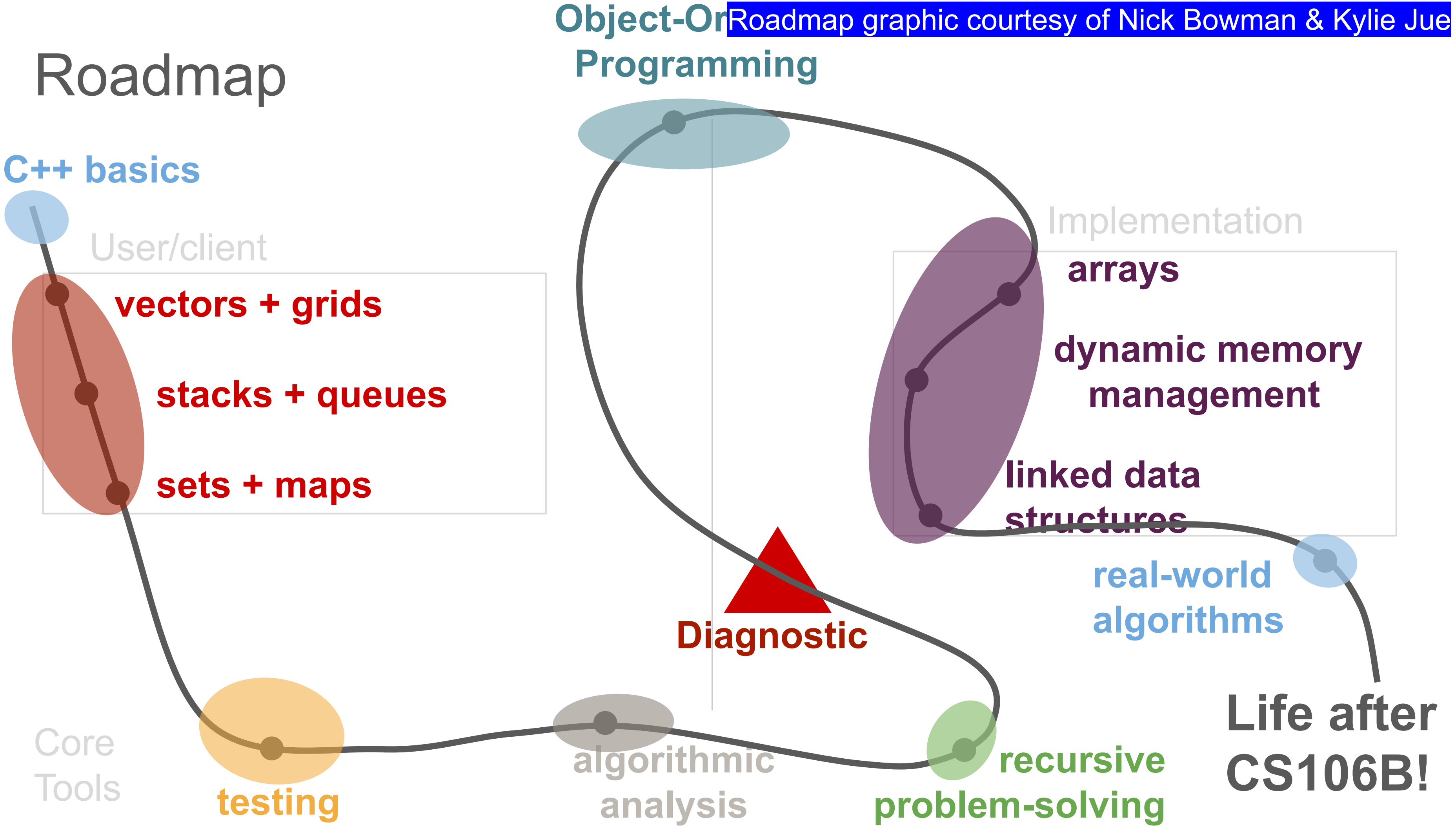
# Trees

**Is there any component of "Life after CS106B" that you would like us to focus on in the final lecture next week?**

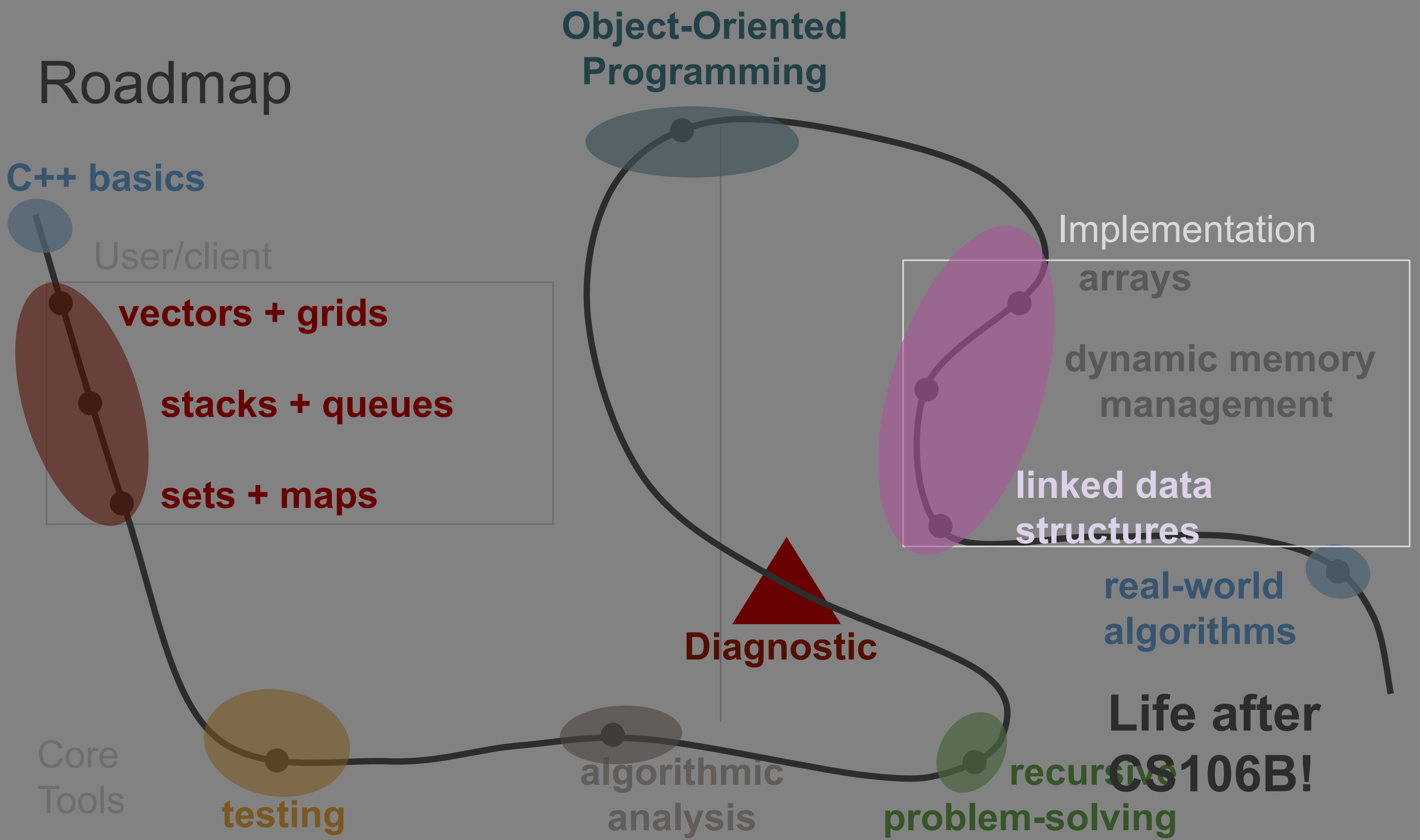
(put your answers in the chat)

# Roadmap




Roadmap graphic courtesy of Nick Bowman & Kylie Jue

# Roadmap



Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

# Today's questions

How can we better organize data stored in a linked data structure?

# Today's topics

1. Linked Data Structure Overview
2. Introduction to Trees
3. Trees in C++

# Review

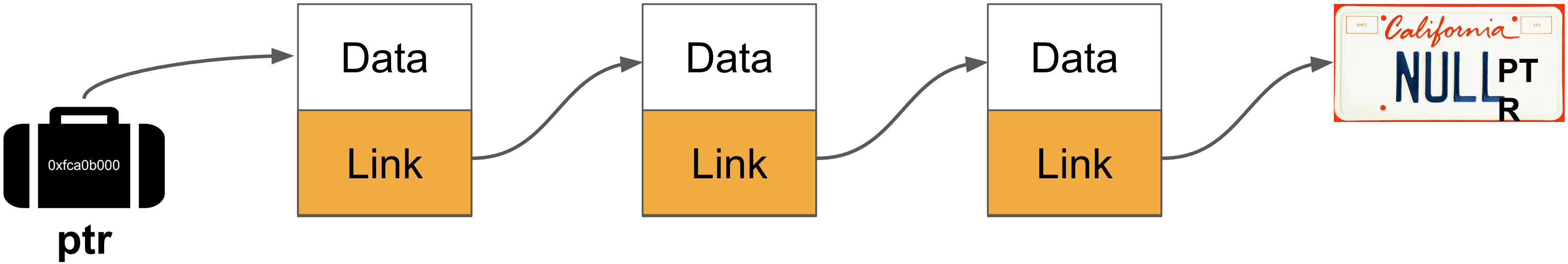
[linked data structures]

# Linked Data Structures

- Last week, we explored linked lists, our first example of a **linked data structure**.

# Linked Data Structures

- Last week, we explored linked lists, our first example of a **linked data structure**.



# Linked Data Structures

- Last week, we explored linked lists, our first example of a **linked data structure**.

- Linked data structures are distinguished by the fact that they stored data in a **distributed** manner. This means that the data is stored across many different locations in computer memory.

# Linked Data Structures

- Last week, we explored linked lists, our first example of a **linked data structure**.

- Linked data structures are distinguished by the fact that they stored data in a **distributed** manner. This means that the data is stored across many different locations in computer memory.

- In order to organize this data, we had to **bundle data alongside pointers** in the concept of a "node."

# Linked Data Structures

- Last week, we explored linked lists, our first example of a **linked data structure**.
- Linked data structures are distinguished by the fact that they stored data in a **distributed** manner. This means that the data is stored across many different locations in computer memory.
- In order to organize this data, we had to **bundle data alongside pointers** in the concept of a "node."
- Using pointers lets us **create links** to other nodes to impose structure (why?)

# Linked List Tradeoffs

- Storing data in a distributed (non-contiguous) manner had some distinct advantages over working with arrays.

# Linked List Tradeoffs

- Storing data in a distributed (non-contiguous) manner had some distinct advantages over working with arrays.
    - Insertion/removal of elements of a linked list was very quick because it only involved fast pointer rewiring operations. We never had to "shift" elements over to make room.

# Linked List Tradeoffs

- Storing data in a distributed (non-contiguous) manner had some distinct advantages over working with arrays.
  - Insertion/removal of elements of a linked list was very quick because it only involved fast pointer rewiring operations. We never had to "shift" elements over to make room.
  - Because all the data was stored in dynamic memory, expanding the size of the linked list was very easy and never required an expensive "re-sizing" operation that had to copy all the data.

# Linked List Tradeoffs

- Storing data in a distributed (non-contiguous) manner had some distinct advantages over working with arrays.

- However, we also ran into some limitations when it came to working with lists:

# Linked List Tradeoffs

- Storing data in a distributed (non-contiguous) manner had some distinct advantages over working with arrays.

- However, we also ran into some limitations when it came to working with lists:
  - Data was organized in a linear structure, which meant the path to traverse between any two nodes (specifically between the front and a node later on in the list) could get very long.

# Linked List Tradeoffs

- Storing data in a distributed (non-contiguous) manner had some distinct advantages over working with arrays.

- However, we also ran into some limitations when it came to working with lists:
  - Data was organized in a linear structure, which meant the path to traverse between any two nodes (specifically between the front and a node later on in the list) could get very long.
  - Finding elements in a linked list is an **O(n)** operation, which can get slow when we want to store many elements.

# Linked List Tradeoffs

- Storing data in a distributed (non-contiguous) manner had some distinct advantages over working with arrays.

- However, we also ran into some limitations when it came to working with lists:
  - Data was organized in a linear structure, which meant the path to traverse between any two nodes (specifically between the front and a node later on in the list) could get very long.
  - Finding elements in a linked list is an **O(n)** operation, which can get slow when we want to store many elements.
  - We couldn't feasibly write recursive algorithms that traversed linked lists, due to stack frame limits that came into play since traversal algorithms required one stack frame per node.

# Linked List Tradeoffs

- Storing data in a distributed (non-contiguous) manner had some distinct advantages over working with arrays.

- However, we also ran into some limitations when it came to working with lists.

- **Question:** Can we organize data in a linked data structure in such a way that the path between the "front" and any element in the structure is short (better than **O(n)**) even if there are many elements?

How can we better organize data stored in a linked data structure?

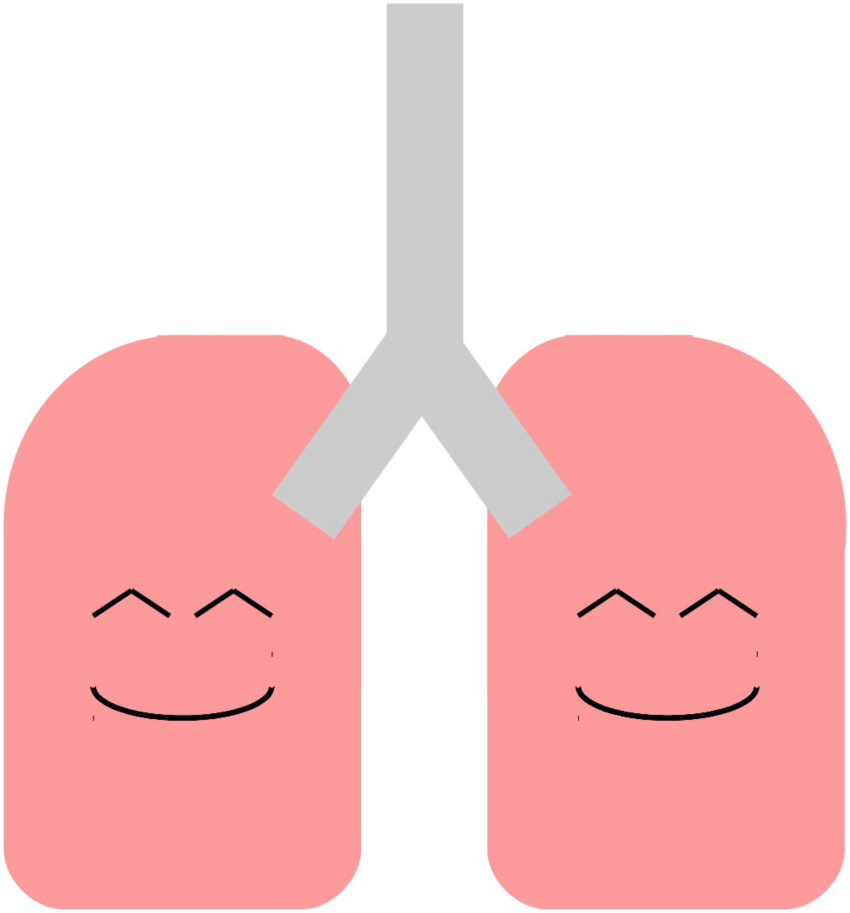
# Interactive Exercise

[borrowed from Keith Schwarz]

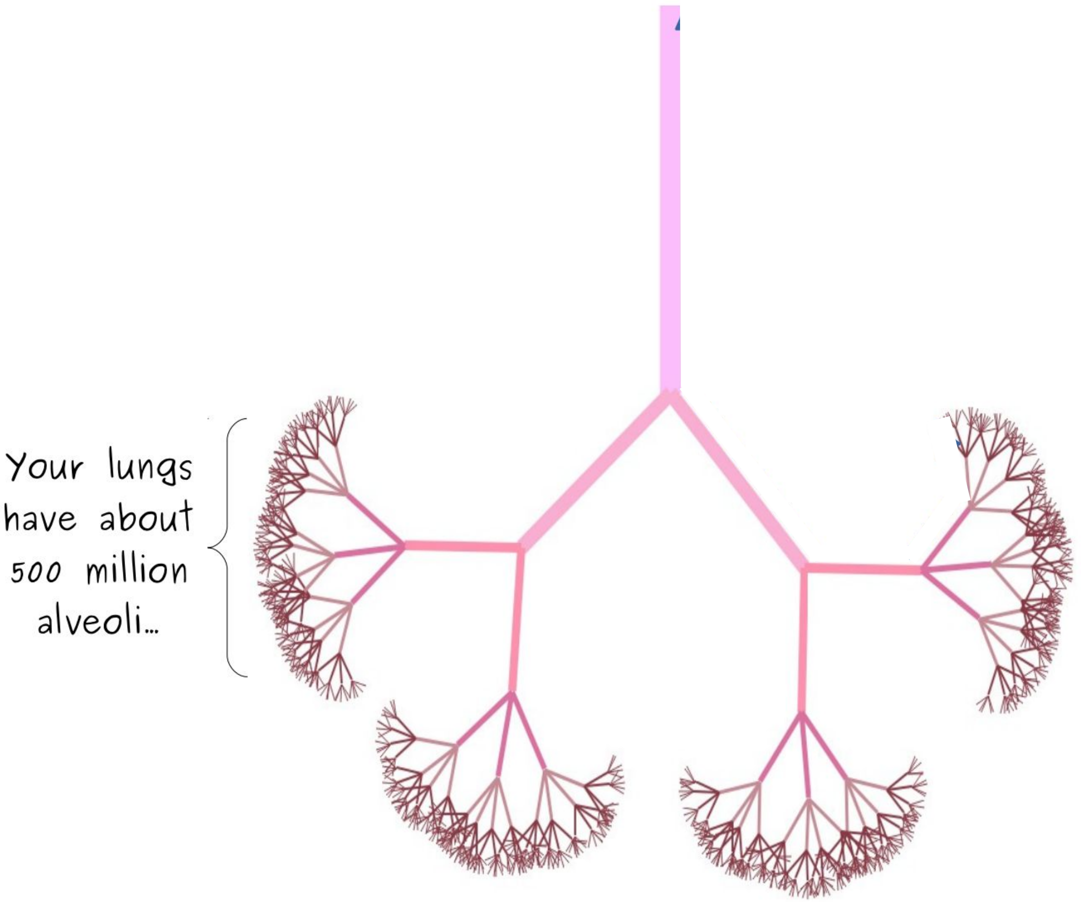
**Take a deep breath.**

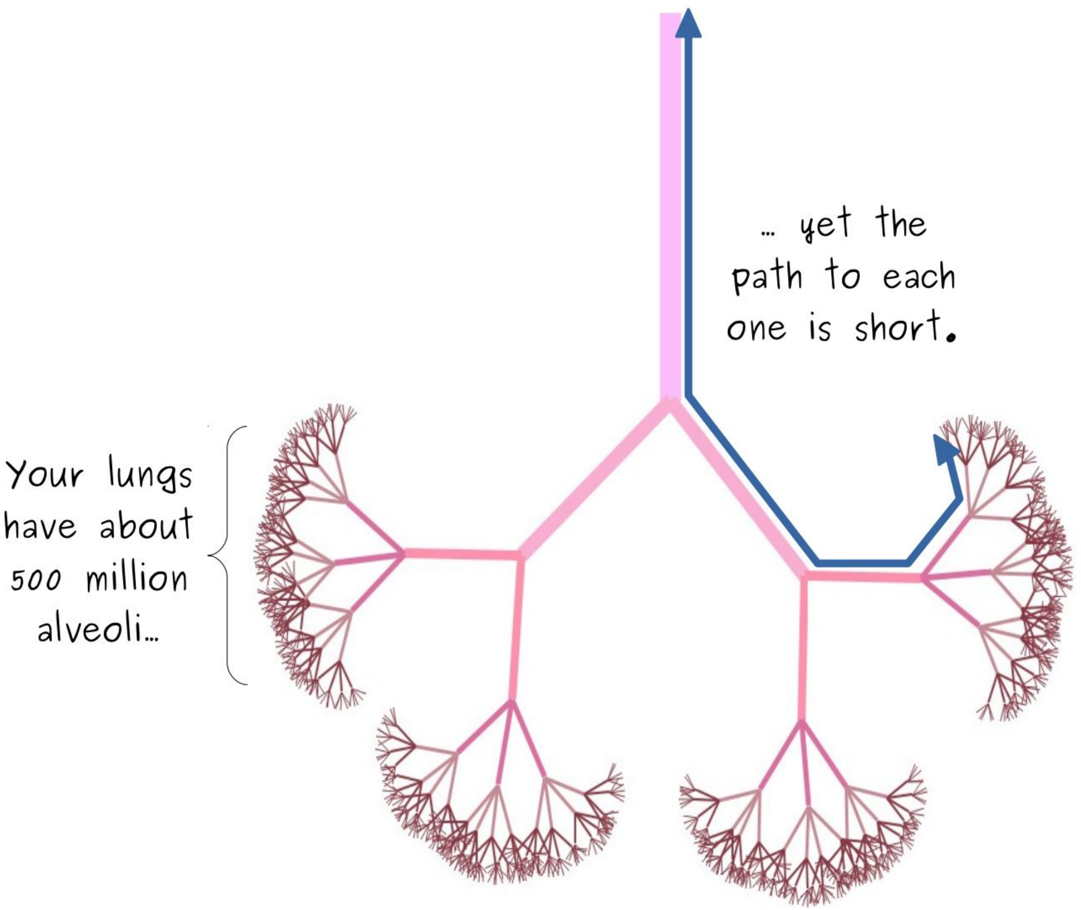
# And exhale...

**Feel nicely oxygenated?**



Beautiful art by Keith


Your lungs
have about
500 million
alveoli...


Your lungs have about 500 million alveoli...
... yet the path to each one is short.

**Key Idea:** The distance from each element in this structure to the top of the structure is small, even if there are many elements.

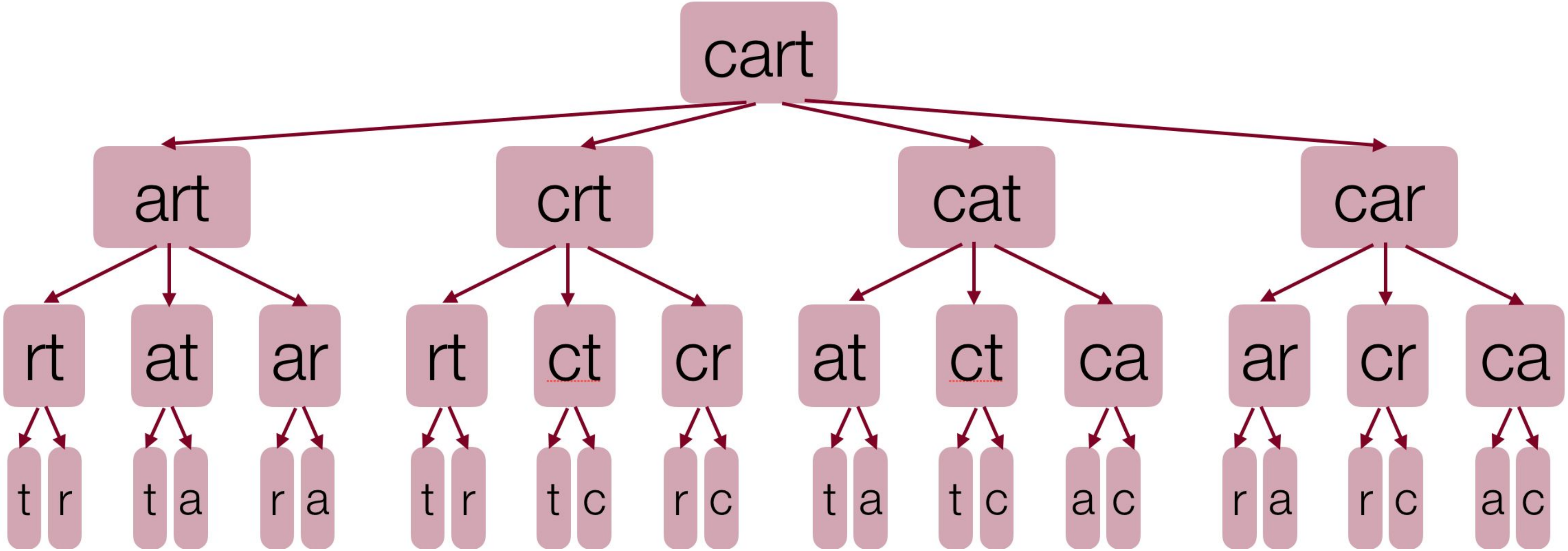
# Trees

# Throwback Thursday (on Monday)

- We've already seen trees before in this class… decision trees!

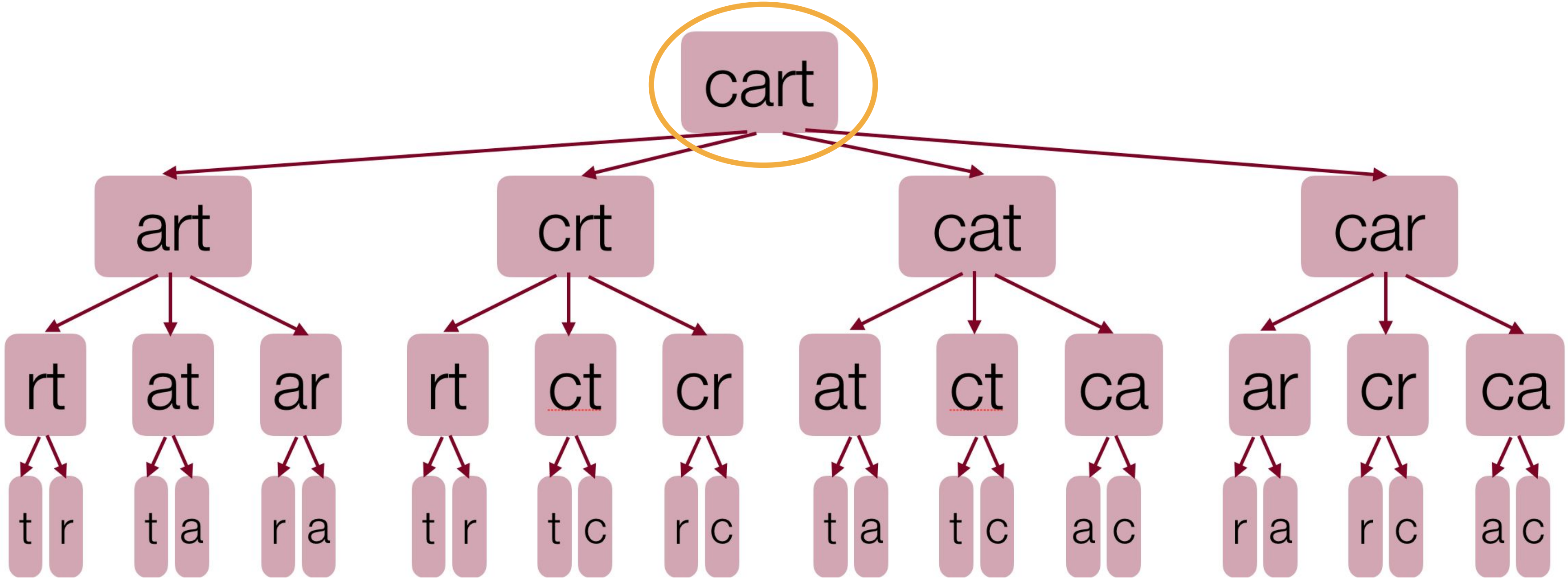
# Throwback Thursday (on Monday)

- We've already seen trees before in this class… decision trees!



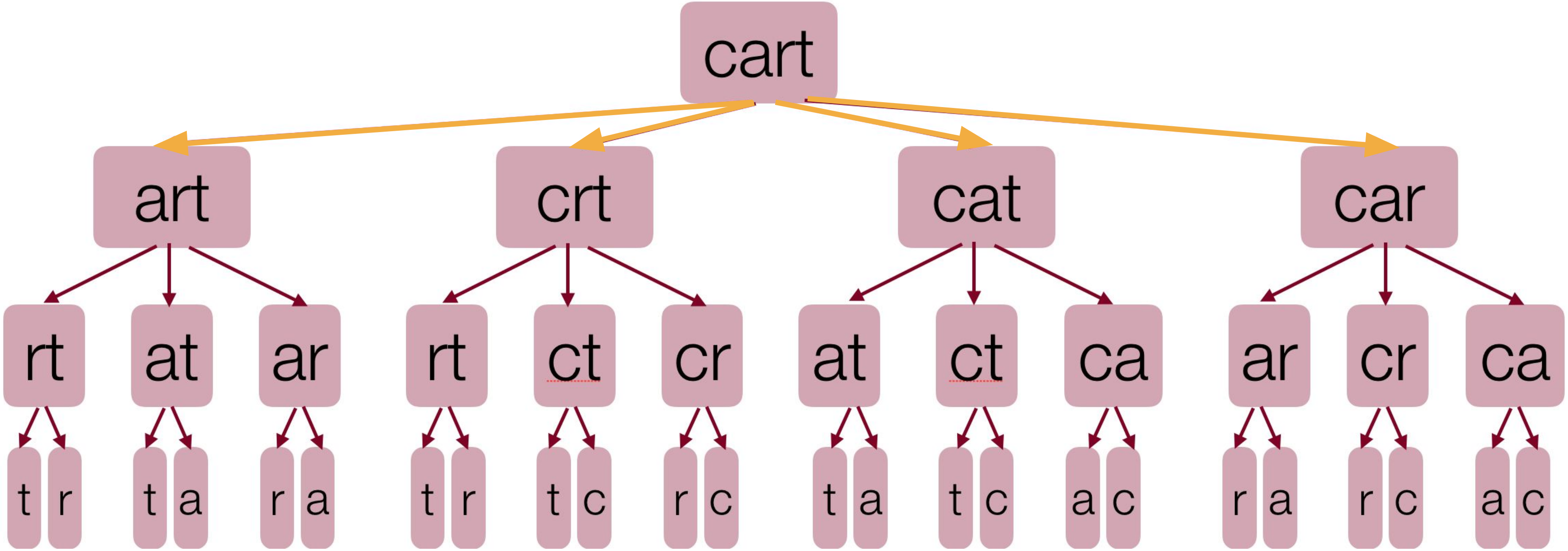
# Throwback Thursday (on Monday)

- We've already seen trees before in this class… decision trees!



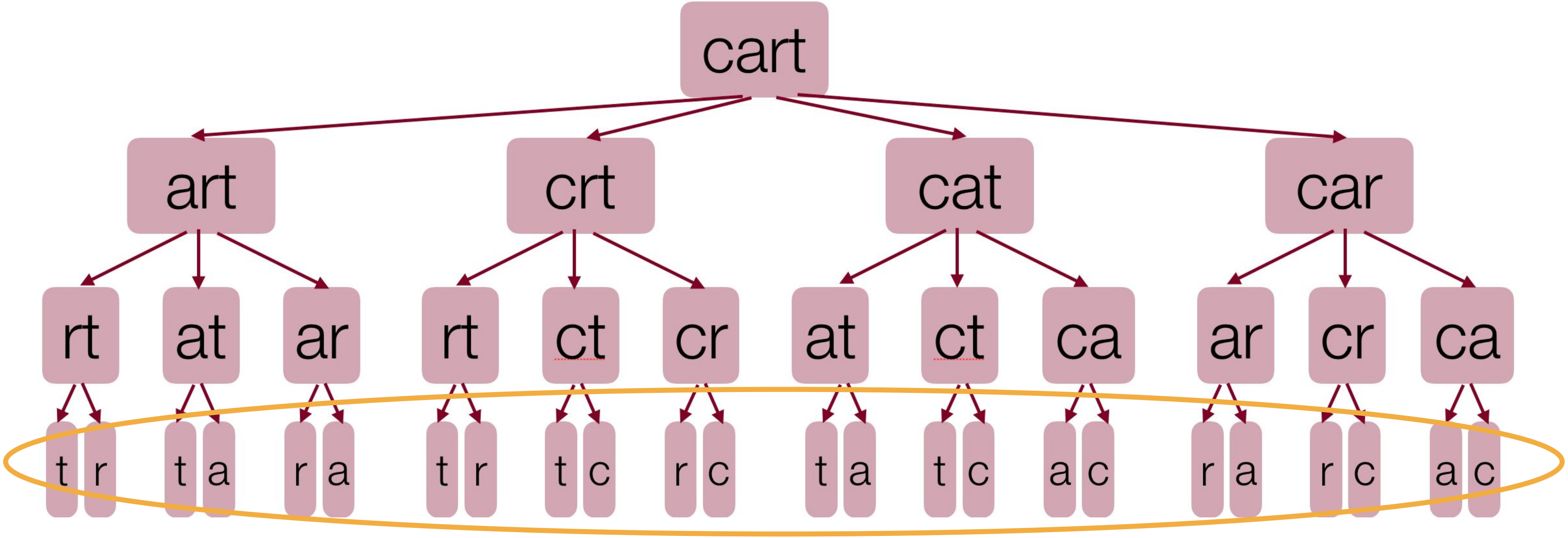
# Throwback Thursday (on Monday)

- We've already seen trees before in this class… decision trees!



# Throwback Thursday (on Monday)

- We've already seen trees before in this class… decision trees!

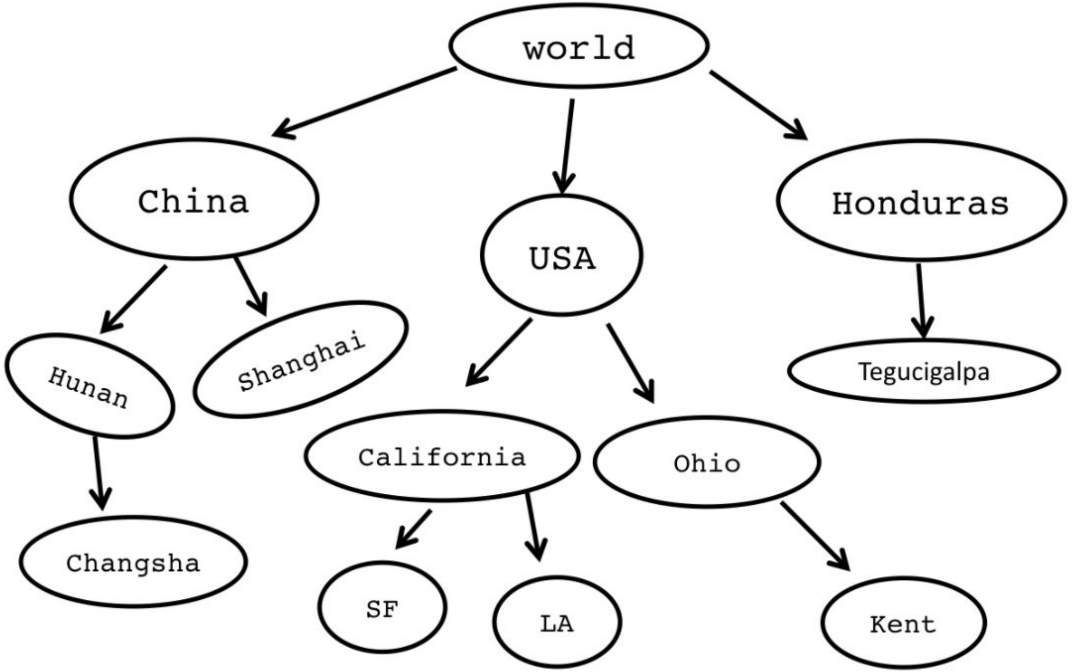


# Trees in the Wild

- Trees are useful in other ways besides just visualizing recursive backtracking.

# Trees in the Wild

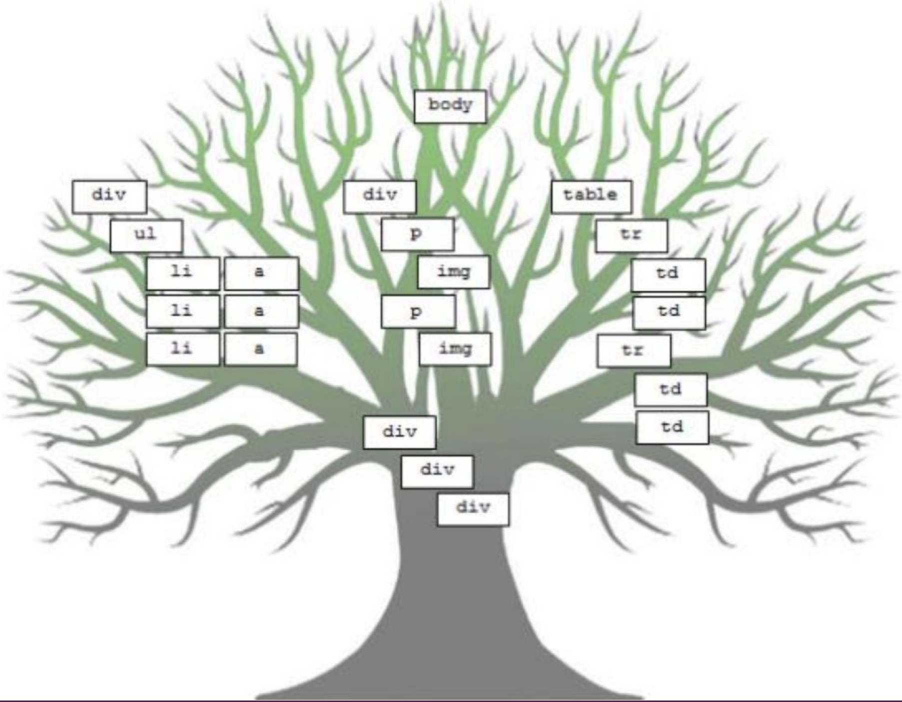
- Trees are useful in other ways besides just visualizing recursive backtracking.




Trees can be used to describe **hierarchies**.

# Trees in the Wild

- Trees are useful in other ways besides just visualizing recursive backtracking.

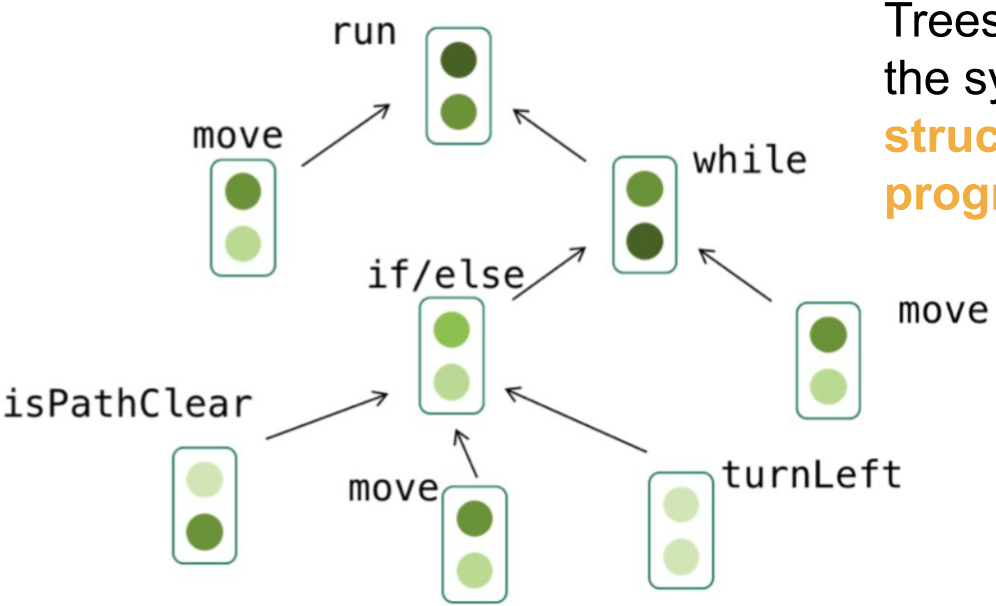



Trees are used to model the **structure of websites.**

# Trees in the Wild

- Trees are useful in other ways besides just visualizing recursive backtracking.

```
def run() {
  move();
  while (notFinished()) {
    if (isPathClear()) {
      move();
    } else {
      turnLeft();
    }
    move();
  }
}
```




Trees describe the syntax **structure of programs.**

# Trees in the Wild

- Trees are useful in other ways besides just visualizing recursive backtracking.

- But, it is not a coincidence that we first saw them appear in conjunction with recursion.

# Trees in the Wild

- Trees are useful in other ways besides just visualizing recursive backtracking.
- But, it is not a coincidence that we first saw them appear in conjunction with recursion.
- Trees are inherently defined recursively!

# What is a tree?

**A tree is either...**

# What is a tree?

**A tree is either...**

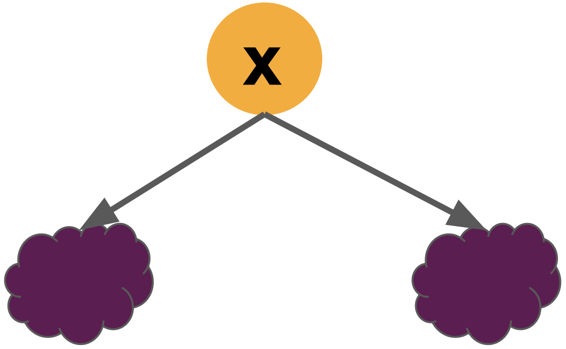
An empty data structure, or...

# What is a tree?

**A tree is either...**

An empty data structure, or...

A single node (parent), with zero or more non-empty subtrees (children)



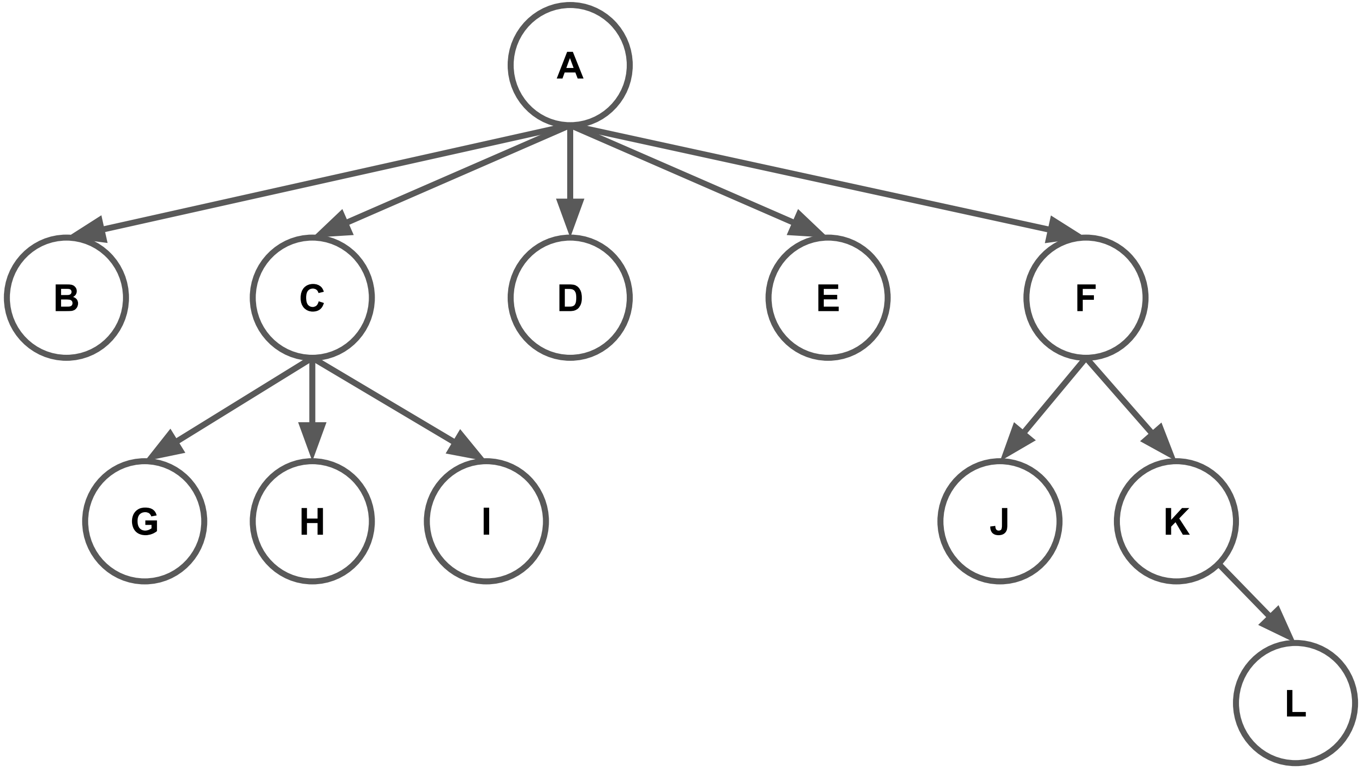
# Definition

**tree**

A tree is hierarchical data organization structure composed of a root value linked to zero or more non-empty subtrees.

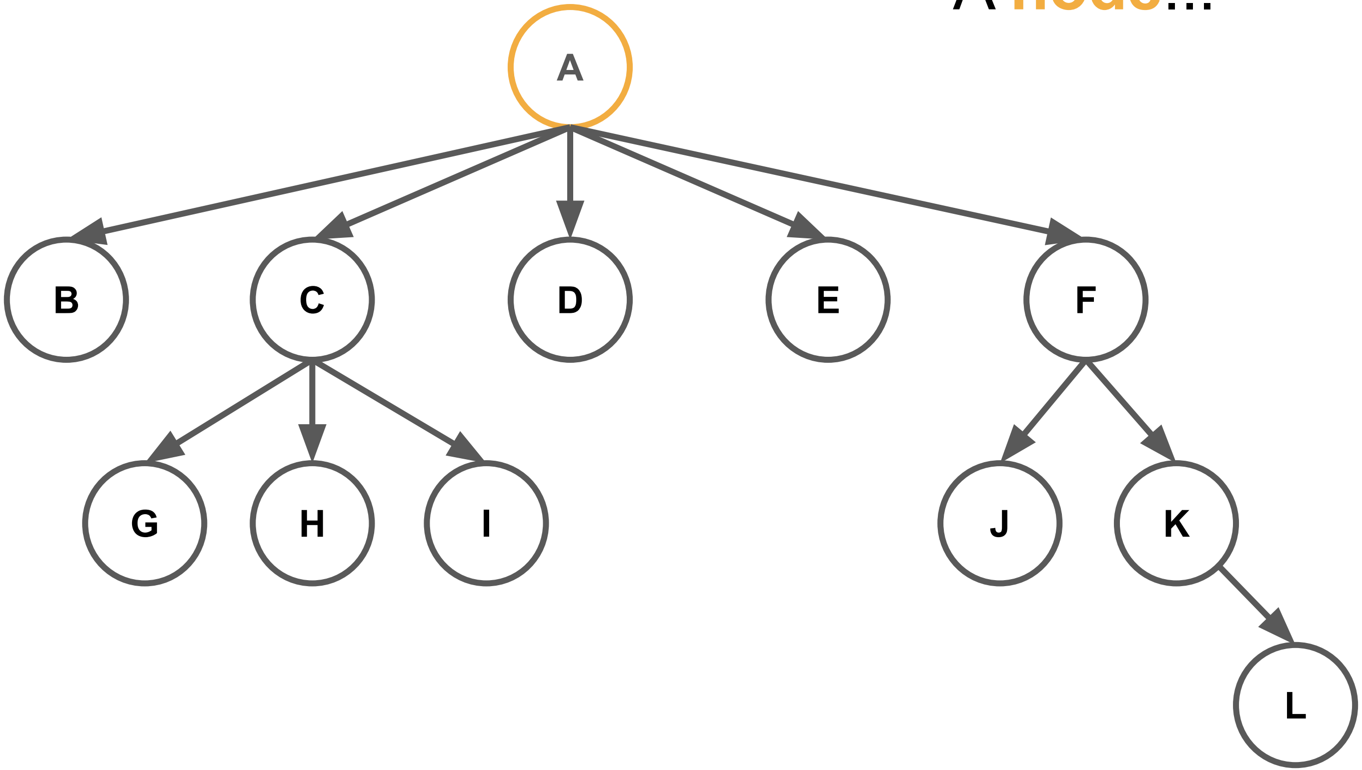
# Tree Terminology

# Tree Terminology

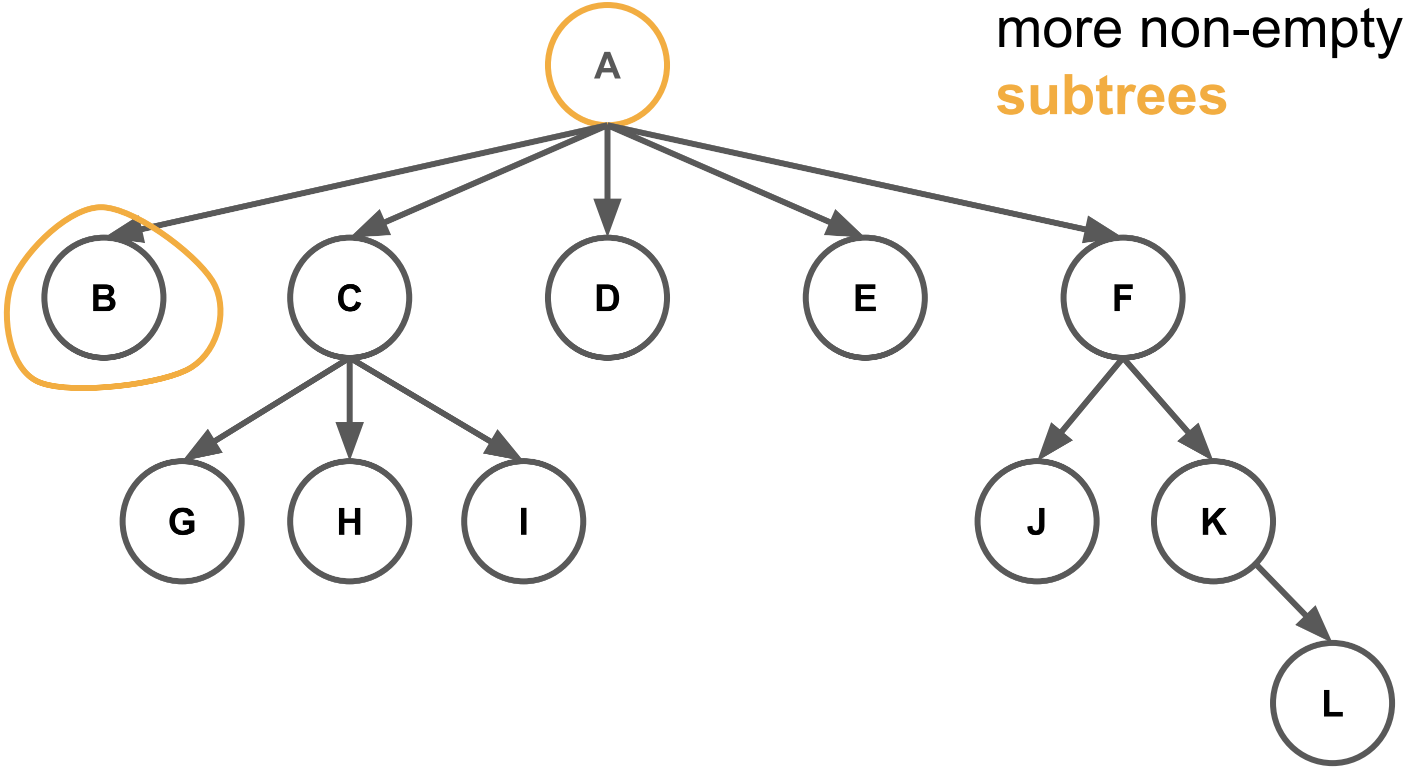


# Tree Terminology

A **node**...

