An Environment

Axel Månson Lokrantz

Spring Term 2023

Introduction

The task for this assignment was to work with maps. A map can be described as a key-value database where each value is associated with a key. The map should be implemented using two different methods where the following interface is provided:

- `new()`: Returns an empty map.
- `add(map, key, value)`: Adds a given key-value pair to the map. If the key is already present in the map the value is updated.
- `lookup(map, key)`: Finds the given key and returns a key-value pair. If the key is not present the function returns `nil`.
- `remove(key, map)`: Returns a copy of the map without the given element.

The program was written in elixir, together with Axel Lystem at KTH.

The task

A map as a list

For smaller maps, a list of key-value tuples can come handy.

```
[{a:, 1}, {b:, 2}, {c:, 3}]
```

In the list atom `a:` is linked to value 1, `b:` to 2 and `c:` to 3. Through function overloading tuples can be added to the list as seen below.

```
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
```

The first function serves as a base case. We insert a key-value pair into an empty list. If the given key is present in the list the second function updates its value. The third function uses tail-recursion which is a special form of recursion where the last operation of a function is a recursive call. To put it simple, if we are given a key and a list we first have to iterate through the list to see if the key is present or not. The recursion continues until we either find the key or exhaust the list, in which case our base case adds the key-value to the list. The `lookup` and `remove` functions can be implemented in a similar fashion with tail-recursion. The only difference is that we for a function like `lookup` return a tuple and therefor do not need to recreate the entire list as a modified copy.

A map as a tree

For larger sets of data a tree structure might be a better alternative. Adding a value to an already existing key will, just as before, only update the value with the matching key as there cannot be any duplicate keys in the tree. The process is fairly straight forward.

```
 def add(nil, key, value) do
    {:node, key, value, nil, nil}
  end
```

The base case handles an empty tree in which case we create a root node with the given key and value.

```
def add({:node, key, _, left, right}, key, value) do
    {:node, key, value, left, right}
end
```

If the given key match the key of the current node we insert the new value.

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

If the key does not match the key of the current node we iterate either left or right until we either find the key or reach a leaf which is our base case and the value is added. The `lookup` function follows the same procedure, however, we do not need to recreate the tree as we only return the key-value pair or `nil` if not found.

Removing a node from the tree is more complicated as there are a few more things to consider. For the tree to remain sorted we follow an algorithm where we first locate the key we want to remove and then replace it with the leftmost key-value pair in the nodes right branch. To do this we implement a function called leftmost as seen below.

```
def leftmost({:node, key, value, nil, rest}) do
    {key, value, rest} end

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

the leftmost function is only called when the node we want to delete has a left and a right branch.

```
def remove({:node, key, _, left, right}, key) do
    {key, value, rest} = leftmost(right)
    {:node, key, value, left, rest}
  end
```

If the node we want to delete lacks a left branch, we return the right branch and vice versa.

```
def remove({:node, key, _, nil, right}, key) do right end
def remove({:node, key, _, left, nil}, key) do left end
```

Benchmark List

n	add	lookup	remove
8	0.1	0.01	0.03
16	0.1	0.1	0.03
32	0.4	0.15	0.02
64	0.6	0.15	0.02
128	0.9	0.2	0.02
256	1.7	0.3	0.02
512	3.5	0.6	0.02
1024	7.2	1.2	0.02
2048	15	2.5	0.02

Table 1: Benchmark for the list implementation where n represent a growing set of data. Run time in micro seconds.

The time complexity for the remove operation is $O(1)$, and $O(n)$ for add the operation since we have to traverse the list of n tuples. If the list was sorted the add operation would be faster, since we only have to search up until a certain value. However our worst case complexity will remain the same.

Benchmark Tree

n	add	lookup	remove
8	0.1	0.1	0.03
16	0.1	0.1	0.02
32	0.1	0.1	0.02
64	0.1	0.1	0.01
128	0.1	0.1	0.01
256	0.2	0.1	0.01
512	0.2	0.1	0.01
1024	0.3	0.15	0.01
2048	0.3	0.15	0.01

Table 2: Benchmark for the tree implementation where n represent a growing set of data. Run time in micro seconds.

The time complexity for a balanced binary search tree is equal to the height of the tree and becomes $O(log(n))$. If the tree is not balanced, the time complexity becomes $O(n)$.

The benchmarks are only slightly slower compared to the built in functionality of `Map` in elixir. The reason is that `Map's put(), get() and delete()` uses a tree of hash tables and is implemented in C++.