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

a) Indentify the following items in the above tree:
- node containing "9"
- 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 7, 5, 3, 8, 4
- 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)

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

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

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

\[
[[\text{"\text{\texttt{\textbackslash r}}"}, \text{"\text{\texttt{\textbackslash +\texttt{}}}"}, [\text{"\text{\texttt{9}}}]], [\text{"\text{\texttt{\textbackslash -\texttt{}}}"}, [\text{"\text{\texttt{5}}}]], [\text{"\text{\texttt{3}}}]], [[\text{"\text{\texttt{\textbackslash -\texttt{}}}"}, [\text{"\text{\texttt{6}}}]], [\text{"\text{\texttt{4}}}]]] \]

3. Consider a "linked" representations of a BinaryTree. For the expression \(((4 + 5) \times 7)\), 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 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)

    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):
        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)
        return self.key

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

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

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)
    return tree.getRootVal()
Data Structures

Lecture 18

Name:

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

<table>
<thead>
<tr>
<th>\texttt{myTree.inorder()}</th>
<th>\texttt{myTree.preorder()}</th>
<th>\texttt{myTree.postorder()}</th>
<th>\texttt{inorder(myTree)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 n</td>
<td>4 n</td>
<td>5 n</td>
<td>4 n</td>
</tr>
<tr>
<td>+ n</td>
<td>n</td>
<td>+ n</td>
<td>n</td>
</tr>
<tr>
<td>5 n</td>
<td>5 n</td>
<td>7 n</td>
<td>7 n</td>
</tr>
<tr>
<td>\times n</td>
<td>\times n</td>
<td>\times n</td>
<td>\times n</td>
</tr>
<tr>
<td>7 n</td>
<td>7 n</td>
<td>5 n</td>
<td>5 n</td>
</tr>
</tbody>
</table>

\( (4 + 5) \times 7 \)

\( 65 \)

\( 1 \)

\( 0 \)

\( -1 \)

\( \emptytree \)

\( \text{def height(self):} \)
\( \text{if self is None:} \)
\( \text{return -1} \)
\( \text{return max(self.leftChild.height(), self.rightChild.height()) + 1} \)

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

\( 6, 9, 32, 47, 34, 30, 58, 80, 70, 50 \)

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

\( 9, 18, 30, 32, 34, 47, 50, 58, 70, 80 \)

\( \text{ascending order} \)

\( 50, 30, 34, 32 \)

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?

\( 50, 70, 58, \text{no right sub-tree of 58} \)

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

\( \text{right child of 58} \)

g. Where would we add “5” and “33”?

Lecture 18 Page 3