```
!date
```

```
Mon Apr 11 03:11:13 UTC 2022
```

Please run the above line to refresh the date before your submission.

CSCI-SHU 210 Data Structures

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

Recitation 9 Trees/Binary trees

```
class LinkedQueue:
    """FIFO queue implementation using a singly linked list for storage."""

    #------------------------- nested _Node class -------------------------
    class _Node:
        """Lightweight, nonpublic class for storing a singly linked node."""
        __slots__ = '_element', '_next'         # streamline memory usage

        def __init__(self, element, next):
            self._element = element
            self._next = next

    #------------------------------- queue methods -------------------------------
    def __init__(self):
        """Create an empty queue."""
        self._head = None
        self._tail = None
        self._size = 0                          # number of queue elements

    def __len__(self):
        """Return the number of elements in the queue."""
        return self._size

    def is_empty(self):
        """Return True if the queue is empty."""
        return self._size == 0

    def first(self):
        """Return (but do not remove) the element at the front of the queue.

        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Exception('Queue is empty')
        return self._head._element              # front aligned with head of list

    def dequeue(self):
        """Remove and return the first element of the queue (i.e., FIFO).

        Raise Empty exception if the queue is empty.
        """
        if self.is_empty():
            raise Exception('Queue is empty')
        answer = self._head._element
        self._head = self._head._next
        self._size -= 1
        if self.is_empty():                     # special case as queue is empty
            self._tail = None                   # removed head had been the tail
        return answer

    def enqueue(self, e):
```

```
        """Add an element to the back of queue."""
        newest = self._Node(e, None)              # node will be new tail node
        if self.is_empty():
            self._head = newest                   # special case: previously empty
        else:
            self._tail._next = newest
        self._tail = newest                       # update reference to tail node
        self._size += 1

    def __str__(self):
        result = []
        curNode = self._head
        while (curNode is not None):
            result.append(str(curNode._element) + " --> ")
            curNode = curNode._next
        result.append("None")
        return "".join(result)
```

```
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)

def PreOrderTraversal(tree):
    # Prints all the elements with pre order traversal, the initial call parameter tree is the root node.
    if tree == None:
        return
    # to do
    print(tree._element, end = " ")
    PreOrderTraversal(tree._left)
    PreOrderTraversal(tree._right)

def PostOrderTraversal(tree):
    # Prints all the elements with post order traversal, the initial call parameter tree is the root node.
    if tree == None:
        return
    # to do
    PostOrderTraversal(tree._left)
    PostOrderTraversal(tree._right)
    print(tree._element, end = " ")

def InOrderTraversal(tree):
    # Prints all the elements with in order traversal, the initial call parameter tree is the root node.
    if tree == None:
        return
    # to do
    InOrderTraversal(tree._left)
    print(tree._element, end = " ")
    InOrderTraversal(tree._right)

def LevelOrderTraversal(tree):
    # Prints all the elements with level order traversal, the initial call parameter tree is the root node.
    if tree == None:
        return
    # to do
    queue = LinkedQueue()
    queue.enqueue(tree)
    while not queue.is_empty():
        next_node = queue.dequeue()
        print(next_node._element, end = " ")
        if next_node._left:
            queue.enqueue(next_node._left)
        if next_node._right:
            queue.enqueue(next_node._right)
```

```
### Uncomment the following code if you want to print the tree.
### We assume that you had variables:  _left, _right, _element in the TreeWithoutParent Class.


def pretty_print(A):
    levels = 3          # Need a function to calculate levels. Use 3 for now.
    print_internal([A], 1, levels)

def print_internal(this_level_nodes, current_level, max_level):
    if (len(this_level_nodes) == 0 or all_elements_are_None(this_level_nodes)):
        return   # Base case of recursion: out of nodes, or only None left

    floor = max_level - current_level;
    endgeLines = 2 ** max(floor - 1, 0);
    firstSpaces = 2 ** floor - 1;
    betweenSpaces = 2 ** (floor + 1) - 1;
    print_spaces(firstSpaces)
    next_level_nodes = []
    for node in this_level_nodes:
        if (node is not None):
            print(node._element, end = "")
            next_level_nodes.append(node._left)
            next_level_nodes.append(node._right)
        else:
            next_level_nodes.append(None)
            next_level_nodes.append(None)
            print_spaces(1)

        print_spaces(betweenSpaces)
    print()
    for i in range(1, endgeLines + 1):
        for j in range(0, len(this_level_nodes)):
            print_spaces(firstSpaces - i)
            if (this_level_nodes[j] == None):
                    print_spaces(endgeLines + endgeLines + i + 1);
                    continue
            if (this_level_nodes[j]._left != None):
                    print("/", end = "")
            else:
                    print_spaces(1)
            print_spaces(i + i - 1)
            if (this_level_nodes[j]._right != None):
                    print("\\", end = "")
            else:
                    print_spaces(1)
            print_spaces(endgeLines + endgeLines - i)
        print()

    print_internal(next_level_nodes, current_level + 1, max_level)

def all_elements_are_None(list_of_nodes):
    for each in list_of_nodes:
        if each is not None:
            return False
    return True

def print_spaces(number):
    for i in range(number):
        print("   ", end = "")



print("\nUsing Tree Data Structure Without a parent")

##Create a expression tree for this expression:  3*2 + 5-2"
## to do
tree = TreeWithoutParent("+")
tree._left = TreeWithoutParent("*")
tree._right = TreeWithoutParent("-")
tree._left._left = TreeWithoutParent("3")
tree._left._right = TreeWithoutParent("2")
tree._right._left = TreeWithoutParent("5")
```

```
tree._right._left = TreeWithoutParent("5")
tree._right._right = TreeWithoutParent("2")

pretty_print(tree) #Call pretty_print to print the tree

print("\nPreOrder:")
PreOrderTraversal(tree)              # should print: + * 3 2 - 5 2
print("\nPostOrder:")
PostOrderTraversal(tree)        # should print: 3 2 * 5 2 - +
print("\nInOrder:")
InOrderTraversal(tree)               # should print: 3 * 2 + 5 - 2
print("\nLevelOrderOrder:")
LevelOrderTraversal(tree)               # should print: + * - 3 2 5 2
```

```
Using Tree Data Structure Without a parent
      +
    /  \
  /      \
  *      -
/  \  /  \
3  2  5  2


PreOrder:
+ * 3 2 - 5 2
PostOrder:
3 2 * 5 2 - +
InOrder:
3 * 2 + 5 - 2
LevelOrderOrder:
+ * - 3 2 5 2
```

```
class Tree:
        class TreeNode:
                def __init__(self, element, parent = None, left = None, right = None):
                        self._parent = parent
                        self._element = element
                        self._left = left
                        self._right = right

                def __str__(self):
                        return str(self._element)

        #-------------------------- binary tree constructor --------------------------
        def __init__(self):
                """Create an initially empty binary tree."""
                self._root = None
                self._size = 0

        #-------------------------- public accessors ---------------------------------
        def __len__(self):
                """Return the total number of elements in the tree."""
                return self._size

        def is_root(self, node):
                """Return True if a given node represents the root of the tree."""
                return self._root == node

        def is_leaf(self, node):
                """Return True if a given node does not have any children."""
                return self.num_children(node) == 0

        def is_empty(self):
                """Return True if the tree is empty."""
                return len(self) == 0

        def __iter__(self):
                """Generate an iteration of the tree's elements."""
                for node in self.nodes():                                    # use same order as nodes()
                        yield node._element                                          # but yield each element
```

```
    def nodes(self):
        """Generate an iteration of the tree's nodes."""
        return self.preorder()                                      # return entire preorder iteration

    def preorder(self):
        """Generate a preorder iteration of nodes in the tree."""
        if not self.is_empty():
            for node in self._subtree_preorder(self._root):   # start recursion
                yield node

    def _subtree_preorder(self, node):
        """Generate a preorder iteration of nodes in subtree rooted at node."""
        yield node                                                                        # visit node before i
        for c in self.children(node):                                             # for each child c
            for other in self._subtree_preorder(c):                   # do preorder of c's subtree
                yield other                                                               # yielding each to ou

    def postorder(self):
        """Generate a postorder iteration of nodes in the tree."""
        if not self.is_empty():
            for node in self._subtree_postorder(self._root):   # start recursion
                yield node

    def _subtree_postorder(self, node):
        """Generate a postorder iteration of nodes in subtree rooted at node."""
        for c in self.children(node):                                             # for each child c
            for other in self._subtree_postorder(c):                 # do postorder of c's subtree
                yield other                                                               # yielding each to ou
        yield node                                                                        # visit node after it
    def inorder(self):
        """Generate an inorder iteration of positions in the tree."""
        if not self.is_empty():
          for node in self._subtree_inorder(self._root):
            yield node

    def _subtree_inorder(self, node):
        """Generate an inorder iteration of positions in subtree rooted at p."""
        if node._left is not None:                    # if left child exists, traverse its subtree
          for other in self._subtree_inorder(node._left):
            yield other
        yield node                                                     # visit p between its subtrees
        if node._right is not None:                  # if right child exists, traverse its subtree
          for other in self._subtree_inorder(node._right):
            yield other


    def breadthfirst(self):
        """Generate a breadth-first iteration of the nodes of the tree."""
        if not self.is_empty():
            fringe = LinkedQueue()                          # known nodes not yet yielded
            fringe.enqueue(self._root)                 # starting with the root
            while not fringe.is_empty():
                node = fringe.dequeue()                         # remove from front of the queue
                yield node                                                # report this node
                for c in self.children(node):
                    fringe.enqueue(c)                             # add children to back of queue


    def root(self):
        """Return the root of the tree (or None if tree is empty)."""
        return self._root

    def parent(self, node):
        """Return node's parent (or None if node is the root)."""
        return node._parent

    def left(self, node):
        """Return node's left child (or None if no left child)."""
        return node._left
```

```python
    def right(self, node):
        """Return node's right child (or None if no right child)."""
        return node._right

    def children(self, node):
        """Generate an iteration of nodes representing node's children."""
        if node._left is not None:
            yield node._left
        if node._right is not None:
            yield node._right

    def num_children(self, node):
        """Return the number of children of a given node."""
        count = 0
        if node._left is not None:       # left child exists
            count += 1
        if node._right is not None:      # right child exists
            count += 1
        return count

    def sibling(self, node):
        """Return a node representing given node's sibling (or None if no sibling)."""
        parent = node._parent
        if parent is None:                                  # p must be the root
            return None                                     # root has no sibling
        else:
            if node == parent._left:
                return parent._right                # possibly None
            else:
                return parent._left                 # possibly None

    #-------------------------- nonpublic mutators --------------------------
    def add_root(self, e):
        """Place element e at the root of an empty tree and return the root node.

        Raise ValueError if tree nonempty.
        """
        if self._root is not None:
            raise ValueError('Root exists')
        self._size = 1
        self._root = self.TreeNode(e)
        return self._root

    def add_left(self, node, e):
        """Create a new left child for a given node, storing element e in the new node.

        Return the new node.
        Raise ValueError if node already has a left child.
        """
        if node._left is not None:
            raise ValueError('Left child exists')
        self._size += 1
        node._left = self.TreeNode(e, node)                 # node is its parent
        return node._left

    def add_right(self, node, e):
        """Create a new right child for a given node, storing element e in the new node.

        Return the new node.
        Raise ValueError if node already has a right child.
        """
        if node._right is not None:
            raise ValueError('Right child exists')
        self._size += 1
        node._right = self.TreeNode(e, node)                # node is its parent
        return node._right

    def _replace(self, node, e):
        """Replace the element at given node with e, and return the old element."""
        old = node._element
        node._element = e
```

```
        return old

    def _delete(self, node):
        """Delete the given node, and replace it with its child, if any.

        Return the element that had been stored at the given node.
        Raise ValueError if node has two children.
        """
        if self.num_children(node) == 2:
            raise ValueError('Position has two children')
        child = node._left if node._left else node._right   # might be None
        if child is not None:
            child._parent = node._parent         # child's grandparent becomes parent
        if node is self._root:
            self._root = child                        # child becomes root
        else:
            parent = node._parent
            if node is parent._left:
                parent._left = child
            else:
                parent._right = child
        self._size -= 1
        return node._element



    def _attach(self, node, t1, t2):
        """Attach trees t1 and t2, respectively, as the left and right subtrees of the external node.

        As a side effect, set t1 and t2 to empty.
        Raise TypeError if trees t1 and t2 do not match type of this tree.
        Raise ValueError if node already has a child. (This operation requires a leaf node!)
        """
        if not self.is_leaf(node):
            raise ValueError('position must be leaf')
        if not type(self) is type(t1) is type(t2):      # all 3 trees must be same type
            raise TypeError('Tree types must match')
        self._size += len(t1) + len(t2)
        if not t1.is_empty():                # attached t1 as left subtree of node
            t1._root._parent = node
            node._left = t1._root
            t1._root = None                          # set t1 instance to empty
            t1._size = 0
        if not t2.is_empty():                # attached t2 as right subtree of node
            t2._root._parent = node
            node._right = t2._root
            t2._root = None                          # set t2 instance to empty
            t2._size = 0

    ## Task 1 ##
    ## Method 1 ##
    def preorderPrint(self,node):
        if node == None:
            return
        for each in self._subtree_preorder(node):
            print(each._element, end = " ")

    ## Task 2 ##
    def postorderPrint(self,node):
        if node == None:
            return
        for each in self._subtree_postorder(node):
            print(each._element, end = " ")

    ## Task 3 ##
    def inorderPrint(self,node):
        if node == None:
            return
        for each in self._subtree_inorder(node):
            print(each._element, end = " ")
```

```
    ## Task 4 ##
    def levelorderPrint(self, node):
        queue = LinkedQueue()
        queue.enqueue(node)
        while not queue.is_empty():
            next_node = queue.dequeue()
            print(next_node._element, end = " ")
            if next_node._left:
                queue.enqueue(next_node._left)
            if next_node._right:
                queue.enqueue(next_node._right)


    ## Task 5 ##
    ## Method 1 ##
    def height(self, node = None):    # time is linear in size of subtree
        if node is None:
            node = self._root
        if self.is_leaf(node):
            return 0
        else:
            l = self.height(node._left)
            r = self.height(node._right)
            return 1 + max(l, r)


    ## Method 2 ##
##    def height2(self, node=None):
##        """Return the height of the subtree rooted at a given node.
##
##        If node is None, return the height of the entire tree.
##        """
##        if node is None:
##            node = self._root
##        if self.is_leaf(node):
##            return 0
##        else:
##            return 1 + max(self.height2(c) for c in self.children(node))


    ## Task 6 ##
    def depth(self, node):
        if self.is_root(node):
            return 0
        else:
            return 1 + self.depth(node._parent)


    ## Task 7 ##
    def return_max(self):
        maximum = self._root._element
        for each in self:   # This calls inorder traversal
            if each > maximum:
                maximum = each
        return maximum


    ## Task 8 ##
    def flip_node(self, node):
        node._left, node._right = node._right, node._left


    ## Task 9 ##
    def flip_tree(self, node = None):
        if node == None:
            node = self._root
        node._left, node._right = node._right, node._left
        if node._left:
            self.flip_tree(node._left)
        if node._right:
            self.flip_tree(node._right)
```

```
def pretty_print(tree):
    # ----------------------- Need to enter height to work -----------------
    levels = tree.height() + 1
    print("Levels:", levels)
    print_internal([tree._root], 1, levels)

def print_internal(this_level_nodes, current_level, max_level):
    if (len(this_level_nodes) == 0 or all_elements_are_None(this_level_nodes)):
        return  # Base case of recursion: out of nodes, or only None left

    floor = max_level - current_level;
    endgeLines = 2 ** max(floor - 1, 0);
    firstSpaces = 2 ** floor - 1;
    betweenSpaces = 2 ** (floor + 1) - 1;
    print_spaces(firstSpaces)
    next_level_nodes = []
    for node in this_level_nodes:
        if (node is not None):
            print(node._element, end = "")
            next_level_nodes.append(node._left)
            next_level_nodes.append(node._right)
        else:
            next_level_nodes.append(None)
            next_level_nodes.append(None)
            print_spaces(1)

        print_spaces(betweenSpaces)
    print()
    for i in range(1, endgeLines + 1):
        for j in range(0, len(this_level_nodes)):
            print_spaces(firstSpaces - i)
            if (this_level_nodes[j] == None):
                    print_spaces(endgeLines + endgeLines + i + 1);
                    continue
            if (this_level_nodes[j]._left != None):
                    print("/", end = "")
            else:
                    print_spaces(1)
            print_spaces(i + i - 1)
            if (this_level_nodes[j]._right != None):
                    print("\\", end = "")
            else:
                    print_spaces(1)
            print_spaces(endgeLines + endgeLines - i)
        print()

    print_internal(next_level_nodes, current_level + 1, max_level)

def all_elements_are_None(list_of_nodes):
    for each in list_of_nodes:
        if each is not None:
            return False
    return True

def print_spaces(number):
    for i in range(number):
        print(" ", end = "")
```

```
''' The following code will construct this tree:
                        -
                    / \
                    /     \
                /          \
                /              \
                *              +
            /  \        /  \
            /      \      /      \
            +      4      -      2
```

```
       / \         / \
       3  1        9  5
'''
t = Tree()
a = t.add_root("-")
b = t.add_left(a, "*")
c = t.add_right(a, "+")
d = t.add_left(b, "+")
e = t.add_right(b, 4)
t.add_left(d, 3)
t.add_right(d, 1)
f = t.add_left(c, "-")
t.add_right(c, 2)
t.add_left(f, 9)
k = t.add_right(f, 5)
pretty_print(t)

print("-------------------Testing task 1 preorder--------------------")
t.preorderPrint(a)   # a is the root
print()
print("-------------------Testing task 2 inorder--------------------")
t.inorderPrint(a)  # a is the root
print()
print("-------------------Testing task 3 postorder--------------------")
t.postorderPrint(a)   # a is the root
print()
print("-------------------Testing task 4 levelorderPrint--------------------")
t.levelorderPrint(a)   # a is the root
print()
```

```
Levels: 4
      -
     / \
    /   \
   /     \
  /       \
  *        +
 / \      / \
/   \    /   \
+    4   -    2
/ \     / \
3 1     9 5

-------------------Testing task 1 preorder--------------------
- * + 3 1 4 + - 9 5 2
-------------------Testing task 2 inorder--------------------
3 + 1 * 4 - 9 - 5 + 2
-------------------Testing task 3 postorder--------------------
3 1 + 4 * 9 5 - 2 + -
-------------------Testing task 4 levelorderPrint--------------------
- * + + 4 - 2 3 1 9 5
```

```
''' The following code will construct this tree:
                    1
                / \
                /     \
            /         \
            /             \
            2             3
        / \
        /     \
        4     5
    / \
    6  7
'''
t2 = Tree()
a2 = t2.add_root(1)
b2 = t2.add_left(a2, 2)
c2 = t2.add_right(a2, 3)
d2 = t2.add_left(b2, 4)
e2 = t2.add_right(b2, 5)
```

```
f2 = t2.add_left(d2, 6)
g2 = t2.add_right(d2, 7)

print("-------------------Testing task 5 height--------------------")
print("Height of Tree 1: Expected: 3;     Your answer:", t.height())
print("Height of Tree 2: Expected: 3;     Your answer:", t2.height())
print("-------------------Testing task 6 depth--------------------")
print("Depth of '*' in Tree 1: Expected: 1;     Your answer:", t.depth(b))
print("Depth of root in Tree 1: Expected: 0;     Your answer:", t.depth(a))
print("Depth of leaf in Tree 1: Expected: 3;     Your answer:", t.depth(k))
print("-------------------Testing task 7 return_max--------------------")
print("Max value within Tree 2: Expected: 7;     Your answer:", t2.return_max())
```

```
-------------------Testing task 5 height--------------------
Height of Tree 1: Expected: 3;   Your answer: 3
Height of Tree 2: Expected: 3;   Your answer: 3
-------------------Testing task 6 depth--------------------
Depth of '*' in Tree 1: Expected: 1;    Your answer: 1
Depth of root in Tree 1: Expected: 0;    Your answer: 0
Depth of leaf in Tree 1: Expected: 3;    Your answer: 3
-------------------Testing task 7 return_max--------------------
Max value within Tree 2: Expected: 7;    Your answer: 7
```

```
print("-------------------Testing task 8 flip_node--------------------")
print("Tree 2 before flip_node:")
pretty_print(t2)
t2.flip_node(t2._root)
print("Tree 2 after flip_node:")
pretty_print(t2)
print("-------------------Testing task 9 flip_tree--------------------")
print("Tree 1 before flip_tree:")
pretty_print(t)
t.flip_tree(t._root)
print("Tree 1 after flip_tree:")
pretty_print(t)
```

```
-------------------Testing task 8 flip_node--------------------
Tree 2 before flip_node:
Levels: 4
       1
      / \
     /   \
    /     \
   /       \
   2       3
  / \
 /   \
 4   5
/ \
6 7

Tree 2 after flip_node:
Levels: 4
       1
      / \
     /   \
    /     \
   /       \
   3       2
          / \
         /   \
         4   5
        / \
        6 7

-------------------Testing task 9 flip_tree--------------------
Tree 1 before flip_tree:
Levels: 4
       -
      / \
     /   \
    /     \
   /       \
   *       +
  / \     / \
```

```
  /   \     /   \
 /     \   /     \
 +      4  -      2
/ \       / \
3 1       9 5

Tree 1 after flip_tree:
Levels: 4
       -
      / \
     /   \
    /     \
   /       \
   +        *
  / \      / \
 /   \    /   \
 2    -   4    +
     / \      / \
     5 9      1 3
```

✓ 0 秒 完成时间：11:11