Mahusay, Divine Mars
Molina, Joshua Ali S.
DSALGO1 IDB2

Team Project #2 Part A

main.py × | Extension: Code Screenshot

main.py > display_traversals

```python
# Mahusay, Divine Mars
# Molina, Joshua Ali S.
# DSALGO1 Project 2
# IDB2
from LinkedBinaryTree import LinkedBinaryTree as Tree


# Function to perform inorder traversal
def inorder(tree, position):
    if position is not None:
        if tree.left(position) is not None:
            inorder(tree, tree.left(position))
        print(position.element(), end=" ")
        if tree.right(position) is not None:
            inorder(tree, tree.right(position))


# Function to display traversals
def display_traversals(tree):
    print("Preorder traversal:", end=" ")
    for pos in tree.preorder():
        print(pos.element(), end=" ")
    print()

    print("Postord  (function) element: Any
    for pos in tre
        print(pos.element(), end=" ")
    print()

    print("Inorder traversal:", end=" ")
    inorder(tree, tree.root())
    print()


# Function to print the tree structure
def print_tree(tree, position, level=0):
    if position is not None:
        print_tree(tree, tree.right(position), level + 1)
        print("    " * level + str(position.element()))
        print_tree(tree, tree.left(position), level + 1)
```

Equation 1

```python
print("Team project #2 Part A")
# Equation 1: (3 * 5) - ((4 * 5) + (6 - 7))
print("Equation 1:")
tree = Tree()
root = tree._add_root("-")
left_mul = tree._add_left(root, "*")
tree._add_left(left_mul, 3)
tree._add_right(left_mul, 5)

right_add = tree._add_right(root, "+")
right_mul = tree._add_left(right_add, "*")
tree._add_left(right_mul, 4)
tree._add_right(right_mul, 5)

right_sub = tree._add_right(right_add, "-")
tree._add_left(right_sub, 6)
tree._add_right(right_sub, 7)

display_traversals(tree)
print("Tree structure:")
print_tree(tree, tree.root())
print()
```

```
Team project #2 Part A
Equation 1:
Preorder traversal: - * 3 5 + * 4 5 - 6 7
Postorder traversal: 3 5 * 4 5 * 6 7 - + -
Inorder traversal: 3 * 5 - 4 * 5 + 6 - 7
Tree structure:
            7
        -
            6
    +
            5
        *
            4
-
        5
    *
        3
```

Equation 2

```python
# Equation 2: ((a + b) * c) - (d - e)
print("Equation 2:")
tree = Tree()
root = tree._add_root("-")
left_mul = tree._add_left(root, "*")

left_add = tree._add_left(left_mul, "+")
tree._add_left(left_add, "a")
tree._add_right(left_add, "b")

tree._add_right(left_mul, "c")

right_sub = tree._add_right(root, "-")
tree._add_left(right_sub, "d")
tree._add_right(right_sub, "e")

display_traversals(tree)
print("Tree structure:")
print_tree(tree, tree.root())
print()

```

```
Equation 2:
Preorder traversal: - * + a b c - d e
Postorder traversal: a b + c * d e - -
Inorder traversal: a + b * c - d - e
Tree structure:
        e
    -
        d
-
        c
    *
            b
        +
            a
```

Equation 3

```python
# Equation 3: ((a ^ b) + (c + d)) + ((e * f) / (g + h))
print("Equation 3:")
tree = Tree()
root = tree._add_root("+")
left_add = tree._add_left(root, "+")
left_exp = tree._add_left(left_add, "^")
tree._add_left(left_exp, "a")
tree._add_right(left_exp, "b")

right_add = tree._add_right(left_add, "+")
tree._add_left(right_add, "c")
tree._add_right(right_add, "d")

right_div = tree._add_right(root, "/")
right_mul = tree._add_left(right_div, "*")
tree._add_left(right_mul, "e")
tree._add_right(right_mul, "f")

right_add2 = tree._add_right(right_div, "+")
tree._add_left(right_add2, "g")
tree._add_right(right_add2, "h")

display_traversals(tree)
print("Tree structure:")
print_tree(tree, tree.root())
print()
```

```
Equation 3:
Preorder traversal: + + ^ a b + c d / * e f + g h
Postorder traversal: a b ^ c d + + e f * g h + / +
Inorder traversal: a ^ b + c + d + e * f / g + h
Tree structure:
            h
        +
            g
    /
            f
        *
            e
+
            d
        +
            c
    +
            b
        ^
            a
```

