Programming in Haskell
Aug-Nov 2015

LECTURE 18

OCTOBER 20, 2015

S P SURESH
CHENNAI MATHEMATICAL INSTITUTE

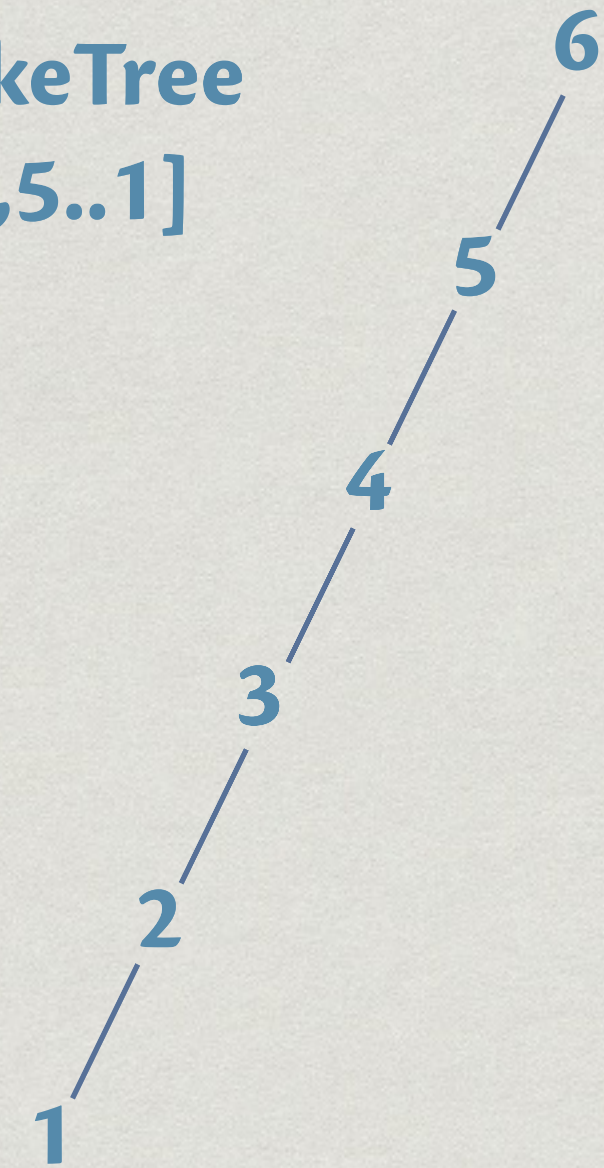
Balance

- The complexity of the key operations on trees depends on the **height** of the tree
- In general, a tree might not be balanced
- Inserting in ascending or descending order results in highly skewed trees

Balance

makeTree
[1..6]

makeTree
[6,5..1]



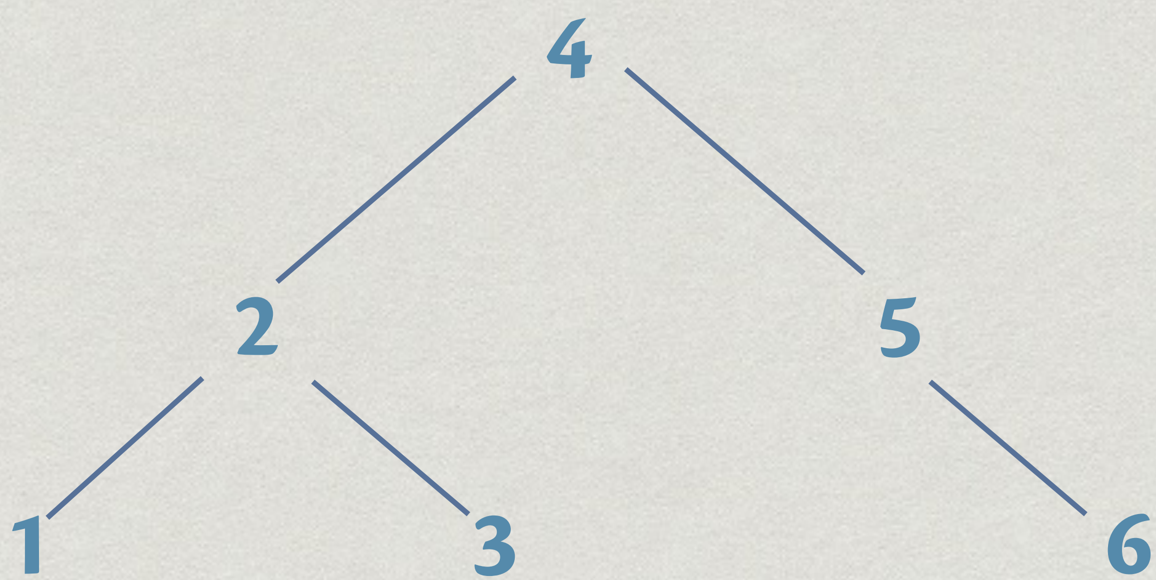
Balanced search trees

- Ideally, for each node, the left and right subtrees differ in size by at most 1
  - Height is guaranteed to be at most $\log N + 1$, where N is the size of the tree
  - When size is 1, height is also $1 = \log 1 + 1$
  - When size is $N > 1$, subtrees are of size at most $N/2$
  - Height is $1 + (\log N/2 + 1) = 1 + (\log N - 1 + 1)$
    $= \log N + 1$

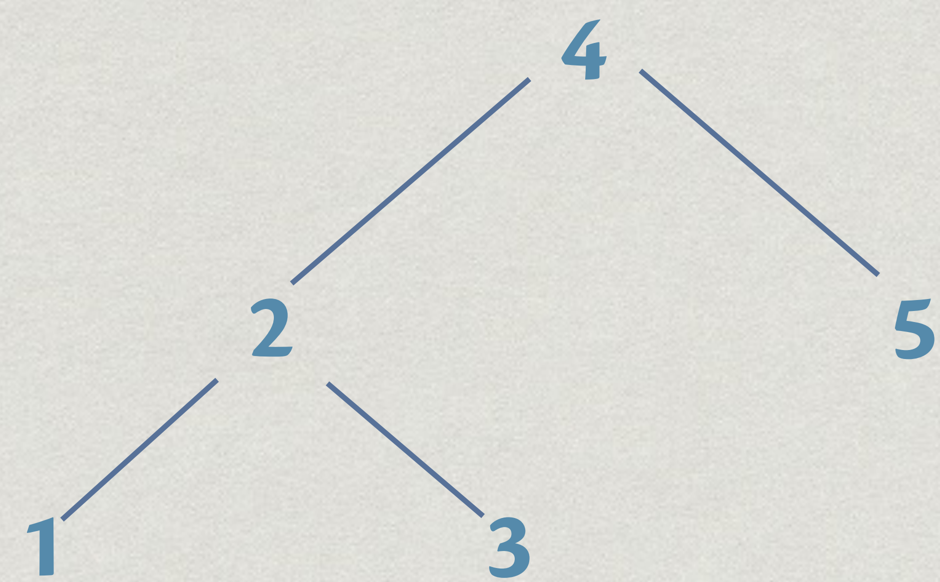
Balanced search trees

- Not easy to maintain size balance
- Maintain height balance instead
- At any node
  - The left and right subtrees differ in **height** by at most 1
  - Somewhat easier to maintain: use tree rotations
  - **AVL trees** (**Adelson-Velskii**, **Landis**)
  - Height is still O(log N)

Balanced search trees



Height-balanced and size-balanced



Height-balanced not size-balanced

Height balanced trees

- For a height-balanced tree of size N, the height is at most $2 \log N$
- Let $S(h)$ be the size of the smallest height-balanced tree of height h
- **Claim**: For $h >= 1$, $S(h) >= 2^{h/2}$
- $S(1) = 1 = 2^{1/2}$
- $S(2) = 2 = 2^{2/2}$

