Key-value Pairs in Lists & Binary Trees

William Frid

Spring 2024

Introduction

When working with labeled data, key-value is one of the possible options. But assume you have multiple pairs and wish to store them together. How would you go about doing that? One option would be to put the key-value pairs in a list, which isn't hard to implement but slow when scaling. Another faster - but somewhat more complicated - option would be to sort the pairs in a binary tree. Key-value pairs stored in a list or binary tree like this are something called "key-value database", or simply a "map".

Note that there is built-in support for maps in Elixir (based on C++), but this report will focus on coding them by hand for a deeper understanding. This report covers the construction of key-value pair **lists** and key-value pair **trees** (binary). The implementation of these two data structures will be done somewhat differently compared to if we were to do it with other programming languages thanks to Elixir's powerful pattern matching.

Structure

The structure for the list and binary tree will be the same. That is, they will both have the functions `add/?`, `lookup/?`, `remove/?`. Note that the list will also have a function called `new/0` as this is needed for its structure (optional). Also, note the "`/?`" representing that the functions do not have a set amount of arguments due to function overloading and differences between the list and binary tree implementation.

List

To minimize the amount of code and logic in each of the three functions (excluding `EnvList.new/0`), each comes in two versions. One where it handles an empty list, and one where it handles the other cases (non-empty). The logic for the three functions are all somewhat similar, thus this report will be focusing on `EnvList.add/3` as seen below.

Assume we were to call upon add with an empty list, like so: `lst = EnvList.add(EnvList.new(), :cat, 10)` This would call the first `EnvList.add/3` which would generate a new empty list, assign a key-value pair and store this in the variables `lst`.

Now assume we would call `lst = EnvList.add(lst, :dog, 15)`. This time, the second function would respond as the list is no longer empty. Inside the function, the head and tail of the list would be split up into two followed by the head being split up into key and value. That key is then compared to check if the value should be updated. But since it doesn't match, the function continues into the "else"-section where it combines the head, with a recursive call, adding to the tail. This time, the first function is called again, returning the new element alone in a new list, which then is combined with the head and returned as a final list.

Finally, we call `lst = EnvList.add(EnvList.new(), :cat, 2)`. This will be picked up by the second function where the first if-statement will be true, which returns the whole list with the first key-value pair being updated.

```
def add([], key, value) do [{key, value }] end
def add([h | t], key, value) do
    {k, v} = h
    if k == key do # Update current.
        [{k, value} | t]
    else # Dive deeper.
        [h | add(t, key, value)]
    end
end
```

Both the function `EnvList.lookup/2` *and* `EnvList.remove/2` *returns* `nil` *upon not finding target element.*

Binary Tree

Assume that the animal-shelter example was to expand into having thousands of key-value pairs. Then a list would no longer be the optimal choice for finding certain pairs. This is where a binary tree comes in.

The binary tree is a well-known data structure that can be tedious to build with the wrong language (Java for instance). But its logic becomes drastically less complex (and requires less writing) thanks to Elixir's pattern matching. Take for example `EnvList.add/3` (see snippet below). Using only a small amount of code, the program can easily add to a binary tree (incl. an empty one) in the correct order.

Assume we start with `tree = EnvTree(tree, :b, :foo)`. This would be handled by the first function due to `tree` being empty. Running the

same code again, only this time with a different value, would be caught by the second function that updates the value (note how `keya` accrues twice in the parameters).

The last two functions act as navigation functions. This means that if we were to call for instance `tree = EnvTree(tree, :a, :foo)`, then the third function would handle the call thanks to its "when"-statement (`when key < k do`) which would update the first node's left branch, with a new (recursive) call that would return simply the new node.

```
def add(nil, key, value) do
    {:node, key, value, nil, nil} end
def add({:node, key, _, left, right}, key, value) do
    {:node, key, value, left, right} end
def add({:node, k, v, left, right}, key, value) when key < k do
    {:node, k, v, add(left, key, value), right} end
def add({:node, k, v, left, right}, key, value) do
    {:node, k, v, left, add(right, key, value)} end
```

Benchmark

The benchmark used for comparison was run 100000 times and calculated a benchmark based on the average time in random scenarios. No code will be shown here due to lack of space and the fact that the code was handed to the students. It is however available on GitHub in the repository "ID1019_VT19_TCOMK_Programming_2".[1]

Looking at table 1 we see that all operations for the list seem to have a time complexity of $O(n)$. While table 2 shows on a $O(logn)$ for all operations. Meaning that the binary tree outperforms the list on all fronts, and thus should be used over a list unless the list is very short (compare the first row of the two tables).

[1] Fridh William, 2024. ID1019_VT19_TCOMK_Programming_2. GitHub. https://github.com/williamfridh/ID1019_VT19_TCOMK_Programming_2.

n	add	lookup	remove
16	0.21	0.19	0.07
32	0.35	0.32	0.12
64	1.40	0.93	0.24
128	1.37	1.33	0.46
256	2.58	2.46	0.75
512	5.28	5.05	1.80
1024	10.32	9.79	3.01
2048	22.28	18.94	6.33
4096	44.22	38.71	15.58
8192	94.02	75.15	31.31

Table 1: Benchmark of list.

n	add	lookup	remove
16	0.21	0.25	0.14
32	0.20	0.16	0.14
64	0.27	0.25	0.17
128	0.23	0.20	0.18
256	0.50	0.23	0.21
512	0.25	0.22	0.27
1024	0.40	0.29	0.37
2048	0.34	0.30	0.32
4096	0.47	0.36	0.38
8192	0.45	0.34	0.46

Table 2: Benchmark of binary tree.

Discussion

The simple benchmarks used for this report should have been more thorough for clearer and more trustworthy results. However, at this point (as this report is part of a program at KTH Royal Institute of Technology), we're already well aware of the speed the binary tree offers. What differs in this case is the key-value pairs which do affect the speed, but not time complexity.

Lastly is the bullet in map support which is based on C++. This built-in support for maps should in theory outperform both of your implementations thanks to the higher speed of C++ compared to Elixir. This will however not be tested in this report. It is however worth mentioning that the first code written for this report did use it, but was later removed as the true goal was to hard-code the two different types of maps.

Conclusion

Constructing more complex data structures using Elixir is more efficient than with some other languages thanks to the way it works. Constructing a binary tree is especially simple and still outperforms the typical list thanks to its binary nature. They are, and will remain more appealing options when the amount of data grows.