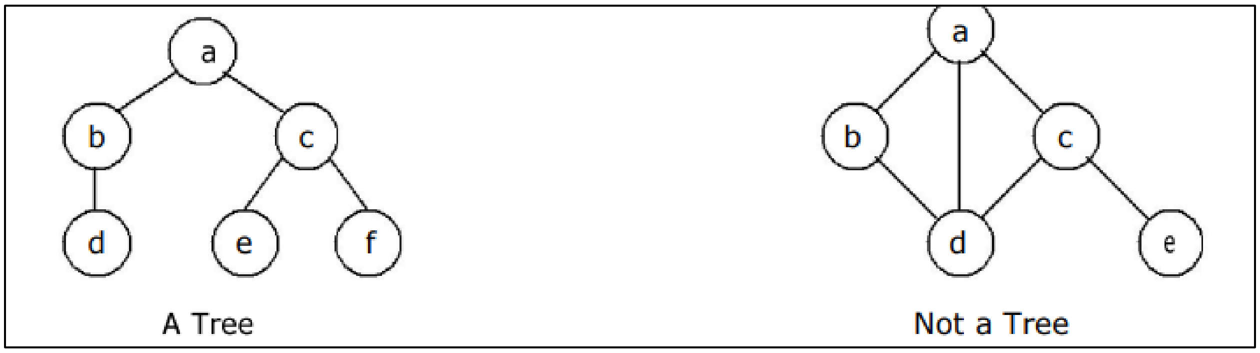
BINARY TREE

A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a subtree. At the bottom of the tree are leaf nodes, which have no children.



BASIC TERMS USED IN TREE DATA STRUCTURE

Root: The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is the root node of the tree. If a node is directly linked to some other node, it would be called a parent-child relationship.

Child node: If the node is a descendant of any node, then the node is known as a child node.

Parent: If the node contains any sub-node, then that node is said to be the parent of that sub-node.

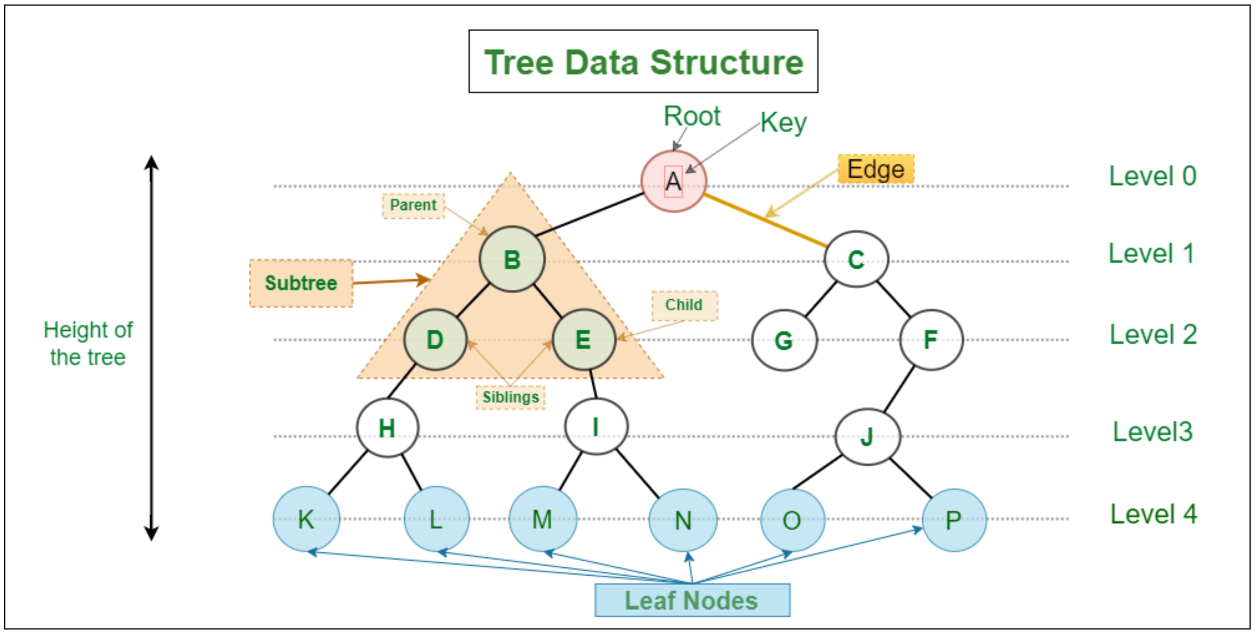
Sibling: The nodes that have the same parent are known as siblings.

Leaf Node: The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.

Internal nodes: A node has atleast one child node known as an internal

Ancestor node: An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.

Descendant: The immediate successor of the given node is known as a descendant of a node. In the below figure, 10 is the descendant of node 5.



