Name: Shikha Singh
Roll no: 25

Assignment No. 2

Que1. What do you mean by tree traversal technique?
Ans:Tree traversal is the process of visiting each node in a tree data structure exactly once in a systematic way to perform some operation, such as accessing, processing, or displaying node data. There are several tree traversal techniques, and they are generally categorized into two main types:

1. Depth-First Traversal (DFT):

In this technique, you go as deep as possible down one branch of the tree before backtracking. The depth-first traversals are further divided into:

- **Inorder Traversal** (Left, Root, Right):
    - Traverse the left subtree.
    - Visit the root node.
    - Traverse the right subtree.
    - Example (Binary Tree): For a tree with root node A, left child B, and right child C, the inorder traversal would be B, A, C.
- **Preorder Traversal** (Root, Left, Right):
    - Visit the root node.
    - Traverse the left subtree.
    - Traverse the right subtree.
    - Example: A, B, C.
- **Postorder Traversal** (Left, Right, Root):
    - Traverse the left subtree.
    - Traverse the right subtree.
    - Visit the root node.
    - Example: B, C, A.

2. Breadth-First Traversal (BFT):

Also called **Level-Order Traversal**, this technique visits all nodes at each level of the tree before moving to the next level. It uses a queue to keep track of the nodes to visit.

- Example: For a tree with root A, left child B, and right child C, the level-order traversal would be A, B, C.

These traversal techniques are useful in various operations such as searching, printing, and processing tree nodes in a specific order.

Que 2. Differentiate between inorder, preorder and postorder traversal?
Ans: Inorder, Preorder, and Postorder are the three depth-first tree traversal techniques, each visiting nodes in a different order. Below is a comparison of these traversal methods based on their process and applications:

1. Inorder Traversal (Left, Root, Right)

- **Process**:
  - First, visit the left subtree.
  - Then, visit the root node.
  - Finally, visit the right subtree.
- **Example**:
  - For a binary tree with root `A`, left child `B`, and right child `C`, the inorder traversal would be `B, A, C`.
- **Applications**:
  - In binary search trees (BST), inorder traversal gives nodes in **ascending order**.
  - Commonly used in expression trees to retrieve the infix expression.

2. Preorder Traversal (Root, Left, Right)

- **Process**:
  - First, visit the root node.
  - Then, traverse the left subtree.
  - Finally, traverse the right subtree.
- **Example**:
  - For the same tree with root `A`, left child `B`, and right child `C`, the preorder traversal would be `A, B, C`.
- **Applications**:
  - Used to create a **copy of a tree**.
  - Can be used to serialize a tree structure.
  - Helpful in prefix notation (Polish notation) for expressions.

3. Postorder Traversal (Left, Right, Root)

- **Process**:
  - First, visit the left subtree.
  - Then, visit the right subtree.

    - Finally, visit the root node.
- **Example**:
    - For the tree with root `A`, left child `B`, and right child `C`, the postorder traversal would be `B, C, A`.
- **Applications**:
    - Useful for **deleting** a tree, as it first removes subtrees and then the root.
    - Evaluating expression trees (used in postfix evaluation).

Key Differences:

Aspect	Inorder	Preorder	Postorder
Order of Nodes	Left → Root → Right	Root → Left → Right	Left → Right → Root
Root Node Visit	After the left subtree	Before any subtree	After both subtrees
Application	Binary search tree (sorted)	Tree copying, serialization	Deleting or evaluating trees

Que 3. Write a program to implement Binary Search Tree algorithm? Explain with the help of an example?
Ans:

Binary Search Tree (BST) Algorithm:

A **Binary Search Tree (BST)** is a binary tree where each node has at most two children. It follows the following properties:

1. The left subtree contains nodes with keys less than the root's key.
2. The right subtree contains nodes with keys greater than the root's key.
3. Both the left and right subtrees must also be binary search trees.

Operations:

The key operations performed on a BST are **Insertion**, **Search**, and **Traversal**.

Program to Implement Binary Search Tree in Java:

```
// Node class representing each node in the BST
class Node {
```

```
    int key;
    Node left, right;

    public Node(int item) {
        key = item;
        left = right = null;
    }
}

// Binary Search Tree class
class BinarySearchTree {
    Node root;

    // Constructor
    BinarySearchTree() {
        root = null;
    }

    // Method to insert a new key into the BST
    void insert(int key) {
        root = insertRec(root, key);
    }

    // Recursive function to insert a new key
    Node insertRec(Node root, int key) {
        // If the tree is empty, return a new node
        if (root == null) {
            root = new Node(key);
            return root;
        }

        // Otherwise, recurse down the tree
        if (key < root.key) {
            root.left = insertRec(root.left, key);
        } else if (key > root.key) {
```

```
        root.right = insertRec(root.right, key);
    }

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

// Method to search for a key in the BST
boolean search(int key) {
    return searchRec(root, key);
}

// Recursive search function
boolean searchRec(Node root, int key) {
    // Base cases: root is null or key is present at root
    if (root == null) {
        return false;
    }
    if (root.key == key) {
        return true;
    }

    // Key is smaller than root's key
    if (key < root.key) {
        return searchRec(root.left, key);
    }

    // Key is greater than root's key
    return searchRec(root.right, key);
}

// Inorder traversal (Left, Root, Right)
void inorder() {
    inorderRec(root);
}
```

