Welcome to Programming Fundamentals 3

University of Luxembourg

2025/2026

Bachelor in Computer Sciences (BICS)
Semester 3

Lecture - Week #10

Dr. Afonso DELERUE ARRIAGA afonso.delerue@uni.lu

Important: Please remember to register your attendance in Moodle!

From unbalanced to self-balancing trees...

* Imagine generated with Microsoft Designer

Pretty printing trees

- Go to Hackage website and look for a package to draw trees.

- I've selected `pretty-tree` by Miljenovic because it includes a function called `drawVerticalTree` that does what I need.
- Install and import package:

```
% cabal update
% cabal install --lib pretty-tree
% cabal install --lib containers-0.6.7

import qualified Data.Tree as T (Tree(..), drawTree)
import Data.Tree.Pretty (drawVerticalTree)
```

- Go to the source code of the package and see how the type Tree is defined:

```
data Tree a = Node {
        rootLabel :: a,         -- ^ label value
        subForest :: Forest a   -- ^ zero or more child trees
    }

type Forest a = [Tree a]
```

- Data type `Tree a` is a rose tree and nodes are labelled.

How to use `Data.Tree.Pretty`

1. Create a function that converts `BTree` to `T.Tree`.
2. Make our `BTree` type an instance of class show using `drawVerticalTree` from `Data.Tree.Pretty` and our conversion function.

Self-balancing trees

- Operations that might unbalance a tree:
    1. `insert`
    2. `delete`
- `search` operations don't change the tree structure.
- There are multiple self-balancing algorithms, most notably:
    - AVL trees
    - Red-Black trees
- Today, we'll implement the algorithm invented by Adelson-Velsky and Landis in their 1962 paper "An algorithm for the organization of information".
- AVL trees have two properties:
    1. Order: Just like BST, each node must be greater than the nodes on its left branch, and lesser than the nodes in its right branch.
    2. Balance: The heights of the two child subtrees of any node differ by at most one.

Steps:

- We start by creating auxiliary functions `height` and `balanceFactor`. Don't worry about efficiency just yet. Let's getting something working first.
- Observe that we can rotate nodes to the left and to the right without compromising the property order. With both rotation we still have a BST. Define functions `rotateLeft` and `rotateRight`.

Source code

```
module AVLTree where

import qualified Data.Tree as T (Tree(..), drawTree)
import Data.Tree.Pretty (drawVerticalTree)

{-
data Tree a = Node {
        rootLabel :: a,         -- ^ label value
        subForest :: Forest a   -- ^ zero or more child trees
    }

type Forest a = [Tree a]
-}

data BTree a = Leaf | Node a (BTree a) (BTree a)
  deriving (Eq)

instance Show a => Show (BTree a) where
    show :: BTree a -> String
    show tree = drawVerticalTree $ convertTree tree

convertTree :: Show a => BTree a -> T.Tree String
convertTree Leaf = T.Node { T.rootLabel="Leaf", T.subForest= []
}
convertTree (Node x left right) = T.Node {T.rootLabel = show x ,
T.subForest = [convertTree left, convertTree right] }

search :: Ord a => a -> BTree a -> Bool
search x Leaf = False
search x (Node y left right)
    | x < y = search x left
    | x > y = search x right
    | otherwise = True

insert :: Ord a => a -> BTree a -> BTree a
insert x Leaf = Node x Leaf Leaf
insert x t@(Node y left right)
    | x < y = Node y (insert x left) right
    | x > y = Node y left (insert x right)
    | otherwise = t
```

```
delete :: Ord a => a -> BTree a -> BTree a
delete x Leaf = error "cannot delete element" -- in the
definition of BTree a, leaves are not labeled
delete x (Node y left right)
    | x < y = Node y (delete x left) right
    | x > y = Node y left (delete x right)
    | left == Leaf = right -- at this point we have that x == y
    | right == Leaf = left
    | otherwise = let z = leftmost right -- alternative, fetch
the rightmost node of the left branch
                  in Node z left (delete z right)
  where
    leftmost :: BTree a -> a
    leftmost Leaf = error "cannot be here"
    leftmost (Node n Leaf _) = n
    leftmost (Node n l r) = leftmost l

t1 :: BTree Int
t1 = foldl (flip insert) Leaf [5,4,3,7,1,10,8]

t2 :: BTree Int
t2 = foldl (flip insert) Leaf [1..10]

-- tree traversals
-- inorder (left, root, right)
inorder :: BTree a -> [a]
inorder Leaf = []
inorder (Node x left right) = inorder left ++ [x] ++ inorder
right

-- preorder traversal (root, left, right)
preorder :: BTree a -> [a]
preorder Leaf = []
preorder (Node x left right) = [x] ++ preorder left ++ preorder
right

-- postorder traversal (left, right, root)
postorder :: BTree a -> [a]
postorder Leaf = []
postorder (Node x left right) = postorder left ++ postorder
right ++ [x]

-- aux functions
height :: BTree a -> Int
height Leaf = 0
height (Node x left right) = 1 + max (height left) (height
right)

balanceFactor :: BTree a -> Int
balanceFactor Leaf = 0
balanceFactor (Node _ left right) = height left - height right

rotateLeft :: BTree a -> BTree a
rotateLeft (Node x t1 (Node y t2 t3)) = Node y (Node x t1 t2) t3
rotateLeft t = t
```

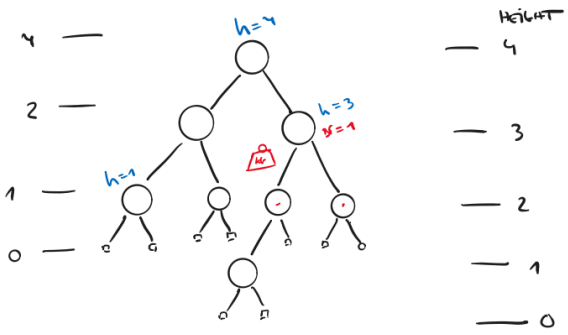
```
rotateRight :: BTree a -> BTree a
rotateRight (Node x (Node y t1 t2) t3) = Node y t1 (Node x t2
t3)
rotateRight t = t
```

Whiteboard

AVL TREES

- ADELSON-VELSKY + LANDIS, 1962
- SELF-BALANCING TREES (ALTERNATIVE: RED-BLACK TREES)
- PROPERTIES: ① ORDER (i.e. BST)

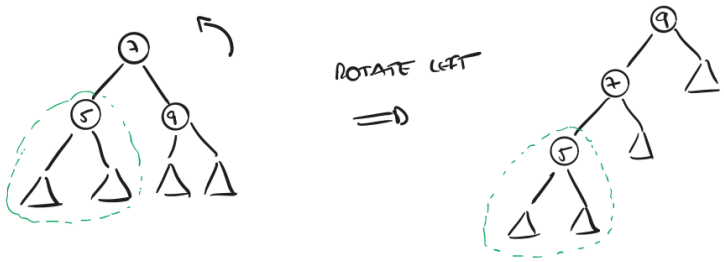
  ② BALANCE
  - THE HEIGHTS OF THE TWO CHILDREN DIFFER BY AT MOST ONE, ∀ NODES.

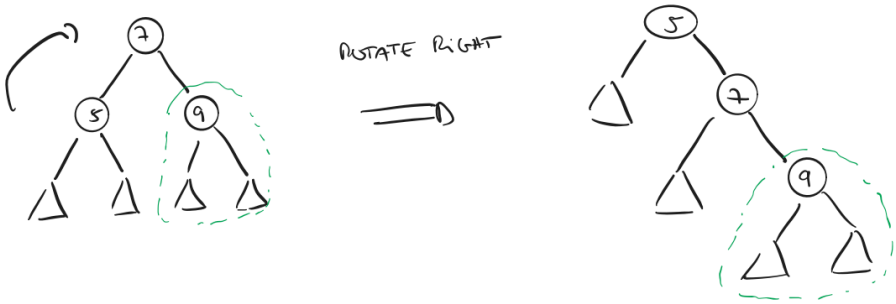


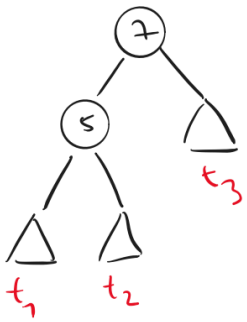
BF = HEIGHT LEFT − HEIGHT RIGHT

- IF A TREE IS BALANCED, INSERT AND DELETE MIGHT UNBALANCE IT.
  - REBALANCING MIGHT BE REQUIRED.

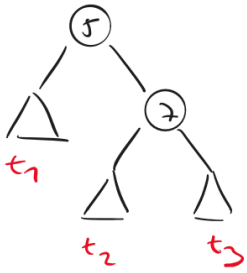
- ROTATIONS THAT PRESERVE ORDER







ROTATE RIGHT
(SIMPLIFIED PATTERN)
⟹



- SOMETIMES, TREES THAT LOOK SLIGHTLY UNBALANCED ARE STILL BALANCED BY DEF.

EXAMPLE