CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu

Dr. Oliver Kennedy
okennedy@buffalo.edu

212 Capen Hall

Day 26
AVL Trees

Announcements

- WA2 due tonight @ 11:59pm

BST Operations

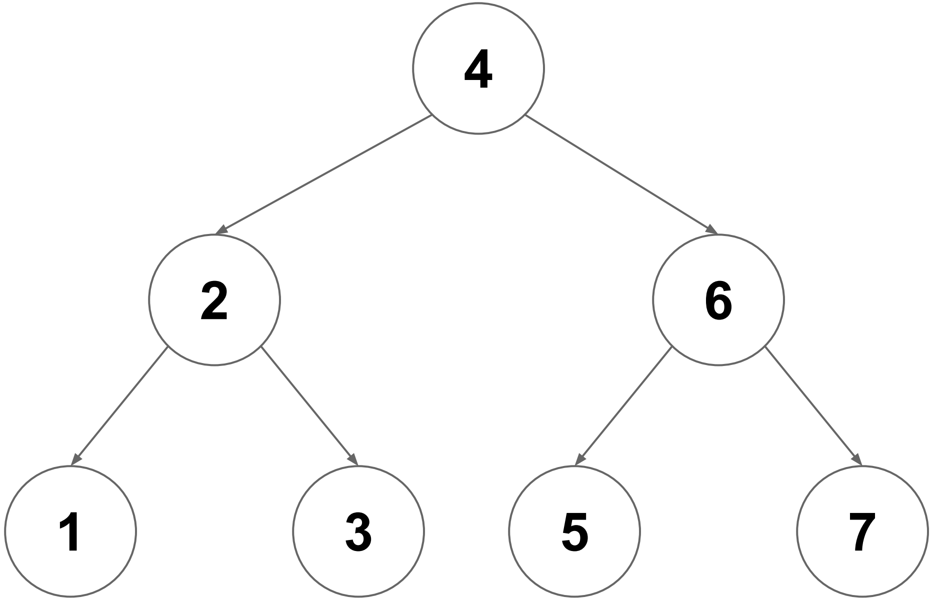
Operation	Runtime
`find`	$O(d)$
`insert`	$O(d)$
`remove`	$O(d)$

*What is the runtime in terms of **n**?* $\boldsymbol{O(n)}$

$$\log(n) \le d \le n$$

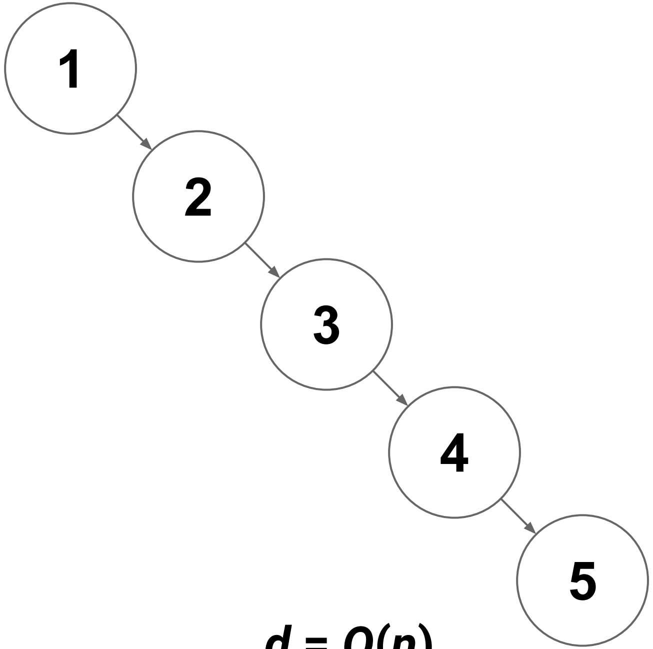
Tree Depth vs Size

If height(left) ≈ height(right)



$d = O(\log(n))$

If height(left) ≪ height(right)



$d = O(n)$

Balanced Trees

Balanced Trees are good: Faster **`find`**, **`insert`**, **`remove`**

Balanced Trees

Balanced Trees are good: Faster `find`, `insert`, `remove`

What do we mean by balanced?

Balanced Trees

Balanced Trees are good: Faster **`find`**, **`insert`**, **`remove`**

What do we mean by balanced? **|height(left) - height(right)| ≤ 1**

Balanced Trees

Balanced Trees are good: Faster **`find`**, **`insert`**, **`remove`**

What do we mean by balanced? **|height(left) - height(right)| ≤ 1**

How do we keep a tree balanced?

Balanced Trees - Two Approaches

Option 1

Keep left/right subtrees within **+/-1** of each other in height

(add a field to track amount of "imbalance")

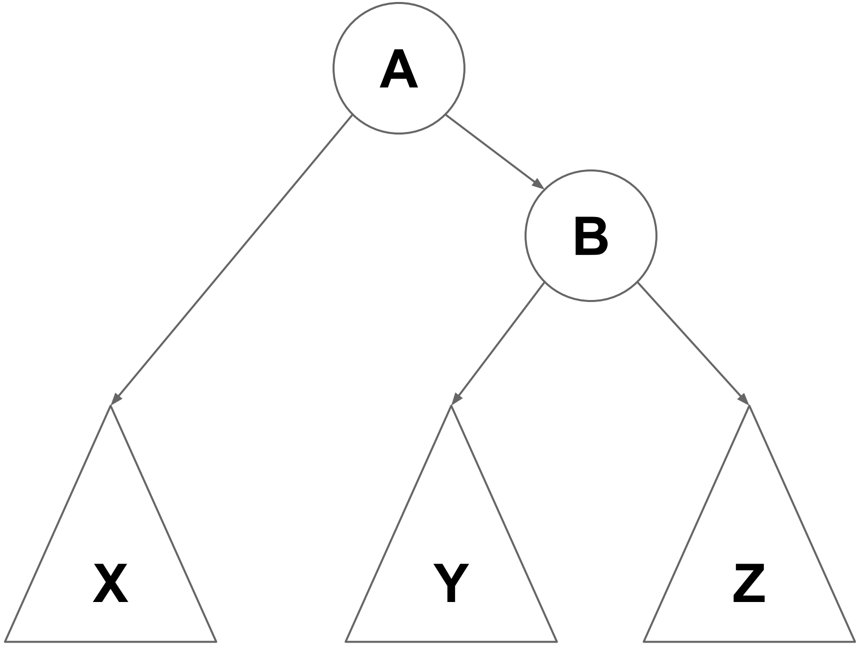
Option 2

Keep leaves at some minimum depth (***d*/2**)

