In [18]:

```
##[2진트리] 다항 연산식을 2진 트리로 구성 후
## Stack 구조를 이용한 2항연산으로 연산
operators = {'+': 1, '-': 1, '*': 2, '/': 2}  # 연산자 우선순위 고려
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def find_priority_operator(expression):
    global operators
    # 우선순위가 가장 빠른 연산자의 index를 찾음
    operator_index = -1  # 선택된 연산자 index
    max_precedence = -1  # 가장 빠른 연산 우선순위 index
    parentheses_count = 0  # ()의 깊이

    for i in range(len(expression) - 1, -1, -1):
        char = expression[i]
        if char == ')':
            parentheses_count += 1
        elif char == '(':
            parentheses_count -= 1
        elif char in operators and parentheses_count == 0:  # operater이
            if operators[char] >= max_precedence:
                operator_index = i
                max_precedence = operators[char]
                return i
    return -1

def construct_binary_tree(expression):
    global operators
    if expression[0] == '(' and expression[-1] == ')':
        expression = expression[1:-1]
    print(expression)
    if len(expression) == 0:
        return None

    # 우선순위가 가장 빠른 연산자의 index를 찾음
```

```
    operator_index = find_priority_operator(expression)
    if operator_index == -1:  # 연산자가 없으면 피연산자로 간주
        node = Node(expression)
    else:  # 연산자를 기준으로 왼쪽과 오른쪽의 표현식을 나눕니다.
        operator = expression[operator_index]
        left_expression = expression[:operator_index]
        right_expression = expression[operator_index + 1:]

        # 노드를 생성하고 재귀적으로 왼쪽과 오른쪽 서브트리를 구성
        node = Node(operator)
        node.left = construct_binary_tree(left_expression)
        node.right = construct_binary_tree(right_expression)
    return node

def in_order_traverse(node):
    if node is not None:
        in_order_traverse(node.left)  # 왼쪽 서브트리 중위 순회
        print(node.value, end=' ')  # 현재 노드 방문
        in_order_traverse(node.right)  # 오른쪽 서브트리 중위 순회

def pre_order_traverse(node):
    if node is not None:
        print(node.value, end=' ')  # 현재 노드 방문
        pre_order_traverse(node.left)  # 왼쪽 서브트리 전위 순회
        pre_order_traverse(node.right)  # 오른쪽 서브트리 전위 순회

def post_order_traverse(node):
    if node is not None:
        post_order_traverse(node.left)  # 왼쪽 서브트리 후위 순회
        post_order_traverse(node.right)  # 오른쪽 서브트리 후위 순회
        print(node.value, end=' ')  # 현재 노드 방문

def evaluate_binary_tree(root):
    stack = []
    if root is not None:
        stack.extend(evaluate_binary_tree(root.left))
        stack.extend(evaluate_binary_tree(root.right))
        if root.value.isdigit():
            stack.append(int(root.value))
            print("<<Push:", root.value)
```

```
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            print(">>Pop :", operand1)
            print(">>Pop :", operand2)
            result = perform_operation(root.value, operand1, operand2)
            print("[Operation])", root.value)
            stack.append(result)
            print("<<Push(result):", result)
    return stack

def perform_operation(operator, operand1, operand2):
    if operator == '+':
        return operand1 + operand2
    elif operator == '-':
        return operand1 - operand2
    elif operator == '*':
        return operand1 * operand2
    elif operator == '/':
        return operand1 / operand2

#expression = "3*4-(2*3+4/2)/2"
expression = input("연산식 입력(숫자와 괄호, 4칙연산 기호만 사용):")
expression = expression.replace(' ', '')
root = construct_binary_tree(expression)
print("\n[Post-Order traverse]", end=' ')
post_order_traverse(root)  # left > right > root 순회/
print('\n')

result_stack = evaluate_binary_tree(root)
result = result_stack[0]
print("\n[Last result]", result)
```

```
3*4-(2*3+4/2)/2
3*4-(2*3+4/2)
3*4
3
4
2*3+4/2
2*3+4
2*3
2
3
4
2
2

[Post-Order traverse] 3 4 * 2 3 * 4 + 2 / - 2 /

<<Push: 3
<<Push: 4
>>Pop : 3
>>Pop : 4
[Operation]) *
<<Push(result): 12
<<Push: 2
<<Push: 3
>>Pop : 2
>>Pop : 3
[Operation]) *
<<Push(result): 6
<<Push: 4
>>Pop : 6
>>Pop : 4
[Operation]) +
<<Push(result): 10
<<Push: 2
>>Pop : 10
>>Pop : 2
[Operation]) /
<<Push(result): 5.0
>>Pop : 12
>>Pop : 5.0
[Operation]) -
<<Push(result): 7.0
<<Push: 2
>>Pop : 7.0
>>Pop : 2
[Operation]) /
<<Push(result): 3.5

[Last result] 3.5
```

In [ ]: