Task 2 Key Value Database

Adrian Jonsson Sjödin

Spring Term 2023

Introduction

In this task we implement two different versions of a key-value database, that is used to look up the value associated with a key. The keys can be anything (atom, int, float, etc.), since we simply sort them using the regular "<" operator. Just keep in mind that in elixir the following apply:
`number < atom < reference < function < port < pid < tuple < map < list < bitstring`

Method

We started with implementing the map using a list since that seemed the easiest. To get started we first watched the prerecorded videos on recursion and trees. Especially the videos about recursion proved helpful since the addressed how one handles lists, and program recursively.

Having watched those we started with the `add/3` function, and the trick here was to cover all the base cases. For `add/3` being 1) add to an empty map, 2) the key we want to add already exists and 3) add a new pair to an empty map.

The code for `add/3` is surprisingly simple and can be seen in code overview 1.

Code Overview 1: `add/3`

```
def add([], key, value), do: [{key, value}]
def add([{key, _} | map], key, value), do: [{key, value} | map]
def add([head | map], key, value), do: [head | add(map, key, value)]
```

The first `add/3` covers the case when we have an empty map, the second the case when we want to change the value, and then we have the third one which will recursively call the `add/3` function and do either the first or second `add/3` on the tail that is passed along, or the third `add/3` again.

Both the `lookup/2` and `remove/2` functions work similarly. We first identify all the cases and then write the corresponding functions matching those cases. For `lookup/2` we have when the map is empty, when we've found the key, and when we still have'nt found it, and for `remove/2` we have the same cases.

The code for this can be seen in code overview 2

Code Overview 2: `lookup/2` and `remove/2`

```
def lookup([], _key), do: nil
def lookup([{key, _value}=pair | _], key), do: pair
def lookup([_ | map], key), do: lookup(map, key)
def remove([], _), do: nil
def remove([{key, _} | map], key), do: map
def remove([head|map], key), do: [head | remove(map, key)]
```

Starting on the tree implementation, quite a lot of code was given in the task description, and it was simply a matter of filling in the blanks. Because of this and the fact that the process was the same as described above, we will not go through the code here, but there will be a link at the end to where the code can be found.

Finally we had to implement a benchmark to measure the performance of our two implementations. Yet again this was simple a matter of following the instructions since the complete code for a benchmark of the list implementation was given. The only thing changed was some of the names for clarification. This benchmark was just for the list implementation but works for the tree implementation as well, if one changes the function being called.

Result

Table 1 shows the result from our benchmark. n denotes the size of the data structure.

	List			Tree		
n	add	lookup	remove	add	lookup	remove
16	0.56	0.23	0.38	0.22	0.16	0.51
32	0.86	0.31	0.49	0.42	0.20	0.30
64	1.25	0.50	1.14	0.37	0.25	0.37
128	2.51	1.06	2.33	0.50	0.29	0.55
256	5.10	1.70	4.81	0.59	0.32	0.55
512	5.75	1.56	4.10	0.64	0.37	0.68
1024	6.44	1.77	4.88	0.95	0.43	0.81
2048	10.1	3.11	9.38	0.81	0.34	0.64
4096	20.2	5.66	16.9	0.58	0.29	0.51
8192	46.3	17.9	44.9	0.52	0.26	0.42
16384	71.4	21.0	69.3	0.48	0.21	0.36
32768	154	39.1	141	0.67	0.25	0.47

Table 1: Benchmark of list and tree implementation of a map.
Values in μs

Discussion

Looking at the result in table 1 we can clearly see that the tree implementation is faster than the list when the size of the data structure grows. The biggest difference is between the `add/3` and `remove/2` functions, where the tree implementation was 230 times and 298 times faster respectively. For the lookup function the difference was smaller but the tree implementation was still 156 times faster.

Furthermore we can see that the time complexity for the list implementations `add/3` function is approximately linear. This makes sense since when we add a new pair it is added at the end and we go through the list recursively until we reach the end. For the other two functions the time increases with the size but there it also depends on which key we're looking for. If it is closer to the head it will go faster than if it is at the end. So the time complexity in these cases would be $\mathcal{O}(\frac{n}{2}) = \mathcal{O}(n)$

However for the tree we have a time complexity of $\mathcal{O}(log(n))$ for all the operations, since the tree is sorted and we don't have to search through the whole tree. This is the reason why it is important that we sort the tree when we add our pairs, because if we did not the time complexity would be the same as in the list implementation and there would be no purpose of having the tree data structure.

All the code can be found here: GitHub.