Lecture 3: Algebraic Data Types

Søren Haagerup

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

September 19, 2017

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 $a \rightarrow b$

The **data** keyword

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

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

This gives rise to constructor functions

- $False :: Bool$
- $True :: Bool$

corresponding patterns *False*, *True*.

Enumeration types in general

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

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

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

Product types

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

$$
\begin{aligned}
&\textbf{type}\ Age \quad = Int\\
&\textbf{type}\ Name = String\\
&\textbf{data}\ Person = Person\ Age\ Name
\end{aligned}
$$

$$
\begin{aligned}
&isOld :: Person \to Bool\\
&isOld\ (Person\ a\ \_)\mid a \geqslant 30 = True\\
&isOld\ \_ \qquad\qquad\qquad\qquad\quad = False
\end{aligned}
$$

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

Difference between tuples and general product types

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

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

- And somewhere else, we have defined

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

- 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

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

  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

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

  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 $add :: Nat \rightarrow Nat \rightarrow Nat$?
- Come up with an alternative definition of *add*.
- How can we implement $toInt :: Nat \rightarrow Int$?
- Come up with an alternative definition of *toInt*.
- Which list function does the functions above remind us of?
- How can we implement $min :: Nat \rightarrow Nat \rightarrow Nat$?
- Which list function does this remind us of?

Example: File System

$$
\begin{array}{l}
\textbf{data}\ \mathit{Entry} = \mathit{File}\ \{ \\
\quad\quad \mathit{name} :: \mathit{String} \\
\quad \}\ |\ \mathit{Folder}\ \{ \\
\quad\quad \mathit{name} :: \mathit{String}, \\
\quad\quad \mathit{entries} :: [\mathit{Entry}] \\
\quad \}
\end{array}
$$

$$
\begin{array}{l}
\mathit{entry} :: \mathit{Entry} \\
\mathit{entry} = \mathit{Folder}\ \texttt{"Documents"}\ [\\
\quad \mathit{File}\ \texttt{"File1"}, \\
\quad \mathit{File}\ \texttt{"File2"}, \\
\quad \mathit{Folder}\ \texttt{"Pictures"}\ [\\
\quad\quad \mathit{File}\ \texttt{"Picture 1"}, \\
\quad\quad \mathit{File}\ \texttt{"Picture 2"} \\
\quad] \\
\quad]
\end{array}
$$

