An Environment

Seema Fatima Bashir

25-01-24

1 Introduction

The assignment aims to implement a key-value database, known as a "map" in Elixir. The implementation should enable the retrieval of values associated with specific keys through predefined operations. The assignment aims to develop two techniques for constructing this map, both encompassing essential functionalities such as creating an empty map, adding or modifying key-value associations, performing key-based value look-ups, and removing associations from the map. The report delves into implementing the map using two primary methods: a list-based approach and a tree-based approach. Lastly, benchmarks are used to assess the performance of the different data structures and compare the efficiency of these implementations.

2 Creating List

In Elixir, lists are fundamental data structures used to store collections of elements. Thereby, the EnvList module is created, it defines a custom list implementation. Each element in the list consists of a key-value pair. The module provides functionality for creating new lists, adding key-value pairs to the list, looking up values by keys, and removing elements from the list.

new()

The new() function plays a crucial role in initializing data structures to their initial state. When considering the creation of a list, the new() function conventionally returns an empty list []. This signifies the instantiating of a list with no elements, providing a clean starting point for adding, removing, or manipulating data within the list.

add(map, key, value)

Additionally, the add functions are implemented within the within the EnvList module. They are created to incorporate key-value pairs into an existing data structure, particularly a map. The function are therefore structured to accommodate different scenarios based on the contents of the map. In the case of the list being empty, the function seamlessly adds the provided key-value pair. In cases where the list already contains elements, the function utilizes pattern matching to inspect each entry. If the specified key already exists in the map, the associated value is updated to reflect the new value, ensuring that the map remains consistent and up-to-date. Conversely, if the key is not found within the map, the function simply adds the key-value pair to the existing map, incorporating the new association seamlessly.

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

lookup(map, key)

The lookup/2 function in the EnvList module searches for key-value associations within a list. It uses pattern matching to handle different scenarios based on the list's structure and the provided key. If the list is empty, the function returns nil, indicating no association is found. Otherwise, it inspects the first entry for a matching key. If found, it returns the associated value as a tuple. If not, it continues searching recursively through the remaining entries until either a match is found, returning the corresponding value, or the end of the list is reached, resulting in a return value of nil.

remove(key, map)

The remove/2 functions employs pattern matching to handle different scenarios based on the contents of the list and the specified key. The first clause of the function, def remove([], key), do: [], serves as the base case where the map is empty. In this scenario, invoking the remove/2 function with an empty map results in an empty map being returned, as there are no associations to remove. Subsequent clauses of the function handle cases where the map contains elements. When the function is invoked with a non-empty map, it inspects the first entry to determine if it contains the specified key. If a match is found, the entry is removed from the map, and the resulting map without the specified key-value association is returned. Conversely, if the specified key is not found within the first entry, the function recursively continues the search through the remaining entries of the map until either a matching key is found, resulting in the removal of the corresponding entry, or the end of the map is reached, leading to the return of the original map.

Sorted vs Unsorted list

Arranging the elements in sorted order could possibly enhance efficiency, especially concerning the lookup operation. Presently, with an unsorted list, finding a specific key involves traversing the list linearly until a match is found or reaching the list's end. By organizing the list in sorted order, a more optimized binary search algorithm could be utilized, thus reducing the time complexity of the 'lookup' operation. However, it's essential to consider the drawbacks of this approach. Sorting the list would require additional effort to maintain the sorted order during insertions and deletions, potentially impacting the performance of the 'add' and 'remove' operations.

3 Creating a Tree

The EnvTree module defines a tree structure in Elixir, allowing for efficient organization and retrieval of data. Unlike lists, which may become inefficient as they grow larger, trees provide a more scalable solution for storing and accessing data. The structure of the tree is represented by the atom nil for an empty tree and by the tuple :node, key, value, left, right for a node, where key and value represent the data associated with the node, and left and right denote the left and right subtrees, respectively.

add, lookup remove

The add/3 function within the module is an important function for maintaining the sorted order of the tree while inserting new nodes. By recursively traversing the tree and comparing keys, this function ensures that the new node is inserted at the appropriate position to preserve the sorted structure. This approach allows for efficient insertion of new elements into the tree without compromising its integrity. Additionally, the binary search-like algorithm employed by the lookup/2 function enables efficient searching within the tree. By recursively traversing the tree based on key comparisons, this function quickly locates the desired node and returns its associated value if found. This binary search approach reduces the time complexity of the lookup operation, making it highly efficient for retrieving data from the tree.

On the other hand, the remove/2 function is responsible for removing nodes from the tree while ensuring that the sorted structure is preserved. This function handles various scenarios, including removing leaf nodes, nodes with one child, or nodes with two children. In cases where a node has two children, the function identifies the leftmost node in the right subtree to maintain the sorted order after removal. By addressing these different scenarios, the remove/2 function effectively maintains the integrity of the tree's structure while accommodating node removal.

4 Benchmark

Lastly, benchmarks will be conducted to assess the performance of different implementations and compare their efficiency. The objective is to evaluate how each implementation handles an increasing number of key-value pairs and measure the time taken to execute various operations. A benchmark will be set up where a map containing a specific number of elements will be constructed. Subsequently, the time taken to perform operations such as insertion, lookup, and removal will be measured for each implementation. To ensure consistency, the same set of key-value pairs will be used for all implementations during the benchmark.

Table 1: Comparison of Benchmarks (Time per Operation in μs)

n	Add			Lookup			Remove		
	List	Tree	Map	List	Tree	Map	List	Tree	Map
32	0.51	0.31	0.20	0.20	0.20	0.10	0.31	0.20	0.20
128	1.84	0.41	0.20	0.72	0.20	0.10	0.82	0.31	0.20
512	4.61	0.51	0.20	2.35	0.31	0.20	5.22	0.72	0.72
2048	21.91	0.82	0.41	6.04	0.31	0.20	15.36	0.61	0.20
4096	47.31	0.72	0.31	13.93	0.41	0.10	47.51	0.92	0.51
8192	95.33	1.23	0.51	42.80	0.51	0.20	87.65	0.51	0.20

(a) Result Table

Comparing the performance of the list and tree implementations with Elixir's built-in Map module, benchmarks were conducted using two random sequences. The first sequence populated the map, while the second measured the time for each method call. Results revealed the Elixir Map module's superior performance, as anticipated. However, the tree implementation, with a time complexity of O(log n) due to its sorted nature, also performed admirably. Conversely, the list implementation, being unsorted with a linear time complexity of O(n), demonstrated lower efficiency. The choice between list and tree depends on specific needs. Lists are straightforward but may become inefficient with large datasets, while trees, especially balanced ones, are better suited for efficient retrieval and addition of elements in such scenarios.

For the list implementation, the time complexity of the remove operation is O(1), and for add, it's O(n) due to list traversal of the first list. Sorting could improve the add operation's efficiency but won't change the worst-case complexity. A balanced binary search tree has a time complexity of O(log(n)), but unbalanced trees result in O(n) complexity. The add operation in the tree implementation has a best and average time complexity of O(log n) due to traversal depth, but can degrade to O(n) in the worst-case scenario of an unbalanced tree. Similarly, the remove operation maintains a best and average time complexity of O(log n) in balanced trees but can become O(n) in unbalanced ones.