Comparing Object-Oriented Programming and Logic Programming

The aim of the second assignment was to implement a binary search tree data structure in both the object-oriented and logic programming paradigm. The languages chosen to to do this assignment were Python3.6 for the object-oriented approach and prolog for the logic programming approach.

The implementation of the BST began with the Python object-oriented paradigm. Before any programming commenced, planning and foresight was needed. Firstly the features of a binary search tree had to be established so the appropriate classes could be designed. The classes that were chosen to do this were that of a Node class since a tree is essentially a collection of nodes and a BinaryTree class to encapsulate these nodes. Each node of a BST has three primary qualities, the data which gets passed on instantiation of a new node, this is an integer in our case, and left and right pointers which default to None. With these three fields we can create our BST.
Next step was to define the BinaryTree class which would nicely encapsulate all of our soon to be created nodes and methods. In our newly defined class there will be one field, the root, since a tree cannot exist without a root. This root is initially set to None but later be used for holding our first new Node that we insert into the tree, therefore we can already see that composition will be heavily used in the BST as the whole data structure relies on this feature of object-oriented programming. There was a bit of conflict in regard to if the root Node should be passed to the BinaryTree class when instantiating a new tree but with further planning it was obvious that if we attempted to insert a new Node into the tree and the root field was still set to None then it will become the root.

Next line of work was to implement all the methods required and specified in the assignment specification which were insert, search and the three tree traversal methods (preorder, inorder, postorder). When planning out how to implement the insert method a bit of a paradox occured. If we were to insert a Node that carried the same data as an already existing Node what would happen? Well it would be inserted to the right of the already existing node but if we tried to traverse the tree it would never be visited, so we needed a search method to ensure that an incoming nodes data didn't already exist but we can't search a tree that has nothing in it.
The first method to be implemented was insert but for the reason stated above it couldn't be fully implemented until a later stage.
The insert method would take two arguments, the data we wanted to insert into our tree and node which will represent the current node of a recursive call to the function.
We started by checking that the trees root existed, like previously said if there is no root then the insert method would insert the root node. If the root node existed we had to do a comparison check, one of the main defining features of a BST, if the data being inserted into the tree was less than the current nodes data we'd check the left pointer of the node, if that left pointer pointed to nothing we'd insert our new Node there, otherwise if there was another Node to the left we'd have to repeat the entire process of the insert method up to this point but this time with the node that was on the left.
We'd repeat the exact same process for the opposite, if the data was larger than the current node we'd check the right pointer and if that was empty we'd insert it otherwise we'd call the method again but this time we'd pass the Node to the right in as an argument.

N.B. we'd pass in data to the method but once it is ready to be inserted into the correct position we'd create a new Node and pass the data to that object.
Moving onto the insert method we pass the same arguments again relying heavily on recursion to traverse the tree in the same manner as the insert method using the same logic, if the data we're looking for is equal to the node we're currently pointing at return that data else if it is smaller than that data, call the search method again and pass it the data we're looking for along with the new node. If the data that we're looking for is greater than the nodes data we're currently looking at go right and check if the data we're looking for is equal to the node on the right. This process will continue until either the number is found or if we hit the bottom of a tree designated by the left or right pointers pointing to None.
Once our search method was fully implemented and tested we could add our final 'if' block to our insert method ensuring that not duplicates can exist in our tree making sure our later traversal methods will be able to visit every single node.
Finally there was the three tree traversals, inorder, preorder and postorder, for the sake of continuity and not repeating myself I will go into depth of the preorder method and briefly describe inorder and postorder as the content of each method is the same but are procedurally different.
With the preorder method due to the structure of the method we can see that the 'print(node.data)' is placed before any recursive call meaning our root is outputted straight away. Then we have a recursive call that continuously goes down the far left hand side of the tree on the way printing out the data of each node visited, once the far left node is visited that has no children it works its way back up the stack and the recursive call on the right is called printing out the greater than children of the less than parents. This exact same process is repeated on the right hand side. All out preorder method is doing is traversing the entire tree, where the actual preordering of the output is coming from is the location of the print statement, based on whether it is before between or after the recursive calls as the recursive stack begins to work its way back up and pop entries off the output is affected.
When implementing a delete method in a bst there are four cases to be aware of. Firstly does the Node exist, if not do nothing. The Node has no children, if this is the case then we can simply make the parent point to None therefore instead of actually deleting the Node we just make it get lost in memory. If the Node has one child then we can take an approach very similar to a linked list deletion, rather than just referencing None we can take the value being pointed at by the Node we want to delete whether it be None or another Node and we can make the parent of the to be deleted Node point at that reference, again not actually deleting the Node but de-referencing it. Finally the the Node has two children. This one is bit more complex. Unlike the cases previously described reordering must occur here. If a node that has two children is removed we must account if the two children have children, if so we must find a new node that can replace the deleted node whilst keeping the tree sorted.
Due to how a BST handles the splitting of data based on it's whether the Nodes data is greater than or less than the root and other nodes its efficiency is quite fast especially with search, insertion and deletion where they are at best O(log(n)) and at worst are O(n log(n)).

Moving onto our second implementation of the binary search tree using a Prolog for our logic programming language.
When implementing the BST in Prolog a lot more planning was required in comparison to the object-oriented approach primarily because of lack of experience with Prolog. A Prolog script or database consists of a collection of horn clauses, variables exist but not outside the scope of facts and rules therefore there is no overall state that we can manipulate and change and all we're left

with are facts, rules and clauses. Therefore a completely new thought process and approach were needed. Since Prolog heavily relies on relations we could inherently approach our problem by rather than asking it to build a BST we could ask to evaluate our input and check if it is in fact a BST. With this mindset and approach we were ready to begin implementing our program.

Insert was the first set of rules to be implemented. Our first rule deals with the empty tree, 'insert(D, nil, t(D,nil,nil)). This rule will seems very redundant at the start as it simply checks if D is equal to the root of the tree, both of which values we'd have to pass in ourself but through the power of Prologs relations we can swap this query from being 'insert(10, nil, t(10,nil,nil)).' which will return true to 'insert(10, nil, X).' which will return X = t(10, nil,nil). Using this new frame of mind we can simply replicate a BST in prolog.

Our next two rules will deal with a non-empty tree. Each rule will take three arguments, data we want to insert, the current existing tree and the output tree but as we said earlier we can replace this tree with a variable X to query the rule so it will return the data structure to satisfy it to true. The two rules do the same thing except in different directions. The first rule for the non-empty tree checks if the data is less than the current root, if so the rule executes the recursively constantly passing in the left node as the new root and update the new output that would satisfy the rule as it goes along. The second rule does the opposite the the case of the comparison and direction. A key thing to notice is that the base case for the recursion is defined as the very first rule, this shows us that when we call insert recursively it checks all three rules we have starting at the empty tree to the recurse right.

Next we have our search rules our first rule acts as our base case and shows the simplest outcome to the problem, which is if the first node visited is the value we're searching for. The next two rules follow the same approach as the insert rules but instead replacing our second argument with a variable we are simply checking to see if the data is in the tree if so it evaluates true otherwise false. However you can insert a variable into the tree and the search rules will act very similarly to the insert rules just with a less user friendly output.

For the tree traversal in the logic paradigm it is essentially more of the same thing therefore I will speak in depth on one of the traversals and simply state how the others differ very little from it. Much like all the previous rules postorder tree traversal consists of three main rules, base case and two traversal rules both of which go in opposite directions. Our first postorder rule ignores any input besides the left node it then recursively goes down the entirety of the left hand side of the tree till it hits the node X which has no children. Once this occurs our base case which is the last rule in our trio (with good cause) catches this and displays the root of the subtree that has no children. Since our recursion has hit the base case it begins to return up the stack and hits the next option other than the base case which is to recurse down the right hand child of the left hand side of the tree. This occurs until all nodes with no children are hit and the each time this happens the base case is hit since it is at the end of the postorder rules and this process occurs all the way up the left hand side of the tree and the same process occurs on the right hand side of the tree. The output of each traversal is determined by where the base case rule is placed within the set of rules for each traversal.

To implement the delete set of rules we have to consider three prime cases, one if the node has no children, two if the node has at most one child and three if the node has two children.

For the first case we could make use of the previously defined search rules and use the cut operator as to drop the node with no children. Next we'd need a get parent so we could search for

the node needing to be deleted stop one before it and make the child of the node to be deleted the child of the parent rather than it’s child.
Finally for the final case as there is no simple answer springing to mind a solution could be to implement a get subtree rule which returns a subtree with the to be deleted node as the root. We could then traverse this subtree using our inorder rules and whilst excluding the root this could be done using the get parent rule previously defined, once we have an array of all values in the tree we could delete the root node in our original tree. Since we now have our array of values if we remove the/ 2 index and insert that into the tree then insert the rest of the list from start to finish using our rule the tree should remain balanced.As prolog uses relations quite heavily its efficiency is affected quite heavily due to its expensive nature.

After both implementations we can clearly see that both paradigms differ immensely especially in the approach and mind set required. In regard of comparing both paradigms the place to begin with is the structure of both programs. With the OOP implementation we made use of classes and and methods within those classes to instantiate new objects and perform operation on those objects however with the logic implementation we don’t have the luxury of objects or methods instead we just have a database full of facts and rules with structures being the most similar comparison between both languages. In our OOP implementation once we instantiated our tree we’d perform operations through method calling on the tree to update it, we could create new tree objects and perform the same methods on them changing the state of the program however we did not have this luxury with Prolog instead of being able to create a tree we must test for a tree. Even though our logic implementation was far shorter than the OOP implementation, a lot more thinking was required due to lack of experience with the paradigm. Once we’d define the rules that would successfully check if a tree inputted was in fact a binary search tree, unlike mapping (one to many) which our OOP paradigm uses our prolog implementation as stated earlier makes heavy use of relations (many to many). These relations can be very powerful but quite expensive. By using these relations we could query our program by passing in a BST and asking with where a node would be inserted and it would procedurally go through the program much like how the OOP implementation would and find the suitable rules that matched the input.Prolog also made use of backtracking, backtracking occurs when the logic program executes a rule that leads to a false outcome rather than deeming the overall query as false it proceeds to go back to the previous state and find an alternative solution where possible. Where the OOP implementation made use of common OOP traits such as fields, methods and composition the logical implementation made use of something similar to Haskell's pattern matching, relations and backtracking.