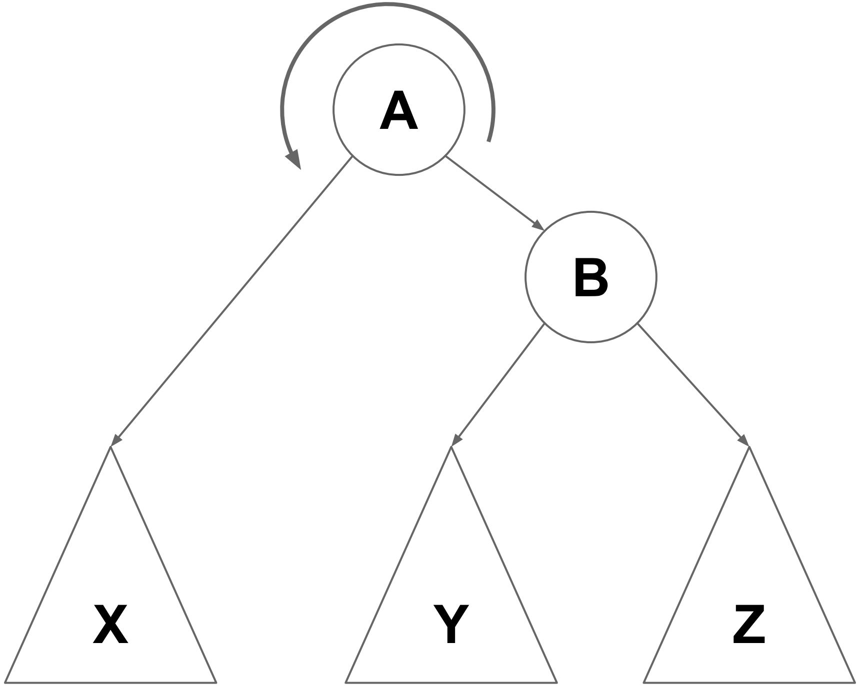
(Add a color to each node marking it as "red" or "black")

Ok…but how do we enforce this…?

Rebalancing Trees (rotations)

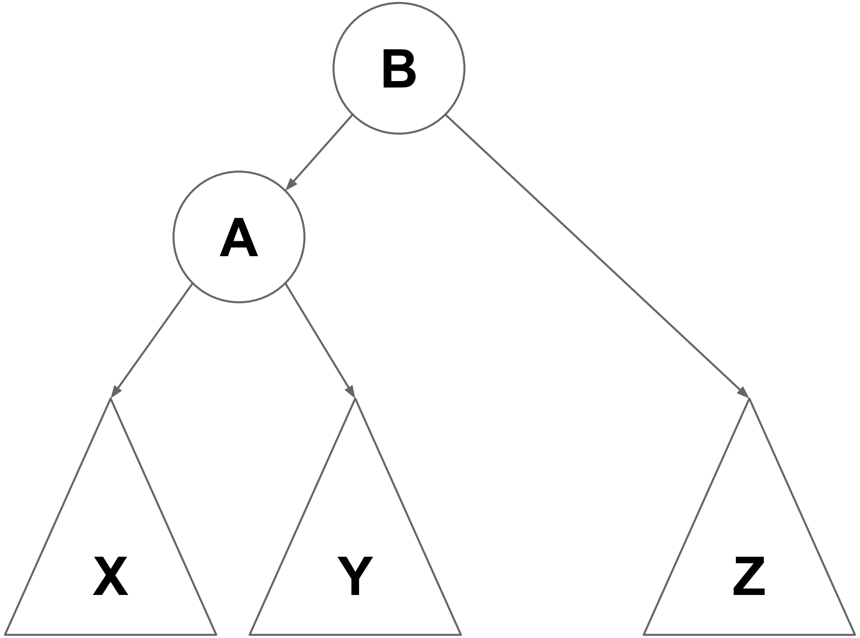


Rebalancing Trees (rotations)



Rotate(A, B)

Rebalancing Trees (rotations)

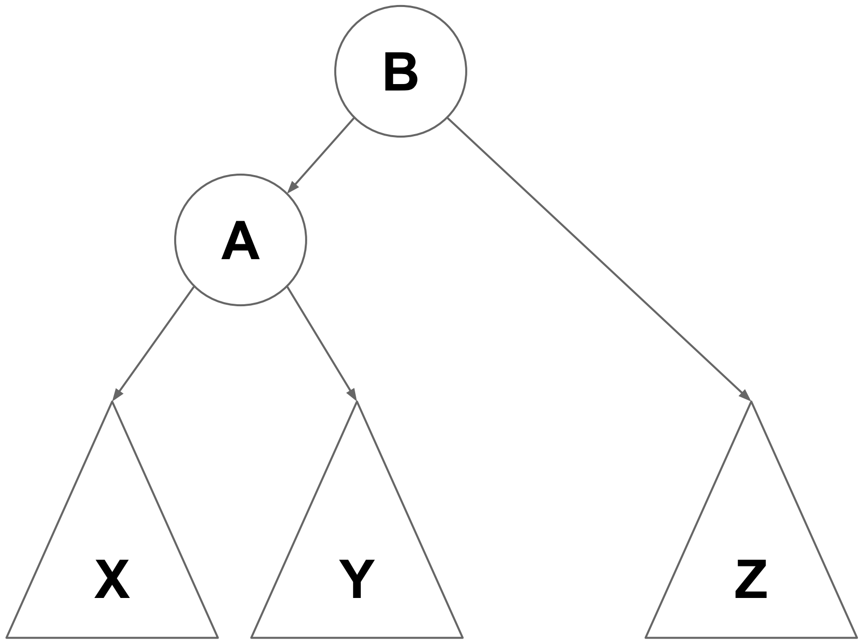


Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child



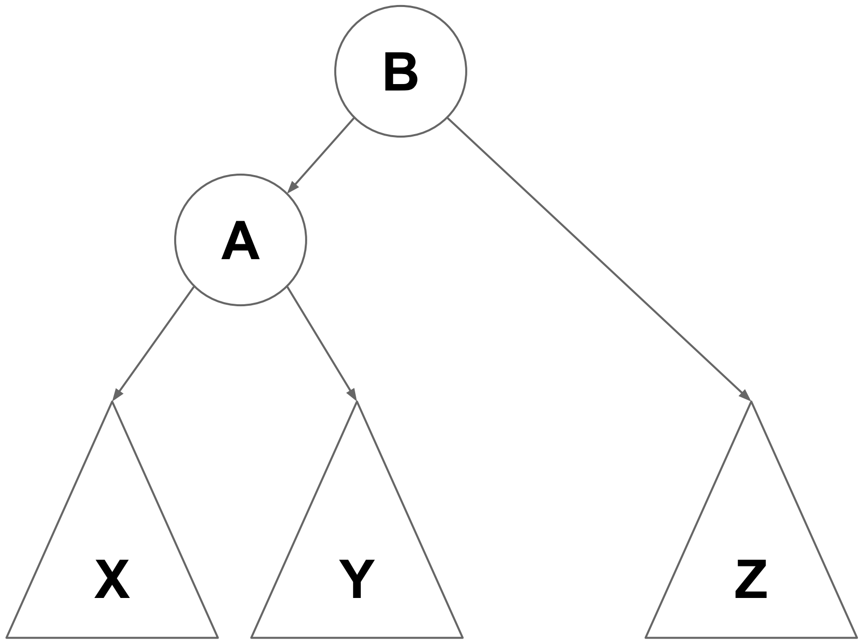
Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

Is ordering maintained?



