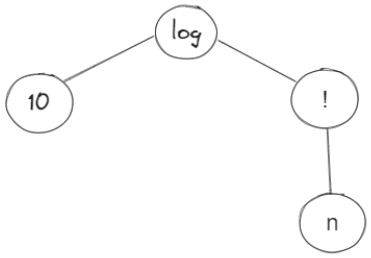
G4

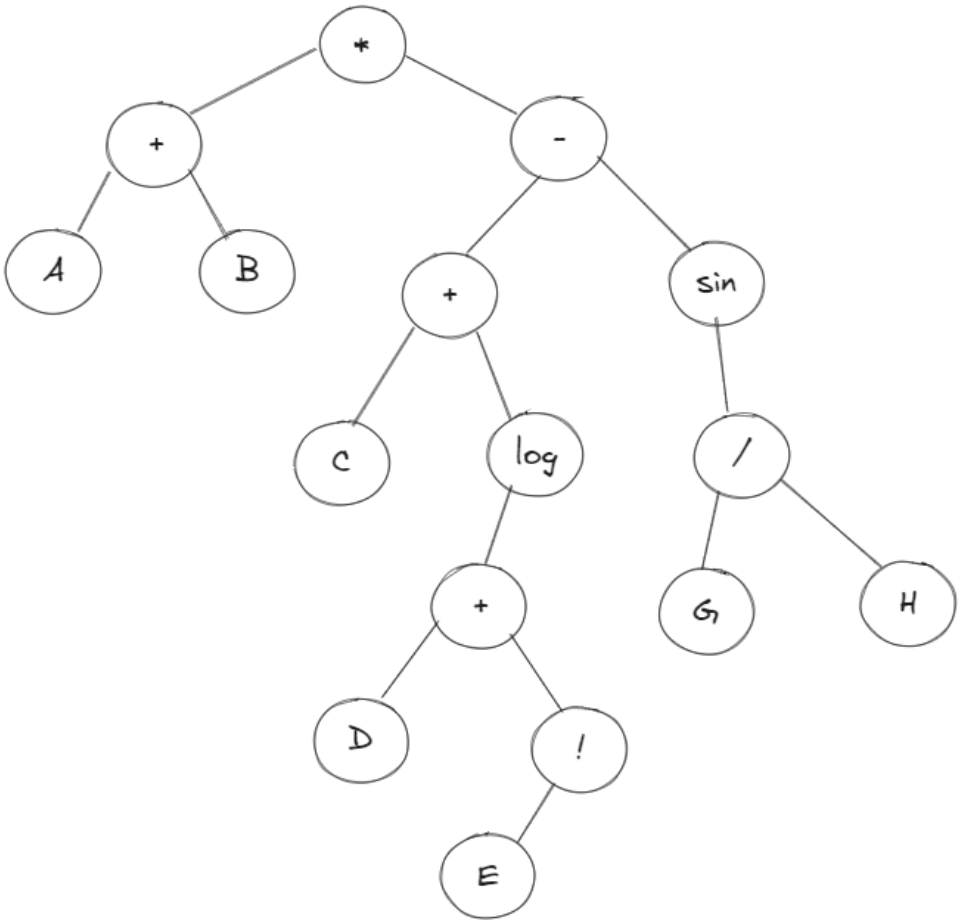
LAB 12

a) Draw expression tree for the following infix expressions and then give their postfix and prefix expressions.

log n!

