Comparing Object-Oriented Programming and Logic Programming

**Alexander Cahill - 15321711**

# **Object-Oriented Implementation**

For the Object-Oriented(OO) implementation I wrote the program in Python3.6.

The Binary Tree in the OO implementation is made up of two classes:

The BinaryNode class.

The BinaryTree class.

The BinaryNode class is very simple. Each BinaryNode object is initialised with a value (input by the user), a left child and a right child, both of which are set to None.

The BinaryNode class also has a string method for printing but is not used by the program as all of the printing is done by the traversal methods contained in the BinaryTree class.

The BinaryTree class contains the methods for creating the binary tree object and setting it’s root node, inserting nodes, searching for nodes and the traversals. Whenever a new BinaryNode object is created it is done within the BinaryTree class.

Only one tree is created in the program and all methods operate on this tree. When the tree is created it is initialised with a root node with it’s value set to None(indicating that it is empty).

Inserting nodes is done with two methods; “insert()” and “insertNode()”. “insert()” checks if the root node of the tree is empty and creates a new BinaryNode setting it as the root node, if it is.

If the root node is not empty the program calls “insertNode()” which recursively tries to find the suitable place for the input value. I found this easier than using one functions as I had difficulty checking and setting the root node while writing these functions recursively. “insertNode()” simply checks if the current nodes left child is empty if it’s value is less than the value to be inserted and inserts it if it is and vice versa if the value of the current node is larger than the value to be inserted.

The same is true for searching for nodes contained in the tree, “search()” calls “searchForNode()” once on the root of the tree which recursively searches for input value and returns true if it contained in the tree and false if it is not. It does this by checking the current nodes value and calling “searchForNode()” again on its left or right children if it is bigger or smaller than the input value.

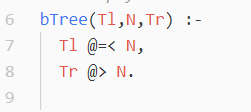
The tree traversals are also implemented recursively by printing each node’s value and appropriately calling the traversal functions again.

# **Logic Implementation**

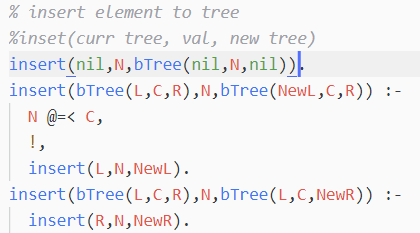
For the Logic Programming Implementation I wrote the program in SWI-Prolog 7.6.4.

In this implementation the binary tree is defined recursively consisting of a left subtree, a value and a right subtree. Each subtree itself is also a binary tree with leaf nodes having ‘nil’ as the value of each of their subtrees. The predicate defining a Binary Tree is as follows.

With “Tl” being the left subtree and “Tr” being the right subtree.



When inserting a new node into the binary tree T the program creates a new binary tree R, this is shown in the following snippet of code where the predicate requires the old tree, the value to be entered and the name of the new tree.



The program recursively searches if a suitable position exists for the node by checking for leaf nodes, if the current node is not empty, greater than or less than the input value then the function is called again.

Searching to see if a value(eg ‘N’) exists in the tree is done by checking if a subtree with value ‘N’ exists at all in the tree. Initially if value ‘N’ is not found a check is done on the current node. If the current node is greater than value ‘N’ then the search function is called on the left subtree and if the current node is less than value ‘N’ then the search function will be called on the right subtree. The program will return True if the subtree with value ‘N’ exists at all, otherwise it will return False.

For each of the traversals Prolog’s ‘append’ predicate is used. Each of the traversals are called on the left and right subtrees but the append functions are used differently to construct the list of visited nodes.

# **Comparison of Implementations**

How the binary trees are represented in both implementations differ greatly.

In the OO implementation both the binary tree and node are objects; instances of their given class which encapsulate their own functions and variables. The binary tree being a collection of object. The binary node being an object with a value and what are essentially pointers to it’s children.

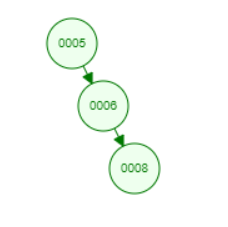
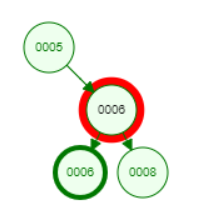
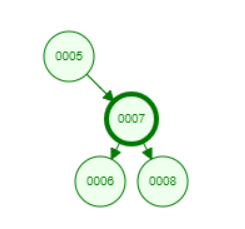
This is in contrast to the Logical implementation where the binary tree is recursively defined as a relation between a value and left and right subtrees. The nodes, by the definition of the tree in the program, are also binary trees.

In my opinion the OO representation of the binary tree is easier to understand at first glance, easier to visualise and to implement as the layer of abstraction and encapsulation allows you to see what the classes are made up of and how they operate.

Though the logic behind how most of the functions or predicates operate is very similar, with nearly all of them being implemented recursively, some of the predicates in Prolog can be defined much more concisely due to its relational nature.

Currently neither implementations have a way to remove nodes. The logic behind implementing this would be similar for both programs. For leaf nodes would be simple enough ; make the parent’s left or right child node equal to None or nil. For example if a node with value 2 had a child node that was also a leaf with the value 3, just make 2’s child node equal to None.

Removing a node that is not a leaf node is trickier. You would have to know the node’s parents and children and determine which node will “take it’s place”. For example consider the tree below.

To remove node 7 one could change node 7’s value to the value one of it’s children(left child in this case) then make it’s left child’s value None/nil. In the OO implementation this is a matter of changing the value and, left and right pointers of nodes.

In the Logical implementation this is accomplished by changing the relation between the parent and the concerned subtree. In essence this removes node as there is no link to it but the node/subtree will still exist in some form for some time in both of the implementations after it is removed from the tree.