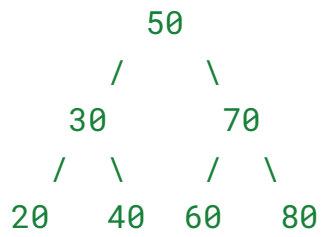
```
// Recursive inorder traversal
void inorderRec(Node root) {
    if (root != null) {
        inorderRec(root.left);
        System.out.print(root.key + " ");
        inorderRec(root.right);
    }
}

// Preorder traversal (Root, Left, Right)
void preorder() {
    preorderRec(root);
}

// Recursive preorder traversal
void preorderRec(Node root) {
    if (root != null) {
        System.out.print(root.key + " ");
        preorderRec(root.left);
        preorderRec(root.right);
    }
}

// Postorder traversal (Left, Right, Root)
void postorder() {
    postorderRec(root);
}

// Recursive postorder traversal
void postorderRec(Node root) {
    if (root != null) {
        postorderRec(root.left);
        postorderRec(root.right);
        System.out.print(root.key + " ");
```

```
        }
    }
}

public class Main {
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();

        // Inserting nodes into the BST
        bst.insert(50);
        bst.insert(30);
        bst.insert(20);
        bst.insert(40);
        bst.insert(70);
        bst.insert(60);
        bst.insert(80);

        // Printing traversals
        System.out.println("Inorder traversal:");
        bst.inorder();  // Output: 20 30 40 50 60 70 80

        System.out.println("\nPreorder traversal:");
        bst.preorder();  // Output: 50 30 20 40 70 60 80

        System.out.println("\nPostorder traversal:");
        bst.postorder(); // Output: 20 40 30 60 80 70 50

        // Searching for a key in the BST
        System.out.println("\nSearch 60: " + bst.search(60));
// Output: true
        System.out.println("Search 90: " + bst.search(90));   //
Output: false
    }
}
```

Explanation with Example:

Let's break down the code and operations using an example:

1. **Insertion**:
    - Insert the keys in the following order: `50, 30, 20, 40, 70, 60, 80`.
    - The resulting BST will look like this:



2.
3. **Traversal**:
    - **Inorder Traversal** (Left, Root, Right):
        - The output will be: `20 30 40 50 60 70 80` (sorted order).
    - **Preorder Traversal** (Root, Left, Right):
        - The output will be: `50 30 20 40 70 60 80`.
    - **Postorder Traversal** (Left, Right, Root):
        - The output will be: `20 40 30 60 80 70 50`.
4. **Search Operation**:
    - **Search 60**: The program will traverse the tree and find `60` in the right subtree of `50`, so it returns `true`.
    - **Search 90**: Since `90` is not present in the tree, it returns `false`.

Key Points:

- **Insertion** in a BST takes O(h)O(h)O(h), where hhh is the height of the tree.
- **Search** is efficient in BSTs because it compares and navigates left or right, similar to binary search, with average time complexity O(logn)O(\log n)O(logn), where nnn is the number of nodes.

Que 4. How to perform BST Deletion? Explain with the help of an example?

Ans:

Binary Search Tree (BST) Deletion:

In a Binary Search Tree (BST), deletion is more complex compared to insertion or search because we must ensure that the BST properties are maintained after the node is deleted.

There are three cases for deleting a node in a BST:

1. **Node with no children** (Leaf node).
2. **Node with one child**.
3. **Node with two children**.

Let's discuss each case in detail with an example.

Case 1: Node with No Children (Leaf Node)

If the node to be deleted is a leaf, we can simply remove it from the tree.

Case 2: Node with One Child

If the node to be deleted has one child, we can bypass the node by linking its child directly to its parent.

Case 3: Node with Two Children

If the node to be deleted has two children, we need to find the **inorder successor** (the smallest node in the right subtree) or **inorder predecessor** (the largest node in the left subtree), replace the node with the successor or predecessor, and then delete the successor or predecessor.

Program to Perform BST Deletion in Java:

```
// Node class for Binary Search Tree
class Node {
    int key;
    Node left, right;

    public Node(int item) {
        key = item;
        left = right = null;
    }
```

```
}

// Binary Search Tree class with deletion function
class BinarySearchTree {
    Node root;

    // Constructor
    BinarySearchTree() {
        root = null;
    }

    // Method to insert a new key into the BST
    void insert(int key) {
        root = insertRec(root, key);
    }

    // Recursive function to insert a new key
    Node insertRec(Node root, int key) {
        if (root == null) {
            root = new Node(key);
            return root;
        }

        if (key < root.key) {
            root.left = insertRec(root.left, key);
        } else if (key > root.key) {
            root.right = insertRec(root.right, key);
        }
        return root;
    }

    // Method to perform inorder traversal
    void inorder() {
        inorderRec(root);
    }
```

