ABSTRACT:

The research paper for designing and analysis of algorithms is based on tree traversals of binary tree which mainly focus on the research of '*Pre-Order and Post-Order Traversals*'. These traversal are based on the *Python* programming language which focuses on the programming design, experimental setup, analysis, and implementation of our semester's research paper.

In computer science, a tree is an abstract model of a hierarchical structure. It consists of nodes with a parent-child relation. Applications of tree traversals include; organization charts, file systems, programming environments etc. A traversal visits the nodes of a tree in a systematic manner. In such a manner, pre-order traversals nodes are visited before its descendant, whereas post-order traversal nodes are visited after its descendants

The background information for tree traversals is presented as well as the design criteria for our research paper. The details and working behind how the algorithm works are examined, along with theimplementations in the algorithm design. A detailed examination of the chosen specifications and the methodology at each step is examined and explained, as well as explanations for certain choices of component values. The courses' structure and learning outcomes, the choices of tools and equipment, how the courses progressed, the lessons learned, the specifics of getting all the group members engaged, and the plans work aimed at improvement of programming and computer networking skills via the involvement in different aspects of programming are discussed. The results of a computer debugging results using the software '*PyCharm*' are shown, along with the explanation of the output accordingly. By the end of implementing this research paper, we will able to understand the algorithm, implement it any development language and analyze the complexity of the algorithm.

TABLE OF CONTENTS:

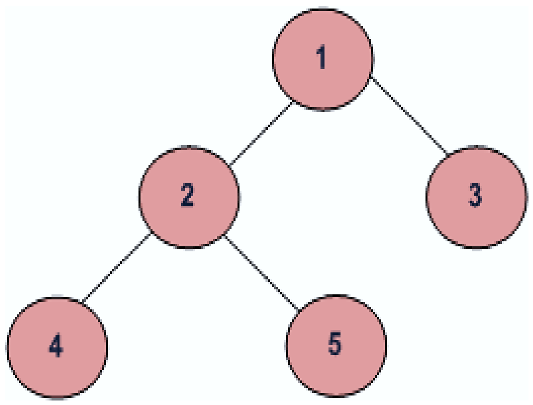
“Pre-Order & Post-Order.”

Chapter 01: Preface

1.1 Introduction

Tree traversal is the process of visiting each node in the tree exactly once, visiting each node in a graph should be done in a systematic manner. If search result in a visit to all the vertices, it is called a traversal. There are three traversal techniques for a binary tree that includes; Pre-order, In-order, and Post-order traversal. Our research paper mainly focuses on the designing and analysis of Pre-order and Post-order traversals' algorithms.

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc.) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Depth First Traversals:

- Pre-Order (Root, Left, Right): *1 2 4 5 3*
- Post-Order (Left, Right, Root): *4 5 2 3 1*

Breadth First or Level Order Traversal: *1 2 3 4 5*

- **Pre-Order Traversal**

Algorithm Pre-Order (tree)

```
1. Visit the root.
2. Traverse the left subtree, i.e., call Pre-order (left subtree).
3. Traverse the right subtree, i.e., call Pre-order (right-subtree).
```

Pseudocode for Pre-Order Traversal

```
preOrder(node):
    if node is not null:
        print node.data
        preOrder (node.left)
        preOrder (node.right)
```

- **Post-Order Traversal**

Algorithm Post-Order (tree)

1. Traverse the left subtree, i.e., call Post-order (left subtree).
2. Traverse the right subtree, i.e., call Post-order (right subtree).
3. Visit the root.

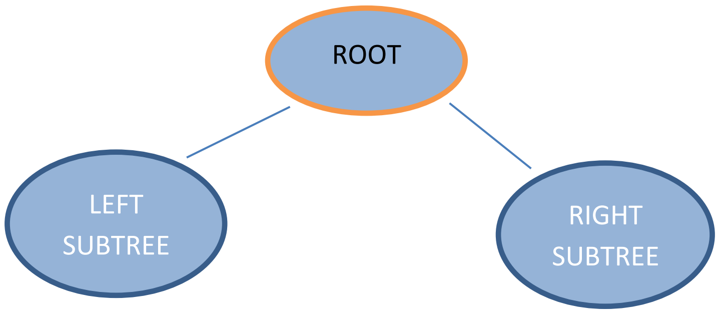
Pseudocode for Post-Order Traversal

```
postOrder(node):
    if node is not null:
        postOrder (node.left)
        postOrder (node.right)
        print node.value
```

Chapter 02: Design & Implementation

2.1 Programming Design

This is general diagram of tree. Now we will discuss different traversal technique applied on it.