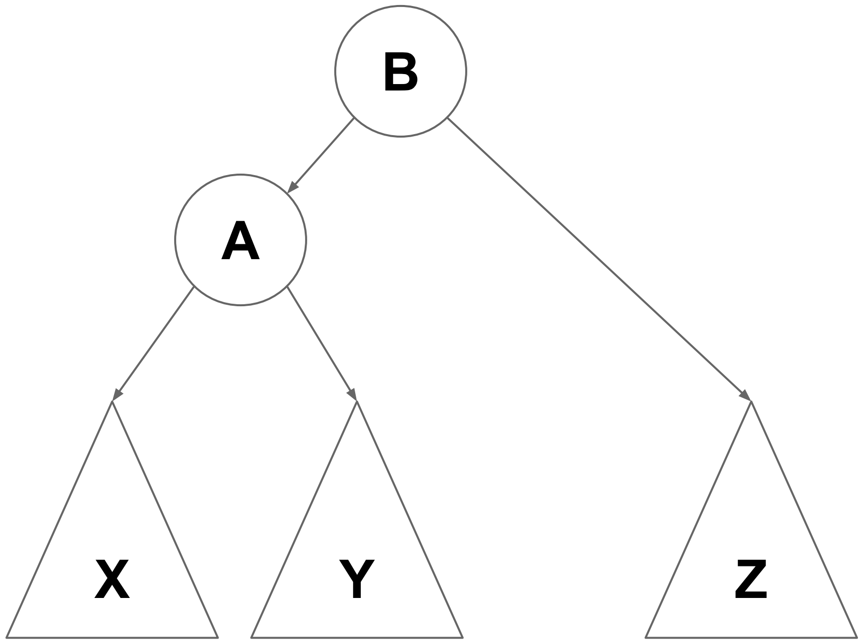
Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

Is ordering maintained? **Yes!**



Rotate(A, B)

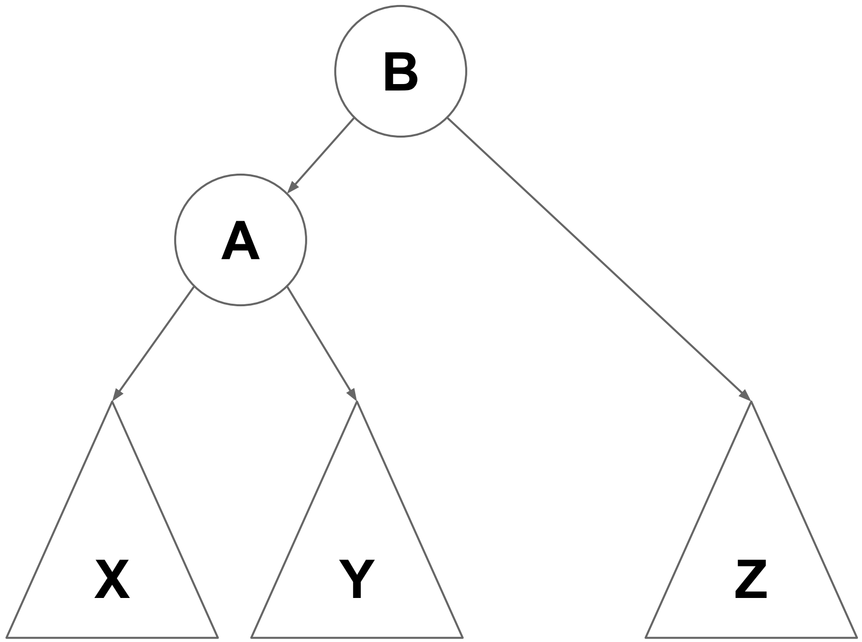
Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

Is ordering maintained? **Yes!**

Complexity?



Rotate(A, B)

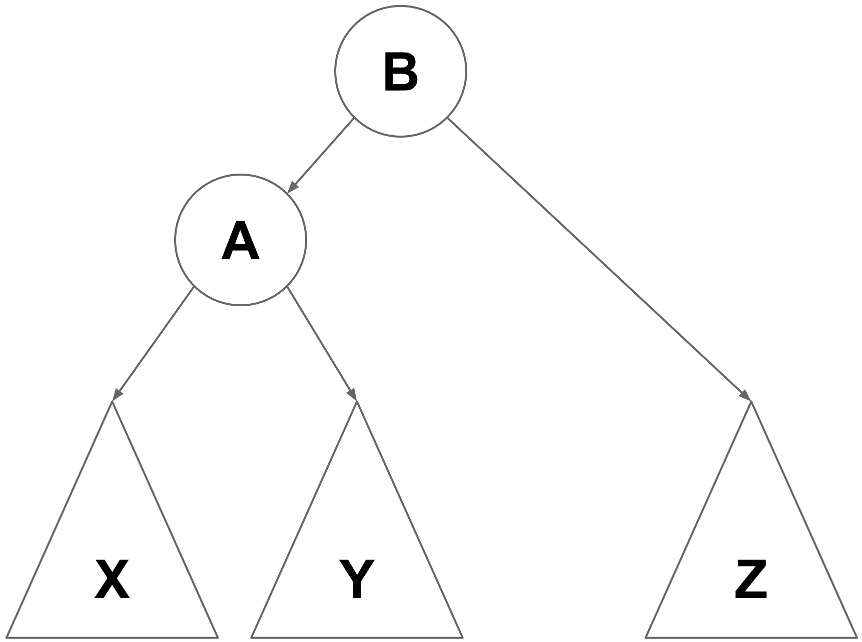
Rebalancing Trees (rotations)

A became **B**'s left child

B's left child became **A**'s right child

Is ordering maintained? **Yes!**

Complexity? ***O*(1)**



Rotate(A, B)

Rebalancing Trees (rotations)