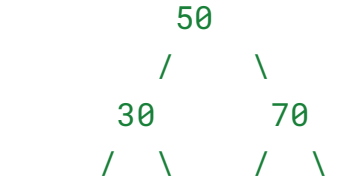
```
// Recursive function for inorder traversal
void inorderRec(Node root) {
    if (root != null) {
        inorderRec(root.left);
        System.out.print(root.key + " ");
        inorderRec(root.right);
    }
}

// Method to delete a key from the BST
void deleteKey(int key) {
    root = deleteRec(root, key);
}

// Recursive function to delete a key
Node deleteRec(Node root, int key) {
    if (root == null) return root;

    // Traverse the tree to find the node to be deleted
    if (key < root.key) {
        root.left = deleteRec(root.left, key);
    } else if (key > root.key) {
        root.right = deleteRec(root.right, key);
    } else {
        // Node to be deleted found

        // Case 1: No child (Leaf node)
        if (root.left == null && root.right == null) {
            return null;
        }

        // Case 2: One child
        if (root.left == null) {
            return root.right;
```

```
            } else if (root.right == null) {
                return root.left;
            }

            // Case 3: Two children
            // Find the inorder successor (smallest node in the
right subtree)
            root.key = minValue(root.right);

            // Delete the inorder successor
            root.right = deleteRec(root.right, root.key);
        }

        return root;
    }

    // Method to find the minimum value in a subtree (inorder
successor)
    int minValue(Node root) {
        int minVal = root.key;
        while (root.left != null) {
            minVal = root.left.key;
            root = root.left;
        }
        return minVal;
    }
}

public class Main {
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();

        // Inserting nodes into the BST
        bst.insert(50);
        bst.insert(30);
```

```
        bst.insert(20);
        bst.insert(40);
        bst.insert(70);
        bst.insert(60);
        bst.insert(80);

        System.out.println("Inorder traversal of the given
tree:");
        bst.inorder();  // Output: 20 30 40 50 60 70 80

        // Deleting node with no child (leaf node)
        System.out.println("\n\nDelete 20 (no child):");
        bst.deleteKey(20);
        bst.inorder();  // Output: 30 40 50 60 70 80

        // Deleting node with one child
        System.out.println("\n\nDelete 30 (one child):");
        bst.deleteKey(30);
        bst.inorder();  // Output: 40 50 60 70 80

        // Deleting node with two children
        System.out.println("\n\nDelete 50 (two children):");
        bst.deleteKey(50);
        bst.inorder();  // Output: 40 60 70 80
    }
}
```

Explanation with Example:

Consider a binary search tree with the following structure:



```
20   40  60   80
```

1. Deletion of a Leaf Node (No Children):

- **Delete 20**: `20` is a leaf node (no children).
    - We simply remove `20`.
    - Tree after deletion:

```
      50
    /    \
  30      70
    \    /  \
    40  60   80
```

-
- Inorder traversal after deletion: `30 40 50 60 70 80`.

2. Deletion of a Node with One Child:

- **Delete 30**: `30` has one child (`40`).
    - We remove `30` and replace it with its child `40`.
    - Tree after deletion:

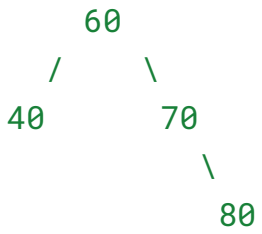
```
      50
    /    \
  40      70
         /  \
       60   80
```

-
- Inorder traversal after deletion: `40 50 60 70 80`.

3. Deletion of a Node with Two Children:

- **Delete 50**: `50` has two children (`40` and `70`).
    - To maintain the BST property, we find the **inorder successor** of `50`, which is the smallest node in the right subtree (i.e., `60`).

- Replace `50` with `60` and then delete `60` from the right subtree.
- Tree after deletion:



- Inorder traversal after deletion: `40 60 70 80`.

Key Points:

- The time complexity of deletion in a BST is **O(h)**, where `h` is the height of the tree.
- The most complex case is when the node has two children, as we need to find the inorder successor or predecessor to maintain the BST structure.

Que 5. What are Balance Tree in Advanced Data Structure?
Ans:

Balanced Trees in Advanced Data Structures

Balanced trees are a category of tree data structures that maintain a balanced height to ensure efficient operations like insertion, deletion, and search. In an unbalanced tree, the time complexity for these operations can degrade to O(n)O(n)O(n) (where nnn is the number of nodes), turning the tree into a linear data structure. To prevent this, balanced trees maintain a height close to O(logn)O(\log n)O(logn), which ensures optimal performance.

Key Characteristics of Balanced Trees:

- **Height Balance**: The height of the left and right subtrees of any node is almost equal or within a small difference (typically 1).
- **Efficiency**: All operations like searching, insertion, and deletion are guaranteed to run in O(logn)O(\log n)O(logn) time.
- **Self-Balancing**: The tree structure automatically balances itself after every insert or delete operation.

Common Types of Balanced Trees:

