Comparing Imperative and Object-Oriented Programming

I decided to do the object-oriented approach to this assignment in Java and the imperative approach in c. Java was chosen because everything in Java is an object and this would make it easier to implement. During the implementation of both programming paradigms, I started to notice the features that each had to offer. As part of this assignment, I had to implement a phonebook program that uses a binary search tree to store, remove and search entries in the phonebook. A Binary Search Tree is a Binary tree where the value of any node is greater than all the values in its left subtree and less than all the values in its right subtree.

I started with the object-oriented approach in Java as I had a good idea of how to implement a binary search tree after learning about them in CA268-programming 3 in 2nd year. Using an object-oriented approach, I was able to create classes. A class is a representation of a type of object. It is a blueprint or template that describes the details of an object. I therefore created a "Node" class. This would represent a node in a binary search tree. It would allow me to create a node object. An object is an instance of a class and it has its own state, behavior and identity. I was also able to decide on certain attributes on the fields and methods of this class e.g. public, private ect. This is known as encapsulation and is a concept used to hide the state of an object from outside the class. I decided to set the fields of the class to public so that the other classes in my program could access them easily. As this was a phonebook program, a node would have to contain a name, an address, a number, a left child node and a right child node. These were all set to public. I also initialized a node with a map which mapped the number to a list containing the name and address. This would make it easier to retrieve such details later in the program. Every field was initialized in the constructor. This is a method used to initialize the state of an object and it gets invoked at the time the object is created. I then created a "toString" method. If you want to print a node, a "toString" method is used. This allowed me to check that a node could be created and printed out so I then knew I was on the right track.

My next task was to create a binary search tree. In this program there needed to be two binary search trees, one where the phone number was used for insertion and searching and the other where the name was used for insertion and searching.

I created a "BinarySearchTree1" class. This would act as the phonebook. It contains all the methods associated with that binary search tree. The first thing I wanted to be able to do with a phonebook is to add an entry to it. An entry in a binary search tree is a node. I previously created a "Node" class and so this is where the object-oriented approach comes in handy. In this "add" method, I pass through a Node(ie the root), a name, an address and a phone number as arguments. The first thing it checks is if the node provided in the arguments has a "null" value. If it does, a new node will be created by accessing the "Node" class. The name, address and number used in the "add" method arguments are then passed through the Node class. This results in a new node(or phonebook contact) been made in the tree. Or if the node is not null, we check if the number is less than the current node's number. If it is less, than we recursively go back through the function adding the node to the left of the current node. If it is greater, we add recursively to the right. We then return this node.

After I created the “add” method, my next task was to get the “searchNameAndAddress” method working. This method would take the current node and a phone number and return the name and address of the person associated with that phone number in a list. It checks if the node is null first. If it is, it returns the node. Otherwise, enter into a while loop which will always result as true until the number is found. When found, we can access the map that was initialized with the node and get the value associated with the number. The value in this case is a list containing the person’s name and address.

The “remove” method was the trickiest to implement as there are many cases involved in-order to remove a node from the binary search tree. The first thing involved is finding the node you want to remove. This involves recursively searching through the left or right nodes depending on if the number is less or greater than the current number. Once the number is equal to the current node number then we know we found the node to remove. The first case we want to check is if the node to remove has no children. A temporary node is created, if the left node is null, let the temporary node equal this, if not let the temporary node equal the right node. If the temporary node is null, return null. Otherwise return the temporary node. The other case involved is if the node to remove has both children. A helper function is created called “getChild” and this takes a node as an argument. This retrieves the child node and returns it, allowing to remove its parent and have the child node still be an entry in the binary search tree.

Lastly, I wanted to print out the nodes in the binary search tree so that I knew it worked as expected. I therefore created an “inOrderTraverseTree”, “preOrderTraverseTree” and “postOrderTraverseTree” methods. These allowed me to traverse through the tree printing out the nodes in whatever order I wanted.

