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

a) Identify the following items in the above tree:
- node containing "*"
- edge from node containing "-" to node containing "4"
- root node
- children of the node containing "+" (9, 5)
- parent of the node containing "3"
- siblings of the node containing "*"
- leaf nodes of the tree (9, 5)
- 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:

\[
[\text{"/"}, [\text{"+"}, [\text{"9"}], [\text{"/"}, [\text{"5"}][\text{"3"}]], [\text{"-"}, [\text{"8"}][\text{"4"}]]]]
\]

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

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

```python
def myTree:
    key = '+'
    leftChild = None
    rightChild = None
```

```python
def key '4':
    leftChild = None
    rightChild = None
```

```python
def key '5':
    leftChild = None
    rightChild = None
```

```python
def key '7':
    leftChild = None
    rightChild = None
```
Data Structures

import operator

class BinaryTreeNode:
    def __init__(self, obj):
        self.key = obj
        self.leftChild = None
        self.rightChild = None

    def insertLeft(self, newNode):
        if self.leftChild == None:
            self.leftChild = BinaryTreeNode(newNode)
        else:
            t = BinaryTreeNode(newNode)
            t.leftChild = self.leftChild
            self.leftChild = t

    def insertRight(self, newNode):
        if self.rightChild == None:
            self.rightChild = BinaryTreeNode(newNode)
        else:
            t = BinaryTreeNode(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 preorder(self):
        if self.leftChild:
            self.leftChild.preorder()
        print(self.key)
        if self.rightChild:
            self.rightChild.preorder()

    def printexp(self):
        if self.leftChild:
            print('(', end=' ')
            self.leftChild.printexp()
            print('+', end=' ')
        if self.key:
            print('+', end=' ')
        if self.rightChild:
            self.rightChild.printexp()
            print(')', end=' ')

    def postorder(self):
        if self.leftChild:
            self.leftChild.postorder()
            print('(', end=' ')
        if self.key:
            print('+', end=' ')
        if self.rightChild:
            self.rightChild.postorder()
            print(')', end=' ')

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('(', end=' ')
    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 preorder(tree):
    if tree:
        sVal = ''
        if tree:
            sVal = '(' + printexp(tree.getLeftChild()) + ')
        sVal = sVal + str(tree.getRootVal())
        sVal = sVal + printexp(tree.getRightChild()) + ')
        return sVal

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

def postorderEval(tree):
    operators = {'+':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 operators[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) * 7`, what gets printed by a call to:

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


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

\[\text{\texttt{((4+5)*7)}}\]


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

63

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

```
def height(self):
    if self == None:
        return -1
    return 1 + max(self.left.height(),
                    self.right.height())
```

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.

```
50 - max(30, 70) - 32 - max(18, 47) - 34 - max(9, 47)
```

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

50, 30, 18, 9, 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?

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. What would be the preorder traversal of the BST?

```
50, 30, 18, 9, 34, 32, 47, 70, 58, 80
```

g. What would be the postorder traversal of the BST?

```
18, 9, 32, 47, 34, 30, 58, 80, 70, 50
```

h. Starting at the root, where would be the "easiest" place to add "65"?

```
where it would have been found
```

i. Where would we add "5" and "33"?
1. Consider the partial TreeNode class and partial BinarySearchTree class.

```python
class TreeNode:
    def __init__(self, key, val, left=None, right=None, parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent

    def hasLeftChild(self):
        return self.leftChild

    def hasRightChild(self):
        return self.rightChild

    def isLeftChild(self):
        return self.parent and self.parent.leftChild == self

    def isRightChild(self):
        return self.parent and self.parent.rightChild == self

    def isRoot(self):
        return not self.parent

    def isLeaf(self):
        return not (self.rightChild or self.leftChild)

    def hasAnyChildren(self):
        return self.rightChild or self.leftChild

    def hasBothChildren(self):
        return self.rightChild and self.leftChild

    def replaceNodeData(self, key, value, lc, rc):
        self.key = key
        self.payload = value
        self.leftChild = lc
        self.rightChild = rc
        if self.hasLeftChild():
            self.leftChild.parent = self
        if self.hasRightChild():
            self.rightChild.parent = self

    def __iter__(self):
        if self:
            if self.hasLeftChild():
                for elem in self.leftChild:
                    yield elem
            yield self.key
            if self.hasRightChild():
                for elem in self.rightChild:
                    yield elem
```

A BinarySearchTree object

![BinarySearchTree diagram](image)

```python
class BinarySearchTree:
    def __init__(self):
        self.root = None
        self.size = 0

    def length(self):
        return self.size

    def __len__(self):
        return self.size

    def __iter__(self):
        return self.root.__iter__()

    def __str__(self):
        """Returns a string representation of the tree
        rotated 90 degrees counter-clockwise""

        def strHelper(root, level):
            resultStr = ""
            if root:
                resultStr += strHelper(root.rightChild, level+1)
                resultStr += "| " * level
                resultStr += str(root.key) + "\n"
                resultStr += strHelper(root.leftChild, level+1)
            return resultStr

        return strHelper(self.root, 0)
```

a) How do the BinarySearchTree __iter__ and __str__ methods work?

- __iter__ calls itself recursively using the "for elem in..."
  to do an in-order traversal.