Programming in Haskell: Lectures 23 & 24

S P Suresh

November 4 & 6, 2019

Balance

- The complexity of the key operations on trees depends on the **height** of the tree

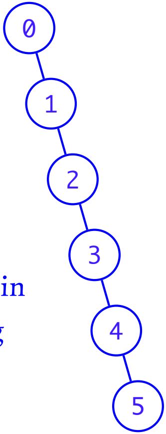
Balance

- The complexity of the key operations on trees depends on the **height** of the tree
- In general, a tree might not be balanced

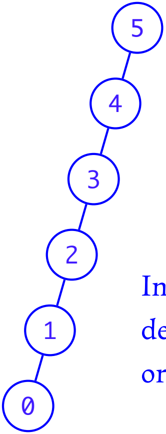
Balance

- 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



Inserting in ascending order



Inserting in descending order

Balanced search trees

- Ideally, for each node, the left and right subtrees differ in size by at most 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

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 $1 = \log 1 + 1$

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 $n/2$

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

Balanced search trees

- Not easy to maintain size balance
- Maintain height balance instead

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

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

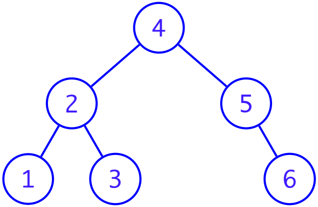
Balanced search trees

- 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)

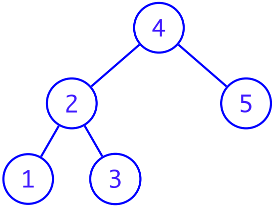
Balanced search trees

- 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 $2 \log n$

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

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 $h \geq 1$, $S(h) \geq 2^{h/2}$

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 $h \geq 1$, $S(h) \geq 2^{h/2}$
  - $S(1) = 1 = 2^{1/2}$ and $S(2) = 2 = 2^{2/2}$

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 $h \geq 1$, $S(h) \geq 2^{h/2}$
  - $S(1) = 1 = 2^{1/2}$ and $S(2) = 2 = 2^{2/2}$
  - If a tree has height $h > 2$

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 $h \geq 1$, $S(h) \geq 2^{h/2}$
  - $S(1) = 1 = 2^{1/2}$ and $S(2) = 2 = 2^{2/2}$
  - If a tree has height $h > 2$
    - one subtree has height $h-1$

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 $h \geq 1$, $S(h) \geq 2^{h/2}$
  - $S(1) = 1 = 2^{1/2}$ and $S(2) = 2 = 2^{2/2}$
  - If a tree has height $h > 2$
    - one subtree has height $h-1$
    - other subtree has height at least $h-2$

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 $h \geq 1$, $S(h) \geq 2^{h/2}$
  - $S(1) = 1 = 2^{1/2}$ and $S(2) = 2 = 2^{2/2}$
  - If a tree has height $h > 2$
    - one subtree has height $h-1$
    - other subtree has height at least $h-2$
  - $S(h) = 1 + S(h-1) + S(h-2) \geq S(h-2) + S(h-2)$

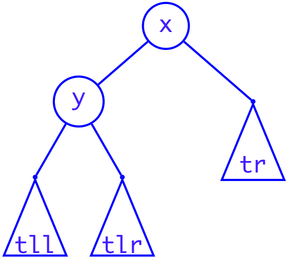
Height-balanced trees

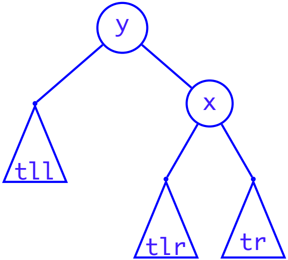
- For a height-balanced tree of size n, the height is at most $2 \log n$
- Let $S(h)$ be the size of the smallest height-balanced tree of height h
- **Claim**: For $h \geq 1$, $S(h) \geq 2^{h/2}$
  - $S(1) = 1 = 2^{1/2}$ and $S(2) = 2 = 2^{2/2}$
  - If a tree has height $h > 2$
    - one subtree has height $h - 1$
    - other subtree has height at least $h - 2$
  - $S(h) = 1 + S(h-1) + S(h-2) \geq S(h-2) + S(h-2)$
  - $S(h) \geq 2^{(h-2)/2} + 2^{(h-2)/2} = 2^{(h-2)/2+1} = 2^{h/2}$

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 $h \geq 1$, $S(h) \geq 2^{h/2}$
  - $S(1) = 1 = 2^{1/2}$ and $S(2) = 2 = 2^{2/2}$
  - If a tree has height $h > 2$
    - one subtree has height $h-1$
    - other subtree has height at least $h-2$
  - $S(h) = 1 + S(h-1) + S(h-2) \geq S(h-2) + S(h-2)$
  - $S(h) \geq 2^{(h-2)/2} + 2^{(h-2)/2} = 2^{(h-2)/2+1} = 2^{h/2}$
- A height-balanced tree of size n has height at most $2\log n$

Tree rotations – rotate right

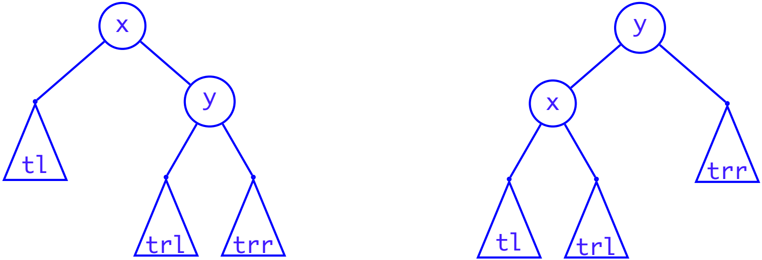




- Useful when `tll` has large height
- In Haskell:

```
rotateRight (Node x (Node y tll tlr) tr)
                      = Node y tll (Node x tlr tr)
```

Tree rotations – rotate left



- Useful when `trr` has large height
- In Haskell:

```
rotateLeft (Node x tl (Node y trl trr))
                      = Node y (Node x tl trl) trr
```

Height-balanced trees

- Assume tree is currently balanced

Height-balanced trees

- Assume tree is currently balanced
- Each insert or delete creates an imbalance

Height-balanced trees

- Assume tree is currently balanced
- Each insert or delete creates an imbalance
- Fix imbalance using a `rebalance` function

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

- 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
- Takes time $O(n)$

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
- Takes time $O(n)$
- Save time by storing height in the node

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
- Takes time $O(n)$
- Save time by storing height in the node
- Need to update height after each operation

AVL trees

- The data type in Haskell:

```
data AVL a = Nil | Node Int a (AVL a) (AVL a)
    deriving (Eq, Ord)
```

AVL trees

- The data type in Haskell:

```haskell
data AVL a = Nil | Node Int a (AVL a) (AVL a)
    deriving (Eq, Ord)
```

- Extracting the height of a tree:

```haskell
height :: AVL a -> Int
height Nil            = 0
height (Node h _ _ _) = h
```

AVL trees

- The data type in Haskell:

```haskell
data AVL a = Nil | Node Int a (AVL a) (AVL a)
    deriving (Eq, Ord)
```

- Extracting the height of a tree:

```haskell
height :: AVL a -> Int
height Nil            = 0
height (Node h _ _ _) = h
```

- Also need a measure of how skewed a tree is – its slope

```haskell
slope :: AVL a -> Int
slope Nil              = 0
slope (Node _ _ tl tr) = height tl - height tr
```

- Check if `t` is an AVL tree:

```
isAVL :: Ord a => AVL a -> Bool
isAVL Nil        = True
isAVL t@(Node _ x tl tr)
                 = abs (slope t) < 2 &&
                     isAVL tl && isAVL tr &&
                     (isEmpty tl || maxt tl < x) &&
                     (isEmpty tr || x < mint tr)
```

AVL trees – rotates

- Since we maintain height at each node, we need to adjust it after each operation:

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

AVL trees – rotates

- Since we maintain height at each node, we need to adjust it after each operation:

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

- Constant time operation

AVL trees – rotates

- Since we maintain height at each node, we need to adjust it after each operation:

