Revision - Trees

Introduction

A tree is a hierarchical data structure composed of nodes. Each node contains a value and a list of references (or pointers) to its child nodes. The topmost node is called the root node, and each node in the tree has zero or more child nodes. A node with no children is called a leaf node. The depth of the node is the distance of the node from the root node.

Example of a tree

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

Binary Tree

A binary tree is a type of tree in which each node has at most two child nodes, referred to as the left child and the right child.

Example

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

Traversal Methods for Binary Trees

Preorder Traversal

In preorder traversal, the current node is visited first, followed by recursively traversing the left subtree and then the right subtree.

CODE

```
Preorder(node):
    if node is not null:
        visit(node)
        Preorder(node.left)
        Preorder(node.right)
```

Inorder Traversal

In inorder traversal, the left subtree is recursively traversed first, then the current node is visited, and finally, the right subtree is traversed.

Code

```
Inorder(node):
    if node is not null:
        Inorder(node.left)
        visit(node)
        Inorder(node.right)
```

Postorder Traversal

In postorder traversal, the left subtree is recursively traversed first, then the right subtree is traversed, and finally, the current node is visited.

Code

```
Postorder(node):
    if node is not null:
        Postorder(node.left)
        Postorder(node.right)
        visit(node)
```

Level Order Traversal

Level order traversal starts from the root node and visits each level of the tree from left to right. It uses a queue to keep track of the nodes to be visited. At each step, the node at the front of the queue is dequeued and visited. Then, its left and right children, if any, are enqueued.

Code

```
LevelOrder(root):
    if root is null:
        return
    queue = create empty queue
    enqueue(root)
    while queue is not empty:
        node = dequeue()
        visit(node)
        if node has left child:
            enqueue(node.left)
        if node has right child:
            enqueue(node.right)
```

Diameter of Binary Tree

The diameter of a binary tree is the length of the longest path between any two nodes in the tree. It does not necessarily pass through the root node.

Solution Approach

The diameter of a binary tree can be calculated recursively. For each node, we calculate the height of its left subtree and the height of its right subtree. Then, we calculate the diameter recursively for the left and right subtrees. The maximum value among the following three options is considered: the sum of left and right subtree diameters plus one, the diameter of the left subtree, or the diameter of the right subtree.

Code

```
Diameter(node):
    if node is null:
        return 0
    leftHeight = Height(node.left)
    rightHeight = Height(node.right)
    leftDiameter = Diameter(node.left)
    rightDiameter = Diameter(node.right)
    return max(leftHeight + rightHeight + 1, max(leftDiameter, rightDiameter))
```

Types of Binary Trees

Full Binary Tree

A binary tree in which every node has either 0 or 2 children.

Complete Binary Tree

A binary tree in which all levels, except possibly the last one, are completely filled, and all nodes are as left as possible.

Perfect Binary Tree

A binary tree in which all interior nodes have exactly 2 children, and all leaf nodes are at the same level.

Binary Search Tree (BST)

A BST is a binary tree in which for each node, the values in the left subtree are less than the node’s value, and the values in the right subtree are greater than the node’s value.

BST supports efficient insertion, search, and deletion operations.

Searching

To search for a value in a BST, we compare the value with the current node's value. If they are equal, we have found the value. If the value is less than the current node's value, we move to the left subtree. Otherwise, we move to the right subtree. We continue this process until we find the value or reach a null position.

CODE

```
Search(root, value):
    if root is null or root.value is equal to value:
        return root
    if value < root.value:
        return Search(root.left, value)
    else:
        return Search(root.right, value)
```

Insertion

To insert a value into a BST, we recursively compare the value with each node and traverse left or right accordingly until we find an appropriate position. If the value is less than the current node's value, we move to the left subtree; if it is greater, we move to the right subtree. Once we reach a null position, we create a new node with the given value.

CODE

```
Insert(root, value):
    if root is null:
        return createNode(value)
    if value < root.value:
        root.left = Insert(root.left, value)
    else if value > root.value:
        root.right = Insert(root.right, value)
    return root
```

Deletion

To delete a value from a BST, we recursively search for the value in the tree. If the value is less than the current node's value, we move to the left subtree, if it is greater, we move to the right subtree. If we find the value, we consider three cases: no child nodes, one child node, or two child nodes. In the case of no child nodes, we simply remove the node. In the case of one child node, we replace the node with its

child. In the case of two child nodes, we find the minimum value in the right subtree (or maximum value in the left subtree), replace the node's value with it, and delete the minimum value node in the right subtree.

Code

```
Delete(root, value):
    if root is null:
        return root
    if value < root.value:
        root.left = Delete(root.left, value)
    else if value > root.value:
        root.right = Delete(root.right, value)
    else:
        if root.left is null:
            return root.right
        else if root.right is null:
            return root.left
        temp = MinimumValue(root.right)
        root.value = temp.value
        root.right = Delete(root.right, temp.value)
    return root
```

LCA in BST

To find the Lowest Common Ancestor (LCA) of two values in a BST, we start from the root and compare it with the given values. If both values are smaller than the current node's value, we move to the left subtree. If both values are greater, we move to the right subtree. If neither condition is met, it means the current node is the LCA since one value is smaller and the other is greater. We continue this process recursively until we find the LCA or reach a null position.

Code

```
LCA(root, value1, value2):
    if root is null:
        return null
    if value1 < root.value and value2 < root.value:
        return LCA(root.left, value1, value2)
    else if value1 > root.value and value2 > root.value:
        return LCA(root.right, value1, value2)
    else:
        return root
```