**A Comparison between an Object Oriented implementation of a Binary Search Tree and a Logical implementation.**

**Object Oriented Implementation**

My object-oriented implementation of a Binary Search Tree, which uses Python 3, is a single file - binarySearchTree.py - and it uses two classes - Node and BST. These classes make storing and testing, in my opinion, a lot easier for a variety of reasons that I will explain throughout this comparison.

The Node class allows us to initialize Nodes - an integral part of making a binary search tree. This class initialises when you pass it a root, and optionally a left and a right. Left and right are the left and right children of a particular node. By default if you don’t pass any parameters for left and right, they both default to None - and the node is a ‘leaf node’.

BST is where the binary search tree is actually built and maintained. When initialized, it will set the root to none. This allows us to add our own root. To do this, we use the add function - which takes in an item, and calls recursive\_add, using the root as the pointer. This function takes the item, and compares it to the ptr. If item is smaller than the pointer, it will call recursive\_add with the left child of the pointer, or ptr.left. If the item is larger, it will call recursive\_add with the right child of the pointer, or ptr.right. When the pointer is None, it will insert the item, initializing it as a Node. It will then return the pointer.

Another function the BST class allows us to implement is the search function. The search function allows us to input a number to check if it is present in the binary search tree. This function then calls recursive\_search, which will allow us to check for the number in the binary search tree, again using the root as a pointer. If the pointer is equal to None, this function will return False, as the number is not in the binary search tree. Likewise, if the pointer is equal to the number, it will return True, as the item is obviously in the binary search tree. As with the recursive\_add function, it will then compare the number to the pointer. If it is smaller, the recursive\_search function will be called again with prt.left as the pointer. If it is larger, it will be called with prt.right as the pointer. This continues until either the value is found, or the pointer is equal to none.

The three ‘printing’ functions are also done recursively, and in very similar ways. The main difference between these three functions is the order in which the nodes are printed and the recursive functions are called. For preorder calling print\_pre\_order will call the recursive function. The recursive function will first check if the pointer is equal to None. If so, it will pass. After this, it will first print the item, then it will call recursive\_print\_pre\_order for the left child of the pointer, ptr.left, and then will call recursive\_print\_pre\_order for the right child of the pointer, ptr.right. This means that item will be printed first, then everything smaller, and then everything larger.

The print\_in\_order function works similarly, calling a recursive function. This function, recursive\_print\_in\_order, will also check if the pointer is equal to none, again passing. After this it will call recursive\_print\_in\_order on the left child, then it will print the item in question, and then will call recursive\_print\_in\_order on the right child. This means that the items in the binary search tree will be printed in numerical order.

The print\_post\_order function works extremely similarly. The recursive\_print\_post\_order function calls recursive\_print\_post\_order on the left child, and then on the right child, and then prints the item it was called with - smaller than root printed first, then larger than root, and root itself.

A delete function could be implemented by keeping track of the parent node of a given node, and when the node is equal to a user input, changing the node to the parent node and recursively doing this for the entire tree.

**Logical Implementation**

For the logical implementation of a Binary Search Tree, I used Prolog. The logic behind the prolog implementation is quite similar to the object oriented implementation, but is carried out far more concisely by Prolog.

In this implementation, the node is a triple which can contain any three things - but for my purposes we use only integers, nodes and ‘nil’. These nodes are how the tree is built in this implementation. A tree is made up of nodes which can have nodes as children. An example of this is node(5,node(4, node( 2, nil, nil)), node(6,nil ,nil)). In my implementation, this is how you would call a tree.

This implementation does insert in an interesting way. If the value you’re trying to insert is the value in the first position of node, it cuts, as there is no need to insert a value which is already in the tree. If the node is (nil, nil, nil), that is, a tree initialised but with nothing in it, it will put the value in as the first nil, and keep the remaining two values as nil. For cases where the root node has child nodes, it will check the value against the first member of the root node - if it is greater, it will call insert on the right child, or the third member of the node. If it is smaller, it will call insert on the left child, or the second member of the node. This works recursively, allowing the value to be inserted to the correct position in the tree.

Search works recursively as well, returning true if the first member of node is the value currently being searched. If it isn’t, and the value we are searching for is less than the value of the first item in node, then search is called with the value we’re searching for and the second item in the node, which will either be another node or nil. If it’s nil, then it’s false - the value isn’t in the binary search tree. If not, and it’s not equal, then search is called again. If the value of the item we are searching for is greater than the value of the first item in the node, search is called with the third item in the node, which represents the right side of the binary search tree. It then works in much the same way.

Listing the nodes inorder is done recursively. If there is nothing in the binary search tree, or it is just ‘nil’, the predicate cuts. If only the first item of a node has a value, and the other two items are ‘nil’, the first item is written, followed by a space, and then the predicate cuts. If the first two items of a node have values, and the third is nil, then first inorder is called with the second value of the node, then the first value is written, followed by a space. When the first and third item in a node have values, and the second is nil, the value of the first item is written, followed a space, and then inorder is called on the third item. When all three have values that aren’t nil, inorder is called on the the second item first, then the first item is printed, followed by a space, then finally inorder is called on the third item. This will print it in numerical order.

Listing the nodes in preorder works very similarly to listing them inorder. Again, the two base cases are the same as in inorder, but after these, they start to differ. When preorder is called with only the initial value and the second value, with the third value being ‘nil’, it first writes the first value, then a space, and then will call preorder on the second value. The same happens when preorder is called on a node which has a value which is not ‘nil’ in the first and third places, but the first value is written, then a space, and then preorder is called on the third value. When all three values are provided, it will write the first value, then a space, then call preorder on the second value and finally preorder.

Postorder, again, works very similarly to the above. The two base cases are the same, but after this there are slight differences. When postorder is called with only the initial value and the second value, with the third being nil, it will call postorder on the second value, then write the first value, and then write a space. If postorder is called with only the initial value and the third value defined, and the second value as ‘nil’, it’ll call postorder on the third value, write the first value and then write a space. When called with all three values initialised, it will call postorder on the second value, then call postorder on the first value, and then write both the value and a space.

Delete is the inverse of the insert, which means that we essentially get it for free when we define insert.

**Comparison**

Both of these implementations have their merits and their faults. The Object Oriented implementation is, in my opinion, easier to understand, allowing users to understand how the code is being added to, or seeing exactly why the program works the way it does. The prolog approach is, in comparison, much more confusing for the user, I believe, making it difficult for new users to understand exactly what the program is doing in each predicate.

This being said, the logic approach, specifically when done in Prolog, cuts down on the amount of work that actually has to be done. When one a predicate is defined, we also get the reverse of the predicate for free. So in this, in the case of this binary search tree implementation, as we have already implemented the insert function, we essentially get the delete function for free. In contrast, the delete function in the Object Oriented implementation requires a lot more to be carried out. It would require a whole new function, which we would have to think through and write carefully.

The logical approach that I took with this program allowed every predicate to cut once it’s required outcome was reached. This reduces the amount of time that the program takes to run, and reduces the amount of memory that the program needs to run. This is achieved through the use of cuts - ‘!’ - in every predicate. This is also possible in the object oriented implementation in Python, through the use of breaks. I, however, used pass, so the program continues to go through everything, but just does nothing in certain situations. This is not an optimal implementation, obviously.

The logical implementation is also shorter than it’s object oriented counterpart. I believe that this truly shows the power Prolog - a few short predicates can carry out in 30 lines of code what takes 80 in Python. Neither of these implementations are perfect, but both have their merits. The object oriented implementation is more user-friendly, while the logical implementation runs more optimally. In the end, both of these methods work well, and it truly comes down to which the developer works better in. An object-oriented approach would probably suit me better, as I have more experience working with python and the like. However, the logical approach is probably a better choice in general.