1. **AVL Tree (Adelson-Velsky and Landis Tree)**:
    - **Balance Condition**: The height difference (balance factor) between the left and right subtrees of any node is at most 1.
    - **Rotations**: After every insertion or deletion, the tree may perform rotations (single or double) to maintain balance.
    - **Time Complexity**: O(logn)O(\log n)O(logn) for search, insert, and delete.
    - **Use Case**: Suitable for applications requiring frequent insertions and deletions where a strict balance is needed.
2. **Red-Black Tree**:
    - **Balance Condition**: It uses a coloring mechanism (each node is either red or black), ensuring that the longest path from the root to any leaf is no more than twice the shortest path.
    - **Rotations and Recoloring**: The tree may rotate or recolor nodes after an insertion or deletion to maintain the red-black properties.
    - **Time Complexity**: O(logn)O(\log n)O(logn) for search, insert, and delete.
    - **Use Case**: Commonly used in databases and language libraries (e.g., `TreeMap`, `TreeSet` in Java) due to its more flexible balancing compared to AVL trees.
3. **B-Tree**:
    - **Balance Condition**: A multi-way balanced search tree where each node can have multiple children. It maintains balance by allowing nodes to store multiple keys, reducing tree height.
    - **Used in Databases**: B-Trees are optimized for reading and writing large blocks of data from disk and are widely used in databases and file systems (e.g., in the implementation of indexes).
    - **Time Complexity**: O(logn)O(\log n)O(logn) for search, insert, and delete.
    - **Use Case**: Excellent for applications with a large amount of disk I/O, like file systems and databases.
4. **Splay Tree**:
    - **Self-Adjusting**: A binary search tree where recently accessed elements are moved closer to the root to improve access times for frequently used elements.
    - **Time Complexity**: Amortized O(logn)O(\log n)O(logn), although individual operations may take O(n)O(n)O(n) in the worst case.
    - **Use Case**: Suitable for applications where frequently accessed elements should be quick to reach (e.g., caches).
5. **2-3 Tree**:
    - **Balance Condition**: Each node can have 2 or 3 children, and all leaves are at the same level.
    - **Operations**: Maintains balance through splitting and merging nodes during insertions and deletions.
    - **Time Complexity**: O(logn)O(\log n)O(logn) for search, insert, and delete.

- **Use Case**: Often used in file systems and databases, where balancing is crucial.

Why Balanced Trees are Important:

- **Consistent Performance**: Balanced trees ensure that all operations—search, insert, and delete—are done in logarithmic time, preventing performance degradation in large datasets.
- **Self-Balancing**: They automatically rebalance themselves after operations, so the tree remains efficient without manual intervention.
- **Memory and Disk Efficiency**: Trees like B-Trees are optimized for minimizing disk I/O operations, making them highly efficient for database indexing.

Example: AVL Tree Insertions

Let's walk through the insertion process in an AVL tree:

1. Insert elements: 10, 20, 30.

After inserting `10`:

```
 10
```

2.

After inserting `20`:

```
 10
   \
    20
```

3.
4. After inserting `30`:
    - The tree becomes unbalanced with a balance factor of 2 at node `10`.
    - To rebalance, a left rotation is performed on `10`, resulting in:

```
   20
  /  \
10    30
```

5.

The AVL tree maintains its balance automatically through rotations, ensuring that the height is kept logarithmic.

Conclusion:

Balanced trees are a fundamental component in advanced data structures, providing efficient and predictable time complexity for dynamic data management. They are used in various real-world applications, including databases, operating systems, and libraries, to optimize search, insert, and delete operations, especially for large datasets.

Que 6.What are AVL Trees and its balancing factor? Draw a tree and show how to balance it?
Ans:

AVL Trees and Balancing Factor

An **AVL tree** (Adelson-Velsky and Landis tree) is a **self-balancing binary search tree (BST)** where the difference in heights between the left and right subtrees (i.e., the balance factor) of any node is either -1, 0, or 1. This property ensures that the height of the tree remains approximately logarithmic, resulting in O(logn)O(\log n)O(logn) time complexity for search, insert, and delete operations.

Balancing Factor in AVL Tree:

- **Balancing Factor** is defined as the difference between the heights of the left and right subtrees of a node. Balance Factor (BF)=Height of Left Subtree−Height of Right Subtree\text{Balance Factor (BF)} = \text{Height of Left Subtree} - \text{Height of Right Subtree}Balance Factor (BF)=Height of Left Subtree−Height of Right Subtree
    - If **BF = 0**: The node's left and right subtrees are of equal height.
    - If **BF = 1**: The left subtree is one level taller than the right subtree.
    - If **BF = -1**: The right subtree is one level taller than the left subtree.
    - If **BF > 1** or **BF < -1**: The tree is unbalanced, and rotations are needed to restore balance.

AVL Rotations:

To restore balance after insertion or deletion in an AVL tree, **rotations** are performed. There are four types of rotations:

1. **Left Rotation (LL)**: When a node is inserted into the right subtree of a right child.

2. **Right Rotation (RR)**: When a node is inserted into the left subtree of a left child.
3. **Left-Right Rotation (LR)**: When a node is inserted into the right subtree of a left child.
4. **Right-Left Rotation (RL)**: When a node is inserted into the left subtree of a right child.

Example of AVL Tree and Balancing:

Let's walk through the insertion process with an example.

1. Inserting values: 30, 20, 10

After inserting `30`:

```
  30
```

-

After inserting `20`:

```
  30
 /
20
```

-
    - The balance factor of node `30` is 1 (left subtree height - right subtree height = 1 - 0 = 1), so it's still balanced.

After inserting `10`:

```
    30
   /
  20
 /
10
```

-
    - The balance factor of node `30` is 2 (left subtree height - right subtree height = 2 - 0 = 2). This tree is **unbalanced** because the balance factor of `30` exceeds 1.

To restore balance, perform a Right Rotation (RR) on node `30`:

Right rotation moves `20` to the root, `30` becomes the right child of `20`, and `10` remains the left child of `20`.

```
    20
   /  \
  10   30
```

-

Now the tree is balanced again, with balance factors of:

- Node `20`: Balance Factor = 0 (left height - right height = 1 - 1 = 0)
- Node `10`: Balance Factor = 0
- Node `30`: Balance Factor = 0

Step-by-Step Visualization of Balancing:

Initial Tree (before balancing):

```
   30
  /
 20
/
```

```
2. After **Right Rotation (RR)** on node `30`:
```

```
 20
 /  \
10   30
```

```
### Another Example: Left-Right (LR) Rotation
```

Now, consider inserting values: 30, 10, 20.

- After inserting `30`:

```
 30
```

- After inserting `10`:

```
 30
 /
10
```

- After inserting `20`:

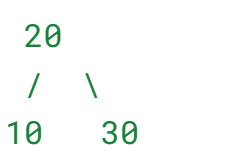
```
 30
 /
10
  \
   20
```

- The balance factor of node `30` is 2, and the balance factor of `10` is -1. This is an **unbalanced tree**.

To restore balance, we need a **Left-Right (LR) Rotation**:
1. First, perform a **Left Rotation** on node `10`:

```
 30
 /
20
```

2. Then, perform a **Right Rotation** on node `30`:



Now, the tree is balanced with balance factors of:
- Node `20`: Balance Factor = 0
- Node `10`: Balance Factor = 0
- Node `30`: Balance Factor = 0

Summary:
- **AVL Tree**: A self-balancing binary search tree where the difference in heights between the left and right subtrees of any node (balance factor) is no more than 1.
- **Balancing Factor**: Determines whether a node is balanced (0, 1, or -1) or unbalanced (greater than 1 or less than -1).
- **Rotations**: Used to restore balance after insertion or deletion.
- Right Rotation (RR)
- Left Rotation (LL)
- Left-Right Rotation (LR)
- Right-Left Rotation (RL)

Balancing ensures that AVL trees provide $O(\log n)$ time complexity for search, insert, and delete operations, making them efficient for dynamic sets of data.

Que 7. What are the rotations in AVL Tree?
Ans:

Rotations in AVL Tree

In an **AVL tree**, rotations are used to restore balance whenever the tree becomes unbalanced after an insertion or deletion. AVL trees maintain a balance factor of -1, 0, or 1 for every node, and if a node's balance factor exceeds these limits, rotations are applied to re-balance the tree.

There are **four types of rotations** in AVL trees:

1. **Left Rotation (LL)**
2. **Right Rotation (RR)**
3. **Left-Right Rotation (LR)**
4. **Right-Left Rotation (RL)**

1. Left Rotation (LL Rotation)

A **Left Rotation** is performed when a node becomes unbalanced due to the insertion of a node in the **right subtree** of its **right child**. This is also known as the **Right-Right Case**.

Example:

Let's assume the AVL tree before the left rotation looks like this:

Here, the balance factor of node `10` becomes -2 due to the insertion in the right subtree of its right child.

Left Rotation Process:

- Make `20` the new root.
- Move `10` to the left child of `20`.
- Keep `30` as the right child of `20`.

```
  20
  /  \
 10   30
```

Now, the tree is balanced again.

2. Right Rotation (RR Rotation)

A **Right Rotation** is performed when a node becomes unbalanced due to the insertion of a node in the **left subtree** of its **left child**. This is also known as the **Left-Left Case**.

Example:

Let's assume the AVL tree before the right rotation looks like this:

```
    30
    /
  20
 /
10
```

Here, the balance factor of node `30` becomes 2 due to the insertion in the left subtree of its left child.

Right Rotation Process:

- Make `20` the new root.
- Move `30` to the right child of `20`.
- Keep `10` as the left child of `20`.

```
  20
  /  \
10    30
```

Now, the tree is balanced again.

3. Left-Right Rotation (LR Rotation)

A **Left-Right Rotation** is performed when a node becomes unbalanced due to the insertion of a node in the **right subtree** of its **left child**. This is also known as the **Left-Right Case**.

Example:

Let's assume the AVL tree before the left-right rotation looks like this:

```
    30
    /
  10
    \
    20
```

Here, the balance factor of node `30` becomes 2, and the balance factor of node `10` becomes -1, causing an imbalance.

Left-Right Rotation Process:
First, perform a **Left Rotation** on the left child (`10`), making `20` the child of `30` and the parent of `10`.

```
    30
   /
 20
 /
10
```

1.

Then, perform a **Right Rotation** on `30`, making `20` the new root, `10` its left child, and `30` its right child.

```
   20
  /  \
 10   30
```

2.

Now, the tree is balanced again.

4. Right-Left Rotation (RL Rotation)

A **Right-Left Rotation** is performed when a node becomes unbalanced due to the insertion of a node in the **left subtree** of its **right child**. This is also known as the **Right-Left Case**.

Example:

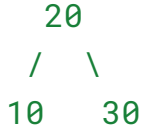
Let's assume the AVL tree before the right-left rotation looks like this:

Here, the balance factor of node `10` becomes -2, and the balance factor of node `30` becomes 1, causing an imbalance.

Right-Left Rotation Process:
First, perform a **Right Rotation** on the right child (`30`), making `20` the child of `10` and the parent of `30`.

1.

Then, perform a **Left Rotation** on `10`, making `20` the new root, `10` its left child, and `30` its right child.



2.

Now, the tree is balanced again.

Summary of Rotations:

- **LL Rotation (Right-Right Case)**: Applied when a right child of a right subtree causes an imbalance. Solution: Perform a **Left Rotation**.
- **RR Rotation (Left-Left Case)**: Applied when a left child of a left subtree causes an imbalance. Solution: Perform a **Right Rotation**.
- **LR Rotation (Left-Right Case)**: Applied when a right child of a left subtree causes an imbalance. Solution: Perform a **Left Rotation** on the child, followed by a **Right Rotation** on the parent.
- **RL Rotation (Right-Left Case)**: Applied when a left child of a right subtree causes an imbalance. Solution: Perform a **Right Rotation** on the child, followed by a **Left Rotation** on the parent.

These rotations ensure that the AVL tree remains balanced, maintaining the height as O(logn)O(\log n)O(logn) and allowing efficient search, insert, and delete operations.

Que 8. What are the properties of Red Black Tree?
Ans:**Properties of Red-Black Tree**

A **Red-Black Tree** is a type of self-balancing binary search tree with additional properties to ensure that the tree remains approximately balanced. The tree's balance ensures that operations like insertion, deletion, and searching run in O(logn)O(\log n)O(logn) time, making it efficient for dynamic data structures.

The key characteristic of a Red-Black Tree is that each node is colored either **red** or **black**. The tree satisfies a set of properties that guarantee a balanced tree structure.

Properties of a Red-Black Tree:

1. **Node Color**: Every node is either **red** or **black**.
2. **Root Property**: The **root node** is always **black**.
3. **Leaf Property**: Every leaf node (also called NIL nodes, used to represent null or empty nodes) is **black**. In some implementations, all leaves are considered **null references**, but for consistency in the tree structure, they are treated as black nodes.
4. **Red Property**: If a node is **red**, then **both** of its children must be **black**. This ensures that there are no two consecutive red nodes along any path (no red-red parent-child relationship is allowed).

5. **Black Height Property**: For each node, every path from that node to its descendant **leaf nodes** must have the **same number of black nodes**. This property ensures that the tree remains balanced because paths from the root to all leaves will have roughly the same length.
6. **Balanced Path Property**: The longest path from the root to a leaf is no more than twice the shortest path from the root to a leaf. This property ensures that the tree is approximately balanced, keeping the height close to O(logn)O(\log n)O(logn).

Example Red-Black Tree:

```
        10 (Black)
        /    \
   5 (Red)  20 (Black)
  /          /     \
1 (Black)  15 (Red)  25 (Black)
```

- In this example, all the properties are satisfied:
    - The root node `10` is black.
    - There are no consecutive red nodes.
    - Every path from a node to its leaf contains the same number of black nodes.

Benefits of Red-Black Trees:

- **Self-Balancing**: The properties of the red-black tree ensure that it remains balanced, keeping the height of the tree logarithmic in terms of the number of nodes.
- **Efficient Operations**: Insertion, deletion, and search operations can be performed in O(logn)O(\log n)O(logn) time due to the logarithmic height.
- **Rotations and Recoloring**: When an insertion or deletion violates the red-black properties, the tree is rebalanced through rotations and recoloring operations, which maintain the properties of the red-black tree.
- **Wide Use**: Red-black trees are widely used in real-world applications such as Java's `TreeMap`, `TreeSet`, and C++'s STL `map` and `set` containers, as well as database indexing systems.

Operations in Red-Black Trees:

- **Insertion**: When a new node is inserted, it is initially colored red. If this insertion violates any of the Red-Black Tree properties (like the Red Property or Root Property), rotations (left or right) and recoloring are used to restore balance.

- **Deletion**: When a node is deleted, if it was black, the deletion might cause violations of the Red-Black Tree properties. A series of rotations and recoloring operations are then performed to restore balance.

Summary:

A Red-Black Tree maintains balance by ensuring the following properties:

1. Each node is either red or black.
2. The root is always black.
3. All leaf nodes are black.
4. Red nodes must have black children (no two consecutive red nodes).
5. All paths from a node to its descendant leaves must have the same number of black nodes.

These properties help ensure that the Red-Black Tree remains approximately balanced, providing efficient performance for operations like insertion, deletion, and search.

Que 9. What are the cases for insertion in Red Black Tree?
Ans:

Cases for Insertion in a Red-Black Tree

When inserting a new node into a **Red-Black Tree**, the tree needs to maintain its properties:

1. Every node is either red or black.
2. The root is black.
3. All leaves (NIL nodes) are black.
4. Red nodes cannot have red children (no consecutive red nodes).
5. All paths from any node to its leaves contain the same number of black nodes (black height).

After inserting a node, the tree may violate these properties, particularly the **red-red violation** (consecutive red nodes). To restore the Red-Black Tree properties, several cases are checked, and corrections are applied using **recoloring** or **rotations**.

Five Cases for Insertion in Red-Black Trees:

1. **Case 1: Newly inserted node is the root**
    - If the new node is the root, simply color it **black** to satisfy the root property.
    - No further changes are needed.

```
Inserting root node: 10 (color it black)
```

2.
3. **Case 2: Parent of the new node is black**
    - If the parent of the newly inserted node is black, no properties are violated.
    - The tree remains a valid Red-Black Tree, and no rebalancing or recoloring is necessary.

```
Parent is black, no changes required.
```

4.
5. **Case 3: Parent and Uncle are both red**
    - If both the parent and the uncle of the new node are red, a **recoloring** is required.
    - Recolor the parent and the uncle to **black** and recolor the grandparent to **red**.
    - After recoloring, the grandparent might violate the Red-Black Tree properties, so we repeat the process starting from the grandparent.

Example:

```
    30 (Black)
    /    \
 20 (Red)  40 (Red)
  /
10 (Red) <-- New node
```

6.
    - Both the parent (`20`) and the uncle (`40`) are red.
    - Recolor `20` and `40` to black, and recolor `30` to red.

After recoloring:

```
    30 (Red)
    /    \
 20 (Black)  40 (Black)
  /
10 (Red)
```

7. If `30` was the root, we recolor it back to black to maintain the root property.

8. **Case 4: Parent is red but uncle is black (or NIL), and the new node is a left child of a right child (or vice versa)**
    - This is a **Left-Right (LR) or Right-Left (RL) imbalance**.
    - A **rotation** is required to fix the imbalance before recoloring. Perform the following steps:
        - Perform a **rotation** (left or right) on the parent node to make it a straight-line (left-left or right-right) case.
        - After the rotation, proceed to **Case 5** to resolve the violation.

Example (Left-Right Case):

```
    30 (Black)
    /
 10 (Red)
    \
    20 (Red) <-- New node
```

9.
    - Parent (`10`) is red, but the uncle (right child of `30`) is black (or NIL).
    - Perform a **left rotation** on node `10` to make it a Left-Left case:

```
    30 (Black)
    /
 20 (Red)
 /
10 (Red)
```

10.
    - Now apply **Case 5**.
11. **Case 5: Parent is red but uncle is black (or NIL), and the new node is a left child of a left child (or a right child of a right child)**
    - This is a **Left-Left (LL) or Right-Right (RR) imbalance**.
    - Perform a **rotation** on the grandparent to balance the tree.
    - After the rotation, recolor the parent and grandparent to maintain Red-Black Tree properties.

Example (Left-Left Case):

```
    30 (Black)
```

```
    /
 20 (Red)
 /
10 (Red) <-- New node
```

12.
    - Parent (`20`) is red, and the uncle is black.
    - Perform a **right rotation** on the grandparent (`30`):

```
    20 (Black)
    /  \
 10 (Red)  30 (Red)
```

13.
    - Now the tree is balanced, and all Red-Black Tree properties are restored.

Summary of Insertion Cases:

1. **Case 1**: The newly inserted node is the root — recolor it to black.
2. **Case 2**: The parent of the new node is black — no violations, no changes.
3. **Case 3**: The parent and uncle are red — recolor the parent and uncle to black and recolor the grandparent to red.
4. **Case 4**: The parent is red, uncle is black, and the new node is a left child of a right child or vice versa — perform a rotation (LR or RL) and proceed to Case 5.
5. **Case 5**: The parent is red, uncle is black, and the new node is a left child of a left child or right child of a right child — perform a rotation (LL or RR) and recolor.

Each of these cases ensures that the Red-Black Tree properties are preserved after insertion, and the tree remains balanced with a logarithmic height.

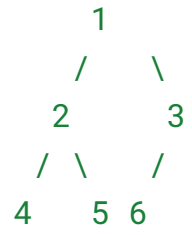
Que 10. What are Complete Binary Tree?
Ans:**Complete Binary Tree**

A **Complete Binary Tree** is a type of binary tree in which all levels are completely filled except possibly for the last level, which is filled from **left to right**. In other words, it is a binary tree where every level, except the last, has the maximum number of nodes, and all nodes at the last level are as far left as possible.

Properties of a Complete Binary Tree:

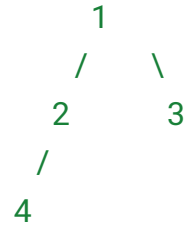
1. **All levels except the last are fully filled**: In a complete binary tree, every level before the last is filled with the maximum number of nodes.
2. **Last level is filled from left to right**: The nodes at the last level are filled in sequence from the leftmost position. If there are any missing nodes, they will be on the right side.
3. **Height of the tree**: The height hhh of a complete binary tree with nnn nodes is approximately log2(n)\log_2(n)log2(n), i.e., h≈⌊log2(n)⌋h \approx \lfloor \log_2(n) \rfloor h≈⌊log2(n)⌋.
4. **Node positions**: In an array representation of a complete binary tree:
    - The parent of a node at index iii is located at index ⌊i−12⌋\left\lfloor \frac{i-1}{2} \right\rfloor⌊2i−1⌋.
    - The left child of a node at index iii is at index 2i+12i + 12i+1.
    - The right child of a node at index iii is at index 2i+22i + 22i+2.

Example of a Complete Binary Tree:



- This tree is complete because:
    - The first and second levels are fully filled.
    - The last level (third level) is filled from left to right, and there are no gaps on the left side.

Non-Complete Binary Tree Example:



- This tree is **not complete** because the third level is not filled left to right. The node 4 is not followed by any node on the right in the third level.

Applications of Complete Binary Trees:

- **Heap data structure**: The binary heap, used for priority queues, is a type of complete binary tree.
- **Efficient storage**: Complete binary trees can be efficiently stored in arrays without using pointers to represent tree structure.

Comparison with Other Binary Trees:

- **Full Binary Tree**: A full binary tree is a tree where every node has either 0 or 2 children, i.e., no nodes have only one child.
- **Perfect Binary Tree**: A perfect binary tree is a tree in which all internal nodes have two children, and all leaves are at the same level.
- **Complete Binary Tree**: As described, it allows the last level to be incompletely filled but always filled from left to right.

Summary:

- A complete binary tree ensures that all levels except the last are fully filled, and the last level is filled from left to right.
- It has efficient storage and access properties, particularly useful in implementing data structures like heaps.

