An environment

Sareh Jalalizad

Spring Term 2023

Introduction

For this assignment, we were supposed to create two different implementations of a key-value database (or "map") that could be used to look up the value associated with a given key. And both of these implementations had to have the following interface:

- new() to return an empty map
- add(map, key, value) adds/updates a given key-value pair to the map
- lookup (map, key) returns a key-value pair if finds the provided key otherwise nil
- remove(key, map) return a map where the key's association has been removed

Map as a list

If the map is to be small, it can be represented as a list of key-value tuples. The list below depicts a map where atoms :a and :b are connected to the numbers 1 and 2, respectively.

```
[{:a, 1}, {:b, 2}]
```

Only the overall solution and the benchmarking results are shown in this section, and parts of the code are not mentioned. The following code demonstrates how to add and remove items from lists.

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

I created three variations for each function in this implementation. The idea was to first catch empty lists and insert a key-value pair into an empty list. Once we have a list, we must iterate through the list to see if the key is present or not, and for this, the third function makes use of tail-recursion. With recursion, the lookup and delete functions can be implemented similarly.

```
def remove([], _) do [] end
def remove([{key, _}|t], key) do t end
def remove([h|t], key) do [h|remove(t, key)] end
```

Map as a tree

A tree structure would definitely be a better approach as the map gets bigger. Implementing the add and lookup functions are straight forward. First, I figured out the base cases and then how to recurse down either the right or left branch.

```
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, right} end
def add({:node, k, val, left, right}, key, value)
   if key < k do
    {:node, k, val, add(left, key, value), right}
   else
    {:node, k, val, left, add(right, k, v)}
end
end

def lookup(nil, _) do nil end
def lookup({:node, key, value, _, _}, key) do {key, value} end
def lookup({:node, k, _, left, right}, key) do
   if key < k do
        lookup(left, key)
   else
        lookup(right, key)
    end
end
```

Benchmark

In order to evaluate and compare the solutions, we had to finally implement a benchmark. Since the whole code for a benchmark of the list was provided, this was a straightforward matter of following the instructions. The run times for three operations (add, lookup, remove) are shown in the following table.

n	**add**	**lookup**	**remove**
16	0.5	0.22	0.33
32	0.53	0.31	0.44
64	0.68	0.46	0.52
128	0.74	0.51	0.65
256	0.88	0.64	0.71
512	1.48	1.06	1.25
1024	3.22	1.59	2.48
2048	6.79	3.72	5.54

Table 1: Benchmark of list implementation, run time in microseconds.

n	**add**	**lookup**	**remove**
16	0.37	0.15	0.46
32	0.42	0.18	0.49
64	0.34	0.20	0.53
128	0.47	0.24	0.51
256	0.56	0.29	0.55
512	0.68	0.27	0.58
1024	0.92	0.31	0.60
2048	0.85	0.34	0.63

Table 2: Benchmark of tree implementation, run time in microseconds.

Conslusion

Overall, this assignment was interesting. From the results, we can clearly see that the list implementation takes a longer time as list sizes grow. Although the work was initially challenging, it provided a good illustration of how a list and tree be used as a map.