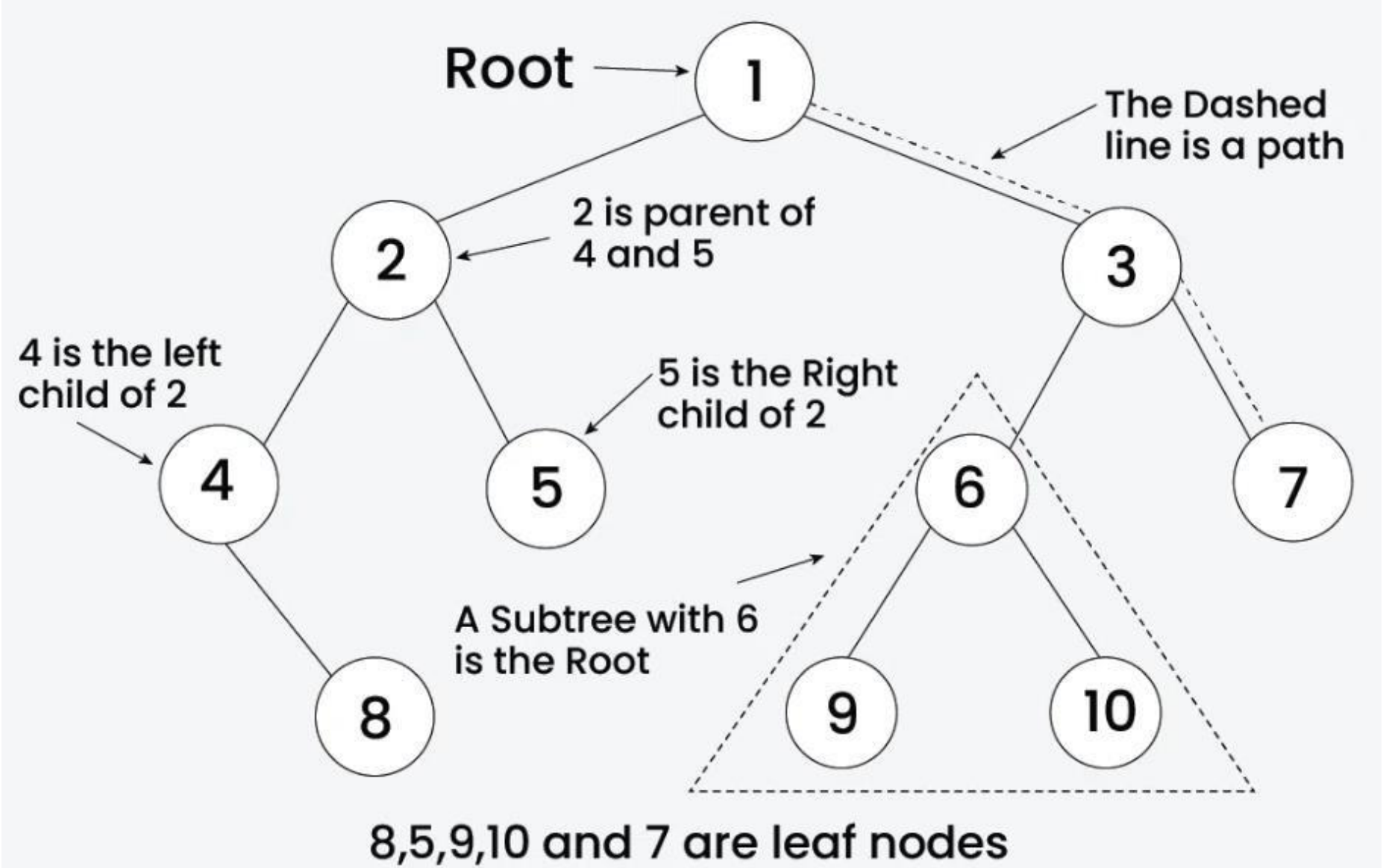
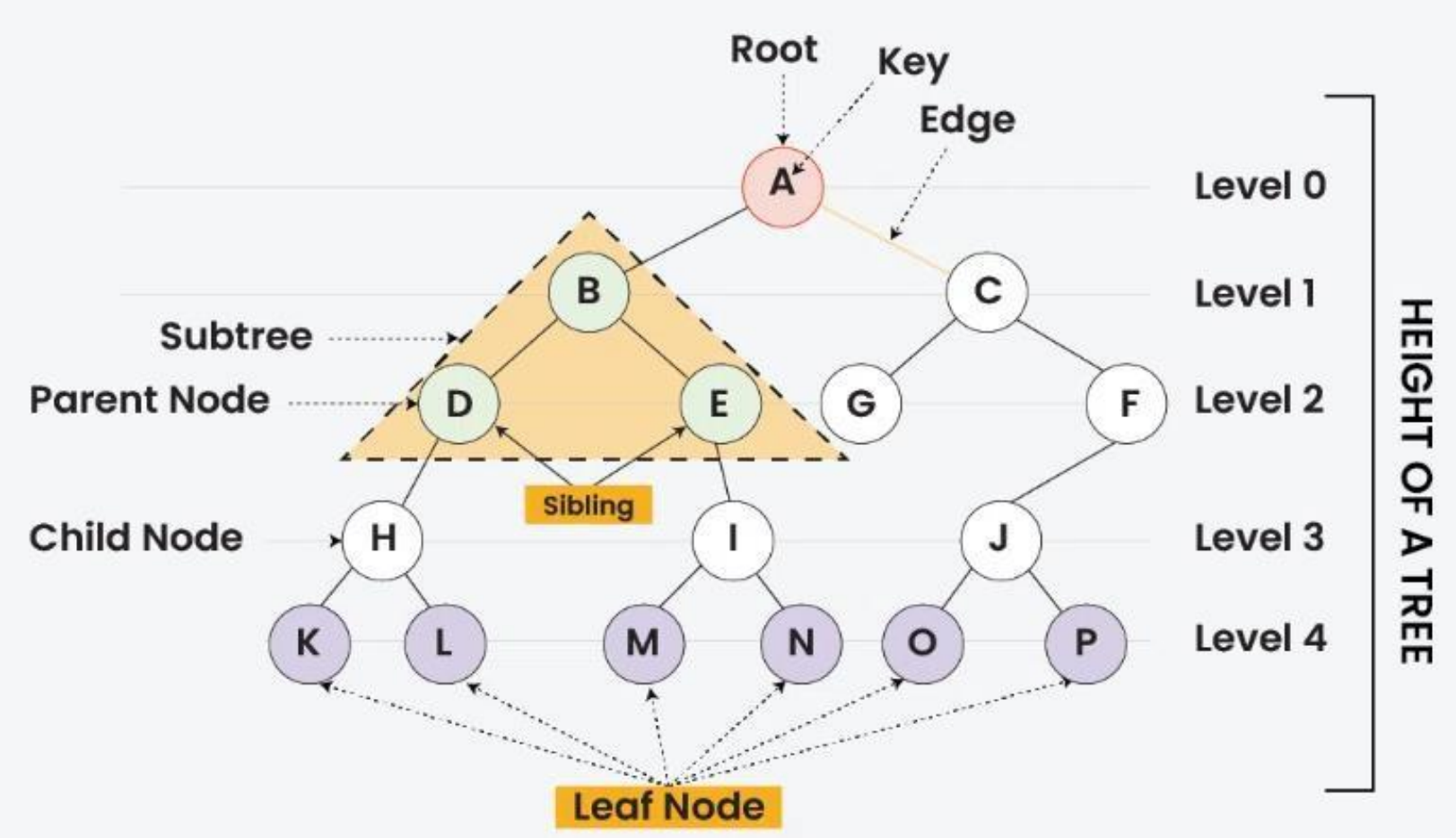

Strategy
Growth
Solutions
Success
Growth
Strategy
Solutions
Sales
Success
Strategy
Business
Solutions
Success
Sales
Sales
Strategy
Business
Solutions
Success
Sales
Growth
Strategy
Business
Solutions

Binary Tree

Binary Tree: Definition

- Binary Tree is a non-linear and hierarchical data structure (general tree), where each node has at most two children referred to as the left child and the right child.
- The topmost node in a binary tree is called the root, and the bottom-most nodes are called leaves.


Root
1
The Dashed
line is a path
2 is parent of
4 and 5
2
3
4 is the left
child of 2
5 is the Right
child of 2
4
5
6
7
A Subtree with 6
is the Root
8
9
10
8,5,9,10 and 7 are leaf nodes



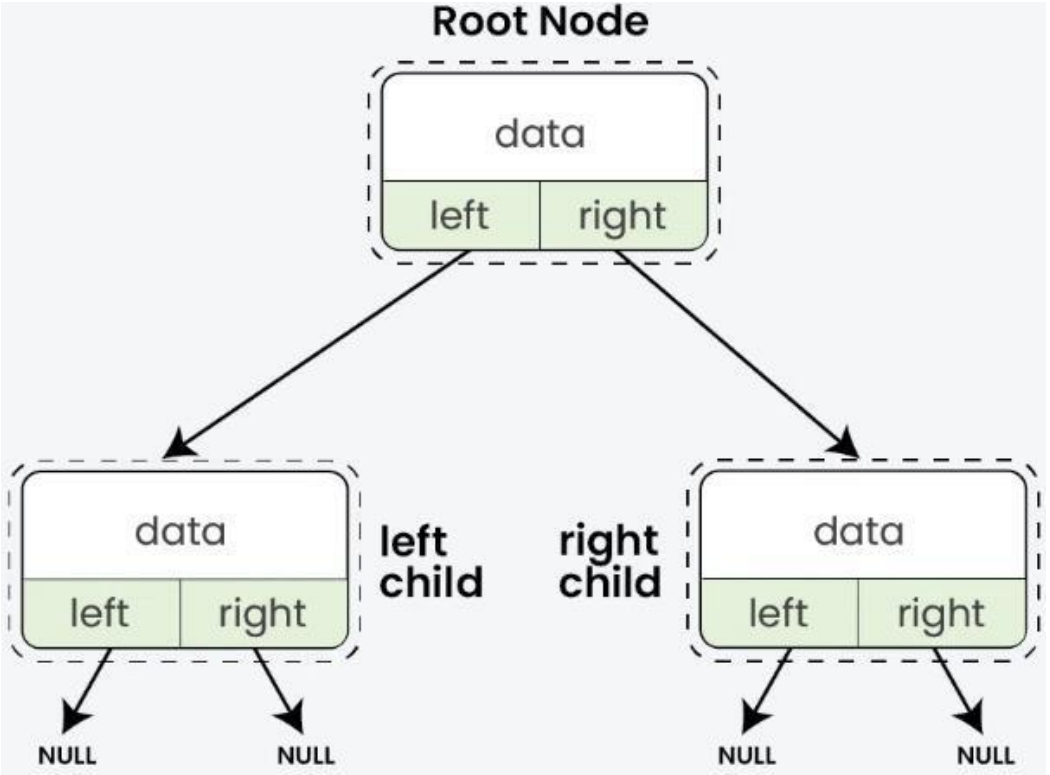
Binary Tree: Terminologies

Binary Tree: Representation

Each node in a Binary Tree has three parts:

