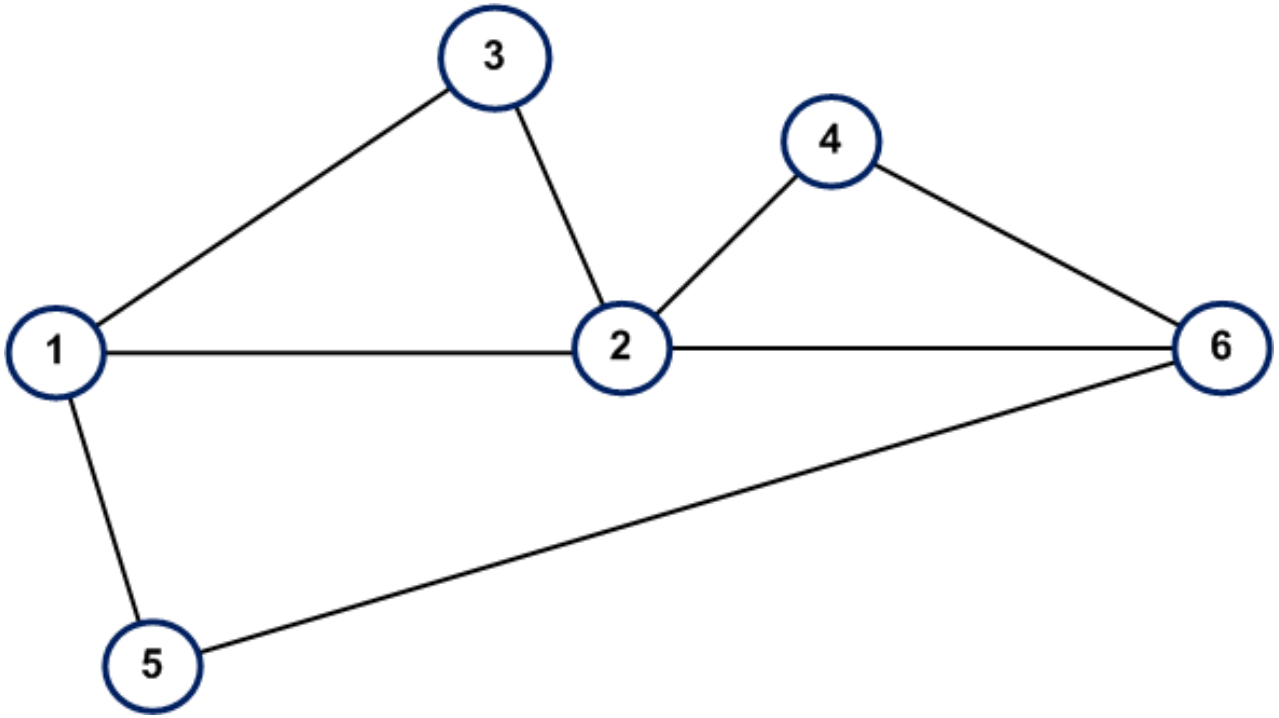
Modalităţi de reprezentare a grafurilor

Reprezentarea grafurilor

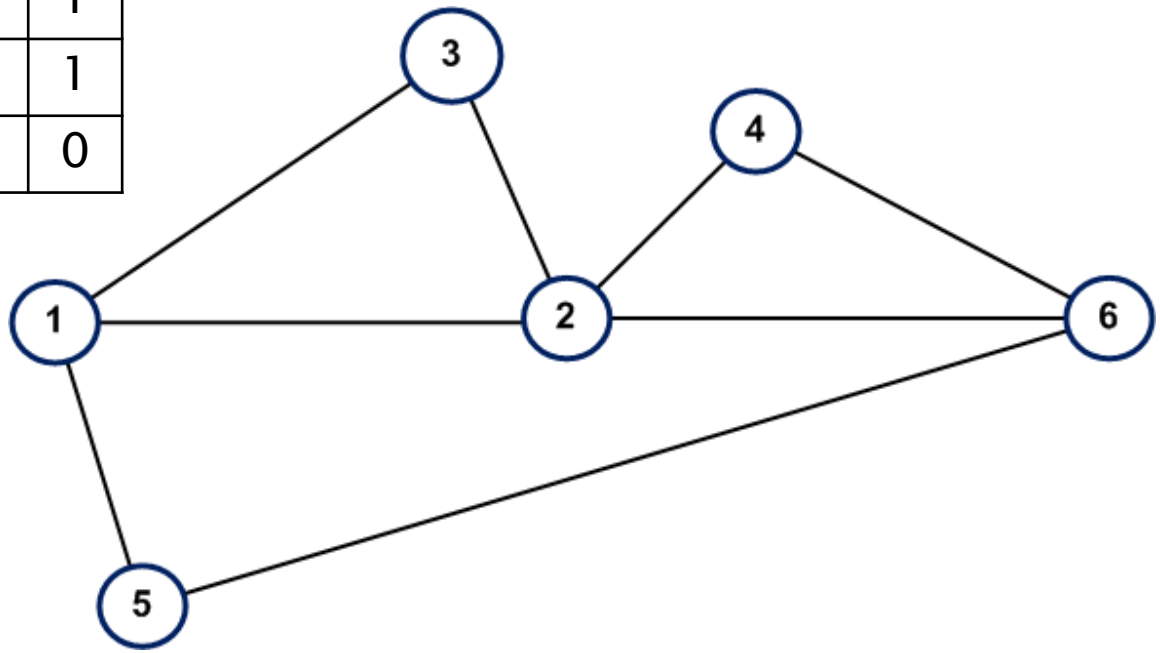
- Matrice de adiacenţă
- Liste de adiacenţă
- Listă de muchii/arce



Reprezentarea grafurilor

Matrice de adiacenţă

	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

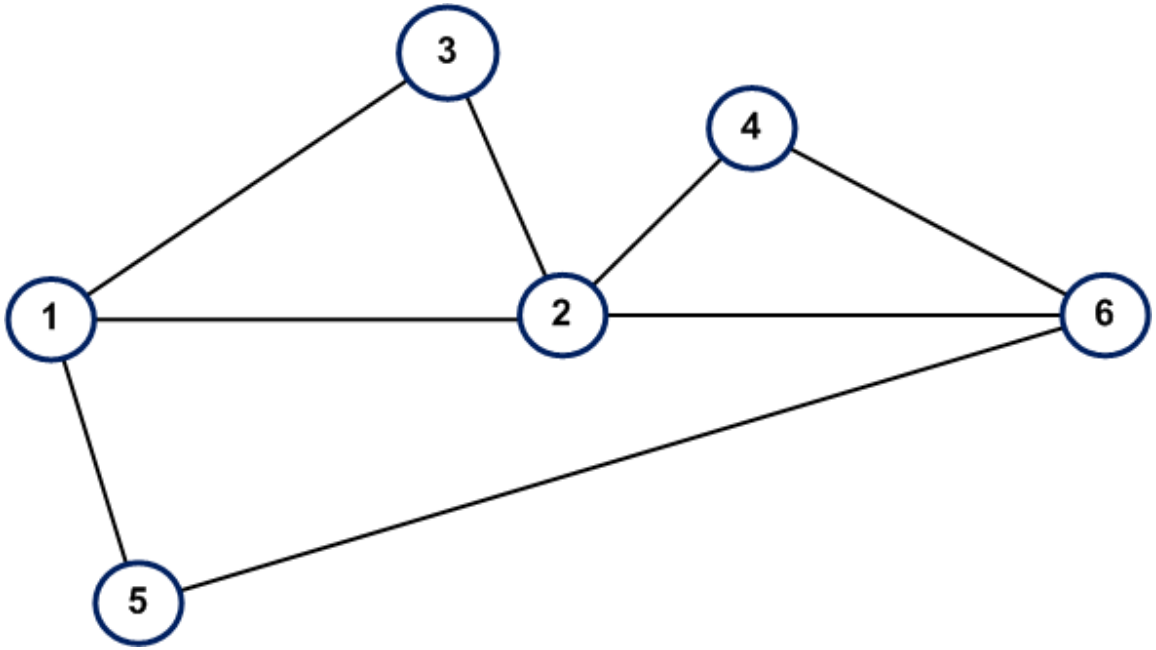


Reprezentarea grafurilor

Lista de adiacenţă

6 noduri:

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



Matrice de adiacenţă

- Construcţia 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 adiacenţă

- Construcţia din lista de muchii

```
void citire(int &n, int**&a, int orientat=0, const char*
nume_fisier="graf.in"){









}
```

Matrice de adiacenţă

- Construcţia din lista de muchii

```
void citire(int &n, int**&a, int orientat=0, const char*
nume_fisier="graf.in"){
        int i,x,y,j,m;
        ifstream f(nume_fisier);
        f>>n>>m;
        a=new int*[n];
        for(i=0;i<n;i++)
            a[i]=new int[n];
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                    a[i][j]=0;









        f.close();
}
```

Matrice de adiacenţă

- Construcţia din lista de muchii

```
void citire(int &n, int**&a, int orientat=0, const char*
nume_fisier="graf.in"){
        int i,x,y,j,m;
        ifstream f(nume_fisier);
        f>>n>>m;
        a=new int*[n];
        for(i=0;i<n;i++)
            a[i]=new int[n];
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                    a[i][j]=0;
        while(f>>x>>y){
                x--; y--;
                a[x][y]=1;
                if(not orientat)
                    a[y][x]=1;
        }
        f.close();
}
```

Matrice de adiacenţă

- Construcţia din lista de muchii

```
def citire(orientat=0,nume_fisier="graf.in"):
    n=0
    a=[]
    with open(nume_fisier) as f:
        linie=f.readline()
        n,m=(int(z) for z in linie.split())
        #aux=linie.split(); n=int(aux[0]); m=int(aux[1])
    return n,a
```

Matrice de adiacenţă

- Construcţia din lista de muchii

```
def citire(orientat=0,nume_fisier="graf.in"):
    n=0
    a=[]
    with open(nume_fisier) as f:
        linie=f.readline()
        n,m=(int(z) for z in linie.split())
        #aux=linie.split(); n=int(aux[0]); m=int(aux[1])
        a=[[0 for i in range(n)] for j in range(n)]
        #a=[[0]*n for i in range(n)]
        for linie in f: #linie=f.readline(); while linie!='':
            x,y=(int(z) for z in linie.split())
    return n,a
```

Matrice de adiacenţă

- Construcţia din lista de muchii

```
def citire(orientat=0,nume_fisier="graf.in"):
    n=0
    a=[]
    with open(nume_fisier) as f:
        linie=f.readline()
        n,m=(int(z) for z in linie.split())
        #aux=linie.split(); n=int(aux[0]); m=int(aux[1])
        a=[[0 for i in range(n)] for j in range(n)]
        #a=[[0]*n for i in range(n)]
        for linie in f: #linie=f.readline(); while linie!='':
            x,y=(int(z) for z in linie.split())
            #x,y=map(int,linie.split())
            x-=1; y-=1
            a[x][y]=1
            if not orientat:
                a[y][x]=1
                            #linie=f.readline()

    return n,a
```

Liste de adiacenţă

- Dinamic
  - folosind tipul vector / list

Liste de adiacenţă – Construcție din lista de muchii

```
void citire(int &n,vector<int> *&la, int orientat=0,
            const char *nume_fisier="graf.in"){

        int i,j,x,y,m;
        ifstream f(nume_fisier);
        f>>n>>m;

        la=new vector<int>[n];









        f.close();
}
```

Liste de adiacenţă – Construcție din lista de muchii

```
void citire(int &n,vector<int> *&la, int orientat=0,
            const char *nume_fisier="graf.in"){

        int i,j,x,y,m;
        ifstream f(nume_fisier);
        f>>n>>m;

        la=new vector<int>[n];

        while(f>>x>>y){ //mergea si cu for i=1,m
              x--;y--; //lucram de la 0

              la[x].push_back(y);
              if (not orientat)
                  la[y].push_back(x);
        }
        f.close();
}
```

Liste de adiacenţă – Construcție din lista de muchii

```
def citire(orientat=False,nume_fisier="graf.in"):
    n=0
    a=[]
    with open(nume_fisier) as f:
        linie = f.readline()
        n, m=(int(z) for z in linie.split())
        la=[[] for i in range(n)]
        #la=n*[[]] #!!!NU
    return n,la
```

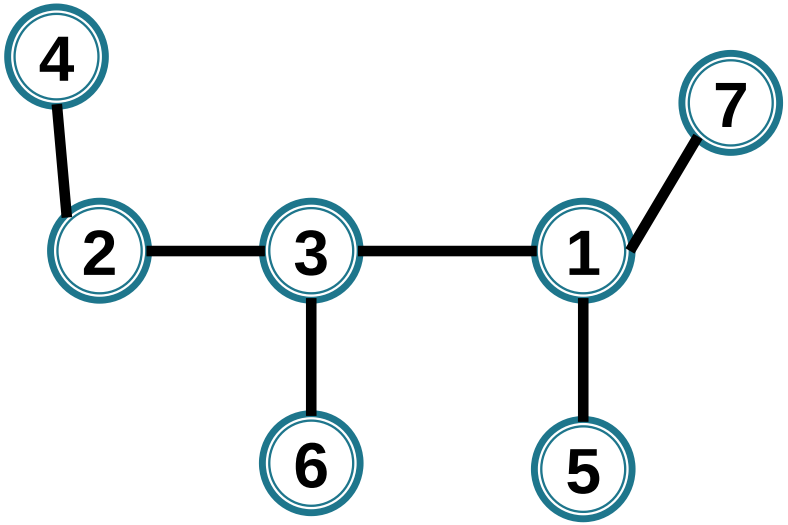
Liste de adiacenţă – Construcţie din lista de muchii

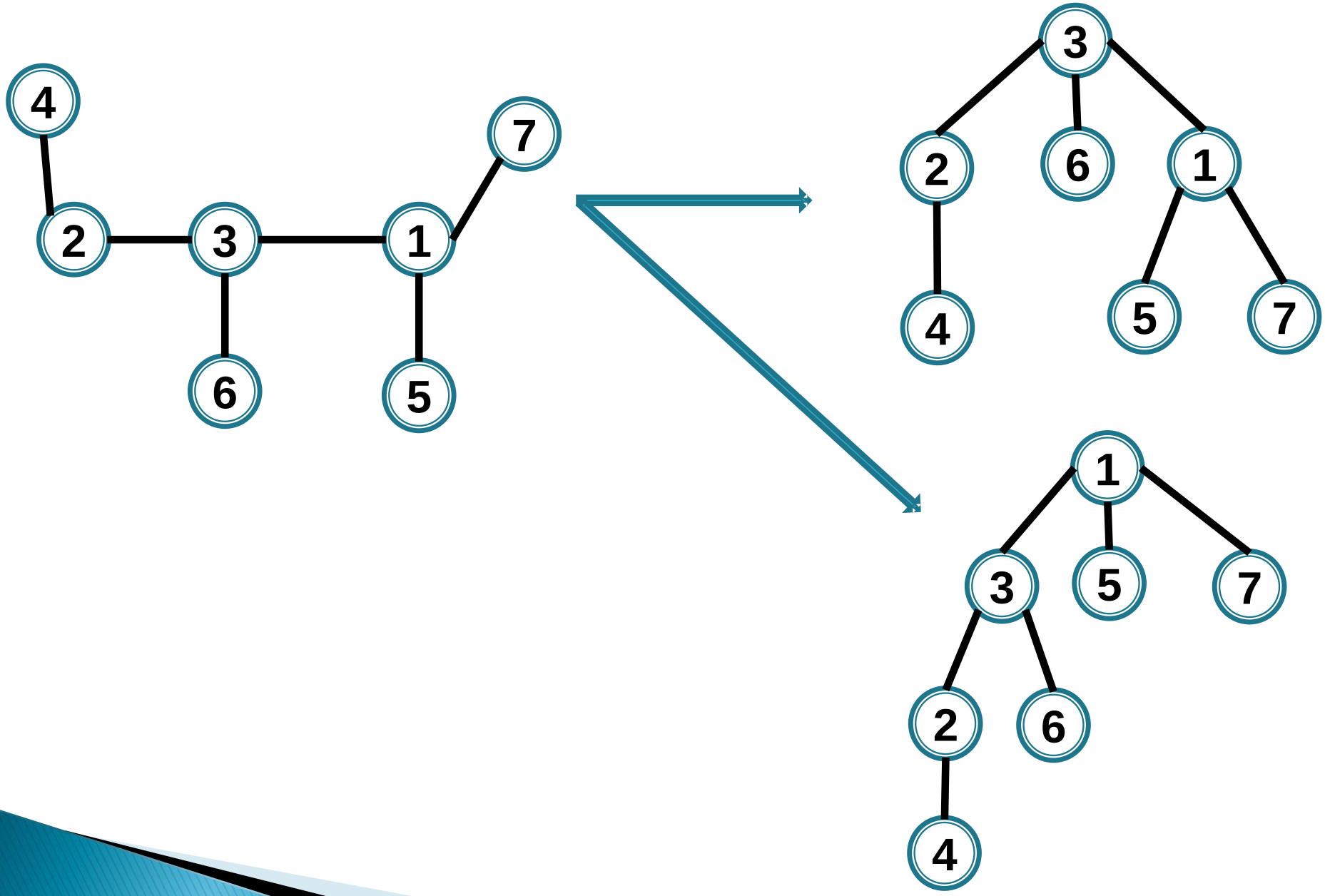
```
def citire(orientat=False,nume_fisier="graf.in"):
    n=0
    a=[]
    with open(nume_fisier) as f:

        linie = f.readline()
        n, m=(int(z) for z in linie.split())

        la=[[] for i in range(n)]
        #la=n*[[]] #!!!NU

        for linie in f:
            x,y=(int(z) for z in linie.split())
            x-=1; y-=1

            la[x].append(y)
            if not orientat:
                la[y].append(x)


    return n,la
```

Liste de adiacenţă – Varianta 2

- **Liste implementate** – pointeri

Arbori cu rădăcină


4
7
2
3
1
6
5


4
2
3
1
7
6
5
3
2
6
1
4
5
7
1
3
5
7
2
6
4

Noțiuni

Arbore cu rădăcină

- După fixarea unei rădăcini, arborele se aşează pe niveluri
- Nivelul unui nod v,

  `niv[v]` = distanţa de la rădăcină la nodul v

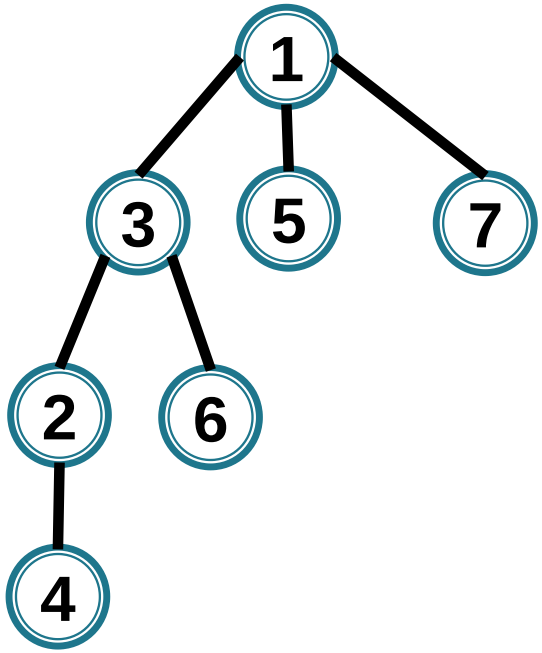
- În arborele cu rădăcină există muchii doar între niveluri consecutive

Noțiuni

- **Tată:** x este tată al lui y dacă există muchie de la x la y şi x se află în arbore pe un nivel cu 1 mai mic decât y
- **Fiu:** y este fiu al lui x $\Leftrightarrow$ x este tată al lui y
- **Ascendent:** x este ascendent a lui y dacă x aparţine unicului lanţ elementar de la y la rădăcină (echivalent, dacă există un lanţ de la y la x care trece prin noduri situate pe niveluri din ce în ce mai mici)
- **Descendent:** y este descendent al lui x $\Leftrightarrow$ x este ascendent a lui y
- **Frunză:** nod fără fii

Noțiuni

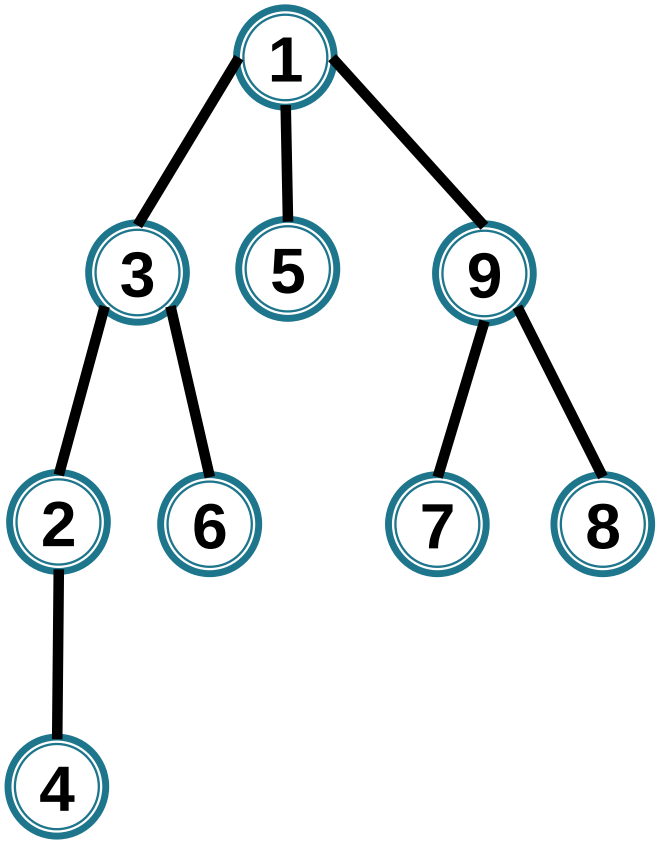
- **Fiu:** fii lui 3 sunt 2 şi 6
- **Tată:** 1 este tatăl lui 7
- **Ascendent:** ascendenţii lui 6 sunt 3 şi 1
- **Descendent:** descendenţii lui 3 sunt 2, 6 şi 4
- **Frunză:** frunzele arborelui sunt 4, 6, 5 şi 7



Modalităţi de reprezentare a arborilor cu rădăcină

Reprezentarea arborilor

- Vector tata
- Lista de fii



Vectorul tata

Folosind vectorul tata putem determina **lanțuri de la orice vârf x la rădăcină**, **urcând** în arbore de la x la rădăcină

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

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

Parcurgerea Grafurilor

Parcurgerea grafurilor

Dat un graf G și un vârf s, care sunt 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

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

Parcurgerea grafurilor

Idee: Dacă

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

atunci v este accesibil din s.

Parcurgerea grafurilor

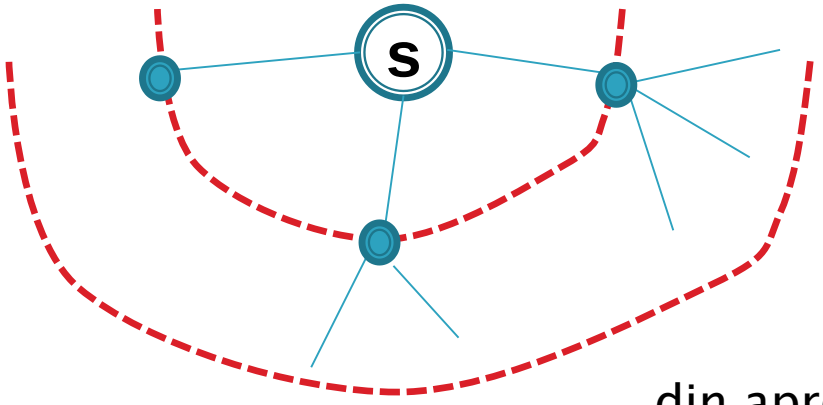
- Parcurgerea în lățime (BF = breadth first)
- Parcurgerea în adâncime (DF = depth first)

Parcurgerea în lăţime

Parcurgerea grafurilor

- **Parcurgerea în lățime:** se vizitează
  - vârful de start **s**
  - vecinii acestuia
  - vecinii nevizitați ai acestora

etc

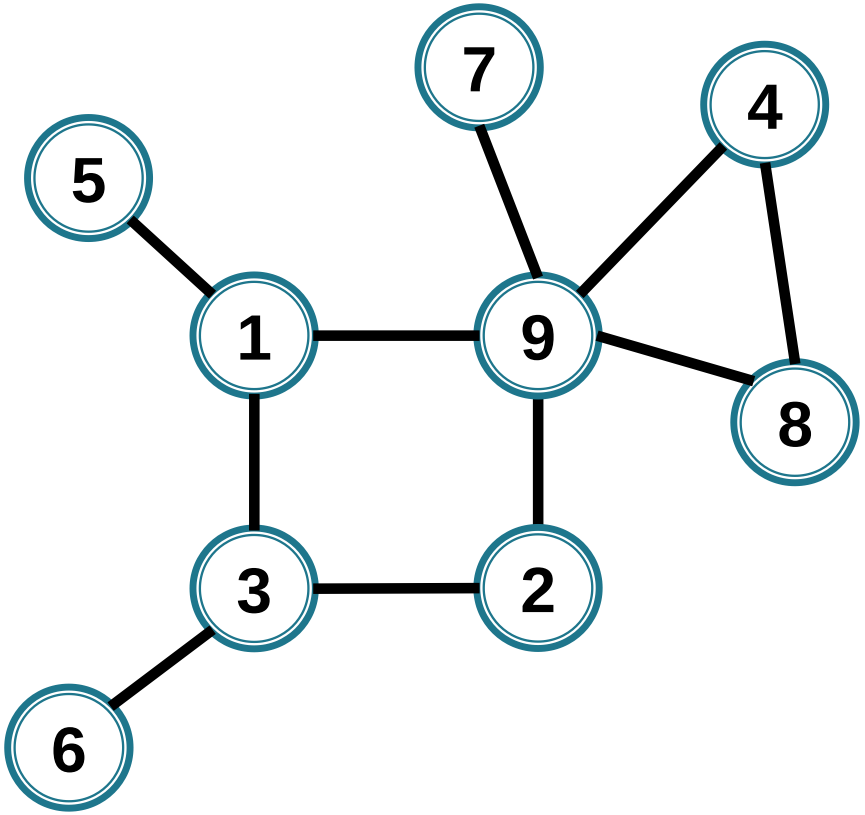


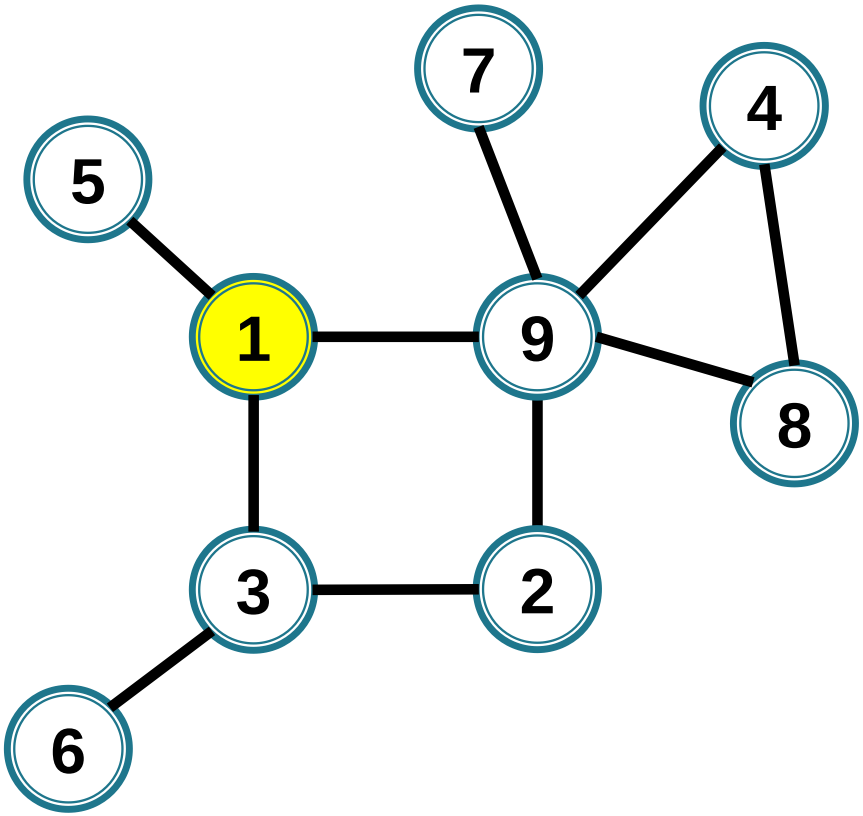
din aproape în aproape

Parcurgerea în lățime

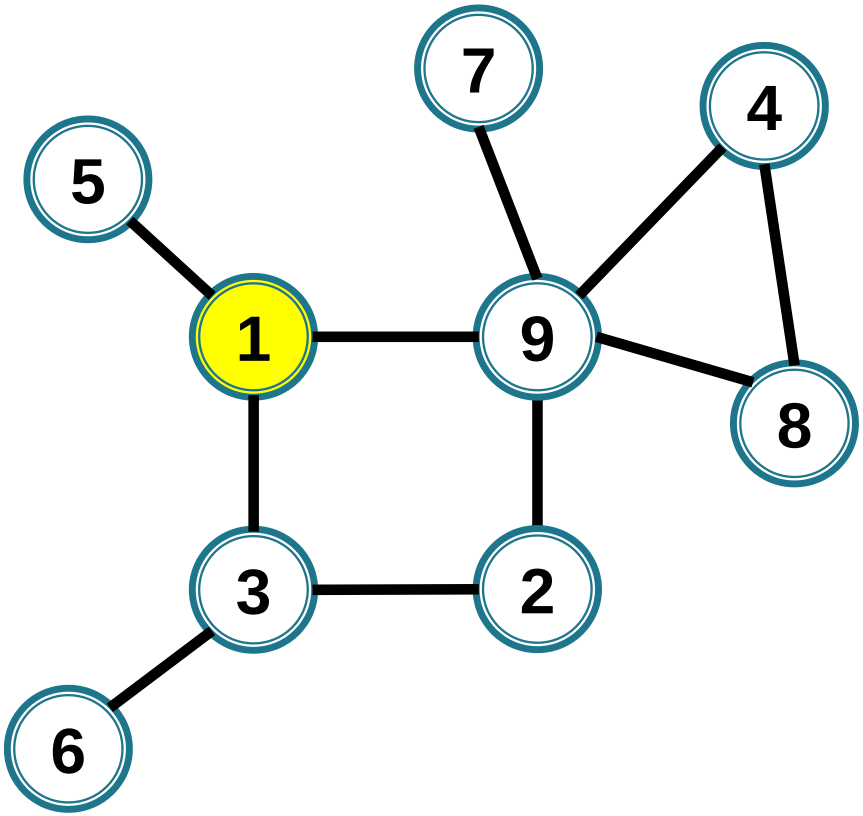
- Pentru gestionarea vârfurilor parcurse care mai pot avea vecini nevizitaţi – o structură de tip **coadă**

