Tree vs List

Lucas Larsson

Spring Term 2022

Introduction

As the title indicates this is a comparison report between Binary Trees and linked lists, below will be a brief discussion for each part and then the results after running operation on both data structures.

List

Not much to say about the lists, it is one of the fundamental data structures, so it is implemented in almost all programming languages, so it was not difficult to build it in Elixir, i basically just needed to look up the syntax that i learned on the first week of this course. A code snippet below:

```
def new_list() do [] end

def list_insert(e, []) do [e] end
def list_insert(e, [h|t]) when e <= h do [e, h|t] end
def list_insert(e, [h|t]) do [ h|list_insert(e,t)] end
```

as mentioned above pretty straight-foreword implementation, first function creates an empty list, the other three handles insertion, the functions state all insertion states, if the list is empty just insert the element, otherwise traverse the list until the element can be inserted in it's ordered position.

Tree

Trees are basically lists but are arranged in specific way "Balanced" as they are called, the arraignment is what give them their name and special properties.

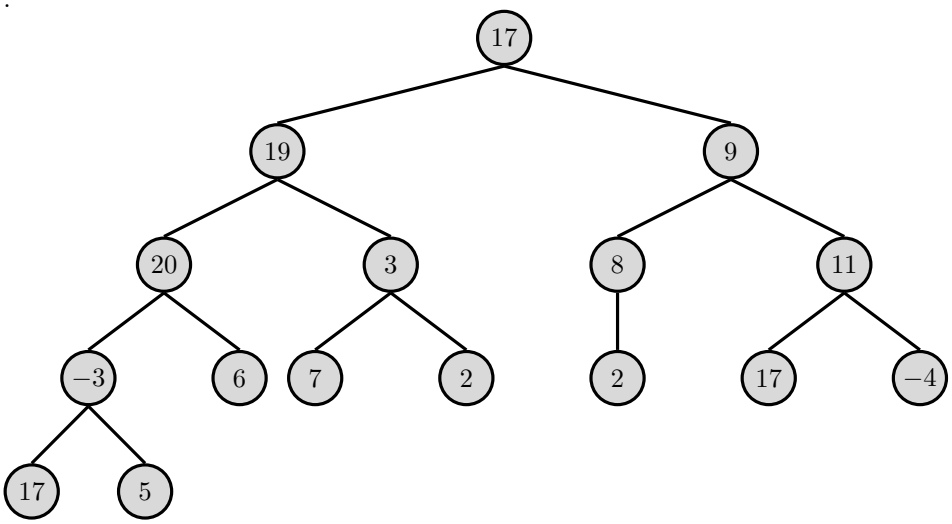
The implementation here was a bit harder to grantee getting a balanced tree, i have worked with binary tree before so i had an idea of how it would look, i watched the lectures given by the professor and they were very helpful. a snippet of the code below:

```elixir
def new_tree() do :nil end

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

similar to the previous shown implementation, first is a method to create a new tree, and then more insert methods handling the input of elements in order to balance the tree. above is shown 2 methods but in total i needed 5 methods.

.



A graph showing how a balanced tree can look like.

Results

Before checking the results i thought we could have a little comparison between trees and lists. Most important thing to check for in any data-structure is the Big O complexity,i.e how does the data structure preform for arbitrary value of data, here we talk only about the time and not the space complexity.

It is due to the trees unique architecture that they are able to deliver such preforms. Below is a table showing the Big O for both data/structures, but just by looking at them i find it difficult to understand the deference. A just classic i try to remember is how long it takes for a search or an insertion in a structure of 1 million element for a list that is a million comparison operations meanwhile it is 19 a tree structure. .

Finally a table showing the run time of the two data structures on my local machine with the size of `N`, the time is in milliseconds except for the last 3 rows that are in seconds for a better understanding of the time. It is clearly noticeable in the last row where the list takes almost a minute to complete a task that takes 1 second on a tree, the first couple of rounds the tree takes more

	`List`	`Tree`
Insert	$O(1)$	$O(log(n))$
Search	$O(n)$	$O(log(n))$
Delete	$O(1)$	$O(log(n))$

Table 1: Big O representation

time than the list, but that just due to allocating memory and other system calls that need to be preformed for any operation Independent of the data structure.

N	**Function** $[ms]$	
	`list_insert`	`tree_insert`
2^7	0,30	0,37
2^8	0,77	1,13
2^9	3,3	2,5
2^{10}	15,3	6,2
2^{11}	52,2	13,8
2^{12}	203	32
2^{13}	812	106
2^{14}	3 sec	0,3 sec
2^{15}	14 sec	0,4 sec
2^{16}	55 sec	1,2 sec

Table 2: A Table showing run time