Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Functioneel programmeren 2012-2013

134. Finger Trees

Atze Dijkstra and Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

December 18, 2012

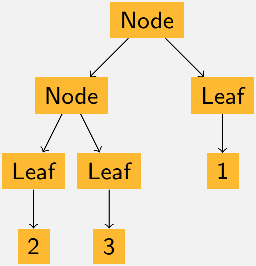
Finger trees

- A general purpose data structure, reminiscent of a Swiss army knife.
  It can be used as:
  - a sequence (split and concatenate, access to both ends in constant time)
  - a priority queue (find the minimum)
  - a search tree (find an element)
  - ...
- Specialized data structures are often slightly more efficient, but finger trees are competitive.
- Available in Data.Sequence.

Tree-like structures

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

Simple Haskell trees are not always balanced:

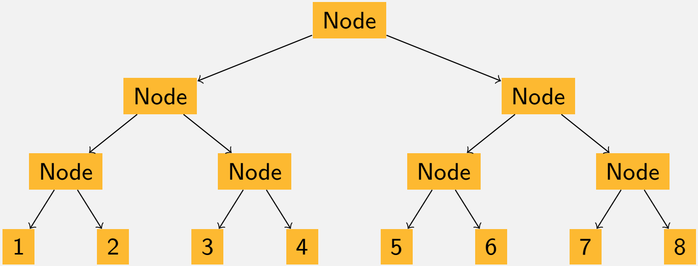


Balanced trees

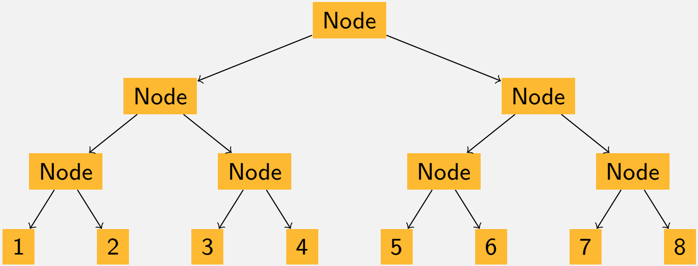
Idea

Let us use Haskell's type system to enforce that trees are balanced.

Example: a balanced complete tree

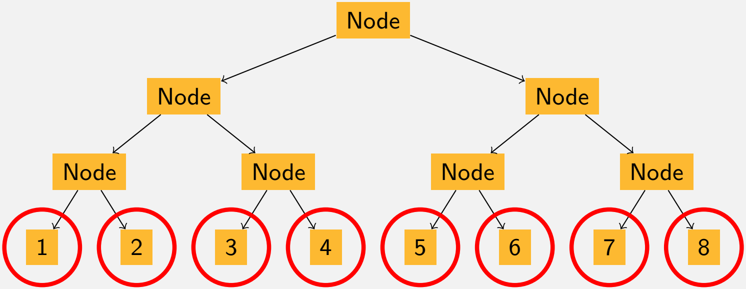


Example: a balanced complete tree



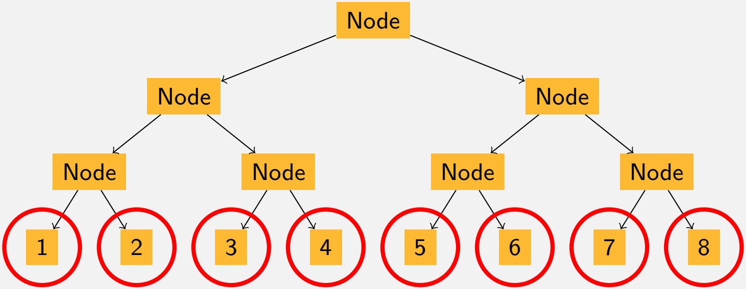
What are the leaves?

Example: a balanced complete tree



What are the leaves?

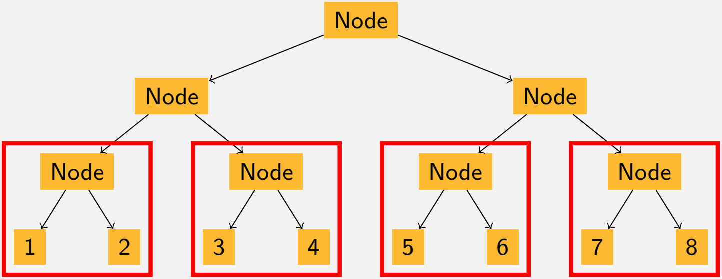
Example: a balanced complete tree



What are the leaves?

Can we define trees that have other trees as leaves?

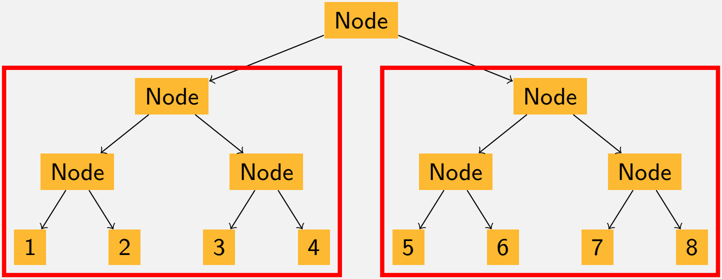
Example: a balanced complete tree



What are the leaves?

