# Binary Trees

A Binary Tree is a type of tree data structure where each node can have a maximum of two child nodes, a left child node and a right child node.

This restriction, that a node can have a maximum of two child nodes, gives us many benefits:

- Algorithms like traversing, searching, insertion, and deletion become easier to understand, implement, and run faster.
- Keeping data sorted in a Binary Search Tree (BST) makes searching very efficient.
- Balancing trees is easier to do with a limited number of child nodes, using an AVL Binary Tree for example.
- Binary Trees can be represented as arrays, making the tree more memory efficient.

## Types of Binary Trees

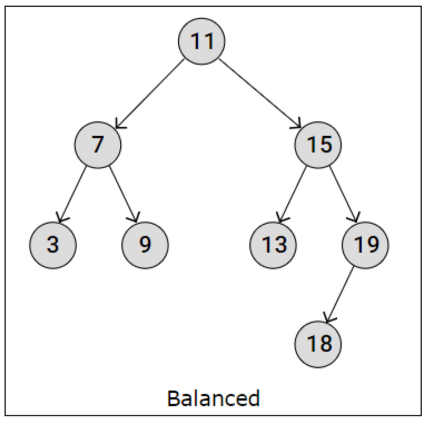
There are different variants, or types, of Binary Trees worth discussing to get a better understanding of how Binary Trees can be structured.

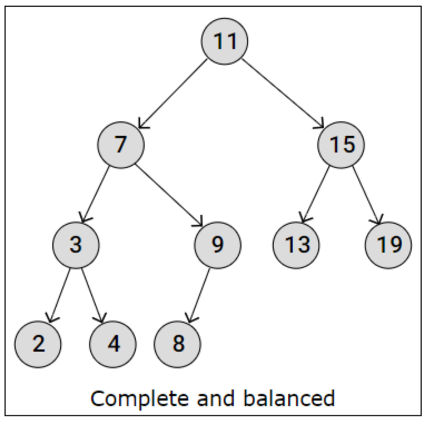
The different kinds of Binary Trees are also worth mentioning now as these words and concepts will be used later in the tutorial.

Below are short explanations of different types of Binary Tree structures, and below the explanations are drawings of these kinds of structures to make it as easy to understand as possible.

A **balanced** Binary Tree has at most 1 in difference between its left and right subtree heights, for each node in the tree.

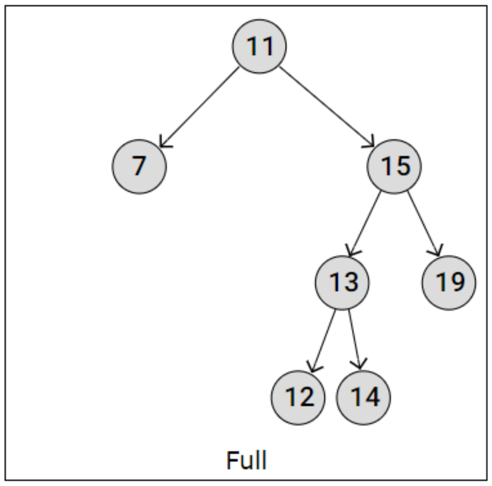
A **complete** Binary Tree has all levels full of nodes, except the last level, which is can also be full, or filled from left to right. The properties of a complete Binary Tree means it is also balanced.

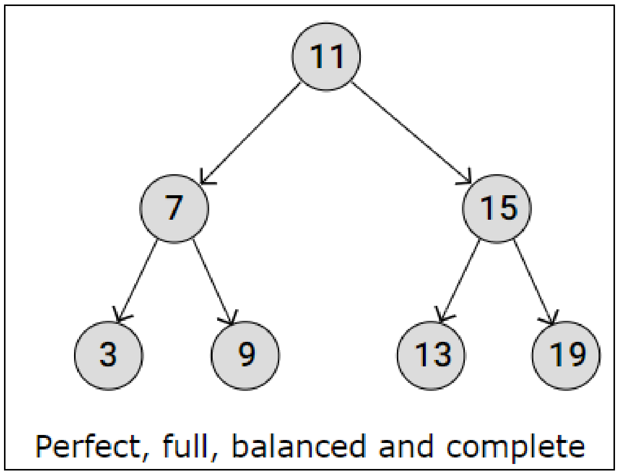




A **full** Binary Tree is a kind of tree where each node has either 0 or 2 child nodes.

A **perfect** Binary Tree has all leaf nodes on the same level, which means that all levels are full of nodes, and all internal nodes have two child nodes.The properties of a perfect Binary Tree means it is also full, balanced, and complete.





## Binary Trees vs Arrays and Linked Lists

Benefits of Binary Trees over Arrays and Linked Lists:

- Arrays are fast when you want to access an element directly, like element number 700 in an array of 1000 elements for example. But inserting and deleting elements require other elements to shift in memory to make place for the new element, or to take the deleted elements place, and that is time-consuming.
- Linked Lists are fast when inserting or deleting nodes, no memory shifting is needed, but to access an element inside the list, the list must be traversed, and that takes time.
- Binary Trees, such as Binary Search Trees and AVL Trees, are great compared to Arrays and Linked Lists because they are BOTH fast at accessing a node, AND fast when it comes to deleting or inserting a node, with no shifts in memory needed.

We will take a closer look at how Binary Search Trees (BSTs) and AVL Trees work on the next two pages, but first let's look at how a Binary Tree can be implemented, and how it can be traversed.

## Binary Tree Traversal

Going through a Tree by visiting every node, one node at a time is called traversal.

Since Arrays and Linked Lists are linear data structures, there is only one obvious way to traverse these: start at the first element, or node, and continue to visit the next until you have visited them all.

But since a Tree can branch out in different directions (non-linear), there are different ways of traversing Trees.

There are two main categories of Tree traversal methods:

Breadth First Search (BFS) is when the nodes on the same level are visited before going to the next level in the tree. This means that the tree is explored in a more sideways direction.

Depth First Search (DFS) is when the traversal moves down the tree all the way to the leaf nodes, exploring the tree branch by branch in a downwards direction.

There are three different types of DFS traversals:

- pre-order
- in-order
- post-order