An Environment

Ruxandra-Stefania Tudose

Spring Term 2024

Introduction

This report aims to open up a window onto two different ways of storing the associated keys with the given values, namely first under the form of a list and then, under the form of a binary tree. Having said that, the goal is in the end to benchmark the two implementations and compare their performances.

The map under the form of a list

As far as I am concerned, the **key idea** that stands behind the implementation of the given methods is that the list has to always thought of as a head and tail. Unlike static programming when we can use a *for* loop to navigate through the data structure, this time we will use recursion merged together with the head and tail structure.

Therefore, the head will match to one element in the list at a time and the tail to the rest of the whole 'list' (a sublist in a way). This is best resumed by the following two lines of code:

```
def add([{key, _}|map], key, value) do  [{key,value}|map] end
#if the key I want is NOT in the first tuple, then I have to keep on looking
def add([ass | map], key, value) do [ass | add(map, key,value)] end
```

In the second code line, the variable map is a sublist of the remaining tuples that have not yet been checked in the original list. If it is empty, the base case will be executed. Otherwise, the key in the head (first tuple of the new list - more of a sublist) of the map will be tried to be matched. If there is nothing to be updated, the new tuple will be added to the list.

This is one mistake I initially did:

```
def add(map, key, value) do map ++ [{key, value}] end
#this doesn't work because I cannot check if the key already exists
#it simply appends the tuple to the list => duplicate keys
```

Another important thing: the *lookup* method has nothing to do with the *add* method in spite of having to check the already existence of a key! As previously mentioned, pattern matching will be used when adding a new element in order to eventually update the value. As far as the implementation is concerned, the *lookup* method is quite similar to the *add* method.

Sorted data is useful when dealing with algorithms such as binary search when efficiency comes from the possibility of being able to 'jump' to the middle using the index. Since in Elixir we cannot do that and above all it a simple linked list (probably a doubly linked list would be needed to move back and forth), it doesn't make sense to keep the list sorted.

Walking through the implementation of *remove* method with an example

The implementation of the *remove* method consists in two base cases and one recursive step as follows:

- base case 1: when the list is empty;
- base case 2: when the element we want to delete is the first in the list/sublist we are looking at;
- the recursive step: when the element we are looking for is not on the first position, therefore we have to go further and go through the same procedure with the remaining list's tail;

```
#the original list - we want to remove key = 10
    //list = [{11,2}, {10,5}, {13,4}]
#check if it's in the first tuple - it's not => go further
    def remove([ass | map], key) do [ass |remove(map, key)] end
    //map = [{10,5}, {13,4}]
#the key is now in the first tuple;
    def remove([{key,value} | map], key) do map end
#stick to the map without the first tuple = that sublist's tail!
#go back recursively => the copy of the list without key = 10
    list = [{11,2}, {13,4}]
```

The map under the form of a tree

The *add* and *lookup* methods are more or less similar to the list's implementation. The most significant difference is that in order to implement the map under the form of a tree, we have to keep in mind that the smaller keys will always be placed to the left, while the greater ones will be found to the right. One such example is as follows, when the node has both left

and right branches, the keys don't match, but it is greater than the node's so we choose the right direction, while keeping the left one as it was.

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

Since these concepts are pretty much familiar by now, I will focus more on the *remove* function since it involves a slightly more complex approach, which can be split into the following main steps:

- first, setup the base cases: when the list is empty and when the keys match while one of the two branches is :nil;
- if both branches are present and the keys don't match, correctly identify the node to be remove; depending on the key being smaller or greater than the key we are looking at, we will go left or right (similar to *lookup* or *add*);
- then, choose that node's right branch and look for the leftmost node;
- the leftmost node is identified when we reach :nil on the left regardless what is on the right;
- once the leftmost node is found, pattern matching is used to place the new identified key, value and the right part of the remaining subtree, keeping the left branch as it was;

Below, I have chosen the part of the *remove* method which seemed the trickiest for me, namely when we have to go on the right branch and use pattern matching to remove the node by replacing it with the leftmost one.

```
#when the keys match to a node with both right and left branches
def remove({:node, key, _, left, right},key) do
  {key, value, rest} = leftmost(right)
#pattern matching is used - rest is the identified subtree of the leftmost node
  {:node, key, value, left, rest}
#no pin operator
#above it is the key and the value of the leftmost node and also
#the tree with the removed desired node
end
```

Benchmark Analysis

It should be taken into account that the benchmark results have been executed for *100000 operations*.

n	addtree[us]	lookuptree[us]	removetree[us]
16	0.08	0.04	0.05
32	0.08	0.05	0.07
64	0.12	0.05	0.08
128	0.11	0.06	0.10
256	0.16	0.08	0.12
512	0.16	0.09	0.15
1024	0.2	0.10	0.17
2048	0.17	0.10	0.17
4096	0.33	0.15	0.29
8192	0.38	0.20	0.39

Table 1: Benchmark analysis for the tree implementation.

n	addlist[us]	lookuplist[us]	removelist[us]
16	0.08	0.04	0.06
32	0.10	0.07	0.09
64	0.17	0.10	0.13
128	0.32	0.18	0.27
256	0.56	0.33	0.50
512	1.02	0.66	1
1024	2.2	1.4	2.2
2048	4.1	2.5	4.1
4096	9.7	6.4	9.8
8192	18	10.8	17.4

Table 2: Benchmark analysis for the list implementation.

Conclusion

All in all, as it can be noticed in the above tables, the map under the form of a tree is undoubtedly more efficient since as the number of elements doubles, for all three operations, the execution time is rather constant (varies very little). On the other hand, as far as the list implementation is concerned, the execution time approximately doubles with the number of elements.