Introduction to Haskell Programming

Data structures – AVL trees

S P Suresh

Chennai Mathematical Institute

Haskell @ CMI

Complexity of binary search tree operations

- Complexity of tree operations depends on the **height** of the tree

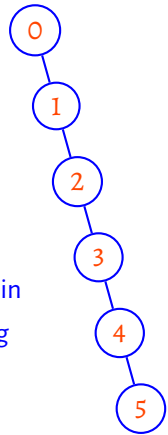
Complexity of binary search tree operations

- Complexity of tree operations depends on the **height** of the tree
- In general, a tree might not be balanced

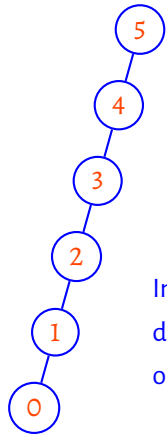
Complexity of binary search tree operations

- Complexity of tree operations 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

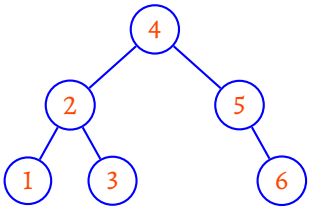
Balanced search trees

- 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)

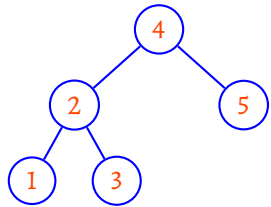
Balanced search trees

- 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 $2 \log n + 1$

Height-balanced trees

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

Height-balanced trees

- For a height-balanced tree of size n, the height is at most $2 \log n + 1$
- Let $S(h)$ be the size of the smallest height-balanced tree of height h
- **Claim**: For $h \geqslant 1$, $S(h) \geqslant 2^{\lfloor h/2 \rfloor}$
  - $S(1) = 1 = 2^0 = 2^{\lfloor 1/2 \rfloor}$ and $S(2) = 2 = 2^1 = 2^{\lfloor 2/2 \rfloor}$

Height-balanced trees

- For a height-balanced tree of size n, the height is at most $2 \log n + 1$
- Let $S(h)$ be the size of the smallest height-balanced tree of height h
- **Claim**: For $h \geqslant 1$, $S(h) \geqslant 2^{\lfloor h/2 \rfloor}$
  - $S(1) = 1 = 2^0 = 2^{\lfloor 1/2 \rfloor}$ and $S(2) = 2 = 2^1 = 2^{\lfloor 2/2 \rfloor}$
  - If a tree has height $h > 2$

Height-balanced trees

- For a height-balanced tree of size n, the height is at most $2 \log n + 1$
- Let $S(h)$ be the size of the smallest height-balanced tree of height h
- **Claim**: For $h \geqslant 1$, $S(h) \geqslant 2^{\lfloor h/2 \rfloor}$
  - $S(1) = 1 = 2^0 = 2^{\lfloor 1/2 \rfloor}$ and $S(2) = 2 = 2^1 = 2^{\lfloor 2/2 \rfloor}$
  - 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 $2 \log n + 1$
- Let $S(h)$ be the size of the smallest height-balanced tree of height h
- **Claim**: For $h \geqslant 1$, $S(h) \geqslant 2^{\lfloor h/2 \rfloor}$
  - $S(1) = 1 = 2^0 = 2^{\lfloor 1/2 \rfloor}$ and $S(2) = 2 = 2^1 = 2^{\lfloor 2/2 \rfloor}$
  - 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 $2 \log n + 1$
- Let $S(h)$ be the size of the smallest height-balanced tree of height h
- **Claim**: For $h \geqslant 1$, $S(h) \geqslant 2^{\lfloor h/2 \rfloor}$
  - $S(1) = 1 = 2^0 = 2^{\lfloor 1/2 \rfloor}$ and $S(2) = 2 = 2^1 = 2^{\lfloor 2/2 \rfloor}$
  - If a tree has height $h > 2$
    - one subtree has height $h - 1$
    - other subtree has height at least $h - 2$
  - $S(h) = 1 + S(h-1) + S(h-2) \geqslant S(h-2) + S(h-2)$

Height-balanced trees

- For a height-balanced tree of size n, the height is at most $2 \log n + 1$
- Let $S(h)$ be the size of the smallest height-balanced tree of height h
- **Claim**: For $h \geqslant 1$, $S(h) \geqslant 2^{\lfloor h/2 \rfloor}$
  - $S(1) = 1 = 2^0 = 2^{\lfloor 1/2 \rfloor}$ and $S(2) = 2 = 2^1 = 2^{\lfloor 2/2 \rfloor}$
  - If a tree has height $h > 2$
    - one subtree has height $h - 1$
    - other subtree has height at least $h - 2$
  - $S(h) = 1 + S(h-1) + S(h-2) \geqslant S(h-2) + S(h-2)$
  - $S(h) \geqslant 2^{\lfloor (h-2)/2 \rfloor} + 2^{\lfloor (h-2)/2 \rfloor} = 2^{\lfloor (h-2)/2 \rfloor + 1} \geqslant 2^{\lfloor h/2 \rfloor}$

Height-balanced trees

- For a height-balanced tree of size n, the height is at most $2 \log n + 1$
- Let $S(h)$ be the size of the smallest height-balanced tree of height h
- **Claim**: For $h \geqslant 1$, $S(h) \geqslant 2^{\lfloor h/2 \rfloor}$
  - $S(1) = 1 = 2^0 = 2^{\lfloor 1/2 \rfloor}$ and $S(2) = 2 = 2^1 = 2^{\lfloor 2/2 \rfloor}$
  - If a tree has height $h > 2$
    - one subtree has height $h - 1$
    - other subtree has height at least $h - 2$
  - $S(h) = 1 + S(h-1) + S(h-2) \geqslant S(h-2) + S(h-2)$
  - $S(h) \geqslant 2^{\lfloor (h-2)/2 \rfloor} + 2^{\lfloor (h-2)/2 \rfloor} = 2^{\lfloor (h-2)/2 \rfloor + 1} \geqslant 2^{\lfloor h/2 \rfloor}$
- A height-balanced tree of size n has height at most $2 \log n + 1$

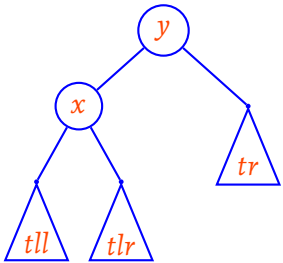
Height-balanced trees

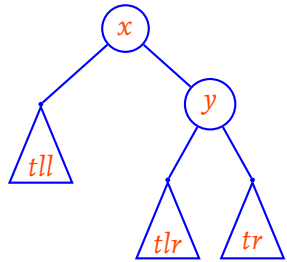
- For a height-balanced tree of size n, the height is at most $2 \log n + 1$
- Let $S(h)$ be the size of the smallest height-balanced tree of height h
- **Claim**: For $h \geqslant 1$, $S(h) \geqslant 2^{\lfloor h/2 \rfloor}$
  - $S(1) = 1 = 2^0 = 2^{\lfloor 1/2 \rfloor}$ and $S(2) = 2 = 2^1 = 2^{\lfloor 2/2 \rfloor}$
  - If a tree has height $h > 2$
    - one subtree has height $h - 1$
    - other subtree has height at least $h - 2$
  - $S(h) = 1 + S(h-1) + S(h-2) \geqslant S(h-2) + S(h-2)$
  - $S(h) \geqslant 2^{\lfloor (h-2)/2 \rfloor} + 2^{\lfloor (h-2)/2 \rfloor} = 2^{\lfloor (h-2)/2 \rfloor + 1} \geqslant 2^{\lfloor h/2 \rfloor}$
- A height-balanced tree of size n has height at most $2 \log n + 1$
  - If its height is h, then $n \geqslant S(h) \geqslant 2^{\lfloor h/2 \rfloor}$

Height-balanced trees

- For a height-balanced tree of size n, the height is at most $2 \log n + 1$
- Let $S(h)$ be the size of the smallest height-balanced tree of height h
- **Claim**: For $h \geqslant 1$, $S(h) \geqslant 2^{\lfloor h/2 \rfloor}$
  - $S(1) = 1 = 2^0 = 2^{\lfloor 1/2 \rfloor}$ and $S(2) = 2 = 2^1 = 2^{\lfloor 2/2 \rfloor}$
  - If a tree has height $h > 2$
    - one subtree has height $h - 1$
    - other subtree has height at least $h - 2$
  - $S(h) = 1 + S(h-1) + S(h-2) \geqslant S(h-2) + S(h-2)$
  - $S(h) \geqslant 2^{\lfloor (h-2)/2 \rfloor} + 2^{\lfloor (h-2)/2 \rfloor} = 2^{\lfloor (h-2)/2 \rfloor + 1} \geqslant 2^{\lfloor h/2 \rfloor}$
- A height-balanced tree of size n has height at most $2 \log n + 1$
  - If its height is h, then $n \geqslant S(h) \geqslant 2^{\lfloor h/2 \rfloor}$
  - So $(h-1)/2 \leqslant \lfloor h/2 \rfloor \leqslant \log n$, and hence $h \leqslant 2 \log n + 1$

Tree rotations – rotate right

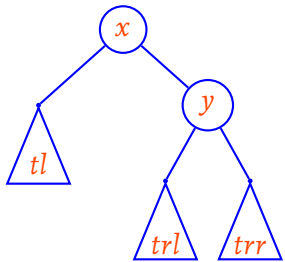


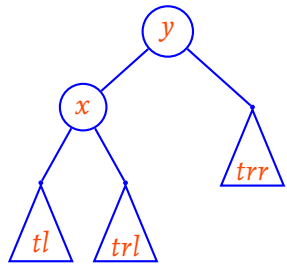


- Useful when *tll* has large height
- In Haskell:

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

Tree rotations – rotate left





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

```
rotateLeft (Node tl x (Node trl y trr))
        = Node (Node tl x trl) y 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 (AVL a) a (AVL a) deriving Eq
```

AVL trees

- The data type in Haskell:

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

- Extracting the height of a tree:

```
height :: AVL a → Int
height Nil            = 0
height (Node h _ _ _) = h
```

AVL trees

- The data type in Haskell:

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

- Extracting the height of a tree:

```
height :: AVL a → Int
height Nil            = 0
height (Node h _ _ _) = h
```

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

```
slope :: AVL a → Int
slope Nil              = 0
slope (Node _ tl _ tr) = height tl - height 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 (Node hl tll x tlr) y tr)
        = Node nh tll x (Node nhr tlr y tr) where
    nh  = 1 + max (height tll) nhr
    nhr = 1 + max (height tlr) (height 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 (Node hl tll x tlr) y tr)
        = Node nh tll x (Node nhr tlr y tr) where
    nh  = 1 + max (height tll) nhr
    nhr = 1 + max (height tlr) (height tr)
```

- **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 tl x (Node hr trl y trr))
        = Node nh (Node nhl tl x trl) y trr where
    nh  = 1 + max nhl (height trr)
    nhl = 1 + max (height tl) (height trl)
```

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 tl x (Node hr trl y trr))
        = Node nh (Node nhl tl x trl) y trr where
    nh  = 1 + max nhl (height trr)
    nhl = 1 + max (height tl) (height trl)
```

- **Constant time** operation

Rebalancing AVL trees

- Recall:

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

Rebalancing AVL trees

- Recall:

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

- Slope of a height balanced tree is either -1 or 0 or 1

Rebalancing AVL trees

- Recall:

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

- Slope of a height balanced tree is either -1 or 0 or 1
- In other words, for a height balanced tree t, $|\text{slope}(t)| < 2$

Rebalancing AVL trees

- Recall:

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

- Slope of a height balanced tree is either -1 or 0 or 1
- In other words, for a height balanced tree t, $|\text{slope}(t)| < 2$
- After an insert or delete, it can happen that $|\text{slope}(t)| = 2$