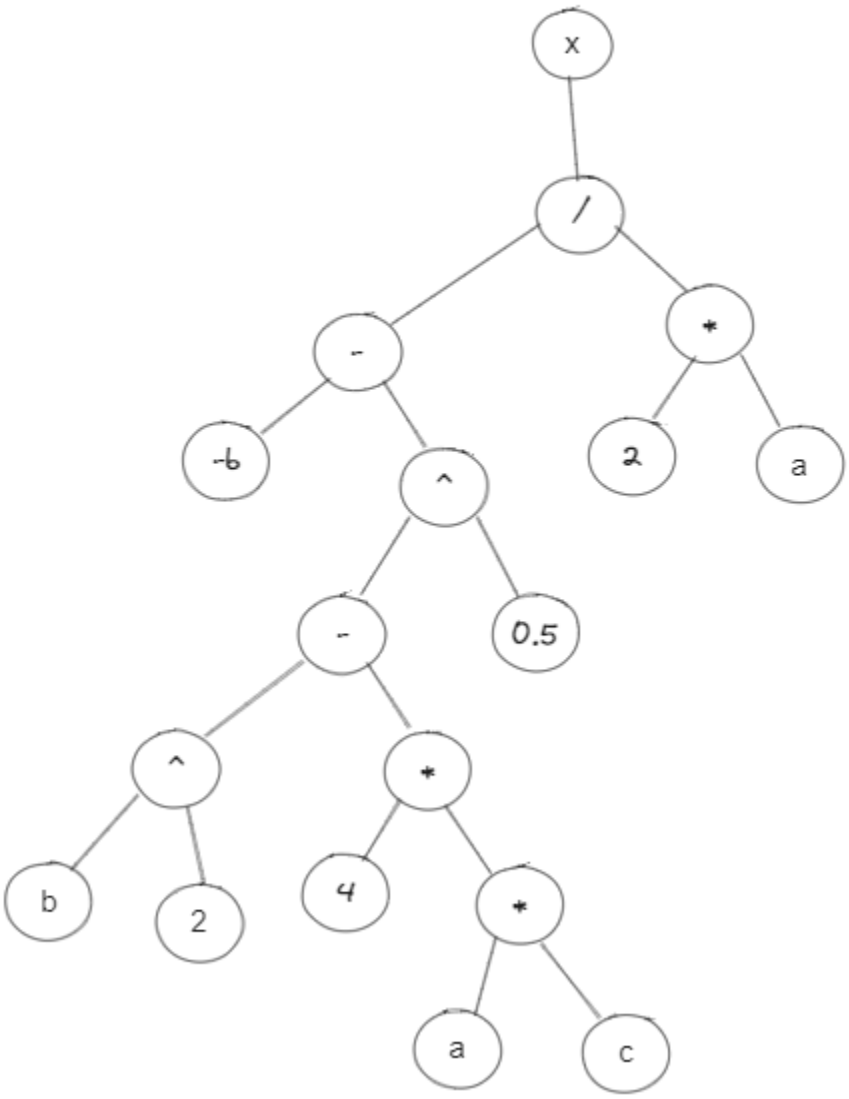


(A+B) * (C + log (D+E!) – sin (G/H))



G4

x=(-b ± √ (b2
- 4ac)) / 2a



G4

b) Write a function TreeSize that will count all the nodes of a linked binary tree.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def TreeSize(root):
    if root is None:
        return 0

    return 1 + TreeSize(root.left) + TreeSize(root.right)

# Driver code
# Create the binary tree
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

# Calculate the size of the tree
size = TreeSize(root)
print("Size of the tree:", size)
```

Output:

```
PS D:\NED University\E
Size of the tree: 5
PS D:\NED University\E
```

G4

c) Write a function ClearTree that will traverse a binary tree (in whatever order you find works best) and assign zero at all the nodes.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def ClearTree(root):
    if root is None:
        return

    # Clear the left subtree
    ClearTree(root.left)
    # Clear the right subtree
    ClearTree(root.right)
    # Assign zero to the current node
    root.data = 0

def display_tree(root):
    if root is None:
        return

    print(root.data, end=" ")
    display_tree(root.left)
    display_tree(root.right)
# Driver code
# Create the binary tree
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

# Display the original tree
print("Original Tree:")
display_tree(root)

# Clear the tree
ClearTree(root)

# Display the cleared tree
print("\nCleared Tree:")
display_tree(root)
```

Output:

```
Original Tree:
1 2 4 5 3
Cleared Tree:
0 0 0 0 0
```

G4

d) Implement an algorithm to perform a double-order traversal of a binary tree, meaning that each node of the tree, the algorithm first visits the node, then traverses the left subtree (in double order), then visits the node again, then traverses its right subtree (in double order).

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

def DoubleOrderTraversal(root):
    if root is None:
        return

    # Visit the node
    print(root.data, end=" ")

    # Traverse the left subtree in double order
    DoubleOrderTraversal(root.left)

    # Visit the node again
    print(root.data, end=" ")

    # Traverse the right subtree in double order
    DoubleOrderTraversal(root.right)

# Driver code
# Create the binary tree
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

# Perform double-order traversal
print("Double-Order Traversal:")
DoubleOrderTraversal(root)
```

Output:

```
Double-Order Traversal:
1 2 4 4 2 5 5 1 3 3
```