Advanced DSA : Trees 4: LCA

Understanding Binary Trees and Binary Search Trees

Binary Trees

A Binary Tree is a hierarchical data structure composed of nodes, where each node has at most two children: a left child and a right child. The top node is called the root, and nodes without children are called leaves.

Binary Search Trees (BSTs)

A Binary Search Tree is a type of binary tree with an additional property: for each node, all nodes in its left subtree have values smaller than the node's value, and all nodes in its right subtree have values greater than the node's value.

Problem 1 Finding the kth Smallest Element in a Binary Search Tree

Given a Binary Search Tree and a positive integer k, the problem is to find the kth smallest element in the BST.

:::warning
Please take some time to think about the brute force approach on your own before reading further.....
:::

In-Order Traversal storing elements in array(Brute Force):

Algorithm

1. Initialize a binary search tree (BST) as root.
2. Iterate through the elements of the array and insert each element into the BST.
3. Perform an in-order traversal of the BST to collect the elements in sorted order.

4. Access the Kth element from the sorted elements list.
5. Return the Kth element as the Kth smallest element.

Pseudocode:

```
// Define a TreeNode structure
Struct TreeNode:
    val
    left
    right

// Function to find the Kth smallest element in a BST
Function findKthSmallestElement(arr, k):
    If arr is empty:
        Return None  // Array is empty, no elements to find

    Root = null  // Initialize the root of the binary search tree

    // Step 1: Create a binary search tree (BST) from the array
    For each element in arr:
        Root = insert(Root, element)

    SortedElements = []  // Initialize an empty list to store sorted elements

    // Step 2: Perform an in-order traversal of the BST
    InorderTraversal(Root, SortedElements)

    // Step 3: Return the Kth element from the sorted elements list
    If 1 <= k <= length(SortedElements):
        Return SortedElements[k - 1]  // Subtract 1 because indices are 0-based
    Else:
        Return None  // Handle the case when k is out of bounds

// Function to insert a value into a BST
Function insert(root, value):
    If root is null:
        Return TreeNode(value)  // Create a new node with the given value

    If value < root.val:
        root.left = insert(root.left, value)  // Insert into the left subtree
    Else:
        root.right = insert(root.right, value)  // Insert into the right subtree

    Return root

// Function to perform an in-order traversal of the BST
```

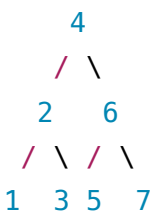
```
Function InorderTraversal(root, result):
    If root is null:
        Return

    // Step 4: Perform an in-order traversal recursively
    InorderTraversal(root.left, result)
    Append root.val to result
    InorderTraversal(root.right, result)

// Example usage:
Elements = [12, 3, 7, 15, 9, 20]
K = 3  // Find the 3rd smallest element

Result = findKthSmallestElement(Elements, K)

If Result is not null:
    Output "The Kth smallest element is: " + Result
Else:
    Output "Invalid value of K: " + K
```

In-Order Traversal Approach(Count Method):

The in-order traversal of a BST visits the nodes in ascending order. Therefore, by performing an in-order traversal and keeping track of the count of visited nodes, we can identify the kth smallest element when the count matches k.

Example: Finding the 3rd Smallest Element in a BST

BST:



Scenario:
We want to find the 3rd smallest element in the given BST.

Solution:

- Perform an in-order traversal of the BST:
- In-order traversal: 1, 2, 3, 4, 5, 6, 7
- The 3rd smallest element is 3.

Pseudocode:

Here's a simplified pseudocode representation of finding the kth smallest element using in-order traversal:

```
function findKthSmallest(root, k):
    count = 0
    stack = []

    while stack or root:
        while root:
            stack.append(root)
            root = root.left

        root = stack.pop()
        count += 1

        if count == k:
            return root.val

        root = root.right
```

Analysis:

The in-order traversal visits every node once, making the time complexity of this algorithm $O(n)$, where n is the number of nodes in the BST. The space complexity is $O(h)$, where h is the height of the BST, due to the stack used for traversal.

Problem 2 Morris Traversal

Morris Traversal Approach:

Morris Traversal takes advantage of unused null pointers in the tree structure to link nodes temporarily, effectively threading the tree. By doing so, it enables us to traverse the tree in a specific order without requiring a stack or recursion.

In-Order Morris Traversal:

- Start at the root.
- Initialize the current node as the root.
- While the current node is not null:
    - If the current node's left child is null, print the current node's value and move to the right child.
    - If the current node's left child is not null:
        - Find the rightmost node in the left subtree.
        - Make the current node the right child of the rightmost node.
        - Move to the left child of the current node.
- Repeat the process until the current node becomes null.

Pre-Order Morris Traversal:

- Start at the root.
- Initialize the current node as the root.
- While the current node is not null:
    - Print the current node's value.
    - If the current node's left child is null, move to the right child.
    - If the current node's left child is not null:
        - Find the rightmost node in the left subtree.
        - Make the current node the right child of the rightmost node.
        - Move to the left child of the current node.
- Repeat the process until the current node becomes null.

Example:

We will carefully go through each step:

- **Step 1: Start at the root node, which is 1.**
    i. Initialize current pointer as current = 1.
- **Step 2: At node 1:**
    i. Check if the left subtree of the current node is null.

ii. Since the left subtree of 1 is not null, find the rightmost node in the left subtree. This is node 5.
iii. Create a thread (temporary link) from 5 to the current node (1): 5 -> 1.
iv. Update the current node to its left child: current = 2.

- **Step 3: At node 2:**
  i. Check if the left subtree of the current node is null.
  ii. The left subtree of 2 is not null, so find the rightmost node in the left subtree of 2, which is 5.
  iii. Remove the thread from 5 to 1 (undoing the link created earlier).
  iv. Print the current node's value, which is 2.
  v. Move to the right child of the current node: current = 3
- **Step 4: At node 3:**
  i. Check if the left subtree of the current node is null.
  ii. The left subtree of 3 is null, so print the current node's value, which is 3.
  iii. Move to the right child of the current node (null): current = None.
- **Step 5:** Since the current node is now None, we've reached the end of the traversal.
  i. The Morris Inorder Traversal of the binary tree 1 -> 2 -> 4 -> 5 -> 3 allows us to visit all the nodes in ascending order without using additional data structures or modifying the tree's structure. It's an efficient way to perform an inorder traversal.

Pseudocode Example (In-Order):

```
function morrisInOrderTraversal(root):
    current = root
    while current is not null:
        if current.left is null:
            print current.value
            current = current.right
        else:
            pre = current.left
            while pre.right is not null and pre.right != current:
                pre = pre.right
            if pre.right is null:
                pre.right = current
                current = current.left
            else:
                pre.right = null
                print current.value
                current = current.right
```

Analysis:

Morris Traversal eliminates the need for an explicit stack, leading to a constant space complexity of $O(1)$. The time complexity for traversing the entire tree remains $O(n)$, where n is the number of nodes.

Question

What is the primary advantage of Morris Traversal for binary trees?

Choices

- [ ] It uses an auxiliary stack to save memory.
- [ ] It guarantees the fastest traversal among all traversal methods.
- [ ] It allows for traversal in reverse order (right-to-left).
- [x] It achieves memory-efficient traversal without using additional data structures.

Problem 3 Finding an element

Approach:

Finding an element in a binary tree involves traversing the tree in a systematic way to search for the desired value. We'll focus on a common approach known as depth-first search (DFS), which includes pre-order, in-order, and post-order traversal methods.

Algorithm:

1. Start at the root node of the binary tree.
2. If the root node is null (indicating an empty tree), return False (element not found).
3. Check if the value of the current node matches the target value:
4. If they are equal, return True (element found).
5. Recursively search for the target element in the left subtree by calling the function with the left child node.
6. Recursively search for the target element in the right subtree by calling the function with the right child node.
7. If the element is found in either the left or right subtree (or both), return True.
8. If the element is not found in either subtree, return False.

Example:

Dry Run:

1. Start at the root (1).
2. Check if it matches the target (3) - No.
3. Move to the left child (2).
4. Check if it matches the target (3) - No.
5. Move to the left child (4).
6. Check if it matches the target (3) - No.
7. Move to the right child (null).
8. Move back to 4's parent (2).
9. Move to the right child (5).
10. Check if it matches the target (3) - No.
11. Move to the left child (null).
12. Move back to 5's parent (2).
13. Move back to 2's parent (1).
14. Check if it matches the target (3) - Yes, found!
15. Finish the search.

Pseudocode Example:

```
function findElement(root, target)
    if root is null
        return False  // Element not found in an empty tree

    if root.value is equal to target
        return True   // Element found at the current node

    // Recursively search in the left subtree
    found_in_left = findElement(root.left, target)

    // Recursively search in the right subtree
    found_in_right = findElement(root.right, target)

    // Return True if found in either left or right subtree
    return found_in_left OR found_in_right
```

Analysis:

The time complexity of finding an element in a binary tree using DFS depends on the height of the tree. In the worst case, it's O(n), where n is the number of nodes in the tree. The space complexity is determined by the depth of the recursion stack.

Problem 4 Path from root to node in Binary Tree

:::warning
Please take some time to think about the solution approach on your own before reading further.....
:::

Approach:

To find the path from the root to a specific node, we'll leverage depth-first search (DFS), a versatile traversal method that includes pre-order, in-order, and post-order traversal techniques.

DFS Pre-Order Traversal for Finding Path:

1. Initialize an empty list path to store the path.
2. Define a recursive function findPath(root, target, path):
3. If root is null, return False.

4. If root matches the target, append it to path and return True.
5. Recursively call findPath on the left subtree and right subtree.
6. If either subtree returns True, append root to path and return True.
7. Start the search from the root node by calling findPath(root, target, path).
8. If the search returns True, reverse the path list to get the path from root to target.
9. Return the reversed path.

Pseudocode Example (DFS Pre-Order):

```
function findPath(root, target):
    if root is null:
        return false  // Element not found in an empty tree

    if root.value == target:
        list.add(root)  // Add the current node to the list (path found)
        return true;    // Element found

    res = findPath(root -> left, target) OR findPath(root -> right, target)

    if res == true:
        list.add(root)  // Add the current node to the list (part of the path)

    return res;

// Reverse the list to get the answer (the path from root to target)
```

Analysis:

The time complexity of finding the path from the root to a node using DFS depends on the height of the tree. In the worst case, it's O(n), where n is the number of nodes in the tree. The space complexity is determined by the depth of the recursion stack and the length of the path.

Question

What is the primary benefit of using depth-first search (DFS) for finding the path from the root to a specific node in a Binary Tree?

Choices

- [ ] DFS guarantees the shortest path between the root and the target node.
- [ ] DFS ensures that the tree remains balanced during traversal.
- [ ] DFS enables efficient path finding with a time complexity of O(log n).
- [x] DFS allows us to explore the structure of the tree while tracking visited nodes.

Problem 5 finding the Lowest Common Ancestor (LCA) of two nodes

Approach:

To find the LCA of two nodes in a binary tree, we'll utilize a recursive approach that capitalizes on the tree's structure. The LCA is the deepest node that has one of the nodes in its left subtree and the other node in its right subtree.

Recursive Algorithm for LCA:

- Start at the root of the binary tree.
- If the root is null or matches either of the target nodes, return the root as the LCA.
- Recursively search for the target nodes in the left and right subtrees of the current root.
- If both target nodes are found in different subtrees, the current root is the LCA.
- If only one target node is found, return that node as the LCA.
- If both target nodes are found in the same subtree, continue the search in that subtree.

Example:

LCA of nodes 6 and 3 is node 1.

1. Start at the root (1).
2. Check for nodes 6 and 3 in the subtree rooted at 1.
3. Recursively search the left subtree (2).
4. Continue searching left (4).

5. Continue searching left (6).
6. Found node 6, but not node 3 in the left subtree.
7. Go back to node 4 and check the right subtree (null).
8. Go back to node 2 and check the right subtree (5).
9. Continue searching right (5).
10. Not found node 6 in the right subtree.
11. Go back to node 2 and check for nodes 6 and 3.
12. Found node 6 in the left subtree.
13. Return node 2 as the Lowest Common Ancestor (LCA).

Pseudocode Example:

```
function findLCA(root, node1, node2):
    # Base case: if root is null or matches either of the nodes, return root
    if root is null or root == node1 or root == node2:
        return root

    # Recursively search for the target nodes in the left and right subtrees
    left_lca = findLCA(root.left, node1, node2)
    right_lca = findLCA(root.right, node1, node2)

    # Determine the LCA based on the search results
    if left_lca and right_lca:
        return root  # Current root is the LCA
    if left_lca:
        return left_lca  # LCA found in the left subtree
    return right_lca  # LCA found in the right subtree
```

Analysis:

The time complexity of finding the LCA in a binary tree using this recursive approach is O(n), where n is the number of nodes in the tree. The space complexity is determined by the depth of the recursion stack.

Problem 6 Lowest Common Ancestor (LCA) in a Binary Search Tree (BST)

Approach:

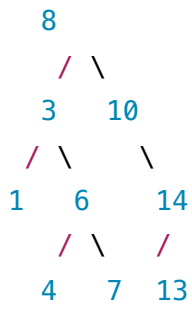
To find the LCA of two nodes in a Binary Search Tree, we'll utilize the properties of BSTs that make traversal and comparison more efficient.

Algorithm for Finding LCA in a BST:

- Start at the root of the BST.
- Compare the values of the root node, node1, and node2.
- If both nodes are smaller than the root's value, move to the left subtree.
- If both nodes are larger than the root's value, move to the right subtree.
- If one node is smaller and the other is larger than the root's value, or if either node matches the root's value, the root is the LCA.
- Repeat steps 2-5 in the chosen subtree until the LCA is found.

Example: Finding LCA in a Binary Search Tree

BST:



Let's find the LCA of nodes 4 and 7 in this BST:

- **Step 1:**
  i. Start at the root, which is node 8.
  ii. Compare node 4 and node 7 with the current node's value (8).
  iii. Both are smaller, so move to the left subtree (node 3).
- **Step 2:**
  i. Move to node 3.
  ii. Compare node 4 and node 7 with the current node's value (3).
  iii. Both are larger, so move to the right subtree (node 6).

- **Step 3:**
    i. Move to node 6.
    ii. Compare node 4 and node 7 with the current node's value (6).
    iii. Node 4 is smaller, and node 7 is larger.
    iv. The current node (6) is the LCA of nodes 4 and 7.
    v. So, in this example, the LCA of nodes 4 and 7 in the BST is node 6.

Pseudocode Example:

```
function findLCA(root, node1, node2):
    if root is null:
        return null  // If the tree is empty, there's no LCA

    while root is not null:
        // If both nodes are smaller than the current node, go left
        if node1.value < root.value and node2.value < root.value:
            root = root.left
        // If both nodes are larger than the current node, go right
        else if node1.value > root.value and node2.value > root.value:
            root = root.right
        // If one node is smaller and the other is larger, or if one matches, this is t
        else:
            return root

    return null  // If no LCA is found (unlikely in a valid BST)
```

Analysis:

The time complexity of finding the LCA in a BST is O(h), where h is the height of the BST. In a balanced BST, the height is log(n), making the LCA operation highly efficient. The space complexity is determined by the depth of the recursion stack.

Problem 7 In-time and Out-time of Binary Tree

Approach:

The Interval Assignment technique involves three main steps: DFS traversal, interval assignment, and construction of the rooted tree.

:::warning
Please take some time to think about the further solution approach on your own before reading further.....
:::

DFS Traversal and Interval Assignment:

- Start a DFS traversal of the tree from any chosen starting node.
- As nodes are visited, assign start times when a node is entered and finish times when the traversal returns from that node. These times define intervals for each node.

Constructing the Rooted Tree:

1. From the DFS traversal, we have a collection of intervals (start and finish times) for each node.
2. Choose the node with the smallest start time as the root of the rooted tree.
3. For each remaining node:
    i. Find the node with the largest start time that is still smaller than the current node's finish time. This node becomes the parent of the current node in the rooted tree.
    ii. Repeat this process for all nodes.

Question

What is the significance of in-time and out-time values in DFS traversal?

Choices

- [ ] They indicate the number of times each node is visited during the traversal.
- [ ] They represent the depth of each node in the tree.
- [x] They help create hierarchical visualizations of trees.
- [ ] They are used to determine the balance of the tree.

Example:

1. We initialize the global time variable to 1.
2. We traverse the tree using Depth-First Search (DFS):
3. Starting at Node 1:
4. In-Time for Node 1 is recorded as 1.
5. We recursively visit the left child, Node 2.
6. At Node 2:
7. In-Time for Node 2 is recorded as 2.
8. We recursively visit the left child, Node 4.
9. At Node 4:
10. In-Time for Node 4 is recorded as 3.
11. Since Node 4 has no further children, we record its Out-Time as 5.
12. Now, we return to Node 2:
13. We recursively visit the right child, Node 5.
14. At Node 5:
15. In-Time for Node 5 is recorded as 8.
16. Since Node 5 has no further children, we record its Out-Time as 10
17. We return to Node 2 and record its Out-Time as 11.
18. We return to Node 1 and recursively visit its right child, Node 3.
19. At Node 3:
20. In-Time for Node 3 is recorded as 12.
21. We recursively visit its right child, but it's null.
22. We record the Out-Time for Node 3 as 14.
23. Finally, we return to Node 1 and record its Out-Time as 15.

The in-time and out-time values are now calculated:

- Node 1 - In-Time: 1, Out-Time: 15
- Node 2 - In-Time: 2, Out-Time: 11
- Node 3 - In-Time: 12, Out-Time: 14
- Node 4 - In-Time: 3, Out-Time: 5
- Node 5 - In-Time: 8, Out-Time: 10

Pseudocode

```
function calculateInTimeOutTime(root):
    global time  // A global variable to keep track of time

    // Initialize arrays to store in-time and out-time for each node
    inTime = [0] * (2 * n)  // Assuming 'n' is the number of nodes in the tree
    outTime = [0] * (2 * n)

    // Helper function for DFS traversal
    function dfs(node):
        nonlocal time

        // Record the in-time for the current node and increment time
        inTime[node] = time
        time = time + 1

        // Recursively visit left child (if exists)
        if node.left is not null:
            dfs(node.left)

        // Recursively visit right child (if exists)
        if node.right is not null:
            dfs(node.right)

        // Record the out-time for the current node and increment time
        outTime[node] = time
        time = time + 1

    // Start DFS traversal from the root
    dfs(root)
```

Problem 8 For multiple queries find LCA(x,y)

Algorithm:

1. Calculate In-Time and Out-Time for Each Node:
2. First, calculate the in-time and out-time for each node in the binary tree as explained in a previous response.

3. Answer LCA Queries:
4. To find the LCA of multiple pairs of nodes (x, y):
5. For each LCA query (x, y):
6. Check if inTime[x] is less than or equal to inTime[y] and outTime[x] is greater than or equal to outTime[y]. If true, it means that node x is an ancestor of node y.
7. Check if inTime[y] is less than or equal to inTime[x] and outTime[y] is greater than or equal to outTime[x]. If true, it means that node y is an ancestor of node x.
8. If neither of the above conditions is met, it means that x and y have different ancestors.
9. In such cases, move up the tree from the deeper node until you find a node that is at the same level as the shallower node. This node will be their LCA.

