An environment

Programming II - Elixir Version

Johan Montelius

Spring Term 2023

Introduction

The name of this assignment might be confusing and it will only be clear later in the course why we call this assignment "An environment". What you should implement is a key-value database, also called a "map", that can be used to look up the value associated with a key. We don't make any assumptions about the keys and will simply compare them using the regular operators.

You should implement the map using two different techniques, both implementations should provide 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

We will in this assignment use keys of atoms and the values will be integers but the implementation should not take this for granted.

a map as a list

If we assume that the map will be small, we can represent it as a list of key-value tuples. The list `[{:a, 12}, {:b, 13}]` would then represent a map where the atom `:a` is associated with 12 and the atom `:b` is associated with 13.

Note - Elixir has a way of printing lists of binary tuples where the first element is an atoms. The above list is printed as `[a: 12, b: 13]`*. You can also use this syntax which makes it easier to write long sequences of key value pairs.*

In a module `EnvList`, implement the functions above and test that you can perform the different operations. Would it make sense to keep the list sorted?

a map as a tree

If the map grows larger the list implementation might not be our best option. A tree structure would probably be a better approach. You could implement you tree any way you want but why not choose a simple representation where an empty tree is represented by the atom `nil` and a node by the structure `{:node, key, value, left, right}`.

In a module `EnvTree`, implement the functions above and test that you can perform the different operations.

The tree should of course be sorted to make the lookup operation efficient. You don't have to implement a balanced tree but this would of course be something that one would need to consider.

The `add/3` and `lookup/2` functions are fairly straight forward to implement. Identify the base cases and then how to recurse down the right or left branch. Use the following skeleton code to get you started:

```
def add(nil, key, value) do
  ... adding a key-value pair to an empty tree ..
end

def add({:node, key, _, left, right}, key, value) do
  ... if the key is found we replace it ..
end

def add({:node, k, v, left, right}, key, value) when key < k do
  ... return a tree that looks like the one we have
  but where the left branch has been updated ...
end
def add({:node, k, v, left, right}, key, value) do
  ... same thing but instead update the right banch
end
```

Remember that we are in fact not "updating" the tree that we have but rather constructing a copy of the tree where we have added the new key-value pair.

Implementing `lookup/2` is very similar to the `add/3` function. The difference is of course that we are not building a new tree but returning the found key-value pair or `nil` if not found.

Implementing `remove/2` is slightly more tricky and you have to remember the algorithm how to do this. The idea is to first locate the key to remove and then replace it with the leftmost key-value pair in the right branch (or the rightmost in the left branch). Removing and returning the leftmost key-value pair should be simple so if you only can handle all special cases you should be up an running in a few minutes.

```
def remove(nil, _) do  ...  end
def remove({:node, key, _, nil, right}, key) do  ...  end
def remove({:node, key, _, left, nil}, key) do ... end
def remove({:node, key, _, left, right}, key) do
  ...   = leftmost(right)
  {:node, ..., ..., ..., ...}
end
def remove({:node, k, v, left, right}, key) when key < k do
   {:node, k, v, ..., right}
end
def remove({:node, k, v, left, right}, key) do
   {:node, k, v, left, ...}
end
def leftmost({:node, key, value, nil, rest}) do ...  end
def leftmost({:node, k, v, left, right}) do
  ... = leftmost(left)
  ...
end
```

benchmark

Now let's do some benchmarks to see how the implementations perform and compare to each other. We want to see how the different implementations work with a growing number of key-value pair so let's set up a benchmark where we first construct a map of a number of elements and then measure the time it takes to perform an operation. We will set up the benchmark so that we use the same key-value pairs for each of the implementations.

Measuring time can be done using the function `:timer.tc(fun)` (this is the Erlang module `timer` and we thus write it using the regular atom syntax). This function will take a function, call this function and return a tuple with the number of microseconds it took and the result. The argument to `tc/1` is a function and as you will see this is quite efficient.

Using functions as arguments we can for example use the function `Enum.each/2` to generate a list of random numbers. Try the following with different values

of n and i:

```
Enum.map(1..n, fn(_) -> :rand.uniform(i) end)
```

To build a key-value store using our `EnvList` module we can do as follows:

```
seq = Enum.map(1..i, fn(_) -> :rand.uniform(i) end)

list = Enum.reduce(seq,  EnvList.new(), fn(e, list) ->
              EnvList.add(list, e, :foo)
              end)
```

We first build a list of random numbers (from 0 to i) and then use this list as keys when we add dummy key-value pairs to an empty store.

Once we have a store of i pairs we can construct another sequence of n random numbers and use them when we benchmark for example adding a new pair:

```
seq = Enum.map(1..n, fn(_) -> :rand.uniform(i) end)

{tla, _} = :timer.tc(fn() -> Enum.each(seq, fn(e) ->
                  EnvList.add(list, e, :foo)
                  end)
                end)
```

Note that we here ignore the constructed store with the new pair, we simply add the item and then drop the result. In this way we can first build a store containing for example 64 entries and then measure the time it takes to add a thousand new pairs without growing the store.

We can now do the same with the other operations and combine everything in one function:

```
def bench(i, n) do
  seq = Enum.map(1..i, fn(_) -> :rand.uniform(i) end)

  list = Enum.reduce(seq,  EnvList.new(), fn(e, list) ->
                EnvList.add(list, e, :foo)
                end)

  seq = Enum.map(1..n, fn(_) -> :rand.uniform(i) end)

  {add, _} = :timer.tc(fn() ->
                Enum.each(seq, fn(e) ->
                  EnvList.add(list, e, :foo)
```

```
                    end)
                  end)

    {lookup, _} = :timer.tc(fn() ->
                    Enum.each(seq, fn(e) ->
                      EnvList.lookup(list, e)
                      end)
                    end)

    {remove, _} = :timer.tc(fn() ->
                    Enum.each(seq, fn(e) ->
                      EnvList.remove(list, e)
                    end)
                   end)

    {i, add, lookup, remove}
  end
```

The `bench/2` function is then used repeatedly with growing values of i. If we print out the result we will have a nice sequence of numbers to plot or present in a table.

The specification `12.2f` *means print a float with 2 decimals right aligned using a width of 12 characters. Change it to anything that suits your needs.*

```
  def bench(n) do

    ls = [16,32,64,128,256,512,1024,2*1024,4*1024,8*1024]

    :io.format("# benchmark with ~w operations, time per operation in us\n", [n])
    :io.format("~6.s~12.s~12.s~12.s\n", ["n", "add", "lookup", "remove"])

    Enum.each(ls, fn (i) ->
      {i, tla, tll, tlr} = bench(i, n)
      :io.format("~6.w~12.2f~12.2f~12.2f\n", [i, tla/n, tll/n, tlr/n])
    end)
  end
```

Once you have your benchmark up an running you can compare your two implementations of the key-value store. You might also wan to see how well you did compared to the key-value store that comes with the Elixir system form start. There is a module called `Map` that does exactly what you want and more (the functions are called `put/3`, `get/2` and `delete/2`). It uses a trier data structure (a tree of hash tables) and is implemented in C++ so it is not strange if it performs better than your solution but my guess is that you're not that far off.

Take a look at the documentation of the `Map` module. From now on you can use it if you want.