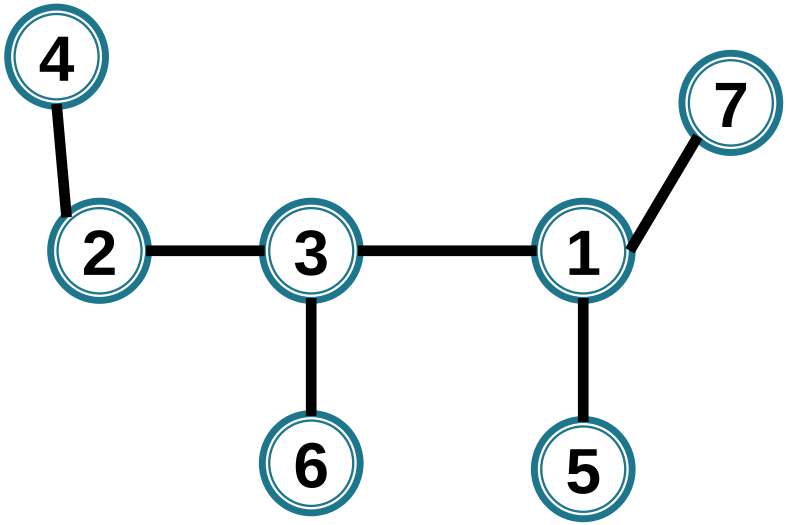
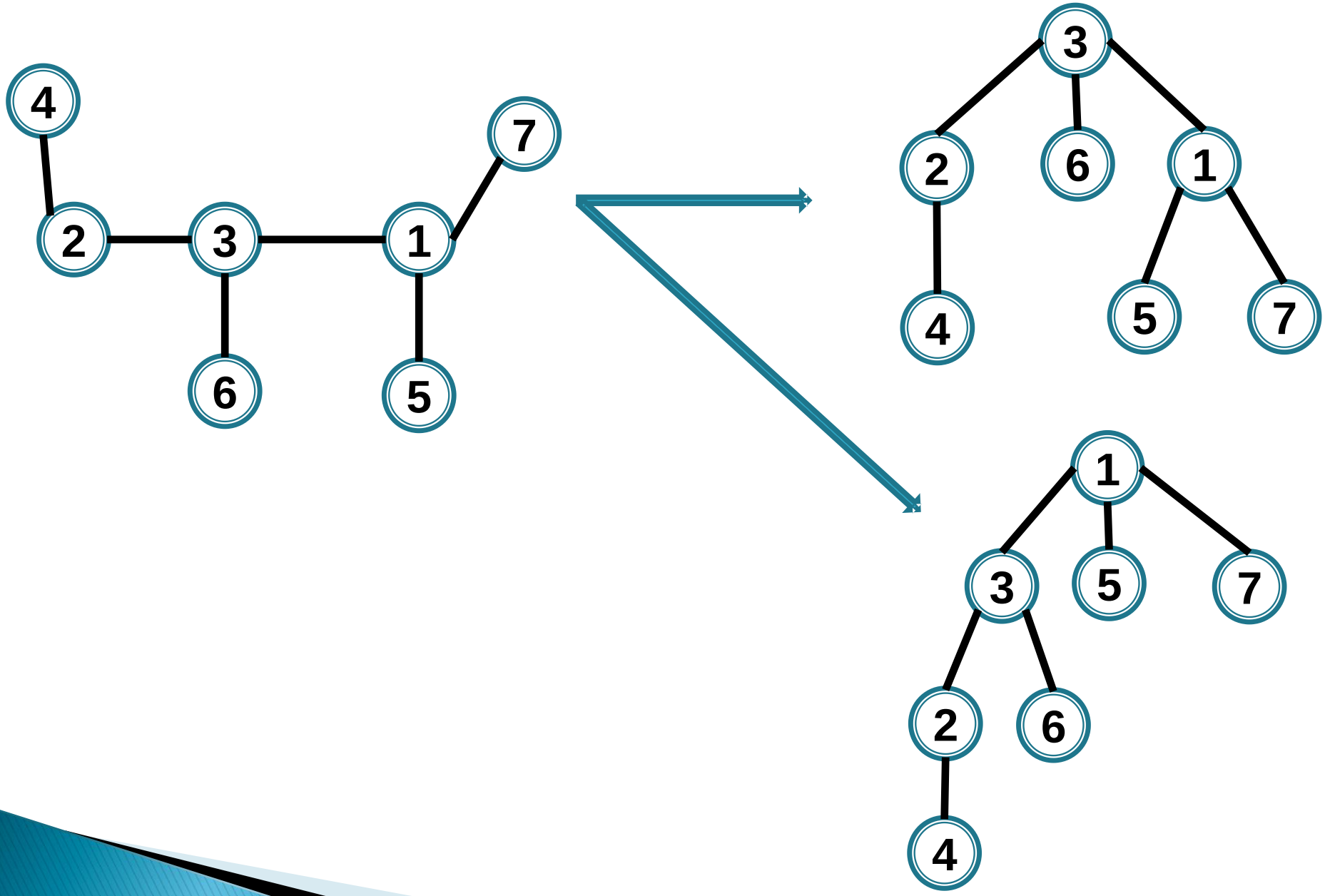
Arbori cu rădăcină


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

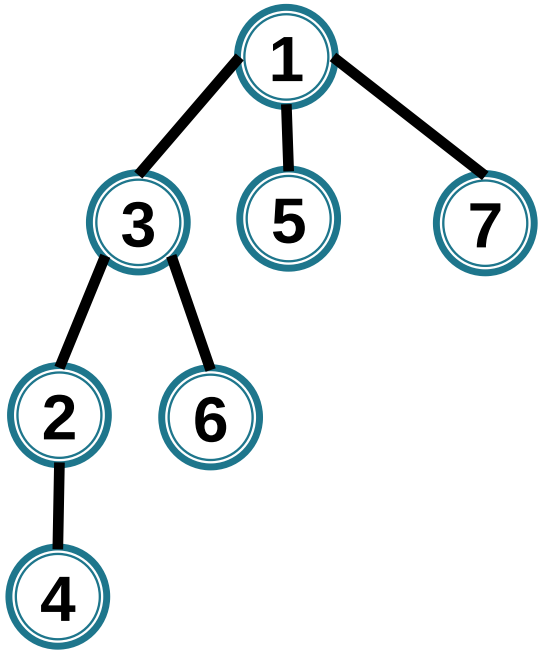
- **Noțiuni**
  - **Arbore cu rădăcină**
    - După fixarea unei rădăcini, arborele se aşează pe niveluri
    - Nivelul unui nod v, niv[v] = distanţa de la rădăcină la nodul v
    - În arborele cu rădăcină există muchii doar între niveluri consecutive

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

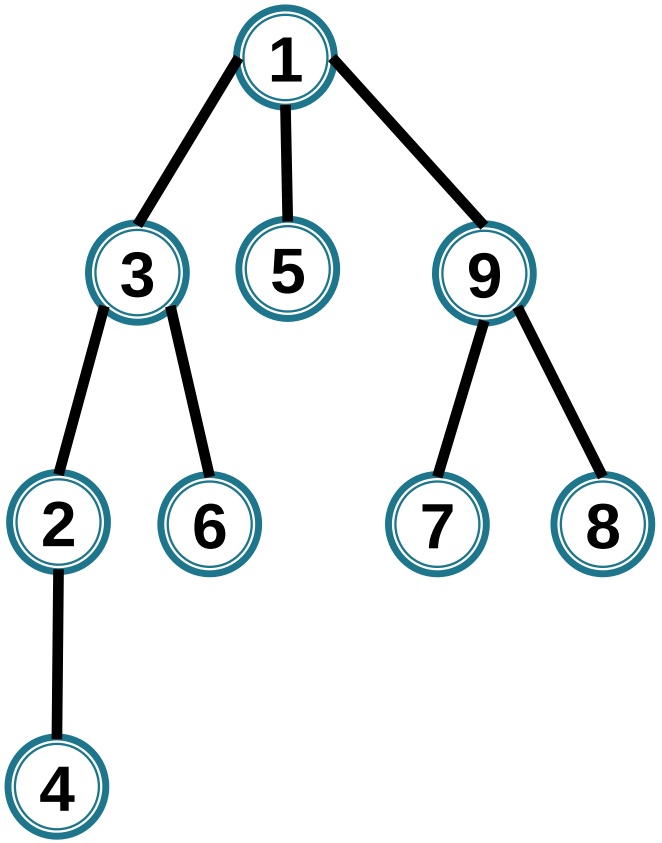
- **Fiu:** fii lui 3 sunt 2 şi 6
- **Tată:** 1 este tatăl lui 7
- **Ascendent:** ascendenţii lui 6 sunt 3 şi 1
- **Descendent:** descendenţii lui 3 sunt 2, 6 şi 4
- **Frunză:** frunzele arborelui sunt 4, 6, 5 şi 7



