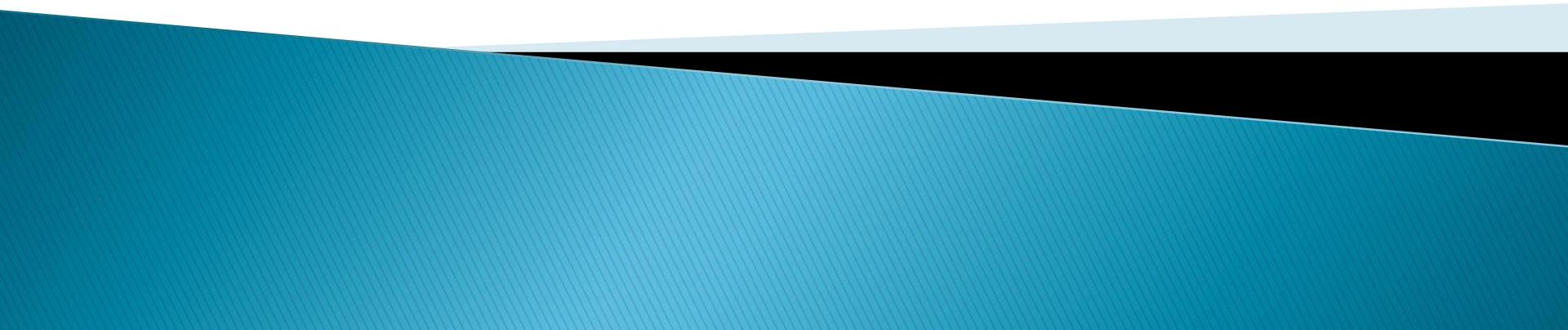


**Facultatea de Matematică și Informatică
Lecții de pregătire – Admitere FMI**

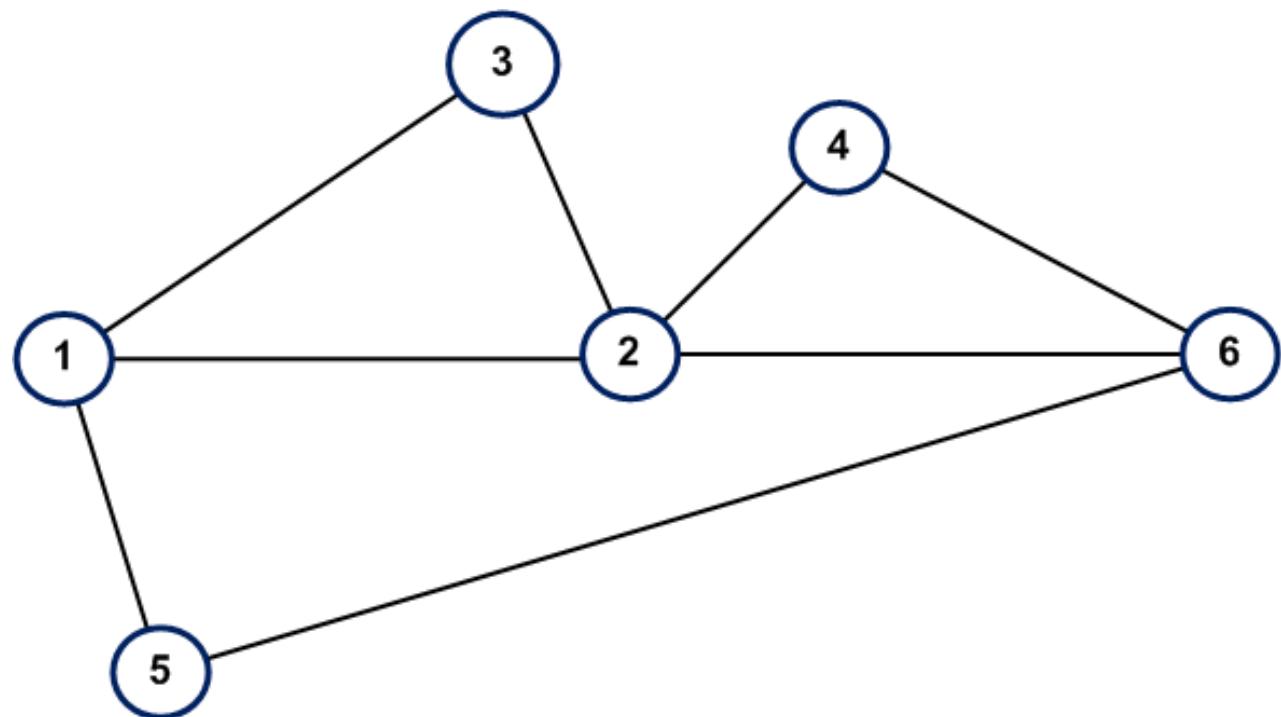
Grafuri

Definiții



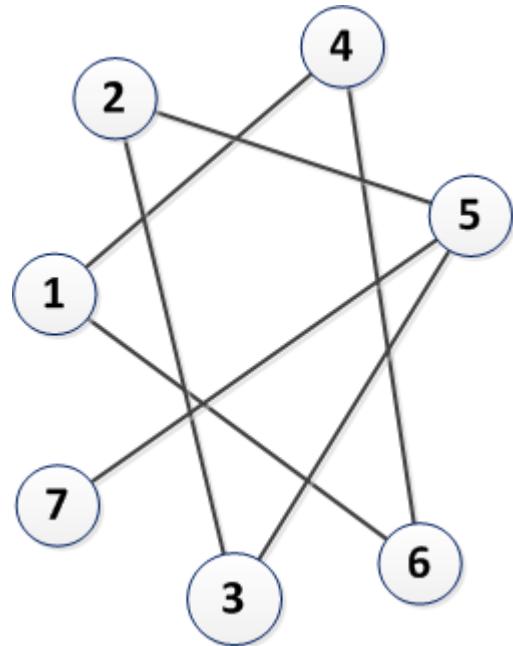
► Graf neorientat: $G = (V, E)$

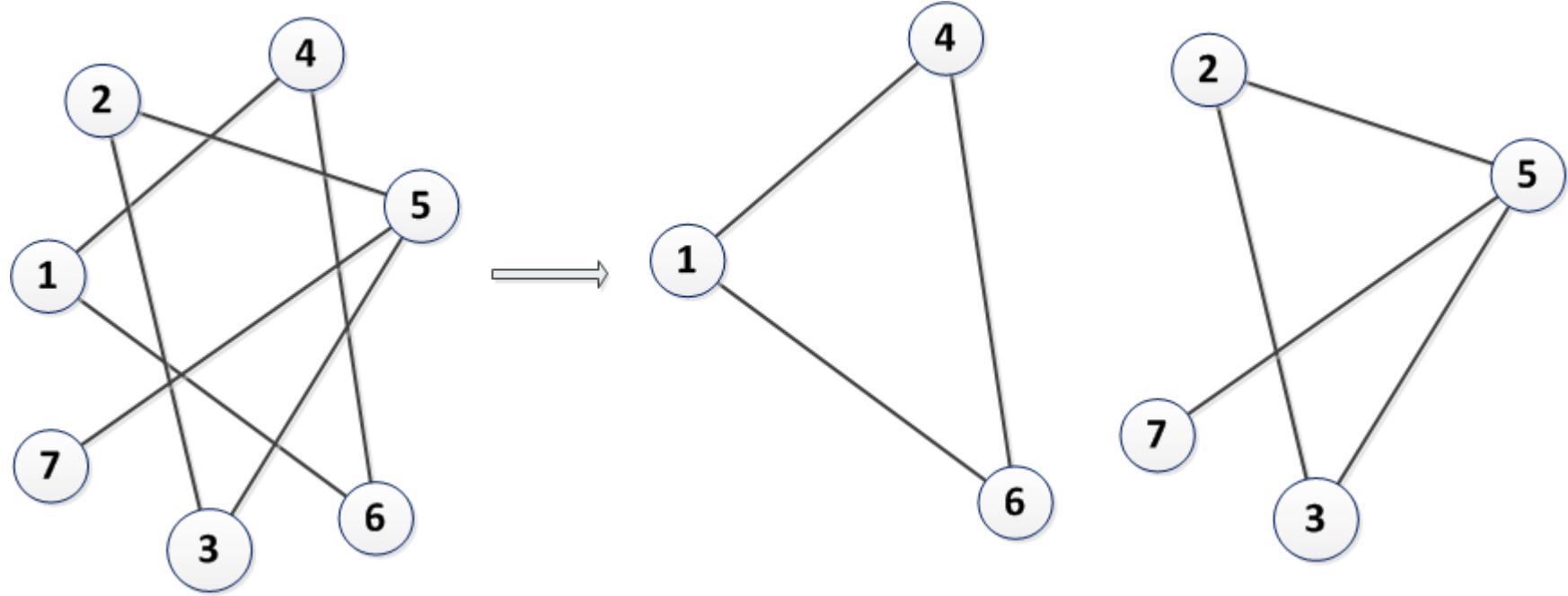
- vârf (nod), muchie
- gradul unui vârf
- vârfuri adiacente
- lanț
- ciclu
- distanță



► Graf neorientat: $G = (V, E)$

- graf conex
- componentă conexă

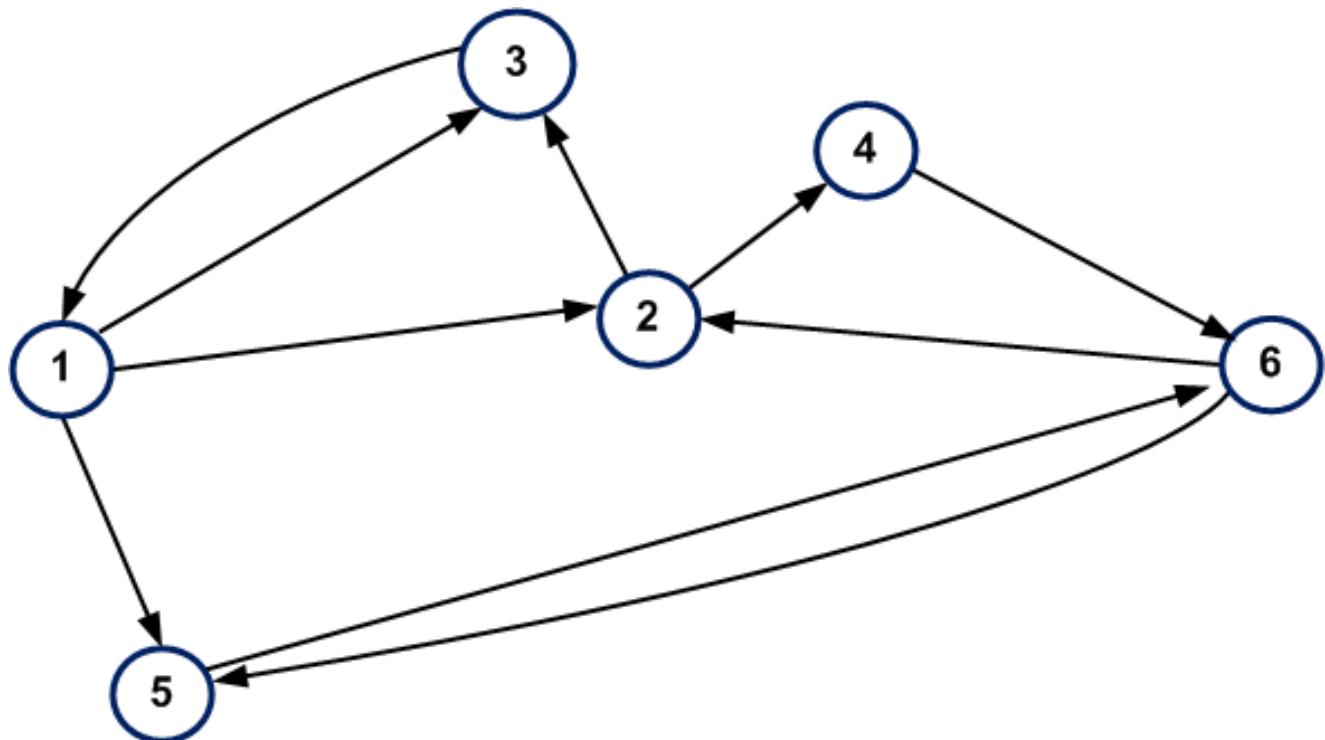




două componente conexe

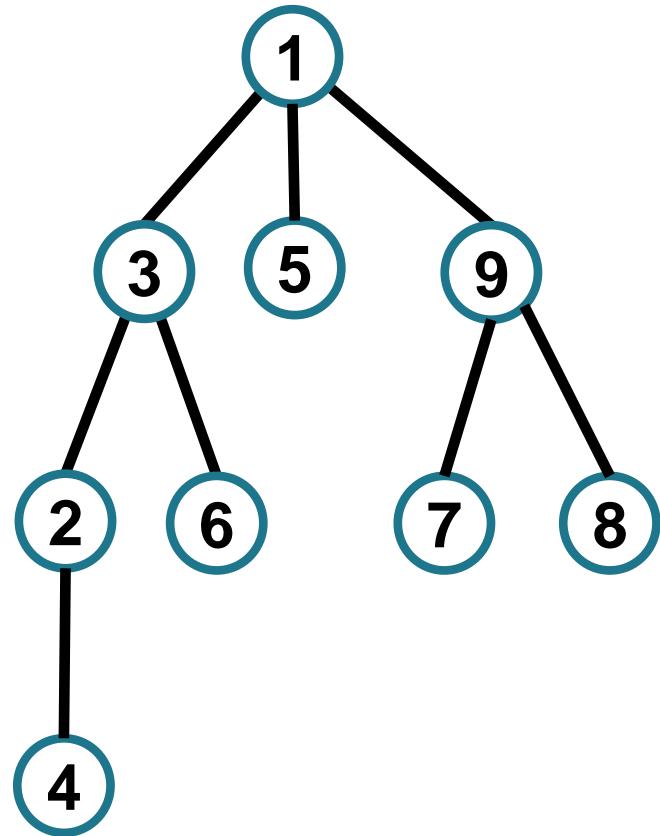
► Graf orientat: $G = (V, E)$

- vârf (nod), arc
- gradul unui vârf – intern, extern
- drum
- circuit



► Arbore

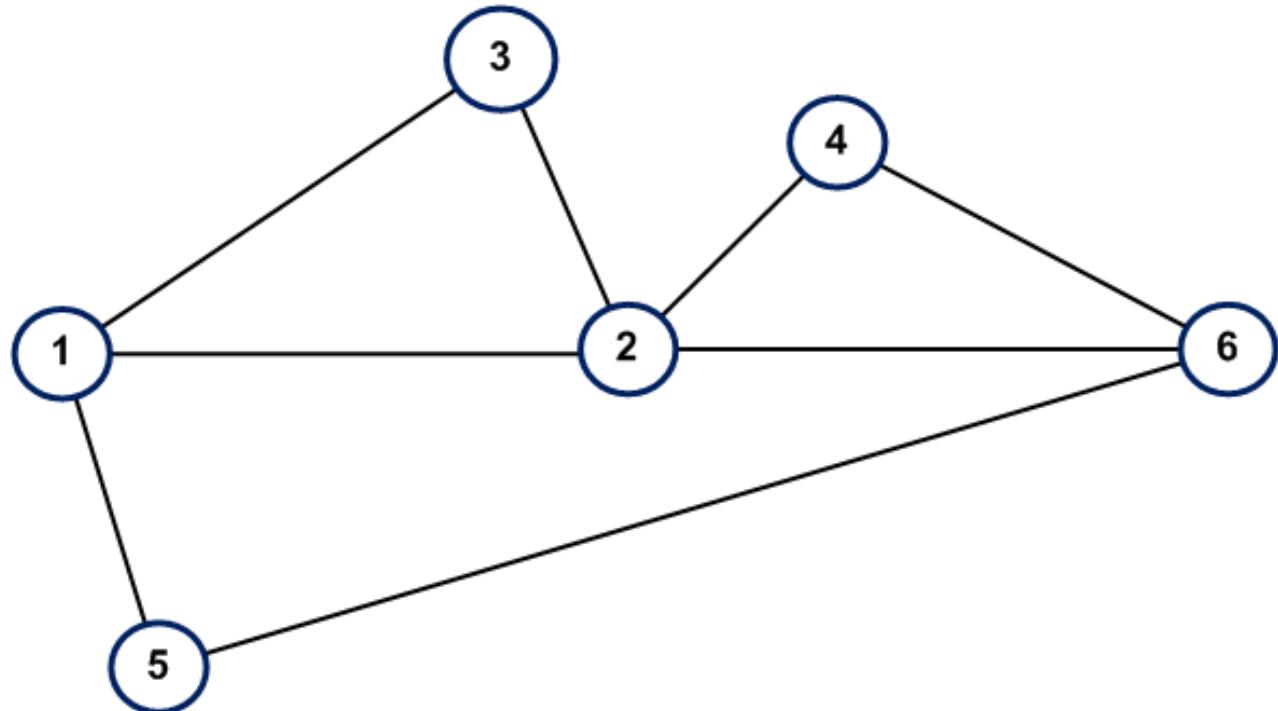
- arbore cu rădăcină
- fiu, tată
- ascendent, descendant
- frunză



Modalități de reprezentare

Reprezentarea grafurilor

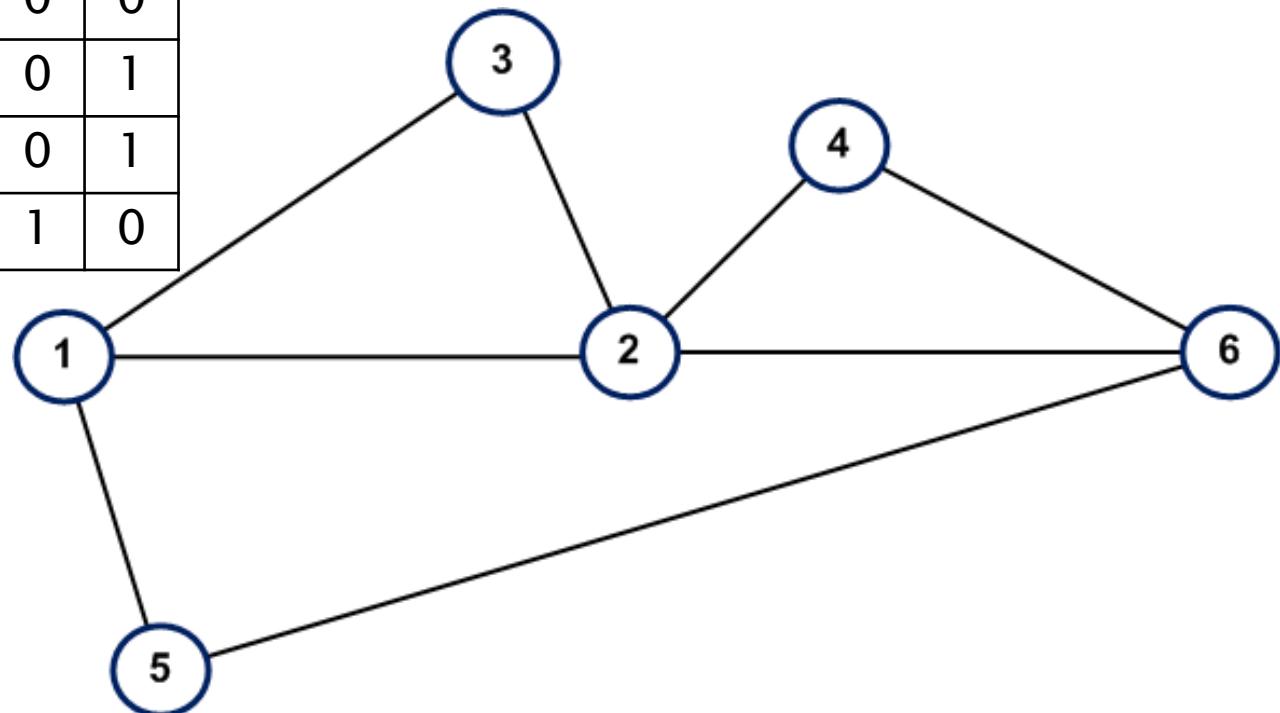
- ▶ Matrice de adiacență
- ▶ Liste de adiacență
- ▶ Listă de muchii/arce



Reprezentarea grafurilor

Matrice de adiacență

	1	2	3	4	5	6
1	0	1	1	0	1	0
2	1	0	1	1	0	1
3	1	1	0	0	0	0
4	0	1	0	0	0	1
5	1	0	0	0	0	1
6	0	1	0	1	1	0

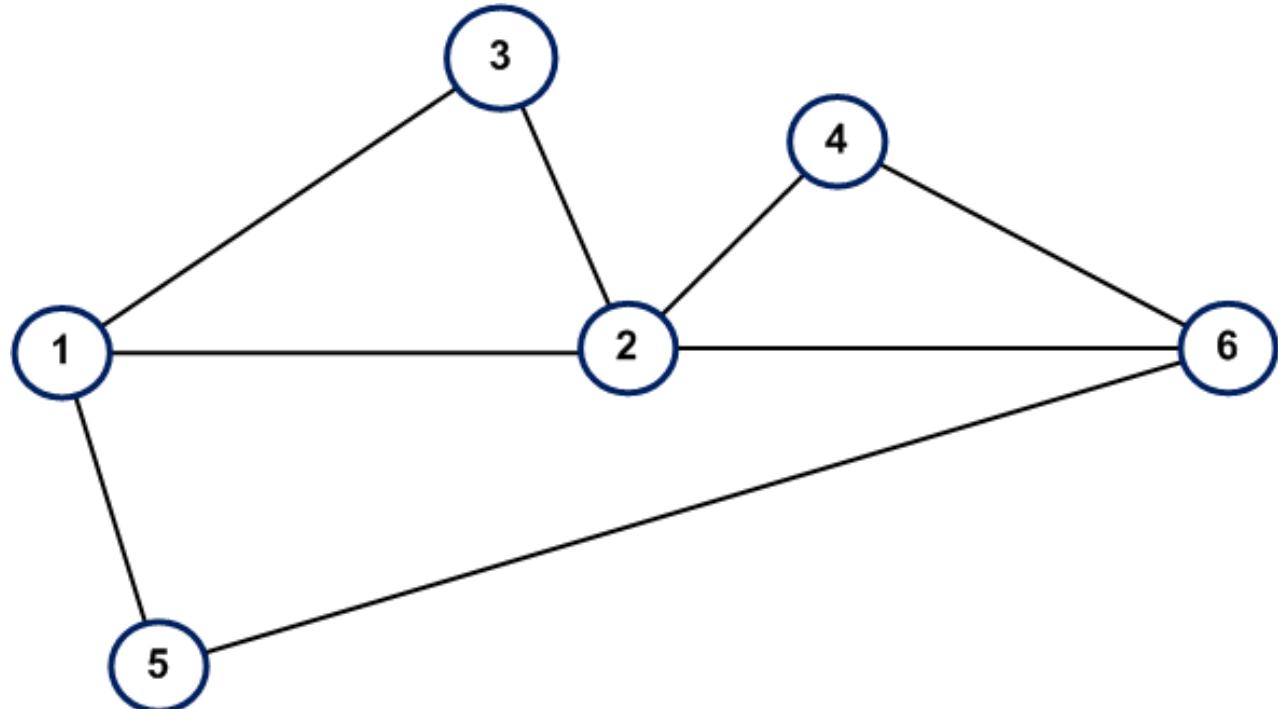


Reprezentarea grafurilor

Listă de adiacență

6 noduri:

- ▶ 2, 3, 5
- ▶ 1, 3, 4, 6
- ▶ 1, 2
- ▶ 2, 6
- ▶ 1, 6
- ▶ 2, 4, 5



Matrice de adiacență

▶ Construcția din lista de muchii

Intrare: n,m și muchiile

6

8

1 2

1 3

2 3

2 4

4 6

2 6

1 5

5 6

Matrice de adiacență

▶ Construcția din lista de muchii

```
int a[20][20], n, m;  
  
void citire() {  
    int i,j,x,y;  
  
    cin>>n>>m;  
    for(i=1;i<=n;i++)  
        for(j=1;j<=n;j++)  
            a[i][j]=0;  
  
}
```

```
var a:array[1..20,1..20] of integer;  
n,m:integer;  
procedure citire;  
var i,j,x,y:integer;  
begin  
    readln(n,m);  
    for i:=1 to n do  
        for j:=1 to n do  
            a[i,j]:=0;  
  
end;
```

Matrice de adiacență

▶ Construcția din lista de muchii

```
int a[20][20], n, m;

void citire() {
    int i,j,x,y;

    cin>>n>>m;
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            a[i][j]=0;

    for(i=1;i<=m;i++) {
        cin>>x>>y;
        a[x][y]=1;
        a[y][x]=1; //neorientat
    }
}
```

```
var a:array[1..20,1..20] of integer;
n,m:integer;
procedure citire;
var i,j,x,y:integer;
begin
    readln(n,m);
    for i:=1 to n do
        for j:=1 to n do
            a[i,j]:=0;

    for i:=1 to m do
begin
    readln(x,y);
    a[x,y]:=1;
    a[y,x]:=1; //neorientat
end;
end;
```

Matrice de adiacență

- ▶ Determinarea gradului unui vârf

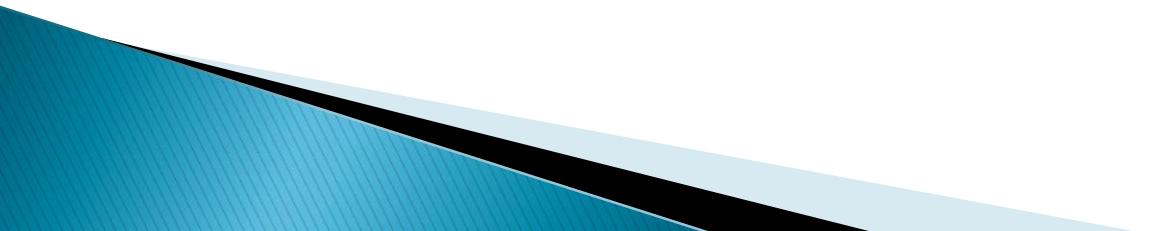
Matrice de adiacență

▶ Determinarea gradului unui vârf

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

```
function grad(x:integer):integer;  
var nr,j:integer;  
begin  
    nr:=0;  
    for j:=1 to n do  
        if a[x,j]=1 then  
            inc(nr);  
    grad:=nr;  
end;
```

Liste de adiacență – Construcție din lista de muchii



Liste de adiacență – Varianta 1

- ▶ Folosind tot matrice $n \times n$ pentru a le memora

```
int l[100][100], n,m;  
//l[i][0]=gradul varfului i  
void citire(){  
    int i,j,x,y,g;  
    cin>>n>>m;  
    for(i=1;i<=m;i++){  
        cin>>x>>y;  
    }  
}
```

```
var l: array[1..100,0..100] of integer;  
var n,m:integer;  
procedure citire;  
    var i,j,x,y,g:integer;  
begin  
    readln(n,m);  
    for i:=1 to n do l[i,0]:=0;  
    for i:=1 to m do begin  
        readln(x,y);  
    end;
```

```
int l[100][100], n,m;
//l[i][0]=gradul varfului i
void citire(){
    int i,j,x,y,g;
    cin>>n>>m;
    for(i=1;i<=m;i++){
        cin>>x>>y;
        /*l[x][0]=gradul lui x=la
al cateleau vecin am ajuns*/
        l[x][0]++;
        l[x][l[x][0]]=y;
    }
}
```

```
var l: array[1..100,0..100] of integer;
var n,m:integer;
procedure citire;
    var i,j,x,y,g:integer;
begin
    readln(n,m);
    for i:=1 to n do l[i,0]:=0;
    for i:=1 to m do begin
        readln(x,y);
        {l[x][0]=gradul lui x=la al cateleau vecin
        am ajuns}
        l[x,0]:=l[x,0]+1;
        l[x,l[x][0]]:=y;
    end;

```

```
int l[100][100], n,m;
//l[i][0]=gradul varfului i
void citire(){
    int i,j,x,y,g;
    cin>>n>>m;
    for(i=1;i<=m;i++){
        cin>>x>>y;
        /*l[x][0]=gradul lui x=la
al cateleau vecin am ajuns*/
        l[x][0]++;
        l[x][l[x][0]]=y;
        //neorientat
        l[y][0]++;
        l[y][l[y][0]]=x;
    }
}
```

```
var l: array[1..100,0..100] of integer;
var n,m:integer;
procedure citire;
    var i,j,x,y,g:integer;
begin
    readln(n,m);
    for i:=1 to n do l[i,0]:=0;
    for i:=1 to m do begin
        readln(x,y);
        {l[x][0]=gradul lui x=la al cateleau vecin
am ajuns}
        l[x,0]:=l[x,0]+1;
        l[x,l[x][0]]:=y;
        //neorientat
        l[y,0]:=l[y,0]+1; l[y,l[y,0]]:=x;
    end;

```

```

int l[100][100], n,m;
//l[i][0]=gradul varfului i
void citire(){
    int i,j,x,y,g;
    cin>>n>>m;
    for(i=1;i<=m;i++){
        cin>>x>>y;
        /*l[x][0]=gradul lui x=la
        al cateleau vecin am ajuns*/
        l[x][0]++;
        l[x][l[x][0]]=y;
        //neorientat
        l[y][0]++;
        l[y][l[y][0]]=x;
    }
    for(i=1;i<=n;i++){
        cout<<i<<" ";
        g=l[i][0];
        for(j=1;j<=g;j++)
            cout<<l[i][j]<<" ";
        cout<<endl;
    }
}

```

```

var l: array[1..100,0..100] of integer;
var n,m:integer;
procedure citire;
    var i,j,x,y,g:integer;
begin
    readln(n,m);
    for i:=1 to n do l[i,0]:=0;
    for i:=1 to m do begin
        readln(x,y);
        {l[x][0]=gradul lui x=la al cateleau vecin
        am ajuns}
        l[x,0]:=l[x,0]+1;
        l[x,l[x][0]]:=y;
        //neorientat
        l[y,0]:=l[y,0]+1; l[y,l[y,0]]:=x;
    end;
    for i:=1 to n do begin
        write(i,' ');
        g:=l[i,0];
        for j:=1 to g do write(l[i,j],' ');
        writeln;
    end; end;

