**Comparing Object-Oriented Programming and Logic Programming**

**Description of the object-oriented programming implementation**

My Object-oriented implementation of the Binary Search Tree structure was created with python 3. Within this implementation, a Binary Search Tree is a tree data structure where each node within a tree can have at most two children, referred to as left and right, with those on the left being less than and the right begin greater than. This way, each node is stored in a certain order in the tree.

Upon first Implementing this version, I created a Node class to represent a single tree node within a given tree. Within it, consists of 3 attributes, those being the left child node, the right child node and the data value of that particular node itself.

Following that, I implemented the BinarySearchTree class, to model the collection of Node objects. This class takes the root as a parameter upon  , optionally. It references the Node class and is required for all it’s methods to work. The methods: insert, search, preorder, inorder and postorder; all belong to this class.

For any one node in a tree, its value must be greater than the nodes within the left subtree and less than those in the right. And so, those two conditions are applied when inserting a new node to the tree through the use of comparison operators. However it first checks for if the tree is empty or not, if it is empty then the node to be added becomes the root, the start of the tree. All mentioned ensures that the binary search tree property is maintained correctly.

For both the traversing and search methods, it uses double recursion for the left and right subtree when carrying out the searching process of each node concurrently. This also applies within the prolog implementation too.

At first, a problem within my code for the Insert method occurred when a duplicate item was added, this caused an error as none of my conditions were suitable to cater it and check for equality. With that I adapted it so when this situation may arise, the duplicate will be added to the right subtree of its matching node. This is the same logic applied in my prolog insert operation. However, this is still not a completely efficient solution either. Replacing the item or to not insert if already exist in the tree could possibly be better alternatives. In addition to that, my insert method does not use recursion, it would be far more efficient if so.

Traversing the tree is simply just the way in which the node values can be retrieved in a given order. Upon traversing, the methods must always start at the root. Even within the search operation, the root must be the starting point for all. The three different traversal methods in my code are implemented recursively and are essentially the same, being the only difference the order of the instructions within the 3 functions, all must retrieve and output the nodes. If these methods were not implemented recursively, it would have search and return each node iteratively. By using recursion, it can continue executing from a given stage within the tree, than having to restart the process and can be done in place.

**Description of the logic programming implementation**

My Logic programming implementation of the Binary Search Tree was created using Prolog.

The binary tree is represented in prolog through using a recursive structure. For the definition of the tree itself, it can either be an empty tree, the empty tree is denoted by the symbol nil, else it requires three arguments, the value, the left subtree and the right children. Those subtrees must be binary trees too.

Before inserting a new node into the tree, the predicate must search for the correct position of where this new node can be placed. It will continue backtracking until it eventually reaches a nil and when it does, the new node will be created at that point.

Within this implementation, the tree traversal operations produce the same output as the OO version does, it is simply just the way in which these operations are carried out. Like OO, there are also three possible modes of traversing the tree within this, those being; preorder, inorder and postorder.

For inorder, it processes the left subtree first, this is followed by the root and then the right subtree. So this will create all the values within the left subtree to appear first in the outputted list.

The same idea applies to both post order and pre order too. Rather, with preorder the root is processed first, then the left and lastly the right. With post order, it goes in the order of the left subtree first, then right and the root. These operations follow the same logic in both versions.

As mentioned in the Object-Oriented implementation, upon implementing a search for an item or where to place a new item, the prolog implementation also invokes double recursion.

The way in which the prolog language manipulates the binary tree is similar to how it would with lists. Upon the definition of the binary tree, when the tree isn't empty, it uses recursion to call on both the left and right subtrees.

**Analytical comparison of the two implementations**

Both Logic and Object-Oriented are two completely different programming paradigms, with the most obvious difference being the implementations themselves. With both Python and Prolog, the order is stated explicitly. However, with Python it is represented as a list of instructions and with Prolog, it displays rules to evaluate, both causing the same behaviour in the end.

In contrast to Python, [Prolog](http://en.wikipedia.org/wiki/Prolog) is a logic and declarative programming language, in which it state facts and rules to derive new facts through the use of relationships and recursive backtracking searches. Prolog uses a built in recursive backtracking search internally within it. That is the reason for the simplicity and shortness of the prolog code that is required, in comparison to the Object-Oriented version.

Upon writing in Prolog, algorithms like Binary Search Tree, aren’t specified in same way as it would with a conventional programming language such as Python and when using the Object-Oriented paradigm. Prolog is more descriptive for each step, its approach is concerned with known facts and relationships based upon those facts within a given program. In contrast to this, the Object-Oriented implementation consists of a sequences of steps within each method in which the computer must do. Similarly, in both the object-oriented implementation and prolog, they are concerned with objects and the relationship between those objects. However, the way in which these objects are described and referred to differs entirely. Prolog doesn’t invoke object variables or inheritance, etc. Within Object-oriented, an object is a data structure, an instance of a class that inherits the various methods and variables from the class.

In terms of the code implementations, prolog is far more simplistic.

However, the implementation of my tree traversals and search methods within OO are recursive like the logic implementation, being as it was the most efficient way of carrying out these operations and reduce the amount of redundant code.

If a deletion operation of a node was to be added to the pre-existing code of both implementations, there would be three cases that would needed to be considered for it and handled. Essentially, this operation would perform quite similarly to the insert operation. First of all, a search would have to be carried out to first locate the node. The first case then would be to test for whether the node to be removed is a leaf node, a node with no children. Here another method would need to be created to examine if that node is a leaf or not. Secondly, the given node could perhaps be a node with one child. Here, a possible solution could be to just replace the child as the new parent and remove the old parent. Not just that, but the child would firstly need to be examined to see if it belong to the right or left subtree of the parent. Lastly, for a node with two children, a way in which this could be carried out could be to find a sufficient node to replace the parent node to be removed. A way in which this is typically done is to find the next largest node to it within the right subtree and chose that to be the one to replace it.