

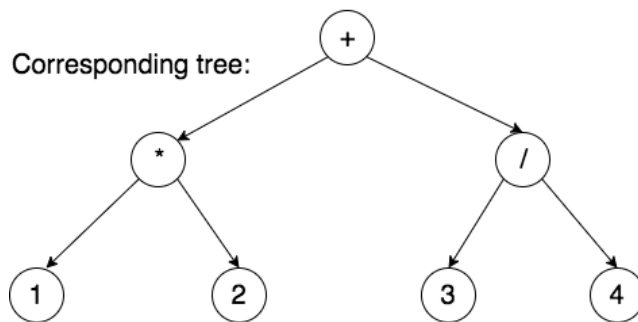
CSCI-SHU 210 Data Structures

Recitation8 Worksheet Trees/Binary trees

In this recitation, we will practice **Binary Tree**.

Today's goals:

- Please keep in mind, what is a
 - **General Tree**
 - **Binary Tree** (this recitation)
 - **Binary Search Tree** (next week)
 - **AVL Tree** (next week)
- Binary tree's property: left child / right child only
- Binary tree traversal algorithms (Pre, in, post, level)
- Can perform recursion on Binary tree data structure.
- Arithmetic expression tree

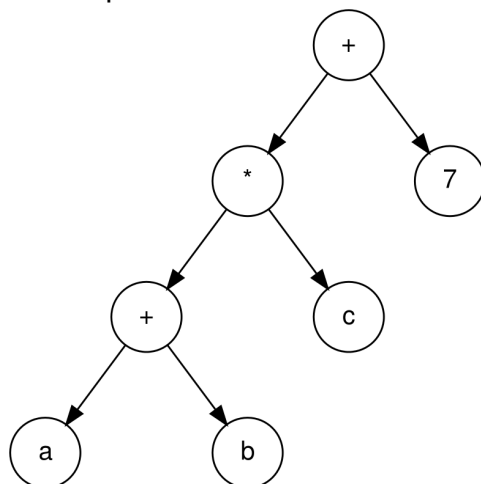


Preorder traversal:
Inorder traversal:
Postorder traversal:

pre. + * 1 2 / 3 4
In 1 * 2 + 3 / 4
post 1 2 * 3 4 / +

+ * 1 2 / 3 4
1 * 2 + 3 / 4
1 2 * 3 4 / +

Warmup Exercise:



What is the preorder traversal result for the arithmetic expression tree above?

+ * + a b c 7

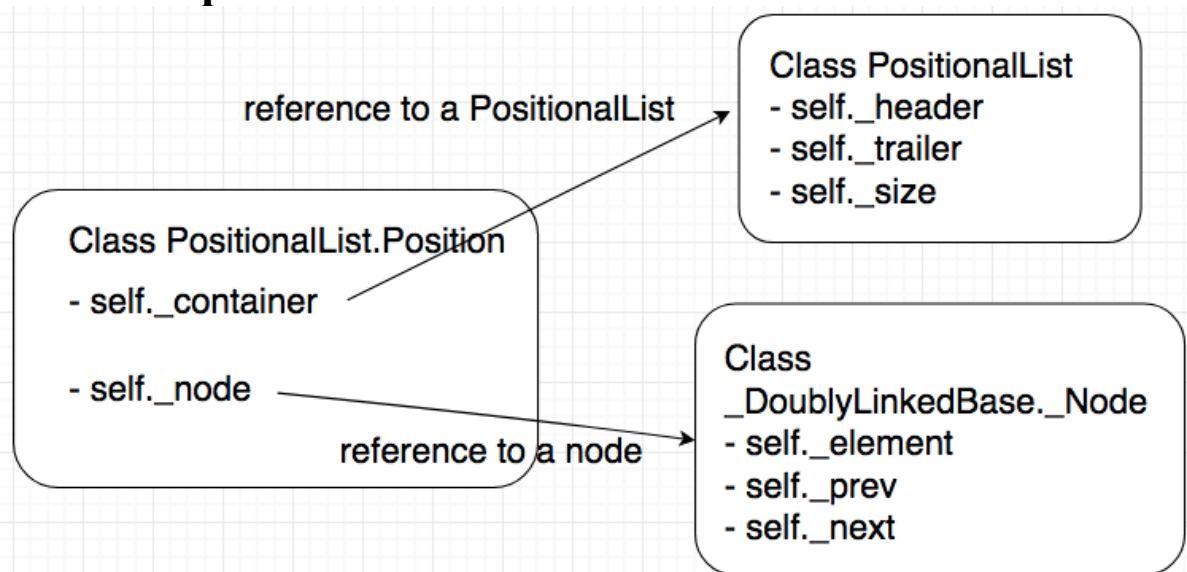
What is the inorder traversal result for the arithmetic expression tree above?

a + b * c + 7

What is the postorder traversal result for the arithmetic expression tree above?

a b + c * 7 +

Part I: A quick look at Positional List:



```
1. def _validate(self, p):
2.     """Return position's node, or raise appropriate error if invalid."""
3.     if not isinstance(p, self.Position):
4.         raise TypeError('p must be proper Position type')
5.     if p._container is not self:
6.         raise ValueError('p does not belong to this container')
7.     if p._node._next is None: #convention for deprecated nodes
8.         raise ValueError('p is no longer valid')
9.     return p._node
```

Your task 1 (Optional): Take a look at class `PositionalList`. Try the test code, then type some test code on your own. Try to understand this class's structure.

You don't need submit anything for task 1.

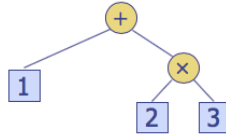
Part II: Simple Binary Tree (Just the Tree node class):

Your task 2: Let's continue from where we left off. Before we start OOP style, let's start from just the binary tree node.

Create a Simple tree for this infix expression: $3 * 2 + 5 - 2$

Simple Binary Tree (w/o parent)

```
class TreeWithoutParent:
    def __init__(self, element, left=None, right=None):
        self.element = element
        self.left = left
        self.right = right
    def __str__(self):
        return str(self.element)
```



Your task 3: Implement function `PreOrderTraversal(tree)` in `simple_Tree_without_parent.py`.

This function should:

Prints all the elements with pre order traversal, the initial call parameter `tree` is the root node.

Your task 4: Implement function `InOrderTraversal(tree)` in `simple_Tree_without_parent.py`.

This function should:

Prints all the elements with in order traversal, the initial call parameter `tree` is the root node.

Your task 5: Implement function `PostOrderTraversal(tree)` in `simple_Tree_without_parent.py`.

This function should:

Prints all the elements with post order traversal, the initial call parameter `tree` is the root node.

Your task 6: Implement function `LevelOrderTraversal(tree)` in `simple_Tree_without_parent.py`.

This function should:

Prints all the elements with Level order traversal, the initial call parameter `tree` is the root node.

Part III: Binary Tree with OOP:

1. Getting familiar with 300 lines of code.

- **Class Tree:**

- class TreeNode
 - self._element
 - self._parent
 - self._left
 - self._right

- `__len__(self)`

----- Traversal functions -----

- `__iter__(self)`
- `children(self, node)`
- `preorder(self)`
- `inorder(self)`
- `postorder(self)`
- `_subtree_preorder(self, node)`
- `_subtree_inorder(self, node)`
- `_subtree_postorder(self, node)`
- `nodes(self)`

----- Boolean functions -----

- `is_root(self, node)`
- `is_leaf(self, node)`
- `is_empty(self)`

----- Function accessors -----

- `root(self)`
- `parent(self, node)`
- `left(self, node)`
- `right(self, node)`
- `sibling(self, node)`
- `num_children(self, node)`

----- Public mutators -----

- `add_root(self, e)`
- `add_left(self, node, e)`
- `add_right(self, node, e)`
- `replace(self, node, e)`
- `delete(self, node)`
- `attach(self, node, t1, t2)`

----- Pretty printing -----

- `pretty_print` methods

----- Today's tasks, starting at line 229 -----

- `preorderPrint(self, node)` # Task 1, optional
- `postorderPrint(self, node)` # Task 2, optional
- `inorderPrint(self, node)` # Task 3, optional
- `levelorderPrint(self, node)` # Task 4
- `height(self, node = None)` # Task 5
- `depth(self, node)` # Task 6
- `return_max(self)` # Task 7
- `flip_node(self, node)` # Task 8
- `flip_subtree(self, node)` # Task 9

Your task 1 (Optional): Implement function `preorderPrint(self,node)` in `BinaryTree.py`.

