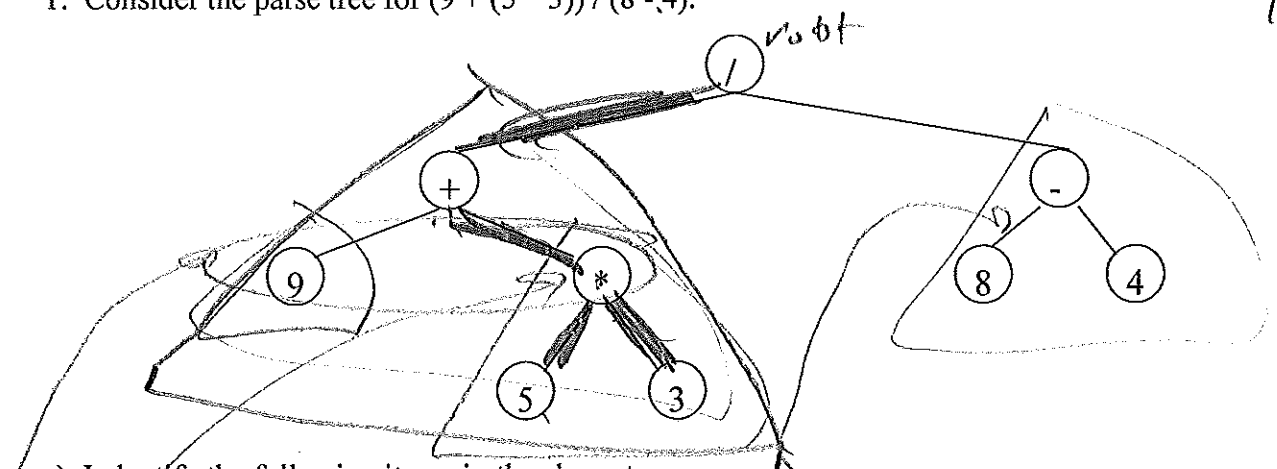
Data Structures (2) (1) (4) (3) Lecture 18 Name:_____

1. Consider the parse tree for (9 + (5 * 3)) / (8 - 4):



a) Indentify the following items in the above tree:

- *node* containing "*"
- *edge* from node containing "-" to node containing "8"
- *root* node
- *children* of the node containing "+" 9 *
- *parent* of the node containing "3" **
- *siblings* of the node containing "*" 9
- *leaf* nodes of the tree 9, 5, . . .
- *subtree* who's root is node contains "+"
- *path* from node containing "+" to node containing "5"
- *branch* from root node to "3" (+, *, 5)

b) Mark the *levels* of the tree (level is the number of edges on the path from the root)

c) What is the *height* (max. level) of the tree? 3

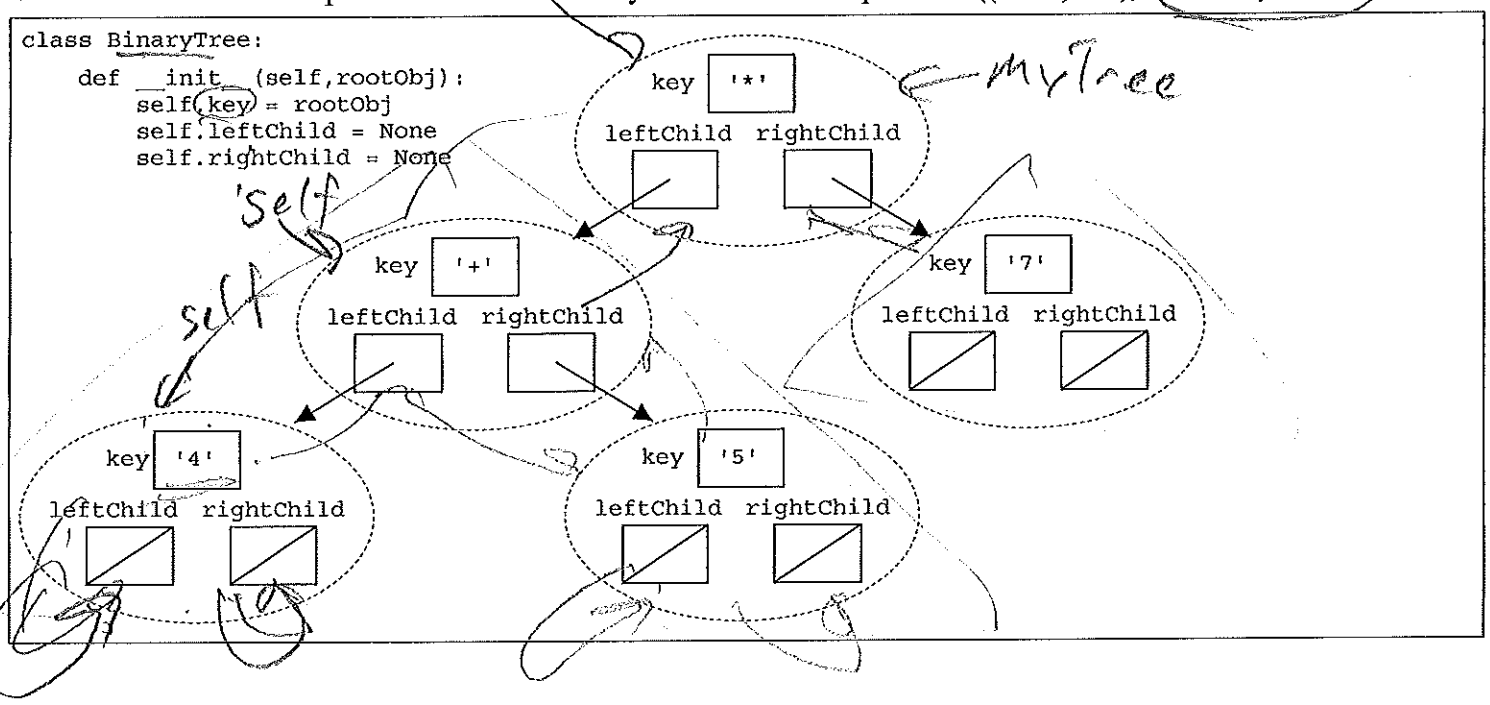
2. In Python an easy way to implement a tree is as a list of lists where a tree look like:

["node value", remaining items are subtrees for the node each implemented as a list of lists]

Complete the list-of-lists representation look like for the above parse tree:

['/', ['+', ['9'], ['*', ['5'] ['3']]], ['-', ['8'] ['4']]]]

3. Consider a "linked" representations of a BinaryTree. For the expression ((4 + 5) * 7), the binary tree would be:



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

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

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

    def isLeaf(self):
        return ((not self.leftChild) and
                (not self.rightChild))

    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

    def inorder(self):
        if self.leftChild:
            self.leftChild.inorder()
        print(self.key)
        if self.rightChild:
            self.rightChild.inorder()

    def postorder(self):
        if self.leftChild:
            self.leftChild.postorder()
        if self.rightChild:
            self.rightChild.postorder()
        print(self.key)
```

a) Fix the `insertLeft` and `insertRight` code:
(Listing 6.6 and 6.7 are wrong in the text on pp. 242-3)

No left and right -- these should be leftChild + rightChild

```
def preorder(self):
    print(self.key)
    if self.leftChild:
        self.leftChild.preorder()
    if self.rightChild:
        self.rightChild.preorder()

def printexp(self):
    if self.leftChild:
        print('(', end=' ')
        self.leftChild.printexp()
    print(self.key, end=' ')
    if self.rightChild:
        self.rightChild.printexp()
        print(')', end=' ')

def postordereval(self):
    opers = {'+':operator.add, '-':operator.sub,
             '*':operator.mul, '/':operator.truediv}
    res1 = None
    res2 = None
    if self.leftChild:
        res1 = self.leftChild.postordereval()
    if self.rightChild:
        res2 = self.rightChild.postordereval()
    if res1 and res2:
        return opers[self.key](res1,res2)
    else:
        return self.key
```

Some corresponding external (non-class) functions:

```
def inorder(tree):
    if tree != None:
        inorder(tree.getLeftChild())
        print(tree.getRootVal())
        inorder(tree.getRightChild())

def printexp(tree):
    if tree.leftChild:
        print('(', end=' ')
        printexp(tree.getLeftChild())
    print(tree.getRootVal(), end=' ')
    if tree.rightChild:
        printexp(tree.getRightChild())
        print(')', end=' ')

def height(tree):
    if tree == None:
        return -1
    else:
        return 1 +
           max(height(tree.leftChild),
               height(tree.rightChild))
```

```
def printexp(tree):
    sVal = ""
    if tree:
        sVal = '(' + printexp(tree.getLeftChild())
        sVal = sVal + str(tree.getRootVal())
        sVal = sVal + printexp(tree.getRightChild()) + ')'
    return sVal

def postordereval(tree):
    opers = {'+':operator.add, '-':operator.sub,
             '*':operator.mul, '/':operator.truediv}
    res1 = None
    res2 = None
    if tree:
        res1 = postordereval(tree.getLeftChild())
        res2 = postordereval(tree.getRightChild())
        if res1 and res2:
            return opers[tree.getRootVal()](res1,res2)
        else:
            return tree.getRootVal()
```

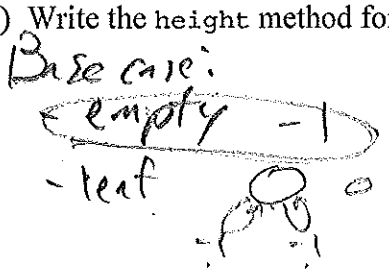
b) If `myTree` is the `BinaryTree` object for the expression: ((4 + 5) * 7), what gets printed by a calls to:

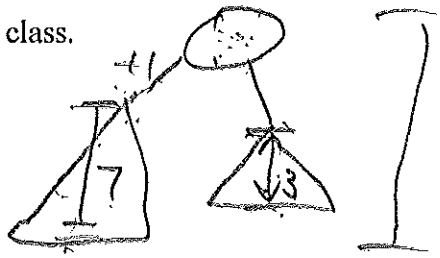
`myTree.inorder()`	`myTree.preorder()`	`myTree.postorder()`	`inorder(myTree)`
4 + 5 * 7	* + 4 5 7	4 5 + 7 *	

c) If `myTree` is the `BinaryTree` object for the expression: ((4 + 5) * 7), what gets printed by a call to `myTree.printexp()`? "((4+5)*7)"

d) If `myTree` is the `BinaryTree` object for the expression: ((4 + 5) * 7), what gets returned by a call to `myTree.postordereval()`? 63

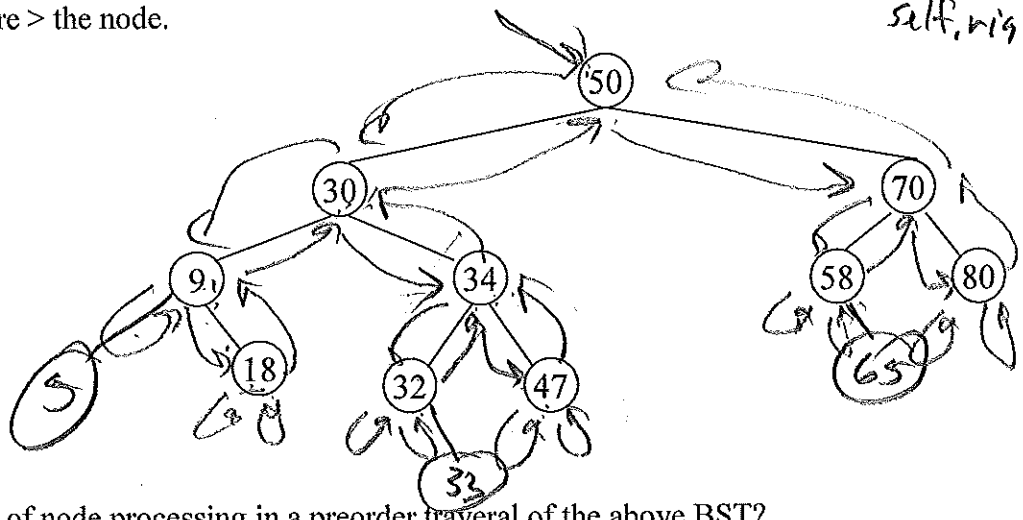
e) Write the `height` method for the `BinaryTree` class.






```
def height(self):
    if self == None:
        return -1
    return 1 + max(self.leftChild.height(), self.rightChild.height())
```

4. Consider the Binary Search Tree (BST). For each node, all values in the left-subtree are < the node and all values in the right-subtree are > the node.



a. What is the order of node processing in a preorder traveral of the above BST?
50, 30, 9, 18, 34, 32, 47, 70, 58, 80,

b. What is the order of node processing in a postorder traveral of the above BST?
18, 9, 32, 47, 34, 30, 58, 80, 70, 50

c. What is the order of node processing in a inorder traveral of the above BST? ascending order

d. Starting at the root, how would you find the node containing "32"?

e. Starting at the root, when would you discover that "65" is not in the BST?

f. What would be the preorder traveral of the BST?

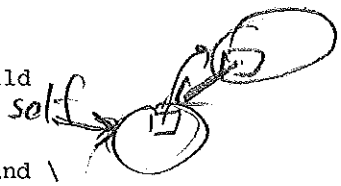
g. What would be the postorder traveral of the BST?

h. Starting at the root, where would be the "easiest" place to add "65"? where it would have been found

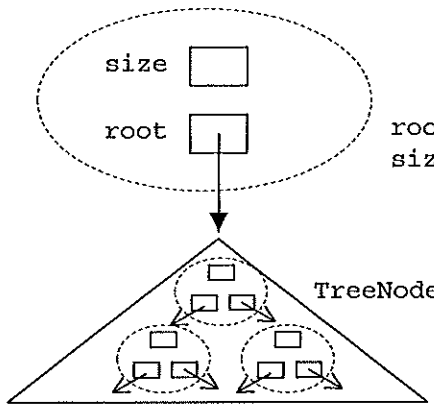
i. Where would we add "5" and "33"?

1. Consider the partial TreeNode class and partial BinarySearchTree class.

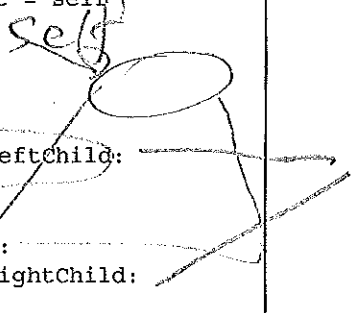
```
class TreeNode:
   def __init__(self,key,val,left=None,right=None,
                                          parent=None):
       self.key = key
       self.payload = val
       self.leftChild = left
       self.rightChild = right
       self.parent = parent

    def hasLeftChild(self):
        return self.leftChild

    def hasRightChild(self):
        return self.rightChild

    def isLeftChild(self):
        return self.parent and \
               self.parent.leftChild == self

    def isRightChild(self):
        return self.parent and \
               self.parent.leftChild == self

    def isRoot(self):
        return not self.parent

    def isLeaf(self):
      return not (self.rightChild or self.leftChild)

    def hasAnyChildren(self):
        return self.rightChild or self.leftChild

    def hasBothChildren(self):
        return self.rightChild and self.leftChild

    def replaceNodeData(self,key,value,lc,rc):
        self.key = key
        self.payload = value
        self.leftChild = lc
        self.rightChild = rc
        if self.hasLeftChild():
            self.leftChild.parent = self
        if self.hasRightChild():
            self.rightChild.parent = self

    def __iter__(self):
       if self:
          if self.hasLeftChild():
               for elem in self.leftChild:
                  yield elem
          yield self.key
          if self.hasRightChild():
               for elem in self.rightChild:
                  yield elem
```



A BinarySearchTree object




```
class BinarySearchTree:
  def __init__(self):
      self.root = None
      self.size = 0

  def length(self):
      return self.size

  def __len__(self):
      return self.size

  def __iter__(self):
      return self.root.__iter__()

  def __str__(self):
   """Returns a string representation of the tree
      rotated 90 degrees counter-clockwise"""

     def strHelper(root, level):
        resultStr = ""
        if root:
           resultStr += strHelper(root.rightChild,
                                  level+1)
           resultStr += "|  " * level
           resultStr += str(root.key) + "\n"
           resultStr += strHelper(root.leftChild,
                                  level+1)
        return resultStr

     return strHelper(self.root, 0)
```



recursive calling of __iter__

a) How do the BinarySearchTree __iter__ and __str__ methods work?

__iter__ calls itself recursively using the "for elem in .." to do an inorder traversal