**Tree Data Structure**

***Tree*** is an abstract data type that stores elements hierarchically. With the exception of the top element, each element in a tree has a ***parent*** element and zero or more ***children*** elements. A tree is usually visualized by placing elements inside ovals or rectangles, and by drawing the connections between parents and children with straight lines. We typically call the top element the ***root*** of the tree, but it is drawn as the highest element, with the other elements being connected below (just the opposite of a botanical tree).

# **Characteristics**

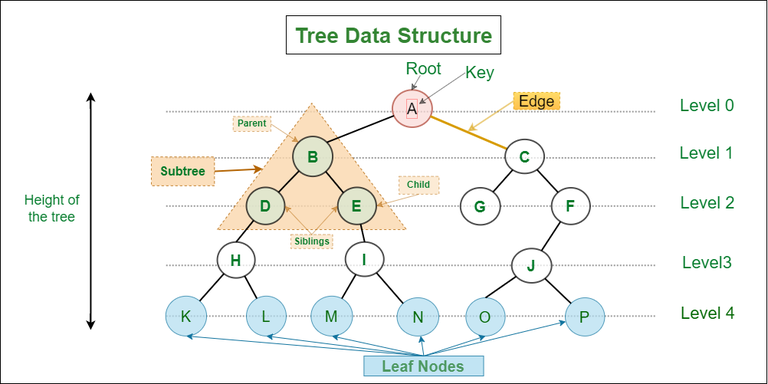
* **Root**: The top node of the tree from which all other nodes branch out. A tree has exactly one root node.
* **Node**: Each element in a tree is called a node. It contains data and may have links/references to other nodes.
* **Parent and Child Nodes**: In a tree, if a node A is directly connected to node B, and A is above B, then A is the parent of B, and B is the child of A.
* **Siblings**: Nodes that are at the same level and have the same parent are called siblings.
* **Leaf Nodes**: Nodes that do not have any children are called leaves or leaf nodes.
* **Edges**: The connection between one node and another.
* **Path**: A sequence of nodes and edges connecting a node with a descendant.
* **Height of a Node**: The number of edges on the longest downward path between that node and a leaf.
* **Depth of a Node**: The number of edges from the root to the node.
* **Height of the Tree**: The height of the root node.
* **Subtree**: A tree consisting of a node and all its descendants.

# **Ancestors**

* An **ancestor** of a node is any predecessor node on the path from the root to that node, including the node itself and the root. It represents the nodes that need to be traversed to reach the node from the root.
* For example, in a binary tree, if you have a path from the root node **A** to a leaf node **D** going through nodes **B** and **C** (i.e., A -> B -> C -> D), then the ancestors of node **D** are **D** itself, **C**, **B**, and **A**.
* The **root node** of a tree is an ancestor to all nodes in the tree.

# **Descendants**

* A **descendant** of a node includes any successor nodes that can be reached from that node, including the node itself. It represents all the nodes that are part of the subtree rooted at the given node.
* Continuing the previous example, if you consider node **B**, the descendants of **B** are **B** itself, **C**, and **D**.
* **Leaf nodes** have no children, so their only descendant is themselves.



# **Types of Trees**

1. **Binary Tree**: A tree in which each node has up to two children, commonly referred to as the left and right children.
2. **Binary Search Tree (BST)**: A binary tree in which for each node, all elements in the left subtree are less than the node, and all elements in the right subtree are greater.
3. **Balanced Tree**: A tree in which the height of the left and right subtrees of any node differ by no more than one.
4. **AVL Tree**: A self-balancing binary search tree where the difference between heights of left and right subtrees cannot be more than one for all nodes.
5. **Red-Black Tree**: Another type of self-balancing binary search tree.
6. **B-Tree/B+ Tree**: Balanced tree data structures optimized for systems that read and write large blocks of data. They are commonly used in databases and file systems.
7. **Trie (Prefix Tree)**: A specialized tree used in searching, especially with strings. Each node represents a character of a string.

