CSE 250 Lecture 25

AVL Trees

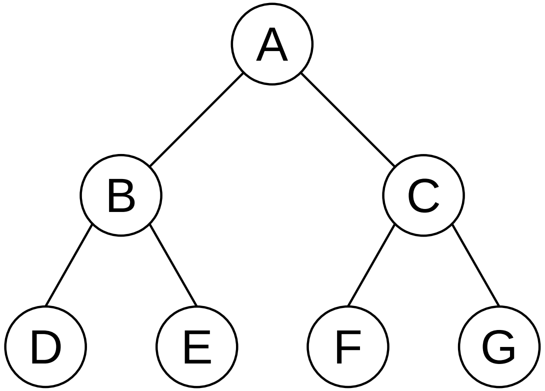
A CAT Tree

BST Operation Costs

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

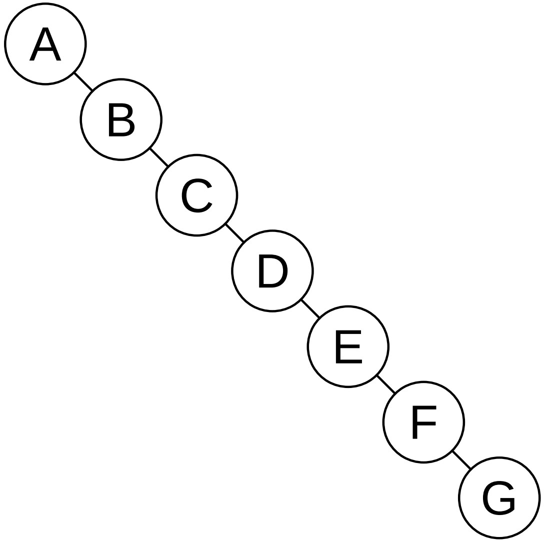
Tree Depth vs Size

height(left) ≈ height(right)



d = O(log(n))

height(left) ⋘ height(right)



d = O(n)

“Balanced” Trees

- Faster search: Want height(left) ≈ height(right)
  - Make it more precise: |height(left) - height(right)| ≤ 1
  - (left, right height differ by at most 1)
- **Question**: How do we keep the tree balanced?
  - Option 1: Keep left/right subtrees within **+/- 1** of each other
    - Add a field to track the “imbalance factor”
  - Option 2: Ensure leaves are at a minimum depth of **d / 2**
    - Add a designation marking each node as red or black

AVL Trees

AVL Trees

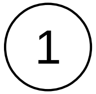
- An AVL tree (Adelson-Velsky and Landis) is a BST where every subtree is “depth-balanced”
    - (remember tree depth = root height)
    - |height(left child) - height(right child)| ≤ 1
- define **`balance(v) = height(v.right) - height(v.left)`**
    - Maintain balance(v) ∈ { -1, 0, 1 }
        - balance(v) = 0 → “v is balanced”
        - balance(v) = -1 → “v is left-heavy”
        - balance(v) = 1 → “v is right-heavy”

AVL Trees

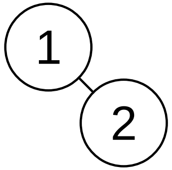
- **Goal**: AVL tree property maintains a nearly balanced tree
  - Depth balance forces a maximum possible depth $d \ll n$
    - ($d \ll n$ means $d \leq c \log(n)$ for some constant $c > 0$)
- **Proof idea**: An AVL tree with depth d has “enough” nodes

