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>7</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>(+)</td>
<td>(+)</td>
<td>(+)</td>
<td></td>
</tr>
<tr>
<td>(\times)</td>
<td>(\times)</td>
<td>(\times)</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()`?

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

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

\[
\text{height}(T) = \max(\text{level}(\text{left}), \text{level}(\text{right})) + 1
\]

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?
   \[18 \ 9 \ 32 \ 47 \ 34 \ 30 \ 58 \ 80 \ 70 \ 50\]

c. What is the order of node processing in an inorder traversal of the above BST?
   \[\text{sorted order}\]

d. Starting at the root, how would you find the node containing “32”?
   \[\text{left of } 50, \text{ right } 70, \text{ left } 34, \text{ find } 32\]

e. Starting at the root, when would you discover that “65” is not in the BST?
   \[\text{walk off the bottom of a branch - no right child of 58}\]

f. Starting at the root, where would be the “easiest” place to add “65”?
   \[\text{add as a leaf where we would have found it}\]

g. 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

- **root**
  - **size**
    - root is None if size is 0

```
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(self, 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)
```

Non-standard traversal:
- Put right subtree in string
- Put root node in string
- Put left subtree in string

a) How do the BinarySearchTree __iter__ and __str__ methods work?
More partial TreeNode class and partial BinarySearchTree class.

```python
class BinarySearchTree:
    ...
    def __contains__(self, key):
        if self._get(key, self.root):
            return True
        else:
            return False
    def get(self, key):
        if self.root:
            res = self._get(key, self.root)
            if res:
                return res.payload
            else:
                return None
        else:
            return None
    def _get(self, key, currentNode):
        if not currentNode:
            return None
        elif currentNode.key == key:
            return currentNode
        elif key < currentNode.key:
            return self._get(key, currentNode.leftChild)
        else:
            return self._get(key, currentNode.rightChild)
    def __getitem__(self, key):
        return self.get(key)
    def __setitem__(self, k, v):
        self.put(k, v)
    def put(self, key, val):
        if self.root:
            self._put(key, val, self.root)
        else:
            self.root = TreeNode(key, val)
            self.size = 1
    def _put(self, key, val, currentNode):
        if key < currentNode.key:
            if currentNode.hasLeftChild():
                self._put(key, val, currentNode.leftChild)
            else:
                temp = currentNode.leftChild
                currentNode.leftChild = temp
                temp.parent = currentNode
            elif key > currentNode.key:
                if currentNode.hasRightChild():
                    self._put(key, val, currentNode.rightChild)
                else:
                    currentNode.rightChild = TreeNode(key, val)
                    currentNode.rightChild.parent = currentNode
                    self.size += 1
        else:
            currentNode.payload = val
```

b) The `get` method is the "work horse" of BST search. It recursively walks `currentNode` down the tree until it finds `key` or becomes `None`.
In English, what are the base and recursive cases?

Base cases:
1) empty subtree if `currentNode` is `None`
2) `currentNode` points to node with `key`
3) if `key < currentNode.key` check left subtree
4) else check right subtree

What is the `put` method doing?
Checks if BST is empty and handle adding first node.
Otherwise, add to existing tree node.

d) Complete the recursive `put` method.

e) Draw the "shape" of the BST after `puts` of: 50, 60, 30, 70, 90, 40, 65

f) If "n" items are in the BST, what is `put`'s: Best-case $O(\_\_\_\_\_)$? Worst-case $O(\_\_\_\_\_)$? Average-case $O(\_\_\_\_\_)$?