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

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];
  }
}
```

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

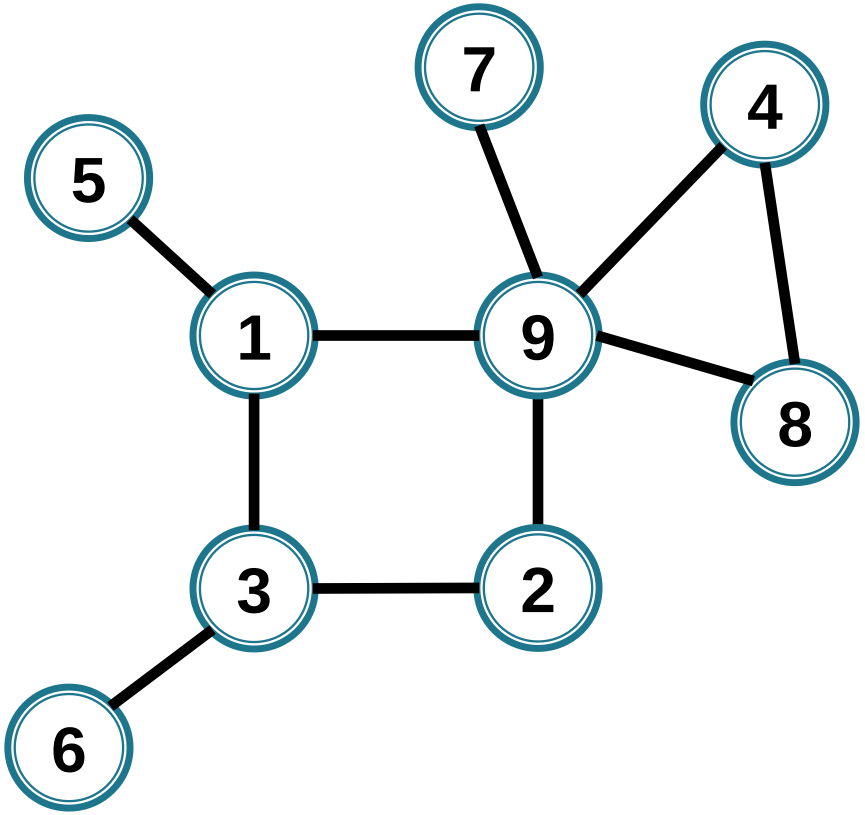
Parcurgerea grafurilor

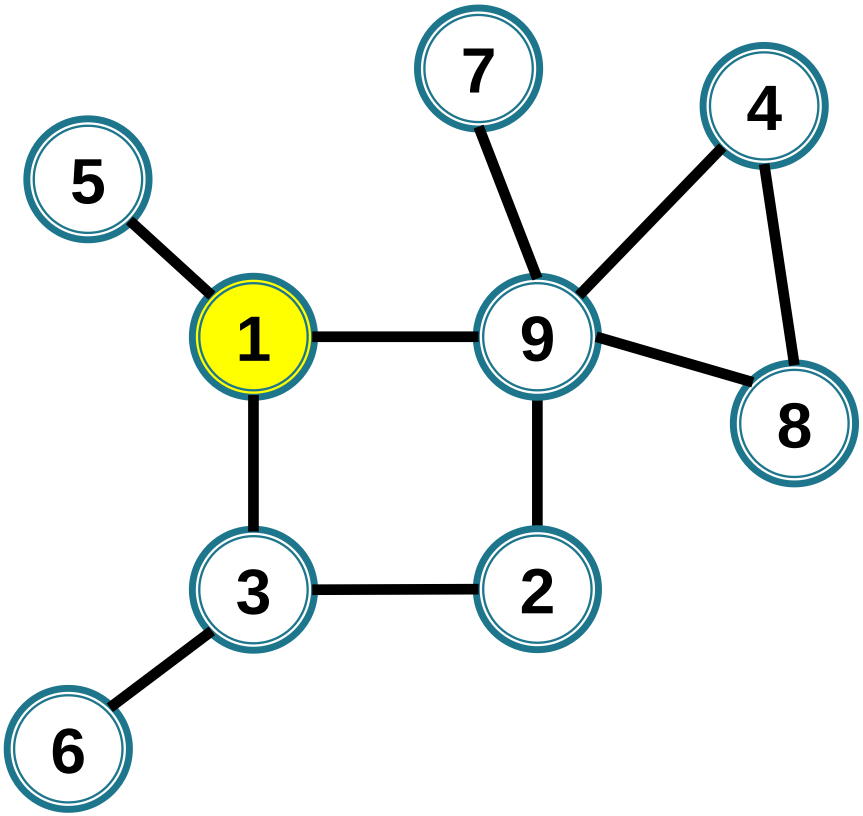
- **Parcurgerea în lățime:** se vizitează
  - vârful de start **s**
  - vecinii acestuia
  - vecinii nevizitați ai acestora

etc

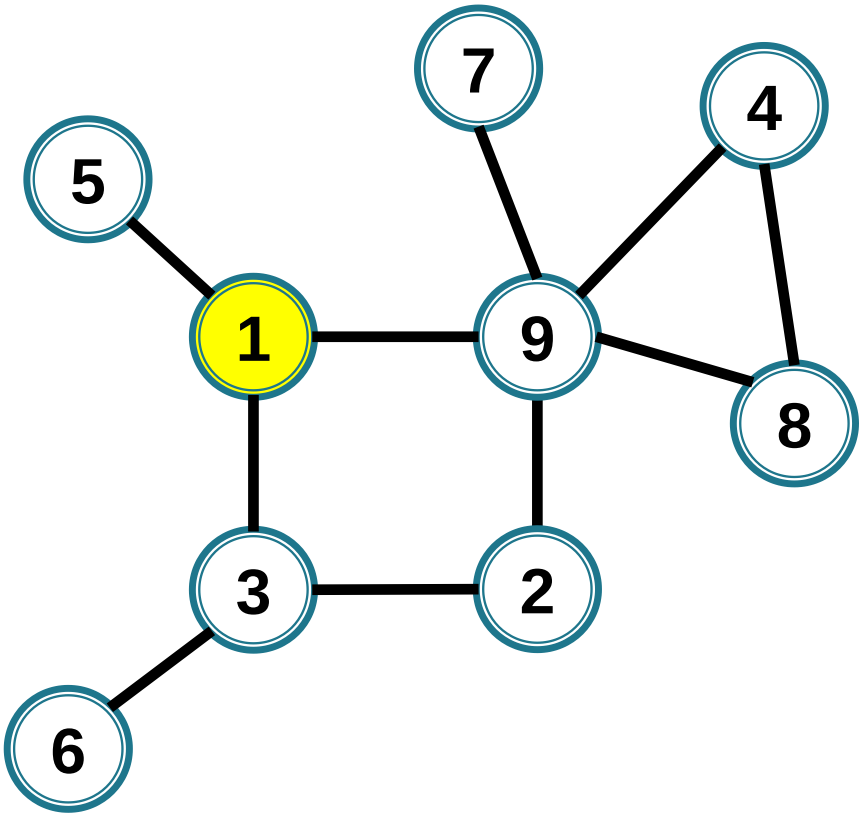
Parcurgerea în lățime

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


7
4
5
1
9
8
3
2
6

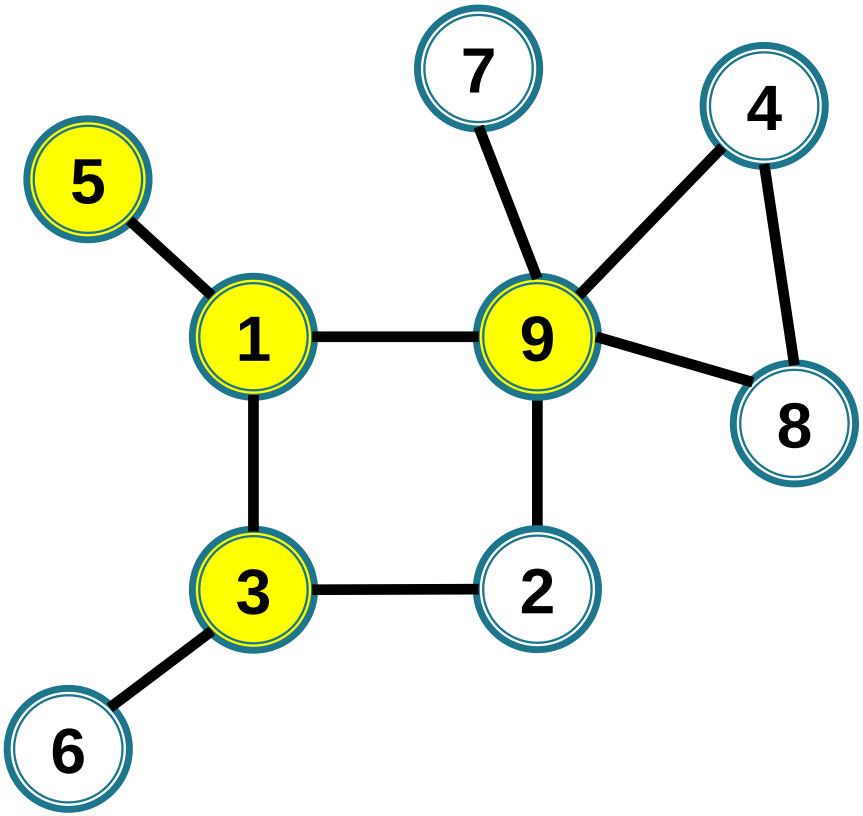


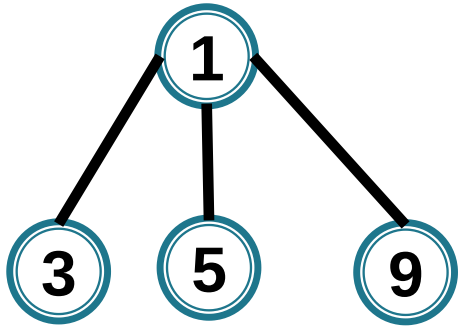
1



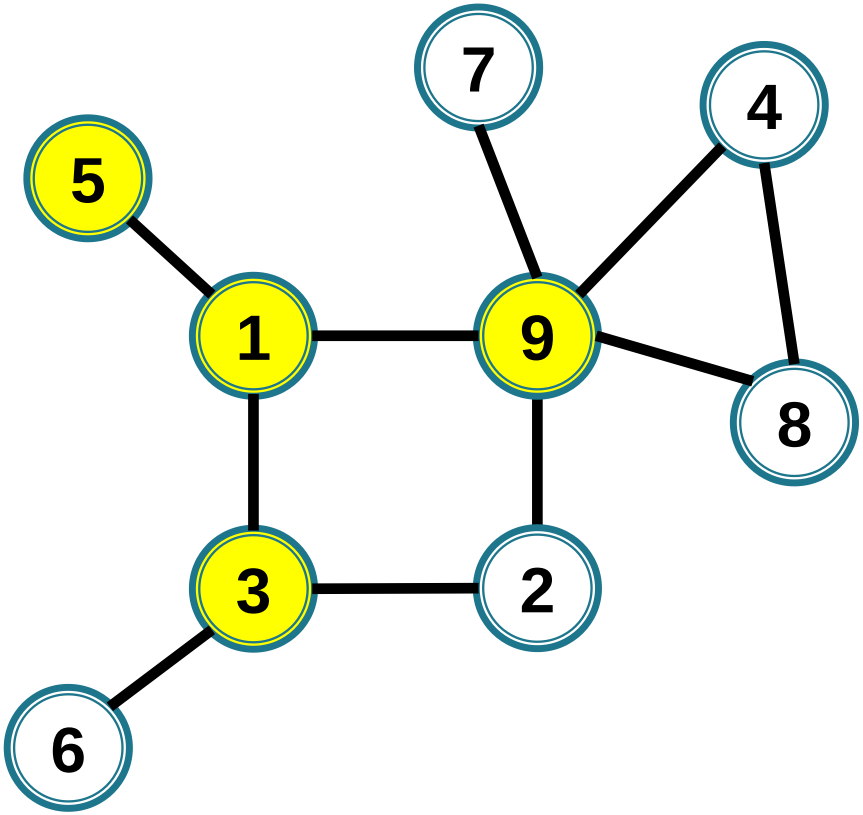
1

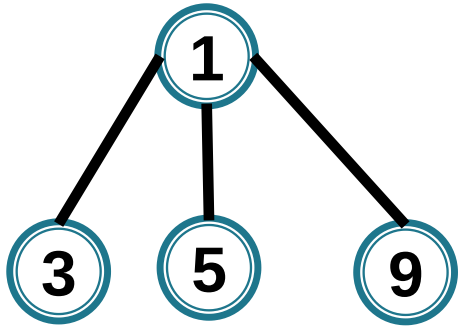
1



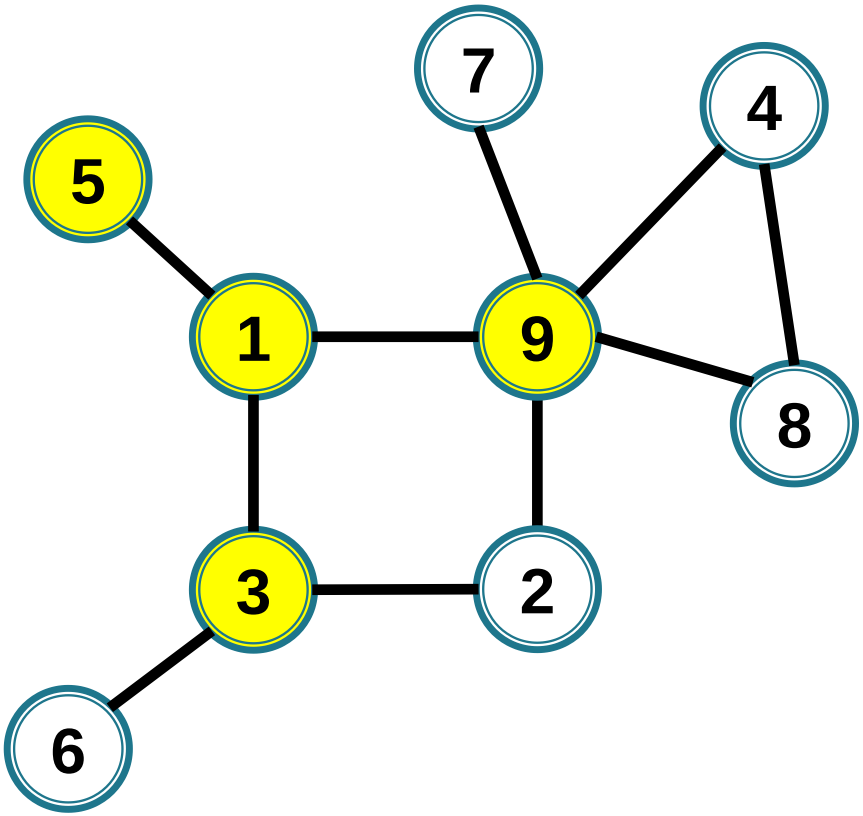


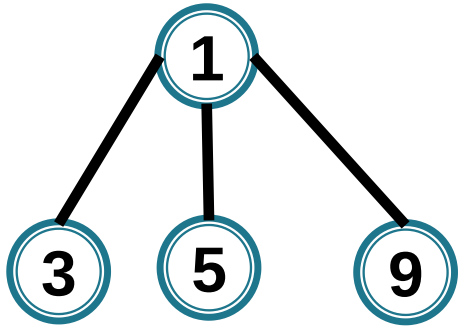
1 3 5 9



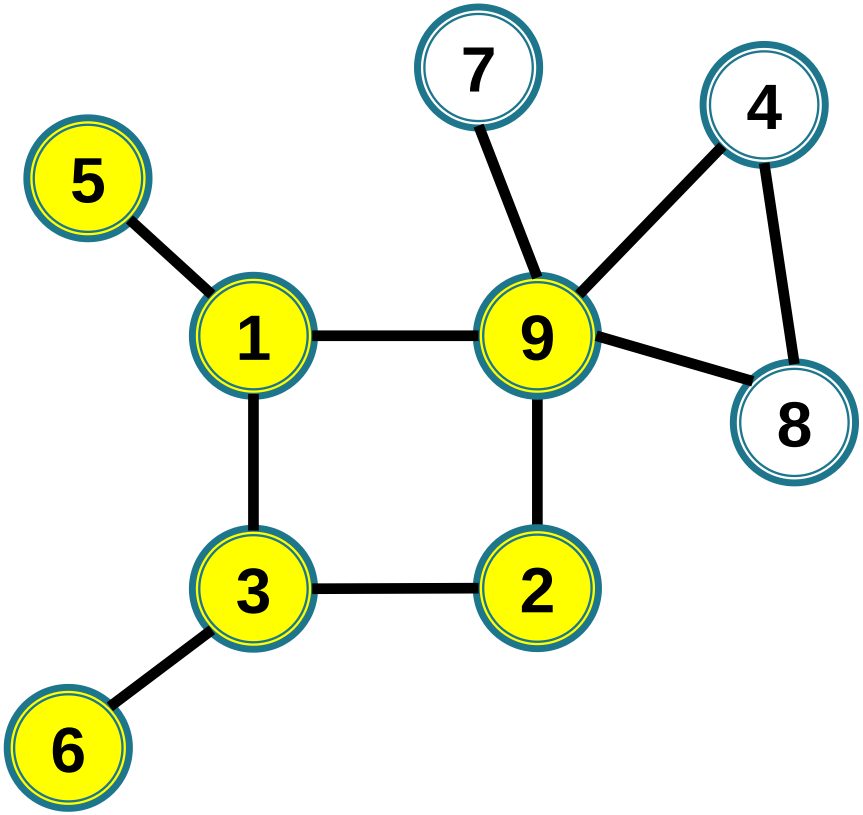


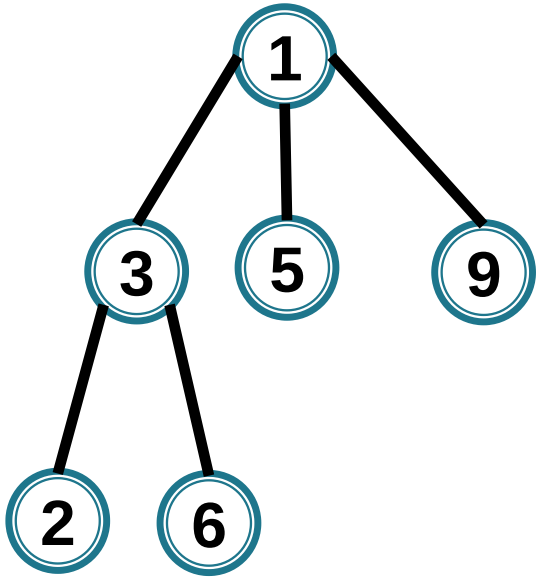
1 3 5 9



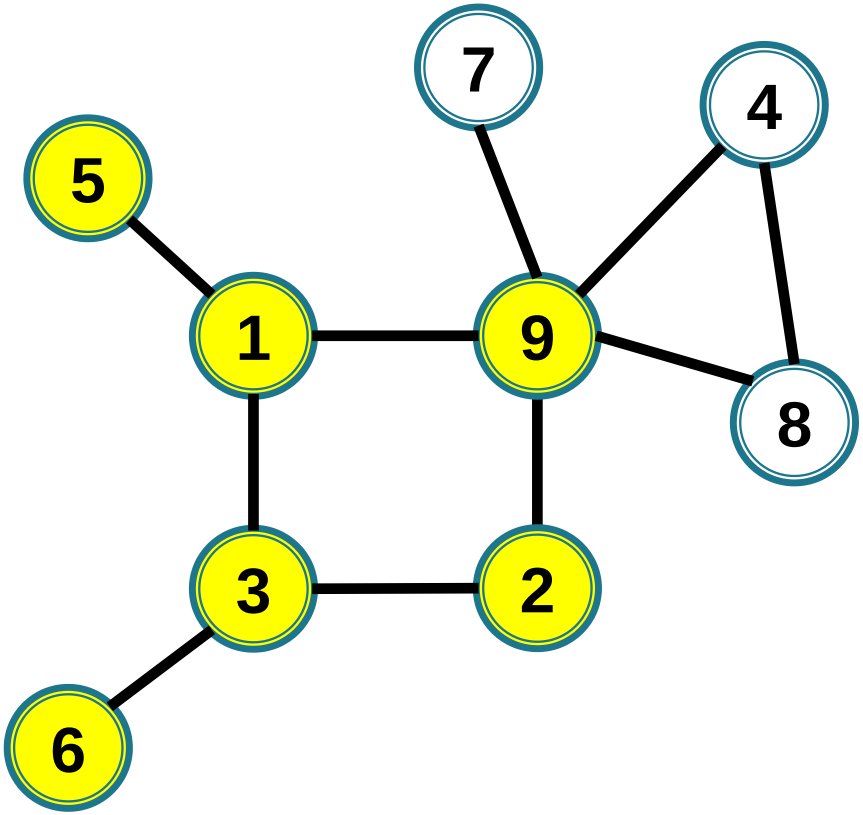


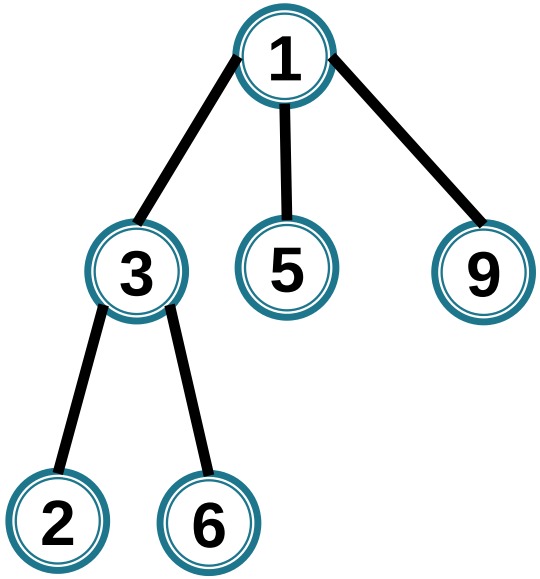
1 3 5 9



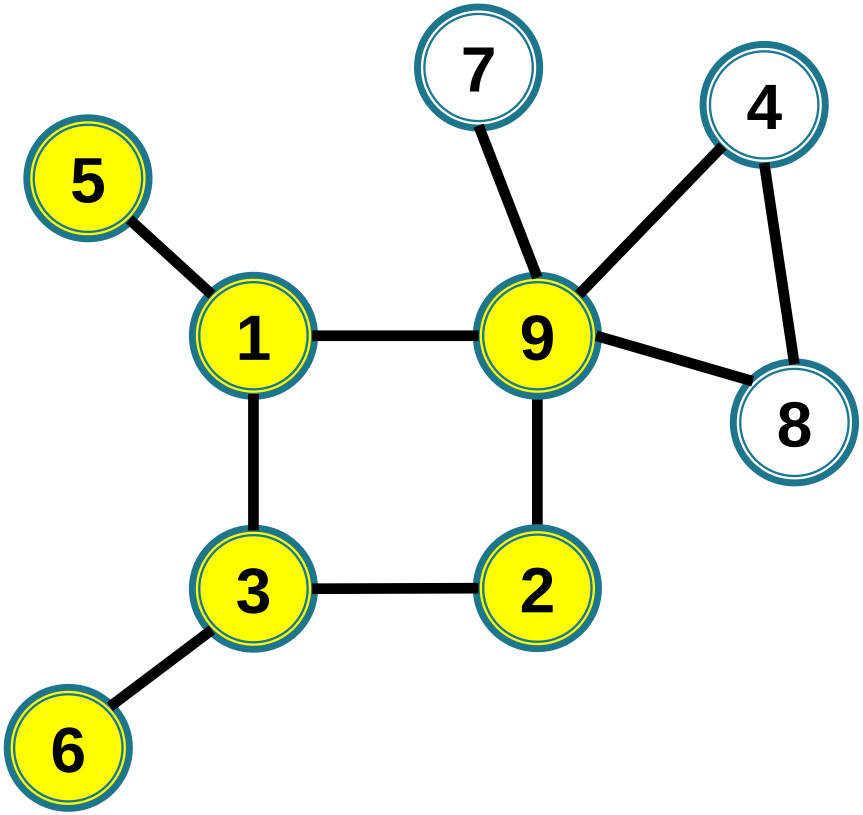


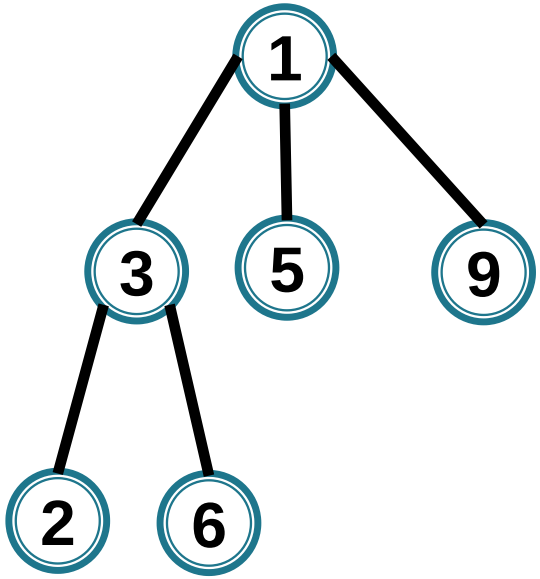
1 3 5 9 2 6



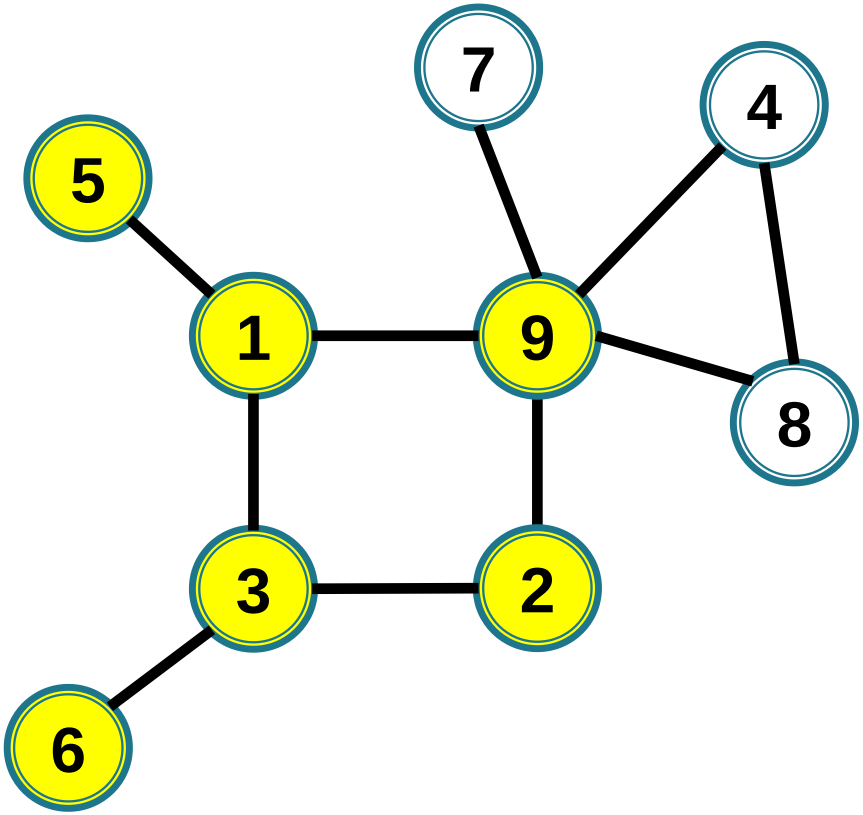