# **Applications**

* **Storing Hierarchical Data**: Trees are ideal for representing hierarchical structures, such as file systems or organizational structures.
* **Databases**: Trees, especially binary search trees, AVL trees, and B-trees, are used in databases and indexing services to efficiently manage large datasets.
* **Searching**: Trees like binary search trees and AVL trees offer efficient search operations.
* **Routing Algorithms**: Trees are used in networking for routing algorithms.
* **Syntax Trees**: Used by compilers and interpreters to parse expressions.

## **Binary Trees**

A binary tree is a fundamental data structure in computer science and programming, used to organize data hierarchically. It is called "binary" because each node in the tree can have at most two children, commonly referred to as the left child and the right child. Here's a closer look at binary trees, including their types, properties, and applications.

### **Structure**

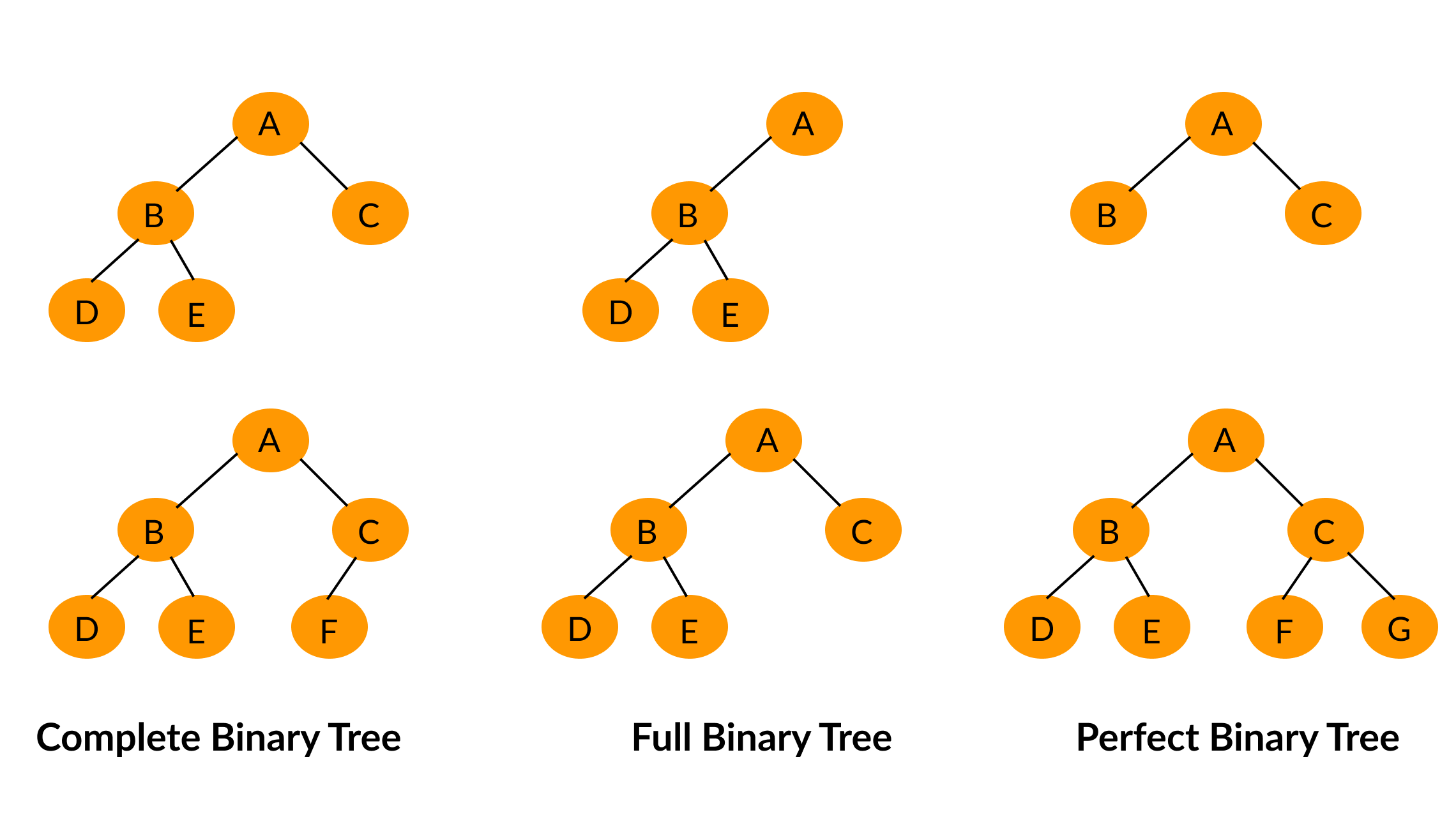
A binary tree consists of nodes, where each node contains the following elements:

* **Data**: The value or record held by the node.
* **Left Pointer**: A reference to the left child node (if any).
* **Right Pointer**: A reference to the right child node (if any).

