Comparing Imperative and Object-Oriented Programming (OOP)

I decided to choose both programs in python programming language as python can be easily do done in Objected-Oriented style and an imperative style. From my research on this assignment I noticed python is an imperative language like many others such as Java, C++, Ruby, Pascal etc… Basically, what this means is imperative languages allows you to change the state while executing them e.g. set variable "i" to 0 do some stuff and then you can set it something else. It allows you to control flow structures of actions several times. With Object-Oriented programming it's the concept of creating "objects" that can contain data and functions. Also, in Object-oriented you can create a "Class" which is basically a blueprint for an object.

The major benefit right off the bat for the OOP program compared to the imperative program. Is that it allows you to create a Class, constructor, an instance, attributes and objects. The beauty of OOP is the **re-usability** throughout the program, being allowed to reuse your code through inheritance. Compared to imperative where creating "constant" variables would be efficient way to refer to different elements in the phonebook entry.

```
class Phonebook:
    def __init__(self, name, address, phone_number):
        # this is the data part
        self.name = name
        self.address = address
        self.phone_number = phone_number
        self.left = None #Left pointer
        self.right = None #right pointer
```

```
LEFT, VALUE, RIGHT = 0,1,2 #Constant variables , VALUE = whole tuple
NAME,ADDRESS,NUMBER = 0,1,2 #Constant variables
```

Not being able to use **Mutable Objects** such as Lists in the imperative program lead to some disadvantages as mutable objects can make changes in-place, without allocating a new object and **Immutable Objects** require a separate object for each independent value. Notice in the image above I had to reference to the "Tuple" index which is my "VALUE" variable. As this was necessary for my search function to find the specific element "NAME, "ADDRESS, or" NUMBER". Note the image below I check the node by referring to the "VALUE "constant variable and the "field" argument which is a specific element from a contact/entry.

```
def search_by(node, value, field):
    if not node:
        return None
    if node[VALUE][field] == value:
        return node[VALUE]
    if node[VALUE][field] < value:
        return search_by(node[RIGHT], value, field)
    return search_by(node[LEFT], value, field)
```

This is a tedious way of searching for an element (image above). Let's compare this to a more efficient search function in the OOP version. (images below)

```
class Node:
    def __init__(self, value, data=None):
        self.value = value
        self.data = data
        self.left = None
        self.right = None
```

```
def search(self, value, compare_fn):
    if self.value == value:
        return self.data
    else:
        if compare_fn(self.value, value) > 0:
            if self.right == None:
                return None
            else:
                return self.right.search(value, compare_fn)
        else:
            if self.left == None:
                return None
            else:
                return self.left.search(value, compare_fn)
```

Compared to the imperative style it's a more **well-structured** way of doing it. Imagine dealing with much larger input, doing it an imperative way would lead to confusion by creating numerous amounts of constant variables. Whereas I said at the beginning the beauty of OOP allows the user to **re-use** code and notice in the search function there is no indexes. I created a "Node" class which defines what a "Node" is basically, I can use the attributes from it in the search function (as "search" is a instance method of the class). This is a key comparison, as OOP has better **productivity** than imperative style.

```
# All contacts (c = contact)
c1 = ("Jason", "London", "004400")
c2 = ("Noel", "Wigan", "004401")
c3 = ("Peter", "Middlesbrough", "004402")
c4 = ("Danny", "Coventry", "004403")
c5 = ("Steven", "Liverpool", "004404")
```

```
phonebook_records = [
    PhoneBookEntry("Darragh", "Tyrone", "0851234567"),
    PhoneBookEntry("Sarah", "Mayo", "0861234567"),
    PhoneBookEntry("Diarmuid", "Dublin", "0871234567"),
    PhoneBookEntry("Amy", "Wicklow", "0591234567"),
    PhoneBookEntry("Oisin", "Kerry", "0831234567")
    ]
```

Going back to my point on immutable and mutable objects. Note the images above, the imperative style (left) and the OOP style (right). In the OOP program I used a list for all my phone book entries "phonebook_records" (note I call the class "PhoneBookEntry" for each entry) which was much easier to implement as the data could be changed with a list (hence "mutable"). As I said previously, I was limited to not being able to use mutable objects for the imperative program therefore using a **Tuple** (immutable object) was the best option. **Major disadvantage** was I had to assign each entry/contact to a variable. Then when doing the binary tree insertion, I had to call the insert function for each entry/contact. (See images of code below)

```
root = (None, None, None)

root = insert(root, c1, NAME)
root = insert(root, c2, NAME)
root = insert(root, c3, NAME)
root = insert(root, c4, NAME)
root = insert(root, c5, NAME)
```

```
root3 = (None,None,None)
root3 = insert(root3, c1, NUMBER)
root3 = insert(root3, c2, NUMBER)
root3 = insert(root3, c3, NUMBER)
root3 = insert(root3, c4, NUMBER)
root3 = insert(root3, c5, NUMBER)
```

I mentioned earlier, imagine having much larger input. The user might have to write endless lines of code and variables for each entry. It's a more **tedious** way of doing it compared to OOP style. Where I used a simple "for loop" to iterate through each entry/contact to insert into the binary tree. (See image of code below)

```
name_bst = ContactBinaryTree(comparing_strings)
phone_number_bst = ContactBinaryTree(comparing_strings)


for contact in phonebook_records:
    # print(contact)
    name_bst.insert(Node(contact.name, contact))
    phone_number_bst.insert(Node(contact.number, contact))
```

Efficiency is a major advantage that OOP programming has over imperative programming. The imperative program is more **tedious** and **less practical** for writing large programs or any program size for a matter of fact.

```
def insert(self, node, compare_fn):
    if compare_fn(self.value, node.value) > 0:
        if self.right == None:
            self.right = node
        else:
            self.right.insert(node, compare_fn)
    else:
        if self.left == None:
            self.left = node
        else:
            self.left.insert(node, compare_fn)
```

OOP ^

```
def insert(node, new_data, field_idx):
    left_child, data, right_child = node
    if data is None:
        return (None, new_data, None)

    if new_data[field_idx] <= data[field_idx]:
        if left_child is None:
            new_node = (None, new_data, None)
            return (new_node, data, right_child)
        left_child = insert(left_child, new_data, 0)
        return (left_child, data, right_child)

    if new_data[field_idx] > data[field_idx]:
        if right_child is None:
            new_node = (None, new_data, None)
            return (left_child, data, new_node)
        right_child = insert(right_child, new_data, 0)
        return (left_child, data, right_child)
```

Imperative ^

Comparing the two insert methods (images above) like my search function. Using indexes and attributes seem to be the main theme again. Both are done in a recursive way, but I can use class features in the OOP method which creates better **structure.** Note in the imperative insert function, I declare left_child , data and right_child . Where I can just simply use the right, left or value attributes of the "Node" class for the OOP insert function. I don't want to constantly be repeating the same statements, but it is more evidence on how OOP is more **practical** than imperative for larger projects especially.