SYDDANSK UNIVERSITET
UNIVERSITY OF SOUTHERN DENMARK

Lecture 3: Algebraic Data Types

Søren Haagerup

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

November 7, 2016

Algebraic Data Types

- Algebraic Data Types, are types which are constructed using *constructor* functions, and destructed using *pattern matching*
- Which Algebraic Data Types have we seen so far?
- *Bool*, *Maybe* a, (a, b),$[a]$
- Which non-algebraic types have we seen so far?
- *Int*, *Integer* etc., functions $a \to b$

The **data** keyword

If we wanted to define our own *Bool* data type, we would write

$$\textbf{data}\ \mathit{Bool} = \mathit{False} \mid \mathit{True}$$

This gives rise to constructor functions

- $\mathit{False} :: \mathit{Bool}$
- $\mathit{True} :: \mathit{Bool}$

corresponding patterns *False*, *True*.

Enumeration types in general

$$\textbf{data}\ \mathit{Bool} = \mathit{False} \mid \mathit{True}$$
$$\textbf{data}\ \mathit{Ordering} = \mathit{LT} \mid \mathit{EQ} \mid \mathit{GT}$$
$$\textbf{data}\ \mathit{Season} = \mathit{Spring} \mid \mathit{Summer} \mid \mathit{Fall} \mid \mathit{Winter}$$
$$\textbf{data}\ \mathit{MessageType} = \mathit{Success} \mid \mathit{Info} \mid \mathit{Warning} \mid \mathit{Danger}$$
$$\textbf{data}\ \mathit{ThreadState} = \mathit{Ready} \mid \mathit{Running} \mid \mathit{Terminated} \mid \mathit{Waiting}$$

If we have fixed number of elements in the type, enumeration types are useful.

$$\begin{aligned}
&\mathit{isHot} :: \mathit{Season} \to \mathit{Bool} \\
&\mathit{isHot}\ \mathit{Spring} \quad = \mathit{True} \\
&\mathit{isHot}\ \mathit{Summer} = \mathit{True} \\
&\mathit{isHot}\ _ \qquad\quad = \mathit{False}
\end{aligned}$$

Product types

- *Product types* correspond to structs/records in other languages. E.g

$$
\begin{aligned}
&\textbf{type } \mathit{Age} \quad = \mathit{Int}\\
&\textbf{type } \mathit{Name} = \mathit{String}\\
&\textbf{data } \mathit{Person} = \mathit{Person}\ \mathit{Age}\ \mathit{Name}\\
&\\
&\mathit{isOld} :: \mathit{Person} \to \mathit{Bool}\\
&\mathit{isOld}\ (\mathit{Person}\ a\ \_) \mid a \geqslant 30 = \mathit{True}\\
&\mathit{isOld}\ \_ \qquad\qquad\qquad\qquad\quad = \mathit{False}
\end{aligned}
$$

- What type does the data constructor *Person* have?
- Why not just define a type synonym
  type $\mathit{Person} = (\mathit{Age}, \mathit{Name})$?

Difference between tuples and general product types

- Let's say that we somewhere in a program have defined

  $$\begin{aligned}&\textbf{type } \mathit{Age} \quad = \mathit{Int}\\&\textbf{type } \mathit{Name} = \mathit{String}\\&\textbf{type } \mathit{Person} = (\mathit{Age}, \mathit{Name})\\&\mathit{isOld} :: \mathit{Person} \to \mathit{Bool}\end{aligned}$$

- And somewhere else, we have defined

  $$\begin{aligned}&\textbf{type } \mathit{ExitCode} = \mathit{Int}\\&\mathit{execCommand} :: \mathit{String} \to (\mathit{ExitCode}, \mathit{String})\end{aligned}$$

- Then we would be able to write

  $$\mathit{isOld}\ (\mathit{execCommand}\ \texttt{"ls"})$$

  which does not make much sense.

Product types

- When having written

  $$\textbf{data}\ Person = Person\ Age\ Name$$

  it is also reasonable to write functions

  $$\begin{aligned} &age :: Person \to Age \\ &age\ (Person\ a\ \_) = a \\ &name :: Person \to Name \\ &name\ (Person\ \_\ n) = n \end{aligned}$$

  Such functions are called “projection functions” or “accessor functions”
- They can be created by Haskell automatically by writing

  $$\begin{aligned} &\textbf{data}\ Person = Person\ \{ \\ &\quad age :: Age \\ &\quad name :: Name \\ &\} \end{aligned}$$

Sum types

- We can also combine the concepts of an “Enumerated types” and “Product type”. We can define a type

  $$\textbf{data}\ \mathit{Number} = \mathit{I}\ \mathit{Integer} \mid \mathit{D}\ \mathit{Double}$$

  This makes it possible to store either an *Integer* or a *Double* in a Number, much like the JavaScript number type.
- Which constructors does it have, and what are the types of the constructors?

Recursive types

- Let's define a type

$$\textbf{data}\ Nat = Z \mid S\ Nat$$

- What does this type represent?
- What are the type signatures for the constructors?
- How can we implement $add :: Nat \to Nat \to Nat$?
- Come up with an alternative definition of add.
- How can we implement $toInt :: Nat \to Int$?
- Come up with an alternative definition of $toInt$.
- Which list function does the functions above remind us of?
- How can we implement $min :: Nat \to Nat \to Nat$?
- Which list function does this remind us of?

Example: File System


```
data Entry = File {
       name :: String
   } | Folder {
       name :: String,
       entries :: [Entry]
   }
entry :: Entry
entry = Folder "Documents" [
   File "File1",
   File "File2",
   Folder "Pictures" [
     File "Picture 1",
     File "Picture 2"
   ]
   ]
```

Bigger example: JSON data


```
data Number = I Integer | D Double
data JsValue  = JsArray [JsValue]
              | JsBoolean Bool
              | JsNull
              | JsNumber Number
              | JsObject [(String, JsValue)]
              | JsString String

instance Show Number where
  show (I i) = show i
  show (D d) = show d

instance Show JsValue where
  show (JsArray xs)   = "[" ++ (concat (intersperse "," (map show xs))) ++ "]"
  show (JsBoolean b)  = if b then "true" else "false"
  show (JsNull)       = "null"
  show (JsNumber n) = show n
  show (JsObject xs)  = "{" ++ concat (intersperse ","
    ["\"" ++ k ++ "\":" ++ show v | (k,v) <- xs]) ++ "}"
  show (JsString s)   = show s
```

Polymorphic types

- Is the following a product or a sum type?

$$\textbf{data}\ Pair\ a\ b = Pair\ a\ b$$

- Another example of a polymorphic type

$$\begin{aligned}
&\textbf{data}\ Maybe\ a &&= Nothing \mid Just\ a\\
&safeHead :: [a] \to Maybe\ a\\
&safeHead\ [\,] &&= Nothing\\
&safeHead\ (x : xs) &&= Just\ x
\end{aligned}$$

- Another example of a polymorphic type

$$\begin{aligned}
&\textbf{data}\ Either\ a\ b &&= Left\ a \mid Right\ b\\
&\textbf{type}\ Error\ a &&= Either\ String\ a\\
&safeHead :: [a] \to Error\ a\\
&safeHead\ [\,] &&= Left\ \texttt{"Error: [] has no head!"}\\
&safeHead\ (x : xs) &&= Right\ x
\end{aligned}$$

Error handling + Record update syntax

$$
\begin{aligned}
&\textbf{data}\ \mathit{Entry} = \mathit{File}\ \{\mathit{name} :: \mathit{String}\} \\
&\qquad\qquad\qquad\quad \mid \mathit{Folder}\ \{\mathit{name} :: \mathit{String}, \mathit{entries} :: [\mathit{Entry}]\} \\
&\qquad\qquad\qquad\quad \textbf{deriving}\ \mathit{Show} \\
&\mathit{rename} :: \mathit{String} \to \mathit{Entry} \to \mathit{Entry} \\
&\mathit{rename}\ n\ e = e\ \{\mathit{name} = n\}
\end{aligned}
$$

Polymorphic, recursive types

We could define a linked list as:

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

The list type which Haskell includes has some syntactic sugar, so Haskells built-in list corresponds to a definition like

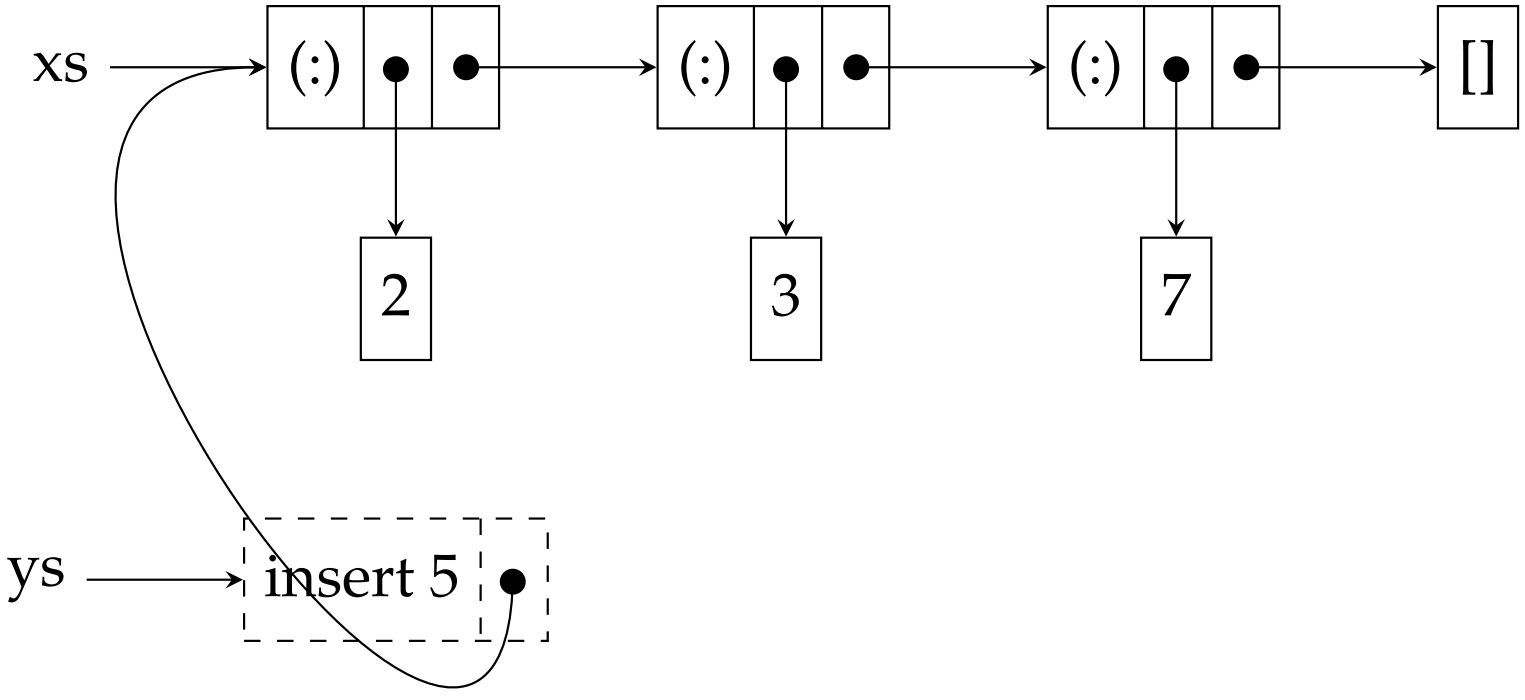
$$\textbf{data}\ [a] = [\,] \mid a : [a] \quad \text{-- (not legal Haskell code!)}$$

How to find the length of a list, defined using our own *List a* type?

$$\mathit{length} :: \mathit{List}\ a \to \mathit{Int}$$

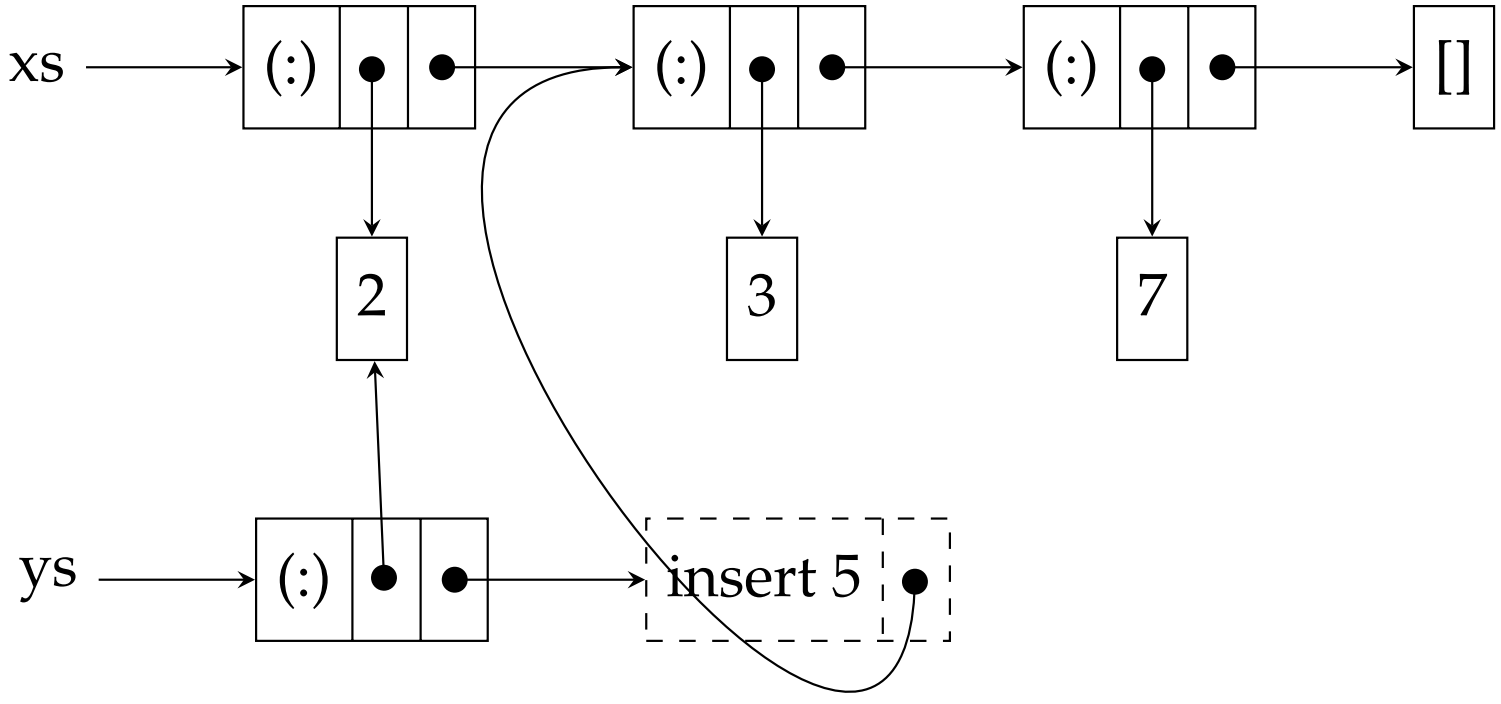
Insertion in a list

- Recall *insert x xs* which insert *x* at the correct position in a sorted list.
- We now want to investigate how the resulting list is defined:



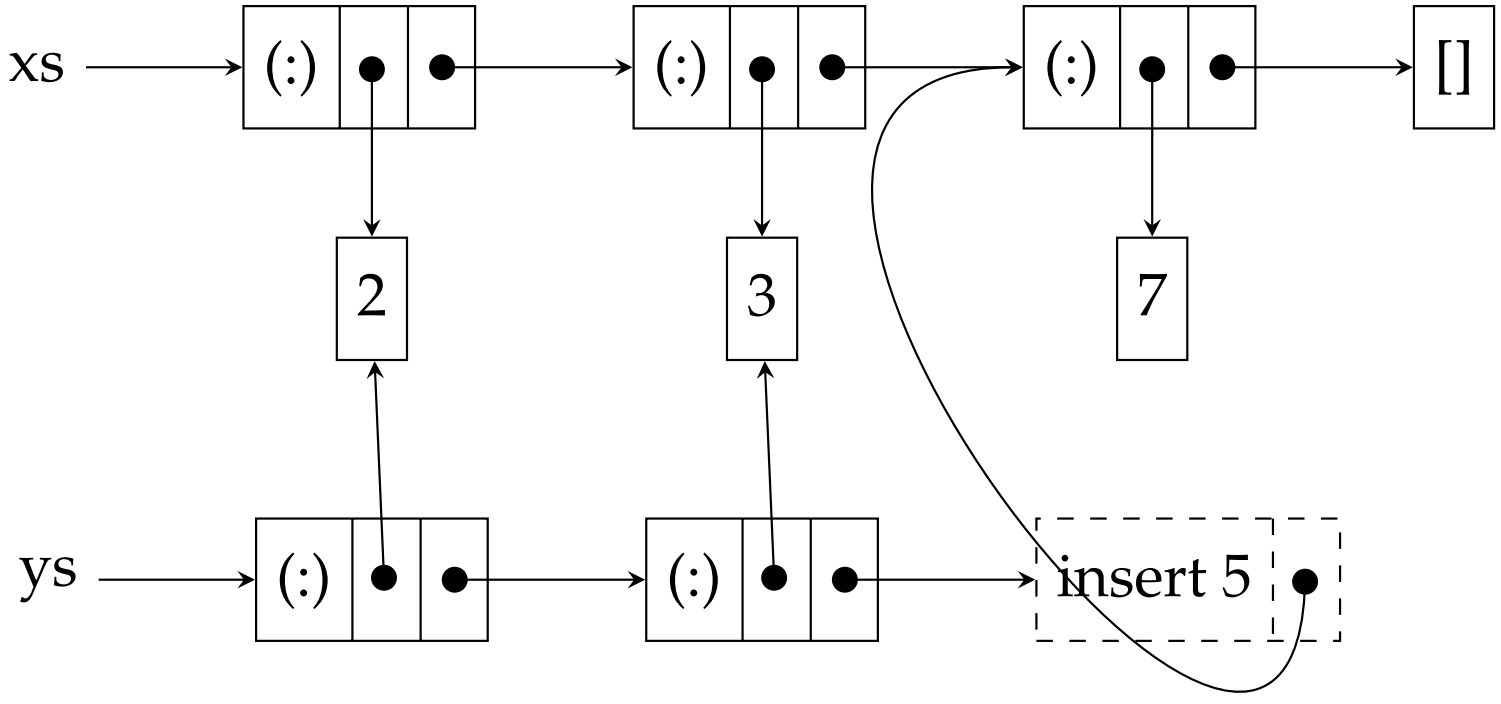
Insertion in a list

- Recall *insert x xs* which insert *x* at the correct position in a sorted list.
- We now want to investigate how the resulting list is defined:



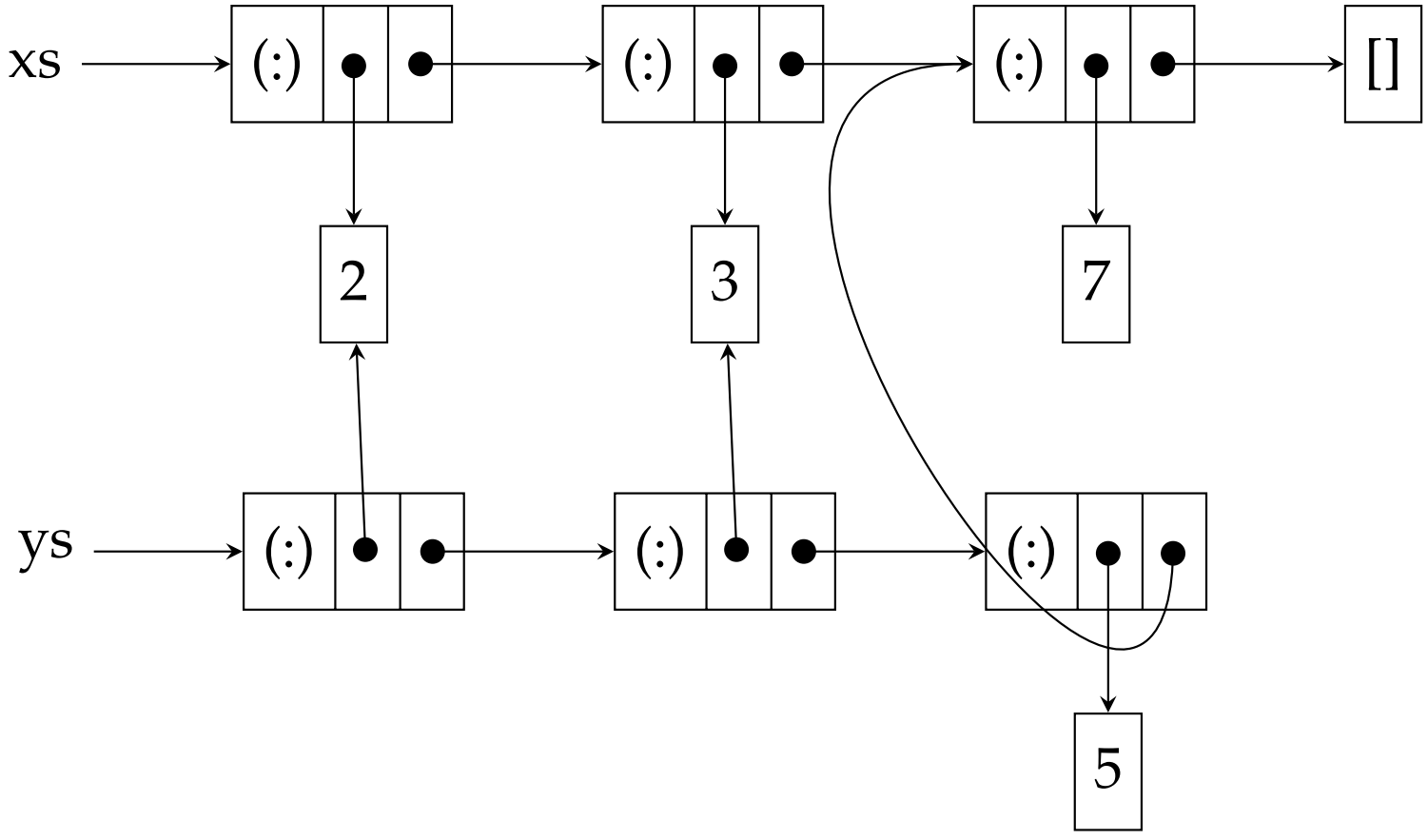
Insertion in a list

- Recall *insert x xs* which insert *x* at the correct position in a sorted list.
- We now want to investigate how the resulting list is defined:



Insertion in a list

- Recall *insert x xs* which insert *x* at the correct position in a sorted list.
- We now want to investigate how the resulting list is defined:



Immutable data structures

- never destroy data - only the garbage collector should do that!
- if we want to “change” an element, we generate a new one instead!

	Imperative	Functional
List	ArrayList	Linked List
Dictionary	HashMap	Balanced Tree

TREE STRUCTURES

A look at Data.Map

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Map}\ k\ a & & \\
\mathit{insert} & :: (\mathit{Ord}\ k) \Rightarrow k \to a \to \mathit{Map}\ k\ a \to \mathit{Map}\ k\ a & \text{-- } O(\log n) \\
\mathit{lookup} & :: (\mathit{Ord}\ k) \Rightarrow k \to \mathit{Map}\ k\ a \to \mathit{Maybe}\ a & \text{-- } O(\log n) \\
\mathit{delete} & :: (\mathit{Ord}\ k) \Rightarrow k \to \mathit{Map}\ k\ a \to \mathit{Map}\ k\ a & \text{-- } O(\log n) \\
\mathit{update} & :: (\mathit{Ord}\ k) \Rightarrow & \\
\quad (a \to \mathit{Maybe}\ a) & \to k \to \mathit{Map}\ k\ a \to \mathit{Map}\ k\ a & \text{-- } O(\log n) \\
\mathit{union} & :: (\mathit{Ord}\ k) \Rightarrow \mathit{Map}\ k\ a \to \mathit{Map}\ k\ a \to \mathit{Map}\ k\ a & \text{-- } O(m+n) \\
\mathit{member} & :: (\mathit{Ord}\ k) \Rightarrow k \to \mathit{Map}\ k\ a \to \mathit{Bool} & \text{-- } O(\log n) \\
\mathit{size} & :: \mathit{Map}\ k\ a \to \mathit{Int} & \text{-- } O(1) \\
\mathit{map} & :: (a \to b) \to \mathit{Map}\ k\ a \to \mathit{Map}\ k\ b & \text{-- } O(n)
\end{array}
$$

How to use Data.Map


```
import Data.Map (Map)
import qualified Data.Map as Map

testMap = Map.fromList (zip [1..13] ['a'..'m'])
```

```
> putStrLn (Map.showTree testMap)
8:='h'
+---4:='d'
|   +---2:='b'
|   |   +---1:='a'
|   |   +---3:='c'
|   +---6:='f'
|       +---5:='e'
|       +---7:='g'
+---12:='l'
    +---10:='j'
    |   +---9:='i'
    |   +---11:='k'
    +---13:='m'
```

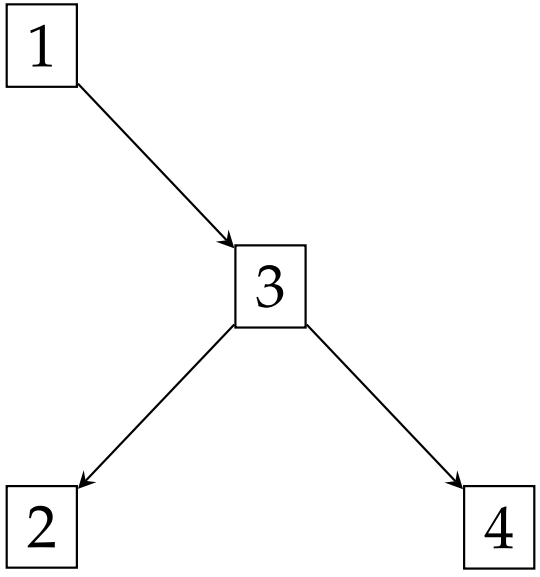
Binary Search Tree

$$\textbf{data}\ \mathit{Tree}\ a = \mathit{Nil} \mid \mathit{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)$$

Example:

$$\mathit{Node}\ 1\ \mathit{Nil}\ (\mathit{Node}\ 3\ (\mathit{Node}\ 2\ \mathit{Nil}\ \mathit{Nil})\ (\mathit{Node}\ 4\ \mathit{Nil}\ \mathit{Nil}))$$

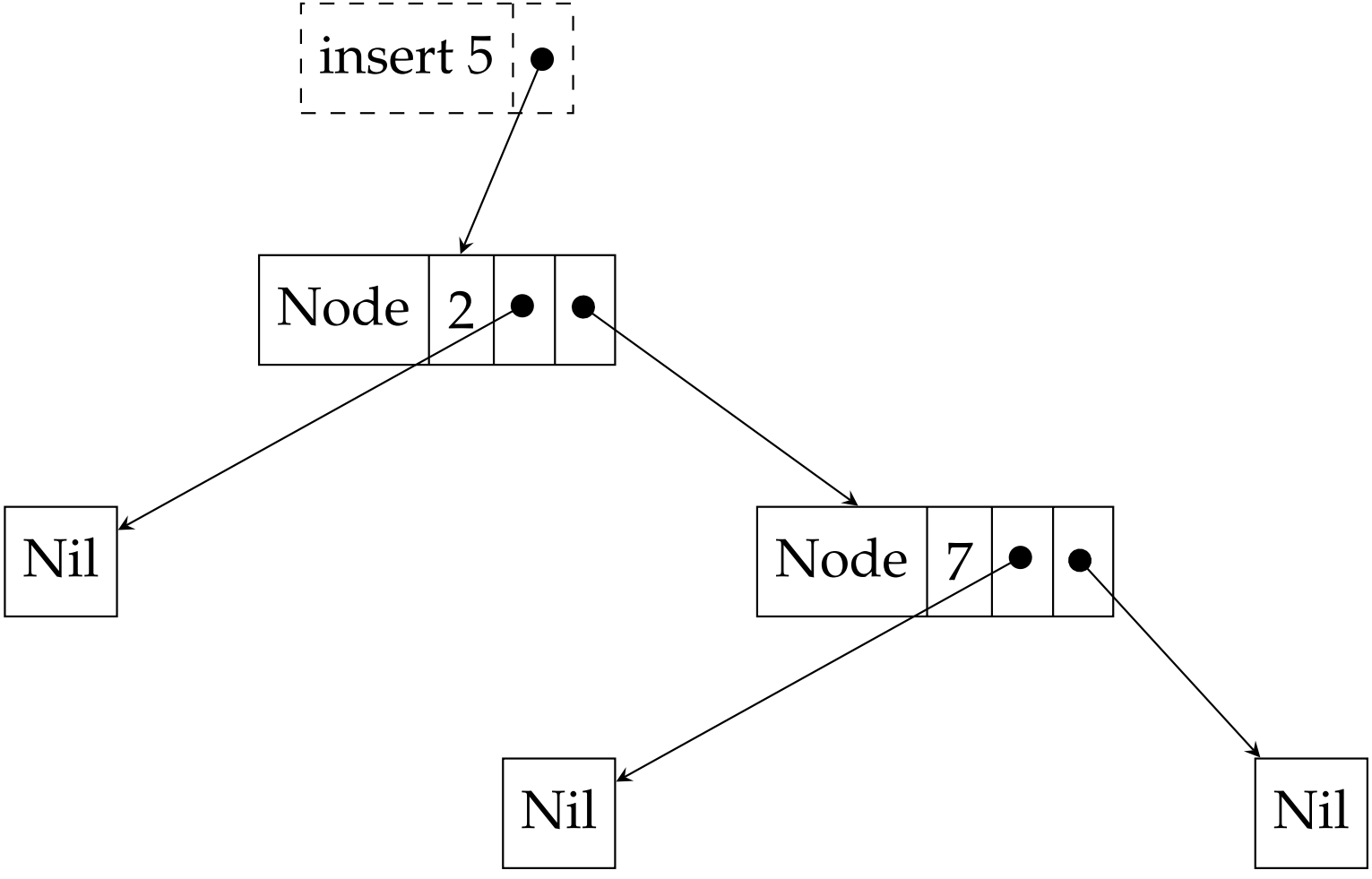
corresponds to the tree



Binary Search Tree

We are given **data** $Tree\ a = Nil \mid Node\ a\ (Tree\ a)\ (Tree\ a)$.

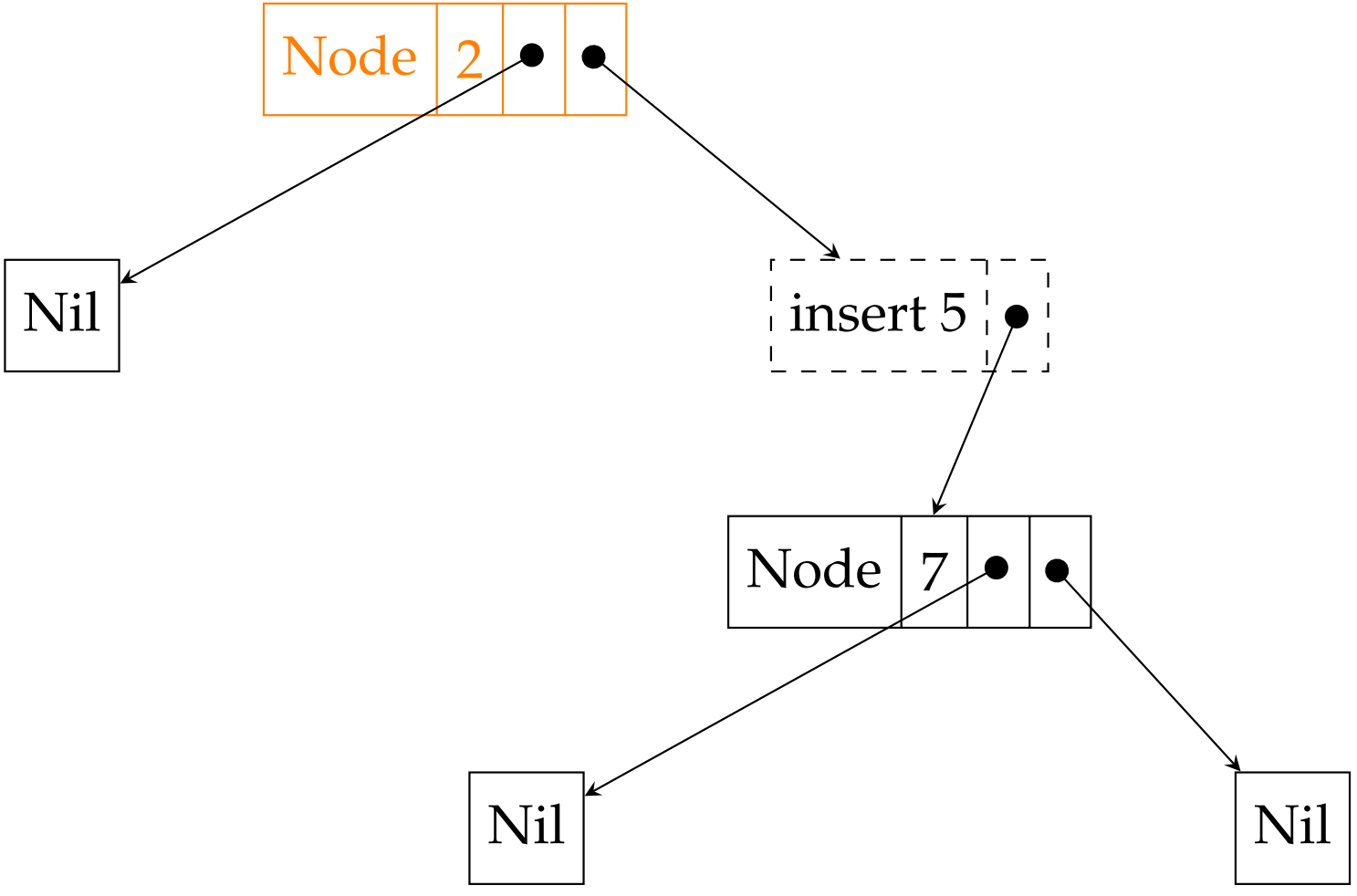
How should we define $insert :: Ord\ a \Rightarrow a \rightarrow Tree\ a \rightarrow Tree\ a$?



Binary Search Tree

We are given **data** $Tree\ a = Nil \mid Node\ a\ (Tree\ a)\ (Tree\ a)$.

