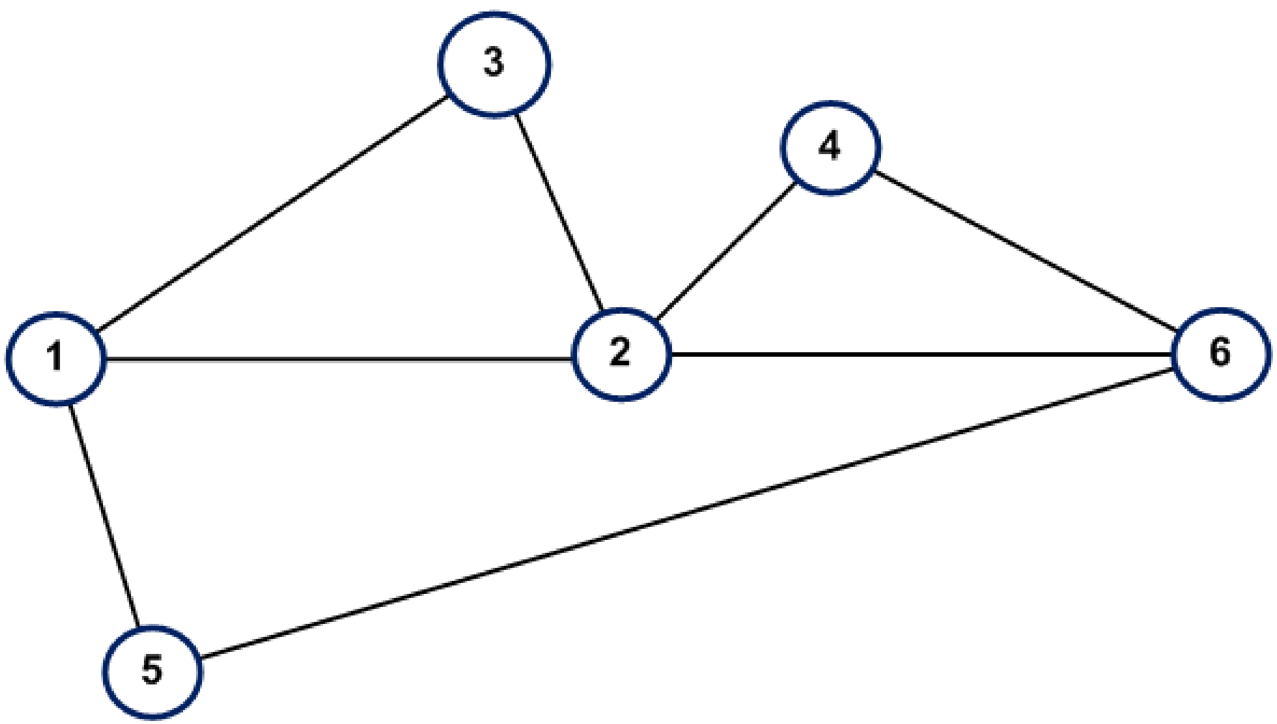
# Parcurgerea Grafurilor

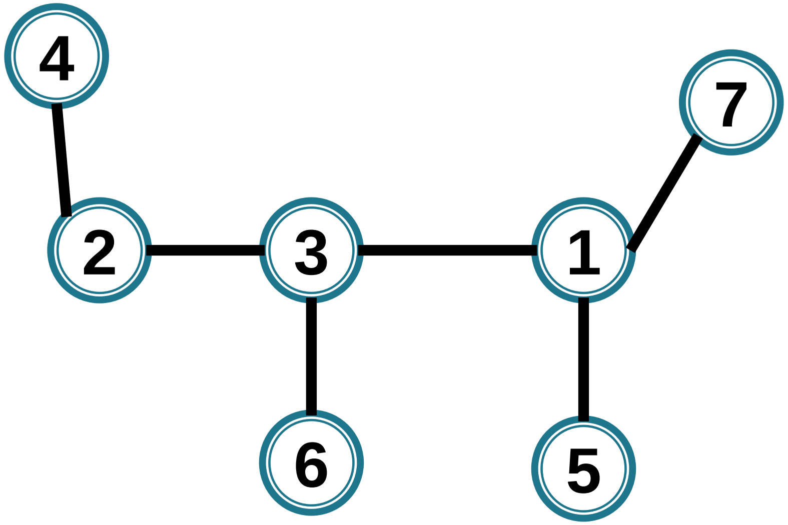
# Modalităţi de reprezentare a grafurilor

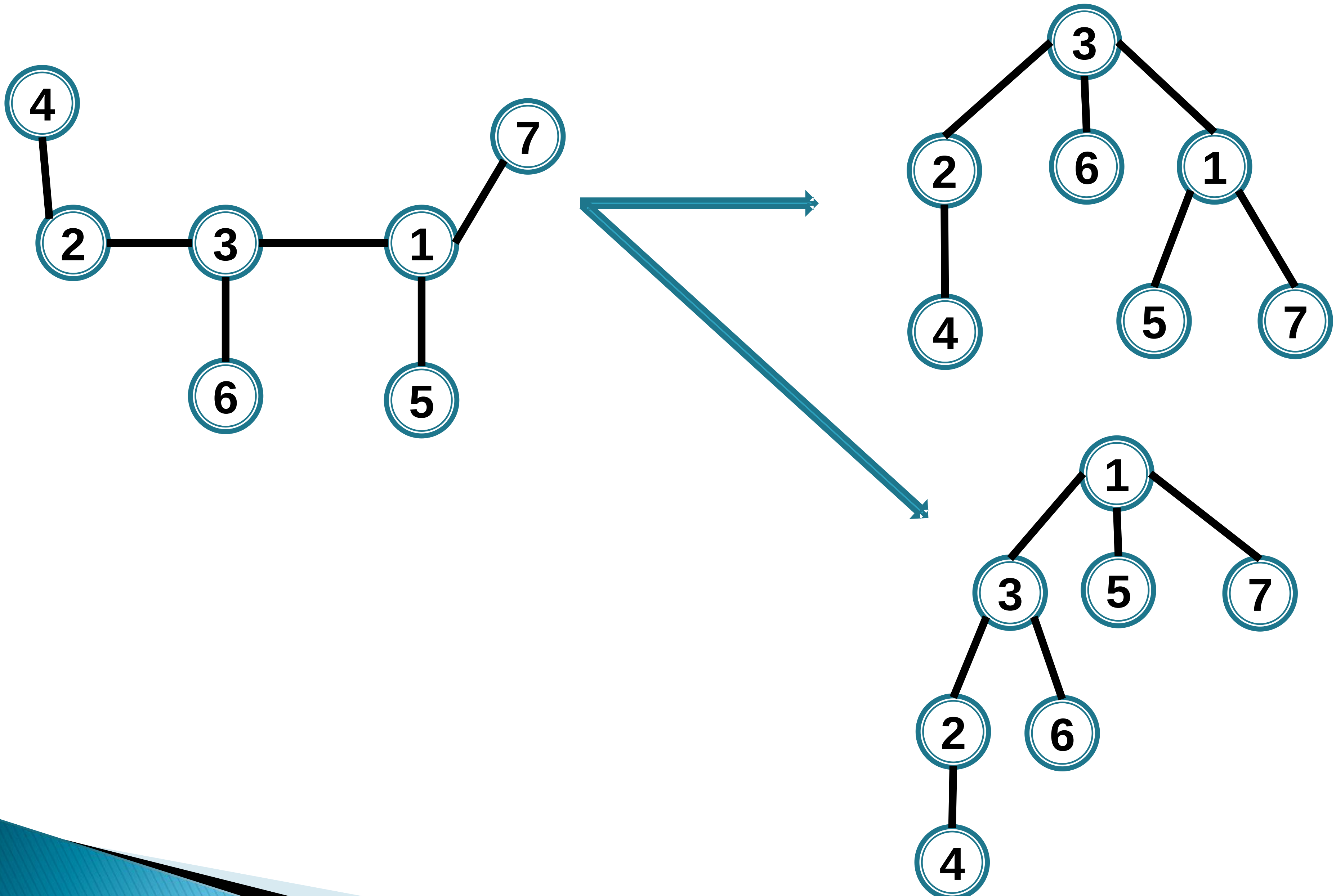
# Reprezentarea grafurilor. Matrice asociate

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

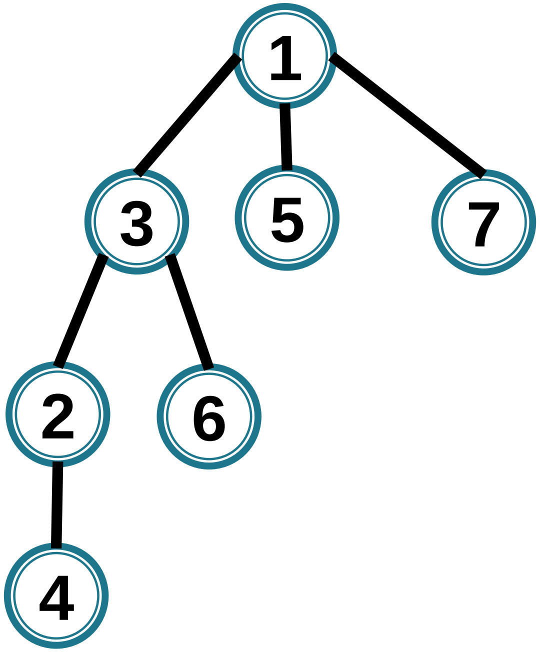


# Arbori cu rădăcină


4
7
2
3
1
6
5


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

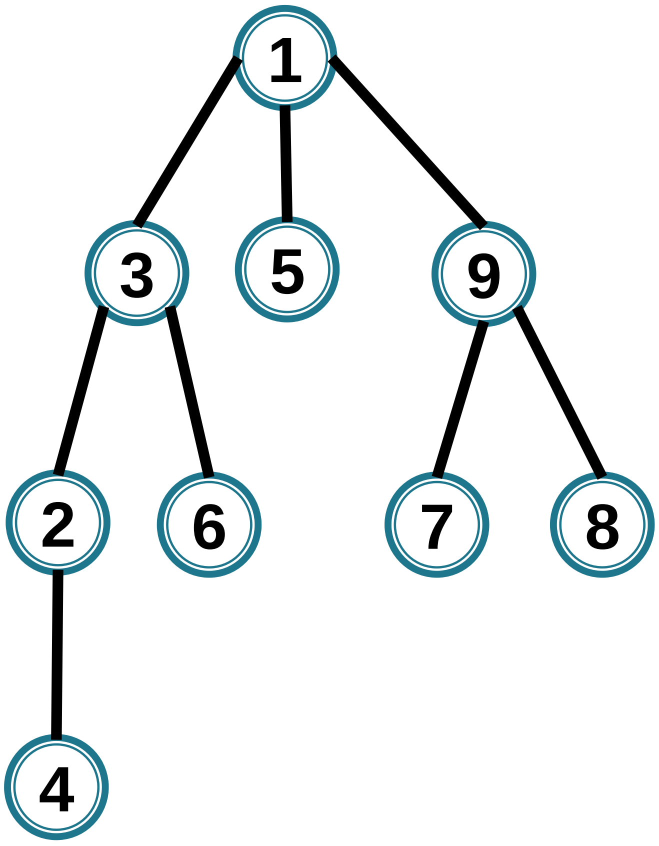
- **Noţiuni**
  - arbore cu rădăcină
  - fiu, tată
  - ascendent, descendent
  - frunză