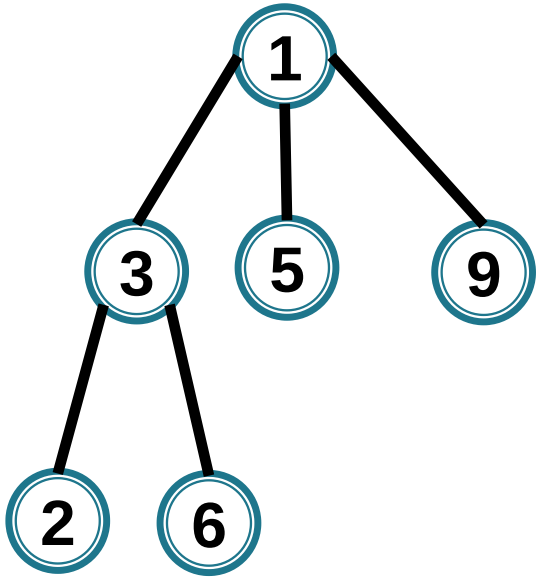
1 3 5 9 2 6



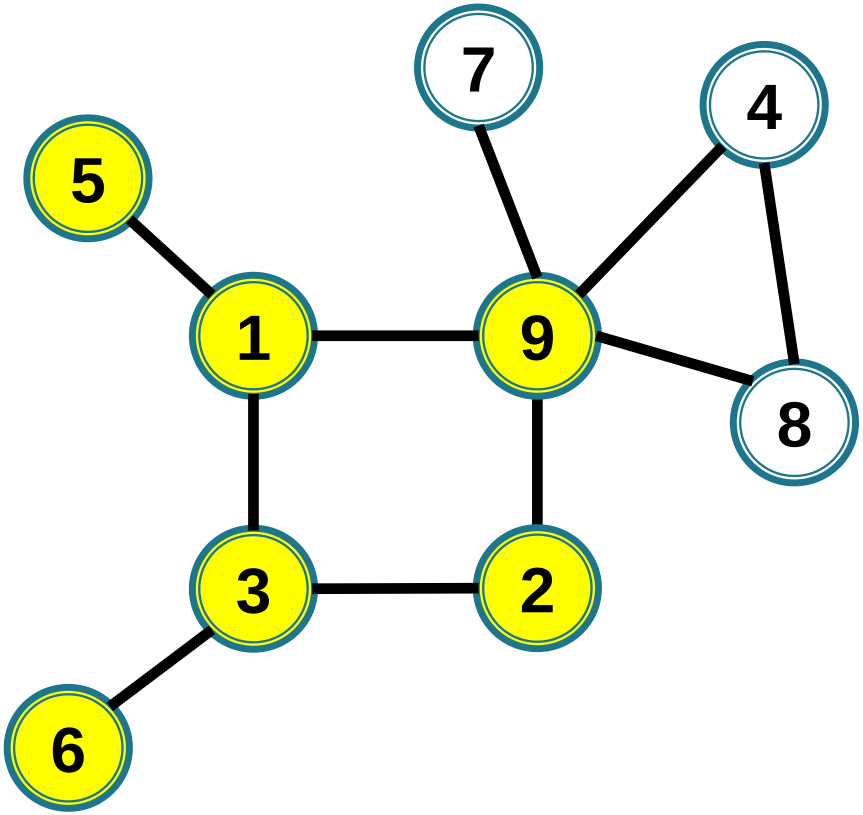


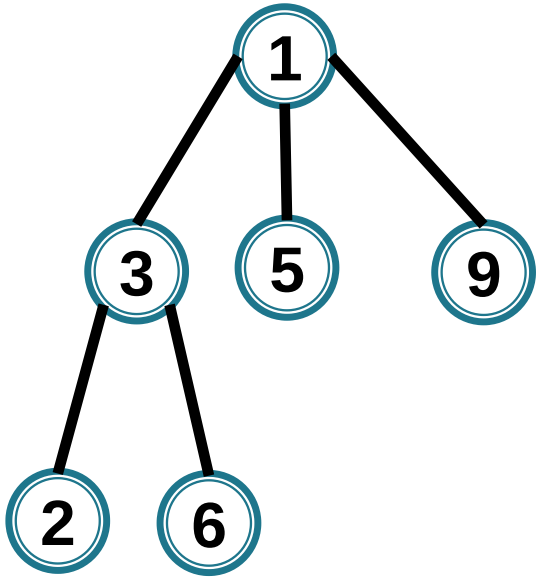
1 3 5 9 2 6



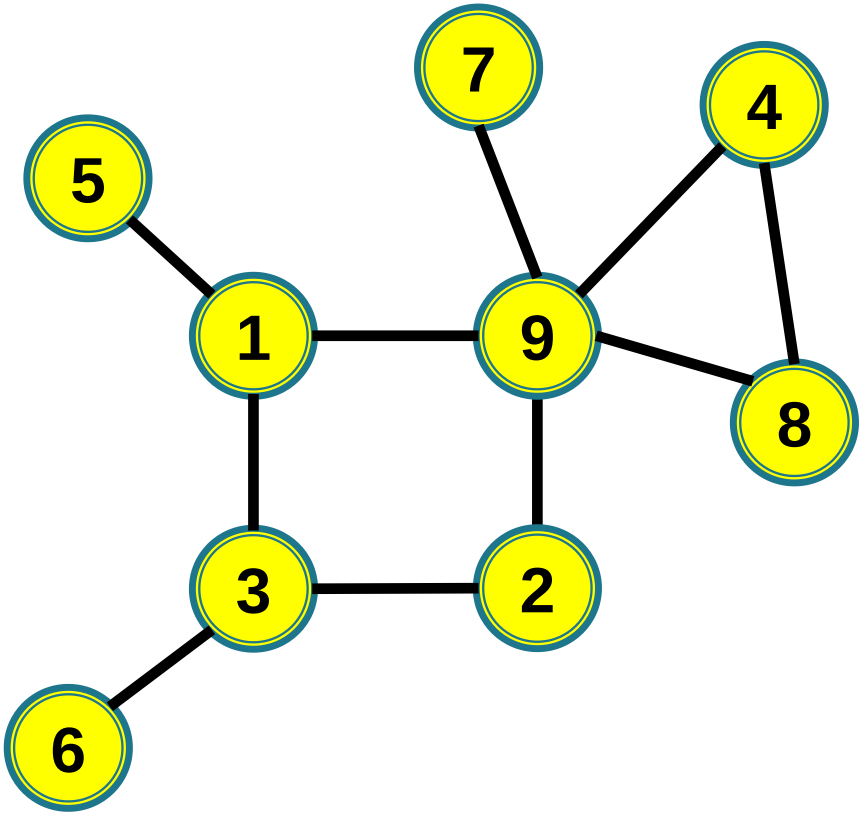


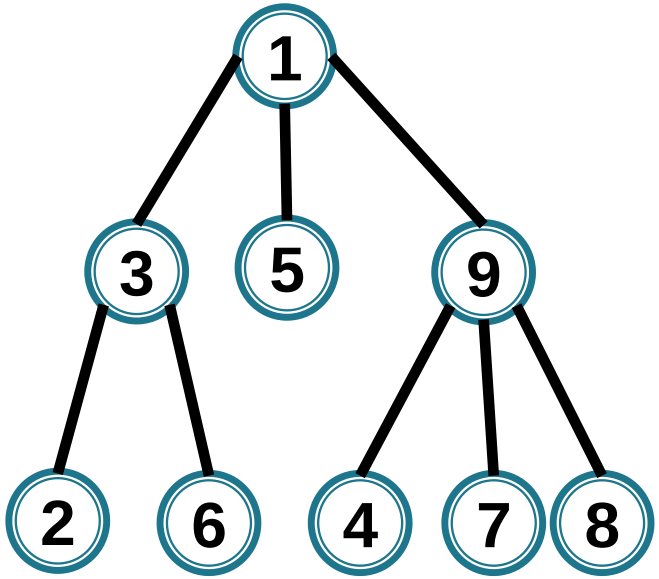
1 3 5 9 2 6



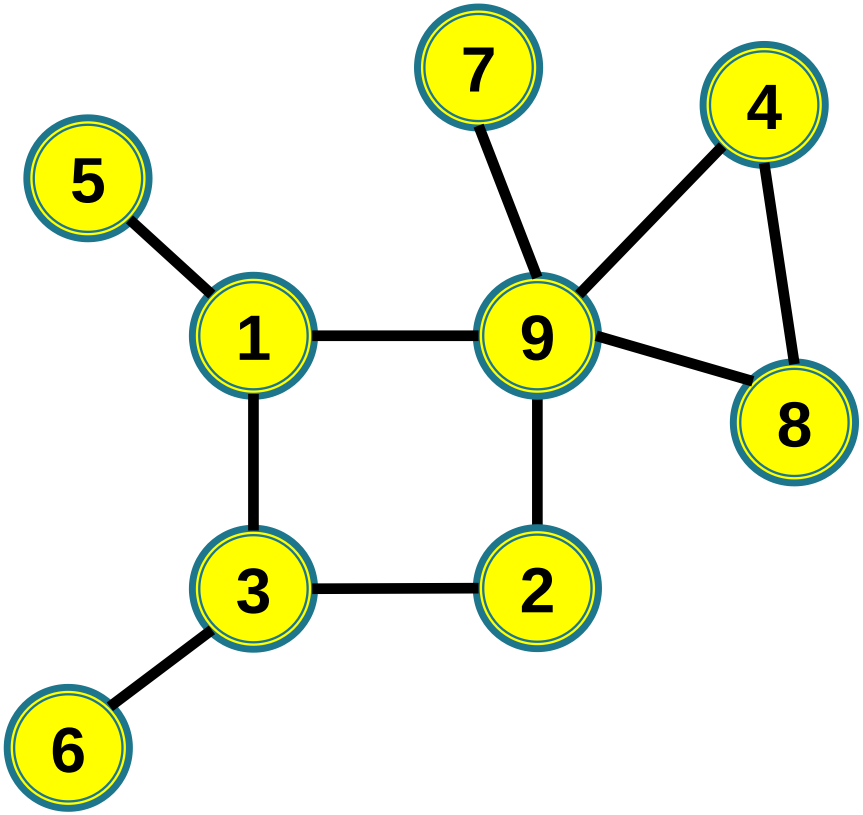


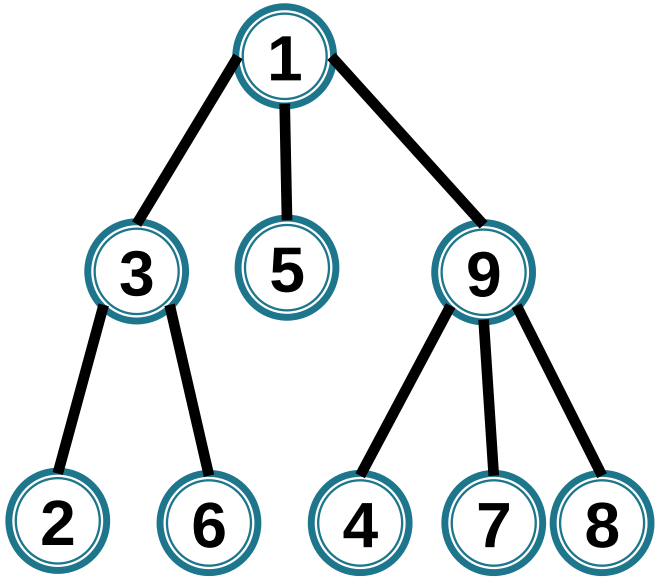
1 3 5 9 2 6



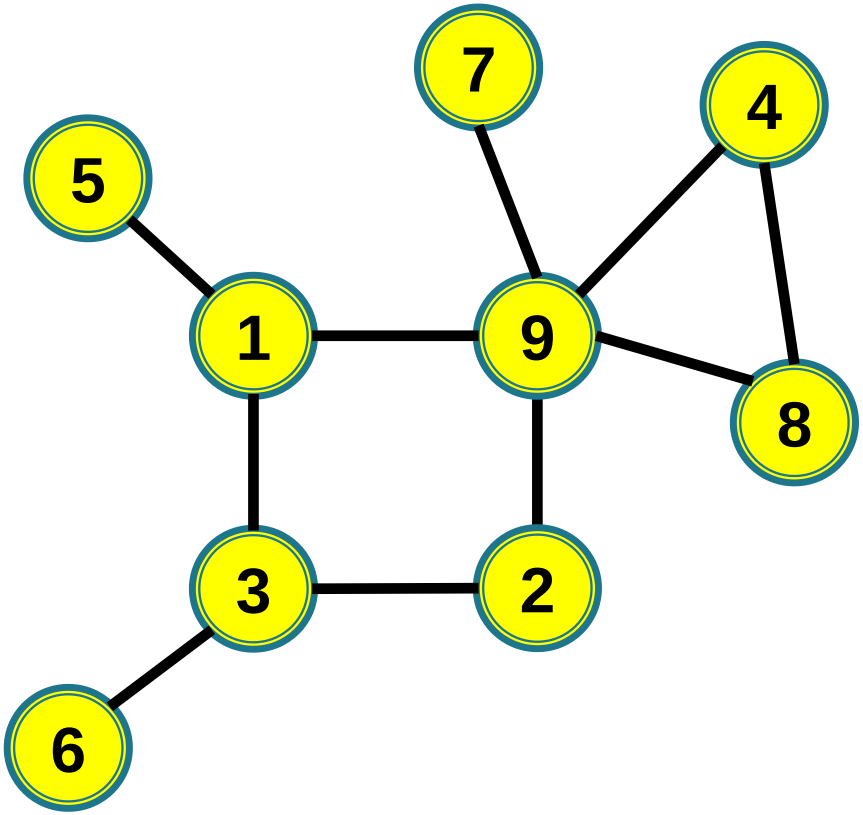


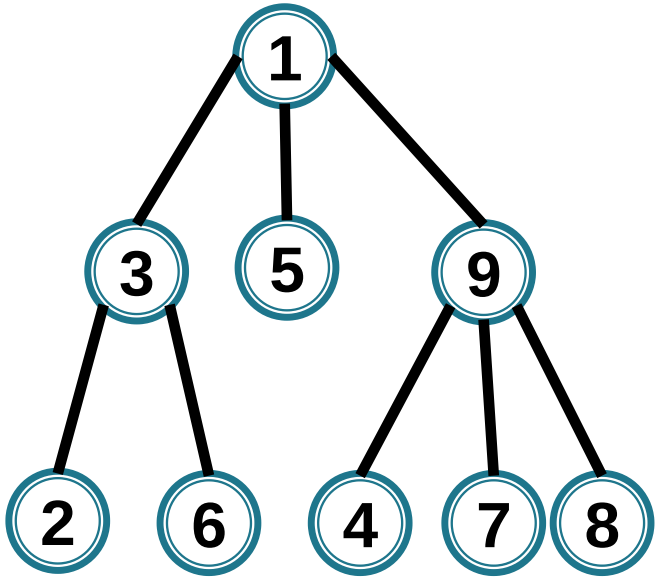
1 3 5 9 2 6 4 7 8



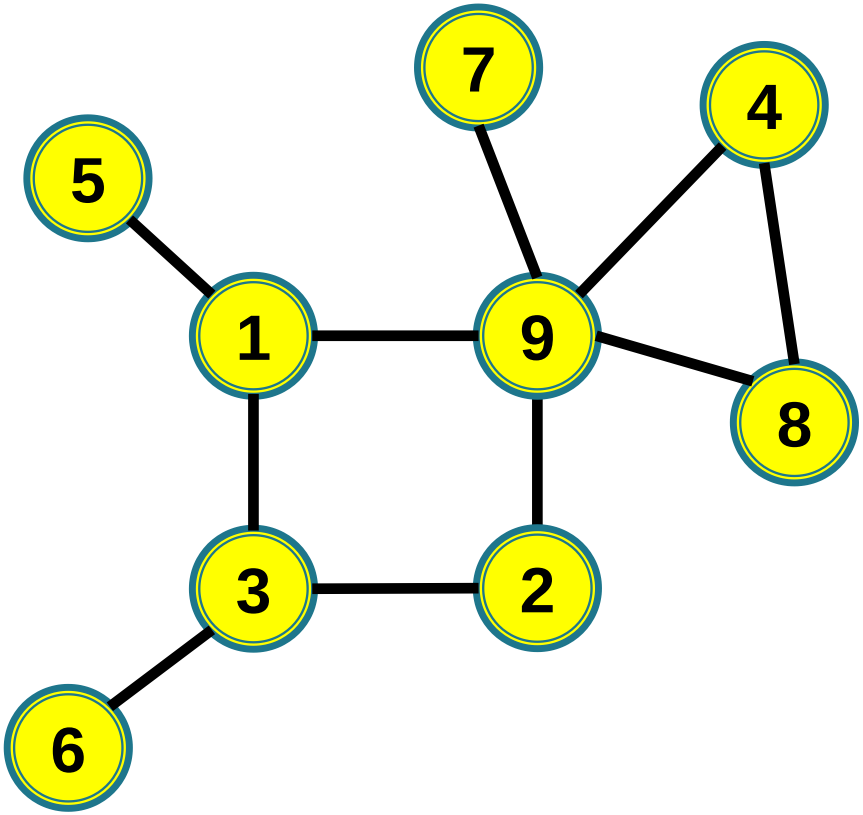


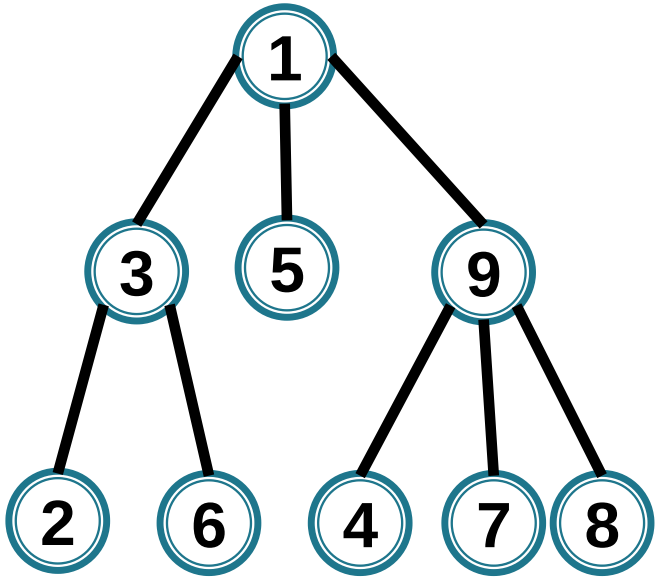
1 3 5 9 2 6 4 7 8



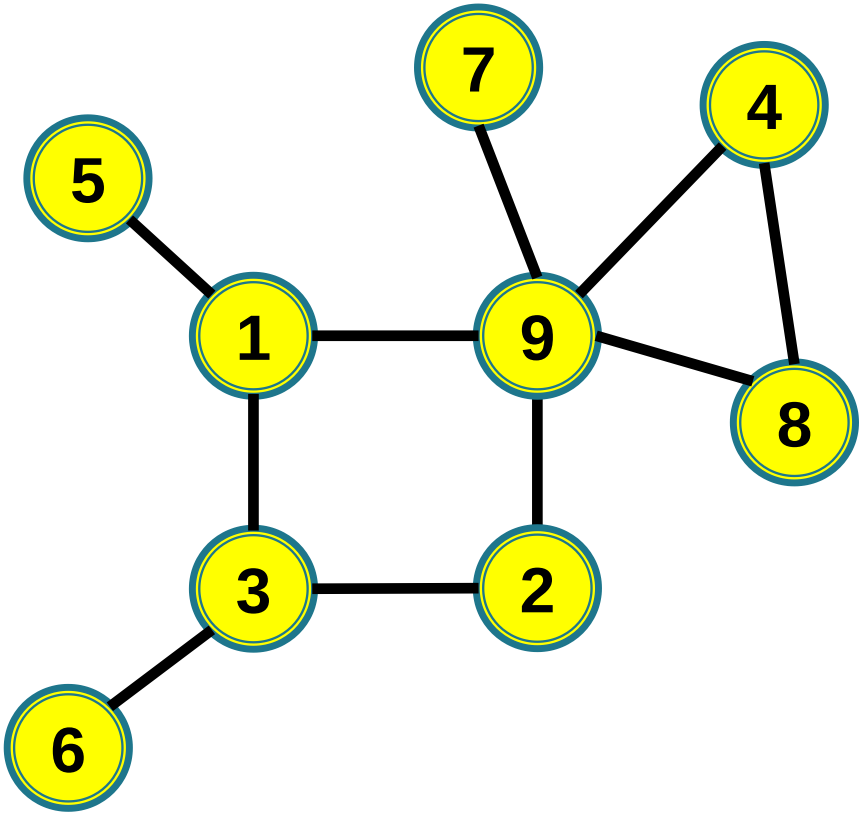


