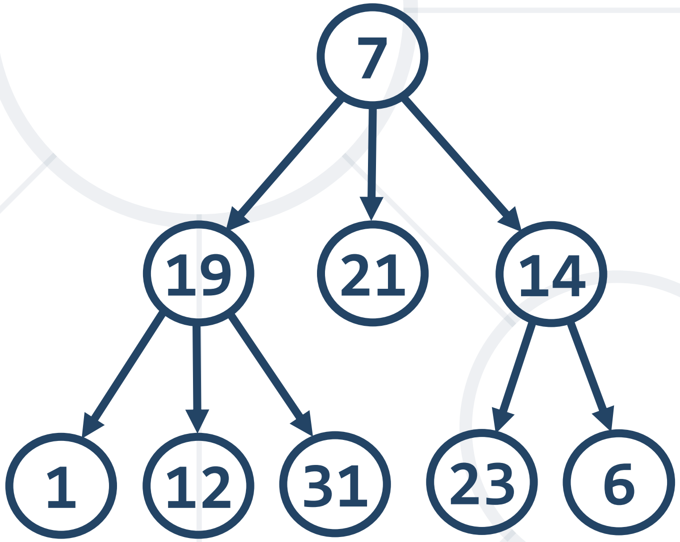
Trees and Graphs

Trees and Graphs Fundamentals, Terminology and Traversal Algorithms



SoftUni

SoftUni Team

Technical Trainers

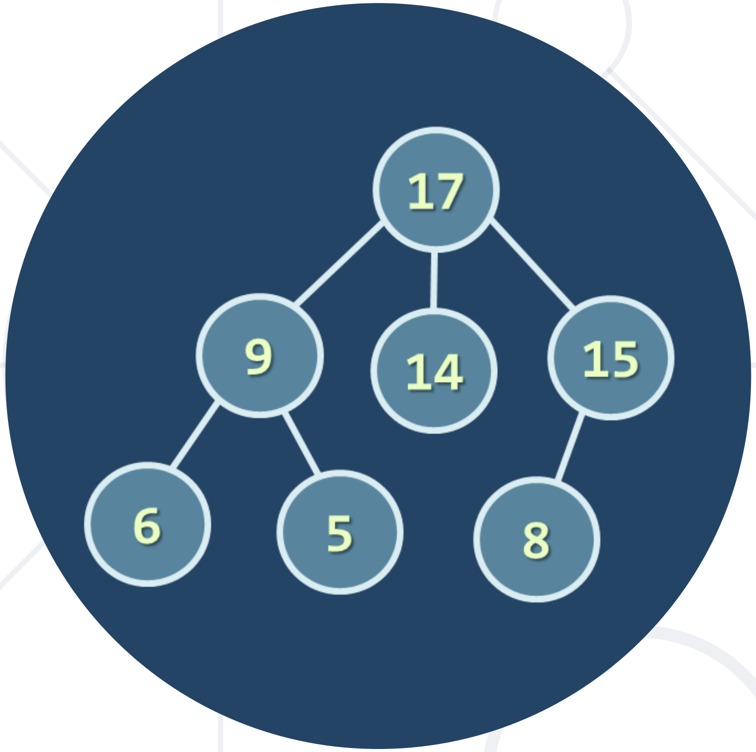
Software University

Software University

https://softuni.bg

Table of Contents

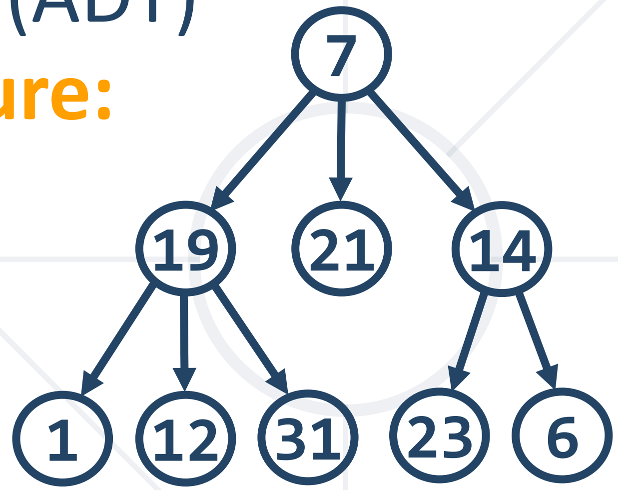
Software University



Trees

Tree Definition

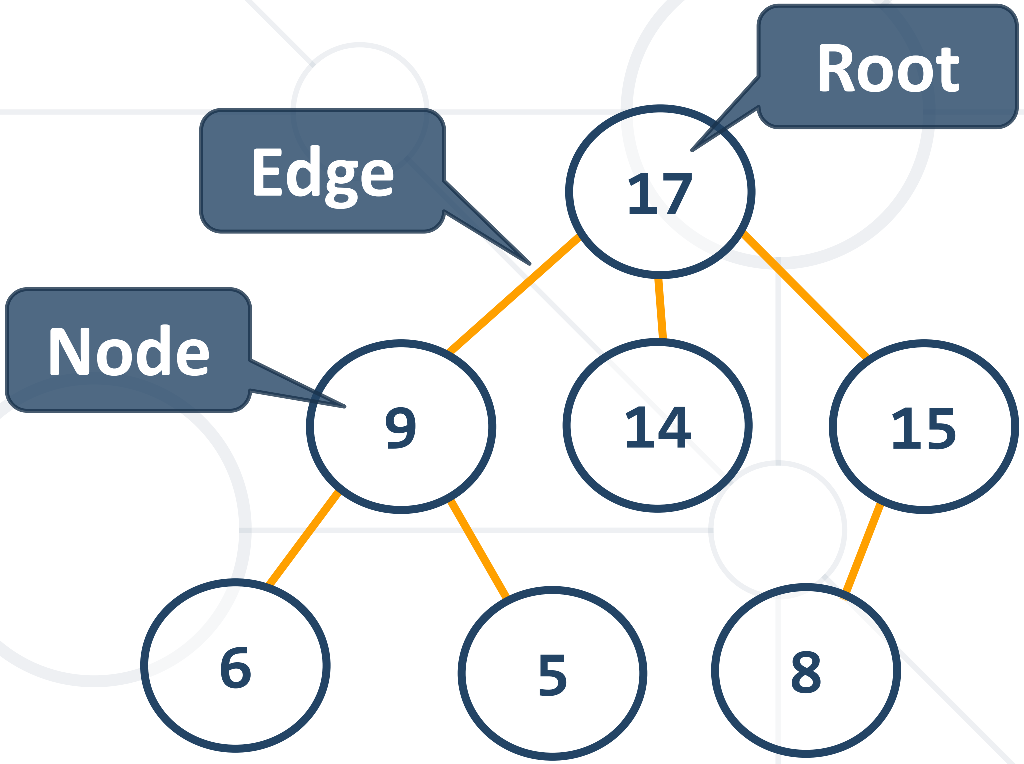
- Tree is a widely used **abstract data type** (ADT) that simulates a hierarchical **tree structure:**
  - **Value**
  - **Parent** – null or another tree reference
  - **Children** – collection of trees



- **Recursive definition** – a tree consists of a value and set of child nodes, which are trees
- By working with trees you can **actually work with:**
  - **Hierarchical** structures, **markup** languages, **DFS** and **BFS** algorithms

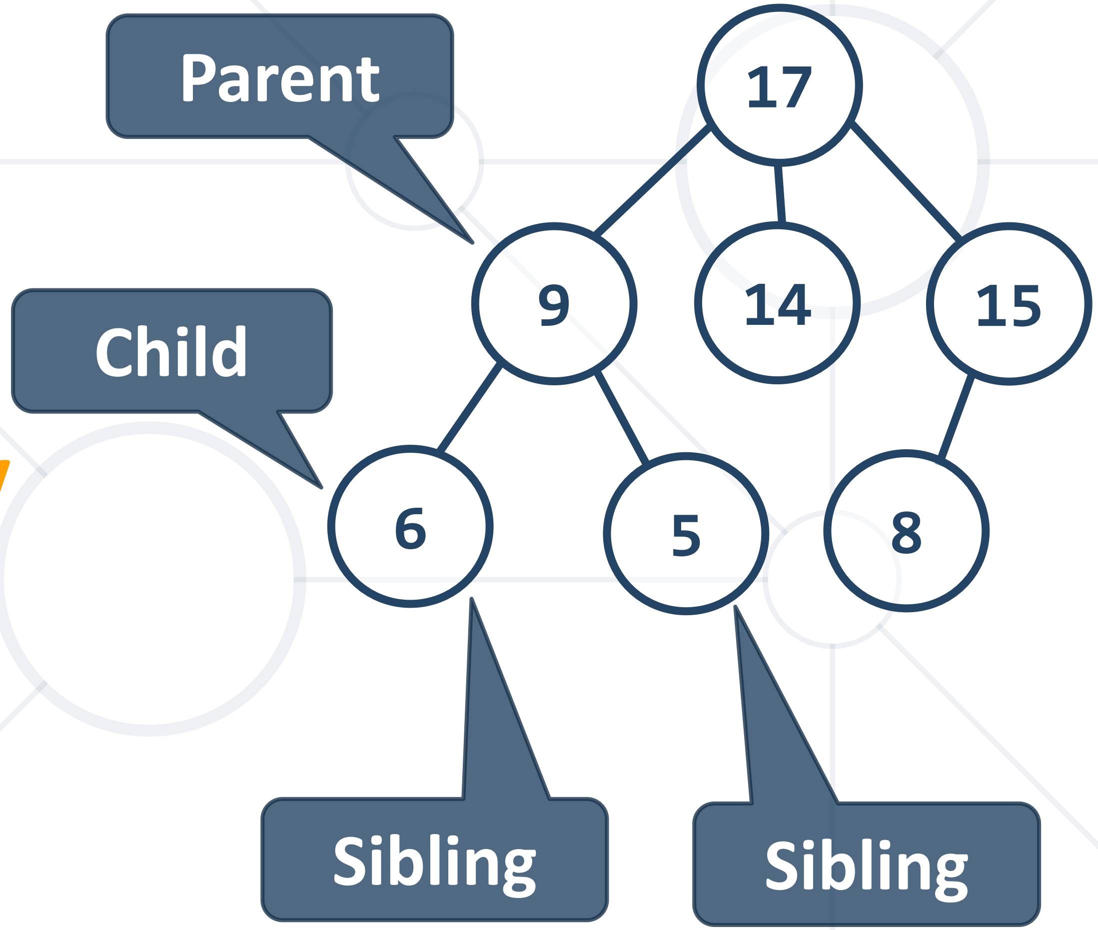
Tree Data Structure – Terminology

- **Node** – a structure which may contain a **value** or **condition** or represent a separate **data structure**
- **Edge** – the **connection between** one **node** and **another**
- **Root** – the **top** node in a **tree**, the **prime ancestor**



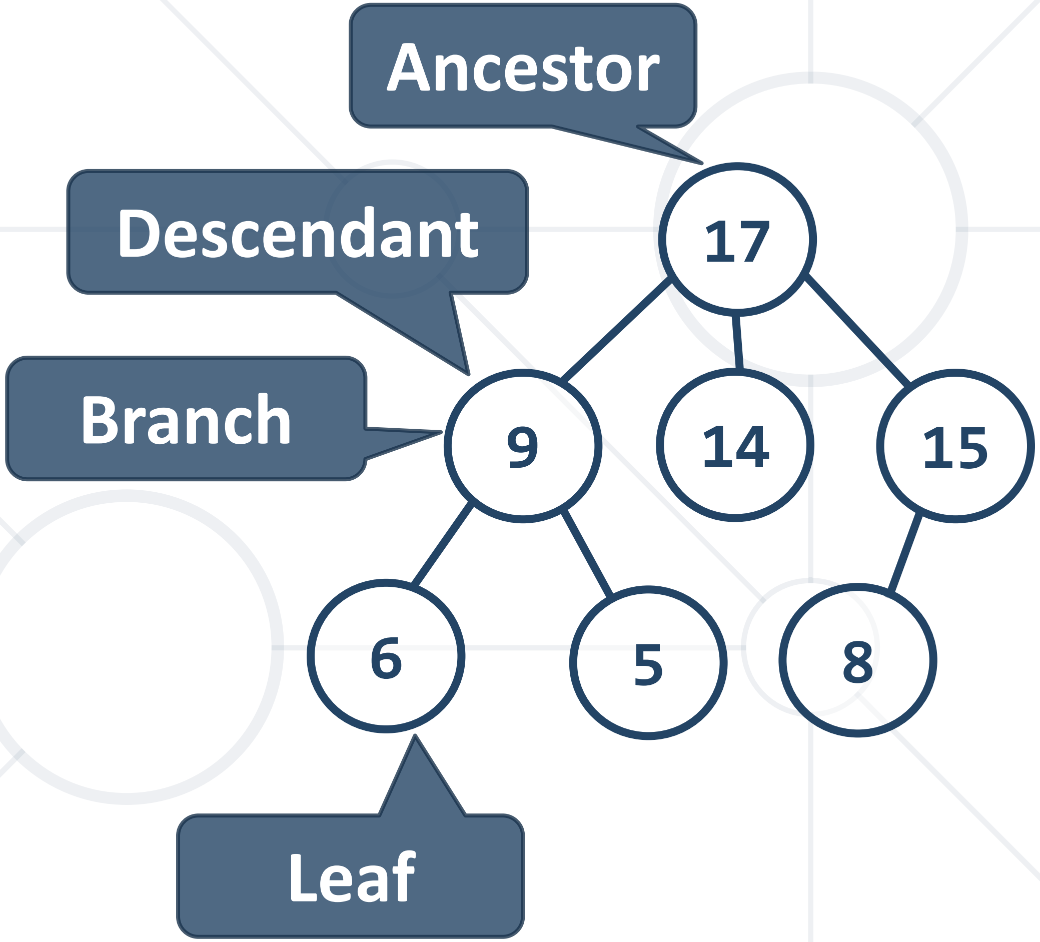
Tree Data Structure – Terminology

- **Parent** – an **immediate ancestor**
  - The **converse** notion of a **child**
- **Child** – an **immediate descendant**
  - Node **directly** connected to **another** node when moving **away** from the **root**
- **Siblings** – a **group** of **nodes** with the **same parent**



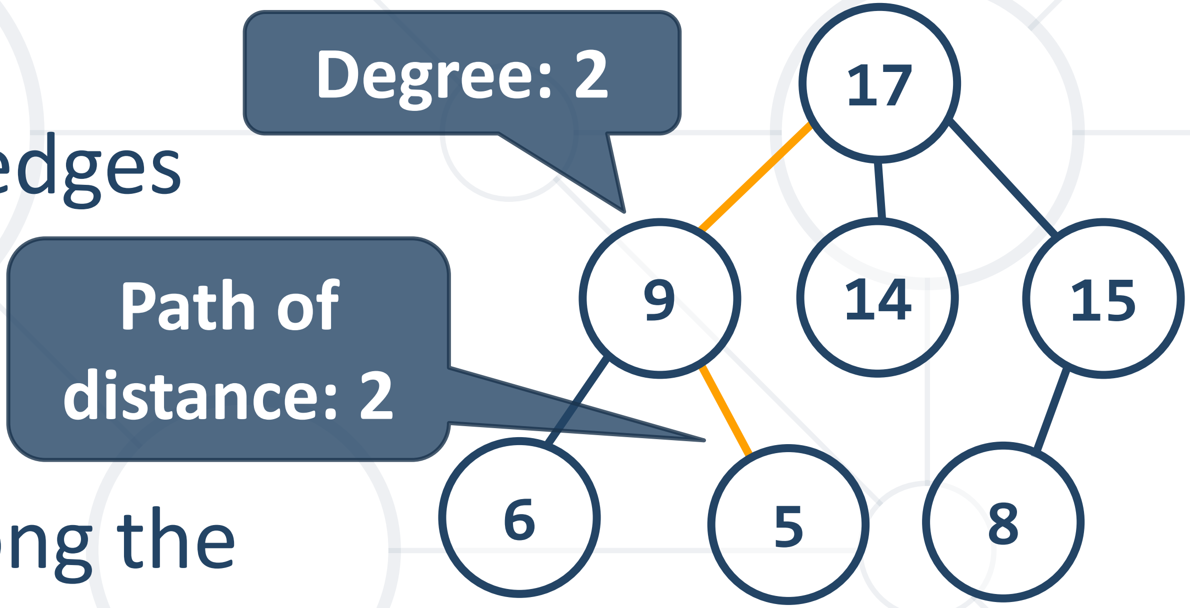
Tree Data Structure – Terminology

- **Ancestor** – node reachable by repeated proceeding **from child to parent**
- **Descendant** – node reachable by repeated proceeding **from parent to child**
- **Leaf** – node with **no children**
- **Branch** – node with **at least one child**



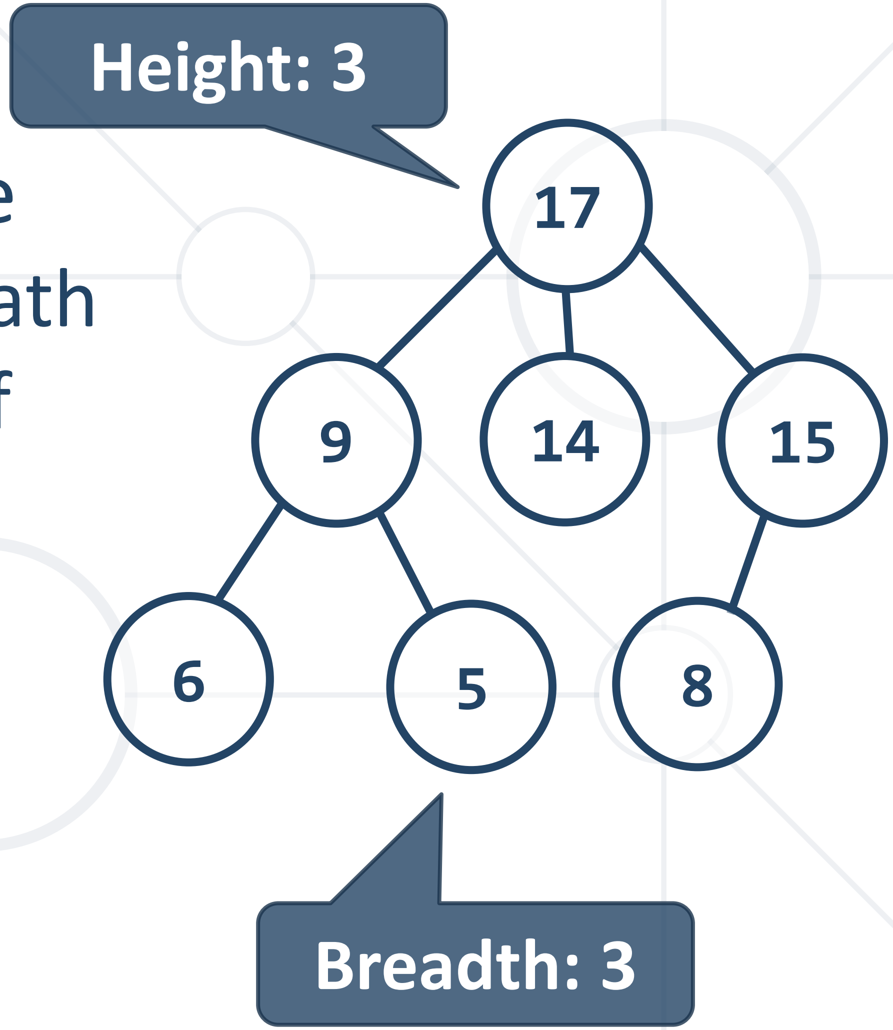
Tree Data Structure – Terminology

- **Degree** – number of children for node zero for a leaf
- **Path** – sequence of nodes and edges connecting a node with a descendant
- **Distance** – number of edges along the shortest path between two nodes
- **Depth** – distance between a node and the root



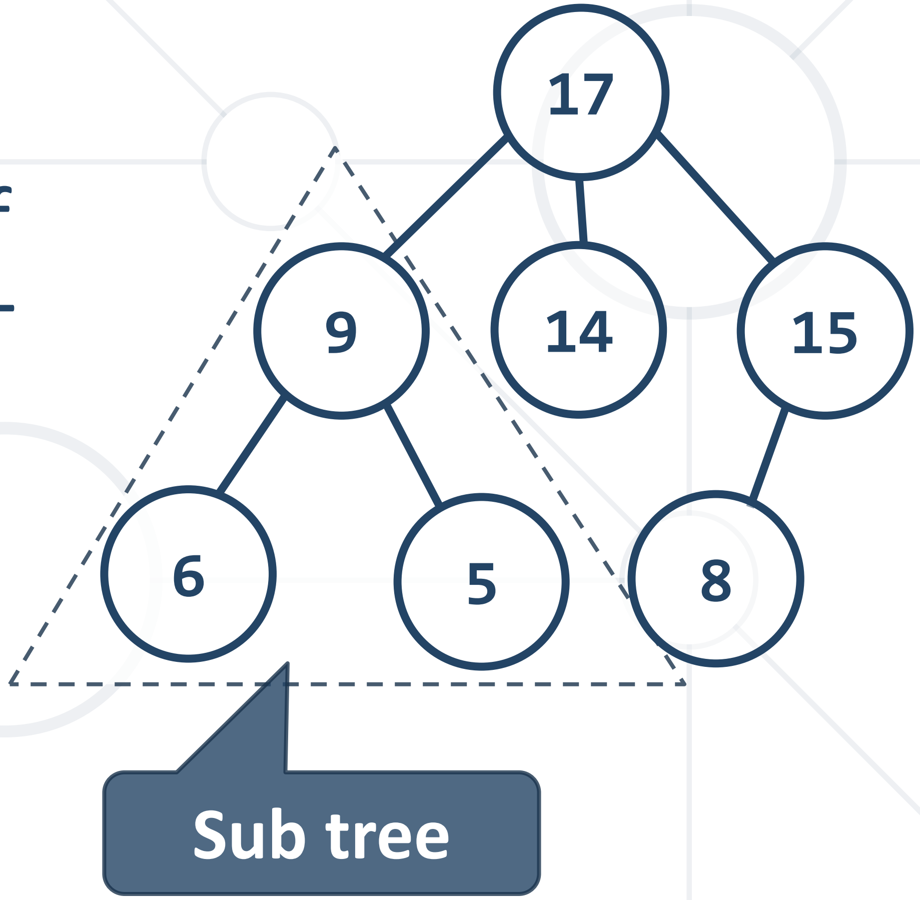
Tree Data Structure – Terminology

- **Level** – depth + 1
- **Height** – the maximum level in the tree
  - The number of edges on the longest path between a node and a descendant leaf
- **Width** – number of nodes in a level
- **Breadth** – number of leaves

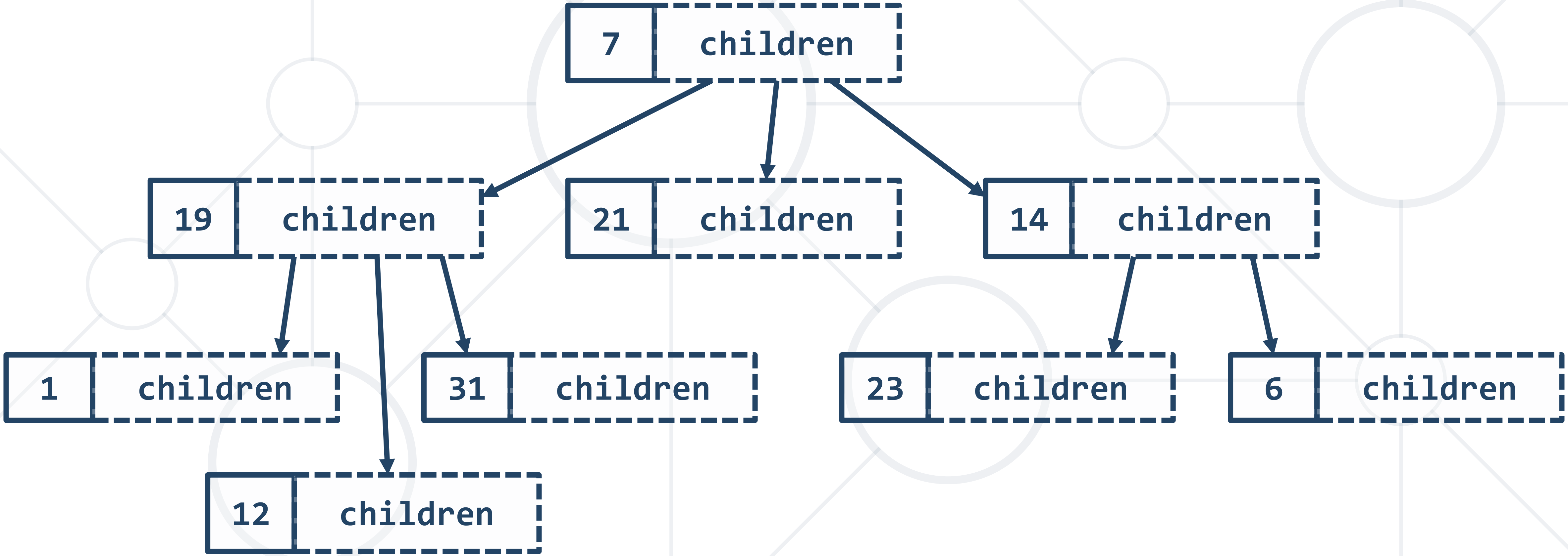


Tree Data Structure – Terminology

- **Forest** – set of disjoint trees
  - {17}, {9, 6, 5}, {14}, {15, 8}
- **Sub tree** – tree T is a tree consisting of a node in T and all its descendants in T

