SCC.121: Fundamentals of Computer Science
Sorting, Trees and Graphs

Tree (Abstract Data Type)

Today's Lecture

Lancaster University

Aim:

- Introduce the **tree** and the corresponding **ADT (Abstract Data Type)**
  - Applications
  - (Mathematical) Properties
  - Operations of the tree ADT
- Describe several **tree traversals**

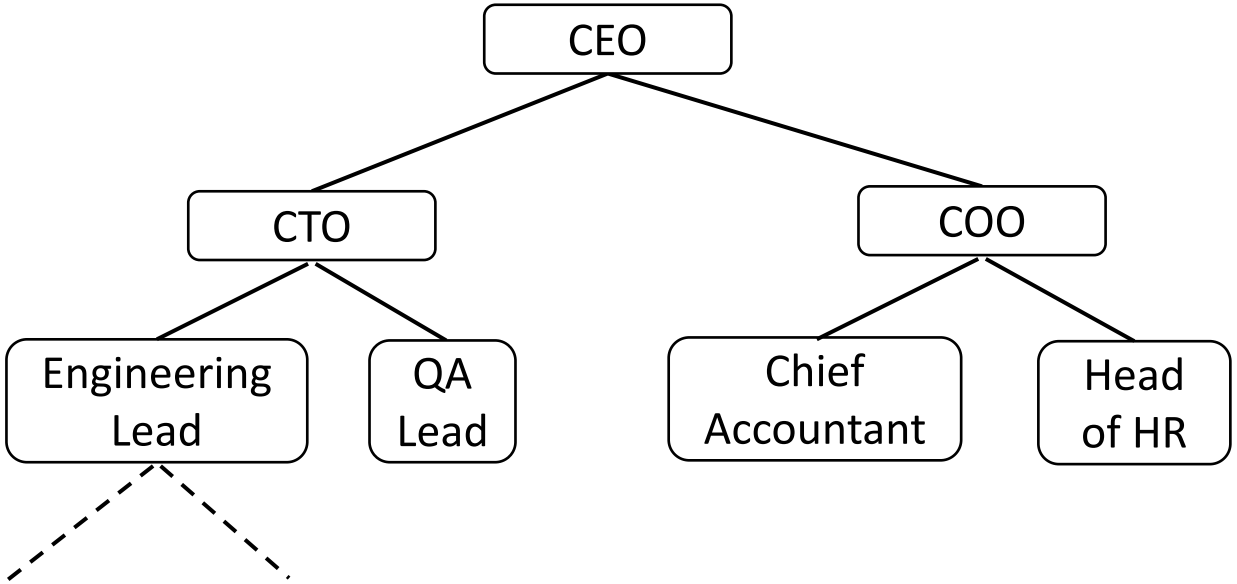
Tree ADT

- **Tree = nonlinear abstract data type** (ADT), **stores elements hierarchically**.

- Extremely popular ADT in software engineering projects.
  1. Matches our tendency to arrange most of our data and organisations hierarchically,
  2. Visually simple representation,
  3. Its operations can be made computationally efficient.
     - Adding or removing elements,
     - Finding an element,
     - Checking that the ADT satisfies some properties.

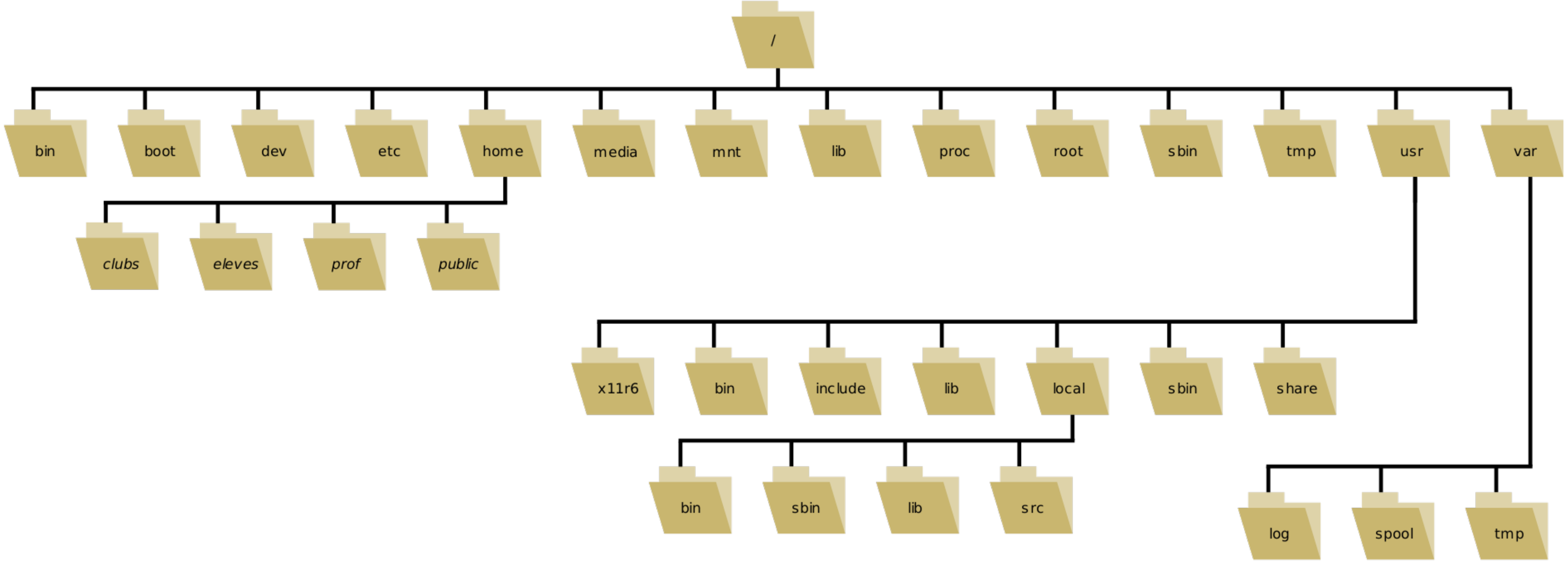
Applications for Trees

The **management structure of an organization** is typically represented by a tree



