

Graphs and Graph Algorithms

Fundamentals, Terminology, Traversal and Algorithms

SoftUni Team
Technical Trainers



SoftUni



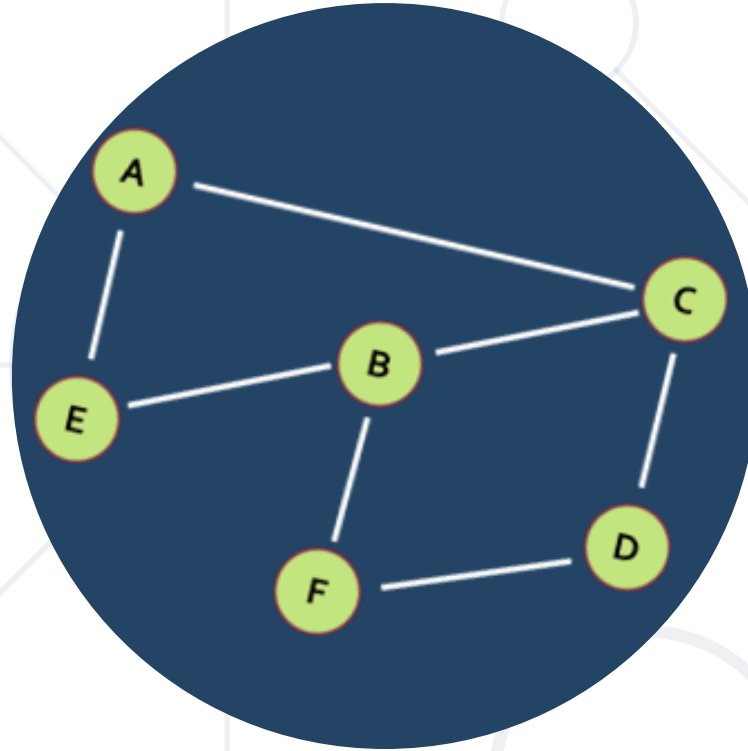
Software University

<https://about.softuni.bg>

Table of Contents

1. Graph Definitions and Terminology
2. Representing Graphs
3. Graph Traversal Algorithms
 - Depth-First-Search (DFS)
 - Breadth-First-Search (BFS)
4. Connected Components
5. Topological Sorting
 - Source Removal and DFS Algorithms
6. Shortest Path in Unweighted Graph

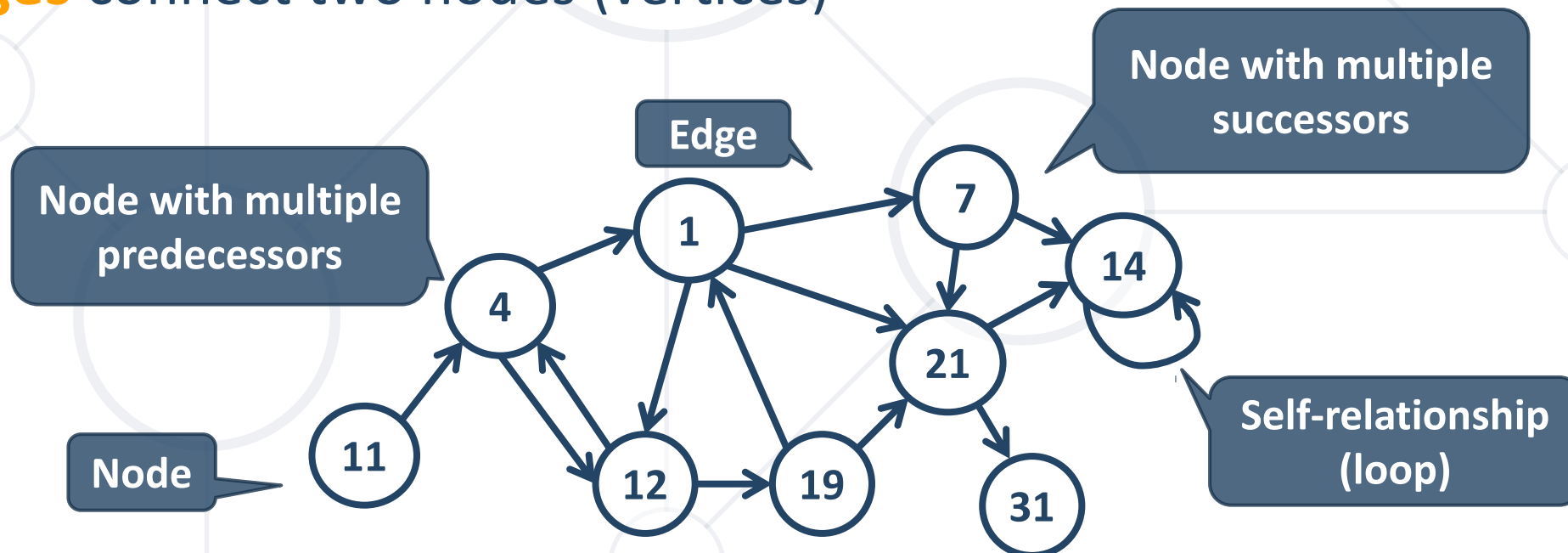




Graphs

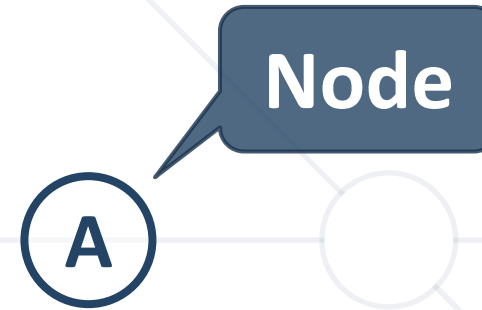
Definitions and Terminology

- **Graph**, denoted as **$G(V, E)$**
 - Set of nodes **V** with many-to-many relationship between them (edges **E**)
 - Each **node (vertex)** has **multiple** predecessors and **multiple** successors
 - **Edges** connect two nodes (vertices)



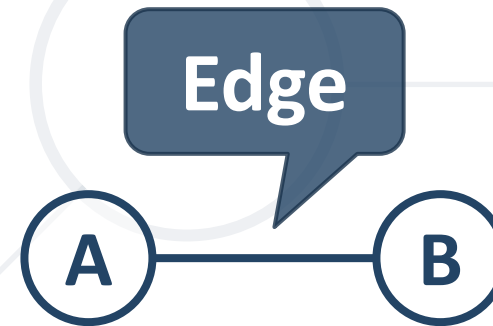
- **Node** (vertex)

- Element of a graph
- Can have name / value
- Keeps a list of adjacent nodes



- **Edge**

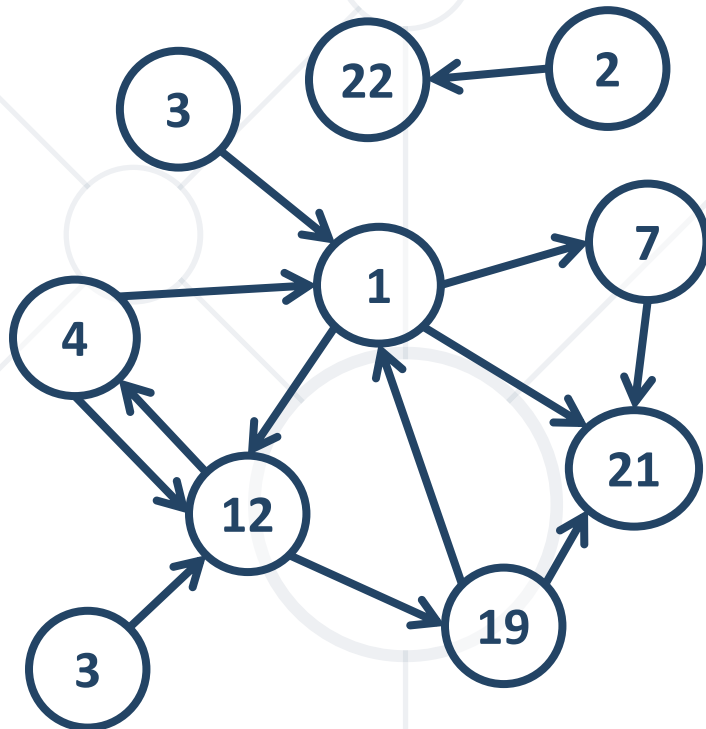
- Connection between two nodes
- Can be directed / undirected
- Can be weighted / unweighted
- Can have name / value



Graph Definitions (2)

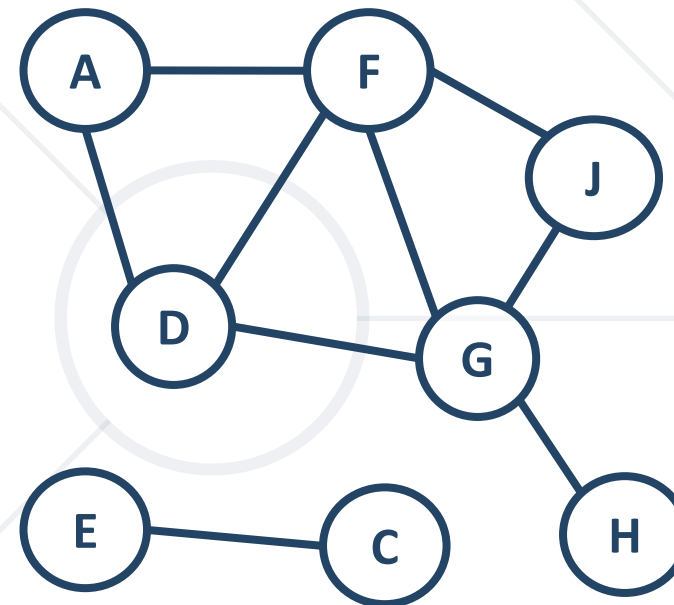
- **Directed graph**

- Edges have direction

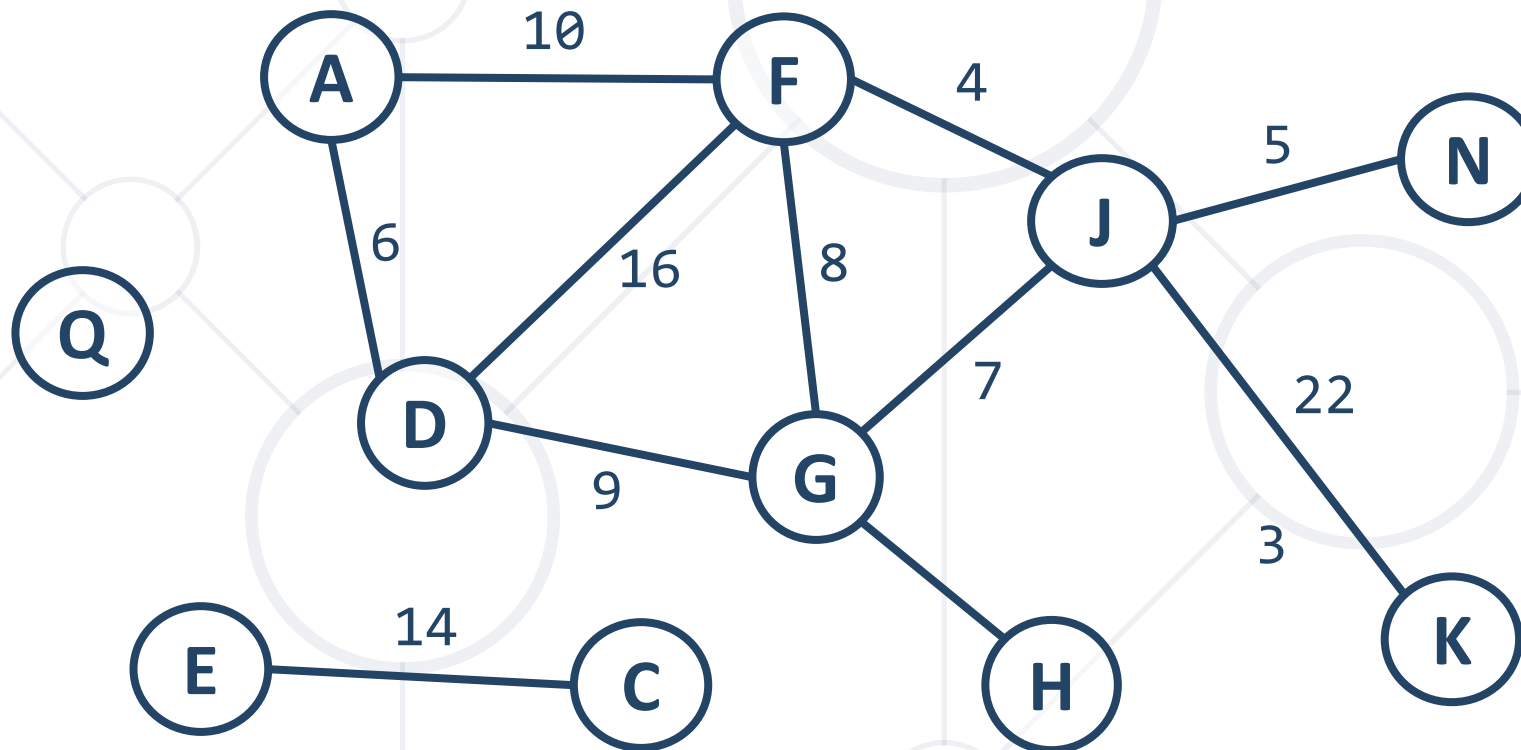


- **Undirected graph**

- Undirected edges



- **Weighted graph**
 - Weight (cost) is associated with each edge

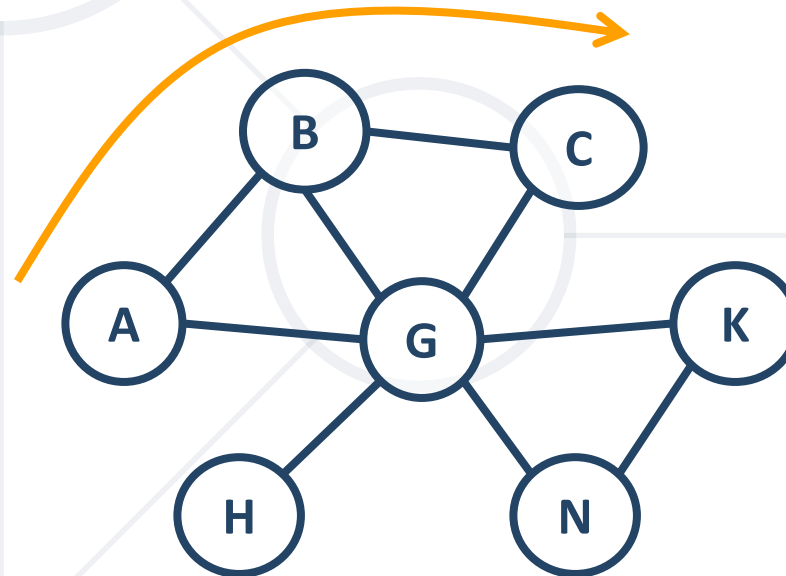


- **Path** (in undirected graph)

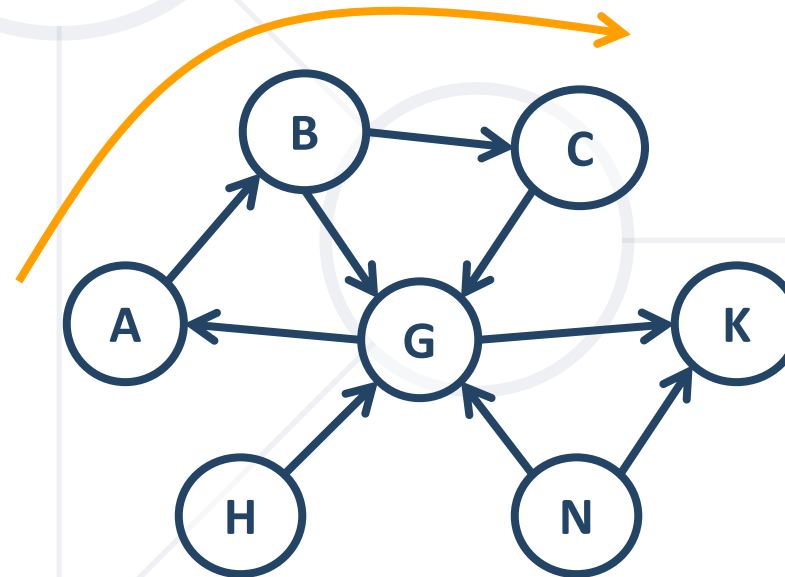
- Sequence of nodes n_1, n_2, \dots, n_k
- Edge exists between each pair of nodes n_i, n_{i+1}

- Examples:

- A, B, C is a path
- A, B, G, N, K is a path
- H, K, C is not a path
- H, G, B, G, N is a path



- **Path** (in directed graph)
 - Sequence of nodes n_1, n_2, \dots, n_k
 - Directed edge exists between each pair of nodes n_i, n_{i+1}
 - Examples:
 - A, B, C is a path
 - N, G, A, B, C is a path
 - A, G, K is not a path
 - H, G, K, N is not a path



- **Cycle**

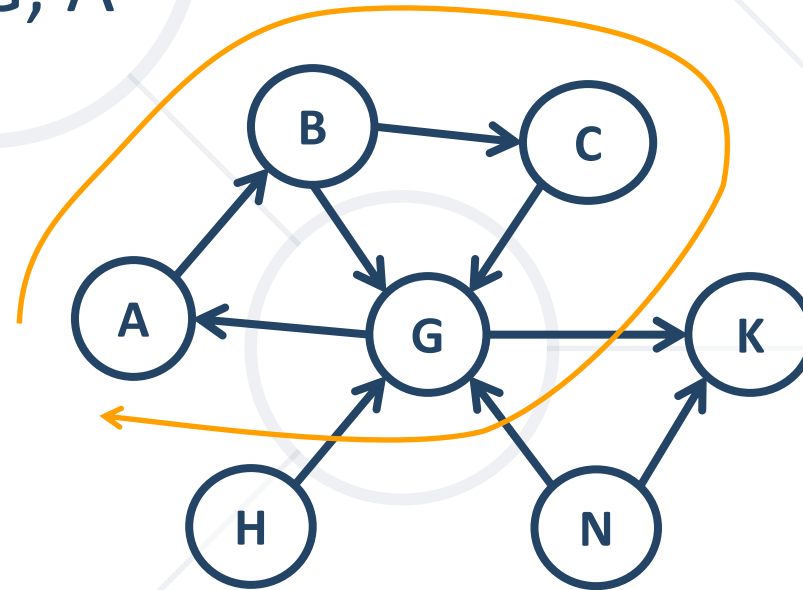
- Path that ends back at the starting node
- Example of cycle: A, B, C, G, A

- **Simple path**

- No cycles in path

- **Acyclic graph**

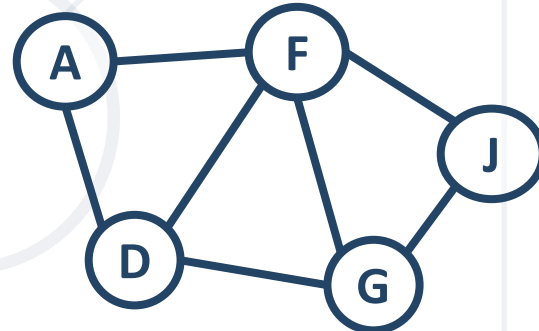
- Graph with no cycles
- Acyclic undirected graphs are trees



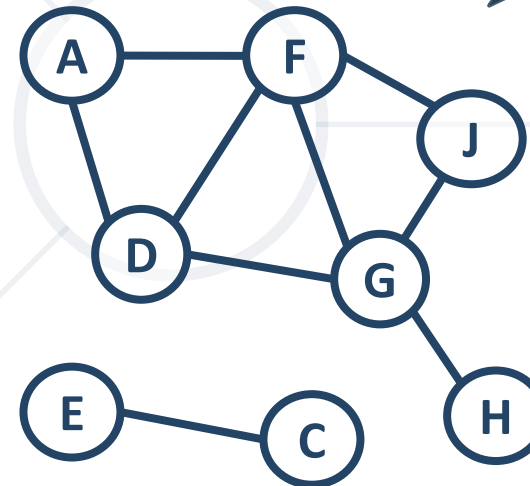
Graph Definitions (7)

- Two nodes are **reachable** if a path exists between them
- Connected graph**
 - Every two nodes are reachable from each other

Connected
graph

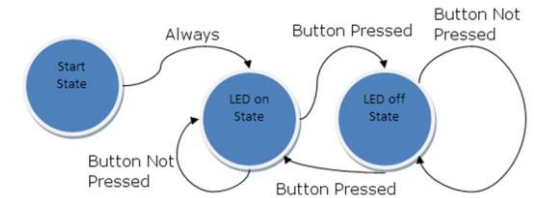


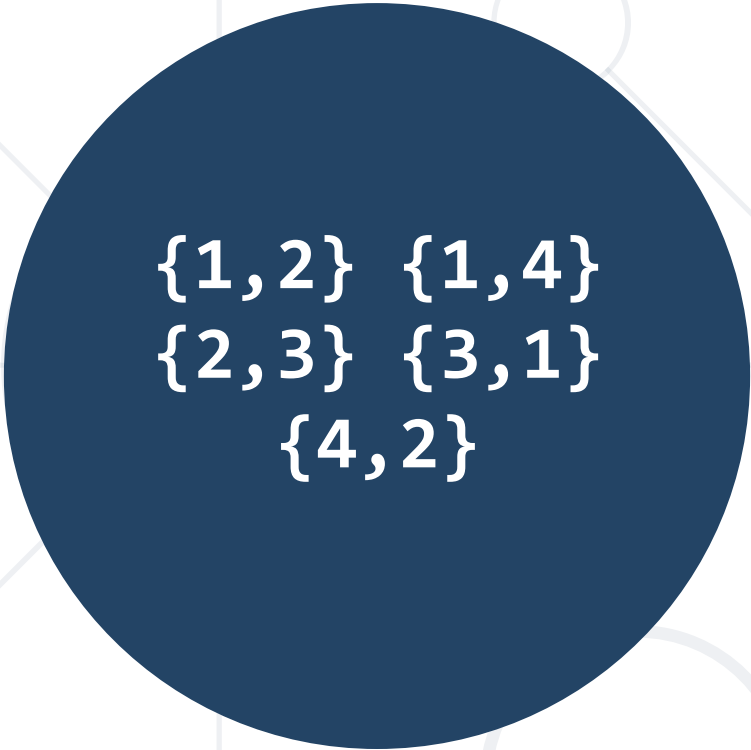
Unconnected graph
holding two connected
components



Graphs and Their Applications

- Graphs have many real-world applications
 - Modeling a **computer network** like the Internet
 - Routes are simple paths in the network
 - Modeling a city **map**
 - Streets are edges, crossings are vertices
 - **Social networks**
 - People are nodes and their connections are edges
 - **State machines**
 - States are nodes, transitions are edges





$\{1, 2\}$ $\{1, 4\}$
 $\{2, 3\}$ $\{3, 1\}$
 $\{4, 2\}$

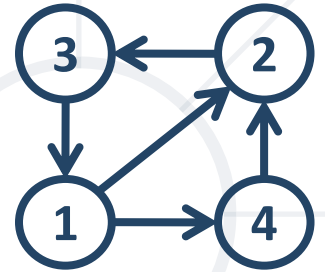
Representing Graphs

Classic and OOP Ways

- **Adjacency list**

- Each node holds a list of its neighbors

```
1 -> {2, 4}
2 -> {3}
3 -> {1}
4 -> {2}
```



- **Adjacency matrix**

- Each cell keeps whether and how two nodes are connected

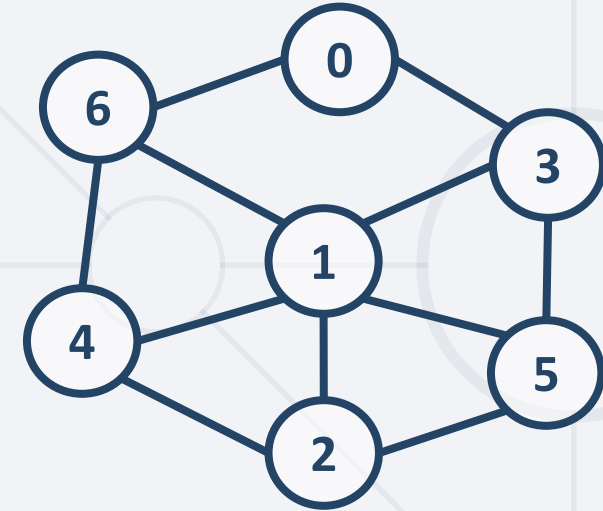
	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	1	0	0	0
4	0	1	0	0

- **List of edges**

{1,2}, {1,4}, {2,3}, {3,1}, {4,2}

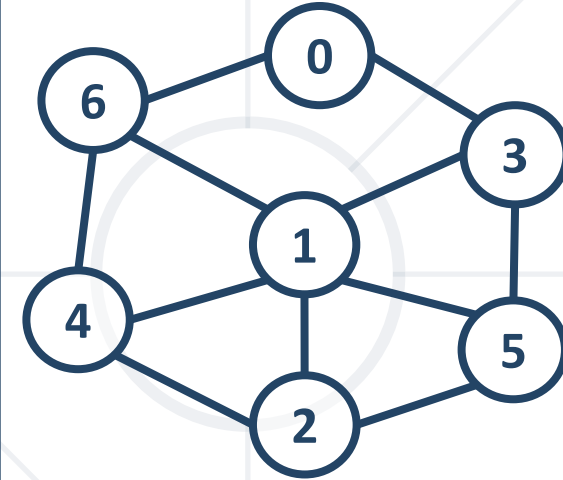
Graph Representation: Adjacency List

```
var g = new List<int>[] {  
    new List<int> { 3, 6},  
    new List<int> { 2, 3, 4, 5, 6},  
    new List<int> { 1, 4, 5},  
    new List<int> { 0, 1, 5},  
    new List<int> { 1, 2, 6},  
    new List<int> { 1, 2, 3},  
    new List<int> { 0, 1, 4}  
};  
  
// Add an edge { 3 <-> 6 }  
g[3].Add(6); g[6].Add(3);  
  
// List the children of node #1  
var childNodes = g[1];
```



Graph Representation: Adjacency Matrix

```
int[][] graph = new int[][] {  
    // 0 1 2 3 4 5 6  
    { 0, 0, 0, 1, 0, 0, 1 }, // node 0  
    { 0, 0, 1, 1, 1, 1, 1 }, // node 1  
    { 0, 1, 0, 0, 1, 1, 0 }, // node 2  
    { 1, 1, 0, 0, 0, 1, 0 }, // node 3  
    { 0, 1, 1, 0, 0, 0, 1 }, // node 4  
    { 0, 1, 1, 1, 0, 0, 0 }, // node 5  
    { 1, 1, 0, 0, 1, 0, 0 }, // node 6  
};  
// Add an edge { 3 -> 6 }  
graph[3][6] = 1;  
// List the children of node #1  
int[] childNodes = graph[1];
```



Graph Representation: List of Edges

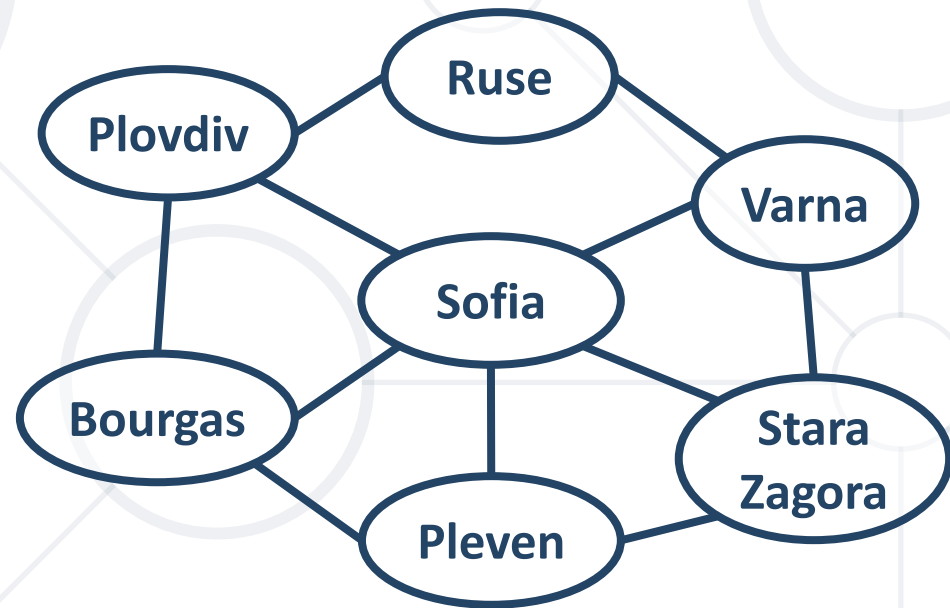
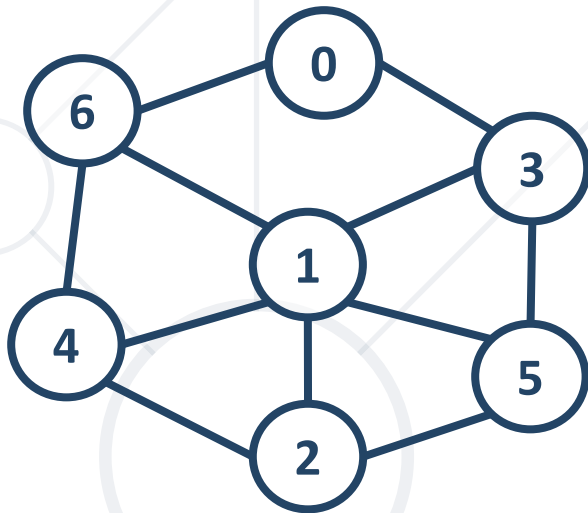
```
class Edge
{
    public int Parent {get; set; }
    public int Child {get; set; }
}

var graph = new List<Edge>() {
    new Edge() { Parent = 0, Child = 3 },
    new Edge() { Parent = 0, Child = 6 }, ...
}

// Add an edge { 3 -> 6 }
graph.Add(new Edge() { Parent = 3, Child = 6 });
// List the children of node #1
var childNodes = graph.Where(e => e.Parent == 1);
```

Numbering Graph Nodes

- A common technique to **speed up** working with graphs
 - **Numbering the nodes** and accessing them by index (not by name)



Graph of **numbered nodes**: [0...6]

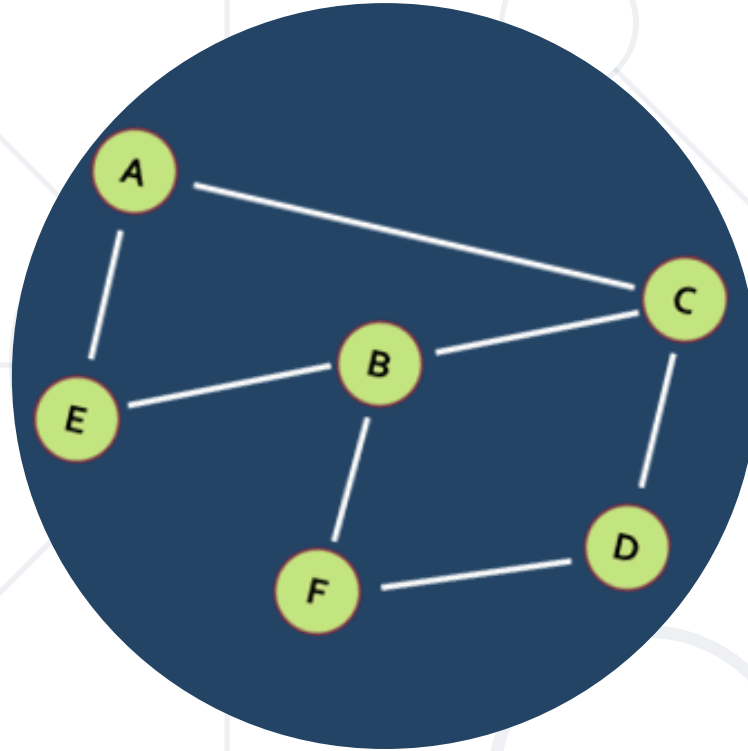
Graph of **named nodes**

Numbering Graph Nodes – How?

- Suppose we have a **graph of n nodes**
 - We can assign a number for each node in the range $[0 \dots n-1]$

#	Node
0	Ruse
1	Sofia
2	Pleven
3	Varna
4	Bourgas
5	Stara Zagora
6	Plovdiv

- Using OOP:
 - Class **Node**
 - Class **Edge (Connection)**
 - Class **Graph**
 - Optional classes
 - Algorithm classes



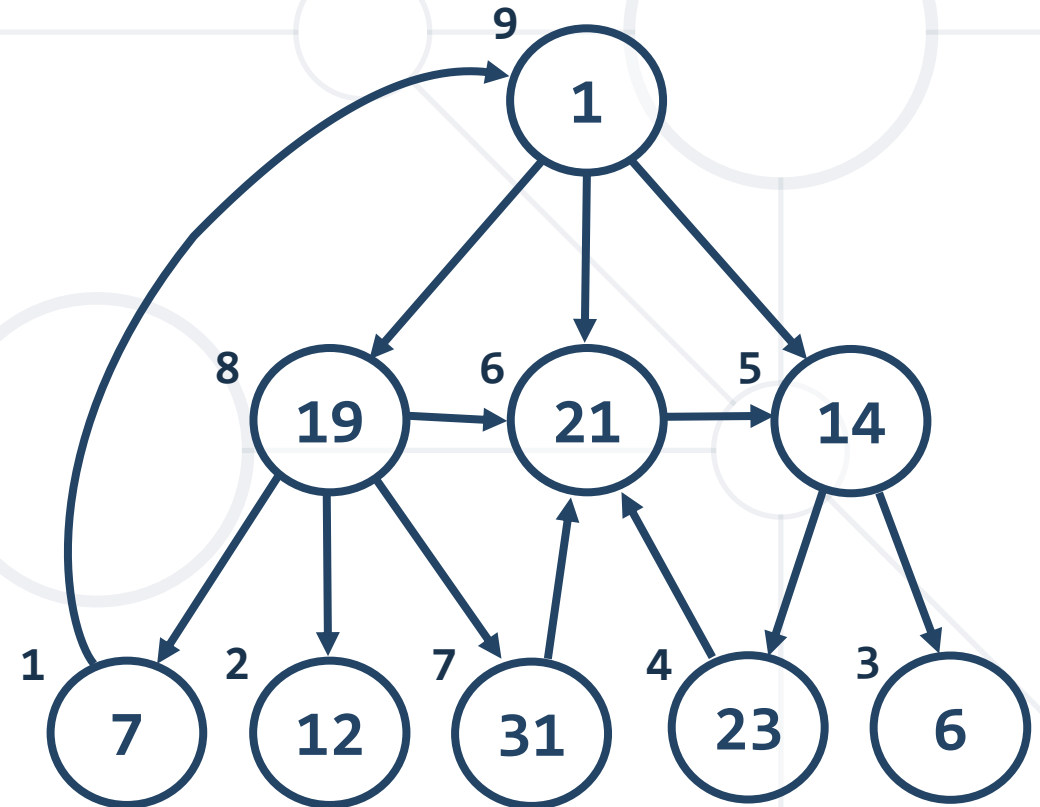
Graphs Traversals

Depth-First Search and Breadth-First Search

- **Traversing a graph** means to visit each of its nodes exactly once
 - The order of visiting nodes may vary on the traversal algorithm
 - **Depth-First Search** (DFS)
 - Visit node's successors first
 - Usually implemented by **recursion**
 - **Breadth-First Search** (BFS)
 - Nearest nodes visited first
 - Implemented with a **queue**

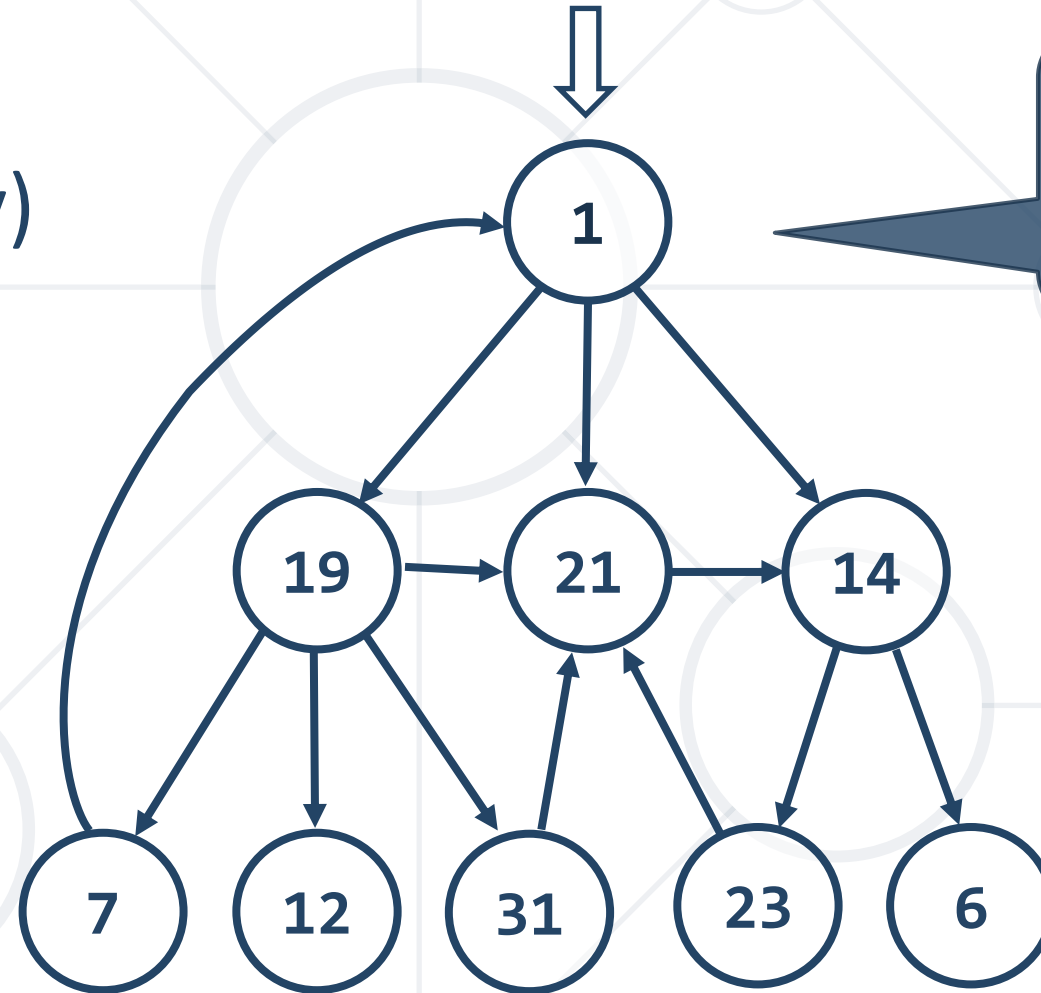
- **Depth-First Search (DFS)** first visits all descendants of given node recursively, finally visits the node itself

```
visited[0 ... n-1] = false;
for (v = 0 ... n-1) dfs(v)
dfs (node) {
    if not visited[node] {
        visited[node] = true;
        for each child c of node
            dfs(c);
        print node;
    }
}
```



DFS in Action (Step 1)

- Stack: 1
- Output: (empty)

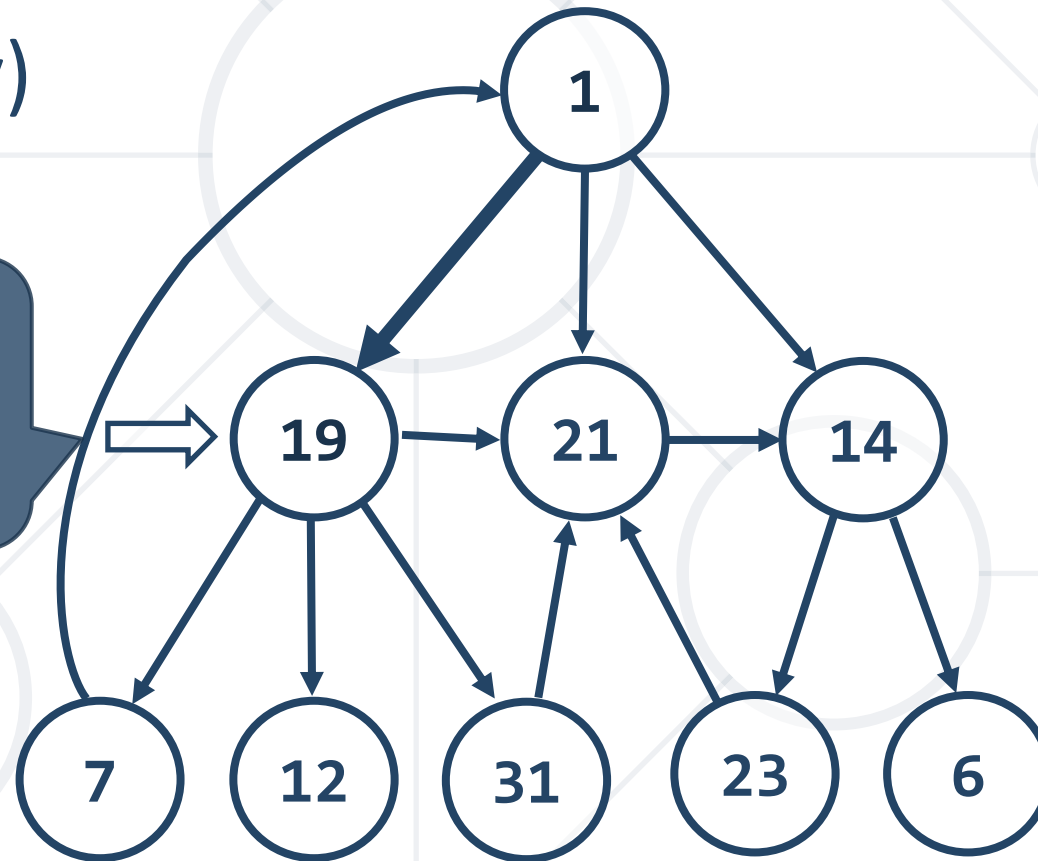


Start DFS from
the initial node **1**

DFS in Action (Step 2)

- Stack: 1, 19
- Output: (empty)

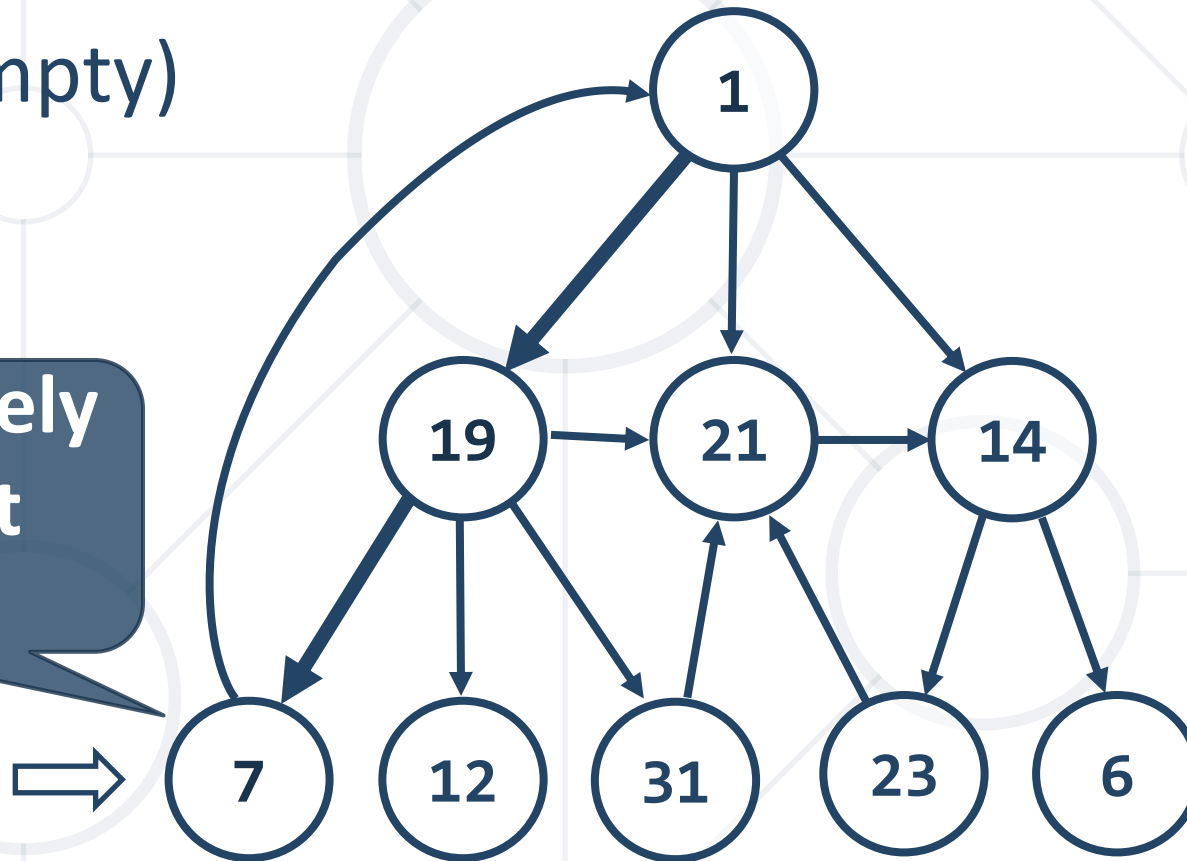
Enter recursively
into the first
child **19**



DFS in Action (Step 3)

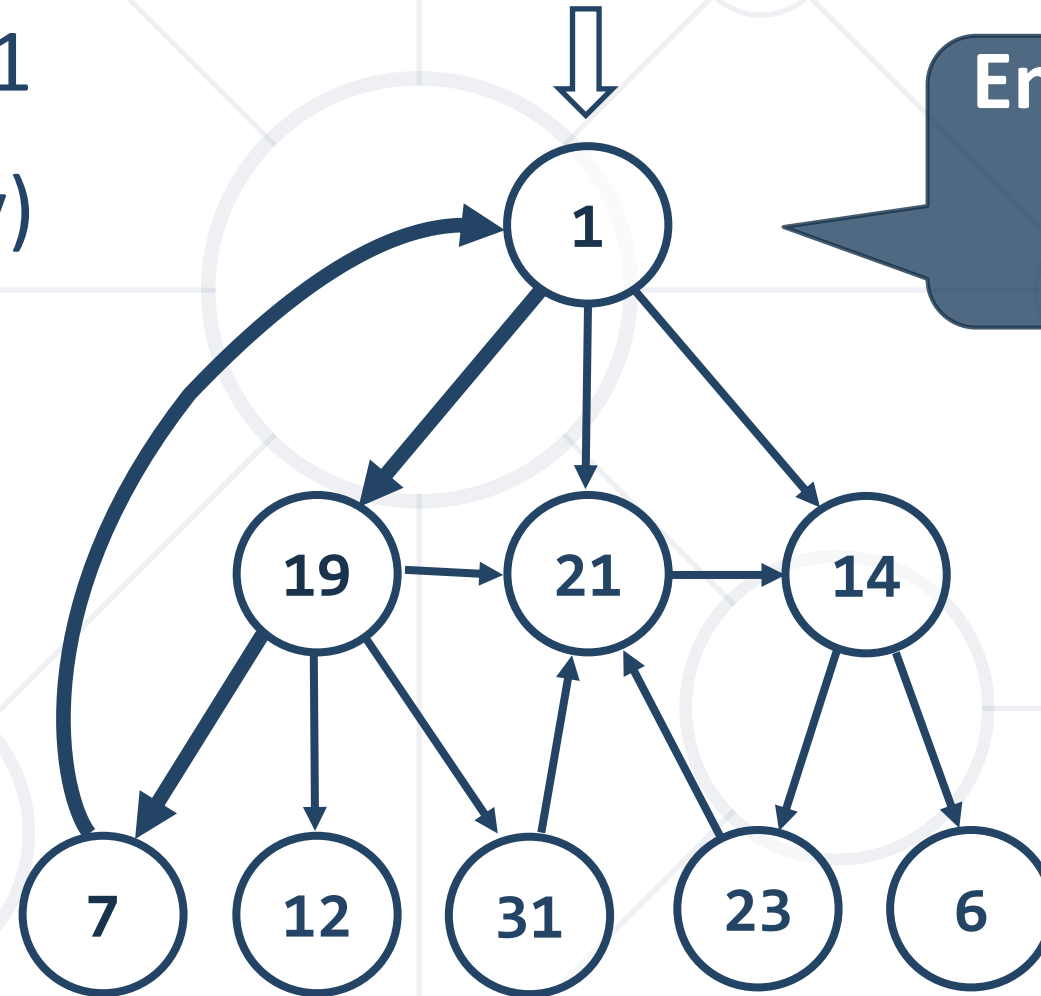
- Stack: 1, 19, 7
- Output: (empty)

Enter recursively
into the first
child **7**



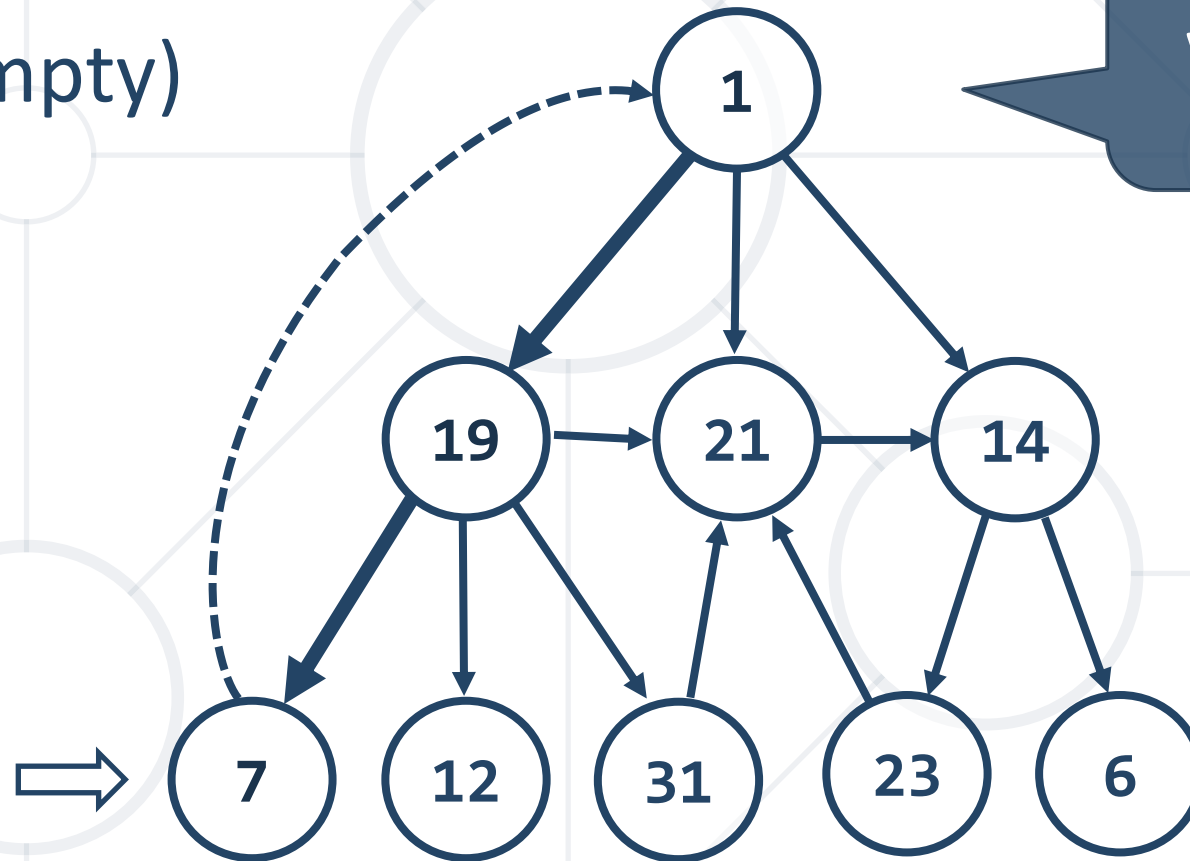
DFS in Action (Step 4)

- Stack: 1, 19, 7, 1
- Output: (empty)



DFS in Action (Step 5)

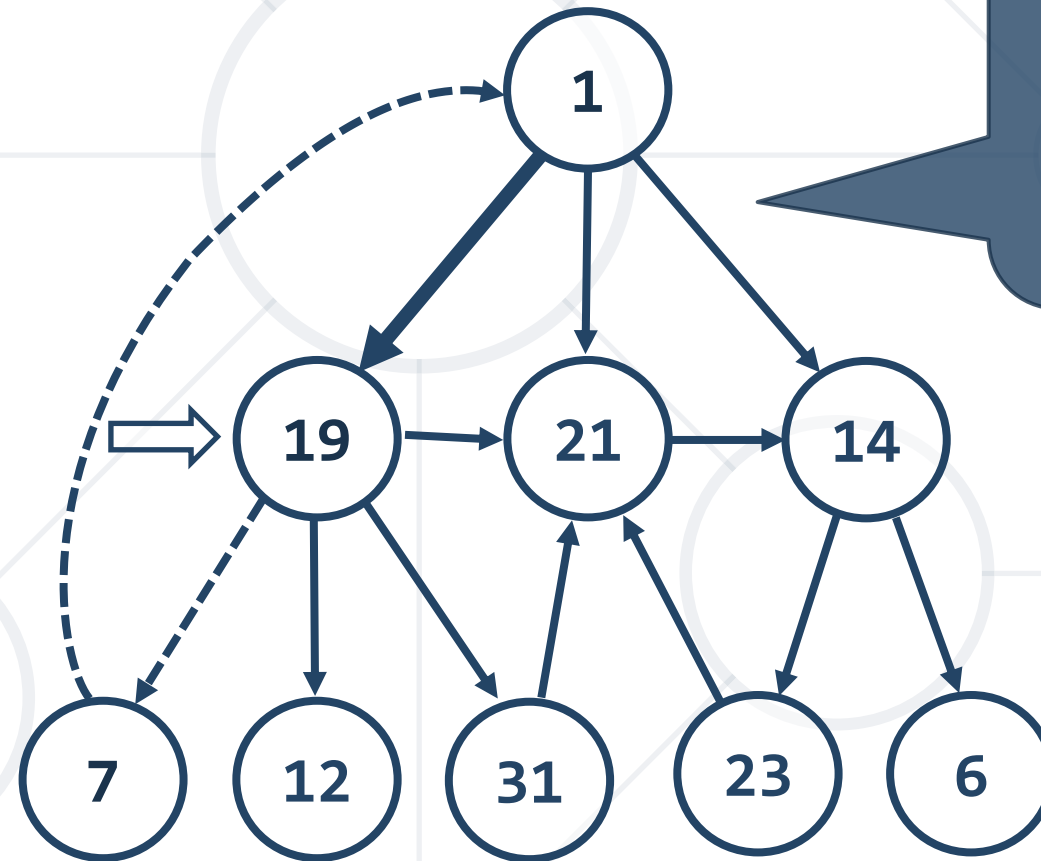
- Stack: 1, 19, 7
- Output: (empty)



Node **1** already
visited - return
back

DFS in Action (Step 6)

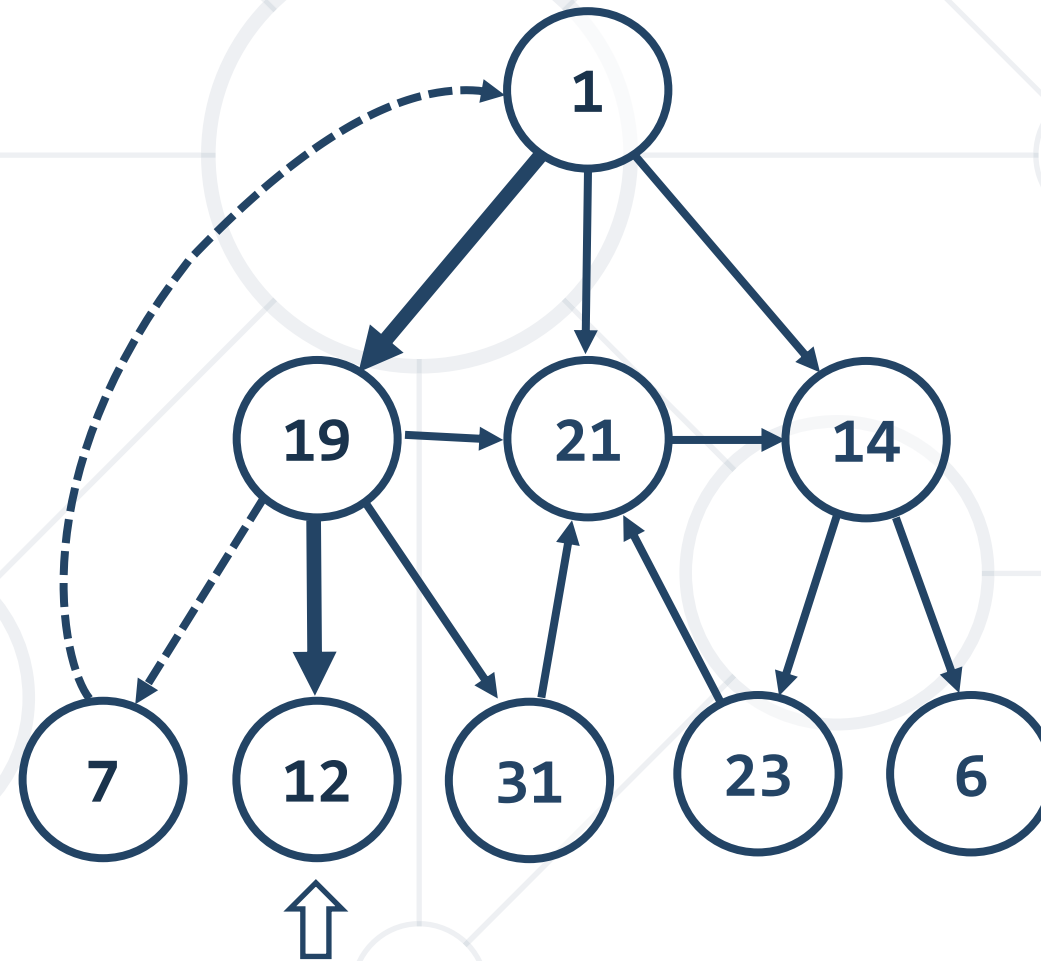
- Stack: 1, 19
- Output: 7



Return back from
recursion and print
the last visited
node - 7

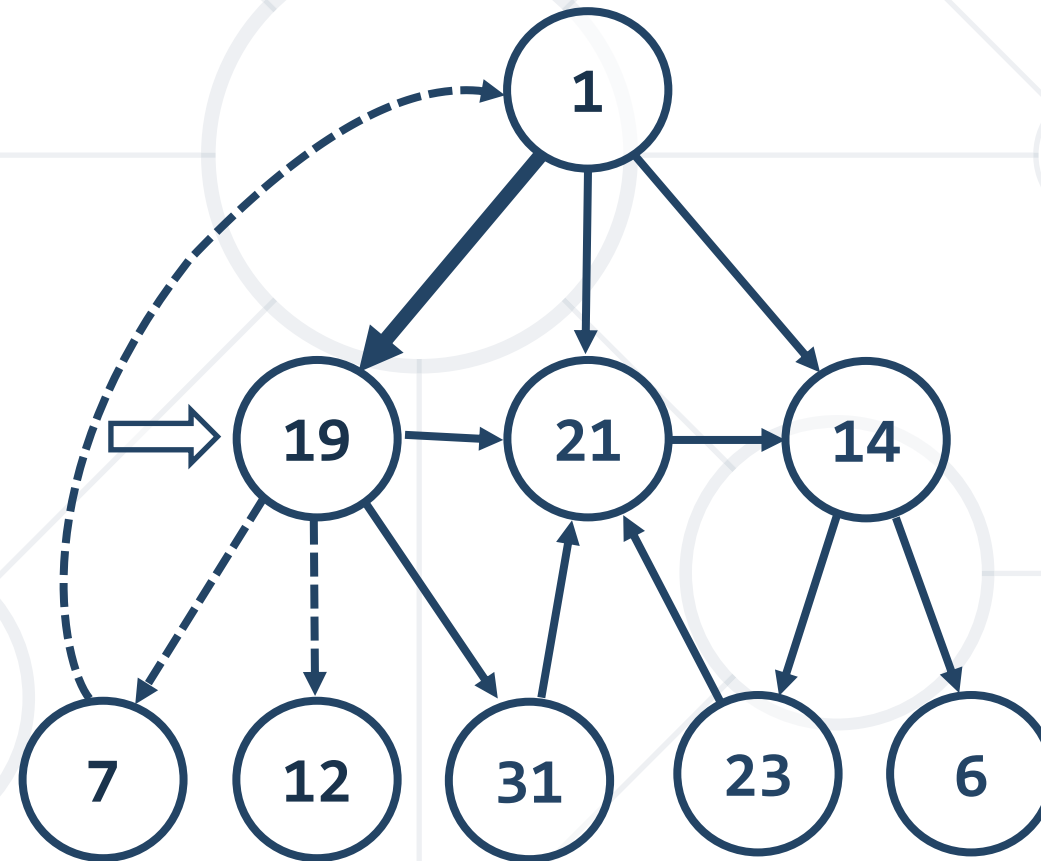
DFS in Action (Step 7)

- Stack: 1, 19, 12
- Output: 7



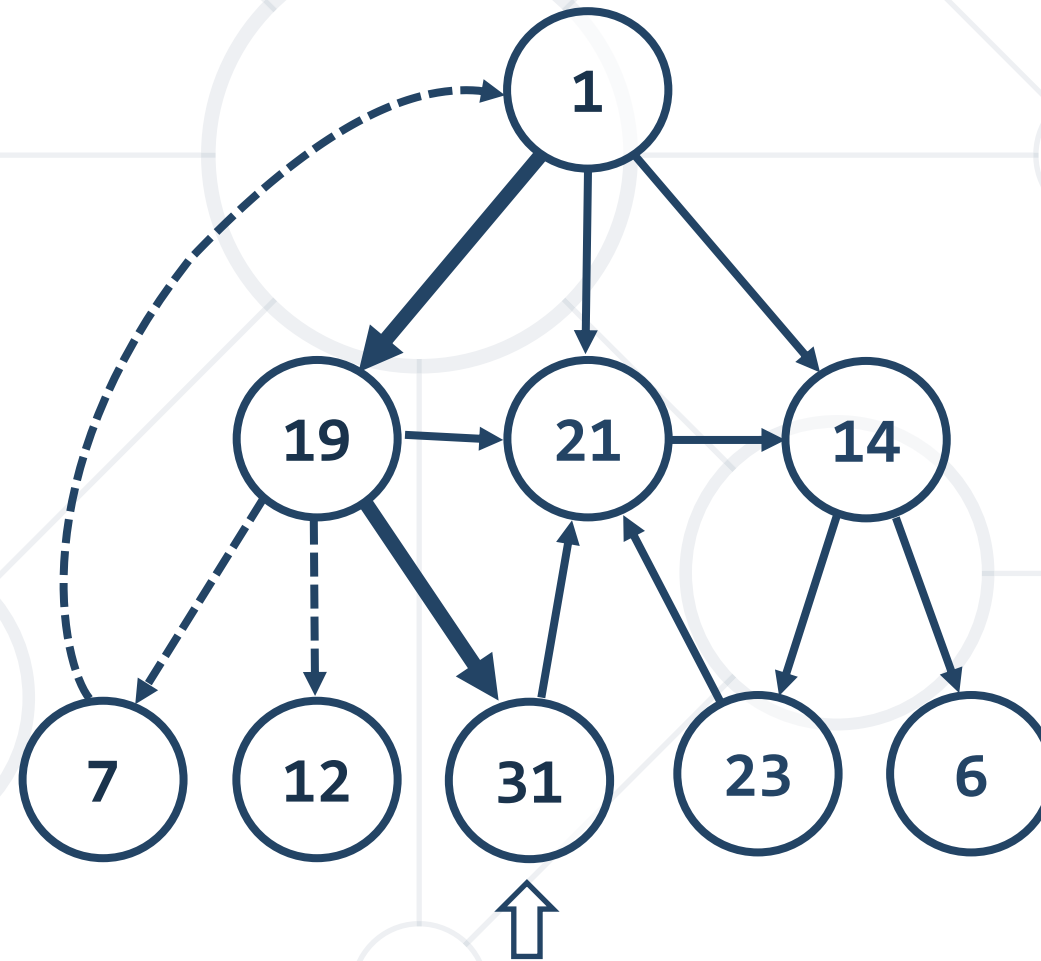
DFS in Action (Step 8)

- Stack: 1, 19
- Output: 7, 12



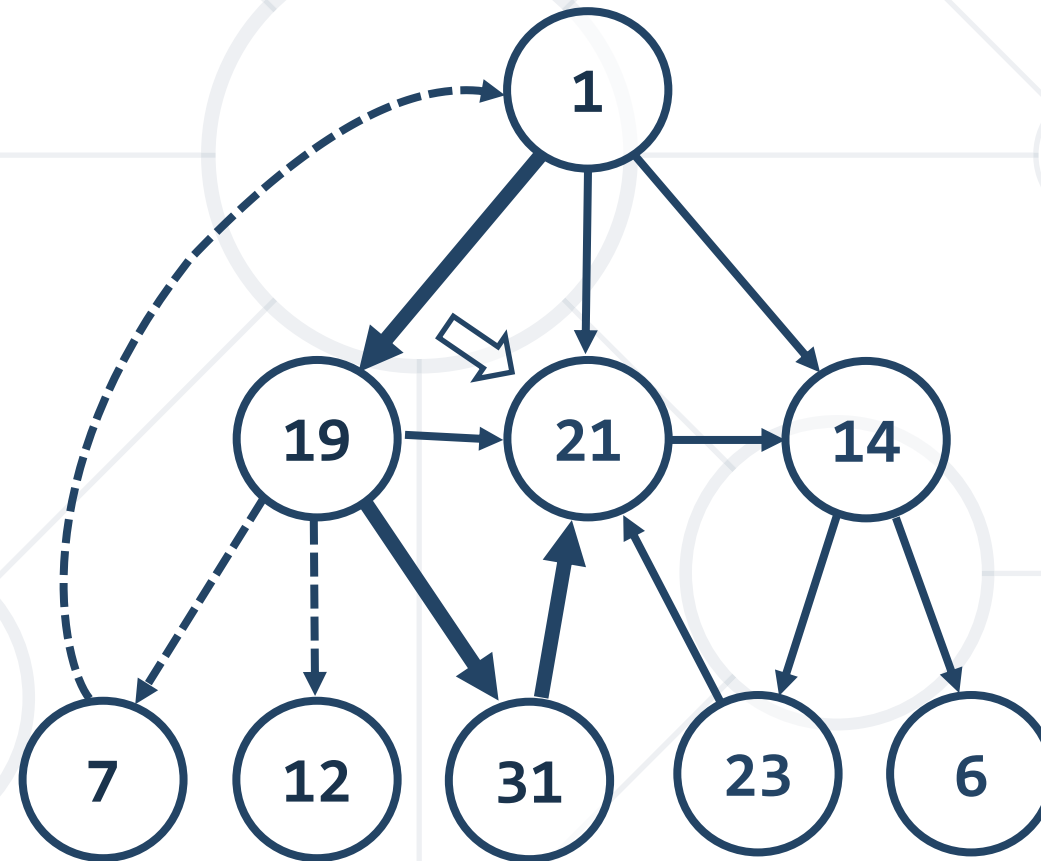
DFS in Action (Step 9)

- Stack: 1, 19, 31
- Output: 7, 12



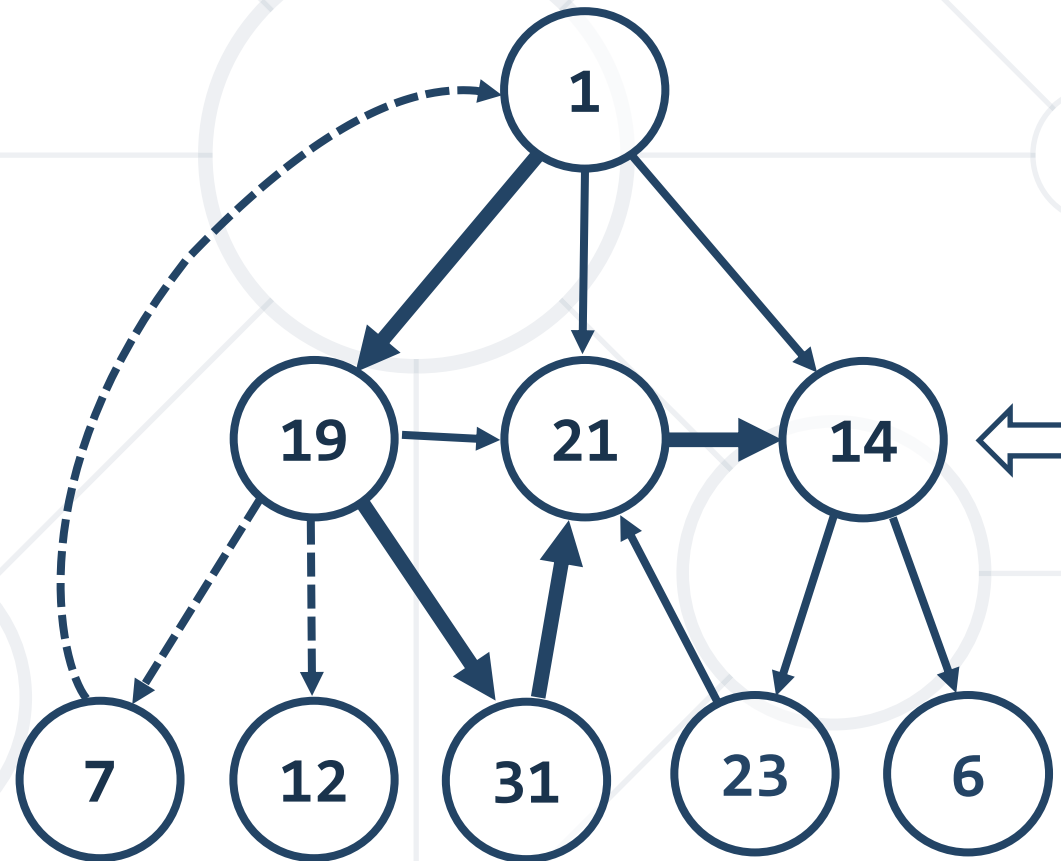
DFS in Action (Step 10)

- Stack: 1, 19, 31, 21
- Output: 7, 12



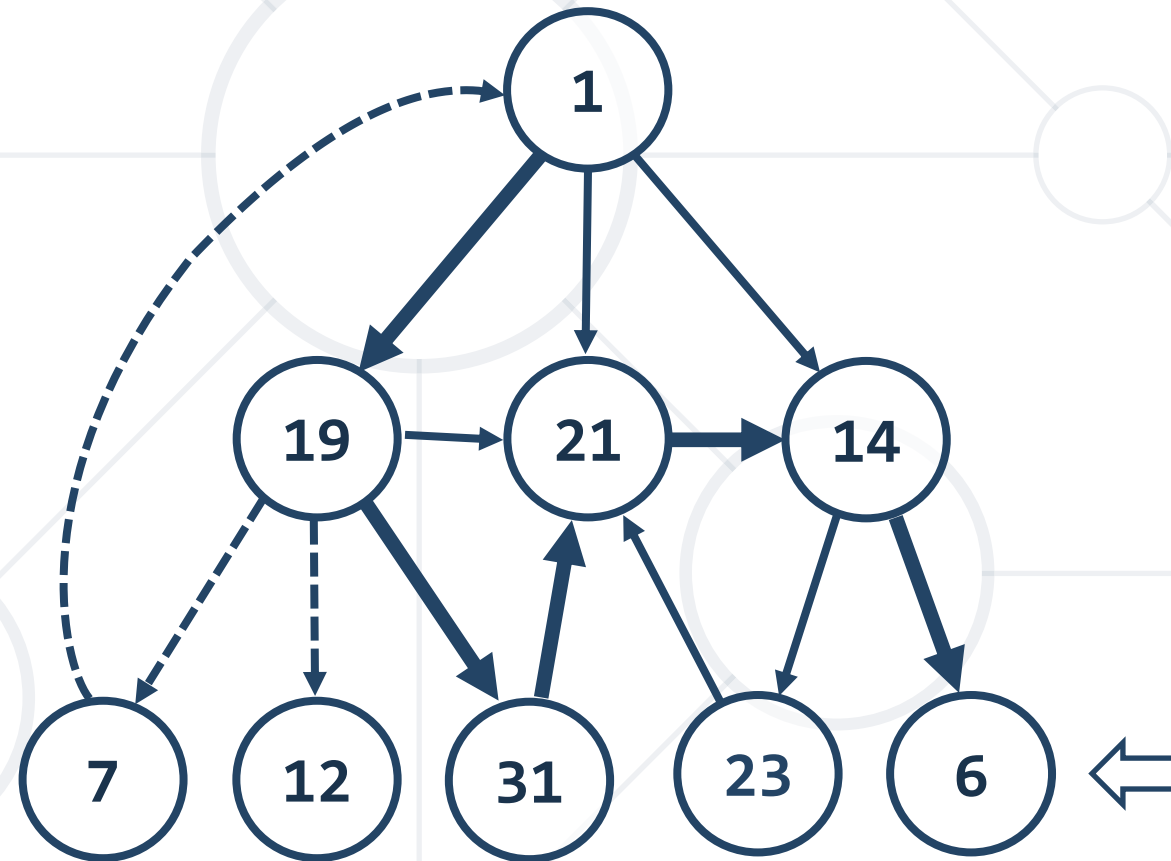
DFS in Action (Step 11)

- Stack: 1, 19, 31, 21, 14
- Output: 7, 12



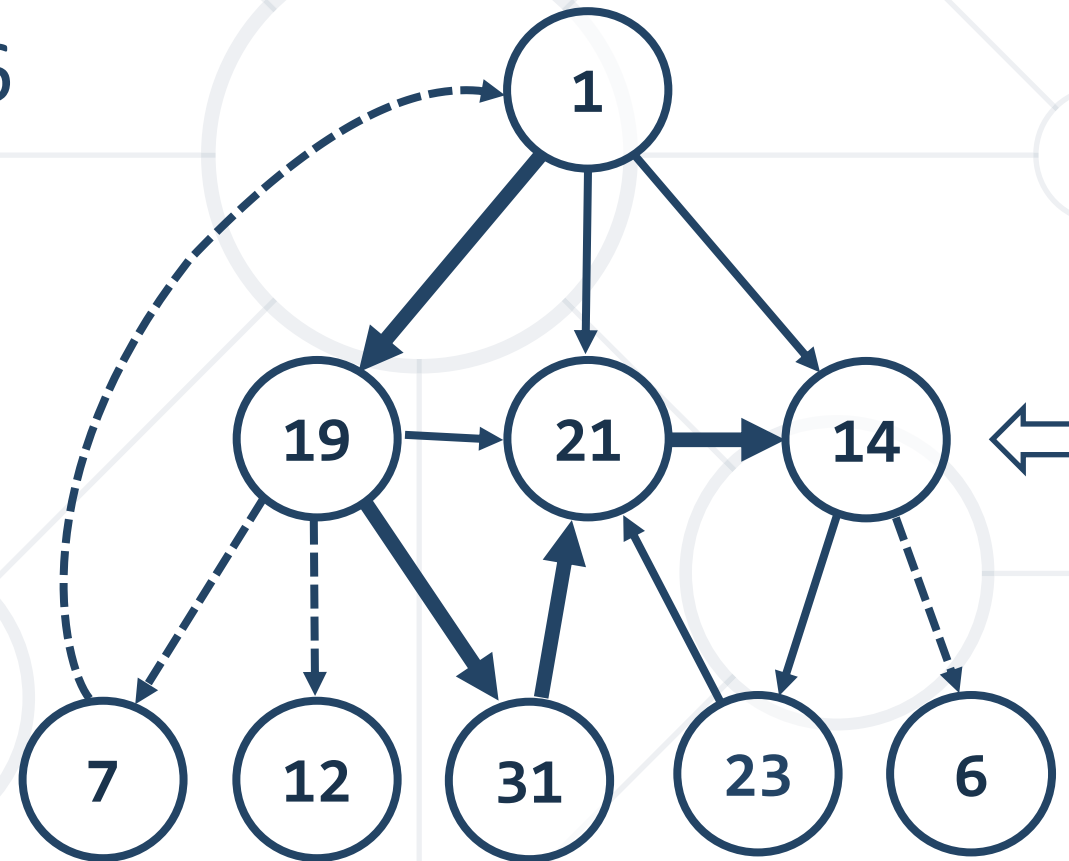
DFS in Action (Step 12)

- Stack: 1, 19, 31, 21, 14, 6
- Output: 7, 12



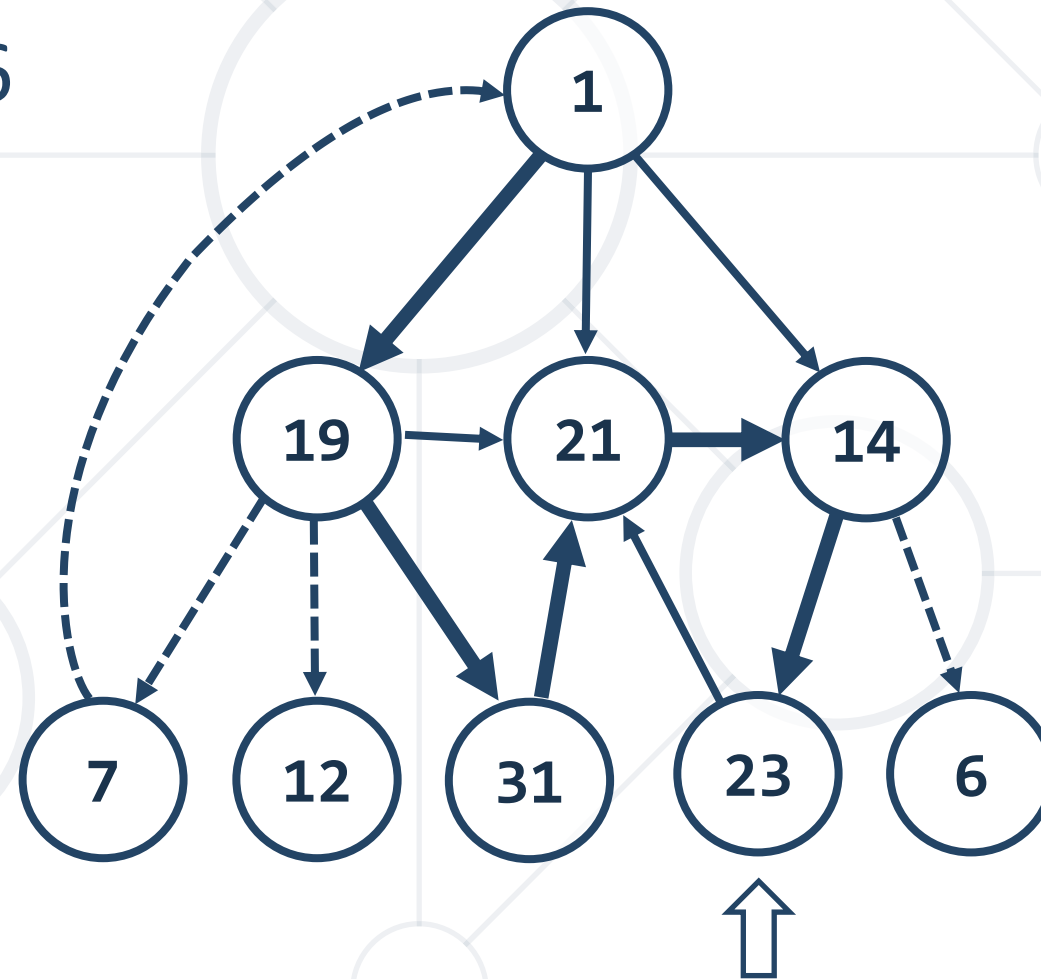
DFS in Action (Step 13)

- Stack: 1, 19, 31, 21, 14
- Output: 7, 12, 6



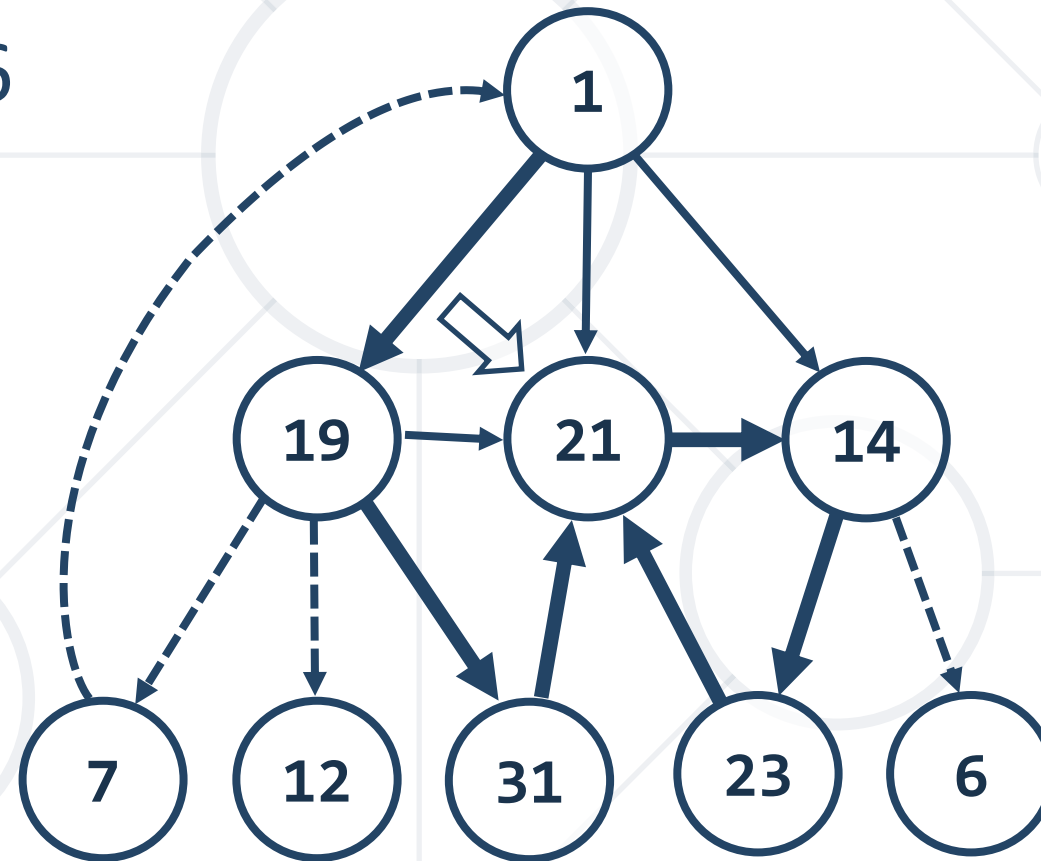
DFS in Action (Step 14)

- Stack: 1, 19, 31, 21, 14, 23
- Output: 7, 12, 6



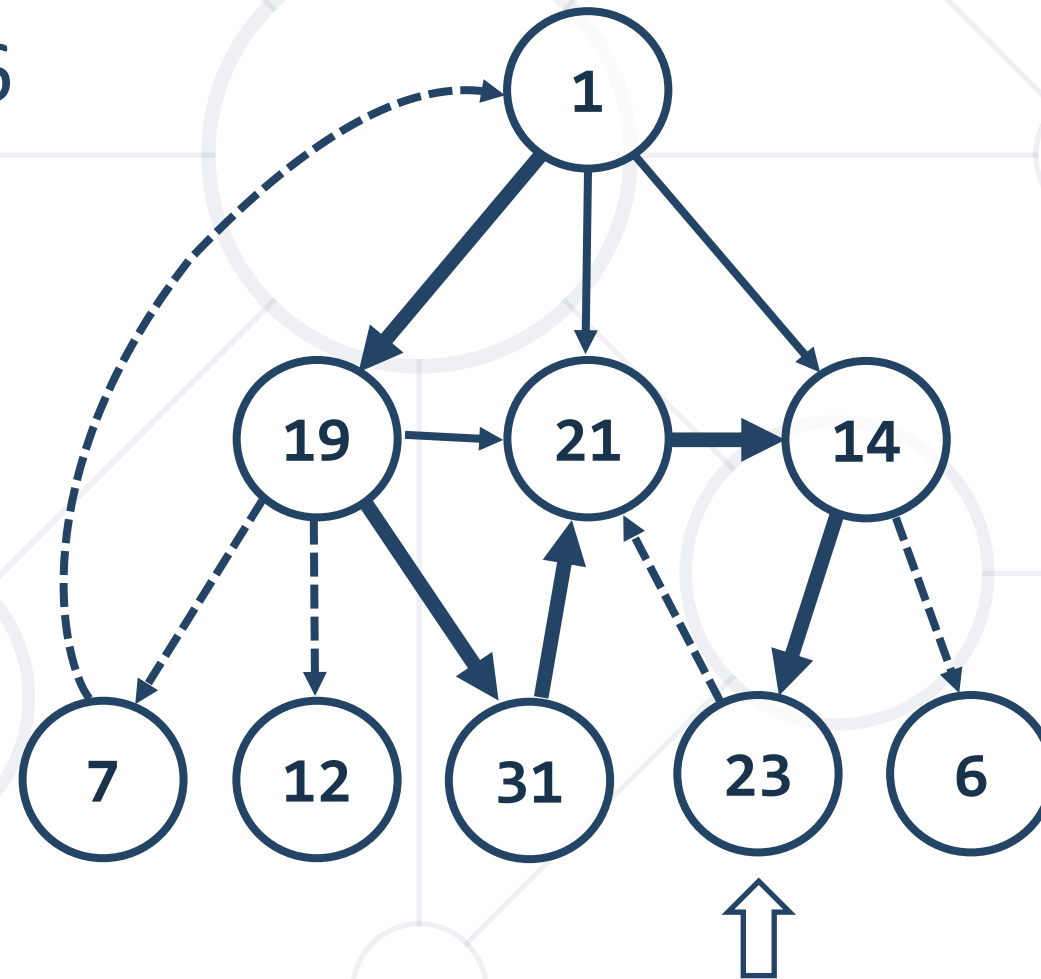
DFS in Action (Step 15)

- Stack: 1, 19, 31, 21, 14, 23, 21
- Output: 7, 12, 6



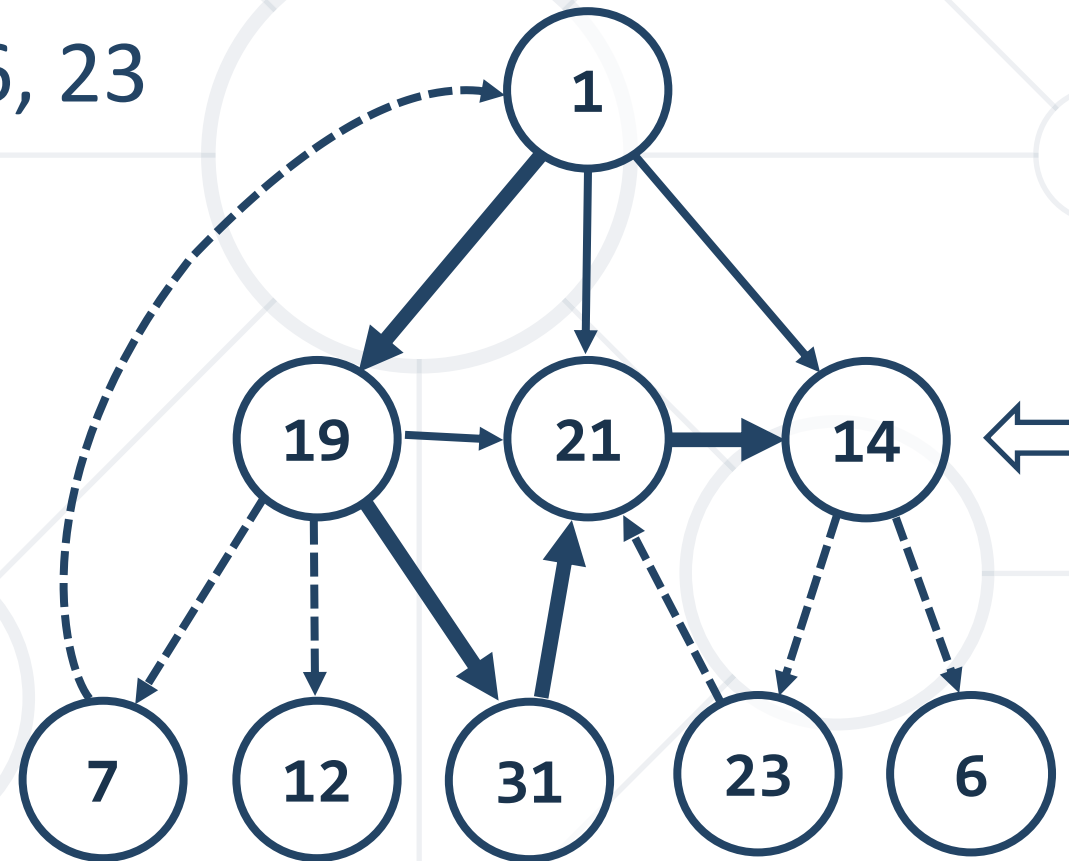
DFS in Action (Step 16)

- Stack: 1, 19, 31, 21, 14, 23
- Output: 7, 12, 6



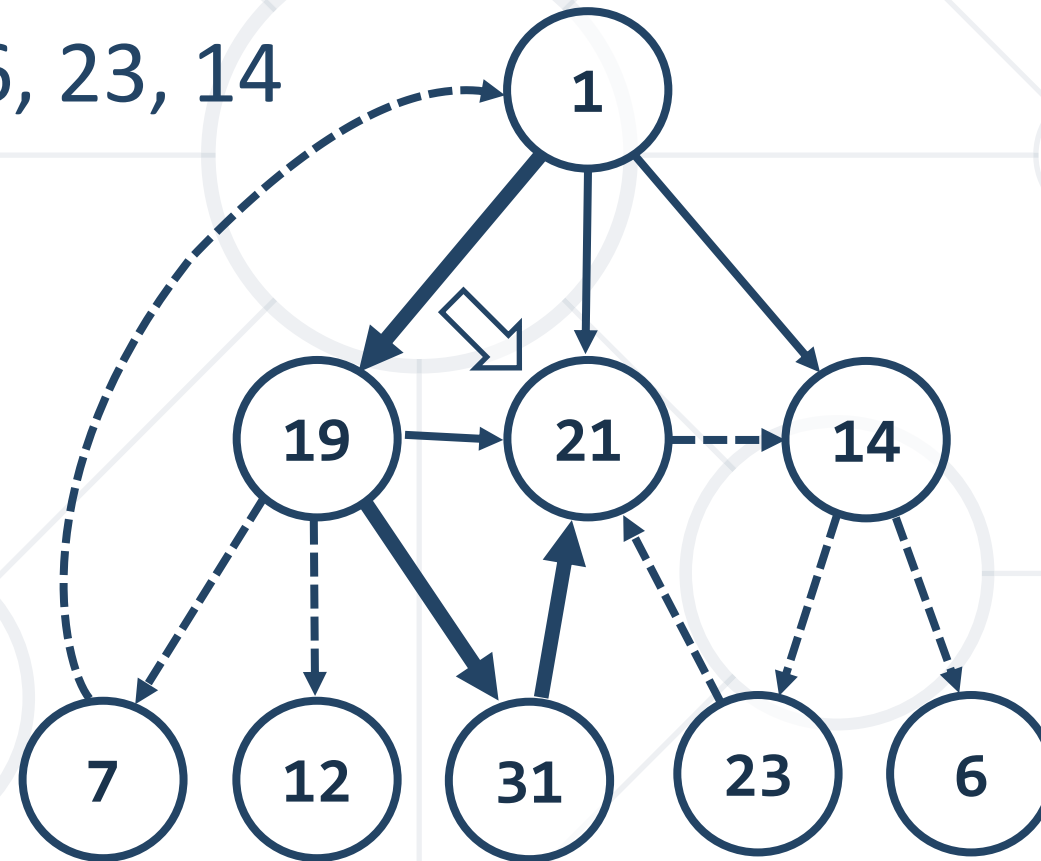
DFS in Action (Step 17)

- Stack: 1, 19, 31, 21, 14
- Output: 7, 12, 6, 23



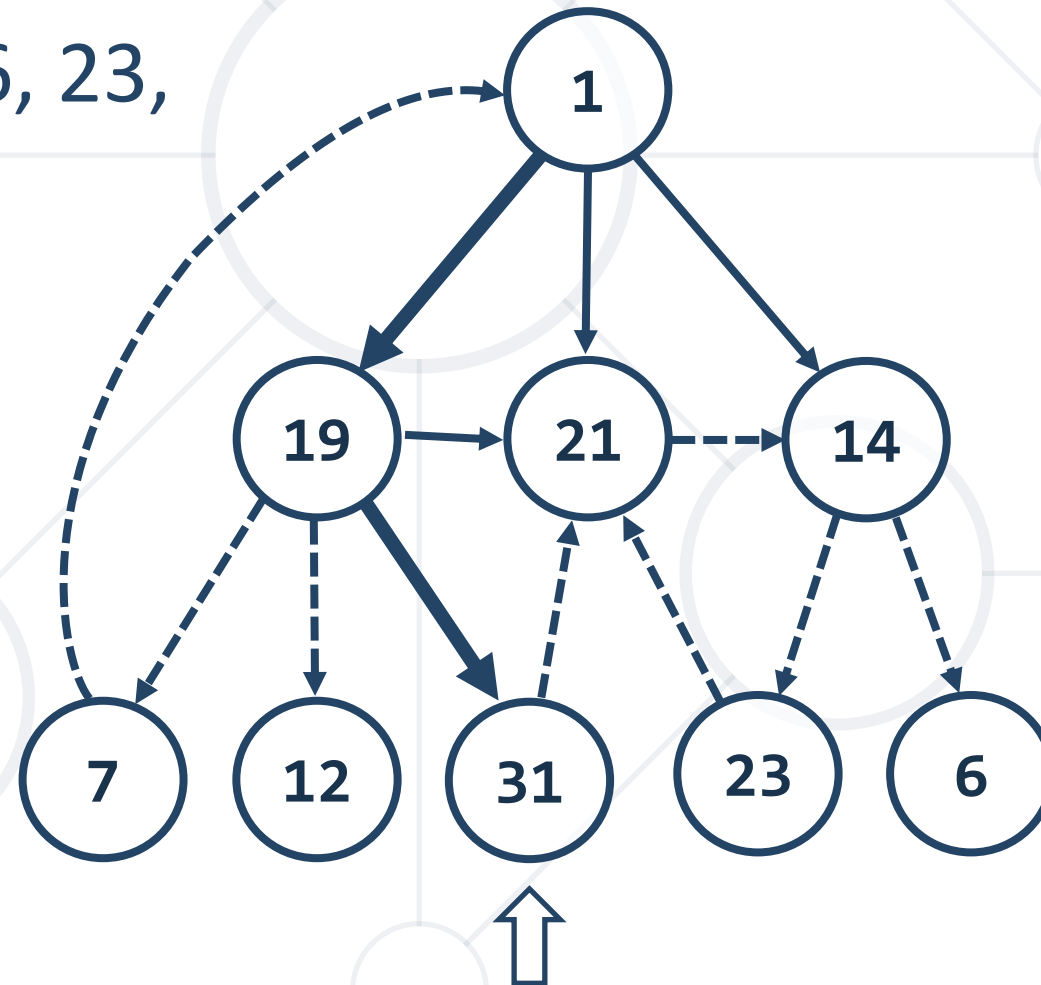
DFS in Action (Step 18)

- Stack: 1, 19, 31, 21
- Output: 7, 12, 6, 23, 14



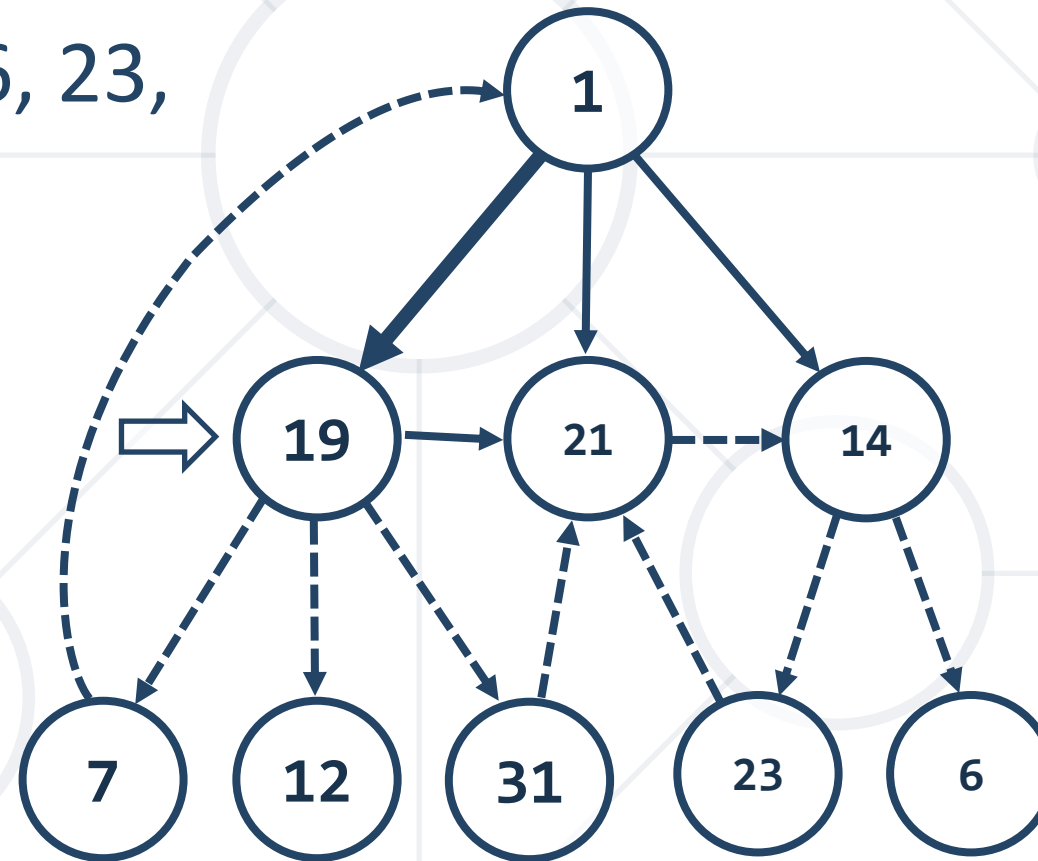
DFS in Action (Step 19)

- Stack: 1, 19, 31
- Output: 7, 12, 6, 23, 14, 21



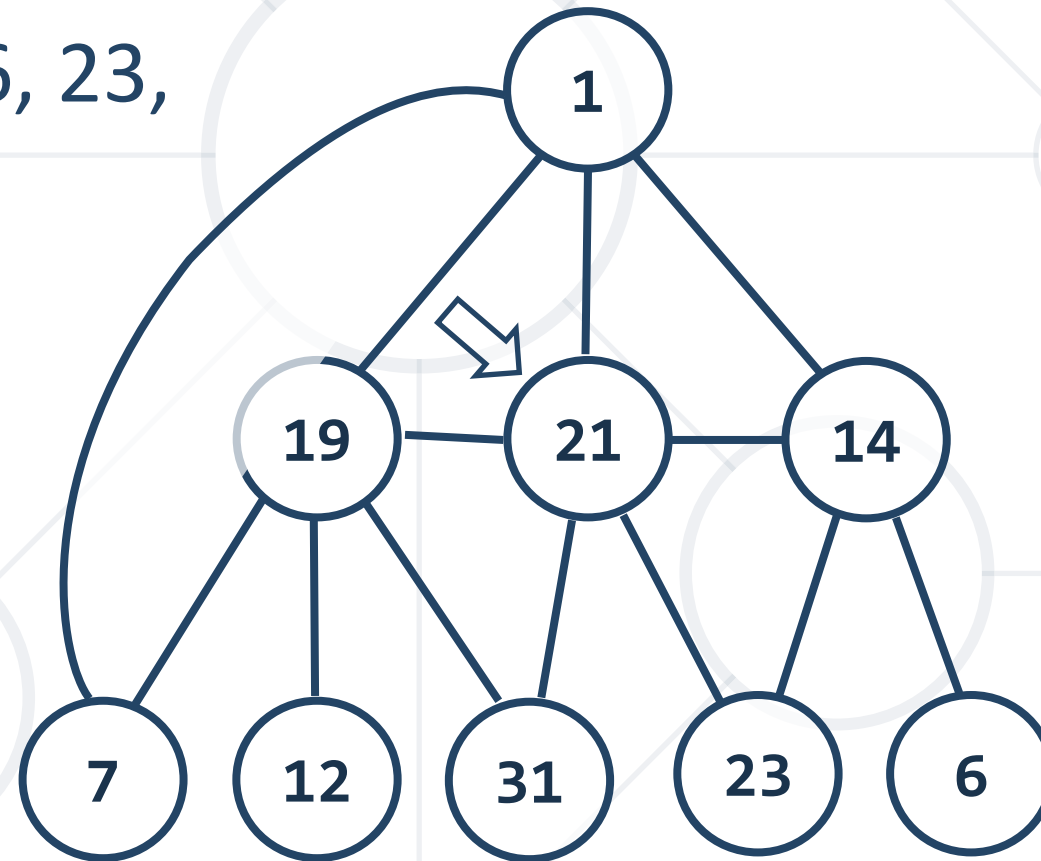
DFS in Action (Step 20)

- Stack: 1, 19
- Output: 7, 12, 6, 23, 14, 21, 31



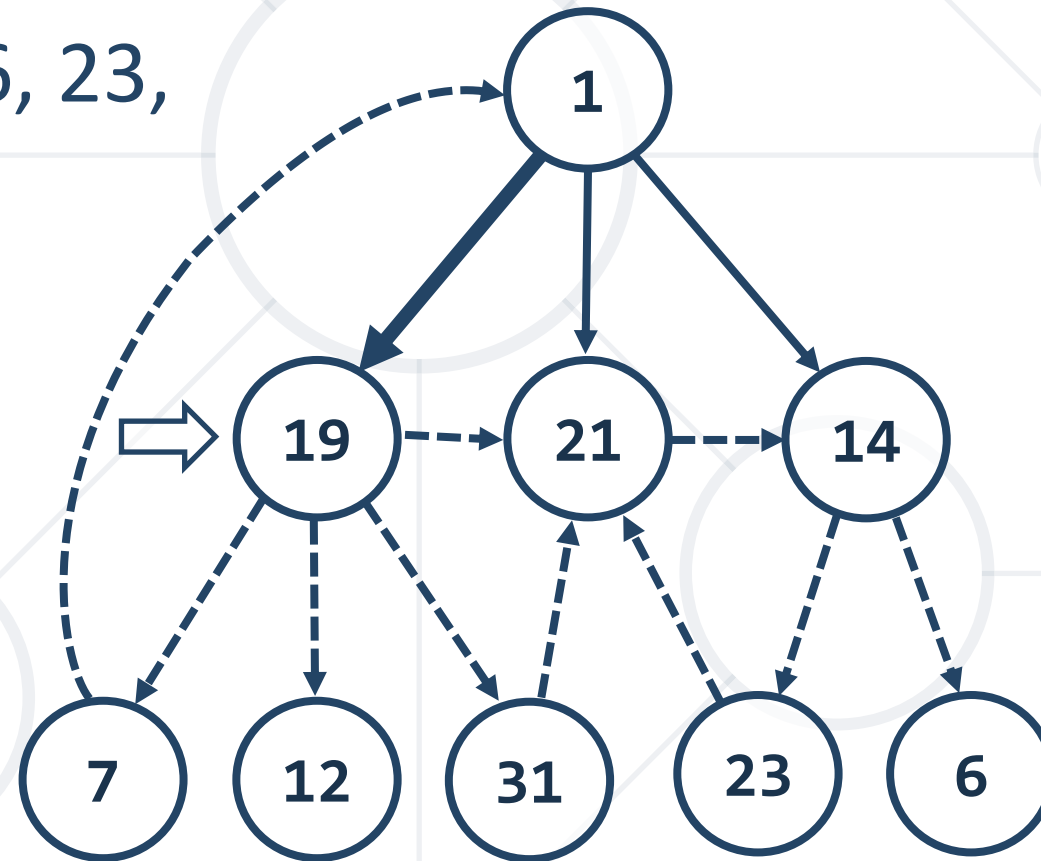
DFS in Action (Step 21)

- Stack: 1, 19, 21
- Output: 7, 12, 6, 23, 14, 21, 31



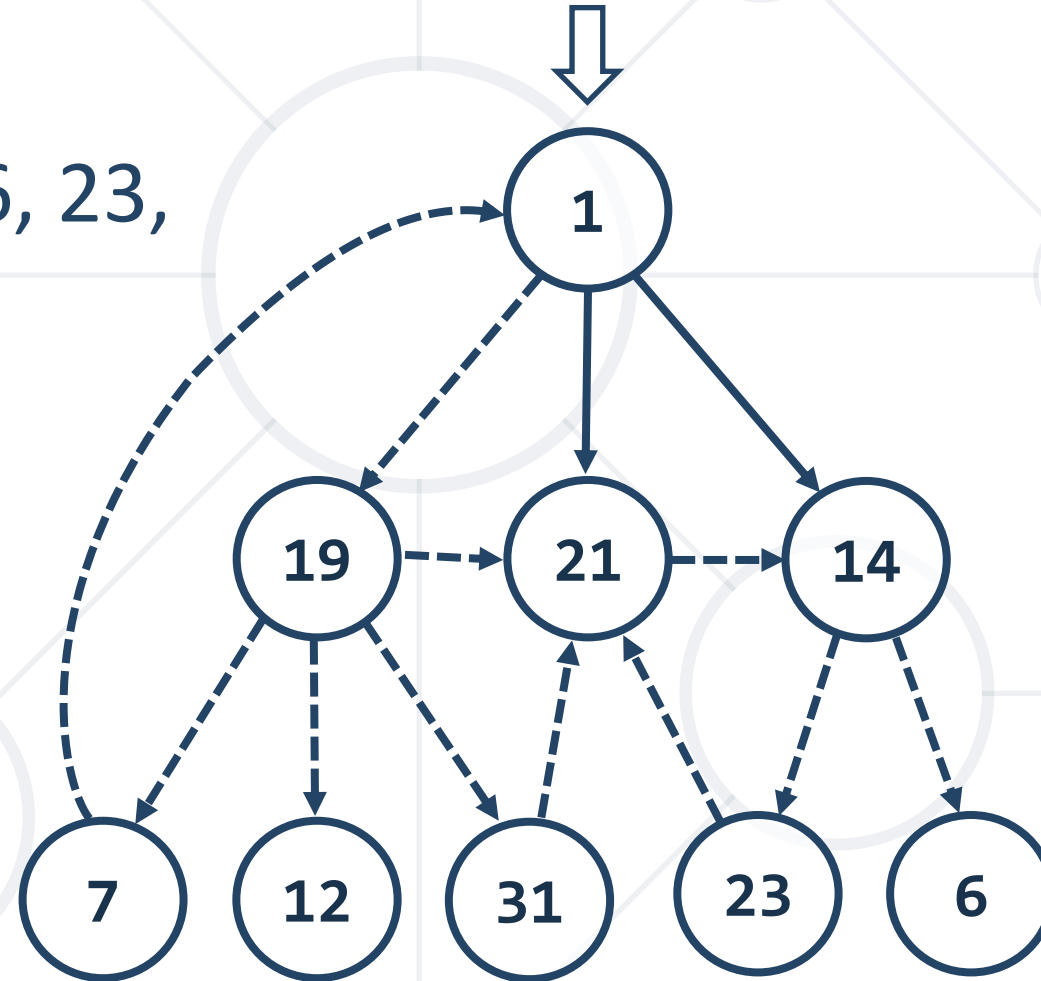
DFS in Action (Step 22)

- Stack: 1, 19
- Output: 7, 12, 6, 23, 14, 21, 31



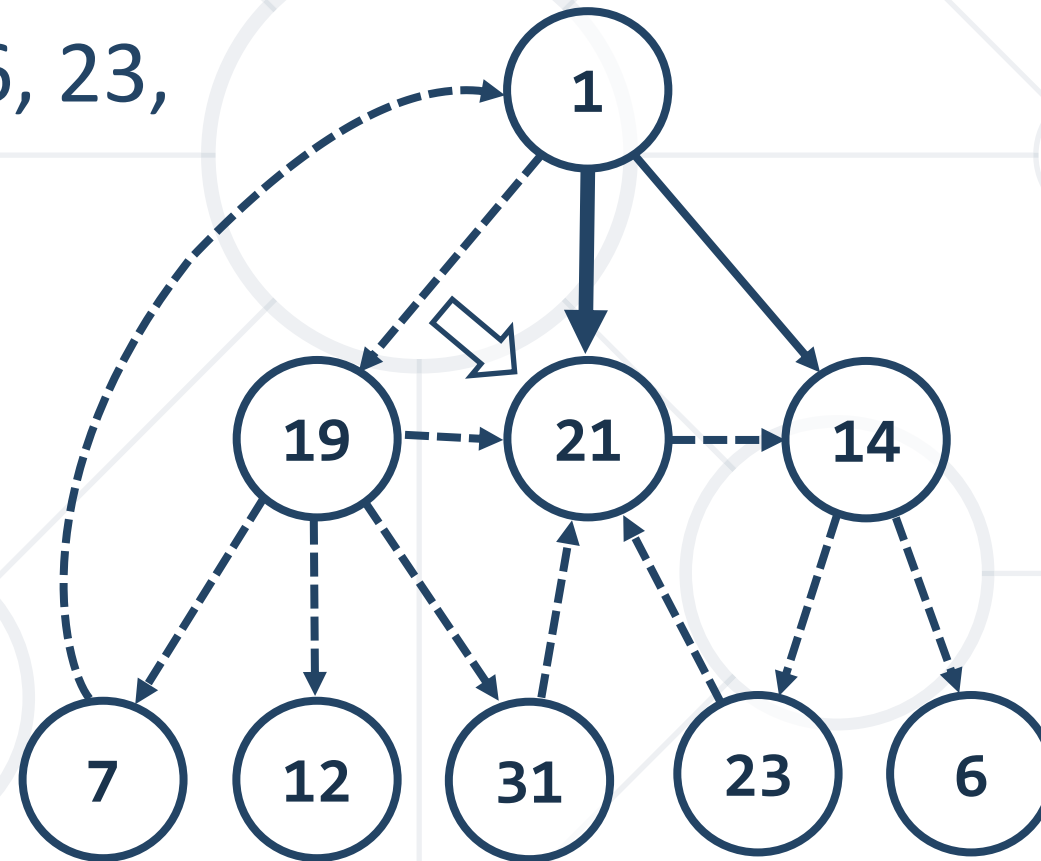
DFS in Action (Step 23)

- Stack: 1
- Output: 7, 12, 6, 23, 14, 21, 31, 19



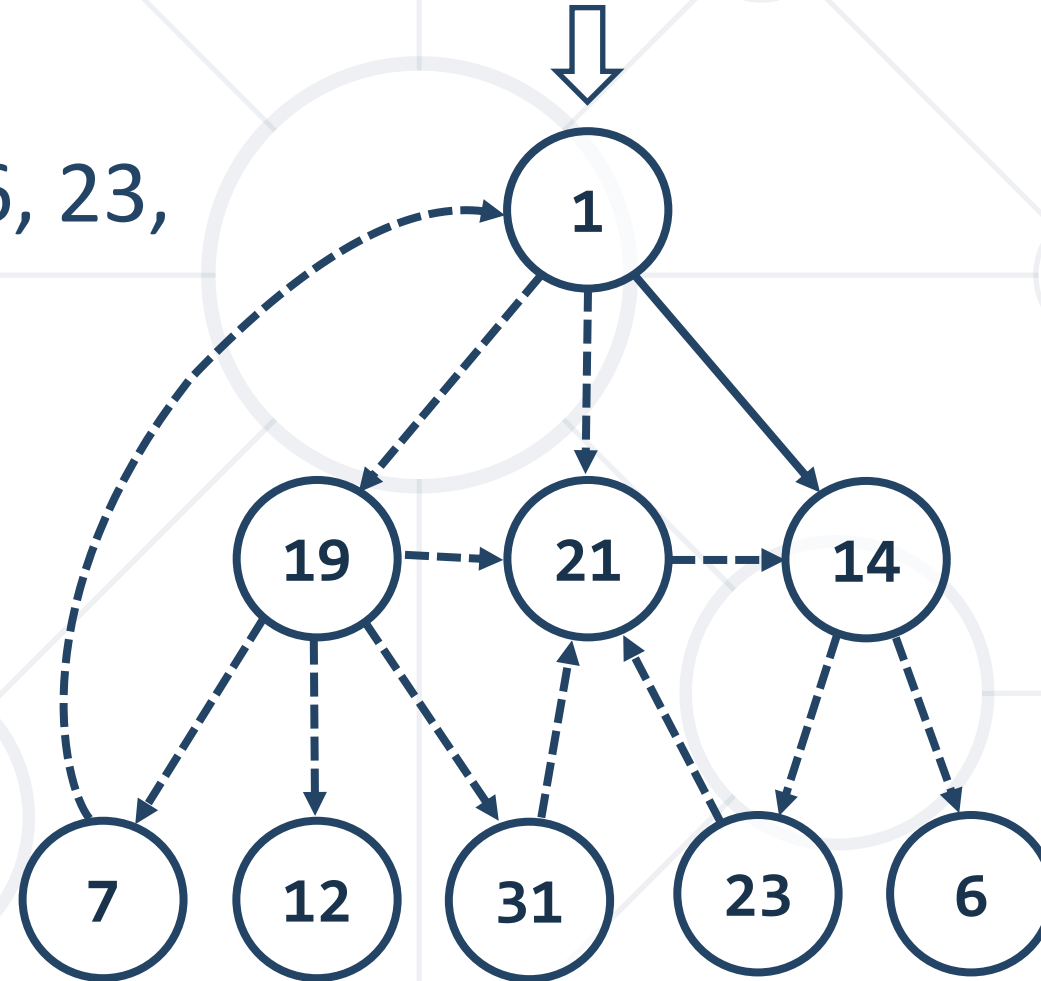
DFS in Action (Step 24)

- Stack: 1, 21
- Output: 7, 12, 6, 23, 14, 21, 31, 19



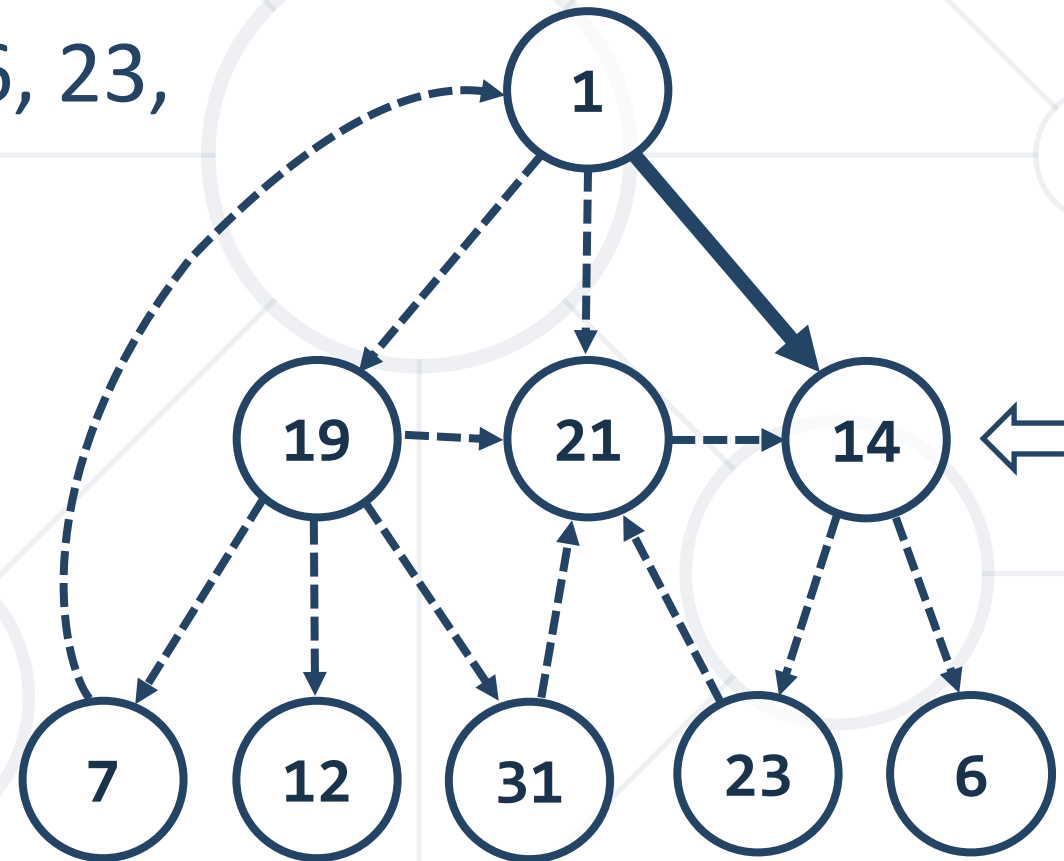
DFS in Action (Step 25)

- Stack: 1
- Output: 7, 12, 6, 23, 14, 21, 31, 19



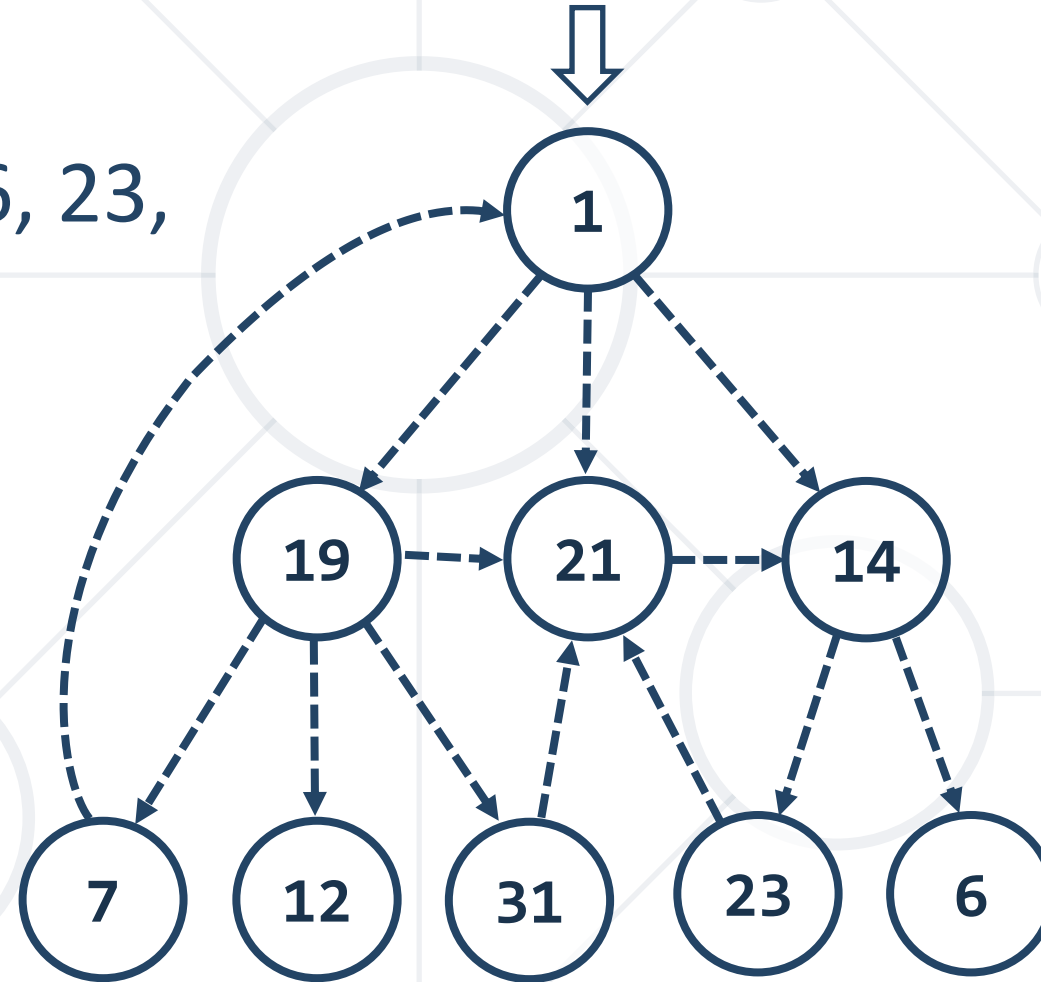
DFS in Action (Step 26)

- Stack: 1, 14
- Output: 7, 12, 6, 23, 14, 21, 31, 19



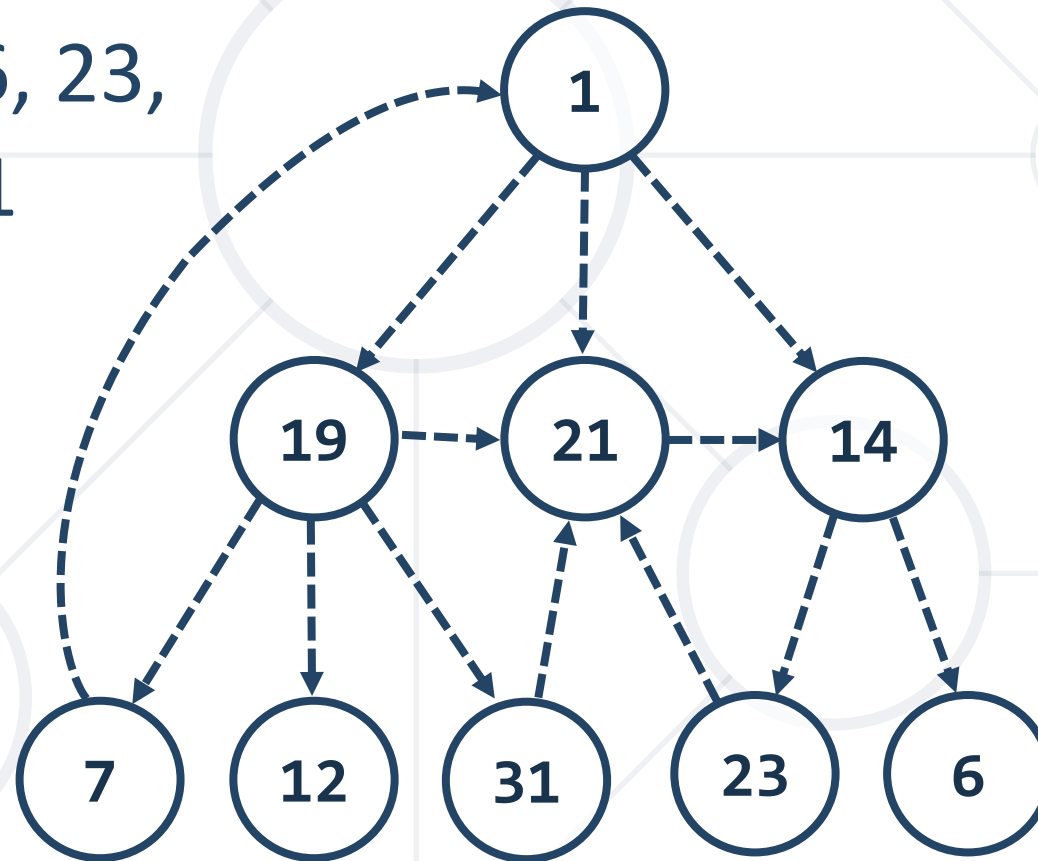
DFS in Action (Step 27)

- Stack: 1
- Output: 7, 12, 6, 23, 14, 21, 31, 19



DFS in Action (Step 28)

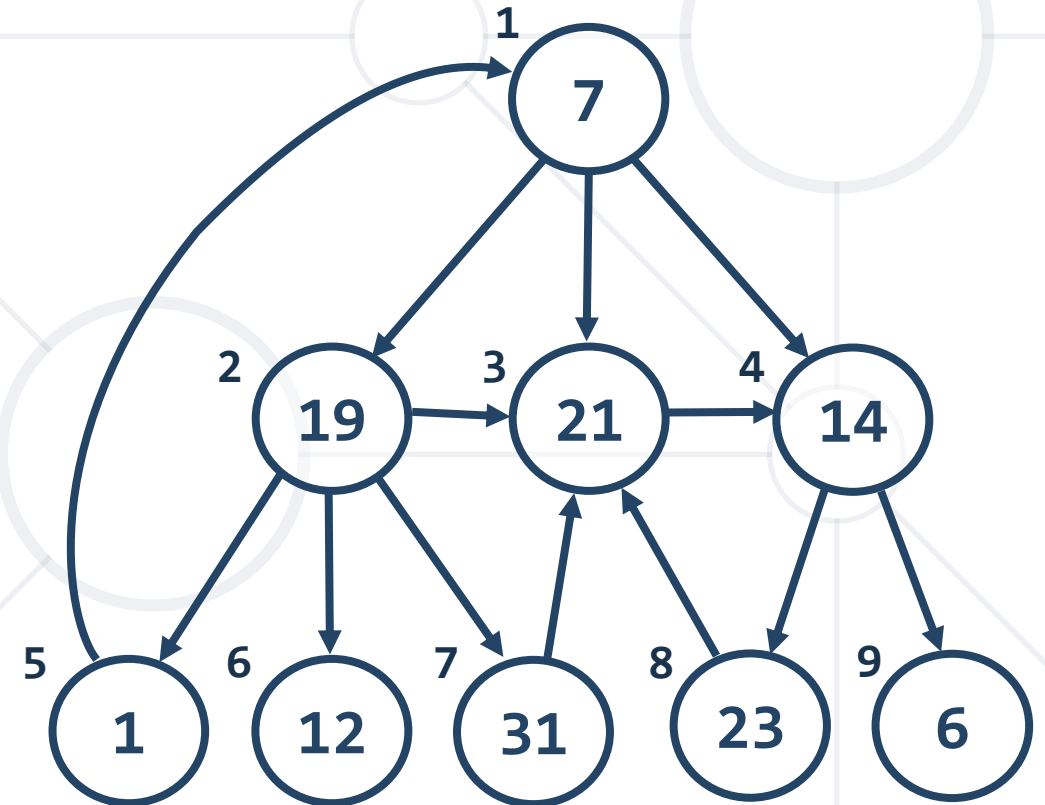
- Stack: (empty)
- Output: 7, 12, 6, 23, 14, 21, 31, 19, 1



Breadth-First Search (BFS)

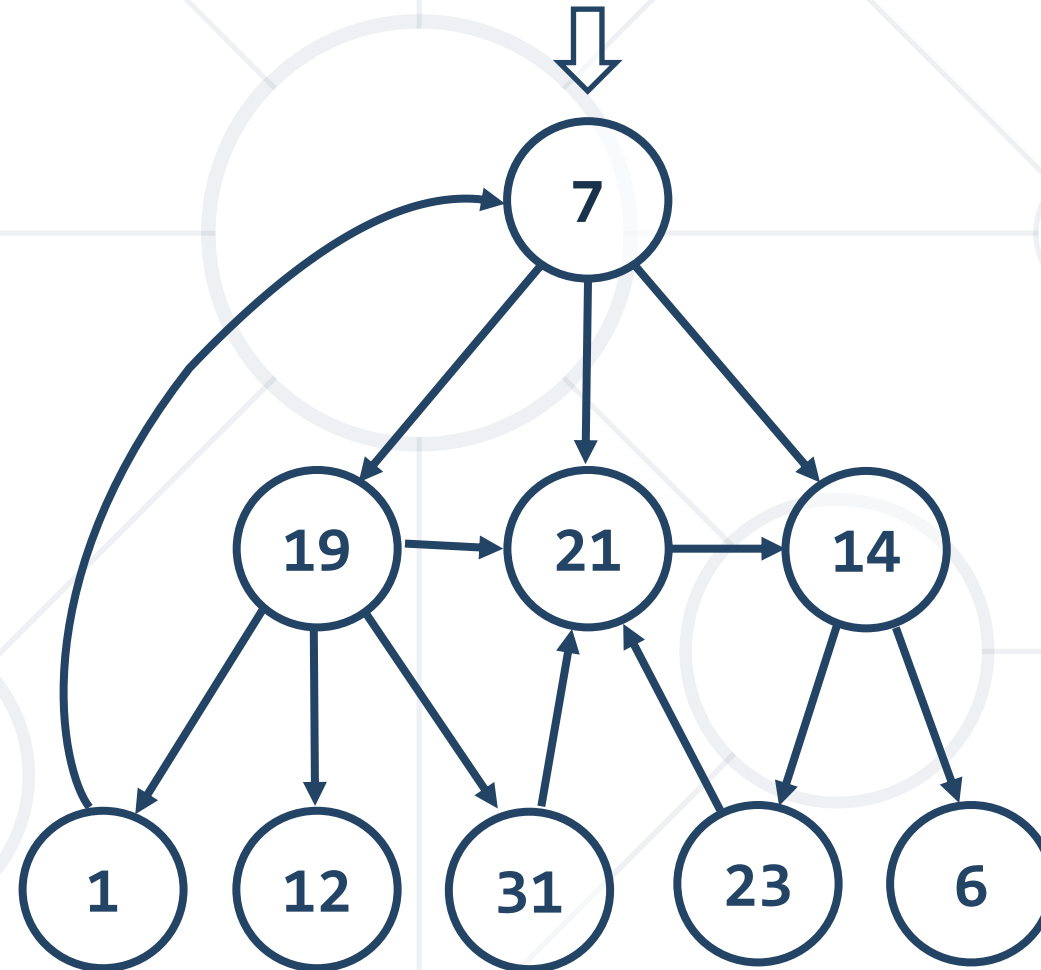
- **Breadth-First Search (BFS)** first visits the neighbor nodes, then the neighbors of neighbors, then their neighbors, etc.

```
bfs(node) {  
  queue ← node  
  visited[node] = true  
  while queue not empty  
    v ← queue  
    print v  
    for each child c of v  
      if not visited[c]  
        queue ← c  
        visited[c] = true  
}
```



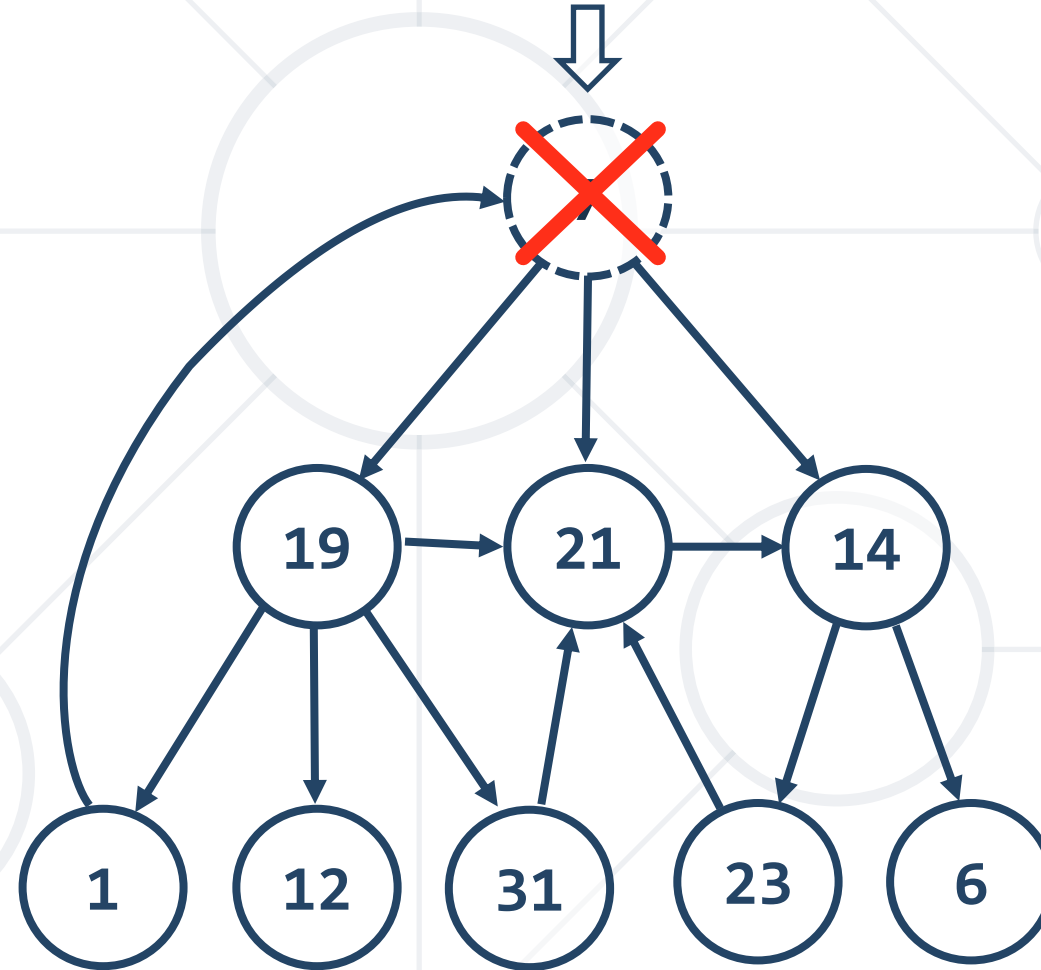
BFS in Action (Step 1)

- Queue: 7
- Output:



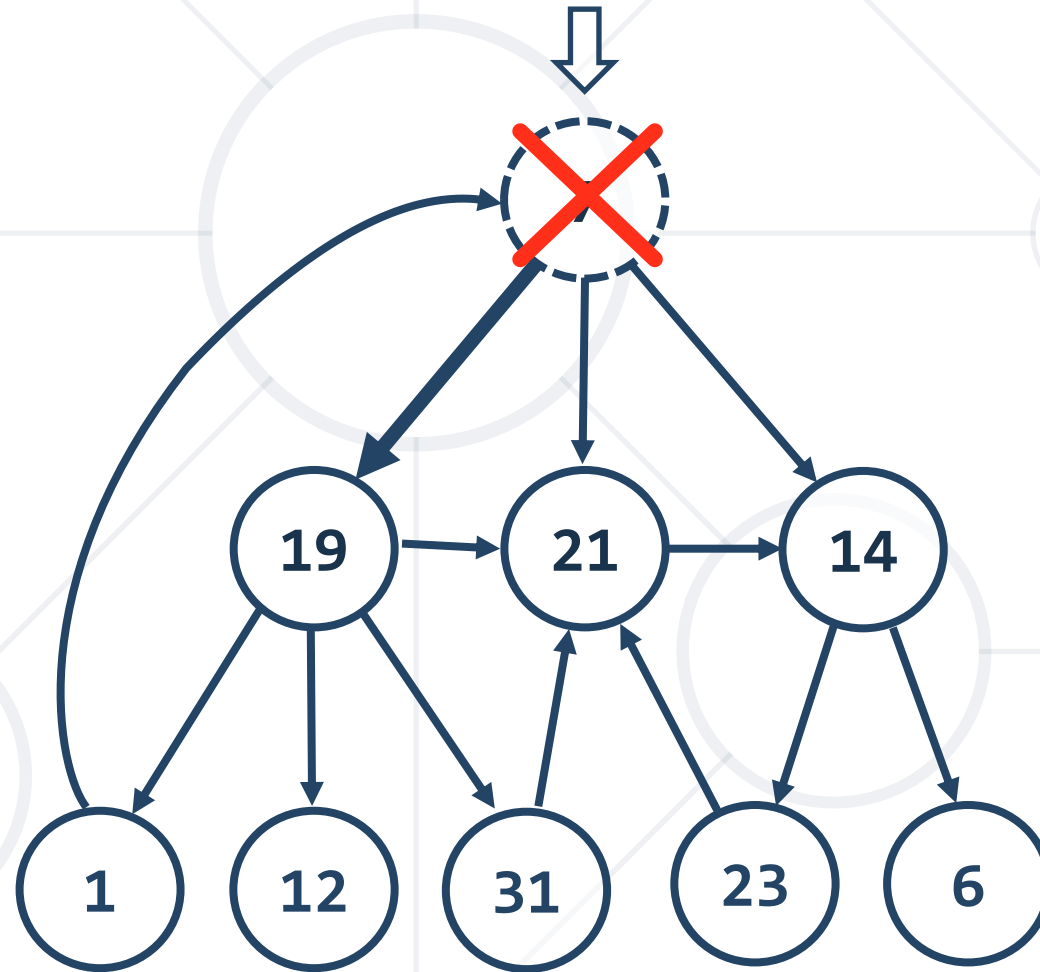
BFS in Action (Step 2)

- Queue: ~~7~~
- Output: 7



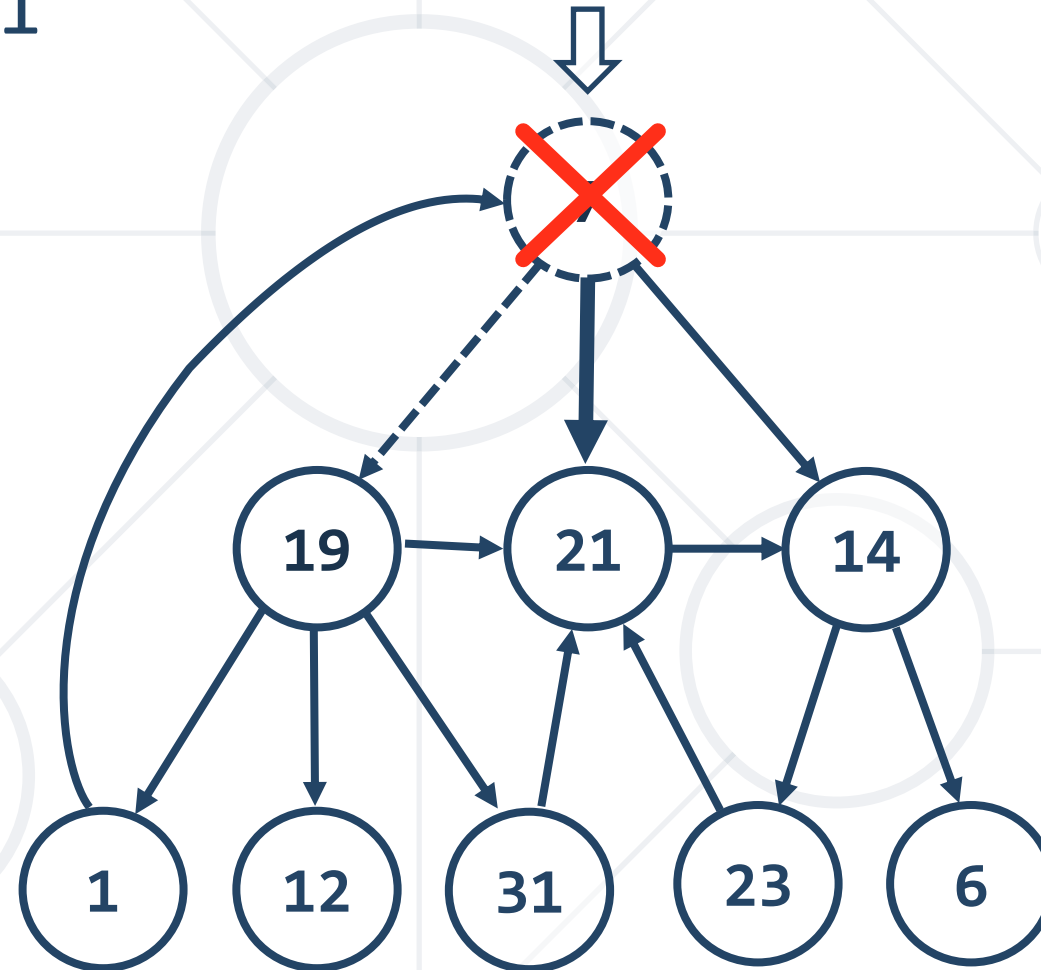
BFS in Action (Step 3)

- Queue: ~~7~~, 19
- Output: 7



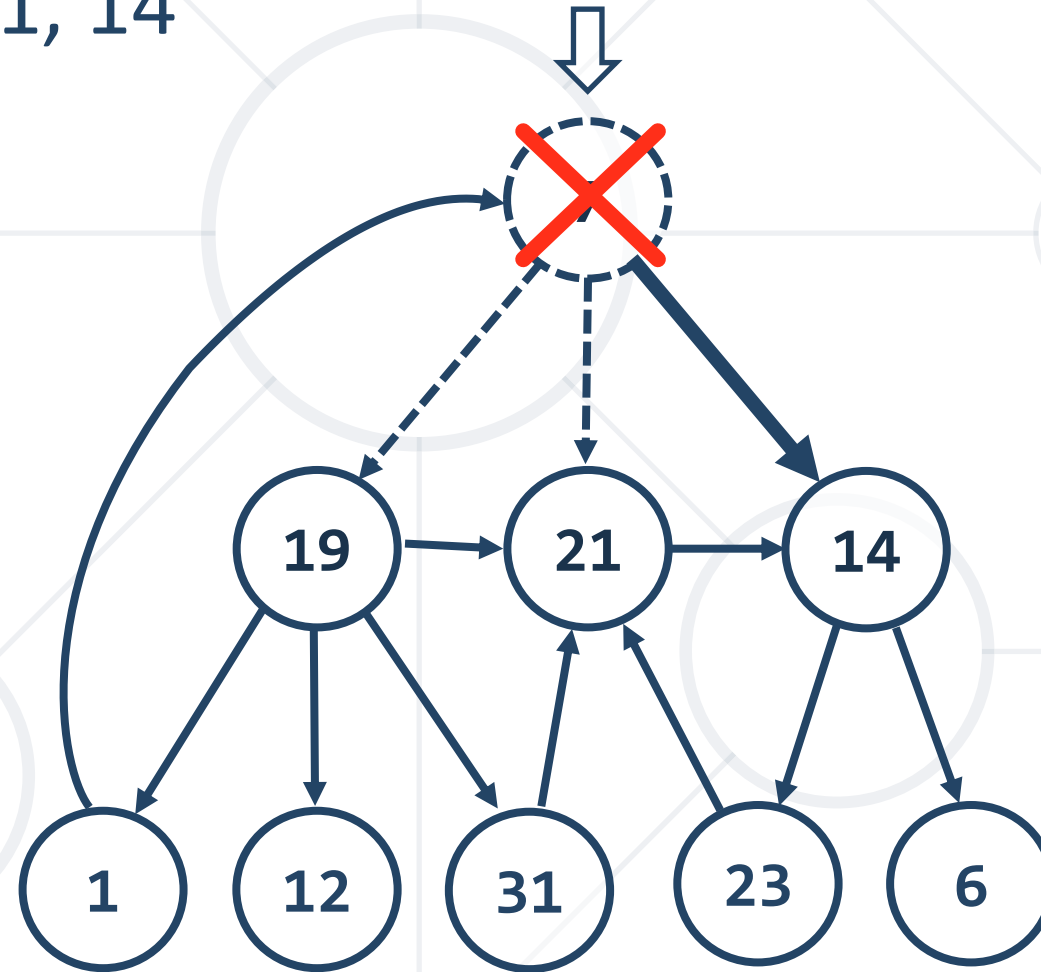
BFS in Action (Step 4)

- Queue: ~~7~~, 19, 21
- Output: 7



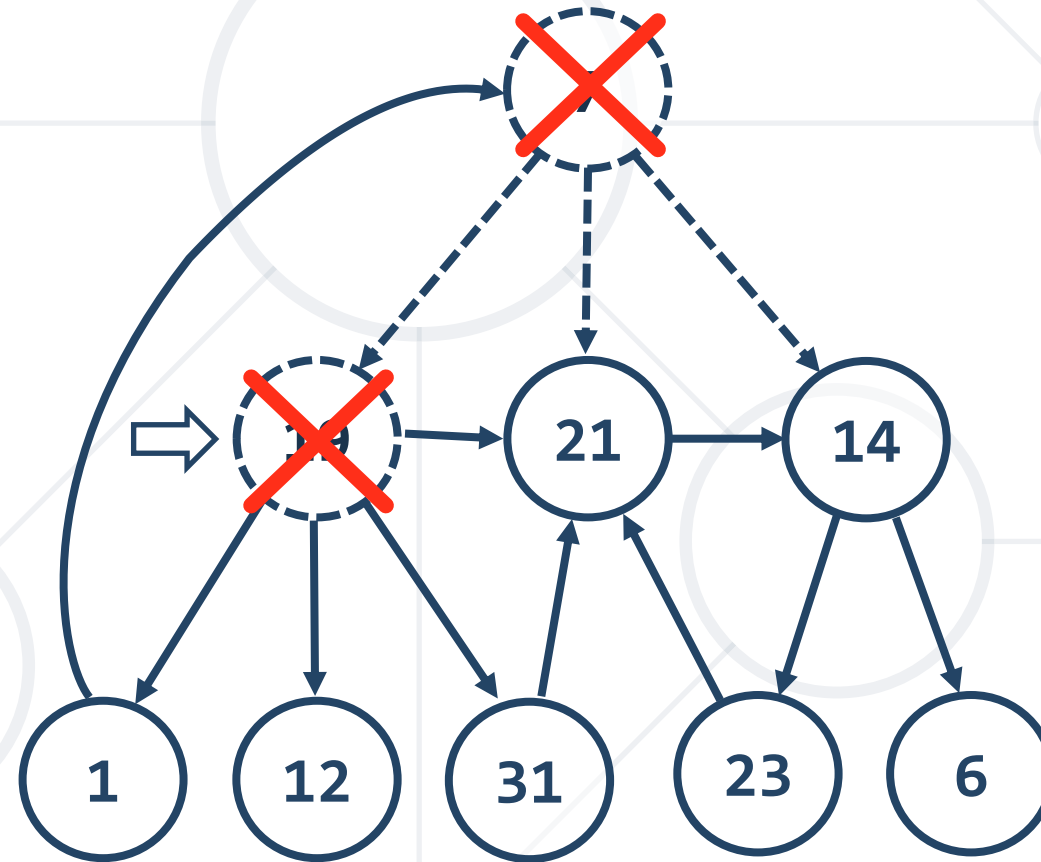
BFS in Action (Step 5)

- Queue: ~~7~~, 19, 21, 14
- Output: 7



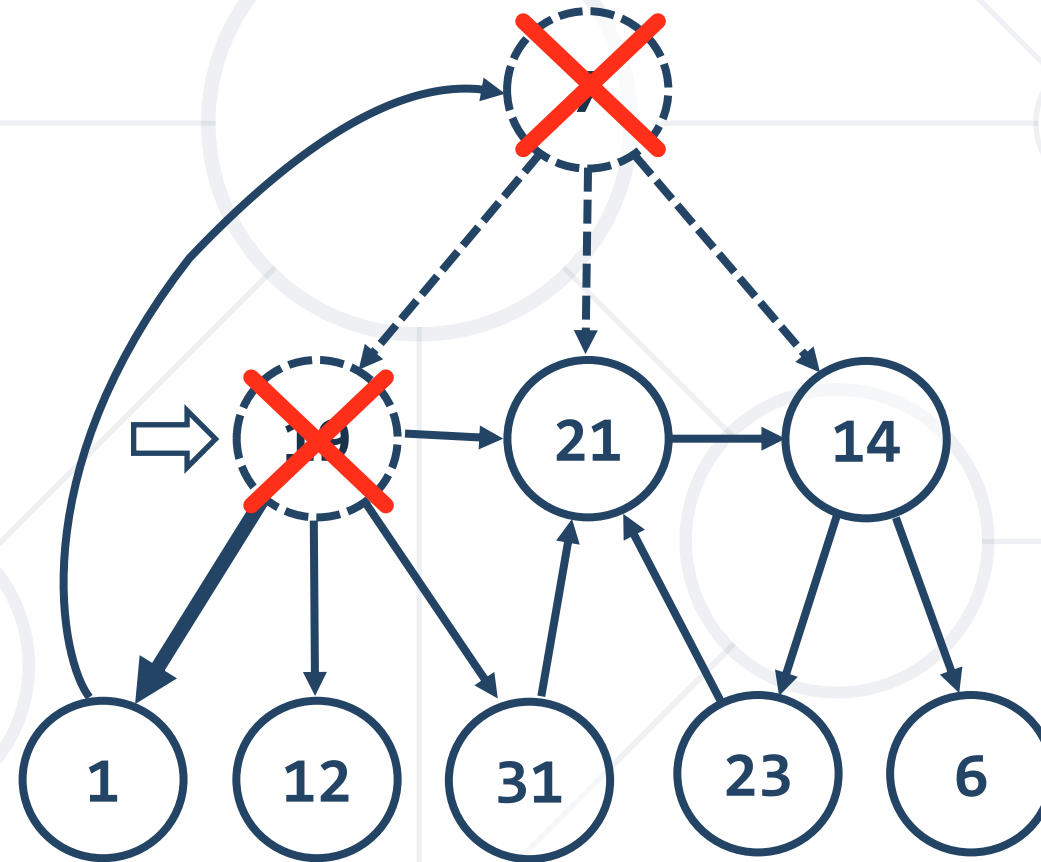
BFS in Action (Step 6)

- Queue: ~~7~~, ~~19~~, 21, 14
- Output: 7, 19



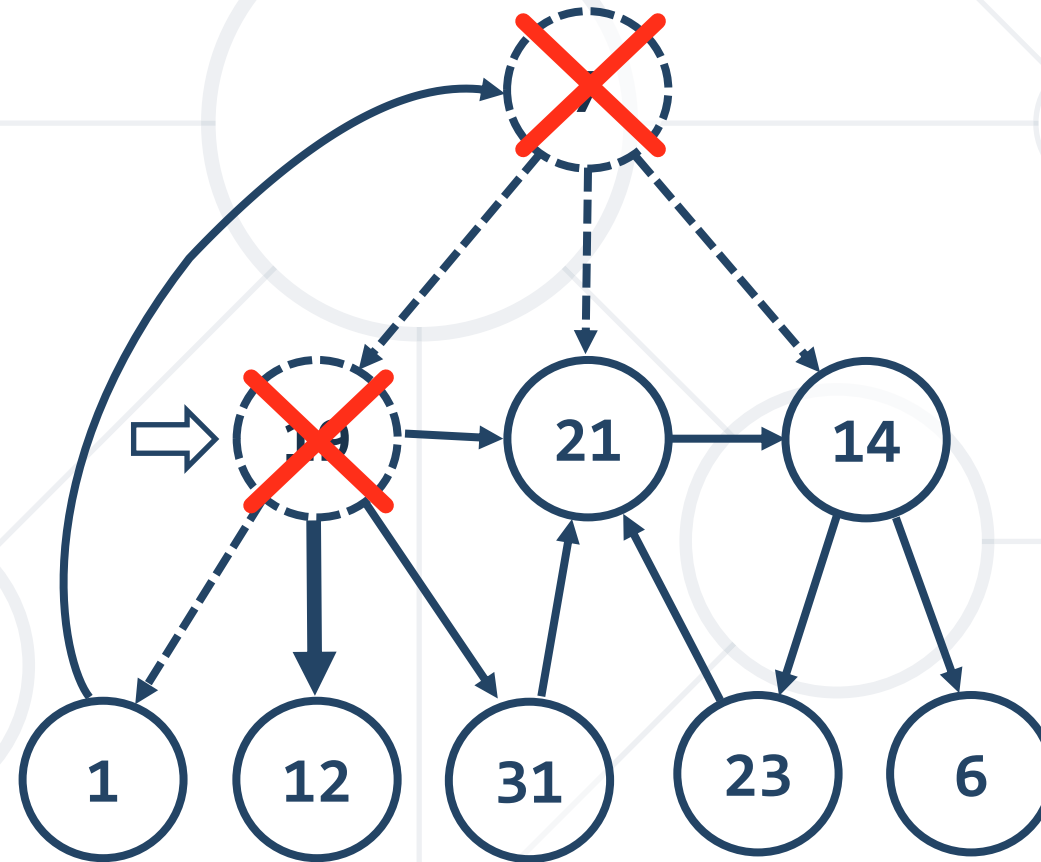
BFS in Action (Step 7)

- Queue: ~~7~~, ~~19~~, 21, 14, 1
- Output: 7, 19



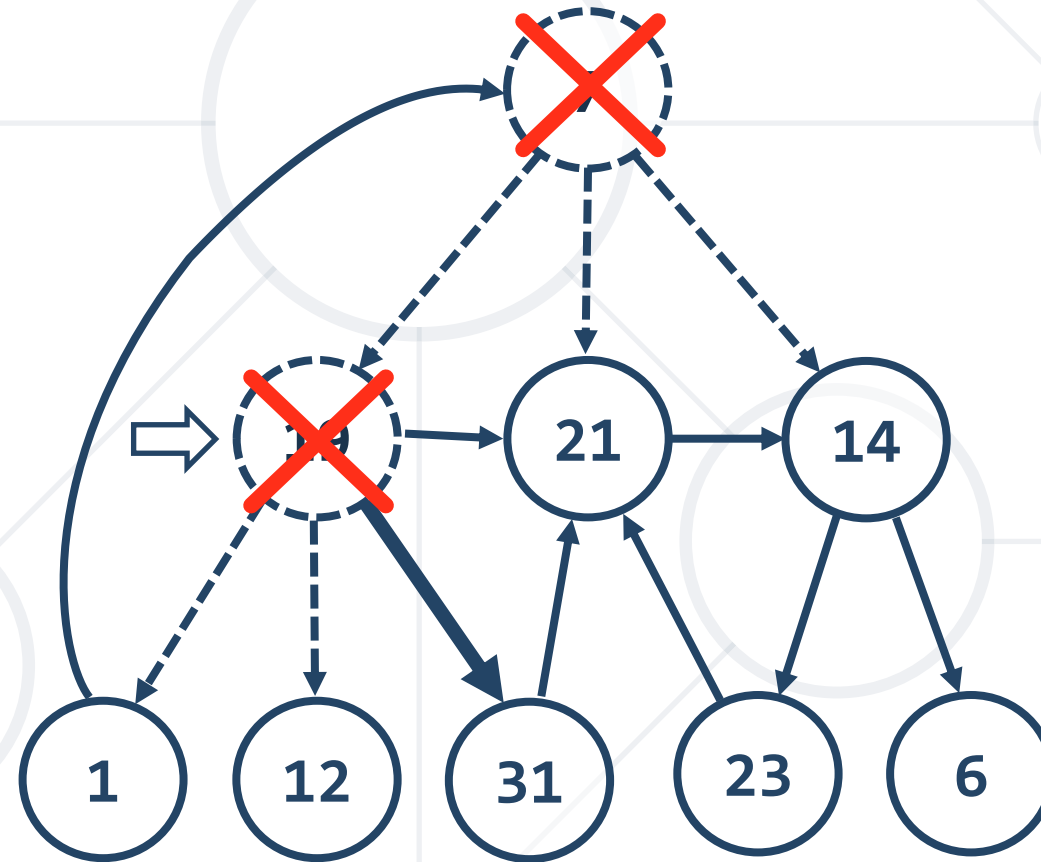
BFS in Action (Step 8)

- Queue: ~~7~~, ~~19~~, 21, 14, 1, 12
- Output: 7, 19



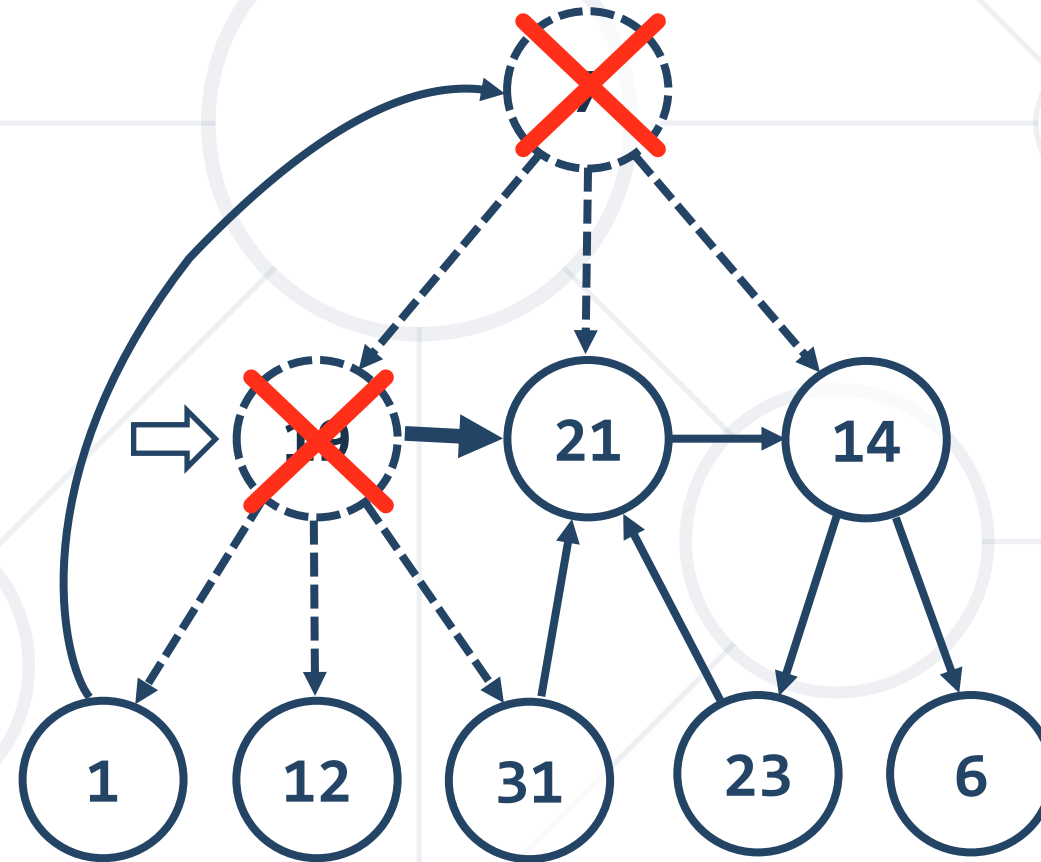
BFS in Action (Step 9)

- Queue: ~~7~~, ~~19~~, 21, 14, 1, 12, 31
- Output: 7, 19



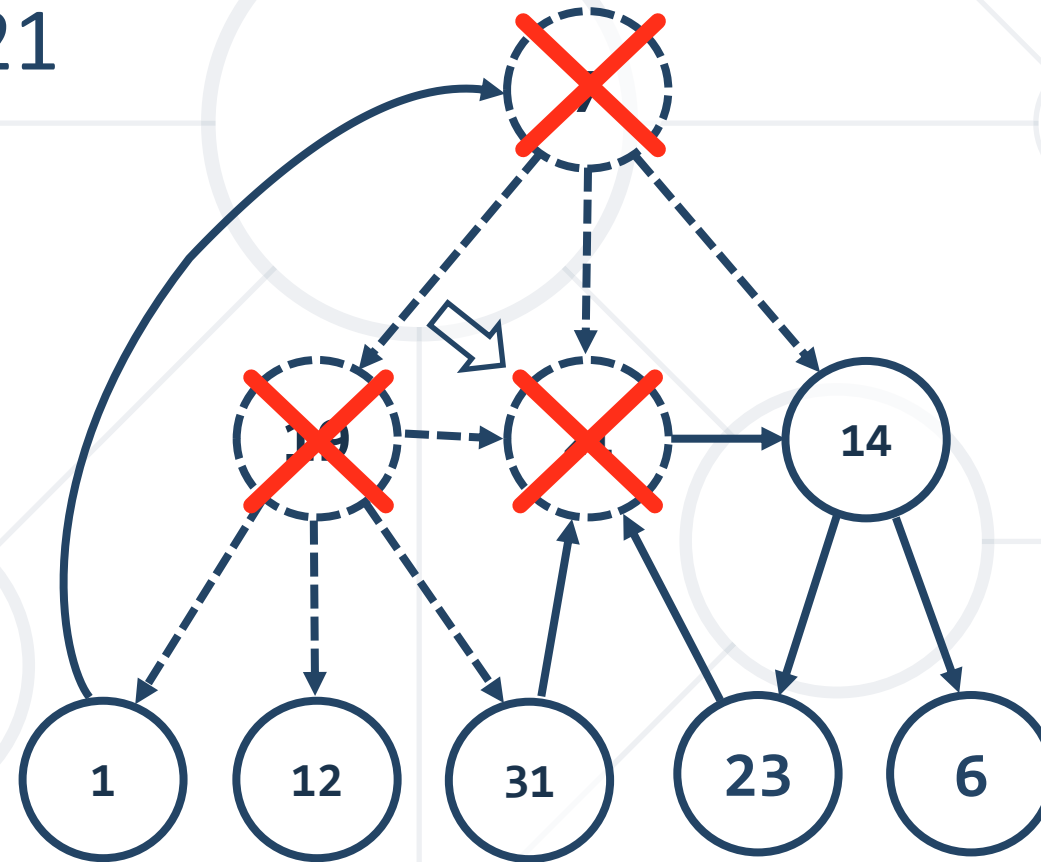
BFS in Action (Step 10)

- Queue: ~~7~~, ~~19~~, 21, 14, 1, 12, 31
- Output: 7, 19



BFS in Action (Step 11)

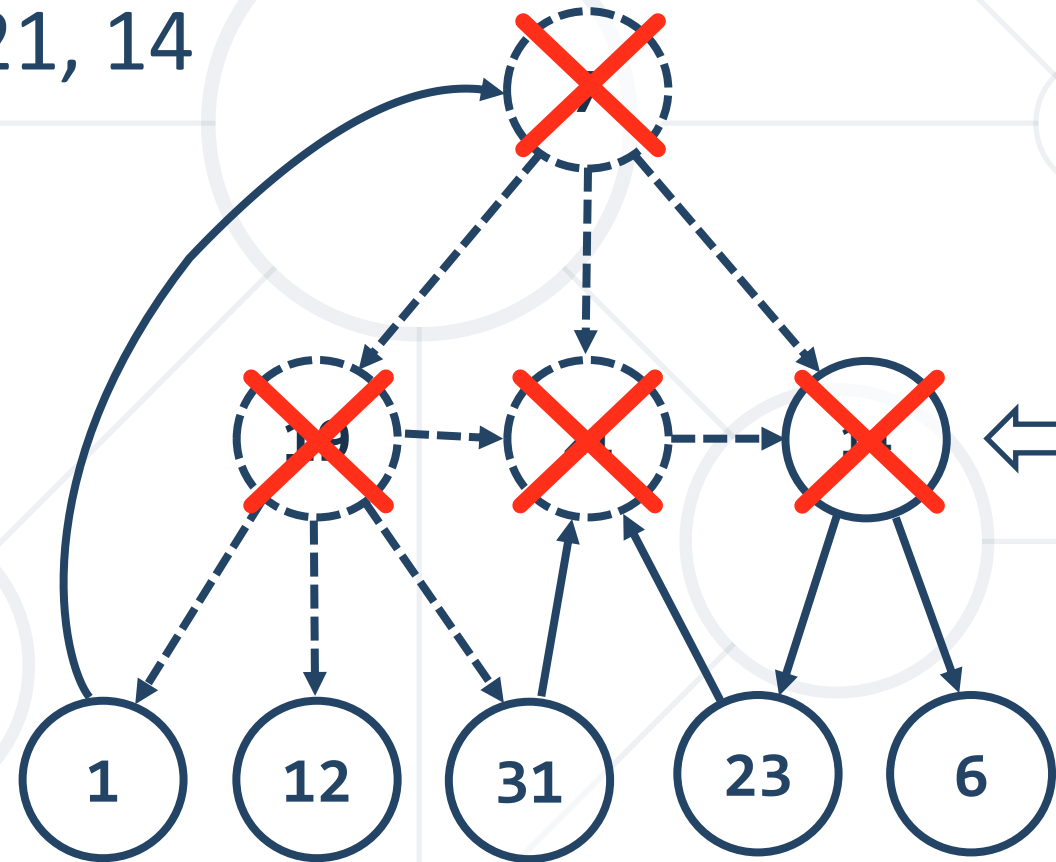
- Queue: ~~7~~, ~~19~~, ~~21~~, 14, 1, 12, 31
- Output: 7, 19, 21



