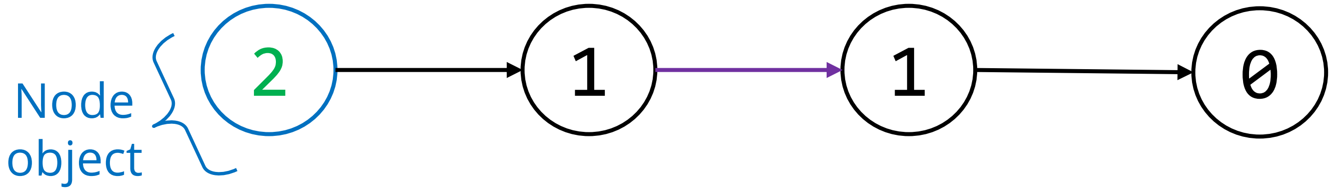
COSC 222 Data Structure

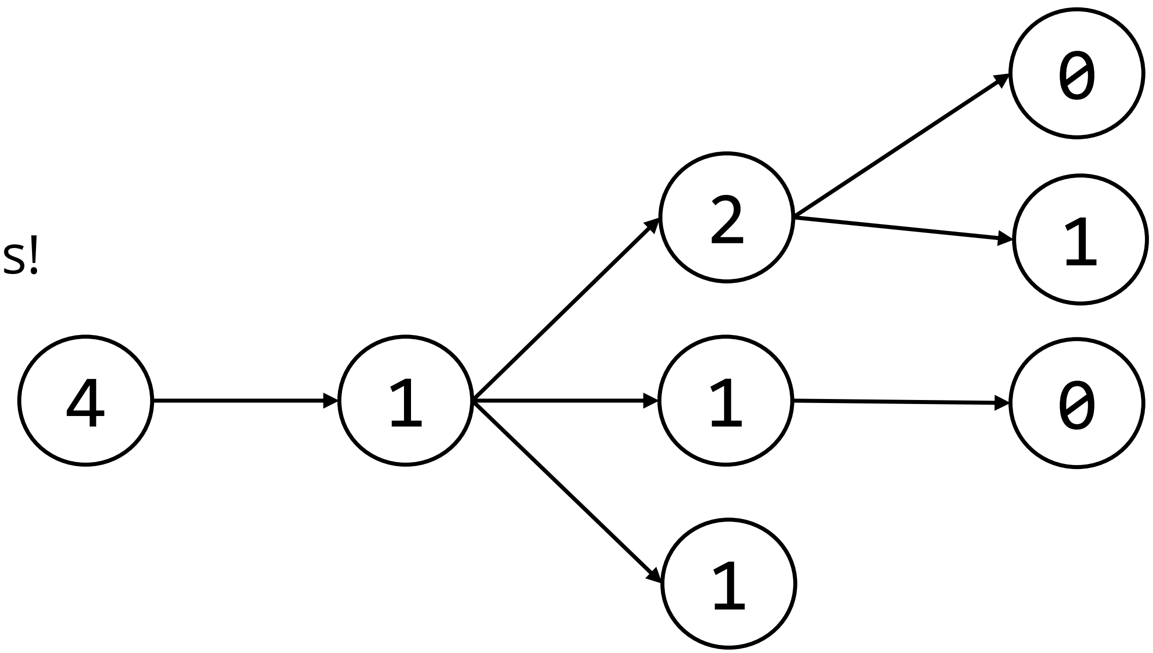
Tree

Trees

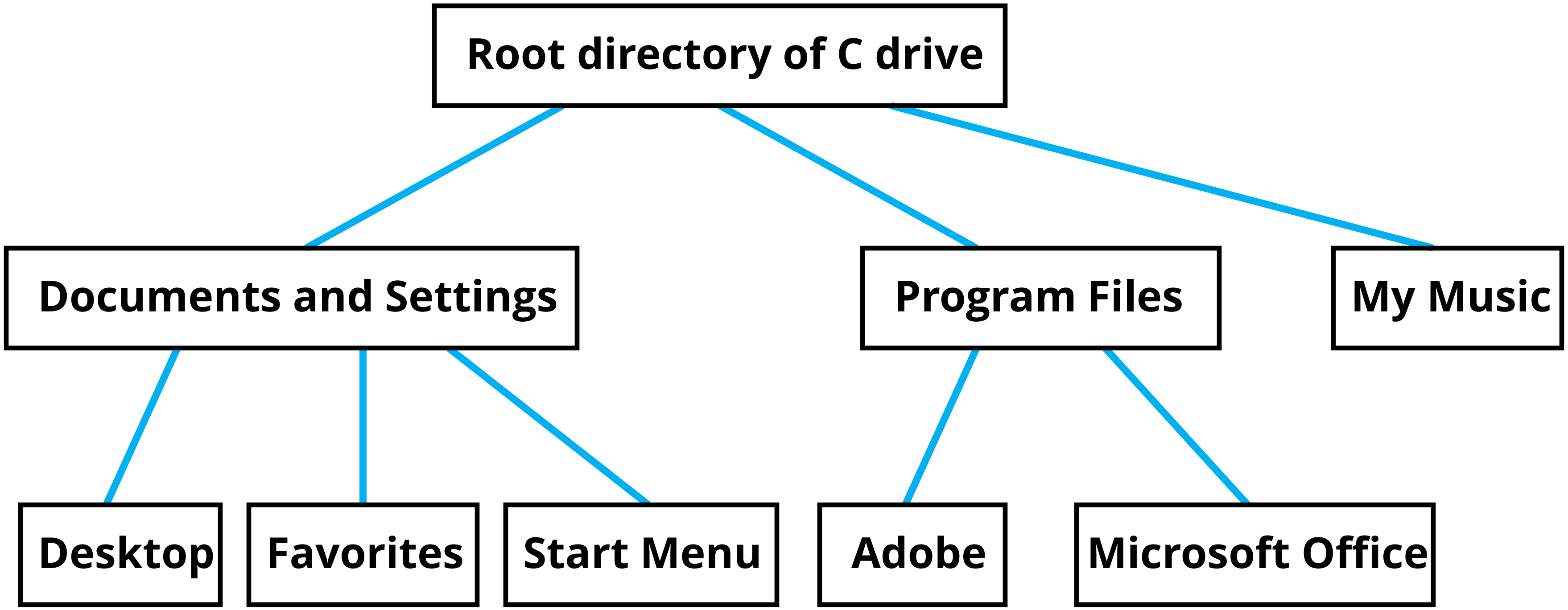
Singly linked list:



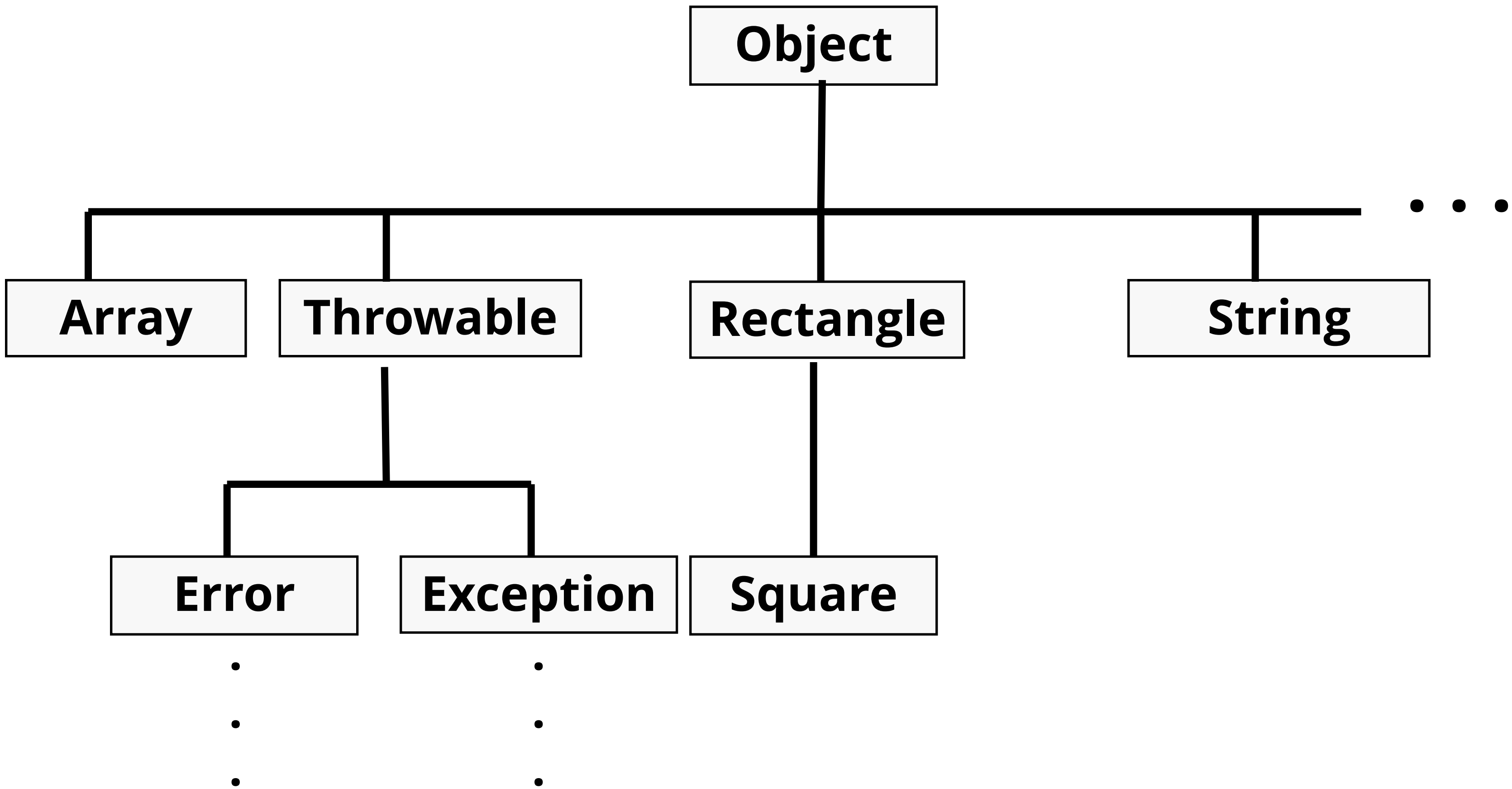
Today: trees!



Tree Example: Computer File System



Example: Java's Class Hierarchy



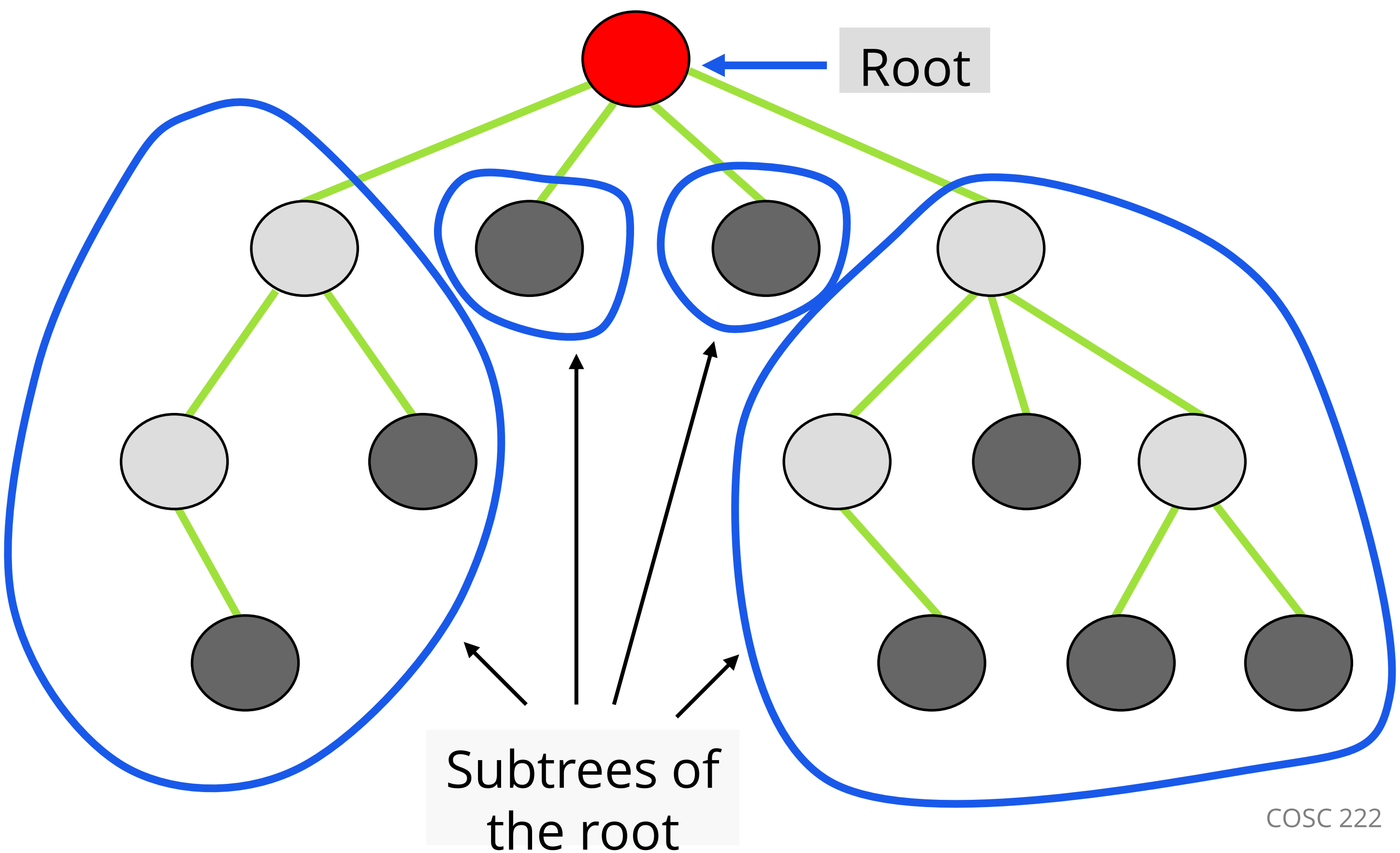
Trees

- A **tree** is a nonlinear data type that stores elements in a hierarchy.
- **Examples** in real life:
  - Computer system (folders and subfolders)
  - Class inheritance hierarchy in Java
  - Decision trees
  - Family tree
  - Table of contents of a book

In CS, we draw trees "**upside down**"

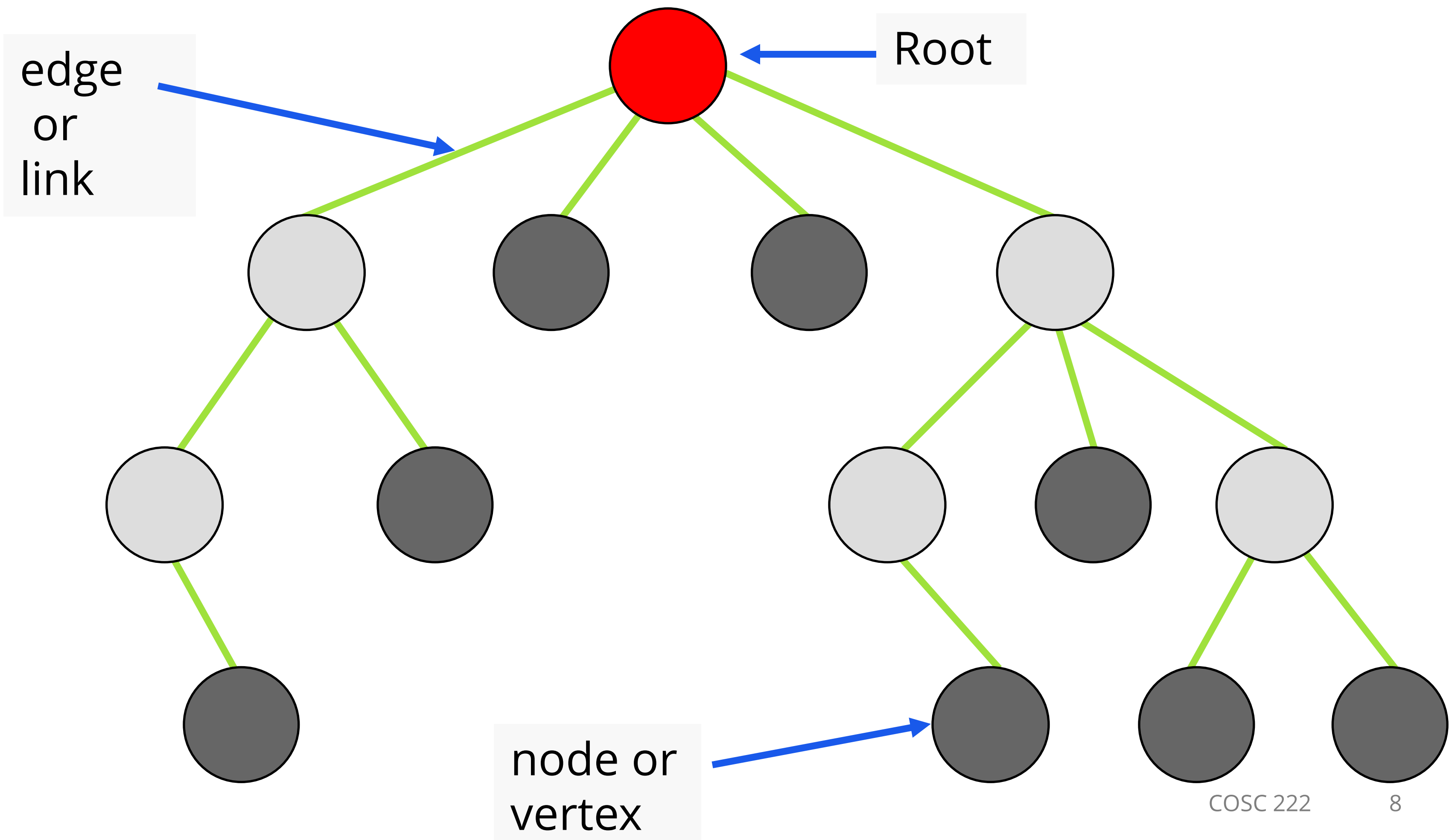
Tree Definition

- **Tree**: a set of elements that either
  - it is empty
  - or, it has a distinguished element called the **root** and zero or more trees (called **subtrees** of the root)



