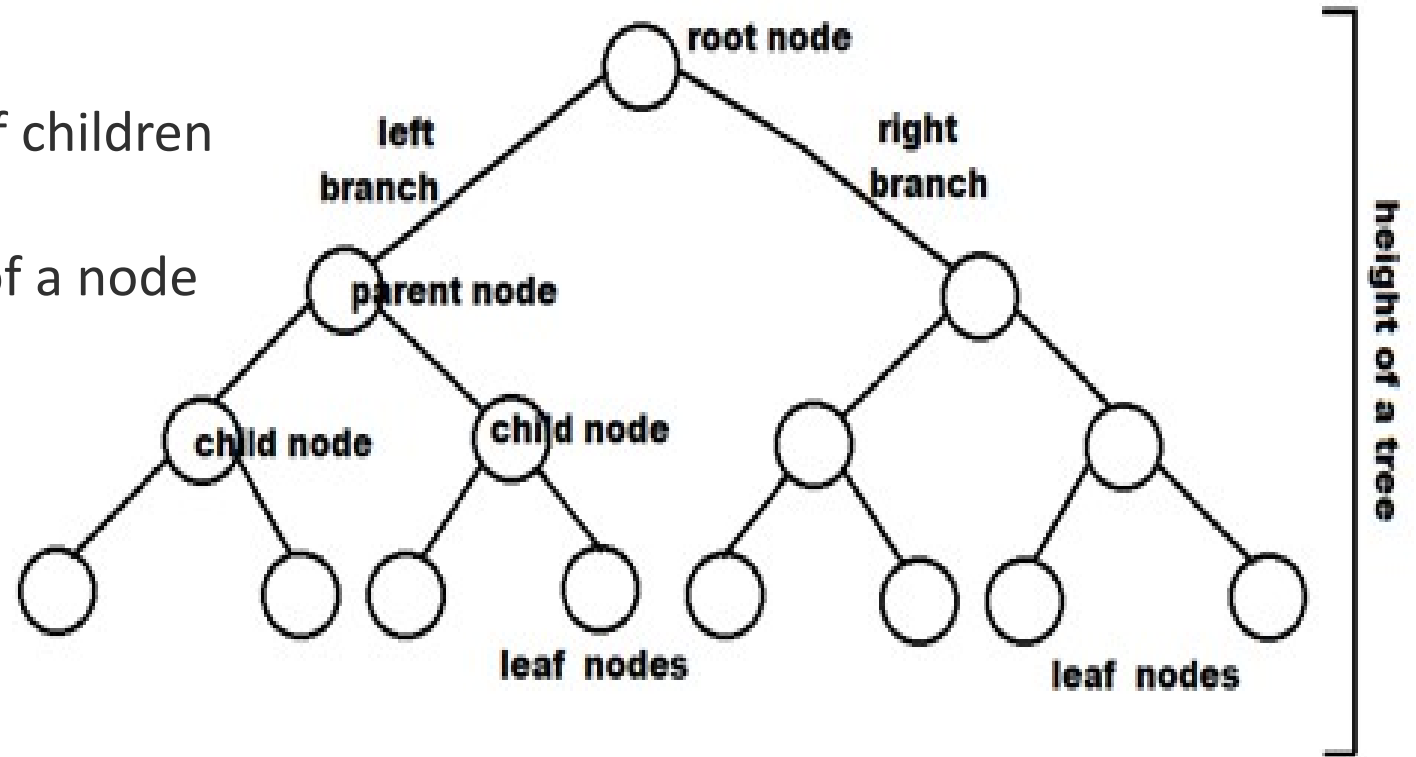


# Binary Tree and Binary Search Tree

# Tree

- ❑ Trees are **Non-Linear** and **Abstract Data Structures** that simulate a hierarchical structure.
- ❑ There is one and only one path between every pair of vertices in a tree.
- ❑ There is  **$(n-1)$**  edges in a tree with  **$n$**  vertices.
  - ❑ Any connected graph with  $n$  vertices and  $(n-1)$  edges is a tree.

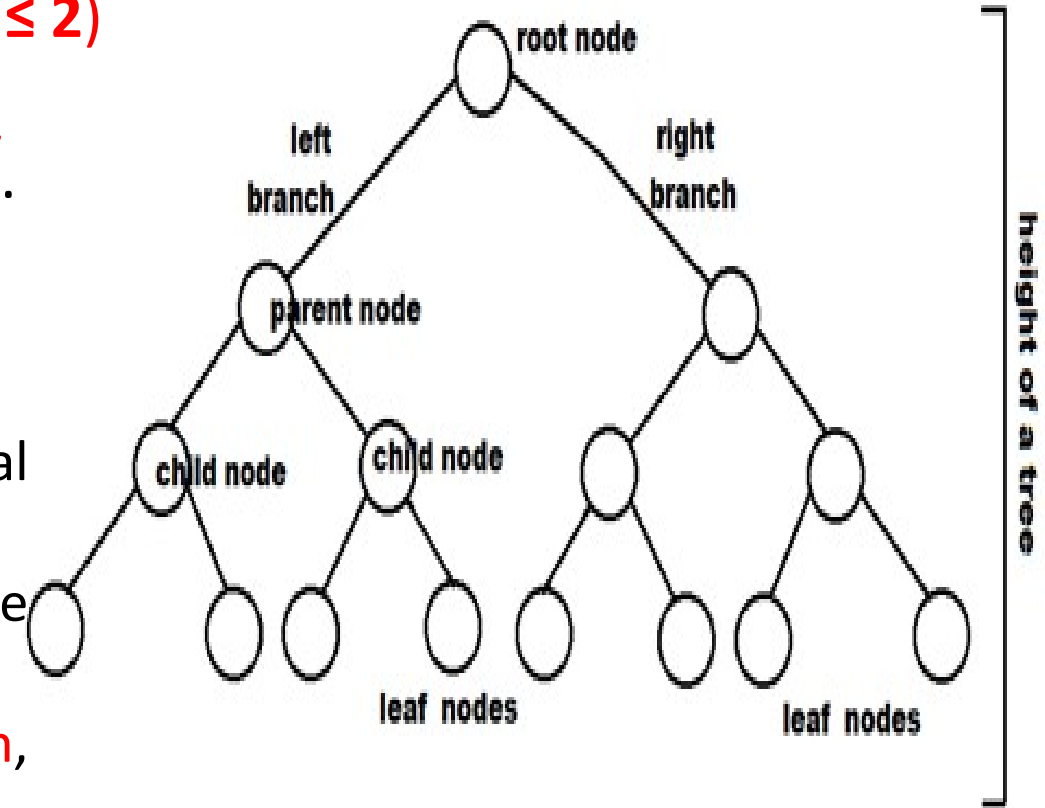
- ❑ **Degree of a node** is the total number of children of that node.
- ❑ **Degree of a tree** is the highest degree of a node among all the nodes in the tree.
- ❑ Height of a node
- ❑ Height of the Tree
- ❑ Depth of a node
- ❑ Forest



# Binary Tree

- ❑ A binary tree is a data structure composed of nodes, where each node contains a **value** and **two references to other nodes**, called the left child and the right child.
- ❑ In a binary tree each node can have  $n$  children ( $0 \leq n \leq 2$ )

- At each level of  $i$ , the maximum number of nodes is  $2^i$ .
- If height = 4, the maximum number of nodes  $(1+2+4+8+16) = 31$ 
  - at height  $h$  is  $2^0 + 2^1 + 2^2 + \dots + 2^h = 2^{h+1} - 1$ .
- The minimum number of nodes possible at height  $h$  is equal to  $h+1$ .
- If the number of nodes is **minimum**, then the height of the tree would be **maximum**.
- On the other hand, if the number of nodes is **maximum**, then the height of the tree would be **minimum**.



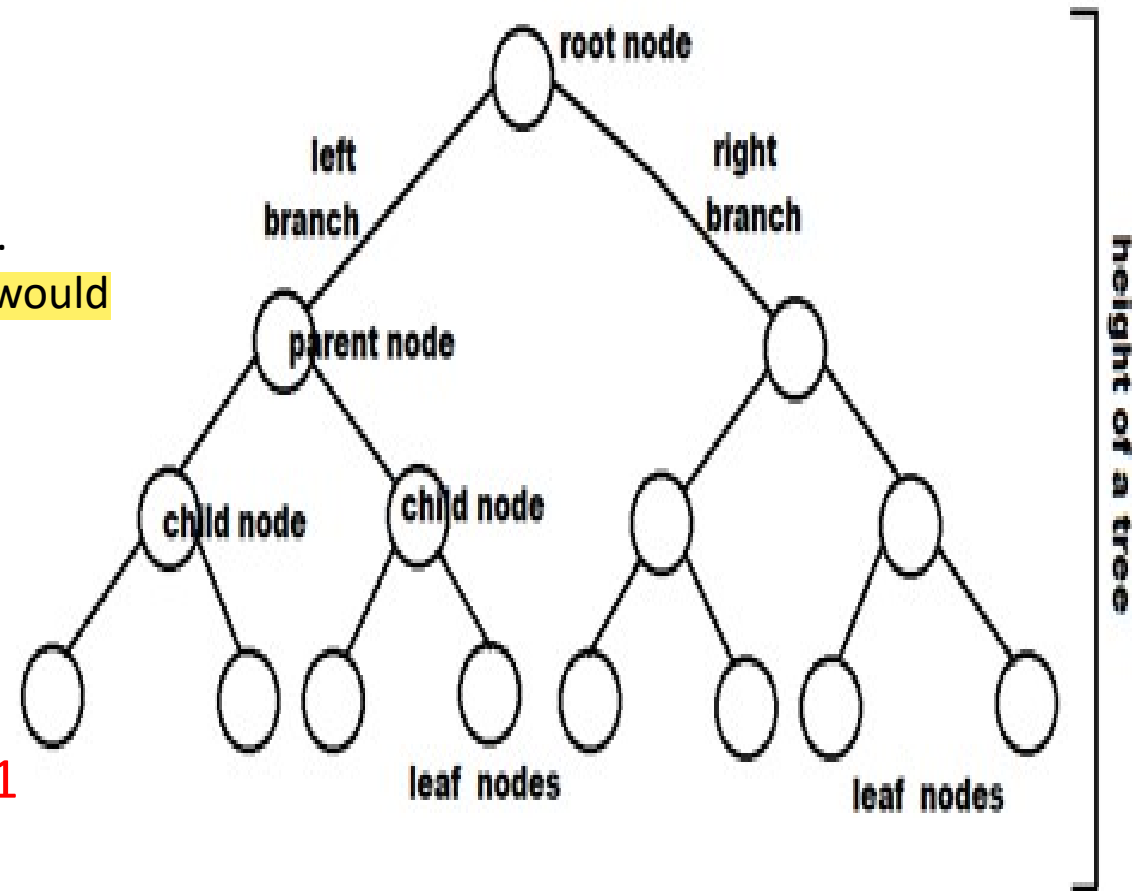
# Binary Tree

- The minimum number of nodes possible at height  $h$  is equal to  $h+1$ .
- If the number of nodes is minimum, then the height of the tree would be maximum.
  - Let's  $n$  is number of nodes in the binary tree.
  - Then,  $n = h+1$ ; meaning  $h = n-1$
- At each level of  $i$ , the maximum number of nodes is  $2^i$ .
- If height = 4, the maximum number of nodes  $(1+2+4+8+16) = 31$ 
  - at height  $h$  is  $2^0 + 2^1 + 2^2 + \dots + 2^h = 2^{h+1} - 1$ .
- if the number of nodes is maximum, then the height of the tree would be minimum.

Let's  $n$  is number of nodes in the binary tree.

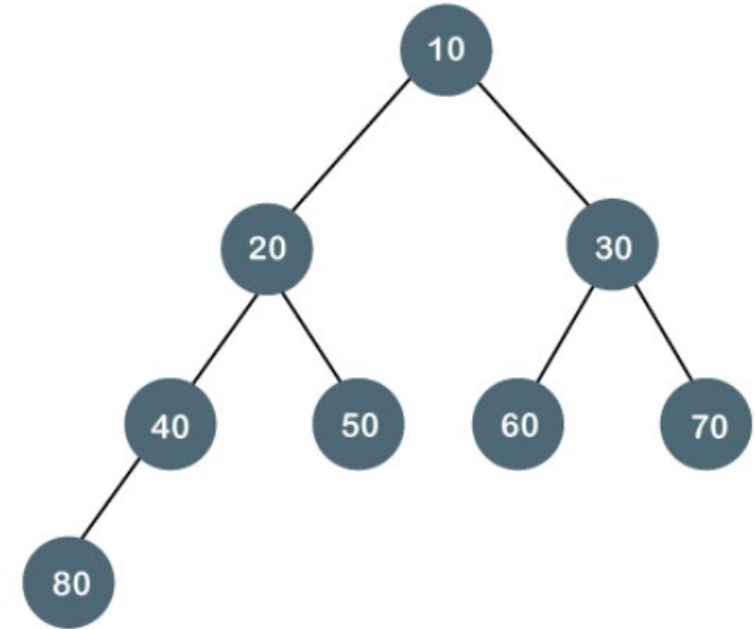
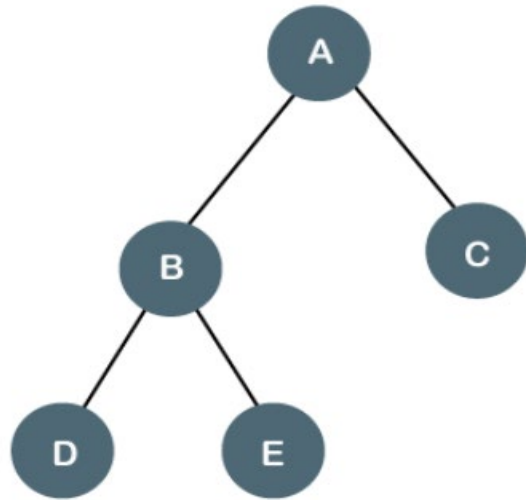
i.e.  $n = 2^{h+1} - 1$ ;

