Environments - Report

Adrian Hjert

26/Jan/2023

Introduction

The aim of this report is to compare environments with key/value pairs as both linked lists and trees as well as briefly explain how some of the necessary functions were implemented in code. Code examples and explanations can be seen in the **Methods** section under the relevant subheading and benchmarks will be presented and discussed in the **Results and Conclusions** section. The code in which the benchmarks are taken are provided in the instructions of the assignment.

Methods

Both the List and Tree implementations require the same basic funtions; *new(), add(), remove() and lookup()* which do what common sense tells us they will do. Their implementations however, differ a little bit.

List Implementation

The list implementation is rather straight forward, as we do not need to worry about left and right pointers. *new()* will create an empty list and *add()* will add a key/value pair to the end of the list. The add method can be seen below.

```
#add stuff to the map/list
  def add([], key, val) do [{key, val}] end
  def add([{key, _} | map], key, val) do [{key, val} | map] end
  def add([ass | map], key, value) do
    [ass | add(map, key, val)]
  end
```

Essentially we have that if we have an empty list, we just return the list of the key (an atom) and the value associated with it. If the list is not empty, we have to cycle through the list until we have a slot to add the key/value pair. Essentially what happens is we, behind the scenes, set the head node to the end node, then add the rest of the list to it, if there is something in the list, we make the head node of the rest into the tail and add the remainder of that to the end, until we get to where the rest is an empty list, then we add the key/value pair. What ends up happening is that the list is created in the order we added the pairs, so first pair we added is all the way to the left in the terminal, and the most recent entry is all the way to the right. This means that the *add()* function should have O(n) time complexity.

The *lookup()* function essentially cycles through the list from left to right until it finds a key that exists in the list and returns the key and the value associated with it. There are of course separate cases for if the key we are searching for is at the beginning of the list, then we just return that, if not, a different lookup call will execute. An empty list will return **nil**. This should also have O(n) time complexity.

The *remove()* function is also rather straight forward. It has 3 cases as can be seen in the following code.

```
#take things away from map/list
  #_key is just "_" but it tells the reader that we mean key
  def remove([], _key) do [] end
  #removes head
  def remove([{key, _} | map], key) do map end
  #kv = key value
  def remove([kv | map], key) do [kv | remove(map, key)] end
```

Once again, the code will check from left to right to find the key/value pair we wish to remove from the list. Esentially, it remembers the key we wish to remove, sets "kv" to the first key value pair, and compares, if these match, it returns the lsit without the key/value pair, otherwise, it moves to the next element(s) in the list until a match is found or until we run out of elements to compare. This should also have an O(n) time complexity unless it we always remove the head/first element.

Tree Implementation

The tree implementation is slightly less straight forward as now we must keep track of left and right nodes and try to keep the tree somewhat balanced though the balancing is not a necessity, it may be nice. The tree implemented is balanced based on the key. The earlier the key comes in the alphabet, the lower value it has. The lower values will be placed in left nodes and the others to the right. This implementation has the same functions as the list implementation albeit slightly different in nature, *new()*, for example now returns **nil** but can be used to create a variable that contains the tree, *add()* will add a key/value node either to the root, or to the left or right of existing nodes, etc. The *add()* function should have a time complexity of O(log n).

The *remove()* function is slightly more complicated. Given that it is a tree, we have more cases we need to account for. Essentially we need to check the key we wish to remove against the key we are iterating through and then iterate either down the left branch or the right branch if they do not match initially. Once we find the key we wish to remove, we replace it with a node from the left. To do this, we have a function called *leftmost()*, which can be seen below.

```
#---------finds an appropriate node to the left to
#---------replace the node we wish to delete-------
  def leftmost({:node, key, val, nil, rest}) do
  {key, val, rest} end
  def leftmost({:node, k, v, left, right}) do
    {key, val, rest} = leftmost(left)
    {key, val, {:node, k, v, rest, right}}
  end
```

Similar to the *add()* function, the *remove()* function should have a time complexity of O(log n).

The *lookup()* function is relatively straight forward, much like that of the list implementation of it, with the major difference being that now we must decide if we iterate to the left or right of the tree. The function can be seen below.

```
#----------looks up the value of a given key-------
  def lookup(nil, _key) do nil end
  def lookup({:node, key, val, _left, _right}, key) do
   {key, val} end
  def lookup({:node, k, _, left, _right}, key)
          when key < k do lookup(left, key) end
  def lookup({:node, _, _, _left, right}, key) do
   lookup(right, key) end
```

This function will first check if the tree is empty, wherein it will return **nil**, otherwise it will compare the first key/value pair, then, if no match is found, it will compare the key given to the key it is looking at and determine if it should go left or right. It does this recursively until a match is found or until it runs out of keys to check. This should follow O(1).

Results and Conclusions

The implementations discussed above were compared using the code for *Bench* given in the instructions. This produced the following results seen on **Table 1** where times are in ms. The built in *map()* function was also benchmarked and the results can be seen in **Table 1**.

Table 1: Showing Benchmark Results

n	*add()*			*lookup()*			*remove()*		
	list	**tree**	**map**	**list**	**tree**	**map**	**list**	**tree**	**map**
16	0.00	0.00	0.00	0.00	0.00	0.00	0.64	0.00	0.00
32	0.32	0.00	0.32	0.00	0.00	0.00	0.00	0.00	0.00
64	0.16	0.00	0.00	0.00	0.16	0.00	0.16	0.00	0.16
128	0.32	0.24	0.08	0.24	0.08	0.00	0.00	0.08	0.08
256	0.68	0.20	0.08	0.36	0.08	0.04	0.00	0.00	0.04
512	0.90	0.24	0.06	0.62	0.10	0.04	0.04	0.02	0.02
1024	2.15	0.43	0.08	1.13	0.12	0.03	0.09	0.04	0.03
2048	4.91	0.40	0.11	2.23	0.13	0.04	0.09	0.07	0.04
4096	13.98	0.49	0.08	4.45	0.16	0.04	0.32	0.03	0.03
8192	16.99	0.49	0.09	9.09	0.15	0.04	0.35	0.03	0.03

We can see that as predicted, the *add()* function followed O(n) for the list implementation and something along the lines of O(log n) for the tree implementation. The *map* gives a constant time for adding an element. Of course, due to the fact that the accuracy of the timers may not be 100% correct, there will be some outliers, and this goes for all the functions that were tested, especially at low values of n.

The *lookup()* also has an O(n) time complexity for the list implementation. For this function, unlike *add()*, the tree implementation follows an O(1) function, much like the *map* equivalent. It is worth noting however that this may vary depending on if the tree is skewed or not.

The *remove()* function for lists gives somewhat erratic results. This is most likely due to the randomness of what element we wish to delete during the benchmarking process, also that it is a very quick process to find an element and remove it but given the way the function behaves, it is O(n) unless we always delete the head element, in which case it would be O(1). The tree implementation we seem to have a O(log n) time complexity based on the relationship between the data points for $n = 512$ through $n = 2048$ and the fact that it decreases in time for $n = 4096$. As with the list version of the function, this is a very fast process so it is likely that the timers lack the accuracy to measure such a process for the amount of elements we are measuring (note that $n = 16$ through $n = 64$ has a measured time of 0.00 ms which shows that it may not be accurate enough). The *map* version of the function, much like the other functions for *map* has a time complexity of O(1). We can also see that the tree is similar in time to *map* across all functions, albeit slightly slower, wth the list implementation being the slowest of them all.