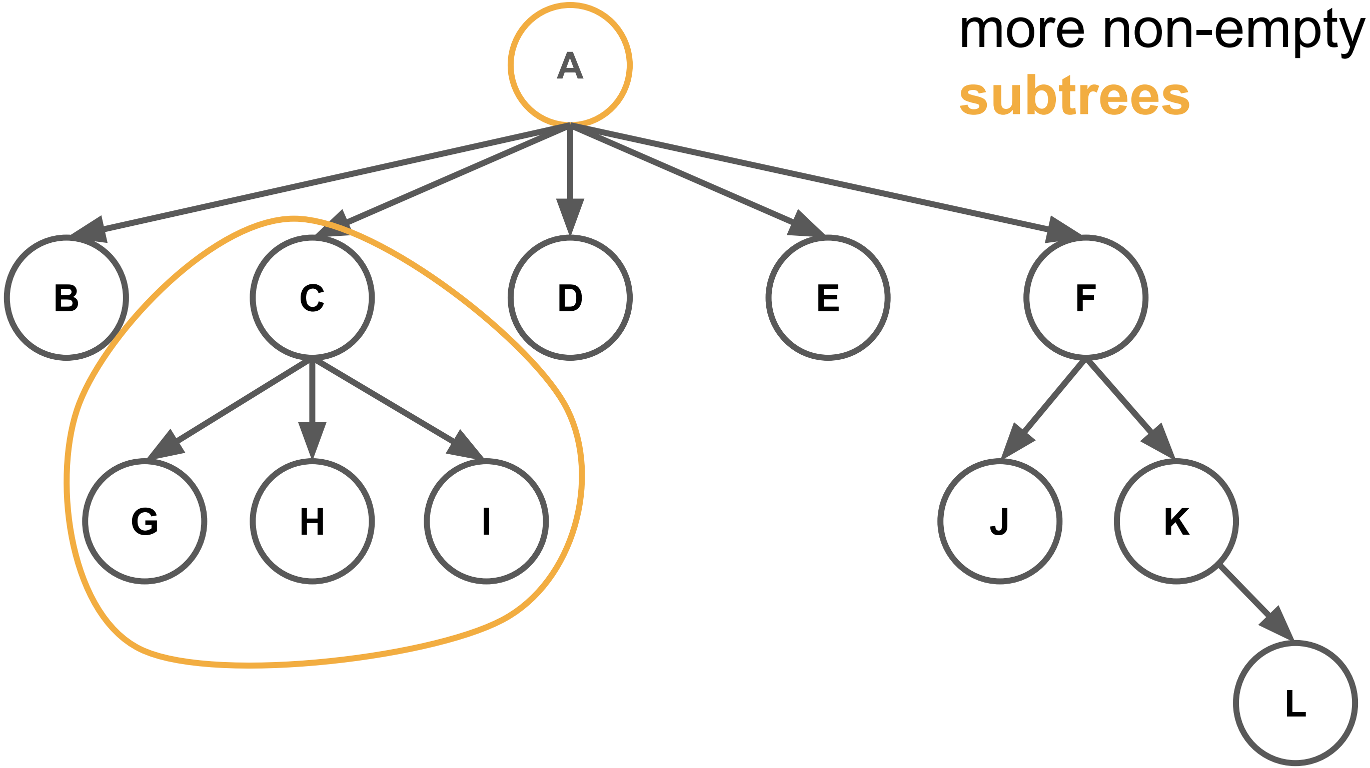


# Tree Terminology



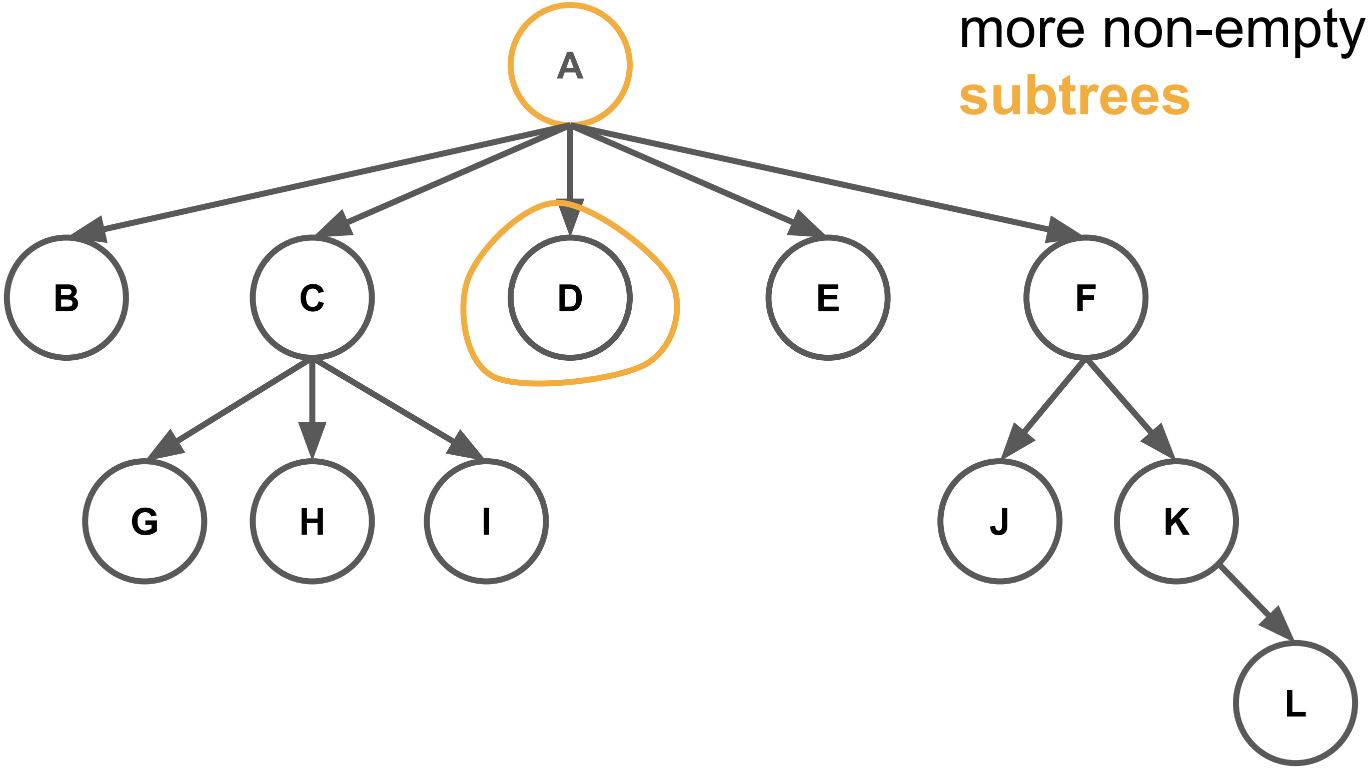

A **node** with 0 or more non-empty **subtrees**

# Tree Terminology



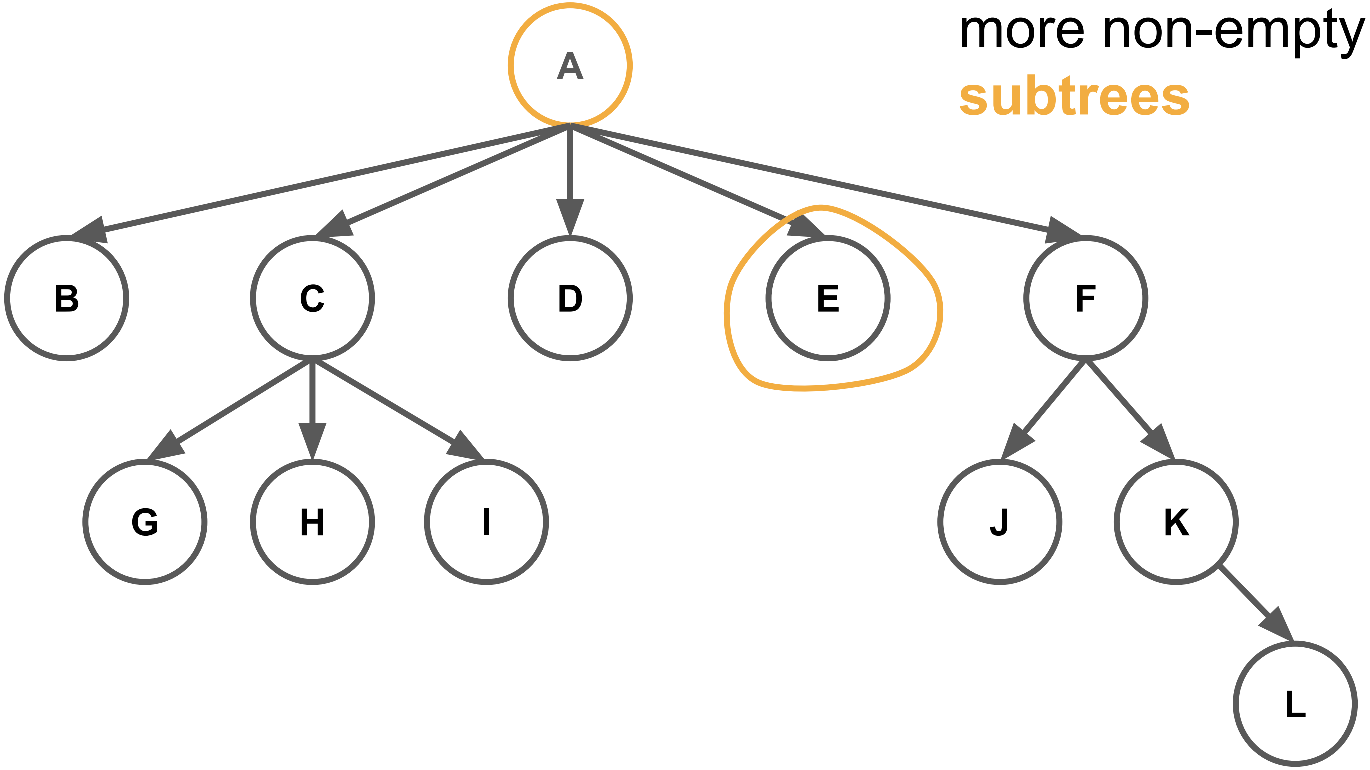

A **node** with 0 or more non-empty **subtrees**

# Tree Terminology



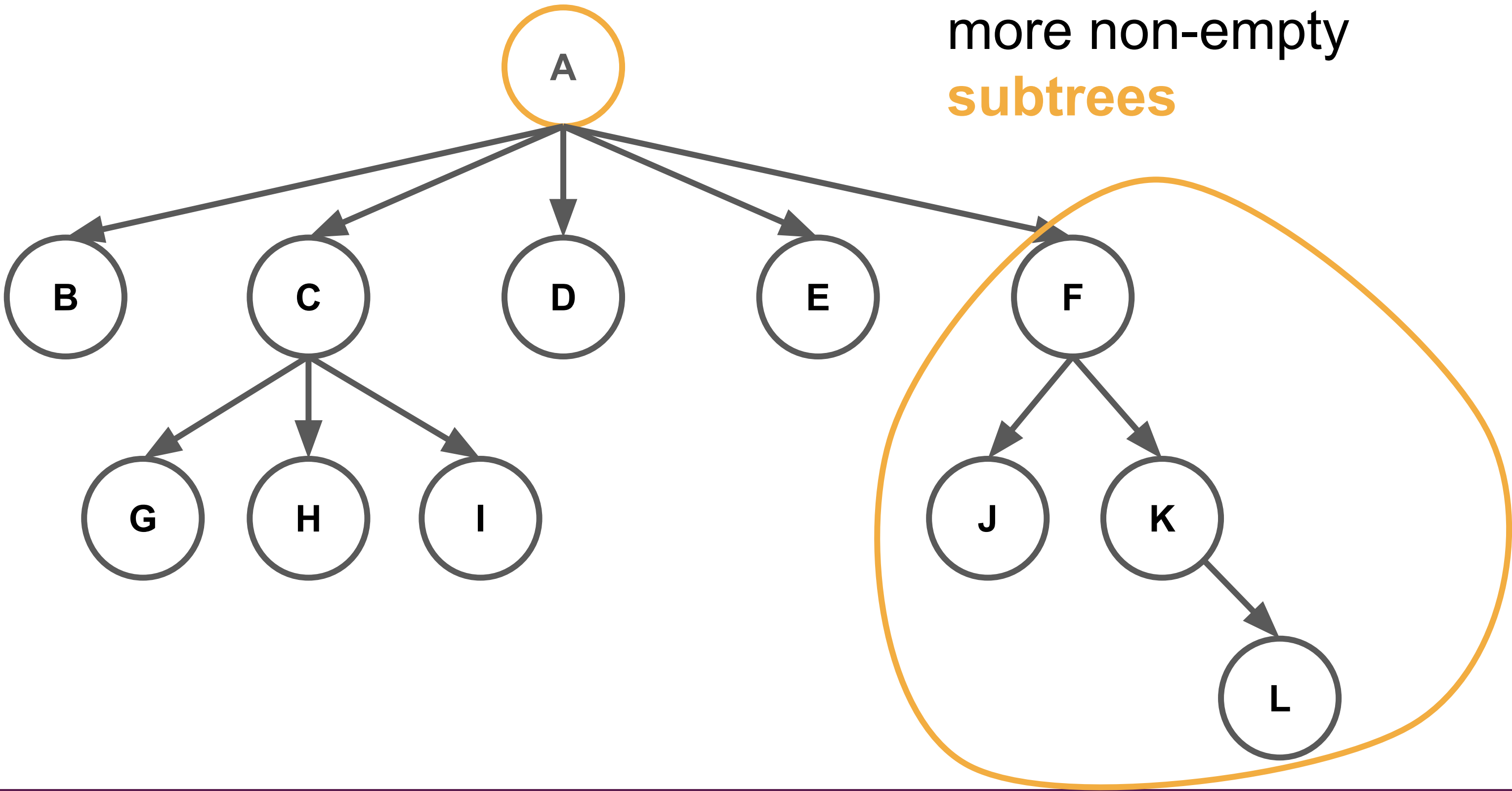

A **node** with 0 or more non-empty **subtrees**

# Tree Terminology



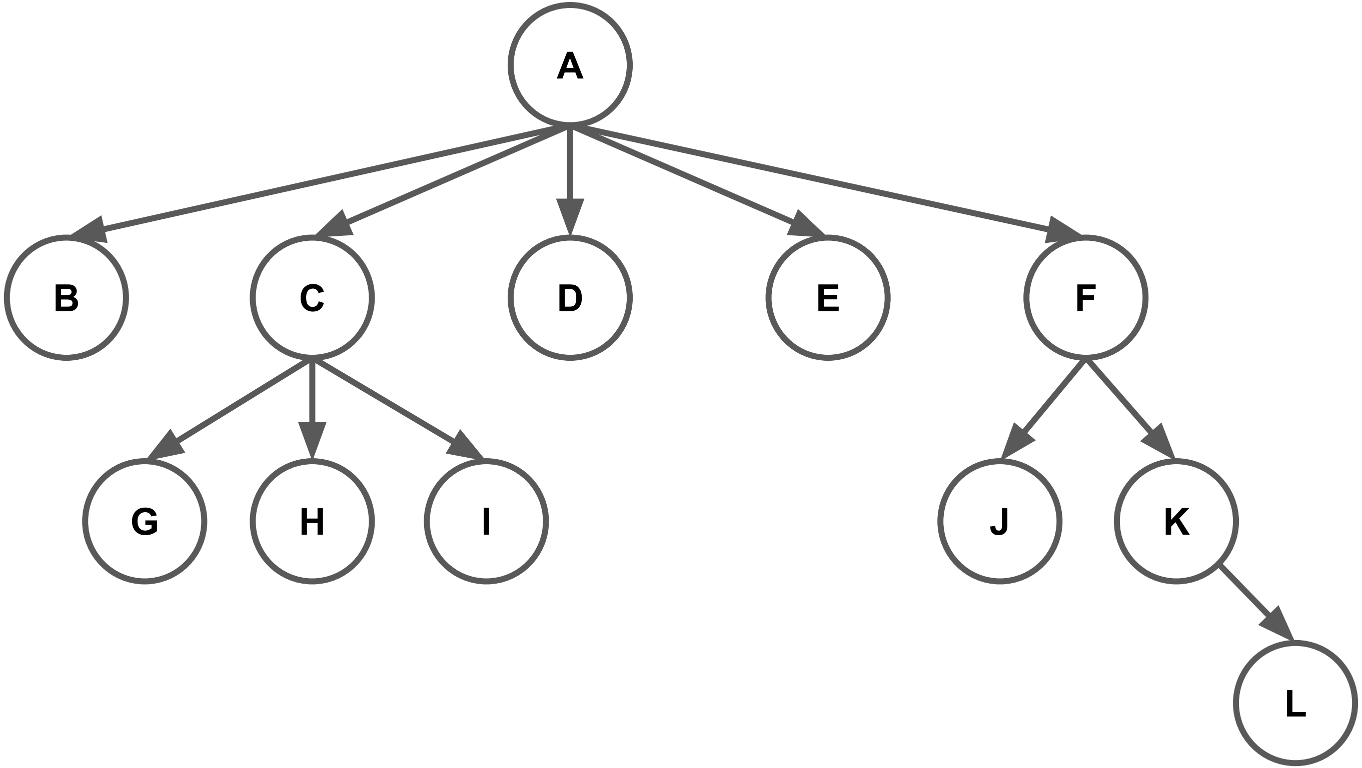

A **node** with 0 or more non-empty **subtrees**

# Tree Terminology

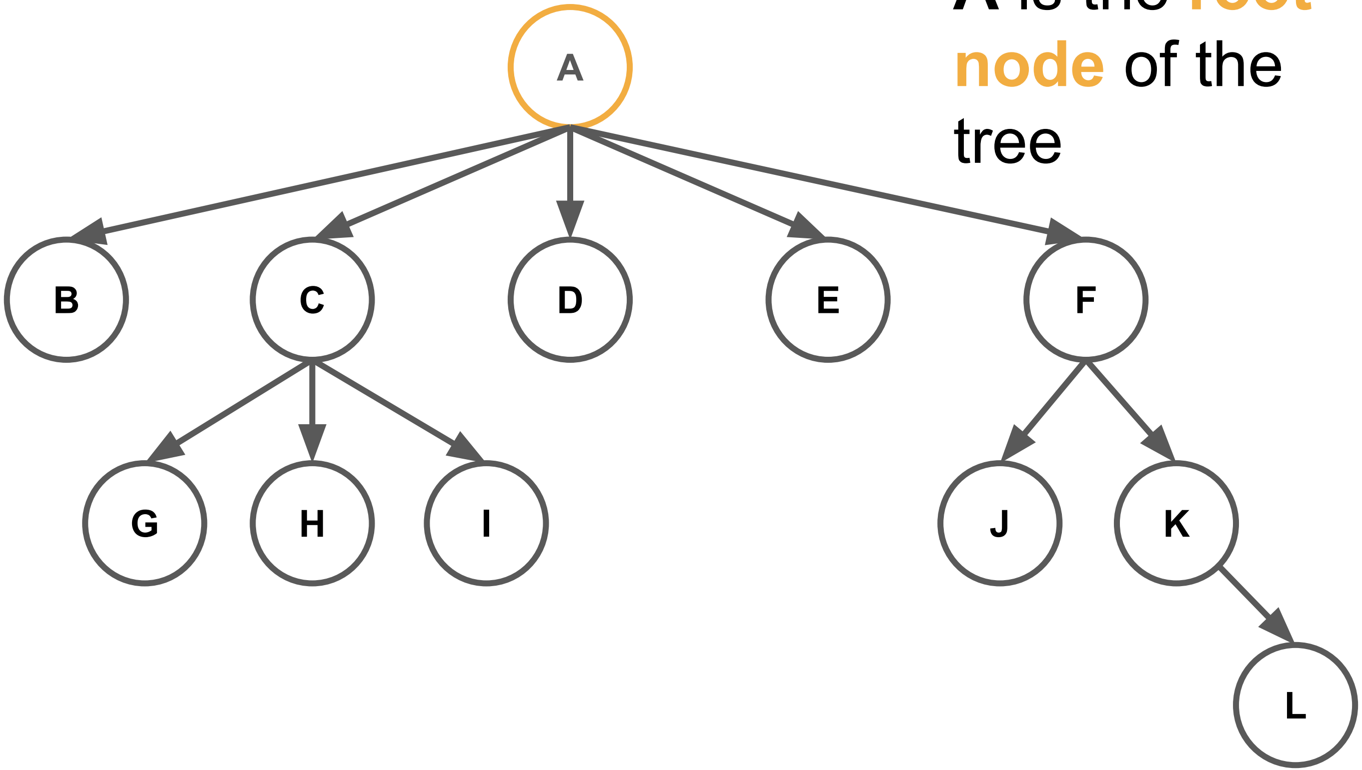



A **node** with 0 or more non-empty **subtrees**

# Tree Terminology

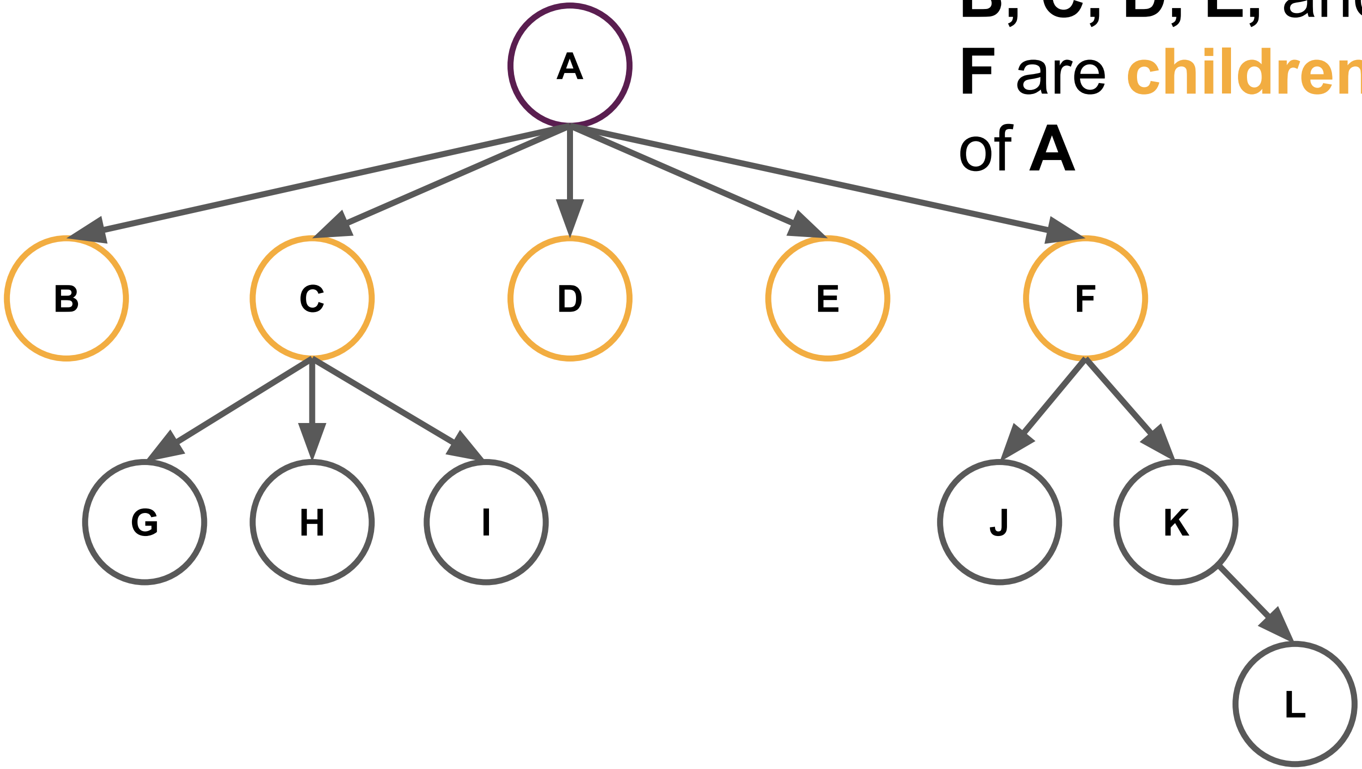


# Tree Terminology



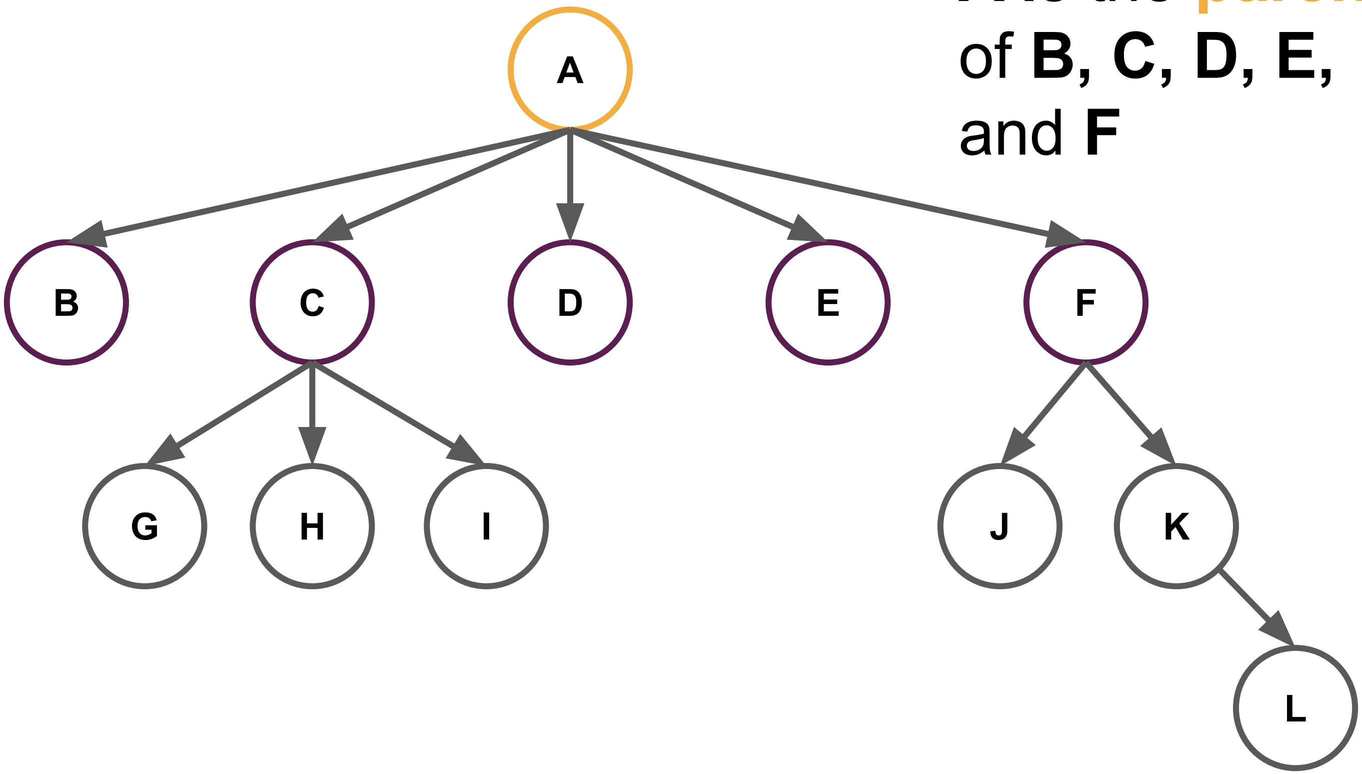

**A** is the **root node** of the tree

# Tree Terminology



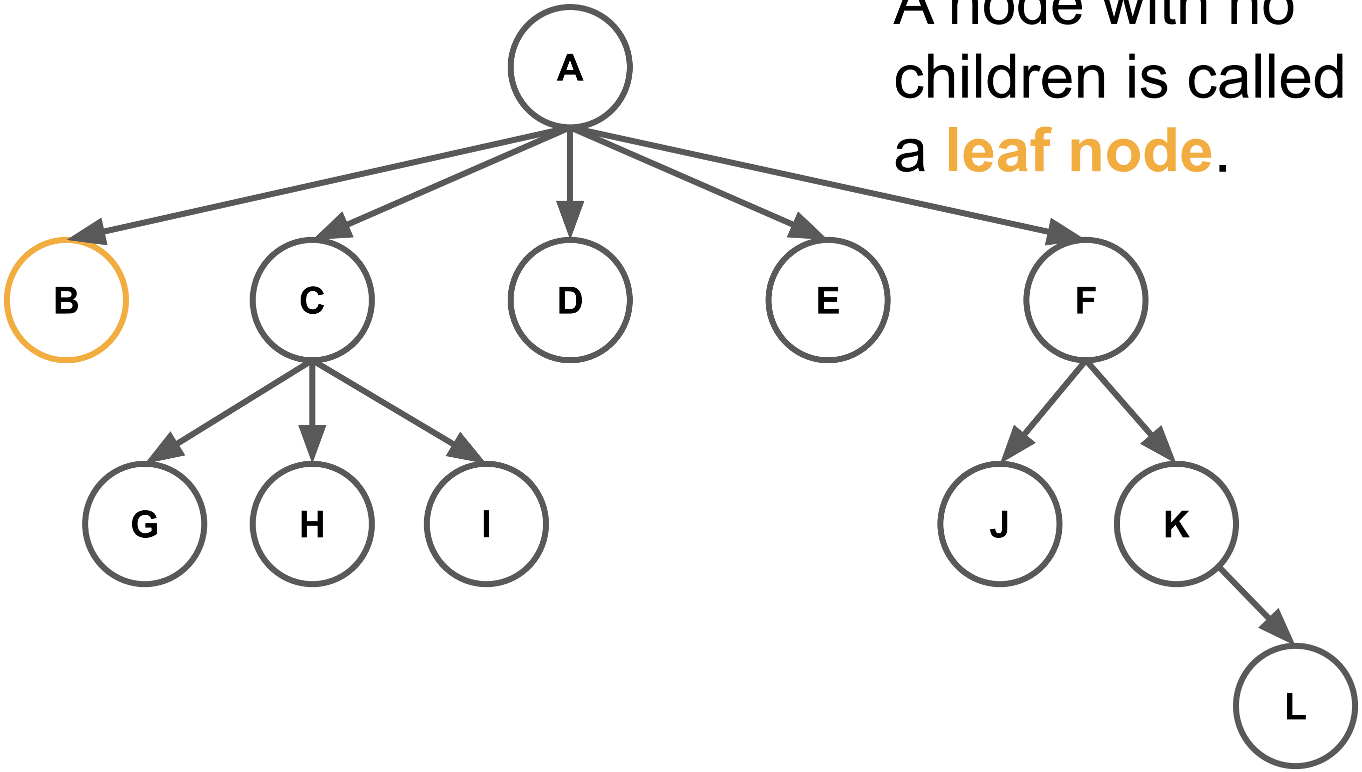

**B**, **C**, **D**, **E**, and **F** are **children** of **A**

# Tree Terminology




**A** is the parent of **B, C, D, E,** and **F**

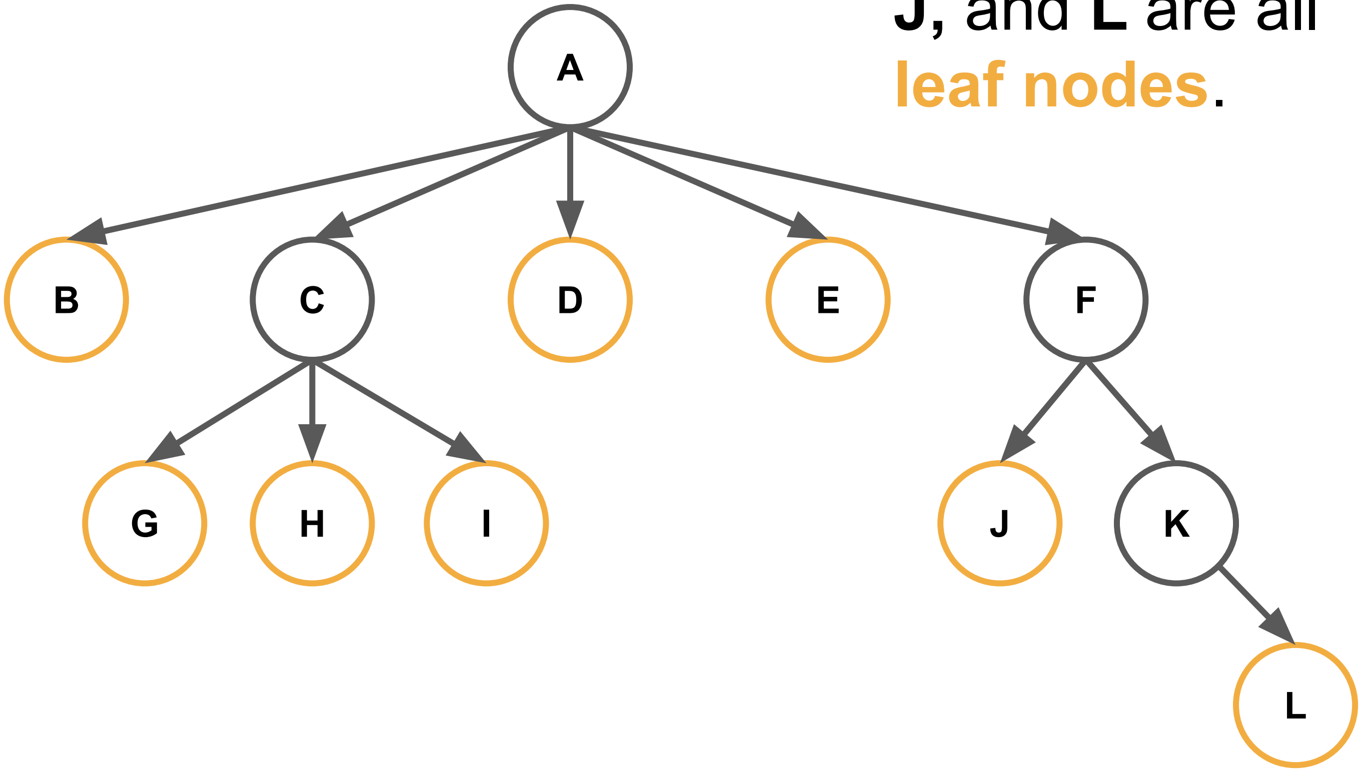