Que 11. Differentiate Between:
Binary Tree, Binary Search Tree, Complete Binary Tree

Ans:

Here's a concise comparison between **Binary Trees**, **Binary Search Trees (BST)**, and **Complete Binary Trees**:

Feature	Binary Tree	Binary Search Tree (BST)	Complete Binary Tree
Definition	A tree structure in which each node has at most two children.	A binary tree that satisfies the binary search property: the left child is less than the parent, and the right child is greater.	A binary tree where all levels are fully filled except possibly the last level, which is filled from left to right.

Node Arrangement	No specific arrangement; can have any structure.	Ordered arrangement based on values.	Specific arrangement: filled left to right.
Search Time Complexity	Varies; could be O(n)O(n)O(n) in the worst case (unbalanced).	Average case O(logn)O(\log n)O(logn), worst case O(n)O(n)O(n) for unbalanced trees.	O(logn)O(\log n)O(logn) for balanced trees; might vary for unbalanced trees.
Insertion and Deletion	Insertion and deletion do not require any specific ordering.	Insertion and deletion require re-arrangement to maintain order.	Insertion is straightforward; deletion requires maintaining completeness.
Use Cases	General purpose tree structures; used in various applications.	Efficient searching, sorting, and dynamic set operations.	Used in heap implementations and scenarios where array representation is beneficial.
Example Structure	1 / \ 2 3 / \ / \ 4 5 6 8	5 / \ 3 7 / \ / \ 1 4 6 8	1 / \ 2 3 / \ / 4 5 6

Summary of Key Differences:

1. **Binary Tree**:
    - The most general form of a tree structure with no specific arrangement of nodes.
2. **Binary Search Tree (BST)**:
    - A specialized form of a binary tree that maintains a sorted order for efficient searching, insertion, and deletion.
3. **Complete Binary Tree**:
    - A specific type of binary tree where all levels are filled except possibly the last, and the last level is filled from left to right. It is not concerned with the ordering of node values.

These distinctions make each type of tree suitable for different applications and use cases in computer science and programming.

Que 12. What are tree manipulation techniques?
Ans:

Tree manipulation techniques are algorithms and methods used to perform operations on tree data structures. These operations include traversals, insertions, deletions, searching, and rebalancing. Here's a breakdown of the common tree manipulation techniques:

1. Tree Traversal Techniques

Traversal techniques allow you to visit all the nodes in a tree in a systematic manner. There are three main types of depth-first traversal methods:

- **Inorder Traversal**:
  - Visit the left subtree, the node, and then the right subtree.
  - Result: For a binary search tree, this yields values in sorted order.

Example: For the tree:

```
  2
 / \
1   3
```

  - Inorder Traversal: `1, 2, 3`
- **Preorder Traversal**:
  - Visit the node first, then the left subtree, followed by the right subtree.
  - Useful for creating a copy of the tree.
  - **Example**: Preorder Traversal of the same tree yields: `2, 1, 3`
- **Postorder Traversal**:
  - Visit the left subtree, the right subtree, and then the node.
  - Useful for deleting the tree or evaluating expressions.
  - **Example**: Postorder Traversal yields: `1, 3, 2`
- **Level Order Traversal**:
  - Visit nodes level by level from top to bottom and left to right.
  - Implemented using a queue.
  - **Example**: Level Order Traversal yields: `2, 1, 3`

2. Insertion Techniques

Insertion involves adding a new node to the tree while maintaining the structure and properties of the tree.

- **Binary Tree Insertion**:
    - Typically involves finding the first available position in a complete binary tree (insert at the end).
- **Binary Search Tree (BST) Insertion**:
    - Start at the root and compare the value of the new node to the current node.
    - If it's less, move to the left child; if more, move to the right child.
    - Insert the new node in the first available position according to the rules.

3. Deletion Techniques

Deletion involves removing a node from the tree while maintaining the structure and properties.

- **Binary Tree Deletion**:
    - Find the node to be deleted, replace it with the last node (usually the rightmost node), and remove the last node.
- **BST Deletion**:
    - Three cases:
        - **Node with no children**: Simply remove the node.
        - **Node with one child**: Remove the node and connect its parent to its child.
        - **Node with two children**: Find the in-order predecessor or successor (the largest node in the left subtree or the smallest node in the right subtree), swap values, and then delete the predecessor or successor.

4. Searching Techniques

Searching allows you to find a specific value within the tree.

- **Binary Search in BST**:
    - Similar to insertion: start at the root and traverse left or right based on comparisons.

5. Balancing Techniques

Balancing techniques are used to maintain the balance of self-balancing trees (like AVL trees and Red-Black trees).

- **Rotations**:
    - Used in AVL trees to maintain balance after insertions or deletions.

    - There are four types of rotations: Left Rotation, Right Rotation, Left-Right Rotation, and Right-Left Rotation.

6. Tree Copying and Serialization

- **Tree Copying**: Creating a new tree that is a duplicate of the existing tree, often performed using a traversal technique.
- **Serialization**: Converting a tree structure into a linear representation (e.g., an array or a string) for storage or transmission.

7. Tree Modification

- **Updating Values**: Changing the value of a node while maintaining the tree structure.
- **Merging Trees**: Combining two trees into one, which may require balancing and restructuring.

Summary

Tree manipulation techniques are essential for effectively managing and utilizing tree data structures in computer science. These operations enable the efficient organization, retrieval, and modification of data in various applications, including databases, search engines, and memory management systems.