Exemplu
pentru graf neorientat

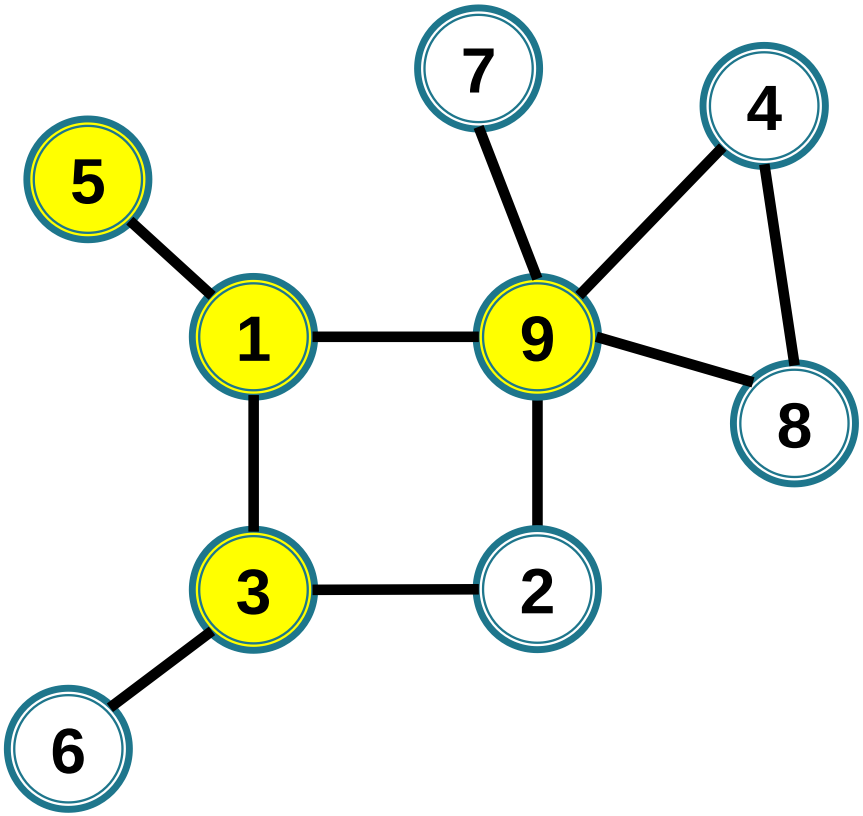

7
4
5
1
9
8
3
2
6

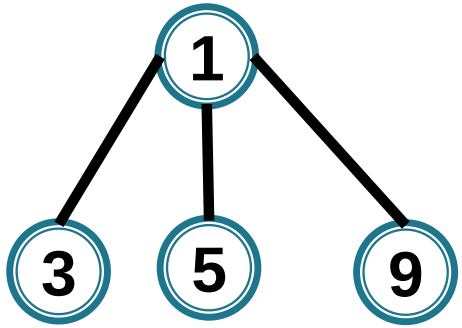


1

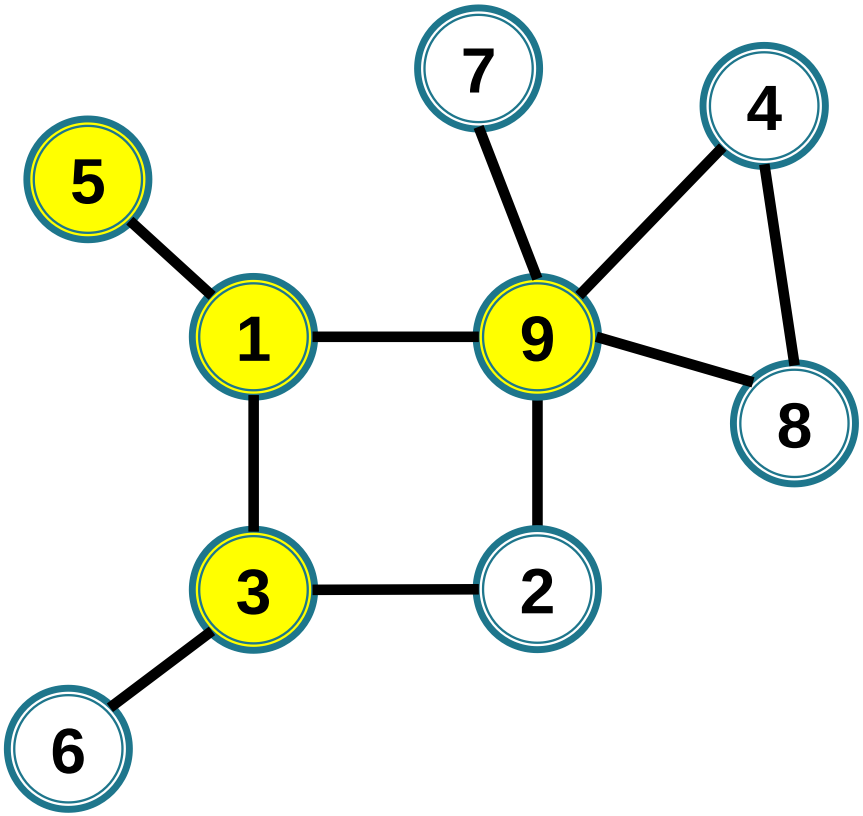


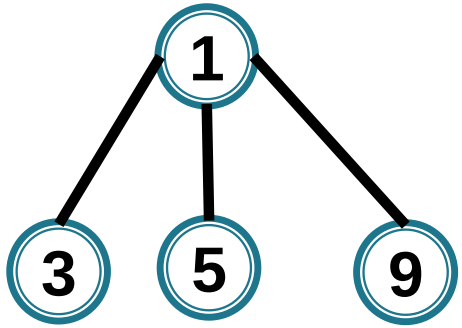
1



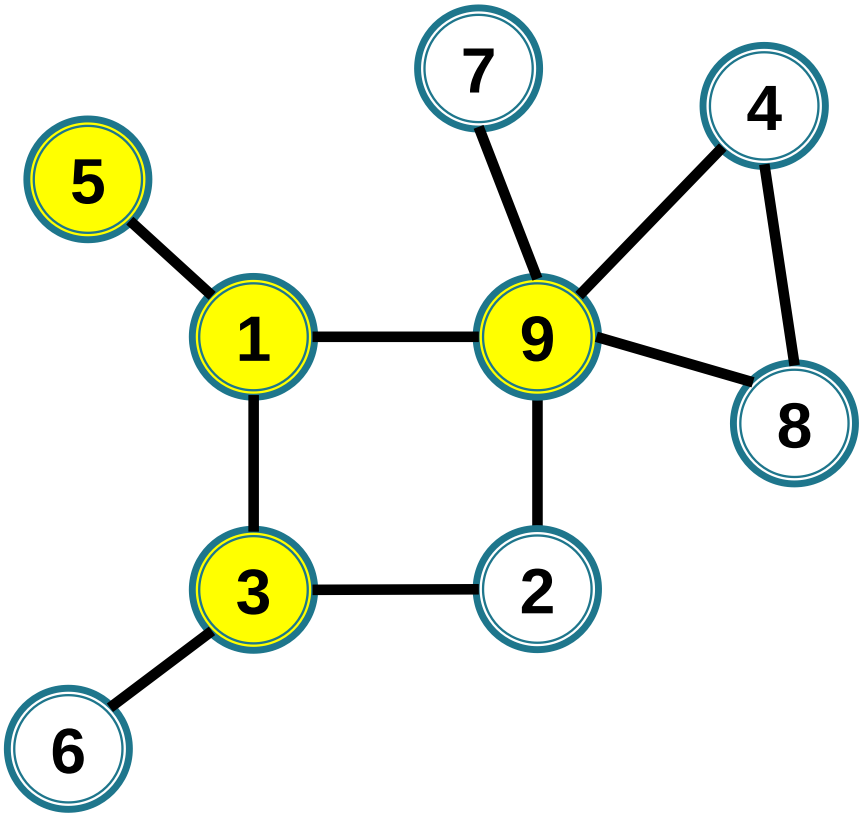


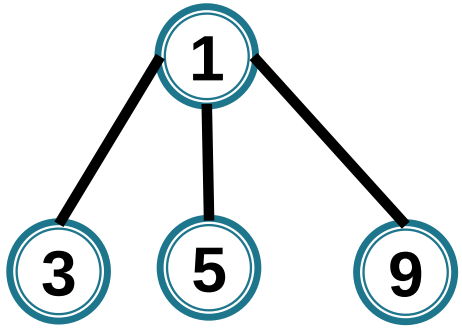
1 3 5 9



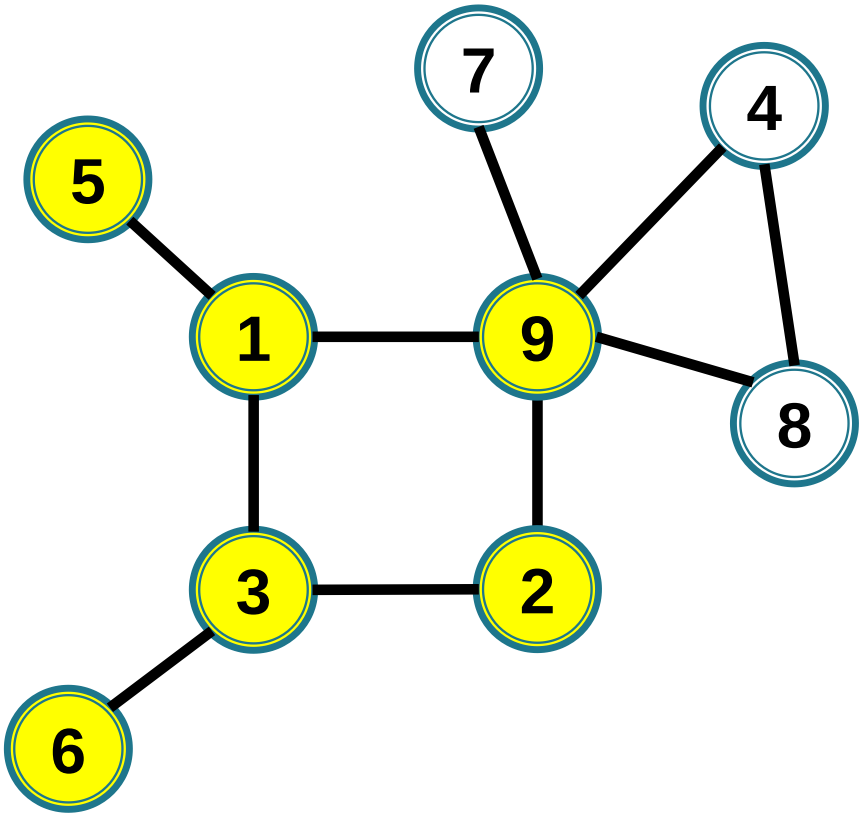


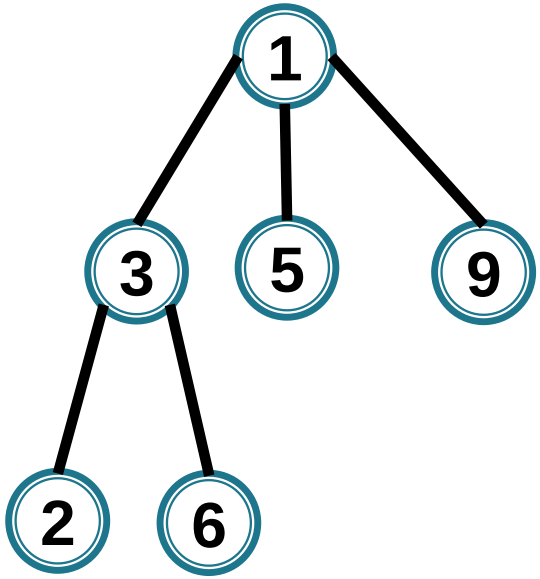
1 3 5 9



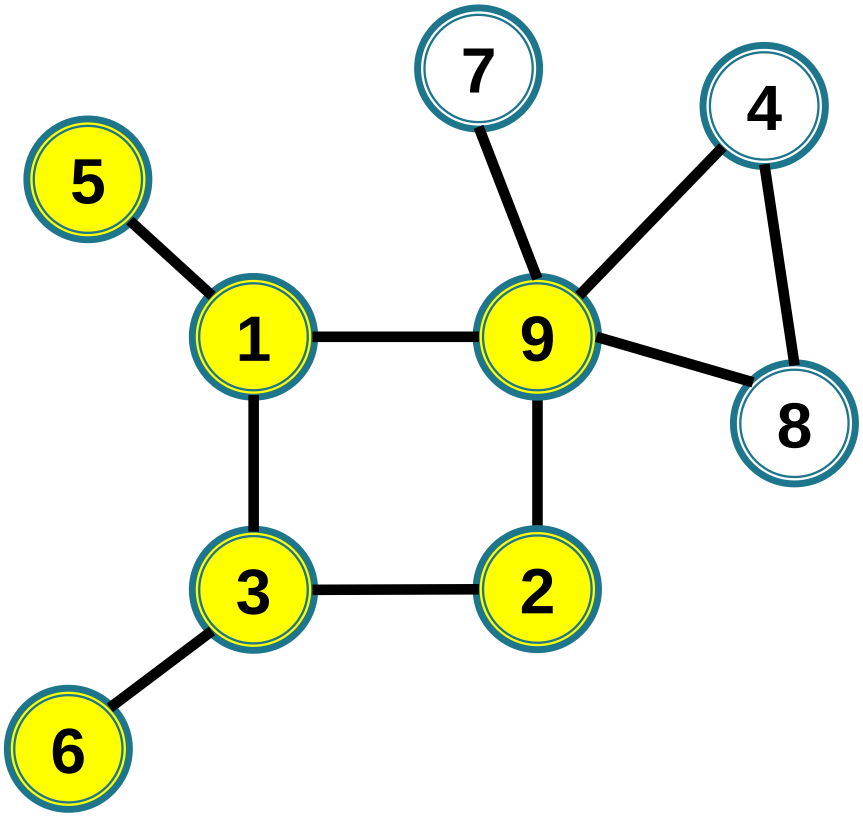


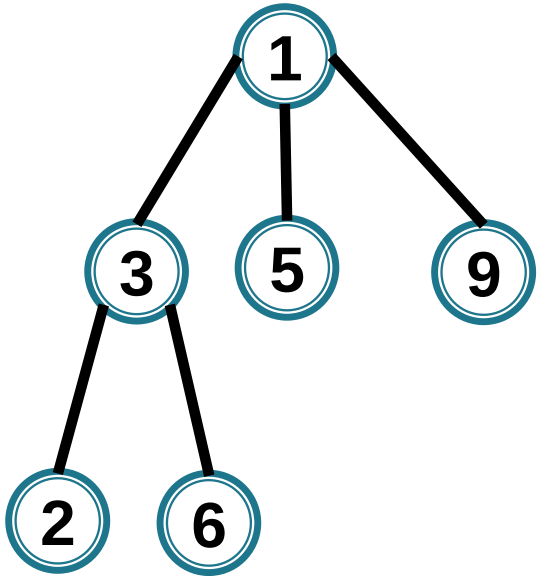
1 3 5 9



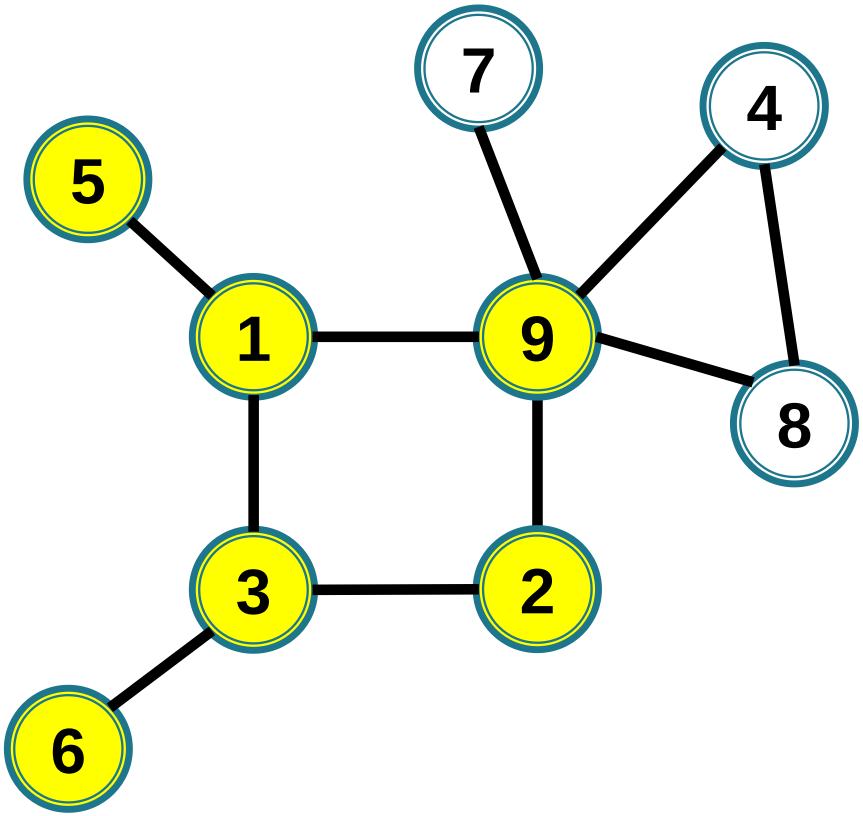


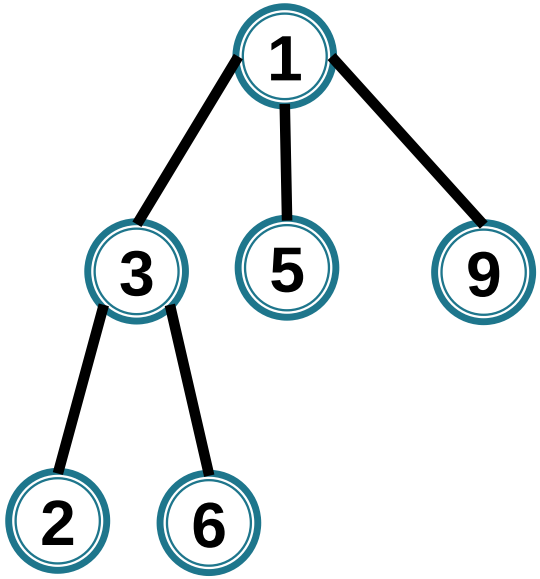
1 3 5 9 2 6



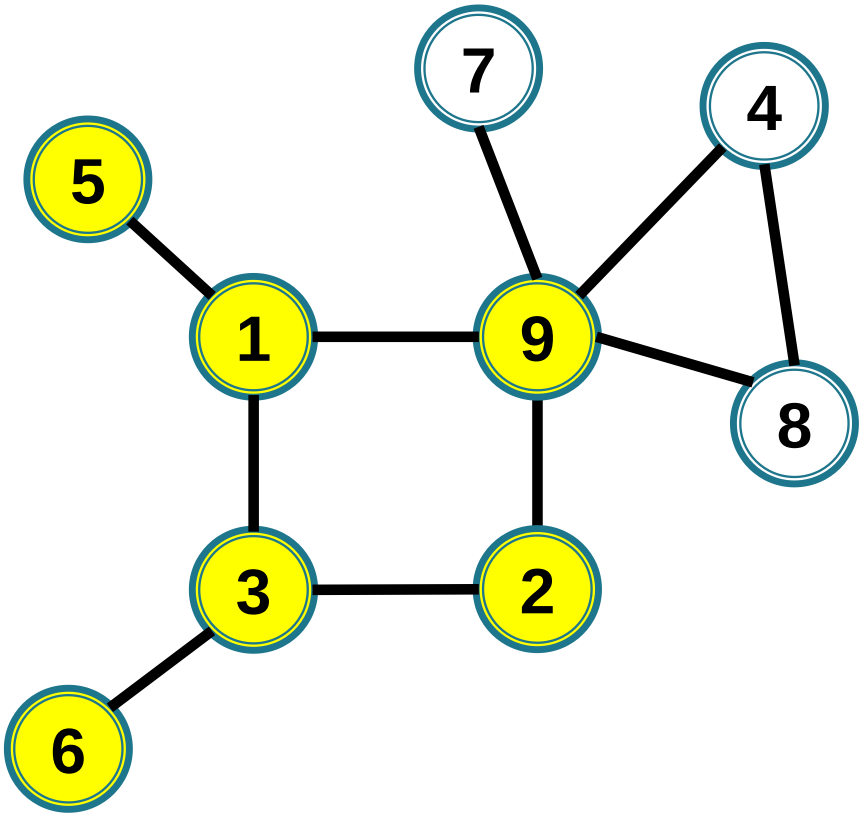


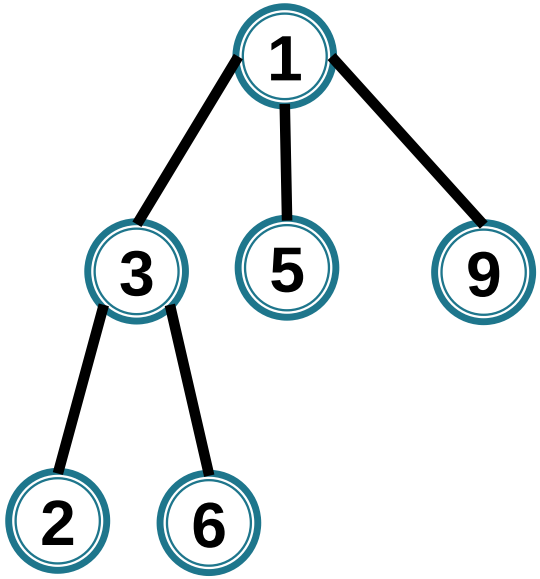
1 3 5 9 2 6



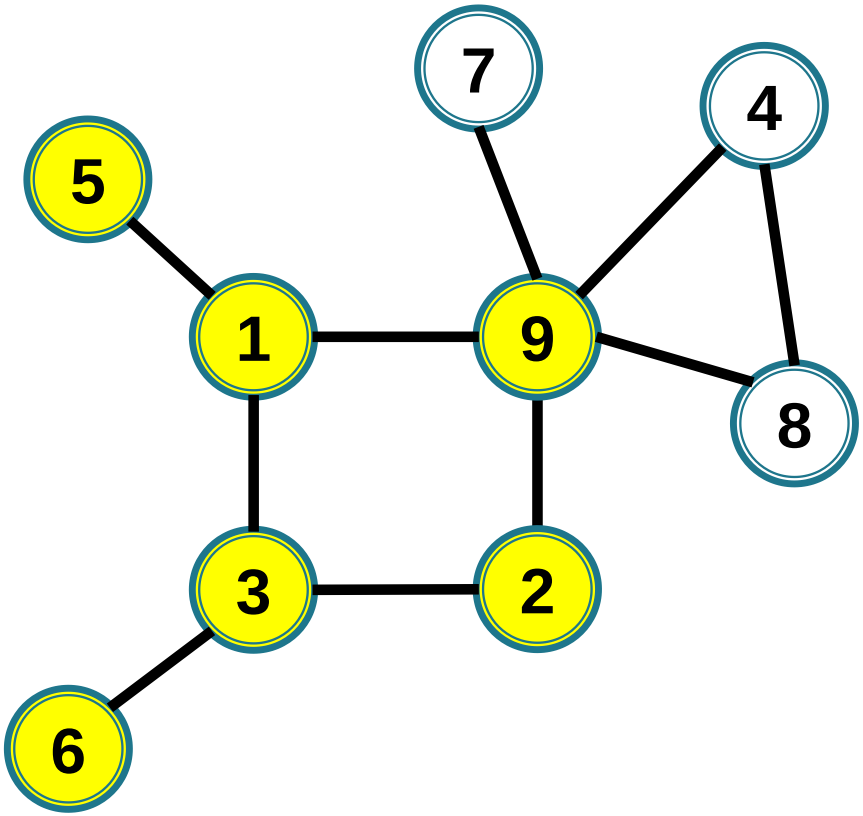


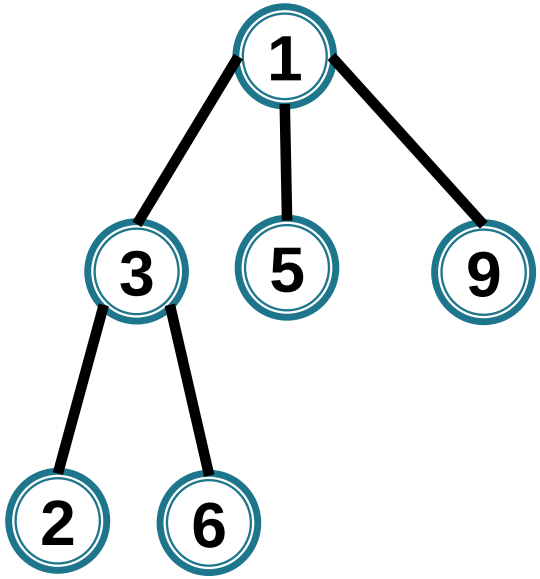
1 3 5 9 2 6



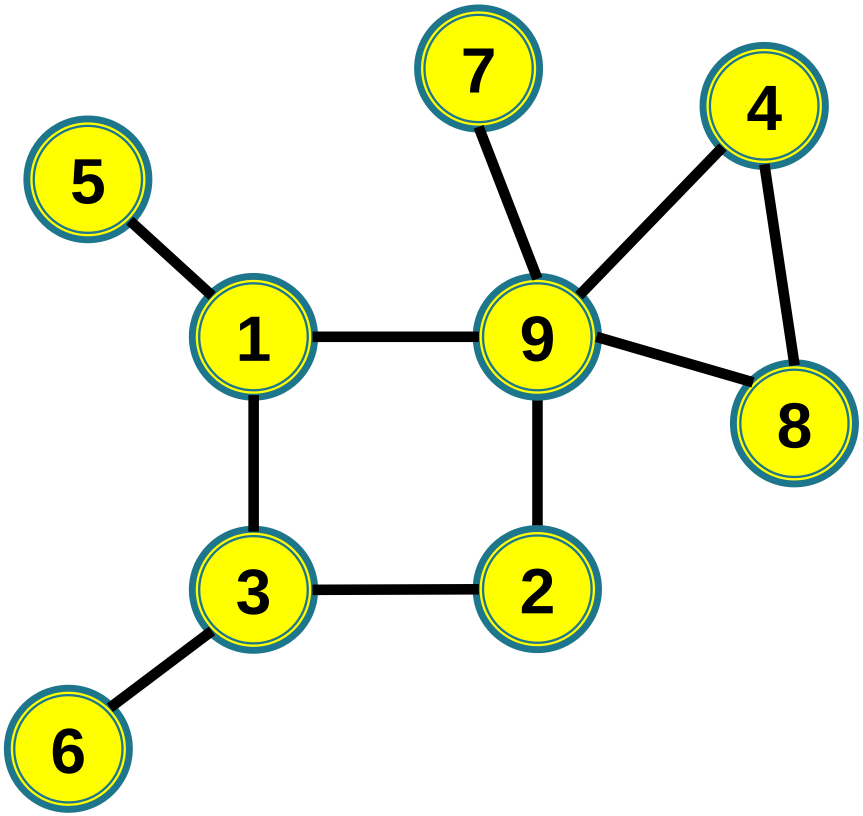


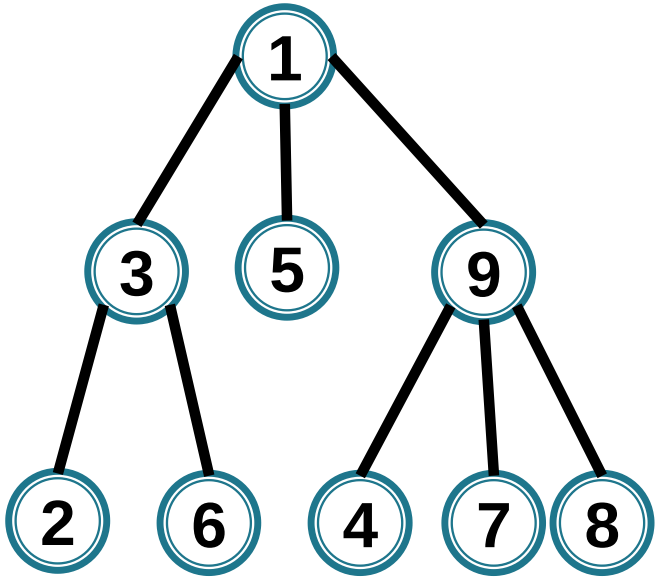
1 3 5 9 2 6



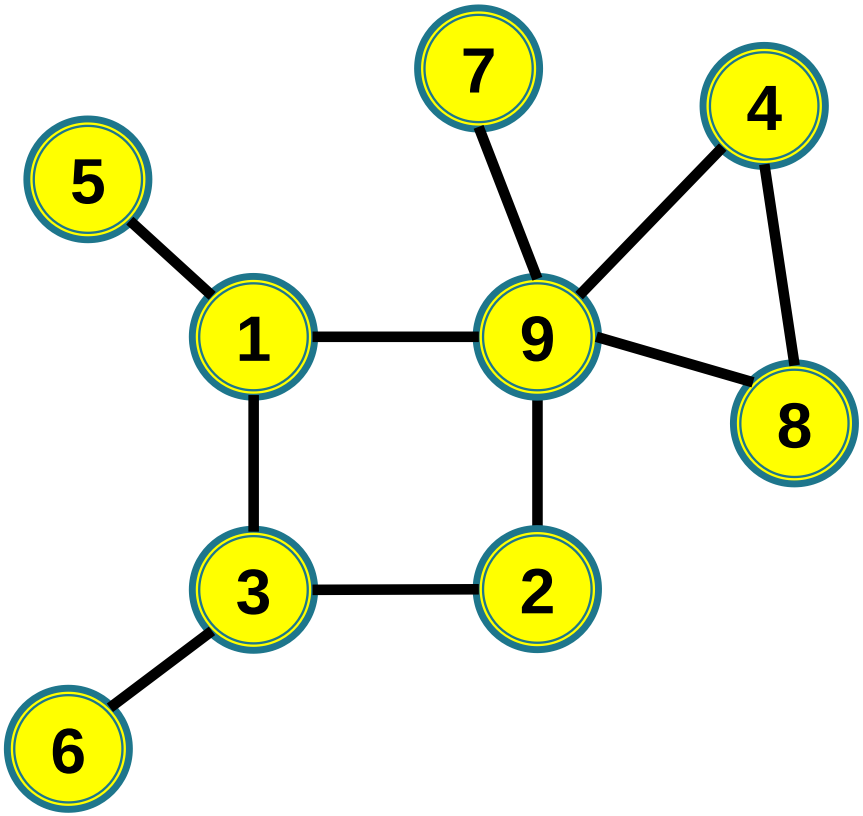


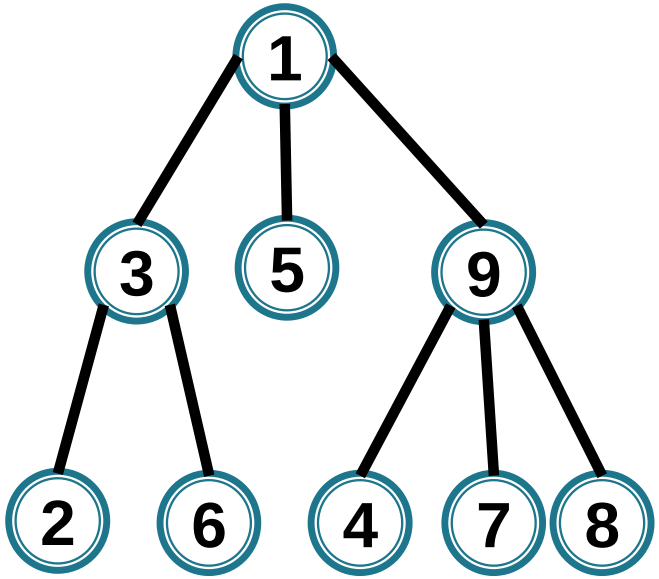
1 3 5 9 2 6



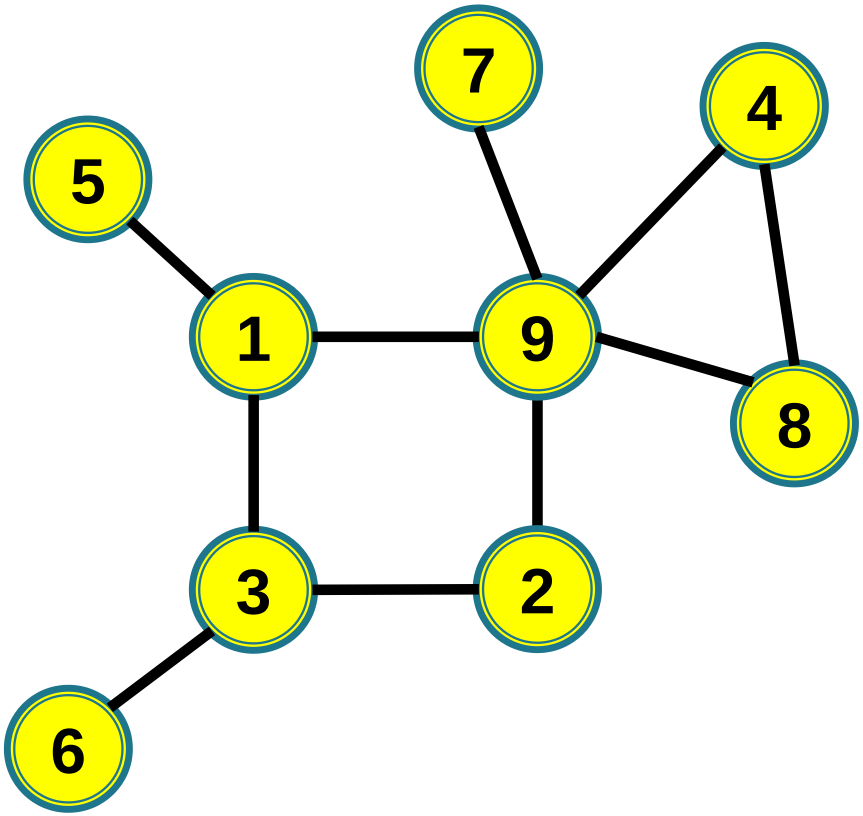


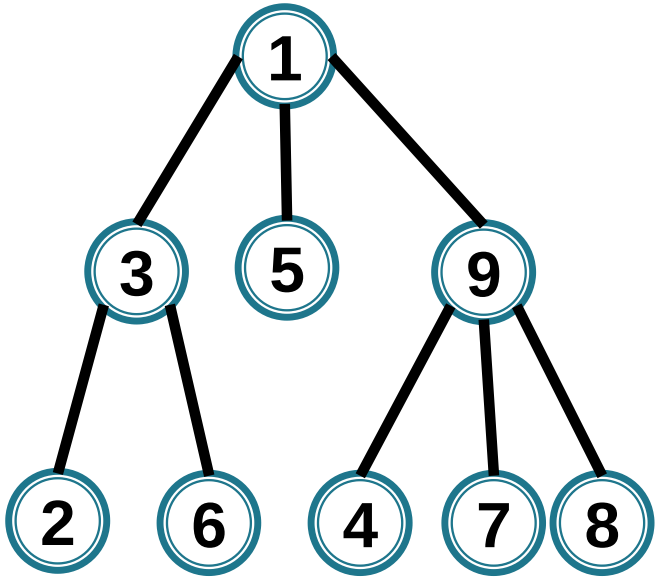
1 3 5 9 2 6 4 7 8



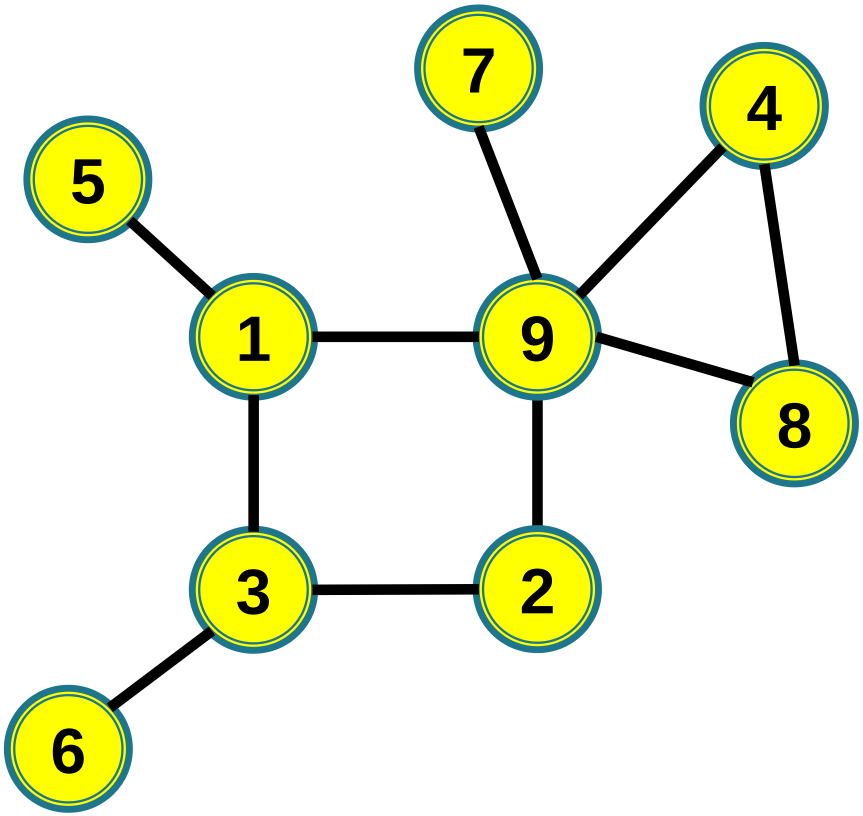


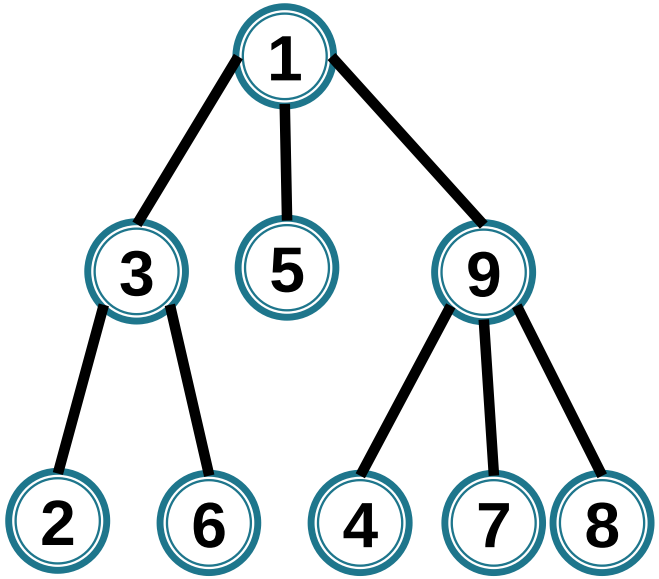
1 3 5 9 2 6 4 7 8



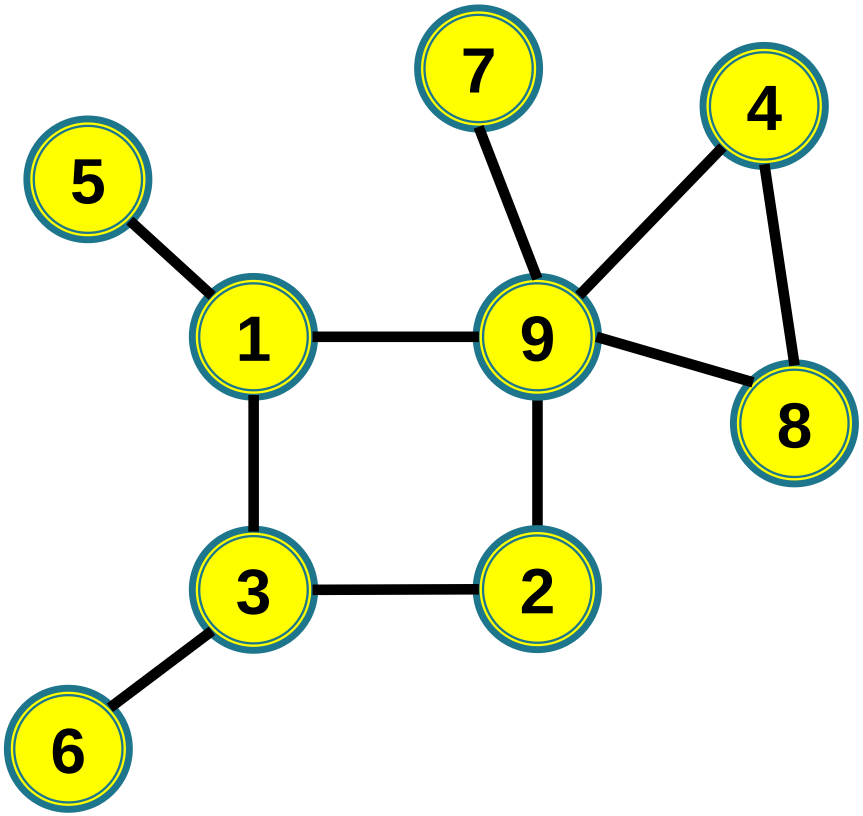


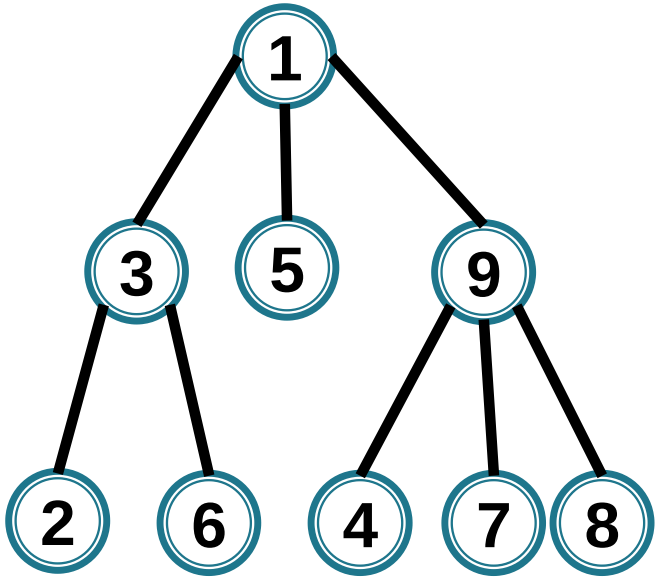
1 3 5 9 2 6 4 7 8



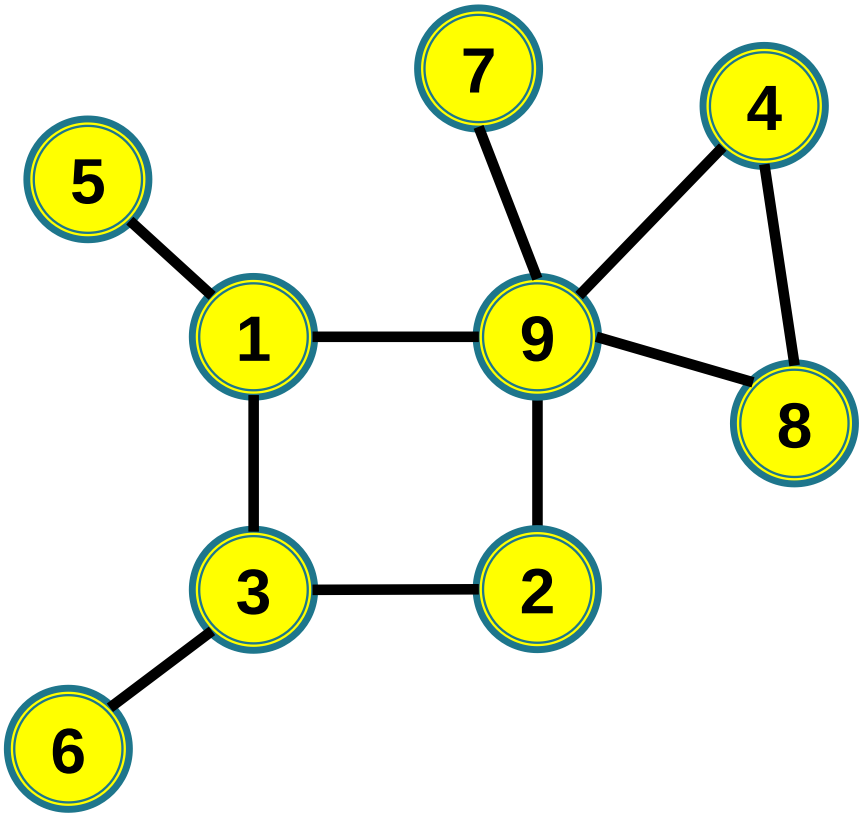


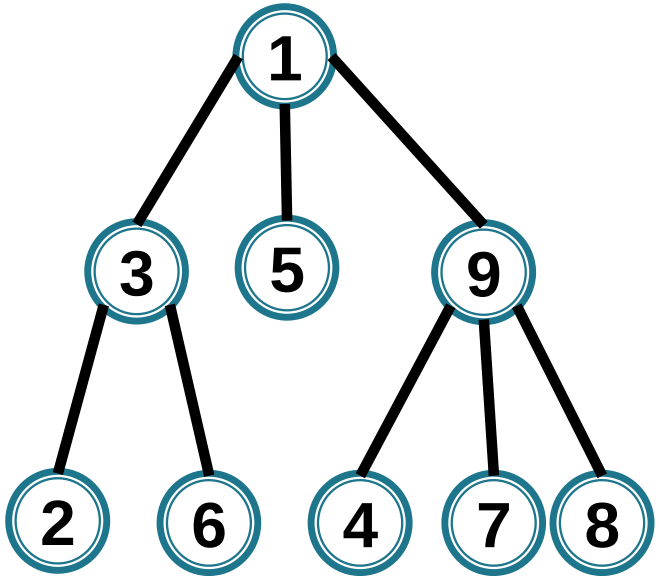
1 3 5 9 2 6 4 7 8



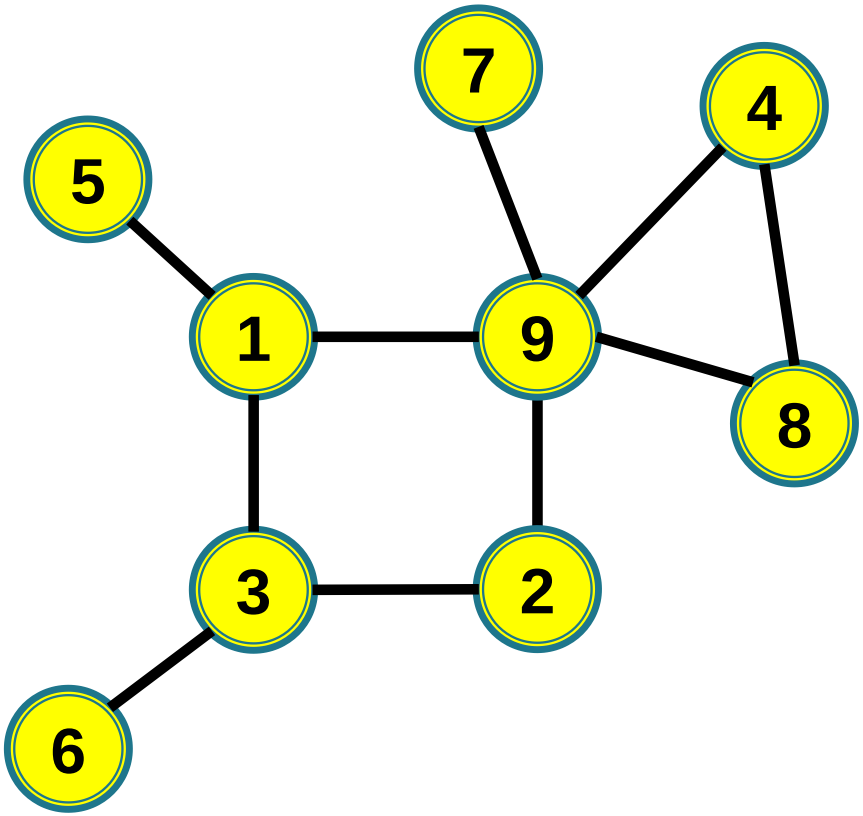


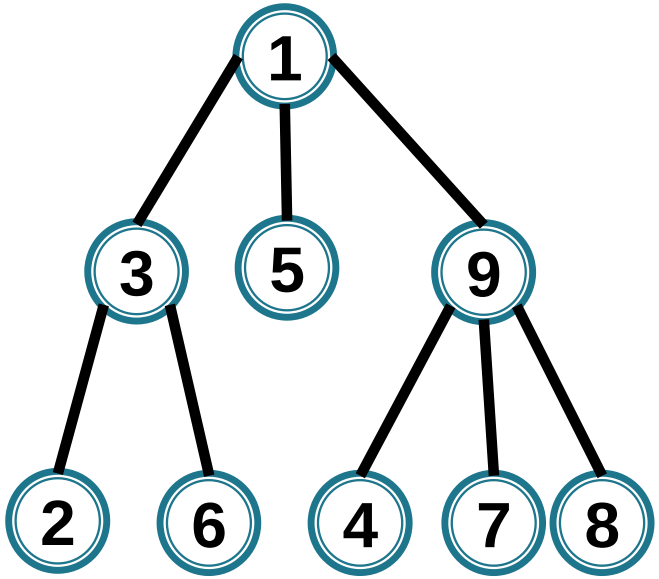
1 3 5 9 2 6 4 7 8



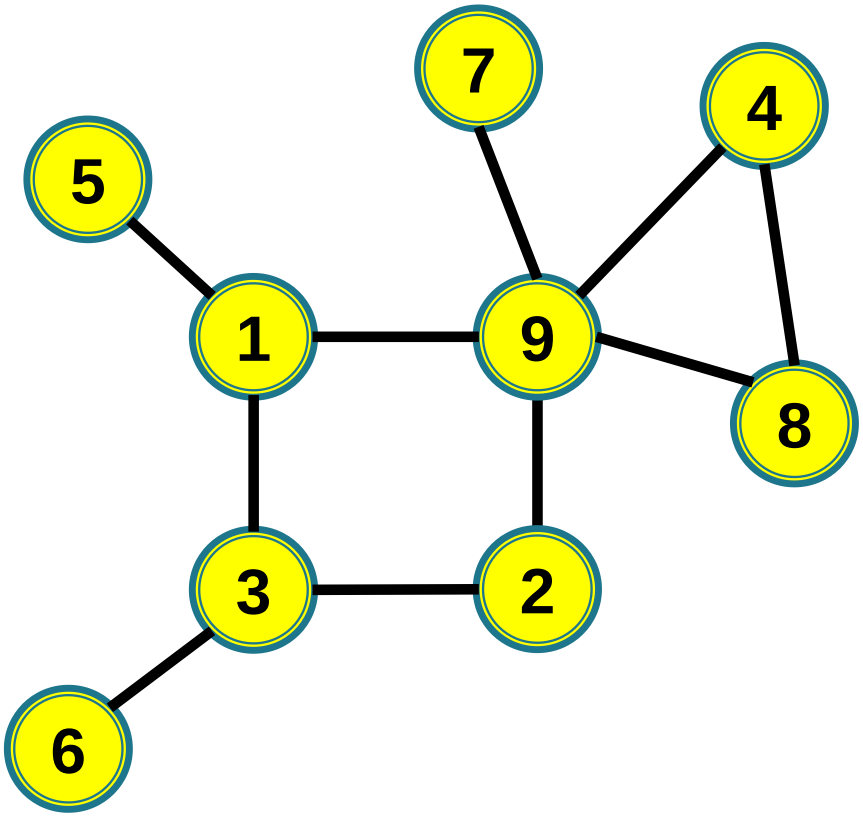


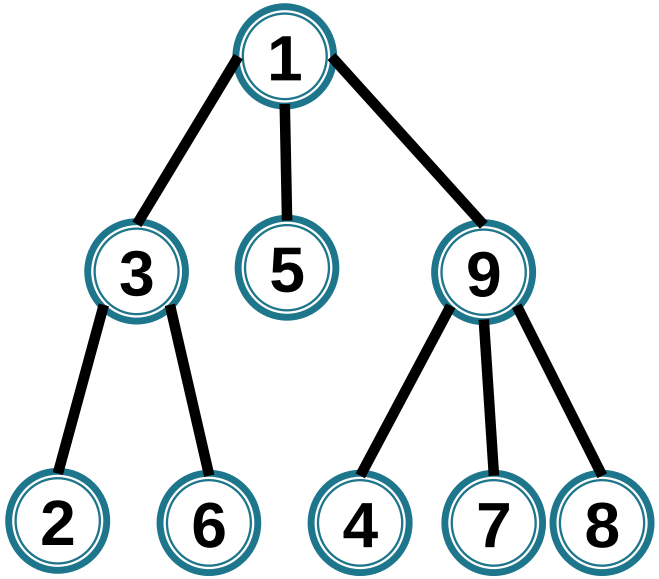
1 3 5 9 2 6 4 7 8



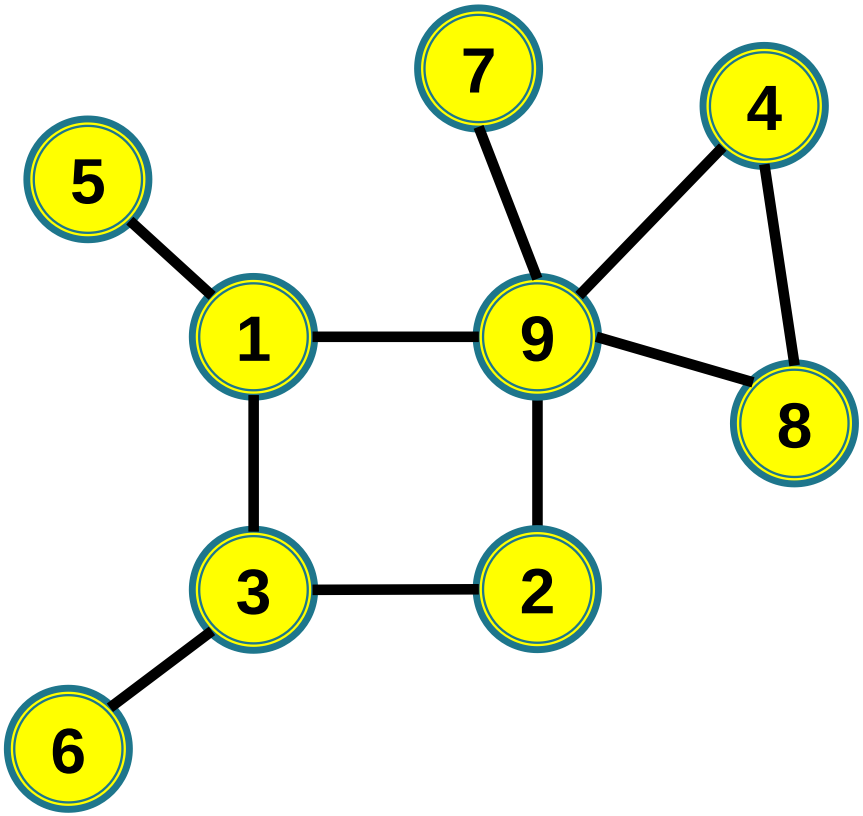


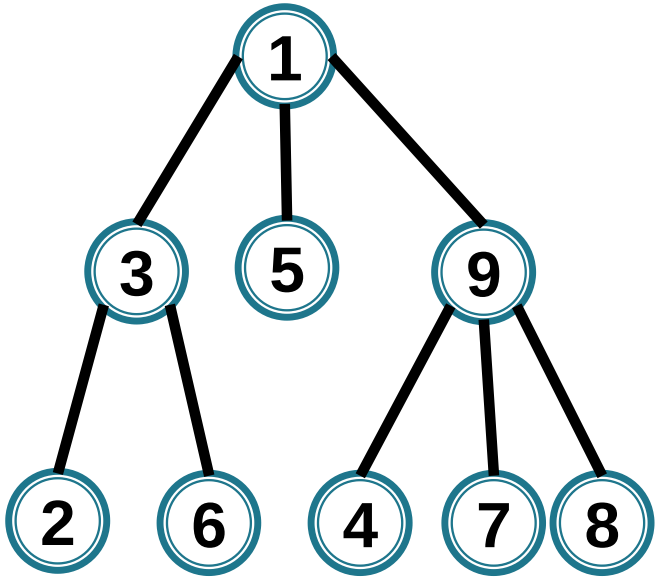
1 3 5 9 2 6 4 7 8



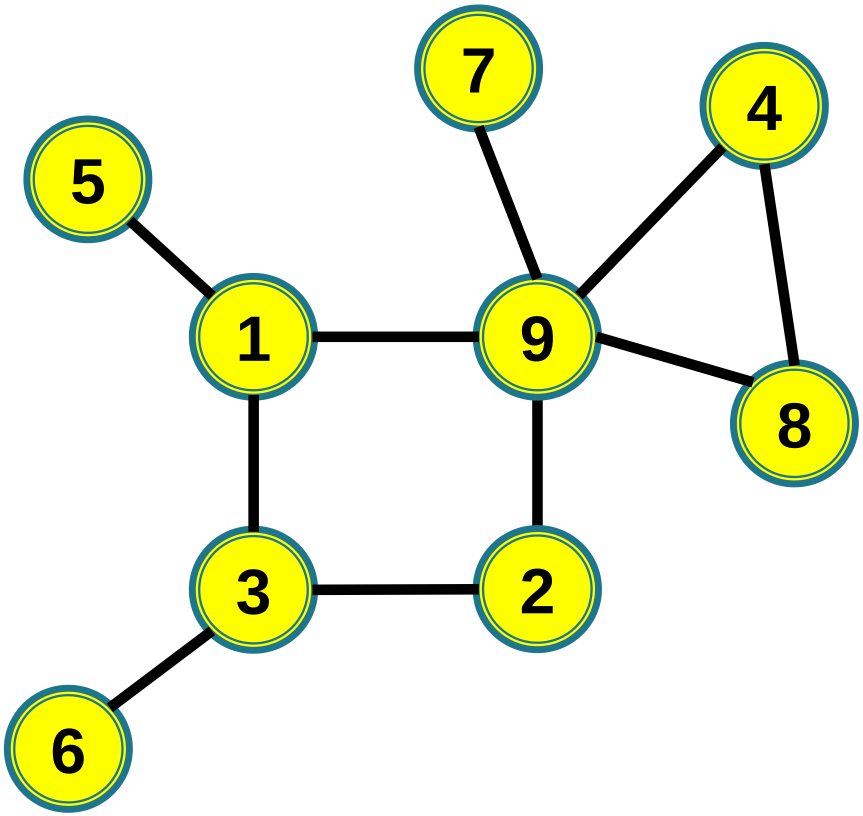


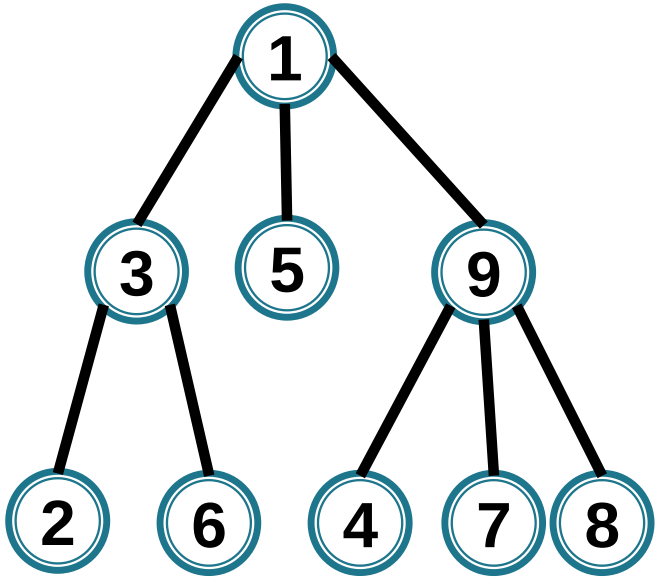
1 3 5 9 2 6 4 7 8



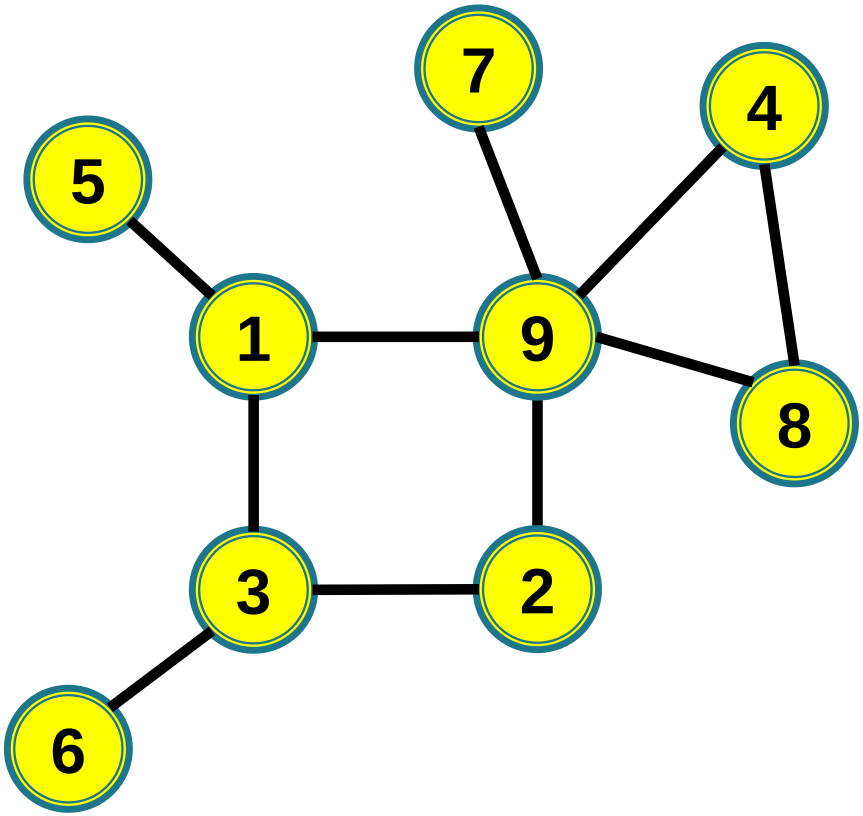


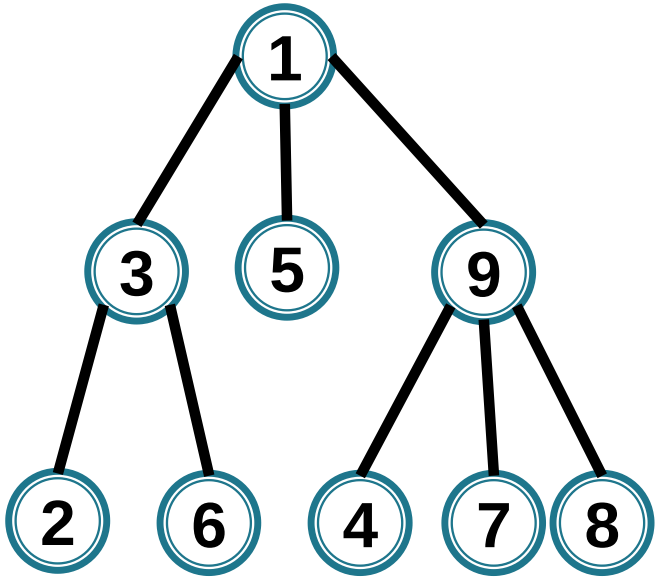
1 3 5 9 2 6 4 7 8



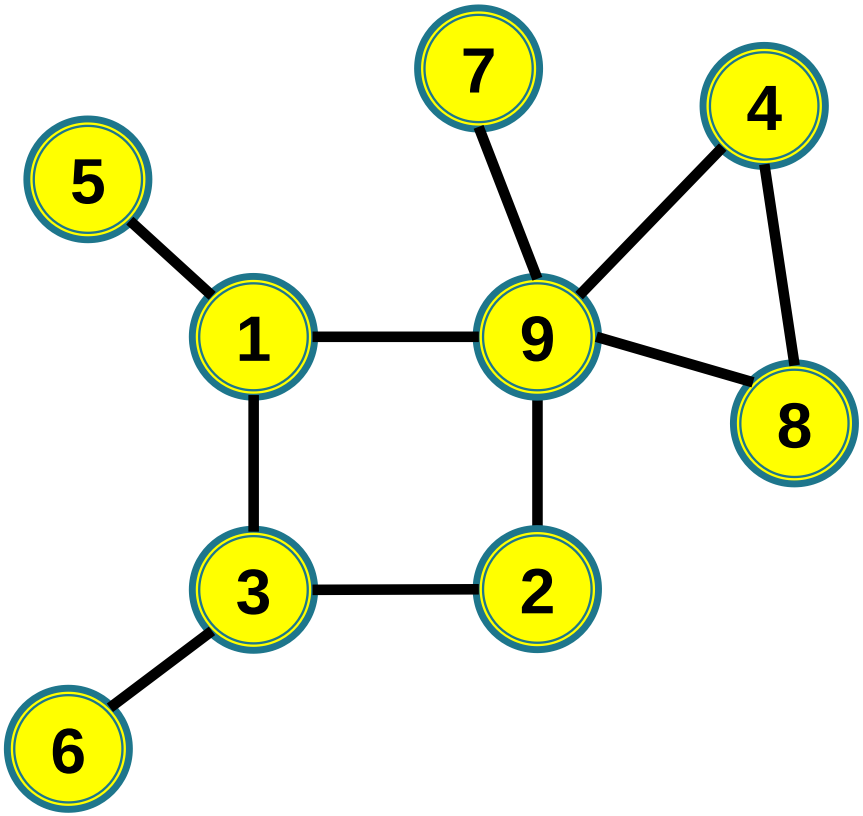


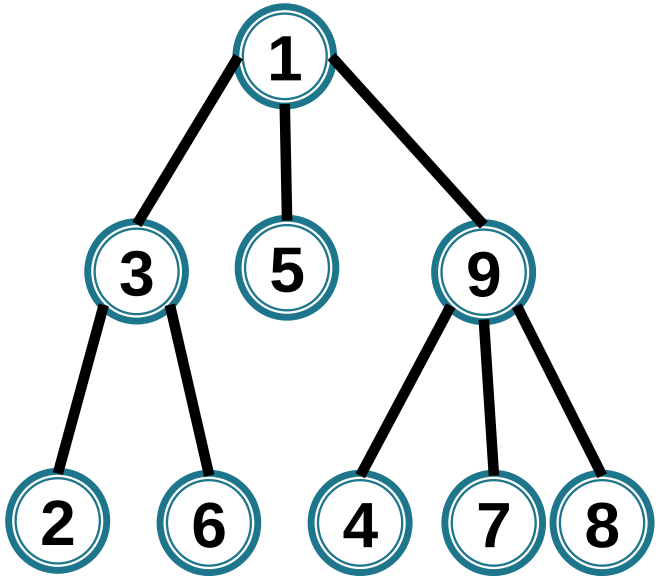
1 3 5 9 2 6 4 7 8





1 3 5 9 2 6 4 7 8



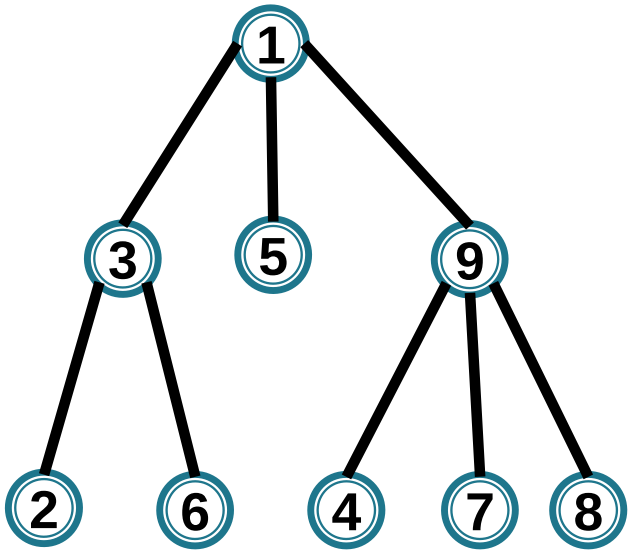


1 3 5 9 2 6 4 7 8

Parcurgerea în lățime – graf neorientat

- Muchiile folosite pentru a descoperi vârfuri noi formează un **arbore** (numit **arbore BF)**
- **Arborele se memorează în BF cu vector tata**

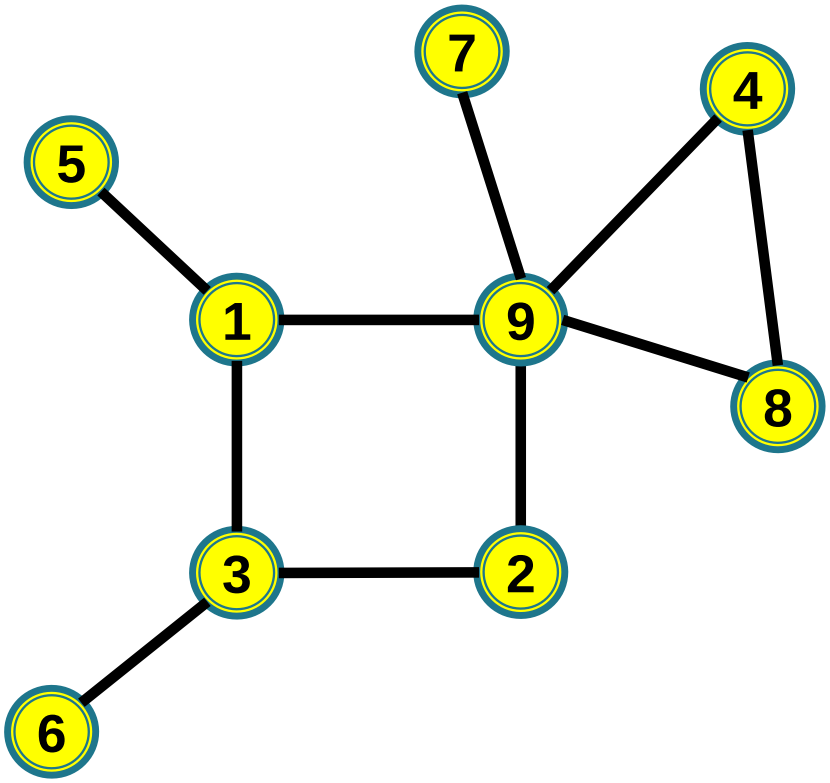
  tata[v] = vârful din care v a fost descoperit (vizitat)

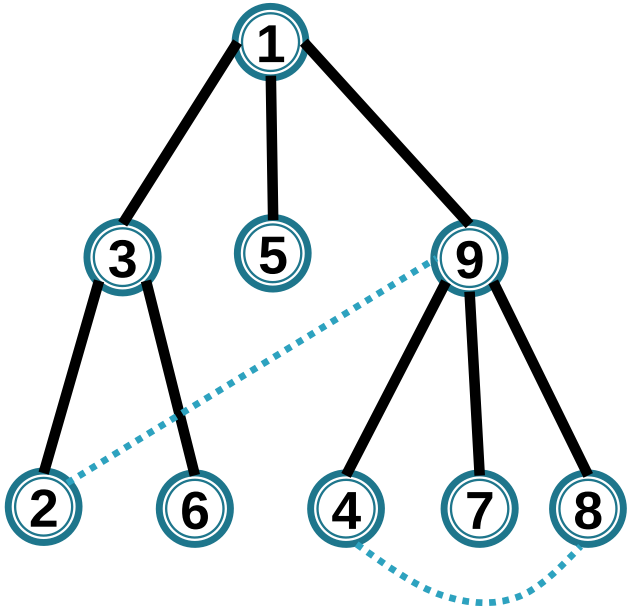


tata = [0, 3, 1, 9, 1, 3, 9, 9, 1]

Parcurgerea în lățime – graf neorientat

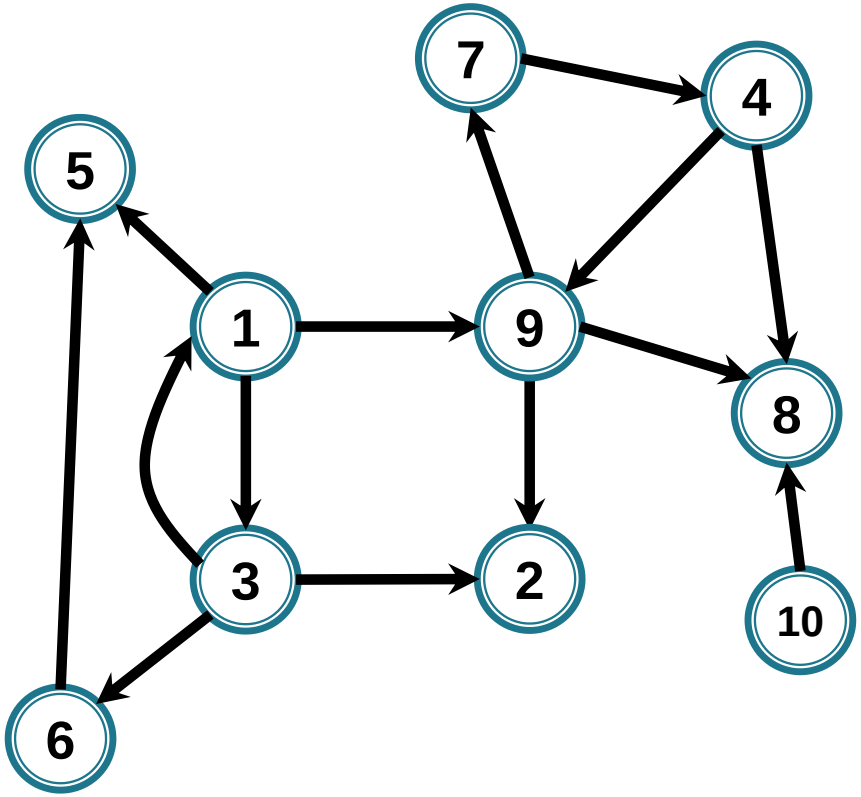
- Muchiile folosite pentru a descoperi vârfuri noi formează un **arbore** (numit **arbore BF)**
- **Muchiile din graf care nu sunt în arbore închid cicluri (cu muchiile din arbore)**





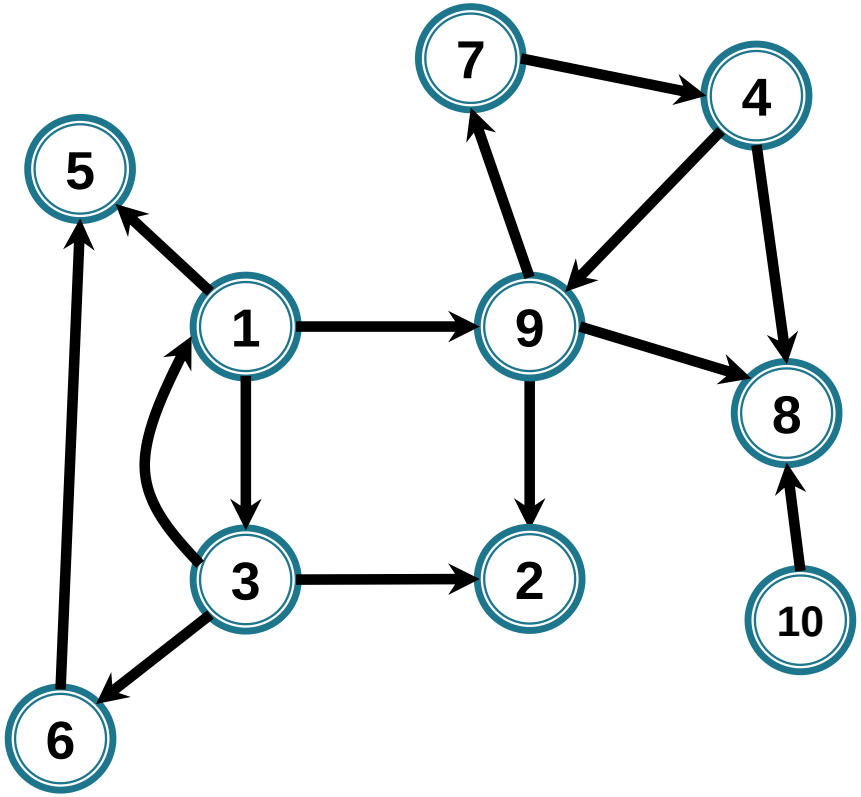
Parcurgerea în lățime

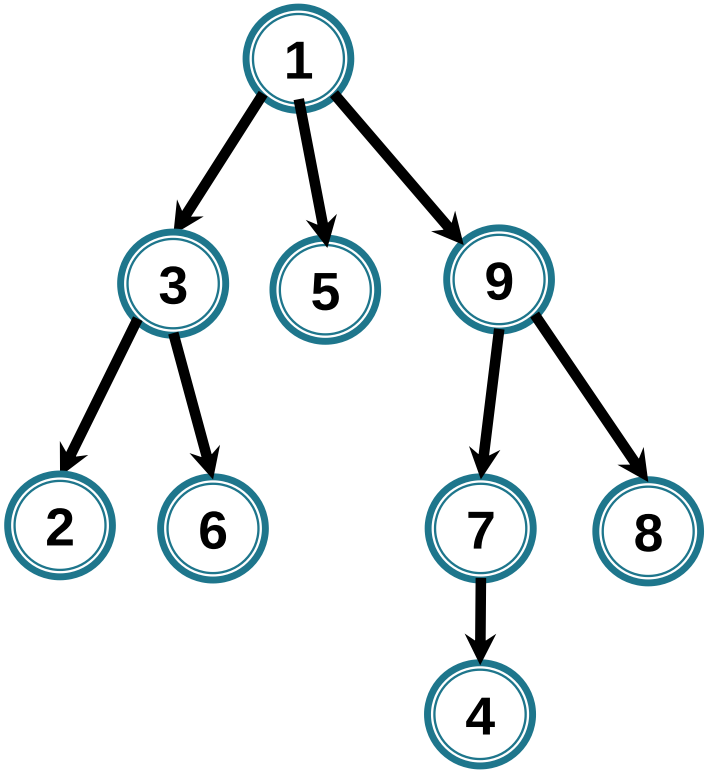
- Exemplu – caz orientat:



Parcurgerea în lățime

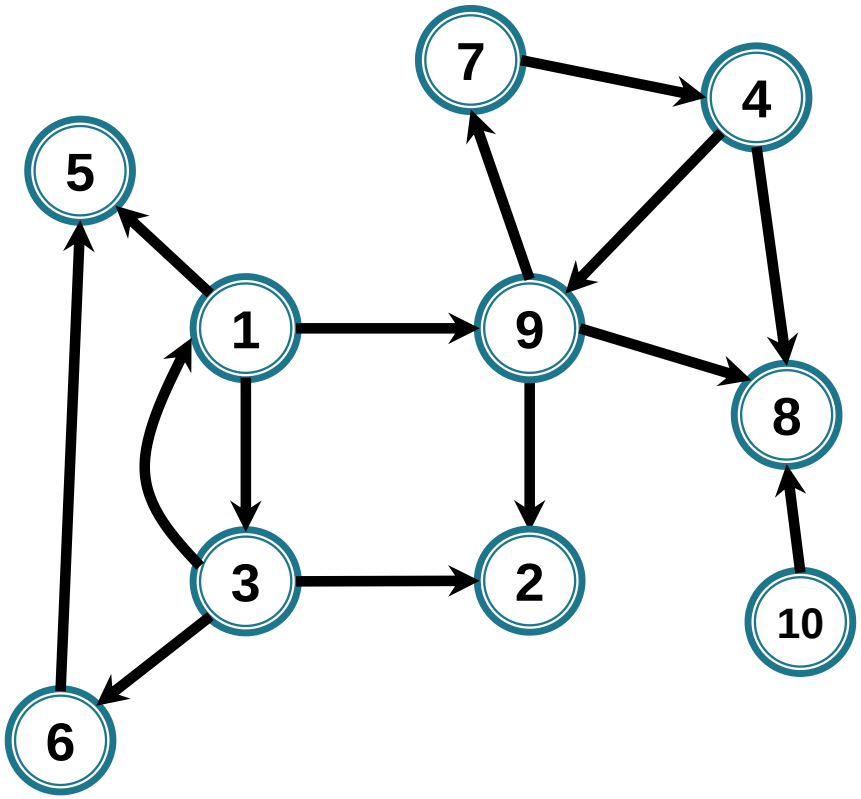
- Exemplu – caz orientat:

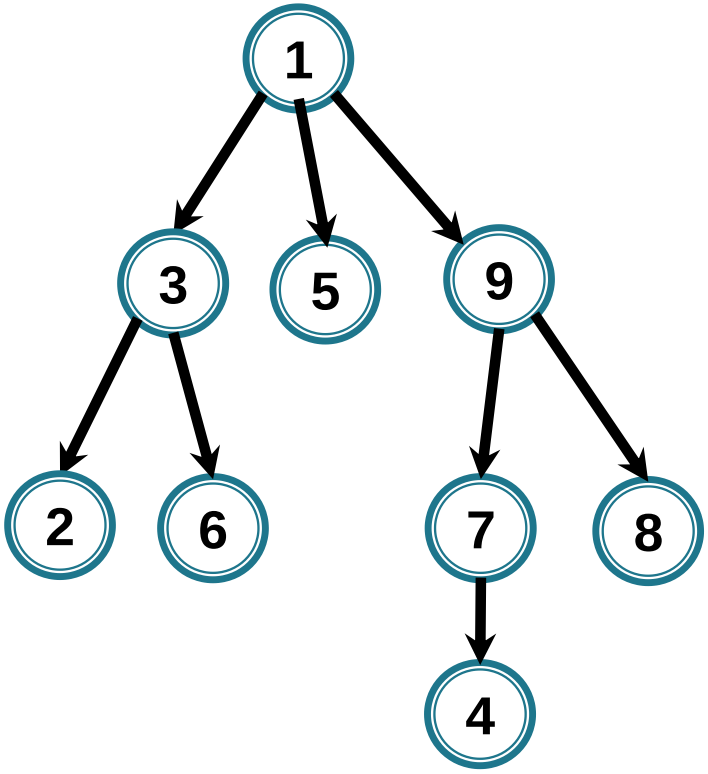




BF(1): 1, 3, 5, 9, 2, 6, 7, 8, 4

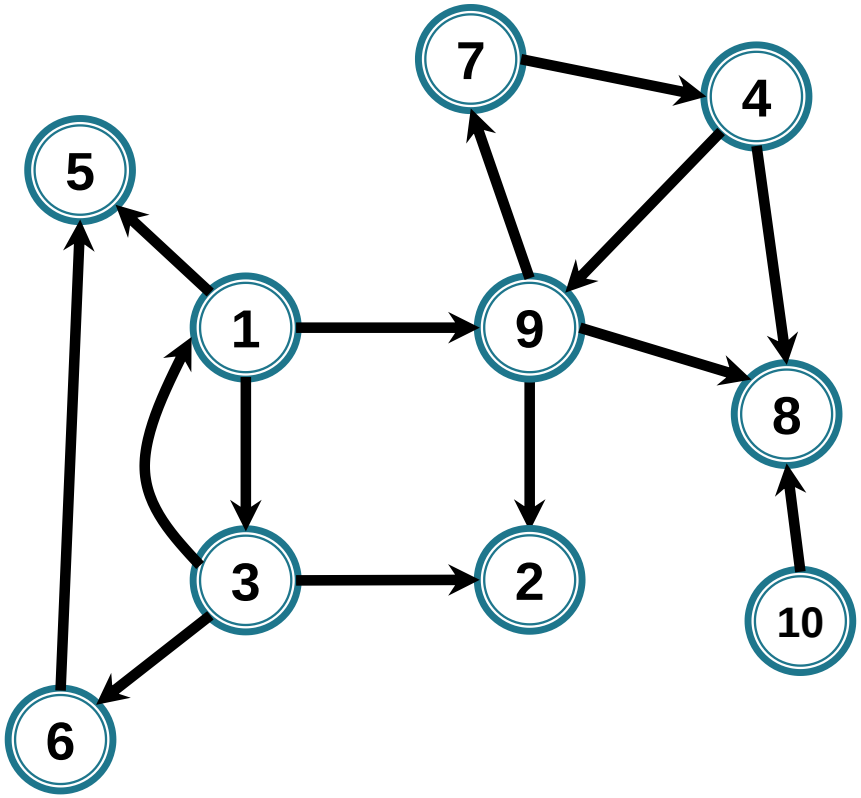
Parcurgerea în lățime

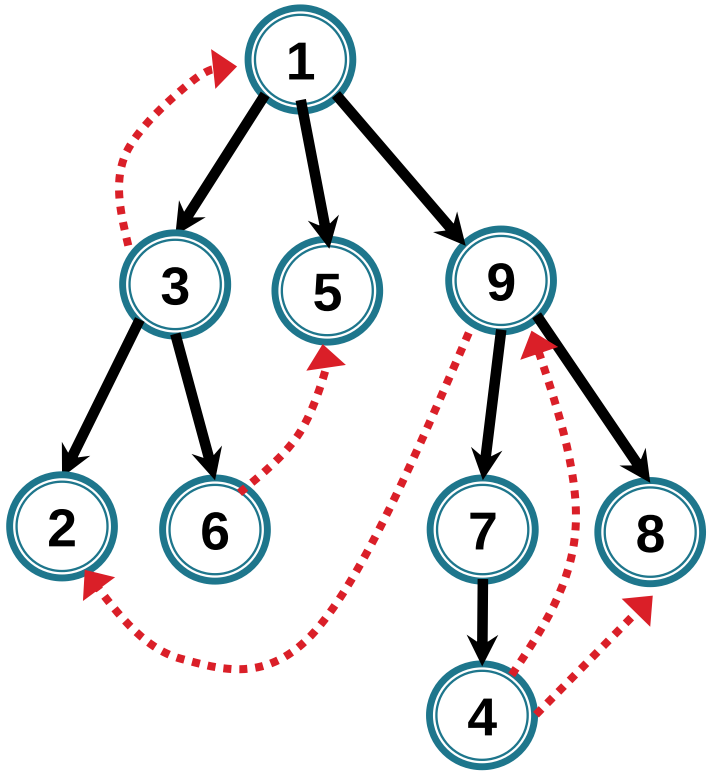




Arbore BF – tot arbore dacă ignorăm orientarea
– arcele corespunzătoare – orientate spre frunze

Parcurgerea în lățime





În arborele BF dacă adăugăm restul arcelor între vârfuri vizitate se închid cicluri, dar nu neapărat circuite

Pseudocod

Parcurgerea în lățime

- Informații necesare:

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

Parcurgerea în lățime

- Informații necesare:

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

Opţional

- **tata[j]** = acel vârf i din care este descoperit (vizitat) j => arborele BF
-

Parcurgerea în lățime

- Informații necesare:

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

Opţional

- **tata[j]** = acel vârf i din care este descoperit (vizitat) j => arborele BF
- **d[j]** = lungimea drumului determinat de algoritm de la s la j =
  = **nivelul lui j în arborele asociat parcurgerii**

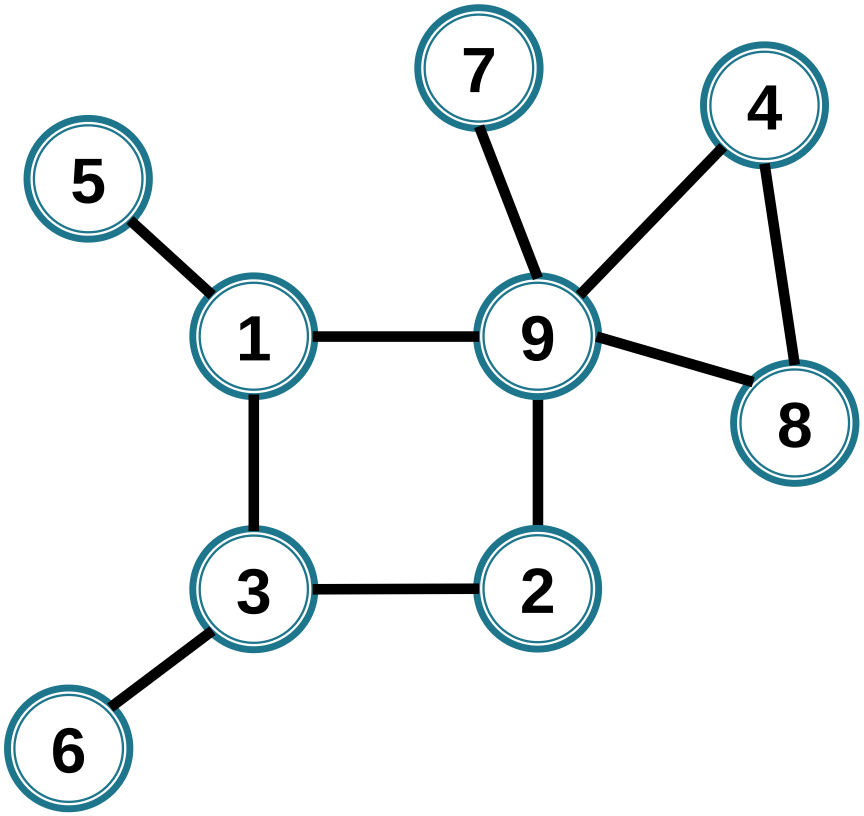
d[j] = d[tata[j]]+1

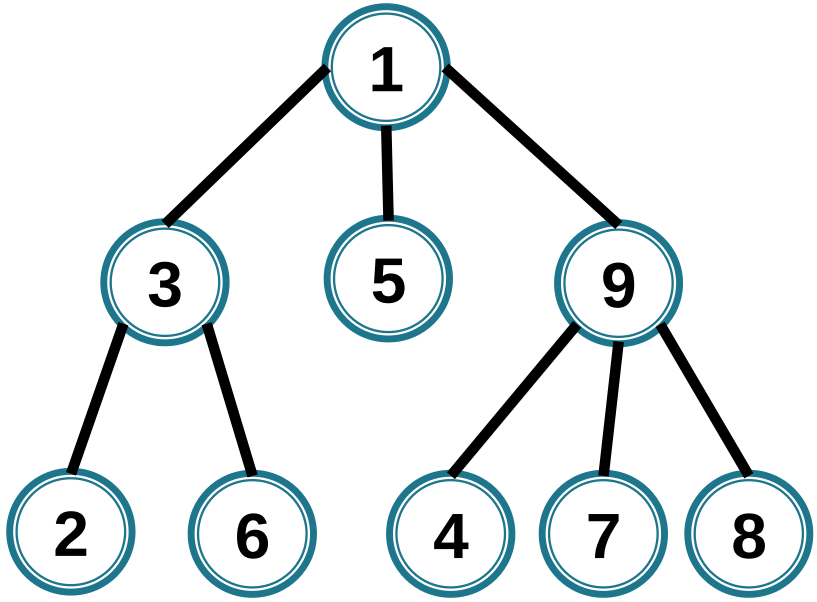
Parcurgerea în lățime

- **Propoziție – Corectitudinea BF**

  d[i] este chiar distanța de la s la i

Demonstraţia – urmează


7
4
5
1
9
8
3
2
6


1
3
5
9
2
6
4
7
8

Inițializări

```
pentru i=1,n executa
    viz[i]←0
    tata[i]←0
    d[i]← ∞
```

```
procedure BF(s)
   coada C ← ∅;
```

```
procedure BF(s)
   coada C ← ∅;
   adauga(s, C)
   viz[s]← 1; d[s] ← 0
```

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

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

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

      pentru j vecin al lui i
```

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

      pentru j vecin al lui i

          daca viz[j]=0 atunci
             adauga(j, C)
```

```
procedure BF(s)
   coada C ← ∅;
   adauga(s, C)
   viz[s]← 1; d[s] ← 0
   cat timp C ≠ ∅ 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
             d[j] ← d[i]+1
```

Complexitate

Complexitate

- Matrice de adiacență
- Liste de adiacență

Complexitate

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

Implementare

Inițializări

```
pentru i=1,n executa
    viz[i]←0
    tata[i]←0
    d[i]← ∞
```

```
for(i=0;i<n;i++){
    viz[i]=0;
    tata[i]=d[i]=-1;//sau n+1
}
```

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

     pentru j vecin al lui i

          daca viz[j]=0
               adauga(j, C)
               viz[j] ← 1
               tata[j] ← i
               d[j] ← d[i]+1
```

```
queue<int> c;
c.push(s);
viz[s]=1; d[s]=0;
while(c.size()>0){
     int x=c.front();c.pop();
     parc_bf.push_back(x+1);
     for (i=0;i<la[x].size();i++){
        int y=la[x][i];
        if(viz[y]==0) {
             c.push(y);
             viz[y]=1;
             tata[y]=x;
             d[y]=d[x]+1;
        }
    }
}
```

Inițializări

```
pentru i=1,n executa
    viz[i]←0                    viz=[0]*n
    tata[i]←0                   tata=[None]*n
    d[i]← ∞                     d=[None]*n
```

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

     pentru j vecin al lui i

          daca viz[j]=0
               adauga(j, C)
               viz[j] ← 1
               tata[j] ← i
               d[j] ← d[i]+1
```

```
q = deque()
q.append(s)
viz[s]=1; d[s]=0
while len(q)>0:
     x=q.popleft()
     parc_bf.append(x+1)

     for y in la[x]:
          if viz[y]==0:
               q.append(y)
               viz[y]=1
               tata[y]=x
               d[y]=d[x]+1
```

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

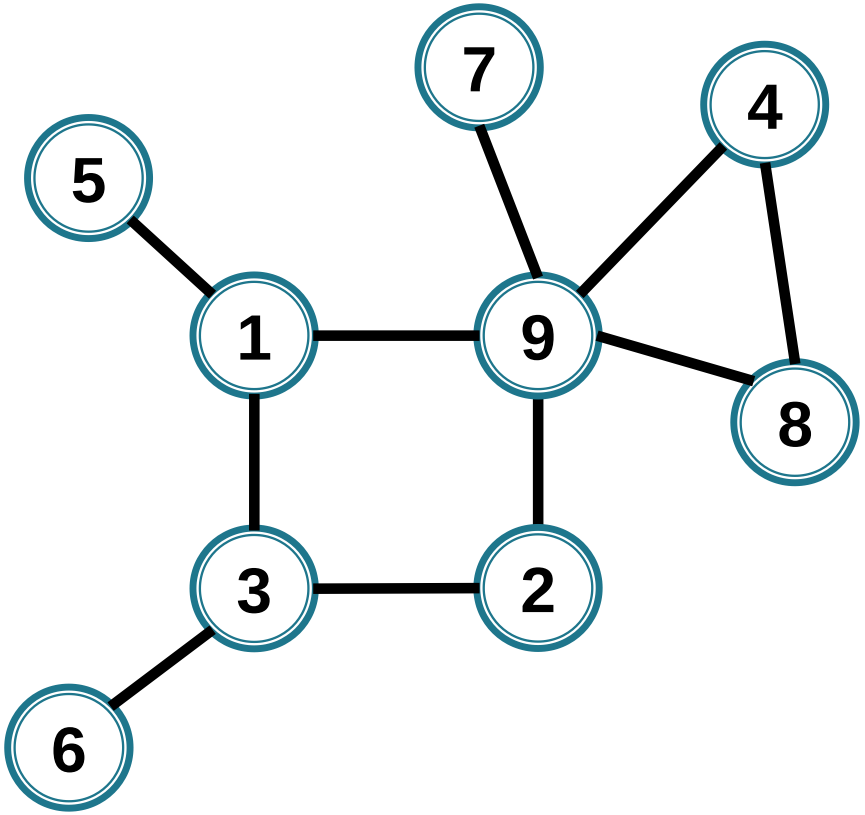
Se vizitează

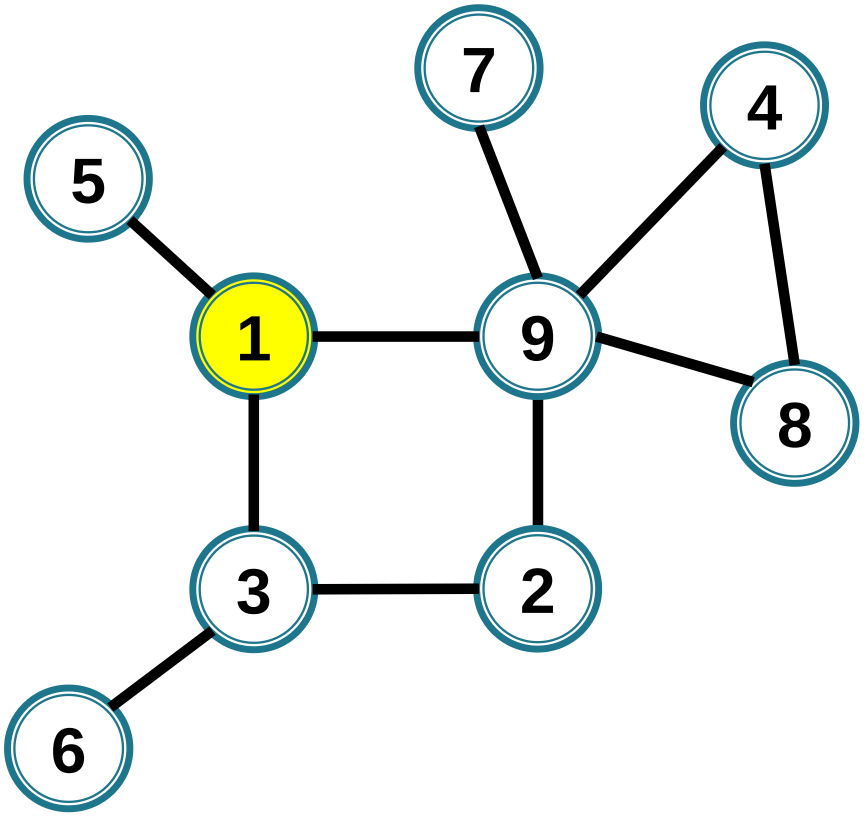
- **Inițial**: vârful de start s – devine vârf curent
- **La un pas**:
  - se trece la primul vecin nevizitat al vârfului curent, **dacă există**
  - altfel
    - **se merge înapoi pe drumul de la s la vârful curent, până se ajunge la un vârf cu vecini nevizitați**
    -

Parcurgerea în adâncime

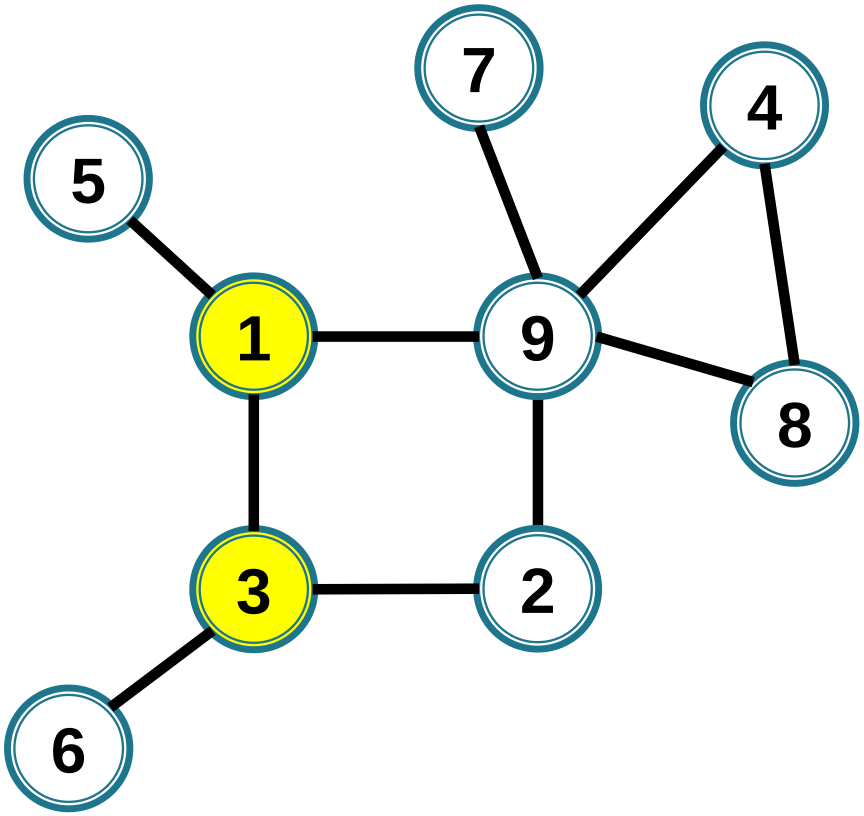
Se vizitează

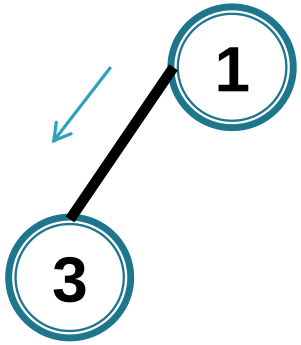
- **Inițial**: vârful de start s – devine vârf curent
- **La un pas**:
  - se trece la primul vecin nevizitat al vârfului curent, **dacă există**
  - altfel
    - **se merge înapoi pe drumul de la s la vârful curent, până se ajunge la un vârf cu vecini nevizitați**
    - **se trece la primul dintre aceștia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