AVL Trees

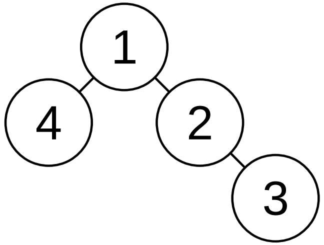
- Let minNodes(d) be the minimum number of nodes in an AVL tree of depth d



minNodes(0) = 1



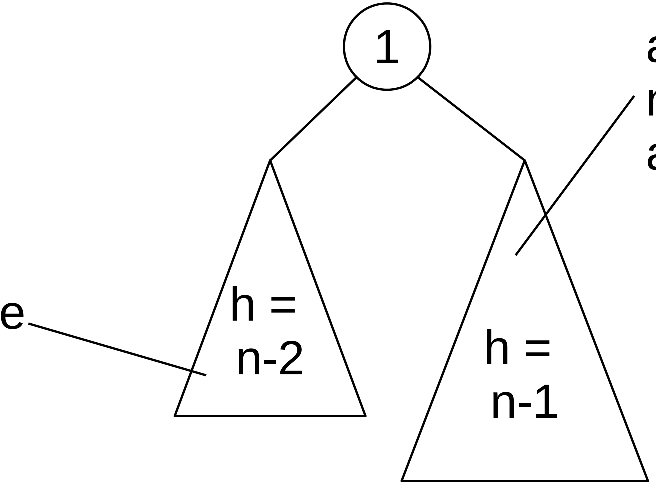
minNodes(1) = 2



minNodes(2) = 4

AVL Trees

For any tree of depth **n**:



at least one subtree needs to have a depth of **n - 1**

subtrees must be balanced, so the other subtree needs to have a depth of at least n-2

$$\mathrm{minNodes}(n) = \left\{ \vphantom{\begin{matrix} a \\ b \\ c \end{matrix}} \right.$$

Enough Nodes?