# 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<<" ";
   }
}
```

# Vectorul tata

- parcurgere de la frunze spre rădăcină

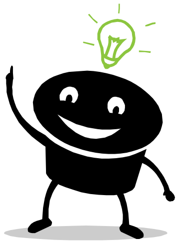
# Parcurgerea Grafurilor

# 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



**Idee**: Dacă

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

atunci v este accesibil din s.

# 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

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

# Parcurgerea în lăţime

# Parcurgerea grafurilor

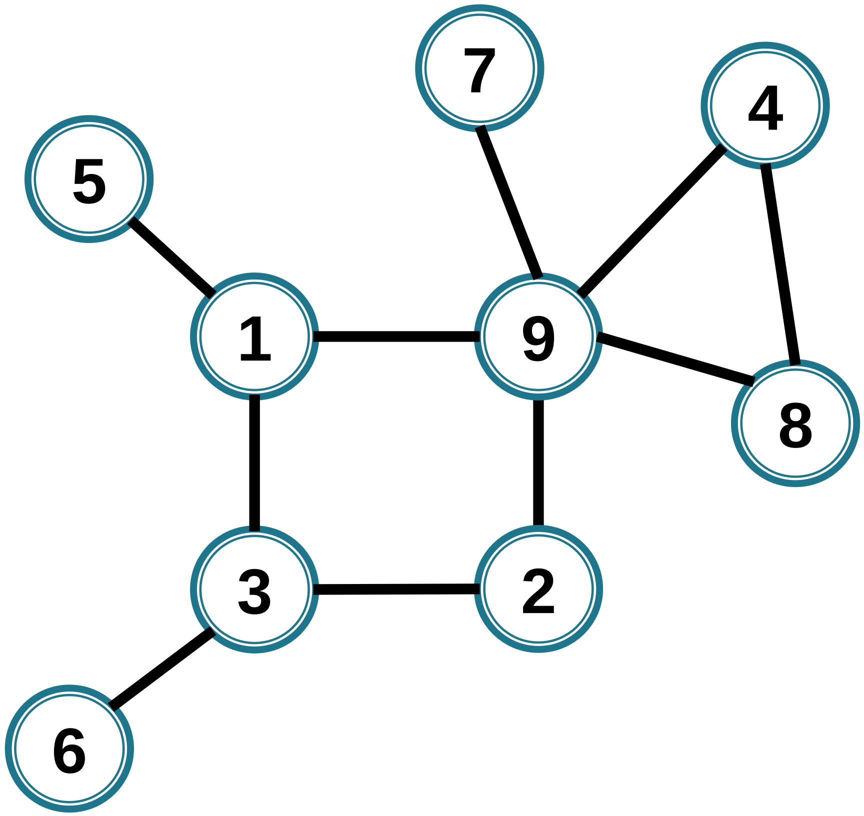
- **Parcurgerea în lățime:** se vizitează

 – vârful de start **s**

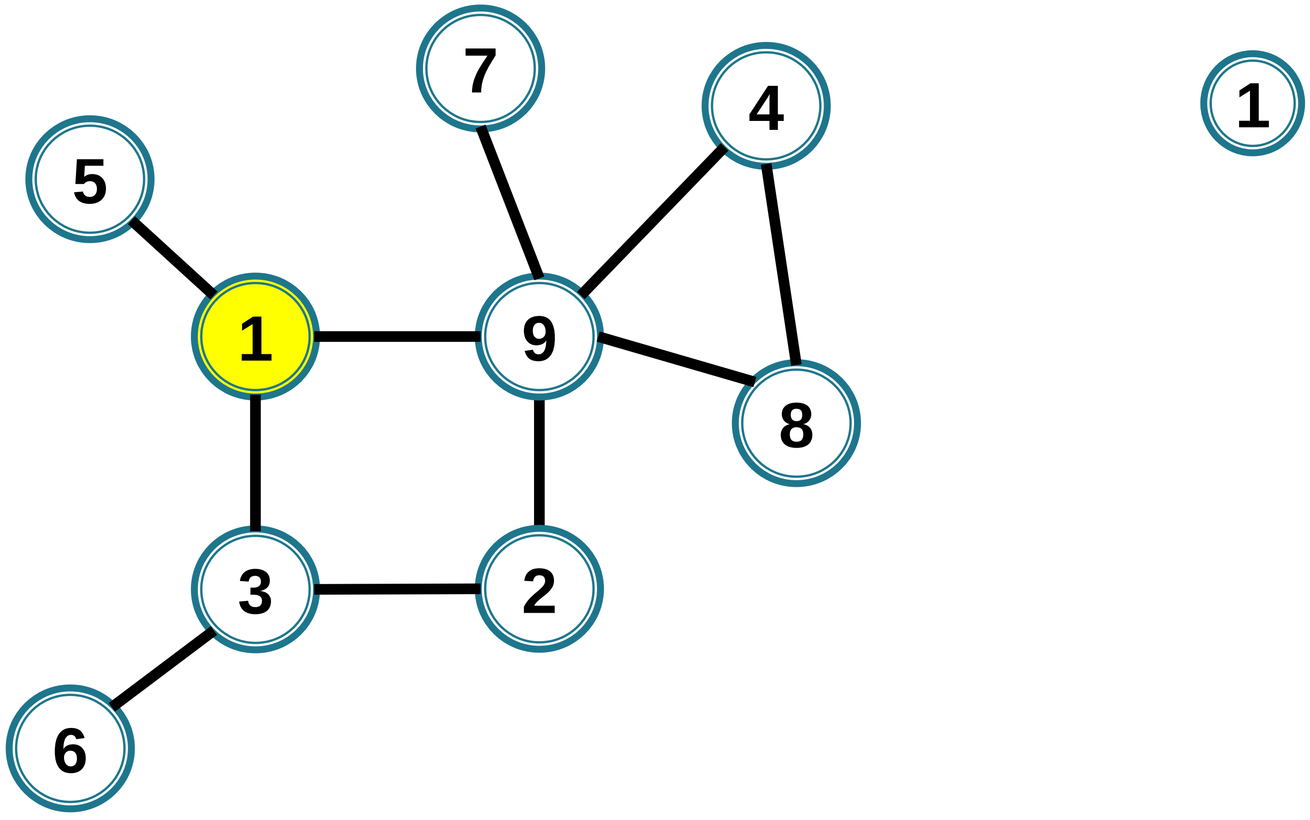
 – vecinii acestuia

 – vecinii nevizitați ai acestora

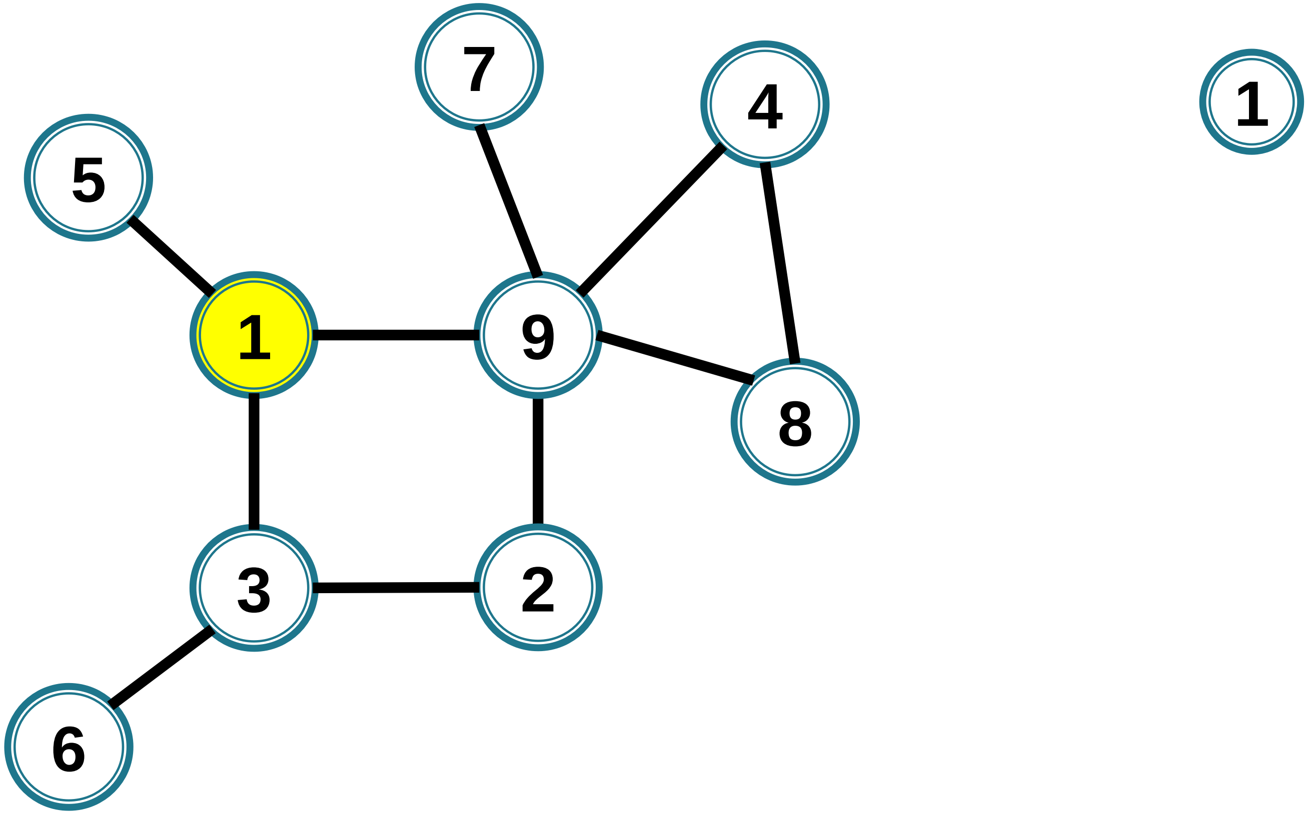
etc



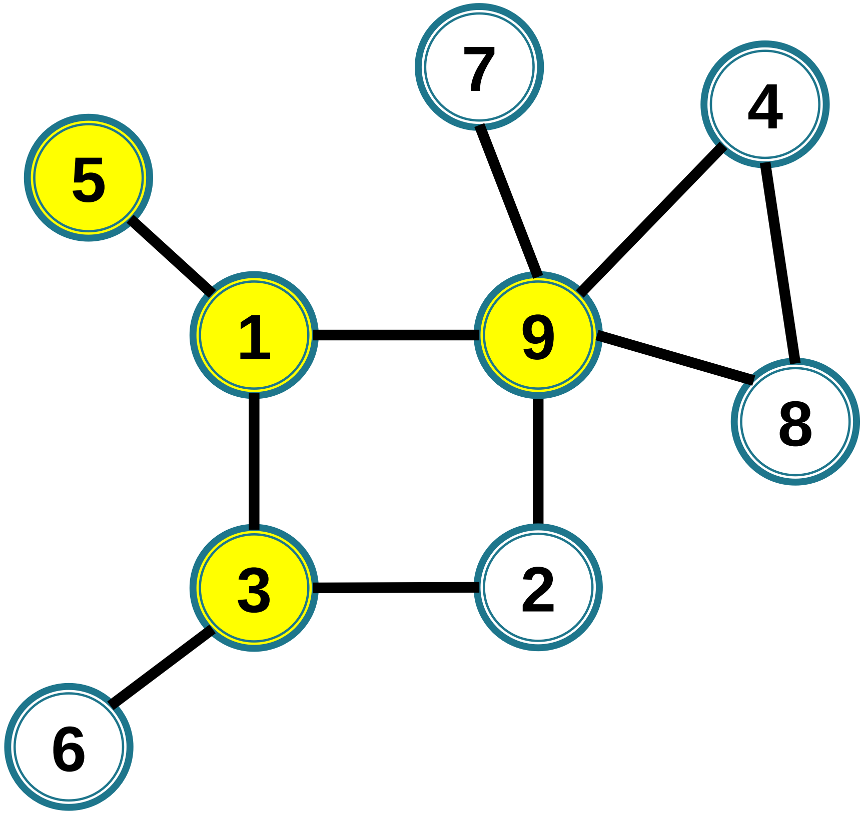
7
4
5
1
9
8
3
2
6

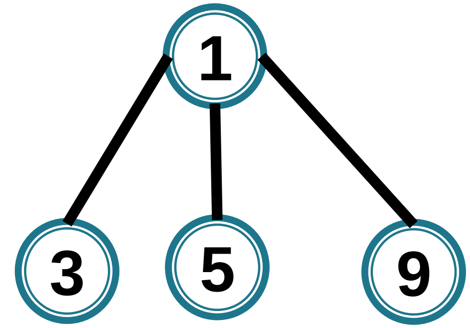



1

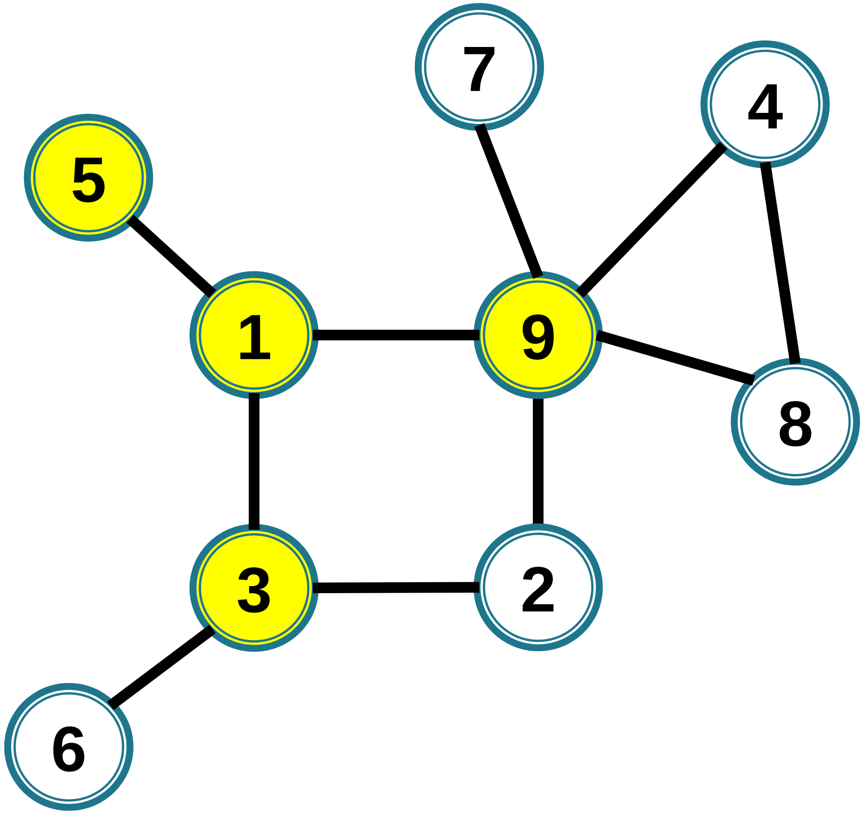


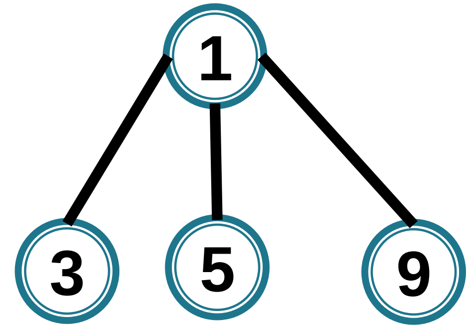

1



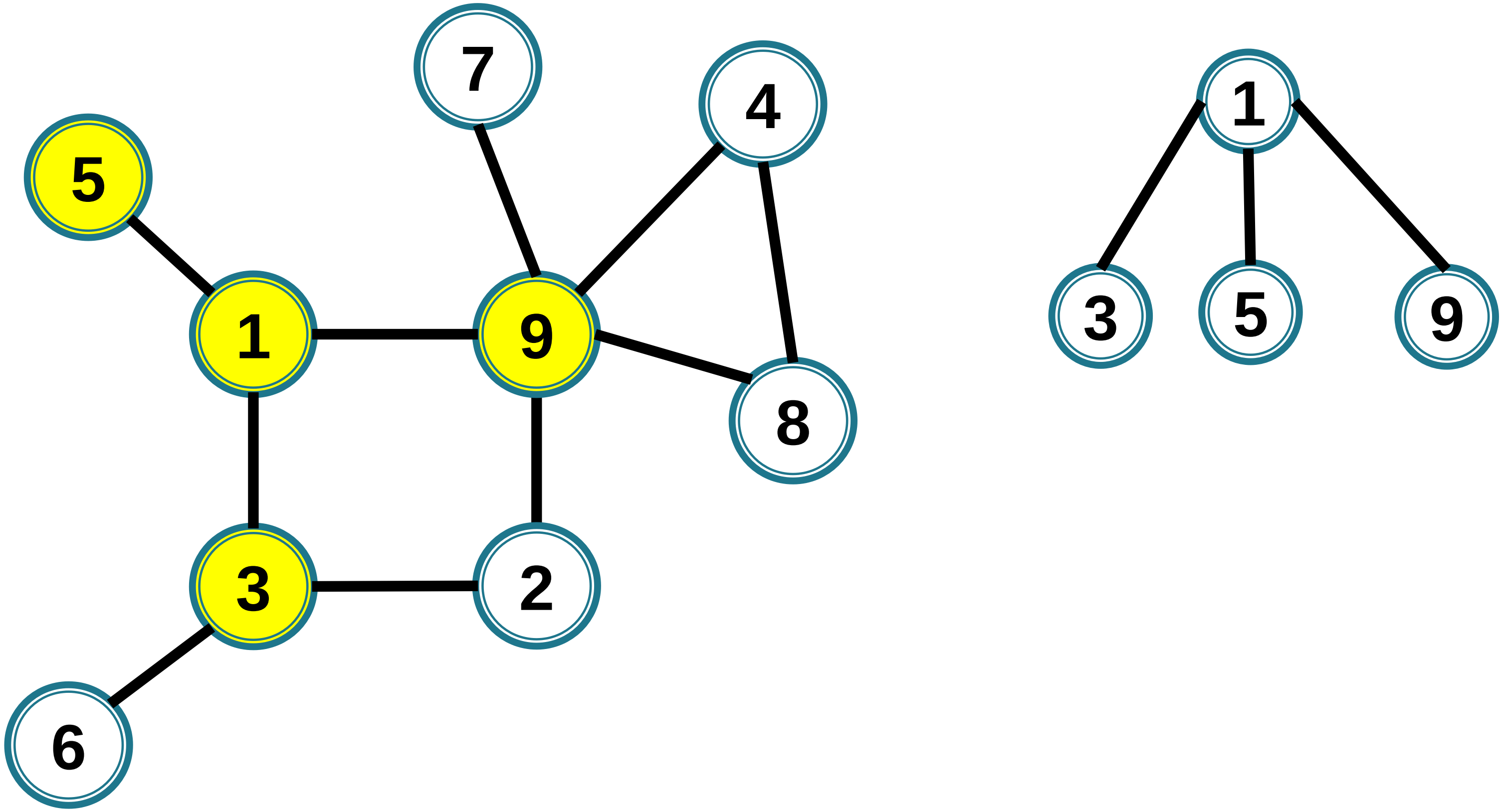




1 3 5 9

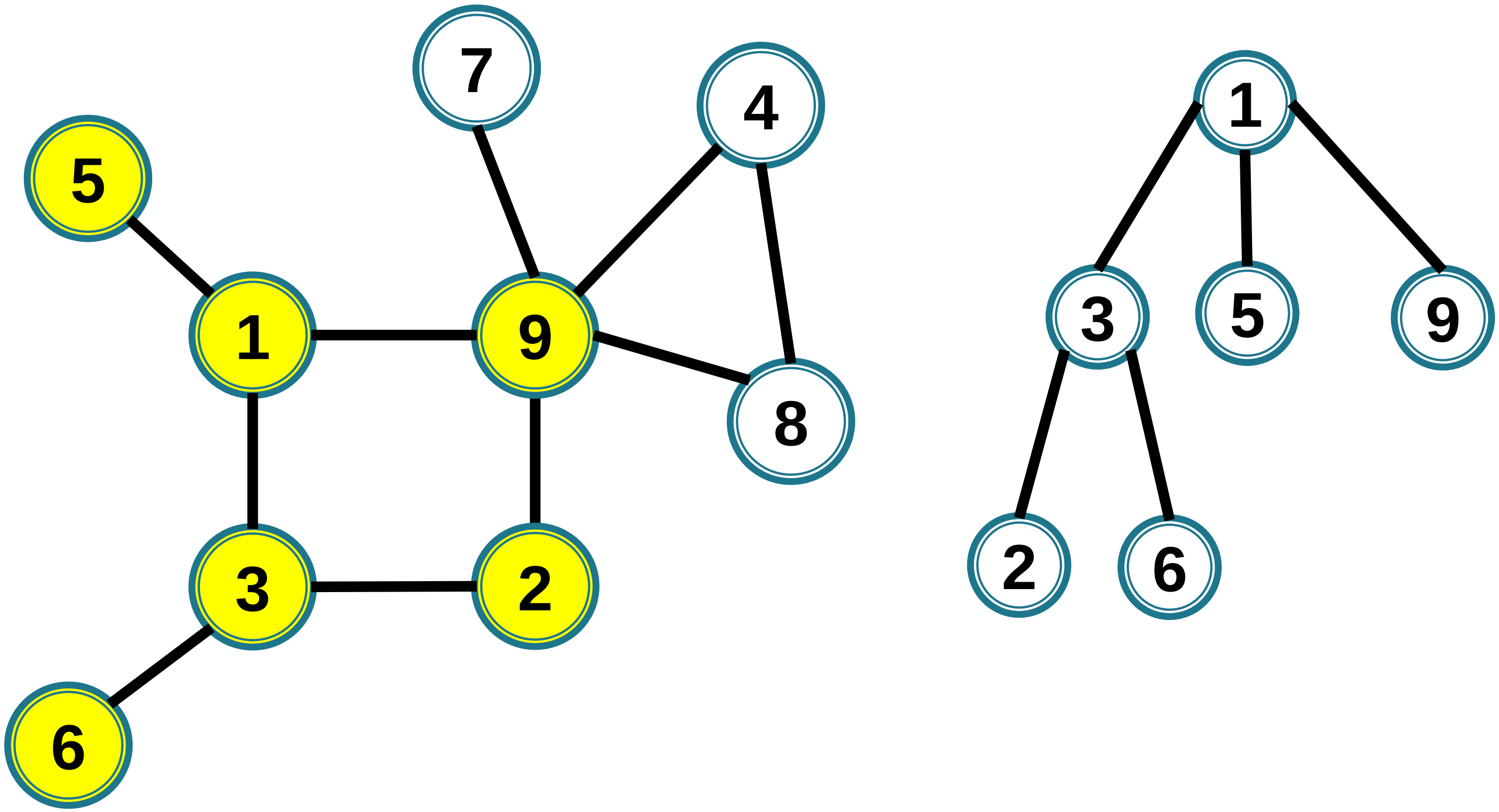


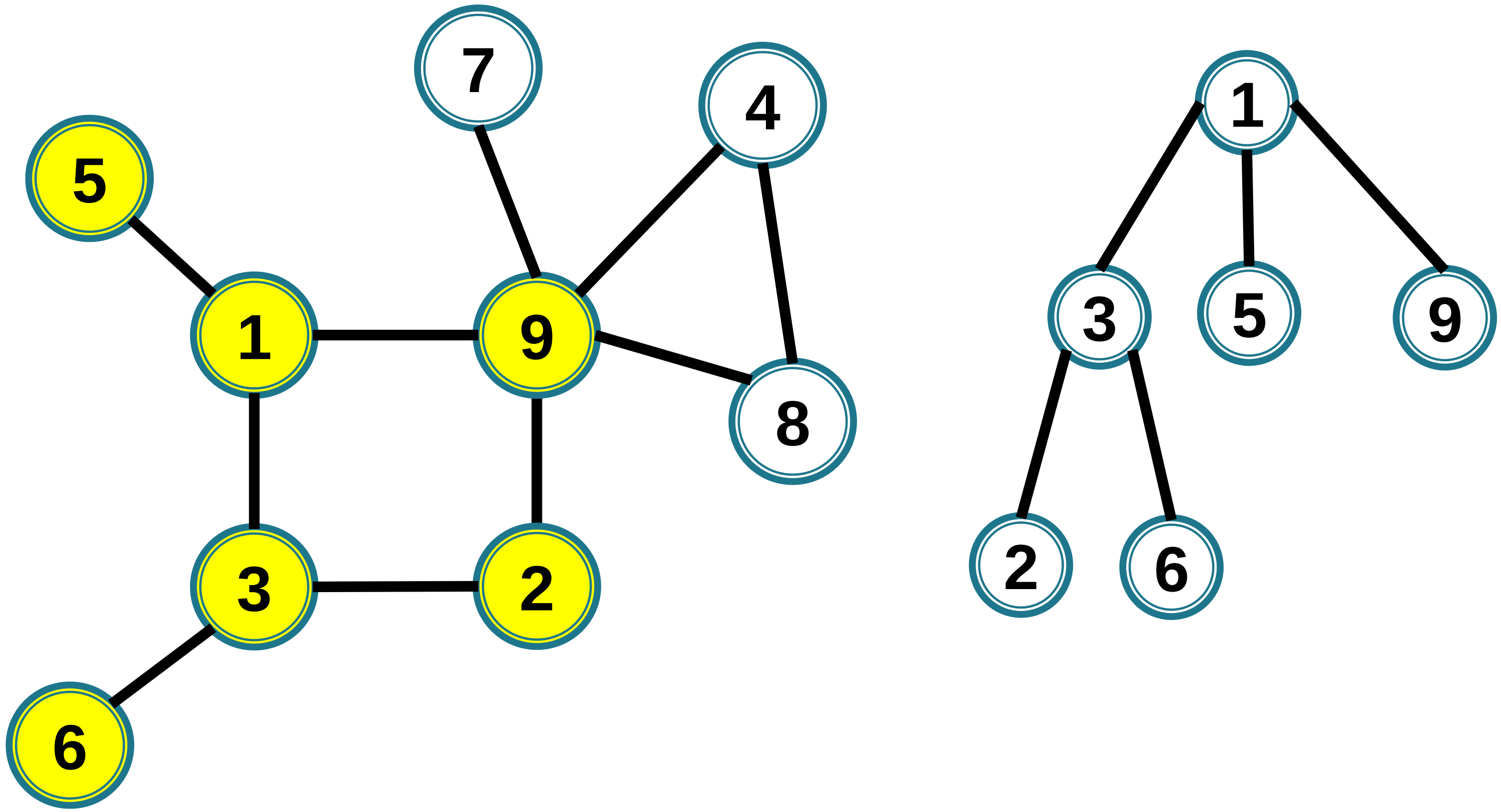

1 3 5 9



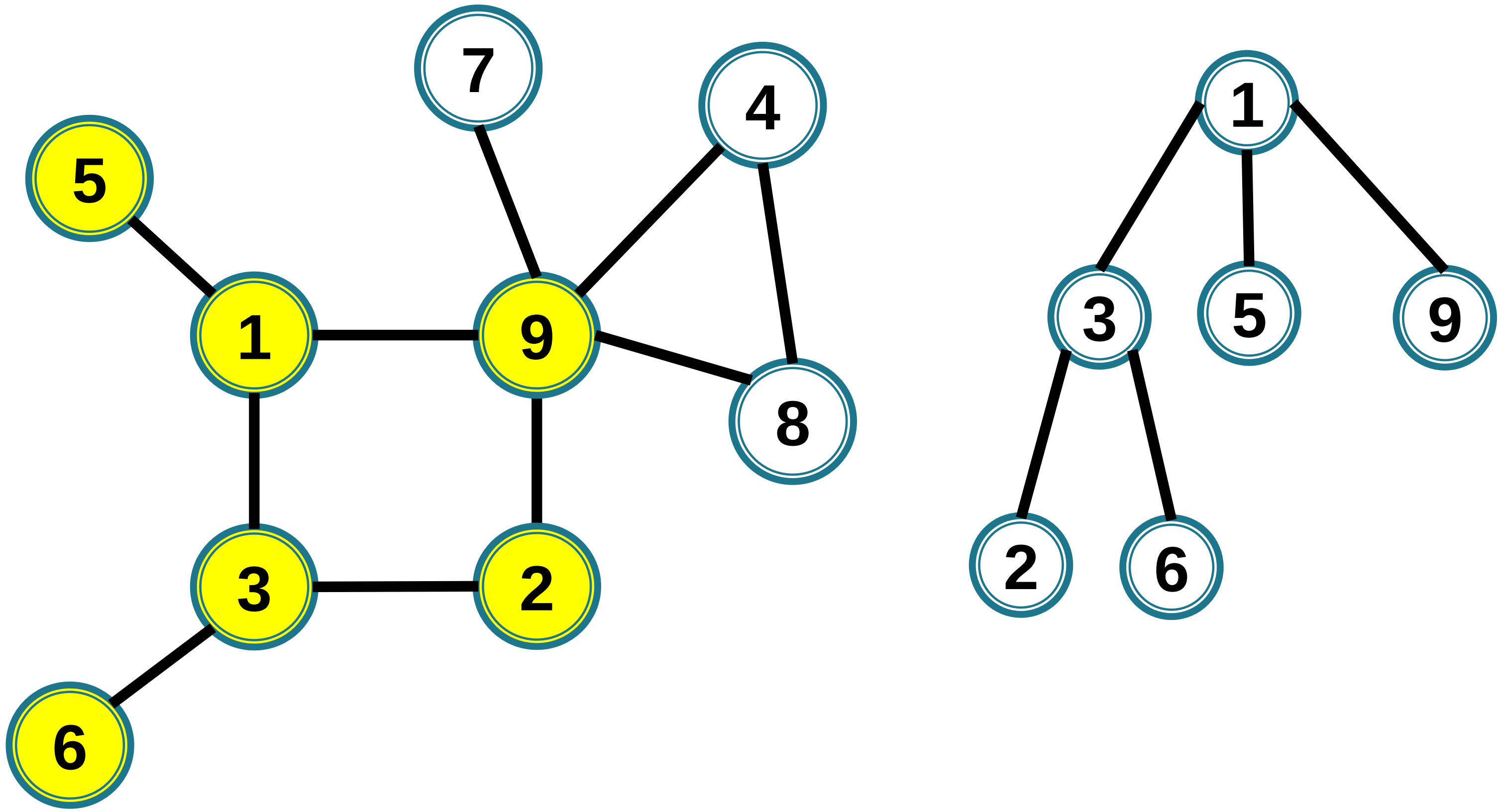

1 3 5 9



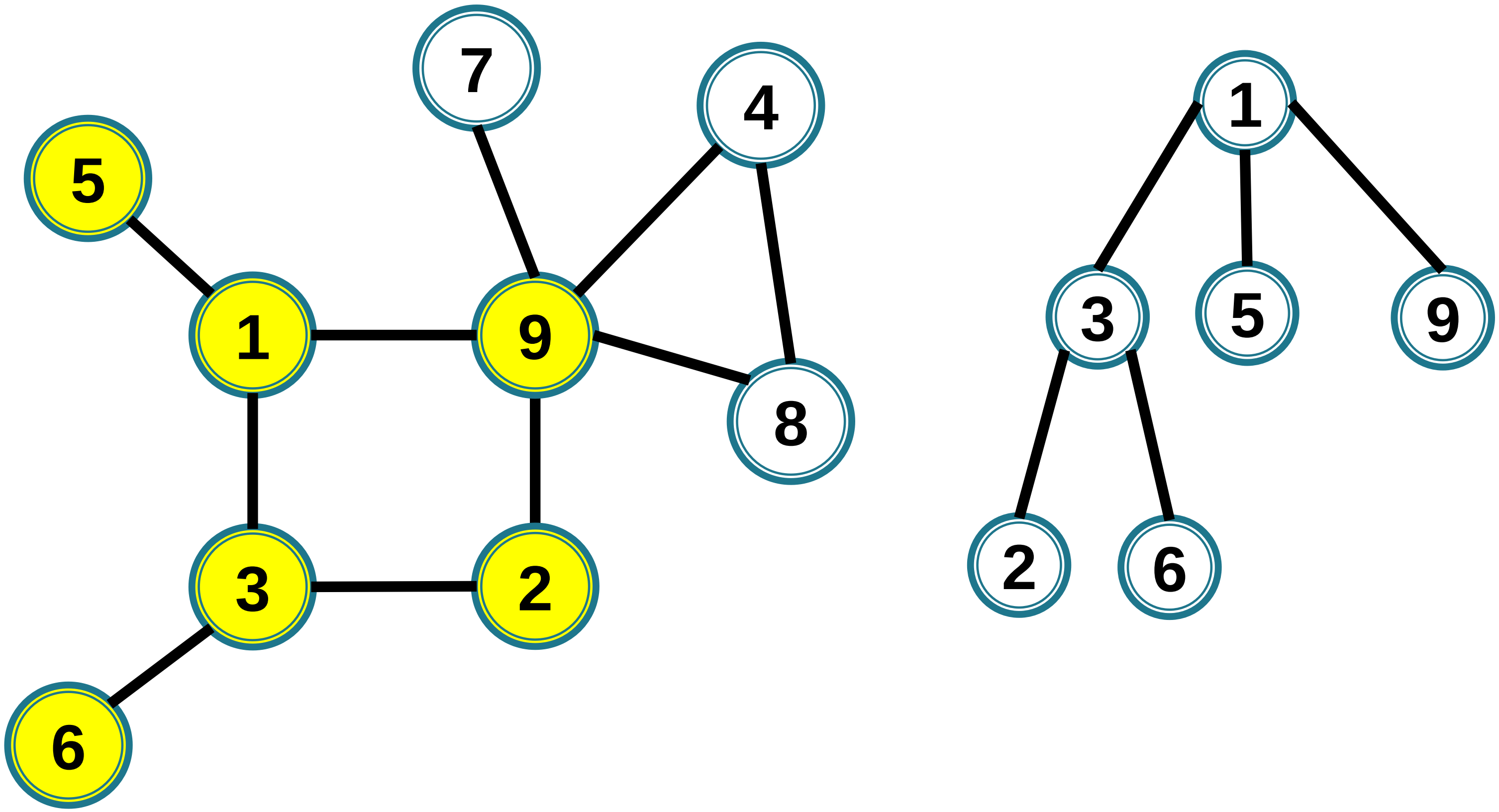

1 3 5 9 2 6



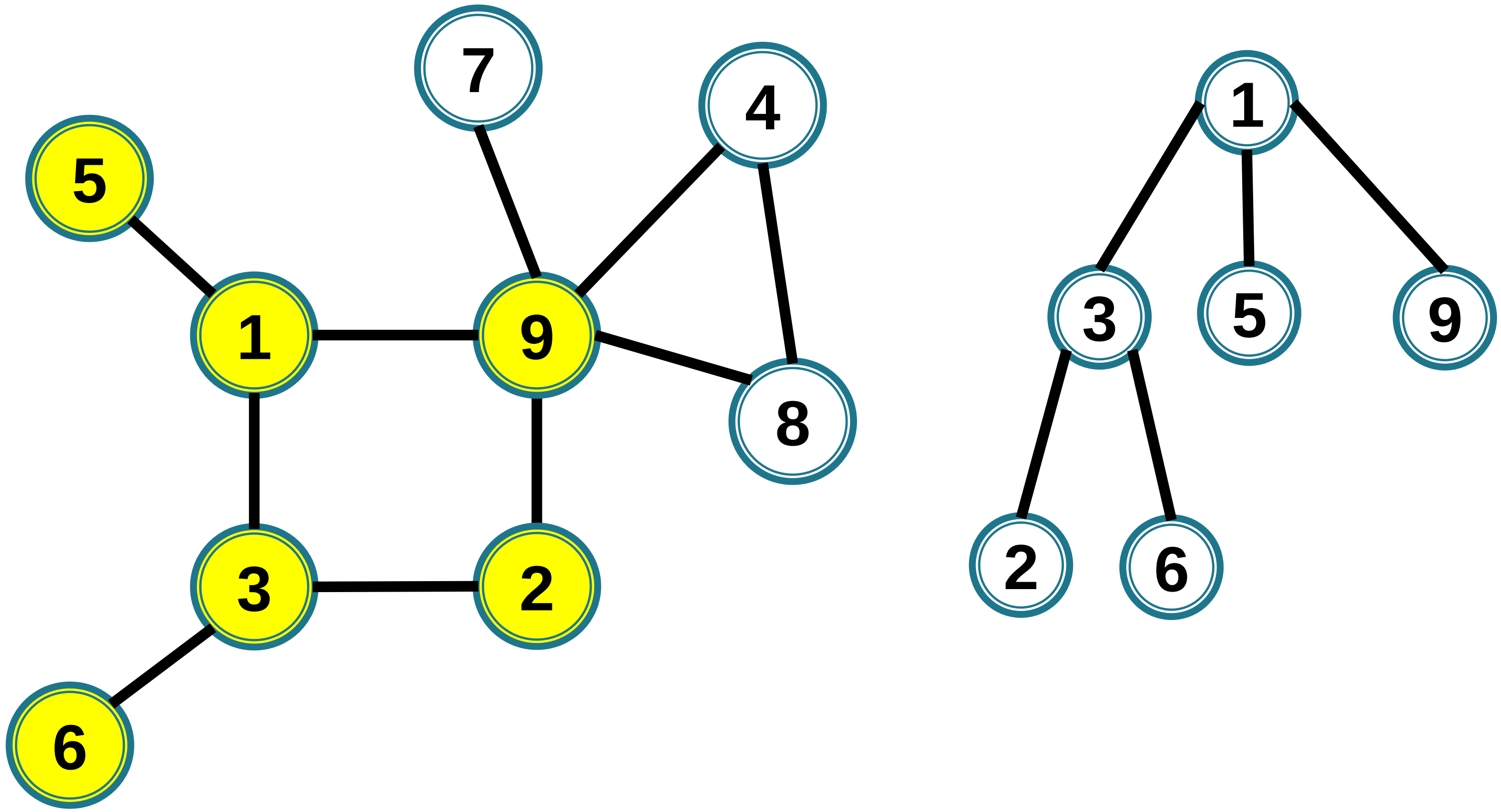

1 3 5 9 2 6



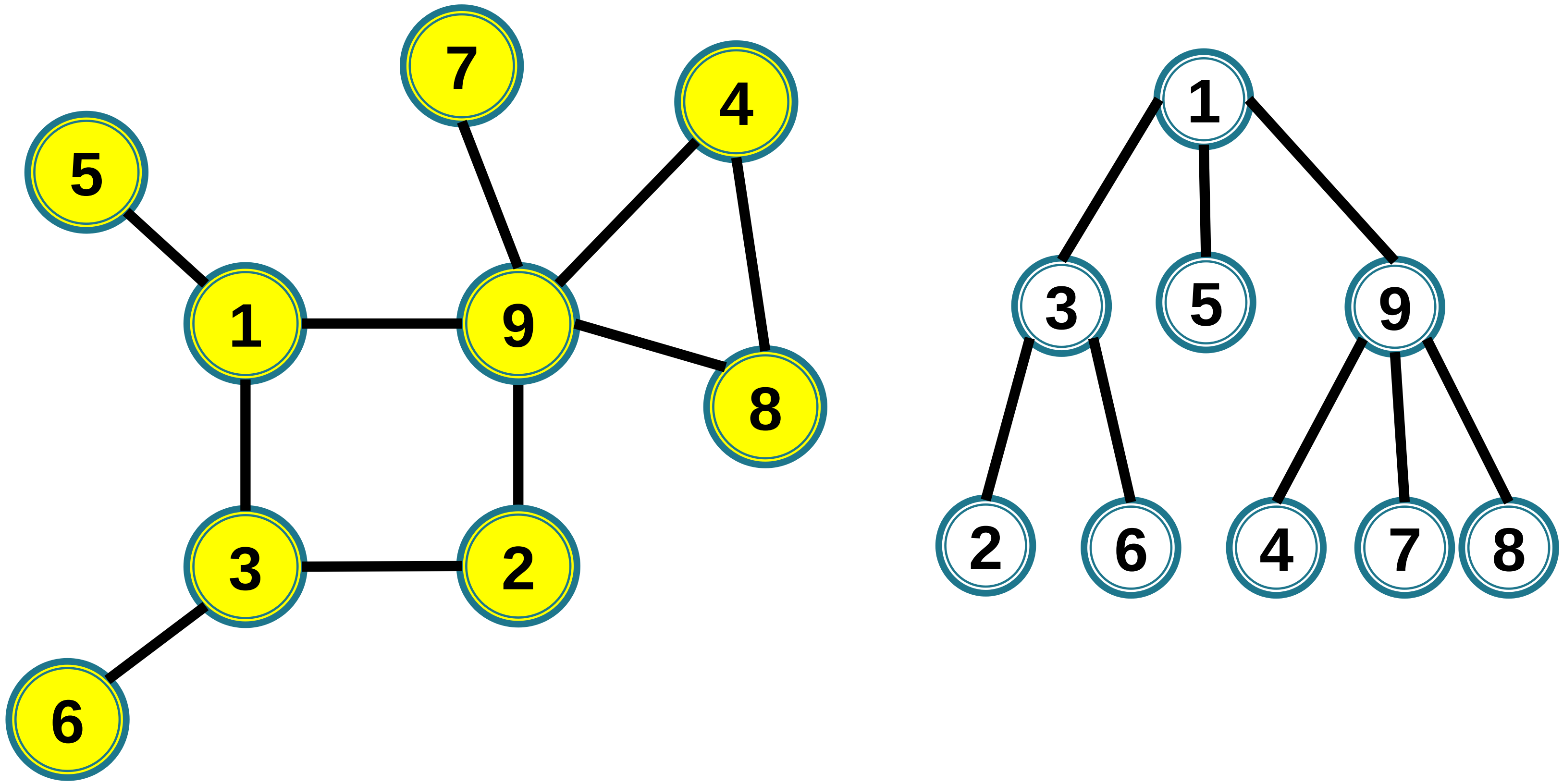

1 3 5 9 2 6



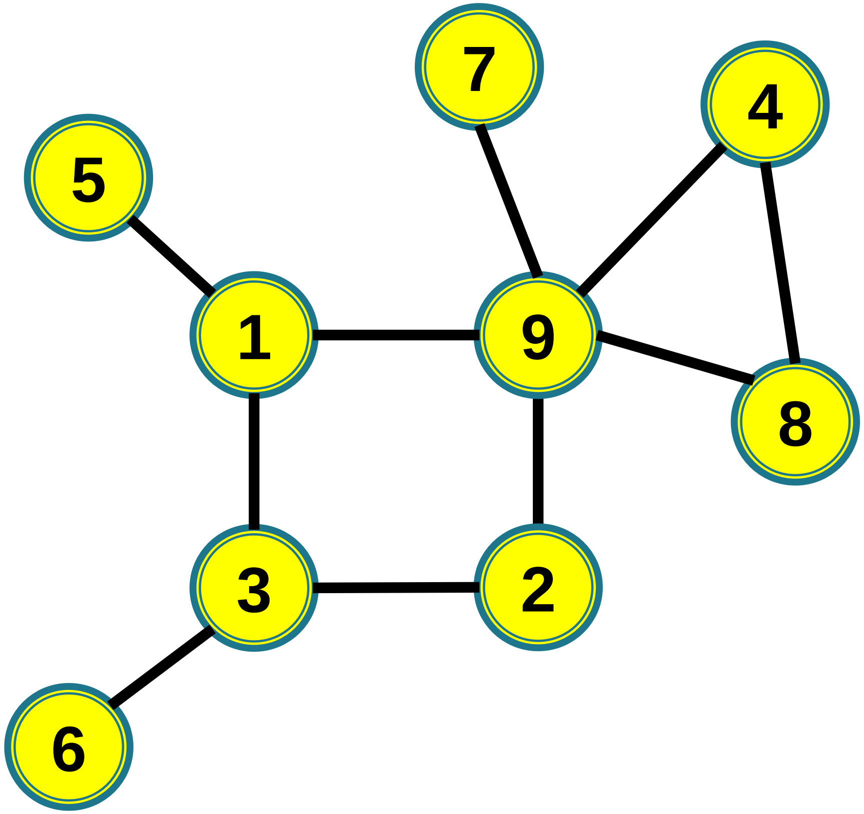

1 3 5 9 2 6

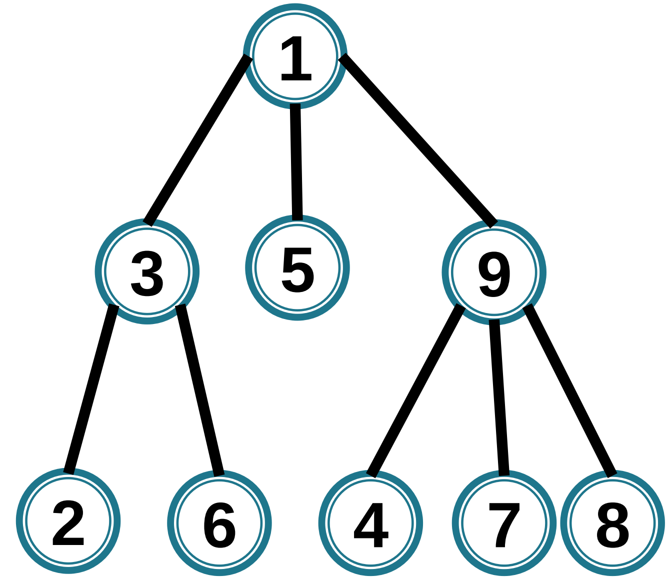



1 3 5 9 2 6

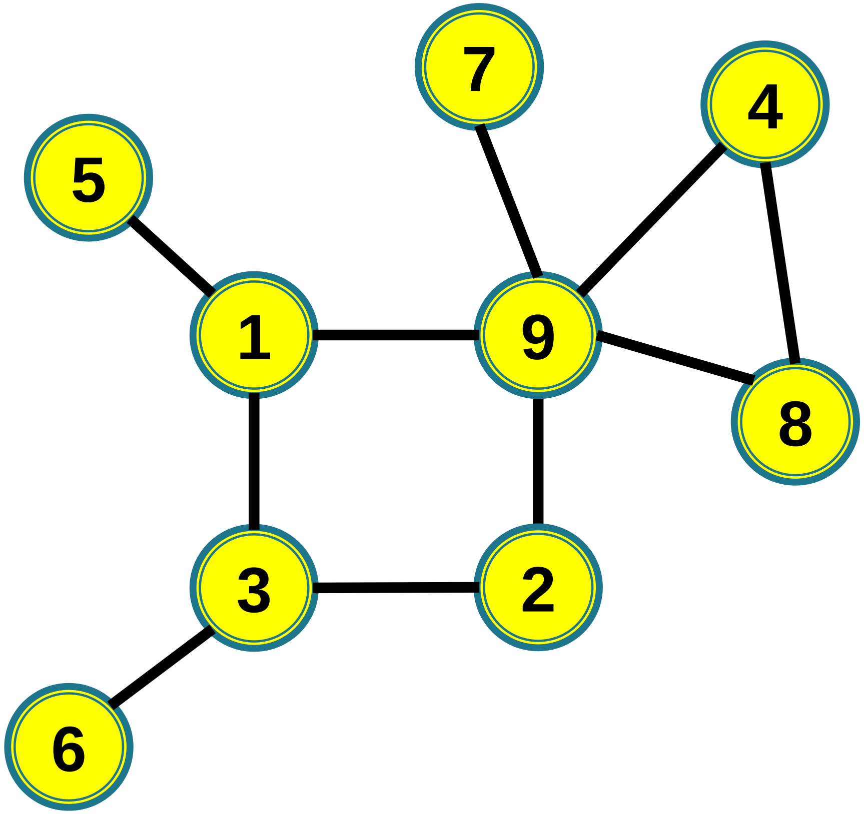


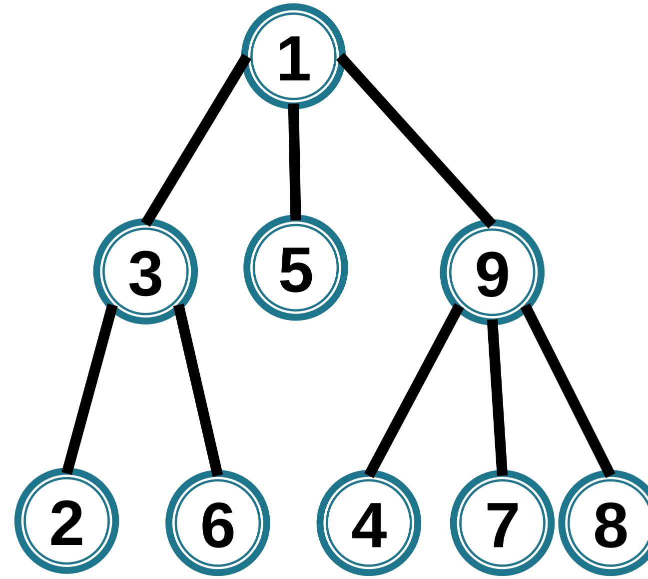

1 3 5 9 2 6 4 7 8



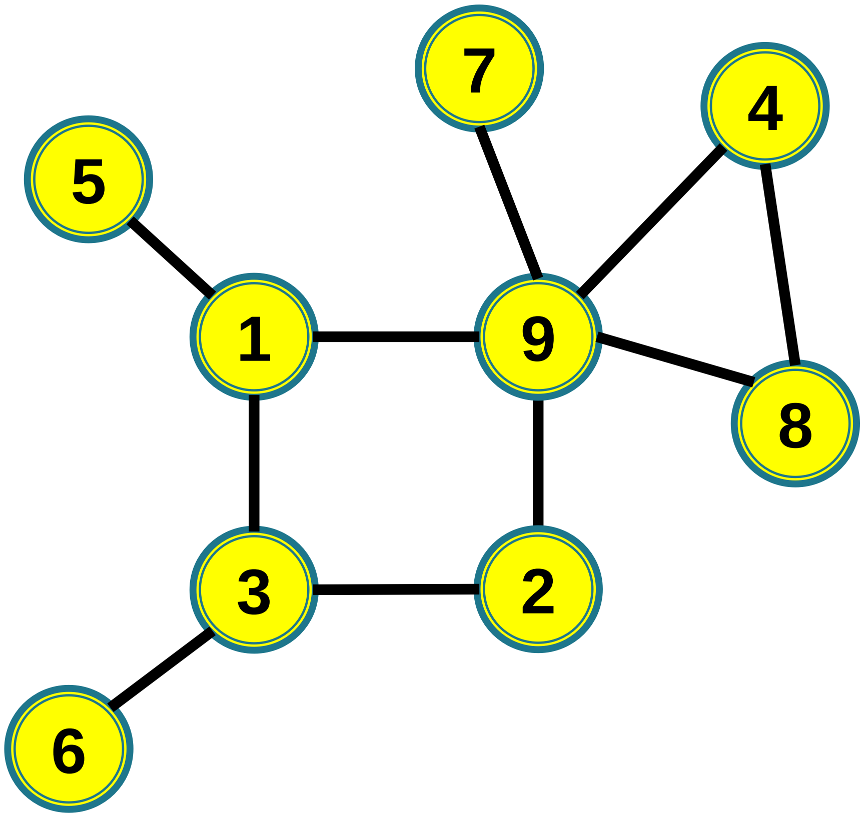



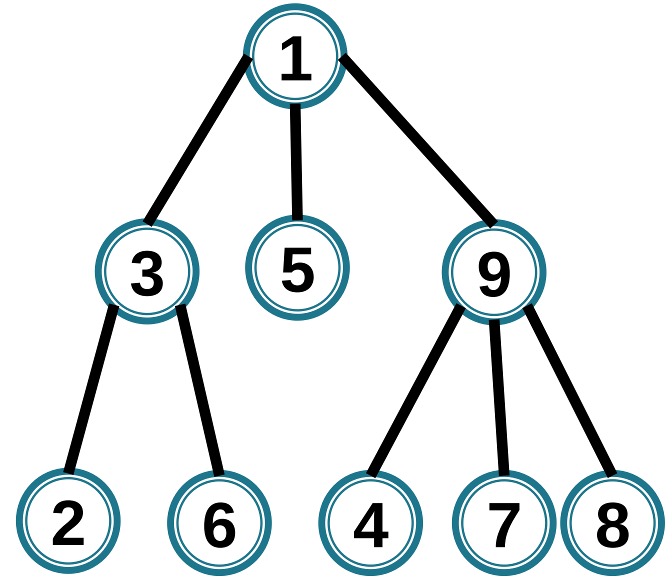

1 3 5 9 2 6 4 7 8



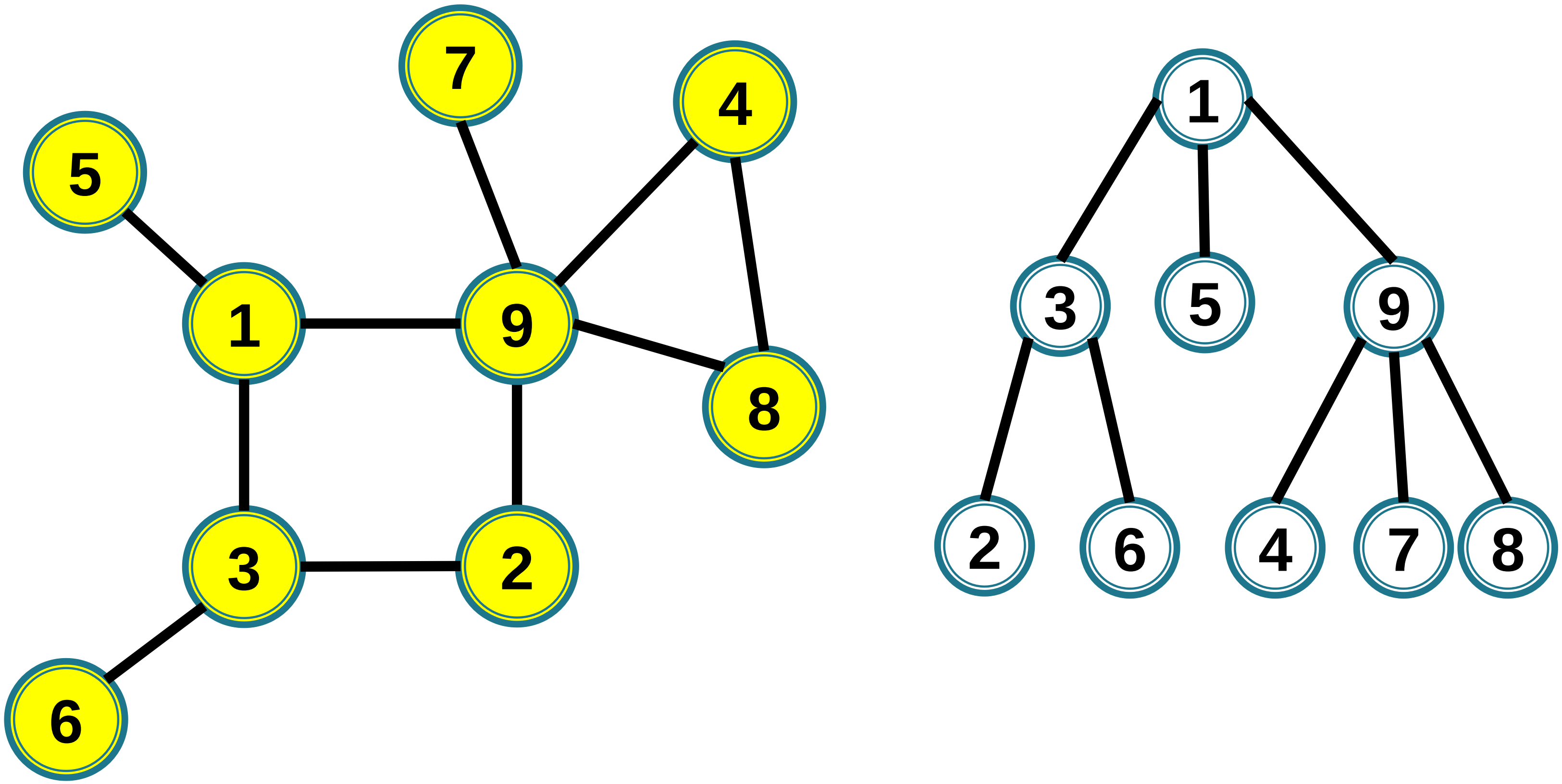




1 3 5 9 2 6 4 7 8

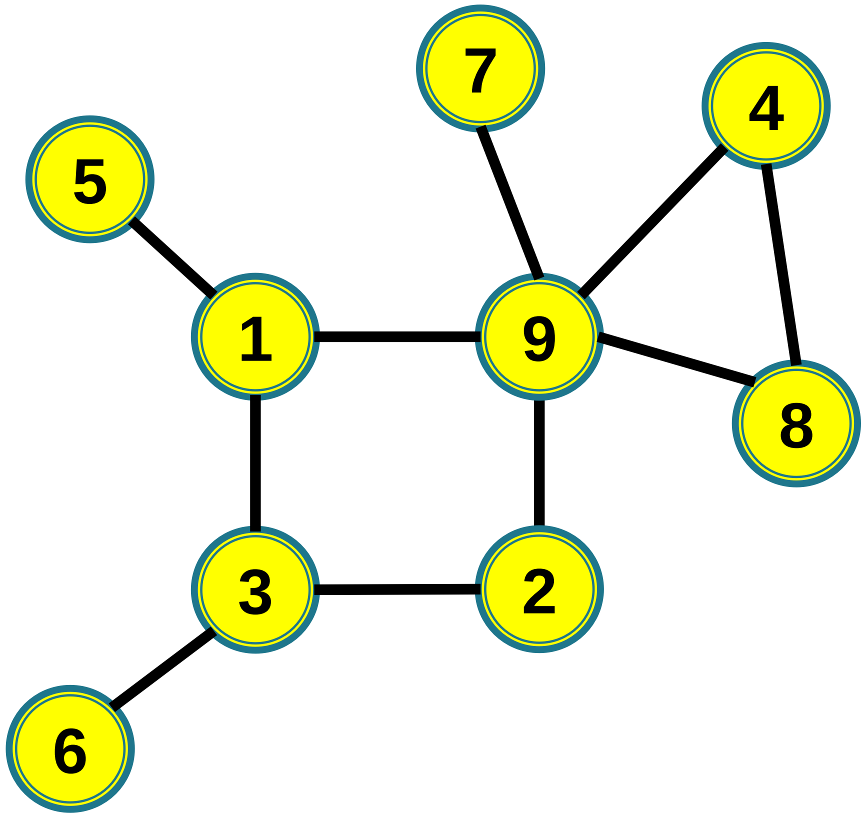


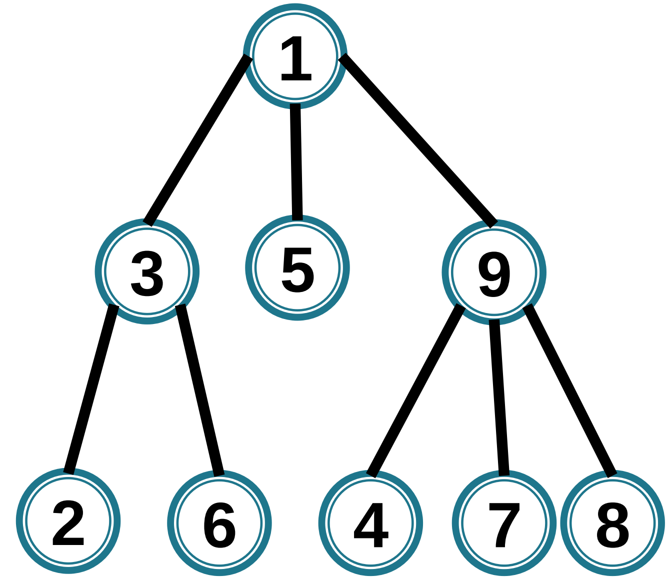




1 3 5 9 2 6 4 7 8

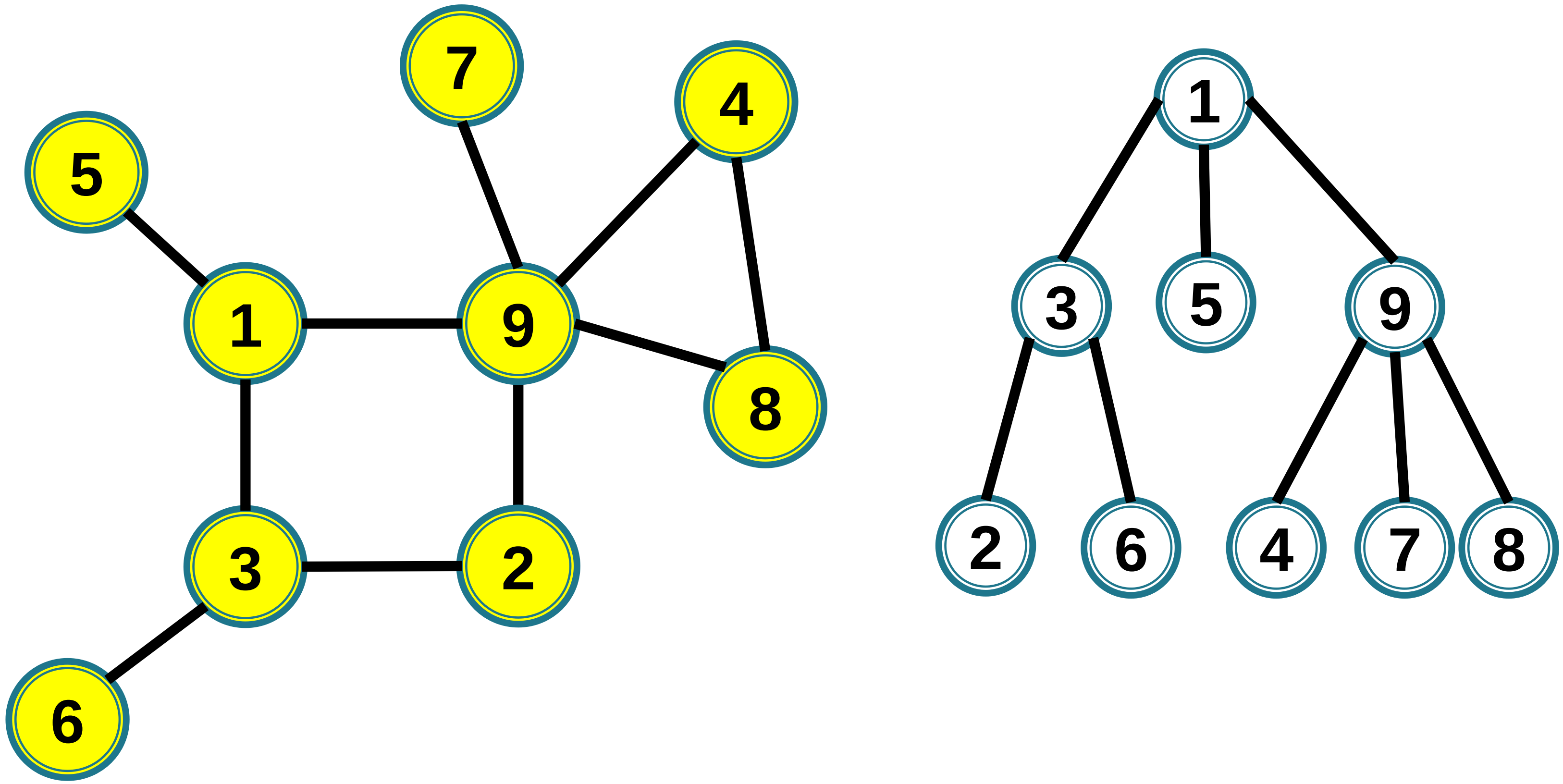



1 3 5 9 2 6 4 7 8

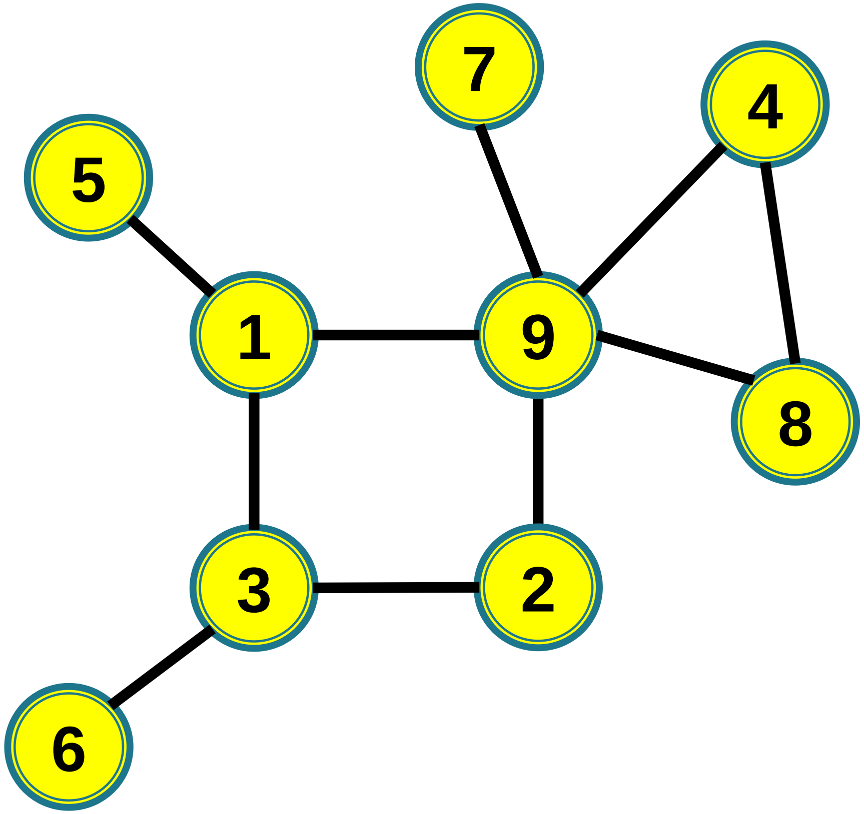


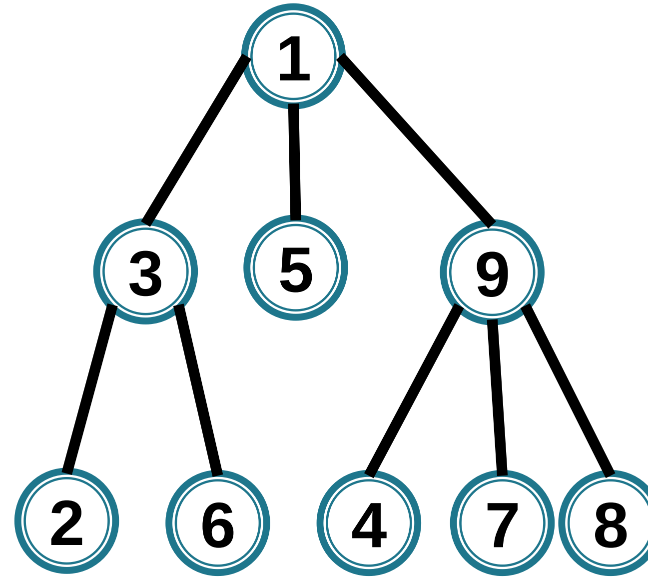




1 3 5 9 2 6 4 7 8

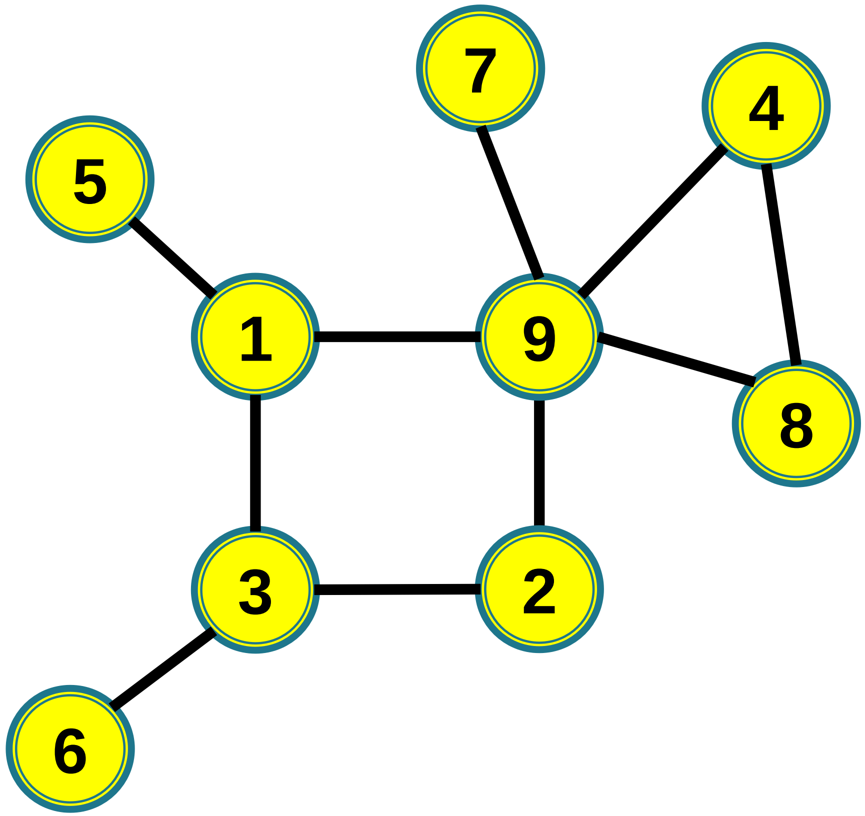


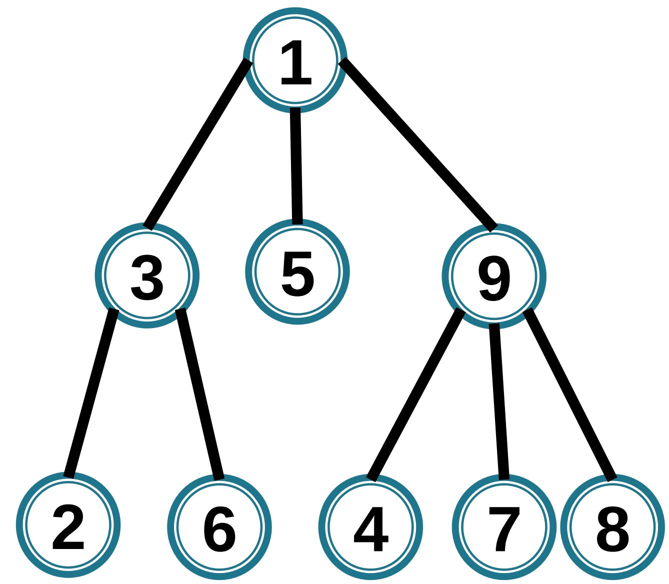

1 3 5 9 2 6 4 7 8



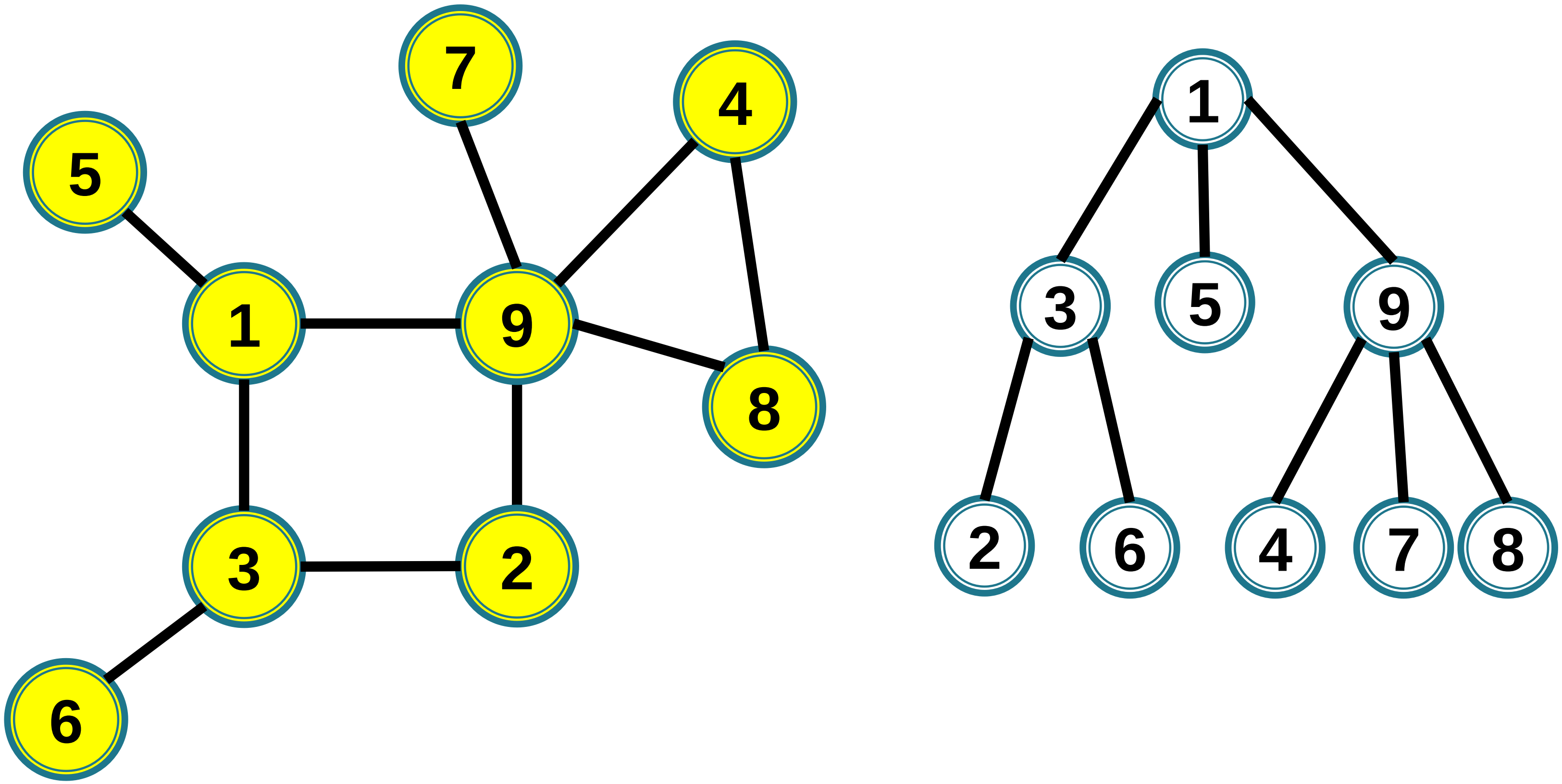



1 3 5 9 2 6 4 7 8

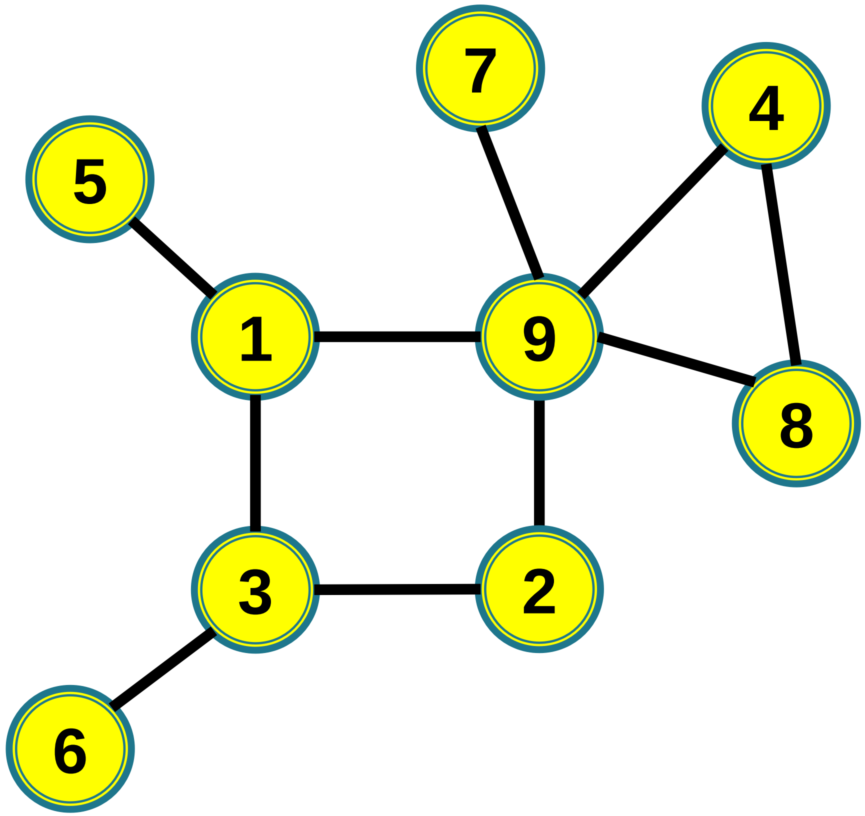


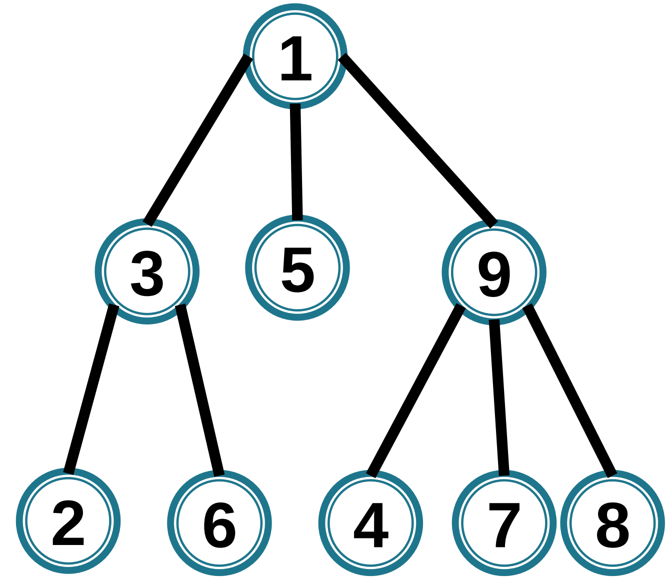




1 3 5 9 2 6 4 7 8

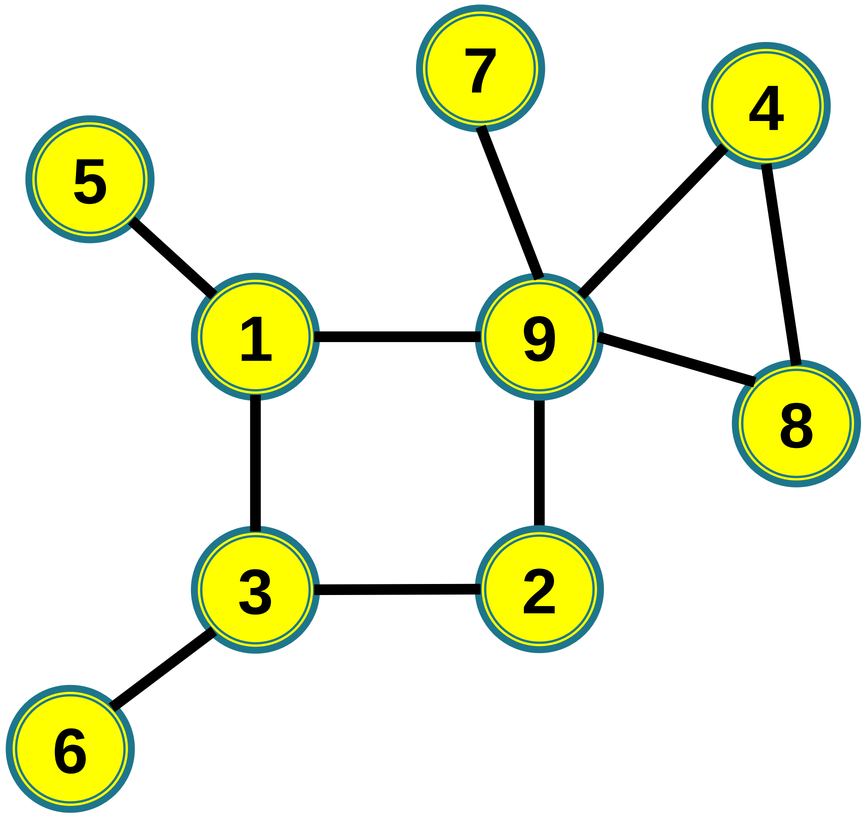


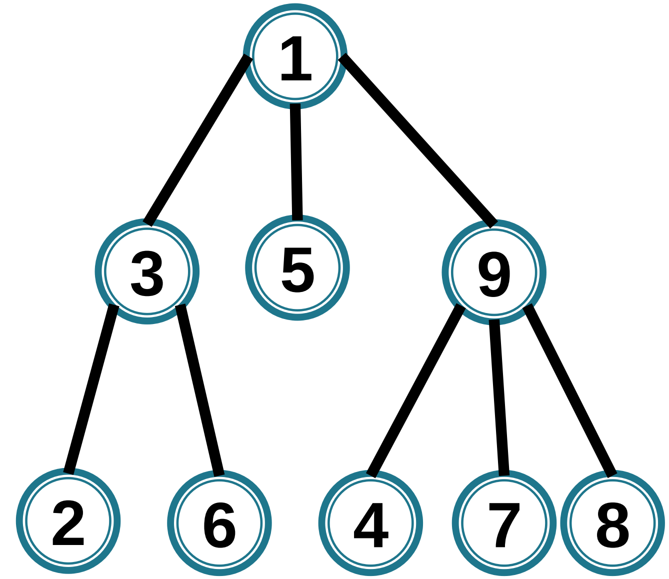

1 3 5 9 2 6 4 7 8




1 3 5 9 2 6 4 7 8




1 3 5 9 2 6 4 7 8

# Implementare

# Parcurgerea în lățime

- Vârfurile vizitate trebuie marcate:

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

# Parcurgerea în lățime

- Muchiile folosite pentru a descoperi vârfuri noi formează un **arbore**

# Parcurgerea în lățime

- Muchiile folosite pentru a descoperi vârfuri noi formează un **arbore**
- Dacă dorim memorarea acestui arbore, pentru:

# Parcurgerea în lățime

- Muchiile folosite pentru a descoperi vârfuri noi formează un **arbore**
- Dacă dorim memorarea acestui arbore, pentru:
  - determinarea unui arbore parțial al grafului,
  - determinarea de lanţuri de la rădăcină la alte vârfuri etc

putem reţine în plus vectorul tata

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

# Parcurgerea în lățime

- Vârfurile sunt vizitate în ordinea distanței față de vârful de start s.

# Parcurgerea în lățime

- Vârfurile sunt vizitate în ordinea distanței față de vârful de start s.
- Dacă dorim şi determinarea de **distanţe minime** de la s la alte vârfuri putem reţine în plus vectorul de distanţe d:

  **d[i]** = lungimea drumului determinat de algoritm de la s la i

# Parcurgerea în lățime

- Vârfurile sunt vizitate în ordinea distanței față de vârful de start s.
- Dacă dorim şi determinarea de **distanţe minime** de la s la alte vârfuri putem reţine în plus vectorul de distanţe d:

  **d[i]** = lungimea drumului determinat de algoritm de la s la i
- Avem

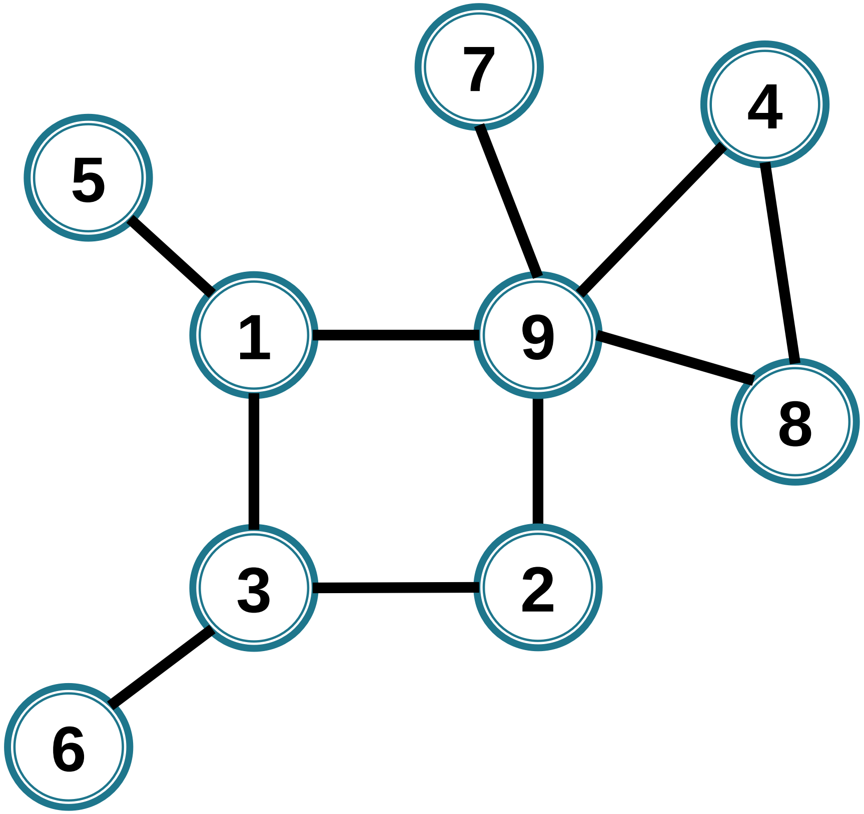
  **d[j]= d[tata[j]]+1**

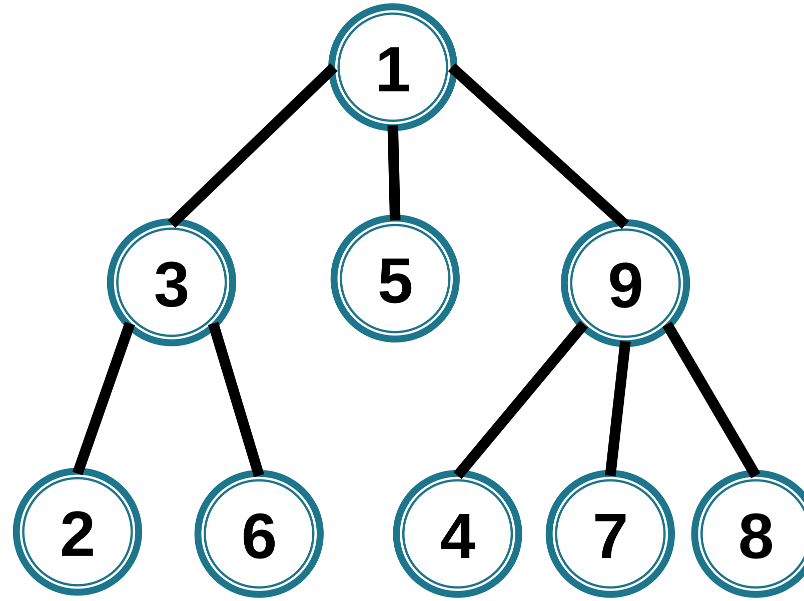
  = nivelul lui j în arborele asociat parcurgerii

# Parcurgerea în lățime

- **Propoziție**

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


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
```