Can we define trees that have other trees as leaves? Yes, of course – the type of leaves is just a parameter.

Example: a balanced complete tree



What are the leaves?

Can we define trees that have other trees as leaves? Yes, of course – the type of leaves is just a parameter.

Trees of a fixed depth

$$
\begin{array}{ll}
\textbf{type}\ \mathsf{Tree}_0\ \mathsf{a} = \mathsf{a} & \\
\textbf{type}\ \mathsf{Tree}_1\ \mathsf{a} = \mathsf{Node\ a} & = \mathsf{Tree}_0\ (\mathsf{Node\ a}) \\
\textbf{type}\ \mathsf{Tree}_2\ \mathsf{a} = \mathsf{Node\ (Node\ a)} & = \mathsf{Tree}_1\ (\mathsf{Node\ a}) \\
\textbf{type}\ \mathsf{Tree}_3\ \mathsf{a} = \mathsf{Node\ (Node\ (Node\ a))} & = \mathsf{Tree}_2\ (\mathsf{Node\ a}) \\
\ldots & \\
\textbf{data}\ \mathsf{Node\ a} = \mathsf{Node\ a\ a} \quad \text{-- a node is a pair!} &
\end{array}
$$

Nested datatypes

Complete trees of a certain depth:

```
type Tree_0     a = a
type Tree_{1+n} a = Tree_n (Node a)
data Node a = Node a a    -- a node is a pair!
```

Nested datatypes

Complete trees of a certain depth:

$$\textbf{type}\ \mathsf{Tree}_0\ \mathsf{a} = \mathsf{a}$$
$$\textbf{type}\ \mathsf{Tree}_{1+\mathsf{n}}\ \mathsf{a} = \mathsf{Tree}_{\mathsf{n}}\ (\mathsf{Node\ a})$$

```
data Node a = Node a a   -- a node is a pair!
```

Combined into a single datatype:

```
data Tree a = Zero a
            | Succ (Tree (Node a))
```

Trees of this datatype are always complete! What's strange about this type?

Nested datatypes

Complete trees of a certain depth:

```
type Tree_0     a = a
type Tree_{1+n} a = Tree_n (Node a)
data Node a = Node a a   -- a node is a pair!
```

Combined into a single datatype:

```
data Tree a = Zero a
            | Succ (Tree (Node a))
```

Trees of this datatype are always complete! What's strange about this type?

Datatypes with non-regular recursion such as Tree are also called nested datatypes.

Example

```
t :: Tree Int
t = Succ (Succ (Succ (Zero (Node (Node (Node 1
                                             2)
                                       (Node 3
                                             4))
                                 (Node (Node 5
                                             6)
                                       (Node 7
                                             8))))))
```

Example

```
t :: Tree Int
t = Succ (Succ (Succ (Zero (Node (Node (Node 1
                                             2)
                                       (Node 3
                                             4))
                                 (Node (Node 5
                                             6)
                                       (Node 7
                                             8))))))
```

The constructors Succ and Zero encode the number of levels in the tree.

Towards 2-3-trees

- Complete binary trees are too limited.
- The number of elements in a complete binary tree is always a power of two.
- It is therefore difficult to implement basic functions such as insertion of a single element – we need more flexibility.

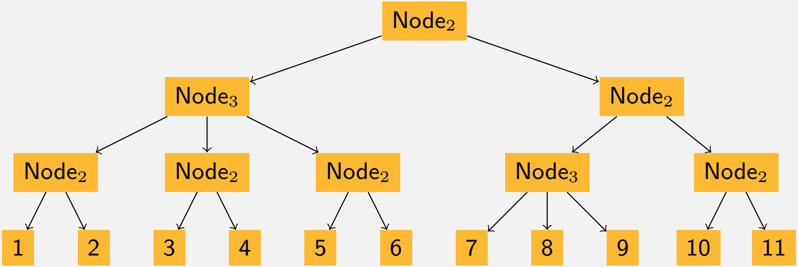
Towards 2-3-trees

- Complete binary trees are too limited.
- The number of elements in a complete binary tree is always a power of two.
- It is therefore difficult to implement basic functions such as insertion of a single element – we need more flexibility.

2-3-trees

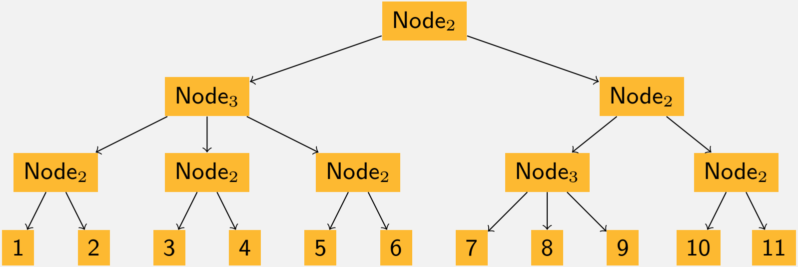
Complete trees with values at the leaves where every node has either two or three children.