- Data
- pointer to the left child
- pointer to the right child

```
// Node Structure
struct Node {
    int  data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = right = nullptr;
    }
};
```



Binary Search Tree

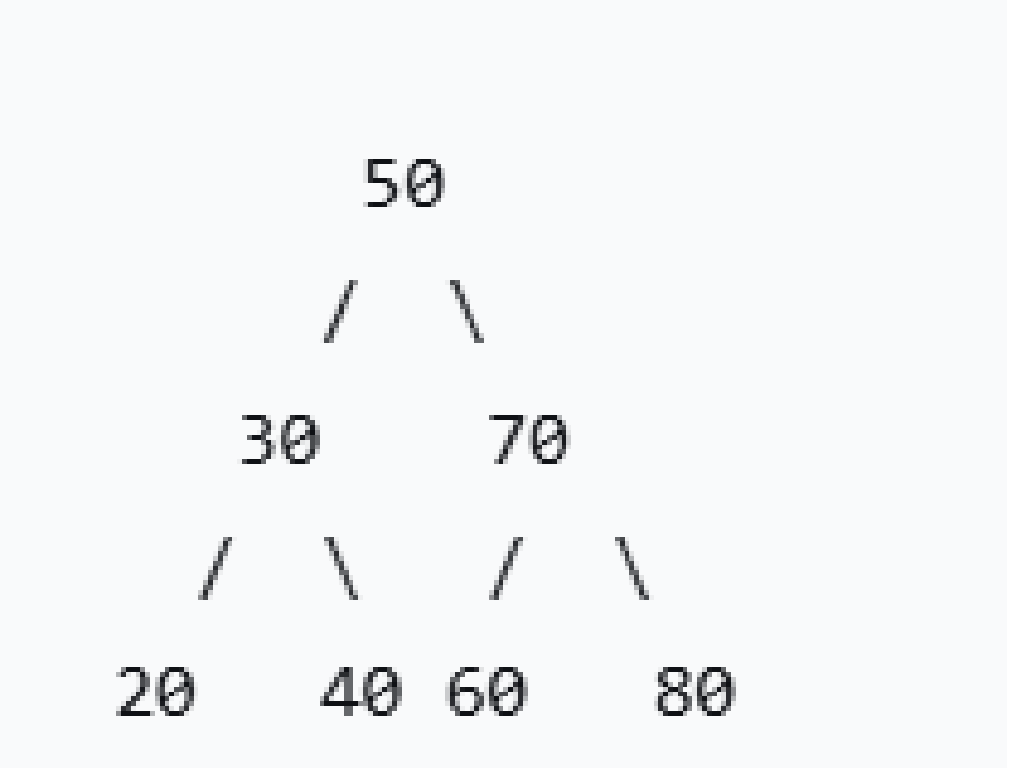
What is a Binary Search Tree?

- A Binary Search Tree is a **node-based binary tree** data structure with the following key properties:
- **Binary Tree Structure:** Each node has at most two children, typically called the **left child** and **right child**.
- **Ordering Property:** For any given node:
  - All values in its **left subtree** are **less than** the node's value.
  - All values in its **right subtree** are **greater than** the node's value.
  - This implies that all values in the tree are **unique** (or if duplicates are allowed, they must be consistently placed, e.g., always in the left or always in the right subtree).

Visual Example

Let's build a BST by inserting the numbers: 50, 30, 70, 20, 40, 60, 80.

- The root is 50.
- The entire left subtree of 50 (30, 20, 40) contains values less than 50.
- The entire right subtree of 50 (70, 60, 80) contains values greater than 50.
- This property holds recursively for every node. For example, node 30 has left child 20 (<30) and right child 40 (>30).

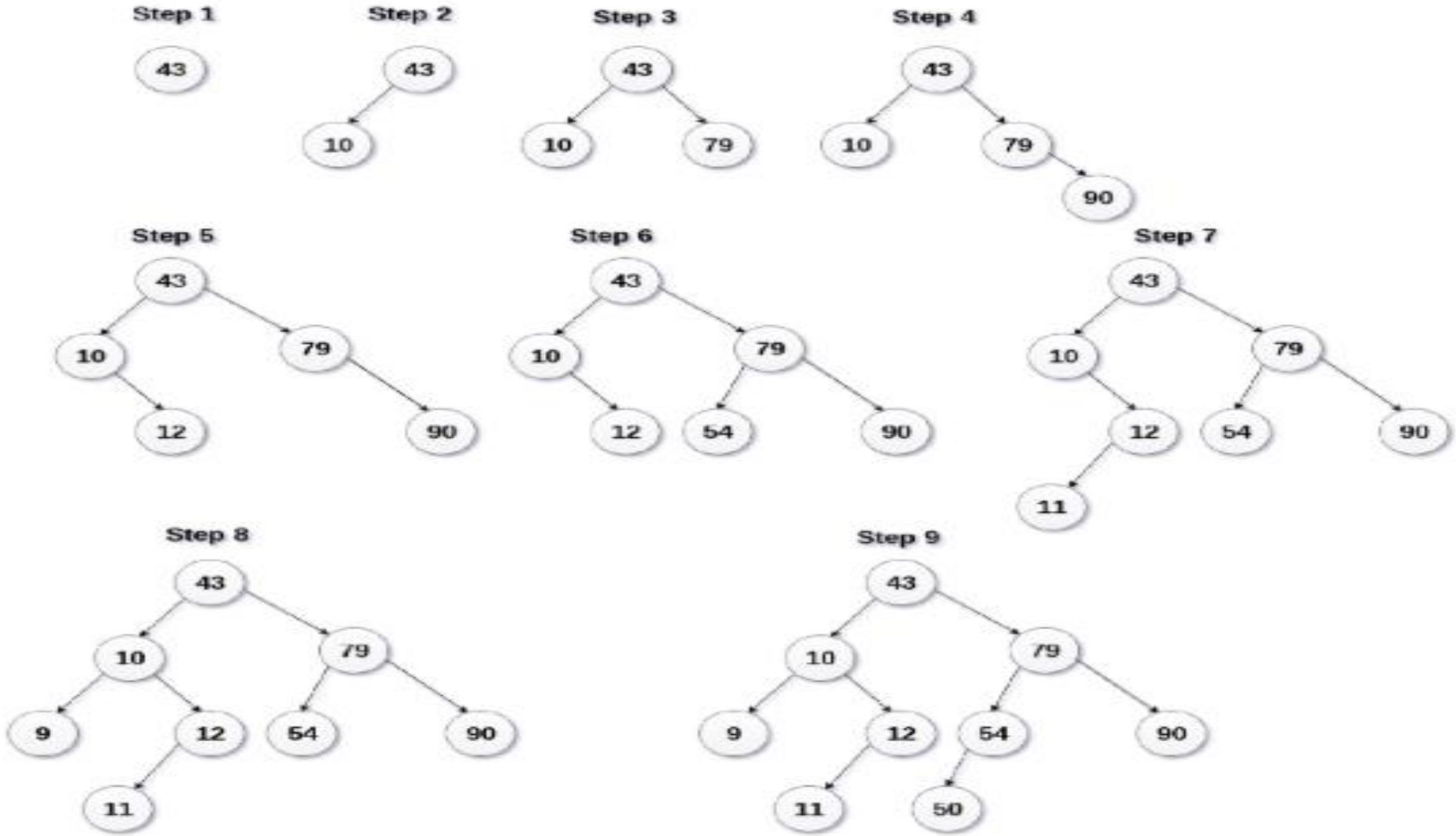


Q. Create the binary search tree using the following data elements. 43, 10, 79, 90, 12, 54, 11, 9, 50 ?

1.Insert 43 into the tree as the root of the tree.

2.Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.

3.Otherwise, insert it as the root of the right of the right sub-tree.



Binary search Tree Creation

Binary Search Tree Operations

Core Operations:

1. Search

Purpose: Find if a value exists

Approach: Start at root, go left if smaller, right if larger.

2. Insertion

Purpose: Add new value while maintaining BST properties

Approach: Find correct position (like search), insert at null location

3. Deletion - Three cases:

No children: Simply remove node

One child: Replace node with its child

Two children: Replace with inorder successor/predecessor

Traversal Operations:

4. In-order Traversal

Order: Left → Root → Right

Result: Returns elements in sorted order

5. Pre-order Traversal

Order: Root → Left → Right

6. Post-order Traversal

Order: Left → Right → Root

1. Search

The **search operation** in a BST finds whether a specific value exists in the tree by leveraging the BST property:

- Left subtree contains smaller values
- Right subtree contains larger values
- This allows us to eliminate half of the remaining tree at each step, making search very efficient.

Algorithm Steps

1. **Start** at the root node
2. **Compare** target value with current node's value:
    - If **equal** → Value found
    - If **smaller** → Go to left child
    - If **larger** → Go to right child
3. **Repeat** until value found or null node reached
4. If null reached → Value not found

```
function search(root, key):
    current = root

    while current is not NULL:
        if key == current.value:
            return true                    // Value found
        else if key < current.value:
            current = current.left         // Search left subtree
        else:
            current = current.right        // Search right subtree

    return false                           // Value not found
```

2.Insertion

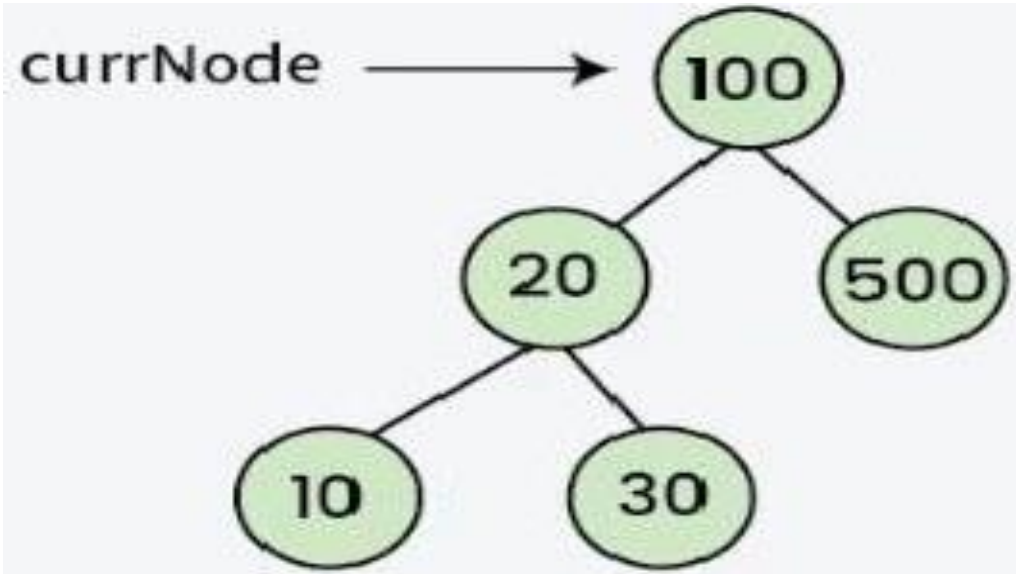
The **insert operation** adds a new value to the BST while maintaining the BST property:

- Left child < Parent < Right child
- The new node is always inserted as a **leaf node**

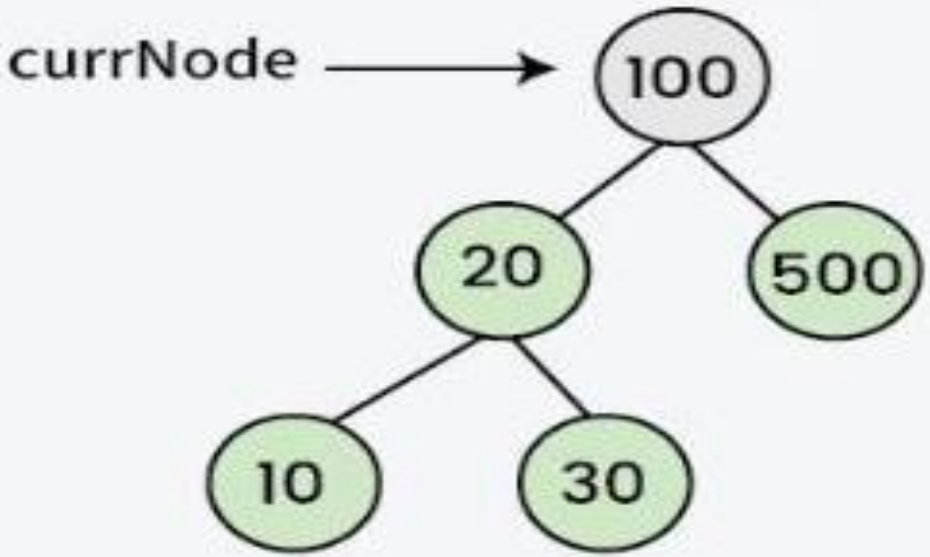
Algorithm Steps

1. **Start** at the root node
2. **Compare** new value with current node:
    - If **smaller** → Go to left child
    - If **larger** → Go to right child
3. **Repeat** until you find a null position
4. **Insert** new node at that null position

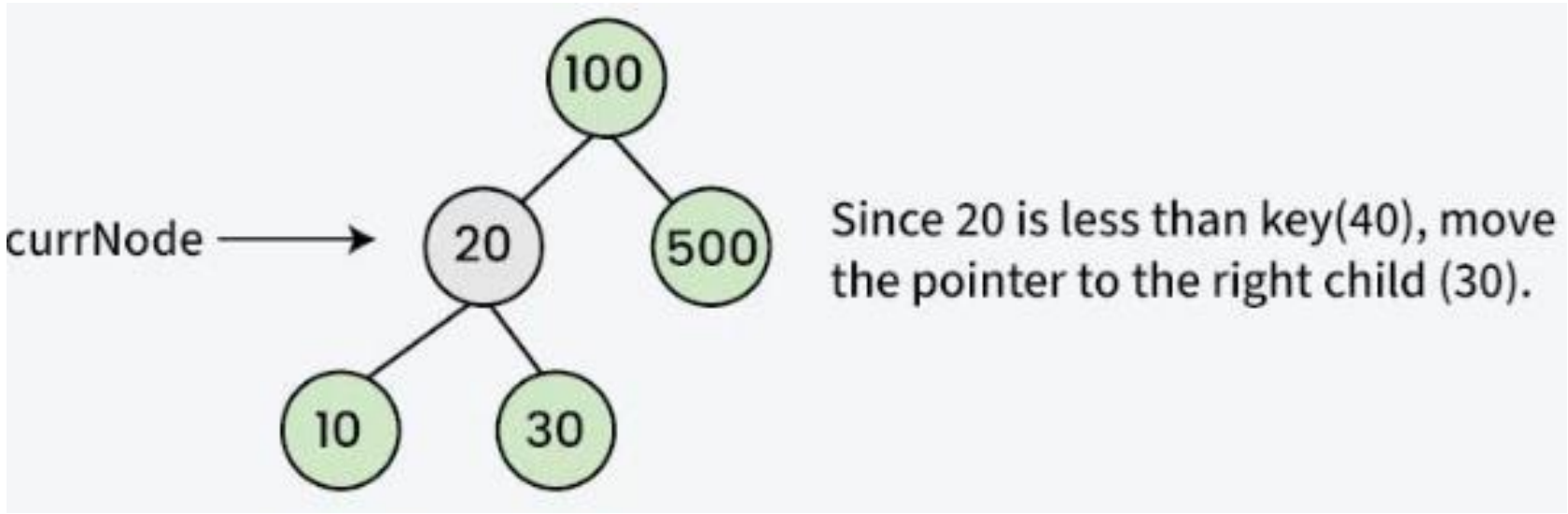
```
FUNCTION insert(root, value):
    IF root == NULL:
        root = NEW_NODE(value)
        RETURN root

    IF value < root.data:
        root.left = insert(root.left, value)
    ELSE IF value > root.data:
        root.right = insert(root.right, value)

    RETURN root
```

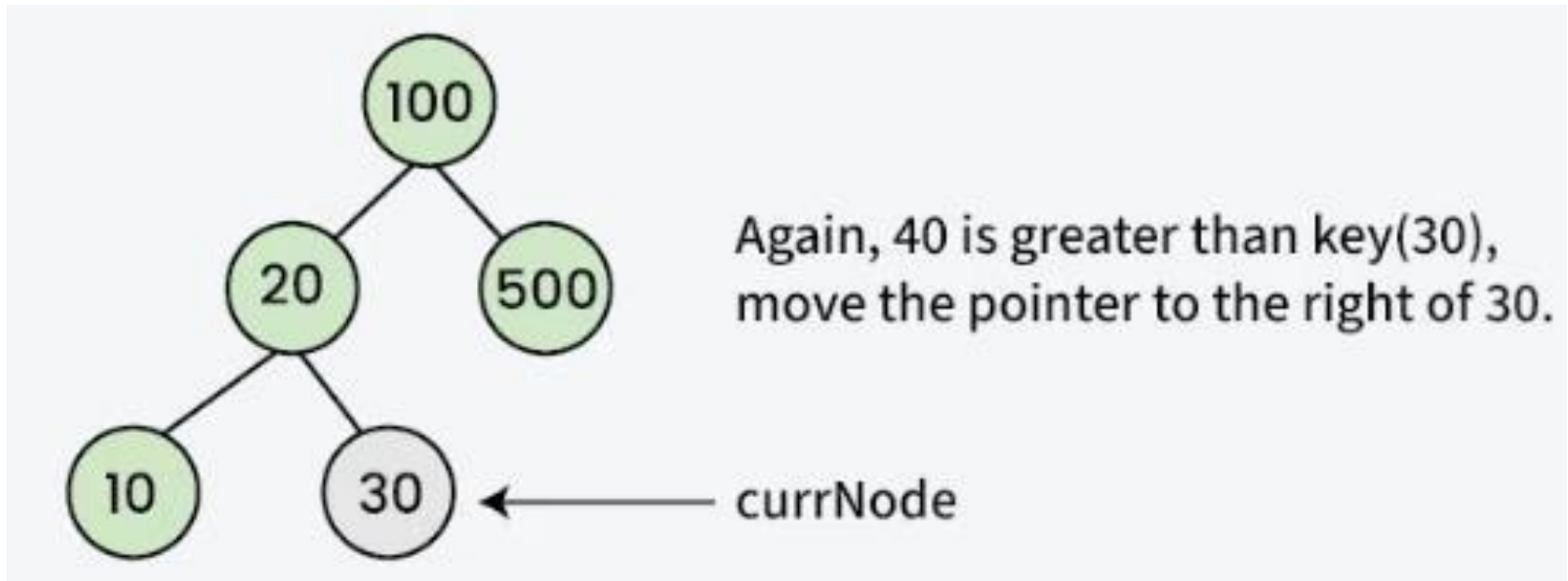


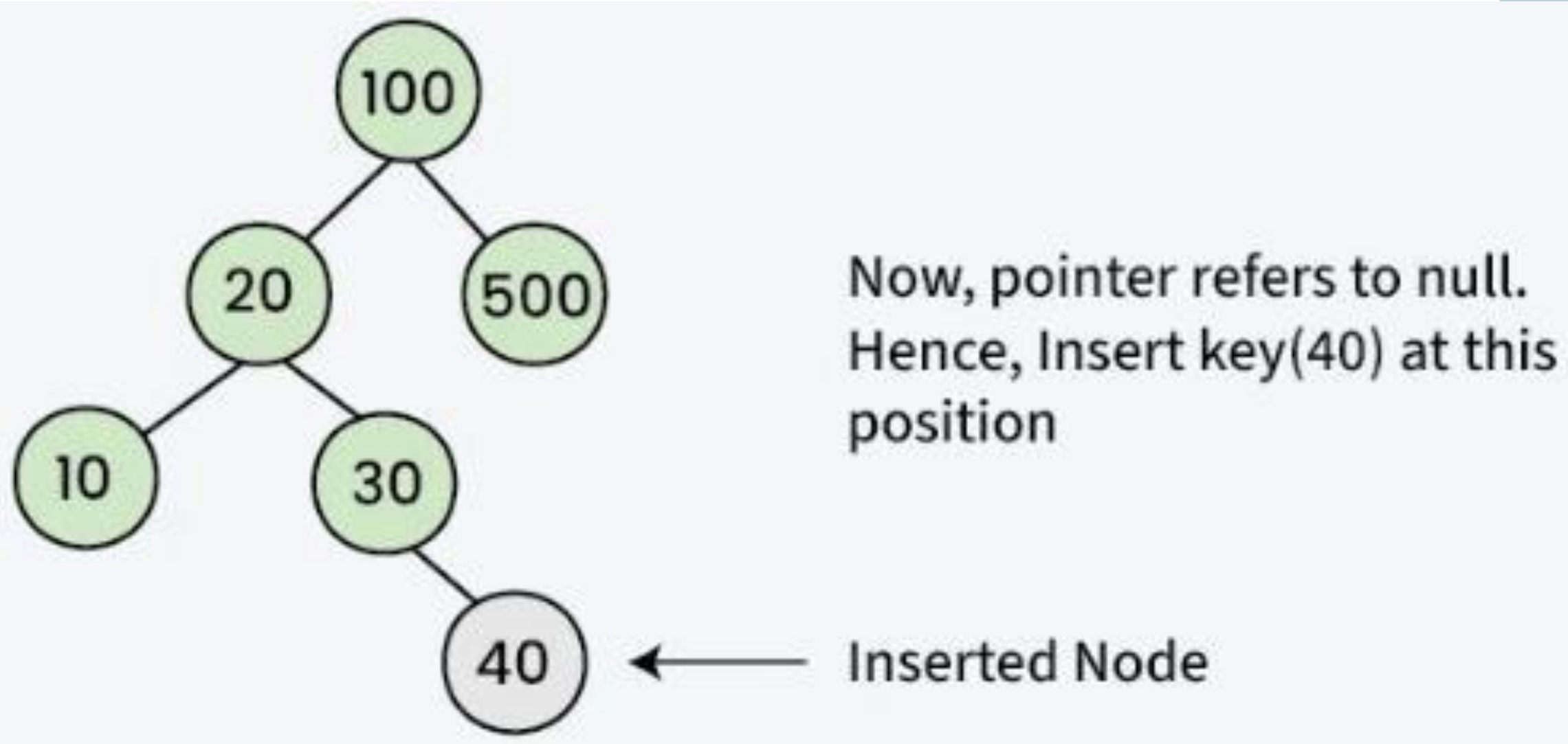
key = 40 (the node to be inserted)



