

# Binary Trees

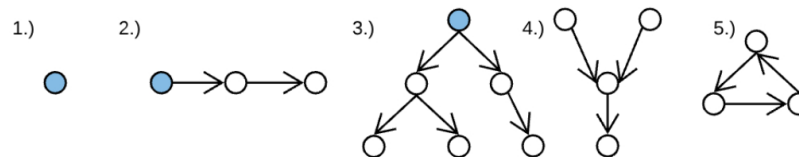
🕒 20 minutes

## Binary Trees

Binary Trees are perhaps the most pervasive data structure in computer science. Let's take a moment to go over the basic characteristics of a Binary Tree before we explore algorithms that utilize this structure.

### What is a Graph?

Before we define what a **Tree** is, we must first understand the definition of a **Graph**. A graph is a collection of **nodes** and any **edges** between those nodes. You've likely seen depictions of graphs before, they usually exist as circles (nodes) and arrows (edges) between those circles. Below are few examples of graphs:



For now, you can ignore the blue coloring. Notice how the graphs above vary greatly in their structure. A graph is indeed a very broad, overarching category. In fact, linked lists and trees are both considered subclasses of graphs. We'll cover algorithms that operate on a general graph structure later, but for now we want to focus on what graphs are trees and what graphs are not. It's worth mentioning that a single node with no edges (image 1) is considered a graph. The empty graph (a graph with 0 nodes and 0 edges, not pictured :) is also still a graph. This line of thinking will help us later when we design graph algorithms.

### What is a Tree?

A **Tree** is a **Graph** that does not contain any cycles. A cycle is defined as a path through edges that begins and ends at the same node. This seems straightforward, but the definition becomes a bit muddled as Mathematicians and Computer Scientists use the term "tree" in slightly different ways. Let's break it down:

- To a Mathematician, graphs 1, 2, 3, and 4 in the above image are trees.
- To a Computer Scientist, only graphs 1, 2, and 3 are trees.

Well, at least both camps agree that graph 5 is most certainly not a tree! This is because of the obvious cycle that spans all three nodes. However, why is there

disagreement over graph 4? The reason is this: In computer science, we use to the term "tree" to really refer to a "rooted tree." A "rooted tree" is a "tree" where there exists a special node from which every other node is accessible; we call this special node the "root". Think of the root as ultimate ancestor, the single node that all other nodes inherit from. Above we have colored all roots in blue. Like you'd probably suspect, in this course we'll subscribe to the Computer Scientist's interpretation. That is, we won't consider graph 4 a tree because there is no such root we can label.

You've probably heard the term "root" throughout your software engineering career: root directory, root user, etc.. All of these concepts branch<sup>†</sup> from the humble tree data structure!

## What is a Binary Tree?

A **Binary Tree** is a **Tree** where nodes have **at most 2 children**. This means graphs 1, 2, and 3 are all Binary Trees. There exist ternary trees (at most 3 children) and n-ary trees (at most n children), but you'll likely encounter binary trees in your job hunt, so we'll focus on them in this course. Based on our final definition for a binary tree, here is some food for thought:

- an empty graph of 0 nodes and 0 edges is a binary tree
- a graph of 1 node and 0 edges is a binary tree
- a linked list is a binary tree

Take a moment to use the definitions we explored to verify that each of the three statements above is true. We bring up these three scenarios in particular because they are the simplest types of Binary Trees. We want to eventually build elegant algorithms and these simple scenarios will fuel our design.

## Representing a Binary Tree with Node Instances

Let's explore a common way to represent binary trees using some object oriented design. A tree is a collection of nodes, so let's implement a `TreeNode` class. We'll use properties of `left` and `right` to reference the children of a `TreeNode`. That is, `left` and `right` will reference other `TreeNode` s:

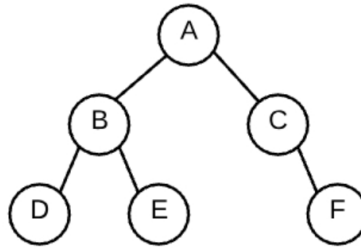
```
class TreeNode {
  constructor(val) {
    this.val = val;
    this.left = null;
    this.right = null;
  }
}
```

Constructing a tree is a matter of creating the nodes and setting `left` and `right` however we please. For example:

```
let a = new TreeNode('a');
let b = new TreeNode('b');
let c = new TreeNode('c');
let d = new TreeNode('d');
let e = new TreeNode('e');
let f = new TreeNode('f');
```

```
a.left = b;  
a.right = c;  
b.left = d;  
b.right = e;  
c.right = f;
```

The visual representation of the tree is:



To simplify our diagrams, we'll omit the arrowheads on the edges. Moving forward you can assume that the top node is the root and the direction of edges points downward. In other words, node A is the Root. Node A can access node B through `a.left`, but Node B cannot access Node A.

We now have a data structure we can use to explore Binary Tree algorithms! Creating a tree in this way may be tedious and repetitive, however it allows us to decide exactly what nodes are connected and in what direction. This is will be useful as we account for edge cases in our design.

## Basic Tree Terminology

- tree - graph with no cycles
- binary tree - tree where nodes have at most 2 nodes
- root - the ultimate parent, the single node of a tree that can access every other node through edges; by definition the root will not have a parent
- internal node - a node that has children
- leaf - a node that does not have any children
- path - a series of nodes that can be traveled through edges - for example A, B, E is a path through the above tree

† Pun Intended


## Traversing trees

Unlike linked lists, arrays and other linear data structures, which are usually traversed in linear order from front to back or back to front, trees may be traversed in multiple ways. They may be traversed in depth-first or breadth-first order. There are three common ways to traverse them in depth-first order: in-order, pre-order and post-order.

### Breadth-first search

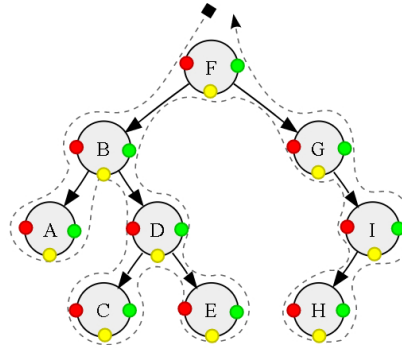
Trees can also be traversed level-by-level, where you visit every node on a level before going to a lower level. This search is referred to as breadth-first search

(BFS), as the search tree is broadened as much as possible on each depth before going to the next depth.

 breadth-first tree traversal

## Depth-first searches

These searches are referred to as depth-first search (DFS), since the search tree is deepened as much as possible on each child before going to the next sibling.



### Pre-order traversal

In the above image, the pre-order is noted by when the dotted line touches the red dot and yields F, B, A, D, C, E, G, I, H. The algorithm is as follows and is "pre-order" because you access the value of the node before recursively descending.

- Access the data of the current node
- Recursively visit the left sub tree
- Recursively visit the right sub tree

### In-order traversal

In the above image, the in-order is noted by when the dotted line touches the yellow dot and yields A, B, C, D, E, F, G, H, I. The algorithm is as follows and is "in-order" because you access the value of the node after descending to the left but before descending to the right.

- Recursively visit the left sub tree
- Access the data of the current node
- Recursively visit the right sub tree

### Post-order traversal

In the above image, the post-order is noted by when the dotted line touches the green dot and yields A, C, E, D, B, H, I, G, F. The algorithm is as follows and is "post-order" because you access the value of the node after descending to both branches.

- Recursively visit the left sub tree
- Recursively visit the right sub tree

- Access the data of the current node

Did you find this lesson helpful?



✓ Mark As Complete

Finished with this task? Click **Mark as Complete** to continue to the next page!