A 2-3-tree



Node₂
Node₃
Node₂
Node₂
Node₂
Node₂
Node₃
Node₂
1
2
3
4
5
6
7
8
9
10
11

A 2-3-tree




```
data Tree a  = Zero a
             | Succ (Tree (Node a)) -- as before
data Node a = Node₂ a a
             | Node₃ a a a          -- pair or triple
```

Number of elements in a 2-3 tree

depth (n)	min elements (2^n)	max elements (3^n)
0	1	1
1	2	3
2	4	9
3	8	27
...		

Number of elements in a 2-3 tree

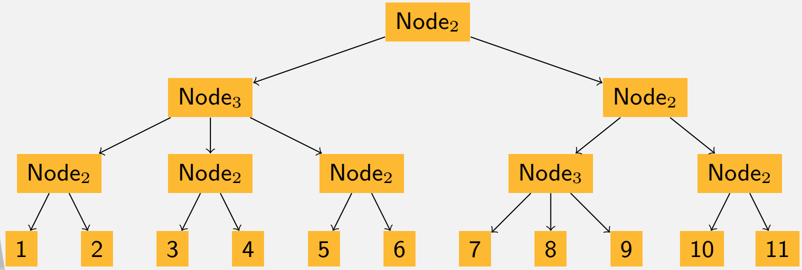
depth (n)	min elements (2^n)	max elements (3^n)
0	1	1
1	2	3
2	4	9
3	8	27
...		

Every number of elements can be represented.

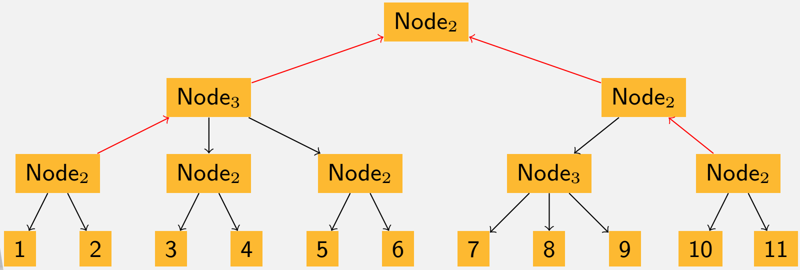
Finger trees

- 2-3-Trees already give us logarithmic access to all elements.
- For sequence operations, we want access to both ends in constant time.
- Finger trees are a reorganisation of 2-3-Trees.

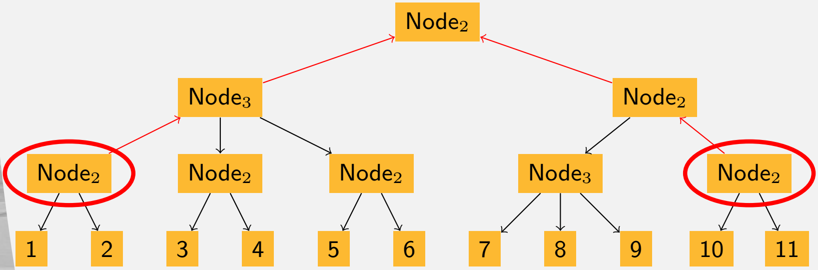
Introducing a "finger" – pointer reversal



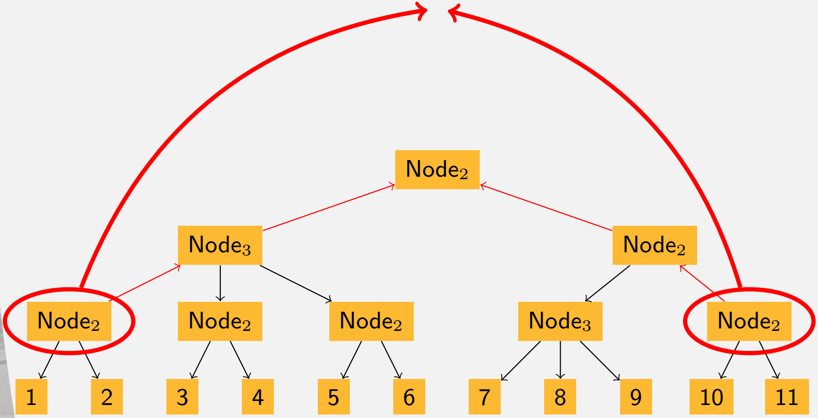
Introducing a "finger" – pointer reversal



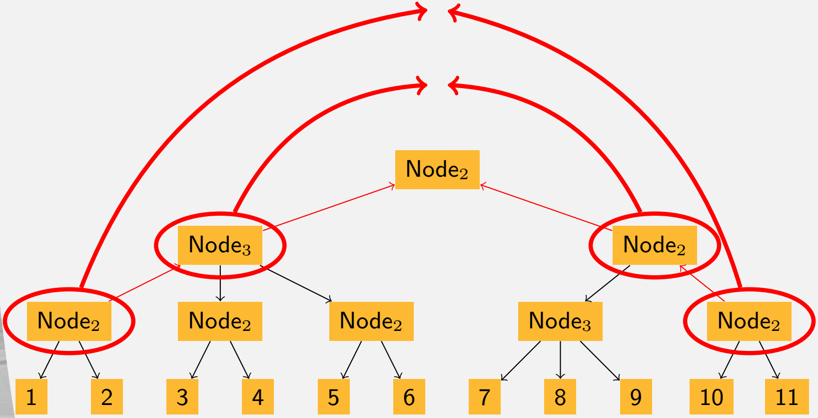
Introducing a “finger” – pointer reversal



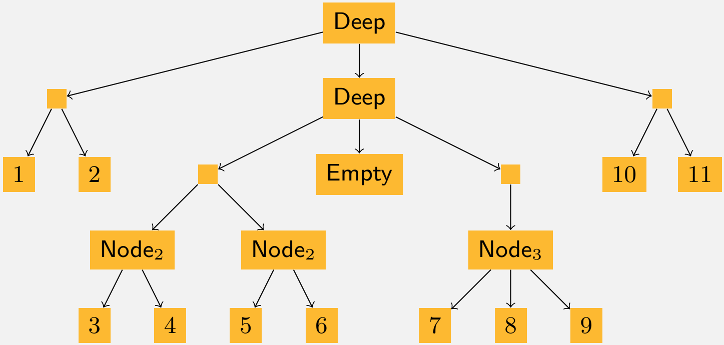
Introducing a “finger” – pointer reversal



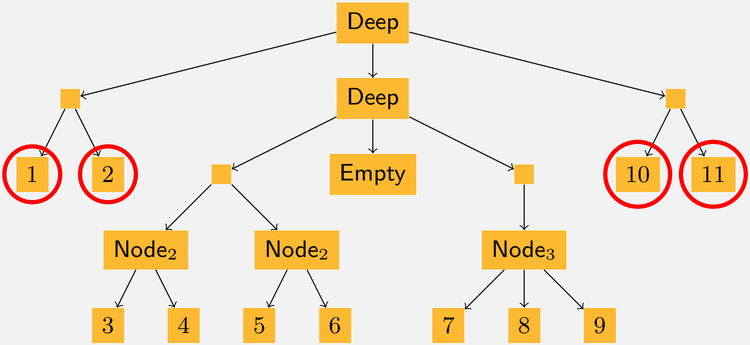
Introducing a “finger” – pointer reversal



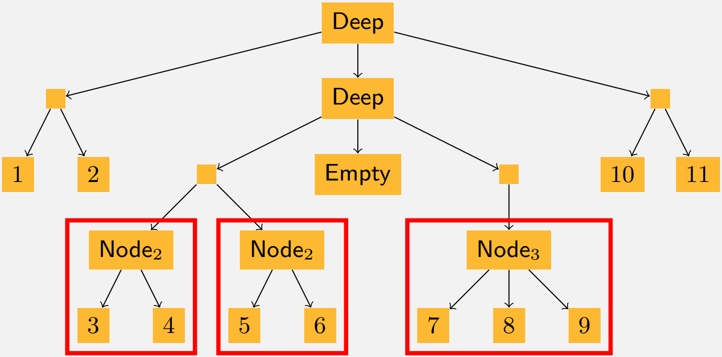
A finger tree



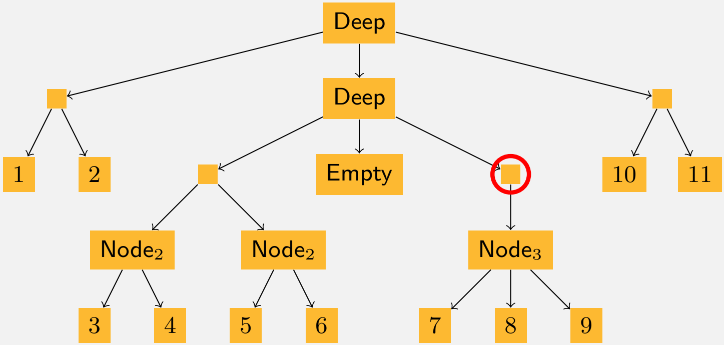
A finger tree



A finger tree

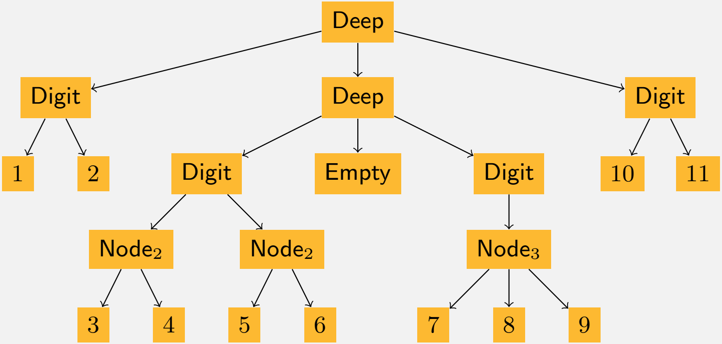


A finger tree



[Faculty of **Science**
Information and Computing Sciences]

A finger tree




```
data FingerTree a =
      Empty
    | Single a
    | Deep (Digit a) (FingerTree (Node a)) (Digit a)
type Digit a = [a] -- one up to four elements
```

Adding a single element

```
infixr 5 ◁
(◁) :: a → FingerTree a → FingerTree a
a ◁ Empty                  = Single a
a ◁ Single b               = Deep [a] Empty [b]
a ◁ Deep [b, c, d, e] m sf = Deep [a, b] (Node3 c d e ◁ m) sf
a ◁ Deep pr m sf           = Deep ([a] ++ pr) m sf
```

- We define our own operator.
- We also define its precendence and associativity.
- Note that (◁) makes use of polymorphic recursion – what is the type of the recursive call?

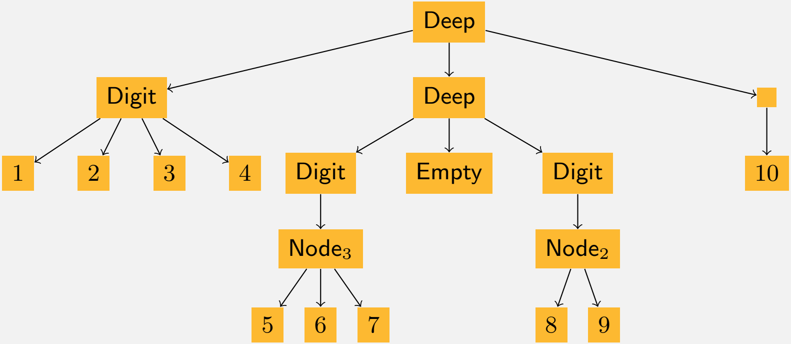
Adding a single element

```
infixr 5 ◁
(◁) :: a → FingerTree a → FingerTree a
a ◁ Empty                 = Single a
a ◁ Single b              = Deep [a] Empty [b]
a ◁ Deep [b, c, d, e] m sf = Deep [a, b] (Node3 c d e ◁ m) sf
a ◁ Deep pr m sf          = Deep ([a] ++ pr) m sf
```

- We define our own operator.
- We also define its precendence and associativity.
- Note that (◁) makes use of polymorphic recursion – what is the type of the recursive call?
- Type inference is not supported for polymorphically recursive functions.

Example: inserting an element

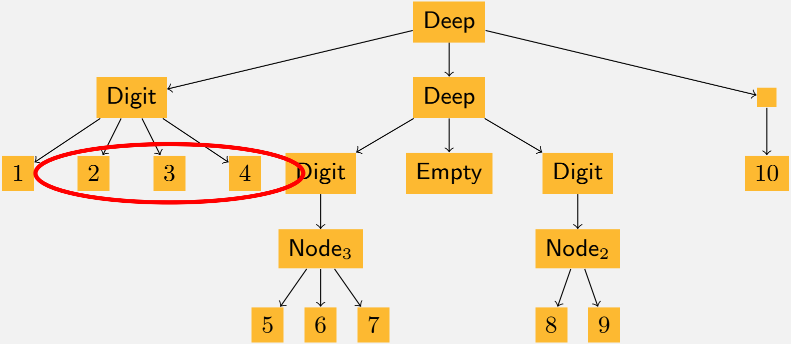
What happens when we insert 0 into the following tree?



Universiteit Utrecht
[Faculty of **Science**
Information and Computing Sciences]

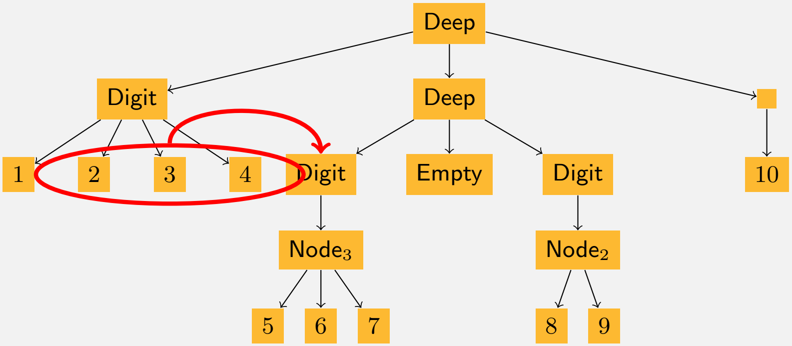
Example: inserting an element

What happens when we insert 0 into the following tree?



Example: inserting an element

What happens when we insert 0 into the following tree?



