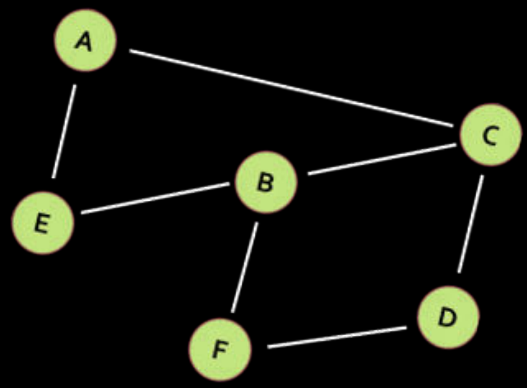
Graphs and Graph Algorithms

Fundamentals, Terminology, Traversal, Algorithms

SoftUni Foundation



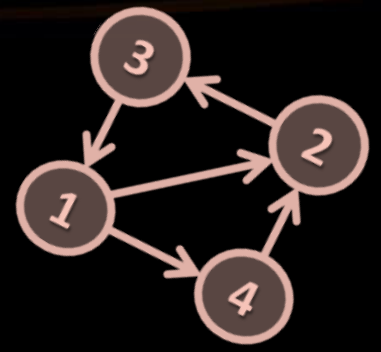
SoftUni Team

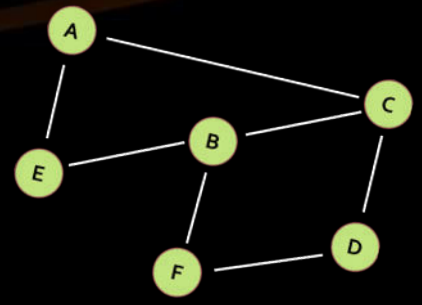
Technical Trainers

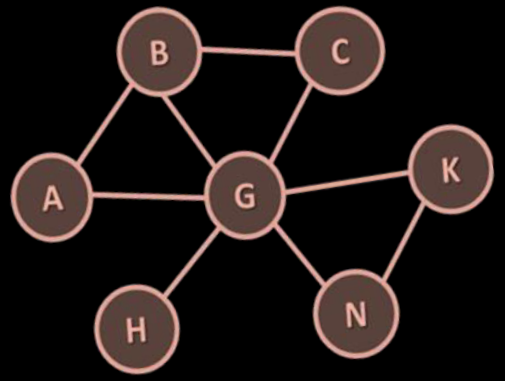
Software University

http://softuni.bg

Table of Contents







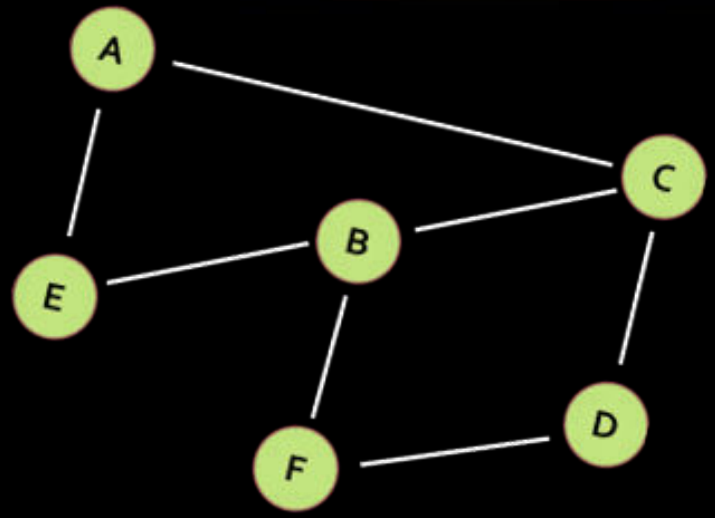
Have a Question?

SoftUni Foundation

sli.do

#DsAlgo

SoftUni Foundation

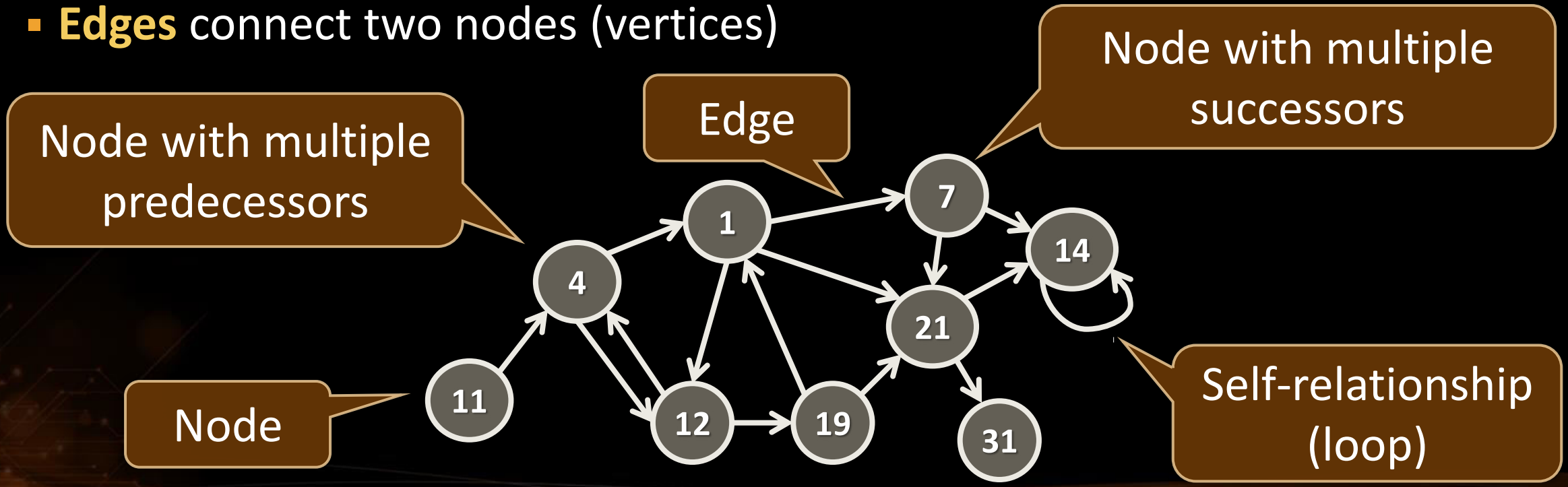


Graphs

Definitions and Terminology

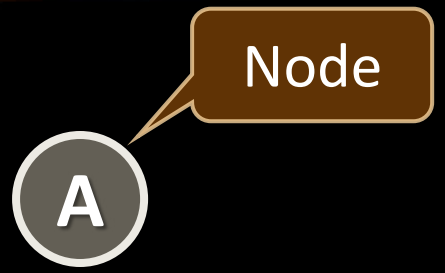
Graph Data Structure

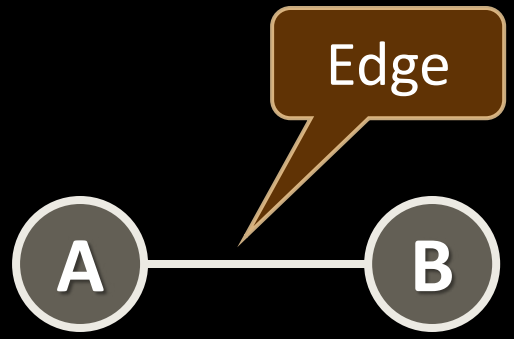
- **Graph**, denoted as **G(V, E)**
  - Set of nodes **V** with many-to-many relationship between them (edges **E**)
  - Each **node** (**vertex**) has **multiple** predecessors and multiple successors
  - **Edges** connect two nodes (vertices)



Graph Definitions

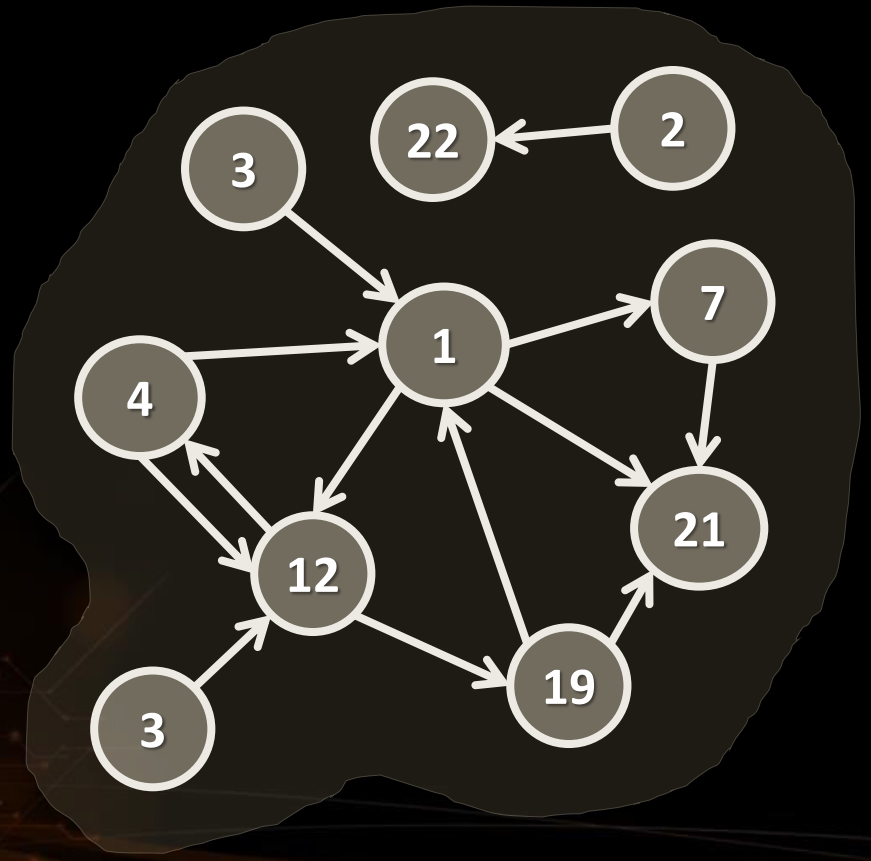
- **Node** (vertex)
  - Element of a graph
  - Can have name / value
  - Keeps a list of adjacent nodes
- **Edge**
  - Connection between two nodes
  - Can be directed / undirected
  - Can be weighted / unweighted
  - Can have name / value



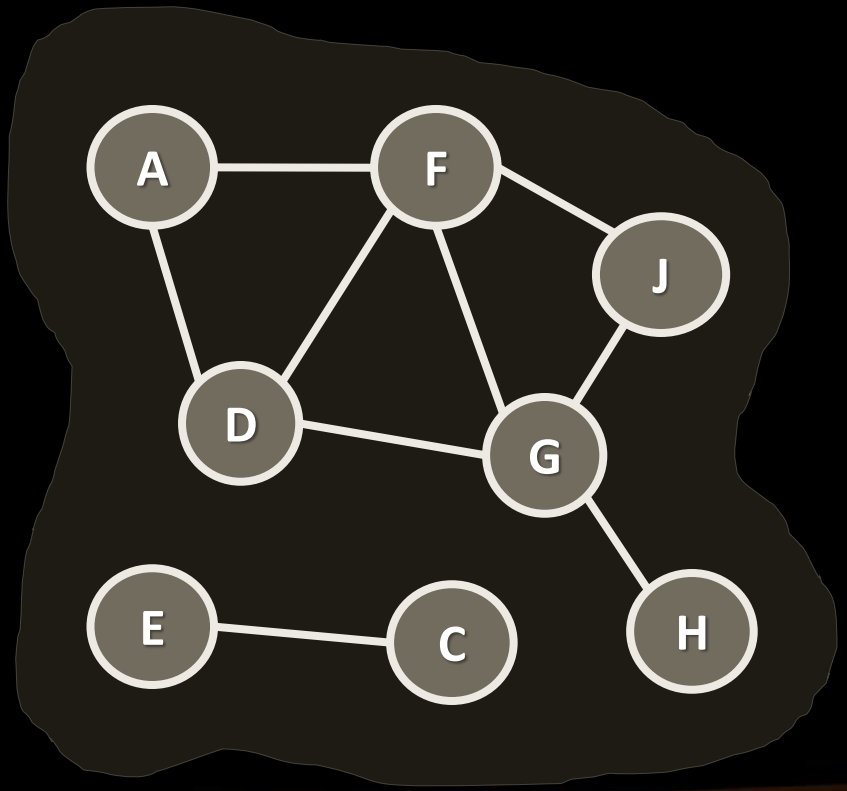


Graph Definitions (2)

- **Directed graph**
  - Edges have direction

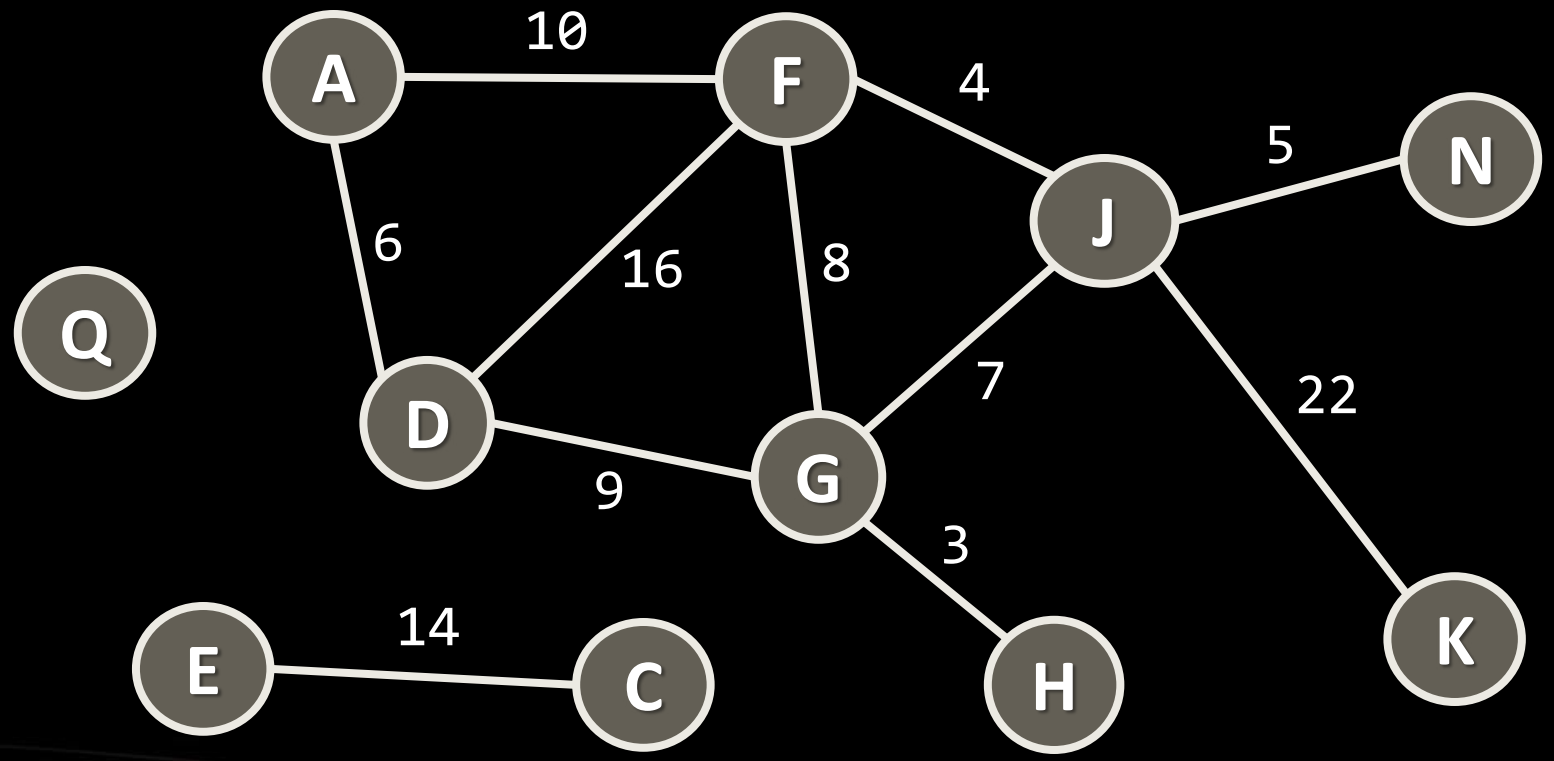


- **Undirected graph**
  - Undirected edges



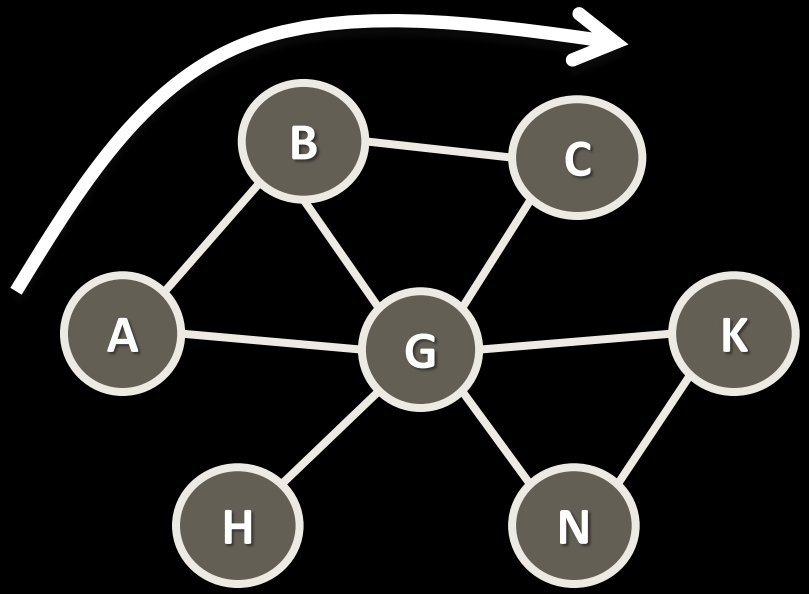
Graph Definitions (3)

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



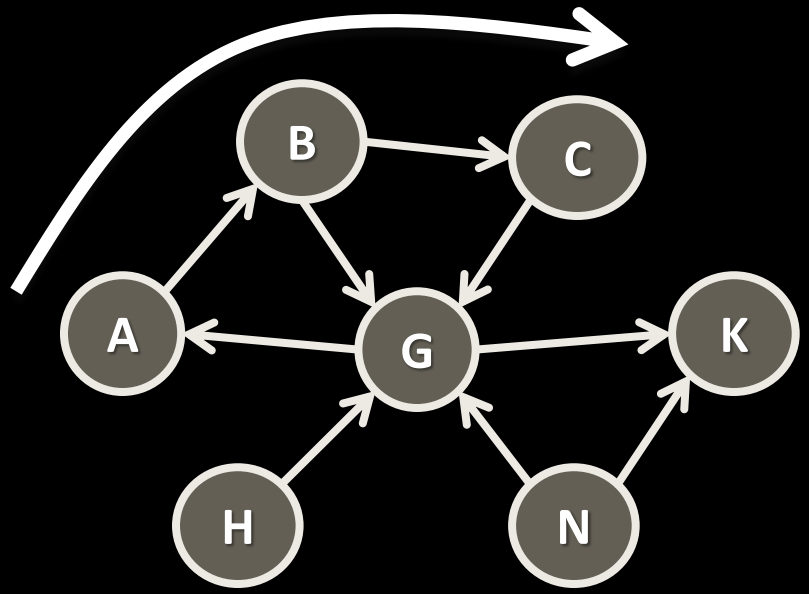
Graph Definitions (4)

- **Path** (in undirected graph)
  - Sequence of nodes n_1, n_2, … 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, G, B, N is not a path



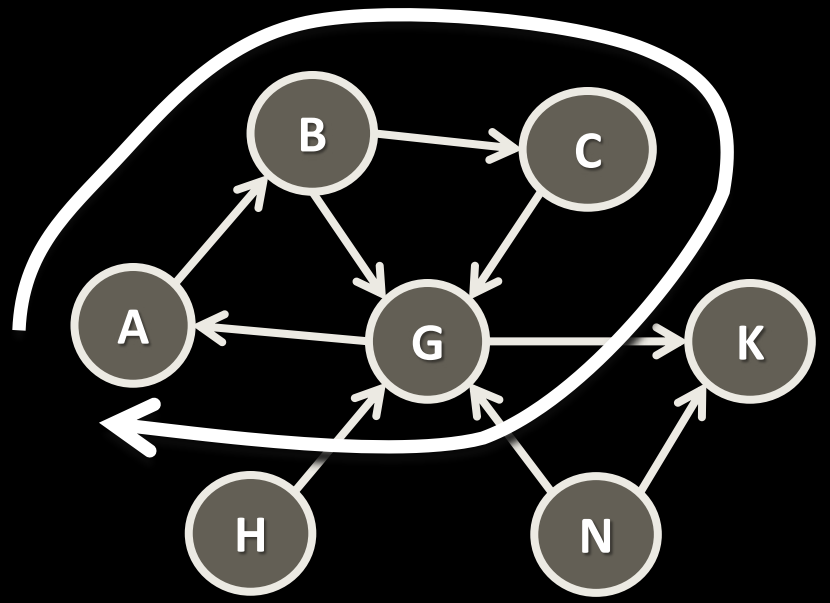
Graph Definitions (5)

- **Path** (in directed graph)
  - Sequence of nodes n_1, n_2, … 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



Graph Definitions (6)

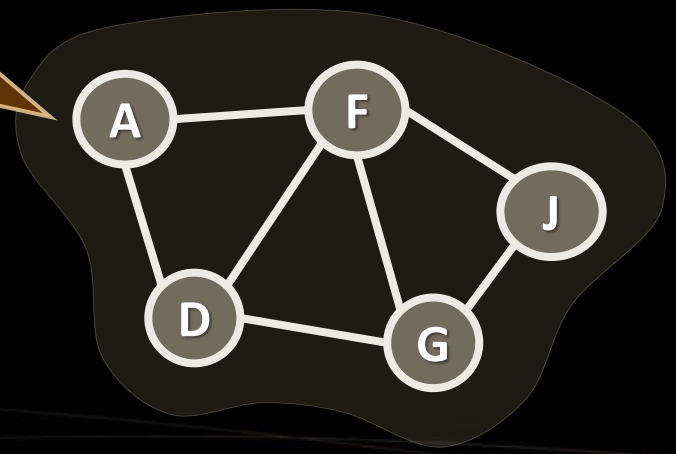
- **Cycle**
  - Path that ends back at the starting node
  - Example of cycle: A, B, C, G, A
- **Simple path**
  - No cycles in path
- **Acyclic graph**
  - Graph with no cycles
  - Acyclic undirected graphs are trees



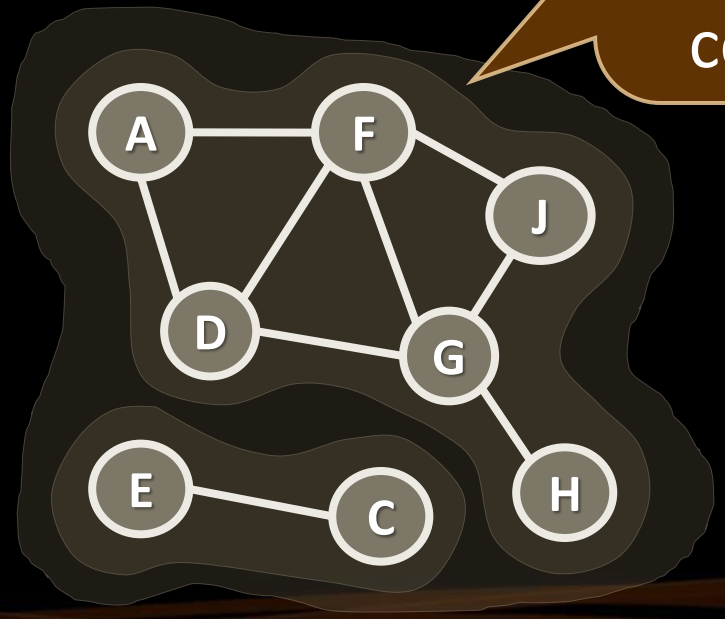
Graph Definitions (7)

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

Connected graph



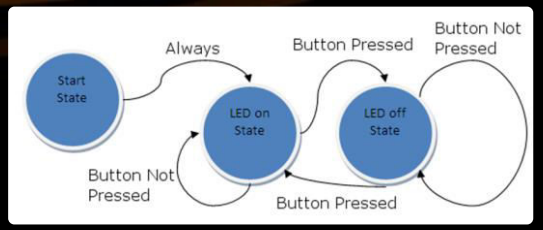
Unconnected graph holding two connected components

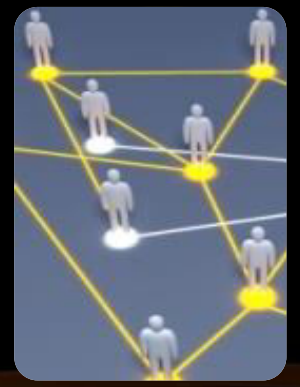


Graphs and Their Applications

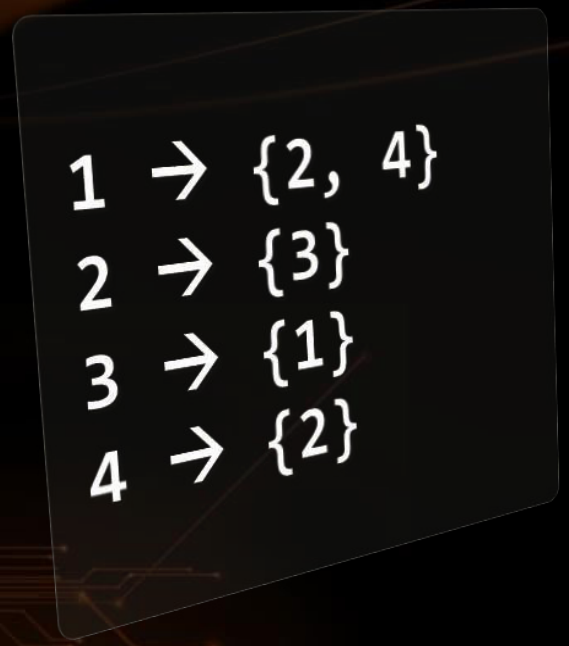
SoftUni Foundation

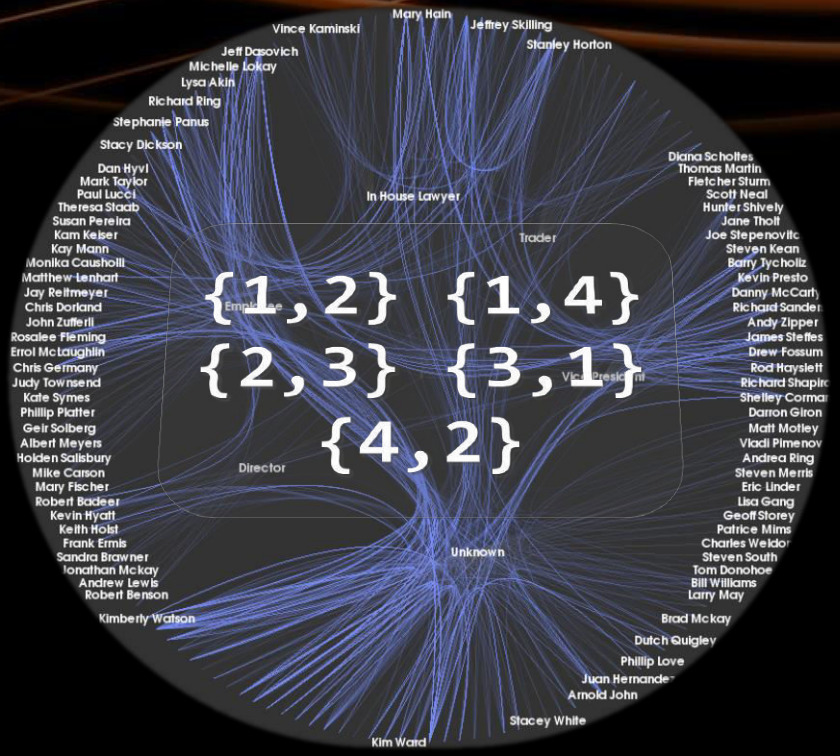
- Graphs have many real-world applications
  - Modeling a **computer network** like the Internet
    - Routes are simple paths in the network
  - Modeling a city **map**
    - Streets are edges, crossings are vertices
  - **Social networks**
    - People are nodes and their connections are edges
  -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

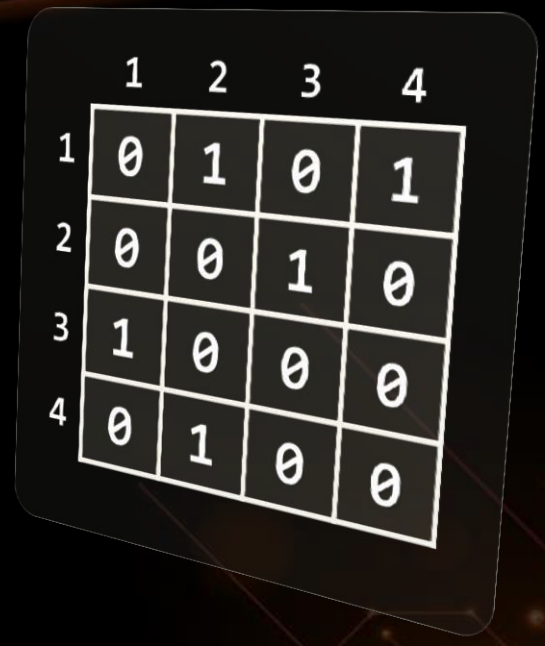




SoftUni Foundation







Representing Graphs

Classic and OOP Ways

Representing Graphs

- **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

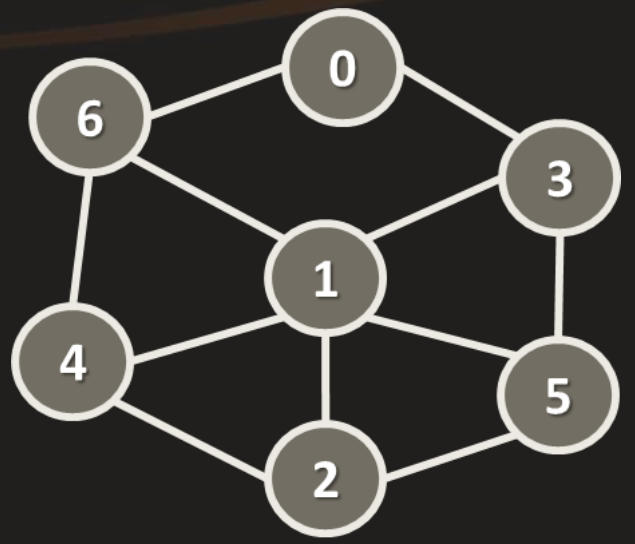
- **List of edges**

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

Graph Representation: Adjacency List

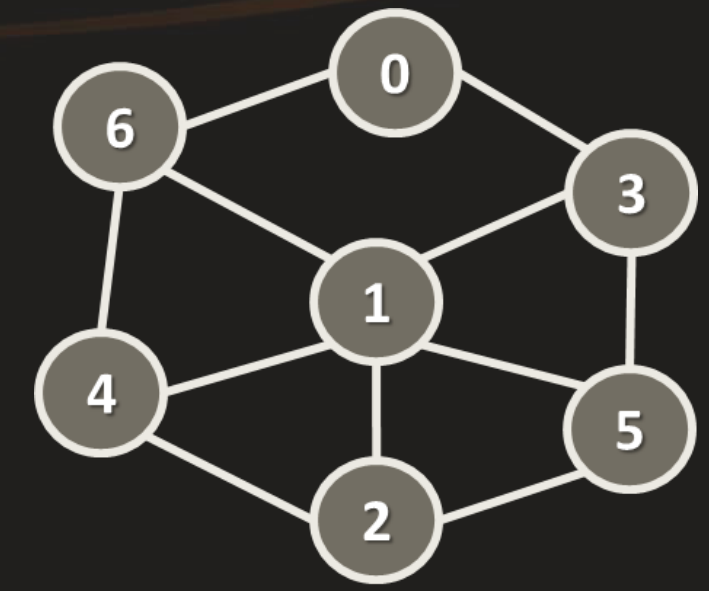
SoftUni Foundation

```
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);

// List the children of node #1
var childNodes = g[1];
```



Graph Representation: Adjacency Matrix

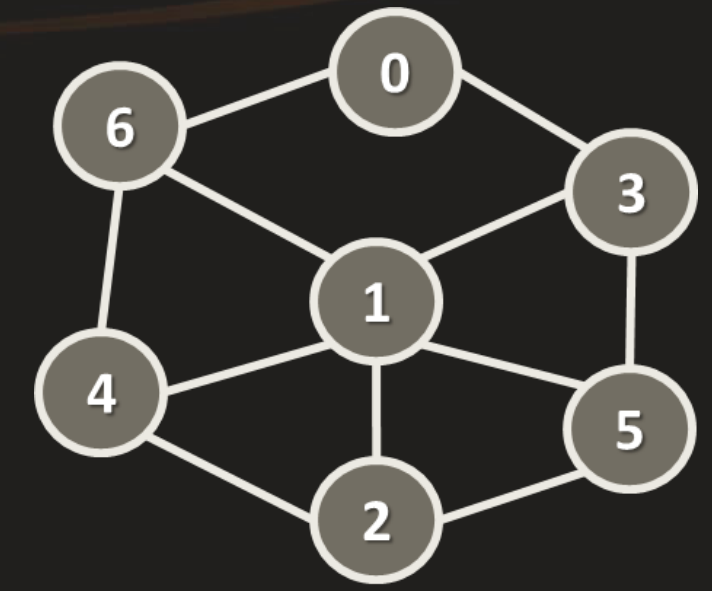

```
var graph = new[,]
{ // 0  1  2  3  4  5  6
    { 0, 0, 0, 1, 0, 0, 1 }, // node 0
    { 0, 0, 1, 1, 1, 1, 1 }, // node 1
    { 0, 1, 0, 0, 1, 1, 0 }, // node 2
    { 1, 1, 0, 0, 0, 1, 0 }, // node 3
    { 0, 1, 1, 0, 0, 0, 1 }, // node 4
    { 0, 1, 1, 1, 0, 0, 0 }, // node 5
    { 1, 1, 0, 0, 1, 0, 0 }, // node 6
};

// Add an edge { 3 → 6 }
g[3, 6] = 1;

// List the children of node #1
var childNodes = g[1];
```



Graph Representation: List of Edges

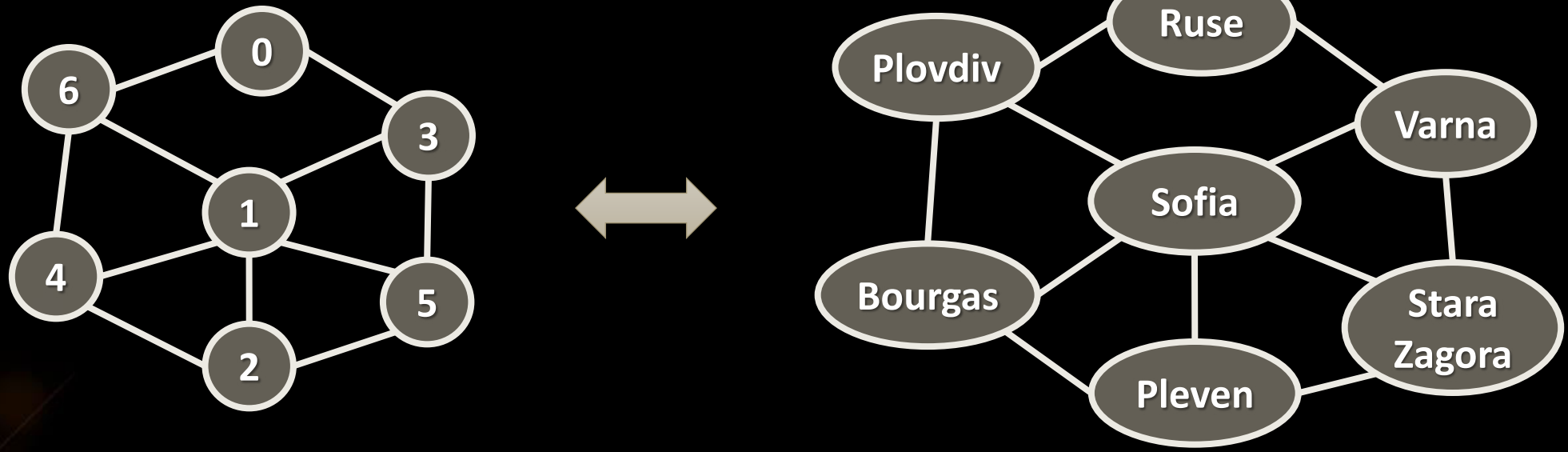
SoftUni Foundation

```
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)

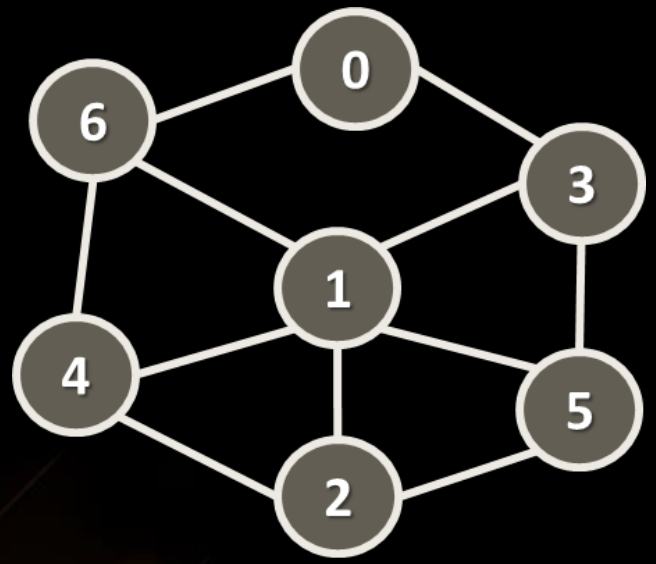


Graph of **numbered nodes**: [0…6]

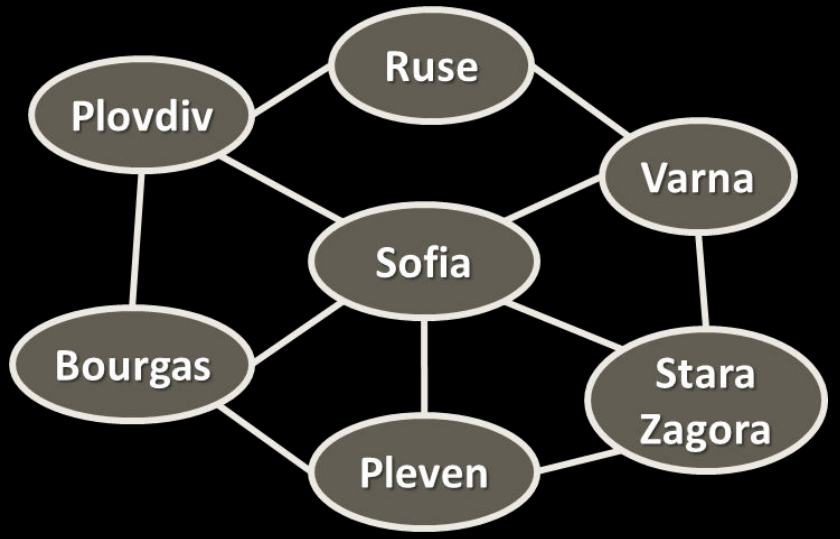
Graph of **named nodes**

Numbering Graph Nodes – How?

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



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



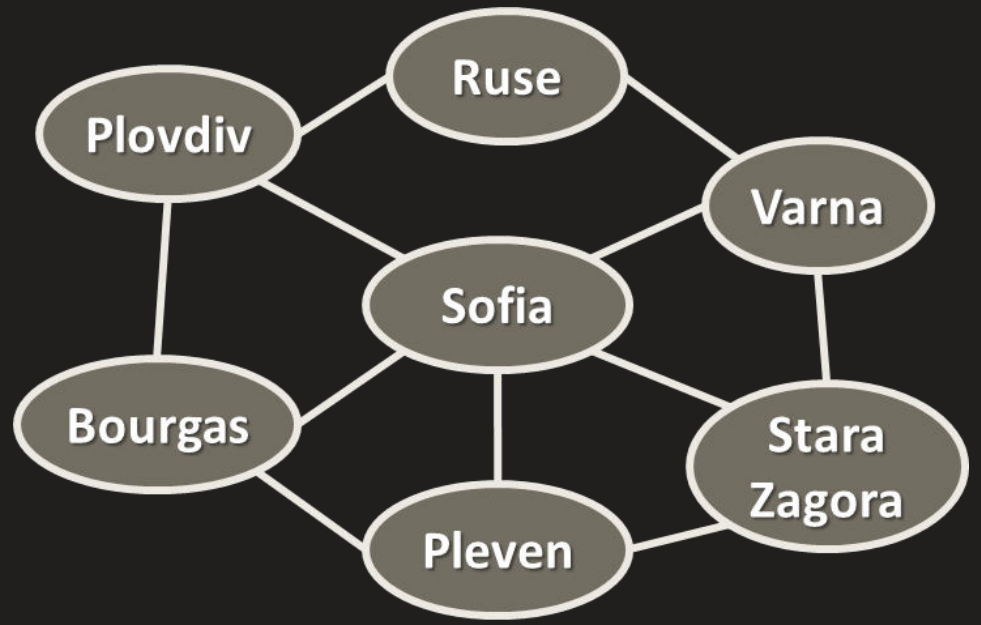

```
var g =
    new List<int>[n]
```

```
var g = new Dictionary<
  string, List<string>>
```

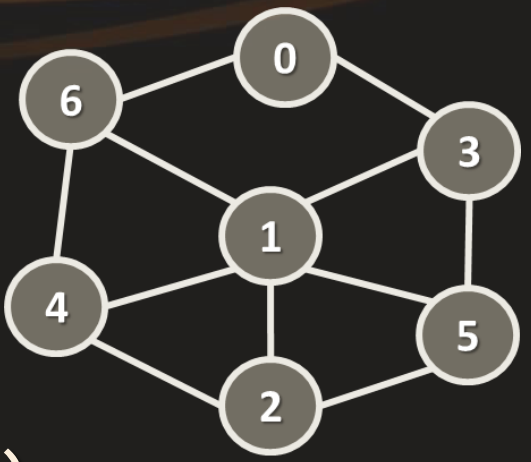
Graph of Named Nodes – Example

SoftUni Foundation

```
var graph = new Dictionary<string, List<string>>() {
  { "Sofia", new List<string>() {
    "Plovdiv", "Varna", "Bourgas", "Pleven", "Stara Zagora" } },
  { "Plovdiv", new List<string>() {
    "Bourgas", "Ruse" } },
  { "Varna", new List<string>() {
    "Ruse", "Stara Zagora" } },
  { "Bourgas", new List<string>() {
    "Plovdiv", "Pleven" } },
  { "Ruse", new List<string>() {
    "Varna", "Plovdiv" } },
  { "Pleven", new List<string>() {
    "Bourgas", "Stara Zagora" } },
  { "Stara Zagora", new List<string>() {
    "Varna", "Pleven" } },
};
```



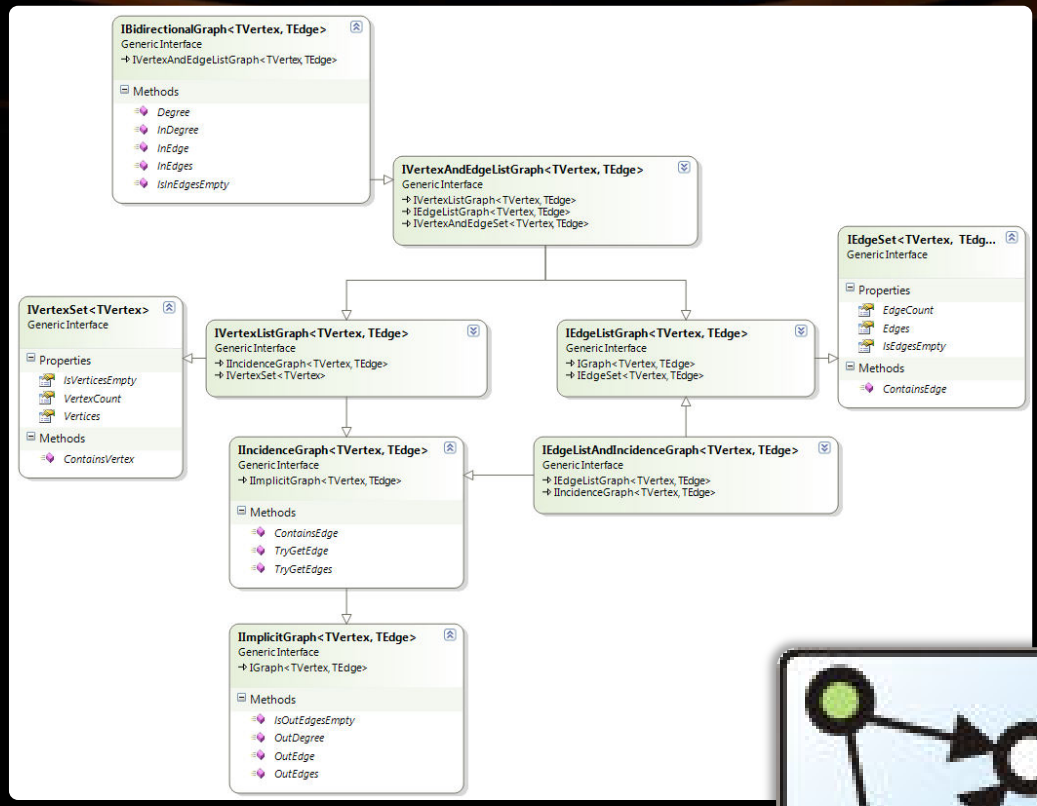
Graph of Numbered Nodes – Example



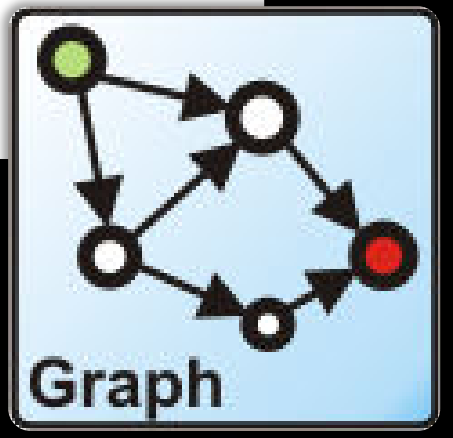

```
public class Graph
{
  List<int>[] childNodes;
  string[] nodeNames;
}

Graph g = new Graph(new List<int>[] {
  new List<int> {3, 6}, // children of node 0 (Ruse)
  new List<int> {2, 3, 4, 5, 6}, // successors of node 1 (Sofia)
  new List<int> {1, 4, 5}, // successors of node 2 (Pleven)
  new List<int> {0, 1, 5}, // successors of node 3 (Varna)
  new List<int> {1, 2, 6}, // successors of node 4 (Bourgas)
  new List<int> {1, 2, 3}, // successors of node 5 (Stara Zagora)
  new List<int> {0, 1, 4}  // successors of node 6 (Plovdiv)
},
new string[] {"Ruse", "Sofia", "Pleven", "Varna", "Bourgas", … });
```

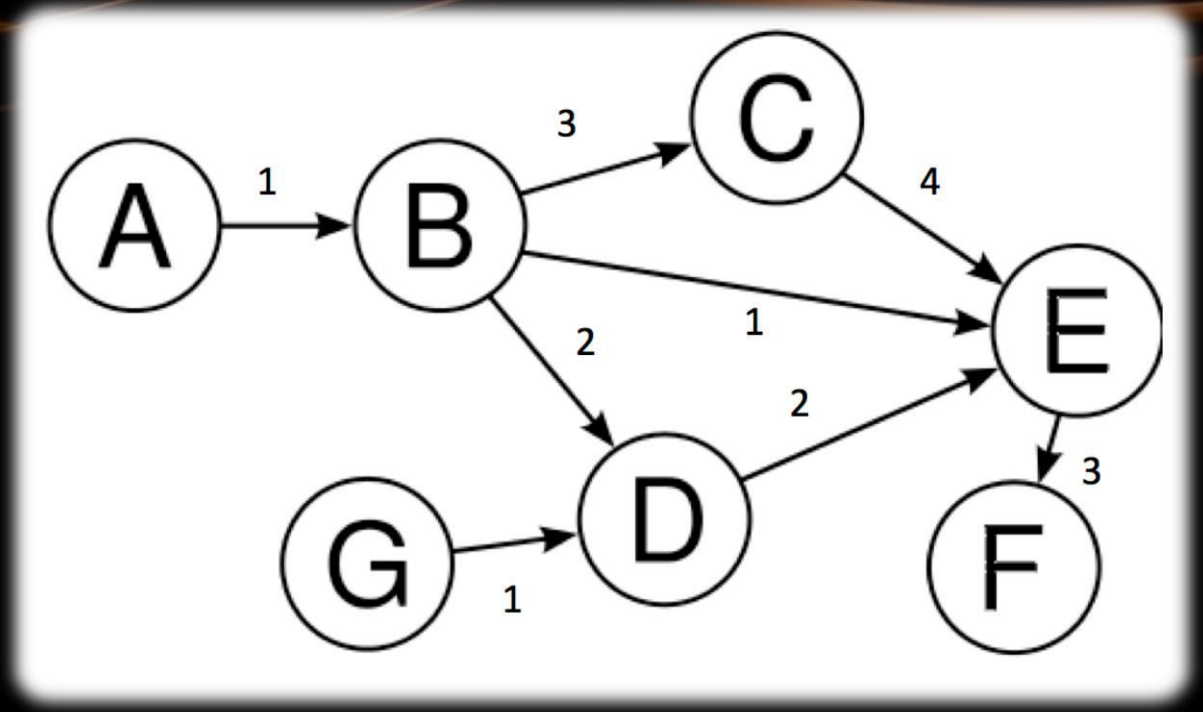
OOP-Based-Graph Representation

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



- Using external graph and algorithms library:
  - **QuickGraph** – http://quickgraph.codeplex.com

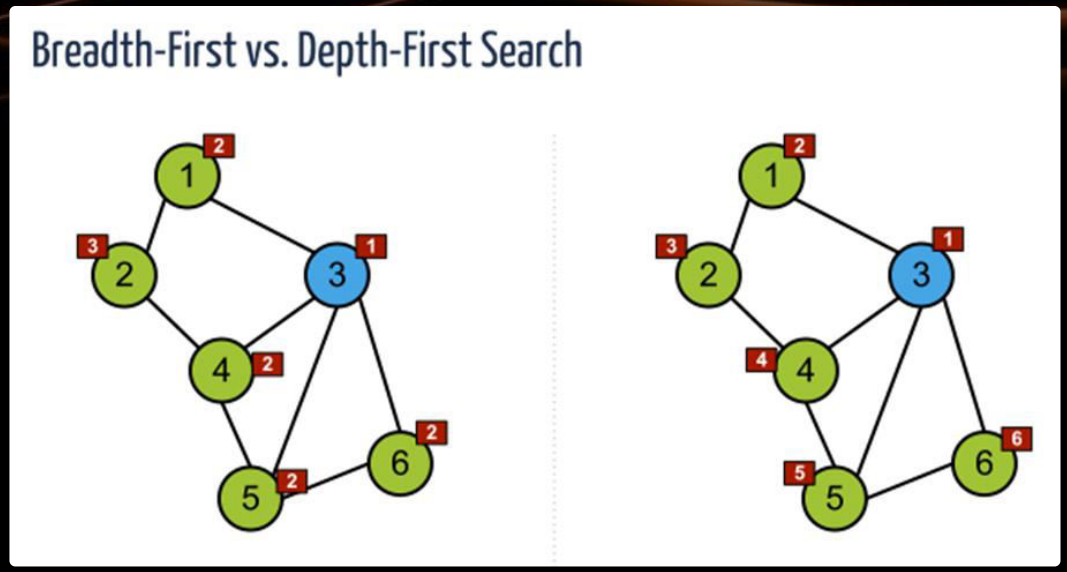


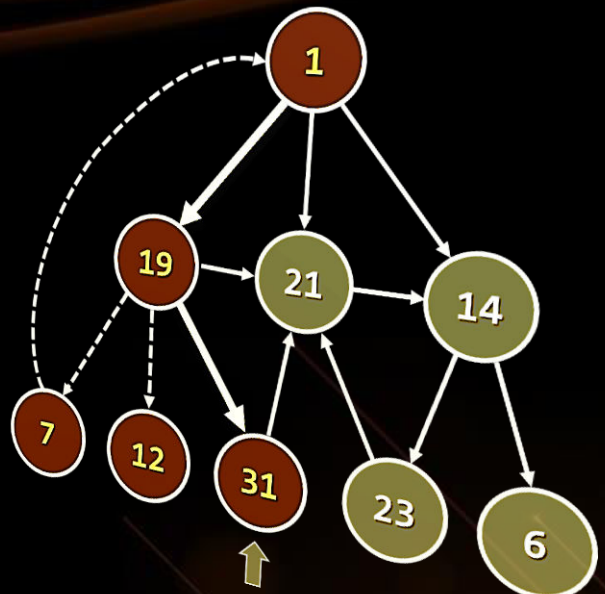


Representing Graphs

Live Demo

SoftUni Foundation



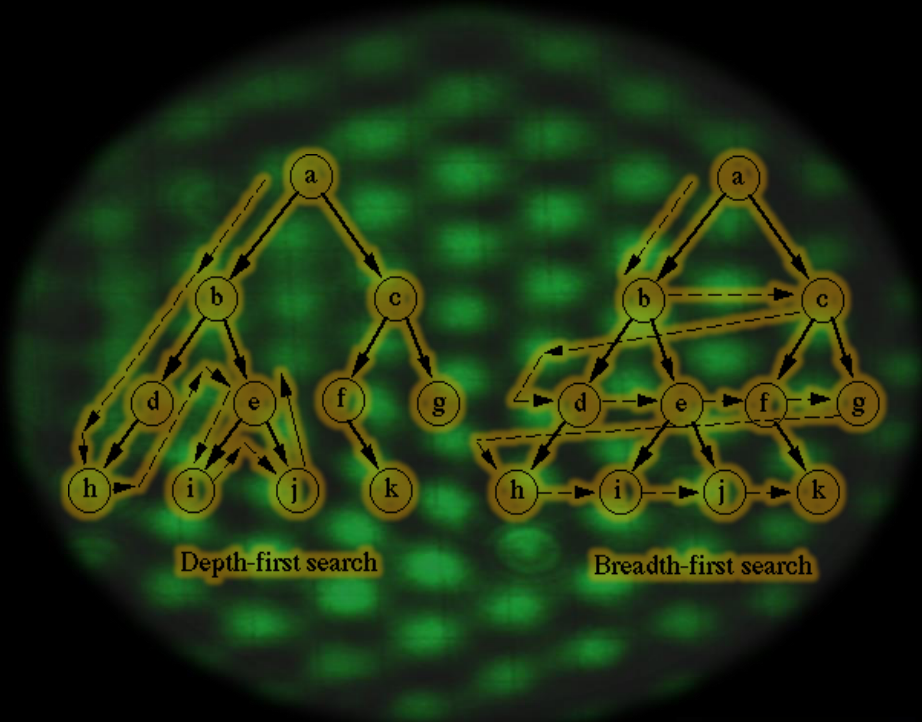


Graphs Traversals

Depth-First Search (DFS) and Breadth-First Search (BFS)

Graph Traversal Algorithms

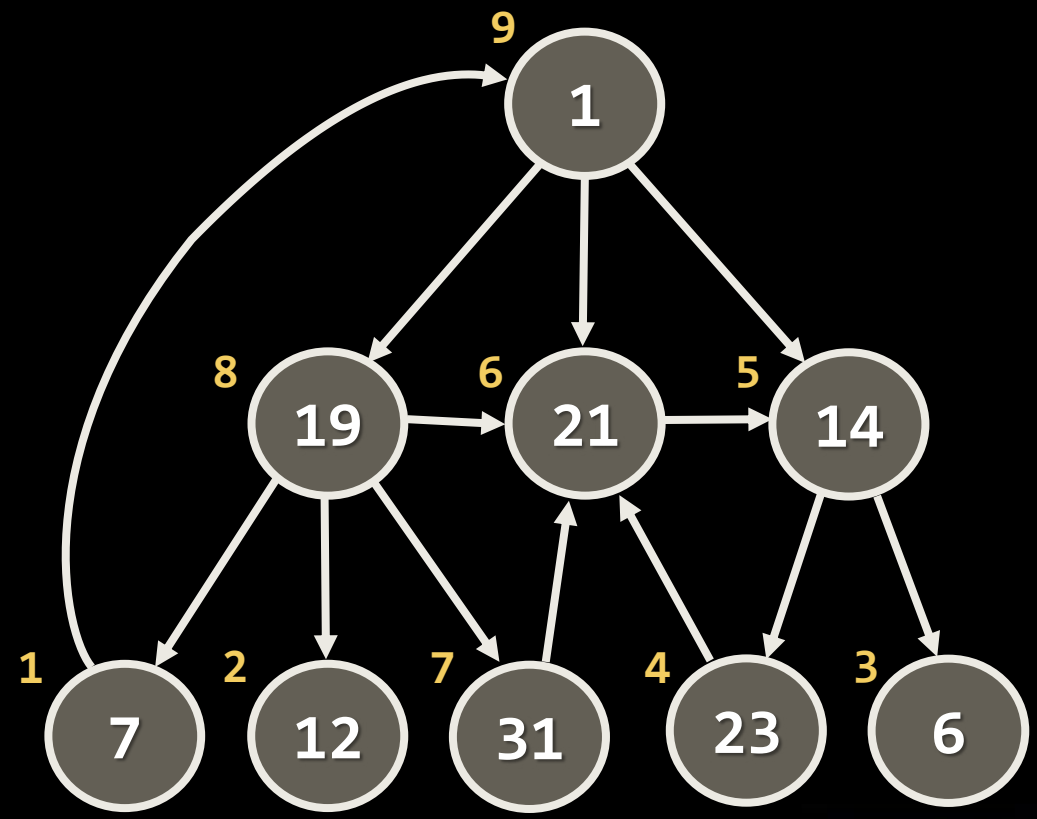
- Traversing a graph means to visit each of its nodes exactly once
  - The order of visiting nodes may vary on the traversal algorithm
  - **Depth-First Search** (DFS)
    - Visit node's successors first
    - Usually implemented by **recursion**
  - **Breadth-First Search** (BFS)
    - Nearest nodes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



Depth-First Search (DFS)

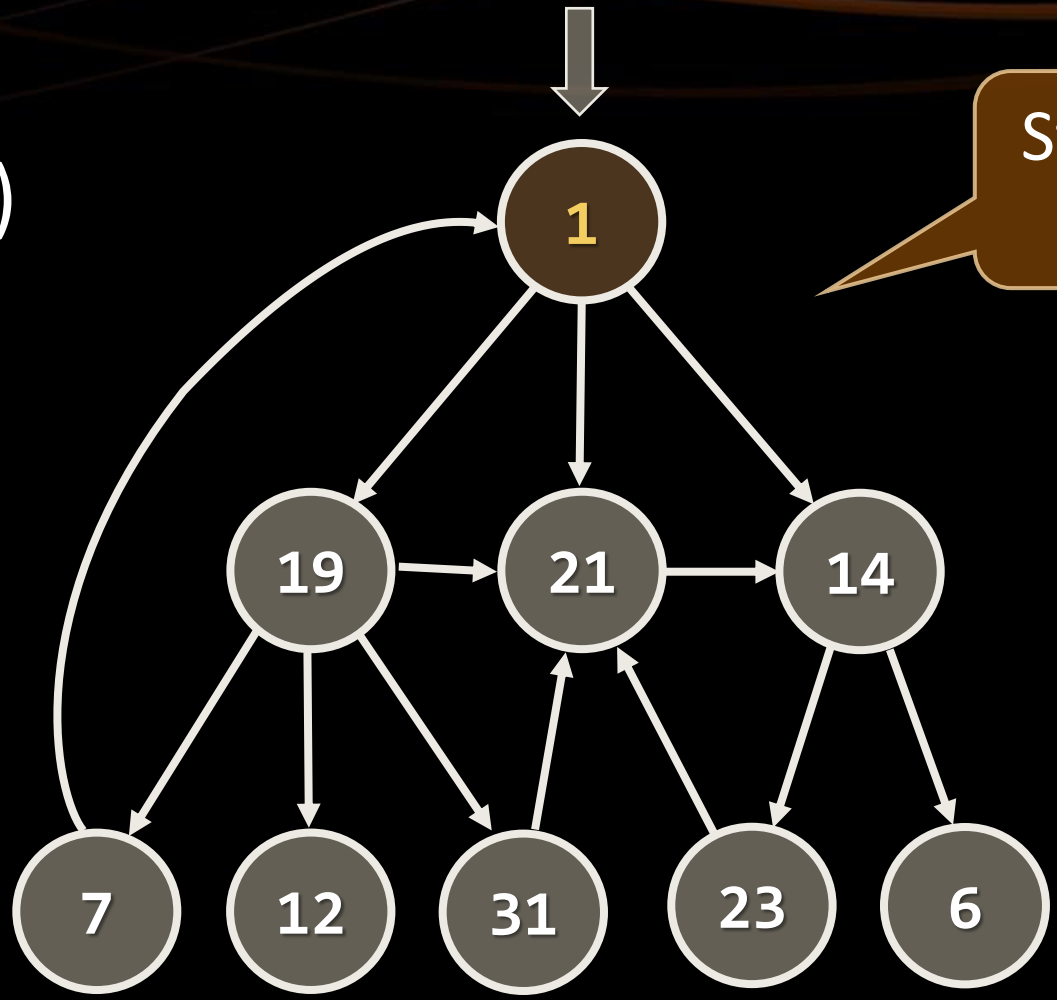
- **Depth-First Search** (**DFS**) first visits all descendants of given node recursively, finally visits the node itself

```
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)

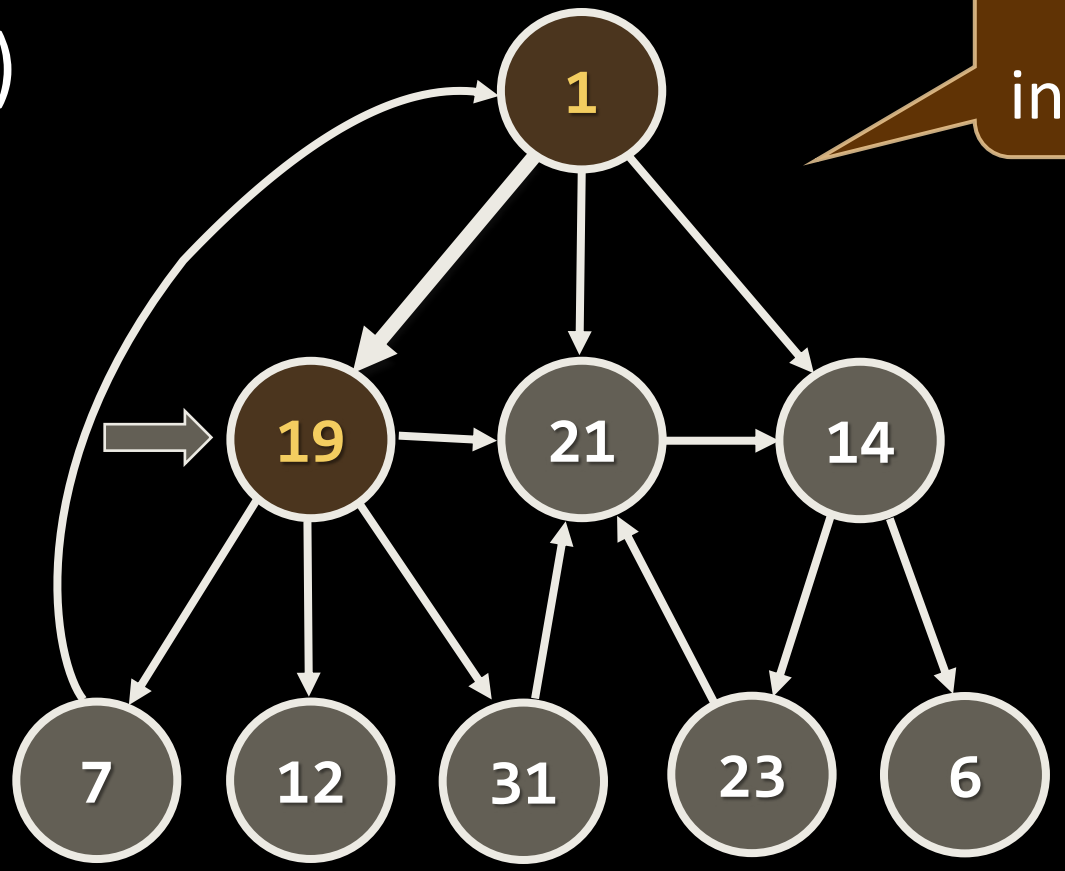
- Stack: 1
- Output: (empty)



Start DFS from the initial node 1

DFS in Action (Step 2)

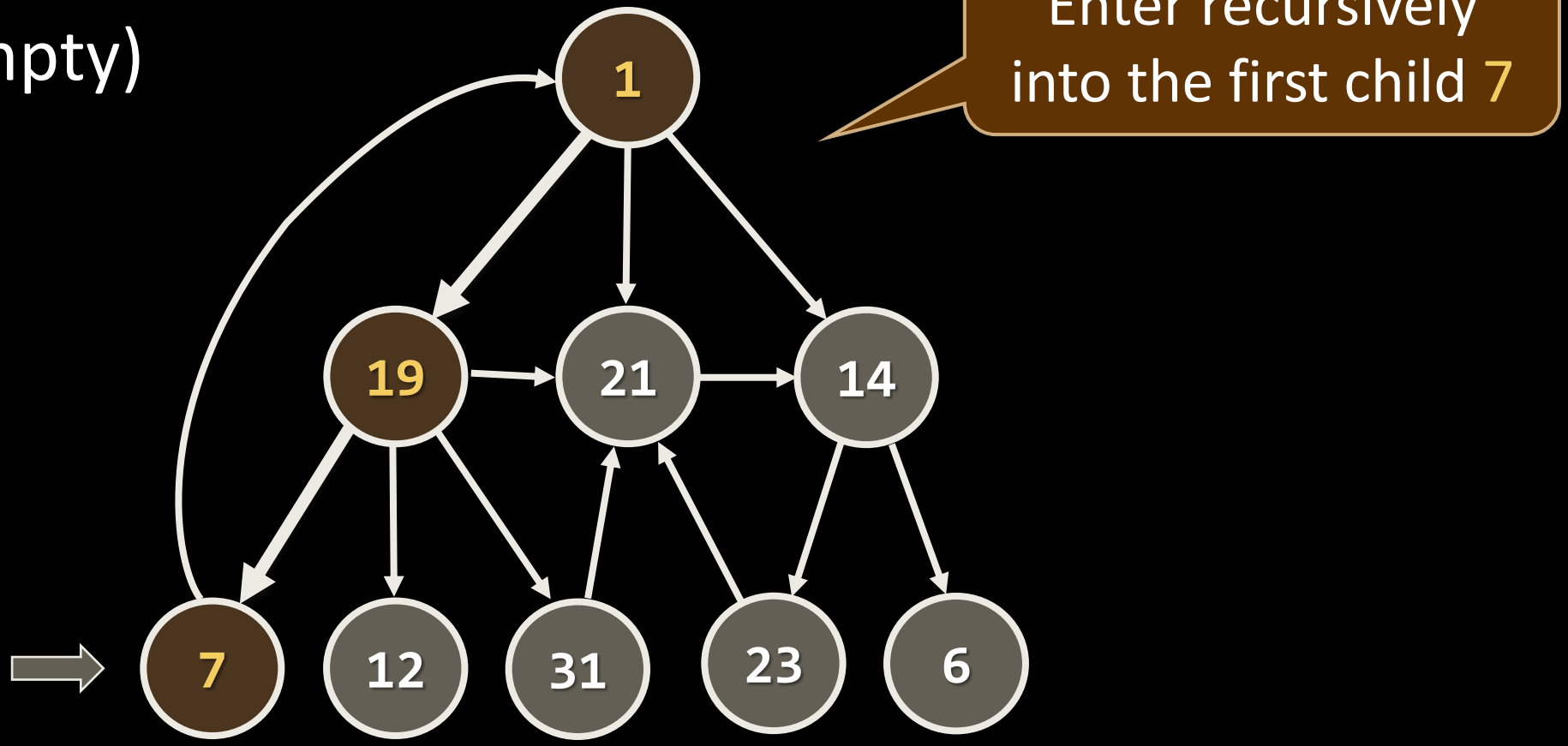
- Stack: 1, 19
- Output: (empty)



Enter recursively into the first child 19

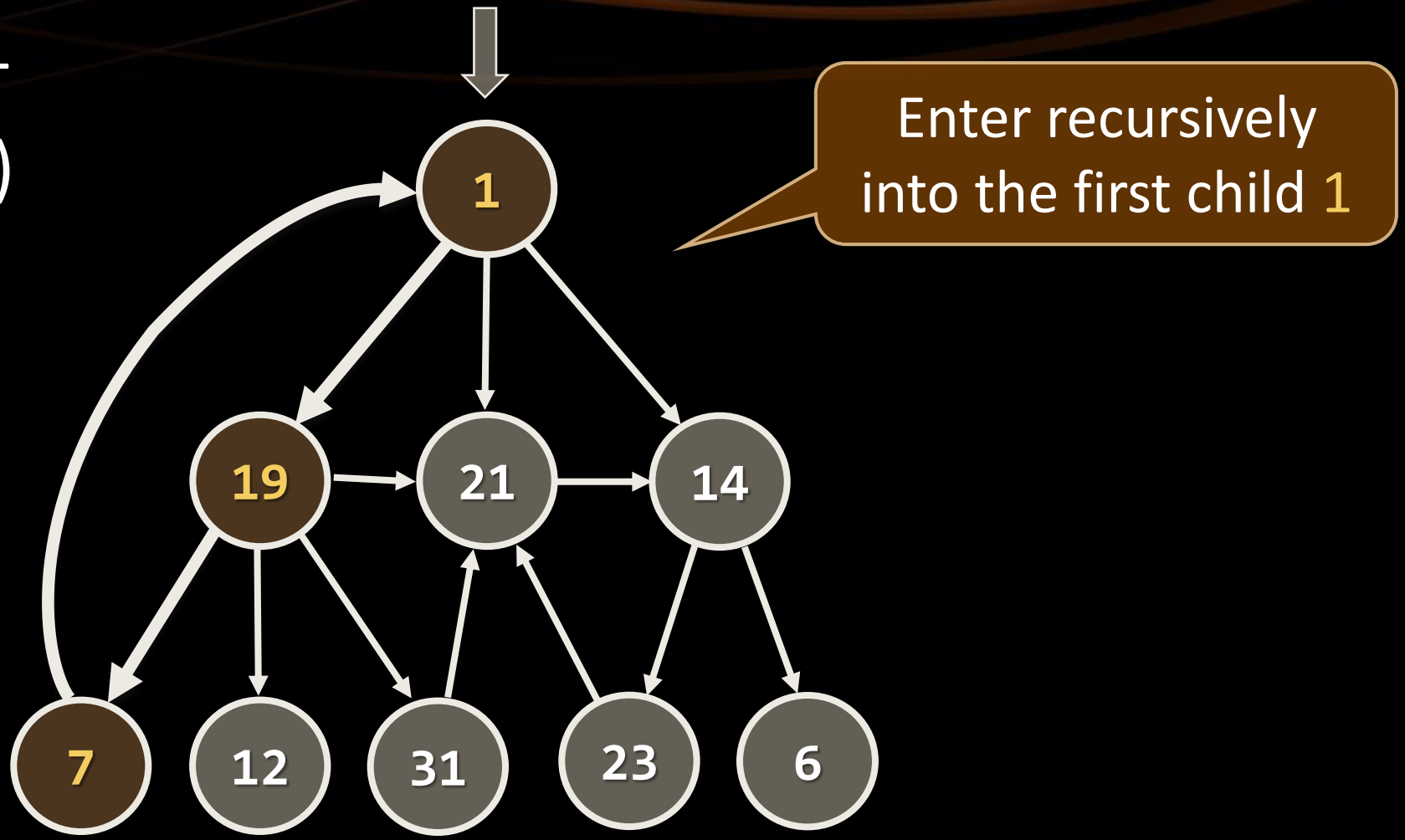
DFS in Action (Step 3)

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



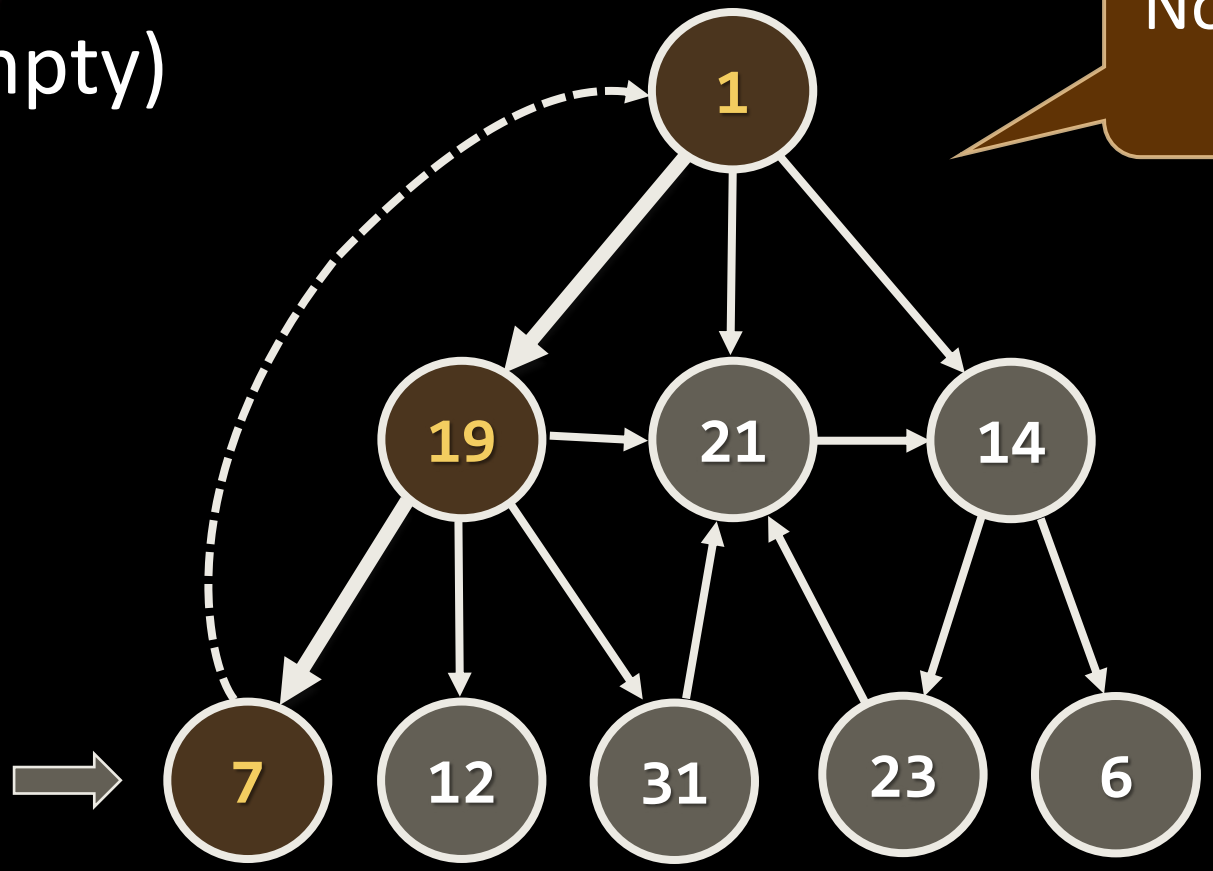
DFS in Action (Step 4)

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



DFS in Action (Step 5)

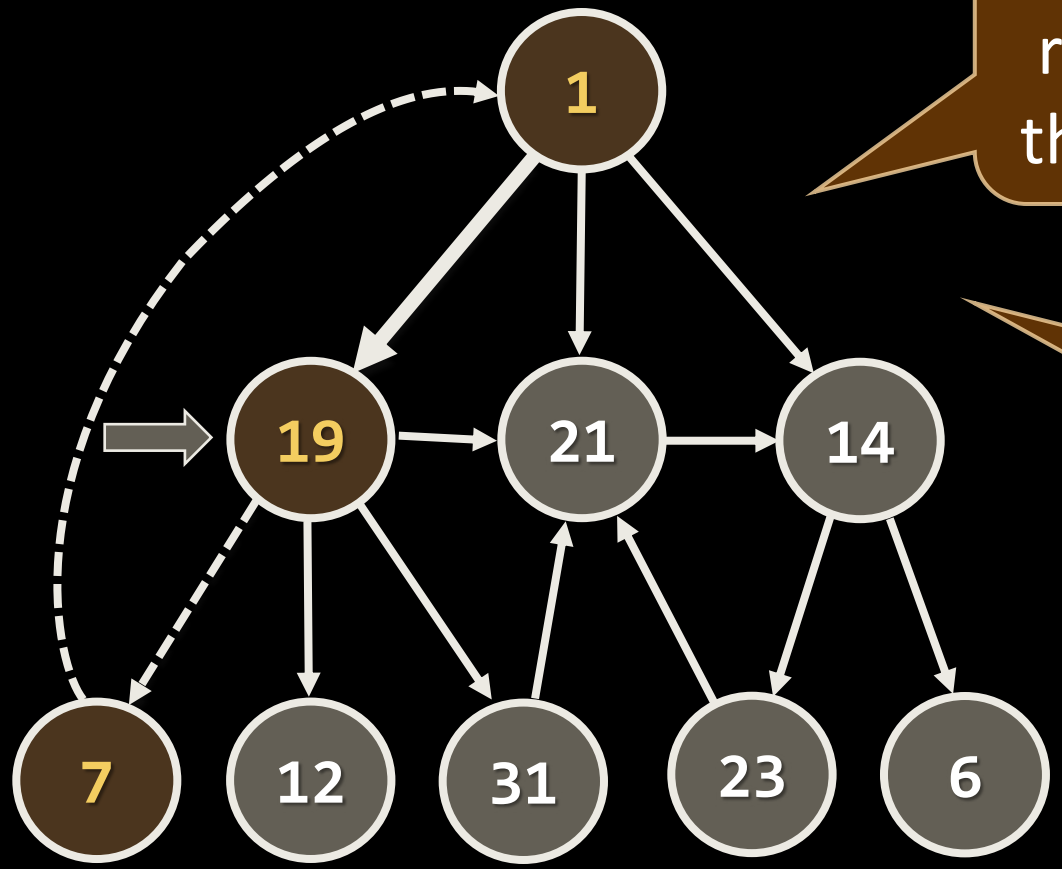
- Stack: 1, 19, 7
- Output: (empty)



Node 1 already visited
→ return back

DFS in Action (Step 6)

- Stack: 1, 19
- Output: 7

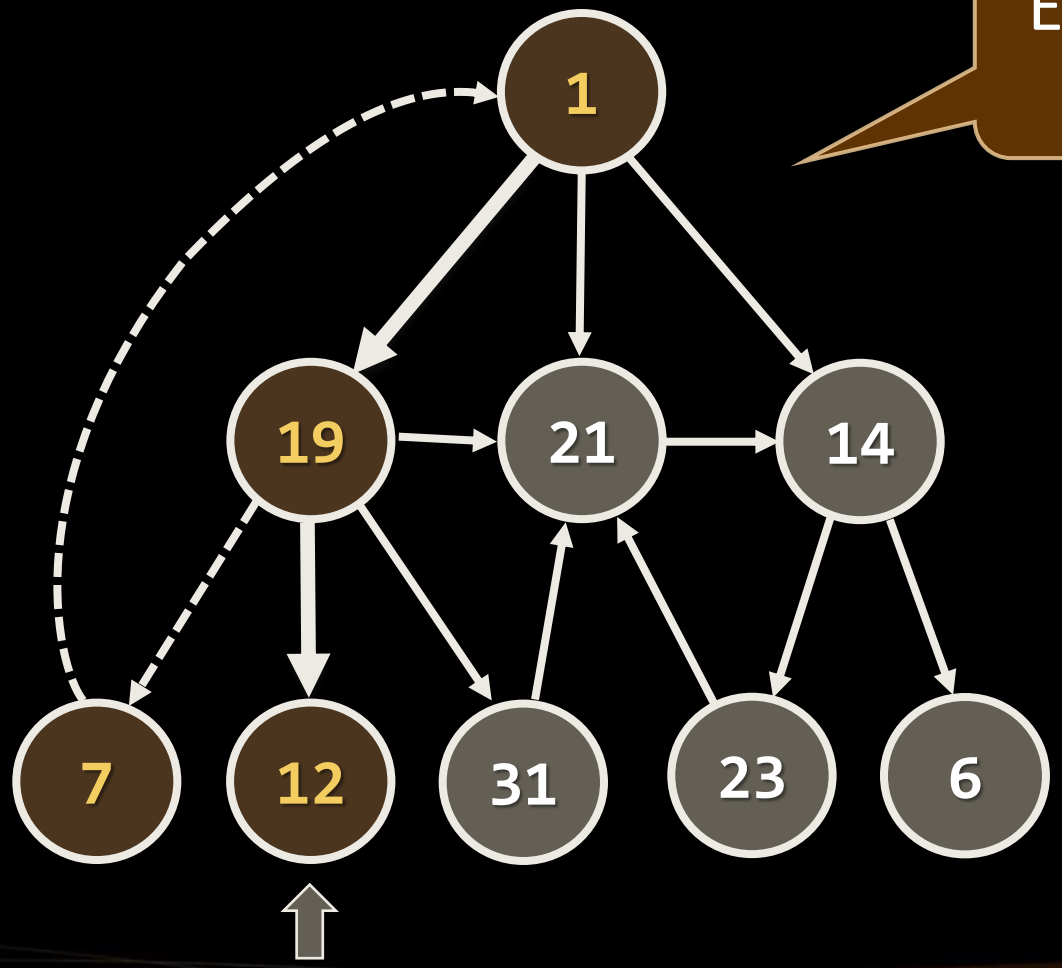


Return back from recursion and print the last visited node

Print node 7

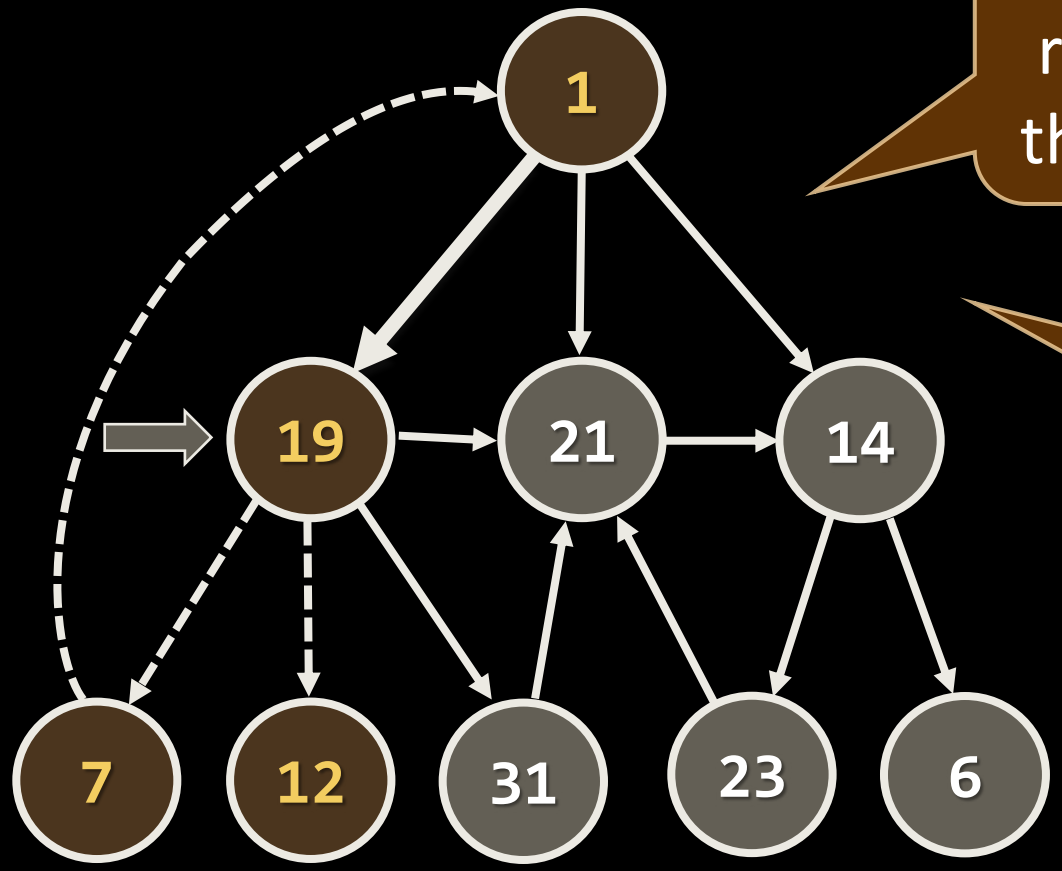
DFS in Action (Step 7)

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



DFS in Action (Step 8)

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

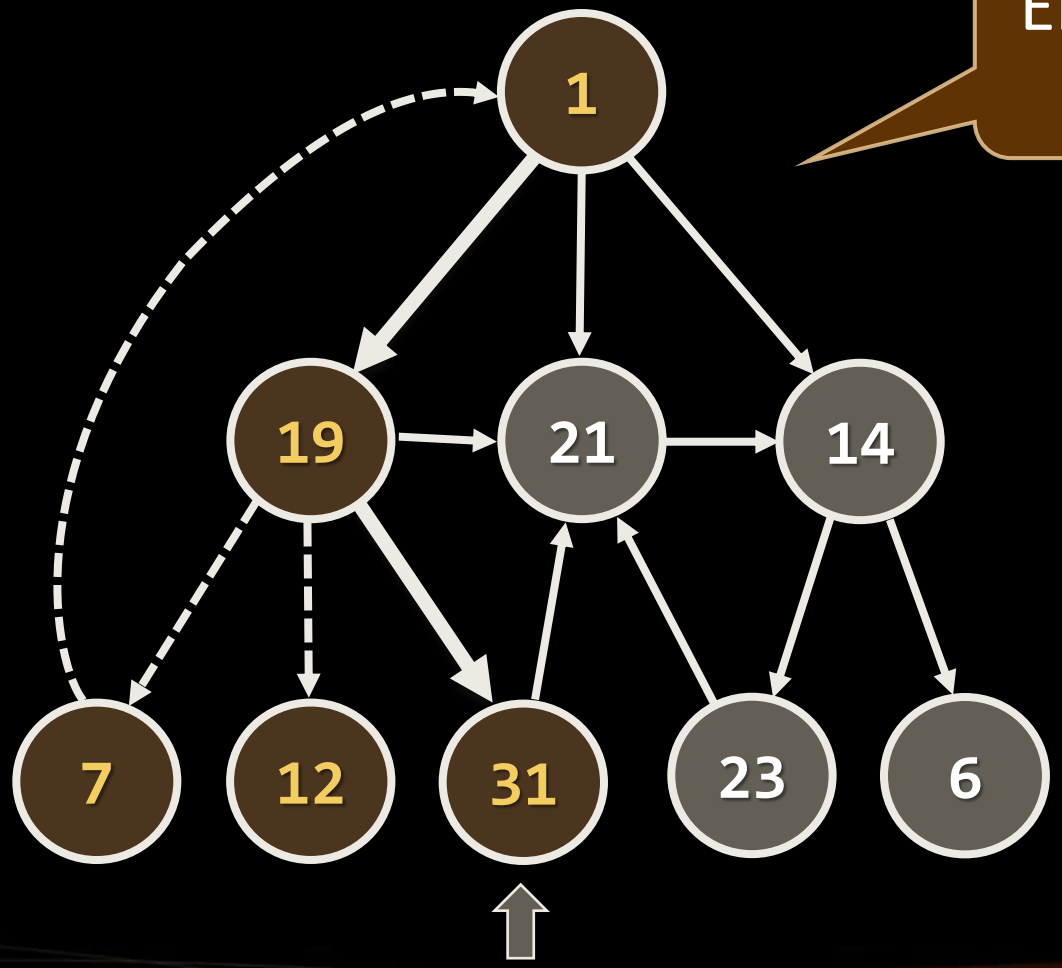


Return back from recursion and print the last visited node

Print node 12

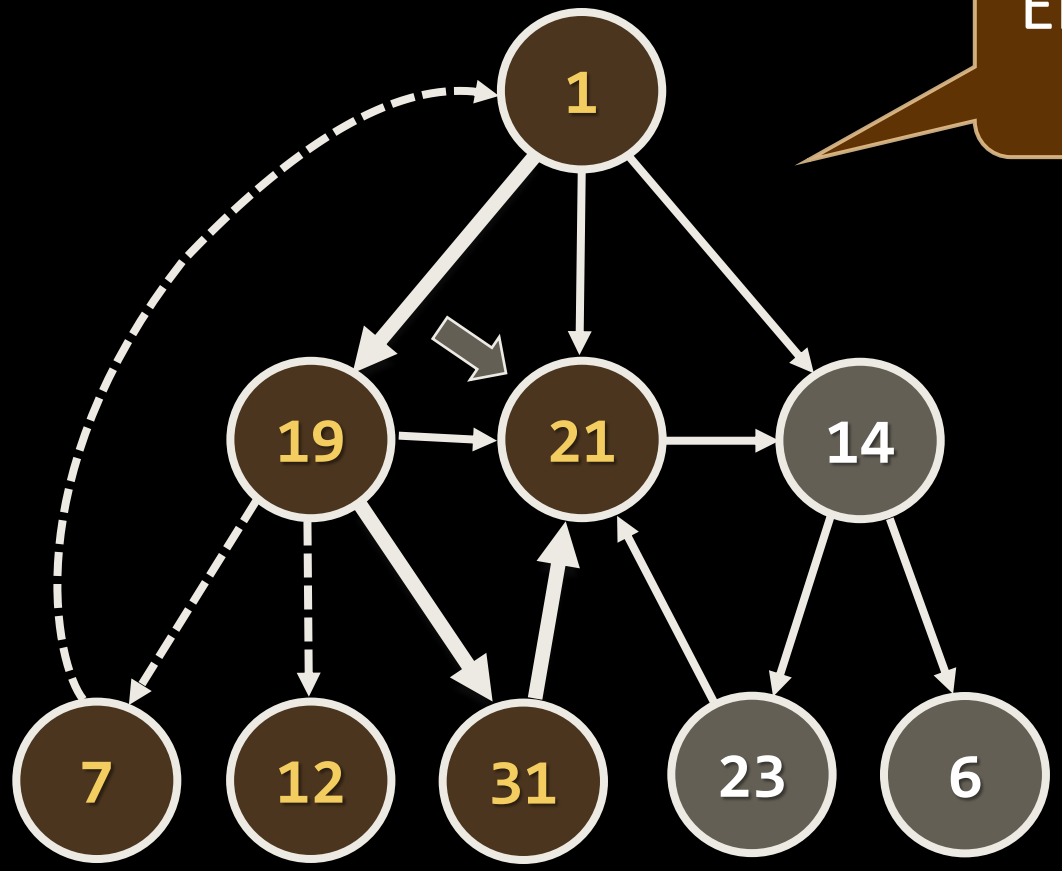
DFS in Action (Step 9)

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



DFS in Action (Step 10)

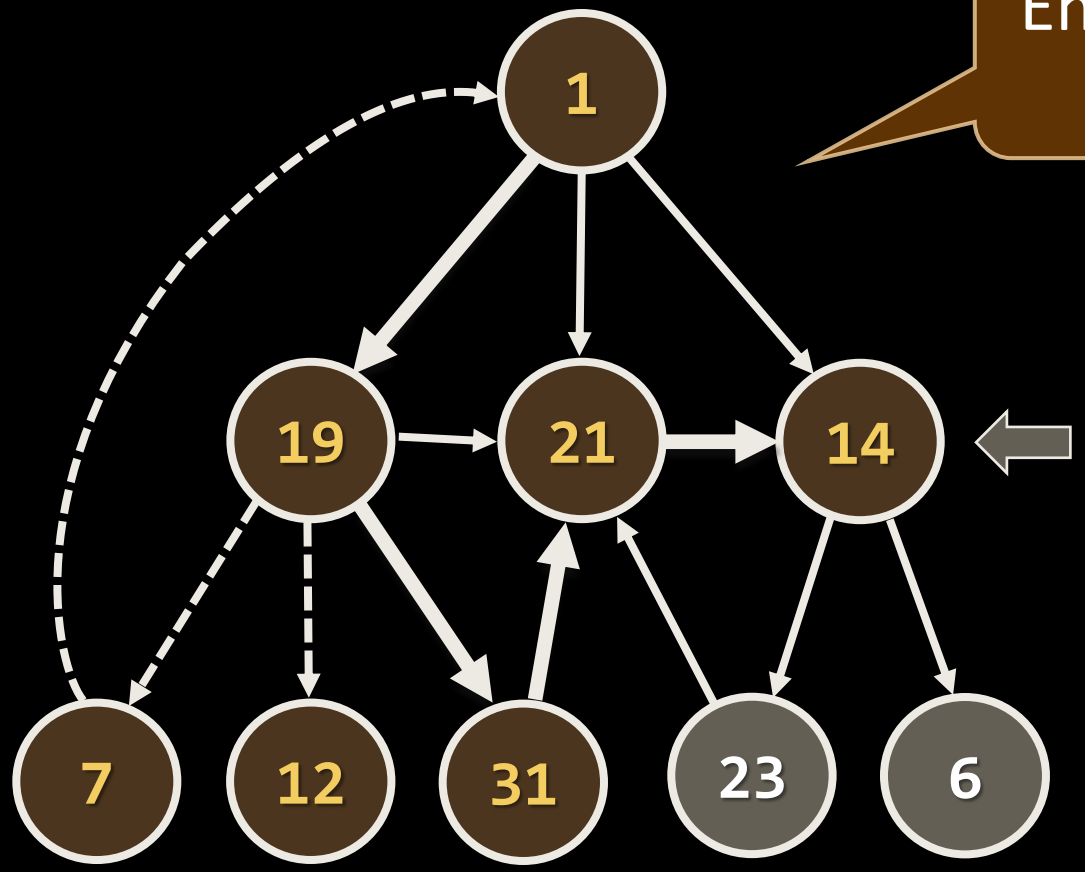
- Stack: 1, 19, 31, 21
- Output: 7, 12



Enter recursively into the first child 21

DFS in Action (Step 11)

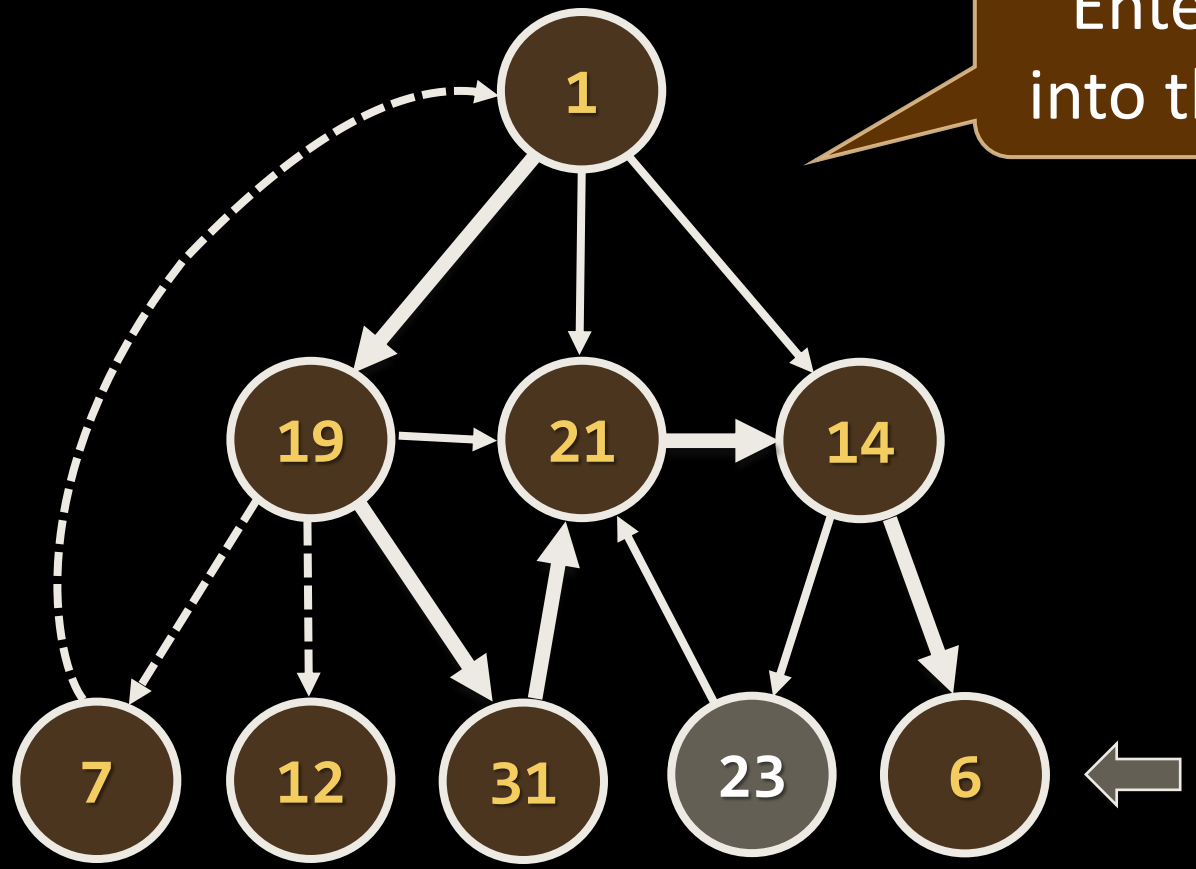
- Stack: 1, 19, 31, 21, 14
- Output: 7, 12



Enter recursively into the first child 14

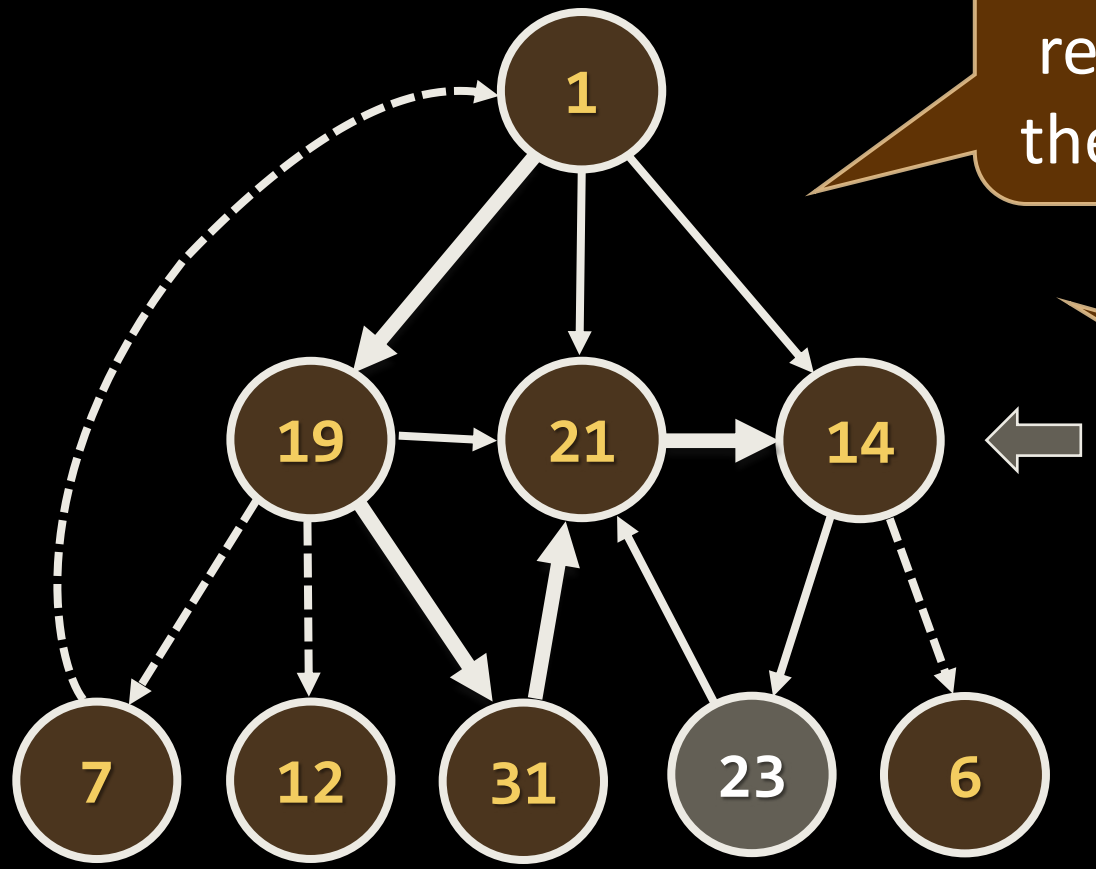
DFS in Action (Step 12)

- 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

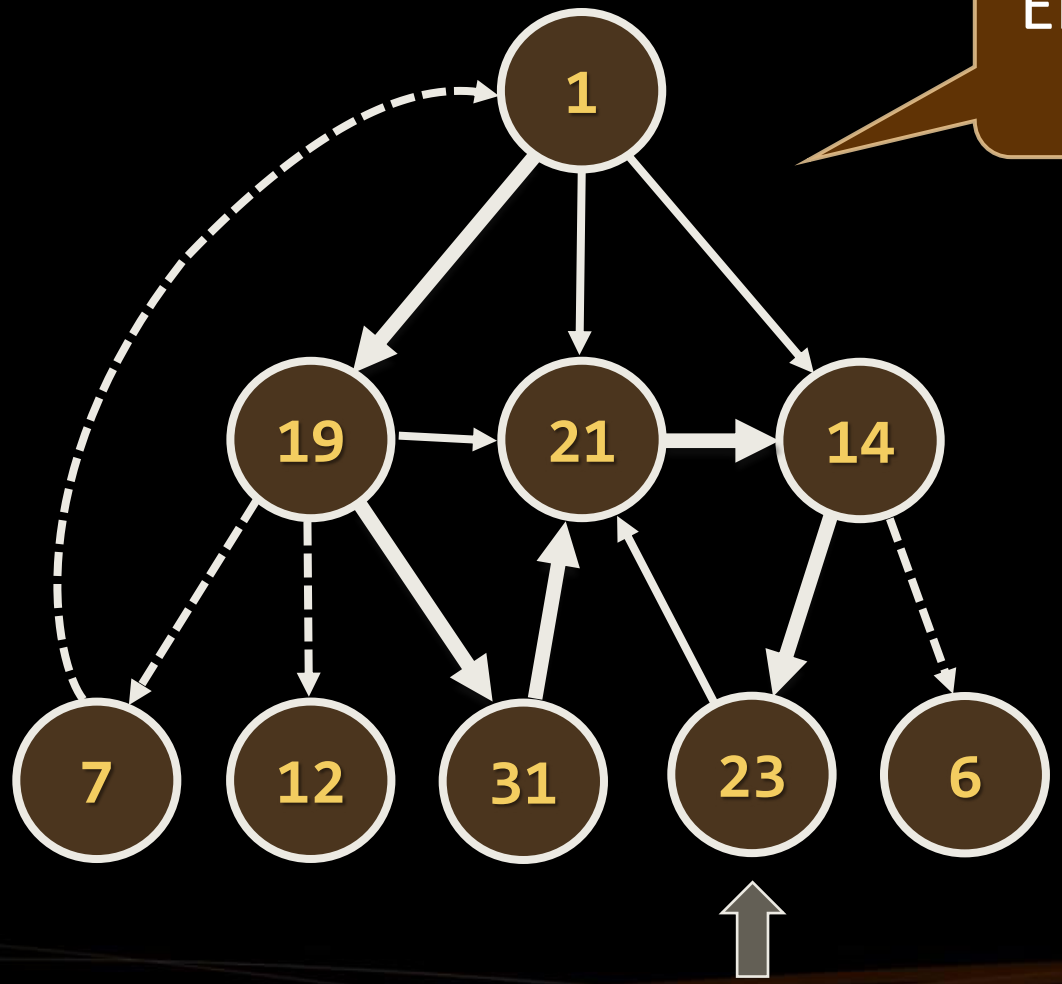


Return back from recursion and print the last visited node

Print node 6

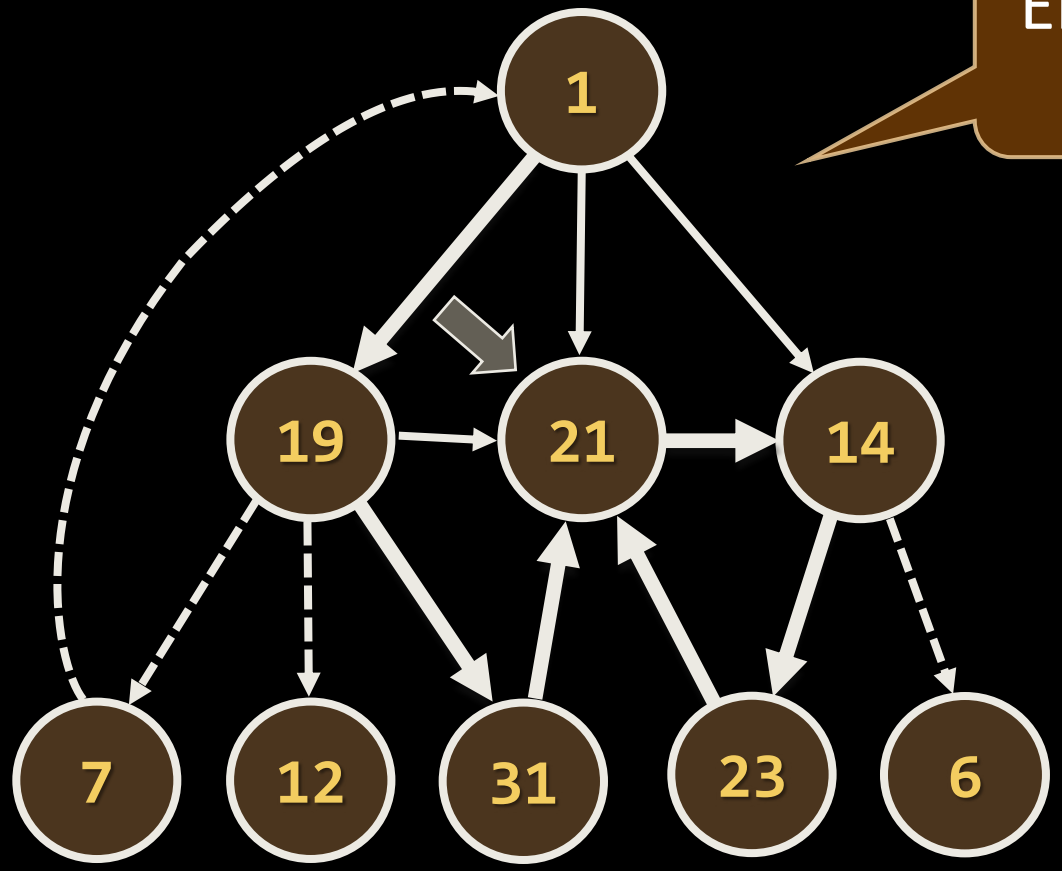
DFS in Action (Step 14)

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



DFS in Action (Step 15)

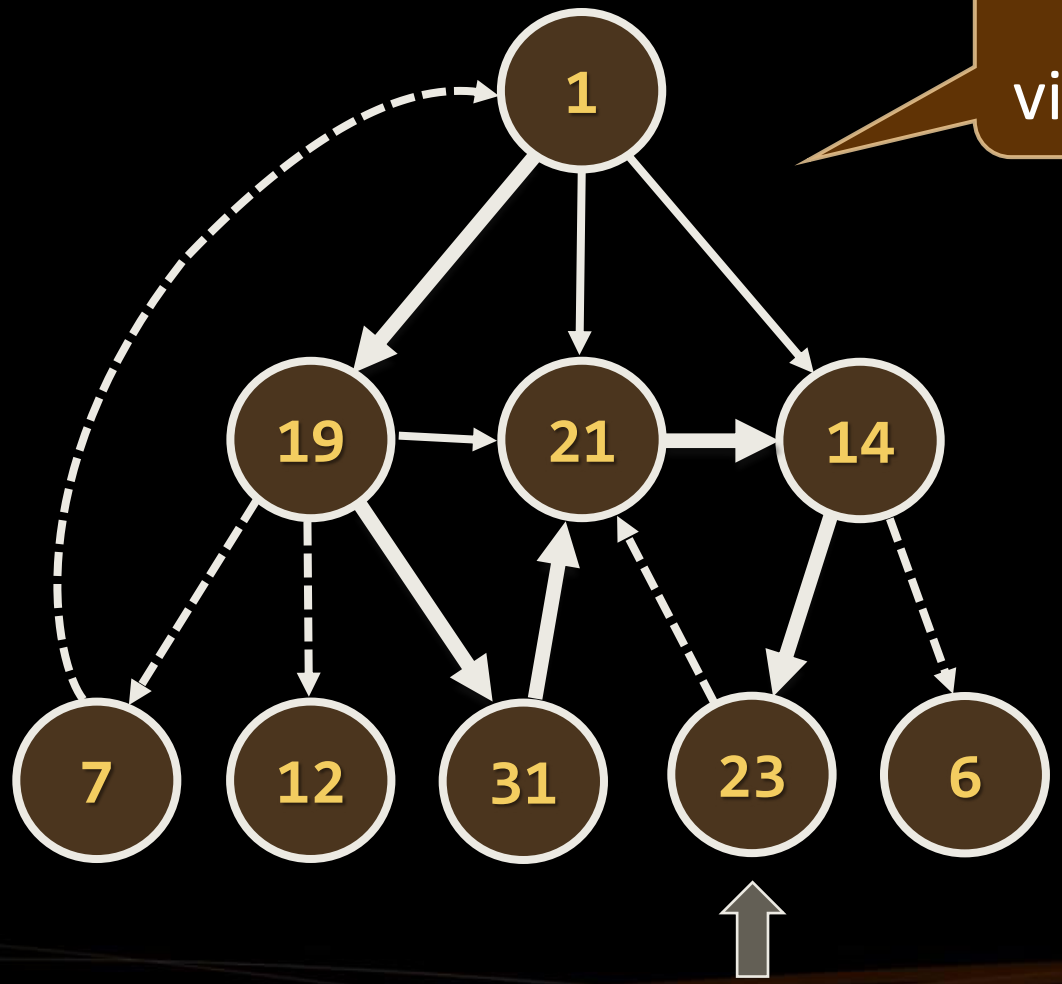
- Stack: 1, 19, 31, 21, 14, 23, 21
- Output: 7, 12, 6



Enter recursively into the first child 21

DFS in Action (Step 16)

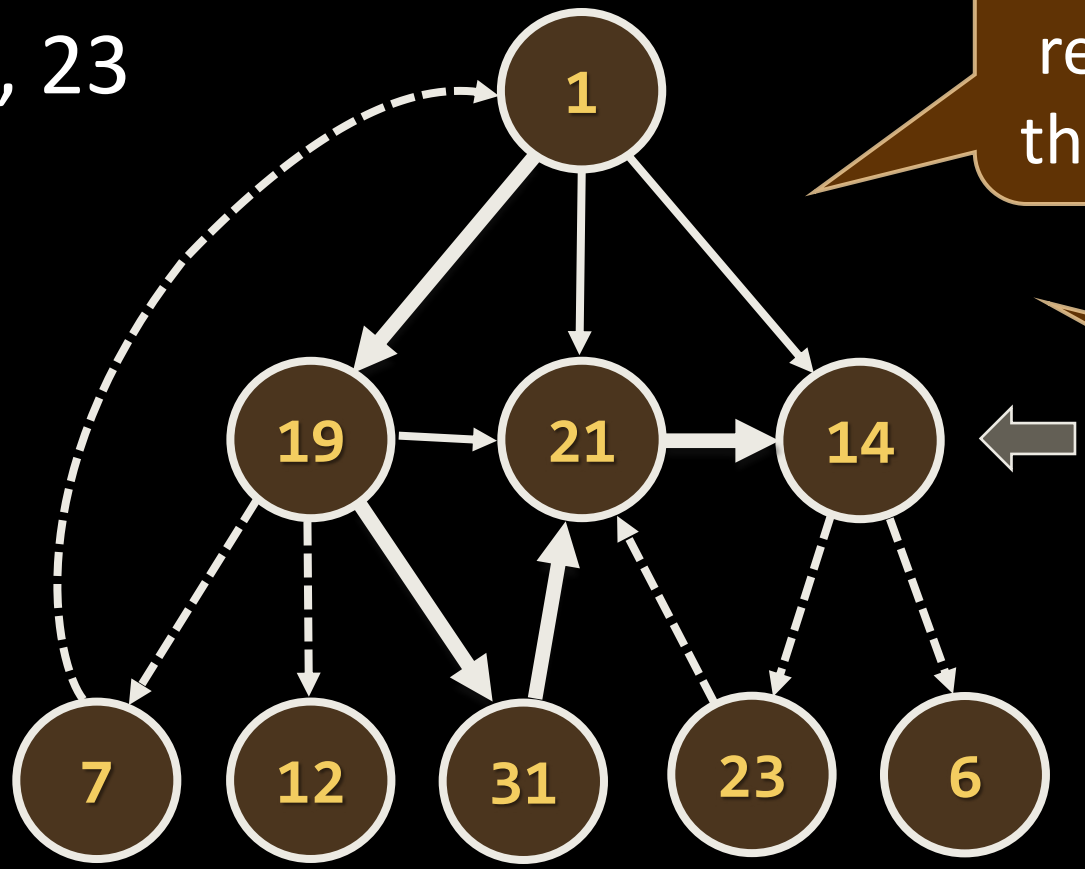
- Stack: 1, 19, 31, 21, 14, 23
- Output: 7, 12, 6



Node 21 already visited → return back

DFS in Action (Step 17)

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

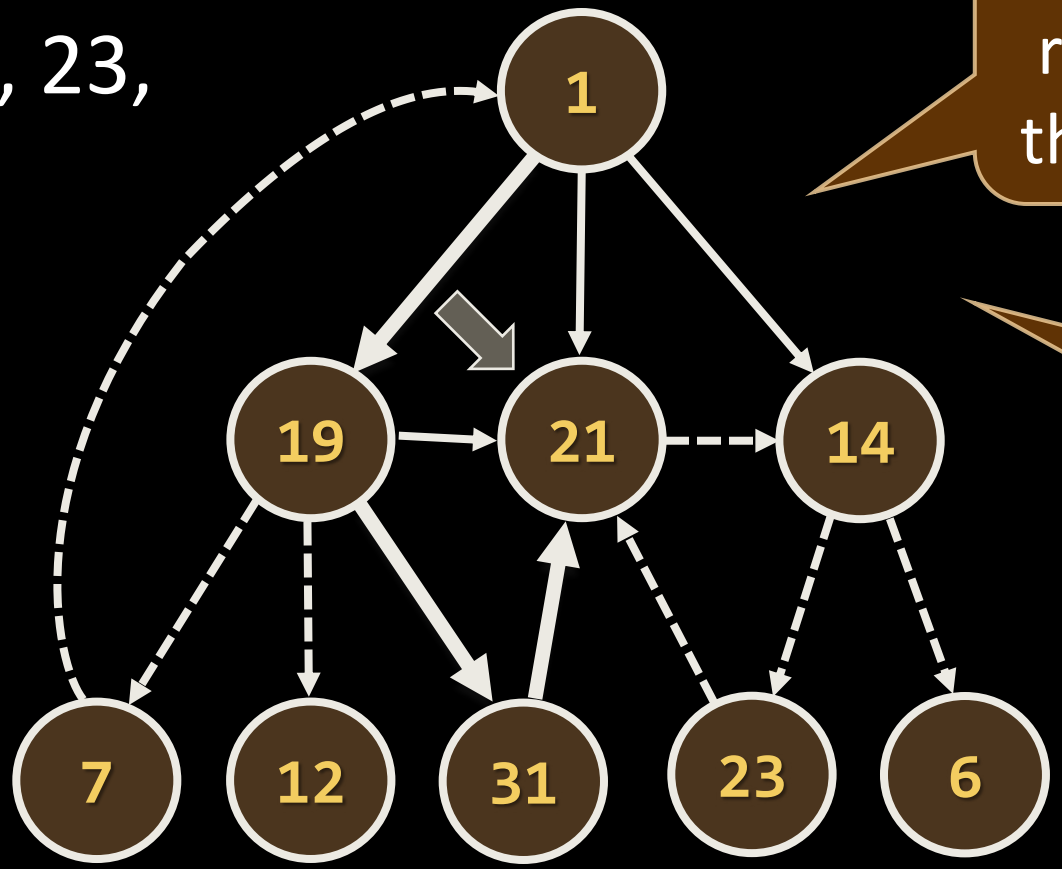


Return back from recursion and print the last visited node

Print node 23

DFS in Action (Step 18)

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

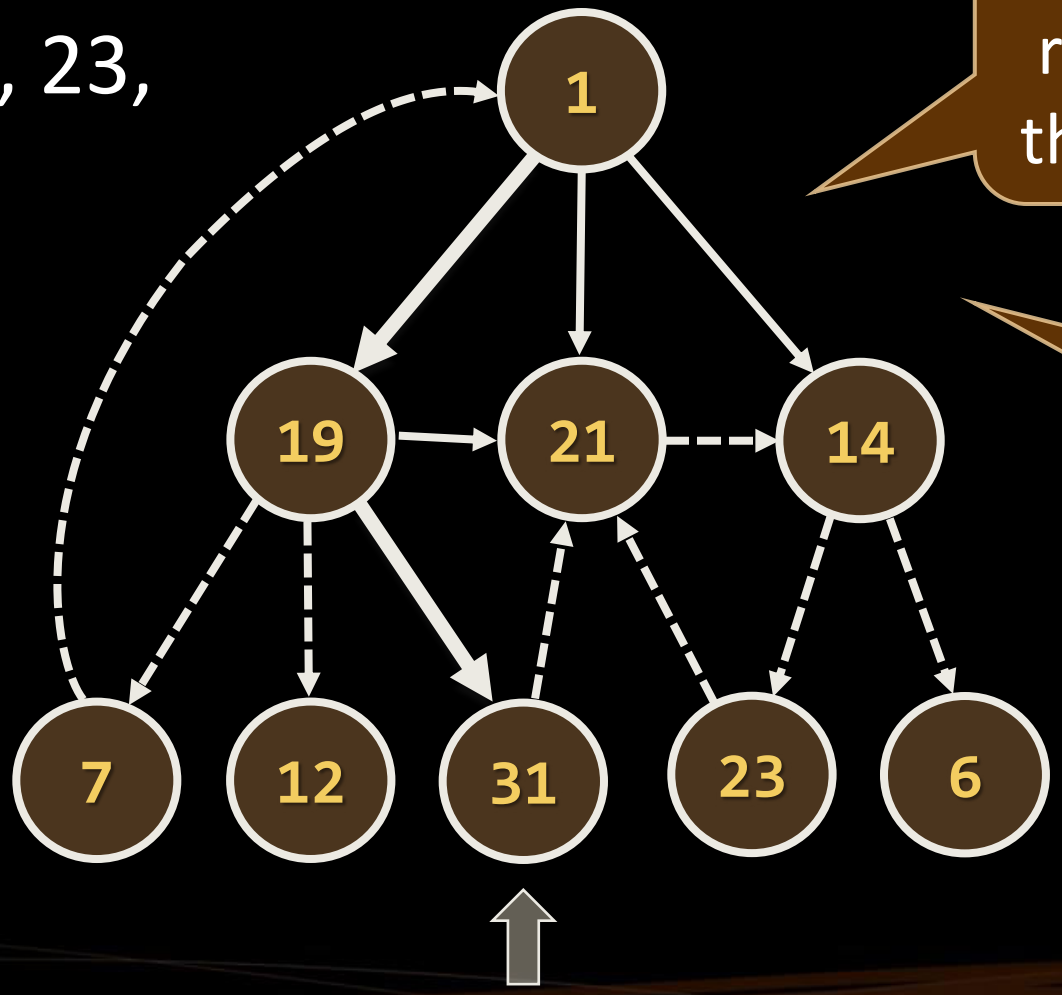


Return back from recursion and print the last visited node

Print node 14

DFS in Action (Step 19)

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

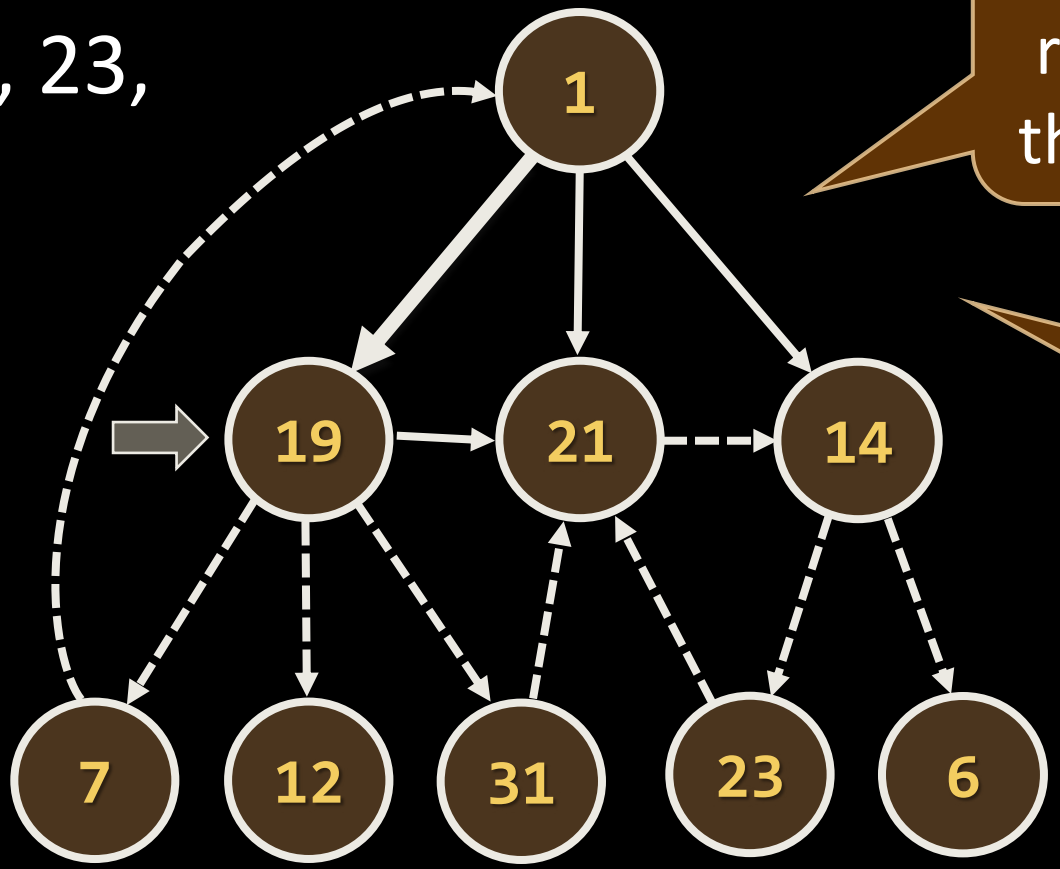


Return back from recursion and print the last visited node

Print node 21

DFS in Action (Step 20)

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

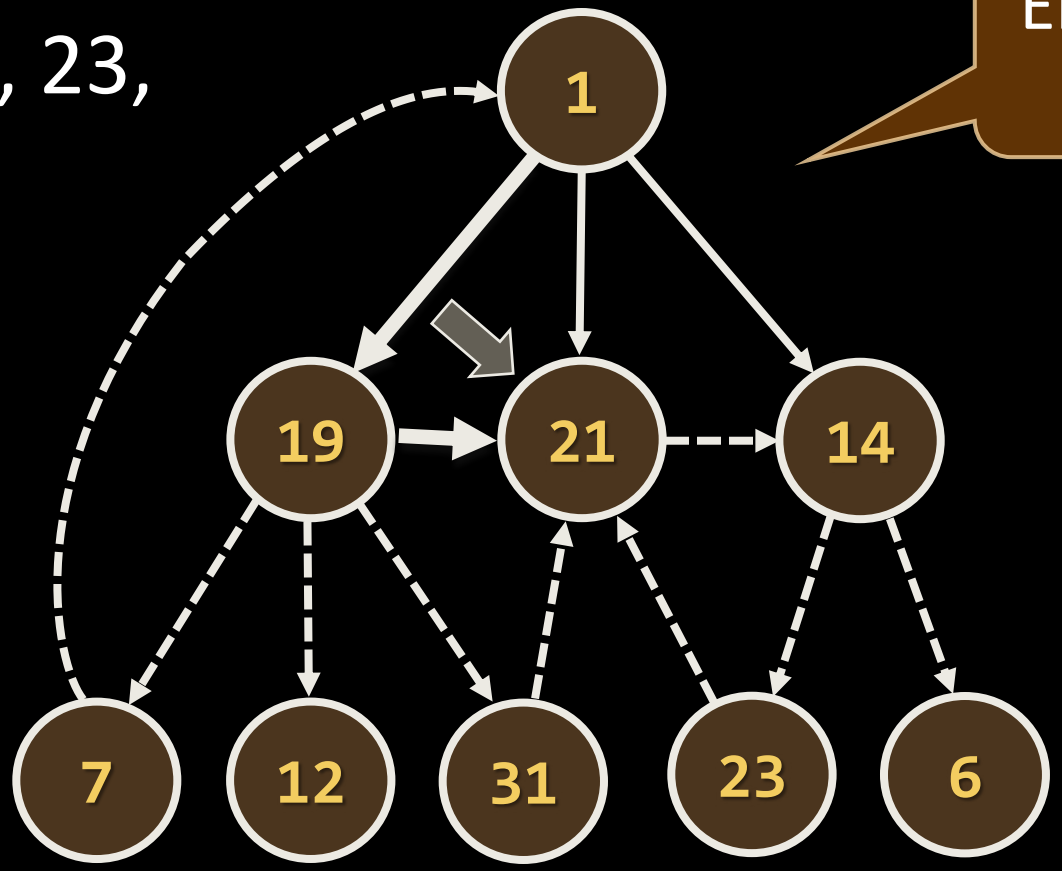


Return back from recursion and print the last visited node

Print node 31

DFS in Action (Step 21)

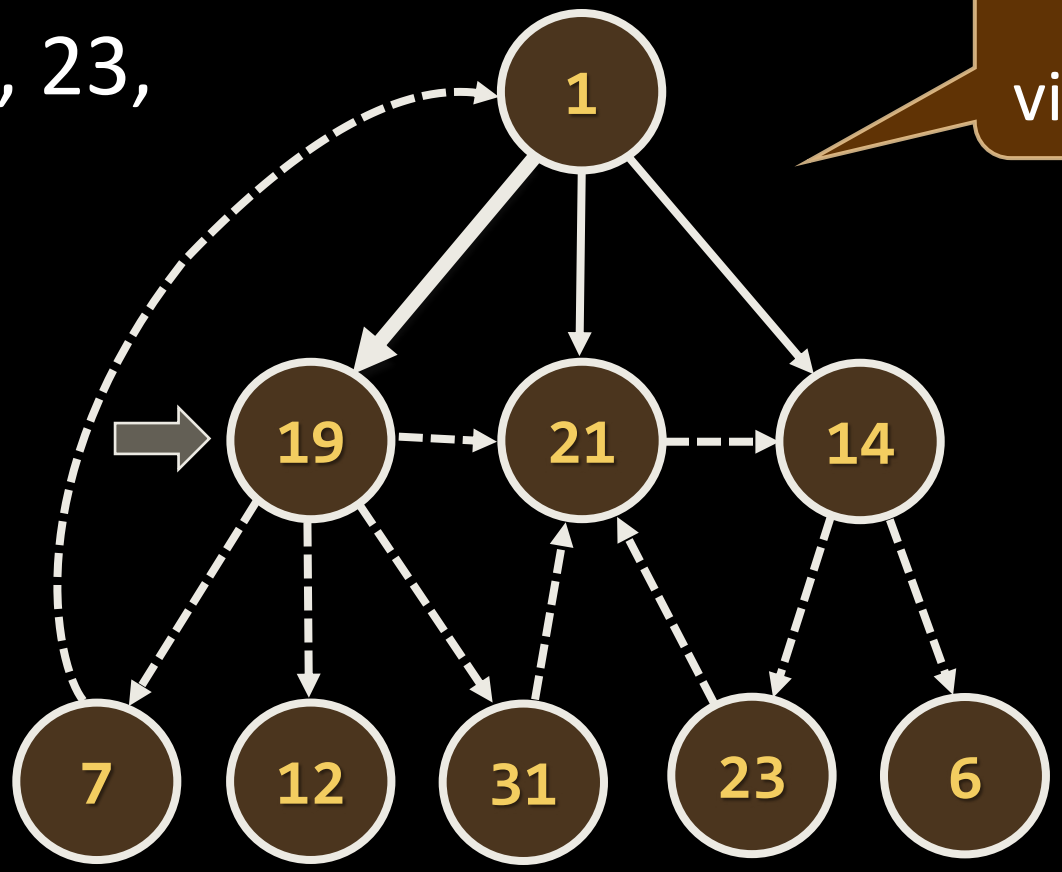
- Stack: 1, 19, 21
- Output: 7, 12, 6, 23, 14, 21, 31



Enter recursively into the next child 21

DFS in Action (Step 22)

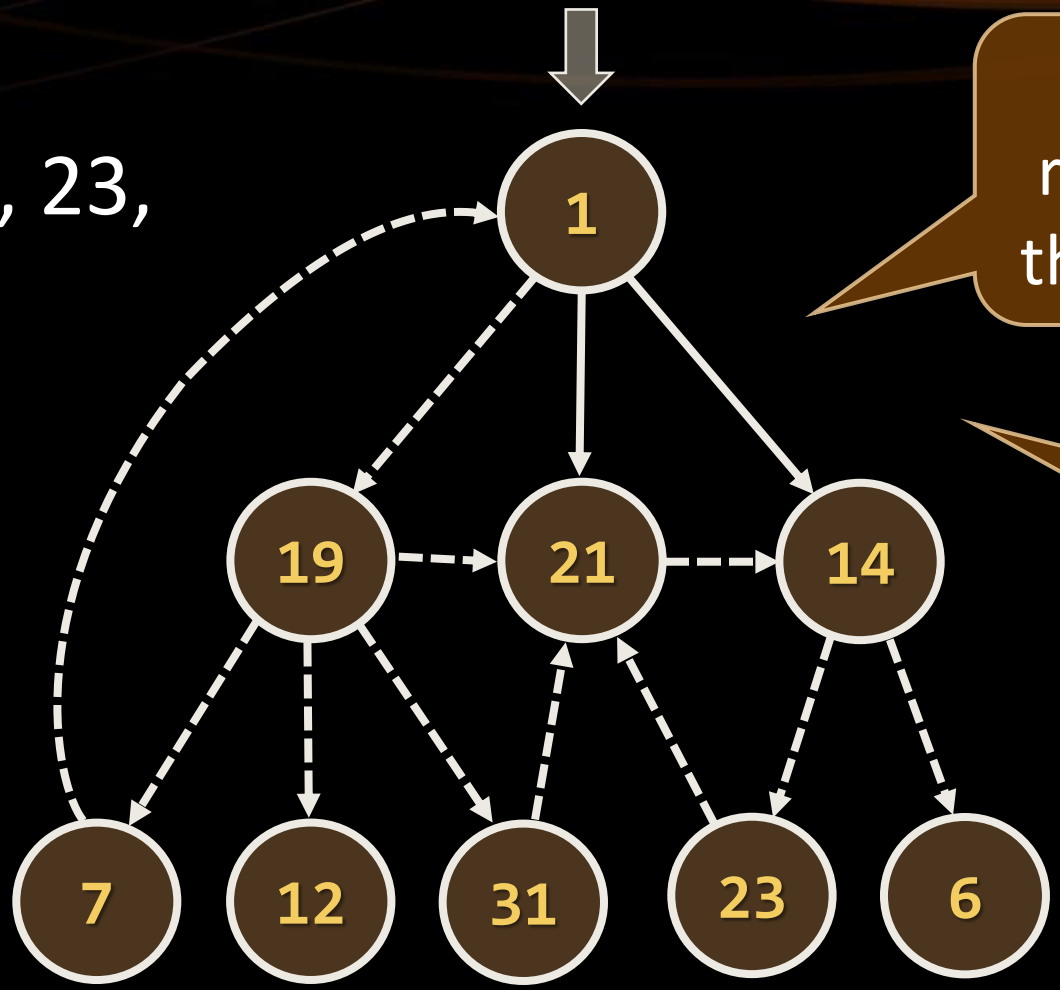
- Stack: 1, 19
- Output: 7, 12, 6, 23, 14, 21, 31



Node 21 already visited → return back

DFS in Action (Step 23)

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

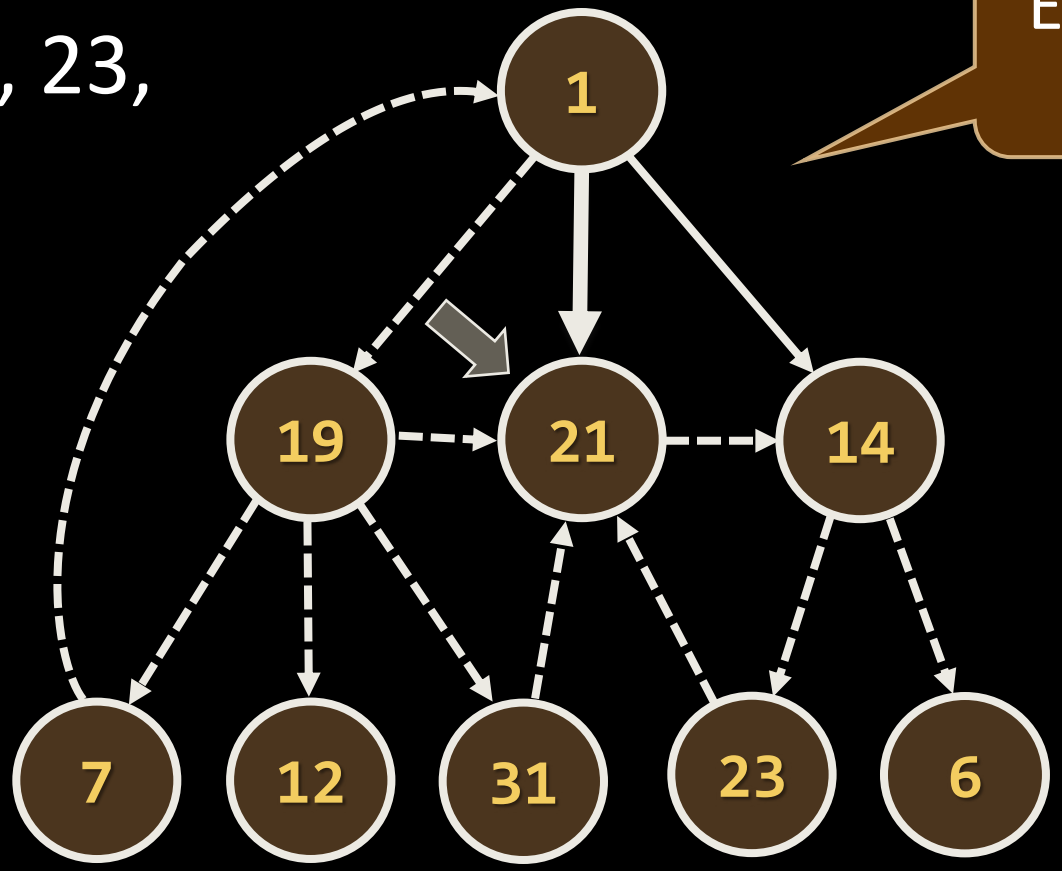


Return back from recursion and print the last visited node

Print node 19

DFS in Action (Step 24)

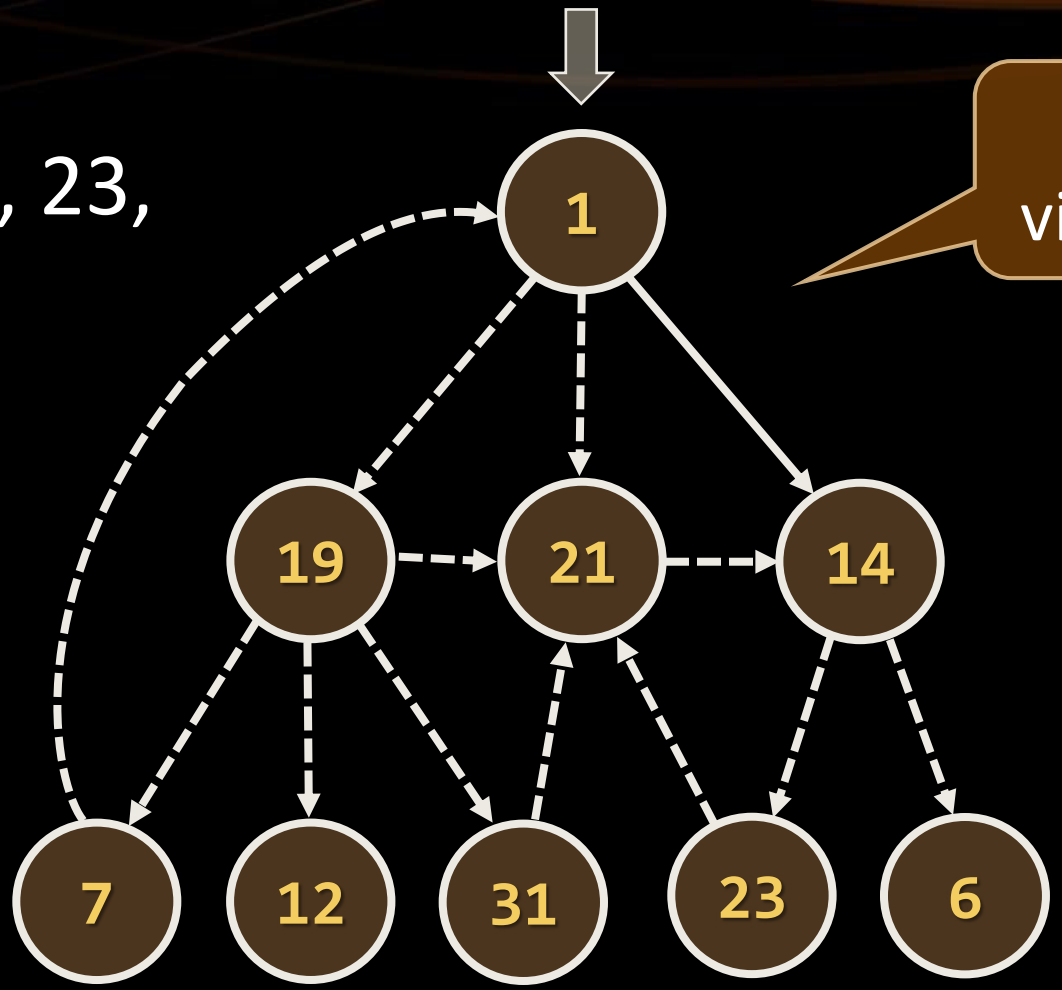
- Stack: 1, 21
- Output: 7, 12, 6, 23, 14, 21, 31, 19



Enter recursively into the next child 21

DFS in Action (Step 25)

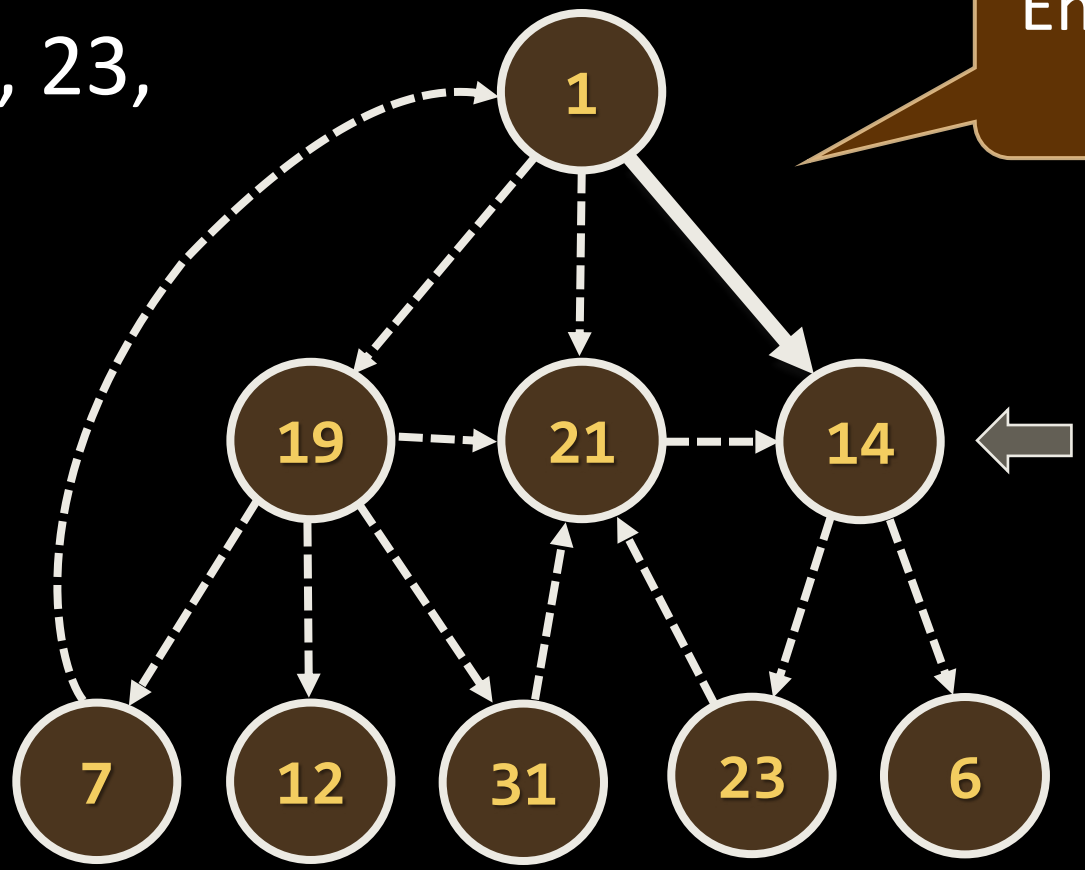
- Stack: 1
- Output: 7, 12, 6, 23, 14, 21, 31, 19



Node 21 already visited → return back

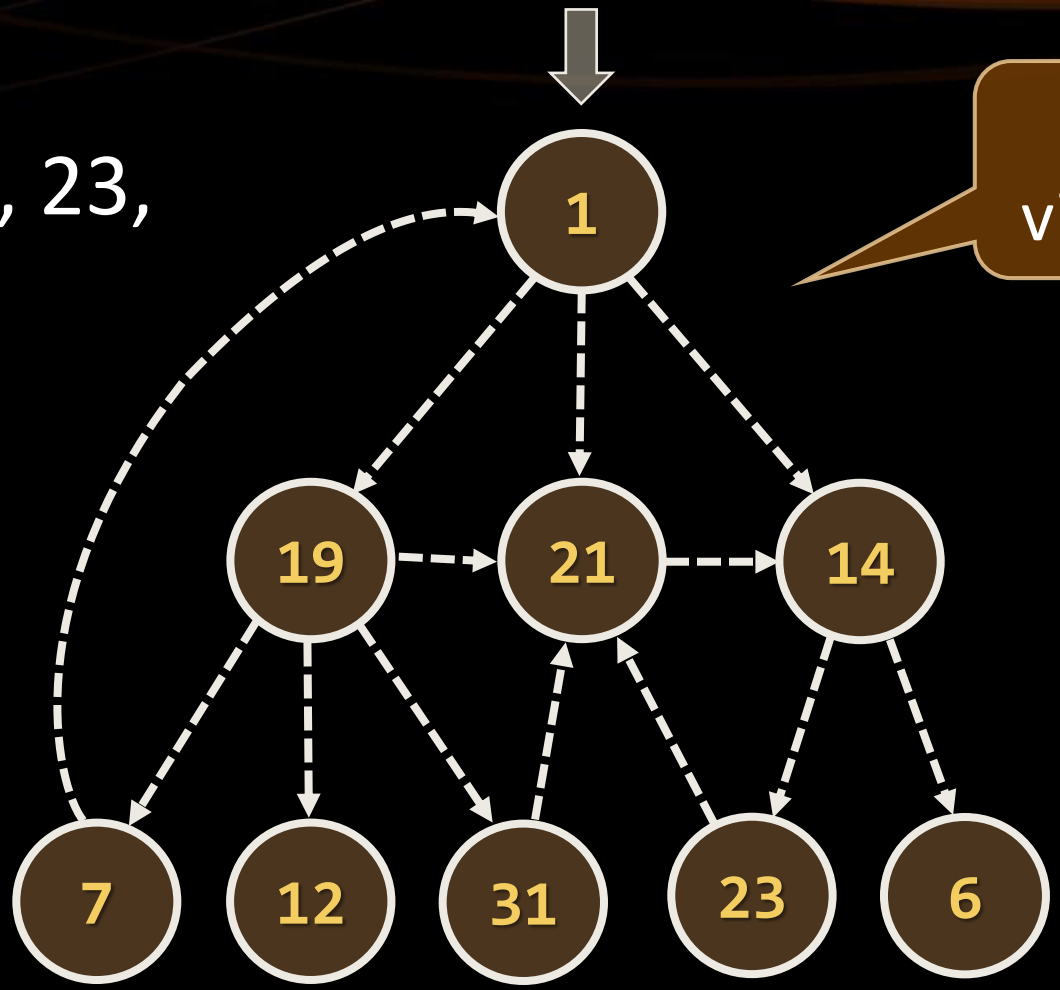
DFS in Action (Step 26)

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



DFS in Action (Step 27)

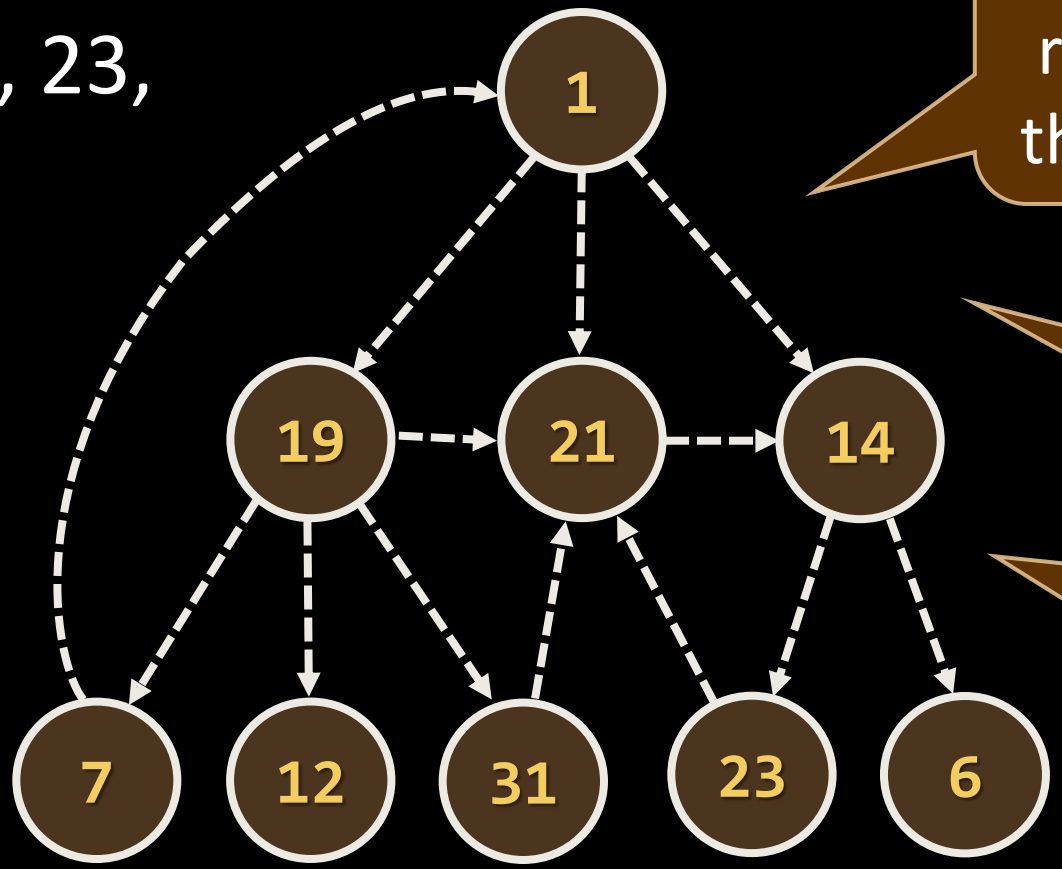
- Stack: 1
- Output: 7, 12, 6, 23, 14, 21, 31, 19



Node 14 already visited → return back

DFS in Action (Step 28)

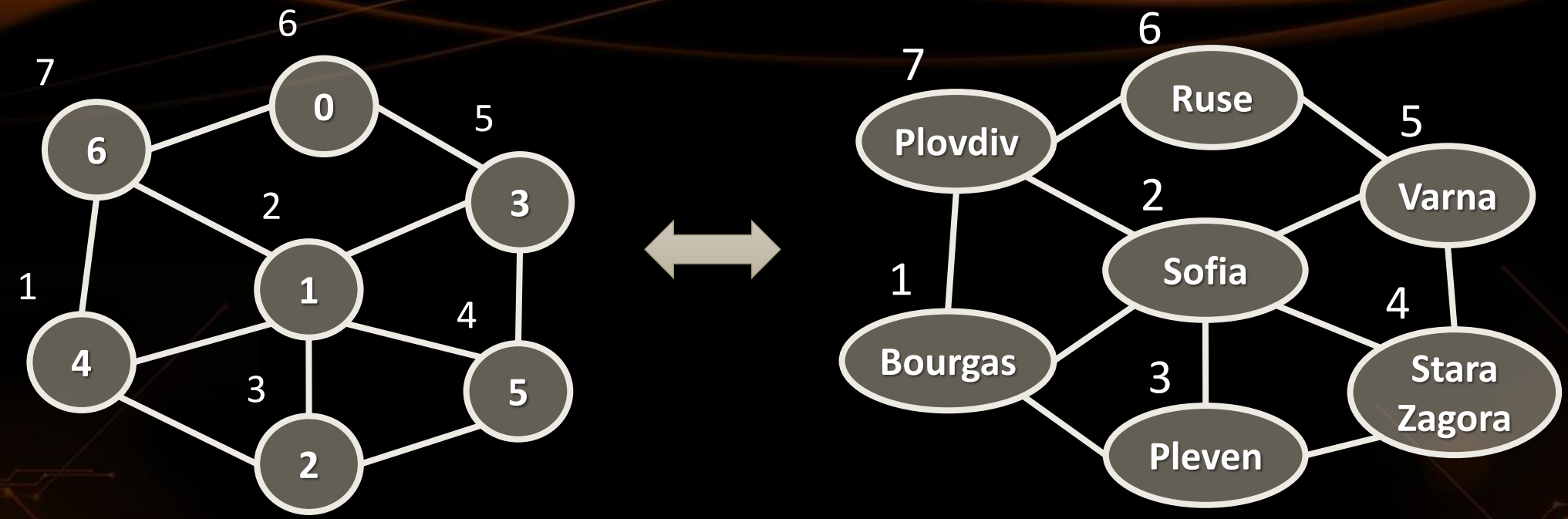
- Stack: (empty)
- Output: 7, 12, 6, 23, 14, 21, 31, 19, 1



Return back from recursion and print the last visited node

Print node 1

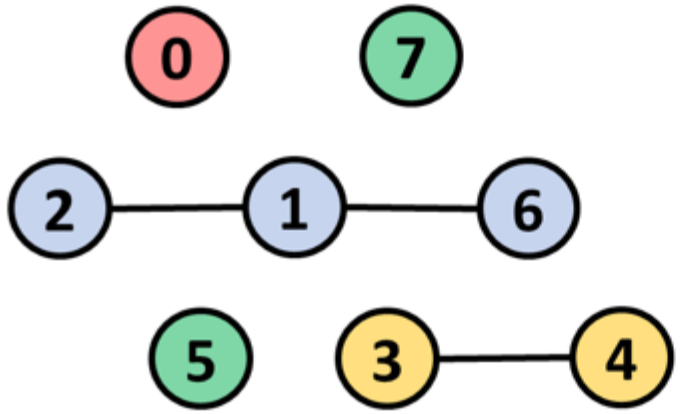
DFS traversal completed



Recursive DFS Graph Traversal

Live Demo

SoftUni Foundation

Input	Graph	Expected Output
7 2 6 1 4 3 1		Connected component: 0 Connected component: 2 6 1 Connected component: 4 3 Connected component: 5

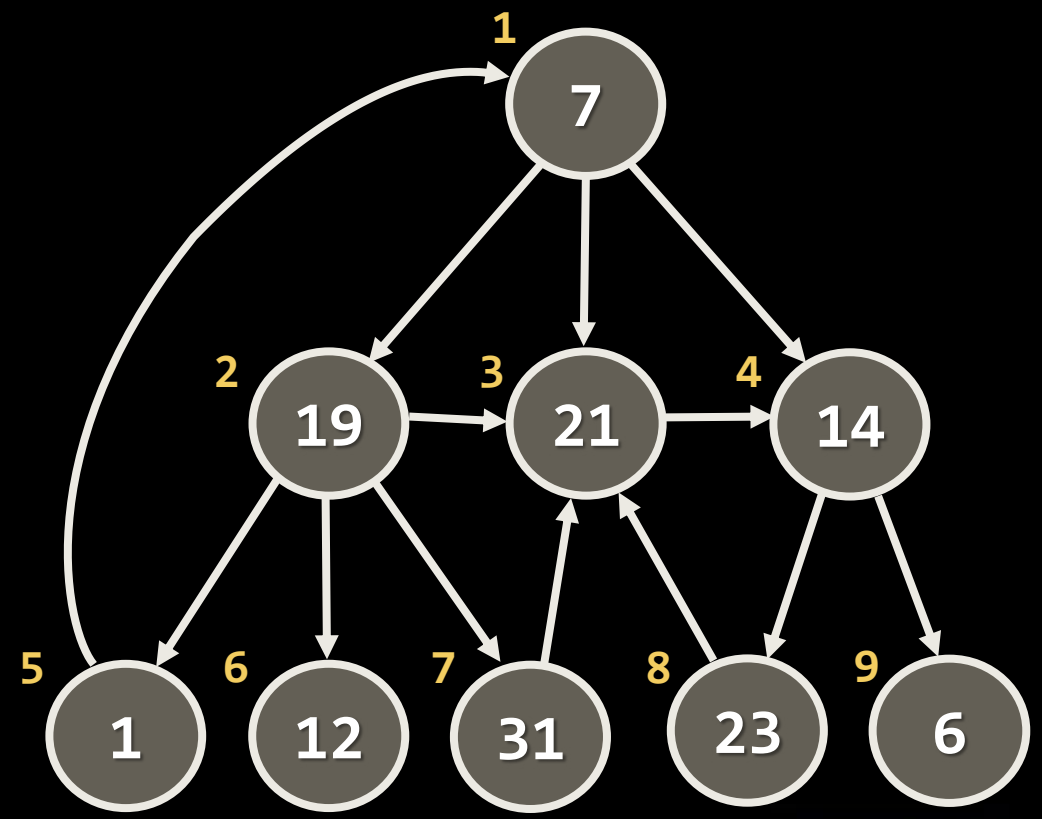
DFS for Graph Traversals

Live Coding (Lab)

Breadth-First Search (BFS)

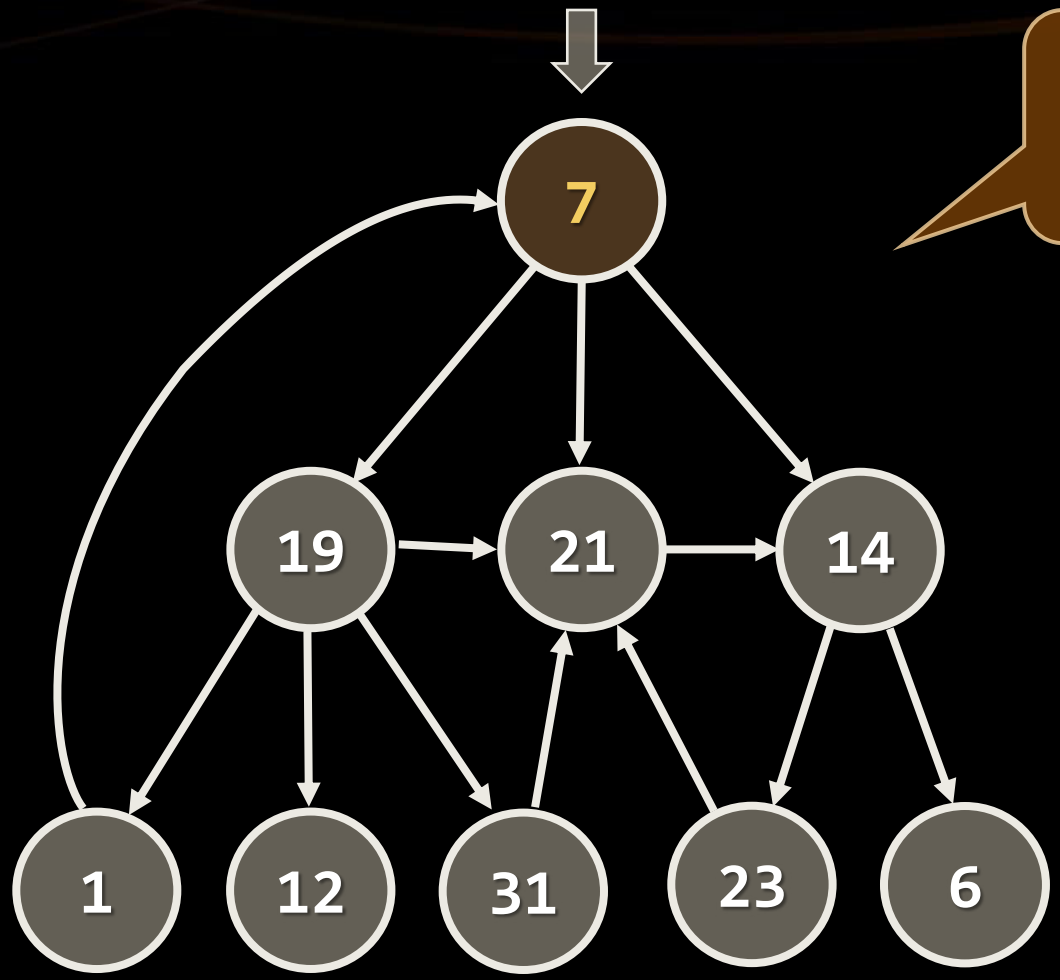
- **Breadth-First Search** (BFS) first visits the neighbor nodes, then the neighbors of neighbors, then their neighbors, etc.

```
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)

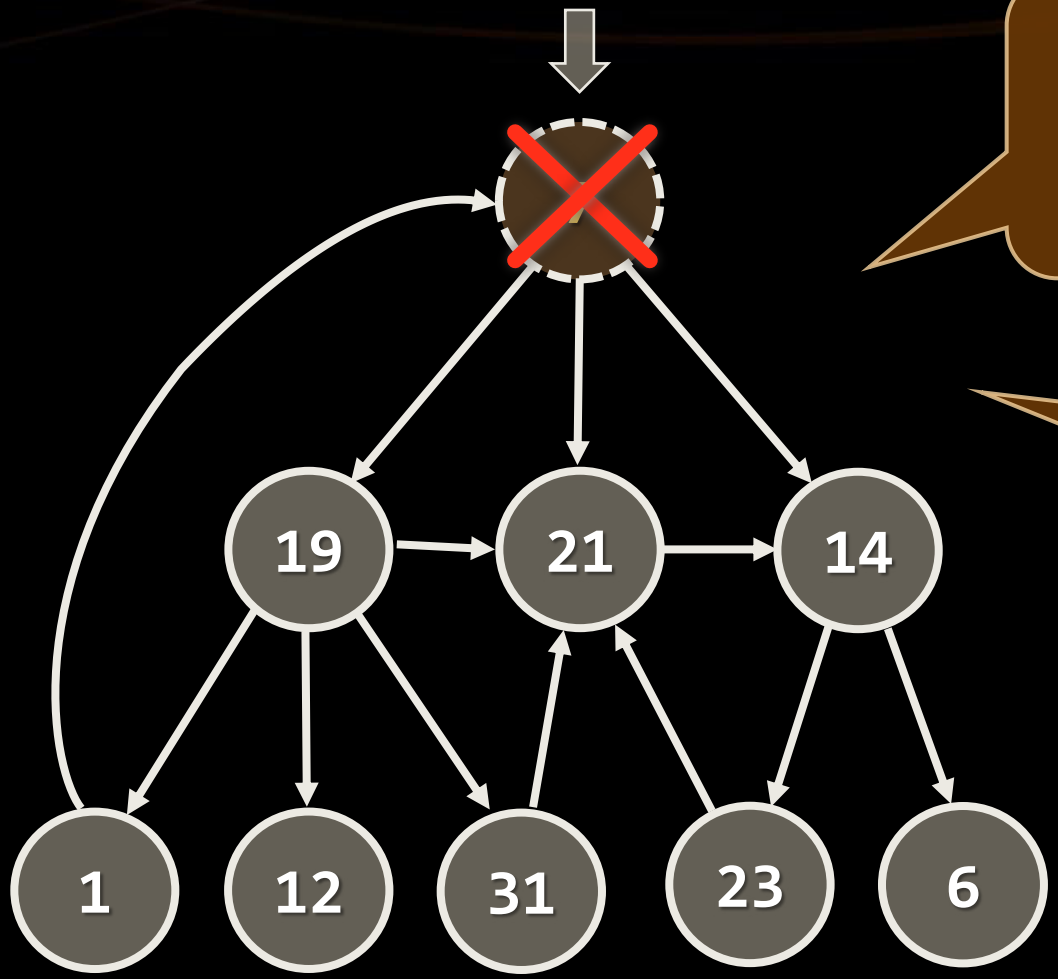
- Queue: 7
- Output:



Initially enqueue the root node 7

BFS in Action (Step 2)

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

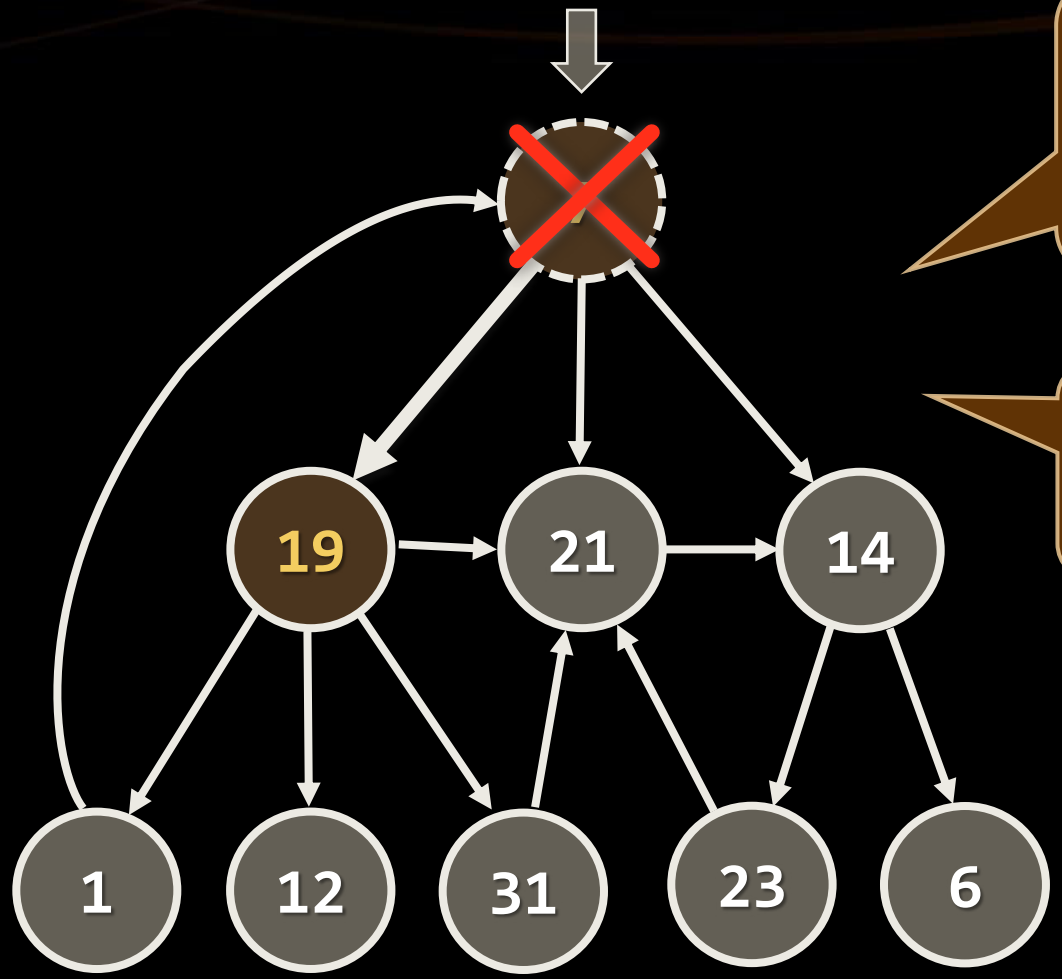


Remove from the queue the next node and print it

Print node 7

BFS in Action (Step 3)

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

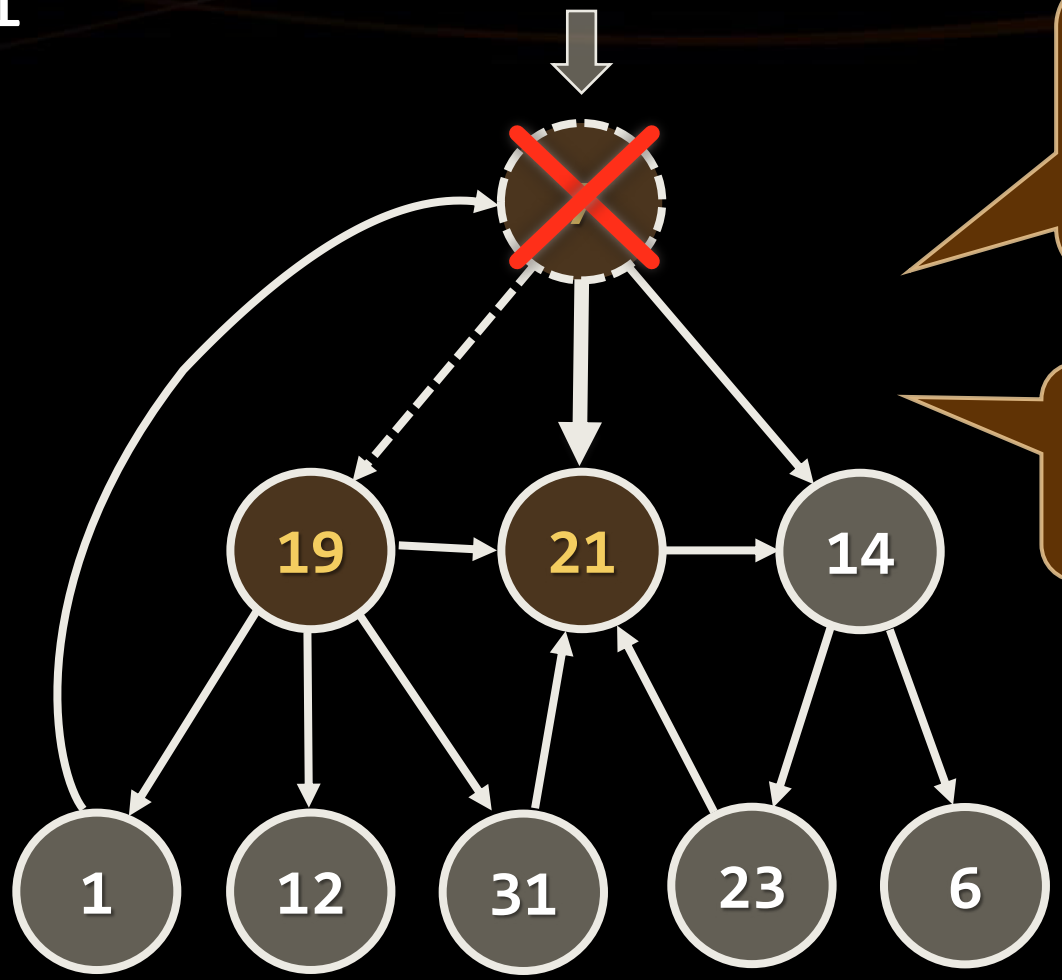


Enqueue all children of the current node

Enqueue the first child 19

BFS in Action (Step 4)

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

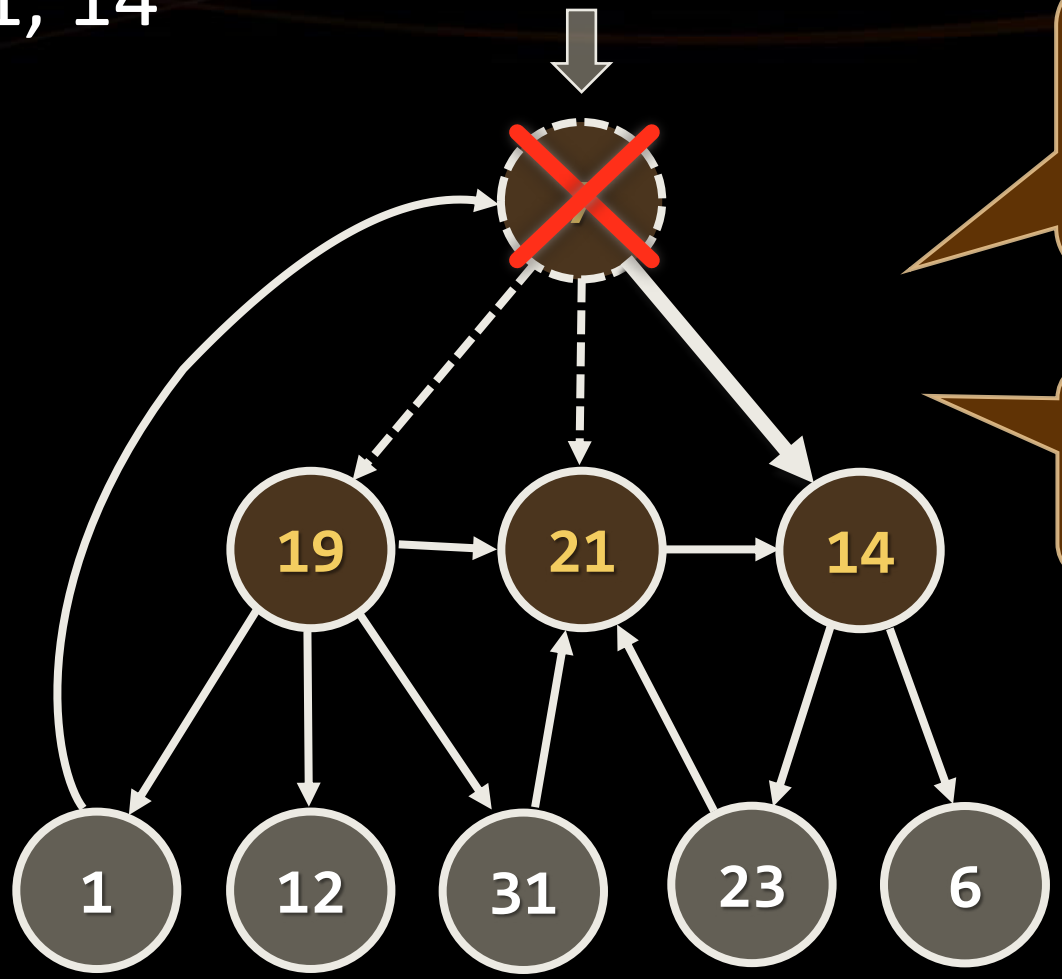


Enqueue all children of the current node

Enqueue the second child 21

BFS in Action (Step 5)

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

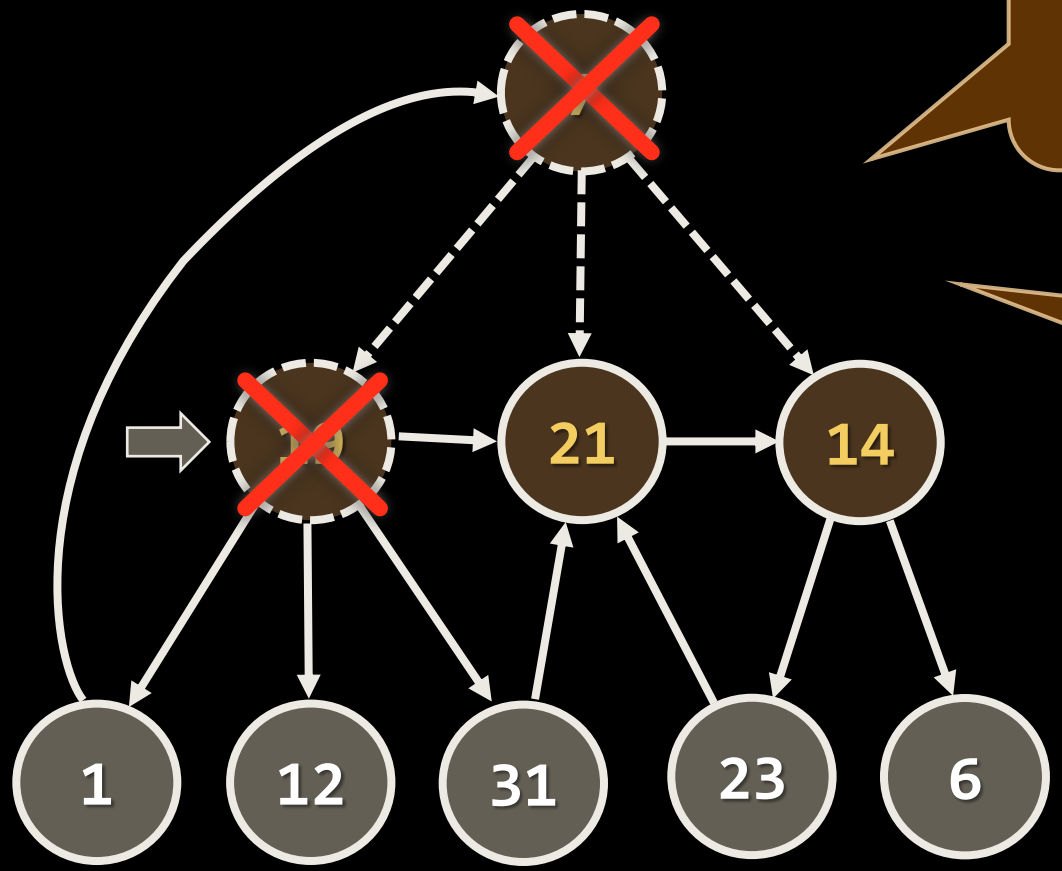


Enqueue all children of the current node

Enqueue the third child 14

BFS in Action (Step 6)

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

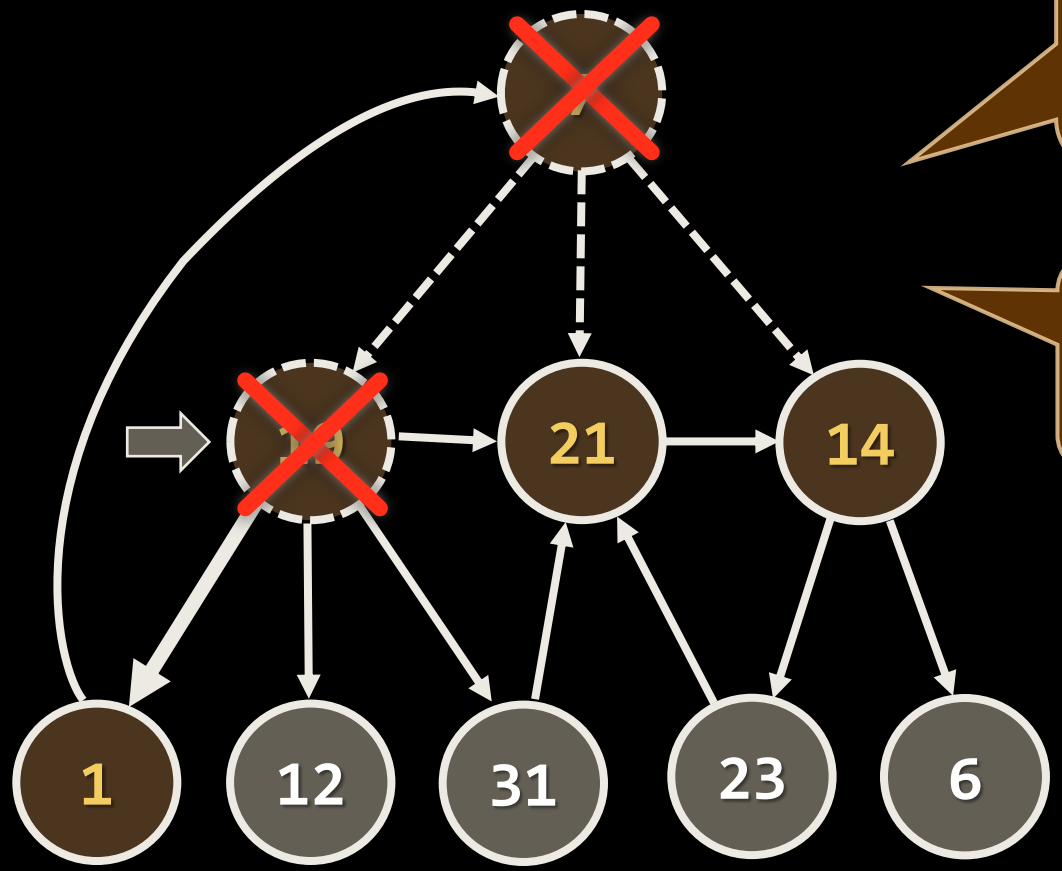


Remove from the queue the next node and print it

Print node 19

BFS in Action (Step 7)

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

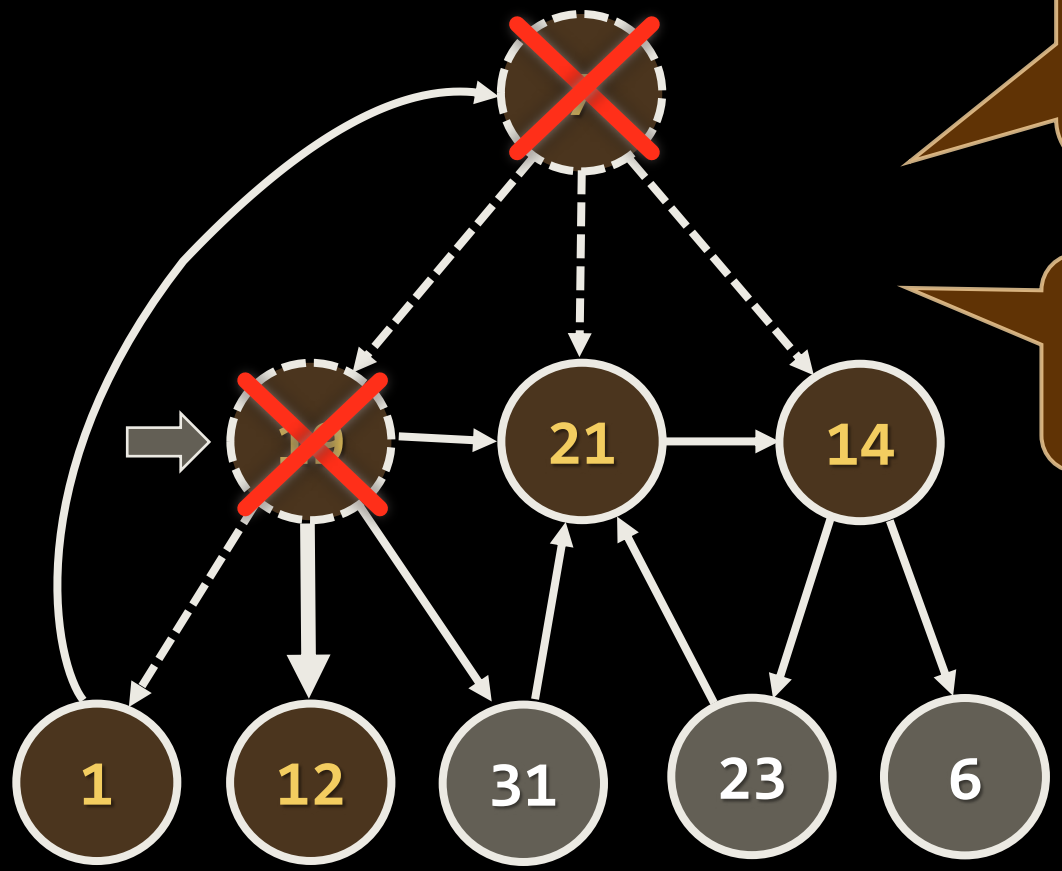


Enqueue all children of the current node

Enqueue the first child 1

BFS in Action (Step 8)

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

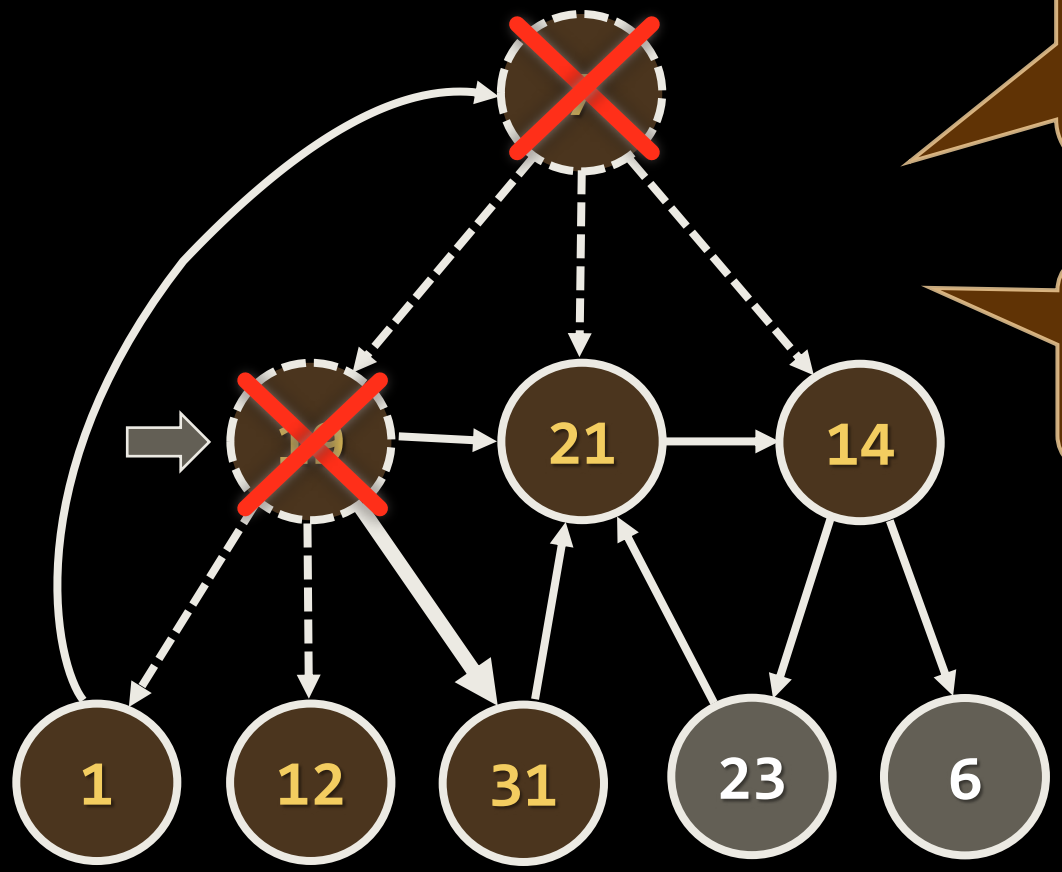


Enqueue all children of the current node

Enqueue the second child 12

BFS in Action (Step 9)

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

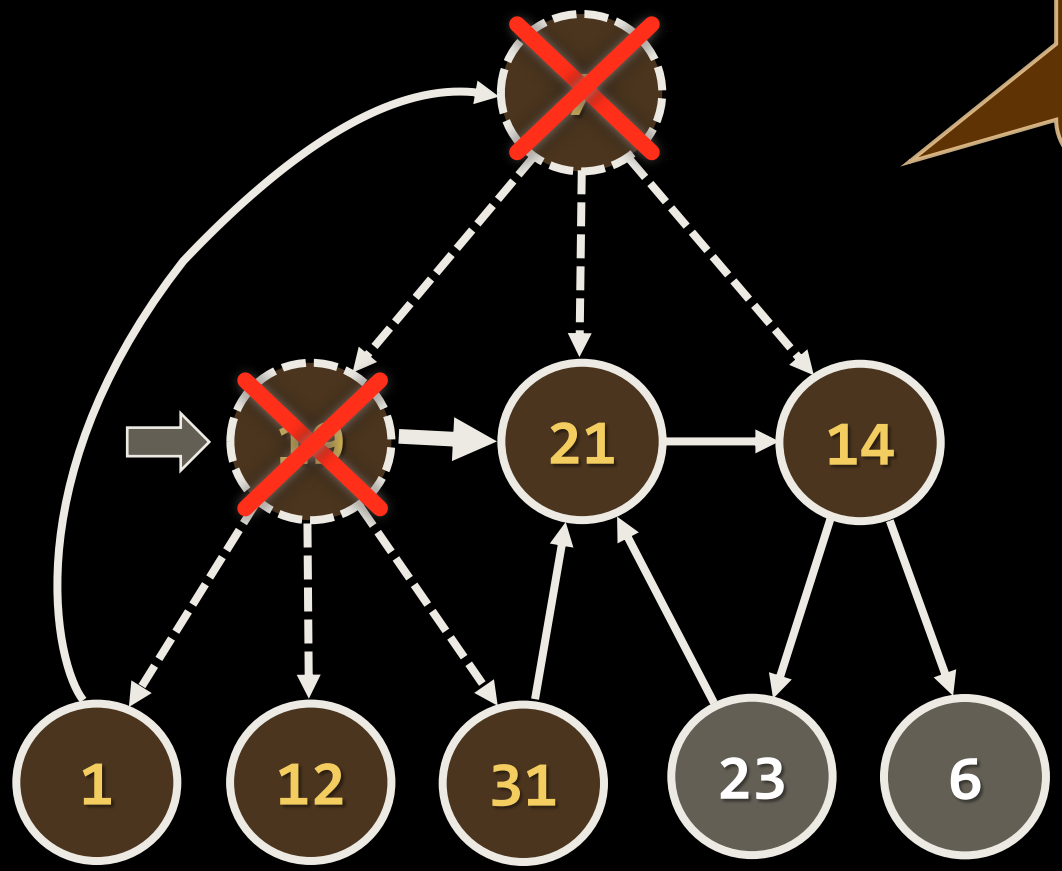


Enqueue all children of the current node

Enqueue the third child 31

BFS in Action (Step 10)

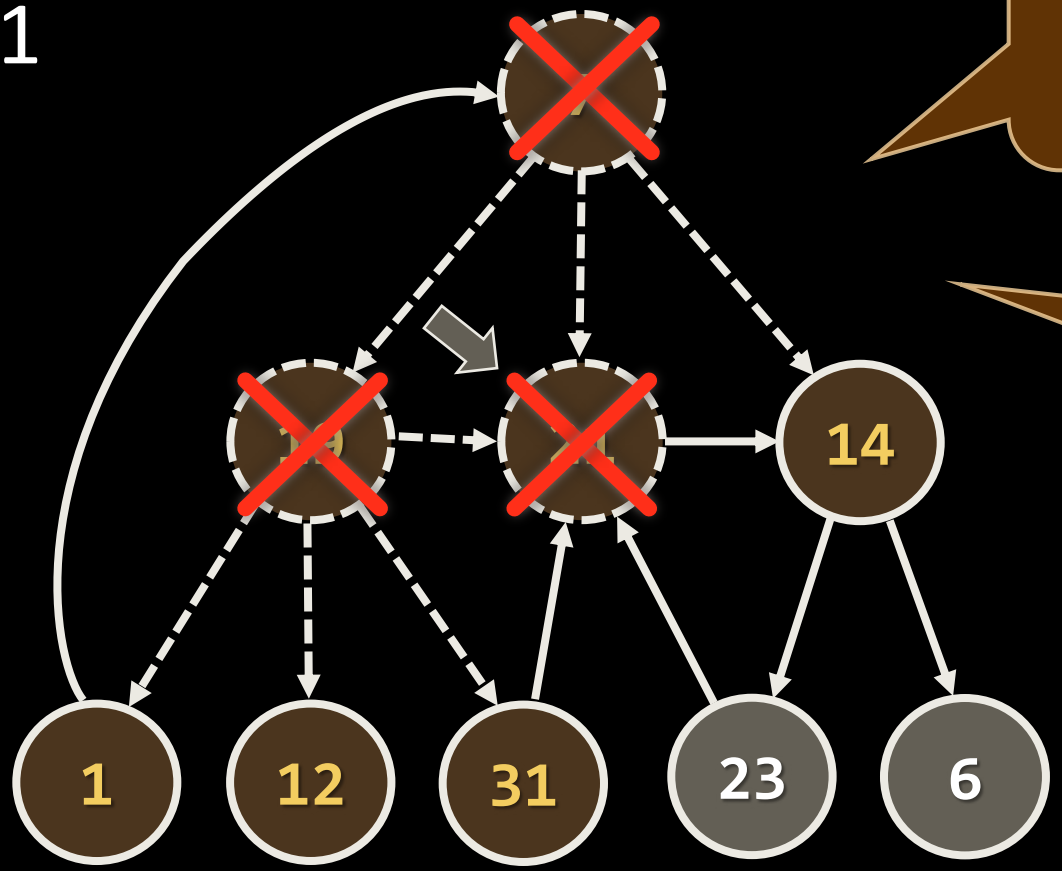
- Queue: ~~7~~, ~~19~~, 21, 14, 1, 12, 31
- Output: 7, 19



Child node 21 already visited → skip it

BFS in Action (Step 11)

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

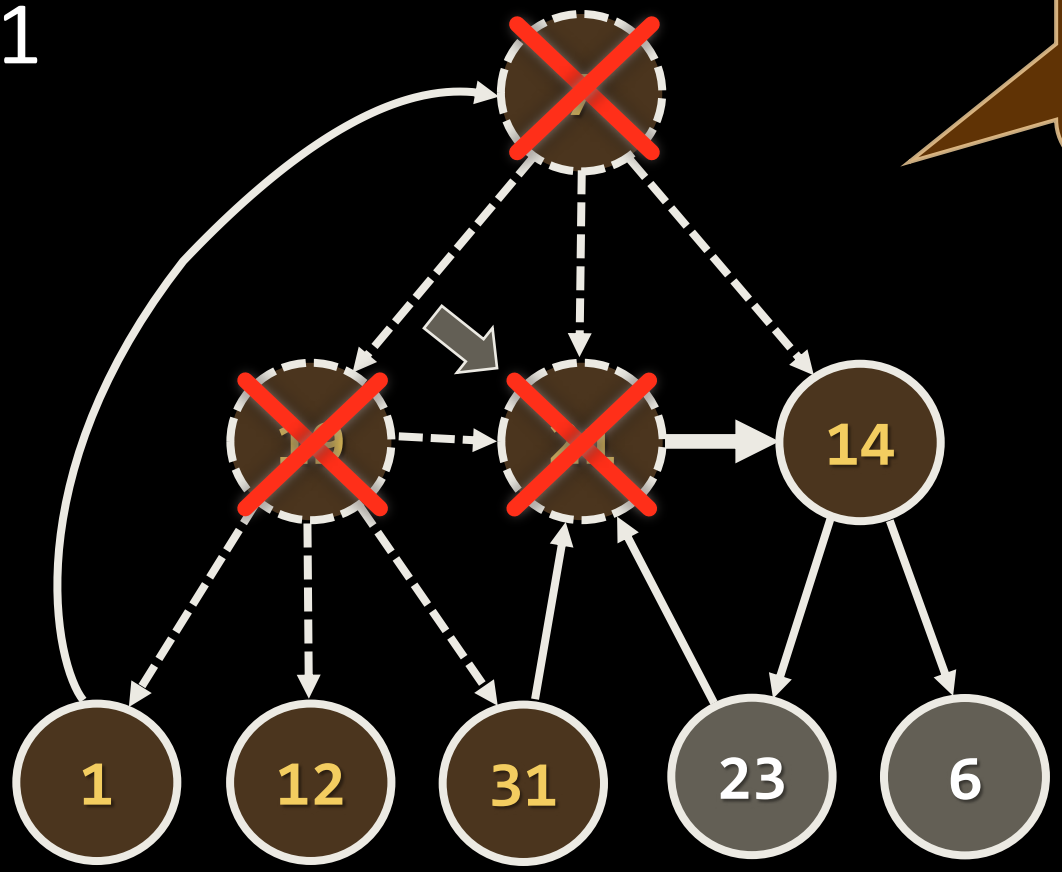


Remove from the queue the next node and print it

Print node 21

BFS in Action (Step 12)

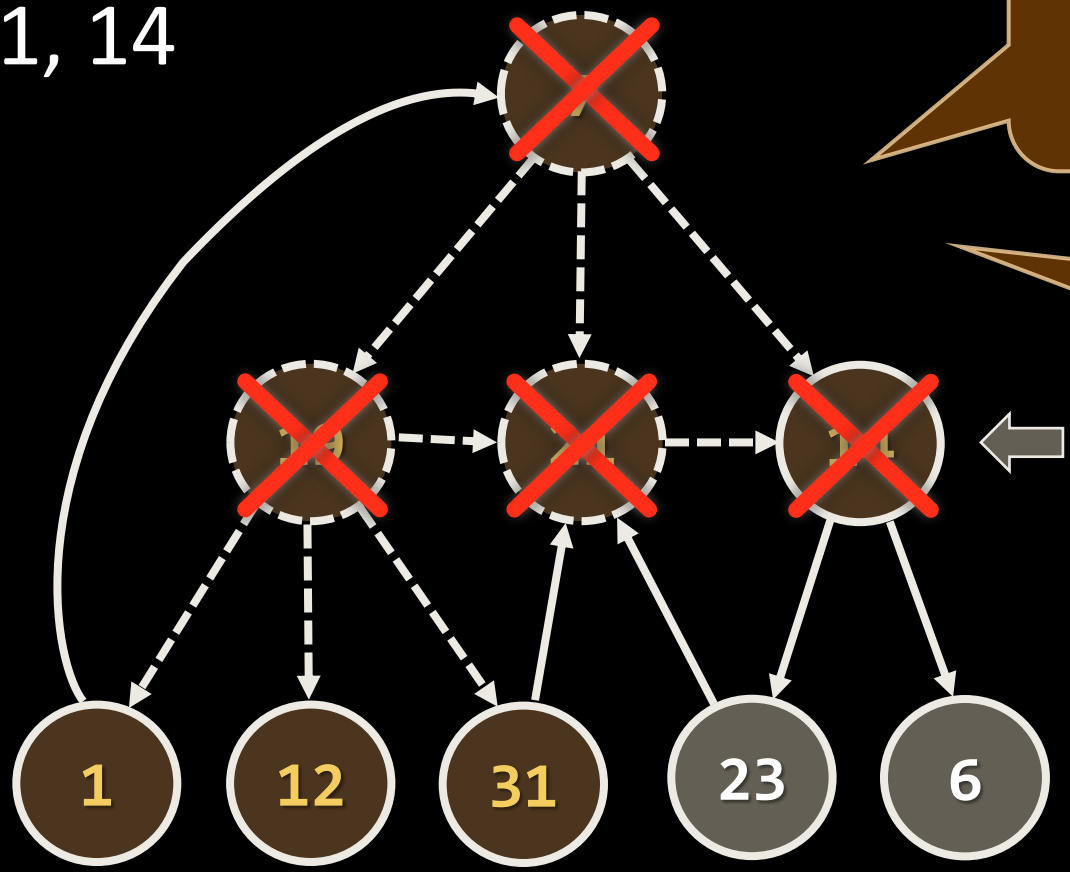
- Queue: ~~7~~, ~~19~~, ~~21~~, 14, 1, 12, 31
- Output: 7, 19, 21



Child node 14 already visited → skip it

BFS in Action (Step 13)

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

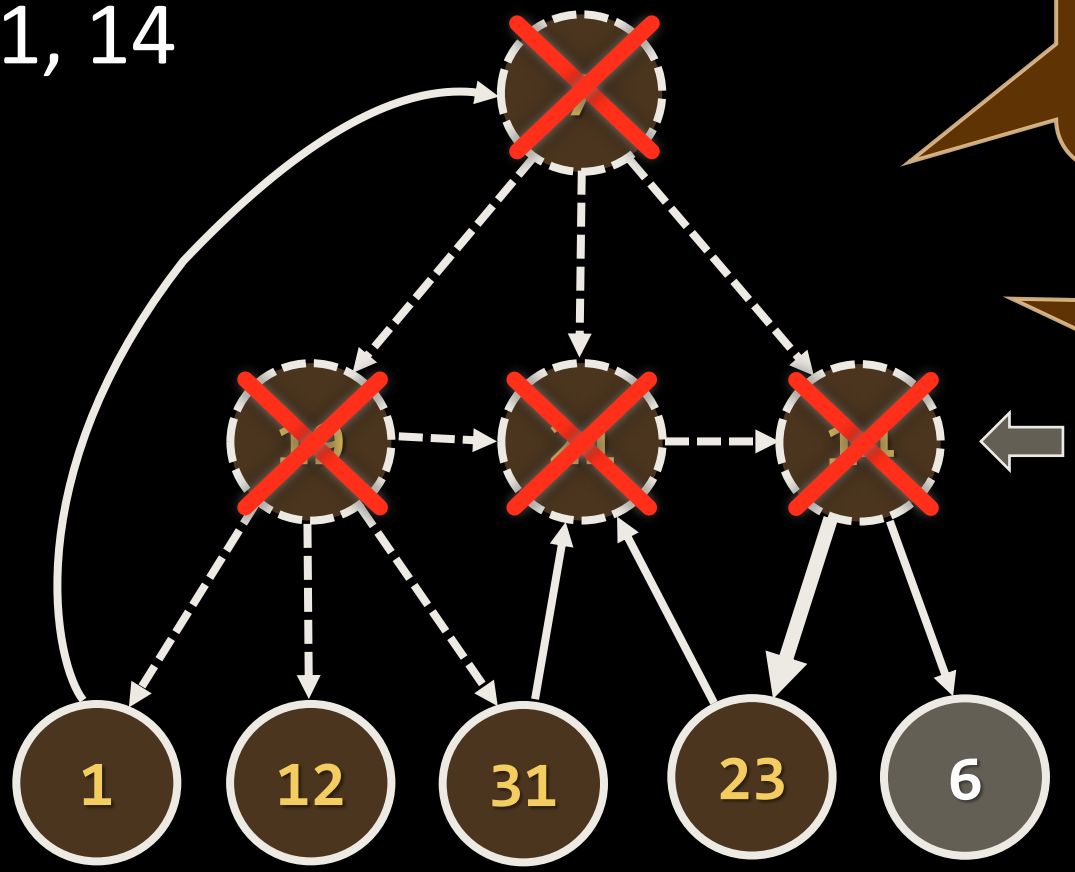


Remove from the queue the next node and print it

Print node 14

BFS in Action (Step 14)

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

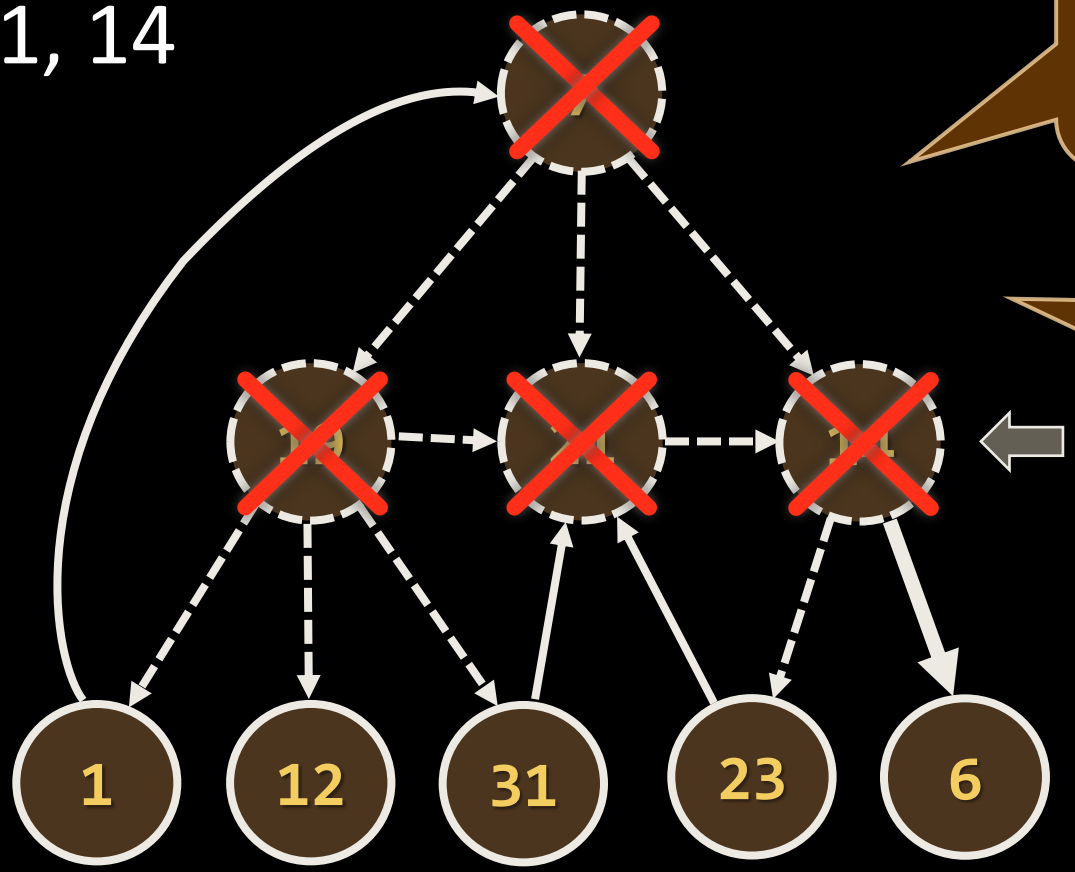


Enqueue all children of the current node

Enqueue the first child 23

BFS in Action (Step 15)

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

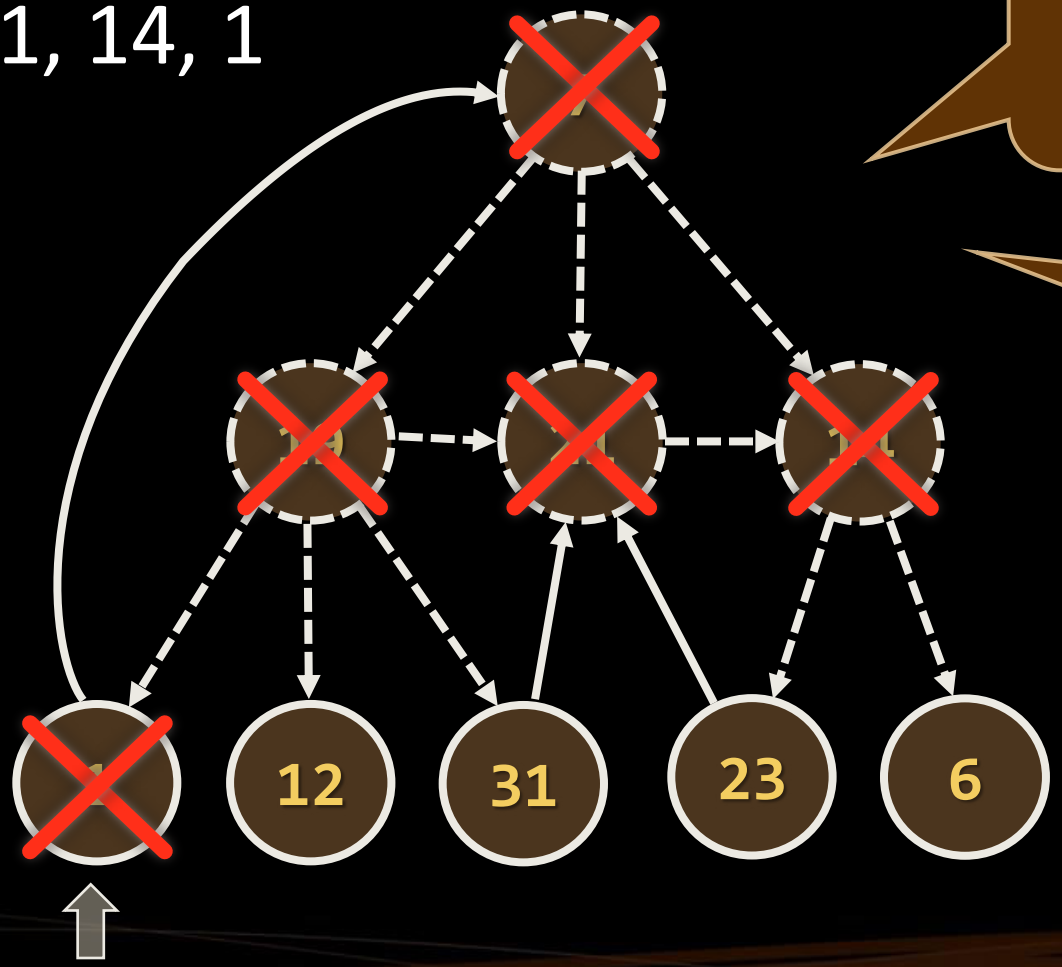


Enqueue all children of the current node

Enqueue the second child 6

BFS in Action (Step 16)

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

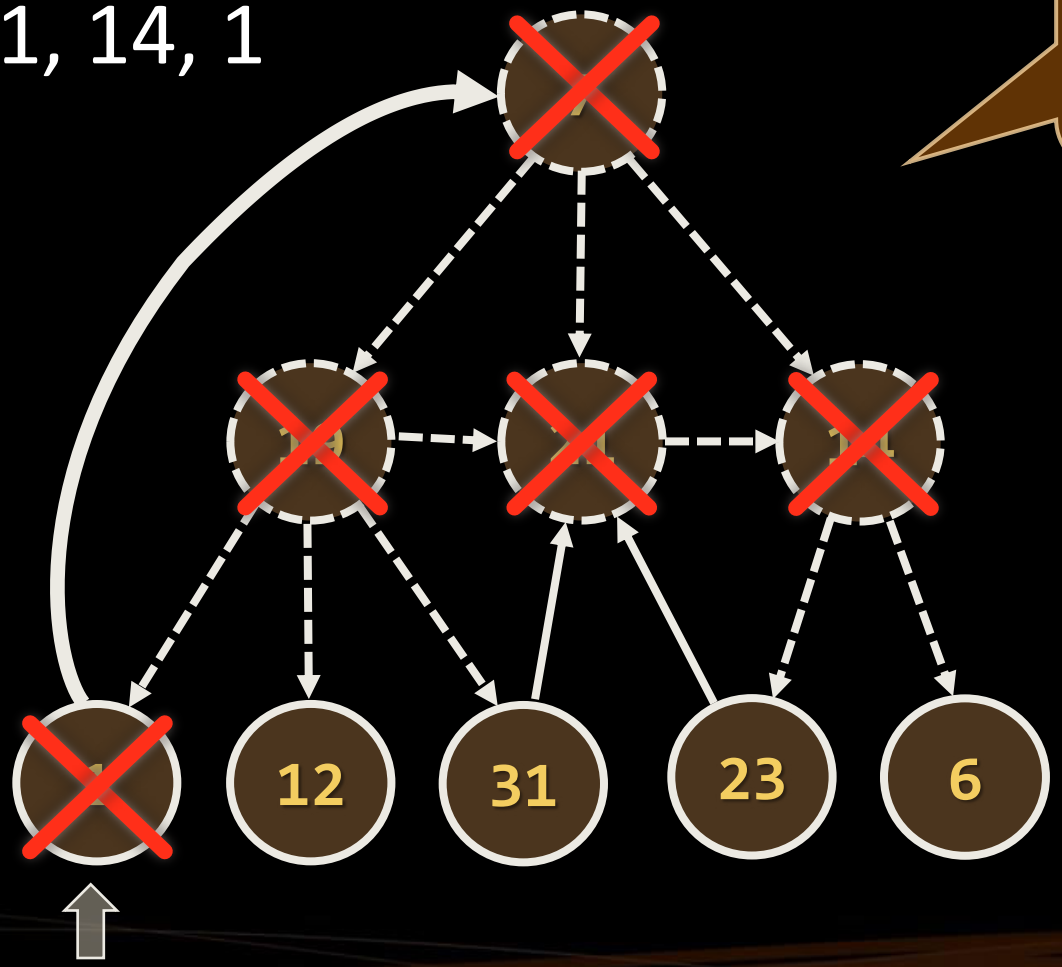


Remove from the queue the next node and print it

Print node 1

BFS in Action (Step 17)

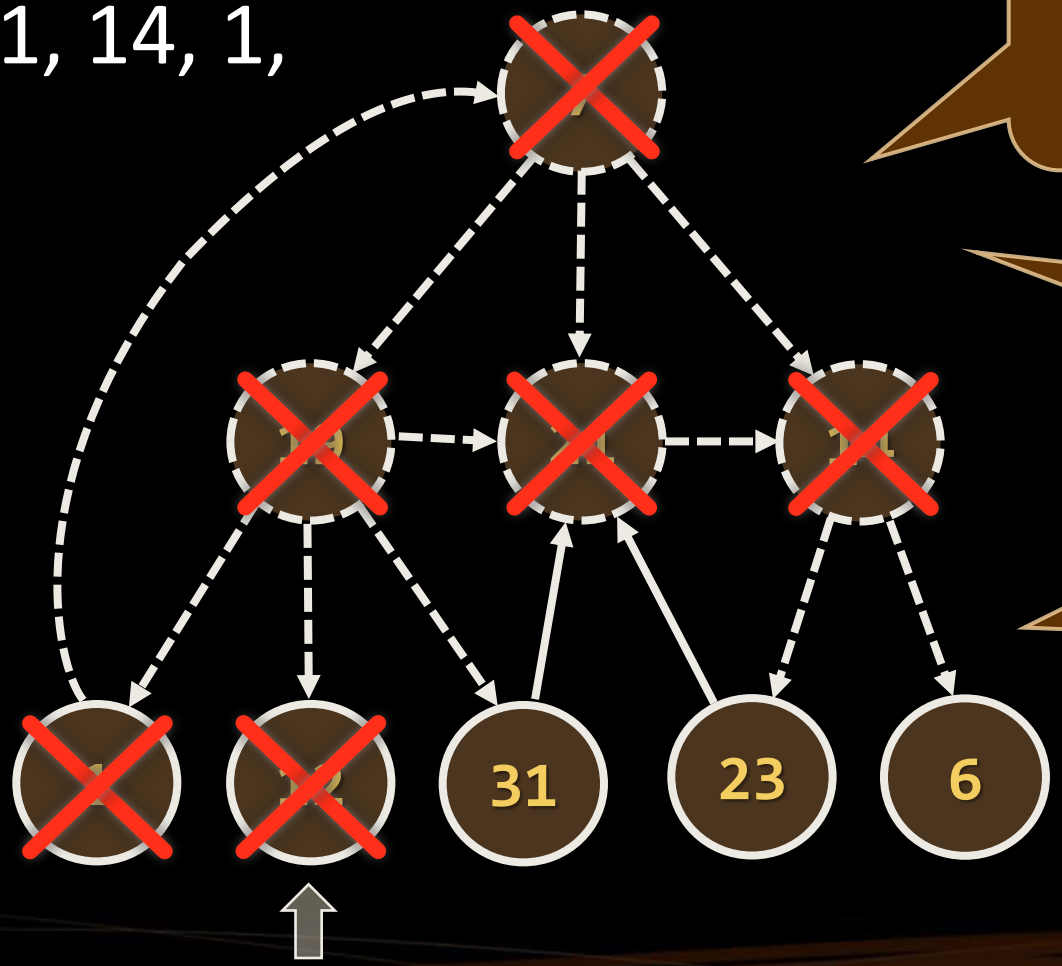
- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, 12, 31, 23, 6
- Output: 7, 19, 21, 14, 1



Child node 7 already visited → skip it

BFS in Action (Step 18)

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



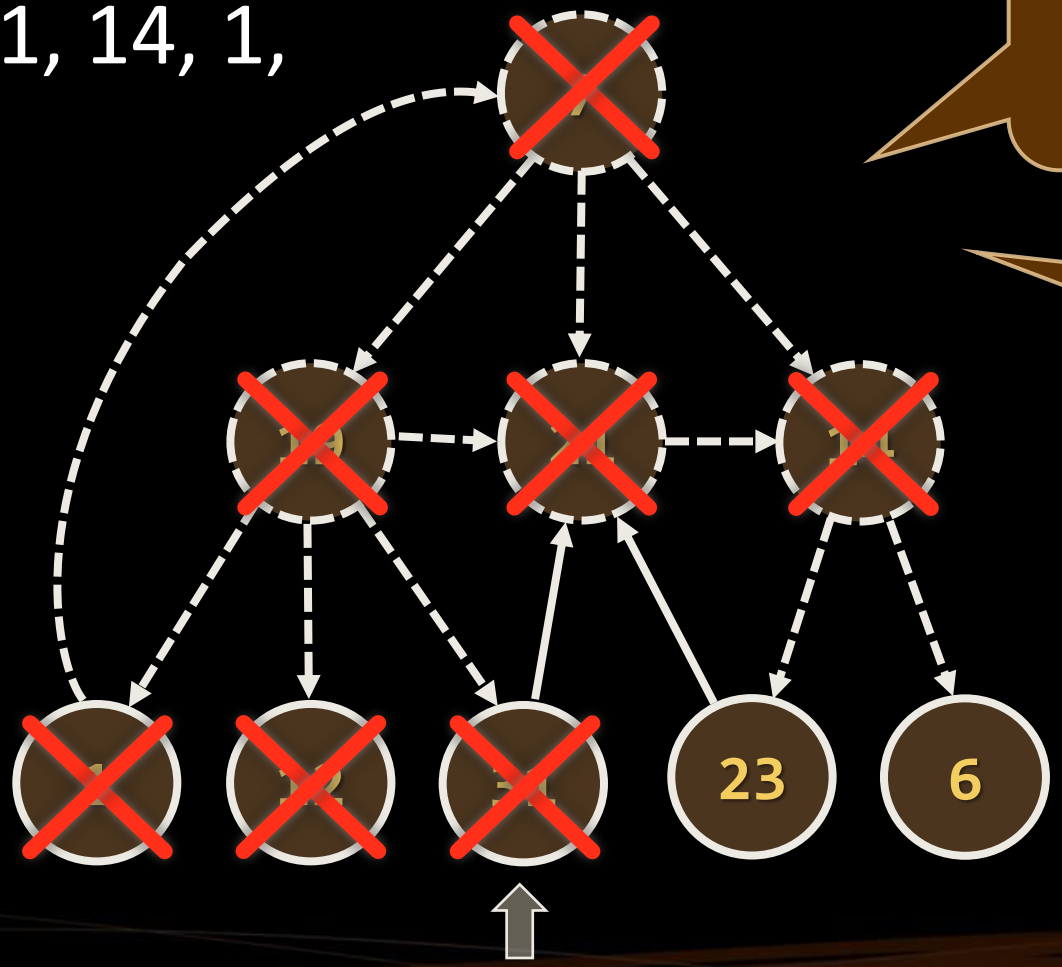
Remove from the queue the next node and print it

Print node 12

No child nodes to enqueue

BFS in Action (Step 19)

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

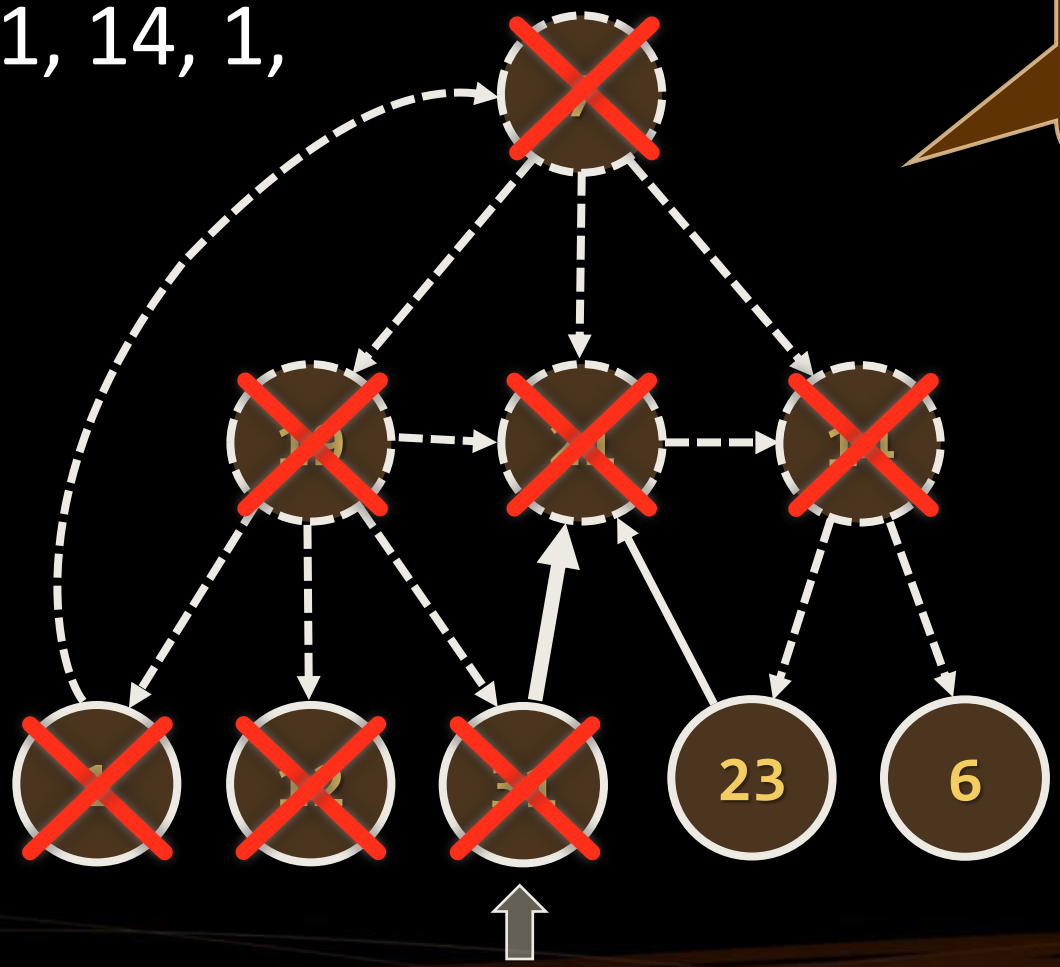


Remove from the queue the next node and print it

Print node 31

BFS in Action (Step 20)

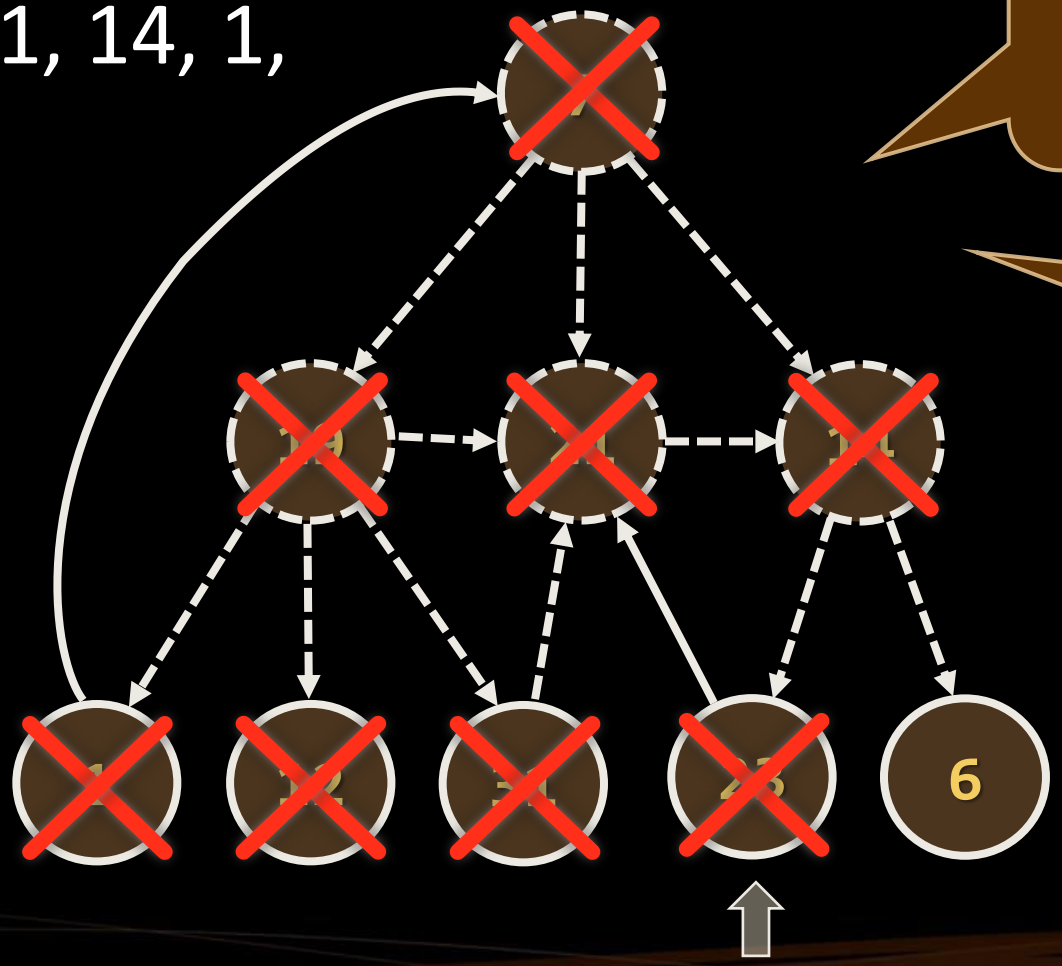
- Queue: ~~7, 19, 21, 14, 1, 12, 31~~, 23, 6
- Output: 7, 19, 21, 14, 1, 12, 31



Child node 21 already visited → skip it

BFS in Action (Step 21)

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

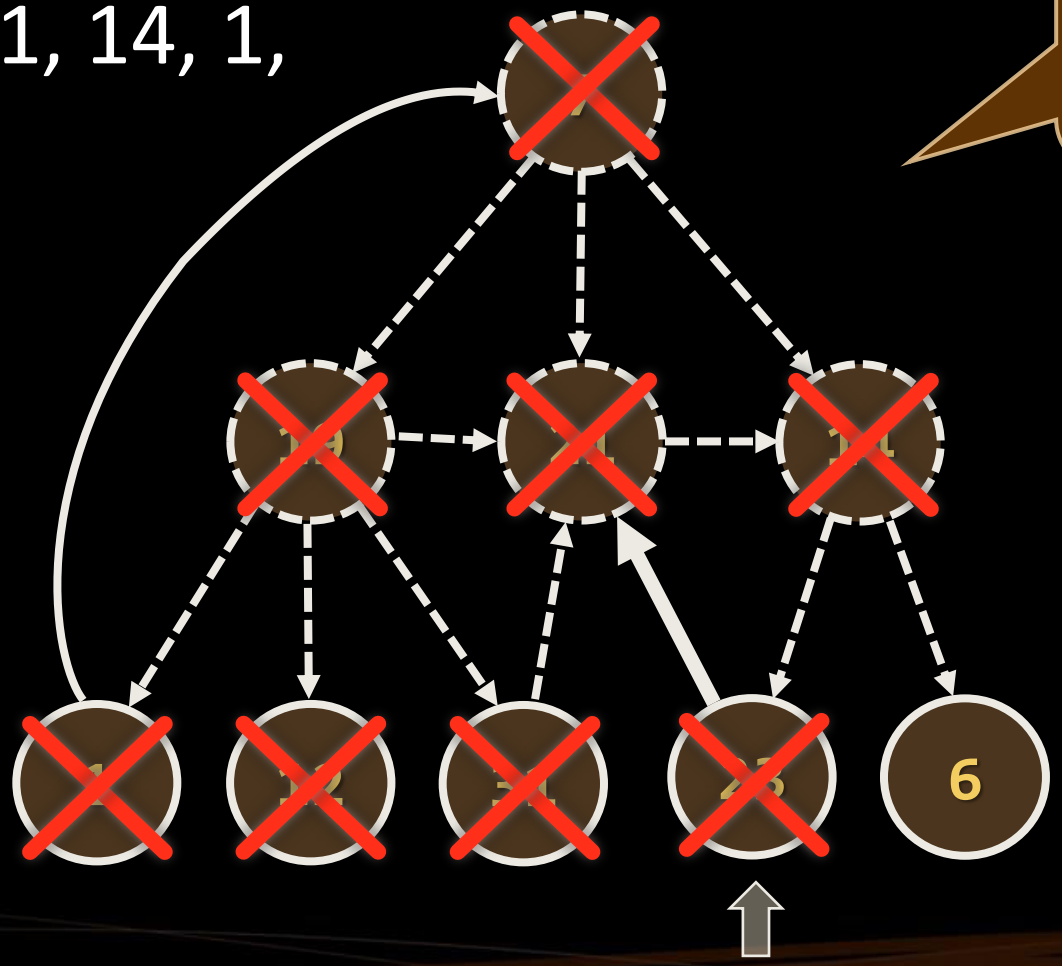


Remove from the queue the next node and print it

Print node 23

BFS in Action (Step 22)

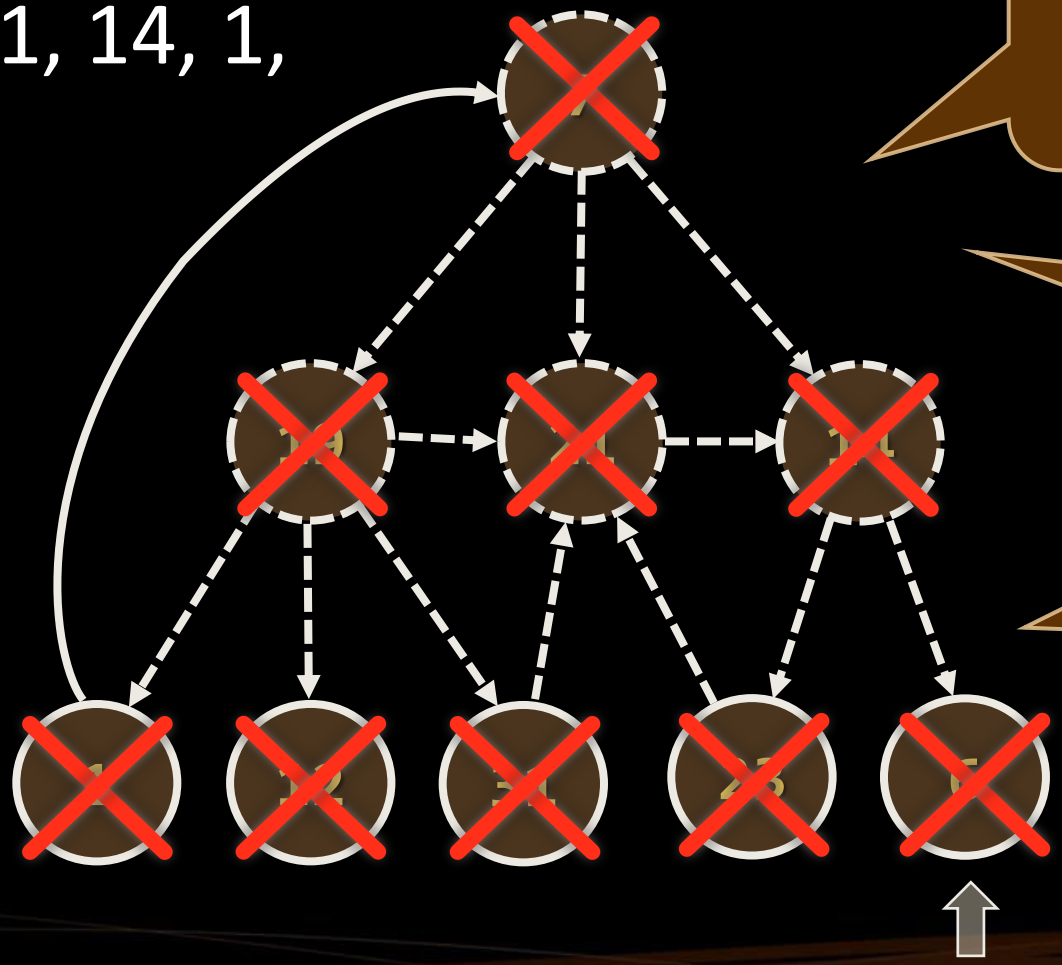
- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, ~~23~~, 6
- Output: 7, 19, 21, 14, 1, 12, 31, 23



Child node 21 already visited → skip it

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



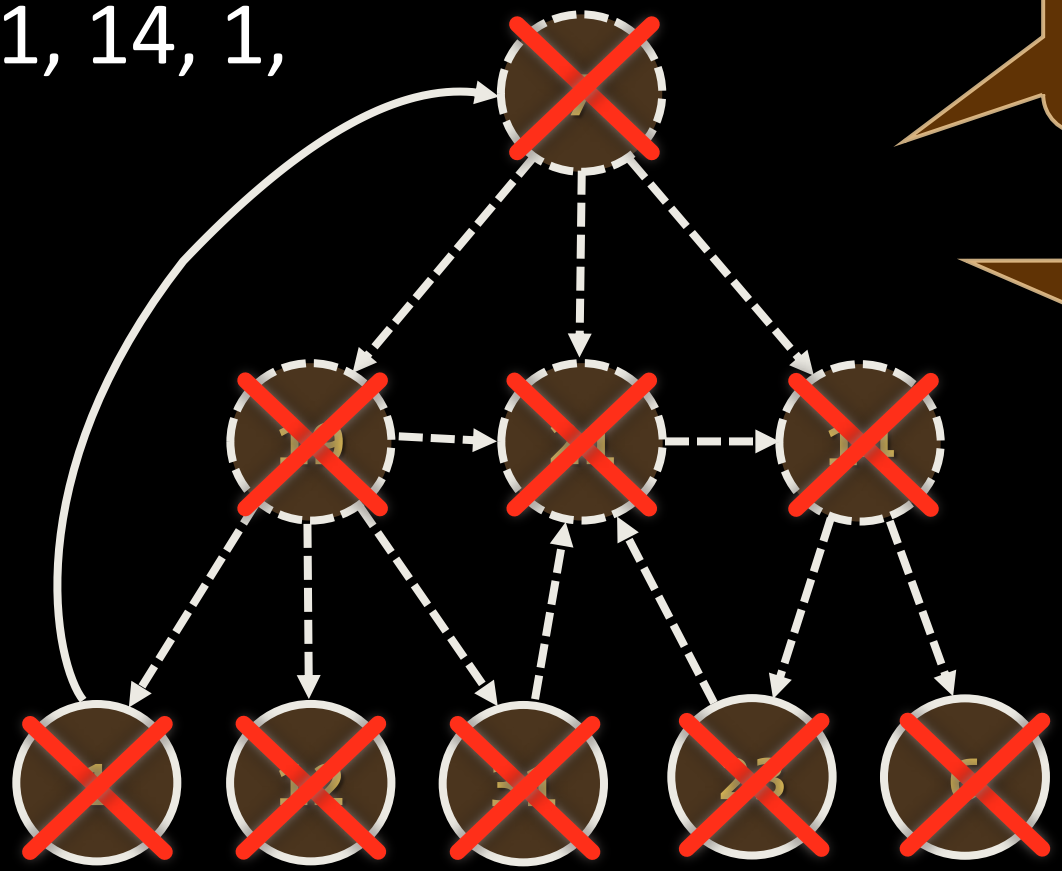
Remove from the queue the next node and print it

Print node 6

No child nodes to enqueue

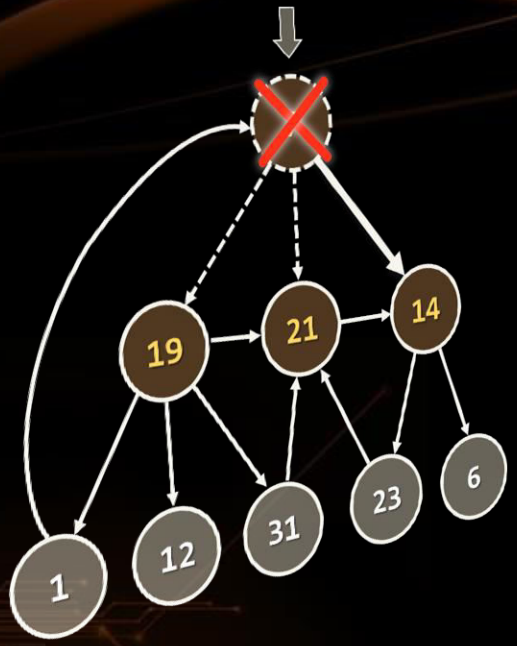
BFS in Action (Step 24)

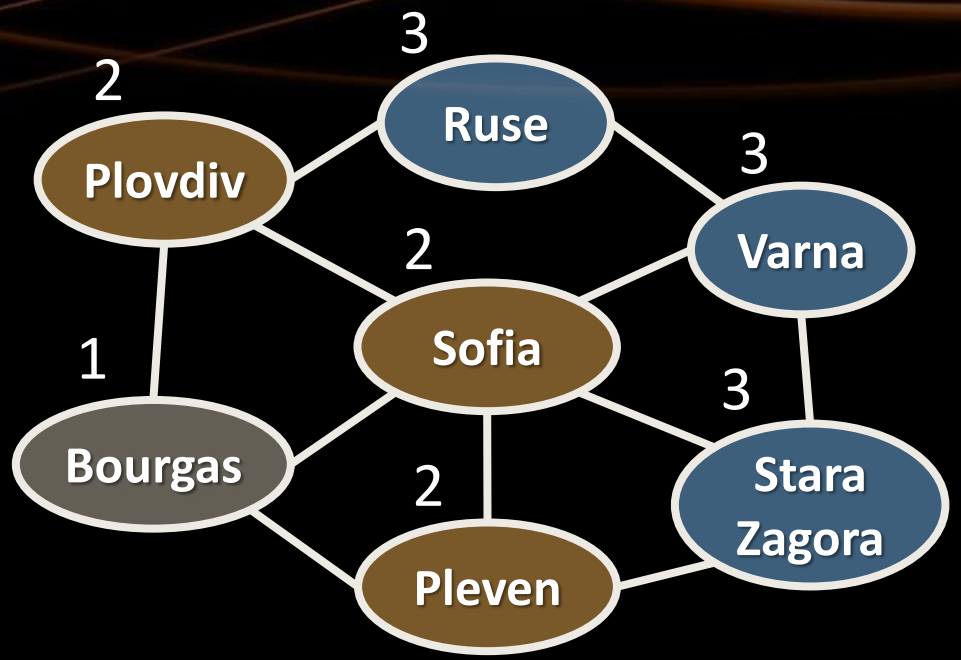
- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, ~~1~~, ~~12~~, ~~31~~, ~~23~~, ~~6~~
- Output: 7, 19, 21, 14, 1, 12, 31, 23, 6

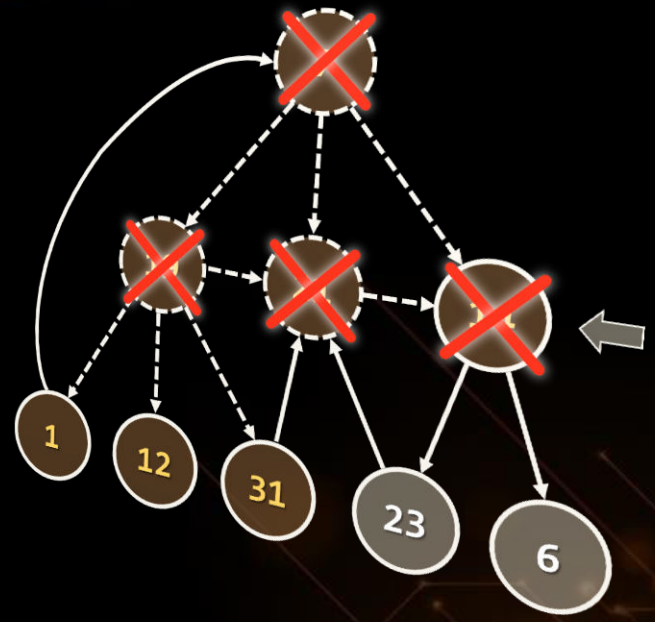


The queue is empty → stop

BFS traversal completed







BFS Graph Traversal

Live Demo

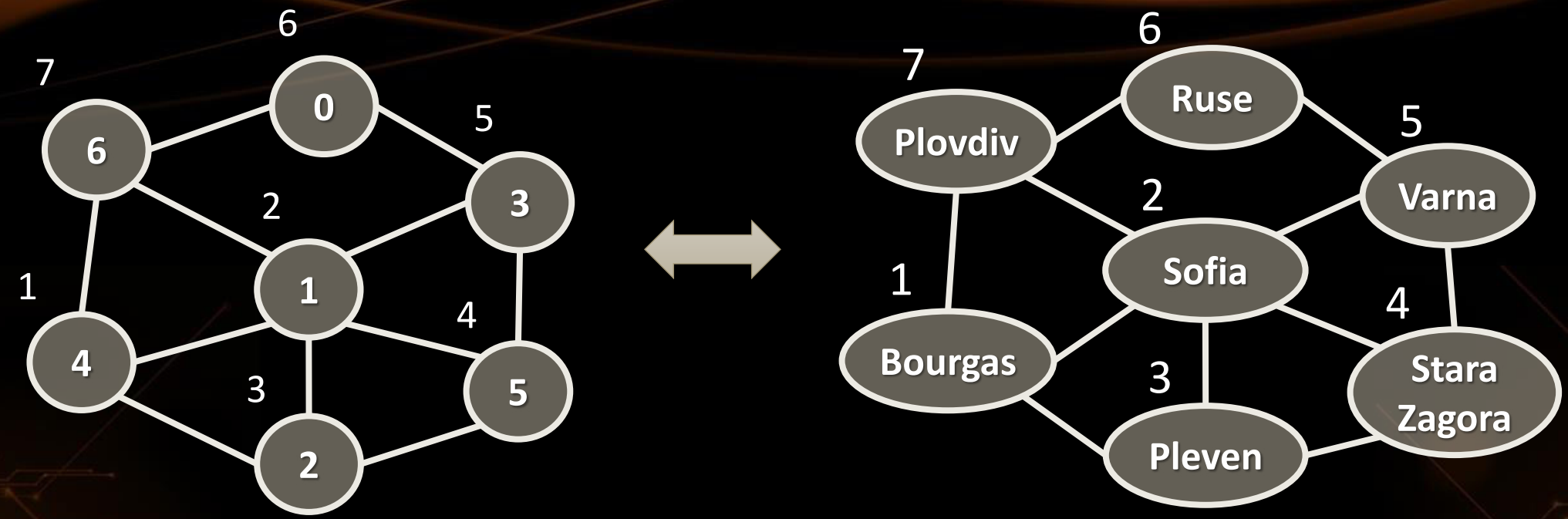
Iterative DFS and BFS

SoftUni Foundation

- 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
}
```