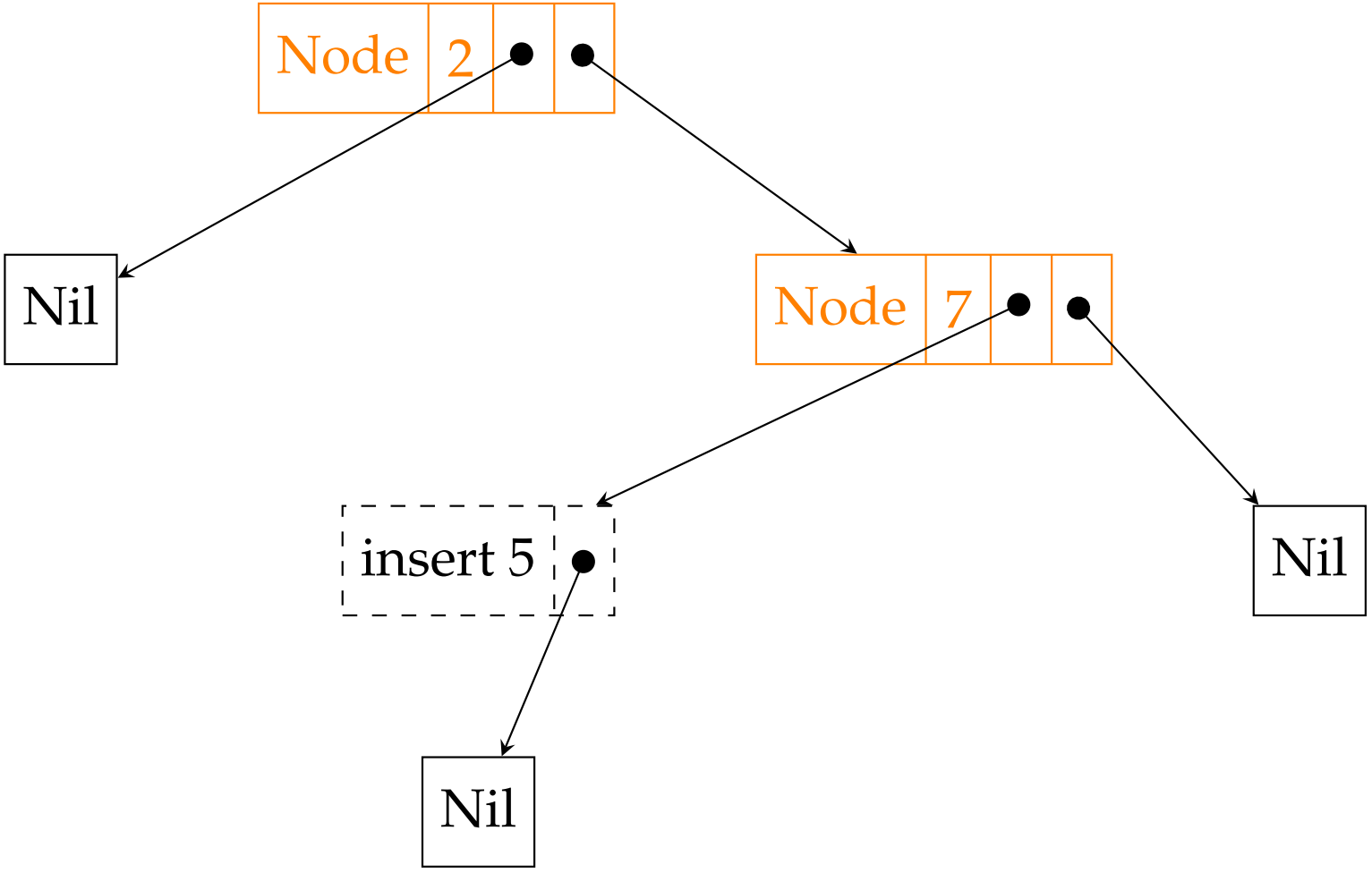
How should we define $insert :: Ord\ a \Rightarrow a \rightarrow Tree\ a$?



Binary Search Tree

We are given **data** $Tree\ a = Nil \mid Node\ a\ (Tree\ a)\ (Tree\ a)$.

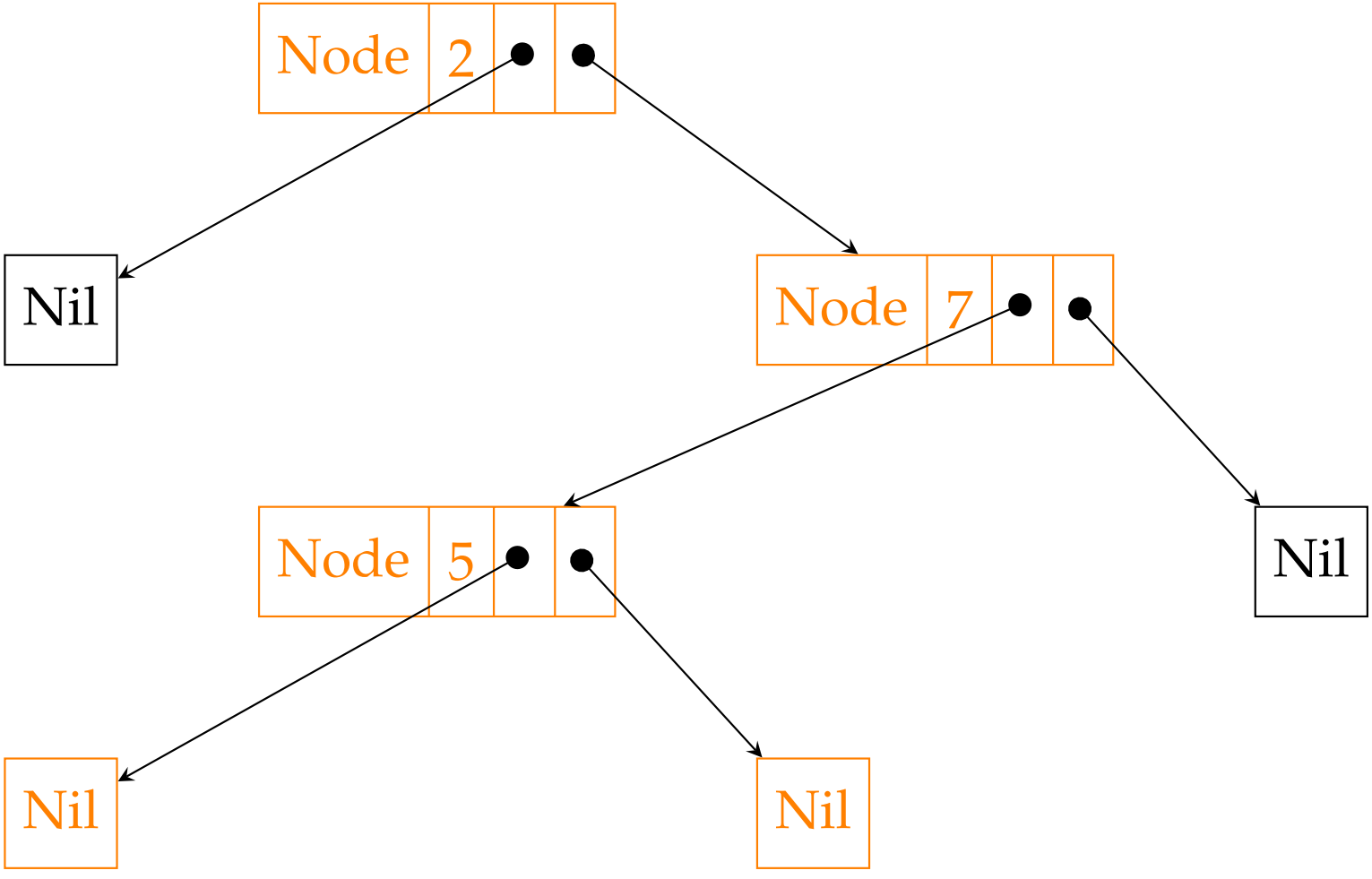
How should we define $insert :: Ord\ a \Rightarrow a \rightarrow Tree\ a$?



Binary Search Tree

We are given **data** $Tree\ a = Nil \mid Node\ a\ (Tree\ a)\ (Tree\ a)$.

How should we define $insert :: Ord\ a \Rightarrow a \rightarrow Tree\ a$?



insert - Binary search tree

$$
\begin{aligned}
&\mathit{insert} :: \mathit{Ord}\ a \Rightarrow a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a \\
&\mathit{insert}\ x\ \mathit{Nil} \qquad\qquad\quad = \mathit{Node}\ x\ \mathit{Nil}\ \mathit{Nil} \\
&\mathit{insert}\ x\ (\mathit{Node}\ y\ l\ r) = \mathbf{case}\ (\mathit{compare}\ x\ y)\ \mathbf{of} \\
&\quad \mathit{LT}\ \rightarrow \mathit{Node}\ y\ (\mathit{insert}\ x\ l)\ r \\
&\quad \mathit{EQ} \rightarrow \mathit{Node}\ x\ l\ r \\
&\quad \mathit{GT} \rightarrow \mathit{Node}\ y\ l\ (\mathit{insert}\ x\ r)
\end{aligned}
$$