hence, minimum height,  $h = \log_2(n+1) - 1$



# Types of Binary tree

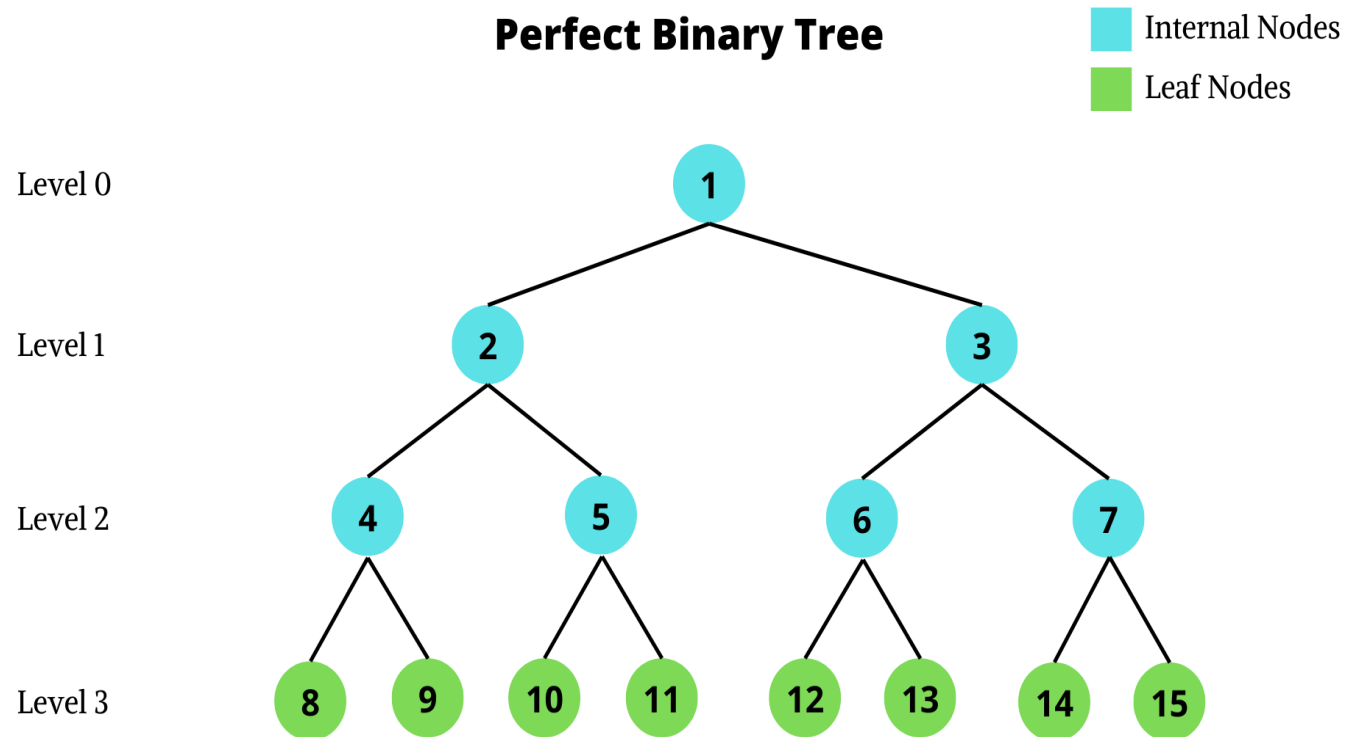
- ❑ Full binary tree: A full binary tree is a binary tree in which every node has **either two children or no children**.
  - ❑ each node must contain 2 children except the leaf nodes.



- ❑ Complete binary tree: A complete binary tree is a binary tree in which all levels except the last are completely filled
  - ❑ All of the nodes must be as far to the **left as feasible** in the **last level**. The nodes should be added from the left

# Types of Binary tree

**Perfect binary tree:** A perfect binary tree is a binary tree in which all **internal** nodes have two children and all **leaf** nodes are at the same level.

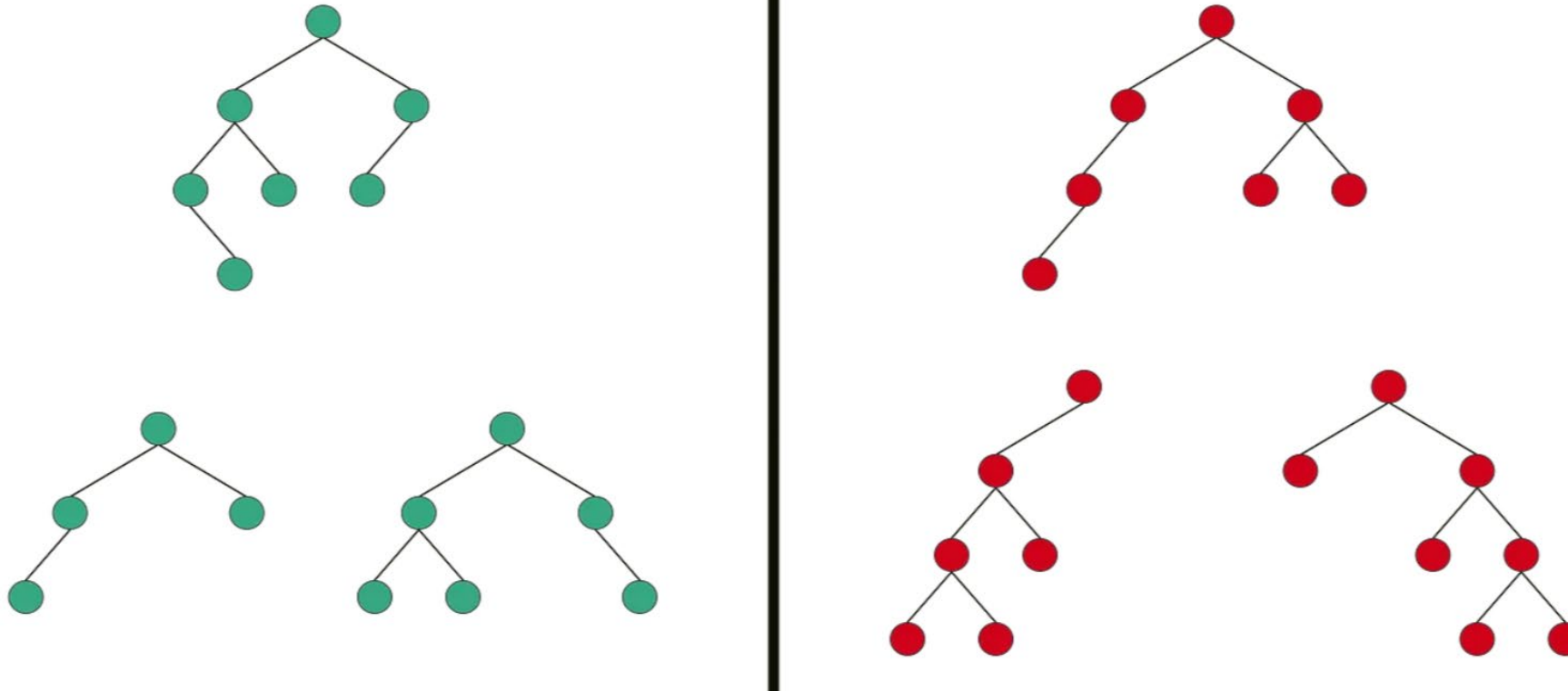


# Types of Binary tree

**Balanced binary tree:** A balanced binary tree is a binary tree in which the heights of the **left and right subtrees of every node** differ by at most one.

