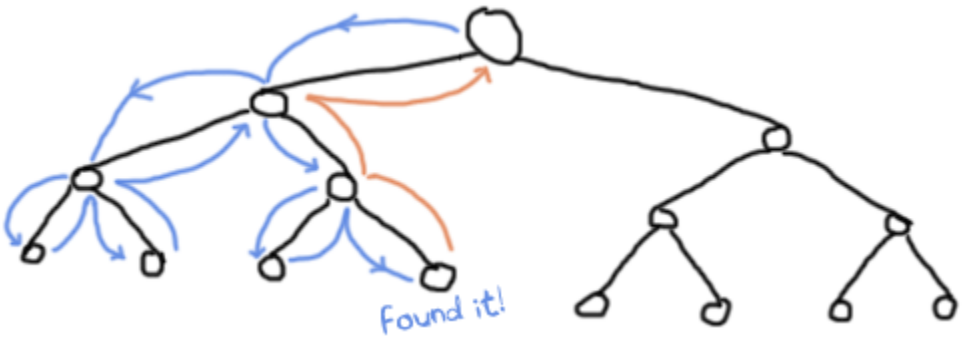
TRY A TRY IN A TREE

To put exceptions in practice, we'll do a little exercise requiring us to dig for our `tree` module. We're going to add a function that lets us do a lookup in the tree to find out whether a value is already present in there or not. Because the tree is ordered by its keys and in this case we do not care about the keys, we'll need to traverse the whole thing until we find the value.

The traversal of the tree will be roughly similar to what we did in `tree:lookup/2`, except this time we will always search down both the left branch and the right branch. To write the function, you'll just need to remember that a tree node is either `{node, {Key, Value, NodeLeft, NodeRight}}` or `{node, 'nil'}` when empty. With this in hand, we can write a basic implementation without exceptions:

```
%% looks for a given value 'Val' in the tree.
has_value(_, {node, 'nil'}) ->
false;
has_value(Val, {node, {_, Val, _, _}}) ->
true;
has_value(Val, {node, {_, _, Left, Right}}) ->
case has_value(Val, Left) of
true -> true;
false -> has_value(Val, Right)
end.
```

The problem with this implementation is that every node of the tree we branch at has to test for the result of the previous branch:

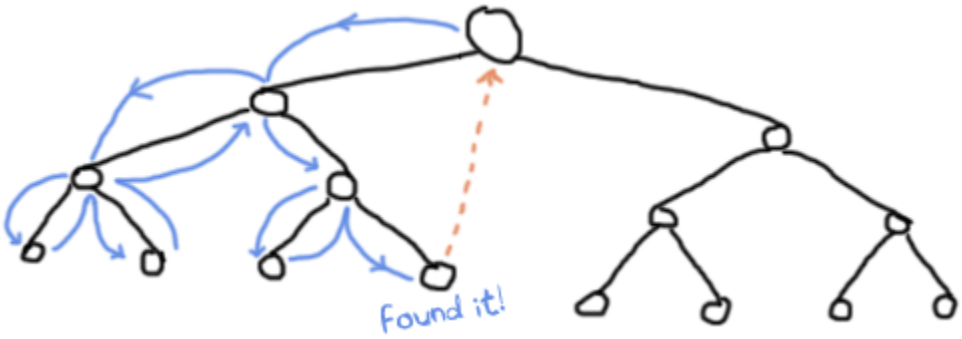


This is a bit annoying. With the help of throws, we can make something that will require less comparisons:

```
has_value(Val, Tree) ->
try has_value1(Val, Tree) of
false -> false
catch
true -> true
end.

has_value1(_, {node, 'nil'}) ->
false;
has_value1(Val, {node, {_, Val, _, _}}) ->
throw(true);
has_value1(Val, {node, {_, _, Left, Right}}) ->
has_value1(Val, Left),
has_value1(Val, Right).
```

The execution of the code above is similar to the previous version, except that we never need to check for the return value: we don't care about it at all. In this version, only a throw means the value was found. When this happens, the tree evaluation stops and it falls back to the `catch` on top. Otherwise, the execution keeps going until the last `false` is returned and that's what the user sees:



Of course, the implementation above is longer than the previous one. However, it is possible to realize gains in speed and in clarity by using non-local returns with a throw, depending on the operations you're doing. The current example is a simple comparison and there's not much to see, but the practice still makes sense with more complex data structures and operations.

That being said, we're probably ready to solve real problems in sequential Erlang.

Source : http://learnyousomeerlang.com/errors-and-exceptions