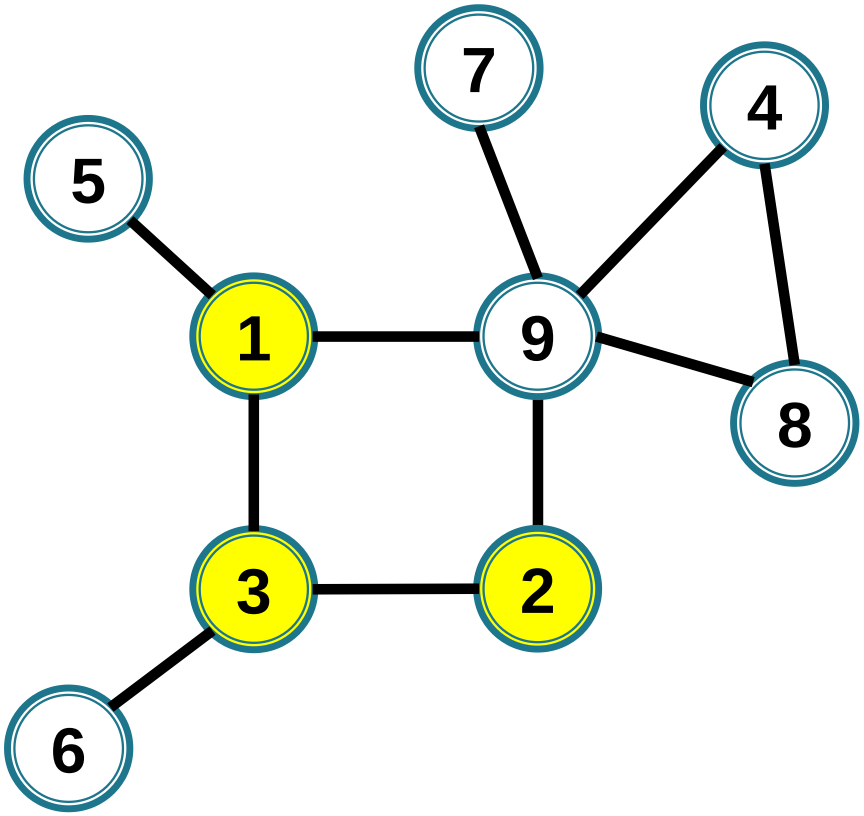

7
4
5
1
9
8
3
2
6

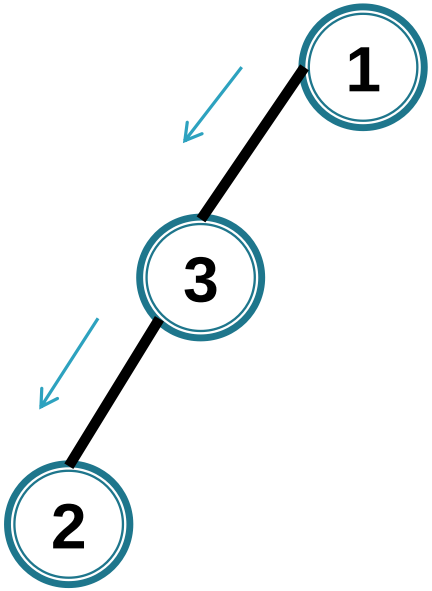

7
4
5
1
9
8
3
2
6

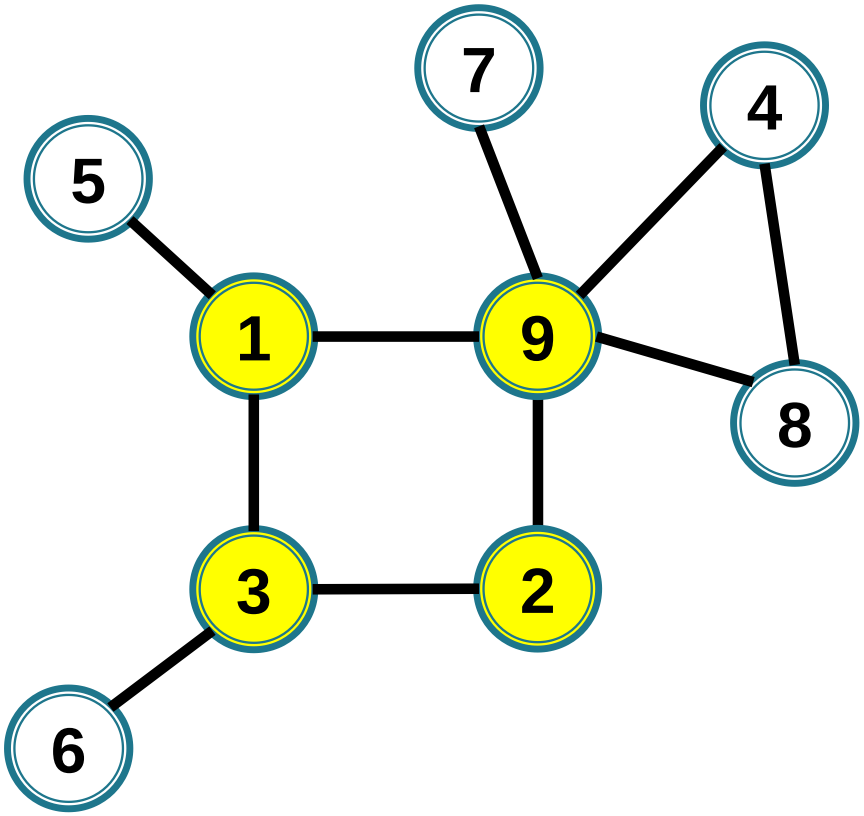

1

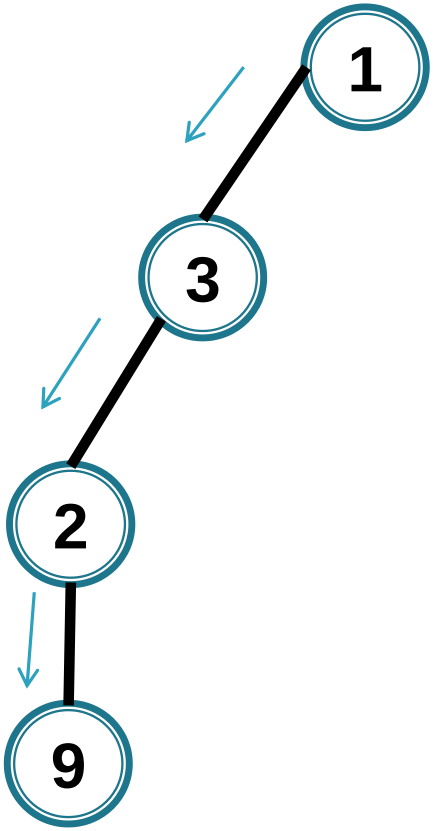

7
4
5
1
9
8
3
2
6

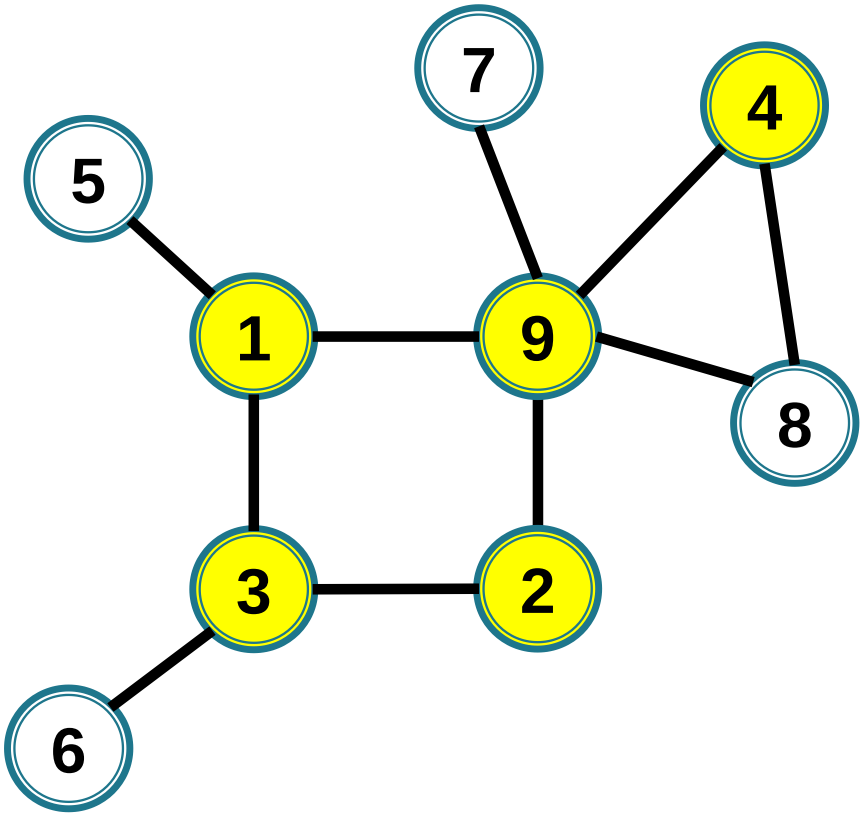

1
3

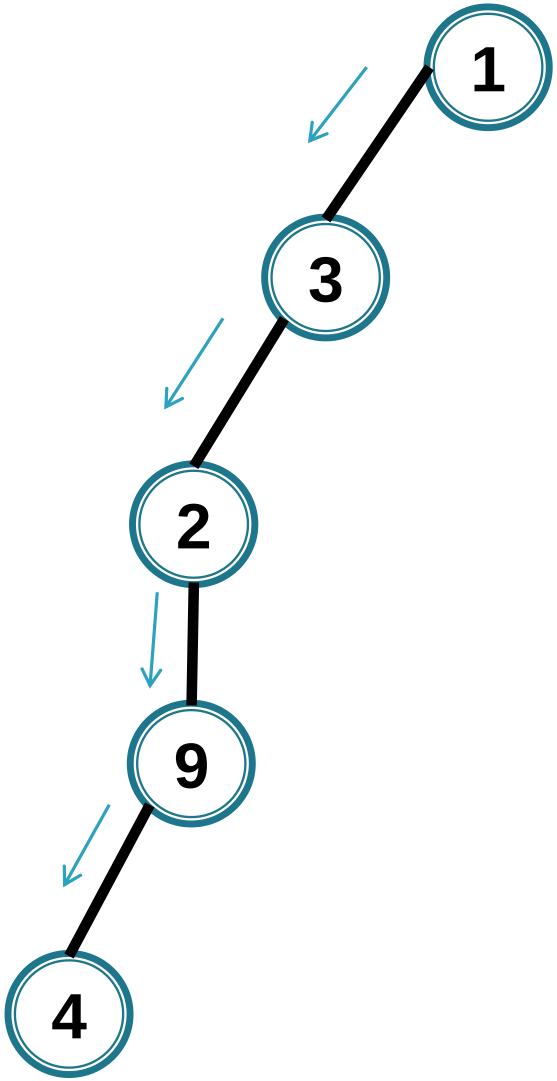

7
4
5
1
9
8
3
2
6

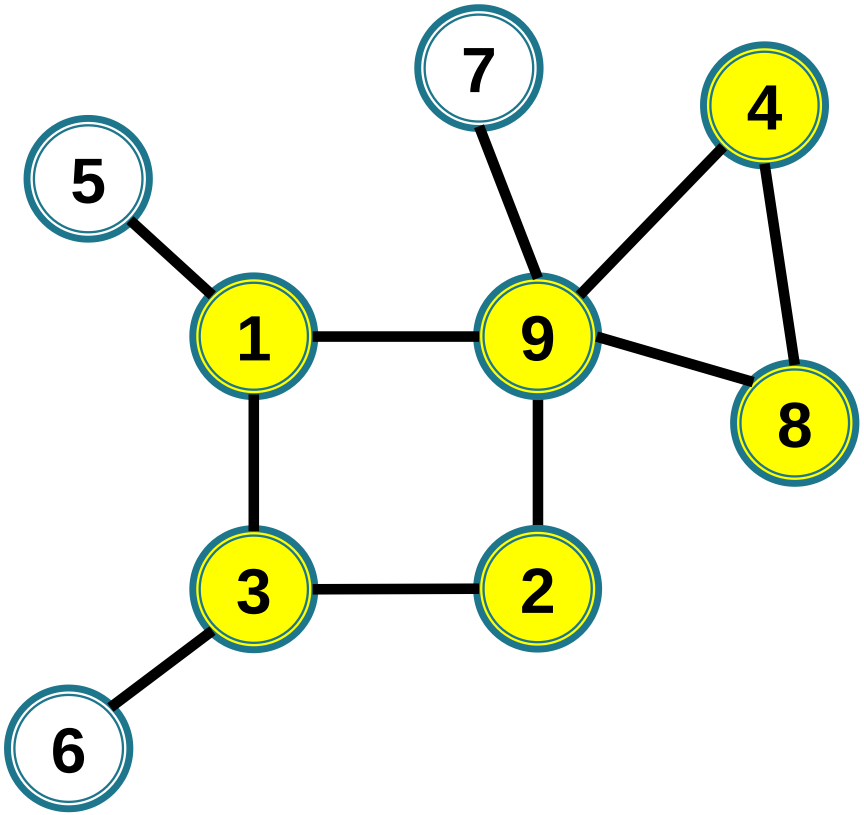

1
3
2

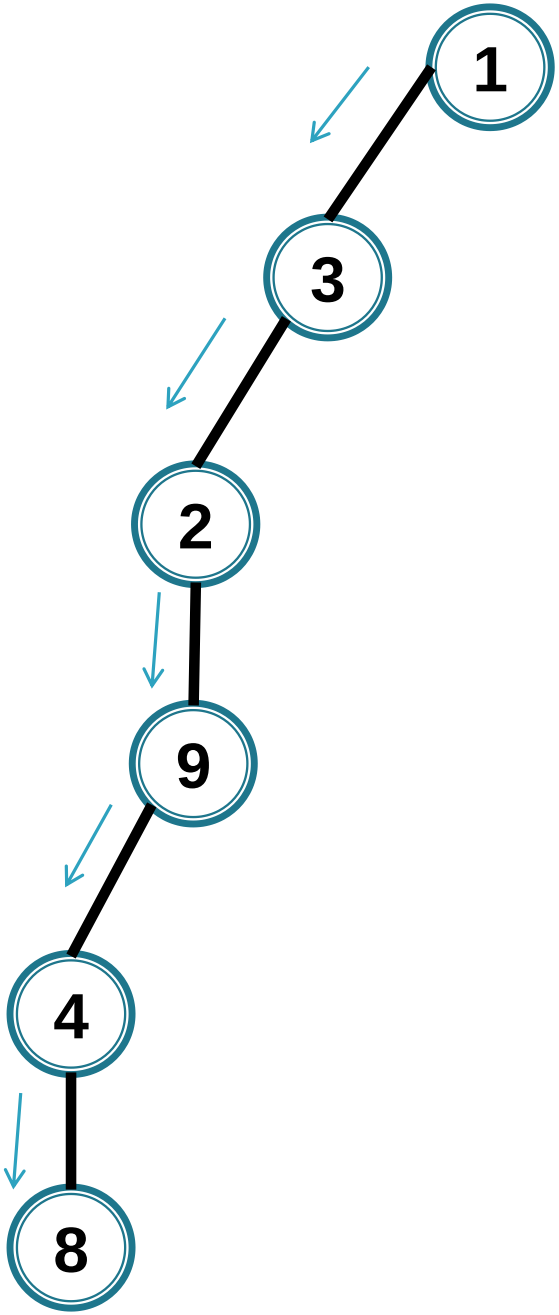

7
4
5
1
9
8
3
2
6

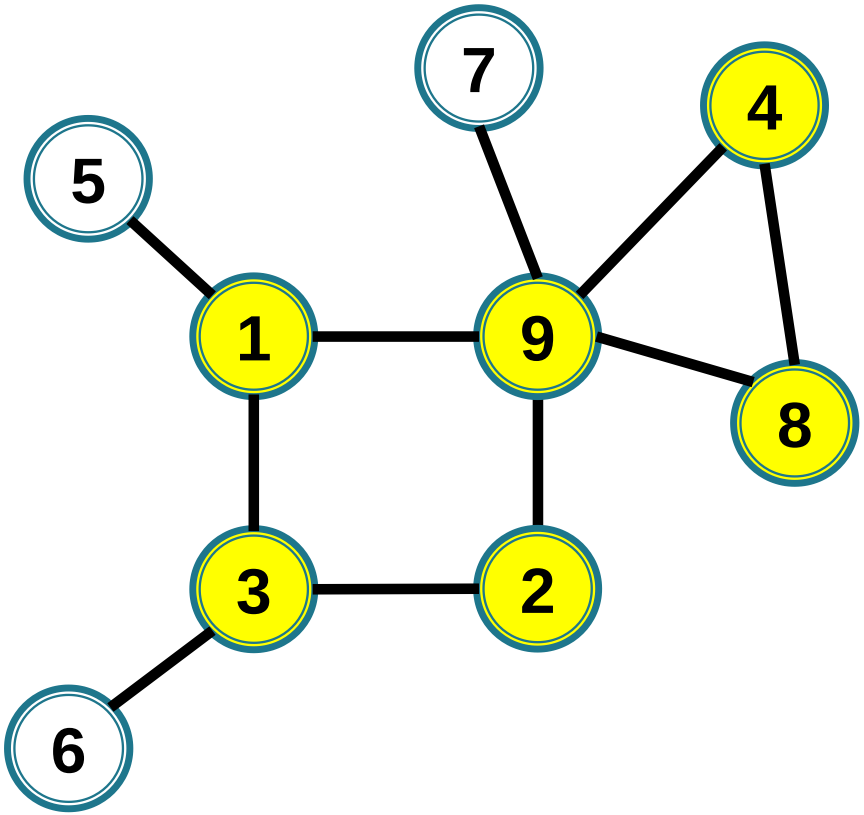

1
3
2
9

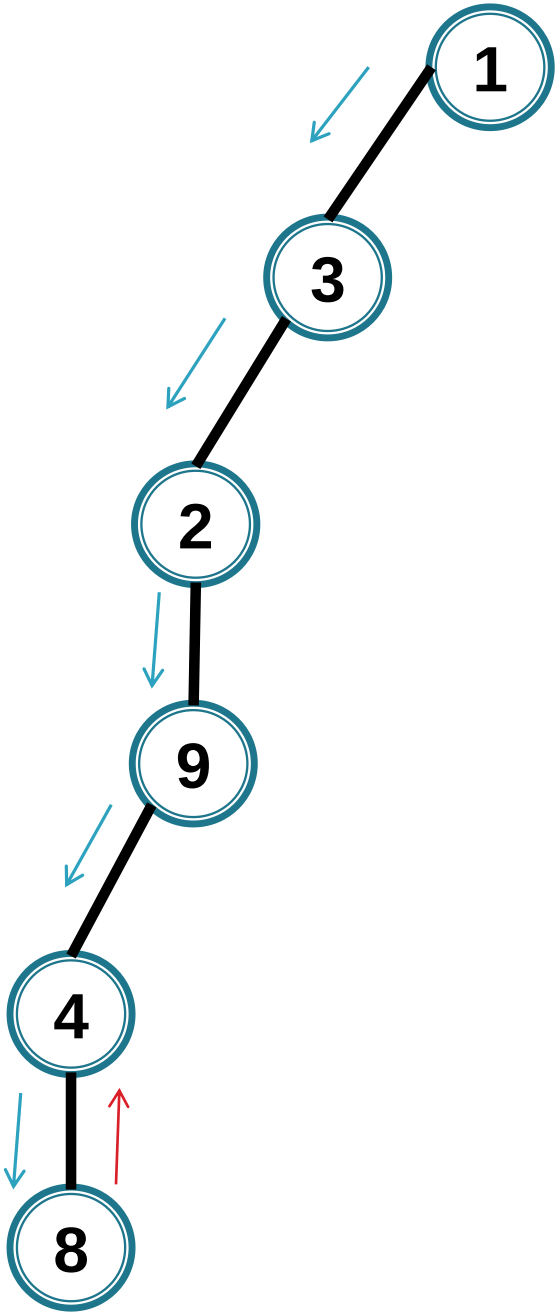

7
4
5
1
9
8
3
2
6

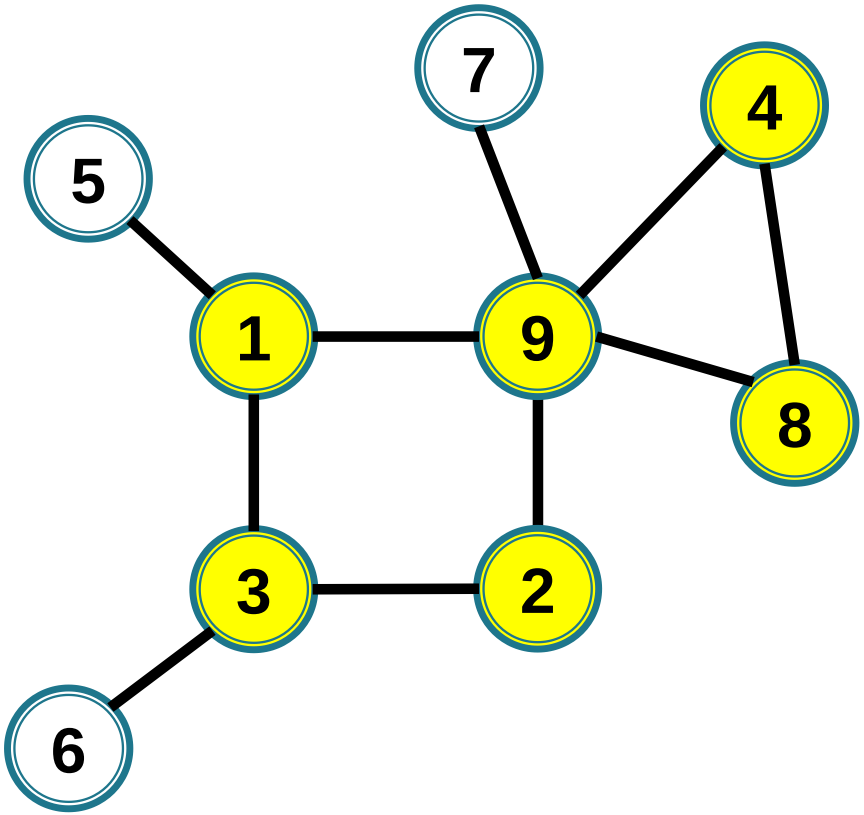

1
3
2
9
4

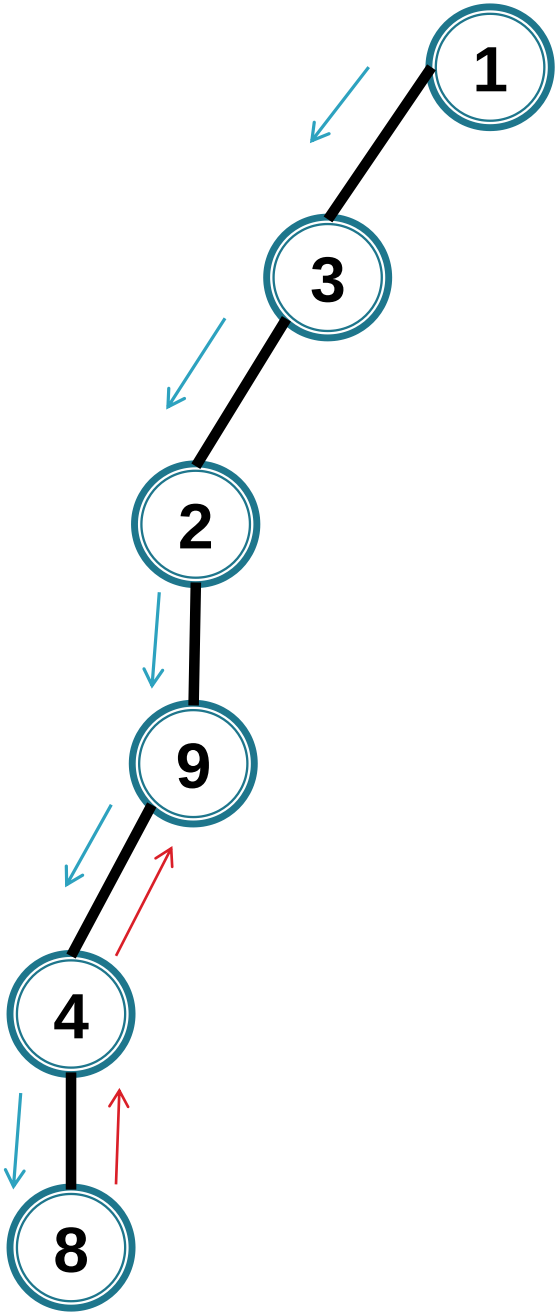

7
4
5
1
9
8
3
2
6

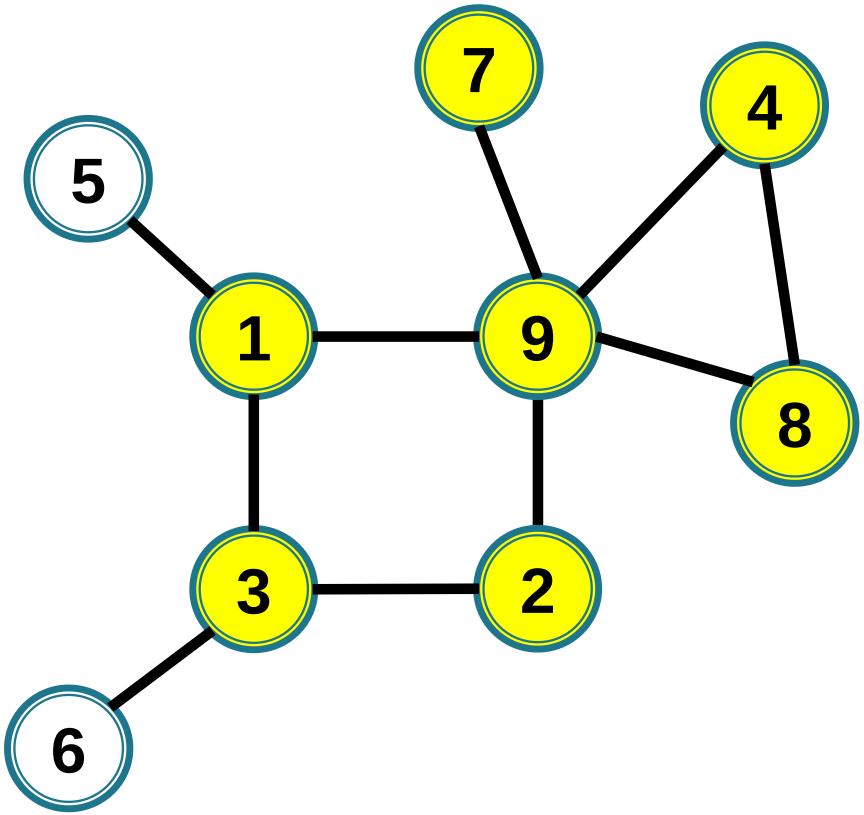

1
3
2
9
4
8

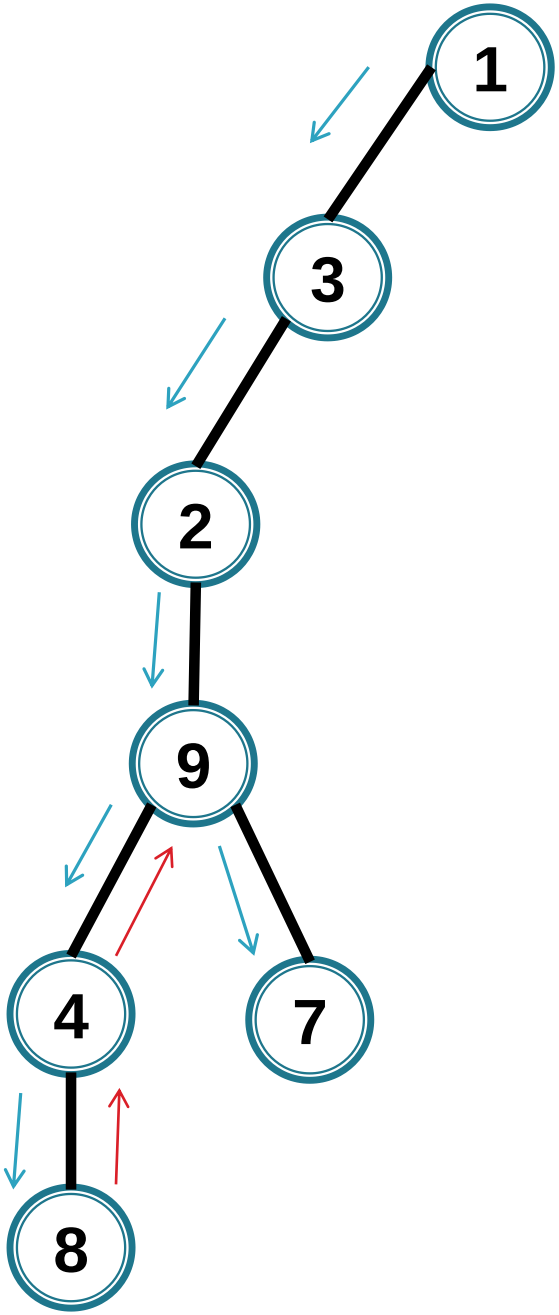

7
4
5
1
9
8
3
2
6

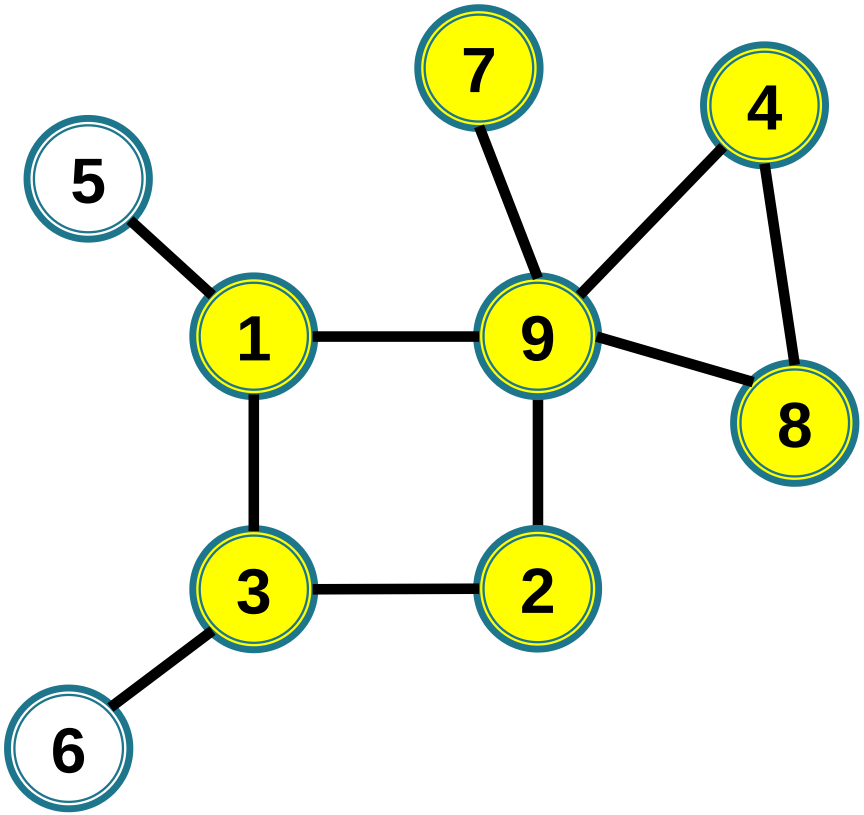

1
3
2
9
4
8

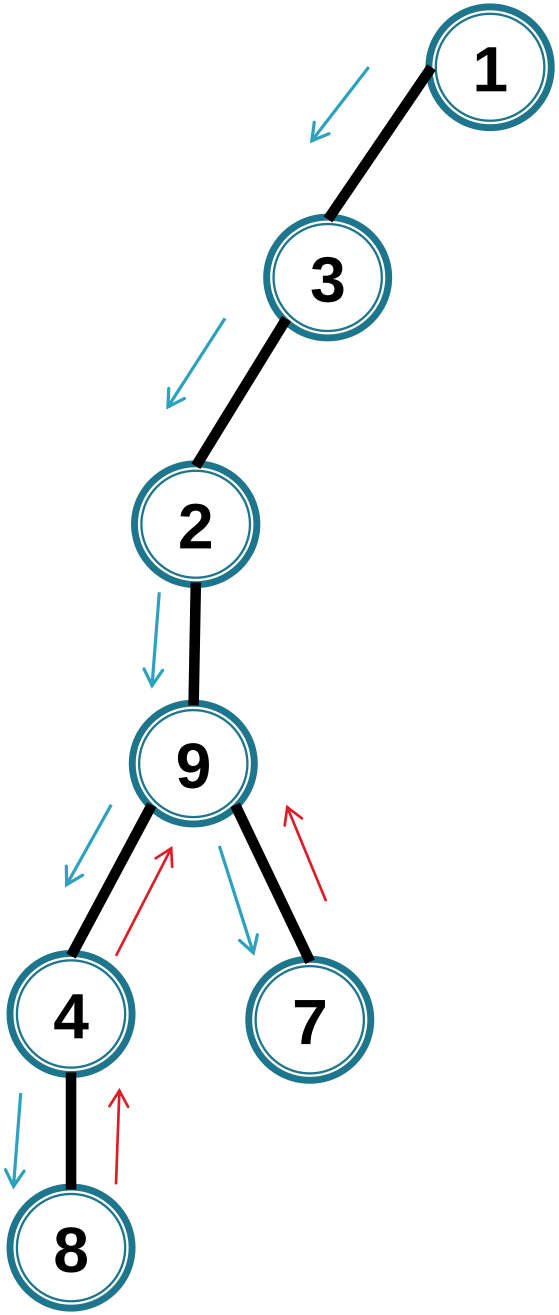

7
4
5
1
9
8
3
2
6

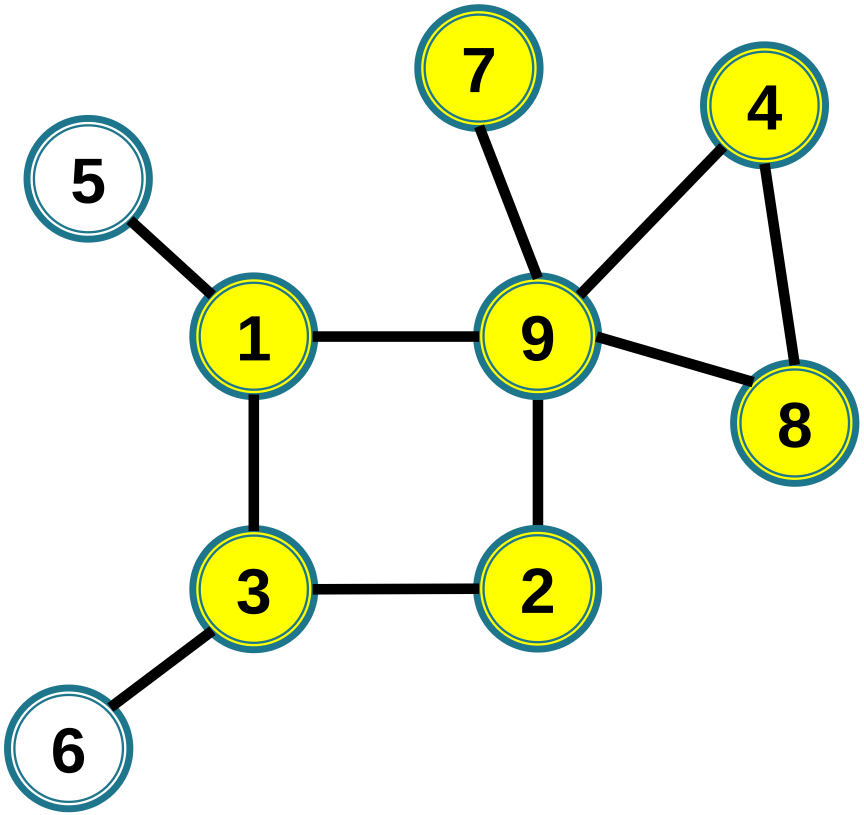

1
3
2
9
4
8

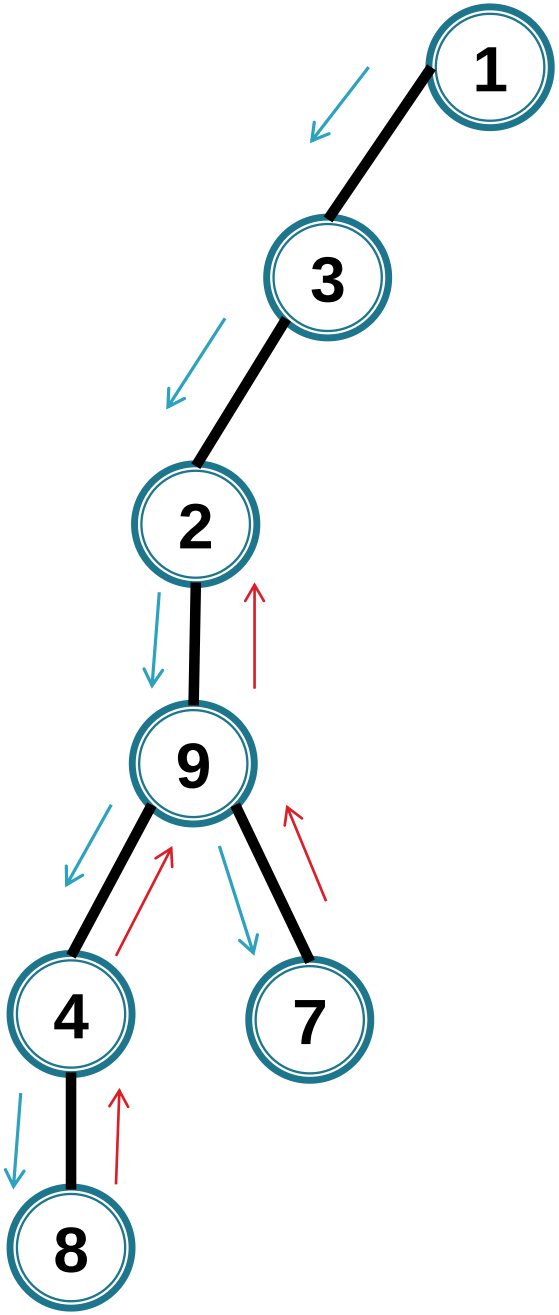

7
4
5
1
9
8
3
2
6

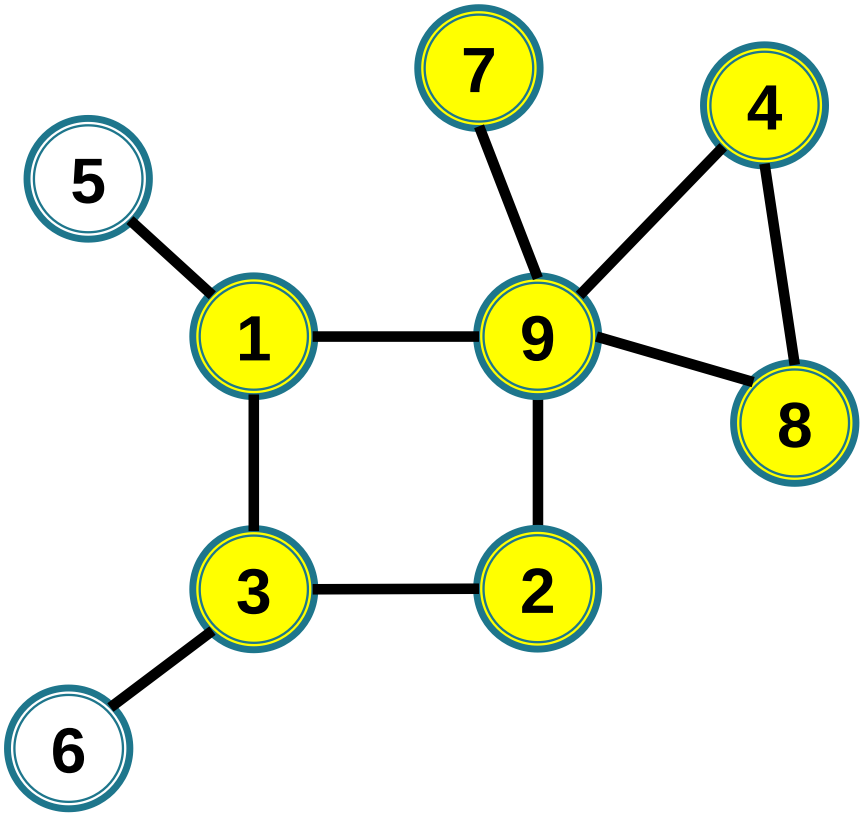

1
3
2
9
4
7
8

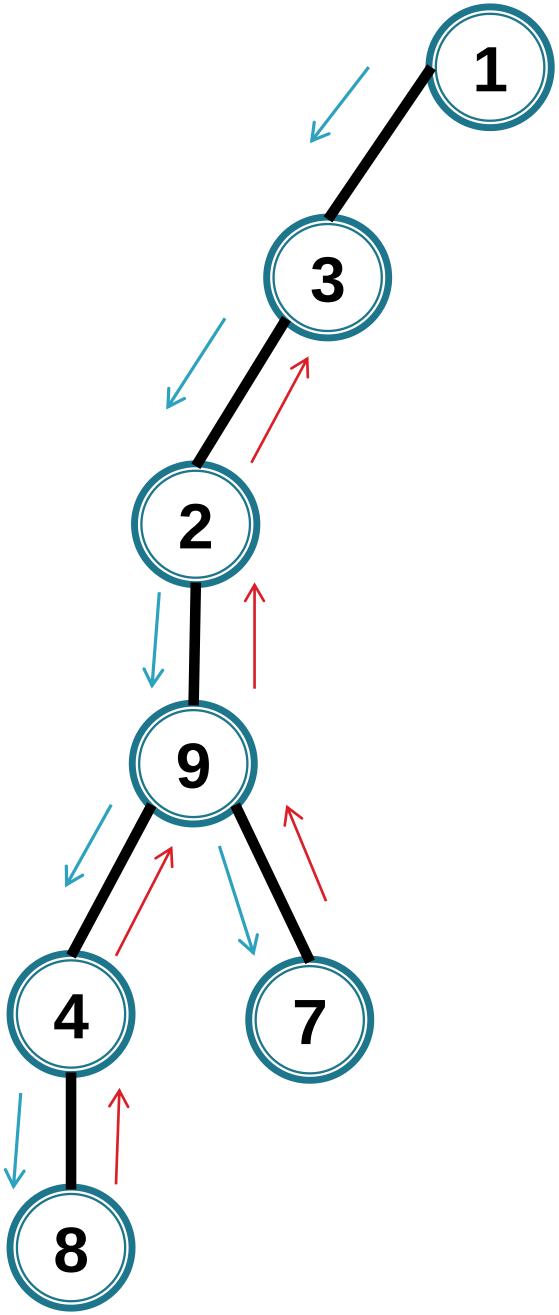

7
4
5
1
9
8
3
2
6

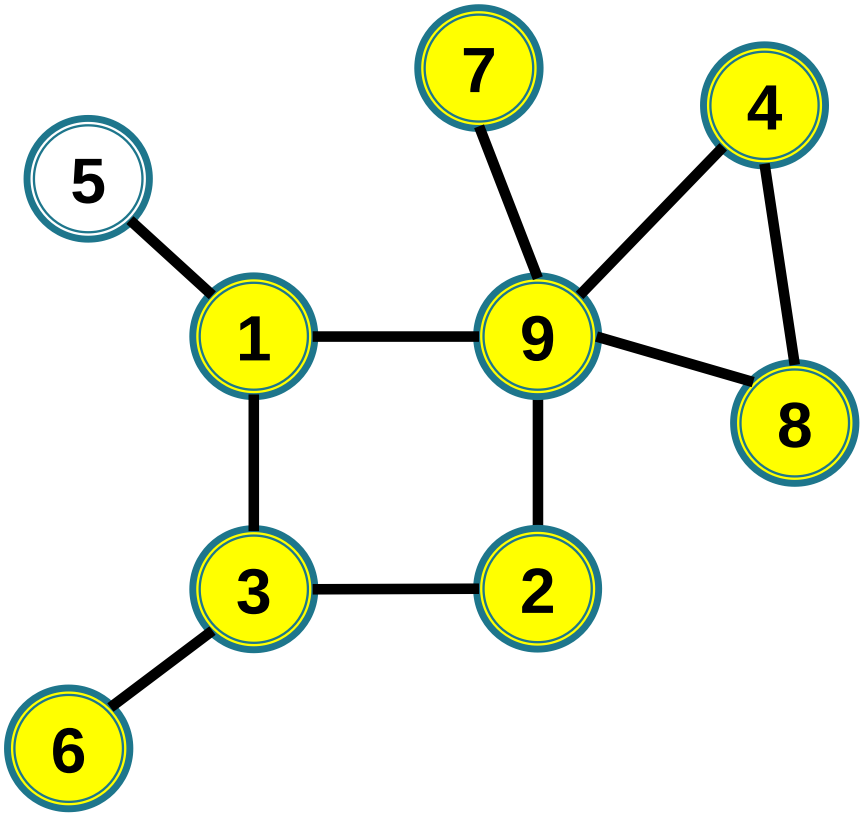

1
3
2
9
