Environment

Ayda Ghalkhanbaz

Spring Term 2024

Introduction

In this task a key-value database is implemented using two different approaches. The keys in the database are atoms while the values are integers. These approaches will be explained and analysed to identify the differences in their time complexity in adding a new pair, looking up for a pair, and removing a key-value pair. The results of the benchmark will be discussed later in this report.

A Map as a List

The first approach involves implementing the map as a list of tuples, where each tuple represents a key-value pair. For each approach, four functions have been implemented, namely: a constructor `new\0`, `add\3`, `lookup\2` and `remove\2`.

The `new\0` constructor is straightforward, returning an empty list. The `add\3` function consists of a base case where a key-value pair is added into an empty list. The next case is when we try to update the value of a key that already exists in the list. This case simply overrides the value of that particular key. The general case of the `add\3` function utilises recursion and adds the key-value pair at the end of the list.

The `lookup\2` function also comprises three different cases. The base case is looking up for a key in an empty list which returns nil. The second case is when the sought key is at the beginning of the list, allowing for an immediate return. The general case employs recursion till the sought key is found.

The `remove\2` function works in a similar way as well. It includes a base case, a second case where the desired key is at the head of the list, and a general case where the function is called recursively. Below you'll find an overview of the `lookup\2` and `remove\2` functions for the first approach.

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

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

    def remove(_, []) do [] end
    def remove(key, [{key,_}|tail]) do tail end
    def remove(key, [head|tail]) do [head|remove(key,tail)] end
```

A Map as a Tree

The second implementation for the database uses a tree structure. This tree is a tuple composed of an atom indicating a node, a key, a value, left and right branches, which are `nil` in case of a leaf.

Same functions as before with similar logic were implemented for the tree. Since it is meant for the tree to be ordered, the `add`\3 function inserts the new key-value pair in order. Specifically, if the key is smaller than the root/current key, the new pair is added to the left; if it is larger, the new node will find its place in the right branch.

The `lookup`\2 function operates similarly in terms of comparing the keys. It calls itself recursively until the sought key is found.

The most challenging part of this implementation was the `remove`\2 function. However the skeleton of the code was provided through the assignment and it was completed after filling missing spots. The `remove`\2 function utilises another function called `leftmost`\1 which returns the leftmost node in the right branch to replace the deleted node. Below the `remove`\2 and `leftmost`\1 functions are shown.

```
def remove(:nil, _) do :nil end
def remove({:node, key, _, :nil, right}, key) do right end
def remove({:node, key, _, left, :nil}, key) do left end
# find the leftmost node in the right branch and replace it
def remove({:node, key, _, left, right}, key) do
    {leftMostKey, leftMostValue, leftMostRight} = leftmost(right)
# left branch is still the same but we have a new right branch
    {:node, leftMostKey, leftMostValue, left, leftMostRight} end
def remove({:node, k , value, left, right}, key) do
    if key < k do
      {:node, k, value, remove(left, key), right}
    else
      {:node, k, value, left, remove(right, key)}
    end
end
```

```
# have we reached the leftmost node?
# return the node with its right branch
def leftmost({:node, key, value, :nil, right}) do
    {key, value, right} end
def leftmost({:node, k, v, left, right}) do
{leftMostKey, leftMostValue, leftMostRight} = leftmost(left)
{leftMostKey, leftMostValue, {:node, k, v, leftMostRight, right}}
end
```

Result

A benchmark was conducted to compare the time taken by each approach to perform operations for a growing size of map n. The results are presented in table 1.

	List			**Tree**		
Size	**add**	**lookup**	**remove**	**add**	**lookup**	**remove**
16	0.07	0.03	0.04	0.07	0.04	0.04
32	0.10	0.05	0.07	0.09	0.05	0.05
64	0.29	0.15	0.26	0.08	0.06	0.07
128	0.95	0.31	0.94	0.09	0.07	0.07
256	1.91	0.47	2.02	0.10	0.08	0.09
512	3.45	0.83	3.42	0.11	0.10	0.11
1024	6.13	1.30	5.58	0.20	0.11	0.13
2048	10.54	1.72	10.34	0.15	0.13	0.15
4096	23.88	3.83	23.80	0.17	0.15	0.17
8192	46.59	11.80	46.63	0.20	0.16	0.18

Table 1: The running time of different operations on the two approaches with a varying size in μs

Discussion

From the outcomes presented in table 1, we can see that the tree implementation significantly outperforms the list implementation as the size of the map grows. The `add\3` function is roughly 230 times faster in the tree implementation. When adding a new key-value pair to the list implementation, one needs to traverse the entire list before adding it to the end of the list. This algorithm performs well when the map size is relatively small (as shown in table 1, when the size is 16 or 32, the list implementation performs similarly to the tree version). However as soon as the map size starts growing, the running time of `add\3` also grows linearly.

The tree implementation is also approximately 74 and 260 times faster than the list implementation in `lookup\2` and `remove\2`, respectively. Once again, the underlying algorithm is the key factor influencing this behaviour. The `lookup\2` function in list implementation may traverse the entire list to find the sought element in the worst case scenario, although not necessarily; since the sought element can be in the beginning or in the middle of the list which is why it is faster than the `add\3` operation but not as fast as the `lookup\2` in the tree implementation.

However it shouldn't be that surprising since we already know that the tree implementation has a time complexity of $\mathcal{O}(log(n))$ on average while lists operate linearly and have $\mathcal{O}(n)$ as time complexity.