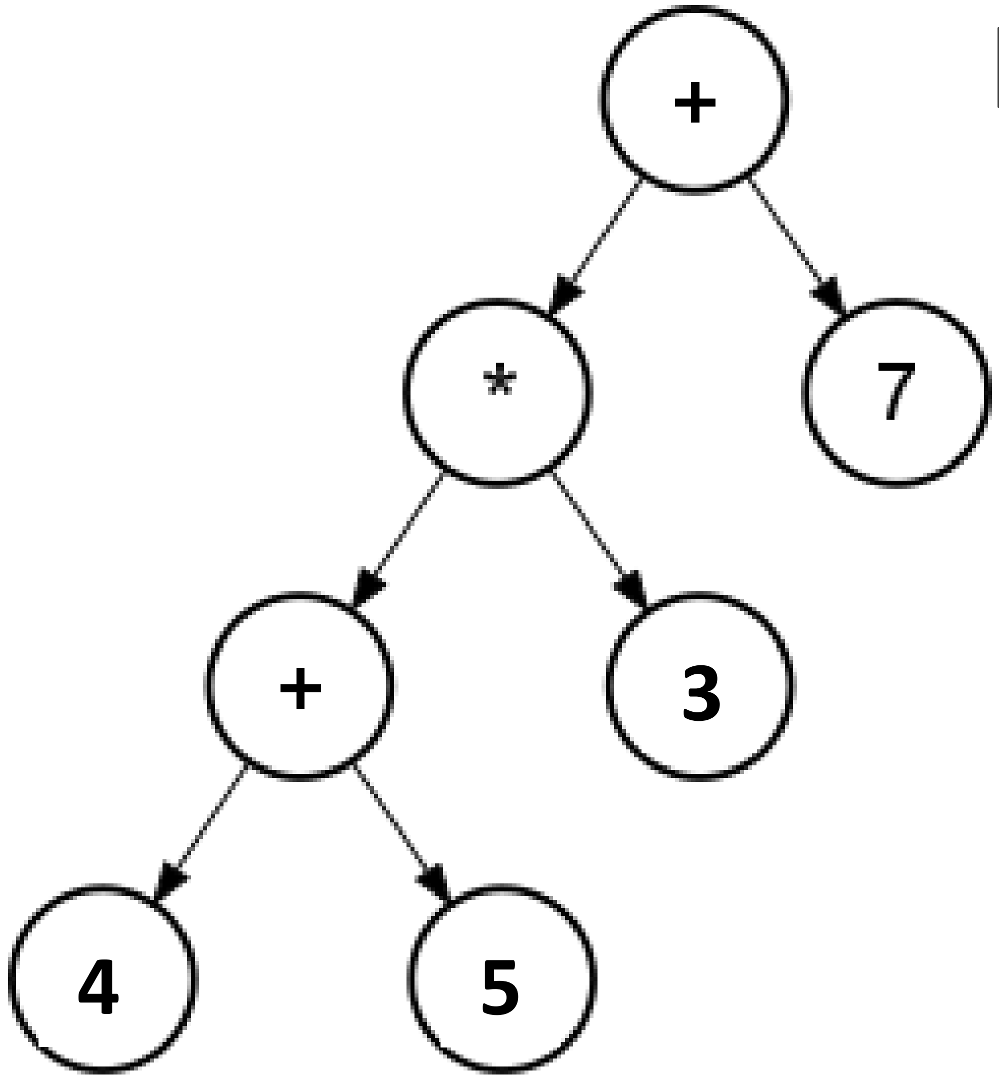
Programming Assignment 4

1. Main Module (calculator.py):
    1. Has the main program
2. ADT Tree (tree.py):
    1. Has two classes BinaryTree and ExpTree
3. ADT Stack (stack.py):
    1. Has a class Stack

Main Module

The main should have two functions:

