Pouria Moradpour
4024023040

Question 1

I start from the root, following a similar process to $BST - Search$ algorithm except it is simultaneously done for two nodes. This process continues until the two nodes take different paths, then I return their last mutual node which is actually the first one when starting from the leaves (LCA: lowest common ancestor), as desired.

$BST - LowestCommonAncestor(root, node_1, node_2)$

```
    while current:
        # If both nodes are smaller than the current node, go left
        if node1.val < current.val and node2.val < current.val:
            current = current.left

        # If both nodes are greater than the current node, go right
        elif node1.val > current.val and node2.val > current.val:
            current = current.right

        # The paths split here, so this is the first mutual node
        else:
            return current
```

Question 2

Please note that the `enumerated` array is supposed to be defined outside the function

$TreeEnumerate(x, a, b)$

```
# enumerated is an Empty array/list defined outside the function (to keep it
from getting emptied with each recursion)
if not x == null AND x < b AND x > a do:
        enumerated.add(x)
        TreeEnumerate(x.left, a, b)
        TreeEnumerate(x.right, a, b)

elif not x == null AND x =< b AND x =< a do:
        TreeEnumerated(x.left, a, b)
```

```
elif not x == null AND x => b AND x >= a do:
        TreeEnumerated(x.right, a, b)
# the desired array of numbers will be accessible from enumerated after the
function is done.
```

since the assignment document does not have a clear instruction about this, it is worth noting that I did not consider the numbers a and b themselves as the numbers *between* them.

Questions 3 and 4 Utilities

Since according to the assignment document there is no guarantee that the expression (or the BET itself in the case of question 3) are valid, I have defined validation algorithms here so I can use them in the solutions.

Validation For Prefix Notation (Question 3)

```
function isValidPrefix(expression):
    stack = empty stack

    # Traverse the expression from right to left
    for each token in expression reversed:
        if token is an operand:
            push token onto the stack
        else if token is an operator:
            if stack.size() < 2:
                return false
            pop two operands from the stack
            push the result back onto the stack

    # At the end, there should be exactly one value in the stack
    return stack.size() == 1
```

Validation For Postfix Notation (Question 4)

```
function isValidPostfix(expression):
    stack = empty stack

    # Traverse the expression from left to right
    for each token in expression:
        if token is an operand:
            push token onto the stack
        else if token is an operator:
            # Pop two operands for a binary operator
```

```
            if stack.size() < 2:
                return false
            pop two operands from the stack
            push the result back onto the stack (to simulate reduced
expression)

    # At the end, there should be exactly one value in the stack
    return stack.size() == 1
```

Question 3

Firstly, I indicate the structure of the BET as desired in the assignment document.

Defining the Structure of BET

as discussed in the lectures, we can define the expressions as a context-free grammar:

Prefix

1. $E \rightarrow Operand$
2. $E \rightarrow \alpha EE$
3. $\alpha \rightarrow / \mid * \mid + \mid - \mid \ldots$

Infix

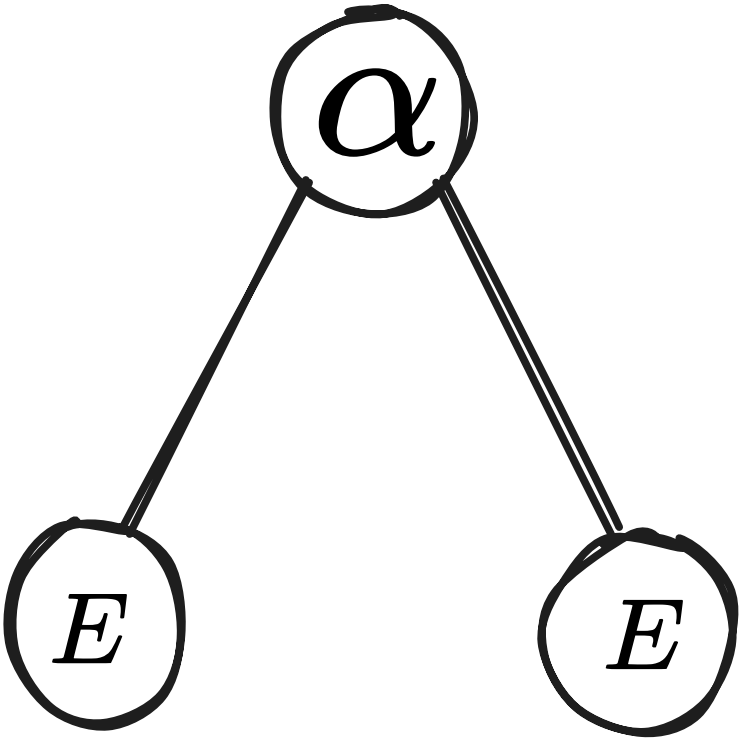
1. $E \rightarrow Operand$
2. $E \rightarrow (E\alpha E)$
3. $\alpha \rightarrow / \mid * \mid + \mid - \mid \ldots$

Postfix

1. $E \rightarrow Operand$
2. $E \rightarrow EE\alpha$
3. $\alpha \rightarrow / \mid * \mid + \mid - \mid \ldots$

So, in a BET we can express any operand or operator as a node such that for operators the children are the corresponding operands.

The following figure corresponds to the defined grammars:



where each E corresponds to an operand which itself can contain operands and operators in the form of above.
Clearly the Prefix, Infix, and Postfix Expressions can be derived from the BET by simply performing Pre-Order, In-Order, and Post-Order traversals respectively.

Deriving an Expression from the BET

we simply need to perform a pre-order traversal and then validate it (using the `isValidPrefix` function defined above in Questions 3 and 4 Utilities), essentially validating the BET itself. IF it was valid, we return the expression

here, I defined a Pre-Order traversal function and the used it along with `isValidPrefix` defined above, to return a prefix expression if the BET is correct.

```
expression = ''  # empty string
function preOrderTraversal(node):
    if node is null:
        return
    expression.add(node.value)
    preOrderTraversal(node.left)
    preOrderTraversal(node.right)

preOrderTraversal(BET)
if isValidPrefix(expression) do:
        return expression
else:
```

```
        return 'the provided tree is not a valid BET'
```

Question 4

initially, I evaluate the postfix expression using the utility function I defined above (in [Questions 3 and 4 Utilities]) in order to create a binary expression tree from the postfix notation I came up with an algorithm that essentially follows a similar process to the evaluation of a postfix expression and uses a stack to gradually create the BET from it's sub-trees.

```
function createBETFromPostfix(postfix):
    stack = empty stack

    for each symbol in postfix:
        if symbol is an operand:
            // Create a tree node for the operand and push it onto the stack
            node = new TreeNode(symbol)
            stack.push(node)
        else if symbol is an operator:
            rightNode = stack.pop()
            leftNode = stack.pop()

            //  the operator
            node = new TreeNode(symbol)

            node.left = leftNode
            node.right = rightNode

            // Push the new subtree back onto the stack
            stack.push(node)

    // The final node in the stack is the root of the BET
    return stack.pop()

// driver
if isValidPostfix(expression):
        return createBETfromPostfix(expression)
else:
        return 'the provided expression is not a correct postfix expression'
```

this returns a BET following the same structure that I stated above in [Defining the Structure of BET]

Question 5

Same as question 4, I follow a similar process to evaluation of the expression except instead of evaluating and pushing to a stack I create subtrees and push them. Also, I were unable to come up with a method that does this with a single stack, this approach uses two: an operator stack and an operand stack and then gradually create the graph to reach the end of the expression whose operator indicates the root of the BET.

```
function createBETFromInfix(infix):
    operatorStack = empty stack
    operandStack = empty stack

    for each symbol in infix:
        if symbol is an operand:
            node = new TreeNode(symbol)
            operandStack.push(node)

        else if symbol is an operator:
            operatorStack.push(symbol)

        else if symbol is ')':
            // Process the last operation indicated by the stacks
            while operatorStack is not empty:
                operator = operatorStack.pop()
                rightNode = operandStack.pop()
                leftNode = operandStack.pop()

                // Create a tree node for the operator
                node = new TreeNode(operator)
                node.left = leftNode
                node.right = rightNode

                operandStack.push(node)

                // Stop processing when the current subexpression is
complete
                if the next symbol in the input is '(':
                    break

    // The last node in the operand stack is the root of the Binary
Expression Tree
```

```
    return operandStack.pop()
```

this returns a BET following the same structure that I stated above in [Defining the Structure of BET]

Question 6

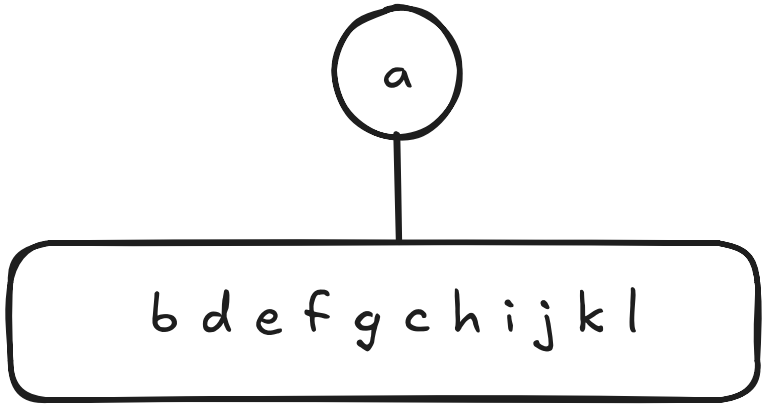
I initially had an idea of creating a script that created every possible Binary tree whose preorder was the given array and then compare it's post-order traversal to the given one. Even though this might have worked for the given problem, the algorithm would have been extremely inefficient, essentially turning our problem into a practically unsolvable one.

