| | |
|---|---|
| Ex. No: 11 | UIT2201 — Programming and Data Structures |
| 08-07-2023 | |

**Aim:**

To execute the following programs and note the output.

AbstractTree.py

```
from abc import abstractmethod
from abc import ABC


class AbstractTree(ABC):
    @abstractmethod
    def getRoot(self):
        """Returns the root position of the tree."""
        pass

    @abstractmethod
    def getParent(self, pos):
        """Returns the parent position of the given position 'pos'."""
        pass

    @abstractmethod
    def getNum_children(self, pos):
        """Returns the number of children of the given position 'pos'."""
        pass

    @abstractmethod
    def getChildren(self, pos):
        """Returns a list of children positions of the given position
'pos'."""
        pass

    @abstractmethod
    def __len__(self):
        """Returns the total number of positions in the tree."""
        pass

    def isRoot(self, pos):
        """Returns True if the given position 'pos' is the root of the
tree, False otherwise."""
        return self.getRoot() == pos

    def isLeaf(self, pos):
        """Returns True if the given position 'pos' is a leaf node (has no
children), False otherwise."""
        return self.getNum_children(pos) == 0

    def isEmpty(self):
        """Returns True if the tree is empty (has no positions), False
otherwise."""
        return len(self) == 0
    def depthN(self, pos):
        """
        Returns the depth of the position 'pos' in the tree.
        Depth is the number of edges in the path from the root to 'pos'.
        """
        if self.isRoot(pos):
            return 0
        return 1 + self.depthN(self.getParent(pos))

    def heightN(self, pos):
        """
        Returns the height of the position 'pos' in the tree.
        Height is the number of edges in the longest path from 'pos' to a
leaf.
        """
        if self.isLeaf(pos):
            return 0
        return 1 + max([self.heightN(child) for child in
self.getChildren(pos)])

    def height(self):
        """Returns the height of the tree (i.e., the height of the root
position)."""
        return self.heightN(self.getRoot())
```

## AbstractBinaryTree.py

```
from abc import abstractmethod

from AbstractTree import AbstractTree


class AbstractBinaryTree(AbstractTree):
    @abstractmethod
    def getLeft(self, pos):
        """Return the left child of the given position."""
        pass

    @abstractmethod
    def getRight(self, pos):
        """Return the right child of the given position."""
        pass

    def getChildren(self, pos):
        """Return the children of the given position."""
        if pos is None:
            return None
        if self.getLeft(pos) is not None:
            yield self.getLeft(pos)
        if self.getRight(pos) is not None:
            yield self.getRight(pos)

    def sibling(self, pos):
        """Return the sibling of the given position."""
        parent = self.getParent(pos)
        if parent is None:
            return None
        if pos == self.getRight(parent):
            return self.getLeft(parent)
        else:
            return self.getRight(parent)
```