Tree Terminology

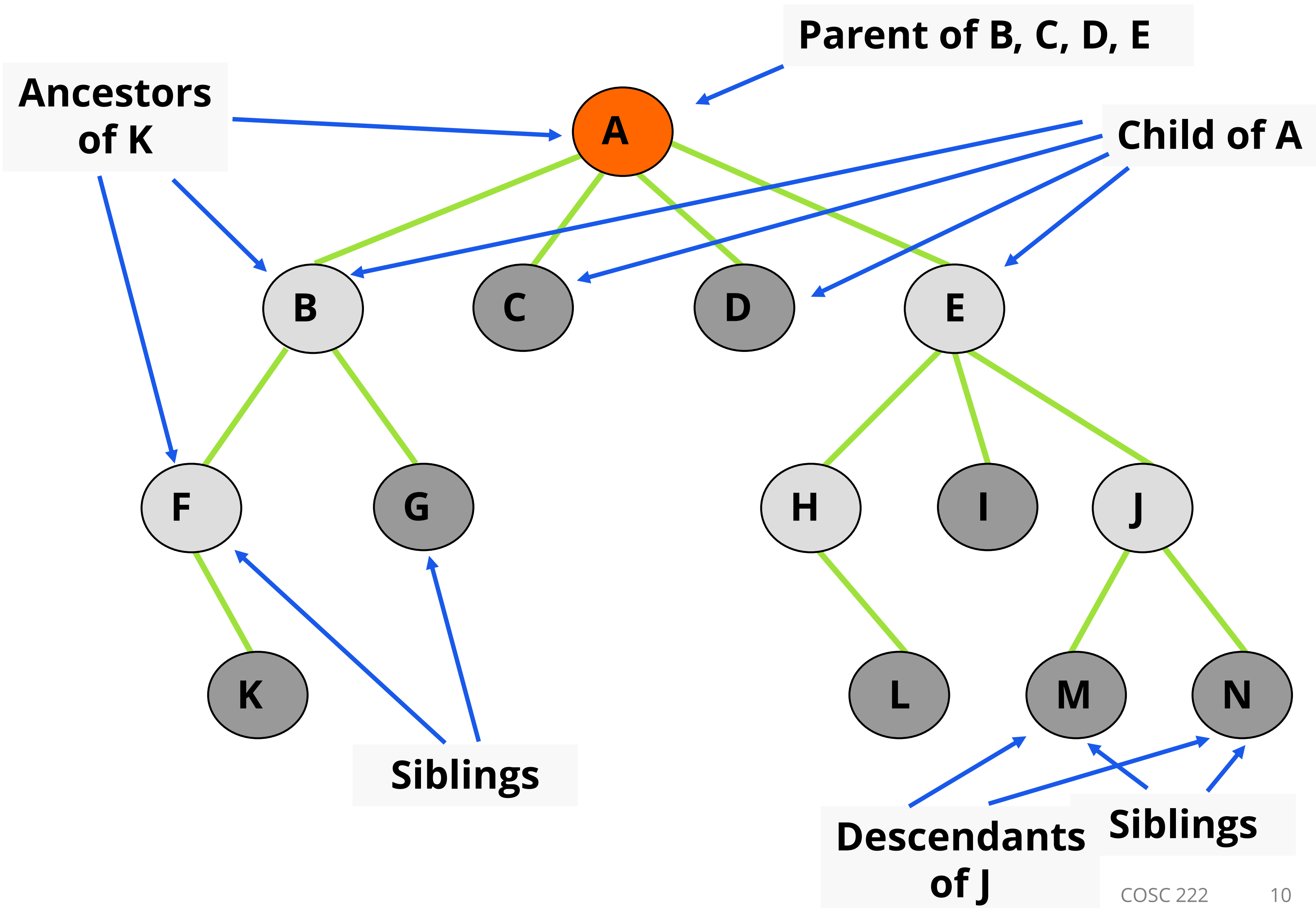
- **Nodes/Vertices**: the elements in the tree
- **Edges**: connections between nodes



Tree Terminology

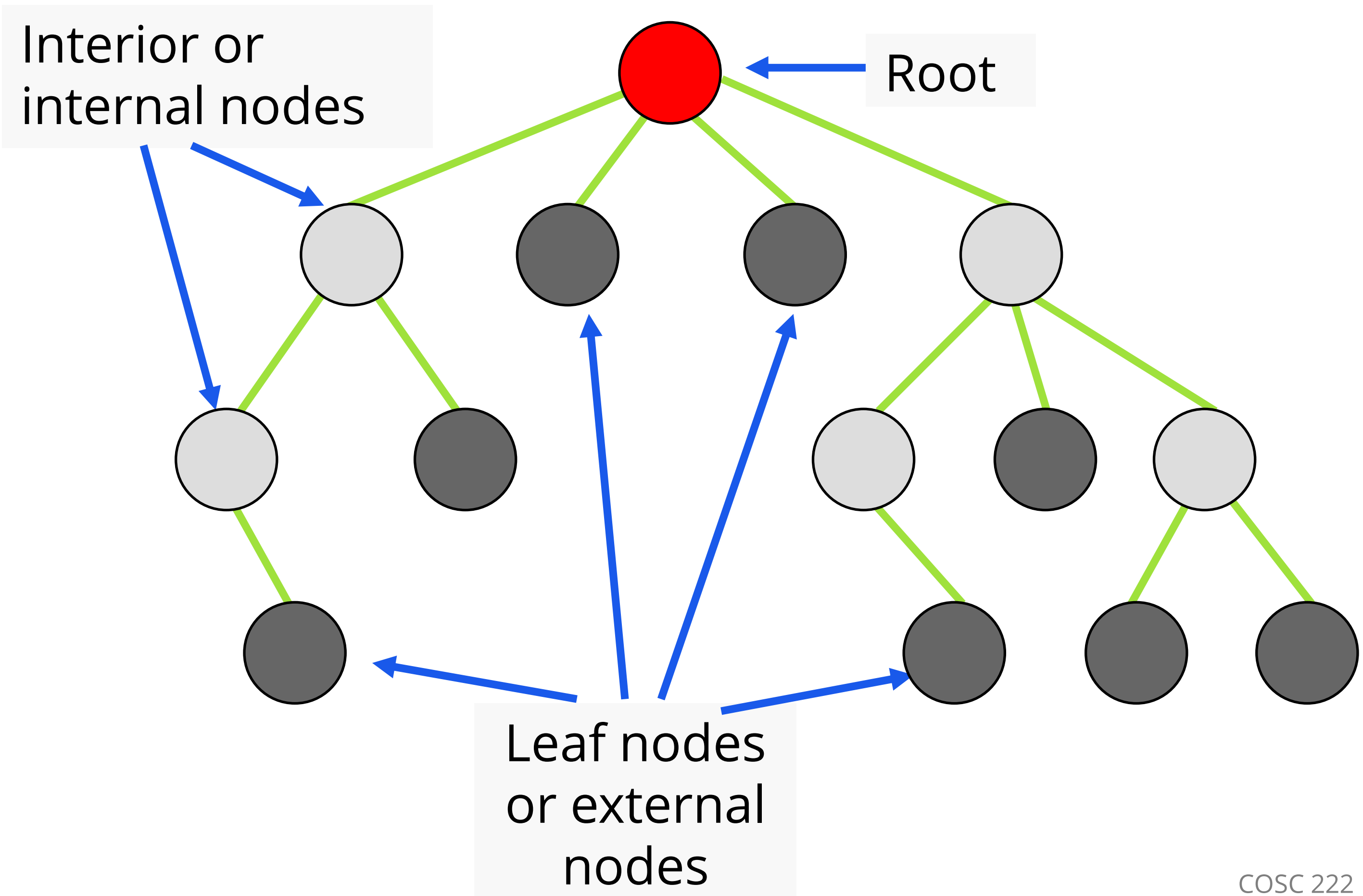
- **Parent** or **predecessor**: the node directly above another node in the hierarchy
  - A node can have only one parent
- **Child** or **successor**: a node directly below another node in the hierarchy
- **Siblings**: nodes that have the same parent
- **Ancestors** of a node: its parent, the parent of its parent, etc.
- **Descendants** of a node: its children, the children of its children, etc.

