Nodes and References

A second method to represent a tree uses nodes and references. In this case we will define a class that has attributes for the root value, as well as the left and right subtrees.

Using nodes and references, we might think of the tree as being structured like the one shown in Figure NR 1.

We will start out with a simple class definition for the nodes and references approach as shown below. The important thing to remember about this representation is that the attributes left and right will become references to other instances of the BinaryTree class. For example, when we insert a new left child into the tree we create another instance of BinaryTree and modify self.left_child in the root to reference the new tree.

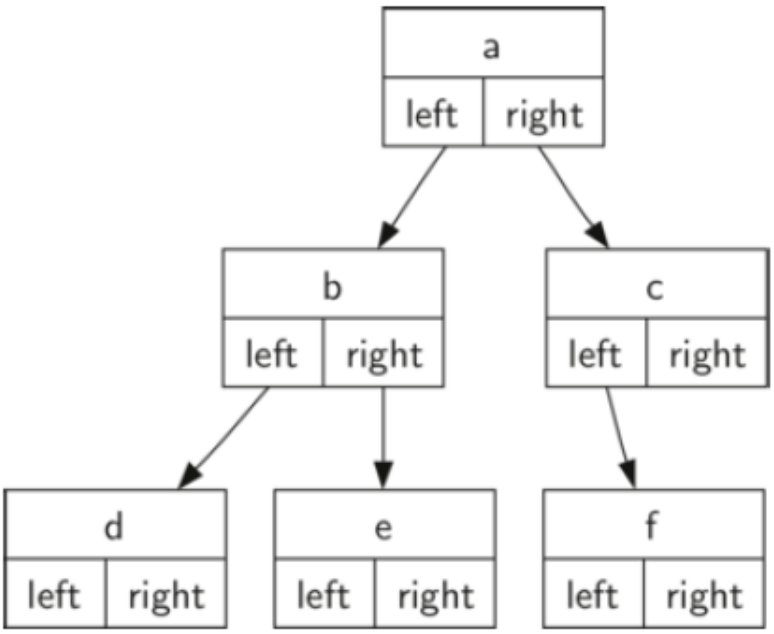


Figure NR 1 - A Simple Tree Using a Nodes and References Approach

```
class BinaryTree:
    def __init__(self, root):
        self.key = root
        self.left_child = None
        self.right_child = None
```

Notice that the constructor function expects to get some kind of object to store in the root. Just like you can store any object you like in a list, the root object of a tree can be a reference to any object. For our early examples, we will store the name of the node as the

root value. Using nodes and references to represent the tree in Figure NR1, we would create six instances of the BinaryTree class.

Next let's look at the functions we need to build the tree beyond the root node. To add a left child to the tree, we will create a new binary tree object and set the left attribute of the root to refer to this new object. The code for insert_left is shown below.

```
def insert_left(self,new_node):
    if self.left_child == None:
        self.left_child = BinaryTree(new_node)
    else:
        t = BinaryTree(new_node)
        t.left_child = self.left_child
        self.left_child = t
```

We must consider two cases for insertion. The first case is characterized by a node with no existing left child. When there is no left child, simply add a node to the tree. The second case is characterized by a node with an existing left child. In the second case, we insert a node and push the existing child down one level in the tree. The second case is handled by the else statement on line 4 of insert_left.

The code for insert_right must consider a symmetric set of cases. There will either be no right child, or we must insert the node between the root and an existing right child. The insertion code is shown below.

```
def insert_right(self,new_node):
    if self.right_child == None:
        self.right_child = BinaryTree(new_node)
    else:
        t = BinaryTree(new_node)
        t.right_child = self.right_child
        self.right_child = t
```

To round out the definition for a simple binary tree data structure, we will write accessor methods for the left and right children, as well as the root values.

```
def get_right_child(self):
    return self.right_child

def get_left_child(self):
    return self.left_child

def set_root_val(self,obj):
    self.key = obj

def get_root_val(self):
    return self.key
```

Now that we have all the pieces to create and manipulate a binary tree, let's use them to check on the structure a bit more. Let's make a simple tree with node a as the root, and add nodes b and c as children. The code below creates the tree and looks at the some of the values stored in key, left, and right. Notice that both the left and right children of the root are themselves distinct instances of the BinaryTree class. As we said in our original recursive definition for a tree, this allows us to treat any child of a binary tree as a binary tree itself.

```
class BinaryTree:
    def __init__(self, root):
        self.key = root
        self.left_child = None
        self.right_child = None

    def insert_left(self, new_node):
        if self.left_child == None:
            self.left_child = BinaryTree(new_node)
        else:
            t = BinaryTree(new_node)
            t.left_child = self.left_child
            self.left_child = t

    def insert_right(self, new_node):
        if self.right_child == None:
            self.right_child = BinaryTree(new_node)
        else:
```

```
class BinaryTree:
    def __init__(self, root):
        self.key = root
        self.left_child = None
        self.right_child = None

    def insert_left(self, new_node):
        if self.left_child == None:
            self.left_child = BinaryTree(new_node)
        else:
            t = BinaryTree(new_node)
            t.left_child = self.left_child
            self.left_child = t

    def insert_right(self, new_node):
        if self.right_child == None:
            self.right_child = BinaryTree(new_node)
        else:
```

```
            t = BinaryTree(new_node)
            t.right_child = self.right_child
            self.right_child = t


    def get_right_child(self):
        return self.right_child

    def get_left_child(self):
        return self.left_child

    def set_root_val(self, obj):
        self.key = obj

    def get_root_val(self):
        return self.key


r = BinaryTree('a')
print(r.get_root_val())
print(r.get_left_child())
r.insert_left('b')
print(r.get_left_child())
print(r.get_left_child().get_root_val())
r.insert_right('c')
print(r.get_right_child())
print(r.get_right_child().get_root_val())
r.get_right_child().set_root_val('hello')
print(r.get_right_child().get_root_val())
```

Food for thought….

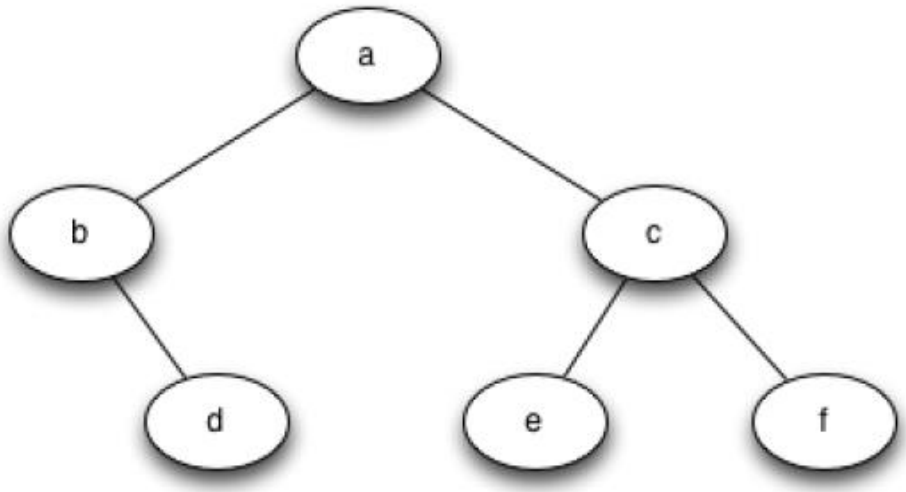
Given the following statements:

```
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:



Write a function build_tree that returns a tree using the nodes and references implementation that looks like this: