Elixir Trees vs Lists

Tomás Belmar da Costa

Spring Term 2022

Introduction

For this assignment we had to compare the tree data structure to the list data structure in elixir. We started off with the following skeleton code:

```
defmodule Bench do

  def bench() do
    bench(100)
  end

  def bench(l) do

     ls = [16,32,64,128,256,512,1024,2*1024,4*1024,8*1024]

    time = fn (i, f) ->
      seq = Enum.map(1..i, fn(_) -> :rand.uniform(100000) end)
      elem(:timer.tc(fn () -> loop(l, fn -> f.(seq) end) end),0)
    end

    bench = fn (i) ->

      list = fn (seq) ->
        List.foldr(seq, list_new(), fn (e, acc) -> list_insert(e, acc) end)
      end

      tree = fn (seq) ->
        List.foldr(seq, tree_new(), fn (e, acc) -> tree_insert(e, acc) end)
      end

      tl = time.(i, list)
      tt = time.(i, tree)

      IO.write("  #{tl}\t\t\t#{tt}\n")
```

```
    end

    IO.write("# benchmark of lists and tree \n")
    Enum.map(ls, bench)

    :ok
  end

  def loop(0,_) do :ok end
  def loop(n, f) do
    f.()
    loop(n-1, f)
  end

  def list_new() do ... end

  def list_insert(e, l) do
     :
  end

  def tree_new() do ... end

  def tree_insert(e, l) do
     :
  end
end
```

We had to complete its functions, tree_new, list_new, list_insert, and tree_insert. Then we had to run the built-in benchmark function to test the speeds of inserting elements into the data structures.

Completing the functions

The first place to start was the list_new and list_insert functions. Creating a new list was as easy as returning an empty list, but inserting to a list in an ordered manner was slightly more complicated. I used a simple Binary Search to find the position that the new element should be in, like so:

```
  def list_new() do
    []
  end

  def list_insert(e, l) do
     list_insert(e, l, 0, length(l))
```

```
end

def list_insert(e, l, low, high) do
  if high <= low do
    List.insert_at(l, low, e)
  else
    mid = floor((low + high) / 2)
    if e < Enum.at(l, mid) do
      list_insert(e, l, low, mid)
    else
      list_insert(e, l, mid+1, high)
    end
  end
end
```

The second set of functions were the tree_new and tree_insert functions which were slightly more complicated, but with recursion relatively simply to implement! Returning a new tree was as simple as returning an empty one, represented by the atom: `:nil`.

```
def tree_new() do
  :nil
end
```

The next set of functions was the tree_insert functions, which had to have several cases to account for:
- When you are inserting at an empty tree
- When you are inserting at the end of a tree to the right of the last leaf
- When you are inserting at the end of a tree to the left of the last leaf
- When you are inserting to the right of a leaf that is not at the end of the tree
- When you are inserting to the left of a leaf that is not at the end of the tree

These cases were all defined by the following function signatures:

```
def tree_insert(e, :nil) do
  # :nil represents an empty list
end

def tree_insert(e, {:leaf, h}=right) when e < h do
  # If we find a leaf and not a node it means we're at the extremity of a tree
  # e < h means we are inserting to the right, as trees are ordered
end

def tree_insert(e, {:leaf, _}=left) do
```

```
  # e >= h necessarily, so we insert to the left
end

def tree_insert(e, {:node, h, left, right}) when e < h do
  # same logic as above but a node was found instead of a leaf
end

def tree_insert(e, {:node, h, left, right}) do
  # same logic as above
end
```

The first function can be completed by simply returning the new tree, which is just the new leaf.

```
def tree_insert(e, :nil) do
  {:leaf, e}
end
```

The next two functions will return a node that has no leaf to the left, and the new leaf to the right, and vice-versa

```
def tree_insert(e, {:leaf, h}=right) when e < h do
  {:node, e, :nil, right}
end

def tree_insert(e, {:leaf, _}=left) do
  {:node, e, left, :nil}
end
```

Finally the last two functions use recursion to keep going down the tree until they finally reach the aforementioned functions

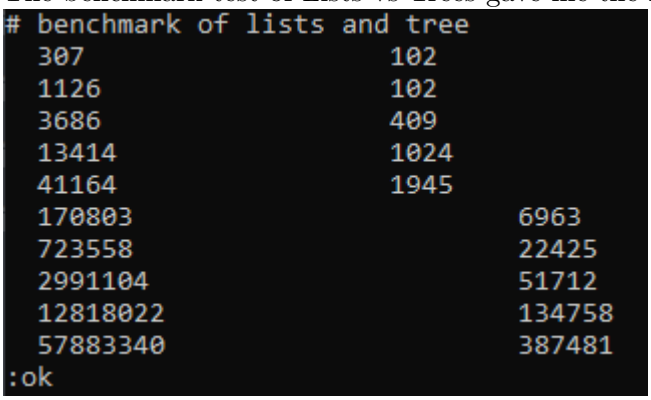
```
def tree_insert(e, {:node, h, left, right}) when e < h do
  {:node, h, tree_insert(e, left), right}
end

def tree_insert(e, {:node, h, left, right}) do
  {:node, h, left, tree_insert(e, right)}
end
```

This is all that is needed to complete the tree, as any new node will recursively traverse down the tree and insert itself in the correct place

Pros and Cons

The benchmark test of Lists vs Trees gave me the following result:



It's clear that for insertions, Trees are the better choice in Elixir. They gave off a better benchmark time for all tests, whether the number of insertions was small or large. At first this surprised me, as Binary Search has `O(log N)` time complexity, but it makes sense when you consider the fact that lists are stored in a similar way to Linked Lists in Elixir. This means that despite the fact that my algorithm would be `O(log N)` if retrieving an item at index `i` could be done in Linear Time, to get an item at a certain position, Elixir requires that we enumerate the list first which should take `O(N)` time. This makes the algorithm be extremely slow for large numbers.

Trees on the other hand, have an `O(log N)` time complexity for insertion, making them an extremely viable way of holding ordered sets. The only disadvantage I can think of is the fact that each node in a tree stores more information than just the element at the tree, and therefore will likely take up more memory.