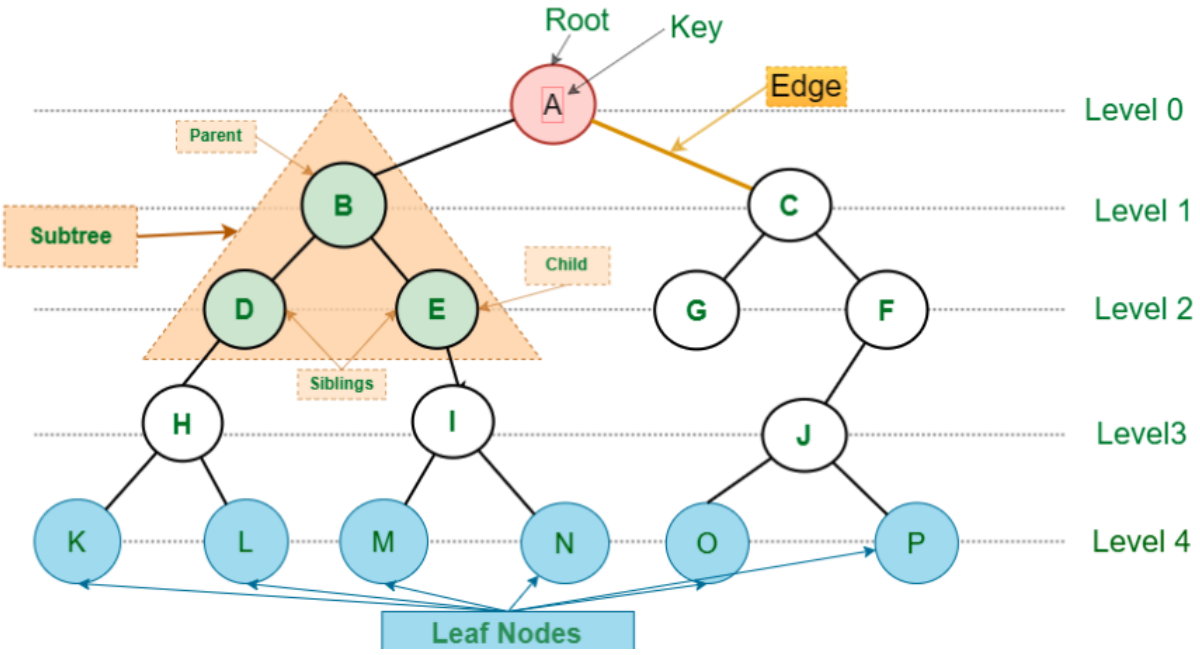
Binary Search Trees

Trees consist of vertices (nodes) and edges that connect them.

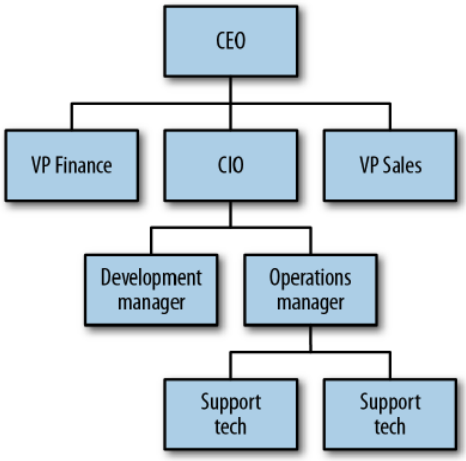
Unlike linear data structures, trees are **hierarchical**. An **edge** connects two nodes in a tree.. Each node in a tree contains a value.

Trees are similar to Graphs, except that a **cycle** cannot exist in a Tree - they are acyclic. It does not contain cycles which means it does not contain any closed loops (aka cycles). In other words, there is always exactly one path between any two nodes.

- A Tree is a type of Graph (aka minimally connected Graph)
- Trees cannot be disconnected (all nodes must connect)
- No more than two connected nodes
- Each node can only have **one parent**
- Nodes are acyclic, they do not have cycles, only point downwards



Examples of trees can be the organizational charts, the **file system** on your computer and the **DOM** of the HTML structure of the websites that you've been building is organized in a tree-like structure.

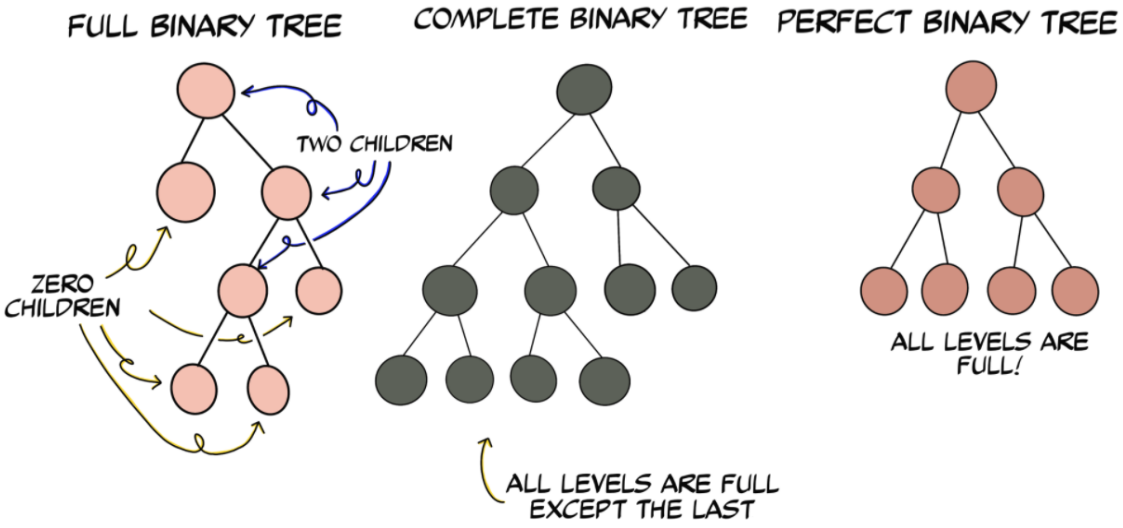


Tree data structures can exist in several forms, including a special variant called the **binary tree.**

A tree can be broken down into levels. The root node is at level 0, its children are at level 1, and so on. A node at any level is considered the **root of a subtree**, which consists of that root node's children, its children's children, etc. The depth of a tree as the number of layers in the tree.

Trees have different classifications:

- A **balanced tree** if each sub-tree is recursively balanced and the height of the two sub-trees differs by at most one.
- A **complete tree** if it is filled left to right with no missing nodes.
- A **binary tree** if all nodes in the tree have at most 2 children.
- A **full tree** if all nodes of the tree have exactly 0 or N children.
- A **perfect tree** if all nodes of the tree have exactly N children.
- A **unary tree** if each node only has 1 child (linked list)



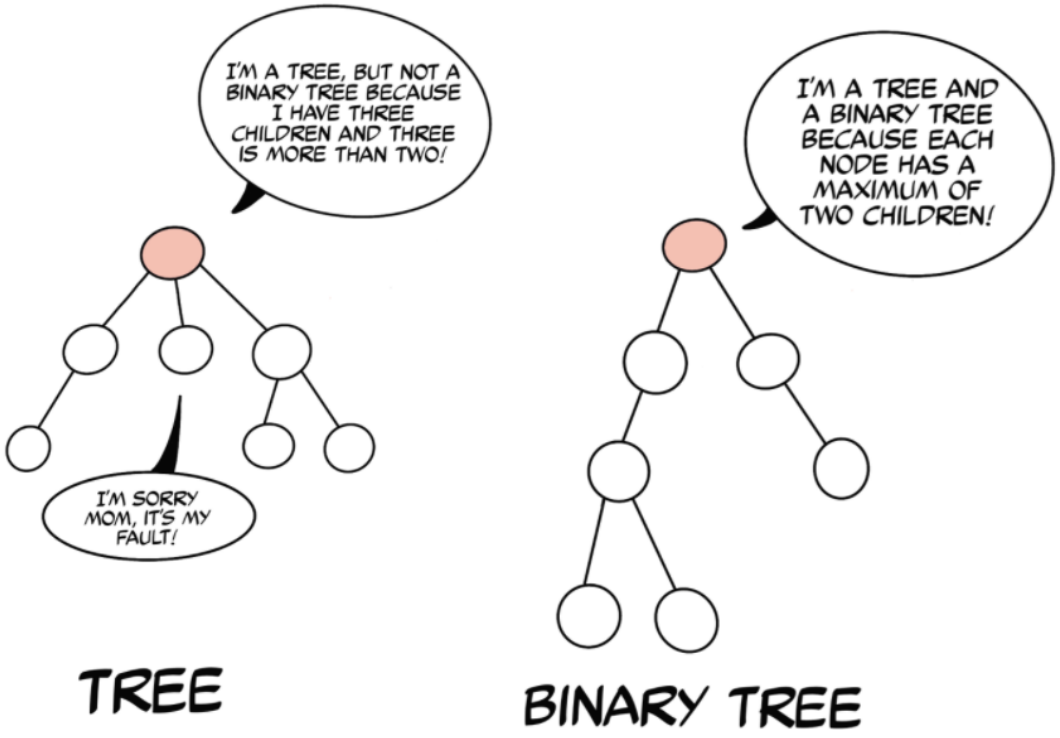
Binary Tree

In a **binary tree**, every node can have at most **two child nodes**. In other words, each node branches out in a maximum of two directions.

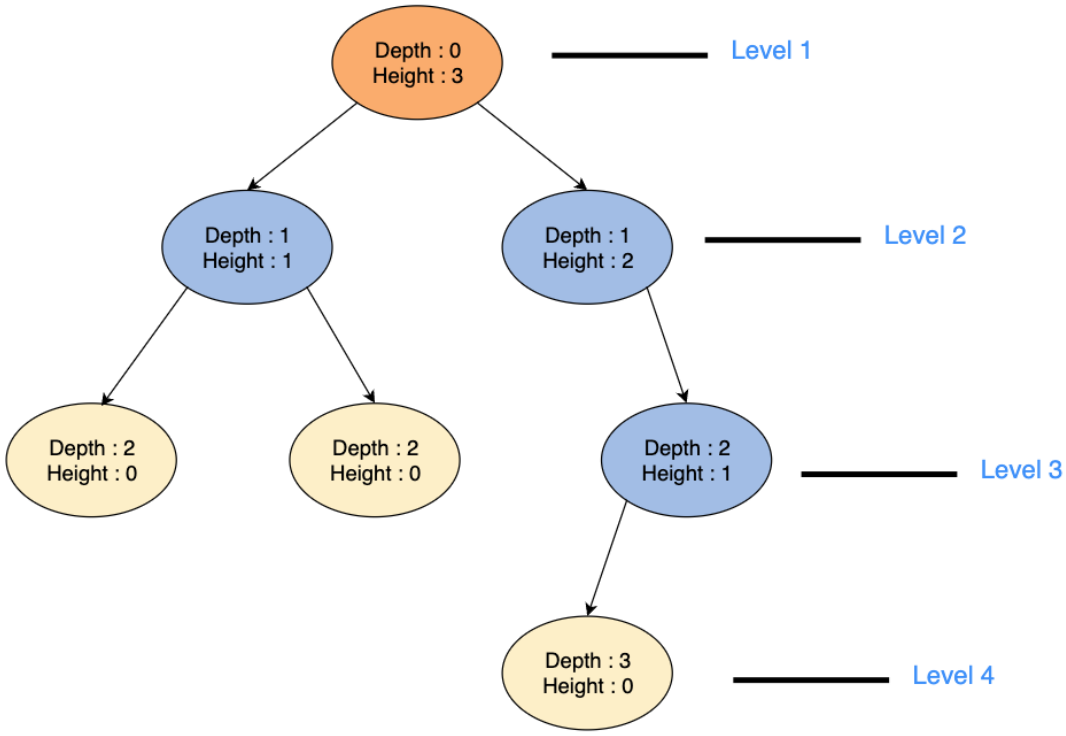
A binary tree is a data structure containing a collection of elements often referred to as nodes. It can store any data item, be it names, places, numbers, etc. of any type. It enables quick search, addition, and removal of items

A binary tree contains a root node with a left subtree and a right subtree. A branch in a tree signifies a **decision path**, a choice that connects one node to another.

A binary tree may have a left branch and a right branch. It may also have subtrees. A **subtree** is a mini tree within a binary tree, whose root can be any node and all of its descendants rooted at that node.

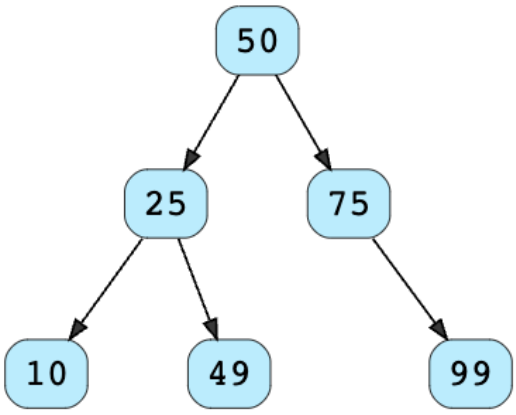


- **root:** the entry point to a tree; the highest level node in the tree
- **child:** the direct descendant of a node
- **parent:** the direct ascendant of a node
- **sibling:** two nodes are siblings if both share the same parent
- **leaf:** a node with no children
- **edge:** connects two nodes in a tree
- **subtree:** a node or collection of connected nodes of a tree
- **degree of a node:** total number of children of a node
- **path:** The path between two nodes *u* and *v* is an alternating sequence of nodes and edges, which starts at *u* and ends at *v*. In a tree, there is exactly one path between any pair of nodes.
- **height of a node:** # of nodes on the path from the node to the deepest leaf (bottom to top)
- **depth of a node:** length of path from a node to root (top to bottom)
- **depth of a tree:** distance (edge count) from root to the farthest leaf



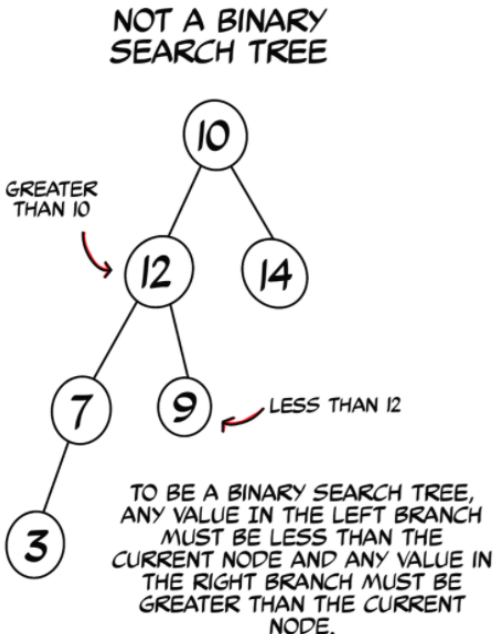
Binary Search Trees (BST)

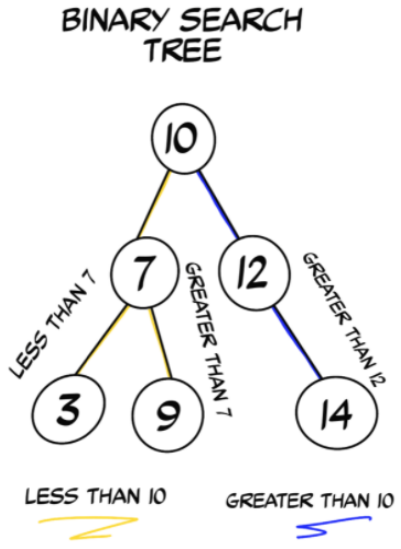
A Binary Search Tree is a binary tree that helps in searching the data in the tree. The **key** of the node is **greater** than all the nodes in its left subtree and is **smaller** than all the nodes in its right subtree



Binary search trees may be used in cases when we need to maintain dynamically changing datasets in sorted order, for some sortable data.

- Each node has zero, one, or two children.
- All nodes in the **left-hand branch** of a node have **lower values** than the node itself.
- All of the nodes in the **right-hand branch** of a node have a **higher value** than the node itself.
- Both the left and right **subtrees** are BSTs themselves.





Implementing a BST

In a binary tree, in many practical situations, a node contains several fields of data with a key. **Each key must be unique.**

Though you may see examples (in interview problems) that are very simplistic that only have a single value like integer which acts both as a **key** and **value**.

Node class

```
class Node {
 constructor(value = null) {
      //this.key = key;
      this.value = value;
      this.left = null;
      this.right = null;
   }
}
```

Does this look familiar to you? It's almost like the linked list node, but instead of having next and previous, it has left and right. That guarantees that we have at most two children.

