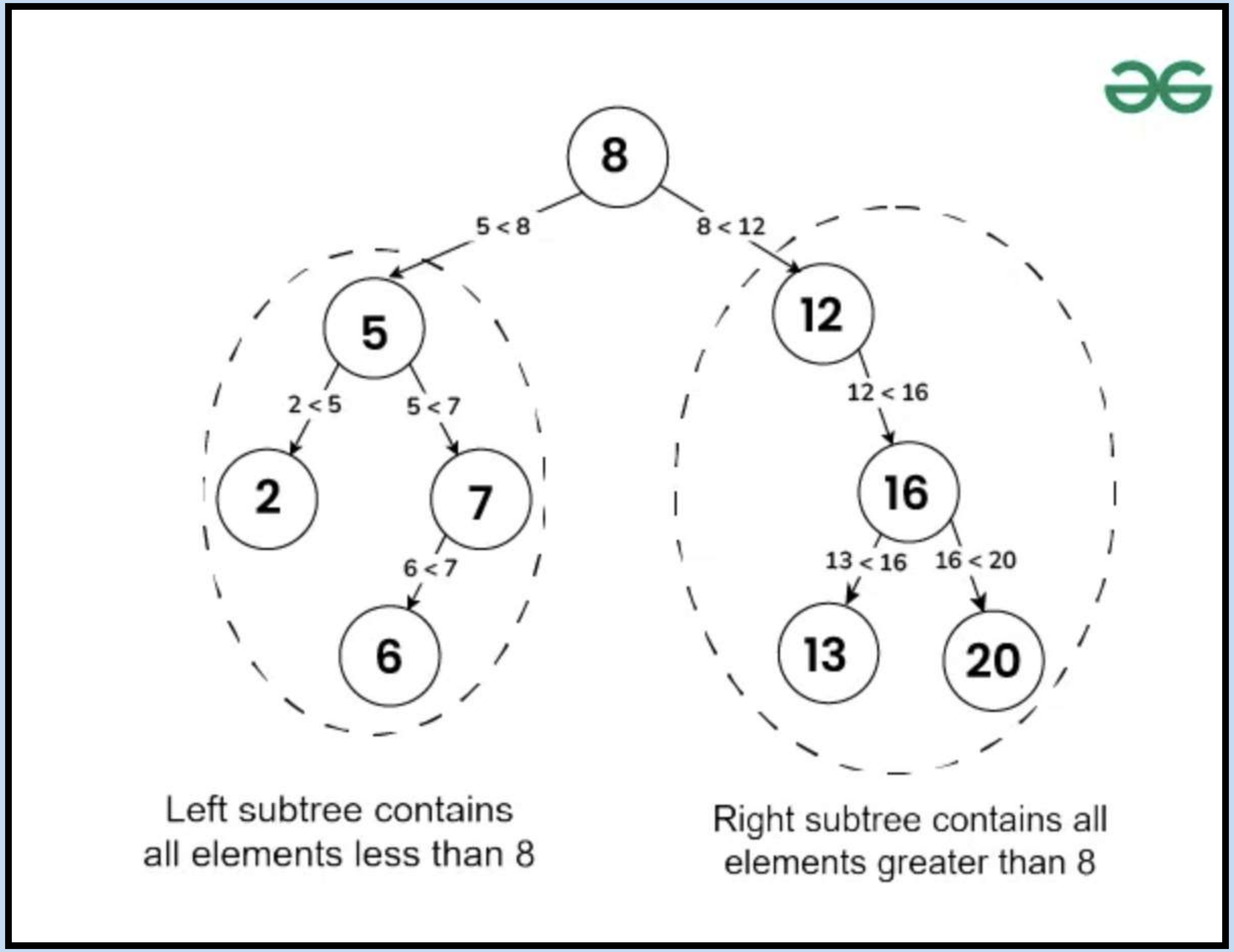
Binary Search Tree

Operations and Time complexity

Vedashree Bhat - dl4176

Binary Search Tree

Binary Search Tree is a Binary tree in which for every node the elements in its left sub tre



Properties of BST

1) The left subtree of a node contains only nodes with keys lesser than the node’s key.

2) The right subtree of a node contains only nodes with keys greater than the node’s key.

3) The left and right subtree each must also be a binary search tree.

4) There must be no duplicate nodes(BST may have duplicate values with different handli

Operations in BST

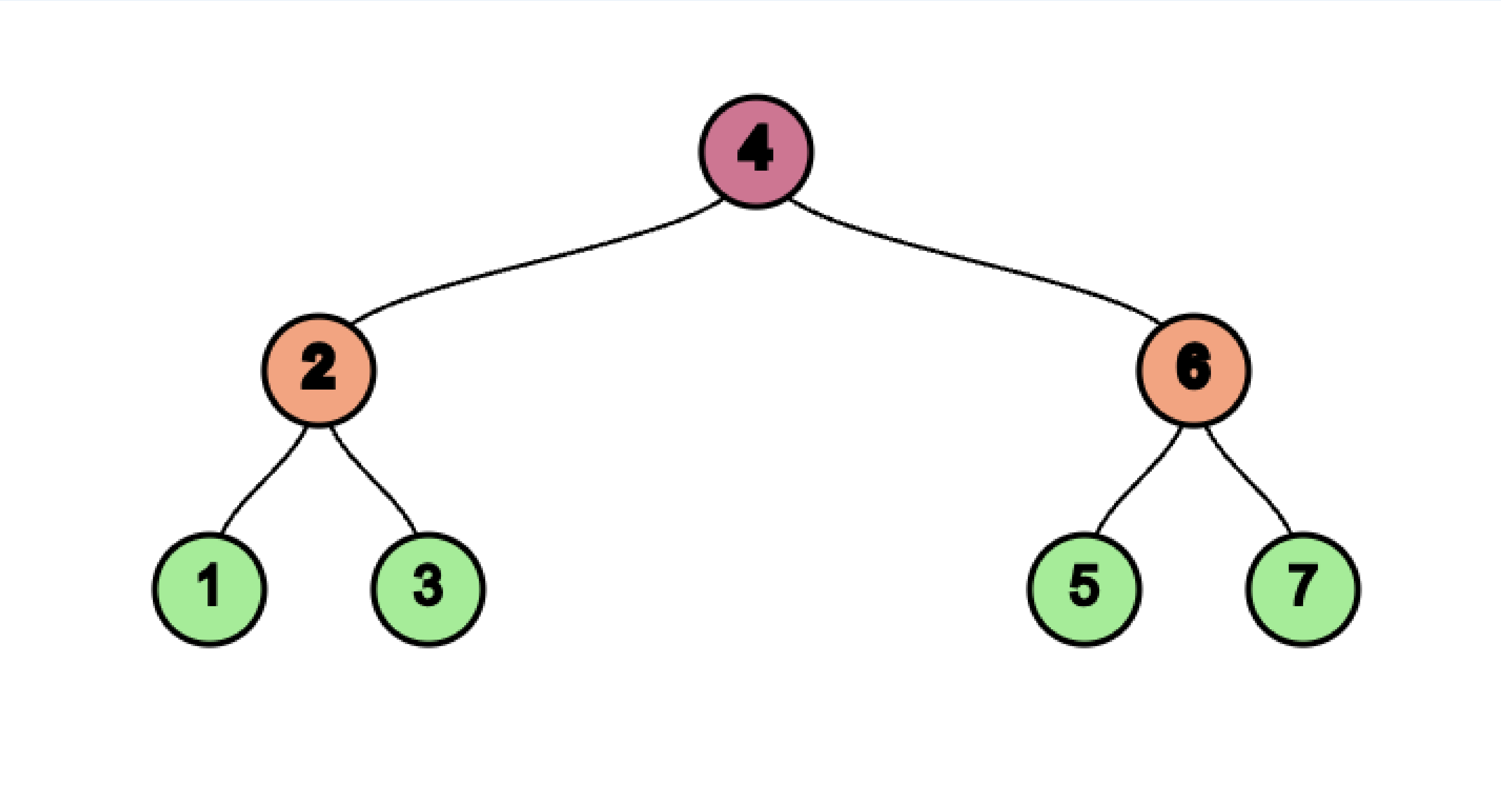
1. Searching
2. Insertion
3. Deletion
4. Traversal

Difference between Binary tree and Binary search tree

Feature	Binary Tree	Binary Search Tree (BST)
Structure	No specific structure	Ordered structure
Node Values	No restrictions	Left < Node < Right
Search Complexity	O(n)	O(log n) average (O(n) worst case)
Insertion Complexity	O(n)	O(log n) average (O(n) worst case)
Traversal	Various methods	In-order yields sorted order