Rebalancing AVL trees

- Recall:

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

- Slope of a height balanced tree is either -1 or 0 or 1
- In other words, for a height balanced tree t, $|\text{slope}(t)| < 2$
- After an insert or delete, it can happen that $|\text{slope}(t)| = 2$
- Violations happen only at nodes visited by operation

Rebalancing AVL trees

- Recall:

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

- Slope of a height balanced tree is either -1 or 0 or 1
- In other words, for a height balanced tree t, $|\text{slope}(t)| < 2$
- After an insert or delete, it can happen that $|\text{slope}(t)| = 2$
- Violations happen only at nodes visited by operation
- We rebalance each node on the path visited by operation

Rebalancing – slope $= 2$

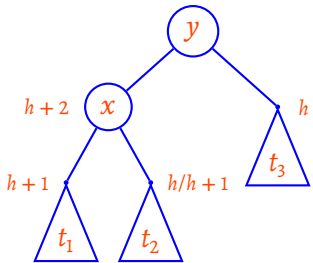
- Assume slope $= 2$ and both subtrees are balanced

Rebalancing – slope $= 2$

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

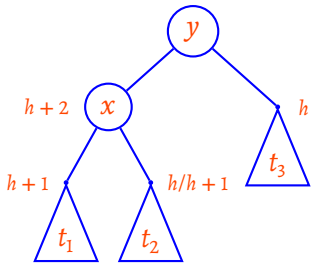
Rebalancing – slope $= 2$

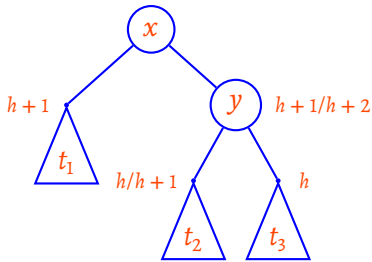
- Assume slope $= 2$ and both subtrees are balanced
- Slope of left subtree is 0 or 1 – **rotate right**



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 $= 2$

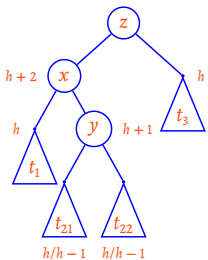
- Assume slope $= 2$ and both subtrees are balanced
- Slope of left subtree is -1

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**

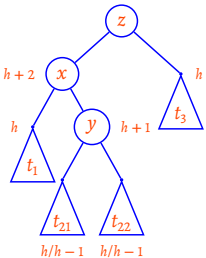
Rebalancing – slope $= 2$

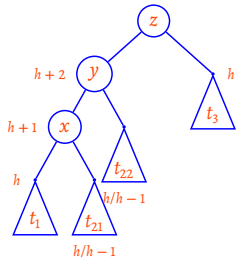
- Assume slope $= 2$ and both subtrees are balanced
- Slope of left subtree is -1
- **Left rotate the left subtree and then right rotate the tree**



Rebalancing – slope $= 2$

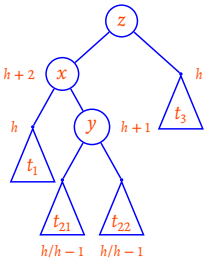
- Assume slope $= 2$ and both subtrees are balanced
- Slope of left subtree is -1
- **Left rotate the left subtree and then right rotate the tree**

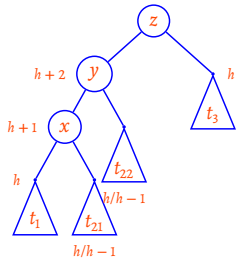


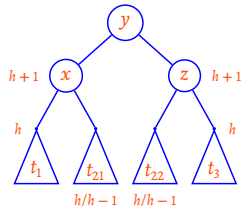


Rebalancing – slope $= 2$

- 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 $= -2$

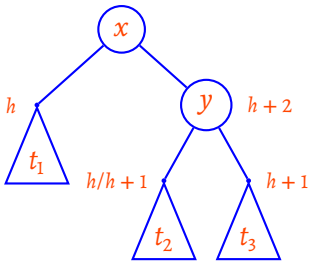
- Assume slope $= -2$ and both subtrees are balanced

Rebalancing – slope $= -2$

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

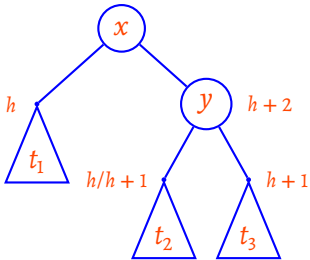
Rebalancing – slope $= -2$

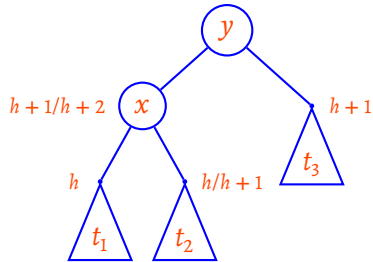
- Assume slope $= -2$ and both subtrees are balanced
- Slope of right subtree is 0 or -1 – **rotate left**



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 $= -2$

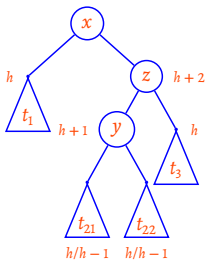
- Assume slope $= -2$ and both subtrees are balanced
- Slope of right subtree is 1

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**

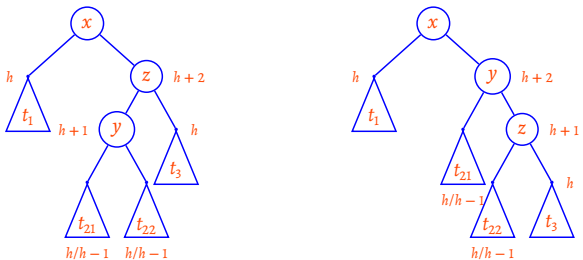
Rebalancing – slope $= -2$

- 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**



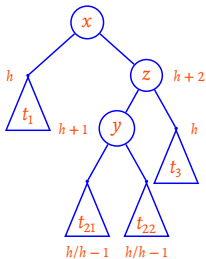
Rebalancing – slope $= -2$

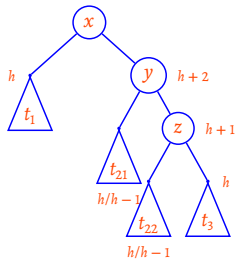
- Assume slope $= -2$ and both subtrees are balanced
- Slope of right subtree is 1
- **Right rotate the right subtree and then left rotate the tree**

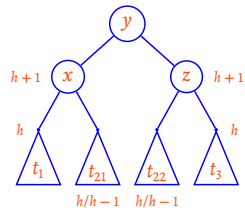


Rebalancing – slope $= -2$

- 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

```
rebalance :: AVL a → AVL a
rebalance t@(Node h tl x tr) = case (slope t, slope tl, slope tr) of
        (2, -1, _)  → rotateRight (Node h (rotateLeft tl) x tr)
        (2, _, _)   → rotateRight t
        (-2, _, 1)  → rotateLeft (Node h tl x (rotateRight tr))
        (-2, _, _)  → rotateLeft t
        _           → t
```

Rebalancing in Haskell

- The `rebalance` function

```
rebalance :: AVL a → AVL a
rebalance t@(Node h tl x tr) = case (slope t, slope tl, slope tr) of
        (2, -1, _)  → rotateRight (Node h (rotateLeft tl) x tr)
        (2, _, _)   → rotateRight t
        (-2, _, 1)  → rotateLeft (Node h tl x (rotateRight tr))
        (-2, _, _)  → rotateLeft t
        _           → t
```

- **Constant time** operation

Searching in an AVL tree

```
searchAVL :: Ord a ⇒ a → AVL a → Bool
searchAVL v Nil                 = False
searchAVL v (Node _ tl x tr)    = case compare v x of
    LT  → searchAVL v tl
    GT  → searchAVL v tr
    EQ  → True
```