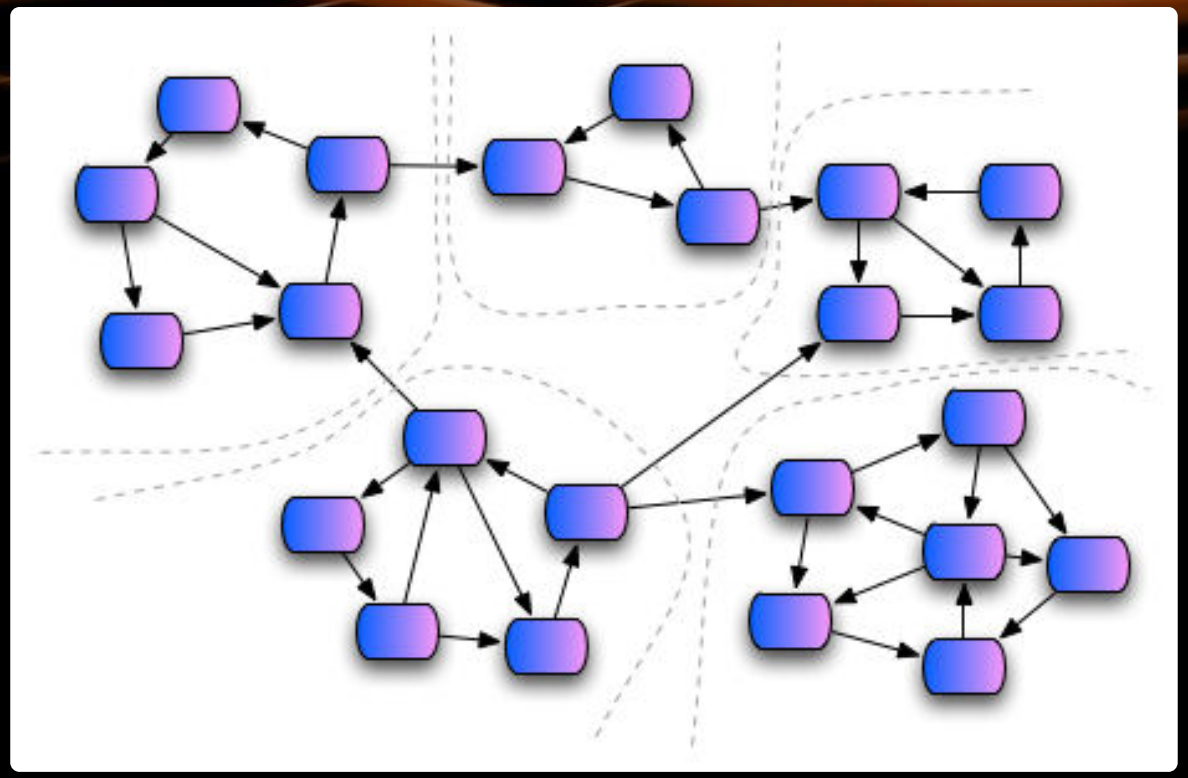
```
DFS (node)
{
  stack ← node
  visited[node] = true
  while stack not empty
    v ← stack
    print v
    for each child c of v
      if not visited[c]
        stack ← c
        visited[c] = true
}
```



Iterative DFS Graph Traversal

Live Demo

SoftUni Foundation

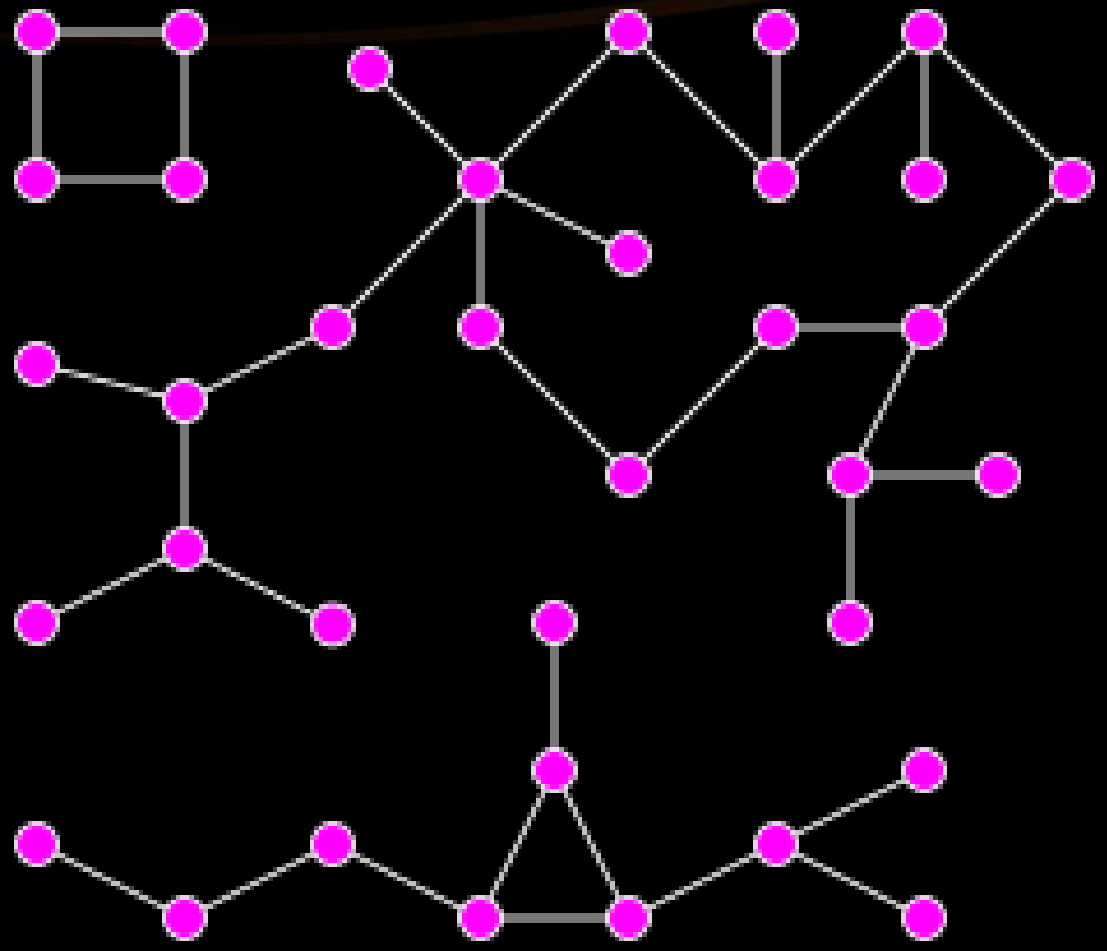


Graph Connectivity

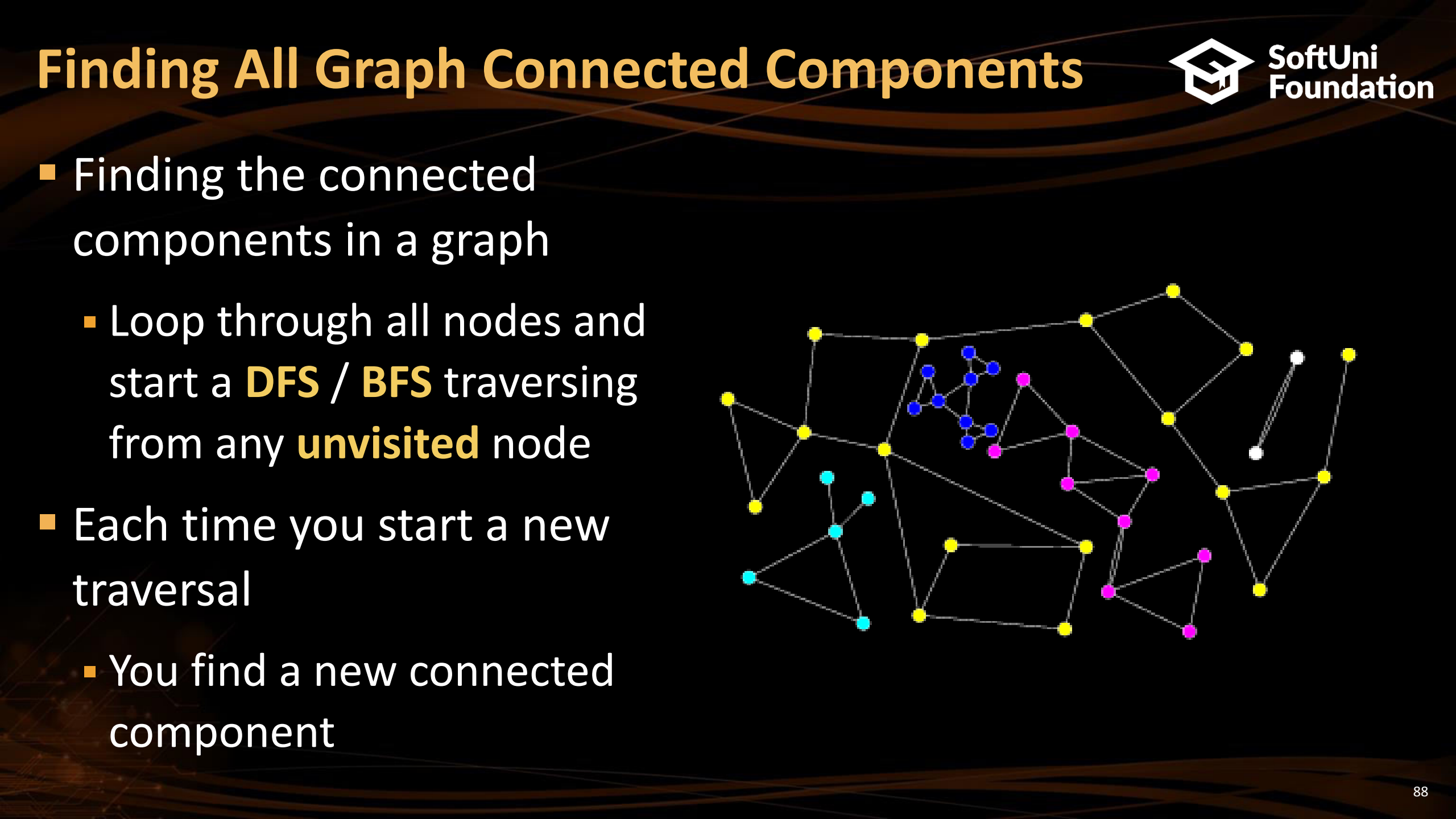
Finding the Connected Components

Graph Connectivity

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



Finding All Graph Connected Components



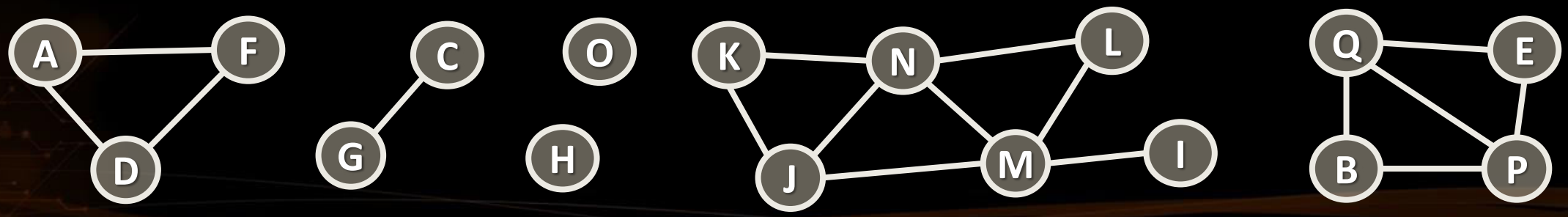
- Finding the connected components in a graph
  - Loop through all nodes and start a **DFS** / **BFS** traversing from any **unvisited** node
- Each time you start a new traversal
  - You find a new connected component

Graph Connected Components: Algorithm

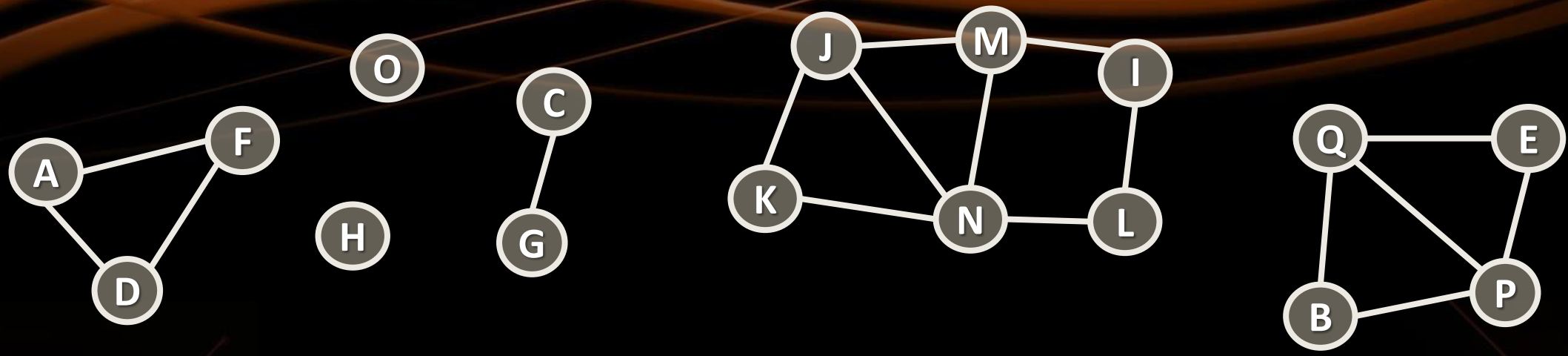
SoftUni Foundation

```
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);
  }
}
```



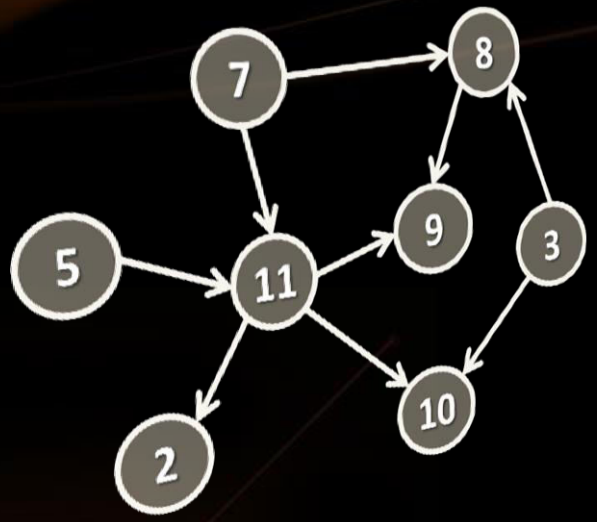
SoftUni Foundation

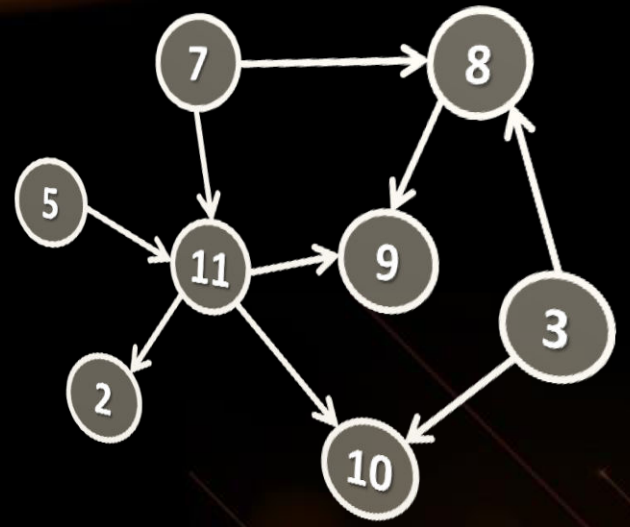


Finding the Connected Components in an Undirected Graph

Live Demo

SoftUni Foundation



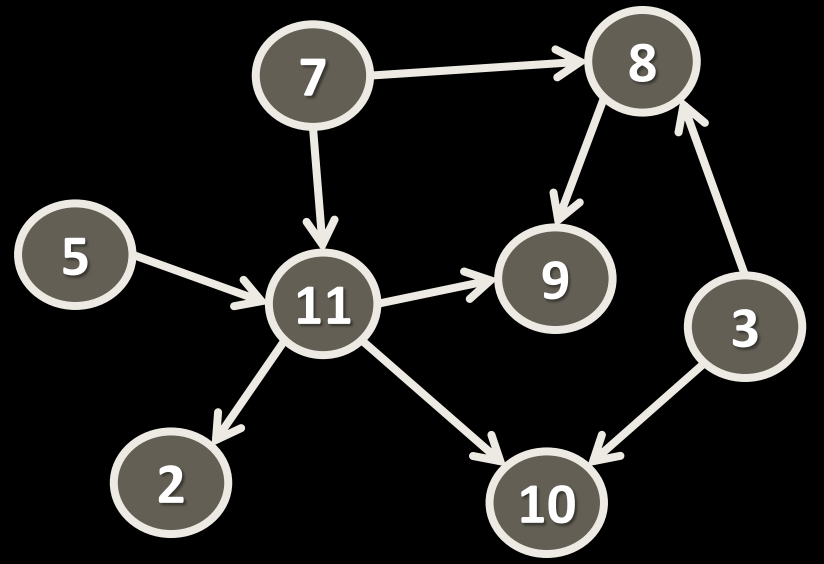


Topological Sorting

Ordering a Graph by Set of Dependencies

Topological Sorting

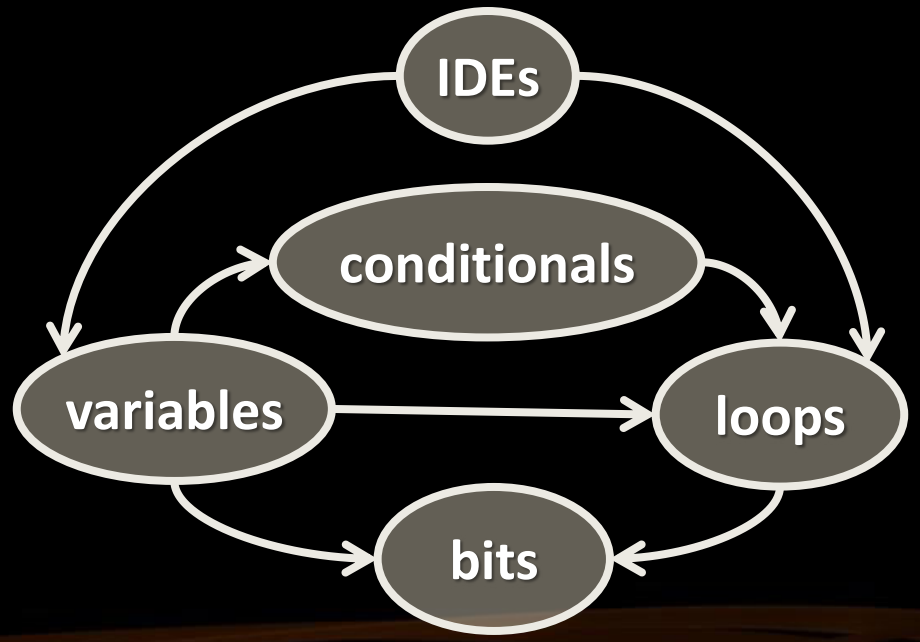
- **Topological sorting** (ordering) of a directed graph
  - Linear ordering of its vertices, such that
  - For every directed edge from vertex ***u*** to vertex ***v***, ***u*** comes before ***v*** in the ordering
- Example:
  - 7 → 5 → 3 → 11 → 8 → 2 → 9 → 10
  - 3 → 5 → 7 → 8 → 11 → 2 → 9 → 10
  - 5 → 7 → 3 → 8 → 11 → 10 → 9 → 2



Topological Sorting – Example

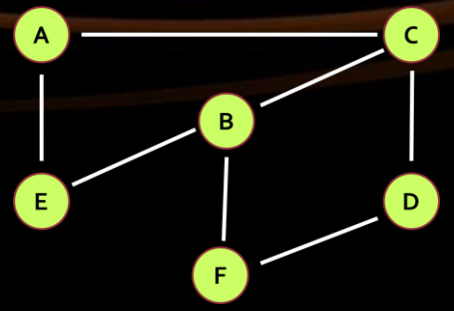
SoftUni Foundation

- 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**

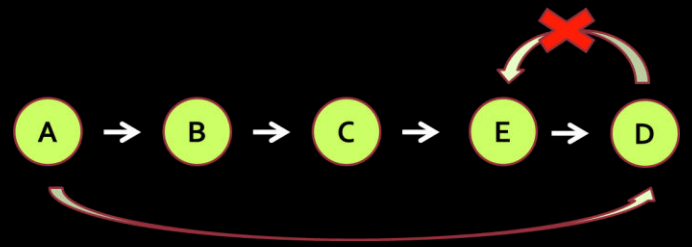


Topological Sorting – Rules

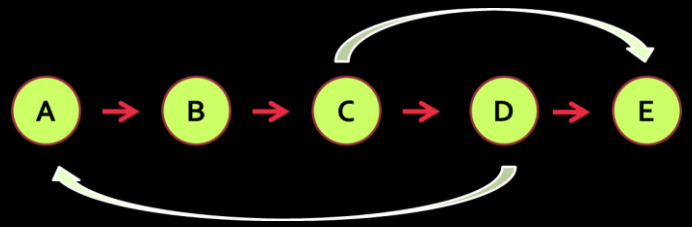
- Rules
  - Undirected graphs cannot be sorted



  - Graphs with cycles cannot be sorted



  - Sorting is not unique



  - Various sorting algorithms exists and they give different results

Topological Sorting: Source Removal Algorithm

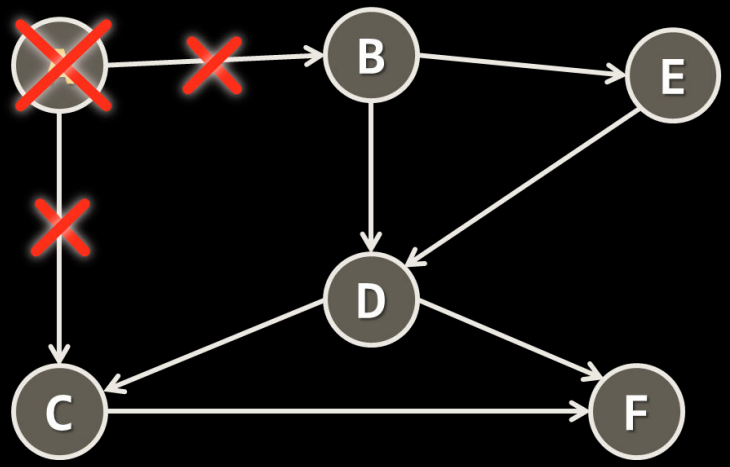
SoftUni Foundation

- **Source removal top-sort algorithm**:

1. Create an empty list

Repeat until the graph is empty:

2. Find a node without incoming edges
3. Print this node
4. Remove the edge from the graph

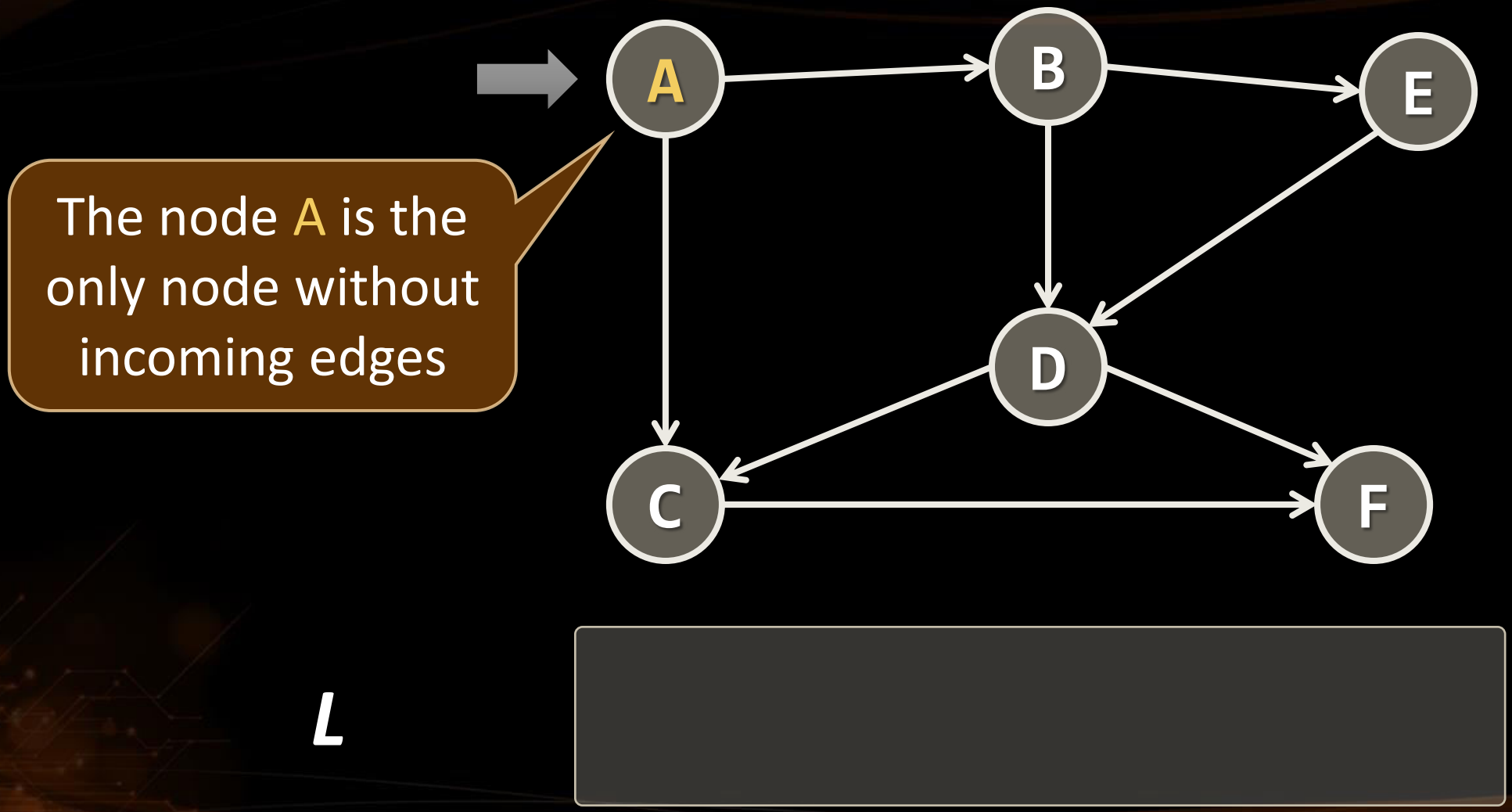


Source Removal Algorithm

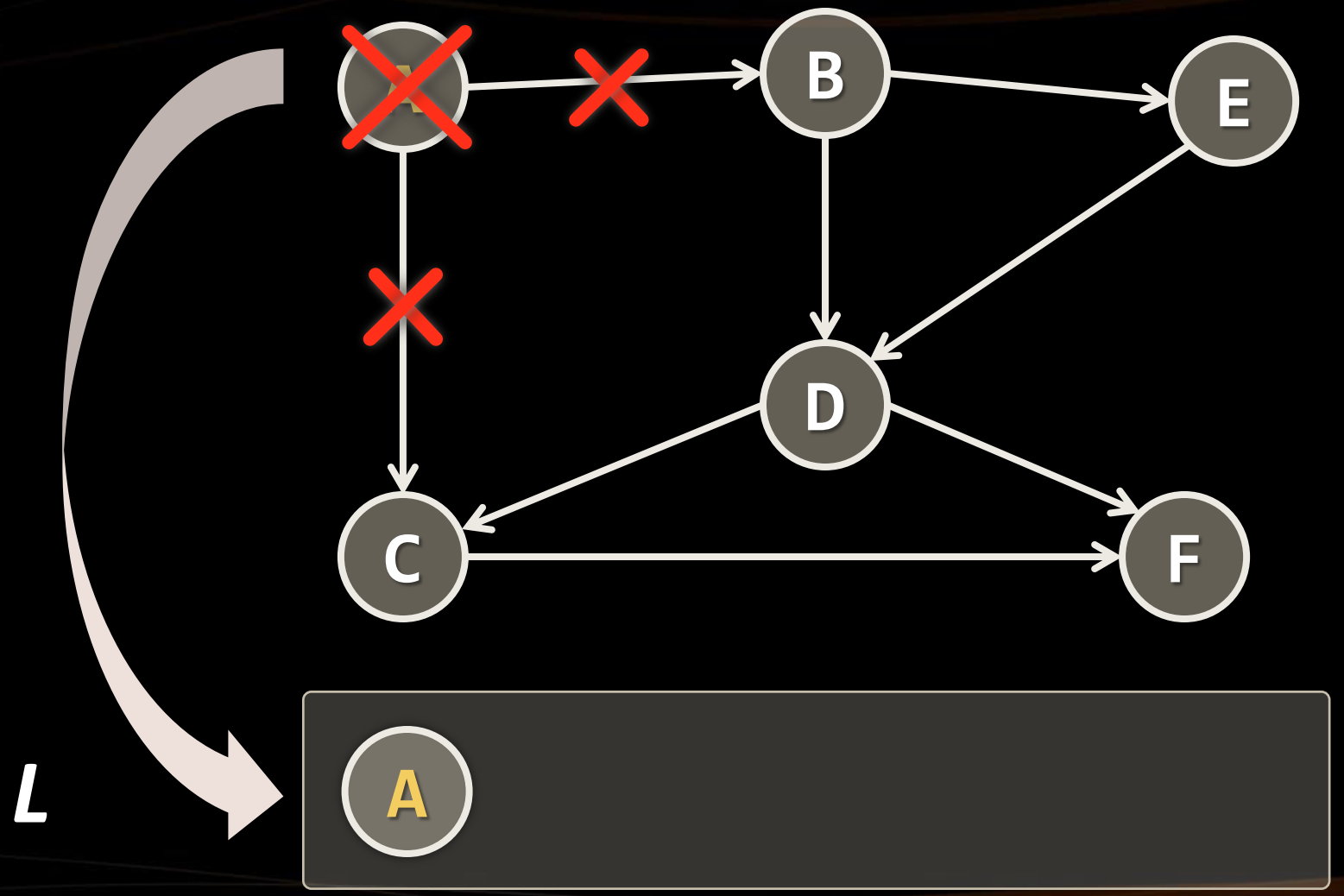
SoftUni Foundation

```
L ← empty list that will hold the sorted elements (output)
S ← 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 → 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"
```

