An environment

Emil Göransson

Spring Term 2023

Introduction

This report will cover the implementation of a program that is able to generate and represent key-value databases of the linked-list and binary tree type. Towards the end of the report a benchmark to compare the two is done.

Linking a list so that it becomes a map

The key-value database (a map) is used to match a key to a value. The list-implementation will use the [{:atom, key, value}] "syntax". The map has the following functions:

- add/2 : Makes it possible to add a value to the list. The keys are unique so adding a key,value pair to the map that already contains the specific key will instead update the "old value" to the "new value"
- new/0: Returns an empty list [].
- lookup/2 : Given a list and a key , returns the key- value pair.
- remove/2 : given a list and a key, returns a copy of the list without the key- value pair.

The implementation

All the implementation mainly uses pattern-matching. The implementation of add/2 is done in the following way:

```
    def add([], key, value) do #1
    [{key, value}]
  end

  def add([{key, _} | t], key, value) do #2
    [{key, value} | t]
  end

  def add([h | t], key, value) do #
    [h | add(t, key, value)]
  end
```

remove/2 works in an similar however instead of adding a value we just toss it by just returning the tail of the list.

lookup/2 is implemented like the following:

```
  def lookup([], _) do
    nil
  end

  def lookup([{cKey, value} | t], key) do
    if cKey == key do
      {cKey, value}
    else
      lookup(t, key)
    end
```

here traverse down the list one by one in a linear manner and check if the candidate key matches the key. If it does we return the cKey, value pair otherwise we check the next element in the list using recursion.

The binary tree map

In this implementation I ignore the [] syntax and only use curly brackets. Every node is represented like the following: {:node, key, value, left,right} where both left and right are :nodes. A simple leaf is represented like this: {:node, key, value, nil, nil}.

the map has the same functionallity as the previous list implementation.

- add/3 : Makes it possible to add a value to the tree. The keys are unique so adding a key,value pair to the map that already contains the specific key will instead update the ”old value” to the ”new value”
- lookup/2 : Given a tree and a key , returns the key- value pair.
- remove/2 : Given a tree and a key, returns a copy of the tree without the key- value pair. Uses a helper-function leftmost/1 to traverse the tree.

The most complex one of the three are by far the remove function so I will focus on that one

Implementing a binary tree map

The function works in such a way that it first using pattern matching checks if the key is found in the root of the tree. If not it will traverse either to the left or the right depending on if the value of the key you want removed (rKey) is smaller than the key of the root. This way it makes it way down the tree until it find a node with a key that matches the rKey. When the correct key has been found there are a couple of cases. Either we are lucky and the node with the matching key has no branch left / right and we just return left or right depending on the case. Otherwise the more common operation is that the node has both a right and left subtree. This means that the helper function leftmost has to come to action. the leftmost function basically traverses down the subtree and moves the values so no values are lost and the tree is ”re balanced”.

the leftmost/1 code is the following:

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

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

and the specific remove/2 function that uses leftmost/1 is:

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

Benchmarking the two

By using the code provided in the task instructions and running both EnvTree.bench(10000) and EnvList.bench(10000) we get the following results for add, lookup, remove (order) when 10000 operations are done:

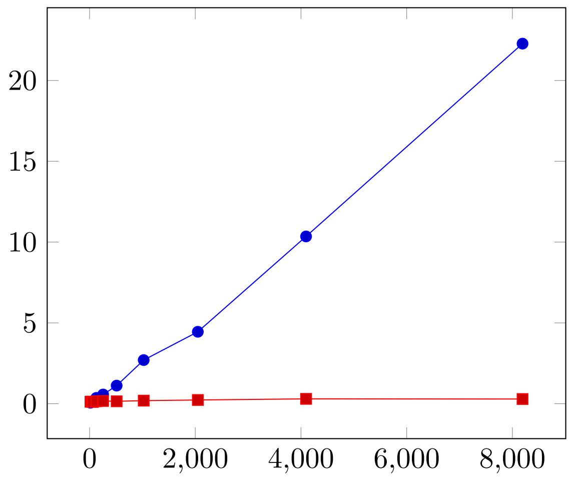

Figure 1: tree.add (red), list.add (blue)

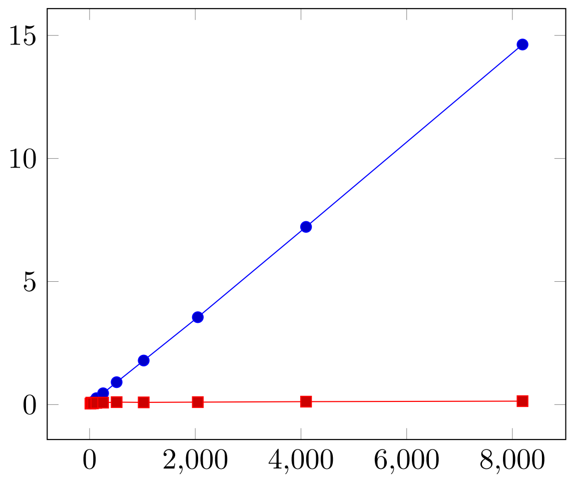

Figure 2: tree.lookup (red), list.lookup (blue)

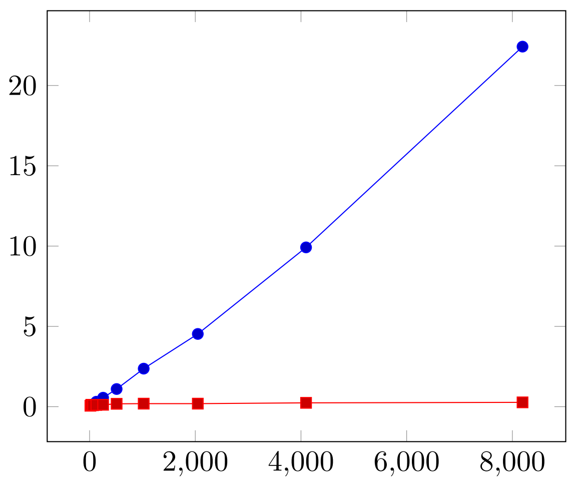

Figure 3: tree.remove (red), list.remove (blue)

Here we can tell that the binary tree is significantly faster than the list implementation.