A became **B**'s left child

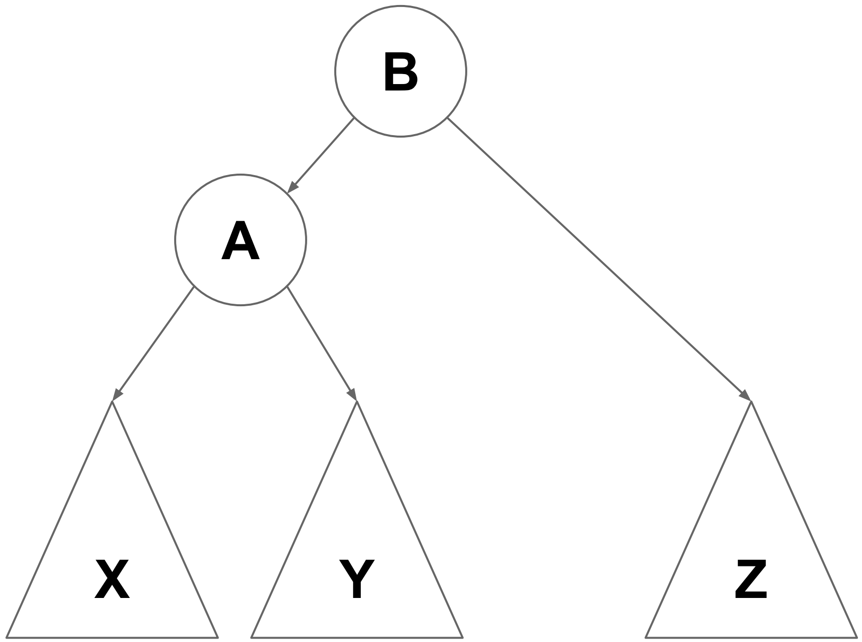
B's left child became **A**'s right child

Is ordering maintained? **Yes!**

Complexity? ***O*(1)**

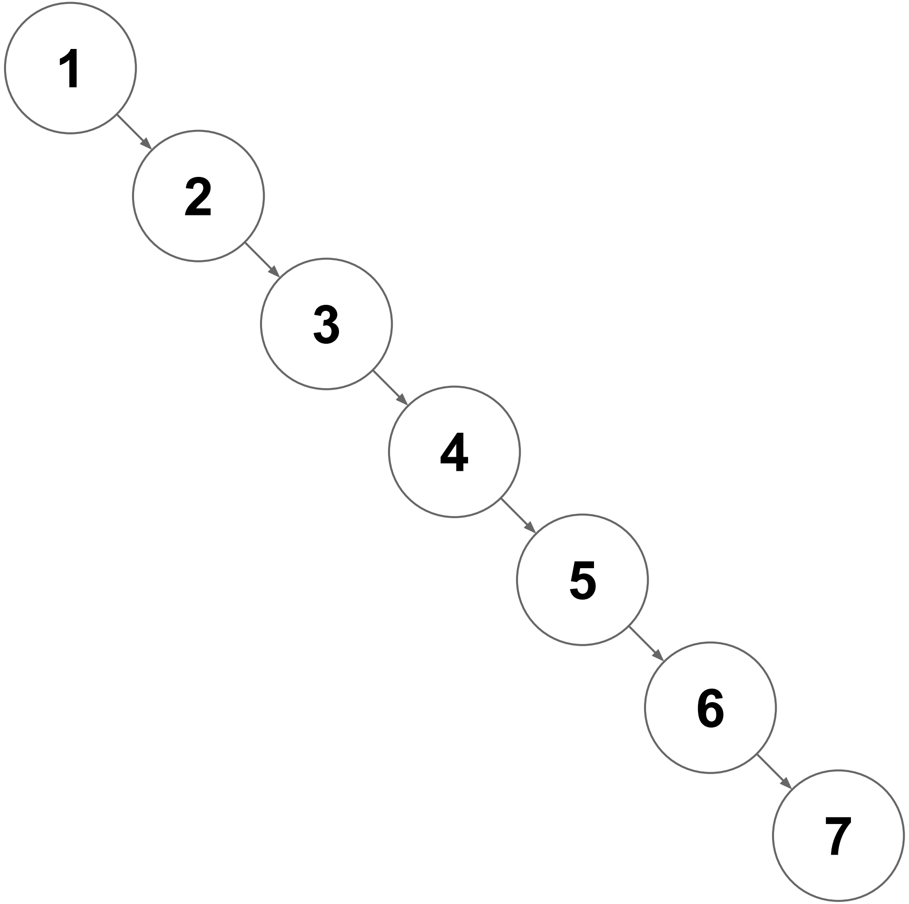
This is called a left rotation

(right rotation is the opposite)



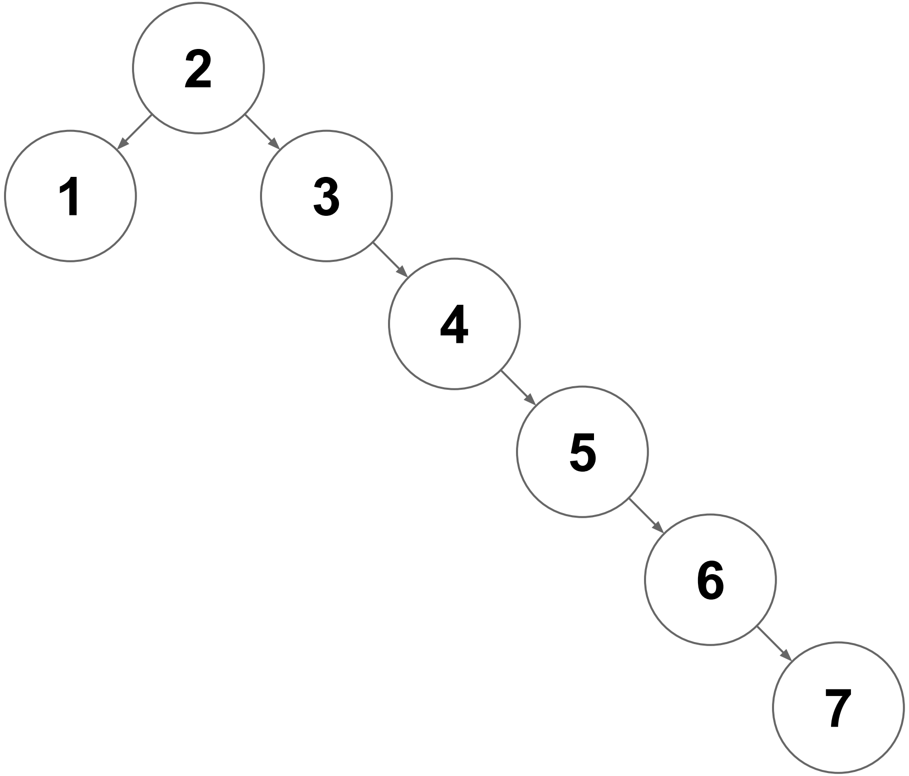
Rotate(A, B)

