Data Structures and Algorithm
Assignment-3(Report)

Aditya Prasad
2022036

Q1)

(a) Postorder and Level-order

Not possible

● Level-order traversal does not reveal the structure inside a level; it just reveals the order of nodes at each level.

● Postorder goes through the root, left, and right subtrees. However, for nodes at the same level, you cannot uniquely infer the left and right subtrees from the postorder sequence without knowing the inorder, which indicates the relative order of nodes.

(b) Inorder and Preorder

Possible

● From left to right, inorder traversal provides the distinct order of nodes.

● Preorder shows you the root, followed by the left and right subtrees.

● You can reconstruct the left and right subtrees in a unique way if you know the relative order from inorder and the root from preorder.

Pseudo Code:

```
function buildTree(inOrder, preOrder, inStart, inEnd)
    static preIndex = 0
    if inStart > inEnd
        return NULL
    newNode = Node(preOrder[preIndex])
    preIndex++
    if inStart == inEnd
        return newNode
    inIndex = findIndex(inOrder, newNode.data, inStart, inEnd)
    newNode.left = buildTree(inOrder, preOrder, inStart, inIndex - 1)
    newNode.right = buildTree(inOrder, preOrder, inIndex + 1, inEnd)
    return newNode

function findIndex(arr, value, start, end)
    for i from start to end
        if arr[i] == value
            return i
```

(c) Inorder and Level-order

Possible

- Each level's root nodes may be found using the level order. Next, we can extract the partitioning node that is the root from the level order and split the inorder into two halves.

Pseudo Code:

```
function extractKeys(inOrder, levelOrder, start, end, n)
    extractedKeys = new array
    for i from 0 to n-1
        if search(inOrder, start, end, levelOrder[i]) != -1
            append levelOrder[i] to extractedKeys
    return extractedKeys

function search(arr, start, end, value)
    for i from start to end
        if arr[i] == value
            return i
    return -1

function buildTree(inOrder, levelOrder, inStart, inEnd, n)
    if inStart > inEnd
        return null
    root = new Node(levelOrder[0])
    if inStart == inEnd
        return root
    index = search(inOrder, inStart, inEnd, root.key)
    leftKeys = extractKeys(inOrder, levelOrder, inStart, index, index - inStart)
    rightKeys = extractKeys(inOrder, levelOrder, index + 1, inEnd, inEnd - index)
    root.left = buildTree(inOrder, leftKeys, inStart, index - 1, index - inStart)
    root.right = buildTree(inOrder, rightKeys, index + 1, inEnd, inEnd - index)

    delete leftKeys
    delete rightKeys
    return root
```

(d) Inorder and Postorder

Possible

● Postorder visits left, right, and then root; inorder gives the distinct order left, root, right.

● Divide the inorder array in half starting with the final element of the postorder; this is known as the left subtree (all items left to the root in the inorder) and the right subtree. In a similar manner, one can continue in the following subtrees in order to obtain n unique trees.

● The tree can be rebuilt uniquely using the relative order and "visit after subtrees" information.

Pseudo Code:

```
function buildTree(inOrder, postOrder, inStart, inEnd)
    static postIndex = length(postOrder) - 1
    if inStart > inEnd
        return NULL
    newNode = Node(postOrder[postIndex])
    postIndex--
    if inStart == inEnd
        return newNode
    inIndex = search(inOrder, inStart, inEnd, newNode.data)
    newNode.left = buildTree(inOrder, postOrder, inStart, inIndex - 1)
    newNode.right = buildTree(inOrder, postOrder, inIndex + 1, inEnd)
    return newNode
```

(e) Postorder and Preorder

Not Possible

The preorder traverses the root, left subtree, and right subtree, while the postorder traverses the left subtree, right subtree, and root. When two distinct structures have identical items in both the left and right subtrees, there is ambiguity. Based only on these traversals, you cannot conclude whether subtree—the left or the right—was visited first.

Q2)

This C++ code defines functions to create a binary tree from an input array representation, traverse the tree using inorder traversal, create an undirected graph from the tree, and perform breadth-first search (BFS) traversal starting from a specified node.

