# Environment report

Praanto Samadder

January 26, 2024

## Introduction

## 1 Map in a list

### 1.1 Sorting our list

It would indeed make sense to keep the list sorted. This would allow us to implement binary search in the lookup function. Since the list we are dealing with is a key-value pair list, we would need to implement a hash-map.

In this report however, we have instead decided to sort the list by the atoms. Three things to consider: atoms are unique, atoms can be converted to strings, strings are represented as binary in Elixir. Which this trick, we can convert the atom to a string to determine the position of key-value pair in our list.

We can then perform a binary search on our list to find the element that we are looking for. Listing 1 shows the implementation of `EnvList`

```
defmodule EnvList do
    def add(list, key, value) do
        new_tuple = {String.to_atom(key), value}
        [new_tuple | list] |> List.keysort(0)
    end

    def lookup(list, key) do
       BinarySearch.search(list, key)
    end
end
```

Listing 1: EnvList module

## 2 Map in a tree

### 2.1 The tree structure

We implement a standard binary tree without a heap structure. When adding an element, if the key is smaller than our current node, we add it to the left branch.

Listing 2 shows how new elements are added

```elixir
defmodule EnvTree do
    def new do
      {:node, nil}
    end

    def add({:node, nil}, value) do
      {:node, value, {:node, nil}, {:node, nil}}
    end

    def add({:node, key, left, right}, new_value) do
      if new_value < key do
        if left == {:node, nil} do
          {:node, key, {:node, new_value, {:node, nil}, {:node, nil
    }}, right}
        else
          {:node, key, add(left, new_value), right}
        end
      else
        if right == {:node, nil} do
          {:node, key, left, {:node, new_value, {:node, nil}, {:
    node, nil}}}
        else
          {:node, key, left, add(left, new_value)}
        end
      end
    end
end
```

Listing 2: Add function EnvTree

## 2.2 Looking up an element

Listing 3 shows how the lookup function is implemented.

```elixir
defmodule EnvTreeSearch do
    def lookup({:node, nil}, key) do
      :no
    end

    def lookup({:node, v, _, _}, v) do
      {:ok, v}
    end

    def lookup({:node, v, left, right}, key) do
      if v > key do
        lookup(left, key)
      else
        lookup(right, key)
      end
    end
end
```

Listing 3: Lookup function EnvTree

### 2.3 Removing an element from the list

Listing 3 shows the remove function being implemented. It makes use of Elixir's pattern matching

```
defmodule EnvTreeRemove do
  # If no element is found
  def remove({:node, nil}, _) do
    :no
  end

  # Removes if the root is the element we look for
  def remove({:node, key, _, _}, key) do
    {:node, nil}
  end

  # Removes the left branch
  def remove({:node, key, {:node, val, _, _}, don_t_care}, val) do
    {:node, key, {:node, nil}, don_t_care}
  end

  # Removes the right branch
  def remove({:node, key, don_t_care, {:node, val, _, _}}, val) do
    {:node, key, don_t_care, {:node, nil}}
  end

  # If neither branch should be removed, then picks a side
  def remove({:node, key, left, right}, val) do
    if val < key do
      {:node, key, remove(left, key), right}
    else {:node, key, left, remove(left, key)}
    end
  end
end
```

Listing 4: Remove function function EnvTree

## 3 Benchmark results

Figure 1 shows the time taken to all the elements into a tree (which was pre-populated with 1.000 elements) as the total number of elements in the list grows. Compared are the add time (in red) and the lookup times (in blue)

As can be seen, the graph looks to be ordo $O(n)$ which is different from a traditional $O(\log n)$ of a binary tree. Two explanations for this: a) the benchmark measures the total time taken for all elements to be added (instead of a single element in each iteration) and b) inefficiencies in Elixir pattern matching which may have influenced our results.

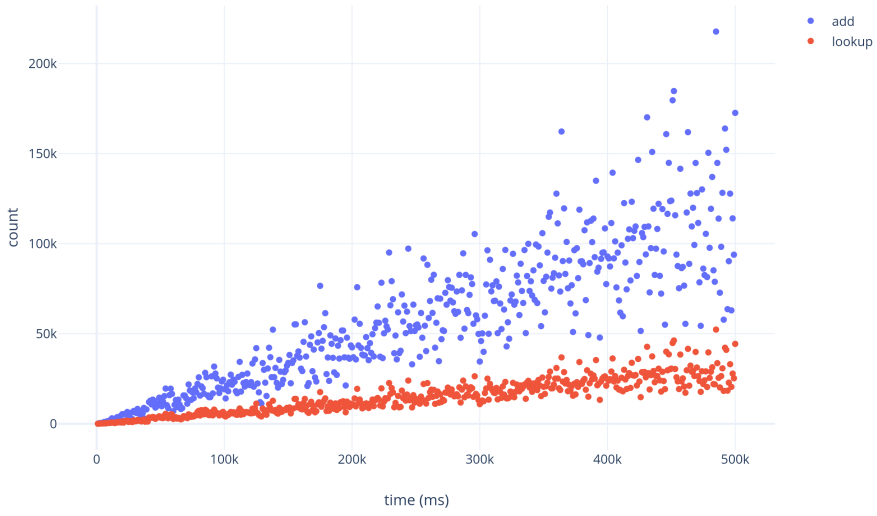


Figure 1: Some caption here