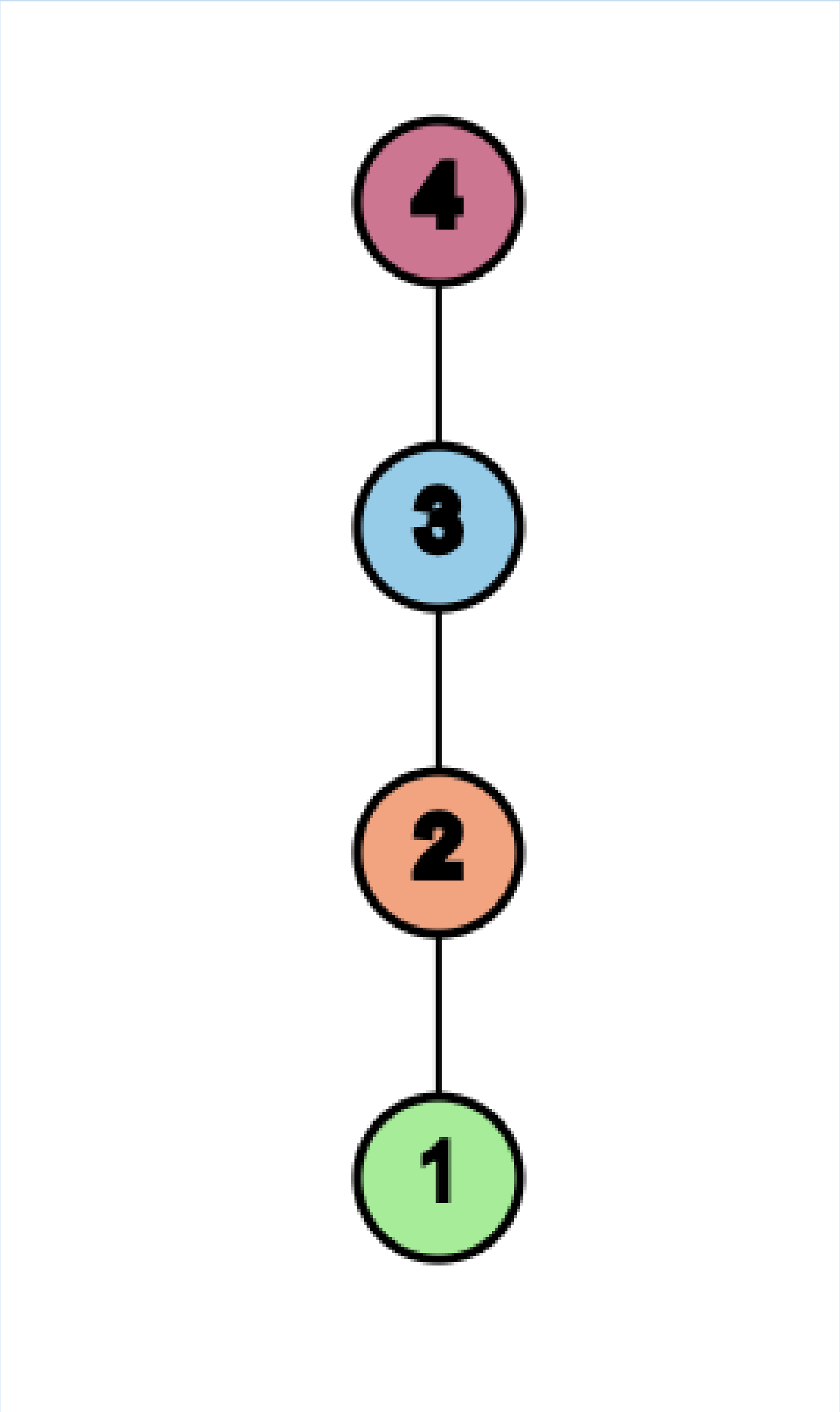
Balanced BST :

A balanced binary search tree is a type of BST where the height of the left and right subtr



Un-balanced BST :

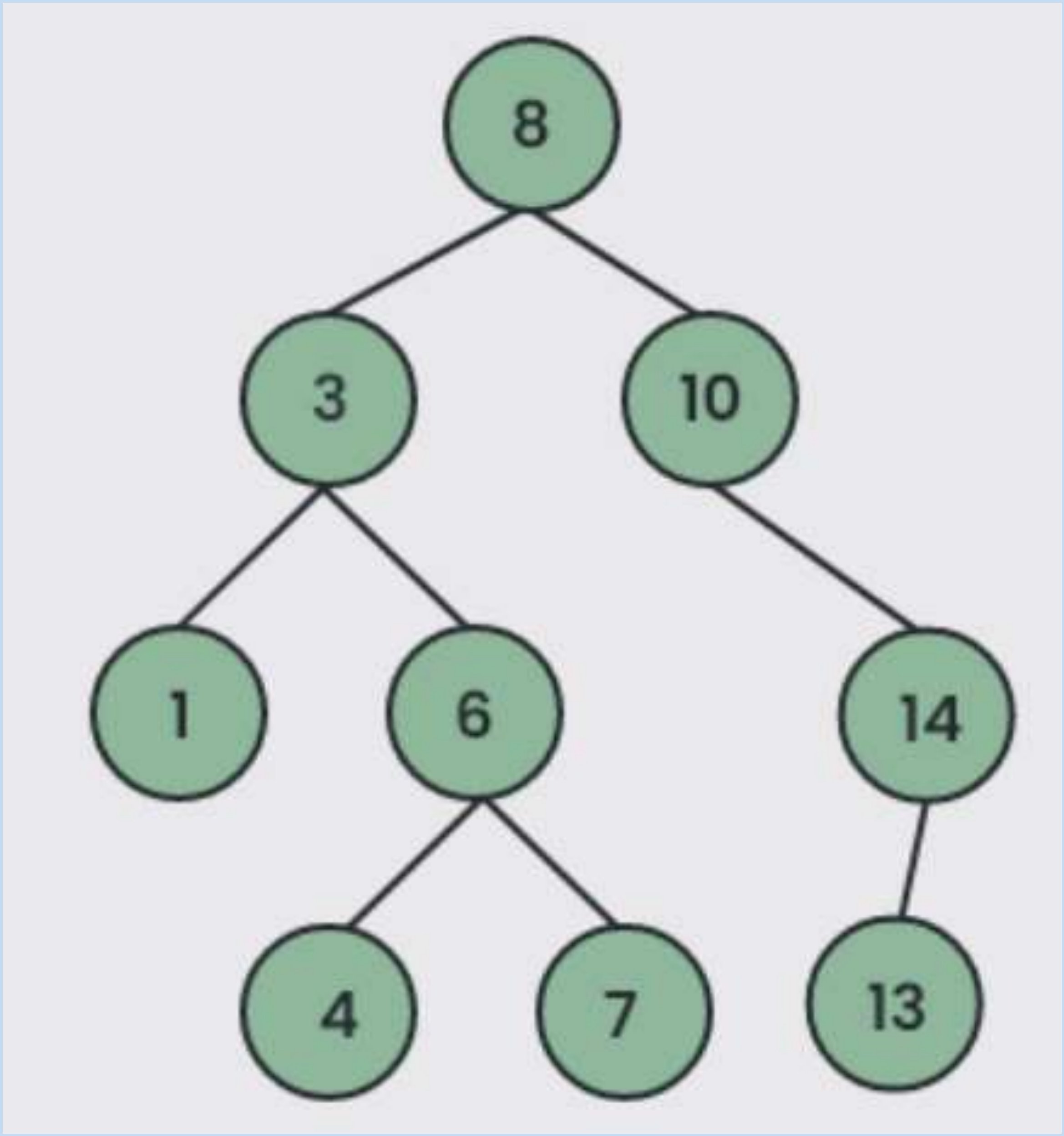
An unbalanced binary search tree is a BST where there is no constraint on the height diff