- For d > 1
  - minNodes(d) = 1 + minNodes(d-1) + minNodes(d-2)
  - This is the Fibbonacci Sequence!
    - minNodes(d) = Fib(d+3)-1
    - Fib(0), Fib(1), Fib(2), … = 0, 1, 1, 2, 3, 5, 8, …
  - 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
)
```

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

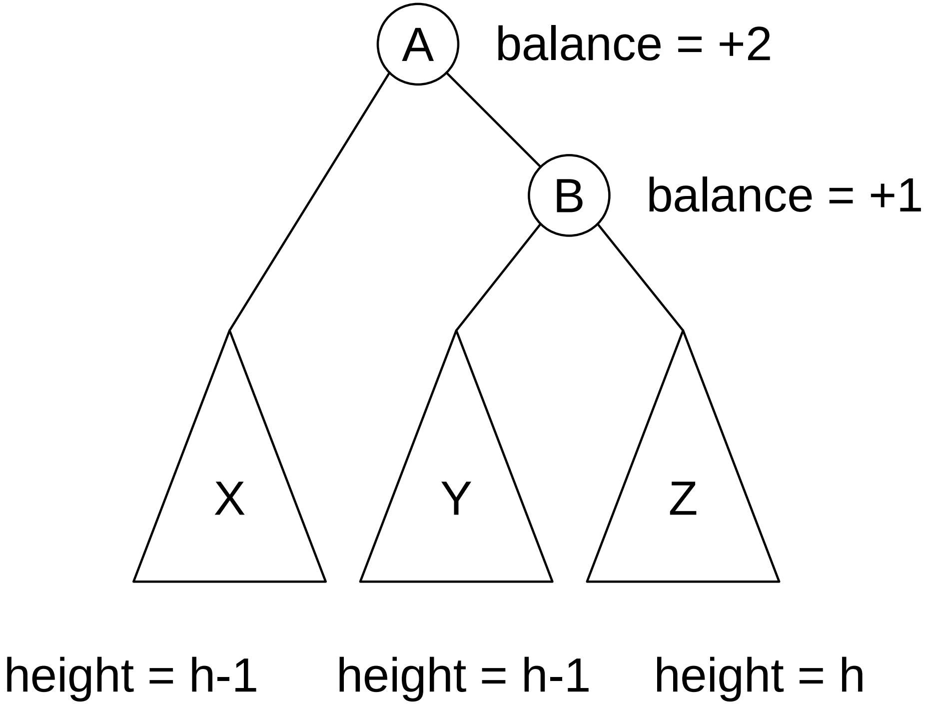
Enforcing the AVL Constraint

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

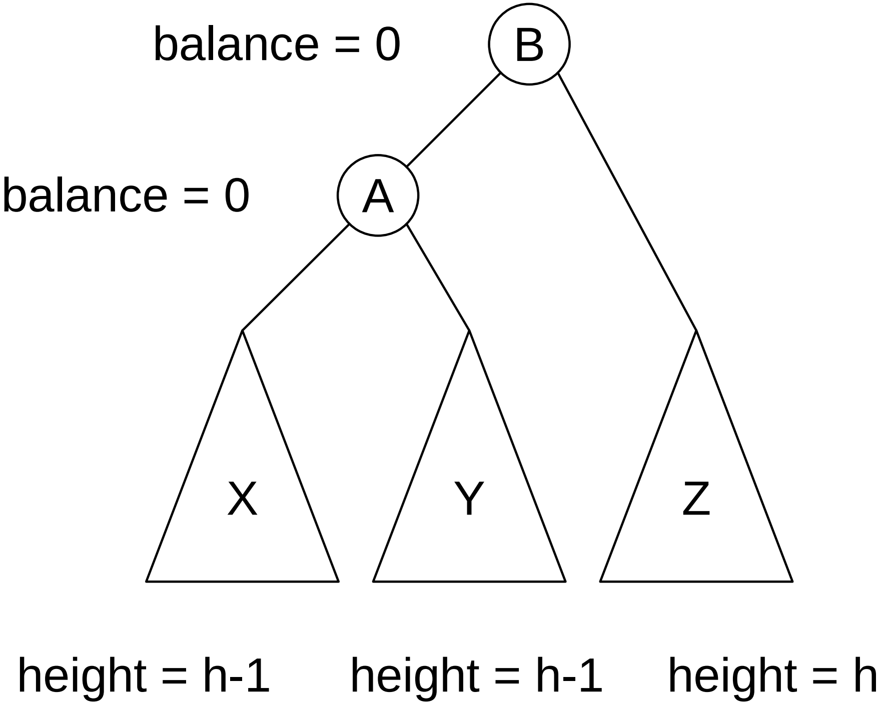
  1) Left subtree of **(B)** becomes right subtree of **(A)**.
  2) **(A)** becomes left subtree of **(B)**
