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 who’s 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”, remainder 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 Binary Tree. For the expression \(((4 + 5) * 7)\), the binary tree would be:

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

def _init_(self, rootObj):
    self.key = rootObj
    self.leftChild = None
    self.rightChild = None

class BinaryTree:
    def __init__(self, rootObj):
        self.key = rootObj
        self.leftChild = None
        self.rightChild = None
```
a) Fix the `insertLeft` and `insertRight` code:
(Listing 6.6 and 6.7 are wrong in the text on pp. 242-3)

Some corresponding external (non-class) functions:

```python
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:
            print(')', end=' ')
    else:
        print(self.key)

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
```

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

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()
```

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

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>myTree.inorder()</td>
</tr>
<tr>
<td>myTree.preorder()</td>
</tr>
<tr>
<td>myTree.postorder()</td>
</tr>
<tr>
<td>inorder(myTree)</td>
</tr>
</tbody>
</table>

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

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

e) From an class/Abstract Data Type (ADT) point of view, why are the external versions of the methods “better”?

f) 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.

![Binary Search Tree Diagram]

a. Starting at the root, how would you find the node containing “32”?
b. Starting at the root, when would you discover that “65” is not in the BST?
c. Starting at the root, where would be the “easiest” place to add “65”?
d. What would be the preorder traversal of the BST?
e. Where would we add “5” and “33”?