LinkedBinaryTree.py

```
from AbstractBinaryTree import AbstractBinaryTree


class LinkedBinaryTree(AbstractBinaryTree):
    class BTNode:
        """A node class for the LinkedBinaryTree."""
        __slots__ = ["item", "left", "right", "parent"]

        def __init__(self, item, left=None, right=None, parent=None):
            """
            Initialize a new BTNode.

            Args:
                item: The item stored in the node.
                left: The left child node.
                right: The right child node.
                parent: The parent node.
            """
            self.item = item
            self.left = left
            self.right = right
            self.parent = parent

        def getitem(self):
            """Return the item stored in the node."""
            return self.item

        def setitem(self, item):
            """Set the item stored in the node."""
            self.item = item

    __slots__ = ["root", "size"]

    def __init__(self, item=None, t_left=None, t_right=None):
        """
        Initialize a new LinkedBinaryTree.

        Args:
            item: The item to be stored in the root node.
            t_left: Another LinkedBinaryTree to be used as the left
subtree.
            t_right: Another LinkedBinaryTree to be used as the right
subtree.
        """
        self.root = None  # Initialize the root node
        self.size = 0  # Initialize the size of the tree
        self.string = ""  # Initialize an empty string
        if item is not None:
            self.root = self.addRoot(item)  # Create the root node with the
given item
        if t_left is not None:
            if t_left.root is not None:
                t_left.root.parent = self.root  # Set the parent of the
left subtree to the root
                self.root.left = t_left.root  # Set the left subtree of the
root
                self.size += t_left.size  # Update the size of the tree
                t_left.root = None  # Clear the root of the left subtree
        if t_right is not None:
            if t_right.root is not None:
                t_right.root.parent = self.root  # Set the parent of the
right subtree to the root
                self.root.right = t_right.root  # Set the right subtree of
the root
                self.size += t_right.size  # Update the size of the tree
                t_right.root = None  # Clear the root of the right subtree

    def addRoot(self, item):
        """
        Adds a root node with the given item to the tree.

        Args:
            item: The item to be stored in the root node.

        Returns:
            The root position of the added node.

        Raises:
            ValueError: If the root already exists.
        """
        if self.root is not None:
            raise ValueError("Root already exists")
        else:
            self.root = self.BTNode(item)
            self.size += 1
            return self.root

    def __len__(self):
        """
        Returns the number of nodes in the tree.

        Returns:
            The size of the tree.
        """
        return self.size

    def getParent(self, pos):
        """
        Returns the parent position of the given position 'pos'.

        Args:
            pos: The position to get the parent of.

        Returns:
            The parent position of 'pos'.
        """
        return pos.parent

    def getLeft(self, pos):
        """
        Returns the left child position of the given position 'pos'.

        Args:
            pos: The position to get the left child of.

        Returns:
            The left child position of 'pos'.
        """
        return pos.left

    def getRight(self, pos):
        """
        Returns the right child position of the given position 'pos'.

        Args:
            pos: The position to get the right child of.

        Returns:
            The right child position of 'pos'.
        """
        return pos.right

    def getRoot(self):
        """
        Returns the root position of the tree.

        Returns:
            The root position.
        """
        return self.root

    def getSize(self):
        """
        Returns the number of nodes in the tree.

        Returns:
            The size of the tree.
        """
        return self.size

    def getNum_children(self, pos):
        """
        Returns the number of children of the given position 'pos'.

        Args:
            pos: The position to get the number of children of.

        Returns:
            The number of children of 'pos'.
        """
        if pos is None:
            return 0
        else:
            return 1 + self.getNum_children(pos.left) +
self.getNum_children(pos.right)

    def addLeft(self, item, pos=None):
        """
        Adds a left child node with the given item to the specified
position 'pos' or the root if 'pos' is None.

        Args:
            item: The item to be stored in the left child node.
            pos: The position to add the left child to. If None, the left
```

```
child is added to the root.

        Returns:
            The position of the added left child node.

        Raises:
            ValueError: If the left child already exists.
        """
        if pos is None:
            pos = self.root
        if self.getLeft(pos) is not None:
            raise ValueError("Left child already exists")
        else:
            pos.left = self.BTNode(item, parent=pos)
            self.size += 1
            return pos.left

    def addRight(self, item, pos=None):
        """
        Adds a right child node with the given item to the specified
position 'pos' or the root if 'pos' is None.

        Args:
            item: The item to be stored in the right child node.
            pos: The position to add the right child to. If None, the right
child is added to the root.

        Returns:
            The position of the added right child node.

        Raises:
            ValueError: If the right child already exists.
        """
        if pos is None:
            pos = self.root
        if self.getRight(pos) is not None:
            raise ValueError("Right child already exists")
        else:
            pos.right = self.BTNode(item, parent=pos)
            self.size += 1
            return pos.right

    def preorder(self, pos):
        """
        Performs a preorder traversal starting from the given position
'pos'.

        Args:
            pos: The starting position for the preorder traversal.
        """
        self.string += str(pos.item) + ","
        if pos.left is not None:
            self.preorder(pos.left)
        if pos.right is not None:
            self.preorder(pos.right)

    def postorder(self, pos):
        """
        Performs a postorder traversal starting from the given position
'pos'.
```

