Ma. Elvira A. Oco

**BINARY TREE**

In [computer science](http://en.wikipedia.org/wiki/Computer_science), a binary tree is a [tree data structure](http://en.wikipedia.org/wiki/Tree_%28data_structure%29) in which each node has at most two [child nodes](http://en.wikipedia.org/wiki/Child_node), usually distinguished as "left" and "right". Nodes with children are [parent nodes](http://en.wikipedia.org/wiki/Parent_node), and child nodes may contain references to their parents. Outside the tree, there is often a reference to the "root" node (the ancestor of all nodes), if it exists. Any node in the data structure can be reached by starting at root node and repeatedly following references to either the left or right child. A tree which does not have any node other than root node is called a null tree. In a binary tree a degree of every node is maximum two. A tree with 'n' number of nodes has exactly 'n-1' branches or degree.

**TRANSVERSAL**

Compared to [linear data structures](http://en.wikipedia.org/wiki/List_of_data_structures#Linear_data_structures) like [linked lists](http://en.wikipedia.org/wiki/Linked_list) and one dimensional [arrays](http://en.wikipedia.org/wiki/Array_data_structure), which have a canonical method of traversal (namely in linear order), tree structures can be traversed in many different ways. Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps (in no particular order) are: performing an action on the current node (referred to as "visiting" the node), traversing to the left child node, and traversing to the right child node.

Traversing a tree involves iterating (looping) over all nodes in some manner. Because from a given node there is more than one possible next node (it is not a linear data structure), then, assuming sequential computation (not parallel), some nodes must be deferred – stored in some way for later visiting. This is often done via a [stack](http://en.wikipedia.org/wiki/Stack_%28abstract_data_type%29) (FILO) or [queue](http://en.wikipedia.org/wiki/Queue_%28abstract_data_type%29) (FIFO). As a tree is a self-referential (recursively defined) data structure, traversal can naturally be described by [recursion](http://en.wikipedia.org/wiki/Recursion) or, more subtly, [corecursion](http://en.wikipedia.org/wiki/Corecursion), in which case the deferred nodes are stored implicitly – in the case of recursion, in the [call stack](http://en.wikipedia.org/wiki/Call_stack).

The name given to a particular style of traversal comes from the order in which nodes are visited. Most simply, does one go down first (depth-first: first child, then grandchild before second child) or across first (breadth-first: first child, then second child before grandchildren)? Depth-first traversal is further classified by position of the root element with regard to the left and right nodes. Imagine that the left and right nodes are constant in space, then the root node could be placed to the left of the left node (pre-order), between the left and right node (in-order), or to the right of the right node (post-order). There is no equivalent variation in breadth-first traversal – given ordering of children, "breadth-first" is unambiguous.

**INORDER TRANSVERSAL**

It is particularly common to use an inorder traversal on a [binary search tree](http://en.wikipedia.org/wiki/Binary_search_tree) because this will return values from the underlying set in order, according to the comparator that set up the binary search tree (hence the name).

**A binary ordered tree example:**

class CNode:

left , right, data = None, None, 0

def \_\_init\_\_(self, data):

# initializes the data members

self.left = None

self.right = None

self.data = data

class CBOrdTree:

def \_\_init\_\_(self):

# initializes the root member

self.root = None

def addNode(self, data):

# creates a new node and returns it

return CNode(data)

def insert(self, root, data):

# inserts a new data

if root == None:

# if there isn't any data

# adds it and returns

return self.addNode(data)

else:

# enters into the tree

if data <= root.data:

# if the data is less than the stored one

# goes into the left-sub-tree

root.left = self.insert(root.left, data)

else:

# processes the right-sub-tree

root.right = self.insert(root.right, data)

return root

def lookup(self, root, target):

# looks for a value into the tree

if root == None:

return 0

else:

# if it has found it...

if target == root.data:

return 1

else:

if target < root.data:

# left side

return self.lookup(root.left, target)

else:

# right side

return self.lookup(root.right, target)

def minValue(self, root):

# goes down into the left

# arm and returns the last value

while(root.left != None):

root = root.left

return root.data

def maxDepth(self, root):

if root == None:

return 0

else:

# computes the two depths

ldepth = self.maxDepth(root.left)

rdepth = self.maxDepth(root.right)

# returns the appropriate depth

return max(ldepth, rdepth) + 1

def size(self, root):

if root == None:

return 0

else:

return self.size(root.left) + 1 + self.size(root.right)

def printTree(self, root):

# prints the tree path

if root == None:

pass

else:

self.printTree(root.left)

print (root.data)

self.printTree(root.right)

if \_\_name\_\_ == "\_\_main\_\_":

# create the binary tree

BTree = CBOrdTree()

# add the root node

root = BTree.addNode(0)

# ask the user to insert values

for i in range(0, 5):

data=int(input("insert the node value # %d: " % i))

# insert values

BTree.insert(root, data)

print

BTree.printTree(root)

print

data=int(input("insert a value to find: "))

if BTree.lookup(root, data):

print ("found")

else:

print ("not found")

print ("The binary tree minimum value is: %i"%BTree.minValue(root))

print ("The binary tree maximum size is: %i"%BTree.maxDepth(root))

print ("The binary tree size is: %i"%BTree.size(root))