```

Liste de adiacență – Varianta 2

- ▶ **Liste implementate – pointeri**

Liste de adiacență - Construcție din lista de muchii

```
struct nod{ int info;
            nod* urm; } ;
nod* l[100]; nod* p;
int n,m;
void citire(){
    int i,x,y;
    cin>>n>>m;
    for(i=1;i<=n;i++)
        l[i]=NULL;
    for(i=1;i<=m;i++) {
        cin>>x>>y;
        p=new nod;
        p->info=y;
        p->urm=l[x];
        l[x]=p;
        //neorientat
        p=new nod;
        p->info=x;
        p->urm=l[y];
        l[y]=p;
    }
}
```

```
type elem^nod;
nod=record  info:integer;
            urm:elem;
        end;
var l: array[1..100] of elem; var p:elem;
var n,m:integer;
procedure citire;
    var i,x,y:integer;
begin
    readln(n,m);
    for i:=1 to n do l[i]:=nil;
    for i:=1 to m do
    begin
        readln(x,y);
        new(p);
        p^.info:=y;
        p^.urm:=l[x];
        l[x]:=p; {neorientat}
        new(p); p^.info:=x;
        p^.urm:=l[y]; l[y]:=p;
    end;
end;
```

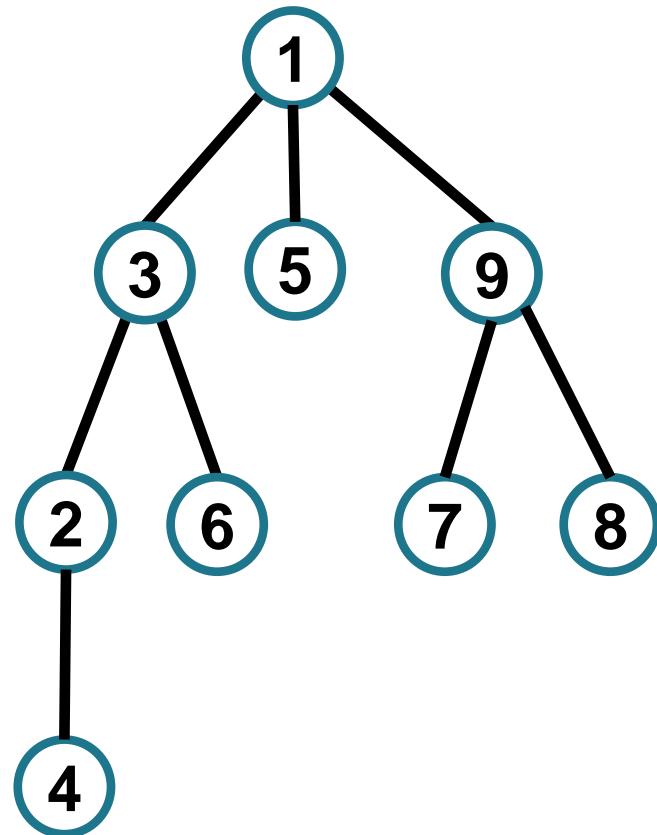
Liste de adiacență – Construcție din lista de muchii

```
//afisare liste
for(i=1;i<=n;i++) {
    cout<<i<<" : ";
    p=l[i];
    while(p!=NULL) {
        cout<<p->info<<" ";
        p=p->urm;
    }
    cout<<endl;
}
}//citire
```

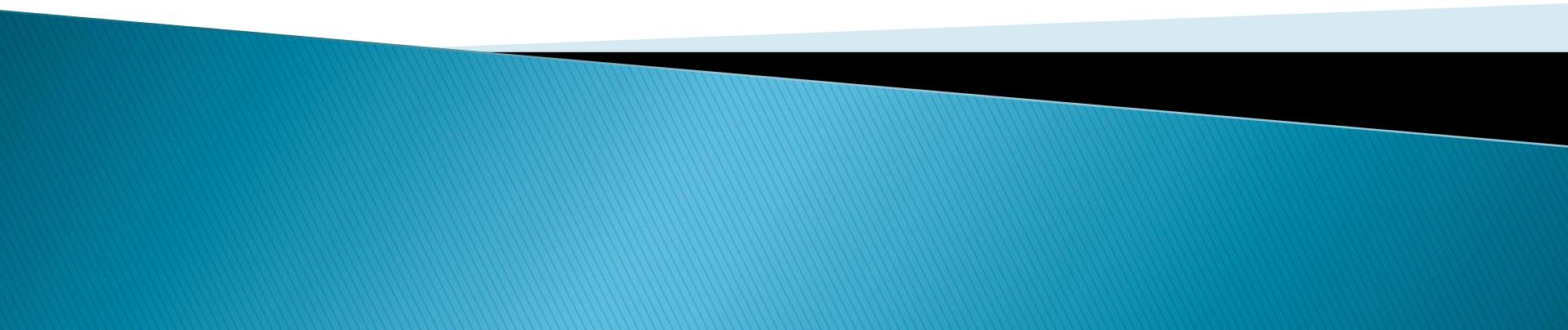
```
{afisare liste}
for i:=1 to n do
begin
    write(i,' : ');
    p:=l[i];
    while p<>nil do
begin
    write(p^.info,' ');
    p:=p^.urm;
end;
writeln;
end;
end;
```

Reprezentarea arborilor

- ▶ Vector tata
- ▶ Lista de fii
(descendenți direcți)



Arbori



Problemă



Dat un **vector tata** asociat unui arbore (arbore genealogic) să se determine:

- 1) rădăcina arborelui
- 2) frunzele arborelui
- 3) un lanț de la un vârf dat la rădăcină, afișat direct și invers = ascendenții unui vârf (afișați de la tată în sus / de la rădăcină în jos)
- 4) nivelul (generația) fiecărui vârf în arbore și înălțimea arborelui (=numărul de generații-1)

1) rădăcina arborelui

1) rădăcina arborelui



Rădăcina este vârful care are tata egal cu 0

1) rădăcina arborelui

```
int radacina(){
    int i;
    for(i=1;i<=n;i++)
        if(tata[i]==0)
            return i;
    return 0;
}
```

```
function radacina:integer;
var i:integer;
begin
    for i:=1 to n do
        if tata[i]=0 then
            radacina:=i;
end;
```

2) frunzele arborelui

2) frunzele arborelui



Frunzele sunt vârfurile care nu apar în vectorul tata

2) frunzele arborelui



**Varianta 1 – Testăm pentru fiecare vârf
 $i=1 \dots n$ dacă apare în vectorul tata**

Complexitate?

2) frunzele arborelui

```
void frunzel() {
    int i,j,ok;
    for(i=1;i<=n;i++) {
        ok=1;
        for(j=1;j<=n;j++)
            if (i==tata[j])
                ok=0;
        if (ok)
            cout<<i<<' ';
    }
    cout<<endl;
}
```

```
procedure frunzel;
var i,j:integer;
    ok:boolean;
begin
    for i:=1 to n do
    begin
        ok:=true;
        for j:=1 to n do
            if i=tata[j] then
                ok:=false;
        if ok then
            write(i,' ');
    end;
    writeln;
end;
```

2) frunzele arborelui



Varianta 1 – Complexitate: $O(n^2)$

2) frunzele arborelui



Varianta 2 – cu vector de apariții

Complexitate: $O(n)$

2) frunzele arborelui

```
void frunze() {  
    int i;  
    int v[100];  
  
    for(i=1;i<=n;i++)  
        v[i]=0;  
  
}
```

```
procedure frunze;  
var i:integer;  
    v:array[1..100] of boolean;  
begin  
    for i:=1 to n do  
        v[i]:=true;  
  
end;
```

2) frunzele arborelui

```
void frunze(){
    int i;
    int v[100];

    for(i=1;i<=n;i++)
        v[i]=0;

    for(i=1;i<=n;i++)
        if(tata[i]>0)
            v[tata[i]]=1;

}
```

```
procedure frunze;
var i:integer;
    v:array[1..100] of boolean;
begin
    for i:=1 to n do
        v[i]:=true;

    for i:=1 to n do
        if tata[i]>0 then
            v[tata[i]]:=false;

end;
```

2) frunzele arborelui

```
void frunze(){
    int i;
    int v[100];

    for(i=1;i<=n;i++)
        v[i]=0;

    for(i=1;i<=n;i++)
        if(tata[i]>0)
            v[tata[i]]=1;

    for(i=1;i<=n;i++)
        if(v[i]==0)
            cout<<i<<" ";
    cout<<endl;
}
```

```
procedure frunze;
var i:integer;
    v:array[1..100] of boolean;
begin
    for i:=1 to n do
        v[i]:=true;

    for i:=1 to n do
        if tata[i]>0 then
            v[tata[i]]:=false;

    for i:=1 to n do
        if v[i] then
            write(i,' ');
    writeln;
end;
```

3) lanț de la vârful x la rădăcină



3) lanț de la vârful x la rădăcină



Se urcă în arbore de la x la rădăcină folosind vectorul tata

3) lanț de la vârful x la rădăcină

```
//de la x la radacina
void drum(int x)
{
    while(x!=0) {
        cout<<x<<" ";
        x=tata[x];
    }
    cout<<endl;
}
```

```
{de la x la radacina}
procedure drum(x:integer);
begin
    while x<>0 do
    begin
        write(x, ' ');
        x:=tata[x];
    end;
    writeln;
end;
```

3) lanț de la vârful x la rădăcină

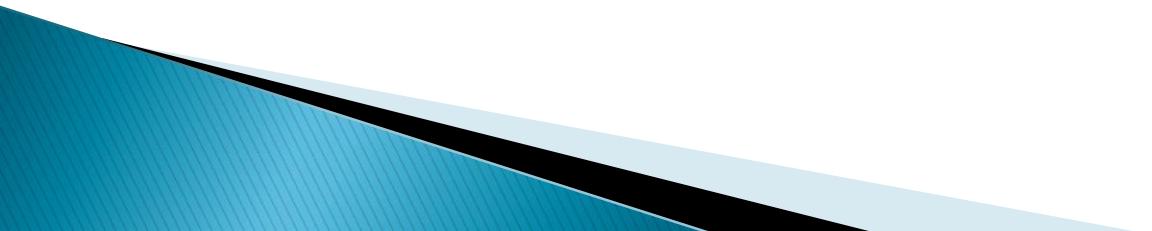
```
//de la x la radacina
void drum(int x)
{
    while(x!=0) {
        cout<<x<<" ";
        x=tata[x];
    }
    cout<<endl;
}
```

```
//de la radacina la x
void drumRec(int x)
{
    if(x!=0) {
        drumRec(tata[x]);
        cout<<x<<" ";
    }
}
```

```
{de la x la radacina}
procedure drum(x:integer);
begin
    while x<>0 do
    begin
        write(x, ' ');
        x:=tata[x];
    end;
    writeln;
end;

{de la radacina la x}
procedure drumRec(x:integer);
begin
    if x<>0 then
    begin
        drumRec(tata[x]);
        write(x, ' ');
    end;
end;
```

4) Înăltimea, niveluri



4) Înăltimea, niveluri



Varianta 1:

- pentru fiecare vârf determinăm lanțul până la rădăcină și lungimea acestuia
- nivelul unui vârf = nivelul tatălui + 1;

Varianta 1: nerecursiv

```
//de la x la radacina
int drumN(int x)
{
    int k=0;
    while(tata[x] !=0) {
        x=tata[x];
        k++;
    }
    return k;
}
```

Varianta 1: recursiv

```
int niv1[100]={0};  
  
//functie care returneaza nivelul lui x  
  
int drumNiv(int x) {  
    if(tata[x]==0) //radacina  
        return 0;  
    return 1+drumNiv(tata[x]);  
}  
  
//in main  
int nivmax=0;  
niv1[radacina()]=0;  
  
for(i=1;i<=n;i++) {  
    niv1[i]=drumNiv(i);  
    if (niv1[i]>nivmax)  
        nivmax=niv1[i];  
}
```

4) Înăltimea, niveluri



Varianta 1: Complexitate

4) Înăltimea, niveluri



Varianta 1:

- $O(n^2)$



Cum evităm parcurgerea de mai multe ori a unui lanț (și trecerea de mai multe ori pe la același vârf)?

4) Înăltimea, niveluri



Varianta 1 - Îmbunătățiri

- putem memora nivelurile deja calculate ale vârfurilor într-un vector, pentru a evita trecerea de mai multe ori prin același vârf și recalcularea nivelului acestuia

```
int niv1[100]={0};  
  
//functie care returneaza nivelul lui x  
int drumNiv(int x){  
    if(tata[x]==0) //radacina  
        return 0;  
    if(niv1[x]==0)//necalculat  
        niv1[x]= 1+drumNiv(tata[x]);  
    return niv1[x];  
}
```

```
int niv1[100]={0};  
  
//functie care returneaza nivelul lui x  
int drumNiv(int x){  
    if(tata[x]==0) //radacina  
        return 0;  
    if(niv1[x]==0)//necalculat  
        niv1[x]= 1+drumNiv(tata[x]);  
    return niv1[x];  
}
```

```
//in main  
int nivmax=0;  
niv1[radacina()]=0;  
  
for(i=1;i<=n;i++){  
    niv1[i]=drumNiv(i);  
    if (niv1[i]>nivmax)  
        nivmax=niv1[i];  
}
```

4) Înăltimea, niveluri



Varianta 1 îmbunătățită:

- $O(n)$

4) Înălțimea, niveluri



Varianta 2: Parcurgem arborele de la rădăcină

- ▶ Similar parcurgerii grafurilor
- ▶ Parcurgere în lățime -> parcurgere pe niveluri a arborelui
- ▶ Este mai eficient în acest caz să reprezentăm arborele cu liste de fii

4) Înăltimea, niveluri



Concluzii:

- ▶ Reprezentarea unui arbore cu vectorul tata permite o parcurgere mai ușoară a arborelui de la frunze spre rădăcină
- ▶ Putem parcurge arborele și pornind din rădăcină și folosi parcurgeri generale de grafuri – lățime (pe niveluri) și adâncime – dar atunci este mai eficient să folosim ca reprezentare **liste de fii / liste de adiacență** (pe care le putem obține din vectorul tata)

Vector tata -> liste de fii

```
int l[100][100], n;  
void tata_fii(){  
    int i,j,x,y,g;  
    for(i=1;i<=n;i++)  
        if(tata[i]>0){  
            //muchie (tata[i],i)  
            x=tata[i];  
            y=i;  
            l[x][0]++;  
            l[x][l[x][0]]=y;  
        }  
  
    for(i=1;i<=n;i++){  
        cout<<i<<" : ";  
        g=l[i][0];  
        for(j=1;j<=g;j++)  
            cout<<l[i][j]<<" ";  
        cout<<endl;  
    }  
}
```

```
var l: array[1..100,0..100] of integer;  
var n:integer;  
procedure tata_fii;  
    var i,j,x,y,g:integer;  
begin  
    for i:=1 to n do l[i,0]:=0;  
    for i:=1 to n do  
        if tata[i]<>0 then begin  
            {muchie (tata[i],i)}  
            x:=tata[i];  
            y:=i;  
            l[x,0]:=l[x,0]+1;  
            l[x,l[x][0]]:=y;  
        end;  
    for i:=1 to n do begin  
        write(i,' : ');  
        g:=l[i,0];  
        for j:=1 to g do write(l[i,j],' ');  
        writeln;  
    end;  
end;
```

Temă



Dat un vector tata asociat unui arbore să se determine:

- 1) gradul unui vârf x dat
- 2) toate vârfurile de pe un nivel p dat
- 3) cea mai apropiată frunză de rădăcină și un lanț de la rădăcină la aceasta

Problemă



Dat un vector tata cu elementele aparținând mulțimii $\{0,1,\dots,n\}$, să se determine dacă există un arbore cu vectorul tata asociat egal cu vectorul dat
(admitere 2007)

Problemă



Trebuie ca

- vectorul să aibă un singur element egal cu 0
- din rădăcină să se poată ajunge în orice vârf sau, echivalent, din orice vârf să se poată ajunge în rădăcină mergând din tată în tată

Problemă



Trebuie ca

- vectorul să aibă un singur element egal cu 0
- din rădăcină să se poată ajunge în orice vârf sau, echivalent, din orice vârf să se poată ajunge în rădăcină mergând din tată în tată
- **Idee similară cu cea de la calculul înălțimii:** pornim dintr-un vârf și urcând spre rădăcină marcăm vârfurile prin care trecem

```
int acc[100]={0};  
int viz[100]={0};  
/*acc[i]=1 ⇔ i este accesibil  
din radacina  
viz[i]=1 ⇔ i a fost vizitat*/  
  
int accesibil(int x){  
    if(tata[x]==0) return 1;  
    if(viz[x]==0){  
        viz[x]=1;  
        acc[x]=accesibil(tata[x]);  
    }  
    return acc[x];  
}
```

```
int validUrcare(){  
    int i;  
    for(i=1;i<=n;i++)  
        {viz[i]=0;acc[i]=0;}  
    for(i=1;i<=n;i++){  
        if(accesibil(i)==0)  
            return 0;  
    }  
    return 1;  
}
```

Problemă



Ca și la calculul înălțimii, sunt posibile și alte abordări:

- ▶ putem verifica și invers, dacă din rădăcină se poate ajunge în orice vârf – parcurgem arborele pornind din rădăcină și verificăm în final dacă au fost vizitate toate vârfurile
- ▶ liste de fii

Problemă



Temă – Dat un vector tata asociat unui arbore și două vârfuri x și y , să se determine cel mai apropiat ascendent (strămoș) comun celor două vârfuri

Convenție: dacă x este ascendentul lui y atunci x este considerat cel mai apropiat ascendent comun al celor două vârfuri

Problemă



Temă – Dat un vector tata asociat unui arbore și două vârfuri x și y , să se determine cel mai apropiat ascendent (strămoș) comun celor două vârfuri

Idee:

- Marcăm x și ascendenții lui x
- Urcăm din y spre rădăcină până la primul vârf marcat

Parcurgerea grafurilor



Parcurgerea grafurilor

Parcurgere = o modalitate prin care, plecând de la un vârf de start și mergând pe arce/muchii să ajungem la toate vârfurile accesibile din s

Un vârf v este **accesibil** din s dacă există un drum/lanț de la s la v în G .

Parcurgerea grafurilor



Idee: Dacă

- u este **accesibil** din s
- $uv \in E(G)$

atunci v este accesibil din s .
("din aproape în aproape")

Parcurgerea grafurilor

- ▶ Parcurgerea în lățime (BF = breadth first)
- ▶ Parcurgerea în adâncime (DF = depth first)

Parcurgerea în lățime

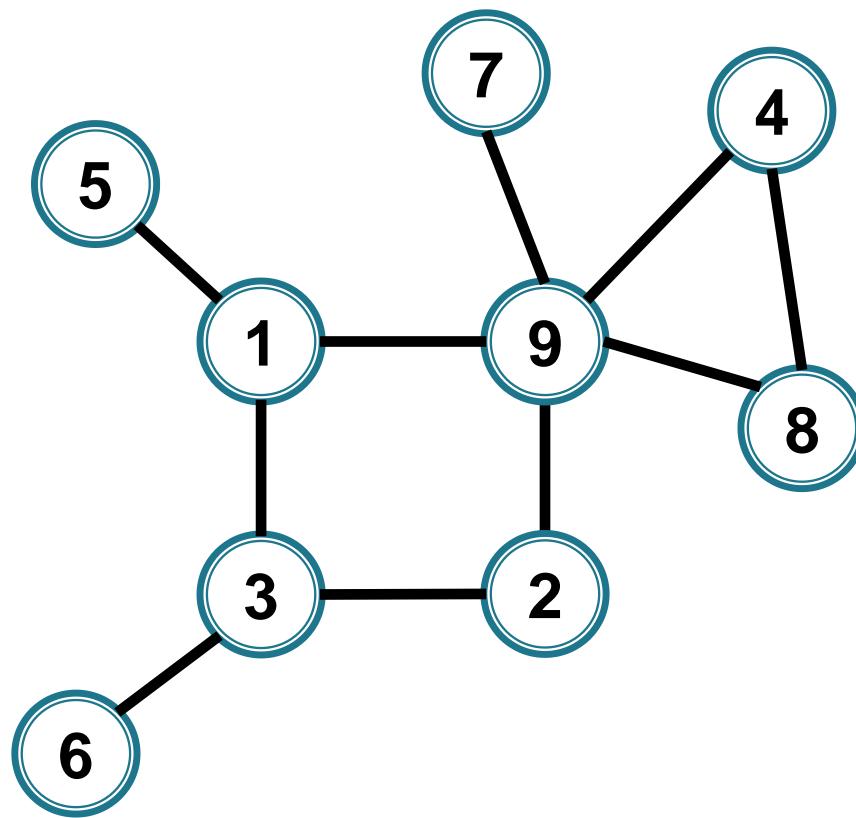


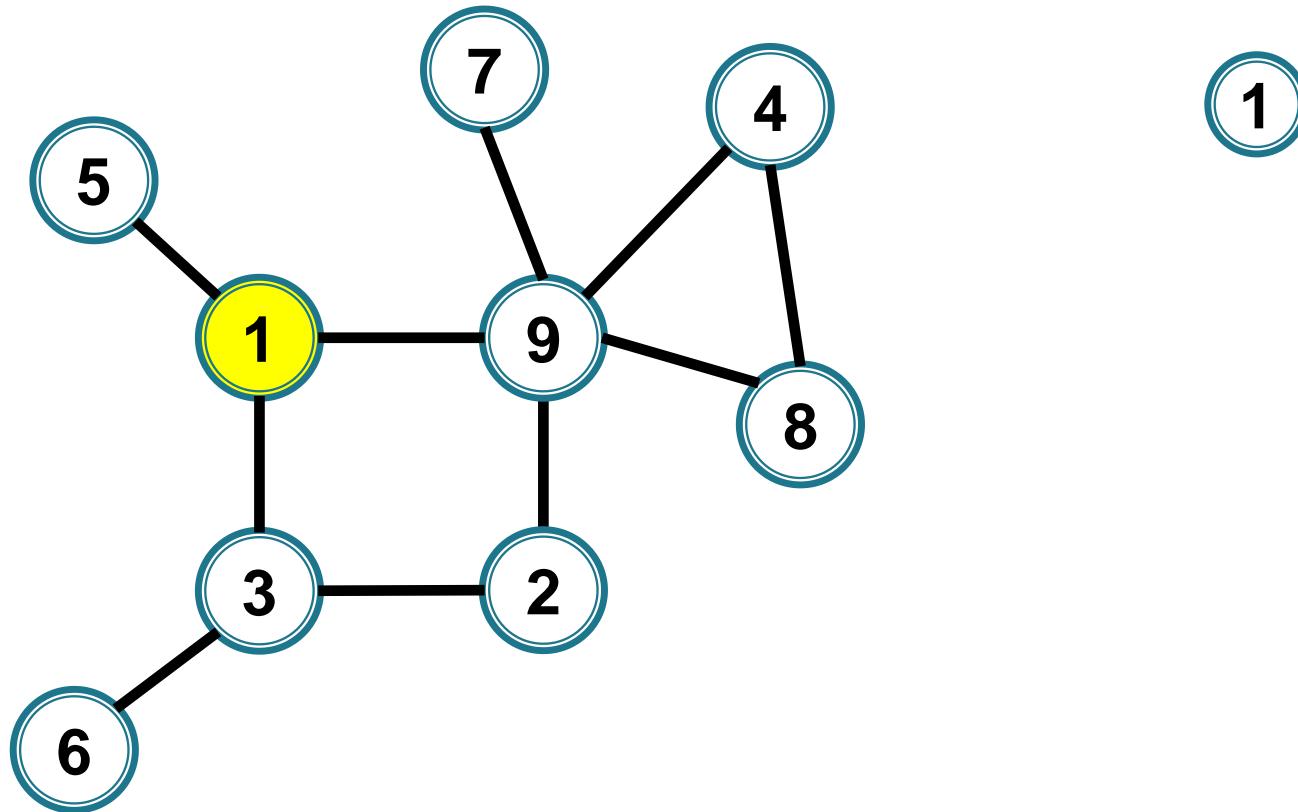
Parcurgerea grafurilor

► **Parcurgerea în lățime:** se vizitează

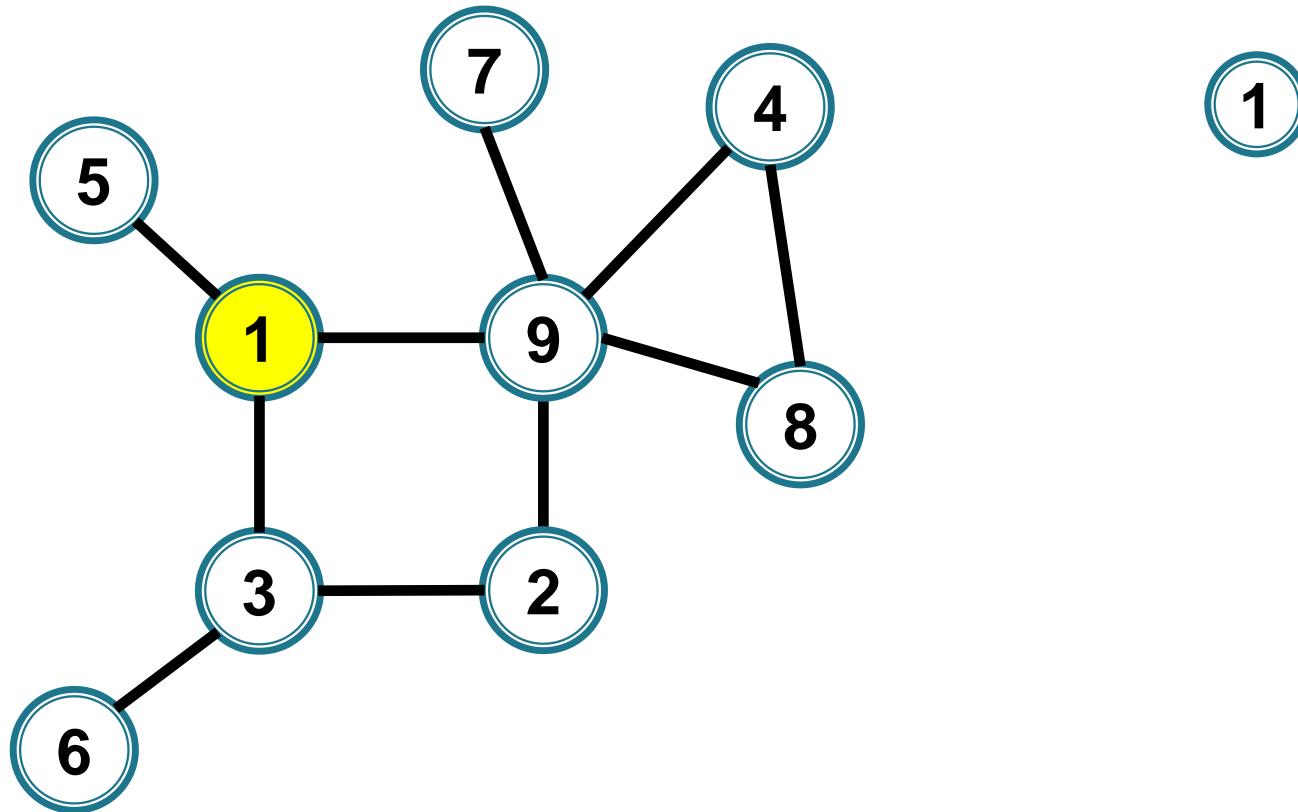
- vârful de start și
- vecinii acestuia
- vecinii nevizitați ai acestora

etc

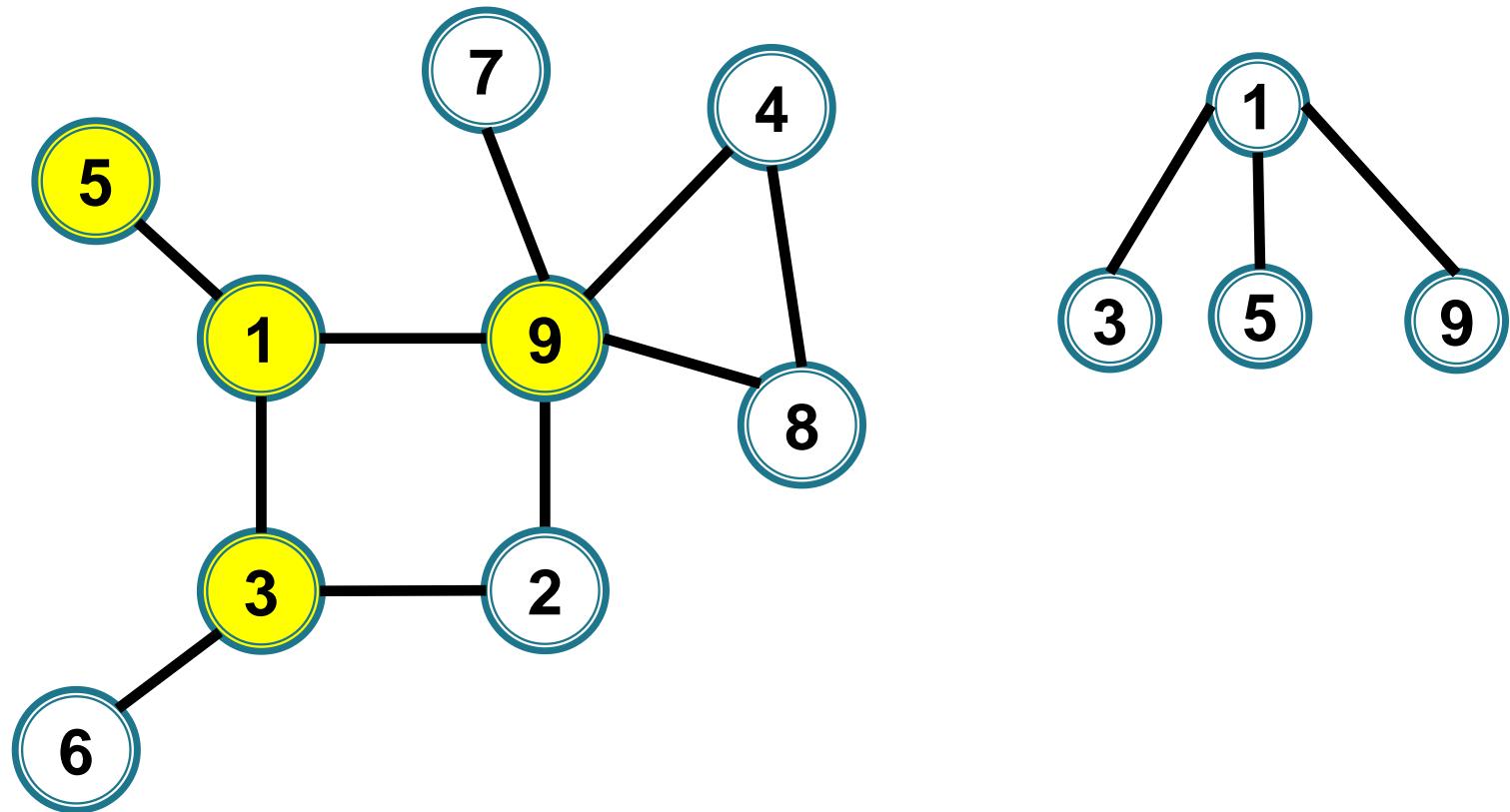




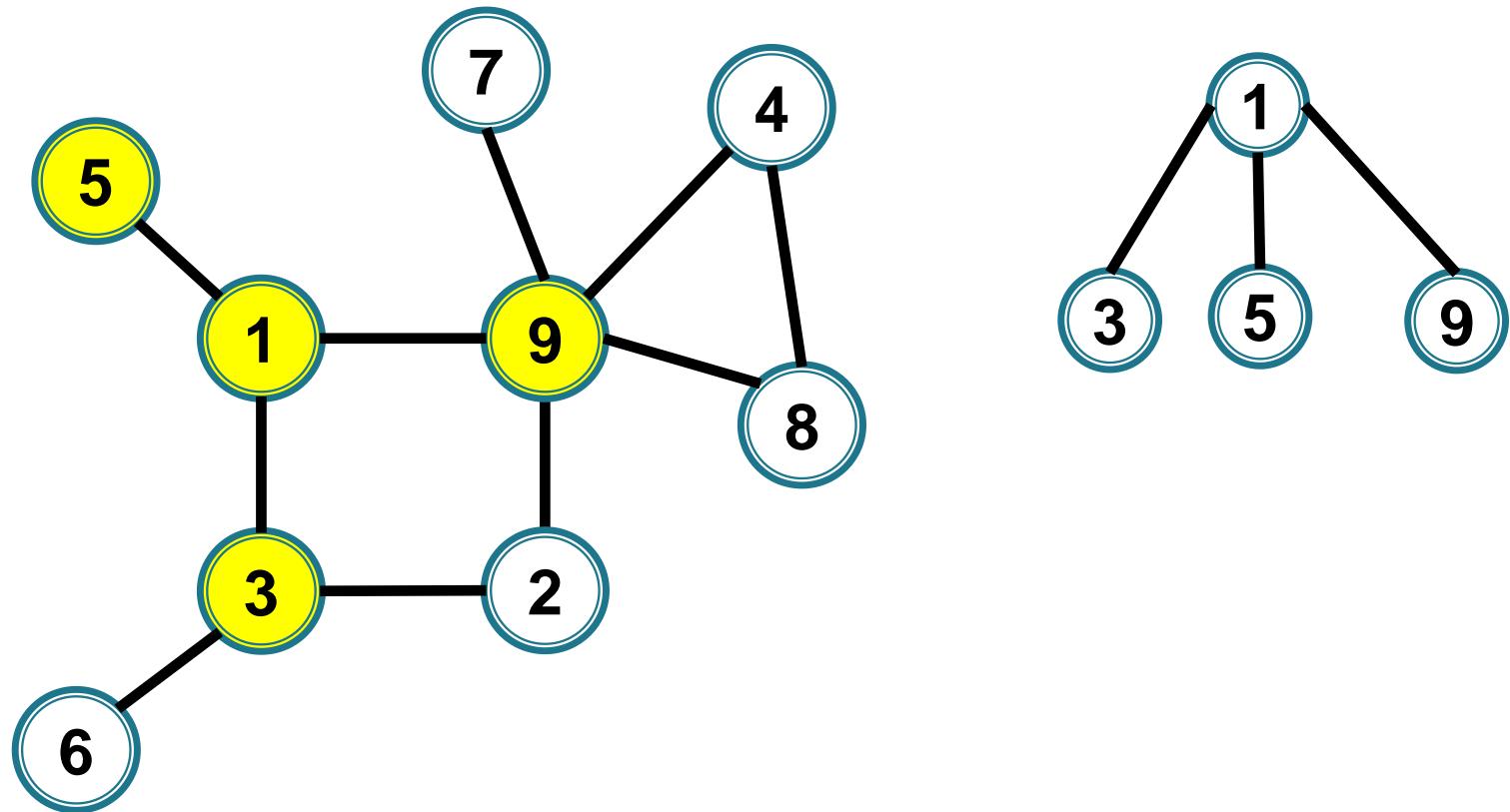
1



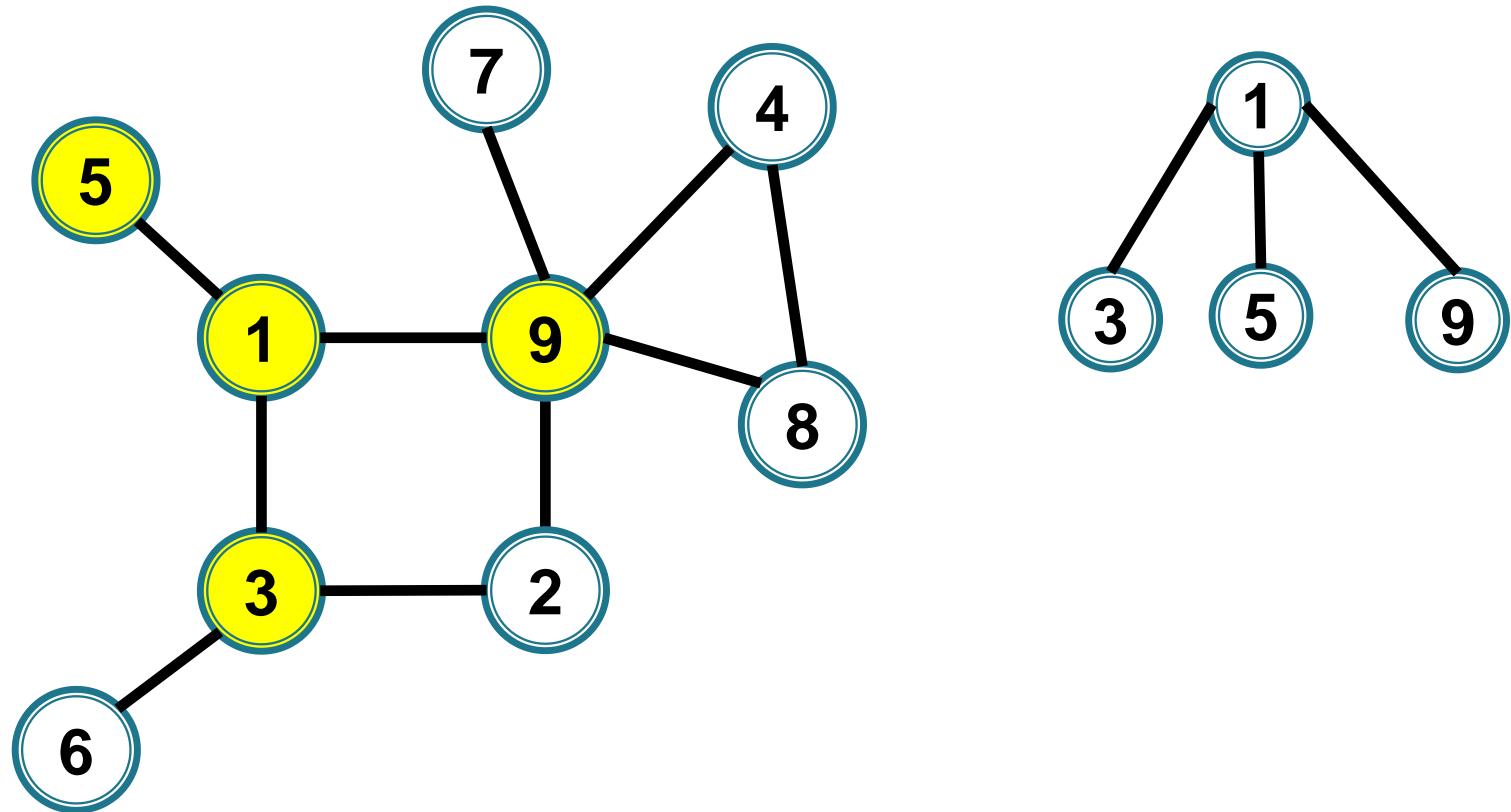
1



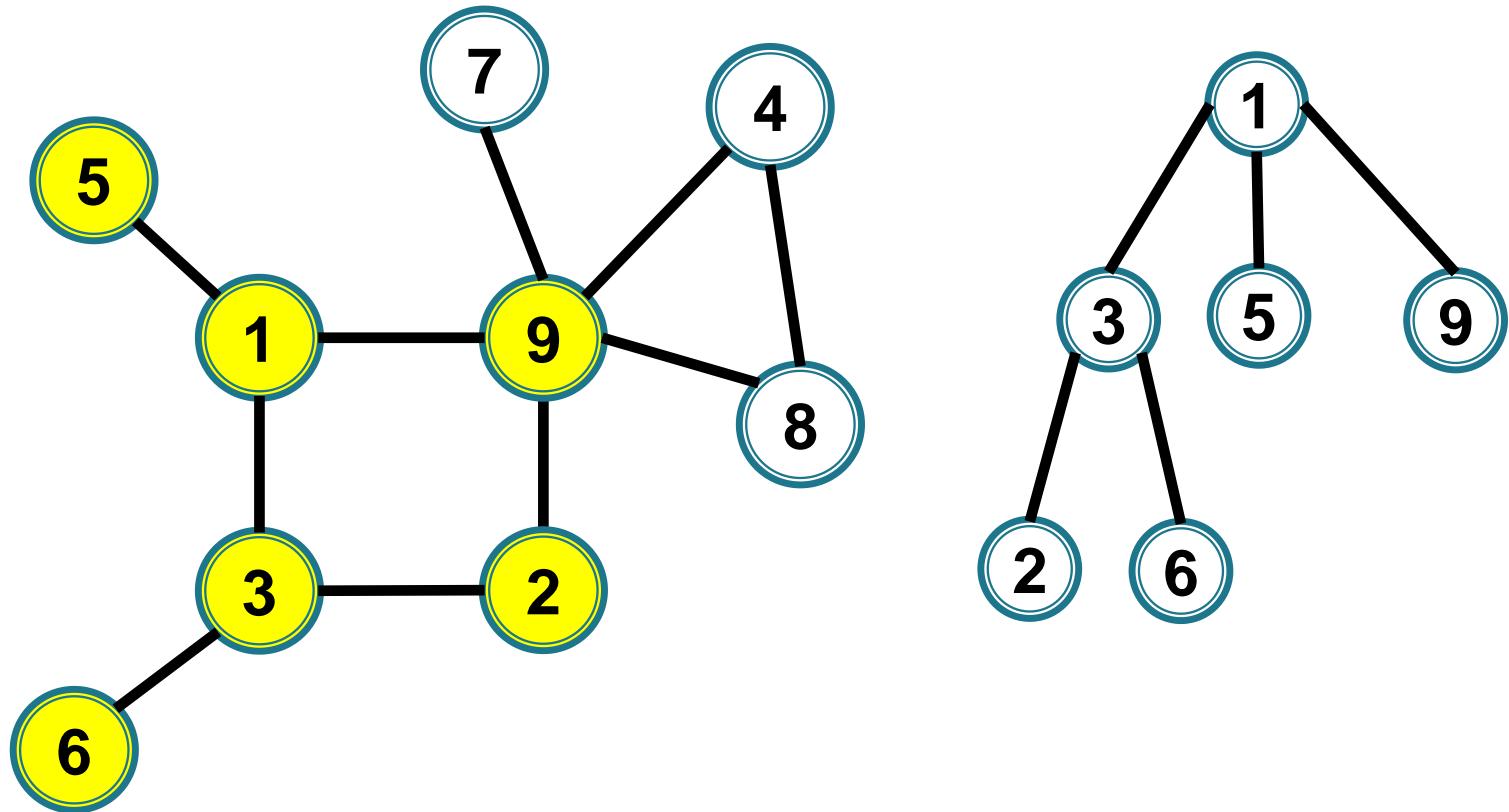
1 3 5 9



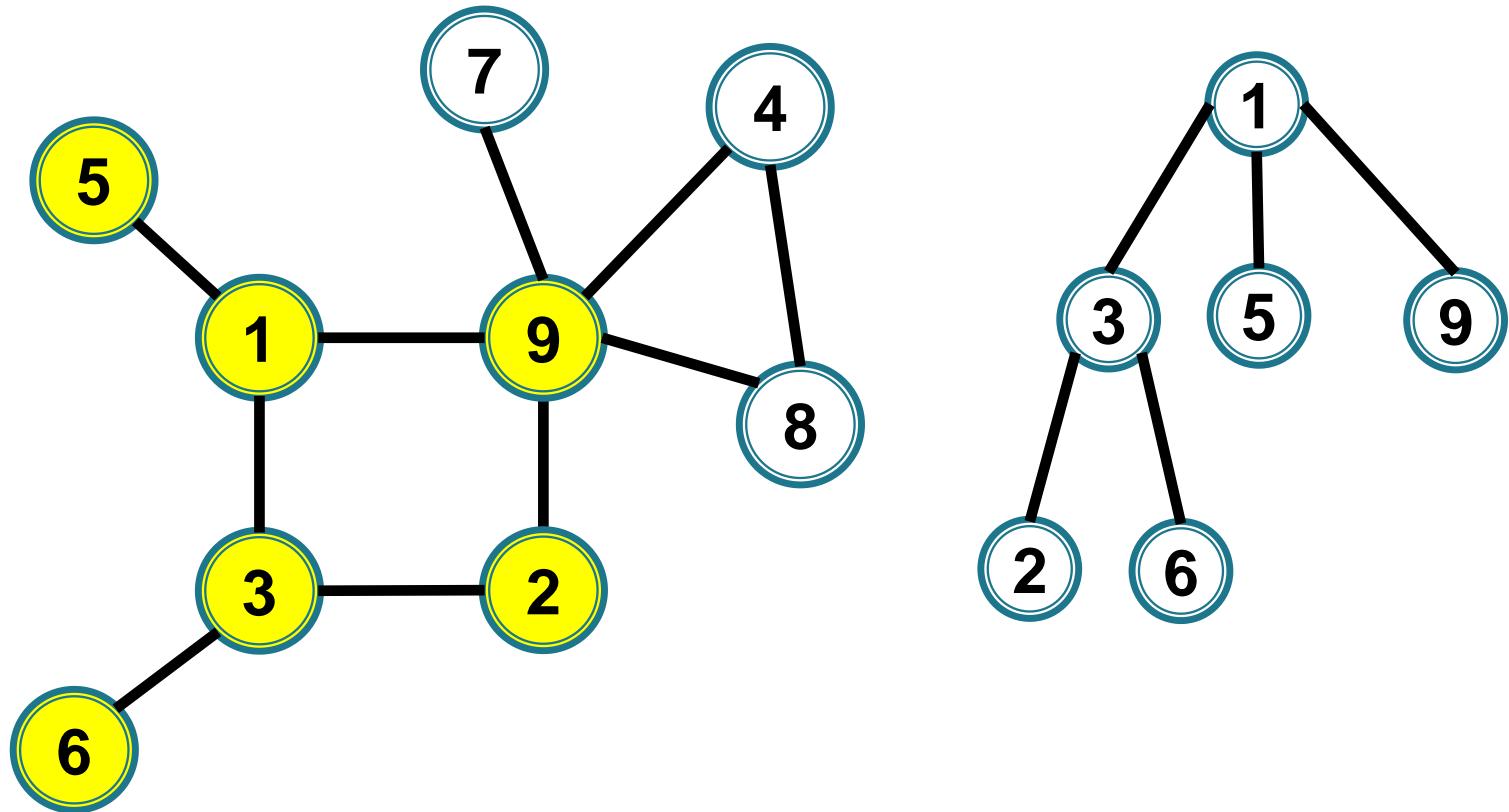
1 3 5 9



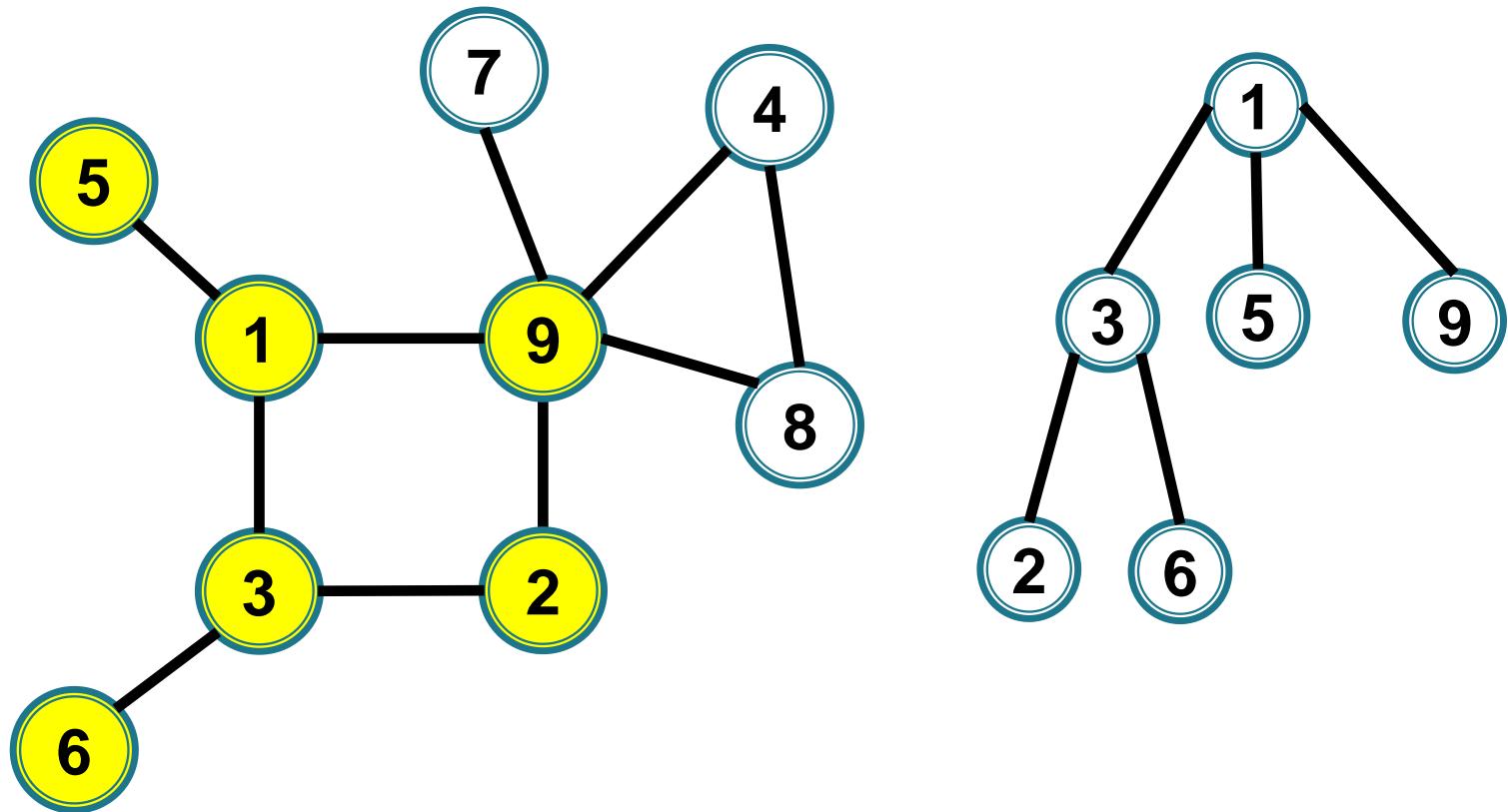
1 3 5 9



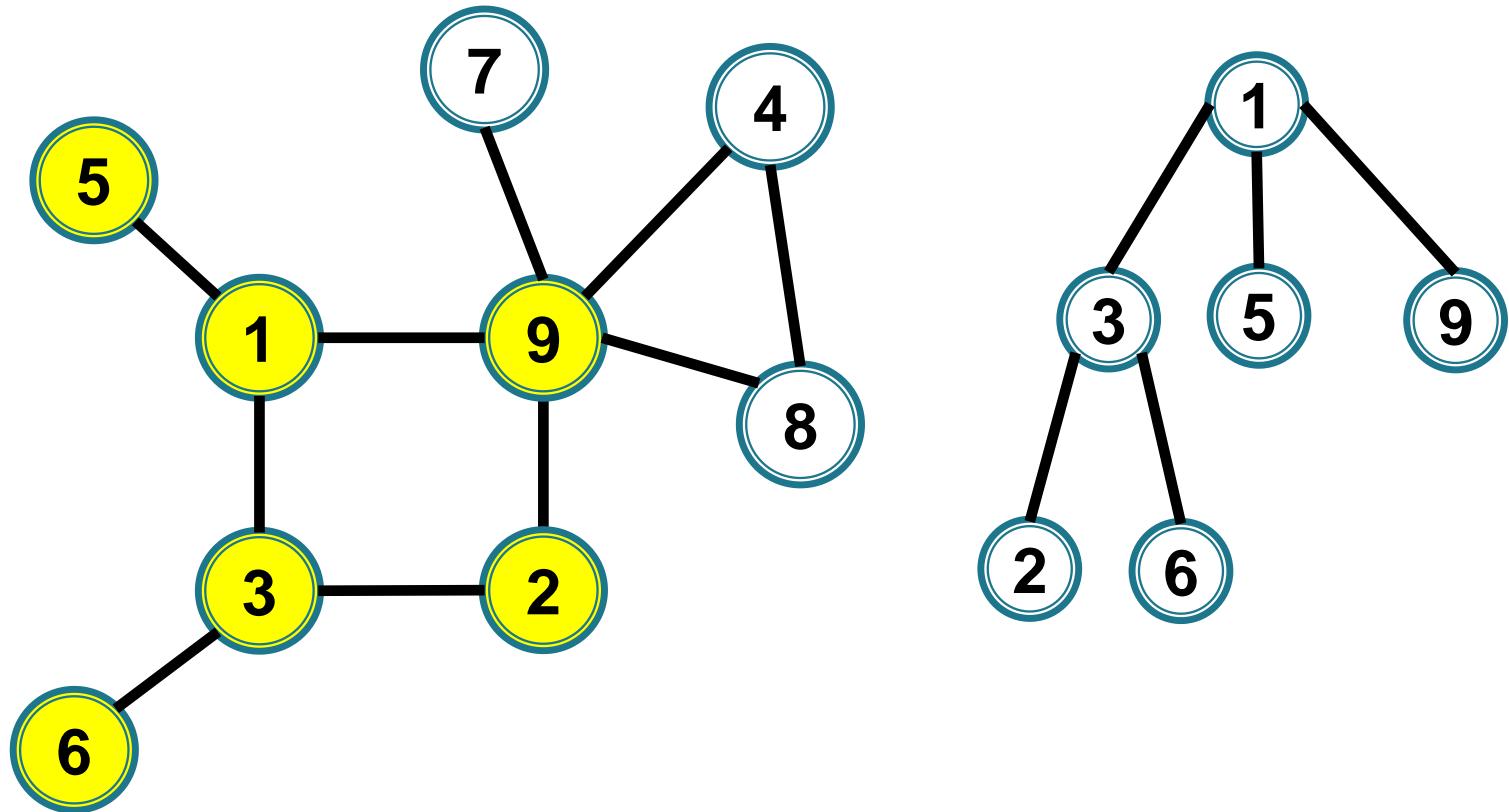
1 3 5 9 2 6



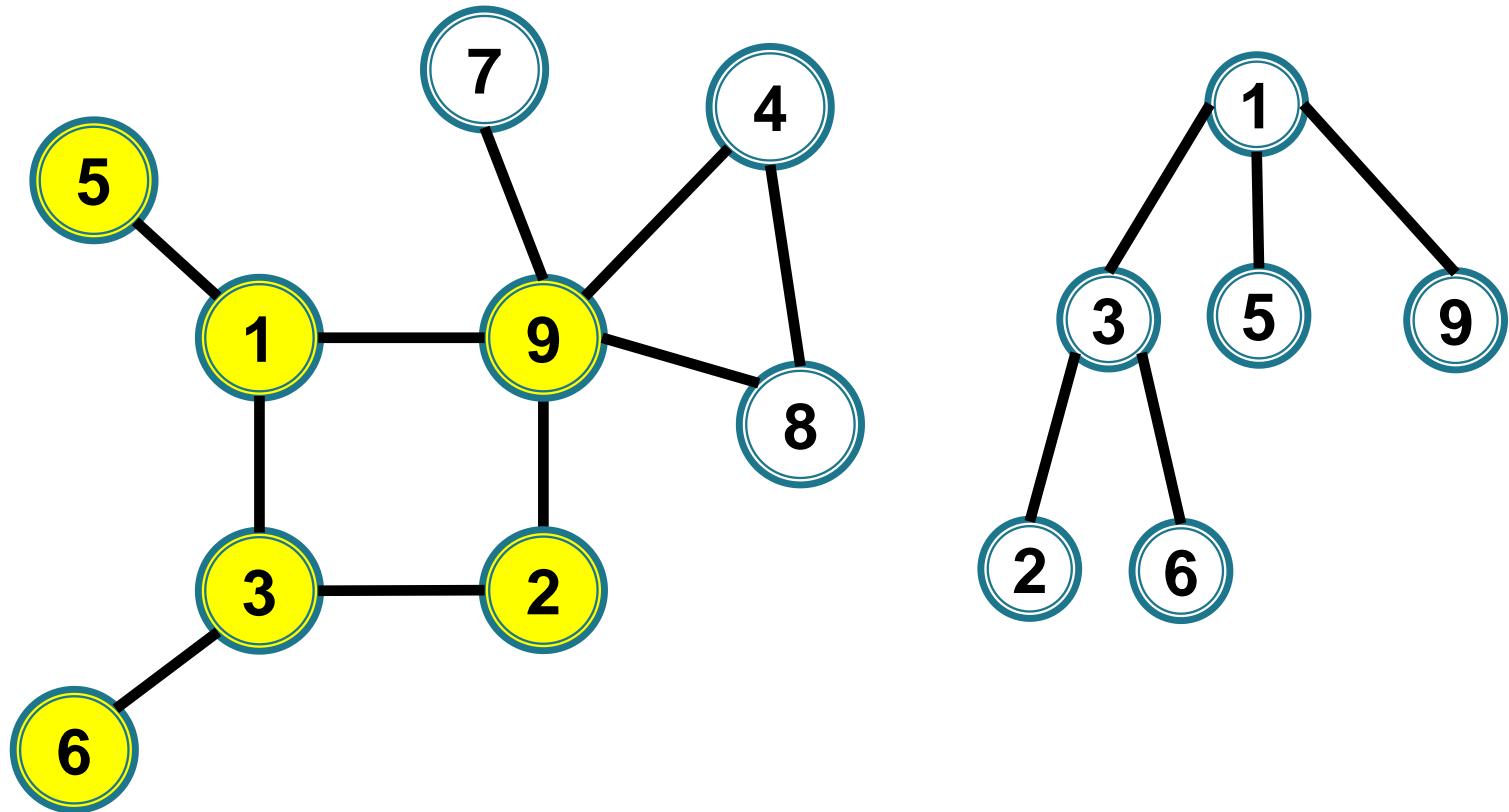
1 3 5 9 2 6



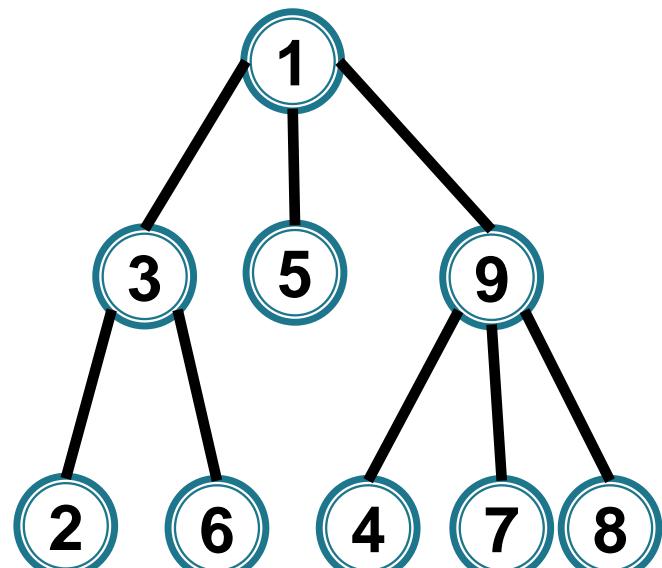
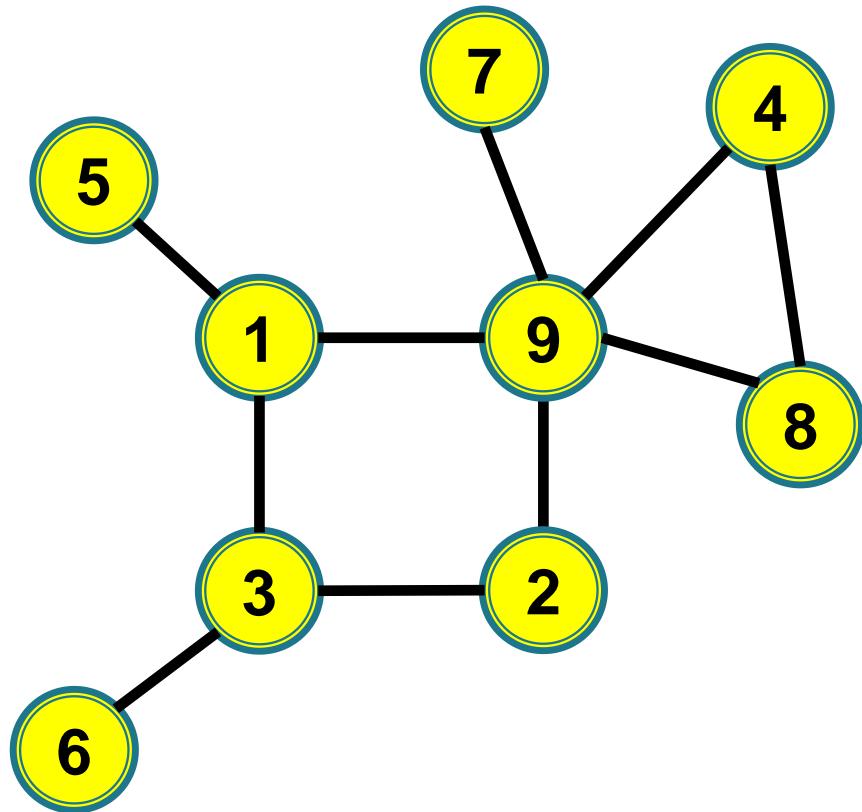
1 3 5 9 2 6



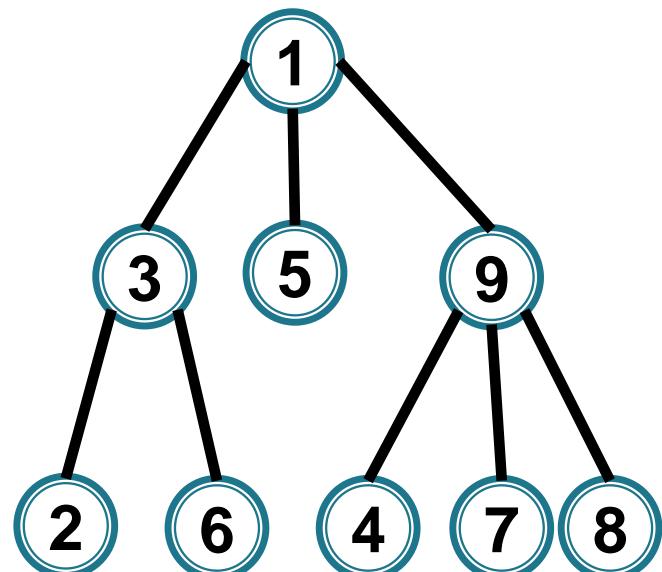
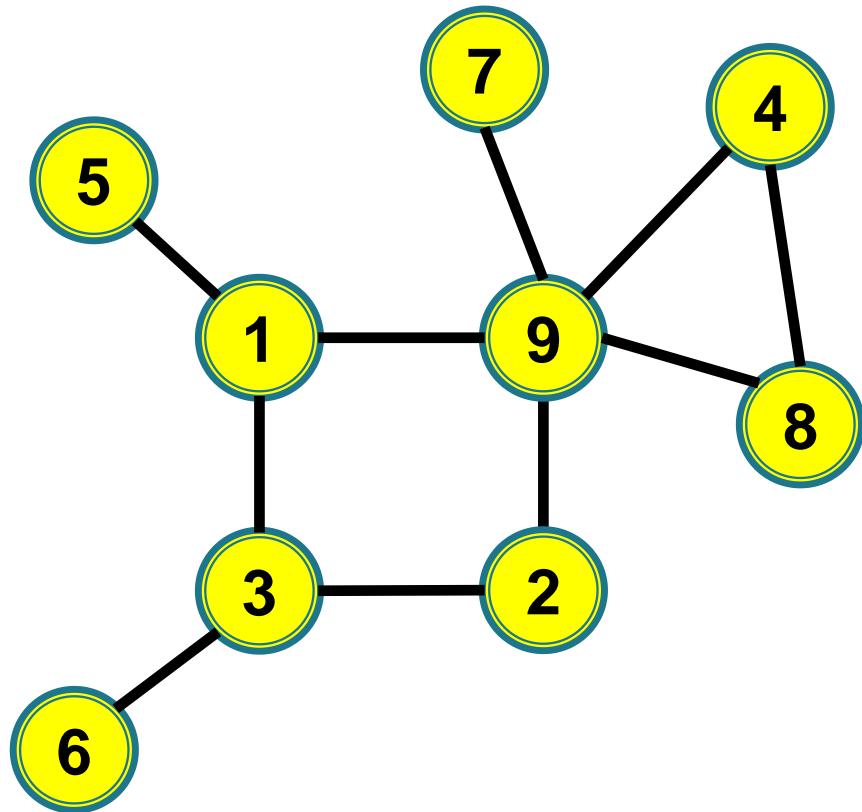
1 3 5 9 2 6



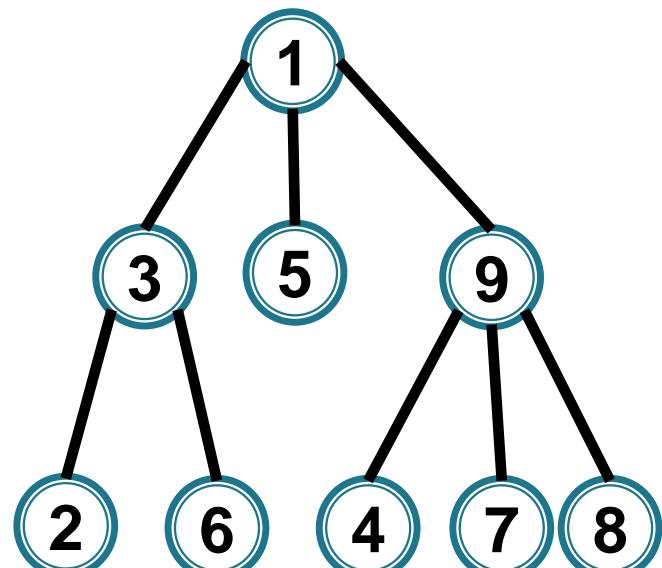
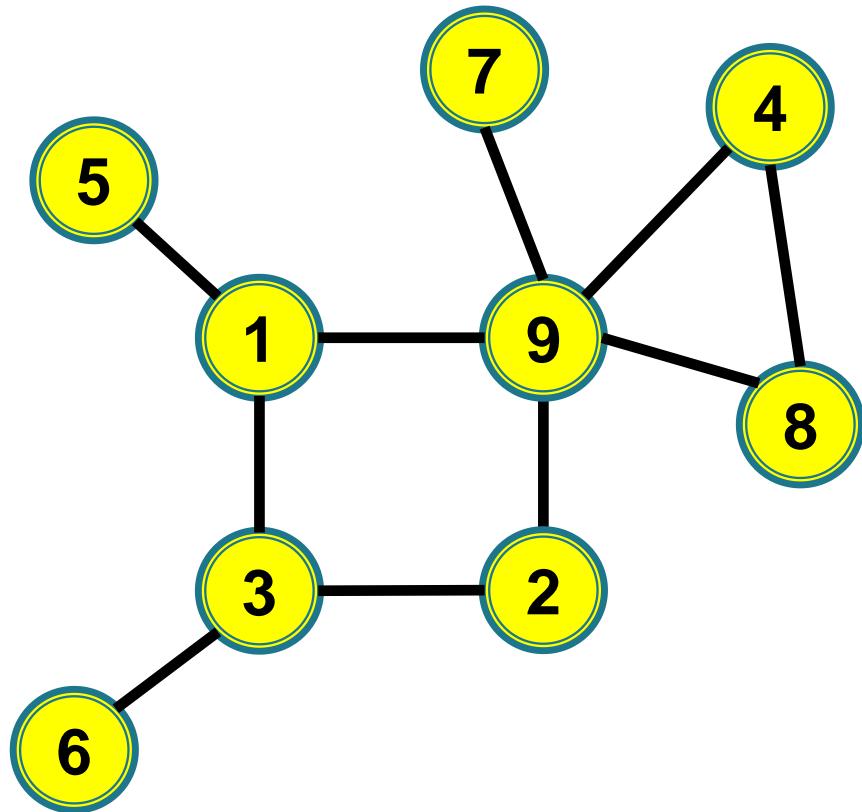
1 3 5 9 2 6



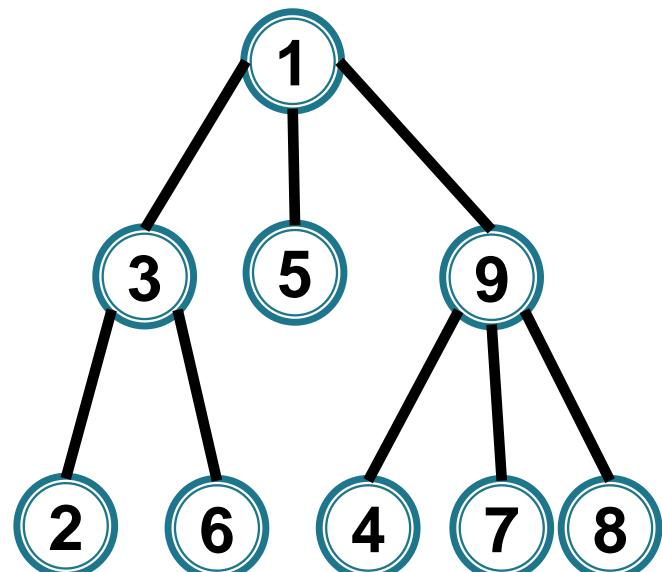
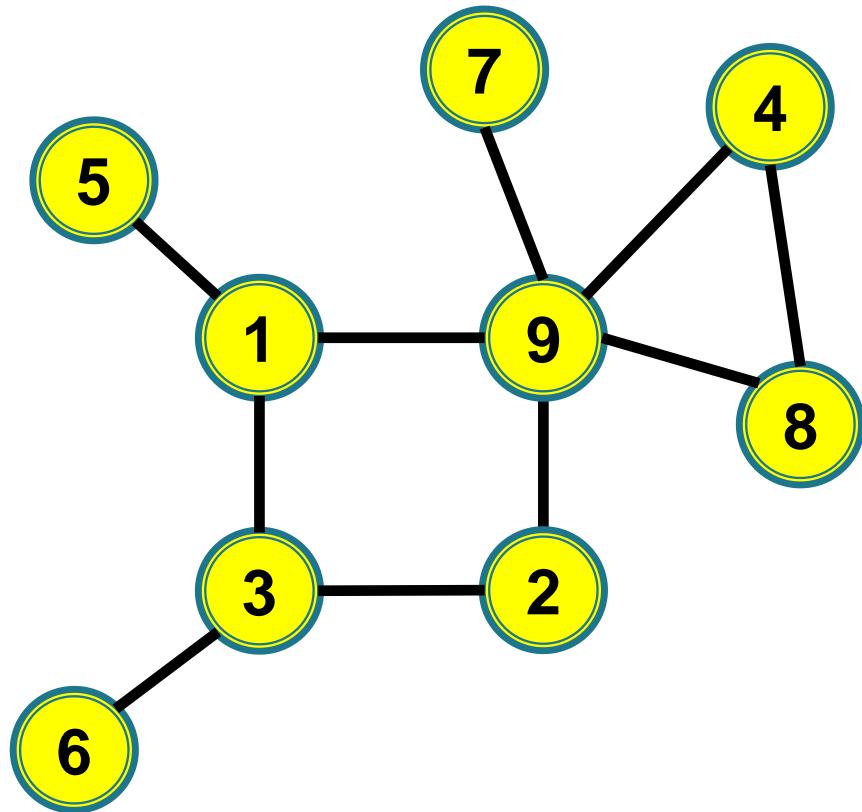
1 3 5 9 2 6 4 7 8



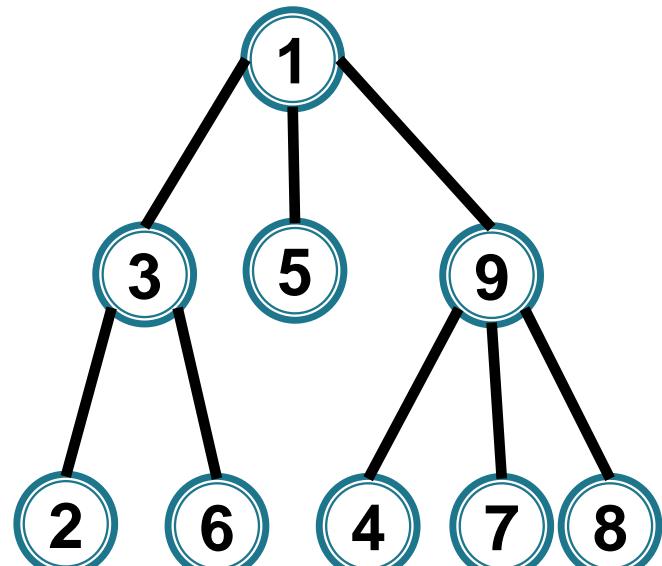
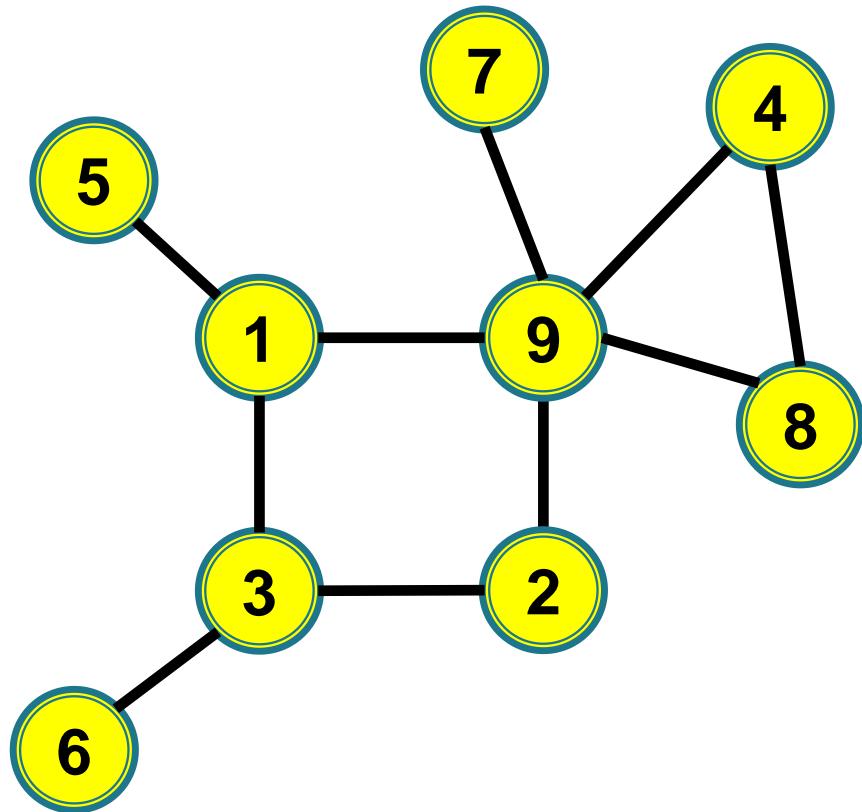
1 3 5 9 2 6 4 7 8



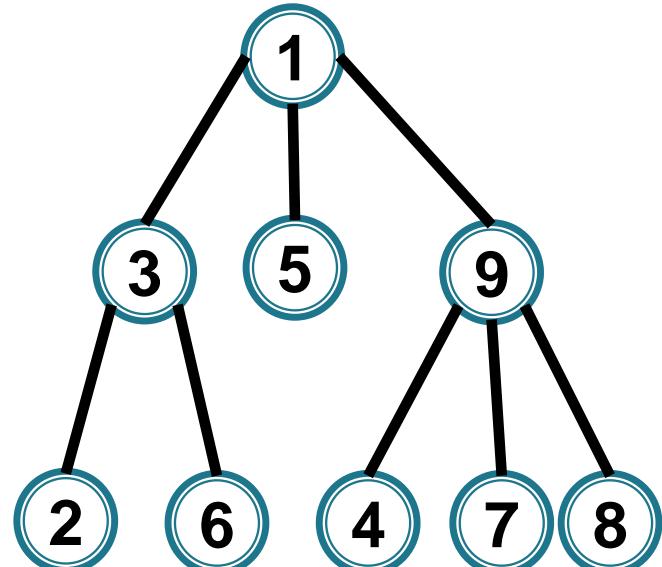
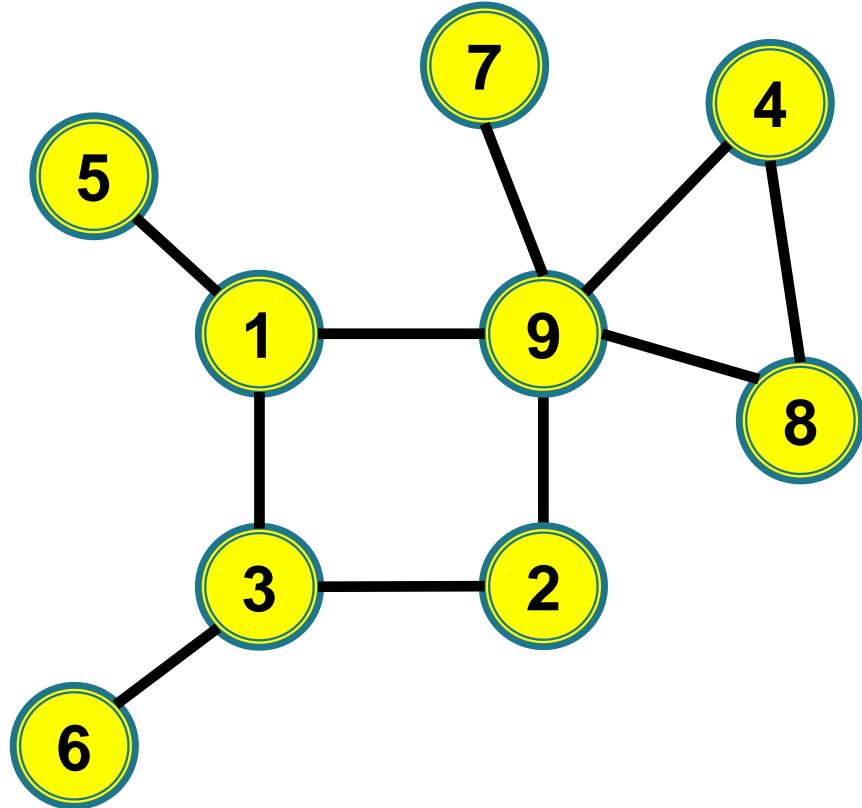
1 3 5 9 2 6 4 7 8



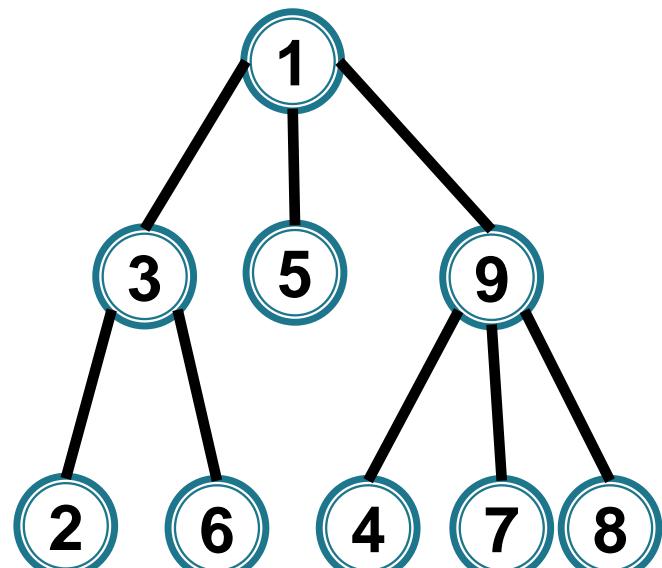
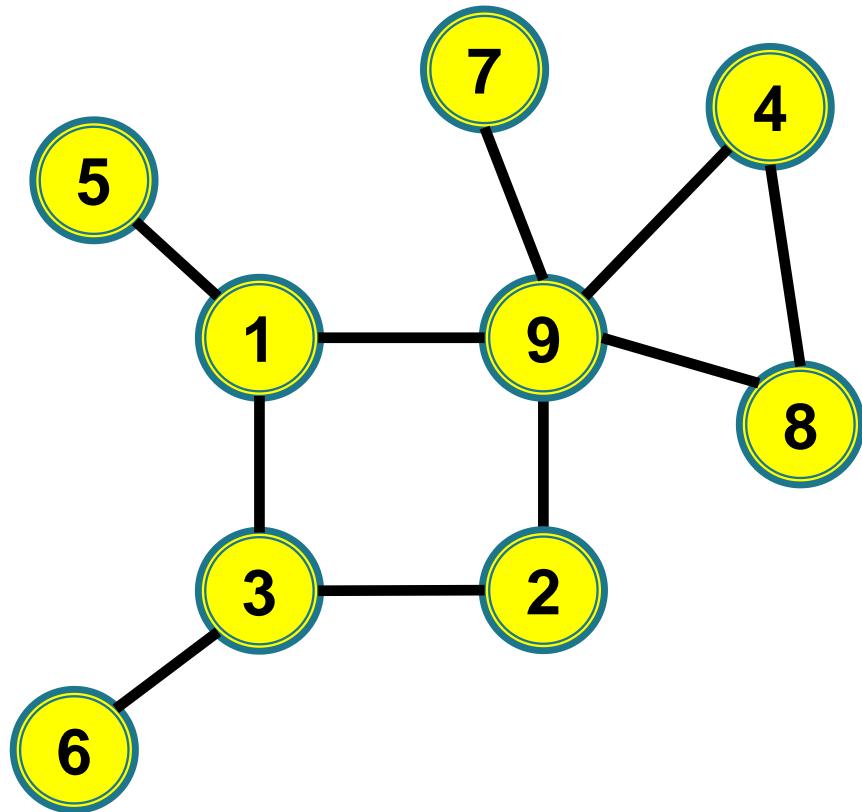
1 3 5 9 2 6 4 7 8



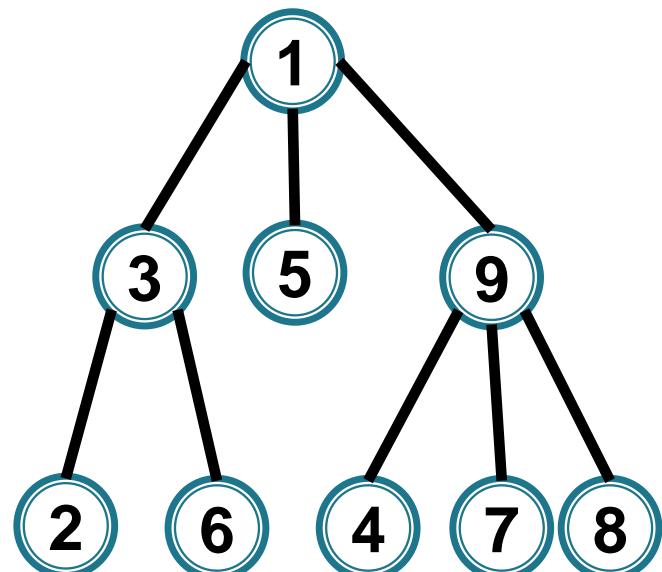
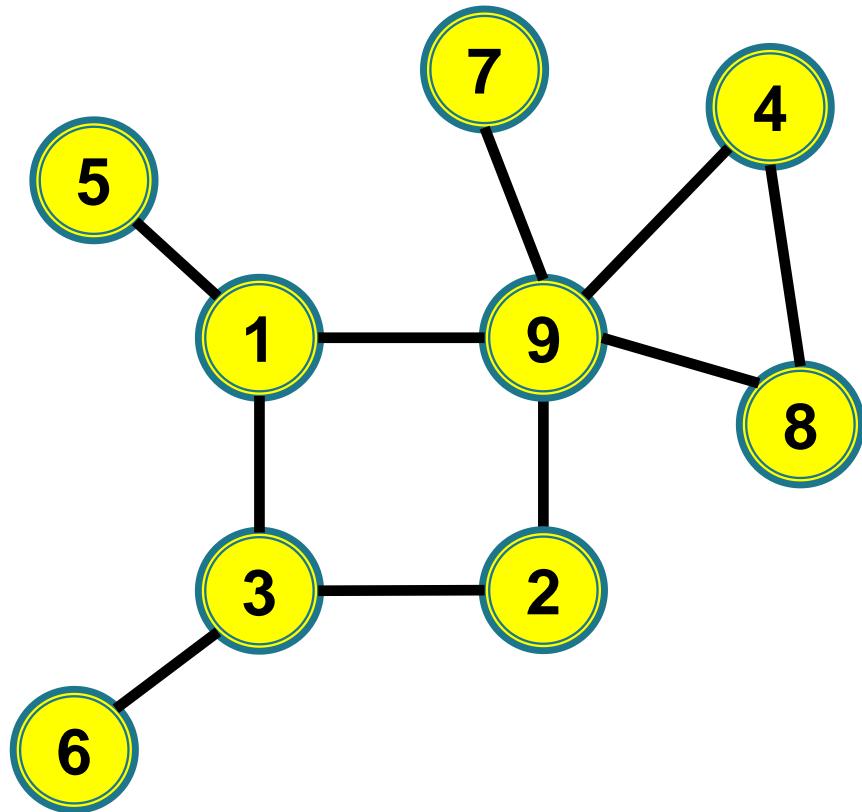
1 3 5 9 2 6 4 7 8



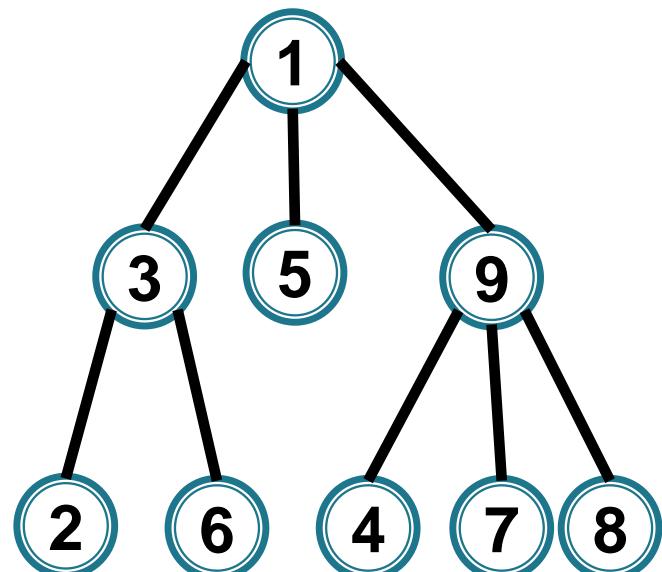
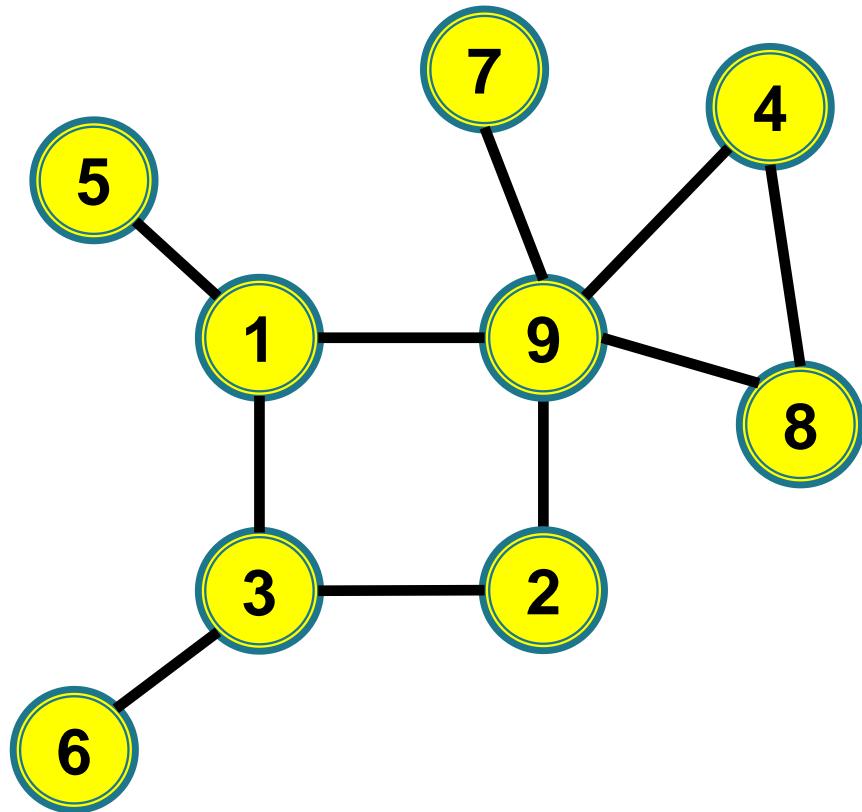
1 3 5 9 2 6 4 7 8



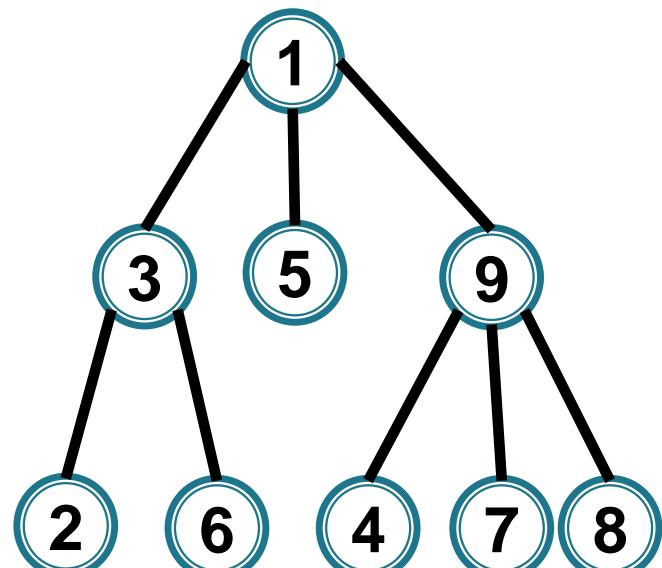
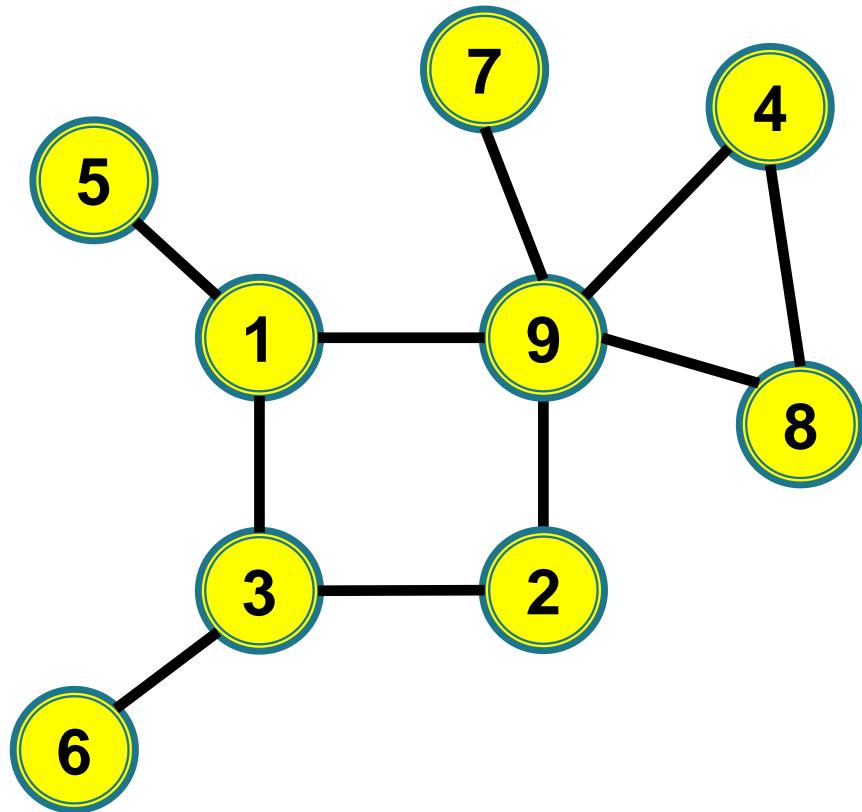
1 3 5 9 2 6 4 7 8



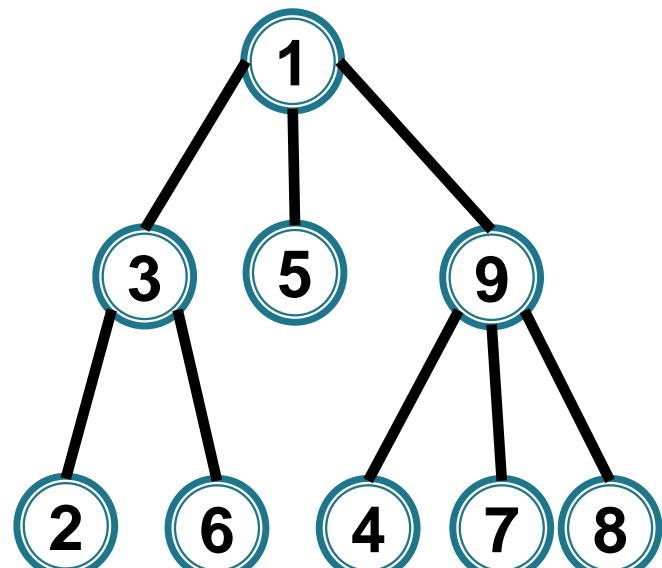
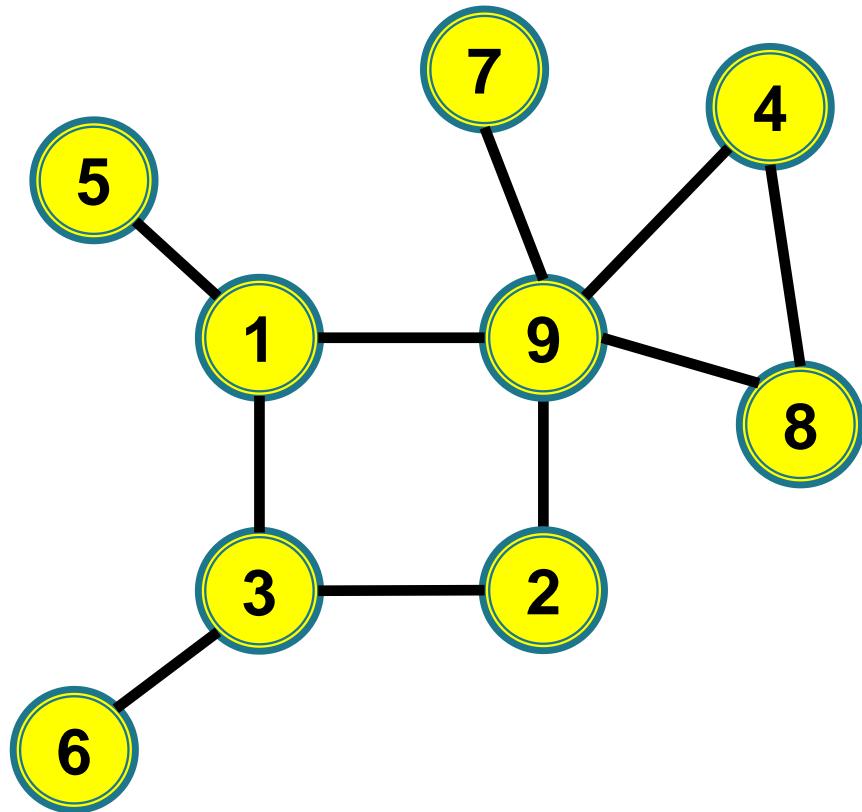
1 3 5 9 2 6 4 7 8



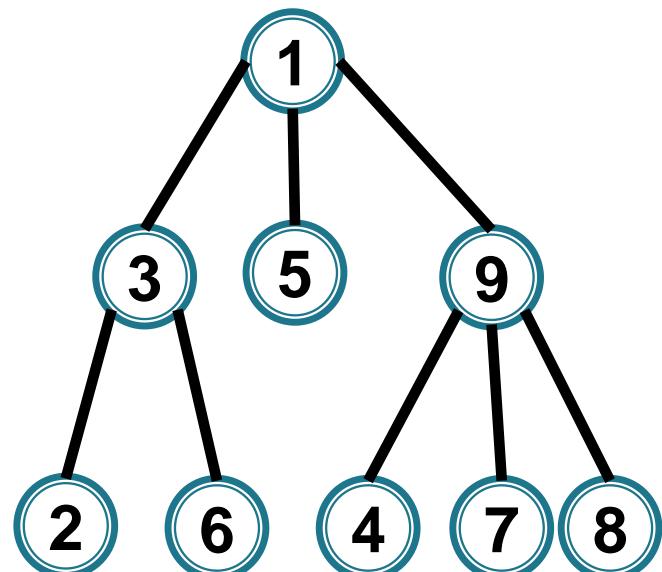
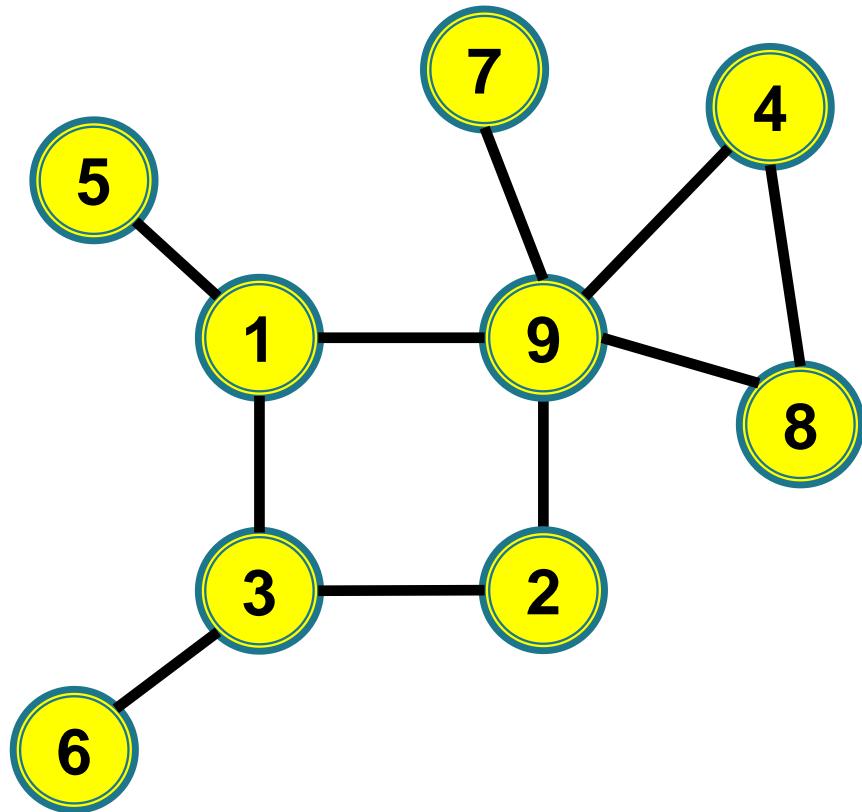
1 3 5 9 2 6 4 7 8



1 3 5 9 2 6 4 7 8



1 3 5 9 2 6 4 7 8



1 3 5 9 2 6 4 7 8

Pseudocod. Implementare

Parcurgerea în lățime

- ▶ Vârfurile vizitate trebuie marcate:

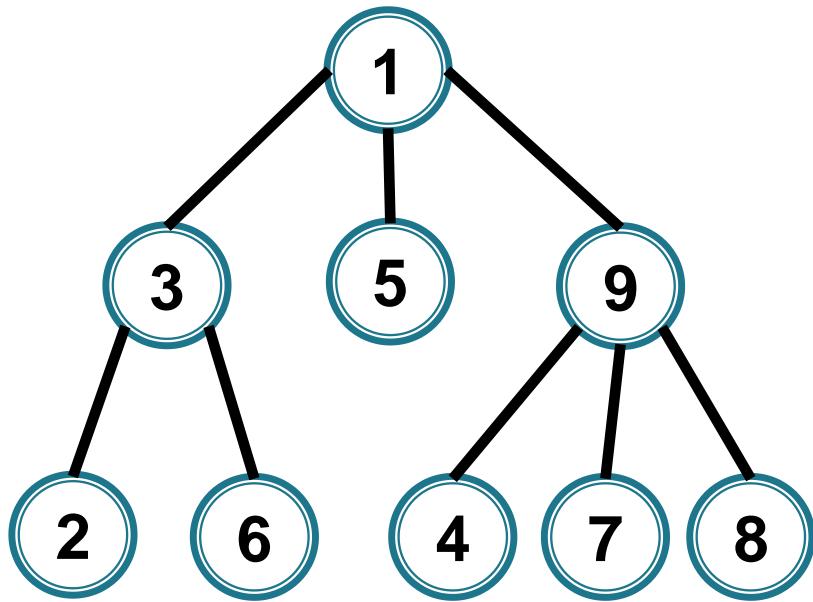
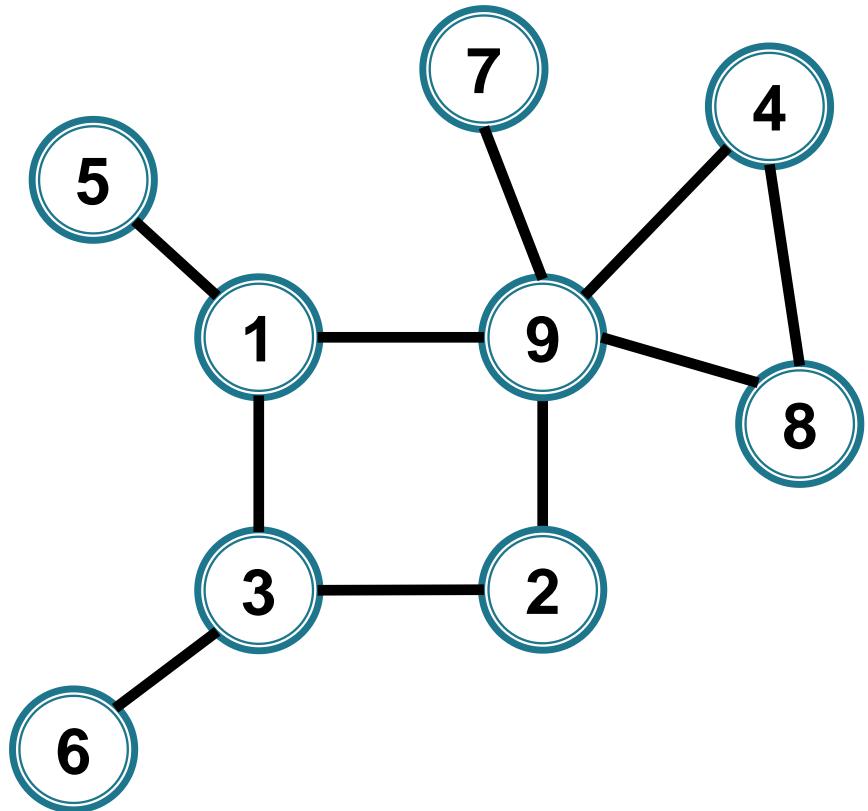
$$\text{viz}[i] = \begin{cases} 1, & \text{dacă } i \text{ a fost vizitat} \\ 0, & \text{altfel} \end{cases}$$

Parcurgerea în lățime

- Dacă dorim și determinarea de **lanțuri** (!!minime) de la rădăcină la alte vârfuri putem reține în plus:

tata[j] = acel vârf i din care este descoperit
(vizitat) j

niv[i] = nivelul lui i în arborele asociat
parcurgerii
= distanța de la s la i



Initializări

pentru $i=1, n$ executa

$viz[i] \leftarrow 0$

$tata[i] \leftarrow 0$

$niv[i] \leftarrow \infty$

procedure BF(s)

coada C $\leftarrow \emptyset$;

```
procedure BF(s)
```

```
    coada C  $\leftarrow \emptyset$ ;
```

```
    adauga(s, C)
```

```
    viz[s]  $\leftarrow 1$ ; niv[s]  $\leftarrow 0$ 
```

```
procedure BF(s)
    coada C ← Ø;
    adauga(s, C)
    viz[s] ← 1; niv[s] ← 0
    cat timp C ≠ Ø executa
```

```
procedure BF(s)
    coada C ← Ø;
    adauga(s, C)
    viz[s] ← 1; niv[s] ← 0
    cat timp C ≠ Ø executa
        i ← extrage(C);
        afiseaza(i);
```

```
procedure BF(s)
    coada C ← Ø;
    adauga(s, C)
    viz[s] ← 1; niv[s] ← 0
    cat timp C ≠ Ø executa
        i ← extrage(C);
        afiseaza(i);

        pentru j vecin al lui i
```

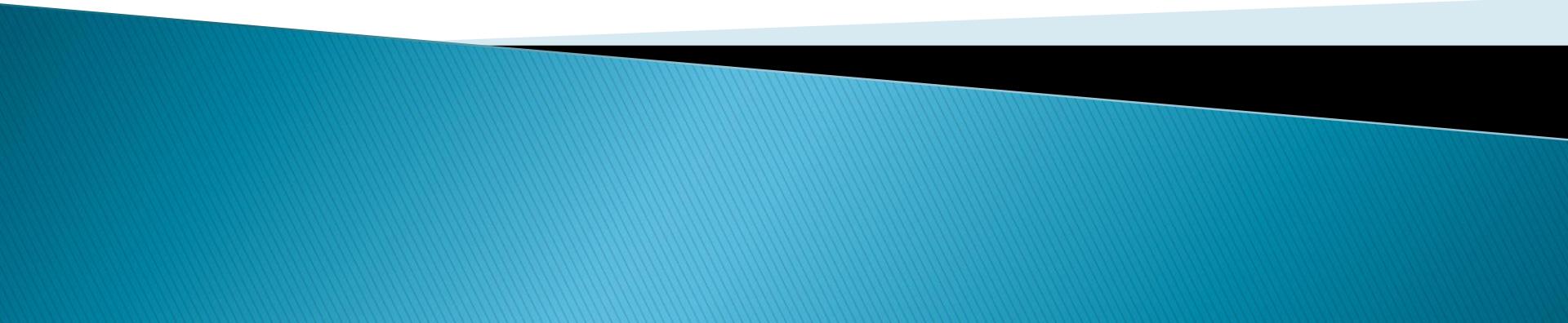
```
procedure BF(s)
    coada C ← Ø;
    adauga(s, C)
    viz[s] ← 1; niv[s] ← 0
    cat timp C ≠ Ø executa
        i ← extrage(C);
        afiseaza(i);

        pentru j vecin al lui i
            daca viz[j]=0 atunci
                adauga(j, C)
                viz[j] ← 1
```

```
procedure BF(s)
    coada C ← Ø;
    adauga(s, C)
    viz[s] ← 1; niv[s] ← 0
    cat timp C ≠ Ø executa
        i ← extrage(C);
        afiseaza(i);

        pentru j vecin al lui i
            daca viz[j]=0 atunci
                adauga(j, C)
                viz[j] ← 1
                tata[j] ← i
                niv[j] ←niv[i]+1
```

Implementare



Initializări

pentru $i=1, n$ executa

$viz[i] \leftarrow 0$

$tata[i] \leftarrow 0$

$niv[i] \leftarrow \infty$

```
int n;  
int a[20][20];  
int viz[20], tata[20], niv[20];  
int p, u, c[20];
```

Initializări

pentru $i=1, n$ executa
 $viz[i] \leftarrow 0$
 $tata[i] \leftarrow 0$
 $niv[i] \leftarrow \infty$

```
int n;  
int a[20][20];  
int viz[20], tata[20], niv[20];  
int p, u, c[20];  
  
for(i=1; i<=n; i++) {  
    viz[i]=0;  
    tata[i]=0;  
    niv[i]=n;  
}
```

```
procedure BF(s)
```

```
    coada C ← Ø;
```

```
    adauga(s, C)
```

```
    viz[s]← 1; niv[s] ← 0
```

```
    cat timp C ≠ Ø executa
```

```
        i ← extrage(C);
```

```
        afiseaza(i);
```

```
        pentru j vecin al lui i
```

```
            daca viz[j]=0 atunci
```

```
                adauga(j, C)
```

```
                viz[j] ← 1
```

```
                tata[j] ← i
```

```
                niv[j] ←niv[i]+1
```

```
void bf(int s) {
```

```
    int p,u,i,j;
```

```
procedure BF(s)
    coada C ← Ø;
    adauga(s, C)
    viz[s] ← 1; niv[s] ← 0
    cat timp C ≠ Ø executa
        i ← extrage(C);
        afiseaza(i);

        pentru j vecin al lui i
            daca viz[j]=0 atunci
                adauga(j, C)
                viz[j] ← 1
                tata[j] ← i
                niv[j] ←niv[i]+1
```

```
void bf(int s) {
    int p,u,i,j;
    p = u = 1; c[1]=s;
```

```
procedure BF(s)
    coada C ← Ø;
    adauga(s, C)
    viz[s]← 1; niv[s] ← 0
    cat timp C ≠ Ø executa
        i ← extrage(C);
        afiseaza(i);

        pentru j vecin al lui i
            daca viz[j]=0 atunci
                adauga(j, C)
                viz[j] ← 1
                tata[j] ← i
                niv[j] ←niv[i]+1
```

```
void bf(int s) {
    int p,u,i,j;
    p = u = 1; c[1]=s;
    viz[s]=1; niv[s]=0;
```

```
procedure BF(s)
    coada C ← Ø;
    adauga(s, C)
    viz[s]← 1; niv[s] ← 0
    cat timp C ≠ Ø executa
        i ← extrage(C);
        afiseaza(i);

        pentru j vecin al lui i
            daca viz[j]=0 atunci
                adauga(j, C)
                viz[j] ← 1
                tata[j] ← i
                niv[j] ←niv[i]+1
```

```
void bf(int s) {
    int p,u,i,j;
    p = u = 1; c[1]=s;
    viz[s]=1; niv[s]=0;
    while(p<=u) {
```

```

procedure BF(s)
  coada C ← Ø;
  adauga(s, C)
  viz[s]← 1; niv[s] ← 0
  cat timp C ≠ Ø executa
    i ← extrage(C);
    afiseaza(i);

    pentru j vecin al lui i
      daca viz[j]=0 atunci
        adauga(j, C)
        viz[j] ← 1
        tata[j] ← i
        niv[j] ←niv[i]+1

```

```

void bf(int s) {
  int p,u,i,j;
  p = u = 1; c[1]=s;
  viz[s]=1; niv[s]=0;
  while(p<=u) {
    i=c[p]; p=p+1;

```

```

procedure BF(s)
  coada C ← Ø;
  adauga(s, C)
  viz[s]← 1; niv[s] ← 0
  cat timp C ≠ Ø executa
    i ← extrage(C);
    afiseaza(i);

    pentru j vecin al lui i
      daca viz[j]=0 atunci
        adauga(j, C)
        viz[j] ← 1
        tata[j] ← i
        niv[j] ←niv[i]+1

```

```

void bf(int s) {
  int p,u,i,j;
  p = u = 1; c[1]=s;
  viz[s]=1; niv[s]=0;
  while(p<=u) {
    i=c[p]; p=p+1;
    cout<<i<<" ";
  }
}

```

```

procedure BF(s)
  coada C ← Ø;
  adauga(s, C)
  viz[s]← 1; niv[s] ← 0
  cat timp C ≠ Ø executa
    i ← extrage(C);
    afiseaza(i);

    pentru j vecin al lui i
      daca viz[j]=0 atunci
        adauga(j, C)
        viz[j] ← 1
        tata[j] ← i
        niv[j] ←niv[i]+1

```

```

void bf(int s) {
  int p,u,i,j;
  p = u = 1; c[1]=s;
  viz[s]=1; niv[s]=0;
  while(p<=u) {
    i=c[p]; p=p+1;
    cout<<i<<" ";
    for(j=1;j<=n;j++)
      if(a[i][j]==1)

```

```

procedure BF(s)
  coada C ← Ø;
  adauga(s, C)
  viz[s]← 1; niv[s] ← 0
  cat timp C ≠ Ø executa
    i ← extrage(C);
    afiseaza(i);

    pentru j vecin al lui i
      daca viz[j]=0 atunci
        adauga(j, C)
        viz[j] ← 1
        tata[j] ← i
        niv[j] ←niv[i]+1

```

```

void bf(int s) {
  int p,u,i,j;
  p = u = 1; c[1]=s;
  viz[s]=1; niv[s]=0;
  while(p<=u) {
    i=c[p]; p=p+1;
    cout<<i<<" ";
    for(j=1;j<=n;j++)
      if(a[i][j]==1)
        if(viz[j]==0) {
          u=u+1; c[u]=j;
        }
    }
}

```

```

procedure BF(s)
  coada C ← Ø;
  adauga(s, C)
  viz[s]← 1; niv[s] ← 0
  cat timp C ≠ Ø executa
    i ← extrage(C);
    afiseaza(i);

    pentru j vecin al lui i
      daca viz[j]=0 atunci
        adauga(j, C)
        viz[j] ← 1
        tata[j] ← i
        niv[j] ←niv[i]+1

```

```

void bf(int s) {
  int p,u,i,j;
  p = u = 1; c[1]=s;
  viz[s]=1; niv[s]=0;
  while(p<=u) {
    i=c[p]; p=p+1;
    cout<<i<<" ";
    for(j=1;j<=n;j++)
      if(a[i][j]==1)
        if(viz[j]==0) {
          u=u+1; c[u]=j;
          viz[j]=1;
          tata[j]=i;
          niv[j]=niv[i]+1;
        }
  }
}

```

Variantă - cu liste de adiacență



```

procedure BF(s)
  coada C ← Ø;
  adauga(s, C)
  viz[s]← 1; niv[s] ← 0
  cat timp C ≠ Ø executa
    i ← extrage(C);
    afiseaza(i);

    pentru j vecin al lui i
      daca viz[j]=0 atunci
        adauga(j, C)
        viz[j] ← 1
        tata[j] ← i
        niv[j] ←niv[i]+1

```

```

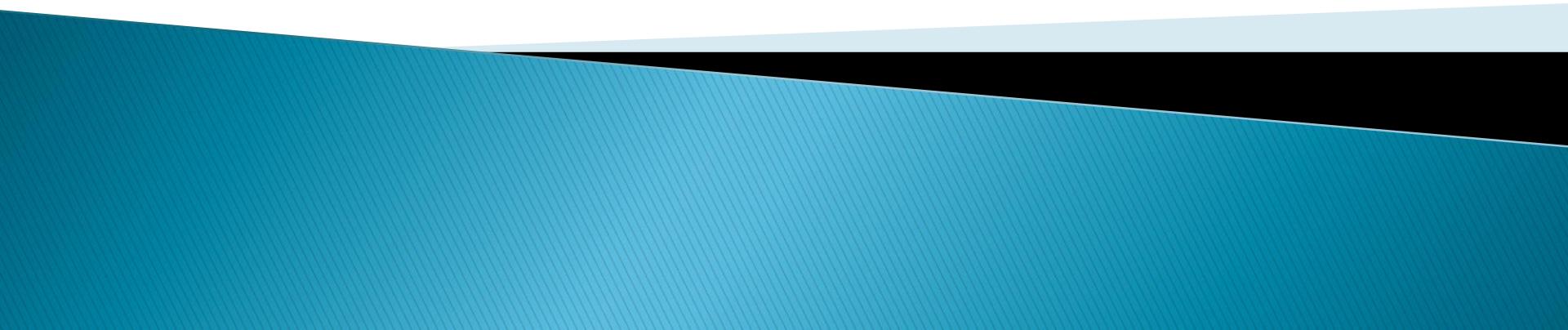
void bf(int s) {
  int p,u,i,j;
  p = u = 1; c[1]=s;
  viz[s]=1; niv[s]=0;
  while(p<=u) {
    i=c[p]; p=p+1;
    cout<<i<<" ";
    for(k=1;k<=l[i][0];k++)
    {
      j=l[i][k];
      if(viz[j]==0){
        u=u+1; c[u]=j;
        viz[j]=1;
        tata[j]=i;
        niv[j]=niv[i]+1;
      }
    }
  }
}

```

Complexitate

- ▶ Matrice de adiacență $O(|V|^2)$
- ▶ Liste de adiacență $O(|V| + |E|)$

Aplicații



Problemă



Dat un graf orientat și două vârfuri u și v , să se determine un drum minim de la u la v

Problemă



- Se apelează $bf(u)$, apoi se afișează drumul de la u la v folosind vectorul tata (ca la arbori), **dacă există**

Problemă



- Se apelează bf(u), apoi se afișează drumul de la u la v folosind vectorul tata (ca la arbori), **dacă există**

```
bf(u) ;  
if(viz[v] == 1)  
    drum(v) ;  
else  
    cout<<"nu exista drum" ;
```

unde funcția drum este cea de la arbori:

Problemă



- Se apelează bf(u), apoi se afișează drumul de la u la v folosind vectorul tata (ca la arbori), **dacă există**

```
bf(u);  
if(viz[v] == 1)  
    drum(v);  
else  
    cout<<"nu există drum";
```

unde funcția drum este cea de la arbori:

```
void drum(int x) {  
    while(x!=0) {  
        cout<<x<<" ";  
        x=tata[x];  
    }  
}  
  
void drumRec(int x) {  
    if(x!=0) {  
        drumRec(tata[x]);  
        cout<<x<<" ";  
    }  
}
```

Observații



- Parcurgerea $bf(u)$ se poate opri atunci când este vizitat v

Observații



Dacă în loc de graf avem un arbore dat prin vectorul tata:

- ▶ putem folosi parcurgerea BF din rădăcina arborelui pentru a determina nivelul fiecărui vârf și înălțimea arborelui
- ▶ putem construi matricea de adiacență sau listele de adiacență/**de fii** asociate arborelui
 - ➡ o nouă soluție pentru problemele anterioare de la arbori

Problemă



Să se verifice dacă un graf dat este conex

Problemă



Graful este conex dacă, prin apelul parcurgerii din vârful 1, sunt vizitate toate vârfurile

Problemă



Graful este conex dacă, prin apelul parcurgerii din vârful 1, sunt vizitate toate vârfurile

```
bf(1) ;  
  
ok=1 ;  
  
for(i=1;i<=n;i++)  
  
    if(viz[i]==0)  
  
        ok=0 ;  
  
    if(ok==1)  
  
        cout<<"conex" ;  
  
else  
  
    cout<<"neconex" ;
```

Problemă



- Într-un grup de n persoane se cunosc perechile de persoane care pot comunica direct.
- a) Să se caore este numărul maxim de persoane din grup care pot comunica oricare două, direct sau prin intermediari
 - b) Să se afișeze persoanele cele mai influente din grup. O persoană este considerată mai influentă decât alta dacă poate comunica direct cu mai multe persoane

Parcurgerea grafurilor



Idee: Asociem grupului un graf în care vârfurile sunt persoane și muchiile unește persoane care pot comunica direct.

Parcurgerea grafurilor



Idee: Asociem grupului un graf în care vârfurile sunt persoane și muchiile unește persoane care pot comunica direct.

- a) Problema este echivalentă cu a determina componenta conexă cu numărul maxim de vârfuri în graful asociat
- b) Problema este echivalentă cu a determina vârfurile de grad maxim – temă

Parcurgerea grafurilor



- Modificăm $bf(s)$ astfel încât să returneze numărul de vârfuri vizitate

Parcurgerea grafurilor



- Modificăm $bf(s)$ astfel încât să returneze numărul de vârfuri vizitate
- Apelăm funcția bf pentru fiecare vârf nevizitat anterior pentru a determina componentele conexe ale grafului și reținem valoarea maximă returnată

```
int n, a[20][20], viz[20];
int p,u,c[20];
int bf_nr(int s){
    int p,u,i,j,nr=0;
    p = u = 1; c[1]=s;
    viz[s]=1;
    while(p<=u) {
        i=c[p]; p=p+1;
        nr++; //cout<<i<<" ";
        for(j=1;j<=n;j++)
            if(a[i][j]==1)
                if(viz[j]==0){
                    u=u+1; c[u]=j;
                    viz[j]=1;
                }
    }
    return nr;
}
```

```
int n, a[20][20], viz[20];
int p,u,c[20];
int bf_nr(int s){
    int p,u,i,j,nr=0;
    p = u = 1; c[1]=s;
    viz[s]=1;
    while(p<=u) {
        i=c[p]; p=p+1;
        nr++; //cout<<i<<" ";
        for(j=1;j<=n;j++)
            if(a[i][j]==1)
                if(viz[j]==0){
                    u=u+1; c[u]=j;
                    viz[j]=1;
                }
    }
    return nr;
}
```

```
//in main
int i,nr,nrmax=0;
for(i=1;i<=n;i++)
    if(viz[i]==0) {
        nr=bf_nr(i);
        if(nr>nrmax)
            nrmax=nr;
    }
cout<<"nr. max:"<<nrmax<<endl;
```

Problemă – sortarea topologică



- ▶ Reguli – Olimpiada de Informatică etapa pe sector 2013

Ionuț a cumpărat un joc format din componentele necesare asamblării unui obiect. A studiat prospectul jocului în care sunt precizate etapele pe care trebuie să le urmeze pentru o asamblare corectă a obiectului. Ionuț a observat că nu este dată ordinea exactă a etapelor, ci se precizează ce **etape de asamblare trebuie realizate înaintea altora**. Ionuț trebuie să asambleze obiectul în n etape, numerotate de la 1 la n , respectând regulile de asamblare prevăzute în prospect. Astfel, dacă sunt trei etape și etapa 2 trebuie efectuată neapărat înaintea etapei 3, atunci o ordine corectă a etapelor poate fi: 1, 2, 3, sau 2, 1, 3, sau 2, 3, 1.

Cerință

Scriți un program care, cunoscând numărul de etape și regulile de succesiune între aceste etape, determină o succesiune corectă de etape prin care să se respecte regulile de asamblare din prospect.

Problemă – sortarea topologică



7 9 – numărul de etape și numărul de reguli

2 1

2 4

1 7

4 7

6 7

5 6

3 6

5 3

1 4

Problemă – sortarea topologică



- Întâi planificăm activitățile care nu trebuie planificate înaintea altor activități,
- Modelăm problema cu grafuri
 - Întâi considerăm vârfurile cu grad interior 0

Problemă – sortarea topologică



- Întâi planificăm activitățile care nu trebuie planificate înaintea altor activități,
- Modelăm problema cu grafuri
 - întâi considerăm vârfurile cu grad interior 0
 - *eliminăm* aceste vârfuri

Problemă – sortarea topologică



- Întâi planificăm activitățile care nu trebuie planificate înaintea altor activități,
- Modelăm problema cu grafuri
 - întâi considerăm vârfurile cu grad interior 0
 - *eliminăm* aceste vârfuri
 - reluăm procedeul

Problemă – sortarea topologică



- Observații

- Vârfurile nu trebuie eliminate, ci doar scăzute gradele interne ale vecinilor
- La un pas pot deveni vârfuri cu grad intern 0 doar acele vârfuri în care intrau arce cu o extremitate în vârfuri eliminate la pasul anterior
- Implementare – similară cu BF

```
#include<fstream>

using namespace std;
ifstream f("reguli.in");
ofstream g("reguli.out");
int a[100][100],x,y,n, gin[100],r,i,j;
//gin[i]=grad intern al lui i

int main(){
    int c[100],p,u;
    f>>n>>r;
    //construim matricea de adiacenta coresp. regulilor
    for(i=1;i<=r;i++) {
        f>>x>>y;
        a[x][y]=1;
        gin[y]++;
    }
}
```

```
//adaugam toate elementele cu grad intern 0 in coada c
p=0;u=-1;
for(i=1;i<=n;i++)
    if(gin[i]==0)
    {
        u++;  c[u]=i;
    }
```

```
/*extragem un element de grad 0 din coada,  
il adaugam in solutie si il eliminam din graf = scadem  
cu 1 gradul intern al vecinilor sai;  
Daca se obtin astfel varfuri de grad intern 0, le  
adaugam in coada c */
```

```
while(p<=u) {  
    x=c[p]; p++;  
    g<<x<<" ";  
    for(j=1;j<=n;j++)  
        if(a[x][j]==1){  
            gin[j]--;  
            if(gin[j]==0){  
                u++;  
                c[u]=j;  
            }  
        }  
}
```

Problemă – sortarea topologică



Temă

1. Modificați programul astfel încât graful să fie memorat cu liste de adiacență
2. Modificați programul astfel încât să fie detectată existența de circuite (date incorecte - dependențe circulare)

Problemă – sortarea topologică



Varianta 2 – folosim parcurgerea în adâncime DF

- Cu cât un vârf este finalizat (explorat cu toate vârfurile accesibile din el) mai târziu în parcurgerea DF, cu atât el este mai la început în ordonare (în sortarea topologică)
 - Adăugăm într-o stivă vârfurile, pe măsura terminării explorării lor (adică explorării tuturor vecinilor)
 - Eliminăm vârfurile din stivă pe rând și le afișăm

Problemă - Transmiterea unui mesaj în rețea



- ▶ Între participanții la un curs s-au legat relații de prietenie și comunică și în afara cursului.

Profesorul vrea să transmită un mesaj participanților și știe ce relații de prietenie s-au stabilit între ei. El vrea să contacteze cât mai puțini participanți, urmând ca aceștia să transmită mesajul între ei.

Ajutați-l pe profesor să decidă cui trebuie să transmită inițial mesajul și să atâșeze la mesaj o listă în care să arate fiecărui participant către ce prieteni trebuie să trimită mai departe mesajul, astfel încât mesajul să ajungă la fiecare participant la curs o singură dată.

Problemă - Transmiterea unui mesaj în rețea



- ▶ Idee – modelăm problema cu un graf neorientat
 - cui trebuie să transmită inițial mesajul = câte un vârf din fiecare componentă conexă

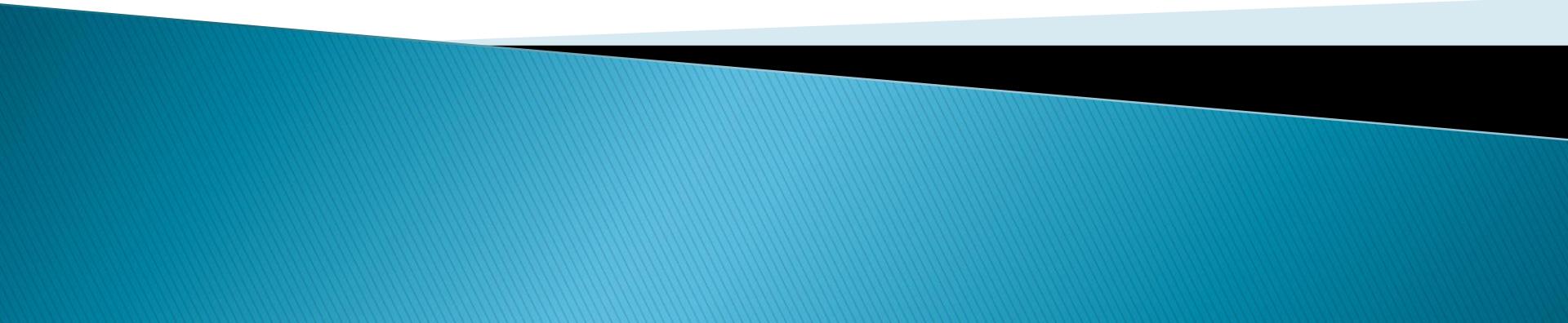
```
for(i=1;i<=n;i++)  
    if(viz[i]==0) {  
        bf(i);  
        cout<<i<<' ';  
    }
```

Problemă - Transmiterea unui mesaj în rețea



- ▶ Idee – modelăm problema cu un graf
 - Traseul mesajului în fiecare componentă = lista de fii pentru arborele bf memorat în vectorul tata (câte un arbore pentru fiecare componentă)
 - vezi programul pentru conversia de la vectorul tata la lista de fii

Parcurgerea în adâncime



Parcurgerea în adâncime

Se vizitează

- Inițial: vârful de start s – devine vârf curent

Parcurgerea în adâncime

Se vizitează

- Inițial: vârful de start s – devine vârf curent
- La un pas:
 - se trece la primul vecin nevizitat al vârfului curent, dacă există

Parcurgerea în adâncime

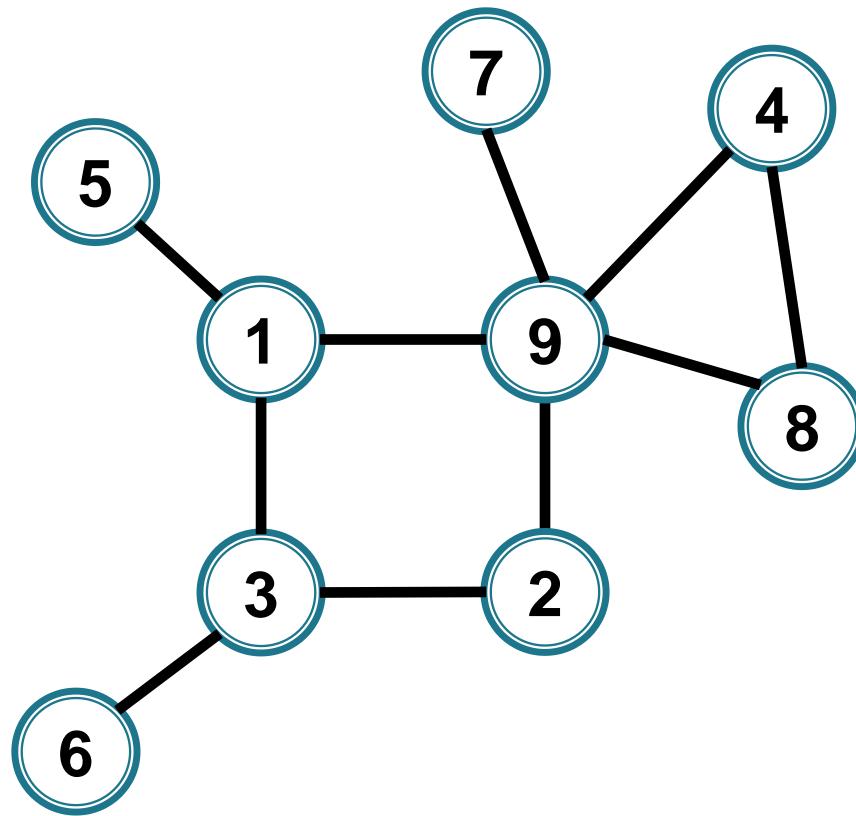
Se vizitează

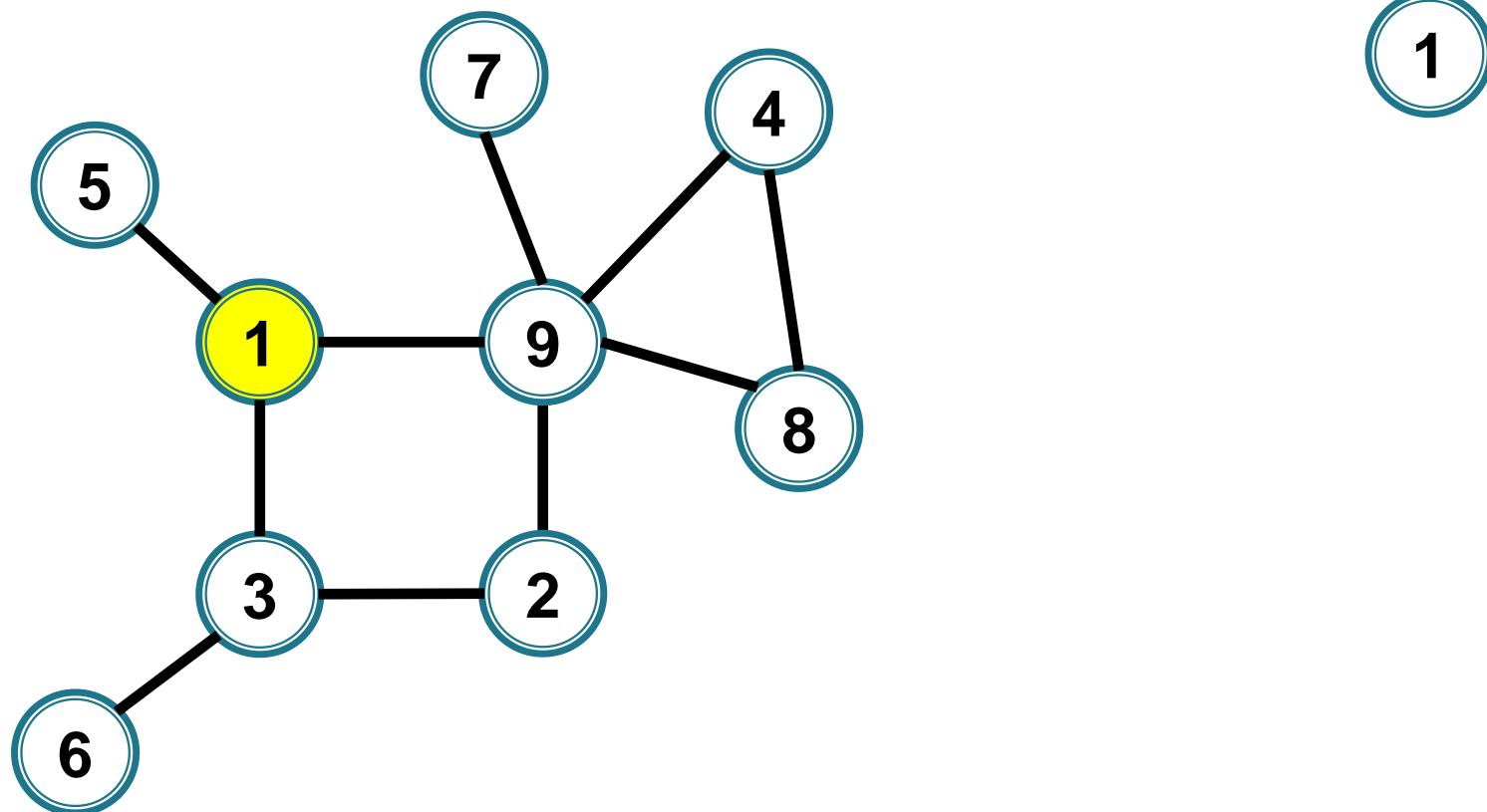
- Inițial: vârful de start s – devine vârf curent
- La un pas:
 - se trece la primul vecin nevizitat al vârfului curent, dacă există
 - altfel
 - se merge **înapoi** pe drumul de la s la vârful curent, până se ajunge la un vârf cu vecini nevizitați
 -

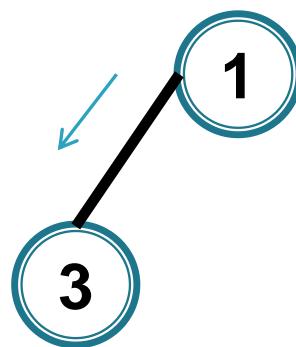
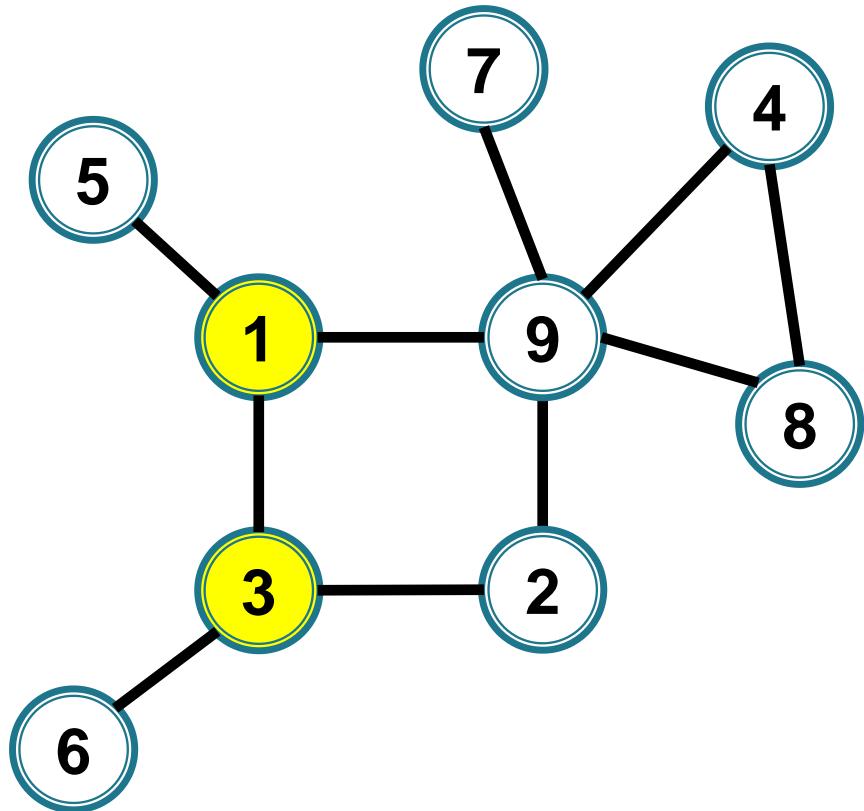
Parcurgerea în adâncime

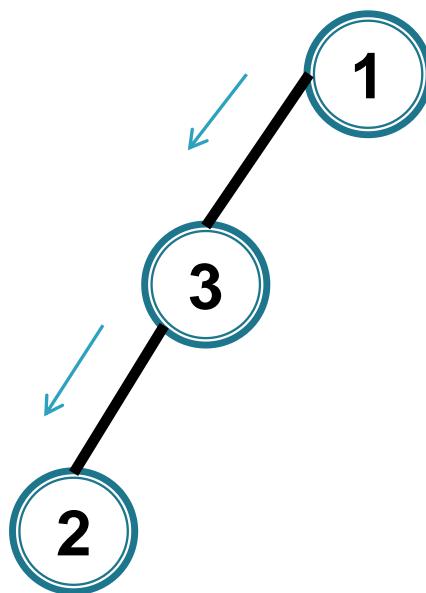
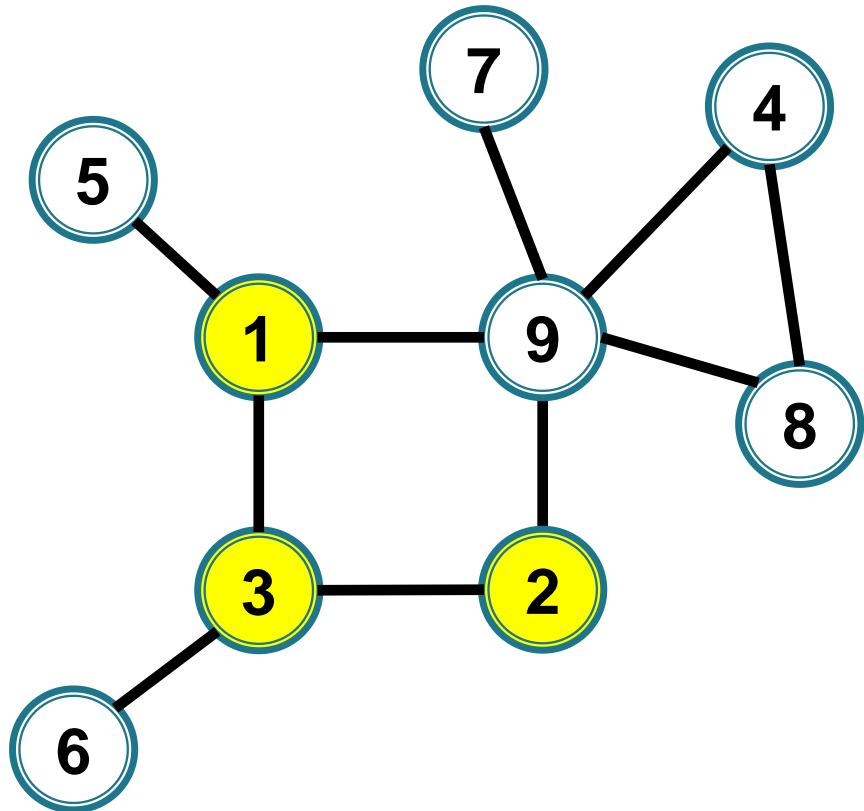
Se vizitează

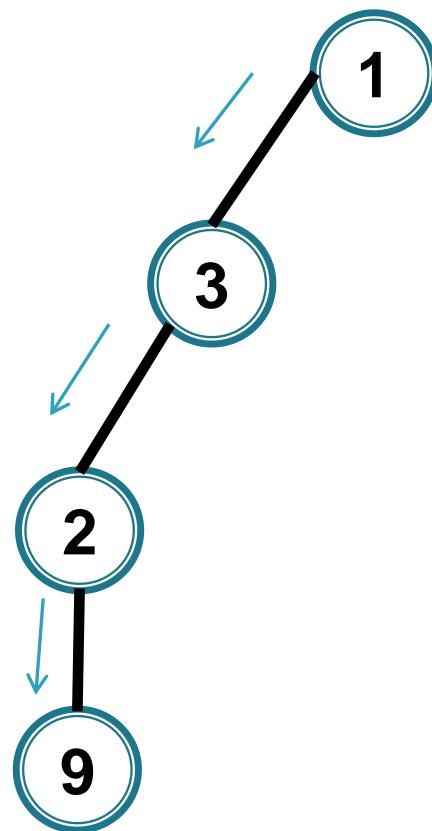
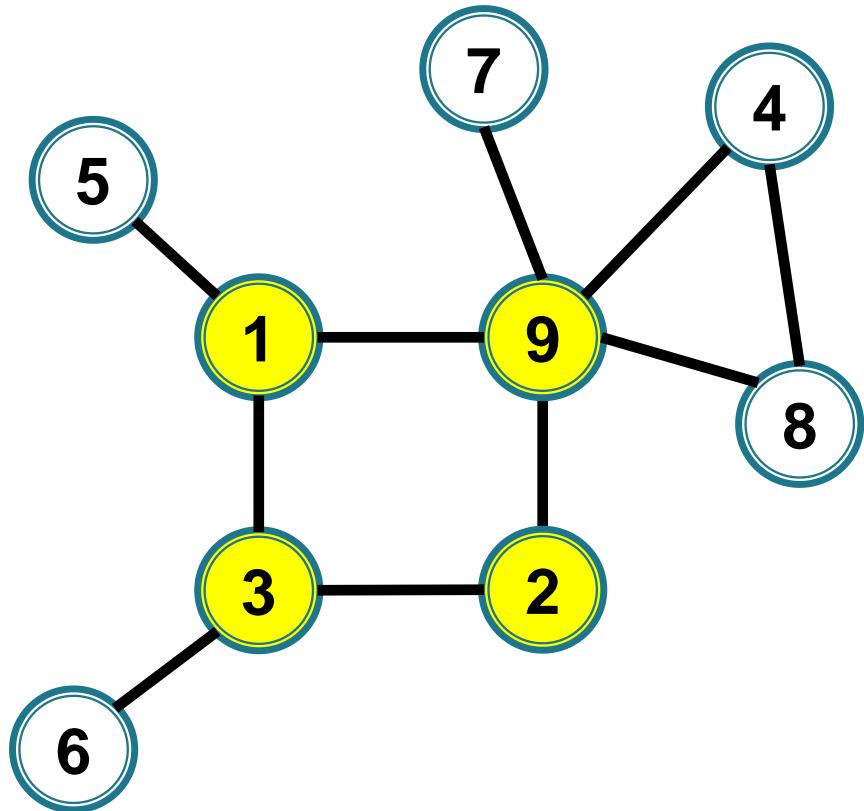
- Inițial: vârful de start s – devine vârf curent
- La un pas:
 - se trece la primul vecin nevizitat al vârfului curent, dacă există
 - altfel
 - se merge **înapoi** pe drumul de la s la vârful curent, până se ajunge la un vârf cu vecini nevizitați
 - se trece la **primul** dintre aceștia și se reia procesul

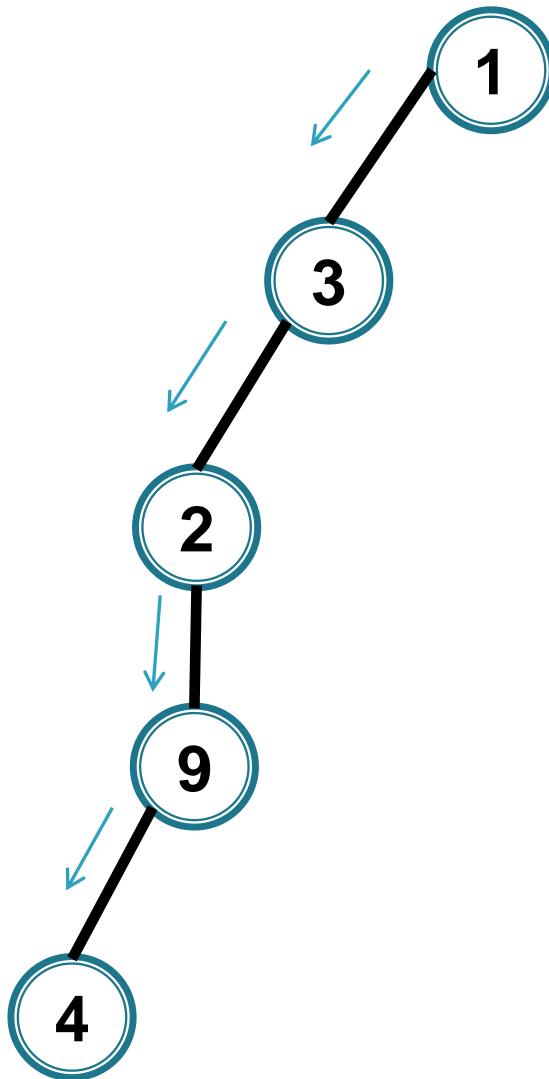
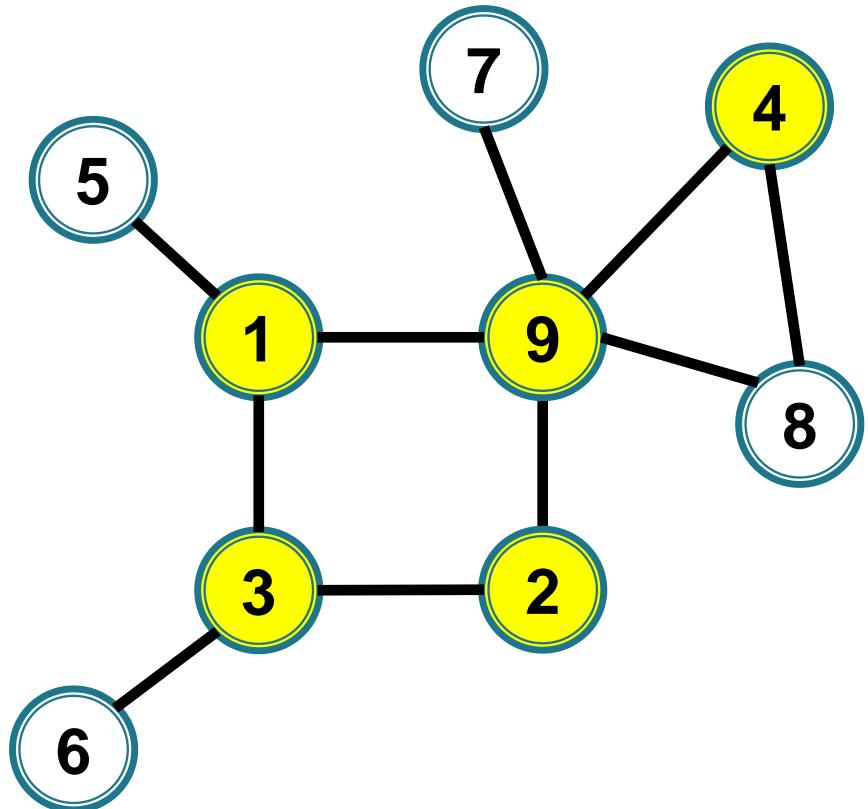


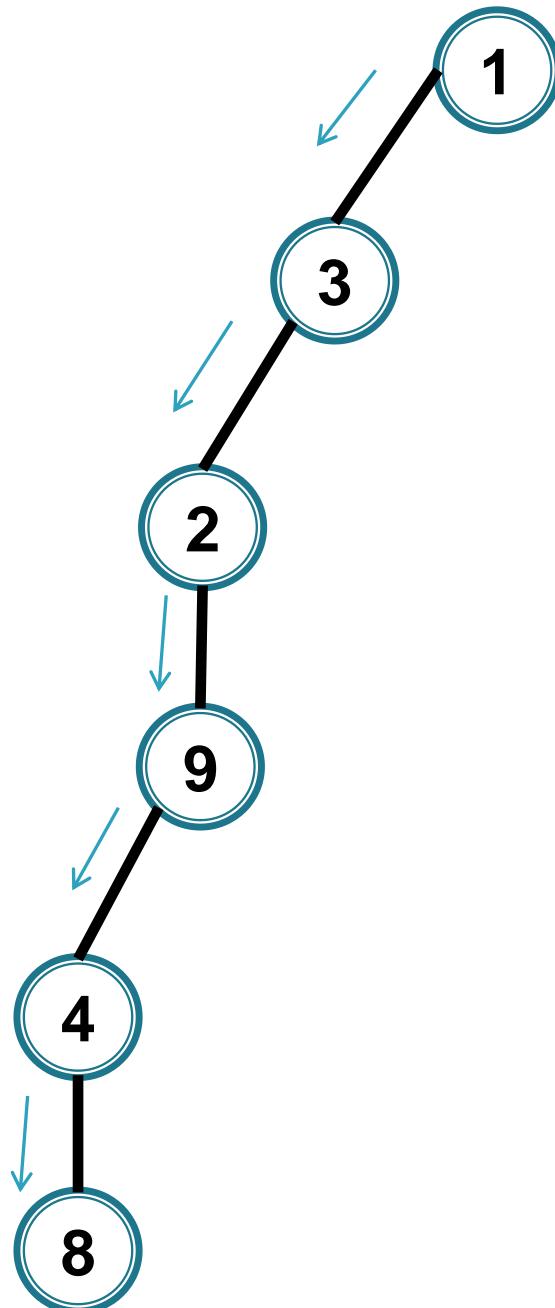
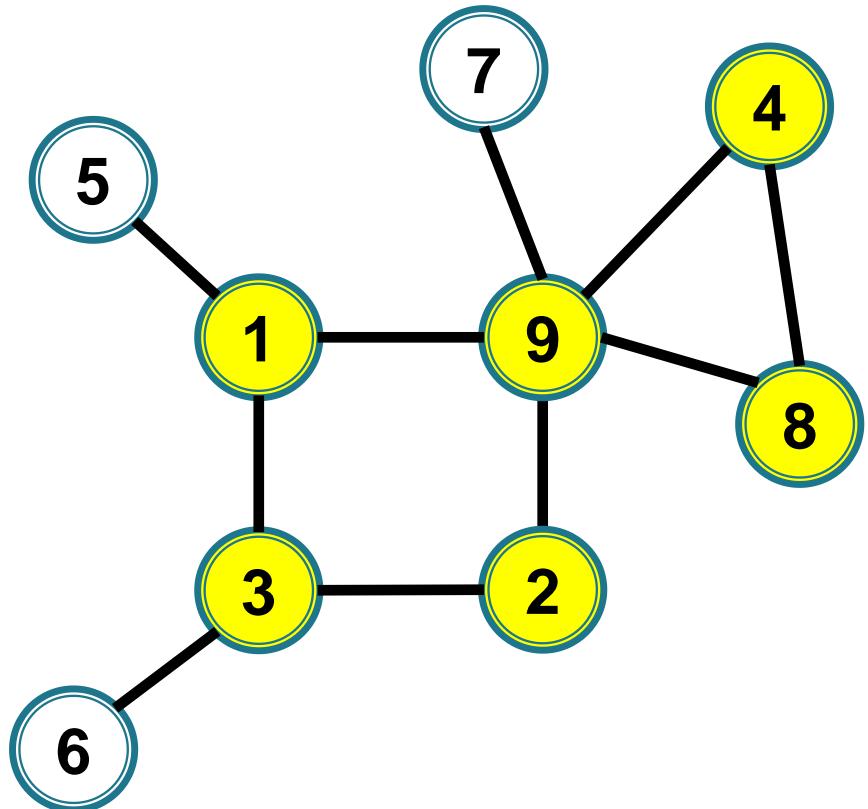


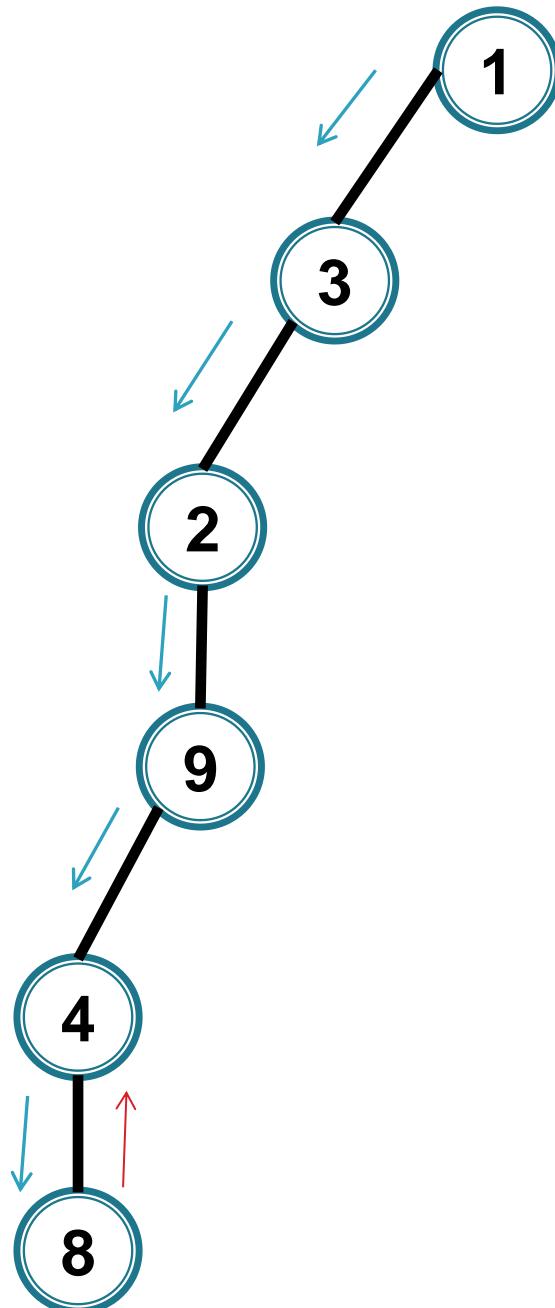
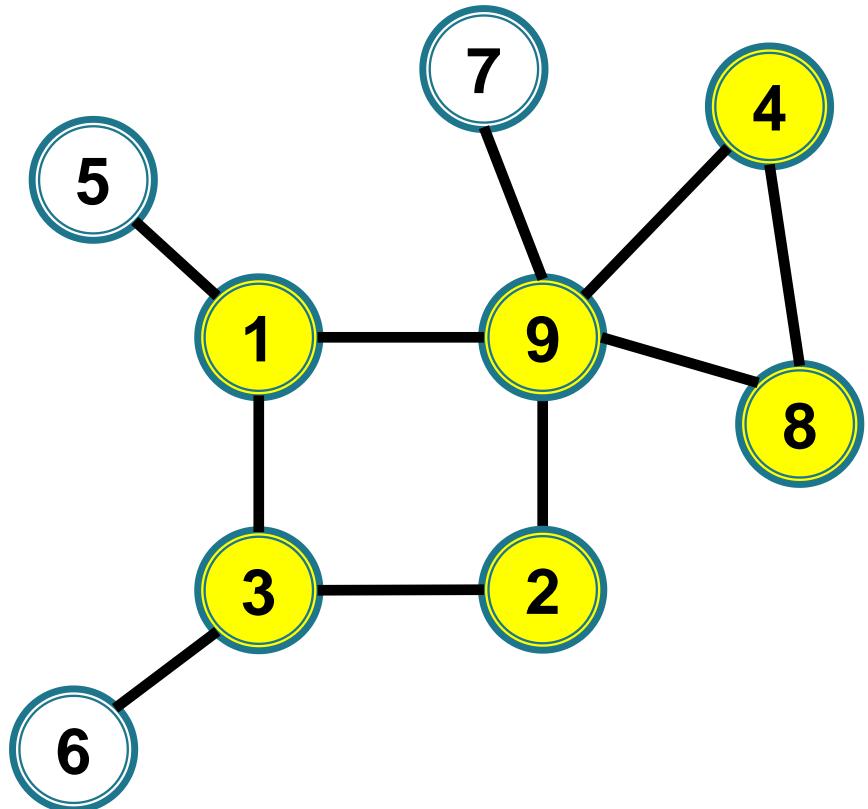


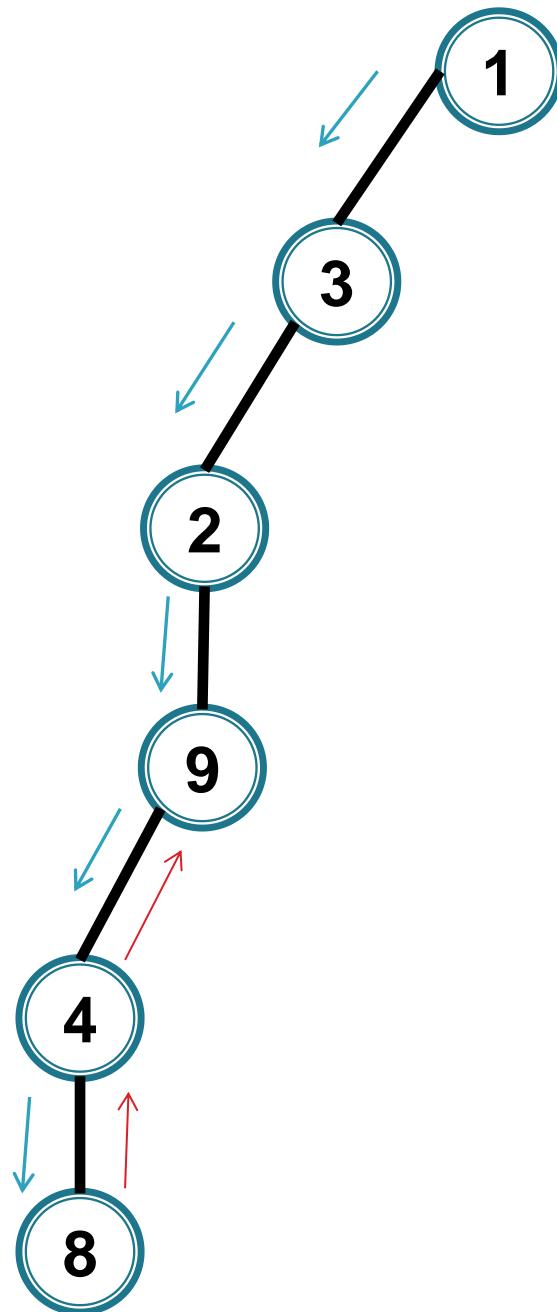
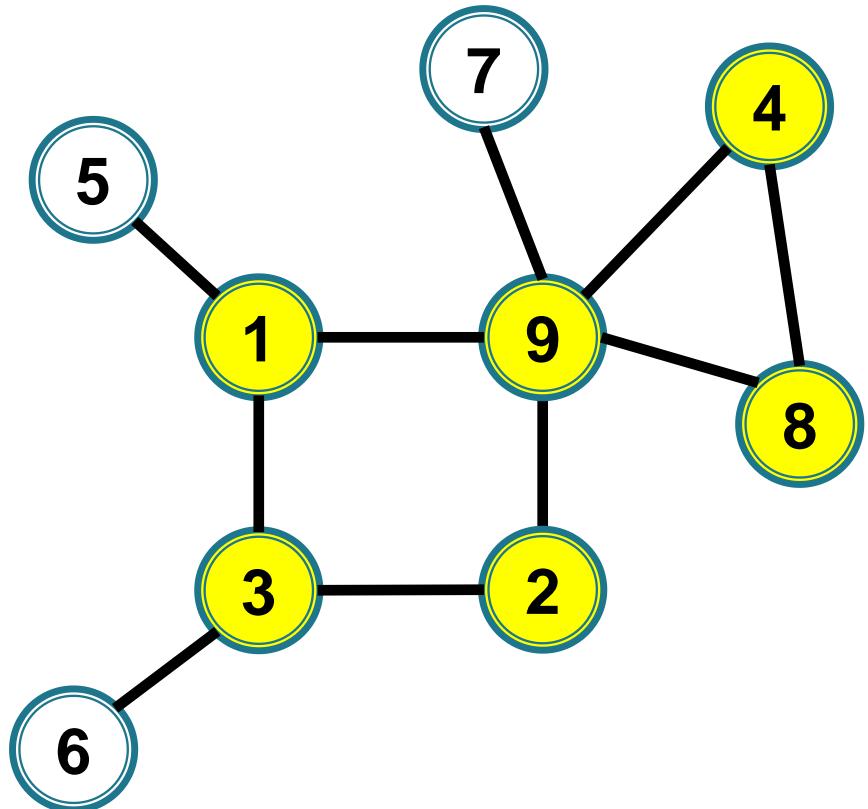


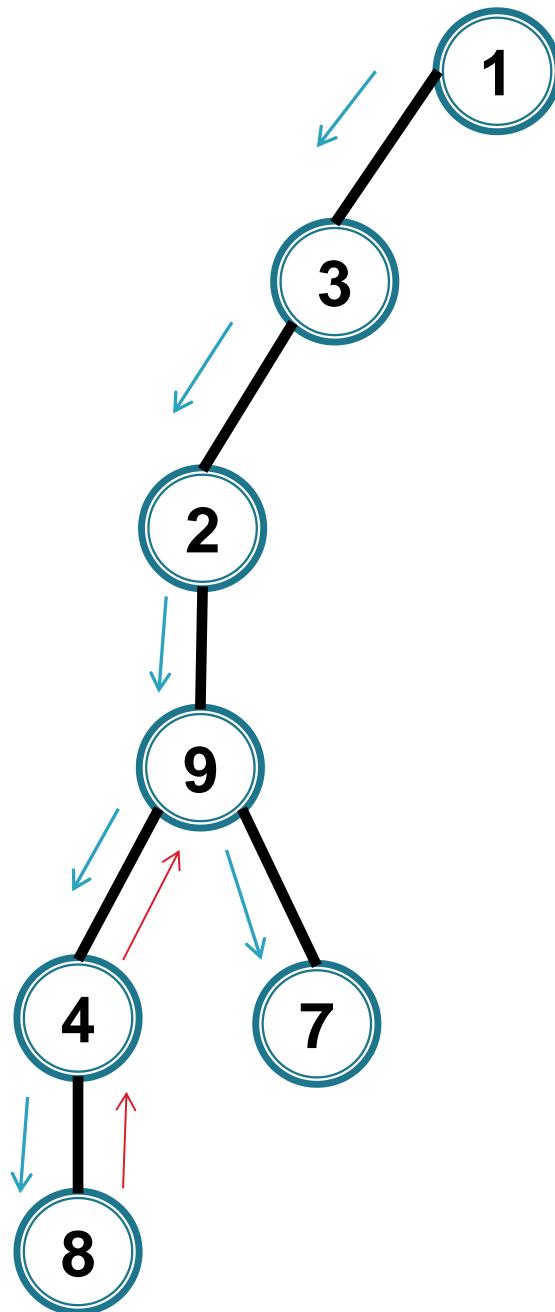
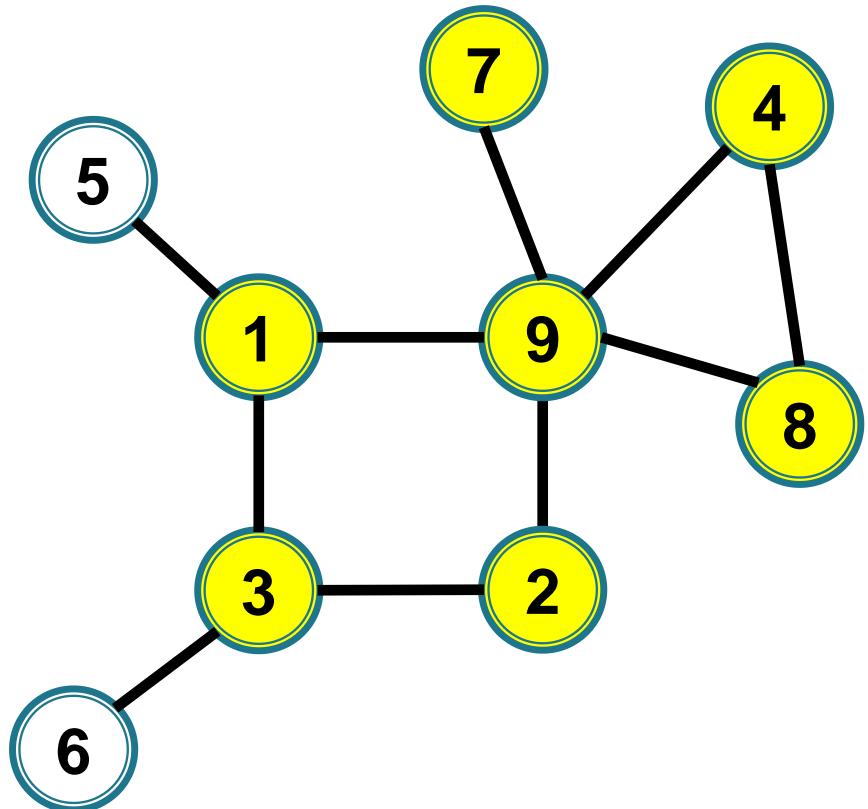


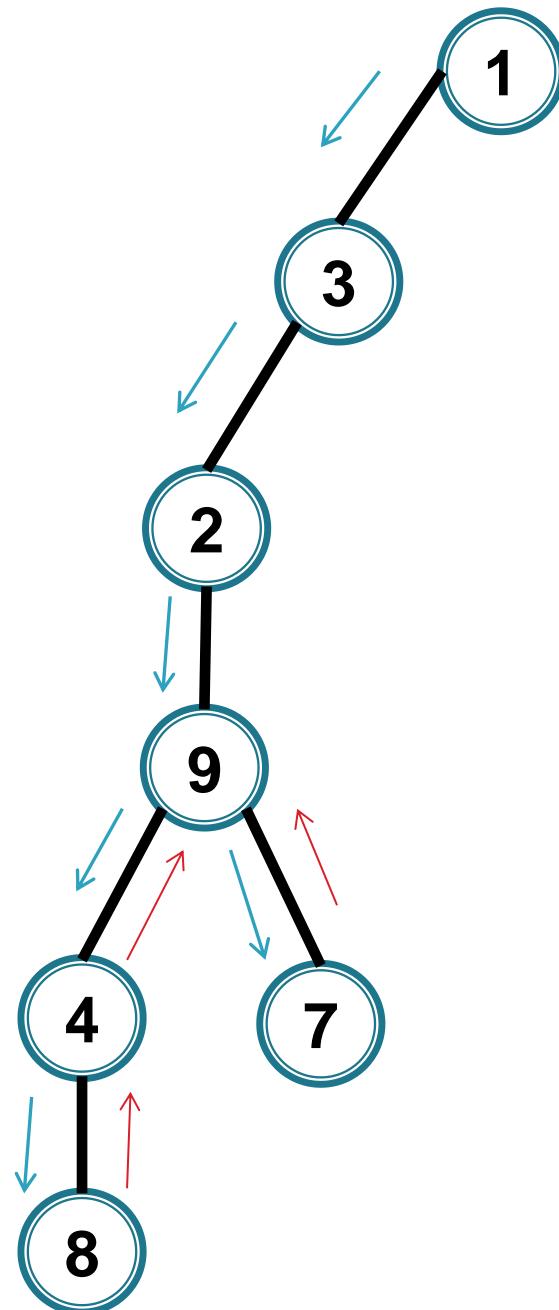
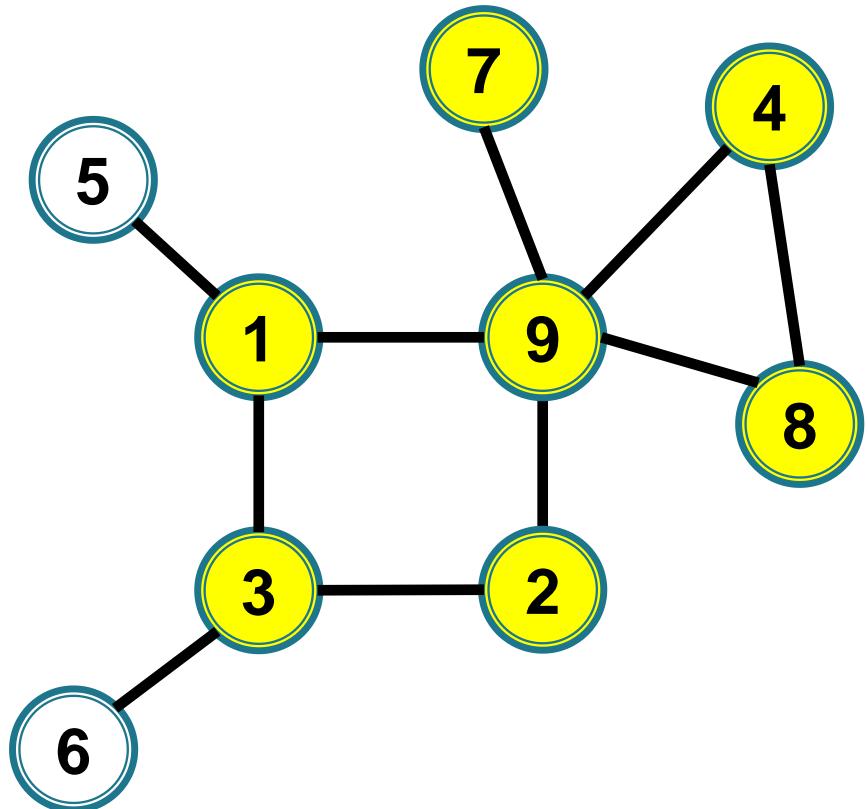


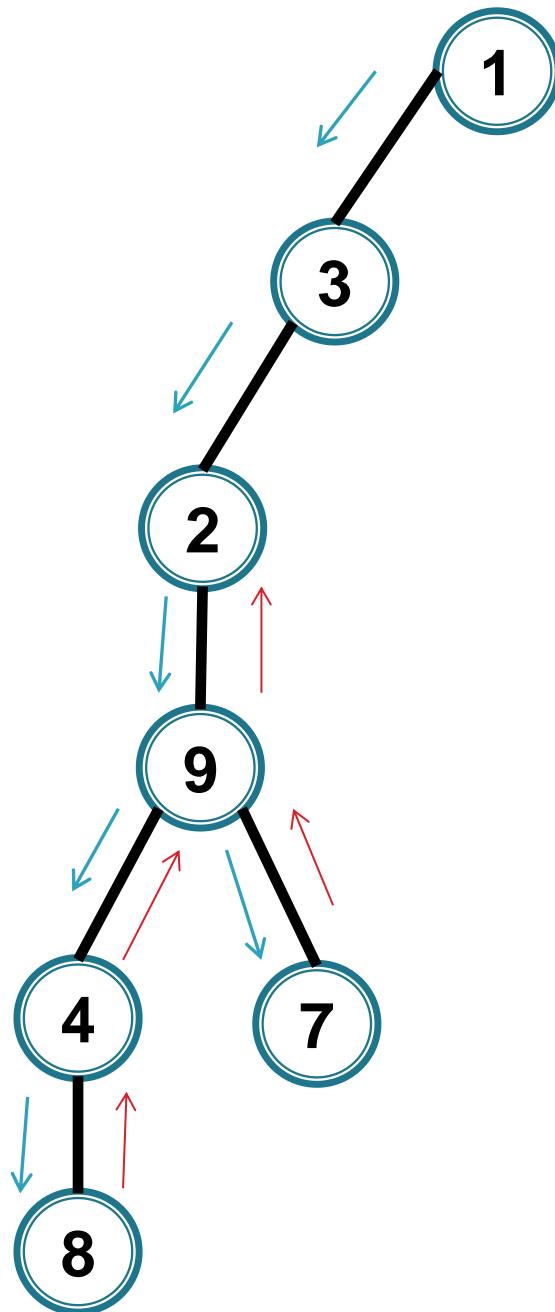
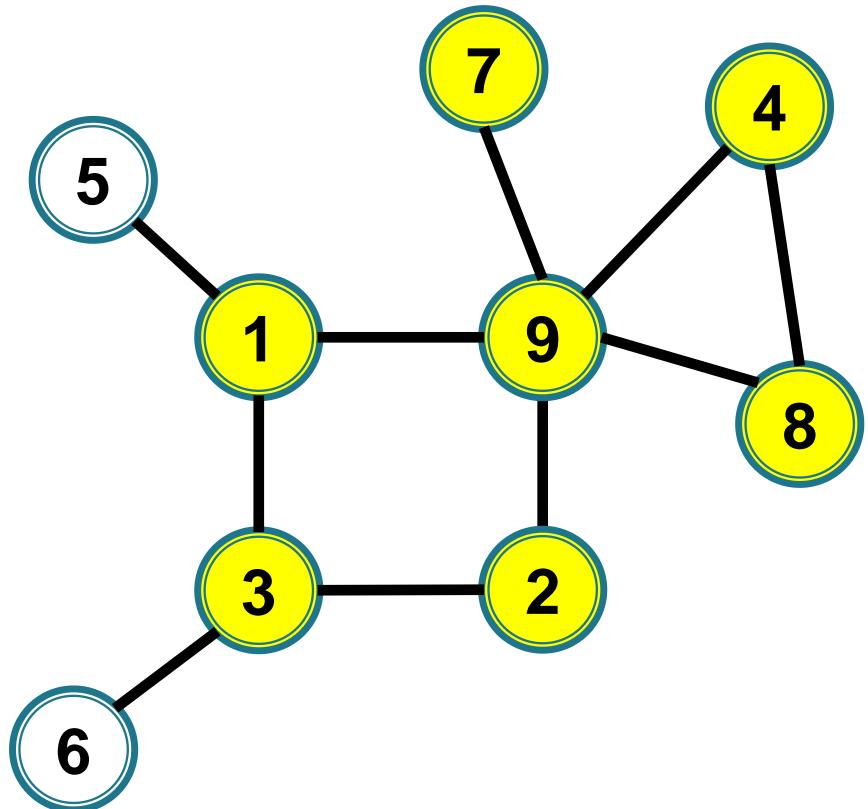


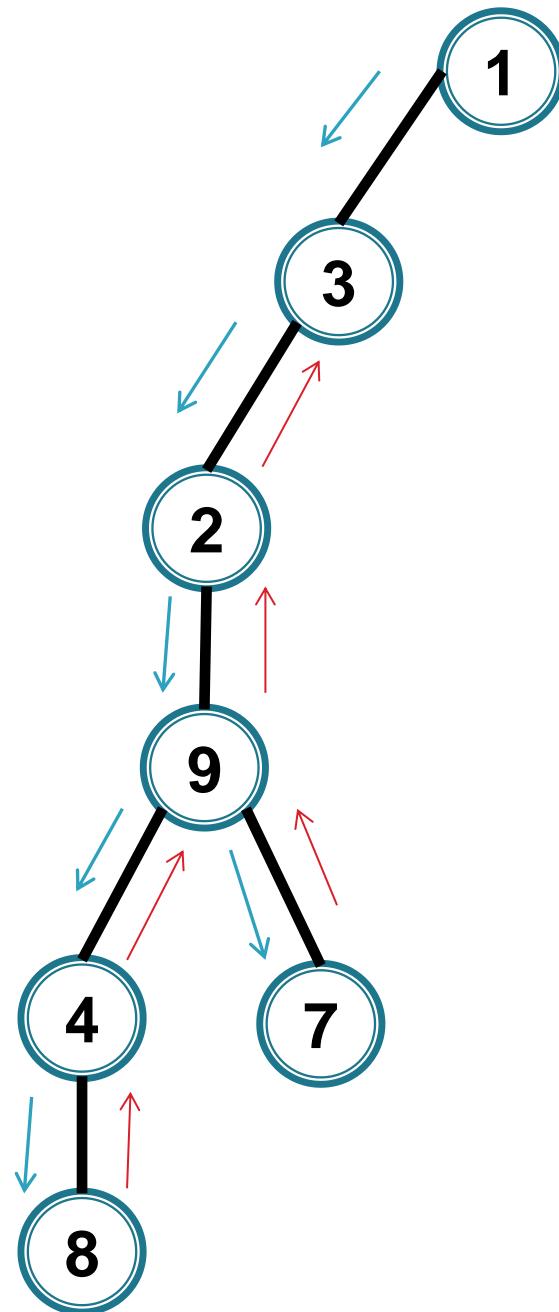
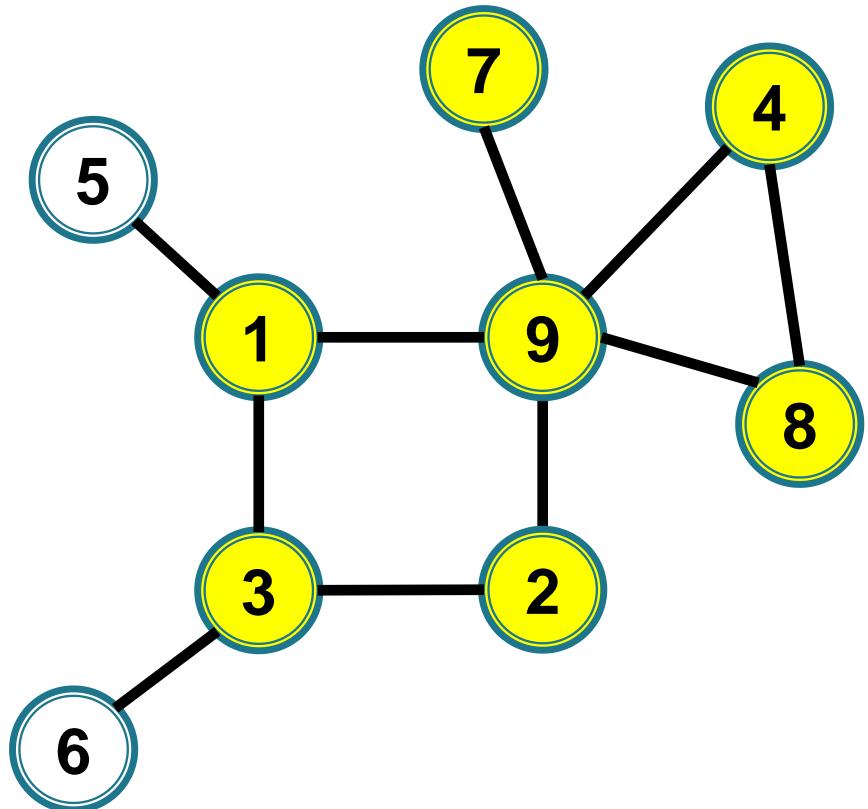


