Student name: Vincent Achukwu

Student Number: 17393546

Project Title: Comparing Imperative and Object-Oriented Programming

Introduction

For this assignment, I have chosen Python3 for the Object-Oriented approach, and C for my imperative approach. I first began by implementing the solution in Python just to get a better understanding of the problem so I could later translate it to the imperative approach in C.

Both programs implement similar logic with the way things are tested in the main() function. As described in the comments of the code, for Python, I created 2 dictionaries, one which maps phone numbers to name and address and called it “infoNums”, the other dictionary maps names to phone number and address, and called that “infoNames”. In C, I did this using a char array in a matrix-like layout, where each item in the array has 3 entries: name, phone number, address for “infoNames”, phone number, name, address for “infoNums” . Both implementations also have a variable called “toBeAdded”, which contains contact information which must be added to their trees (depending on whether they are present in their tree(s) or not). Any contact in the name-mapping phonebook but not in the phone number-mapping phonebook is added to the phone number-mapping phonebook, and vice versa. This allows for the implementation of 2 binary trees as desired, with each entry having one copy, and that copy being used in both trees. My test cases in the main function tests for insertion, deletion, and searching. Below, you can see how you can run both programs from the terminal.

```
>>> python3 objectOriented.py


>>> gcc imperative.c

>>> ./a.out
```

Imperative Programming Approach

In imperative programming, the program executes in a sequence of steps, instructions, or statements in some order to accomplish the goal and change the program’s state. This style of programming explicitly tells the computer how to accomplish tasks. The order in which the sequence of statements is executed impacts the program’s state, as changing the sequence of commands can change the program state.

Some advantages of imperative programming are how much quicker it can be for a developer to write small-scale programs. Another advantage is that it is easier to read due to less syntax involved. This allows the conceptual model to be easy for beginners to understand and makes it relatively easy to learn. Some disadvantages are how quickly the code starts to grow when more complex problems are being solved. There is also a higher risk of errors during the process of making the program, and the developer does not have access to objects from extensions which are available in the declarative paradigm. It is difficult to add new functionality when the program can only do one thing.

Object Oriented Programming Approach

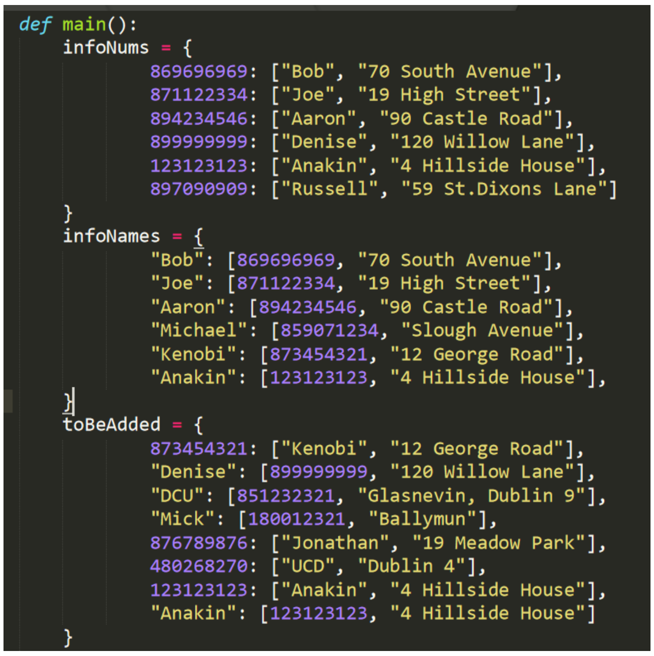
This programming paradigm shows everything as an object. This allows for developers to use interacting objects to solve more complex problems. This programming paradigm allows extensions, as objects can be extended to include some more attributes and other kinds of behaviour. Objects are also reusable and can inherit one another. Object-oriented paradigm is seen as the natural extension of data abstraction, where the objects can represent a general thing which other objects can inherit from and apply more specific attributes or methods. Rather than stepping through the whole program like in imperative programming, the program can jump from one area of code to another. This makes object-oriented programming much more maintainable and flexible if the developer is looking to optimize the code.

Some other advantages include the ability to reuse the code for faster development, which is possible with the use of libraries of objects. It also has the advantage of improved software-development productivity, allowing the developer to have a more fluid style of developing programs. Some disadvantages include having large program sizes. With more lines of code, beginner programmers may find it difficult to follow along the program execution. Also, this programming paradigm may not be suitable for all problems, as it is not always necessary to use classes and objects to do something simple, which may have more overhead and make it complex.

Comparison of solutions

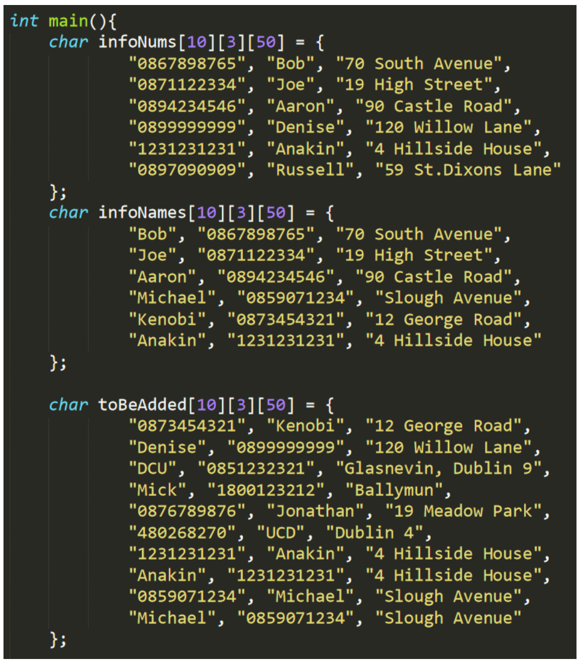
After finishing the OO solution, I simply translated it to an imperative style by using appropriate data types and constantly comparing it to the OO solution. Both solutions begin by storing contact information in the appropriate data structure. In the OO solution, I stored the phone number to (name, address) mapping in a dictionary, and name to (phone number, address) mapping in another dictionary. Missing contacts are also stored in a dictionary, which includes a combination of the mapping orientation from the other 2 dictionaries. The imperative solution begins in the same way, except the contacts are stored in a character array of multiple dimensions (with specified length, number of characters per entry, and maximum length of longest character in the array). This in a way simulates a dictionary. The left screenshot shows part of the OO solution, the right screenshot shows part of the imperative solution. Some contacts in “toBeAdded” may appear in one tree but not the other (for corresponding name/number), and other contacts may not be in either tree or will be added to their respective trees.

Object Oriented:



```
def main():
    infoNums = {
            869696969: ["Bob", "70 South Avenue"],
            871122334: ["Joe", "19 High Street"],
            894234546: ["Aaron", "90 Castle Road"],
            899999999: ["Denise", "120 Willow Lane"],
            123123123: ["Anakin", "4 Hillside House"],
            897090909: ["Russell", "59 St.Dixons Lane"]
    }
    infoNames = {
            "Bob": [869696969, "70 South Avenue"],
            "Joe": [871122334, "19 High Street"],
            "Aaron": [894234546, "90 Castle Road"],
            "Michael": [859071234, "Slough Avenue"],
            "Kenobi": [873454321, "12 George Road"],
            "Anakin": [123123123, "4 Hillside House"],
    }
    toBeAdded = {
            873454321: ["Kenobi", "12 George Road"],
            "Denise": [899999999, "120 Willow Lane"],
            "DCU": [851232321, "Glasnevin, Dublin 9"],
            "Mick": [180012321, "Ballymun"],
            876789876: ["Jonathan", "19 Meadow Park"],
            480268270: ["UCD", "Dublin 4"],
            123123123: ["Anakin", "4 Hillside House"],
            "Anakin": [123123123, "4 Hillside House"]
    }
```

Imperative:



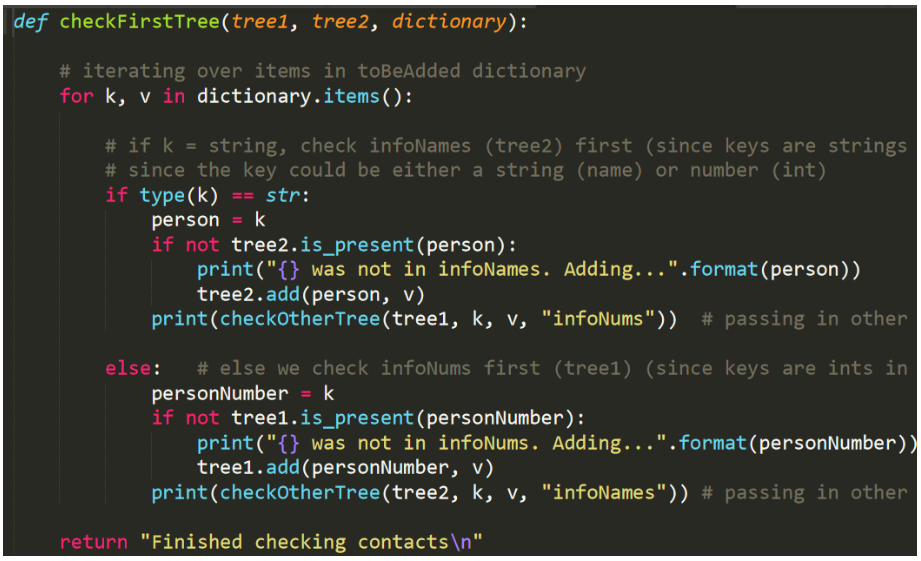
```
int main(){
    char infoNums[10][3][50] = {
            "0867898765", "Bob", "70 South Avenue",
            "0871122334", "Joe", "19 High Street",
            "0894234546", "Aaron", "90 Castle Road",
            "0899999999", "Denise", "120 Willow Lane",
            "1231231231", "Anakin", "4 Hillside House",
            "0897090909", "Russell", "59 St.Dixons Lane"
    };
    char infoNames[10][3][50] = {
            "Bob", "0867898765", "70 South Avenue",
            "Joe", "0871122334", "19 High Street",
            "Aaron", "0894234546", "90 Castle Road",
            "Michael", "0859071234", "Slough Avenue",
            "Kenobi", "0873454321", "12 George Road",
            "Anakin", "1231231231", "4 Hillside House"
    };

    char toBeAdded[10][3][50] = {
            "0873454321", "Kenobi", "12 George Road",
            "Denise", "0899999999", "120 Willow Lane",
            "DCU", "0851232321", "Glasnevin, Dublin 9",
            "Mick", "1800123212", "Ballymun",
            "0876789876", "Jonathan", "19 Meadow Park",
            "480268270", "UCD", "Dublin 4",
            "1231231231", "Anakin", "4 Hillside House",
            "Anakin", "1231231231", "4 Hillside House",
            "0859071234", "Michael", "Slough Avenue",
            "Michael", "0859071234", "Slough Avenue"
    };
```

In the OO solution, it has a node class which defines a node. Each node in the tree has a left and right child (default to None), and a value. The BST class uses the node to get information about each item in the tree. I created 2 tree objects in which one tree gets a phonebook added to it from “infoNums”, and the other has “infoNames” added to it, and this is done in a loop, adding each item to the tree. Once that is done, you can perform other insertions, deletions, and searches in the tree. This makes it quick and easy for the user to call methods from the class instead of having to have things in a certain sequence for the program to run (i.e. the imperative solution).

