Traversal Algorithms

In this lesson, you will learn how to traverse binary trees using a depth-first search.

We'll cover the following

Tree Traversal

Tree Traversal is the process of visiting (checking or updating) each node in a tree data structure, exactly once. Unlike linked lists or one-dimensional arrays that are canonically traversed in linear order, trees may be traversed in multiple ways. They may be traversed in *depth-first* or *breadth-first* order.

There are three common ways to traverse a tree in depth-first order:

1. In-order
2. Pre-order
3. Post-order

Let’s begin with the Pre-order Traversal.

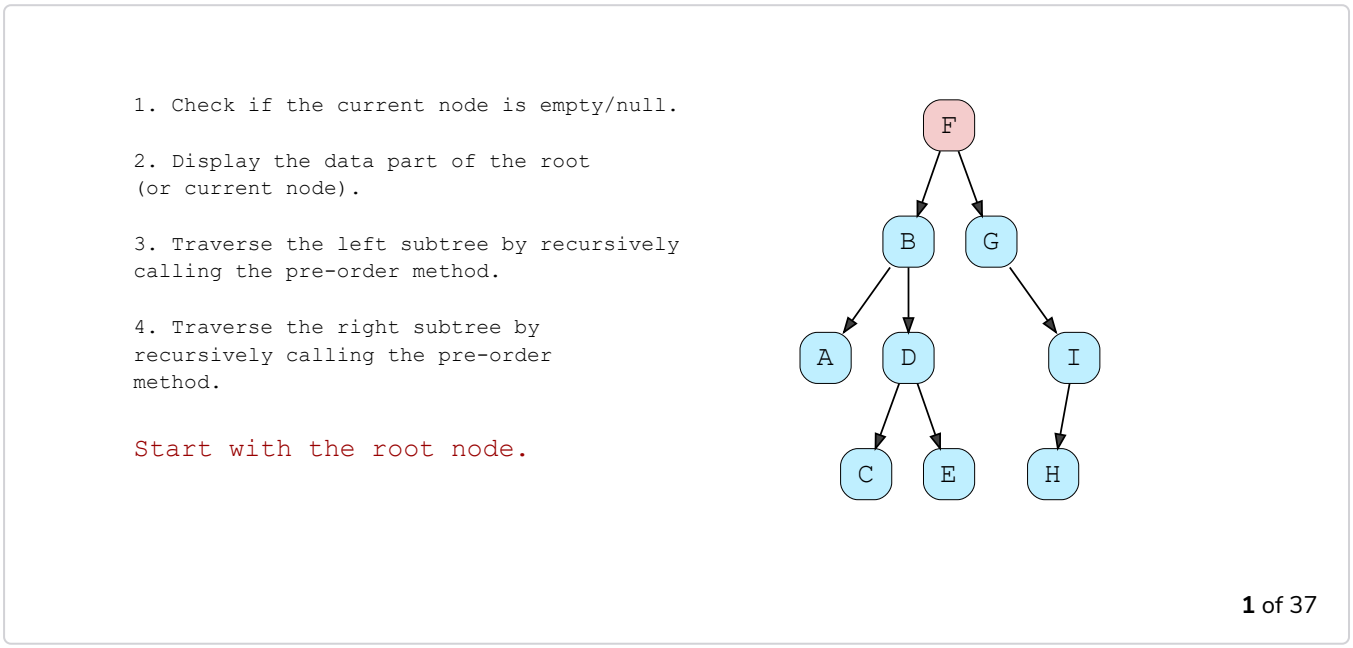
Pre-order Traversal

Here is the algorithm for a pre-order traversal:

1. Check if the current node is empty/null.
2. Display the data part of the root (or current node).

3. Traverse the left subtree by recursively calling the pre-order method.

4. Traverse the right subtree by recursively calling the pre-order method.



I hope the illustrations have made the algorithm pretty clear. Let's go over its implementation in Python:

```
def preorder_print(self, start, traversal):
  """Root->Left->Right"""
  if start:
    traversal += (str(start.value) + "-")
    traversal = self.preorder_print(start.left, traversal)
    traversal = self.preorder_print(start.right, traversal)
  return traversal
```

preorder_print(self, start, traversal)