Rebalancing Trees



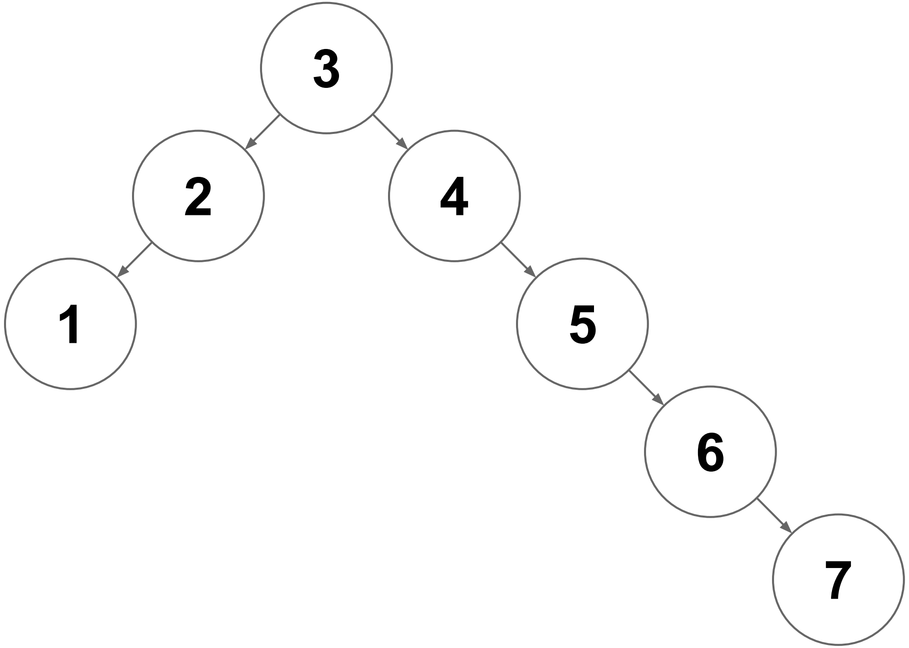
Rebalancing Trees

`Rotate(1,2)`



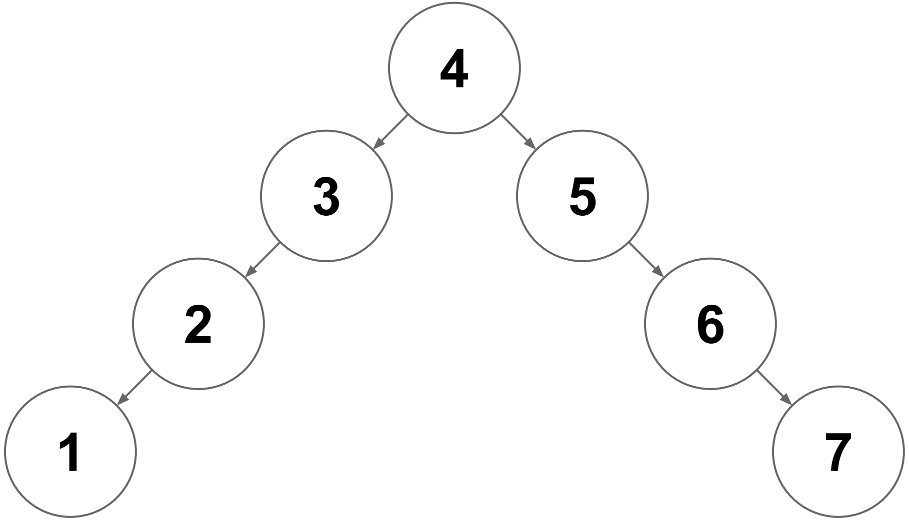
Rebalancing Trees

`Rotate(2,3)`



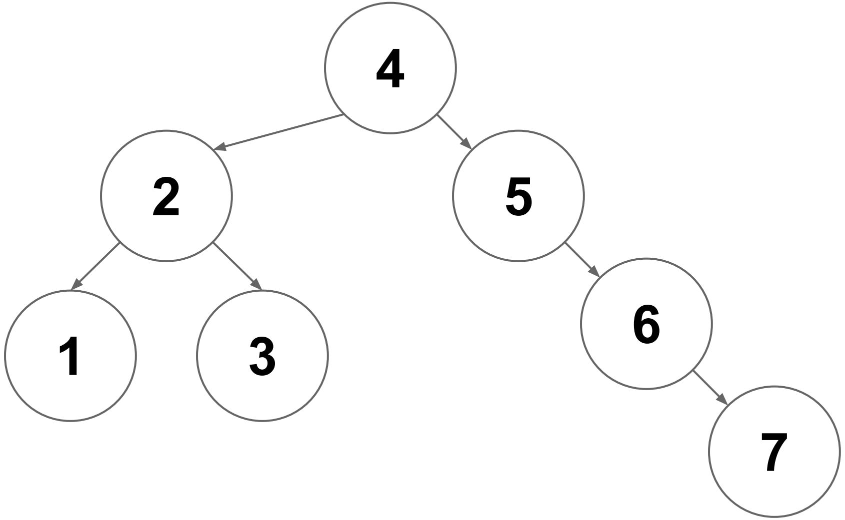
Rebalancing Trees

`Rotate(3,4)`



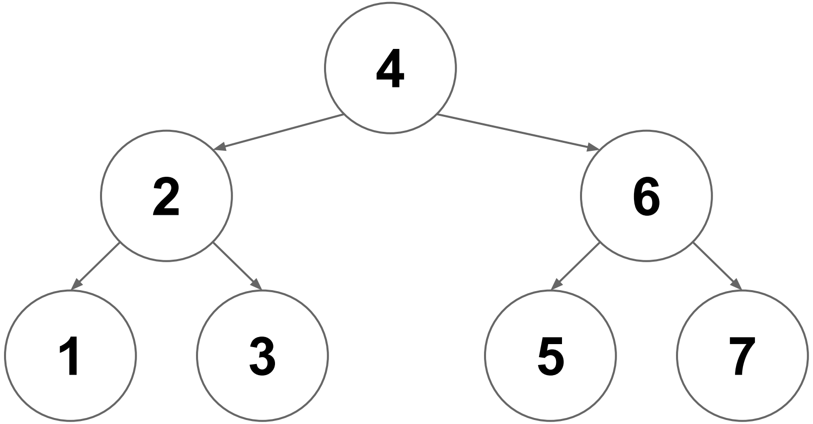
Rebalancing Trees

`Rotate(3,2)`



Rebalancing Trees

`Rotate(5,6)`



AVL Trees

AVL Trees

An **AVL tree** (**A**delson-**V**elsky and **L**andis) is a ***BST*** where every subtree is depth-balanced

Remember: Tree depth = height(root)

Balanced: |height(root.left) - height(root.right)| ≤ 1

AVL Trees

Define **balance(*v*) = height(*v.right*) - height(*v.left*)**

Goal: Maintaining **balance(*v*) ∈ { -1, 0, 1 }**

- **balance(*v*) = 0** → "***v*** is balanced"
- **balance(*v*) = -1** → "***v*** is left-heavy"
- **balance(*v*) = 1** → "***v*** is right-heavy"

AVL Trees

Define **balance(*v*) = height(*v.right*) - height(*v.left*)**

Goal: Maintaining **balance(*v*) ∈ { -1, 0, 1 }**

- **balance(*v*) = 0** → "***v*** is balanced"
- **balance(*v*) = -1** → "***v*** is left-heavy"
- **balance(*v*) = 1** → "***v*** is right-heavy"

What does enforcing this gain us?

