MORE THAN LISTS

By reading this chapter, you might be starting to think recursion in Erlang is mainly a thing concerning lists. While lists are a good example of a data structure that can be defined recursively, there's certainly more than that. For the sake of diversity, we'll see how to build binary trees, and then read data from them.

First of all, it's important to define what a tree is. In our case, it's nodes all the way down. Nodes are tuples that contain a key, a value associated to the key, and then two other nodes. Of these two nodes, we need one that has a smaller and one that has a larger key than the node holding them. So here's recursion! A tree is a node containing nodes, each of which contains nodes, which in turn also contain nodes. This can't keep going forever (we don't have infinite data to store), so we'll say that our nodes can also contain empty nodes.

To represent nodes, tuples are an appropriate data structure. For our implementation, we can then define these tuples as `{node, {Key, Value, Smaller, Larger}}` (a tagged tuple!),
where *`Smaller`* and *`Larger`* can be another similar node or an empty node (`{node, nil}`). We won't actually need a concept more complex than that.
Let's start building a module for our very basic tree implementation. The first function,`empty/0`, returns an empty node. The empty node is the starting point of a new tree, also called the *root*:

```
-module(tree).
-export([empty/0, insert/3, lookup/2]).

empty() -> {node, 'nil'}.
```

By using that function and then encapsulating all representations of nodes the same way, we hide the implementation of the tree so people don't need to know how it's built. All that information can be contained by the module alone. If you ever decide to change the representation of a node, you can then do it without breaking external code.

To add content to a tree, we must first understand how to recursively navigate through it. Let's proceed in the same way as we did for every other recursion example by trying to find the base case. Given that an empty tree is an empty node, our base case is thus logically an empty node. So whenever we'll hit an empty node, that's where we can add our new key/value. The rest of the time, our code has to go through the tree trying to find an empty node where to put content.

To find an empty node starting from the root, we must use the fact that the presence of `Smaller` and `Larger` nodes let us navigate by comparing the new key we have to insert to the current node's key. If the new key is smaller than the current node's key, we try to find the empty node inside `Smaller`, and if it's larger, inside `Larger`. There is one last case, though: what if the new key is equal to the current node's key? We have two options there: let the program fail or replace the value with the new one. This is the option we'll take here. Put into a function all this logic works the following way:

```
insert(Key, Val, {node, 'nil'}) ->
{node, {Key, Val, {node, 'nil'}, {node, 'nil'}}};
insert(NewKey, NewVal, {node, {Key, Val, Smaller,
Larger}}) when NewKey < Key ->
{node, {Key, Val, insert(NewKey, NewVal, Smaller), Larger}};
insert(NewKey, NewVal, {node, {Key, Val, Smaller,
Larger}}) when NewKey > Key ->
{node, {Key, Val, Smaller, insert(NewKey, NewVal, Larger)}};
insert(Key, Val, {node, {Key, _, Smaller, Larger}}) ->
{node, {Key, Val, Smaller, Larger}}.
```

Note here that the function returns a completely new tree. This is typical of functional languages having only single assignment. While this can be seen as inefficient, most of the underlying structures of two versions of a tree sometimes happen to be the same and are thus shared, copied by the VM only when needed.

What's left to do on this example tree implementation is creating a `lookup/2` function that will let you find a value from a tree by giving its key. The logic needed is extremely similar to the one used to add new content to the tree: we step through the nodes, checking if the lookup key is equal, smaller or larger than the current node's key. We have two base cases: one when the node is empty (the key isn't in the tree) and one when the key is found. Because we don't want our program to crash each time we look for a key that doesn't exist, we'll return the atom `'undefined'`. Otherwise, we'll return `{ok, Value}`. The reason for this is that if we only returned `Value` and the node contained the atom `'undefined'`, we would have no way to know if the tree did return the right value or failed to find it. By wrapping successful cases in such a tuple, we make it easy to understand which is which. Here's the implemented function:

```
lookup(_, {node, 'nil'}) ->
undefined;
lookup(Key, {node, {Key, Val, _, _}}) ->
{ok, Val};
```

```
lookup(Key, {node, {NodeKey, _,
Smaller, _}}) when Key < NodeKey ->
lookup(Key, Smaller);
lookup(Key, {node, {_, _, _, Larger}}) ->
lookup(Key, Larger).
```

And we're done. Let's test it with by making a little email address book. Compile the file and start the shell:

```
1> T1 = tree:insert("Jim
Woodland", "jim.woodland@gmail.com", tree:empty()).
{node,{"Jim Woodland","jim.woodland@gmail.com",
{node,nil},
{node,nil}}}
2> T2 = tree:insert("Mark Anderson", "i.am.a@hotmail.com",
T1).
{node,{"Jim Woodland","jim.woodland@gmail.com",
{node,nil},
{node,{"Mark Anderson","i.am.a@hotmail.com",
{node,nil},
{node,nil}}}}}
3> Addresses = tree:insert("Anita
Bath", "abath@someuni.edu", tree:insert("Kevin
Robert", "myfairy@yahoo.com", tree:insert("Wilson
Longbrow", "longwil@gmail.com", T2))).
{node,{"Jim Woodland","jim.woodland@gmail.com",
{node,{"Anita Bath","abath@someuni.edu",
{node,nil},
{node,nil}}},
{node,{"Mark Anderson","i.am.a@hotmail.com",
{node,{"Kevin Robert","myfairy@yahoo.com",
{node,nil},
{node,nil}}},
{node,{"Wilson Longbrow","longwil@gmail.com",
{node,nil},
{node,nil}}}}}}
```

And now you can lookup email addresses with it:

```
4> tree:lookup("Anita Bath", Addresses).
{ok, "abath@someuni.edu"}
5> tree:lookup("Jacques Requin", Addresses).
undefined
```

That concludes our functional address book example built from a recursive data structure other than a list! *Anita Bath* now...

Note: Our tree implementation is very naive: we do not support common operations such as deleting nodes or rebalancing the tree to make the following lookups faster. If you're interested in implementing and/or exploring these, studying the implementation of Erlang's `gb_trees`module (`otp_src_R<version>B<revision>/lib/stdlib/src/gb_trees.erl`) is a good idea. This is also the module you should use when dealing with trees in your code, rather than reinventing your own wheel.

Source : http://learnyousomeerlang.com/recursion