4
7
8

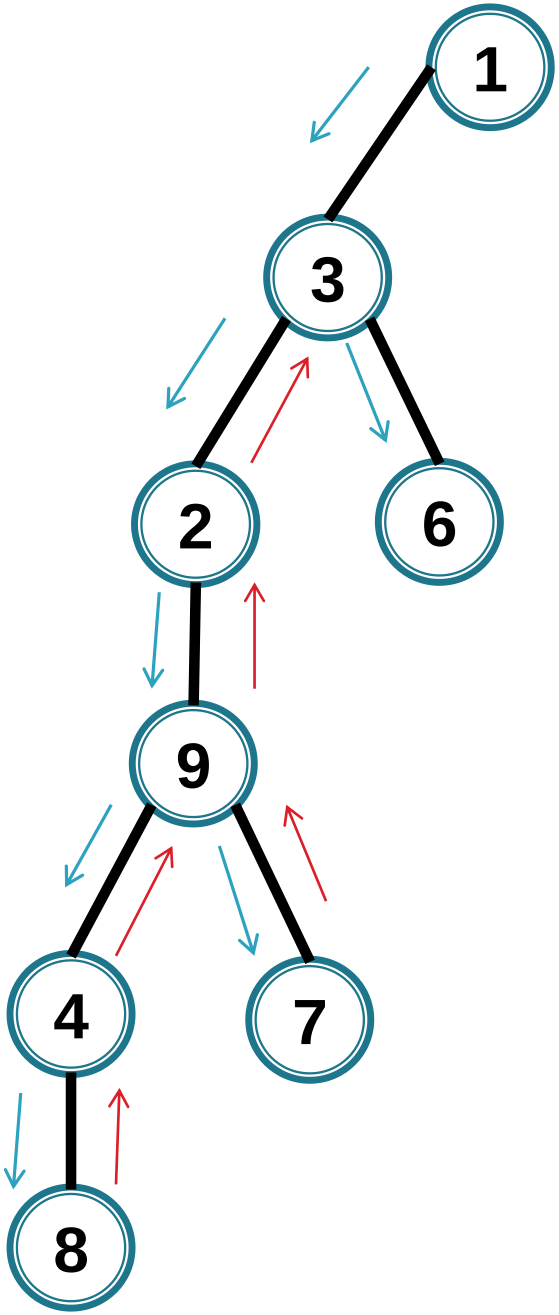

7
4
5
1
9
8
3
2
6

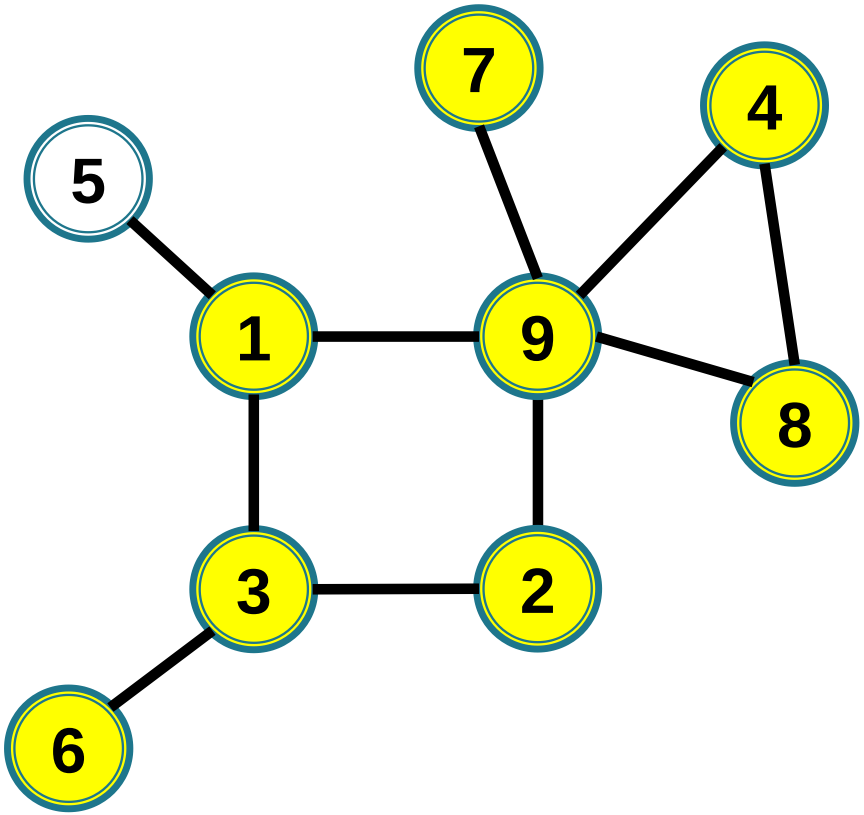

1
3
2
9
4
7
8

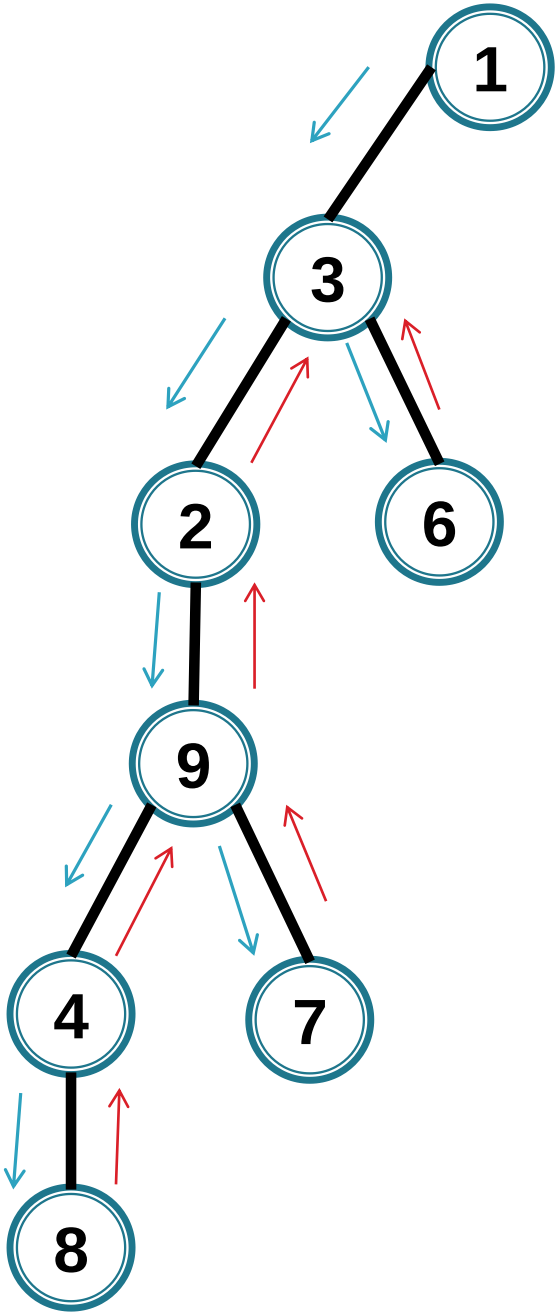

7
4
5
1
9
8
3
2
6

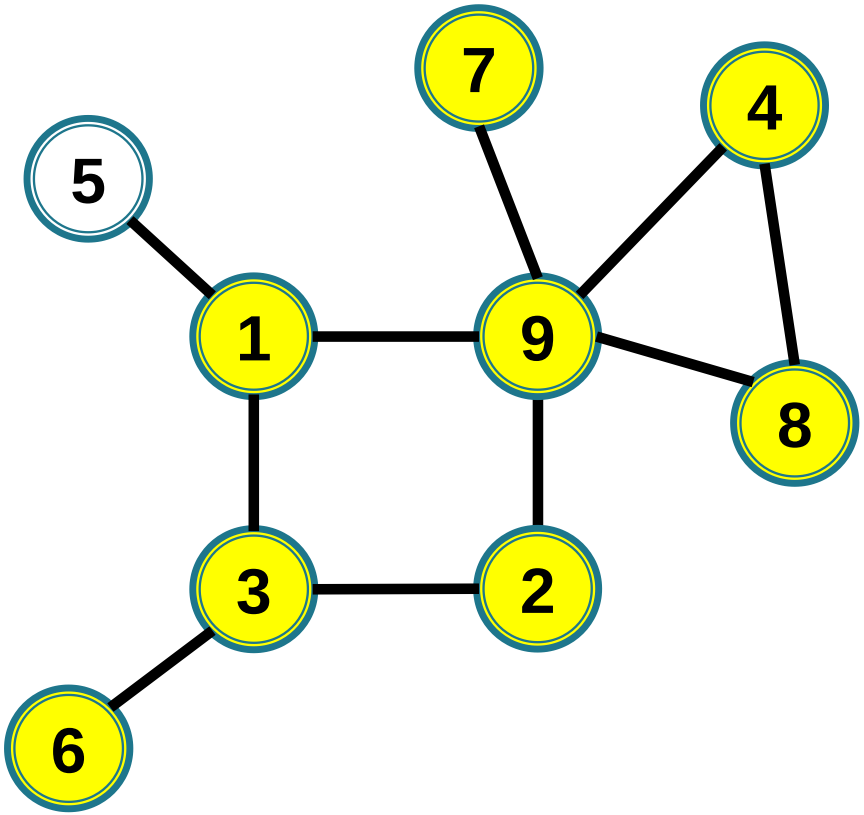

1
3
2
9
4
7
8

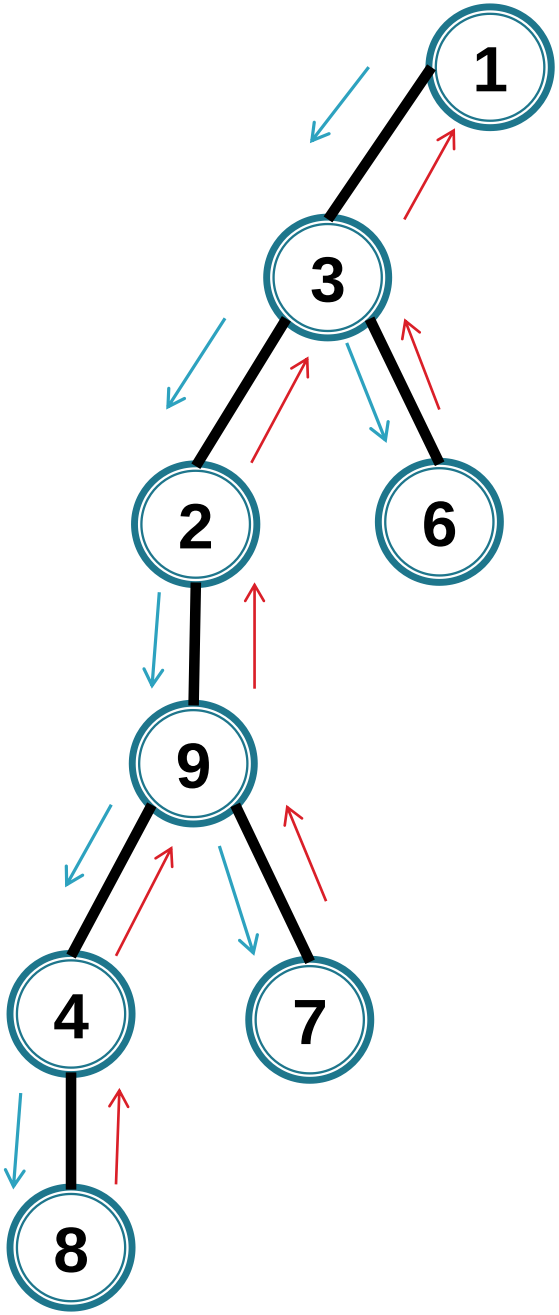

7
4
5
1
9
8
3
2
6

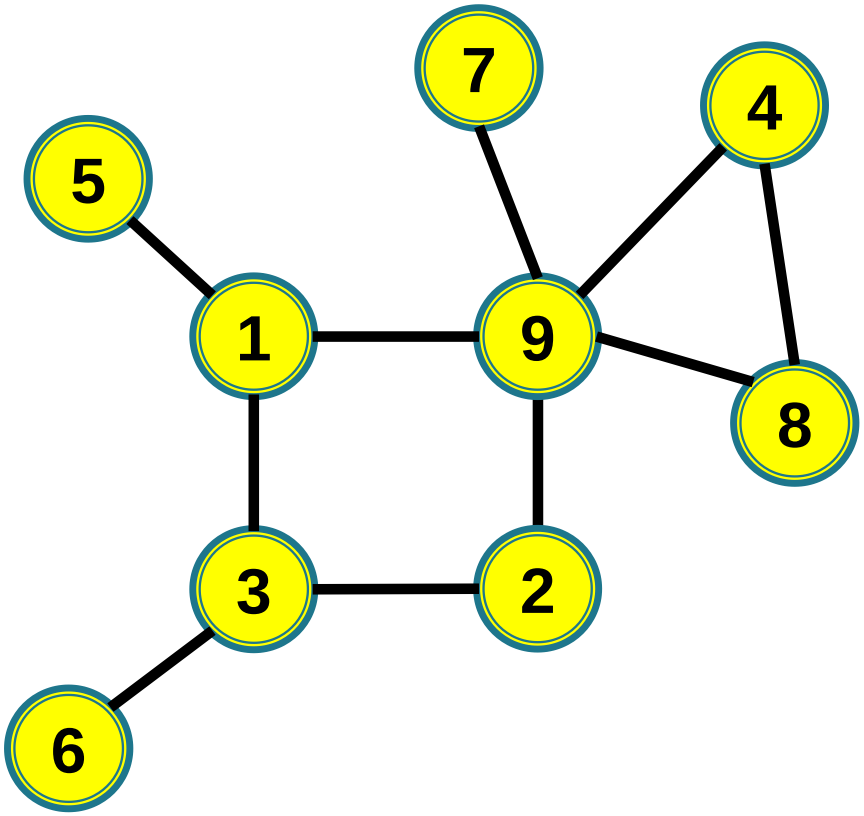

1
3
2
6
9
4
7
8

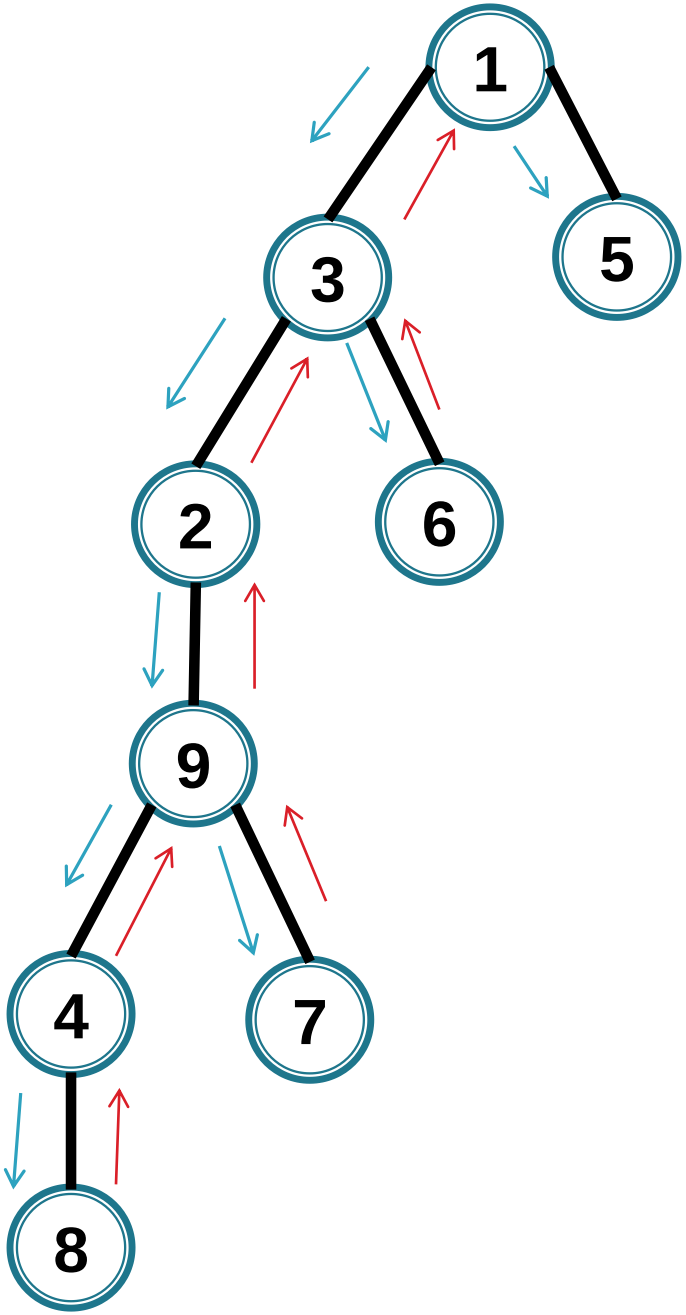

7
4
5
1
9
8
3
2
6

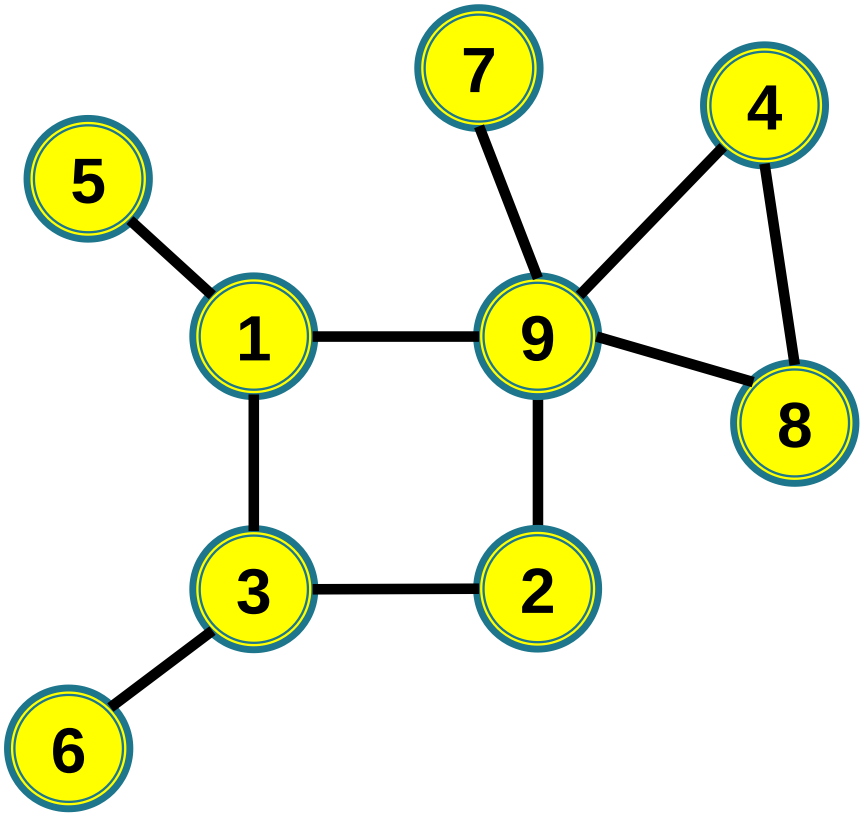

1
3
2
6
9
4
7
8

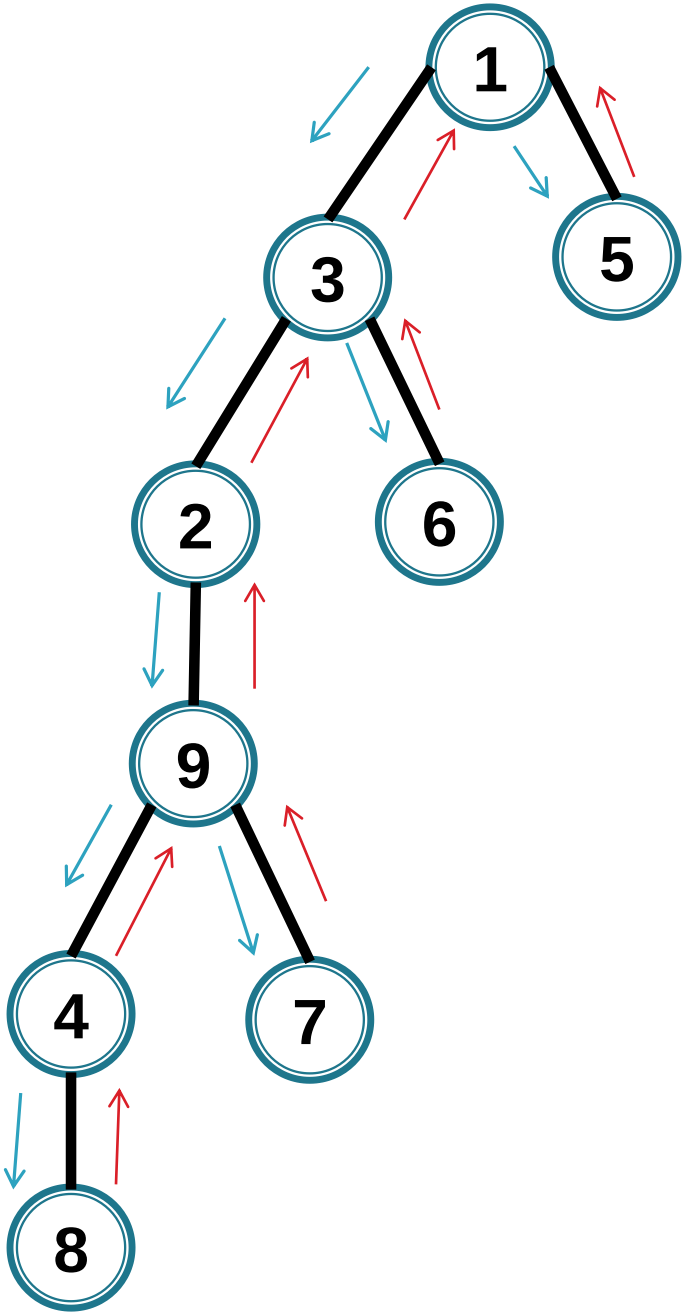

7
4
5
1
9
8
3
2
6

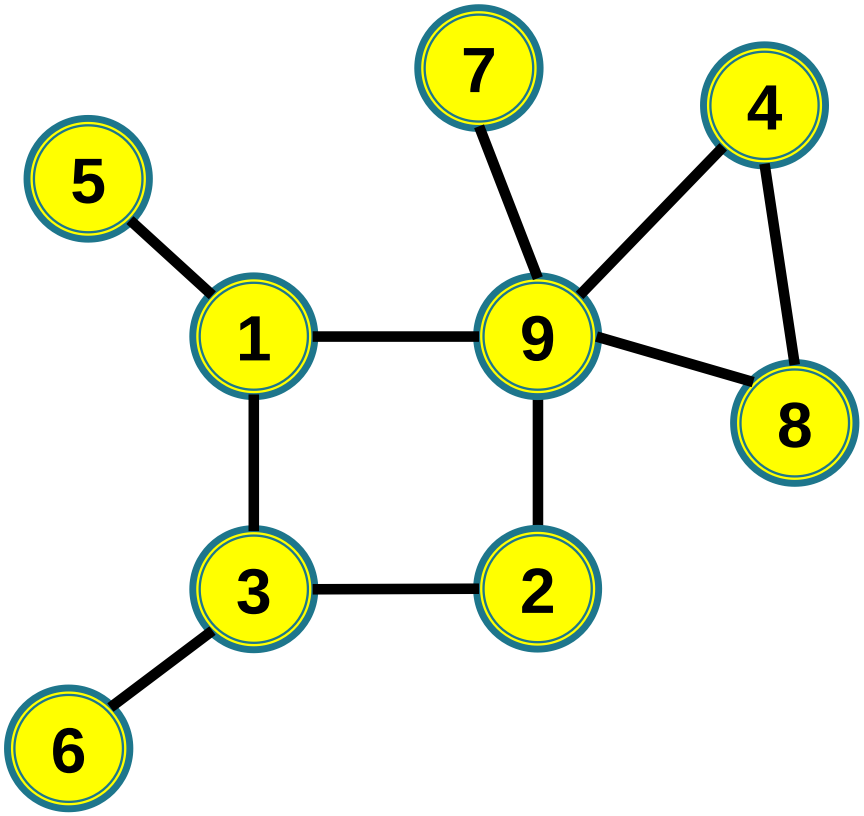

1
3
2
6
9
4
7
8

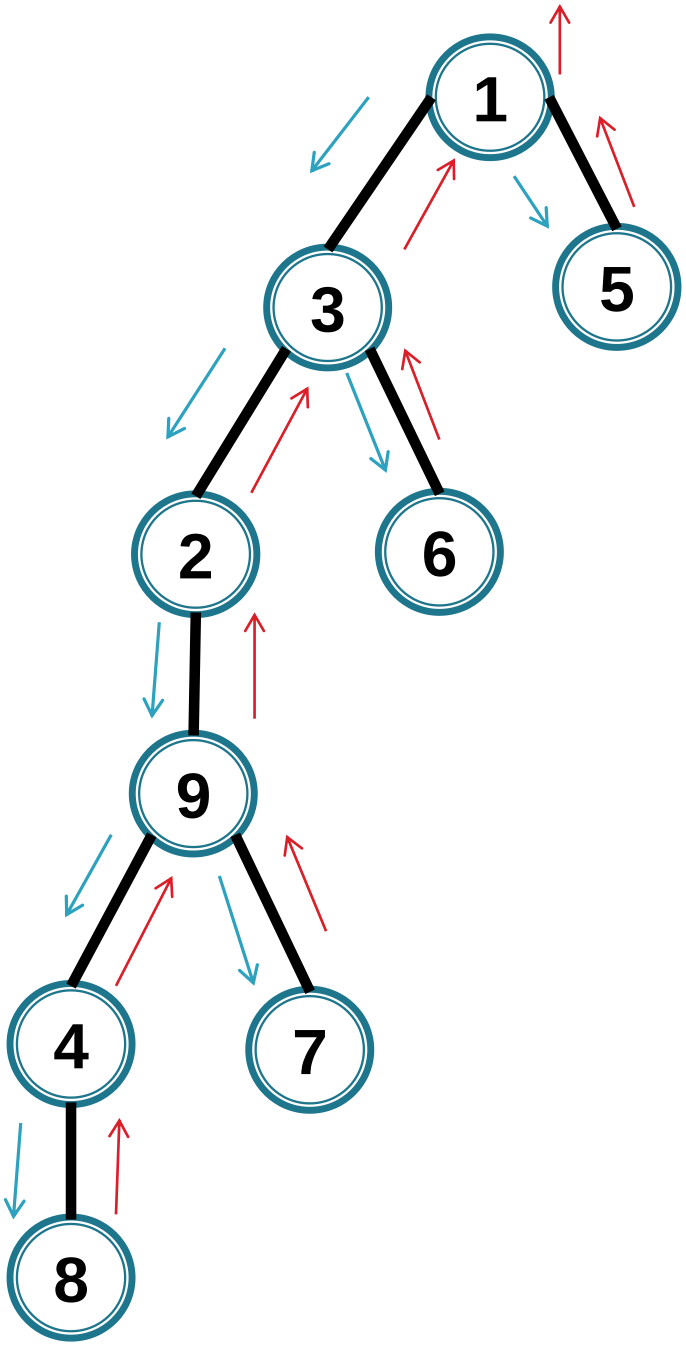

7
4
5
1
9
8
3
2
6

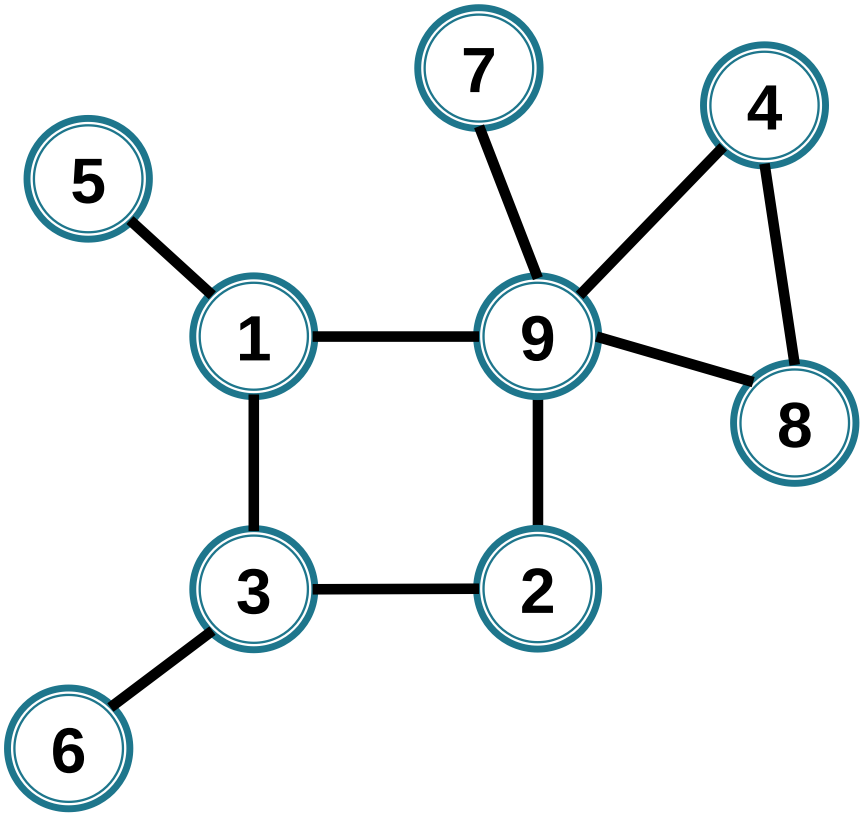

1
3
5
2
6
9
4
7
8

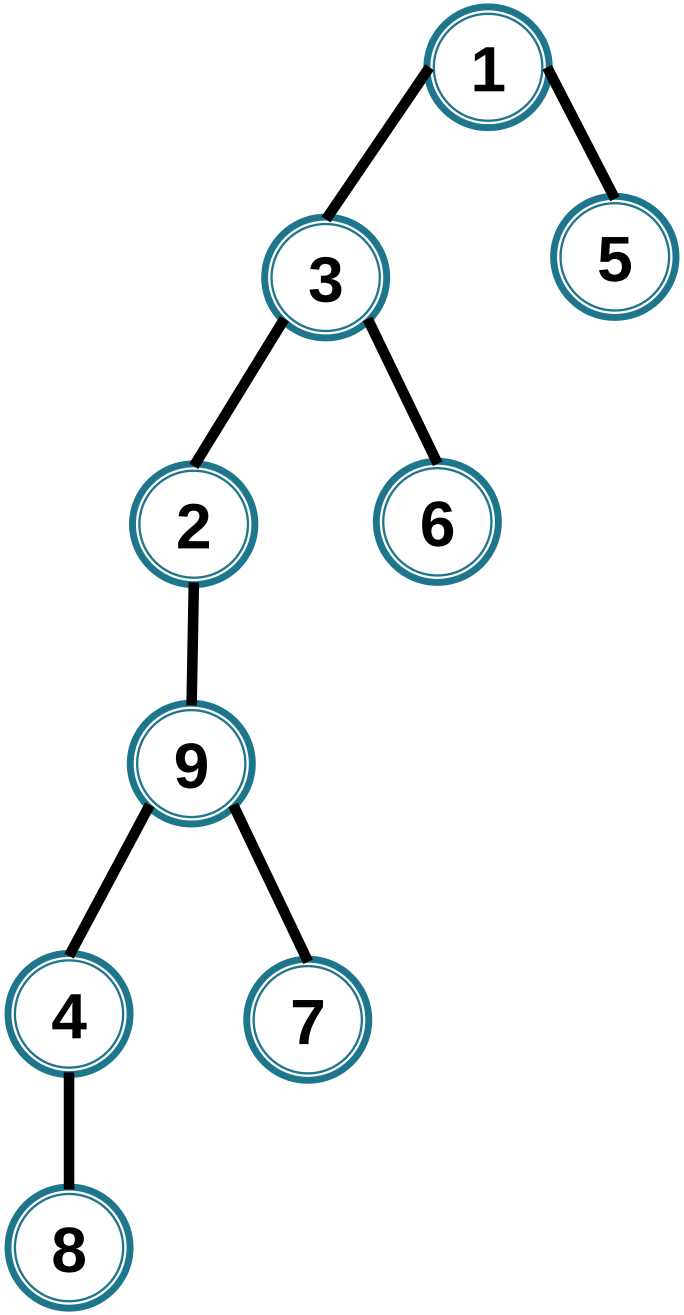

7
4
5
1
9
8
3
2
6


1
3
5
2
6
9
4
7
8


7
4
5
1
9
8
3
2
6


1
3
5
2
6
9
4
7
8

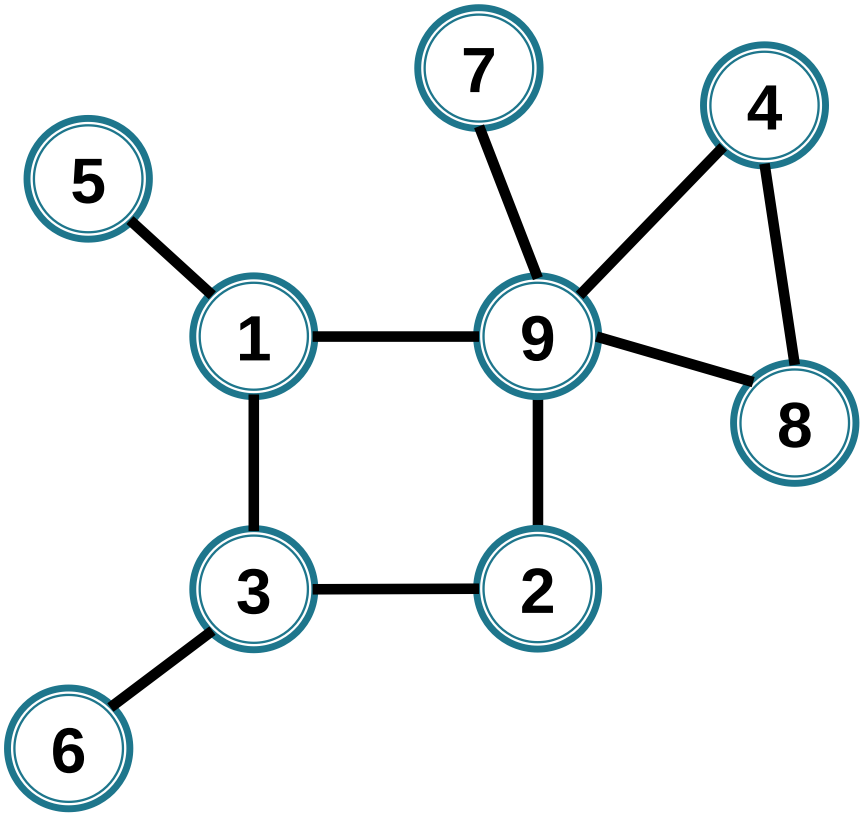

7
4
5
1
9
8
3
2
6

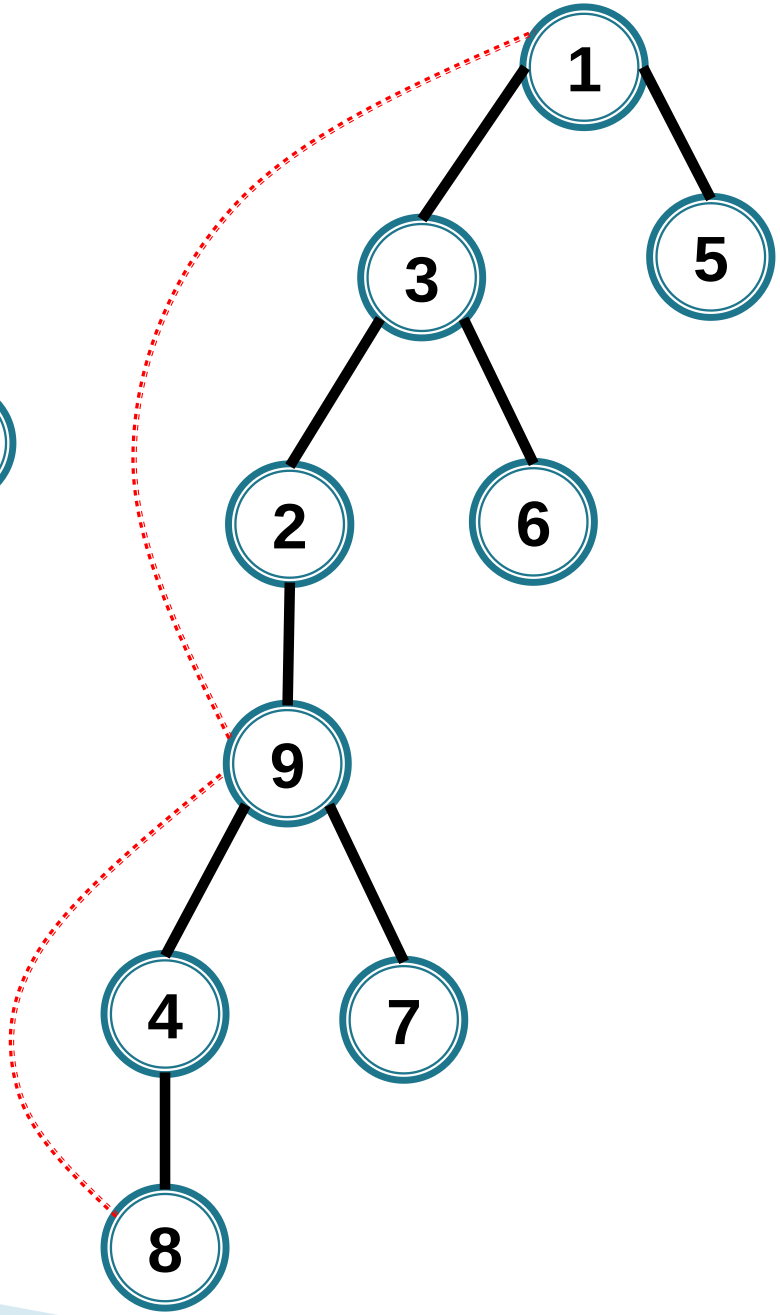

1
3
5
2
6
9
4
7
8

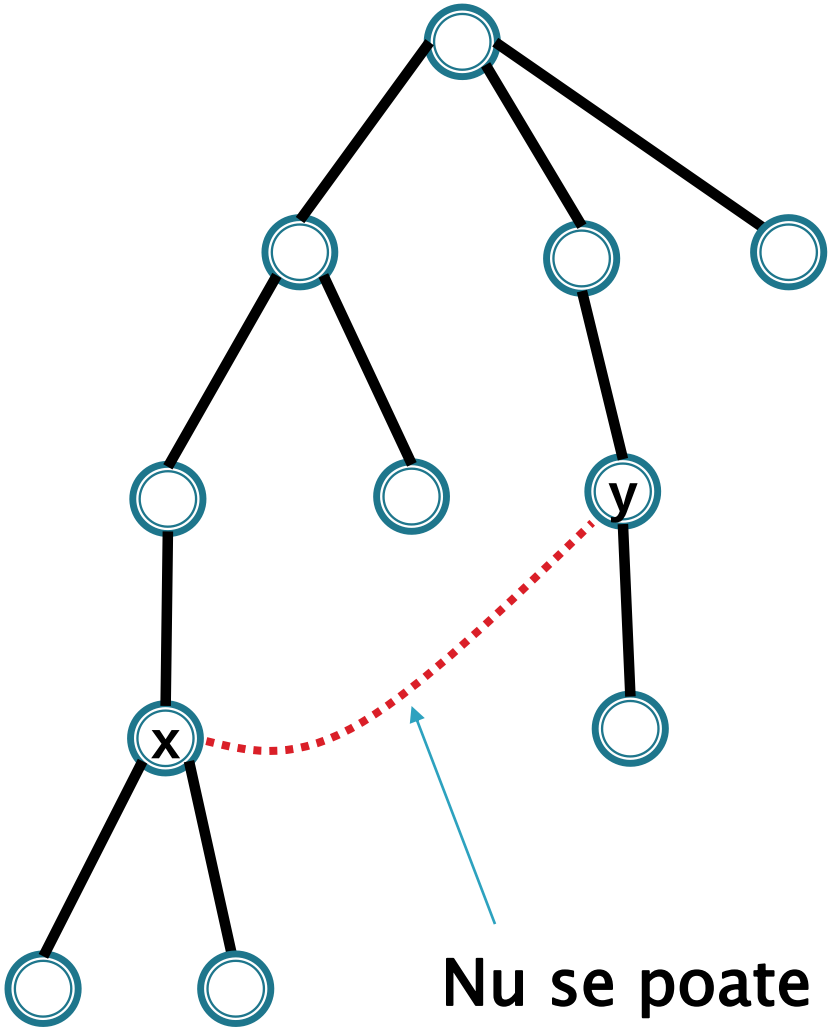
Parcurgerea în adâncime

- Muchiile folosite pentru a descoperi vârfuri noi formează un **arbore** (numit **arbore DF**) -> **se numesc** **muchii de avansare**

- Muchiile din graf care nu sunt în arbore închid cicluri (cu muchiile din arbore), mai exact **unesc un vârf cu un ascendent al lui în arborele DF -> se numesc** **muchii de întoarcere**

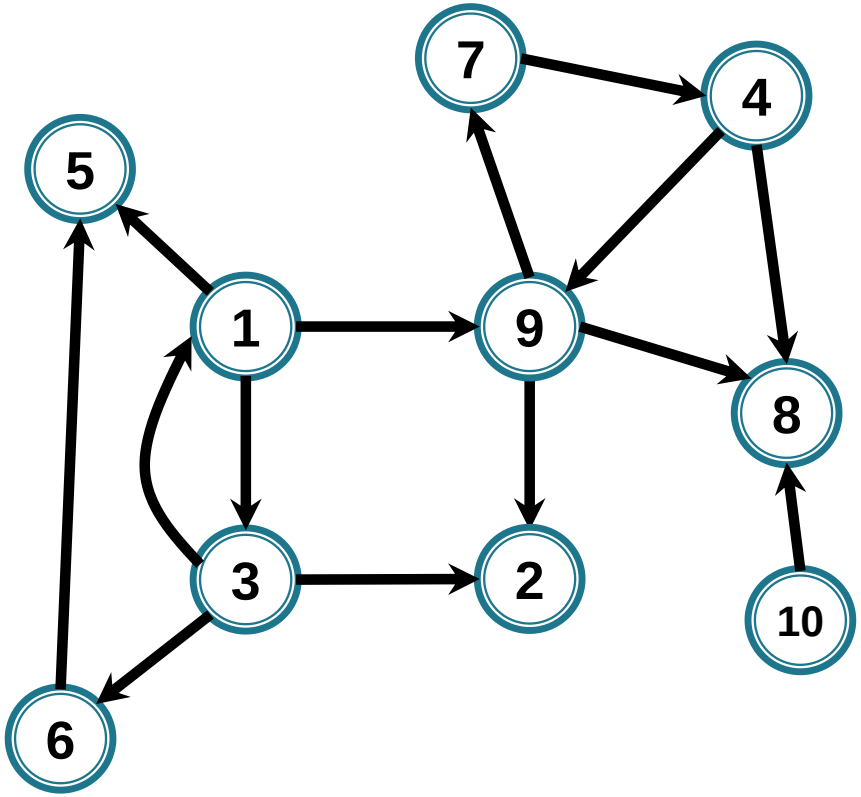

7
4
5
1
9
8
3
2
6

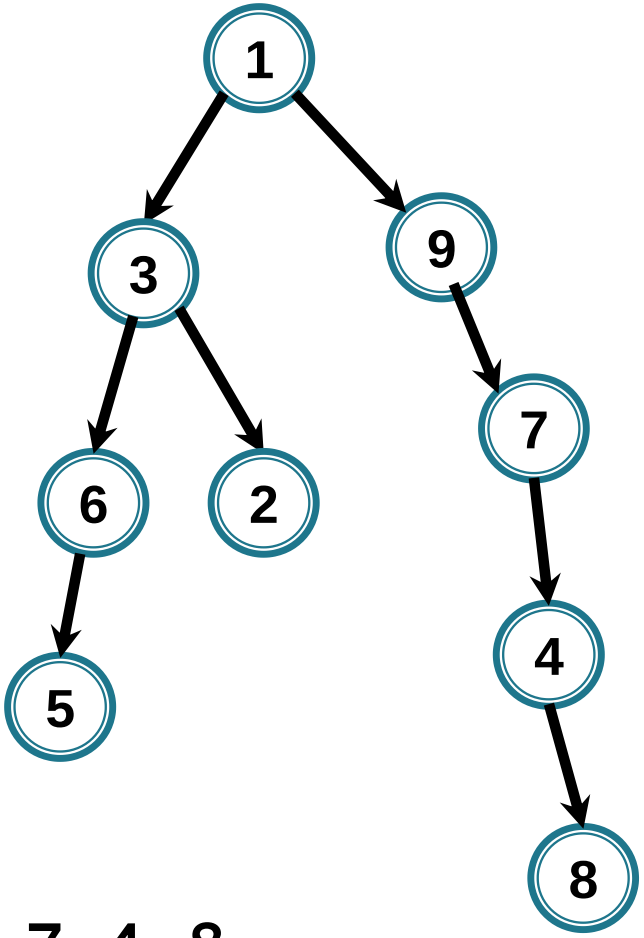

1
3
5
2
6
9
4
7
8


y
x
Nu se poate

Parcurgerea în adâncime

- Exemplu – caz orientat:

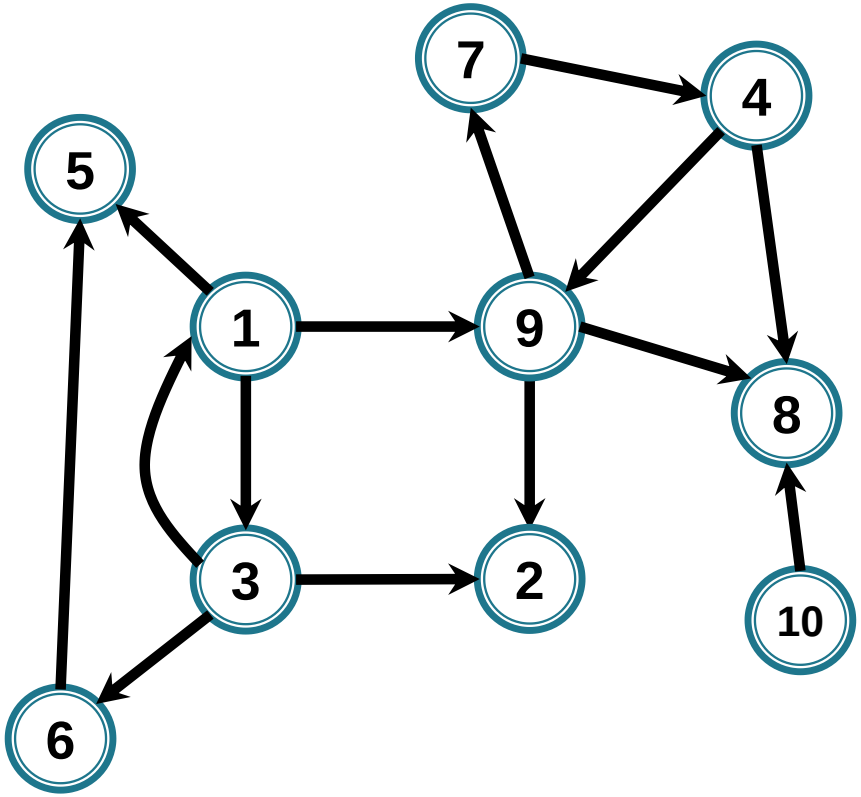




DF(1): 1, 3, 6, 5, 2, 9, 7, 4, 8

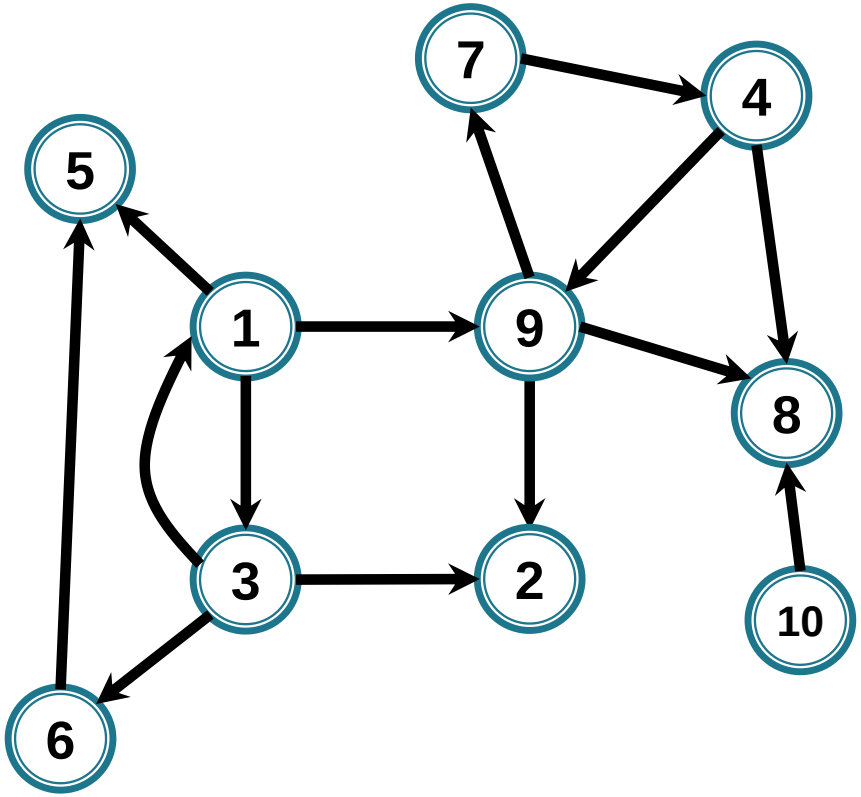
Parcurgerea în adâncime

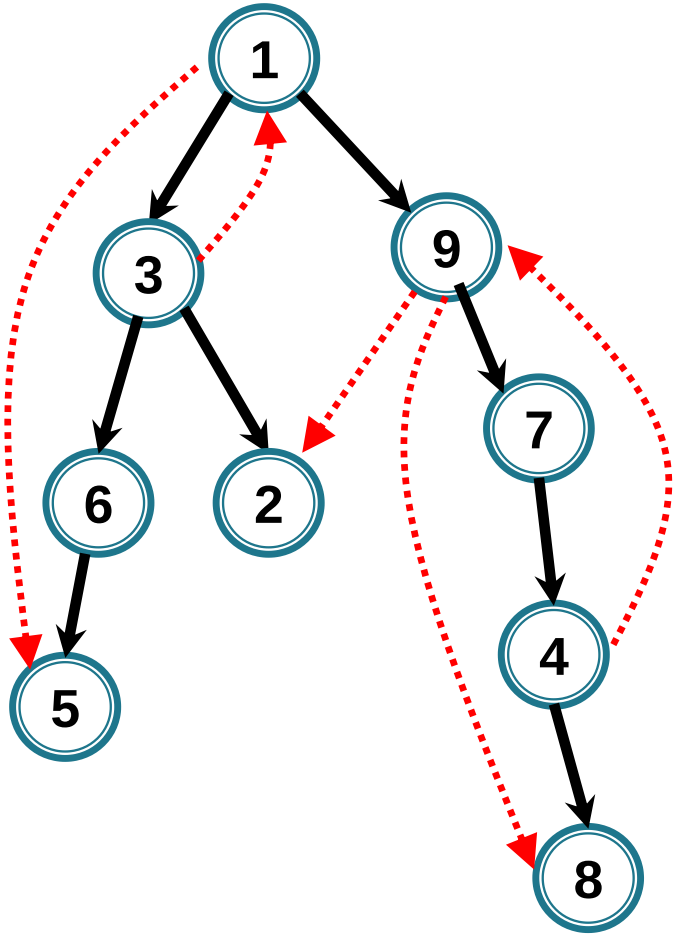
- Exemplu – caz orientat:



Parcurgerea în adâncime

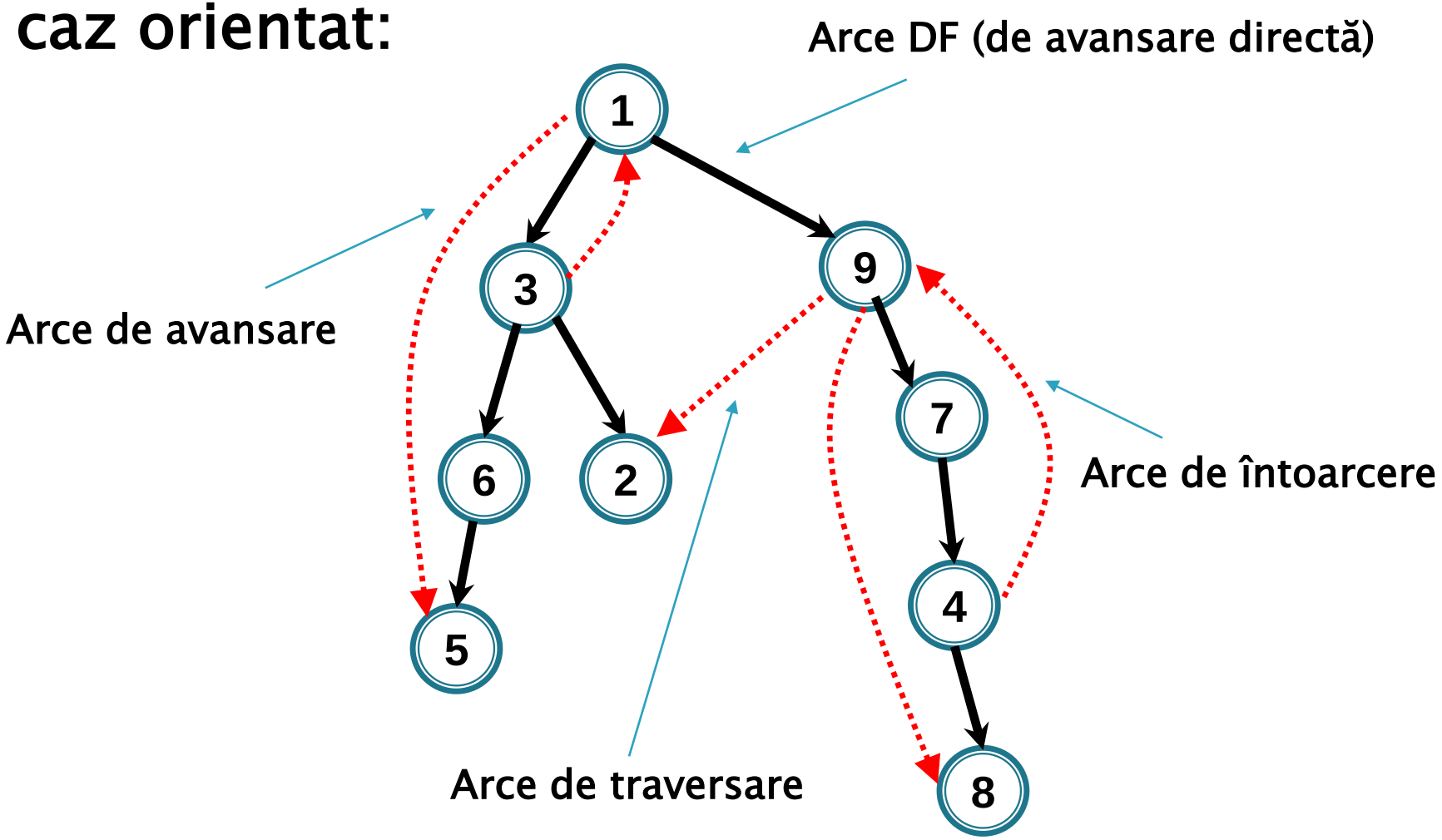
- Exemplu – caz orientat:





Parcurgerea în adâncime

caz orientat:



Doar arcele de întoarcere închid circuite

```
void df(int x){
     //incepe explorarea varfului x
     viz[x]=1;
     for(int i=0;i<la[x].size();i++){
        int y=la[x][i];
        if (viz[y]==0){
              tata[y]=x;
              d[y] = d[x]+1; //nivel, nu distanta
              df(y);
        }
     //s-a finalizat exlorarea varfului x
}

Apel:
     df(s)
```

x alb

x gri

x negru

Parcurgerea în adâncime

- Culorile nodurilor după ce 7 devine vârf curent: