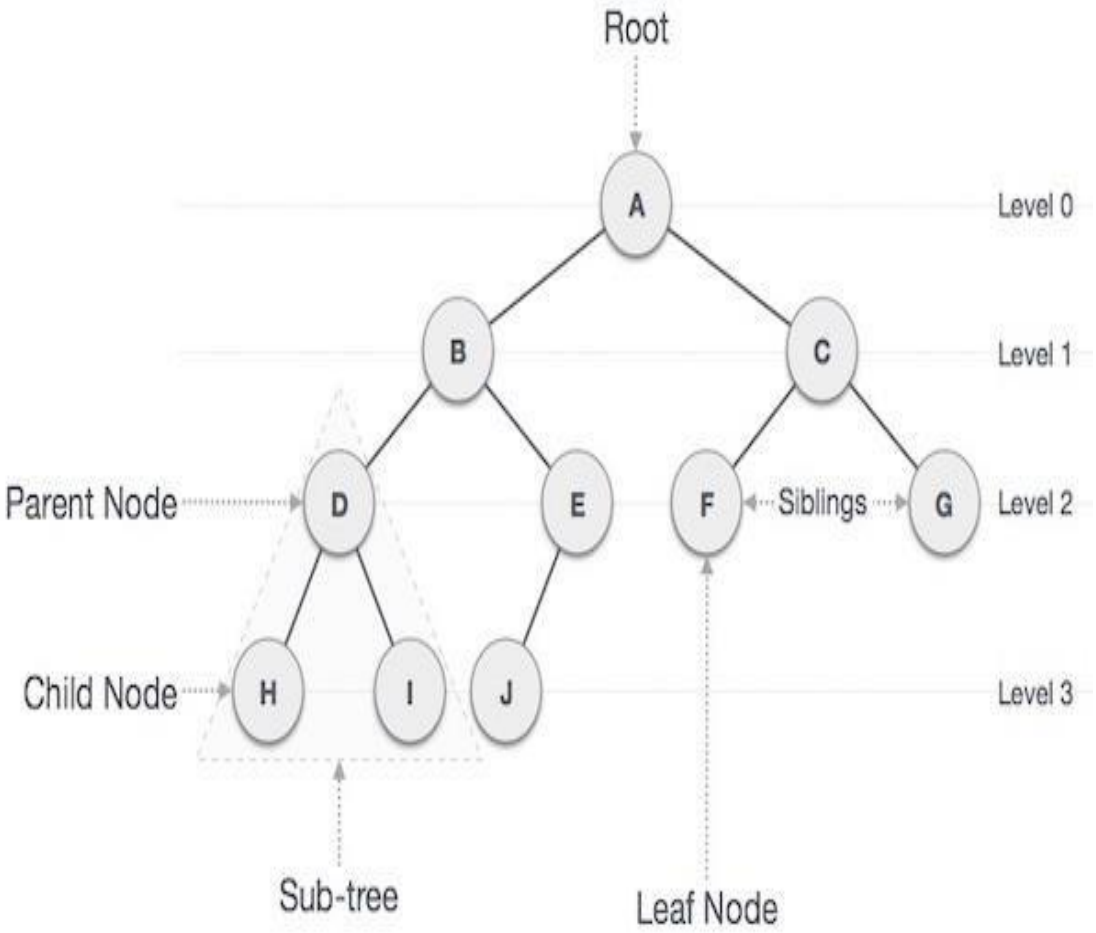
Tree Data Structure

Introduction

- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type.
- Each node contains some data and the link or reference of other nodes that can be called children.

Basic terminologies



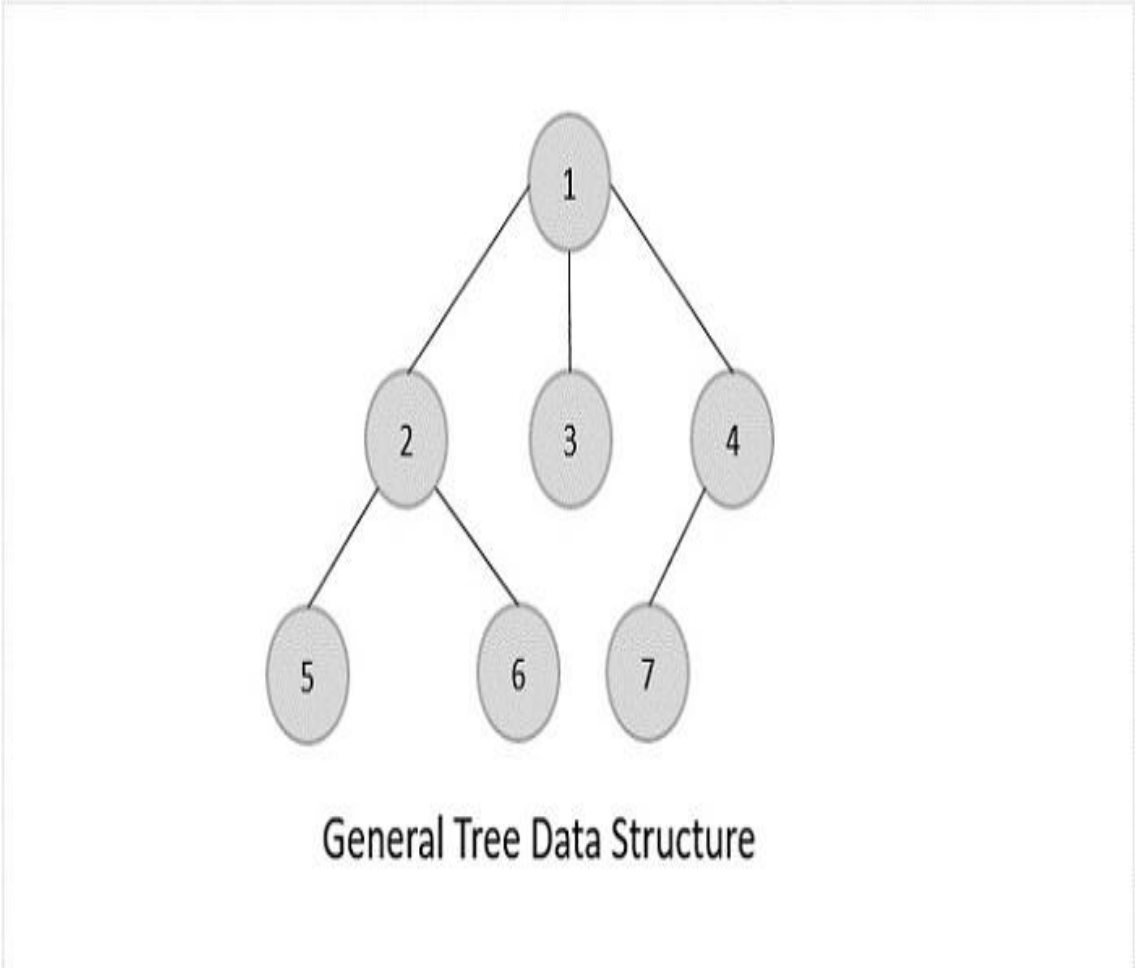
Root − The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
Parent − Any node except the root node has one edge upward to a node called parent.
Child − The node below a given node connected by its edge downward is called its child node.
Leaf − The node which does not have any child node is called the leaf node.
Subtree − Subtree represents the descendants of a node.
Visiting − Visiting refers to checking the value of a node when control is on the node.
Traversing − Traversing means passing through nodes in a specific order.
Levels − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
Keys − Key represents a value of a node based on which a search operation is to be carried out for a node.
Edge: A connection between two nodes. It defines the relationship between parent and child nodes.
Path − Path refers to the sequence of nodes along the edges of a tree.
Height of a node- The number of edges on the longest path from that node to a leaf.
Height of a tree- The height of the root node.
Depth of a node-The number of edges from the root to that node.
Degree of a node- The number of children a node has