This function should:

Prints all the elements with pre order traversal, treating parameter node as the root Position.

Your task 2 (Optional): Implement function `postorderPrint(self, node)` in `BinaryTree.py`.

This function should:

Prints all the elements with post order traversal, treating parameter node as the root Position.

Your task 3 (Optional): Implement function `inorderPrint(self, node)` in `BinaryTree.py`.

This function should:

Prints all the elements with in order traversal, treating parameter node as the root Position.

Your task 4 (Optional): Implement function `levelorderPrint(self, node)` in `BinaryTree.py`.

This function should:

Prints all the elements with level order traversal, treating parameter node as the root Position.

Your task 5: Implement function `height(self, node = None)` in `BinaryTree.py`.

This function should:

Return the height of the subtree rooted at a given node.

If node is None, return the height of the entire (self) tree.

Your task 6: Implement function `depth(self, node)` in `BinaryTree.py`.

This function should:

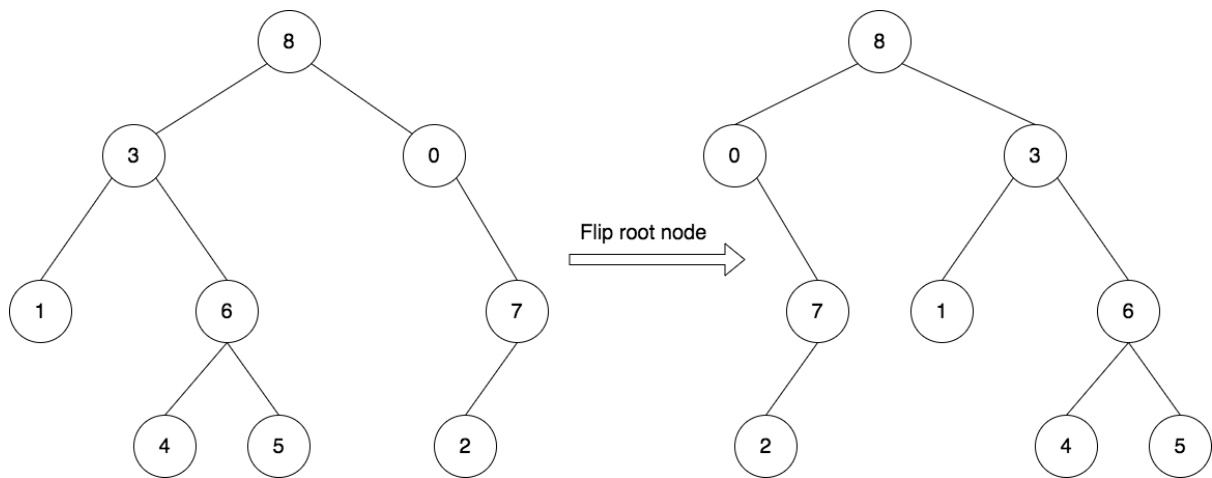
Return the depth of a given node, in the entire (self) tree.

Your task 7: Implement function `return_max(self)` in `BinaryTree.py`.

This function should:

Traverse the tree and return the maximum value stored within the tree.

Your task 8: Implement method `flip(self, node)` in `BinaryTree.py`, which flips the left and right children of a given node.



Your task 9: Implement method `flip_subtree(self, node=None)` in `BinaryTree.py` which flips the left and right children all nodes in the subtree of given node, and if `p` is omitted it flips the entire tree.

Your method must be recursive.

