Binary Trees

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

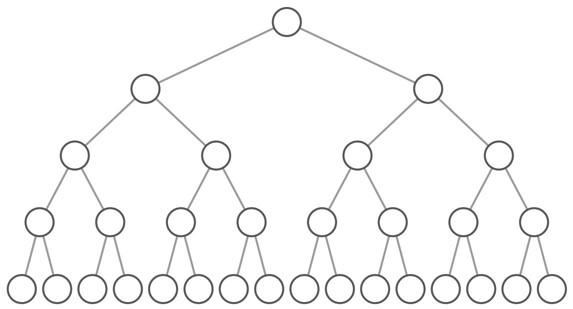
Java ▾

```java
public class BinaryTreeNode {

    public int value;
    public BinaryTreeNode left;
    public BinaryTreeNode right;

    public BinaryTreeNode(int value) {
        this.value = value;
    }
}
```

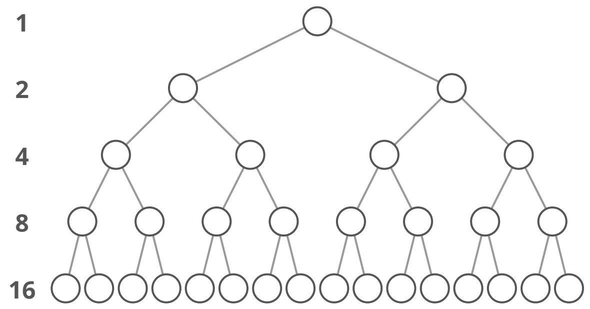
This lets us build a structure like this:



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, and the height of the tree h. h can also be thought of as the "number of levels."

If we had h, how could we calculate 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 xth level is exactly 2^x!

1. Level 0: 2^0 nodes,
2. Level 1: 2^1 nodes,
3. Level 2: 2^2 nodes,
4. Level 3: 2^3 nodes,
5. *etc*

So our total number of nodes is:

$$n = 2^0 + 2^1 + 2^2 + 2^3 + \ldots + 2^{h-1}$$

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

$$n = 2^{h-1} * 2 - 1$$

$$n = 2^{h-1} * 2 - 1$$

$$n = 2^{h-1} * 2^1 - 1$$

$$n = 2^{h-1+1} - 1$$

$$n = 2^h - 1$$

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

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

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

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

So in this case we can take the $\log_2$ of both sides:

$$n = 2^h - 1$$

$$n + 1 = 2^h$$

$$\log_2\left((n+1)\right) = \log_2\left(2^h\right)$$

$$\log_2\left(n+1\right) = h$$

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

See also:

- Binary Search Algorithm (/concept/binary-search)

Binary Tree Coding Interview Questions

8 **Balanced Binary Tree »**
Write a function to see if a binary tree is 'superbalanced'--a new tree property we just made up. keep reading »
(/question/balanced-binary-tree)

9 **Binary Search Tree Checker »**
Write a function to check that a binary tree is a valid binary search tree. keep reading »
(/question/bst-checker)

10 **2nd Largest Item in a Binary Search Tree »**
Find the second largest element in a binary search tree. keep reading »
(/question/second-largest-item-in-bst)

All Questions ➜ (/all-questions)