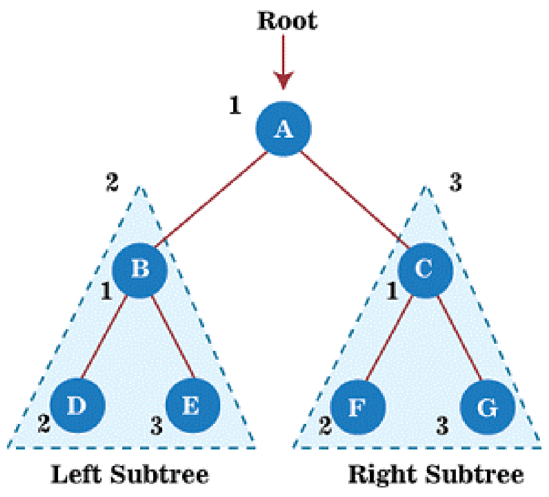
- ### Pre-Order Traversal:

This technique follows the *'ROOT- LEFT- RIGHT'* policy. Output of the preorder traversal of the above tree is:

A → B → D → E → C → F → G

Explanation

First, we traverse the root node *A*. Then move to its left subtree *B,* then to its right subtree *C* which will be traversed in pre-order.



- **Root A**

First, we traverse the root node *A.*

- **Left subtree B**

For left sub-tree *B*, the root node **B** is traversed followed by its left subtree *D*. Since node *D* does not have any children, then move to its right subtree *E.* As node *E* also does not have any children, the traversal of the left subtree of root node *A* is completed.

- **Right subtree C**

For right subtree *C*, the root node *C* has traversed followed by the left subtree *F*. Since node *F* does not have any children, move to the right subtree *G*. As node G also does not have any children, traversal of the right subtree of root node *A* is completed.

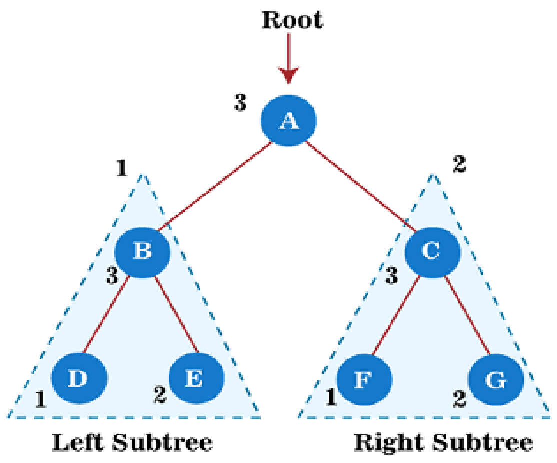
- **Post-Order Traversal:**

This technique follows *'LEFT- RIGHT- ROOT'* policy. The output of the postorder traversal of the above tree is

D → E → B → F → G → C → A

Explanation

First, we traverse the left subtree B in post-order. Then the right subtree *C* followed by the root node of the above tree, i.e., *A*, is traversed.



- **Left Sub-tree *B***

For left sub-tree *B*, its left subtree *D* is traversed. Since it does not have any children, traverse the right subtree *E*. As node *E* also does not have any children, move to the root node *B*. After traversing node *B*, the traversal of the left subtree of root node *A* is completed.

- **Right Sub-tree *C***

Now, move towards the right sub-tree. The left subtree ***F*** is traversed. Since it does not have any children, traverse the right subtree *G*. As node *G* also does not have any children, therefore, finally, the root node of the right subtree, i.e., *C,* is traversed. The traversal of the right subtree is completed.

- **Root *A***

At last, we traverse the root node *A*.

2.2 Experimental Setup

To determine the accuracy of the tree traversal algorithm, and its use of both memory and time we test different algorithms and created a collection of programs to determine if the analysis correctly identifies standard tree traversals. All the experiments were performed using the *PyCharm.* Instrumentation was added to the code by making minor modifications and add function calls to the program to make program efficient wherever a tree pointer was dereferenced. A proper selection of a representative input to profiling could help ensure that the analysis does not take too long. Most of the work is performed by a static library used to profile instrumented code which only slightly increases memory use. After analyzing different codes and examining their effectiveness in properly determination of tree traversals, our code gives us best optimal results. The best optimal running time that we get is ***O (n)*** and ***O (1)*** space complexity.

- **Time Complexity:**

Time Complexity: ***O(n)***
Complexity function *T(n)*, for all problem where tree traversal is involved can be defined as:

$$T(n) = T(k) + T(n - k - 1) + c$$

Where *k* is the number of nodes on one side of root and *n - k - 1* on the other side.

- **Space Complexity**

If we don't consider size of stack for function calls, then *O (1)* otherwise *O (n).*

2.3 Code

```
# Python Program for Pre-Order and Post-Order Traversal.
# 'Node' Class.
class Node:
   def __init__(self, data):
      self.data = data
      self.right = None
      self.left = None

# Function to Display "Pre-Order Traversal" for the Tree.
def preOrder(p):
   if (p == None):
      return

   else:
      print(p.data, end=' ')
      preOrder(p.left)
      preOrder(p.right)

# Function to Display "Post-Order Traversal" for the Tree.
def postOrder(p):
   if (p == None):
      return

   else:
      postOrder(p.left)
      postOrder(p.right)
      print(p.data, end=' ')

# Main Code.
if __name__ == '__main__':
```

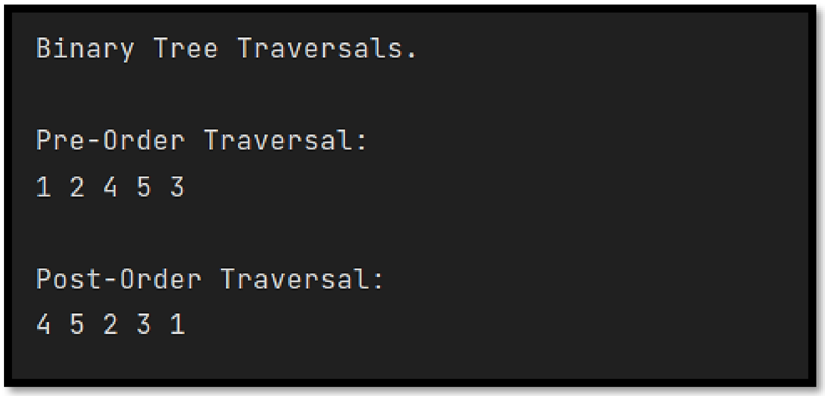
```
# Root Node of the Tree.
root = Node('1')

# Left SubTree.
root.left = Node('2')
root.left.left = Node('4')
root.left.right = Node('5')

# Right SubTree.
root.right = Node('3')

# Displaying the Pre-Order and Post-Order Traversals.
print("\nBinary Tree Traversals.")

print("\nPre-Order Traversal: ")
preOrder(root)
print()

print("\nPost-Order Traversal: ")
postOrder(root)
print()
```

Chapter 03: Debugging Result

3.1 Result and Discussion

Separate functions for *'Pre-Order'* and *'Post-Order' traversal* are implemented in the code according to their node's placement, and then later on called in the main driver function for displaying the respective root node, left and right nodes of the binary tree.

The result output for the Pre-Order and Post-Order traversal of the binary tree is shown below.



```
Binary Tree Traversals.

Pre-Order Traversal:
1 2 4 5 3

Post-Order Traversal:
4 5 2 3 1
```

Chapter 04: Conclusion

Our final ***Design and Analysis of Algorithm*** (*209111*) semester project is based on implementing the research paper on '***Pre-Order and Post-Order Traversals'***. This report includes the concepts of data structures, depth-first search, and binary tree traversals in *Python* along with implementation of algorithms, pseudocodes, and detailed explanation.

While working on this research paper, we have learned numerous things about the classes and algorithm specifically, as well as the design, syntax, and reusability of the coding process in general. We hope to use the knowledge and experience obtained working on this project on future tasks in practical life.

Chapter 05: References

[1] https://www.programiz.com/dsa/tree-traversal

[2] https://medium.com/codex/in-order-pre-order-post-order-traversal-in-binary-trees-explained-in-python-1fc0c77f007f

[3] https://www.tutorialspoint.com/python_data_structure/python_tree_traversal_algorithms.htm