The topmost node in a binary tree is called the **root**. A node without children is called a leaf node. The depth of a node is the number of edges from the root to the node, and the height of a binary tree is the maximum depth among all its nodes.

**Types of Binary Trees**

1. **Full Binary Tree**: Every node has either 0 or 2 children.
2. **Complete Binary Tree**: All levels are fully filled, except possibly the last level, which is filled from left to right.
3. **Perfect Binary Tree**: All internal nodes have two children, and all leaves are at the same level.
4. **Balanced Binary Tree**: The height of the left and right subtrees of any node differ by no more than 1.
5. **Binary Search Tree (BST)**: A special kind of binary tree where the value of each node is greater than all values in its left subtree and less than all values in its right subtree.

****

### **Operations**

Common operations performed on binary trees include:

* **Traversal**: Visiting all nodes in some order. Common traversal algorithms include In-order, Pre-order, Post-order, and Level-order traversals.
* **Insertion**: Adding a new node. The specifics of insertion can vary, especially in the case of binary search trees.
* **Deletion**: Removing a node. This operation can be complex, especially in binary search trees, due to the need to maintain specific properties.
* **Searching**: Finding a node with a given value, particularly efficient in binary search trees.
* **Balancing**: Adjusting the tree to improve operation efficiency, applicable in balanced binary trees like AVL trees and Red-Black trees.

### **Applications**

Binary trees, and binary search trees, in particular, are widely used in computing for various applications, including:

* **Database Indexing**: BSTs provide efficient search operations, making them suitable for database indexing.
* **Expression Parsing**: Binary trees can represent and evaluate arithmetic expressions.
* **Priority Queues**: Binary heap trees are used to implement priority queues.
* **Huffman Coding Tree**: Used in data compression algorithms.

## Decision Trees vs Full Binary Trees

**Full Binary Trees**

A **full binary tree** (sometimes called a **proper binary tree** or a **strictly binary tree**) is a type of binary tree in which every node has either zero or two children. This means that no node in a full binary tree has only one child. Full binary trees are defined strictly by this structural property, regardless of the data they contain or their application.

**Decision Trees**

**Decision trees**, on the other hand, are a model used in decision-making and machine learning to represent decisions and their possible consequences, including chance event outcomes, resource costs, and utility. Decision trees are used for classification and regression in machine learning, where the decisions or splits at each node are based on feature values, aiming to classify data points or predict outcomes.

While decision trees are structurally binary trees (since each decision leads to two subsequent possibilities), they are not necessarily full binary trees. A node in a decision tree can have two children (when a decision splits into two paths) but might also have a single child in certain scenarios, especially when pruning techniques are applied to simplify the model or avoid overfitting. The focus in decision trees is more on the decision-making process and the flow from a root question to conclusions, rather than strictly adhering to the full binary tree's structural requirement.

**Key Differences**

* **Purpose and Application**: Full binary trees are a generic concept used in computer science for various applications, focusing on structure. Decision trees are specifically used in decision-making processes and machine learning, focusing on modeling decisions and outcomes.
* **Structural Requirements**: Full binary trees require each node to have exactly zero or two children. Decision trees may have nodes with one or two children, depending on the decisions and outcomes being modeled.
* **Representation**: While both can be represented as binary trees, decision trees typically include additional information such as decision criteria, probabilities, and outcomes at each node.

An arithmetic expression can be represented by a binary tree whose leaves are associated with variables or constants, and whose internal nodes are associated with one of the operators +, −, ×, and /.

A diagram of a tree

Description automatically generated