Environment

Saina Shamshirdar

Spring Term 2024

Introduction

A key-value database, a map, is going to be implemented in this task in two different ways. One using lists and another one using trees as the data structure. The implementation should include functionalities add, lookup and delete meaning it should be possible to add a new key-value pair to the map, to look up a certain one and to be able to delete a pair if desired. It should also be possible to create or return an empty map.

Map as a list

To examine whether the functions operate correctly a list was implemented as followed:

```
map = [{:a,17}, {:b, 2}, {:c, 25}, {:d, 13}, {:e,8}]
```

The add function was implemented and tested by adding the tuple :f, 46 to the list by calling the function on the list above, add(map, :f, 46). The outcome was the following list below:

```
[a: 17, b: 2, c: 25, d: 13, e: 8, f: 46]
```

It is shown that the key-value pair :f, 46 has successfully been added to the list.

Now, a search for the key ”d” is performed by the lookup function. If the key exists in the map, the function should return the key-value pair as a tuple. So the functions us called,lookup(map, :d). And the output is successfully represented, which is :d, 13.

remove

To test the remove function, the function was called to delete the key ”c”, remove(:c, map). The outcome was as followed which successfully deleted the desired key.

```
[a: 17, b: 2, d: 13, e: 8]
```

It is not necessary to have a sorted list, since the functions don't depend on the value of the keys. For example, the add function doesn't care about where to place the new element. It always place it at the end of the list by moving along the list.

Neither does the lookup function bother to care about the order of the values. It only pattern matches. If the searched element is not in the head of the list, it simply continues the search through the rest of the list, which is the tail. If anywhere along the way, the pattern matches, the function returns the pair.

The same principle applies to the remove function. It also pattern matches through the list until it fins the specific key-value to be removed.

Map as a tree

On the other hand, the tree must be sorted in order to lookup a key. First, to test the add function, a tree was constructed as followed.

```
tree = {:node, :a, 56, {:node, :b, 23,{:node, :d,13, nil, nil},
{:node, :e, 43, nil, nil} },
{:node, :c, 78, nil, nil}}
```

The add function was called upon it, add(tree, :f, 36) and the result was as below:

```
{:node, :a, 56,
 {:node, :b, 23, {:node, :d, 13, nil, nil},
 {:node, :e, 43, nil, nil}},
 {:node, :c, 78, nil, {:node, :f, 36, nil, nil}}}
```

The new key-value pair was successfully added to the tree.

To look for a special key in the tree, as said earlier, the tree has to be sorted. Therefore, the function checks if the searched key is less than the root node value. If it is, it goes on and searches the left subtree recursively. Otherwise it starts to search the right subtree. The function terminates whenever it has reached a node with no left or right branch, which is also he base case.

To remove a key-value pair from the tree, there are some different cases. If the node to be deleted has only a right branch, then the right branch is simply returned. On the contrary, the left branch is returned.

Another case is when the key to be deleted has a smaller value than the root. In this case the remove is recursively called upon the left subtree, otherwise its called on the right subtree.

If it is required to delete a key that has both a left and a right subtree, the key should be replaced with something, otherwise the rest of the tree is lost. One way to handle this case is to travel to the leftmost leaf of the right

subtree, replace the root value with the value of the leftmost key and delete the leftmost key. This operation keeps the tree sorted and intact. In order to do this a leftmost function is needed that travels through the tree given to it and returns the key and value of the leftmost leaf and also returns the rest of the tree without the leftmost leaf.

The remove function can now use this leftmost function to find the leftmost leaf of the right subtree, replace the key to be deleted with its value and then delete the leaf. It constructs a new tree where the key and the value has the key and the value of that leftmost key, the left subtree remains unedited and the right subtree is the subtree obtained from the leftmost function which had the leftmost leaf deleted.

Bench

The goal here is to measure how much time it takes to perform and operation on the list, tree and the map from the Elixir library.

size	**list**	**tree**	**map**
32	0.51	0.31	0.20
128	1.84	0.41	0.20
512	4.61	0.51	0.20
2048	21.91	0.82	0.41
4096	47.31	0.72	0.31
8192	95.33	1.23	0.51

Table 1: add operation on different list, tree and map sizes, time per operation in μs.

The measurements in table 1 show that for large number of elements in a list, the runtime for add operation increases very much. This is because the recursion repeats for every element until it reaches the end of the list. If the list is long, of course it takes a very long time to terminate. The time complexity is therefore O(n) where n is the number of key-value pairs in the list. The tree implementation takes less time compared to the list. It increases slightly, but it is still not as much as list does. This is because the function doesn't need to traverse the whole tree, or visit every key. When adding an element to an empty tree or updating the value of an existing key, the time complexity is constant, O(n). The worst case would be when the tree is unbalanced, which would have the complexity O(n). Since this is not the case and the tree is balanced the time complexity is O(log(n)) where n is the number of keys in the tree. O(log(n)) is indeed the height of the tree. Elixir's map module is definitely the most sufficient implementation in this case and seems to be quite constant.

size	list	tree	map
32	0.20	0.20	0.10
128	0.72	0.20	0.10
512	2.35	0.31	0.20
2048	6.04	0.31	0.20
4096	13.93	0.41	0.10
8192	42.80	1.51	0.20

Table 2: lookup operation on different list, tree and map sizes, time per operation in μs.

Same as the add operation, the time complexity of the lookup operation on the list increases as the list grows in size (see table 2). When the list is empty or the key is found in the first element the time complexity is constant. Otherwise the list has to be traversed recursively which takes O(n) where n is the number of elements in the list until the searched element. The lookup on a tree takes much less time. If the tree is empty or the searched key is in the root, the time complexity is constant. In other cases, the function compares the searched with the current key and depending on that it either does a recursive call on the left or the right subtree. This leads to a time complexity of O(log(n)) in the best case being the tree is balanced, which is valid in this assignment. The map implementation is indeed constant when it comes to time complexity and is therefore super efficient.

size	list	tree	map
32	0.31	0.20	0.20
128	0.82	0.31	0.20
512	5.22	0.72	0.72
2048	15.36	0.61	0.20
4096	47.51	0.92	0.51
8192	87.65	0.51	0.20

Table 3: Delete operation on different list, tree and map sizes, time per operation in μs.

Finally, the delete operation runtime on a list is observed in table 3. It is clearly increasing rapidly as the list grows.When the list is empty or the key to be deleted is the first element in the list, the time complexity is constant, O(1). Otherwise, in the worst case, the function has to traverse the whole list by recursively calling the function on the tail of the list which takes linear time, O(n), where n is the number of elements in the list.

Deletion on a tree takes much less time and is almost constant according

to the measurements. When the tree is empty, the time complexity is O(n). When the current node is the one to be deleted and it only has a right subtree, the sight subtree is returned or vice versa. This takes contant time too. When the target node to be deleted has both a left and a right subtree the leftmost function helps to find the leftmost key of the right subtree and replace the value with the target node. The complexity in this case is O(log(n)) where n is the number of nodes in the tree. Another case is when the function compares the key to be deleted to the current key in the tree. If it is smaller, the remove is recursively called on the left subtree and on the right subtree otherwise. This takes logarithmic time too.

Finally, not surprisingly, the map implementation is the most efficient in removing too.