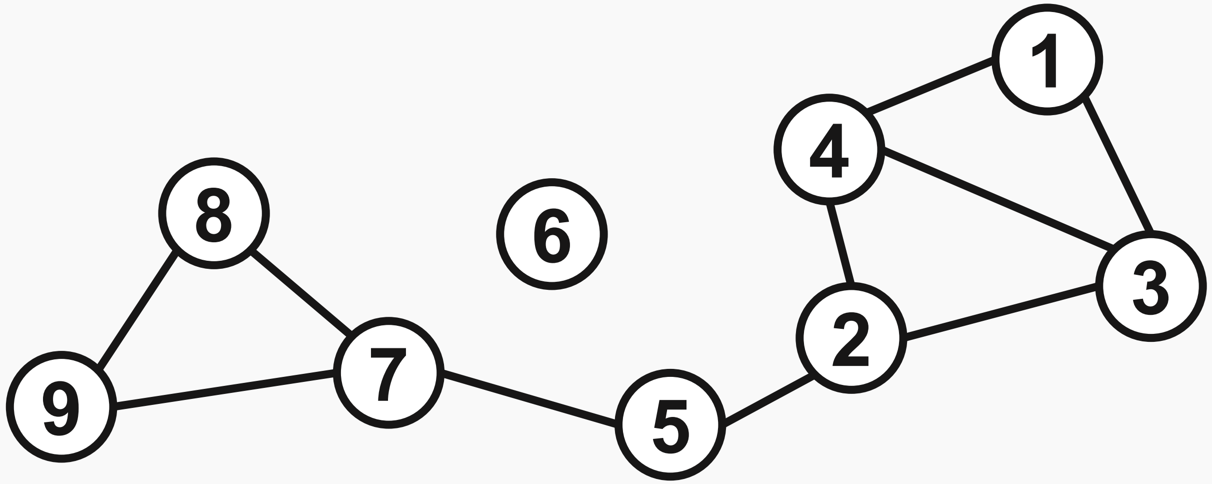
Structuri de date și algoritmi
Grafuri

Lect. Dr. Ing. Cristian Chilipirea – cristian.chilipirea@mta.ro

Graf definiție

Graf $G = (V, E)$. Unde:

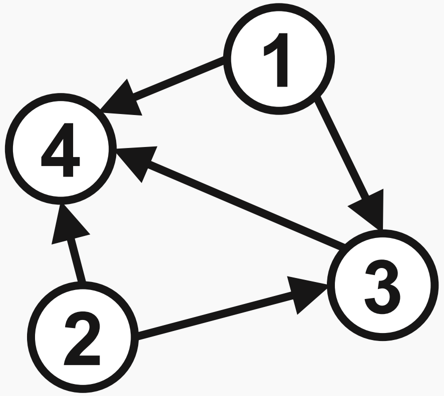
- V – setul de noduri – Vertex
- E – setul de muchii – Edges
  - $E \subseteq \{(x, y) | (x, y) \in V^2 \wedge x \neq y\}$



Graf orientate vs Graf neorientate

Muchiile au direcție

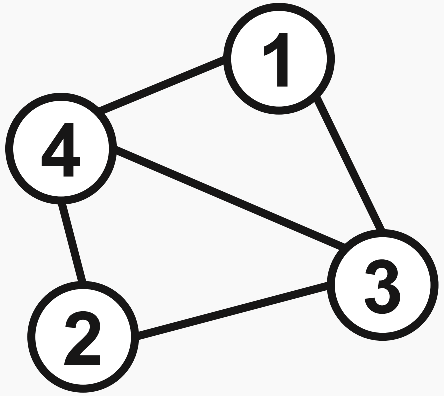
Nu se poate ajunge din 3 în 1



Toate muchiile sunt bidirecționale

$$\forall (x, y) \in E \; \rightarrow \; \exists (y, x) \in E$$

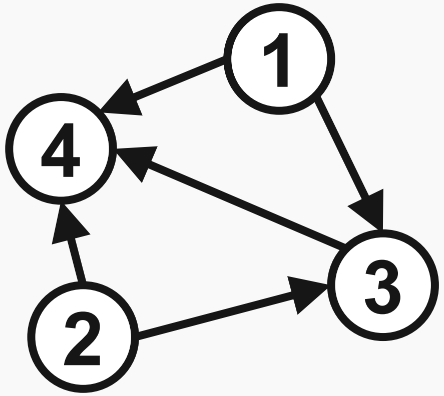
Convenția permite memorarea doar uneia din cele 2 muchii

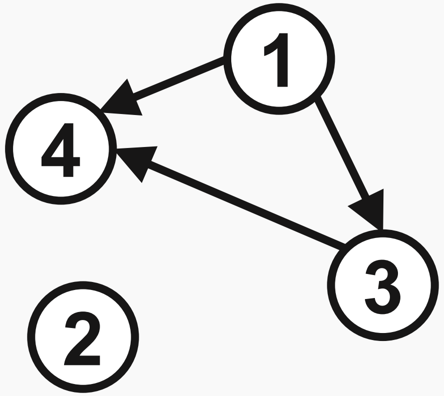


Graf parțial vs Subgraf

$$G_1 = (V_1, E_1)$$
$$G_2 = (V_2, E_2)$$
$$V_2 = V_1$$
$$E_2 \subseteq E_1$$

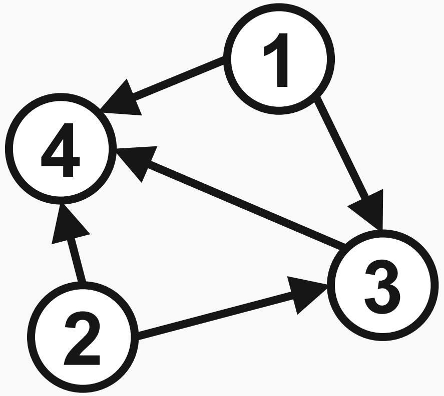
G_2 graf parțial pentru G_1

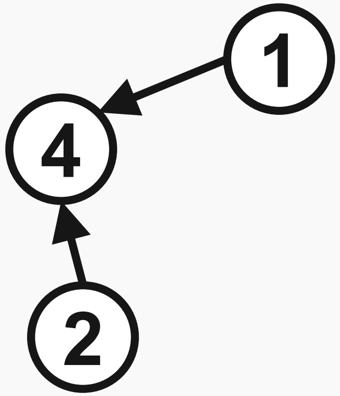




$$G_1 = (V_1, E_1)$$
$$G_2 = (V_2, E_2)$$
$$V_2 \subseteq V_1$$
$$E_2 \subseteq E_1$$

G_2 subgraf pentru G_1

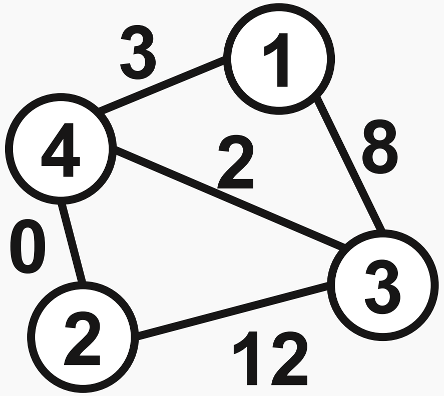




Grafuri ponderate – Weighted Graphs

Pentru un graf $G = (V, E)$ se adaugă funcția W ce asociază un cost fiecărei muchii.

$$w(1,4) = 3$$
$$w(2,3) = 12$$
$$w(2,4) = 0$$

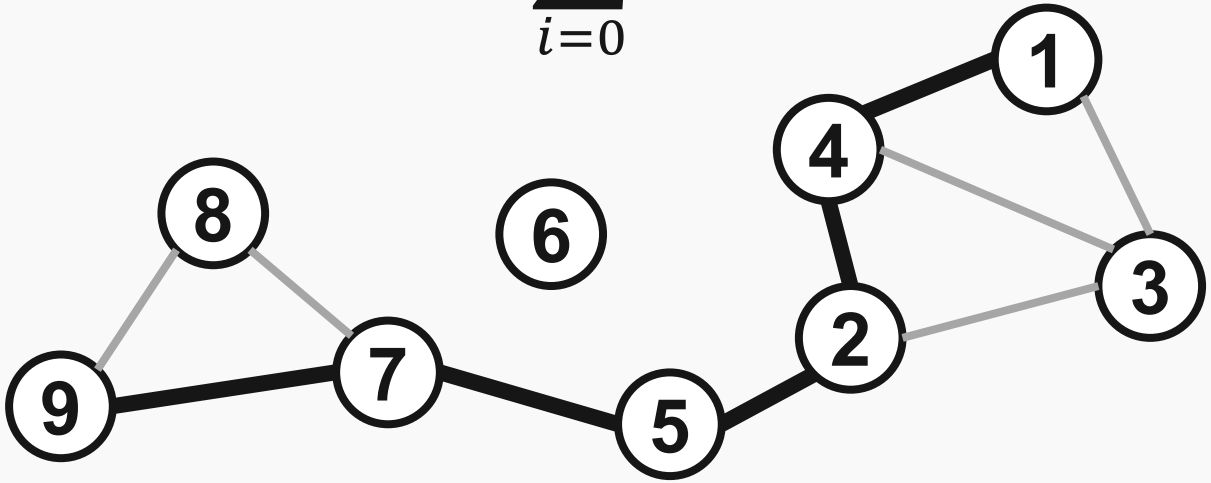


Drumuri - Paths

Pentru un graf $G = (V, E)$ un drum este un set de noduri $P = (v_1, v_2, v_3, \dots, v_N)$ cu $(v_i, v_{i+1}) \in E, \forall i$

Lungimea unui drum este:

$$w(P) = \sum_{i=0}^{N-1} w(v_i, v_{i+1})$$



Drumuri - Paths

Pentru un graf $G = (V, E)$ un drum este un set de noduri $\mathrm{P} = (v_1, v_2, v_3, \dots, v_N)$ cu $(v_i, v_{i+1}) \in E, \forall i$

Lungimea unui drum este:

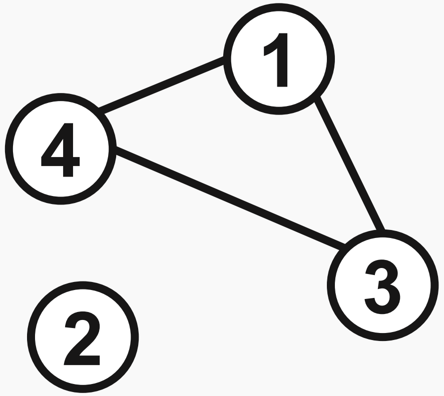
$$w(P) = \sum_{i=0}^{N-1} w(v_i, v_{i+1})$$

Dacă un nod se repetă se numește **walk** în loc de **path**

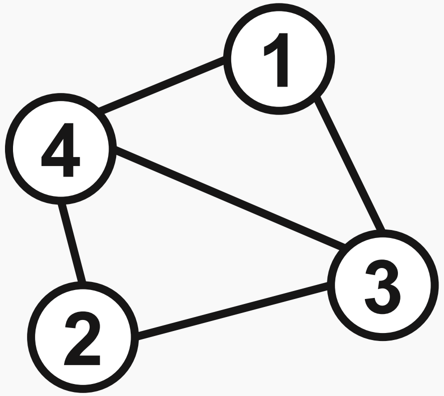
Graf conex

Un graf se numește conex dacă există o cale (path) de la oricare nod din graf la oricare altul.

$$\exists P(v_i, v_j), \forall v_i, v_j \in V$$



NU conex



conex

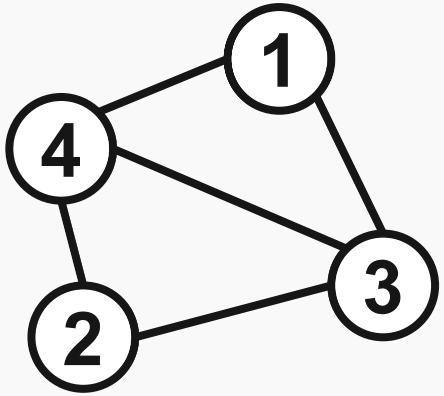
Graf complet

Un graf se numește complet dacă toate nodurile sunt conectate cu toate celelalte.

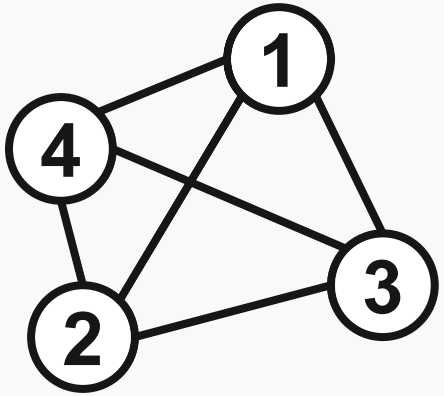
$$\forall v_i;\ grad(v_i) = N - 1$$

Unde: grad – numărul de muchii conectate la nod

N – numărul de noduri



NU complet



complet

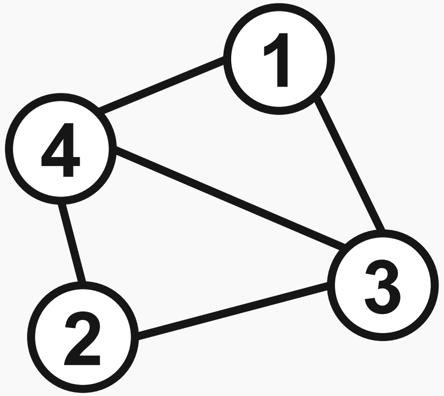
Reprezentarea grafurilor

Reprezentarea grafurilor – Matrice adiacență

- Matrice A de mărime NxN.
- N-numărul de noduri

$$A[i][j] = \begin{cases} 1\ dacă\ \exists(i,j) \in E \\ 0\ dacă\ \nexists(i,j) \in E \end{cases}$$

	[][1]	[][2]	[][3]	[][4]
[1][]	0	0	1	1
[2][]	0	0	1	1
[3][]	1	1	0	1
[4][]	1	1	1	0

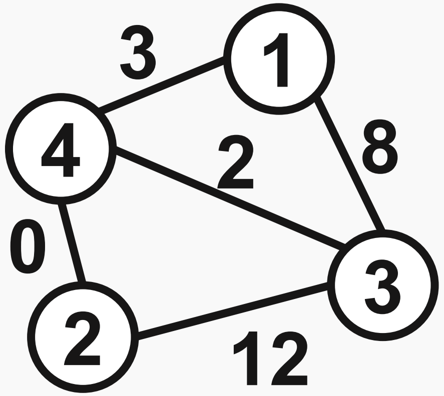


Reprezentarea grafurilor – Matrice costuri

- Matrice A de mărime NxN.
- N-numărul de noduri

$$A[i][j] = \begin{cases} w(i,j)\ dacă\ \exists(i,j) \in E \\ \infty\ dacă\ \nexists(i,j) \in E \end{cases}$$

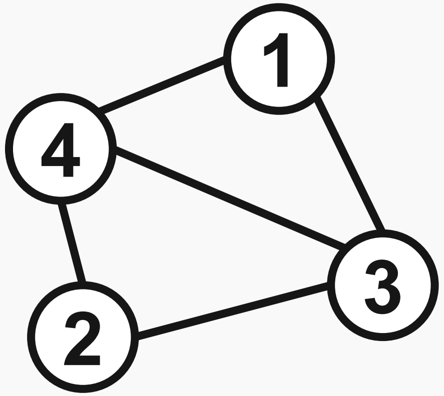
	[][1]	[][2]	[][3]	[][4]
[1][]	0	∞	8	3
[2][]	∞	0	12	0
[3][]	8	2	0	12
[4][]	3	0	3	0



Walk matrix

- $B = A * A$ unde A este matrice de adiacență
- B reprezintă câte plimbări (walks) de lungime **2** sunt între cele **2** noduri, reprezentând linia și coloana.

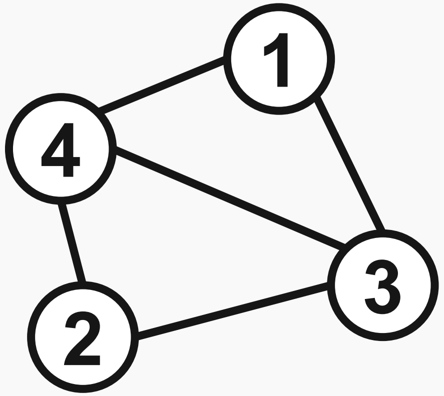
	[][1]	[][2]	[][3]	[][4]
[1][]	2	2	1	1
[2][]	2	2	1	1
[3][]	1	1	3	2
[4][]	1	1	2	3



Walk matrix

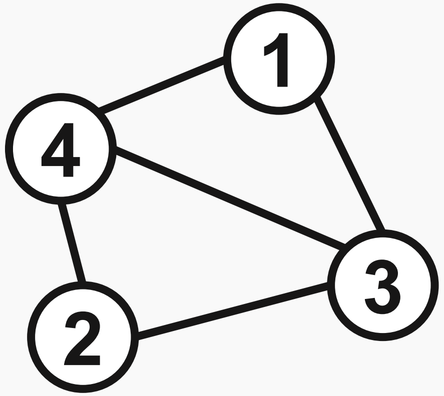
- $B = A * A * A = A^3$
- B reprezintă câte plimbări (walks) de lungime **3** sunt între cele **2** noduri, reprezentând linia și coloana.

	[][1]	[][2]	[][3]	[][4]
[1][]	2	2	5	5
[2][]	2	2	5	5
[3][]	5	5	4	5
[4][]	5	5	5	4



Walk matrix

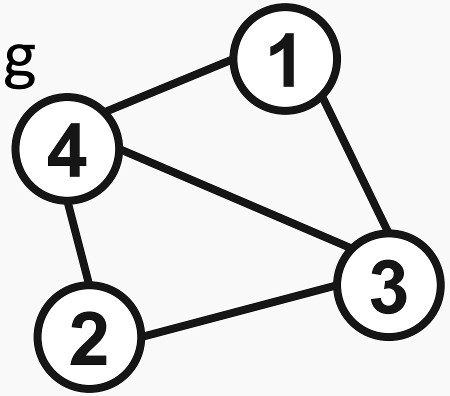
- $B = A^k$
- B reprezintă câte plimbări (walks) de lungime **k** sunt între cele **2** noduri, reprezentând linia și coloana.
- Dacă avem 0 pe o poziție înseamnă că nu se poate ajunge la acel nod în **k** pași.
- Pentru graf neorientat:
  - Dacă k=N putem determina dacă graful e conex
- Calcul A^N are complexitate $O(N^4)$



Reprezentarea grafurilor – Listă vecini (adiacență)

Se rețin nodurile, și pentru fiecare lista sa de vecini (pot fi direct pointeri, sau int-uri).

```
typedef struct vertex {
    int name;
    struct vertex ** neighbors;
    int numNeighbors; //grad outgoing
    int* weights;
}vertex;

vertex vertexes[4];
```



Parcurgeri

Parcurgere adâncime – depth first

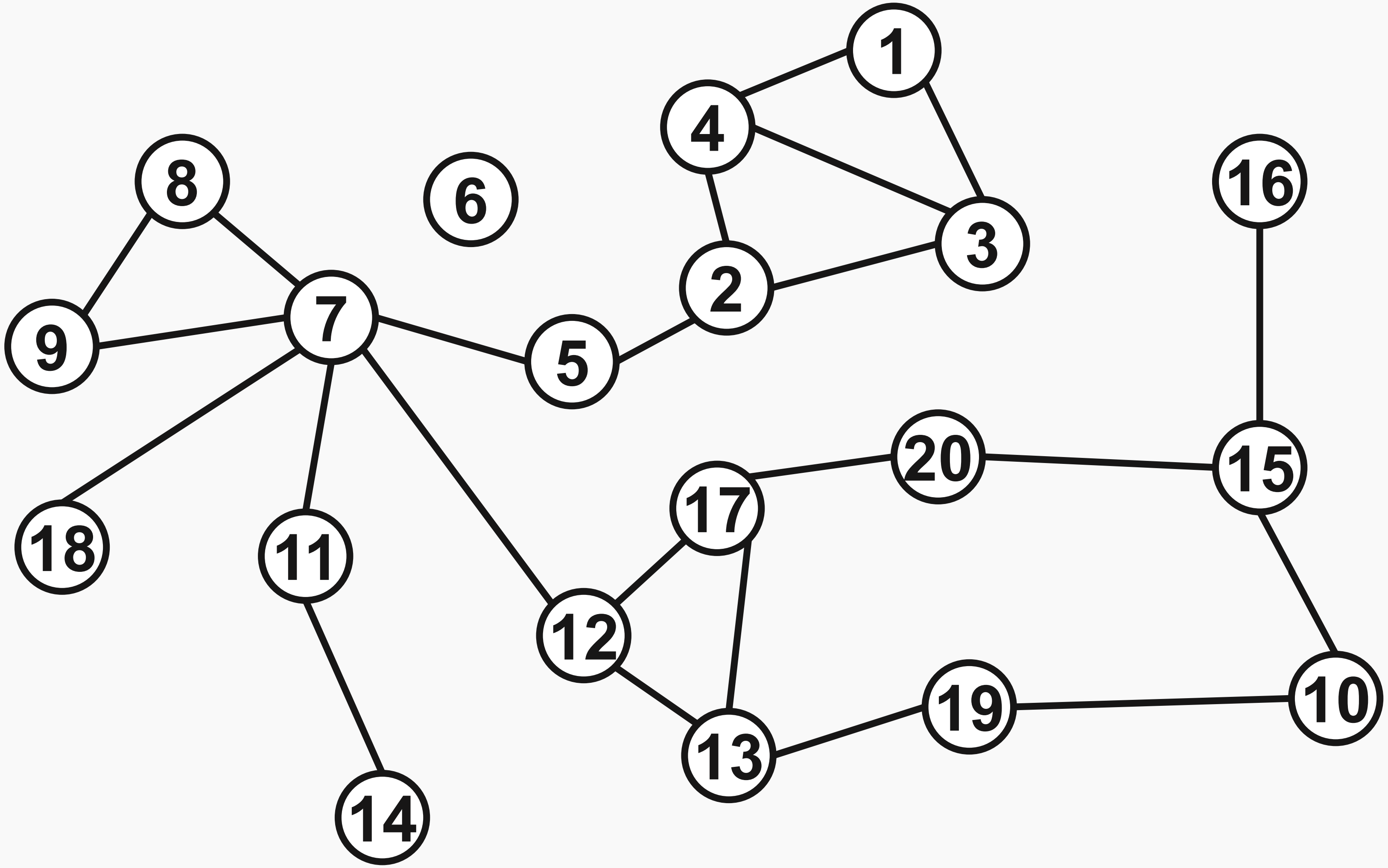
Charles Pierre Trémaux (1859–1882)

```
int visited[N] = { 0 };
void DFS(vertex currentNode) {
    visited[currentNode.name] = 1;
    for (int i = 0; i < currentNode.numNeighbors; i++) {
        if (!visited[currentNode.neighbors[i]->name])
            DFS(*(currentNode.neighbors[i]));
    }
}
```

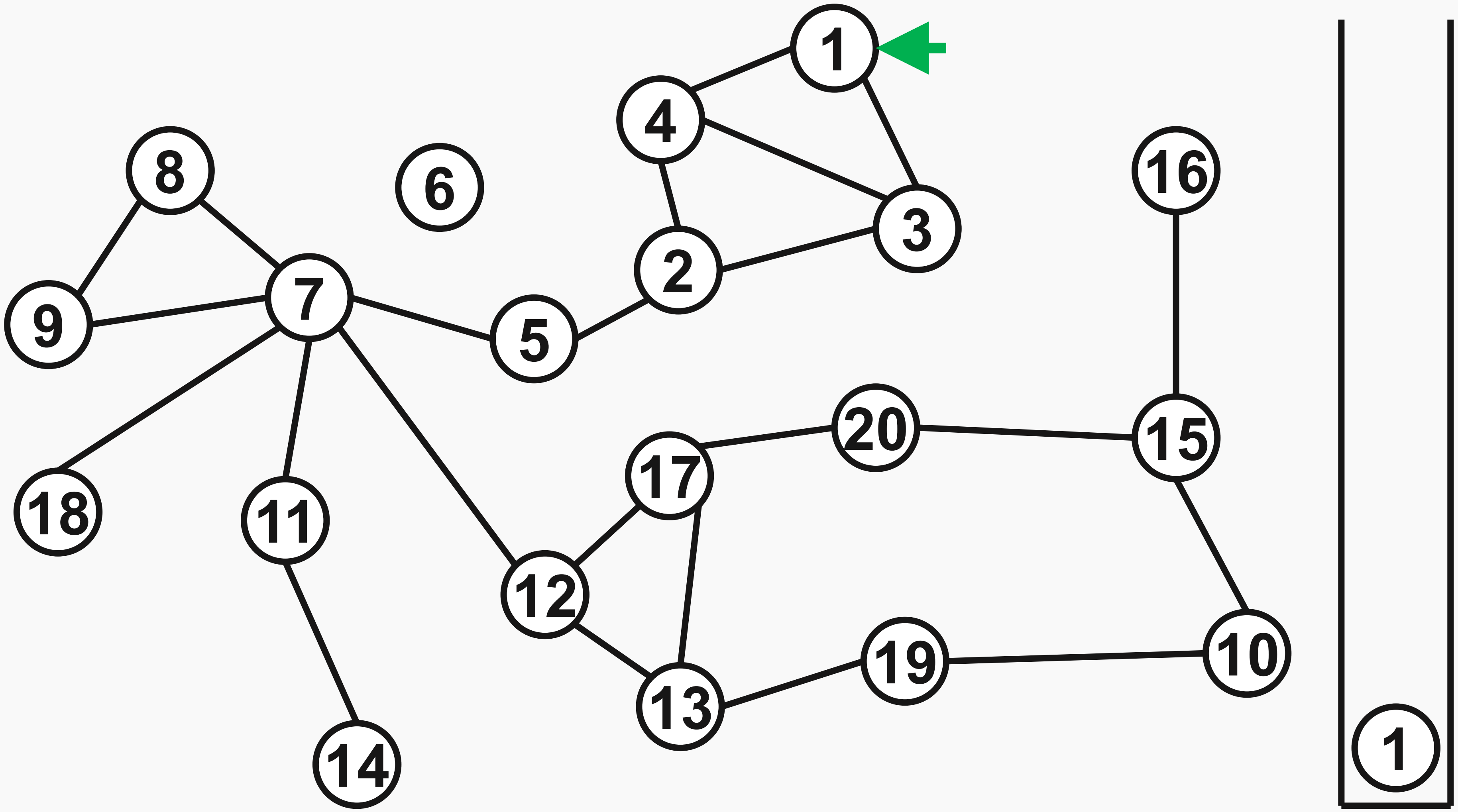
Parcurgere adâncime – depth first

```c
int visited[N] = { 0 };
void DFS_sequential(vertex startNode) {
    push(stack, startNode);
    while (!isEmpty(stack)) {
        currentNode = pop(stack);
        visited[currentNode.name] = 1;
        for (int i = 0; i < currentNode.numNeighbors; i++) {
            if (!visited[currentNode.neighbors[i]->name])
                push(stack, *(currentNode.neighbors[i]));
        }
    }
}
```

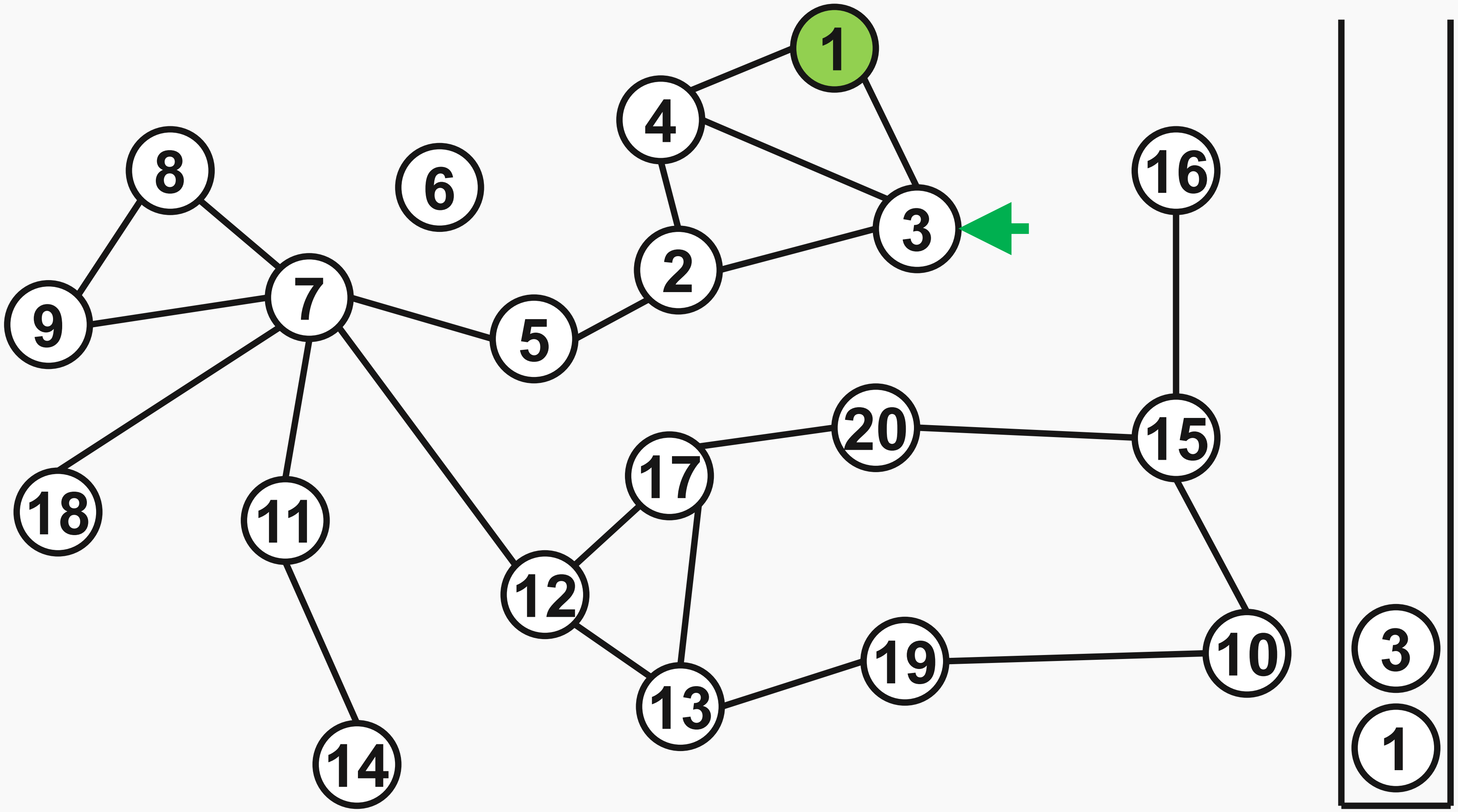
Parcurgere adâncime – depth first



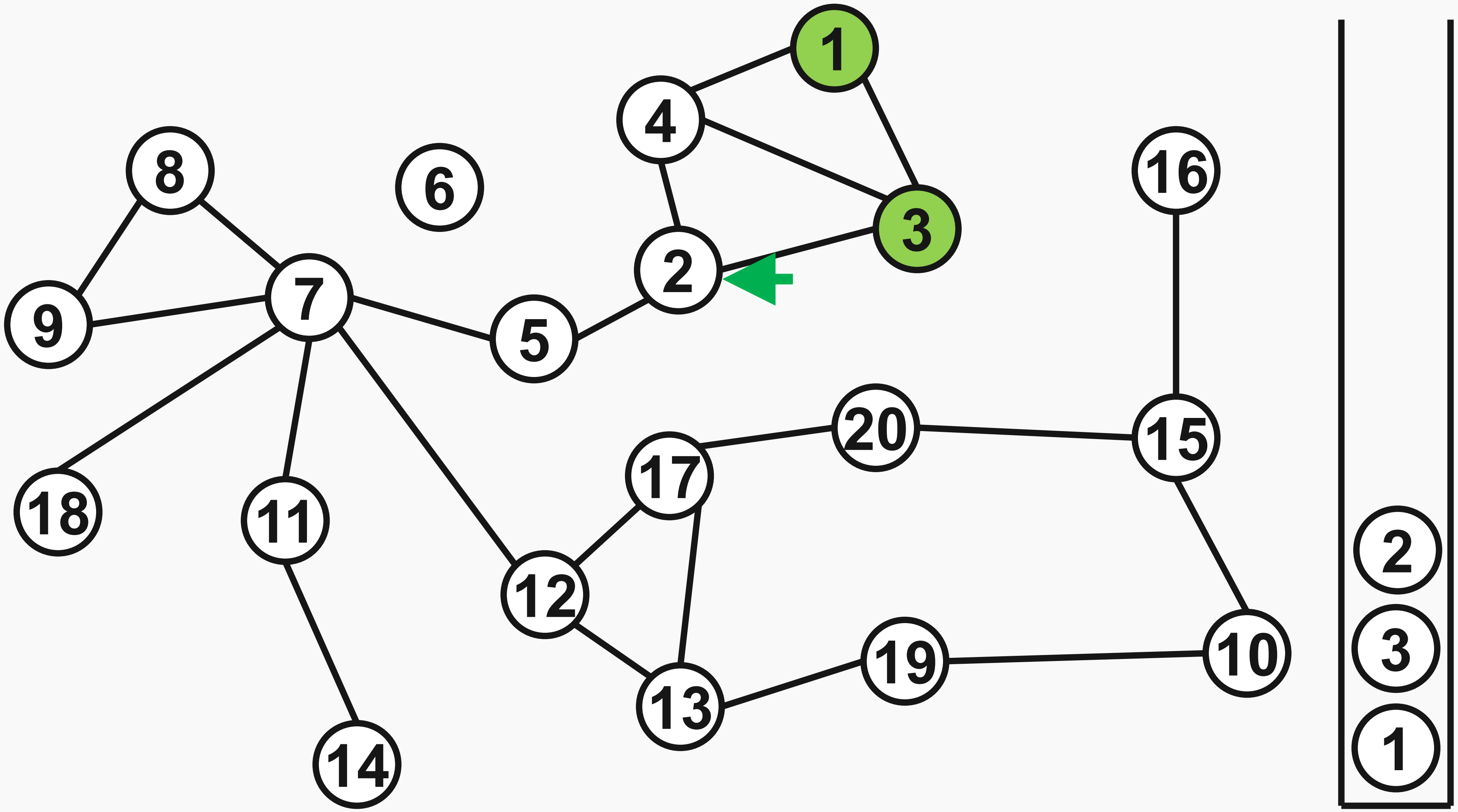
Parcurgere adâncime – depth first



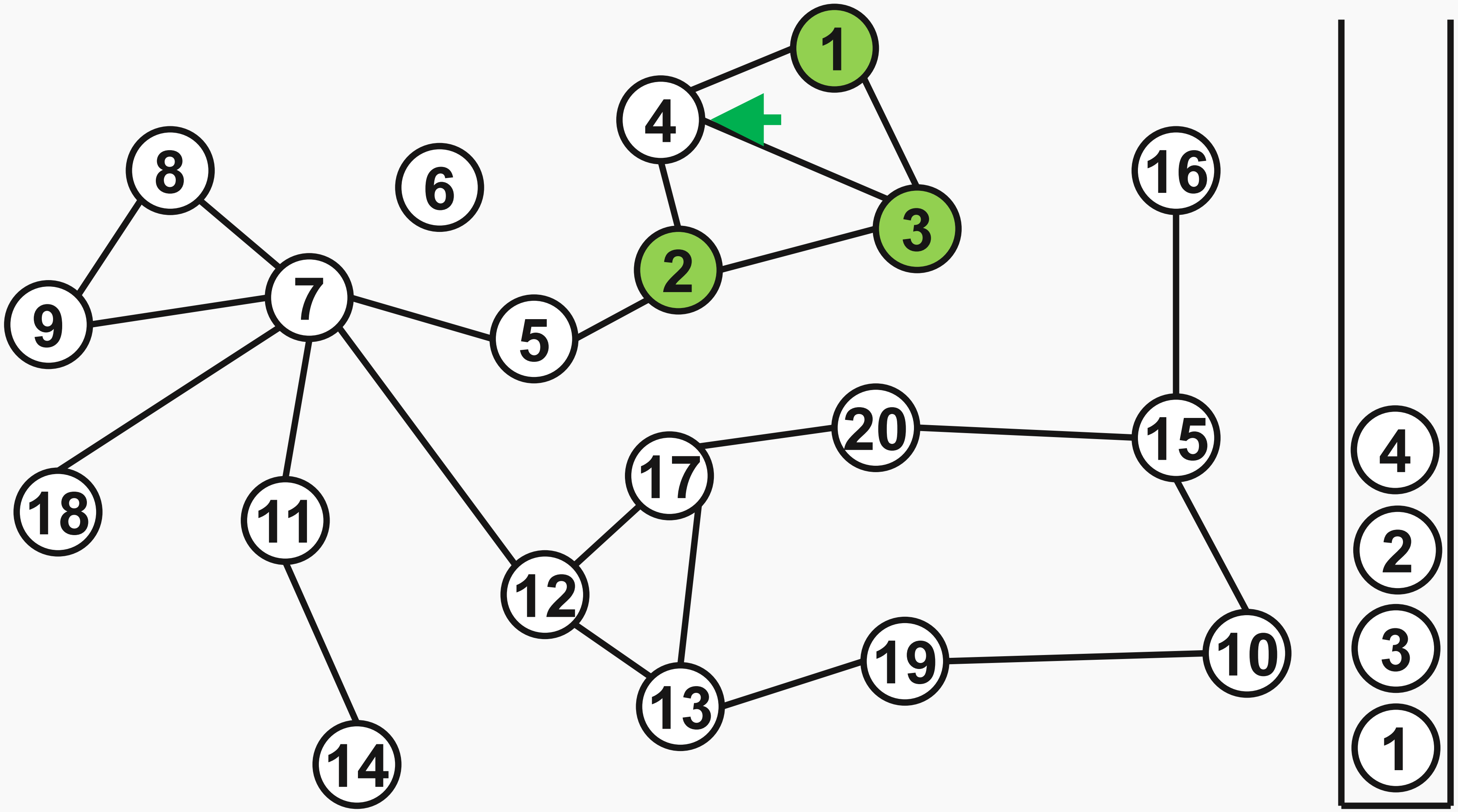
Parcurgere adâncime – depth first



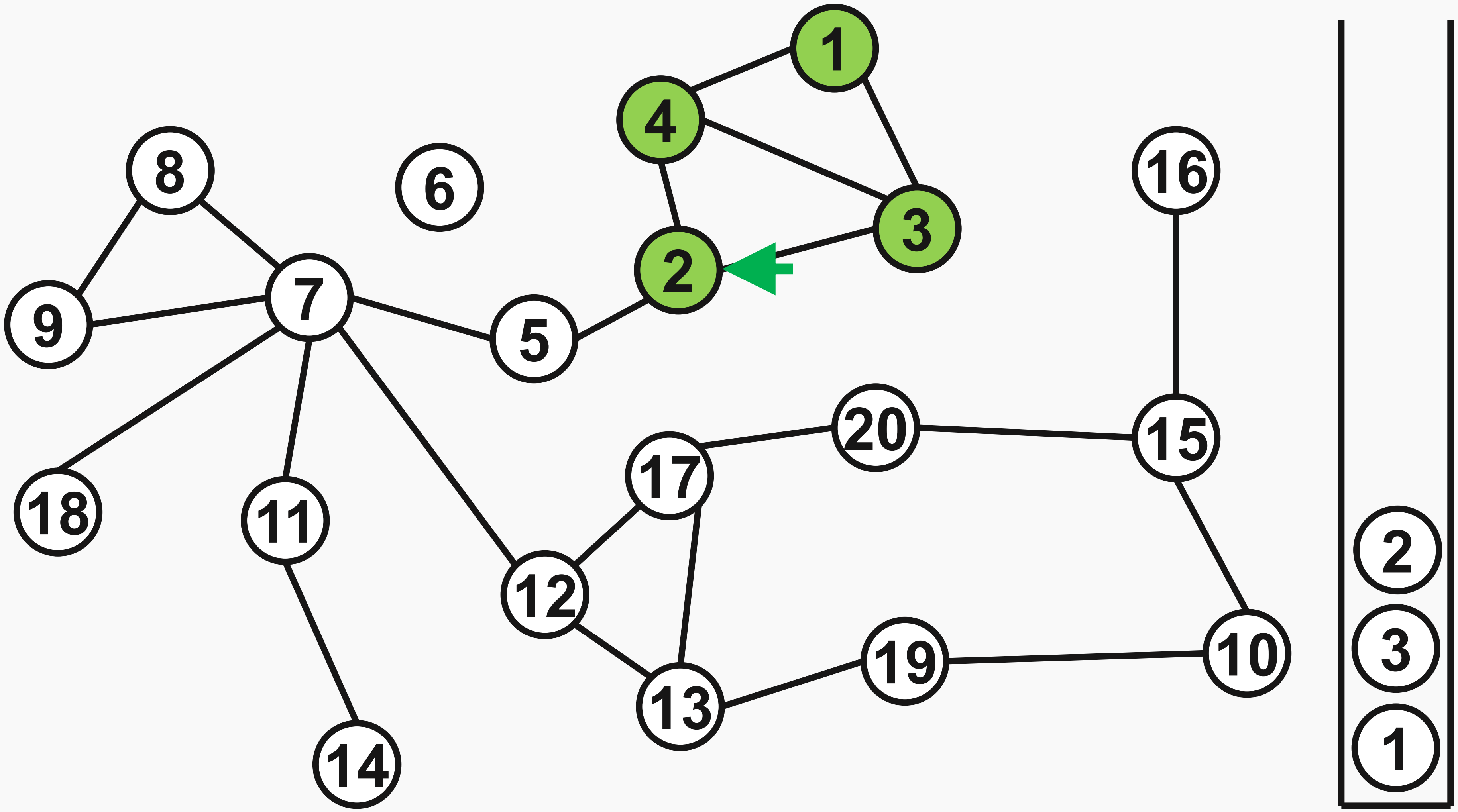
Parcurgere adâncime – depth first



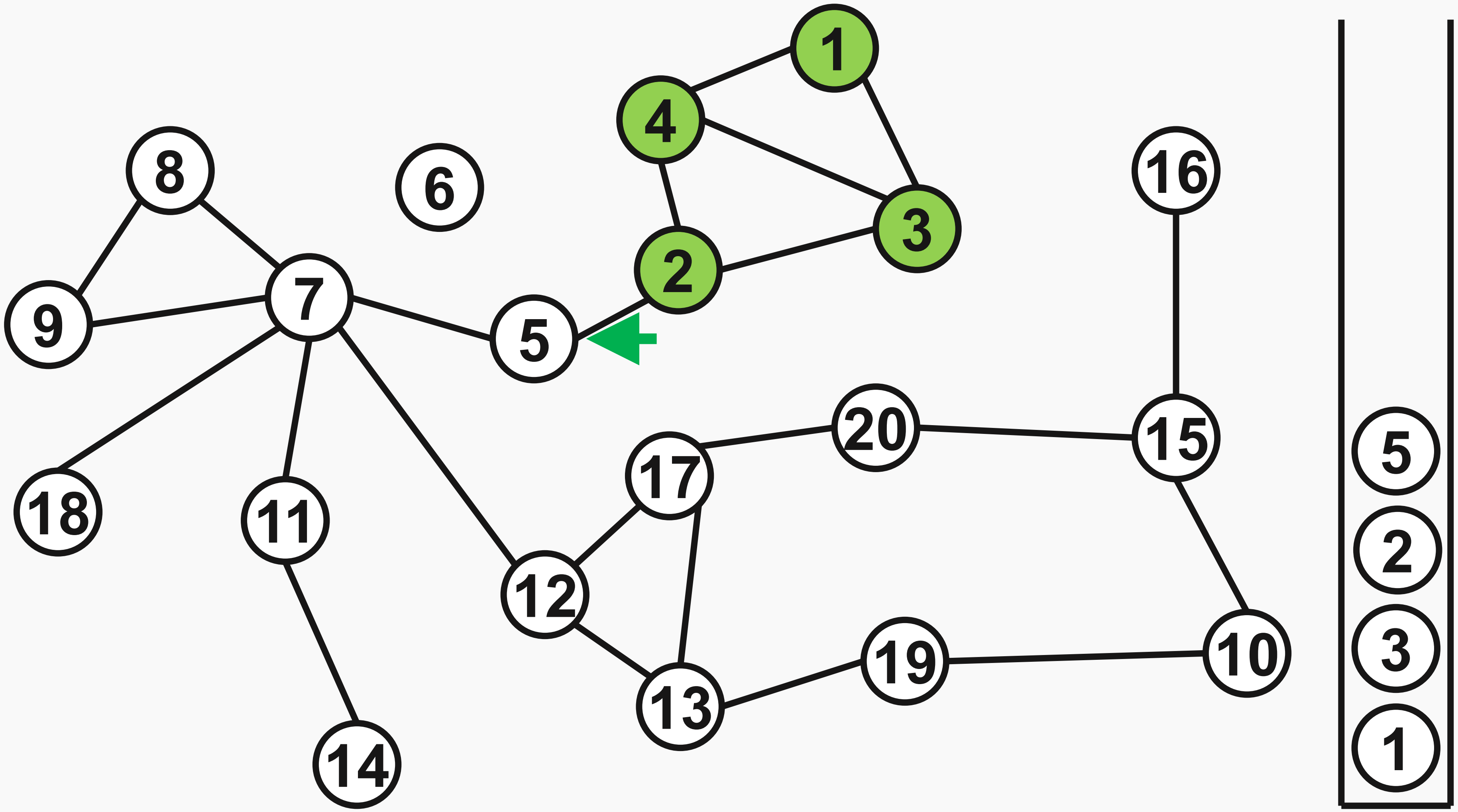
Parcurgere adâncime – depth first



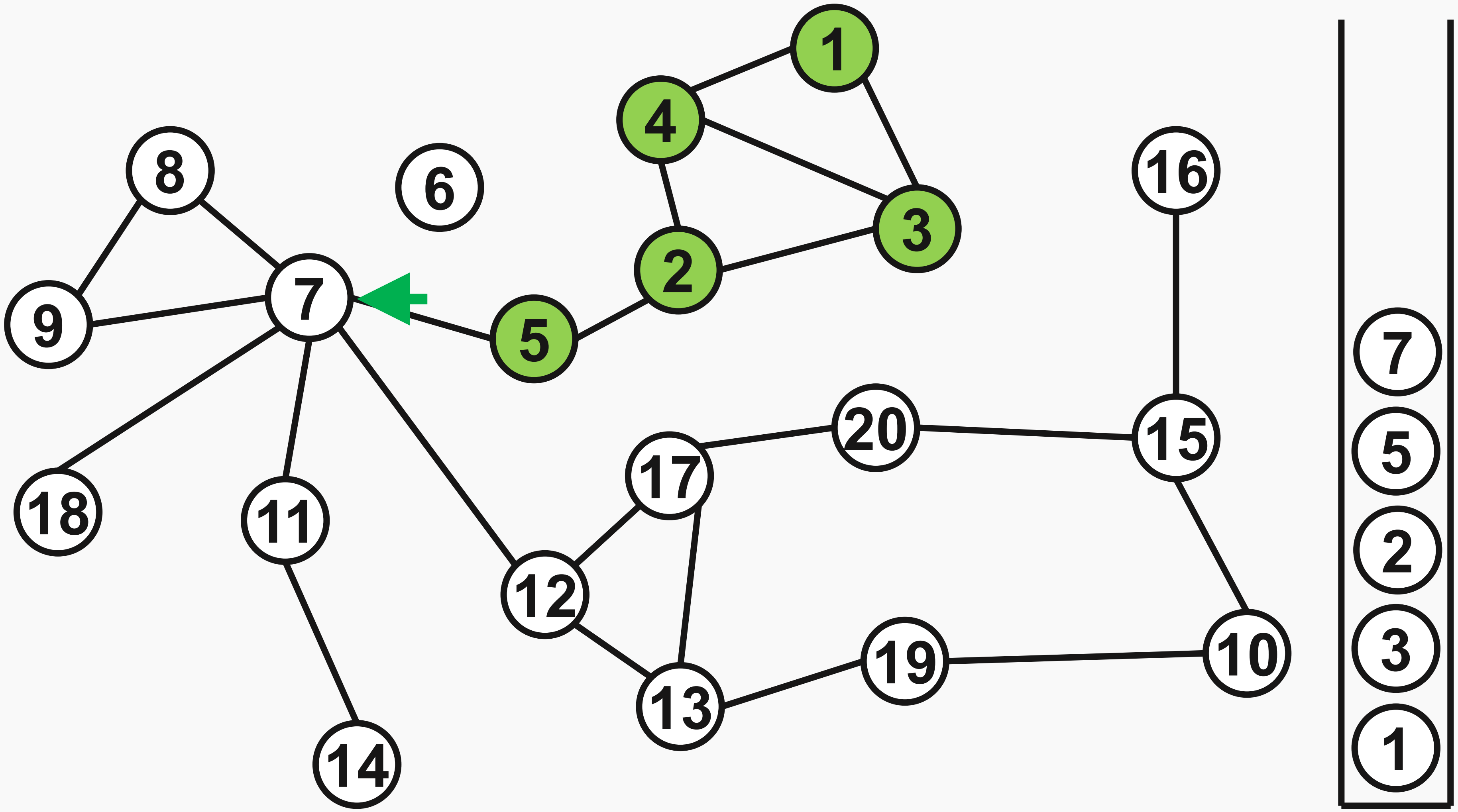
Parcurgere adâncime – depth first



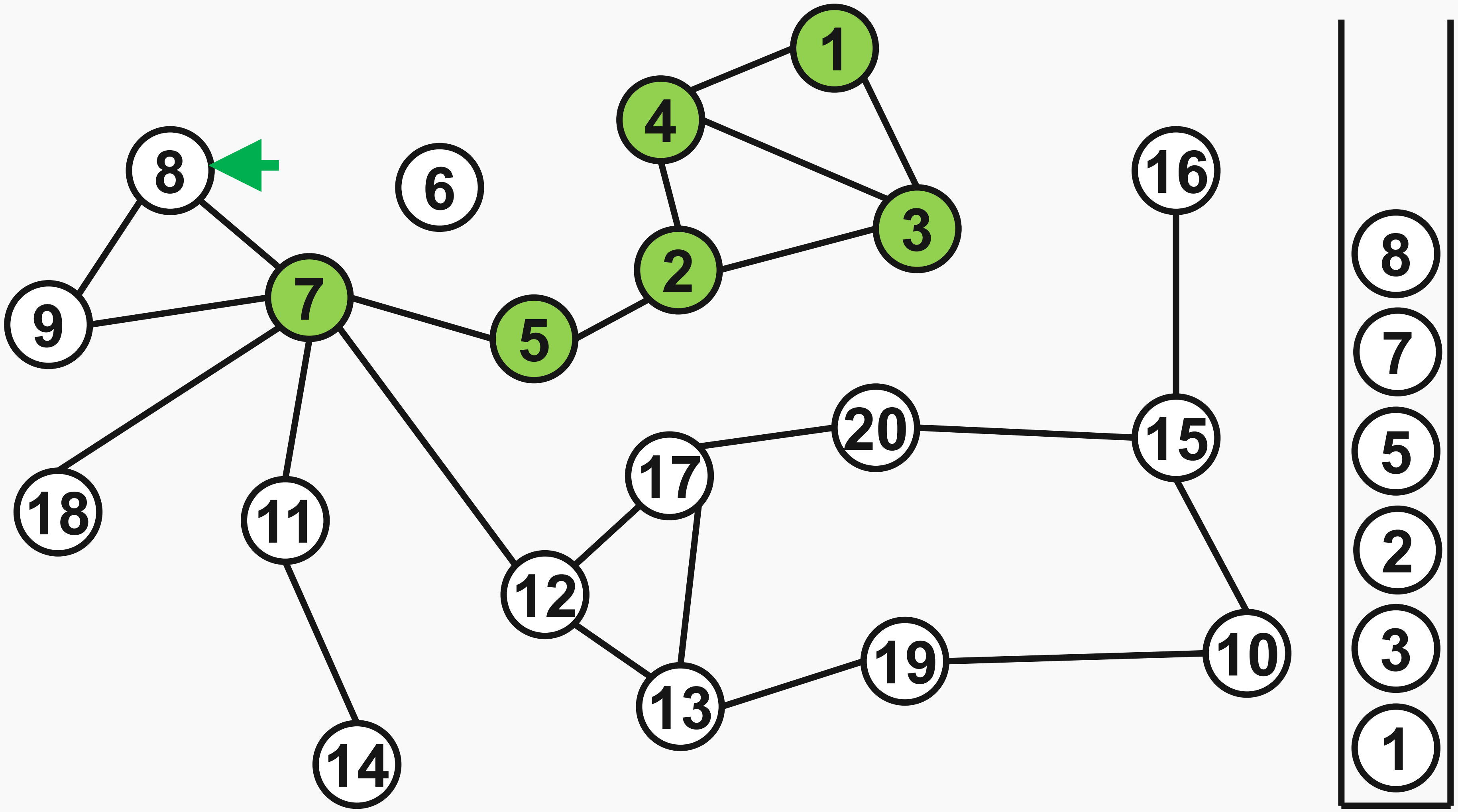
Parcurgere adâncime – depth first



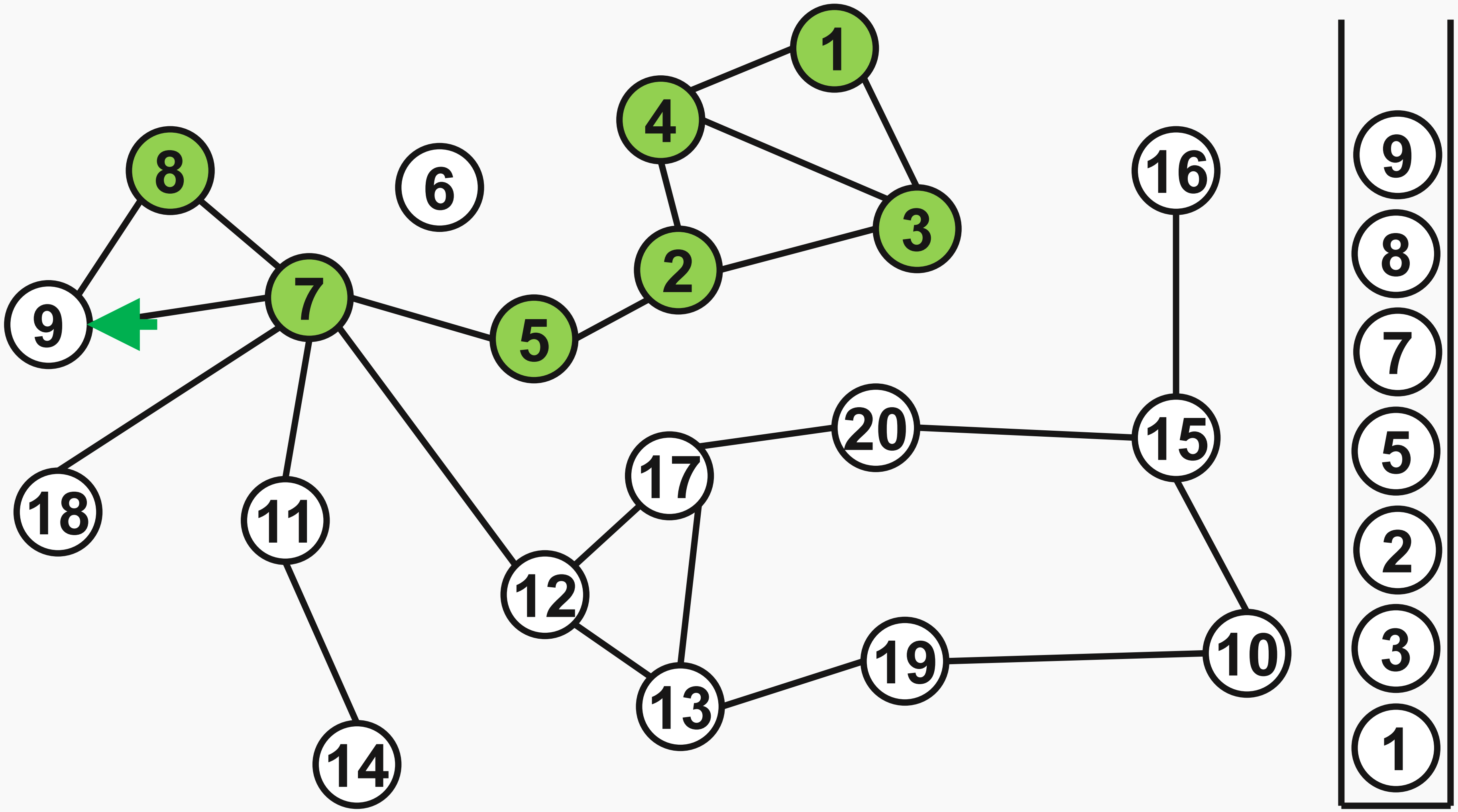
Parcurgere adâncime – depth first



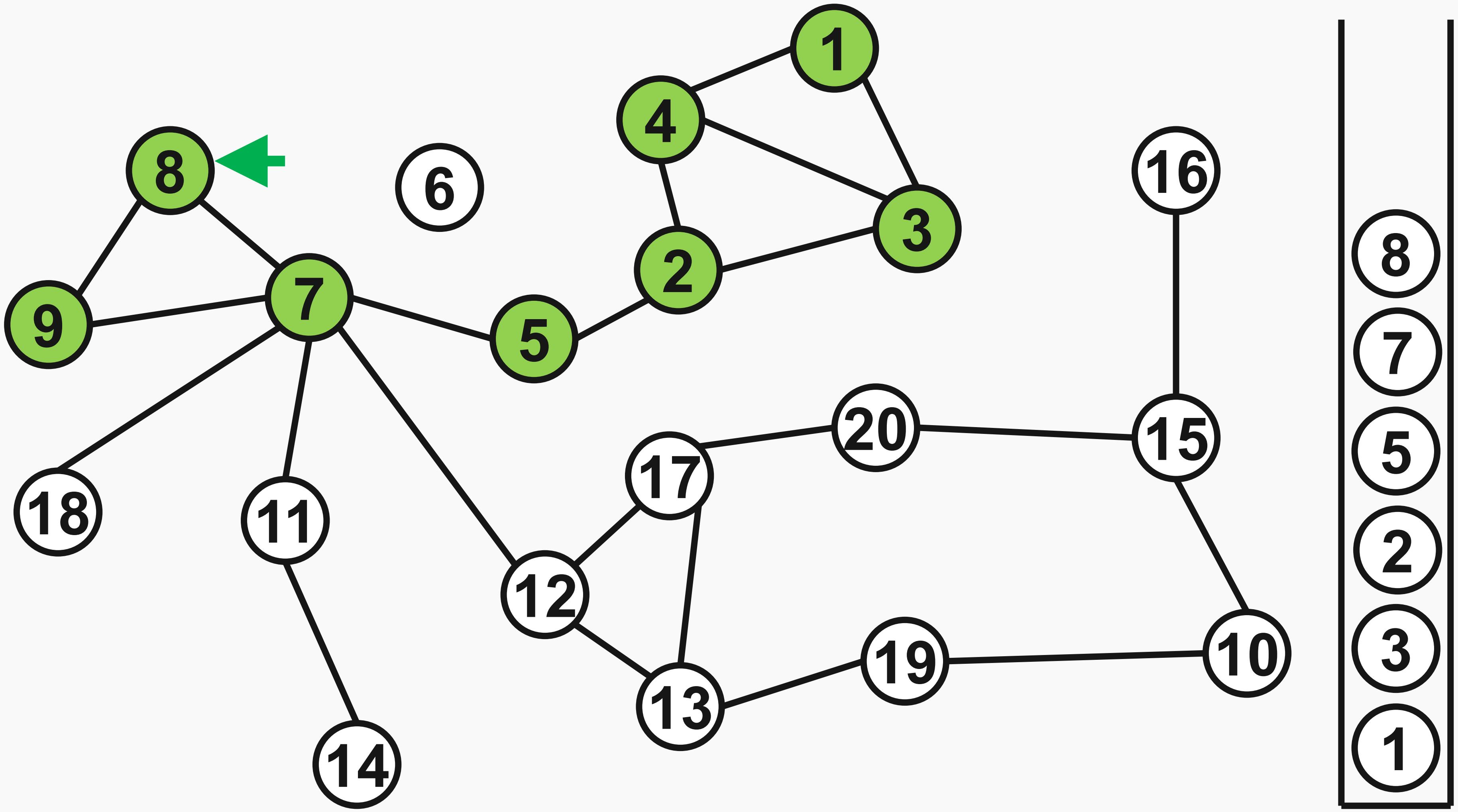
Parcurgere adâncime – depth first



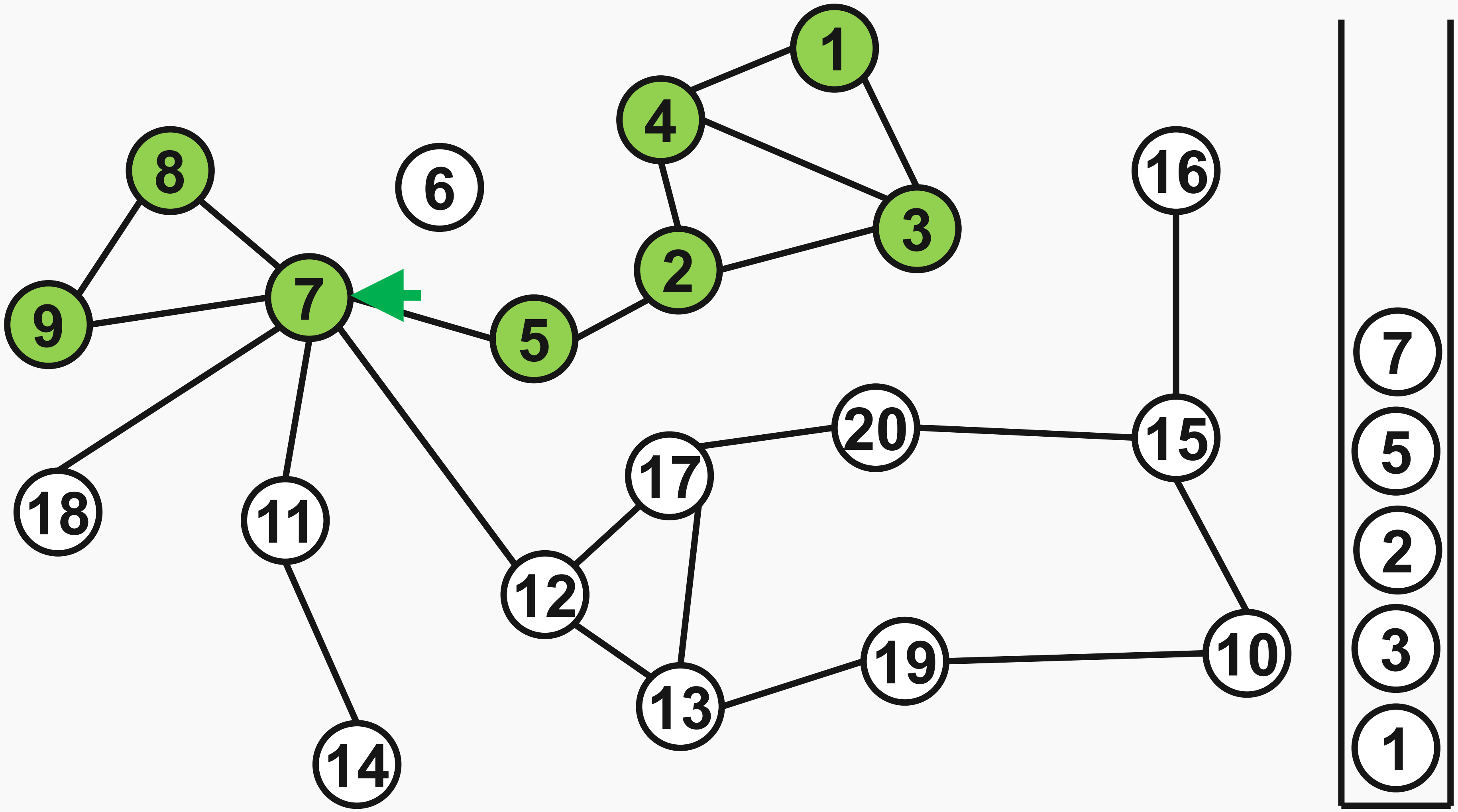
Parcurgere adâncime – depth first



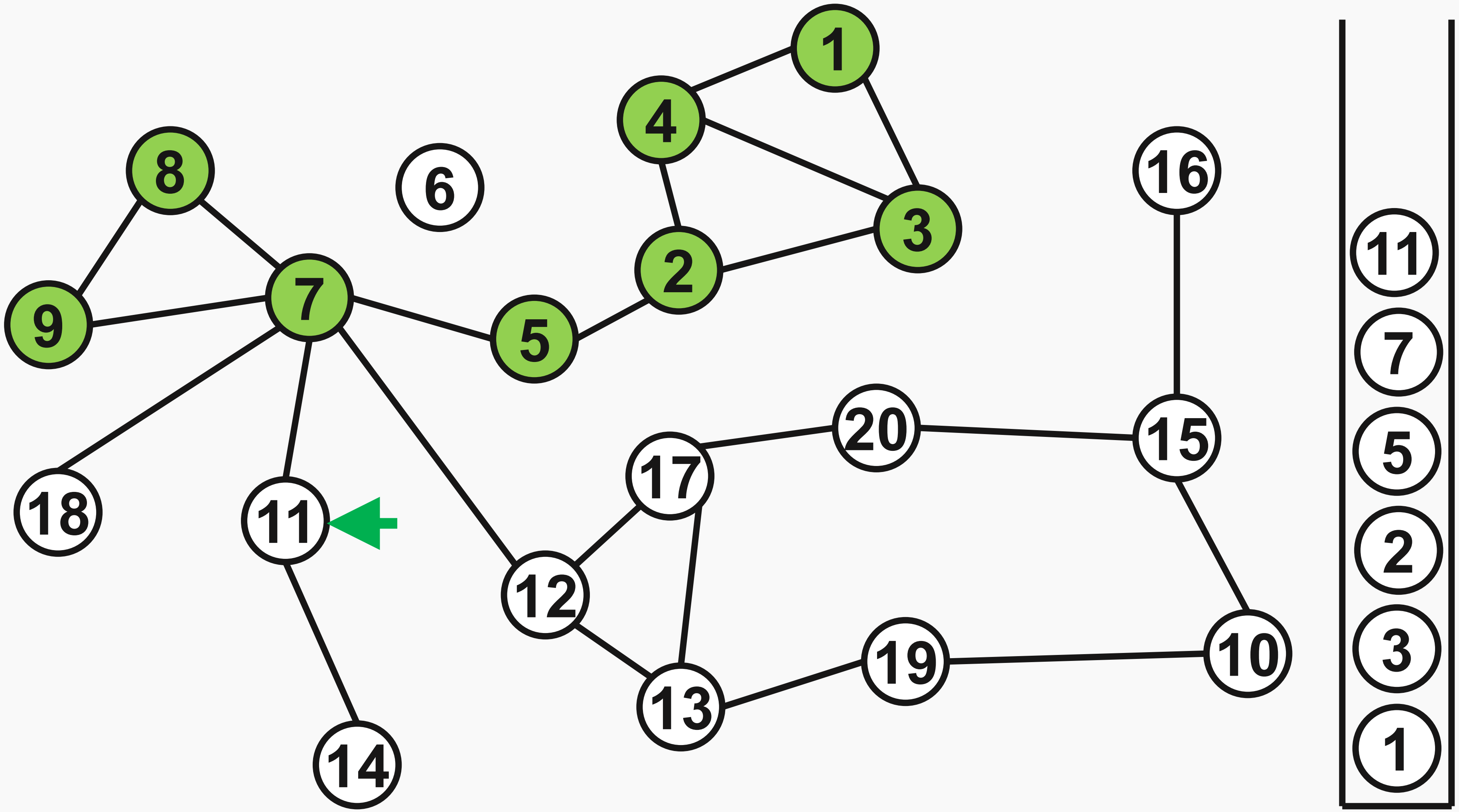
Parcurgere adâncime – depth first



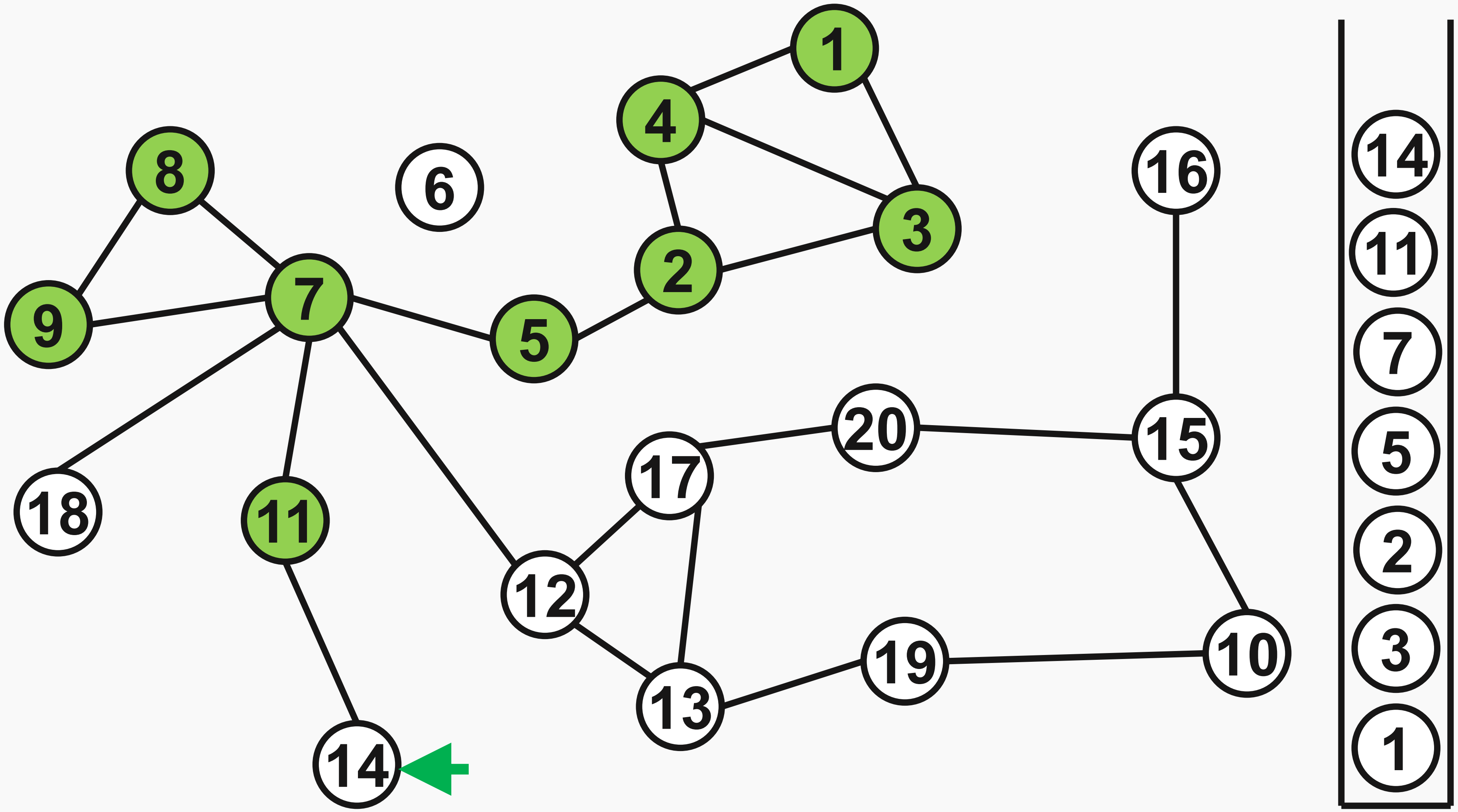
Parcurgere adâncime – depth first



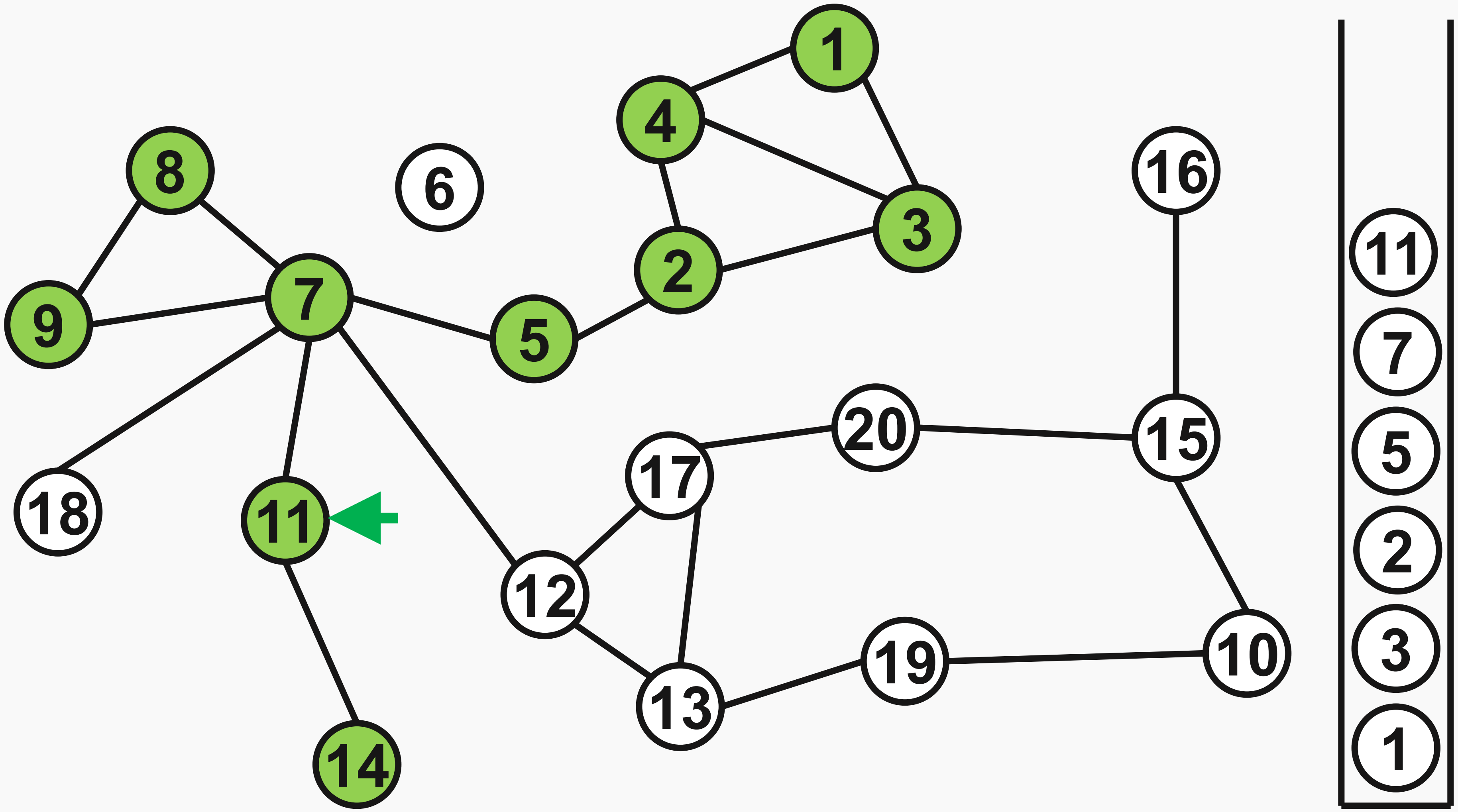
Parcurgere adâncime – depth first



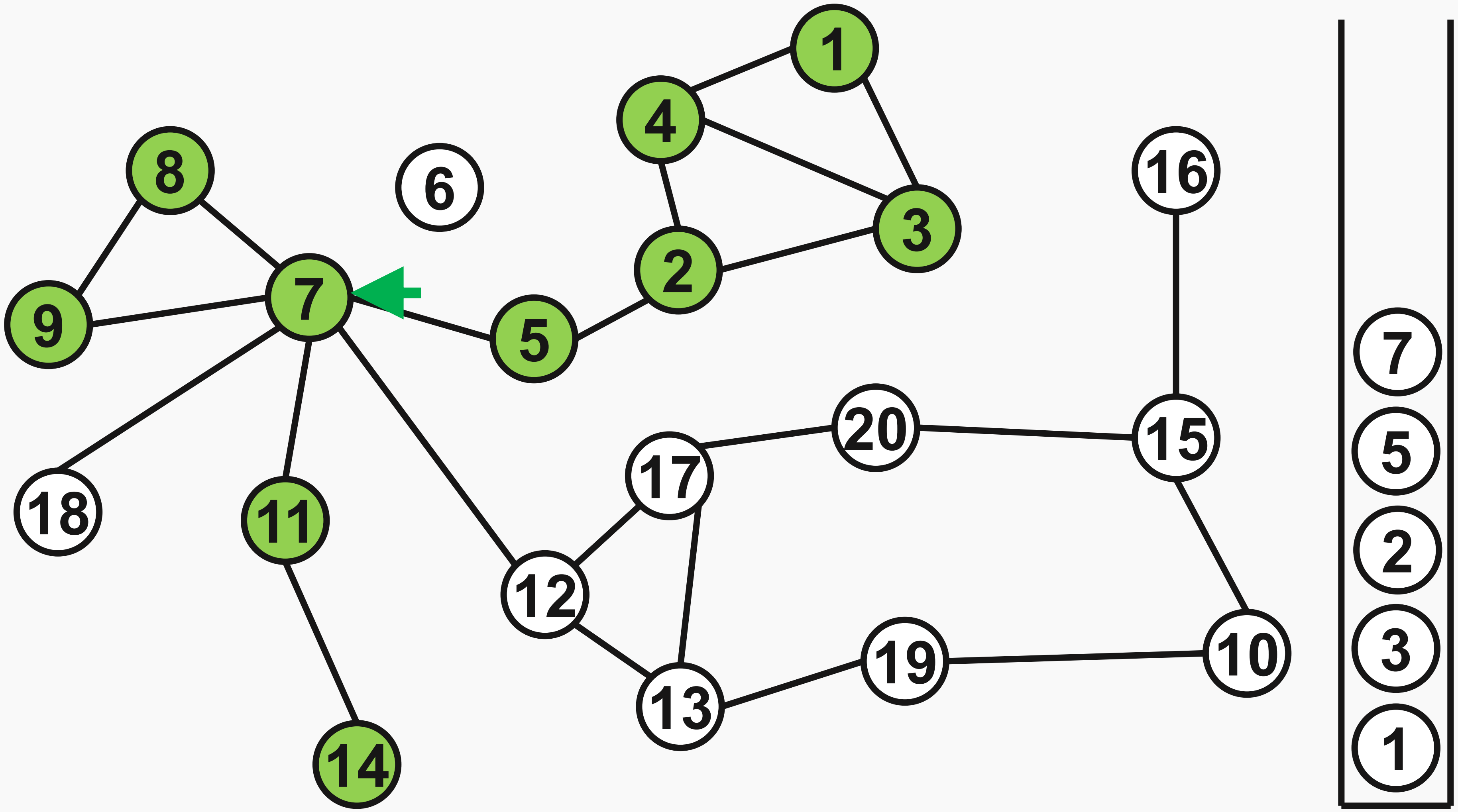
Parcurgere adâncime – depth first



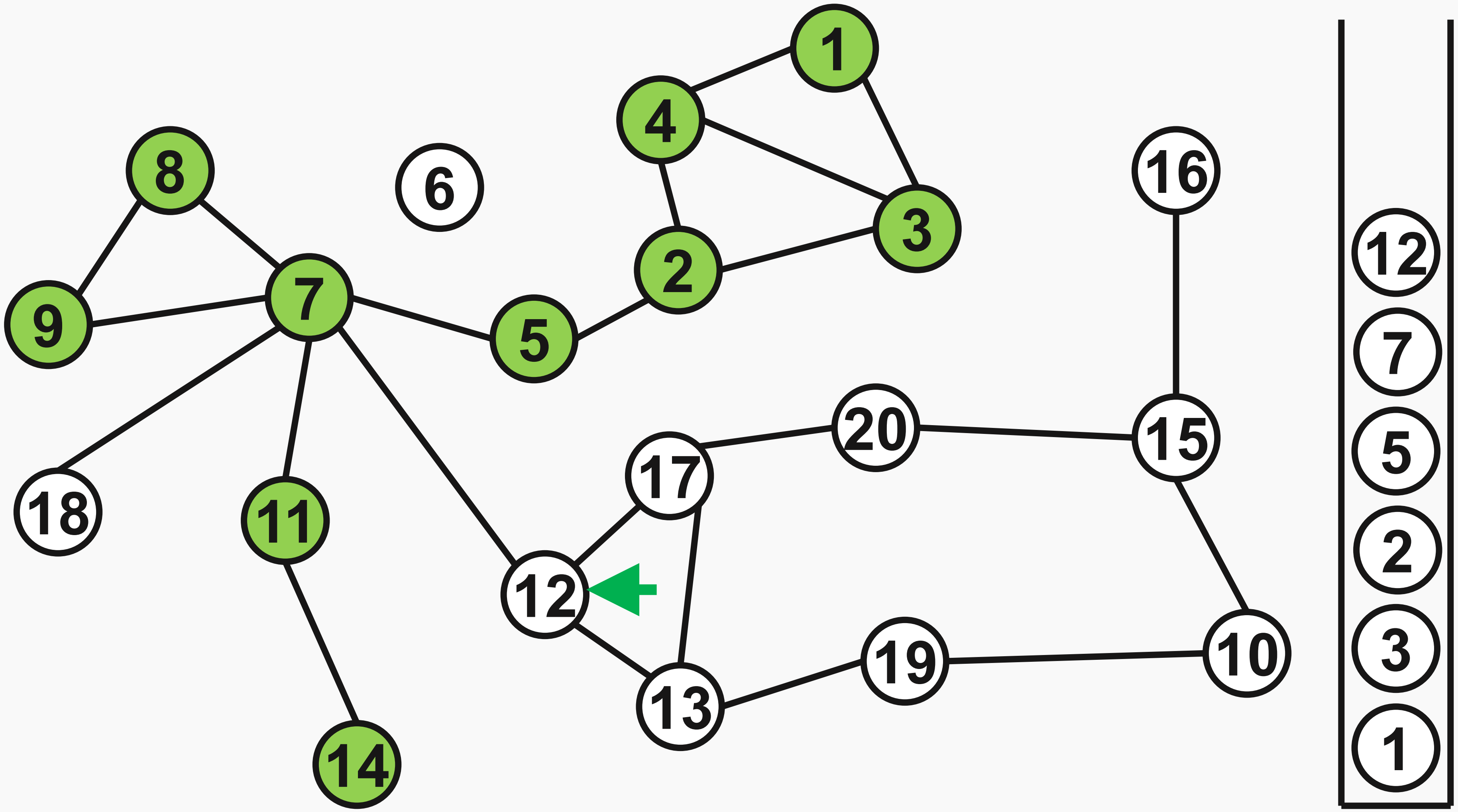
Parcurgere adâncime – depth first



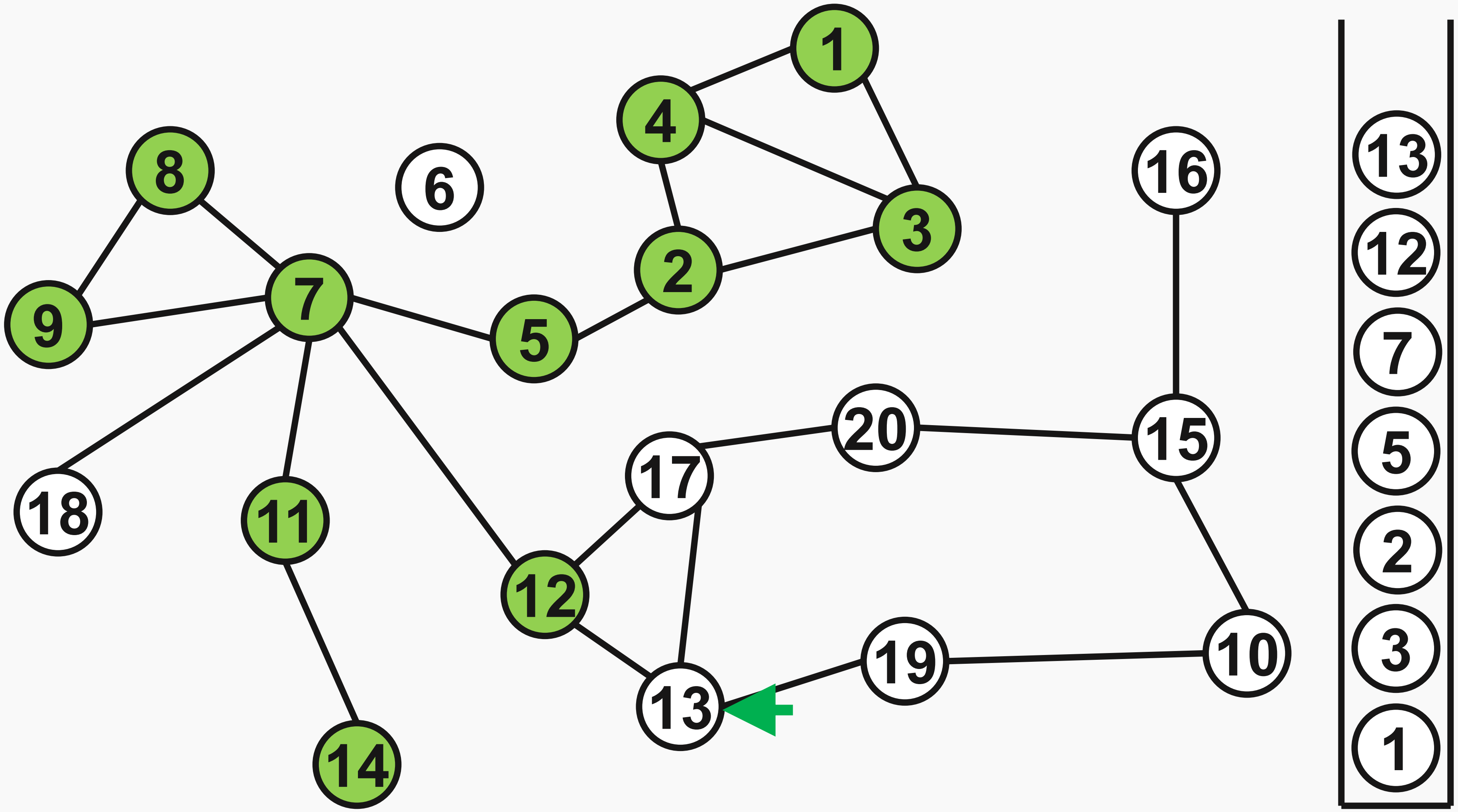
Parcurgere adâncime – depth first



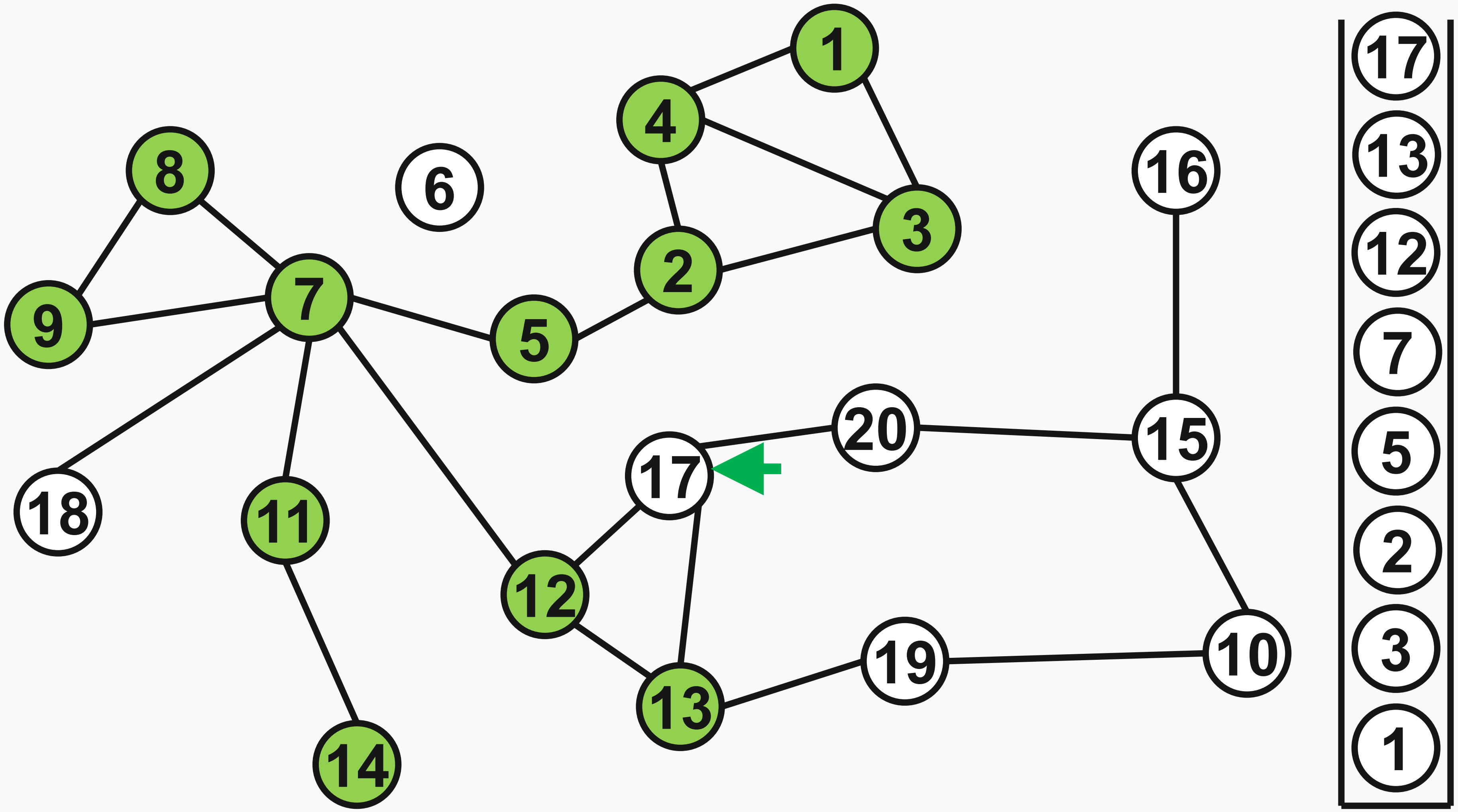
Parcurgere adâncime – depth first



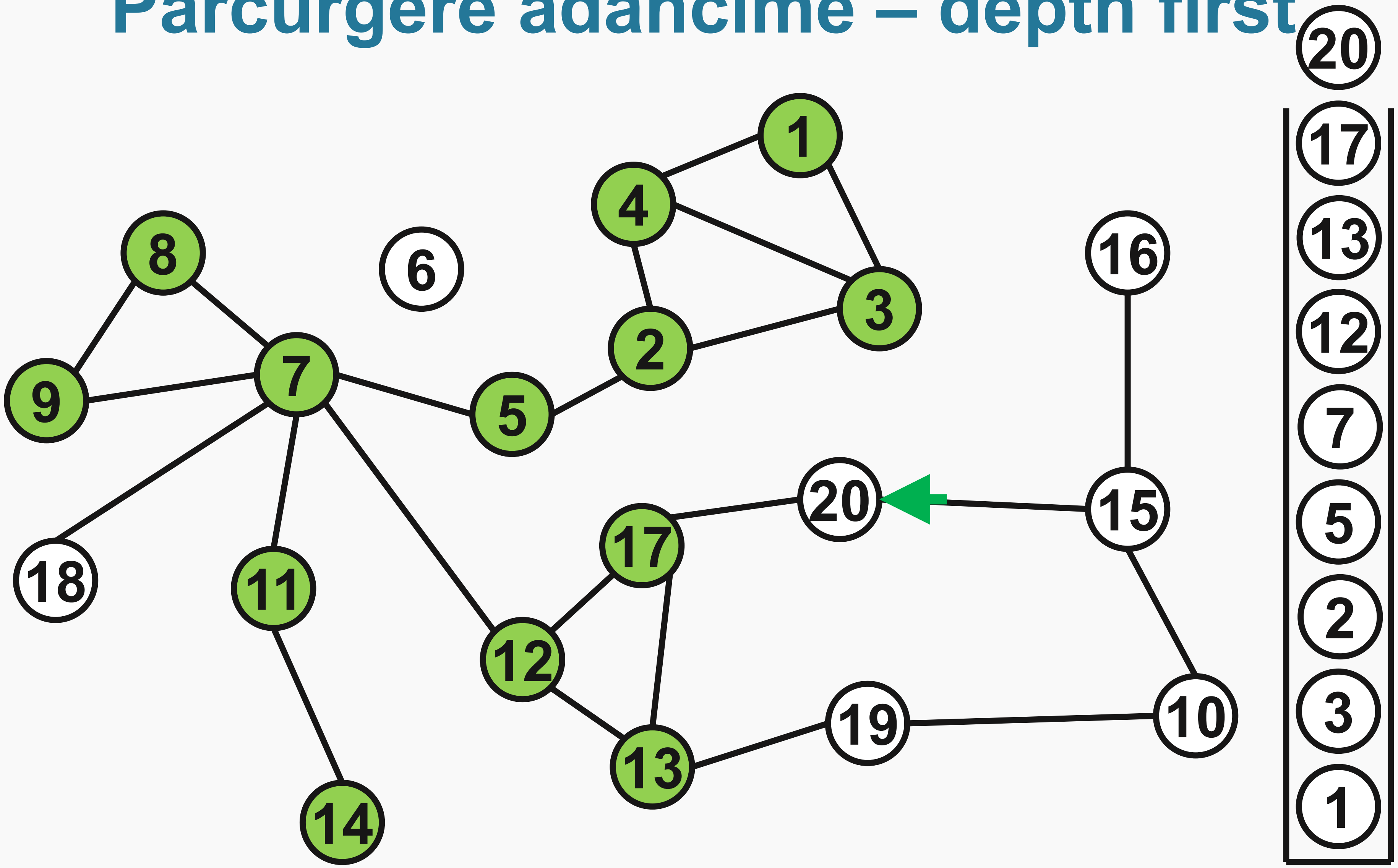
Parcurgere adâncime – depth first



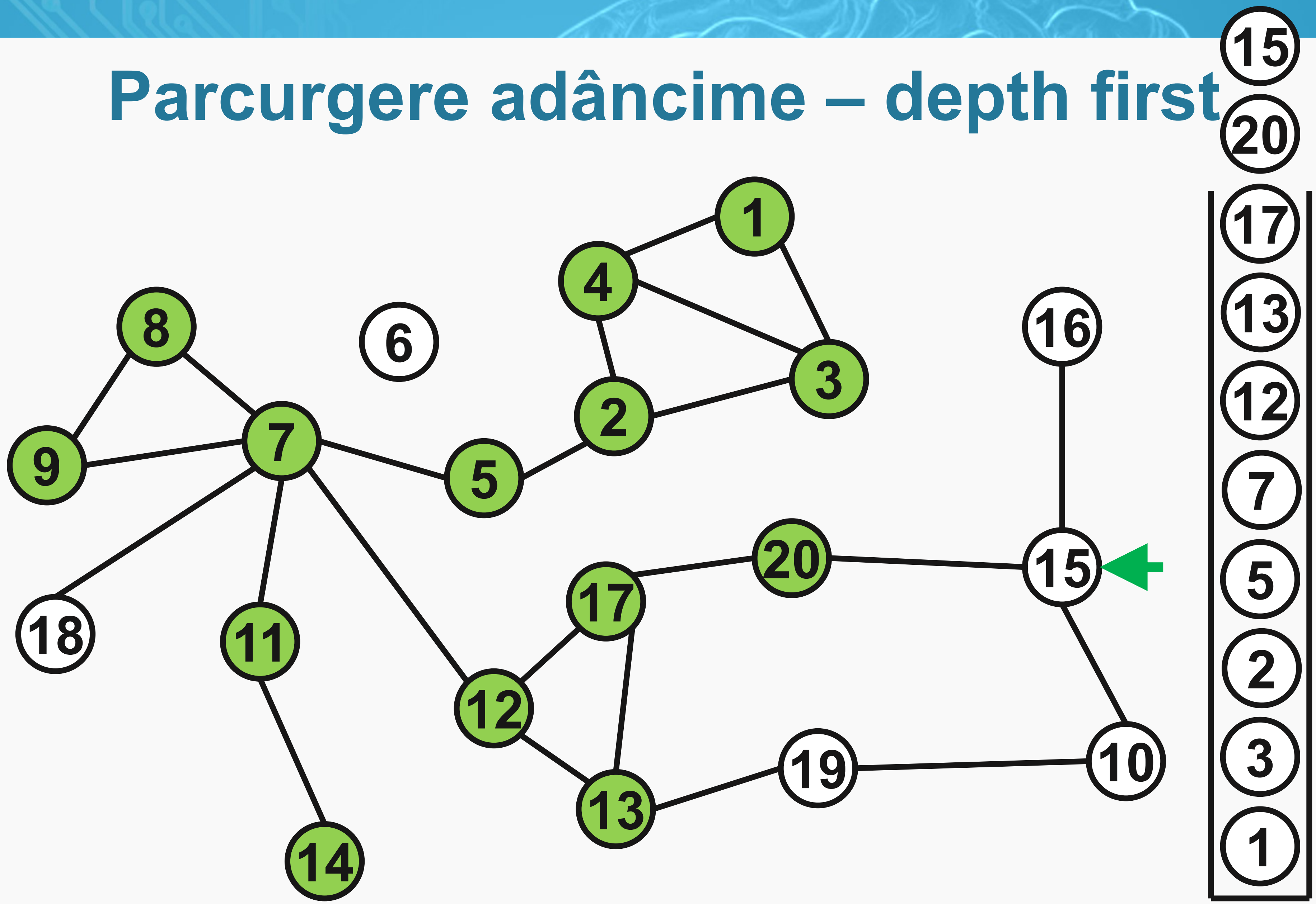
Parcurgere adâncime – depth first



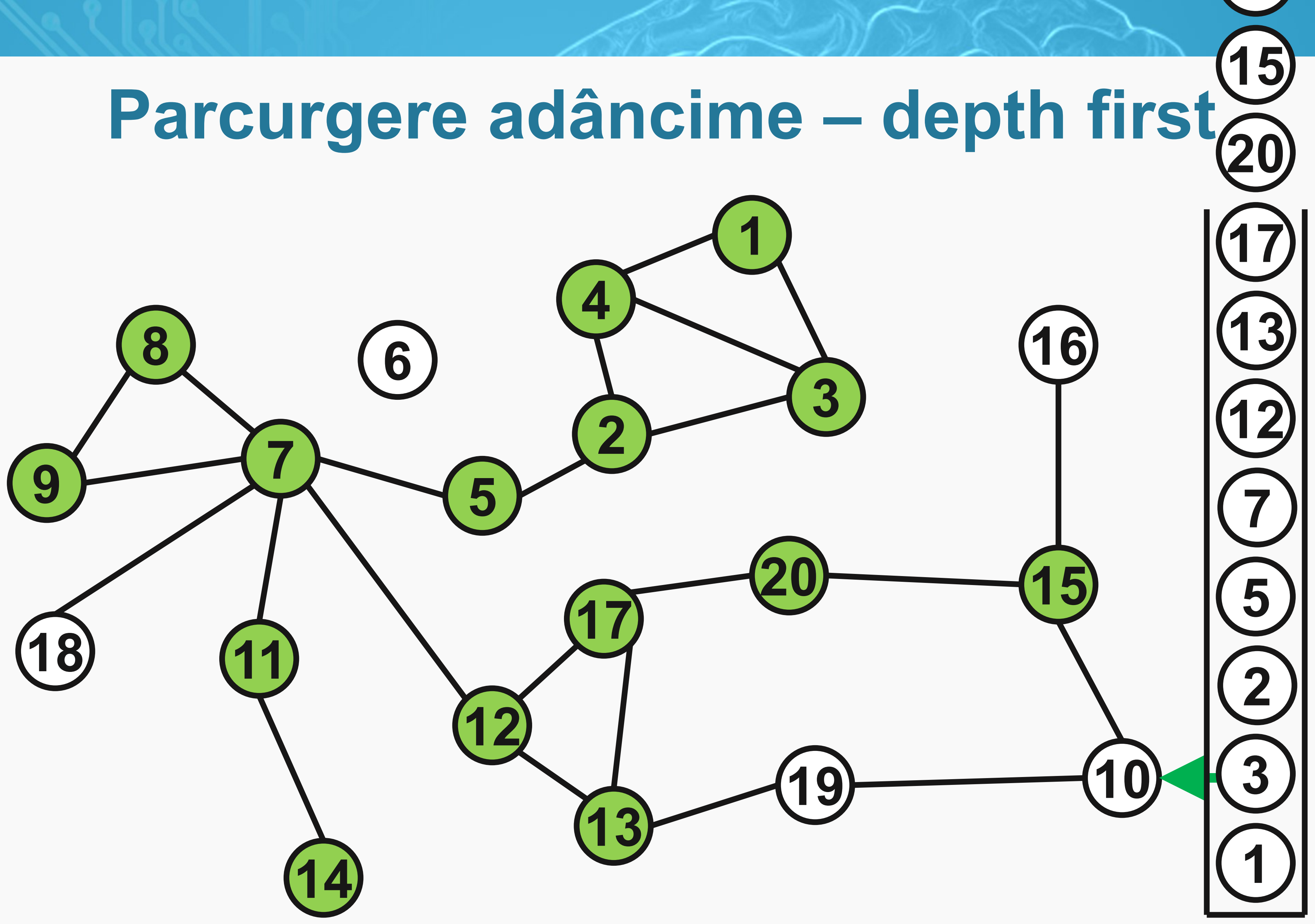
Parcurgere adâncime – depth first



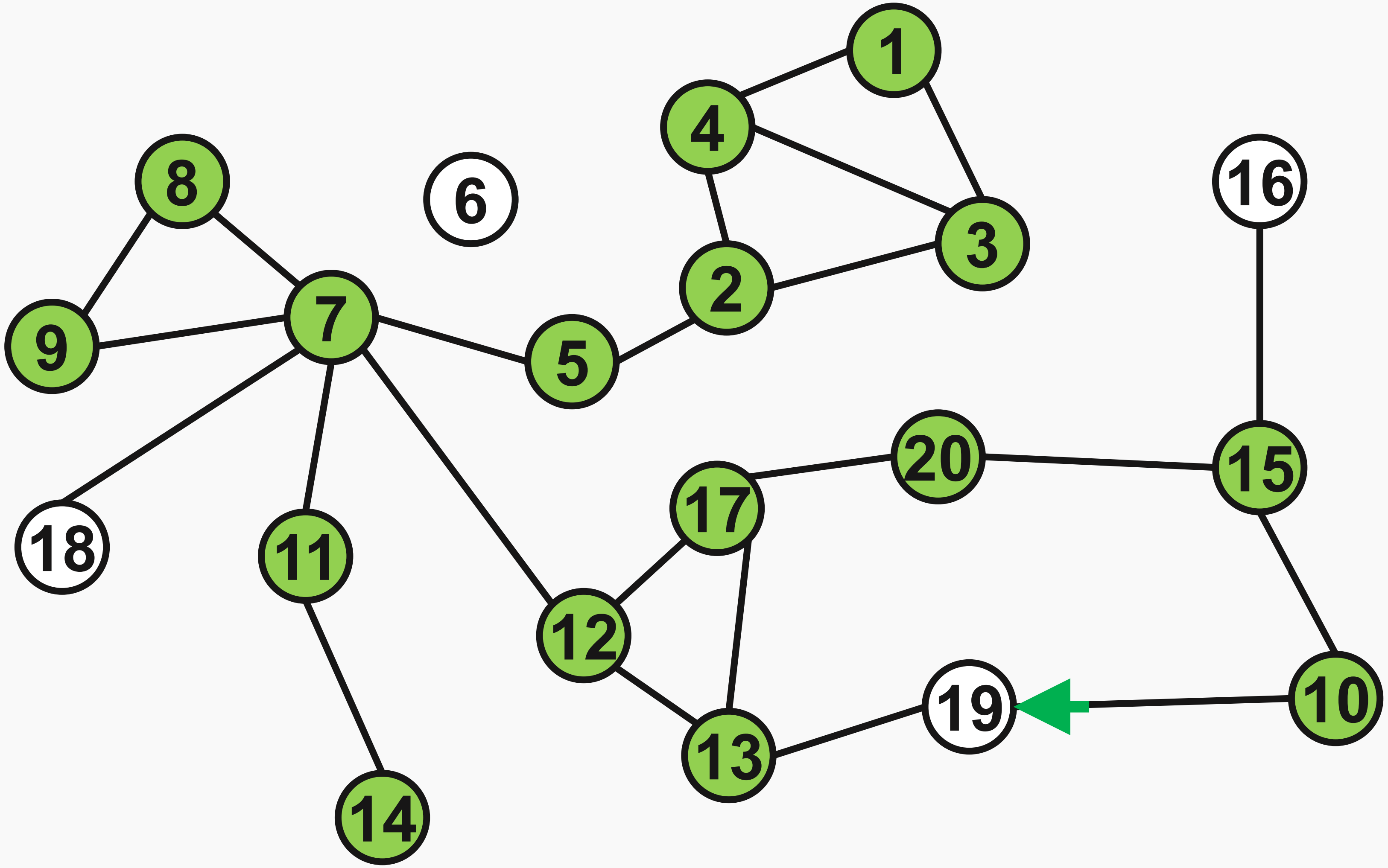
Parcurgere adâncime – depth first

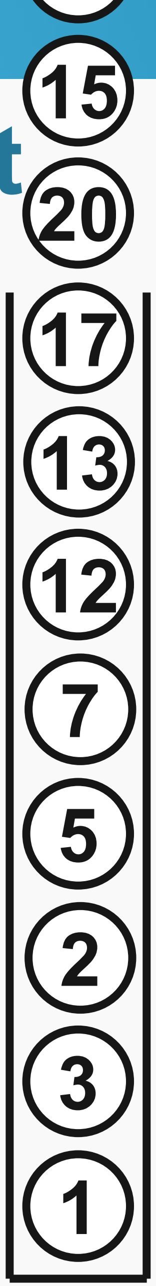


Parcurgere adâncime – depth first

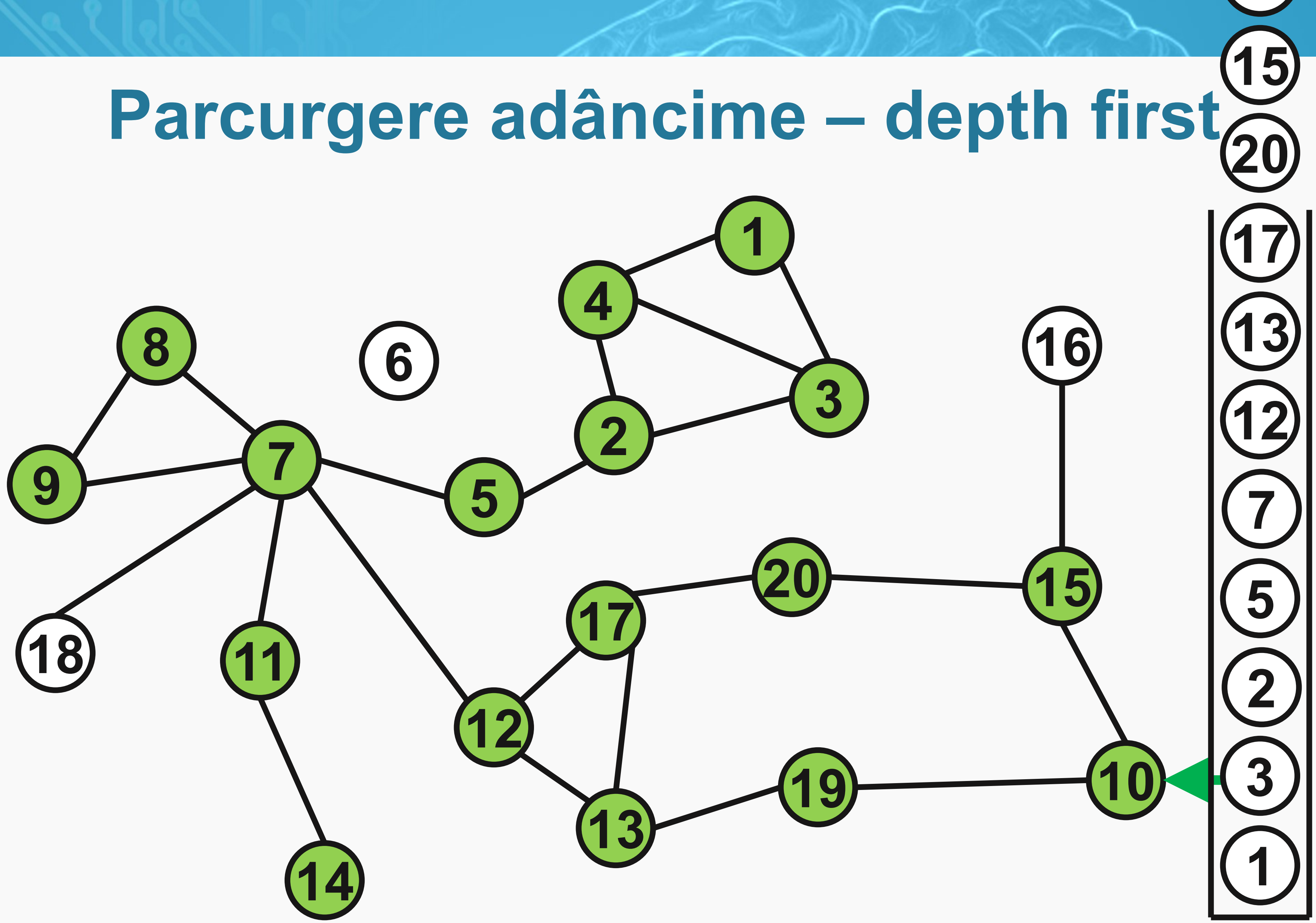


Parcurgere adâncime – depth first

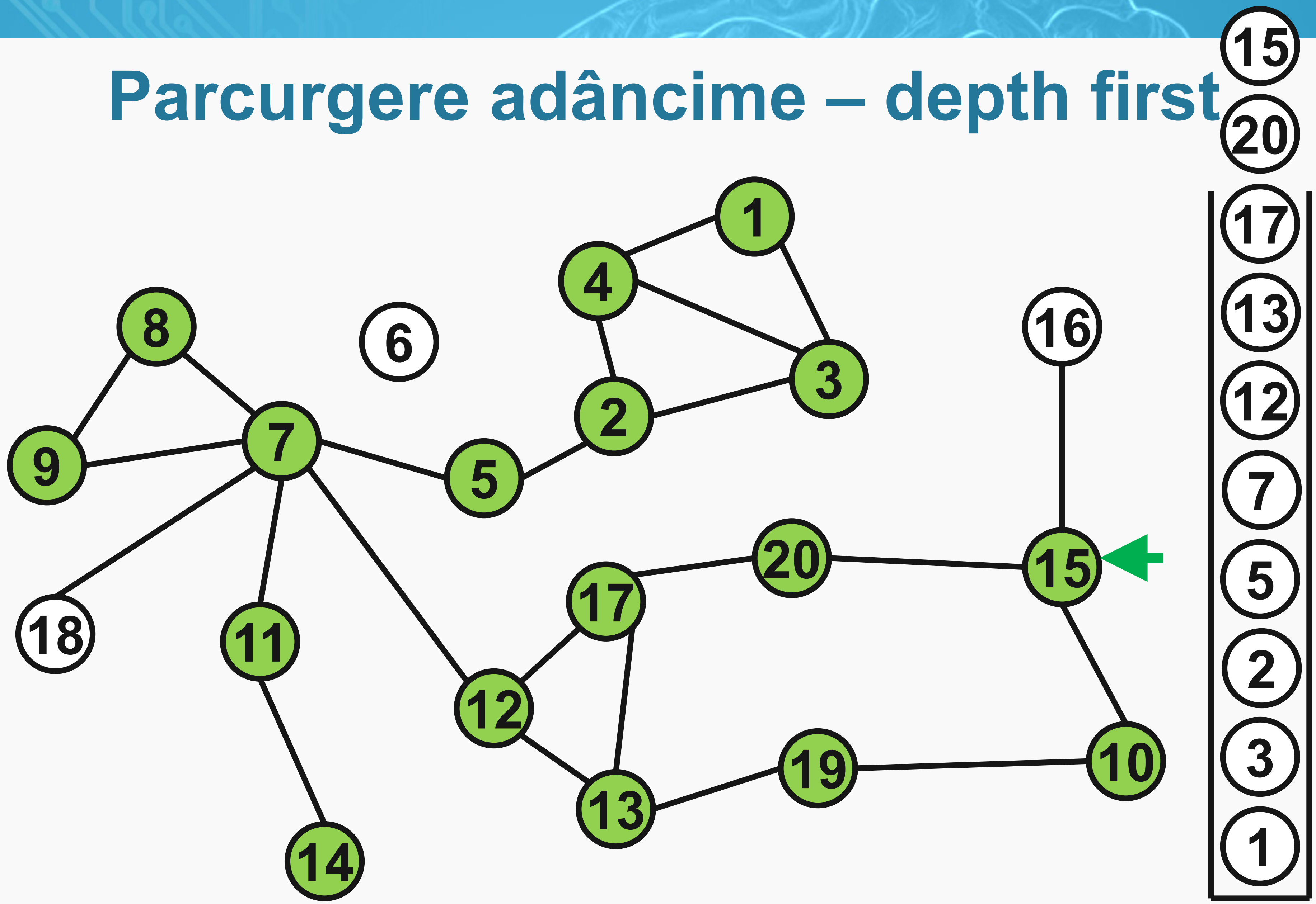




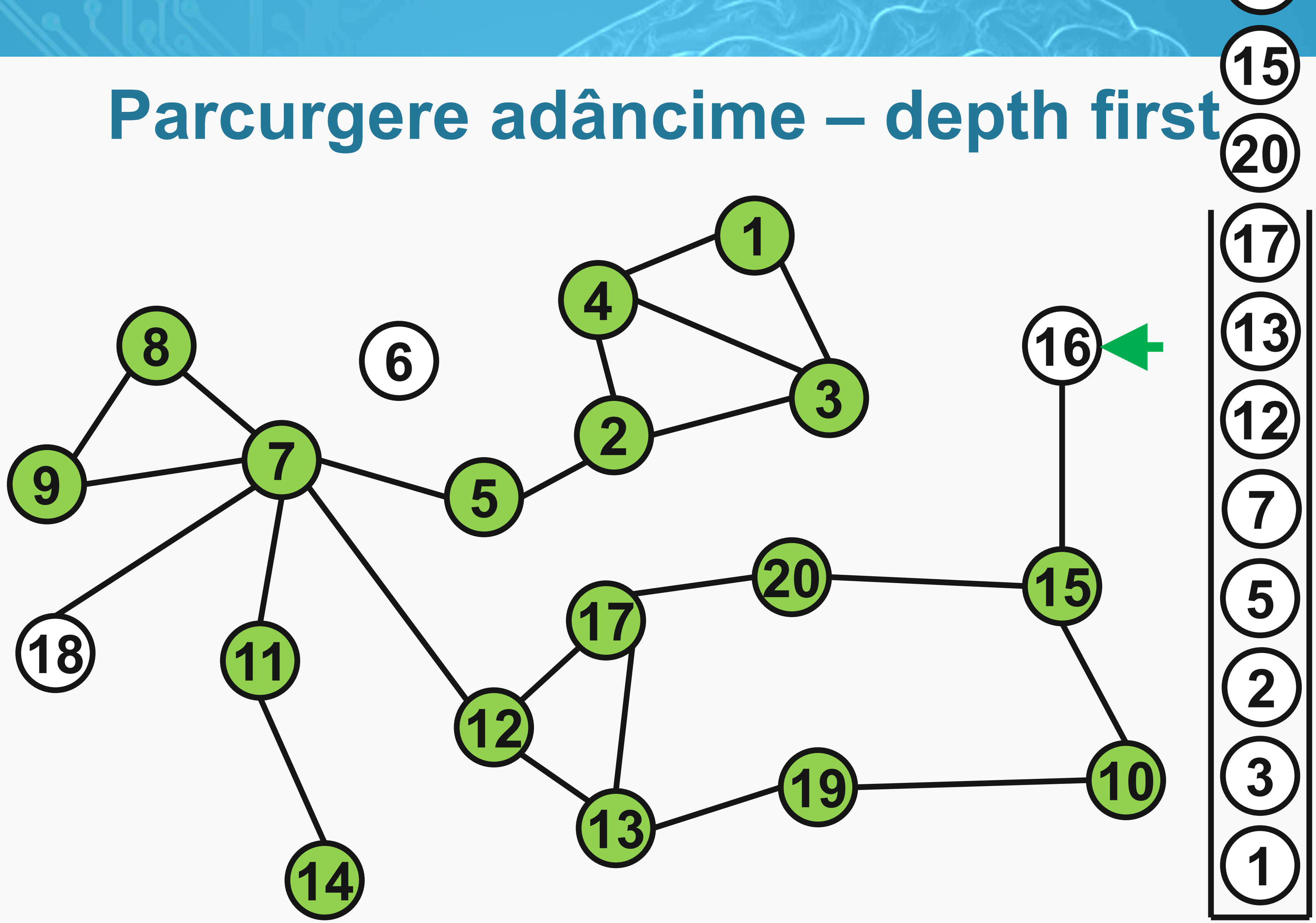
Parcurgere adâncime – depth first



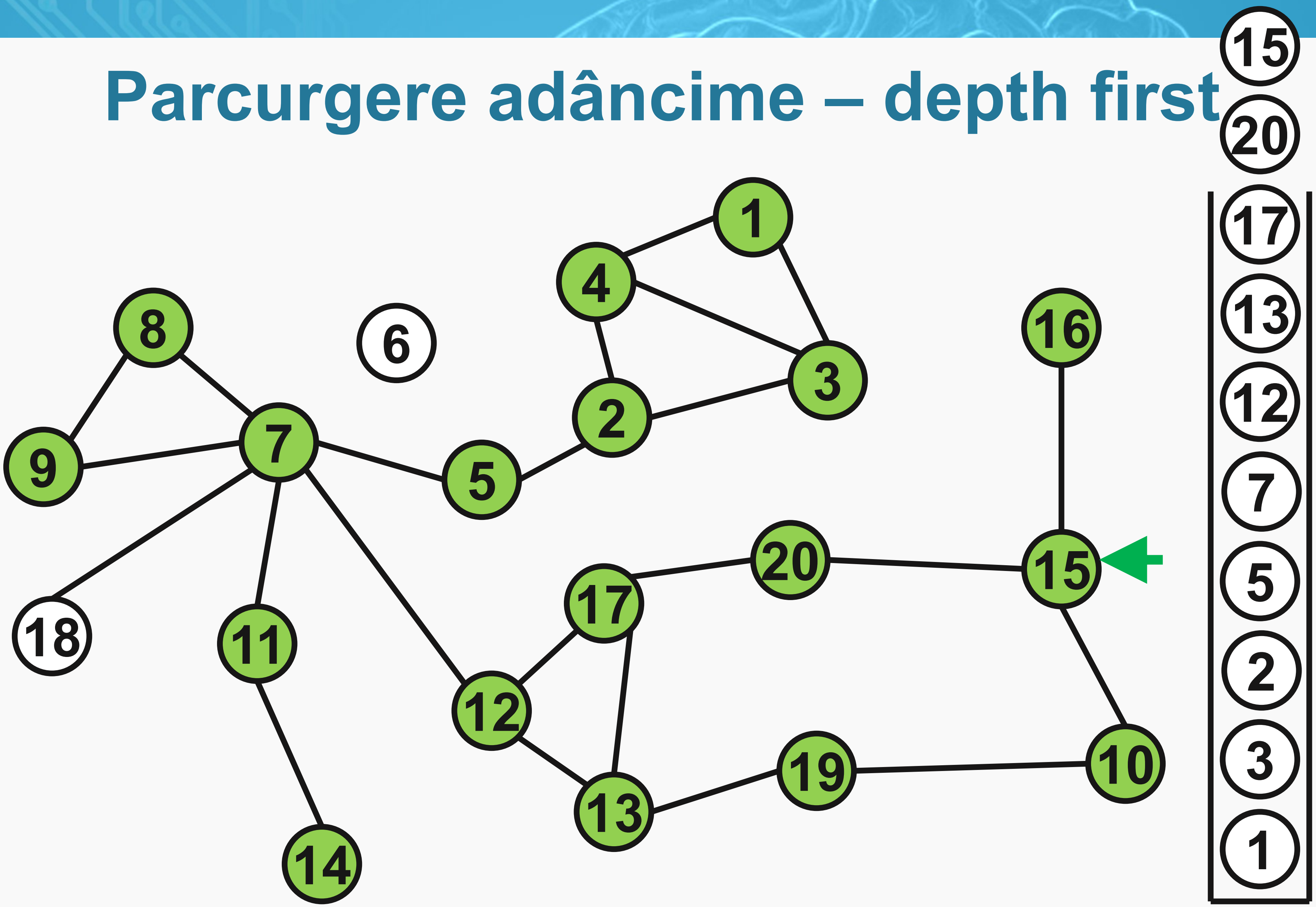
Parcurgere adâncime – depth first



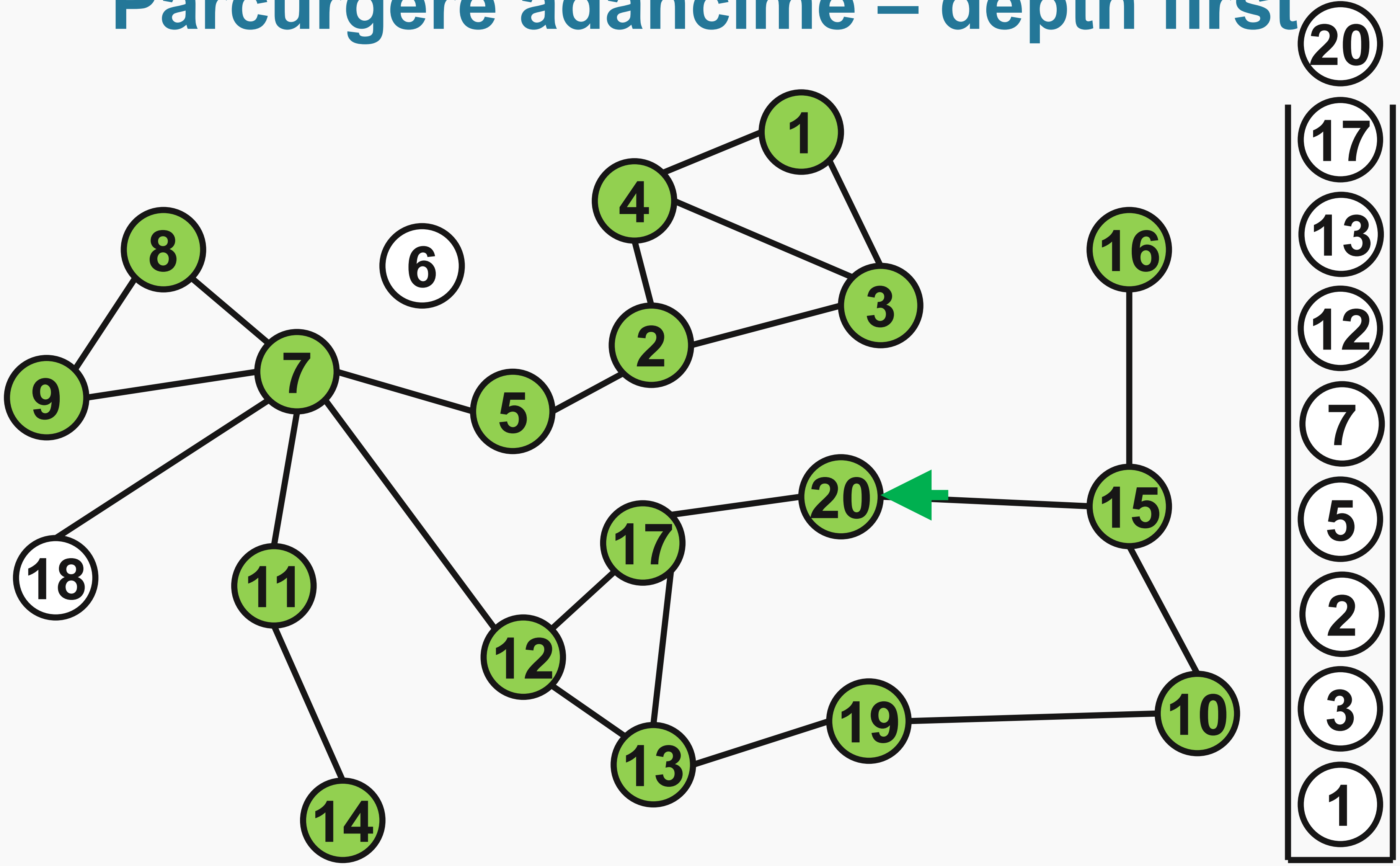
Parcurgere adâncime – depth first



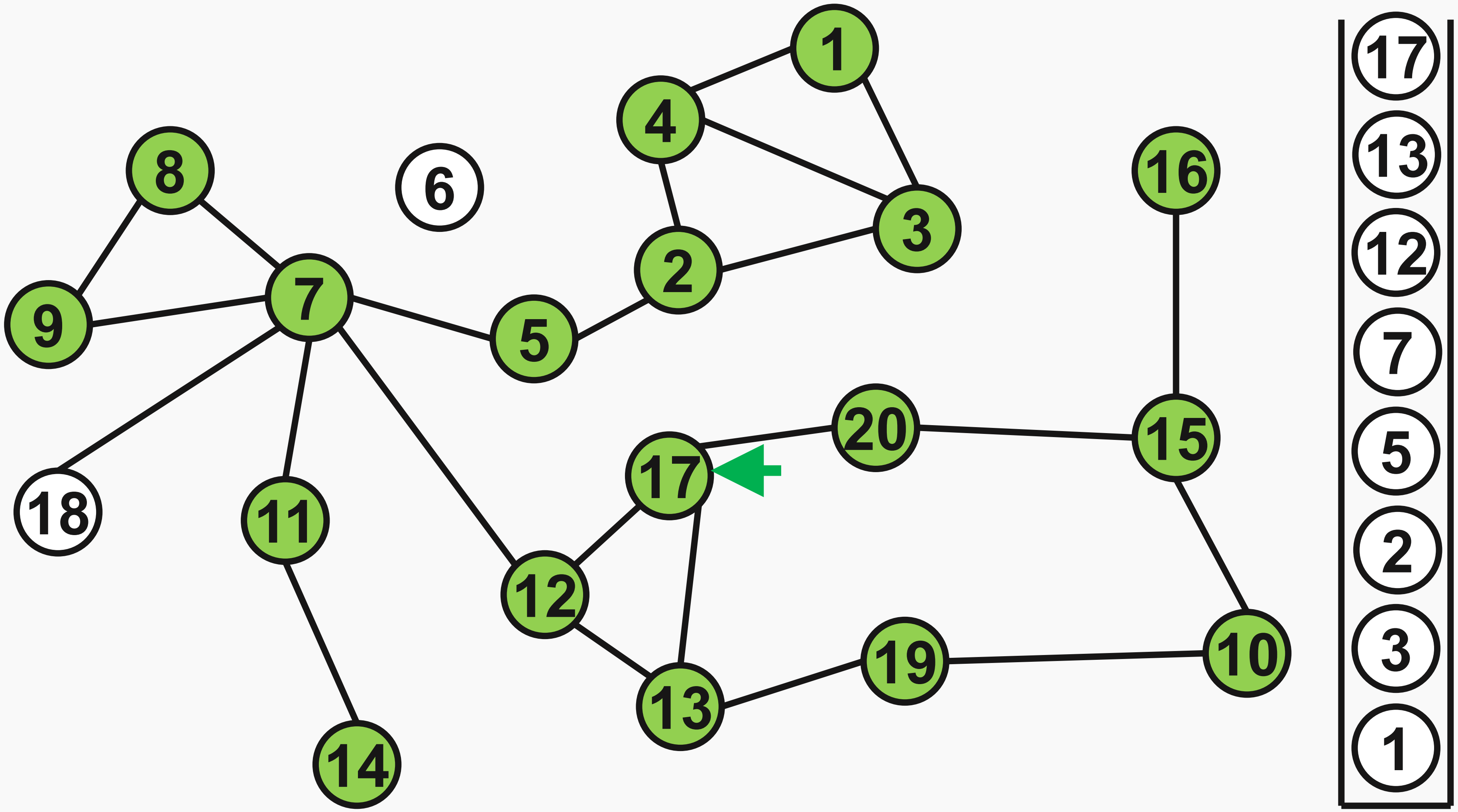
Parcurgere adâncime – depth first



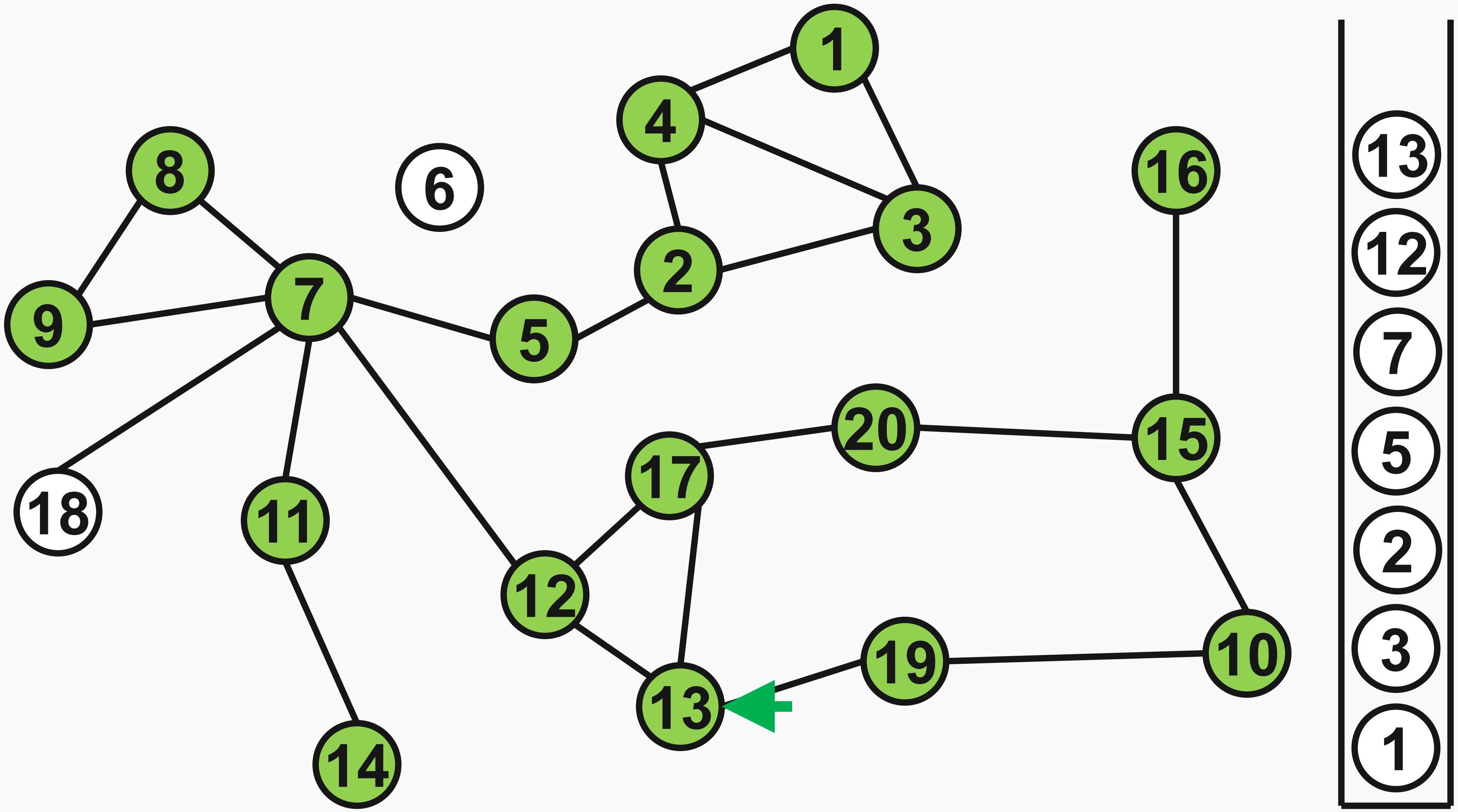
Parcurgere adâncime – depth first



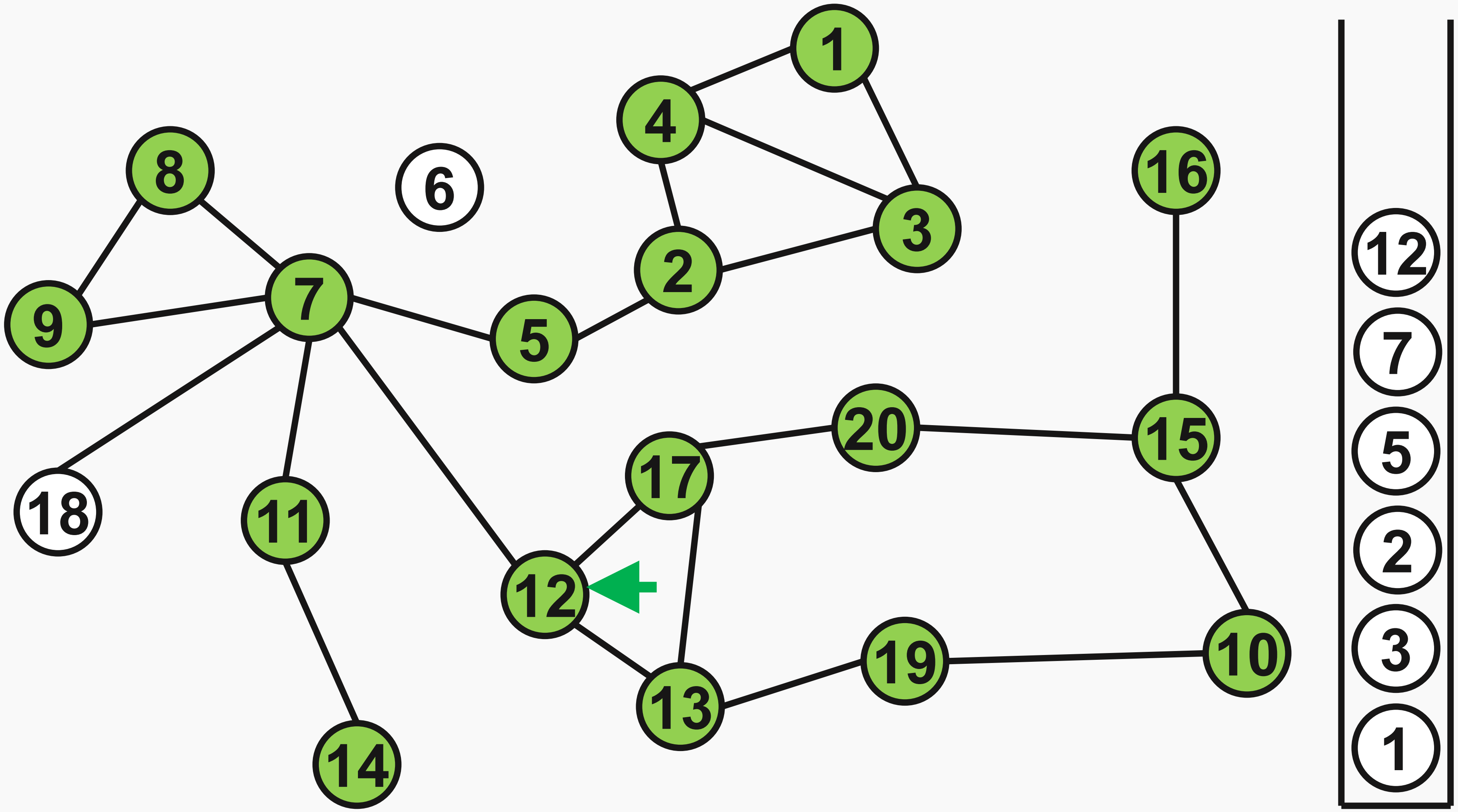
Parcurgere adâncime – depth first



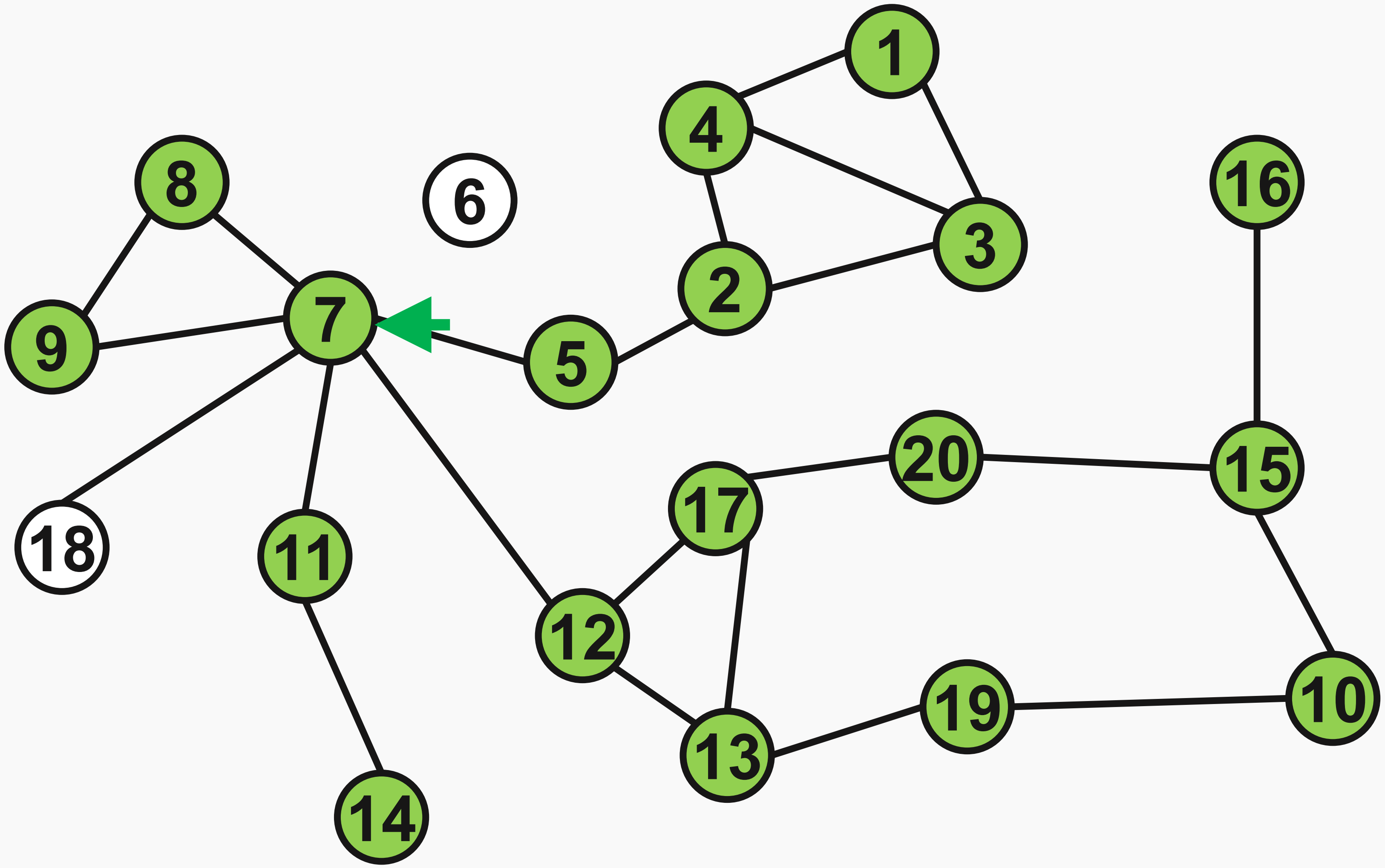
Parcurgere adâncime – depth first

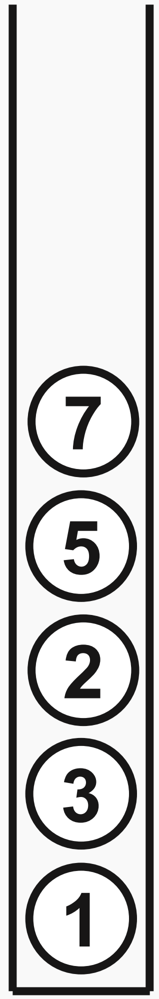


Parcurgere adâncime – depth first

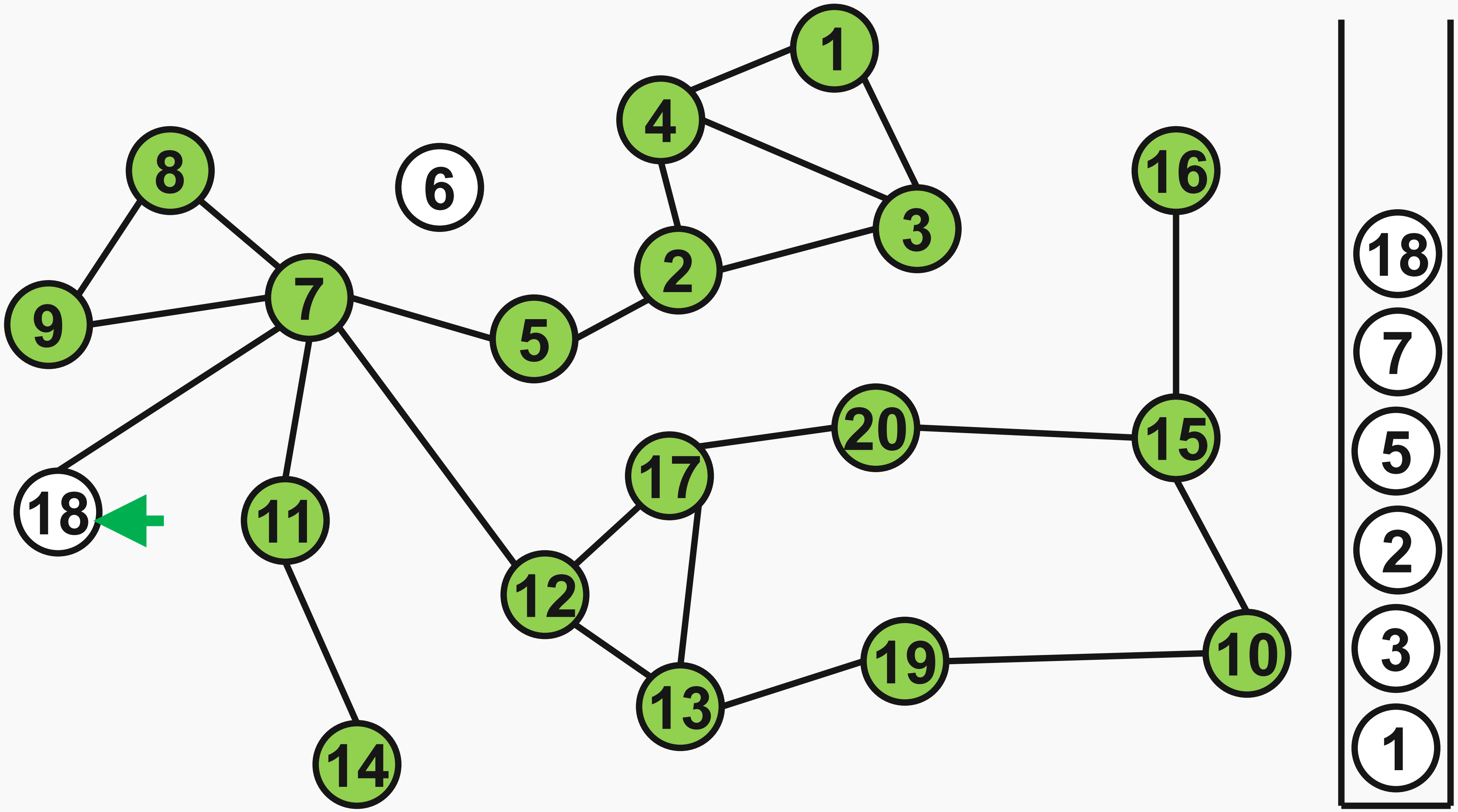


Parcurgere adâncime – depth first

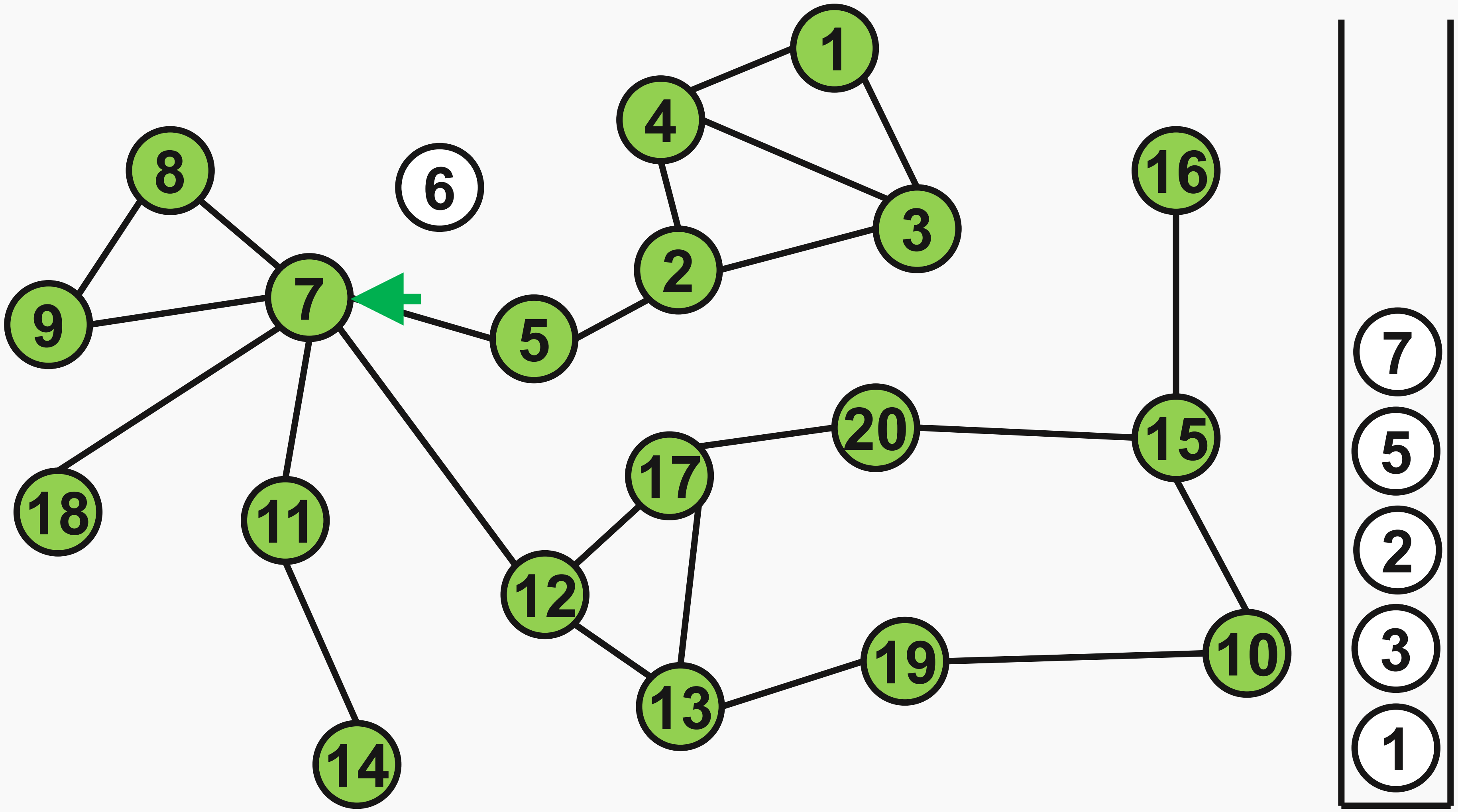




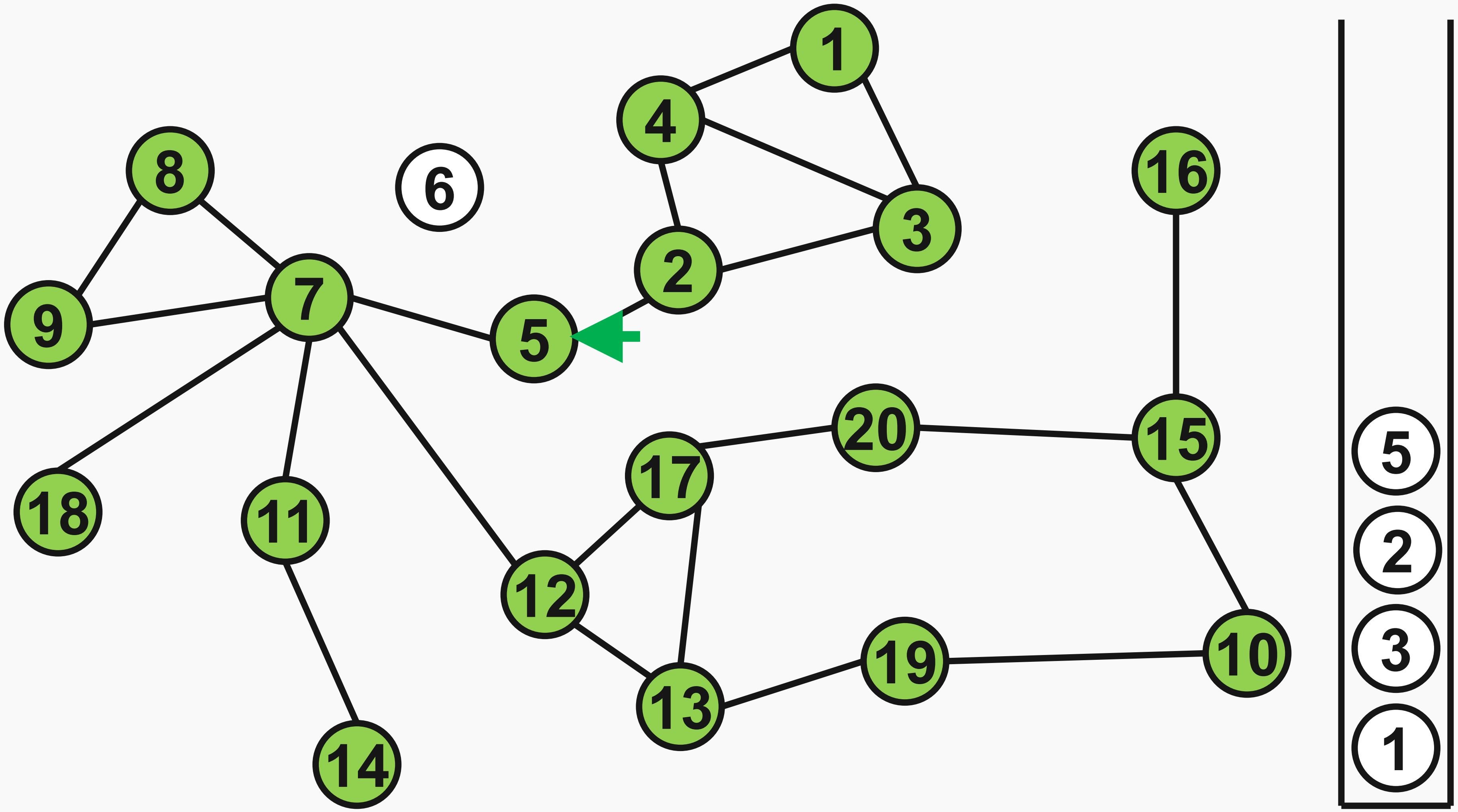
Parcurgere adâncime – depth first



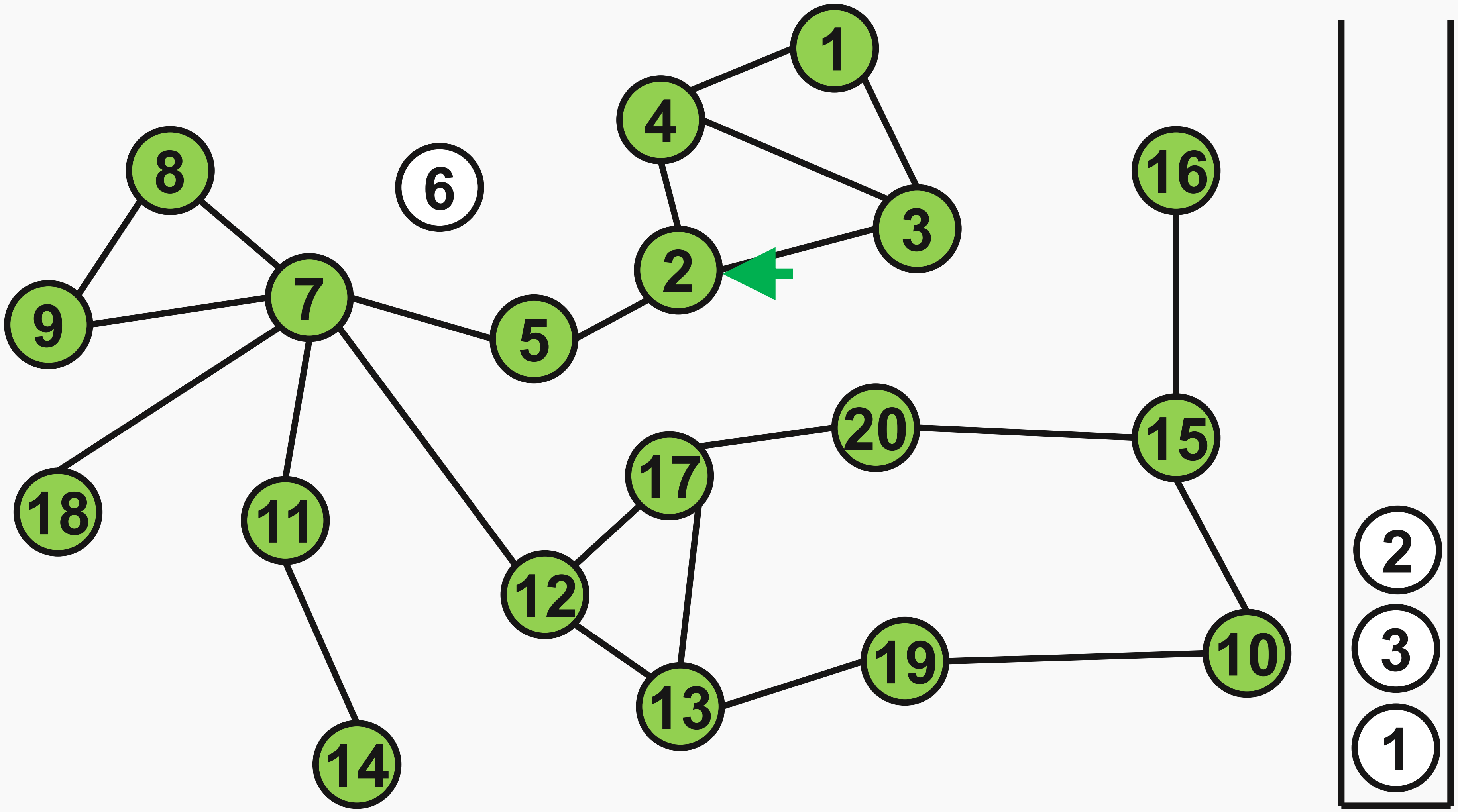
Parcurgere adâncime – depth first



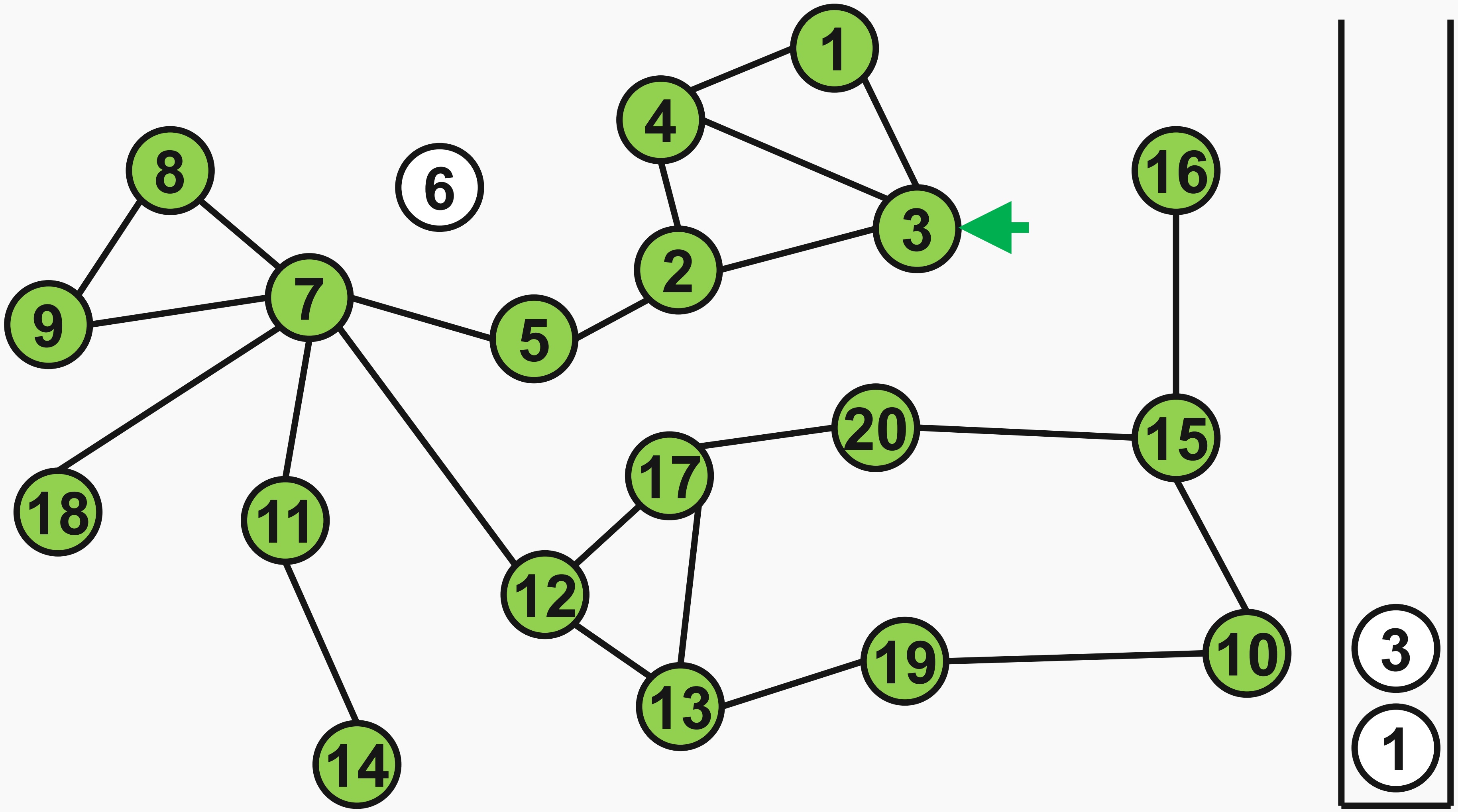
