Key-Value Store

Dhruba Das

Spring Term 2024

Introduction: Problems to solve

There are in total 3 problems I had to solve in this assignment: Implementation of the store as a list, as a binary tree, and benchmarking these two structures against each other.

EnvList

This implementation handles the key-value store as a linked list; it starts by creating a new empty list, given by calling the method *new*(), handles addition of a key-value pair to this empty list by simply adding/inserting the tuple/node to the list, and the last method handles addition of a pair to an existing list by using recursion:

```
def add([h|t], key, value) do
    case lookup([h|t], key) do
      nil ->
      z = add(t, key, value)
      [h | z]
      {_key, _value} ->
        n = remove([h|t], key)
        add(n, key, value)
    end
  end
```

As we can see, the function updates the value by removing the current node, and adding the new node with the updated value.

Moving on to the *lookup* functions, the first one simply returns nil if it's looking at an empty list; the second one tries to match given key with the key of the first node (using **pattern matching**) returning only the key-value pair if the keys match. Finally, the last *lookup* function simply uses recursion to keep looking through the list if the key doesn't match that of the first node:

```
def lookup([_ | rest], key) do lookup(rest, key) end
```

Lastly, *remove* is implemented simply by returning an empty list if the list is empty, returning the rest of the list if the key matches with that of the first node, and then using recursion to remove the node if it's key doesn't match that of the first node:

```
def remove([h|t], key) do
  case lookup([h|t], key) do
    nil -> nil
  {_key, _value} ->
    z = remove(t, key)
    [h | z]
  end
end
```

EnvTree

The implementation of this structure, admittedly, was much simpler compared to implementing it in Java. Like *EnvList*, we implement the *add*, *lookup*, and *remove* methods, with now the addition of the $leftmost$ methods.

Starting with the *add* methods, they're quite simple to implement; we start with a tree which is *nil*, given to us by the *new*() method. We then add the first node, defined by the structure $\{: node, key, value, left, right\}$, though $left$ and *right* are *nil* for the first node. We then handle the updating of a node, and the addition of new nodes using recursion and adding to the left branch if k is less than the key of the current node, and to the right branch if k is greater:

```
def add({:node, key, _, left, right}, key, value) do
  {:node, key, value, left, right}
end
def add({:node, k, v, left, right}, key, value) when key < k do
  {:node, k, v, add(left, key, value), right}
end
def add({:node, k, v, left, right}, key, value) when key > k do
  {:node, k, v, left, add(right, key, value)}
end
```

The *lookup* functions were also fairly simple to implement, as we only return *nil* when given an empty tree, the key value pair when it's found, and if not found, continue down the tree, down the right branch if given key is greater than the current branch, down the left if it's less than:

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

The *remove* functions were the most difficult to implement, as I wasn't able to quite grasp the idea of how to replace the node with the given key, but going to lecture helped to clear up my doubts, as I realized what the assignment meant by : "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)" . The implementation then becomes fairly simple:

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

The function for *remove* which I haven't included here simply returns *nil* when we try to remove from an empty tree.

The $leftmost$ methods are simple, with the first one simply returning a tuple containing the key, value and the right branch if we are already at the leftmost node, and the second one using recursion to traverse the tree if we're not at the leftmost branch; the result is then used to construct a new node with the current node's key, value, the left branch from the recursive call, and the original right branch :

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

Benchmarks and Conclusion

Below are tables containing the runtimes for 1000 operations, for both the List and the Tree:

n	add	lookup	remove
16	0.86	0.09	0.24
32	1.24	0.07	0.25
64	5.50	0.12	0.94
128	10.3	0.13	1.67
256	37.8	0.24	6.41
512	121.1	0.43	21.8
1024	461.0	0.77	74.5
2048	1770.0	1.36	284.0
4096	7470.0	4.26	1330
8192	30700.0	7.19	5360.0

(a) EnvList; times in microseconds

n	add	lookup	remove
16	0.08	0.04	0.08
32	0.13	0.06	0.10
64	0.15	0.06	0.12
128	0.10	0.08	0.12
256	0.13	0.09	0.14
512	0.27	0.10	0.16
1024	0.20	0.12	0.19
2048	0.30	0.13	0.21
4096	0.36	0.14	0.22
8192	0.52	0.23	0.40

(b) EnvTree; times in microseconds

Table 1: Time for 1000 operations; time per operation is shown

CONCLUSION: As we can see, the time taken by the List is MUCH LARGER compared to the time taken by the Tree, even for smaller number of nodes, making the Tree a much more efficient (albeit slightly more expensive when it comes to coding time) than the List, for a key-value store data structure.