AVL Trees - Depth Bounds

Question: Does the AVL property result in any guarantees about depth?

AVL Trees - Depth Bounds

Question: Does the AVL property result in any guarantees about depth?

YES! Depth balance forces a maximum possible depth of **log(*n*)**

AVL Trees - Depth Bounds

Question: Does the AVL property result in any guarantees about depth?

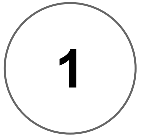
YES! Depth balance forces a maximum possible depth of **log(*n*)**

Proof Idea: An AVL tree with depth ***d*** has "enough" nodes

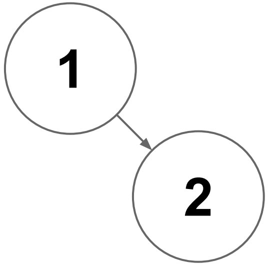
AVL Trees - Depth Bounds

Let **minNodes(*d*)** be the minimum number of nodes an in AVL tree of depth ***d***

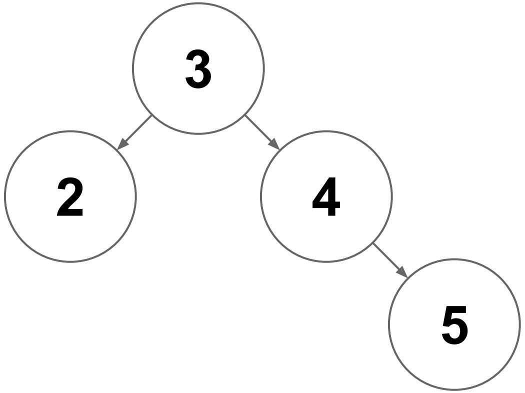
minNodes(0) = 1



minNodes(1) = 2

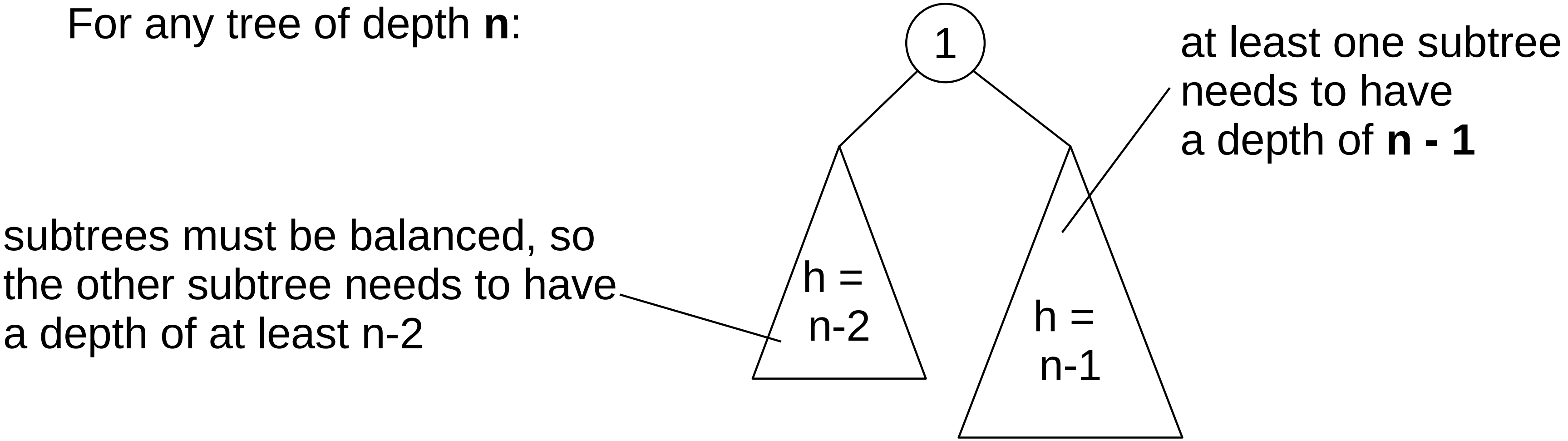


minNodes(2) = 4



AVL Trees

For any tree of depth **n**:



$$\text{minNodes}(n) = \begin{cases} \end{cases}$$

Enough Nodes?

- For $d > 1$
  - $\text{minNodes}(d) = 1 + \text{minNodes}(d-1) + \text{minNodes}(d-2)$
  - This is the Fibbonacci Sequence!
    - $\text{minNodes}(d) = \text{Fib}(d+3)-1$
    - $\text{Fib}(0), \text{Fib}(1), \text{Fib}(2), \ldots = 0, 1, 1, 2, 3, 5, 8, \ldots$
  - $\text{minNodes}(d) = \Omega(1.5^d)$

Enough Nodes?

- minNodes(d) = $\Omega(1.5^d)$

$$n \geq c1.5^d$$

$$\frac{n}{c} \geq 1.5^d$$

$$\log_2\left(\frac{n}{c}\right) \geq \log_2\left(1.5^d\right)$$

$$\log_2\left(\frac{n}{c}\right) \geq \log_{1.5}(1.5^d)\log_2 1.5$$

$$\log_2\left(\frac{n}{c}\right) \geq d\log_2(1.5)$$

$$\frac{\log_2\left(\frac{n}{c}\right)}{\log_2(1.5)} \geq d$$

constant

$$\frac{\log_2(n)}{\log_2(1.5)} - \frac{\log_2(c)}{\log_2(1.5)} \geq d$$

$$O\left(\log_2(n)\right) \geq d$$

A tree with n nodes and the AVL constraint has logarithmic depth in n

Enforcing the AVL Constraint

- Computing balance() on the fly is expensive
  - balance calls height() twice
  - Computing height requires visiting every node
    - (linear in the size of the subtree)
- **Idea**: Store height of each node at the node
  - **Better idea**: Store balance factor (only requires 2 bits)

Enforcing the AVL Constraint

maintaining _parent makes it possible to traverse up the tree (helpful for rotations), but is not possible in an immutable tree.

```
class AVLNode[K, V](
  var _key: K,
  var _value: V,
  var _parent: Option[AVLNode[K,V]],
  var _left: AVLNode[K,V],
  var _right: AVLNode[K,V],
  var _isLeftHeavy: Boolean,  // true if balance(this) == -1
  var _isRightHeavy: Boolean, // true if balance(this) == 1
)
```

$$balance(n) = \begin{cases} -1 & \textbf{if } \texttt{n.\_isLeftHeavy} = \mathbf{T} \\ +1 & \textbf{if } \texttt{n.\_isRightHeavy} = \mathbf{T} \\ 0 & \textbf{otherwise} \end{cases}$$

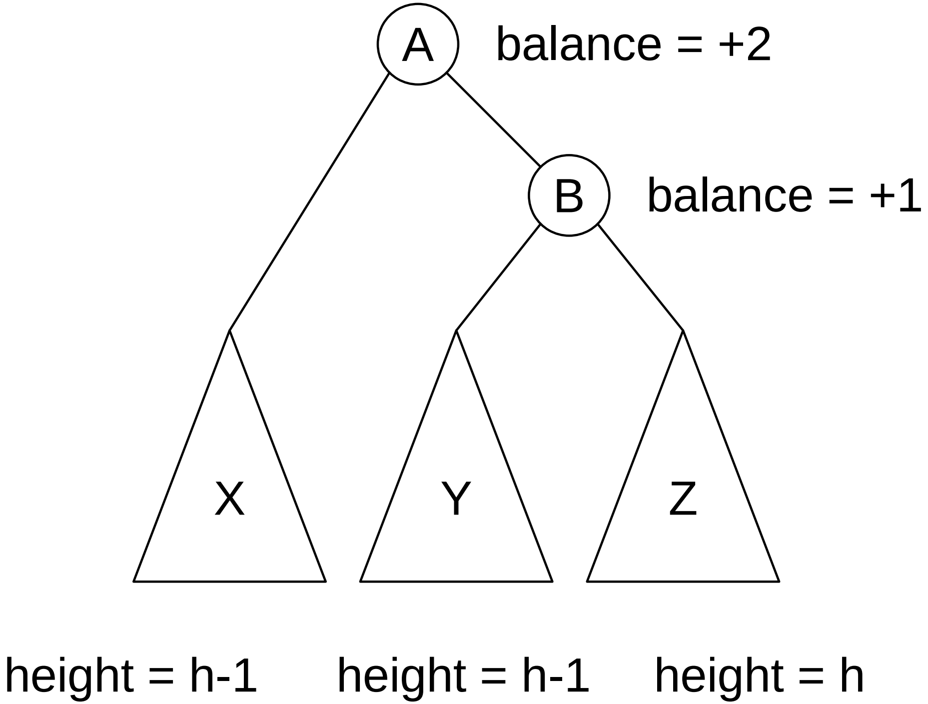
Enforcing the AVL Constraint

- Left Rotation
  - Before
    - **(A)** root; balance(**A**) = +2 (<u>too</u> right heavy)
    - **(B)** root.right; balance(**B**) = +1 (right heavy)

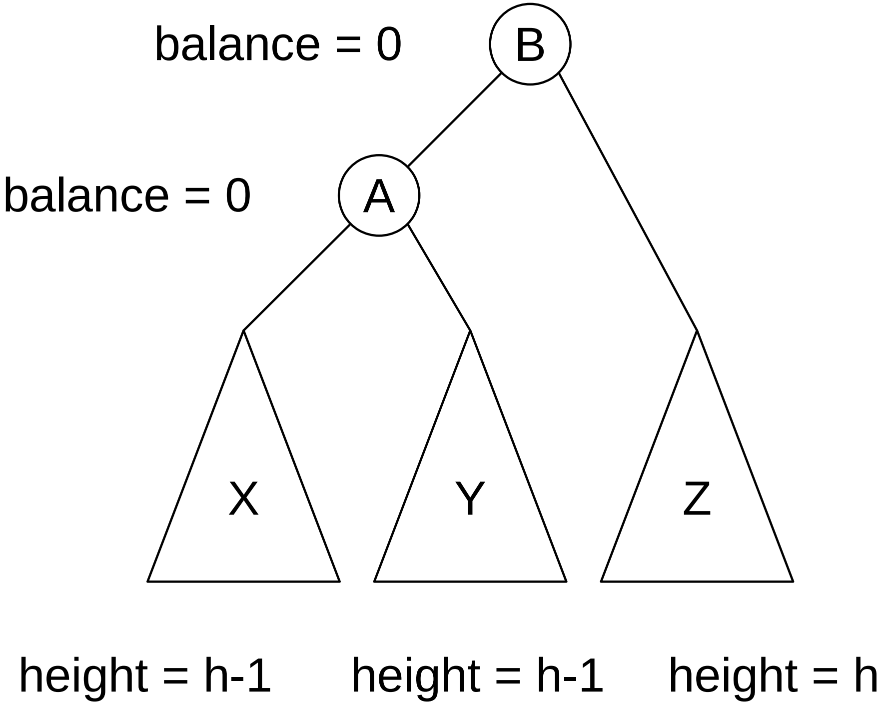
1) Left subtree of **(B)** becomes right subtree of **(A)**.
2) **(A)** becomes left subtree of **(B)**
3) **(B)** becomes root

  - After
    - balance(**A**) = 0, balance(**B**) = 0

Enforcing the AVL Constraint



Enforcing the AVL Constraint



Enforcing the AVL Constraint

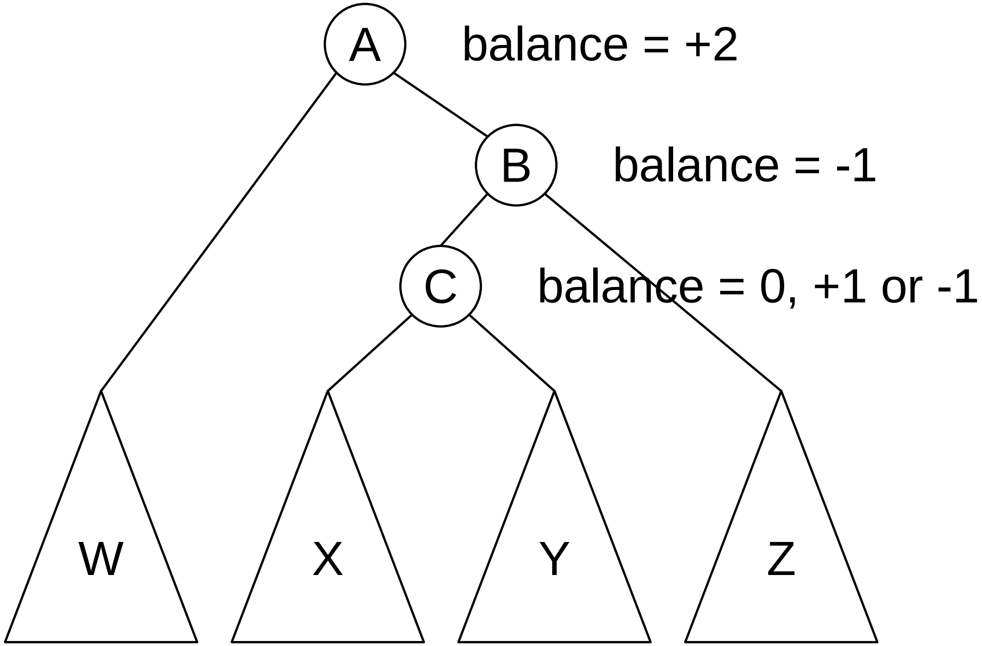
- Right-Left Rotation
  - Before
    - **(A)** root; balance(**A**) = +2 (too right heavy)
    - **(B)** root.right; balance(**B**) = -1 (left heavy)
    - **(C)** right.left.right

