*Reading: Section 1-3 of paper by Hinze and Paterson*

# Finger Tree

Lecture 130 of Advanced Programming

Nov 29, 2018

**Andrzej Wasowski & Zhoulai Fu**

**IT University of Copenhagen**

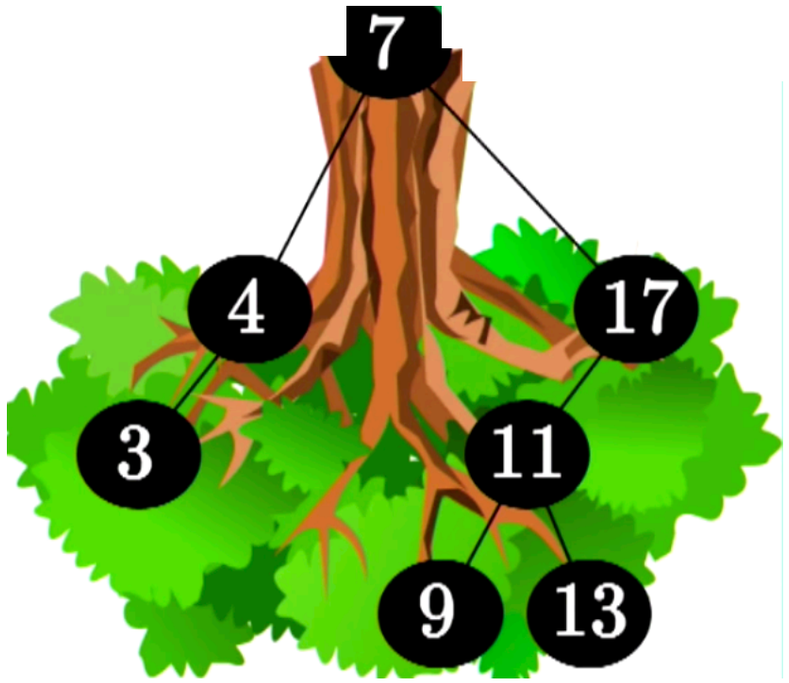
# *Finger trees: a simple general-purpose data structure*

RALF HINZE
Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* ralf@informatik.uni-bonn.de)

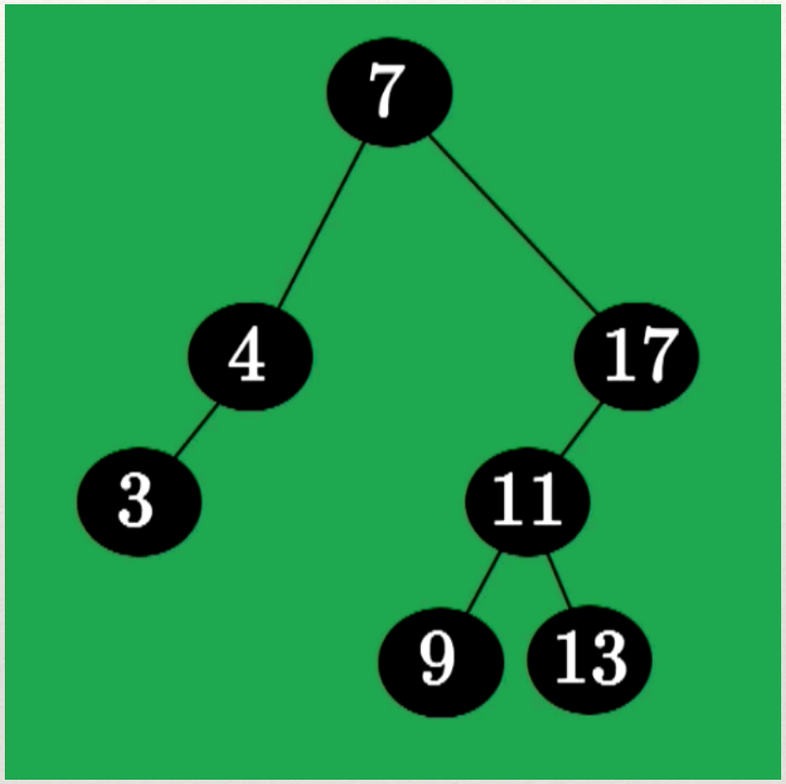
ROSS PATERSON
Department of Computing, City University
London EC1V OHB, UK
(*e-mail:* ross@soi.city.ac.uk)

# What is a tree?

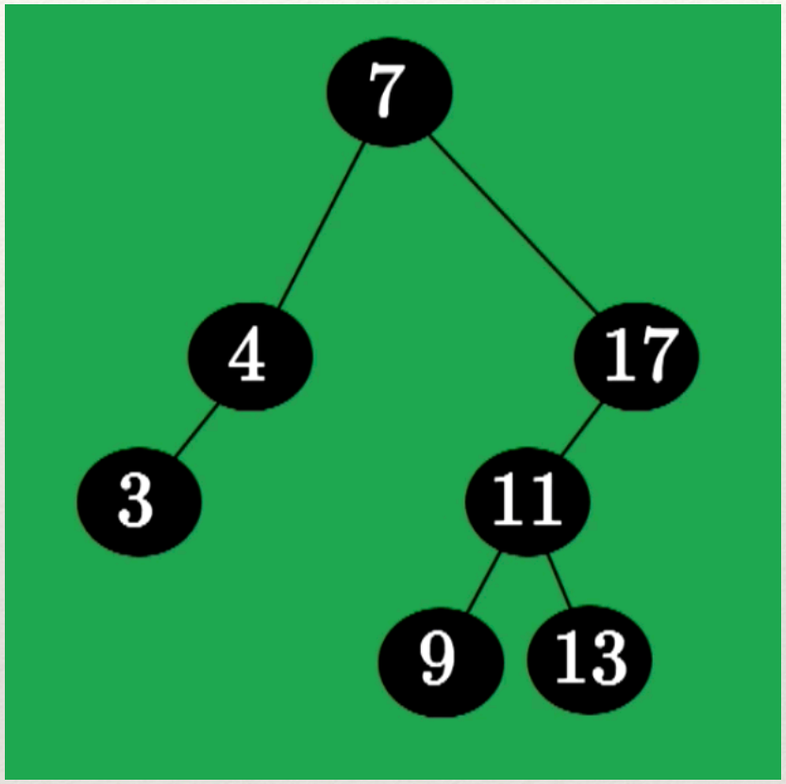
Material from: https://www.youtube.com/watch?v=C_q5ccN84C8



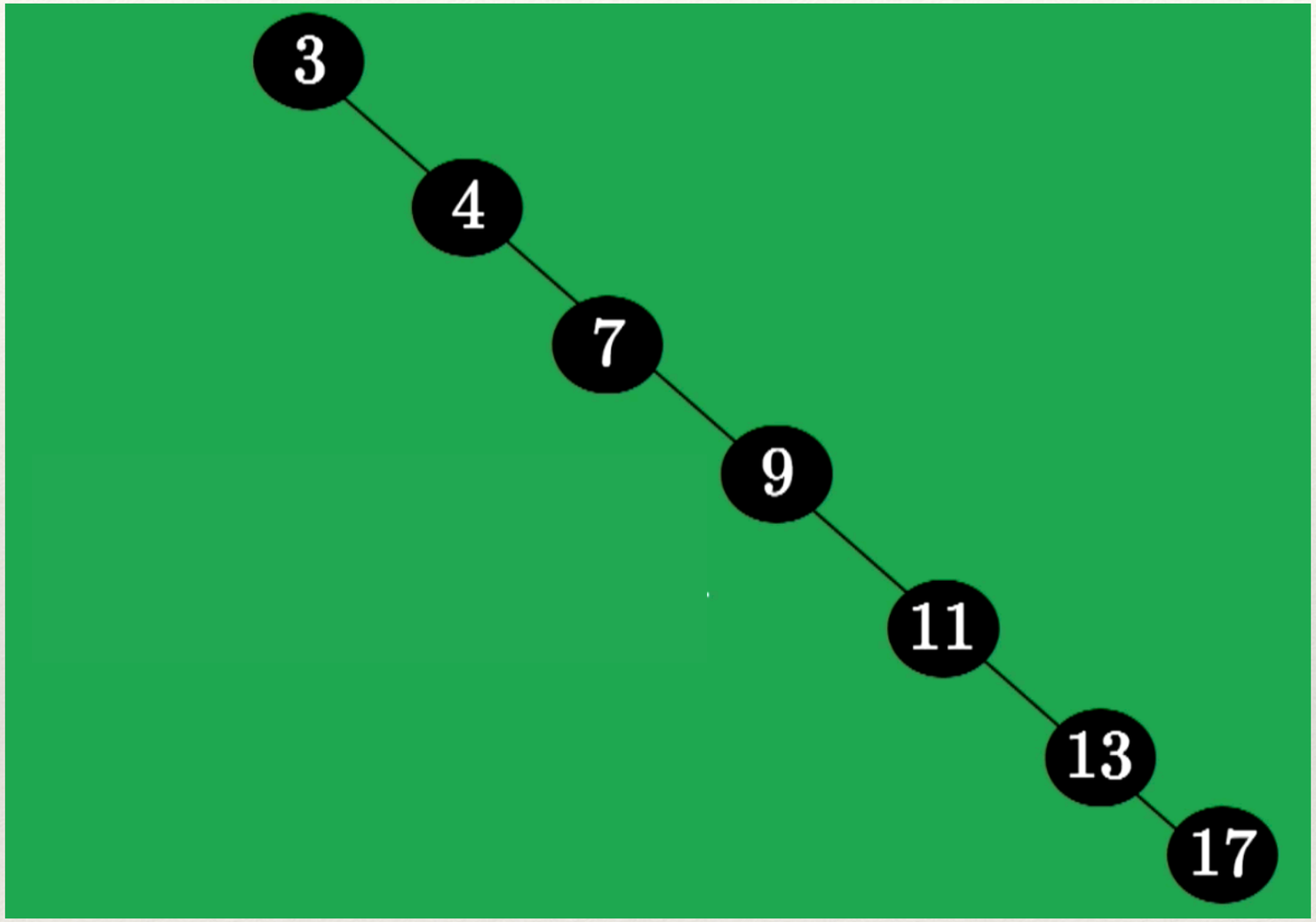

Tree: structure for data storage and retrieval




Binary search tree:

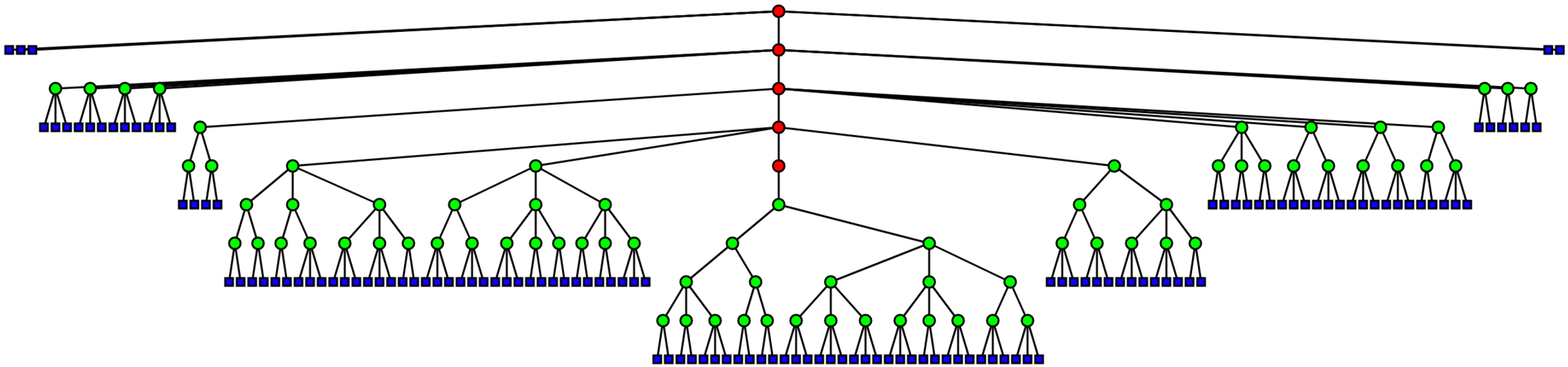



Question: complexity for inserting an element?



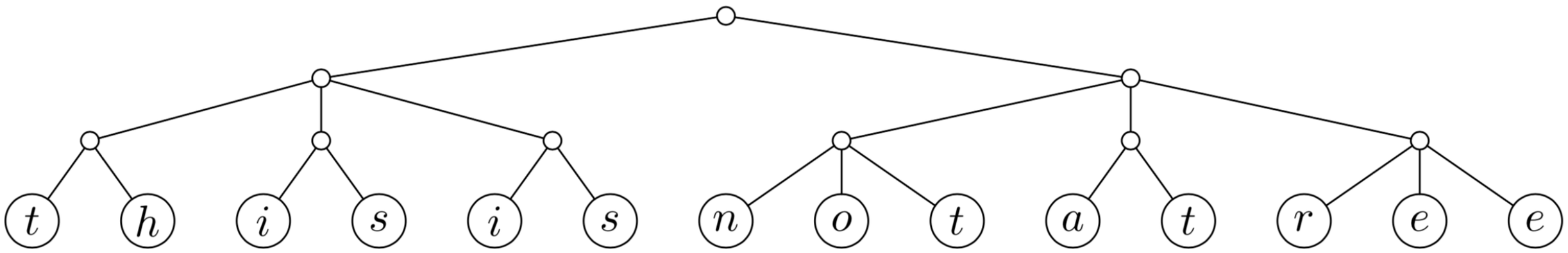

Worst case : O(n)

# We want to reduce O(n) to O(1)



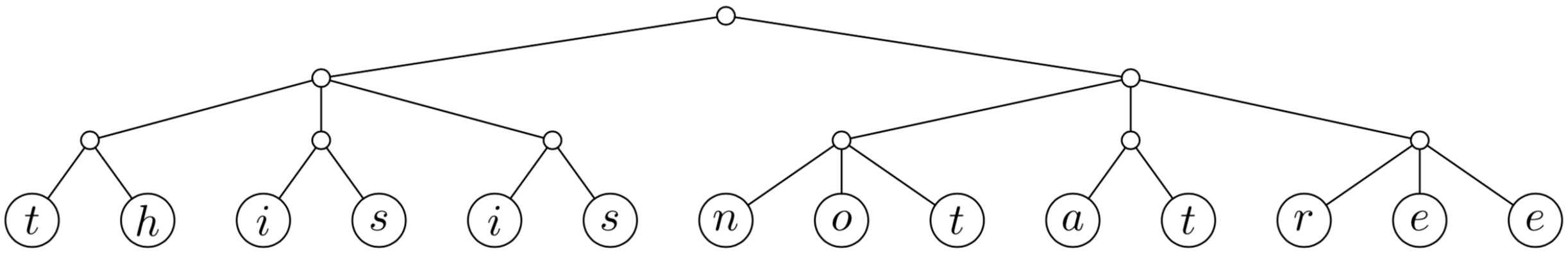
- Balanced.
- Nodes of Finger Tree — spine
- Nodes of 2-3 Tree
- Nodes of Digits — fingers

# Prerequisite: 2-3 Tree



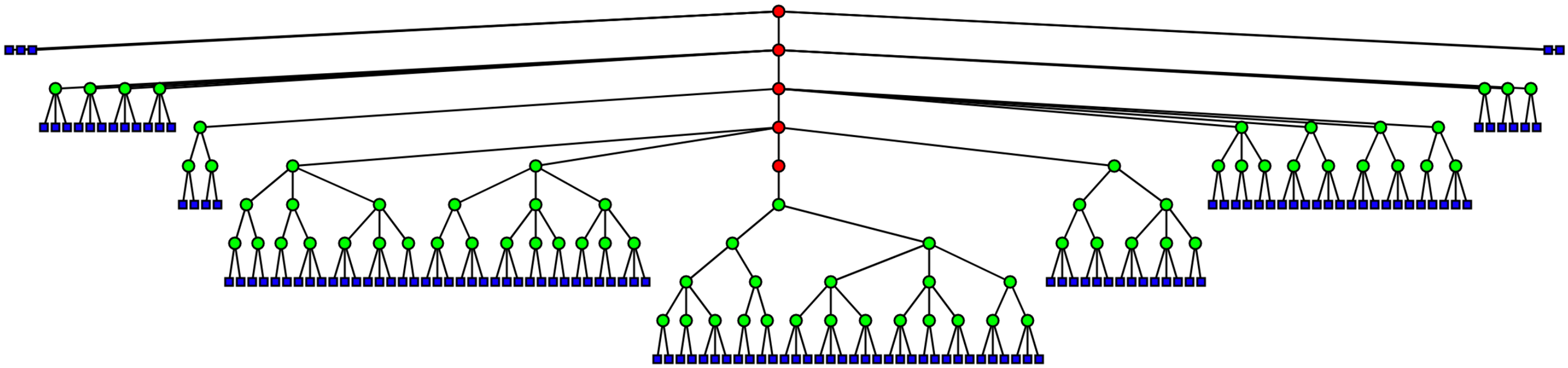

- Node has two or three leaves
- All leaves at the same level
- Naive representation: **Node a = Node2 a a | Node3 a a a**

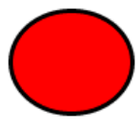
# Prerequisite: 2-3 Tree




- Naive representation: **Node a = Node2 a a | Node3 a a a**
- On-class exercise
  - Create the tree above with the naive representation
  - Point out the issue of this naive representation

# Finger Tree

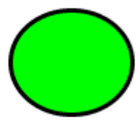




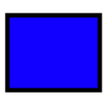
```
data FingerTree a = Empty
                  | Single a
                  | Deep (Digit a) (FingerTree (Node a)) (Digit a)
```