1. calculate
2. infix_to_postfix

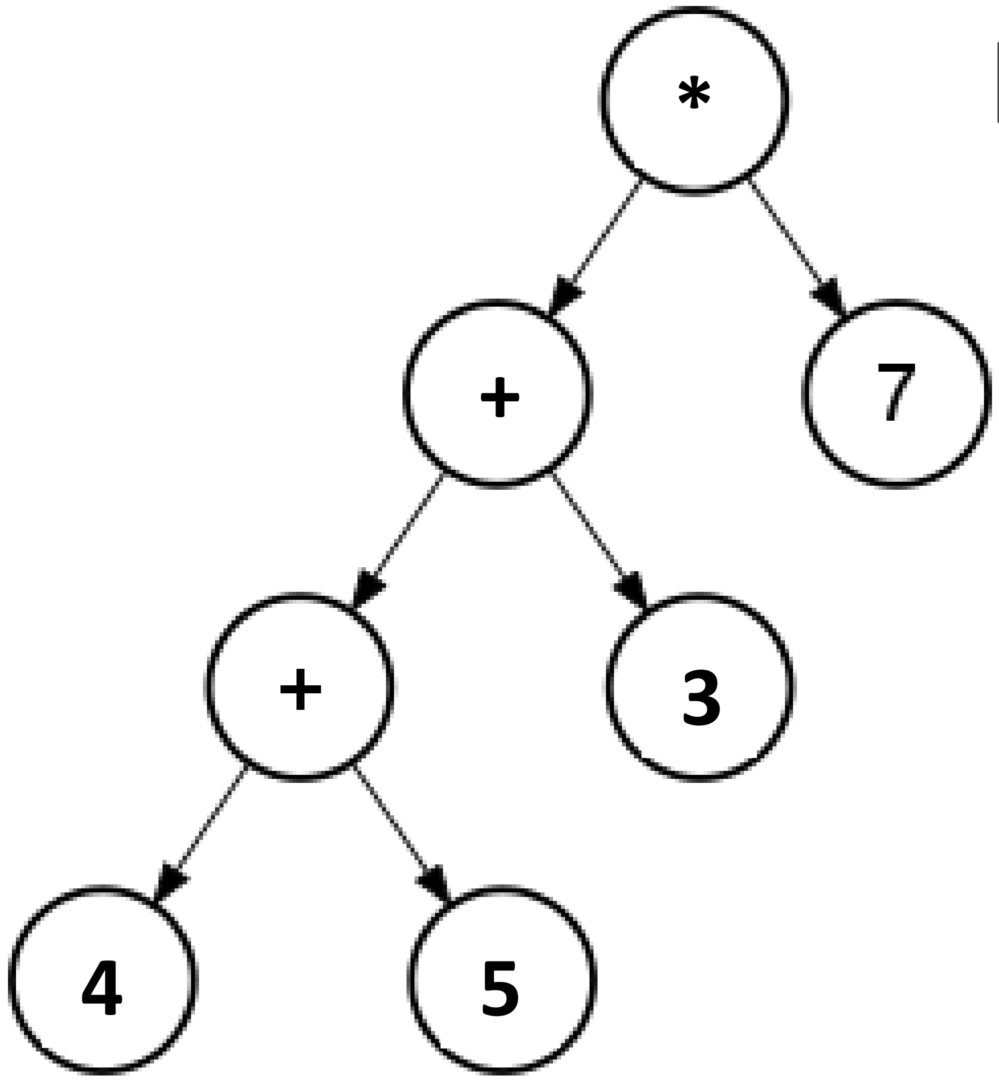


Infix to Postfix Conversion

EXAMPLE 1: infix (4 + 5) *3 + 7 -> postfix 4 5 + 3 * +

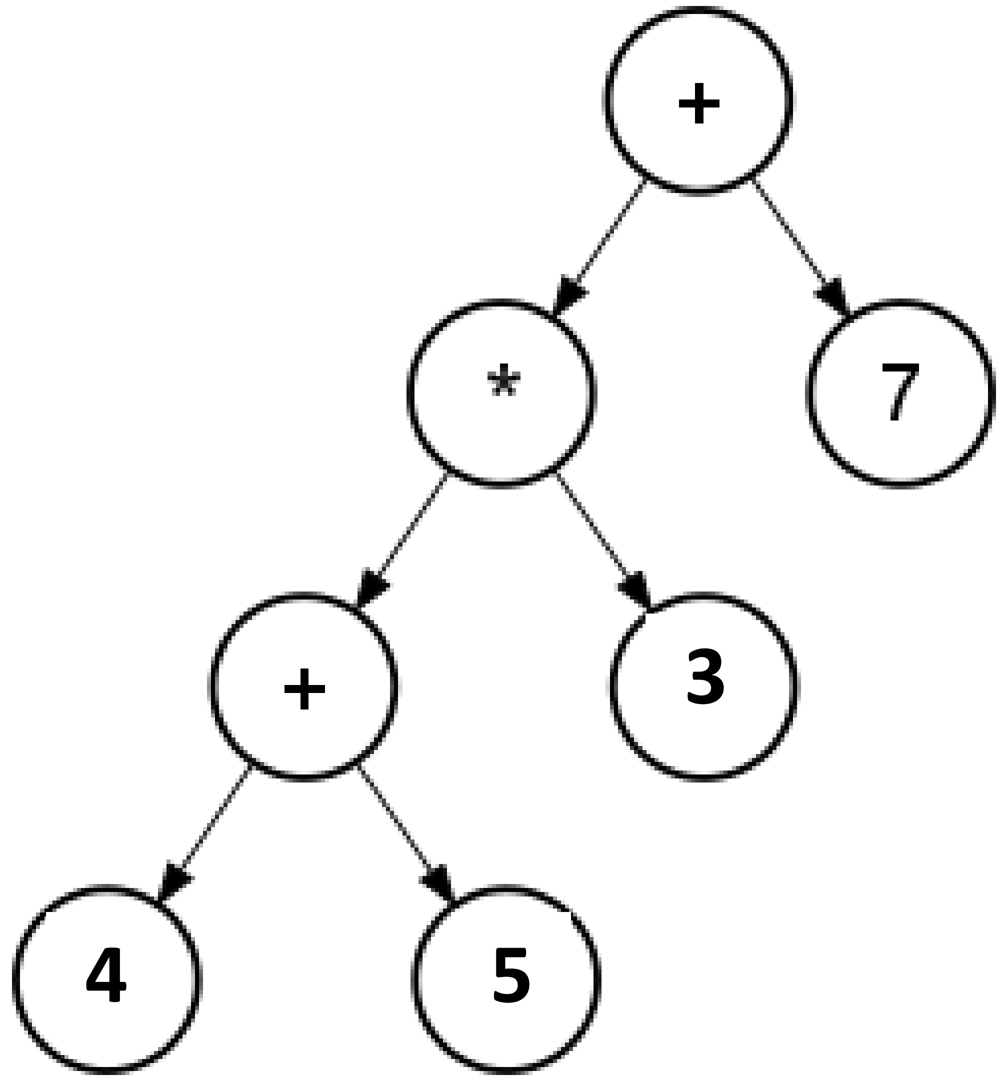
Step	Postfix	Num	Op Stack
1 (			(
2 4		4	(
3 +	4		(+
4 5	4	5	(+
5)	4 5 +		
6 *	4 5 +		*
7 3	4 5 +	3	*
8 +	4 5 + 3 *		+
9 7	4 5 + 3 *	7	+
10	4 5 + 3 * 7 +		

Infix to Postfix Conversion



EXAMPLE 2: infix (4 + 5) + 3 * 7 -> postfix 4 5 + 3 7 * +

Step	Postfix	Num	Op Stack
1 (			(
2 4		4	(
3 +	4		(+
4 5	4	5	(+
5)	4 5 +		
6 +	4 5 +		+
7 3	4 5 +	3	+
8 *	4 5 + 3		+ *
9 7	4 5 + 3	7	+ *
10	4 5 + 3 7 * +		



Infix to Postfix Conversion

Infix	Postfix
(4 + 5) *3 + 7	4 5 + 3 * 7 +
(4 + 5) +3*7	4 5 + 3 7 * +
(42+5)*3+7	42 5 + 3 * 7 +

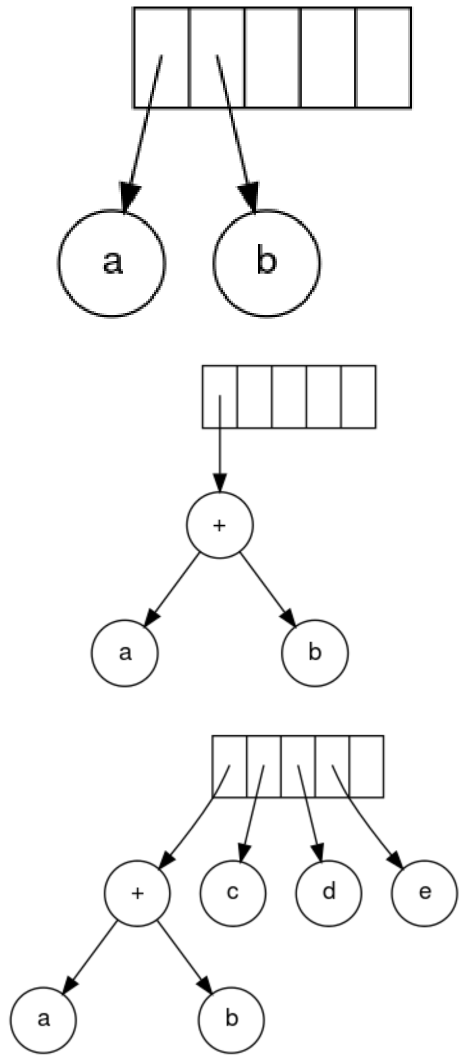
Infix to Postfix Algorithm

1. Make a stack that holds operators (called op. stack)
2. Make a postfix and num strings that hold the postfix expression and the current number
3. Traverse the infix expression using a loop
    1. if char is a digit or a dot add it to num
    2. if char is not a digit or a dot, add num to the postfix and assign it to an empty string
    3. If char is ‘(‘ , append it to the op. stack
    4. If char is ‘)’, remove operators from the op. stack until ‘(‘ and add them to the postfix, the last added ‘(‘ should be removed from the stack
    5. If char is an operator :
        1. add it to the stack if the operator has the higher precedence than the previous operator in the stack has
        2. otherwise, remove the previous operator from the stack and add it to the postfix, then add a new operator to the stack
4. When you add tokens to the postfix, add a space to the postfix too to separate them from each other.
5. When the loop is over, add the num and all remaining operators in the op. stack to the postfix

Tree ADT

The module tree.py should have two classes:

1. BinaryTree (you should already implement it in lab 6)
2. ExpTree (a subclass of BinaryTree)
    1. make_tree(postfix)
    2. __str__(self)
    3. preorder()
    4. postorder()
    5. inorder()
    6. evaluate()



Constructing an Expression Tree

1. If a symbol is an operand, make a tree node and add it to a stack
   1. For example:

      ```
      stack.push(ExptTree(symbol))
      ```

2. If a symbol is an operator, make a tree node and add two trees as its left and right children, add it to a stack
   1. For example:

      ```
      temp = ExpTree(symbol)
      temp.rightChild = stack.pop()
      temp.leftChild = stack.pop()
      stack.push(temp)
      ```

Constructing an Expression Tree

Postfix: 4 5 + 3 * 7 +

Step	Symbol	Stack
1	4	Node 4
2	5	Node 4, Node 5
3	+	Node + (Node 4, Node 5)
4	3	Node + (Node 4, Node 5), Node 3
5	*	Node * (Node +(Node 4, Node 5), Node 3)
6	7	Node * (Node +(Node 4, Node 5), Node 3), Node 7
7	+	Node + (Node * (Node +(Node 4, Node 5), Node 3), Node 7)

Preorder, Inorder, Postorder Traversals

```
s = ''
    if tree != None:
      s = tree.getRootVal()
      s += ExpTree.preorder(tree.getLeftChild())
      s += ExpTree.preorder(tree.getRightChild())
    return s
```

Evaluate an Expression Tree Algorithm

1. If tree is None do nothing
2. Else
    1. If a node is not an operator, return its value
    2. Else:
        1. Evaluate the left subtree (left_value)
        2. Evaluate the right subtree (right_value)
        3. Evaluate the tree depending on the root value:
            1. If the root value is ‘+’ add the left and right subtree values
            2. if the root value is ‘-’ subtract the left and right subtree values
            3. etc.

Stack ADT

1. The module stack.py should contain one class Stack
    1. __init__(self)
    2. isEmpty(self)
    3. push(self, item)
    4. pop(self)
    5. peek(self)
    6. size(self)

Help on PA4

If you need more clarification, post your questions on Ed

Attend office hours and sections