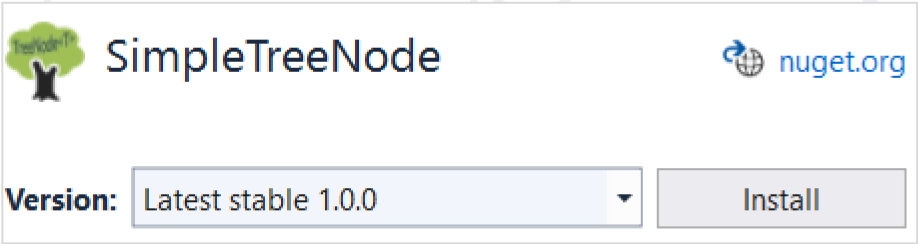


Tree<int> Structure – Example



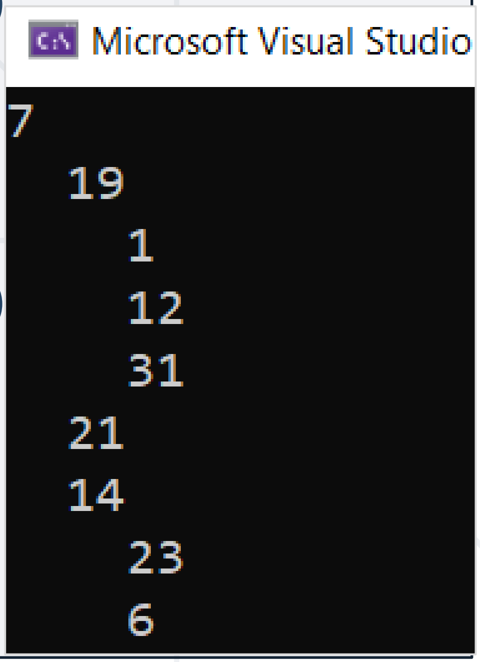
Tree<int> Structure – Example

- First, install the **SimpleTreeNode** NuGet package



- Use the given code to create a **tree**

```
TreeNode<int> tree = new
  TreeNode<int>(7,
   new TreeNode<int>(19,
      new TreeNode<int>(1),
      new TreeNode<int>(12),
      new TreeNode<int>(31)
   ),
   new TreeNode<int>(21),
   new TreeNode<int>(14,
      new TreeNode<int>(23)
      new TreeNode<int>(6)
   )
);
Console.WriteLine(tree);
```

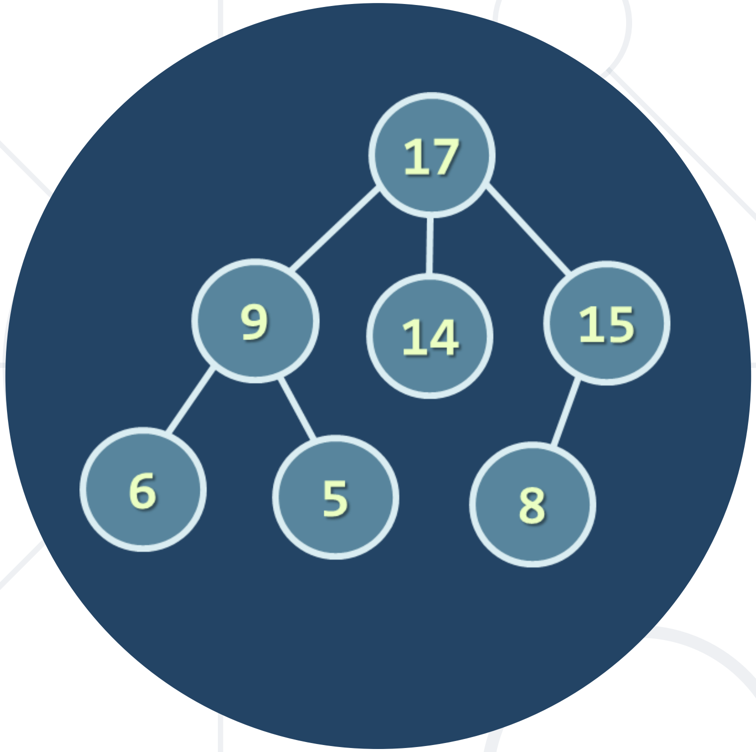


Example: XML Tree in C#

```
XElement contacts =
    new XElement("Contacts",
        new XElement("Contact",
            new XElement("Name", "Patrick Hines"),
            new XElement("Phone", "206-555-0144"),
            new XElement("Address",
                new XElement("Street1", "123 Ma
                new XElement("City", "Mercer Is
                new XElement("State", "WA"),
                new XElement("Postal", "68042")
            )
        )
    );

Console.WriteLine(contacts);
```

Microsoft Visual Studio Debug Console

```
<Contacts>
  <Contact>
    <Name>Patrick Hines</Name>
    <Phone>206-555-0144</Phone>
    <Address>
      <Street1>123 Main St</Street1>
      <City>Mercer Island</City>
      <State>WA</State>
      <Postal>68042</Postal>
    </Address>
  </Contact>
</Contacts>
```



DFS and BFS Traversals

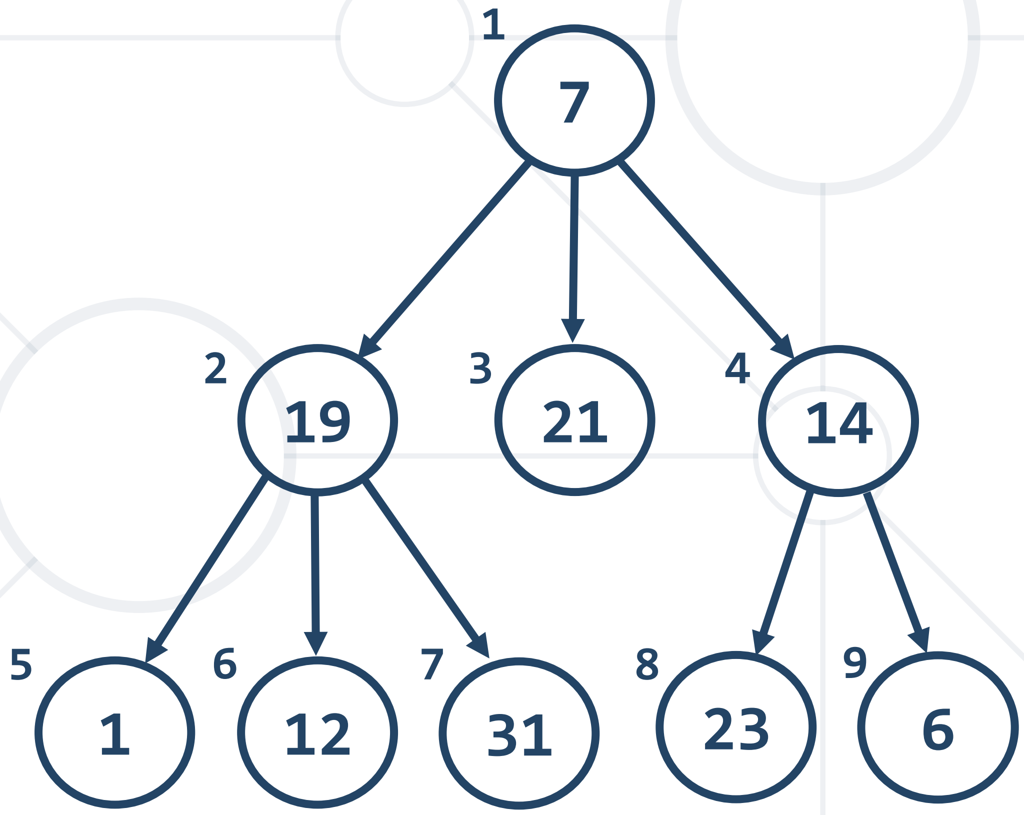
Tree Traversal Algorithms

- **Traversing a tree** 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 by a **queue**

Breadth-First Search (BFS)

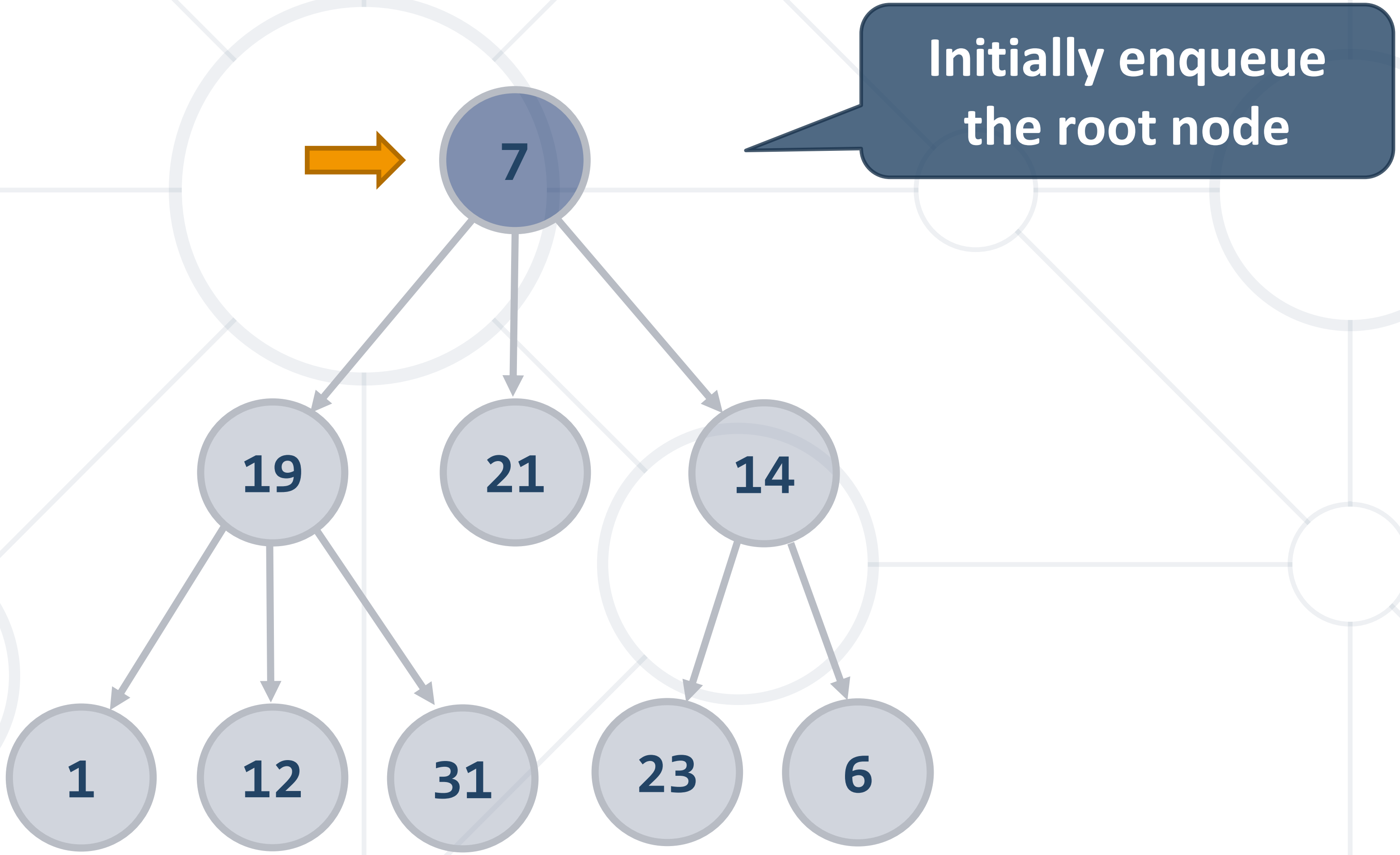
- **Breadth-First Search** (BFS) first visits the **neighbor nodes**, then the **neighbors of neighbors**, etc.
- **BFS** algorithm pseudo code:

```
BFS (node) {
  queue ← node
  while queue not empty
    v ← queue
    print v
    for each child c of v
      queue ← c
}
```



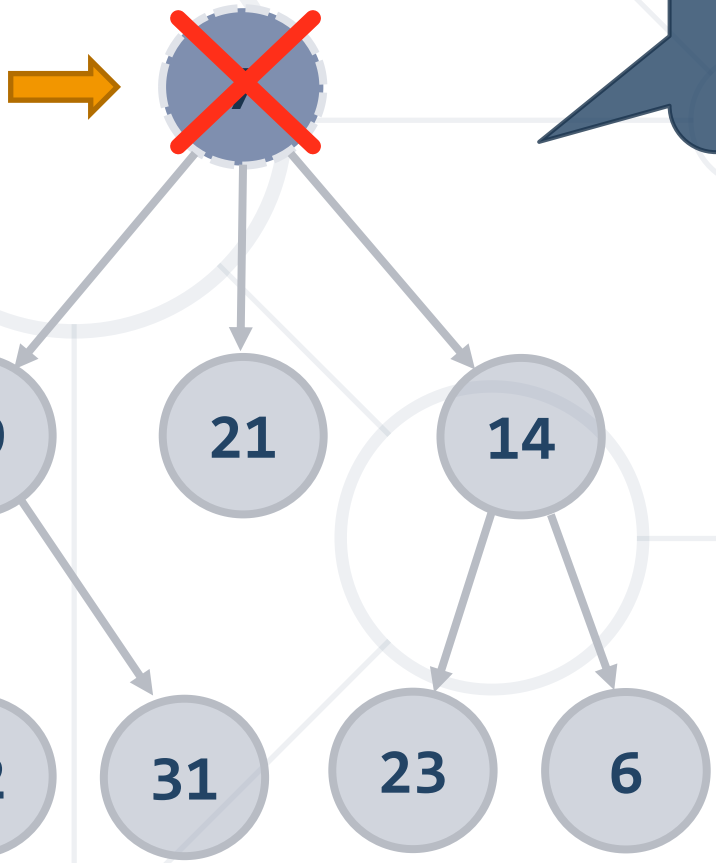
BFS in Action (Step 1)

- Queue: 7
- Output:



BFS in Action (Step 2)

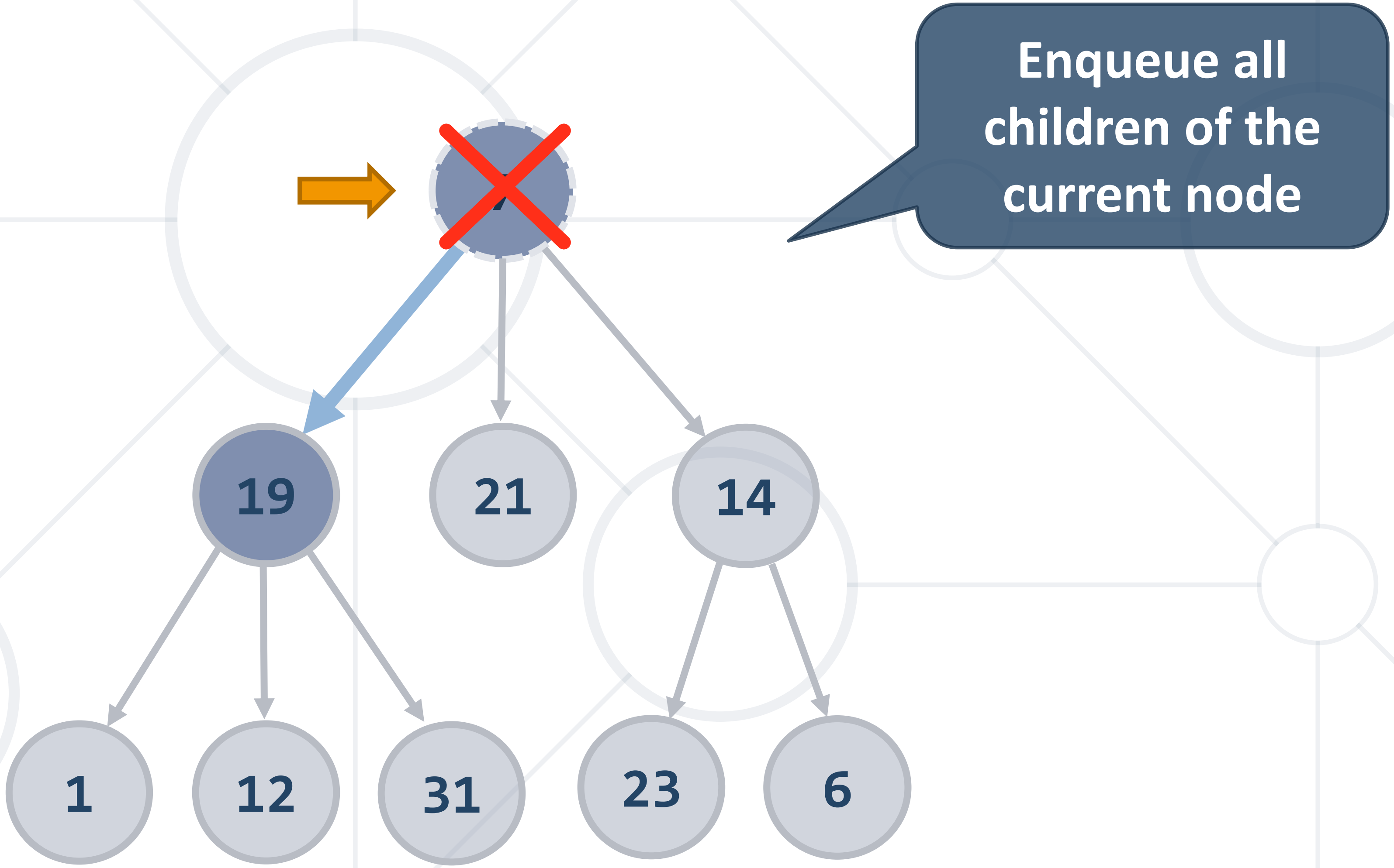
- Queue: ~~7~~
- Output: 7



Remove from the queue the next node and print it

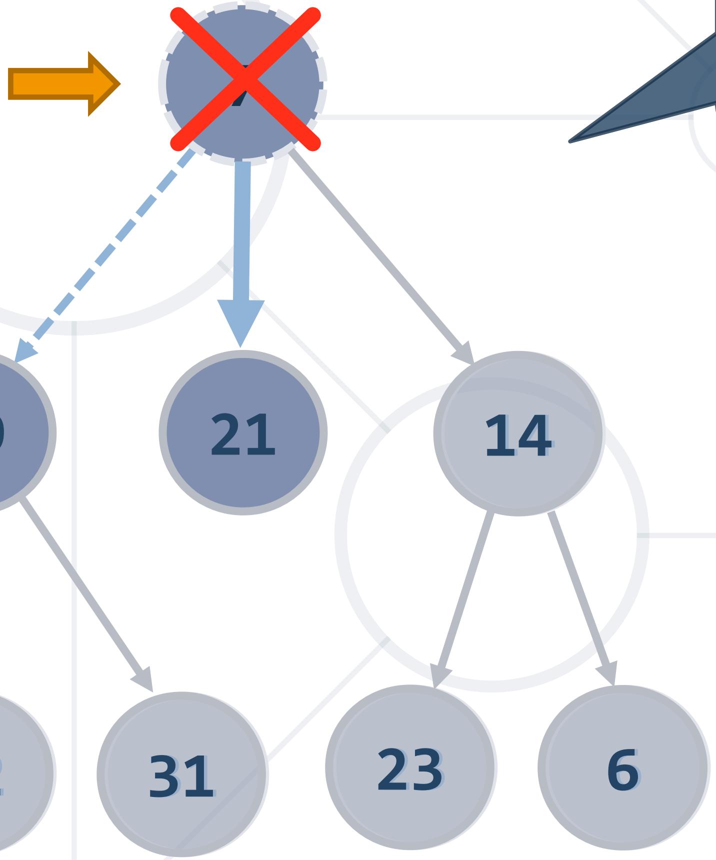
BFS in Action (Step 3)

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



BFS in Action (Step 4)

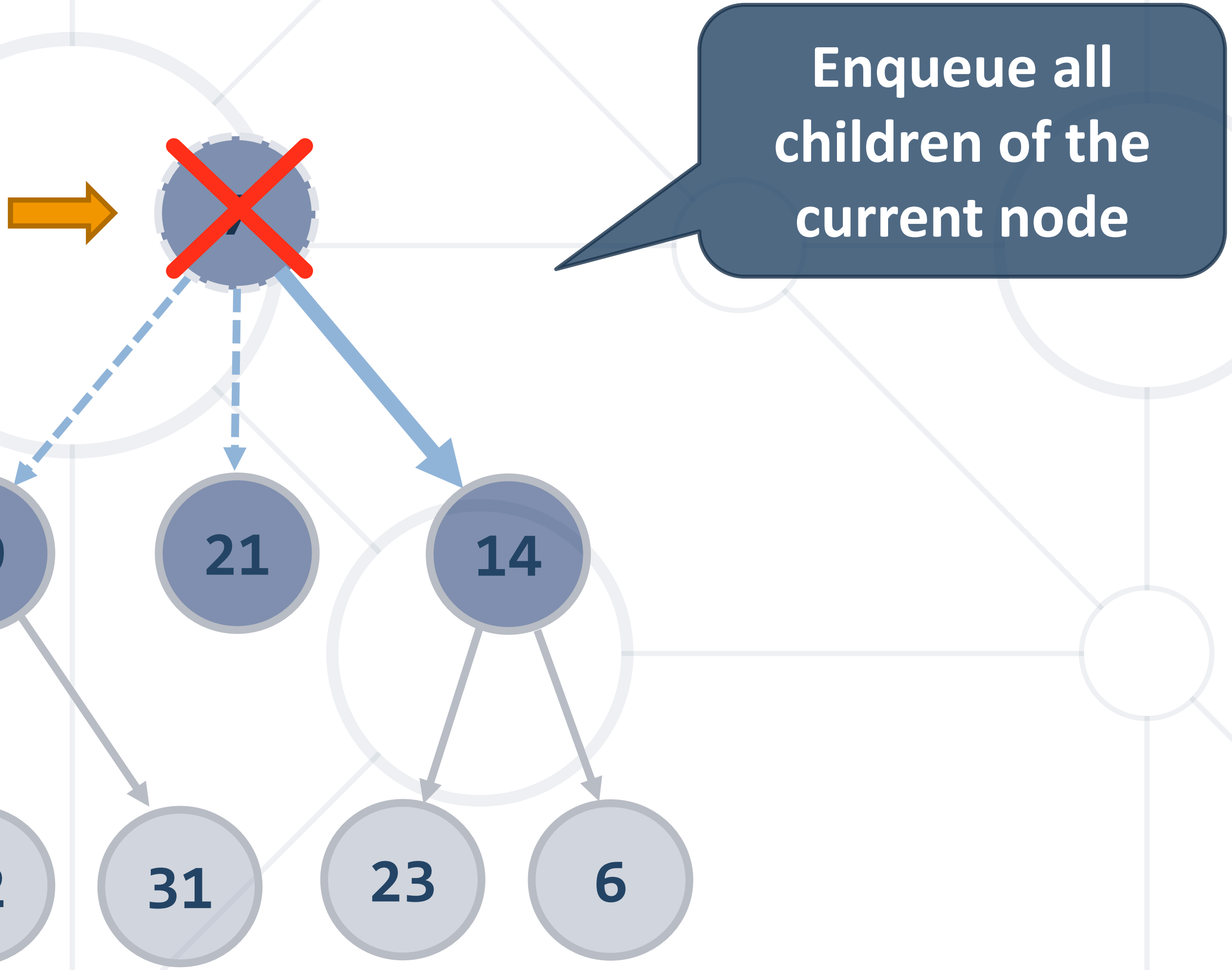
- Queue: ~~7~~, 19, 21
- Output: 7



Enqueue all children of the current node

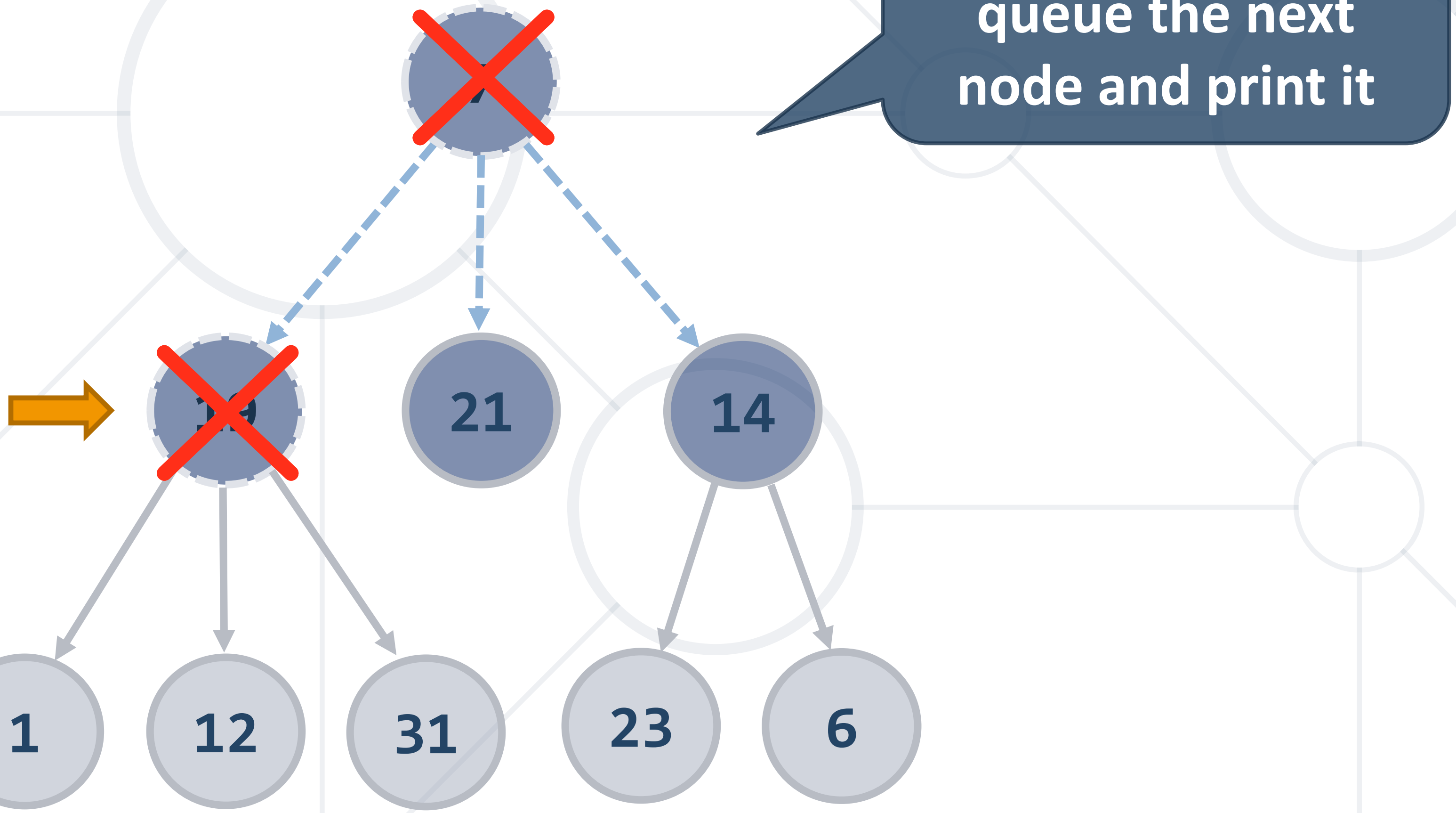
BFS in Action (Step 5)

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



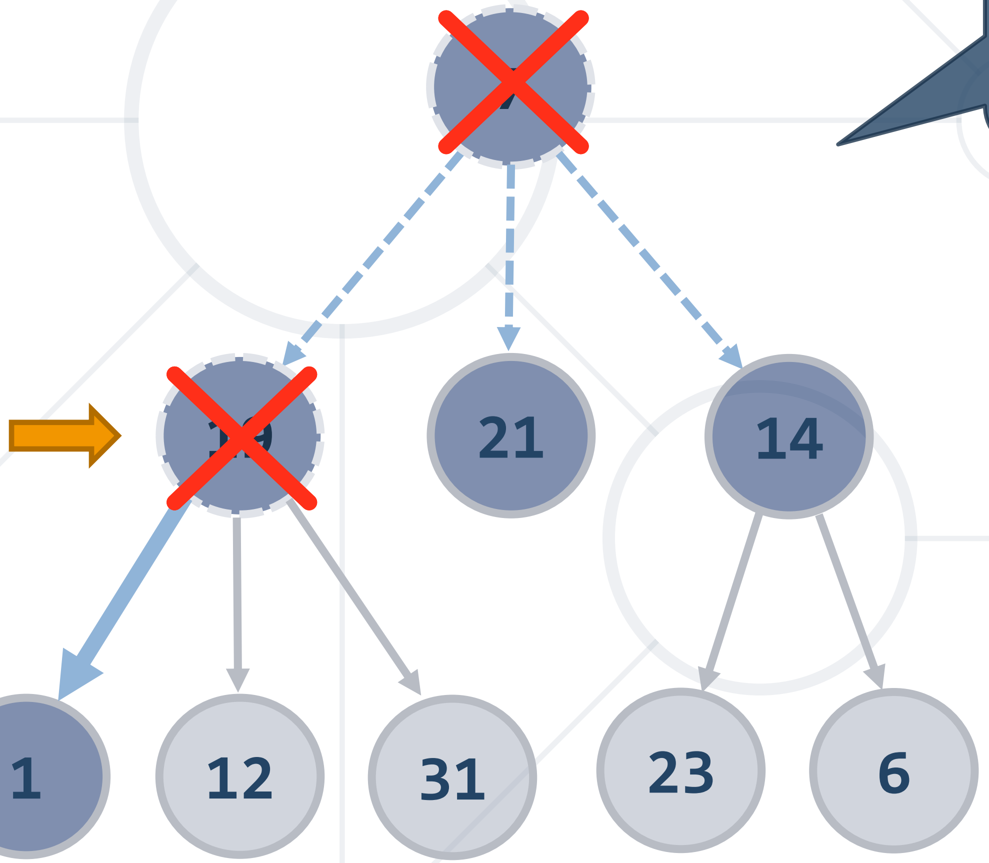
BFS in Action (Step 6)

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



BFS in Action (Step 7)

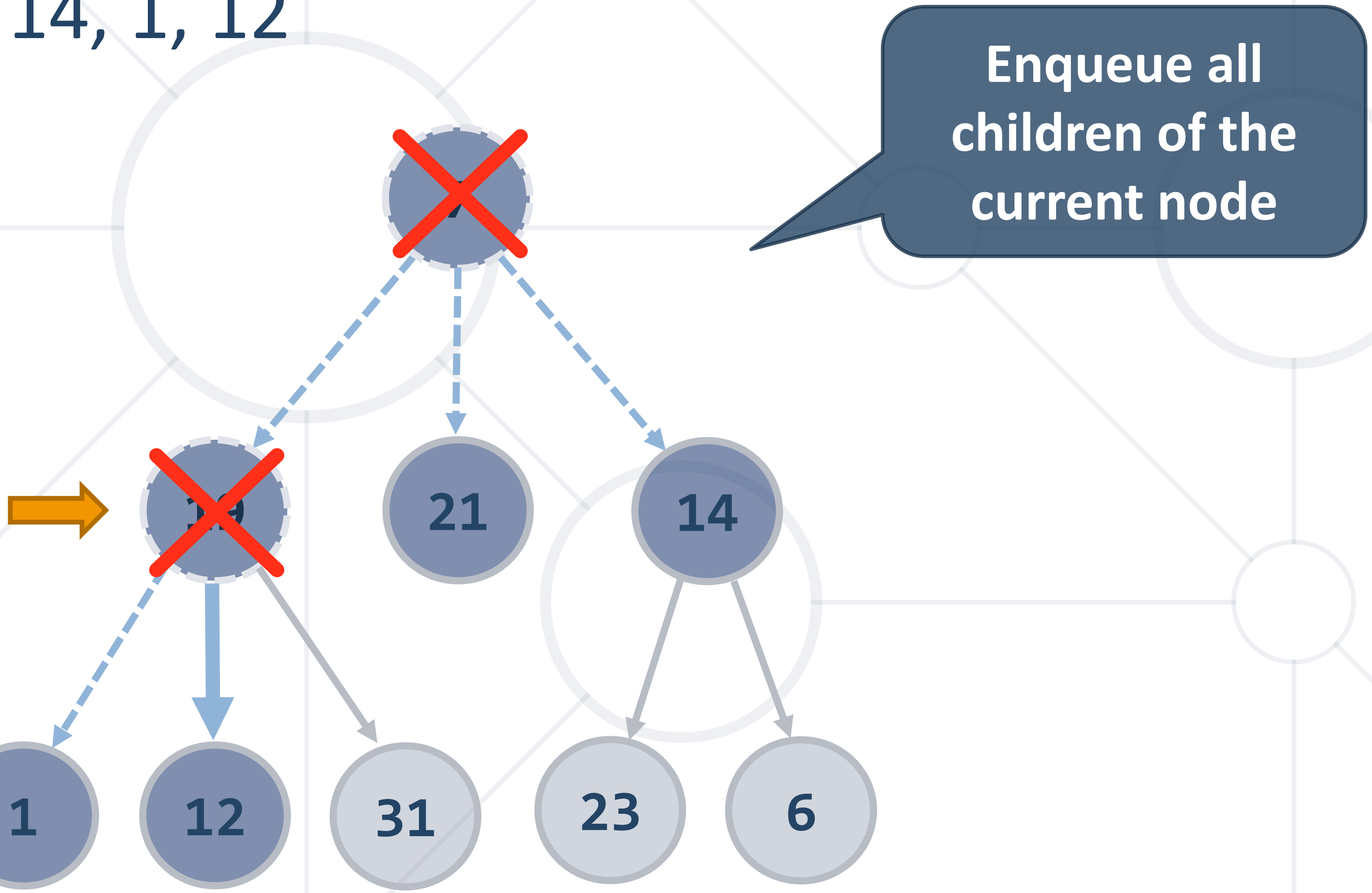
- Queue: ~~7~~, ~~19~~, 21, 14, 1
- Output: 7, 19



Enqueue all children of the current node

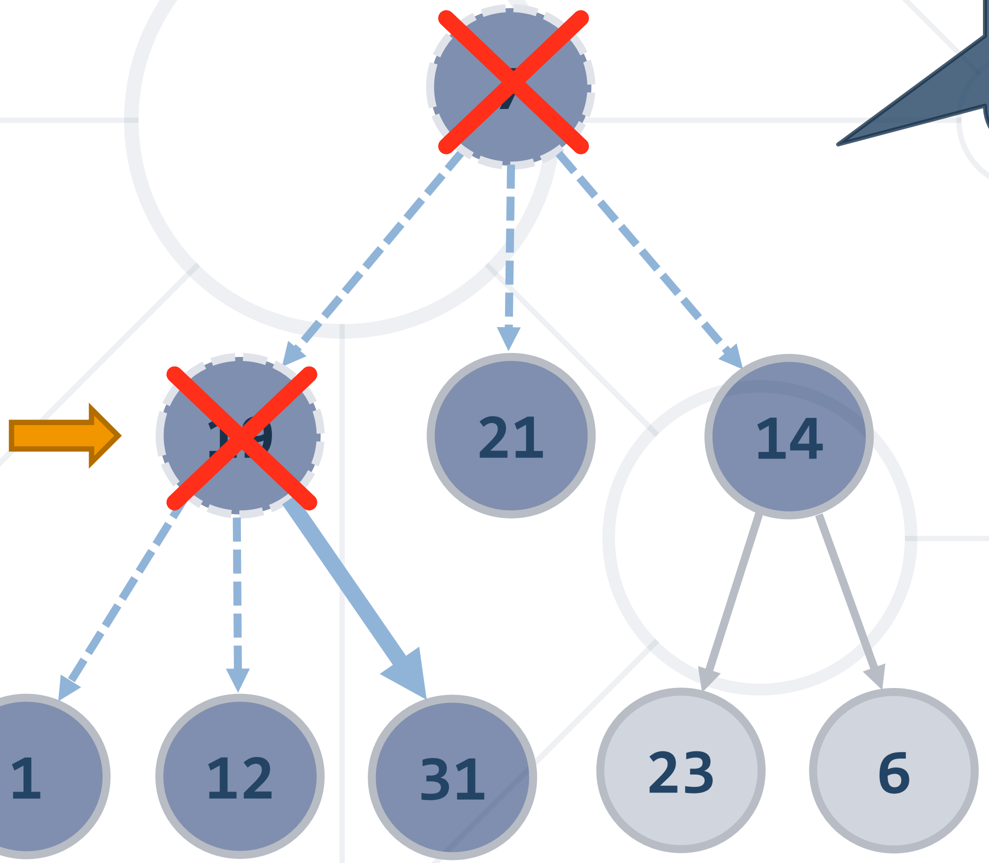
BFS in Action (Step 8)

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



BFS in Action (Step 9)

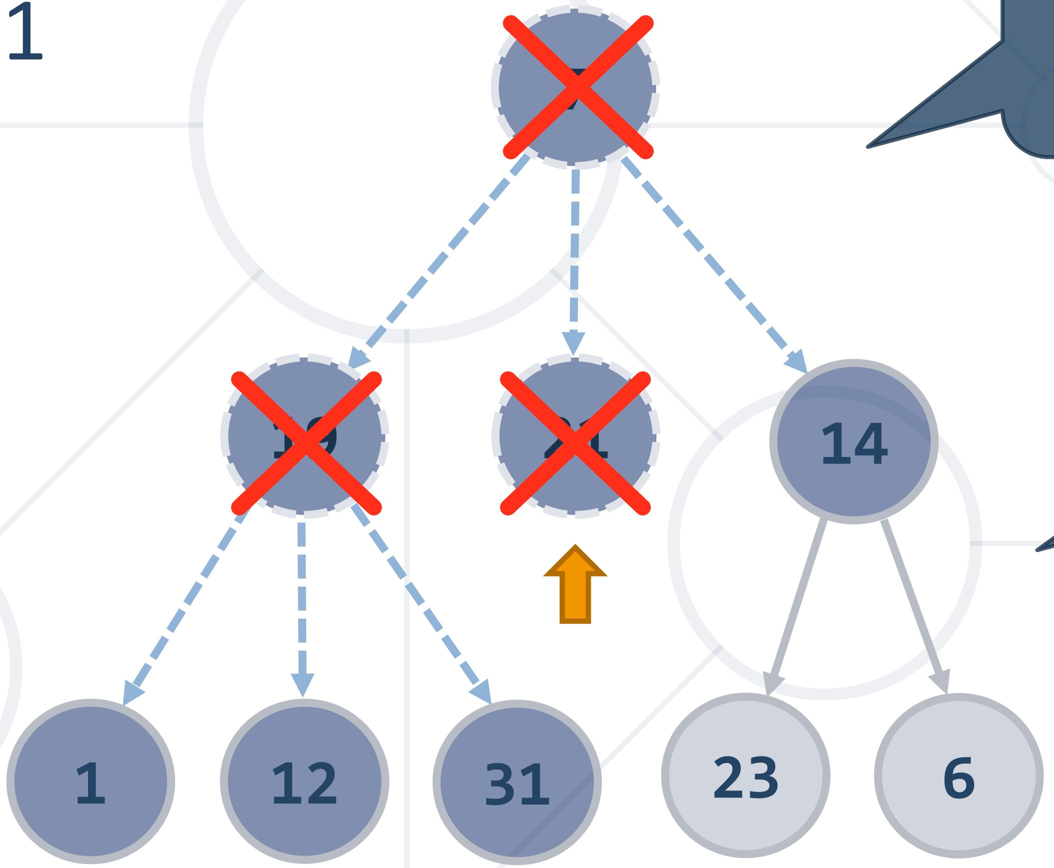
- Queue: ~~7~~, ~~19~~, 21, 14, 1, 12, 31
- Output: 7, 19



Enqueue all children of the current node

BFS in Action (Step 10)

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

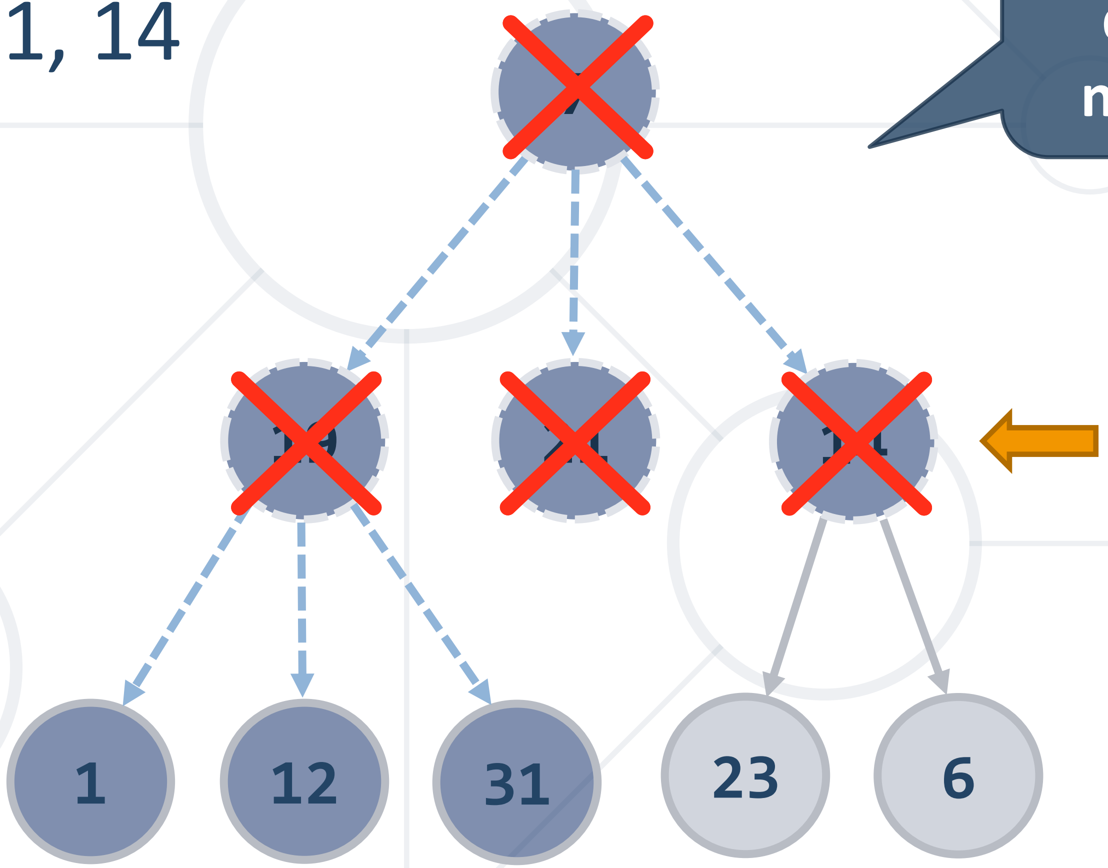


Remove from the queue the next node and print it

No child nodes to enqueue

BFS in Action (Step 11)

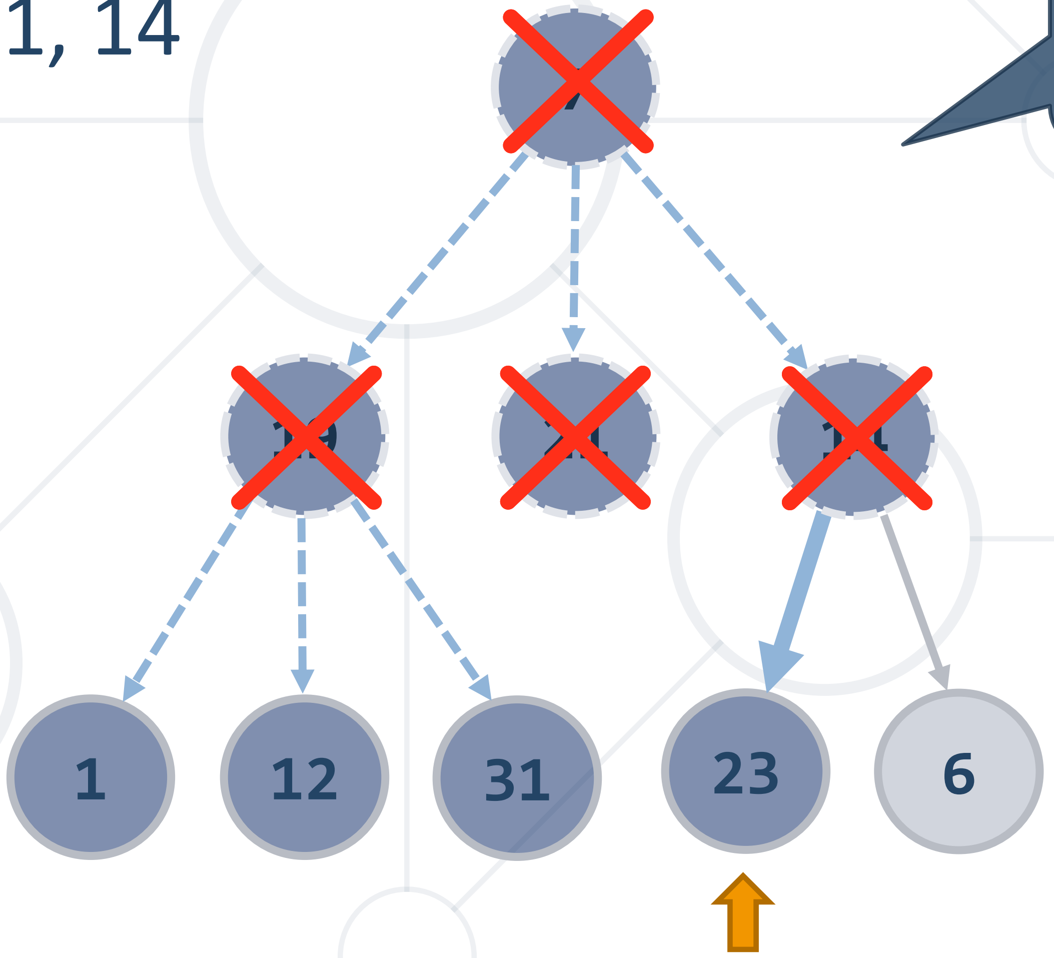
- Queue: ~~7~~, ~~19~~, ~~21~~, ~~14~~, 1, 12, 31
- Output: 7, 19, 21, 14



Remove from the queue the next node and print it

BFS in Action (Step 12)

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

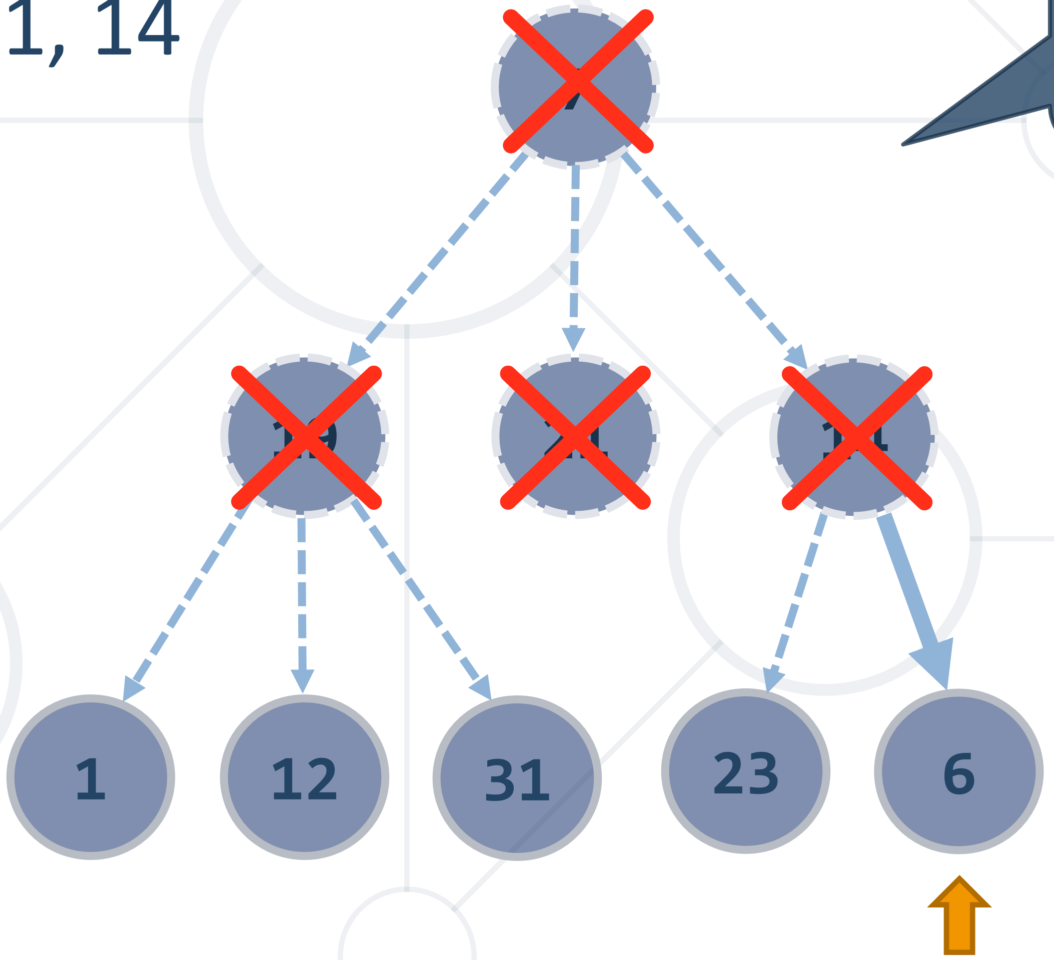


Enqueue all children of the current node

BFS in Action (Step 13)

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

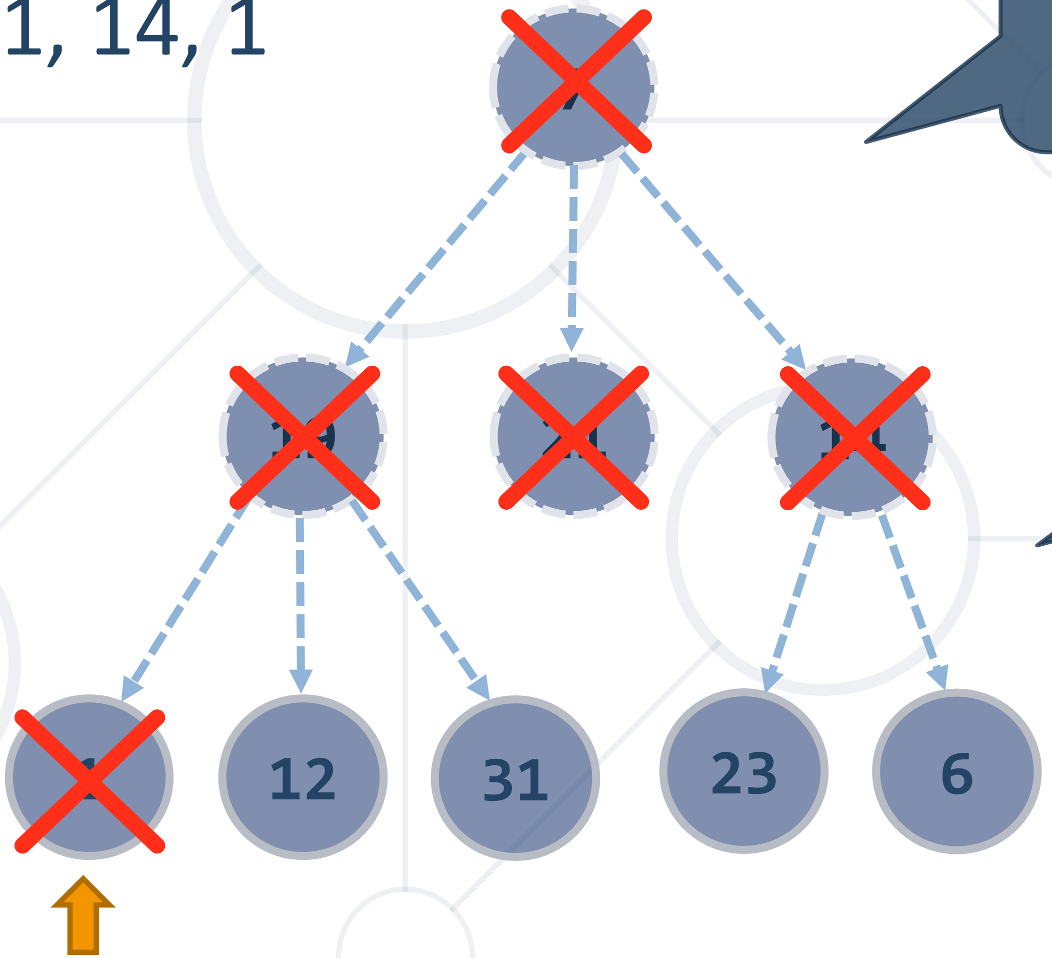
Enqueue all children of the current node



BFS in Action (Step 14)

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

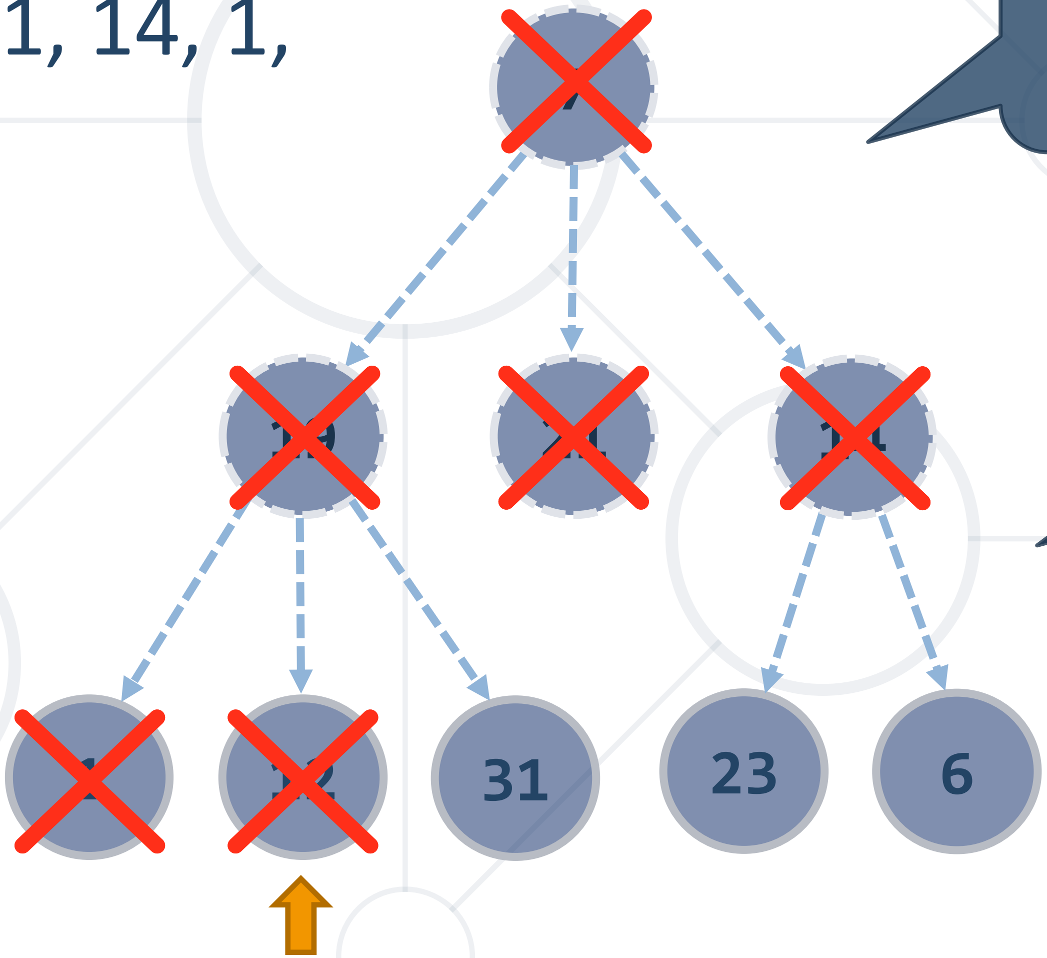
Remove from the queue the next node and print it



No child nodes to enqueue

BFS in Action (Step 15)

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

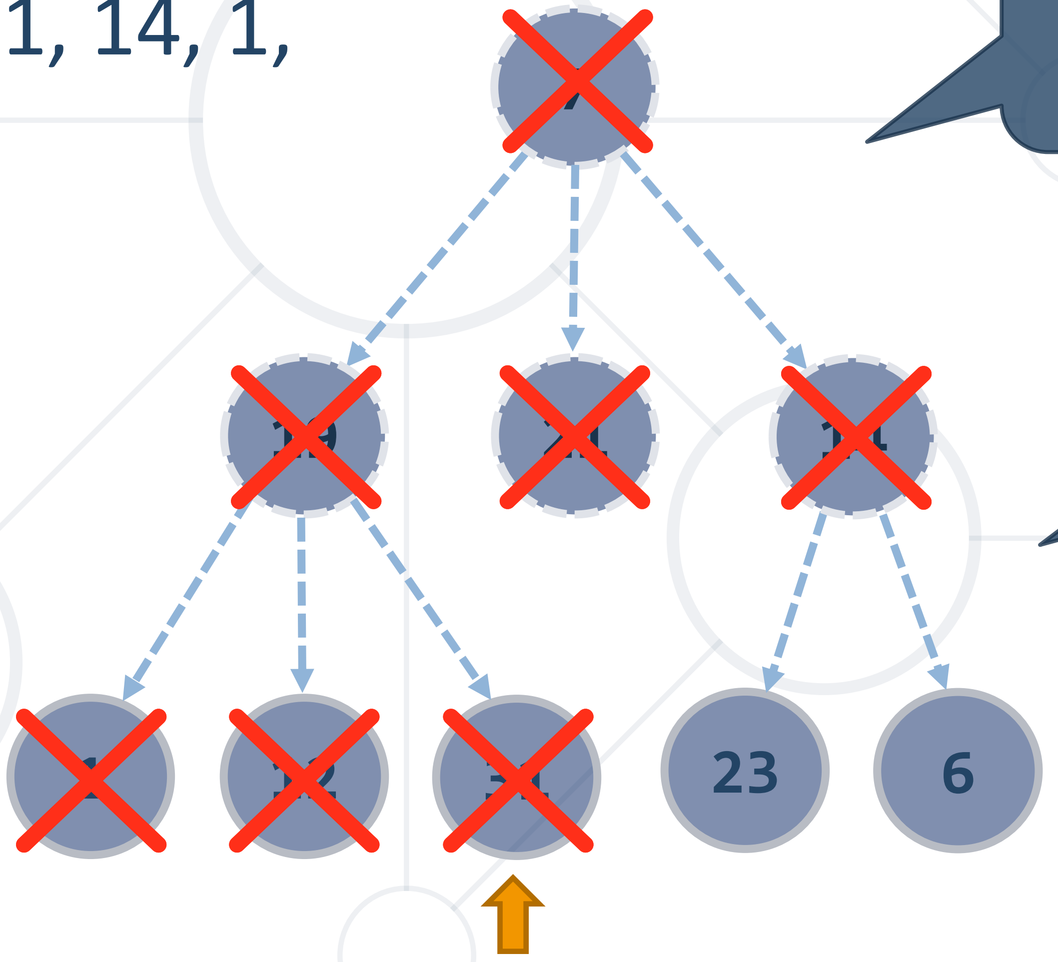


Remove from the queue the next node and print it

No child nodes to enqueue

BFS in Action (Step 16)

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

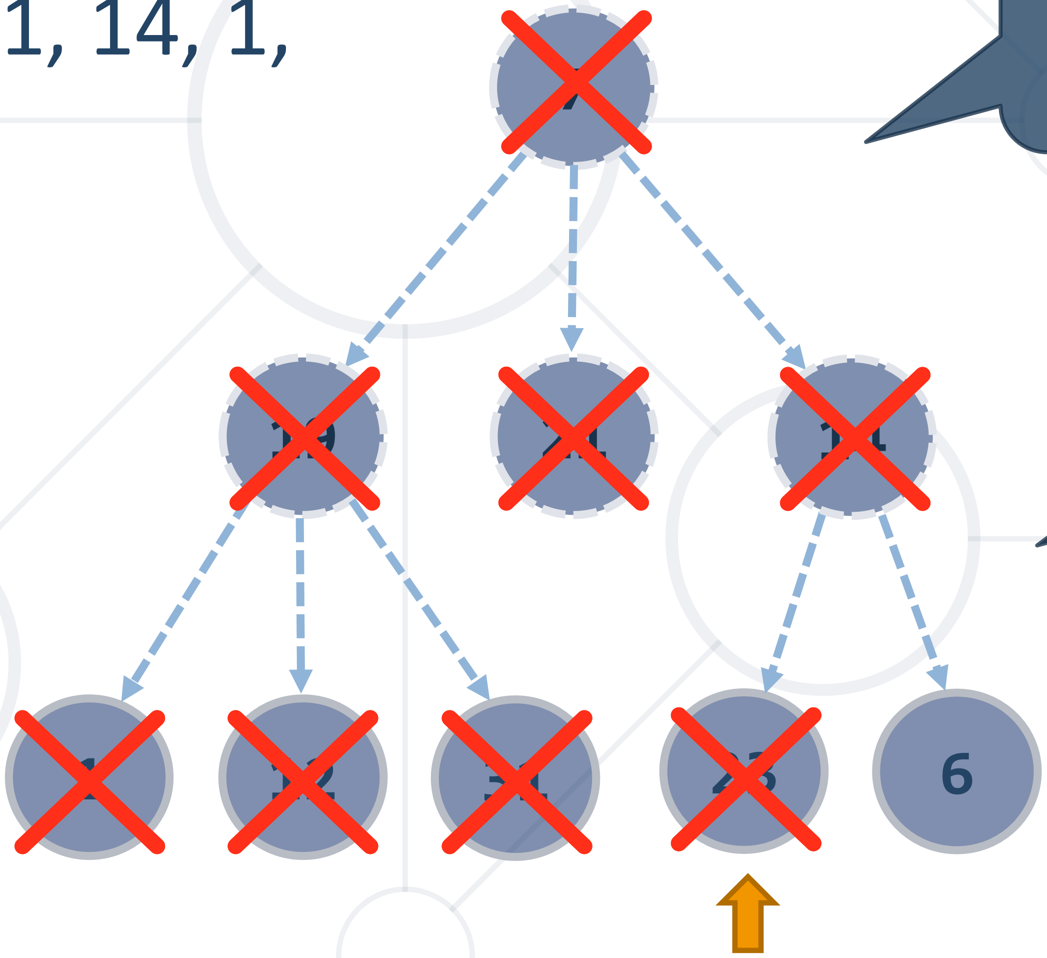


Remove from the queue the next node and print it

No child nodes to enqueue

BFS in Action (Step 17)

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

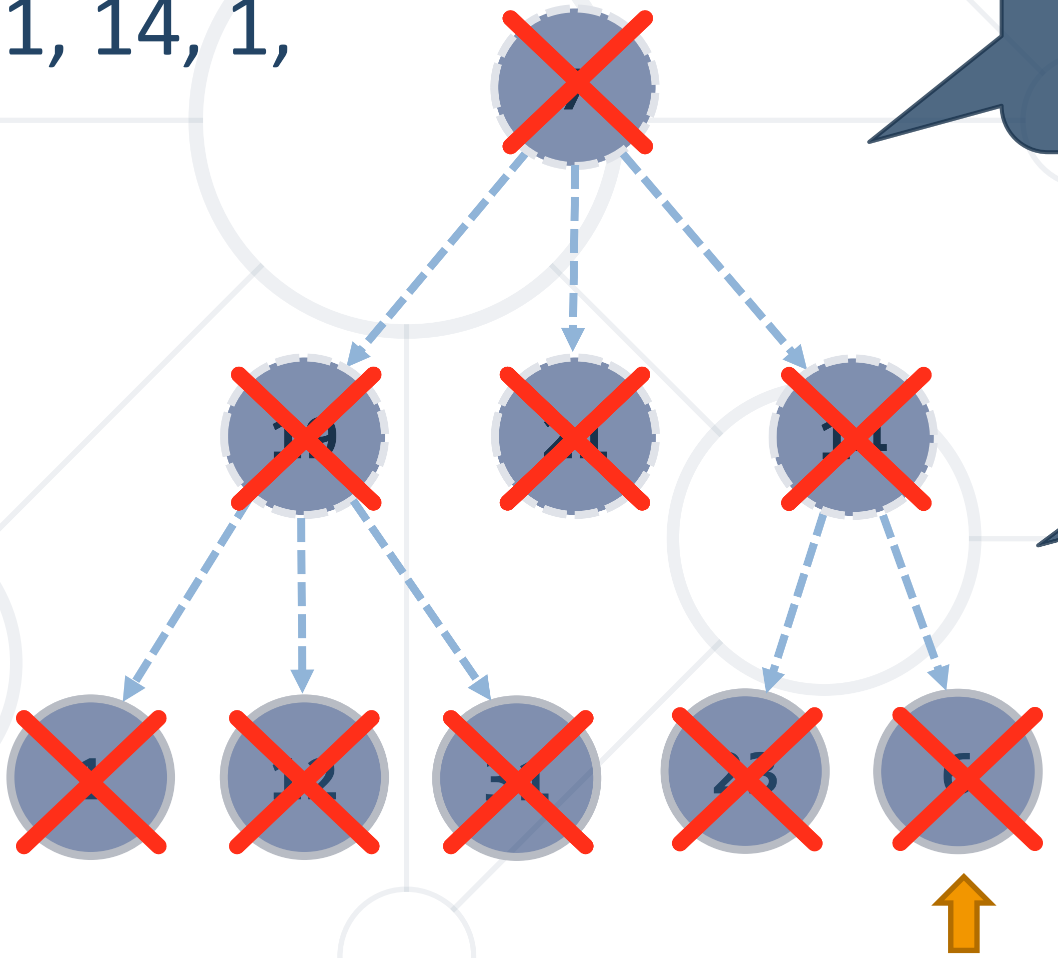


Remove from the queue the next node and print it

No child nodes to enqueue

BFS in Action (Step 18)

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

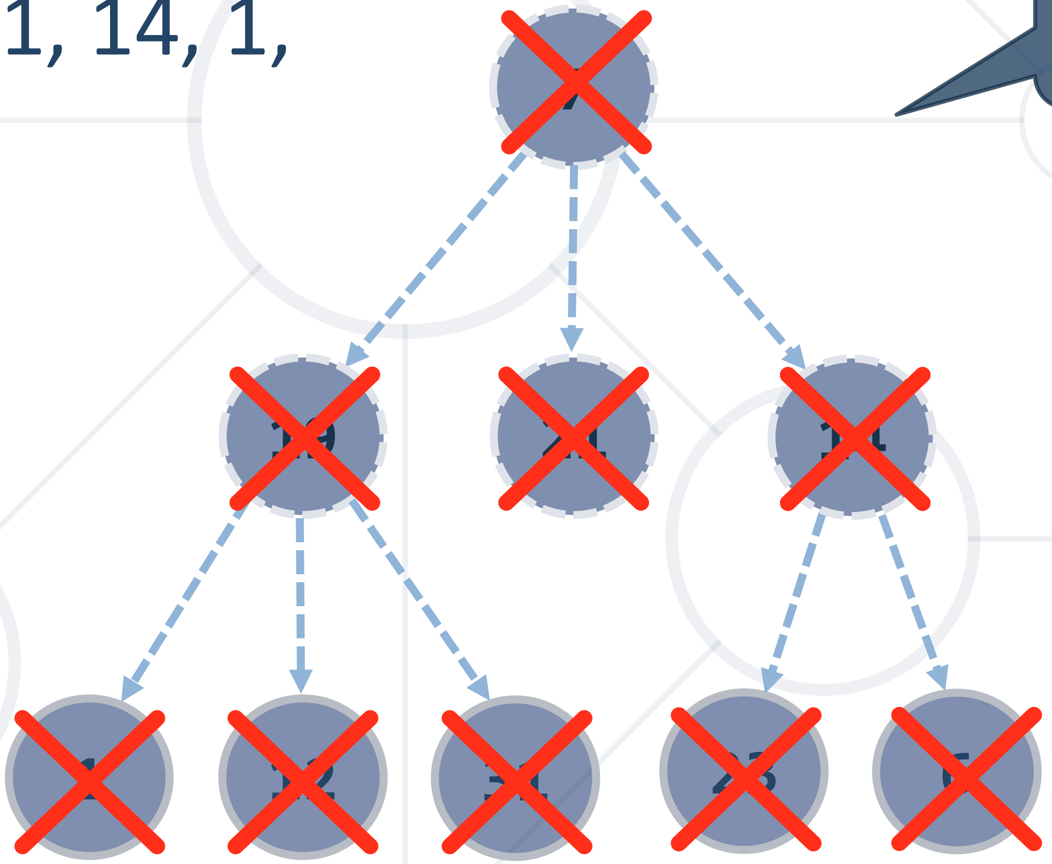


Remove from the queue the next node and print it

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, 23, 6



The queue is empty → stop

BFS Example: Traverse Folders

```
static void TraverseDirBFS(string directoryPath) {
   var visitedDirsQueue = new Queue<DirectoryInfo>();
   visitedDirsQueue.Enqueue(new DirectoryInfo(directoryPath));

   while (visitedDirsQueue.Count > 0) {
      DirectoryInfo currentDir = visitedDirsQueue.Dequeue();
      Console.WriteLine(currentDir.FullName);
      DirectoryInfo[] children = currentDir.GetDirectories();

      foreach (DirectoryInfo child in children)
         visitedDirsQueue.Enqueue(child);
   }
}

static void Main() {
   TraverseDirBFS(@"C:\Windows\assembly");
}
```

DirectoryInfo class allows accessing directories

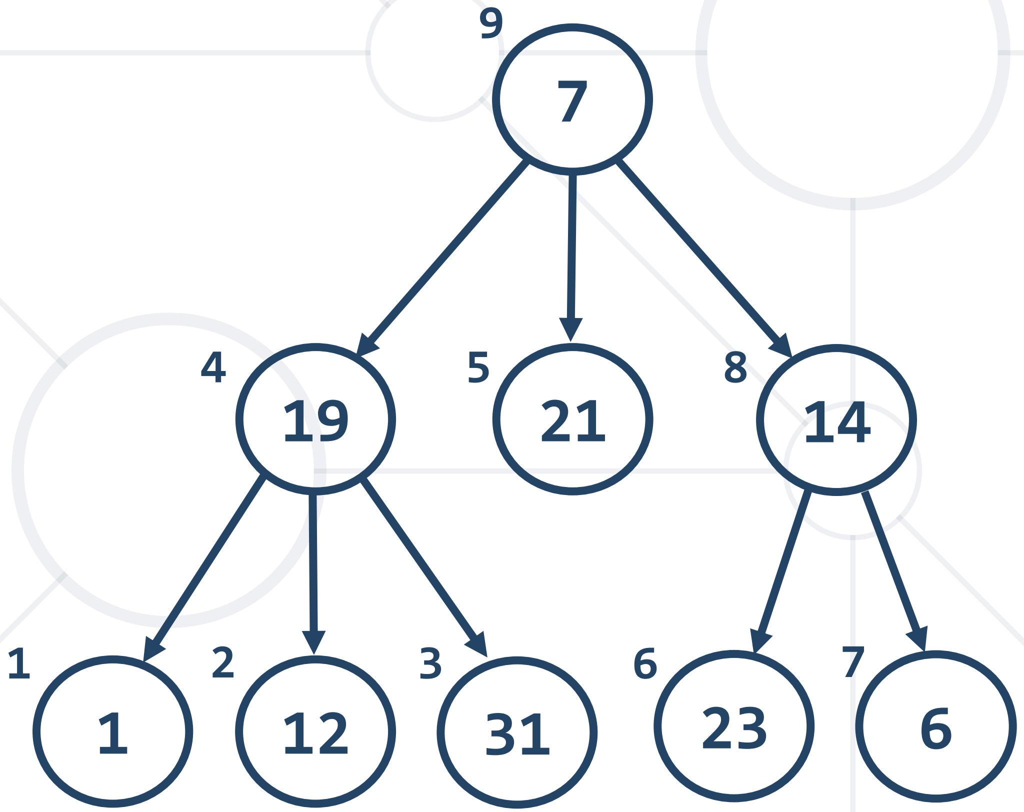
Store visited directories in a queue

Use GetDirectories() method to get sub-directories

Depth-First Search (DFS)

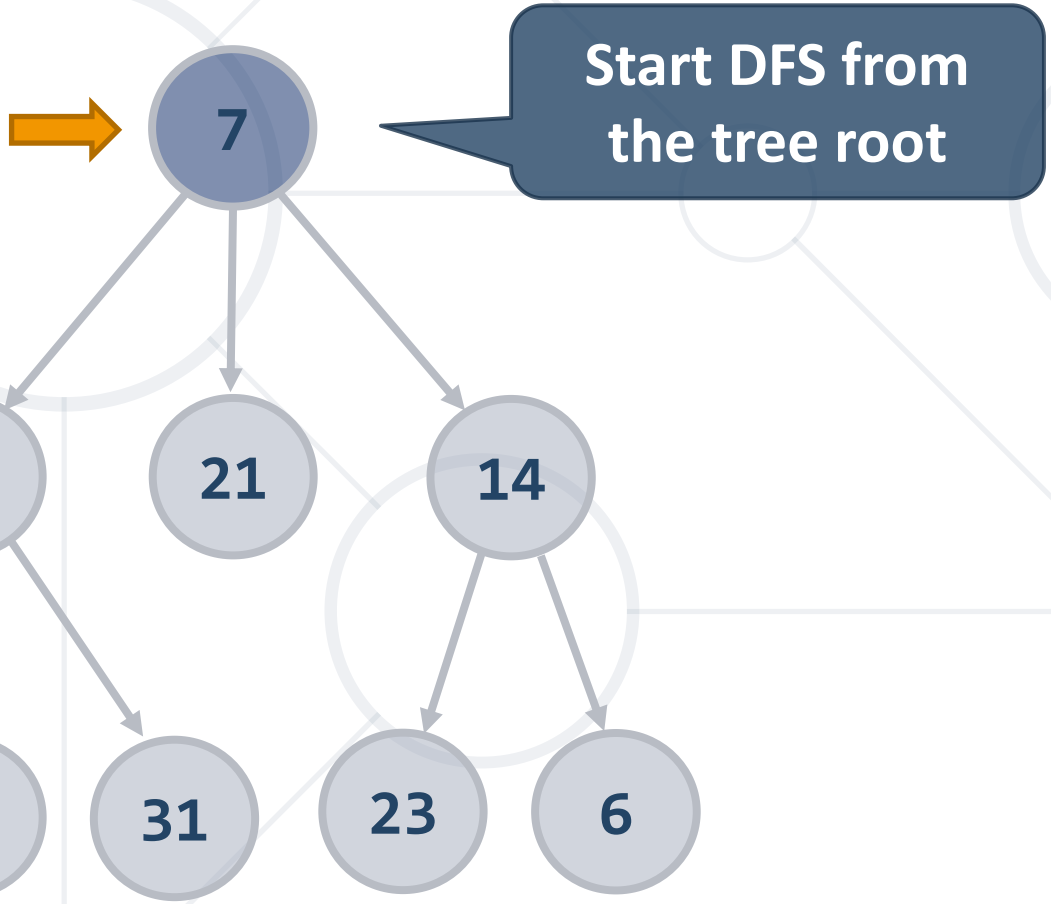
- **Depth-First Search** (**DFS**) first visits all descendants of given node recursively, finally visits the node itself
- **DFS** algorithm pseudo code:

```
DFS (node) {
  for each child c of node
    DFS(c);
  print node;
}
```



