COMP5450 Assessment 4

Preamble

With this assessment you will learn how to use processes like a datatype. A little bit of Object-Oriented thinking may help too: a process identifier is a little bit like an object reference, and spawning a process is a little bit like creating an object reference using `new`. One important difference though is that processes can change their behaviour and the data it holds.

Your completed project should be submitted as a single Erlang file, with the name `a4_login.erl`, except that "login" should be your login, so I would be submitting `a4_smk.erl`.

The Data Type

The task is to implement binary search trees (a form of finite map) as trees of processes. Every tree node, including empty trees, is a process. The behaviour of these processes can evolve, so in particular each empty subtree is a different process.

We need two kinds of tree process behaviour: empty trees, and binary tree nodes. What they should *do* is dependent on the protocol which we will come to later. The data these behaviours carry (parameters of their functions) is as follows:

- empty trees: no data
- binary tree nodes: a key, a value (both of these are arbitrary Erlang values), and two tree nodes. One of the tree nodes contains in its expanded tree only nodes whose keys are smaller than this key, the other contains only nodes whose keys are larger. Note: "tree node" means here that they are process identifiers.

Note that in contrast to the binary trees you have seen earlier in the module (lecture 8) the tree nodes of this assessment carry the key *and* a value. This means that these trees implement finite maps (mapping keys to values), the trees from lecture 8 just realise sets of keys.

The Protocol

The trees get their meaning from the messages they can process. We have request messages, and corresponding respond messages. All tree nodes (empty or binary) need to be able to process all request messages. All messages, including all responses, should eventually be processed, i.e. messages should not pile up arbitrarily in mailboxes of processes. All request messages are tuples which have as their last component a process identifier, which is the process to which "the result" should be send.

We require the following request messages:

- `{is_empty,PID}`. This is for checking whether a node is an empty tree, or not. The responses should be `true` or `false`, reflecting what kind of node they are.
- `{get,K,PID}`. This message requests the value associated with key K in the tree. If there is no binary node with key K in the tree the response should be `nothing`. If there is, the response should be the message `{just,V}`, where V is the value kept in that tree node.
- `{put,K,V,PID}`. This places the key/value pair K/V in the tree; either by overwriting the current value of a node that already holds K, or — if the key K is fresh to the tree — by placing a fresh binary node with this key/value pair in an appropriate place of the tree. The response is an acknowledgement message: `done`.
- `{fold,FE,FB,PID}` It is somewhat annoying that, in order to increase the functionality of our trees, we need to keep changing their protocol. This `fold` message allows us to realise quite a few operations on trees. The FE component is a value, the FB component a function with 4 arguments. The response message for this should be of the form `{folded,RES,PID2}`, where PID2 is the process identifier of the responder. The value RES is computed as follows: for an empty tree it is simply `FE`; for a binary tree node it should be `FB(LV,K,V,RV)`, where K/V are key and value of the binary node, and LV and RV are the corresponding folded values of the left and right subtree of the node, respectively. This means that a binary node would need to send `fold` messages (with the same values for FE and FB) to both its subtrees, retrieve the values from their responses.

  As an example of how to use this: if we wanted to compute the number of binary nodes in the tree, we would send a message `{fold,0,FB,self()}` to our tree, with `FB=fun(L1,_,_,L2)-> L1+L2+1 end`.

Functions

In particular, the following functions should be written. You can add auxiliary functions or testing functions if you like.

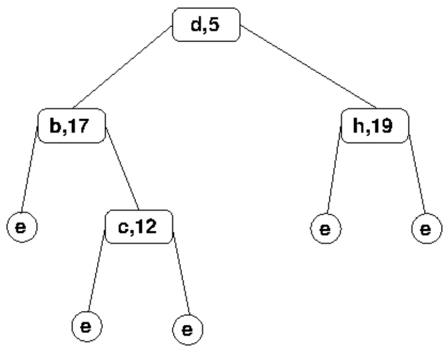
1. (25 marks): `empty_tree/0`. This is the behaviour of the empty tree.
2. (5 marks) `empty/0` This should create a fresh empty tree behaviour process and return its process identifier.
3. (50 marks) `binary_tree/4`. This is the behaviour of a binary tree node. When processing a `put` request, the call should not block access to the node until the key/value pair is in place. It is OK though to block the node while a `fold` is being processed.
4. (20 marks) It is somewhat awkward at the user level to engage with the tree via communication. Therefore, we want to hide the communications behind a functional interface, for each of the request messages. Write functions `get/2`, `put/3`, `is_empty/1` and `fold/3` that do just that. For all 4 functions, the first parameter is the tree. The second parameter for `get/put` is the key K, the third parameter for `put` is the value V. All functions (except `fold`) should return their response message as the result. The second and third parameter of `fold` are FE and FB, and it should return the RES component of the `folded` response.

Example

If everything is working, we could create a tree as follows:

`X=empty(), put(X,d,5), put(X,b,17), put(X,h,19), put(X,c,12).`

This should create a tree as in the picture. Notice that at this point 9 processes represent this tree, the 4 process nodes carrying a key and a value, plus the 5 empty tree processes. We could then call `get(X,c)`, which should evaluate



to {`just,12`}. A call of `fold(X,0,fun(L,_,V,R)->L+V+R end)` should sum all the values of the tree, i.e. return 53.