Tree data structures have many things in common with their botanical cousins. A tree data structure has a root, branches, and leaves. The difference between a tree in nature and a tree in computer science is that a tree data structure has its root at the top and its leaves on the bottom.

Notice that you can start at the top of the tree and follow a path made of circles and arrows all the way to the bottom. At each level of the tree we might ask ourselves a question and then follow the path that agrees with our answer. For example (Figure T1) we might ask, “Is this animal a

Chordate or an Arthropod?” If the answer is “Chordate” then we follow that path and ask, “Is this Chordate a Mammal?” If not, we are stuck (but only in this simplified example). When we are at the Mammal level we ask, “Is this Mammal a Primate or a Carnivore?” We can keep following paths until we get to the very bottom of the tree where we have the common name.

A second property of trees is that all of the children of one node are independent of the children of another node. For example, the Genus Felis has the children Domestica and Leo. The Genus Musca also has a child named Domestica, but it is a different node and is independent of the Domestica child of Felis. This means that we can change the node that is the child of Musca without affecting the child of Felis.

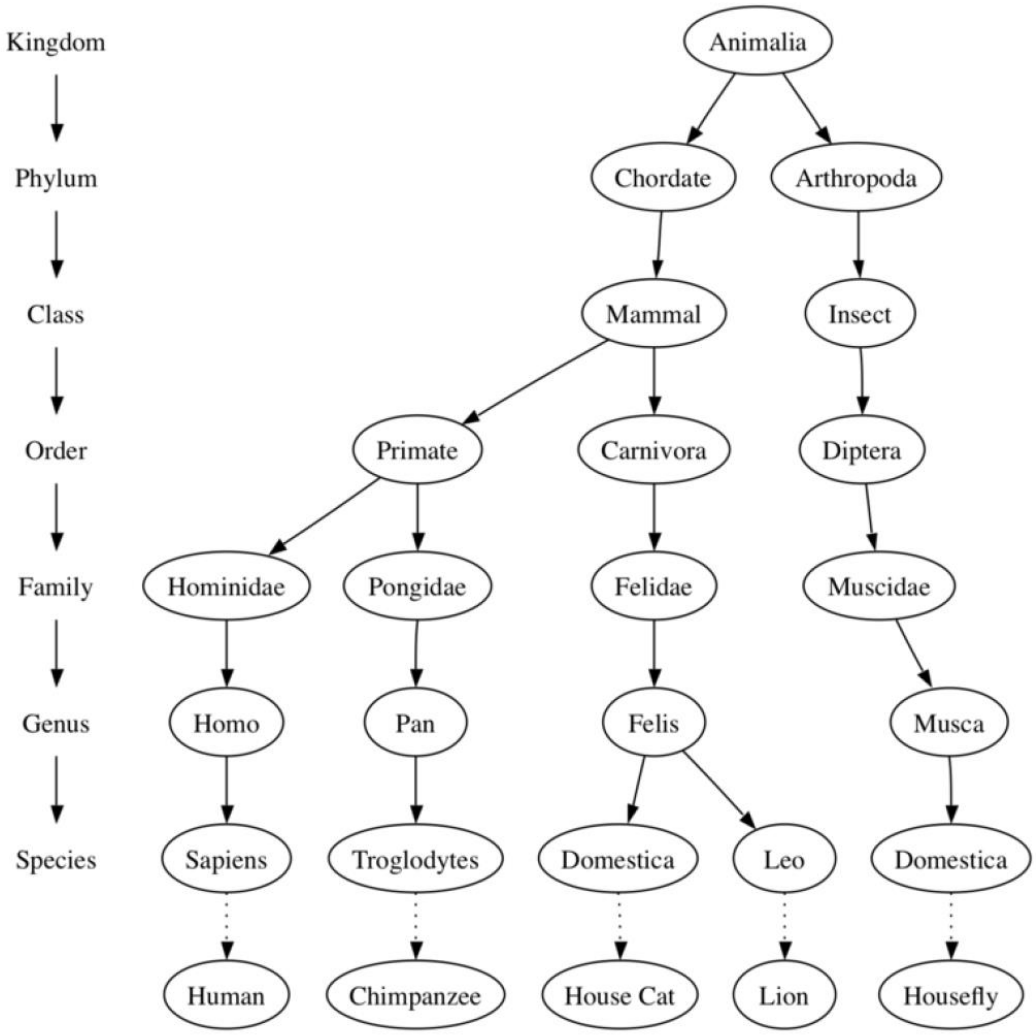

Figure T1 - Taxonomy of Some Common Animals Shown as a Tree

A third property is that each leaf node is unique. We can specify a path from the root of the tree to a leaf that uniquely identifies each species in the animal kingdom; for example, Animalia→Chordate→ Mammal→ Carnivora→Felidae → Felis→ Domestica.

Node A node is a fundamental part of a tree. It can have a name, which we call the "key." A node may also have additional information. We call this additional information the "payload." While the payload information is not central to many tree algorithms, it is often critical in applications that make use of trees.
Edge An edge is another fundamental part of a tree. An edge connects two nodes to show that there is a relationship between them. Every node (except the root) is connected by exactly one incoming edge from another node. Each node may have several outgoing edges.
Root The root of the tree is the only node in the tree that has no incoming edges.
Path A path is an ordered list of nodes that are connected by edges. For example, Mammal→ Carnivora→ Felidae→ Felis→ Domestica is a path.
Children The set of nodes c that have incoming edges from the same node to are said to be the children of that node.
Parent A node is the parent of all the nodes it connects to with outgoing edges.

Sibling Nodes in the tree that are children of the same parent are said to be siblings.
Subtree A subtree is a set of nodes and edges comprised of a parent and all the descendants of that parent.
Leaf Node A leaf node is a node that has no children. For example, Human and Chimpanzee are leaf nodes in Figure T1.
Level The level of a node n is the number of edges on the path from the root node to n. For example, the level of the Felis node in Figure T1 is five. By definition, the level of the root node is zero.
Height The height of a tree is equal to the maximum level of any node in the tree.

Definition One A tree consists of a set of nodes and a set of edges that connect pairs of nodes.
A tree has the following properties:
• One node of the tree is designated as the root node.
• Every node n, except the root node, is connected by an edge from exactly one other node p, where p is the parent of n.
• A unique path traverses from the root to each node.
• If each node in the tree has a maximum of two children, we say that the tree is a binary tree.
Figure T2 illustrates a tree that fits definition one. The arrowheads on the edges indicate the direction of the connection.

Definition Two A tree is either empty or consists of a root and zero or more subtrees, each of which is also a tree. The root of each subtree is connected to the root of the parent tree by an edge. Figure T3 illustrates this recursive definition of a tree. Using the recursive definition of a tree, we know that the tree in Figure T3 has at least four nodes, since each of the triangles representing a subtree must have a root. It may have many more nodes than that, but we do not know unless we look deeper into the tree.