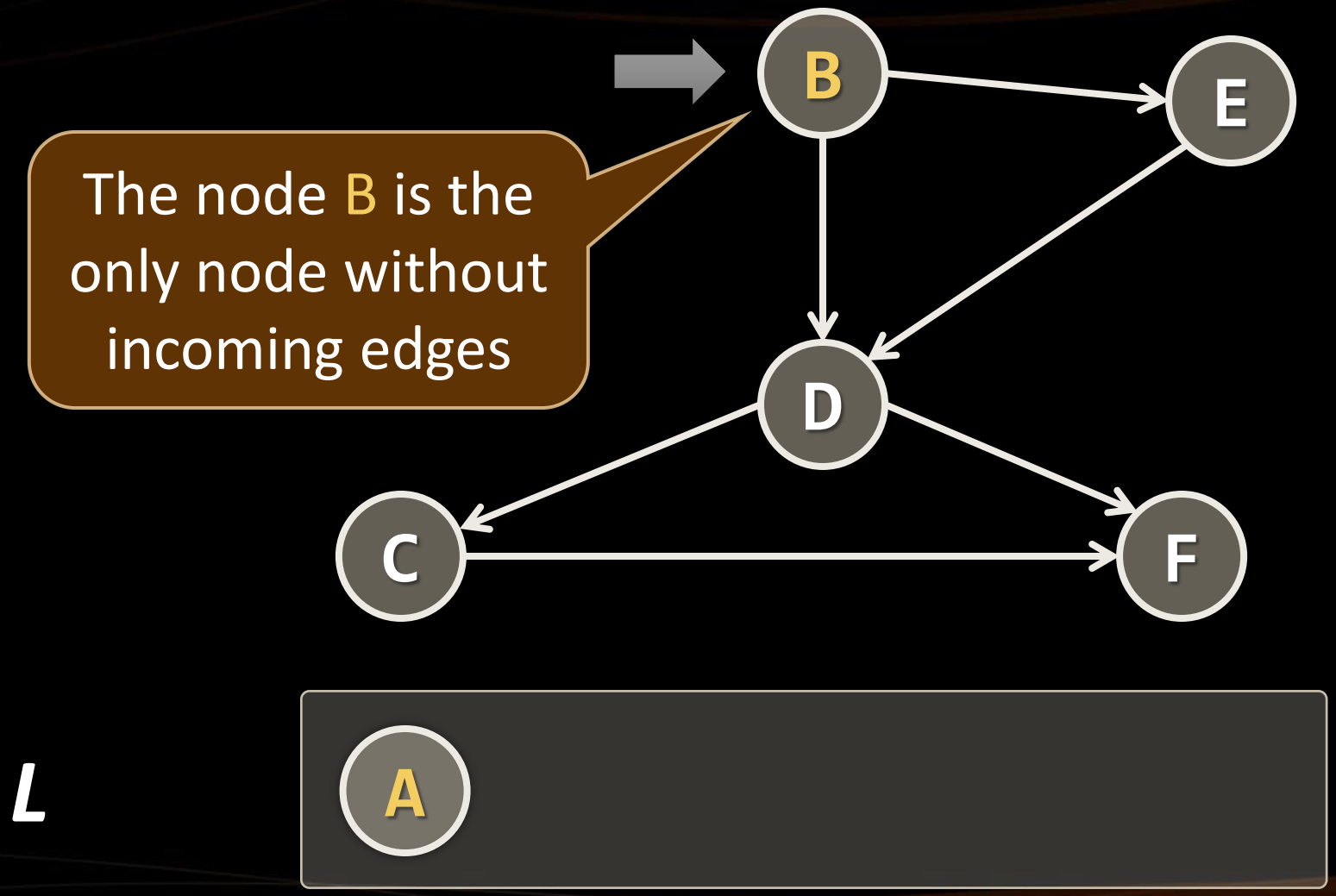
Step #1: Find a Node with No Incoming Edges



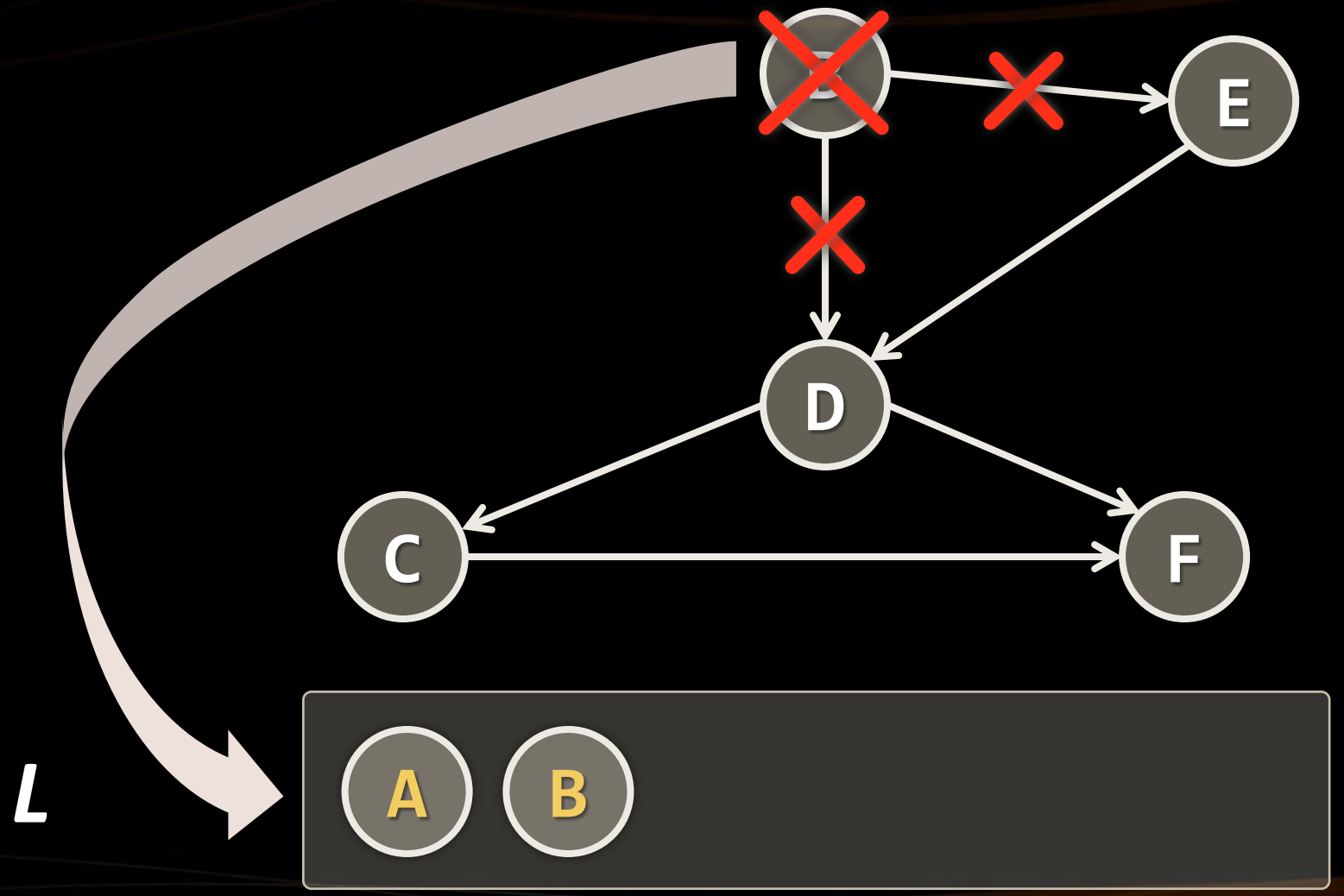
Step #2: Remove Node A with Its Edges



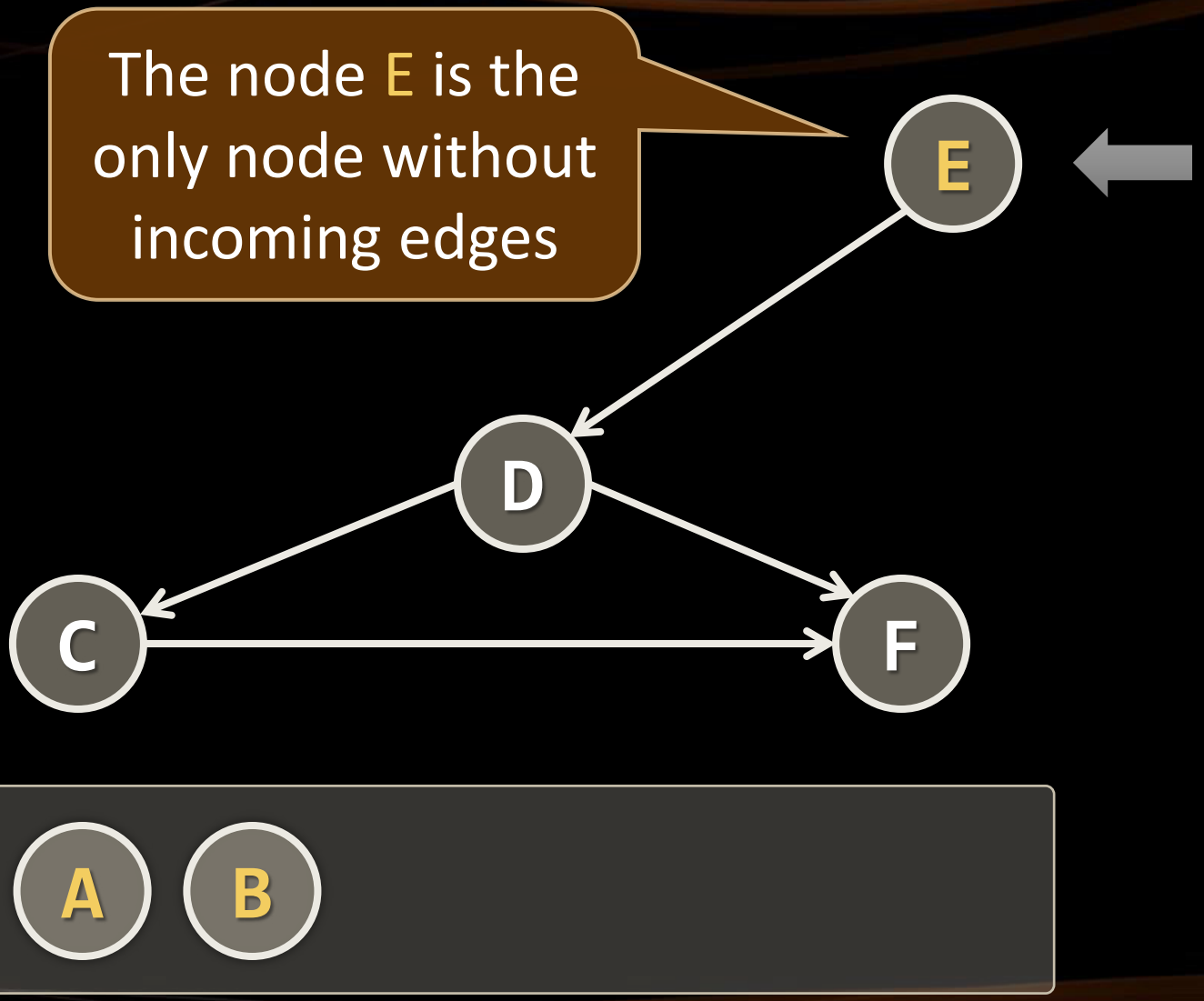
Step #3: Find a Node with No Incoming Edges



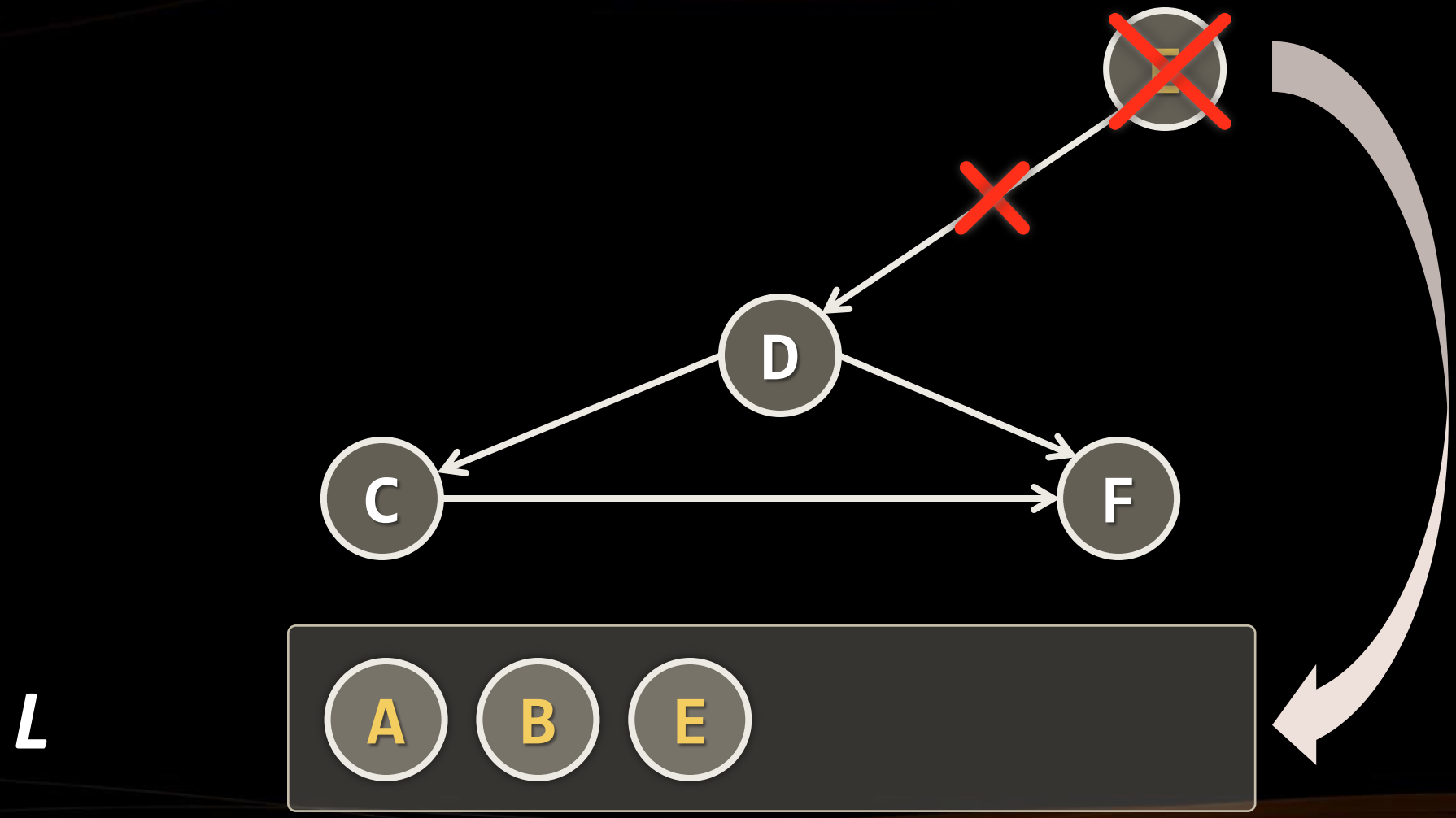
Step #4: Remove Node B with Its Edges



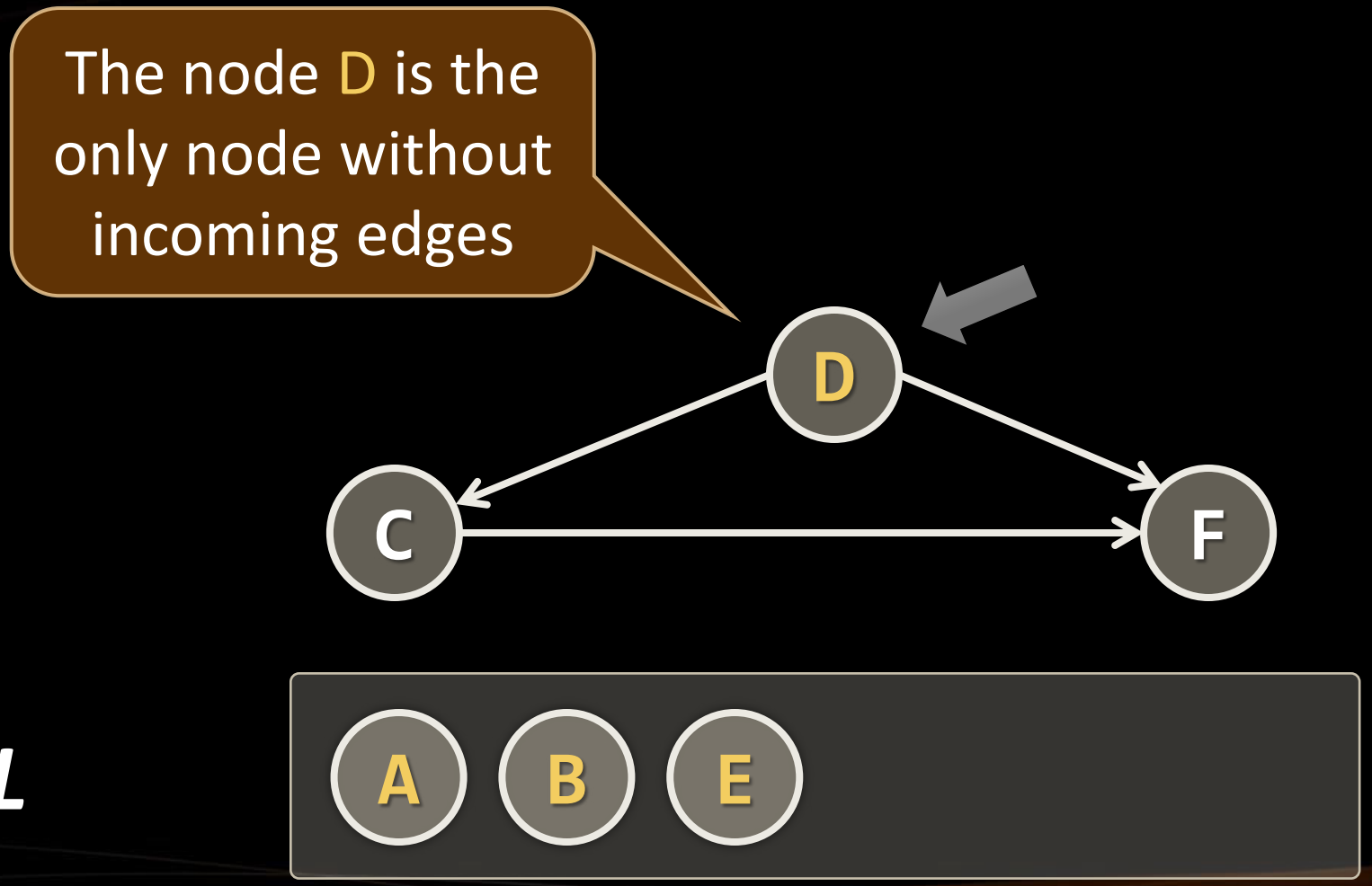
Step #5: Find a Node with No Incoming Edges



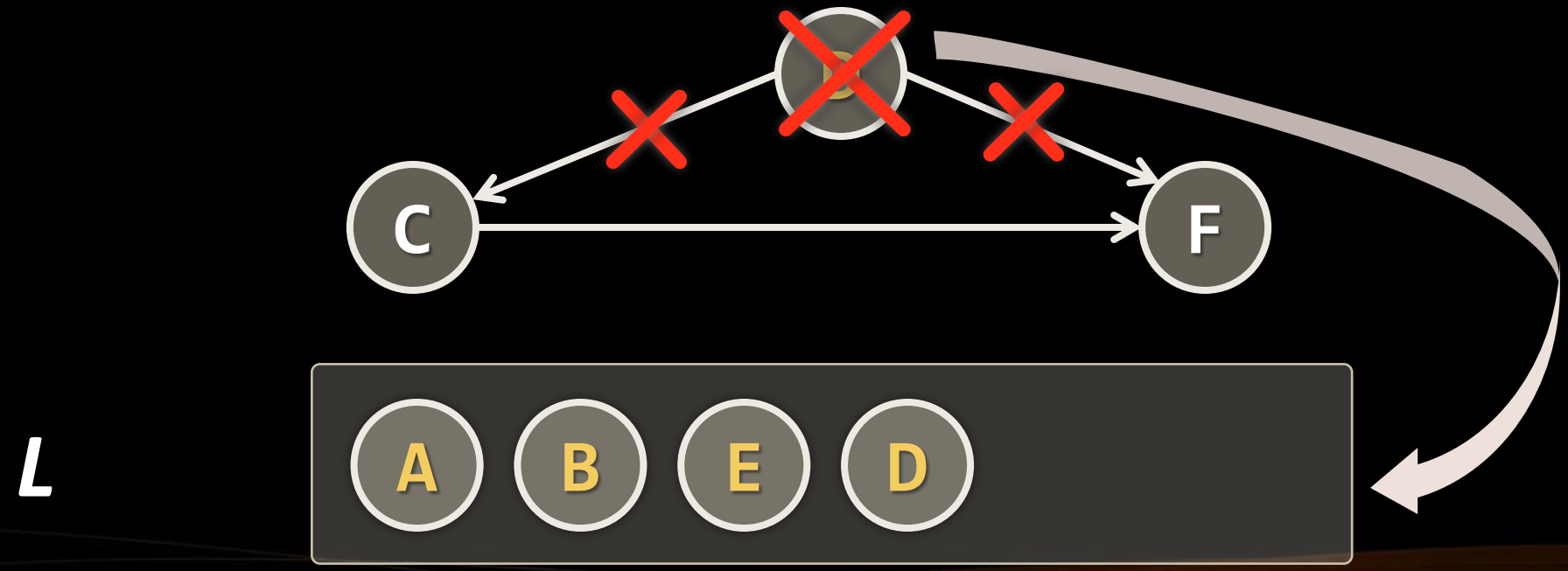
Step #6: Remove Node E with Its Edges



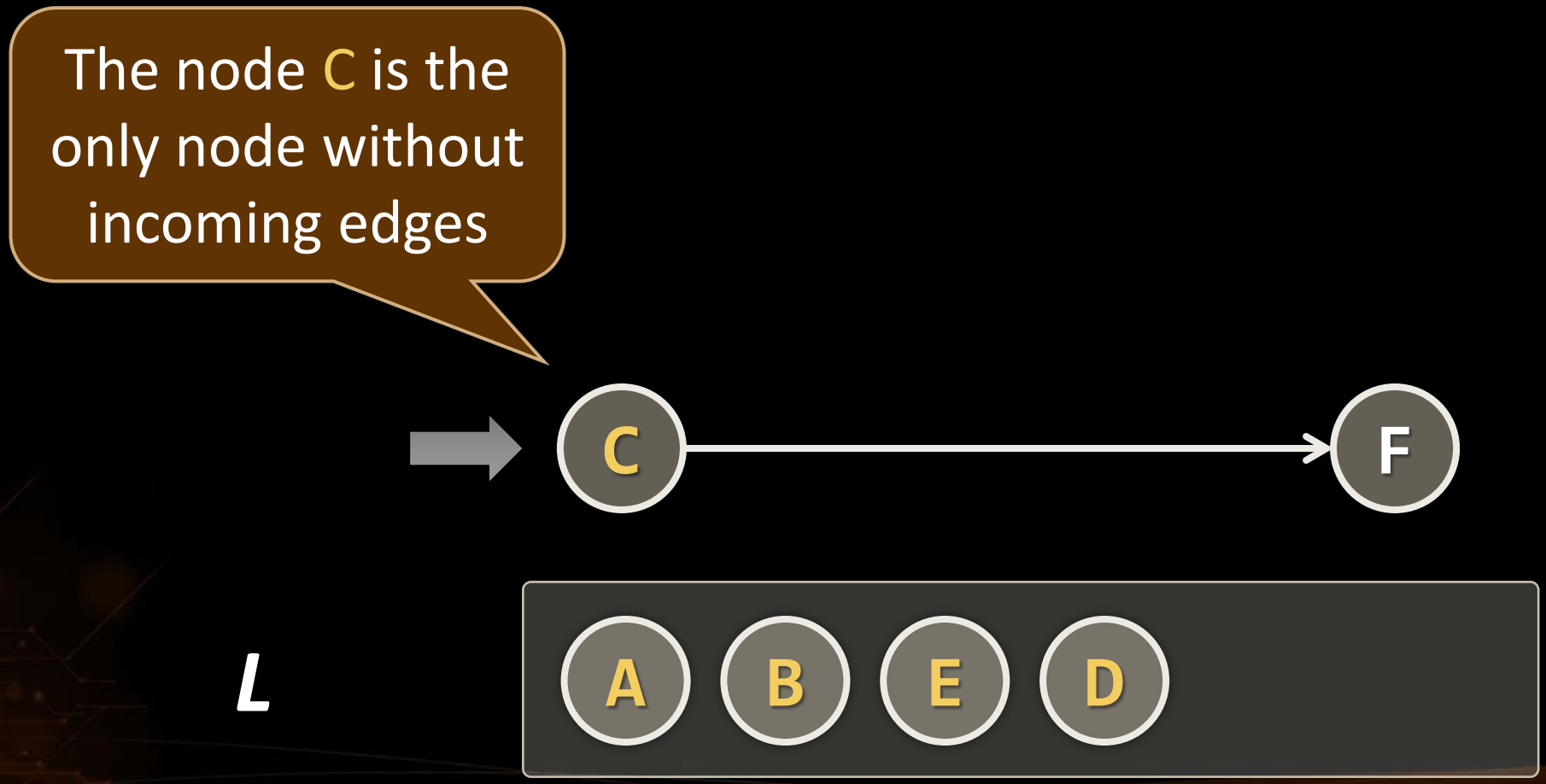
Step #7: Find a Node with No Incoming Edges



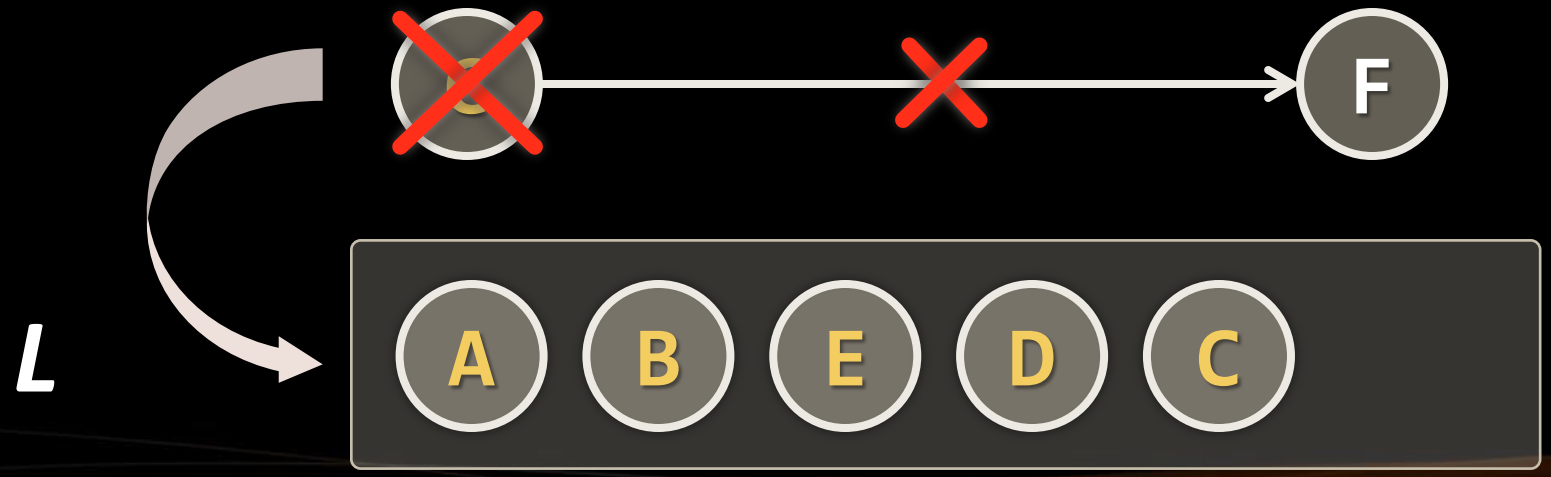
Step #8: Remove Node D with Its Edges



Step #9: Find a Node with No Incoming Edges



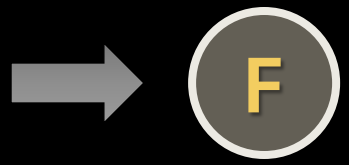
Step #10: Remove Node C with Its Edges



Step #11: Find a Node with No Incoming Edges

SoftUni Foundation

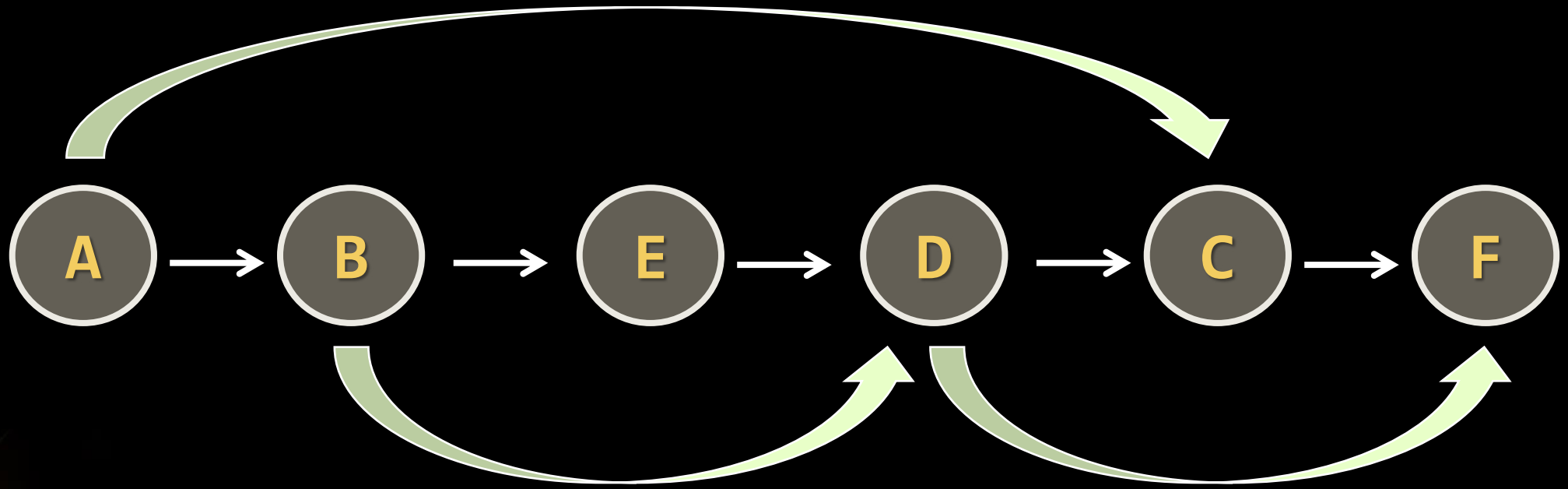
The node F is the only node without incoming edges



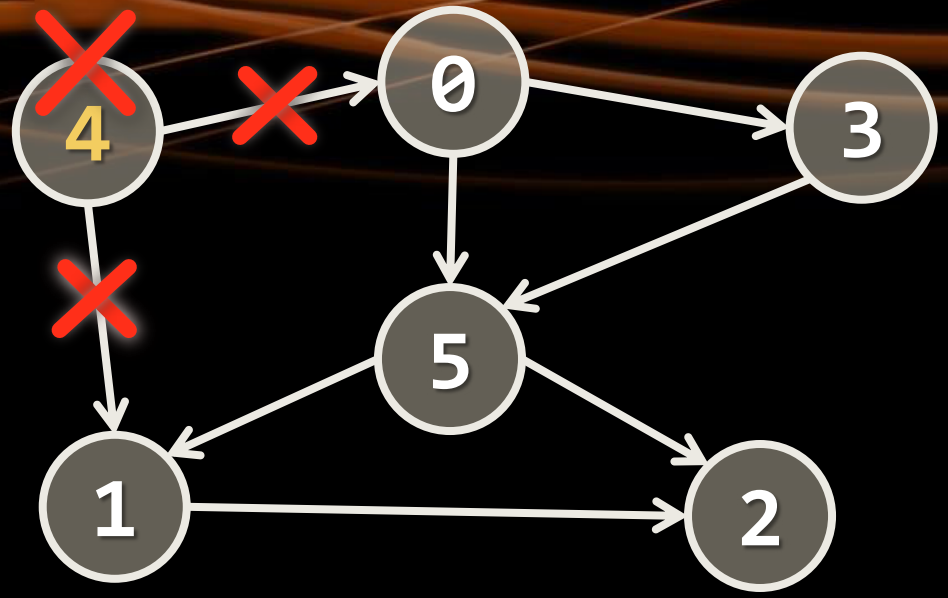
L

Step #12: Remove Node F with Its Edges

Result: Topological Sorting



SoftUni Foundation



Topological Sorting: Source Removal Algorithm

Live Demo

Topological Sorting: DFS Algorithm

SoftUni Foundation

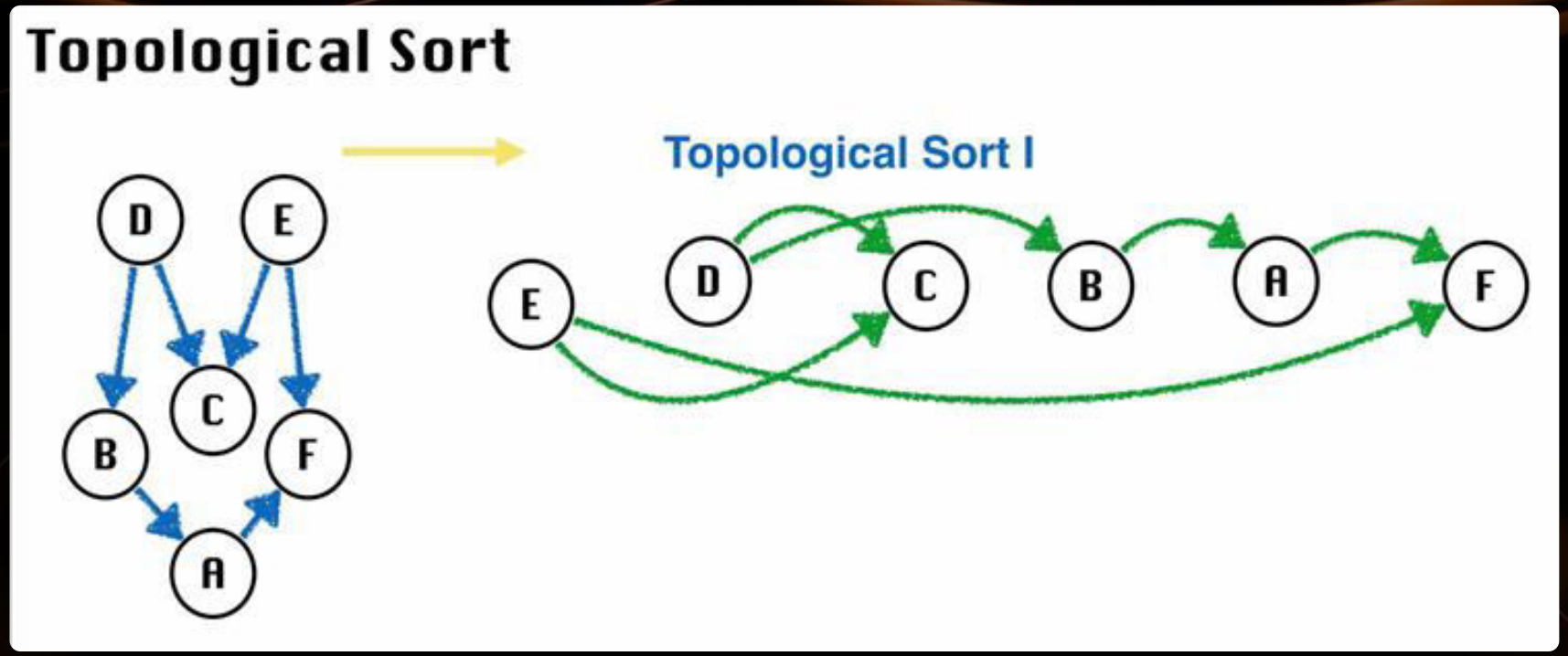
```
sortedNodes = { }  // linked list to hold the result
visitedNodes = { }  // set of already visited nodes
foreach node in graphNodes
    TopSortDFS(node)

TopSortDFS(node)
    if node ∉ visitedNodes
        visitedNodes ← node
        for each child c of node
            TopSortDFS(c)
        insert node upfront in the sortedNodes
```

Visualization: https://www.cs.usfca.edu/~galles/visualization/TopoSortDFS.html

TopSort: DFS Algorithm + Cycle Detection

SoftUni Foundation

```
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
```



Topological Sorting

Live Coding (Lab)

Summary

- 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** – source removal algorithms and DFS algorithm

Graphs and Graph Algorithms

SoftUni Foundation

Questions?

https://softuni.bg/opencourses/algorithms

License

SoftUni Foundation

- This course (slides, examples, labs, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

- Attribution: this work may contain portions from
  - "Fundamentals of Computer Programming with C#" book by Svetlin Nakov & Co. under CC-BY-SA license
  - "Data Structures and Algorithms" course by Telerik Academy under CC-BY-NC-SA license

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
- Software University Foundation
  - http://softuni.foundation/
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

Software University

SoftUni Foundation

FORUM