Data Structures Lecture 18

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

   ![Parse Tree Diagram]

   a) Identify the following items in the above tree:
   - node containing "*"
   - edge from node containing "+" to node containing "8"
   - root node (1)
   - children of the node containing "+" (9, 3)
   - parent of the node containing "3" (5)
   - siblings of the node containing "*" (9, 3)
   - leaf nodes of the tree (9, 3, 8, 4)
   - subtree whose root is node contains "+"
   - path from node containing "+" to node containing "5"
   - branch from root node to "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 looks like:

   ```python
   ["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:

   ```python
   [[9, ['+', [5, 3]], 8, [4, 3]]]
   ```

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

   ![Linked Binary Tree Diagram]
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 height(self):
        if self == None:
            return 0
        else:
            return 1 + max(self.leftChild.height(), self.rightChild.height())

# 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:

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('(')
        print(self.key)
        if self.rightChild:
            print(')')

    def postorder(self):
        if self.leftChild:
            self.leftChild.postorder()
        if self.rightChild:
            self.rightChild.postorder()
        return self

    def height(self):
        return 1 + max(self.leftChild.height(), self.rightChild.height())

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

    def postOrder(self):
        self.leftChild.postorder()
        self.rightChild.postorder()
        return self

    def height(self):
        return 1 + max(self.leftChild.height(), self.rightChild.height())

    def printexp(self):
        if self.leftChild:
            sVal = '(' + printexp(self.leftChild) + ')'
        if self.rightChild:
            sVal += printexp(self.rightChild) + ')'
        return sVal

    def eval(self):
        if self.key == '+':
            return self.leftChild.eval() + self.rightChild.eval()
        else:
            return self.eval()
b) If myTree is the BinaryTree object for the expression: \((4 + 5) \times 7\), what gets printed by a calls to:

- myTree.inorder()
- myTree.preorder()
- myTree.postorder()
- inorder(myTree)

<table>
<thead>
<tr>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>5</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>9</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()?

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

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

63

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.

![Binary Search Tree Diagram]

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?

18 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?

sorted 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. Starting at the root, where would be the “easiest” place to add “65”? Where would we look for them?

g. Where would we add “5” and “33”? 