Binary Tree - Complete Notes

Table of Contents

Introduction

A **Binary Tree** is a hierarchical data structure where each node has at most two children, referred to as the left child and right child.

Basic Structure

```
class TreeNode {
    int data;
    TreeNode left;
    TreeNode right;

    TreeNode(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}
```

Types of Binary Trees

1. Perfect Binary Tree

A binary tree is **perfect** if:

- All internal nodes have exactly two children
- All leaf nodes are at the same level
- The tree is completely filled up to the nth level

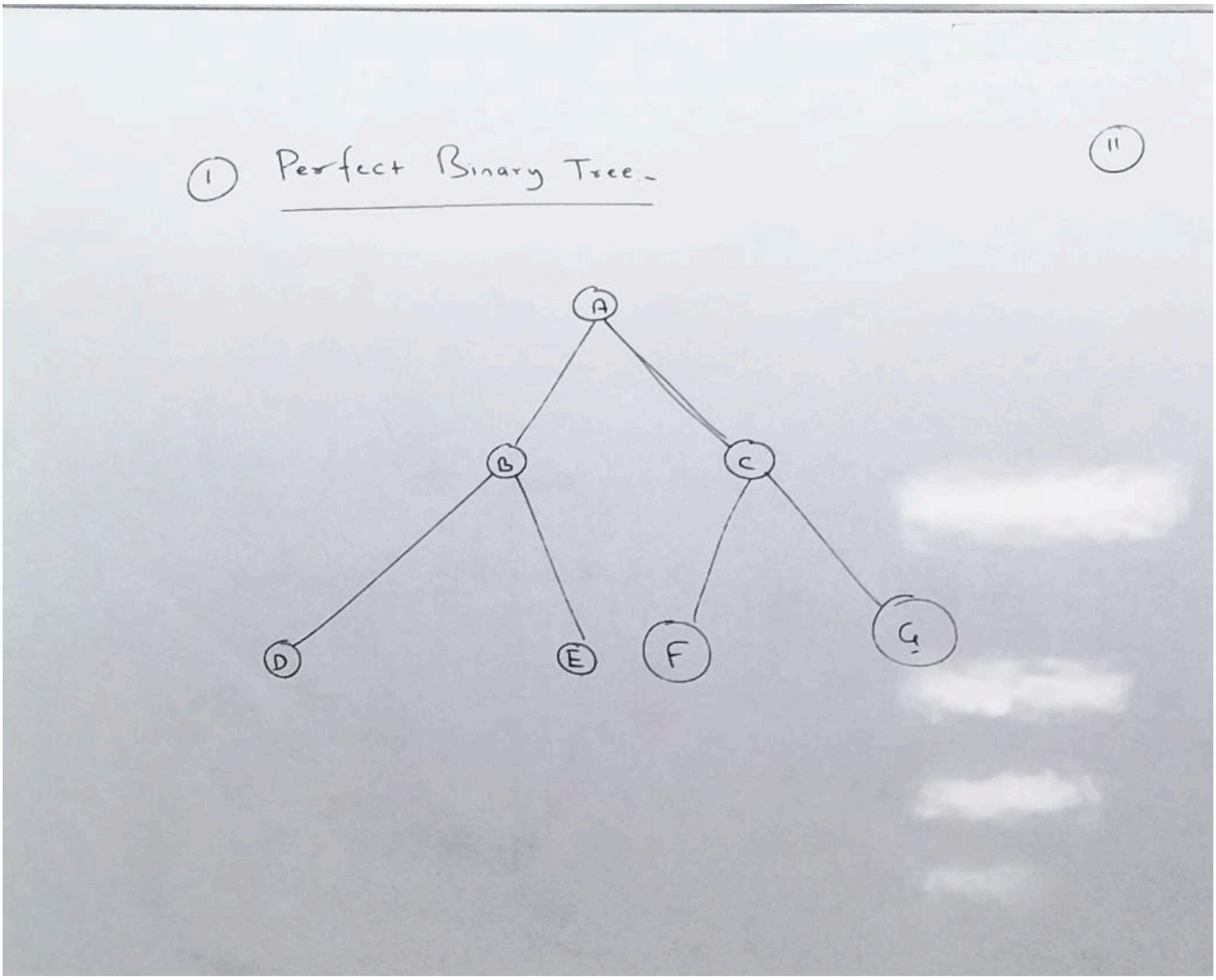
Properties:

- Total nodes = 2^(h+1) - 1, where h is height
- Leaf nodes = 2^h

- Height = $\log_2(n+1) - 1$

Example:

```
        1
       / \
      2   3
     / \ / \
    4  5 6  7
```



2. Full Binary Tree

A binary tree is **full** if every node has either:

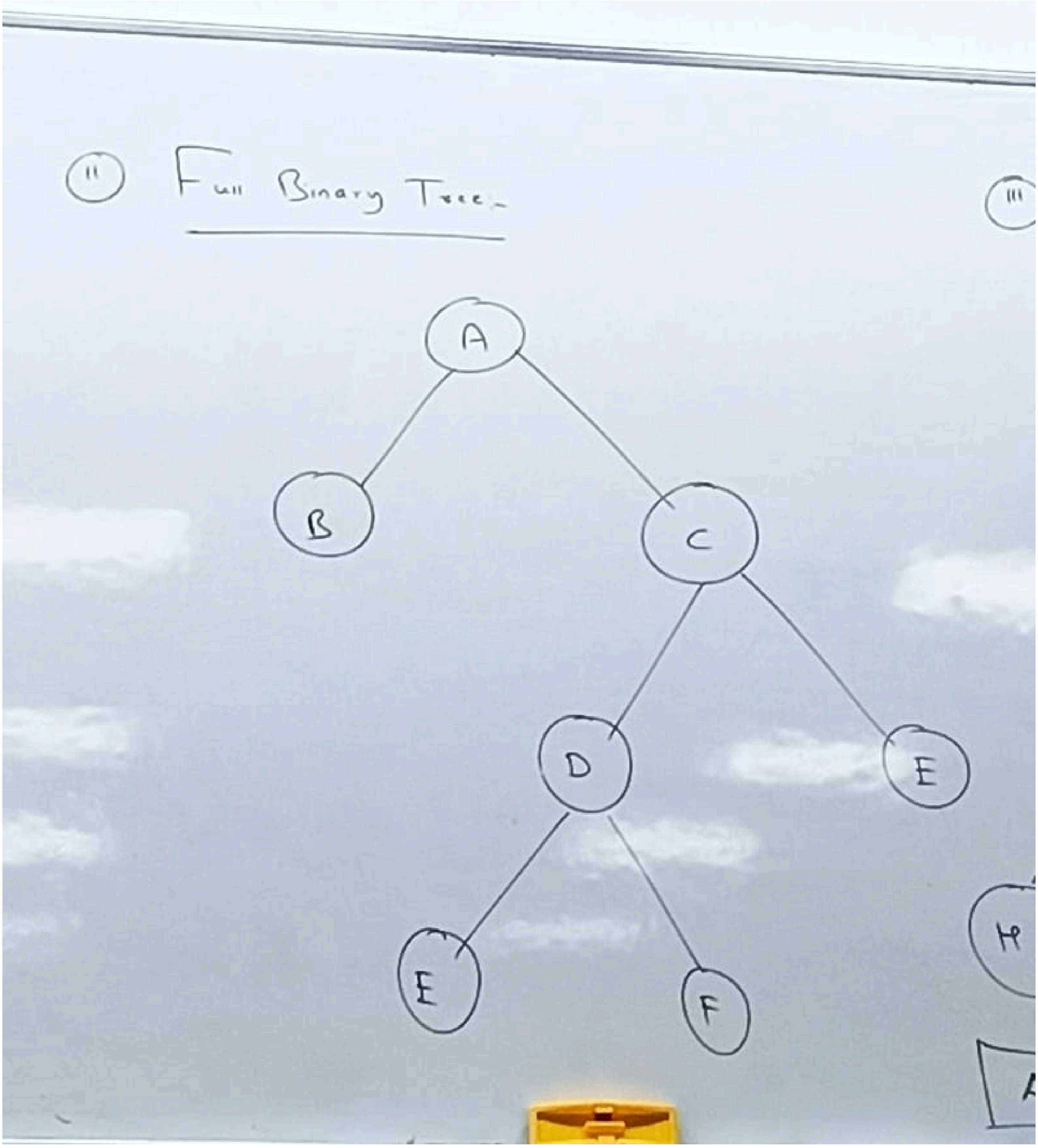
- Zero children (leaf node), OR
- Two children (internal node)

Properties:

- If there are n nodes, then there are (n+1)/2 leaf nodes
- Number of internal nodes = (n-1)/2

Example:

```
        1
       / \
      2   3
     / \
    4   5
```



---

3. Complete Binary Tree

A binary tree is **complete** if:

- All levels are completely filled except possibly the last level
- The last level is filled from left to right

