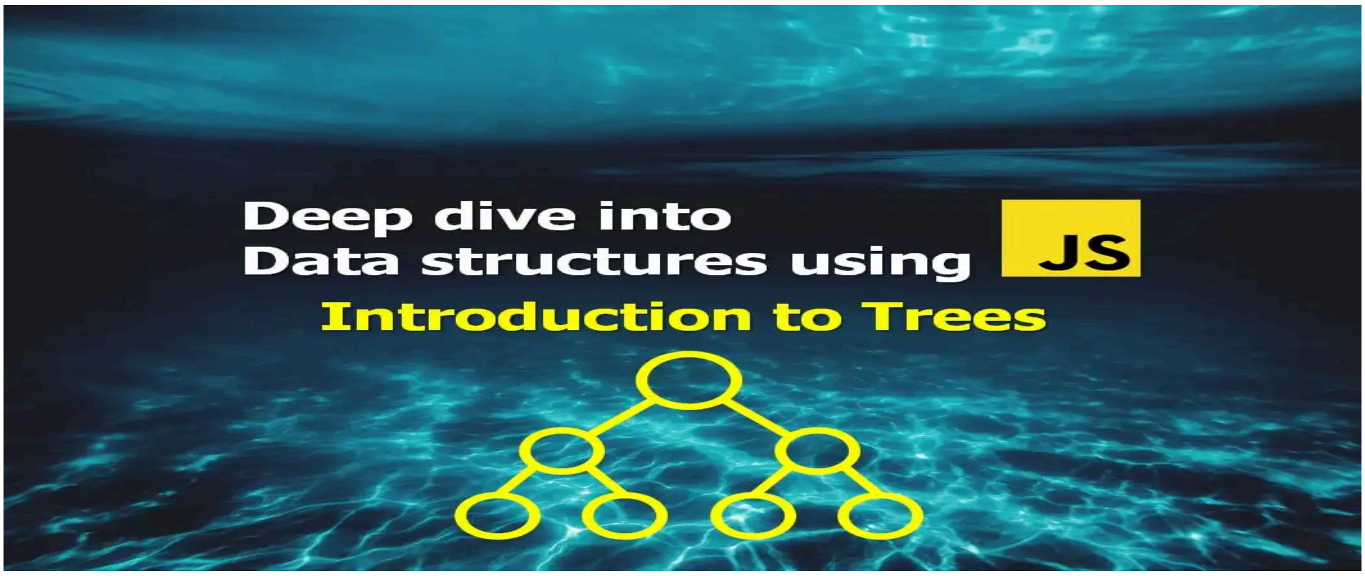
Deep Dive into Data structures using Javascript - Introduction to Trees



What is a Tree data structure?

A Tree is a non-linear, advanced and a powerful data structure which is widely used for storing hierarchically relational data. This ability comes out of the box along with moderate performance on insertions, deletions and searching that are much faster compared to linear data structures like Arrays or Linked Lists if the Tree is balanced.

Tree is an important concept, because it opens the door to understanding advanced data structures. Having a solid understanding of Tree data structure is crucial to be able to understand and work with different variants of Trees, as well as Graphs.

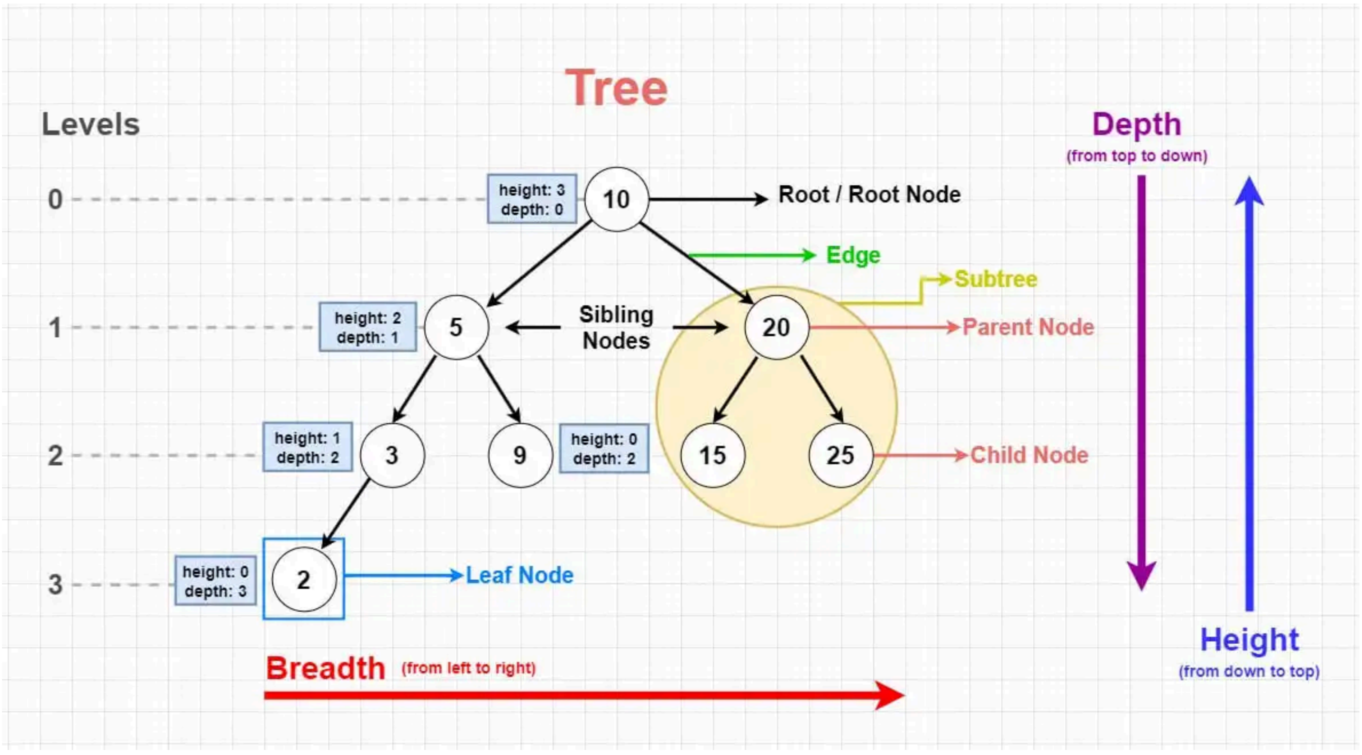
Trees are widely used to solve problems for countless real life applications. Some examples are:

- Computer file systems
- HTML (uses DOM Tree) / XML
- Game graphics, visual effects
- 3D video games (Space partitioning)
- Indexing databases (B+ tree)
- Machine learning
- Organization charts
- Domain Name Server (DNS)

Trees in Javascript

There is no built-in Tree data structure variant in Javascript, but it is possible to build custom versions. We won't be building any Tree variant in this article. Since Trees have a rich terminology and set of rules, the focus will be getting familiar with the concept as a first step rather than diving into implementation details.

Anatomy of a Tree data structure



Trees have a complex and advanced anatomy compared to linear data structures. Before diving deeper into types and inner workings of Trees, let's get familiar with the main terminologies:

Tree terminologies:

- **Node:** It is the atomic building block of a Tree, very similar to a Linked List node. A Tree Node consists of following properties: value, pointers to connect to other nodes. A node can have one or more pointers to connect to other nodes.
- **Edge:** Edge, which is also sometimes called "branch" is the link connects any Nodes with eachother inside the tree.
- **Parent node:** Parent, sometimes also called "predecessor" is the node that has direct connection / edge to any other node that is one level below. A parent node can have any number of child nodes.
- **Child node:** A node that is directly connected to any particular node one level above on it's level is called child of that node. Except the Root node, every node in the Tree is a Child Node.
- **Root node:** The first node of the tree which all others nodes are derived from. A Tree can only have a single root node.
- **Subtree:** A tree that is formed by the particular node and all it's descendants inside the tree, which can be also treated as an independent tree.
- **Sibling:** Nodes that shares the same parent node.
- **Cousins:** Nodes that shares the same level with different parent nodes.
- **Leaf node:** A node that has no child nodes below.
- **Internal node:** Nodes that has at least a single child node inside the tree.
- **Path:** It is a sequence directed from top to down, starting from any internal node of the tree to the target descendant node. Path consists of both nodes and edges, not only the edges between the nodes. Path starts from a node and ends at another node or leaf.
- **Degree of a node:** It is the total amount of child nodes connected to any node. Leaf nodes has a degree of zero.
- **Height of a node:** Amount of edges that is on the longest path from the particular node to a leaf node.
- **Height of a tree:** Height of a tree is equal to the height of it's root node. This is calculated from down to the top.

When calculating the height, it is very important to make the distinction between if the calculation is being made for the edge count or the node count.

When it is edge count - Leaf nodes always have a height of 0.

When it is node count - Leaf nodes always start from a height of 1.

Then starting from deepest leaf node, each level above is incremented by 1 until the root node is reached to get the total height.

- **Depth of a node:** Amount of edges that is from a particular node to the Tree's root node. Depth of a Tree is calculated by the depth of it's deepest leaf node. Root node always have the depth of 0, depth is incremented by 1 until reaching down to the deepest leaf node.
- **Level of a tree:** Each downwards step from top of the tree (root node) to the bottom is called as level of a tree. Root node starts from 0 and increments by 1 at each step until the leaf node.

Types of Tree data structure

Tree data structure is a very broad term that includes variety of different Tree implementations. But in general, we can divide them into 2 main categories: General Trees and it's descendants. Descendants of General Tree comes with their unique restrictions and rule sets.

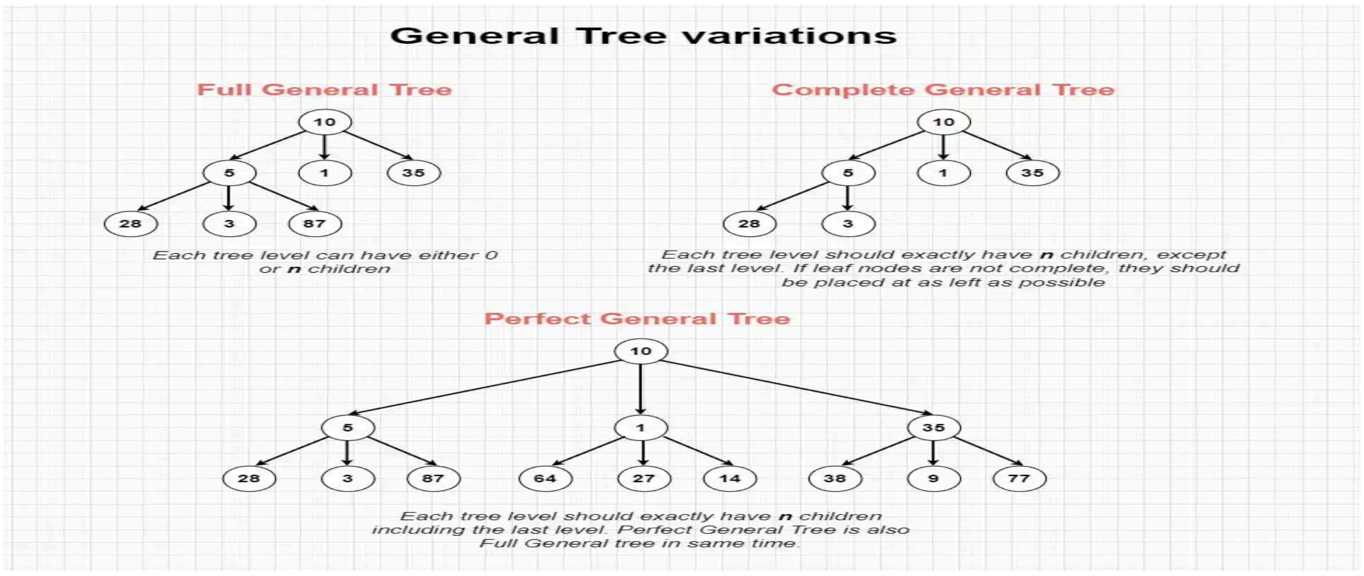
General Tree

General Tree - which is also called ***Generic***, ***n-ary***, ***m-ary***, ***k-ary*** or ***k-way Tree*** is a rooted Tree that is the base for all Tree variations. A node can have zero or any number of children, as well as there is no restrictions on how nodes are being inserted.

"n", **"m"** or **"k"** letters in the name variations represents the upper bound / maximum number of child nodes on any level. For example, most popular descendant **Binary Trees** have the ***"n = 2"*** as a restriction, another descendant **Ternary Trees** is restricted with ***"n = 3"***.

All Tree variations derives from General Tree. In other words, it is the common ancestor for all Tree types.

Variations of General tree


General Tree variations
Full General Tree
Complete General Tree
Each tree level can have either 0 or n children
Each tree level should exactly have n children, except the last level. If leaf nodes are not complete, they should be placed at as left as possible
Perfect General Tree
Each tree level should exactly have n children including the last level. Perfect General Tree is also Full General tree in same time.
10
5
1
35
28
3
87
64
27
14
38
9
77

Full General Tree: Where each tree level can have either 0 or "n" children.

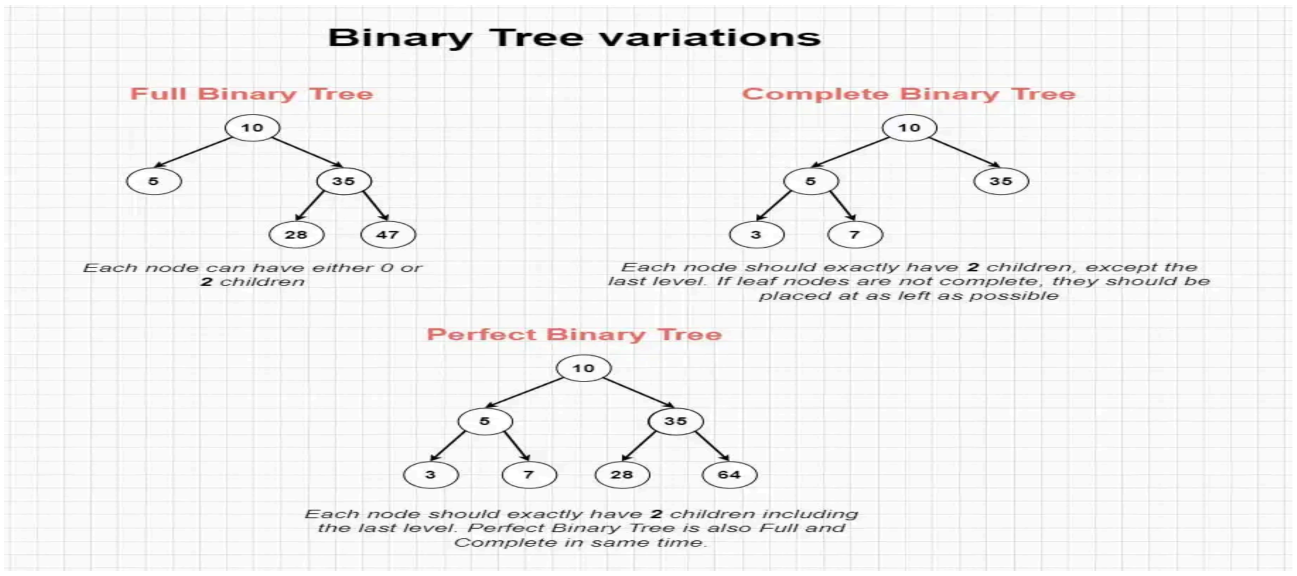
Complete General Tree: Where each tree level should exactly have "n" children, except the last level. If leaf nodes are not complete, they should be placed at as left as possible.

Perfect General Tree: Where each tree level should exactly have "n" children including the last level. Perfect General Tree is also Full General tree in same time.

Binary Tree

Binary Tree is the most popular specialized version of the General Tree, where each node inside the Binary Tree can have maximum 2 children. Because of this, the maximum number of nodes doubles as we go down on each level.

Variations of Binary Tree


Binary Tree variations
Full Binary Tree
10
5
35
28
47
Each node can have either 0 or 2 children
Complete Binary Tree
10
5
35
3
7
Each node should exactly have 2 children, except the last level. If leaf nodes are not complete, they should be placed at as left as possible
Perfect Binary Tree
10
5
35
3
7
28
64
Each node should exactly have 2 children including the last level. Perfect Binary Tree is also Full and Complete in same time.

Full Binary Tree: Where each node can have either 0 or 2 children.

Complete Binary Tree: Where each node should exactly have 2 children, except the last level. If leaf nodes are not complete, they should be placed at as left as possible.

Perfect Binary Tree: Where each node should exactly have 2 children including the last level. Perfect Binary Tree is also both complete and full in same time.

Calculating the maximum possible number of nodes in Binary Tree

Binary trees comes with a standard way of calculating the maximum possible number of nodes - which is by using “Exponentiation”.

To fully understand how does this help us with calculations in Binary Tree, let’s have a brief introduction to the concept. “Exponentiation” is a mathematical operation that involves two numbers which can be shown as b^n. It is often pronounced like “b raised to the power of n”.

- “b” represents the base number.
- “n” represents the exponent number.

For example when we say 2^3 or 2 (b) raised to the power of 3 (n), this is what happens:

2 x 2 x 2 = 8

In simpler terms, we basically say “multiply this number by itself n amount of times” - as long as the exponent is not zero. Which brings us to another important key point: Whenever we use the zero as an exponent number - no matter which number we use as a base, the result will be always 1:

2 to the power of 0 = 1

4 to the power of 0 = 1

This is because when we have the zero as an exponent, there is no amount of multiplication that needs to be done for the number, so we are left with the number itself at the end. Then we look at how can we get the number itself after a multiplication. Since the only way to get the number itself in a multiplication is using number 1, we return the number 1 as the result. I know this might sound confusing, if you'd like to explore further I'd suggest you to research about the "Empty product rule" in mathematics.

As a side note, "Exponentiation" might look very similar to the "Square of a number" concept. It is good to remember the distinction between them even though they look very similar:

Square of a number is simply means "multiply this number by itself". Therefore the exponent or power is always 2.

Square of 2 -> 2^2 -> 2 x 2 = 4

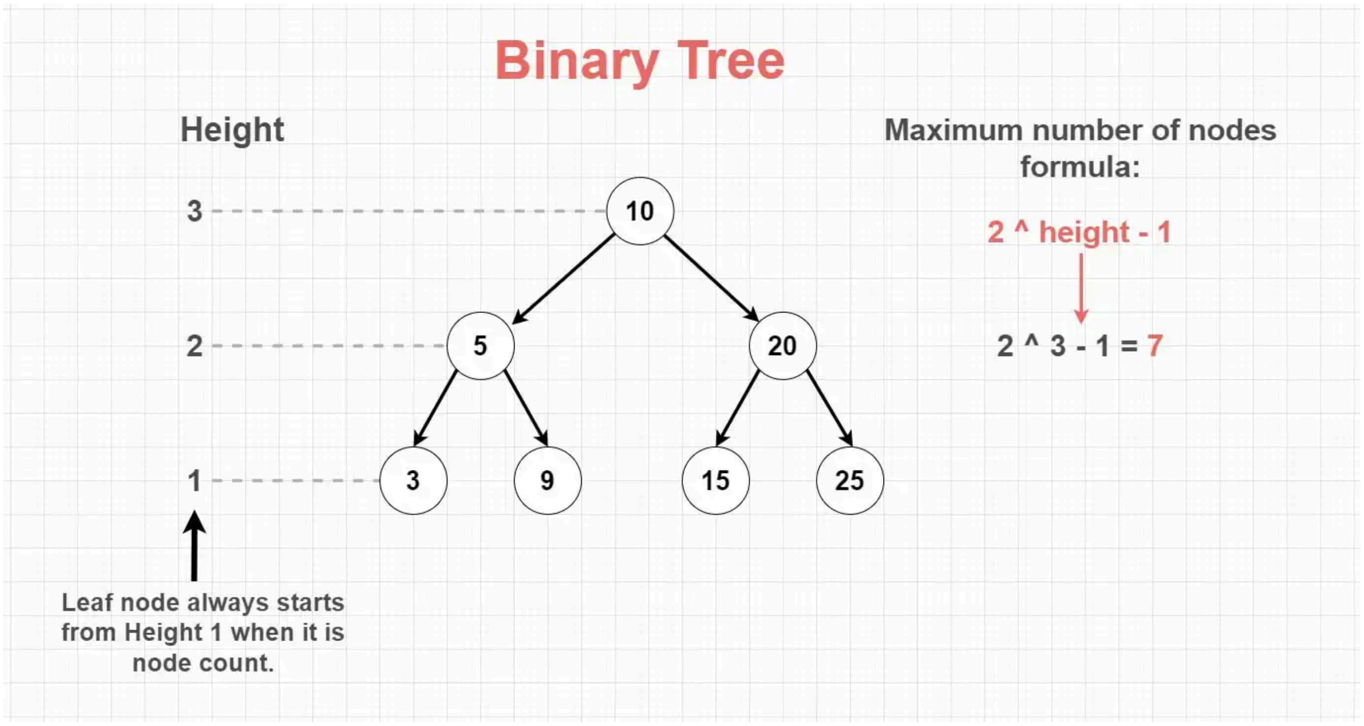
Square of 4 -> 4^2 -> 4 x 4 = 16

Exponentiation simply means the base number can be multiplied by any exponent number, not just 2.

Now we have the "Exponentiation" concept cleared up, let's proceed with how to calculate the maximum number of nodes in a Binary Tree. The formula is:

(2 ^ Height - 1) OR (2 to the power of Height - 1)

Take a look at the visual below to see the formula in action:


Binary Tree
Height
3
2
1
10
5
20
3
9
15
25
Leaf node always starts from Height 1 when it is node count.
Maximum number of nodes formula:
2 ^ height - 1
2 ^ 3 - 1 = 7

We can also reach to the same result if we use the sum of all levels. Since Tree levels always starts from zero; result would be:

2 ^ Level 0 = 1

2 ^ Level 1 = 2

2 ^ Level 2 = 4

Total: 1 + 2 + 4 = 7

Applications of Binary Tree

Binary Tree is one of the most used Tree data structure in real life applications.

Similar to General Tree, Binary Tree also has variety of specialized versions. Some examples are: Binary Search Tree, AVL Tree, Red Black Tree, BSP tree, Binary Heap, Treap and more.