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

AVL trees – rotates

- Since we maintain height at each node, we need to adjust it after each operation:

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

- Constant time operation

Rebalancing AVL trees

- Recall:

```
slope (Node h x tl tr) = height tl - height tr
```

Rebalancing AVL trees

- Recall:

```
slope (Node h x tl tr) = height tl - height tr
```

- In a height balanced tree, `abs slope < 2`

Rebalancing AVL trees

- Recall:

```
slope (Node h x tl tr) = height tl - height tr
```

- In a height balanced tree, `abs slope < 2`
- After an insert or delete, it can happen that `abs slope == 2`

Rebalancing AVL trees

- Recall:

```
slope (Node h x tl tr) = height tl - height tr
```

- In a height balanced tree, `abs slope < 2`
- After an insert or delete, it can happen that `abs slope == 2`
- Violations happen only at nodes visited by operation

Rebalancing AVL trees

- Recall:

```
slope (Node h x tl tr) = height tl - height tr
```

- In a height balanced tree, `abs slope < 2`
- After an insert or delete, it can happen that `abs slope == 2`
- Violations happen only at nodes visited by operation
- We rebalance each node on the path visited by operation

Rebalancing – `slope == 2`

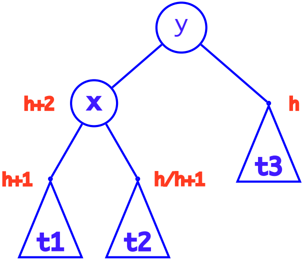
- Assume `slope == 2` and both subtrees are balanced

Rebalancing – `slope == 2`

- Assume `slope == 2` and both subtrees are balanced
- Slope of left subtree is `0` or `1` – **rotate right**

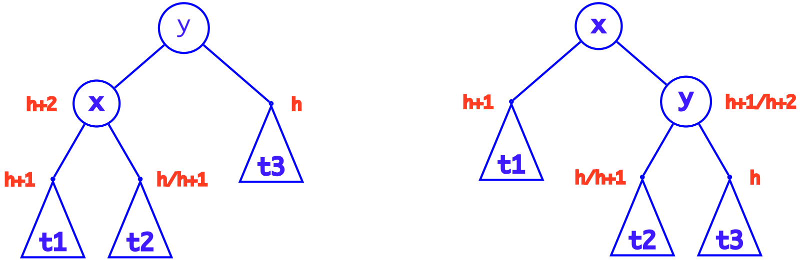
Rebalancing – `slope == 2`

- Assume `slope == 2` and both subtrees are balanced
- Slope of left subtree is `0` or `1` – rotate right



Rebalancing – `slope == 2`

- Assume `slope == 2` and both subtrees are balanced
- Slope of left subtree is `0` or `1` – rotate right



Rebalancing – `slope == 2`

- Assume `slope == 2` and both subtrees are balanced

Rebalancing – `slope == 2`

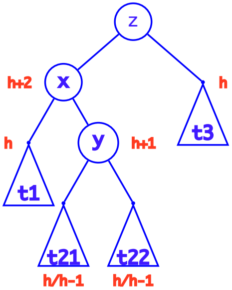
- Assume `slope == 2` and both subtrees are balanced
- Slope of left subtree is `-1`

Rebalancing – `slope == 2`

- Assume `slope == 2` and both subtrees are balanced
- Slope of left subtree is `-1`
- **Left rotate the left subtree and then right rotate the tree**

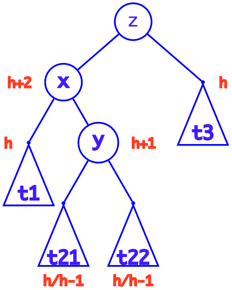
Rebalancing – `slope == 2`

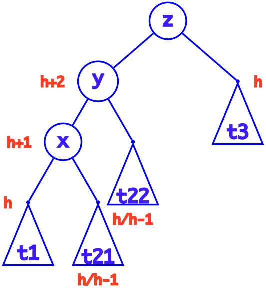
- Assume `slope == 2` and both subtrees are balanced
- Slope of left subtree is `-1`
- Left rotate the left subtree and then right rotate the tree