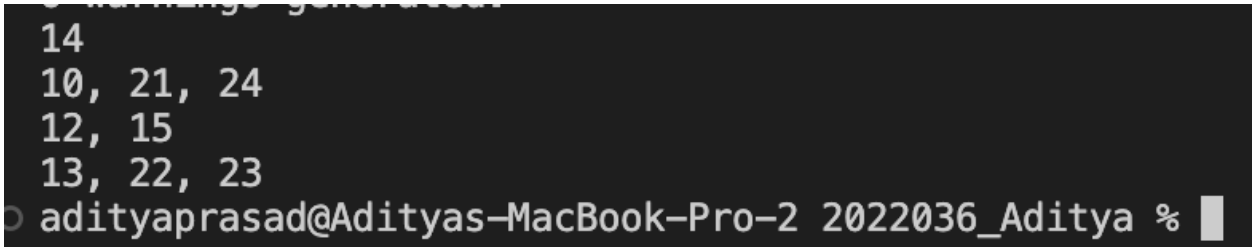
The `TreeNode` struct represents nodes in the binary tree, and the `createTree` function constructs the tree from an input array where each element represents a node's value or "N" for null nodes. Inorder traversal is implemented using a stack.

The code also contains functions to create an undirected graph from the tree structure (`createGraph`), where each node in the tree becomes a vertex in the graph, and BFS traversal (`bfs`) to visit nodes level by level starting from a given node.

The `max_value` function calculates the maximum value in the input array, and `initialise_graph` initializes the graph with the appropriate size based on the maximum value found in the input array.

In the `main` function, an input string representing a binary tree is parsed into an array, the tree is created, and then functions for graph creation and BFS traversal are called. The BFS traversal is initiated from node 14.

OUTPUT:



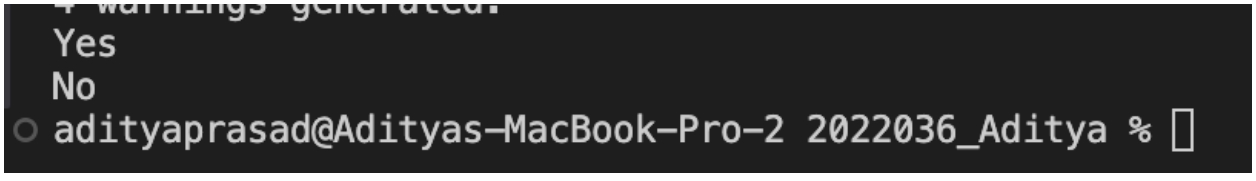
Q3)

a)When determining if two nodes in a binary tree are relatives, the best solution can be found in O(n) time complexity, where n is the total number of nodes in the tree. This is so that we can ascertain the relationships between the nodes in the tree by visiting each one once.

b)In order to determine how many tree traversals are necessary, let's examine the given code:

- To visit every node in the tree, the code uses a queue to do a level-order traversal. It goes level by level through the tree, determining whether the nodes provided are cousins.
- It adds all of the child nodes—including null nodes, which indicate the end of the level—to the queue for each level. It then determines whether the nodes provided are siblings at every level. When one of the specified nodes is located, the sibling flag is set to true. It returns the sibling flag negation, indicating whether the nodes are cousins, if the second provided node is discovered at the same level.
- As a result, the number of levels in the tree equals the number of tree traversals needed. In the worst situation, the number of traversals would be equal to the height of the tree, which could be O(n) in the worst case, if the tree was totally unbalanced (like a linked list).
- Given that n is the number of nodes in the tree, the number of tree traversals needed could potentially be as high as O(n).

OUTPUT:



Q4)

This C++ code implements an AVL tree, a self-balancing binary search tree. The `AVLNode` struct represents nodes in the AVL tree, with each node storing data, height, and pointers to left and right children.

The `AVLTree` class provides methods to perform various operations on the AVL tree, such as insertion, removal, finding the maximum or minimum value, finding the kth smallest value, and finding the median value.

The tree is balanced using rotation operations (`rotateLeft` and `rotateRight`) and updated heights to maintain AVL properties. The `insertNode` function inserts a new node into the AVL tree while maintaining balance, and `removeNode` removes a node while also balancing the tree. The `getKthSmallestTask` and `getMedianTask` functions utilize inorder traversal to find the kth smallest value and median value, respectively.

In the `main` function, a sample AVL tree is created, and various queries are performed on it. These queries include finding the maximum or minimum value, finding the kth smallest value, and finding the median value. Each query is followed by an output displaying the result of the operation.

OUTPUT:

```
Input Query: 1
Output: 56
Input Query: 2
Enter k: 3
Output: 34
Input Query: 3
Output: 12
adityaprasad@Adityas-MacBook-Pro-2 2022036_AdityaPrasad %
```

```
Input Query: 1
Output: 10
Input Query: 2
Enter k: 3
Output: 44
Input Query: 3
Output: 34
adityaprasad@Adityas-MacBook-Pro-2 2022036_AdityaPrasad %
```