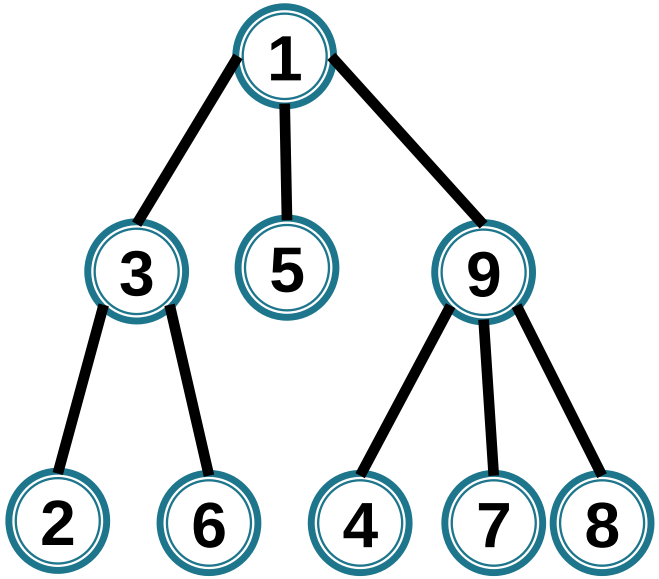
1 3 5 9 2 6 4 7 8



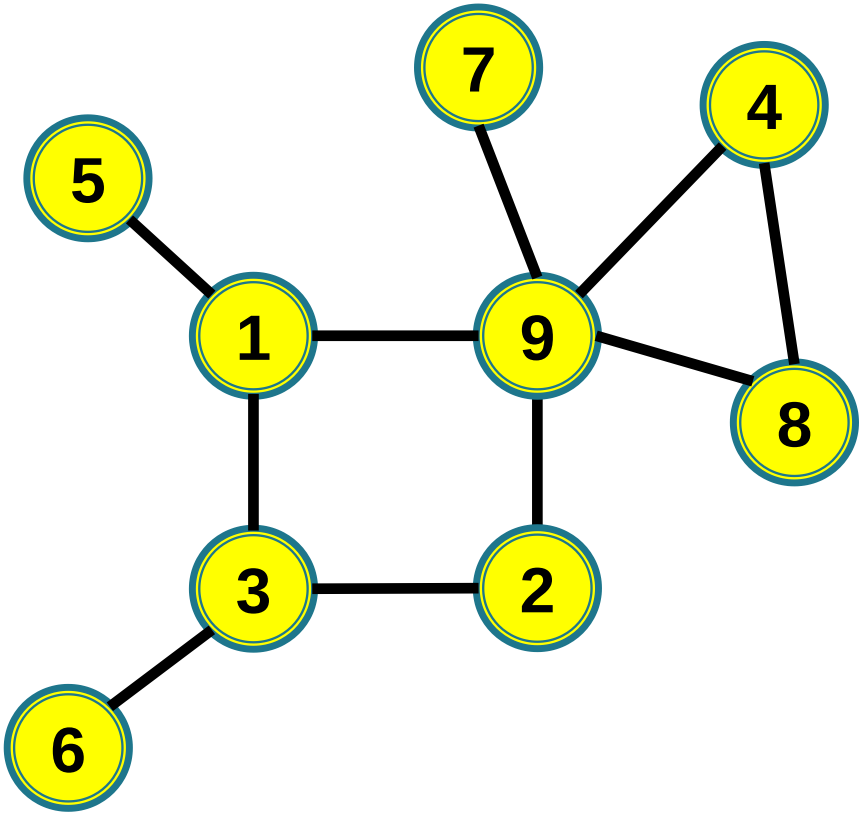


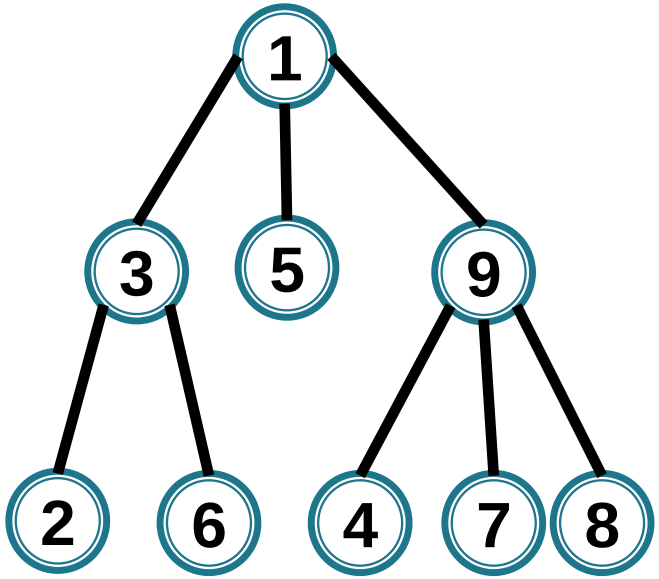
1 3 5 9 2 6 4 7 8



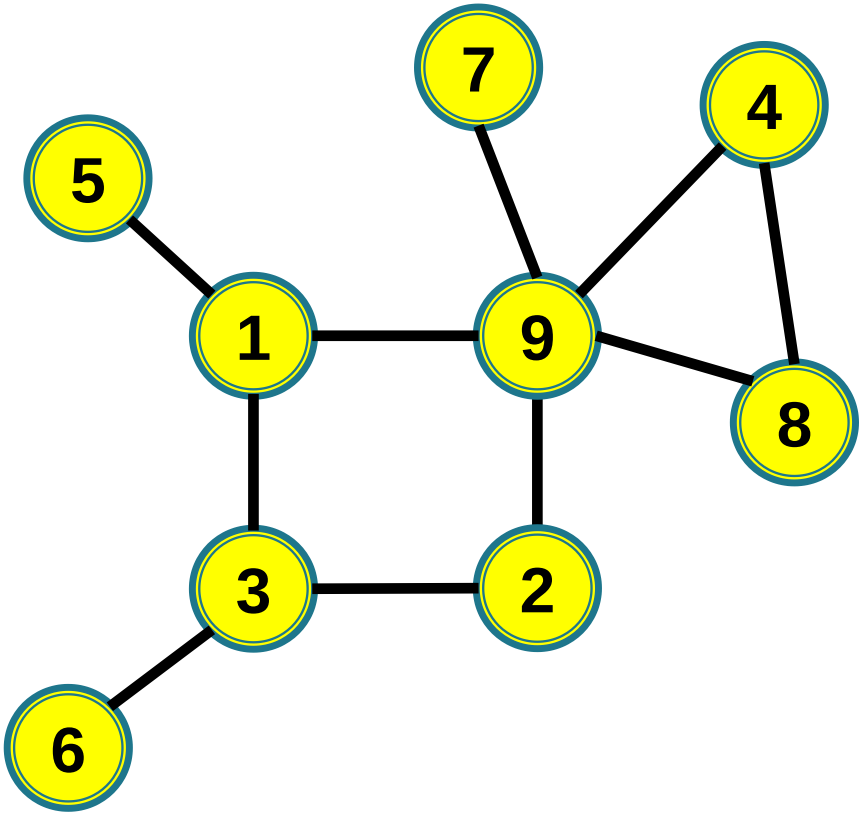


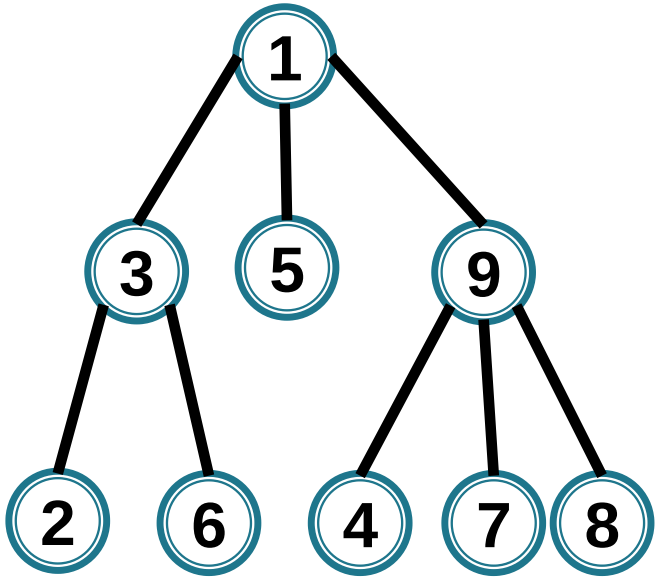
1 3 5 9 2 6 4 7 8



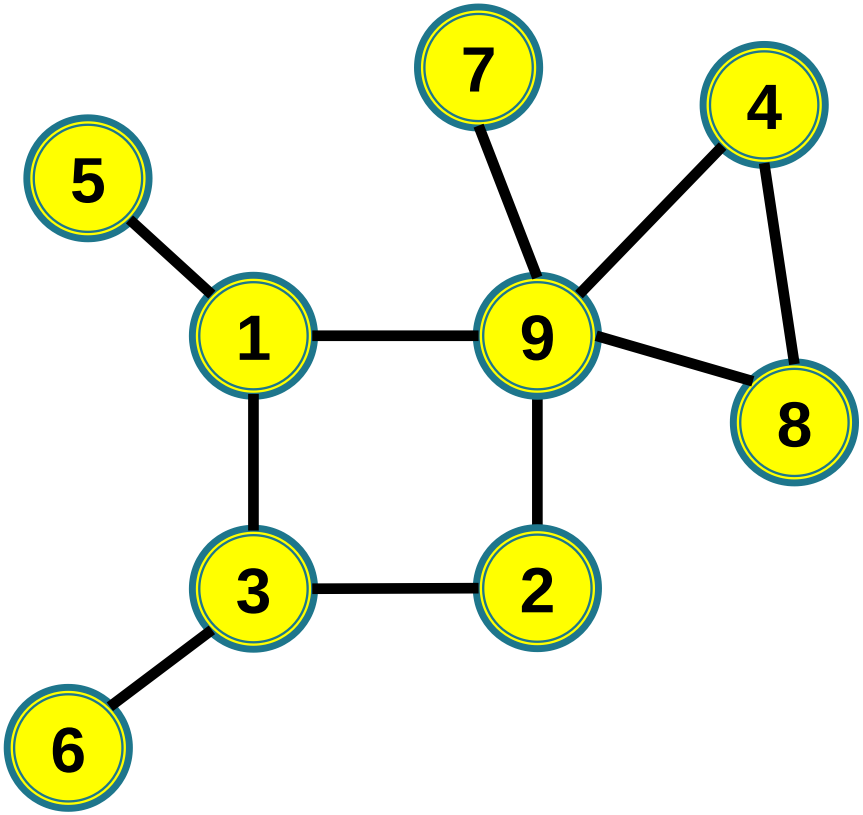


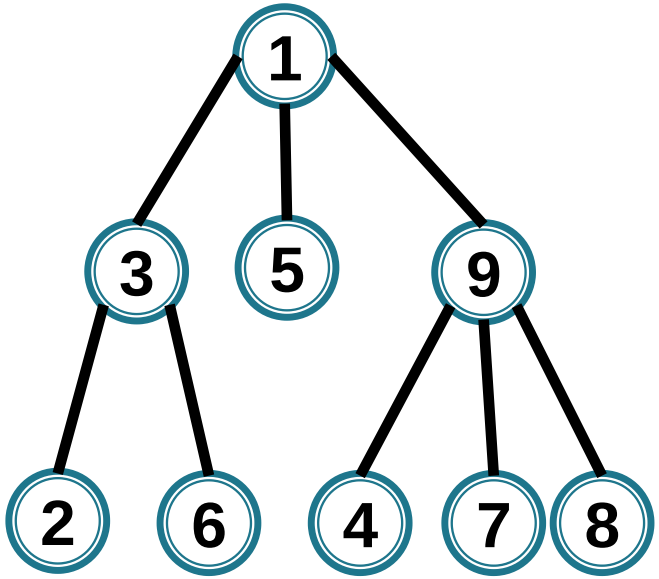
1 3 5 9 2 6 4 7 8



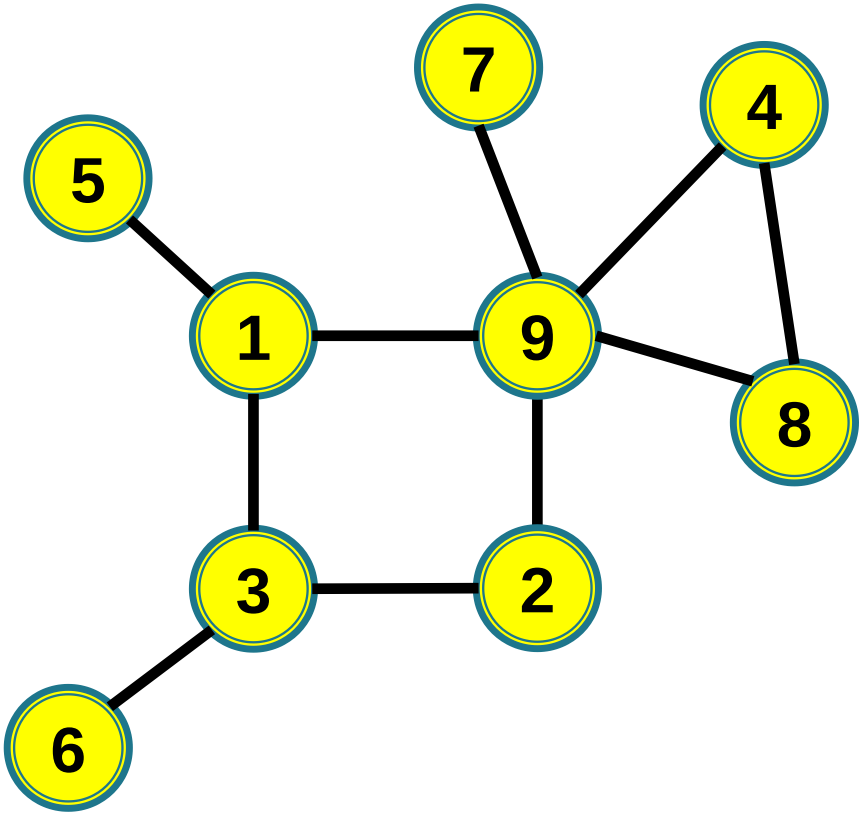


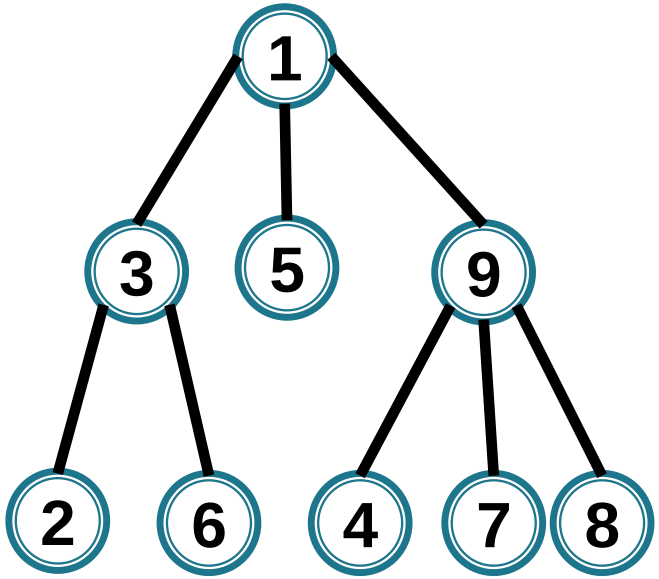
1 3 5 9 2 6 4 7 8



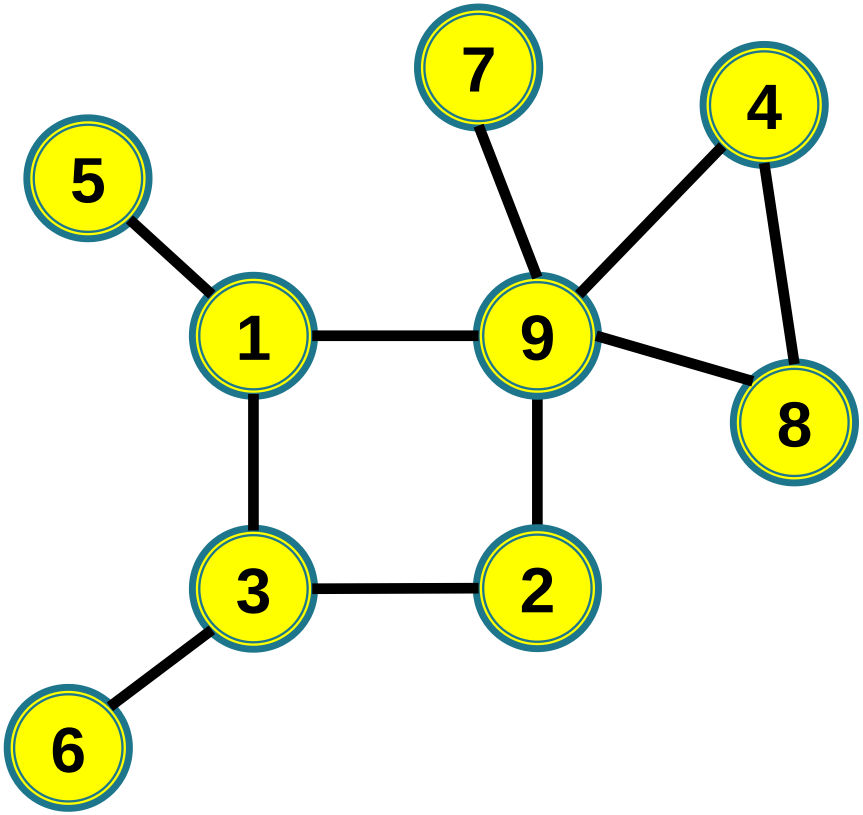


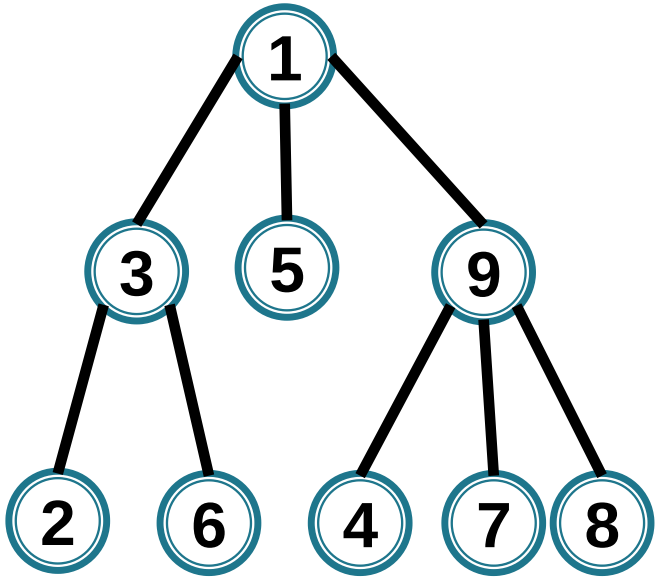
1 3 5 9 2 6 4 7 8



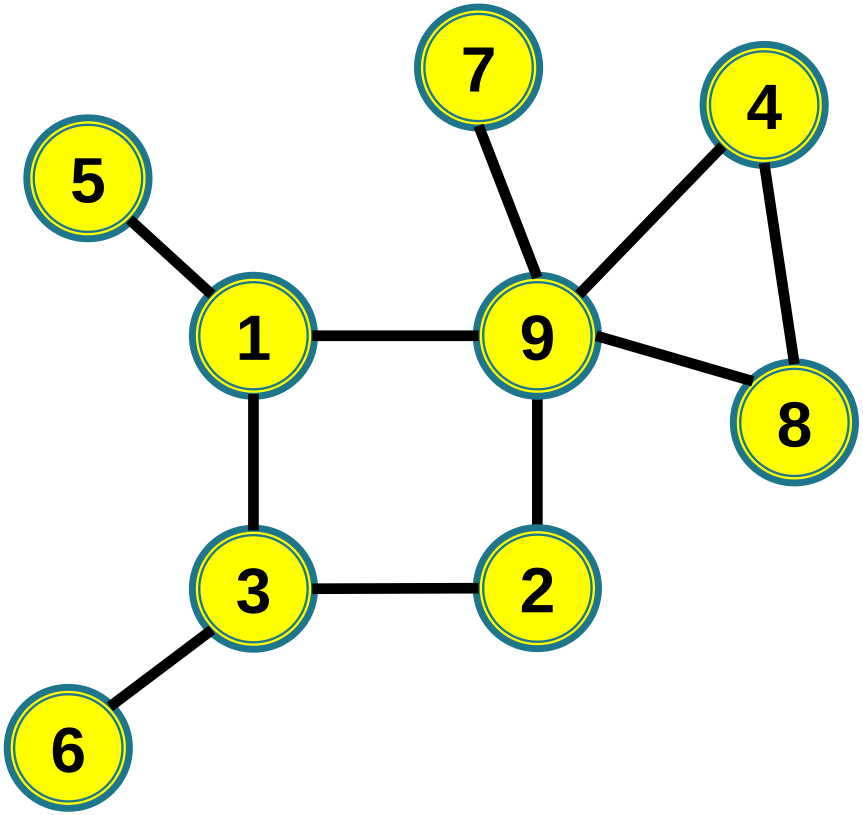


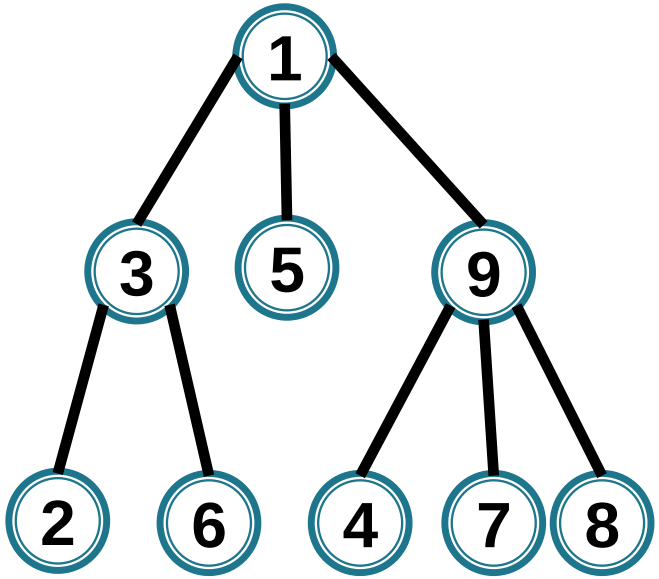
1 3 5 9 2 6 4 7 8



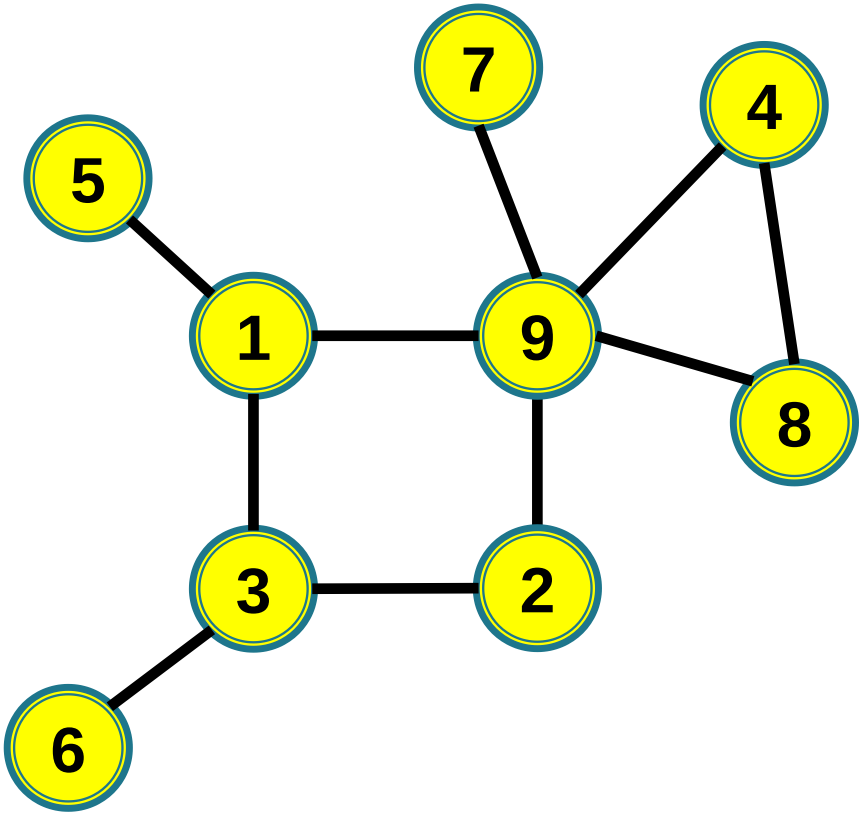


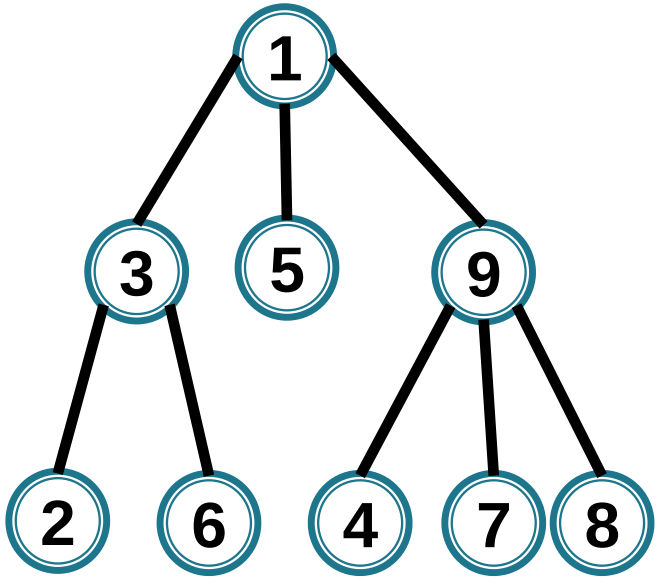
1 3 5 9 2 6 4 7 8





1 3 5 9 2 6 4 7 8





1 3 5 9 2 6 4 7 8

Parcurgerea în lățime

- Muchiile folosite pentru a descoperi vârfuri noi formează un **arbore** (numit **arbore BF)**

Pseudocod

Parcurgerea în lățime

- Informatii necesare (vectori):

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

Parcurgerea în lățime

- Informatii necesare (vectori):

