Lab 2 - A key-value data base

Weichen Chai

Spring 2023

Introduction

This paper seeks to answer the questions presented in A key-value data base for ID1019. In this assignment, we will explore how the functional language Elixir can be utilized in a key-value database. We will work with both lists and trees, and in the end handle a benchmark.

Tasks

a map as a list

In this task, the assignment asks for us to implement EnvList, which contains: new, add, lookup, and remove functions. While it may sound complex, it is not so. add, remove and lookup all share the same three states, so an example of add will be given below

```
def new() do [] end //Create new empty list
//Taking add as an example
...
def add([], key, value) do [{key, value}] end
//when empty
def add([{key, _} | tail], key, value) do [{key, value} | tail] end
//when key is same, simply change value
def add([ite|map], key, value) do [ite|add(map, key, value)] end
//When key is added
```

Similarly, the three implementations in lookup and remove are when the map is empty, and when key is and isn't in the map. Through running a test on this function, ie. defining a list and attempting to remove and add things to it.

a map as a tree

Since using a list implementation is slower than a tree operation for add, remove and search in cases where there are a larger number of items, we attempt to implement the tree where an empty tree can be represented as nil. Following the structure given in the assignment, we implement add, remove and search functions in tree structure. We are given the general structure of the add and remove function that we should follow, which is finished below:

```
def add(nil, key, value) do
{:node, key, value, nil, nil} end // adding to empty tree

def add({:node, key, _, left, right}, key, value) do
//value in tree doesn't matter when replacing
{:node, key, value, left, right} end
//Place new value to tree and keep left and right.

def add({:node, k, v, left, right}, key, value) when key < k do
{:node, k, v, add(left, key, value), right}
// New tree returned with a new left branch
else
{:node, k, v, left, add(right, key, value)}end    //right "updated"
```

The updating of the left and right branches has been combined for clarity. The search or lookup function in a tree is quite similar to the add function that we have implemented above, while the remove function has some differences, looking at the structure of the remove function given in the assignment, we can write it as:

```
def remove(nil, _) do nil end
//empty tree
def remove({:node, key, _, nil, right}, key) do right end
//no left branch
def remove({:node, key, _, left, nil}, key) do left end
//no right branch

def remove({:node, key, _, left, right}, key) do
//if the node has both left and right branches
  {key, value, rest} = leftmost(right)
  //return left most branch of the right node to the node we remove
  {:node, key, value, left, rest}
end
def remove({:node, k, v, left, right}, key) when key < k do
//go down the branches if key is smaller than current.
```

```
    {:node, k, v, remove(left, key), right}
  end
  def remove({:node, k, v, left, right}, key) do
  //go right if larger
    {:node, k, v, left, remove(right, key)}
  end
```

As shown above, the tricky part is to find the key and replace it with the left most key value of the right branch, which requires the addition of the leftmost function. The leftmost function is quite simple to implement:

```
def leftmost({:node, key, value, nil, rest}) do
//return leftmost nodes' value, key and right branch.
{key, value, rest} end

def leftmost({:node, k, v, left, right}) do
//recursive
   {key, value, rest} = leftmost(left)
   {key, value, {:node, k, v, rest, right}}
 end
```

Now, that everything has been completed, a test was run and the results were indeed correct.

benchmark

In this benchmark we are asked to test the implementations with growing number of elements and measure the time differences between them through the usage of the timer module in Erlang. The benchmark is given to us in the task, and we can start to test it with the two implementations, list, and tree separately. They will be run 10000 times and we will find the average in [us]

As shown above, the time taken for tree is generally O(lg n) and the time complexity for lists is O(n), which is as expected, the tree structure is visibly faster than list in both add and lookup, and still some what faster if the number of elements get larger in remove, and this falls inline with the results that we have received in the Algorithms and data structure course. In this assignment, another task was to compare the answer with a inbuilt Map module which comes with the Elixir system. Similarly, it uses put, get and delete functions, which is tested as below.

The in-built key-value store implementation in Elixir clearly does better than both implementations, this may be due to the nature of the Trie structure used by the Map module, since the Trie structure also acts as a tree, it will also have the time complexity of O(lg n), However, due to the nature of the way it stores data, it will be faster than tree.

n	add		lookup		remove	
	list	tree	list	tree	list	tree
16	0.08	0.08	0.06	0.06	0.06	0.07
32	0.18	0.1	0.07	0.05	0.1	0.11
64	0.31	0.17	0.15	0.06	0.21	0.12
128	0.6	0.19	0.24	0.07	0.4	0.17
256	1.01	0.16	0.57	0.08	0.77	0.21
512	1.7	0.19	0.89	0.19	1.61	0.21
1024	3.31	0.23	1.83	0.12	3.08	0.24
2048	6.73	0.25	3.81	0.16	6.87	0.26
4096	15.35	0.41	8.88	0.15	22.39	0.33
8192	56.97	0.59	17.43	0.19	54.75	0.68

n	add	lookup	remove
16	0.05	0.03	0.03
32	0.07	0.03	0.05
64	0.12	0.03	0.08
128	0.1	0.04	0.08
256	0.12	0.04	0.13
512	0.1	0.04	0.08
1024	0.09	0.04	0.08
2048	0.13	0.05	0.09
4096	0.13	0.05	0.11
8192	0.19	0.05	0.21