Equation 4

```python
# Equation 4: (a + b) / (c * (d - (e ^ f)))
print("Equation 4:")
tree = Tree()
root = tree._add_root("/")
left_add = tree._add_left(root, "+")
tree._add_left(left_add, "a")
tree._add_right(left_add, "b")

right_mul = tree._add_right(root, "*")
tree._add_left(right_mul, "c")

right_sub = tree._add_right(right_mul, "-")
tree._add_left(right_sub, "d")

right_exp = tree._add_right(right_sub, "^")
tree._add_left(right_exp, "e")
tree._add_right(right_exp, "f")

display_traversals(tree)
print("Tree structure:")
print_tree(tree, tree.root())
print()
```

```
Equation 4:
Preorder traversal: / + a b * c - d ^ e f
Postorder traversal: a b + c d e f ^ - * /
Inorder traversal: a + b / c * d - e ^ f
Tree structure:
                f
            ^
                e
        -
            d
    *
        c
/
        b
    +
        a
```

Equation 5

```python
# Equation 5: ((a - b) + c) * ((d + e) * (f / g))
print("Equation 5:")
tree = Tree()
root = tree._add_root("*")

left_add = tree._add_left(root, "+")
left_sub = tree._add_left(left_add, "-")
tree._add_left(left_sub, "a")
tree._add_right(left_sub, "b")
tree._add_right(left_add, "c")

right_mul = tree._add_right(root, "*")
right_add = tree._add_left(right_mul, "+")
tree._add_left(right_add, "d")
tree._add_right(right_add, "e")

right_div = tree._add_right(right_mul, "/")
tree._add_left(right_div, "f")
tree._add_right(right_div, "g")

display_traversals(tree)
print("Tree structure:")
print_tree(tree, tree.root())
print()
```

```
Equation 5:
Preorder traversal: * + - a b c * + d e / f g
Postorder traversal: a b - c + d e + f g / * *
Inorder traversal: a - b + c * d + e * f / g
Tree structure:
            g
        /
            f
    *
            e
        +
            d
*
        c
    +
            b
        -
            a
```

Equation 6

```python
print("Equation 6:")
tree = Tree()
root = tree._add_root("*")

# Left subtree: (((5 + 2) * (2 - 1)) / ((2 + 9) + ((7 - 2) - 1)))
left_div = tree._add_left(root, "/")
left_mul = tree._add_left(left_div, "*")

left_add = tree._add_left(left_mul, "+")
tree._add_left(left_add, 5)
tree._add_right(left_add, 2)

left_sub = tree._add_right(left_mul, "-")
tree._add_left(left_sub, 2)
tree._add_right(left_sub, 1)

right_add1 = tree._add_right(left_div, "+")
right_add2 = tree._add_left(right_add1, "+")
tree._add_left(right_add2, 2)
tree._add_right(right_add2, 9)

right_sub = tree._add_right(right_add1, "-")
left_sub2 = tree._add_left(right_sub, "-")
tree._add_left(left_sub2, 7)
tree._add_right(left_sub2, 2)
tree._add_right(right_sub, 1)
```

```python
# Right subtree: *8
tree._add_right(root, 8)

display_traversals(tree)
print("Tree structure:")
print_tree(tree, tree.root())
print()
```

```
Equation 6:
Preorder traversal: * / * + 5 2 - 2 1 + + 2 9 - - 7 2 1 8
Postorder traversal: 5 2 + 2 1 - * 2 9 + 7 2 - 1 - + / 8 *
Inorder traversal: 5 + 2 * 2 - 1 / 2 + 9 + 7 - 2 - 1 * 8
Tree structure:
    8
*
                1
            -
                    2
                -
                    7
        +
                9
            +
                2
    /
                1
            -
                2
        *
                2
            +
                5
```