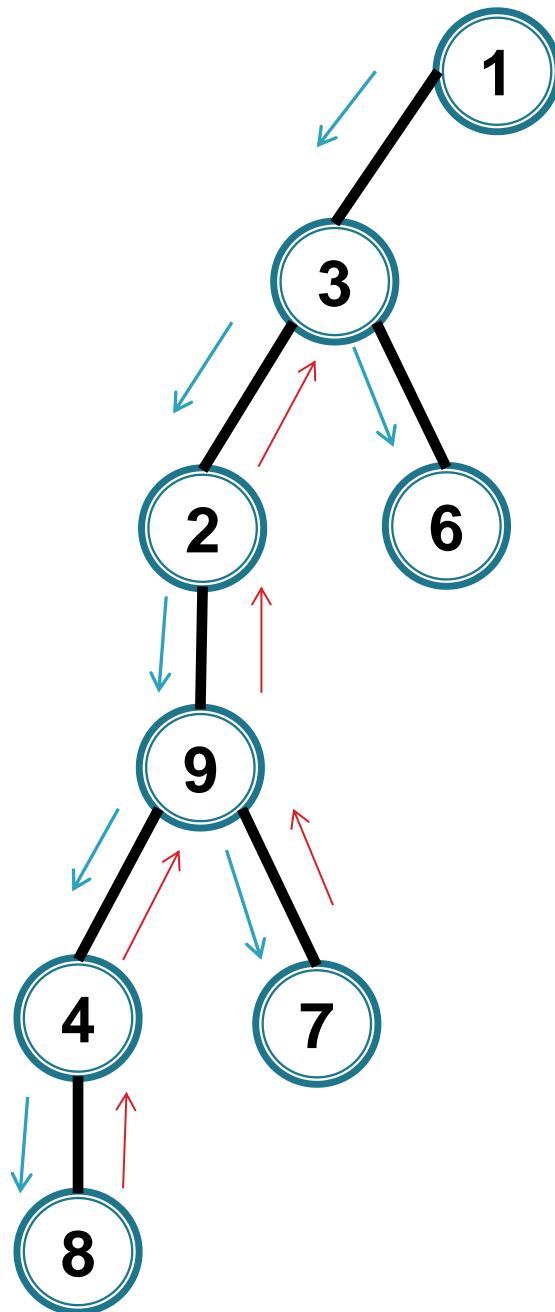
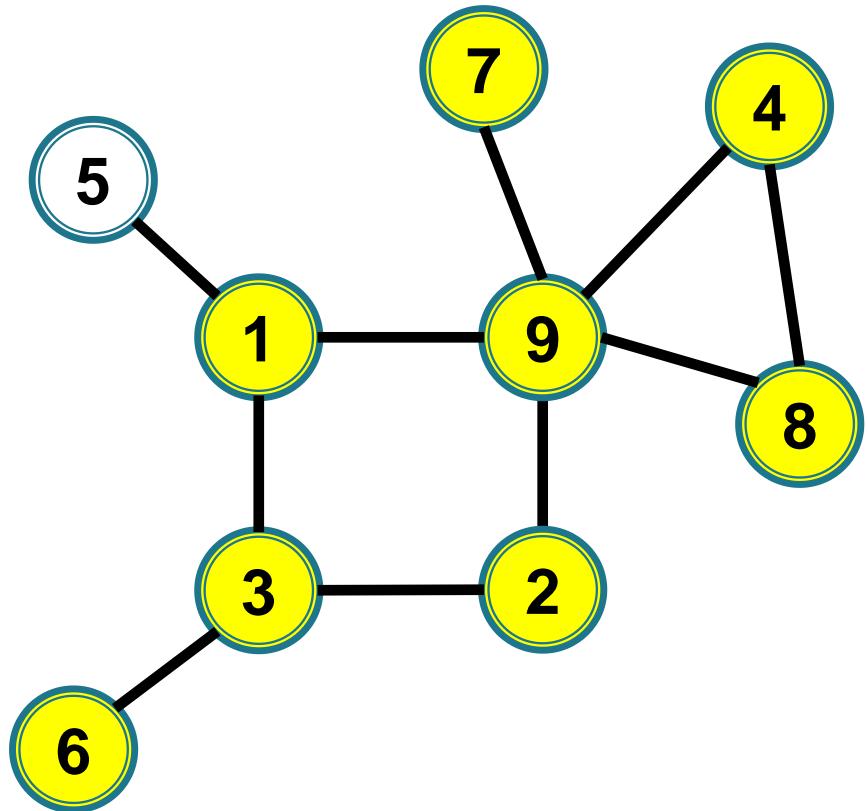


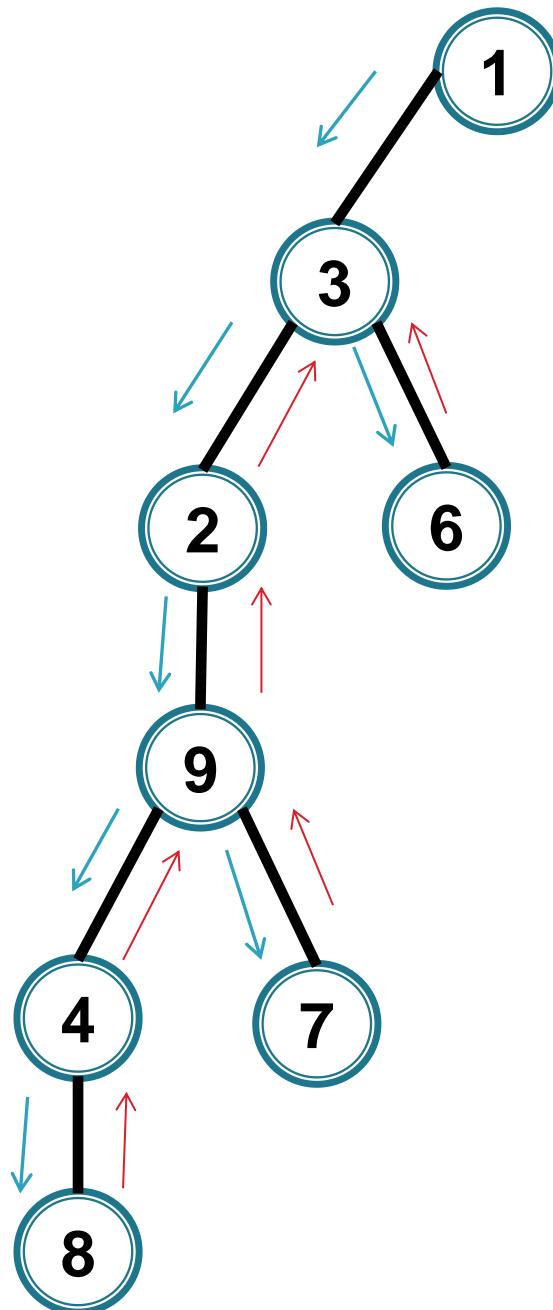
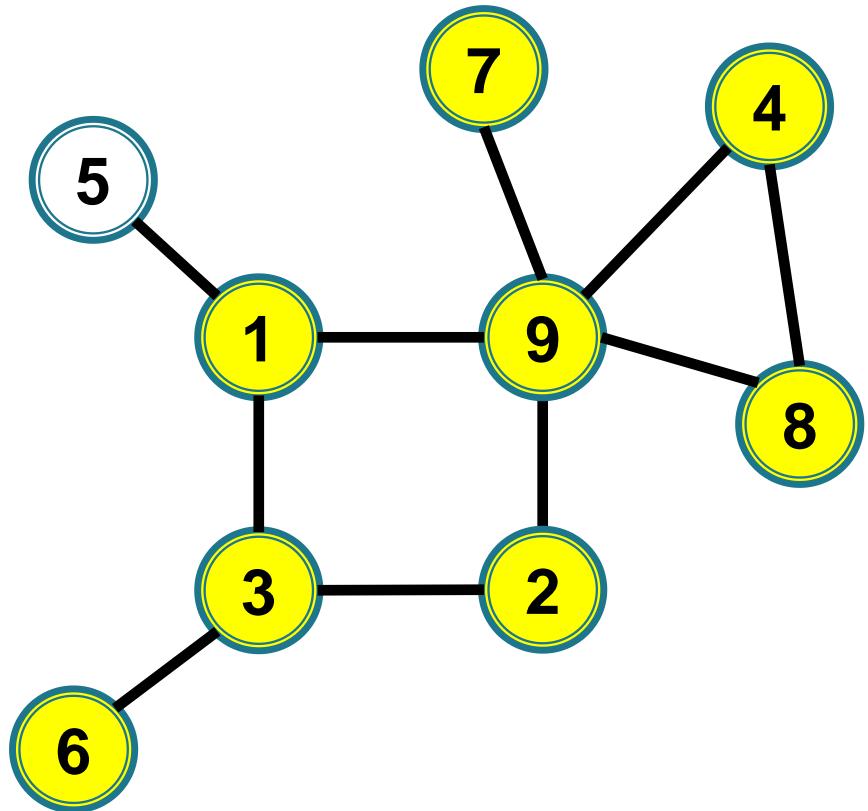


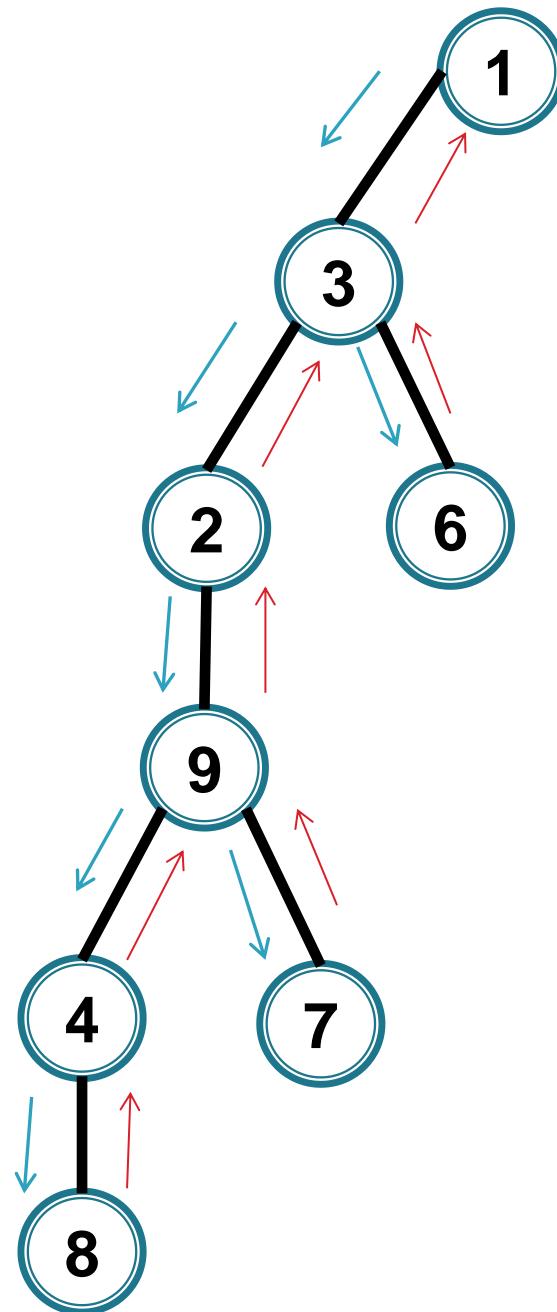
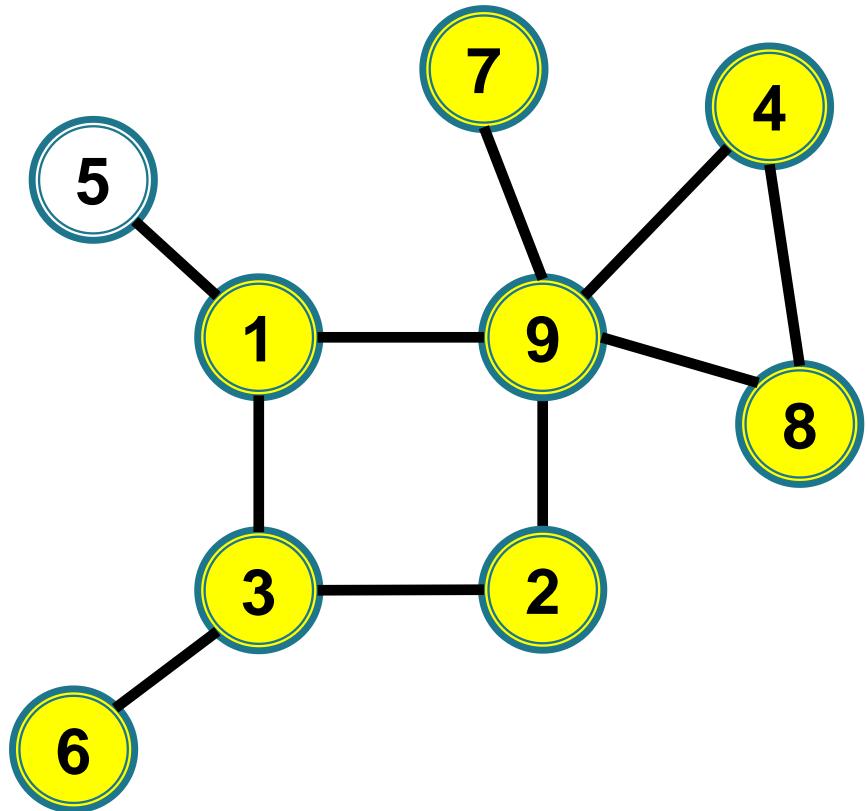


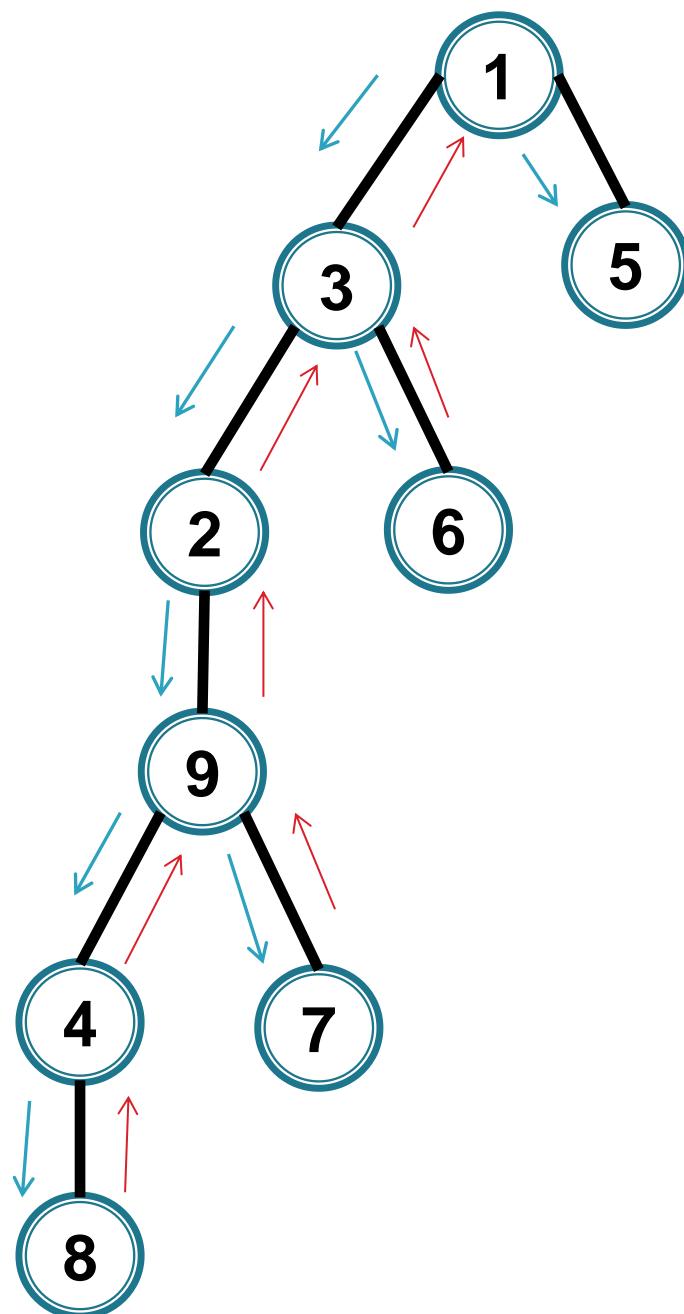
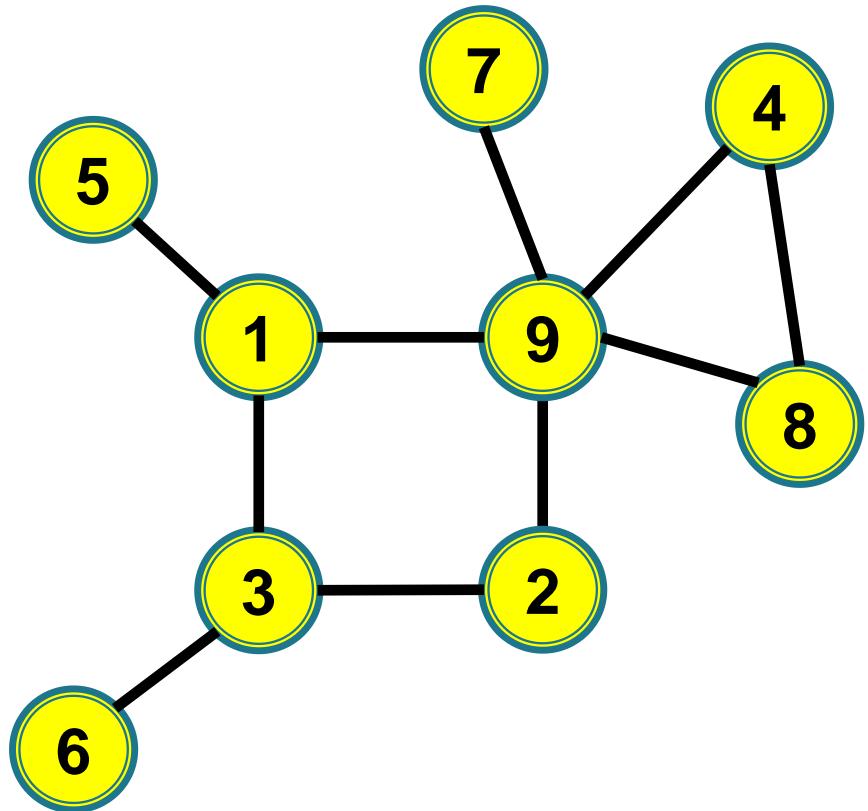


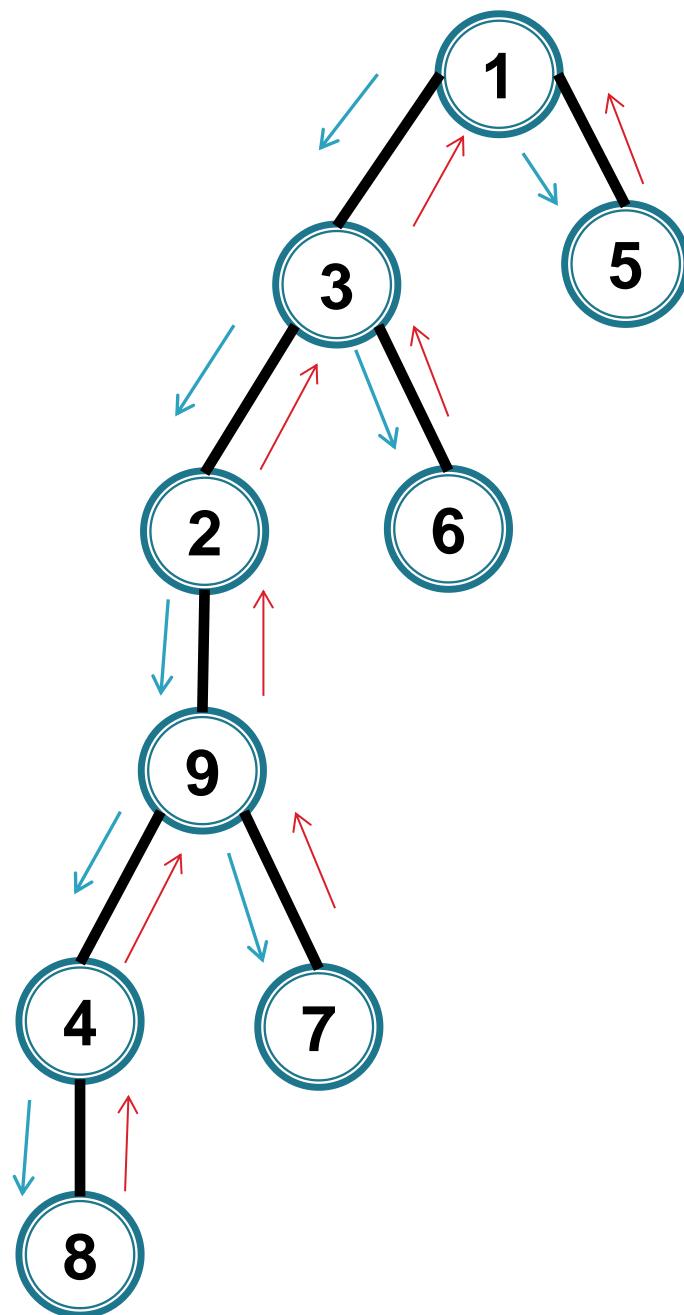
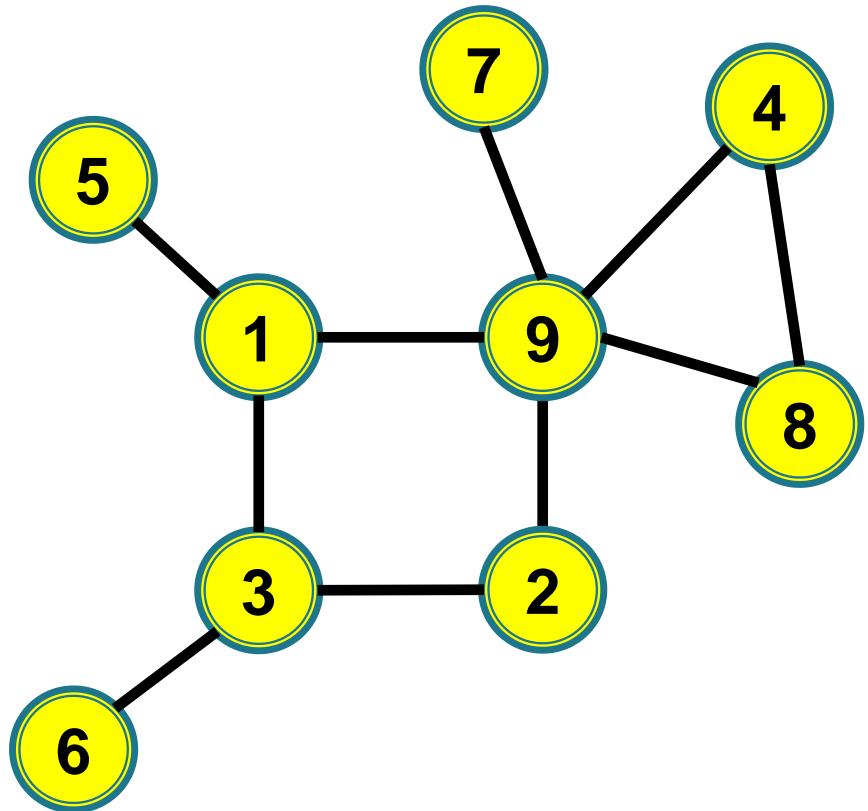


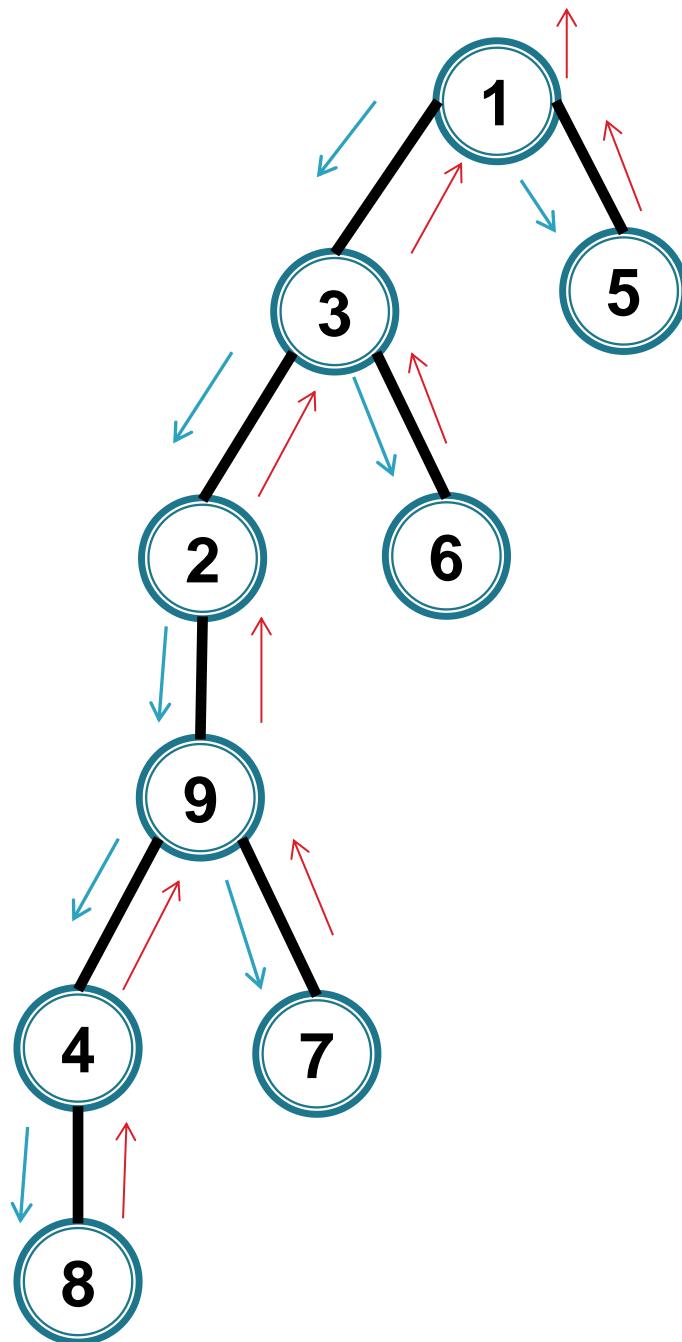
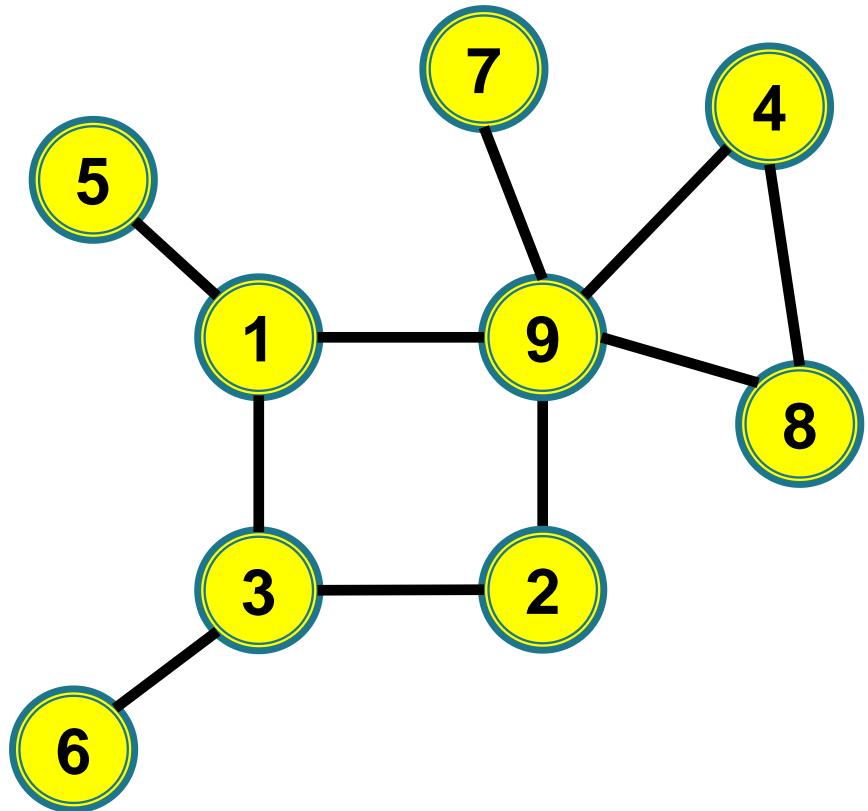












```
void df(int i) {
```

```
void df(int i){  
    viz[i]=1;  
    cout<<i<<" ";
```

```
void df(int i){  
    viz[i]=1;  
    cout<<i<<" ";  
    for(int j=1;j<=n ;j++)  
        if(a[i][j]==1)
```

```
void df(int i){  
    viz[i]=1;  
    cout<<i<<" ";  
    for(int j=1;j<=n ;j++)  
        if(a[i][j]==1)  
            if(viz[j]==0){  
                tata[j]=i;  
                df(j);  
            }  
}
```

```
void df(int i){  
    viz[i]=1;  
    cout<<i<<" ";  
    for(int j=1;j<=n ;j++)  
        if(a[i][j]==1)  
            if(viz[j]==0){  
                tata[j]=i;  
                df(j);  
            }  
    }  
}
```

Apel:

df(s)

Temă

- ▶ Dat un graf neorientat, să se verifice dacă graful conține cicluri și, în caz afirmativ, să se afișeze un ciclu al său

Succes la examenul de admitere!