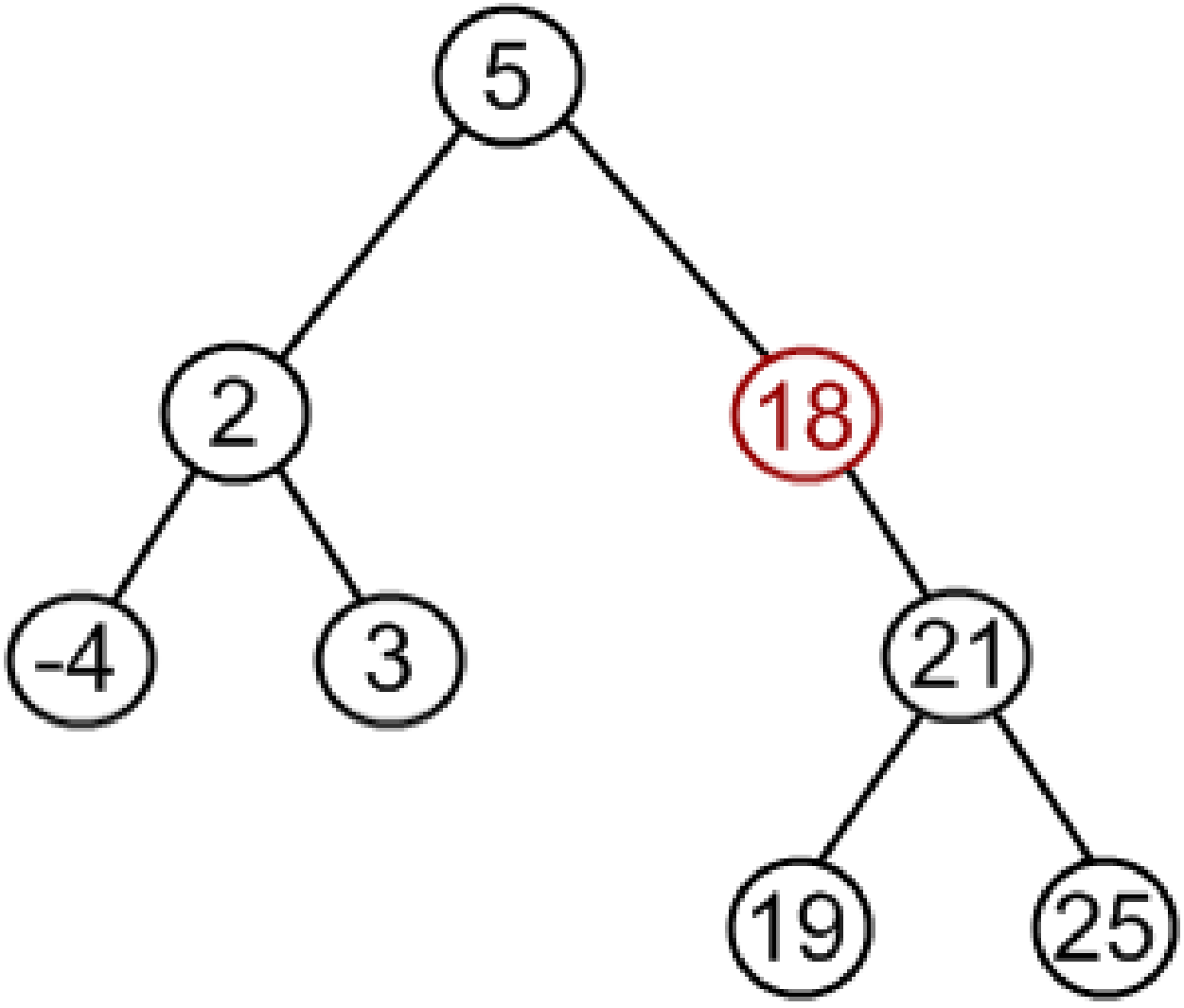
exists - Binary search tree

$$exists :: Ord\ a \Rightarrow a \rightarrow Tree\ a \rightarrow Bool$$

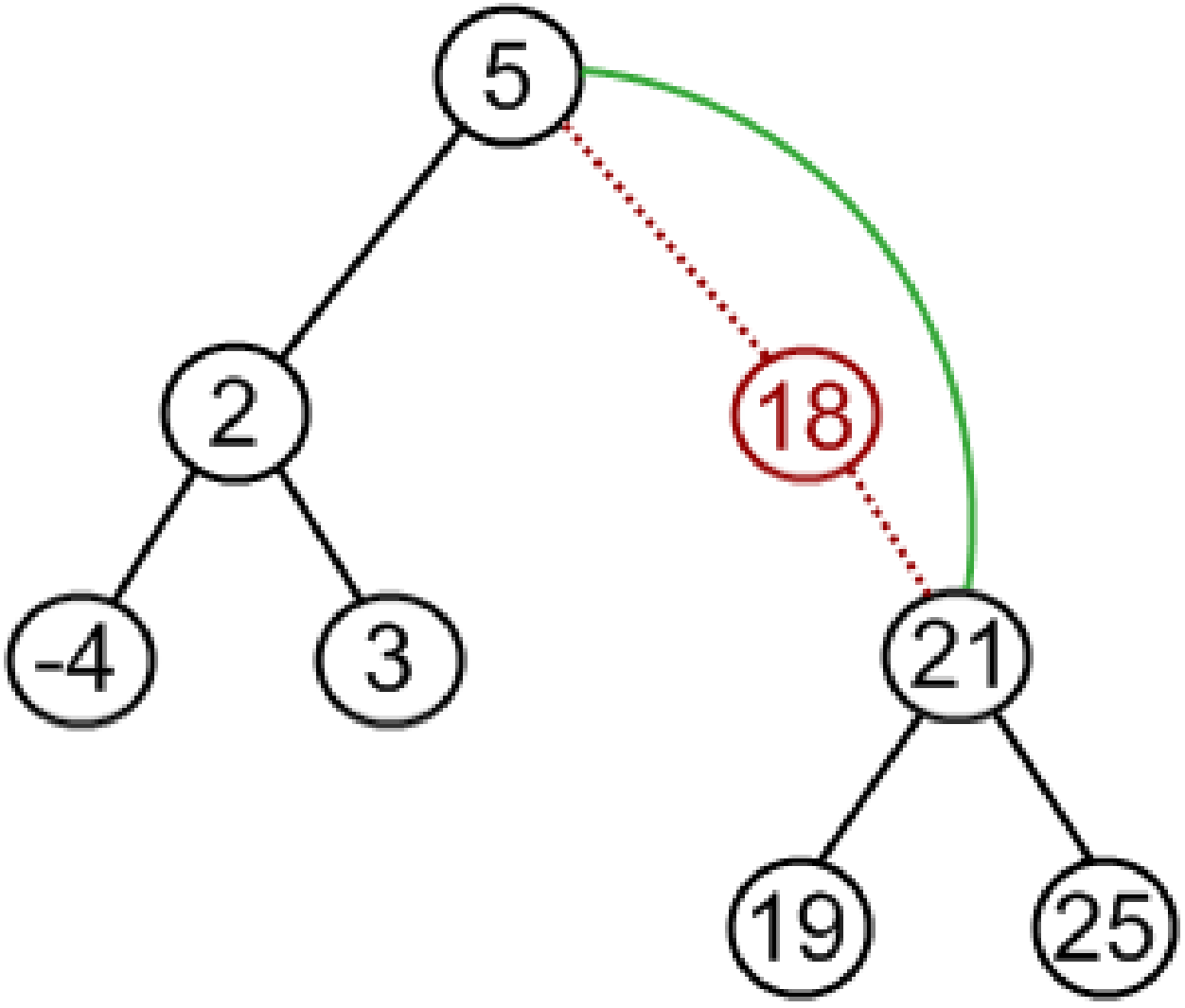
exists - Binary search tree

$$
\begin{aligned}
&\mathit{exists} :: \mathit{Ord}\ a \Rightarrow a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Bool} \\
&\mathit{exists}\ x\ \mathit{Nil} \qquad\qquad\quad = \mathit{False} \\
&\mathit{exists}\ x\ (\mathit{Node}\ y\ l\ r) = \mathbf{case}\ (\mathit{compare}\ x\ y)\ \mathbf{of} \\
&\quad \mathit{LT} \rightarrow \mathit{exists}\ x\ l \\
&\quad \mathit{EQ} \rightarrow \mathit{True} \\
&\quad \mathit{GT} \rightarrow \mathit{exists}\ x\ r
\end{aligned}
$$