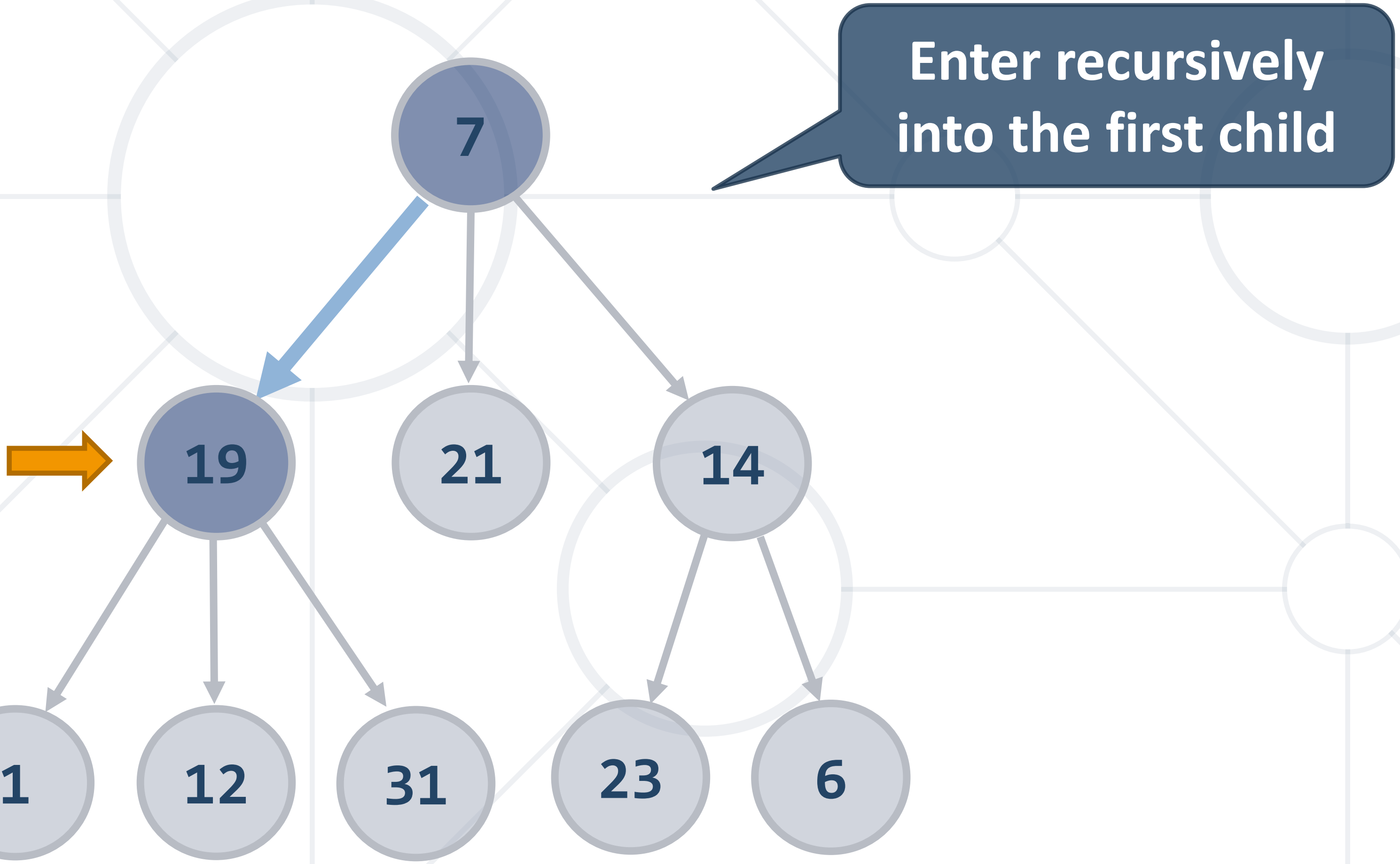
DFS in Action (Step 1)

- Stack: 7
- Output: (empty)



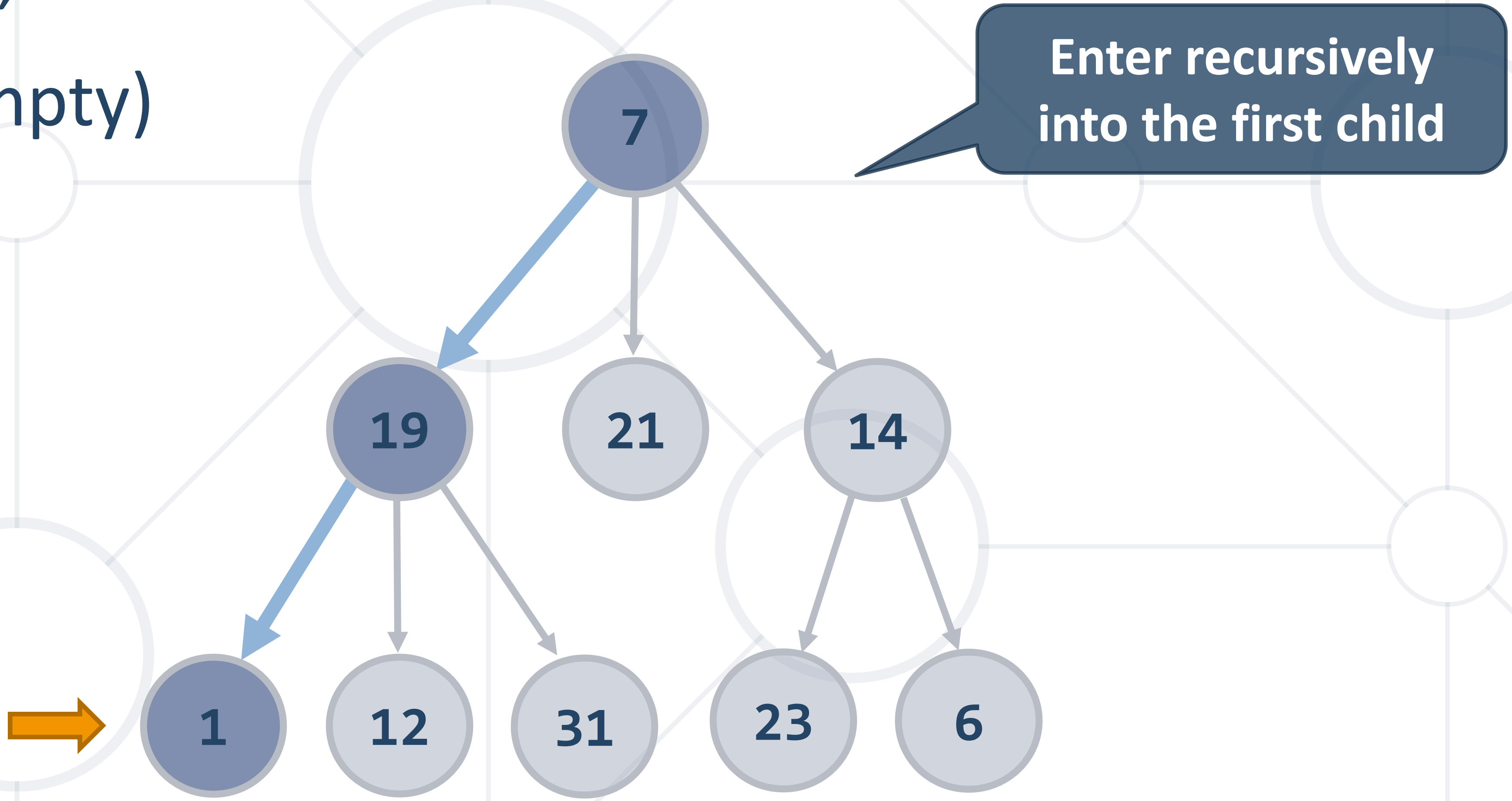
DFS in Action (Step 2)

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



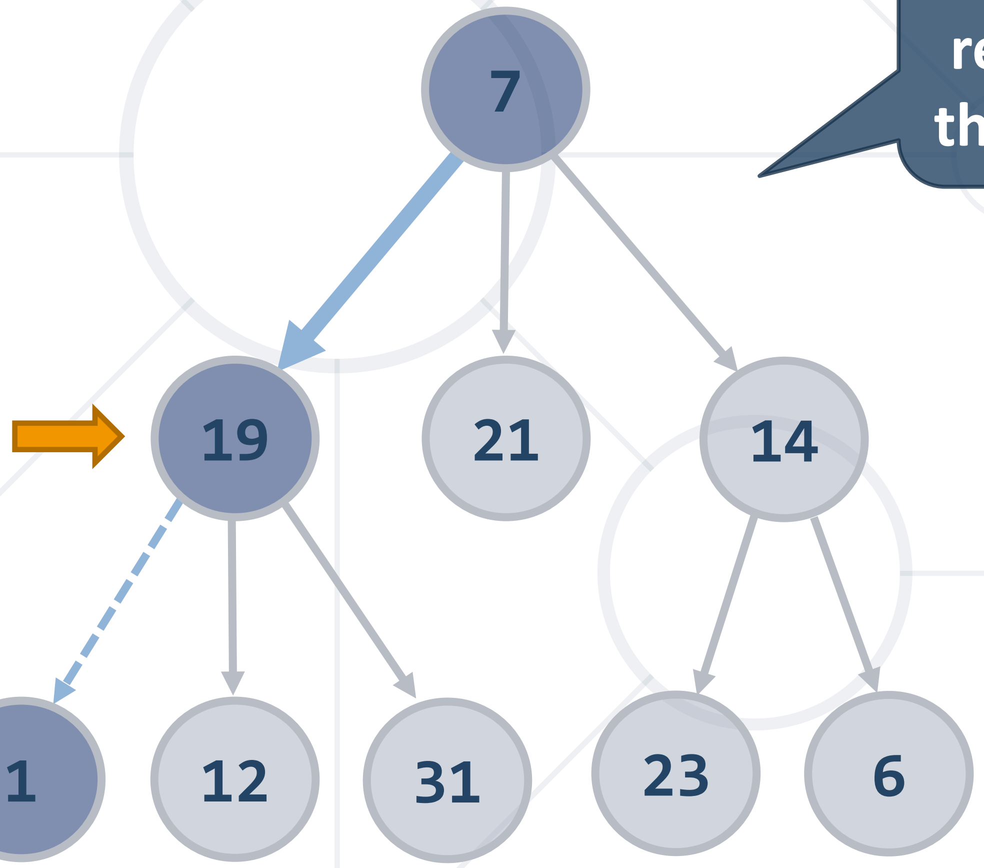
DFS in Action (Step 3)

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



DFS in Action (Step 4)

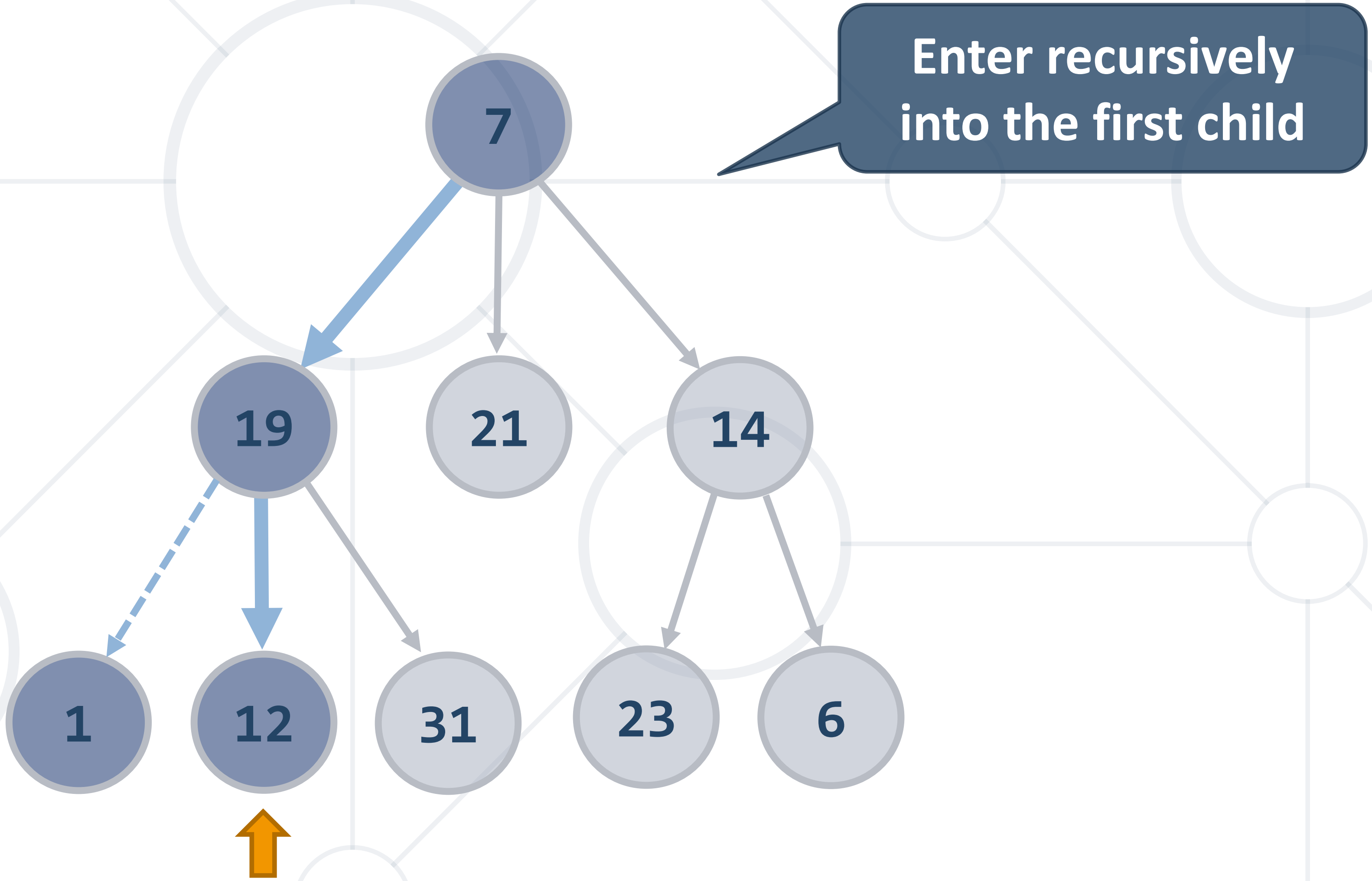
- Stack: 7, 19
- Output: 1



Return back from recursion and print the last visited node

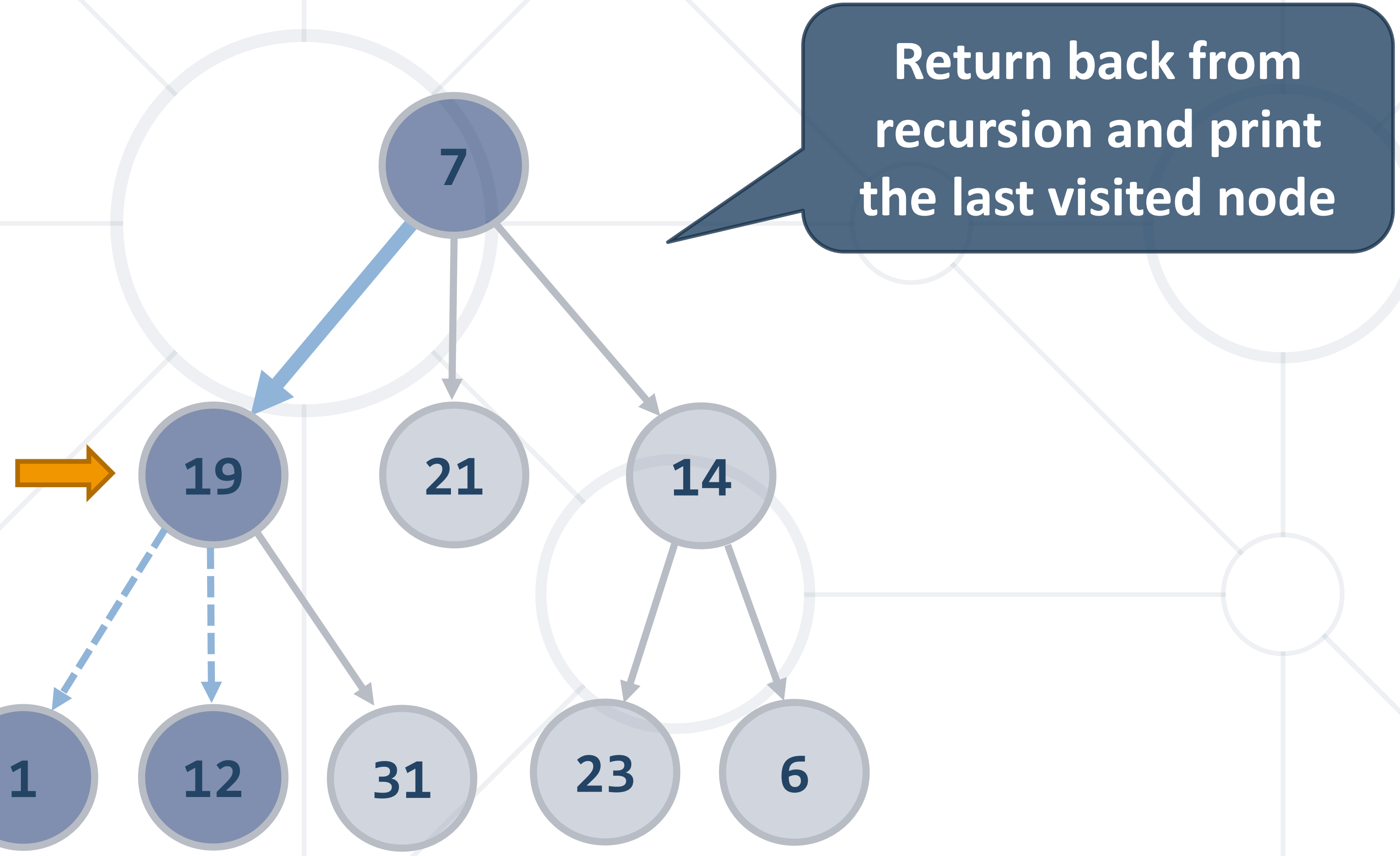
DFS in Action (Step 5)

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



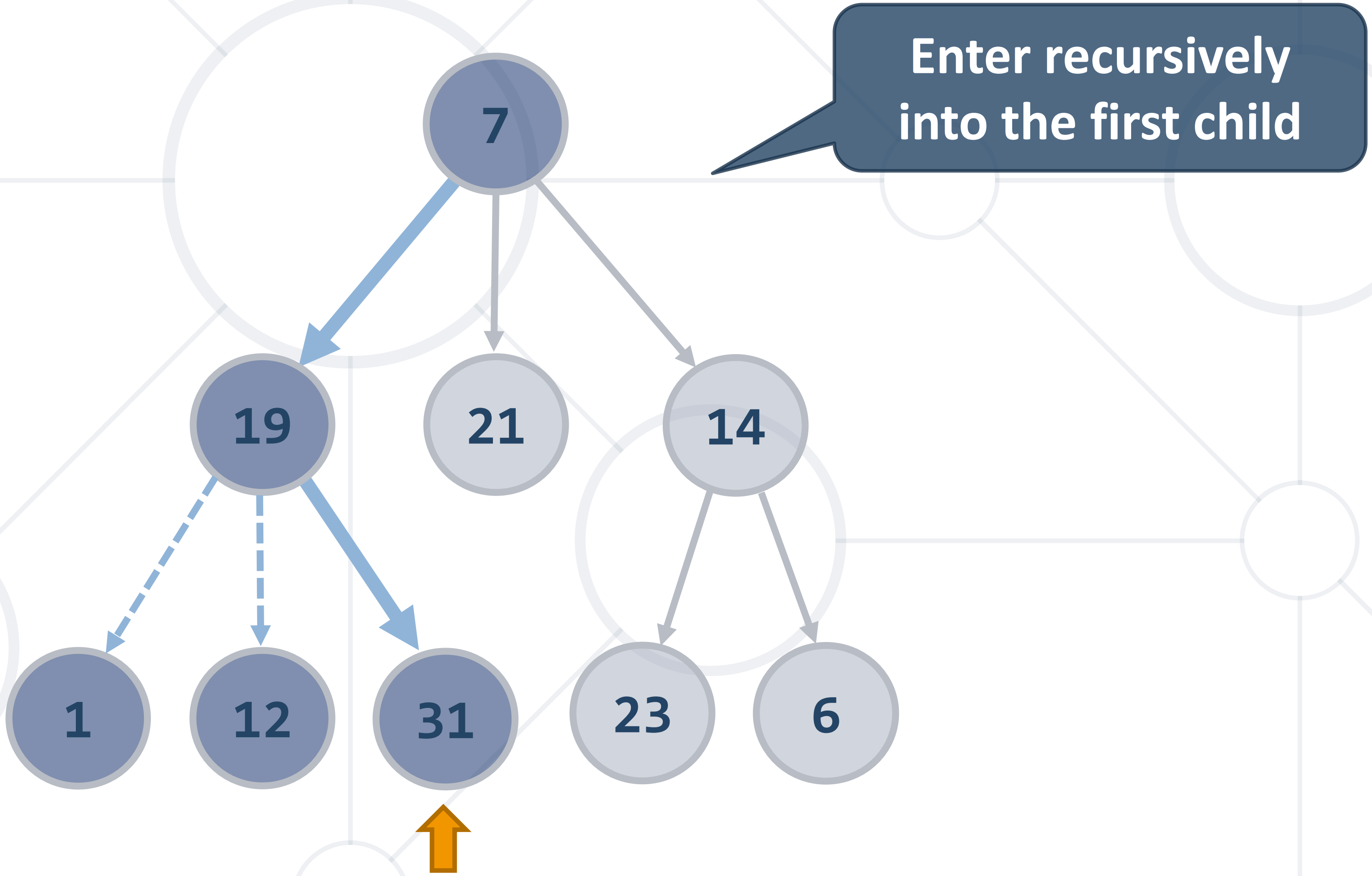
DFS in Action (Step 6)

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



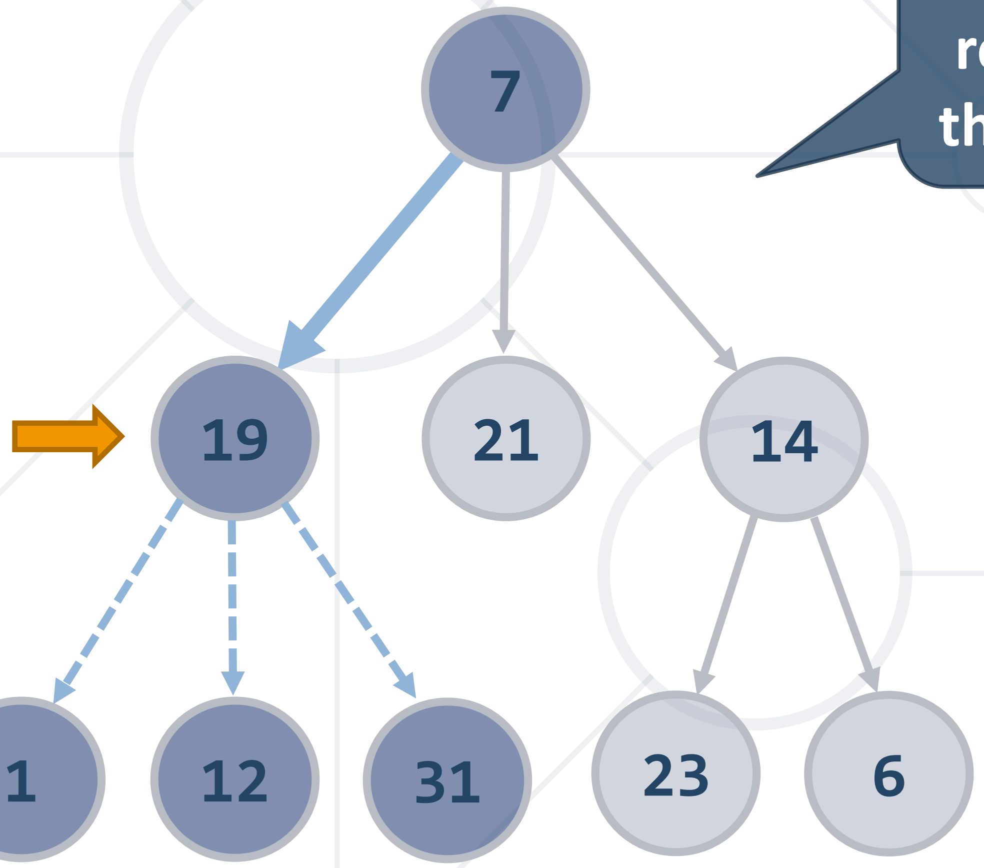
DFS in Action (Step 7)

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



DFS in Action (Step 8)

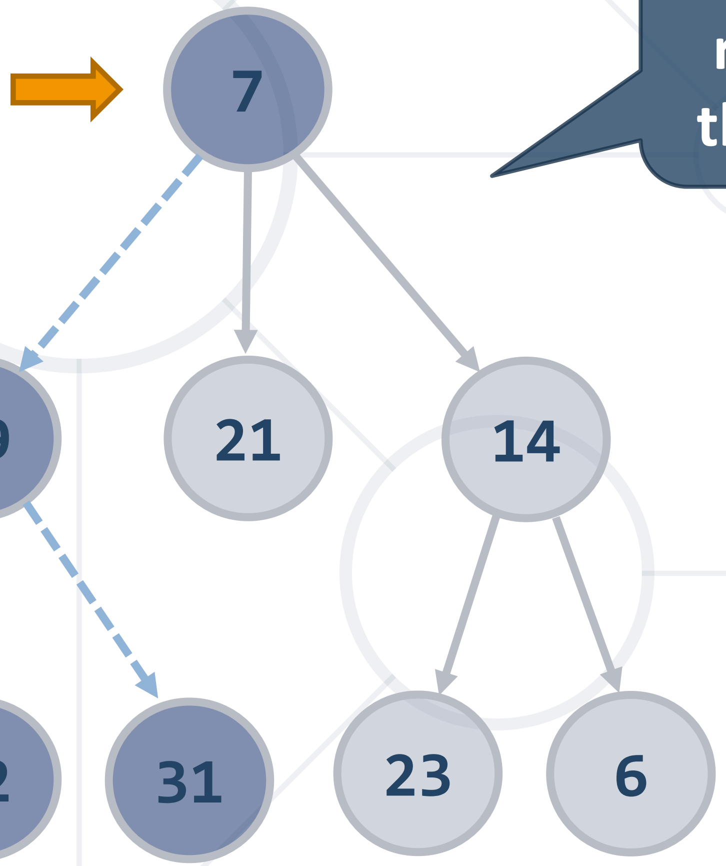
- Stack: 7, 19
- Output: 1, 12, 31



Return back from recursion and print the last visited node

DFS in Action (Step 9)

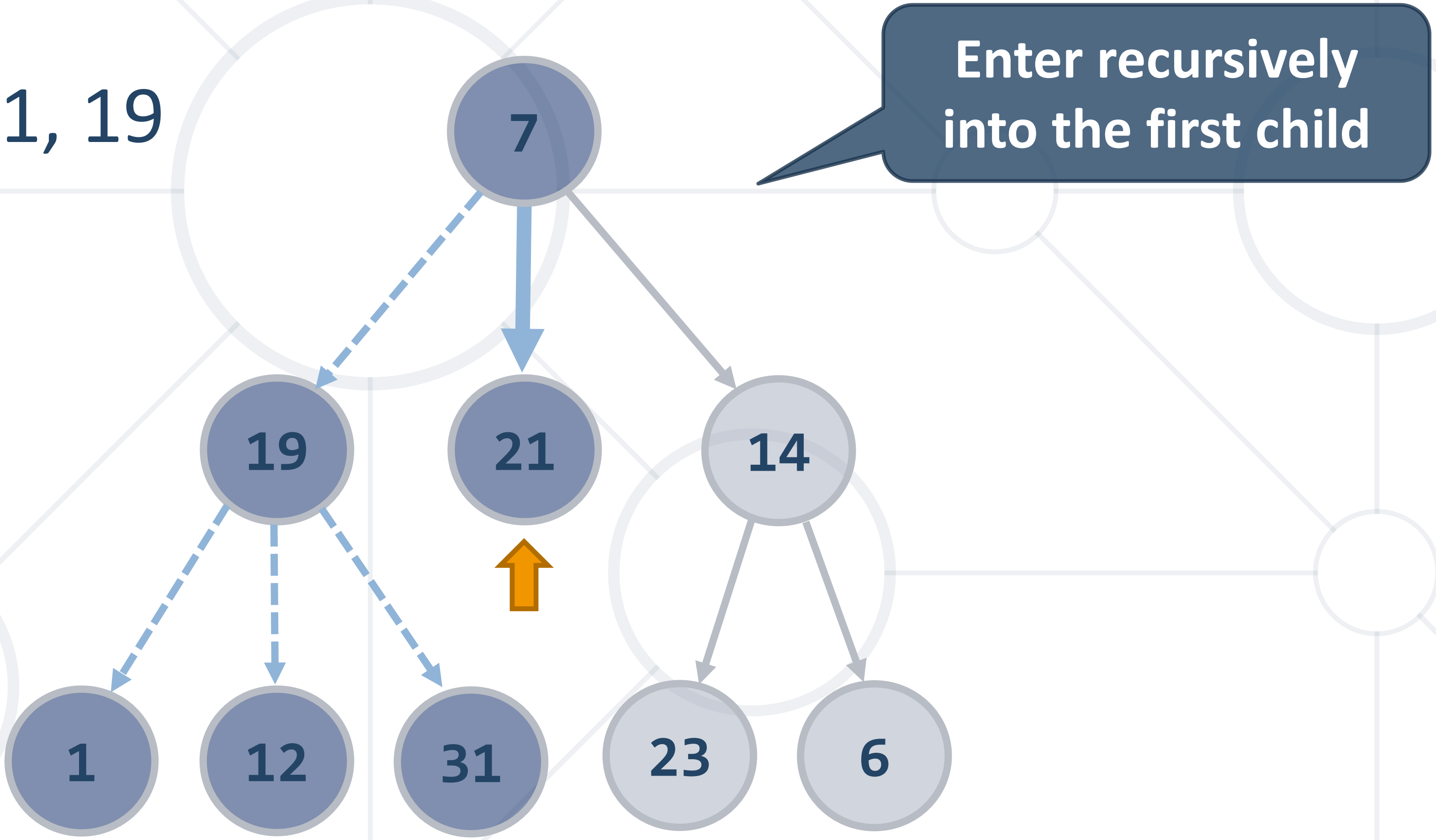
- Stack: 7
- Output: 1, 12, 31, 19



Return back from recursion and print the last visited node

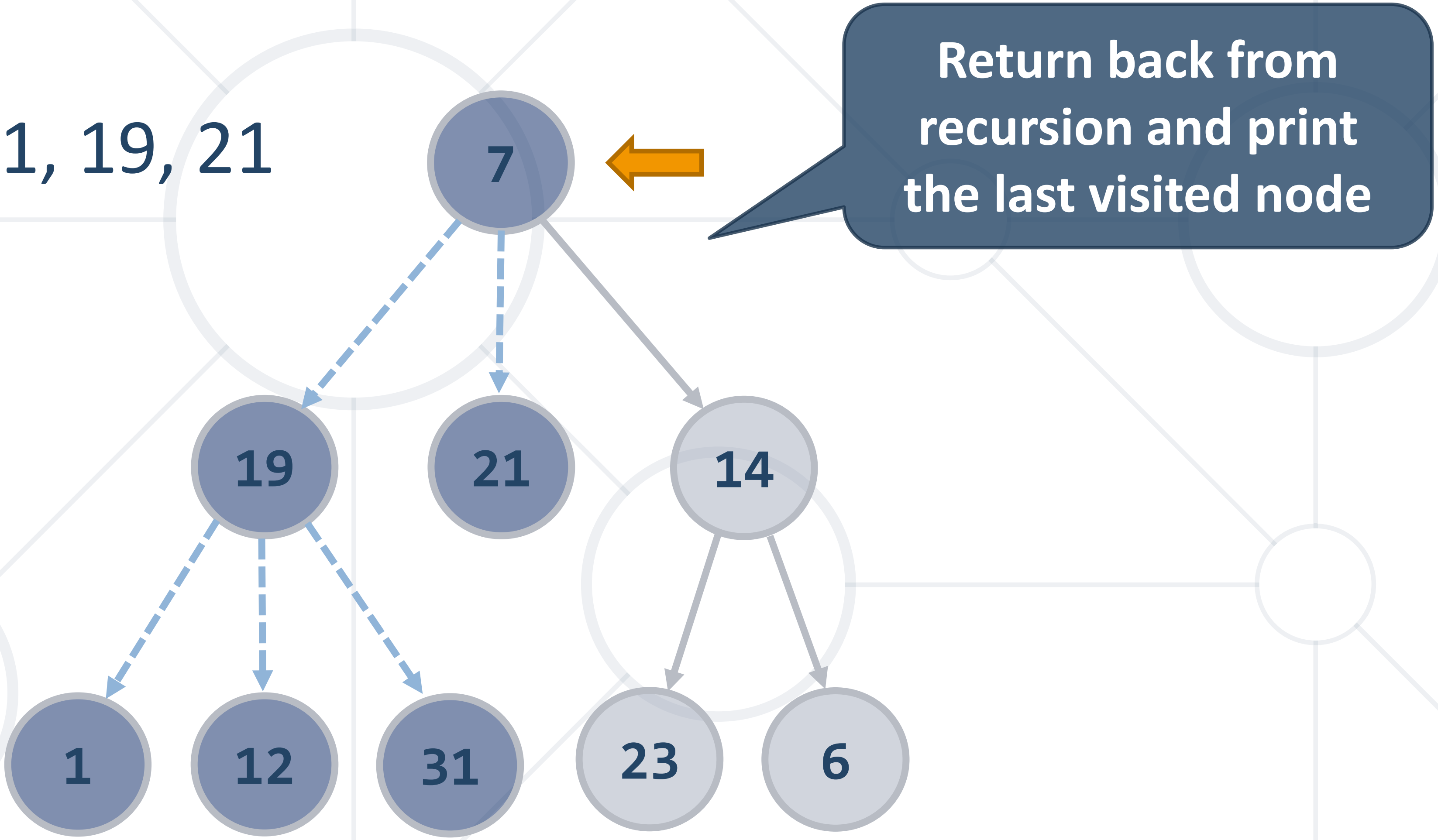
DFS in Action (Step 10)

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



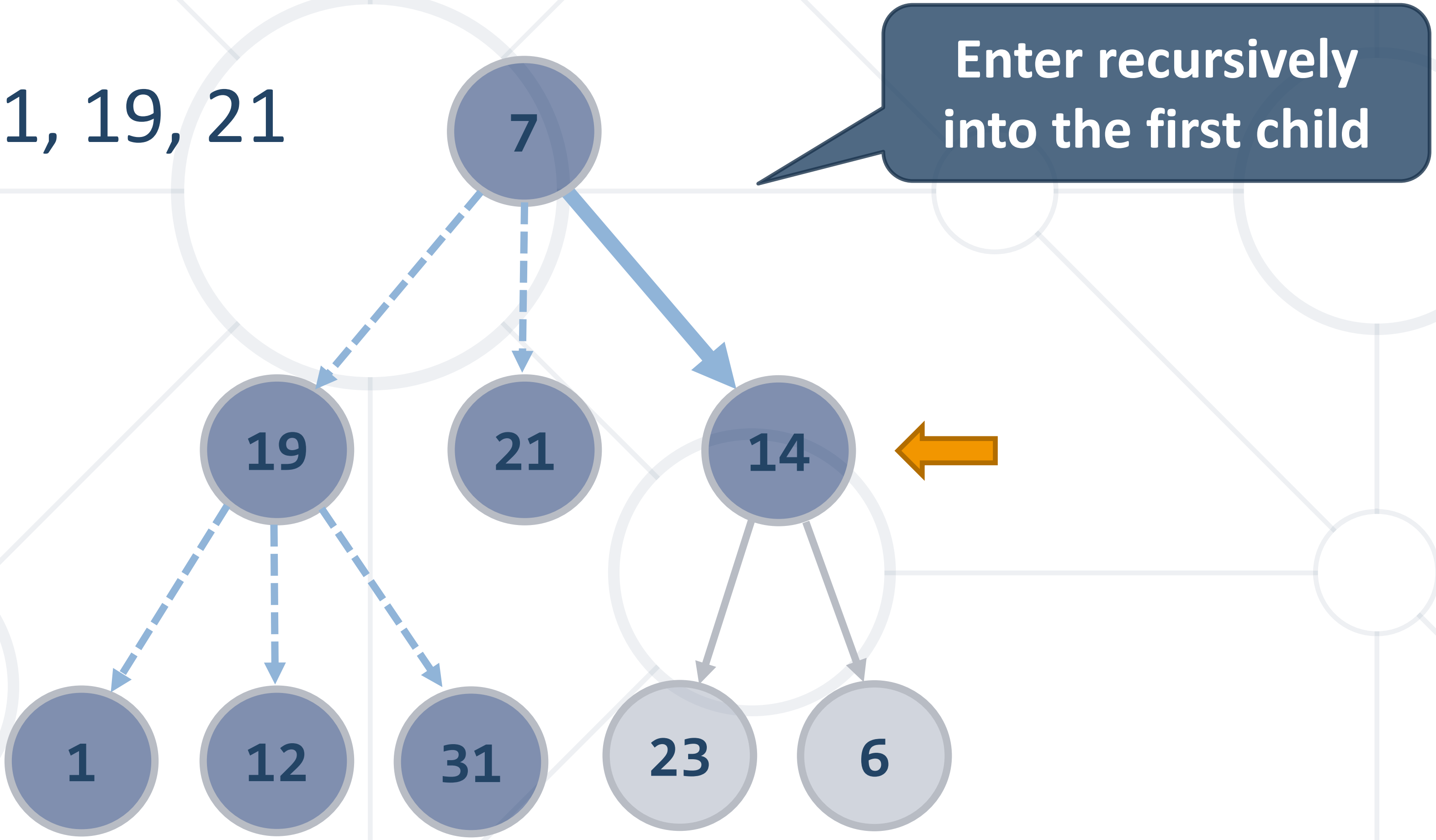
DFS in Action (Step 11)

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



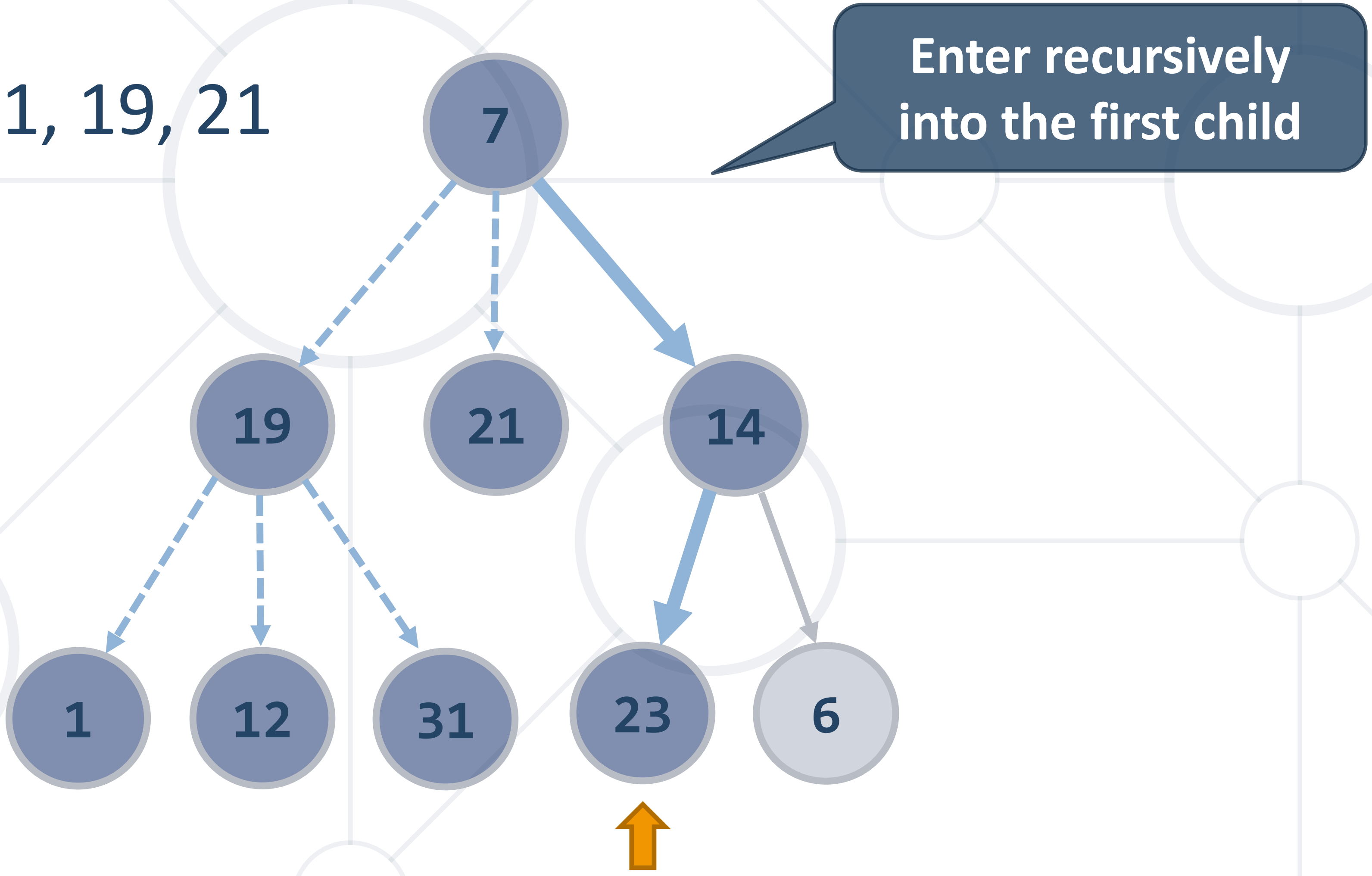
DFS in Action (Step 12)

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



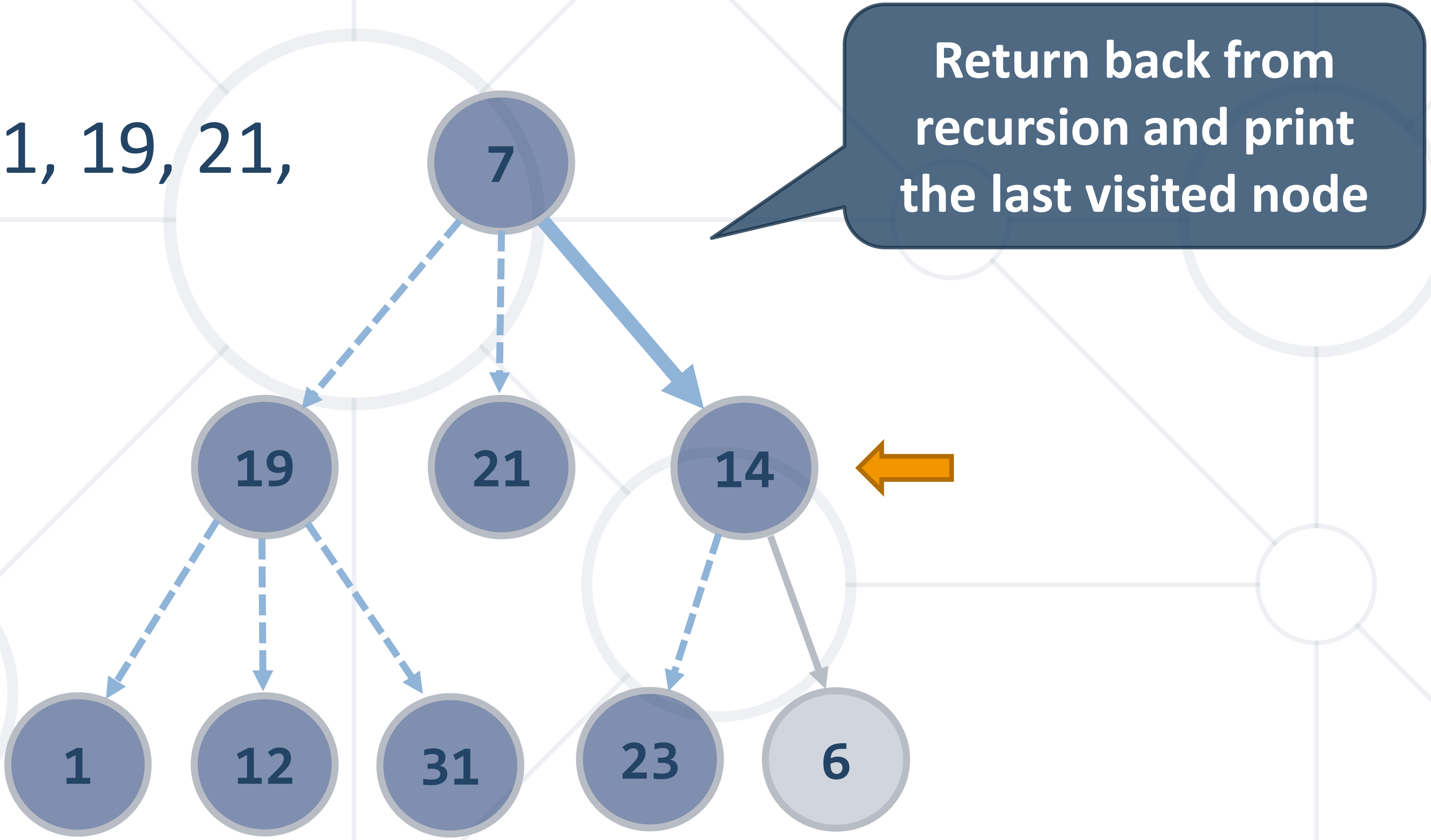
DFS in Action (Step 13)

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



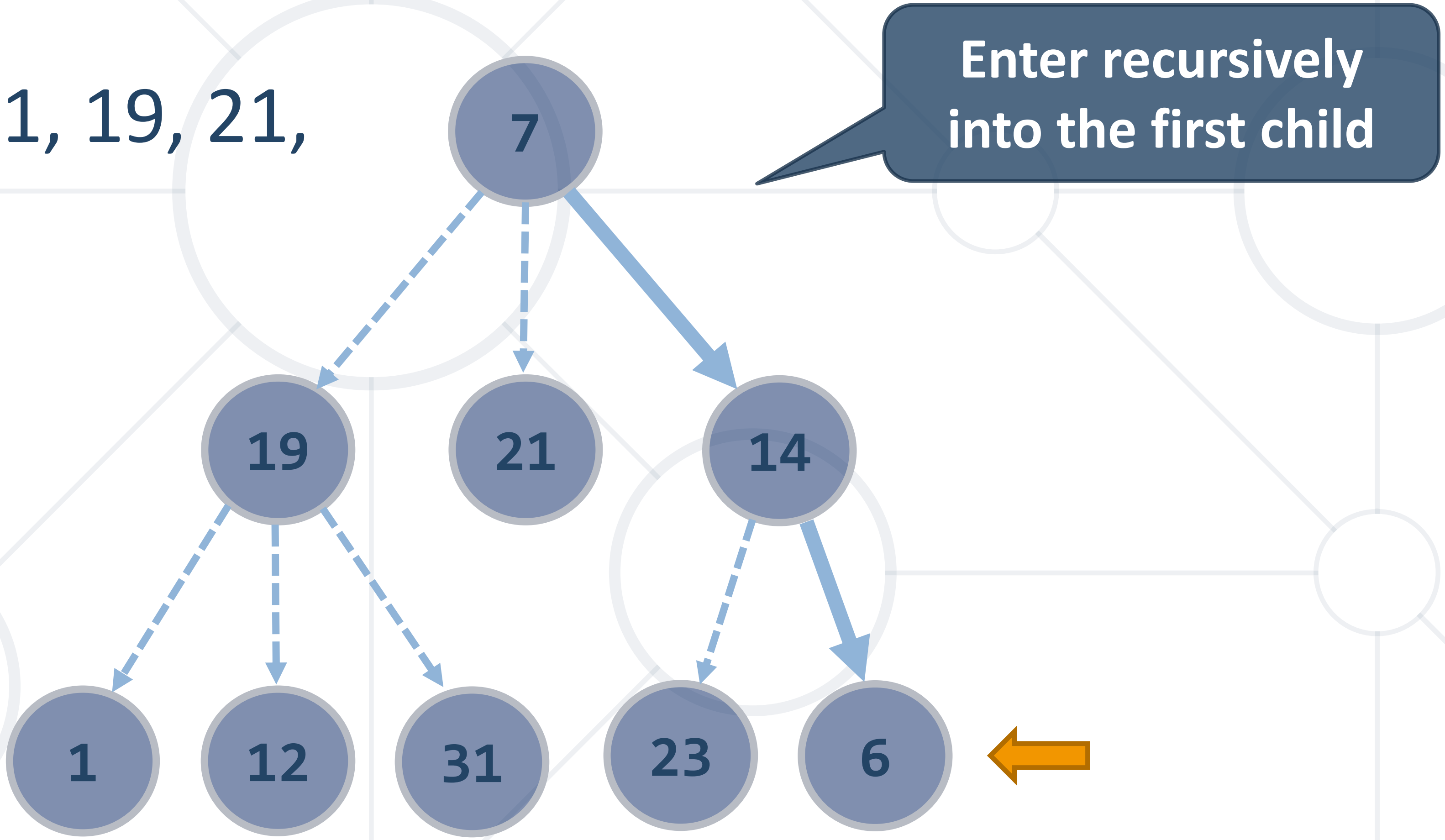
DFS in Action (Step 14)

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



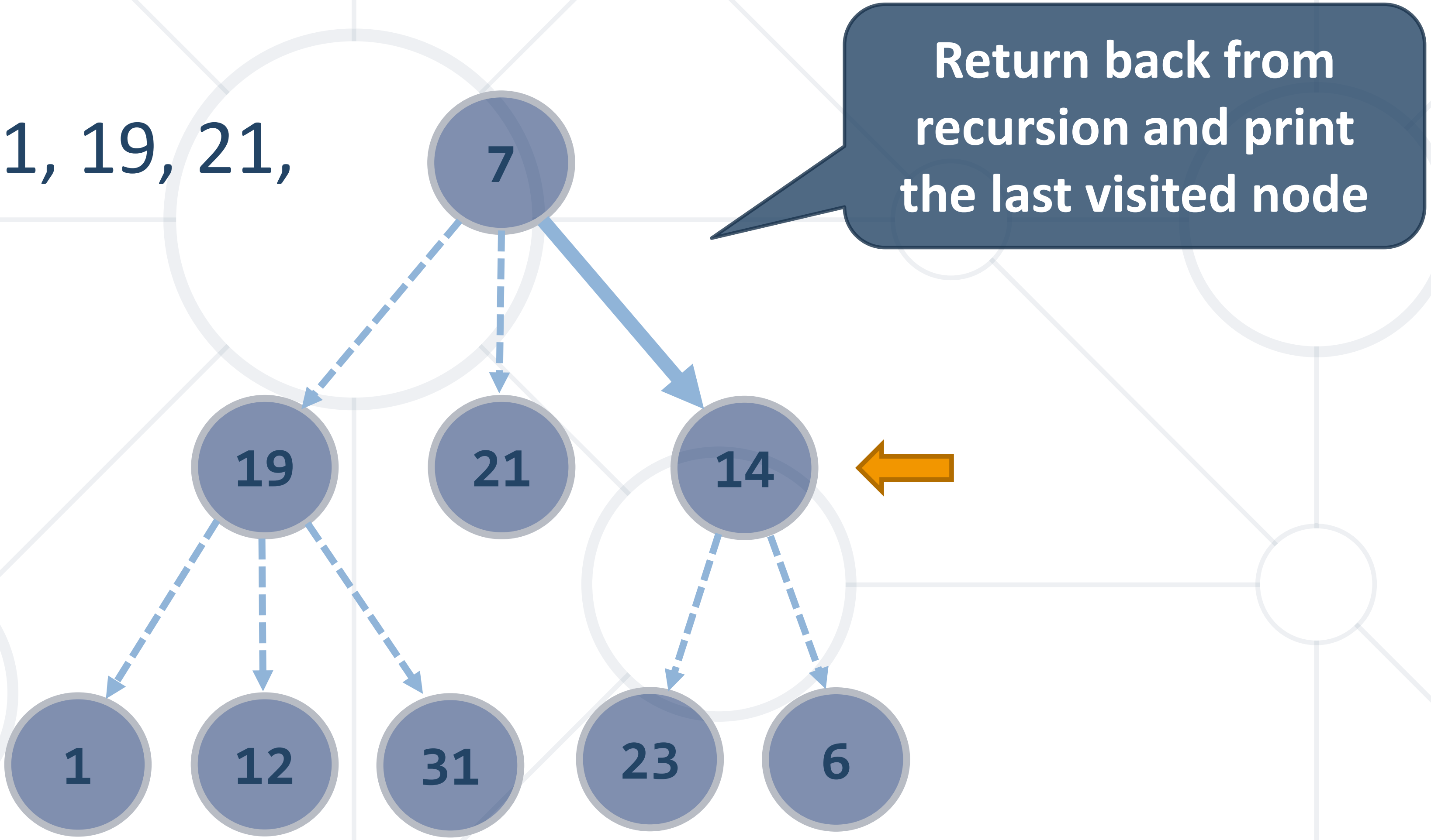
DFS in Action (Step 15)

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



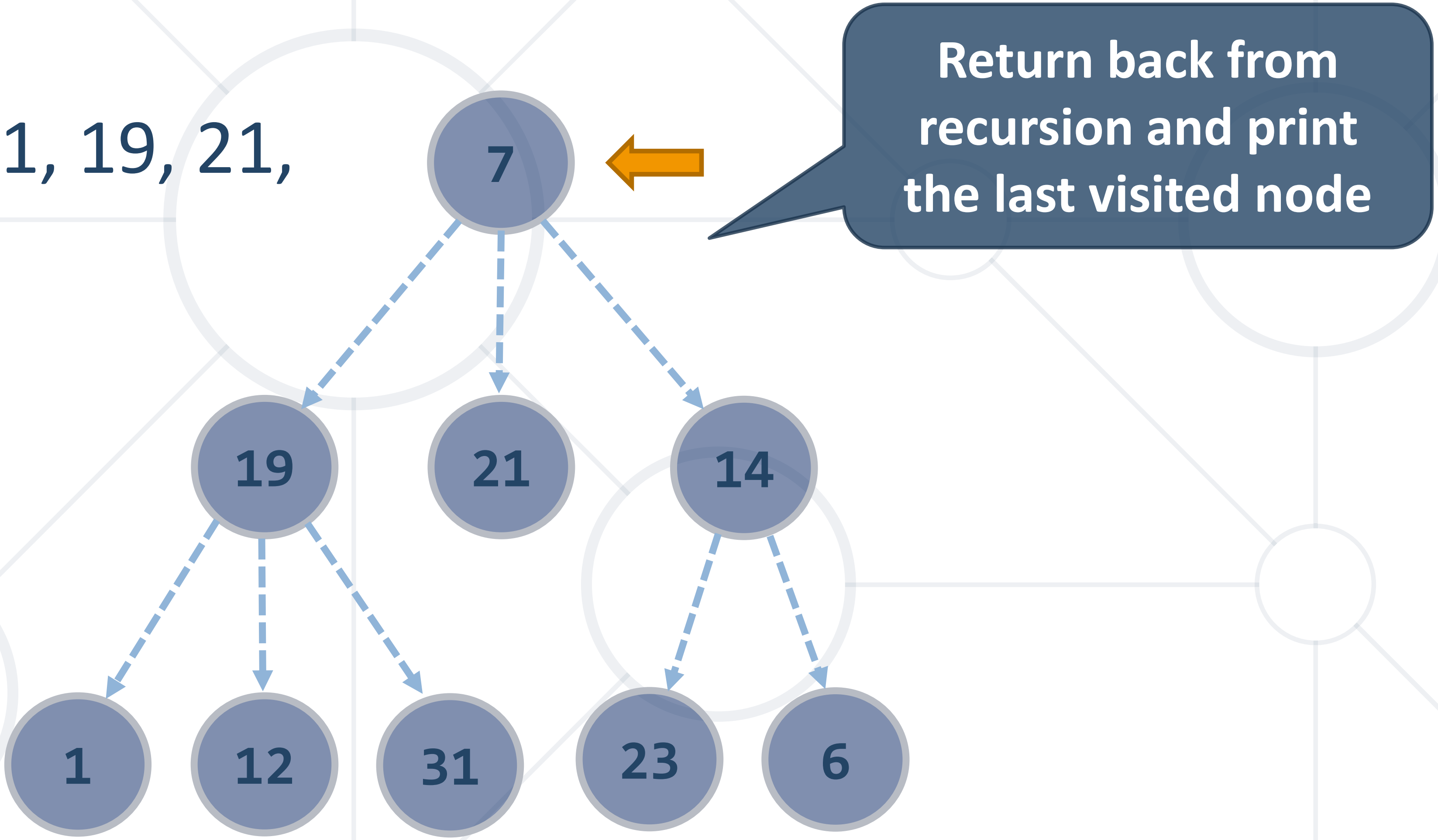
DFS in Action (Step 16)

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



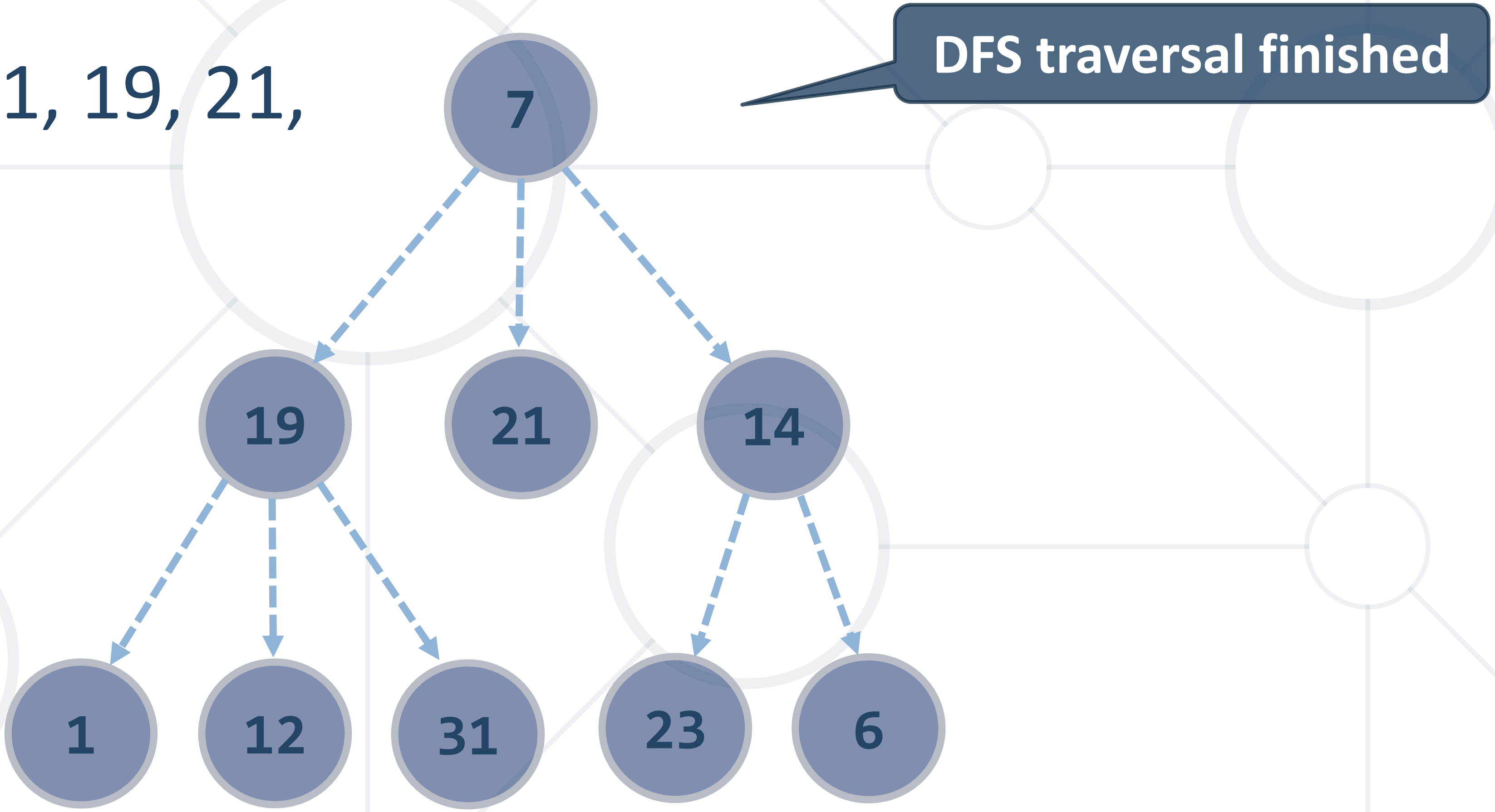
DFS in Action (Step 17)

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



DFS in Action (Step 18)

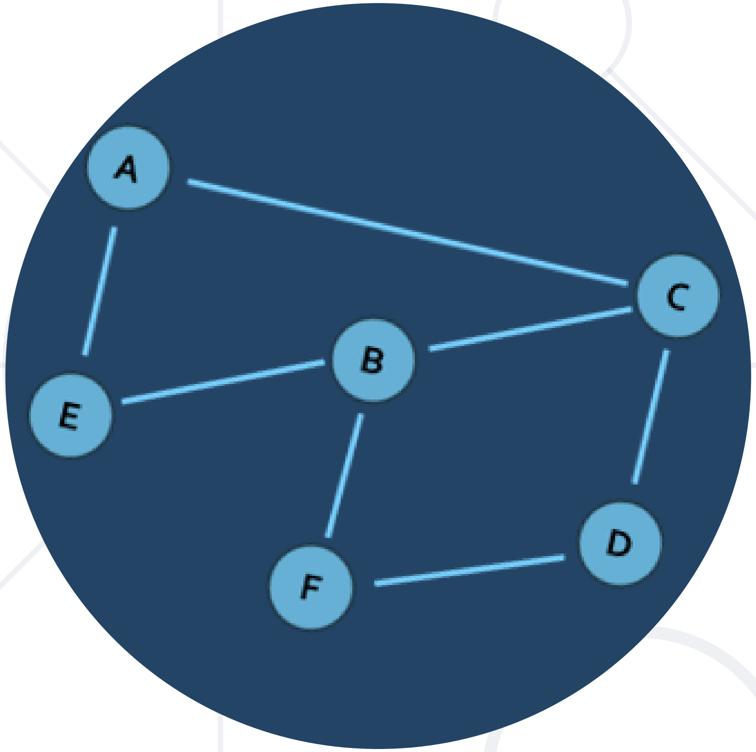
- Stack: (empty)
- Output: 1, 12, 31, 19, 21, 23, 6, 14, 7



DFS Example: Traverse Folders

```
private static void TraverseDir(DirectoryInfo dir, string spaces)
{
   Console.WriteLine(spaces + dir.FullName);
   DirectoryInfo[] children = dir.GetDirectories();
   foreach (DirectoryInfo child in children)
      TraverseDir(child, spaces + "  ");
}
static void TraverseDir(string directoryPath)
{
   TraverseDir(new DirectoryInfo(directoryPath), string.Empty);
}
static void Main()
{
   TraverseDir(@"C:\Windows\assembly");
}
```

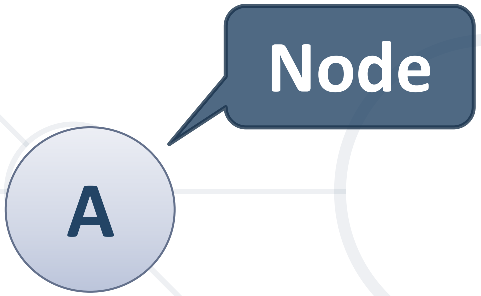
Use recursion to traverse folders

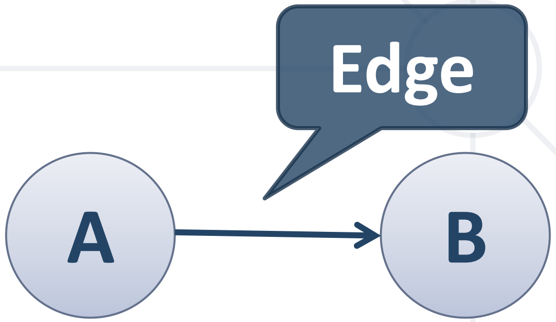


Graphs

Graph Definitions

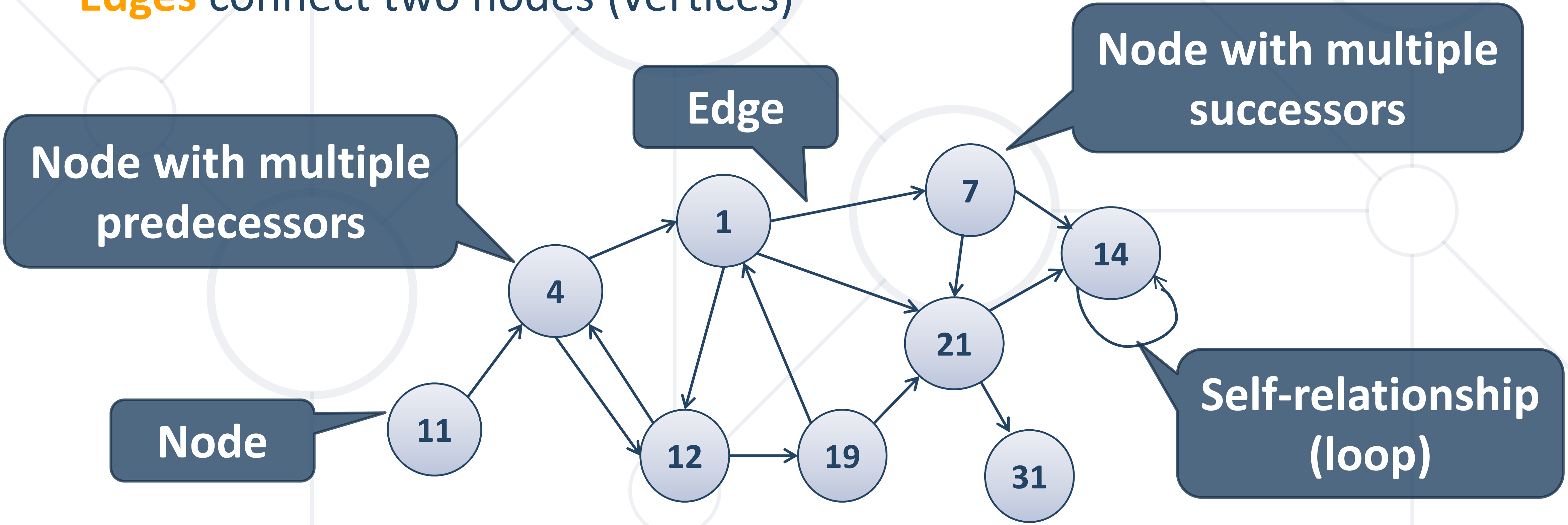
- **Node** (vertex)
  - Element of a graph
  - Can have name / value
  - Keeps a list of adjacent nodes
- **Edge**
  - Connection between two nodes
  - Can be directed / undirected
  - Can be weighted / unweighted
  - Can have name / value





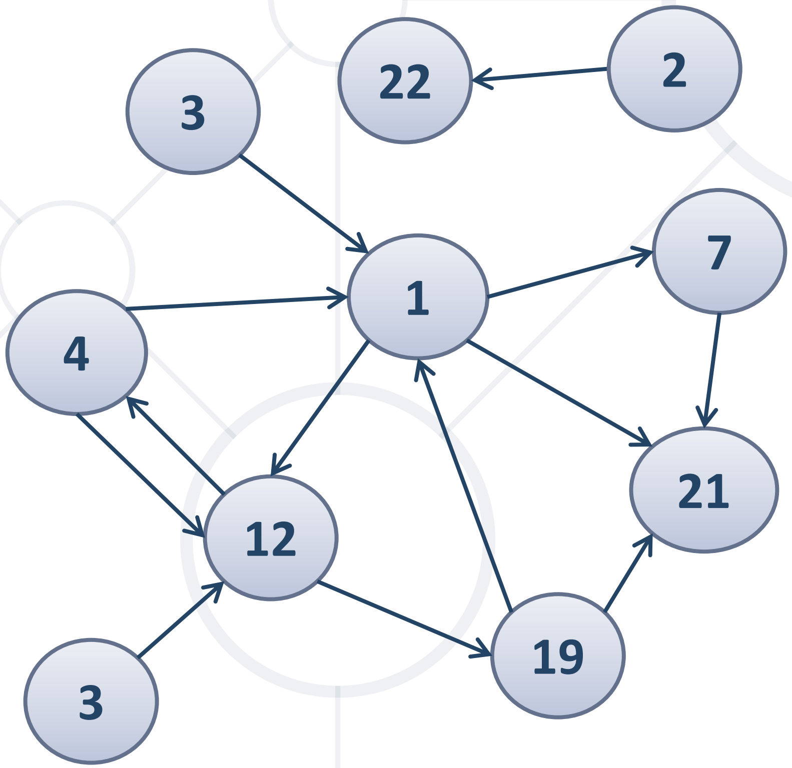
Graph Data Structure

- **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)

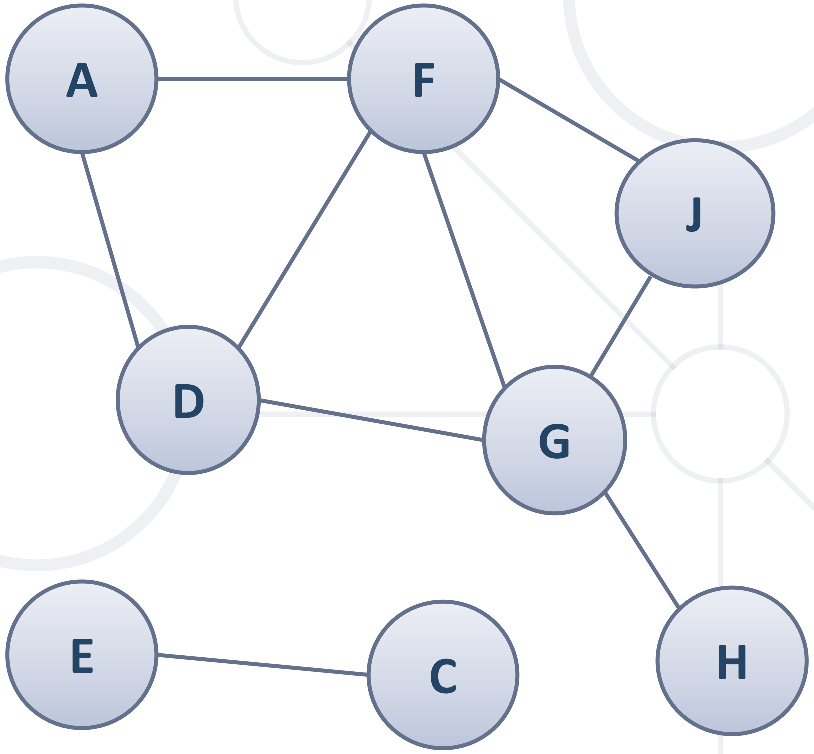


Graph Definitions: Directed / Undirected

- **Directed graph**
  - Edges have direction

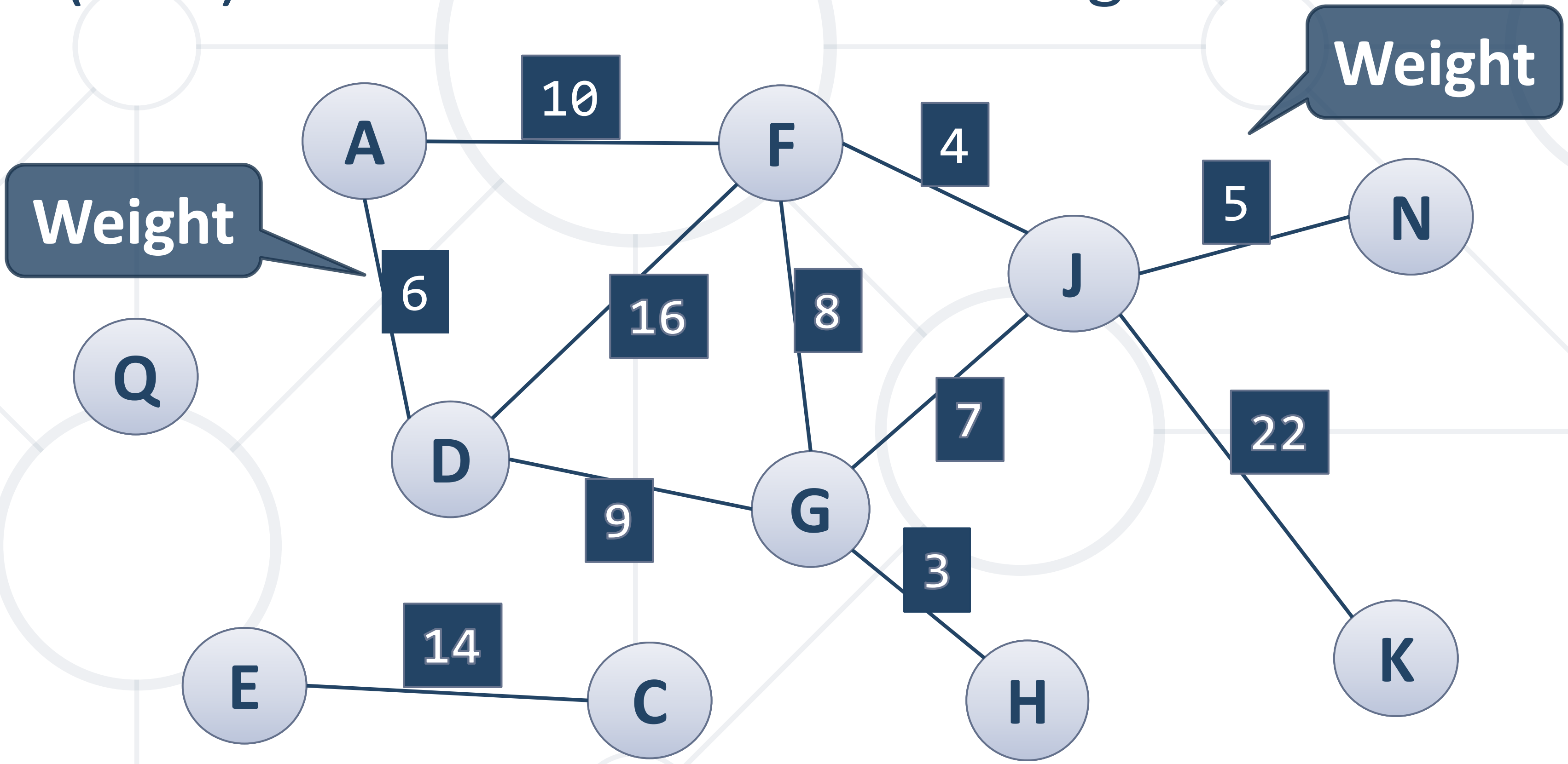


- **Undirected graph**
  - Undirected edges



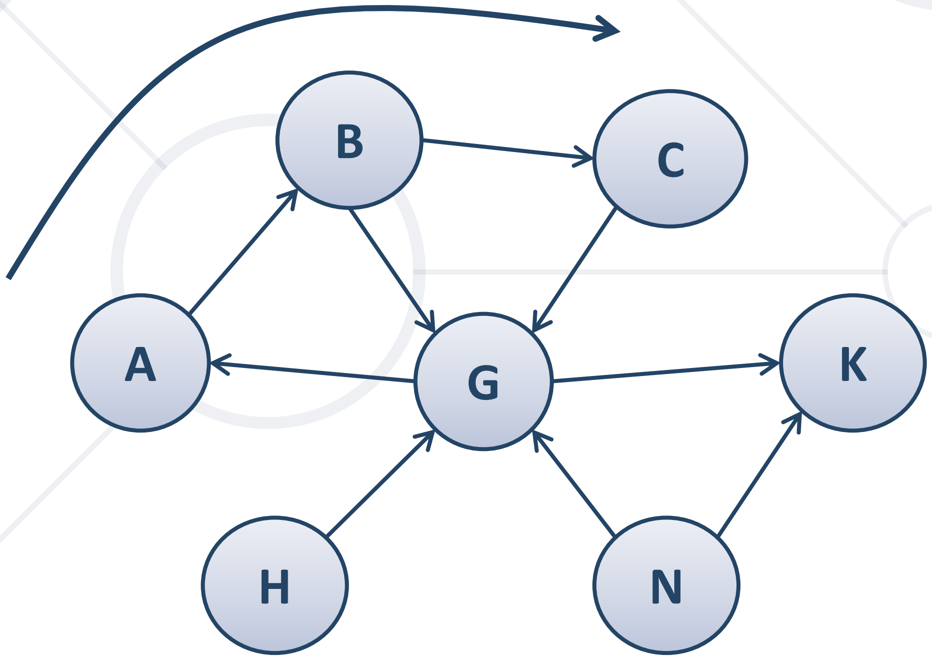
Graph Definitions: Weighted Graph

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



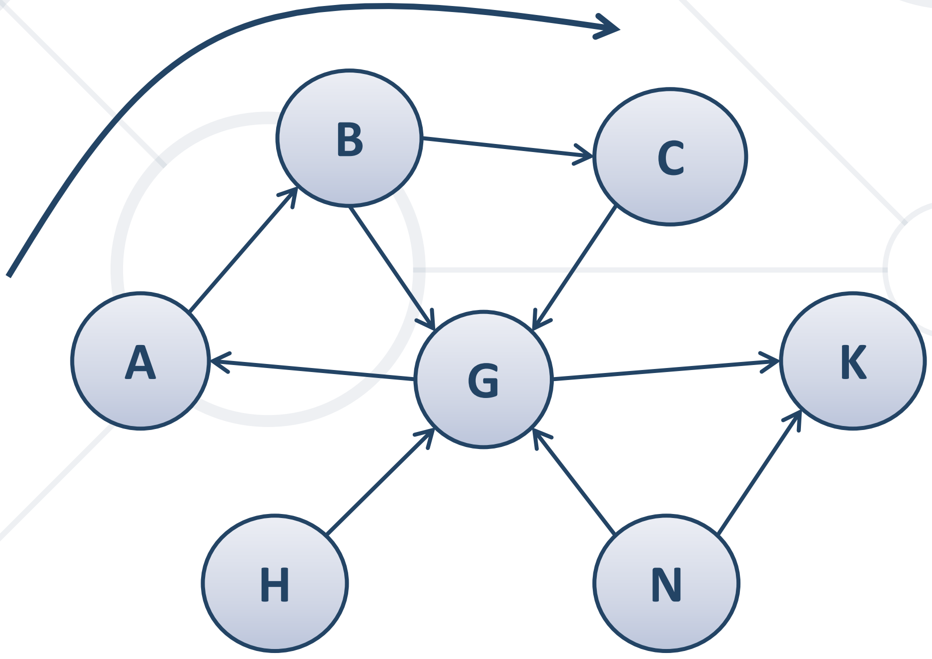
Graph Definitions: Path

- **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



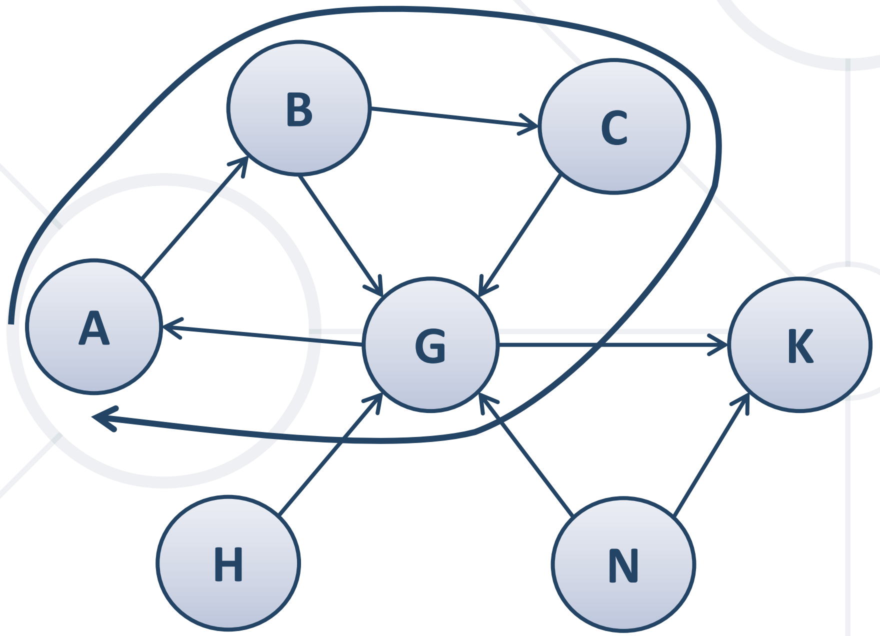
Graph Definitions: Directed Path

- **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: Cycles

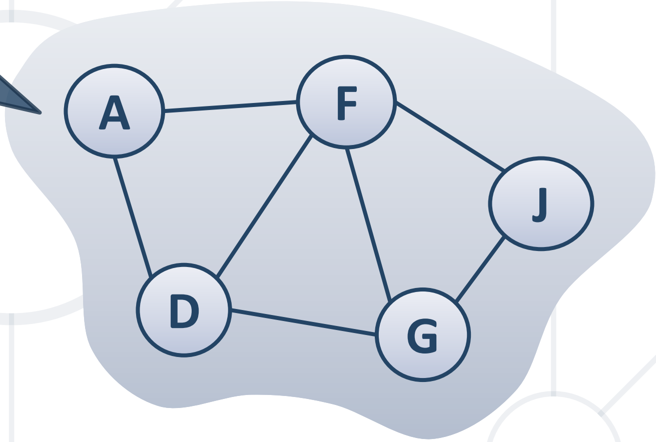
- **Cycle**
  - Path that ends back at the starting node
  - Example of cycle: A, B, C, G, A
- **Simple path**
  - No cycles in path
- **Acyclic graph**
  - Graph with no cycles
  - Acyclic undirected graphs are trees



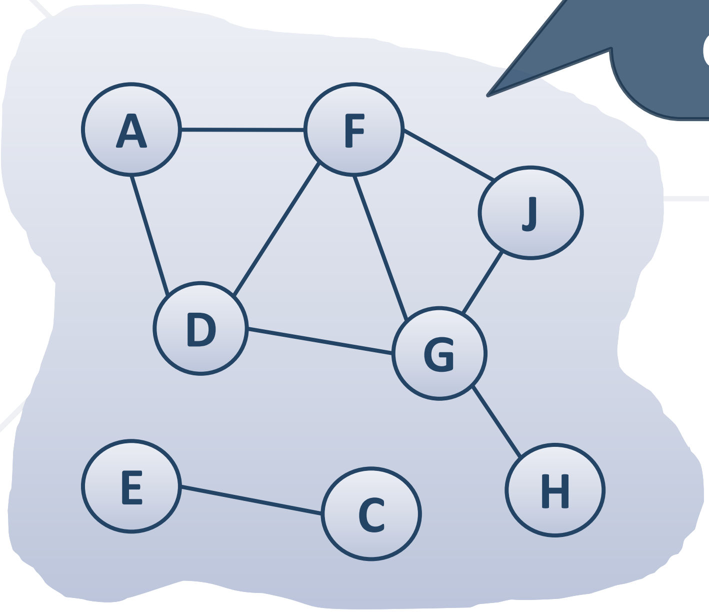
Graph Definitions: Connectivity

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

Connected graph



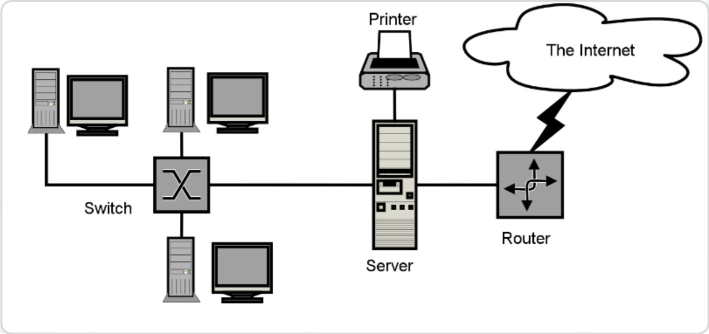
Disconnected graph holding two connected components

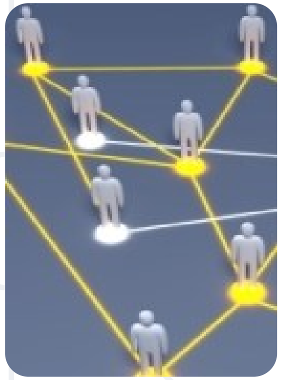


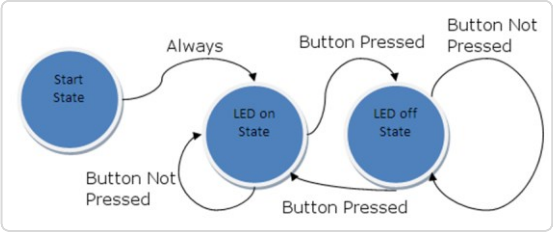
Graphs and Their Applications

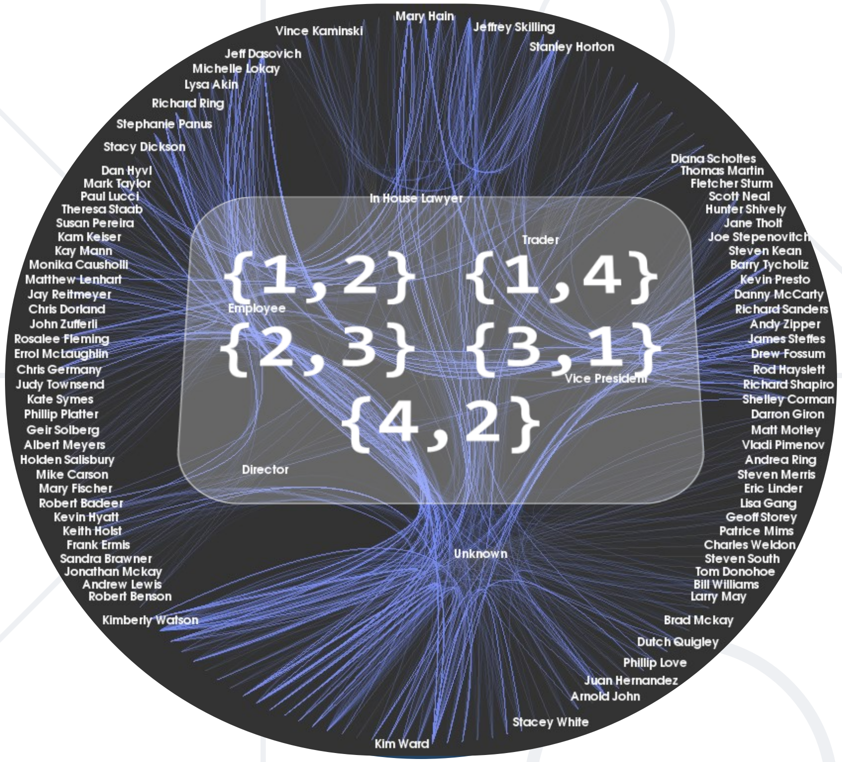
Software University

- Graphs have many real-world applications
  - Modeling a **computer network**
    - Routes are 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







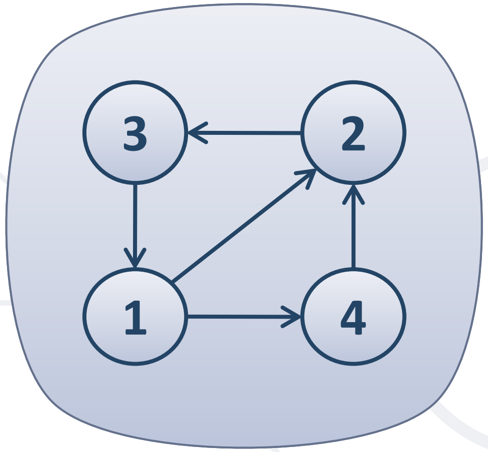


Classic and OOP Ways

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

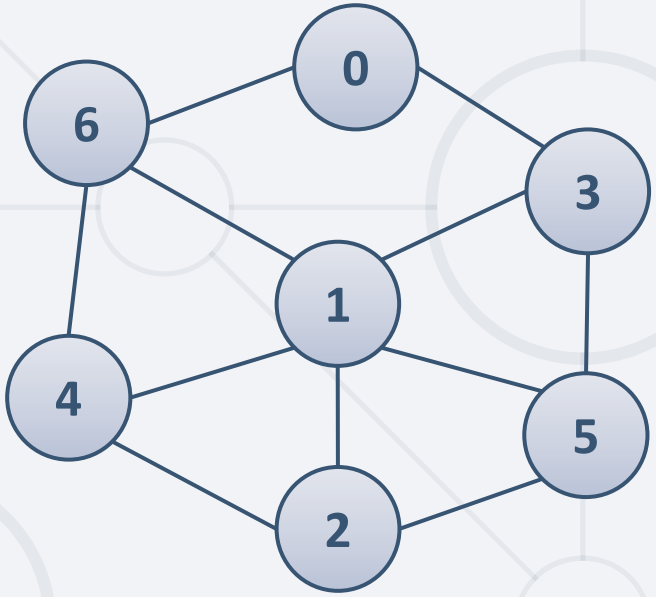
	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	1	0	0	0
4	0	1	0	0

- **List of edges**

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

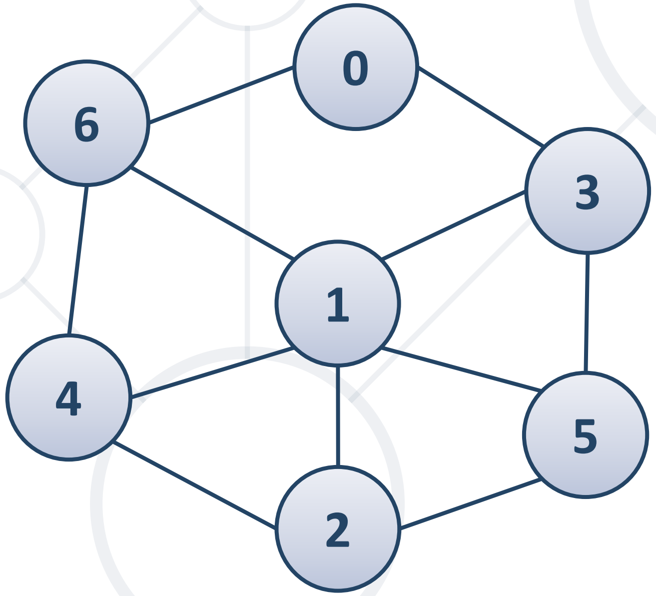
Graph Representation: Adjacency List

```
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}
};

g[3].Add(6); // Add an edge { 3 → 6 }

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

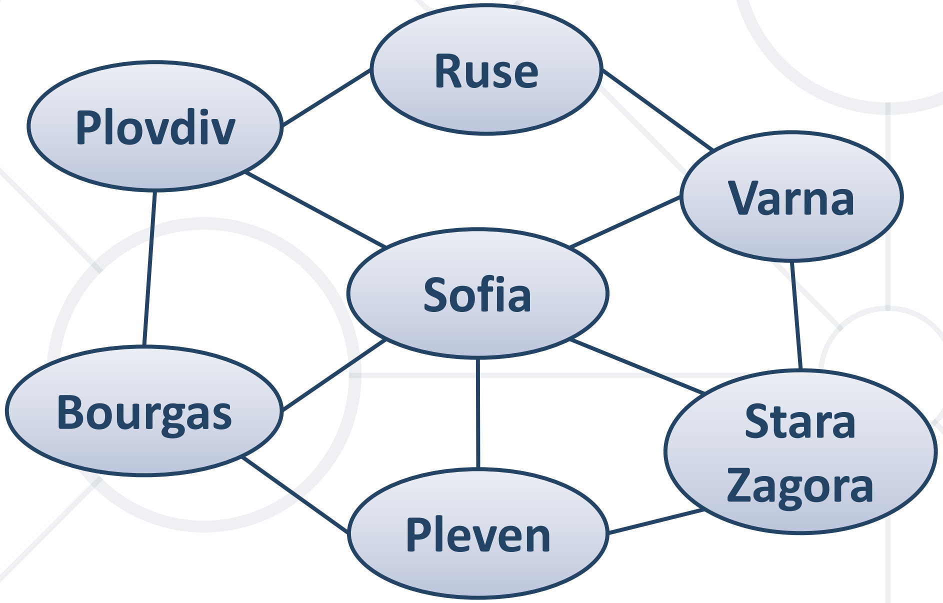


Numbering Graph Nodes

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



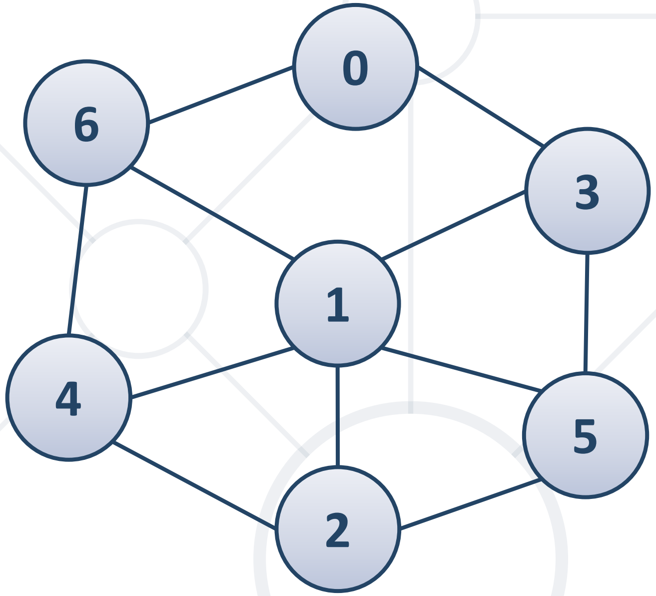
Graph of **numbered nodes**: [0…6]



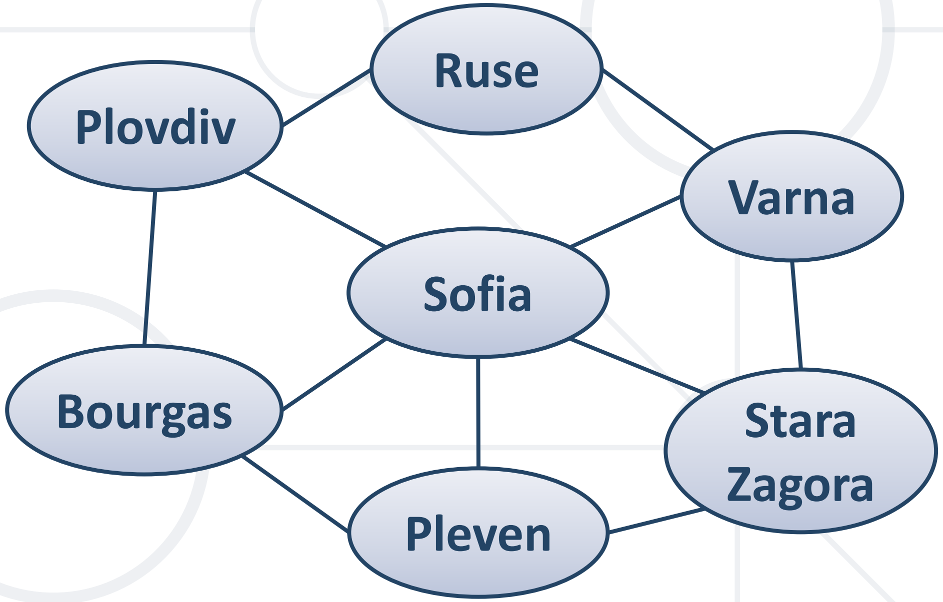
Graph of **named nodes**

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

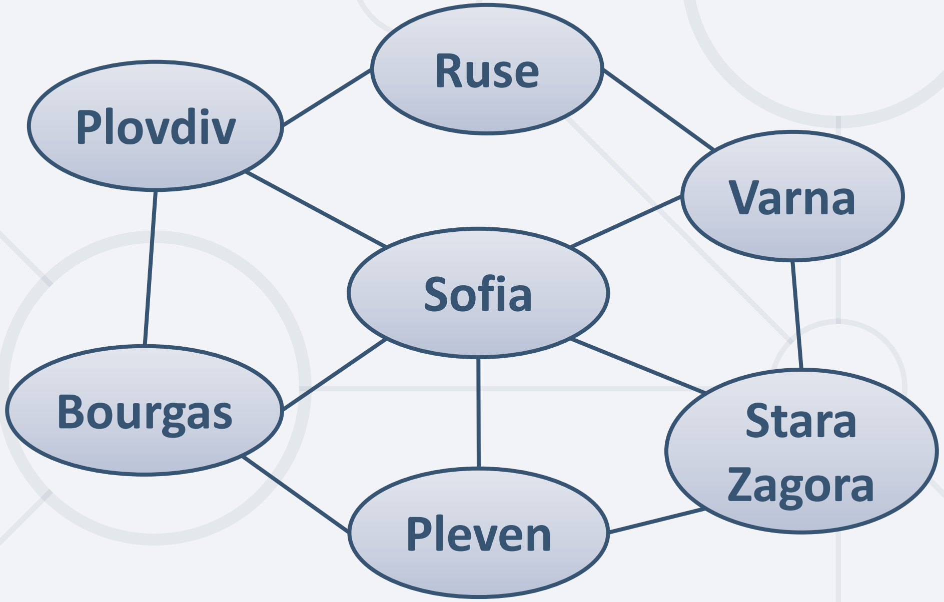



```
var g =
  new List<int>[n]
```

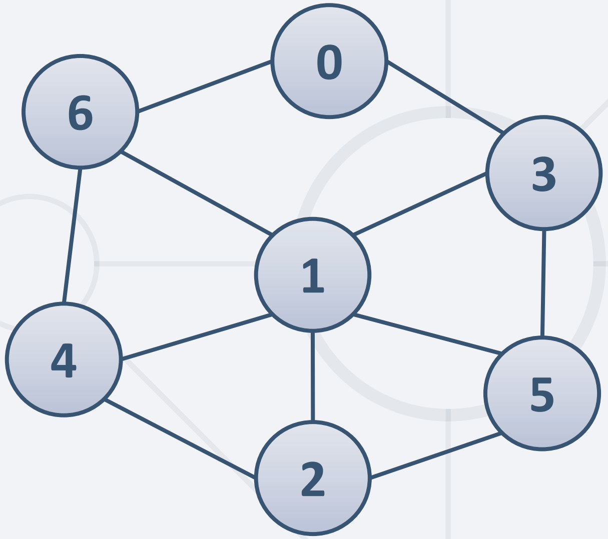
```
var g = new Dictionary<
  string, List<string>>
```

Graph of Named Nodes – Example

```
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", … });
```

Summary

Software University

- **Trees** are recursive data structures
  - A tree is a node holding a set of children (which are also nodes)
  - Edges connect nodes
- **DFS** traversal → children first
- **BFS** traversal → root first

Summary

- Representing graphs in the memory
  - **Adjacency list** holding each node's children
  - **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

Software University

Questions?

SoftUni

Software University

SoftUni Svetlina

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

License

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

Trainings @ Software University (SoftUni)

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