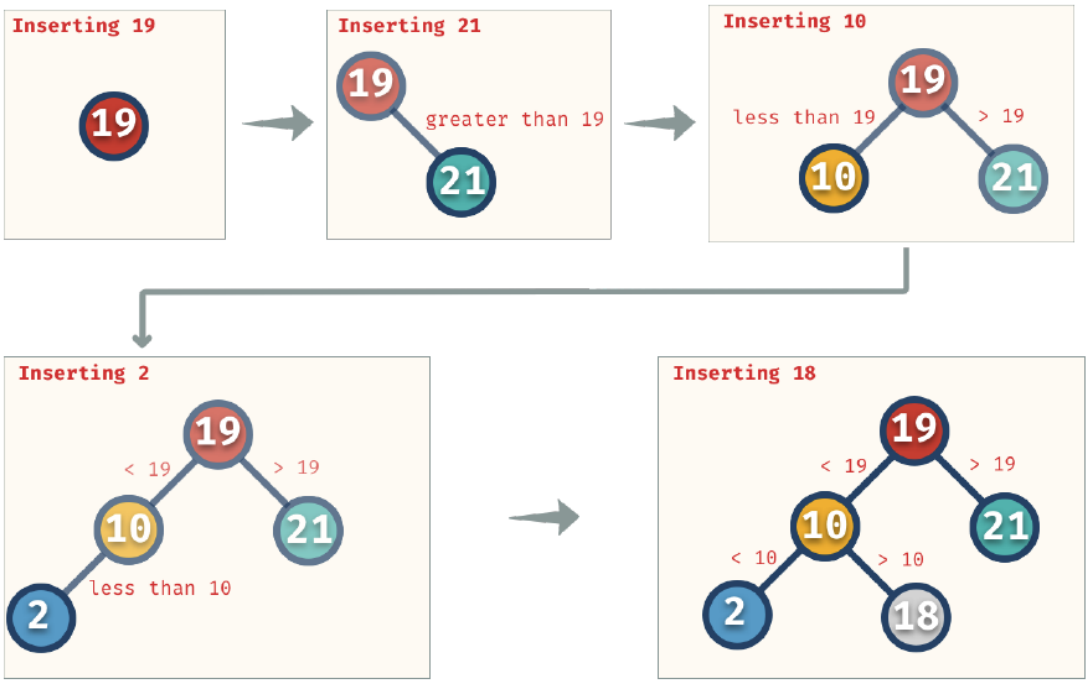
BST class

```
class BSTe {
 constructor() {
     This.root = null;
   }
}
```

Inserting new elements in a BST

1. If the tree is empty (root element is null), we add the newly created node as root, and that's it!
2. If the root is not null. Start from it and compare the node's value against the new element. If the node has higher than a new item, we move to the right child, otherwise to the left. We check each node recursively until we find an empty spot to put the new element.

Let's say that we want to insert the values 19, 21, 10, 2, 18:



We start by the root when we insert node 18 (19). Since 18 is less than 19, then we move left. Node 18 is greater than 10, so we move right. There's an empty spot, and we place it there.

Binary Search Tree Insertion (Implementation)

```
insert(value) {

    const newNode = new Node(value);

    // If no root node exists, set root to new node and return
    if (this.root === null) {
      this.root = newNode;
      return;
    }

    // Starting from the root node
    let current = this.root;

    while(true) {

      if (value < current.value) {
        if (current.left === null) {
          current.left = newNode;
          return;
        } else {
          current = current.left;
        }
      } else {
        if (current.right === null) {
          current.right = newNode;
          return;
        } else {
          current = current.right;
        }
      }
    }
  }

const bst = new BST();
bst.insert(19);
bst.insert(21);
bst.insert(10);
bst.insert(2);
bst.insert(18);
```

The purpose of the loop is to continuously traverse the binary search tree, moving either to the left or right child of each node based on the comparison between the value to be inserted and the current node's value. The two return statements that will eventually be reached and will exit the loop.

- If the new value is less than the current node's value, move to the left child.
- If the left child is null, insert the new node as the left child and return.
- If the new value is greater than or equal to the current node's value, move to the right child.
- If the right child is null, insert the new node as the right child and return.

Note: This method could also be written recursively.

Test out yourself visually: https://visualgo.net/en/bst