Inserting in a tree

```
insertAVL :: Ord a ⇒ a → AVL a → AVL a
insertAVL v Nil                 = Node 1 Nil v Nil
insertAVL v t@(Node h tl x tr)  = let
    (ntl, ntr)      = (insertAVL v tl, insertAVL v tr)
    nhl             = 1 + max (height ntl) (height tr)
    nhr             = 1 + max (height tl) (height ntr)
        in case compare v x of
        LT  → rebalance (Node nhl ntl x tr)
        GT  → rebalance (Node nhr tl x ntr)
        EQ  → t
```

Deleting the maximum element

```
deleteMax :: AVL a → (a,AVL a)
deleteMax (Node _ tl x Nil) = (x, tl)
deleteMax (Node h tl x tr)  = (y, rebalance (Node nh tl x 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 tl x tr)  = let
    (y,ty)      = deleteMax tl
    (ntl, ntr)  = (deleteAVL v tl, deleteAVL v tr)
    nhy         = 1 + max (height ty) (height tr)
    nhl         = 1 + max (height ntl) (height tr)
    nhr         = 1 + max (height tl) (height ntr)
        in case compare v x of
        LT  → rebalance (Node nhl ntl x tr)
        GT  → rebalance (Node nhr tl x ntr)
        EQ  → if isEmpty tl then tr
                else rebalance (Node nhy ty y 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 $\leqslant 2 \log n + 1$

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 $\leqslant 2 \log n + 1$
- 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 $\leqslant 2 \log n + 1$
- 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 = foldr insertAVL emptyAVL
```

Other useful functions

- Create a sorted list from an AVL tree:

```
inorder :: Ord a ⇒ AVL a → [a]
inorder t                    = t +: [] where
    Nil             +:  xs   = xs
    Node _ tl x tr  +:  xs   = tl +: (x:(tr +: xs))
```

Other useful functions

- Create a sorted list from an AVL tree:

```
inorder :: Ord a ⇒ AVL a → [a]
inorder t                    = t +: [] where
    Nil             +:  xs   = xs
    Node _ tl x tr  +:  xs   = tl +: (x:(tr +: xs))
```

- $T(n) = T(m) + T(n - m - 1) + c$, so $T(n) = cn = O(n)$

Other useful functions

- Create a sorted list from an AVL tree:

```
inorder :: Ord a ⇒ AVL a → [a]
inorder t                    = t +: [] where
    Nil             +:  xs   = xs
    Node _ tl x tr  +:  xs   = tl +: (x:(tr +: xs))
```

- $T(n) = T(m) + T(n - m - 1) + c$, so $T(n) = cn = O(n)$
- 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

```
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 tl y 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,
        insertAVL, deleteAVL, searchAVL,
        createAVL, inorder, inorderTree) where

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

instance Show a ⇒ Show (AVL a) where
    show t = ...
```

A module for AVL trees

- Saved in `AVL.hs`

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

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

instance Show a ⇒ Show (AVL a) where
    show t = ...
```

- Can be used to define the `Set` ADT

The Set ADT again

```
module Set(Set, emptySet, createSet,
        insertInto, deleteFrom, search,
        setUnion, setIntersect, setDiff) 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

```
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

```
setUnion :: Ord a ⇒ Set a → Set a → Set a
setUnion (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

```
setIntersect :: Ord a ⇒ Set a → Set a → Set a
setIntersect (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

```
setDiff :: Ord a ⇒ Set a → Set a → Set a
setDiff (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 tree is a fundamental, but non-trivial data structure

Summary

- AVL tree is a fundamental, but non-trivial data structure
- Allows us to efficiently implement the `Set` ADT

Summary

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

Summary

- AVL tree is a fundamental, but non-trivial data structure
- 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 tree is a fundamental, but non-trivial data structure
- 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 tree is a fundamental, but non-trivial data structure
- 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 tree is a fundamental, but non-trivial data structure
- 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 tree is a fundamental, but non-trivial data structure
- 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