Parcurgere adâncime – depth first



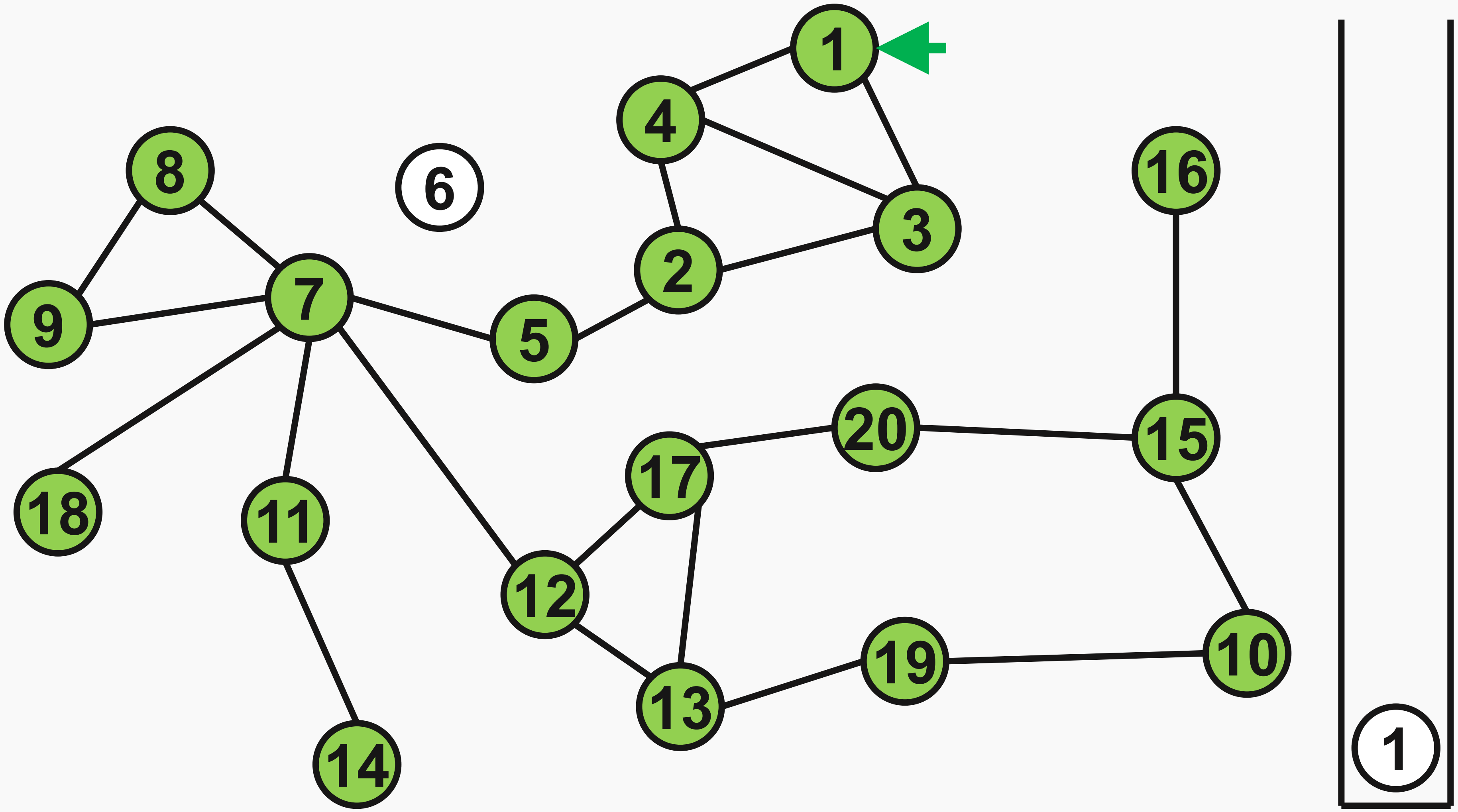
Parcurgere adâncime – depth first



Parcurgere adâncime – depth first



Parcurgere adâncime – depth first

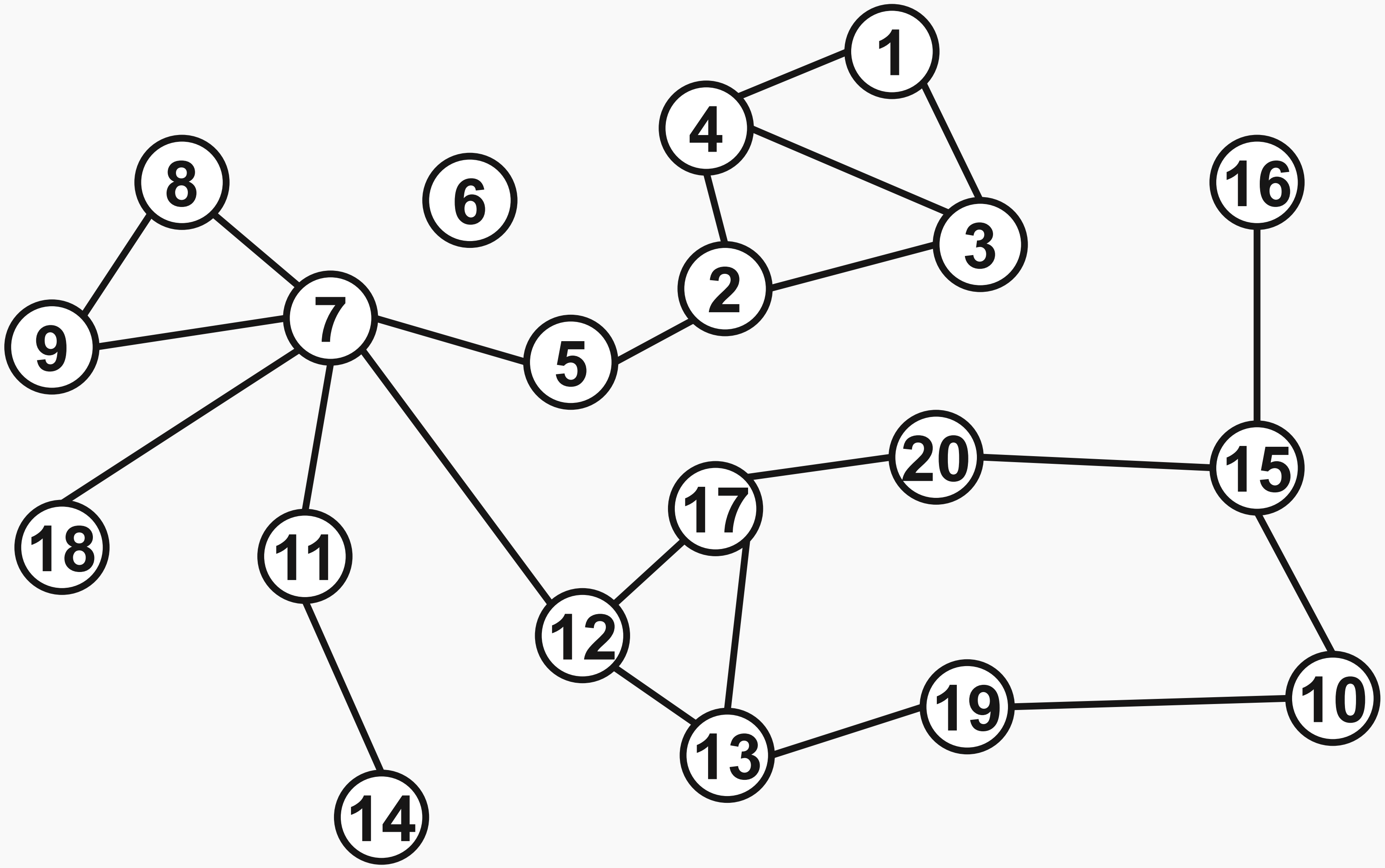


Parcurgere lățime – breadth first

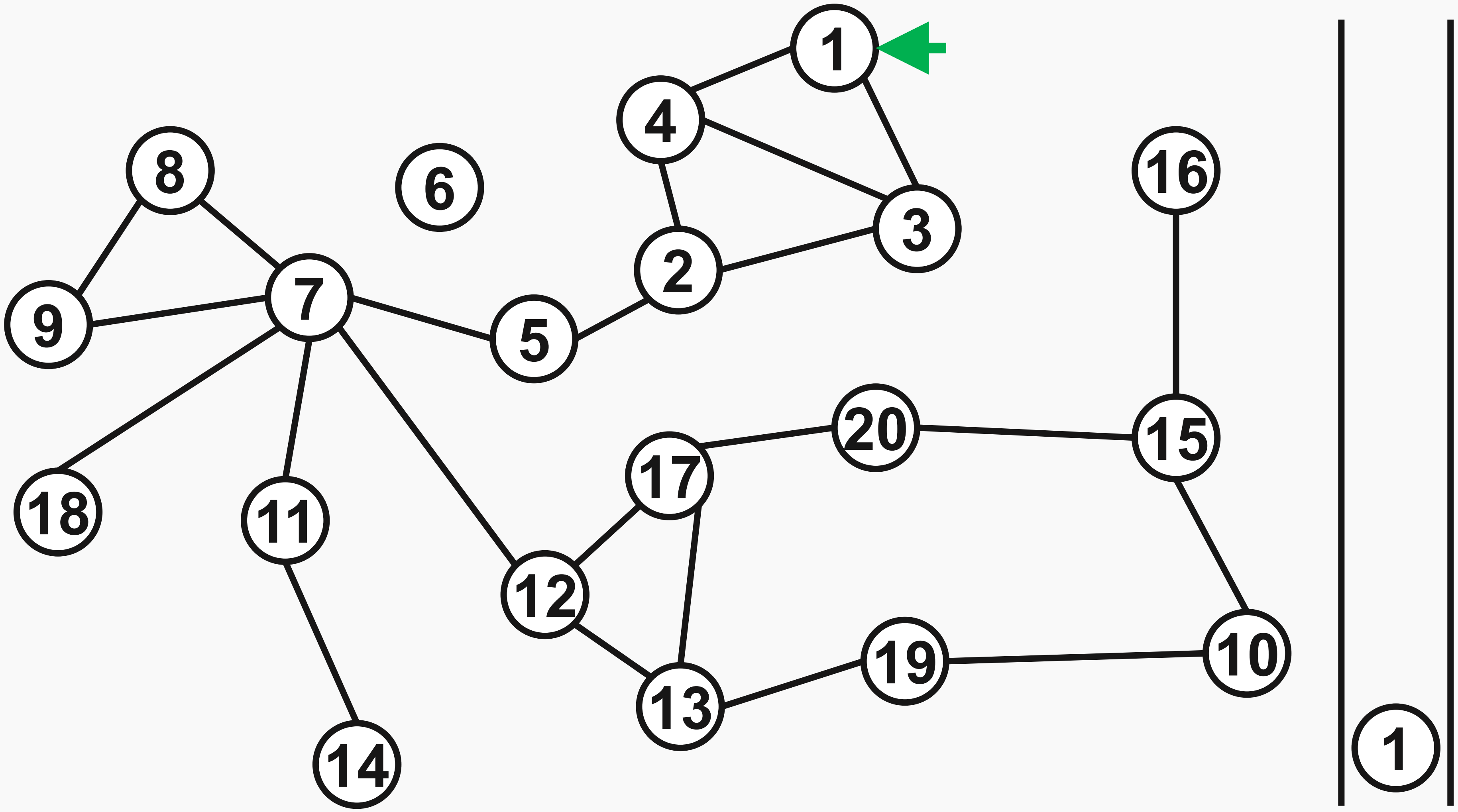
1945 by Konrad Zuse, in his (rejected) Ph.D. thesis on the Plankalkül programming language, but this was not published until 1972

```
int visited[N] = { 0 };
void BFS(vertex startNode) {
    push(queue, startNode);
    while (!isEmpty(queue)) {
        currentNode = pop(queue);
        visited[currentNode.name] = 1;
        for (int i = 0; i < currentNode.numNeighbors; i++) {
            if (!visited[currentNode.neighbors[i]->name]) {
                push(queue, *(currentNode.neighbors[i]));
                visited[currentNode.neighbors[i]->name] = 2;
            }
        }
    }
}
```

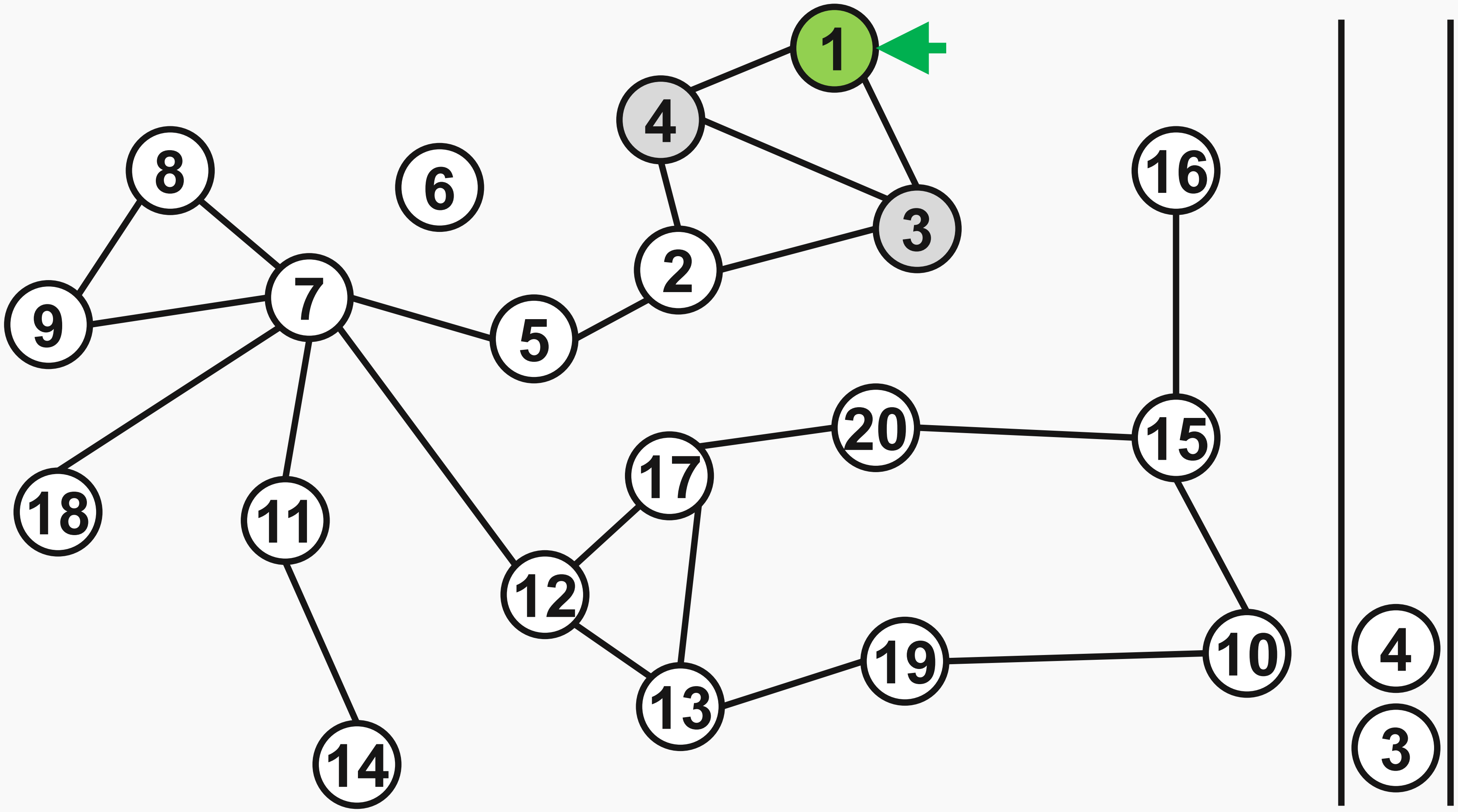
Parcurgere lățime – breadth first



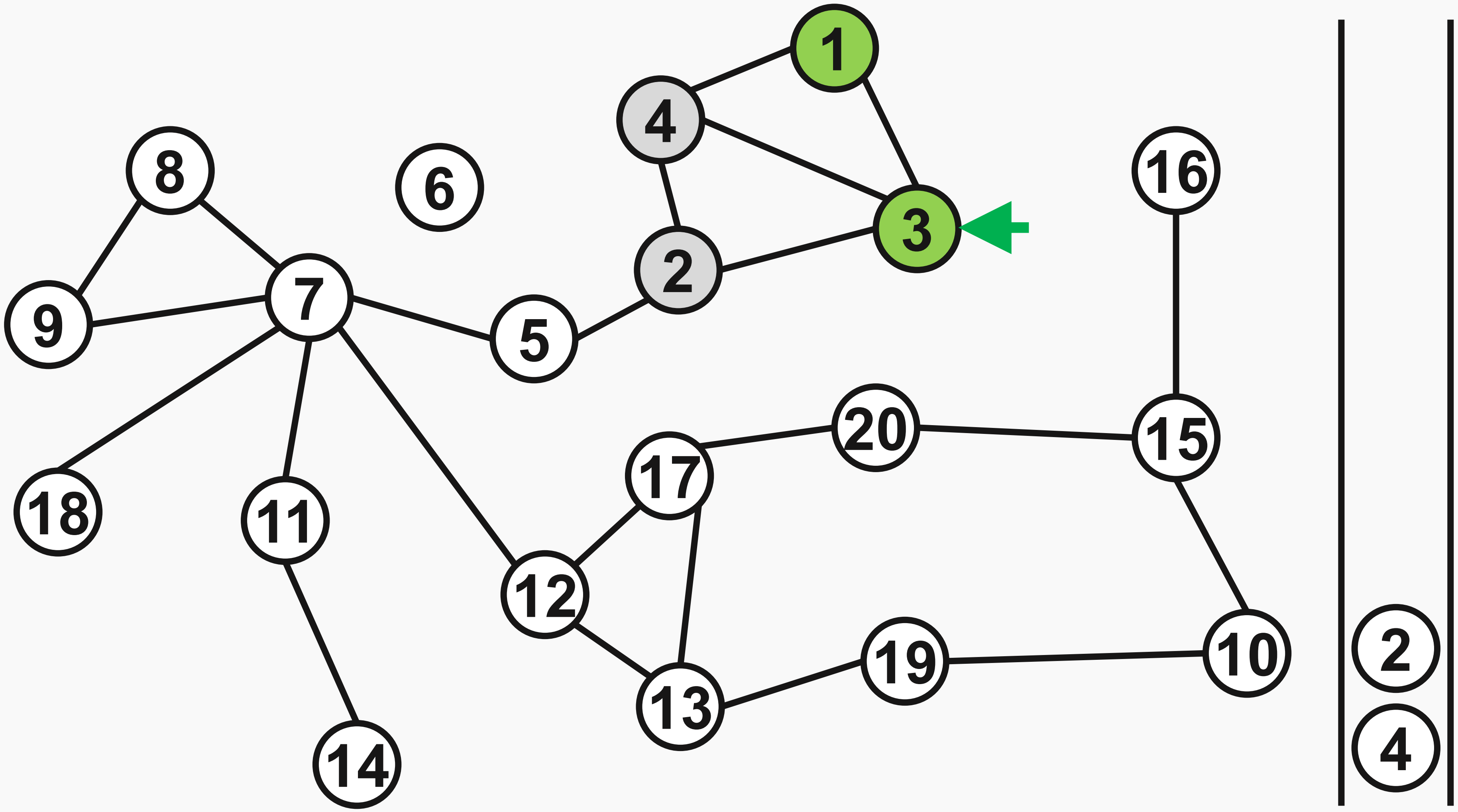
Parcurgere lăţime – breadth first



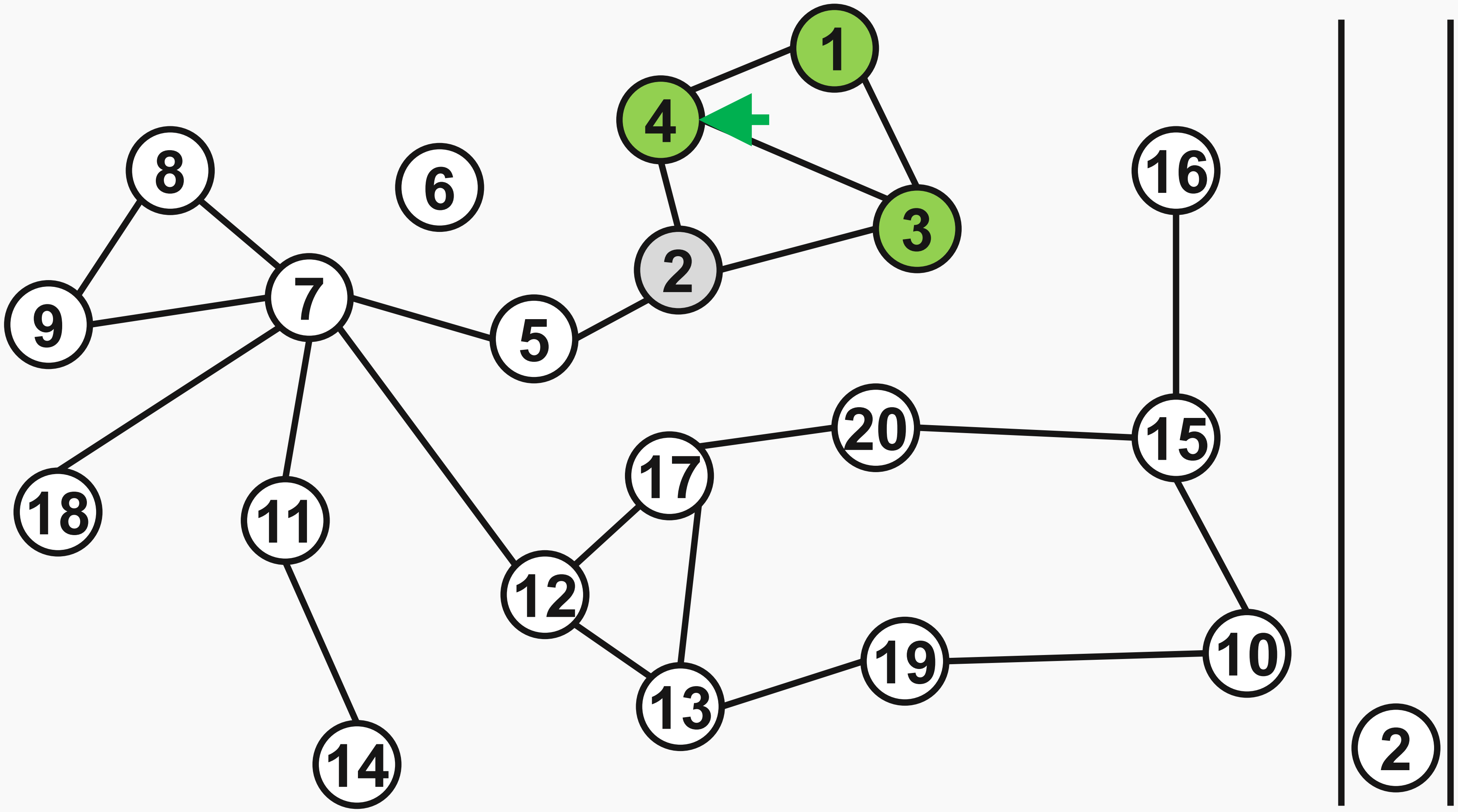
Parcurgere lățime – breadth first



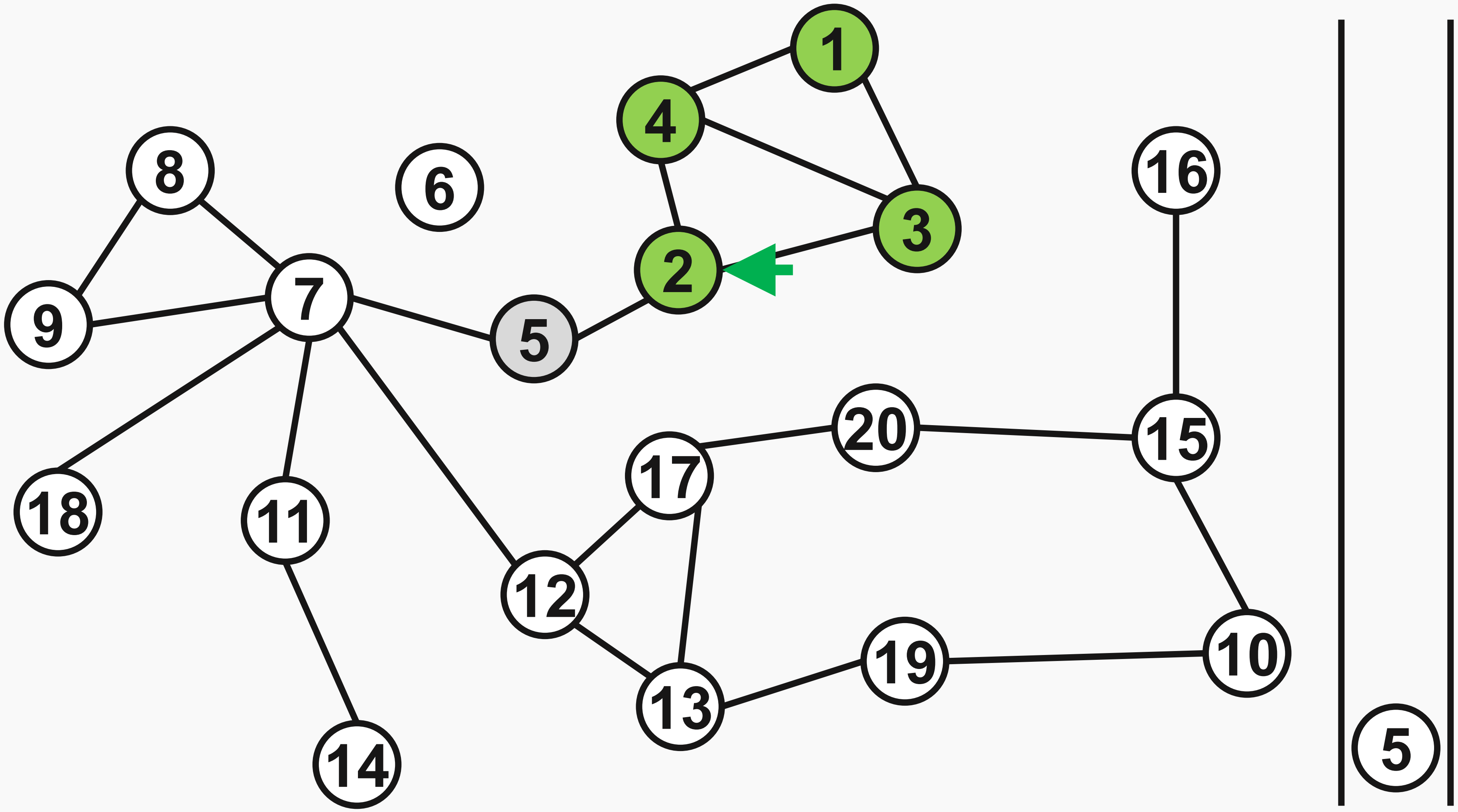
Parcurgere lățime – breadth first



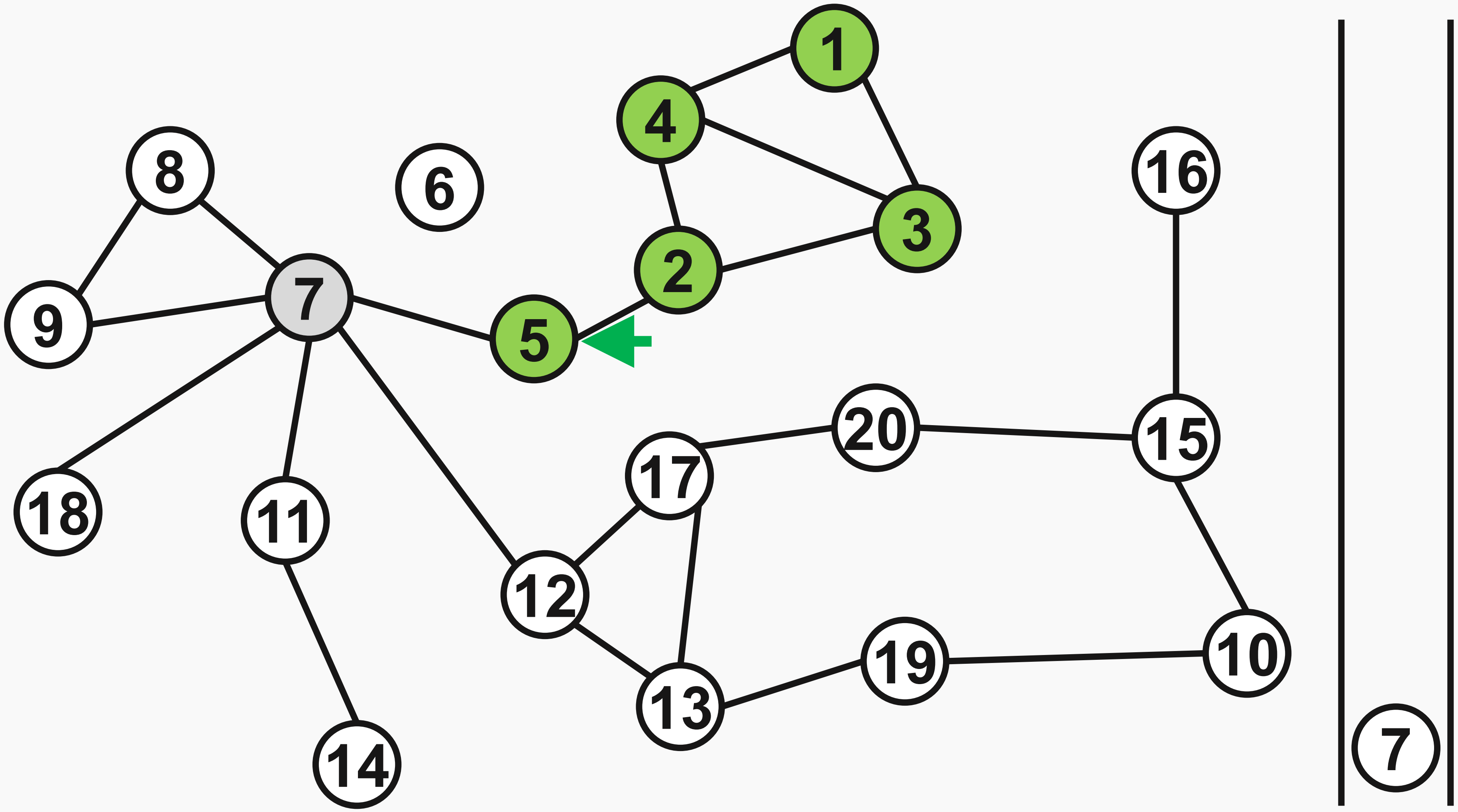
Parcurgere lățime – breadth first



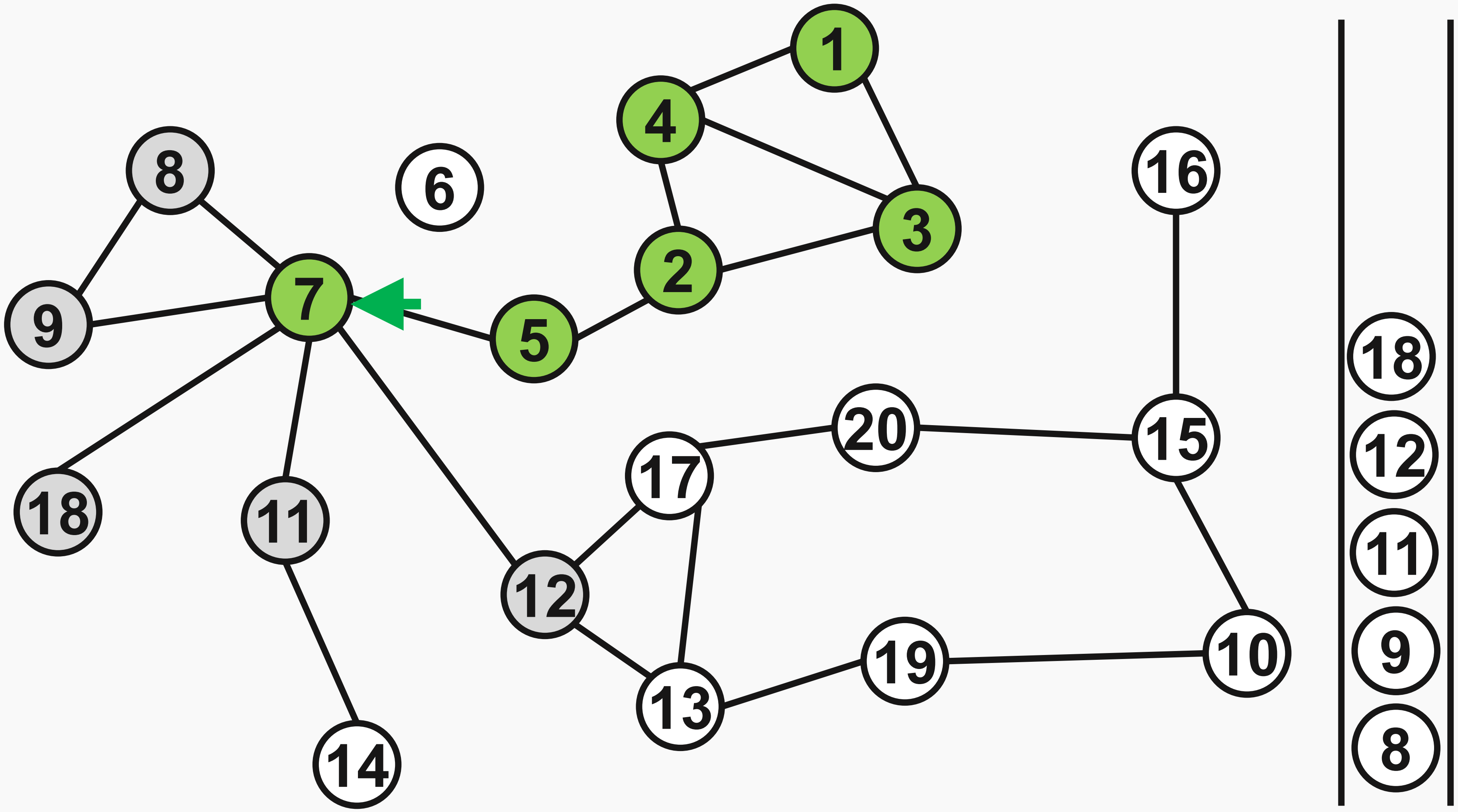
Parcurgere lățime – breadth first



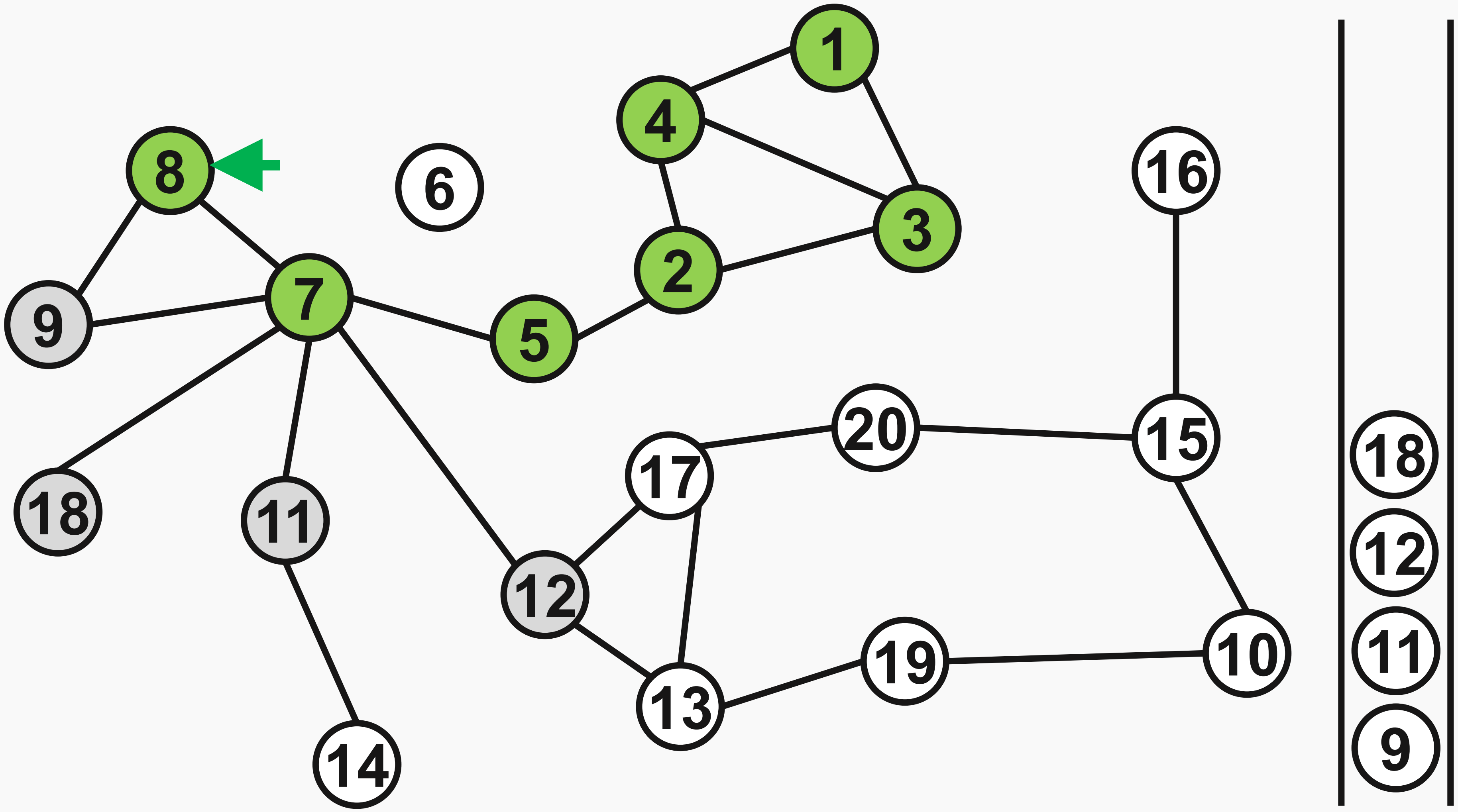
Parcurgere lățime – breadth first



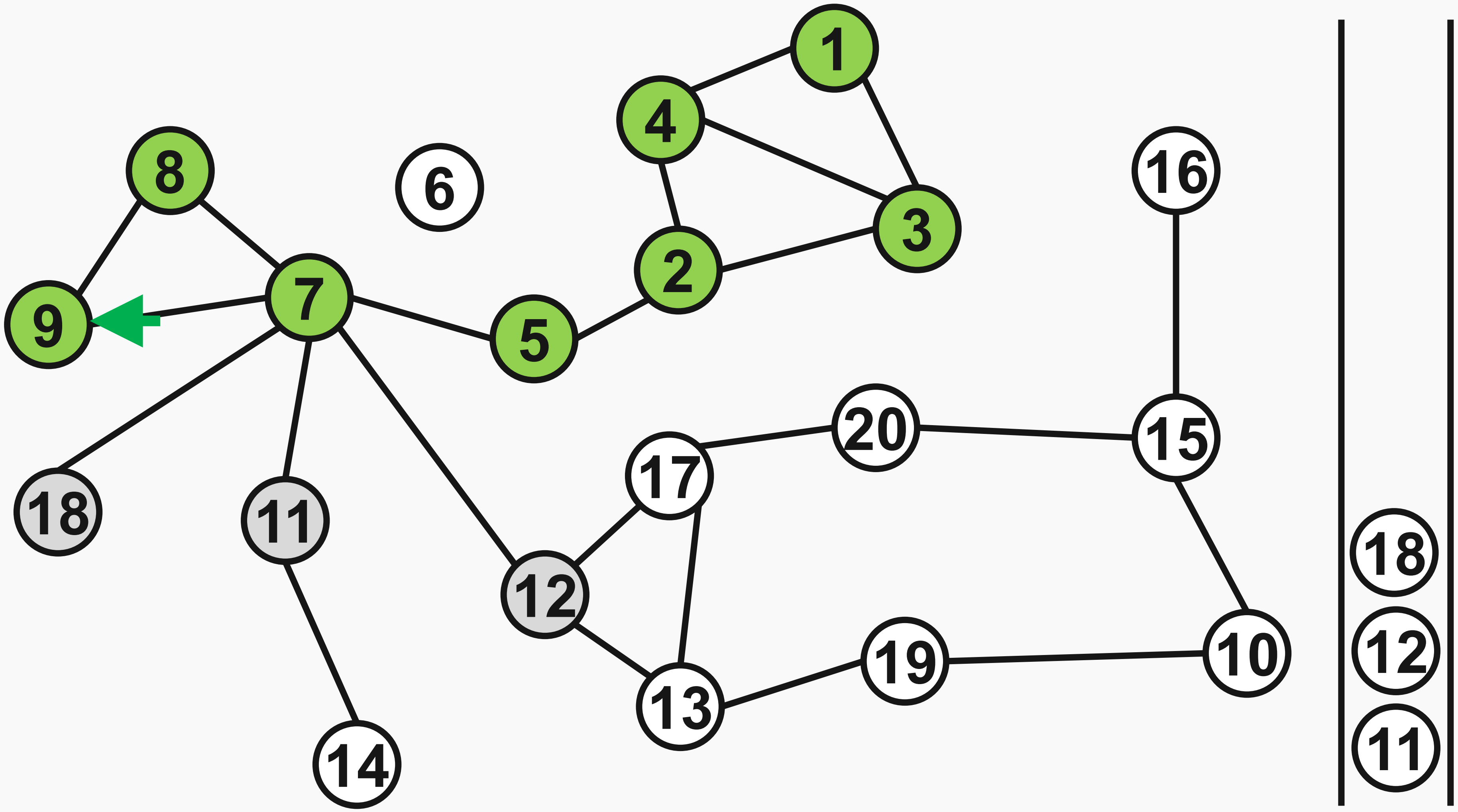
Parcurgere lățime – breadth first



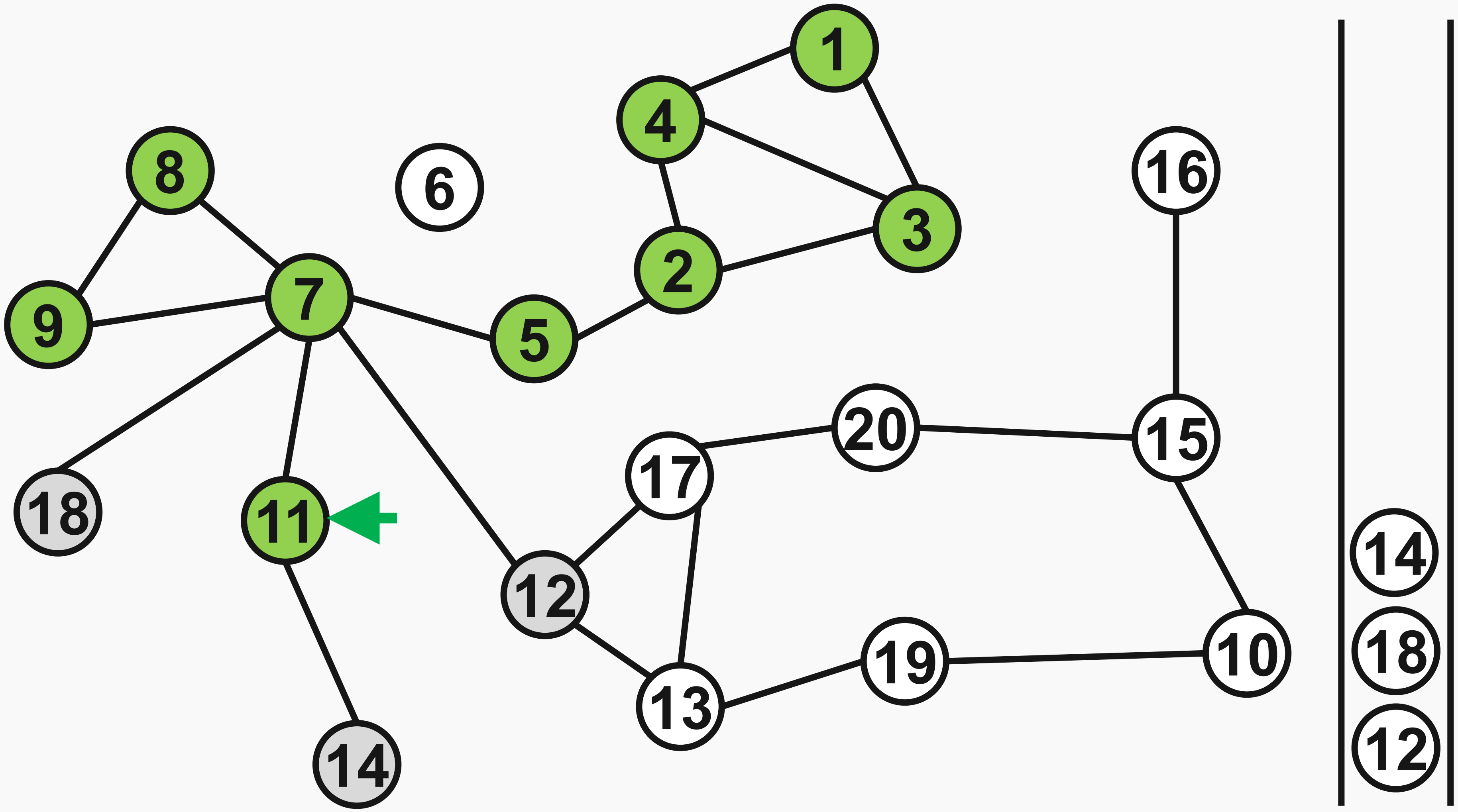
Parcurgere lățime – breadth first



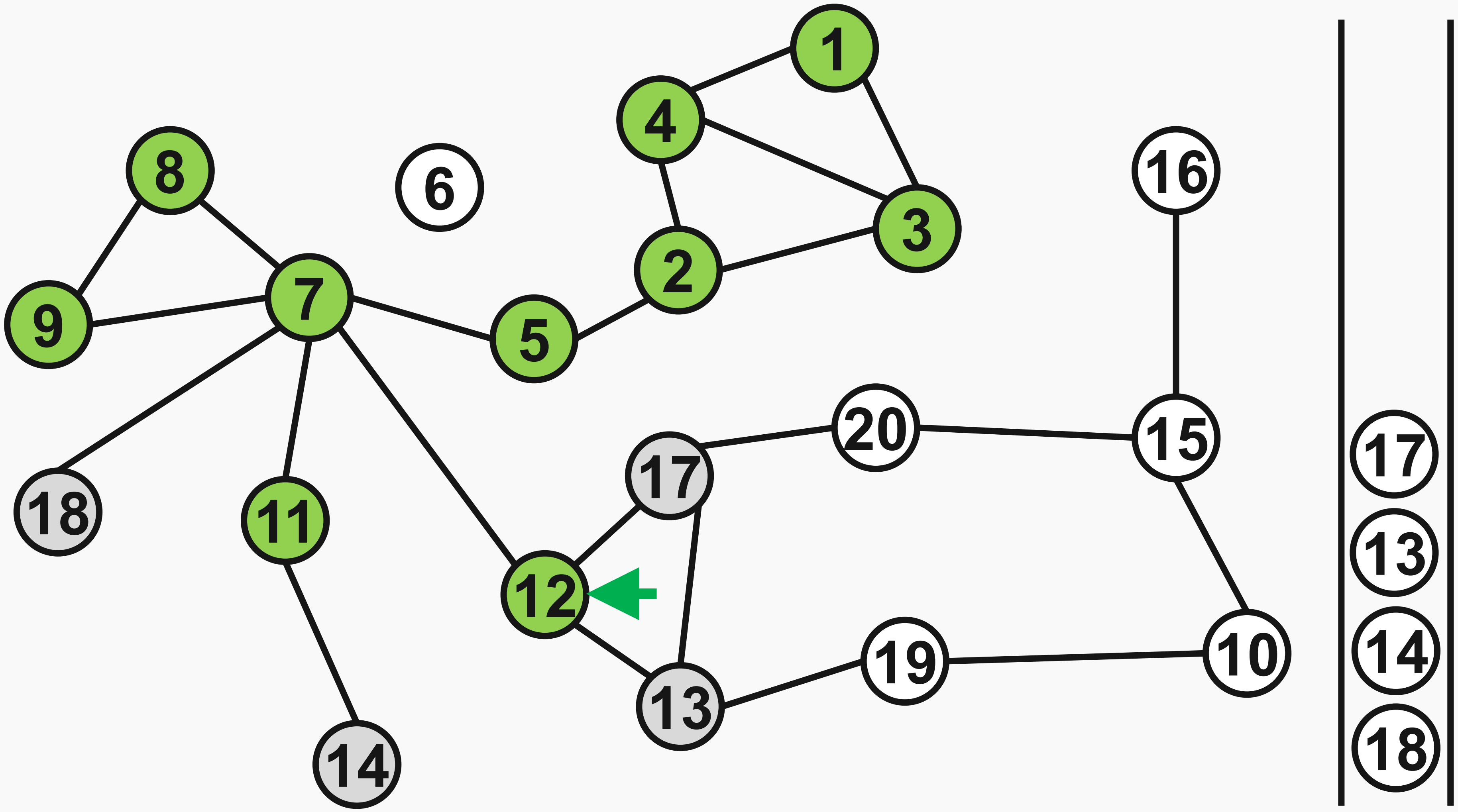
Parcurgere lățime – breadth first



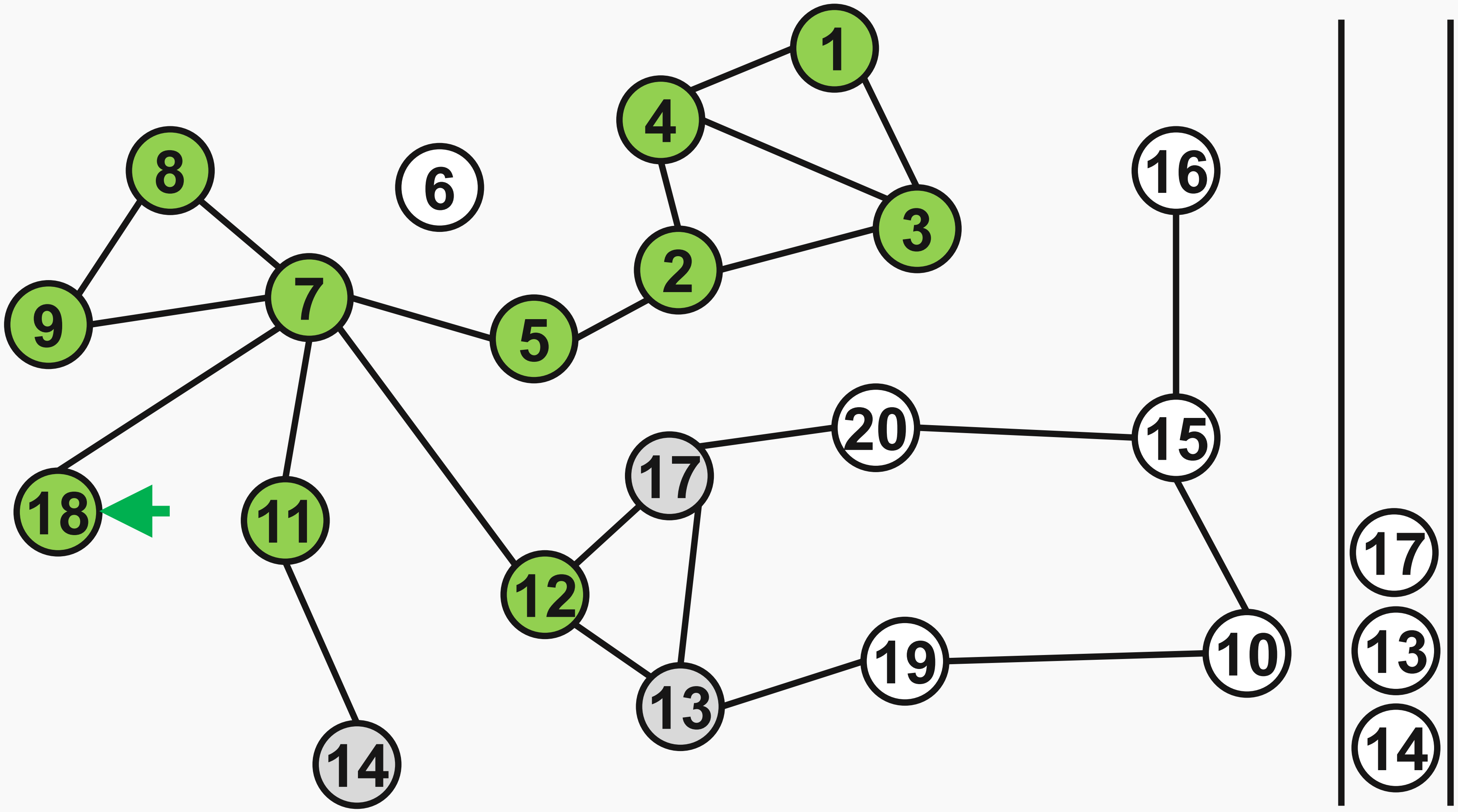
Parcurgere lățime – breadth first



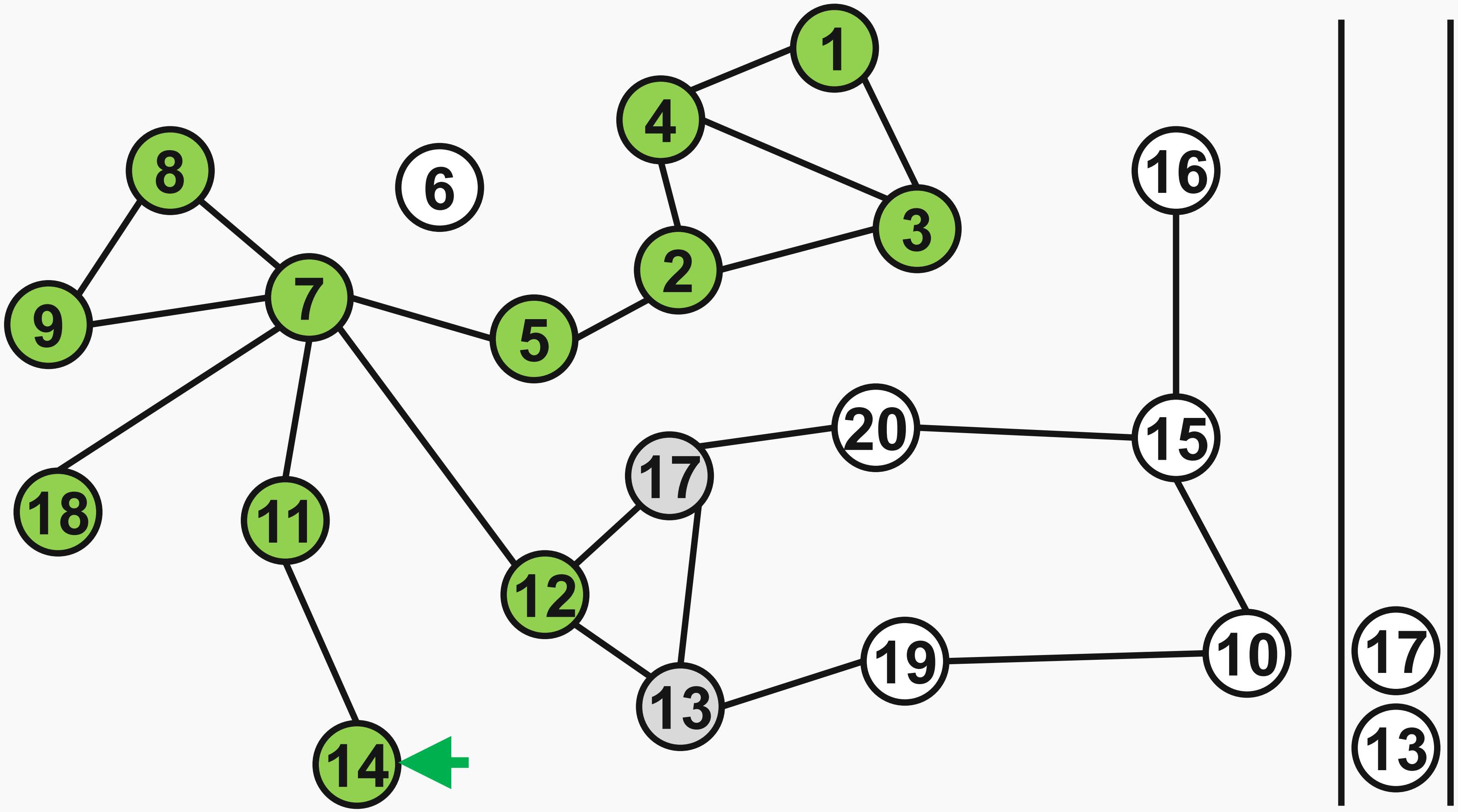
Parcurgere lățime – breadth first



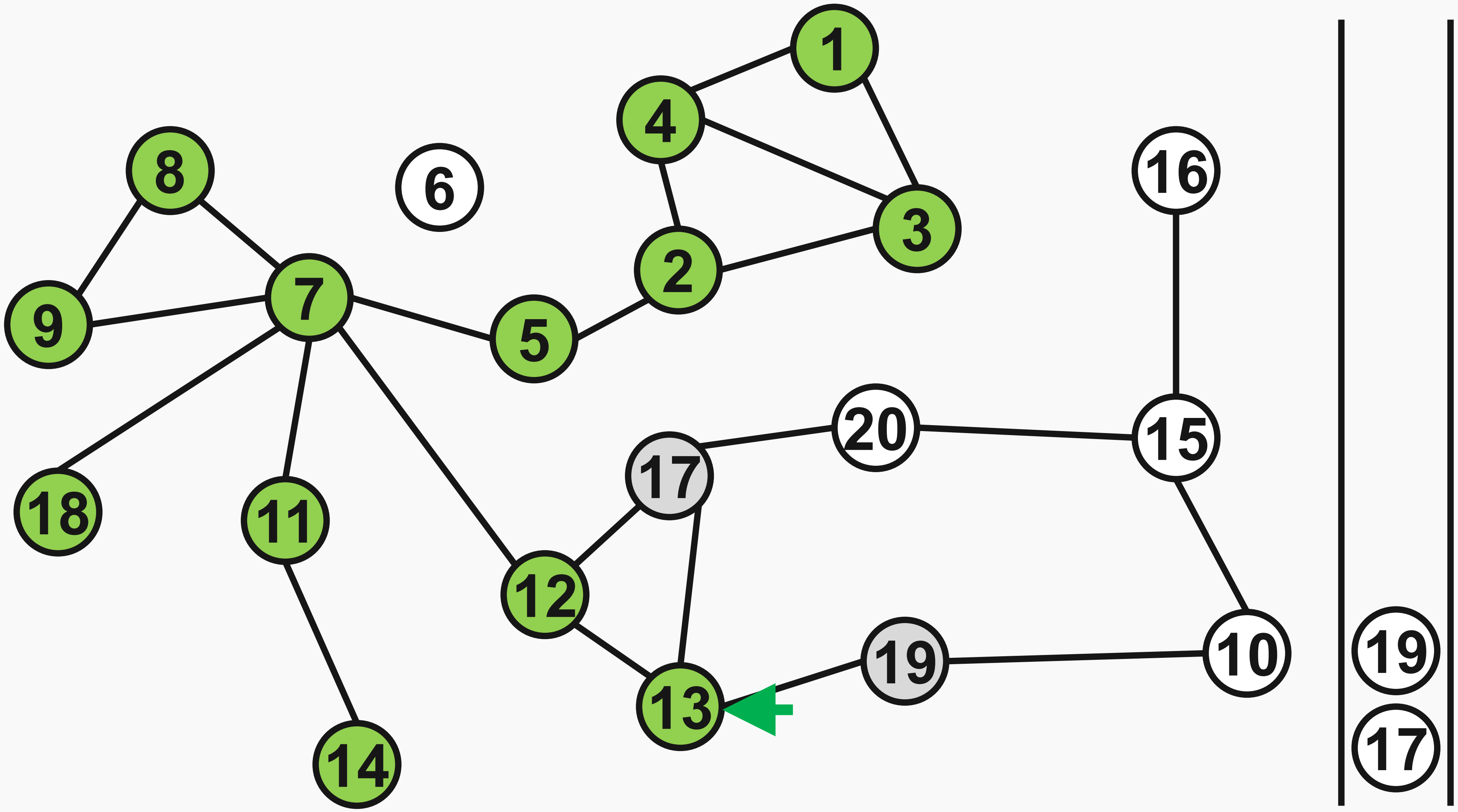
Parcurgere lățime – breadth first



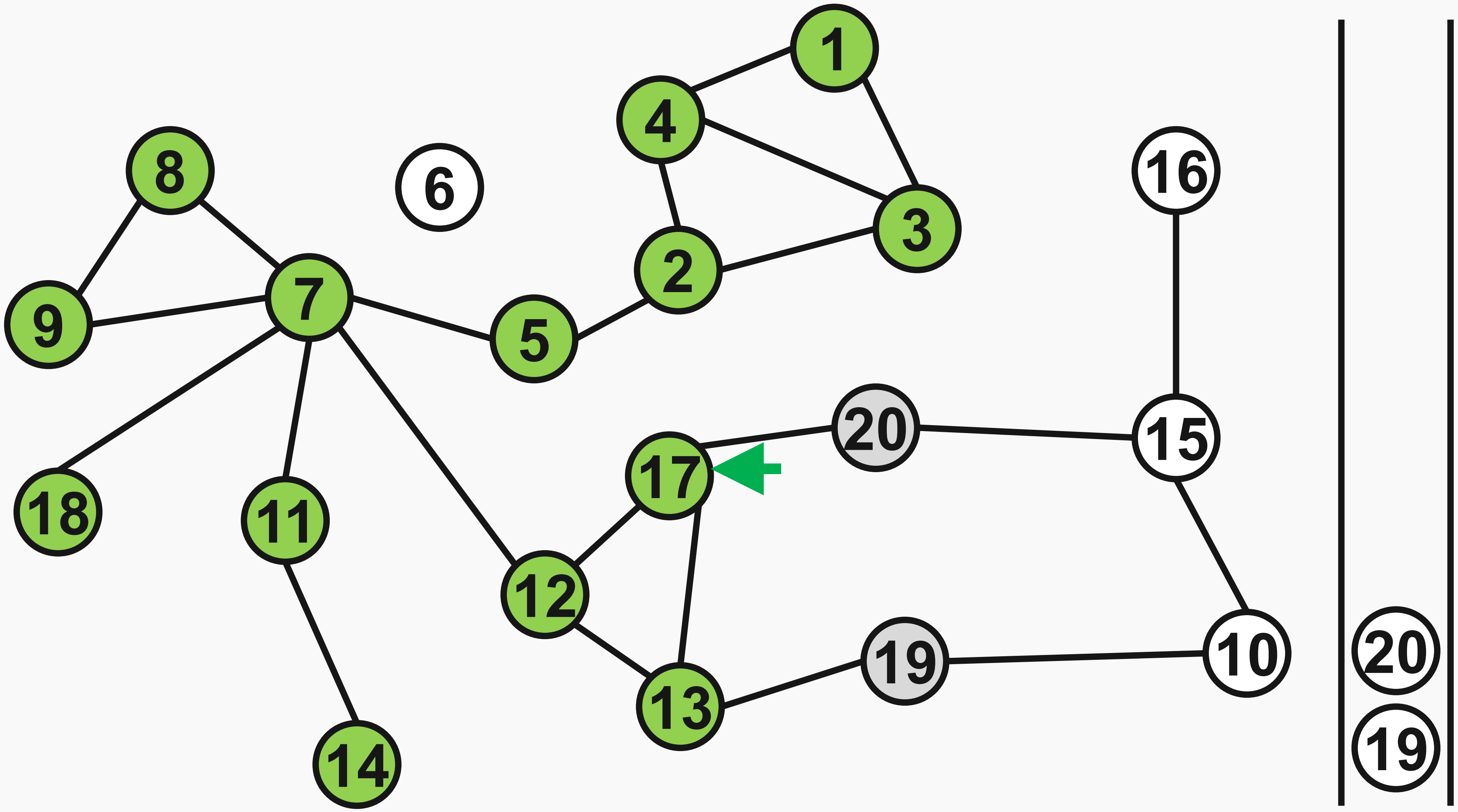
Parcurgere lățime – breadth first



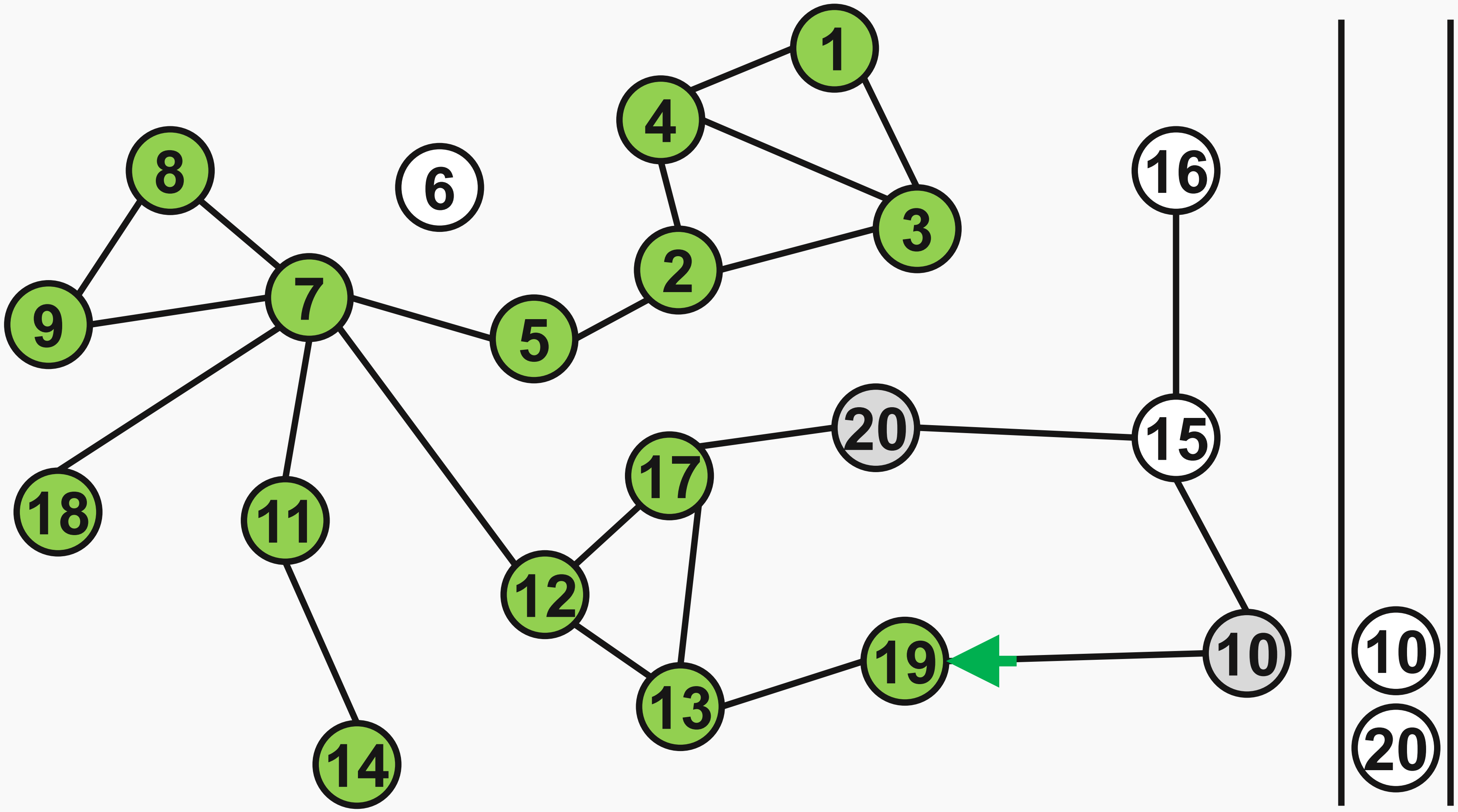
Parcurgere lățime – breadth first



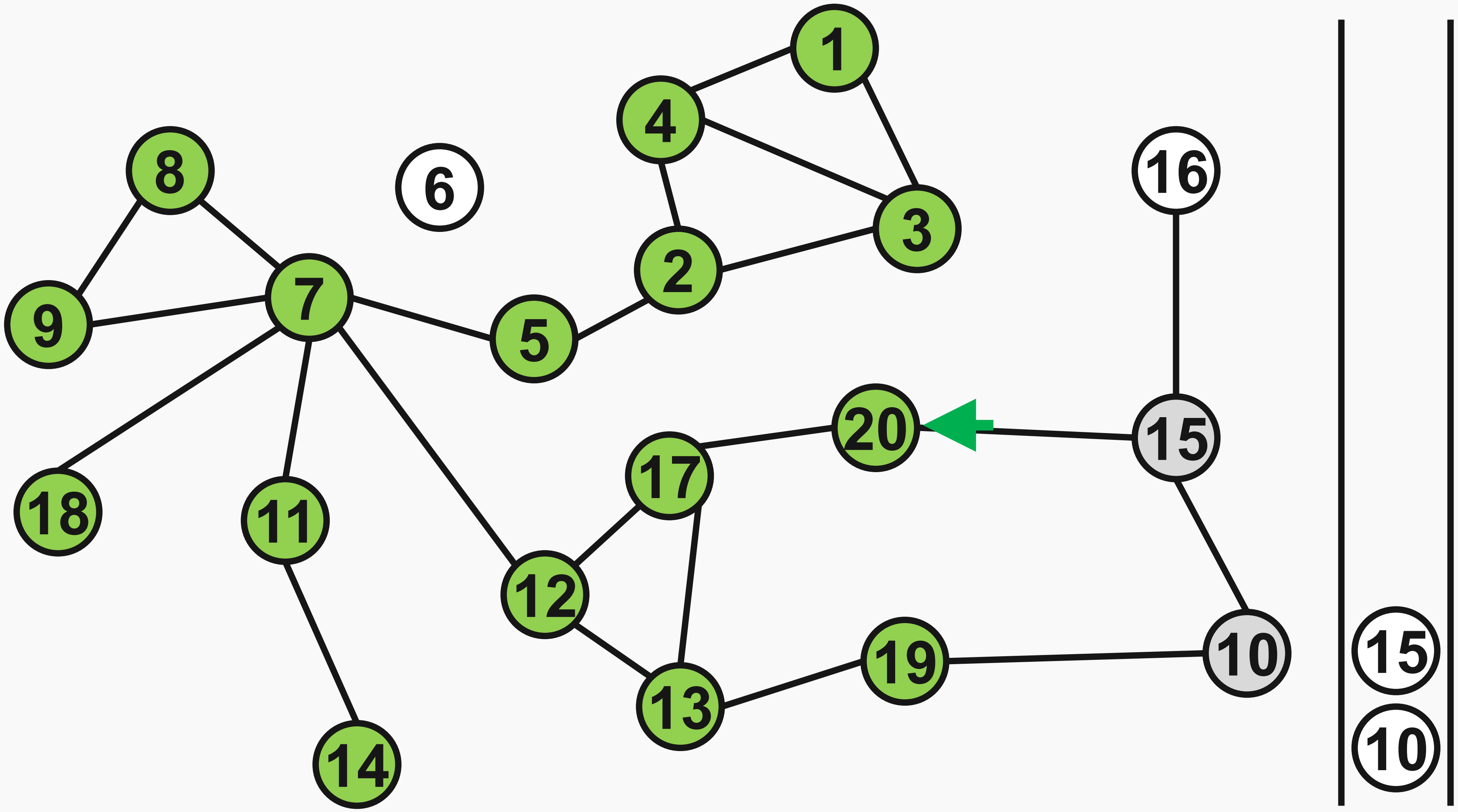
Parcurgere lăţime – breadth first



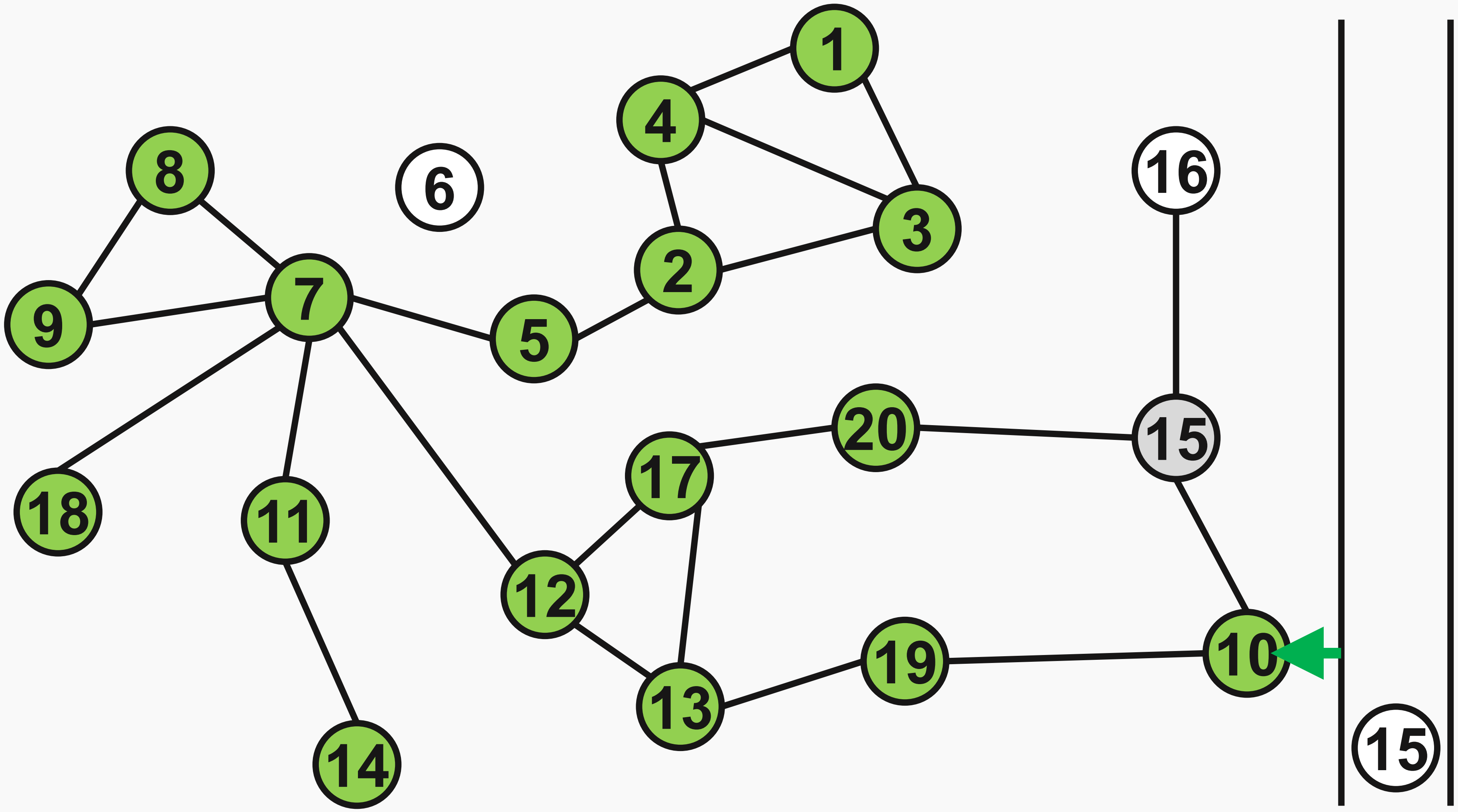
Parcurgere lățime – breadth first



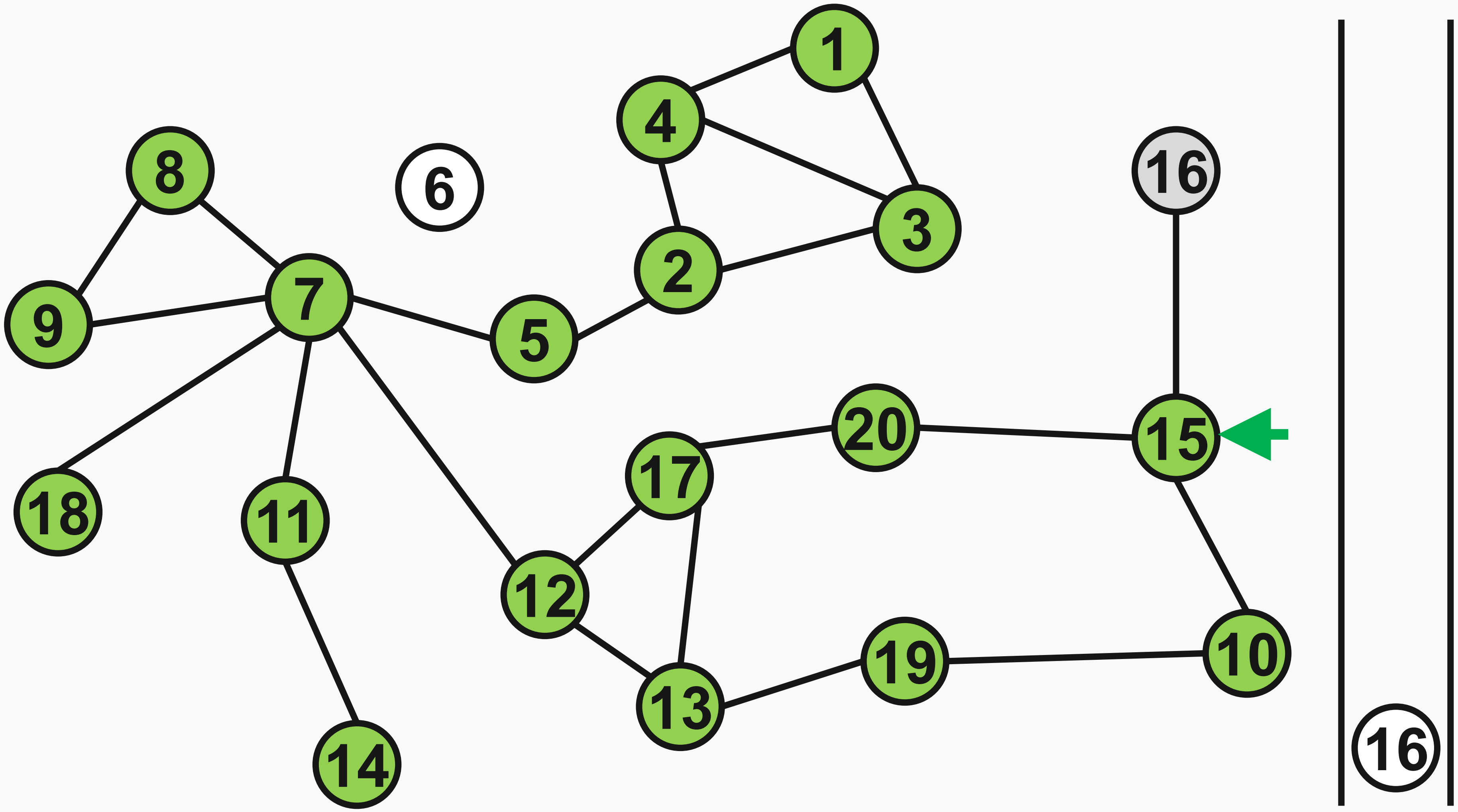
Parcurgere lățime – breadth first



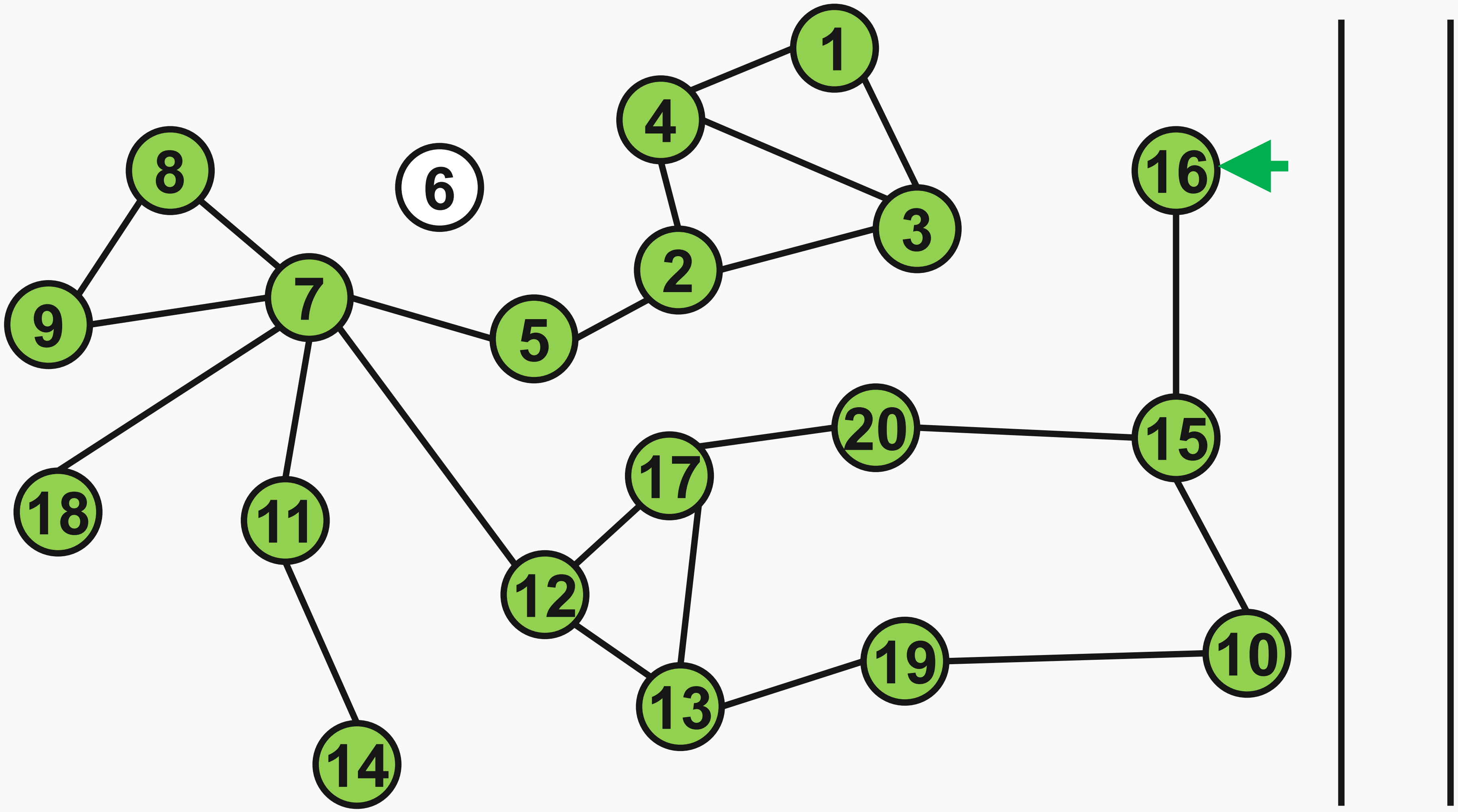
Parcurgere lățime – breadth first



Parcurgere lățime – breadth first



Parcurgere lățime – breadth first



Complexitate parcurgeri?

```
int visited[N] = { 0 };
void BFS(vertex startNode) {
    push(queue, startNode);
    while (!isEmpty(queue)) {
        currentNode = pop(queue);
        visited[currentNode.name] = 1;
        for (int i = 0; i < currentNode.numNeighbors; i++) {
            if (!visited[currentNode.neighbors[i]->name]) {
                push(queue, *(currentNode.neighbors[i]));
                visited[currentNode.neighbors[i]->name] = 2;
            }
        }
    }
}
```

Complexitate parcurgeri?

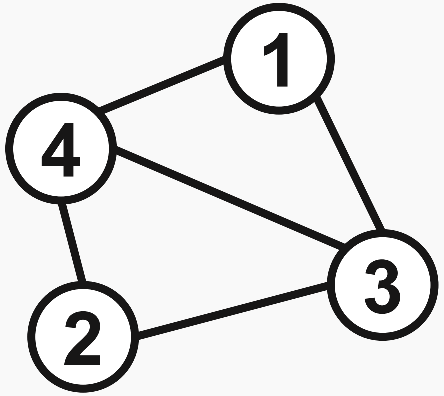
$$O(|V| + |E|)$$

Reprezentarea grafurilor – Listă muchii

Într-o structură de date (vectori, liste) sunt reținute toate muchiile.
$E = \{(1,3),(1,4),(2,3),(2,4),(3,4)\}$

Pentru un graf neorientat putem nota sau nu muchiile în ambele direcții.

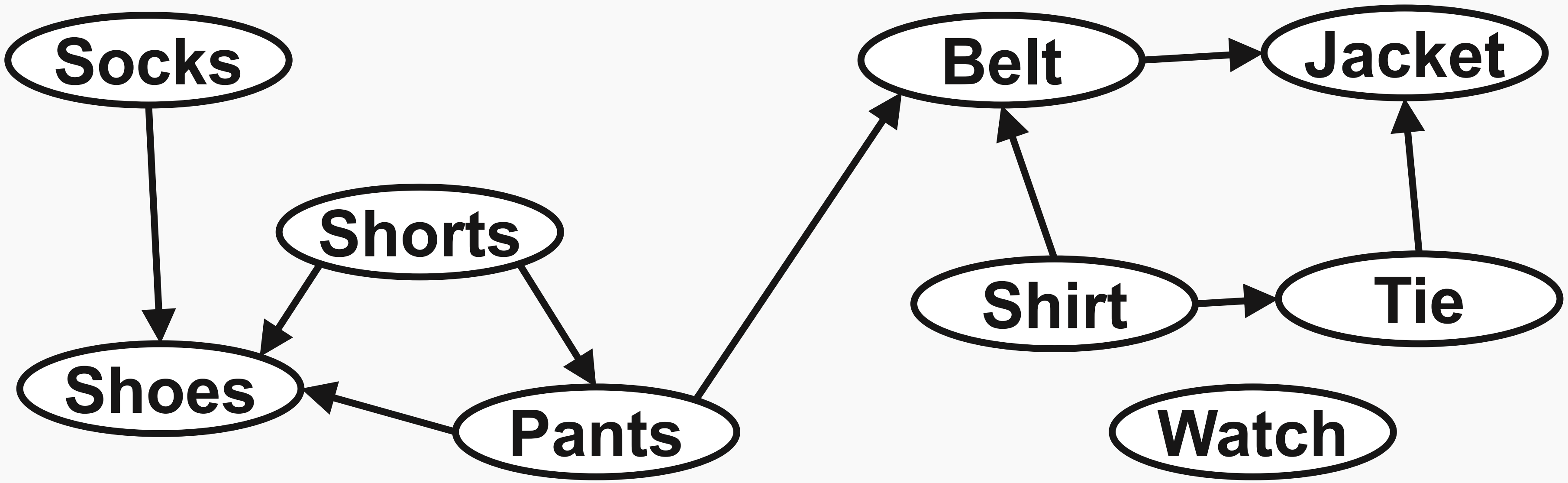
```c
typedef struct edge {
    int nodeA;
    int nodeB;
    int weight;
}edge;

edge edges[5];
```



Sortarea topologică

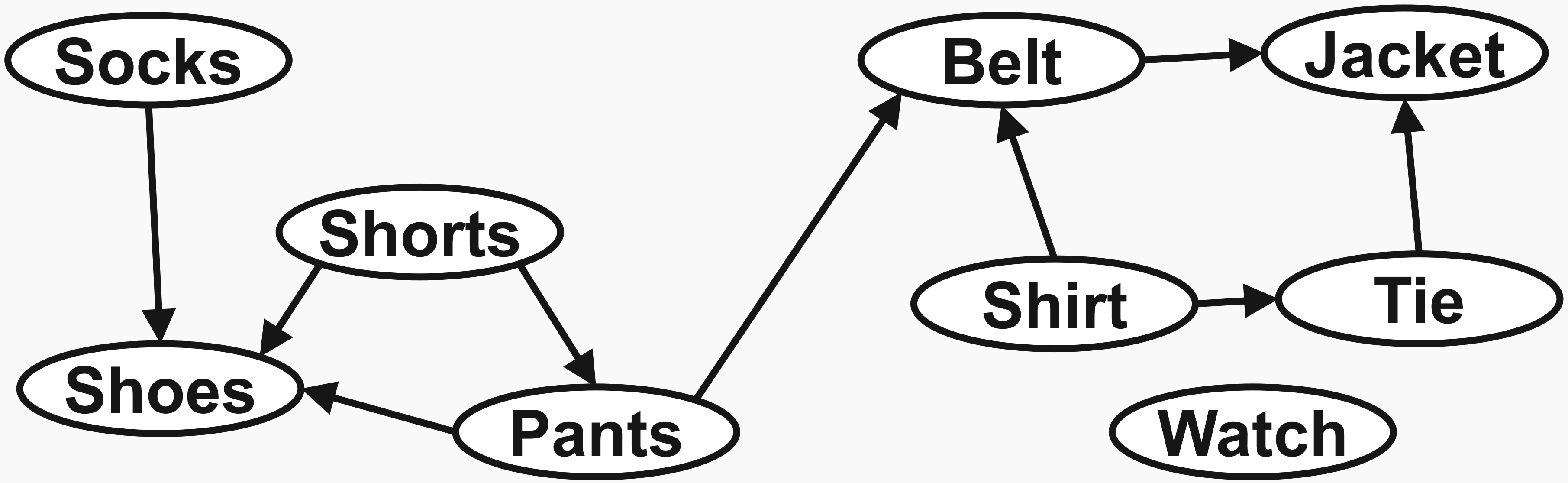
Ordine parțială

- Fie un set de elemente cu o **ordine parțială**.
  - Un element trebuie făcut înaintea altuia.
  - Ex: Un anumit articol de îmbrăcăminte trebuie pus înainte altuia.
- Ordine parțială deoarece între 2 elemente din set poate să nu existe relație de ordine (Watch & orice) (Belt & Shoes)



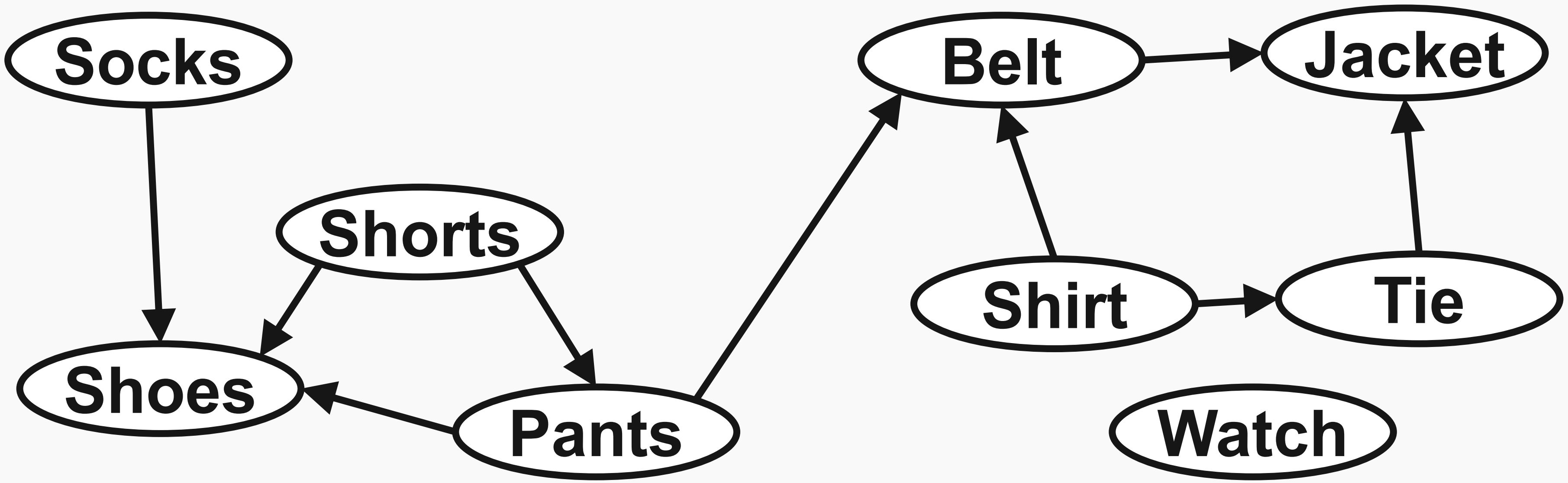
Directed Acyclic Graph

- Ordinea parțială poate fi reprezentată ca un DAG – Directed Acyclic Graph.
  - Absolut necesar să nu avem ciclu în graf, altfel nu vom putea alege cu care element să începem.



Sortarea topologică

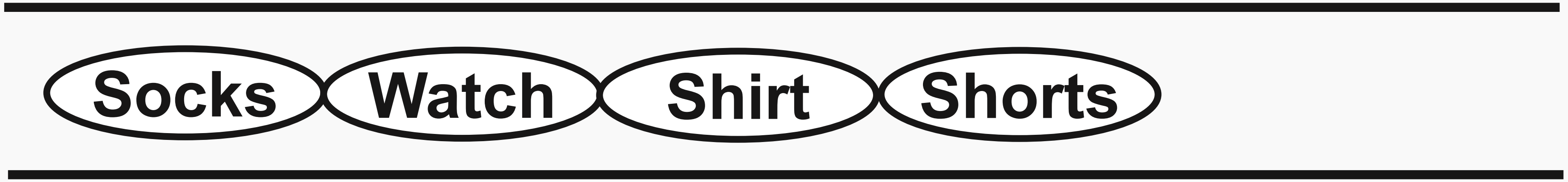
- Procesul de conversie de la un DAG la o listă.
- Se găsește astfel o ordine totală pentru cea parțială
  - Pot exista multiple soluții: Watch poate fi plasat oriunde în listă.

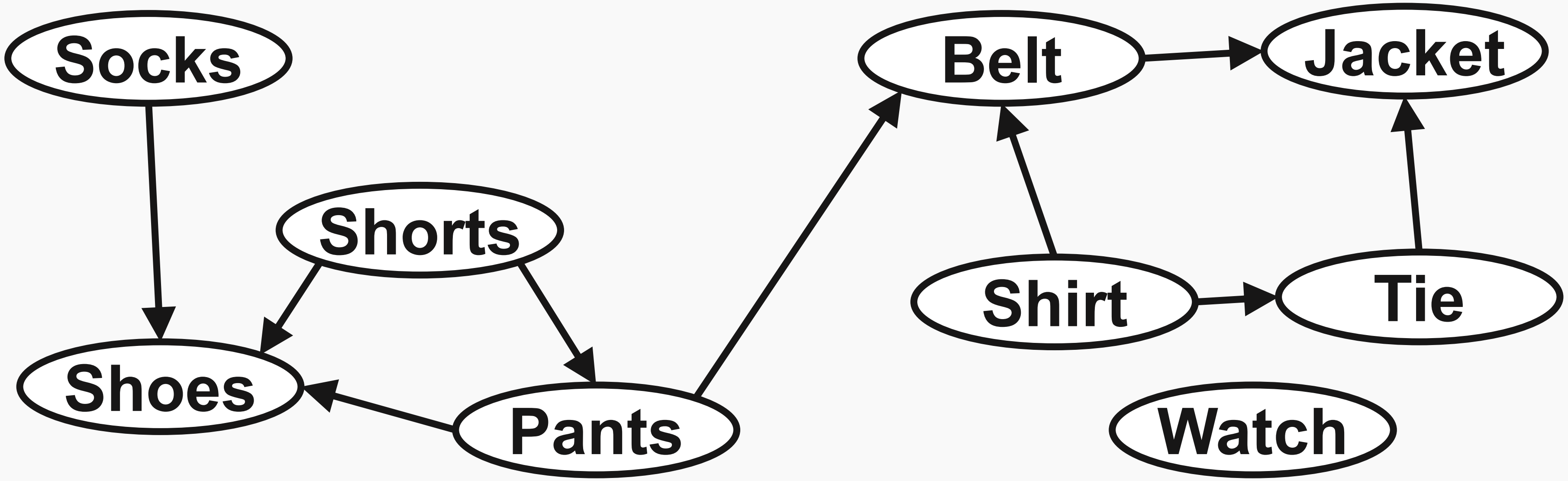


Sortarea topologică – Kahn algorithm (1962)

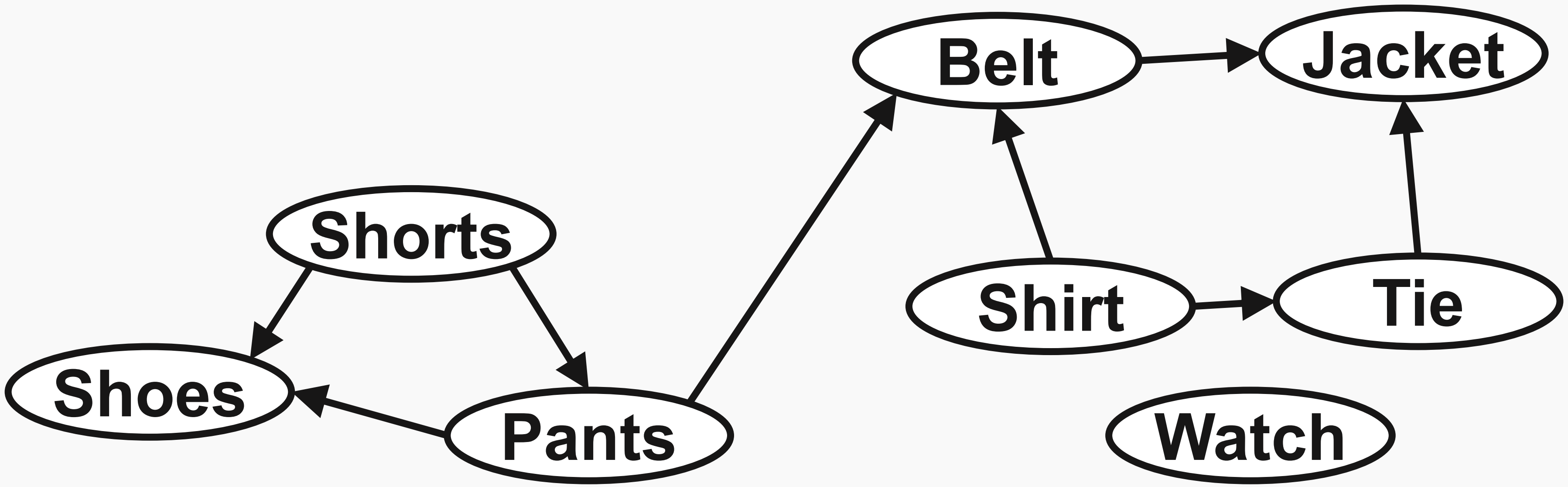
```
push(queue, all nodes with no incoming edge);

while (!isEmpty(queue)) {
    node = pop(queue);
    push(totalSortList, node);
    for each (edge in edges) {
        if(edge[startNode] == node)
            remove(edge);
    }
    push(queue, all nodes with no incoming edge);
}
if (!isEmpty(edges))
    return("graph has cycle");
else
    return totalSortList;
```

Sortarea topologică

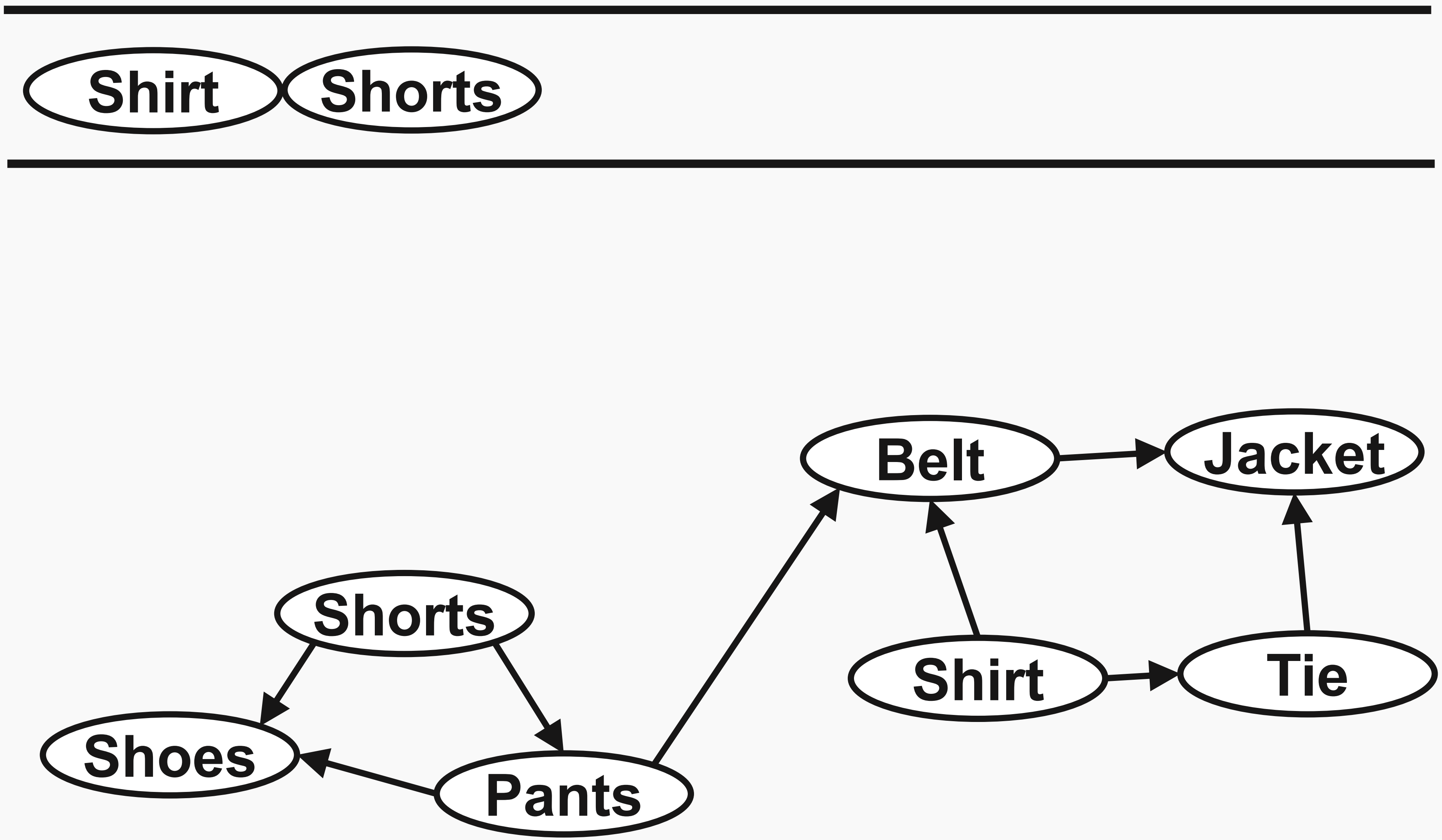




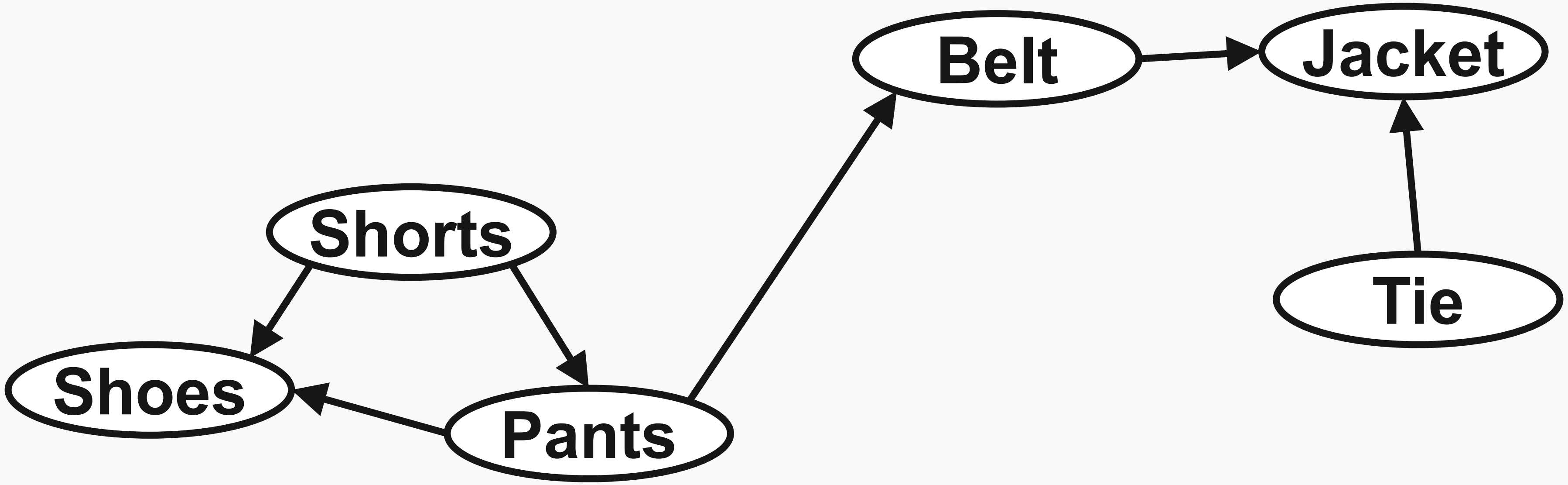
Sortarea topologică



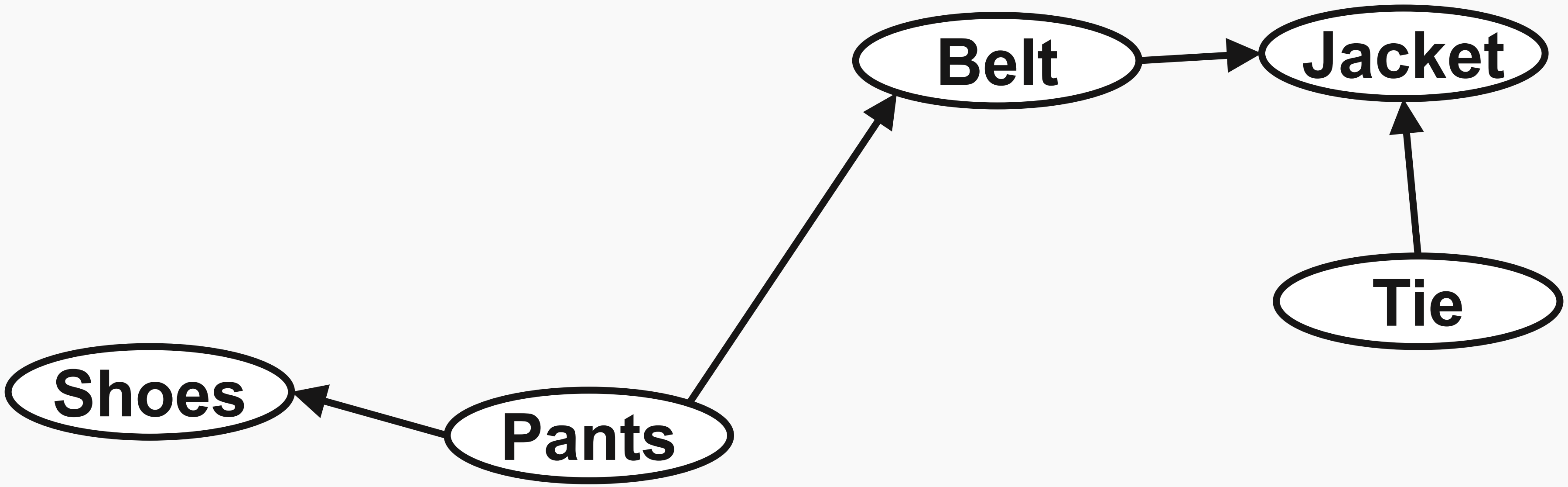
Sortarea topologică



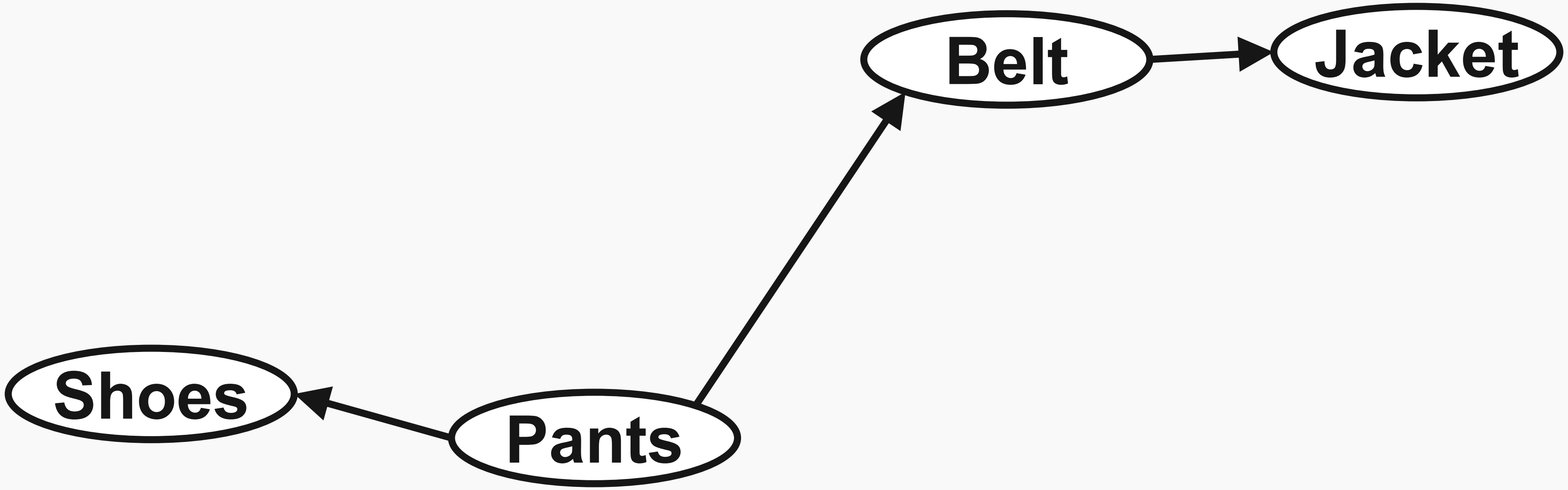
Sortarea topologică



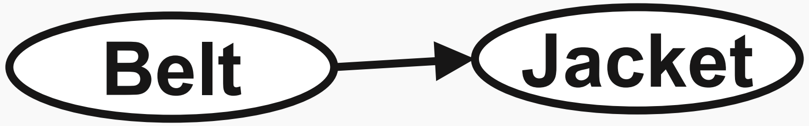
Sortarea topologică



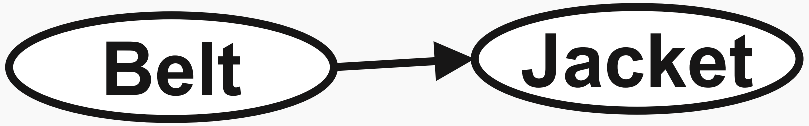
Sortarea topologică



Sortarea topologică



Sortarea topologică



Sortarea topologică

Jacket

Jacket

Sortarea topologică

Jacket

Complexitate?

```
push(queue, all nodes with no incoming edge);

while (!isEmpty(queue)) {
    node = pop(queue);
    push(totalSortList, node);
    for each (edge in edges) {
        if(edge[startNode] == node)
            remove(edge);
    }
    push(queue, all nodes with no incoming edge);
}
if (!isEmpty(edges))
    return("graph has cycle");
else
    return totalSortList;
```

Complexitate?

$$\boldsymbol{O}(|\boldsymbol{V}| + |\boldsymbol{E}|)$$