Some operations in a an unbalanced tree may take longer than  $O(\log n)$  time

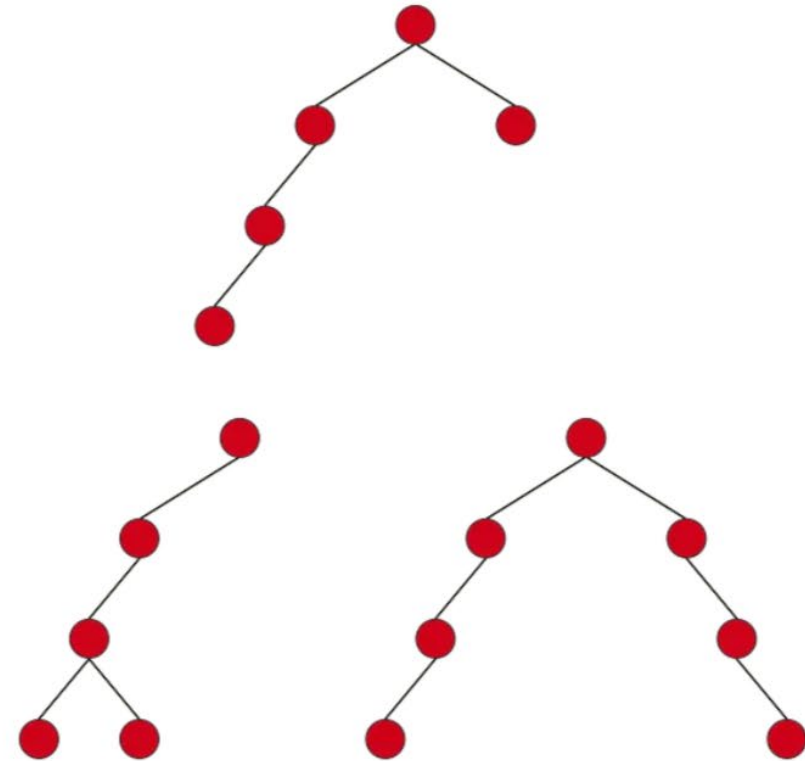
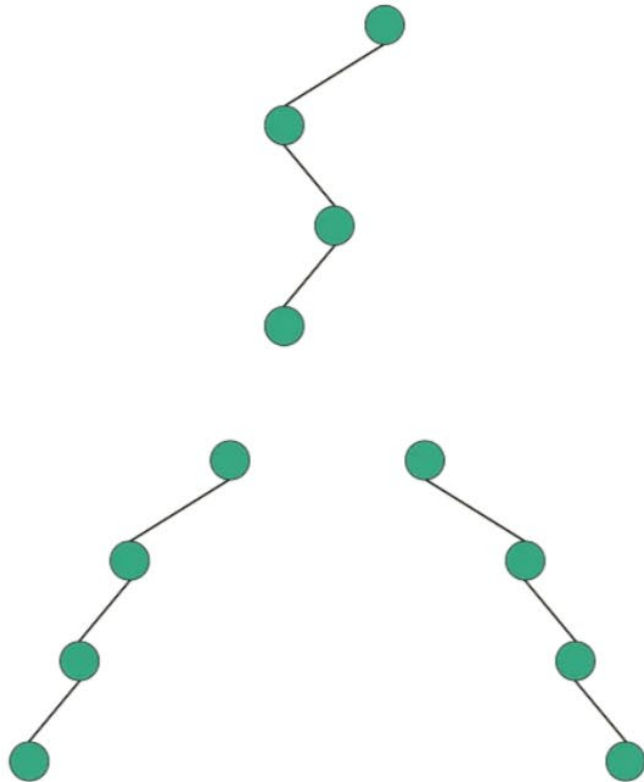
***AVL Tree and Red-Black Tree are well-known data structure to generate/maintain Balanced Binary Search Tree.***



# Types of Binary tree

**Degenerate (or pathological) tree:** A degenerate (or pathological) tree is a tree in which each parent node has only one associated child node.

*Height of a Degenerate Binary Tree is equal to Total number of nodes in that tree.*

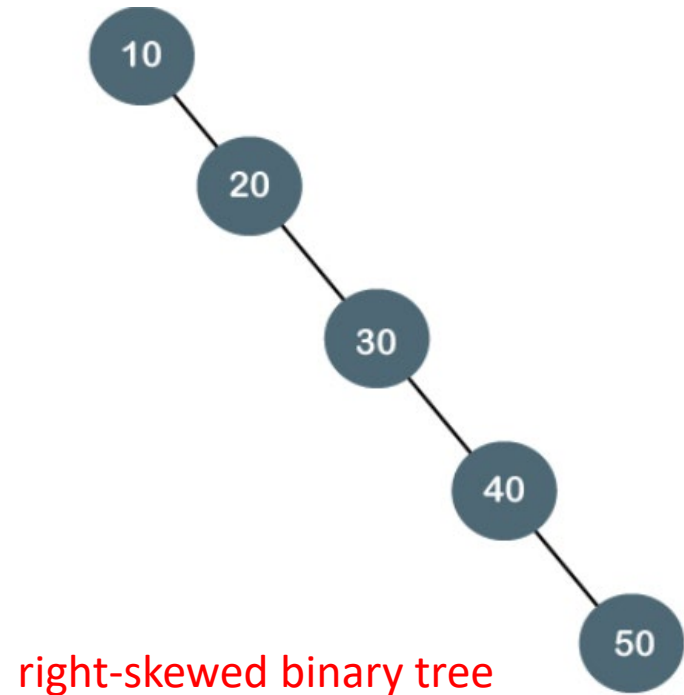
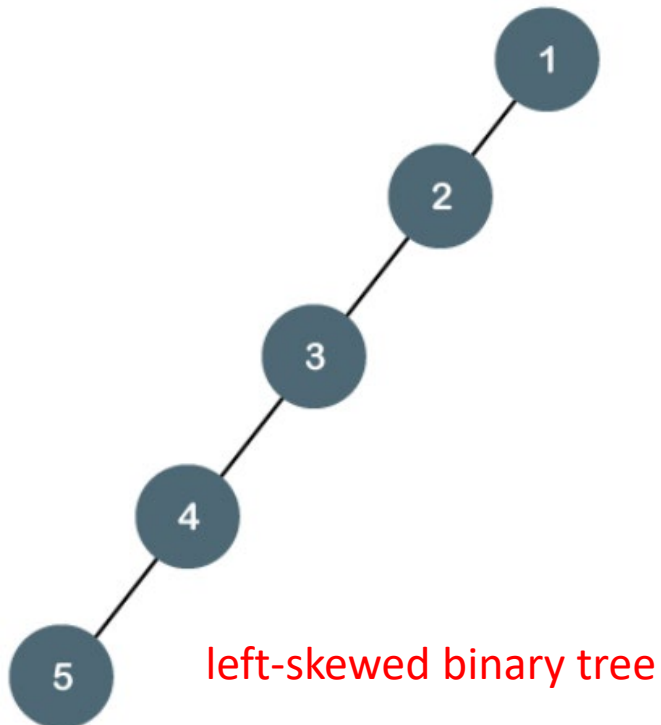




# Types of Binary tree

## Skewed binary tree

- In a skewed binary tree is a special type of Degenerate (or pathological) tree,
- Here, all the nodes are either left-skewed or right-skewed.
- A **left-skewed binary tree** is a tree in which each node has at most one right child.
- A right-skewed binary tree is a tree in which each node has at most one left child.

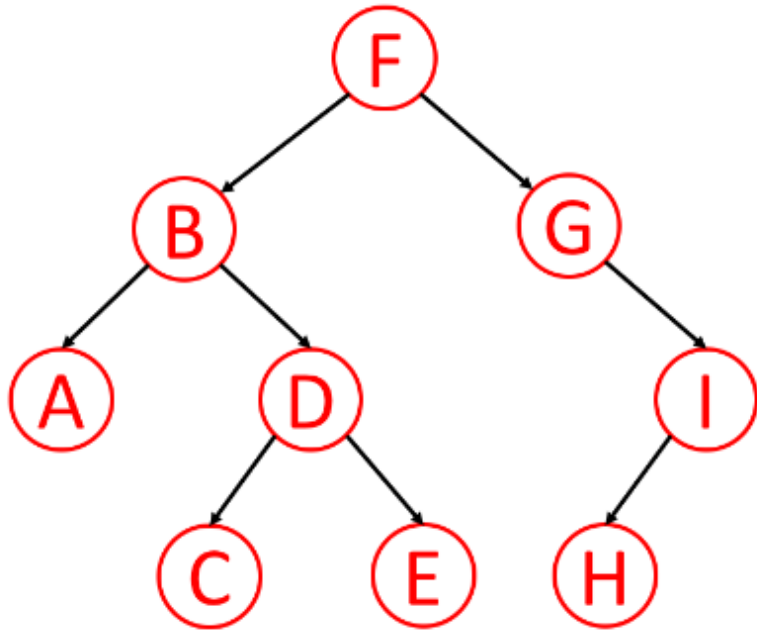


# Binary Search Tree Traversal

- ❑ Traverse the BST to visit all the nodes in the tree in a specific order. There are three main types of traversal: **in-order, pre-order, and post-order**.
- ❑ In-order traversal visits the nodes in ascending order of their keys
- ❑ pre-order traversal visits the current node before its children
- ❑ post-order traversal visits the current node after its children
- ❑ Traversal can be implemented using recursion or an iterative algorithm, and the time complexity is  **$O(n)$** , where  $n$  is the number of nodes in the tree.

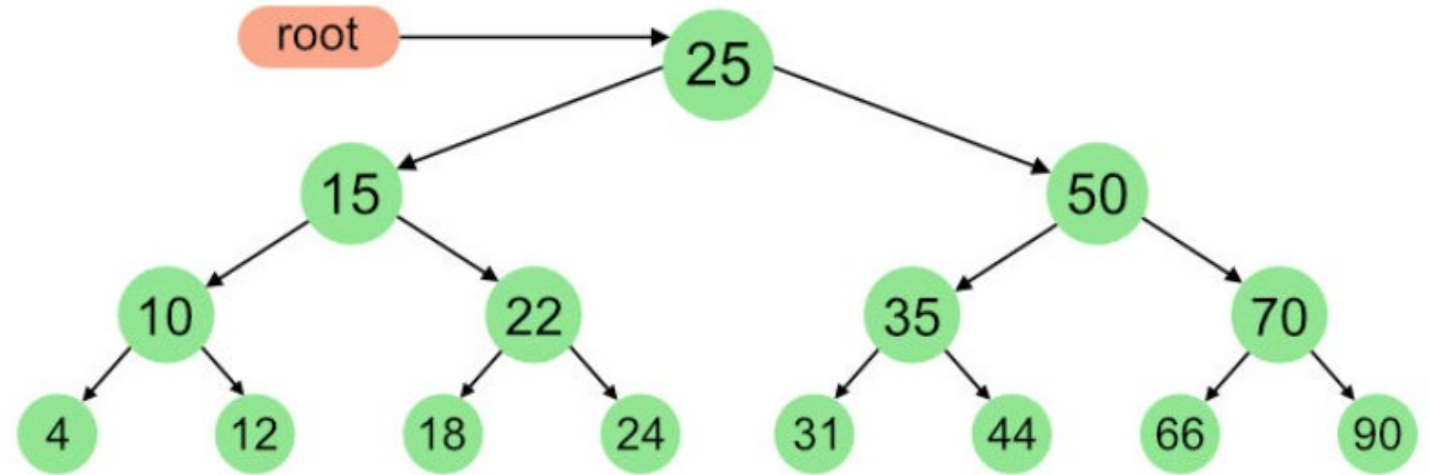
# Binary Tree: Pre-order Traversal

❑ Pre-order traversal is to visit the **root** first. Then traverse the **left subtree**. Finally, traverse the **right subtree**.



Preorder:

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| F | B | A | D | C | E | G | I | H |
|---|---|---|---|---|---|---|---|---|

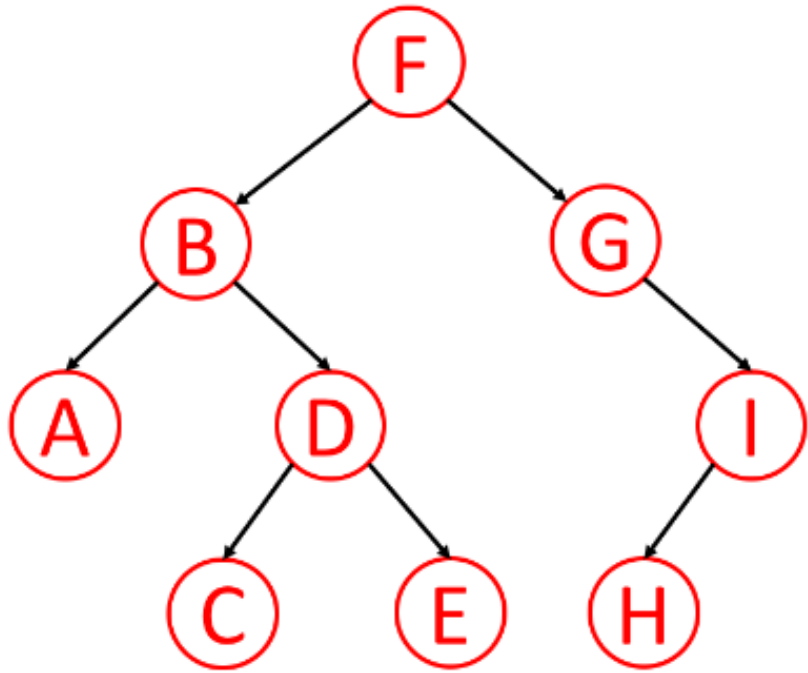


25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

```
def preorder(self):  
    print(self.val)  
    if self.left is not None:  
        self.left.preorder()  
    if self.right is not None:  
        self.right.preorder()
```

