NAME _____ **ICA-14**

NetID _____@email.arizona.edu

Problem 1. [Traversals]

Suppose that you have an empty BST (binary search tree). Insert the following values into the tree, in this order: `12 80 76 62 5 68 85 30`. Draw the resulting tree.

Give the preorder and inorder traversals of the tree.

Problem 2. Consider the definition below of a `BinarySearchTree`:

```
class BinarySearchTree:
    def __init__(self, value):
        self._value = value
        self._lchild = None
        self._rchild = None
```

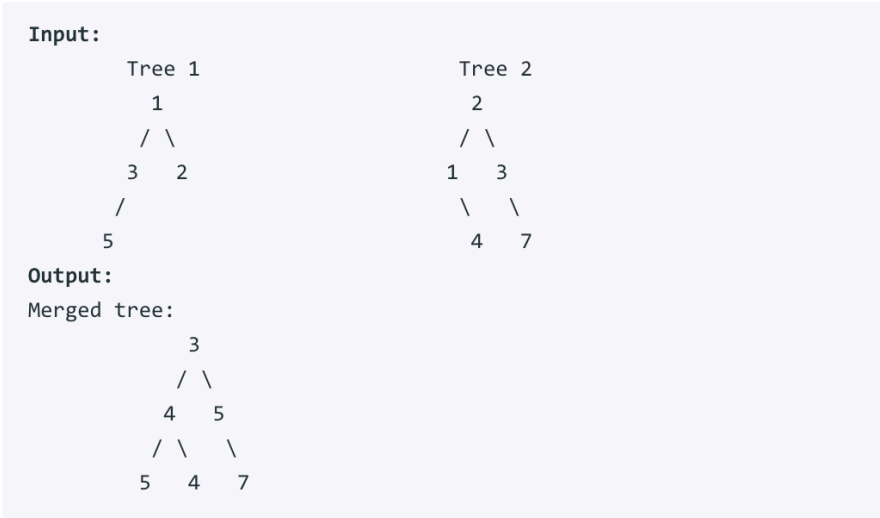
Write a function `print_pre_order(node)` which prints a preorder traversal of the values in the tree, one value per line. **Use a recursive solution!**

For convenience, you may directly access the attributes of the `BinarySearchTree` class when writing your function.

Problem 3. [Merge Two Binary Trees] Given two binary trees and imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not.

You need to merge them into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of new tree.

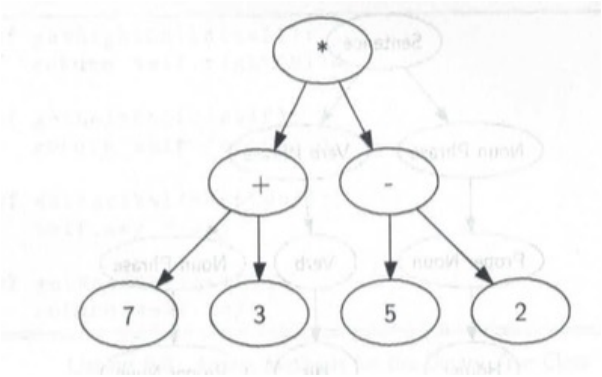
Example 1:



You are giving class TreeNode, and need to implement a function named `mergeTrees(t1, t2)` that takes two TreeNode objects as arguments and returns the merged tree

```
class TreeNode:
    def __init__(self, x):
    self.val = x
    self.left = None
    self.right = None
```

Problem 4. [Binary Tree Applications] We can represent mathematical expressions such as `((7+3)*(5-2))` as a parse tree. The parentheses make the precedence explicit; the order imposed by precedence is represented in the hierarchy of the corresponding parse tree:



Note that:

- leaves contain operands
- operators are in the internal nodes
- the deeper the operator, the higher its precedence (in that expression)

This problem focuses on learning how to build a parse tree from a fully parenthesized mathematical expression. First, we break up the expression string into a list of "tokens". There are four kinds of **tokens**, as listed below:

- left parentheses
- right parentheses
- operand
- operator

A left parenthesis starts a new expression, and requires a new tree, and a right parenthesis finishes that expression. An operand is a leaf node and an operator has a left and right child. (We are not considering unary operators at the moment.)

