**Procedural (Python)**

I implemented the tree by using a dictionary with the keys val, and children. Where val is the value of the node and children is an array containing the child nodes. Each element of the children array is the same structure. This implementation ended up being very similar to my function data structure, but this used a Python dictionary instead of a Racket list, and in my opinion is much more human readable.

I did this implementation in Python. My thought process was to not use any of the object oriented features of the language and to stay away from recursion as I see that as a functional feature. A recursive approach was the first thing that came to my head for the depth first traversal because I knew I should utilize a stack for this. I decided to wait to use recursion only in the functional paradigm. I then had to think about how to replicate this recursion using an iterative approach and a stack. My idea was to keep track of the visited children in a set (for the constant lookup time) and to essentially add the first unvisited child to the stack then go to that child and repeat until there is a leaf node or a node whose children have all been visited. At that point, visit the node and add it to the visited set. This was accomplished with a while loop where the condition is the stack isn’t empty (the first element of the tree gets added before the loop) and a for loop to iterate through the children. For the Breadth first traversal I used a queue and simply visited the current node, added its children to queue, then removed it and continued with the loop.

**Object Oriented (Java)**

I implemented the tree using a class, TreeNode with the attributes value and children where value is the value of the node and children is an arrayList of TreeNodes. The class had the methods, addChild, getChildren, and getValue. I feel like the object-oriented approach of constructing the tree is the best out of the three paradigms. This is because it allows me to be clear about the structure and everything is self-contained in its own object. It’s also helpful that methods can be added to the class to do various functions.

The implementation of the algorithms here was very similar to the procedural approach. The Java Stack and Queue classes were helpful. One thing I noticed while working on the object-oriented approach was the helpfulness of the type checking and built in type safety with things like generics. It was definitely more to type out, but it saved me from a couple bugs.

**Functional (Racket)**

In racket the tree implementation was pretty similar to the procedural implementation but instead of dictionaries, plain lists were used. So, the first value of a list would always be the value and the others would the children. I will say this implementation, even with proper indentation, was very hard to read. However, I do feel like it worked well with Racket because to access the value you could get the car and to access the children you simply get the cdr.

The functional implementation was the most confusing and time-consuming for me. This is probably largely because I have by far the least amount of experience working in this paradigm and it all felt pretty foreign to me. For my depth first traversal, the basic plan was to first see if the node is a leaf node (base case), if so, return the node. If not, map over the children, recurse, and add them to a list with the current node at the end. The call stack works as my external stack here. This approach is similar to the two others, but iteration is swapped for recursion and the external queue is swapped for the call stack. I struggled for a while with the breadth first traversal because I felt like it was very strange to do in a recursive context. Due to the inherent call stack (not call queue). Ultimately, I ended up adding in a queue and a result list that get passed between recursive calls. The steps were to remove the car of the queue and add its car (the value) to the result list. Add each child to the queue as a separate element at the back of the queue (used append). Then recurse with the new queue and result. This approach works and is actually very little code, but I don’t really like it. I feel like it kind of works around the functional paradigm by creating the queue and result lists. It also created kind of an awkward function that just instantiates these lists then passes them into the traverse function. This was done so that when a user calls the breadth first method they don’t have to pass in a list for the queue and result.