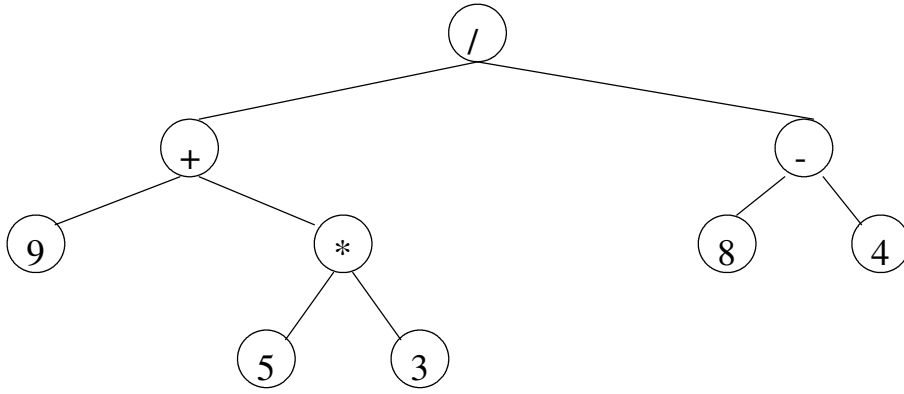


1. Consider the parse tree for  $(9 + (5 * 3)) / (8 - 4)$ :



a) Identify the following items in the above tree:

- node containing “\*”
- edge from node containing “-” to node containing “8”
- root node
- children of the node containing “+”
- parent of the node containing “3”
- siblings of the node containing “\*”
- leaf nodes of the tree
- subtree whose root is node contains “+”
- path from node containing “+” to node containing “5”
- branch from root node to “3”

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?

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:

[ '/', [ '+', \_\_\_\_\_ ], [ '-', \_\_\_\_\_ ] ]

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

```

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

```

graph TD
    Root["key: '*'"] --> Plus["key: '+'"]
    Root --> 7["key: 7"]
    Plus --> 4["key: 4"]
    Plus --> 5["key: 5"]
    7 --> 7L["/"]
    7 --> 7R["/"]
    4 --> 4L["/"]
    4 --> 4R["/"]
    5 --> 5L["/"]
    5 --> 5R["/"]
    
```

```

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)

```

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 call to:

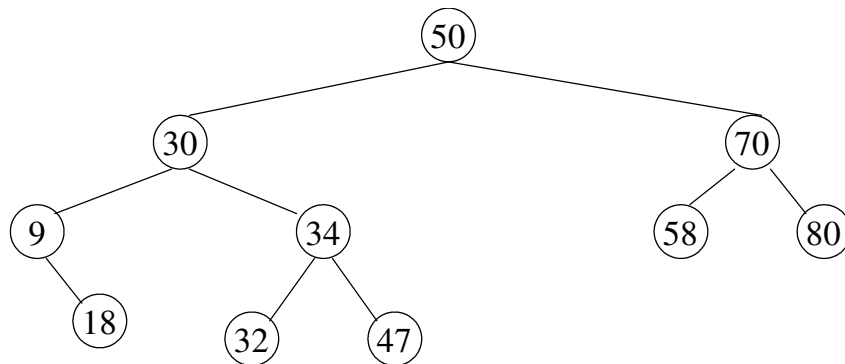
<code>myTree.inorder()</code>	<code>myTree.preorder()</code>	<code>myTree.postorder()</code>	<code>inorder(myTree)</code>

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

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

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

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 traversal of the above BST?

b. What is the order of node processing in a postorder traversal of the above BST?

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

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. Starting at the root, where would be the “easiest” place to add “65”?

g. Where would we add “5” and “33”?