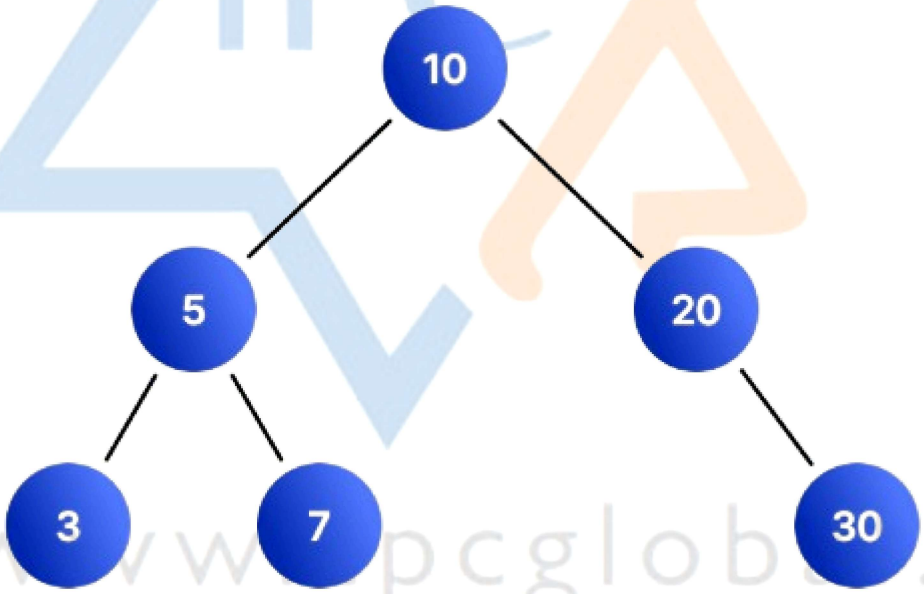
Binary Search Tree (BST)

Introduction

A **Binary Search Tree (BST)** is a special type of **Binary Tree** where the nodes are arranged in a specific order:

- **Left Subtree** → contains only nodes with values **less than** the parent.
- **Right Subtree** → contains only nodes with values **greater than** the parent.
- No duplicate nodes (in most standard definitions).



Properties of BST

- **Inorder Traversal** (Left → Root → Right) of BST always gives **sorted values.**
    - Example above: `1, 3, 4, 6, 7, 8, 10, 13, 14`
- **Time Complexity** (on balanced BST):
    - Search → `O(log n)`
    - Insertion → `O(log n)`
    - Deletion → `O(log n)`
    - In the worst case (skewed tree), operations degrade to `O(n)`.

Basic BST Operations

A. Search in BST

Algorithm:

1. Start from the root.
2. If `key == root->data`, return found.
3. If `key < root->data`, search in **left subtree.**
4. Else search in **right subtree.**

B. Insertion in BST

Algorithm:

1. Start from root.

2. If tree is empty → create new node.
3. If `key < root->data` → go to left subtree.
4. Else → go to right subtree.
5. Insert node at correct position.

C. Deletion in BST

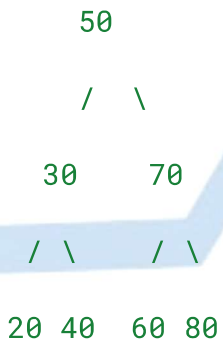
There are **3 cases**:

1. **Node has no child (leaf)** → just delete.
2. **Node has one child** → replace node with child.
3. **Node has two children** →
    - Find **Inorder Successor** (smallest node in right subtree).
    - Copy its value to current node.
    - Delete inorder successor.

Example Walkthrough

Let's insert values: `50, 30, 70, 20, 40, 60, 80`

e started with:

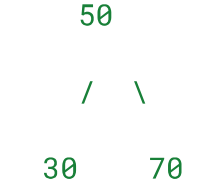


Now, we want to **insert(25)**:

1. Compare with `50` → since `25 < 50`, go **left** to `30`.
2. Compare with `30` → since `25 < 30`, go **left** to `20`.
3. Compare with `20` → since `25 > 20`, go **right.**

The right of `20` is empty → insert here.

So the tree becomes:



```
        / \     / \
      20  40   60  80
        \
       25
```