Welcome to Programming Fundamentals 3

University of Luxembourg

2025/2026

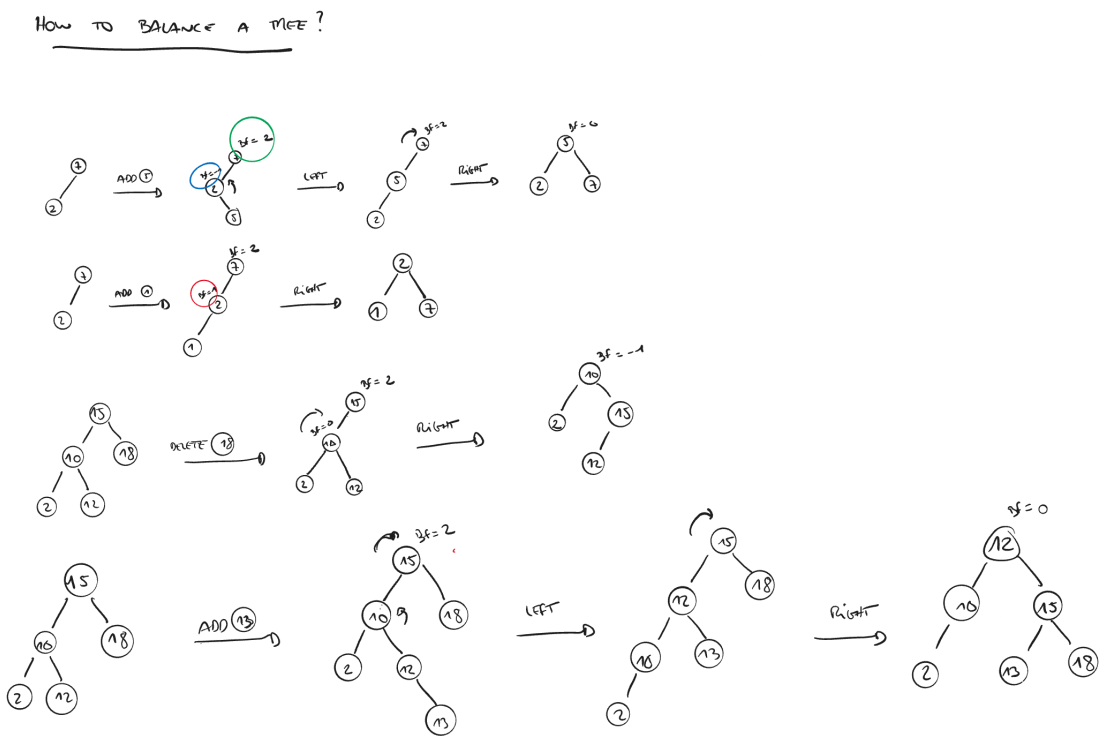
Bachelor in Computer Sciences (BICS)
Semester 3

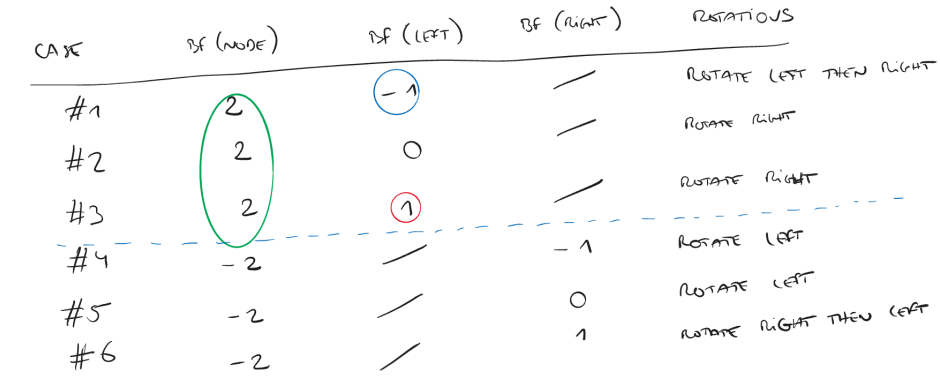
Lecture - Week #10

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

Important: Please remember to register your attendance in Moodle!