All these methods were part of the “BinarySearchTree1” class. Once I had all these defined, it allowed me to go into my main method and create a binary search tree object. Once this object was created, I had access to all the methods associated with that object. I added nodes to the tree, searched for the name and address given a number, removed an entry from the tree and then traversed through the tree in order.

As this assignment needed two binary search trees, I then created another class – “BinarySearchTree2”. This class is very similar to the previous binary search tree created except it uses the name of the person for insertion and searching. When adding a node or searching for a node, I used the “compareTo” method to compare two names alphabetically. This method returns an integer either < 0 or greater than 0 indicating if it is higher or lower alphabetically. This allowed me to traverse through the node’s left and right children easily and to perform the same functionality as in the previous binary search tree.

Moving onto the imperative approach. I never learned the c programming language before so there was quite a bit of research to be done. I found a good example of a binary search tree implemented in c online(from codesdope.com) so this was a good starting point. I learned about “struct” and why they would be useful in implementing a binary search tree. A struct is a composite data type declaration that defines a grouped list of variables under one name in a block of memory. These variables would be able to be accessed by a pointer. This would work well for the binary search tree as I could define a struct called “node” and similar to the node class in the object-oriented approach, this struct would contain some variables such as a name, address, number, a node pointing to the right child and a node pointing to the left child. Interestingly, a string in c is defined by a list of characters, so the name and address variables had to be assigned this way. After this, I decided to use functions for the rest of the program. It makes the code easier to read and ensured that I didn’t repeat myself. Similar to my previous approach, I knew I needed to be able to create a new node so that

this node could be added to the binary search tree. I created a function “new_node” which takes a name, an address and a phone number as arguments. In this function, a pointer “*p” is created. A pointer holds an address in memory. “*p” accesses the memory pointer “p” points to. “p->name[31]” accesses the element “name” of a struct that “p” points to. When creating p, use the library malloc(size) to allocate memory dynamically. So rather than creating a new instance of a node like what was done in the object-oriented approach, a pointer to a node is used instead. The logic for the rest of the functions such as “add”, “search” and “remove” is very similar to the object-oriented approach but in these functions pointers were used. Pointers to the nodes allow us to access the elements at these memory locations. With this in mind, I was able to compare the current phone number with the phone number provided in the arguments. This allowed me to traverse through the tree in the same way as I did in the object-oriented approach. In this way, I could again add, remove and search nodes. As I was only learning c for the first time, I didn’t get the two binary search trees working. With more time and learning, I definitely feel like I could get this program fully complete.

Once I had finished both versions of the phonebook, I was able to clearly see the main differences between both programming paradigms. In my Java object-oriented approach, all the major parts of the program were divided into classes. This made it easier to identify what was going on in the program as the classes were labelled really well. In comparison, the imperative approach in c didn’t use classes but to make it easier to read I divided everything into functions. It wasn’t as clear as the object-oriented version but the best that I could do given the limitations of the imperative approach. The imperative approach in c could be read procedurally from top to bottom unlike the object-oriented approach in java which called classes within other classes making it a bit harder to read through. My Java implementation was divided up into four separate files. This is very beneficial as you don’t have to read through one massive piece of code, it is broken up instead. Having separate files can be awkward to navigate through but is so much better for anyone trying to understand your code. In an imperative approach, you can assign variables and overwrite these at any time. In object-oriented programming, you can use encapsulation which allows you to stop the ability to overwrite variables. Object-oriented programming also allows for abstraction to be used by creating abstract classes and it also allows inheritance to be used too. Inheritance is a concept where one class shares the structure and behaviour defined in another class. I didn’t need to implement either of these as the program didn’t require it but the option is there. Java classes can also be used as a data type unlike in an imperative approach where the usual data types would have to be used. In terms of scalability, object-oriented programming would be able to handle this much better. Dividing everything into classes would therefore work in our favor and if anything needs to be modified it can be done within a class. This sums up my comparison on object oriented and imperative programming paradigms.