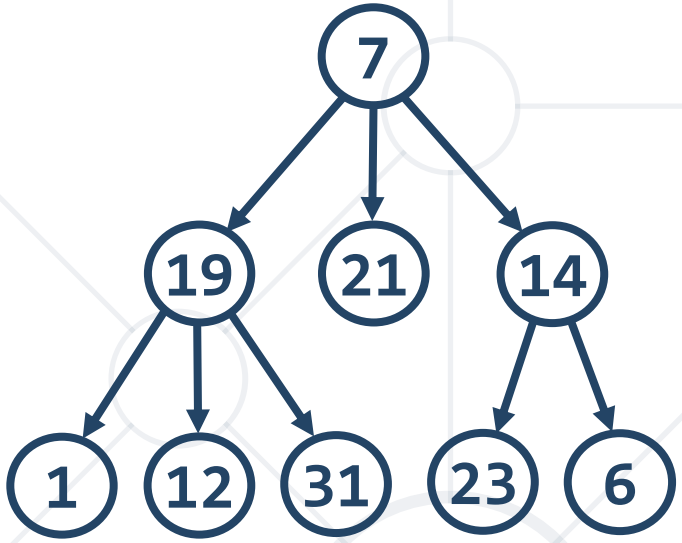
Trees Representation and Traversal (BFS, DFS)

Trees Related Terminology and Traversal Algorithms



SoftUni Team

Technical Trainers

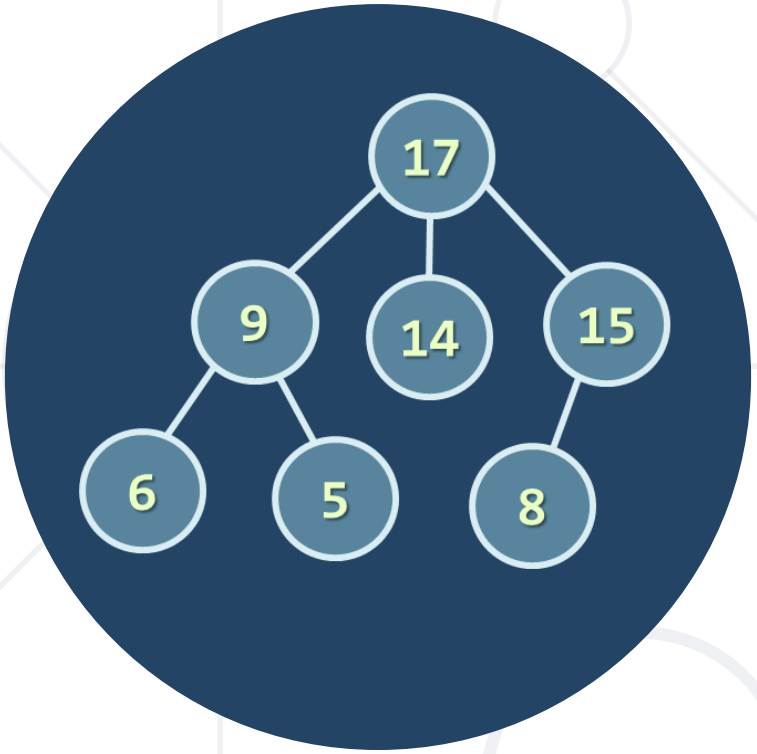
Software University

SoftUni

Software University

https://softuni.bg

Table of Contents



Why Trees?

Definition and Use Cases of Trees

Summary

- So far, we have learned how to implement linear data structures like: List, Queue, Stack, LinkedList etc…
- We did a great job and learned how to take the best complexity we can, **was that enough**?
- Actually more of the operations we want to do like **search**, **insert** or **remove** are **linear** for **unordered** structures (sometimes we can do O(1)) but **not for search**

Linear Data Structures - Types

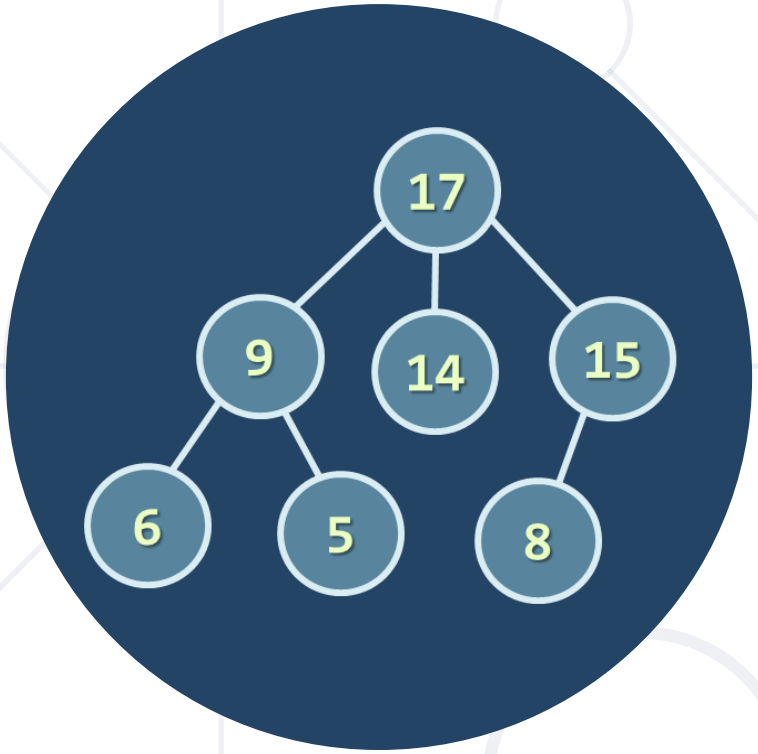
- Atop an **array**
  - Ability to **add elements with O(1)**
  - Removing and searching with **O(n)**
  - Inside sorted arrays remove, search - **O(log(n))**, but sorting is needed on every new element added
- By using **Node** implementation
  - Ability to **add and remove with O(1)**
  - Every other **operation is O(n)**
  - Even sorted **O(log(n))** is not achievable. But why?

Why Trees?

- We want not only to store data **add** or **remove** elements in efficient manner but also to **search** for elements, but **can** we do better than **O(n)**?
- Lets try to get **down** to **O(log(n))** by using **trees** and see if we can

Other Tree Benefits

- By learning how to work with trees you **actually** learn how to **work with**:
  - **Hierarchical** structures like: file system, project structures and code branching, NoSQL data storage etc…
  - **Markup** languages:
    - HTML
    - XML
- **DFS** and **BFS** algorithms



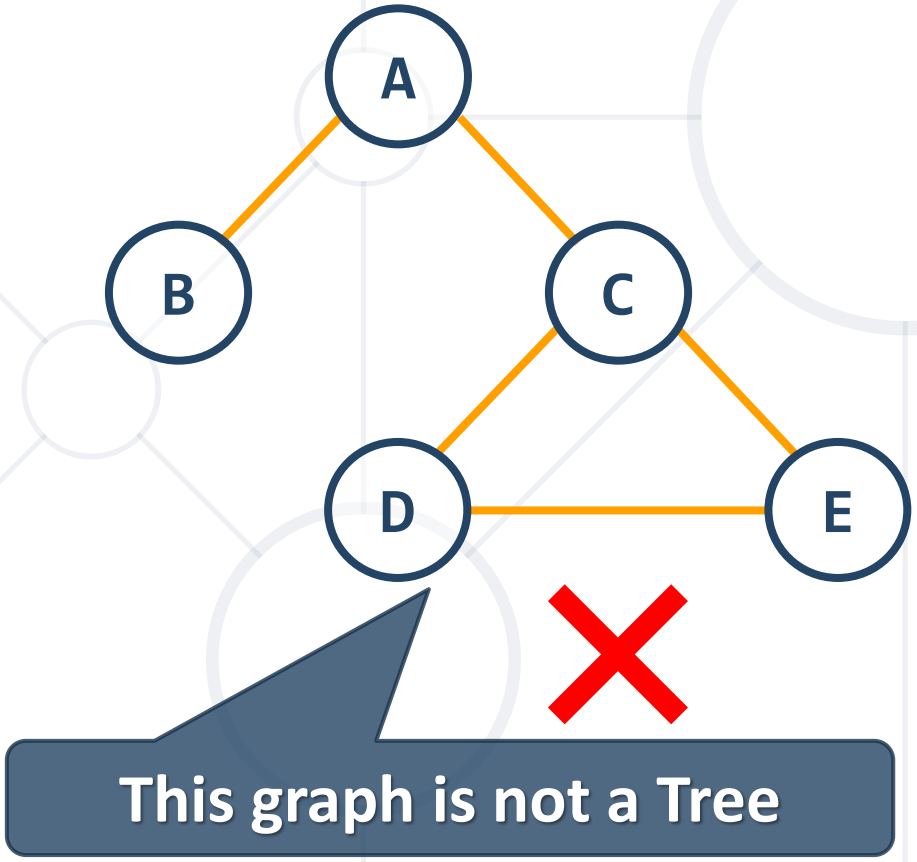
Trees and Related Terminology

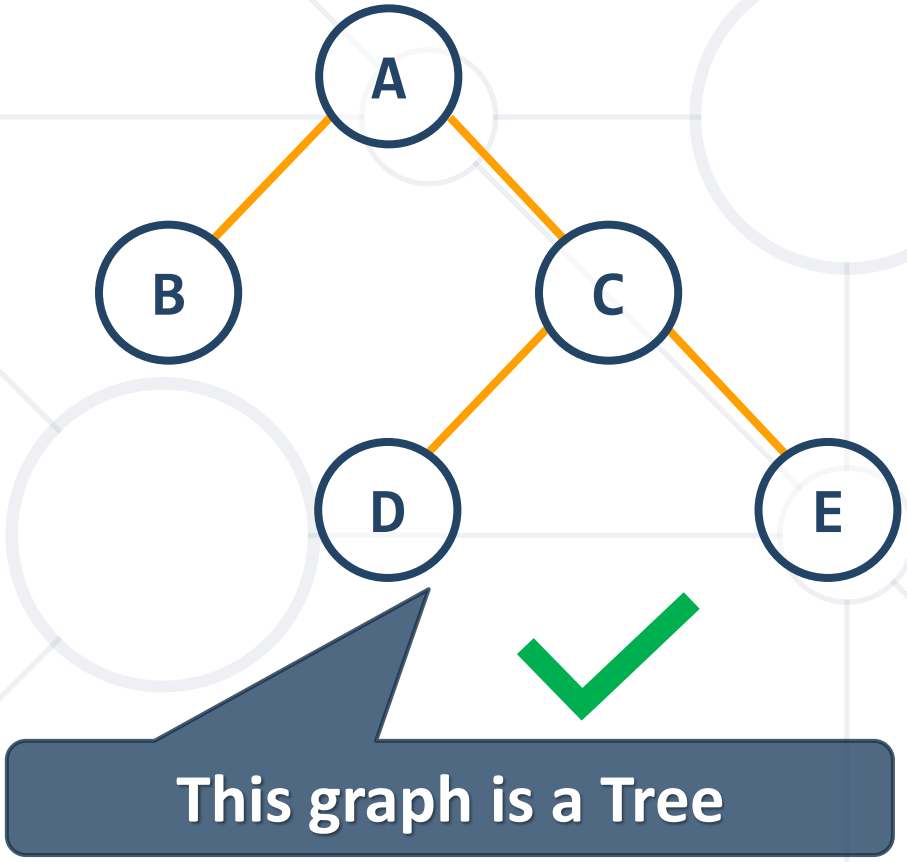
Node, Edge, Root, Etc.

Tree Definition

- Tree is a **non-linear** data structure which organizes data in a **hierarchical** structure, and this is a recursive definition.
- A tree is a **connected graph** without any circuits.
- If in a **graph**, there is one and **only one path between every pair of vertices**, then the graph is called as a tree.

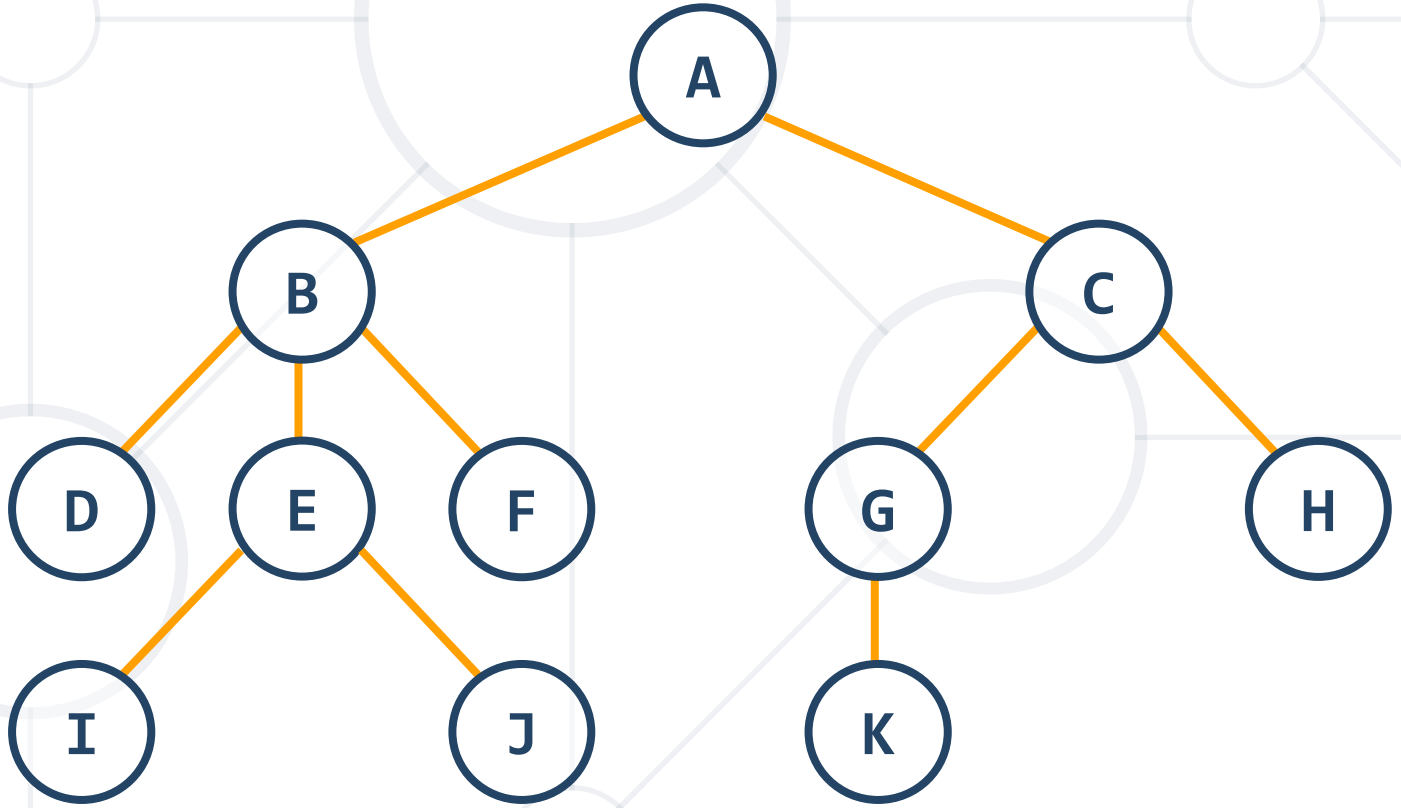
Tree Definition - Example





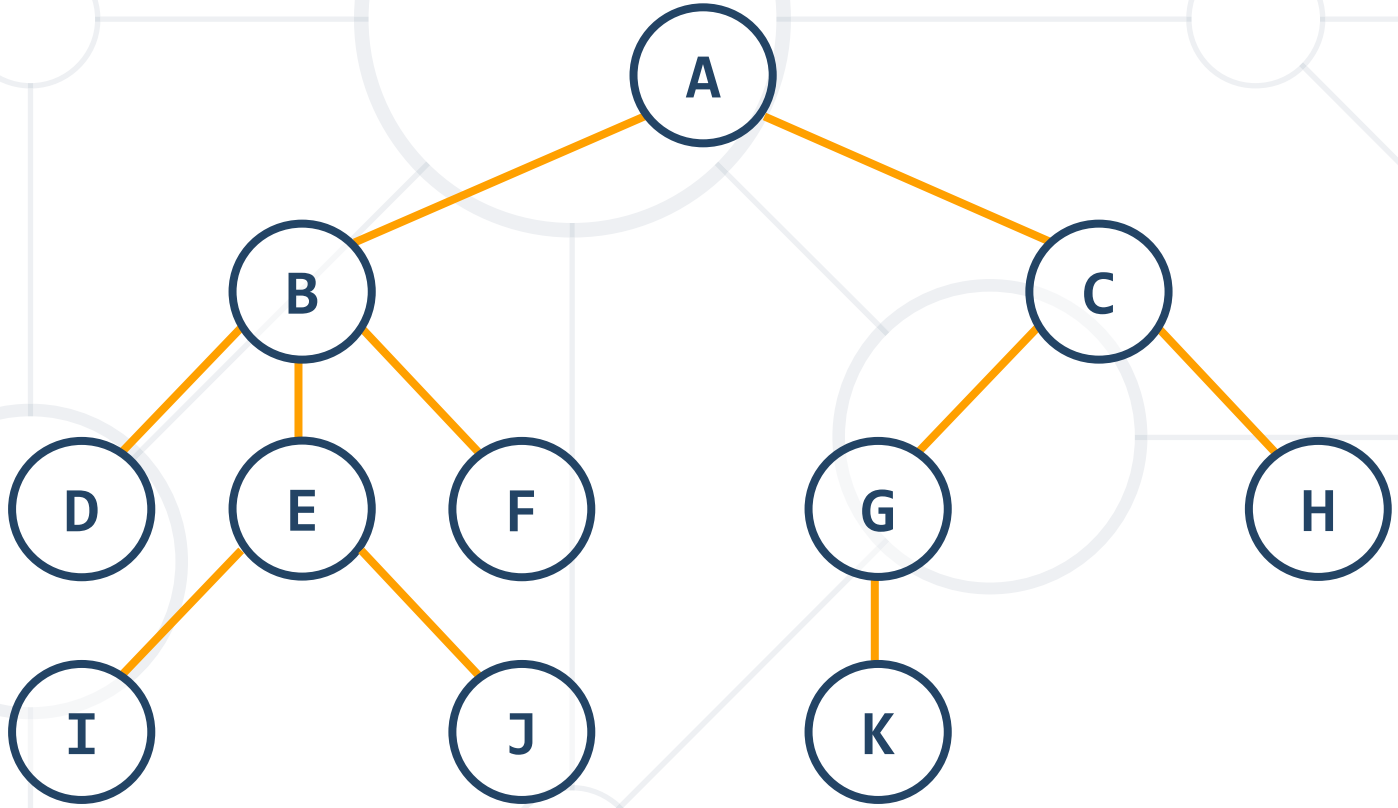
Tree Data Structure – Terminology

- **Root** – the **top node** in a **tree**, the **prime ancestor**, there must be only one root node



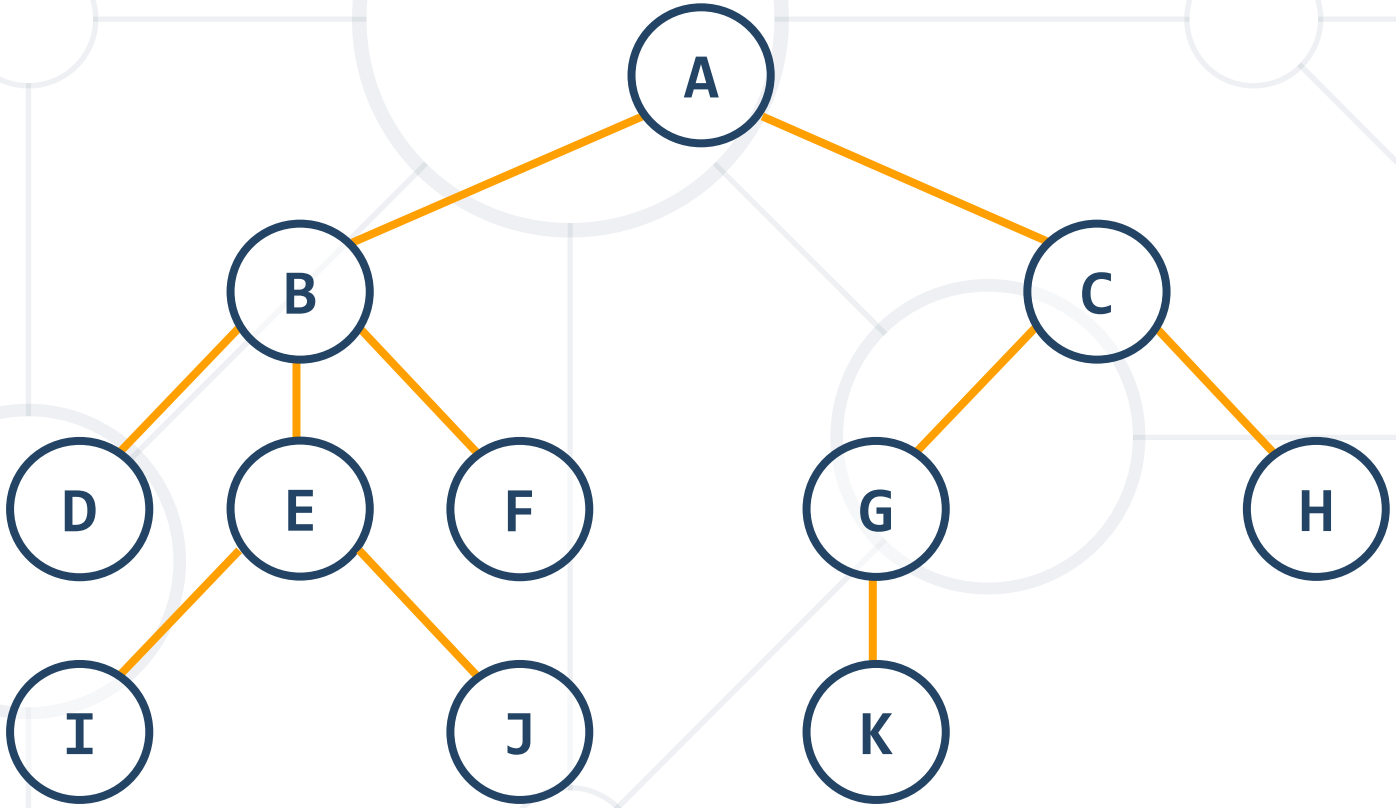
Tree Data Structure – Terminology

- **Edge** – the **connection between** one **node** and **another**



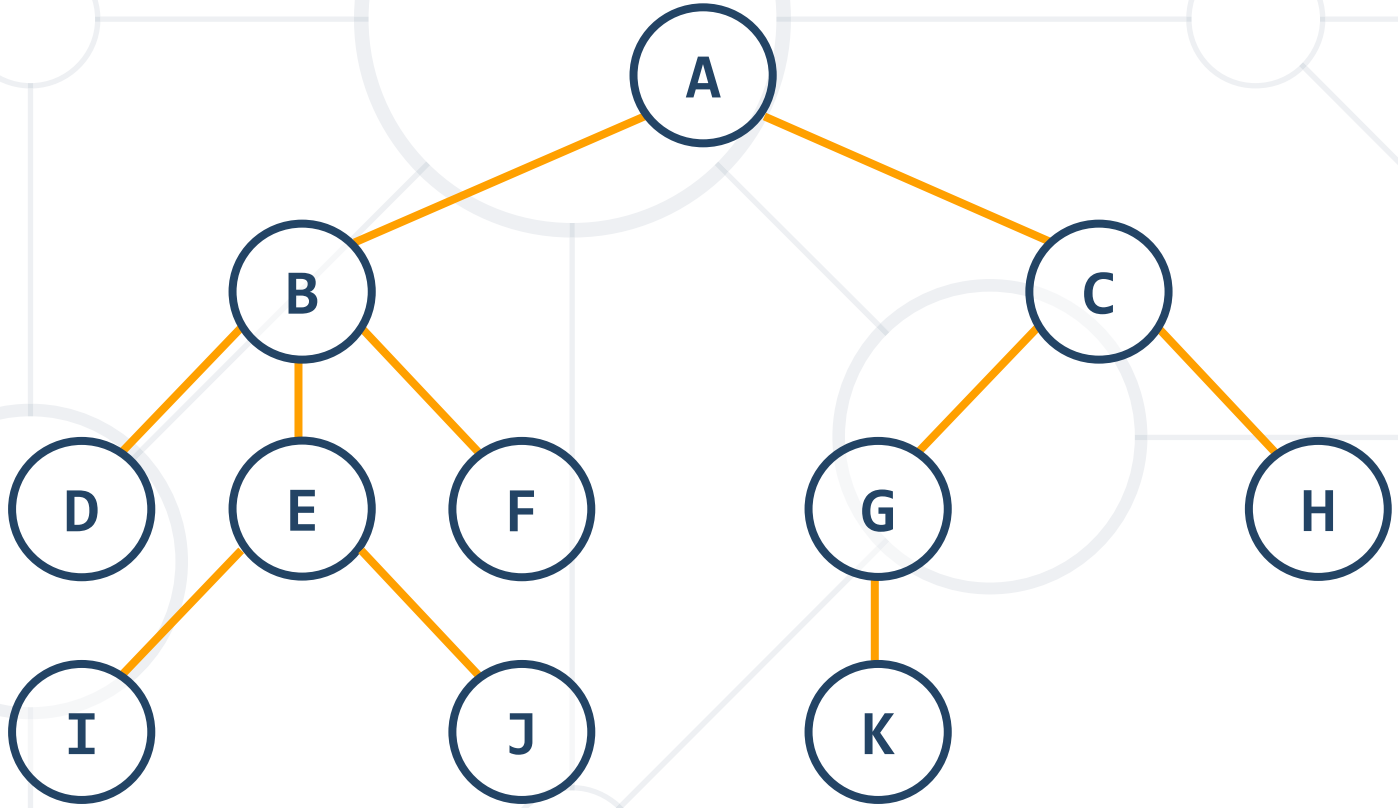
Tree Data Structure – Terminology

- **Parent** – node, which has one or more children



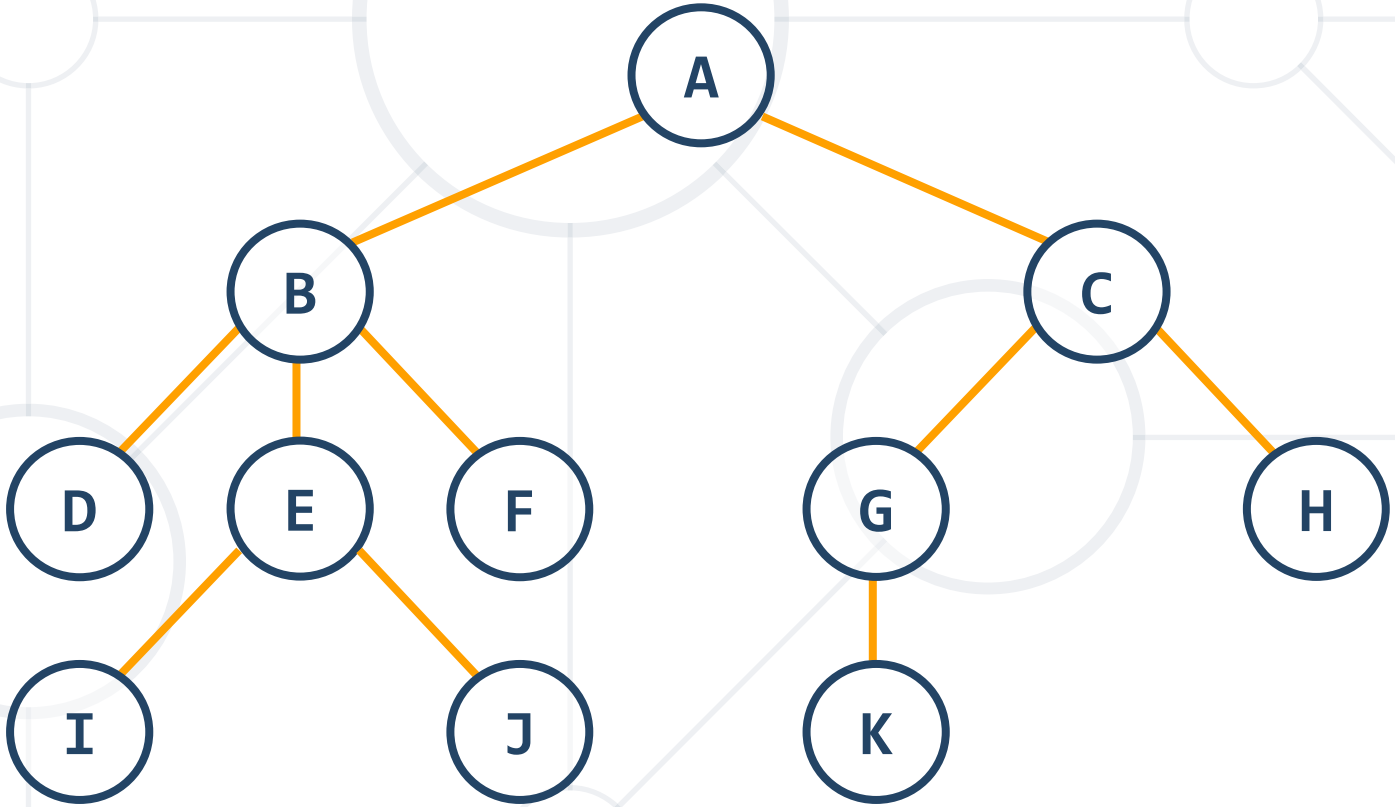
Tree Data Structure – Terminology

- **Child** – node **directly** connected to **another** node when moving **away** from the **root**, an immediate descendant



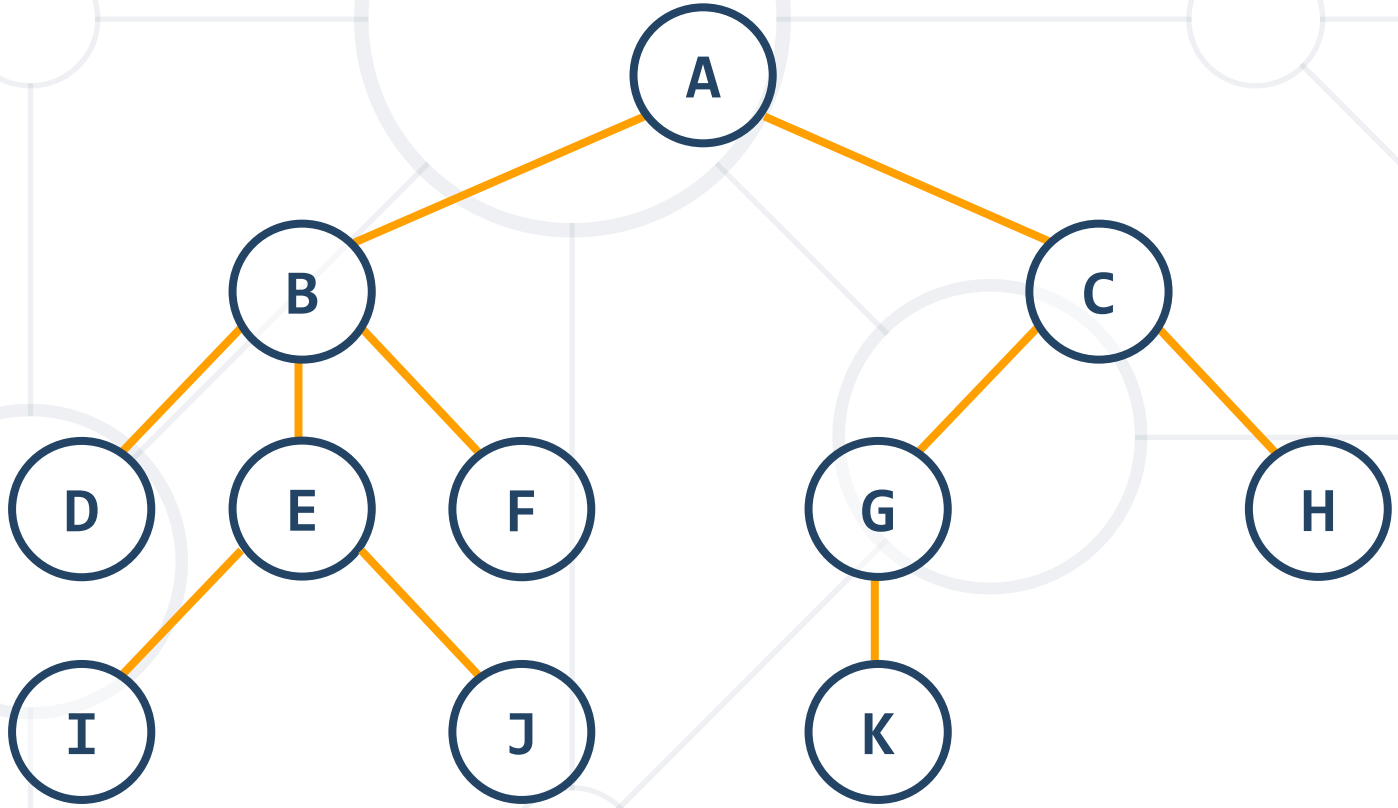
Tree Data Structure – Terminology

- **Siblings** – nodes with the **same parent**



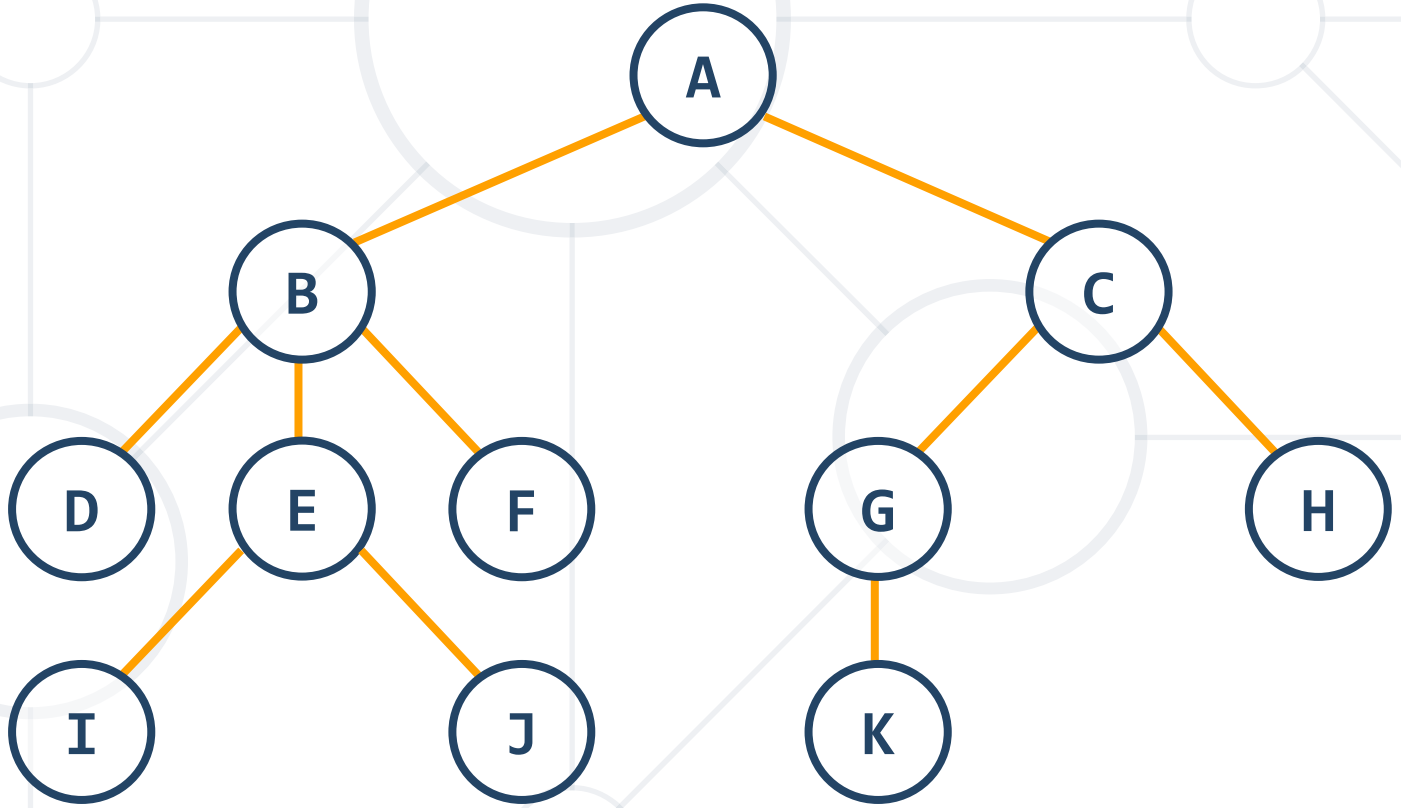
Tree Data Structure – Terminology

- **Ancestor** – node reachable by repeated proceeding from child to parent



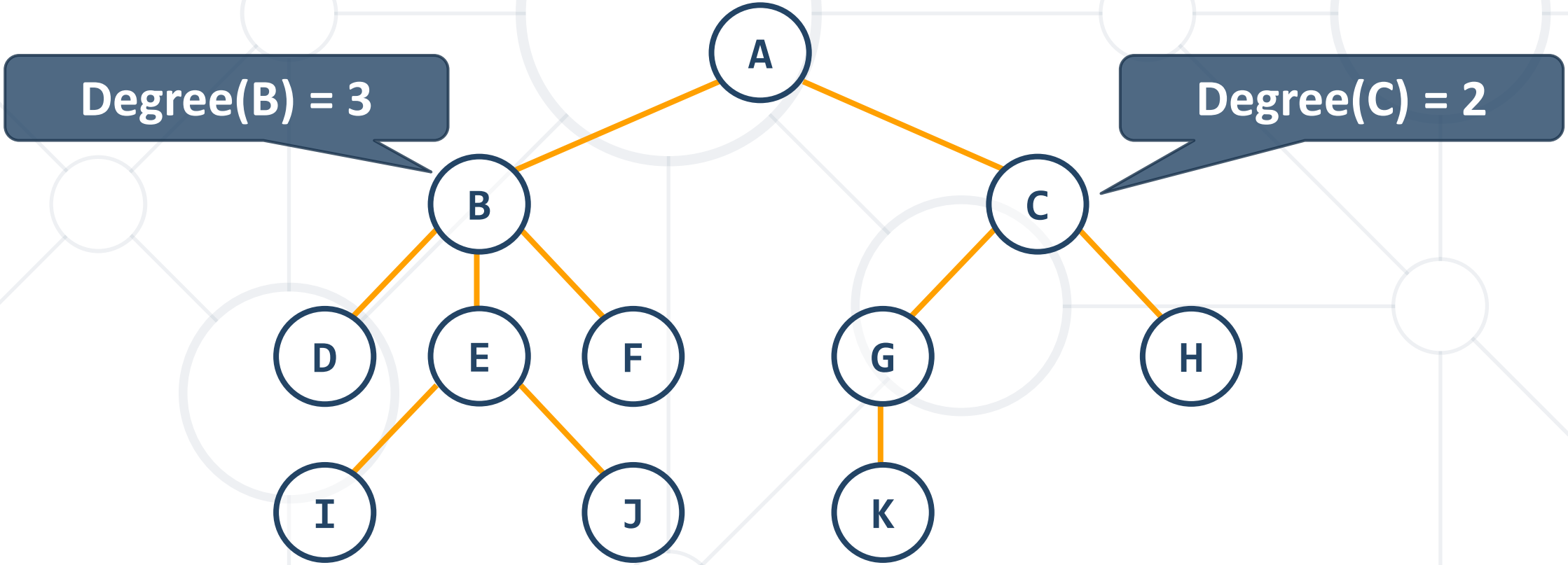
Tree Data Structure – Terminology

- **Descendant** – node reachable by repeated proceeding from parent to child



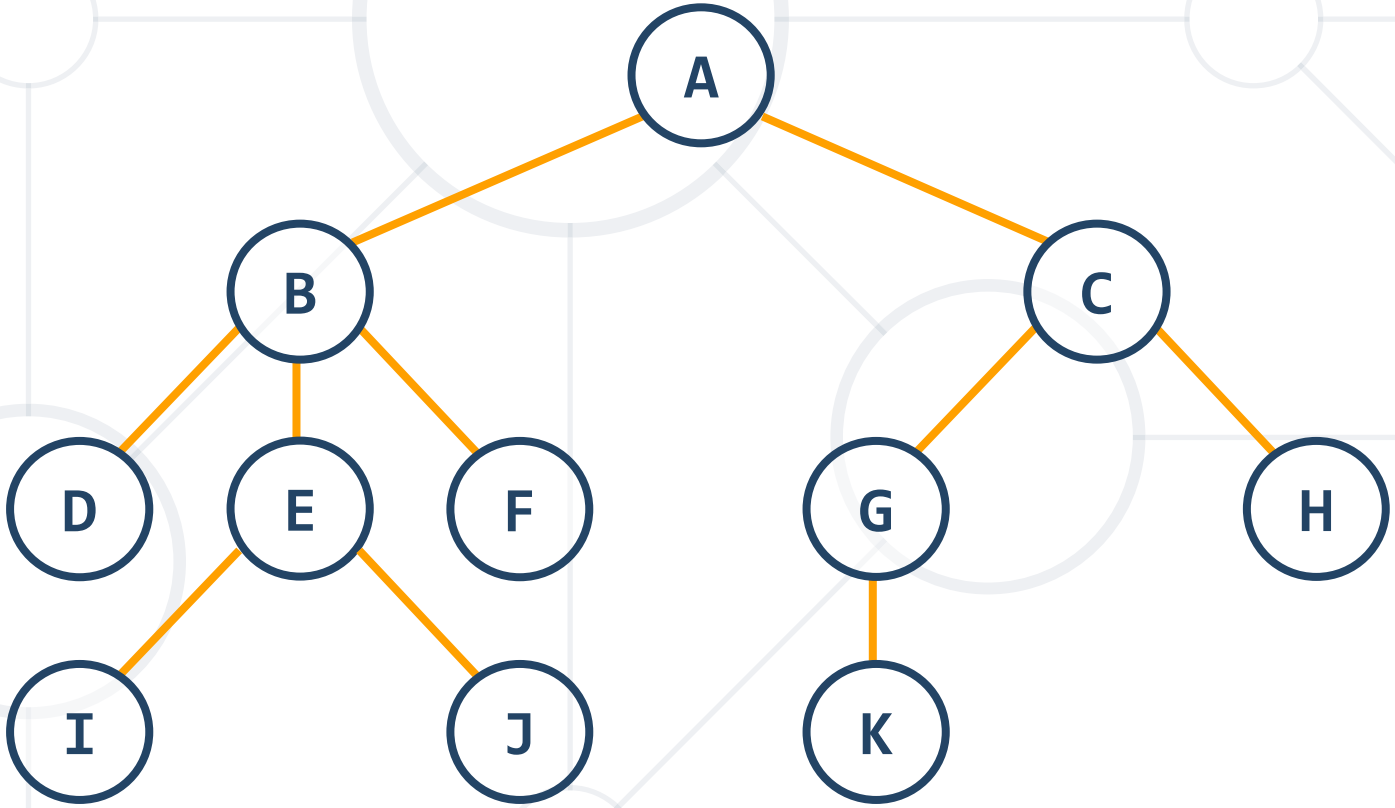
Tree Data Structure – Terminology

- **Degree** – total number of a children for a node



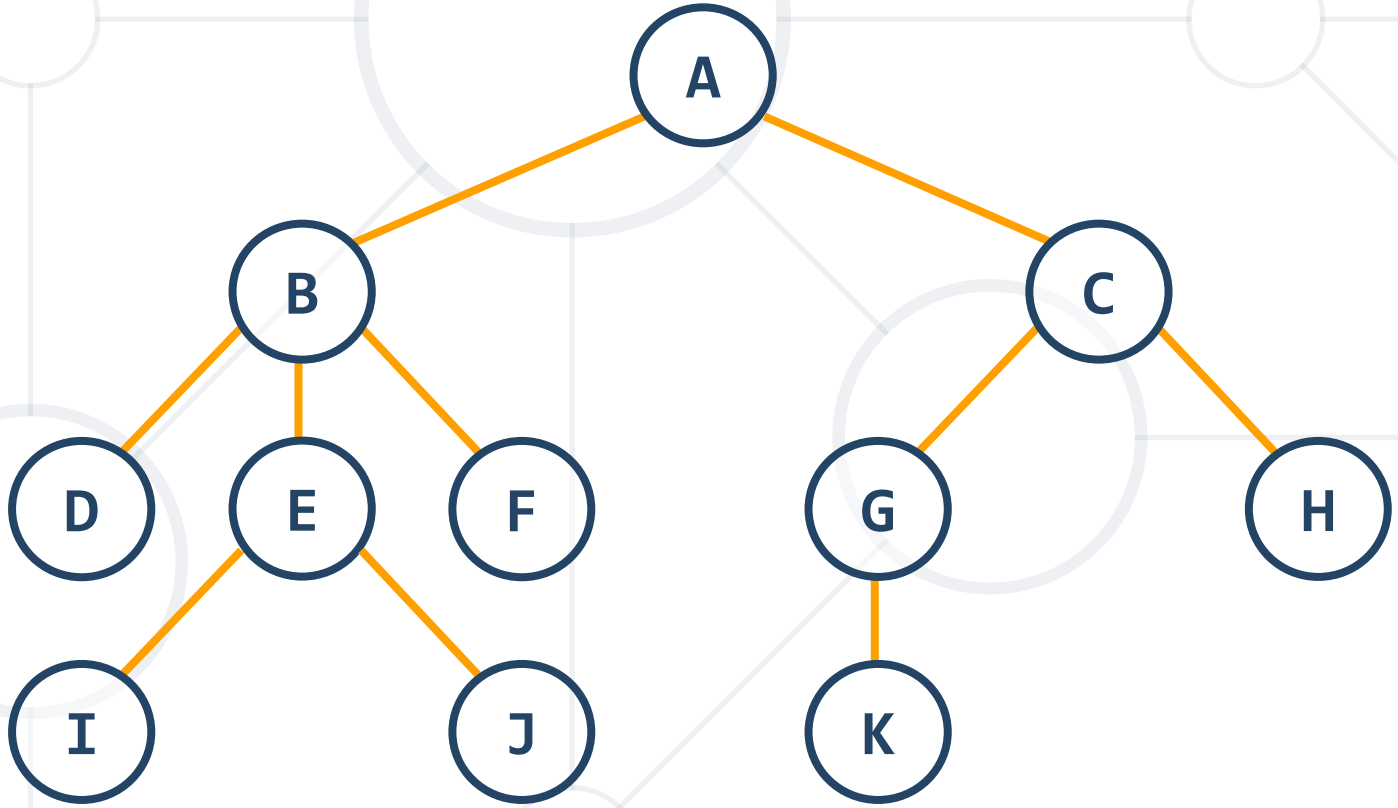
Tree Data Structure – Terminology

- **Leaf Node** – node without any children



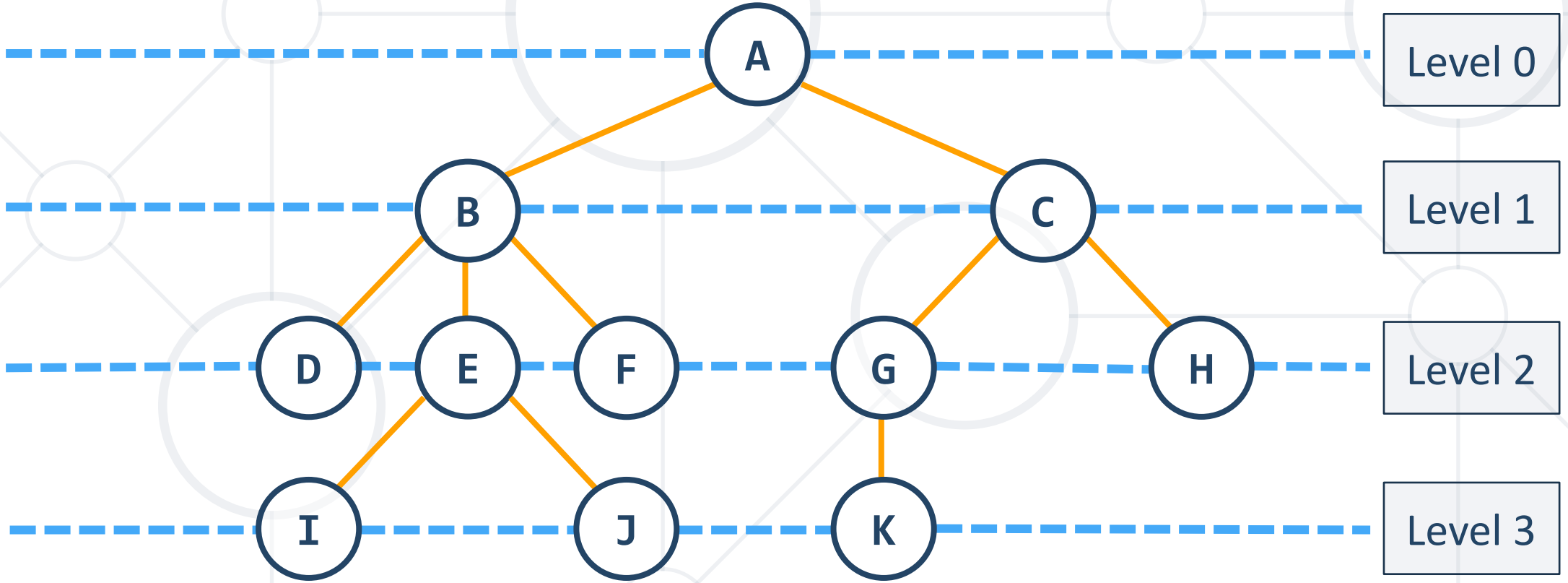
Tree Data Structure – Terminology

- **Internal Node** – every non-leaf node, also know as **branch**



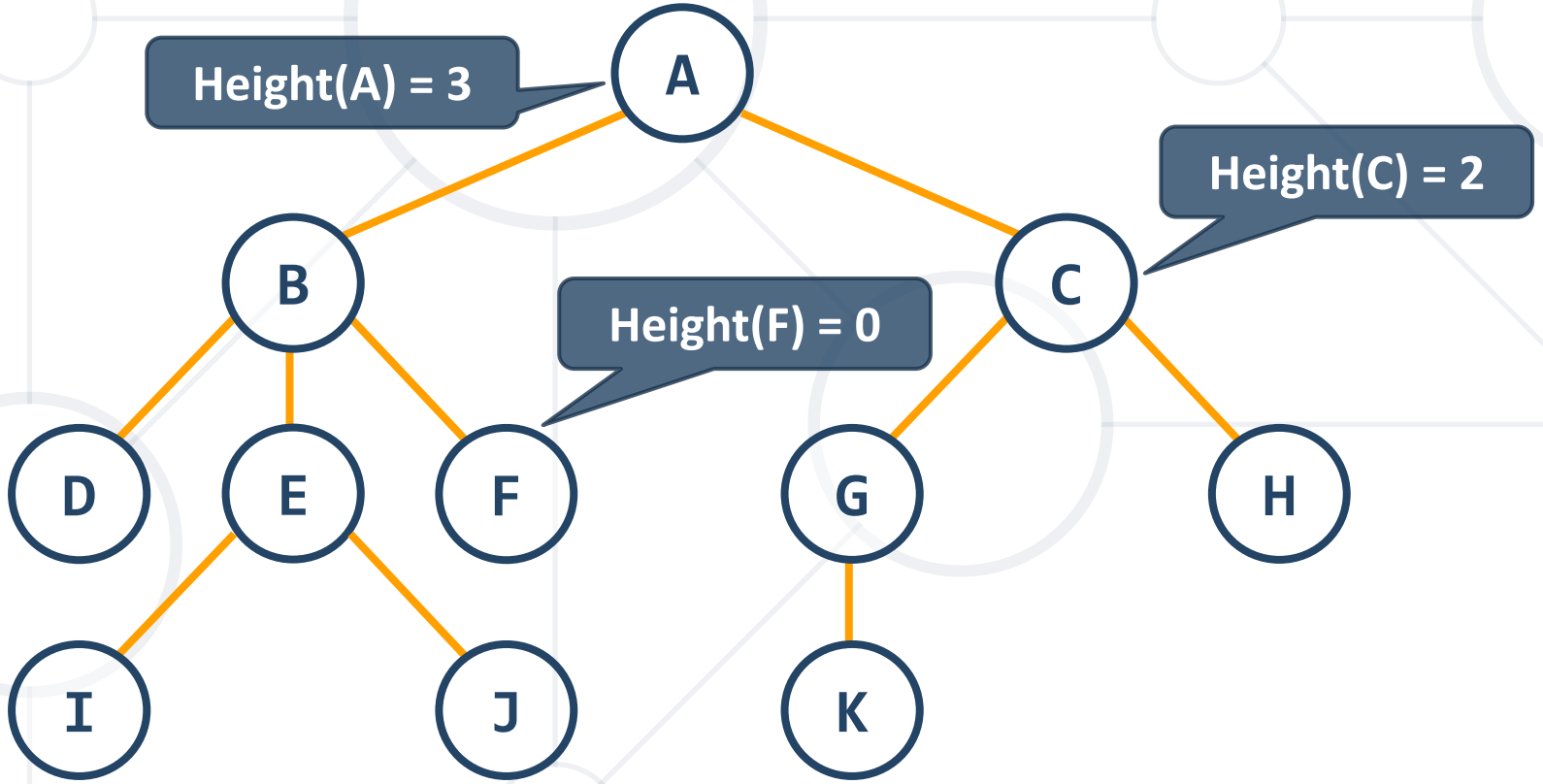
Tree Data Structure – Terminology

- **Level** – each step from **top to bottom** starting from 0 is called level of a tree



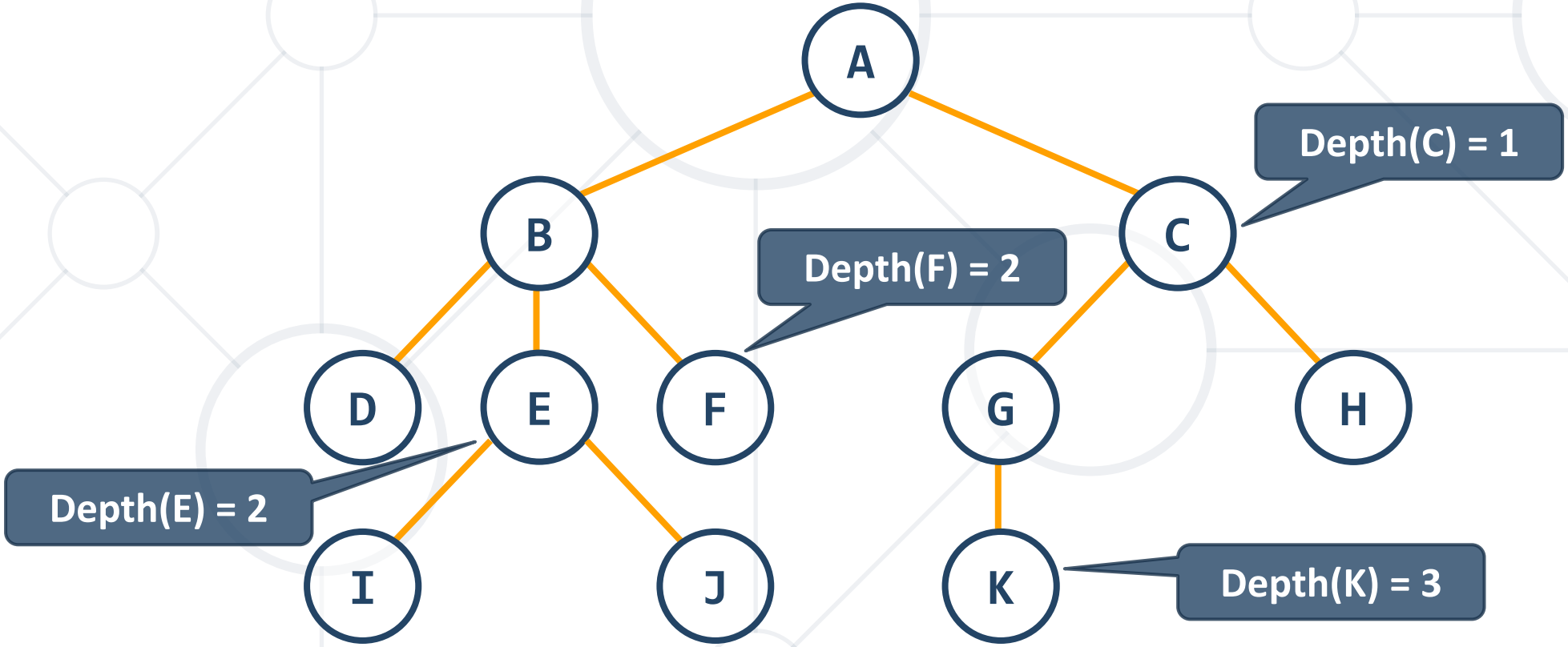
Tree Data Structure – Terminology

- **Height** – longest number of edges from a leaf to a specific node is called height of a node. Height of a tree is the height of the root node



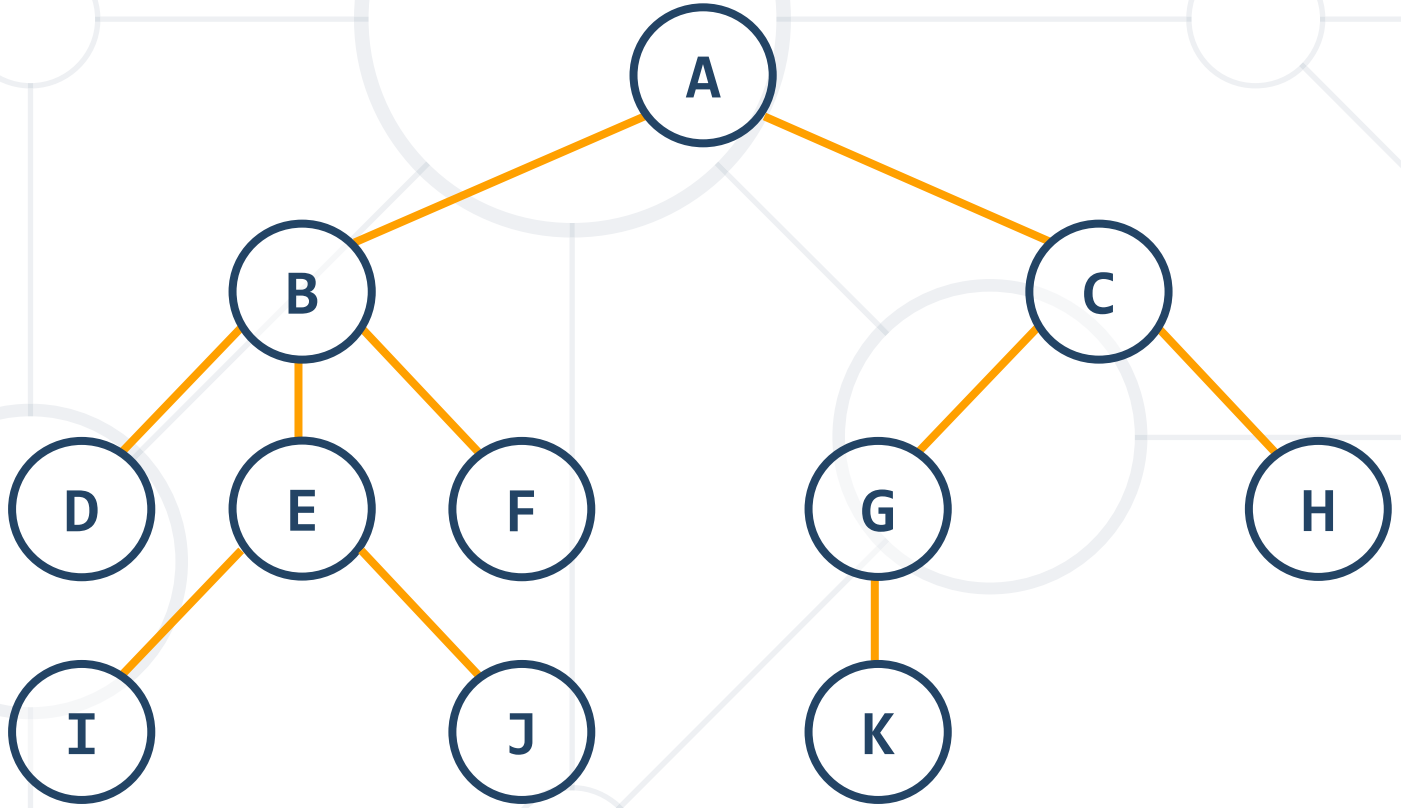
Tree Data Structure – Terminology

- **Depth** – the total number of edges from the root to a particular node



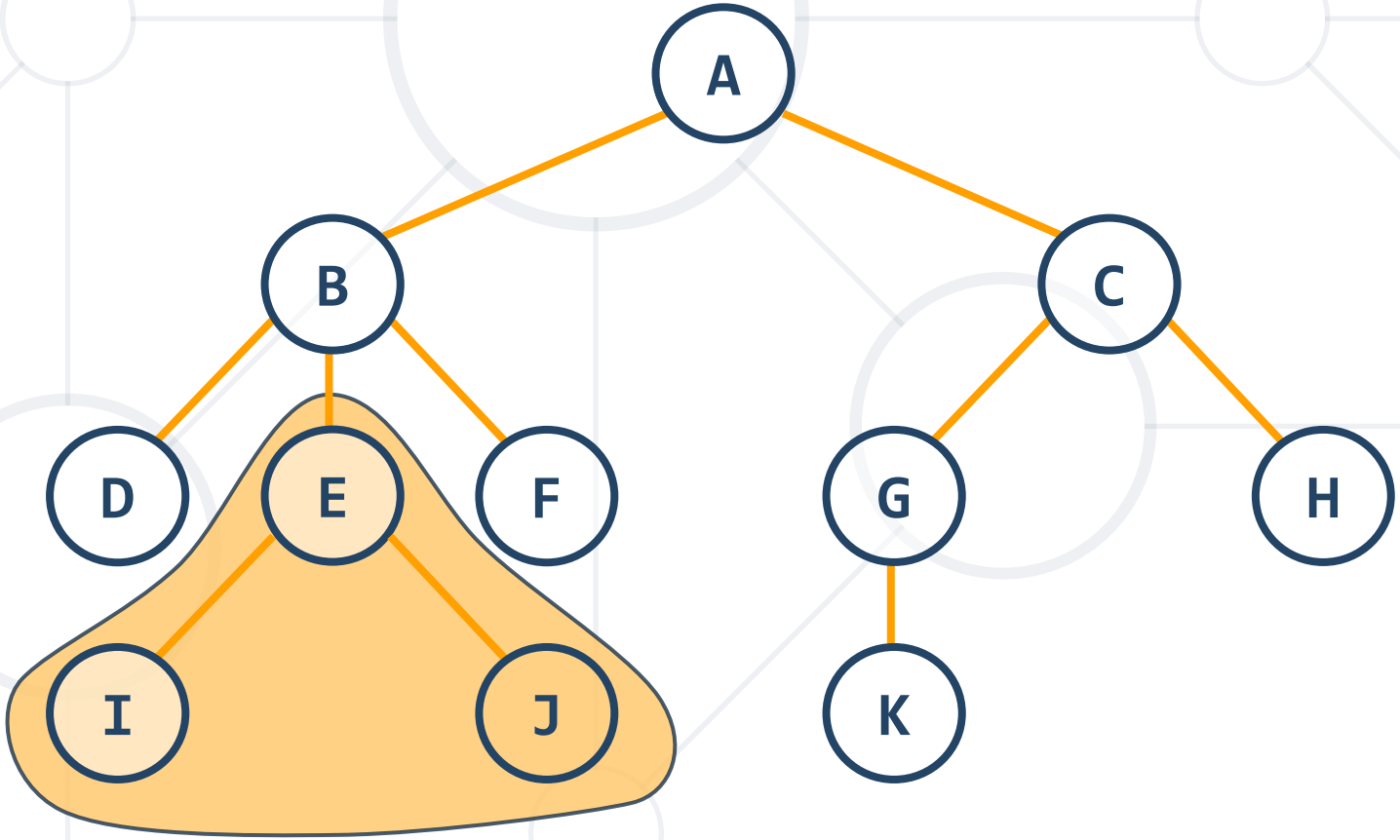
Tree Data Structure – Terminology

- **Path** – the sequence of nodes and edges from one node to another. **Length of the path** is the number of nodes



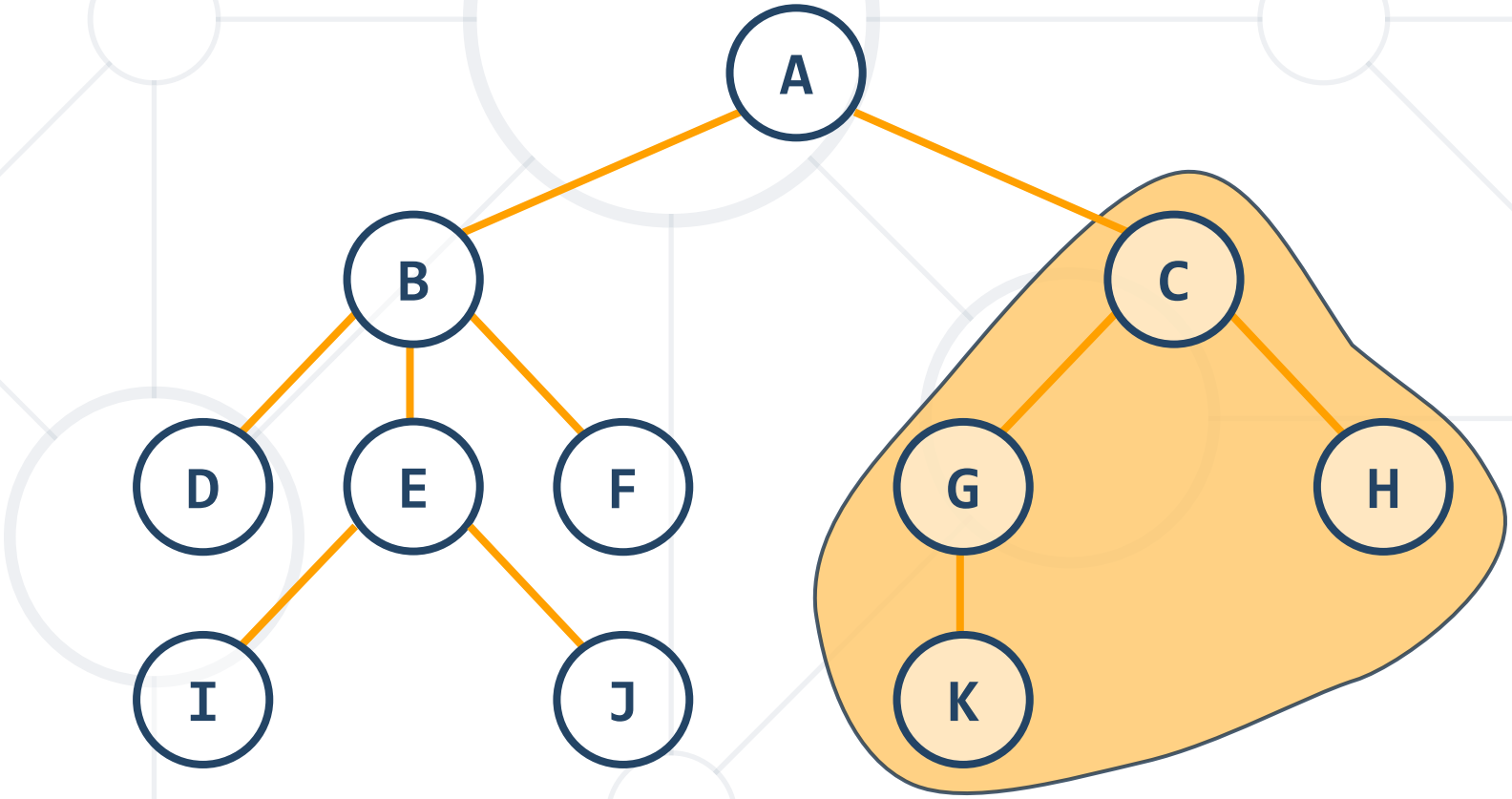
Tree Data Structure – Terminology

- **Subtree** – each child from a node forms a subtree recursively



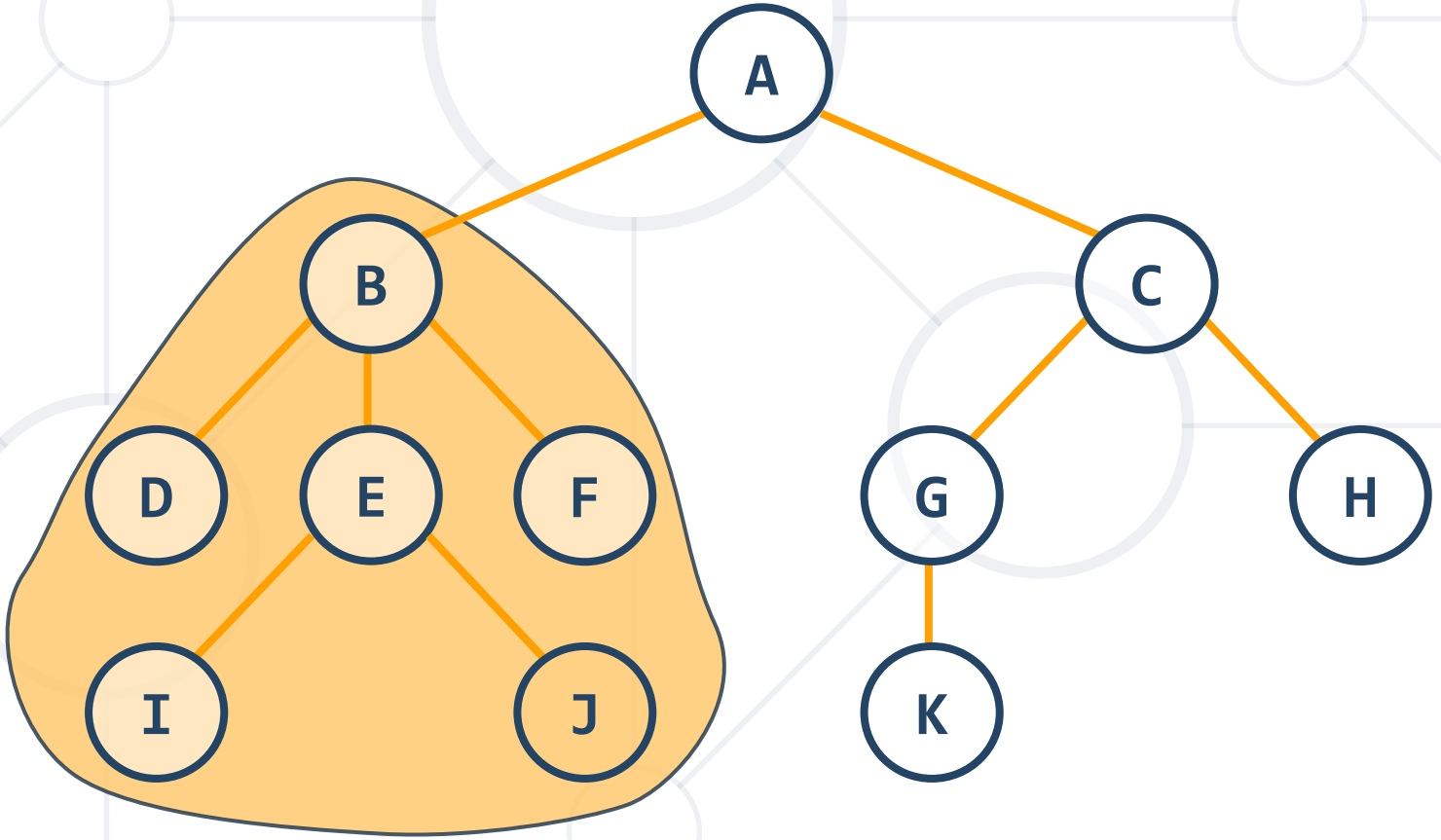
Tree Data Structure – Terminology

- **Subtree** – each child from a node forms a subtree recursively



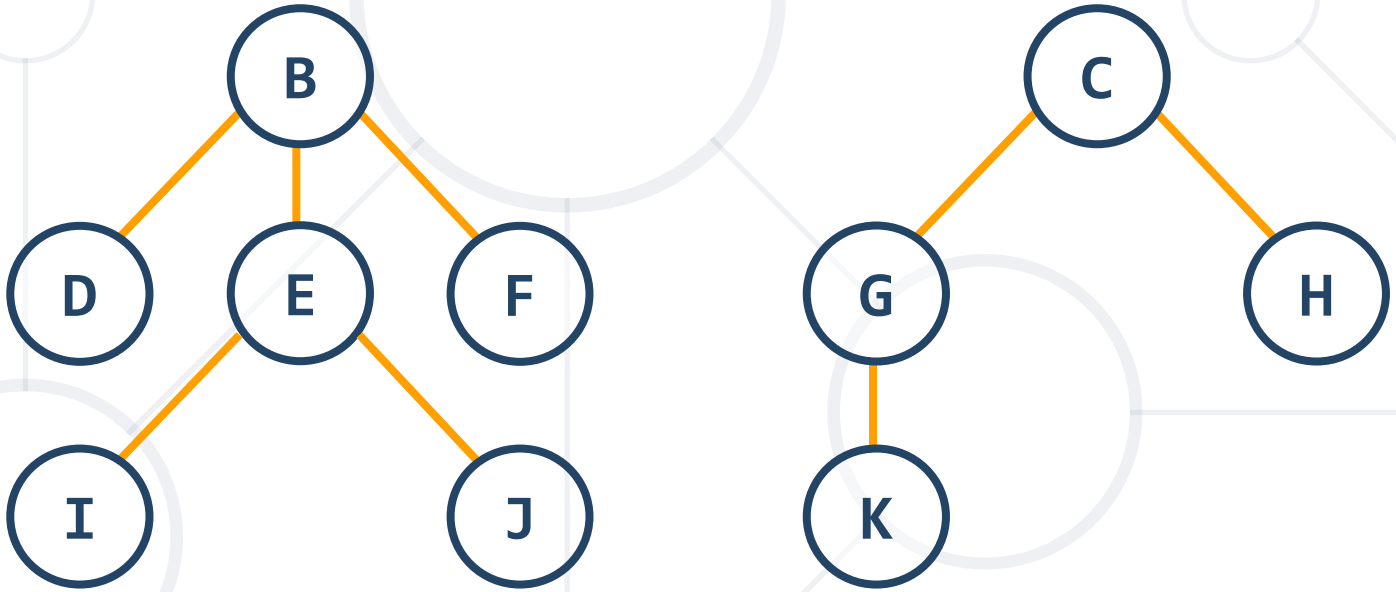
Tree Data Structure – Terminology

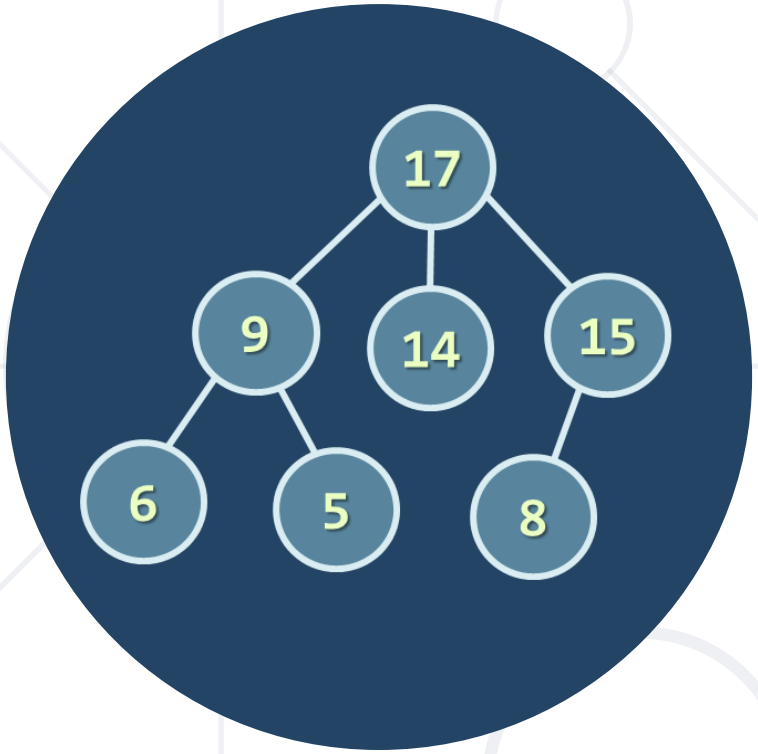
- **Subtree** – each child from a node forms a subtree recursively



Tree Data Structure – Terminology

- **Forest** – Set of disjoint trees





Implementing Trees

Recursive Tree Data Structure

Recursive Tree Definition

- The recursive definition for **tree** data structure:
  - A single node **is a tree**
  - Nodes have **zero or multiple children** that are **also trees**

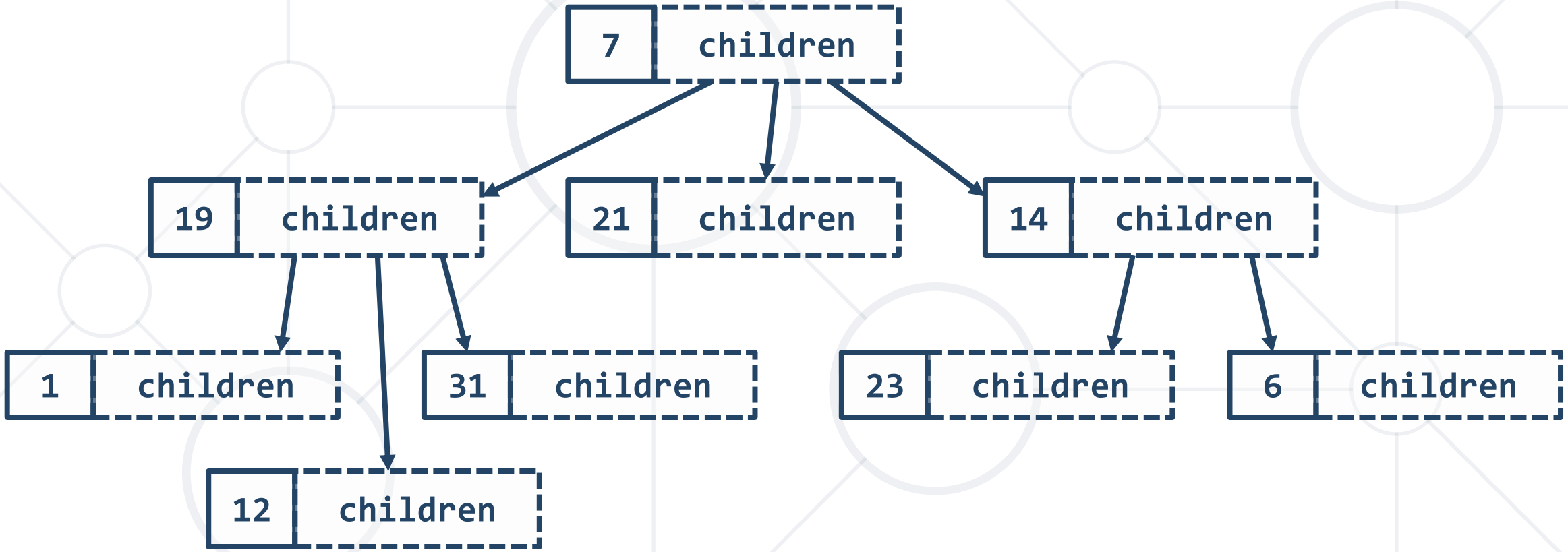
```
public class Tree<T> : IAbstractTree<T>
{
    private T value;
    private Tree<T> parent;
    private List<Tree<T>> children;
}
```

The stored value

The parent

List of child nodes

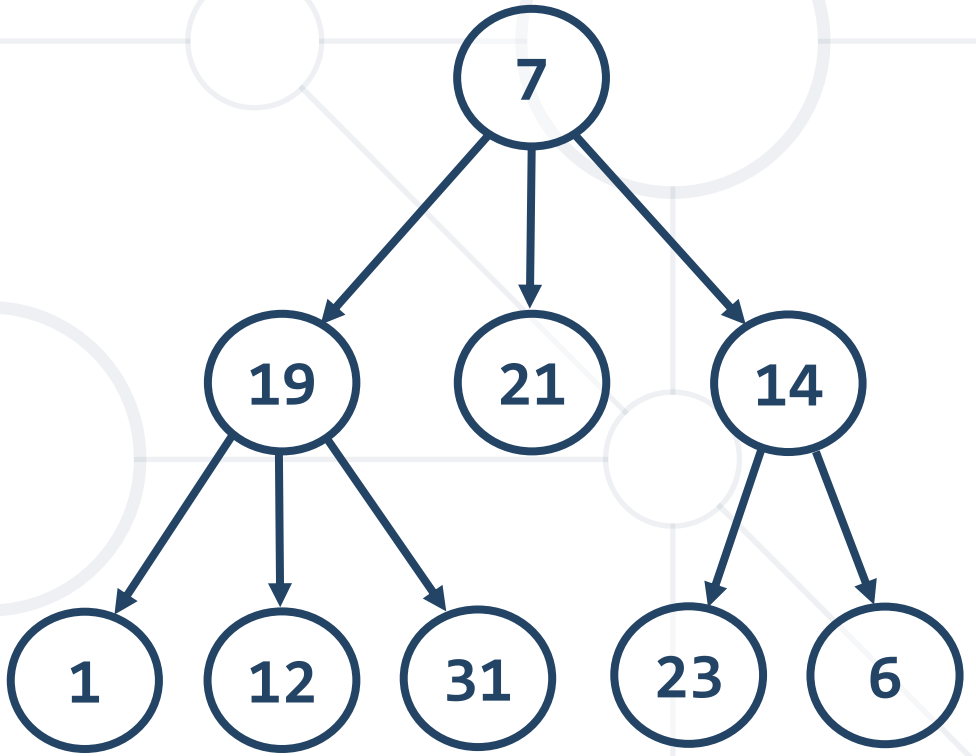
Tree<int> Structure – Example



Problem: Implement Tree Constructor

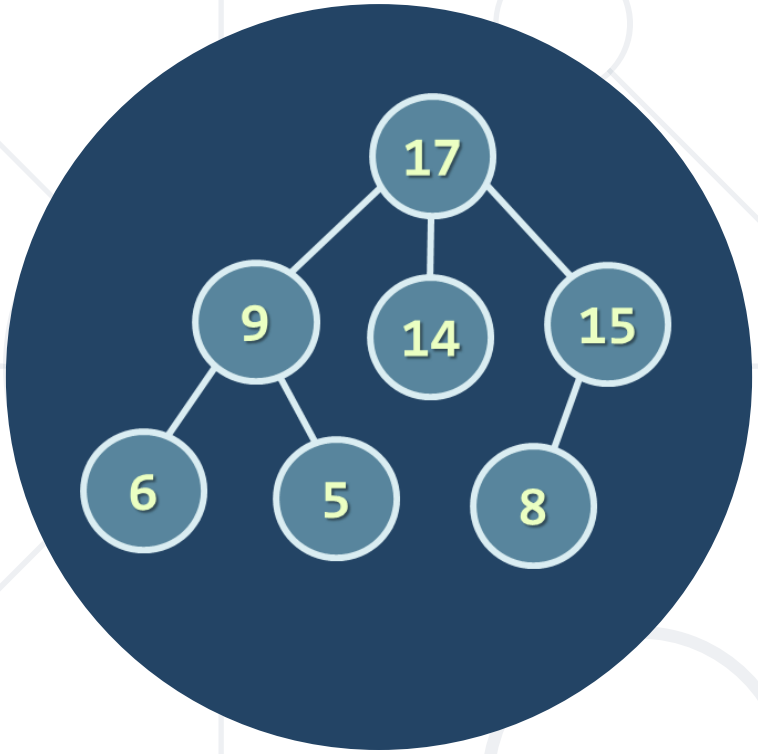
- Create a **Tree** constructor

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



Solution: Implement Tree Constructor

```
public class Tree<T> : IAbstractTree<T>
{
    private List<Tree<T>> children;
    private T Value;
    public Tree(T value, params Tree<T>[] children)
    {
        this.Value = value;
        this.children = new List<Tree<T>>();
        for (Tree<T> child : children)
        {
            this.children.Add(child);
        }
    }
}
```



Traversing Tree-Like Structures

DFS and BFS Traversals

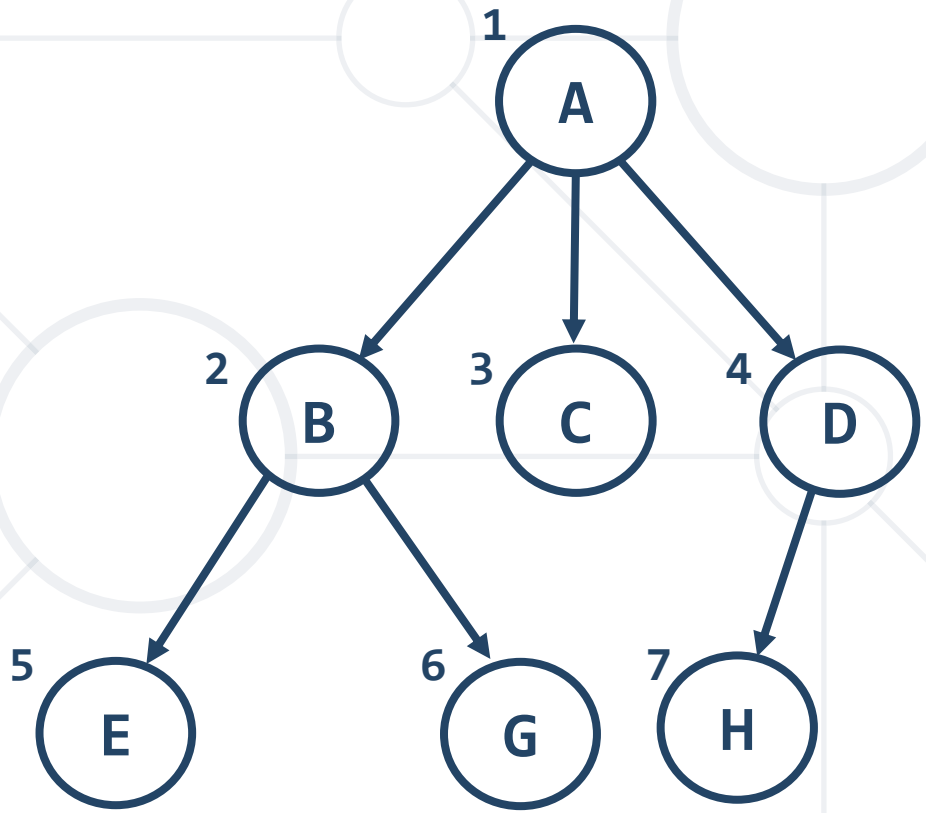
Tree Traversal Algorithms

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

Breadth-First Search (BFS)

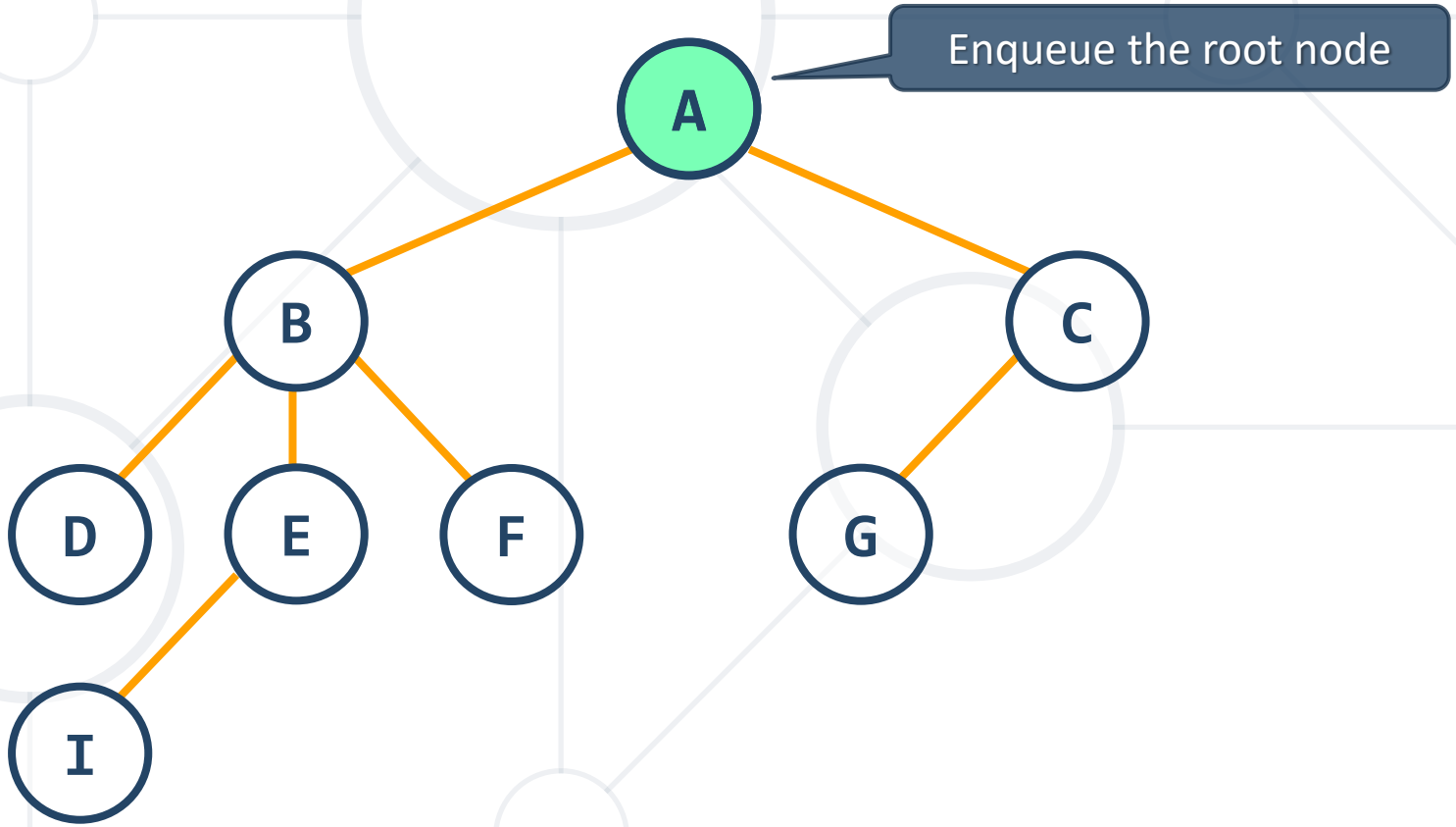
- **Breadth-First Search** (BFS) first traverses the tree level by level
- BFS algorithm pseudo code:

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



BFS Visualization

- Queue:
- Output:

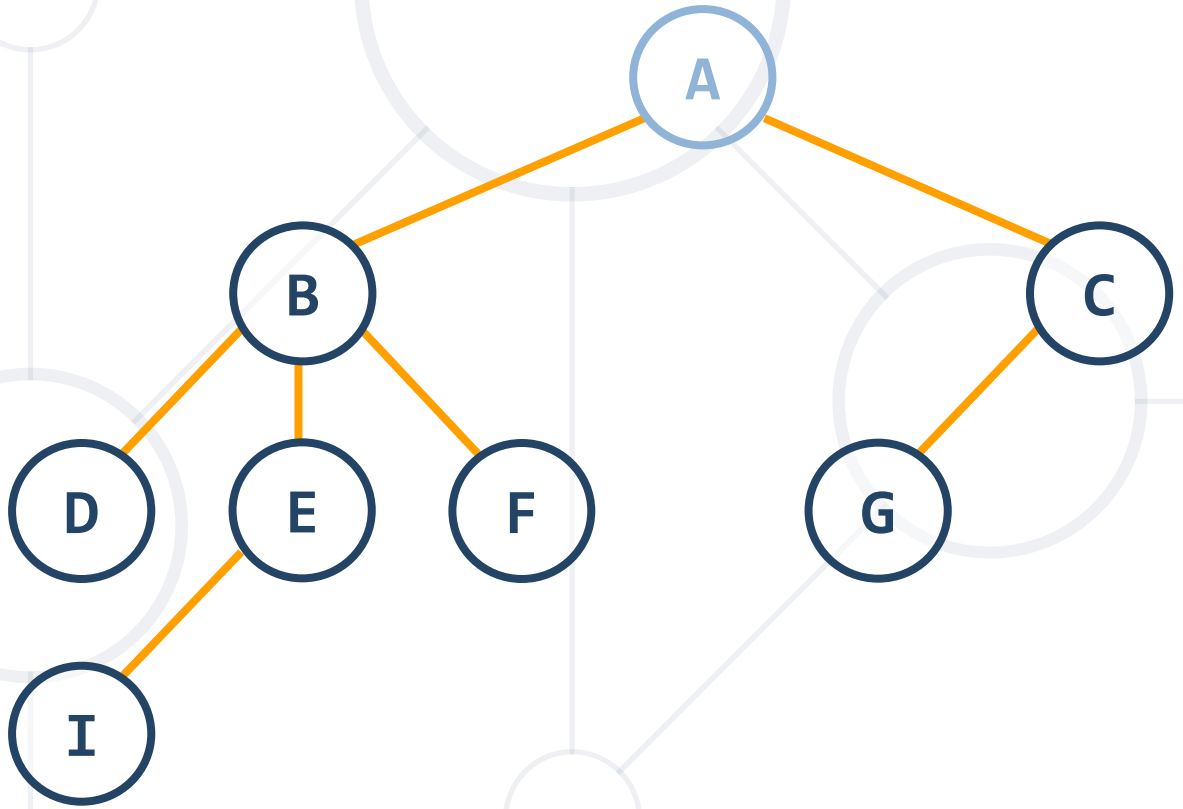


BFS Visualization

- Queue: A

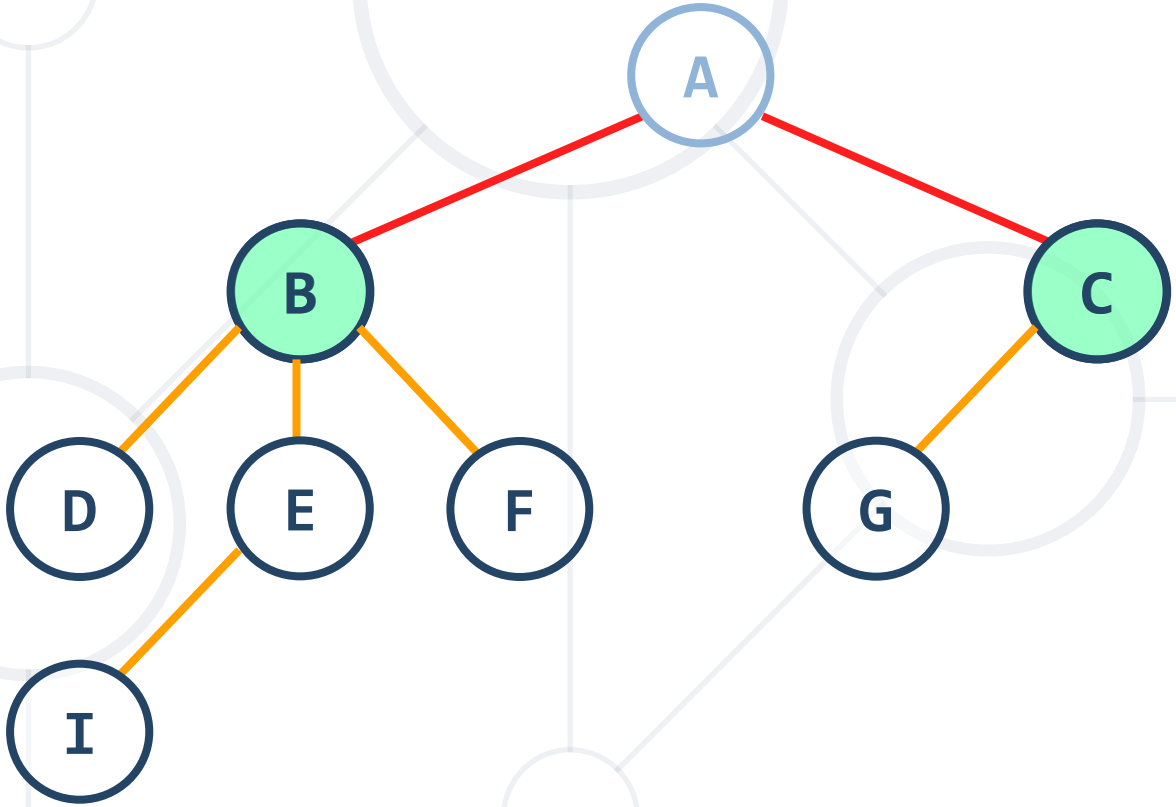
Dequeue and print the current node and add its children to the queue

- Output:



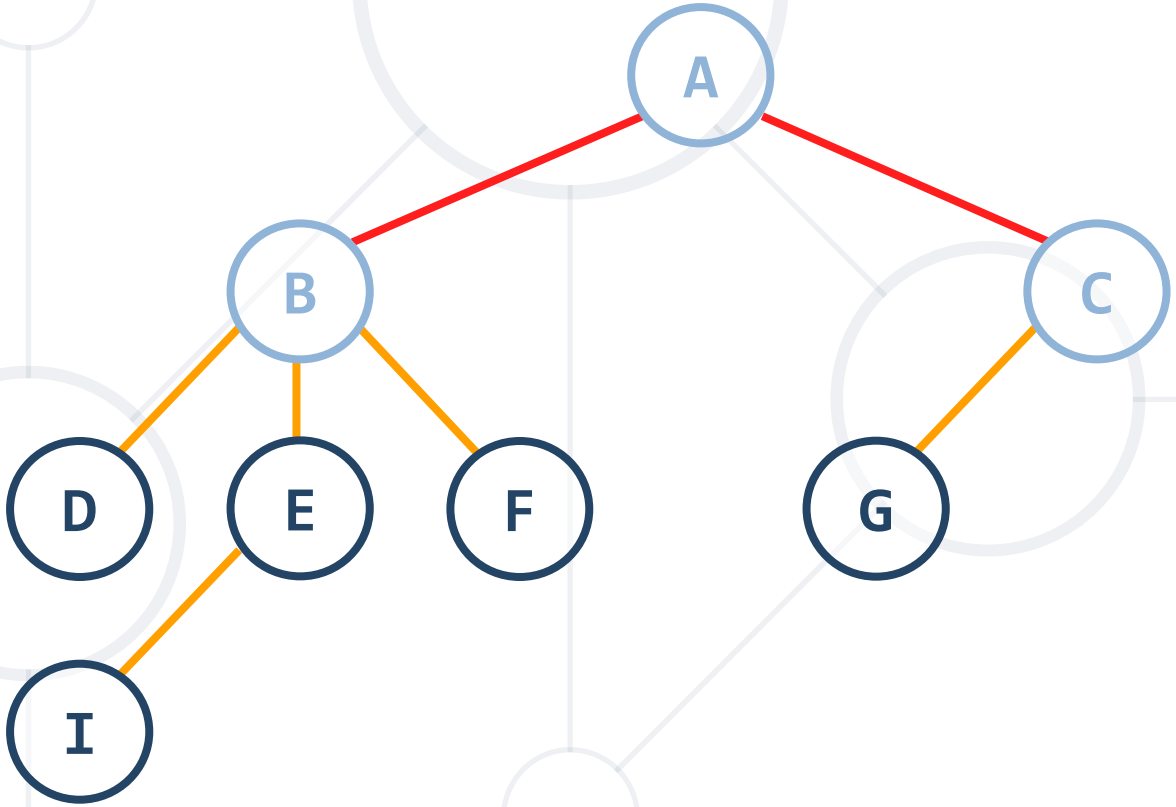
BFS Visualization

- Queue: A
- Output:



BFS Visualization

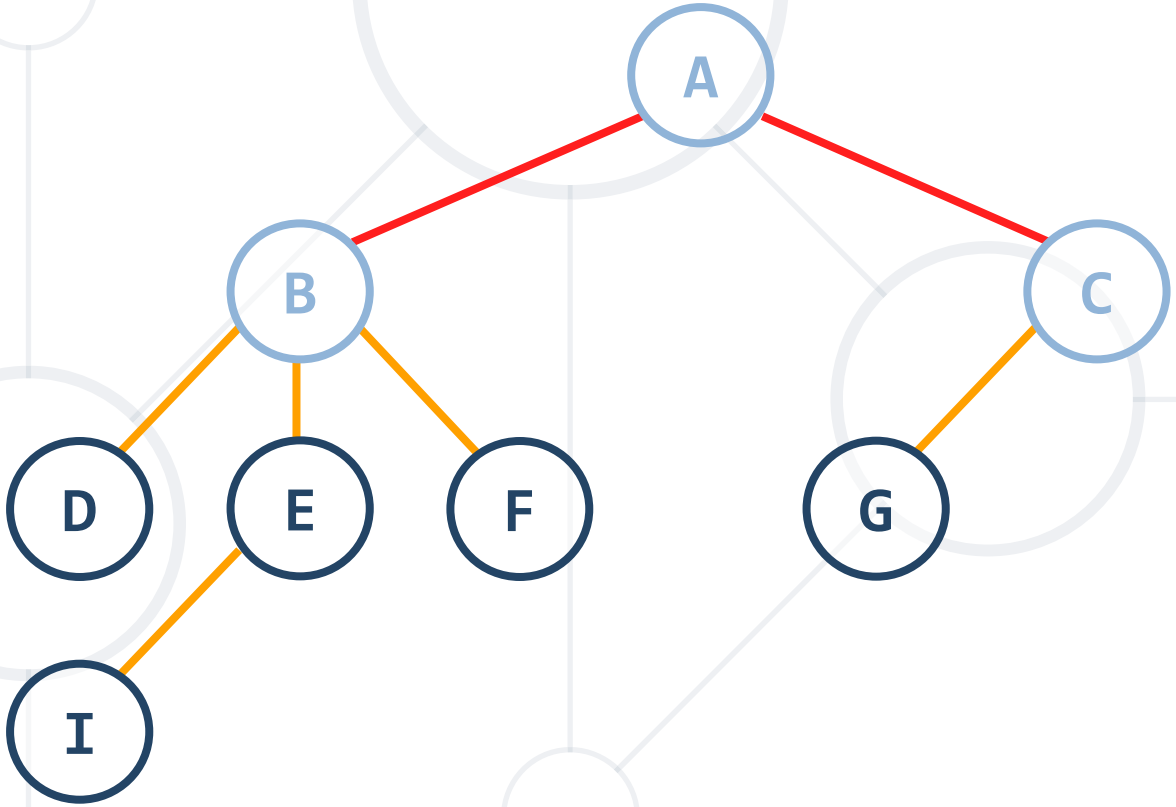
- Queue: B C
- Output: A



BFS Visualization

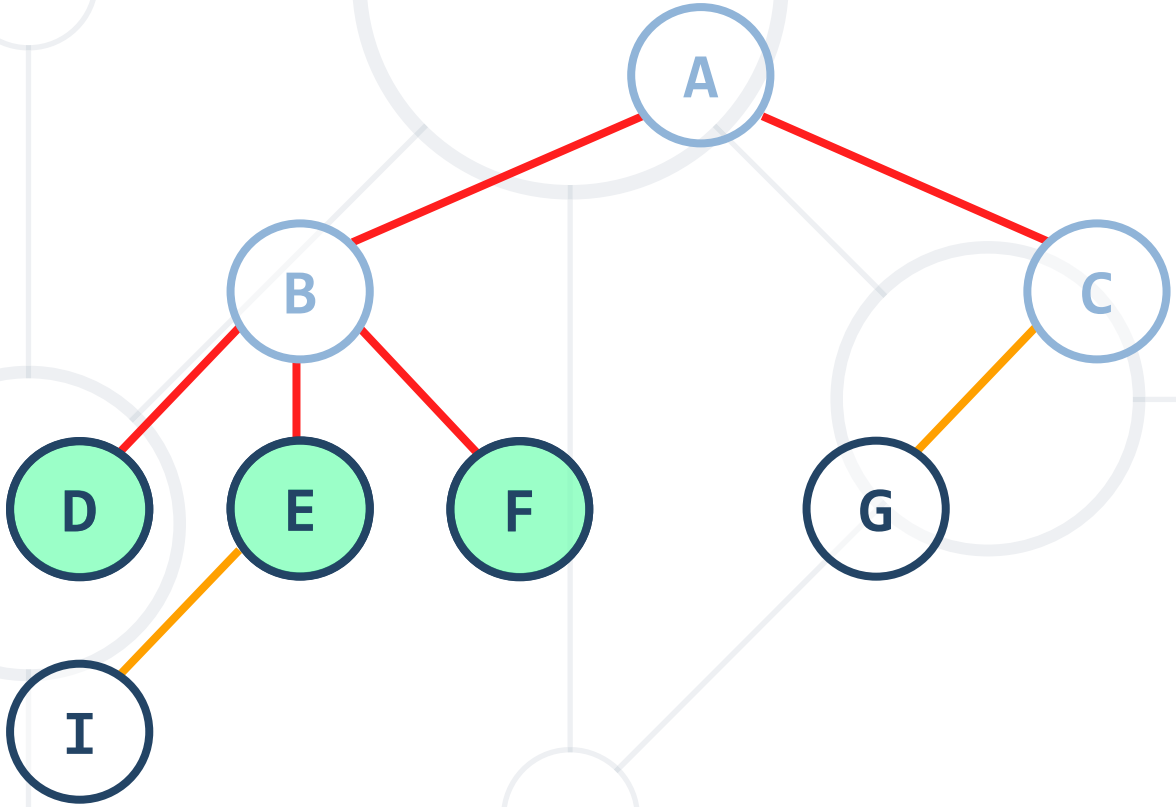
- Queue: B C
- Output: A

Dequeue and print the current node and add its children to the queue



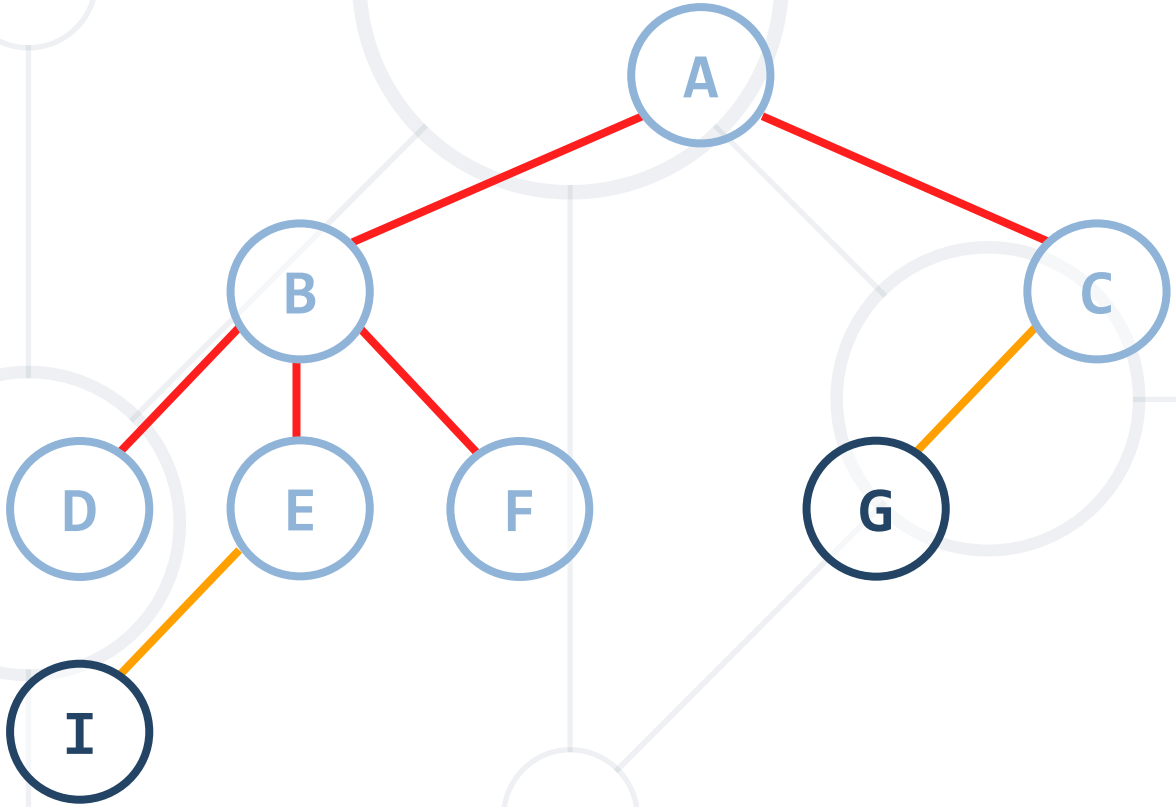
BFS Visualization

- Queue: C
- Output: A B



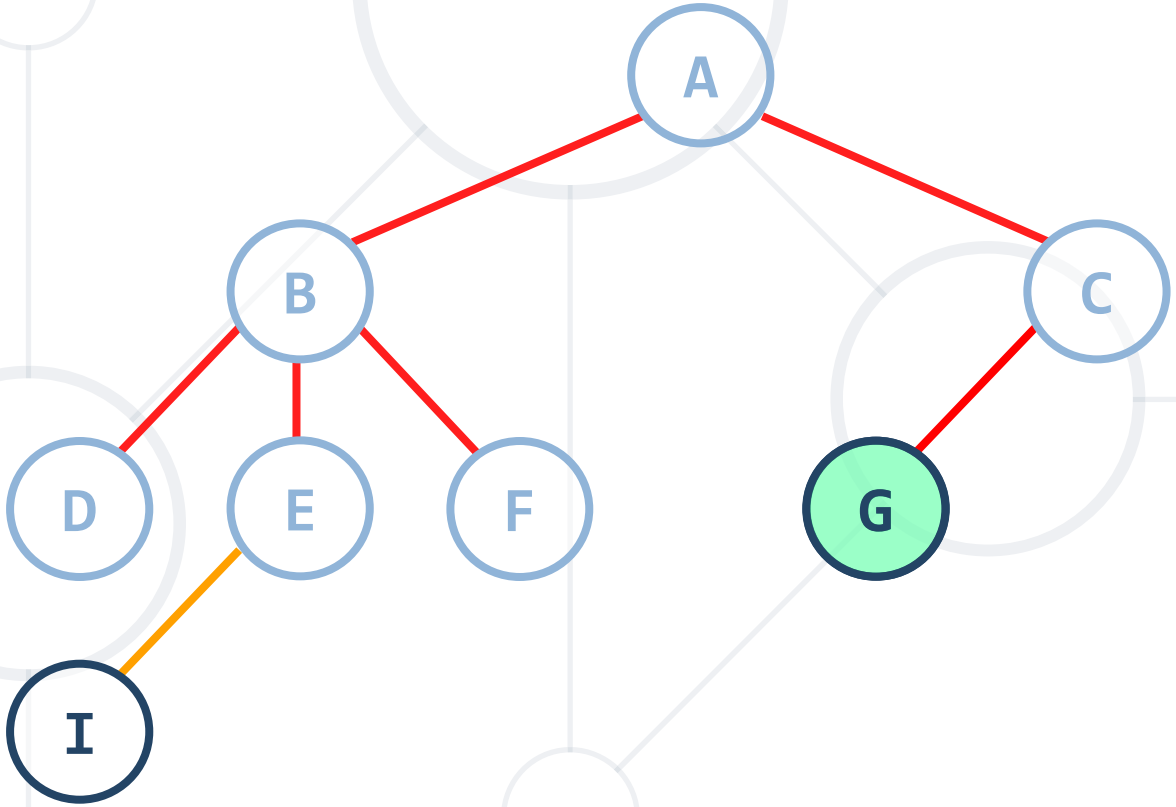
BFS Visualization

- Queue: C D E F
- Output: A B



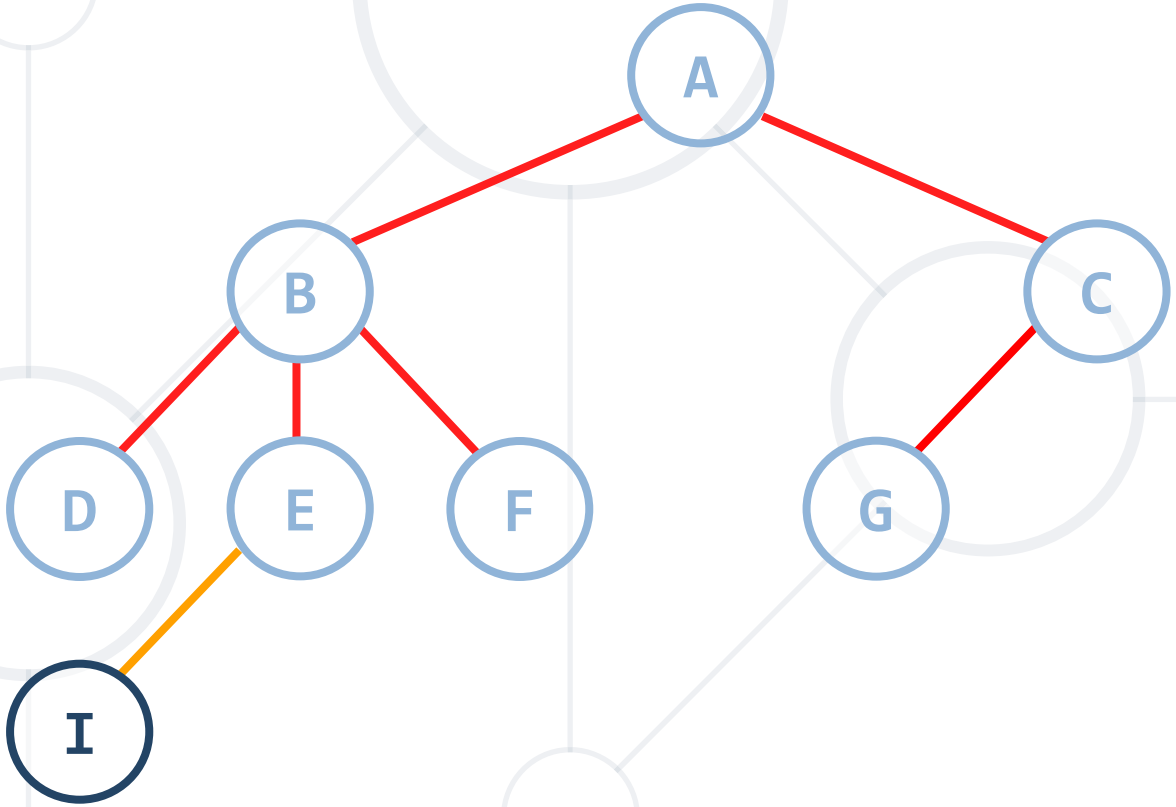
BFS Visualization

- Queue: D E F
- Output: A B C



BFS Visualization

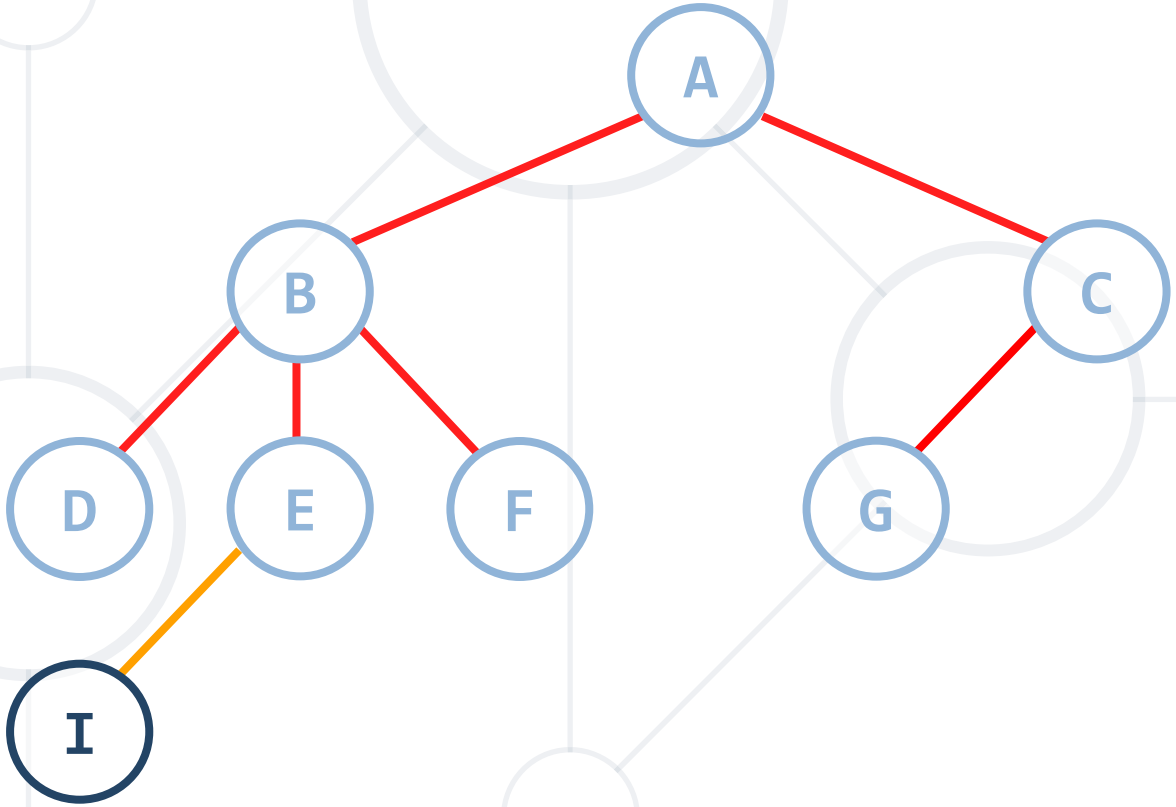
- Queue: D E F G
- Output: A B C



BFS Visualization

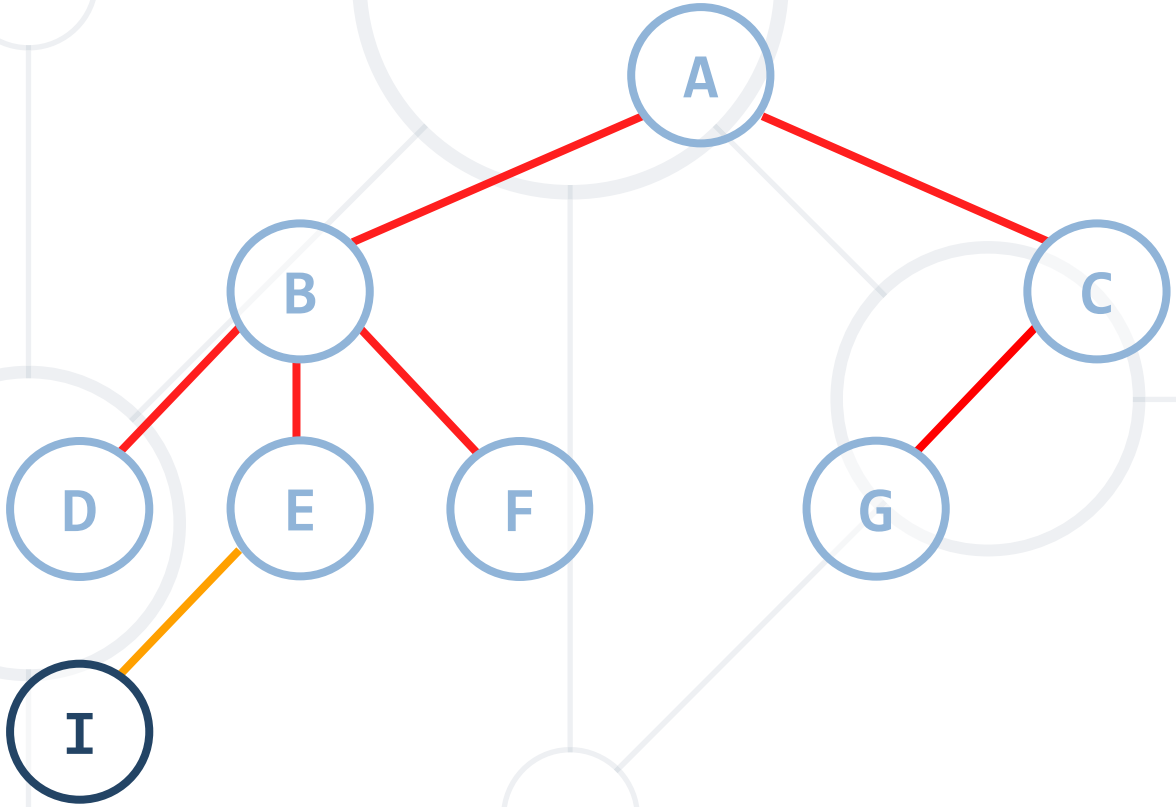
- Queue: D E F G
- Output: A B C

The first element doesn't have children, so we dequeue and go the next one



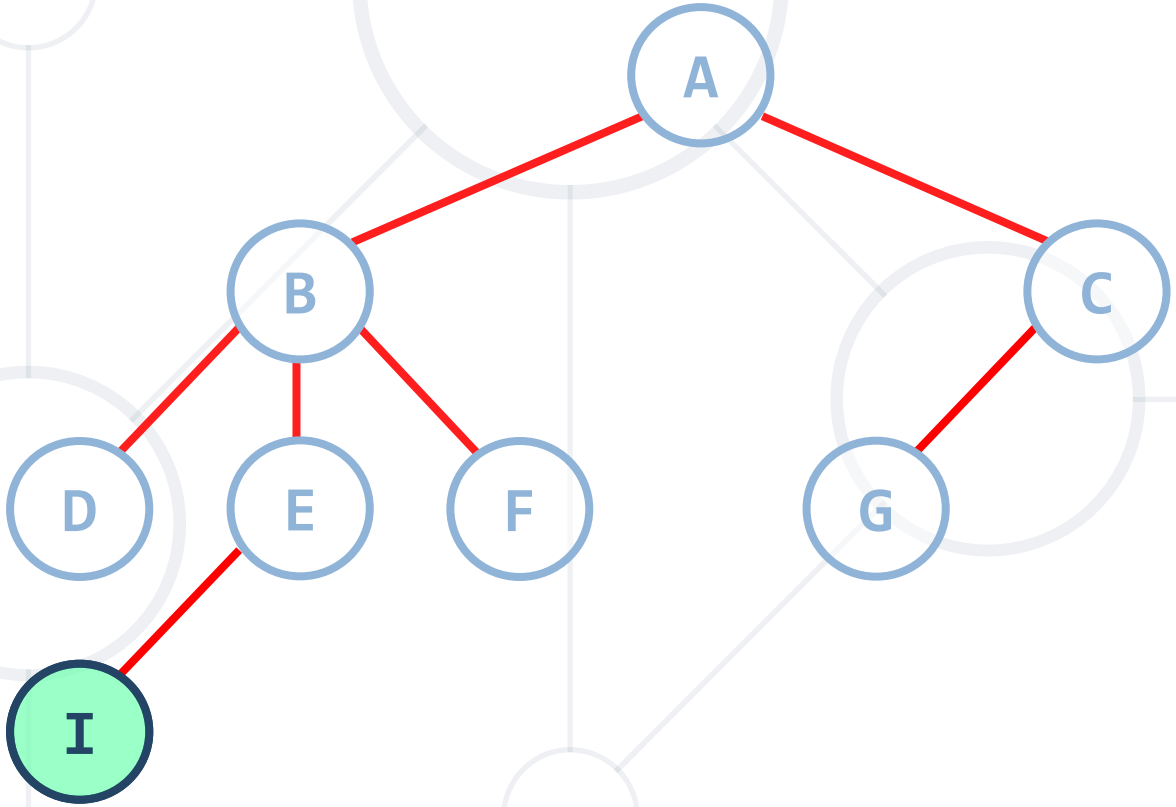
BFS Visualization

- Queue: E F G
- Output: A B C D



BFS Visualization

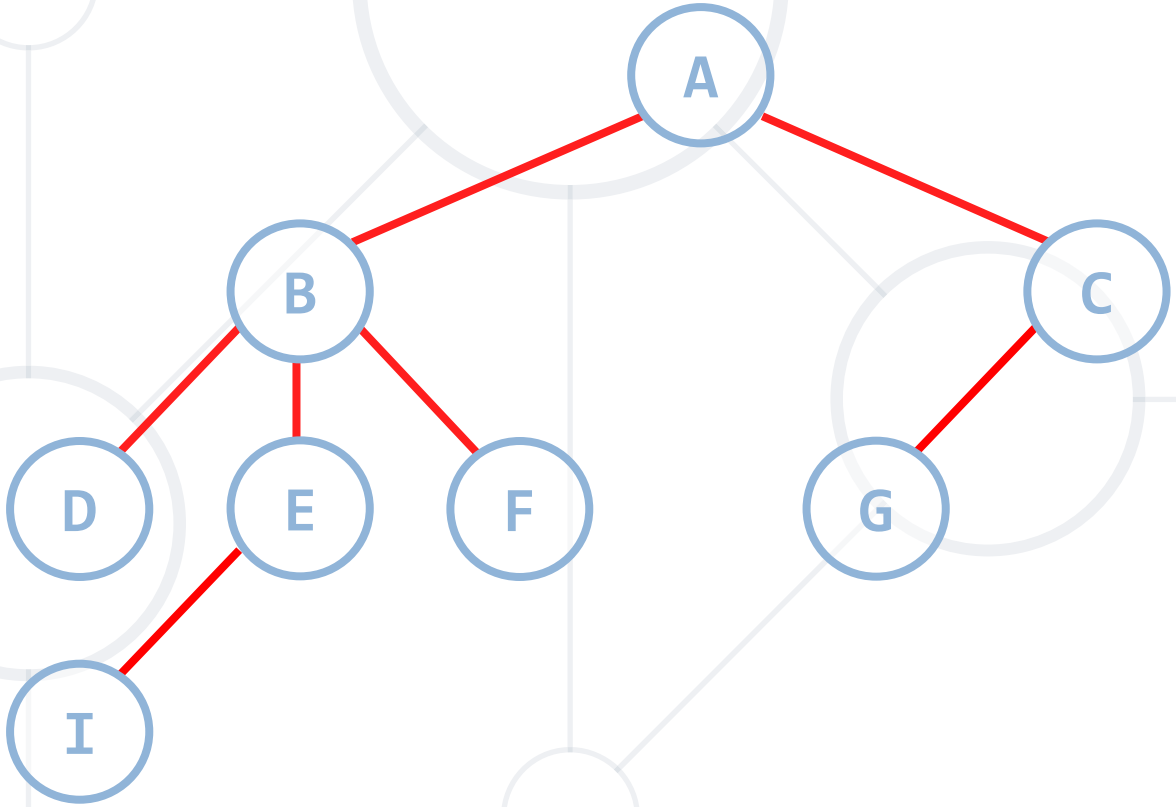
- Queue: F G
- Output: A B C D E



BFS Visualization

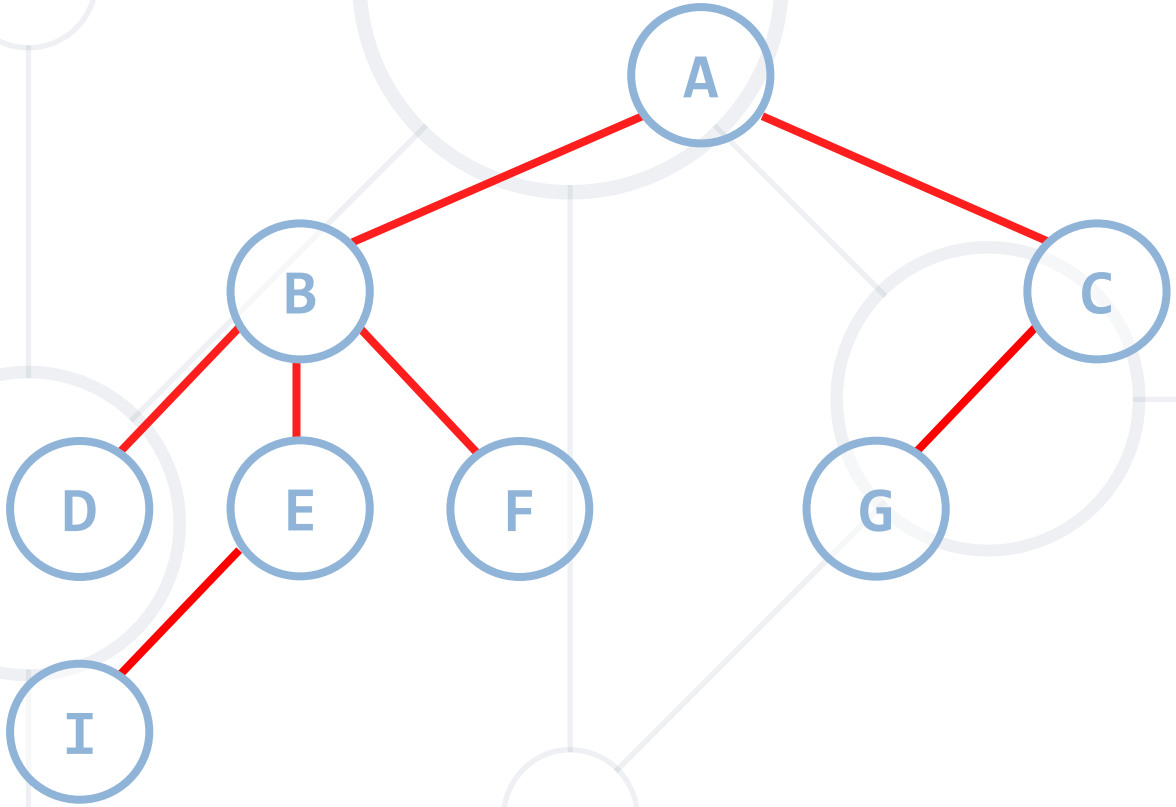
- Queue: F G I
- Output: A B C D E

We continue the steps until the queue is empty



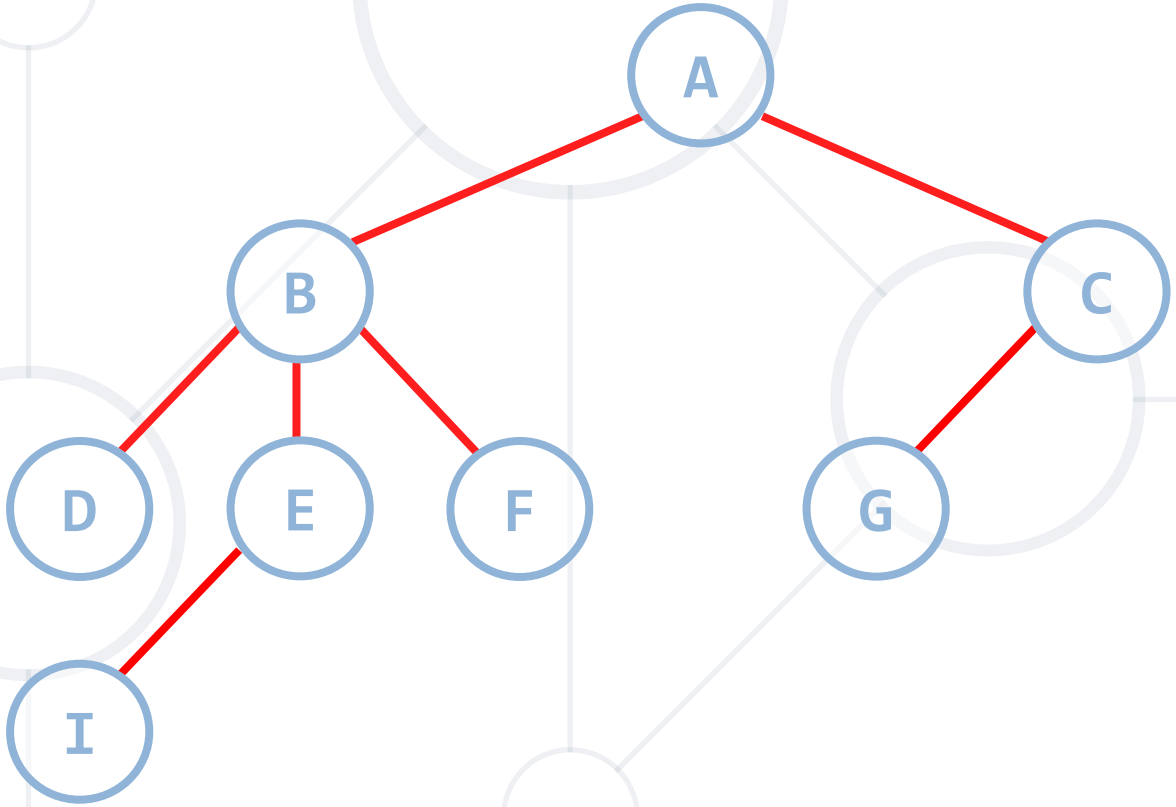
BFS Visualization

- Queue: G I
- Output: A B C D E F



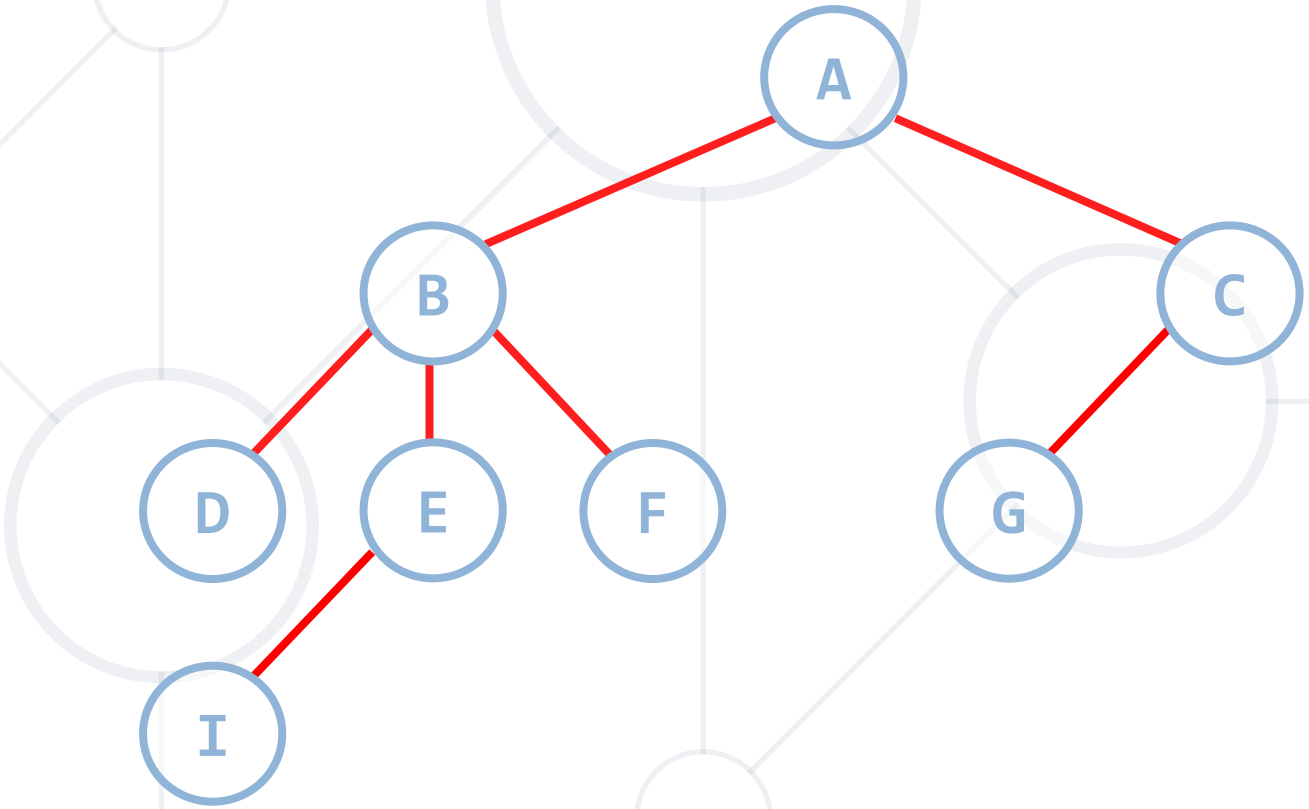
BFS Visualization

- Queue: I
- Output: A B C D E F G



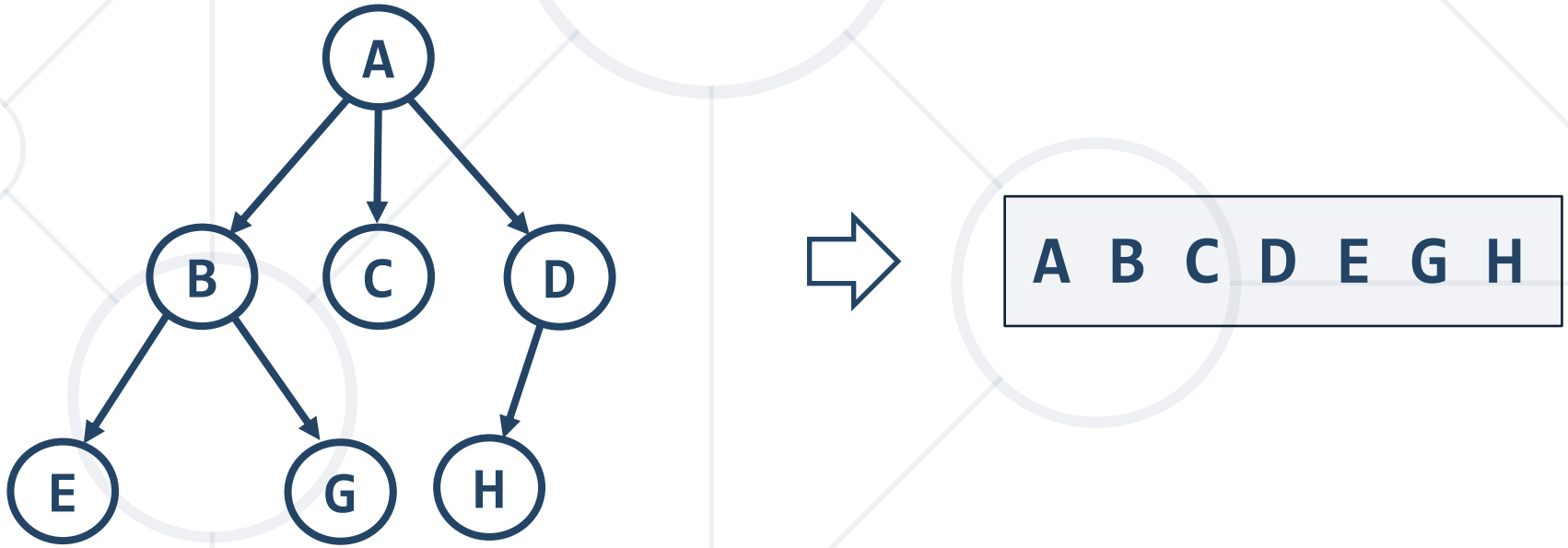
BFS Visualization

- Queue:
- Output: A B C D E F G I



Problem: Order BFS

- Given the **Tree<T>** structure, define a method
  - **IEnumerable<T> OrderBfs()**



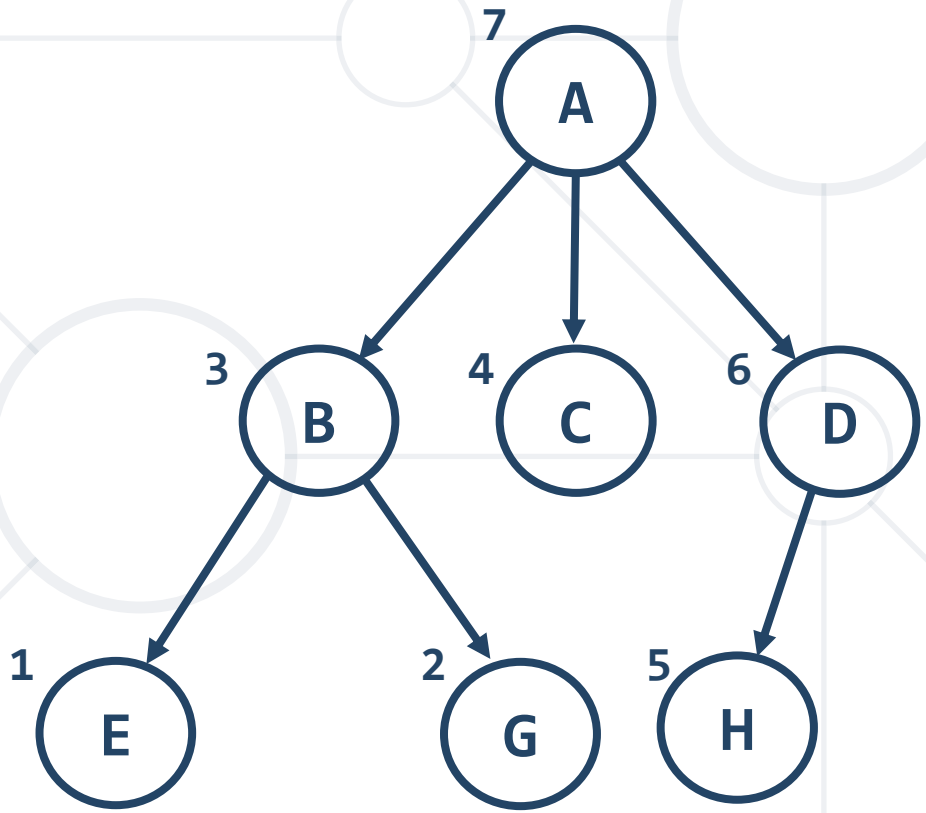
Solution: Order BFS

```
public IEnumerable<T> OrderBfs()
{
  var result = new List<T>();
  var queue = new Queue<Tree<T>>();
  queue.Enqueue(this);
  while (queue.Count > 0)
  {
    // To Do: Implement this part
  }
  return result;
}
```

Depth-First Search (DFS)

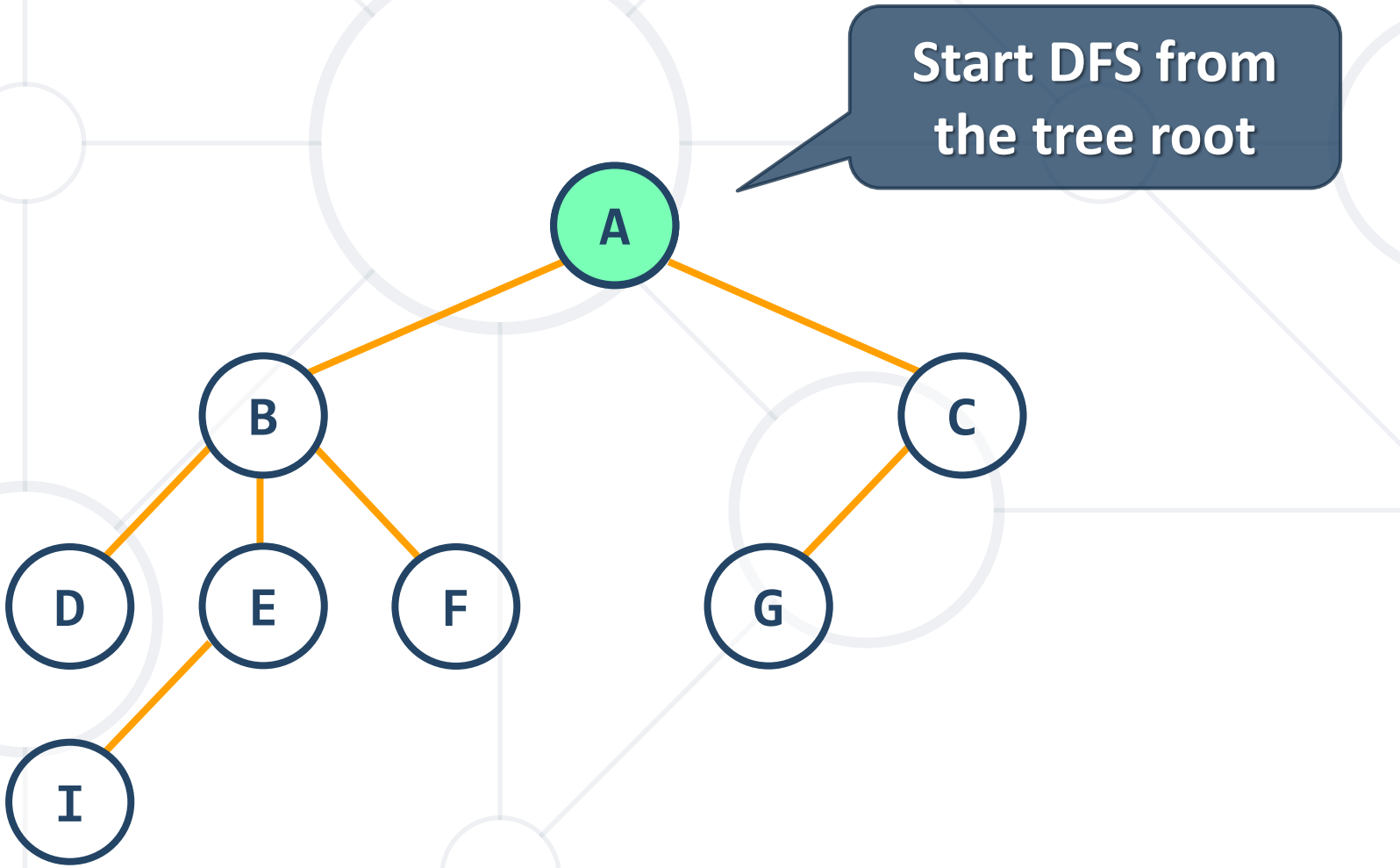
- **Depth-First Search** (**DFS**) first visits all descendants of given node recursively, finally visits the node itself or vice-versa
- DFS algorithm pseudo code:

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



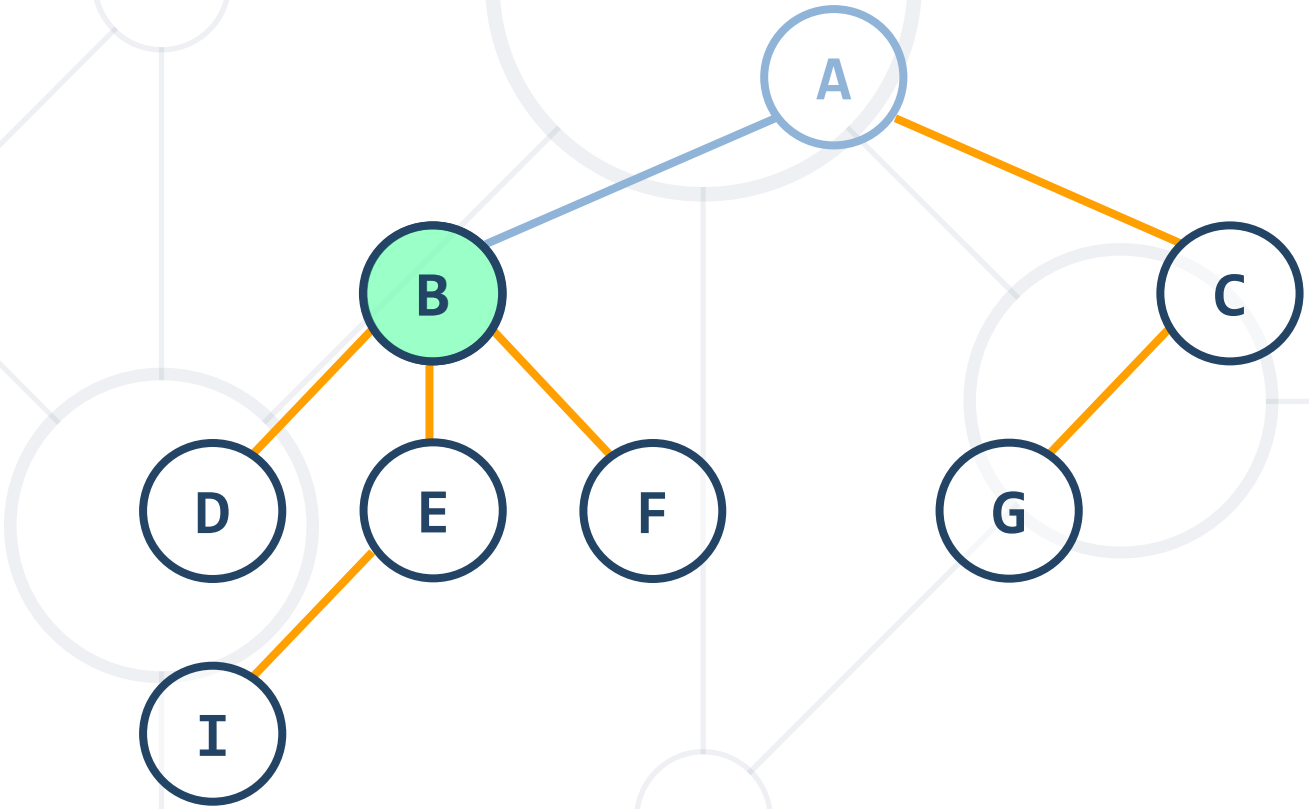
DFS Visualization

- Stack:
- Output:



DFS Visualization

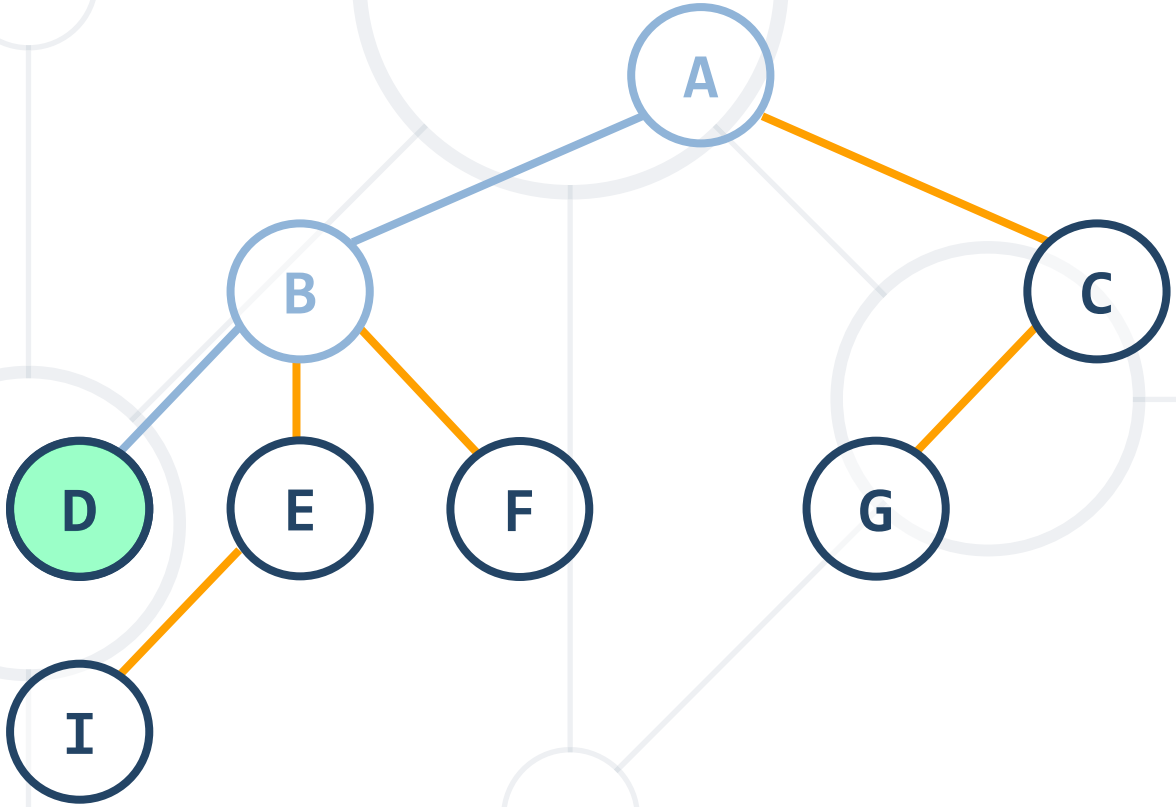
- Stack: A
- Output:



Enter recursively into the first child

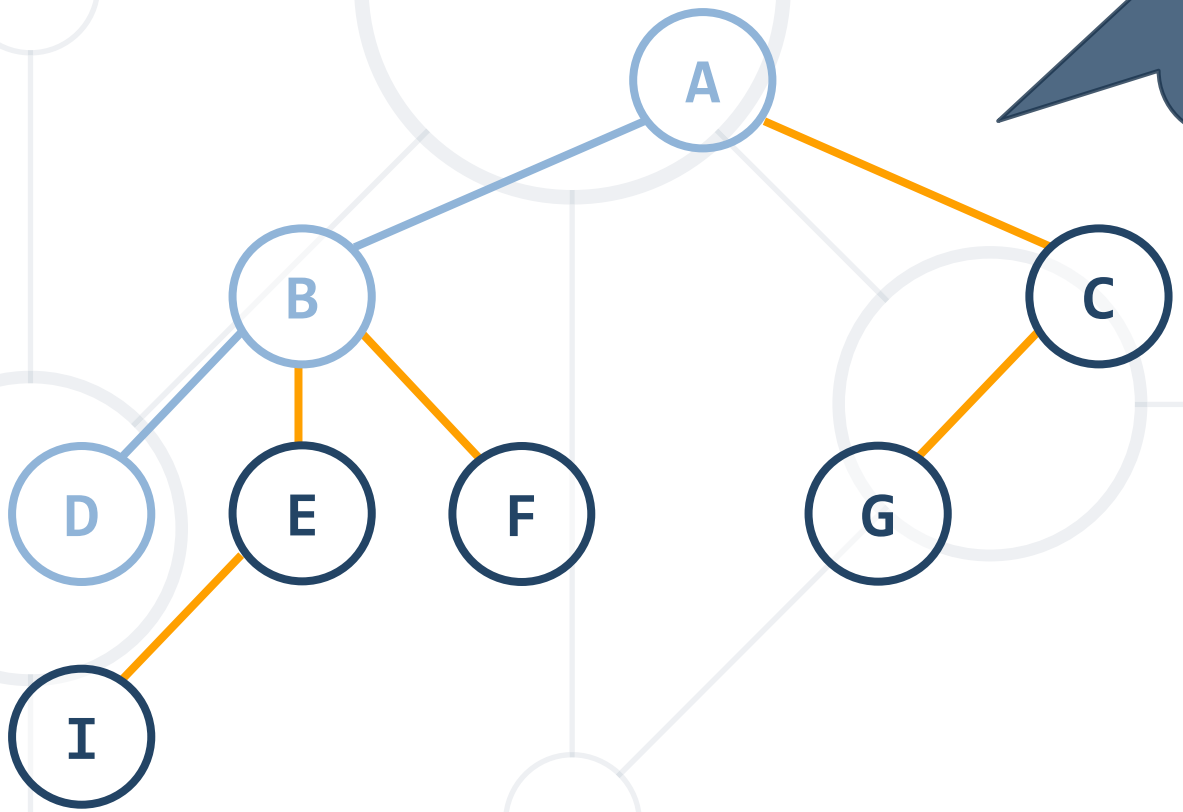
DFS Visualization

- Stack: A B
- Output:



DFS Visualization

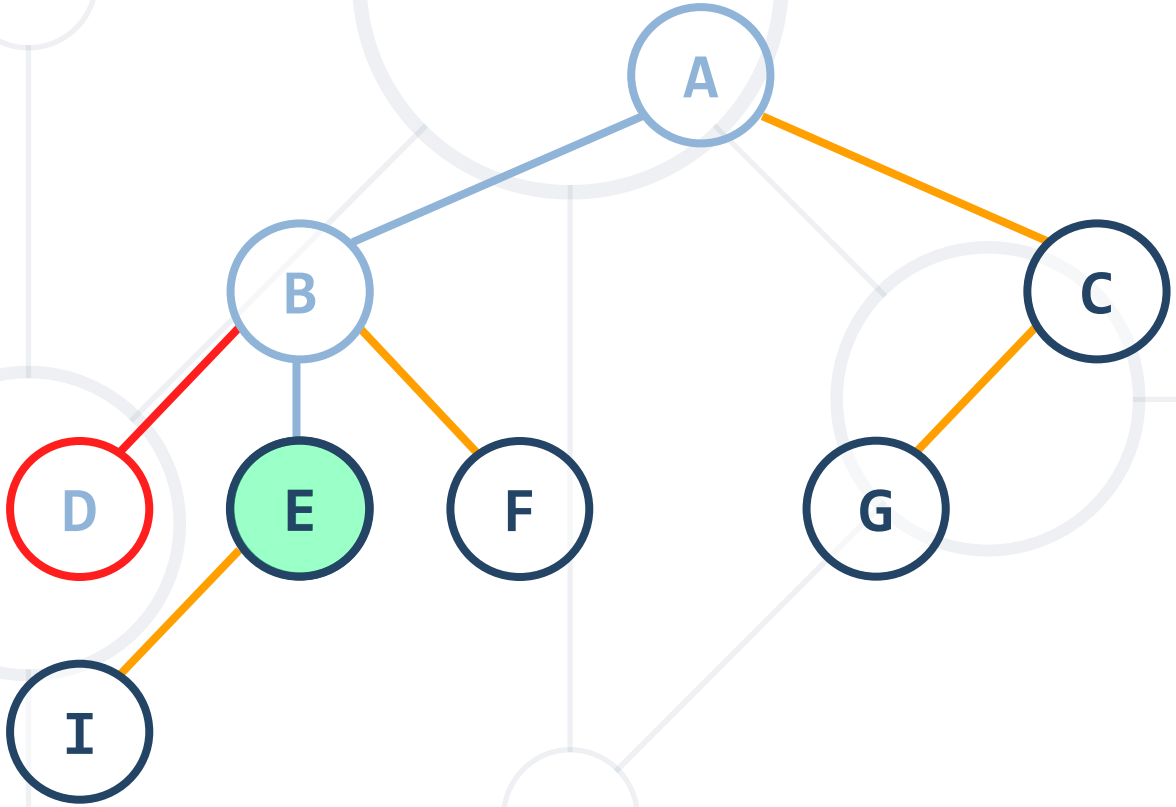
- Stack: A B D
- Output:



"D" has no children, so print it and go back to the other children of "B"

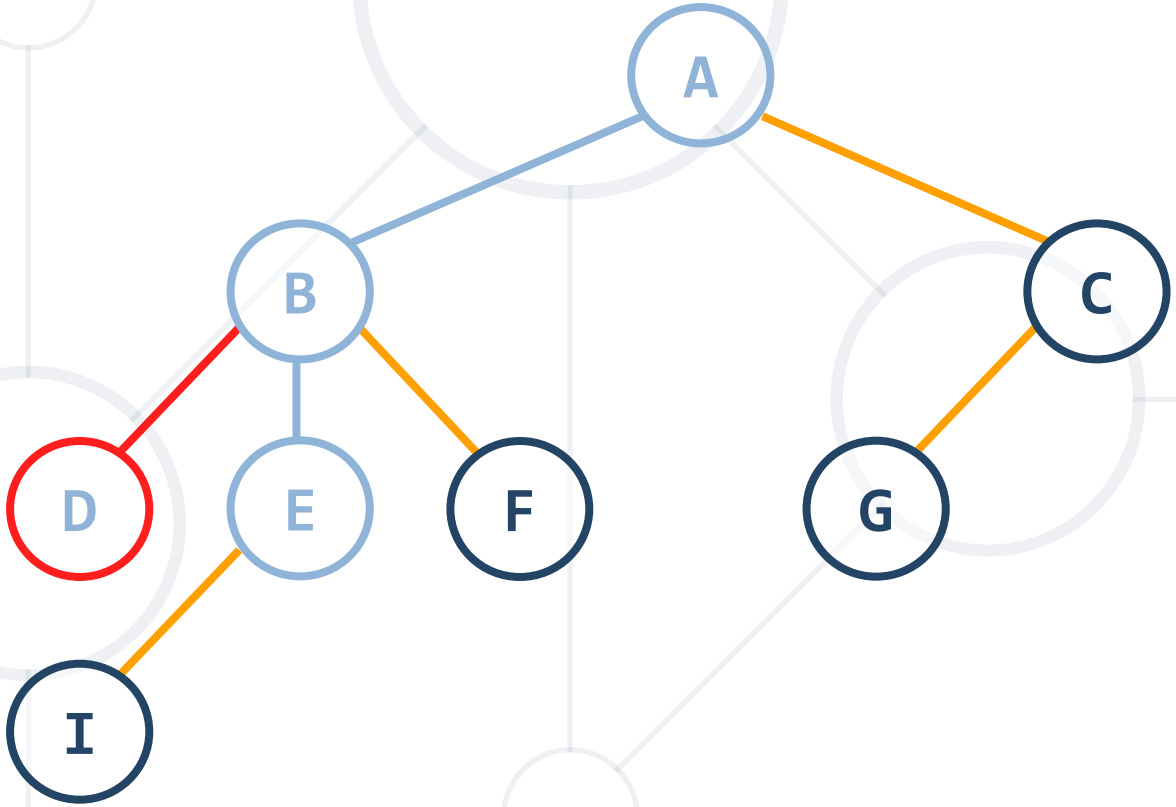
DFS Visualization

- Stack: A B
- Output: D



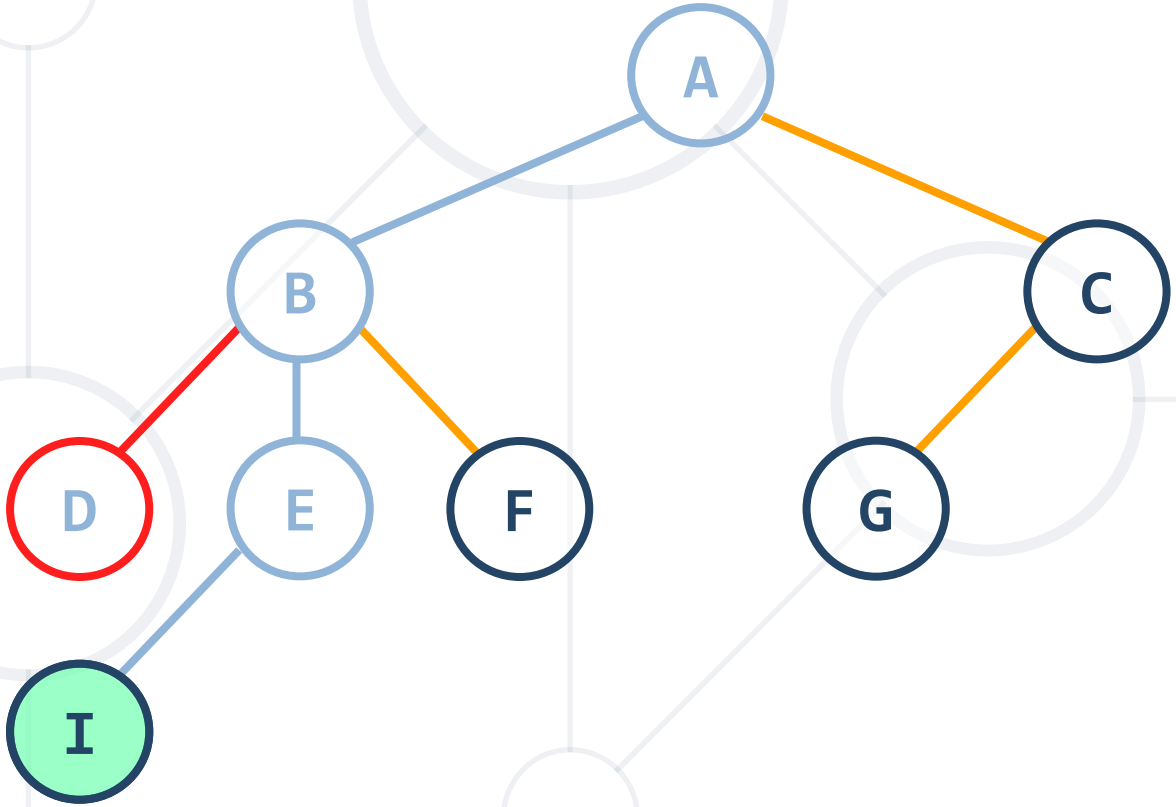
DFS Visualization

- Stack: A B E
- Output: D



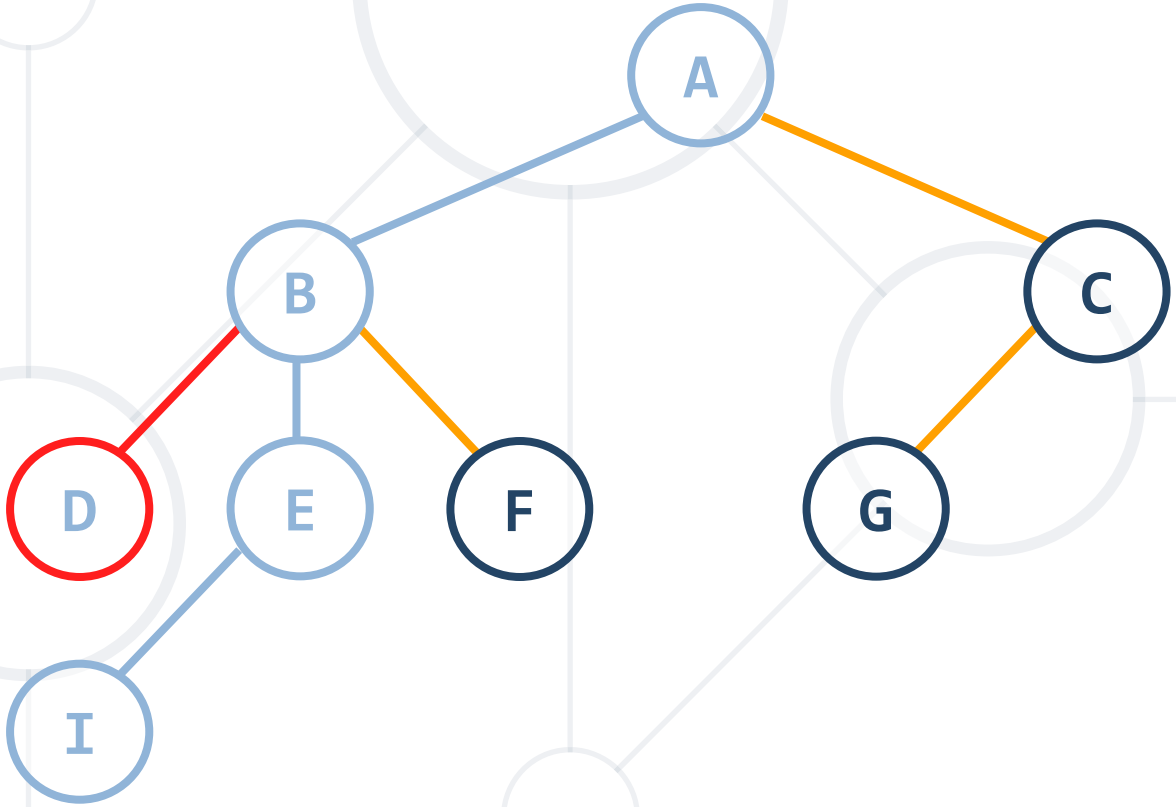
DFS Visualization

- Stack: A B E
- Output: D



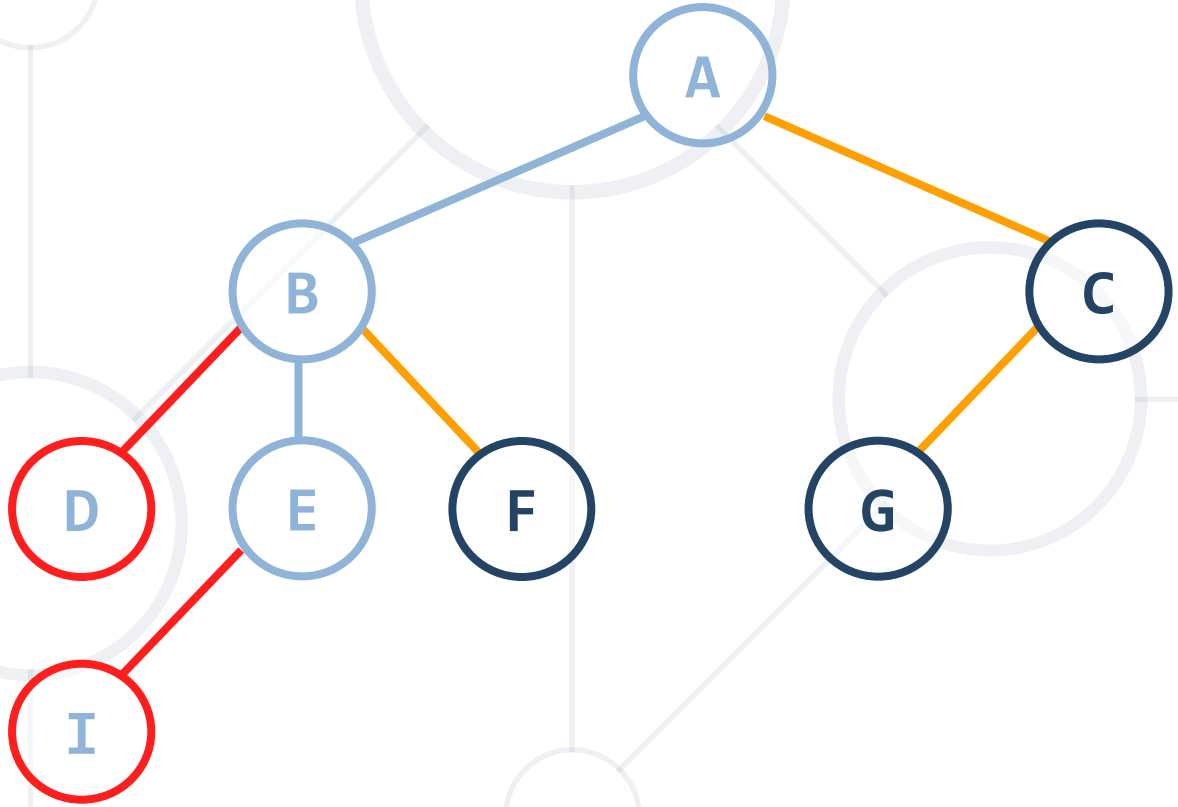
DFS Visualization

- Stack: A B E I
- Output: D



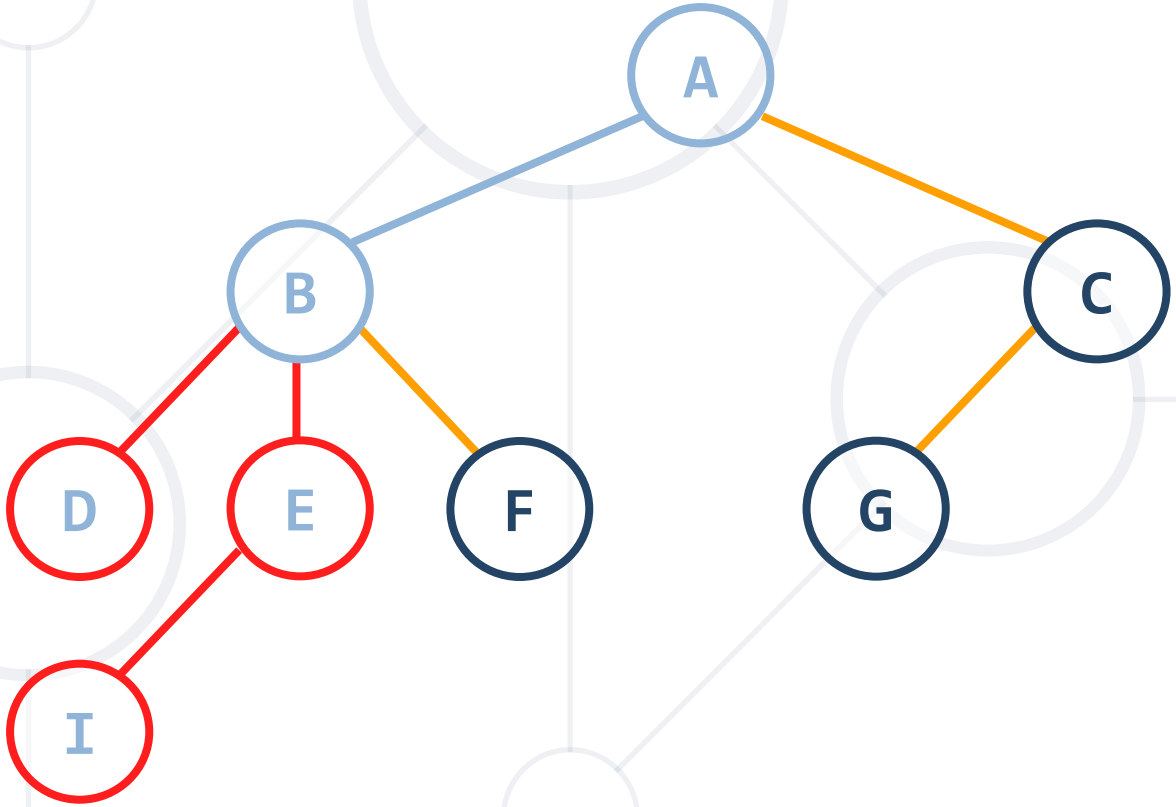
DFS Visualization

- Stack: A B E
- Output: D I



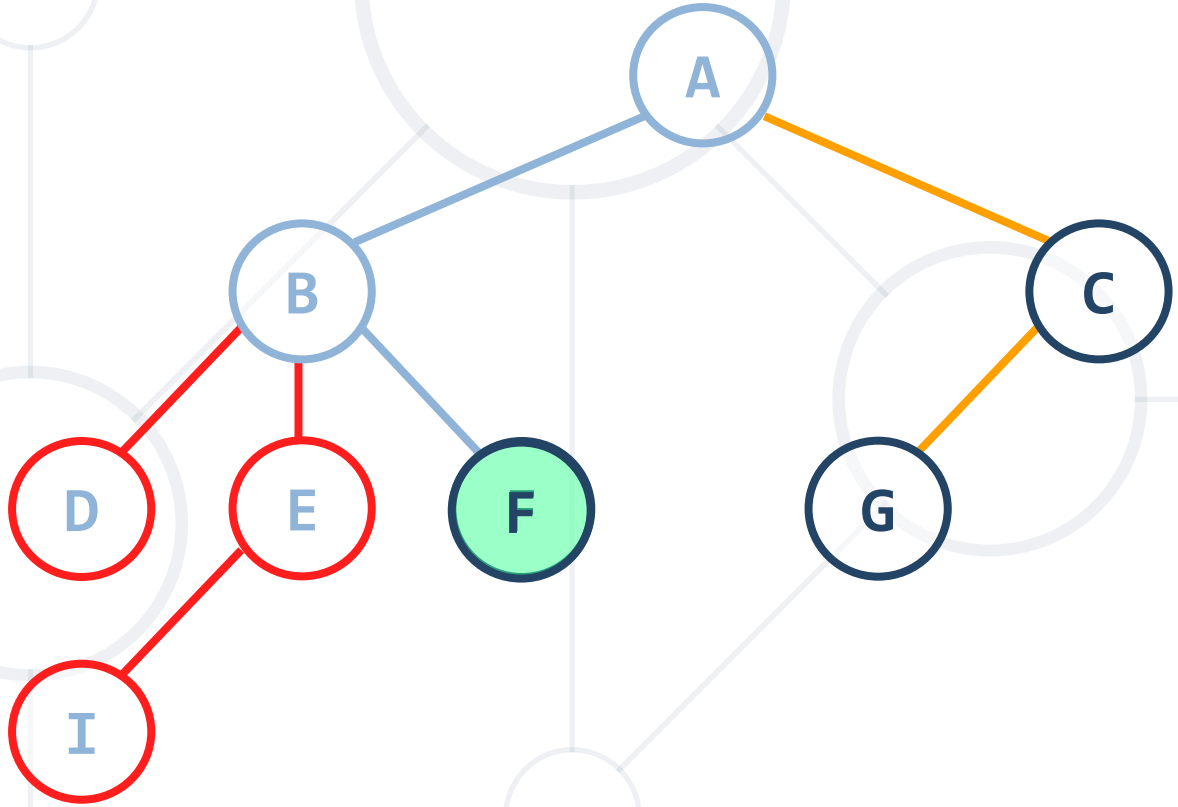
DFS Visualization

- Stack: A B
- Output: D I E



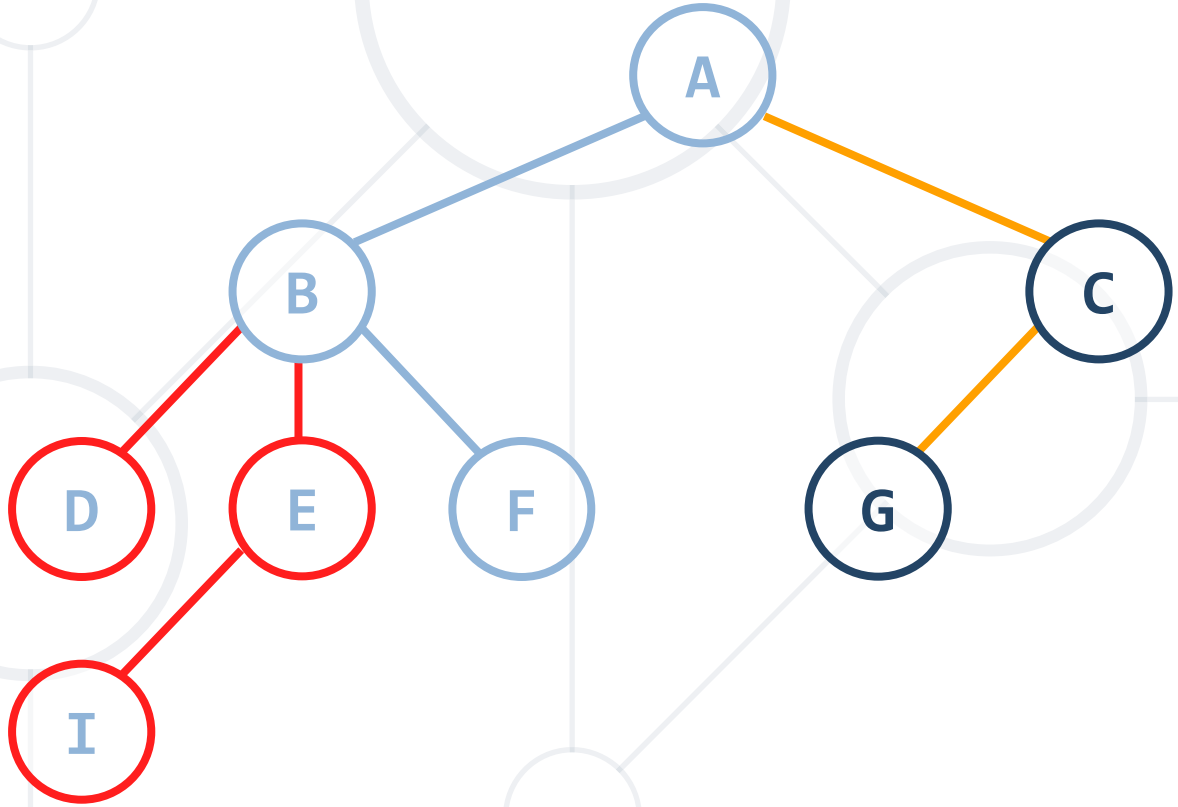
DFS Visualization

- Stack: A B
- Output: D I E



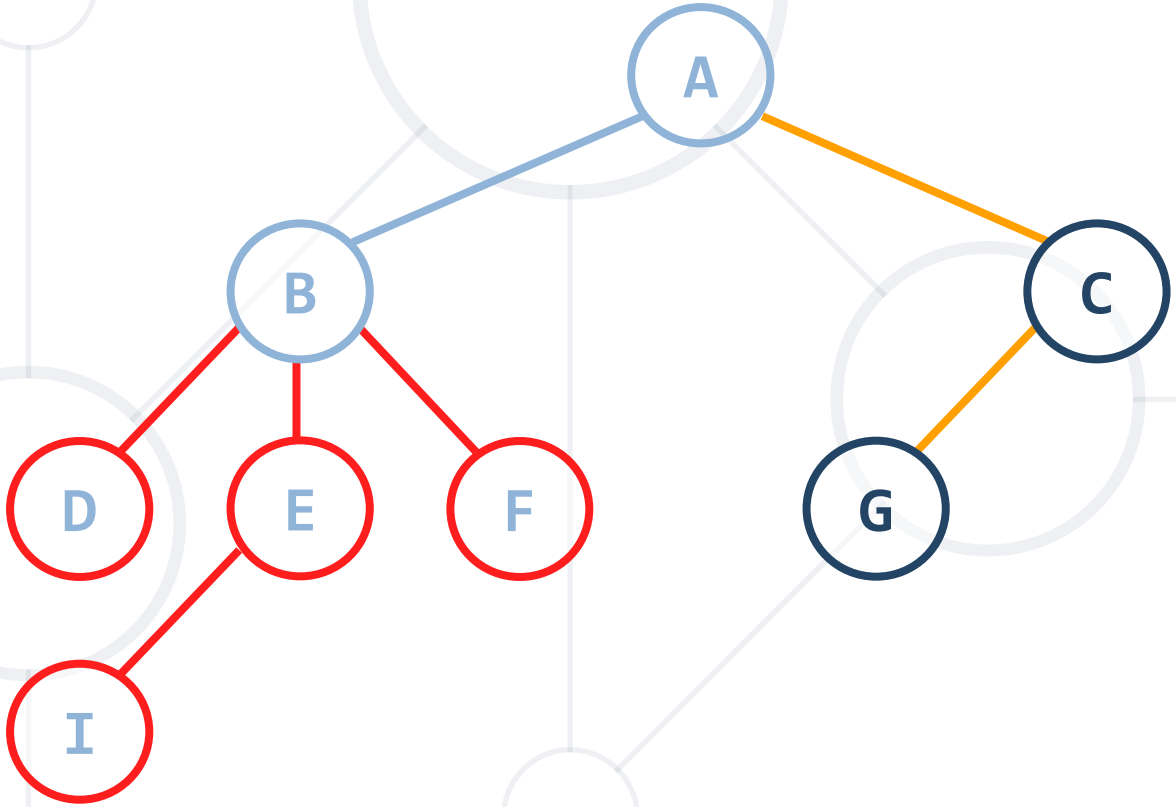
DFS Visualization

- Stack: A B F
- Output: D I E



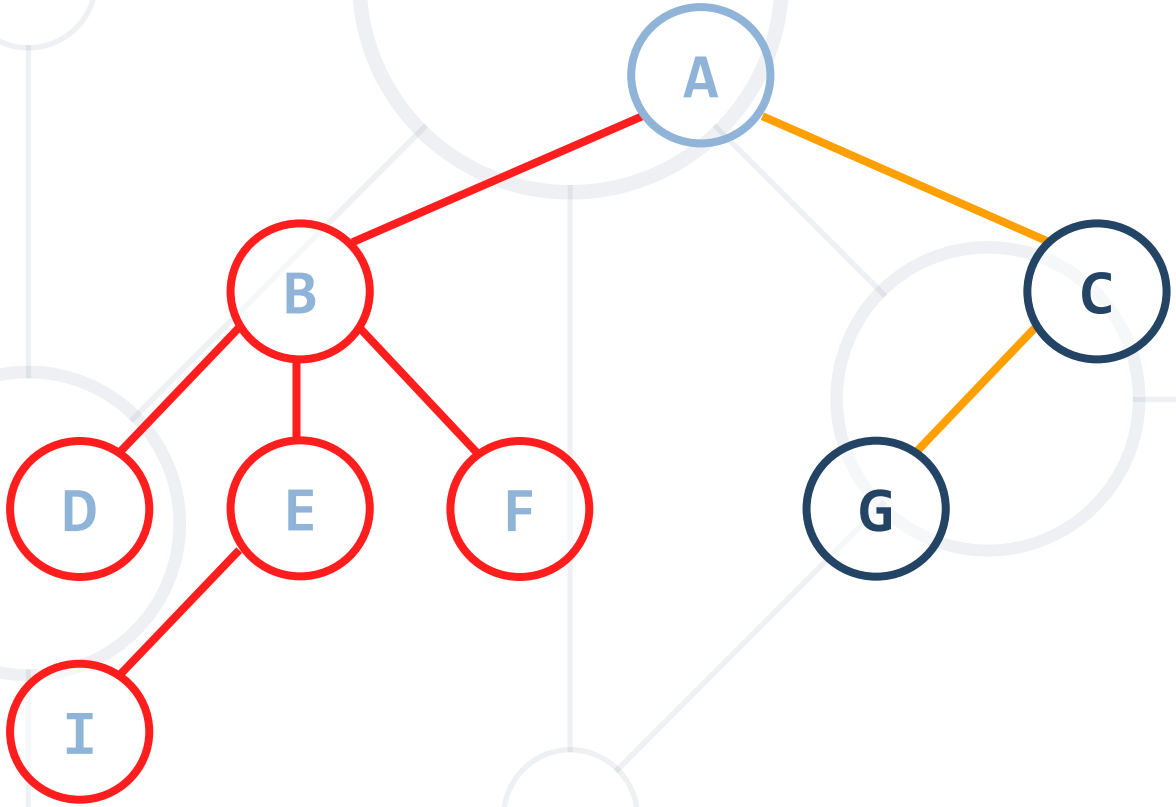
DFS Visualization

- Stack: A B
- Output: D I E F



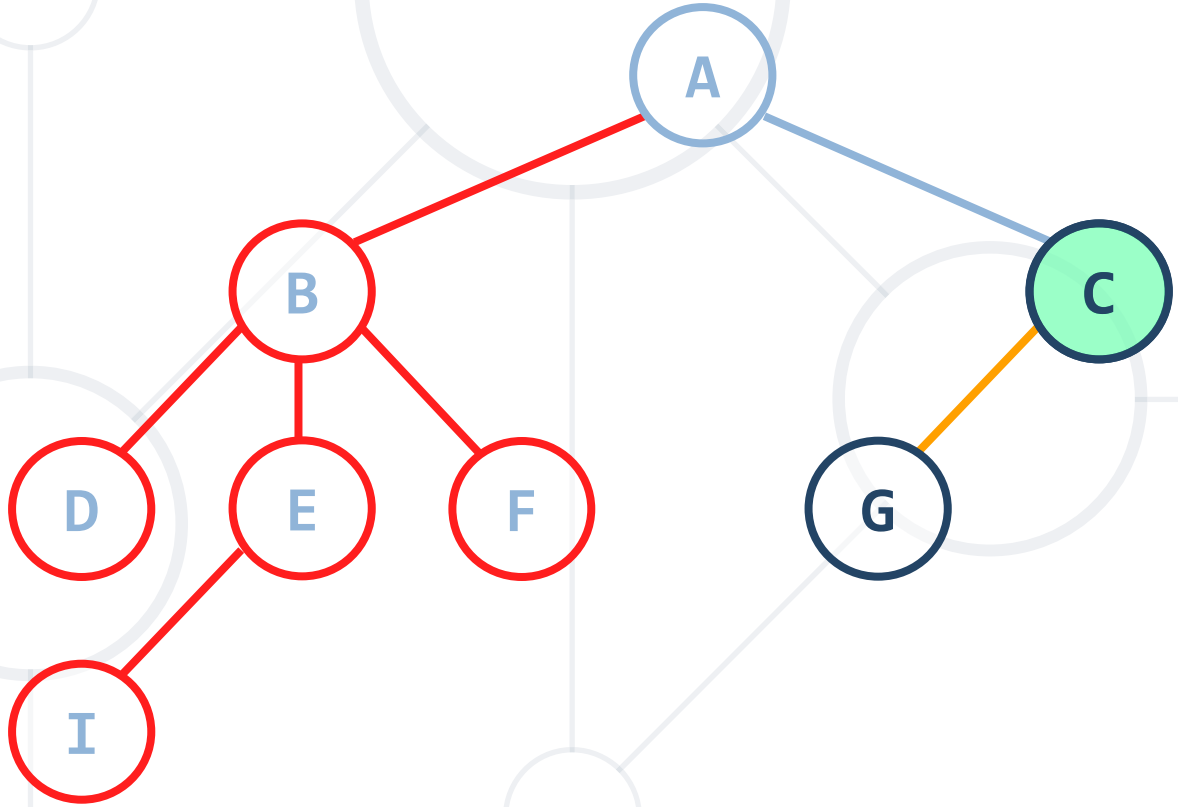
DFS Visualization

- Stack: A
- Output: D I E F B



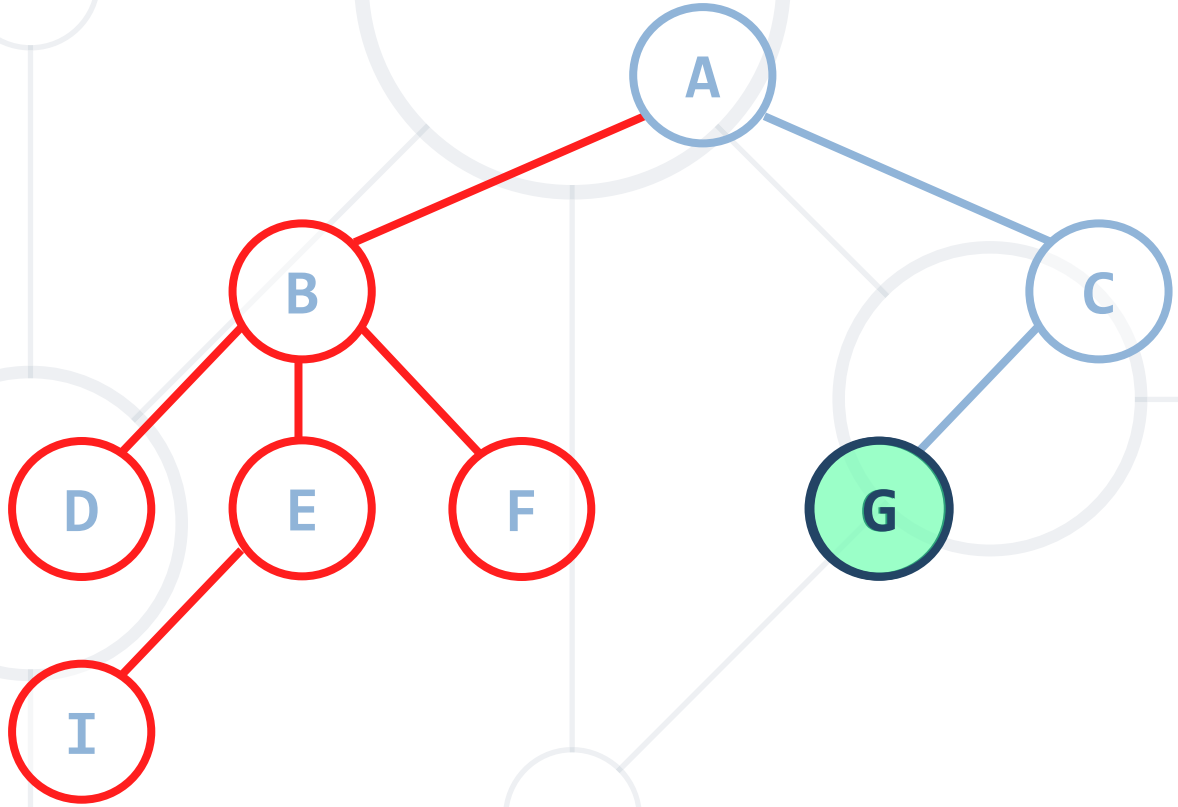
DFS Visualization

- Stack: A
- Output: D I E F B



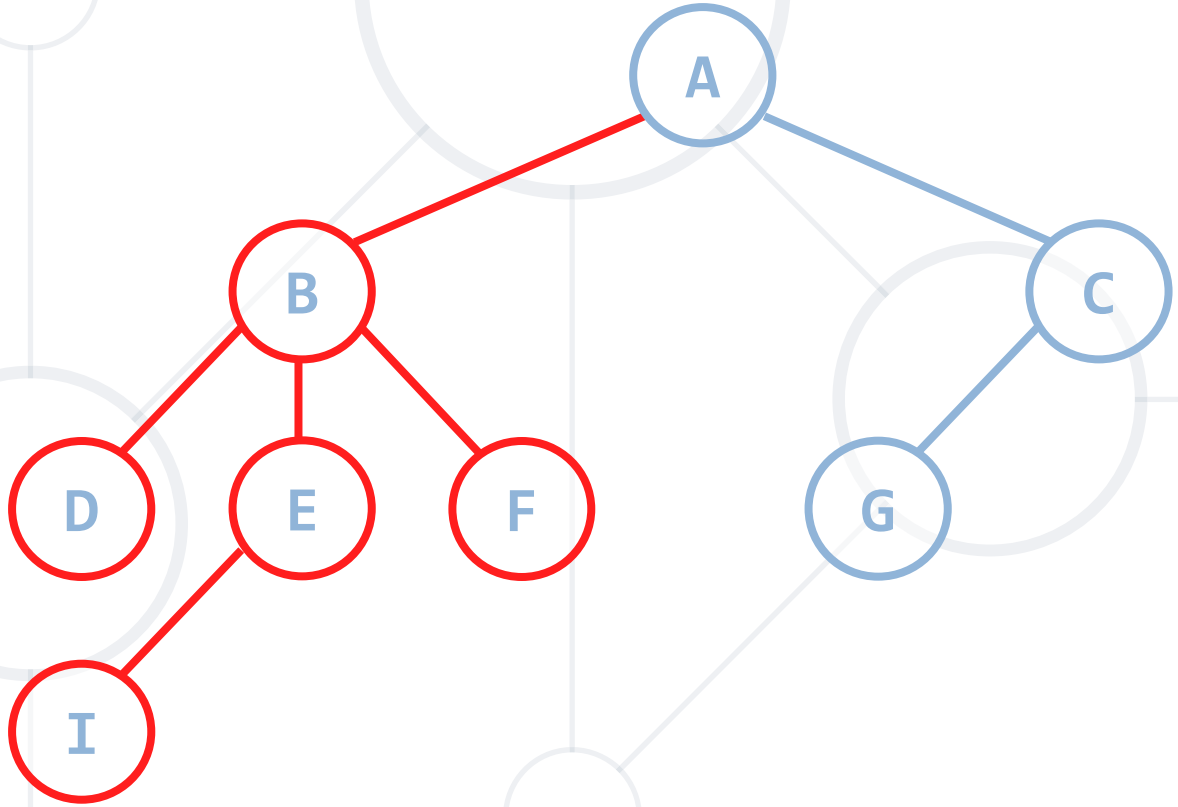
DFS Visualization

- Stack: A C
- Output: D I E F B



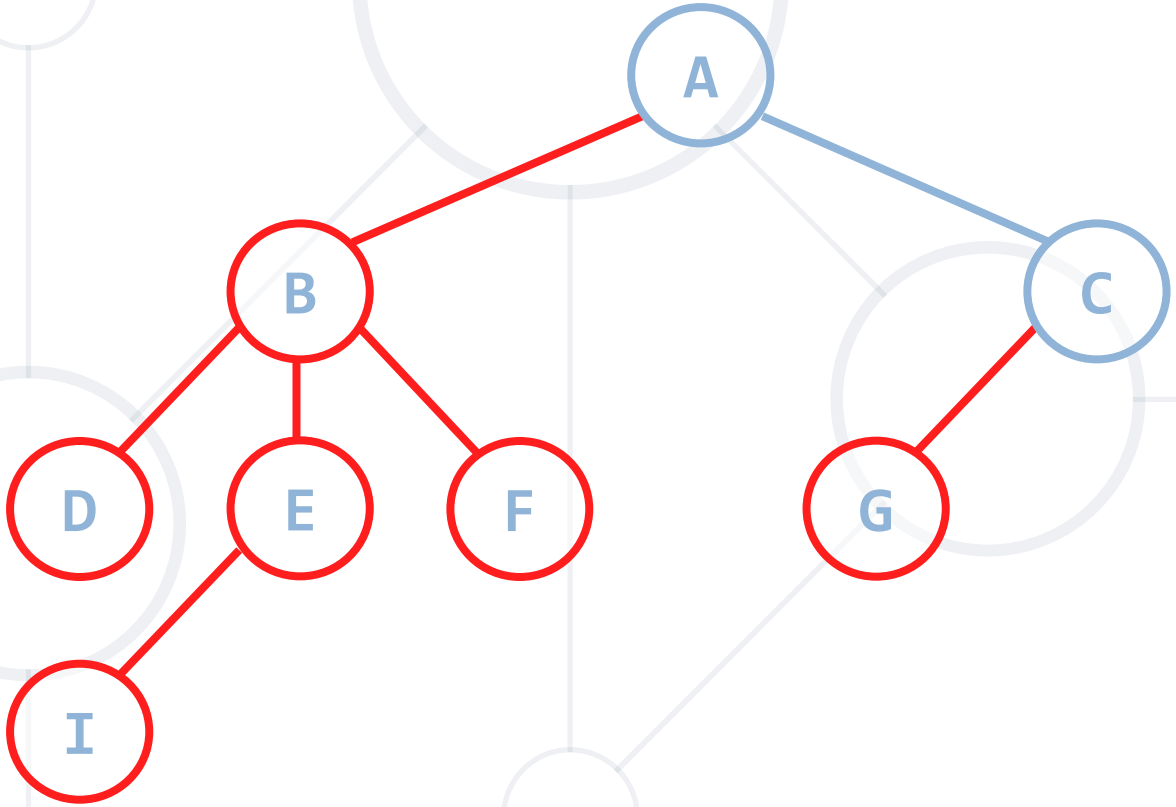
DFS Visualization

- Stack: A C G
- Output: D I E F B



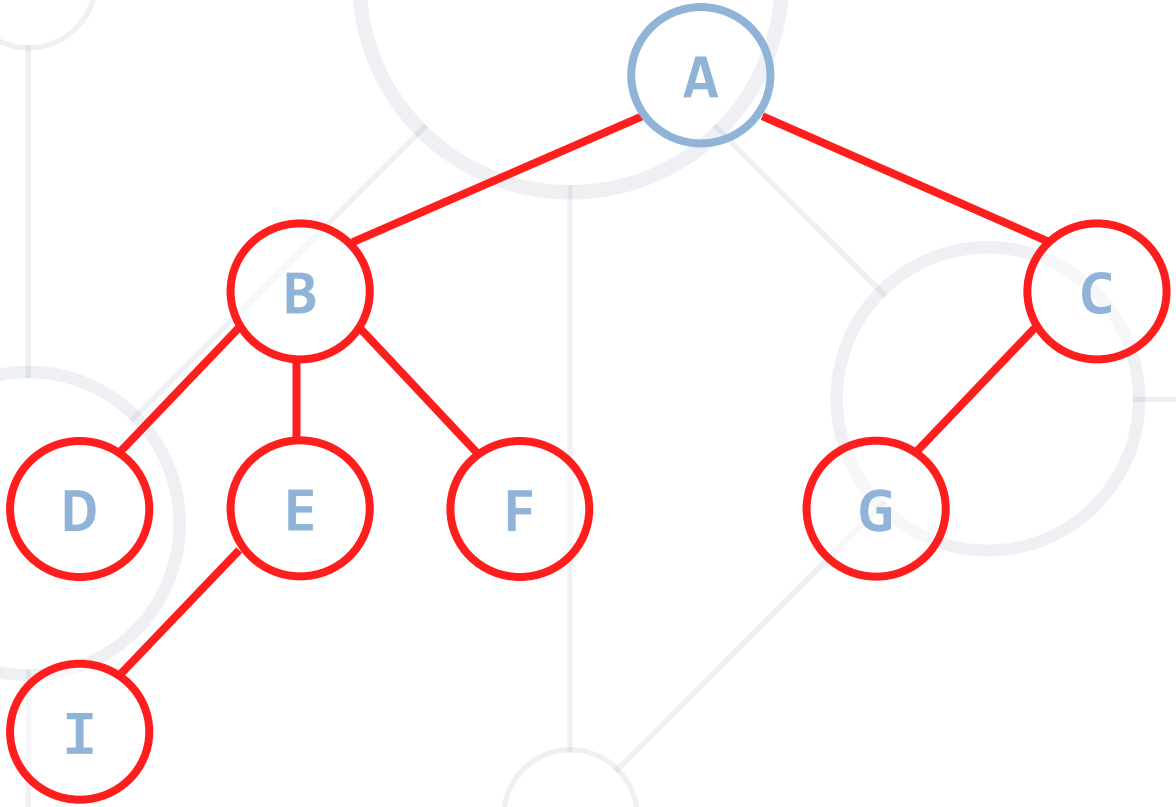
DFS Visualization

- Stack: A C
- Output: D I E F B G



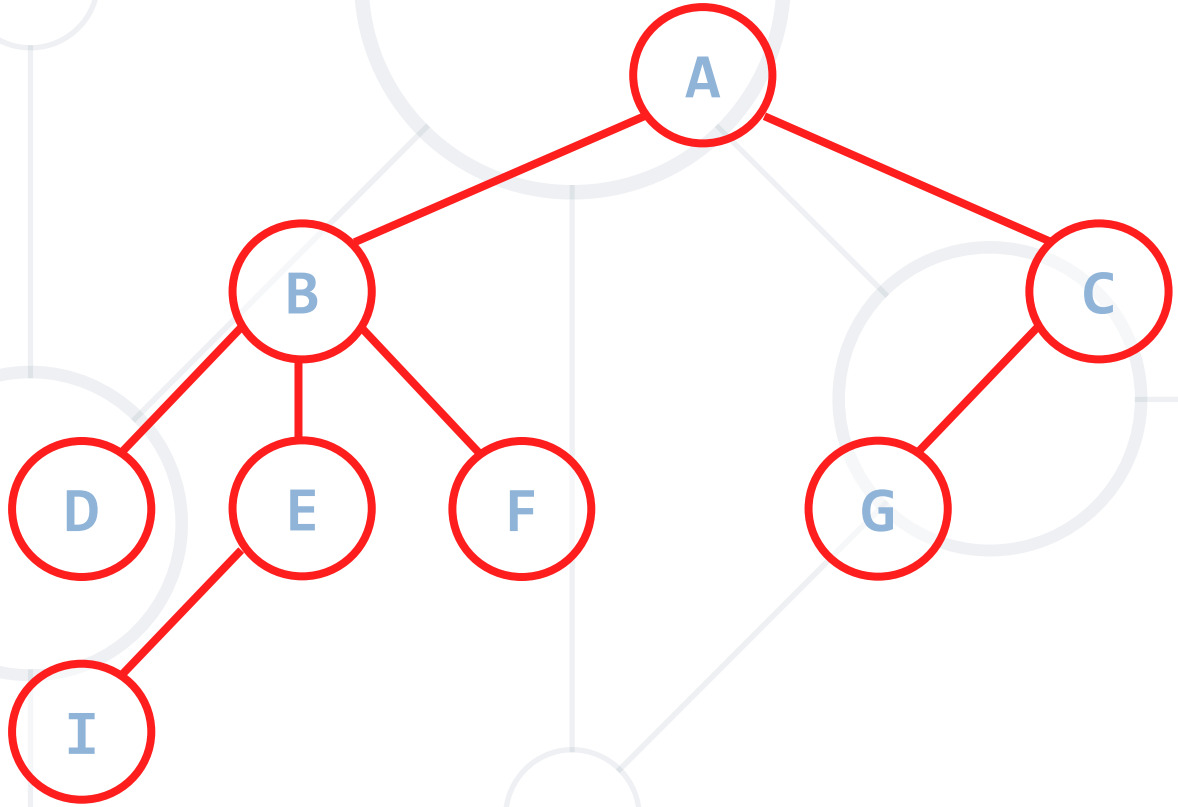
DFS Visualization

- Stack: A
- Output: D I E F B G C



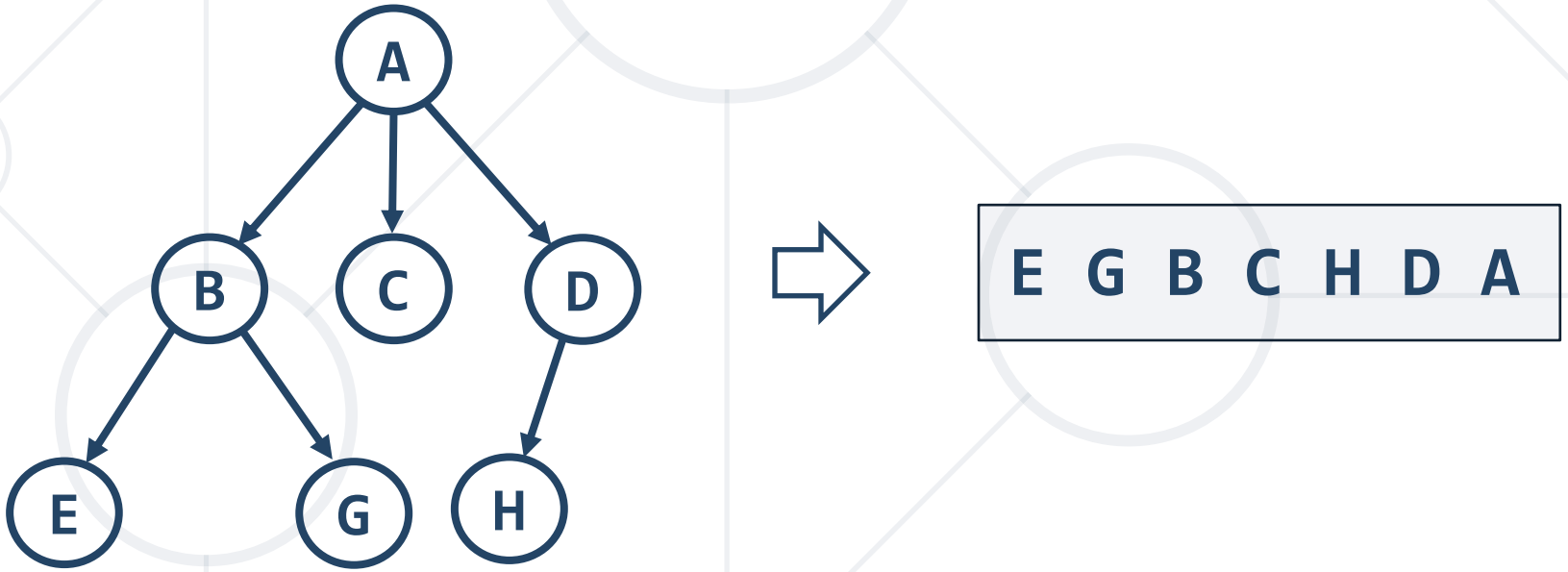
DFS Visualization

- Stack:
- Output: D I E F B G C A



Problem: Order DFS

- Given the **Tree<T>** structure, define a method
  - **IEnumerable<T> OrderDfs()**



Solution: Order DFS with Stack

```
public IEnumerable<T> OrderDfs()
{
  var result = new Stack<T>();
  var stack = new Stack<Tree<T>>();
  stack.Push(this);
  while (stack.Count > 0)
  {
    // To Do: Implement this part
  }
  return result;
}
```

Solution: Recursive Order DFS

```
public IEnumerable<T> OrderDfs()
{
  var order = new List<T>();
  this.Dfs(this, order);
  return order;
}

private void Dfs(Tree<T> tree, List<T> order)
{
  // To Do: Implement
}
```

Conclusion

- What did we get so far?
  - Had we achieved any **better complexity**?
  - Are we working **with O(log(n))**?
- Well, the answer is…
  - **No!**
  - Why? Still, we are stuck at **linear complexity** for searching operations
- We will try to solve that with a **Binary Search Tree**

Summary

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

Questions?

SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

Finance Academy

SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

Flutter™ International

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

BOSCH

Postbank
Решения за твоето утре

PHAR VISION

SmartIT

DXC TECHNOLOGY

createX

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://about.softuni.bg/
- © Software University – https://softuni.bg

Trainings @ Software University (SoftUni)

- 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