Tree_traversal

September 17, 2023

1 Tree Traversal Techniques

Tree traversal is a fundamental operation in computer science used to explore and process the nodes of a tree data structure. There are two primary methods for traversing trees: depth-first traversal and breadth-first traversal.

In depth-first traversal, we start at the root node and explore as far down a branch as possible before backtracking. There are three common ways to perform depth-first traversal: pre-order, in-order, and post-order. Pre-order visits the current node before its children, in-order visits the left child before the current node and then the right child, and post-order visits the children before the current node.

In contrast, breadth-first traversal explores all the nodes at the current level before moving on to the next level. This method uses a queue data structure to keep track of the nodes to be processed, ensuring that nodes at the same level are visited before moving deeper into the tree.

The choice of traversal method depends on the specific problem and the order in which you need to access the nodes. Tree traversal is a crucial concept in various applications, including searching, sorting, and parsing data in hierarchical structures.

1.1 Defining a binary tree

[ ]:
```
class Node:
    def __init__(self, data):
        self.left = None
        self.right= None
        self.data = data
```

1.2 Filling up the binary tree

For now, we'll manually populate the binary tree. Later, we'll develop a method to automate this process. As we progress, we'll also explore how the population strategy varies depending on the chosen tree traversal technique.

[ ]:
```
#My_Tree = BinaryTree(1)
tree = Node(1)  # root
tree.left = Node(2)
tree.left.left = Node(4)
tree.left.left.left = Node(8)
```

```
tree.left.left.right = Node(9)

tree.left.right = Node(5)
tree.left.right.left = Node(10)
tree.left.right.right = Node(11)

tree.right = Node(3)
tree.right.left = Node(6)
tree.right.left.left = Node(12)
tree.right.left.right = Node(13)


tree.right.right = Node(7)
tree.right.right.left = Node(14)
tree.right.right.right = Node(15)
```

1.3 Depth-First Traversal: Pre-order

[ ]:
```
def PreorderTraversal(root):
    if root == None:
        return

    print(root.data)
    PreorderTraversal(root.left)
    PreorderTraversal(root.right)

PreorderTraversal(tree)
```

1.4 Depth-First Traversal: In-order

[ ]:
```
def InorderTraversal(root):
    if root == None:
        return

    InorderTraversal(root.left)
    print(root.data)
    InorderTraversal(root.right)

InorderTraversal(tree)
```

1.5 Depth-First Traversal: Post-order

[ ]:
```
def PostorderTraversal(root):
    if root == None:
        return
```

```
    PostorderTraversal(root.left)
    PostorderTraversal(root.right)
    print(root.data)

PostorderTraversal(tree)
```

2 Breadth-First Traversal

[ ]:
```
def BreadthFirstTraversal(root, queue=[]):
    if root == None:
        return

    print(root.data)
    if root.left !=None:
        queue.append(root.left)
    if root.right != None:
        queue.append(root.right)

    if len(queue)>0:
        next_node = queue.pop(0)
        BreadthFirstTraversal(next_node,queue)
    else:
        return

queue = []
BreadthFirstTraversal(tree,queue)
```

3 Print Tree Data with Levels

[ ]:
```
def BreadthFirstTraversalLevel(root,level=0,queue=[]):
    if root == None:
        return

    print(root.data, level)
    if root.left != None:
        queue.append([root.left,level+1])

    if root.right != None:
        queue.append([root.right,level+1])

    if len(queue)>0:
        next_node = queue.pop(0)
        BreadthFirstTraversalLevel(next_node[0],next_node[1],queue)
    else:
        return
```

```
queue = []
BreadthFirstTraversalLevel(tree,0,queue)
```

3.1 Inserting a Node in a Binary Tree

[ ]:
```
def Insert(root,value):
    if root == None:
        return Node(value)

    if value < root.data:
        root.left = Insert(root.left,value)
    else:
        root.right = Insert(root.right,value)

    return root
```

[ ]:
```
BreadthFirstTraversal(tree)
```

[ ]:
```
Insert(tree,6)

queue=[]
BreadthFirstTraversal(tree)
```

3.2 Binary Search Tree

[ ]:
```
def BinarySearchTree(List):
    bst = Node(List.pop(0))

    for l in List:
        Insert(bst,l)
    return bst
```

[ ]:
```
List = [34, 17, 92, 53, 76, 45, 61, 29, 88, 12, 5, 68, 41, 20, 97]
Bst = BinarySearchTree(List)
PreorderTraversal(Bst)
```

[ ]:
```
List = [34, 17, 92, 53, 76, 45, 61, 29, 88, 12, 5, 68, 41, 20, 97]
```

3.3 Balanced Binary Search Tree

[ ]:
```
def BalancedBinarySearchTree(List):
    List.sort()
    root = Node(0)
    if len(List) == 0:
```

```
        return None

    if len(List) == 1:
        root.data = List[0]

    if len(List) >= 2:
        mid = len(List)//2
        root.data = List[mid]
        root.left = BalancedBinarySearchTree(List[:mid])
        root.right = BalancedBinarySearchTree(List[mid+1:])
    return root
```

[ ]:
```
Bbst = BalancedBinarySearchTree(List)
```

[ ]:
```
InorderTraversal(Bbst)
```

3.4 Counting the nodes

[ ]:
```
def CountNodes(root, count=0):

    if root == None:
        return count

    count = count + 1 # Counting the root
    count = CountNodes(root.left,count) #   Adding the nodes in the left subtree
    count = CountNodes(root.right,count) #  Adding the nodes of the right␣
 ↪subtree

    return count
```

3.5 Checking if a Binary Seach Tree is Balanced|

[ ]:
```
def Is_Balanced(root):
    if root == None:
        return

    left_nodes = CountNodes(root.left)
    right_nodes = CountNodes(root.right)

    if abs(left_nodes-right_nodes)<= 1:
        return True
    else:
        return False
```

[ ]:
```
Is_Balanced(Bst)
```

3.6 Linear Representation of a Tree

[ ]:
```
def Linear_Tree(root, linear=[[],[]]):
    if root == None:
        return linear
    if root.left != None:
        linear[0].append(1)
    else:
        linear[0].append(0)

    if root.right != None:
        linear[1].append(1)
    else:
        linear[1].append(0)

    linear = Linear_Tree(root.left, linear)
    linear = Linear_Tree(root.right, linear)
    return linear
```

[ ]:
```
def TreeToList(root,list=[],level=0):
    if root == None:
        return
    if len(list)>level:
        list[level].append(root.data)
    else:
        list.append([])
        list[level].append(root.data)

    TreeToList(root.left,list,level+1)
    TreeToList(root.right,list,level+1)

    return list
```

[ ]:
```
def PrintZigZag(root):
    if root==None:
        return

    Lista = TreeToList(root)

    for i in range(len(Lista)):
        if i%2==0:
            print(Lista[i])
        else:
            print(Lista[i][::-1])
```

[ ]:
```
PrintZigZag(Bbst)
```

[ ]:
```

```