KEY-VALUE STORES

I've had you build a tree back a few chapters, and the use was to use it as a key-value store for an address book. That book sucked: we couldn't delete or convert it to anything useful. It was a good demonstration of recursion, but not much more. Now is the time to introduce you to a bunch of useful data structures and modules to store data under a certain key. I won't define what every function does nor go through all the modules. I will simply link to the doc pages. Consider me as someone responsible about 'raising awareness about key-value stores in Erlang' or something. Sounds like a good title. I just need one of these ribbons.

For small amounts of data, there are basically two data structures that can be used. The first one is called a*proplist*. A proplist is any list of tuples of the form `[{Key,Value}]`. They're a weird kind of structure because there is no other rule than that. In fact the rules are so relaxed that the list can also contain boolean values, integers and whatever you want. We're rather interested by the idea of a tuple with a key and a value in a list here, though. To work with proplists, you can use the proplists module. It contains functions such as `proplists:delete/2`,`proplists:get_value/2`, `proplists:get_all_values/2`, `proplists:lookup/2` and `proplists:lookup_all/2`.

You'll notice there is no function to add or update an element of the list. This shows how loosely defined proplists are as a data structure. To get these functionalities, you must cons your element manually (`[NewElement|OldList]`) and use functions such as `lists:keyreplace/4`. Using two modules for one small data structure is not the cleanest thing, but because proplists are so loosely defined, they're often used to deal with configuration lists, and general description of a given item. Proplists are not exactly complete data structures. They're more of a common pattern that appears when using lists and tuples to represent some object or item; the proplists module is a bit of a toolbox over such a pattern.

If you do want a more complete key-value store for small amounts of data, the orddict module is what you need. Orddicts (ordered dictionaries) are proplists with a taste for formality. Each key can be there once, the whole list is sorted for faster average lookup, etc. Common functions for the CRUD usage include `orddict:store/3`,`orddict:find/2` (when you do not know whether the key is in the dictionaries), `orddict:fetch/2` (when you know it is there or that it **must** be there) and `orddict:erase/2`.

Orddicts are a generally good compromise between complexity and efficiency up to about 75 elements (see my benchmark). After that amount, you should switch to different key-value stores.

There are basically two key-value structures/modules to deal with larger amounts of data: dicts and gb_trees. Dictionaries have the same interface as orddicts:`dict:store/3`, `dict:find/2`, `dict:fetch/2`, `dict:erase/2`and every other function, such as `dict:map/2` and`dict:fold/2` (pretty useful to work on the whole data structure!) Dicts are thus very good choices to scale orddicts up whenever it is needed.

General Balanced Trees, on the other hand, have a bunch more functions leaving you more direct control over how the structure is to be used. There are basically two modes for gb_trees: the mode where you know your structure in and out (I call this the 'smart mode'), and the mode where you can't assume much about it (I call this one the 'naive mode'). In naive mode, the functions are `gb_trees:enter/3`, `gb_trees:lookup/2` and `gb_trees:delete_any/2`. The related smart functions are `gb_trees:insert/3`, `gb_trees:get/2`, `gb_trees:update/3` and `gb_trees:delete/2`.

There is also `gb_trees:map/2`, which is always a nice thing when you need it.

The disadvantage of 'naive' functions over 'smart' ones is that because gb_trees are balanced trees, whenever you insert a new element (or delete a bunch), it might be possible that the tree will need to balance itself. This can take time and memory (even in useless checks just to make sure). The 'smart' function all assume that the key is present in the tree: this lets you skip all the safety checks and results in faster times.

When should you use gb_trees over dicts? Well, it's not a clear decision. As the benchmark module I have written will show, gb_trees and dicts have somewhat similar performances in many respects. However, the benchmark demonstrates that dicts have the best read speeds while the gb_trees tend to be a little quicker on other operations. You can judge based on your own needs which one would be the best.

Oh and also note that while dicts have a fold function, gb_trees don't: they instead have an *iterator* function, which returns a bit of the tree on which you can call `gb_trees:next(Iterator)` to get the following values in order. What this means is that you need to write your own recursive functions on top of gb_trees rather than use a generic fold. On the other hand, gb_trees let you have quick access to the smallest and largest elements of the structure with `gb_trees:smallest/1` and `gb_trees:largest/1`.
I would therefore say that your application's needs is what should govern which key-value store to choose. Different factors such as how much data you've got to store, what you need to do with it and whatnot all have their importance. Measure, profile and benchmark to make sure.

Note: some special key-value stores exist to deal with resources of different size. Such stores are ETS tables, DETS tables and the mnesia database. However, their use is strongly related to the concepts of multiple processes and distribution. Because of this, they'll only be approached later on. I'm leaving this as a reference to pique your curiosity and for those interested.

Update:
Starting with version 17.0, the language supports a new native key-value data type, described in Postscript: Maps. They should be the new de-facto replacement for `dict`s.

Source : http://learnyousomeerlang.com/a-short-visit-to-common-data-structures