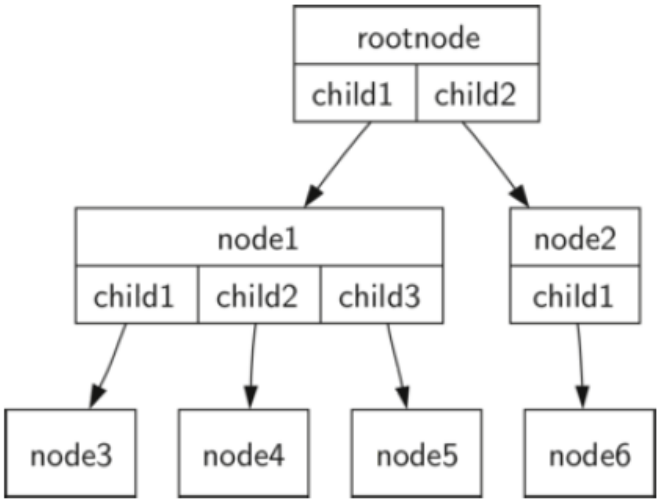

Figure T2 - A Tree Consisting of a Set of Nodes and Edges

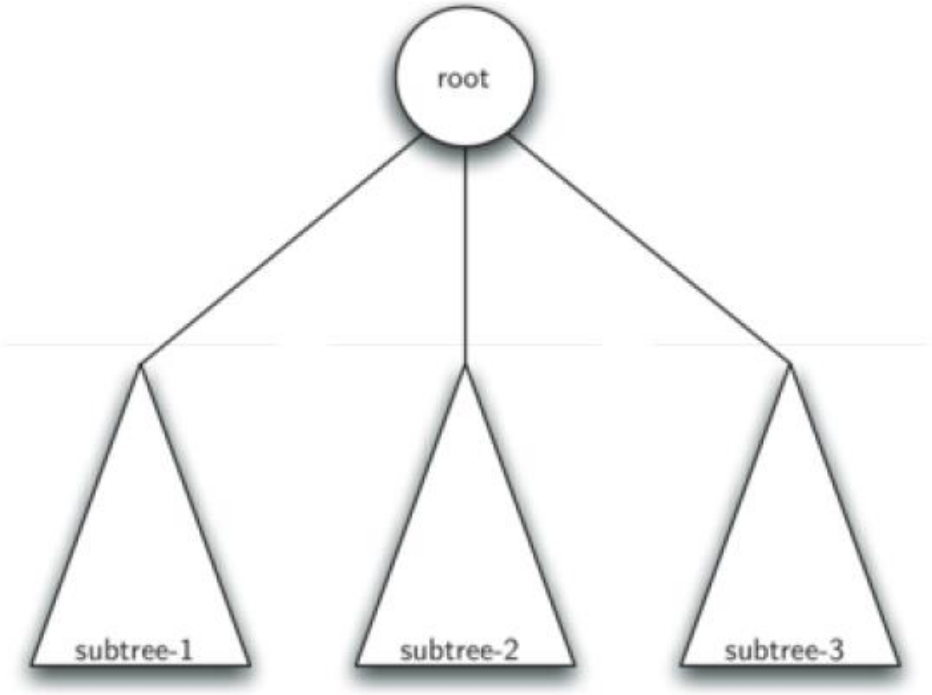

Figure T3 - A recursive Definition of a tree

The following functions can be used to create and manipulate a binary tree:

- `BinaryTree()` creates a new instance of a binary tree.
- `get_left_child()` returns the binary tree corresponding to the left child of the current node.
- `get_right_child()` returns the binary tree corresponding to the right child of the current node.
- `set_root_val(val)` stores the object in parameter val in the current node.
- `get_root_val()` returns the object stored in the current node.
- `insert_left(val)` creates a new binary tree and installs it as the left child of the current node.
- `insert_right(val)` creates a new binary tree and installs it as the right child of the current node.

The key decision in implementing a tree is choosing a good internal storage technique. Python allows us two very interesting possibilities, so we will examine both before choosing one. The first technique we will call "list of lists," the second technique we will call "nodes and references."

In a tree represented by a list of lists, we will begin with Python's list data structure and write the functions defined above. Although writing the interface as a set of operations on a list is a bit different from the other abstract data types we have implemented, it is interesting to do so because it provides us with a simple recursive data structure that we can look at and examine directly. In a list of lists tree, we will store the value of the root node as the first element of the list. The second element of the list will itself be a list that represents the left subtree. The third element of the list will be another list that represents the right subtree. To illustrate this storage technique, let's look at an example. Figure T4 shows a simple tree and the corresponding list implementation.

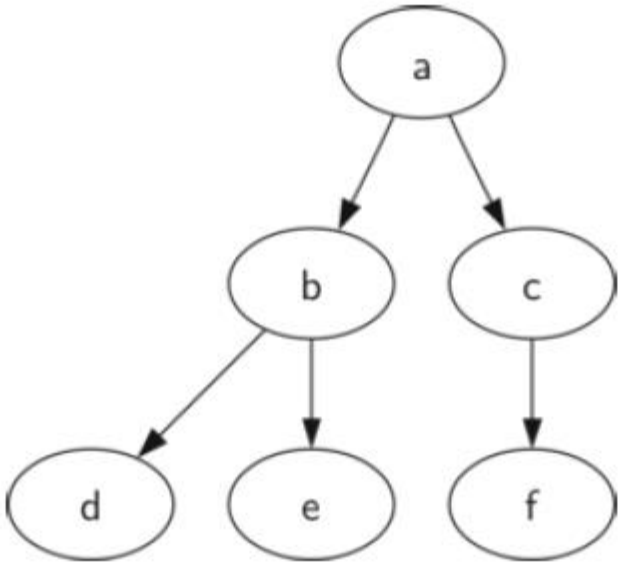

Figure T4 – A Small Tree

```
my_tree = ['a', #root
      ['b', #left subtree
       ['d' [], []],
       ['e' [], []] ],
      ['c', #right subtree
       ['f' [], []],
       [] ]
     ]
```

Notice that we can access subtrees of the list using standard list indexing. The root of the tree is my_tree[0], the left subtree of the root is my_tree[1], and the right subtree is my_tree[2]. The code below illustrates creating a simple tree using a list. Once the tree is constructed, we can access the root and the left and right subtrees. One very nice property of this list of lists approach is that the structure of a list representing a subtree adheres to the structure defined for a tree; the structure itself is recursive! A subtree that has a root value and two empty lists is a leaf node. Another nice feature of the list of lists approach is that it generalizes to a tree that has many subtrees. In the case where the tree is more than a binary tree, another subtree is just another list.

```
my_tree = ['a', ['b', ['d',[],[]], ['e',[],[]] ], ['c',
   ['f',[],[]], []] ]
print(my_tree)
print('left subtree = ', my_tree[1])
print('root = ', my_tree[0])
print('right subtree = ', my_tree[2])
```