Applications for Trees

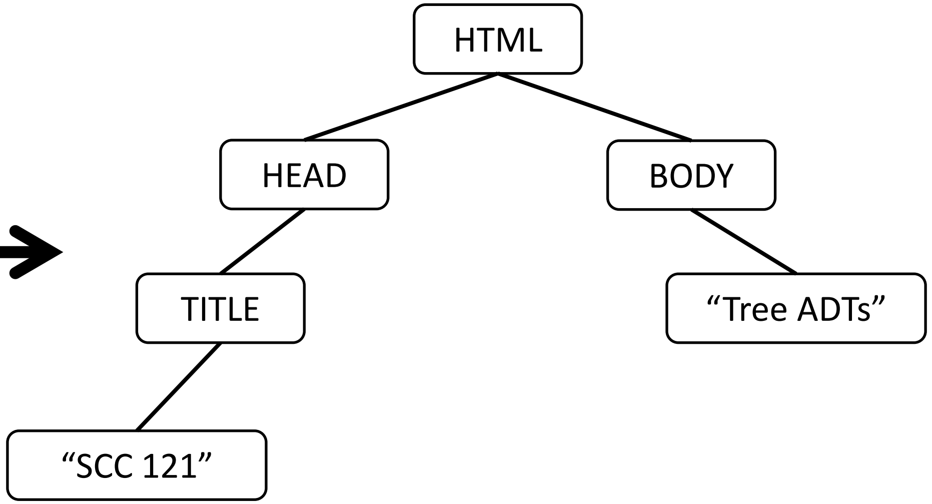
The **directory structure** is how an operating system (or computer) arrange the files you can access. These are typically displayed as trees.



Applications for Trees

Computer document formats like XML or HTML describe a tree of relationships between elements, and they're generally parsed into a tree ADT in programs.

```
<!DOCTYPE HTML>
<html>
<head>
  <title>SCC 121</title>
</head>
<body>
  Tree ADTs.
</body>
</html>
```

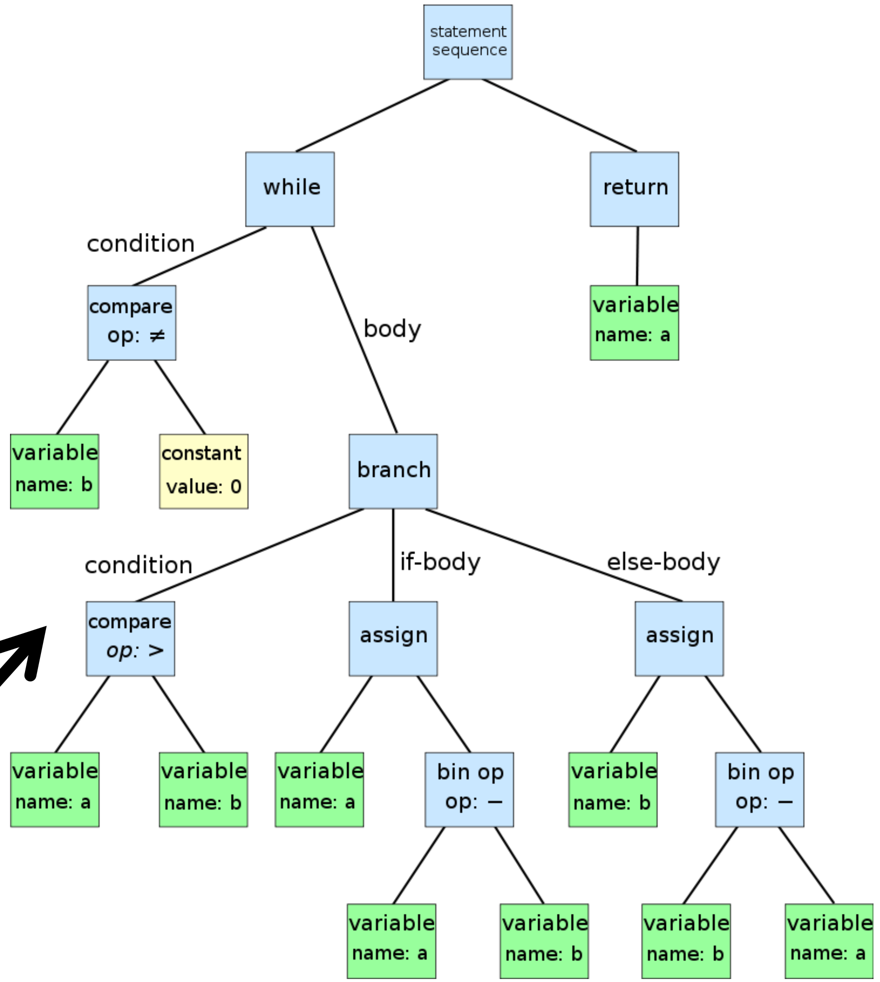


Applications for Trees

A compiler takes **source code**

```
Gcd(int a, int b) { //Euclidean algorithm
    while (b != 0){
        if (a > b) a = a - b
        else b = b - a
    }
    return a;
}
```

and converts it into an **abstract syntax tree**, to make it easy to evaluate the program in correct order.

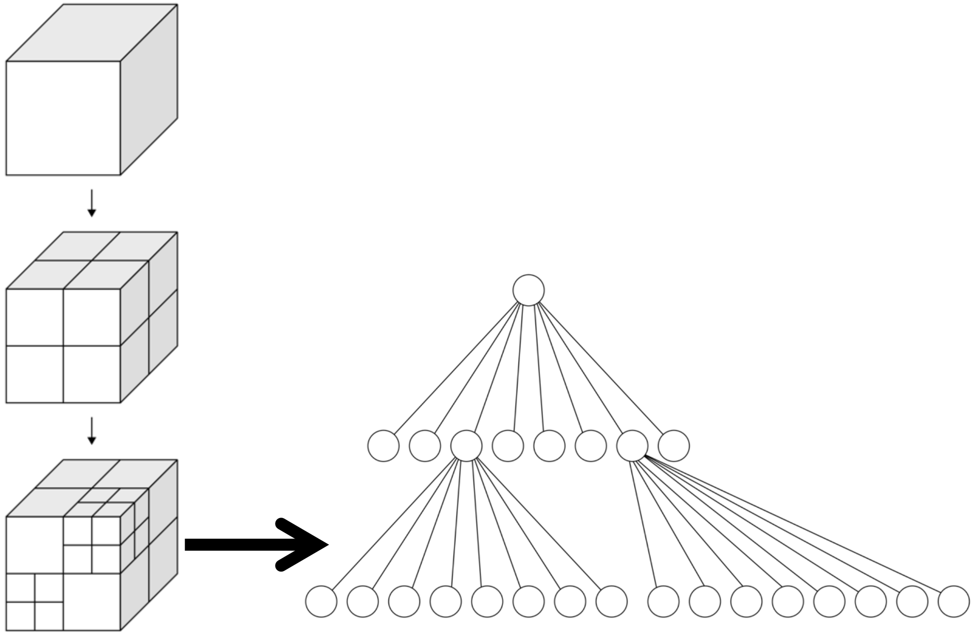


Applications for Trees

Physic simulations, image processing and game design use tree ADTs.