Rebalancing – `slope == 2`

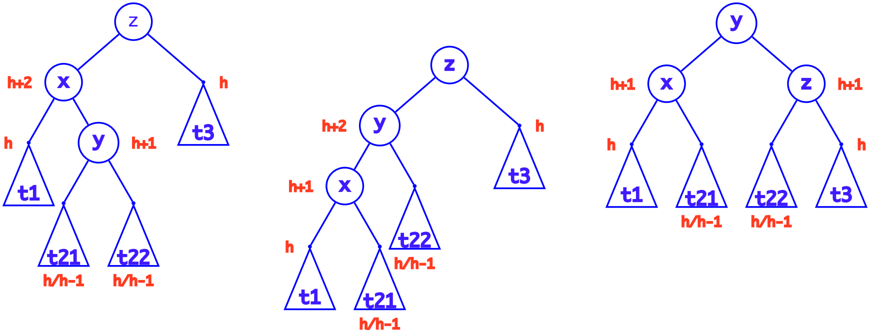
- Assume `slope == 2` and both subtrees are balanced
- Slope of left subtree is `-1`
- Left rotate the left subtree and then right rotate the tree





Rebalancing – `slope == 2`

- Assume `slope == 2` and both subtrees are balanced
- Slope of left subtree is `-1`
- Left rotate the left subtree and then right rotate the tree



Rebalancing – `slope == -2`

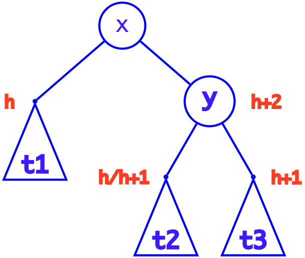
- Assume `slope == -2` and both subtrees are balanced

Rebalancing – `slope == -2`

- Assume `slope == -2` and both subtrees are balanced
- Slope of right subtree is `0` or `-1` – **rotate left**

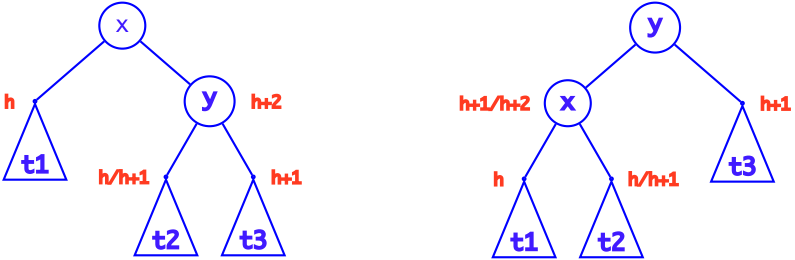
Rebalancing – `slope == -2`

- Assume `slope == -2` and both subtrees are balanced
- Slope of right subtree is `0` or `-1` – rotate left



Rebalancing – `slope == -2`

- Assume `slope == -2` and both subtrees are balanced
- Slope of right subtree is `0` or `-1` – rotate left



Rebalancing – `slope == -2`

- Assume `slope == -2` and both subtrees are balanced

Rebalancing – `slope == -2`

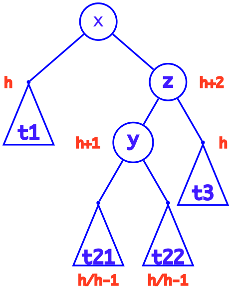
- Assume `slope == -2` and both subtrees are balanced
- Slope of right subtree is `1`

Rebalancing – `slope == -2`

- Assume `slope == -2` and both subtrees are balanced
- Slope of right subtree is `1`
- Right rotate the right subtree and then left rotate the tree

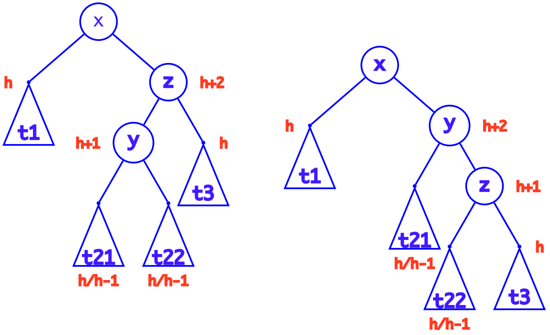
Rebalancing – `slope == -2`

- Assume `slope == -2` and both subtrees are balanced
- Slope of right subtree is `1`
- Right rotate the right subtree and then left rotate the tree



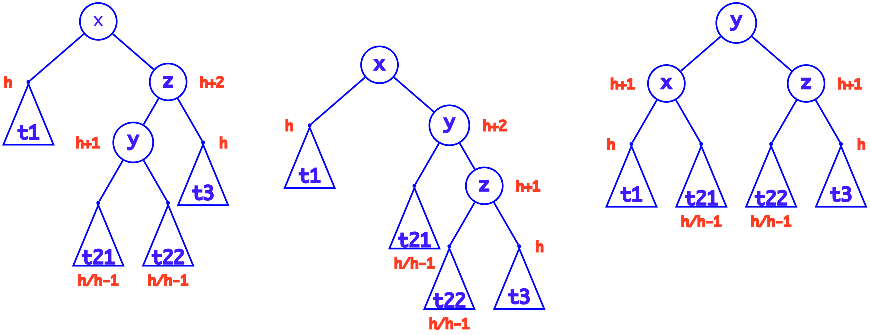
Rebalancing – `slope == -2`

- Assume `slope == -2` and both subtrees are balanced
- Slope of right subtree is `1`
- Right rotate the right subtree and then left rotate the tree



Rebalancing – `slope == -2`

- Assume `slope == -2` and both subtrees are balanced
- Slope of right subtree is `1`
- Right rotate the right subtree and then left rotate the tree



Rebalancing in Haskell

- The `rebalance` function

```haskell
rebalance :: Ord a => AVL a -> AVL a
rebalance t@(Node h x tl tr)
    | abs st < 2 = t
    | st == 2 = if stl == -1 then
                rotateRight (Node h x (rotateLeft tl) tr)
                else rotateRight t
    | st == -2 = if str == 1 then
                 rotateLeft (Node h x tl (rotateRight tr))
                 else rotateLeft t
    where (st, stl, str) = (slope t, slope tl, slope tr)
```

Rebalancing in Haskell

- The `rebalance` function

```
rebalance :: Ord a => AVL a -> AVL a
rebalance t@(Node h x tl tr)
    | abs st < 2 = t
    | st == 2 = if stl == -1 then
                rotateRight (Node h x (rotateLeft tl) tr)
                else rotateRight t
    | st == -2 = if str == 1 then
                 rotateLeft (Node h x tl (rotateRight tr))
                 else rotateLeft t
    where (st, stl, str) = (slope t, slope tl, slope tr)
```

- Constant time operation

Searching in an AVL tree

```
searchAVL :: Ord a => a -> AVL a -> Bool
searchAVL v Nil = False
searchAVL v (Node _ x tl tr)
    | v == x = True
    | v < x  = searchAVL v tl
    | v > x  = searchAVL v tr
```

Inserting in a tree

```haskell
insertAVL :: Ord a => a -> AVL a -> AVL a
insertAVL v Nil = Node 1 v Nil Nil
insertAVL v t@(Node h x tl tr)
    | v < x  = rebalance (Node nhl x ntl tr)
    | v > x  = rebalance (Node nhr x tl ntr)
    | v == x = t
    where
        ntl = insertAVL v tl
        ntr = insertAVL v tr
        nhl = 1 + max (height ntl) (height tr)
        nhr = 1 + max (height tl) (height ntr)
```

Deleting the maximum element

```
deleteMax :: Ord a => AVL a -> (a, AVL a)
deleteMax (Node _ x tl Nil) = (x, tl)
deleteMax (Node h x tl tr)  = (y, rebalance (Node nh x tl ty))
    where
        (y, ty) = deleteMax tr
        nh      = 1 + max (height tl) (height ty)
```

