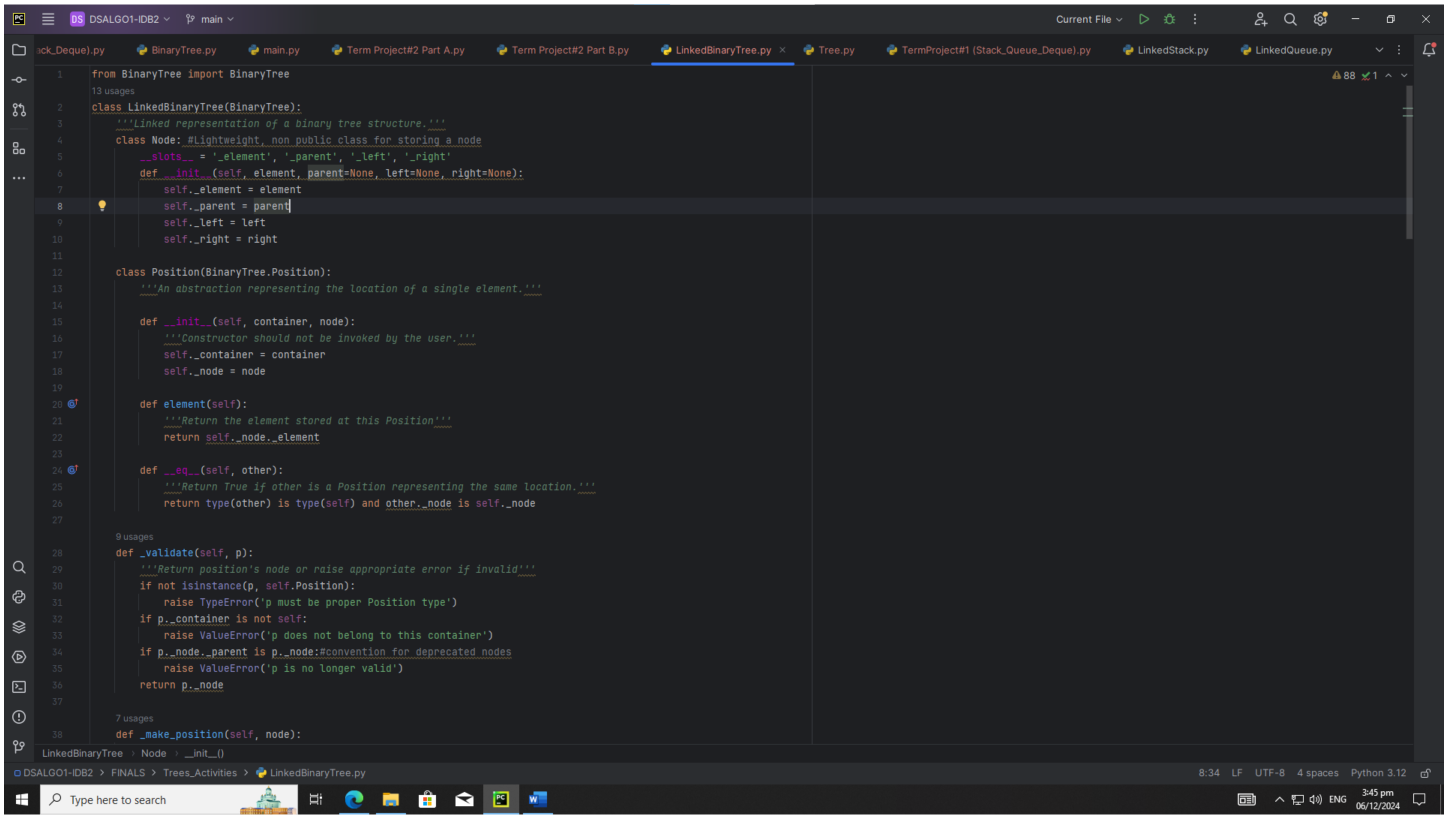
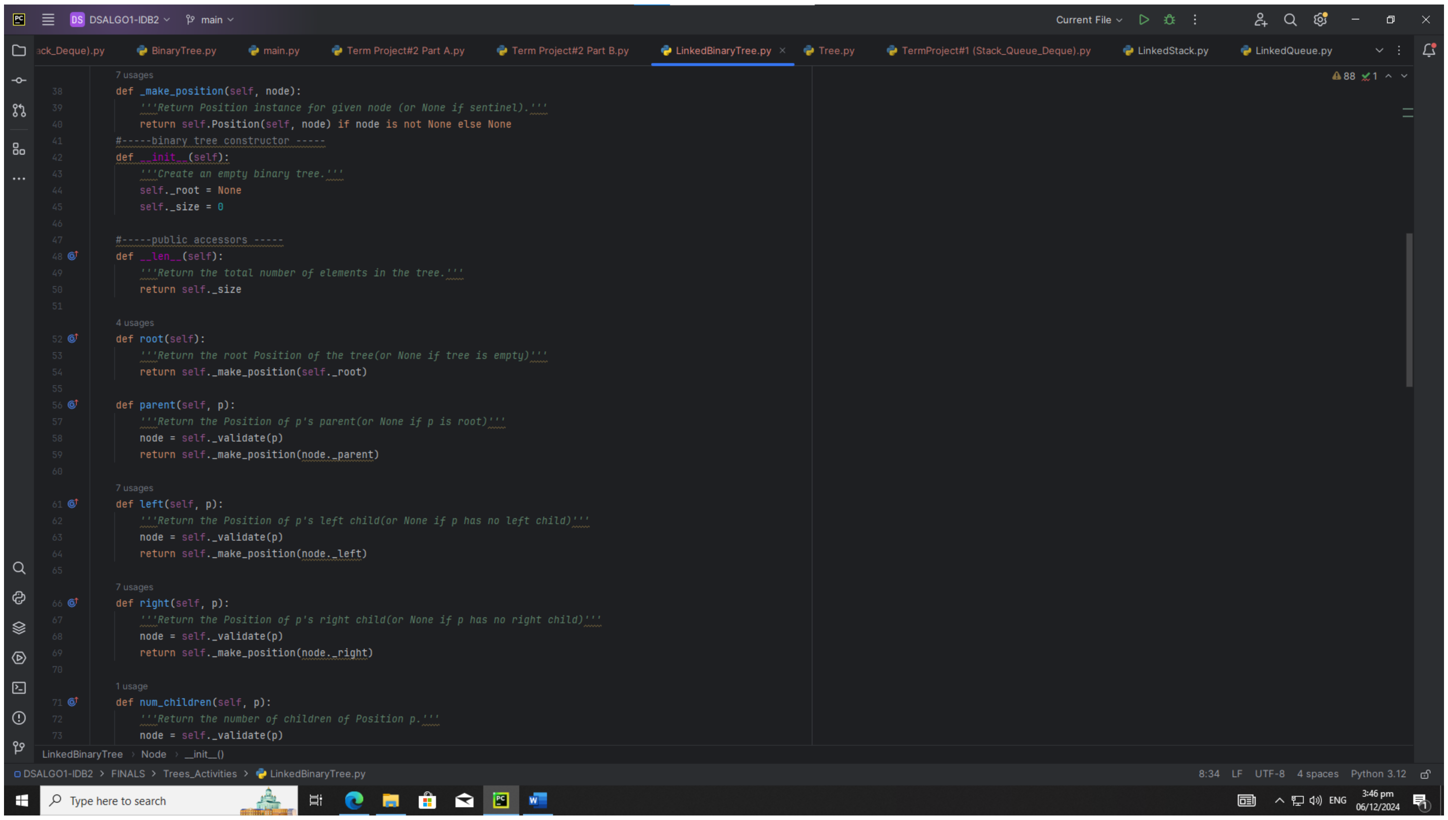
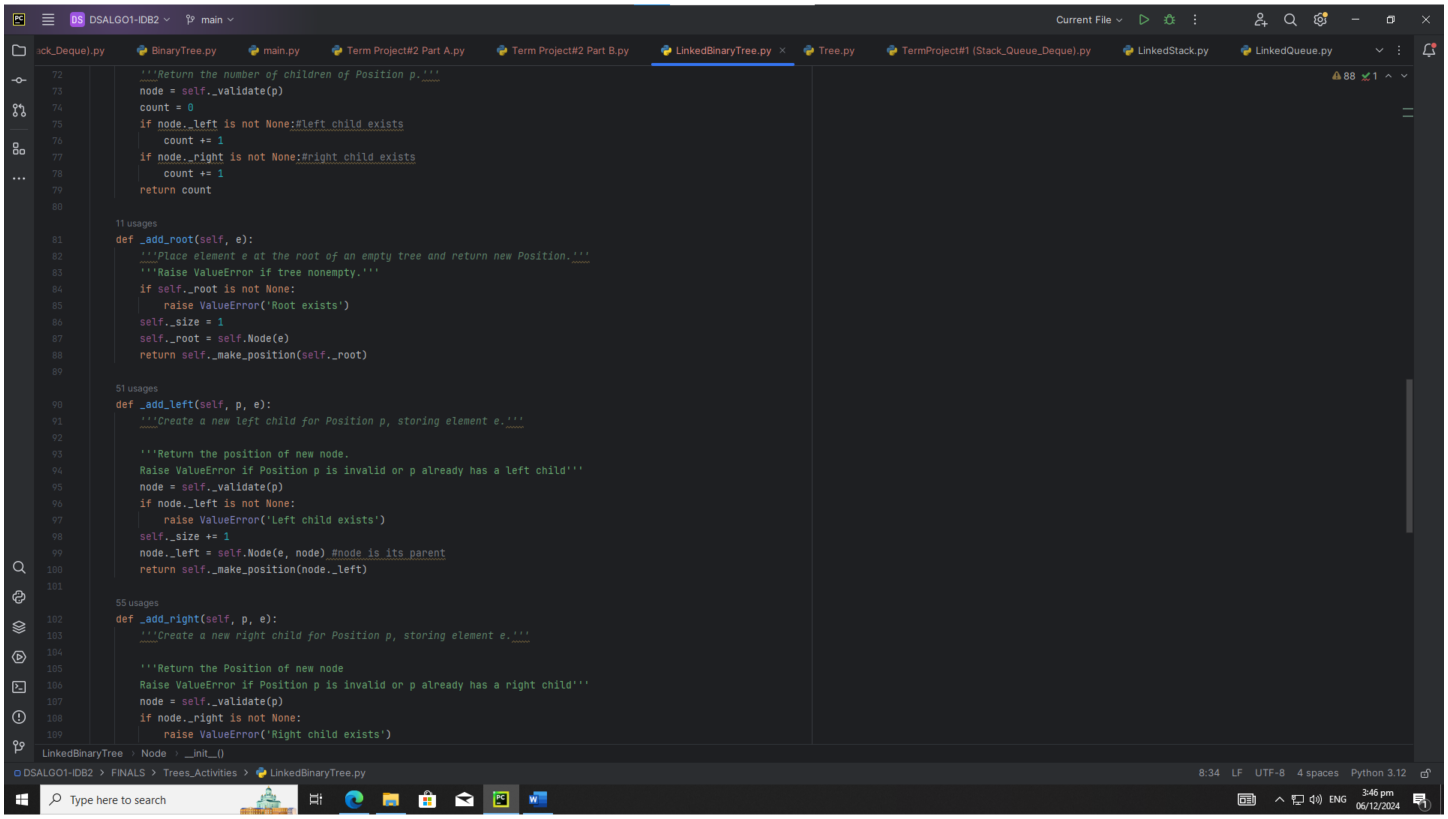
SOURCE CODE: PART A



```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            raise ValueError('p is no longer valid')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):
```



