CS 310: Tree Rotations and AVL Trees

Chris Kauffman

Week 12-2

Practice/Demo Sites

- jGrasp is so-so for seeing tree operations
- Play with Blanced Binary Search Trees online using the following applets (titles hyperlinked)

Arsen Gogeshvili: Tree Applet

- Requires Flash
- Standard BSTs
  - Manual Rotation
  - Great Practice
- AVL Trees
- Undo/Redo to rewatch
- Step by step logging

Adjustable Demo

- Standard BSTs
- All three Balanced
  - AVL, Red black, Splay
- Slow down, pause, show balance factors

Scaling AVL (broken)

- AVL Tree only
- Scaling view for large trees

Why Worry About Insertion and Removal?

- Q: Why worry about `insert/remove` messing with the tree? What affect can it have on the performance of future ops on the tree?
- Q: What property of a tree dictates the runtime complexity of its operations?

Balancing Trees

- `add`/`remove`/`find` complexity *O(height(t))*
- Degenerate tree has height *N*: a linked list
- Prevent this by re-balancing on `insert`/`remove`
- Several kinds of trees do this

AVL	left/right subtree height differ by max 1
Red-black	preserve 4 red/black node properties
AA	red-black tree + all left nodes black
Splay	amoritized bound on ops, very different

The AVL Tree

The AVL tree is named after its two Soviet inventors, Georgy Adelson-Velsky and E. M. Landis, who published it in their 1962 paper "An algorithm for the organization of information".
– Wikip: AVL Tree

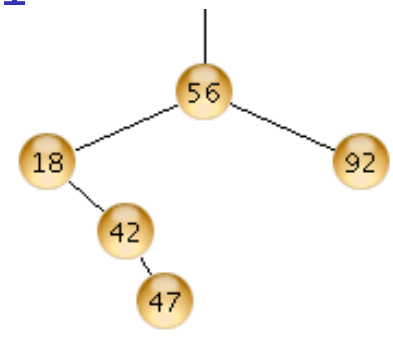
- A self-balancing tree
- Operations
- Proof of logarithmic height

AVL Balance Property

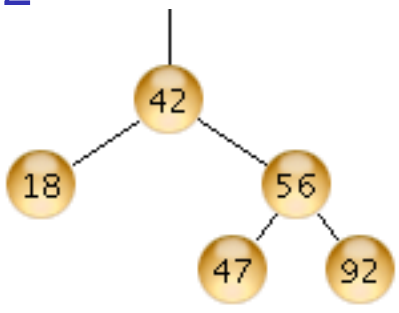
`T` is an AVL tree if and only if

- `T.left` and `T.right` differ in height by at most 1
- AND `T.left` and `T.right` are AVL trees

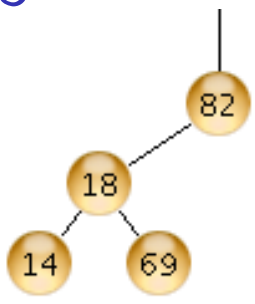
1



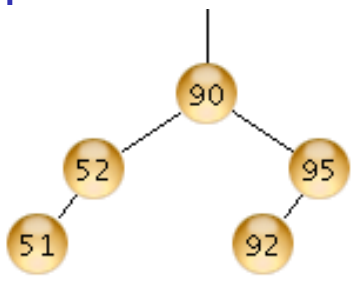
2



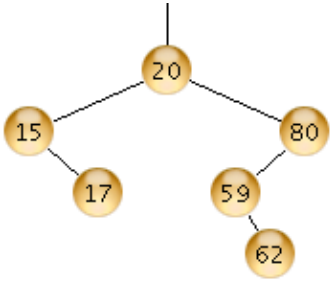
3



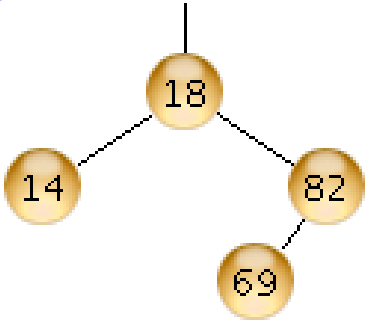
4



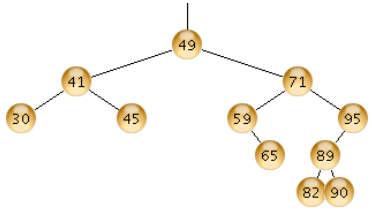
5



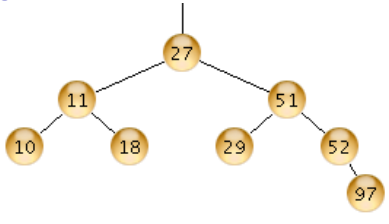
6



7



8

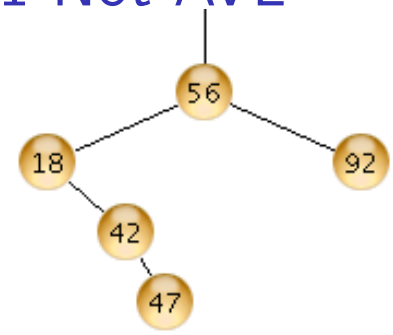


Answers

`T` is an AVL tree if and only if

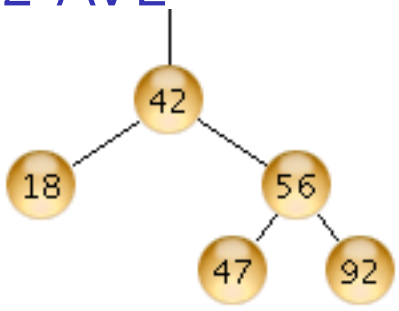
- `T.left` and `T.right` differ in height by at most 1
- AND `T.left` and `T.right` are AVL trees

1 Not AVL

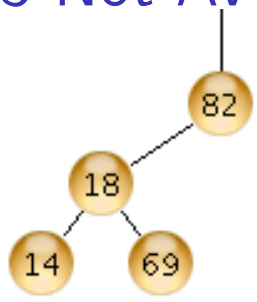


Left 0, Right 1

2 AVL

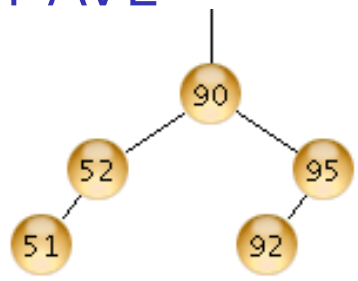


3 Not AVL

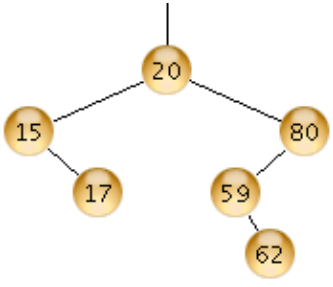


Left 2, Right 0

4 AVL

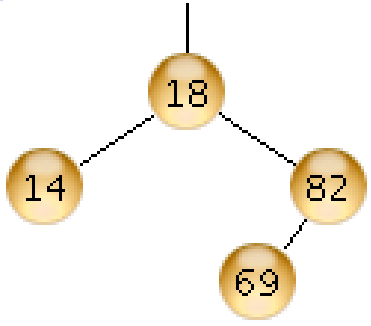


5 Not AVL

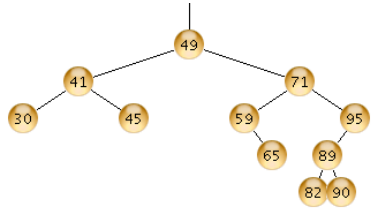


80 not AVL

6 AVL

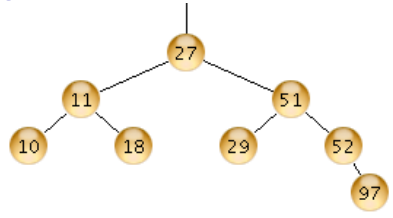


7 Not AVL



Left 2, Right 4
95 not AVL

8 AVL



Nodes and Balancing in AVL Trees

Track *Balance Factor* of trees

- `balance = height(t.left) - height(t.right);`
- Must be -1, 0, or +1 for AVL
- If -2 or +2, must fix

```
class Node<T>{
  Node<t> left,right;
  T data;
  int height;
}
```

Don't explicitly calculate `height`

- Adjust balance factor on `insert/delete`
- Recurse down to add/remove node
- Unwind recursion up to adjust balance of ancestors
- When unbalanced, rotate to adjust heights
- Single or Double rotation can *always* adjust heights by 1

Rotations

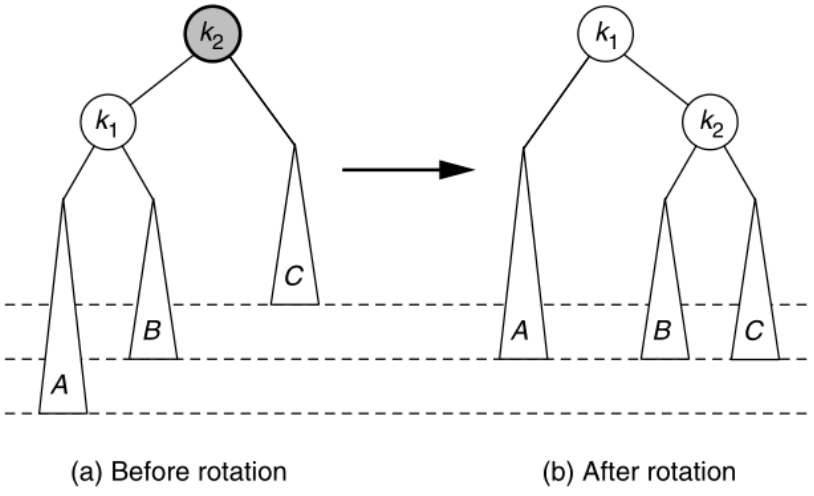
Re-balancing usually involves

- Drill down during `insert/remove`
- Follow path back up to make adjustments
- Adjustments even out height of subtrees
- Adjustments are usually rotations
- Rotation changes structure of tree without affecting ordering