- For mesh generation (computer graphics),
- For collision detection.

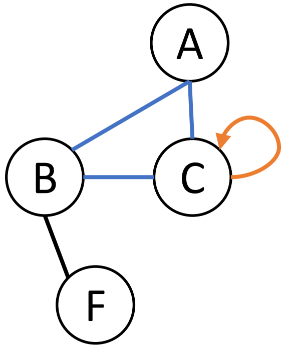
- Quadtrees (2D), **Octrees** (3D)
  - Octree = Hierarchical structure of 3D space,
  - Octree divides 3D space into eight regions,
  - Any region with > 1 points is subdivided, and so on.



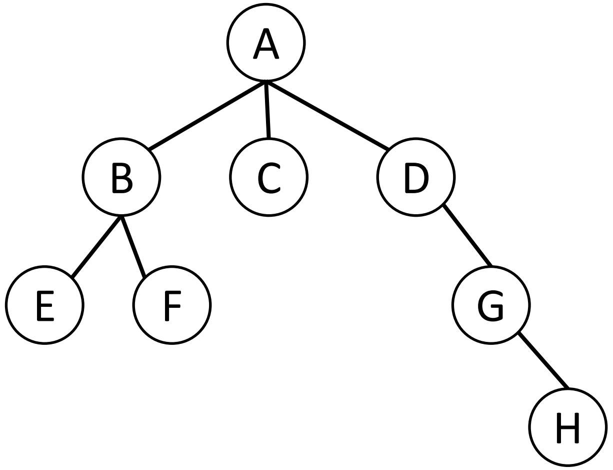
Definition of Trees, and Quick Tour of their Properties

Defining a Tree

- **Tree =** Nodes connected by edges (with no cycles or loops)



$\neq$

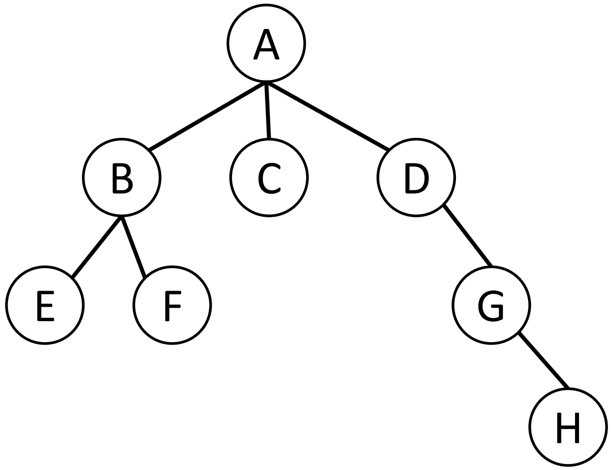


- Cycle
- Loop

Defining Nodes within a Tree

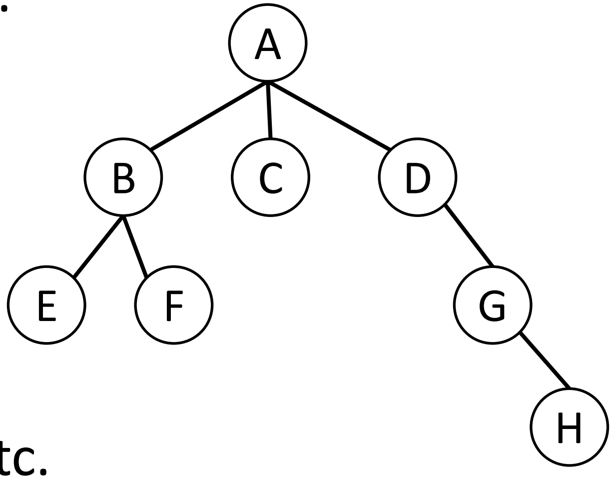
- <u>Hierarchical data structure:</u>
  - From **root** node,
  - To its **children**, and so on,
  - Until a **leaf** (node without children) is reached.

- <u>Examples</u>:
  - A is the root node,
  - E and F are children of B,
  - E, F, C and H are leaves.



Defining Relationships between Nodes

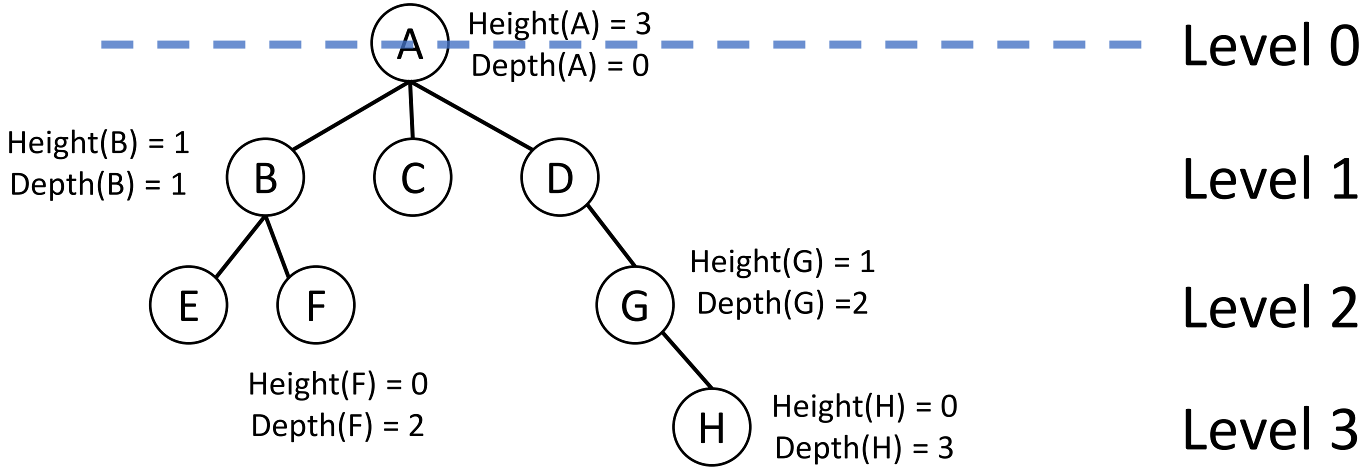
- **Descendants** of node v = children of v, and their children, etc.
  - G and H are the descendants of D
  - All nodes are descendants of the root.



- **Ancestor** of (non-root) node v = parent of v, and its parent, etc.
  - G, D and A are ancestors of H
  - B and A are the ancestors of F

Defining Distances in a Tree

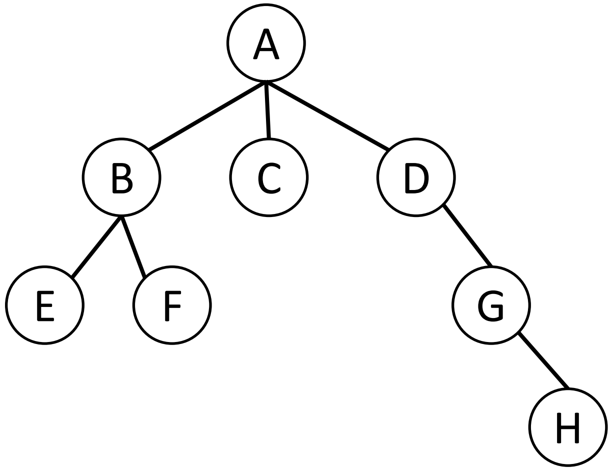
1. **Height of a node v** = number of nodes on longest path from v to some leaf.
2. **Depth of a node v** = number of nodes from v to root.
3. **Level $\boldsymbol{i}$** = all nodes at depth i.



Defining Distances in a Tree

Height of tree = height of its root, or also, depth of its deepest nodes.

Diameter (or **width**) **of tree** = number of nodes on longest path between any two leaves.

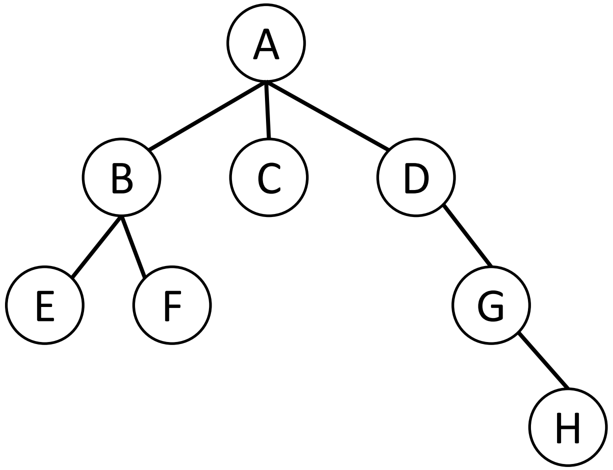


Height (Tree) = 3

Diameter(Tree) = 5

Trees have a Fixed Number of Edges

Take a tree with n nodes. Then, the **number of edges** in that tree is $n-1$.



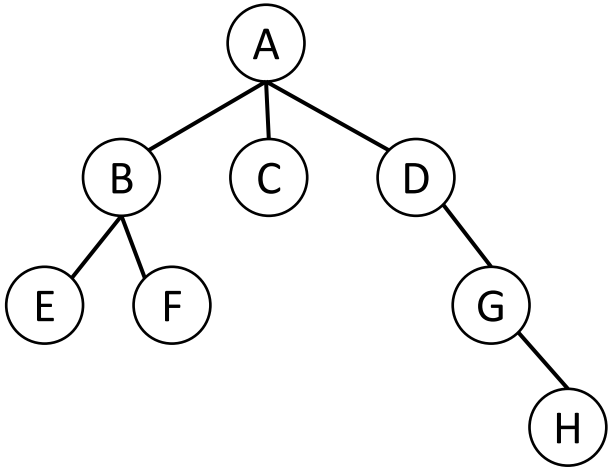
8 nodes, 7 edges

Why?

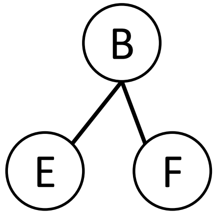
- Each edge goes from a parent node to a child node. Associate that edge with the child node.
- There are exactly $n-1$ children nodes in the tree.
- So there are $n-1$ edges.

Defining Subtrees

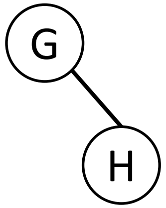
For any tree and node v, you can form the **subtree rooted at v** by taking v and all of its descendants.



Subtree rooted at B



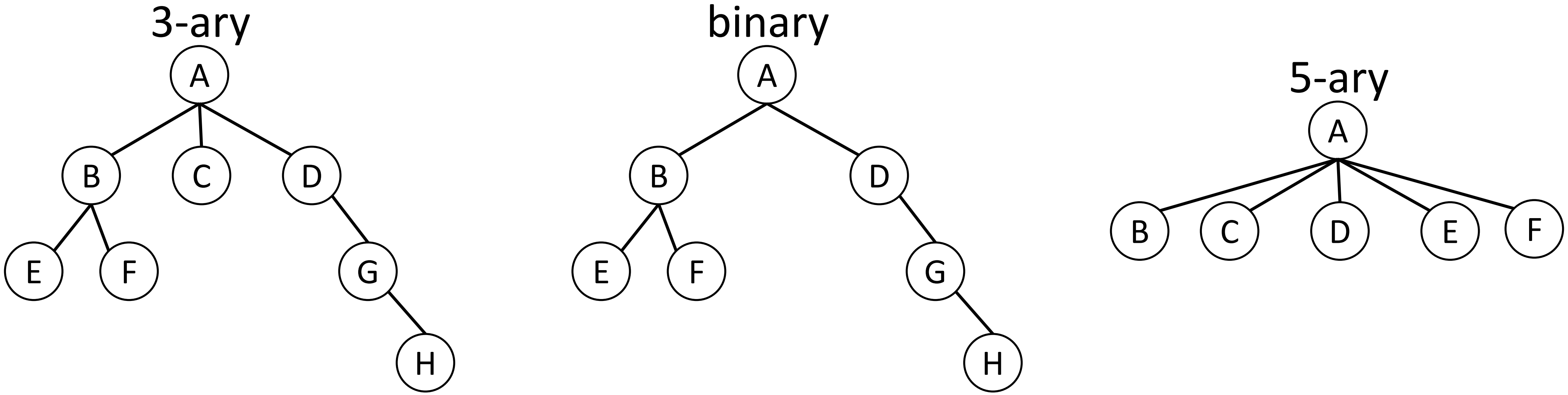
Subtree rooted at G



Defining k-ary Trees

A **k-ary** tree is a tree that imposes a <u>maximum</u> number of children to each node.

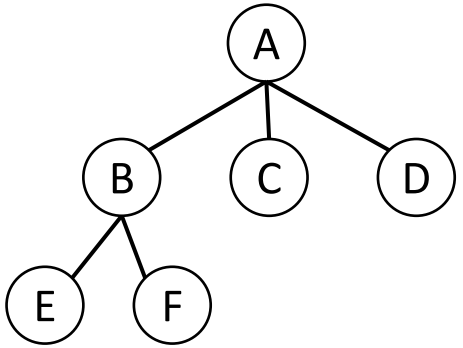
Binary trees (with at most 2 children per node) are especially popular.



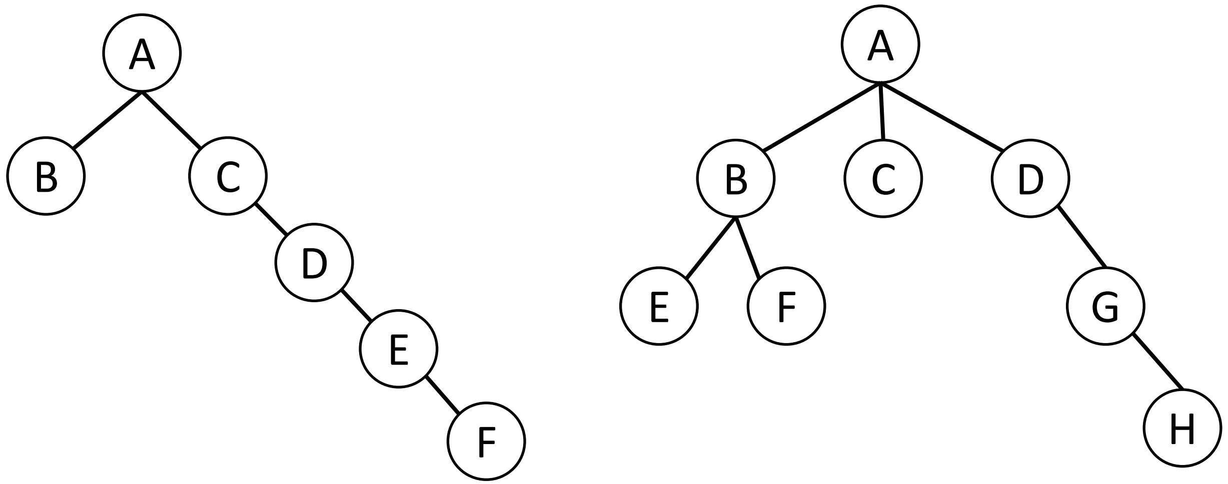
Defining Balanced Trees

Balanced tree = for any node, the heights of its child subtrees differ by ≤1

Balanced Tree



Unbalanced Trees



Operations of the Tree ADT

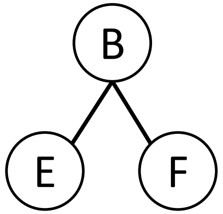
Tree ADT: Values

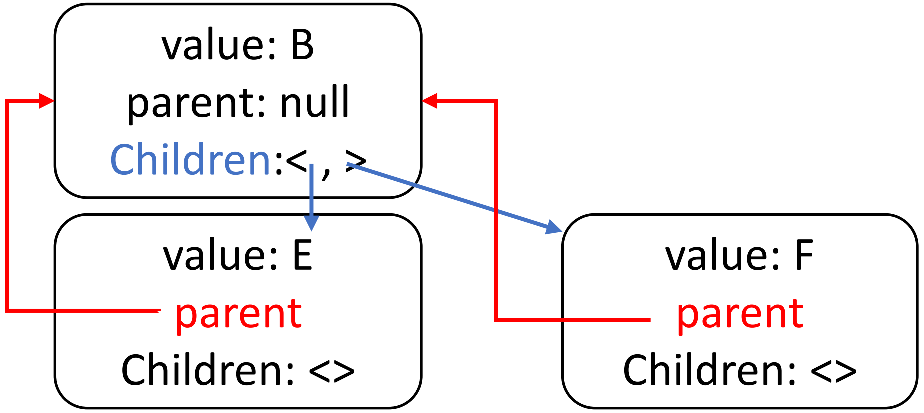
- Trees are represented in the code using the Tree class.

```java
import java.util.*;

public class Tree {
    Object value;
    Tree parent;
    List<Tree> children;

    public Tree(Object val) {
        this.value = val;
        this.children = new LinkedList<Tree>();
    }
}
```

- Values in tree = (java) Object class

Example:





Tree ADT: Operations

- Operations:

```
void add(Object n, Object m)
Tree find(Object n)
Tree remove(Object n)
void move(Object n, Object m)
```

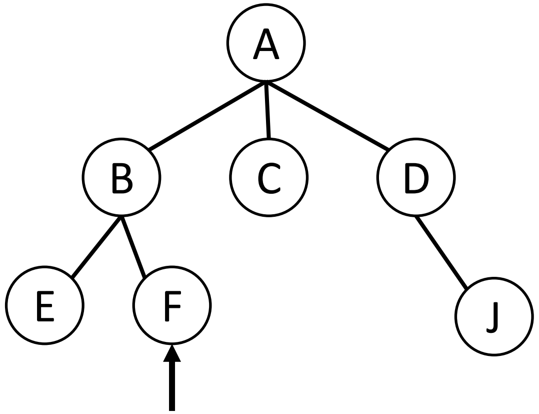
```
public class Tree {
    Object value;
    Tree parent;
    List<Tree> children;

    public Tree(Object val) {…}

    Tree find(Object val) {…}
    void add(Object val, Object parent) {…}
    Tree remove(Object val) {…}
    void move(Object val, Object newParent) {…}
}
```

- Semantics:
  - **Adding and moving:** Specify the (new) parent node as 2nd input
  - **Finding and removing:** Returns a Tree object

Tree ADT: Operations Example

Take the following sequence of operations:

- root.add(J,D)
- root.remove(C)
- root.find(F)

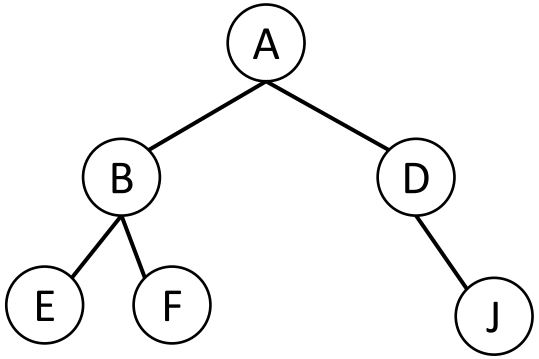


Returns the Tree (node)
with value F

Tree ADT: Operations Example

<u>Take the following sequence of operations:</u>

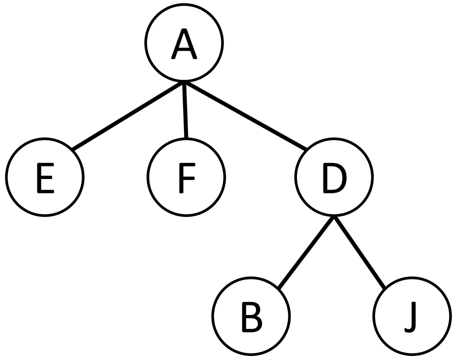
- root.add(J,D)
- root.remove(C)
- root.find(F)
- **root.move(B,D)**



Tree ADT: Operations Example

<u>Take the following sequence of operations:</u>

- root.add(J,D)
- root.remove(C)
- root.find(F)
- **root.move(B,D)**



Tree ADT: Implementing Node Finding

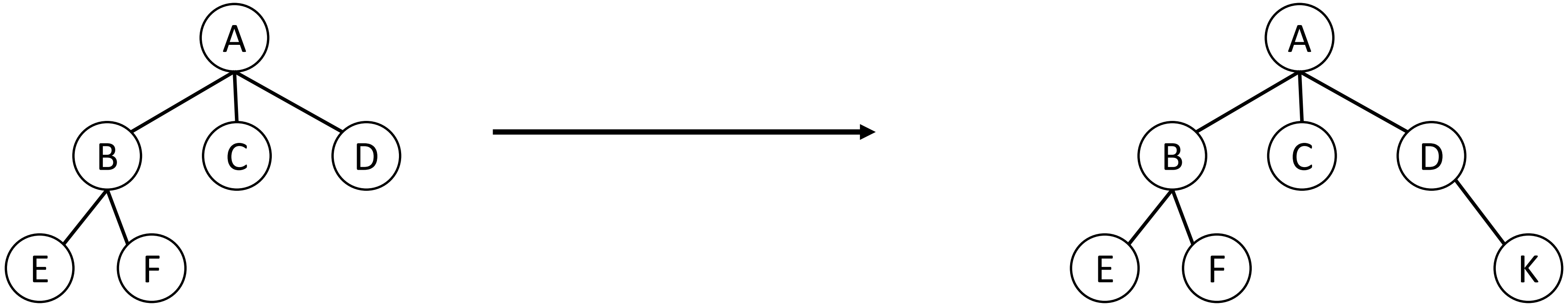
- Due to hierarchical structure, cannot sequentially iterate through the tree (unlike array)

- So instead:
  1. Start at root,
  2. Check if it is the Tree (node) we are looking for,
  3. If not, call recursively on the children.

```
Tree find(Object val) {
    if (value.equals(val)) {
        return this;
    }
    else {
        if(children.isEmpty()) {
            return null;
        }
        for(Tree child: children){
            Tree attemptNode = child.find(val);
            if(attemptNode != null) {
                return attemptNode;
            }
        }
        return null;
    }
}
```

Tree ADT: Implementing Node Addition

Operation = root.add(K,D):



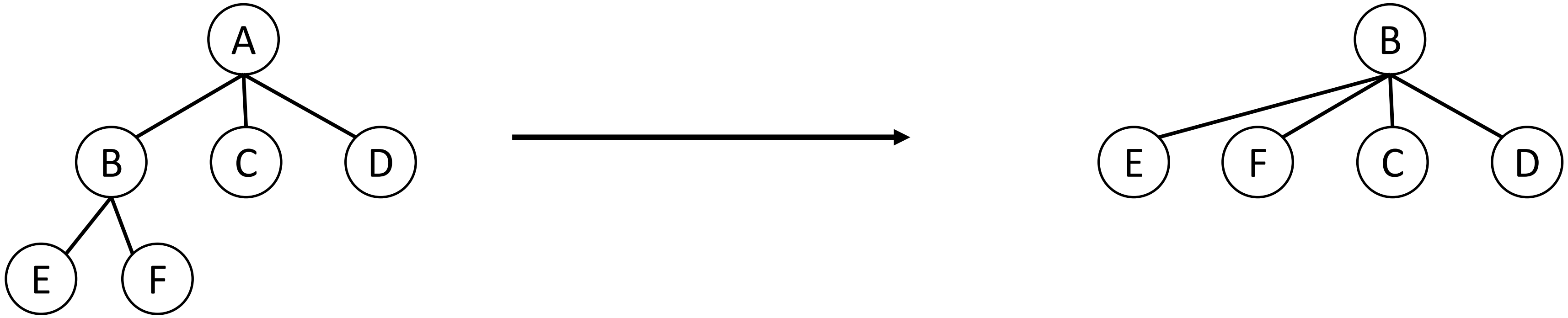
Tree ADT: Implementing Node Addition

- Adding a node is simple,
- Just make sure the newly added edge is added on both sides.

```
void add(Object n, Object parent) {
    Tree parentNode = find(parent);
    if (parentNode == null) return;

    Tree nodeToAdd = new Tree(n);
    parentNode.children.add(nodeToAdd); // Different add here
    nodeToAdd.parent = parentNode;
}
```

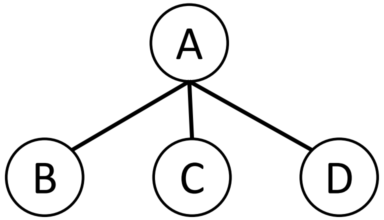
Tree ADT: Implementing Node Removal

Operation = root.remove(A):



Tree ADT: Implementation

- Find the node to remove,
- If it is the root, then make sure one of its children becomes the new root, and rewire the edges
- Take for example:




```
Tree remove(Object val) {
    Tree nodeToRemove = find(val);
    if (nodeToRemove == null) return null;

    if (nodeToRemove.parent == null){
        if (! nodeToRemove.children.isEmpty()) {
            Tree newRoot = nodeToRemove.children.pop();
            newRoot.parent = null;
            newRoot.children.addAll(nodeToRemove.children);
            for (Tree child : nodeToRemove.children){
                child.parent = newRoot;
            }
            return newRoot;
        }
        else return null;
    }
…
```

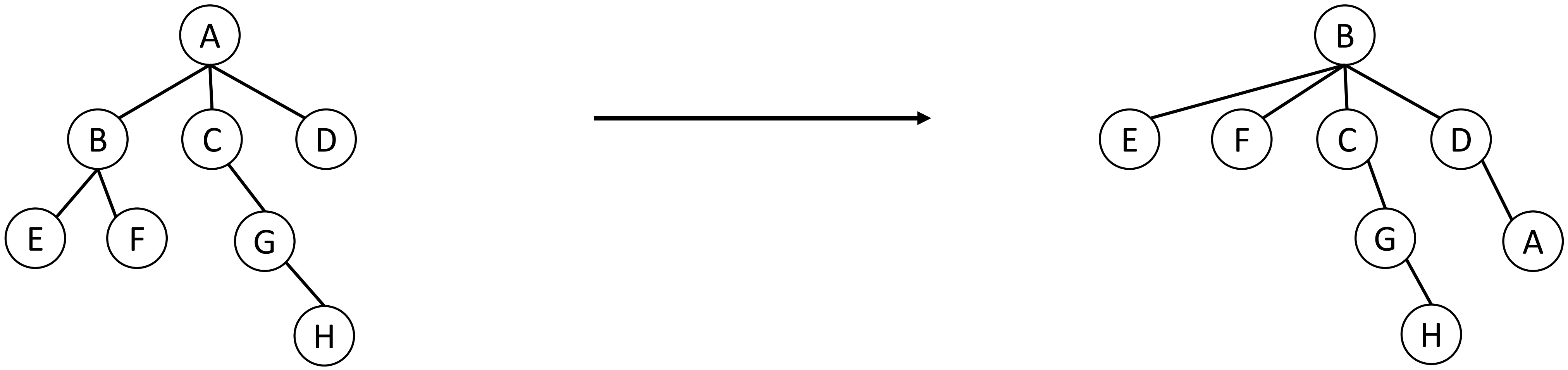
Tree ADT: Implementation

- If it is not the root, rewire the tree from the parent of the removed node to its grandchildren, and inversely.

```
…
    else {
        int index = nodeToRemove.parent.children.indexOf(nodeToRemove);
        nodeToRemove.parent.children.remove(nodeToRemove);
        if (nodeToRemove.children.isEmpty() == false) {
            for (Tree child : nodeToRemove.children){
                child.parent = nodeToRemove.parent;
            }
            nodeToRemove.parent.children.addAll(index,nodeToRemove.children);
            return nodeToRemove.parent.children.get(0);
        }
        else return null;
    }
}
```

Tree ADT: Implementing Node Move

Operation = root.move(A,D):



Tree ADT: Implementing Node Move

- Simple to implement once you have both add and remove functions

```
void move(Object val, Object newParent) {
    Tree nodeToMove = find(val);
    remove(val);
    Tree newParentNode = find(newParent);

    if (nodeToMove != null && newParentNode != null) {
        nodeToMove.parent = newParentNode;
        newParentNode.children.add(nodeToMove);
    }
}
```

Tree ADT: Implementing Node Finding

- Operations add, move and remove build upon **find**.

- **Many implementations but same concept:**
  - Start at root, and if root is not node you are looking for,
  - Go to its children and repeat, etc., until you reach a leaf.

- Whether you look at the value of a node before that of its children, or inversely, impacts the order in which you go through the tree's values
  - **Preorder**, **Postorder**, **Inorder** traversals
  - **Breadth-first search** (or level order) traversal

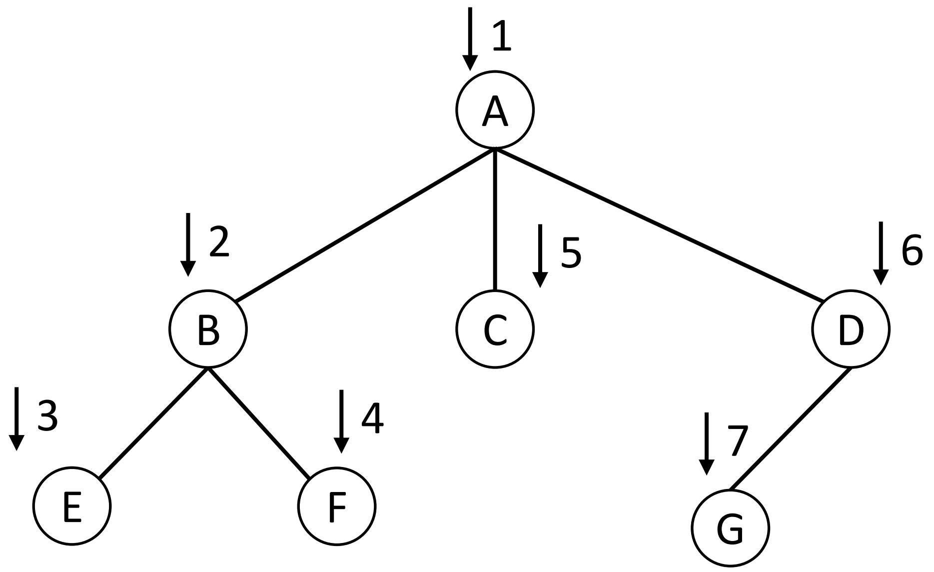
Tree Traversals

Preorder (DFS) Tree Traversal

Lancaster University

1. Look at root's value,
2. Traverse the first subtree, then next, …
3. Until all subtrees have been traversed.

```
void traversePreOrder() {
    System.out.println(this.value);

    if(!children.isEmpty()) {
        for(Tree child: children){
            child.traversePreOrder();
        }
    }}
```

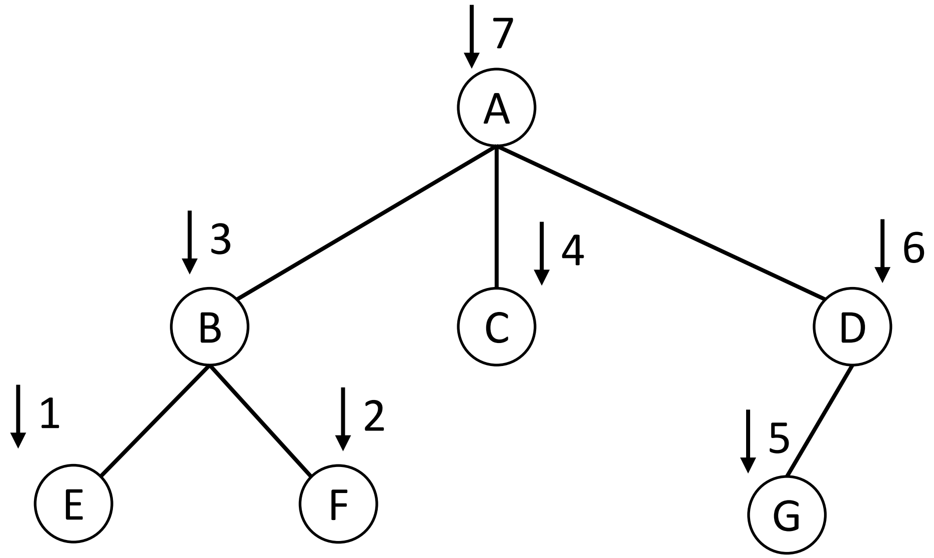


Postorder (DFS) Tree Traversal

Lancaster University

1. Traverse the first subtree, then next, …
2. Once all subtree have been visited, look at the root's value.

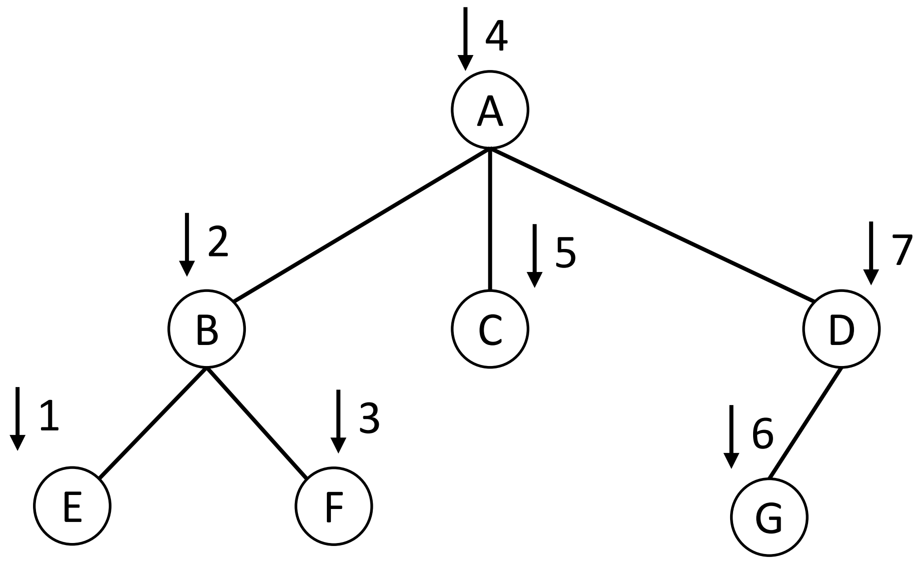
```
void traversePostOrder() {
    if(! children.isEmpty()) {
        for(Tree child: children){
            child.traversePostOrder();
        }
    }
    System.out.println(this.value);
}
```



Inorder (DFS) Tree Traversal

1. Traverse the left subtrees in order,
2. Look at the root's value,
3. Traverse the right subtrees in order.

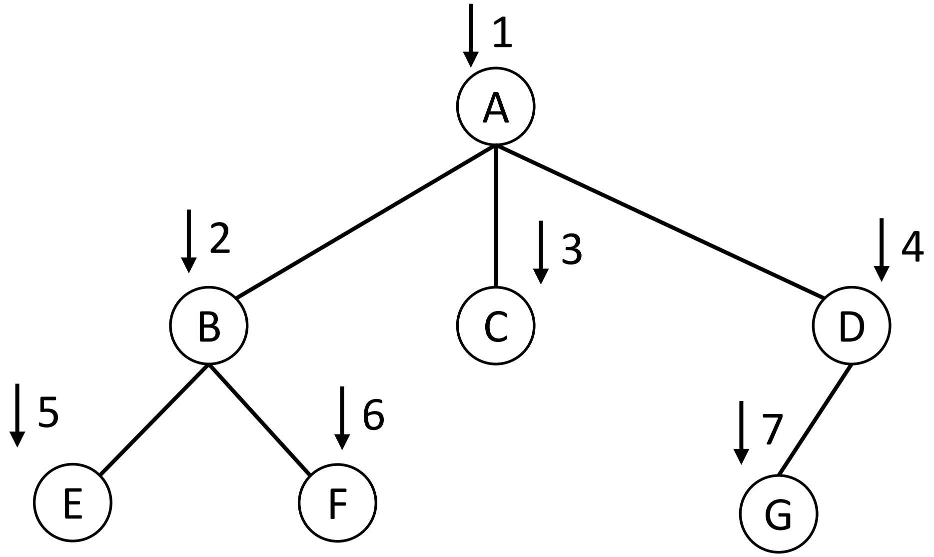
```
void traverseInOrder() {
    if(! children.isEmpty()) {
        children.get(0).traverseInOrder();
    }
    System.out.println(this.value);

    if(children.size() > 1) {
        for(Tree child: children.subList(1, 
children.size())){
            child.traverseInOrder();
        }
    }}
```



Breadth-First Search Tree Traversal

1. Traverse the tree from left to right at each level, starting from the root.
2. By using a Queue!

```
void traverseBFS() {
    Queue<Tree> queue = new LinkedList<Tree>();
    queue.add(this);
    while(queue.size() > 0){
        Tree node = queue.remove();
        System.out.println(this.value);

        if(! node.children.isEmpty()){
            for (Tree child: node.children){
                queue.add(child);
            }
        }
    }}
```



Tree Traversals

- **Traversals** (preoder, postorder, inorder, level order) can enumerate all elements, or decide in which way we look through the tree to find elements

- All (DFS) traversals can be implemented in a **non-recursive** manner by explicitly creating a **stack** and using it for the traversal.

Summary

Today's lecture:

- Introduction to the **Tree ADT** and its operations
- Descriptions of different **tree traversals**

- **Next Lecture:** Binary search trees (BST) and self-balancing BSTs.
- **Any questions?**