Name: Samuel Wood

Student Number: 16476066

CA341 assignment 2

*Comparing Object-orientated programming and Logic programming*

**Object-orientated approach**

*Classes*

The Object-orientated implementation has two classes used for the implementation of a binary search tree. The first is the Node class. The variable components of this Node class are “value”, “left” and “right”. The constructor or “\_\_init\_\_” method initialises the new member of it’s class. The node value is definitely required if the Node is made and the left and right attributes are default set to None. This is because when the Node is created at first, there is initially no children.

*Insert*

To insert a method into a tree requires the use of two methods, “insert\_rec” and “insert”. The insert method calls the recursive “insert\_rec” method to recursively add the value to its tree. It checks to see if the value being added is greater or less than the value or values already in the tree to help find its new location.

*Search*

The search operation is also split into two functions. They are “rec\_search” and “search”. If the value being searched for is equal to the value of the pointer in the Node True is returned. The function then recursively moves through the tree by repetitively calling the “rec\_search” function, taking the left pointer and the right pointers as their parameters along with the value that is being searched for.

*Preorder*

My view of the preorder traversal in a binary tree is that it travels in the sequence ‘visit, left , right’. For example, from the very first node which is firstly visited, it will travel left down the tree, visit that node and continue to travel down nodes by moving left until it can’t move left any more. In this case it will move right, Hence ‘visit, left, right’. In other words, if a node is arrived at it will visit it, move to its left child and if moving left has no more part to play it will move to its right child where the same sequence will begin. Similar to the insert methods, the way to view the Nodes in preorder traversal is split into 2 methods. The “pre\_order”  function calls “pre” function which doesn’t end until the point attribute is equal to ‘None’. In this case there will be no more nodes to print. The lines of the “pre” function “print(point.value)”, “self.pre(point.left)”, “self.pre(point.right)” resemble the sequence “visit, left, right”.

*Inorder*

My view of the inorder traversal in a binary tree is that it travels in the sequence ‘left, visit, right’. For example, beginning at the very first Node, the leftmost child will be visited first. Next it travels child’s right child but from there we will move left if possible, then until we can no longer move left, visit the current node and then move right. This operation is split into 2 methods “in\_o” and “in\_order”. The “in\_order” function calls “in\_o” which doesn’t end until the point attribute is equal to ‘None’. In this case there will be no more Nodes to print. The lines “self.in\_o(point.left)”, “print(point.value)” and “self.in\_o(point.right)” in the “in\_o” function resemble the sequence “left, visit, right”.

*Postorder*

My view of the postorder traversal in a binary tree is that it travels in the sequence “left, right, visit”. For example, beginning at the very first Node, we will move to the leftmost child first and from there the right most until we actually visit the a node. Like the other two traversals, they are split into 2 methods, “post” and “post\_order”. The postorder function calls the post function and doesn’t stop running until the point attribute is equal to “None”, the same as the other two traversals. The lines “self.post(point.left)” , “self.post(point.right)” and “print(point.value)” in the “post” function resemble the sequence “left, right, visit”.

*Remove*

If I was to implement a delete function in the code I would do it in a similar way to the other functions. Using my previous functions as a guide I would construct 2 recursive functions that would iterate through the nodes of the tree. The search function would have a huge part to play in this function, by enabling it to find the function that is to be removed.

**Logic approach**

*Insert*

In the logic implementation of the programme, there are 2 methods to insert values into the tree or create a tree. These 2 methods are ‘insert1’ and ‘insert’. The insert method takes the nodes to be placed in the tree, the old tree and the new tree. The first of the nodes run in the insert function or if there is only one is placed in the first position of the ‘insert1’ function. The insert then recursively designs the tree by comparing the values of the nodes.

*Search*

The search method in the logic approach takes 2 parameters, a number to search for amongst the nodes of the tree and the tree itself. The base case is used to terminate the function when the number N is found in the tree. If the value N is not equal to the value V in the tree it will search for it in the 2 other variables in the tree by calling itself.

**Comparison**

*Insert*

One thing that both implementations have in common is that both operations use two methods. Both of which are recursive. Since the use of both implementations is recursive both implementations run smoothly. Both implementations require some amount of logic in their code to properly insert a the node into the tree.

*Search*

The search implementations in the Object-orientated approach uses 2 methods, unlike the logic which only uses one. Both implementations are recursive, both get terminated when the base case is reached. One good thing is that the logic version has much less lines than the object orientated.

The above is my analysis on an object\_orienated binary search tree with the operations “insert” ,“search”, “preorder”, “inorder” ,“postorder” and the operation “delete” is also discussed, compared to a logic implementation with “insert” and “search” operations.