Deleting from a tree

```
deleteAVL :: Ord a => a -> AVL a -> AVL a
deleteAVL v Nil = Nil
deleteAVL v t@(Node h x tl tr)
    | v < x     = rebalance (Node nhl x ntl tr)
    | v > x     = rebalance (Node nhr x tl ntr)
    | v == x    = if isEmpty tl then tr
                  else rebalance (Node nhy y ty tr)
  where
    (y, ty)     = deleteMax tl
    (ntl, ntr) = (deleteAVL v tl, deleteAVL v tr)
    nhl         = 1 + max (height ntl) (height tr)
    nhr         = 1 + max (height tl) (height ntr)
    nhy         = 1 + max (height ty) (height tr)
```

Tree operations – complexity

- Left and right rotates take constant time

Tree operations – complexity

- Left and right rotates take constant time
- Rebalance of a tree takes constant time, when both subtrees are balanced

Tree operations – complexity

- Left and right rotates take constant time
- Rebalance of a tree takes constant time, when both subtrees are balanced
- Search takes time proportional to height of the tree

Tree operations – complexity

- Left and right rotates take constant time
- Rebalance of a tree takes constant time, when both subtrees are balanced
- Search takes time proportional to height of the tree
- Insert and delete spend constant time on each node of some maximal path

Tree operations – complexity

- Left and right rotates take constant time
- Rebalance of a tree takes constant time, when both subtrees are balanced
- Search takes time proportional to height of the tree
- Insert and delete spend constant time on each node of some maximal path
- Take time proportional to height of the tree

Tree operations – complexity

- Left and right rotates take constant time
- Rebalance of a tree takes constant time, when both subtrees are balanced
- Search takes time proportional to height of the tree
- Insert and delete spend constant time on each node of some maximal path
- Take time proportional to height of the tree
- Height of a tree with n nodes is $\leq 2\log n$

Tree operations – complexity

- Left and right rotates take constant time
- Rebalance of a tree takes constant time, when both subtrees are balanced
- Search takes time proportional to height of the tree
- Insert and delete spend constant time on each node of some maximal path
- Take time proportional to height of the tree
- Height of a tree with n nodes is $\leq 2 \log n$
- Thus each operation takes $O(\log n)$ time

Tree operations – complexity

- Left and right rotates take constant time
- Rebalance of a tree takes constant time, when both subtrees are balanced
- Search takes time proportional to height of the tree
- Insert and delete spend constant time on each node of some maximal path
- Take time proportional to height of the tree
- Height of a tree with n nodes is $\leq 2\log n$
- Thus each operation takes $O(\log n)$ time
- A sequence of n operations take at most $O(n \log n)$ time

Other useful functions

- Create an empty AVL tree:

```
emptyAVL :: AVL a
emptyAVL = Nil
```

Other useful functions

- Create an empty AVL tree:

```
emptyAVL :: AVL a
emptyAVL = Nil
```

- Check if a tree is empty:

```
isEmpty :: AVL a -> Bool
isEmpty Nil = True
isEmpty _   = False
```

Other useful functions

- Create an empty AVL tree:

```
emptyAVL :: AVL a
emptyAVL = Nil
```

- Check if a tree is empty:

```
isEmpty :: AVL a -> Bool
isEmpty Nil = True
isEmpty _   = False
```

- Create an AVL tree from a list ($O(n \log n)$ time):

```
createAVL :: Ord a => [a] -> AVL a
createAVL = foldl' (flip insertAVL) emptyAVL
```

Other useful functions

- Create a sorted list from an AVL tree:

```
inorder :: Ord a => AVL a -> [a]
inorder Nil                = []
inorder (Node _ x tl tr) = inorder tl ++ [x] ++ inorder tr
```

Other useful functions

- Create a sorted list from an AVL tree:

```
inorder :: Ord a => AVL a -> [a]
inorder Nil                = []
inorder (Node _ x tl tr) = inorder tl ++ [x] ++ inorder tr
```

- $T(n) = 2T(n/2) + O(n)$, so $T(n) = O(n \log n)$

Other useful functions

- Create a sorted list from an AVL tree:

```
inorder :: Ord a => AVL a -> [a]
inorder Nil                = []
inorder (Node _ x tl tr) = inorder tl ++ [x] ++ inorder tr
```

- $T(n) = 2T(n/2) + O(n)$, so $T(n) = O(n \log n)$
- Culprit is `++`, which takes $O(n)$ time

