# Tree vs List

Azer Hojlas

Spring Term 2022

## Introduction

In this assignment we are supposed to create an ordered list and an order tree with various input sizes. Because you already understand your code I will only mention the benchmark briefly. Because the writer was unsure of the term "ordered tree", two separate versions of said tree were implemented. One I developed on my own, and the other I took from a classmate. It seemed to me that the person had implemented a unique tree with sinking leaves which was really exciting, but I have no way of actually proving if it is ordered. Regardless of classification, it performs satisfactory and ordered operations can be performed on it. Hence the writers tree will be called BST (Binary Search Tree) and the classmates sinking leaves tree.

## Benchmark

The toughest part was not actually implementing the data structures, but to understand the benchmark and how it worked. For an inexperienced programmer, even after going through it thrice, there were occasional lapses in understanding. From what has been understood, it inserts elements, to a data structure from a list, 100 times where the elements are random integers uniformly distributed in the range of 1 to 100 000. The length of this list is determined by yet another list: [16,32,64,128,256,512,1024,2*1024,4*1024,8*1024], where each successive element is mapped to the benchmark with Enum, i.e the length of the list that is to be inserted in both data structures changes with each iteration of the list above. We measure the runtime of the insert operations of both data structures, by timing how long it takes to insert the random list 100 times. The runtime is measured and outputed 10 times, due to the length of the list above.

The benchmark did not include a dummy function call, and the author did not implement one either.

## Code explanation

Here I explain my code structures.

### List-implementation

The basic list implementation will be explained first. The bench initially creates an empty list, after which elements start filling it. The base case simply fills an empty list with the element in question. This base case is applicable to two scenarios. Either the list, as mentioned above, is initially empty, or the program has recursed through a non-empty list where the last tail is empty. In the recursive clause, if the element that is to be inserted is bigger than the head, the program continues down the list, otherwise the element gets appended before the head in its appropriate position. As the runtime quadruples when the input doubles, it would suggest a time-complexity of $O(n^2)$.

```
def list_new() do [] end

def list_insert(e, []) do [e] end #If list is empty, insert
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
```

### Sinking leaves tree

The following is the sinking leaves tree. It is unique due to the fact that leaf structures rise and sink. That is, whenever a new element is inserted unto a leaf, the existing leaf element sinks either to the left or right branch depending on its size relative to the element. It has been found useful when elements are inserted in perfect order, besides that no other considerable advantage or disadvantage has been established. Below follows and illustration by example: Say a client inputs a list in perfect order into a BST. The BST will for all nodes have empty left branches, thus the time complexity for insert/access operations is O(n). This implementation however, allows for sinking leaves, in which newly inserted elements will employ both branches at the bottom of the tree.

```
def tree_new() do :nil end

def tree_insert(e, :nil) do {:leaf, e} end
def tree_insert(e, {:leaf, head} = right) when e < head do {:node, e, :nil, right} e
def tree_insert(e, {:leaf, _} = left) do {:node, e, left, :nil} end

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

```
    {:node, head, left, tree_insert(e, right)}
end
```

In order to illustrate this not so easy to understand point, the following different tree structures are submited for BST and the sinking leaves tree with input (4, 2, 6, 3). The author has no proof for what data structure is better with regards to tree height and time-complexity, as is confirmed by the results.

```
Sinking leaves height: 4
{:node, 2, nil, {:node, 6, {:node, 3, nil, {:leaf, 4}}, nil}}