Height balanced trees

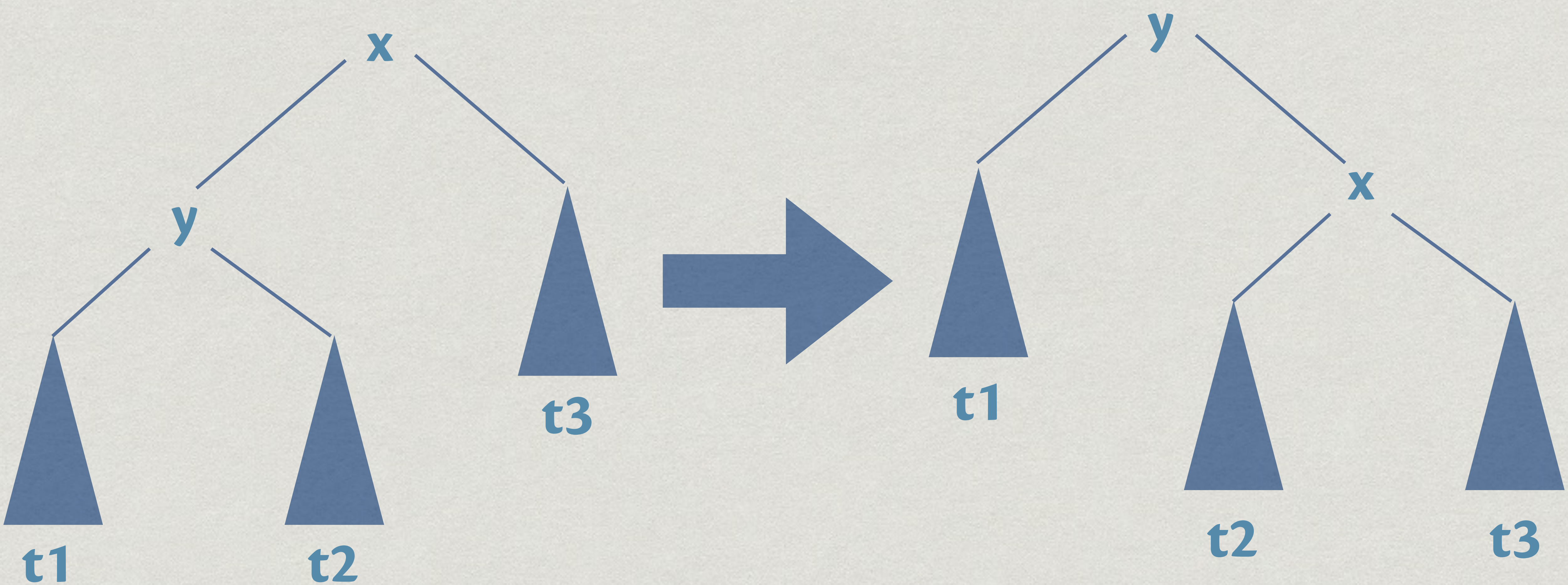
- **Claim**: For $h >= 1$, $S(h) >= 2^{h/2}$
- If a tree has height h, then one of the subtrees is of height $h-1$ and the other has height at least $h-2$
- $S(h) = 1 + S(h-1) + S(h-2) >= S(h-2) + S(h-2)$

$$= 2^{(h-2)/2} + 2^{(h-2)/2}$$

$$= 2^{(h-2)/2+1} = 2^{h/2}$$

- A height-balanced tree with N nodes has height at most $2 \log N$

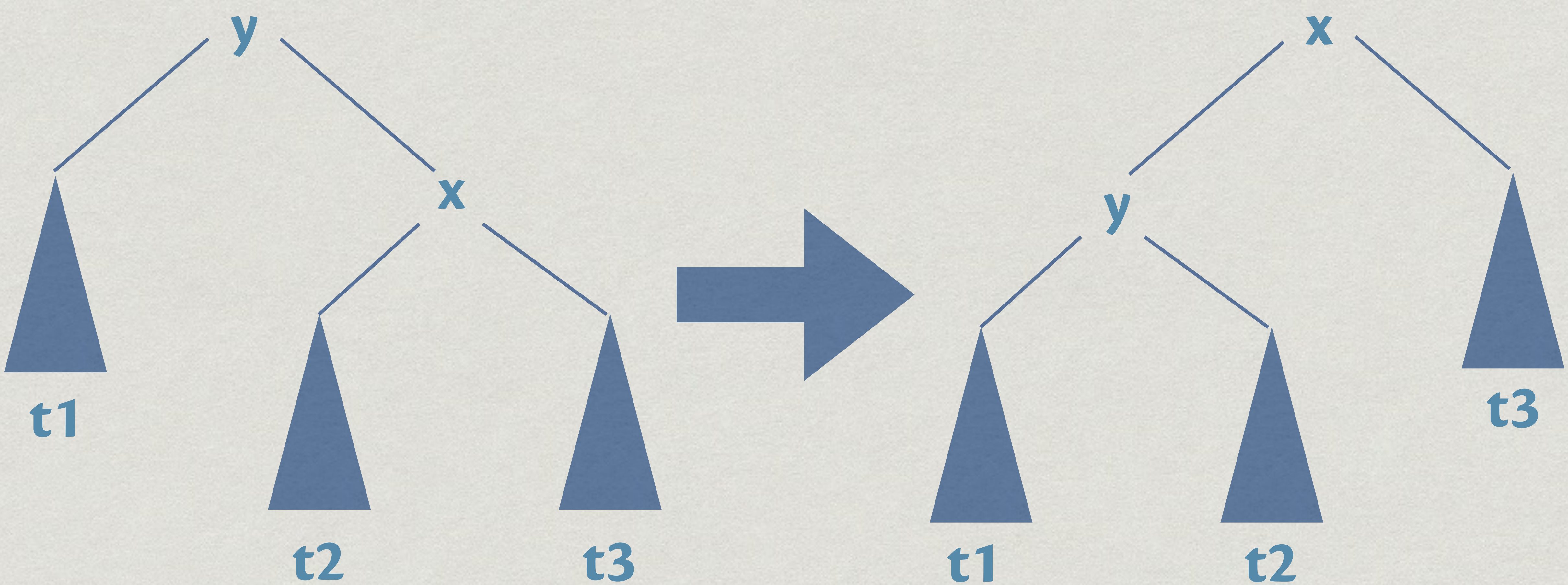
Tree rotations: rotate right



Useful when t1 has large height

```
rotateright (Node (Node t1 y t2) x t3) = Node t1 y (Node t2 x t3)
```

Tree rotations: rotate left



Useful when t3 has large height

```
rotateleft (Node t1 y (Node t2 x t3)) = Node (Node t1 y t2) x t3
```

Height balanced trees

- Assume tree is currently balanced
- Each insert or delete creates an imbalance
- Fix imbalance using a **rebalance** function
- We need to compute height of a tree (and subtrees) to check for imbalance

Height balanced trees

- We need to compute height of a tree (and subtrees) to check for imbalance

- ```
  height Nil                = 0
  height (Node tl x tr) =
                   1 + max (height tl) (height tr)
  ```

- This takes O(N) time

- Save effort by storing height at each node
  ```

AVL trees

- ```
  data AVLTree a = Nil
      | Node (AVLTree a) a Int (AVLTree a)
  ```
- ```
  height :: AVLTree a -> Int
  height Nil                = 0
  height (Node tl x h tr) = h
  ```
- We also need a measure of how skewed a tree is: its **slope**
- ```
  slope :: AVLTree a -> Int
  slope Nil                  = 0
  slope (Node tl x h tr)   = height tl - height tr
  ```
  ```

AVL trees: rotates

- Since we store the height at each node, we need to adjust it after each operation

- ```
  rotateright :: AVLTree a -> AVLTree a
  rotateright (Node (Node tll y hl tlr) x h tr) =
              Node tll y nh (Node tlr x nhr tr)
    where
      nhr     = 1 + max (height tlr) (height tr)
      nh      = 1 + max (height tll) nhr
  ```

- Constant time operation
  ```

AVL trees: rotates

- Since we store the height at each node, we need to adjust it after each operation

- ```
  rotateleft :: AVLTree a -> AVLTree a
  rotateleft (Node tl y h (Node trl x hr trr)) =
             Node (Node tl y nhl trl) x nh trr
    where
      nhl     = 1 + max (height tl) (height trl)
      nh      = 1 + max nhl (height trr)
  ```

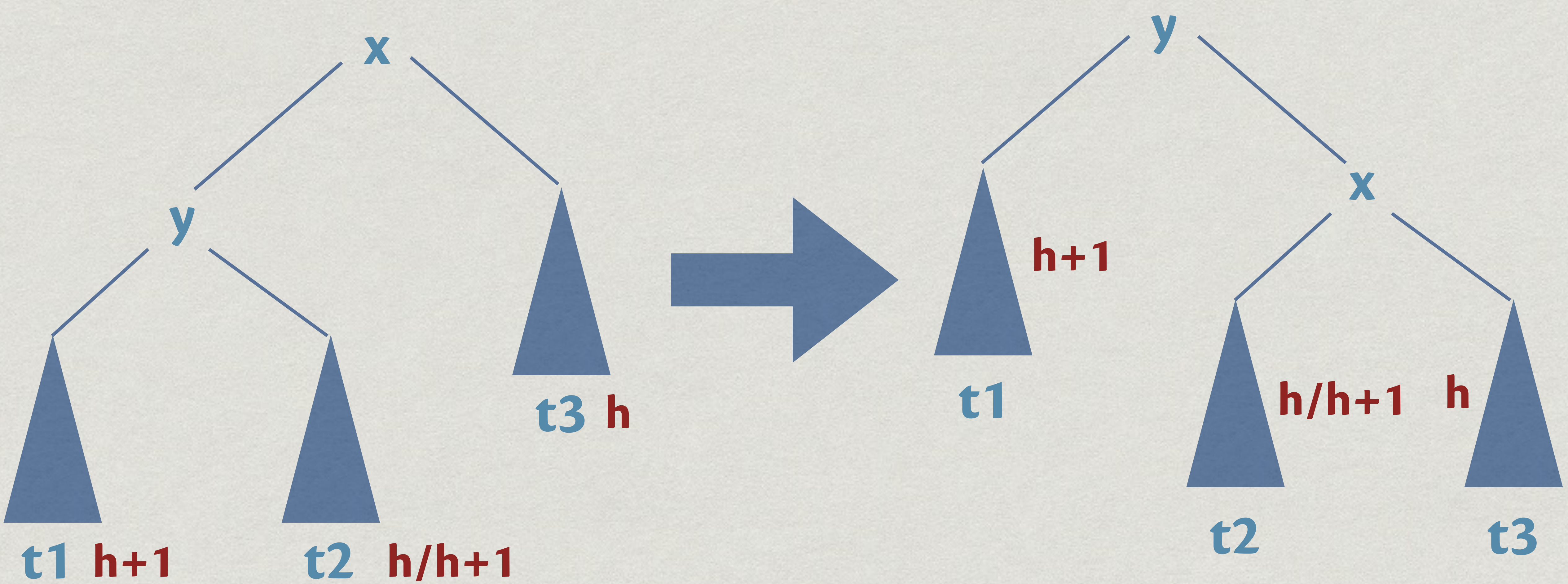
- Constant time operation
  ```

Rebalancing trees

- Recall:
  `slope (Node tl x h tr) = height tl - height tr`
- In a height balanced tree, slope is -1, 0, or 1
- After an insert or delete, slope can be -2, -1, 0, 1, or 2
- Violations happen only at nodes visited by operation
- We rebalance each node on the path visited by operation