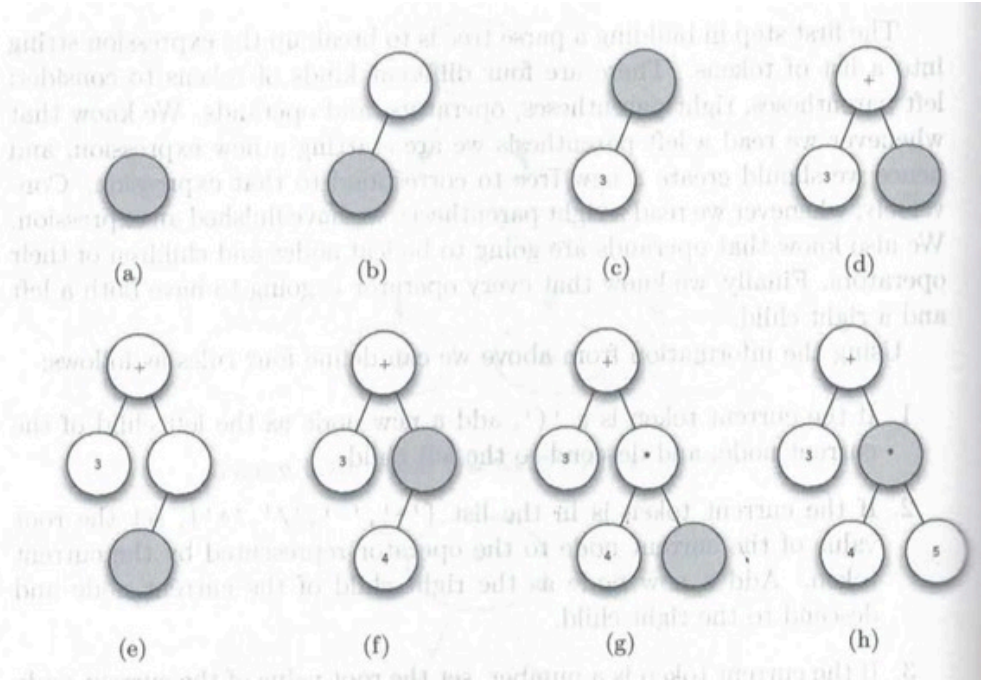
We can use the following four **rules** to construct a tree:

1. If the current token is a '(', add a new node as the left child of the current node and descend to the left child
2. If the current token is in the list ['+', '-', '/', '*'], set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child.
3. If the current token is a number, set the root value of the current node to the number and return to the parent.
4. If the current token is a ')', go to the parent of the current node.

*from *Problem Solving with Algorithms and Data Structures using Python*, Miller and Ranum

We will walk through using these rules to create a tree from the expression (3+ (4*5)). First, assume the expression has been parsed into the list of tokens **['(', '3', '+', '(', '4', '*', '5', ')', ')'].** The figure included below shows the tree after each step, with the current node shaded in grey:

a) Create an empty tree.
b) Read **(** as the first token. By rule 1, create a new node as the left child of the root. Make the current node this new child.
c) Read **3** as the next token. By rule 3, set the root value of the current node to 3 and go back up the tree to the parent.
d) Read **+** as the next token. By rule 2, set the root value of the current node to + and add a new node as the right child. The new right child becomes the current node.
e) Read **a** (as the next token. By rule, 1 create a new node as the left child of the current node. The new left child becomes the current node.
f) Read **4** as the next token. By rule 3, set the value of the current node to 4. Make the parent of 4 the current node.
g) Read ***** as the next token. By rule 2, set the root value of the current node to * and create a new right child. The new right child becomes the current node.
h) Read **5** as the next token. By rule 3, set the root value of the current node to 5. Make the parent of 5 the current node.
i) Read **)** as the next token, By rule 4, make the parent of * the current node.
j) Read **)** as the next token. By rule 4 we make the parent of + the current node. At this point there is a not parent for + so we are done.



We need to keep track of the current node as well as the parent of the current node. We can get to the child by the left and right references, but that doesn't help with the parent. However, we can keep track of the parent as we traverse the tree by using a stack. Each time we descend to the child, we push the parent on the stack. When we want to return to the parent of the current node, we pop the parent off the stack.

Design your problem using Binary Search Tree, Stack, and parsing functions. Fill out the details of the `build_parsetree` function according to the description above.

Usage:

```
>>> build_parsetree("7")
['7']
--------
token = 7
Rule 3
(7 None None)
<binary_tree.BinaryTree object at 0x101197550>
>>> build_parsetree("( 7 )")
['(', '7', ')']
--------
token = (
Rule 1
(@ (@ None None) None)
--------
token = 7
Rule 3
(@ (7 None None) None)
--------
token = )
Rule 4
(@ (7 None None) None)
<binary_tree.BinaryTree object at 0x101197400>
>>> build_parsetree("( 7 + 3 )")
['(', '7', '+', '3', ')']
--------
token = (
Rule 1
(@ (@ None None) None)
--------
token = 7
Rule 3
(@ (7 None None) None)
--------
token = +
Rule 2
(+ (7 None None) (@ None None))
--------
token = 3
Rule 3
(+ (7 None None) (3 None None))
--------
token = )
Rule 4
(+ (7 None None) (3 None None))
<binary_tree.BinaryTree object at 0x1011974e0>
>>>
```