Elixir map - key value database

Zak Ora

Spring Term 2023

Introduction

A database that stores a key and a value associated with that key is often referred to as a map, or a key-value database. The main task was to make a program in elixir that can create said database, remove desired keys and lookup the values related to a key. Furthermore, this was done in two different ways, first as a linked list structure, and then as tree structure. Lastly, these two implementations were benchmarked in relation to their time efficiency, in which the results are displayed at the end of the report together with graphs and tables.

List map

Creating the list map was quite hard in the beginning, mostly because coding recursively required different thinking compared to previous courses that used Java or C. However, after trying different approaches and doing some minor changes, it became fairly simple. Albeit the remove functions had to be remade a few times, since an implementation using the built-in functions hd(list) and tl(list) was tried. The idea was to recursively go through the list:

```
def remove([_|tail], key) do remove(tail, key) end
```

Until the second function clause was matched and returned a list without the specified key, using the hd(list) function:

```
def remove([{key, _}|tail], key) do [hd([{key,_}|tail])|tail] end
```

However, this did not work as intended as the hd(list) function was misunderstood. Instead of returning a list with all elements before the key, and all elements after, it instead returned a list with the first (head) element and the keys tail. Thus, the remove functions were remade into the code below instead:

```
def remove([], _) do nil end
def remove([{key, _}|tail], key) do tail end
def remove([head|tail], key) do [head|remove(tail, key)] end
```

Tree map

Instead of inserting every key in a list, a map was also created with a tree structure, where keys to the left have a smaller value, and keys to the right have a larger value. This was done with the keyword `when key < k` as seen in the code below:

```
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
```

These two clauses complement each other by travelling as far left as possible when the added node has a smaller key than the existing node. When the case `key < k` is not met, the other clause is used to travel to the right until the end of the tree (if it is not already to the far left) or if the key already exists in which the value is changed. Furthermore, the lookup method is a bit more complex than the list implementations, since the algorithm has to traverse the tree and find the key. This was done by using the `cond` keyword, which compared the two nodes to each other:

```
def lookup({}, _) do nil end
def lookup({:node, key, value, left, right}, key) do {key, value} end
def lookup({:node, key, value, left, right}, keyFind) do
  cond do
    keyFind < key -> lookup(left, keyFind)
    keyFind > key -> lookup(right, keyFind)
  end
end
```

Lastly, the remove function was much more challenging compared to the linked list equivalent, since it has to traverse the tree and find the key. This was completed with the line `when key < k`, which was also used in one of the add clauses, but also with the help of a new function `leftmost` which updates the values of the variables `key, value, rest`.

Benchmarks

The benchmarks for both implementations are shown below in the form of tables and graphs.

Linked List

n	Add	Look up	Remove
16	0.11	0.04	0.06
32	0.13	0.05	0.11
64	0.31	0.07	0.24
128	0.57	0.12	0.54
256	1.26	0.21	1.13
512	2.35	0.40	2.31
1024	5.17	0.82	4.95
2048	9.83	1.61	9.46
4096	19.98	4.87	19.94
8192	38.16	6.43	38.96

Table 1: Benchmark with 100 000 operations, where n is the amount of elements in the list and the time is measured in us

Tree structure

n	Add	Look up	Remove
16	0.09	0.03	0.07
32	0.11	0.04	0.10
64	0.17	0.05	0.12
128	0.16	0.05	0.14
256	0.21	0.06	0.18
512	0.24	0.07	0.19
1024	0.23	0.08	0.24
2048	0.27	0.09	0.27
4096	0.31	0.09	0.27
8192	0.34	0.11	0.34

Table 2: Benchmark with 100 000 operations, where n is the amount of elements in the list and the time is measured in us

From these results, one can see quite easily that the tree structure has a much faster execution time by about 10 times, especially for higher numbers

of n. Although, for smaller numbers (around 32 or less) the execution time is almost the same.