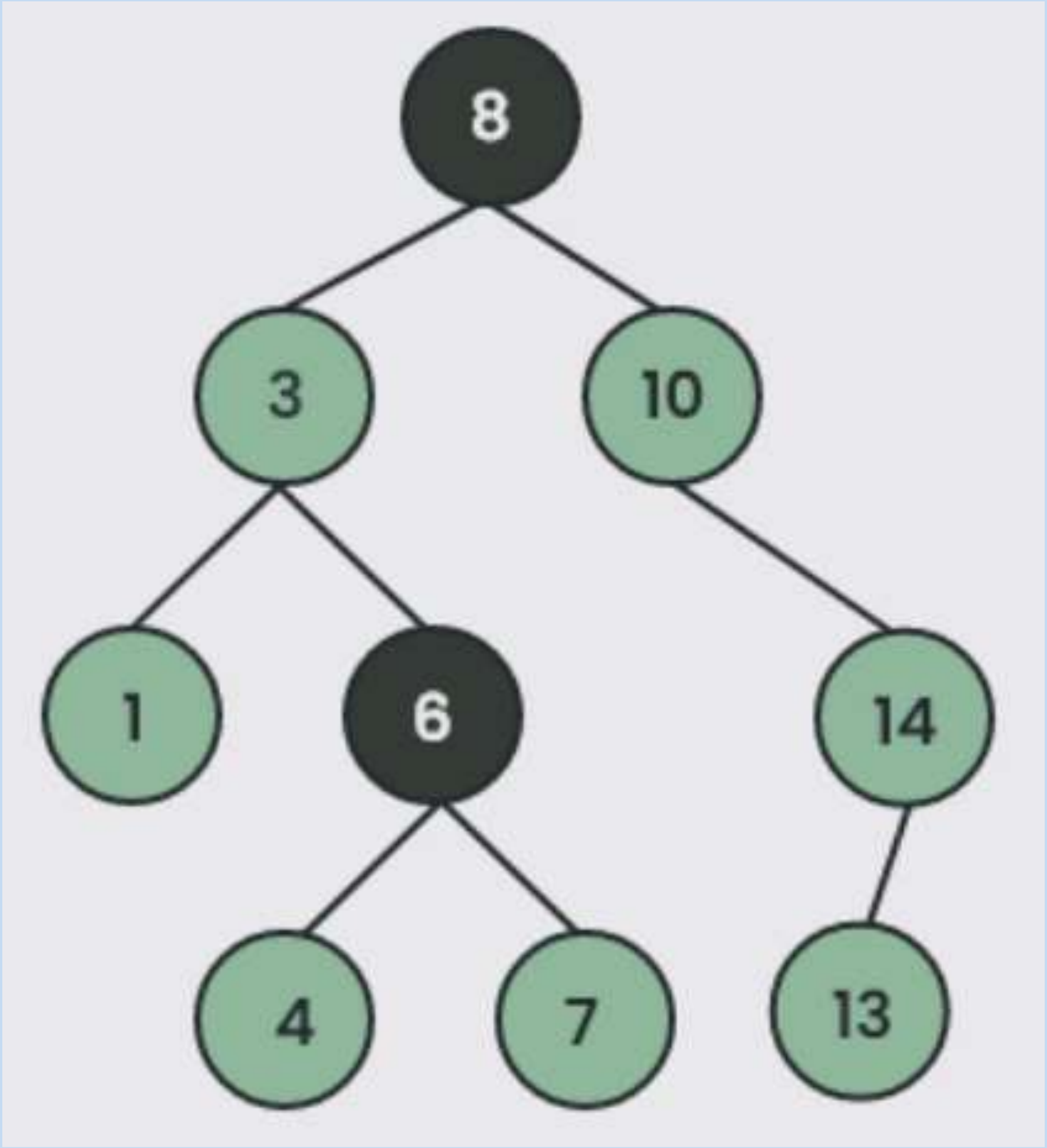
Searching operation

Algorithm :

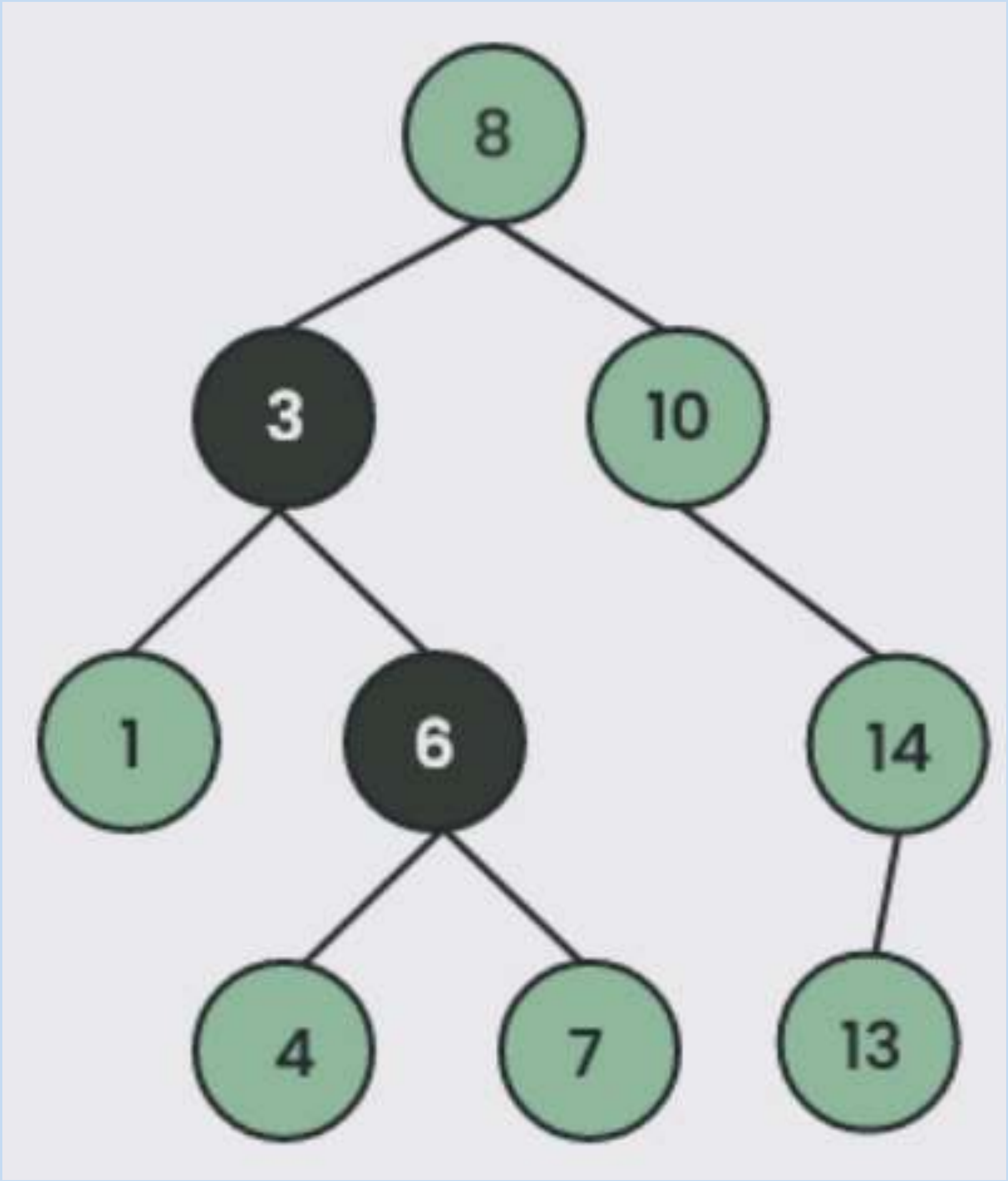
1) First, compare the element to be searched with the root element of the tree.
2) If root is matched with the target element, then return the node's location.
3) If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4) If it is larger than the root element, then move to the right subtree.
5) Repeat the above procedure recursively until the match is found.
6) If the element is not found or not present in the tree, then return NULL.



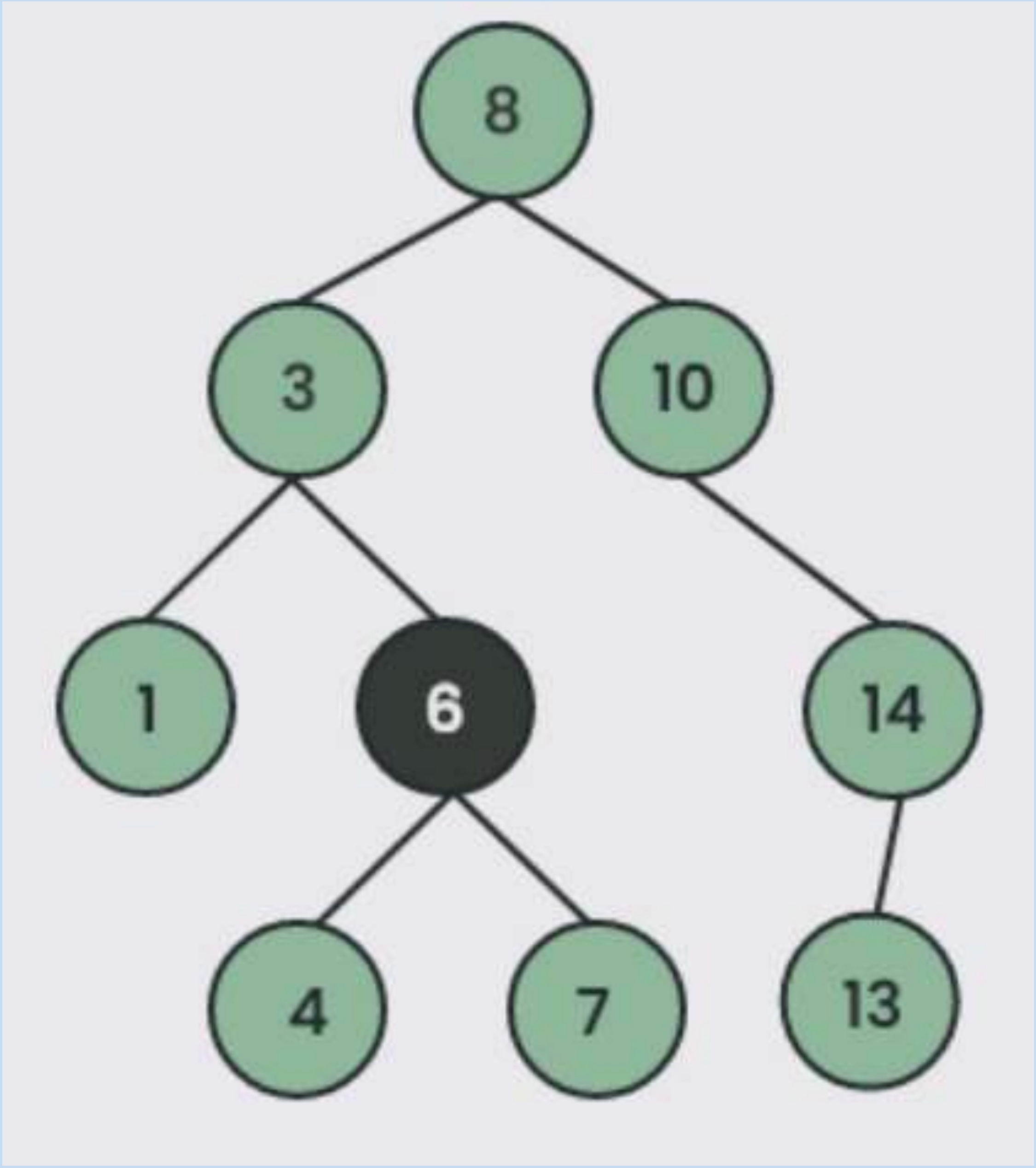
Consider the following BST, let’s search key=6



Compare key with root, 8 here. As 6<8, search in left subtree of 8



As key 6>3, search in the right subtree of 3



Element found

Code

```
// function to search a key in a BST
static Node search(Node root, int key)
{
    // Base Cases: root is null or key is present at
    // root
    if (root == null || root.key == key)
        return root;

    // Key is greater than root's key
    if (root.key < key)
        return search(root.right, key);

    // Key is smaller than root's key
    return search(root.left, key);
}
```

The time taken depends on the **height of the BST**, denoted as h.

Best case : **O(1)**
O(1) – Key is found at the root.

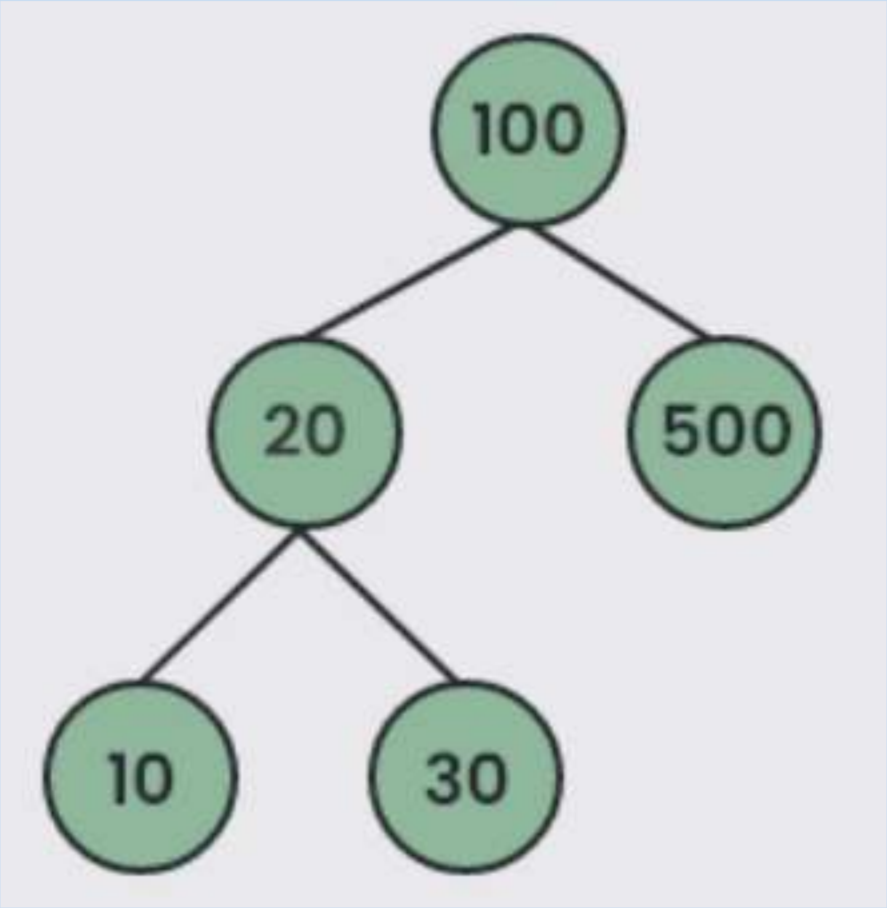
Average case:**O(log n)**
For a balanced BST, we search only one side at each level, and the total height is log n.

Worst case: O(n) **For an unbalanced BST (like a skewed tree), the height becomes `n`.**

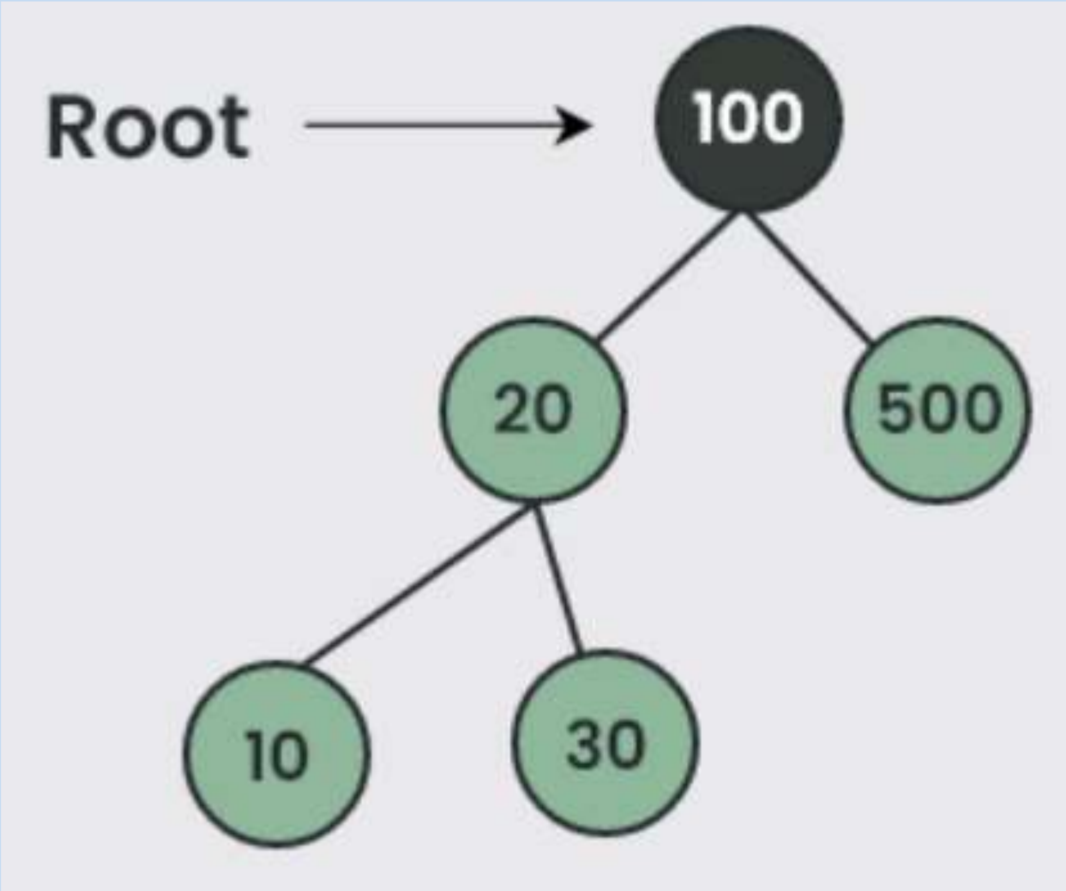
Thus, the **overall time complexity** of the recursive search operation is: O(h) where `h` is the height of the tree. In a **bala**
h=O(log n) while in the worst case, h = O(n)

Insertion Operation

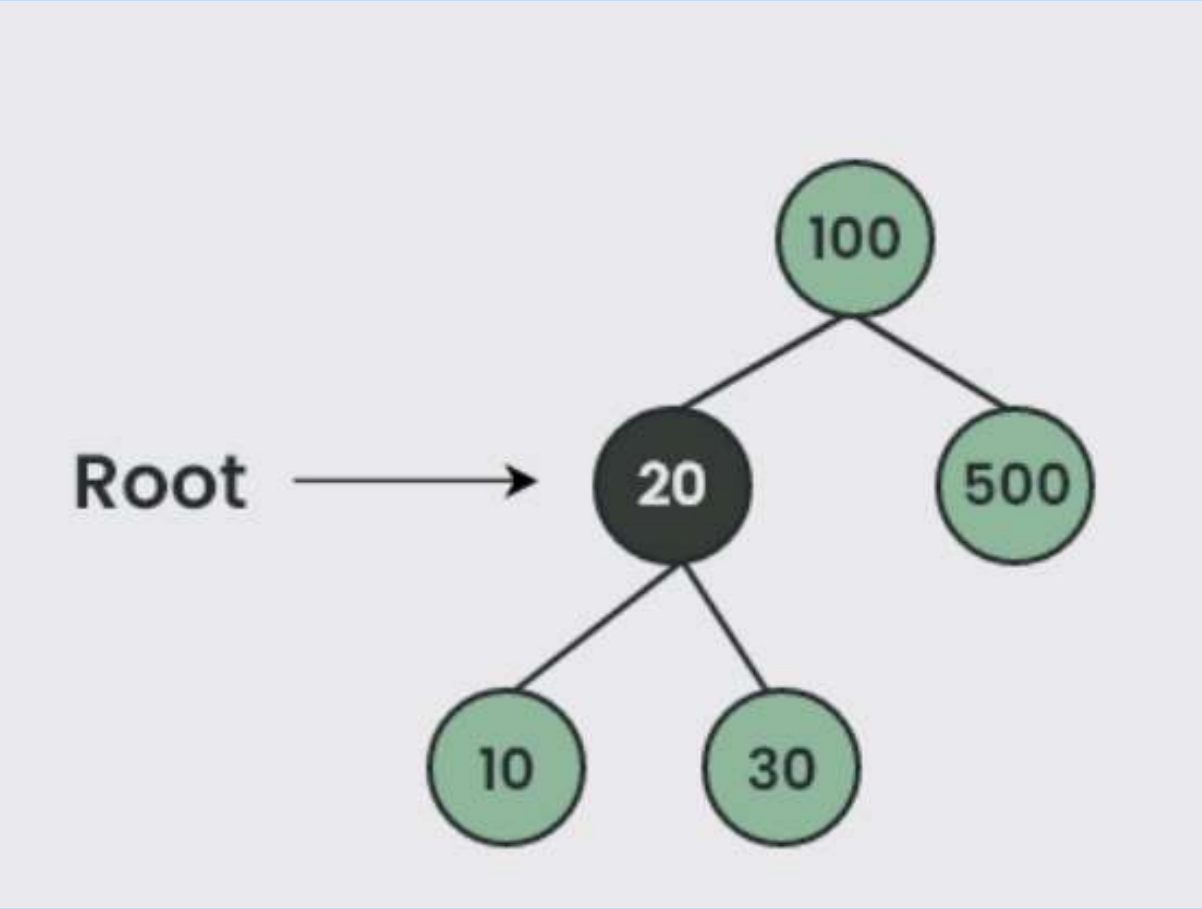
A new key is always inserted at the leaf. Start searching a key from the root till a leaf node. Once a leaf n



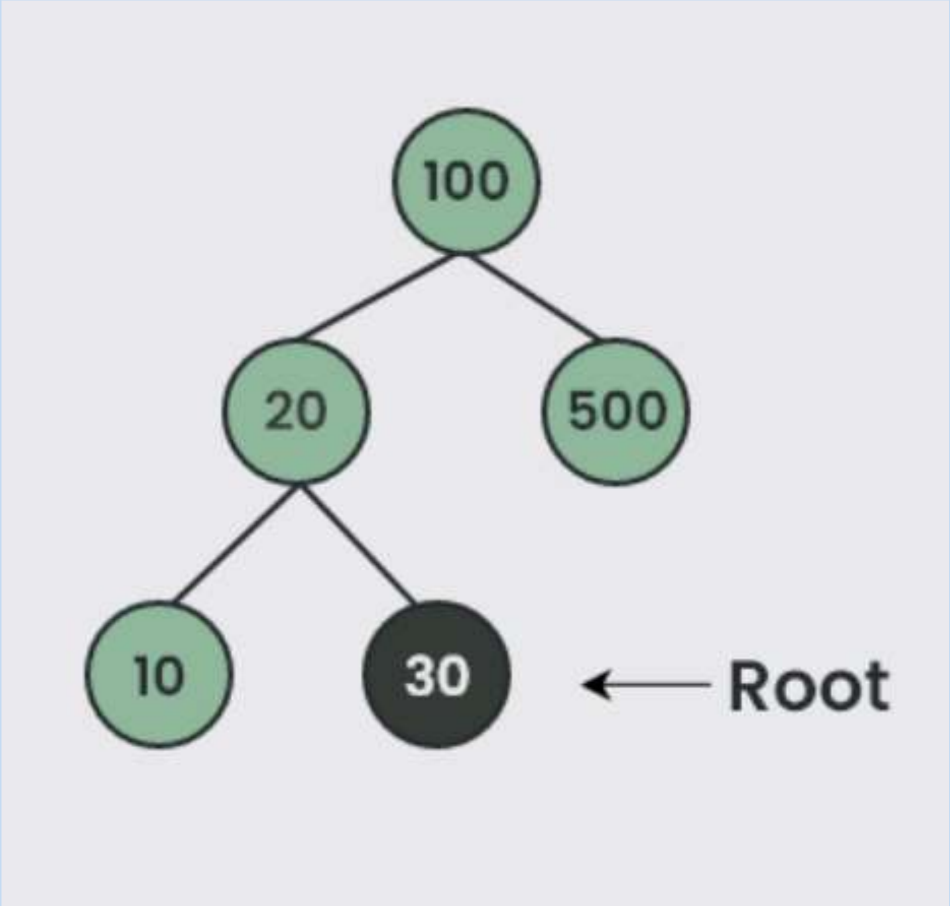
Let X=40, be the node to be inserted



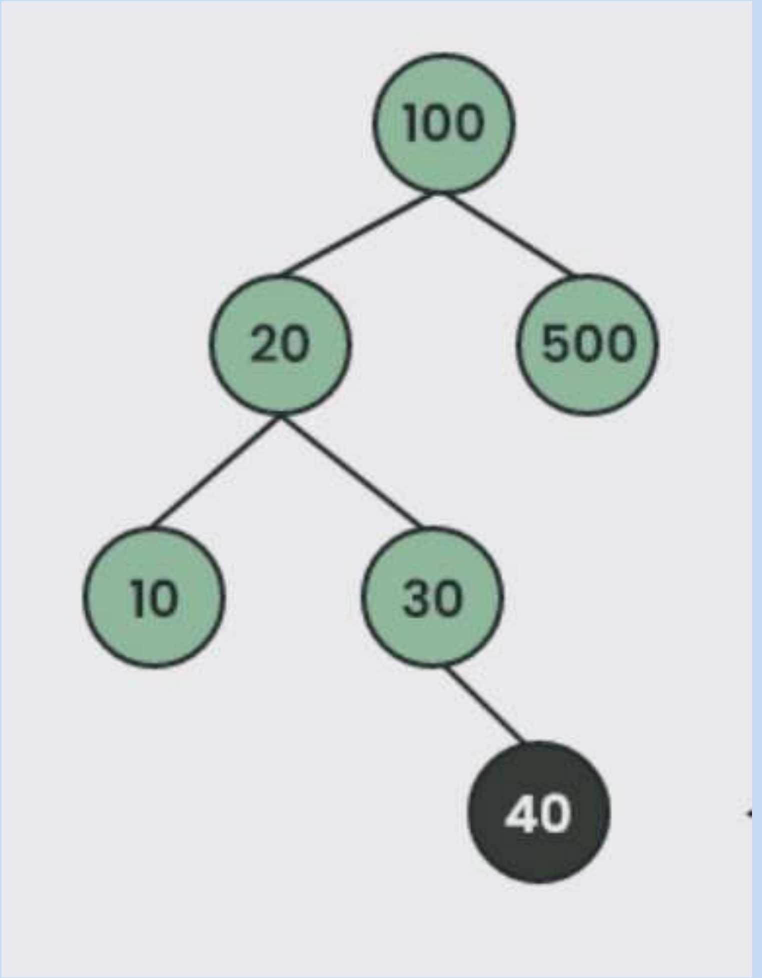
Since, 100 is greater than 40, move pointer to the left child (20)



Since 20 is less than 40, move pointer to the right child (30)



Again 40 is greater than 30, move pointer to the right side of 30



As 40 is greater than node 30, thus it will be inserted to the right of 30

Code

```java
// A utility function to insert a new node
// with the given key
static Node insert(Node root, int key)
{

    // If the tree is empty, return a new node
    if (root == null)
        return new Node(key);

    // If the key is already present in the tree,
    // return the node
    if (root.key == key)
        return root;

    // Otherwise, recur down the tree
    if (key < root.key)
        root.left = insert(root.left, key);
    else
        root.right = insert(root.right, key);

    // Return the (unchanged) node pointer
    return root;
}
```

The time taken depends on the **height of the BST**, denoted as h.

Best case : **O(1)**
O(1) – Tree is empty and new node is the root node.

Average case:**O(log n)**

Worst case: O(n) For an unbalanced BST (like a skewed tree), the height becomes `n`.

Thus, the **overall time complexity** of the recursive search operation is: O(h) where `h` is the height of the tree. In a **bal**
h=O(log n) while in the worst case, h = O(n)