(none)

```
data Digit a      = One a | Two a a | Three a a a | Four a a a a
```



```
data Node a       = Node2 a a | Node3 a a a
```



elements of arbitrary type

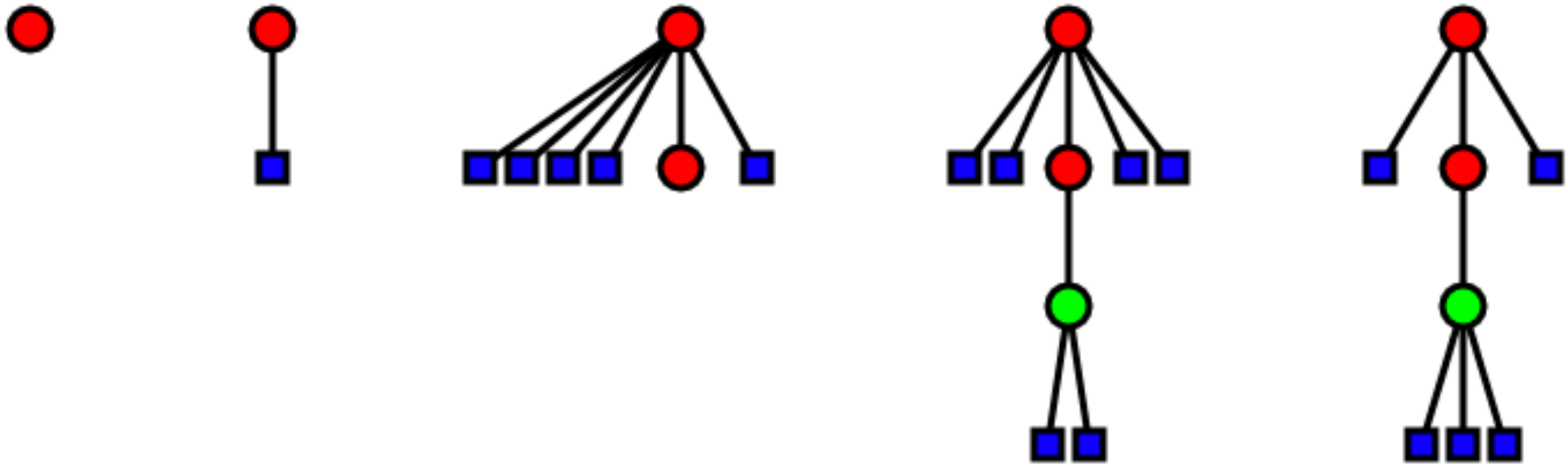
http://www.staff.city.ac.uk/~ross/papers/FingerTree.html

# Implementation (use Haskell as in the paper)

```haskell
-- As usual, the type parameter represents what type
-- of data is stored in this finger tree.
data FingerTree a
  = Empty        -- We can have empty trees.
  | Single a     -- We need a special case for trees of size one.

  -- The common case with a prefix, suffix, and link to a deeper tree.
  | Deep {
    prefix :: Digit a,                -- Values on the left.
    deeper :: FingerTree (Node a), -- The deeper finger tree, storing deeper
      2-3 trees.
    suffix :: Digit a                 -- Values on the right.
  }
```

# Examples



```
data FingerTree a = Empty
                  | Single a
                  | Deep (Digit a) (FingerTree (Node a)) (Digit a)
```

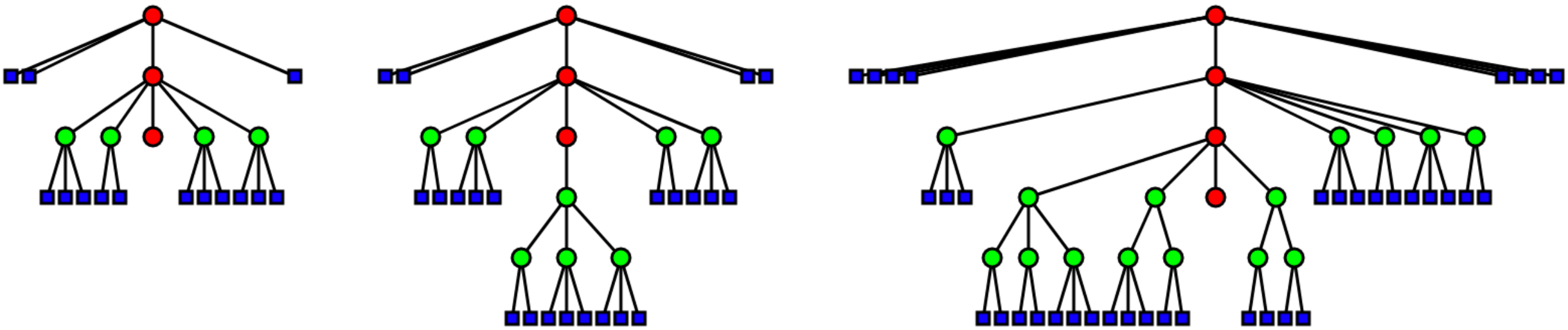
none)

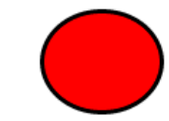
```
data Digit a      = One a | Two a a | Three a a a | Four a a a a
```

```
data Node a       = Node2 a a | Node3 a a a
```

elements of arbitrary type

# On-class Exercises: Create these Finger Trees

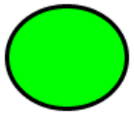




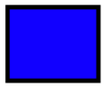
```
data FingerTree a = Empty
                  | Single a
                  | Deep (Digit a) (FingerTree (Node a)) (Digit a)
```

(none)

```
data Digit a      = One a | Two a a | Three a a a | Four a a a a
```



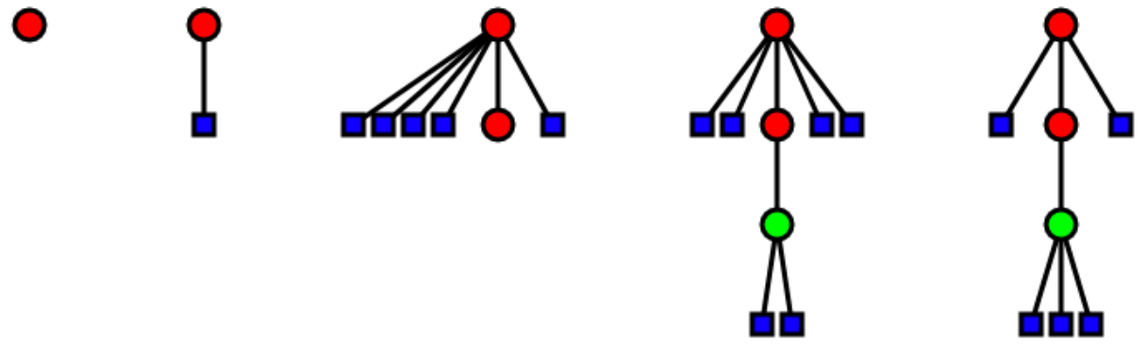
```
data Node a       = Node2 a a | Node3 a a a
```