Example

And we'll find the Lowest Common Ancestor (LCA) for a few pairs of nodes (x, y) using the in-time and out-time approach.

- **Step 1: Calculate In-Time and Out-Time**
  We've already calculated the in-time and out-time values for this tree as follows:
  i. Node 1 - In-Time: 1, Out-Time: 10
  ii. Node 2 - In-Time: 2, Out-Time: 7
  iii. Node 3 - In-Time: 8, Out-Time: 9
  iv. Node 4 - In-Time: 3, Out-Time: 4
  v. Node 5 - In-Time: 5, Out-Time: 6
- **Step 2: Find LCA for Pairs**
  - Find LCA(4, 5):
  - Check in-time and out-time:
  - In-Time(4) <= In-Time(5) and Out-Time(4) >= Out-Time(5) is true.
  - So, LCA(4, 5) is 4.
  - Find LCA(2, 3):
  - Check in-time and out-time:
  - In-Time(2) <= In-Time(3) and Out-Time(2) >= Out-Time(3) is false.
  - Now, bring both nodes to the same depth:
  - Move 2 up once: 2 is now at the same depth as 3.

- Continue moving both nodes up:
- LCA(2, 3) is 1.
- Find LCA(4, 3):
- Check in-time and out-time:
- In-Time(4) <= In-Time(3) and Out-Time(4) >= Out-Time(3) is false.
- Now, bring both nodes to the same depth:
- Move 4 up once: 4 is now at the same depth as 3.
- Continue moving both nodes up:
- LCA(4, 3) is 1.
- Find LCA(5, 2):
- Check in-time and out-time:
- In-Time(5) <= In-Time(2) and Out-Time(5) >= Out-Time(2) is false.
- Now, bring both nodes to the same depth:
- Move 5 up once: 5 is now at the same depth as 2.
- Continue moving both nodes up:
- LCA(5, 2) is 1.

Pseudocode:

```
function findLCA(x, y):
    if inTime[x] <= inTime[y] and outTime[x] >= outTime[y]:
        return x  # x is an ancestor of y

    if inTime[y] <= inTime[x] and outTime[y] >= outTime[x]:
        return y  # y is an ancestor of x

    # Move x and y up the tree to the same depth
    while depth[x] > depth[y]:
        x = parent[x]

    while depth[y] > depth[x]:
        y = parent[y]

    # Move x and y up simultaneously until they meet at the LCA
    while x != y:
        x = parent[x]
        y = parent[y]

    return x  # LCA found
```

Question

What is the primary purpose of constructing a rooted tree using the start and finish times obtained during the DFS traversal?

Choices

- [ ] To optimize the tree structure for faster traversal.
- [ ] To visualize the tree with nodes arranged in increasing order.
- [x] To efficiently represent the hierarchy and relationships within the tree.
- [ ] To eliminate the need for recursion in tree traversal.

Observations

- **In-Order Traversal:**
  It visits Binary Search Tree (BST) nodes in ascending order, enabling efficient kth smallest element retrieval.
- **Morris Traversal:**
  An efficient memory-saving tree traversal method with O(1) space complexity.
- **Path from Root:**
  DFS traversal is used to find the path from the root to a node, with space complexity tied to recursion depth.
- **Lowest Common Ancestor (LCA) in Tree:**
  LCA is found through recursion with O(n) time complexity and stack space.
- **In-Time & Out-Time:**
  These values in DFS help create hierarchical visualizations of trees.
- **Interval Assignment Visualization:**
  Provides a visual hierarchy for analyzing complex structures in various fields.
- **Finding LCA for Multiple Queries:**
  LCA retrieval for multiple pairs involves adjusting node depths until they meet.