Whiteboard 🌳





Haskell source-code

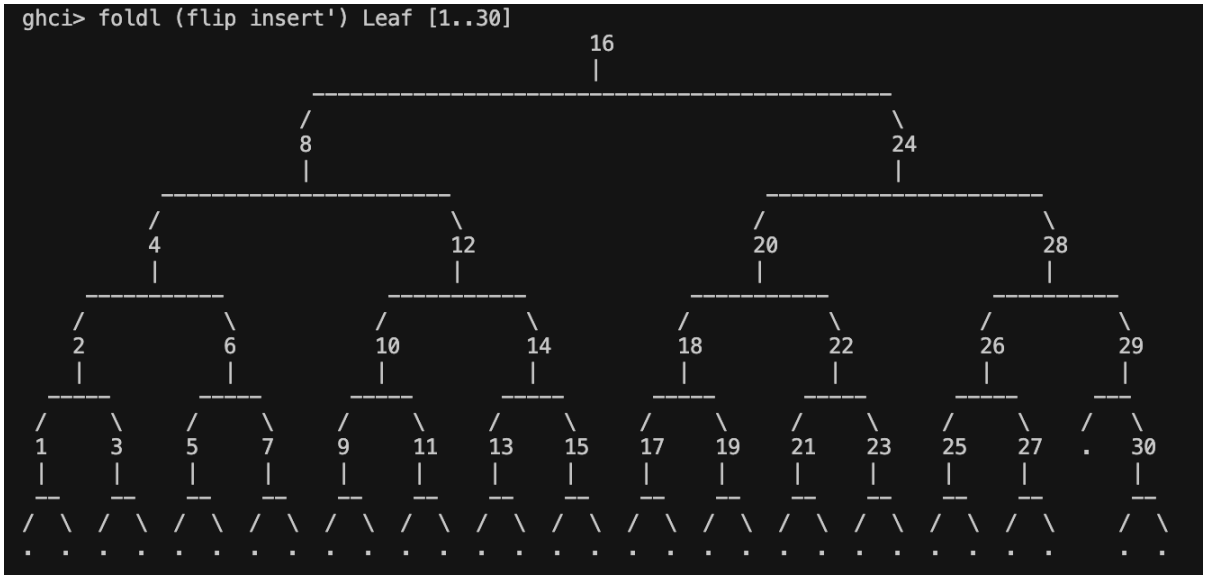
```
module AVLTree where

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) Int
  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 _ _ _ h) = h

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
h1) t3 h2
  where
    h1 = 1 + max (height t1) (height t2)
    h2 = 1 + max h1 (height t3)
rotateLeft t = t

rotateRight :: BTree a -> BTree a
rotateRight (Node x (Node y t1 t2 _) t3 _) = Node y t1 (Node x
t2 t3 h1) h2
  where
    h1 = 1 + max (height t2) (height t3)
    h2 = 1 + max (height t1) h1
rotateRight t = t

-- if a BTree is balanced, adding or deleting a node can only
sightly unbalance the tree
bf = balanceFactor

rebalance :: BTree a -> BTree a
rebalance Leaf = Leaf
rebalance t@(Node x left right h)
    | bf t == 2  && bf left == -1  = rotateRight $ Node x
(rotateLeft left) right 0
    | bf t == 2  && bf left == 0   = rotateRight t
    | bf t == 2  && bf left == 1   = rotateRight t
    | bf t == -2 && bf right == -1 = rotateLeft t
    | bf t == -2 && bf right == 0  = rotateLeft t
    | bf t == -2 && bf right == 1  = rotateLeft $ Node x left
(rotateRight right) 0
    | otherwise = t

insert' :: Ord a => a -> BTree a -> BTree a
insert' x Leaf = Node x Leaf Leaf 1
```

```
insert' x t@(Node y left right h)
    | x < y = rebalance $ Node y newLeft right h1
    | x > y = rebalance $ Node y left newRight h2
    | otherwise = t
  where
    newLeft  = insert' x left
    newRight = insert' x right
    h1 = 1 + max (height newLeft) (height right)
    h2 = 1 + max (height left) (height newRight)

t3 :: BTree Int
t3 = foldl (flip insert') Leaf [1..10]

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 h)
    | x < y = rebalance $ Node y newLeft right h1
    | x > y = rebalance $ Node y left newRight h2
    | 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
                      newRight' = delete' z right
                      h3 = 1 + max (height left) (height
newRight')
                  in rebalance $ Node z left newRight' h3
  where
    newLeft = delete' x left
    newRight = delete' x right
    h1 = 1 + max (height newLeft) (height right)
    h2 = 1 + max (height left) (height newRight)
    leftmost :: BTree a -> a
    leftmost Leaf = error "cannot be here"
    leftmost (Node n Leaf _ _) = n
    leftmost (Node n l r _) = leftmost l

-- test efficiency
-- > :set +s

-- it's not very efficient because `height` function requires
full tree traversal
-- instead, store the height of each node in BTree and update
code

-- CODED IN CLASS; PLEASE TEST AT HOME AND REPORT BUGS (IF ANY)
--
```

Self-balancing BTree


ghci> foldl (flip insert') Leaf [1..30]
16
8
24
4
12
20
28
2
6
10
14
18
22
26
29
1
3
5
7
9
11
13
15
17
19
21
23
25
27
.
30