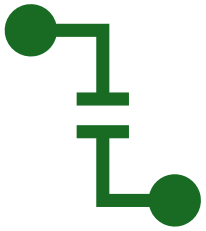
Since 100 is greater than key(40), move the pointer to the left child (20).


100
currNode
20
500
10
30
Since 20 is less than key(40), move the pointer to the right child (30).

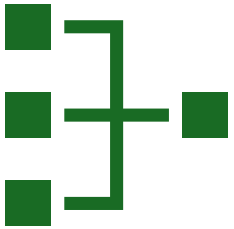

100
20
500
10
30
currNode
Again, 40 is greater than key(30), move the pointer to the right of 30.


100
20
500
10
30
40
Now, pointer refers to null.
Hence, Insert key(40) at this position
Inserted Node

4. Traversing a BST



Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way.



There are four different algorithms for tree traversals, which differ in the order in which the nodes are visited:

Pre-order

In-order

Post-order

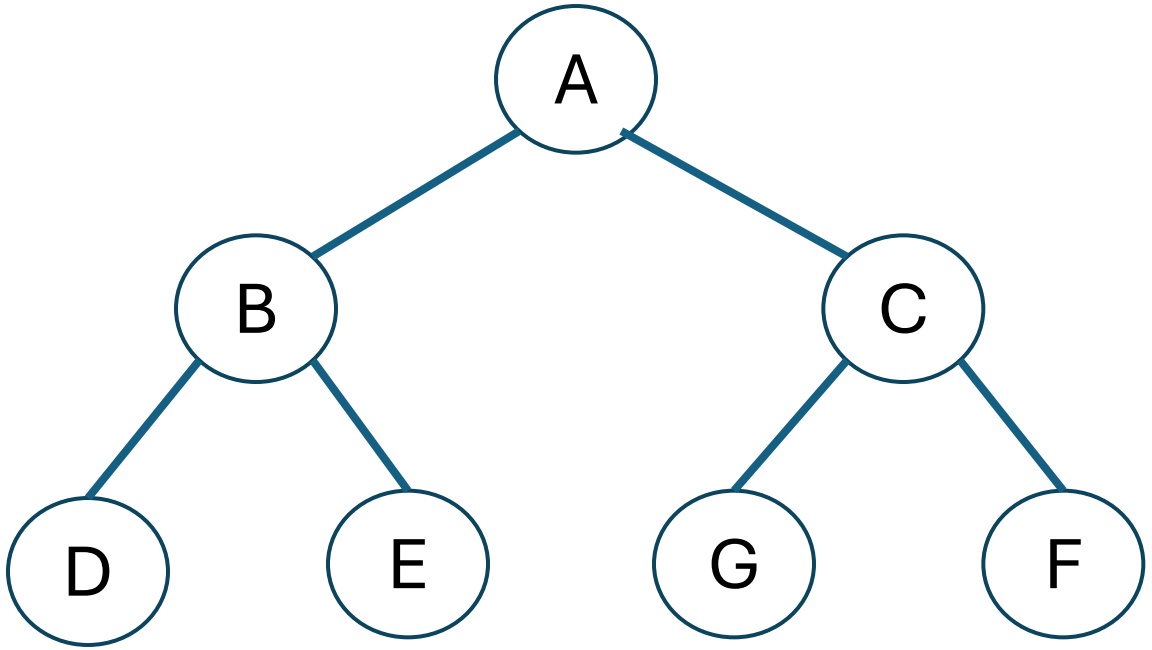
Pre-order

- To traverse a non-empty binary tree in preorder, the following operations are performed recursively at each node.

- The algorithm starts with the root node of the tree and continues by:
  - Visiting the root node
  - Traversing the left subtree
  - Traversing the right subtree