Single Rotation Basics

Right Rotation

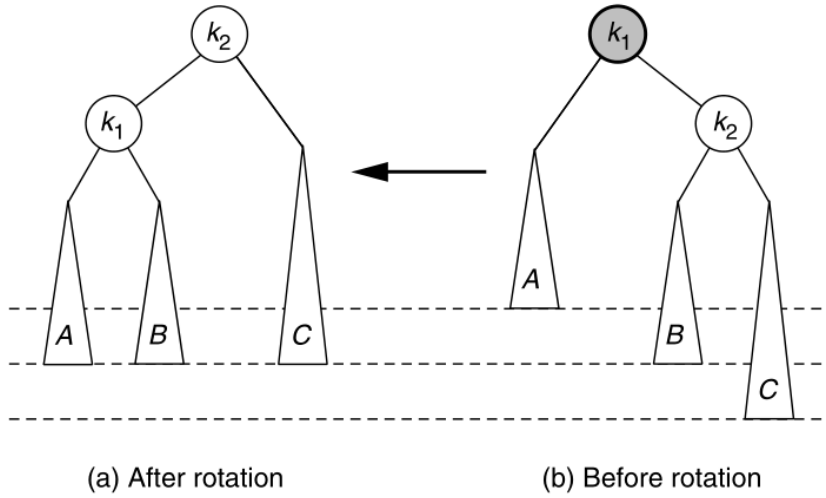
Rotation node becomes the right subtree



(a) Before rotation (b) After rotation

Left Rotation

Rotation node becomes the left subtree

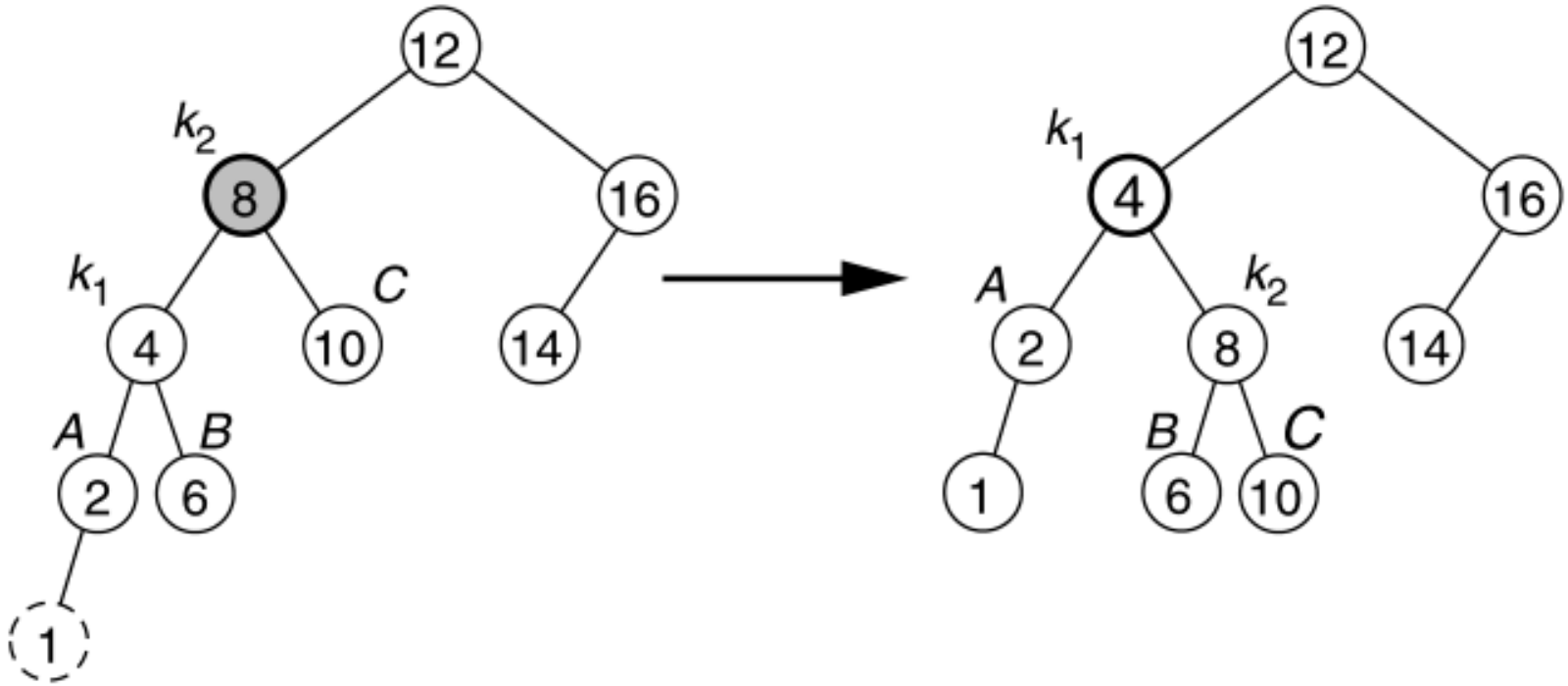


(a) After rotation (b) Before rotation

Fixing an Insertion with a Single Rotation

Insert 1, perform rotation to balance heights

- Right rotation at 8



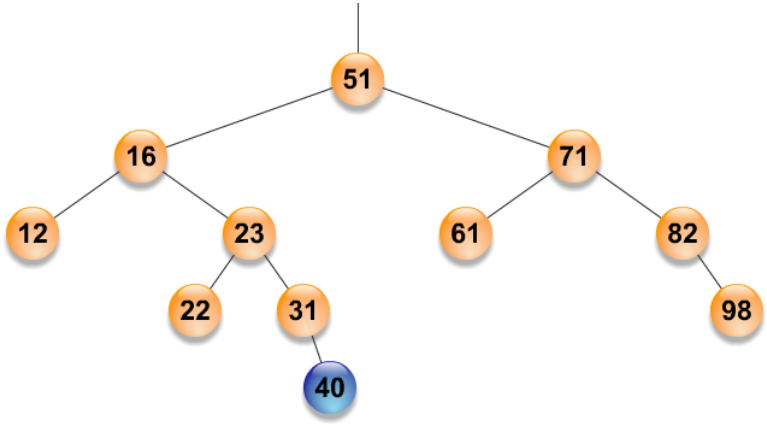
(a) Before rotation

(b) After rotation

Single Rotation Practice

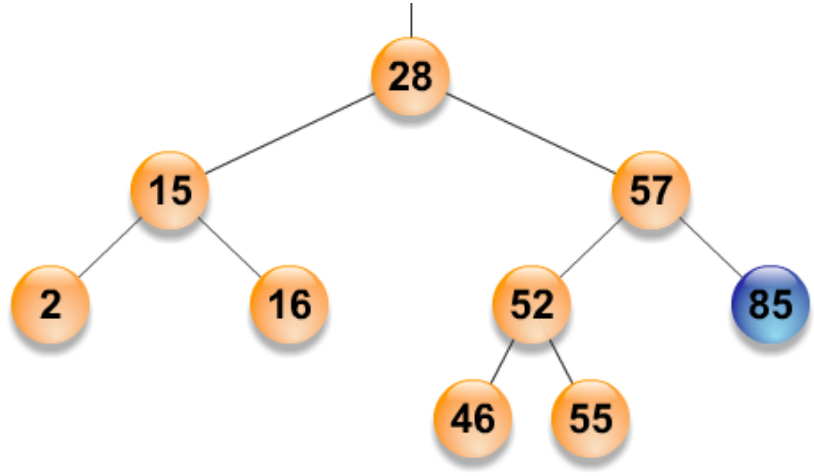
Problem 1

- 40 was just inserted
- Rebalance tree rooted at 16
- Left-rotate 16



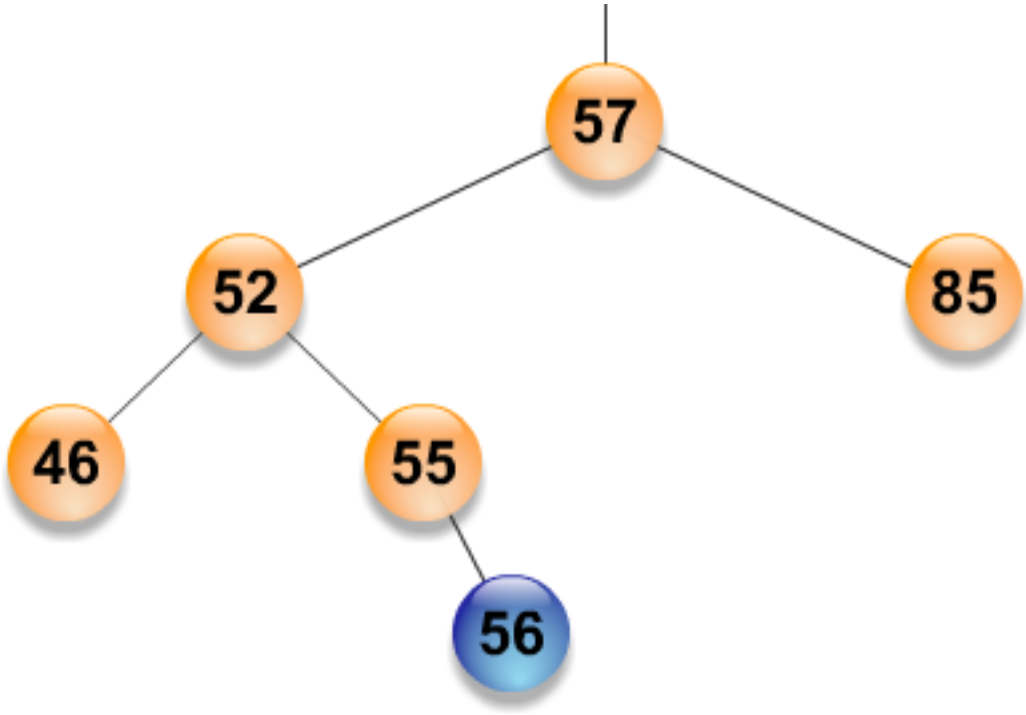
Problem 2

- 85 is being removed
- Rebalance tree rooted at 57
- Right rotate 57



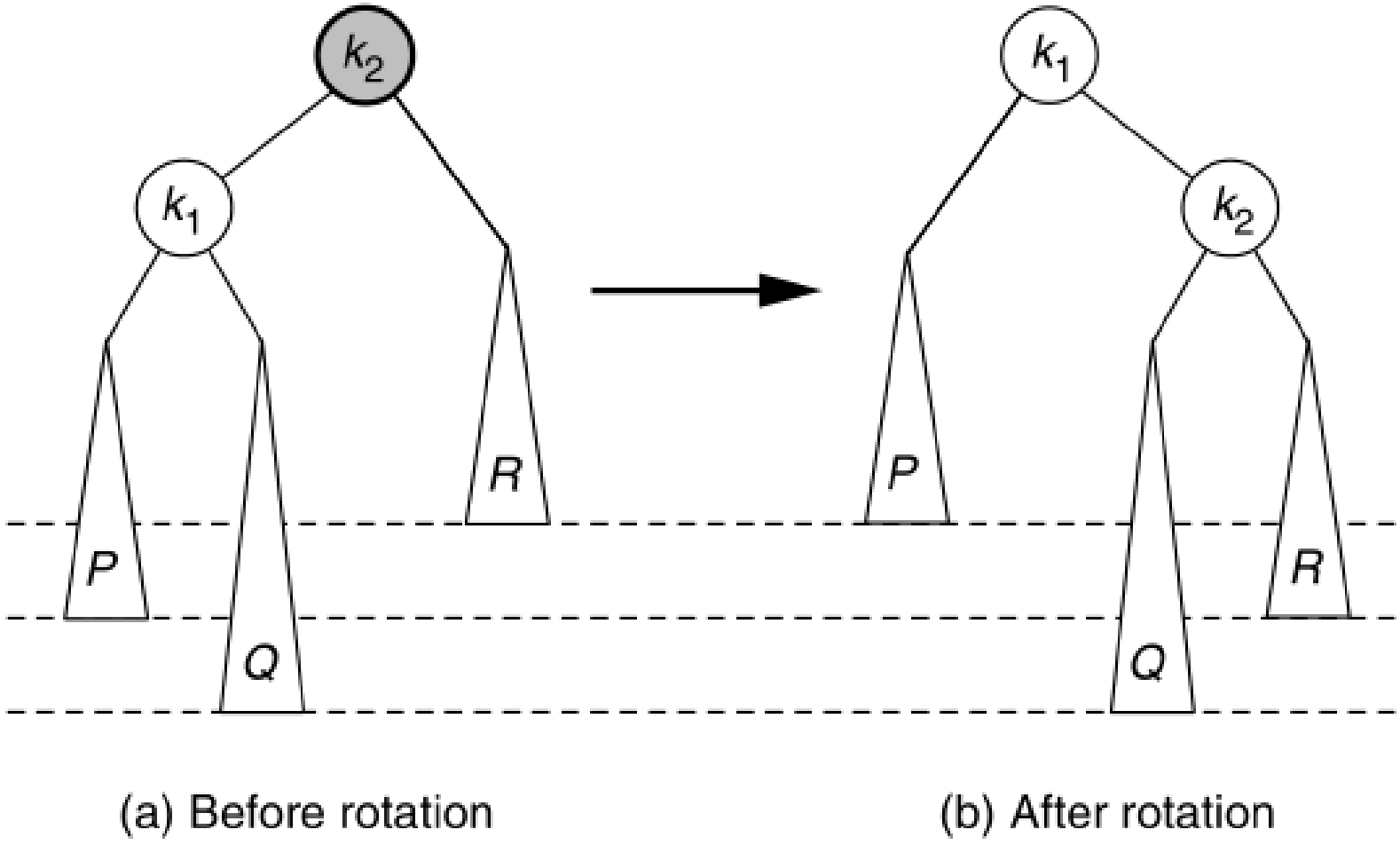
Question: Can this be fixed with single rotation?

56 was just inserted : restore AVL property with a single rotation?