I manually written the trees and tracked the subtrees which can have different distinct trees while retaining the given traversals, here is the process:

$Preorder$: $abdefgchijkl$
$Postorder$: $edgfbihkljca$

Step 1:

Considering the NLR and LRN form of pre-order and post-order traversals and the fact that pre-order starts with a and pos-torder ends with it we can conclude that node **a** is the root of the tree.

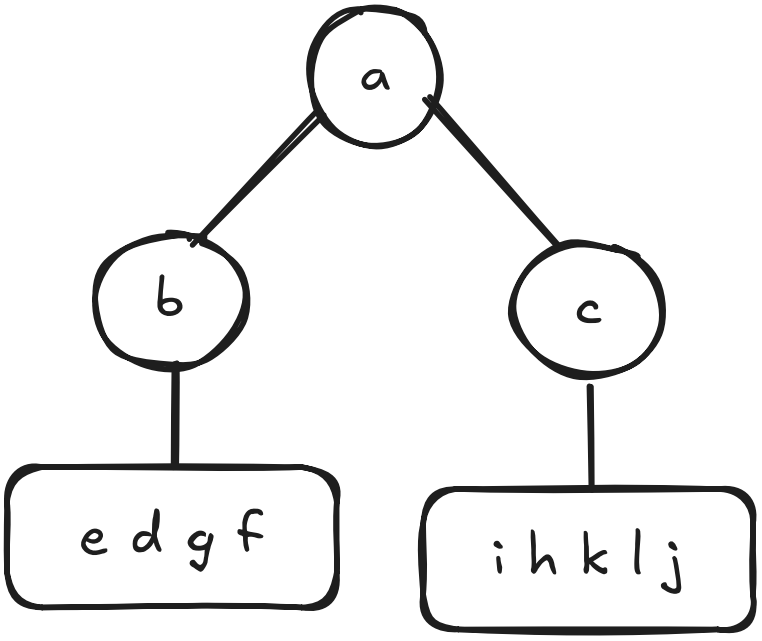


Step 2:

We are left with:
$Preorder$: $bdefgchijkl$
$Postorder$: $edgfbihkljc$

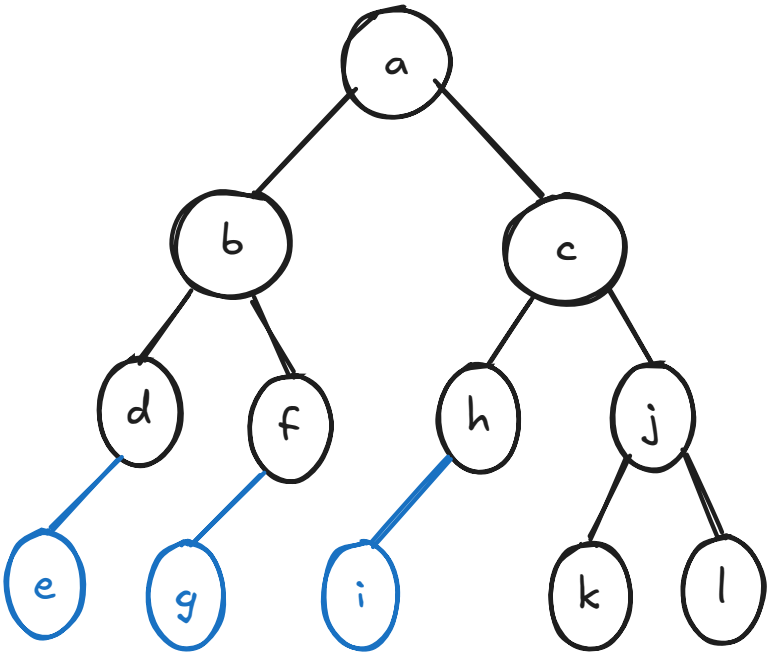
Since **b** is the second node in the pre-order and all of the nodes appearing before it in post-order appear directly after it in pre-order we can conclude that it is a direct child of a. performing the same process on the node **c**, (since it is the first node in pre-order that did not appear before b in post-order) indicates that c is a direct child of a.

It is also clear that b is the right child since if it were the left one we would have seen a directly after b in the post-order traversal.
we are left with the following distinct tree so far:



Step 3:

now, we just need to determine how many distinct trees can be made by just investigating the remaining ambiguous nodes:
by following a similar process to step 2 and comparing the two traversals we are left with:



where each of the blue nodes can also be the left child of their parent. So, there are 3 nodes that can vary in 2 ways in the tree itself while still corresponding to the given traversals which indicates that **there are $2^3 = 8$ binary trees corresponding to the given traversals.**