# List Representations

In a tree represented by a list of lists, we will begin with Python's list data structure. 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, the chart below shows a simple tree and the corresponding list implementation.
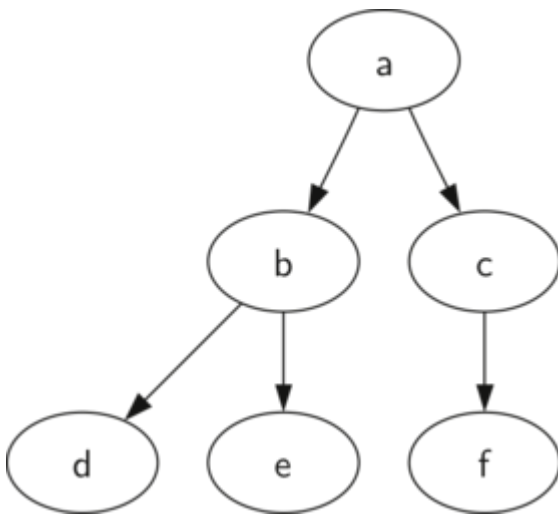

```
myTree = ['a',   #root
      ['b',  #left subtree
       ['d', [], []],
       ['e', [], []] ],
      ['c',  #right subtree
       ['f', [], []],
       [] ]
    ]
```

Notice that we can access subtrees of the list using standard list indexing. The root of the tree is `myTree[0]`, the left subtree of the root is `myTree[1]`, and the right subtree is `myTree[2]`. 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.

Let's 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.

```python
def BinaryTree(r):
    return [r, [], []]
```

The Binary Tree Function

The `BinaryTree` 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. Listing 1 shows the Python code for inserting a left child.

Listing 1

```python
def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    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 `insertRight` is similar to `insertLeft` and is shown in Listing 2.

Listing 2

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

To round out this set of tree-making functions, look at Listing 3. Let's write a couple of access functions for getting and setting the root value, as well as getting the left or right subtrees.

Listing 3

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

def setRootVal(root,newVal):
    root[0] = newVal

def getLeftChild(root):
    return root[1]

def getRightChild(root):
    return root[2]
```

Source: **Problem Solving and Algorithms in Python** **(http://interactivepython.org/runestone/static/pythonds/index.html#)** from Bradley Miller on **www.interactivepython.org** **(http://interactivepython.org/runestone/static/pythonds/index.html#)** .