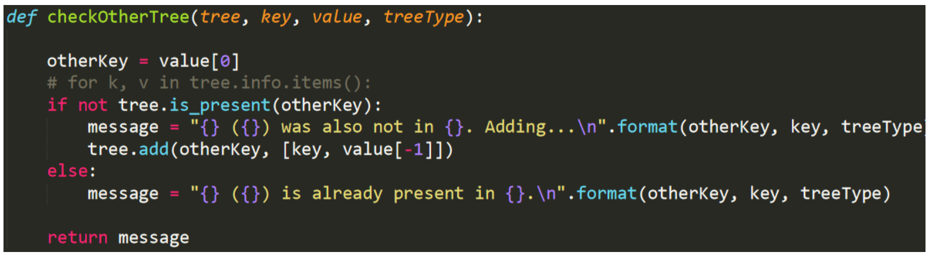
On the imperative solution, again, it is like the OO solution, but there are no objects being created and certain variables must be declared before they can be used. I used a struct to define a node which has a value and two children default to NULL. Instead of a BST “class” like in the OO solution, the tree has pointers which points to different values in the tree, and the tree initially starts as a pointer set to null. This solution was more complicated to write, but with the guide of the OO solution, it made it easier to understand what had to be done. It is clear here why large imperative programs get more complex as they grow.

Both solutions have these methods called “checkFirstTree()” and “checkOtherTree()”. If the item passed to “checkFirstTree()” (from “toBeAdded”) is a name, it checks if it is in its appropriate tree (name to (phone number, address) mapping). If it is missing from its tree, it gets added, then it checks if the phone number associated with the name is in its correct tree (phone number to (name, address) mapping) by calling “checkOtherTree()”. If not, it gets added to its appropriate tree. If the item passed to “checkFirstTree()” is a phone number, the same process takes place, then it calls the “checkOtherTree()” function for the name associated with the number. This was more complex to implement via imperative solution as the size of the program grew.

Object-Oriented “checkFirstTree()” and “checkOtherTree()”:

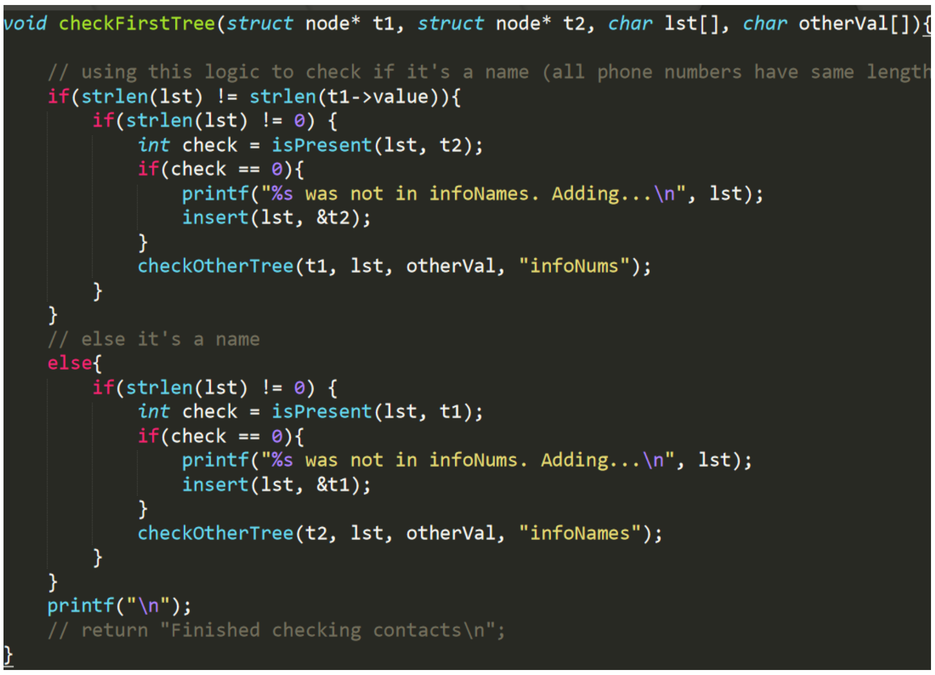


```python
def checkFirstTree(tree1, tree2, dictionary):

    # iterating over items in toBeAdded dictionary
    for k, v in dictionary.items():

        # if k = string, check infoNames (tree2) first (since keys are strings
        # since the key could be either a string (name) or number (int)
        if type(k) == str:
            person = k
            if not tree2.is_present(person):
                print("{} was not in infoNames. Adding...".format(person))
                tree2.add(person, v)
            print(checkOtherTree(tree1, k, v, "infoNums"))  # passing in other

        else:   # else we check infoNums first (tree1) (since keys are ints in
            personNumber = k
            if not tree1.is_present(personNumber):
                print("{} was not in infoNums. Adding...".format(personNumber))
                tree1.add(personNumber, v)
            print(checkOtherTree(tree2, k, v, "infoNames")) # passing in other

    return "Finished checking contacts\n"
```

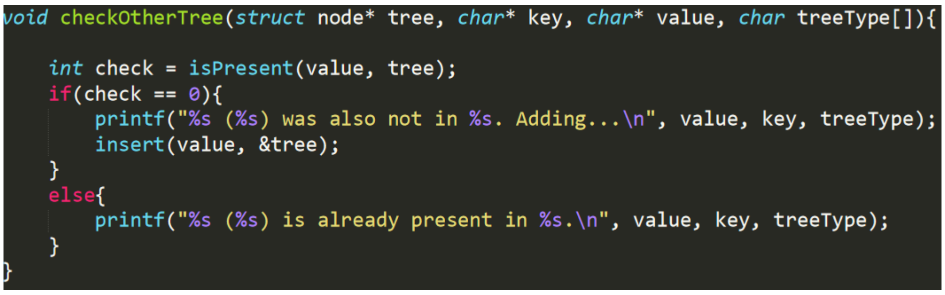


```python
def checkOtherTree(tree, key, value, treeType):

    otherKey = value[0]
    # for k, v in tree.info.items():
    if not tree.is_present(otherKey):
        message = "{} ({}) was also not in {}. Adding...\n".format(otherKey, key, treeType
        tree.add(otherKey, [key, value[-1]])
    else:
        message = "{} ({}) is already present in {}.\n".format(otherKey, key, treeType)

    return message
```

Imperative “checkFirstTree()” and “checkOtherTree()”:



```c
void checkFirstTree(struct node* t1, struct node* t2, char lst[], char otherVal[]){

    // using this logic to check if it's a name (all phone numbers have same length
    if(strlen(lst) != strlen(t1->value)){
        if(strlen(lst) != 0) {
            int check = isPresent(lst, t2);
            if(check == 0){
                printf("%s was not in infoNames. Adding...\n", lst);
                insert(lst, &t2);
            }
            checkOtherTree(t1, lst, otherVal, "infoNums");
        }
    }
    // else it's a name
    else{
        if(strlen(lst) != 0) {
            int check = isPresent(lst, t1);
            if(check == 0){
                printf("%s was not in infoNums. Adding...\n", lst);
                insert(lst, &t1);
            }
            checkOtherTree(t2, lst, otherVal, "infoNames");
        }
    }
    printf("\n");
    // return "Finished checking contacts\n";
}
```



```
void checkOtherTree(struct node* tree, char* key, char* value, char treeType[]){

    int check = isPresent(value, tree);
    if(check == 0){
        printf("%s (%s) was also not in %s. Adding...\n", value, key, treeType);
        insert(value, &tree);
    }
    else{
        printf("%s (%s) is already present in %s.\n", value, key, treeType);
    }
}
```

Conclusions

After doing some research and implementing both solutions in a similar fashion, I found that both have their pros and cons in how they do things. The imperative approach got complex as the code grew, and with the OO approach, the use of classes and objects made it easier to maintain. If a developer were to write a short program with the limitation of certain functions, they can go with the imperative approach, but for better code quality, readability, and reusability, one should go with the OO approach. It makes it much easier to use with built-in functionality and the use of objects. But the OO solution is larger in size which is a known disadvantage of OO programming.

References

1. **BST deletion guide:**
    a. https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/
2. **CA341 David Sinclair Comparative Programming languages:**
    a. https://www.computing.dcu.ie/~davids/courses/CA341/CA341.html
3. **Imperative programming: Overview of the oldest programming paradigm:**
    a. https://www.ionos.com/digitalguide/websites/web-development/imperative-programming/
4. **OOP Advantages and Disadvantages:**
    a. https://resources.saylor.org/wwwresources/archived/site/wp-content/uploads/2013/02/CS101-2.1.2-AdvantagesDisadvantagesOfOOP-FINAL.pdf