Types of Trees

- General Trees
- Binary Trees
- Binary Search Trees

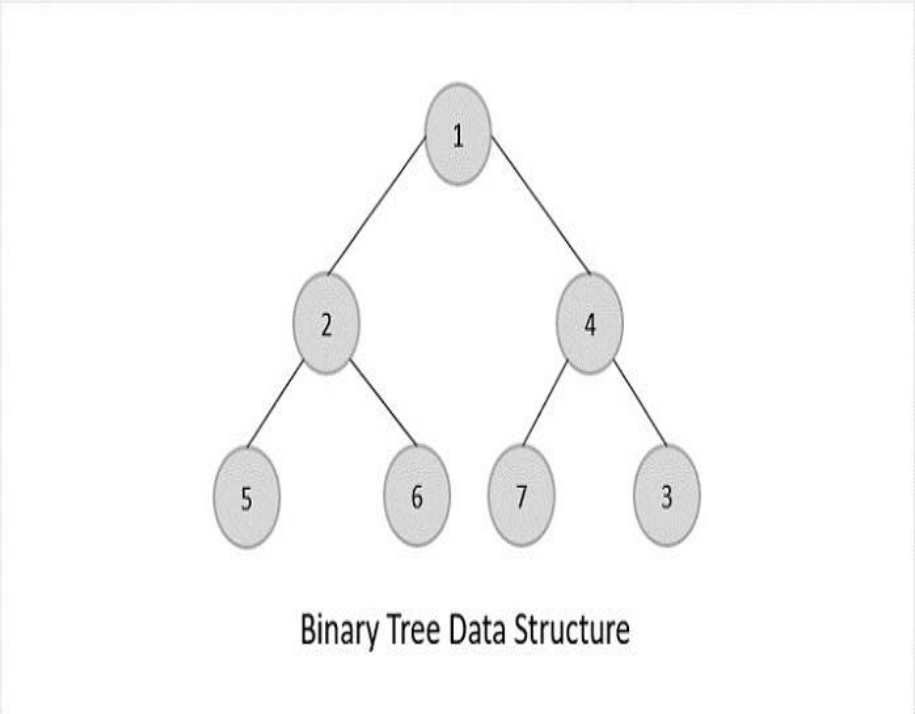
General Trees

- General trees are unordered tree data structures where the root node has minimum 0 or maximum ‘n’ subtrees.
- The General trees have no constraint placed on their hierarchy. The root node thus acts like the superset of all the other subtrees.

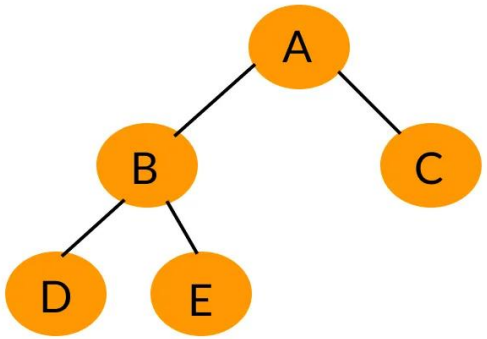


Binary Trees

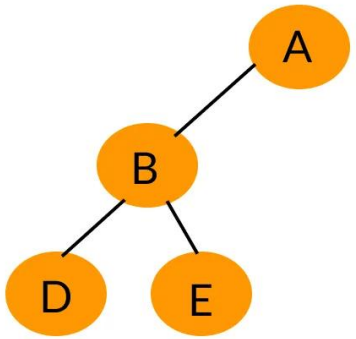
- Binary Trees are general trees in which the root node can only hold up to maximum 2 subtrees: left subtree and right subtree.
- Based on the number of children, binary trees are divided into three types:
  - Full Binary Tree
    - A full binary tree is a binary tree type where every node has either 0 or 2 child nodes.
  - Complete Binary Tree
    - A complete binary tree is a binary tree type where all the leaf nodes must be on the same level.
    - Root and internal nodes in a complete binary tree can either have 0, 1 or 2 child nodes.
  - Perfect Binary Tree
    - A perfect binary tree is a binary tree type where all the leaf nodes are on the same level and every node except leaf nodes have 2 children.



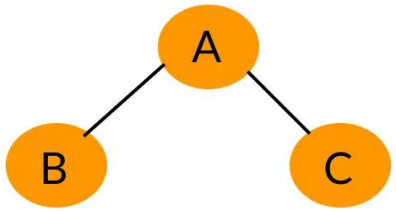
Binary Tree Data Structure



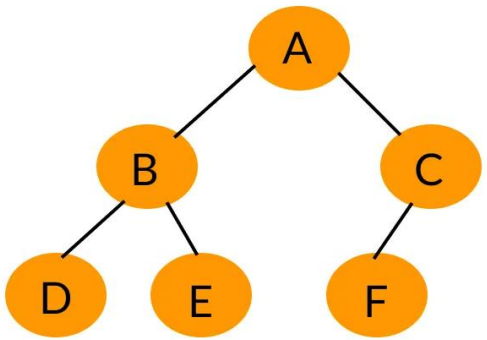
Complete Binary Tree

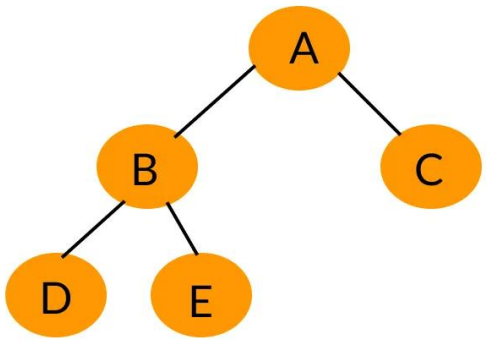


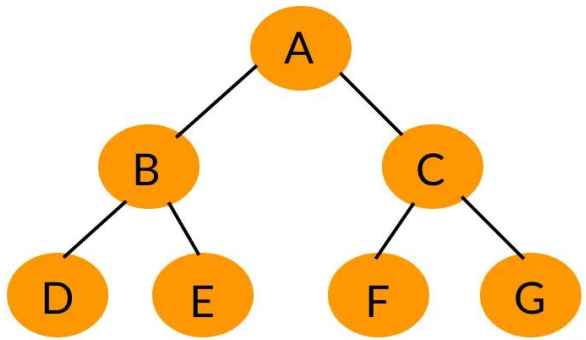
Full Binary Tree



Perfect Binary Tree

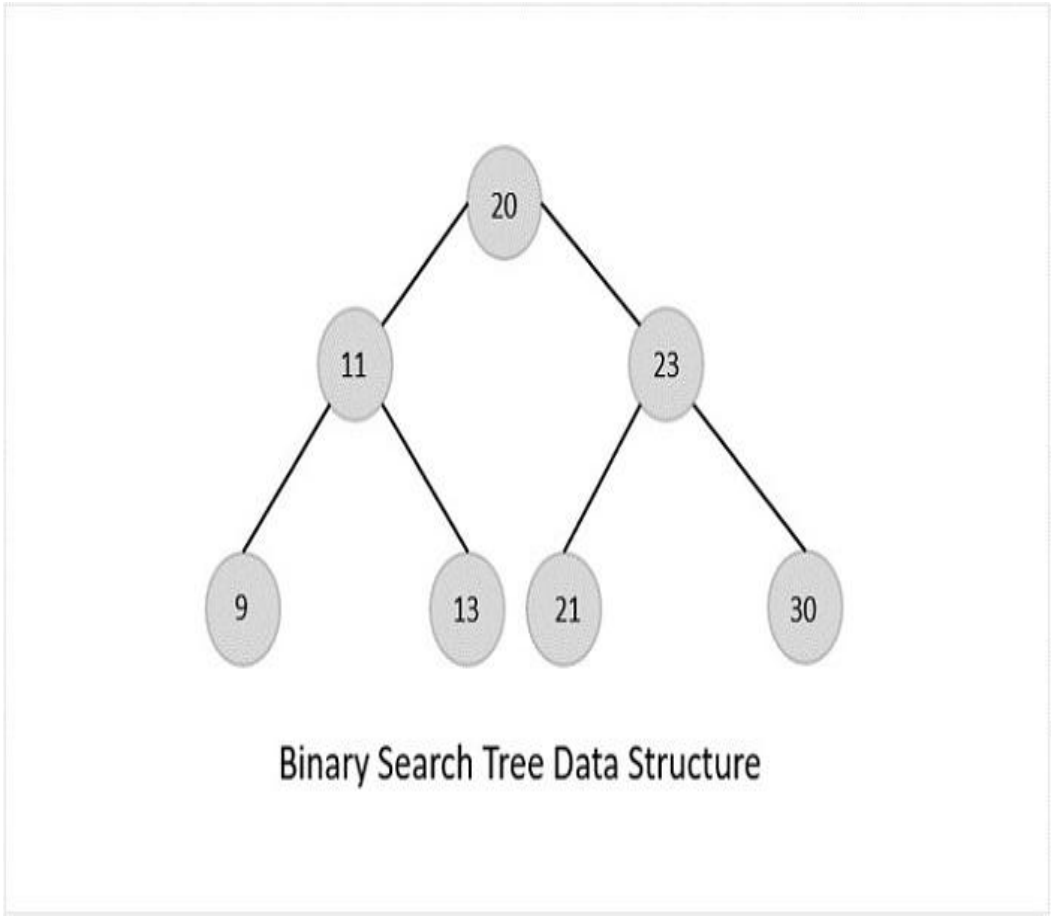






Binary Search Trees

- Binary Search Trees possess all the properties of Binary Trees including some extra properties of their own, based on some constraints, making them more efficient than binary trees.
- The data in the Binary Search Trees (BST) is always stored in such a way that the values in the left subtree are always less than the values in the root node and the values in the right subtree are always greater than the values in the root node, i.e. left subtree < root node ≤ right subtree.



Binary Search Tree Data Structure

Advantages of BST

- Binary Search Trees are more efficient than Binary Trees since time complexity for performing various operations reduces.
- Since the order of keys is based on just the parent node, searching operation becomes simpler.
- The alignment of BST also favors Range Queries, which are executed to find values existing between two keys. This helps in the Database Management System.

Disadvantages of BST

- This **skewness** will make the tree a linked list rather than a BST, since the worst case time complexity for searching operation becomes O(n).

Tree Operations

- **Insertion**: Adding a node to the tree while maintaining its properties.
- **Deletion**: Removing a node from the tree while maintaining the tree's properties.
- **Traversal**: Visiting all nodes in a particular order:
  - ✓ **In-order** (left, root, right): Often used in binary search trees to retrieve elements in sorted order.
  - ✓ **Pre-order** (root, left, right): Useful for copying trees.
  - ✓ **Level-order** (breadth-first): Visiting nodes level by level from the root.
  - ✓ **Post-order** (left, right, root): Useful for deleting trees.

Applications of Trees

- **File systems**: Hierarchical storage of files and directories.
- **Databases**: B-trees and variants are used for indexing.
- **Networking**: Routing algorithms use tree structures.
- **Expression evaluation**: Expression trees are used to evaluate algebraic expressions.

Binary Search Tree Implementation

```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};

struct Node *newNode(int data) {
    struct Node *node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}
```

```c
struct Node *insert(struct Node *node, int data) {
    if (node == NULL) {
        return newNode(data);
    }

