Nicolas LIDIN,
French international student.

Object-Oriented and Imperative paradigm comparison in BinarySearchTree implementation:

I wrote the two implementations of the binary search tree in Imperative and Object-Oriented paradigm with Kotlin language. Kotlin is a good language for this kind of comparison cause it is a multi-paradigm language. Indeed even if Kotlin is 100% interoperable with Java, it allows us to develop with Imperative and Functional paradigm in addition to the Object-Oriented one. (In this assignment only Imperative and Oriented Object paradigms are used)
I think it's interesting to use one single language for the two examples cause it allows us to focus specifically on the differences caused by both paradigms and not by the features and syntax of different languages.

The two implementations have several main differences:

1. The main difference between the two implementations is the structure of the program:

In imperative paradigm BinarySearchTree implementation, we just use DataClass which is the same as a Structure in C (we can just have properties and no methods). Consequently, it means that all functions related to the BinarySearchTree data structure can't be directly associated to it and should be declared (and implemented) outside of the BinarySearchTree data-structure. By doing this, all the methods related to BinarySearchTree need to receive the BinarySearchTree reference of the instance to manipulate the data (rootNode property here)

On the other hand, object-oriented implementation is better structured. Indeed all functions associated with the BinarySearchTree class are methods inside this class.
It acts as a namespace with encapsulation and it is semantically clearer. So it's more clear when we want to see which methods are related to a specific class, but it also clearer at the usage. Indeed we don't need to pass the reference of the binary tree when calling a method, the call will be made from an instance of BinarySearchTree (as we can see in the main).

2. The other main difference between these two implementations is the Generic Constraint feature:

The generics feature is a feature allowing to have a generic type with potential constraint. This feature of constraint on generics is available only for the Oriented-Object implementation. It allows receiving for example a generic type that has the constraint to implement a specific interface. It's what I did for the Node type in the BinarySearchTree class, indeed the content of a node is Pair<K, V> where K implements the interface Comparable. It means we can receive all type that is comparable.
The implementation of a BinarySearchTree shows the importance of this feature.
This feature is very useful, in a lot of programs, it's very important to have good genericity and type safety with constraint.

To conclude, an implementation with Oriented-Object Paradigm is preferable cause it allows having better visibility, modularity, abstraction, and genericity with all features related to class inheritance and interface implementation.
However, class instantiation with associated methods is not always useful and it can be sometimes more interesting to have functions not scoped in a class.
It’s why multi-paradigm are becoming more and more important in the industry. Rust, Kotlin, and Typescript are good candidates for multi-paradigm languages.