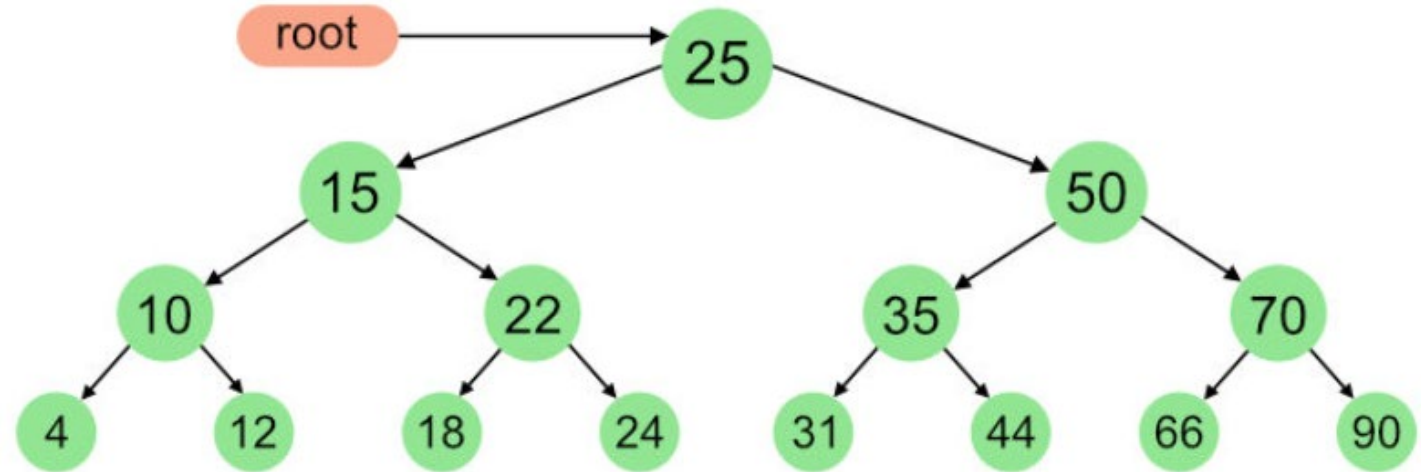
# Binary Tree: In-order Traversal

❑ In-order traversal is to traverse the **left subtree first**. Then visit the **root**. Finally traverse the **right subtree**.



Inorder:

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|

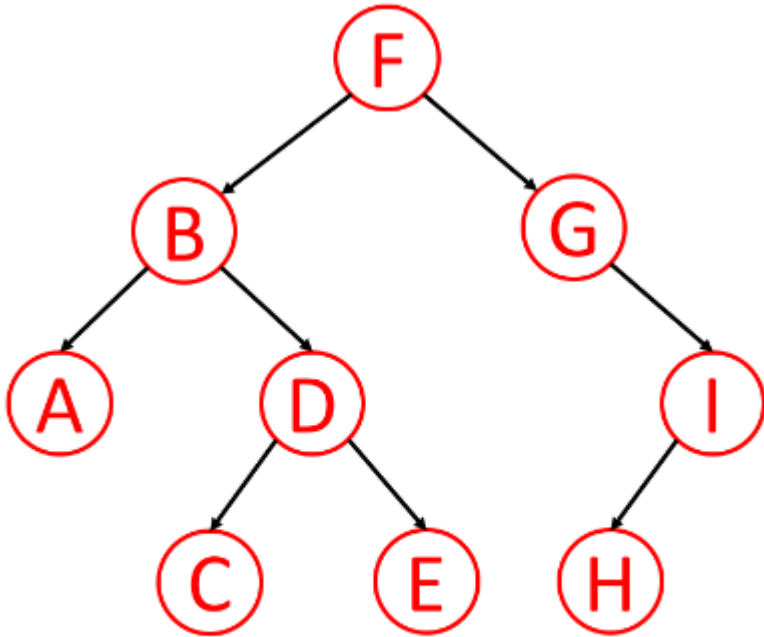


4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

```
def inorder(self):  
    if self.left is not None:  
        self.left.inorder()  
    print(self.val)  
    if self.right is not None:  
        self.right.inorder()
```

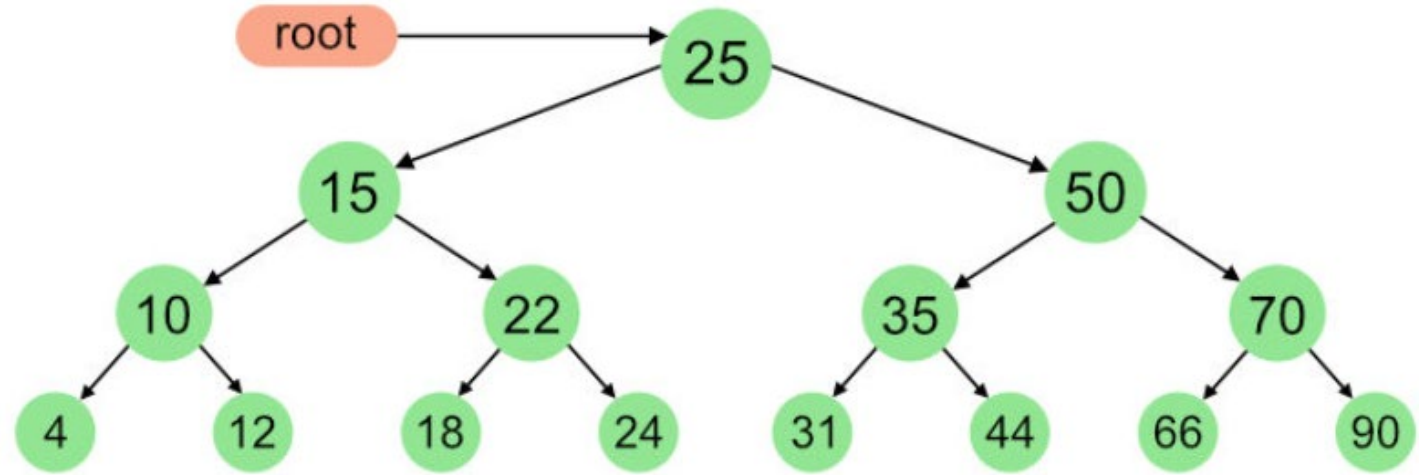
# Binary Tree: Post-order Traversal

❑ Post-order traversal is to traverse the **left subtree first**. Then traverse the **right subtree**. Finally, visit the **root**.



Postorder:

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | C | E | D | B | H | I | G | F |
|---|---|---|---|---|---|---|---|---|



4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

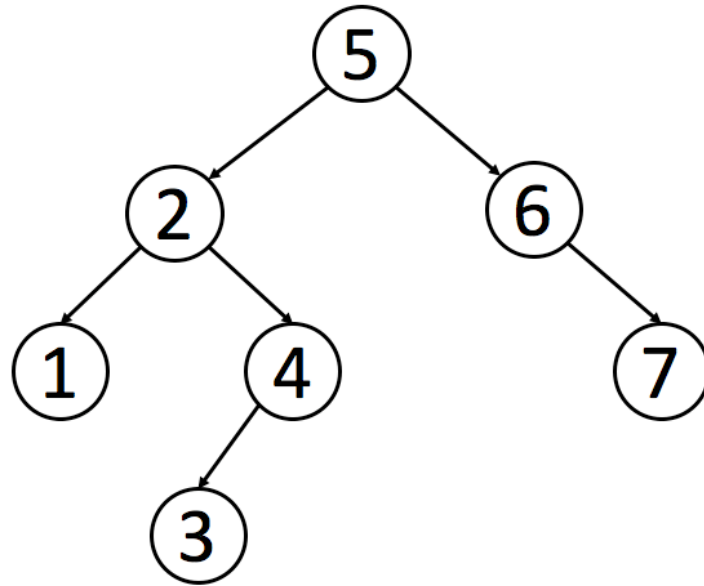
```
def postorder(self):  
    if self.left is not None:  
        self.left.postorder()  
    if self.right is not None:  
        self.right.postorder()  
    print(self.val)
```

# Traversal code

```
def inorder(self):  
    if self.left is not None:  
        self.left.inorder()  
    print(self.val)  
    if self.right is not None:  
        self.right.inorder()  
  
def preorder(self):  
    print(self.val)  
    if self.left is not None:  
        self.left.preorder()  
    if self.right is not None:  
        self.right.preorder()  
  
def postorder(self):  
    if self.left is not None:  
        self.left.postorder()  
    if self.right is not None:  
        self.right.postorder()  
    print(self.val)
```

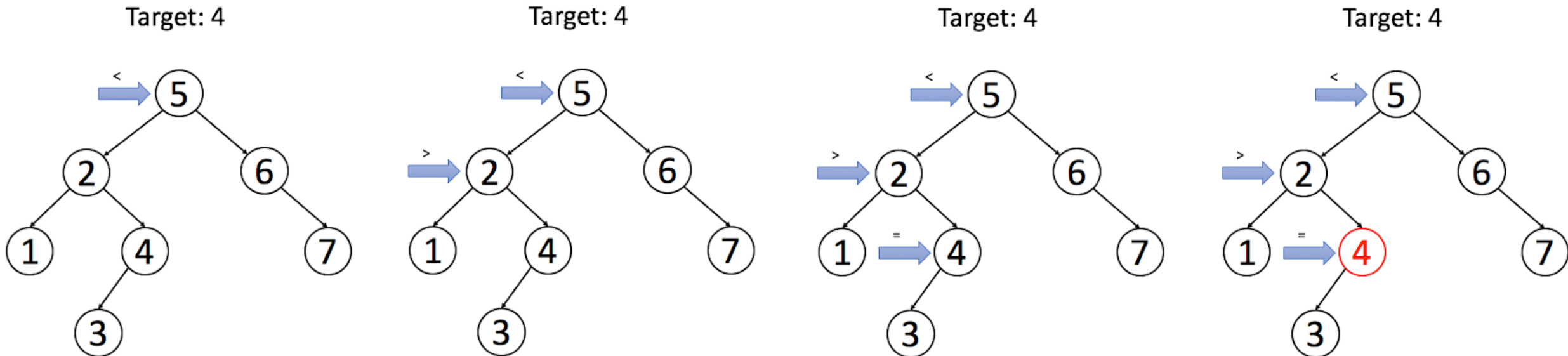
# Binary Search Tree

- ❑ A Binary Search Tree is a special form of a binary tree.
- ❑ The value in each node must be **greater than (or equal to)** any values in its **left subtree**, but **less than (or equal to)** any values in its **right subtree**.



# Binary Search Tree – Search Operation

- ❑ return the node if the target value is **equal** to the value of the node;
- ❑ continue searching in the **left subtree** if the target value is less than the value of the node;
- ❑ continue searching in the **right subtree** if the target value is larger than the value of the node.
- ❑ The time complexity of the search operation in a **balanced BST** is  $O(\log n)$ , where  $n$  is the number of nodes in the tree.





# Binary Search Tree – Search Operation

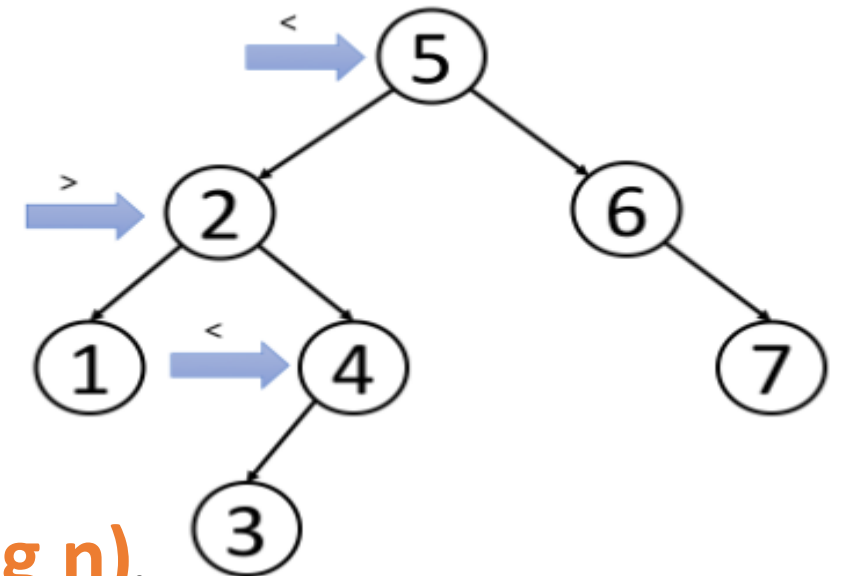
```
def search(self, val):  
    return self._search(val, self.root)  
  
def _search(self, val, node):  
    if not node:  
        return False  
    elif node.val == val:  
        return True  
    elif val < node.val:  
        return self._search(val, node.left)  
    else:  
        return self._search(val, node.right)
```

# Binary Search Tree – Insertion Operation

□ Similar to the search strategy, for each node, we will:

1. search the **left or right subtrees** according to the relation of the value of the node and the value of our target node;
2. repeat STEP 1 until reaching an external node;
3. add the new node as its left or right child depending on the relation of the value of the node and the value of our target node.

Input Array: [5, 2, 6, 1, 7, 4, 3]



**Time complexity** of the insertion operation in a balanced BST is also  **$O(\log n)$** .

# Binary Search Tree – Insertion Operation

```
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
```

```
class BinarySearchTree:
    def __init__(self):
        self.root = None

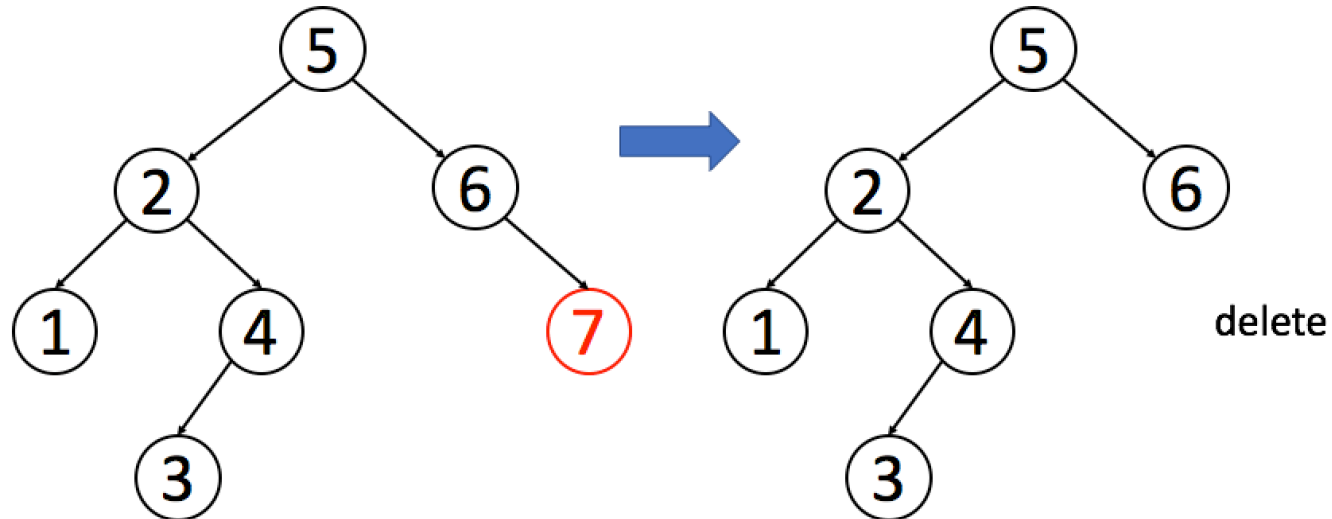
    def insert(self, val):
        if not self.root:
            self.root = TreeNode(val)
        else:
            self._insert(val, self.root)

    def _insert(self, val, node):
        if val < node.val:
            if node.left:
                self._insert(val, node.left)
            else:
                node.left = TreeNode(val)
        else:
            if node.right:
                self._insert(val, node.right)
            else:
                node.right = TreeNode(val)
```