# Tree Terminology



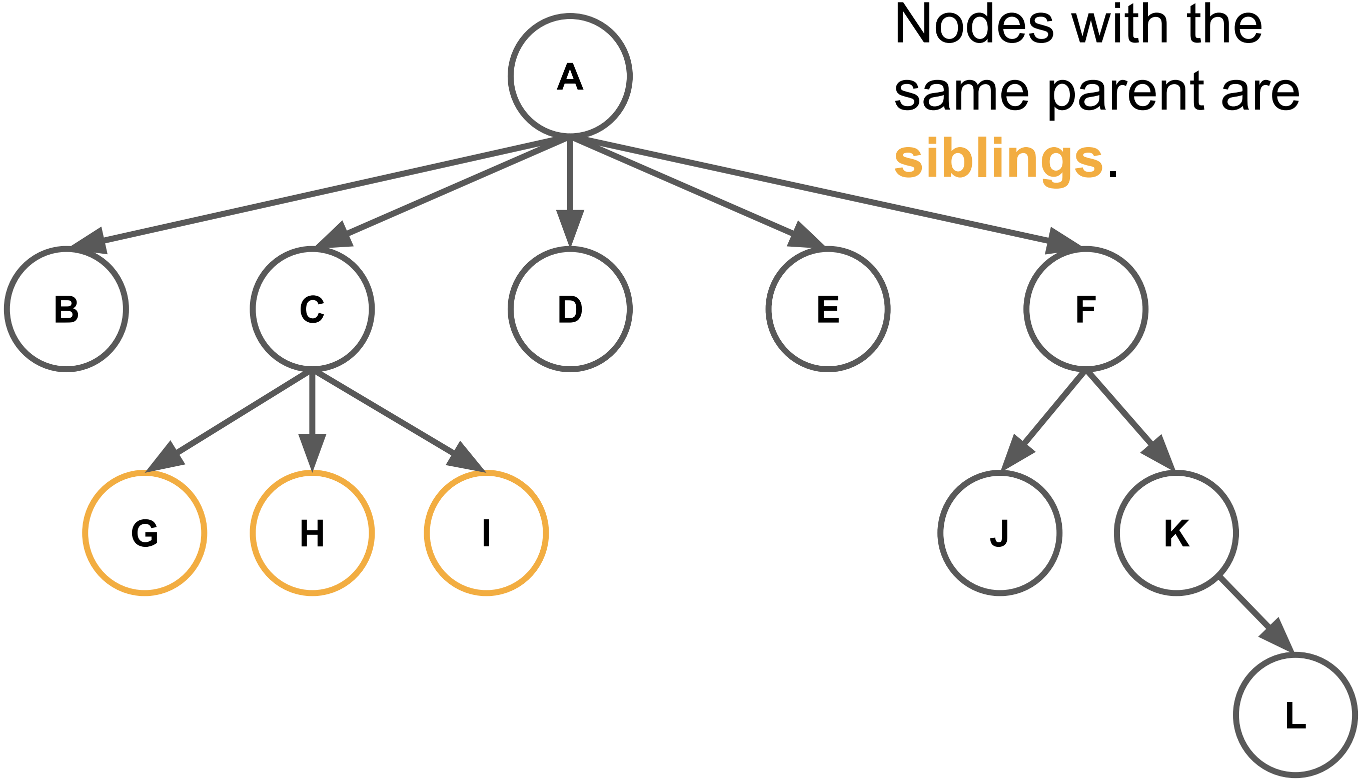

**B** has no children. A node with no children is called a **leaf node**.

# Tree Terminology

**B**, **G**, **H**, **I**, **D**, **E**, **J**, and **L** are all **leaf nodes**.

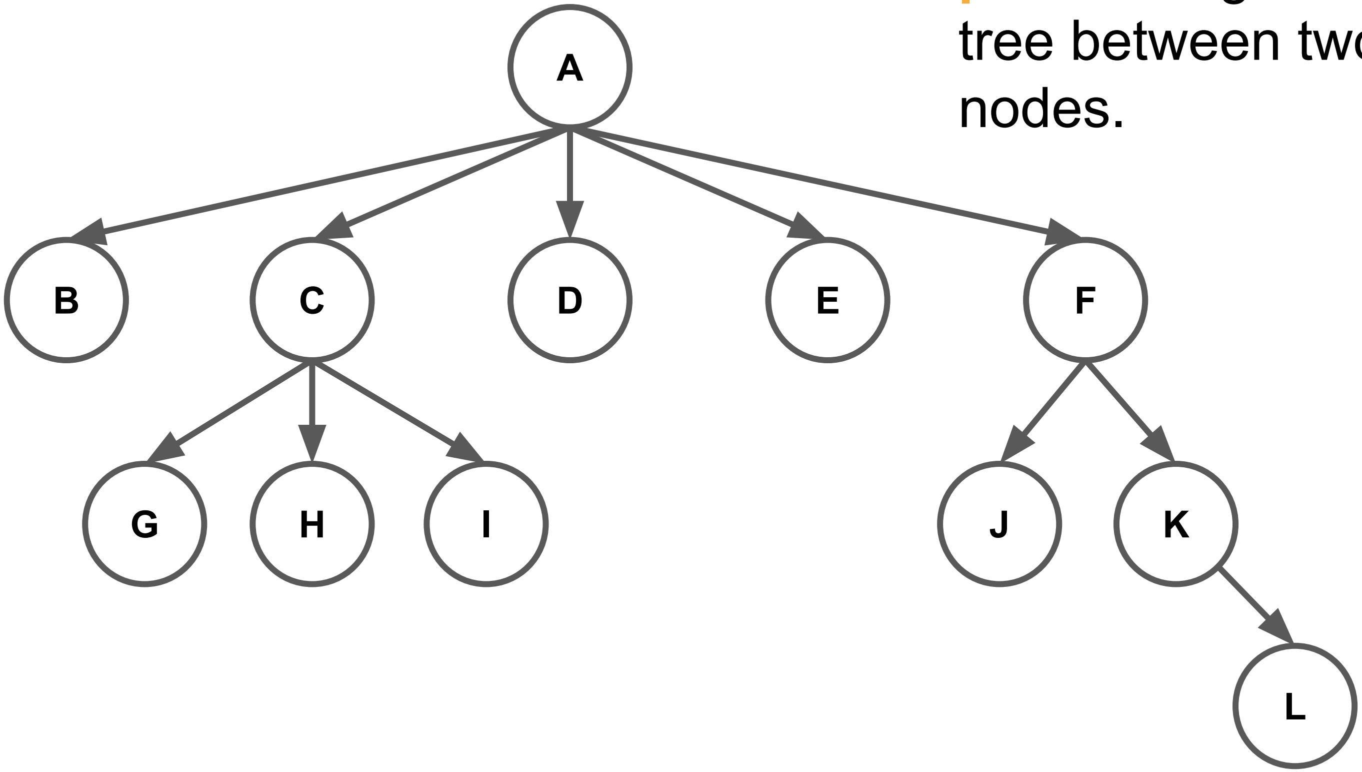


# Tree Terminology




**G**, **H** and **I** all have the same parent. Nodes with the same parent are **siblings**.

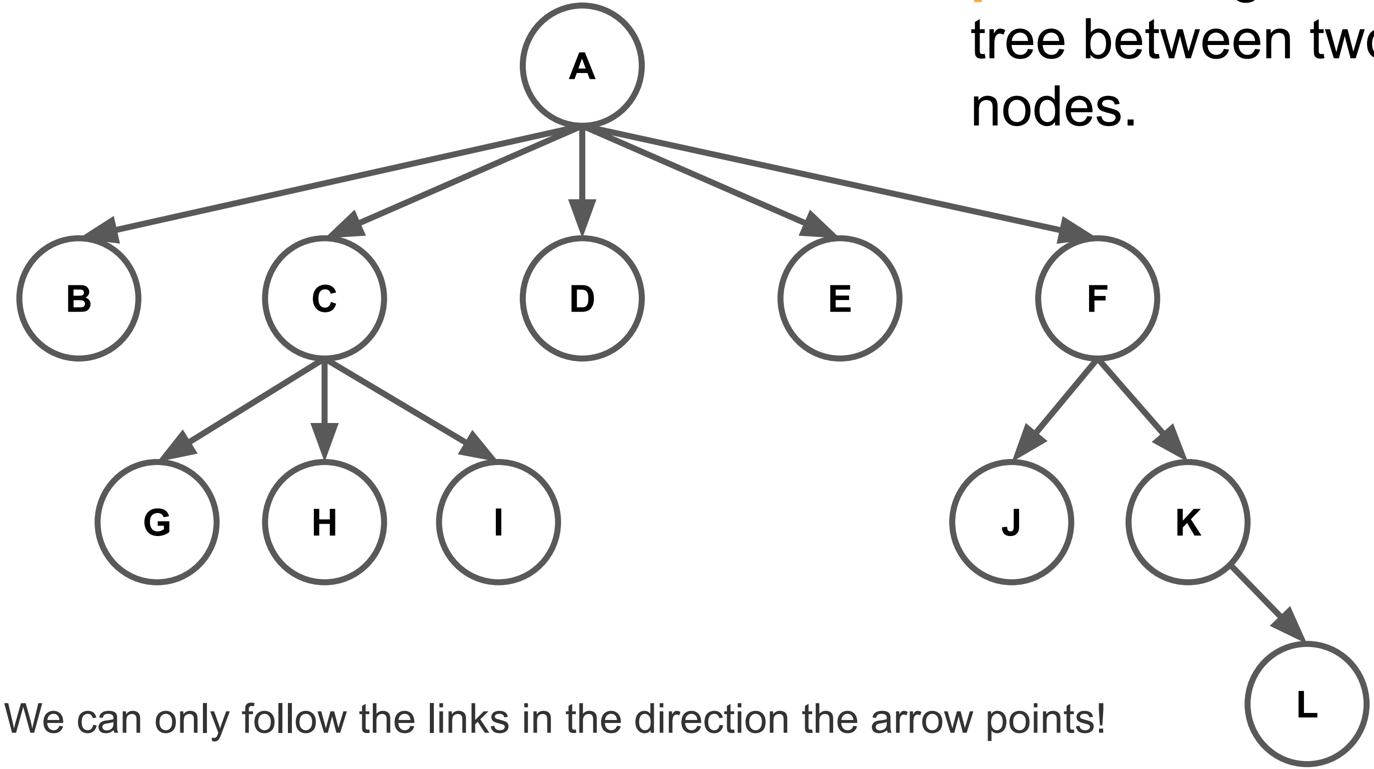
# Tree Terminology




We can define a **path** through the tree between two nodes.

# Tree Terminology

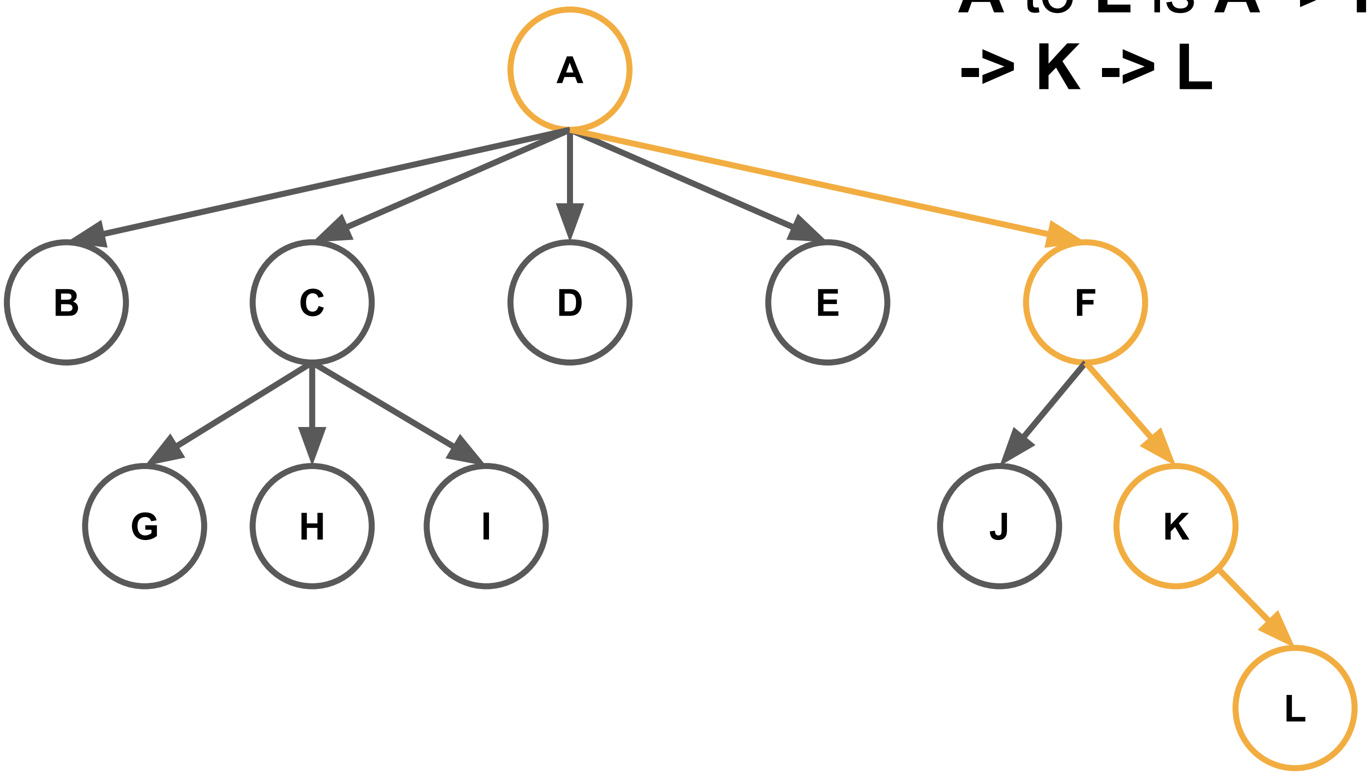
We can define a **path** through the tree between two nodes.




**Note:** We can only follow the links in the direction the arrow points!

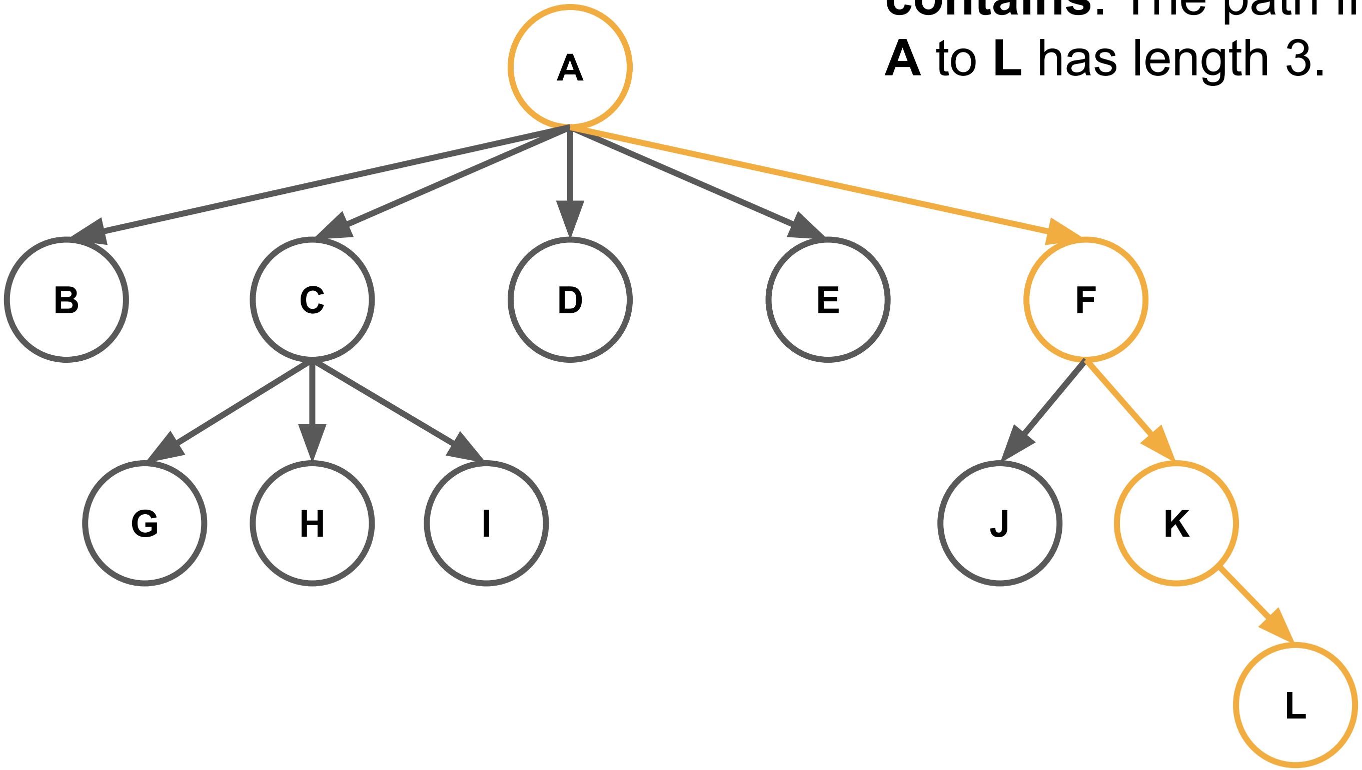
# Tree Terminology

The **path** from **A** to **L** is **A -> F -> K -> L**



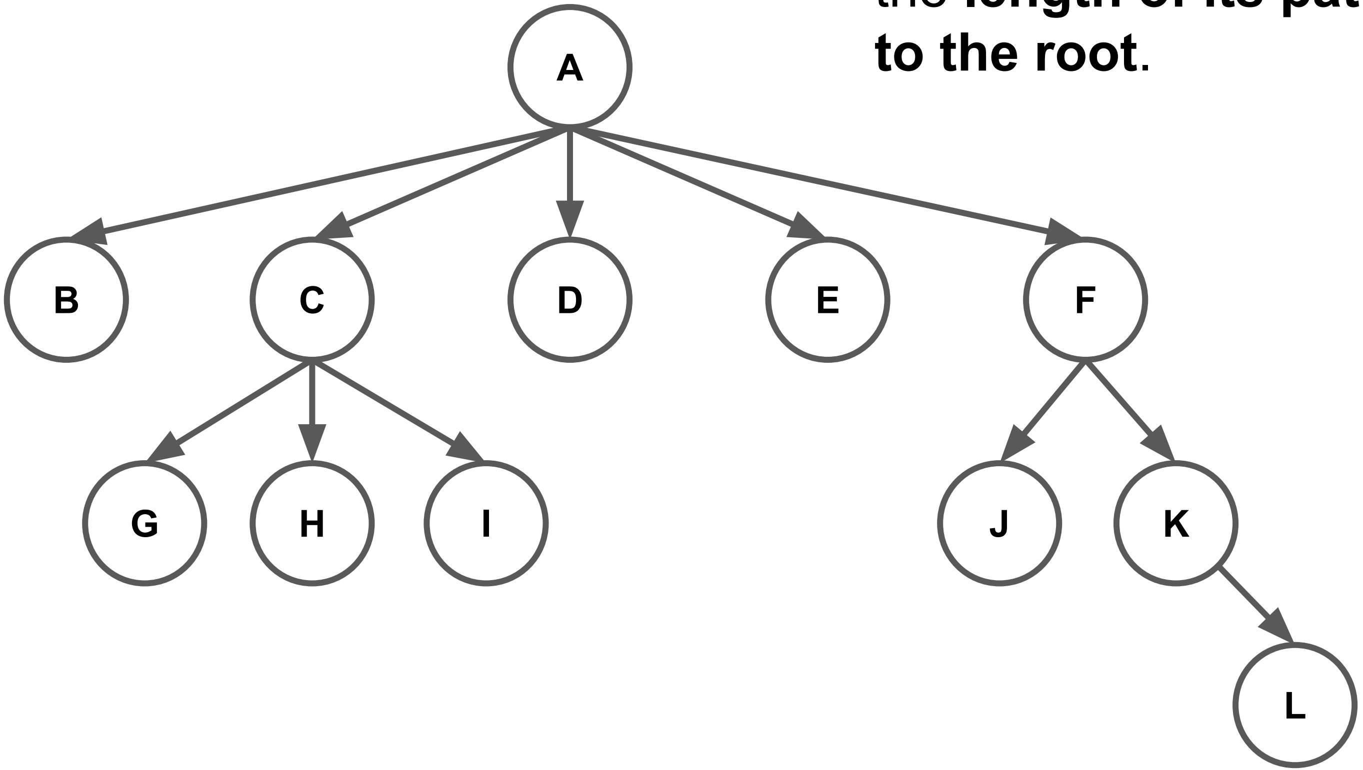
# Tree Terminology

The length of the path is **number of edges it contains**. The path from **A** to **L** has length 3.