  3) **(B)** becomes root

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

Enforcing the AVL Constraint



Enforcing the AVL Constraint



Enforcing the AVL Constraint

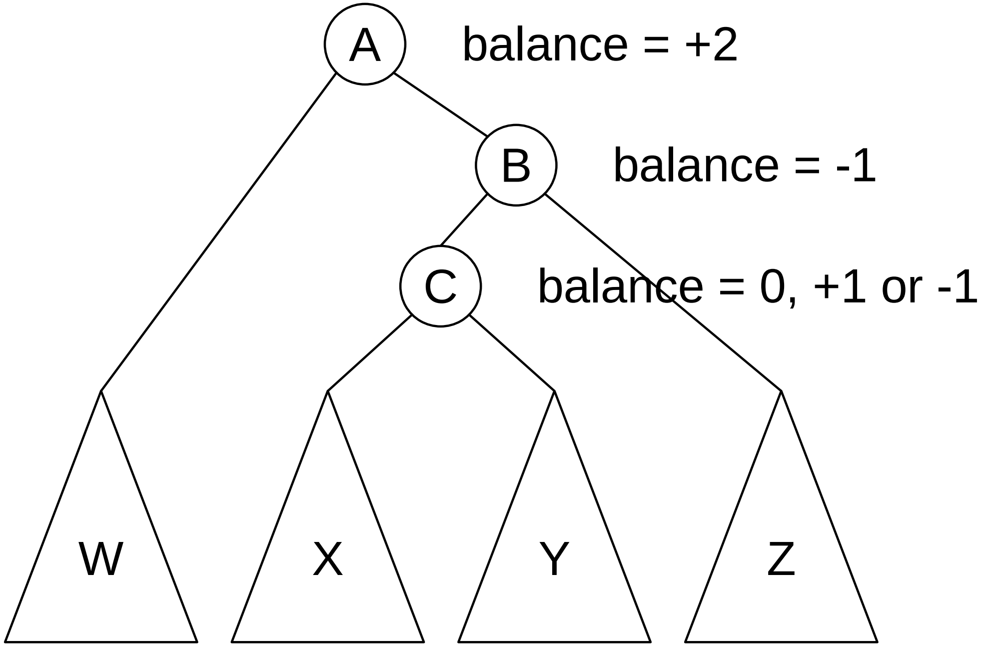
- Right-Left Rotation
  - Before
    - **(A)** root; balance(**A**) = +2 (too right heavy)
    - **(B)** root.right; balance(**B**) = -1 (left heavy)
    - **(C)** right.left.right

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



balance = +2

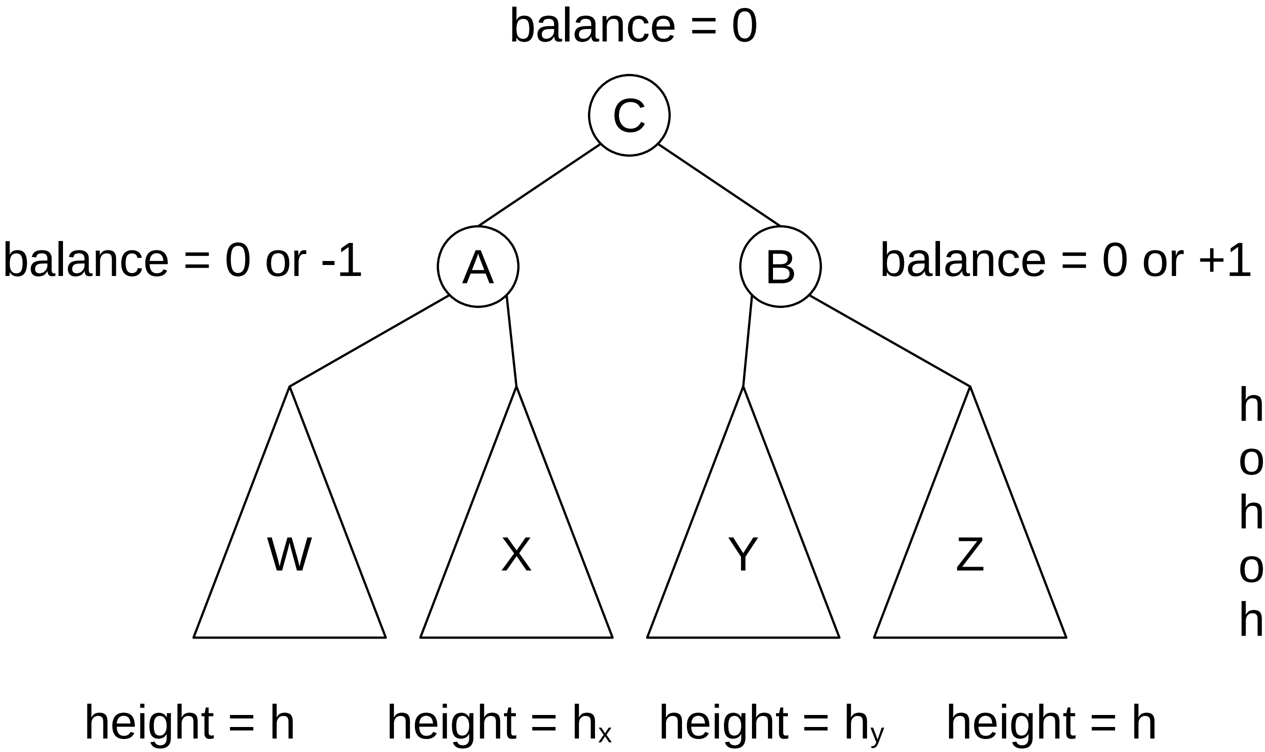
balance = -1

balance = 0, +1 or -1

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)