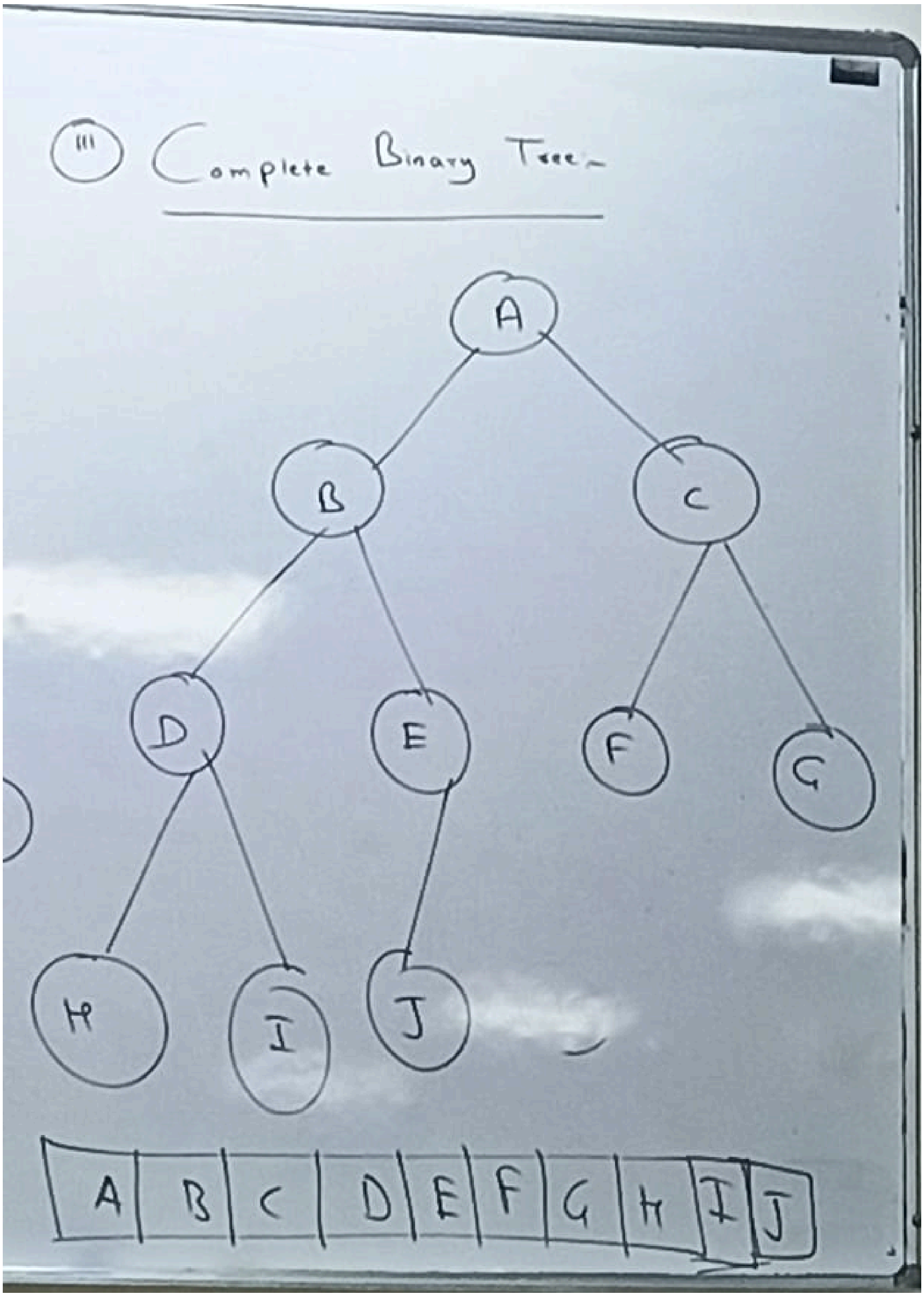
- Used in heap data structures

Properties:

- Height = $\lfloor \log_2(n) \rfloor$
- Efficiently represented using arrays
- Parent of node at index i is at $\lfloor (i-1)/2 \rfloor$
- Left child at 2i + 1, Right child at 2i + 2

Example:

```
      1
     / \
    2   3
   / \  /
  4  5 6
```



---

4. Degenerate (Skewed) Binary Tree

Every parent node has only one child. Performance degrades to O(n) like a linked list.

Example (Left Skewed):

```
    1
   /
  2
 /
3
```

---

5. Balanced Binary Tree

A binary tree where the height difference between left and right subtrees is at most 1 for every node. Examples: AVL Tree, Red-Black Tree.

Tree Traversal Methods

1. Inorder Traversal (Left → Root → Right)

Visits nodes in ascending order for BST.

```
public void inorderTraversal(TreeNode root) {
    if (root == null) return;

    inorderTraversal(root.left);
    System.out.print(root.data + " ");
    inorderTraversal(root.right);
}
```

Time Complexity: O(n) **Space Complexity:** O(h) - recursion stack, where h is height

---

2. Preorder Traversal (Root → Left → Right)

Used to create a copy of the tree or get prefix expression.

```
public void preorderTraversal(TreeNode root) {
    if (root == null) return;

    System.out.print(root.data + " ");
    preorderTraversal(root.left);
    preorderTraversal(root.right);
}
```

Time Complexity: O(n) **Space Complexity:** O(h)

---

3. Postorder Traversal (Left → Right → Root)

Used to delete the tree or get postfix expression.

```
public void postorderTraversal(TreeNode root) {
    if (root == null) return;

    postorderTraversal(root.left);
    postorderTraversal(root.right);
    System.out.print(root.data + " ");
}
```

Time Complexity: O(n) **Space Complexity:** O(h)

4. Level Order Traversal (BFS)

Visits nodes level by level from left to right.

```java
import java.util.Queue;
import java.util.LinkedList;

public void levelOrderTraversal(TreeNode root) {
    if (root == null) return;

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty()) {
        TreeNode current = queue.poll();
        System.out.print(current.data + " ");

        if (current.left != null) queue.offer(current.left);
        if (current.right != null) queue.offer(current.right);
    }
}
```

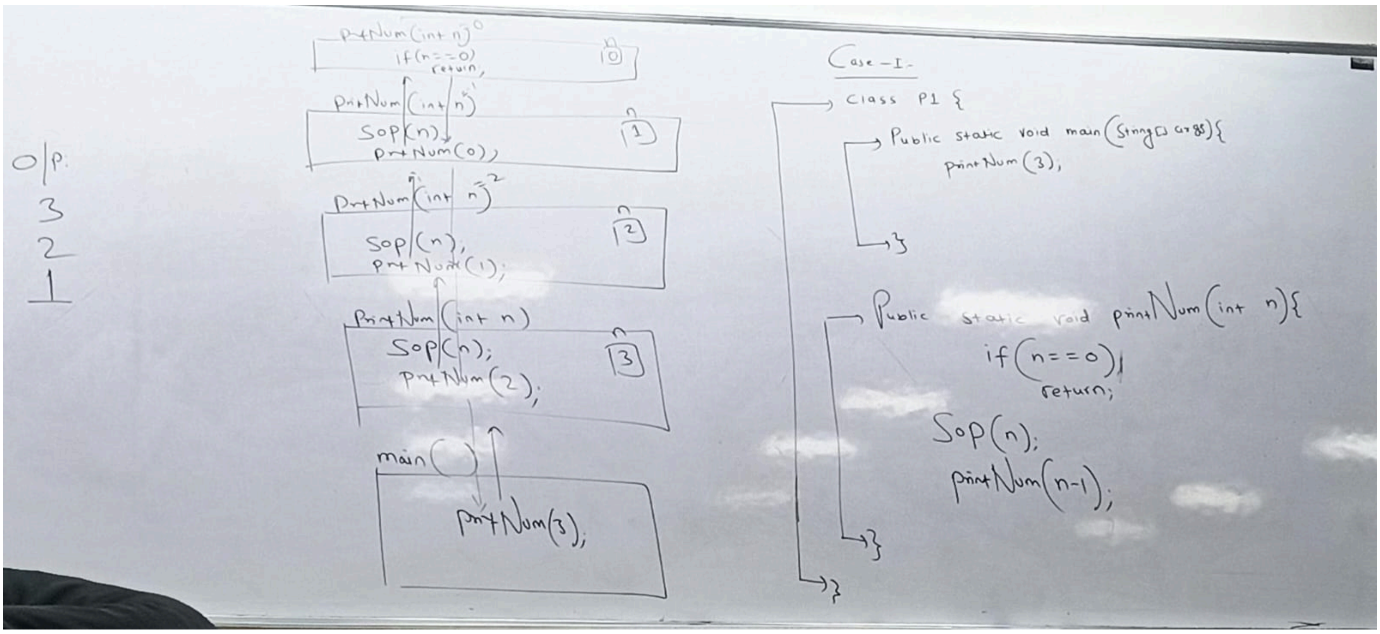
Time Complexity: O(n) **Space Complexity:** O(w) - where w is maximum width of tree

---

Traversal Cases Summary

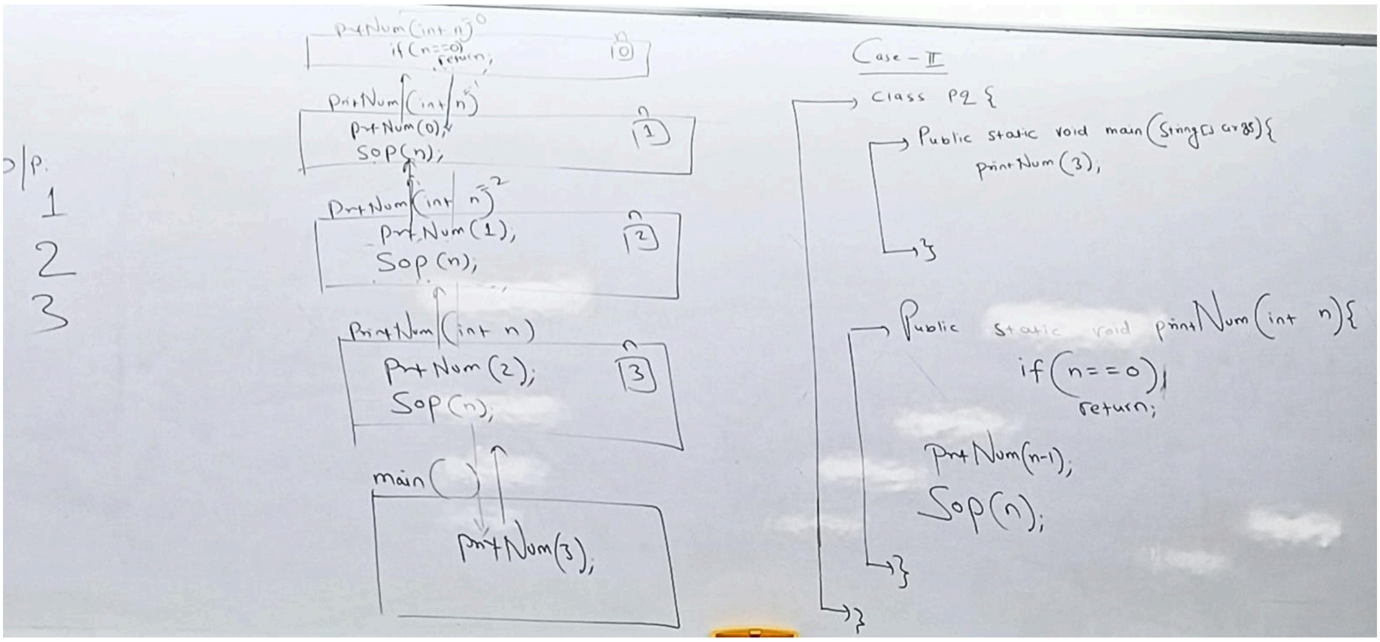
Case 1: Print First, Then Recurse (Preorder)

```java
void case1(TreeNode root) {
    if (root == null) return;
    System.out.print(root.data + " ");  // Print first
    case1(root.left);                    // Then recurse left
    case1(root.right);                   // Then recurse right
}
```



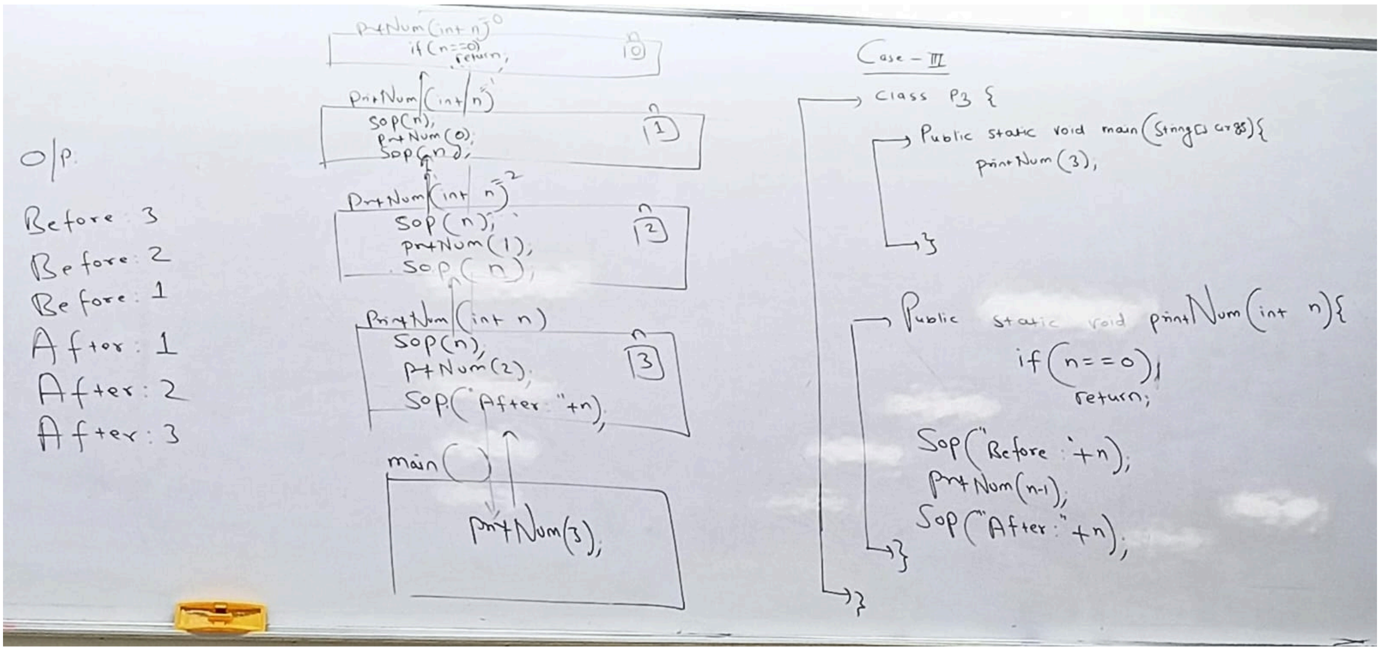
Case 2: Recurse First, Then Print (Postorder)

```java
void case2(TreeNode root) {
    if (root == null) return;
    case2(root.left);                    // Recurse left first
    case2(root.right);                   // Then recurse right
    System.out.print(root.data + " ");  // Print last
}
```



Case 3: Print Before and After Recursion (Inorder with extras)

```java
void case3(TreeNode root) {
    if (root == null) return;
    System.out.print(root.data + " ");  // Print before
    case3(root.left);                    // Recurse left
    case3(root.right);                   // Recurse right
    System.out.print(root.data + " ");  // Print after
}
```



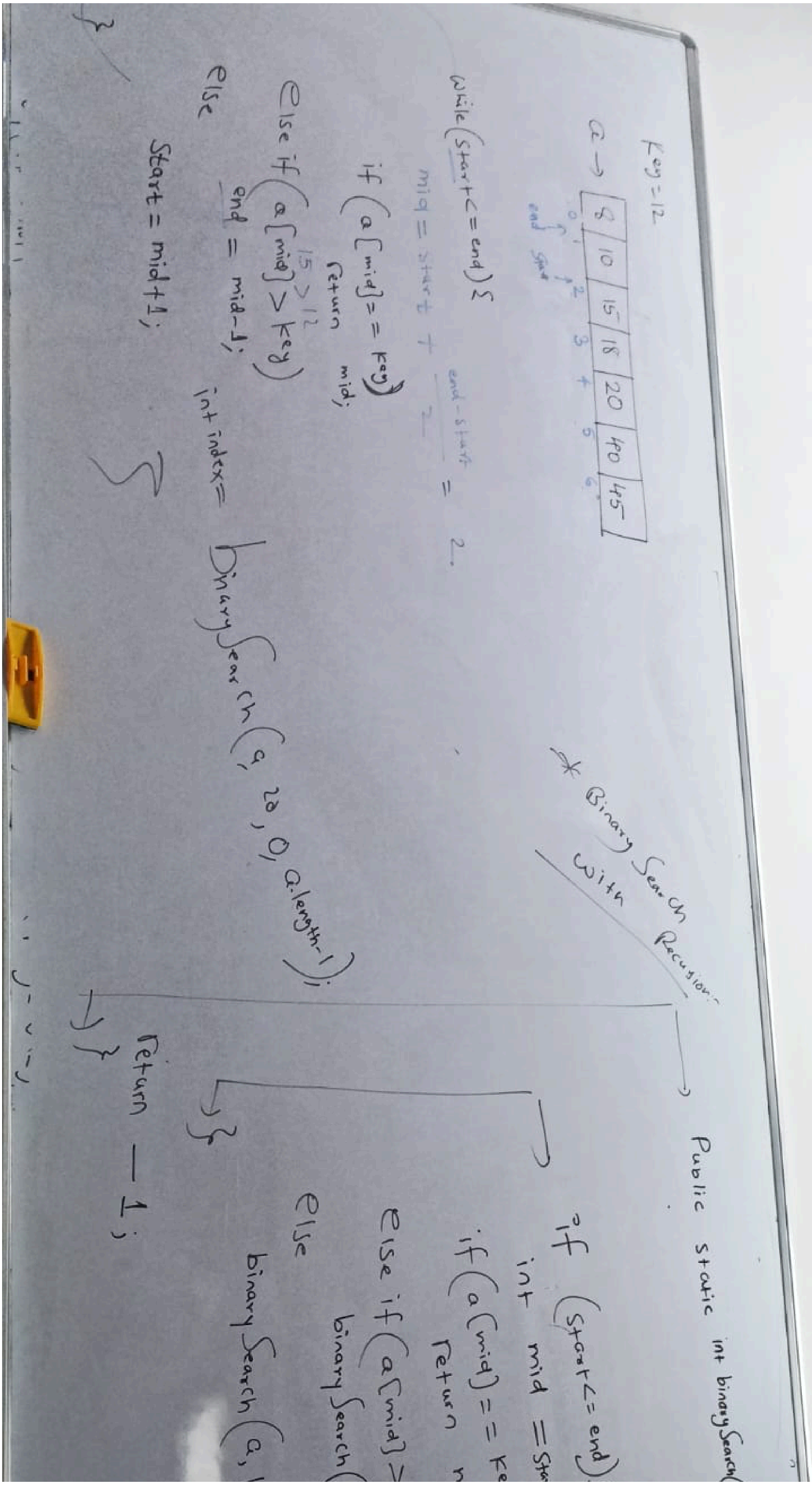
Binary Search Tree (BST)

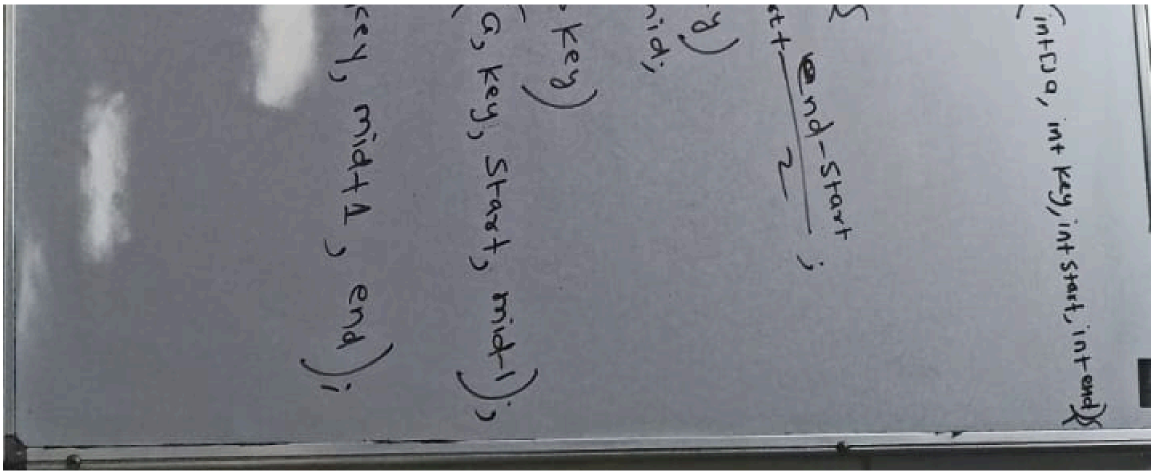
A BST is a binary tree where:

- Left subtree contains only nodes with keys less than the node's key
- Right subtree contains only nodes with keys greater than the node's key
- Both subtrees are also BSTs

Search in BST (Recursive)

```
public TreeNode search(TreeNode root, int key) {
    // Base case: root is null or key is present at root
    if (root == null || root.data == key) {
        return root;
    }

    // Key is greater than root's key
    if (key > root.data) {
        return search(root.right, key);
    }

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


Key = 12
a → 8 10 15 18 20 40 45
while(start<=end){
mid = start + (end-start)/2
if (a[mid] == key)
return mid;
else if (a[mid] > key)
end = mid-1;
else
start = mid+1;
int index= binarySearch(a, 20, 0, a.length-1);
* Binary Search With Recursion:
Public static int binarySearch(
if (start<=end)
int mid = Sta
if(a[mid] == Ke
return
else if (a[mid] >
binarySearch(
else
binarySearch(a,
return -1;

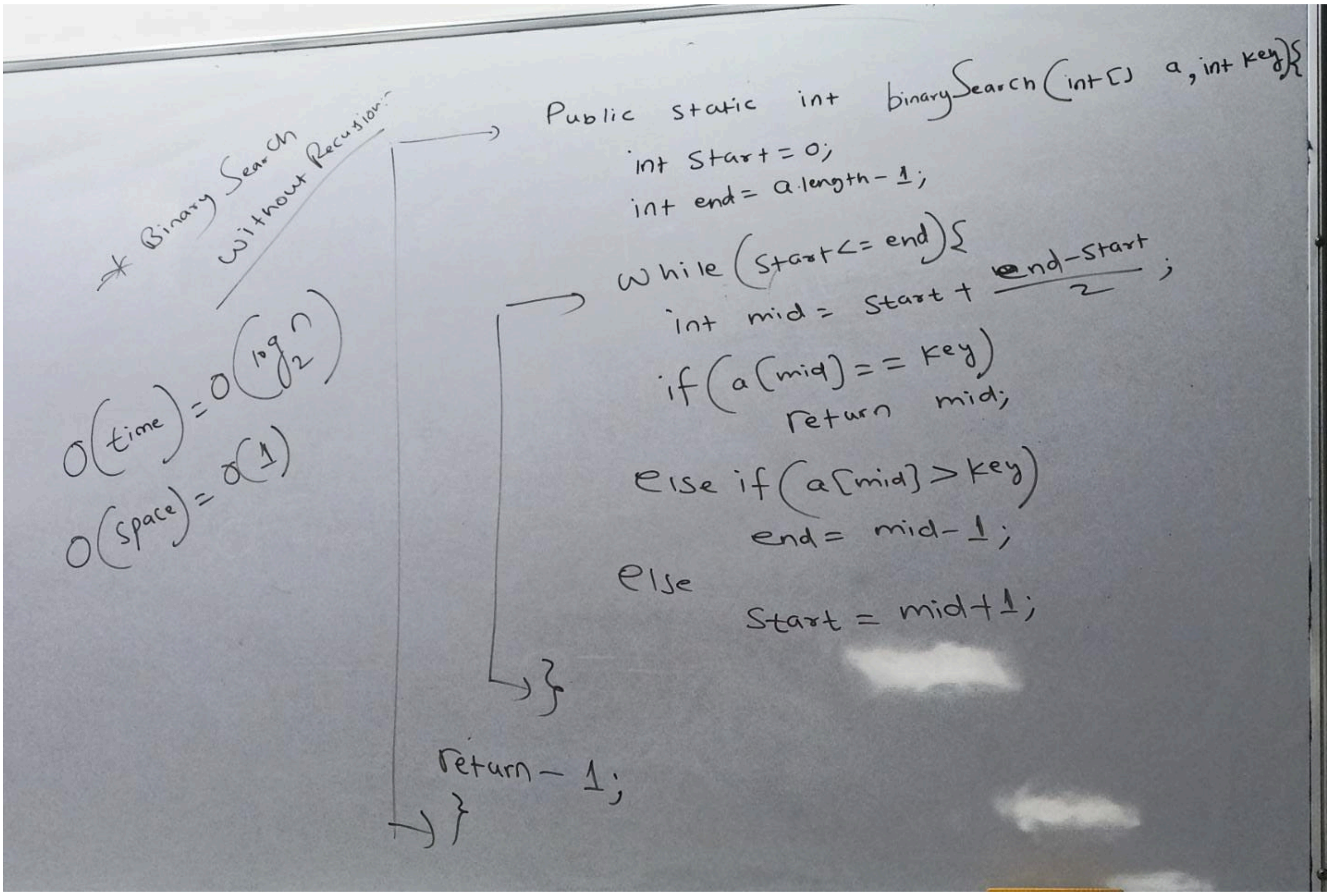


Time Complexity: O(h) - O(log n) for balanced, O(n) for skewed **Space Complexity:** O(h) - recursion stack

---

Search in BST (Iterative)

```
public TreeNode searchIterative(TreeNode root, int key) {
    TreeNode current = root;

    while (current != null && current.data != key) {
        if (key > current.data) {
            current = current.right;
        } else {
            current = current.left;
        }
    }

    return current;
}
```



Time Complexity: O(h) **Space Complexity:** O(1)

Insert in BST

```java
public TreeNode insert(TreeNode root, int key) {
    // If tree is empty, create new node
    if (root == null) {
        return new TreeNode(key);
    }

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

    // Return unchanged node pointer
    return root;
}
```

Time Complexity: O(h) **Space Complexity:** O(h) - recursion stack

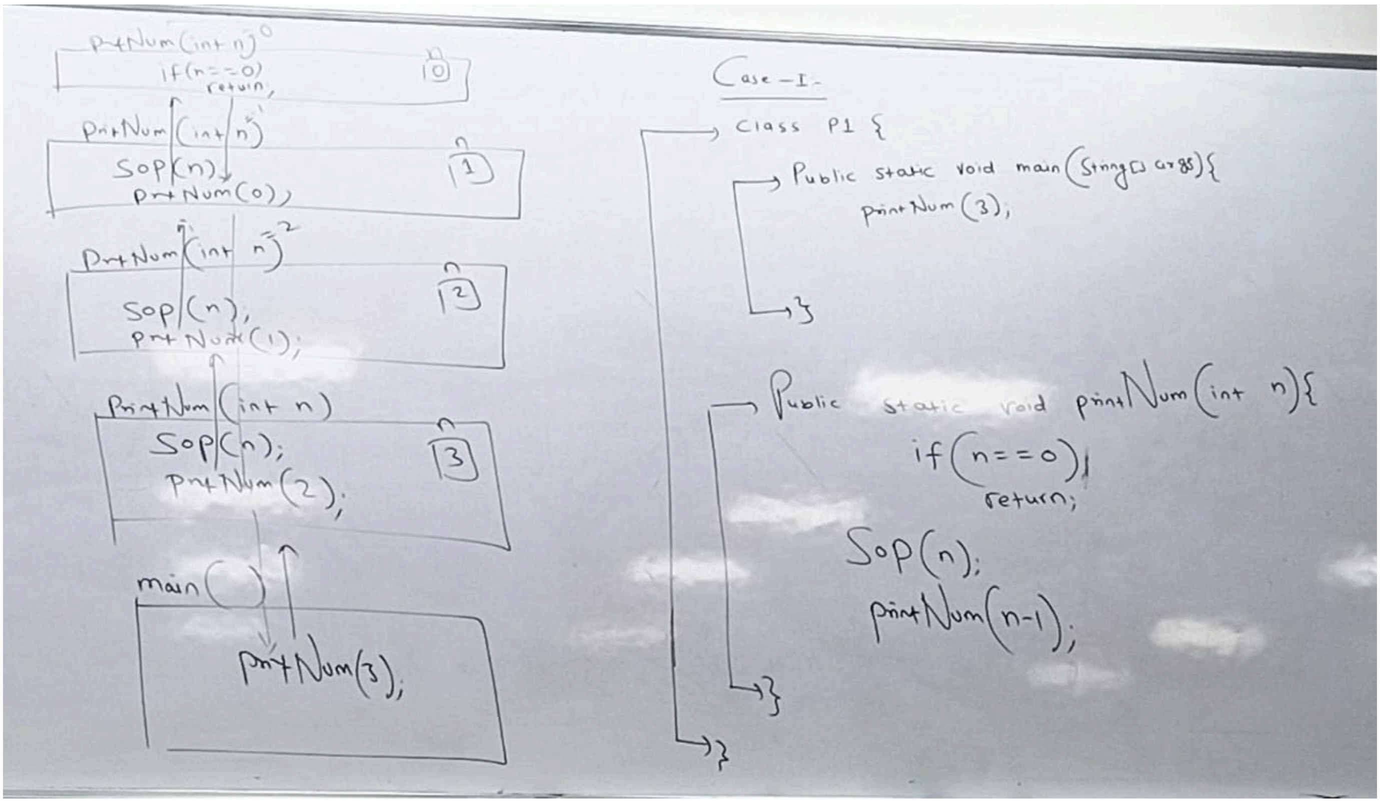
---

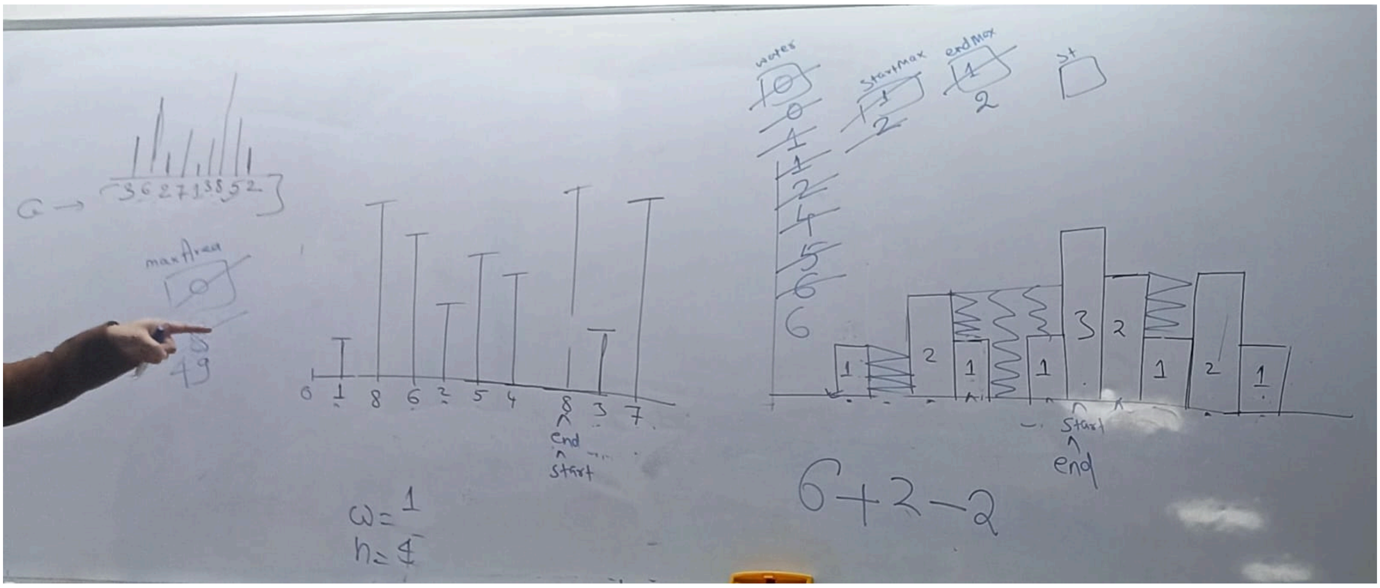
Delete in BST

```java
public TreeNode delete(TreeNode root, int key) {
    if (root == null) return null;

    // Find the node to delete
    if (key < root.data) {
        root.left = delete(root.left, key);
    } else if (key > root.data) {
        root.right = delete(root.right, key);
    } else {
        // Node with only one child or no child
        if (root.left == null) {
            return root.right;
        } else if (root.right == null) {
            return root.left;
        }

        // Node with two children: Get inorder successor
        root.data = minValue(root.right);
        root.right = delete(root.right, root.data);
    }

    return root;
}

private int minValue(TreeNode root) {
    int minValue = root.data;
    while (root.left != null) {
        minValue = root.left.data;
        root = root.left;
    }
    return minValue;
}
```

Time Complexity: O(h) **Space Complexity:** O(h)

Common Operations





Find Height of Binary Tree

```java
public int height(TreeNode root) {
    if (root == null) return -1;  // or 0 based on definition

    int leftHeight = height(root.left);
    int rightHeight = height(root.right);

    return Math.max(leftHeight, rightHeight) + 1;
}
```

Time Complexity: O(n) **Space Complexity:** O(h)

Count Total Nodes

```
public int countNodes(TreeNode root) {
    if (root == null) return 0;

    return 1 + countNodes(root.left) + countNodes(root.right);
}
```

Time Complexity: O(n) **Space Complexity:** O(h)

Check if Binary Tree is Balanced

```
public boolean isBalanced(TreeNode root) {
    return checkBalance(root) != -1;
}

private int checkBalance(TreeNode root) {
    if (root == null) return 0;

    int leftHeight = checkBalance(root.left);
    if (leftHeight == -1) return -1;

    int rightHeight = checkBalance(root.right);
    if (rightHeight == -1) return -1;

    if (Math.abs(leftHeight - rightHeight) > 1) return -1;

    return Math.max(leftHeight, rightHeight) + 1;
}
```

Time Complexity: O(n) **Space Complexity:** O(h)

Lowest Common Ancestor (LCA)

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null || root == p || root == q) {
        return root;
    }

    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);

    if (left != null && right != null) return root;

    return left != null ? left : right;
}
```

Time Complexity: O(n) **Space Complexity:** O(h)

Validate Binary Search Tree

```
public boolean isValidBST(TreeNode root) {
    return validate(root, Long.MIN_VALUE, Long.MAX_VALUE);
}

private boolean validate(TreeNode root, long min, long max) {
    if (root == null) return true;

    if (root.data <= min || root.data >= max) return false;

    return validate(root.left, min, root.data) &&
           validate(root.right, root.data, max);
}
```

Time Complexity: O(n) **Space Complexity:** O(h)

Mirror/Invert Binary Tree

```
public TreeNode invertTree(TreeNode root) {
    if (root == null) return null;

    // Swap left and right children
    TreeNode temp = root.left;
    root.left = root.right;
    root.right = temp;

    // Recursively invert subtrees
    invertTree(root.left);
    invertTree(root.right);

    return root;
}
```

Time Complexity: O(n) **Space Complexity:** O(h)

Complexity Analysis

Time Complexities Summary

Operation	BST (Balanced)	BST (Skewed)	General Binary Tree
Search	O(log n)	O(n)	O(n)
Insert	O(log n)	O(n)	O(n)
Delete	O(log n)	O(n)	O(n)
Traversal	O(n)	O(n)	O(n)
Height	O(n)	O(n)	O(n)

Space Complexities

Operation	Space Complexity	Notes
Recursive Traversal	O(h)	Recursion stack
Iterative Traversal	O(w)	Queue for level order
Search (Recursive)	O(h)	Stack space
Search (Iterative)	O(1)	No extra space

Where:

- n = number of nodes
- h = height of tree
- w = maximum width of tree

Best Practices

1. **Always check for null** before accessing node properties
2. **Use iterative approaches** when possible to save stack space
3. **For BST operations**, maintain the BST property
4. **Choose balanced trees** (AVL, Red-Black) for guaranteed O(log n) operations
5. **Use level order traversal** for problems requiring level-wise processing
6. **Consider space-time tradeoffs** when choosing recursive vs iterative

---

Common Patterns

1. **Two Pointer**: Left and right child pointers
2. **Recursion**: Most tree problems solved recursively
3. **DFS**: Preorder, Inorder, Postorder
4. **BFS**: Level order traversal using queue
5. **Backtracking**: Path finding problems
6. **Divide and Conquer**: Split problem into left and right subtrees

---

Additional Resources

- Practice platforms: LeetCode, HackerRank, GeeksforGeeks
- Key topics: Tree construction, serialization, Morris traversal
- Advanced: Segment trees, Fenwick trees, Trie

Happy Coding! 🌳