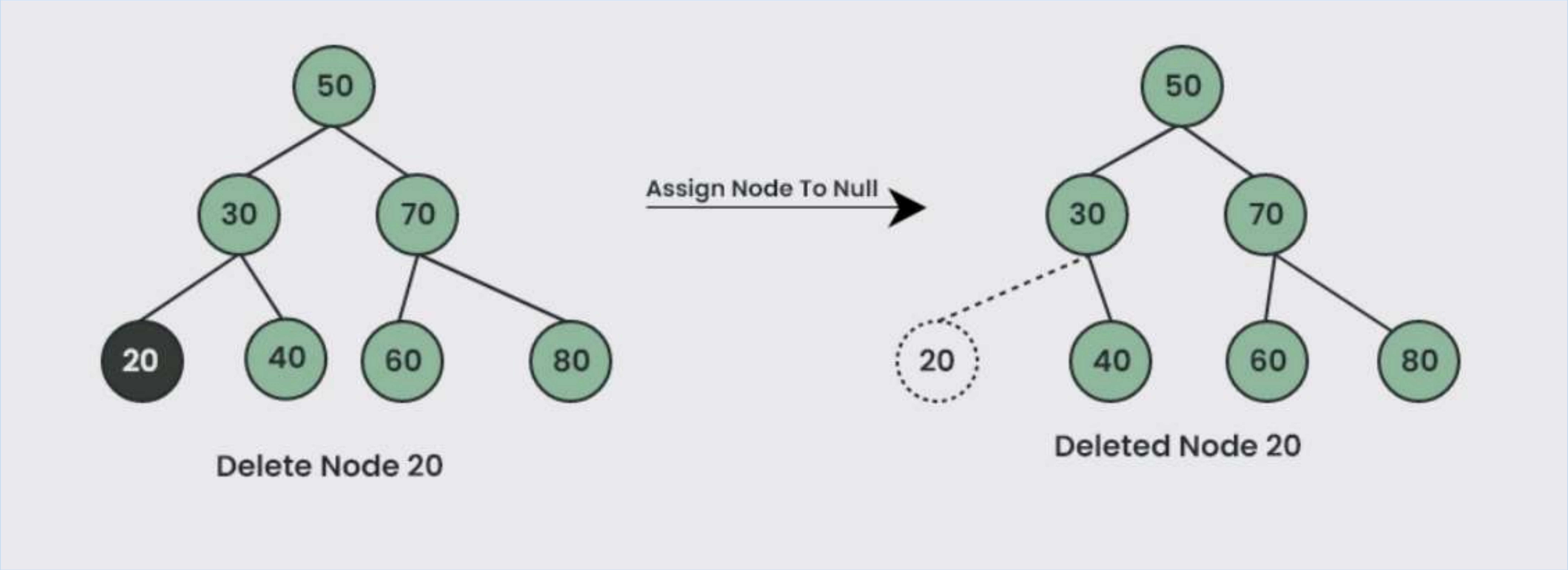
Deletion Operation

It is used to delete a node with specific key from the BST and return the new BST

Different Scenarios for deleting a node :

1. Node to be deleted is the leaf node
2. Node to be deleted has one child
3. Node to be deleted has two children

Node to be deleted is the leaf node



Code

```java
Node deleteLeaf(Node root, int key) {
    if (root == null) {
        return null;
    }

    if (key < root.key) {
        root.left = deleteLeaf(root.left, key);
    } else if (key > root.key) {
        root.right = deleteLeaf(root.right, key);
    } else {
        // Node is a leaf
        if (root.left == null && root.right == null) {
            return null; // Deleting leaf node
        }
    }
    return root;
}
```

The time taken depends on the **height of the BST**, denoted as h.

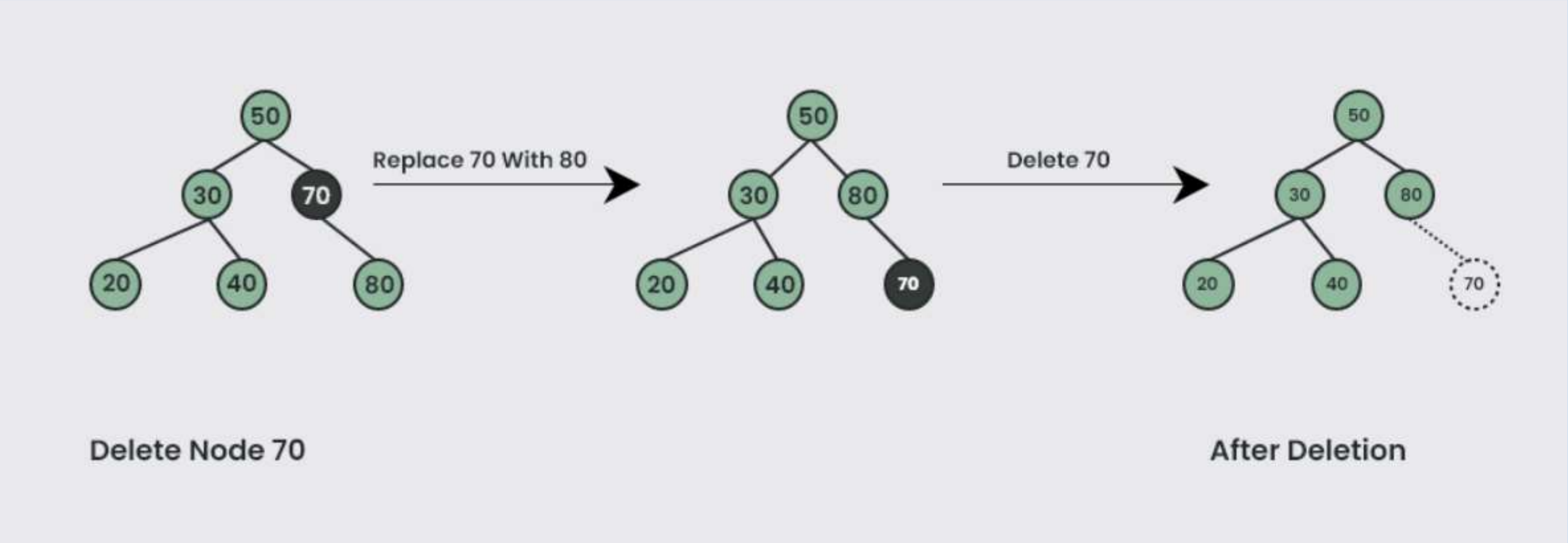
Best case : **O(1)**

Average case:**O(log n)**

Worst case: O(n) **For an unbalanced BST (like a skewed tree), the height becomes `n`.**

Thus, the **overall time complexity** of the recursive search operation is: O(h) where `h` is the height of the tree. In a **bal**
h=O(log n) while in the worst case, h = O(n)

Node to be deleted has one child



Code

```
Node deleteNodeWithOneChild(Node root, int key) {
    if (root == null) {
        return null;
    }

    if (key < root.key) {
        root.left = deleteNodeWithOneChild(root.left, key);
    } else if (key > root.key) {
        root.right = deleteNodeWithOneChild(root.right, key);
    } else {
        // Node with one child
        if (root.left == null) {
            return root.right; // Replace node with its right child
        } else if (root.right == null) {
            return root.left; // Replace node with its left child
        }
    }
    return root;
}
```

The time taken depends on the **height of the BST**, denoted as h.

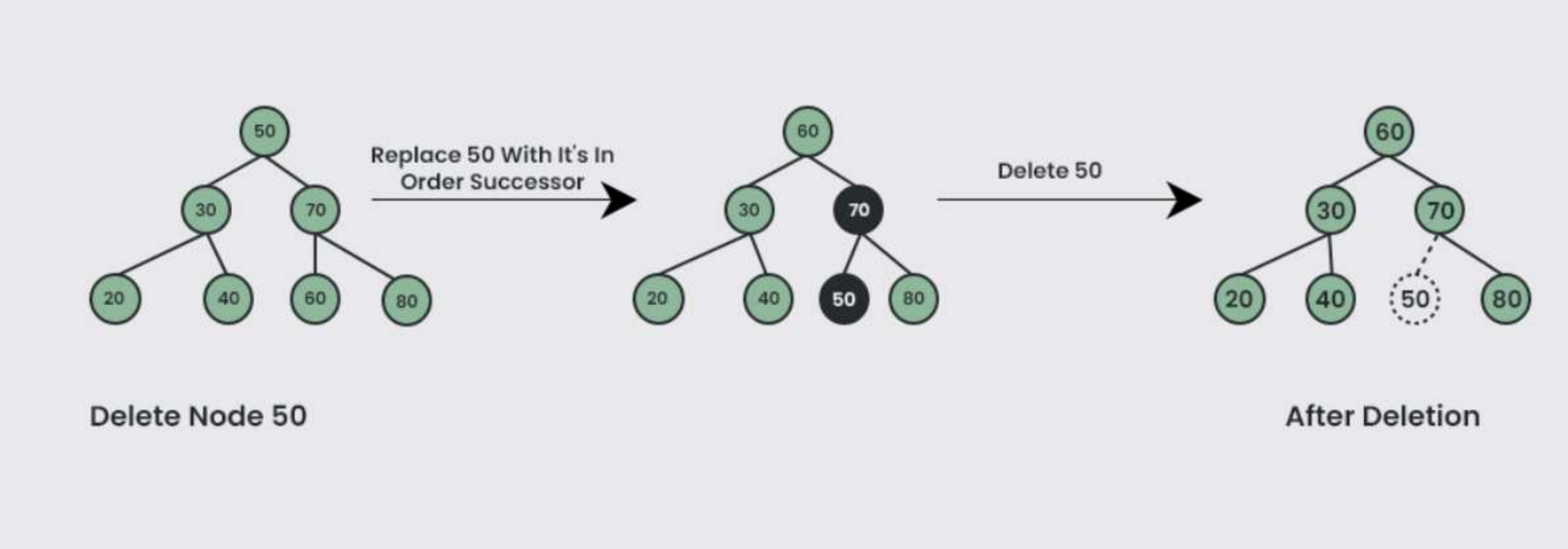
Best case : **O(1)**

Average case:**O(log n)**

Worst case: O(n) **For an unbalanced BST (like a skewed tree), the height becomes n.**

Thus, the **overall time complexity** of the recursive search operation is: O(h) where h is the height of the tree. In a **balanced BST**,
h=O(log n) while in the worst case, h = O(n)

Node to be deleted has two children



Code

```
Node deleteNodeWithTwoChildren(Node root, int key) {
    if (root == null) {
        return null;
    }

    if (key < root.key) {
        root.left = deleteNodeWithTwoChildren(root.left, key);
    } else if (key > root.key) {
        root.right = deleteNodeWithTwoChildren(root.right, key);
    } else {
        // Node with two children
        Node successor = findMin(root.right); // Find the inorder successor
        root.key = successor.key; // Replace with successor's key
        root.right = deleteNodeWithTwoChildren(root.right, successor.key); /
    }
    return root;
}

Node findMin(Node root) {
    while (root.left != null) {
        root = root.left;
    }
    return root;
}
```

The time taken depends on the **height of the BST**, denoted as h.

Best case : **O(1)**
O(1) – Key is found at the root.

Average case:**O(log n)**
For a balanced BST, we search only one side at each level, and the total height is log n.

Worst case: O(n) **For an unbalanced BST (like a skewed tree), the height becomes `n`.**

Thus, the **overall time complexity** of the recursive search operation is: O(h) where `h` is the height of the tree. In a **balanced BST**,
h=O(log n) while in the worst case, h = O(n)

Tree Traversal :

Inorder Traversal :

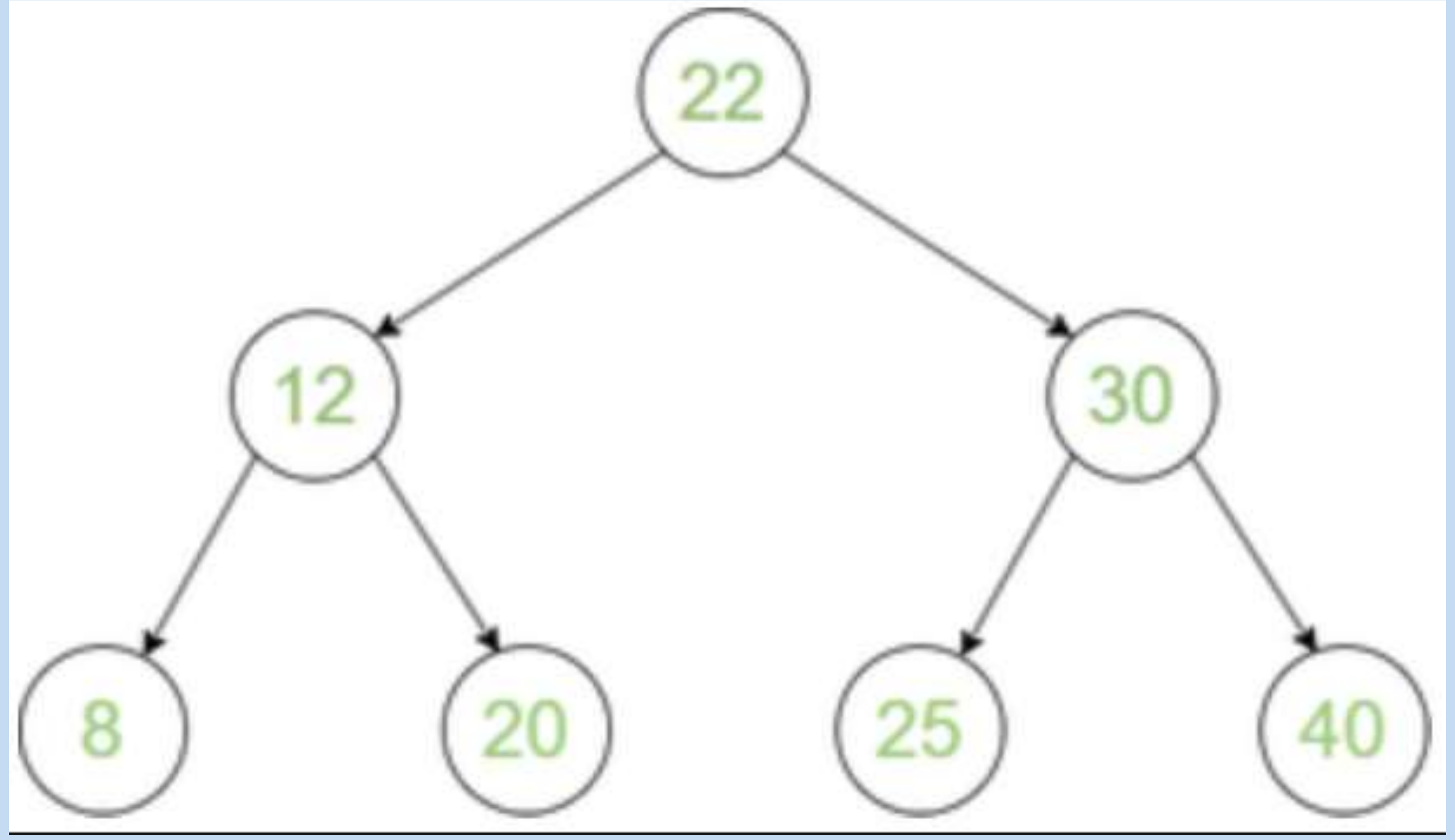
1) Traverse left subtree
2) Visit the root and print the data.
3) Traverse the right subtree

```
// Inorder Traversal
public static void printInorder(Node node)
{
    if (node == null)
        return;

    // Traverse left subtree
    printInorder(node.left);

    // Visit node
    System.out.print(node.data + " ");

    // Traverse right subtree
    printInorder(node.right);
}
```



Inorder Traversal

8,12,20,22,25,30,40

Preorder Traversal :

1) Visit the root and print the data.

2) Traverse left subtree

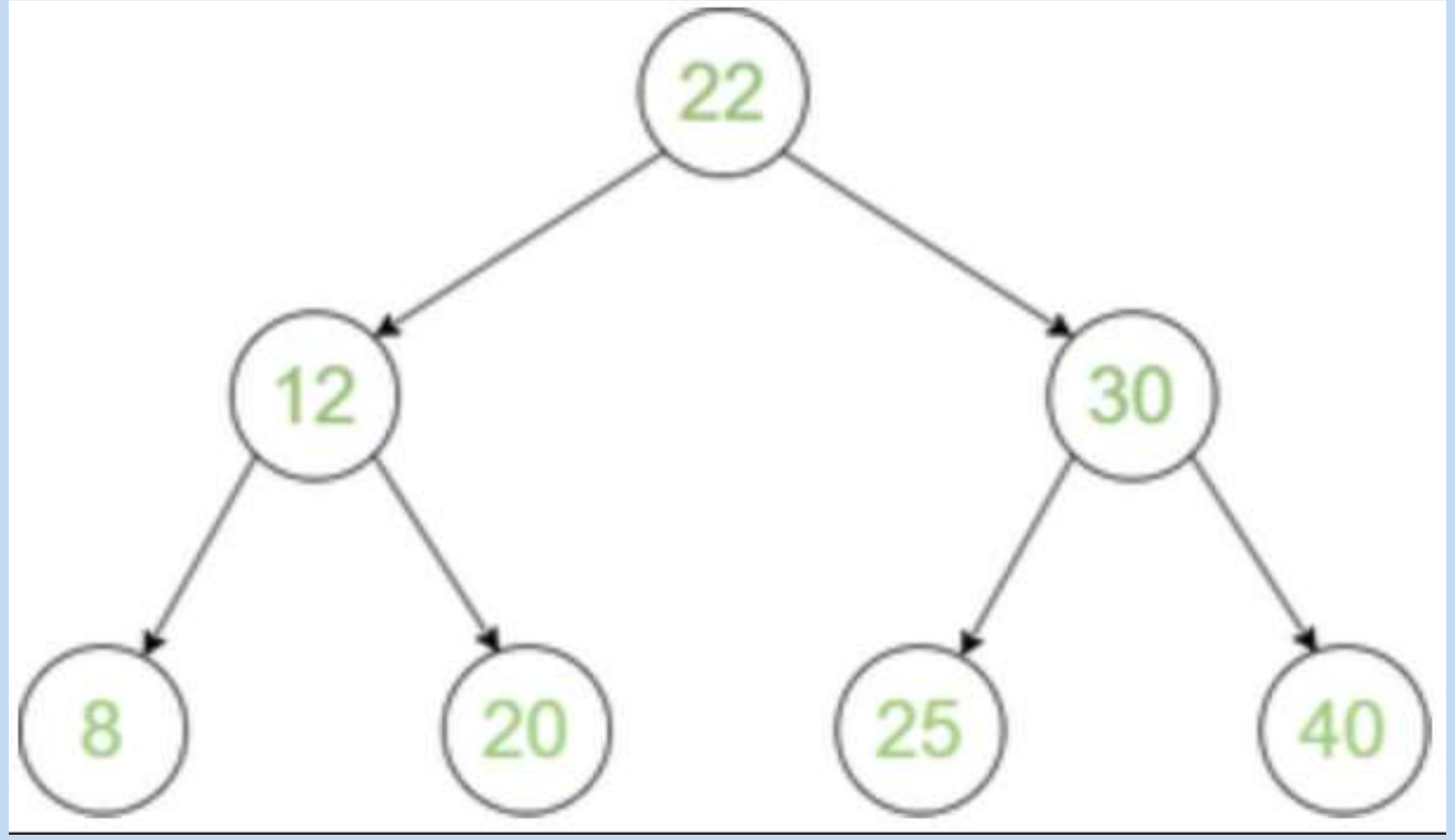
3) Traverse the right subtree

```java
// Preorder Traversal
public static void printPreorder(Node node)
{
  if (node == null)
    return;

  // Visit node
  System.out.print(node.data + " ");

  // Traverse left subtree
  printPreorder(node.left);

  // Traverse right subtree
  printPreorder(node.right);
}
```



