
```

fractie f;

Referirea la membrii obiectului "f" se face astfel:

f.nr    f.nm    f.cit();    f.afis();

Programarea orientată pe obiecte îmbină trei concepte fundamentale:

1. **încapsularea**
2. **moștenirea**
3. **polimorfism**

pe care le vom prezenta în continuare.

### Încapsularea

**Definiție:** Proprietatea obiectelor, de a îngloba într-o singură structură sintactică atât date cât și metodele care prelucrează eventual aceste date, se numește **încapsulare**.

**Observații:** 1. Un principiu de bază al OOP cere ca accesul la datele unui obiect să fie făcut exclusiv prin metodele sale.  
 2. Dacă avem la dispoziție un tip abstract class cu metoda "metoda1()", definirea metodei "metoda1()" o putem face:  
 a) în cadrul clasei atunci când **corpul funcției nu este mare**, altfel spus, **nu contine instrucțiuni de ciclare**; avem, în acest caz, de-a face cu funcții inline (la apel sunt înlocuite cu codul lor).  
 b) **în exteriorul funcției** atunci când **corpul funcției este mare**; în acest caz, se folosește operatorul de rezoluție ":" în antetul funcției astfel:  
**tip\_funcție nume\_clasa::nume\_funcție(...)**  
 În cazul în care metoda este definită în exteriorul clasei, în cadrul clasei se trece prototipul metodei.

♦ Exemplu: În continuare, vom prezenta un program în care sunt scoase în evidență observațiile prezentate mai sus și pe care-l comentăm astfel:

– se construiește clasa **fractie** care conține:

*datele:*

nr și nm care reprezintă numărătorul și numitorul unei fracții;

*metodele:*

funcția **cit** de citire a datelor obiectului;

funcția **afis** de afișare a datelor obiectului;

funcția **numarator** care returnează numărătorul fracției reprezentată de obiectul curent (adică returnează valoarea datei nr);

funcția **numitor** care returnează numitorul fracției reprezentată de obiectul curent (adică returnează valoarea datei nm);

funcția **aduna** care modifică fracția reprezentată de datele obiectului curent adunând-o cu numărul întreg "n";

– se declară un obiect "f" asupra căruia ulterior se vor face următoarele operații:

– i se citesc datele;

– i se afișează datele;

– se adună fracția reprezentată de datele sale cu un număr natural "n";

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

```
class fractie{
public:
    int nr;
    int nm;

    void cit();
    void afis();
    void aduna(int );
    int numarator(){
        return nr;
    }
    int numitor(){
        return nm;
    }
};
```

```
void fractie::cit(){
    cin>>nr>>nm;
}
```

```
void fractie::afis(){
    cout<<nr<<" / "<<nm<<endl;
}
void fractie::aduna(int n){
    nr=nr+nm*n;
}

fractie f;
main(){
    clrscr();
    f.cit();
    f.afis();
    cout<<f.numarator()<<" / "<<f.numitor()<<endl;
    f.aduna(3);
    cout<<endl;
    f.afis();
    getch();
}
```

Comentarii: Dacă se citește 3 și 4, se afișează:

3/4

3/4

15/4

## Constructori

**Definire:** Constructorul unei clase este o metodă specială care are rolul de a permite declararea obiectelor de tipul clasei respective.

### Caracteristici ale constructorilor

Constructorii se clasifică astfel:

1. constructor generat implicit
2. constructor definit de utilizator

#### 1. Constructor generat implicit

Are următoarele caracteristici:

- a) Se generează automat în lipsa oricărui alt constructor. Altfel spus, se generează atunci când programatorul nu a definit nici un constructor.

- b) Nu are parametrii formali.
- c) Nu permite inițializarea, la declarare, a obiectelor.

♦ Exemplu de clasă care are un constructor generat implicit.