# Binary Search Tree – Deletion Operation

- ❑ Deletion is more complicated than the two operations we mentioned before.
- ❑ There are also many different strategies for deletion.
- If the target node has **no child**, we can simply **remove** the node

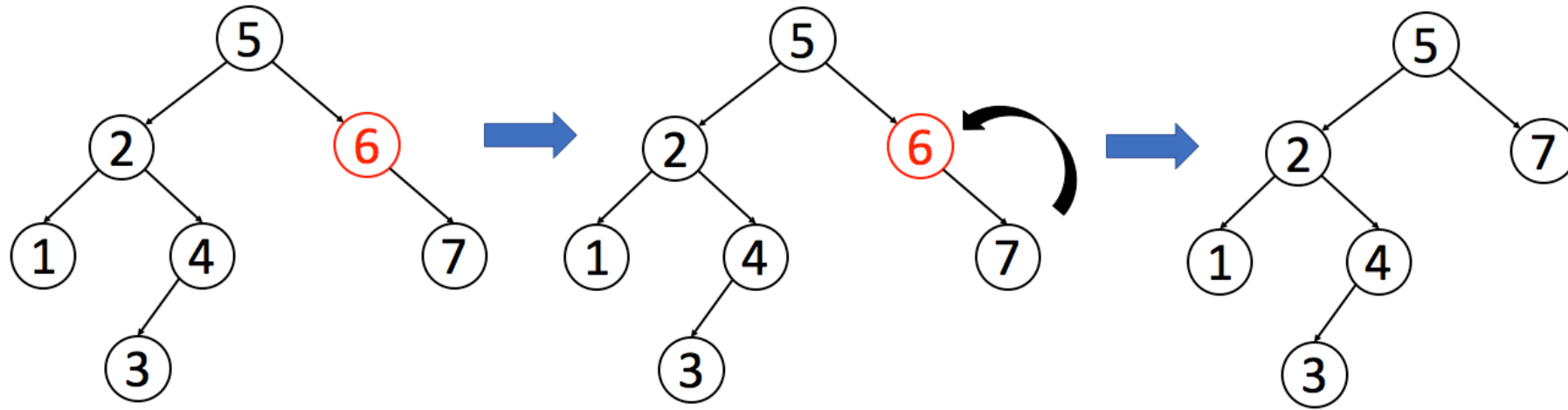
Case 1: No Child



# Binary Search Tree – Deletion Operation

- If the target node has ***no child***, we can simply remove the node
- If the target node has ***one child***, we can use its child to replace itself

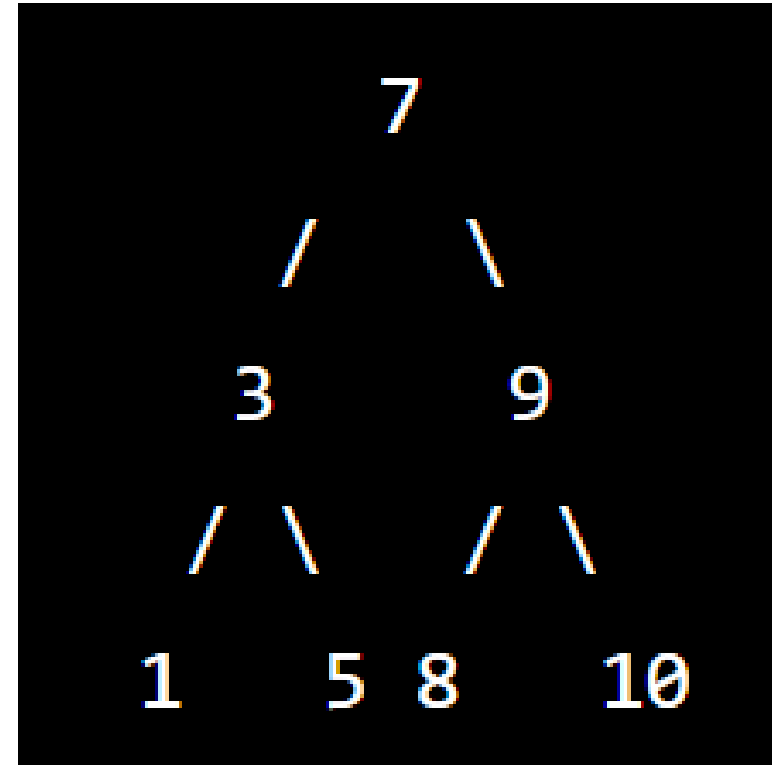
Case 2: One Child



# Binary Search Tree – Deletion Operation

- If the target node has **two children**, replace the node with its **in-order successor** or **predecessor** node and delete that node.

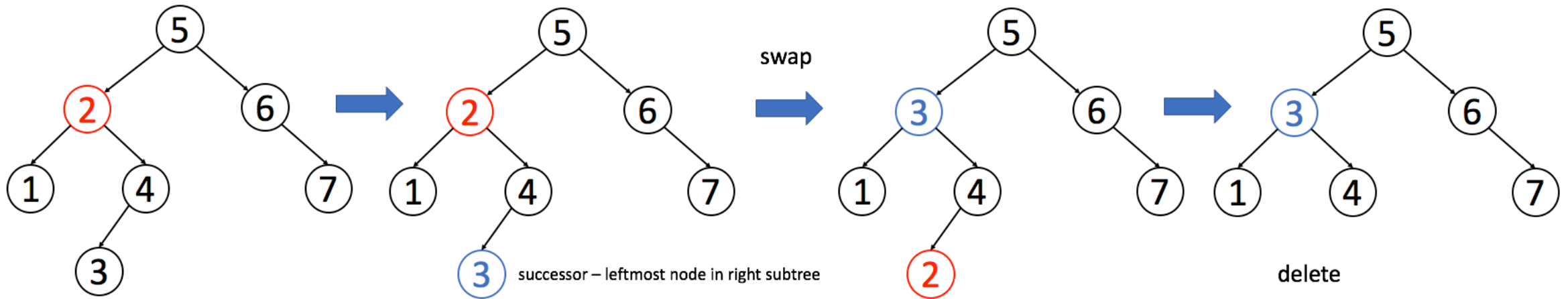
Inorder: 1, 3, 5, 7, 8, 9, 10



# Binary Search Tree – Deletion Operation

- If the target node has **two children**, replace the node with its **in-order successor** or predecessor node and delete that node.

Case 3: Two Children



## ❑ Delete Operation

```
def delete(self, val):
    self.root = self._delete(val, self.root)

def _delete(self, val, node):
    if not node:
        return node

    if val < node.val:
        node.left = self._delete(val, node.left)
    elif val > node.val:
        node.right = self._delete(val, node.right)
    else:
        # Case 1: Node has no children
        if not node.left and not node.right:
            node = None
        # Case 2: Node has one child
        elif not node.left:
            node = node.right
        elif not node.right:
            node = node.left
        # Case 3: Node has two children
        else:
            successor = self._find_min(node.right)
            node.val = successor.val
            node.right = self._delete(successor.val, node.right)

    return node

def _find_min(self, node):
    while node.left:
        node = node.left
    return node
```