Comparing Object-Oriented and Imperative Programming Language Performance With a Binary Tree Algorithm

Introduction

An *object-oriented* approach to programming includes creating smaller subproblems called ‘objects’, to which you assign some data. The program will follow a *bottom-up* approach and can easily change / add / remove the data attached to the object. The data inside the object is typically considered more significant than the object / function itself, making it a more secure method of programming.

An *imperative* or procedural approach to programming acts by dividing segments of code into separators in the form of functions. In an opposite fashion to OO programs, it follows a *top-down approach* in such programs it is much trickier to access the data inside the functions in order to alter, add or remove it. Once again in an opposite fashion to OO programming, the function is more significant than the data attached to it.

It is clear from the differences above that these languages will act oppositely when executing a program.

A *binary search* tree program works as follows:
Because the tree is *binary*, each node or point on the tree that we will visit can only have a maximum of 2 children, so, the node will either have zerochildren or two children.

For my Object-Oriented language, I have chosen Python.

For my Imperative / Procedural language, I have chosen C.

I have experience working with these languages, which is mainly why I chose them. However, I felt that they were well known for the titles I would be operating them under.

Object Oriented Approach

As seen in my code, my Python implementation of a binary search tree is split into three main parts; the Node class, the Name class and the Phone class. As the name suggests, the Node class is the part responsible for the creation of nodes; a node is a point on the tree that will represent the person; their name, phone number and address. We create nodes so we can easily add, delete and modify the data attached to it. In this class we simply initialize the nodes into existence.
In the following Name class, we can create, add, remove and search names in the tree. The class contains 6 functions, to create, add, remove and search for a supplied name, and one to iterate through the tree in order so that at any instance of modification (add, remove or search) , the tree will remain in order and not just be dealt with in a random fashion.

My Phone class takes on a very similar structure, containing functions to create, add, remove and search for a supplied phone number. Once again the Phone class has a function to keep the order of the tree in check.
Lastly, I included a main function containing test data which can be used to test the classes functions explained above.

Each function can be easily accessed from any point in the program, from other classes outside it's own. This is due to the objects inside classes having the same variable components and operations. Inheritance is an example of this; where we can create a new class that carries over methods and properties of another pre-existing class without much modification to the existing class.
To refer to my code, in my Node class I initialized my nodes, which I can use and modify in the following classes without many limitations.

This approach was not too difficult to implement and is structured in an organized fashion, easily making sense to the reader.

Imperative Approach

From my implementation of the binary search tree in C, you can see that rather than having classes to separate my data declarations, I have a selection of different functions not separated by anything which enable initialization, search, add and delete.

Three nodes are created, along with the left and right children at initialization.
The functions carry out the same duties as the ones in my Python implementation, however by using pointers. Pointers are variables that store memory addresses, which enables the moving and tracking of the data attached to the node in question. The pointer allows us to move through the tree, going from node to node as it stores the memory address of each piece of data. While variables that store data also exist in C, we use pointers instead as they can handle the variable parameters passed to functions.
We must implement an Order function as the program doesent support random access. This program is surrounded by limitations, as it will only work well with sorted datasets (unless I had implemented a sorting function).

Comparison

As detected from the explanation of my implementations above, the Object-Oriented approach to the Binary Search program was far easier to implement and handle than the Imperative approach.

What I found most difficult to grapple with was the fact that the Imperative style of the C implementation doesent support inheritance, meaning cose re-usability wasn't possible. I found myself relying on this in the Object-Oriented approach.

Another issue I learned from researching the functionality is the data privacy differences between the two approaches. Object-Oriented languages enable *data encapsulation,* a feature that allows the hiding of values or states of data objects in classes. What this means is that the data contained in the object can only be accessed through the object's methods. There are multiple benefits of this; for example, in C programming, you must re-declare instances of functionality but in Python, this is not required. This opens up a can of works in C, as these re-declarations allow our data to be modified by code in random parts of our program. This tells us that Object-Oriented programming is much more secure and based on the real world.

Lastly, a feature available to us in Object-Oriented languages is *accessor methods*. These are methods that allow us to acces private data inside objects. *Encapsulation*, as mentioned above, is the hiding of values, so naturally, these accessor methods would be required for us to access the data it protects. But with these methods come many benefits; it allows us to call instances of the data easily from anywhere in the program to modify it. There is no such feature in Imperative languages.

What I gathered in conclusion from this exercise, is that Imperative approaches are much harder to train to handle complex problems, whereas Object-Oriented features make complex problems much easier to implement, and promote a much more organized appearance.

***References / Sources**:*

1. *'Difference between Object-Oriented and Procedural programming':* *https://www.geeksforgeeks.org/differences-between-procedural-and-object-oriented-programming/*

2. *'Binary Tree searching explained':* *https://medium.com/@mbetances1002/data-structures-binary-search-trees-explained-5a2eeb1a9e8b*

3. *'Pointers in C Programming':* *https://www.guru99.com/c-pointers.html*

4. *'What is Data Encapsulation?':* *https://press.rebus.community/programmingfundamentals/chapter/encapsulation/*