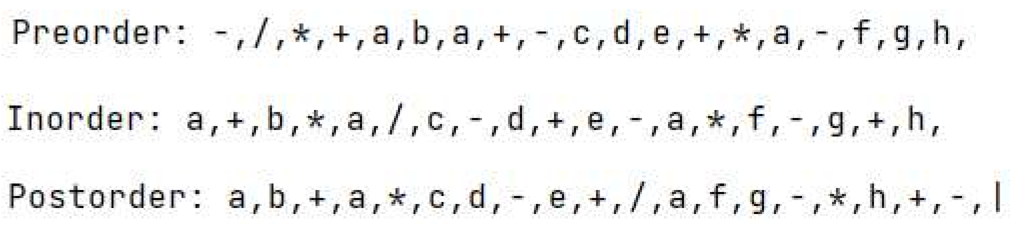
```
        Args:
            pos: The starting position for the postorder traversal.
        """
        if pos.left is not None:
            self.postorder(pos.left)
        if pos.right is not None:
            self.postorder(pos.right)
        self.string += str(pos.item) + ","

    def inorder(self, pos):
        """
        Performs an inorder traversal starting from the given position
'pos'.

        Args:
            pos: The starting position for the inorder traversal.
        """
        if pos.left is not None:
            self.inorder(pos.left)
        self.string += str(pos.item) + ","
        if pos.right is not None:
            self.inorder(pos.right)

    def __str__(self):
        """
        Returns a string representation of the tree by performing preorder,
inorder, and postorder traversals.

        Returns:
            A string representation of the tree.
        """
        self.string = "Preorder: "
        self.preorder(self.root)
        self.string += "|Inorder: "
        self.inorder(self.root)
        self.string += "|Postorder: "
        self.postorder(self.root)
        self.string += "|"
        return self.string

    def mirror(self, pos):
        """
        Create a new LinkedBinaryTree representing the mirror image of the
original tree.

        Returns:
            A new LinkedBinaryTree that is the mirror image of the original
tree.
        """
        if self.isLeaf(pos):
            return None
        if pos is not None:
            pos.left, pos.right = pos.right, pos.left
            self.mirror(pos.left)
            self.mirror(pos.right)
```

1. Write a parser that takes an expression string in postfix notation (for eg, "ab+a*cd-e+/afg-*h+-) and constructs the corresponding expression tree. You may assume that only binary operators are used in the expression and all the identifiers are single characters only.

**Code:**

```python
from LinkedBinaryTree import LinkedBinaryTree


class ExpressionTree(LinkedBinaryTree):
    def __init__(self, item=None, t_left=None, t_right=None):
        super().__init__(item, t_left, t_right)

    def construct(self, string):
        """
        Constructs an expression tree from a postfix expression string.

        Args:
            string: A string representing a postfix expression.

        Returns:
            The root position of the constructed expression tree.
        """
        s = []
        for ch in string:
            if ch in "+-*/":
                r_child = s.pop()
                l_child = s.pop()
                s.append(ExpressionTree(ch, l_child, r_child))
            else:
                s.append(ExpressionTree(ch))

        self.root = s.pop().getRoot()
        return self.root


if __name__ == "__main__":
    E = ExpressionTree()
    E.construct("ab+a*cd-e+/afg-*h+-")
    print(E)
```

**Inputs and Output:**



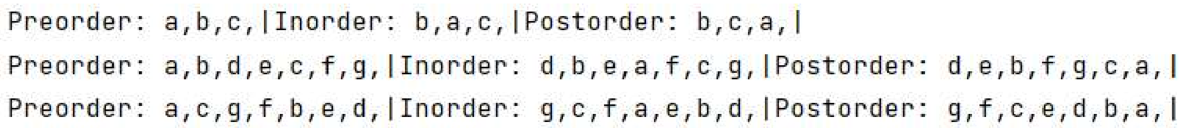
```
Preorder: -,/,*,+,a,b,a,+,-,c,d,e,+,*,a,-,f,g,h,

Inorder: a,+,b,*,a,/,c,-,d,+,e,-,a,*,f,-,g,+,h,

Postorder: a,b,+,a,*,c,d,-,e,+,/,a,f,g,-,*,h,+,-,
```

2. Given a binary tree, write a Python code to convert the binary tree into its Mirror tree. Mirror of a Binary Tree T is another Binary Tree M(T) with left and right children of all non-leaf nodes interchanged.

**Code:**

```
from LinkedBinaryTree import LinkedBinaryTree


def main():
    tree = LinkedBinaryTree()
    tree.addRoot("a")
    tree.addLeft("b")
    tree.addRight("c")
    print(tree)
    tree.addLeft("d", tree.root.left)
    tree.addRight("e", tree.root.left)
    tree.addLeft("f", tree.root.right)
    tree.addRight("g", tree.root.right)
    print(tree)
    tree.mirror(tree.root)
    print(tree)

if __name__ == "__main__":
    main()
```

**Output:**



```
Preorder: a,b,c,|Inorder: b,a,c,|Postorder: b,c,a,|
Preorder: a,b,d,e,c,f,g,|Inorder: d,b,e,a,f,c,g,|Postorder: d,e,b,f,g,c,a,|
Preorder: a,c,g,f,b,e,d,|Inorder: g,c,f,a,e,b,d,|Postorder: g,f,c,e,d,b,a,|
```

========================