An Environment: Assignment 2

Aditya Gupta

January 2023

1 Introduction

We implement a key-value database using a list and a tree structure in this assignment.

2 List Implementation

2.1 The add() Function

We represent the key-value pair which we want to insert into the list as a tuple. For example, a key-value pair can be represented using something like :

```
kvPair = {key, value}
list = [kvPair]
```

Whenever the add() function is called, we add a key-value tuple to the end of the list. If the list was previously empty we just directly put the key-value tuple into the list. In the scenario where the key was already part of the list then we just update the value associated with it.

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

2.2 The lookup() Function

For the lookup() function, if the list is empty from the beginning, we just return nil as there are no tuples in the list that are associated with the key we searched for. Otherwise, we search for the key recursively in the list. We return the tuple for the key we searched for if we find that value in the list.

```
def lookup([], key) do nil end
def lookup([ {key, value} | _], key) do {key, value} end
def lookup([{_,_} | rest], key) do lookup(rest,key) end
```

2.3 The remove() Function

This function is implemented in a similar way to the lookup function. We search for the key in the list and if we find it then we return the list with all elements besides the tuple that contains that key.

```
def remove([], key) do nil end
def remove([{key, _} | rest], key) do rest end
def remove([map | {key, _}], key) do map end
def remove([kv | map], key) do [kv | remove(map, key)] end
```

3 Tree implementation

We now implement the key-value database using a tree data structure. This is because we know that implementing it as a string wasn't the most efficient solution.

```
treeNode = {:node, key, value, left, right}
```

The key-value pairs are now stored inside nodes containing the key, value, and pointers to the relative left and right nodes.

We add key-value pairs to the tree by using an add() function. We add based on the key of the current node. If the argument key is smaller than the current key then we jump left of the current node otherwise we jump to the right of the current node. We follow the following process until we reach an empty node and then we just fill it with our key-value data. If the keys are the same then we just update the value of that tree node.

As we are implementing a binary tree, whenever data is added to the tree, everything remains sorted. The lookup function is quite similar to the add function in its implementation. If the key we are looking for is smaller than the current key then we jump to the left otherwise we jump to the right until we reach the correct node which contains the same key. We return the key-value pair as a tuple when we reach the correct node and we return null if the key isn't part of the tree database.

The final part is implementing a remove function for the tree data structure. This function was a bit more tricky to implement. When we remove a key from the tree we need to locate the correct node. We remove the key and replace it with either the leftmost node of the right branch or the rightmost node of the left branch. The function is then called recursively with the key-value pair we used to replace the key-value pair we remove earlier as input.

```
def remove({:node, key, _, left, right}, key) do
  {:node, k1, value } = leftmost(right)
  {:node, k1, value, left, remove(right, k1)}
end
```

4 Benchmarking

We run the benchmark 10,000 times for all implementations, from which we get the following results:

4.1 List Implementation

We see from the graph that the add(), lookup() and the remove() function have a time complexity of O(n). In my implementation I always add values to the end of the list as no rules specified on which end we add the key-value pairs. If key-value pairs were added to the beginning instead then we would get a time complexity of O(1) instead. We get a time complexity of O(n) for the lookup and the remove function as we need to search for the key inside of the list which takes linear time.

Keeping the list sorted can potentially be better for adding, finding or removing keys. But our current implementation's time complexity mainly hinges on the number of elements inside of the list. This means that having a larger data-set can would mean that the run-time would increase linearly.