elements of arbitrary type

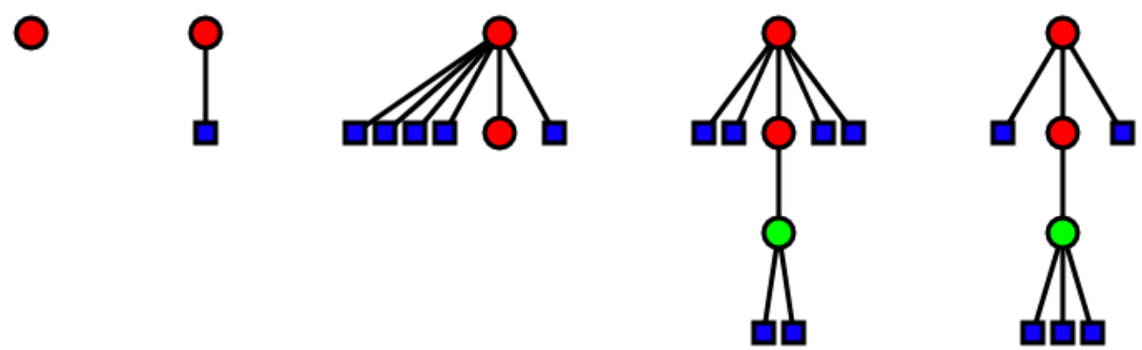
# Finger Tree as List

- The motivation behind finger tree was to have an efficient List-like data structure.
- A finger tree can be transformed to list of prefix, deep, and suffix recursively.
- What about operations?



# Prepend: Add an element to the front of a Finger Tree

- Ideally, prepending to the left hand side finger
- But it would *not* work on four-element fingers
- The trick is to *split*



# How to Prepend

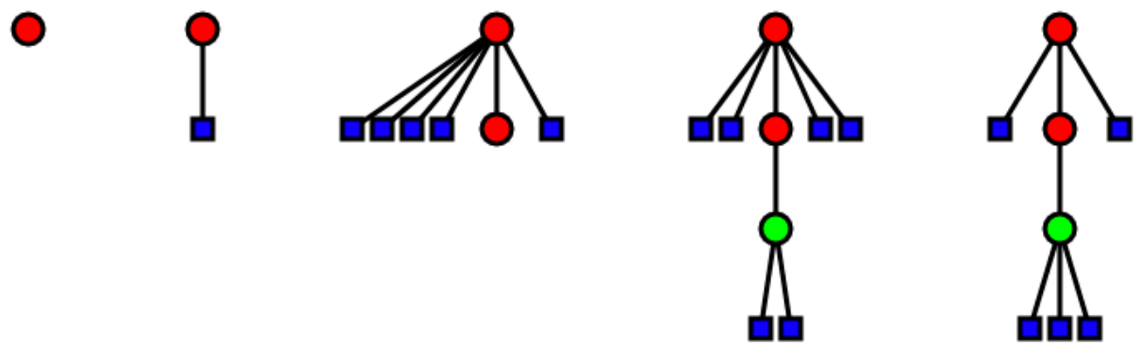
```haskell
-- Use <| to prepend. It's the finger tree analogue of : for lists.
infixr 5 <|
(<|) :: a -> FingerTree a -> FingerTree a

-- Base case #1: If this is an empty finger tree, make it one element.
x <| Empty = Single x

-- Base case #2: For a single tree, upgrade it to a deep one.
-- Remember that the list syntax is actually creating 'Affix a' values.
x <| Single y = Deep [x] Empty [y]

-- Recursive case: if we have a prefix with four elements, we have to
-- use the last 2 elements with the new one to create a node, and then
-- we prepend that node to the deeper finger tree which contains nodes.
x <| Deep [a, b, c, d] deeper suffix = Deep [x, a] (node <| deeper) suffix
  where
    node = Node3 b c d

-- Non-recursive case: we can just prepend to the prefix.
x <| tree = tree { prefix = affixPrepend x $ prefix tree }
```

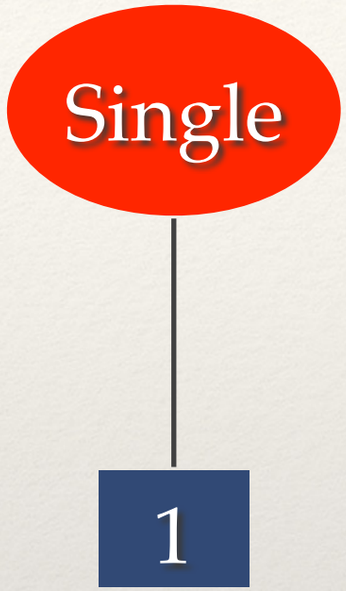


# Example (blackboard)

- Starting from Empty, prepend 1,2,3, until 9

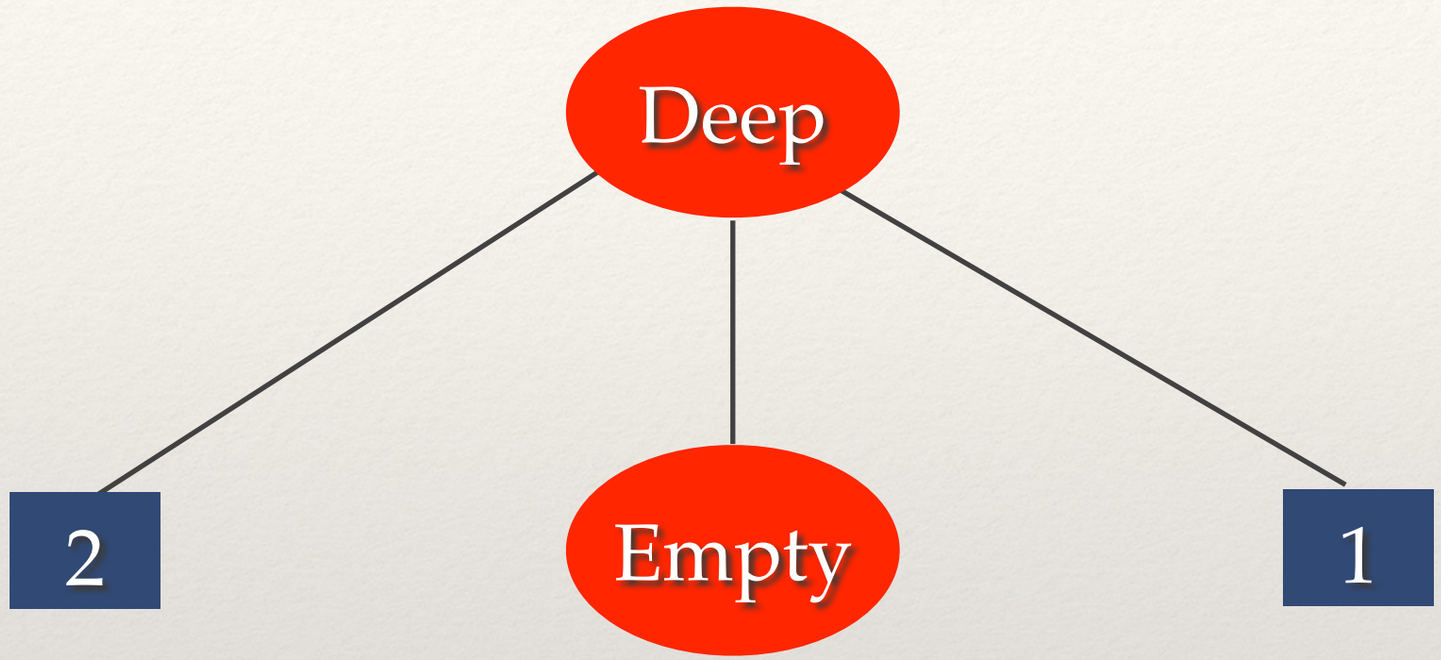
# At the very beginning

# Prepending 1 =>
# Single(1)



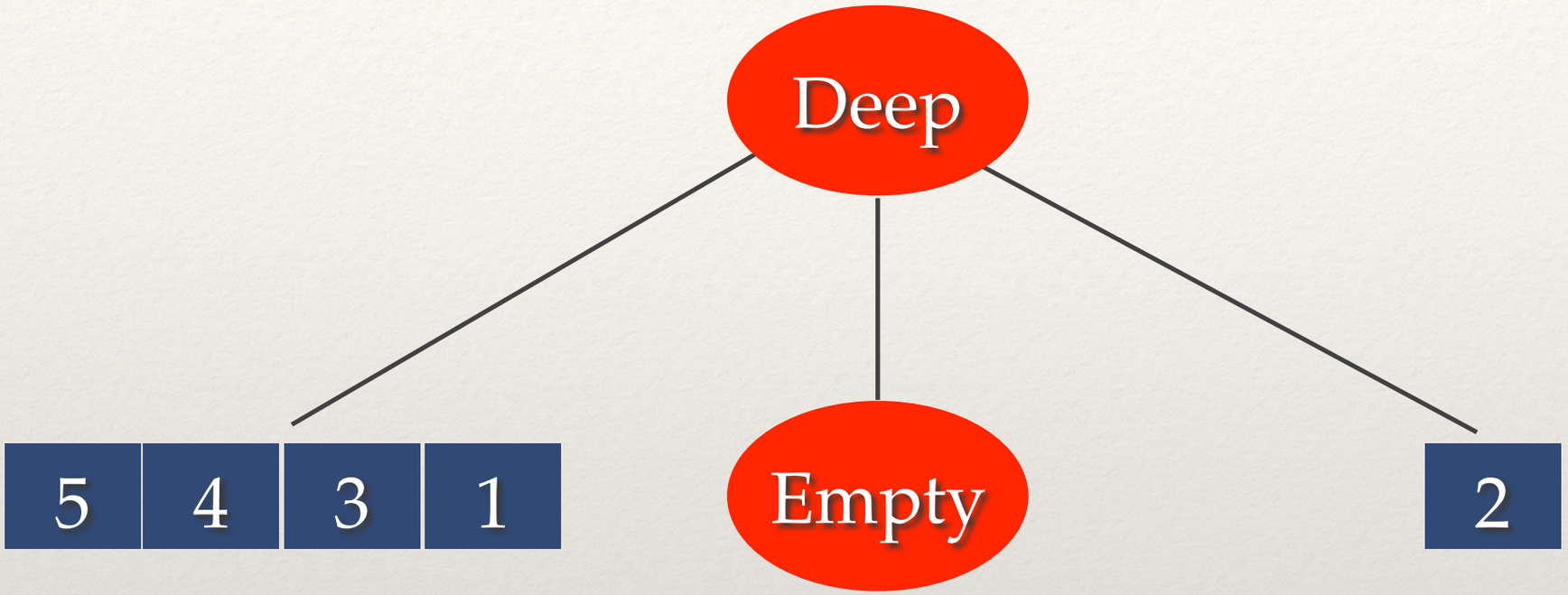

```
-- Base case #1: If this is an empty finger tree, make it one element.
x <| Empty = Single x
```

Prepending 2 =>
Deep(One(2),Empty,One(1))



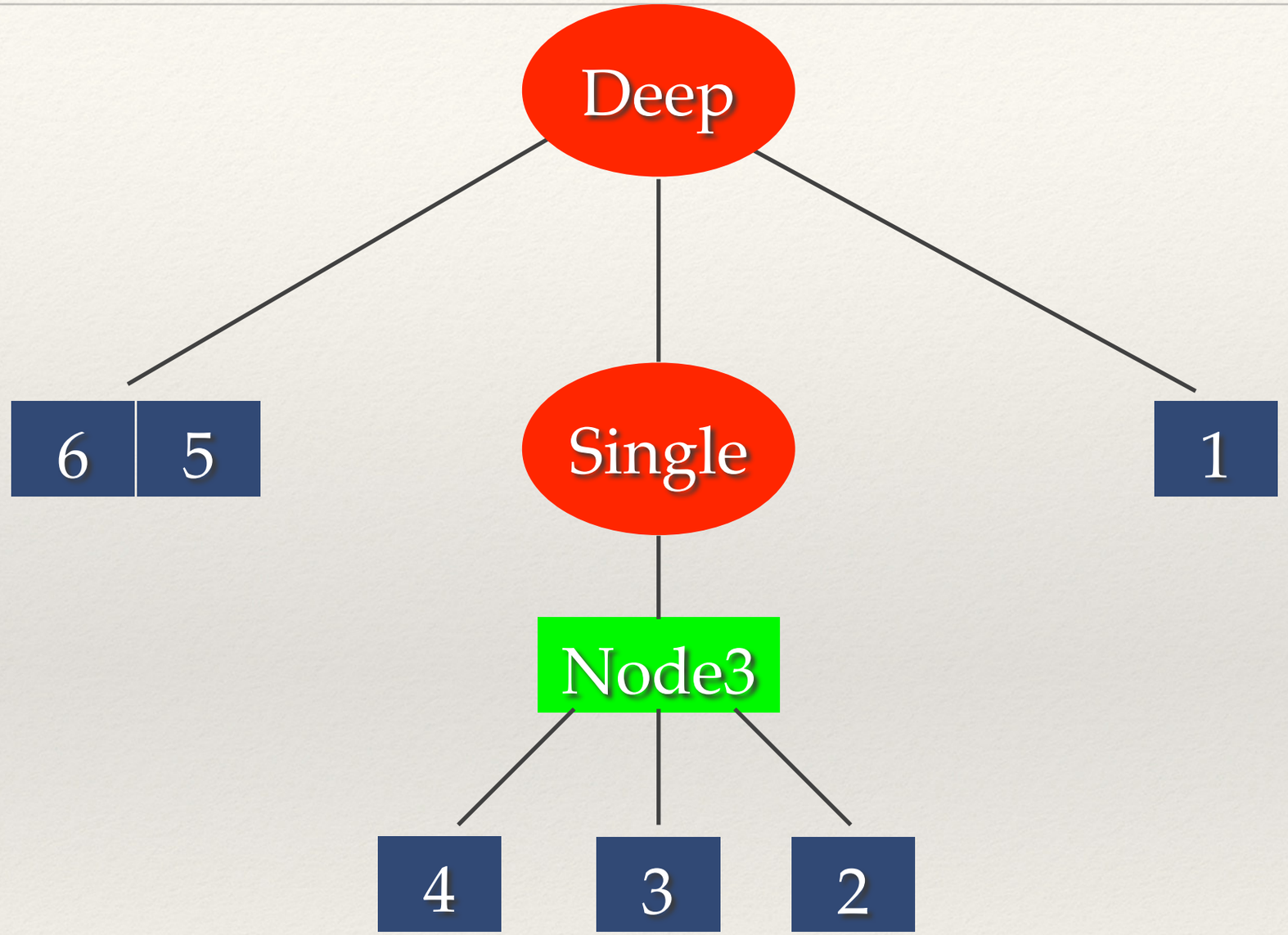

```haskell
-- Base case #2: For a single tree, upgrade it to a deep one.
-- Remember that the list syntax is actually creating 'Affix a' values.
x <| Single y = Deep [x] Empty [y]
```

# Prepending 3,4,5 =>
# Deep(Four(5,4,3,2),Empty,One(1))



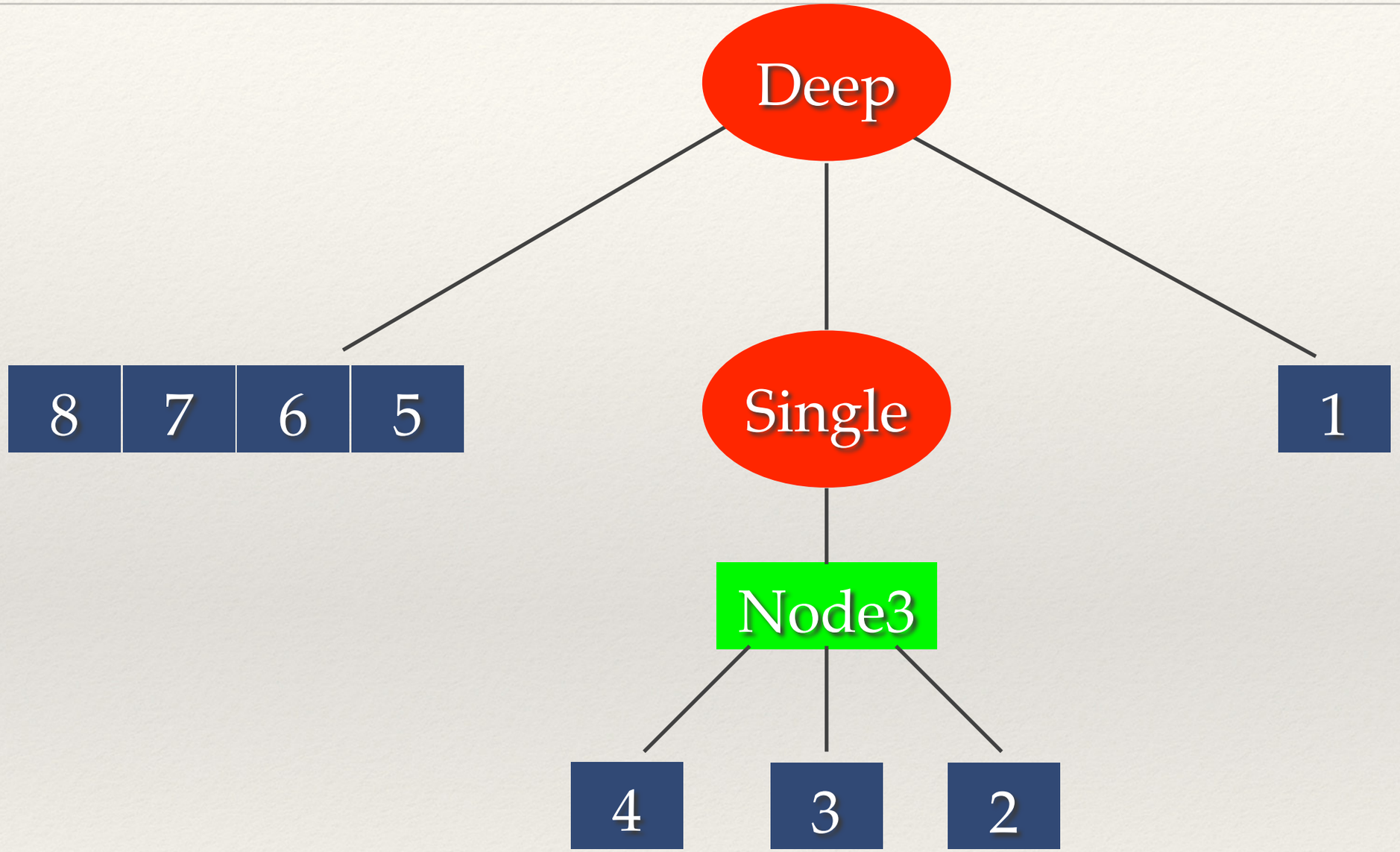

```
-- Non-recursive case: we can just prepend to the prefix.
x <| tree = tree { prefix = affixPrepend x $ prefix tree }
```

# Prepending 6 =>
# Deep(Two(6,5),Single(Node3(4,3,2)),One(1))



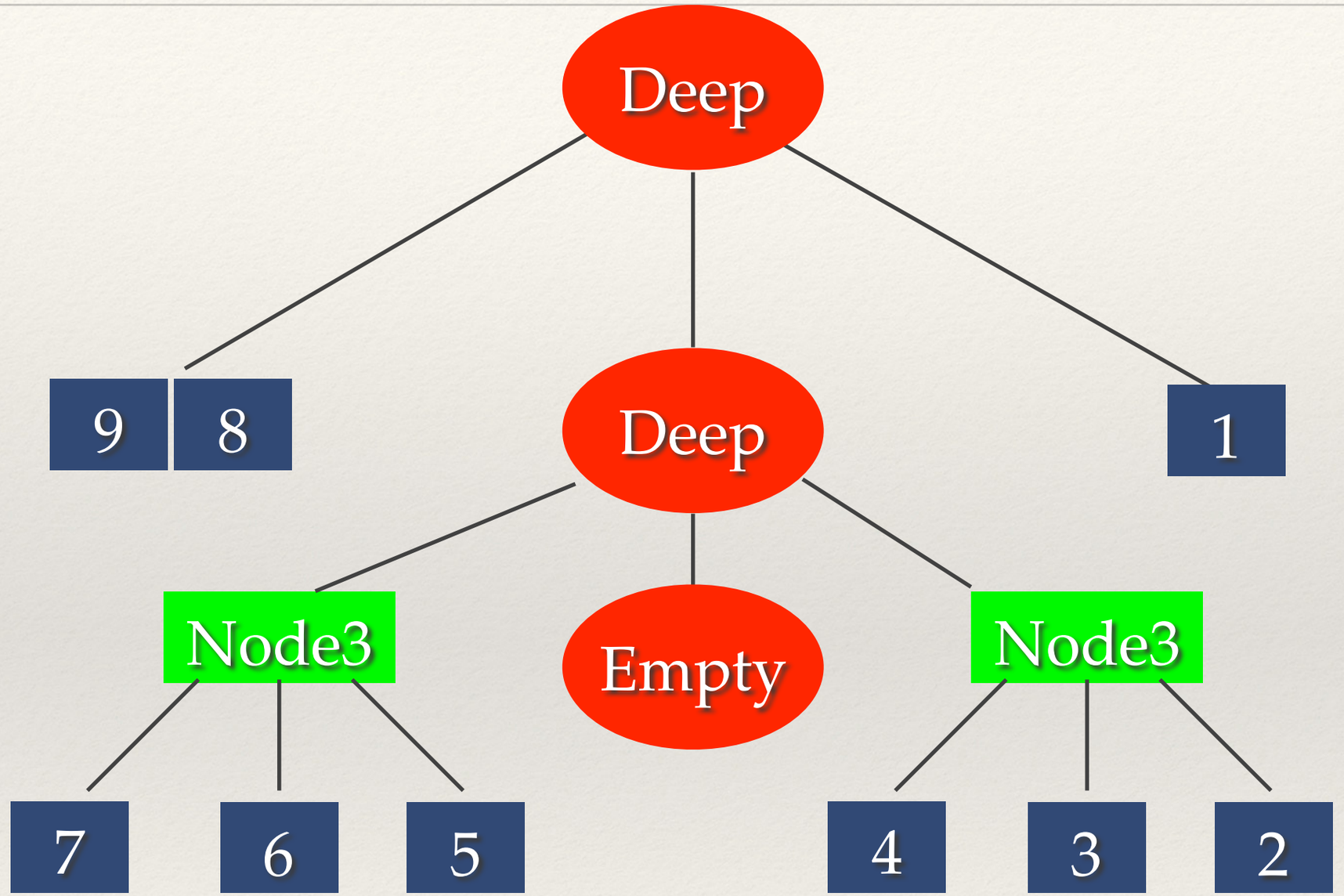

```
-- Recursive case: if we have a prefix with four elements, we have to
-- use the last 2 elements with the new one to create a node, and then
-- we prepend that node to the deeper finger tree which contains nodes.
x <| Deep [a, b, c, d] deeper suffix = Deep [x, a] (node <| deeper) suffix
  where
    node = Node3 b c d
```

# Prepending 7,8 =>
# Deep(Four(8,7,6,5),Single(Node3(4,3,2)),One(1))




```
-- Non-recursive case: we can just prepend to the prefix.
x <| tree = tree { prefix = affixPrepend x $ prefix tree }
```

Prepending 9 =>
Deep(Two(9,8),Deep(One(Node3(7,6,5)),Empty,One(Node3(4,3,2))),One(1))




```
-- Recursive case: if we have a prefix with four elements, we have to
-- use the last 2 elements with the new one to create a node, and then
-- we prepend that node to the deeper finger tree which contains nodes.
x <| Deep [a, b, c, d] deeper suffix = Deep [x, a] (node <| deeper) suffix
  where
    node = Node3 b c d
```

# Complexity

- An insertion can only propagate to the next level if the Digit is full
- **At most half operations descend one level**
- In n operations 1 + 1/2 + 1/4 + ... + 1/2^n... <= 2
- ==> *Amortized* cost is constant.

# Conclusion

## Finger Tree:

## An efficient and general-purpose data structure