Splitting off the first element

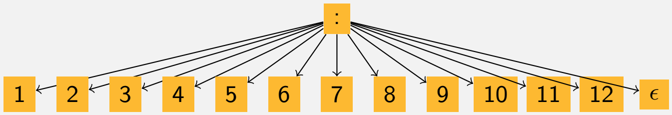
$$\textbf{data}\ \text{View}_L\ s\ a = \text{Nil}_L \mid \text{Cons}_L\ a\ (s\ a)$$
$$\text{view}_L :: \text{FingerTree}\ a \to \text{View}_L\ \text{FingerTree}\ a$$

Using these definitions, it is easy to deconstruct a finger tree:

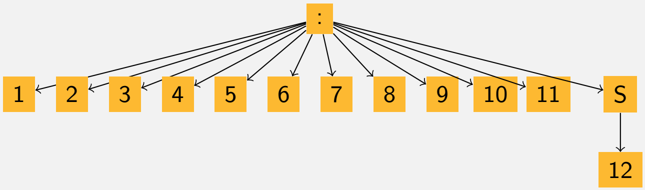
$$\text{isEmpty} :: \text{FingerTree}\ a \to \text{Bool}$$
$$\text{isEmpty}\ x = \textbf{case}\ \text{view}_L\ x\ \textbf{of}\ \text{Nil}_L \to \text{True}$$
$$\qquad\qquad \text{Cons}_L\ \_\ \_ \to \text{False}$$
$$\text{head}_L :: \text{FingerTree}\ a \to a$$
$$\text{head}_L\ x = \textbf{case}\ \text{view}_L\ x\ \textbf{of}\ \text{Cons}_L\ a\ \_ \to a$$
$$\text{tail}_L :: \text{FingerTree}\ a \to \text{FingerTree}\ a$$
$$\text{tail}_L\ x = \textbf{case}\ \text{view}_L\ x\ \textbf{of}\ \text{Cons}_L\ \_\ y \to y$$

All these operations (and also $(\triangleleft)$) take $O\ (1)$ time.

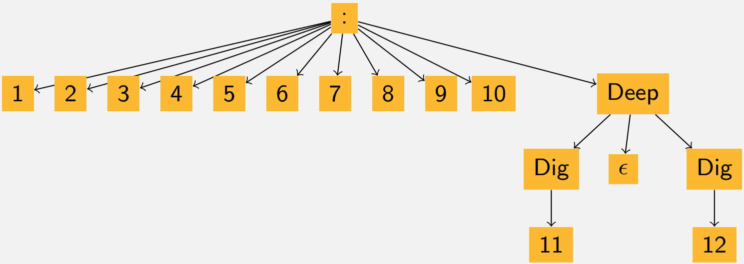
Animation of insertion into Finger Trees



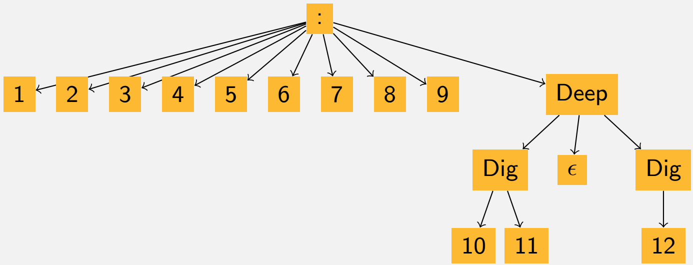
Animation of insertion into Finger Trees



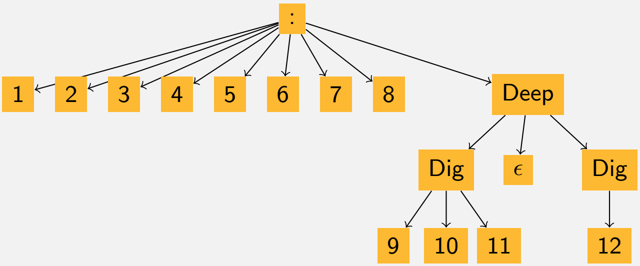
Animation of insertion into Finger Trees



Animation of insertion into Finger Trees

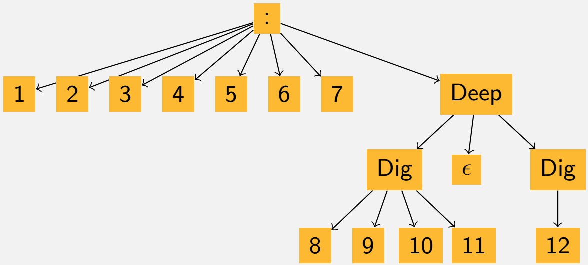


Animation of insertion into Finger Trees

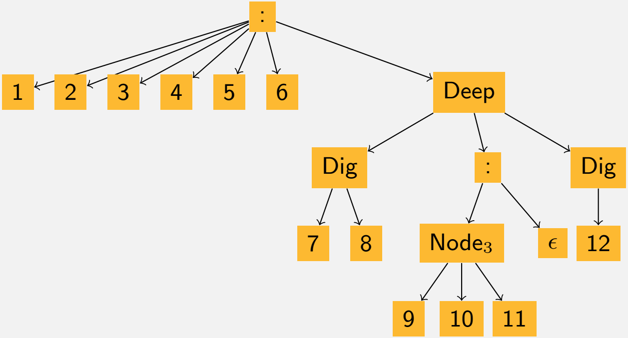


Universiteit Utrecht
[Faculty of **Science**
Information and Computing Sciences]

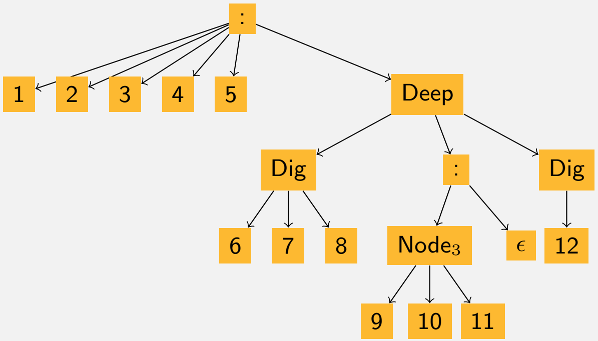
Animation of insertion into Finger Trees



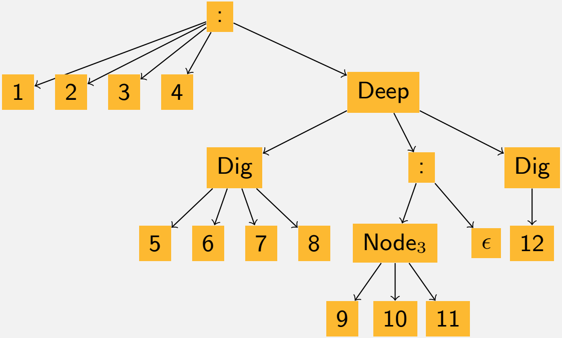
Animation of insertion into Finger Trees



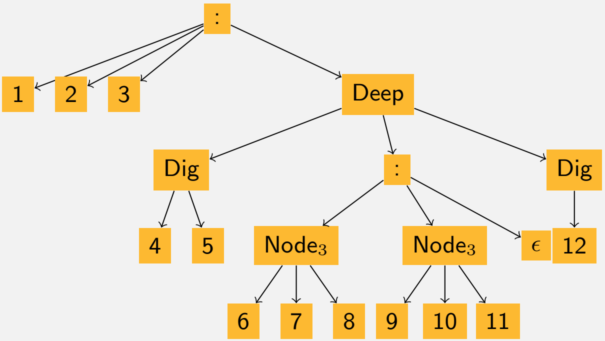
Animation of insertion into Finger Trees



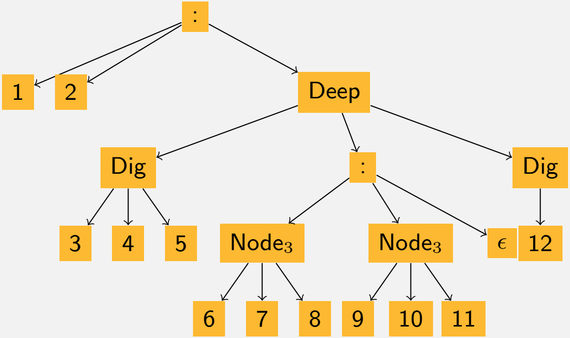
Animation of insertion into Finger Trees



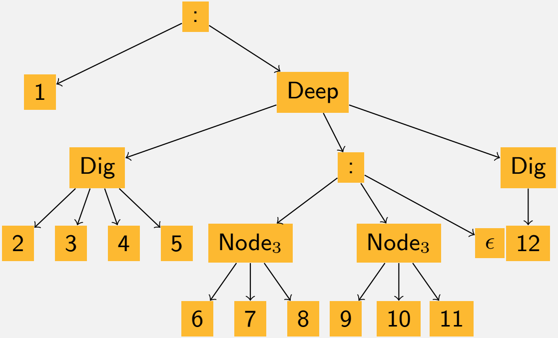
Animation of insertion into Finger Trees



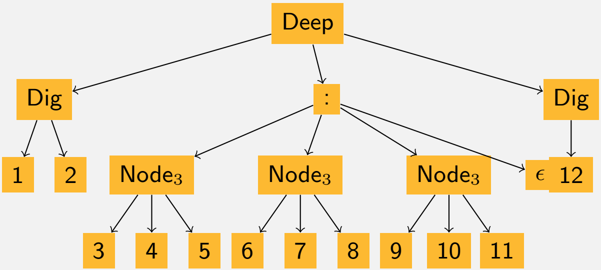
Animation of insertion into Finger Trees



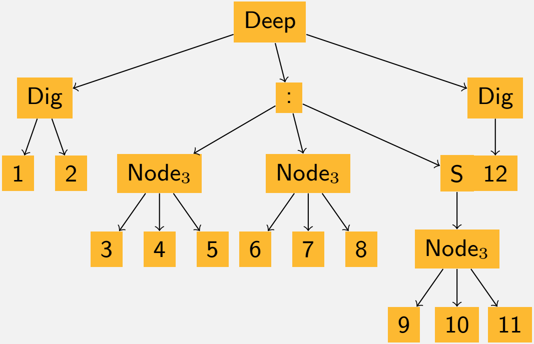
Animation of insertion into Finger Trees



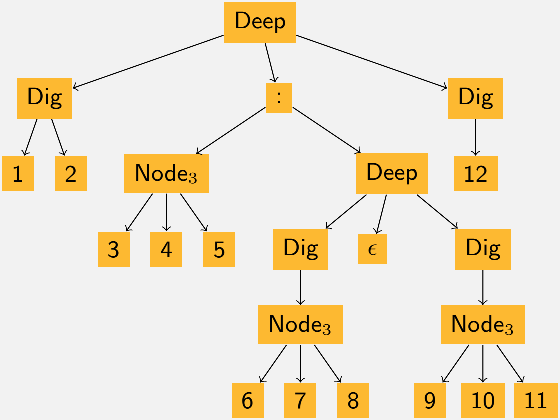
Animation of insertion into Finger Trees



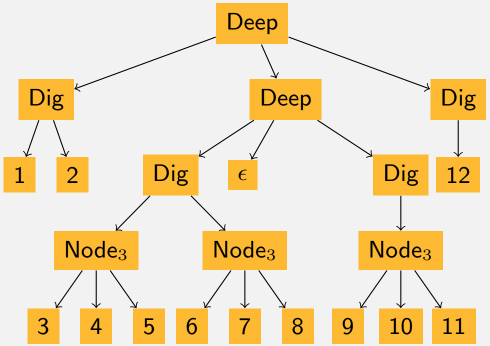
Animation of insertion into Finger Trees



Animation of insertion into Finger Trees



Animation of insertion into Finger Trees



Amortized complexity analysis

- These complexity bounds are amortized bounds:

Amortized complexity analysis

- These complexity bounds are amortized bounds:
  - A single operation may take longer than expected, but on average these operations take a constant amount of time.

Amortized complexity analysis

- These complexity bounds are amortized bounds:
  - A single operation may take longer than expected, but on average these operations take a constant amount of time.
  - Think of credit that may be distributed among operations.
  - If the timeout of an operation is T, and an operation actually finishes at time t before T, then it collects $T - t$ units of credit.
  - If a later operation takes longer than T, it may use the credit accumulated thus far to pay for the extra time.

Amortized complexity analysis

- These complexity bounds are amortized bounds:
  - A single operation may take longer than expected, but on average these operations take a constant amount of time.
  - Think of credit that may be distributed among operations.
  - If the timeout of an operation is T, and an operation actually finishes at time t before T, then it collects $T - t$ units of credit.
  - If a later operation takes longer than T, it may use the credit accumulated thus far to pay for the extra time.
- In a lazy setting with persistent data structures, we have to refine this analysis.

Complexity of adding an element

- Let us call a digit safe if it has two or three elements.
- Let us call it dangerous otherwise.
- The operation $(\triangleleft)$ only propagates to the next level on a dangerous digit, but makes it safe at the time.

$$
\begin{aligned}
a \triangleleft \text{Empty} &= \text{Single } a \\
a \triangleleft \text{Single } b &= \text{Deep } [a] \text{ Empty } [b] \\
a \triangleleft \text{Deep } [b, c, d, e] \text{ m sf} &= \text{Deep } [a, b] \; (\text{Node}_3 \; c \; d \; e \triangleleft m) \text{ sf} \\
a \triangleleft \text{Deep pr m sf} &= \text{Deep } ([a] +\!\!+ \text{pr}) \text{ m sf}
\end{aligned}
$$

Complexity of adding an element

- Let us call a digit safe if it has two or three elements.
- Let us call it dangerous otherwise.
- The operation $(\triangleleft)$ only propagates to the next level on a dangerous digit, but makes it safe at the time.

$$
\begin{aligned}
a \triangleleft \text{Empty} &= \text{Single } a \\
a \triangleleft \text{Single } b &= \text{Deep } [a] \text{ Empty } [b] \\
a \triangleleft \text{Deep } [b, c, d, e] \text{ m sf} &= \text{Deep } [a, b] \ (\text{Node}_3 \ c \ d \ e \triangleleft m) \text{ sf} \\
a \triangleleft \text{Deep pr m sf} &= \text{Deep } ([a] +\!\!+ \text{pr}) \text{ m sf}
\end{aligned}
$$

- At most every second operation propagates to next level.
- Gives us a (ephemeral) amortized bound of 2 steps per call.

Complexity of adding an element

- To make the analysis work in a persistent setting, we need laziness.
- Laziness ensures that expensive operations are delayed, and can only be forced by performing a sufficient number of further operations to pay for the cost.

Many more operations on finger trees

Data.Sequence extend finger trees further and define many more operations – an excerpt:

```
data Seq a    -- abstract, essentially FingerTree a
(⋈)     :: Seq a → Seq a → Seq a              -- O (log (min (m, n)))
length  :: Seq a → Int                         -- O (1)
index   :: Seq a → Int → a                     -- O (log n)
update  :: Int → a → Seq a → Seq a             -- O (log n)
splitAt :: Int → Seq a → (Seq a, Seq a)        -- O (log n)
reverse :: Seq a → Seq a                       -- O (n)
```

Data.FingerTree (and corresponding paper by Hinze & Paterson) also describe how to implement other data structures using finger trees.