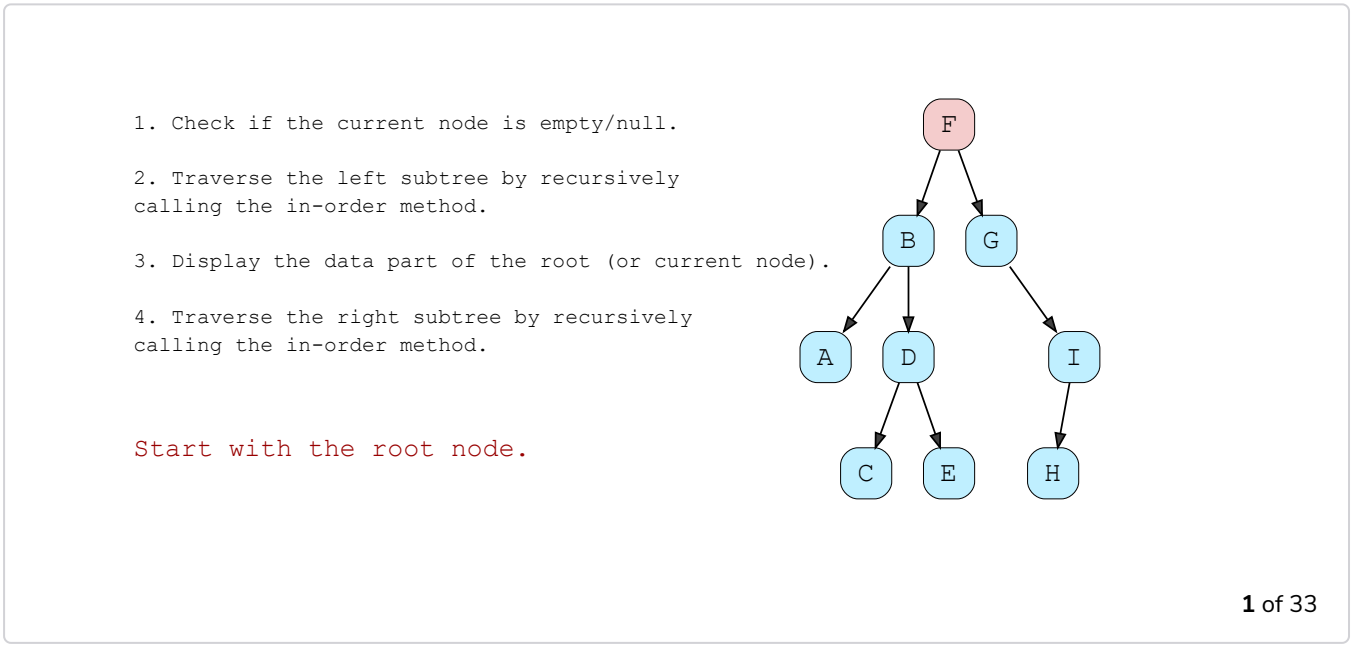
Just as specified in the algorithm, we check if `start` (i.e., the current node) is empty or not. If not, then we append `start.value` to the `traversal` string and recursively call `preorder_print` on `start.left` and `start.right` which are the right and left child of the current node. Finally, we return `traversal` from the method after we have returned from all the recursive calls in case `start` is not `None`. `traversal` is just a string that will concatenate the value of nodes in an order that we visited them.

In-order Traversal

Here is the algorithm for an in-order traversal:

Here is the algorithm for an in-order traversal:

1. Check if the current node is empty/null.
2. Traverse the left subtree by recursively calling the in-order method.
3. Display the data part of the root (or current node).
4. Traverse the right subtree by recursively calling the in-order method.



Now that you are familiar with the algorithm, let's jump to the code in Python:

```
def inorder_print(self, start, traversal):
        """Left->Root->Right"""
        if start:
            traversal = self.inorder_print(start.left, traversal)
            traversal += (str(start.value) + "-")
            traversal = self.inorder_print(start.right, traversal)
        return traversal
```

inorder_print(self, start, traversal)

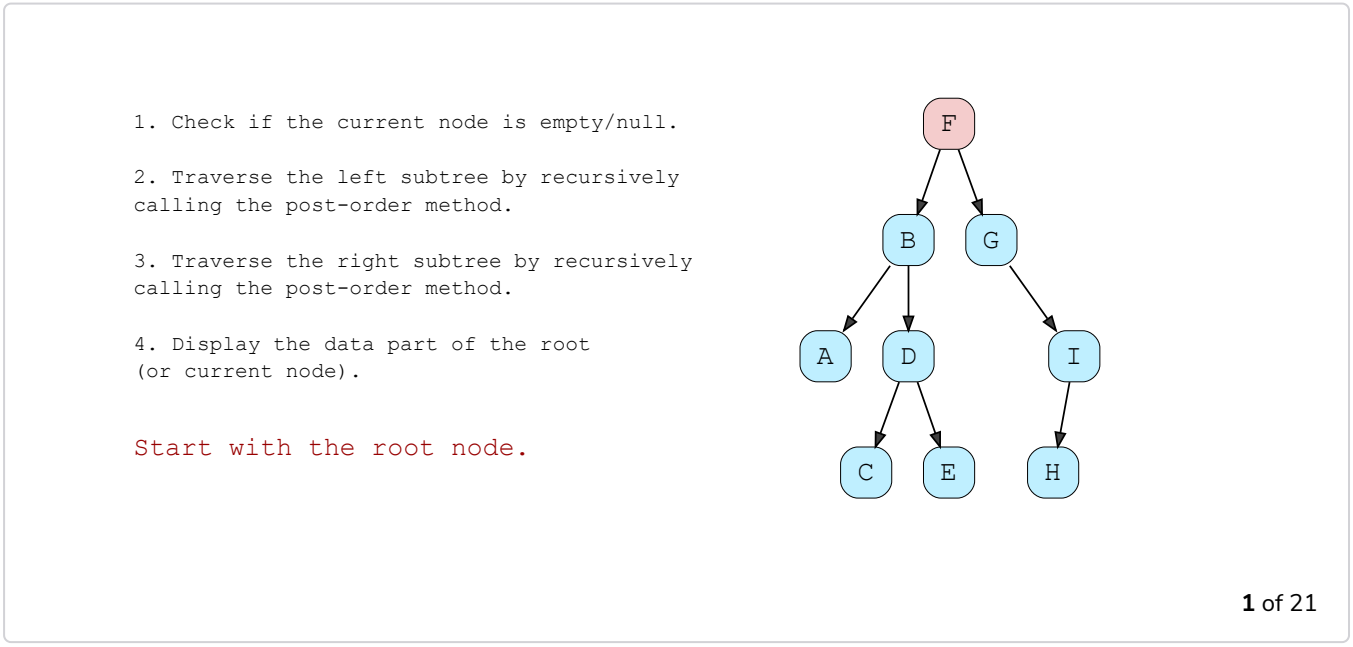
The `inorder_print` is pretty much the same as the `preorder_print` except that the order *Root->Left->Right* from pre-order changes to *Left->Root->Right* in in-order traversal. In order to achieve this order, we just change the order of statements in the if-condition, i.e., we first make a recursive call on the left child and after we are done will all the subsequent calls from **line 4**, we concatenate the value of the current node with `traversal` on **line 5**. Then, we can make a recursive call to right subtree on **line 6**. This will help us keep the

order required for the in-order traversal.

Post-order Traversal

At this point, it will be very easy for you to guess the algorithm for post-order traversal. There you go:

1. Check if the current node is empty/null.
2. Traverse the left subtree by recursively calling the post-order method.
3. Traverse the right subtree by recursively calling the post-order method.
4. Display the data part of the root (or current node).



Here is the implementation of post-order traversal in Python:

```
def postorder_print(self, start, traversal):
  """Left->Right->Root"""
  if start:
    traversal = self.postorder_print(start.left, traversal)
    traversal = self.postorder_print(start.right, traversal)
    traversal += (str(start.value) + "-")
  return traversal
```

postorder_print(self, start, traversal)

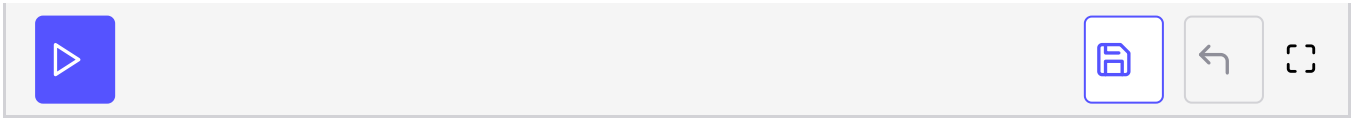
Again, we just changed the order of statements. The recursive calls to the left and the right subtree have been placed before concatenating the value of the current node to `traversal`.

Helper Method

Below is the implementation of all the tree traversal methods within the `Binary Tree` class. Additionally, there is a helper method `print_tree(self, traversal_type)` which will invoke the specified method according to `traversal_type`.

```
class Node(object):
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


class BinaryTree(object):
    def __init__(self, root):
        self.root = Node(root)

    def print_tree(self, traversal_type):
        if traversal_type == "preorder":
            return self.preorder_print(tree.root, "")
        elif traversal_type == "inorder":
            return self.inorder_print(tree.root, "")
        elif traversal_type == "postorder":
            return self.postorder_print(tree.root, "")

        else:
            print("Traversal type " + str(traversal_type) + " is not supported.")
            return False

    def preorder_print(self, start, traversal):
        """Root->Left->Right"""
        if start:
            traversal += (str(start.value) + "-")
            traversal = self.preorder_print(start.left, traversal)
            traversal = self.preorder_print(start.right, traversal)
        return traversal

    def inorder_print(self, start, traversal):
        """Left->Root->Right"""
        if start:
            traversal = self.inorder_print(start.left, traversal)
            traversal += (str(start.value) + "-")
            traversal = self.inorder_print(start.right, traversal)
        return traversal

    def postorder_print(self, start, traversal):
        """Left->Right->Root"""
        if start:
            traversal = self.postorder_print(start.left, traversal)
            traversal = self.postorder_print(start.right, traversal)
            traversal += (str(start.value) + "-")
        return traversal

# 1-2-4-5-3-6-7-
# 4-2-5-1-6-3-7
# 4-2-5-6-3-7-1
```

```
#               1
#           /       \
#          2          3
#         /  \      /   \
#        4    5     6   7

# Set up tree:
tree = BinaryTree(1)
tree.root.left = Node(2)
tree.root.right = Node(3)
tree.root.left.left = Node(4)
tree.root.left.right = Node(5)
tree.root.right.left = Node(6)
tree.root.right.right = Node(7)

#print(tree.print_tree("preorder"))
#print(tree.print_tree("inorder"))
print(tree.print_tree("postorder"))
```



A suggestion is to take out a piece of paper and traverse a sample binary tree yourself according to all the traversal types. Once that is done, you can confirm your results by playing around with the implementation provided above.

Hope you find these depth-first tree traversals useful! See you in the next lesson for level-order traversal which is a kind of breadth-first tree traversal.