Tree Terminology



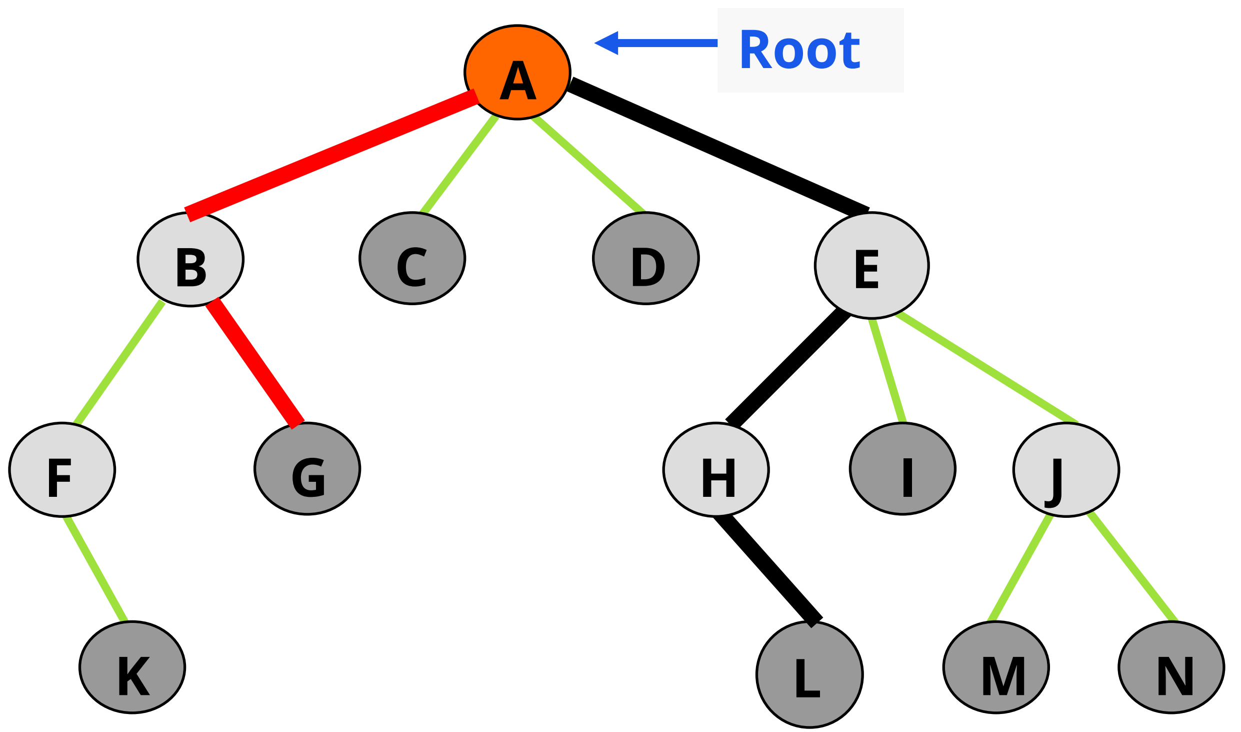
Tree Terminology

- **Leaf node**: a node without children
- **Internal node**: a node that is not a leaf node



Height of a Tree

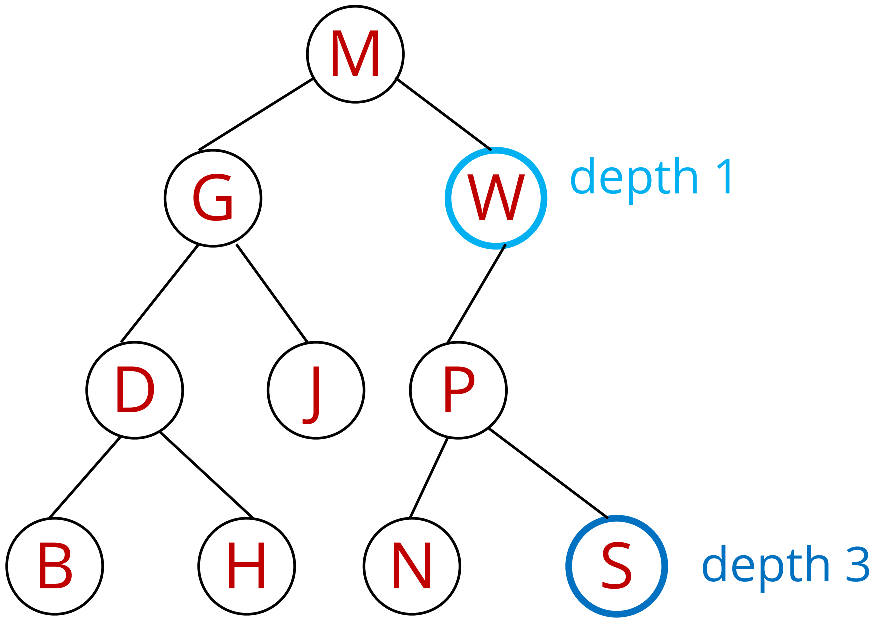
- A **path** is a sequence of edges leading from one node to another
- **Length of a path**: number of edges on the path
- **Height of a (non-empty) tree** : A tree's (or subtree's) length of the longest path from the root to a leaf



