Binary Tree

Nomor 1

```
class BinaryTree:
    def __init__(self, root):
        self.key = root
        self.leftChild = None
        self.rightChild = None

    def insertLeft(self, newNode):
        if self.leftChild is None:
            self.leftChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t

    def insertRight(self, newNode):
        if self.rightChild is None:
            self.rightChild = BinaryTree(newNode)
        else:
            t = BinaryTree(newNode)
            t.rightChild = self.rightChild
            self.rightChild = t

    def getRightChild(self):
        return self.rightChild

    def getLeftChild(self):
        return self.leftChild

    def setRootVal(self, obj):
        self.key = obj

    def getRootVal(self):
        return self.key


tree = BinaryTree('a')
tree.insertLeft('b')
tree.insertRight('c')
tree.getLeftChild().insertRight('d')
tree.getRightChild().insertLeft('e')
tree.getRightChild().insertRight('f')

def print_tree_preorder(node, path="Root"):
    if node is not None:
        print(f"{node.getRootVal()} --> Path: {path}")
        if node.getLeftChild() is not None:
            print_tree_preorder(node.getLeftChild(), f"{path} -> Left")
        if node.getRightChild() is not None:
            print_tree_preorder(node.getRightChild(), f"{path} -> Right")
print_tree_preorder(tree)
```

```
 a --> Path: Root
 b --> Path: Root -> Left
 d --> Path: Root -> Left -> Right
 c --> Path: Root -> Right
 e --> Path: Root -> Right -> Left
 f --> Path: Root -> Right -> Right
```

Nomor 2

```
class BinaryTree:
    def __init__(self, root):
        self.key = root
        self.leftChild = None
        self.rightChild = None
```

```
    def insertLeft(self, new_node):
        if self.leftChild == None:
            self.leftChild = BinaryTree(new_node)
        else:
            t = BinaryTree(new_node)
            t.leftChild = self.leftChild
            self.leftChild = t

    def insertRight(self, new_node):
        if self.rightChild == None:
            self.rightChild = BinaryTree(new_node)
        else:
            t = BinaryTree(new_node)
            t.rightChild = self.rightChild
            self.rightChild = t
    def getLeftChild(self):
        return self.leftChild

    def getRightChild(self):
        return self.rightChild


    def getRootVal(self):
        return self.key

    def setRootVal(self, obj):
        self.key = obj



def buildTree(expression):
    tokens = expression.split()
    stack = []
    for token in tokens:
        if token.isdigit():
            node = BinaryTree(int(token))
            stack.append(node)
        else:
            rightOperand = stack.pop()
            leftOperand = stack.pop()
            node = BinaryTree(token)
            node.insertLeft(leftOperand)
            node.insertRight(rightOperand)
            stack.append(node)
    return stack.pop()

def evaluate(node):
    if node is None:
        return 0

    if node.getLeftChild() is None and node.getRightChild() is None:
        return node.getRootVal()

    leftValue = evaluate(node.getLeftChild().getRootVal())
    rightValue = evaluate(node.getRightChild().getRootVal())

    operator = node.getRootVal()

    if operator == '+':
        return leftValue + rightValue
    elif operator == '-':
        return leftValue - rightValue
    elif operator == '*':
        return leftValue * rightValue
    elif operator == '/':
        return leftValue / rightValue
    else:
        print(f'{operator}  operator tersebut tidak dikenali')


postfix_expr1 = "2 8 9 + *"
postfix_expr2 = "2 4 + 3 5 * -"
postfix_expr3 = "10 3 2 12 + - *"
```

```
resTree1 = buildTree(postfix_expr1)
print(evaluate(resTree1))
resTree2 = buildTree(postfix_expr2)
print(evaluate(resTree2))
resTree3 = buildTree(postfix_expr3)
print(evaluate(resTree3))
```

```
34
-9
-110
```