Environment

Simon Yllmark

January 2024

1 Introduction

In this assignment I implemented a key-value database, also called a ”map”. There is a module called Map in the Elixir system, It uses a trier data structure (a tree of hash tables) and is implemented in C++. In this assignment I created two homemade maps using a simple linked list and a tree which in this case is a linked list with two pointers. The two pointers help systematising the data as a tree so when doing lookup,remove and add functions we do not have to traverse the hole tree.

The two homemade maps must have the following interface:

- new() : return an empty map
- add(map, key, value) : return a map where an association of the key key and the data structure value has been added to the given map. If there already is an association of the key the value is changed.
- lookup(map, key) : return either key, value, if the key key is associated with the data structure value, or nil if no association is found.
- remove(key, map) : returns a map where the association of the key key has been removed

They could have other functions if I wish to expand on the idea but that is outside the scope of this assignment. This will be useful later when I will take benchmarks of the two homemade maps and the Map module provide by the Elixir system.

2 List

I implemented a linked list that can store keys and it associated value. The idea in general is to remove and add nodes to a existing list recursively. Here is a code snippet from my implemented program to illustrate with an example of the idea behind the add(), lookup() and remove functions.

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

Step 1 if we would add a node to a empty list then we get back a list containing one node. Step 2 if we add a new node to the list with a key value that already exist in the designated list then the program update that value, Considering that objects in Elixir is immutable we rather get a new list. Step 3 if we want to add a node to a list with at least one element then the program would traverse the list until the it reaches the end of the list then the object map would point to an empty list and the program then goes back to Step 1.

3 Tree list

I implemented a double linked list that can store keys and it associated value. The advantage with a double linked list is that we can organise the nodes in a tree like structure, then the program can organise the nodes with the lowest value nodes to outer right of the tree and vise versa. Here is a code snippet from my implemented program to illustrate with an example of the idea behind the add(), lookup() and remove functions.

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

The idea behind this code have similarities to the code snippet from section List. Like in the List section there is a step 2 that updates values to an already existing key. The idea to step 1 and 3 are similar but in this case we can add a new node to the left or right a node in the designated list.

4 Bench

This section contains benchmarks of how efficient the 3 different "map" implementations work linked list, double linked list and Map module provided by the Elixir system in regards to 3 of the 4 function from the required interface as following:

- add(map, key, value) : return a map where an association of the key key and the data structure value has been added to the given map. If there already is an association of the key the value is changed.

- lookup(map, key) : return either key, value, if the key key is associated with the data structure value, or nil if no association is found.
- remove(key, map) : returns a map where the association of the key key has been removed

In figure 1 the result from the benchmarks can be viewed. The variable n represent the number of nodes added, lookup and removed. The time-unit is measured as the amount of second it take to iterate through all the nodes n per n, meaning time it takes(s) / number of nodes(n). According

Bench			
n	add	lookup	remove
	list tree map	list tree map	list tree map
16	0.41 0.41 0.10	0.31 0.00 0.10	0.10 0.31 0.10
32	0.61 0.10 0.41	0.10 0.10 0.10	0.20 0.20 0.20
64	0.82 0.41 0.10	0.31 0.10 0.00	0.61 0.41 0.10
128	1.02 0.31 0.00	0.31 0.10 0.00	0.82 0.31 0.00
256	1.74 0.51 0.10	0.41 0.10 0.00	1.54 0.20 0.10
512	3.58 0.31 0.10	0.92 0.10 0.00	3.79 0.20 0.20
1024	6.35 0.31 0.10	1.84 0.10 0.10	7.17 0.41 0.20
2048	15.46 0.31 0.10	2.76 0.10 0.10	12.08 0.31 0.20
4096	27.14 0.31 0.10	5.22 0.10 0.10	45.57 0.72 0.10
8192	53.35 0.51 0.10	11.47 0.20 0.10	49.25 0.41 0.20

Figure 1: Happy now

to the result from figure 1 there is a clear winner which is the Elixir Map module(map) a clear second which is the double linked list(tree) and at last place the linked list(list).

5 Conclusion

Considering that the Elixir Map module uses a trier data structure (a tree of hash tables) perform better than the 2 homemade "map" solutions is not surprising and the double linked list performed better then the linked list is not surprising because in the case of the linked list the program would have to traverse the hole list for lookup, add and remove while it becomes a $O(\log_2(n))$ operation where n is the number of nodes in the list.

What I found interesting in this assignment is how Elixir handles memory leak. Considering that object in this programming language are immutable meaning for example when we want to make changes to an already existing object a new copy has to be created. According to stack overflow "Erlang programs, on the other hand, can have thousands of independent heaps which are garbage-collected separately" meaning that the performance penalty is spread out over time which is good for cases like long-running applications.

According to this source we as the user should view the garbage collection as something that happens under the hood that is done automatically for us but this source also mentions the Elixir programs that I use for this course will be broken down into processes and each has a stack and a heap and those processes does not share data. The source also mentions this "Data is copied only when one process sends a message to another.". Apparently there is a lot that happens under the hood that we as the programmers do not get any notification of.