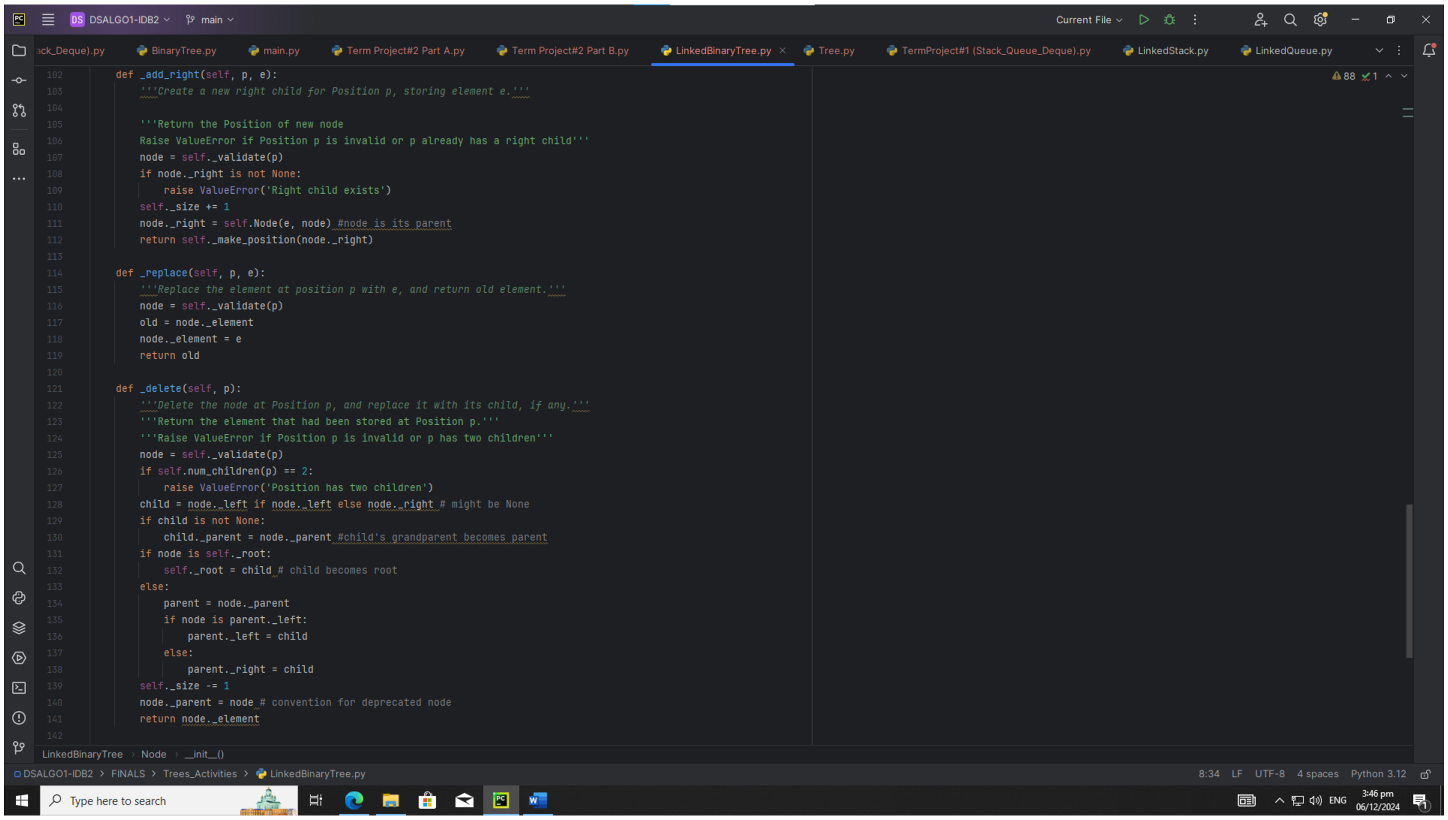
```python
    def _make_position(self, node):
        '''Return Position instance for given node (or None if sentinel).'''
        return self.Position(self, node) if node is not None else None
    #-----binary tree constructor -----
    def __init__(self):
        '''Create an empty binary tree.'''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



```
        '''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
            count += 1
        return count

    11 usages
    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)

    51 usages
    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)

    55 usages
    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')
```

LinkedBinaryTree › Node › __init__()

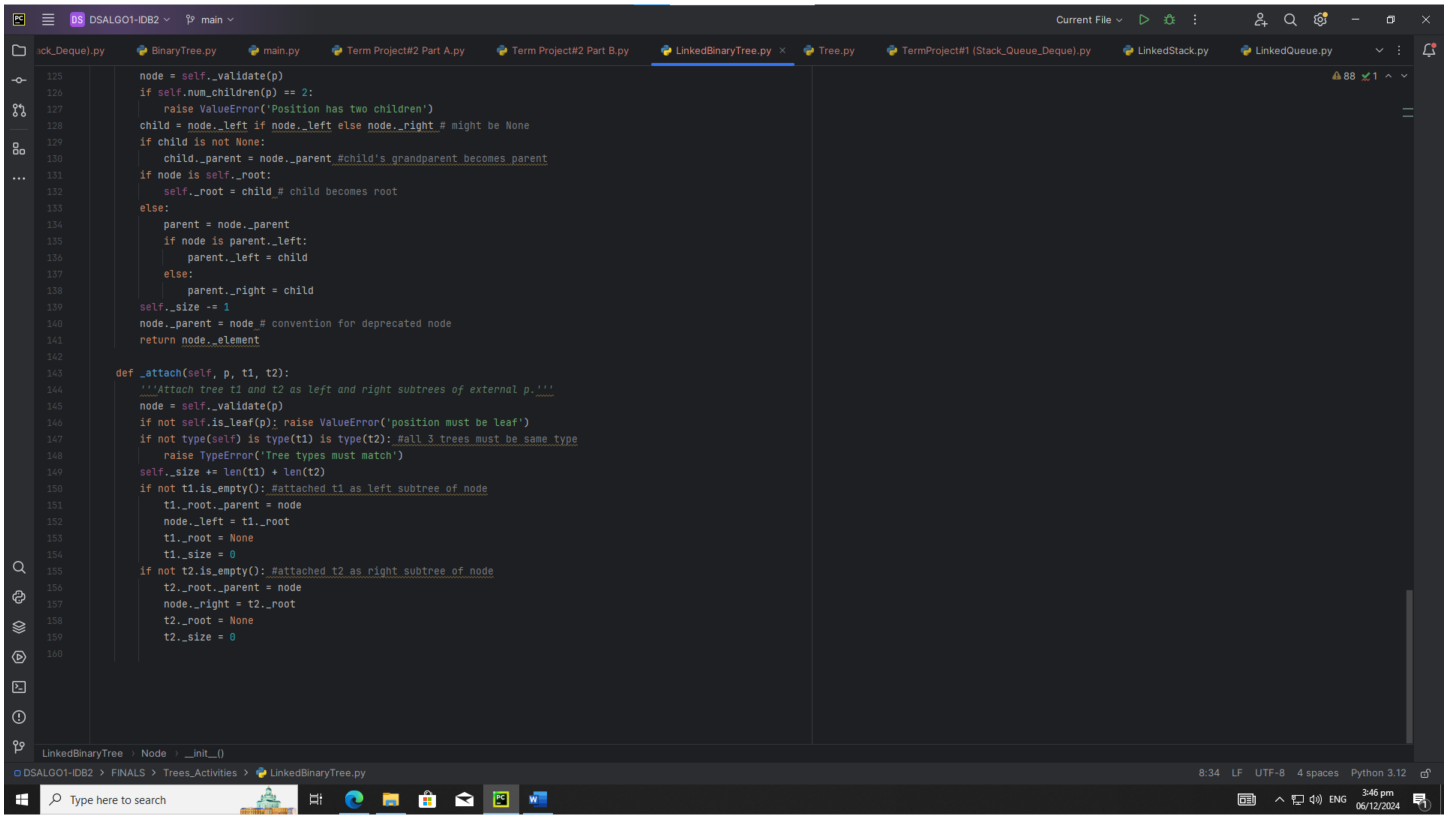
DSALGO1-IDB2 > FINALS > Trees_Activities > LinkedBinaryTree.py   8:34 LF UTF-8 4 spaces Python 3.12



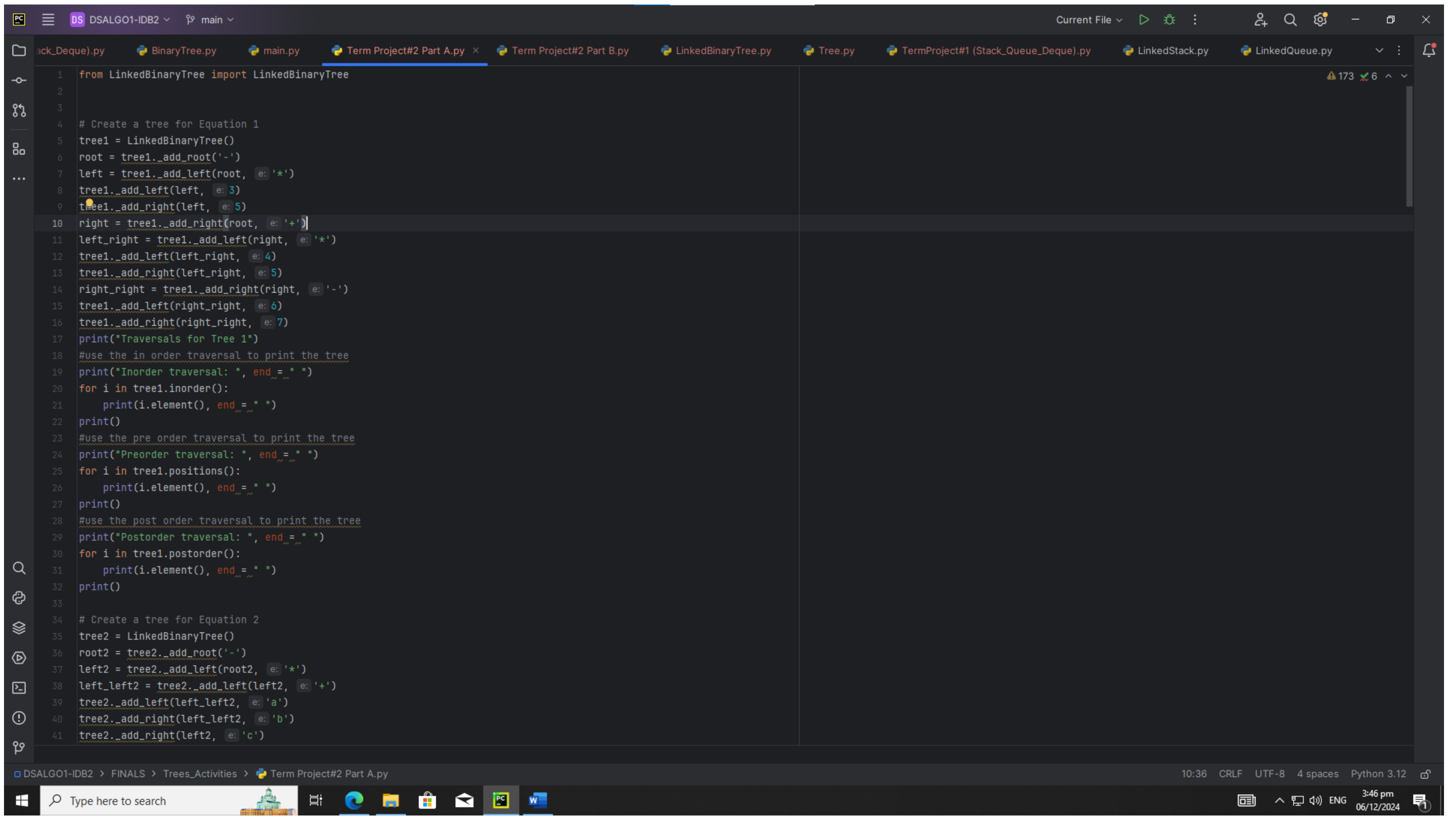
```
    def _add_right(self, p, e):
        '''Create a new right child for Position p, storing element e.'''

        '''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):
        '''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 p 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
```

LinkedBinaryTree › Node › __init__()

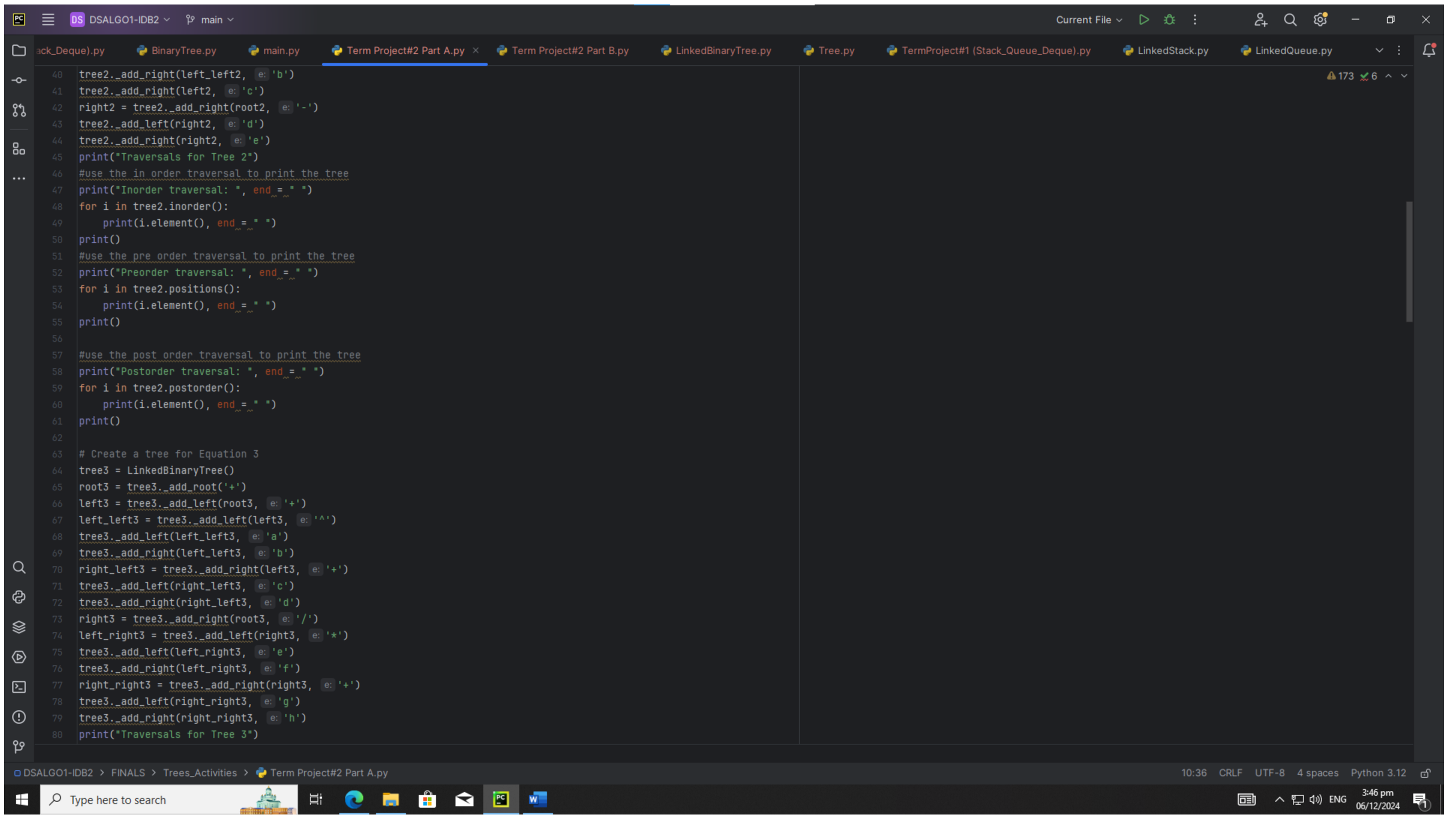
DSALGO1-IDB2 > FINALS > Trees_Activities > LinkedBinaryTree.py   8:34 LF UTF-8 4 spaces Python 3.12



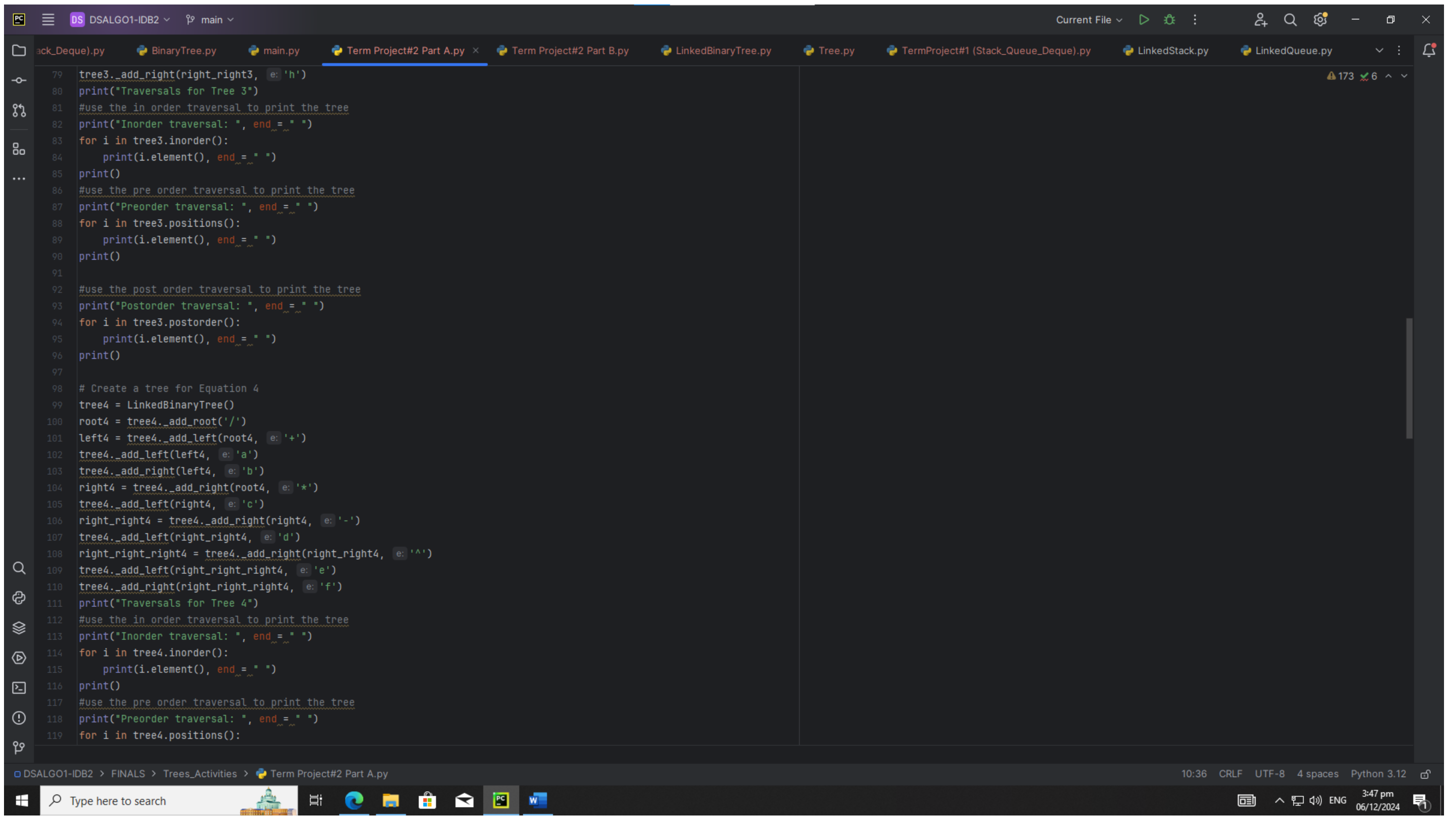
```python
        node = self._validate(p)
        if self.num_children(p) == 2:
            raise ValueError('Position has two children')
        child = node._left if node._left else node._right # might be None
        if child is not None:
            child._parent = node._parent #child's grandparent 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

```



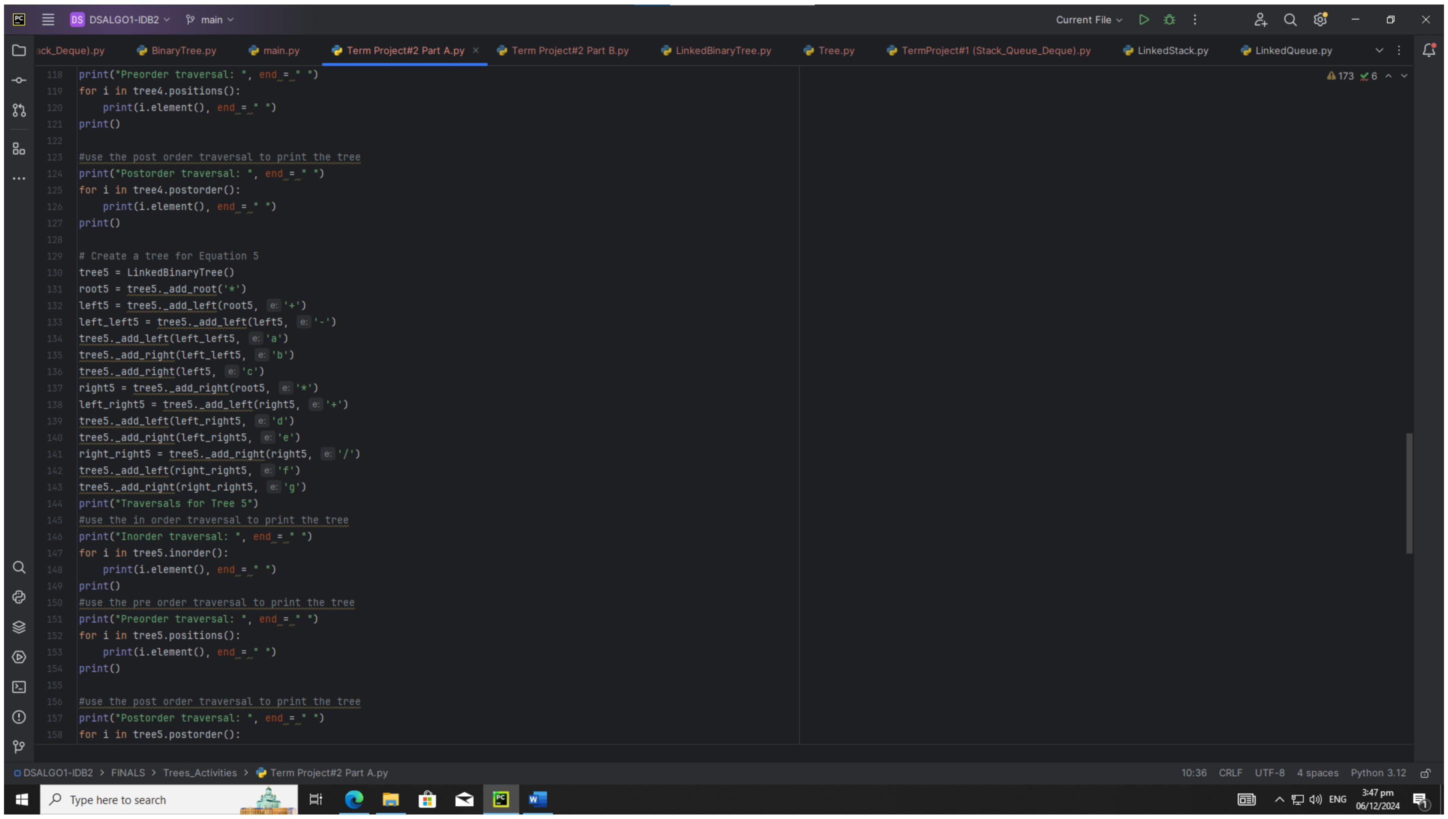
```python
from LinkedBinaryTree import LinkedBinaryTree


# Create a tree for Equation 1
tree1 = LinkedBinaryTree()
root = tree1._add_root('-')
left = tree1._add_left(root, '*')
tree1._add_left(left, 3)
tree1._add_right(left, 5)
right = tree1._add_right(root, '+')
left_right = tree1._add_left(right, '*')
tree1._add_left(left_right, 4)
tree1._add_right(left_right, 5)
right_right = tree1._add_right(right, '-')
tree1._add_left(right_right, 6)
tree1._add_right(right_right, 7)
print("Traversals for Tree 1")
#use the in order traversal to print the tree
print("Inorder traversal: ", end = " ")
for i in tree1.inorder():
    print(i.element(), end = " ")
print()
#use the pre order traversal to print the tree
print("Preorder traversal: ", end = " ")
for i in tree1.positions():
    print(i.element(), end = " ")
print()
#use the post order traversal to print the tree
print("Postorder traversal: ", end = " ")
for i in tree1.postorder():
    print(i.element(), end = " ")
print()

# Create a tree for Equation 2
tree2 = LinkedBinaryTree()
root2 = tree2._add_root('-')
left2 = tree2._add_left(root2, '*')
left_left2 = tree2._add_left(left2, '+')
tree2._add_left(left_left2, 'a')
tree2._add_right(left_left2, 'b')
tree2._add_right(left2, 'c')
```



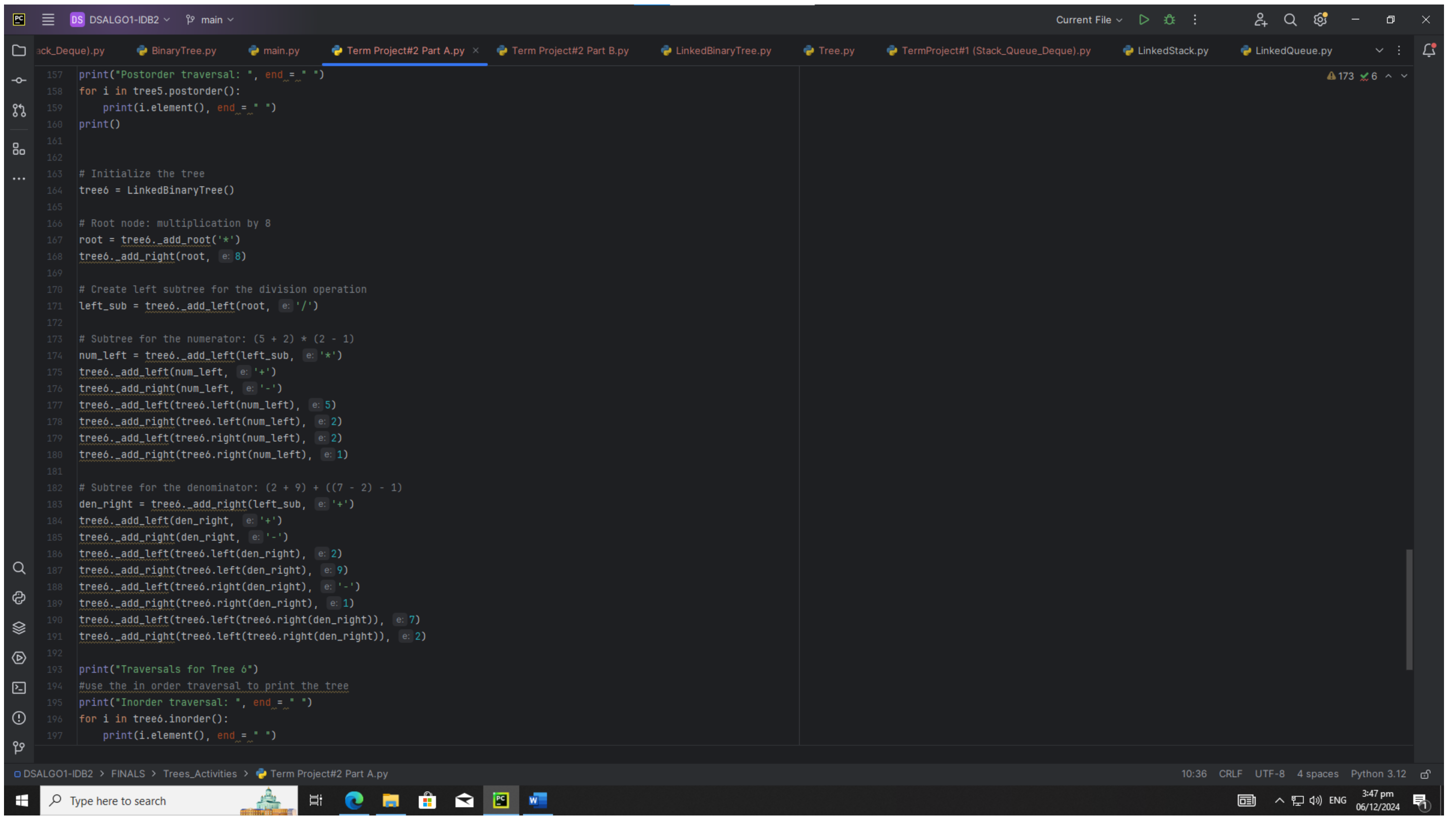
```python
tree2._add_right(left_left2, 'b')
tree2._add_right(left2, 'c')
right2 = tree2._add_right(root2, '-')
tree2._add_left(right2, 'd')
tree2._add_right(right2, 'e')
print("Traversals for Tree 2")
#use the in order traversal to print the tree
print("Inorder traversal: ", end = " ")
for i in tree2.inorder():
    print(i.element(), end = " ")
print()
#use the pre order traversal to print the tree
print("Preorder traversal: ", end = " ")
for i in tree2.positions():
    print(i.element(), end = " ")
print()

#use the post order traversal to print the tree
print("Postorder traversal: ", end = " ")
for i in tree2.postorder():
    print(i.element(), end = " ")
print()

# Create a tree for Equation 3
tree3 = LinkedBinaryTree()
root3 = tree3._add_root('+')
left3 = tree3._add_left(root3, '+')
left_left3 = tree3._add_left(left3, '^')
tree3._add_left(left_left3, 'a')
tree3._add_right(left_left3, 'b')
right_left3 = tree3._add_right(left3, '+')
tree3._add_left(right_left3, 'c')
tree3._add_right(right_left3, 'd')
right3 = tree3._add_right(root3, '/')
left_right3 = tree3._add_left(right3, '*')
tree3._add_left(left_right3, 'e')
tree3._add_right(left_right3, 'f')
right_right3 = tree3._add_right(right3, '+')
tree3._add_left(right_right3, 'g')
tree3._add_right(right_right3, 'h')
print("Traversals for Tree 3")
```



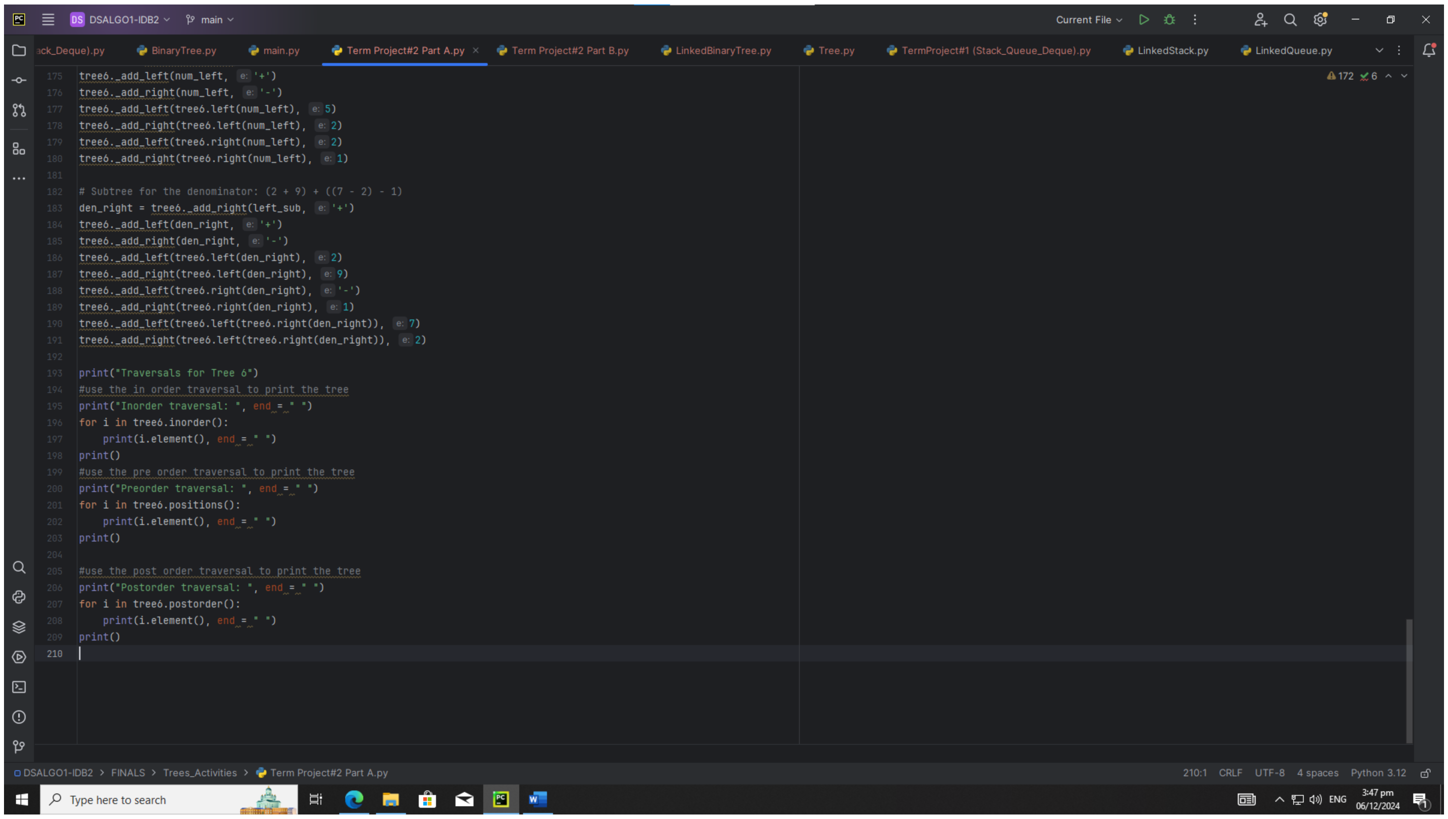
```python
tree3._add_right(right_right3, 'h')
print("Traversals for Tree 3")
#use the in order traversal to print the tree
print("Inorder traversal: ", end = " ")
for i in tree3.inorder():
    print(i.element(), end = " ")
print()
#use the pre order traversal to print the tree
print("Preorder traversal: ", end = " ")
for i in tree3.positions():
    print(i.element(), end = " ")
print()

#use the post order traversal to print the tree
print("Postorder traversal: ", end = " ")
for i in tree3.postorder():
    print(i.element(), end = " ")
print()

# Create a tree for Equation 4
tree4 = LinkedBinaryTree()
root4 = tree4._add_root('/')
left4 = tree4._add_left(root4, '+')
tree4._add_left(left4, 'a')
tree4._add_right(left4, 'b')
right4 = tree4._add_right(root4, '*')
tree4._add_left(right4, 'c')
right_right4 = tree4._add_right(right4, '-')
tree4._add_left(right_right4, 'd')
right_right_right4 = tree4._add_right(right_right4, '^')
tree4._add_left(right_right_right4, 'e')
tree4._add_right(right_right_right4, 'f')
print("Traversals for Tree 4")
#use the in order traversal to print the tree
print("Inorder traversal: ", end = " ")
for i in tree4.inorder():
    print(i.element(), end = " ")
print()
#use the pre order traversal to print the tree
print("Preorder traversal: ", end = " ")
for i in tree4.positions():
```



```python
print("Preorder traversal: ", end = " ")
for i in tree4.positions():
    print(i.element(), end = " ")
print()

#use the post order traversal to print the tree
print("Postorder traversal: ", end = " ")
for i in tree4.postorder():
    print(i.element(), end = " ")
print()

# Create a tree for Equation 5
tree5 = LinkedBinaryTree()
root5 = tree5._add_root('*')
left5 = tree5._add_left(root5, '+')
left_left5 = tree5._add_left(left5, '-')
tree5._add_left(left_left5, 'a')
tree5._add_right(left_left5, 'b')
tree5._add_right(left5, 'c')
right5 = tree5._add_right(root5, '*')
left_right5 = tree5._add_left(right5, '+')
tree5._add_left(left_right5, 'd')
tree5._add_right(left_right5, 'e')
right_right5 = tree5._add_right(right5, '/')
tree5._add_left(right_right5, 'f')
tree5._add_right(right_right5, 'g')
print("Traversals for Tree 5")
#use the in order traversal to print the tree
print("Inorder traversal: ", end = " ")
for i in tree5.inorder():
    print(i.element(), end = " ")
print()
#use the pre order traversal to print the tree
print("Preorder traversal: ", end = " ")
for i in tree5.positions():
    print(i.element(), end = " ")
print()

#use the post order traversal to print the tree
print("Postorder traversal: ", end = " ")
for i in tree5.postorder():
```

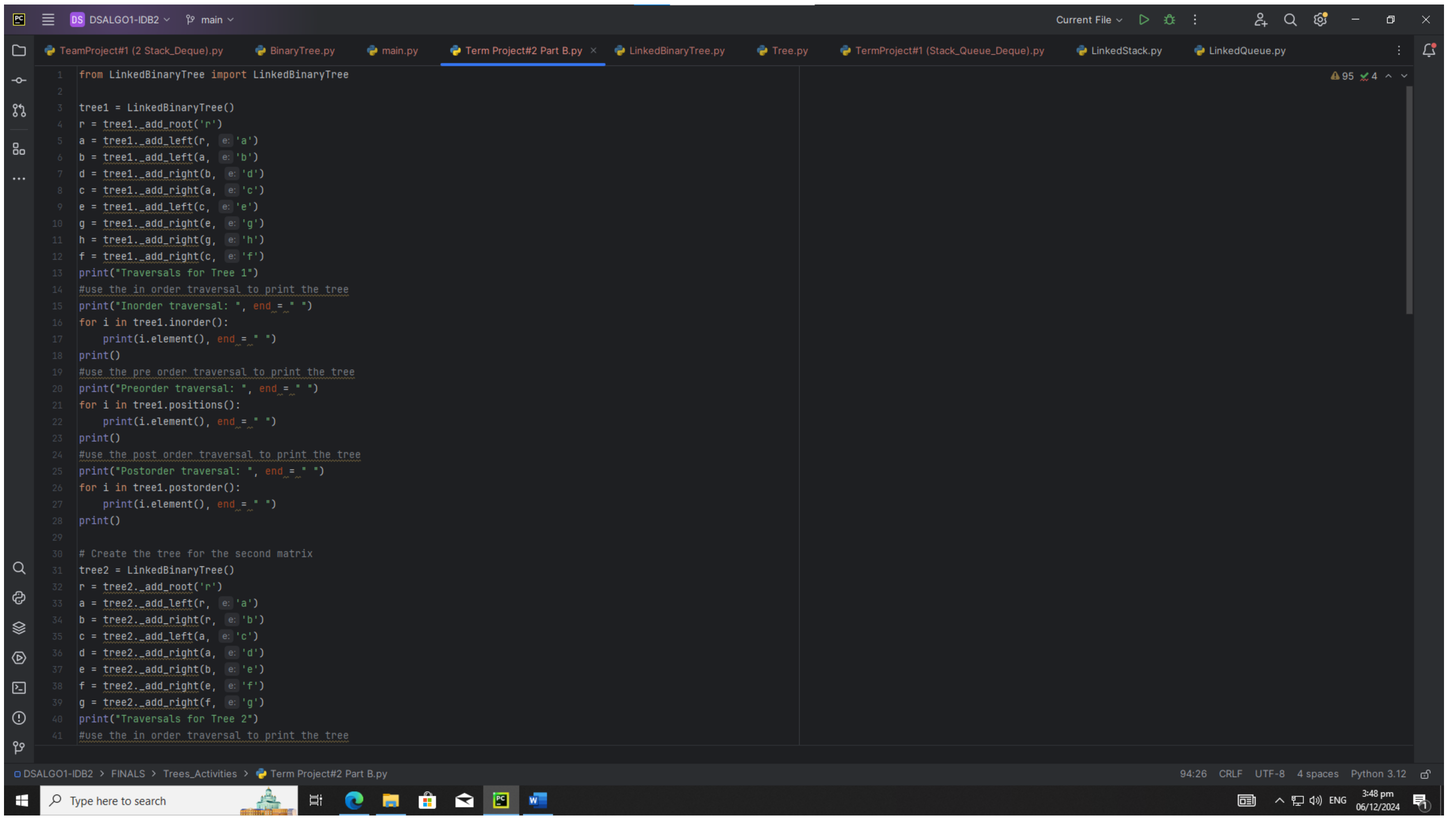


```python
print("Postorder traversal: ", end = " ")
for i in tree5.postorder():
    print(i.element(), end = " ")
print()


# Initialize the tree
tree6 = LinkedBinaryTree()

# Root node: multiplication by 8
root = tree6._add_root('*')
tree6._add_right(root, 8)

# Create left subtree for the division operation
left_sub = tree6._add_left(root, '/')

# Subtree for the numerator: (5 + 2) * (2 - 1)
num_left = tree6._add_left(left_sub, '*')
tree6._add_left(num_left, '+')
tree6._add_right(num_left, '-')
tree6._add_left(tree6.left(num_left), 5)
tree6._add_right(tree6.left(num_left), 2)
tree6._add_left(tree6.right(num_left), 2)
tree6._add_right(tree6.right(num_left), 1)

# Subtree for the denominator: (2 + 9) + ((7 - 2) - 1)
den_right = tree6._add_right(left_sub, '+')
tree6._add_left(den_right, '+')
tree6._add_right(den_right, '-')
tree6._add_left(tree6.left(den_right), 2)
tree6._add_right(tree6.left(den_right), 9)
tree6._add_left(tree6.right(den_right), '-')
tree6._add_right(tree6.right(den_right), 1)
tree6._add_left(tree6.left(tree6.right(den_right)), 7)
tree6._add_right(tree6.left(tree6.right(den_right)), 2)

print("Traversals for Tree 6")
#use the in order traversal to print the tree
print("Inorder traversal: ", end = " ")
for i in tree6.inorder():
    print(i.element(), end = " ")
```

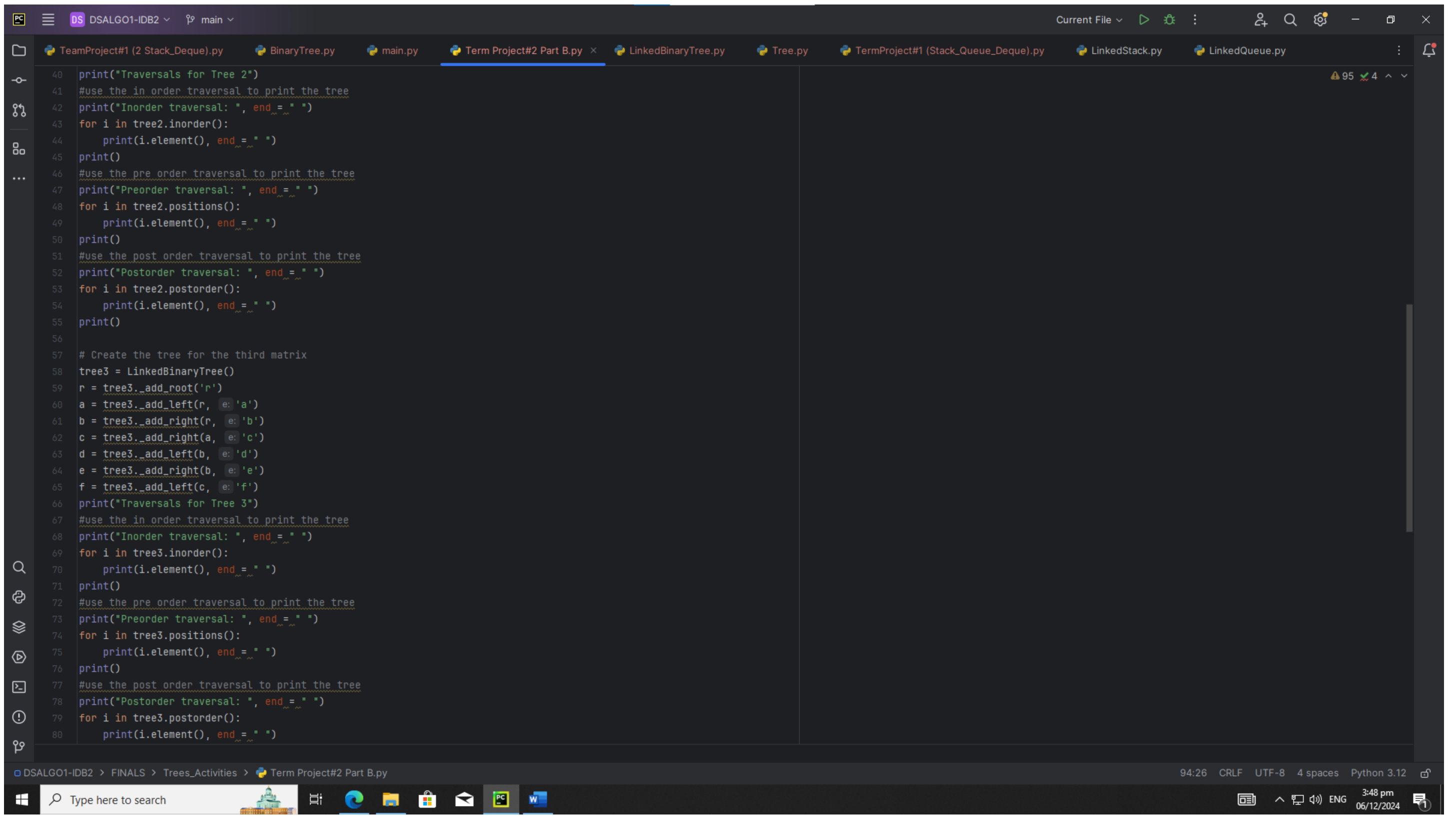


```python
tree6._add_left(num_left, '+')
tree6._add_right(num_left, '-')
tree6._add_left(tree6.left(num_left), 5)
tree6._add_right(tree6.left(num_left), 2)
tree6._add_left(tree6.right(num_left), 2)
tree6._add_right(tree6.right(num_left), 1)

# Subtree for the denominator: (2 + 9) + ((7 - 2) - 1)
den_right = tree6._add_right(left_sub, '+')
tree6._add_left(den_right, '+')
tree6._add_right(den_right, '-')
tree6._add_left(tree6.left(den_right), 2)
tree6._add_right(tree6.left(den_right), 9)
tree6._add_left(tree6.right(den_right), '-')
tree6._add_right(tree6.right(den_right), 1)
tree6._add_left(tree6.left(tree6.right(den_right)), 7)
tree6._add_right(tree6.left(tree6.right(den_right)), 2)

print("Traversals for Tree 6")
#use the in order traversal to print the tree
print("Inorder traversal: ", end = " ")
for i in tree6.inorder():
    print(i.element(), end = " ")
print()
#use the pre order traversal to print the tree
print("Preorder traversal: ", end = " ")
for i in tree6.positions():
    print(i.element(), end = " ")
print()

#use the post order traversal to print the tree
print("Postorder traversal: ", end = " ")
for i in tree6.postorder():
    print(i.element(), end = " ")
print()
```

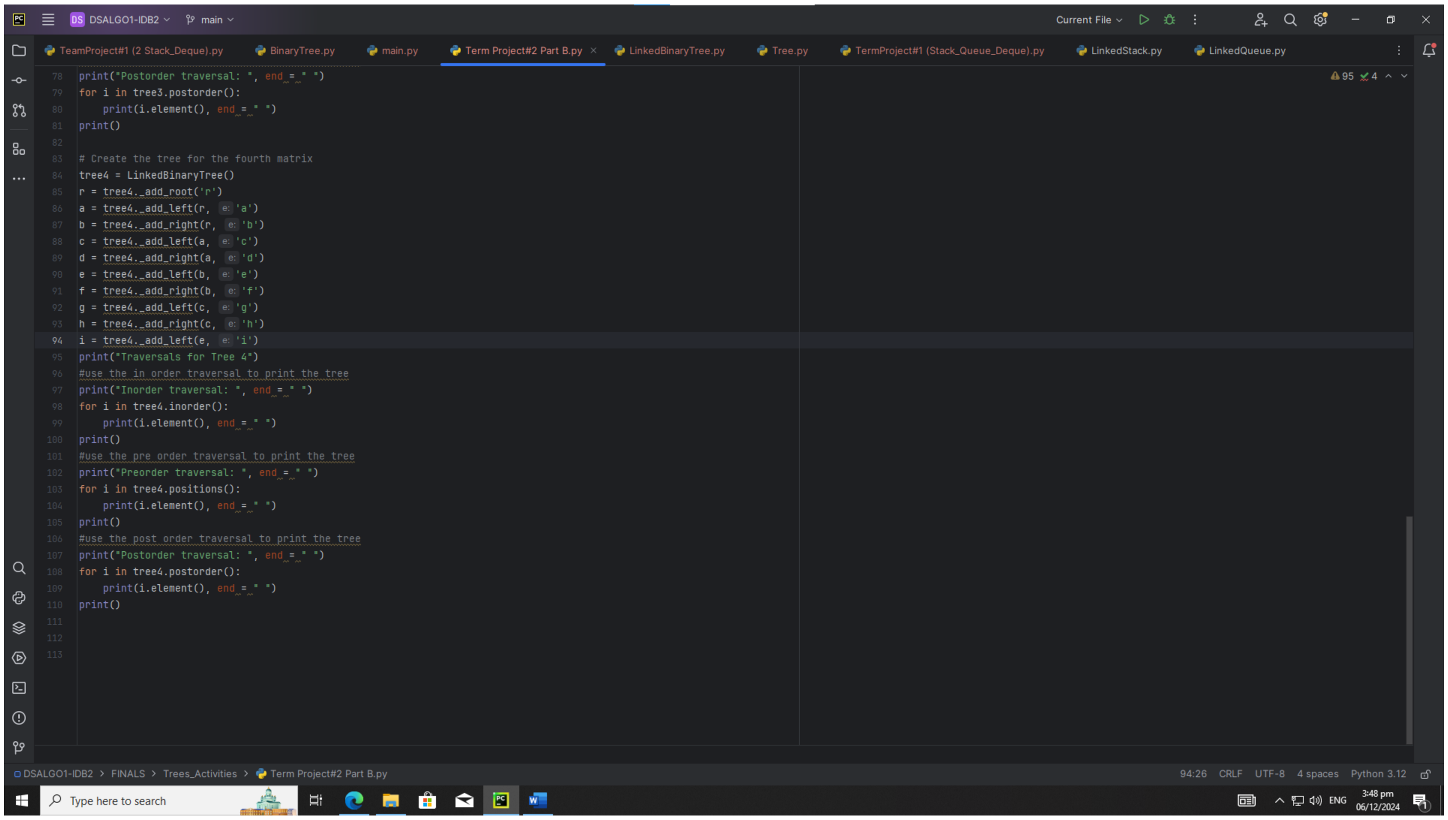
SOURCE CODE PART B:



```python
from LinkedBinaryTree import LinkedBinaryTree

tree1 = LinkedBinaryTree()
r = tree1._add_root('r')
a = tree1._add_left(r, 'a')
b = tree1._add_left(a, 'b')
d = tree1._add_right(b, 'd')
c = tree1._add_right(a, 'c')
e = tree1._add_left(c, 'e')
g = tree1._add_right(e, 'g')
h = tree1._add_right(g, 'h')
f = tree1._add_right(c, 'f')
print("Traversals for Tree 1")
#use the in order traversal to print the tree
print("Inorder traversal: ", end = " ")
for i in tree1.inorder():
    print(i.element(), end = " ")
print()
#use the pre order traversal to print the tree
print("Preorder traversal: ", end = " ")
for i in tree1.positions():
    print(i.element(), end = " ")
print()
#use the post order traversal to print the tree
print("Postorder traversal: ", end = " ")
for i in tree1.postorder():
    print(i.element(), end = " ")
print()

# Create the tree for the second matrix
tree2 = LinkedBinaryTree()
r = tree2._add_root('r')
a = tree2._add_left(r, 'a')
b = tree2._add_right(r, 'b')
c = tree2._add_left(a, 'c')
d = tree2._add_right(a, 'd')
e = tree2._add_right(b, 'e')
f = tree2._add_right(e, 'f')
g = tree2._add_right(f, 'g')
print("Traversals for Tree 2")
#use the in order traversal to print the tree
```



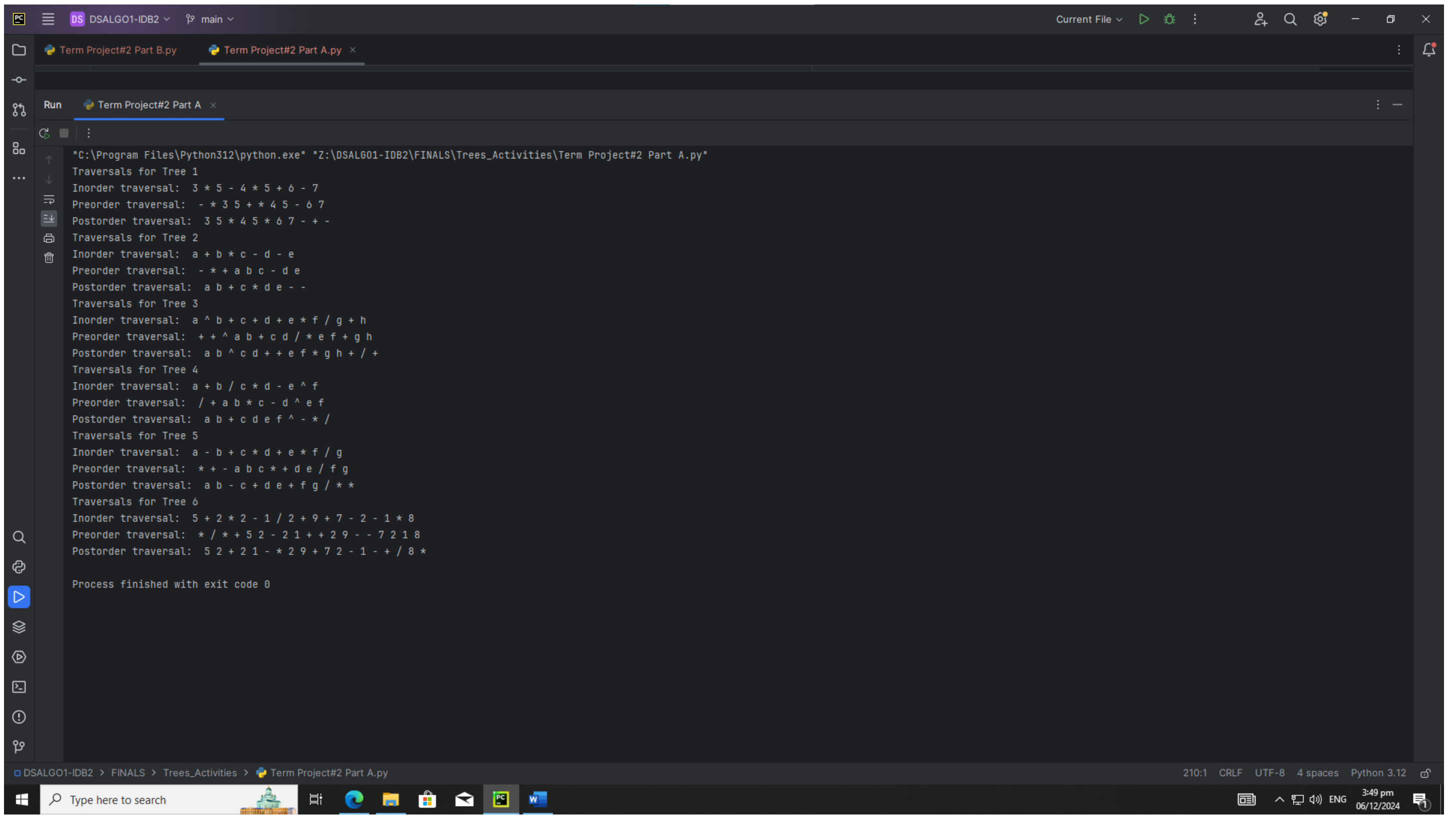
```python
print("Traversals for Tree 2")
#use the in order traversal to print the tree
print("Inorder traversal: ", end = " ")
for i in tree2.inorder():
    print(i.element(), end = " ")
print()
#use the pre order traversal to print the tree
print("Preorder traversal: ", end = " ")
for i in tree2.positions():
    print(i.element(), end = " ")
print()
#use the post order traversal to print the tree
print("Postorder traversal: ", end = " ")
for i in tree2.postorder():
    print(i.element(), end = " ")
print()