Bigger example: JSON data

```haskell
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?

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

- Another example of a polymorphic type

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

- Another example of a polymorphic type

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

Error handling + Record update syntax

Syddansk Universitet
University of Southern Denmark

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

$$\begin{aligned}
&\mathit{rename} :: \mathit{String} \to \mathit{Entry} \to \mathit{Entry}\\
&\mathit{rename}\ n\ e = e\ \{\mathit{name} = n\}
\end{aligned}$$

Polymorphic, recursive types

We could define a linked list as:

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

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

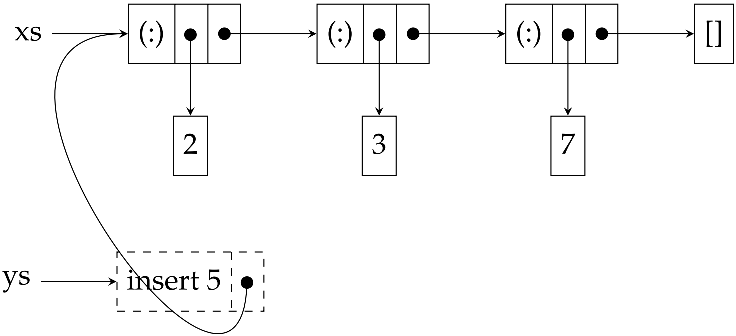
$$\textbf{data}\ [a] = [\,] \mid a : [a] \quad \text{-- (not legal Haskell code!)}$$

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

$$length :: List\ a \rightarrow Int$$

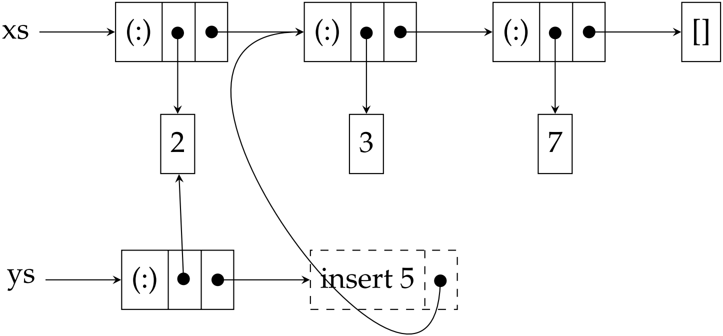
Insertion in a list

- 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:



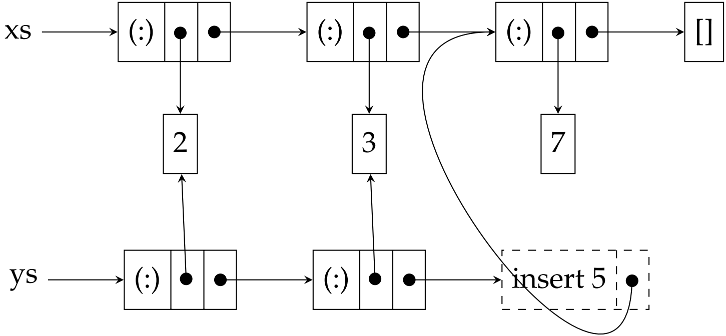
Insertion in a list

- 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:



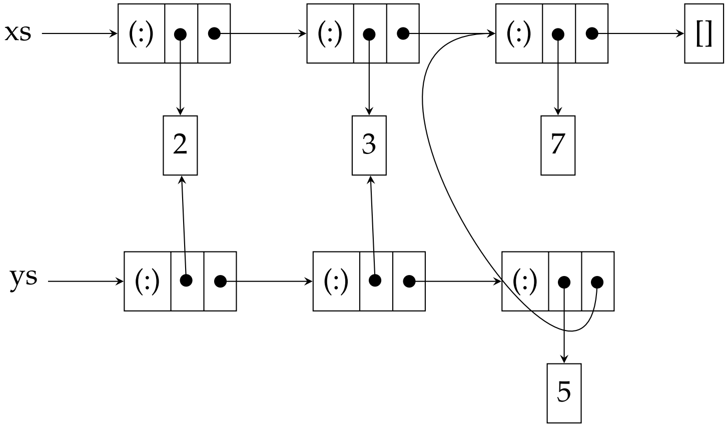
Insertion in a list

- 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

Syddansk Universitet
University of Southern Denmark

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

How to use Data.Map