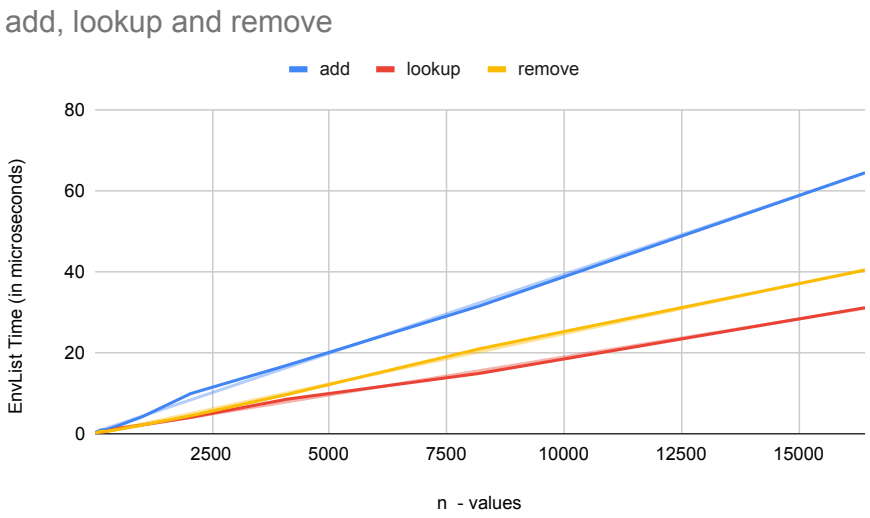

Figure 1: List Implementation Benchmark

4.2 Tree Implementation

As seen in the graph we clearly see that the add(), lookup() and the remove() functions follow a logarithmic trend. This is because our tree is a binary tree which means as we traverse it we reduce the number of possible paths by half

each time as it can either go in the left or the right direction. This means we have a time complexity of O(log(n)) for this implementation.

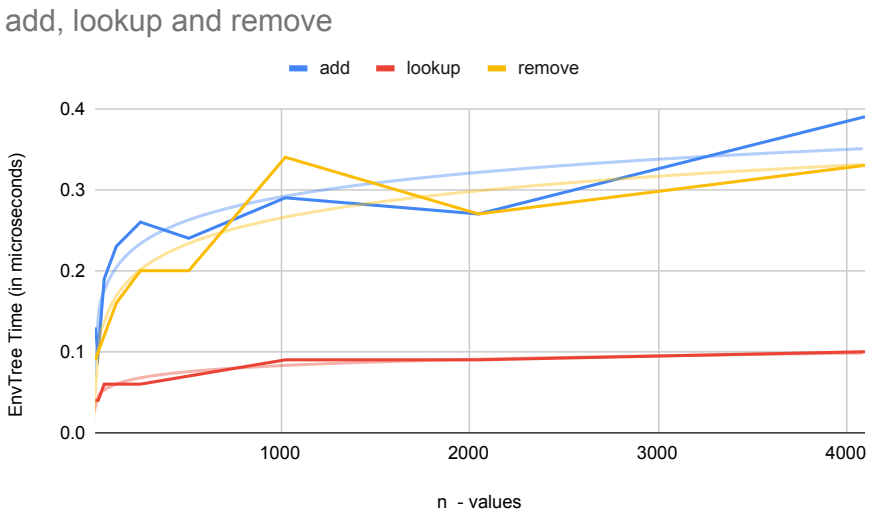

Figure 2: Tree Implementation Benchmark

4.3 Discussion

As we anticipated the list implementation is rather slow and performs the worst out of all implementations. We get a time complexity of O(n) for all the operations. In the tree implementation we get a time complexity of O(log(n)) instead which is a bit better. We were also told to benchmark and compare against the Elixir map implementation which is unsurprisingly the fastest because of its C++ trier data structure implementation. We get a time complexity of O(1) for all operations for the Elixir map and thus it performs the best out of all implementations.

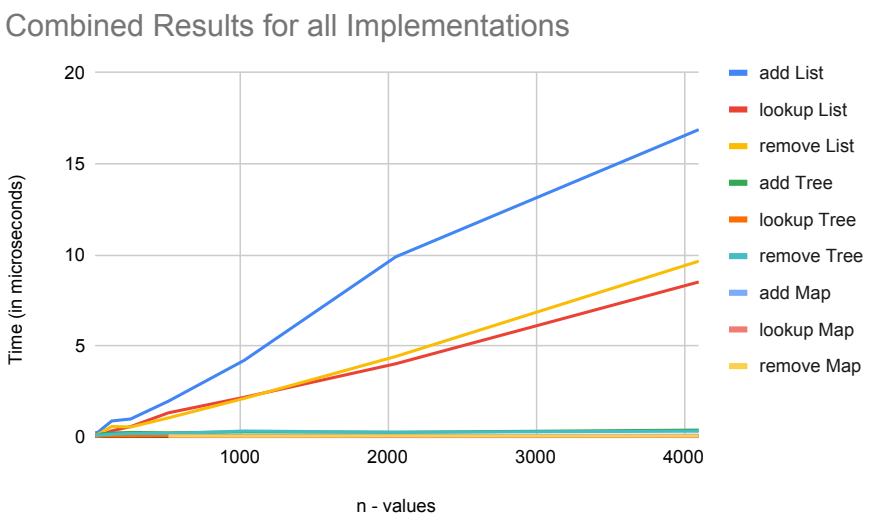

Figure 3: Combined graph to show Benchmark for all implementations