# Create the tree for the third matrix
tree3 = LinkedBinaryTree()
r = tree3._add_root('r')
a = tree3._add_left(r, 'a')
b = tree3._add_right(r, 'b')
c = tree3._add_right(a, 'c')
d = tree3._add_left(b, 'd')
e = tree3._add_right(b, 'e')
f = tree3._add_left(c, 'f')
print("Traversals for Tree 3")
#use the in order traversal to print the tree
print("Inorder traversal: ", end = " ")
for i in tree3.inorder():
    print(i.element(), end = " ")
print()
#use the pre order traversal to print the tree
print("Preorder traversal: ", end = " ")
for i in tree3.positions():
    print(i.element(), end = " ")
print()
#use the post order traversal to print the tree
print("Postorder traversal: ", end = " ")
for i in tree3.postorder():
    print(i.element(), end = " ")
```



```python
print("Postorder traversal: ", end = " ")
for i in tree3.postorder():
    print(i.element(), end = " ")
print()

# Create the tree for the fourth matrix
tree4 = LinkedBinaryTree()
r = tree4._add_root('r')
a = tree4._add_left(r, 'a')
b = tree4._add_right(r, 'b')
c = tree4._add_left(a, 'c')
d = tree4._add_right(a, 'd')
e = tree4._add_left(b, 'e')
f = tree4._add_right(b, 'f')
g = tree4._add_left(c, 'g')
h = tree4._add_right(c, 'h')
i = tree4._add_left(e, 'i')
print("Traversals for Tree 4")
#use the in order traversal to print the tree
print("Inorder traversal: ", end = " ")
for i in tree4.inorder():
    print(i.element(), end = " ")
print()
#use the pre order traversal to print the tree
print("Preorder traversal: ", end = " ")
for i in tree4.positions():
    print(i.element(), end = " ")
print()
#use the post order traversal to print the tree
print("Postorder traversal: ", end = " ")
for i in tree4.postorder():
    print(i.element(), end = " ")
print()
```

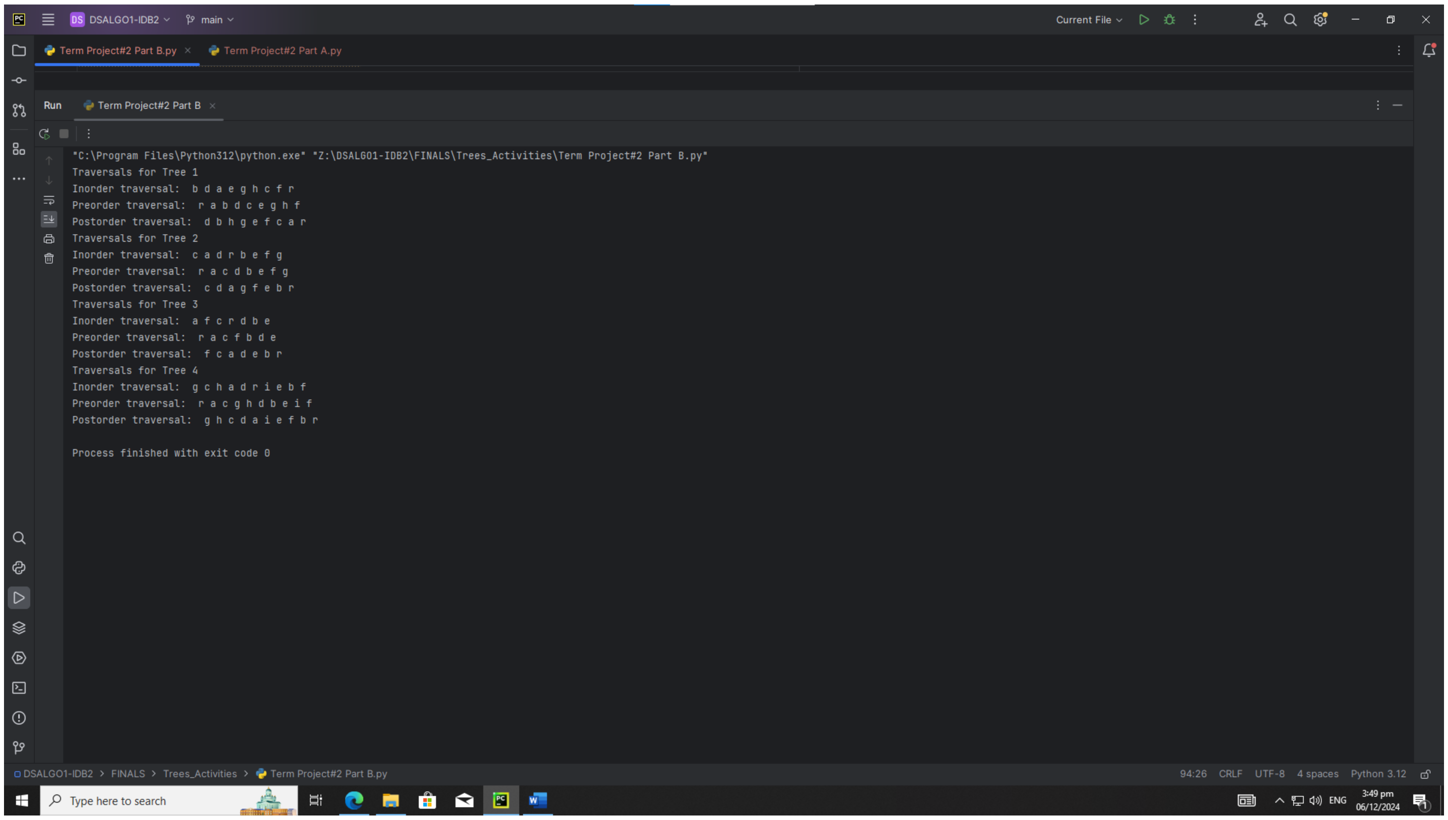
OUTPUT:

TERM PROJECT PART A



```
"C:\Program Files\Python312\python.exe" "Z:\DSALGO1-IDB2\FINALS\Trees_Activities\Term Project#2 Part A.py"
Traversals for Tree 1
Inorder traversal:  3 * 5 - 4 * 5 + 6 - 7
Preorder traversal:  - * 3 5 + * 4 5 - 6 7
Postorder traversal:  3 5 * 4 5 * 6 7 - + -
Traversals for Tree 2
Inorder traversal:  a + b * c - d - e
Preorder traversal:  - * + a b c - d e
Postorder traversal:  a b + c * d e - -
Traversals for Tree 3
Inorder traversal:  a ^ b + c + d + e * f / g + h
Preorder traversal:  + + ^ a b + c d / * e f + g h
Postorder traversal:  a b ^ c d + + e f * g h + / +
Traversals for Tree 4
Inorder traversal:  a + b / c * d - e ^ f
Preorder traversal:  / + a b * c - d ^ e f
Postorder traversal:  a b + c d e f ^ - * /
Traversals for Tree 5
Inorder traversal:  a - b + c * d + e * f / g
Preorder traversal:  * + - a b c * + d e / f g
Postorder traversal:  a b - c + d e + f g / * *
Traversals for Tree 6
Inorder traversal:  5 + 2 * 2 - 1 / 2 + 9 + 7 - 2 - 1 * 8
Preorder traversal:  * / * + 5 2 - 2 1 + + 2 9 - - 7 2 1 8
Postorder traversal:  5 2 + 2 1 - * 2 9 + 7 2 - 1 - + / 8 *

Process finished with exit code 0
```

PART B:



```
"C:\Program Files\Python312\python.exe" "Z:\DSALGO1-IDB2\FINALS\Trees_Activities\Term Project#2 Part B.py"
Traversals for Tree 1
Inorder traversal:  b d a e g h c f r
Preorder traversal:  r a b d c e g h f
Postorder traversal:  d b h g e f c a r
Traversals for Tree 2
Inorder traversal:  c a d r b e f g
Preorder traversal:  r a c d b e f g
Postorder traversal:  c d a g f e b r
Traversals for Tree 3
Inorder traversal:  a f c r d b e
Preorder traversal:  r a c f b d e
Postorder traversal:  f c a d e b r
Traversals for Tree 4
Inorder traversal:  g c h a d r i e b f
Preorder traversal:  r a c g h d b e i f
Postorder traversal:  g h c d a i e f b r

Process finished with exit code 0
```