Let us formalize this definition of the tree data structure by providing some functions that make it easy for us to use lists as trees. Note that we are not going to define a binary tree class. The functions we will write will just help us manipulate a standard list as though we are working with a tree.

```
def binary_tree(r):
   return [r, [], []]
```

The binary_tree function simply constructs a list with a root node and two empty sublists for the children. To add a left subtree to the root of a tree, we need to insert a new list into the second position of the root list. We must be careful. If the list already has something in the second position, we need to keep track of it and push it down the tree as the left child of the list we are adding. The code below shows the Python code for inserting a left child.

```
def insert_left(root, new_branch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1, [new_branch, t, []])
    else:
        root.insert(1, [new_branch, [], []])
    return root
```

Notice that to insert a left child, we first obtain the (possibly empty) list that corresponds to the current left child. We then add the new left child, installing the old left child as the left child of the new one. This allows us to splice a new node into the tree at any position. The code for insert_right is similar to insert_left and is shown below.

```
def insert_right(root, new_branch):
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2, [new_branch, [], t])
    else:
        root.insert(2, [new_branch, [], []])
    return root
```

```
def get_root_val(root):
    return root[0]

def set_root_val(root,new_val):
    root[0] = new_val

def get_left_child(root):
    return root[1]
def get_right_child(root):
    return root[2]
```

The following code exercises the tree functions we have just written. You should try it out for yourself.

```
def binary_tree(r):
   return [r, [], []]

def insert_left(root, new_branch):
   t = root.pop(1)
   if len(t) > 1:
      root.insert(1, [new_branch, t, []])
   else:
      root.insert(1, [new_branch, [], []])
   return root

def insert_right(root, new_branch):
   t = root.pop(2)
   if len(t) > 1:
      root.insert(2, [new_branch, [], t])
   else:
      root.insert(2, [new_branch, [], []])
   return root

def get_root_val(root):
   return root[0]

def set_root_val(root, new_val):
   root[0] = new_val

def get_left_child(root):
   return root[1]

def get_right_child(root):
   return root[2]

r = binary_tree(3)
insert_left(r, 4)
insert_left(r, 5)
insert_right(r, 6)
insert_right(r, 7)
l = get_left_child(r)
print(l)

set_root_val(l, 9)
print(r)
insert_left(l, 11)
print(r)
```

```
print(get_right_child(get_right_child(r)))
```

Exercise 1

Draw the trees and implement the code to extract the following questions

Who does Elias work for?
Who does Richard work for?
The University is run by the Vice Chancellor
The Vice Chancellor is assisted by a number of Pro Vice Chancellors, these are ProVC for Education, Research, Finance, and Estates
There are four ProVCs / Executive Deans who are in charge of the four Faculties They report to the ProVC for Education.
Each Faculty has four Department Heads.
Each Department has four Associate Heads. Academic, Student Engagement, Research, Enterprise.
There are a number of Academics with different roles, some are Professors and others Lecturers. Some of the Academics are programme leaders.
Programme Leaders report to the HoD, as do Professors.
Lectures report to the Associate Head Academic.
Richard is a Lecturer and so is John, David and Peter.
Jim is a Professor.
Elias is a Lecturer and a Programme Leader.

Exercise 2

Consider your family to a depth of four generations.
Implement a tree that will allow you to represent your family structure, enter member details and allow for future generations to be added.

Other Exercises:

Given the following statements:

```
x = binary_tree('a')
insert_left(x,'b')
insert_right(x,'c')
insert_right(get_right_child(x), 'd')
insert_left(get_right_child(get_right_child(x)), 'e')
```

Which of the answers is the correct representation of the tree?

1. `['a', ['b', [], []], ['c', [], ['d', [], []]]]`
2. `['a', ['c', [], ['d', ['e', [], []], []]], ['b', [], []]]`
3. `['a', ['b', [], []], ['c', [], ['d', ['e', [], []], []]]]`
4. `['a', ['b', [], ['d', ['e', [], []], []]], ['c', [], []]]`

Write a function build_tree that returns a tree using the list of lists functions that looks like this: