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

   
   \[
   \begin{array}{c}
   \text{+} \\
   \text{\downarrow} \\
   \text{-} \\
   \text{\downarrow} \\
   \text{1} \\
   \text{2} \\
   \text{3} \\
   \text{4} \\
   \text{5} \\
   \text{6} \\
   \text{7} \\
   \text{8} \\
   \text{9} \\
   \end{array}
   \]

   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 "5"
   - parent of the node containing "3"
   - siblings of the node containing "3"
   - leaf nodes of the tree
   - subtree whose root is node containing "+"
   - 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 looks like:

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

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

   \[
   \text{["+", ["-", ["5", ["4"], ["5"]], ["9", ["3", ["3", ["7"]], ["1", ["8", ["6"]], ["4", ["7"]]]]]]]}
   \]

3. Consider a “linked” representation 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.leftChild = self.leftChild
        self.leftChild = t

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


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


def println(self):
    if self.leftChild:
        self.leftChild.println()
    if self.rightChild:
        self.rightChild.println()


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


def postordereval(self):
    oper = {'+': 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 oper[self.key](res1, res2)
    else:
        return self.key


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


def println(tree):
    if tree:
        println(tree.leftChild)
        print(tree.getRootVal(), end='')
        println(tree.rightChild)


def printexp(tree):
    if tree.leftChild:
        println(tree.leftChild)
    println(tree.getRootVal(), end='')
    if tree.rightChild:
        println(tree.rightChild)


def postordereval(tree):
    oper = {'+': 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 oper[tree.getRootVal()](res1, res2)
    else:
        return tree.getRootVal()
Data Structures

Lecture 18

Name:

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

<table>
<thead>
<tr>
<th>myTree.inorder()</th>
<th>myTree.preorder()</th>
<th>myTree.postorder()</th>
<th>inorder(myTree)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>+</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</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()?

\[
\text{printexp}\left(\left(\left(4 + 5\right)\times 7\right)\right)
\]

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

63

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

The BinaryTree class might not store expression trees, so method
printexp and postorderEval might not make sense. External versions would
know that an expression type is being stored.

f) Write the height method for the BinaryTree class.

```python
def height(self):
    if self.leftChild:
        left_height = self.leftChild.height()
    else:
        left_height = -1
    if self.rightChild:
        right_height = self.rightChild.height()
    else:
        right_height = -1
    if left_height < right_height:
        return right_height + 1
    else:
        return left_height + 1
```

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. 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? 50, 30, 9, 18, 34, 32, 47, 70, 58, 65, 80

e. Where would we add “5” and “33”?