Height = 3

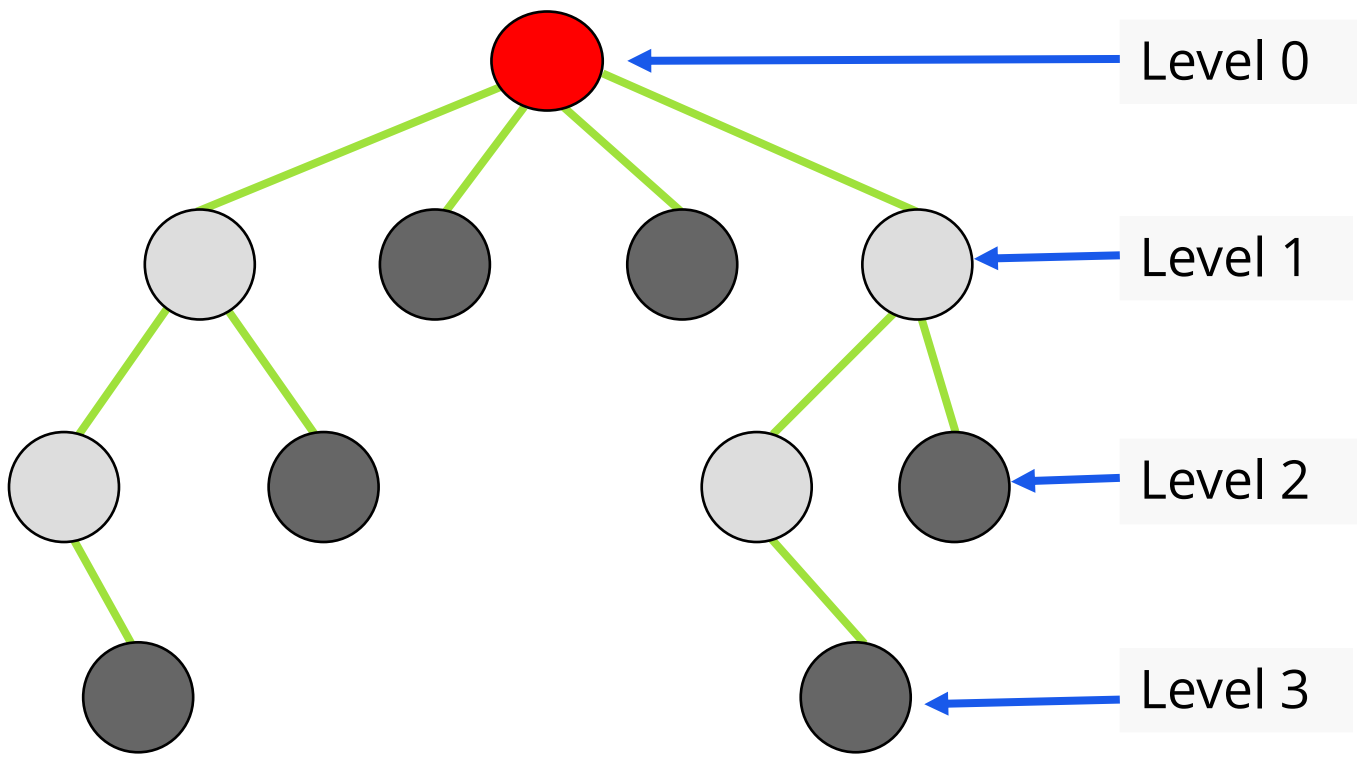
Tree Terminology

- A node's **depth** is the length of the path to the root.



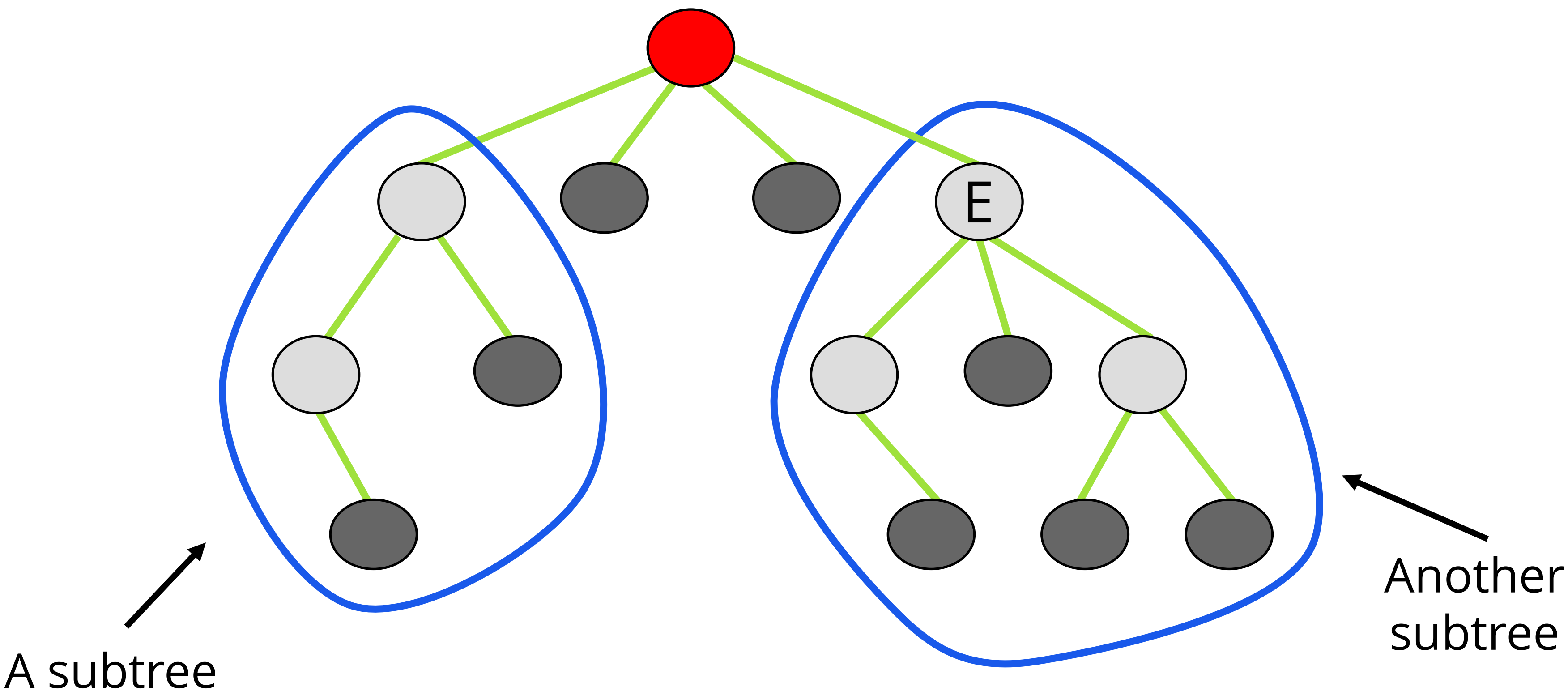
Level of a Node

- **Level of a node**: number of edges between root and the node
  - It can be defined recursively:
  - Level of root node is 0
  - Level of a node that is not the root node is level of its parent + 1



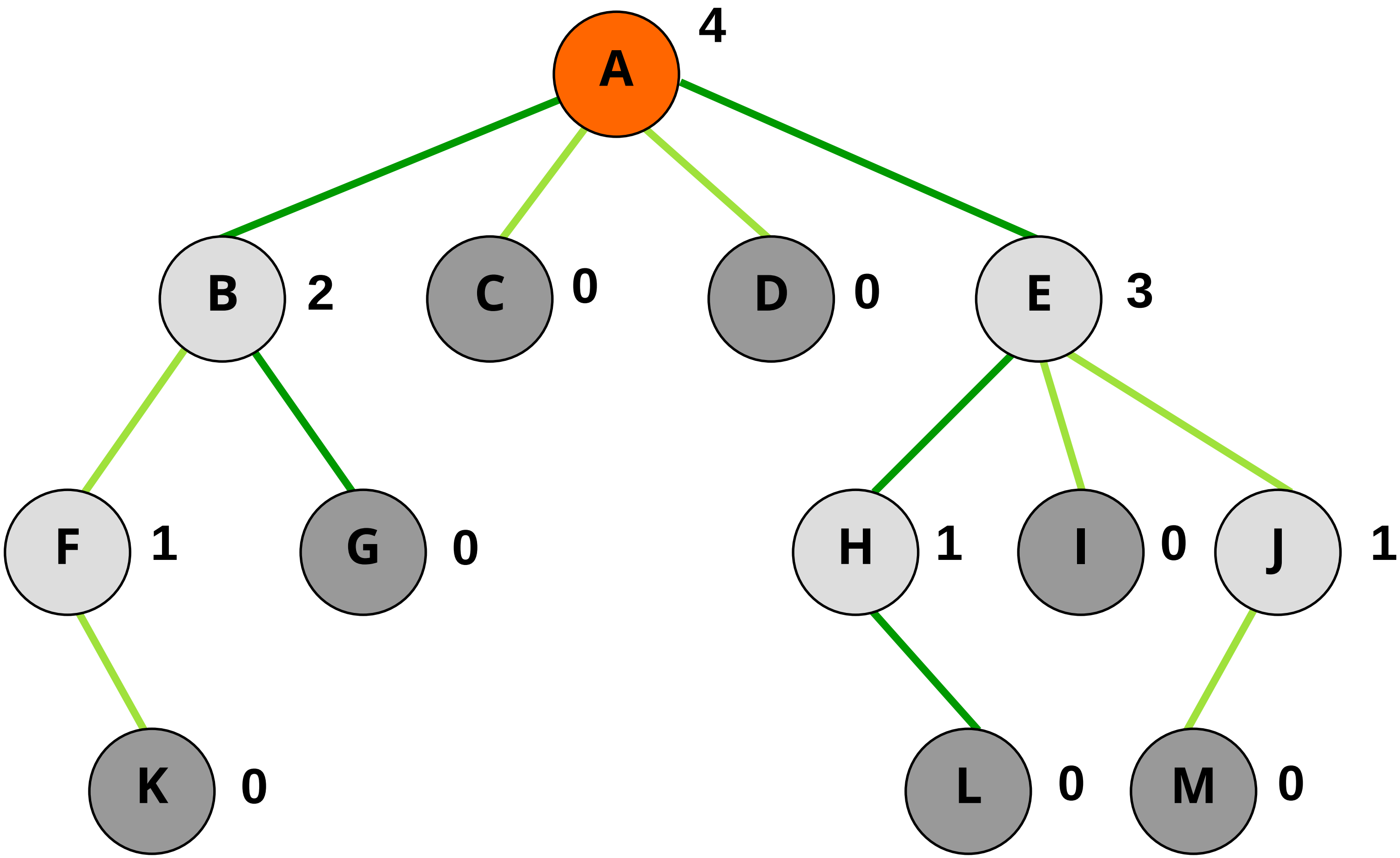
Subtrees

- **Subtree of a node**: consists of a child node and all its descendants
  - A subtree is itself a **tree**
  - A node may have many subtrees



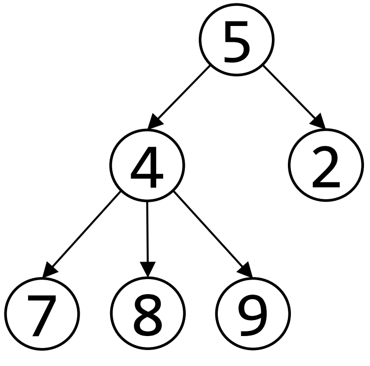
More Tree Terminology

- **Degree of a node**: the number of children it has
- **Degree of a tree**: the maximum of the degrees of the tree's nodes

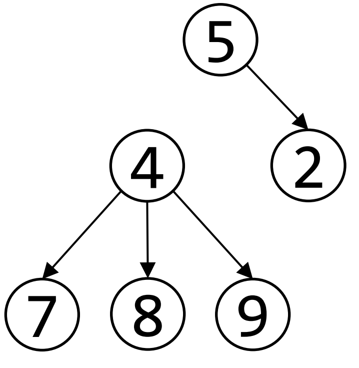


Properties

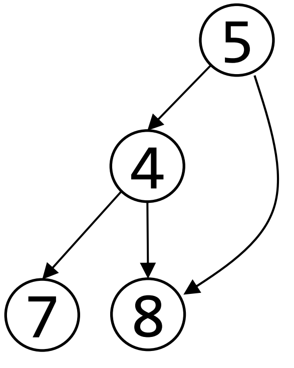
- The important properties of tree data structure are-
  - Each node may have zero or more successors (children)
  - Each node has exactly one predecessor (parent) except the root, which has none
  - All nodes are reachable from root