-

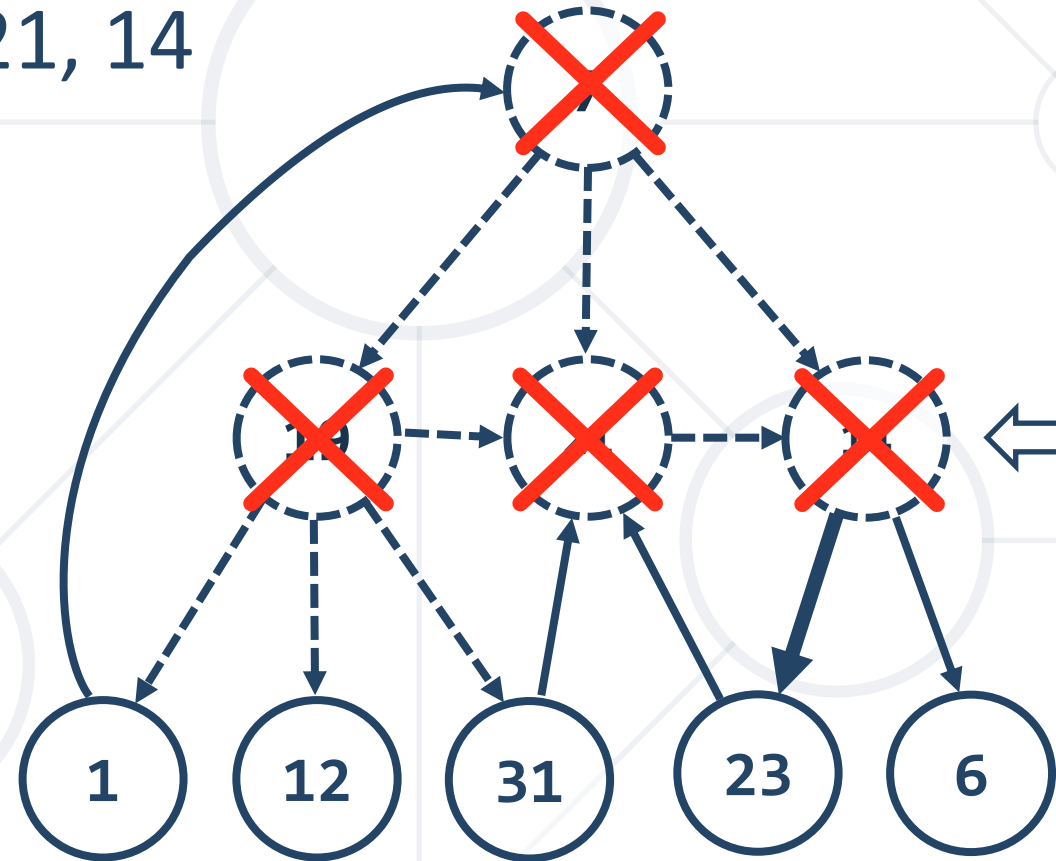
BFS in Action (Step 13)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, 1, 12, 31
- Output: 7, 19, 21, 14



BFS in Action (Step 14)

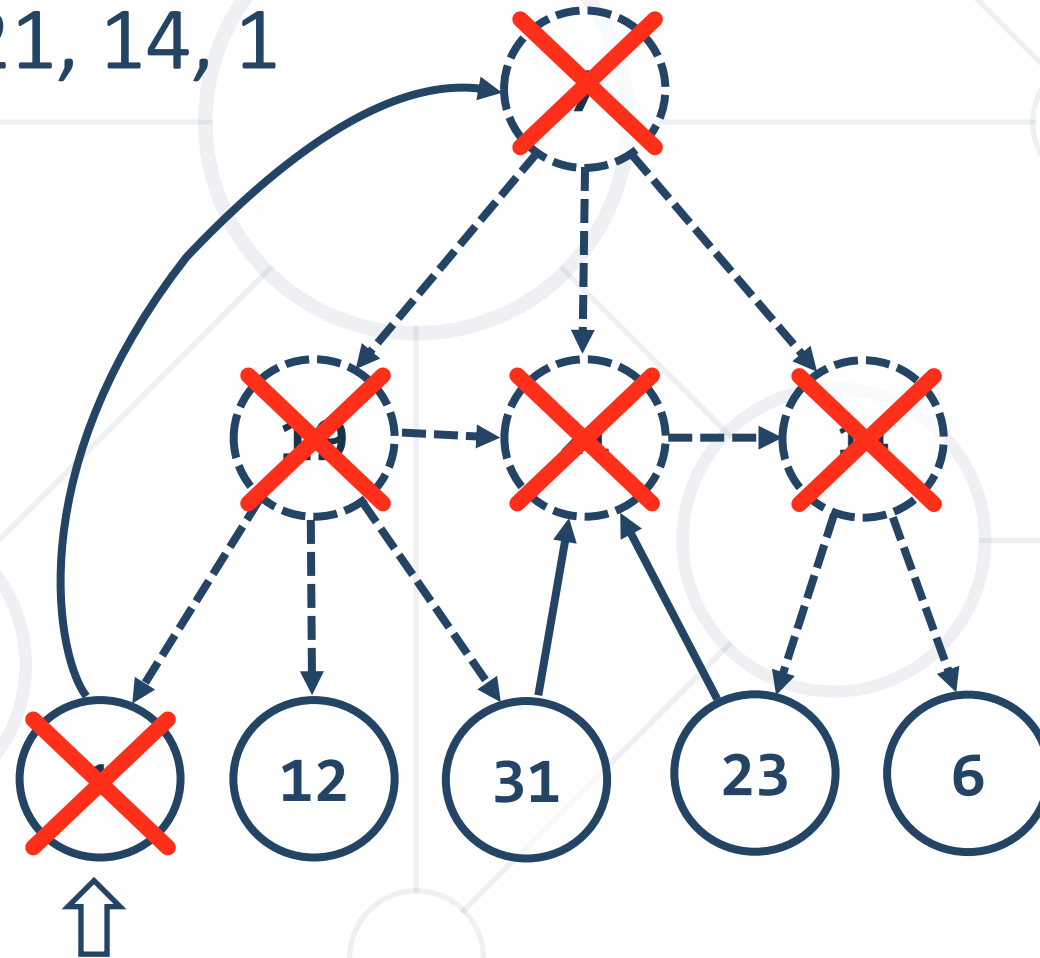
- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, 1, 12, 31, 23
- Output: 7, 19, 21, 14



-

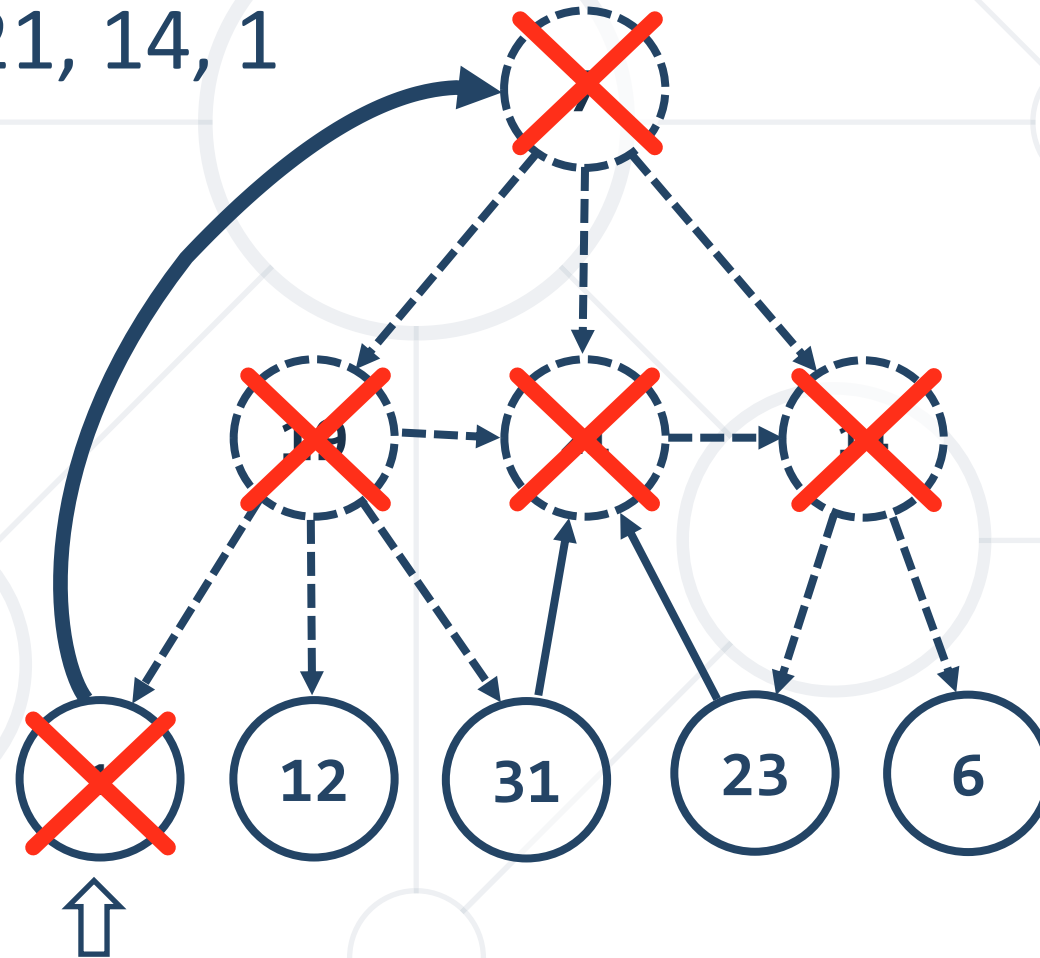
BFS in Action (Step 16)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, 12, 31, 23, 6
- Output: 7, 19, 21, 14, 1



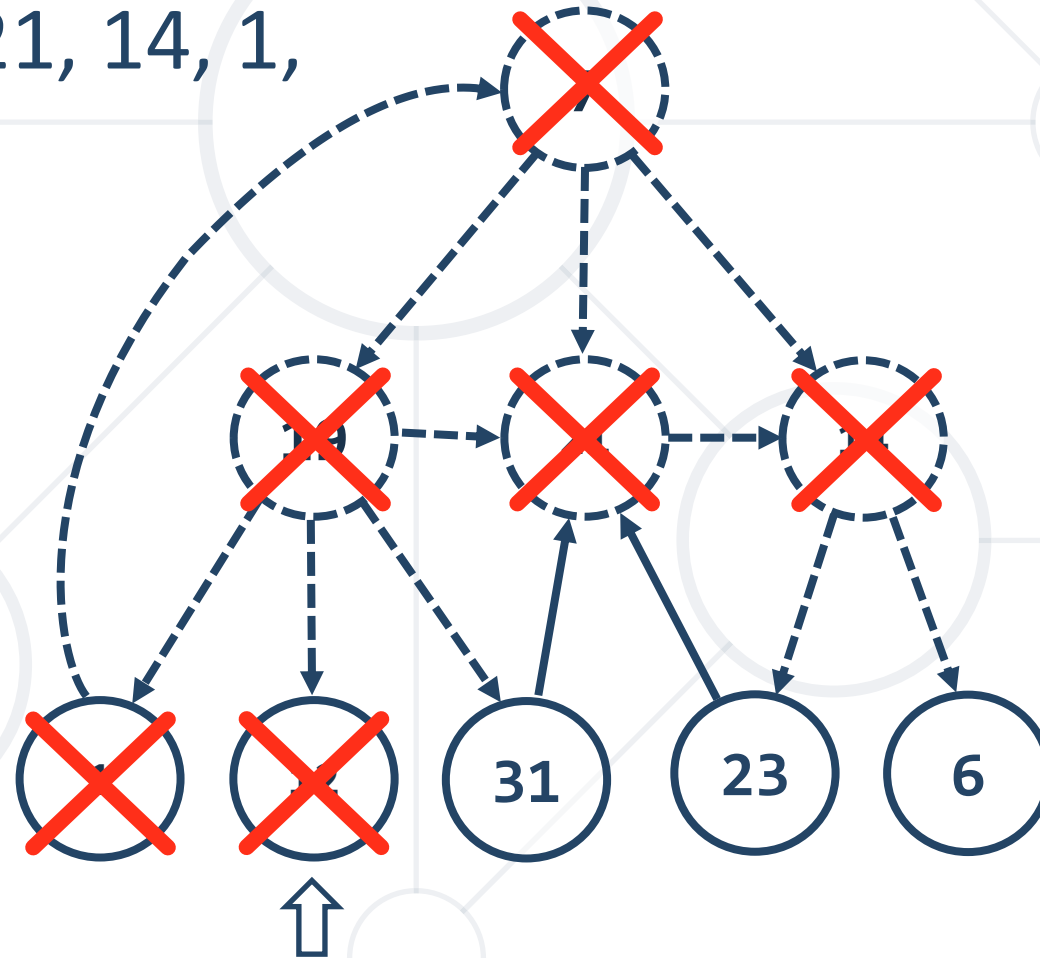
BFS in Action (Step 17)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, 12, 31, 23, 6
- Output: 7, 19, 21, 14, 1



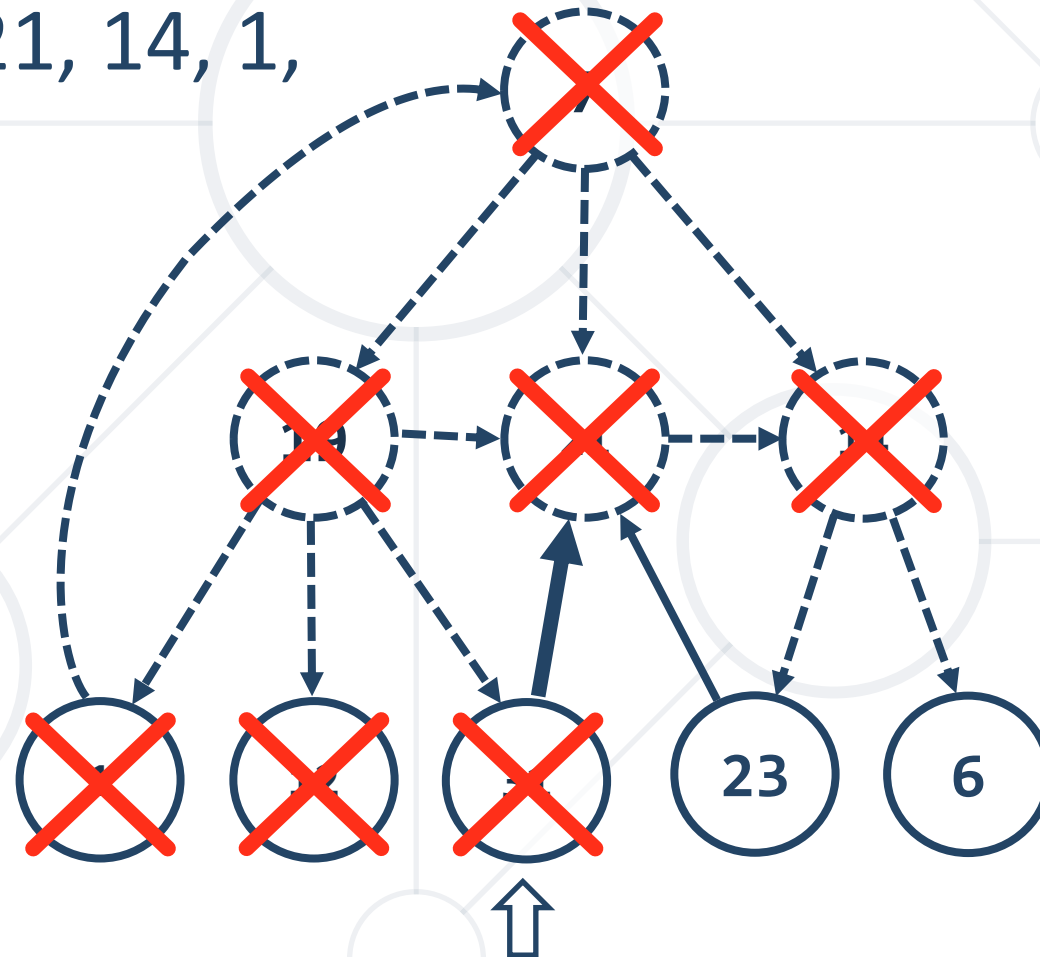
BFS in Action (Step 18)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, 31, 23, 6
- Output: 7, 19, 21, 14, 1,
12



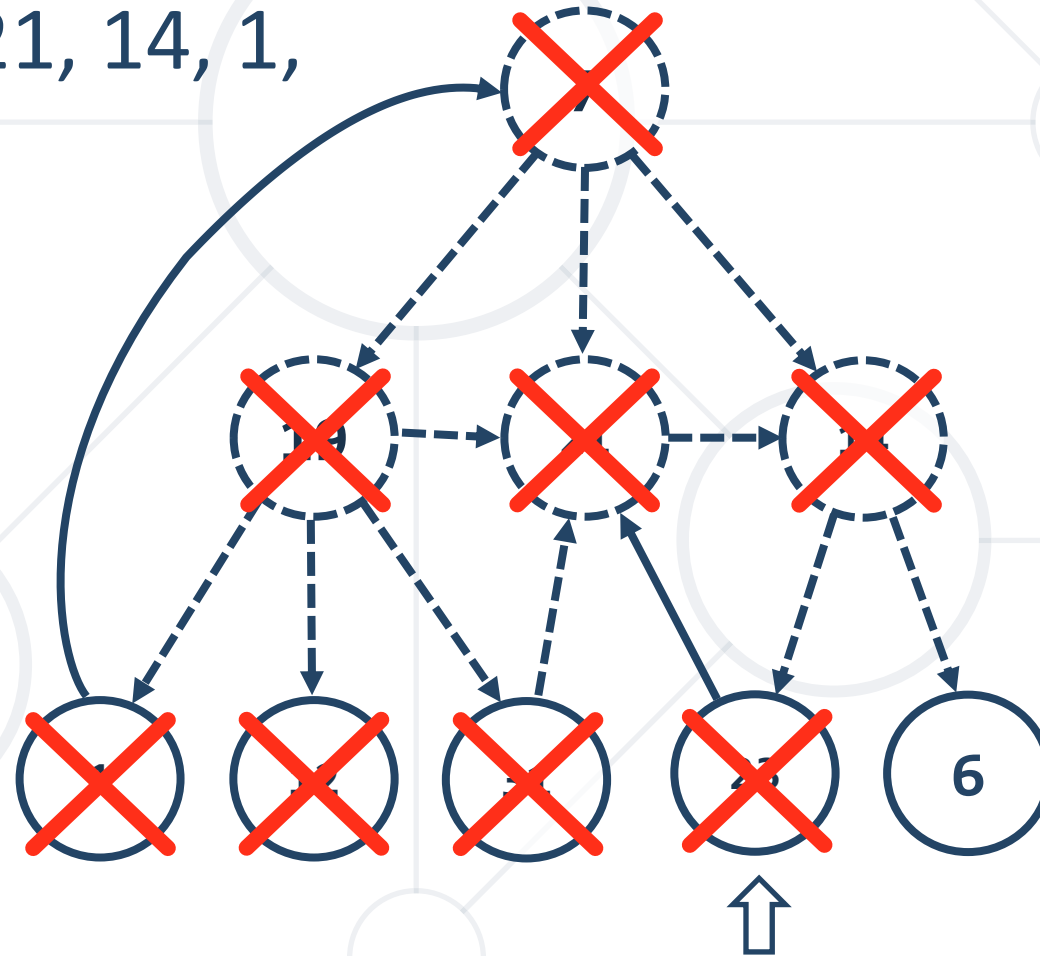
- [illegible]

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, 23, 6
- Output: 7, 19, 21, 14, 1, 12, 31



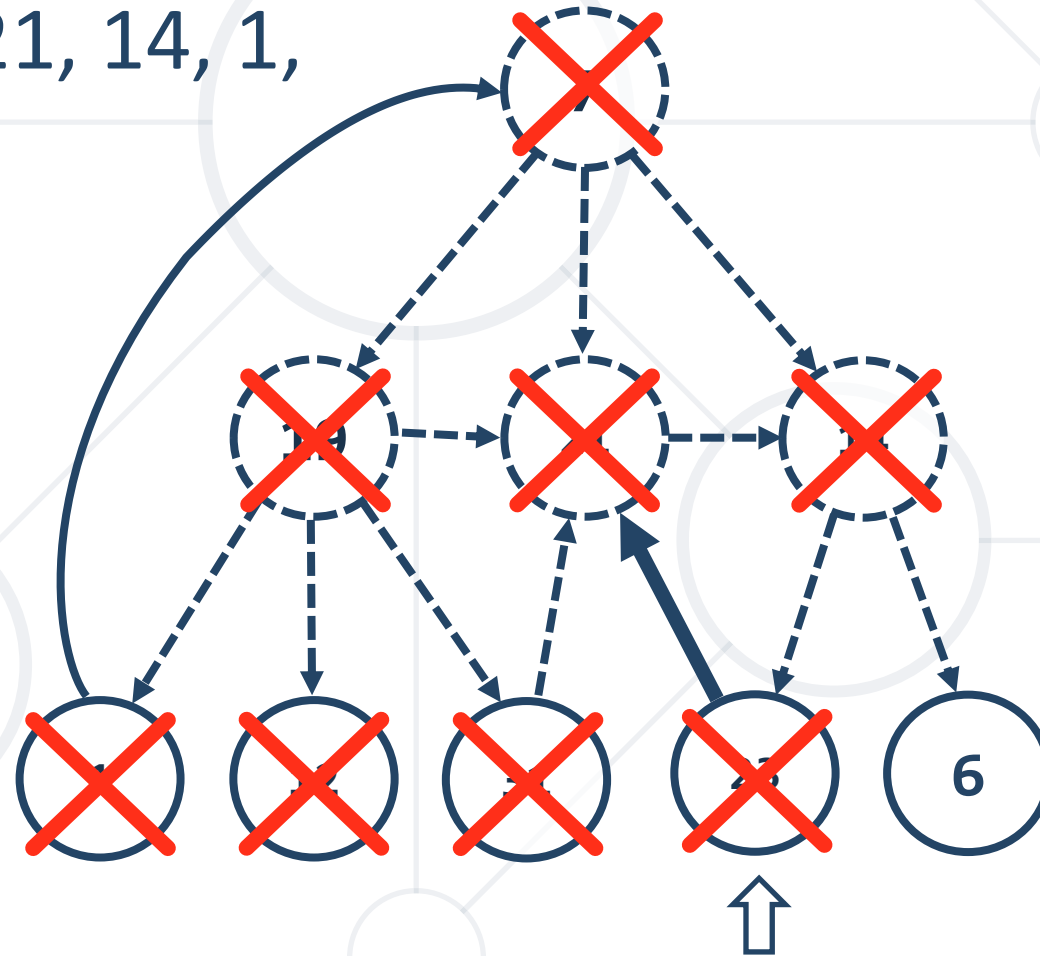
BFS in Action (Step 21)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, ~~23~~, 6
- Output: 7, 19, 21, 14, 1,
12, 31, 23



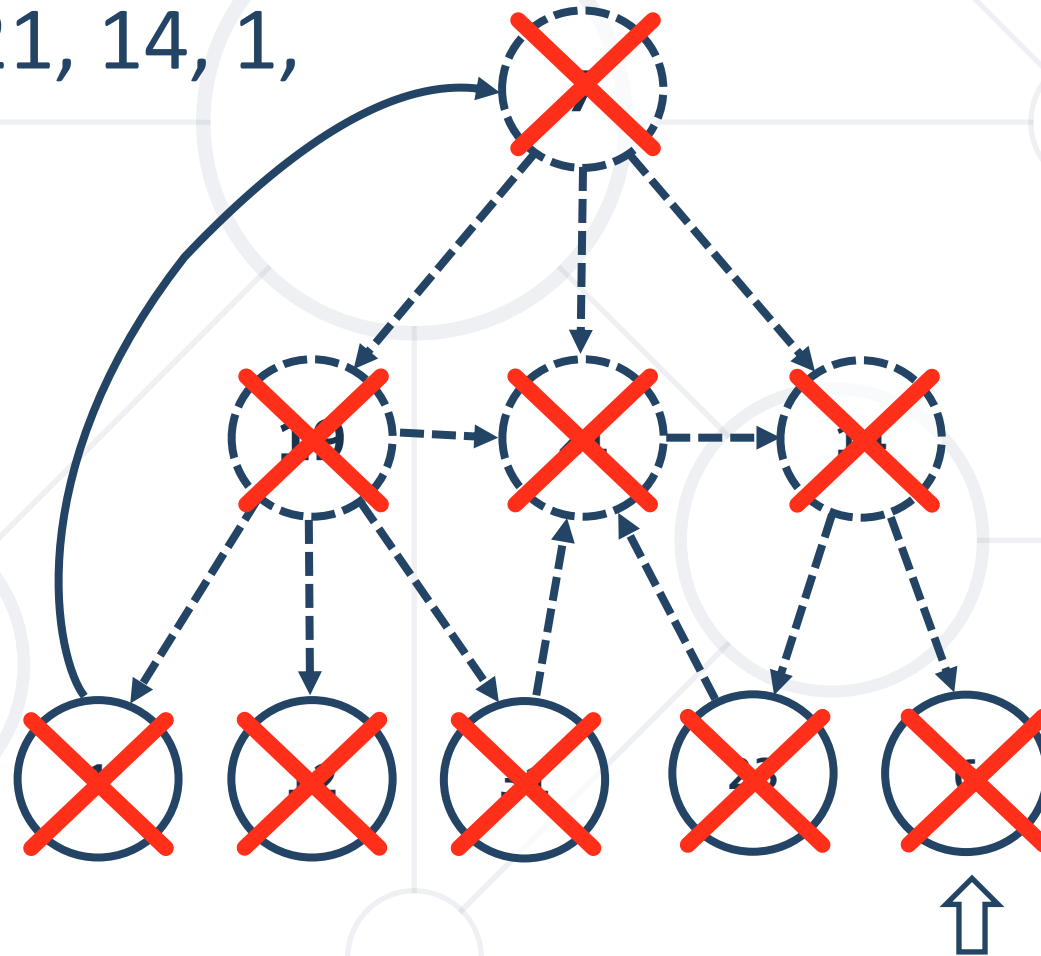
BFS in Action (Step 22)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, ~~23~~, 6
- Output: 7, 19, 21, 14, 1, 12, 31, 23



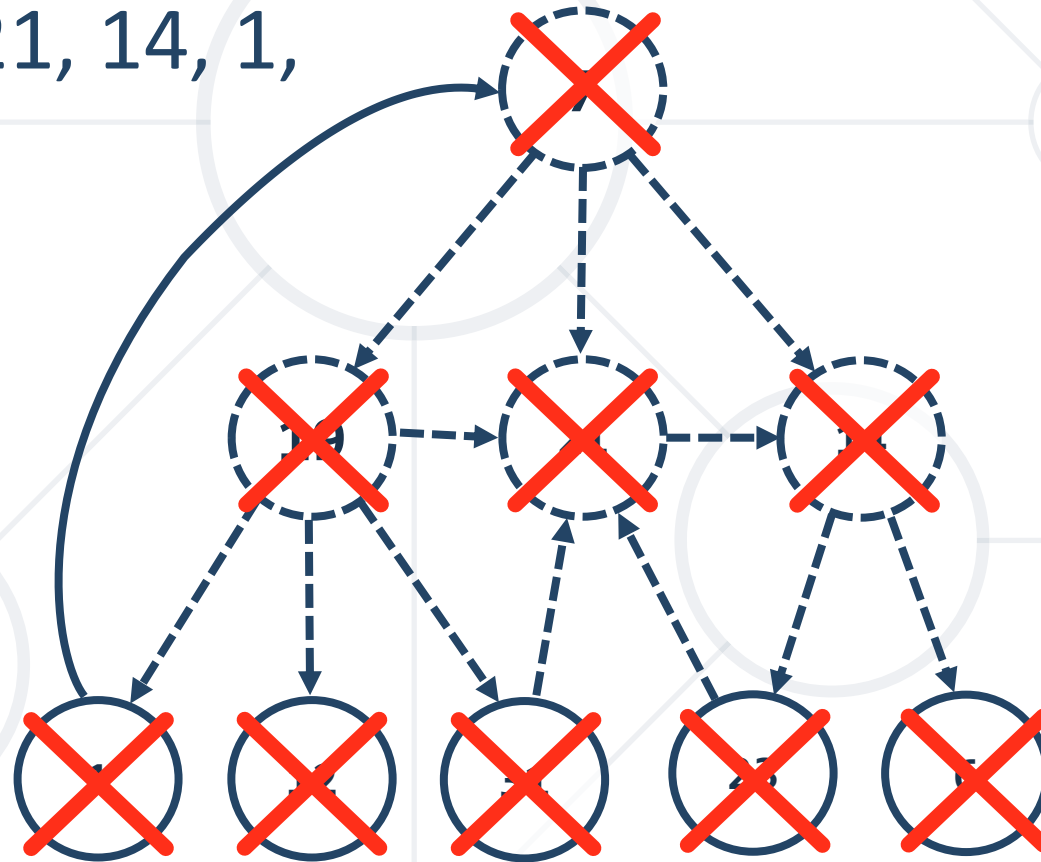
BFS in Action (Step 23)

- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, ~~23~~, ~~6~~
- Output: 7, 19, 21, 14, 1,
12, 31, 23, 6



BFS in Action (Step 24)

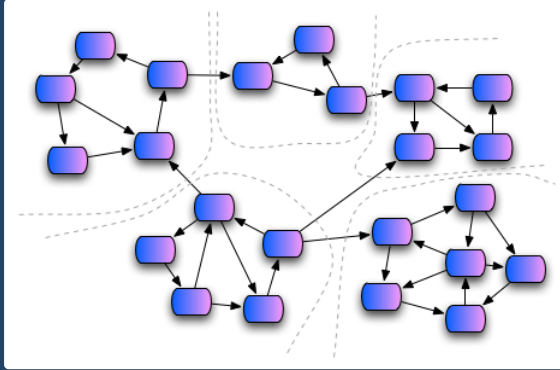
- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, ~~23~~, ~~6~~
- Output: 7, 19, 21, 14, 1,
12, 31, 23, 6



- What will happen if in the **Breadth-First Search (BFS)** algorithm we change the **queue** with a **stack**?
 - An iterative stack-based **Depth-First Search (DFS)**

```
bfs(node) {  
  queue ← node  
  visited[node] = true  
  while queue not empty  
    v ← queue  
    print v  
    for each child c of v  
      if not visited[c]  
        queue ← c  
        visited[c] = true  
}
```

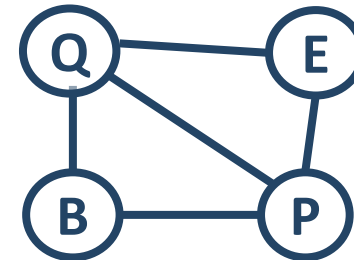
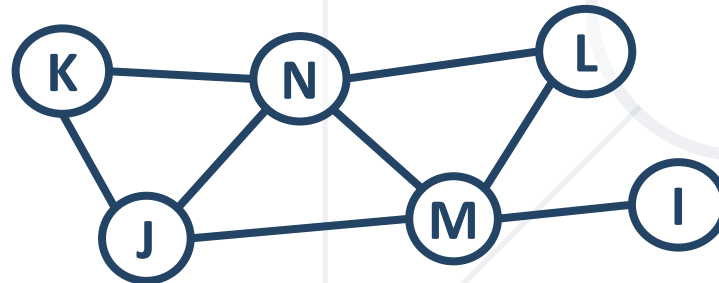
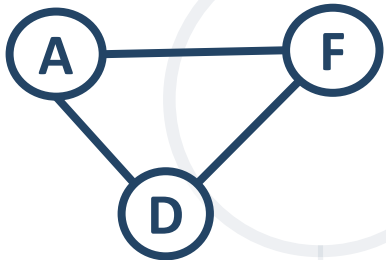
```
dfs(node) {  
  stack ← node  
  visited[node] = true  
  while stack not empty  
    v ← stack  
    print v  
    for each child c of v  
      if not visited[c]  
        stack ← c  
        visited[c] = true  
}
```



Graph Connectivity

Finding the Connected Components

- **Connected component** of undirected graph
 - A sub-graph in which **any two nodes are connected** to each other by paths
 - E.g. the graph below consists of 3 connected components



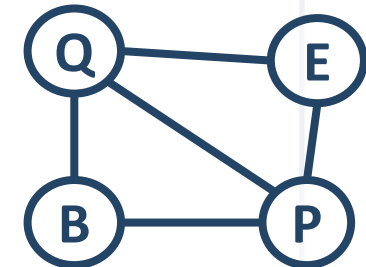
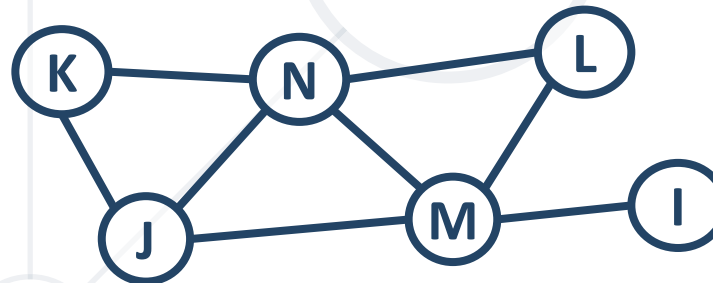
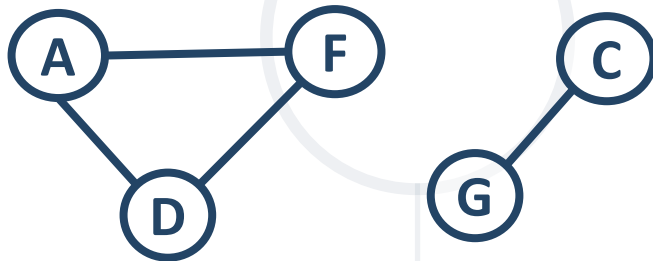
Finding All Graph Connected Components

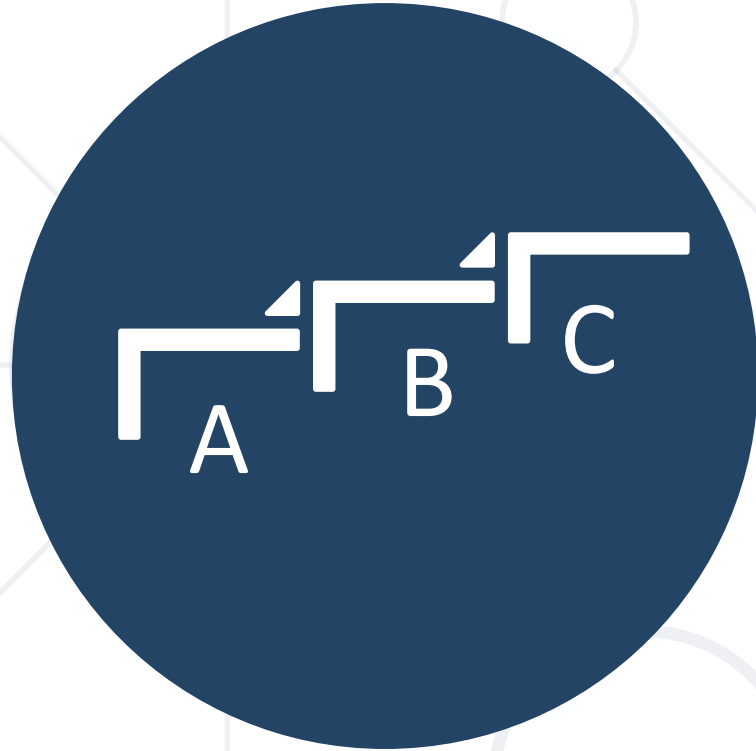
- Finding the connected components in a graph
 - Loop through all nodes and start a **DFS** / **BFS** traversing from any **unvisited** node
- Each time you start a new traversal
 - You find a new connected component

Graph Connected Components: Algorithm

```
visited[] = false;  
foreach node from graph G {  
    if (not visited[node]) {  
        dfs(node);  
        countOfComponents++;  
    }  
}
```

```
dfs(node) {  
    if (not visited[node]) {  
        visited[node] = true;  
        foreach c in node.children  
            dfs(c);  
    }  
}
```





Topological Sorting

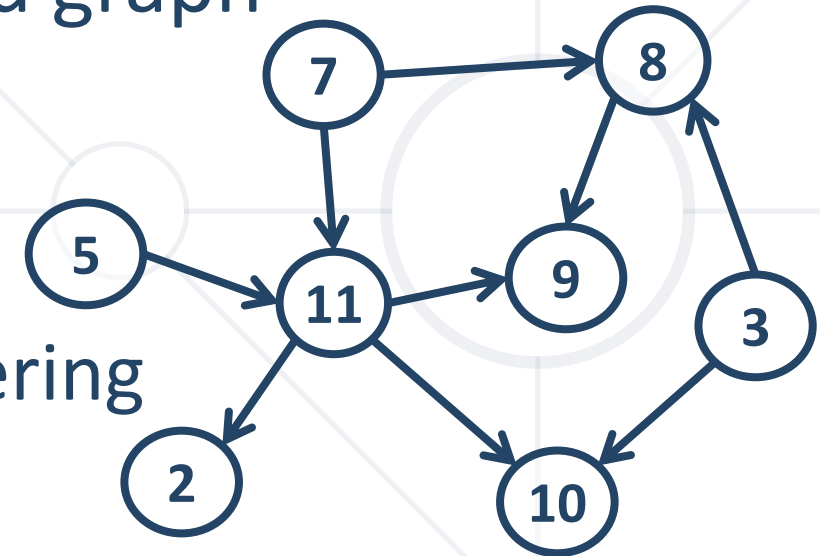
Ordering a Graph by Set of Dependencies

- **Topological sorting** (ordering) of a directed graph

- Linear ordering of its vertices, such that
- For every directed edge from vertex u to vertex v , u comes before v in the ordering

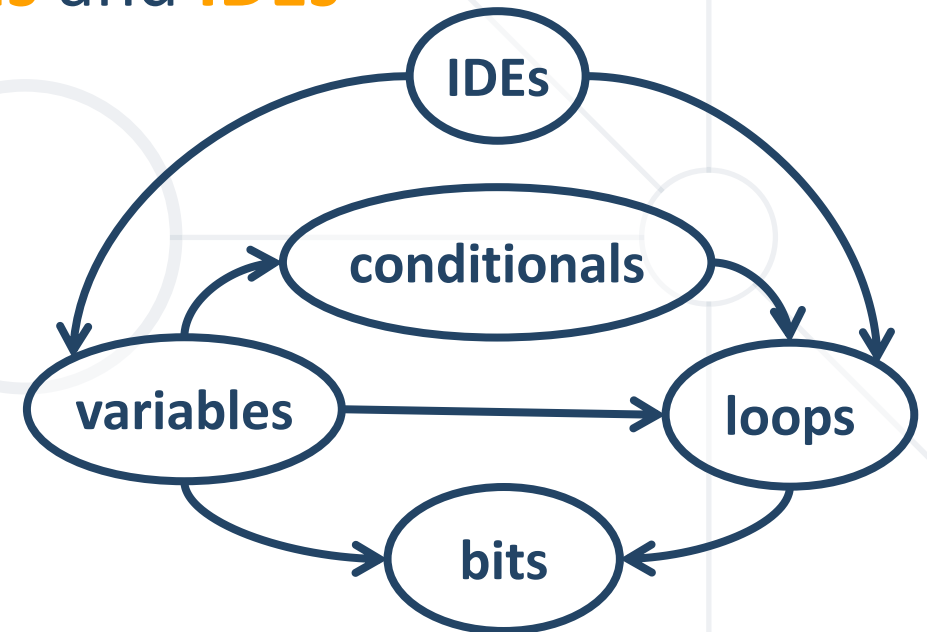
- Example:

- $7 \rightarrow 5 \rightarrow 3 \rightarrow 11 \rightarrow 8 \rightarrow 2 \rightarrow 9 \rightarrow 10$
- $3 \rightarrow 5 \rightarrow 7 \rightarrow 8 \rightarrow 11 \rightarrow 2 \rightarrow 9 \rightarrow 10$
- $5 \rightarrow 7 \rightarrow 3 \rightarrow 8 \rightarrow 11 \rightarrow 10 \rightarrow 9 \rightarrow 2$



Topological Sorting – Example

- We have a set of learning **topics** with **dependencies**
 - Order the topics in such order that all dependencies are met
- Example:
 - **Loops** depend on **variables**, **conditionals** and **IDEs**
 - **Variables** depend on **IDEs**
 - **Bits** depend on **variables** and **loops**
 - **Conditionals** depend on **variables**
- Ordering:
 - **IDEs** → **variables** → **loops** → **bits**



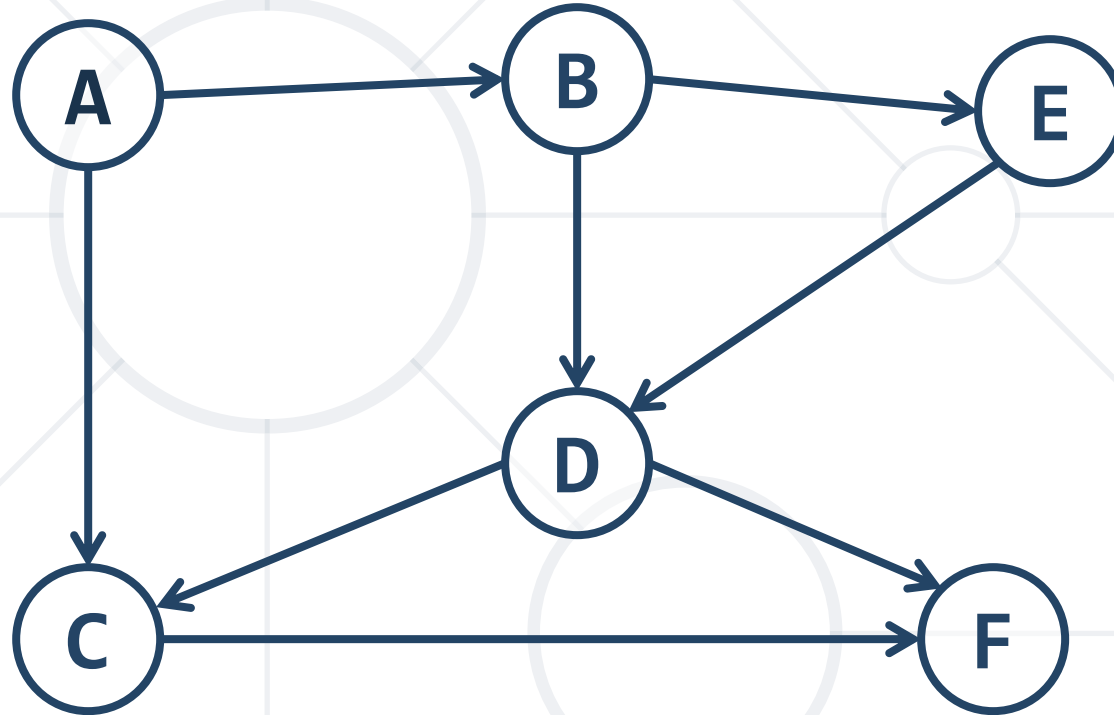
- Rules
 - Undirected graphs cannot be sorted
 - Graphs with cycles cannot be sorted
 - Sorting is not unique
 - Various sorting algorithms exists, and they give different results

- **Source removal top-sort algorithm**
 - Create an empty list
 - Repeat until the graph is empty:
 - Find a node without incoming edges
 - Print this node
 - Remove the edge from the graph

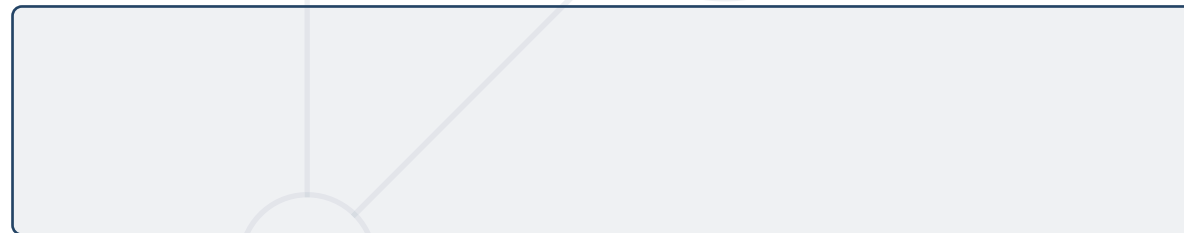
Source Removal Algorithm

- $L \leftarrow$ empty list that will hold the sorted elements (output)
- $S \leftarrow$ set of all nodes with no incoming edges
- while S is non-empty do
 - remove some node n from S
 - append n to L
 - for each node m with an edge $e: \{ n \text{ through } m \}$
 - remove edge e from the graph
 - if m has no other incoming edges then
 - insert m into S
- if graph is empty
 - return L (a topologically sorted order)
- else
 - return "Error: graph has at least one cycle"

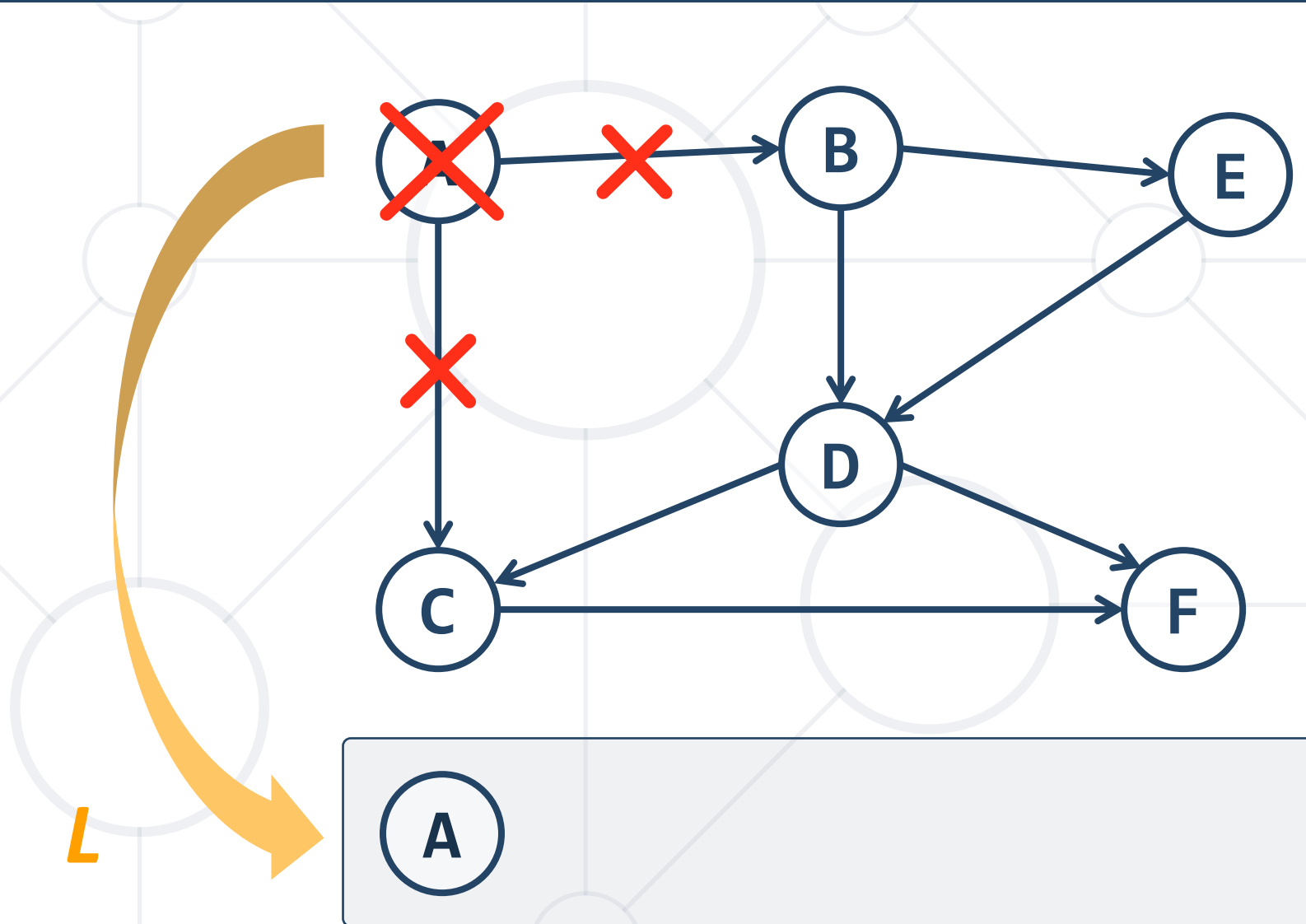
Step #1: Find a Node with No Incoming Edges



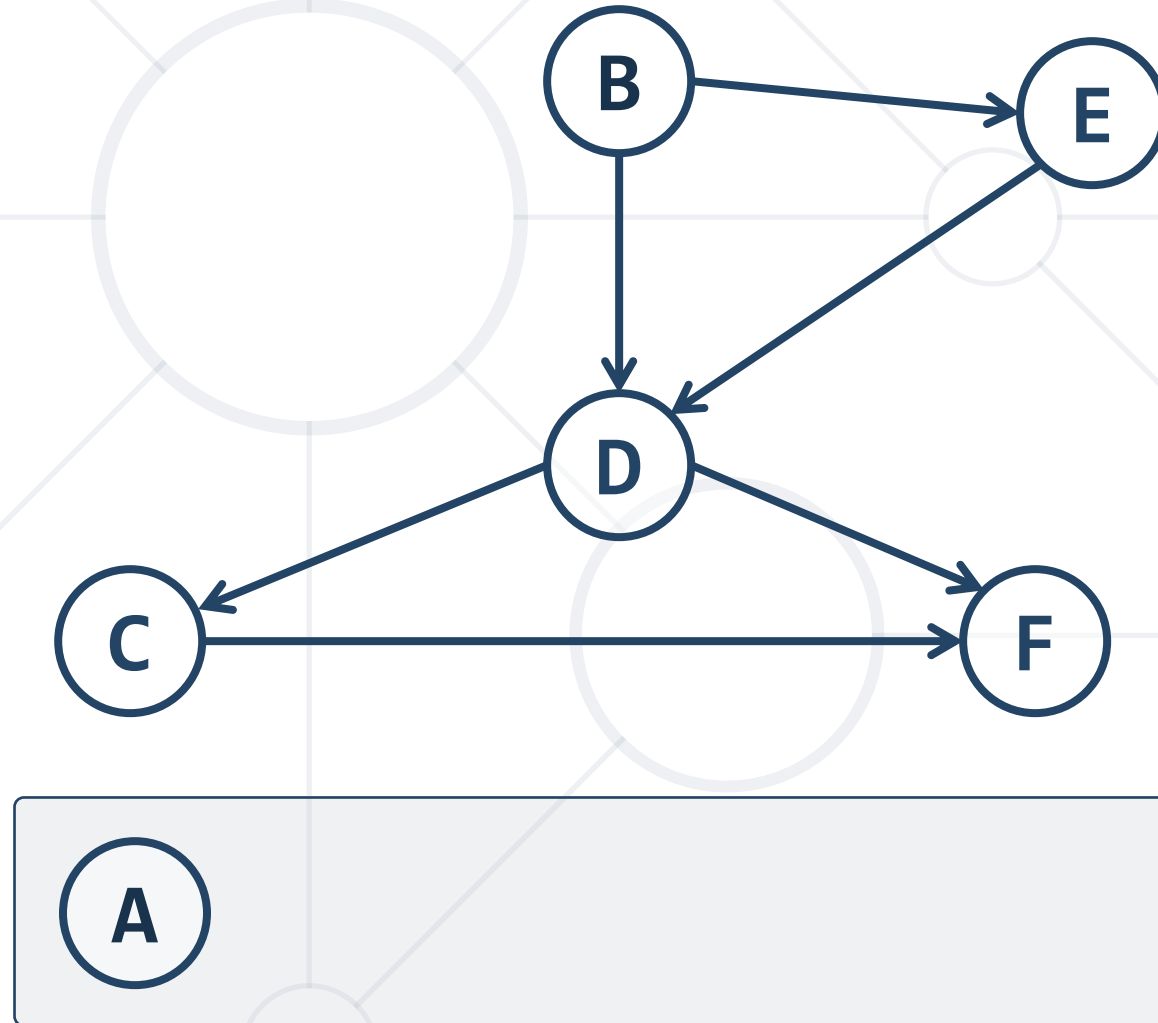
L



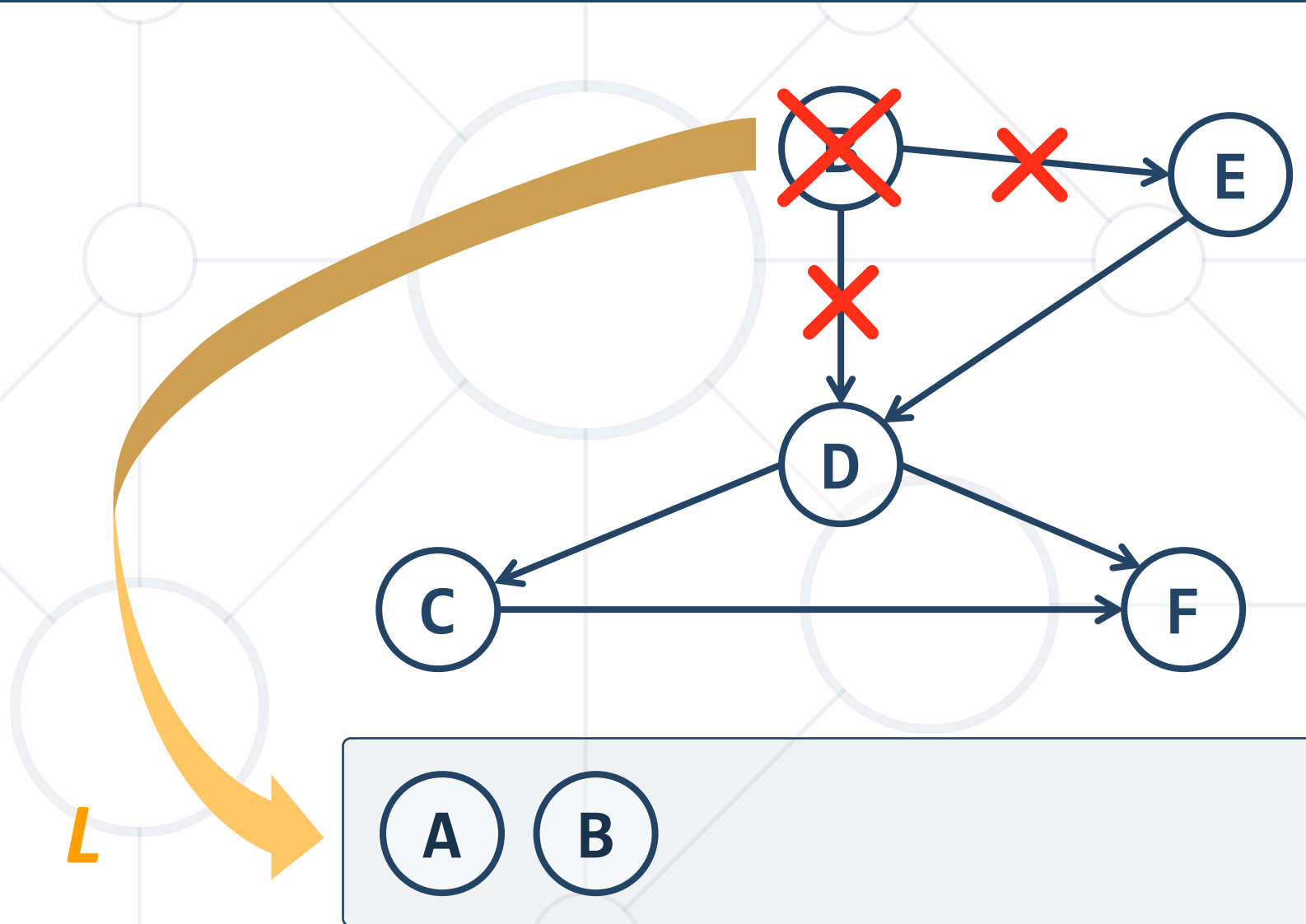
Step #2: Remove Node A with Its Edges



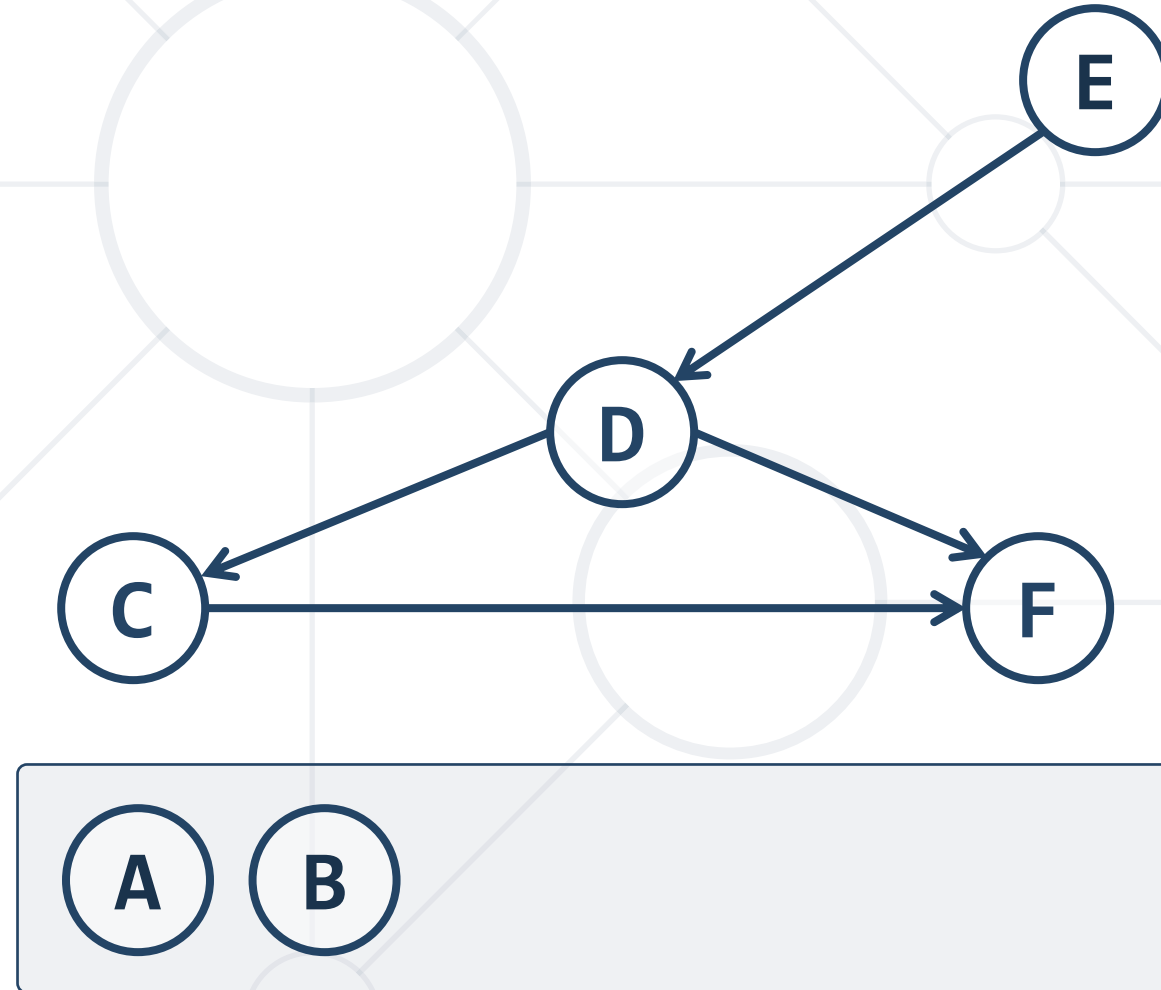
Step #3: Find a Node with No Incoming Edges



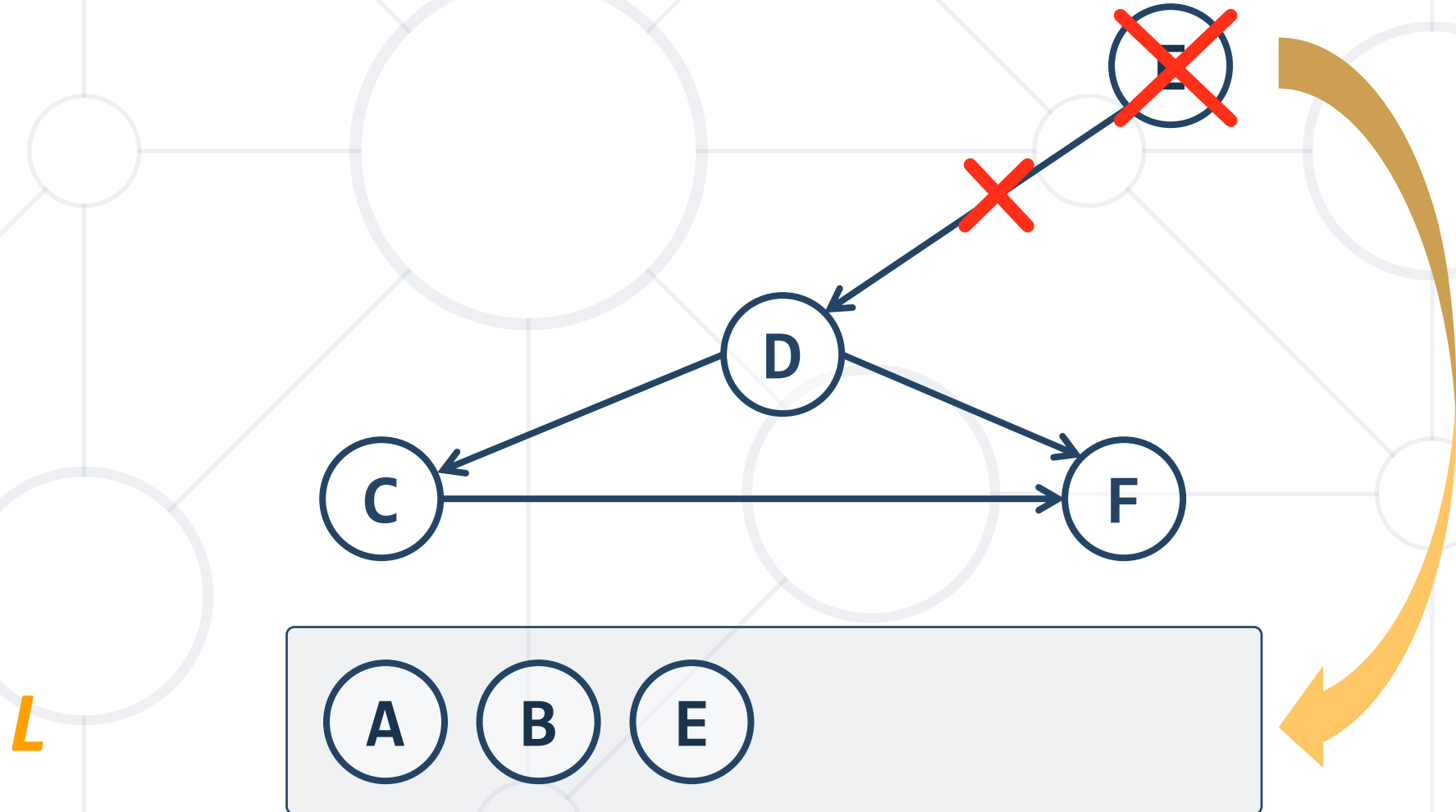
Step #4: Remove Node B with Its Edges



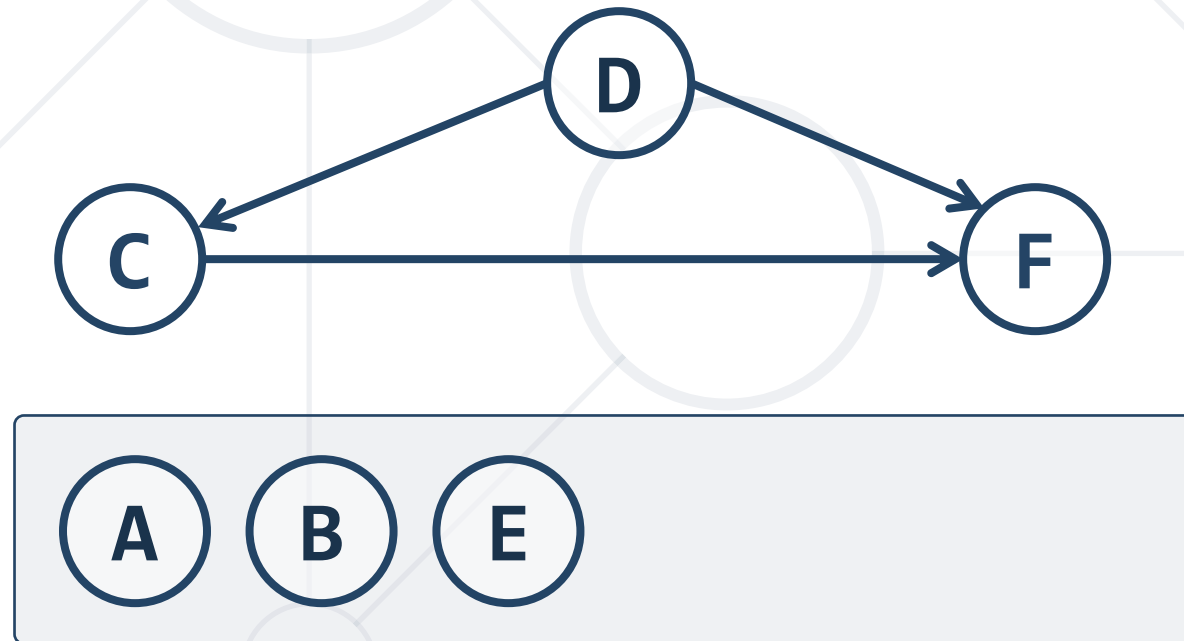
Step #5: Find a Node with No Incoming Edges



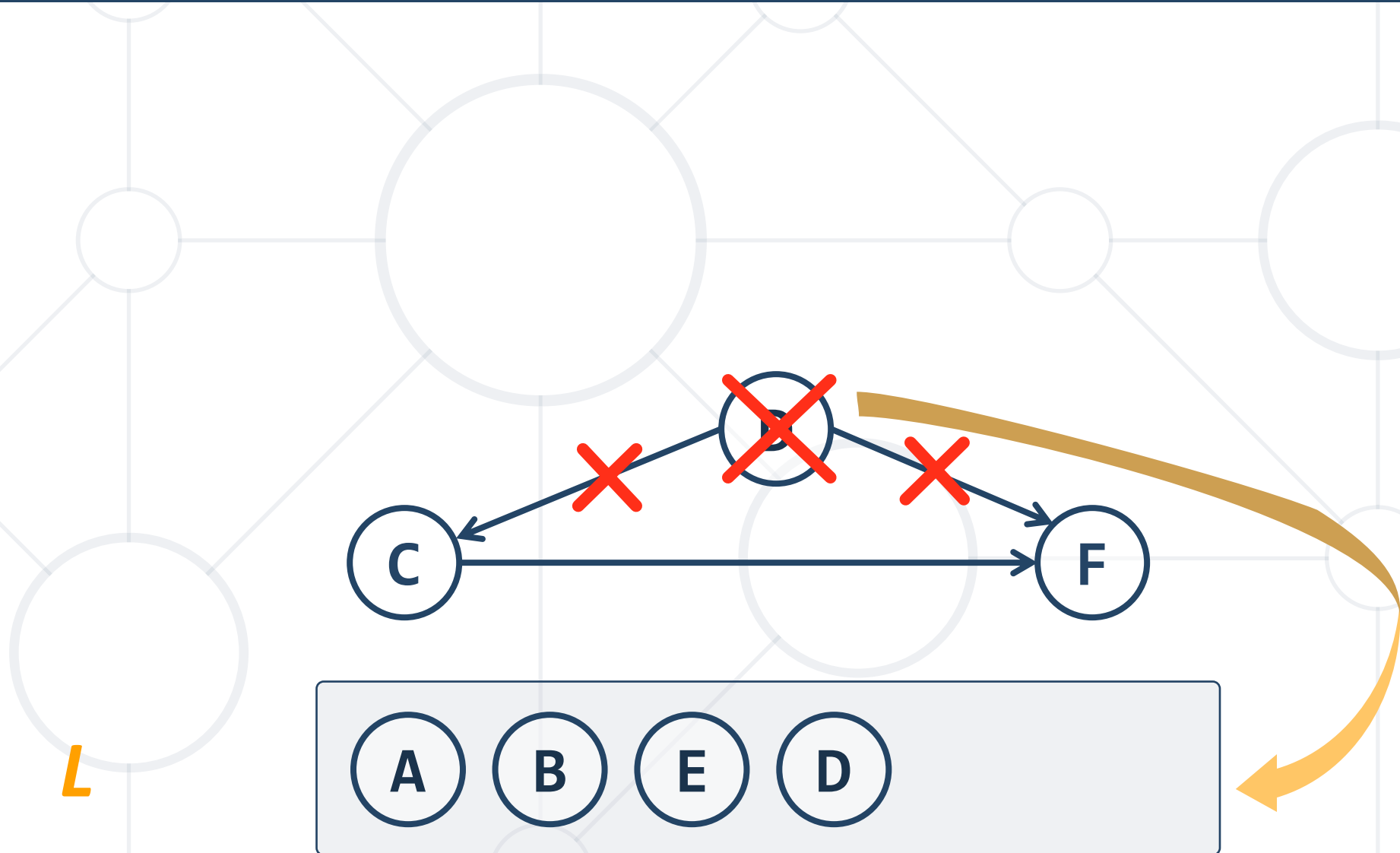
Step #6: Remove Node E with Its Edges



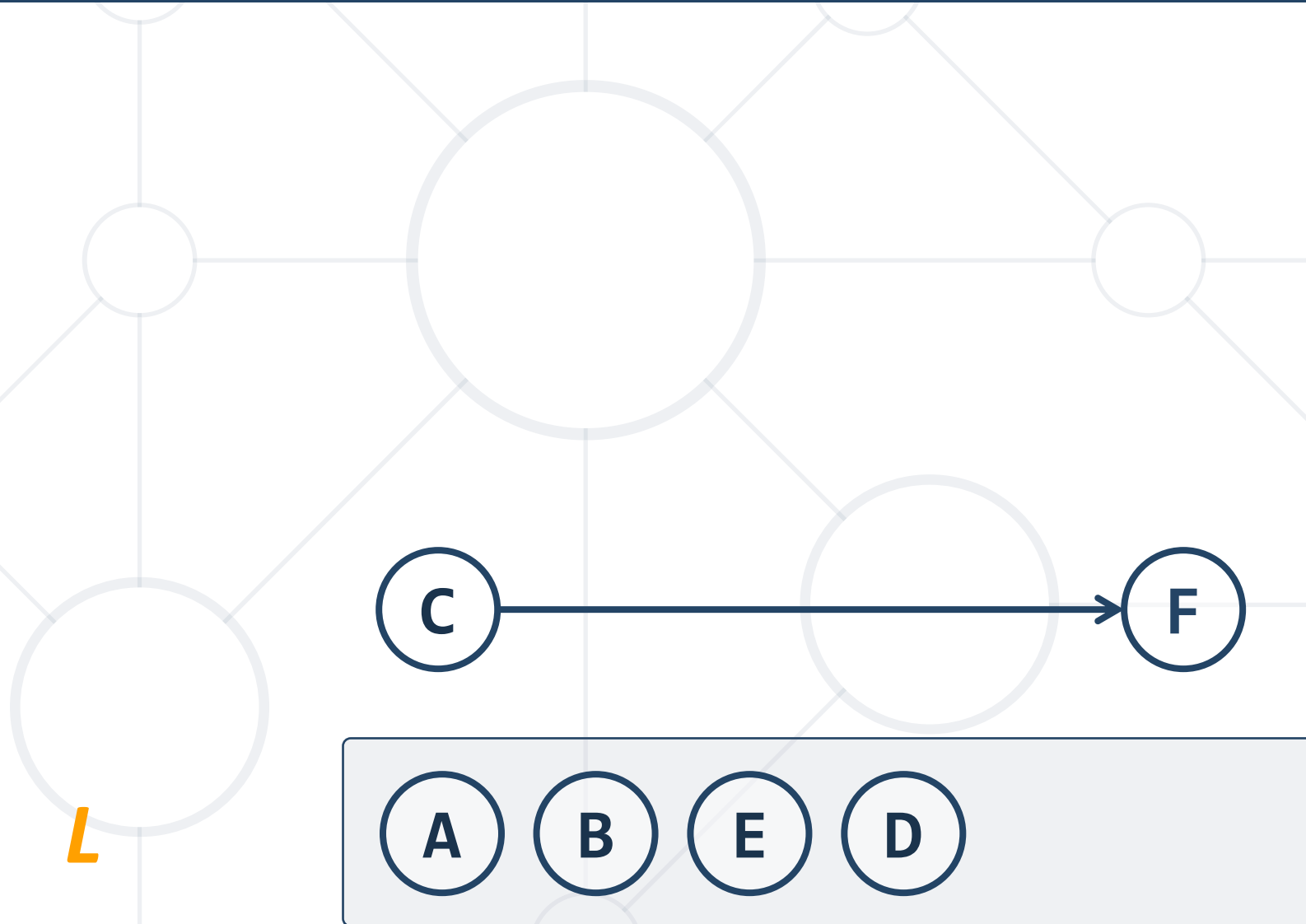
Step #7: Find a Node with No Incoming Edges



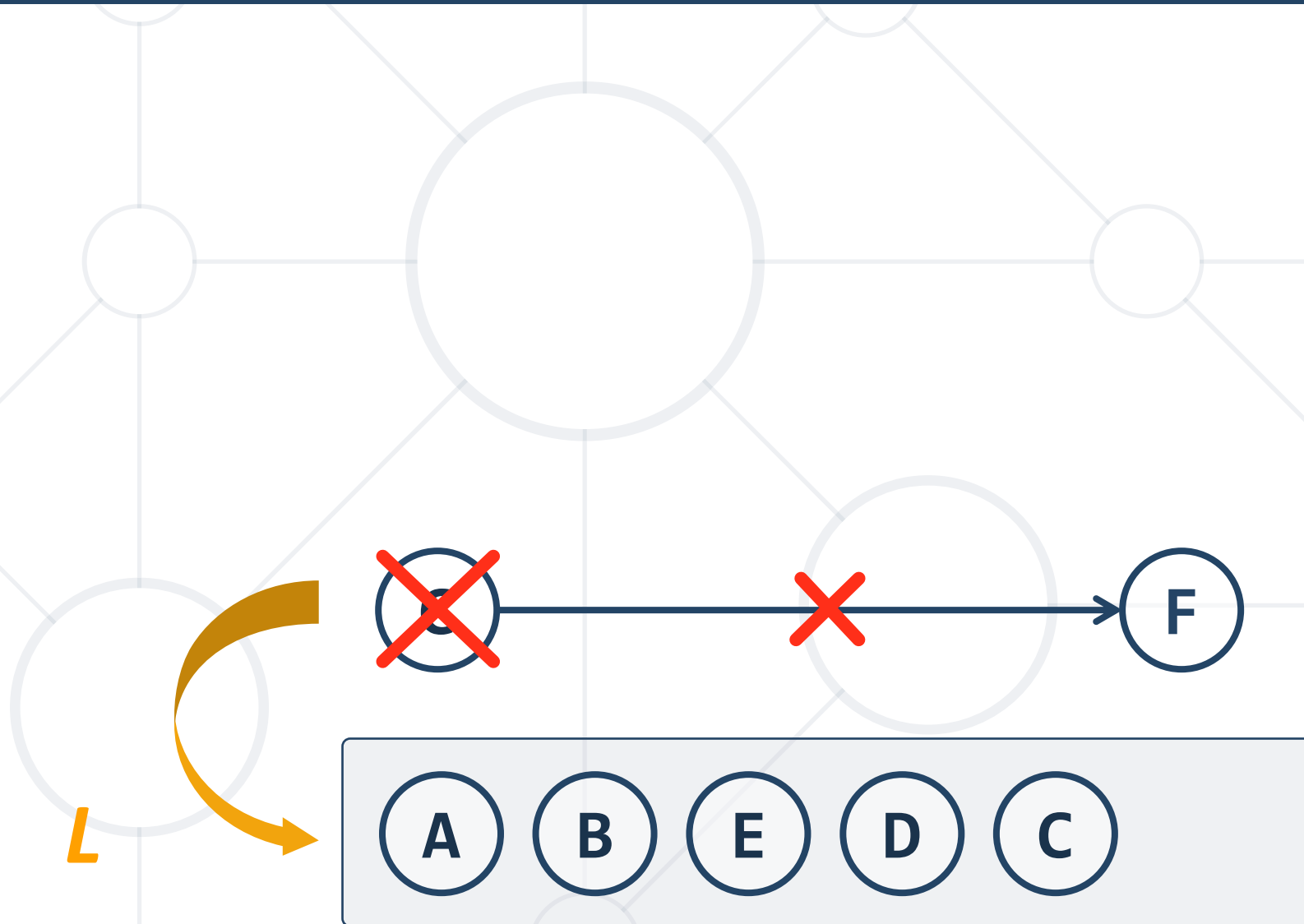
Step #8: Remove Node D with Its Edges



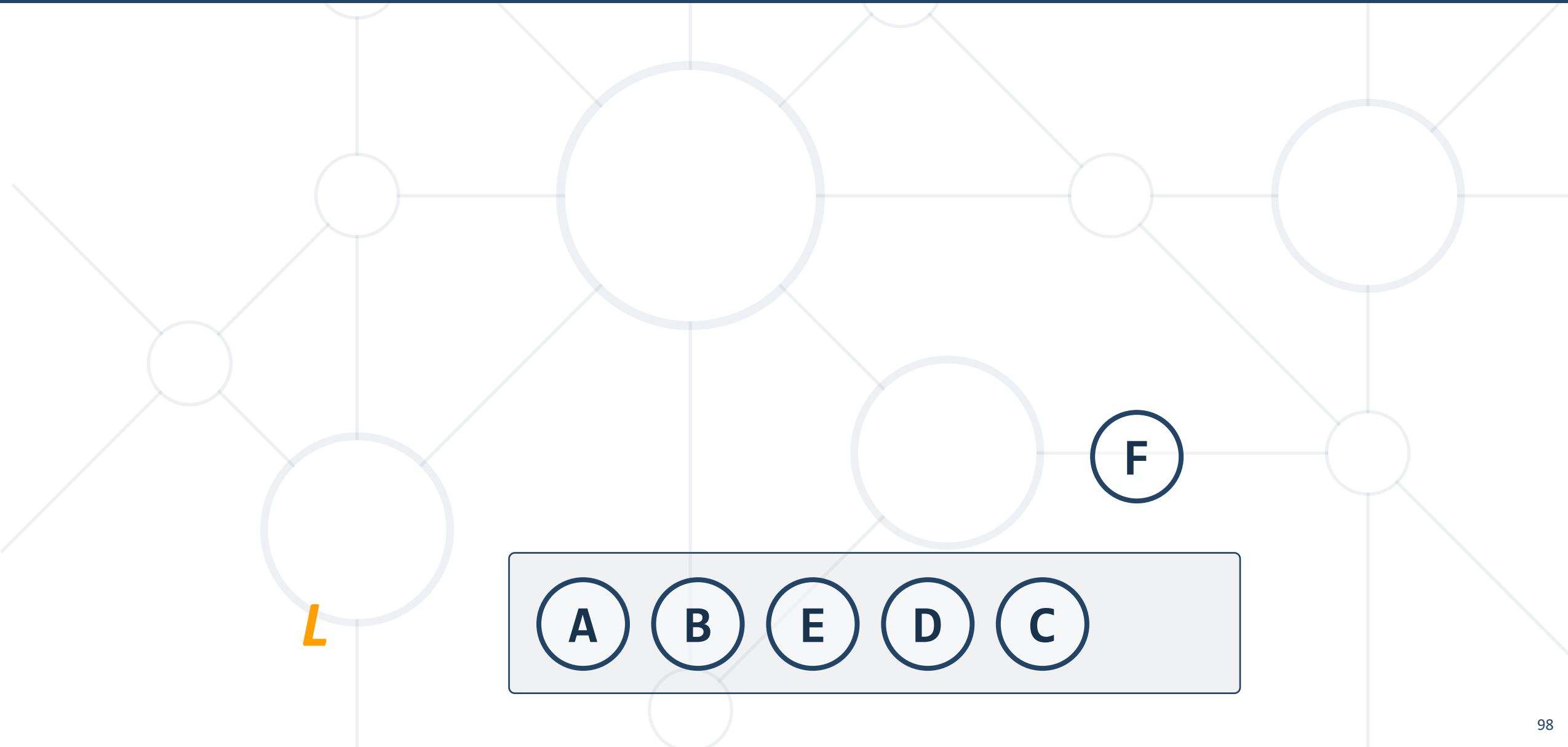
Step #9: Find a Node with No Incoming Edges



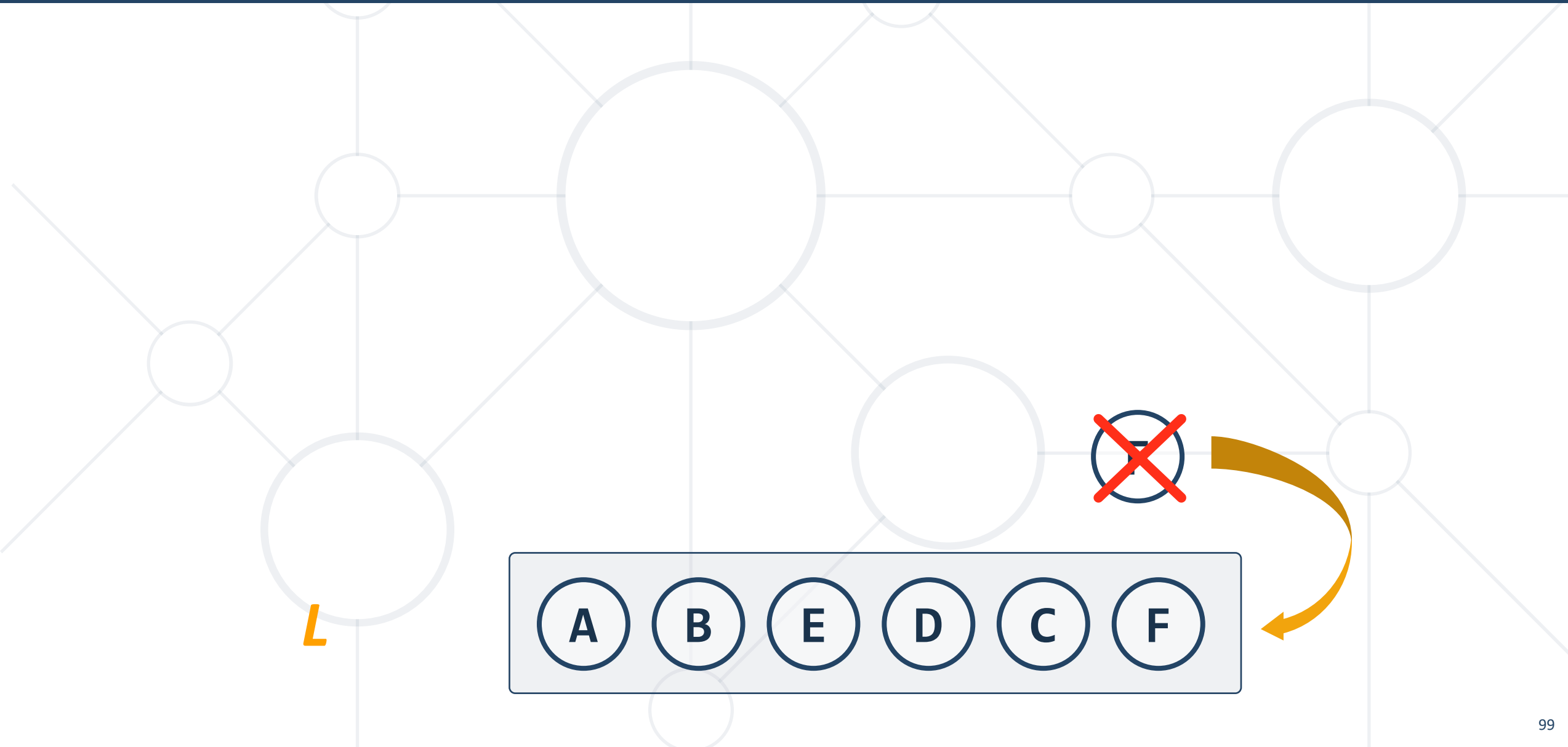
Step #10: Remove Node C with Its Edges



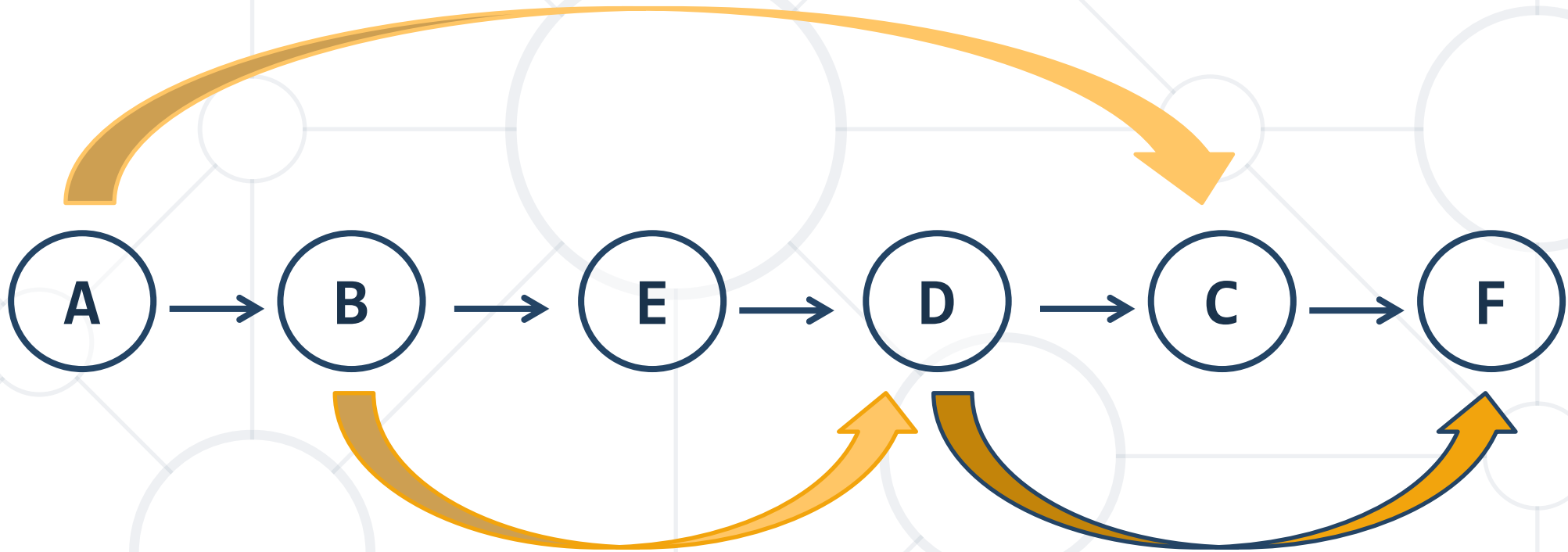
Step #11: Find a Node with No Incoming Edges



Step #12: Remove Node F with Its Edges



Result: Topological Sorting

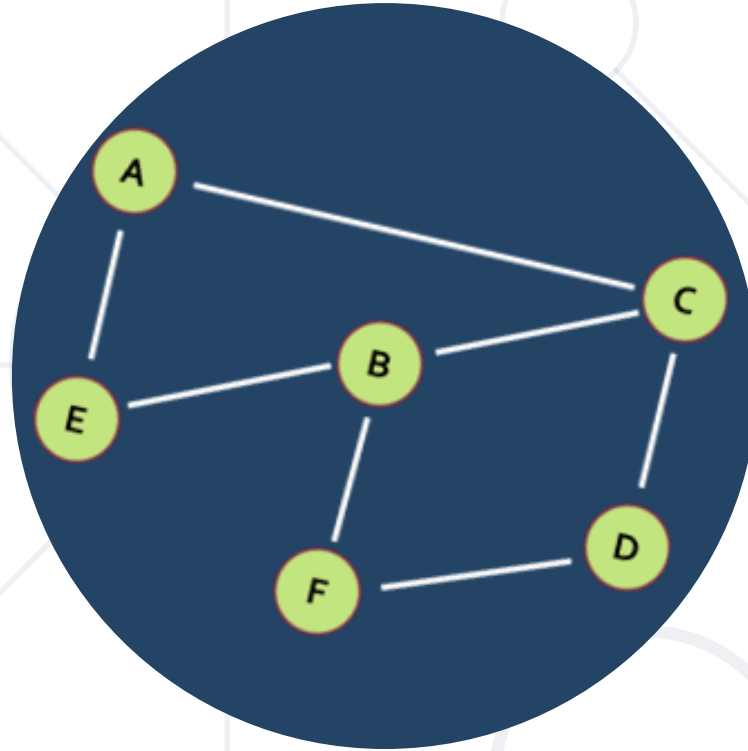


Topological Sorting: DFS Algorithm

```
sortedNodes = { }    // linked list to hold the result
visitedNodes = { }   // set of already visited nodes
foreach node in graphNodes
    topSortDFS(node)
topSortDFS(node)
    if node ∉ visitedNodes
        visitedNodes ← node
        for each child c of node
            TopSortDFS(c)
        insert node upfront in the sortedNodes
```

TopSort: DFS Algorithm + Cycle Detection

```
sortedNodes = { }    // linked list to hold the result
visitedNodes = { }   // set of already visited nodes
cycleNodes = { }     // set of nodes in the current DFS cycle
foreach node in graphNodes
    topSortDFS(node)
topSortDFS(node)
    if node ∈ cycleNodes
        return "Error: cycle detected"
    if node ∉ visitedNodes
        visitedNodes ← node
        cycleNodes ← node
        for each child c of node
            topSortDFS(c)
        remove node from cycleNodes
        insert node upfront in the sortedNodes
```

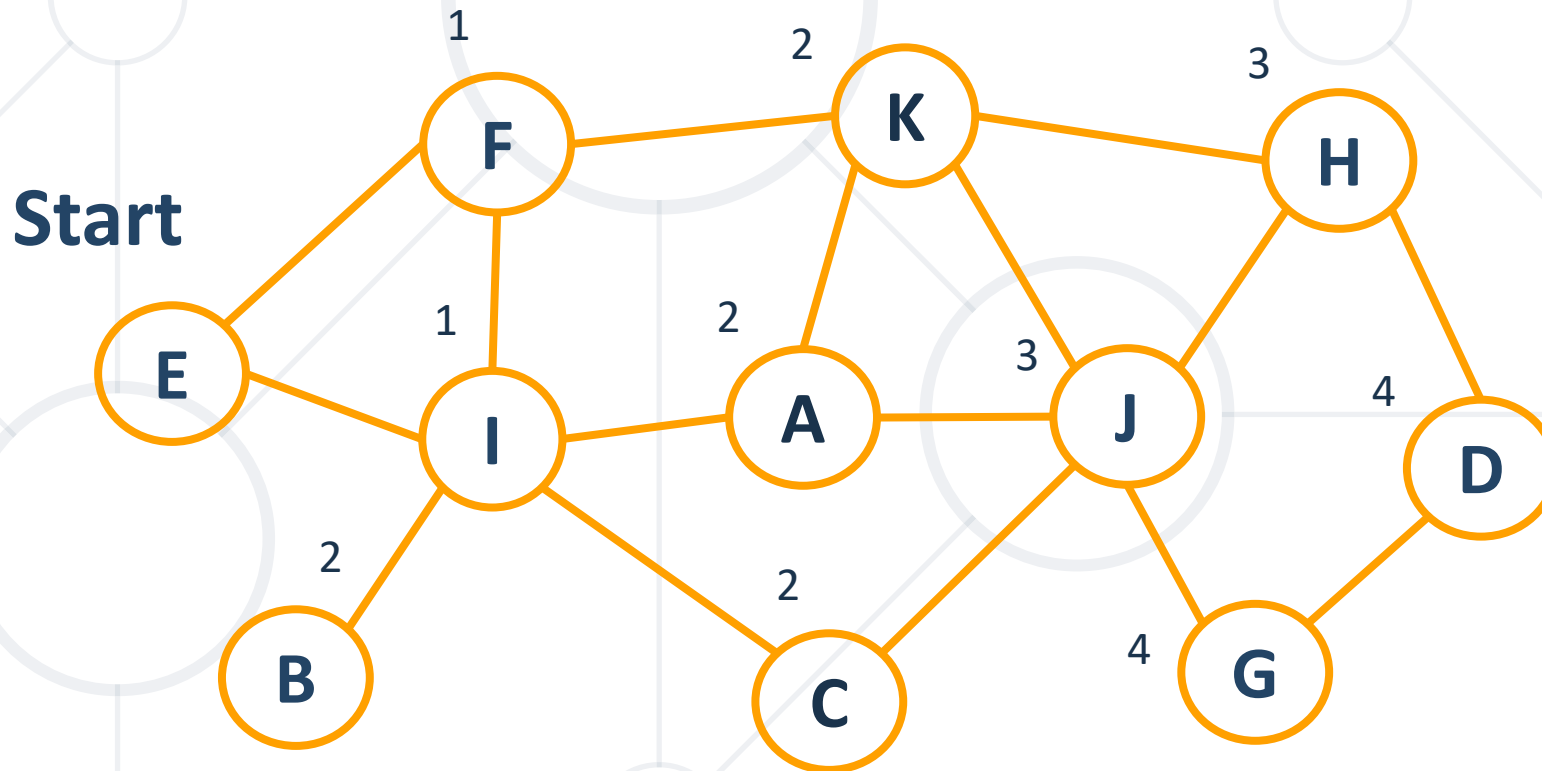


Shortest Path

Shortest Path in Unweighted Graph

Shortest Path in Unweighted Graph

- In **unweighted** graphs finding the **shortest path** can be done with **BFS** (all edges have the same weight):

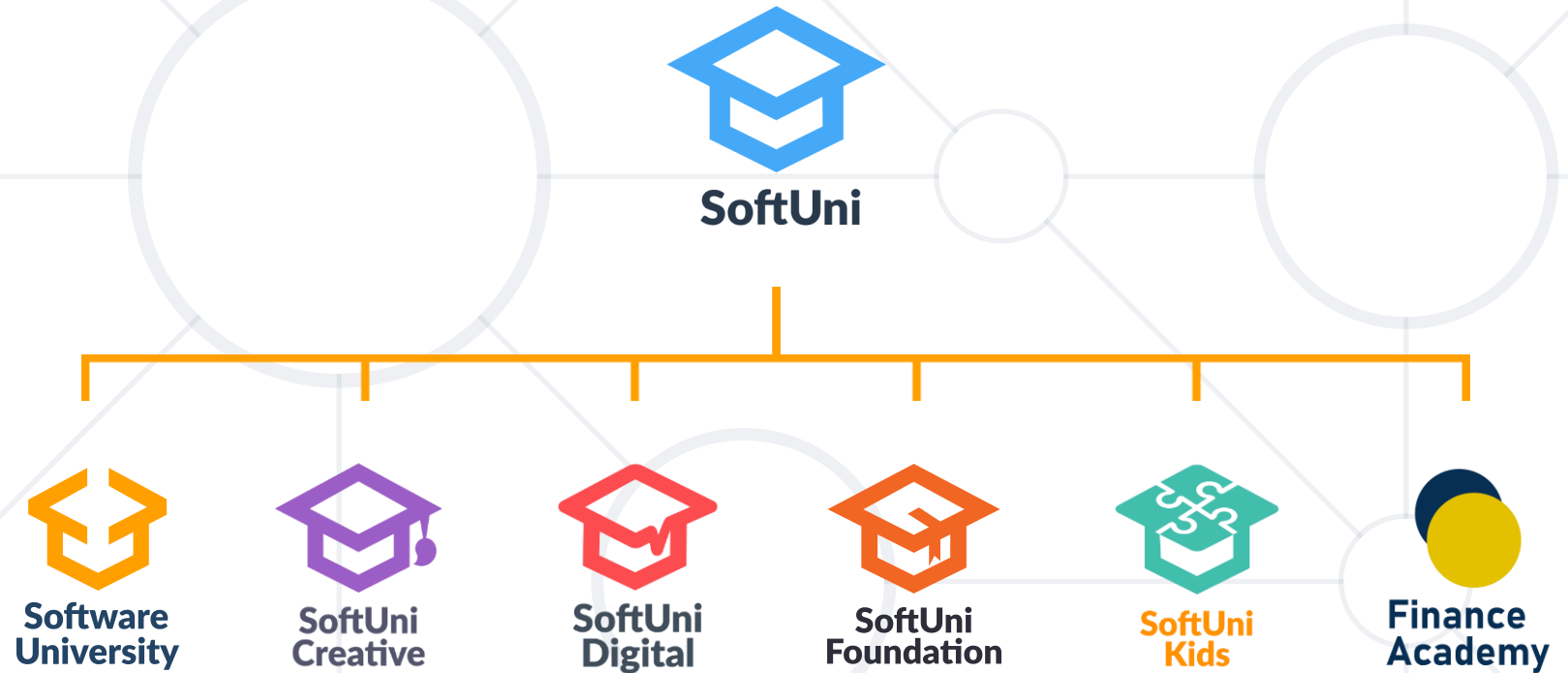



```
bfs(G, start, end)
    visited[start] = true
    queue.enqueue(start)
    while (!queue.isEmpty())
        v = queue.dequeue()
        if v is end
            return v
        for all edges from v to w in G.adjacentEdges(v) do
            if w is not labeled as discovered then
                label w as discovered
                w.parent = v
                queue.enqueue(w)
```

- Representing graphs in memory
 - **Adjacency list** holding the children for each node
 - **Adjacency matrix**
 - **List of edges**
 - Numbering the nodes for faster access
- Depth-First Search (**DFS**) – recursive in-depth traversal
- Breadth-First Search (**BFS**) – in-width traversal with a queue
- **Topological sorting**
- **Shortest path** in unweighted graph



Questions?



SoftUni Diamond Partners

**SUPER
HOSTING
.BG**



**Coca-Cola HBC
Bulgaria**



POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers



AMBITIONED

 **DRAFT
KINGS**



**SOFTWARE
GROUP**

createX



Postbank
Решения за твоето утре

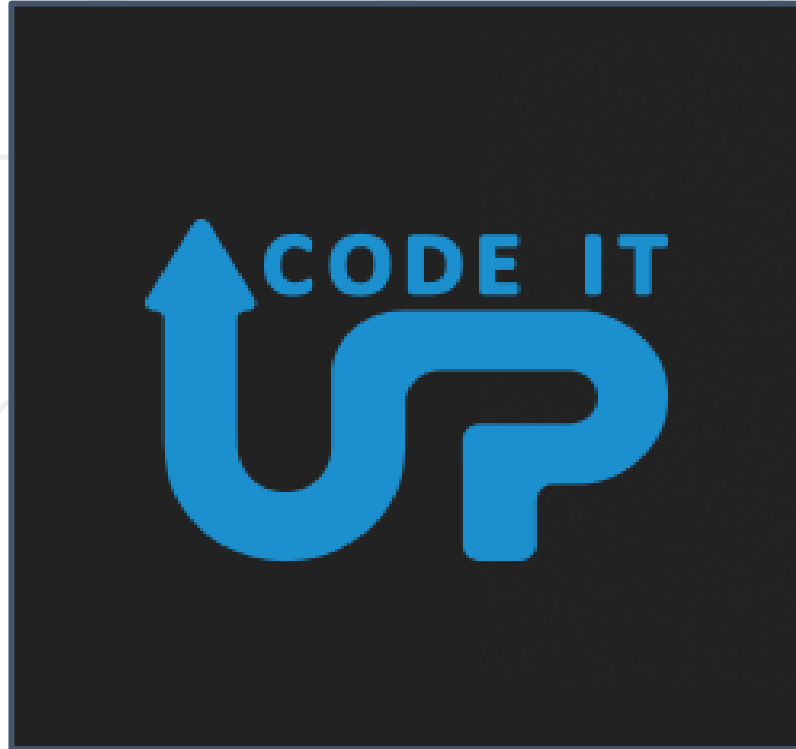


BOSCH

DXC
TECHNOLOGY



SmartIT



- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg/>
- © Software University – <https://softuni.bg>



- Software University – High-Quality Education, Profession and Job for Software Developers

- softuni.bg, about.softuni.bg

- Software University Foundation

- softuni.foundation

- Software University @ Facebook

- facebook.com/SoftwareUniversity

- Software University Forums

- forum.softuni.bg



Software University

