Universiteit Utrecht

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

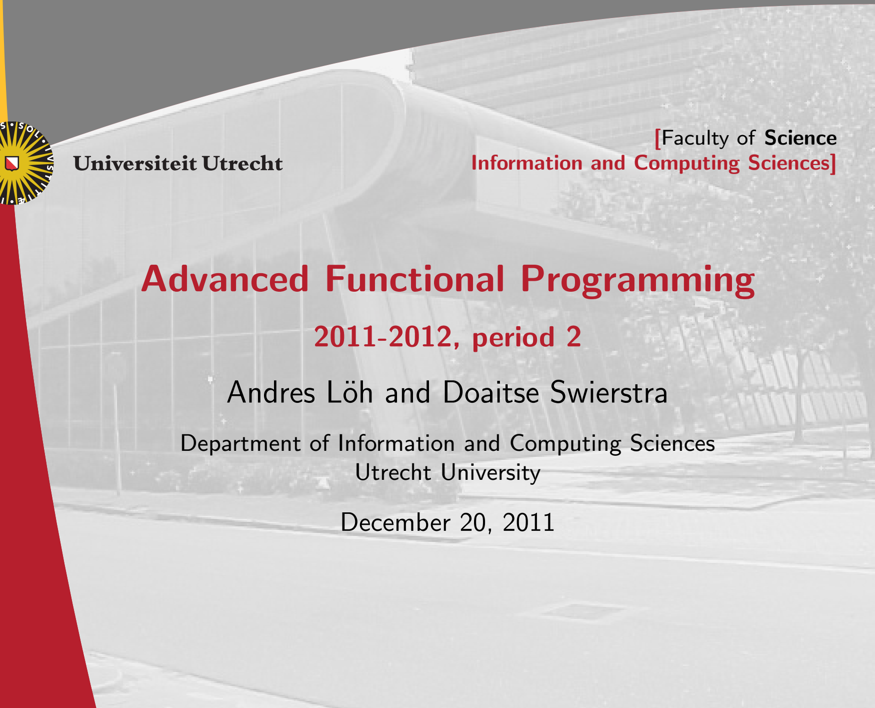
Advanced Functional Programming

2011-2012, period 2

Andres Löh and Doaitse Swierstra

Department of Information and Computing Sciences
Utrecht University

December 20, 2011



11. Trees and complexity

11.1 Sets and Finite Maps

Finite maps

- A finite map is a function with a finite domain (type of keys).
- Useful for a wide variety of applications (tables, environments, arrays (!)).
- Inefficient representation: **type** Map a b = [(a, b)].
- Can be implemented efficiently using balanced search trees:
    - Available in Data.Map and Data.IntMap for Int as key type.
    - Requires the keys to be ordered.
    - Keys are stored ordered in the tree, so that efficient lookup is possible.
    - Inserting and removing elements can trigger rotations to rebalance the tree.

Sets

- Sets are a special case of finite maps:
  type Set a = Map a ().
- A specialized set implementation is available in Data.Set and Data.IntSet, but the idea is the same as for finite maps.

Finite map interface

This is an excerpt from the functions available in Data.Map:

```
data Map k a   -- abstract
insert  :: (Ord k) ⇒ k → a → Map k a → Map k a   -- O (log n)
lookup  :: (Ord k) ⇒ k → Map k a → Maybe a       -- O (log n)
delete  :: (Ord k) ⇒ k → Map k a → Map k a       -- O (log n)
update  :: (Ord k) ⇒ (a → Maybe a) →
                     k → Map k a → Map k a       -- O (log n)
union   :: (Ord k) ⇒ Map k a → Map k a → Map k a-- O (m + n)
member  :: (Ord k) ⇒ k → Map k a → Bool          -- O (log n)
size    :: Map k a → Int                         -- O (1)
map     :: (a → b) → Map k a → Map k b           -- O (n)
```

The interface for Set is very similar.

11.2 Finger trees

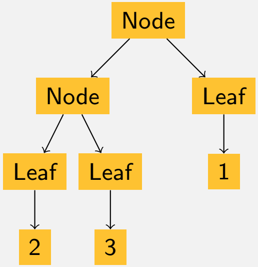
Finger trees

- 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:

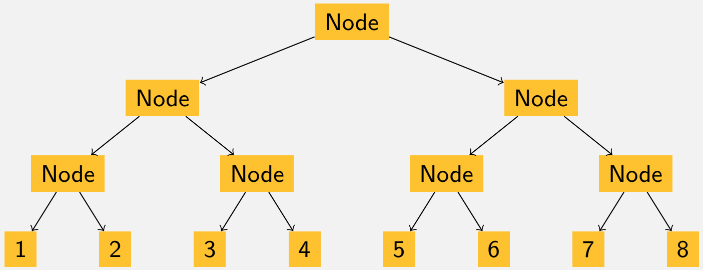


Balanced trees

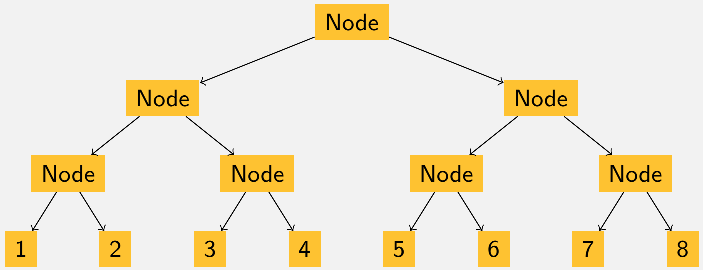
Idea

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

Example: a balanced complete tree

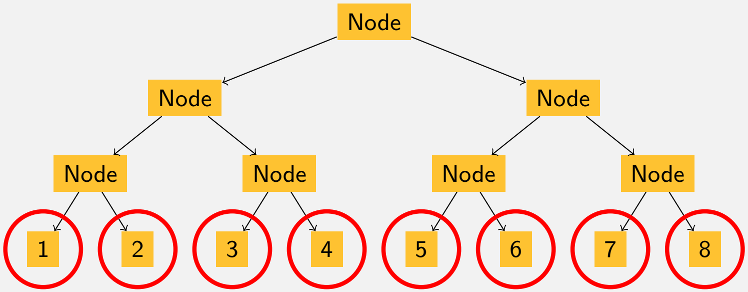


Example: a balanced complete tree



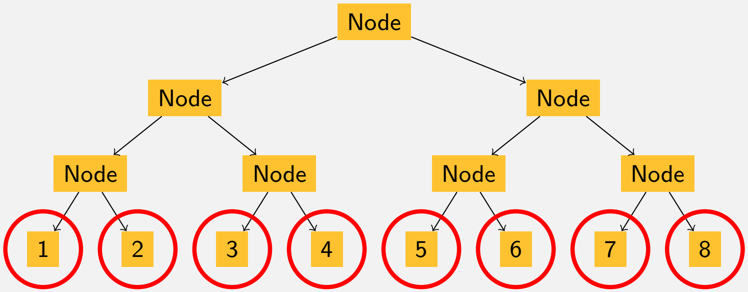
Wat are the leaves?

Example: a balanced complete tree



Wat are the leaves?

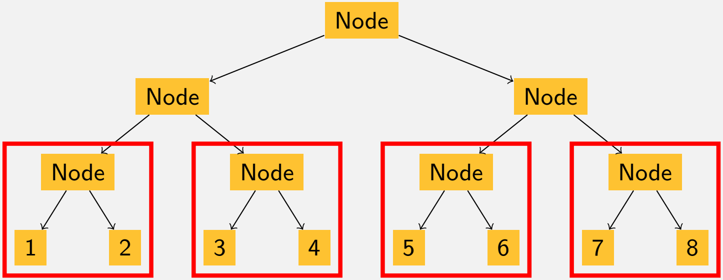
Example: a balanced complete tree



Wat are the leaves?

Can we define trees that have other trees as leaves?

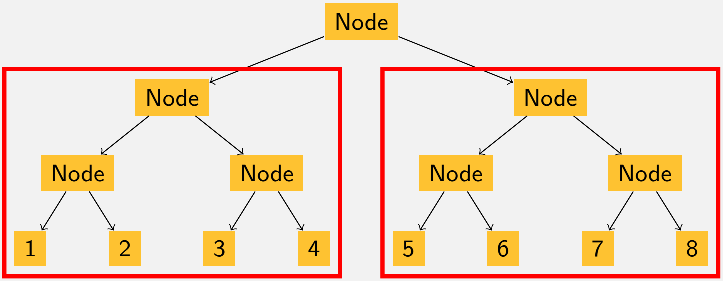
Example: a balanced complete tree



Wat 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



Wat 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

```
type Tree_0 a = a
type Tree_1 a = Node a                        = Tree_0 (Node a)
type Tree_2 a = Node (Node a)                 = Tree_1 (Node a)
type Tree_3 a = Node (Node (Node a))          = Tree_2 (Node a)
...
data Node a = Node a a   -- a node is a pair!
```

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:

```
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?

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.

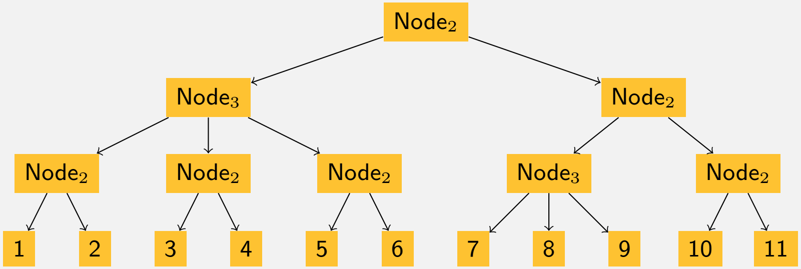
Towards 2-3-trees

- 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.

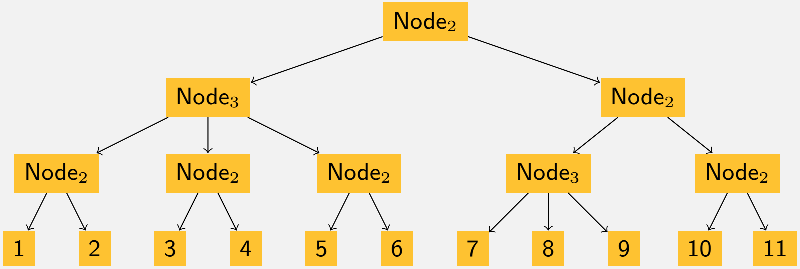
2-3-trees

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

A 2-3-tree


Node_2
Node_3
Node_2
Node_2
Node_2
Node_2
Node_3
Node_2
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
```

Universiteit Utrecht

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

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

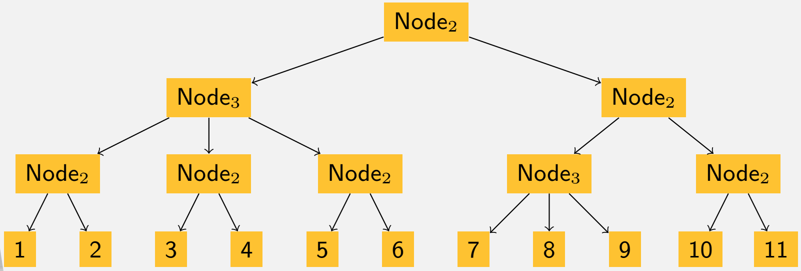
depth (n)	min elements (2^n)	max elements (3^n)
0	1	1
1	2	3
2	4	9
3	8	27
...		

Every number of elements can be represented.

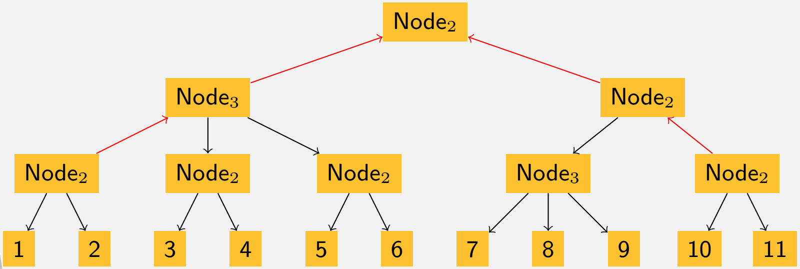
Finger trees

- 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.

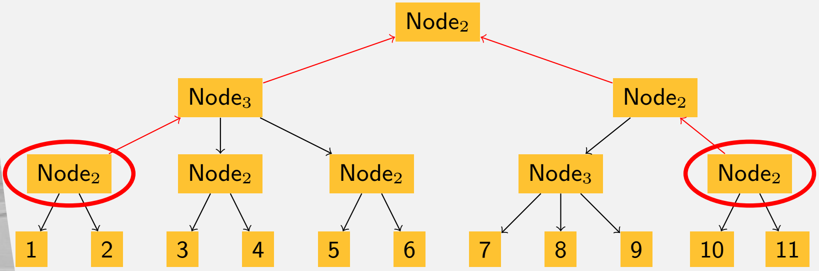
Introducing a “finger” – pointer reversal



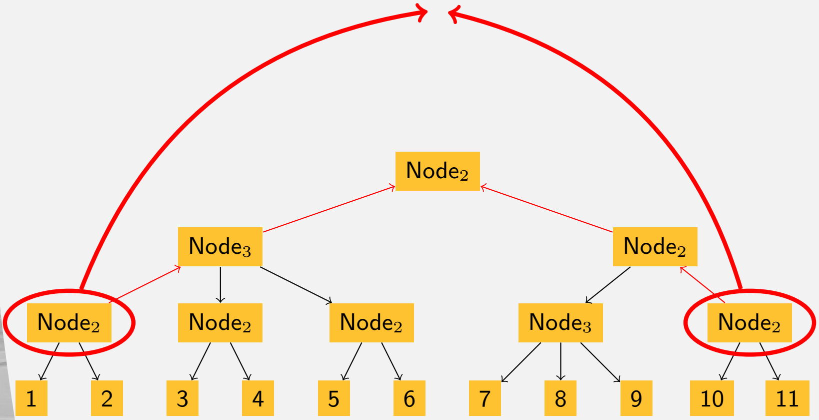
Introducing a “finger” – pointer reversal



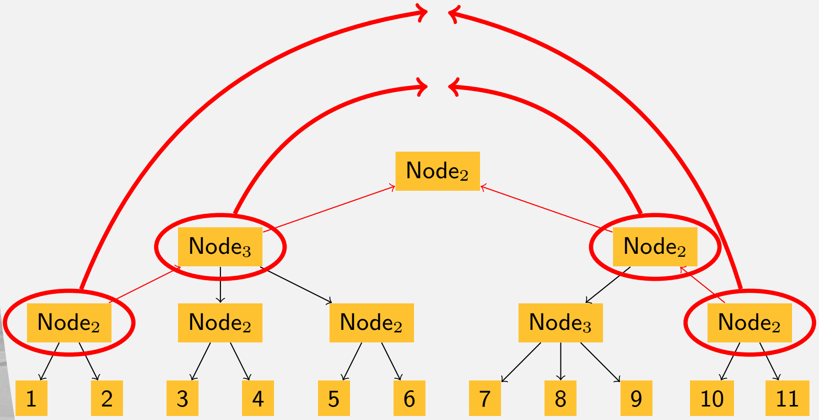
Introducing a “finger” – pointer reversal



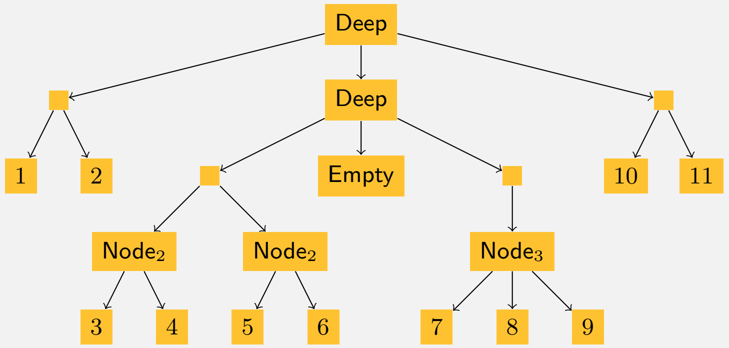
Introducing a “finger” – pointer reversal



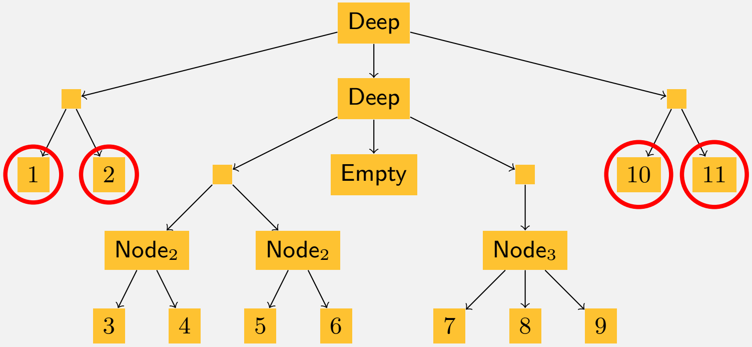
Introducing a “finger” – pointer reversal



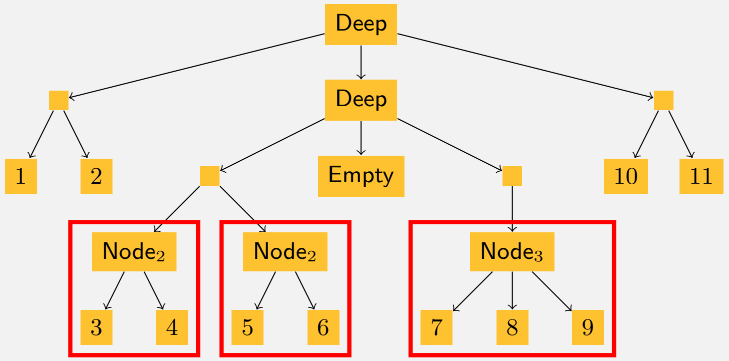
A finger tree



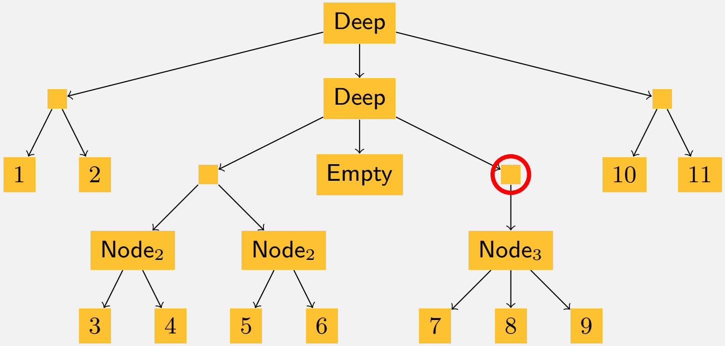
A finger tree



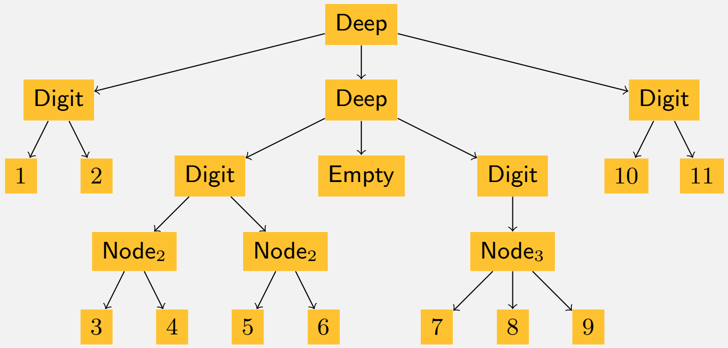
A finger tree



A finger tree



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?

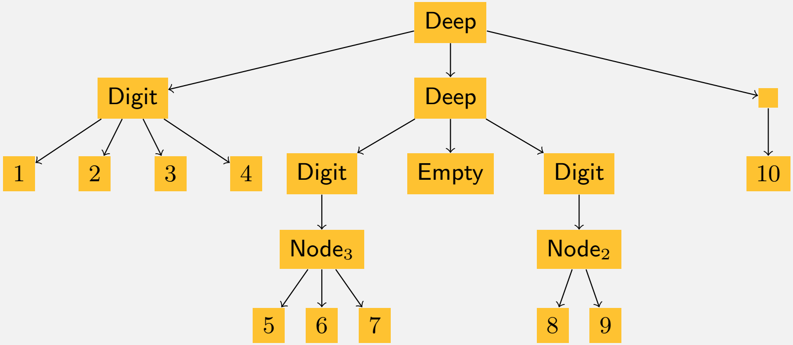
Adding a single element

```
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.

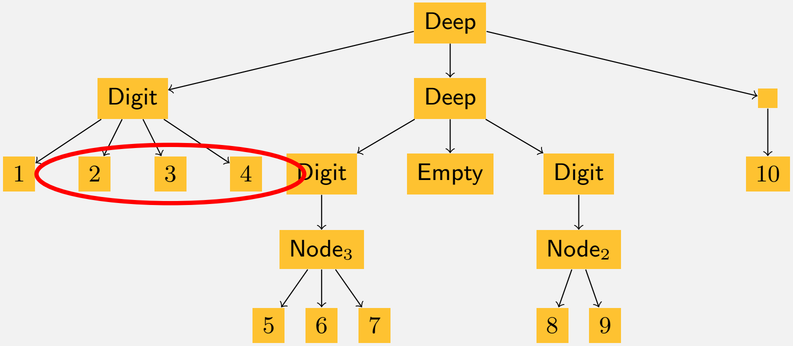
Example: inserting an element

What happens when we insert 0 into the following tree?



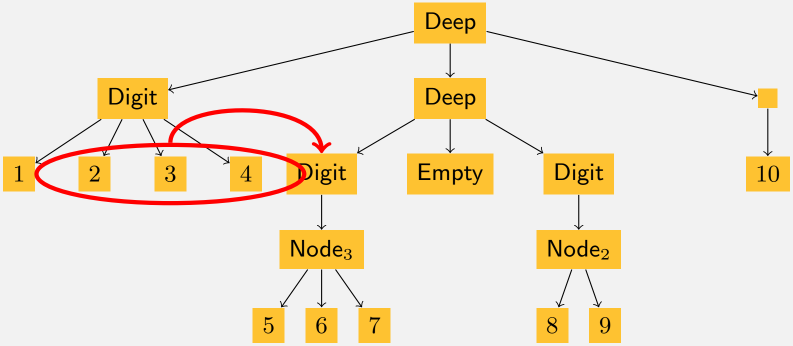
Example: inserting an element

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

```
data ViewL s a = NilL | ConsL a (s a)
viewL :: FingerTree a → ViewL FingerTree a
```

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

```
isEmpty :: FingerTree a → Bool
isEmpty x = case viewL x of NilL        → True
                            ConsL _ _   → False
headL :: FingerTree a → a
headL x = case viewL x of ConsL a _ → a
tailL :: FingerTree a → FingerTree a
tailL x = case viewL x of ConsL _ y → y
```

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

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.

```
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
```

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.

```
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
```

- 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

Paper (and 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 (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)
```

The paper also describes how to implement other data structures using finger trees.

11.3 Other useful data structures

Other useful data structures

A quick (incomplete) look through Data.∗:

Data.Complex	complex numbers
Data.Ratio	rational numbers (underestimated!)
Data.Tree	trees (simplistic)
Data.Graph	graphs (straight-forward)
Data.Graph.Inductive	functional graph library
Data.HashTable	ephemeral hash tables (IO)
Data.IORef	mutable references (IO)
Data.Time	time and calendar operations
Data.Unique	unique symbol generator (IO)

Conclusions

- Know the major data structures and their strengths and weaknesses.
- Look around once in a while for new libraries (HackageDB, Haskell mailing list, Haskell Communities and Activities Report).
- Choose a data structure that is suited for the job.
- If unsure which data structure to use, try to define your own interface and abstract from that interface, so that it would be easy to switch later.