Team Project #2 Part B:

Matrix 1

```python
print("Matrix 1:")
tree = Tree()
root = tree._add_root("r")
a = tree._add_left(root, "a")

b = tree._add_left(a, "b")
tree._add_right(b, "d")

c = tree._add_right(a, "c")
f = tree._add_right(c, "f")

e = tree._add_left(c, "e")
g = tree._add_right(e, "g")
h = tree._add_right(g, "h")
display_traversals(tree)
print("Tree structure:")
print_tree(tree, tree.root())
print()
```

```
Team project #2 Part B
Matrix 1:
Preorder traversal: r a b d c e g h f
Postorder traversal: d b h g e f c a r
Inorder traversal: b d a e g h c f r
Tree structure:
r
            f
        c
                    h
                g
            e
    a
            d
        b
```

Matrix 2

```python
print("Matrix 2:")
tree = Tree()
root = tree._add_root("r")
a = tree._add_left(root, "a")
b = tree._add_right(root, "b")
tree._add_left(a, "c")
tree._add_right(a, "d")

e = tree._add_right(b, "e")
f = tree._add_right(e, "f")
g = tree._add_right(f, "g")
display_traversals(tree)
print("Tree structure:")
print_tree(tree, tree.root())
print()
```

```
Matrix 2:
Preorder traversal: r a c d b e f g
Postorder traversal: c d a g f e b r
Inorder traversal: c a d r b e f g
Tree structure:
                g
            f
        e
    b
r
        d
    a
        c
```

Matrix 3

```python
print("Matrix 3:")
tree = Tree()
root = tree._add_root("r")
a = tree._add_left(root, "a")
b = tree._add_right(root, "b")

c = tree._add_right(a, "c")

f = tree._add_left(c, "f")

d = tree._add_left(b, "d")
e = tree._add_right(b, "e")

display_traversals(tree)
print("Tree structure:")
print_tree(tree, tree.root())
print()

```

```
Matrix 3:
Preorder traversal: r a c f b d e
Postorder traversal: f c a d e b r
Inorder traversal: a f c r d b e
Tree structure:
        e
    b
        d
r
        c
            f
    a
```

Matrix 4

```python
print("Matrix 4:")
tree = Tree()
root = tree._add_root("r")
a = tree._add_left(root, "a")
b = tree._add_right(root, "b")

c = tree._add_left(a, "c")
d = tree._add_right(a, "d")
g= tree._add_left(c, "g")
h= tree._add_right(c, "h")

e = tree._add_left(b, "e")
f = tree._add_right(b, "f")
i = tree._add_left(e, "i")
display_traversals(tree)
print("Tree structure:")
print_tree(tree, tree.root())
print()

```

```
Matrix 4:
Preorder traversal: r a c g h d b e i f
Postorder traversal: g h c d a i e f b r
Inorder traversal: g c h a d r i e b f
Tree structure:
        f
    b
        e
            i
r
        d
    a
            h
        c
            g
```