```haskell
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'
```

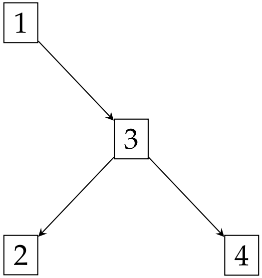
Binary Search Tree

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

Example:

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

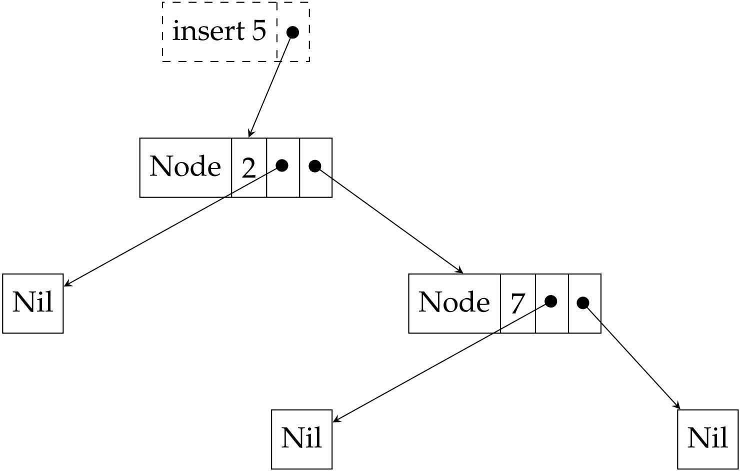
corresponds to the tree



Binary Search Tree

We are given **data** *Tree a* = *Nil* | *Node a* (*Tree a*) (*Tree a*).

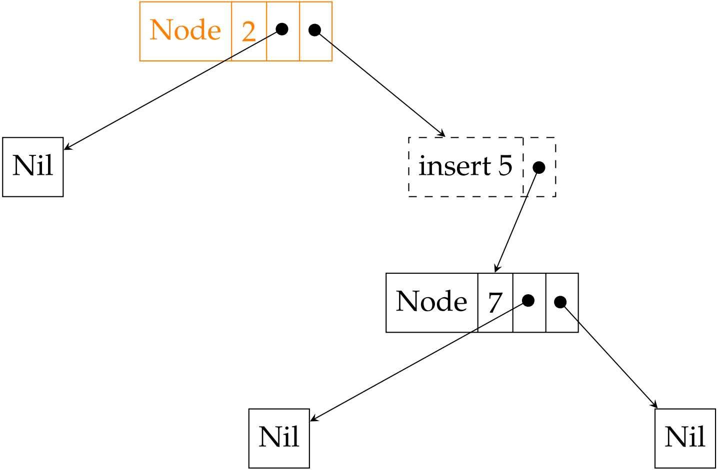
How should we define *insert* :: *Ord a* ⇒ *a* → *Tree a* → *Tree a*?



Binary Search Tree

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

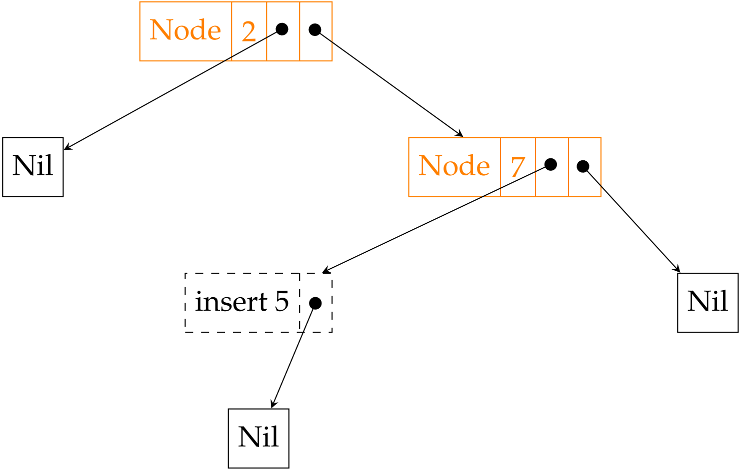
How should we define $insert :: Ord\ a \Rightarrow a \rightarrow Tree\ a$?



Binary Search Tree

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

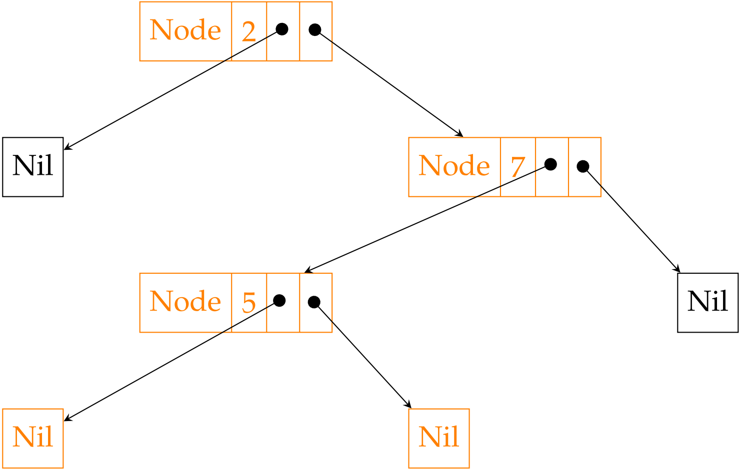
How should we define $insert :: Ord\ a \Rightarrow a \rightarrow Tree\ a$?



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

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

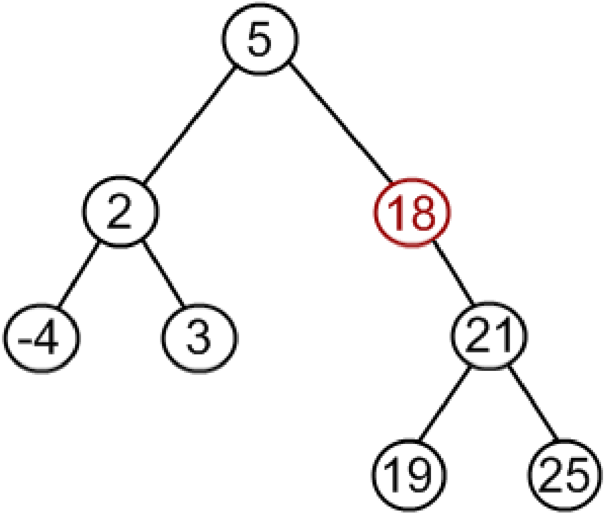
exists - Binary search tree

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

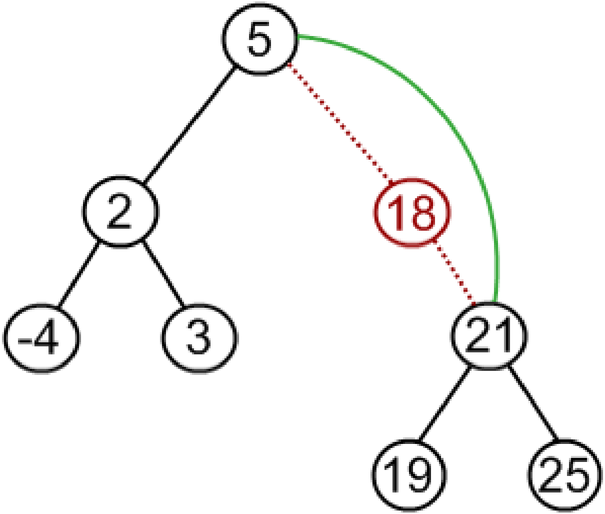
exists - Binary search tree

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

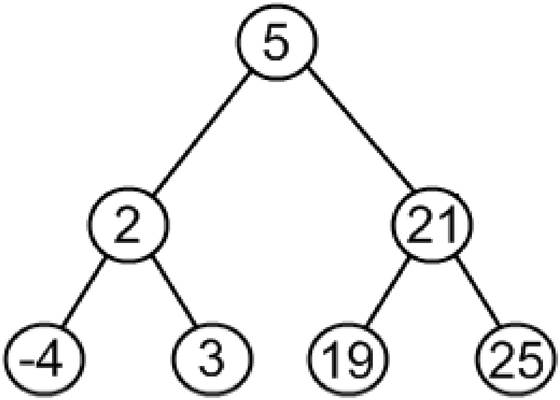
delete - Removal of node with 1 child



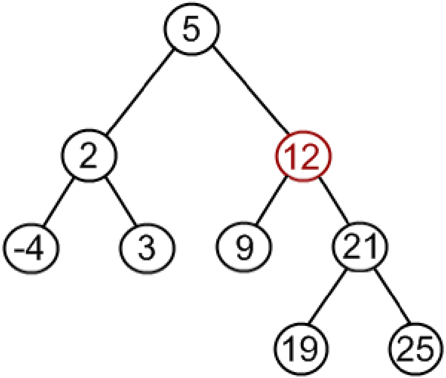
delete - Removal of node with 1 child



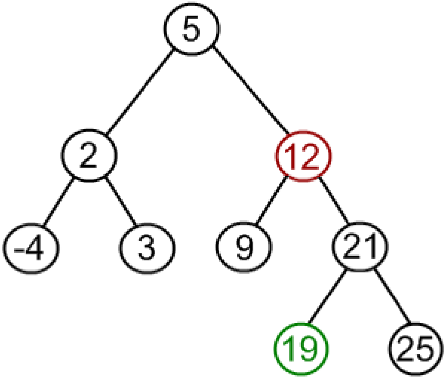
delete - Removal of node with 1 child



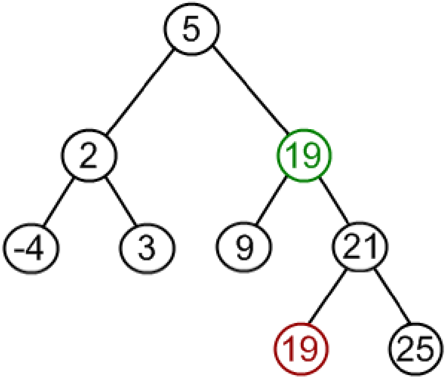
delete - Removal of node with 2 children



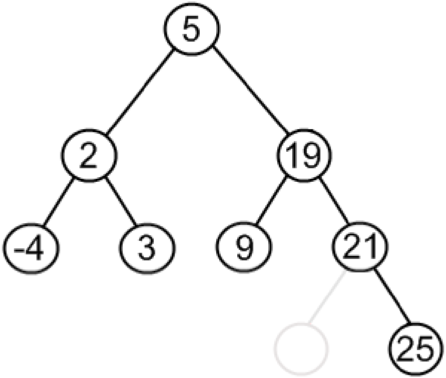
delete - Removal of node with 2 children



delete - Removal of node with 2 children



delete - Removal of node with 2 children



height and *exists* of tree - any binary tree

Syddansk Universitet
University of Southern Denmark

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

$$height :: Tree\ a \to Int$$

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

height and *exists* of tree - any binary tree

$$\begin{aligned}
&\mathit{height} :: \mathit{Tree}\ a \rightarrow \mathit{Int}\\
&\mathit{height}\ \mathit{Nil} \qquad\qquad\qquad = 0\\
&\mathit{height}\ (\mathit{Node}\ _\ l\ r) = 1 + \mathit{max}\ (\mathit{height}\ l)\ (\mathit{height}\ r)
\end{aligned}$$

$$\begin{aligned}
&\mathit{exists} :: \mathit{Eq}\ a \Rightarrow a \rightarrow \mathit{Tree}\ a \rightarrow \mathit{Bool}\\
&\mathit{exists}\ x\ \mathit{Nil} \qquad\qquad\quad = \mathit{False}\\
&\mathit{exists}\ x\ (\mathit{Node}\ y\ l\ r) = x \equiv y \vee \mathit{exists}\ x\ l \vee \mathit{exists}\ x\ r
\end{aligned}$$

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.

$$\mathit{flatten} :: \mathit{Tree}\ a \rightarrow [a]$$

Depth-first search - Inorder tree walk

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

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.

$$\mathit{bfflatten} :: \mathit{Tree}\ a \rightarrow [a]$$

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.

$$\mathit{dfsnumber} :: \mathit{Tree}\ a \to \mathit{Tree}\ \mathit{Int}$$

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.

$$
\begin{aligned}
&\mathit{dfsnumber} :: \mathit{Tree}\ a \rightarrow \mathit{Tree}\ \mathit{Int} \\
&\mathit{dfsnumber}\ t = \mathit{snd}\ (\mathit{dfs}\ (1, t))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{dfs} :: (\mathit{Int}, \mathit{Tree}\ a) \rightarrow (\mathit{Int}, \mathit{Tree}\ \mathit{Int}) \\
&\mathit{dfs}\ (n, \mathit{Nil}) = (n, \mathit{Nil}) \\
&\mathit{dfs}\ (n, \mathit{Node}\ v\ l\ r) = (n'', \mathit{Node}\ n\ l'\ r') \\
&\quad \mathbf{where} \\
&\qquad (n', l') = \mathit{dfs}\ (n + 1, l) \\
&\qquad (n'', r') = \mathit{dfs}\ (n', r)
\end{aligned}
$$

Red-Black Trees

$$
\begin{array}{l}
\textbf{data}\ Color = R \mid B \\
\textbf{data}\ Tree\ a = E \mid T\ Color\ (Tree\ a)\ a\ (Tree\ a)
\end{array}
$$

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

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.

$$
\begin{array}{l}
\textbf{class}\ Eq\ a\ \textbf{where} \\
\quad (\equiv) :: a \rightarrow a \rightarrow Bool
\end{array}
$$

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

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

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

- This can be generalized further. Consider:

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

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

  $$\begin{aligned}
  &\textbf{instance}\ (\mathit{Eq}\ a) \Rightarrow \mathit{Eq}\ (\mathit{Tree}\ a)\ \textbf{where}\\
  &\quad \mathit{Nil} \equiv \mathit{Nil} = \mathit{True}\\
  &\quad (\mathit{Node}\ v1\ l1\ r1) \equiv (\mathit{Node}\ v2\ l2\ r2)\\
  &\qquad = v1 \equiv v2 \wedge l1 \equiv l2 \wedge r1 \equiv r2\\
  &\quad _ \equiv _ \qquad = \mathit{False}
  \end{aligned}$$

- This function can be automatically derived by

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

- 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