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

Opţional

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

Parcurgerea în lățime

- Informatii necesare (vectori):

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

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**

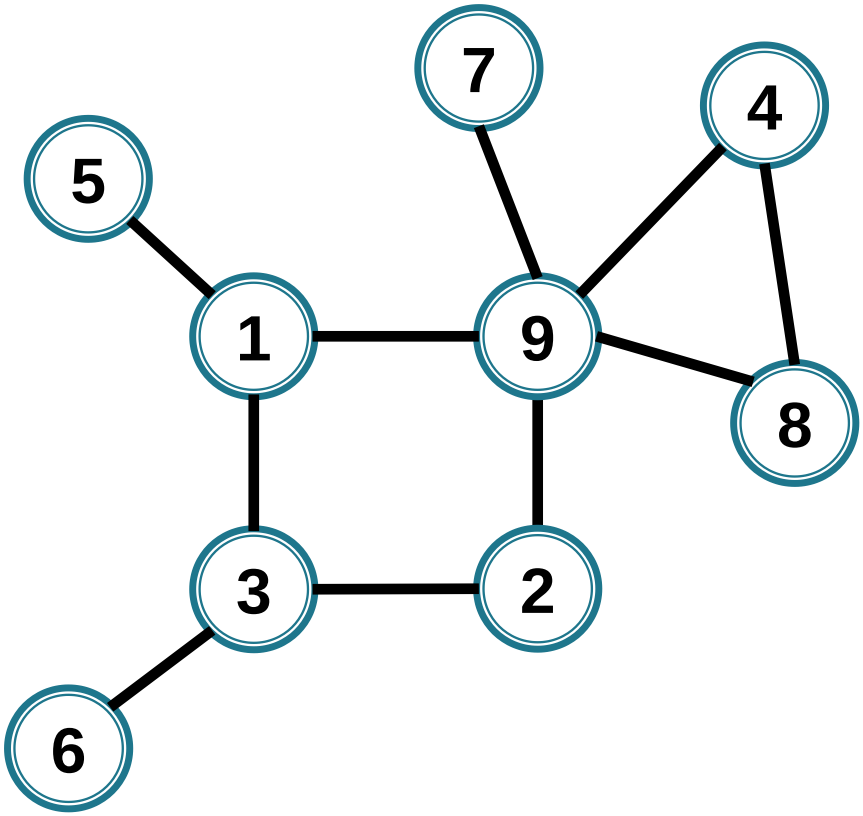
d[j] = d[tata[j]]+1

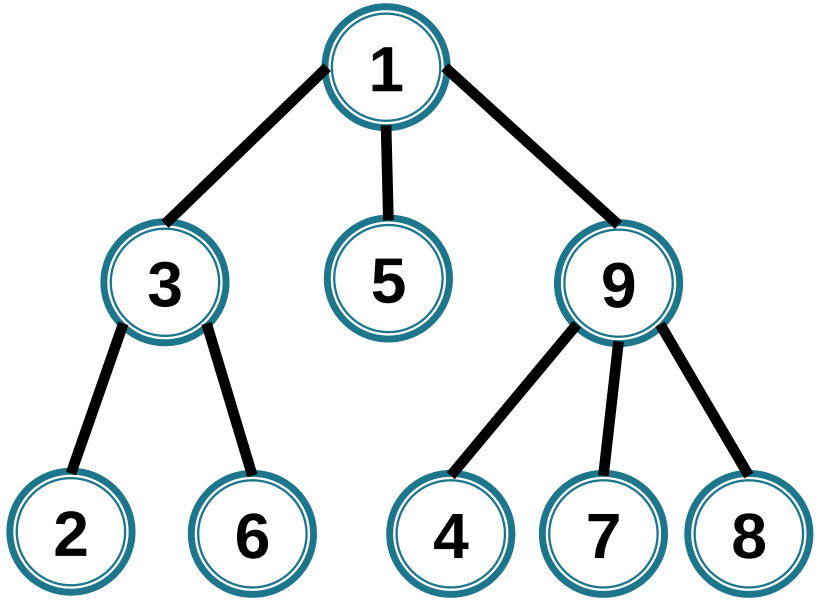
Parcurgerea în lățime

- **Propoziție – Corectitudinea BF**

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

Demonstraţia – după pseudocod


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
```

Implementare

Parcurgerea în lățime

- Coada – vector
  - p = poziţia primului element din coadă
  - u = poziţia ultimului element din coadă
  - Extragerea unui element din coadă:
  - Adăugarea unui element în coadă:

Inițializări

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

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

for(i=1;i<=n;i++) {
        viz[i]=0;
        tata[i]=0;
        d[i]=32000;//d[i]=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 atunci
             adauga(j, C)
             viz[j] ← 1
             tata[j] ← i
             d[j] ← d[i]+1
```

```
void bf(int s){
    int p,u,i,j;
```

```
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
```

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

```
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
```

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

```
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
```

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

```
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
```

```
void bf(int s){
    int p,u,i,j;
    p = u = 1;  c[1]=s;
    viz[s]=1; d[s]=0;
    while(p<=u){
      i=c[p]; p=p+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 atunci
             adauga(j, C)
             viz[j] ← 1
             tata[j] ← i
             d[j] ← d[i]+1
```

```
void bf(int s){
    int p,u,i,j;
    p = u = 1;  c[1]=s;
    viz[s]=1; d[s]=0;
    while(p<=u){
      i=c[p]; p=p+1;
      cout<<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)
             viz[j] ← 1
             tata[j] ← i
             d[j] ← d[i]+1
```

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

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

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

Complexitate

Complexitate

- Matrice de adiacență
- Liste de adiacență

Complexitate

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

Aplicații

- Test graf conex

Aplicații

- Test graf conex

```
bf(1)
testăm dacă toate vârfurile au fost vizitate
```

Aplicații

- Determinarea numărului de componente conexe

Aplicații

- Determinarea numărului de componente conexe

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

Aplicații

- Determinarea unui arbore parțial al unui graf conex

Aplicații

- Determinarea unui arbore parțial al unui graf conex

Muchiile **{tata[x], x}, x ≠ s**

Aplicații

- Determinarea unui lanț/drum minim între două vârfuri date u și v

Aplicații

- Determinarea unui lanț/drum minim între două vârfuri date u și v

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

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

Aplicații

- Determinarea unui lanț/drum minim între două vârfuri date u și v

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

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

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

Corectitudine

Corectitudine

- **Observația 1.** Dacă în coada **C** avem: $v_1, v_2, \ldots, v_r$ (la un moment al execuției algoritmului), atunci

$$d[v_1] \leq d[v_2] \leq \ldots \leq d[v_r] \leq d[v_1] + 1$$

Corectitudine

- **Observația 1.** Dacă în coada **C** avem: $v_1, v_2, \ldots, v_r$ (la un moment al execuției algoritmului), atunci

$$d[v_1] \leq d[v_2] \leq \ldots \leq d[v_r] \leq d[v_1] + 1$$

- **Observația 2.** Dacă $d[v] = k$, atunci există în G un drum de la s la v de lungime k

Corectitudine

- **Observația 1**. Dacă în coada **C** avem: $v_1, v_2, \ldots, v_r$ (la un moment al execuției algoritmului), atunci

$$d[v_1] \leq d[v_2] \leq \ldots \leq d[v_r] \leq d[v_1] + 1$$

- **Observația 2**. Dacă $d[v] = k$, atunci există în G un drum de la s la v de lungime k

- **Consecințe**.
  - $d[v] \geq d(s,v)$
  - $d(s,v) = \infty \Rightarrow d[v] = \infty$

Corectitudine

- **Observația 3.** Dacă $P = [v_1, v_2, \ldots, v_r]$ este drum/lanț minim de la v_1 la v_r, atunci subdrumul lui P dintre vârfurile v_i și v_j $[v_i, \ldots v_j]$ este drum minim de la v_i la v_j

Corectitudine

- **Propoziție.** **Pentru orice vârf v avem**

  d[v] = d(s, v) = distanța de la s la v