Other useful functions

- Smarter `inorder`:

```
inorder :: Ord a => AVL a -> [a]
inorder t                       = go t []
    where go Nil l              = l
          go (Node _ x tl tr) l = go tl (x:go tr l)
```

Other useful functions

- Smarter `inorder`:

```
inorder :: Ord a => AVL a -> [a]
inorder t                        = go t []
    where go Nil l               = l
          go (Node _ x tl tr) l = go tl (x:go tr l)
```

- $T(n) = 2T(n/2) + c$, so $T(n) = O(n)$

Other useful functions

- Smarter `inorder`:

```
inorder :: Ord a => AVL a -> [a]
inorder t                        = go t []
    where go Nil l               = l
          go (Node _ x tl tr) l = go tl (x:go tr l)
```

- $T(n) = 2T(n/2) + c$, so $T(n) = O(n)$
- We can sort a list in $O(n \log n)$ time by:

```
treesort :: Ord a => [a] -> [a]
treesort = inorder . createAVL
```

Other useful functions

- `inorder (createAVL l)` sorts list `l`

Other useful functions

- `inorder (createAVL l)` sorts list `l`
- What if we wanted a size-balanced tree `t` such that `inorder t == l`?

Other useful functions

- `inorder (createAVL l)` sorts list `l`
- What if we wanted a size-balanced tree `t` such that `inorder t == l`?
- `t` will not be a search tree in general

Other useful functions

- `inorder (createAVL l)` sorts list `l`
- What if we wanted a size-balanced tree `t` such that `inorder t == l`?
- `t` will not be a search tree in general
- If `l` itself is sorted, `t` is search tree

Other useful functions

- `inorder (createAVL l)` sorts list `l`
- What if we wanted a size-balanced tree `t` such that `inorder t == l`?
- `t` will not be a search tree in general
- If `l` itself is sorted, `t` is search tree
- This is just the smart `createTree` we saw in a previous class

Other useful functions

- `inorder (createAVL l)` sorts list `l`
- What if we wanted a size-balanced tree `t` such that `inorder t == l`?
- `t` will not be a search tree in general
- If `l` itself is sorted, `t` is search tree
- This is just the smart `createTree` we saw in a previous class
- Works in $O(n)$ time

inorderTree

```haskell
inorderTree :: [a] -> AVL a
inorderTree l = fst (go (length l) l)
    where
        go :: Int -> [a] -> (AVL a, [a])
        go 0 xs = (Nil, xs)
        go n xs = (Node h y tl tr, zs)
            where
                m = n `div` 2
                (tl, y:ys) = go m xs
                (tr, zs) = go (n-m-1) ys
                h = 1 + max (height tl) (height tr)
```

A module for AVL trees

- Saved in `AVL.hs`

```
module AVL(AVL, emptyAVL, isEmpty, isAVL,
           insertAVL, deleteAVL, searchAVL,
           createAVL, inorder, inorderTree) where

data AVL a = Nil | Node Int a (AVL a) (AVL a)
    deriving (Eq, Ord)

instance Show a => Show (AVL a) where
    show t = intercalate "\n" (draw t)
```

A module for AVL trees

- Saved in `AVL.hs`

```
module AVL(AVL, emptyAVL, isEmpty, isAVL,
            insertAVL, deleteAVL, searchAVL,
            createAVL, inorder, inorderTree) where

data AVL a = Nil | Node Int a (AVL a) (AVL a)
    deriving (Eq, Ord)

instance Show a => Show (AVL a) where
    show t = intercalate "\n" (draw t)
```

- Can be used to define the `Set` ADT

The Set ADT again

```haskell
module Set(Set, emptySet, createSet,
           insertInto, deleteFrom, search,
           union, intersect, diff) where
import AVL

data Set a = Set (AVL a)
instance (Ord a, Show a) => Show (Set a) where
    show (Set t) = show (inorder t)

emptySet :: Ord a => Set a
emptySet = Set emptyAVL
```

The Set ADT again

```haskell
createSet :: Ord a => [a] -> Set a
createSet = Set . createAVL

search :: Ord a => a -> Set a -> Bool
search x (Set t) = searchAVL x t

insertInto :: Ord a => a -> Set a -> Set a
insertInto x (Set t) = Set (insertAVL x t)

deleteFrom :: Ord a => a -> Set a -> Set a
deleteFrom x (Set t) = Set (deleteAVL x t)
```

More set operations

```haskell
union :: Ord a => Set a -> Set a -> Set a
union (Set t1) (Set t2) = Set $ inorderTree $
                unionMerge (inorder t1) (inorder t2)


unionMerge :: Ord a => [a] -> [a] -> [a]
unionMerge [] ys = ys
unionMerge xs [] = xs
unionMerge (x:xs) (y:ys)
    | x < y  = x:unionMerge xs (y:ys)
    | y < x  = y:unionMerge (x:xs) ys
    | x == y = x:unionMerge xs ys
```

More set operations

```haskell
intersect :: Ord a => Set a -> Set a -> Set a
intersect (Set t1) (Set t2) = Set $ inorderTree $
                   intersectMerge (inorder t1) (inorder t2)

intersectMerge :: Ord a => [a] -> [a] -> [a]
intersectMerge [] ys = []
intersectMerge xs [] = []
intersectMerge (x:xs) (y:ys)
    | x < y  = intersectMerge xs (y:ys)
    | y < x  = intersectMerge (x:xs) ys
    | x == y = x:intersectMerge xs ys
```

More set operations

```
diff :: Ord a => Set a -> Set a -> Set a
diff (Set t1) (Set t2) = Set $ inorderTree $
              diffMerge (inorder t1) (inorder t2)

diffMerge :: Ord a => [a] -> [a] -> [a]
diffMerge [] ys = []
diffMerge xs [] = xs
diffMerge (x:xs) (y:ys)
    | x < y  = x:diffMerge xs (y:ys)
    | y < x  = diffMerge (x:xs) ys
    | x == y = diffMerge xs ys
```

Summary

- AVL trees are a fundamental, but non-trivial data structures

Summary

- AVL trees are a fundamental, but non-trivial data structures
- Allows us to efficiently implement the `Set` ADT

Summary

- AVL trees are a fundamental, but non-trivial data structures
- Allows us to efficiently implement the `Set` ADT
  - `search`, `insertInto` and `deleteFrom` in $O(\log n)$ time

Summary

- AVL trees are a fundamental, but non-trivial data structures
- Allows us to efficiently implement the `Set` ADT
  - `search`, `insertInto` and `deleteFrom` in $O(\log n)$ time
  - `union`, `intersect` and `diff` in $O(m+n)$ time

Summary

- AVL trees are a fundamental, but non-trivial data structures
- Allows us to efficiently implement the `Set` ADT
    - `search`, `insertInto` and `deleteFrom` in $O(\log n)$ time
    - `union`, `intersect` and `diff` in $O(m+n)$ time
- An illustration of the power of Haskell

Summary

- AVL trees are a fundamental, but non-trivial data structures
- Allows us to efficiently implement the `Set` ADT
    - `search`, `insertInto` and `deleteFrom` in $O(\log n)$ time
    - `union`, `intersect` and `diff` in $O(m+n)$ time
- An illustration of the power of Haskell
- Mathematical definitions almost directly transcribed to code

Summary

- AVL trees are a fundamental, but non-trivial data structures
- Allows us to efficiently implement the `Set` ADT
  - `search`, `insertInto` and `deleteFrom` in $O(\log n)$ time
  - `union`, `intersect` and `diff` in $O(m+n)$ time
- An illustration of the power of Haskell
- Mathematical definitions almost directly transcribed to code
- Pattern matching is very powerful

Summary

- AVL trees are a fundamental, but non-trivial data structures
- Allows us to efficiently implement the `Set` ADT
  - `search`, `insertInto` and `deleteFrom` in $O(\log n)$ time
  - `union`, `intersect` and `diff` in $O(m+n)$ time
- An illustration of the power of Haskell
- Mathematical definitions almost directly transcribed to code
- Pattern matching is very powerful
- Allows us to easily specify complex transformations on data