ID1019: An environment

Alexander Lundqvist

Spring Term 2023

Introduction

In this task we are supposed to create a key-value database, otherwise known as a "map", in two different ways where each implementation has to have a certain interface. The first implementation will be based on a list structure while the second will utilize a tree structure. The implementations must have functions for creating a new map, adding new key-value pairs, removing key-value pairs and looking up an value with its associated key. Furthermore we will perform benchmarks on the different implementations to compare their performance.

Method

To start this assignment I watched the videos on Canvas regarding recursion and trees. I also went back to my material from the course *ID1021 Algorithms and data structures* to refresh my memory on lists, trees and key-value based data structures as well as benchmarking functions. I also had to read in the elixir language documentation as well as the information on elixir website during the development. The benchmark program was mostly based on the one found in the course repository.

Result

I will omit most of the code in this section and just present the general solution and the results from the benchmarking. The entire code is available to read at this github repository.

List implementation

For the list implementation I ended up creating three variations for each method where the idea was to first catch empty lists and then the following statements would be kind of a switch case that would check whether a certain key or key-value pair was present in the list and execute depending on the answer. The following snippet shows the lookup method as an example.

```
def lookup([], _key) do nil end

def lookup([{key, value}|tail], key) do value end

def lookup([head|tail], key) do lookup(tail, key) end
```

Tree implementation

The tree implementation required a bit more code and the methods differ somewhat from the list implementation since now we have to deal with a node structure. Again, we use the lookup method as an example. Since there are more methods and they are longer I won't write it all here. Here we can at least see the node structure being used. With the pattern matching in the arguments we can easily find the target key, if it exits.

```
def lookup(nil, _key) do nil end

def lookup({:node, key, value, _left, _right}, key) do
  {key, value}
end
```

Benchmark

The following is a short presentation of the results from the benchmark. The benchmark code is omitted since I stated previously that it was based heavily on the one in the course repository. When performing numerous tests on my implementations with varying list sizes n as well as Elixir's built in Map module we got the following results. Note that all time measurements are in microseconds! The following table describes the run time for the add operation.

The following table describes the run time for the lookup operation.

The following table describes the run time for the remove operation.

n	**EnvList**	**EnvTree**	**Map**
16	0.13	0.10	0.12
32	0.12	0.08	0.10
64	0.31	0.14	0.07
128	0.35	0.16	0.06
256	0.60	0.18	0.06
512	1.37	0.27	0.06
1024	2.52	0.36	0.06
2048	5.30	0.45	0.07
4096	11.97	0.49	0.07
8192	23.09	0.62	0.08

Table 1: Run time of the add method between the different implementations. The size of the data structure is denoted by n.

n	**EnvList**	**EnvTree**	**Map**
16	0.08	0.09	0.03
32	0.07	0.04	0.03
64	0.14	0.07	0.04
128	0.16	0.05	0.03
256	0.32	0.08	0.05
512	0.58	0.10	0.04
1024	1.09	0.12	0.04
2048	2.15	0.13	0.04
4096	4.47	0.16	0.05
8192	8.99	0.18	0.05

Table 2: Run time of the lookup method between the different implementations. The size of the data structure is denoted by n.

n	**EnvList**	**EnvTree**	**Map**
16	0.03	0.06	0.03
32	0.01	0.01	0.03
64	0.02	0.02	0.03
128	0.01	0.02	0.03
256	0.01	0.01	0.02
512	0.01	0.01	0.02
1024	0.02	0.01	0.02
2048	0.02	0.01	0.02
4096	0.03	0.01	0.02
8192	0.06	0.01	0.02

Table 3: Run time of the add method between the different implementations. The size of the data structure is denoted by n.

Discussion

From the benchmark result we can clearly see that the list implementation has a hard time to keep up with increasing list sizes. We can also note that Elixir's Map has the overall best performance except for the remove method where there seems to be a tie. This is as expected as mentioned in the instructions, mainly because a hash table has a constant time complexity for both add, lookup and remove operations provided that the hashing function has an even distribution of the hashes. There is also the case that Elixir is a higher level language than C++ and inherently uses more memory resources. We can also argue while the Elixir implementations were made by one confused student hardly can compete with highly optimized code designed, written and maintained by an IT-professional.

In the assignment we are asked the question "Would it make sense to keep the list sorted?" and I think yes because a sorted list will inherently have better performance since the lookup procedure only have to look until it either finds the element or the following keys value is greater than the searched key. Unfortunately I missed this part when doing the first implementation and noticed it first when I was already writing the discussion. If it was required then I will of course re-do that part.

The task was a bit difficult to start with. I had a hard time to understand the theory that was taught in the Canvas videos so I took a sneak peak on the solution in the course github. It was like gibberish to me so I had to spend most of the week reading up on Elixir to get myself in the right frame of mind. After getting some clarifications on the syntax and conventions as well as re-watching some of the videos I think I got it. Still, the whole subject is a bit unclear and I kind of feel like I have to learn how to walk again. I do however start to appreciate the fact that you can use the "function overloading", that being multiple functions with the same name, as a switch-case statement.