Tree.py class:

```
class Tree:
    '''Abstrtact base class representing a tree structure'''
    #---------------------------nestedf Position Class ------------------------
    class Position:
        '''Abstraction representing the location of a single element'''
        def element(self):
            '''Return the element stored at this Position'''
            raise NotImplementedError('must be implemented by subclass')
        def __eq__(self, other):
            '''Return True if other is a Position representing the same location'''
            raise NotImplementedError('must be implemented by subclass')
        def __ne__(self, other):
            '''Return True if other does not represent the same location'''
            return not (self == other) #opposite of __eq__
    #---------------------------abstract methods----------------------------
    def root(self):
        '''Return the root Position of the tree (or None if tree is empty)'''
        raise NotImplementedError('must be implemented by subclass')
    def parent(self, p):
        '''Return the Position of p's parent (or None if p is root)'''
        raise NotImplementedError('must be implemented by subclass')
    def num_children(self, p):
        '''Return the number of children that Position p has.'''
        raise NotImplementedError('must be implemented by subclass')
    def children(self, p):
        '''Generate an iteration of Position representing p's children'''
        raise NotImplementedError('must be implemented by subclass')
    def __len__(self):
        '''Return the total number of elements in the tree'''
        raise NotImplementedError('must be implemented by subclass')
    #-------------------------Concrete methods------------------------------
    def is_root(self, p):
        '''Return True if Position p represents the root of the tree'''
        return self.root() == p
    def is_leaf(self, p):
        '''Return True if Position p does not have any children'''
        return self.num_children(p) == 0
    def is_empty(self):
        '''Return True if the tree is empty'''
        return len(self) == 0

    def depth(self, p):
        '''Return the number of levels separating Position p from the root.'''
        if self.is_root(p):
```

```python
            return 0
        else:
            return 1 + self.depth(self.parent(p))
    def _height1(self):
        '''Return the height of the tree'''
        return max(self.depth(p) for p in self.positions() if self.is_leaf(p))
    def _height2(self, p):
        '''Return the height of the subtree rooted at Position p'''
        if self.is_leaf(p):
            return 0
        else:
            return 1 + max(self._height2(c) for c in self.children(p))
    def height(self, p=None):
        '''Return the height of the subtree rooted at Position p.'''
        '''If p is None, return the height of the entire tree.'''
        if p is None:
            p = self.root()
        return self._height2(p) #start_height2 recursion
    def __iter__(self):
        '''Generate an iteration of the tree's elements'''
        for p in self.positions(): #use same order as positions
            yield p.element()#but yield each element
    def preorder(self):
        '''Generate a preorder iteration of positions in the tree.'''
        if not self.is_empty():
            for p in self._subtree_preorder(self.root()):#start recursion
                yield p
    def _subtree_preorder(self, p):
        '''Generate a preorder iteration of positions in subtree rooted at p.'''
        yield p #visit p before its subtrees
        for c in self.children(p): #visit each child
            for other in self._subtree_preorder(c): #do preorder of c
                yield other #yield all other preorder trees

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

    def postorder(self):
        '''Generate a postorder iteration of positions in the tree.'''
        if not self.is_empty():
            for p in self._subtree_postorder(self.root()):#start recursion
                yield p
```

```python
    def _subtree_postorder(self, p):
        '''Generate a postorder iteration of positions in subtree rooted at p.'''
        for c in self.children(p): #visit each child
            for other in self._subtree_postorder(c): #do postorder of c
                yield other #yield each to our caller
        yield p #visit p after its subtrees

    def positions2(self):
        '''Generate an iteration of the tree;s positions'''
        return self.postorder()
```