Single Rotations Aren't Enough

Cannot fix following class of situations with a single rotation



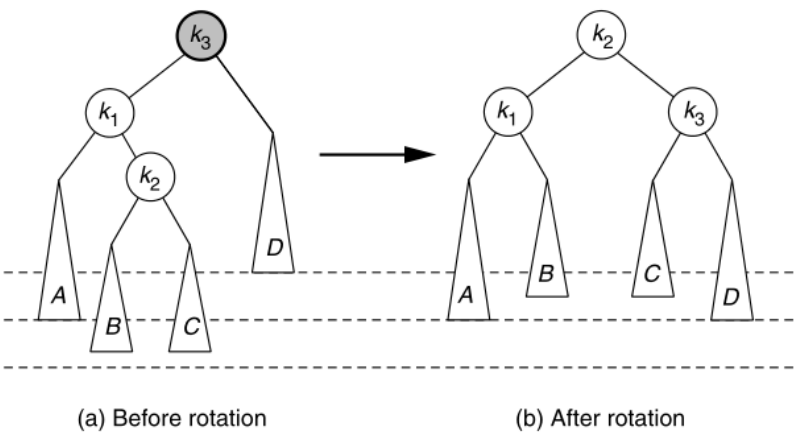
(a) Before rotation

(b) After rotation

Double Rotation Overview

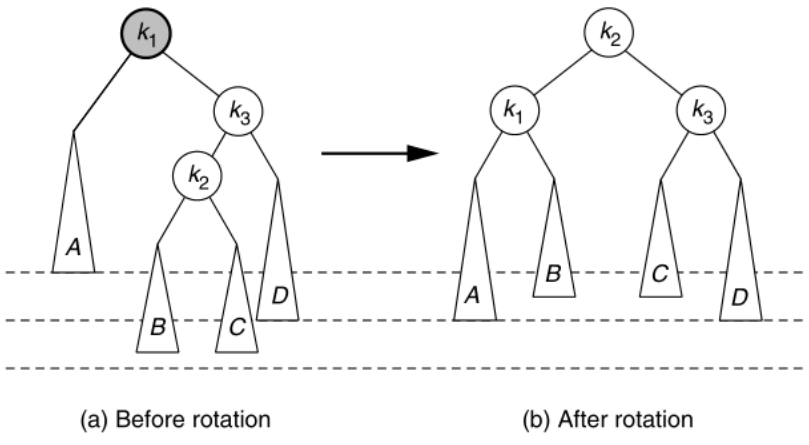
Left-Right

- Left Rotate at k_1
- Right-rotate at k_3



Right-Left

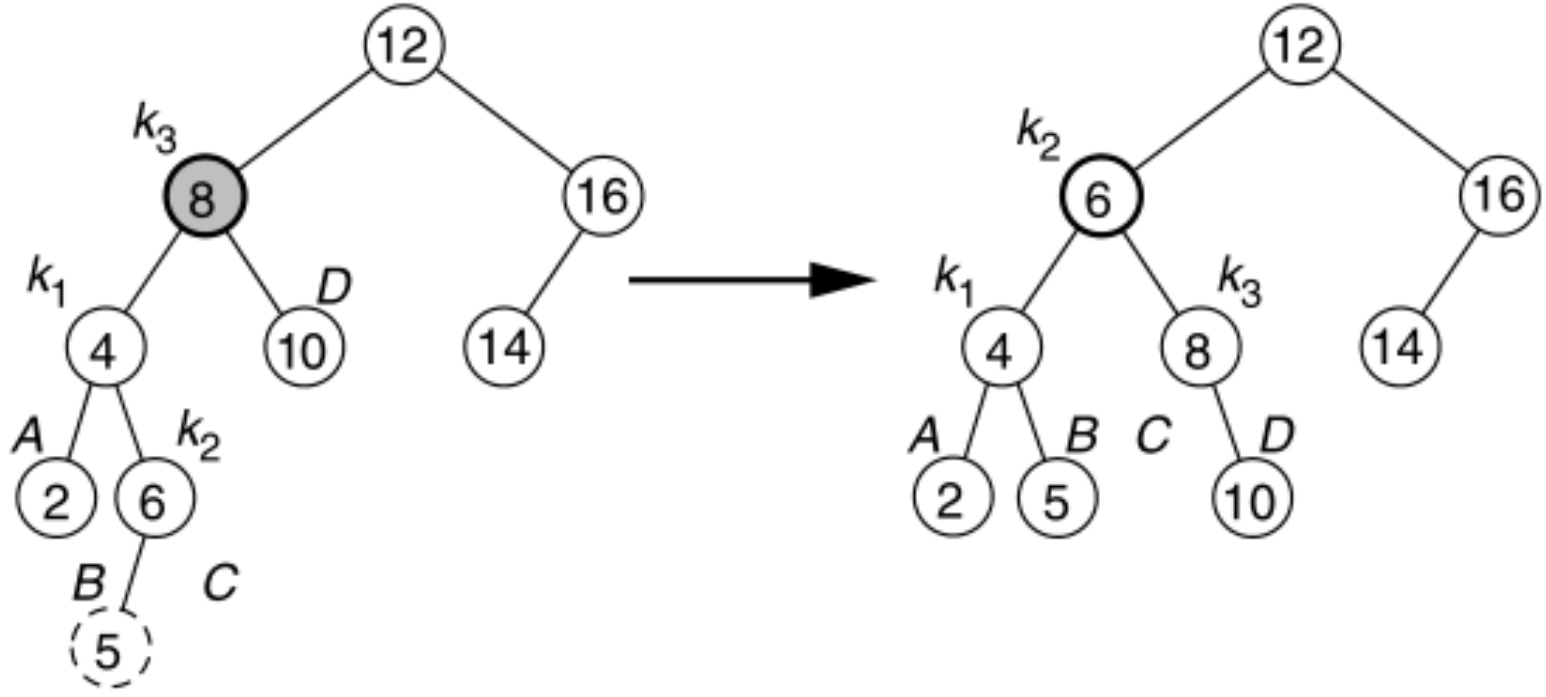
- Right Rotate at k_3
- Left Rotate at k_2



Fixing an Insertion with a Double Rotation

Insert 5, perform two rotations to balance heights

- Problem is at 8: left height 3, right height 1
- Left rotate 4 (height imbalance remains)
- Right rotate 8 (height imbalance fixed)

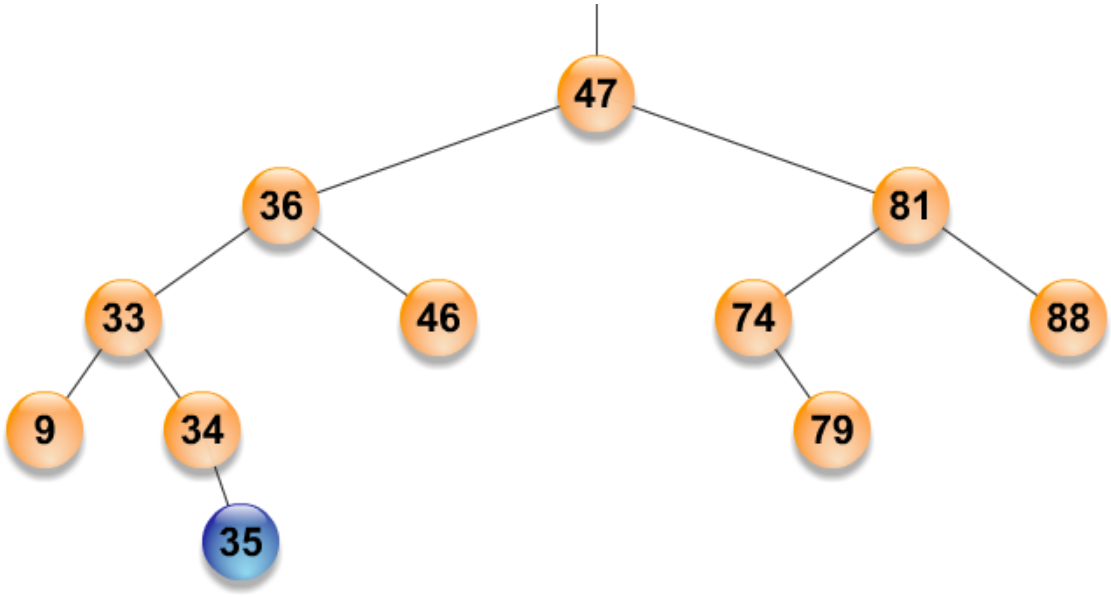


(a) Before rotation

(b) After rotation

Double Rotation Practice

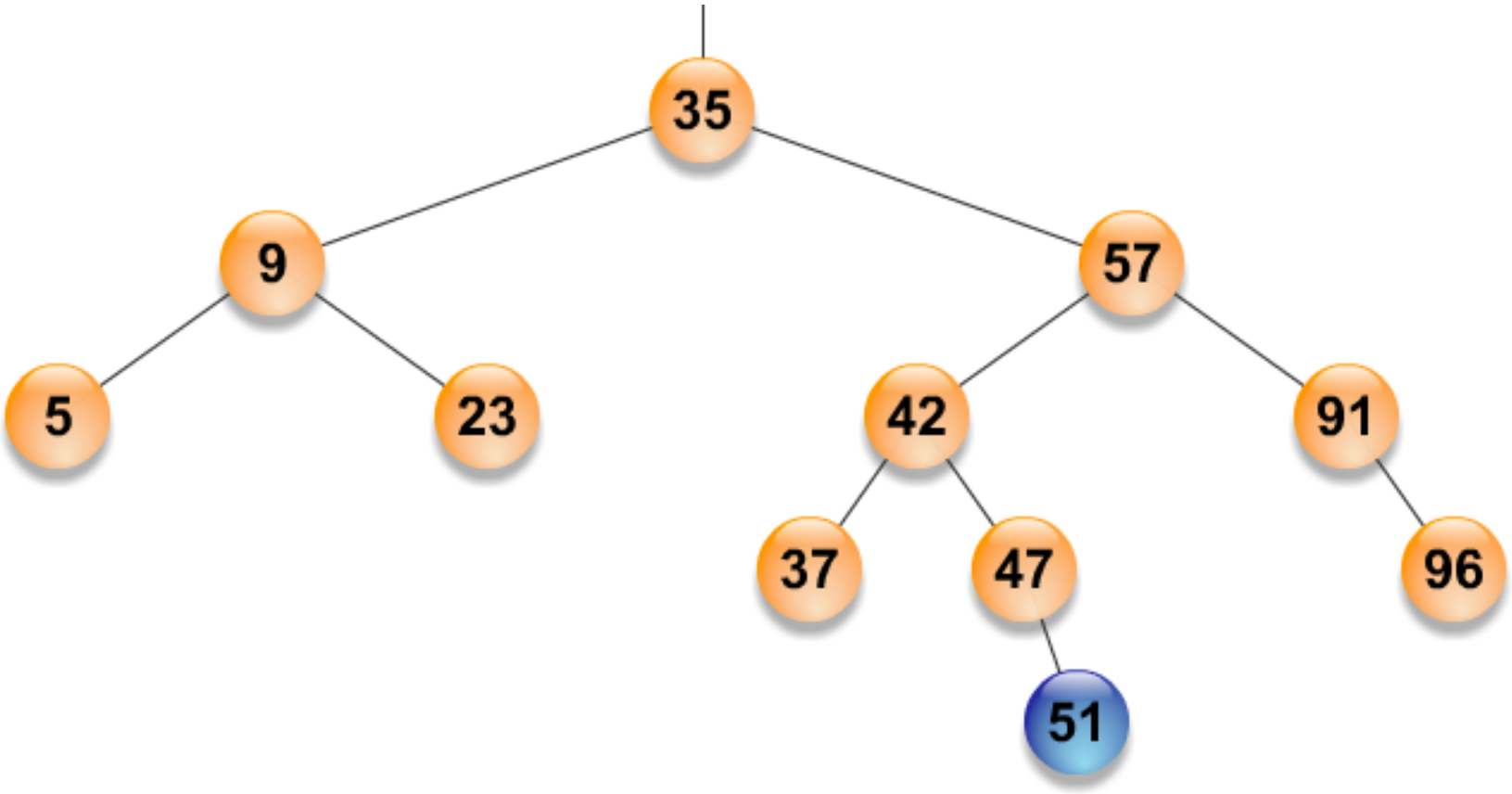
- 35 was just inserted
- Rebalance the tree rooted at 36
- Use two rotations, at 33 and 36
- 36 should move



Warm-up: BST and AVL Tree

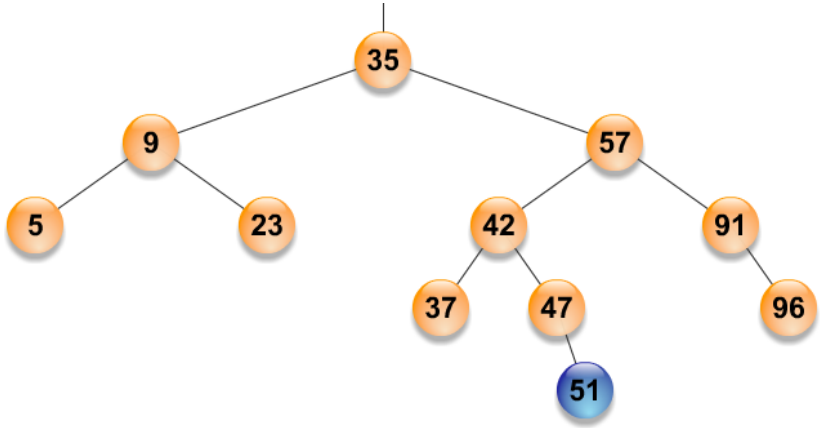
1. What is the binary search tree property?
2. What property of BSTs dictates the complexity of `find(x)`, `insert(x)`, `remove(x)`?
3. What is the memory overhead of a BST?
4. What distinguishes an AVL tree from a BST?
   - Is every AVL tree a BST?
   - Is every BST and AVL tree?
5. What kind of operation is used to maintain AVL trees during insertion/removal?

Exercise: Rebalance This AVL Tree

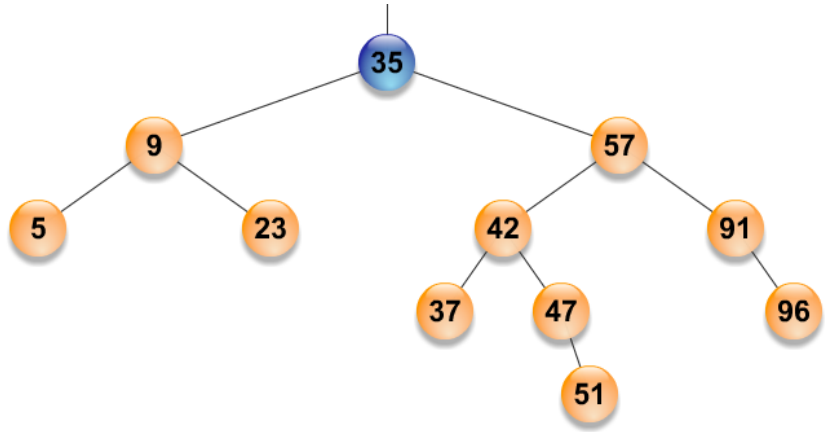


- Inserted 51
- Which node is unbalanced?
- Which rotation(s) required to fix?

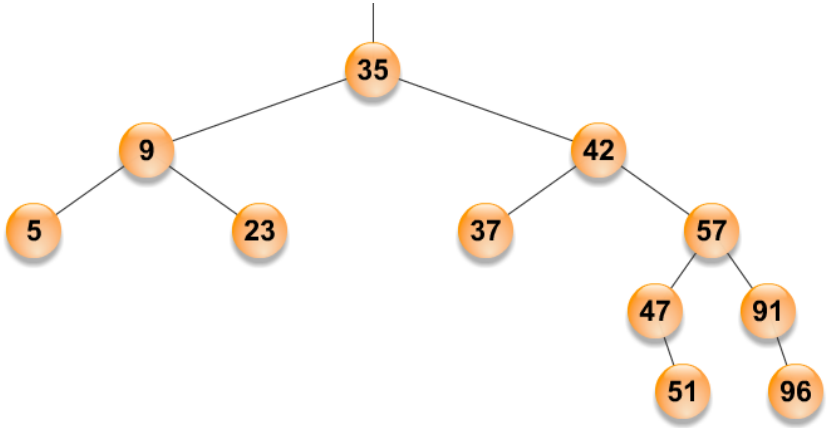
Rebalancing Answer



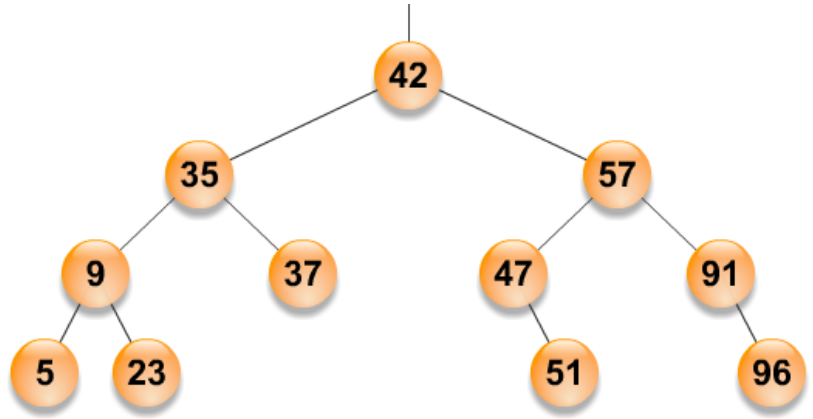
Insert 51



35 Unbalanced, inserted right-left



Right rotate 57



Left rotate 35

Code for Rotations?

```
class Node<T>{
  Node<T> left, right;
  T data;
  int height;
}
```

Write the following codes for single/double rotations:

```
// Single Right rotation
// t becomes right child, t.left becomes new
// root which is returned
Node<T> rightRotate( Node<T> t ) { ... }

// Left-Right Double Rotation:
// left-rotate t.left, then right-rotate t
Node<T> leftRightRotate( Node<T> t ){ ... }
```

Example Rotation Codes

```
// Single Right rotation
Node<T> rightRotate( Node<T> t ) {
  Node<T> newRoot = t.left;
  t.left = newRoot.right;
  newRoot.right = t;
  t.height = Math.max(t.left.height,
                      t.right.height)+1;
  newRoot.height = Math.max(newRoot.left.height,
                            newRoot.right.height)+1;
  return newRoot;
}

// Left-Right Double Rotation:
// left-rotate t.left, then right-rotate t
Node<T> leftRightRotate( Node<T> t ){
  t.left = leftRotate(t.left);
  return rightRotate(t);
}
```

Computational complexities of these methods?

Rotations During Insertion

- Insertion works by first recursively inserting new data as a leaf
- Tree is "unstitched" - waiting to assign left/right branches of intermediate nodes to answers from recursive calls
- Before returning, check height differences and perform rotations if needed
- Allows left/right branches to change the nodes to which they point

Double or Single Rotations?

- Insert / remove code needs to determine rotations required
- Can simplify this into 4 cases

Tree `T` has left/right imbalance after `insert(x)` / `remove(x)`

Zig-Zig `T.left > T.right+1` and
`T.left.left > T.left.right`
Single Right Rotation at `T`

Zag-Zag `T.right > T.left+1`
`T.right.right > T.right.left`
Single Left Rotation at `T`

Zig-ZAG `T.left > T.right+1` and
`T.left.right > T.left.left`
Double Rotation: left on `T.left`, right on `T`

Zag-Zig `T.right > T.left+1` and
`T.right.left > T.right.right`
Double Rotation: right on `T.right`, left on `T`

Excerpt of Insertion Code

From old version of Weiss `AvlTree.java`, in this week's codepack

- Identify subtree height differences to determine rotations
- Useful in removal as well

```
private AvlNode insert( Comparable x, AvlNode t ){
  if( t == null ){                                  // Found the spot to insert
    t = new AvlNode( x, null, null );               // return new node with data
  }
  else if( x.compareTo( t.element ) < 0 ) {         // Head left
    t.left = insert( x, t.left );                   //   Recursively insert
  } else{                                           // Head right
    t.right = insert( x, t.right );                 //   Recursively insert
  }                                                 //
  if(height(t.left) - height(t.right) == 2){        // t.left deeper than t.right
    if(height(t.left.left) > t.left.right) {        // outer tree unbalanced
      t = rightRotate( t );                         //   single rotation
    } else {                                        // x went left-right:
      t = leftRightRotate( t );                     //   double rotation
    }
  }
  else{ ... } // Symmetric cases for t.right deeper than t.left
  return t;
```

Does This Accomplish our Goal?

- Runtime complexity for BSTs is `find(x)`, `insert(x)`, `remove(x)` is $O(Height)$
- Proposition: Maintaining the AVL Balance Property during `insert/remove` will yield a tree with N nodes and Height $O(\log N)$
- Proving this means AVL trees have $O(\log N)$ operations
- Prove it: What do AVL trees have to do with rabbits?

AVL Properties Give $log(N)$ height

Lemma (little theorem) *(Thm 19.3 in Weiss, pg 708, adapted)*

An AVL Tree of height H has at least $F_{H+2} - 1$ nodes where F_i is the *ith* Fibonacci number.

Definitions

- F_i: *ith* Fibonacci number (0,1,1,2,3,5,8,13,...)
- S: size of a tree
- H: height (assume roots have height 1)
- S_H is the smallest size AVL Tree with height H

Proof by Induction: Base Cases True

Tree	height	Min Size	Calculation
empty	$H = 0$	S_0	$F_{(0+2)} - 1 = 1 - 1 = 0$
root	$H = 1$	S_1	$F_{(1+2)} - 1 = 2 - 1 = 1$
root+(left or right)	$H = 2$	S_2	$F_{(2+2)} - 1 = 3 - 1 = 2$

Inductive Case Part 1

Consider an Arbitrary AVL tree T

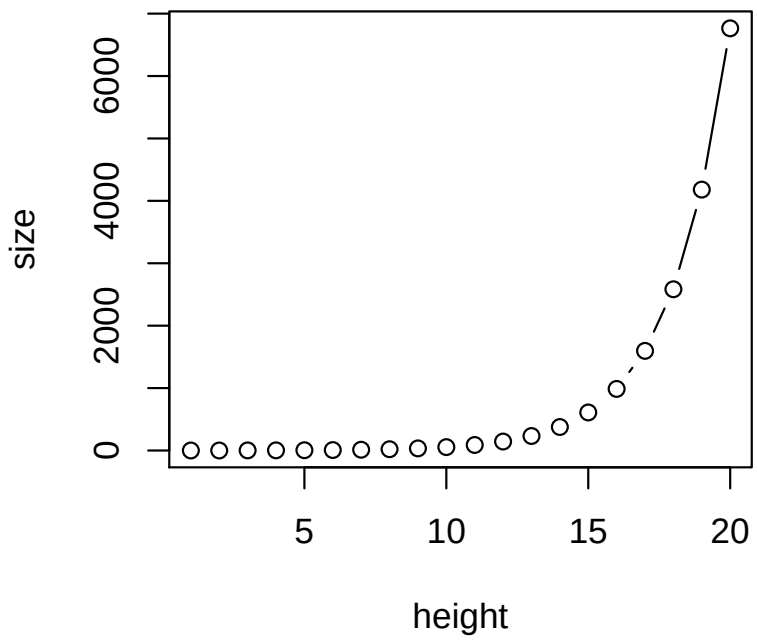
- T has height H
- S_H smallest size for tree T
- Assume equation true for smaller trees
  - Notice: Left/Right are smaller AVL trees
  - Notice: Left/Right differ in height by at most 1

Inductive Case Part 2

- T has height H
- Assume for height $h < H$, smallest size of T is $S_h = F_{h+2} - 1$
- Suppose Left is 1 higher than Right
- Left Height: $h = H - 1$
- Left Size: $F_{(H-1)+2} - 1 = F_{H+1} - 1$
- Right Height: $h = H - 2$
- Right Size: $F_{(H-2)+2} - 1 = F_H - 1$

$$\begin{aligned} S_H &= size(Left) + size(Right) + 1 \\ &= (F_{H+1} - 1) + (F_H - 1) + 1 \\ &= F_{H+1} + F_H - 1 \\ &= F_{H+2} - 1 \ \blacksquare \end{aligned}$$

Fibonacci Growth



AVL Tree of with height H has at least $F_{H+2} - 1$ nodes.

- How does F_H grow wrt H?
- Exponentially:
  $F_H \approx \phi^H = 1.618^H$
- ϕ: The Golden Ratio
- So, $\log(F_H) \approx H \log(\phi)$
- Or, $\log(N) \approx \mathit{height} \times \phi$
- Or,
  $\log(\mathit{size}) \approx \mathit{height} * \mathit{constant}$