BST height: 3
{:node, 4, {:node, 2, nil, {:leaf, 3}}, {:leaf, 6}}
```

The node-processing function will be explained thoroughly in the BST implementation as they are identical for both trees. The authors solution only differs in leaf implementation.

### Binary Search tree

The following is the binary search tree. It is what would be expected from an ordered tree. The nodes can hold two possible values aside from the head, left and right. These can either be :nil (only one position can however be occupied by :nil) or a leaf with a non-null value (:leaf, e). In the base case, if a left- or right structure is nil and an element is added, it immediately converts into a leaf. In the other base case, if it isn't nil, it compares the respective values. If the inserted element is higher, it takes the right branch as a leaf, and vice versa if it's smaller. What was the leaf becomes a node because it connects to the new leaf (only nodes can hold branches). The recursive node function is straight-forward, if the inserted element is smaller, it gets passed along to the same function call to the left branch, if higher, to the right.

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

  def tree_insert_BST(e, {:leaf, head}) do
    case e <= head do
      true -> {:node, head, {:leaf, e}, :nil}
      false -> {:node, head, :nil, {:leaf, e}}
    end
  end

  def tree_insert_BST(e, {:node, head, left, right}) do
    case e <= head do
```

```
      true -> {:node, head, tree_insert_BST(e, left), right}
      false -> {:node, head, left, tree_insert_BST(e, right)}
    end
  end
```

## Tables

Here the runtimes are presented for each structure

| Length | List | Sinking leaves | BST | Fastest |
|---|---|---|---|---|
| 16 | 0,389 | 0,236 | 0,196 | BST |
| 32 | 0,92 | 0,376 | 0,381 | Sinking leaves |
| 64 | 2,41 | 1,25 | 1,33 | Sinking leaves |
| 128 | 6,32 | 1,86 | 1,95 | Sinking leaves |
| 256 | 18,9 | 4,70 | 6,16 | Sinking leaves |
| 512 | 74,9 | 12,6 | 12,7 | Sinking leaves |
| 1024 | 344 | 36,5 | 32 | BST |
| 2*1024 | 1460 | 105 | 90,1 | BST |
| 4*1024 | 8060 | 205 | 203 | BST |
| 8*1024 | 28400 | 484 | 556 | Sinking leaves |

Table 1: Runtime in milliseconds, 100 loops for each row with varying list length input, maximum of three significant numbers.

The tree structures were faster than the list in each iteration. Because the inputs are random values, the extremely small chance of the elements getting inserted in perfect order is negligible. The average time complexity for BST is log(n) due to the fact that the required amount of compares decreases ideally by half(due to branching), but on average the time-complexity is $2 * ln(n)$ (Sedgewick And Wayne, 2011). The list on the other hand, has an average time-complexity of (c*n), as the amount of compares necessary is more or less proportional to the input.

## Graphs

The author decided to not use gnuplot in this assignment even though it is preferable. Due to the fact that the author was unable to insert four different columns of value into GNUPLOT, it was easier to use google spreadsheets.

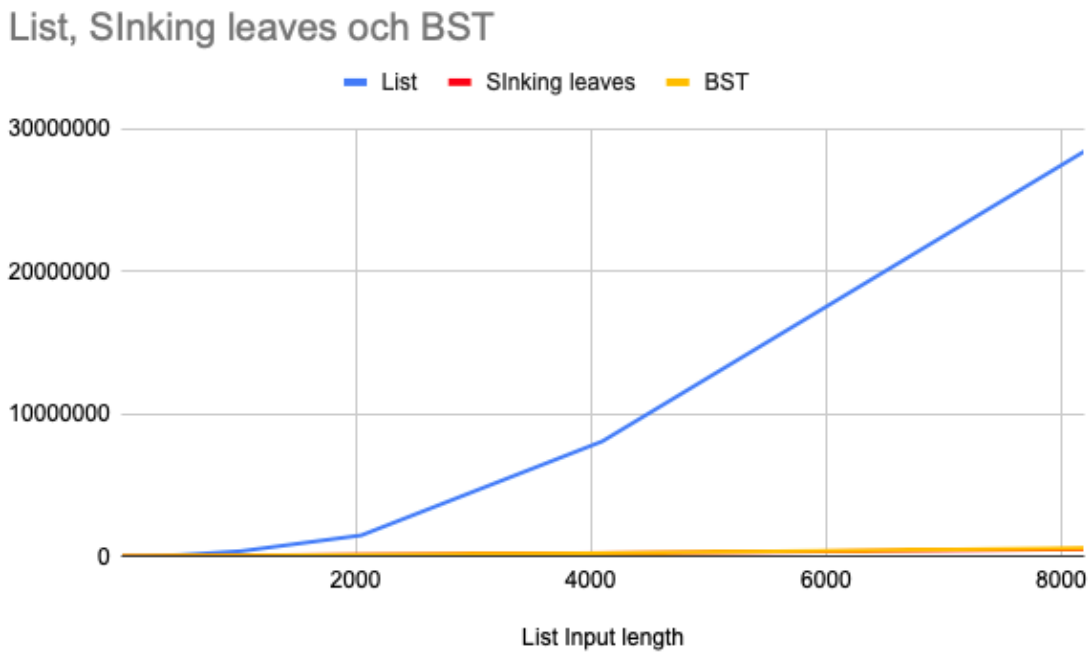


Figure 1: This is a the resulting runtime in microseconds; proportional scale

In order to not confuse the reader, another graph is submitted below with the y-axis logarithmically scaled in order to better compare the two tree structures and also because the graph above implies that said tree structures have constant time operations.

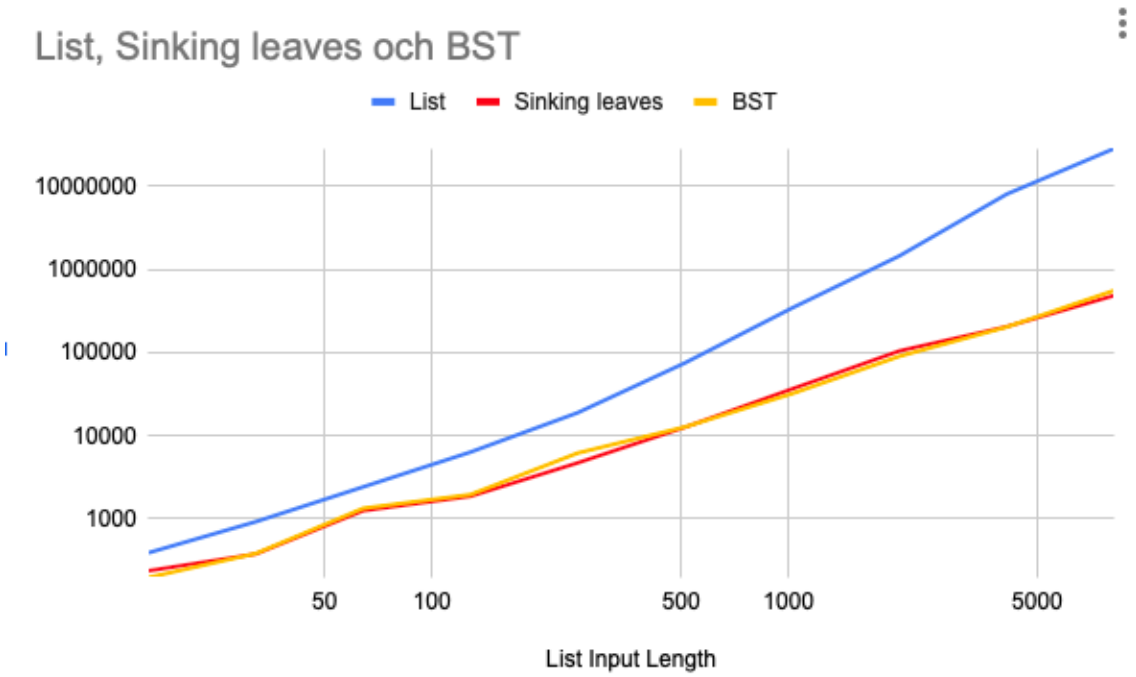


Figure 2: This is a the resulting runtime in microseconds; logarithmic y- and x-axis scale

For the sake of not making the report to long, there will be no in-depth comparison of the two tree structures. What is clear is that they are similar in runtimes and time complexity, as can be derived from the graph above.

## Conclusion

This assignment has been instructive with regards to what data structures developers should use in different situations. It has proved a truthful but often disregarded paradigm true, which is that one must always choose their data structures with great care. Furthermore, the separate tree structures seemed to be proportional to each other. Any feedback on this would be greatly appreciated.

## Reference list

Robert Sedgewick and Kevin Wayne, Algorithms: Compressed online edition `https://algs4.cs.princeton.edu/32bst/`