LinkedBinaryTree.py

```python
from BinaryTree import BinaryTree
class LinkedBinaryTree(BinaryTree):
    '''Linked representation of a binary tree structure.'''
    class Node: #Lightweight, non public class for storing a node
        __slots__ = '_element', '_parent', '_left', '_right'
        def __init__(self, element, parent=None, left=None, right=None):
            self._element = element
            self._parent = parent
            self._left = left
            self._right = right

    class Position(BinaryTree.Position):
        '''An abstraction representing the location of a single element.'''

        def __init__(self, container, node):
            '''Constructor should not be invoked by the user.'''
            self._container = container
            self._node = node

        def element(self):
            '''Return the element stored at this Position'''
            return self._node._element

        def __eq__(self, other):
            '''Return True if other is a Position representing the same location.'''
            return type(other) is type(self) and other._node is self._node

    def _validate(self, p):
        '''Return position's node or raise appropriate error if invalid'''
        if not isinstance(p, self.Position):
            raise TypeError('p must be proper Position type')
        if p._container is not self:
            raise ValueError('p does not belong to this container')
```

```
        if p._node._parent is p._node:#convention for deprecated nodes
            raise ValueError('p is no longer valid')
        return p._node

    def _make_position(self, node):
        '''Return Position instance for given node (or None if sentinel).'''
        return self.Position(self, node) if node is not None else None
    #-----binary tree constructor -----
    def __init__(self):
        '''Create an 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 root(self):
        '''Return the root Position of the tree(or None if tree is empty)'''
        return self._make_position(self._root)

    def parent(self, p):
        '''Return the Position of p's parent(or None if p is root)'''
        node = self._validate(p)
        return self._make_position(node._parent)

    def left(self, p):
        '''Return the Position of p's left child(or None if p has no left child)'''
        node = self._validate(p)
        return self._make_position(node._left)

    def right(self, p):
        '''Return the Position of p's right child(or None if p has no right child)'''
        node = self._validate(p)
        return self._make_position(node._right)

    def num_children(self, p):
        '''Return the number of children of Position p.'''
        node = self._validate(p)
        count = 0
        if node._left is not None:#left child exists
            count += 1
        if node._right is not None:#right child exists
```

```python
            count += 1
        return count

    def _add_root(self, e):
        '''Place element e at the root of an empty tree and return new Position.'''
        '''Raise ValueError if tree nonempty.'''
        if self._root is not None:
            raise ValueError('Root exists')
        self._size = 1
        self._root = self.Node(e)
        return self._make_position(self._root)

    def _add_left(self, p, e):
        '''Create a new left child for Position p, storing element e.'''

        '''Return the position of new node.
        Raise ValueError if Position p is invalid or p already has a left child'''
        node = self._validate(p)
        if node._left is not None:
            raise ValueError('Left child exists')
        self._size += 1
        node._left = self.Node(e, node) #node is its parent
        return self._make_position(node._left)

    def _add_right(self, p, e):
        '''Create a new right child for Position p, storing element e.'''

        '''Return the Position of new node
        Raise ValueError if Position p is invalid or p already has a right child'''
        node = self._validate(p)
        if node._right is not None:
            raise ValueError('Right child exists')
        self._size += 1
        node._right = self.Node(e, node) #node is its parent
        return self._make_position(node._right)

    def _replace(self, p, e):
        '''Replace the element at position p with e, and return old element.'''
        node = self._validate(p)
        old = node._element
        node._element = e
        return old

    def _delete(self, p):
```

```python
        '''Delete the node at Position p, and replace it with its child, if any.'''
        '''Return the element that had been stored at Position p.'''
        '''Raise ValueError if Position is invalid or p has two children'''
        node = self._validate(p)
        if self.num_children(p) == 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
        node._parent = node # convention for deprecated node
        return node._element

    def _attach(self, p, t1, t2):
        '''Attach tree t1 and t2 as left and right subtrees of external p.'''
        node = self._validate(p)
        if not self.is_leaf(p): 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
            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
            t2._size = 0
```

BinaryTree.py

```
from Tree import Tree
class BinaryTree(Tree):
    '''Abstract base class representing a binary tree structure.'''
    #----------------------------nested Position class----------------------------
    def left(self, p):
        '''Return a position representing p's left child.
        Return None if p does not have a left child.'''
        raise NotImplementedError('must be implemented by subclass')
    def right(self, p):
        '''Return a Position representing p's right child.
        Return None if p does not have a right child.'''
        raise NotImplementedError('must be implemented by subclass')
    #----------------------------concrete methods implemented in this
class----------------------------
    def sibling(self, p):
        '''Return a Position representing p's sibling(or None if no sibling)'''
        parent = self.parent(p)
        if parent is None:#p must be the root
            return None#the root has no sibling
        else:
            if p == self.left(parent):
                return self.right(parent)#possibly None
            else:
                return self.left(parent)#possible Nont
    def children(self, p):
        '''Generate an iteration of Positions representing p's children.'''
        if self.left(p) is not None:
            yield self.left(p)
        if self.right(p) is not None:
            yield self.right(p)

    '''In order traversal'''
    def inorder(self):
        '''Generate an inorder iteration of positions in the tree.'''
        if not self.is_empty():
            for p in self._subtree_inorder(self.root()):
                yield p

    def _subtree_inorder(self, p):
        '''Generate an inrder iteration of positions in subtree rooted at p.'''
        if self.left(p) is not None: #if left child exists, traverse its subtree
            for other in self._subtree_inorder(self.left(p)):
                yield other
```

```
        yield p #visit p between its subtrees
        if self.right(p) is not None: #if right child exists, traverse its subtree
            for other in self._subtree_inorder(self.right(p)):
                yield other
```