```
#include <stdio.h>
#include <conio.h>
class fractie{
public:
    int nr;
    int nm;
};

fractie f1, f2;
main(){
    clrscr();
    getch();
}
```

**Comentarii:** La declararea obiectelor f1 și f2, s-a apelat constructorul generat implicit deoarece nu există nici un alt constructor definit de programator.

## 2. Constructor definit de utilizator

Are următoarele caracteristici:

- a) Are întotdeauna numele clasei din care face parte.
- b) Este o funcție care nu întoarce nici un rezultat dar, cu toate acestea, nu se trece în antet cuvântul cheie void.
- c) Este o funcție care poate să aibă sau să nu aibă parametrii. În cazul în care nu are parametrii, la declararea unui obiect se apelează acesta nu cel generat implicit.
- d) Într-o clasă pot fi definiți mai mulți constructori. În acest caz, trebuie să se aibă în vedere tipul și numărul parametrilor pentru că aceștia fac diferențierile la declararea unui obiect.

♦ Exemplu de clasă care are mai mulți constructori definiți de utilizator.

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

class fractie{
public:
    int nr;
    int nm;
```

```
fractie(){
    cout<<"constructor fara parametrii"<<endl;
}
fractie(int a, int b){
    nr=a; nm=b;
    cout<<"constructor cu doi parametri intregi"<<endl;
}
fractie(int a){
    nr=a; nm=1;
    cout<<"constructor cu un parametru intreg"<<endl;
}
main(){
    clrscr();
    fractie f1;
    fractie f2(7,9);
    fractie f3(7);
    cout<<endl<<endl;
    cout<<"S-a terminat programul"<<endl;
    getch();
}
```

**Comentarii:** După execuția programului, pe ecran, se va afișa:

```
constructor fara parametrii
constructor cu doi parametri intregi
constructor cu un parametru intreg

S-a terminat programul
```

deoarece:

- la declararea obiectului f1, s-a apelat constructorul fără parametrii definit de programator pentru că nu a avut valori de inițializare; dacă acest constructor nu ar fi existat, s-ar fi apelat constructorul generat implicit, de care am vorbit mai sus;
- la declararea obiectului f2, s-a apelat constructorul cu doi parametri definit de programator pentru că a avut două valori de inițializare;
- la declararea obiectului f3, s-a apelat constructorul cu un parametru definit de programator pentru că a avut o valoare de inițializare.

## Destructori

**Definiție:** Destructorul unei clase este o metodă specială care face operațiile legate de eliberarea memoriei ocupată de un obiect de tipul clasei respective atunci când acesta nu mai este folosit.

### Caracteristici ale destructorilor

Destructorii se clasifică astfel:

1. destructor generat implicit
2. destructor definit de utilizator

#### 1. Destructor generat implicit

Se generează automat în lipsa oricărui alt destructor. Altfel spus, se generează atunci când programatorul nu a definit nici un destructor.

♦ Exemplu de clasă care are un destructor generat implicit.

```
#include <stdio.h>
#include <conio.h>
class fractie{
public:
    int nr;
    int nm;
};

fractie f1;
main(){
    clrscr();
    if (3) {
        fractie f2;
    }
    getch();
}
```

**Comentarii:** Pentru dezalocarea spațiului ocupat de obiectele f1 și f2, s-a apelat destructorul generat implicit. Acesta s-a apelat prima dată pentru obiectul f2 (f2 se distrugă odată cu terminarea blocului în care a fost declarat) și a doua oară pentru obiectul f1 (f1 se distrugă după execuția ultimei instrucțiuni din main() pentru că este declarat global).

#### 2. Destructor definit de utilizator

Are următoarele caracteristici:

- a) Are întotdeauna numele clasei din care face parte, dar precedat de caracterul ~.
- b) Se apelează automat când durata de viață a unui obiect se încheie.
- c) Este o funcție care nu are parametri formali.

d) Într-o clasă poate fi definit decât un destructor.

♦ Exemplu de clasă care are definit un destructor.

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

class fractie{
public:
    int nr;
    int nm;

    fractie(){
        cout<<"constructor fara parametrii"<<endl;
    }

    fractie(int a, int b){
        nr=a; nm=b;
        cout<<"constructor cu doi parametri intregi"<<endl;
    }

    ~fractie(){
        cout<<"s-a distrus un obiect"<<endl;
    }
};

fractie f1;
main(){
    cout<<"incepe programul"<<endl<<endl;
    if (3){
        cout<<"incepe secventa"<<endl;
        fractie f2(7,9);
    }
    cout<<endl<<endl;
    cout<<"S-a terminat programul"<<endl;
}
```

**Comentarii:** După execuția programului, pe ecran, se va afișa:

constructor fara parametrii  
incepe programul

incepe secventa  
constructor cu doi parametri intregi  
s-a distrus un obiect

s-a terminat programul  
s-a distrus un obiect

deoarece:

se apelează constructorul fără parametrii la declararea lui f1, se afișează textul de începere a programului, se afișează textul de începere a secevnței, se apelează constructorul cu doi parametrii pentru f2, se apelează destritorul pentru f2 (este declarat local), s-au produs două salturi, se afișează mesajul de terminare a programului și se apelează destritorul pentru f1 (este declarat global).

### Functii prieten

**Definicie.** Se numește **functie prieten** a unei clase aceea funcție care, deși nu face parte din clasa respectivă, poate accesa datele private ale acesteia.

#### Caracteristici ale funcțiilor prieten

- Pentru ca o funcție să devină **functie prieten** a unei clase, trebuie:
  - să se scrie în cadrul clasei prototipul ei precedat de cuvântul cheie **friend**
  - să se scrie definiția acesteia în afara clasei
- O funcție prieten accesază datele private ale unei clase prin intermediul unui obiect pe care-l primește ca parametru.

♦ Exemplu de clasă care are o funcție prieten.

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

class fractie{
    int nr;
    int nm;
public:
    fractie(int a, int b){
        nr=a; nm=b;
    }
    int numitor(){
        return nr;
    }
}
```

### PROGRAMAREA PE OBIECTE

```
int numitor(){
    return nm;
}
friend void inmultire(int , fractie &);

void inmulteste(int n, fractie & f){
    f.nr=f.nr*n;
}

fractie f1(4,5);
main(){
    inmulteste(3,f1);
    cout<<f1.numarator()<<" / "<<f1.numitor();
}
```

**Comentarii:** După execuția programului, pe ecran, se va afișa:

12/5

deoarece:

“**functia prieten inmulteste**” înmulțește numărătorul unei fracții cu numărul reprezentat de primul ei parametru.

### Moștenirea

**Definicie.** Proprietatea claselor, prin care o clasă nou construită preia datele și metodele unei alte clase, definite deja, se numește **moștenire**.

Acest mecanism mai este întâlnit și sub numele de **derivarea claselor**.

**Observație.** Având în vedere proprietatea de mai sus, tragem concluzia că dacă avem la dispoziție o clasă, de exemplu **clasaM** (clasa moștenită), putem construi o altă clasă, de exemplu **clasaD** (clasa derivată), care, pe lângă datele și metodele proprii, poate prelua datele și metodele clasei **clasaM**, astfel:

La definirea clasei **clasaD** se scrie

```
class clasaD : public clasaM
```

♦ **Exemplu:** În continuare, vom prezenta un program în care având la dispoziție clasa **fractieM**, cu datele "nr" și "nm" și cu metodele "numitor()" și "numitor()", construim clasa **fractieD** care pe lângă metodele sale proprii "cit()" și "afis()" preia datele și metodele clasei **fractieM**.

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
class fractieM{
public:
    int nr;
    int nm;
    int numarator(){
        return nr;
    }
    int numitor(){
        return nm;
    }
};
class fractieD : public fractieM{
public:
    void cit(){
        cin>>nr>>nm;
    }
    void afis(){
        cout<<numarator()<<" / "<<numitor();
    }
};
fractieD f;
main(){
    clrscr();
    f.cit();
    f.afis();
    getche();
}
```

**Comentarii:** La execuția programului dacă se tastează 3 și 4, pe ecran, se va afișa:

3/4

**Observație:** În cazul în care o clasă, de exemplu **clasaD**, preia datele și metodele unui altă clasa definită deja, de exemplu **clasaM**, se numește **clăsă derivată**.

**Exemplu:** În cazul claselor din programul prezentat mai sus, clasa **fractieD** se numește **clăsă derivată** a clasei **fractieM**, iar clasa **fractieM** se numește **clăsă moștenitoare** de clasa **fractieD**.

**Observație:** Un lucru extrem de important îl reprezintă faptul că **instructiunea de atribuire funcționează și cu variabile obiect**, numai că trebuie să se ia cont de următoarele:

Instructiunea **O1 = O2;** (unde O1 și O2 sunt variabile obiect) este corectă dacă:

1. O1 și O2 sunt obiecte de același tip clasă;
2. O1 este obiect al clasei moștenite iar obiectul O2 este obiect al clasei derivate;

este incorrectă dacă:

1. O1 și O2 sunt obiecte total diferite (ne referim în acest caz la obiectele care nu moștenesc nimic unul de la celălalt);
2. O1 este obiect al clasei derivate iar obiectul O2 este obiect al clasei derivate (este normal să fie asta, deoarece, descendental are, eventual, mai multe date, care ar rămâne necompletate).

♦ **Exemplu:** Dacă luăm în calcul obiectele **fractieM** și **fractieD**, definite în programul de mai sus, putem spune:

Dacă      fractieM f, g;  
              fractieD f1, g1;

următoarele instrucțiuni sunt corecte:

f=g; g1=f1; f=f1; f=g1;

următoarele instrucțiuni nu sunt corecte:

f1=f; g1=f;

**Observație:** Dacă plecând de la clasa **clasaM**, cu datele **data1** și cu metoda **metoda1**, obținem clasa derivată **clasaD**, în care redefinim data **data1** și metoda **metoda1**, și definim un obiect obiect de tipul **clasaD**, pentru a referi:

<b>data1</b> din <b>clasaD</b> scriem:	<b>obiect.data1</b>
<b>metoda1</b> din <b>clasaD</b> scriem:	<b>obiect.metoda1</b>
<b>data1</b> din <b>clasaM</b> scriem:	<b>obiect.clasaM::data1</b>
<b>metoda1</b> din <b>clasaD</b> scriem:	<b>obiect.clasaM::metoda1</b>

♦ **Exemplu:** În continuare, vom prezenta un program în care având la dispoziție clasa **fractieM**, cu datele nr și nm și cu metoda test(), construim clasa **fractieD**, în care redefinim data nr și metoda test().

```
#include <iostream.h>
class fractieM{
public:
    int nr, nm;
    void test(){
        cout<<"clasa moștenitoare"<<endl;
    }
};
```

```

class fractieD : public fractieM{
    public:
        int nr;
        void test(){
            cout<<"clasa derivata"<<endl;
        }
};

fractieD f;

main(){
    clrscr();
    f.nr=4;
    f.fractieM::nr=10;
    f.test();
    f.fractieM::test();
    cout<<f.nr<< " <<f.fractieM::nr;
    getch();
}

```

**Comentarii:** La execuția programului, se va afișa:

```

clasa derivata
clasa mostenita
4 10

```

### Despre constructori și destructori în cazul moștenirilor

#### Despre constructori

Presupunând că având la dispoziție clasa **clasaM**, s-a obținut prin derivare clasa **clasaD**, atunci când se declară un obiect de tipul clasei **clasaD**, constructorii se comportă astfel:

- în cazul în care cele două clase au câte un constructor fără parametrii, se apelează mai întâi constructorul clasei **clasaM** și apoi constructorul clasei **clasaD**;
- în cazul în care cele două clase au fiecare mai mulți constructori, se apelează constructorul fără parametrii din clasa **clasaM** și apoi constructorul corespunzător initializării din clasa **clasaD**.

#### Despre constructori

Presupunând că având la dispoziție clasa **clasaM** s-a obținut prin derivare clasa **clasaD**, atunci când se renunță la un obiect de tipul clasei **clasaD**, destructorii se apelează

în ordinea inversă apelării constructorilor, adică se apelează mai întâi destructorul clasei **clasaD** și apoi destructorul clasei **clasaM**.

♦ **Exemplu:** În continuare, vom prezenta un program care scoate în evidență cele spuse mai sus.

```

#include <stdio.h>
#include <conio.h>
#include <iostream.h>
class fractieM{
    public:
        fractieM(){
            cout<<"constructorul clasei fractieM"<<endl;
        }
        ~fractieM(){
            cout<<"destructorul clasei fractieM"<<endl;
        }
};

class fractieD : public fractieM{
    public:
        fractieD(){
            cout<<"constructorul clasei fractieD fara param"<<endl;
        }
        fractieD(int){
            cout<<"constructorul clasei fractieD cu param"<<endl;
        }
        ~fractieD(){
            cout<<"destructorul clasei fractieD"<<endl;
        }
};

fractieD f;
main(){
    clrscr();
    fractieD f(2), g;
    getch();
}

```

**Comentarii:** La execuția programului, se va afișa:

```

constructorul clasei fractieM
constructorul clasei fractieD cu param

```

```

constructorul clasei fractieM
constructorul clasei fractieD fara param
destructorul clasei fractieD
destructorul clasei fractieM
destructorul clasei fractieD
destructorul clasei fractieM

```

## Polimorfism

**Polimorfismul** se realizează prin redefinirea funcțiilor și a operatorilor, acceptat fiind, în limbajul C++, prin intermediul **funcțiilor virtuale**.

### Funcții virtuale

**Funcțiile virtuale** sunt funcțiile care pot fi redefinite într-o clasă derivată fără a li se modifică numărul sau tipul parametrilor.

Concret, avem de-a face cu polimorfismul în situații de genul:

Având la dispoziție clasa **clasaM**, cu metoda **Met1** care apelează metoda **apell** (a acestei clase), și definind altă clasă, **clasaD**, care să o moștenească pe aceasta, în care se redifineste metoda **apell**, se pune problema care dintre metodele **apell** se va apela la referirea metodei **Met1** a unui obiect de tipul **clasaM**?

Răspunsul la probleme de acest gen este relevat în următoarele două exemple.

```

#include<conio.h>
#include<iostream.h>
class fractieM{
public:
    int nr, nm;
    void scrie(){
        numarator();
        cout<<endl;
        numitor();
    }
    void numarator(){
        cout<<nr;
    }
    void numitor(){
        cout<<nm;
    }
};

```

```

class fractieD: public fractieM{
public:
    void numarator(){
        cout<<"numarator = "<<nr;
    }
    void numitor(){
        cout<<"numitor = "<<nm;
    }
};

fractieD o;
main(){
    clrscr();
    cin>>o.nr>>o.nm;
    o.scrie();
    getch();
}

```

**Comentarii** Dacă la execuție se citesc valorile 4 și 5, se afișează:

```

4
5

```

deoarece au fost apelate metodele **numarator()** și **numitor()** din **fractieM**.

Dacă lucrurile ar sta numai așa, ar apărea întrebarea firească "De ce se mai redifineste metodele în clasa derivată, dacă se apelează tot cele din clasa moștenită?"

Pentru a schimba lucrurile, adică să se apeleze metodele redefinite în clasa derivată, trebuie să se facă apel la **funcțiile virtuale** (sunt funcțiile la care se scrie cuvântul **virtual** în fața tipului lor), ca în exemplul de mai jos.

```

#include<conio.h>
#include<iostream.h>
class fractieM{
public:
    int nr, nm;
    void scrie(){
        numarator();
        cout<<endl;
        numitor();
    }
    virtual void numarator(){
        cout<<nr;
    }
};

```

```

virtual void numitor(){
    cout<<nm;
}

class fractieD: public fractieM{
public:
    virtual void numarator(){
        cout<<"numarator = "<<nr;
    }
    virtual void numitor(){
        cout<<"numitor = "<<nm;
    }
};

fractieD o;
main(){
    clrscr();
    cin>>o.nr>>o.nm;
    o.scrieO();
    getch();
}

```

**Comentarii** Dacă la execuție se citesc valorile 4 și 5, se afișează:

```

numarator =4
numitor =5

```

deoarece au fost apelate metodele numarator() și numitor() din fractieD.

## 5.2. PROBLEME PROPUSE

Să se definească tipurile abstracte class, înzestrăte cu date și metode, astfel încât să permită prelucrări de orice fel asupra

1. numerelor complexe
2. numerelor raționale
3. stringurilor
4. punctelor din plan
5. punctelor din spațiu

# CUPRINS

<b>1. ALOCARE DINAMICĂ .....</b>	<b>5</b>
1.1. Aspecte teoretice .....	5
1.2. Liste simplu înlățuită .....	17
1.3. Probleme propuse .....	25
1.4. Indicații și răspunsuri .....	32
1.5. Liste dublu înlățuită .....	72
1.6. Liste circulare .....	80
1.7. Stive și cozi .....	84
<b>2. GRAFURI NEORIENTATE .....</b>	<b>87</b>
2.1. Aspecte teoretice .....	87
2.1.1. Noțiunea de graf neorientat .....	87
2.1.2. Noțiunea de graf parțial .....	88
2.1.3. Noțiunea de subgraf .....	89
2.1.4. Gradul unui vârf .....	90
2.1.5. Graf complet .....	91

2.1.6. Graf bipartit .....	92
2.1.7. Graf bipartit complet .....	93
2.1.8. Reprezentarea grafurilor neorientate .....	94
2.1.9. Parcursarea grafurilor .....	103
2.1.10. Conexitate .....	110
2.1.11. Grafuri Hamiltoniene .....	115
2.1.12. Grafuri Euleriene .....	119
2.2. Probleme propuse.....	126
2.3. Indicații și răspunsuri .....	134
<b>3. ARBORI .....</b>	<b>179</b>
3.1. Aspecte teoretice .....	179
3.1.1. Noțiunea de arbore .....	179
3.1.2. Arborele parțial de cost minim .....	186
3.1.3. Arbori binari .....	190
3.1.4. Reprezentarea arborilor binari .....	193
3.1.5. Parcursarea arborilor binari .....	197
3.1.6. Arbori binari de căutare .....	205
3.2. Probleme propuse .....	215
3.3. Indicații și răspunsuri .....	219

<b>4. GRAFURI ORIENTATE .....</b>	<b>249</b>
4.1. Aspecte teoretice .....	249
4.1.1. Noțiunea de graf orientat .....	249
4.1.2. Noțiunea de graf parțial .....	251
4.1.3. Noțiunea de subgraf .....	252
4.1.4. Gradul unui vârf .....	253
4.1.5. Graf complet .....	256
4.1.6. Conexitate .....	257
4.1.7. Reprezentarea grafurilor orientate .....	261
4.1.8. Tare conexitate .....	276
4.1.9. Drumuri minime și maxime .....	281
4.2. Probleme propuse .....	292
<b>5. PROGRAMAREA PE OBIECTE .....</b>	<b>295</b>
5.1. Aspecte teoretice .....	295
5.2. Probleme propuse .....	312

