A **binary tree** is a **tree** where every node has two or fewer children. The children are usually called left and right.

public class BinaryTreeNode {

public int value;

public BinaryTreeNode left;

public BinaryTreeNode right;

public BinaryTreeNode(int value) {

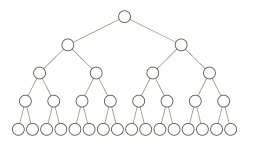
this.value = value;

}

}

****

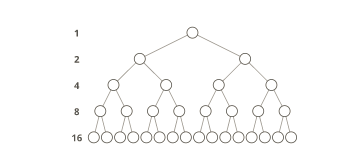
This lets us build a structure like this:



That particular example is special because every level of the tree is completely full. There are no "gaps." We call this kind of tree "**perfect**."

Binary trees have a few interesting properties when they're perfect:

**Property 1: the number of total nodes on each "level" doubles as we move down the tree.**



**Property 2: the number of nodes on the last level is equal to the sum of the number of nodes on all other levels (plus 1).**In other words, about *half* of our nodes are on the last level.

Let's call the number of nodes **n*n***, and the height of the tree **h*h***. **h*h*** can also be thought of as the "number of levels."

If we had **h*h***, how could we calculate **n*n***?

Let's just add up the number of nodes on each level! How many nodes are on each level?

If we zero-index the levels, the number of nodes on the **x*x***th level is exactly **2^x2​*x*​​**.

1. Level **00**: **2^02​0​​** nodes,
2. Level **11**: **2^12​1​​** nodes,
3. Level **22**: **2^22​2​​** nodes,
4. Level **33**: **2^32​3​​** nodes,
5. *etc*

So our total number of nodes is:

**n = 2^0 + 2^1 + 2^2 + 2^3 + ... + 2^{h-1}*n*=2​0​​+2​1​​+2​2​​+2​3​​+...+2​*h*−1​​**

Why only up to **2^{h-1}2​*h*−1​​**? Notice that we started counting our levels at 0. So if we have **h*h*** levels in total, the last level is actually the "**h-1*h*−1**"-th level. That means the number of nodes on the last level is **2^{h-1}2​*h*−1​​**.

But we can simplify. Property 2 tells us that the number of nodes on the last level is (1 more than) half of the total number of nodes, so we can just take the number of nodes on the last level, multiply it by 2, and subtract 1 to get the number of nodes overall. We know the number of nodes on the last level is **2^{h-1}2​*h*−1​​**, So:

**n = 2^{h-1} \* 2 - 1*n*=2​*h*−1​​∗2−1n = 2^{h-1} \* 2^1 - 1*n*=2​*h*−1​​∗2​1​​−1n = 2^{h-1+1}- 1*n*=2​*h*−1+1​​−1n = 2^{h} - 1*n*=2​*h*​​−1**

So that's how we can go from **h*h*** to **n*n***. What about the other direction?

We need to bring the **h*h*** down from the exponent. That's what logs are for!

First, some quick review. **\log\_{10} (100)log​10​​(100)** simply means, **"What power must you raise 10 to in order to get 100?"**. Which is 2, because **10^2 = 10010​2​​=100**.

We can use logs in algebra to bring variables down from exponents by exploiting the fact that we can simplify **\log\_{10}(10^2)log​10​​(10​2​​)**. What power must we raise **1010** to in order to get **10^210​2​​**? That's easy—it's **22**.

So in this case we can take the **\log\_{2}log​2​​** of both sides:

**n = 2^{h} - 1*n*=2​*h*​​−1n + 1 = 2^{h}*n*+1=2​*h*​​\log\_{2}{((n+1))} = \log\_{2}{(2^{h})}log​2​​((*n*+1))=log​2​​(2​*h*​​)\log\_{2}{(n+1)} = hlog​2​​(*n*+1)=*h***

So that's the relationship between height and total nodes in a perfect binary tree.

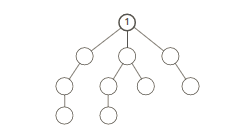
A **leaf node** is a tree node with no children.

It's the "end" of a path to the bottom, from the root.

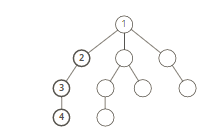
**Depth-first search** (DFS) is a method for exploring a tree or graph. In a DFS, you go as deep as possible down one path before backing up and trying a different one.

Depth-first search is like walking through a corn maze. You explore one path, hit a dead end, and go back and try a different one.

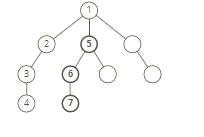
Here's a how a DFS would traverse this tree, starting with the root:



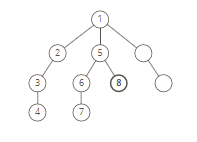
We'd go down the first path we find until we hit a dead end:



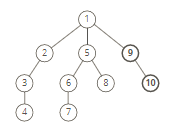
Then we'd do the same thing again—go down a path until we hit a dead end:



And again:



And again:



Until we reach the end.

Depth-first search is often compared with **breadth-first search**.

Advantages:

* Depth-first search on a binary tree *generally* requires less memory than breadth-first.
* Depth-first search can be easily implemented with recursion.

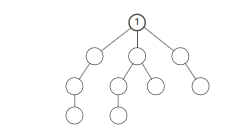
Disadvantages

* A DFS doesn't necessarily find the shortest path to a node, while breadth-first search does.

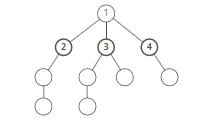
**Breadth-first search** (BFS) is a method for exploring a tree or graph. In a BFS, you first explore all the nodes one step away, then all the nodes two steps away, etc.

Breadth-first search is like throwing a stone in the center of a pond. The nodes you explore "ripple out" from the starting point.

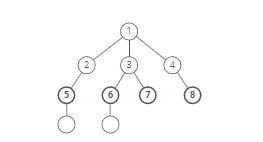
Here's a how a BFS would traverse this tree, starting with the root:



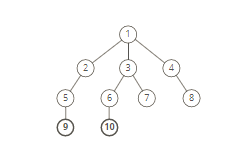
We'd visit all the immediate children (all the nodes that're one step away from our starting node):



Then we'd move on to all *those* nodes' children (all the nodes that're *two steps* away from our starting node):



And so on:



Until we reach the end.

Breadth-first search is often compared with **depth-first search**.

Advantages:

* A BFS will find the **shortest path** between the starting point and any other reachable node. A depth-first search will not necessarily find the shortest path.

Disadvantages

* A BFS on a binary tree *generally* requires more memory than a DFS.