    if (data < node->data) {
        node->left = insert(node->left, data);
    } else if (data > node->data) {
        node->right = insert(node->right, data);
    }

    return node;
}
```

```c
// Inorder traversal (Left, Root, Right)
void inorder(struct Node *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// Preorder traversal (Root, Left, Right)
void preorder(struct Node *root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

// Postorder traversal (Left, Right, Root)
void postorder(struct Node *root) {
    if (root != NULL) {
        postorder(root->left);
        postorder(root->right);
        printf("%d ", root->data);
    }
}
```

```c
int main() {
    struct Node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    printf("Inorder traversal: ");
    inorder(root);
    printf("\n");

    printf("Preorder traversal: ");
    preorder(root);
    printf("\n");

    printf("Postorder traversal: ");
    postorder(root);
    printf("\n");

    return 0;
}
```

Deleting a node in a **Binary Search Tree (BST)**

- The process depends on the node you want to delete and how many children it has.
- Here are the three main cases to consider when deleting a node in a BST:

1. **Node has no children (Leaf node)**:
    - Simply remove the node.
2. **Node has one child**:
    - Remove the node and replace it with its child.
3. **Node has two children**:
    - Find the node's **inorder predecessor/ inorder successor**, copy its value to the node being deleted, and then delete the predecessor or successor.

```c
// Function to delete a node in BST
struct Node* deleteNode(struct Node* root, int key) {
    // Base case: if the tree is empty
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the root's key, go to
        the left subtree
    if (key < root->data)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key, go to
        the right subtree
    else if (key > root->data)
        root->right = deleteNode(root->right, key);

    // If key is the same as root's key, then this is the node to be
        deleted
    else {
        // Case 1: Node with only one child or no child
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }

        // Case 2: Node with two children
        // Get the inorder successor (smallest in the right subtree)
        struct Node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->data = temp->data;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->data);
    }

    return root;
}
```

```c
// Function to find the minimum value node in a subtree (used in case 3)
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}
```