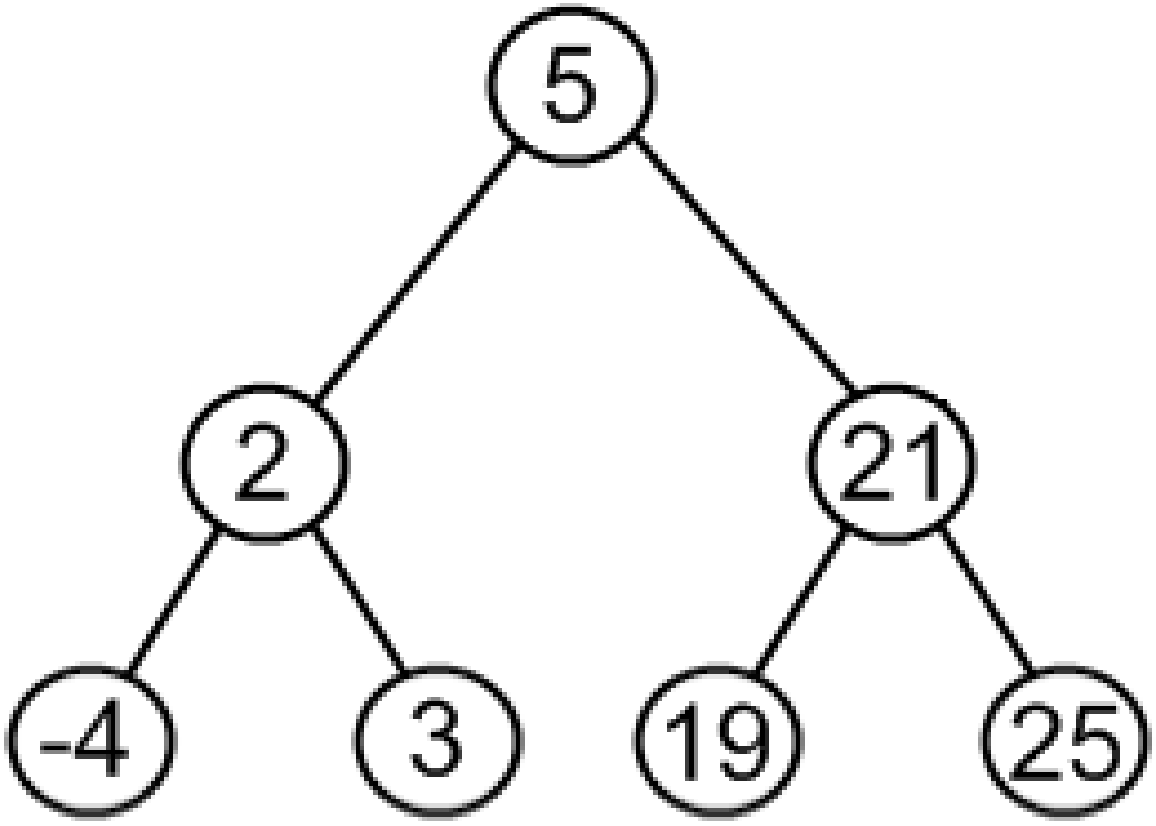
delete - Removal of node with 1 child



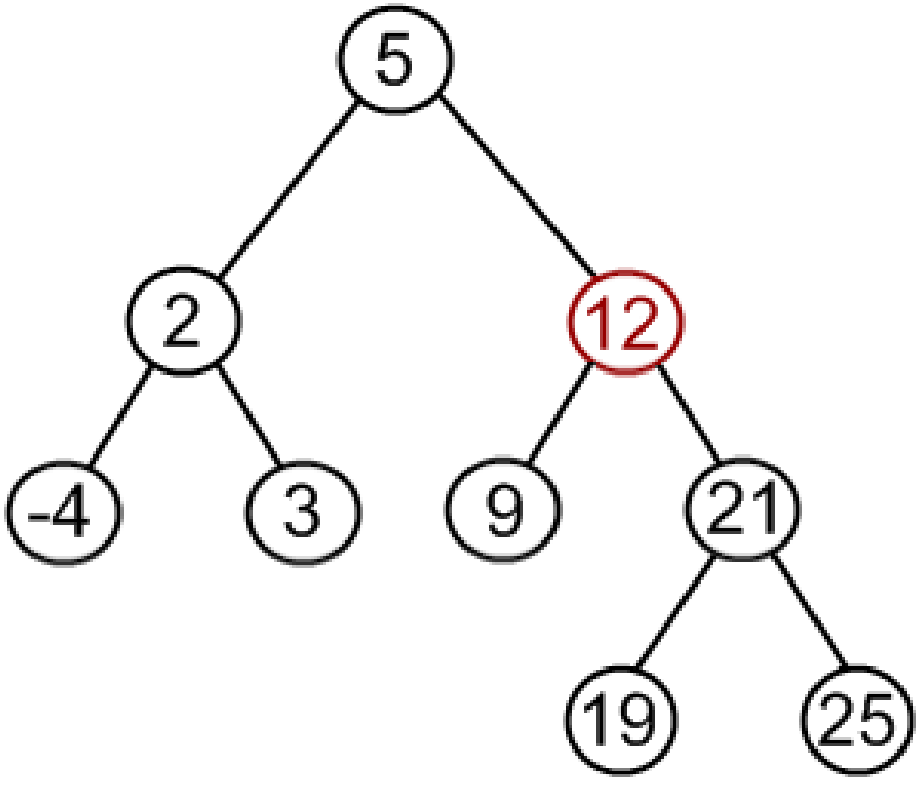
delete - Removal of node with 1 child



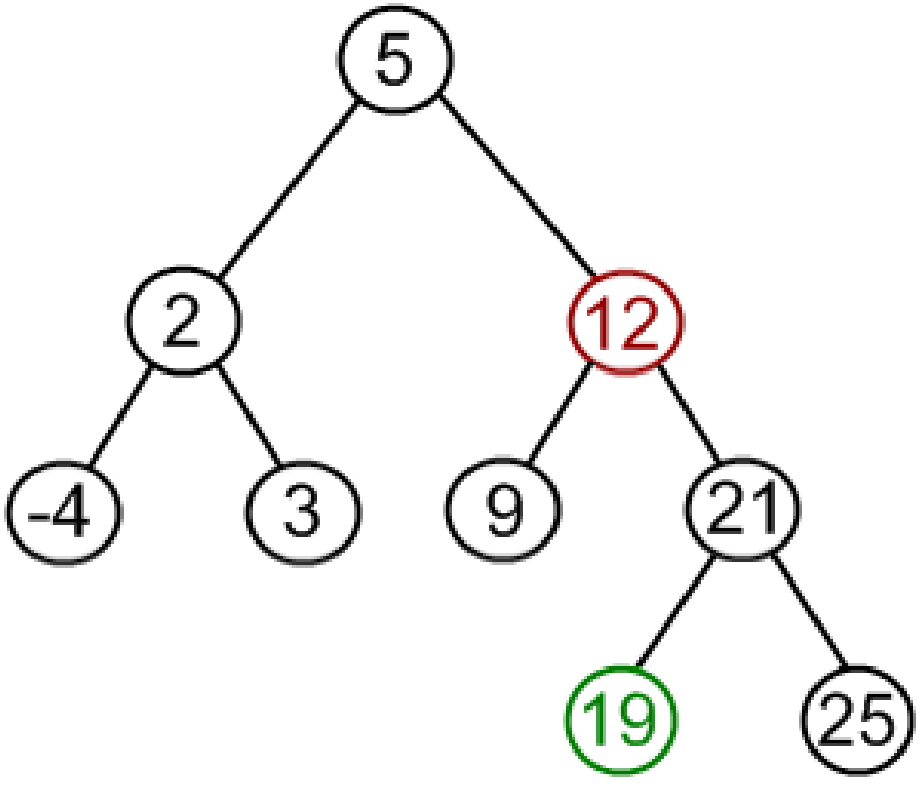
delete - Removal of node with 1 child



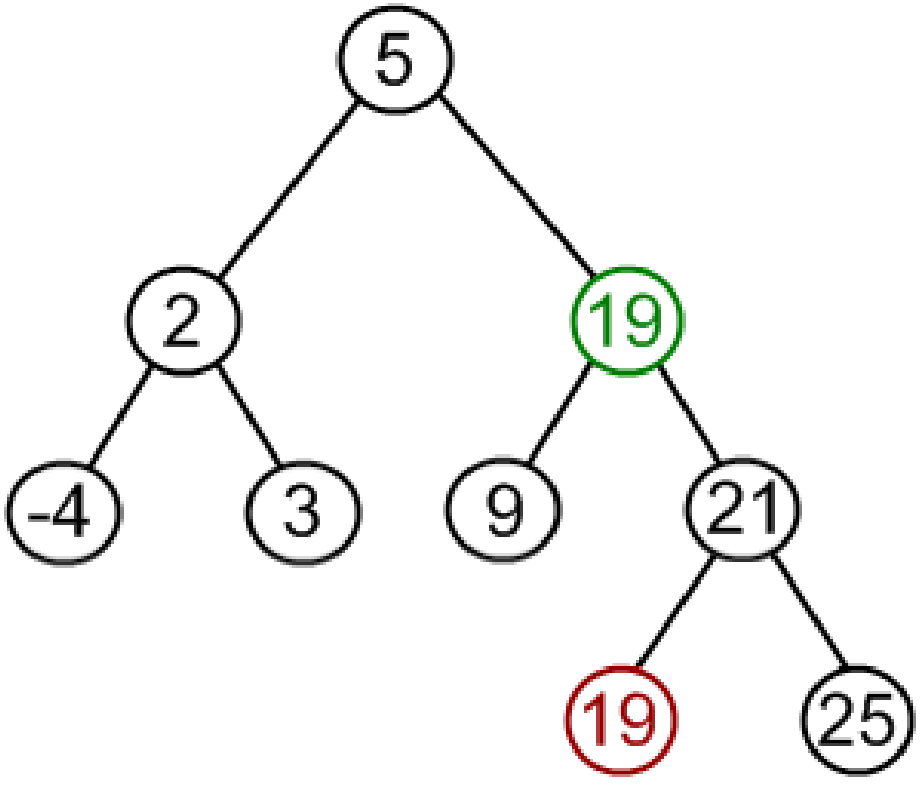
delete - Removal of node with 2 children



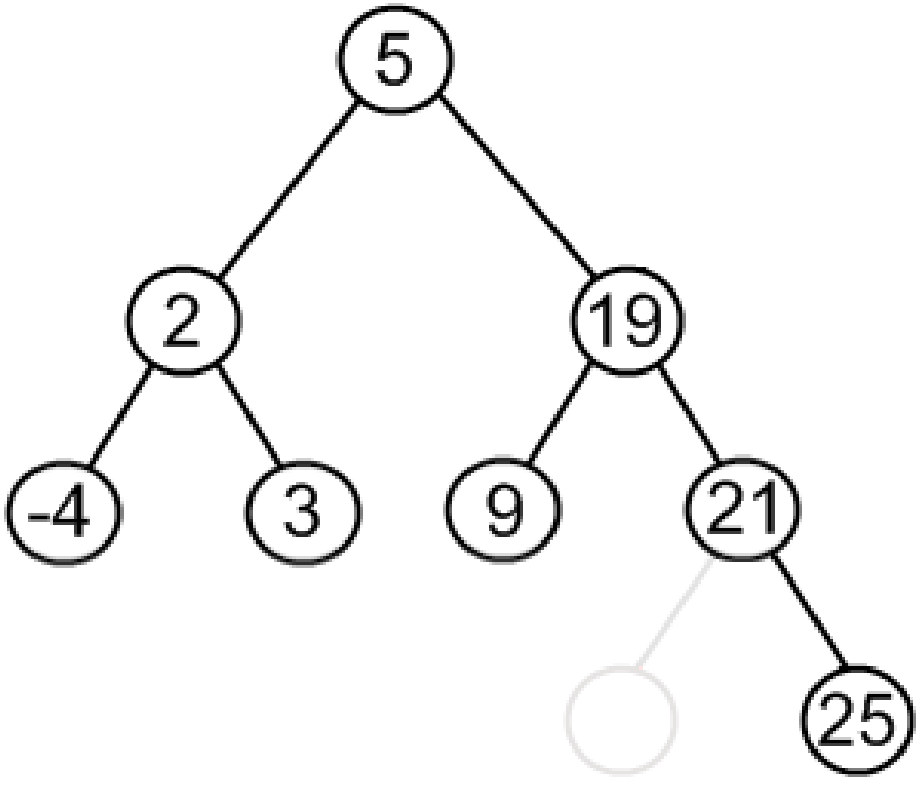
delete - Removal of node with 2 children



delete - Removal of node with 2 children



delete - Removal of node with 2 children



height and *exists* of tree - any binary tree

Now consider a binary tree without ordering constraints. How should we define these functions?

$$height :: Tree\ a \rightarrow Int$$

$$exists :: Eq\ a \Rightarrow a \rightarrow Tree\ a \rightarrow Bool$$

height and *exists* of tree - any binary tree

$$
\begin{aligned}
&height :: Tree\ a \to Int \\
&height\ Nil \qquad\qquad = 0 \\
&height\ (Node\ \_\ l\ r) = 1 + max\ (height\ l)\ (height\ r) \\
&exists :: Eq\ a \Rightarrow a \to Tree\ a \to Bool \\
&exists\ x\ Nil \qquad\qquad = False \\
&exists\ x\ (Node\ y\ l\ r) = x \equiv y \vee exists\ x\ l \vee exists\ x\ r
\end{aligned}
$$

Depth-first search - Inorder tree walk

We want to list all elements of the tree.
So for a binary search tree, the output will be a sorted list.

$$flatten :: Tree\ a \rightarrow [a]$$

Depth-first search - Inorder tree walk

$$
\begin{aligned}
&\mathit{flatten} :: \mathit{Tree}\ a \to [a] \\
&\mathit{flatten}\ \mathit{Nil} &&= [\,] \\
&\mathit{flatten}\ (\mathit{Node}\ x\ l\ r) &&= \mathit{flatten}\ l +\!\!+ [x] +\!\!+ \mathit{flatten}\ r
\end{aligned}
$$

Breath-first flatten

We want to output all nodes in the tree, but now as they would occur in a Breath-first search: All nodes from a level are returned from left to right, before the nodes from the next level.

$$bfflatten :: Tree\ a \to [a]$$

Breath-first flatten

$$bfflatten\ t = bfs'\ [t]$$

$$bfs' :: [Tree\ a] \to [a]$$

$$bfs'\ [\,] = [\,]$$

$$bfs'\ ts = [x \mid Node\ x\ \_\ \_ \leftarrow ts] +\!\!+ bfs'\ [t \mid Node\ \_\ l\ r \leftarrow ts, t \leftarrow [l, r]]$$

Challenge: Depth first enumeration

Given any tree *Tree a*, output the tree *Tree Int* where each node should be numbered by its occurrence in a depth first search.

$$dfsnumber :: Tree\ a \rightarrow Tree\ Int$$

Challenge: Depth first enumeration

Given any tree *Tree a*, output the tree *Tree Int* where each node should be numbered by its occurrence in a depth first search.

$$
\begin{aligned}
&dfsnumber :: Tree\ a \to Tree\ Int \\
&dfsnumber\ t = snd\ (dfs\ (1, t)) \\
&dfs :: (Int, Tree\ a) \to (Int, Tree\ Int) \\
&dfs\ (n, Nil) = (n, Nil) \\
&dfs\ (n, Node\ v\ l\ r) = (n'', Node\ n\ l'\ r') \\
&\quad \textbf{where} \\
&\qquad (n', l') = dfs\ (n + 1, l) \\
&\qquad (n'', r') = dfs\ (n', r)
\end{aligned}
$$

Red-Black Trees

$$\textbf{data}\ \mathit{Color} = R \mid B$$
$$\textbf{data}\ \mathit{Tree}\ a = E \mid T\ \mathit{Color}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)$$

$$\mathit{insert} :: \mathit{Ord}\ a \Rightarrow a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Tree}\ a$$
$$\mathit{insert}\ x\ s = \mathit{makeBlack}\ (\mathit{ins}\ s)$$
$$\quad \textbf{where}\ \mathit{ins}\ E = T\ R\ E\ x\ E$$
$$\mathit{ins}\ (T\ \mathit{color}\ a\ y\ b) \mid x < y = \mathit{balance}\ \mathit{color}\ (\mathit{ins}\ a)\ y\ b$$
$$\mid x \equiv y = T\ \mathit{color}\ a\ y\ b$$
$$\mid x > y = \mathit{balance}\ \mathit{color}\ a\ y\ (\mathit{ins}\ b)$$
$$\mathit{makeBlack}\ (T\ \_\ a\ y\ b) = T\ B\ a\ y\ b$$

Red-Black Trees

$$\textbf{data}\ \mathit{Color} = R \mid B$$
$$\textbf{data}\ \mathit{Tree}\ a = E \mid T\ \mathit{Color}\ (\mathit{Tree}\ a)\ a\ (\mathit{Tree}\ a)$$

$$\begin{aligned}
&\mathit{balance}\ B\ (T\ R\ (T\ R\ a\ x\ b)\ y\ c)\ z\ d = T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d)\\
&\mathit{balance}\ B\ (T\ R\ a\ x\ (T\ R\ b\ y\ c))\ z\ d = T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d)\\
&\mathit{balance}\ B\ a\ x\ (T\ R\ (T\ R\ b\ y\ c)\ z\ d) = T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d)\\
&\mathit{balance}\ B\ a\ x\ (T\ R\ b\ y\ (T\ R\ c\ z\ d)) = T\ R\ (T\ B\ a\ x\ b)\ y\ (T\ B\ c\ z\ d)\\
&\mathit{balance}\ \mathit{color}\ a\ x\ b = T\ \mathit{color}\ a\ x\ b
\end{aligned}$$

Type classes for your ADT - *Eq*, *Ord*, *Show*

If the type a is an instance of the type class $Eq\ a$, we know that $(\equiv)$ is defined for it.

$$\textbf{class}\ Eq\ a\ \textbf{where}$$
$$\quad (\equiv) :: a \to a \to Bool$$

We can implement equality for $\textbf{data}\ Bool = False \mid True$ by

$$\textbf{instance}\ Eq\ (Bool)\ \textbf{where}$$
$$\begin{aligned} \quad True \equiv True &= True \\ \quad False \equiv False &= True \\ \quad \_ \equiv \_ &= False \end{aligned}$$

Type classes for your ADT - *Eq*, *Ord*, *Show*

- This can be generalized further. Consider:

  data $Tree\ a = Nil \mid Node\ a\ (Tree\ a)\ (Tree\ a)$

  Assuming we can compare values of a, we can recursively define comparison for $Tree\ a$ by

  instance $(Eq\ a) \Rightarrow Eq\ (Tree\ a)$ **where**
  $Nil \equiv Nil = True$
  $(Node\ v1\ l1\ r1) \equiv (Node\ v2\ l2\ r2)$
  $\quad = v1 \equiv v2 \wedge l1 \equiv l2 \wedge r1 \equiv r2$
  $\_ \equiv \_ \quad = False$

- This function can be automatically derived by

  data $Tree\ a = Nil \mid Node\ a\ (Tree\ a)\ (Tree\ a)$ **deriving** Eq

- How would the "trivial" functions look like for *Ord* and *Show*?

Remember labs this Friday!

- 9 exercises about binary search trees
- You can apply many of the ideas we have seen today
- Ex. 8 and 9 about how to make it balanced