PROPERTIES OF TREE DATA STRUCTURE

Recursive data structure: The tree is also known as a recursive data structure.

Number of edges: If there are n nodes, then there would n-1 edges.

Depth of node x: The depth of node x can be defined as the length of the path from the root to the node x.

Height of node x: The height of node x can be defined as the longest path from the node x to the leaf node.

APPLICATIONS OF TREES

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.

- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

TYPES OF TREE DATA STRUCTURE

- Binary tree
- Binary Search tree
- AVL tree

BINARY TREE

The Binary tree means that the node can have maximum two children. Each node can have either 0, 1 or 2 children.

PROPERTIES OF BINARY TREES

1. If h = height of a binary tree, then
    a. Maximum number of leaves = 2^h
    b. Maximum number of nodes = $2^{h+1} - 1$
2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l node at level l.
4. The total number of edges in a full binary tree with n node is n - 1.

TYPES OF BINARY TREES

Full Binary Tree: Every node has either 0 or 2 children. A full binary tree of height hhh will have 2h2^h2h leaf nodes and 2h+1−12^{h+1} - 12h+1−1 total nodes.

Complete Binary Tree: All levels are fully filled except possibly for the last level, which is filled from left to right. A complete binary tree is very useful for implementing data structures like heaps.

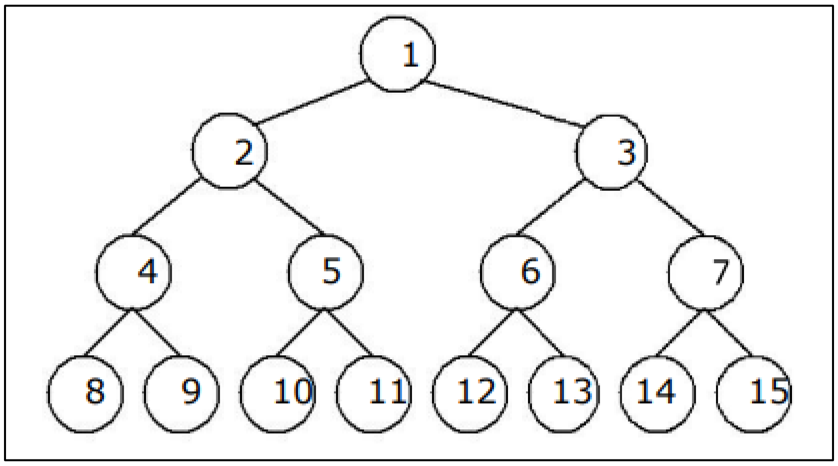
Perfect Binary Tree: Every internal node has exactly two children, and all leaf nodes are at the same level.

Balanced Binary Tree: A tree where the height difference between the left and right subtrees of every node is at most 1.

FULL BINARY TREE

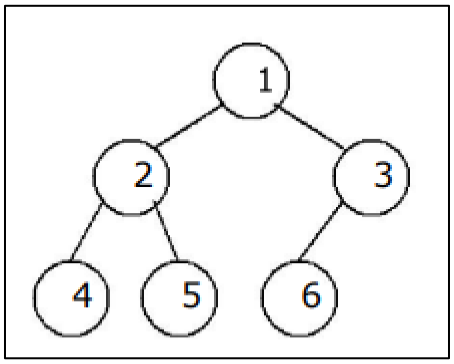
A full binary tree of height h has all its leaves at level h. Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

A full binary tree with height *h* has $2^{h + 1}$ - 1 nodes. A full binary tree of height h is a *strictly binary tree* all of whose leaves are at level h. Figure 5.2.3(d) illustrates the full binary tree containing 15 nodes and of height 3.
A full binary tree of height h contains 2^h leaves and, 2^h - 1 non-leaf nodes.



COMPLETE BINARY TREE

A binary tree with n nodes is said to be complete if it contains all the first n nodes of the above numbering scheme. Figure shows examples of complete and incomplete binary trees. A complete binary tree of height h looks like a full binary tree down to level h-1, and the level h is filled from left to right. A complete binary tree with n leaves that is not strictly binary has 2n nodes. For example, the tree of Figure 5.2.3(c) is a complete binary tree having 5 leaves and 10 nodes.



TREE TRAVERSAL

The process of visiting the nodes is known as tree traversal. There are three types traversals used to visit a node:

1. **Inorder traversal**

The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for inorder traversal is as follows:

```
void inorder(node *root)
{
        if(root != NULL)
        {
                inorder(root->lchild);
                print root -> data;
                inorder(root->rchild);
        }
}
```

2. Preorder traversal

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

```
void preorder(node *root)
{
        if( root != NULL )
        {
                print root -> data;
                preorder (root -> lchild);
                preorder (root -> rchild);
        }
}
```

3. Postorder traversal

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

The algorithm for postorder traversal is as follows:

```
void postorder(node *root)
{
        if( root != NULL )
        {
                postorder (root -> lchild);
                postorder (root -> rchild);
                print (root -> data);
        }
}
```

BINARY TREE C++ IMPLEMENTATION

```
#include <iostream>

using namespace std;



// Definition of the Node structure

class Node {

public:
    int data;
    Node* left;
    Node* right;

    // Constructor to initialize a new node
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

// BinaryTree class encapsulates all the operations related to the tree
class BinaryTree {
private:
    Node* root;  // Root of the binary tree

    // function to insert a new value recursively
    Node* insert(Node* node, int value) {
        if (node == nullptr) {
            return new Node(value);  // Create a new node if current node is null
        }
        if (value < node->data) {
            node->left = insert(node->left, value);  // Insert in the left subtree
        } else {
            node->right = insert(node->right, value); // Insert in the right subtree
        }
        return node;
    }

    // function for in-order traversal (left, root, right)
```

```
void inorderTraversal(Node* node) const {
   if (node == nullptr) return;
   inorderTraversal(node->left);
   cout << node->data << " ";
   inorderTraversal(node->right);
}

// function for pre-order traversal (root, left, right)
void preorderTraversal(Node* node) const {
   if (node == nullptr) return;
   cout << node->data << " ";
   preorderTraversal(node->left);
   preorderTraversal(node->right);
}

// function for post-order traversal (left, right, root)
void postorderTraversal(Node* node) const {
   if (node == nullptr) return;
   postorderTraversal(node->left);
   postorderTraversal(node->right);
   cout << node->data << " ";
}

// function to delete a node from the binary tree
Node* deleteNode(Node* node, int value) {
   if (node == nullptr) return node;
```

```
    if (value < node->data) {
        node->left = deleteNode(node->left, value);  // Go to left subtree
    } else if (value > node->data) {
        node->right = deleteNode(node->right, value);  // Go to right subtree
    } else {
        // Node found, now handle the cases
        if (node->left == nullptr) {
            Node* temp = node->right;
            delete node;
            return temp;
        } else if (node->right == nullptr) {
            Node* temp = node->left;
            delete node;
            return temp;
        }
        // Node with two children: find the inorder successor (smallest in right subtree)
        Node* temp = findMin(node->right);
        node->data = temp->data;
        node->right = deleteNode(node->right, temp->data);  // Delete the successor
    }
    return node;
}

// function to find the node with minimum value (used for deletion)
Node* findMin(Node* node) {
    while (node->left != nullptr) {
        node = node->left;
    }
    return node;
}

// function to delete the entire tree
void deleteTree(Node* node) {
    if (node == nullptr) return;
    deleteTree(node->left);
    deleteTree(node->right);
    cout << "Deleting node with value: " << node->data << endl;
    delete node;
}

// function to print the tree in a simple 2D format
void printTree(Node* node, int space) const {
    if (node == nullptr) return;

    space += 5;

    // Print right subtree first
    printTree(node->right, space);

    // Print current node after spacing
    cout << endl;
    for (int i = 5; i < space; i++) cout << " ";
    cout << node->data << "\n";
```

```
        // Print left subtree
        printTree(node->left, space);
    }

public:
    // Constructor to initialize an empty tree
    BinaryTree() : root(nullptr) {}

    // Insert a new value into the binary tree
    void insert(int value) {
        root = insert(root, value);
    }

    // Delete a node from the binary tree
    void deleteNode(int value) {
        root = deleteNode(root, value);
    }

    // Perform in-order traversal
    void inorderTraversal() const {
        inorderTraversal(root);
        cout << endl;
    }

    // Perform pre-order traversal
    void preorderTraversal() const {
        preorderTraversal(root);
        cout << endl;
    }

    // Perform post-order traversal
    void postorderTraversal() const {
        postorderTraversal(root);
        cout << endl;
    }

    // Print the tree structure
    void printTree() const {
        printTree(root, 0);
    }

    // Destructor to clean up the tree's memory
    ~BinaryTree() {
        deleteTree(root);
    }
};

int main() {
    BinaryTree tree;

    // Insert nodes into the binary tree
    tree.insert(10);
    tree.insert(5);
    tree.insert(15);
tree.insert(3);

tree.insert(7);

tree.insert(12);

tree.insert(18);


// Perform different traversals

cout << "In-order traversal: ";

tree.inorderTraversal();


cout << "Pre-order traversal: ";

tree.preorderTraversal();


cout << "Post-order traversal: ";

tree.postorderTraversal();


// Print the binary tree structure

cout << "\nBinary Tree Structure:\n";

tree.printTree();


// Delete a node

cout << "\nDeleting node with value 10\n";

tree.deleteNode(10);


// Print the tree after deletion

cout << "\nBinary Tree Structure After Deletion:\n";

tree.printTree();
```

```
    return 0;
}
```