Task 4 - Tree vs List

Casper Kristiansson

2022-02-02

Introduction

The fourth exercise of the course Programming II consisted of implementing an ordered list and ordered tree. The goal of the task is to compare the time of inserting elements into those data structures. The author will be discussing how the problem was solved, how the different data structures was implemented and compare those data structures to each other.

Method

The author firstly started off by watching the last week lecture on trees. The lecture covered how a tree should be implemented using tuples and the basic methods of a tree such as inserting, removing, lookup, etc. The lecture gave sufficient knowledge to implement the tree data structure in elixir using recursion.

Result

The first step of the exercise is to implement two distinct functions: one for creating an empty list, and one for inserting elements. Because the program constantly inserts elements into the list in an order fashion, it becomes easy to implement. But because the insert method inserts in an ordered way the time complexity will be O(n). Figure one consists of both functions to create an empty list and the function for inserting into it.

The function for inserting an element into the list in an ordered fashion works by splitting the list with its head and body. If the head is bigger than the new element the program calls itself recursively with the new element and body. If the element is less or equal to the head, it gets added to the list.

```
def list_new() do [] end
def list_insert(e, []) do [e] end
def list_insert(e, [h | t]) do
  if e < h do
    [e | [h | t]]
  else
    [h | list_insert(e, t)]
  end
end
```

Figure 1: Creating and inserting elements into a list

The tree data structure is implemented using tuples, where there will be an atom as either a :node or a :leaf. If a node doesn't connect to another node its slot gets filled with a :nil atom. The tree structure will be following two different tuple forms, one as :node, value, left, right where left is element less than value and right consists of elements equal or greater than value. The other will be following the leaf structure as :leaf, value. Figure two consists of the function to create an empty tree by declaring a :node and its value as :nil. Because the tree data structure constantly will be cutting the amount of traversing in half the time complexity is O(log n) for inserting elements.

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

Figure 2: The function for creating a empty tree

The function for inserting an element into an ordered tree consists of three different situations. The author will be dividing the solution into three different parts, one for each situation, and discussing how the solution was implemented.

1. Empty or bottom of a tree

If either an empty tree or the bottom of a tree has been reached the goal is to create a leaf which consists of the atom :leaf and the value. Using this method all of the "outer elements" will have the atom :leaf and will not consist of any edges.

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

Figure 3: Function for traversing the tree: empty or bottom

2. Both left and right exists

When traversing the tree to find the correct position to insert the new element, the second situation is where both the left and right node already exists. In this situation it is needed to compare the new value with the current node value and if the value is less than the current node the tree will recursively navigate left. If its bigger or equal it will continue navigating right in the tree.

```
def tree_insert(e, {:node, value, left, right}) do
  if e < value do
    {:node, value, tree_insert(e, left), right}
  else
    {:node, value, left, tree_insert(e, right)}
  end
end
```

Figure 4: Function for traversing the tree: If both left and right exists

3. Reaching a leaf when traversing

When traversing the tree and a leaf has been located the goal is to create a node out of it and insert a new leaf in the correct position by comparing the node value and the new value. If the value is less than the leaf value, it will be stored in the left child while if the value is bigger or equal to the leaf value it will be stored in the right child.

```
def tree_insert(e, {:leaf, value}) do
  if e < value do
    {:node, value, {:leaf, e}, :nil}
  else
    {:node, value, :nil, {:leaf, e}}
  end
end
```

Figure 5: Function for traversing the tree: Reaching a leaf

Comparison

Figure six consists of a table which is the run time of the list and the tree data structure. In the column ratio it can be easily seen that the list data structure grows extremely fast relatively with the tree. As discussed above this is because the insertion for the list is O(n) while the tree structure is O(log n). In the figure the ratio explains how many times faster the run

time of the tree is compared to the run time of the list. The bench program did not actually count any time up until 0.102ms and therefore no ratio is declared for them.

Amount	**Run time List (ms)**	**Run time Tree (ms)**	**ratio**
16	0	0	-
32	0	0	-
64	0	0	-
128	0.10	0	-
256	0.20	0	-
512	0.51	0.10	5.0
1024	2.05	0.10	20.1
2048	9.52	0.41	23.3
4096	40.55	1.54	26.4
8192	198.14	3.79	52.3

Figure 6: The benchmark between list and tree data structure in milliseconds

Discussion

As predicted, because the goal when inserting into both the list and the tree is to have an ordered fashion the time complexity between the two different data structures is big. For example, in the last two tests the list grows 480 percent in time while the tree only grows about 246 percent (4096 elements to 8192 elements). This is because the tree structure constantly will be cutting the comparison between elements in half after comparing with an element. But in some cases, the tree could become extremely unbalanced where the height of the tree could equal the size of the tree. This is a problem which could be solved by creating a function for re-balancing the tree where the goal is to select the root as the most middle element. But because in these tests the bench function generates a number between 0 and 100 000 it becomes relatively low chance of the tree becoming extremely unbalanced. But if the test is being run multiple times it can be seen in some cases that the run time of the tree becomes much larger.