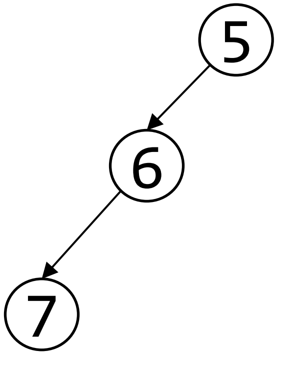
A tree



Not a tree



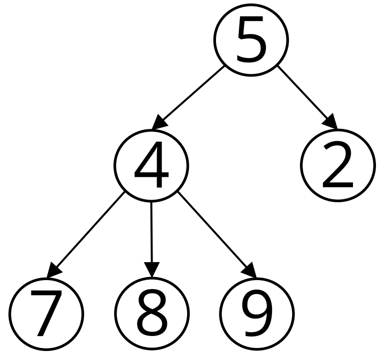
Not a tree



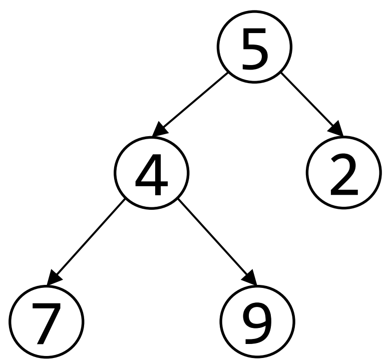
A tree

Tree Classifications

- **General tree**: a tree each of whose nodes may have any number of children
- **n-ary tree**: a tree each of whose nodes may have no more than n children
- **Binary tree**: a tree each of whose nodes may have no more than 2 children
  - i.e. a binary tree is a tree with degree 2
- The **children** (if present) are called the left child and right child



General tree



Binary tree

Recap: Recursion

- The basic concept of recursion is:

A method can call itself

The base/easy case

- Any recursive method
  - Cannot *always* call itself, or else the chain of calls will never end.

    An infinite recursion

    Always results in a "stack overflow"

    The recursive equivalent of an infinite loop

  - Must have some easy, non-recursive "base case" or "easy case".
  - The recursive calls must always lead, sooner or later, to this base case.

The traditional example

- n factorial (n!) can be defined and programmed using *recursion*:

$n! = 1 \quad$ (if $n \leq 1$)

$n! = n(n-1)! \quad$ (if $n > 1$)

```
public static long fact(int n) {
  if(n<=1)
    return 1;
  else
    return n*fact(n-1);
}
```

```
public static long fact(int 4){
      if(n<=1) return 1;
      else return 4*fact(4-1);
   }
```

```
//main method
system.out.println(fact(4));
```

```
public static long fact(int 3){
      if(n<=1) return 1;
      else return 3*fact(3-1);
   }
```

```
public static long fact(int 2){
      if(n<=1) return 1;
      else return 2*fact(2-1);
   }
```

```
public static long fact(int 1){
      if(n<=1) return 1;
      else return 1*fact(n-1);
   }
```

```
public static long fact(int 4){
      if(n<=1) return 1;
      else return 4*fact(4-1);
   }
```

```
public static long fact(int 3){
      if(n<=1) return 1;
      else return 3*fact(3-1);
   }
```

```
public static long fact(int 2){
      if(n<=1) return 1;
      else return 2*fact(2-1);
   }
```

```
public static long fact(int 1){
      if(n<=1) return 1;
      else return 1*fact(n-1);
   }
```

return 1

```
public static long fact(int 4){
      if(n<=1) return 1;
      else return 4*fact(4-1);
   }
```

```
public static long fact(int 3){
      if(n<=1) return 1;
      else return 3*fact(3-1);
   }
```

```
return 2
```

```
public static long fact(int 2){
      if(n<=1) return 1;
      else return 2*1;
   }
```

```
public static long fact(int 4){
        if(n<=1) return 1;
        else return 4*fact(4-1);
    }
```

return 6

```
public static long fact(int 3){
        if(n<=1) return 1;
        else return 3*2;
    }
```

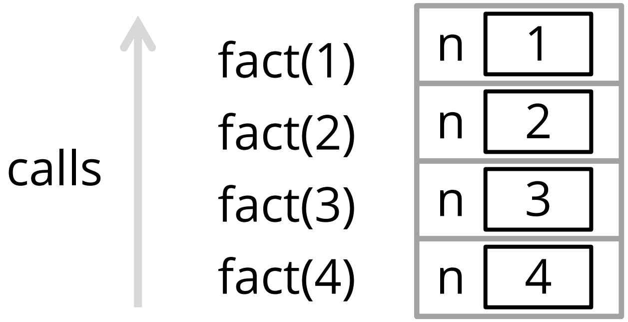
```
public static long fact(int 4){
      if(n<=1) return 1;
      else return 4*6;
   }
```

```
//main method
system.out.println(fact(4));
```

24

How recursion works

- Each time a method is called, a whole new set of local variables (including parameters) are created.
- Many instances of one method can all be running simultaneously, each with its own variables.
- These sets of variables are stored on the stack.



There are four separate versions of the **fact** method running, each with its own parameter **n**.

Use the Debugger to stop at the n<=1 easy case. Look at the stack.

Don't worry about the stack

- You don't need to visualize the stack, and all of the calls, and all of the returns.

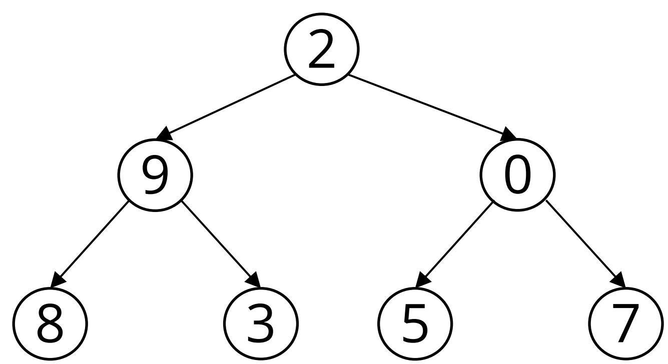
- All you have to do to write a recursive method is focus on one step:
  - 1) Find a way to solve the problem by using the solution to a slightly smaller version of the same problem.
  - 2) Find an easy case (base case) that will always be reached if the problem keeps getting smaller.

- While writing methodX, simply assume that methodX will always work on any smaller case. Trust it.

Binary Trees

- **Recursive definition** of a binary tree:
  it is
  - The empty tree
  - Or, a tree which has a root whose left and right subtrees are binary trees

Binary Tree



Left subtree, which is also a binary tree

Right subtree (also a binary tree)