- Implemented by recursive function

Pre-order



Pre-order A, B, D, E, C, G, F

Pre-order Implementation

- Implement by recursive function

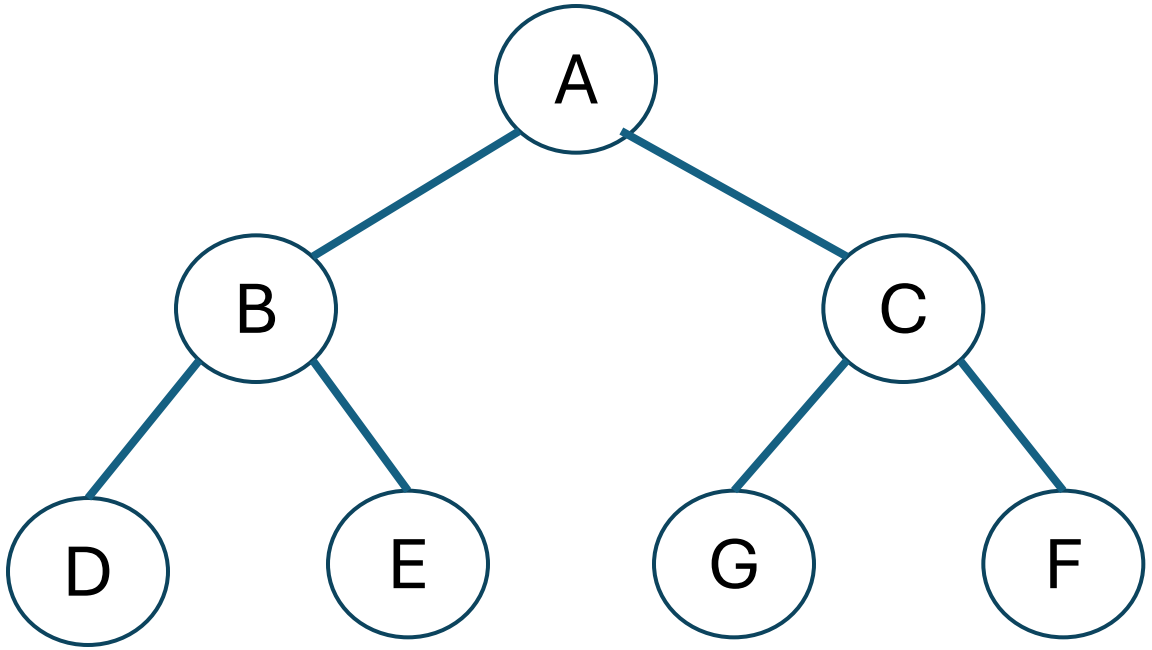
Example

```
void  print_preorder(node *root) {
        if (root) {
                printf("%d ", root->data);
                print_preorder(root->left);
                print_preorder(root->right);
        }
}
```

In-order

- To traverse a non-empty binary tree in **in-order**, the following operations are performed recursively at each node.
- The algorithm starts with the root node of the tree and continues by,
  - Traversing the left subtree
  - Visiting the root node
  - Traversing the right subtree

In-order



In-order: D, B, E, A, G, C, F

In-order Implementation

- Implement by recursive function

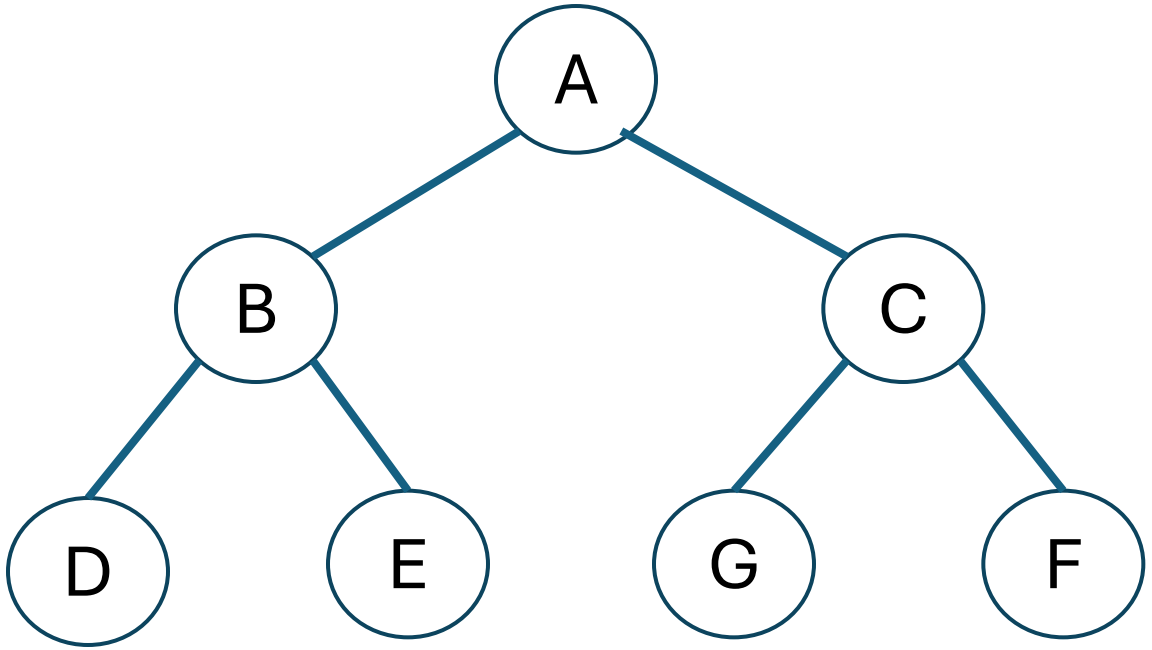
Example

```
void  print_inorder(node *root) {
        if (root) {
                print_inorder(root->left);
                printf("%d ", root->data);
                print_inorder(root->right);
        }
}
```

Post-order

- To traverse a non-empty binary tree in **post-order**, the following operations are performed recursively at each node.
- The algorithm starts with the root node of the tree and continues by,
  - Traversing the left subtree
  - Traversing the right subtree
  - Visiting the root node
- Implemented by recursive function

Post-order



Post-order: D, E, B, G, F, C, A

post-order Implementation

- Implement by recursive function

Example

```
void  print_postorder(node *root) {
        if (root) {
                print_postorder(root->left);
                print_postorder(root->right);
                printf("%d ", root->data);
        }
}
```