# Tree Terminology

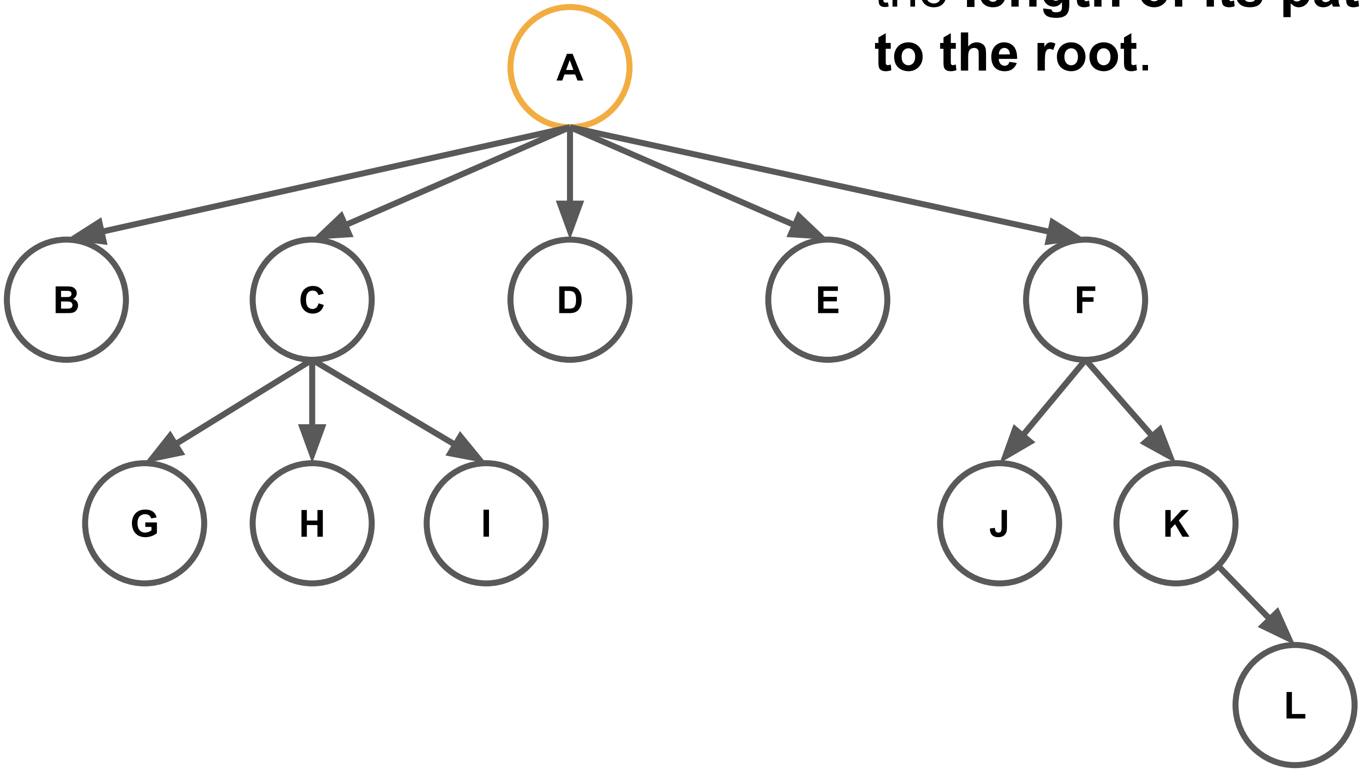
The **depth** of a node is the **length of its path to the root**.



# Tree Terminology

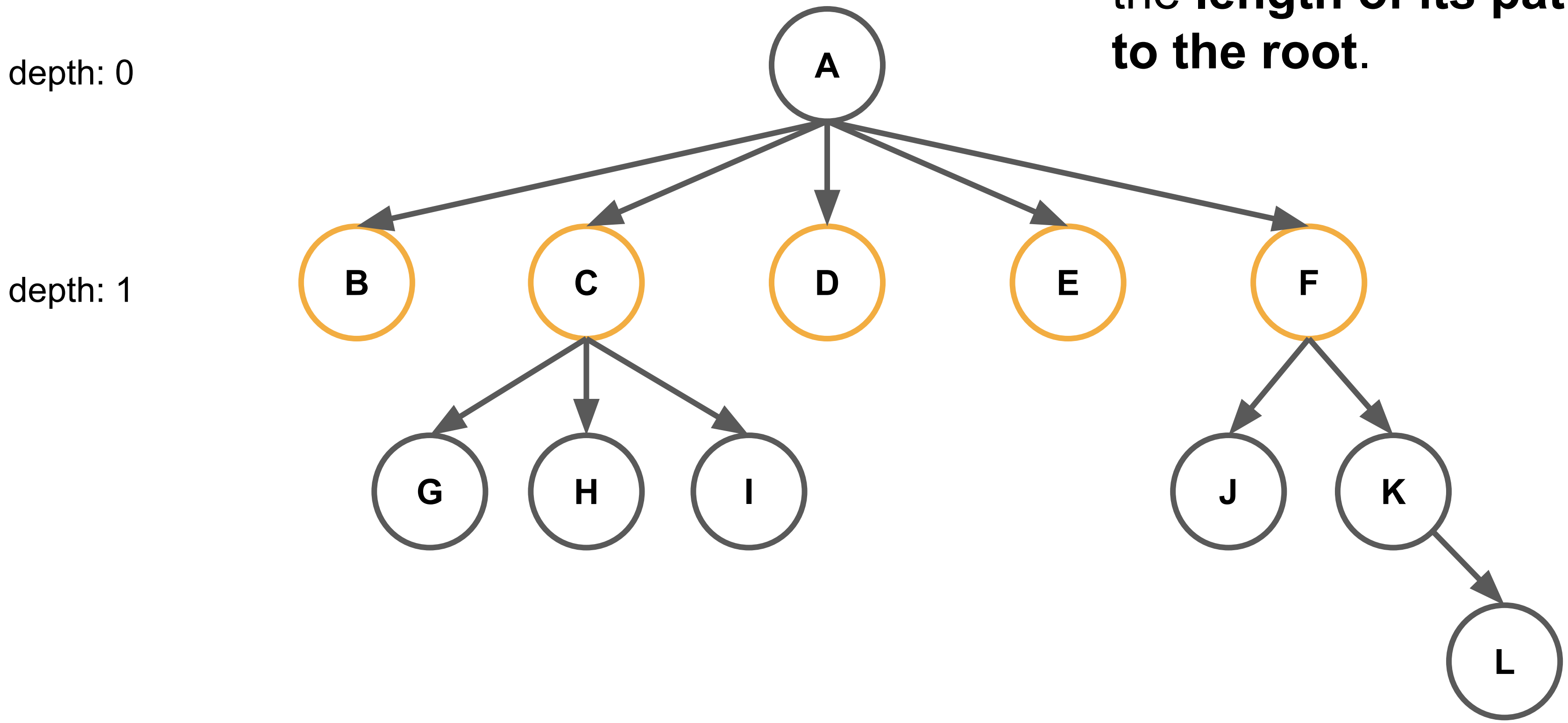
depth: 0

The **depth** of a node is the **length of its path to the root**.



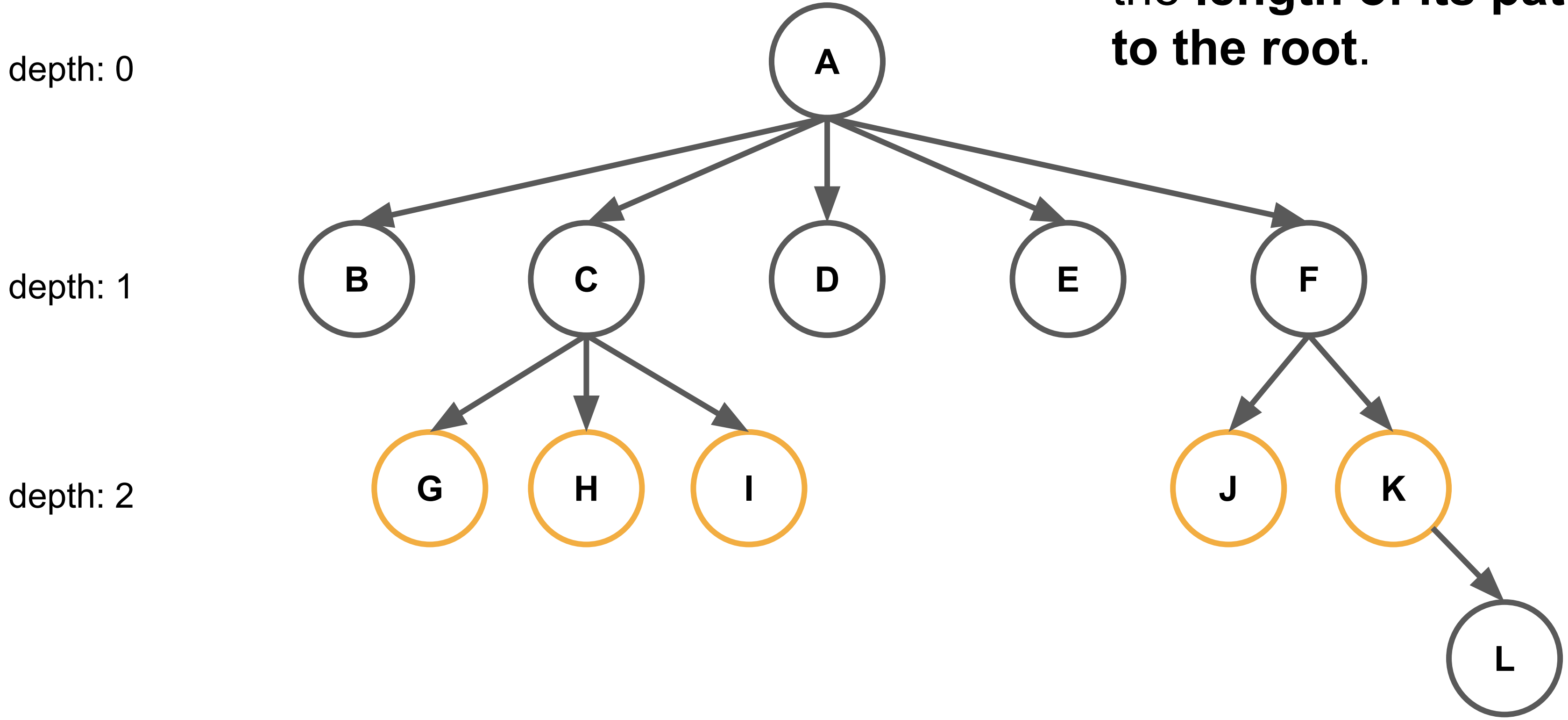
# Tree Terminology

The **depth** of a node is the **length of its path to the root**.



# Tree Terminology

The **depth** of a node is the **length of its path to the root**.



# Tree Terminology

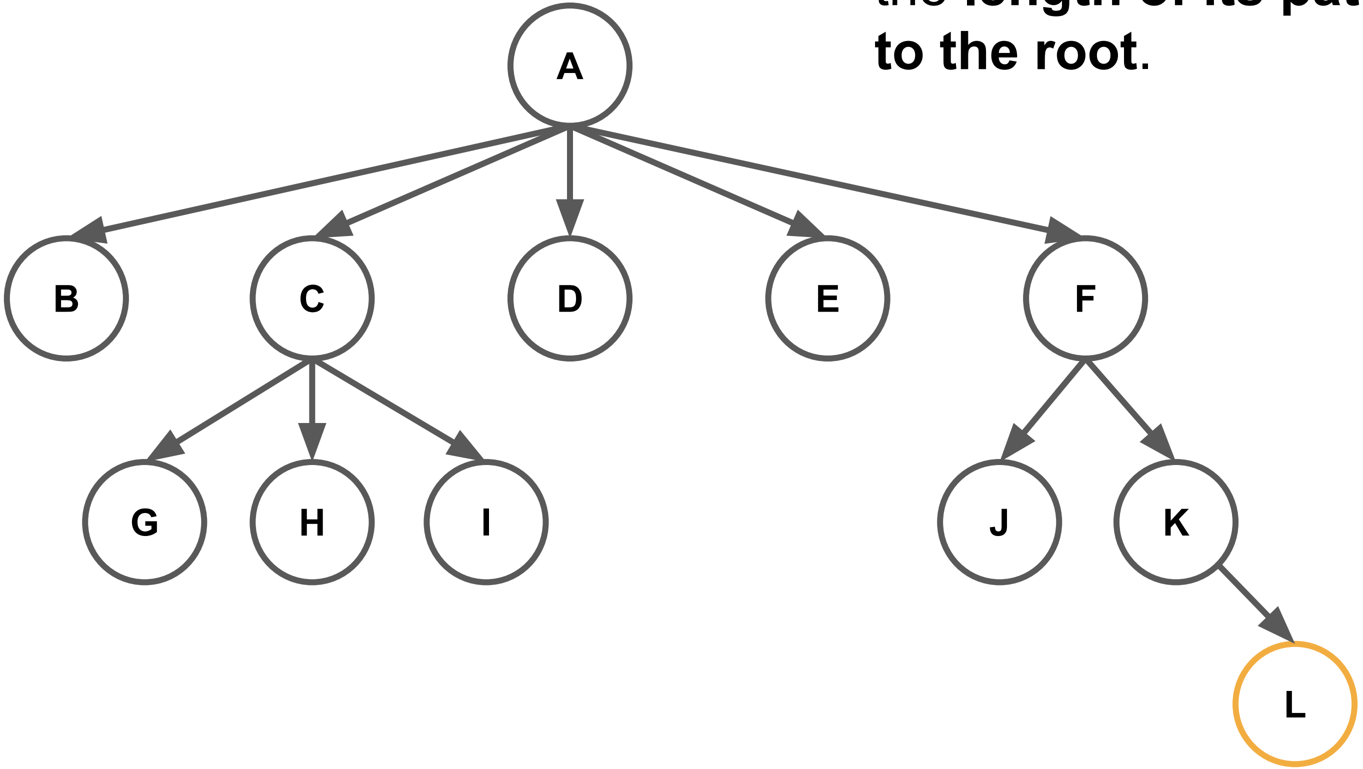
The depth of a node is the **length of its path to the root**.

depth: 0

depth: 1

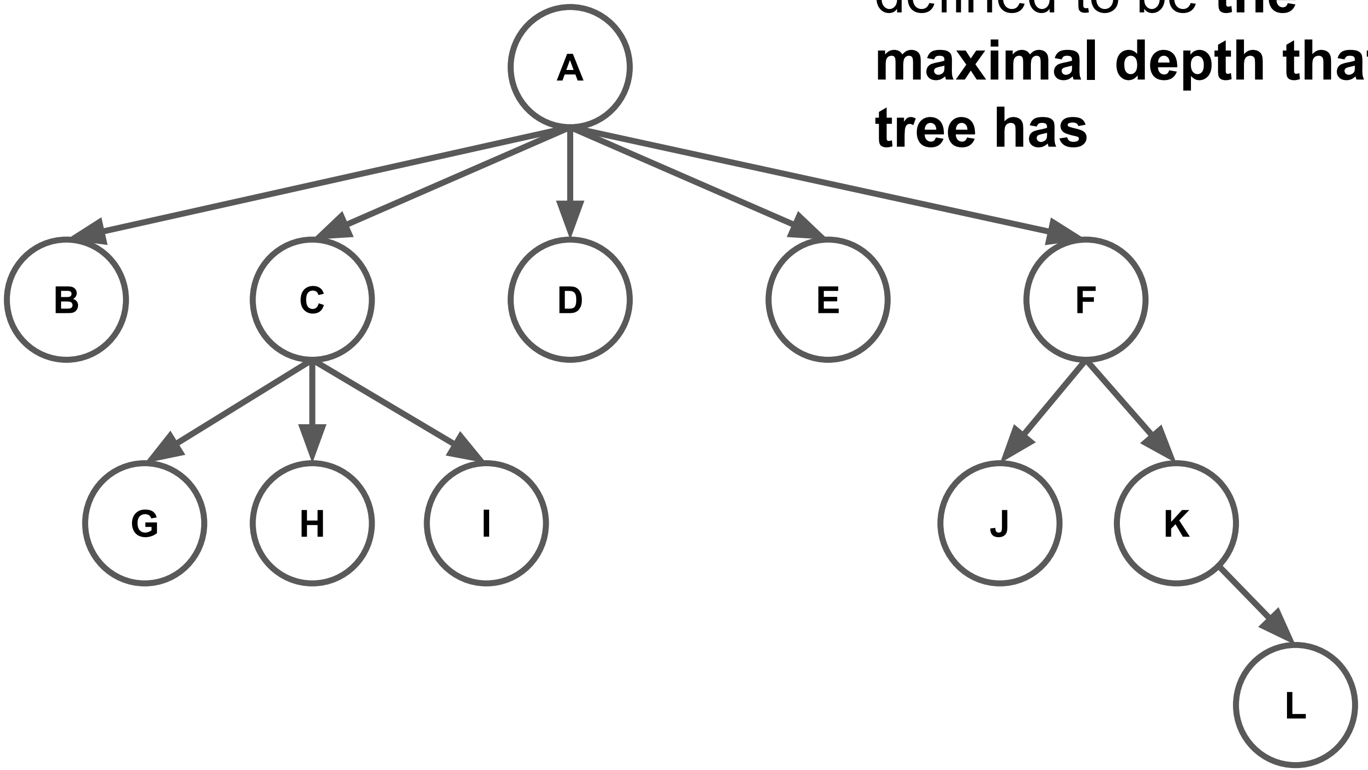
depth: 2

depth: 3



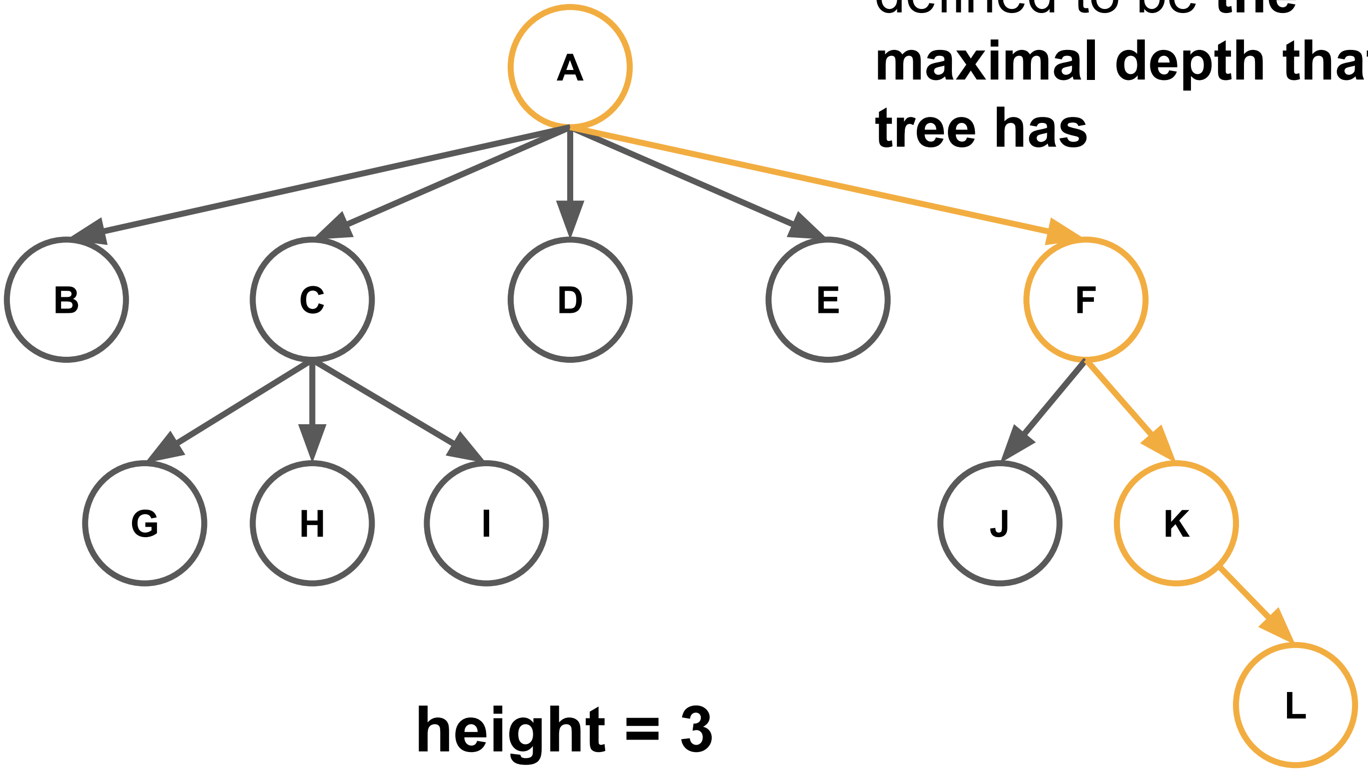
# Tree Terminology

The **height** of a tree is defined to be **the maximal depth that a tree has**



# Tree Terminology

The height of a tree is defined to be **the maximal depth that a tree has**




**height = 3**

# Tree Terminology Summary

- Every non-empty tree has a **root node** that defines the "top" of the tree.
- Every node has 0 or more **children** nodes descended from it. Nodes with no children are called **leaf nodes**.
- Every node in a tree has exactly one **parent** node (except for the root node).
- A **path** through the tree traverses edges between parents and their children.
- The **depth** of a node is the number of **edges** between the root and that node. A tree's **height** is the number of **edges** in the longest path through the tree.

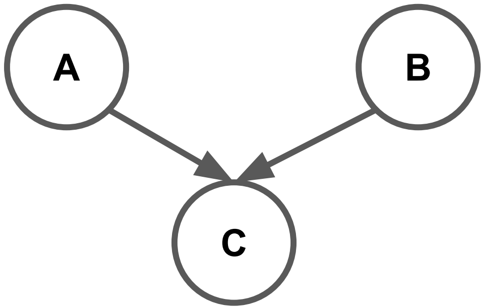
# Tree Properties

# Tree Properties

- Any node in a tree can only have one parent.

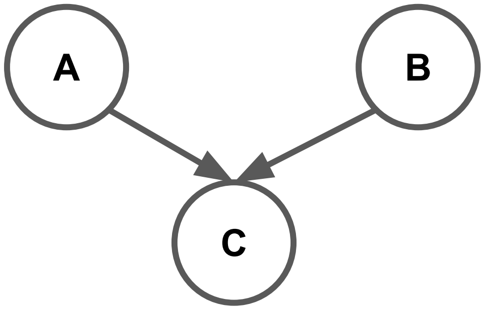
# Tree Properties

- Any node in a tree can only have one parent.



# Tree Properties

- Any node in a tree can only have one parent.




Not a
tree!

# Tree Properties

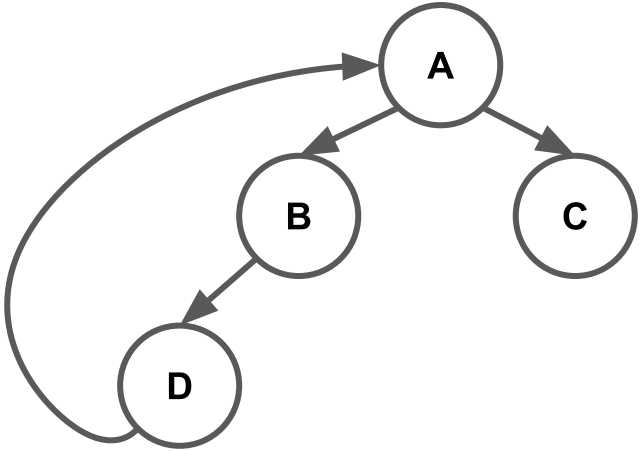
- Any node in a tree can only have one parent.

- The tree cannot have any cycles. That is, there should be no way to make a complete loop through the tree.

# Tree Properties

- Any node in a tree can only have one parent.

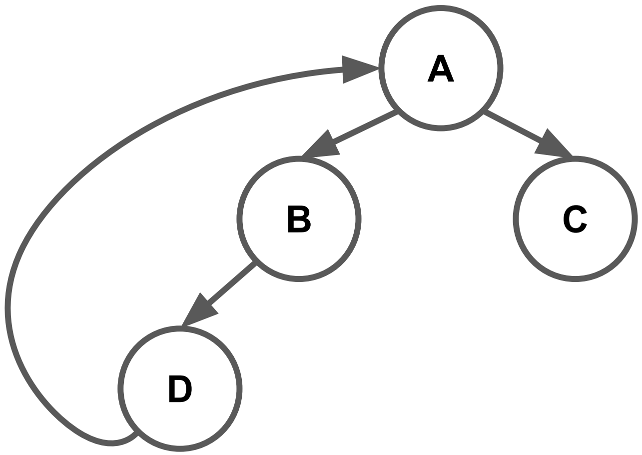
- The tree cannot have any cycles. That is, there should be no way to make a complete loop through the tree.



# Tree Properties

- Any node in a tree can only have one parent.

- The tree cannot have any cycles. That is, there should be no way to make a complete loop through the tree.




Not a tree!

# Announcements

# Announcements

- Assignment 5 is due on **Tuesday, August 3 at 11:59pm PDT**.
    - On the short answer, problem 7 asks about 2 sort prototypes. You should pretend that question only says one!
- Trip's group OH will be moved From Wednesday, 8/4 to Thursday 8/5, still from 10am-12pm PT
    - Trip still has office hours on 8/3 from 9-11am PT. Come thru!
- Assignment 6 will be released on Wednesday and will be be due on **Wednesday, August 11 at 11:59pm PDT**. This is a hard deadline – there is **no grace period and no submissions will be accepted after this time**.

- The End-quarter Assessment will take place over 3 days from **Friday, August 13 to Sunday, August 15.** More information will be released soon.

# Trees in C++

# Binary Trees

- In general, we've seen that nodes in a tree can have variable numbers of children (subtrees) and sometimes very, very many.

# Binary Trees

- In general, we've seen that nodes in a tree can have variable numbers of children (subtrees) and sometimes very, very many.

- However, when working with trees in computer programs, it is common to work mostly with **binary trees**.

# Binary Trees

- In general, we've seen that nodes in a tree can have variable numbers of children (subtrees) and sometimes very, very many.

- However, when working with trees in computer programs, it is common to work mostly with **binary trees**.

- A **binary tree** is a tree where every node has either 0, 1, or 2 children. No node in a binary tree can have more than 2 children.

# Binary Trees

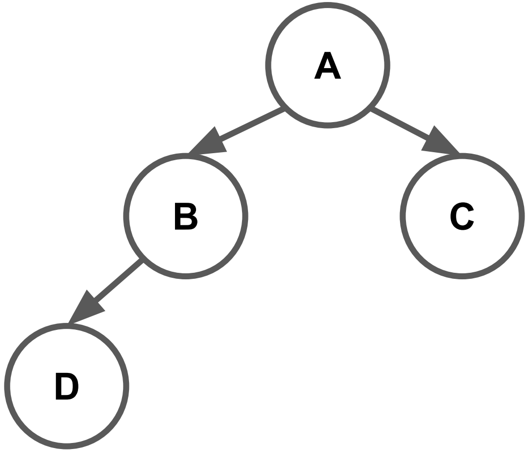
- In general, we've seen that nodes in a tree can have variable numbers of children (subtrees) and sometimes very, very many.

- However, when working with trees in computer programs, it is common to work mostly with **binary trees**.

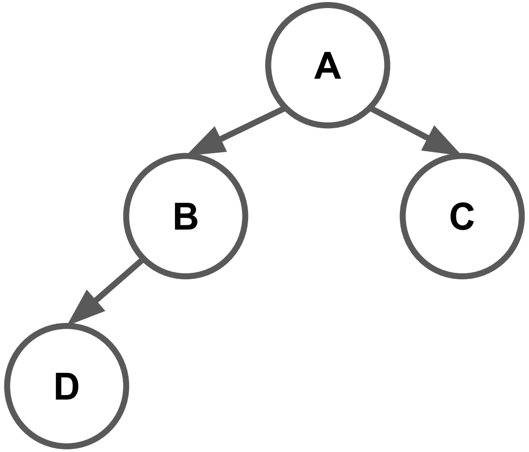
- A **binary tree** is a tree where every node has either 0, 1, or 2 children. No node in a binary tree can have more than 2 children.

- Typically, the two children of a node in a binary tree are referred to as the **left child** and the **right child**.

# Binary Trees

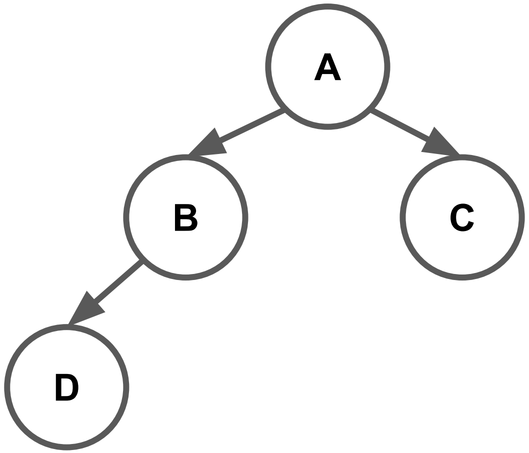


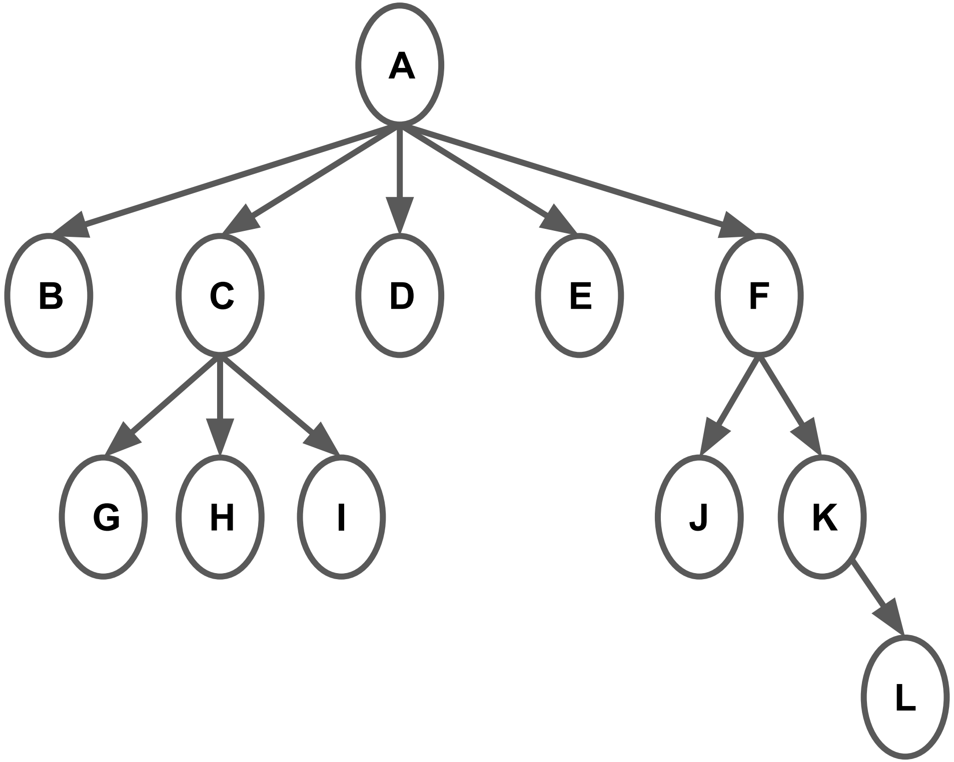
# Binary Trees




Binary
Tree!

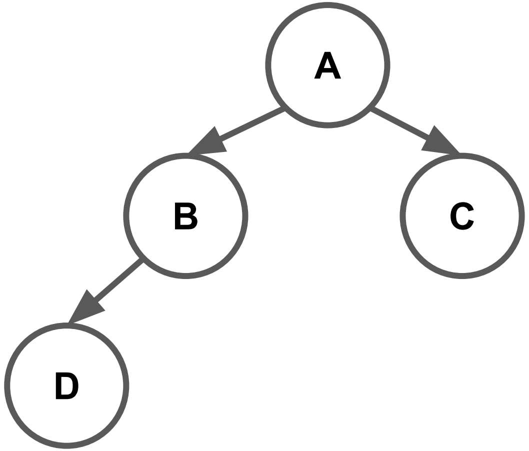
# Binary Trees



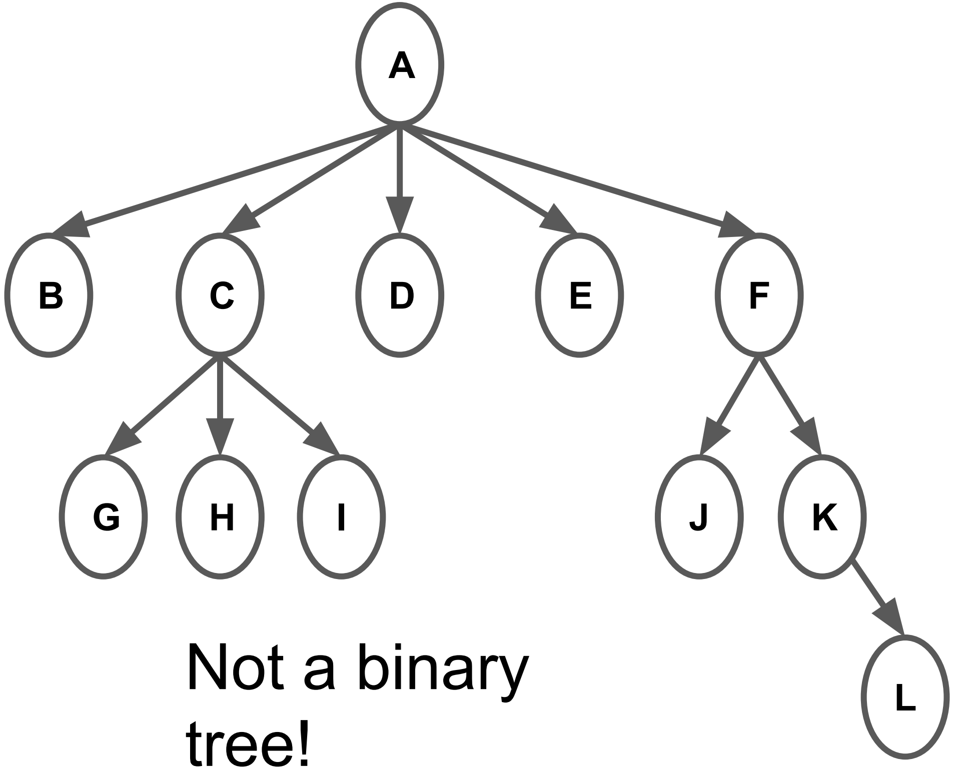




Binary
Tree!

# Binary Trees




Binary
Tree!




Not a binary
tree!

# Building Trees Programmatically

- To build a tree in C++, we need a new version of the Node struct we've seen before.

# Building Trees Programmatically

- To build a tree in C++, we need a new version of the Node struct we've seen before.

- In this case, we want each Node to have a data value (like a linked list), but now we want two pointers, one to the left child, and one to the right child.

# Building Trees Programmatically

- To build a tree in C++, we need a new version of the Node struct we've seen before.

- In this case, we want each Node to have a data value (like a linked list), but now we want two pointers, one to the left child, and one to the right child.

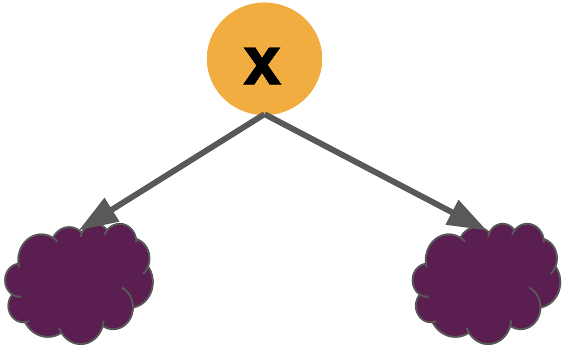
```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
}
```

# What is a tree in C++?

**A tree is either...**

An empty data structure, or...

A single node (parent), with zero or more non-empty subtrees (children)

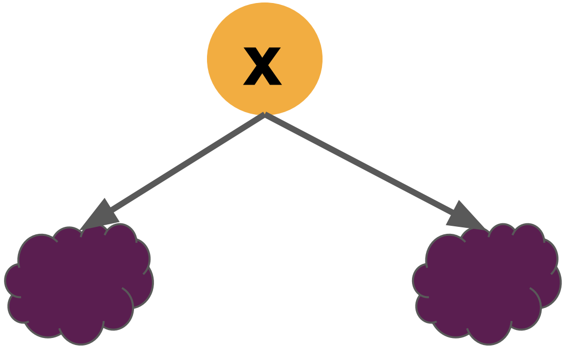


# What is a tree in C++?

**A tree is either...**

An empty tree represented by **nullptr**, or...


A single node (parent), with zero or more non-empty subtrees (children)

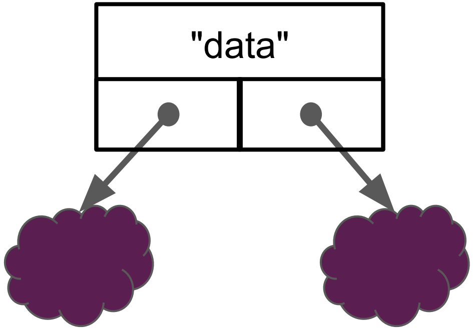


# What is a tree in C++?

**A tree is either...**

An empty tree represented by **nullptr**, or...


A single **TreeNode,** with 0, 1, or 2 non-null pointers to other **TreeNodes**



# Building Trees Programmatically

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
}
```

# Building Trees Programmatically

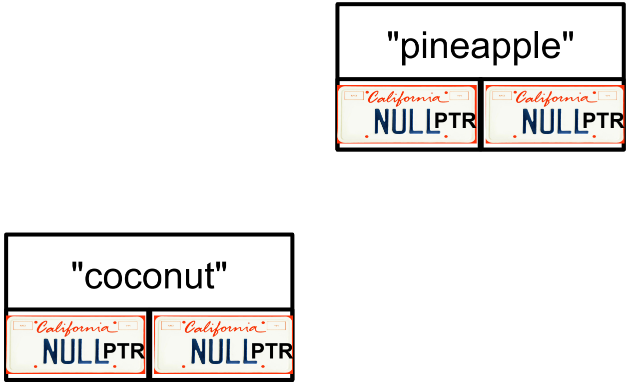
```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
}
```

# Building Trees Programmatically

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
}
```

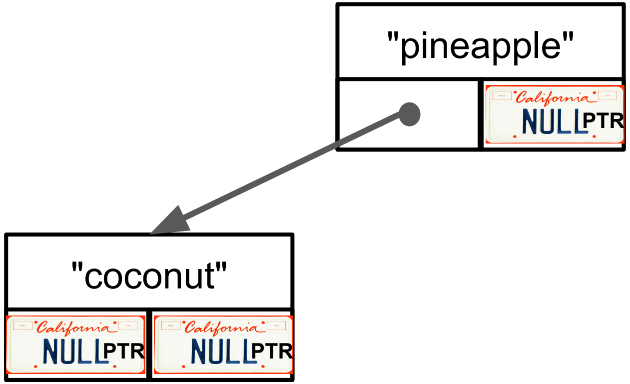
# Building Trees Programmatically

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
}
```



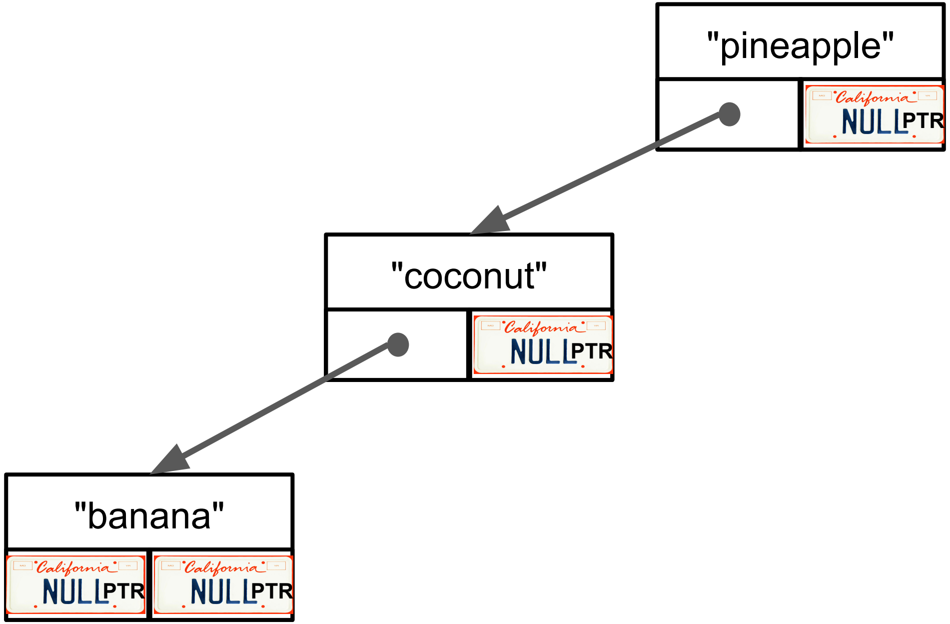
# Building Trees Programmatically

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
}
```



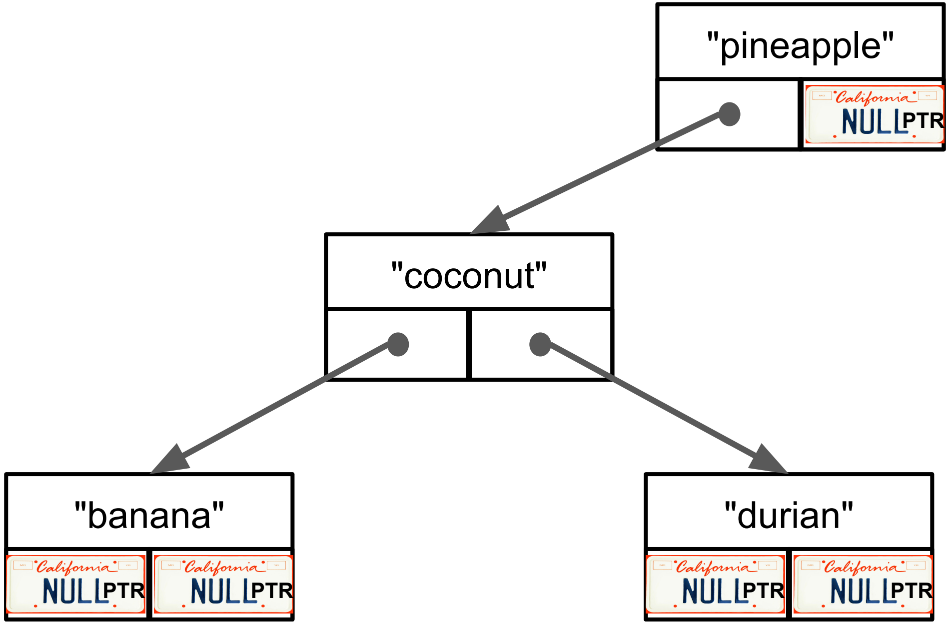
# Building Trees Programmatically

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
}
```



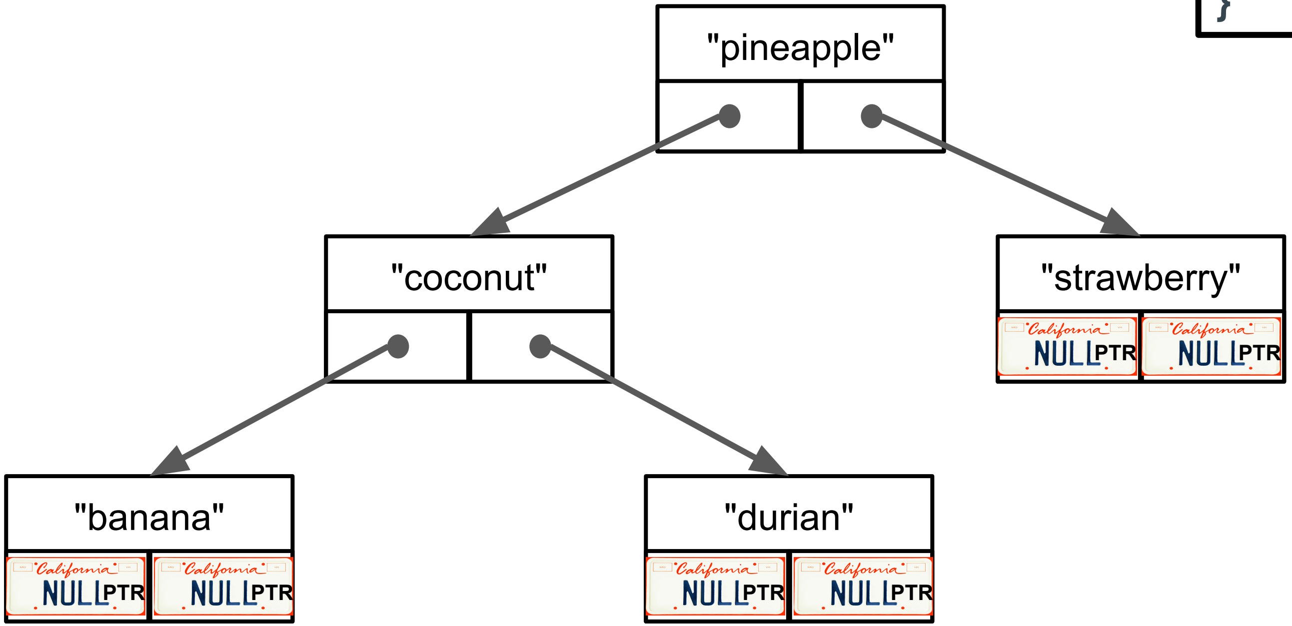
# Building Trees Programmatically

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
}
```



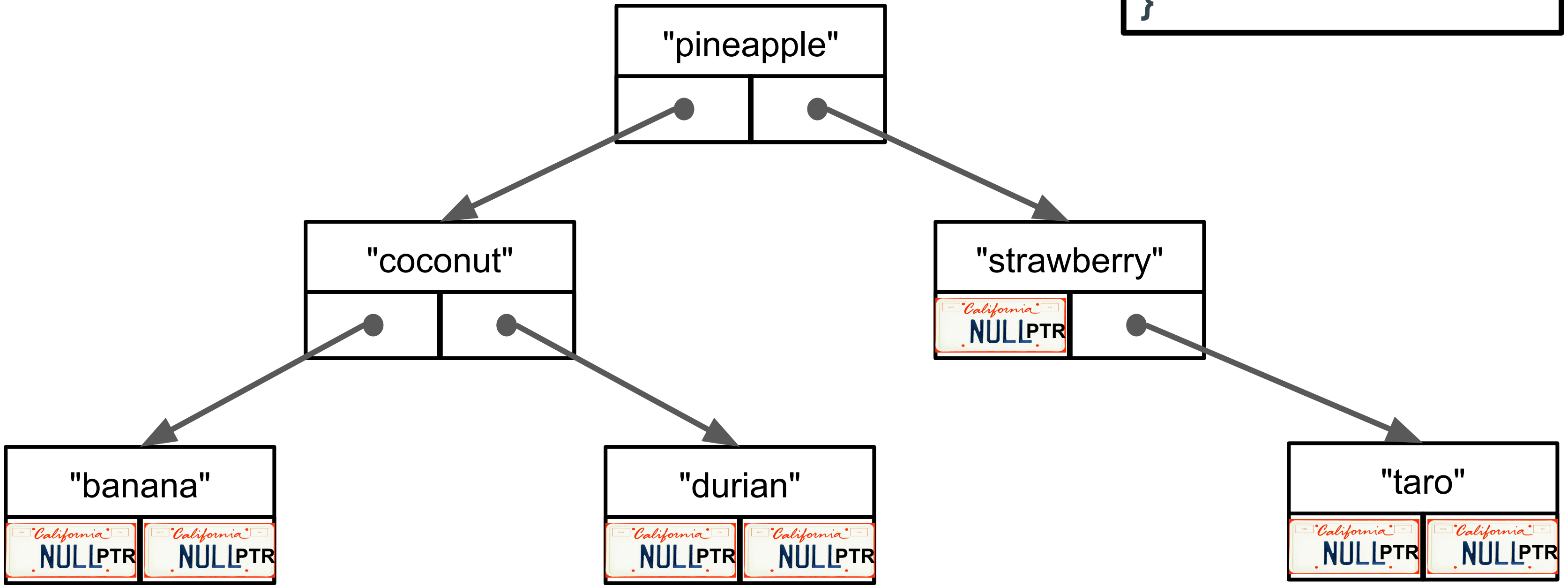
# Building Trees Programmatically

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
}
```



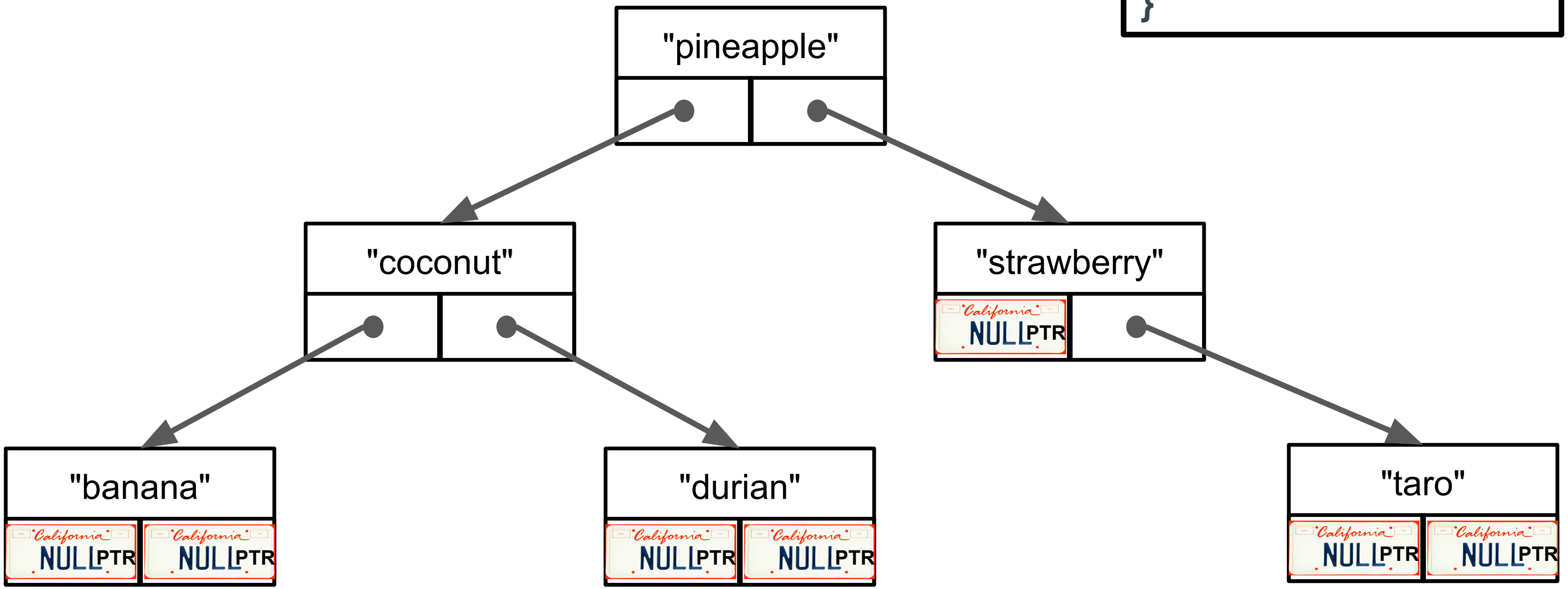
# Building Trees Programmatically

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
}
```



# Building Trees Programmatically

```
struct TreeNode {
    string data;
    TreeNode* left;
    TreeNode* right;
}
```




Note:  Trees do not have to be complete, like heaps. **Any** node can have 0, 1, or 2 children.

# Let's code it!

**buildExampleTree()**

# Building a Tree Takeaways

- Building a tree is very similar to the process of building a linked list.

- We create new nodes of the tree by dynamically allocating memory.

- We integrate these new nodes into the tree by rewiring the **left** and **right** pointers of existing nodes in the tree.

# Tree Traversals

# Tree Traversals

- Often, we will want to "do something" with each node in a tree. Like linked lists, we can do so by **traversing the tree**. With the branching involved, this is a slightly more involved process than traversing a linked list!

# Tree Traversals

- Often, we will want to "do something" with each node in a tree. Like linked lists, we can do so by **traversing the tree**. With the branching involved, this is a slightly more involved process than traversing a linked list!

- There are three main ways to traverse a binary tree:
  - Pre-order traversal
  - In-order traversal
  - Post-order traversal

# Tree Traversals

- Often, we will want to "do something" with each node in a tree. Like linked lists, we can do so by **traversing the tree**. With the branching involved, this is a slightly more involved process than traversing a linked list!

- There are three main ways to traverse a binary tree:
    - Pre-order traversal
    - In-order traversal
    - Post-order traversal

- Due to the recursive nature of trees, all of these algorithms are most easily defined **recursively**.

# Pre-order Traversal

- The algorithm for a pre-order traversal is defined as follows:
  - "Do something" with the current node
  - Traverse the left subtree
  - Traverse the right subtree

- For example purposes, let's have our "do something" to be printing the contents of the current node, which will allow us to print the overall tree.

# Let's code it!

**preorderPrintTree()**

# Pre-order Traversal

- The algorithm for a pre-order traversal is defined as follows:
    - "Do something" with the current node
    - Traverse the left subtree
    - Traverse the right subtree

- For example purposes, let's have our "do something" be printing the contents of the current node, which will allow us to print the overall tree.

- Output: **pineapple coconut banana durian strawberry taro**

# In-order Traversal

- The algorithm for an in-order traversal is defined as follows:
  - Traverse the left subtree
  - "Do something" with the current node
  - Traverse the right subtree

# Let's code it!

**inorderPrintTree()**

# In-order Traversal

- The algorithm for an in-order traversal is defined as follows:
  - Traverse the left subtree
  - "Do something" with the current node
  - Traverse the right subtree

- Output: **banana coconut durian pineapple strawberry taro**

- Observation: The output of this traversal gives as all the values in alphabetical order. Is this a coincidence?
  - No! We'll see why tomorrow! (for now, just note that this phenomena is **not** guaranteed for all binary trees.)

# Post-order Traversal

- The algorithm for a post-order traversal is defined as follows:
  - Traverse the left subtree
  - Traverse the right subtree
  - "Do something" with the current node

# Try it yourself!

**postorderPrintTree()**

# Post-order Traversal

- The algorithm for a post-order traversal is defined as follows:
  - Traverse the left subtree
  - Traverse the right subtree
  - "Do something" with the current node

- Output: **banana durian coconut taro strawberry pineapple**

- Application: Freeing trees! (we'll see this in lecture tomorrow)

# Summary

# Trees Summary

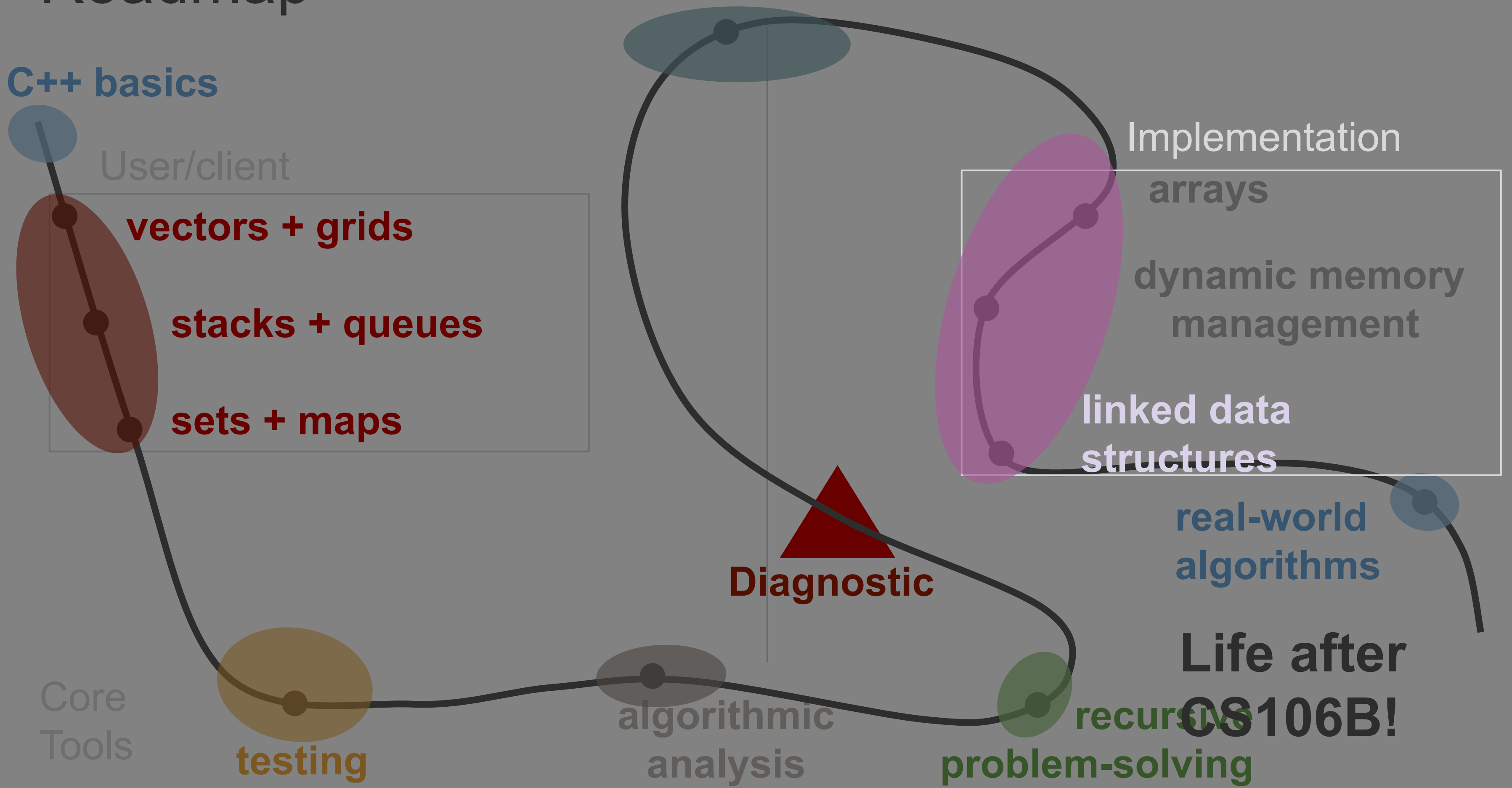
- Trees allow us to organize information in a linked data structure such that the distance to any element is short, even if there are many elements.

- Trees organize nodes in a hierarchical manner, where each element contains connections to children nodes that exist "lower" in the tree.

- There are three main ways to traverse the nodes in a tree, and each type of traversal visits the nodes of the tree in a distinctly different order.

# What’s next?

# Roadmap


Object-Oriented Programming
C++ basics
User/client
vectors + grids
stacks + queues
sets + maps
Implementation
arrays
dynamic memory management
linked data structures
real-world algorithms
Diagnostic
Life after CS106B!
Core Tools
testing
algorithmic analysis
recursive problem-solving

# Binary Search Trees