Binary Tree - CA341

# Object-oriented implementation

## I chose to do this in Python.

* For my OO implementation I began by creating the Node class which includes a constructor method which defined the value of the node and the values for its right and left children.
* Then I created the Binary tree class ‘Btree’. This class includes a constructor method that defines the value of the root of the tree which then is converted into a Node():
* The first function I created for the binary tree is the insert method, this simply takes a value and inserts it into the binary tree. It does this by first checking if the root is not None if it is None the value the user supplies becomes the root. Otherwise the method proceeds to look for the correct place to insert the value this is done through a series of Boolean operations which check if the value is larger or smaller than the root. The method checks every element recursively this way replacing the root passed to it as it goes along when it finds a root that doesn’t have left/right child in the appropriate place it will place the new value there. There is also a helper function called call\_insert() which passes the root of the tree to the insert method so that the user only needs to pass the value.
* The next three functions are the traversal algorithms. The first one in-order traversal this is the inorder() method it requires the root node to be passed to it along with a list which the program does. This will then produce the in-order traversing algorithm as a list. The method first checks if the root is None or not and then proceeds to first add all the left side of the tree recursively then the tree root node then recursively appends the right side.
* The next algorithm is pre order traversal and is done through the preorder() method this is nearly identical to the inorder() method due to the power of recursion as the operation to recurse through the tree is identical only the order in which the method visits the nodes change. This time the method first appends the root then appends the left side of the tree then the right side.
* The last algorithm is post-order traversal and is done through the postorder() method. Like the two before check if the root is not None then follows the rules of the post-order algorithm print the left side of the tree, the right side of the tree and finally the root node.
* Due to the power of recursion only minimal changes are required to code the 3 flavours of traversal algorithms which increases efficiency and makes the code much clearer to read and understand.
* The last method in the OO implementation is the search() method. This method when supplied with the root and the value (x) the user is looking for will recursively go through the tree and prints True or False depending on whether the value exists in the Binary tree. If the root value is equal to x the method returns True otherwise it recursively searches through all the branches checking each Node until either the value is found, and the method returns True else the method will return False. The root node passed to the method changes and this allows the method to search deeper into the tree.

# Logic programming implementation

### (I was not able to fully complete the logic programming implementation in time for the dead line, I was only able to code small sections of the program. However, I will give my best attempt at explaining how I would have implemented a binary search tree with the required functions in Prolog with code examples where possible.)

* I began by creating the recursive tree structure and give it three arguments, the key of the root, the right subtree and the left which both follow the same rules as the parent tree. And the empty tree denoted with nil ‘emptyTree :- nil.’.
* The three traversal algorithms inorder, preorder and postorder only change depending on which nodes are passed first. Similar to the OO implementation due to the power and beauty of recursion the three implementations are extremely similar, and this is the same in prolog. The in-order traversal order is the left sub tree, the root then the right sub tree. In prolog the function recursively calls the binary tree and all of its left and right sub trees until all nodes have been visited and appends them to a list *‘ inorder(tree(Y, Left, Right) Nodes)’* then the left Nodes of the left side are visited *‘inorder(Left, Left Nodes)’* then the right Nodes of the right side *‘inorder(Right, Right Nodes)*’. All 3 of the algorithm implementation in prolog follow this structure only changing in the order of the nodes visited.
* Next I created the search function. The implementation is very similar to the one for OO in python as recursion creates very similar code. The function first checks if the current node is equal to the value we are searching for if not it will continue and work through the nodes from left subtree to right and will continue until it either has found the Node or a value *‘nil’* is reached.
* Finally the insert functions inserts each key into the prolog binary tree as a leaf Node. However before doing this it first searches for the appropriate position, when the function reaches a *‘nil’* it will insert the node.

# Remove element operation

* The remove element operation is slightly more complex than the previous functions discussed in this report. However, by breaking it down into 3 separate cases it can be clearly described, and the method is similar in both the logic programming implementation and the object oriented one.
* If the node to be deleted is a leaf, then the node can be simply removed from the Binary tree without affecting the overall structure this is the simplest remove case as the order and structure do not need to alter.
* If the node to be removed has one child only then the child node will be copied to the parent node and the child node will be removed. Since only one value this is still quite simple to remove as it’s the extremity of the binary tree and still doesn’t greatly affect the over all structure.
* Finally, if the node to be removed has 2 children we will need to restructure some or all the binary tree. First, we will need to perform the in-order traversal on the Binary tree and from the output extract the in-order successor this can be done with the functions defined in the above implementations. Then the in-order successor will be coped to the node that is to be removed and then remove the in-order success resulting in a new binary tree that is still balance with the element the user chose removed.

# Analytical comparison

* The comparison between the two paradigms logic programming and object-oriented programming can be summarised by saying that object-oriented approach provides flexibility but brings in some complexity however the logic paradigm provides simplicity and hides complexity, but it might now solve the problem.
* For the OO implementation the program specifies the exact steps and procedure for the code to follow telling the program how to for example perform in order traversal or insert an item in to the Binary tree whereas the logic paradigm only requires the end goal and will execute the program and produce the output.
* The way in which logic programming operates means its quite strict and rigid in the way it handles and process data and provides very little flexibility, OO however leaves it up to the user do define the instructions clearly. This could create problems if the user is unsure of how to reach the goal this is where prolog outshines python as the user does not need to know every step only the goal.
* Classes in OO implementation provide a very big advantage over the logic implementation as they provide much higher security and robustness. The programmer can choose to hide certain elements of the Binary tree in this assignment if they wish to increase security and call the classes from separate files and classes.