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

**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];
```

**Inițializări**

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

```
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 **$O(|V|^2)$**

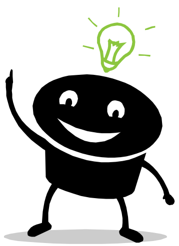
- Liste de adiacență **$O(|V|+|E|)$**

# 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

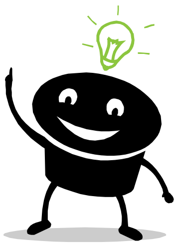
```
nrcomp=0;
for(i=1;i<=n;i++)
   if(viz[i]==0){
        nrcomp++;
        bf(i);
   }
```

# Aplicații

- Determinarea unui arbore parțial al unui graf conex

# Aplicații

- Determinarea unui arbore parțial al unui graf conex



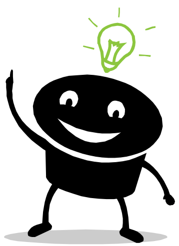
Muchiile **{tata[x], x}, x ≠ s**

# Aplicații

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

# Aplicații

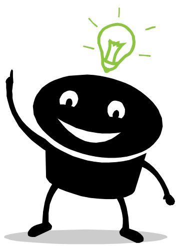
- Determinarea unui lanț/drum minim între două vârfuri date u și v



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

# Aplicații

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

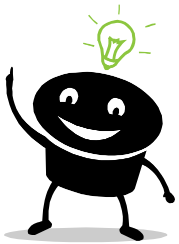


Se apelează bf(u), apoi se afişează drumul de la u la v folosind vectorul tata (ca la arbori), **dacă există**

```
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

- **Lema 1**. Dacă în coada **C** avem: $v_1$, $v_2$,…, $v_r$ (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

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

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

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

# Corectitudine

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

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

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

- **Consecință**. $d[v] \geq d(s,v)$

# Corectitudine

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

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

# 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

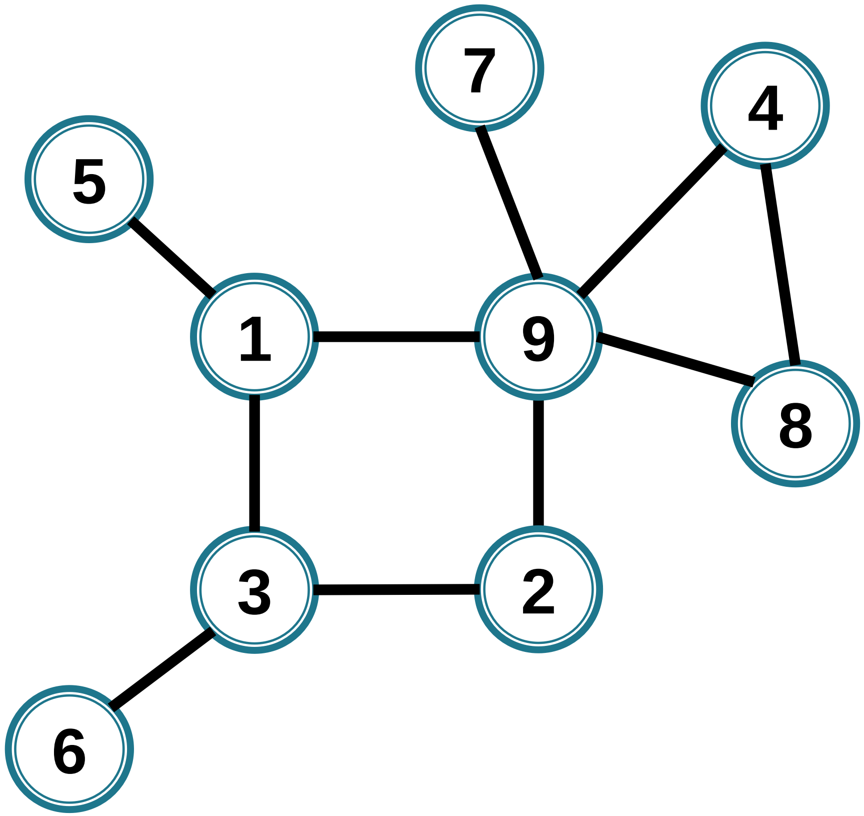
Se vizitează

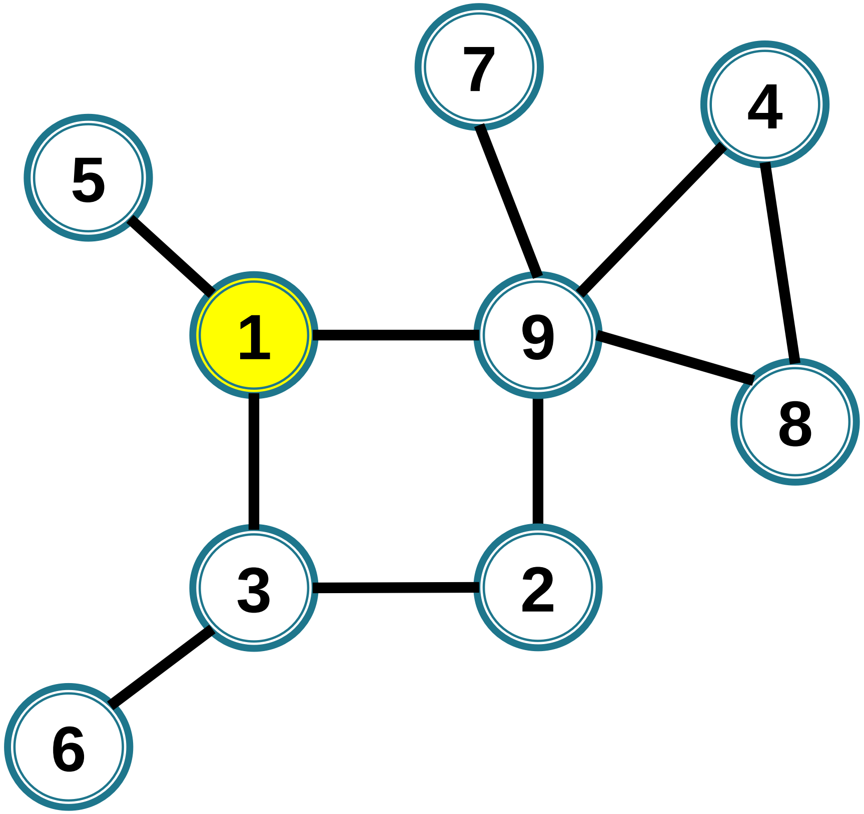
- **Inițial**: vârful de start s - devine vârf curent
- **La un pas**:
  - se trece la primul vecin nevizitat al vârfului curent, **dacă există**
  - altfel
    - **se merge înapoi pe drumul de la s la vârful curent, până se ajunge la un vârf cu vecini nevizitați**
    -

# Parcurgerea în adâncime

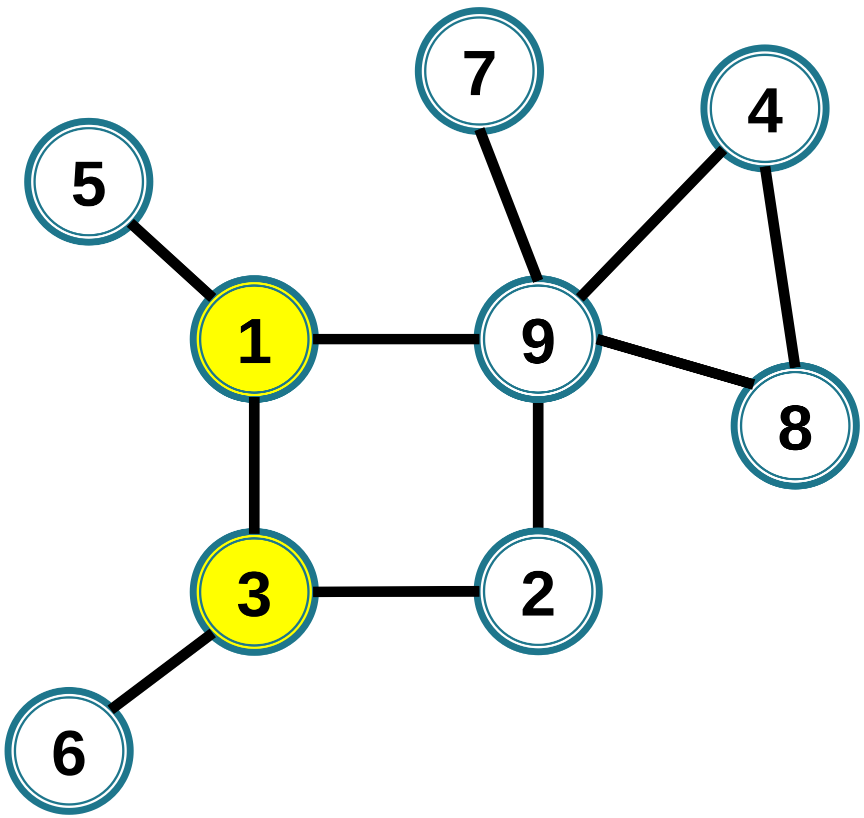
Se vizitează

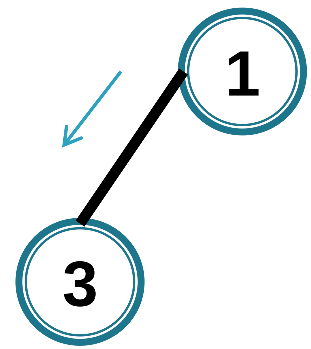
- **Inițial**: vârful de start s – devine vârf curent
- **La un pas**:
  - se trece la primul vecin nevizitat al vârfului curent, **dacă există**
  - altfel
    - **se merge înapoi pe drumul de la s la vârful curent, până se ajunge la un vârf cu vecini nevizitați**
    - **se trece la primul dintre aceștia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

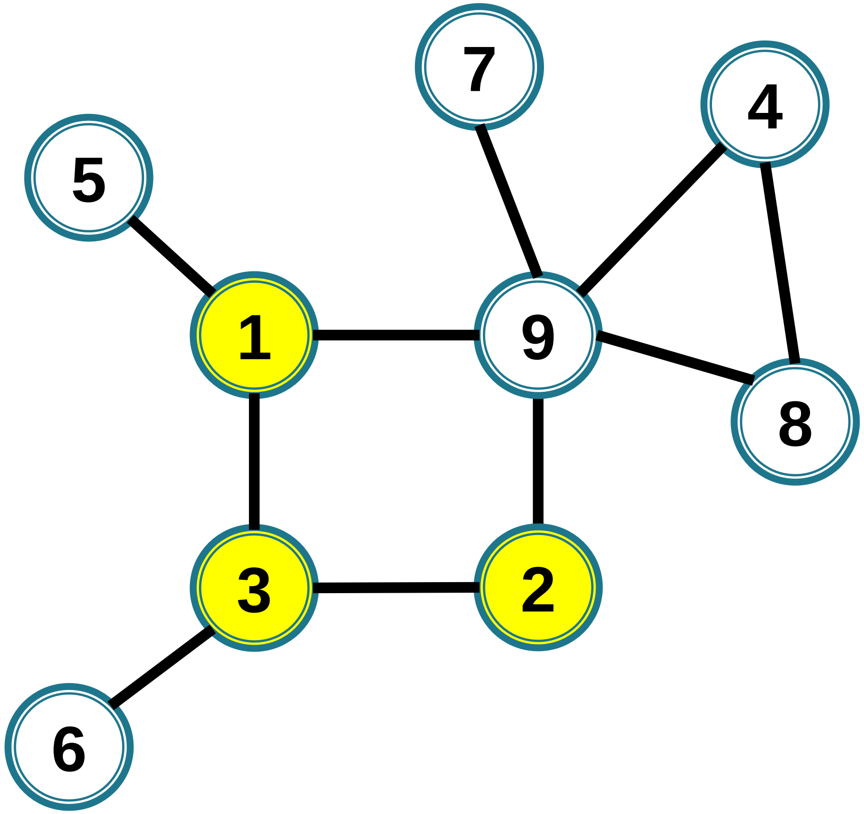

7
4
5
1
9
8
3
2
6


7
4
5
1
9
8
3
2
6

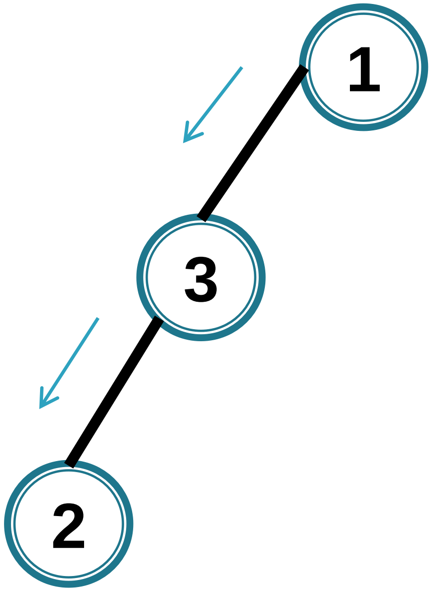

1


5
7
4
1
9
8
3
2
6

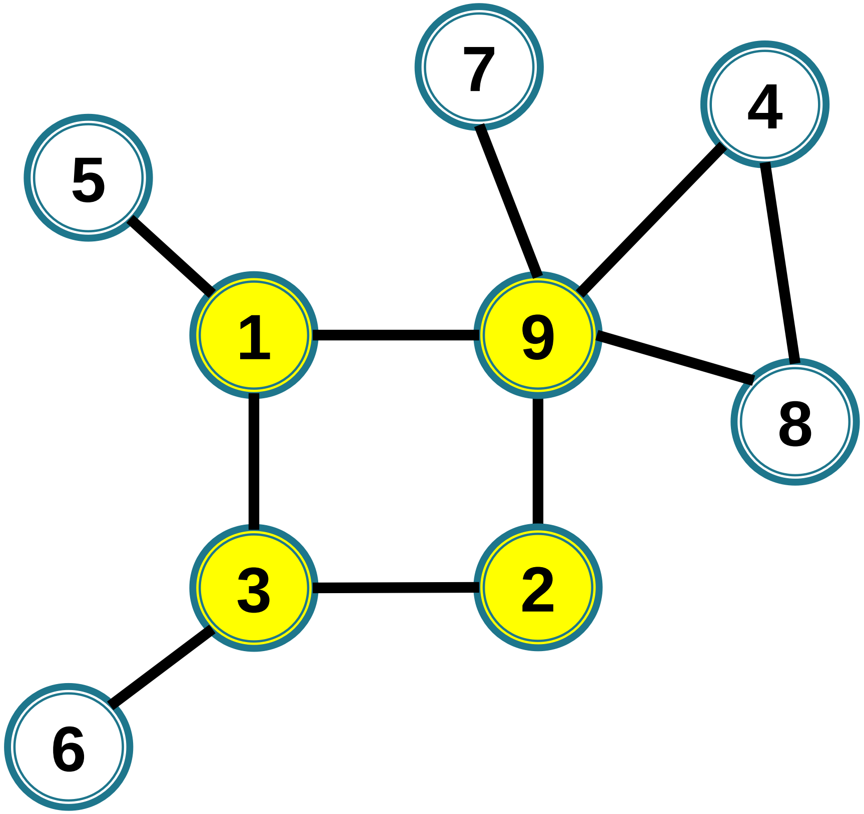

1
3

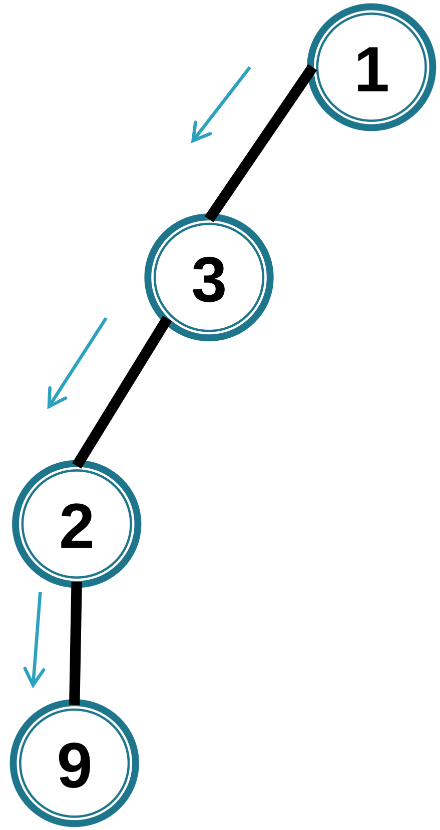


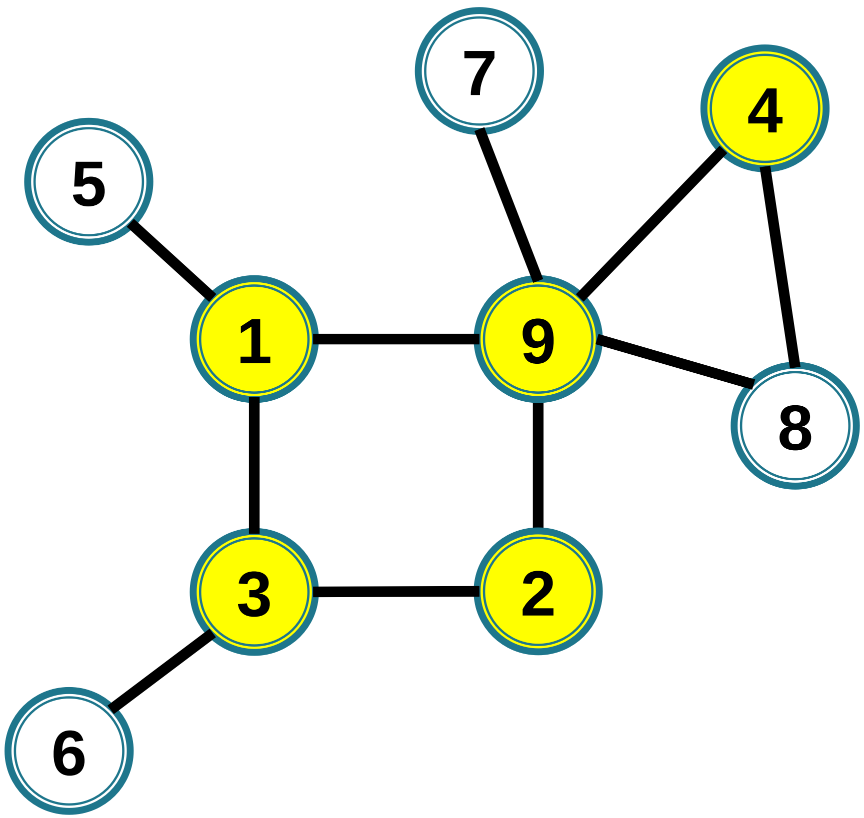
7
4
5
1
9
8
3
2
6

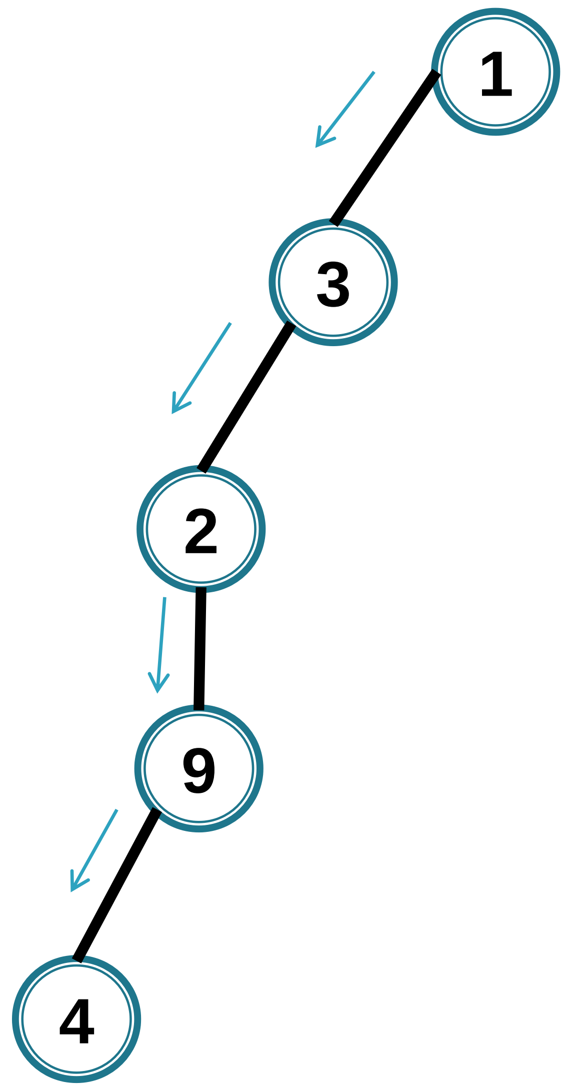


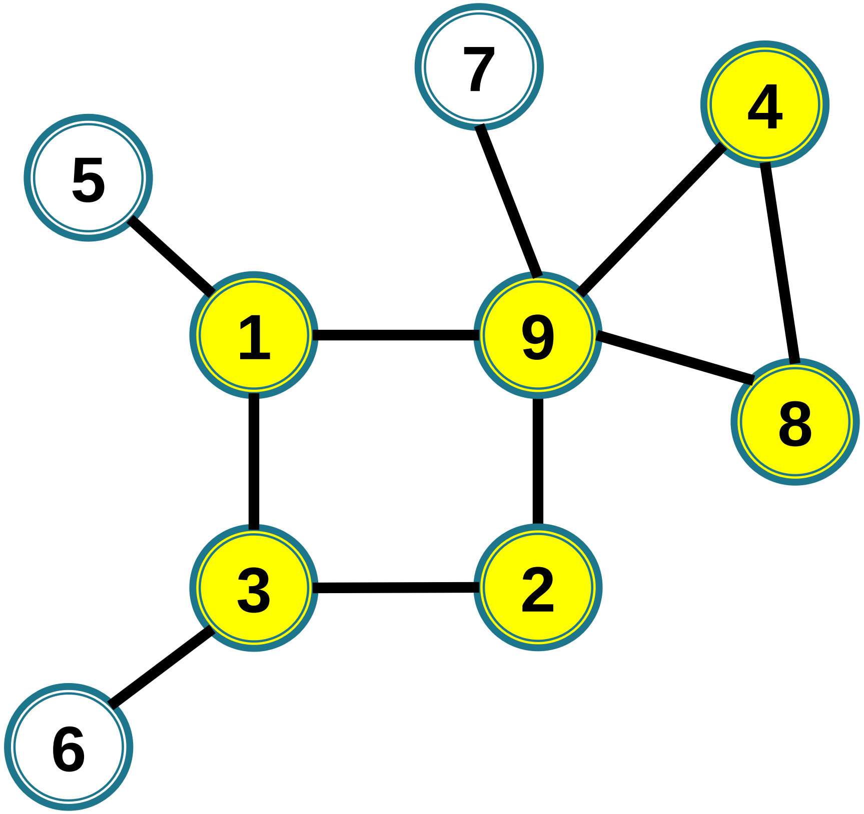
1
3
2

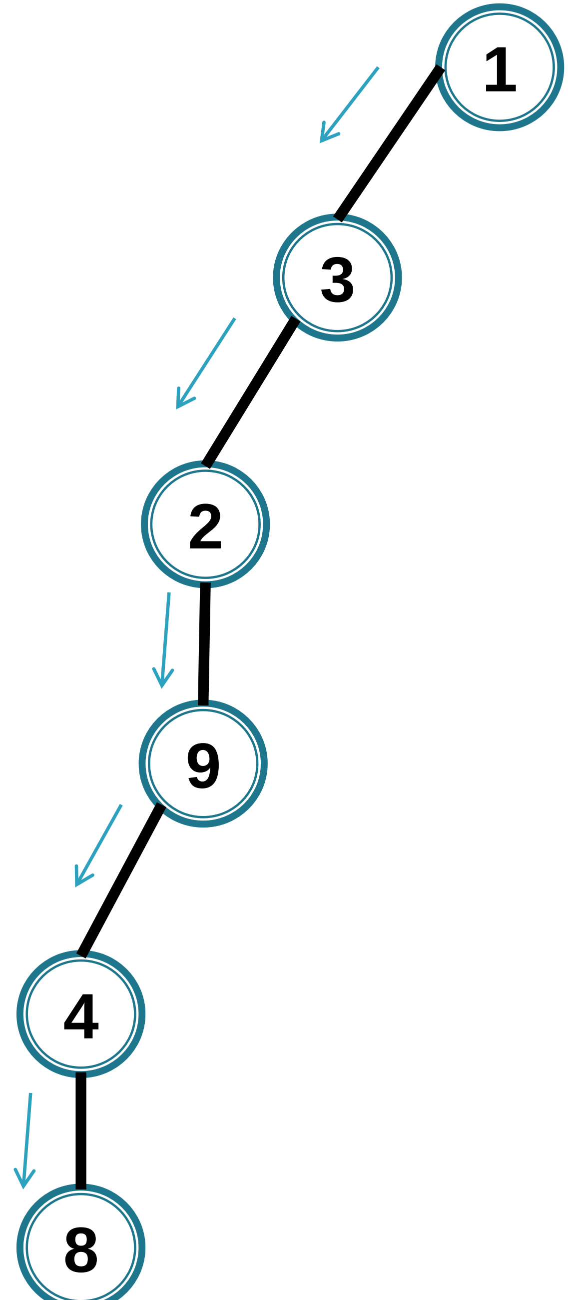

7
4
5
1
9
8
3
2
6

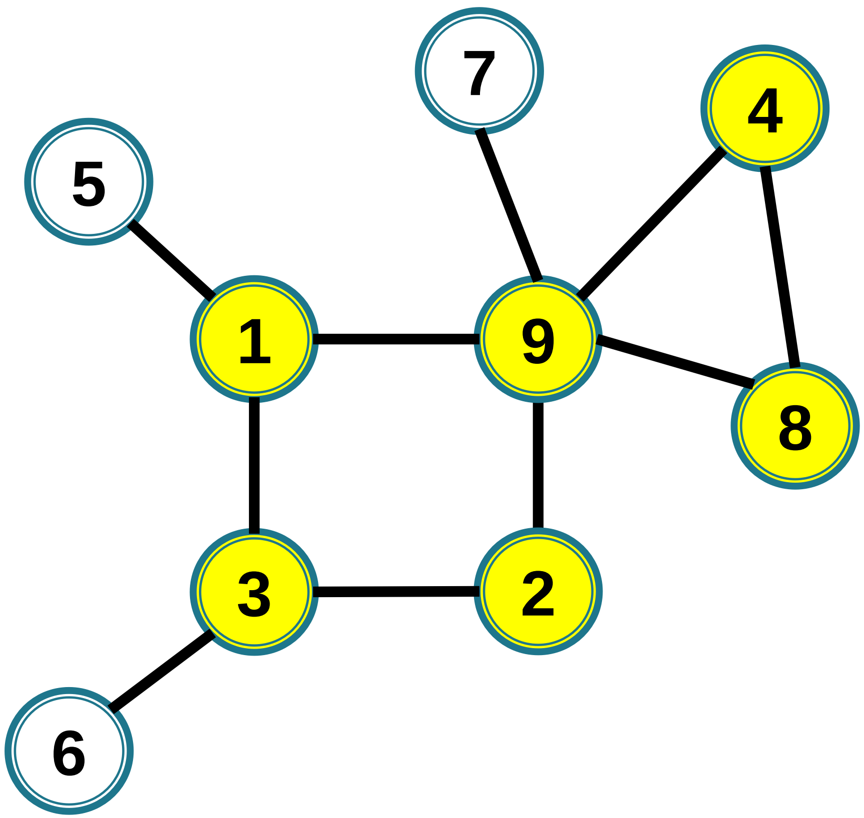

1
3
2
9

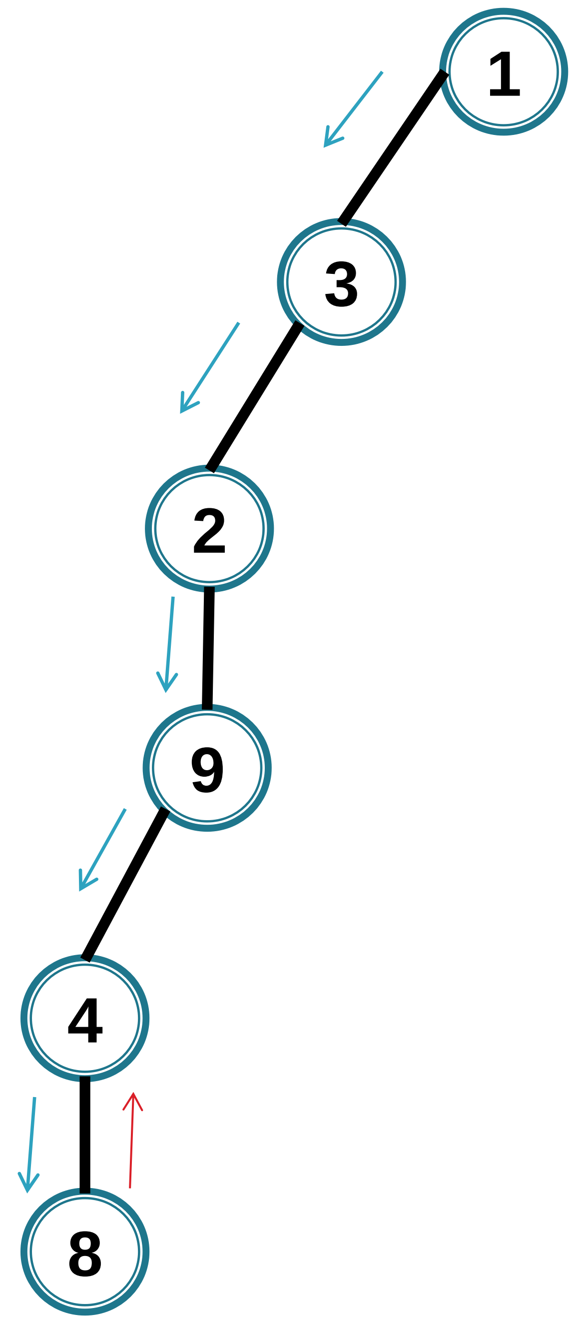

7
4
5
1
9
8
3
2
6

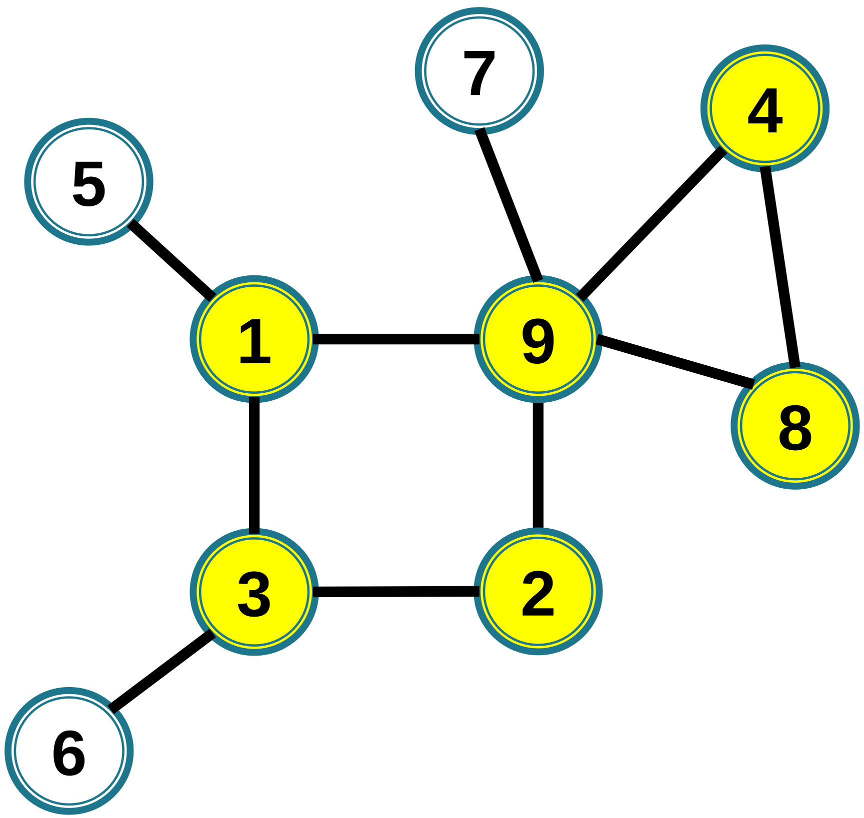

1
3
2
9
4

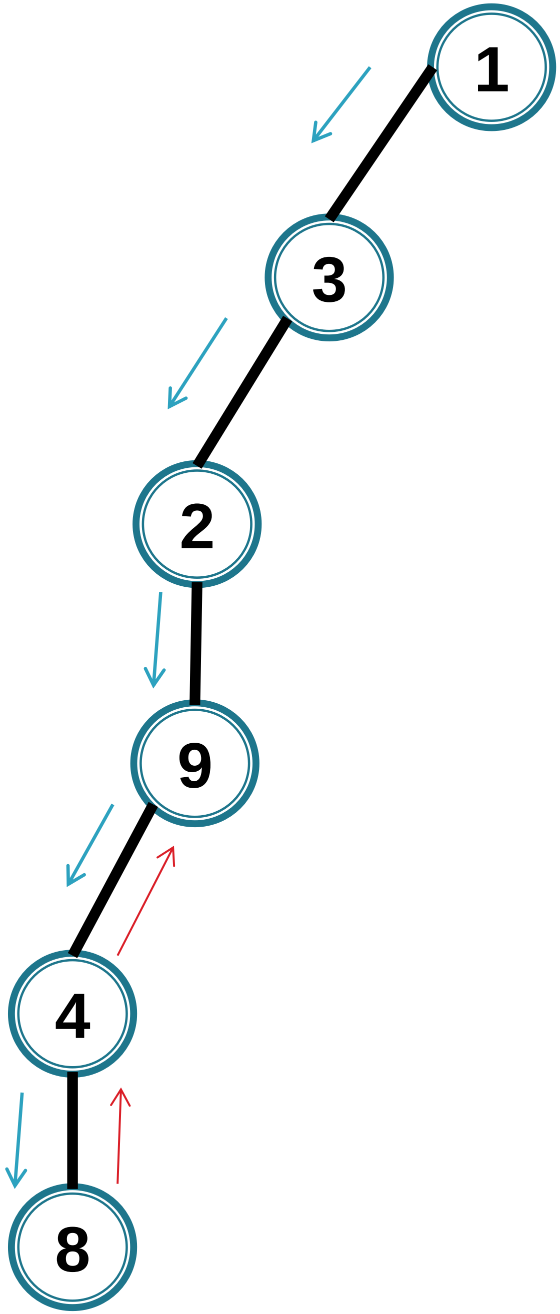

7
4
5
1
9
8
3
2
6

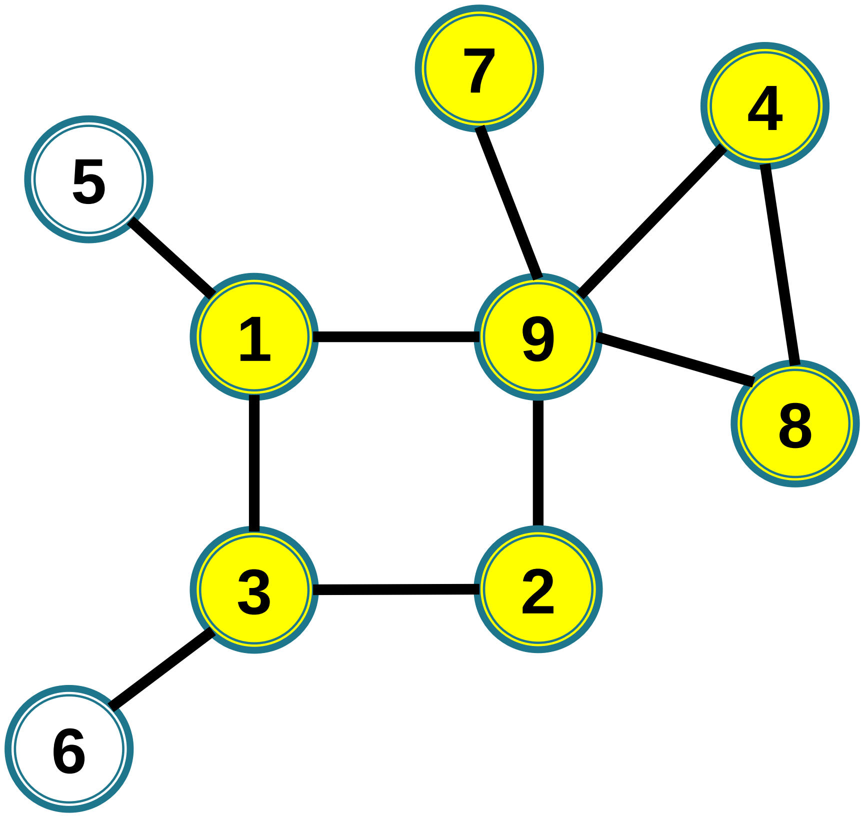

1
3
2
9
4
8

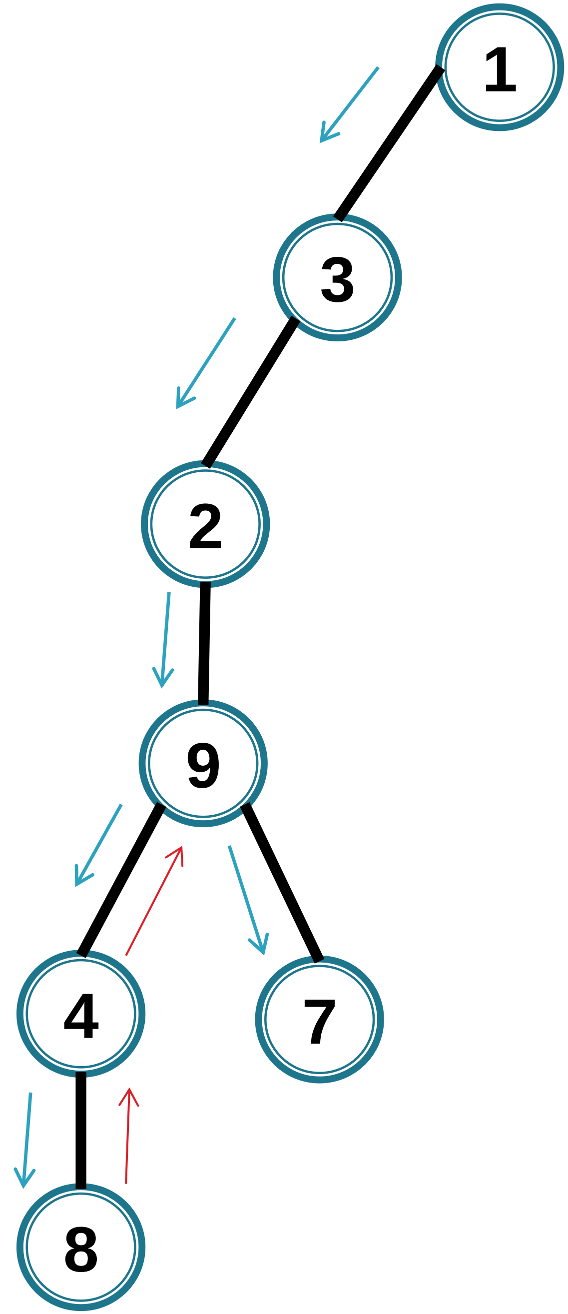

7
4
5
1
9
8
3
2
6

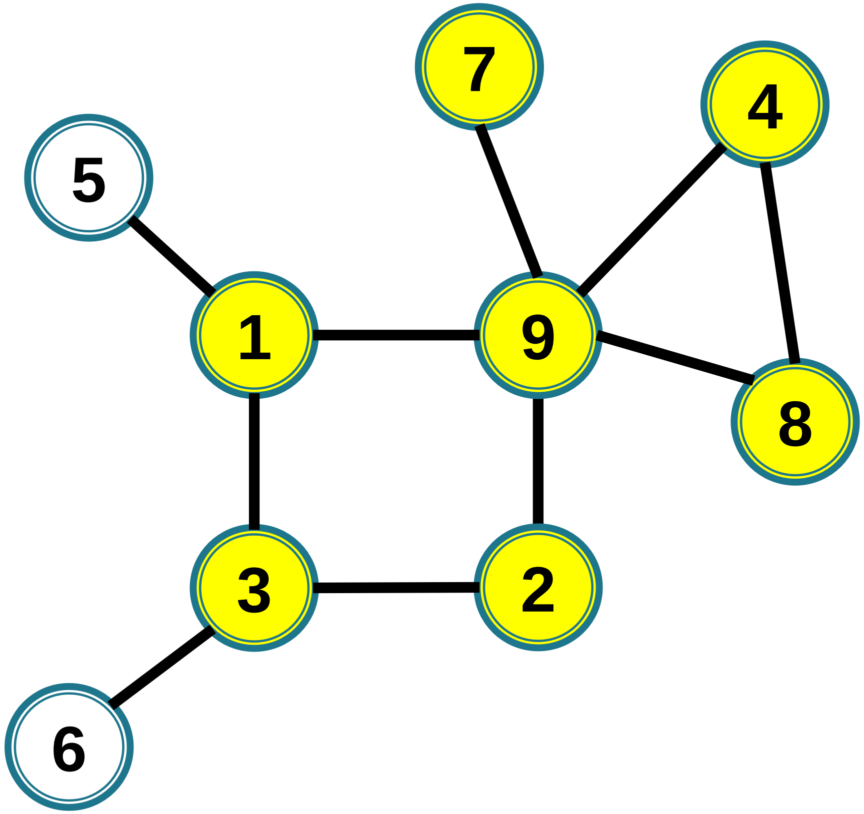

1
3
2
9
4
8

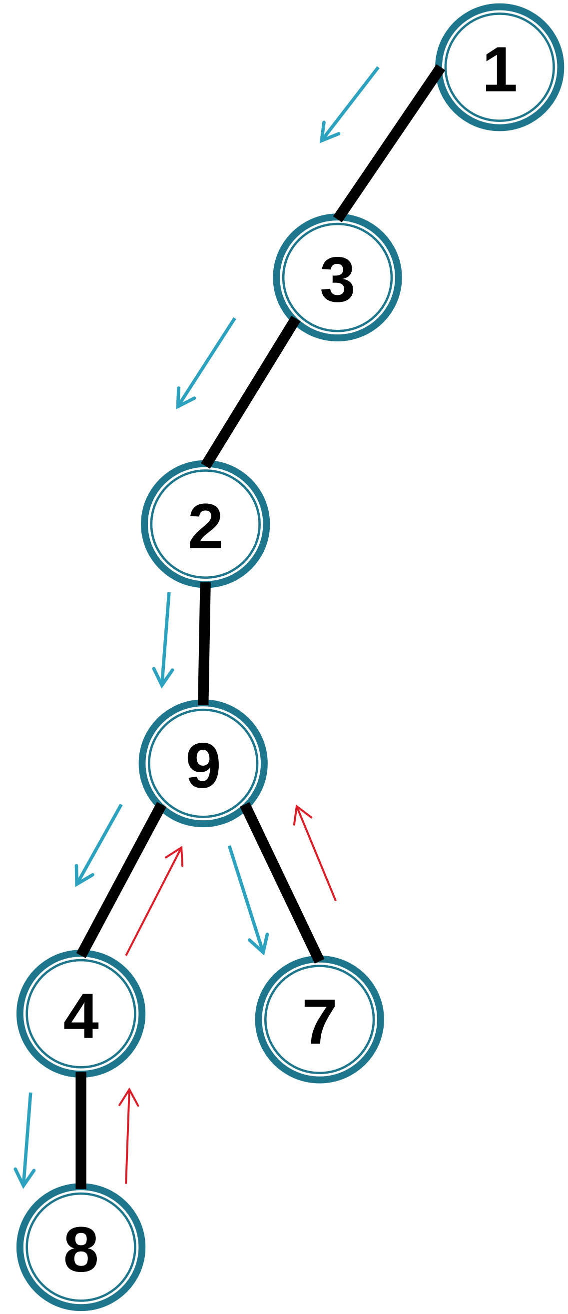

7
4
5
1
9
8
3
2
6

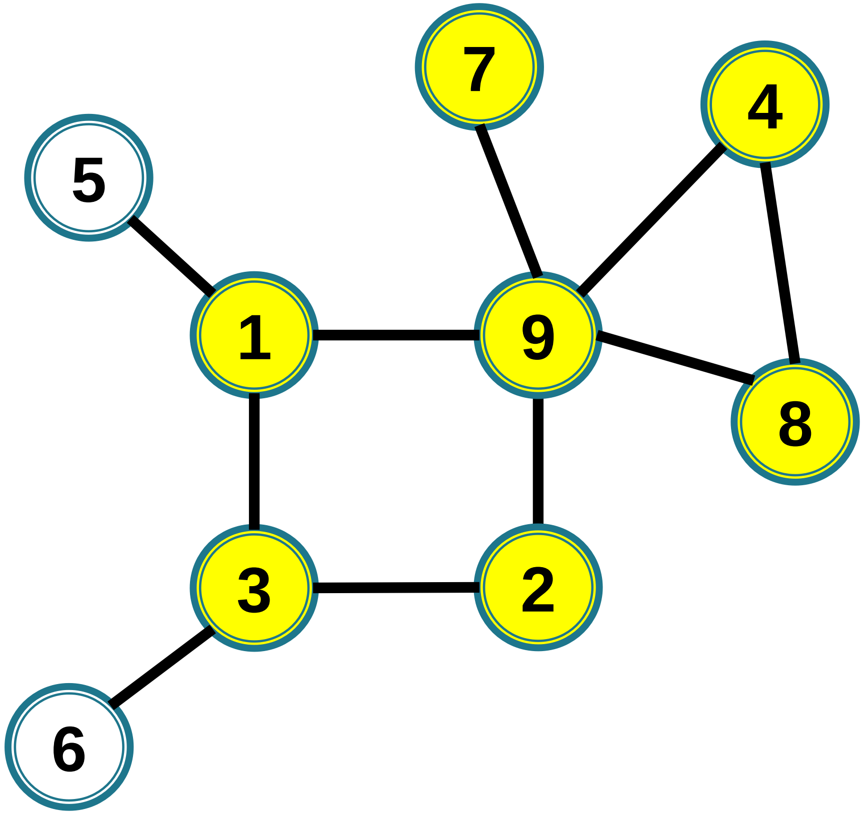

1
3
2
9
4
8


7
4
5
1
9
8
3
2
6

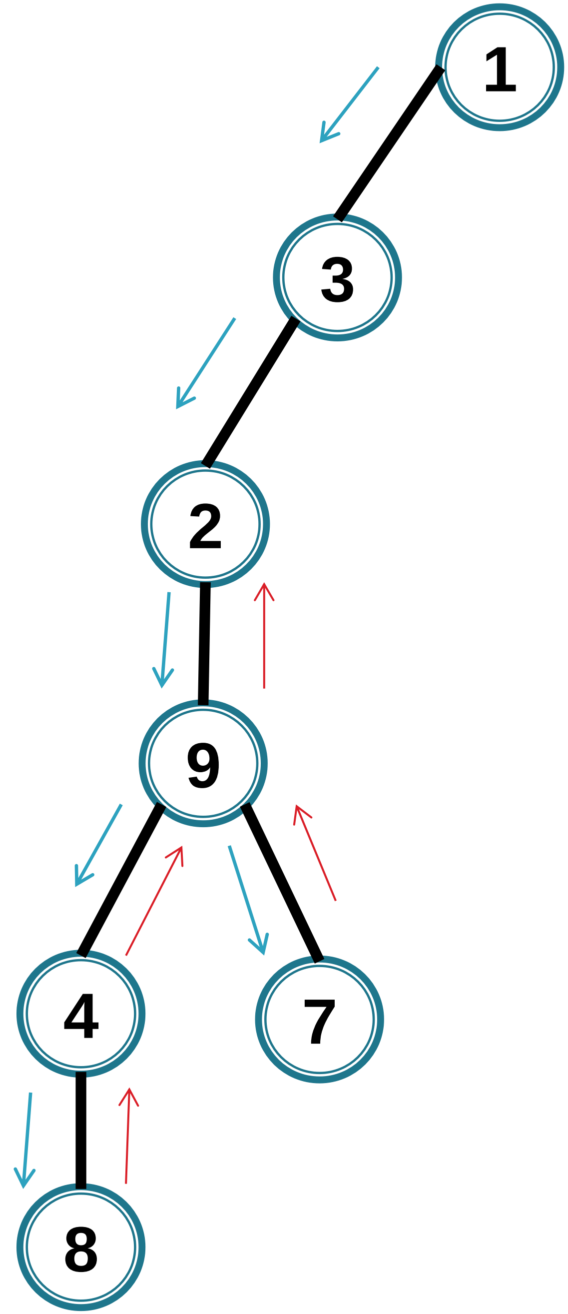

1
3
2
9
4
7
8


7
4
5
1
9
8
3
2
6

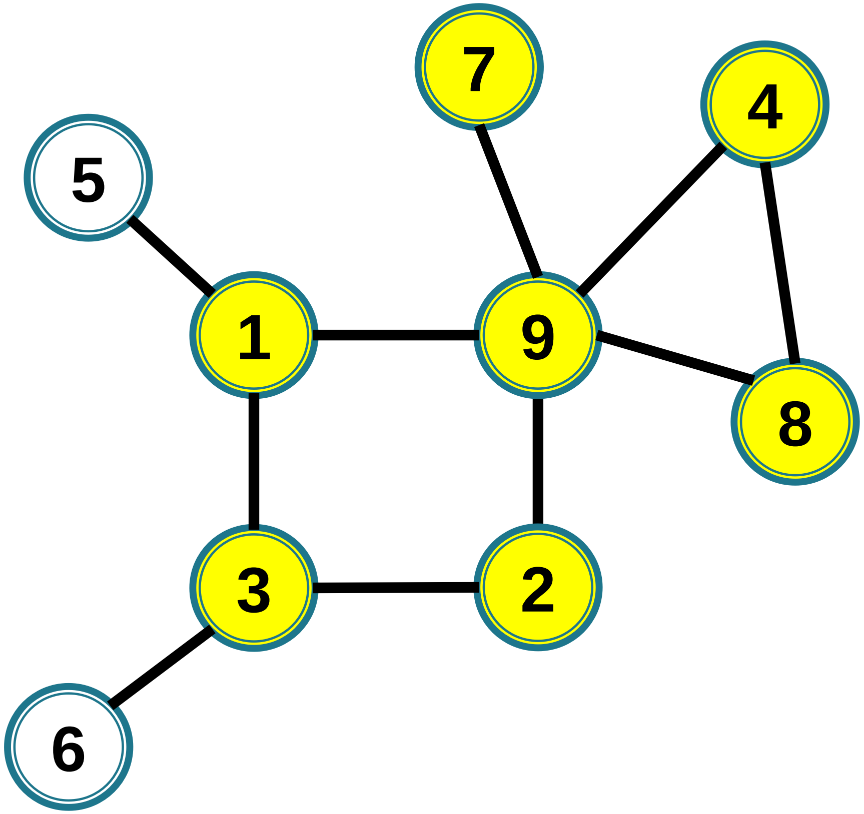

1
3
2
9
4
7
8

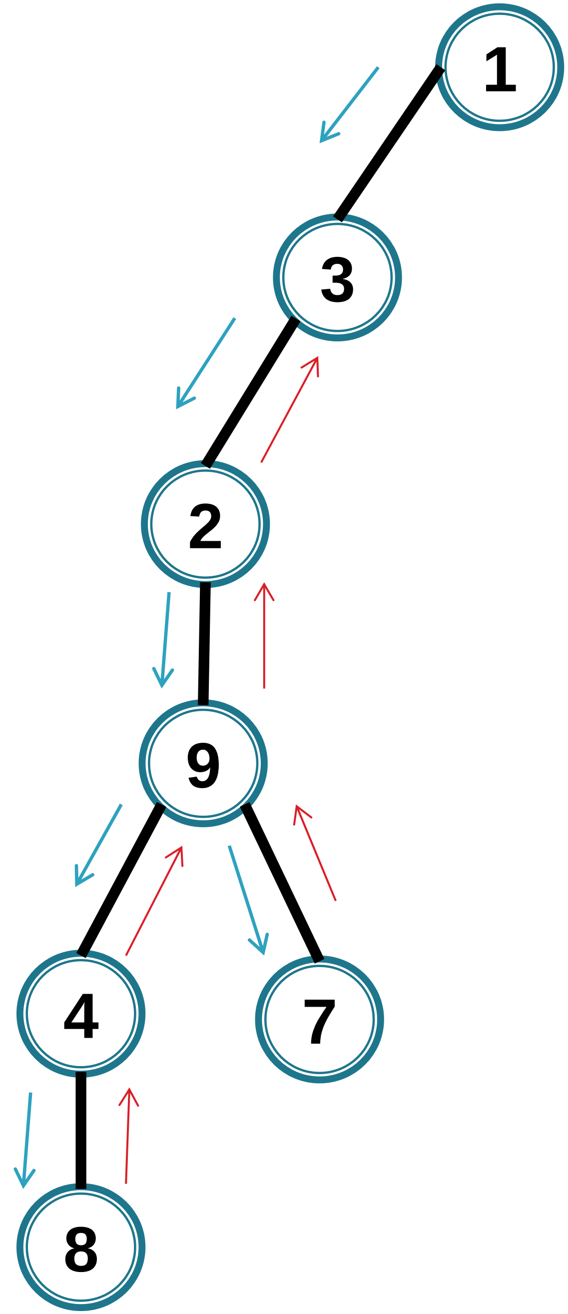


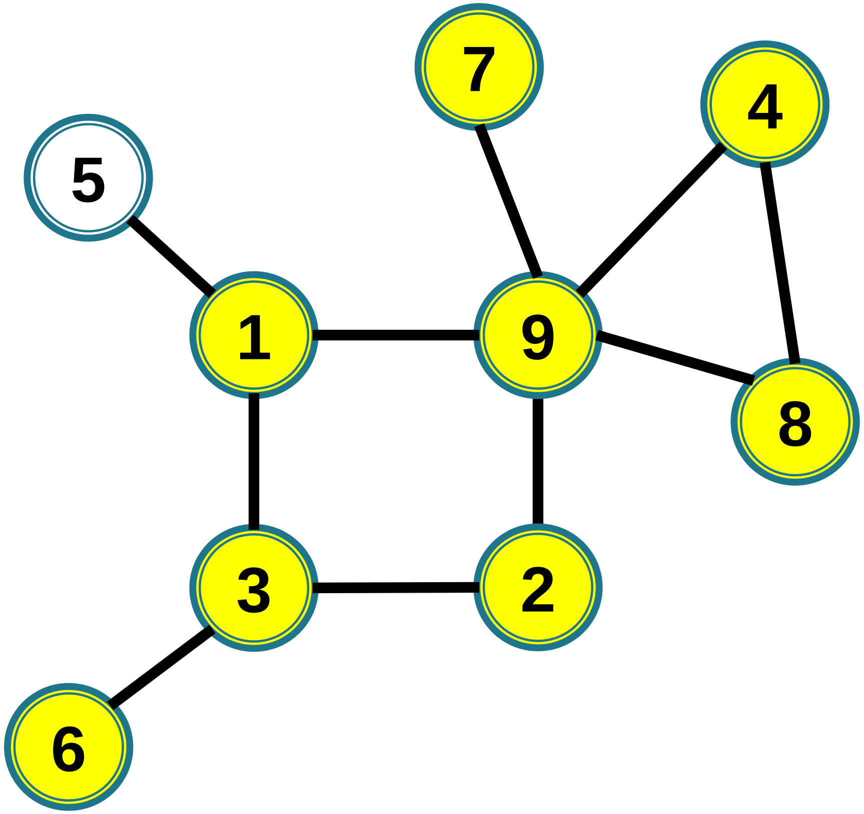
7
4
5
1
9
8
3
2
6

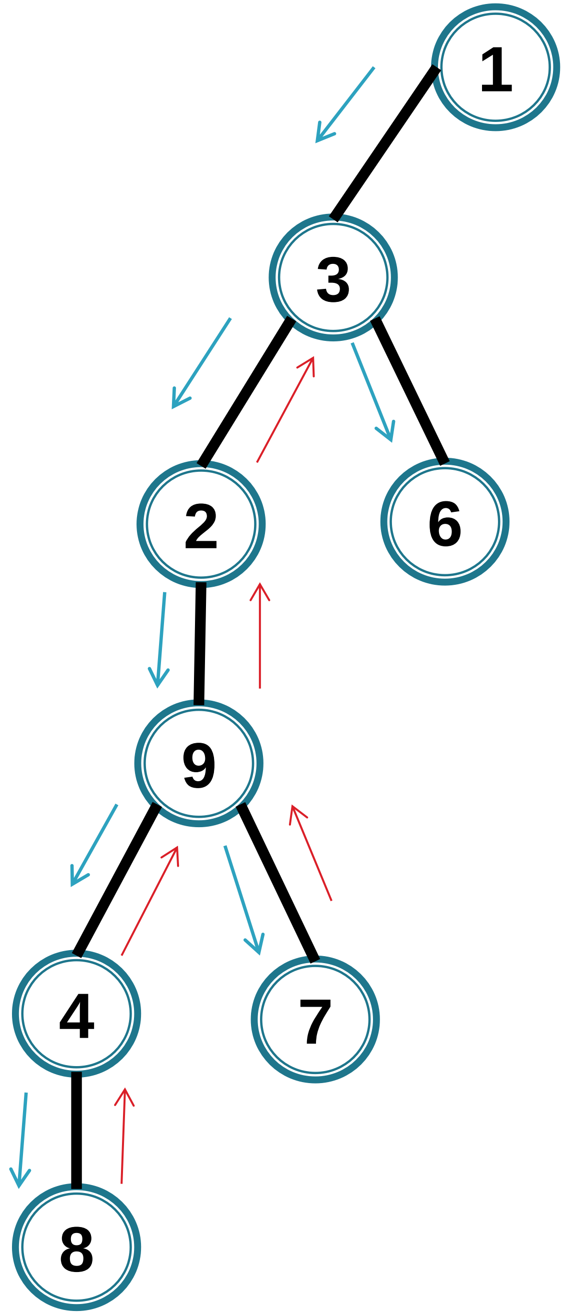



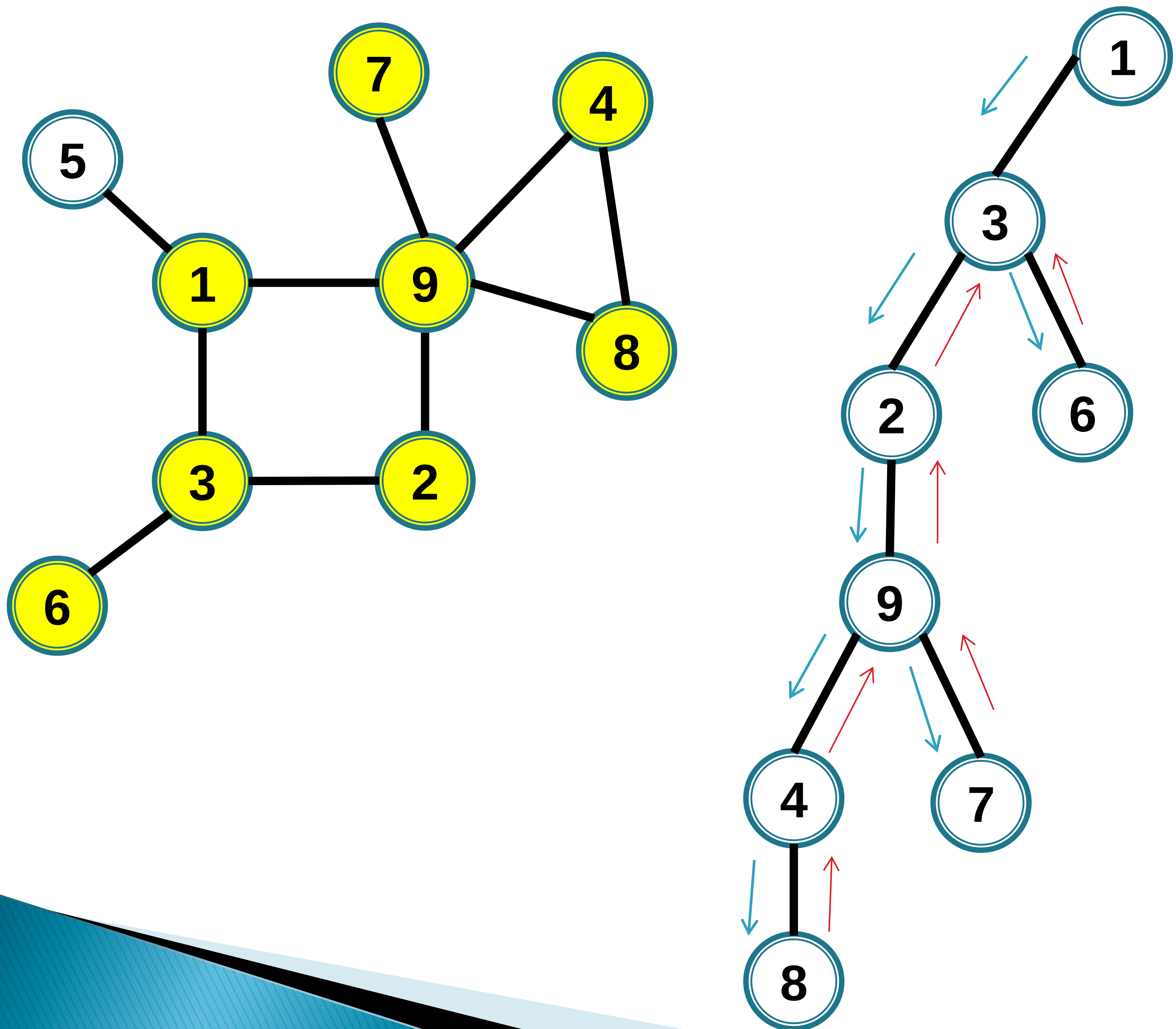
1
3
2
9
4
7
8


7
4
5
1
9
8
3
2
6

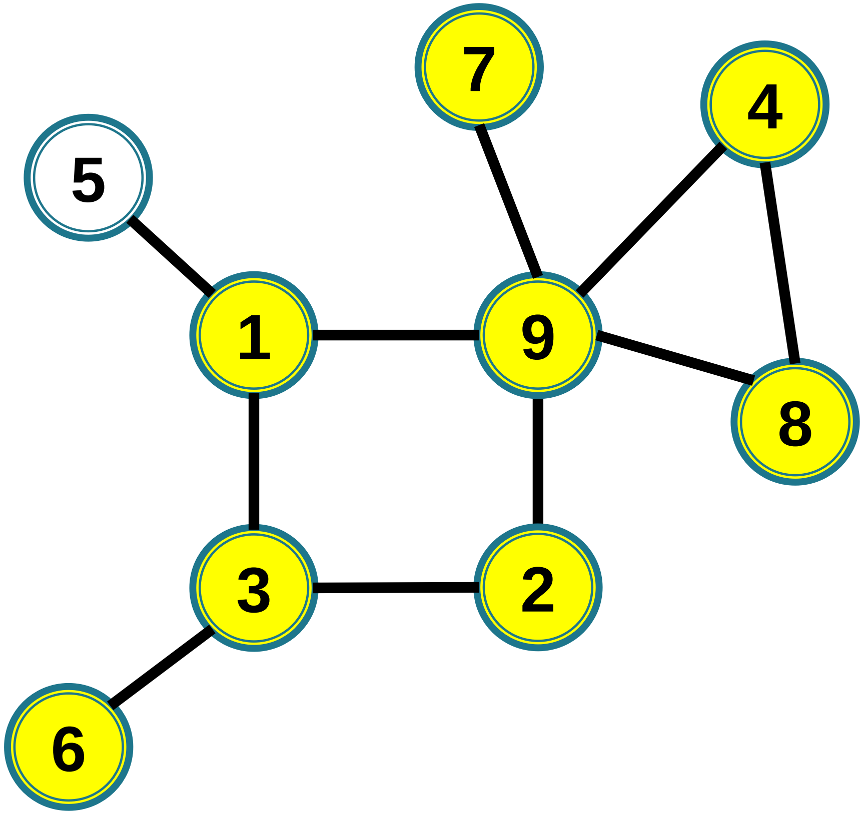

1
3
2
9
4
7
8

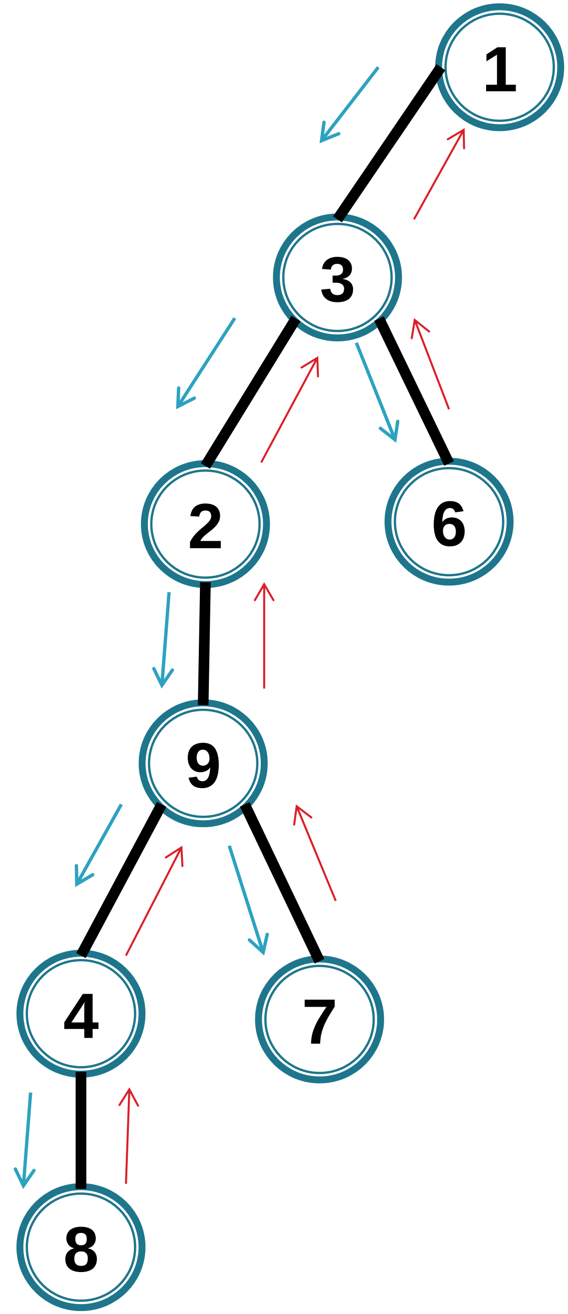

7
4
5
1
9
8
3
2
6

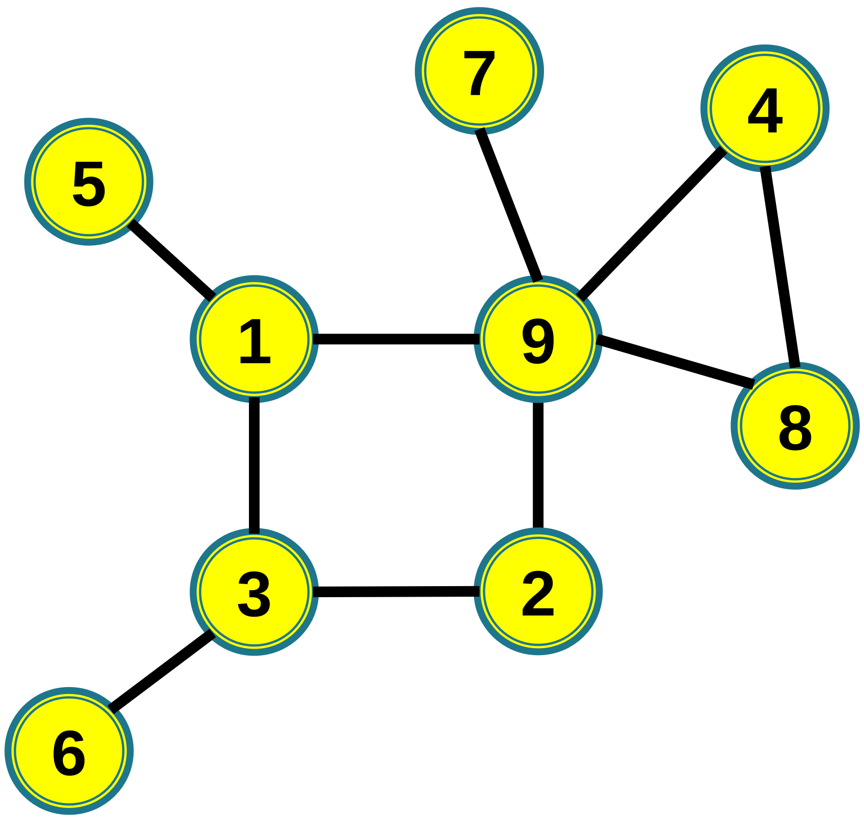

1
3
2
6
9
4
7
8

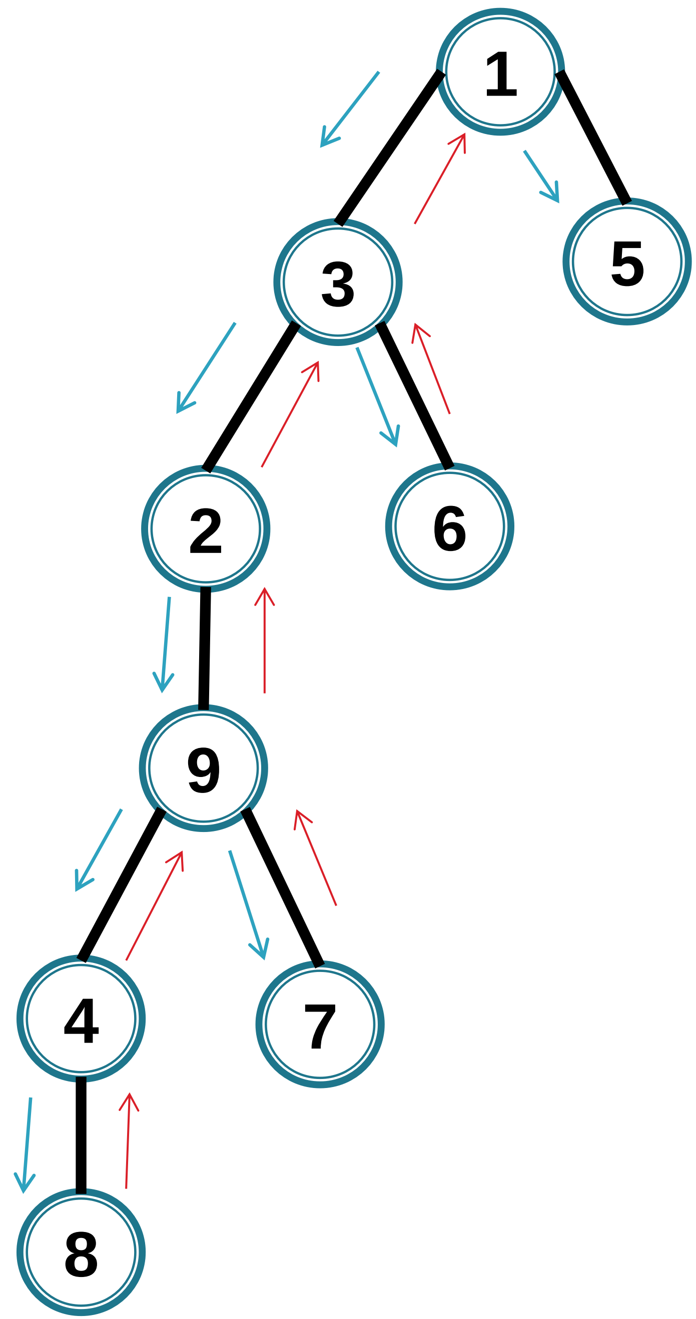


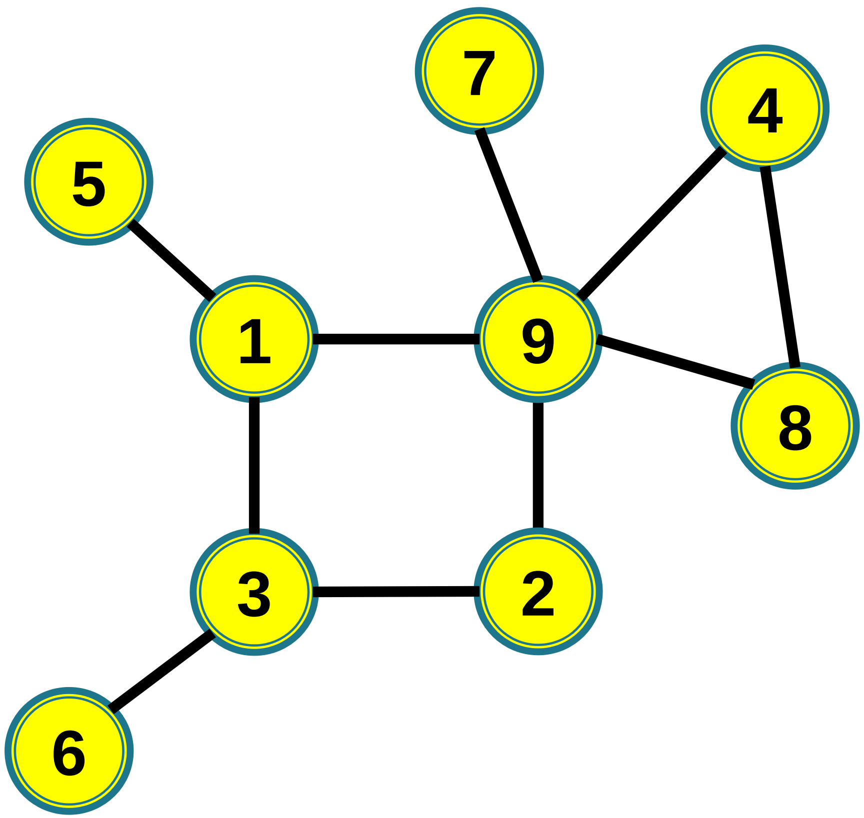
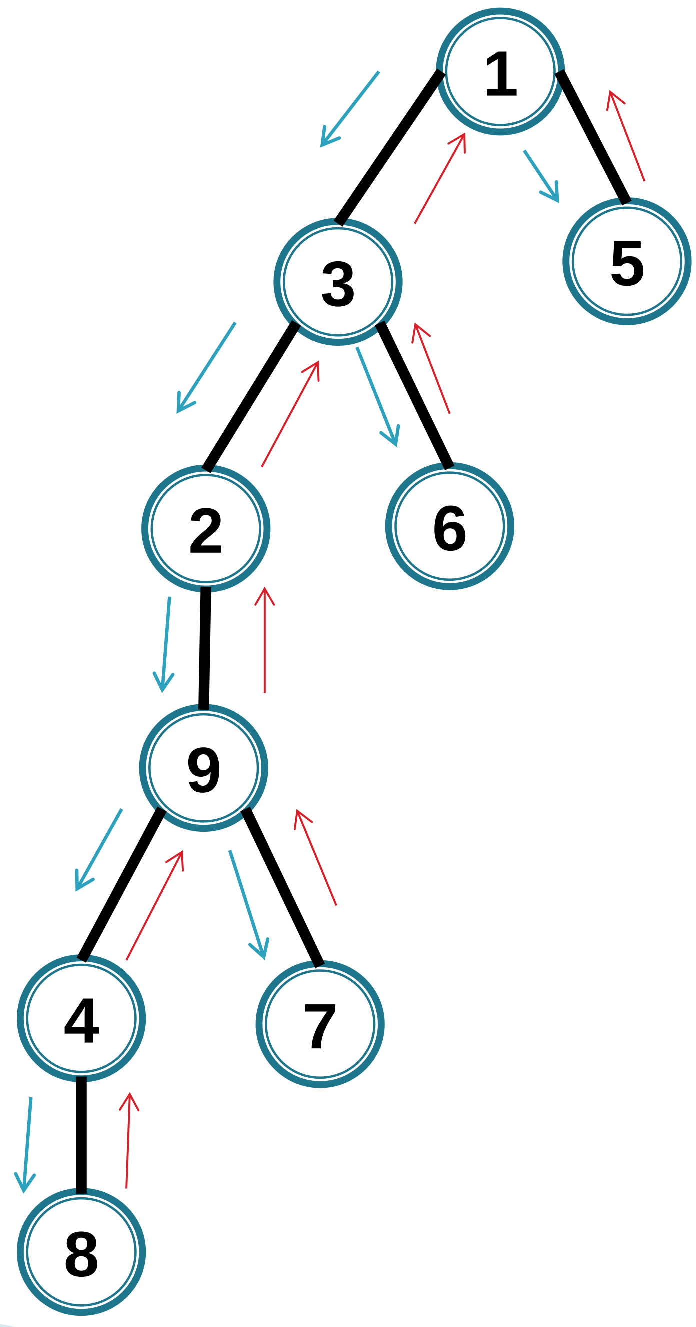
7
4
5
1
9
8
3
2
6
1
3
2
6
9
4
7
8

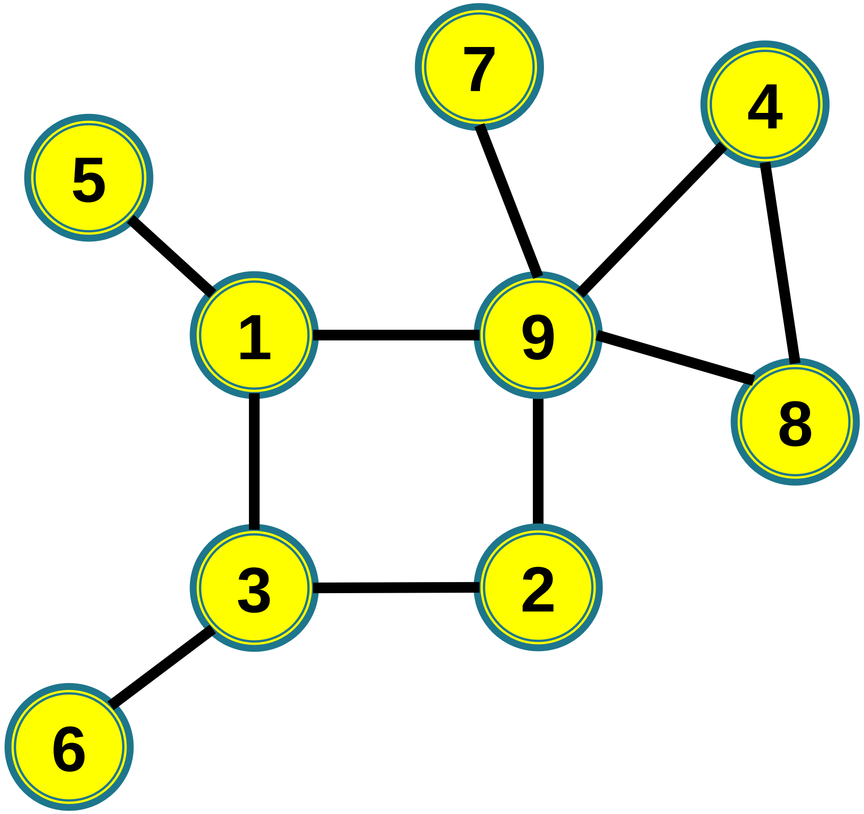

5
7
4
1
9
8
3
2
6


1
3
2
6
9
4
7
8

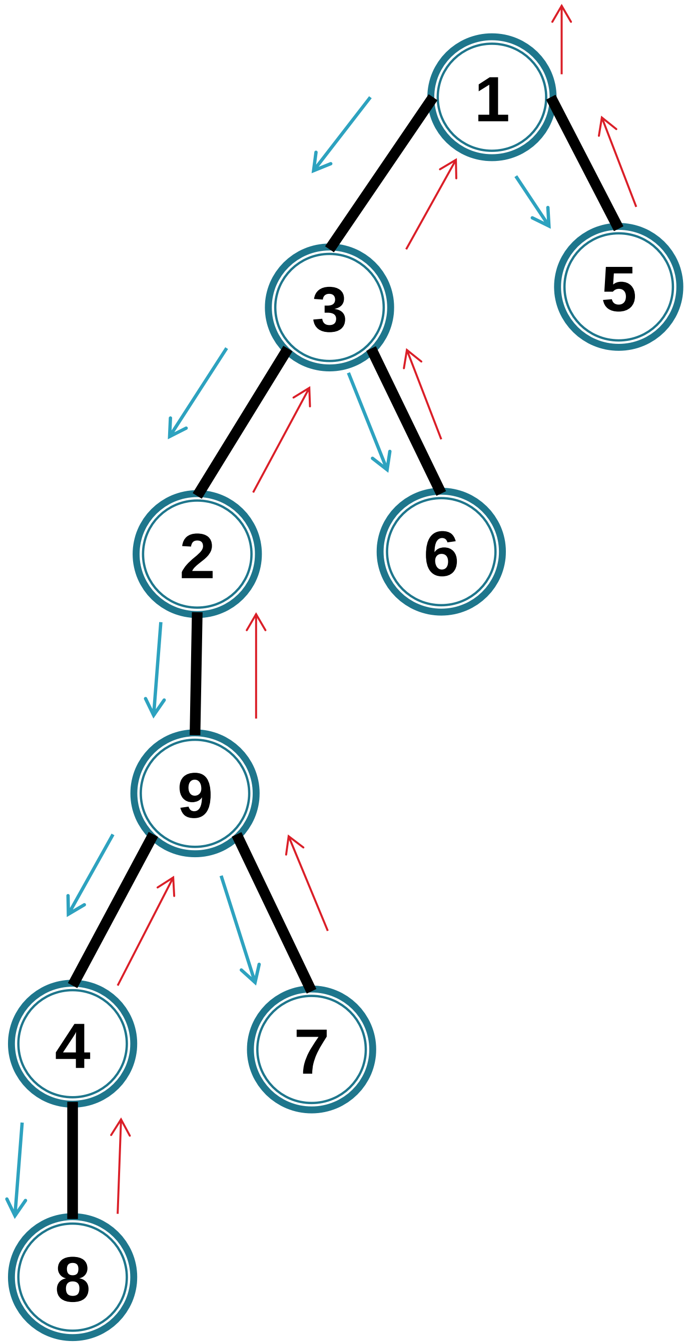

5
7
4
1
9
8
3
2
6


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

```
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**:

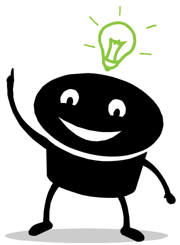
```
     df(s)
```

# Alte aplicații

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

# Alte aplicații

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



Un ciclu se închide în parcurgere când vârful curent are un vecin deja vizitat,  care nu este tatăl lui

# Alte aplicații

Să se verifice dacă un graf neorientat dat este bipartit

# Graf bipartit

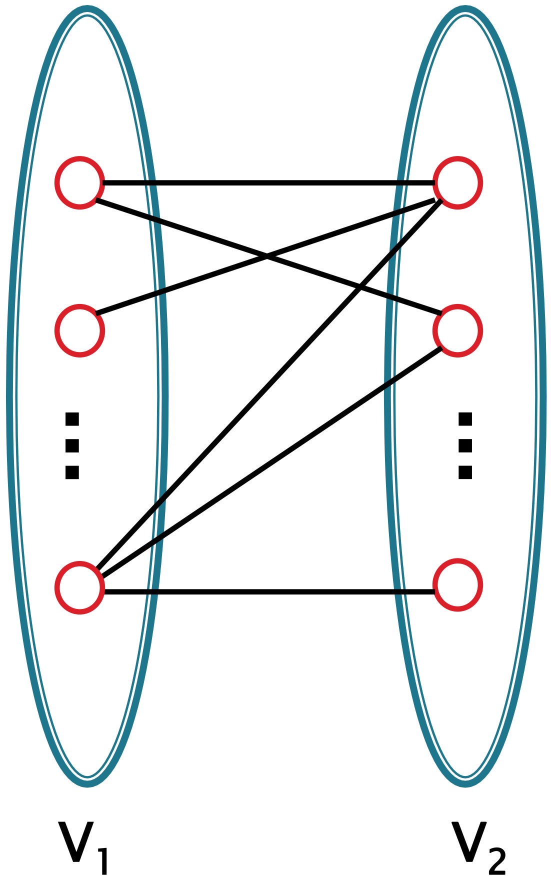
- Un graf neorientat $G = (V, E)$ se numește **bipartit** $\Leftrightarrow$ există o partiție a lui V în două submulțimi nevide $V_1$, $V_2$ (**bipartiție**):

$$V = V_1 \cup V_2$$

$$V_1 \cap V_2 = \emptyset$$

astfel încât orice muchie $e \in E$ are o extremitate în $V_1$ și cealaltă în $V_2$:

$$|e \cap V_1| = |e \cap V_2| = 1$$


$V_1$
$V_2$

# Graf bipartit

- $G = (V, E)$ **bipartit** $\Leftrightarrow$ există o colorare a vârfurilor cu două culori:

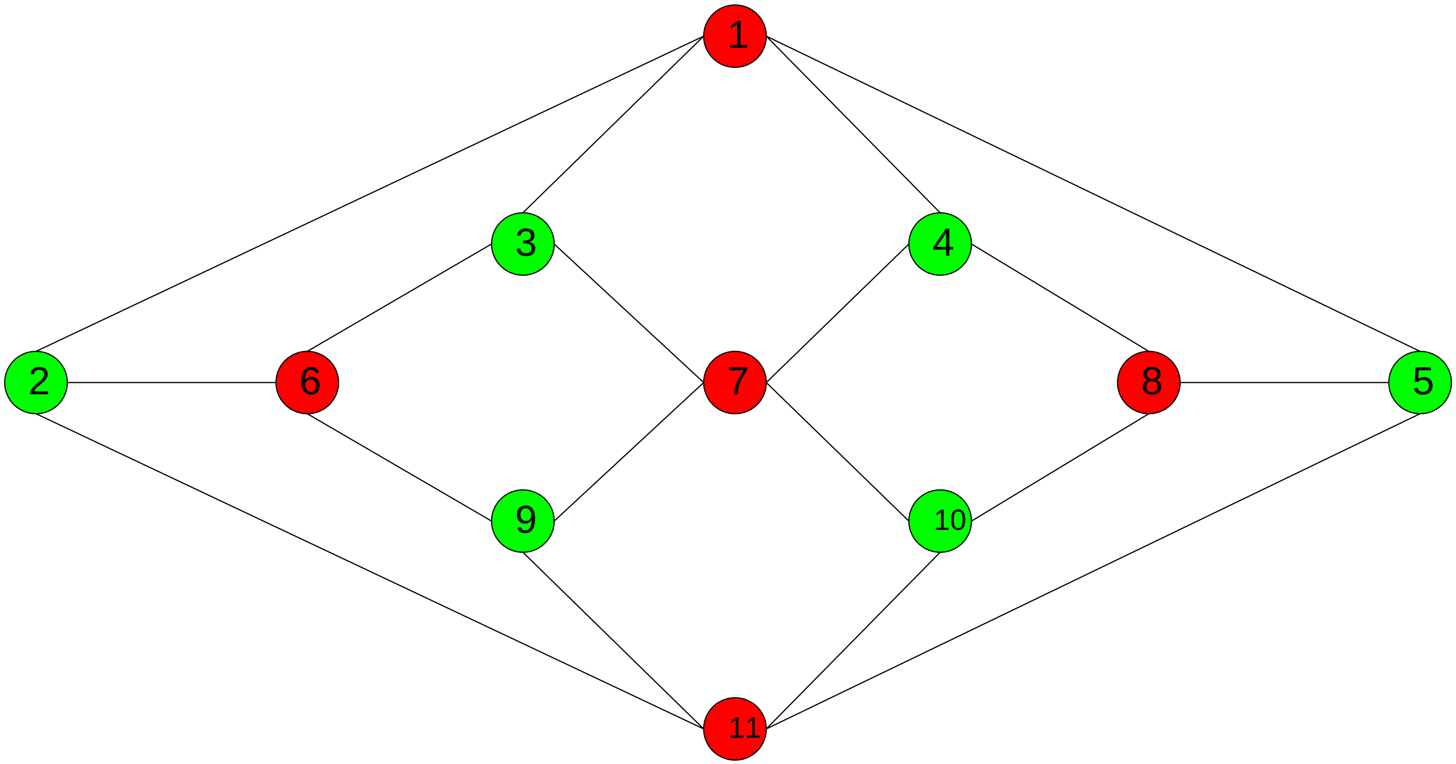
$$c : V \rightarrow \{1, 2\}$$

astfel încât pentru orice muchie $e=xy \in E$ avem

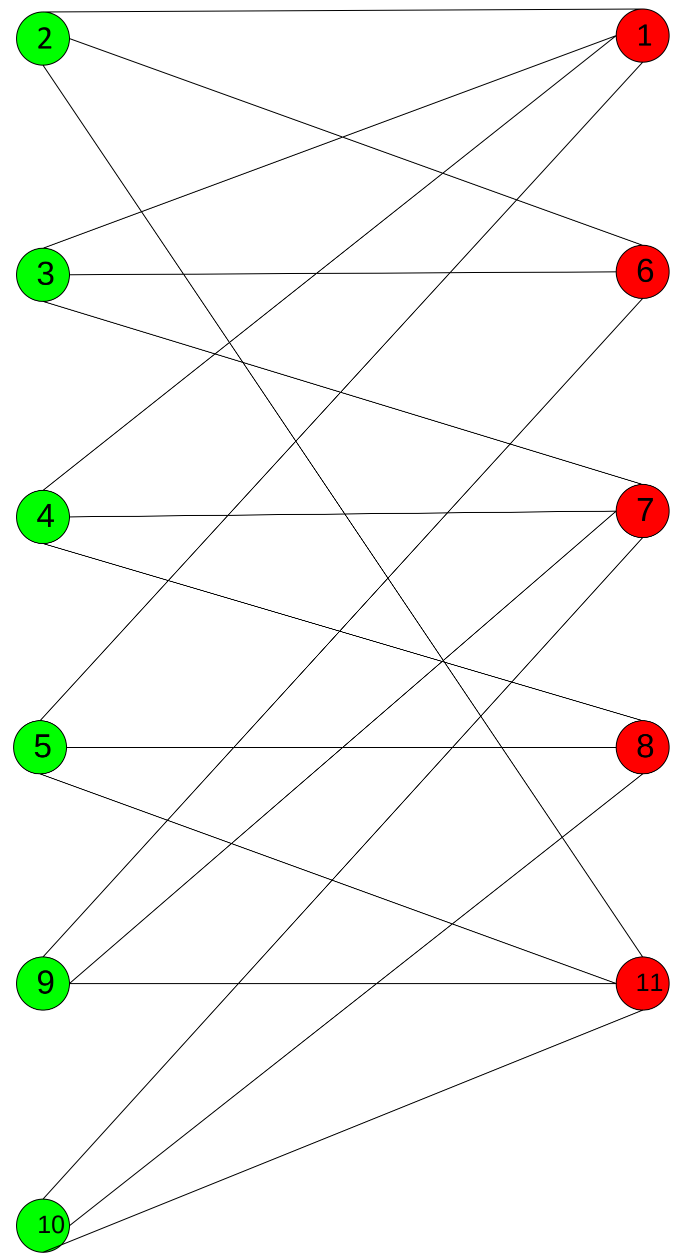
$$c(x) \neq c(y)$$

(**bicolorare**)

# Graf bipartit



# Graf bipartit



# Graf bipartit

- **Propoziție**

  Un arbore este graf bipartit

# Graf bipartit

- **Teorema König – Caracterizarea grafurilor bipartite**

  Fie $G = (V, E)$ un graf simplu cu $n \geq 2$ vârfuri.

  Avem

  $G$ este bipartit $\Leftrightarrow$ toate ciclurile elementare din $G$ sunt pare