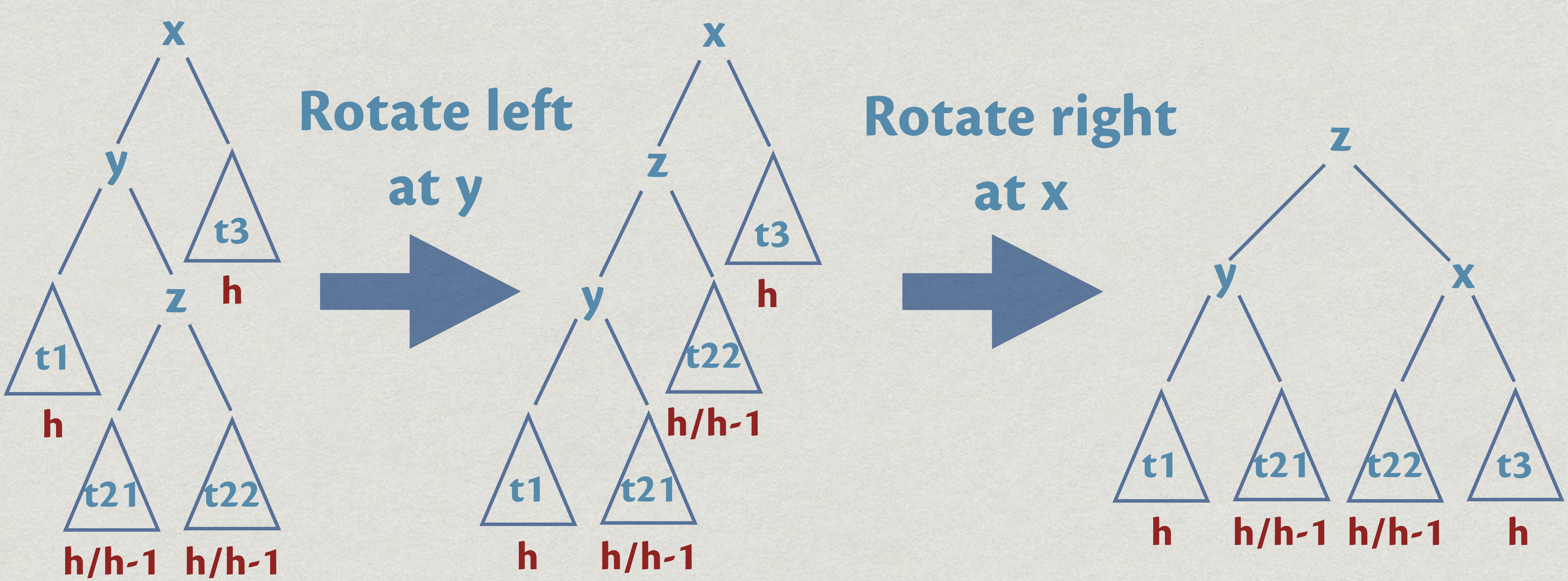
Rebalancing: slope = 2

- Assume slope of a tree is 2 and both subtrees are balanced
- **Case 1**: slope of left subtree is 0 or 1. **Rotate right**



Rebalancing: slope = 2

- Assume slope of a tree is 2 and both subtrees are balanced
- **Case 2**: slope of left subtree is -1. **Rotate left and rotate right**



Rebalancing: slope = -2

- Symmetric to the slope = 2 case
- Two subcases:
  - slope of right subtree is 0 or -1
  - slope of right subtree is 1
- Handled symmetrically

The rebalance function

- ```
  rebalance :: AVLTree a -> AVLTree a
  rebalance (Node tl x h tr)
      | abs (st) < 2              = Node tl x h tr
      | st == 2 && stl /= -1      = rotateright (Node tl x h tr)
      | st == 2 && stl == -1      = rotateright (Node
                                         (rotateleft tl) x h tr)
      | st == -2 && str /= 1      = rotateleft (Node tl x h tr)
      | st == -2 && str == 1      = rotateleft (Node tl x h
                                                (rotateright tr))
      where
          st                   = slope (Node tl x h tr)
          stl                  = slope tl
          str                  = slope tr
  ```
- Constant time operation
  ```

Searching in a tree

- ```
  search :: Ord a=> AVLTree a -> a -> Bool
  search Nil v                      = False
  search (Node tl x h tr) v
      | x == v                      = True
      | v < x                       = search tl v
      | otherwise                   = search tr v
  ```

- Time taken: proportional to **height** (= 2 log N)
  ```

Inserting in a tree

- ```
  insert :: Ord a => AVLTree a -> a -> AVLTree a
  insert Nil v              = Node Nil v 1 Nil
  insert (Node tl x h tr) v
    | x == v                = Node tl x h tr
    | v < x                 = rebalance (Node ntl x nhl tr)
    | otherwise             = rebalance (Node tl x nhr ntr)
    where
       ntl                  = insert tl v
       ntr                  = insert tr v
       nhl                  = 1 + max (height ntl) (height tr)
       nhr                  = 1 + max (height tl) (height ntr)
  ```

- Time taken: proportional to **height** (= 2 log N)
  ```

Deleting from a tree

- ```
  delete :: Ord a => AVLTree a -> a -> AVLTree a
  delete Nil v             = Nil
  delete (Node tl x h tr) v
    | v < x                = rebalance (Node ntl x nhl tr)
    | v > x                = rebalance (Node tl x nhr ntr)
    | otherwise            = if (tl == Nil) then tr else
                                  rebalance (Node ty y hyr tr)
    where
      (y, ty)              = deletemax tl
      ntl                  = delete tl v
      ntr                  = delete tr v
      nhl                  = 1 + max (height ntl) (height tr)
      nhr                  = 1 + max (height tl) (height ntr)
      hyr                  = 1 + max (height ty) (height tr)
  ```
- Time taken: proportional to **height** (= 2 log N), assuming `deletemax` behaves well
  ```

deletemax

- ```
  deletemax :: AVLTree a -> (a, AVLTree a)
  deletemax (Node tl x h Nil)  = (x, tl)
                            -- At the rightmost node
  deletemax (Node tl x h tr)   =
                    (y, rebalance (Node tl x nh ty))

                           -- Always descend right
      where
          (y, ty) = deletemax tr
          nh      = 1 + max (height tl) (height ty)
  ```

- Time taken: proportional to **height** (= 2 log N)
  ```

Summary

- Each operation (`insert`, `delete`, `search`) on an AVL tree takes O(log N) time
- A sequence of N operations takes O(N log N) time
- Fundamental, but non-trivial data structure
- Excellent example of the power of Haskell
- Mathematical definitions transcribed almost directly to code