Preorder Traversal →

22,12,8,20,30,25,40

Postorder Traversal :

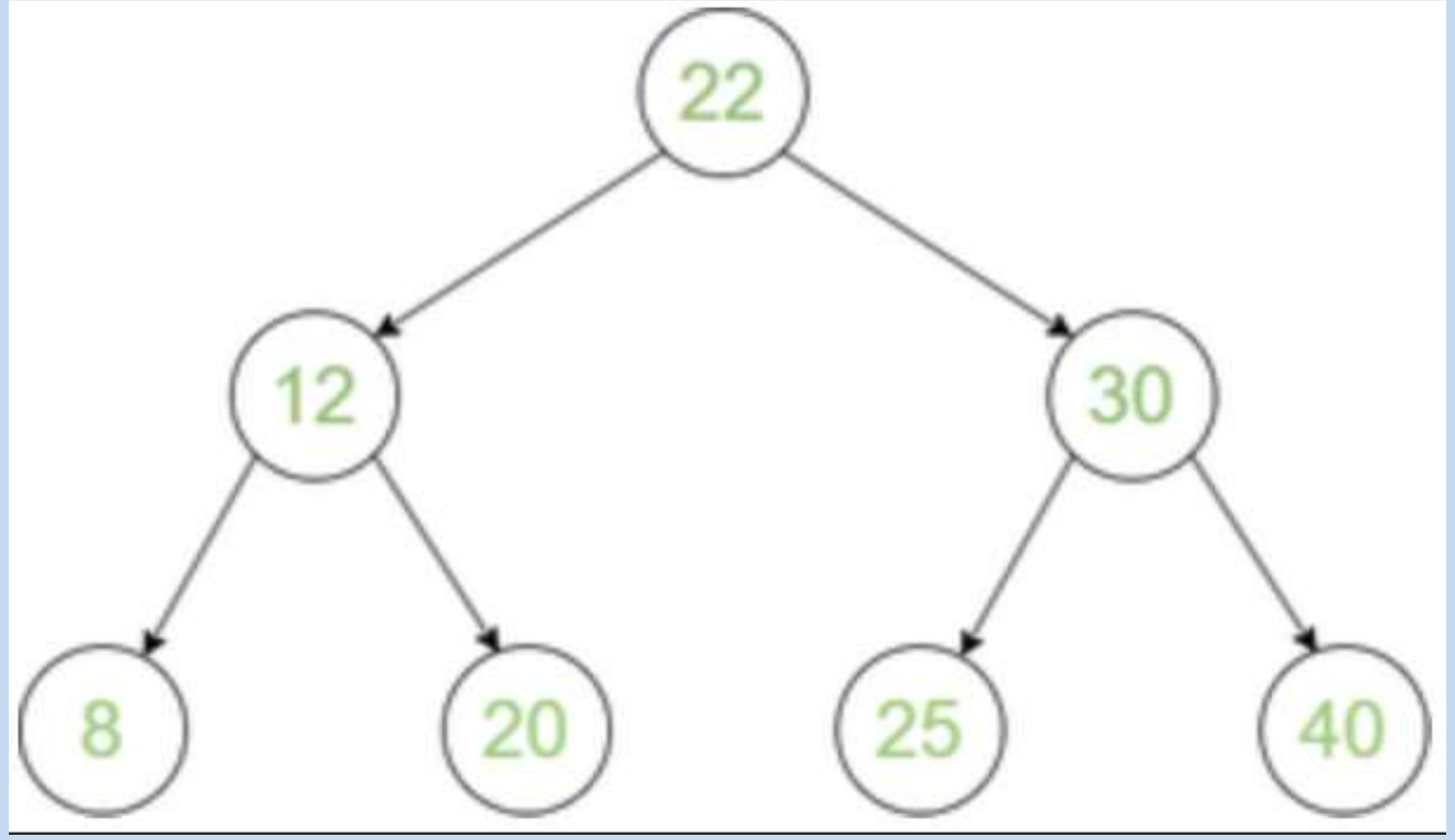
1) Traverse left subtree
2) Traverse the right subtree
3) Visit the root and print the data.

```java
public static void printPostOrder(Node node)
{
  if (node == null)
    return;

  // Traverse left subtree
  printPostOrder(node.left);

  // Traverse right subtree
  printPostOrder(node.right);

    // Visit node
  System.out.print(node.data + " ");
}
```



Postorder Traversal →

8,20,12,25,40,30,22

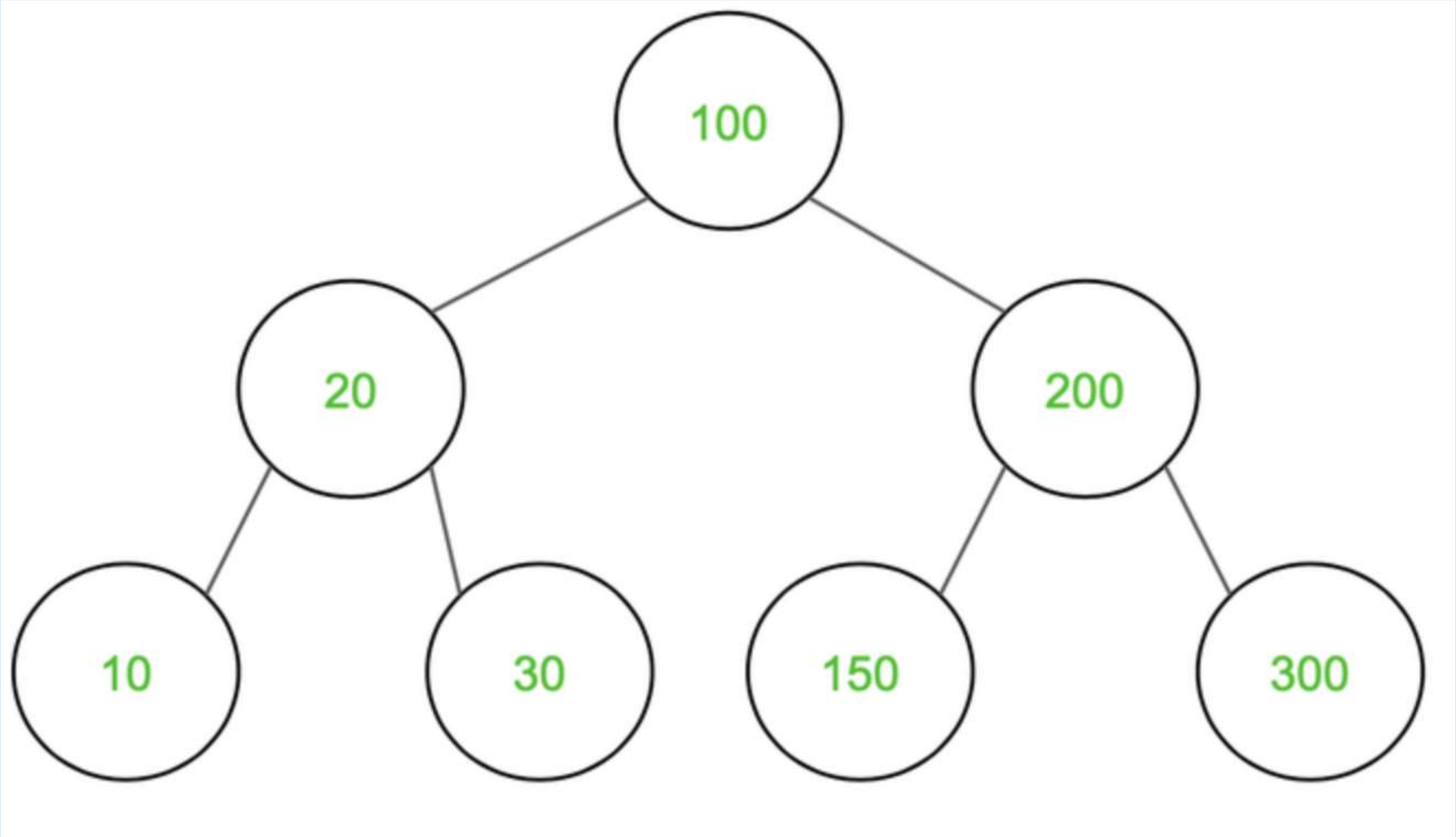
In a Binary Tree, Traversals can be done in various ways (in-order, pre-order, post-order, level-order) b

But in Binary Search Tree (BST): In-order traversal of a BST results in a sorted sequence of values, ma

Time complexity of Tree traversal

In all traversal types, **each node is visited exactly once**, so the time complexity is O(n) , where n is the number of nodes in the tree.

Can you find inorder, preorder, and postorder traversal of the below tree ?



Answers :

Inorder - 10,20,30,100,150,200,300

Preorder - 100,20,10,30,200,150,300

Postorder - 10,30,20,150,300,200,100

Thank you!