1) Left subtree of **(C)** becomes right subtree of **(A)**.
2) Right subtree of **(C)** becomes left subtree of **(B)**.
3) **(A)** becomes left subtree of **(C)**
4) **(B)** becomes right subtree of **(C)**
5) **(C)** becomes root

Enforcing the AVL Constraint

- After
  - if **(C)**'s BF was originally 0
    - **(A)** BF = 0; **(B)** BF = 0; **(C)** BF = 0
  - if **(C)**'s BF was originally -1
    - **(A)** BF = 0; **(B)** BF = +1; **(C)** BF = 0
  - if **(C)**'s BF was originally +1
    - **(A)** BF = -1; **(B)** BF = 0; **(C)** BF = 0

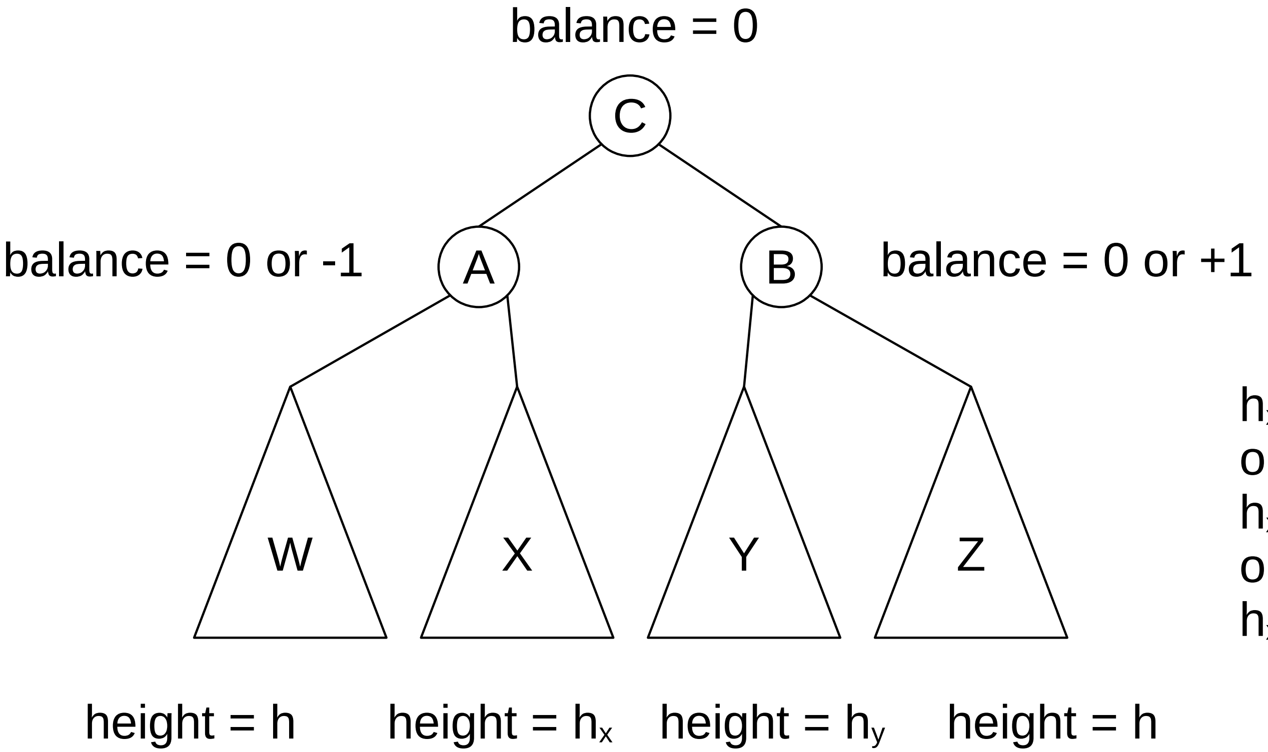
Enforcing the AVL Constraint



height = h height = h_x height = h_y height = h

$h_x = h_y = h$
or
$h_x = h - 1; h_y = h$
or
$h_x = h; h_y = h - 1$

Enforcing the AVL Constraint



$h_x = h_y = h$
or
$h_x = h - 1; h_y = h$
or
$h_x = h; h_y = h - 1$

Enforcing the AVL Constraint

- Rotate Right
  - Symmetric to rotate left
- Rotate Left-Right
  - Symmetric to rotate right-left

Inserting Records

- Inserting Records
  - Find insertion as in BST
  - Set balance factor of new leaf to 0
    - _isLeftHeavy = _isRightHeavy = false
  - Trace path up to root, updating balance factor
    - Rotate if balance factor off

Inserting Records

```
def insert[K, V](key: K, value: V, root: AVLNode[K, V]): Unit =
{
  var node = findInsertionPoint(key, root)
  node._key = key;   node._value = value
  node._isLeftHeavy = node._isRightHeavy = false
  while(node._parent.isDefined){
    if(node._parent._left == node){
      if(node._parent._isRightHeavy){
        node._parent._isRightHeavy = false; return
      } else if(node._parent._isLeftHeavy) {
        if(node._isLeftHeavy){ node._parent.rotateRight() }
        else { node._parent.rotateLeftRight() }
        return
      } else {
        node._parent.isLeftHeavy = true
      }
    } else {
      /* symmetric to above */
    }
    node = node._parent
} }
```

O(d) = O(log(n))

O(d) = O(log(n)) loops

O(1) per loop

Total Runtime = O(log(n))

Removing Records

- Removing Records
  - Remove the node
    - Find the node containing the value as in BST
      - If it doesn’t exist, return false
    - If the node is a leaf, remove it
    - If the node has one child, the child replaces the node
    - If the node has two children
      - copy smaller child value into node
      - remove smaller child node
  - Fix balance factors
    - Inverse of insertion

Maintaining Balance

- **Claim**: Only the balance factors of ancestors are impacted
  - The height of a node is only affected by its descendents
- **Claim**: Only one rotation will fix any remove/insert imbalance
  - Insert/remove change the height by at most one
- Only log(n) rotations are required for any insert/remove
  - Insert/remove are still log(n)