CPSC 221: Algorithms and Data Structures
Lecture #6
Balancing Act

Steve Wolfman

2014W1

Today's Outline

- Addressing one of our problems
- Single and Double Rotations
- AVL Tree Implementation

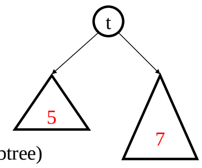
Beauty is Only Θ(log n) Deep

- Binary Search Trees are fast if they're shallow:
  - perfectly complete
  - perfectly complete except the one level fringe (like a heap)
  - anything else?

What *matters* here?

Problems occur when one subtree is much taller than the other!

Balance

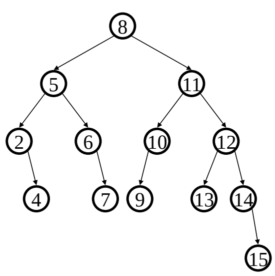


- Balance
  - height(left subtree) - height(right subtree)
  - zero everywhere ⇒ perfectly balanced
  - small everywhere ⇒ balanced enough

Balance between -1 and 1 everywhere ⇒ maximum height of ~1.44 lg n

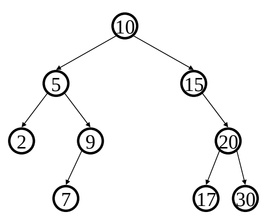
AVL Tree
Dictionary Data Structure

- Binary search tree properties
  - binary tree invariant
  - search tree invariant
- Balance invariant
  - balance of every node is: $-1 \le b \le 1$
  - result:
    - depth is Θ(log n)



Note that (statically… ignoring how it updates) an AVL tree *is* a BST.

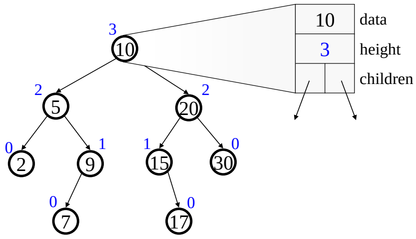
Testing the Balance Property



NULLs have height **-1**

FIRST calculate heights
THEN calculate balances

An AVL Tree



Today's Outline

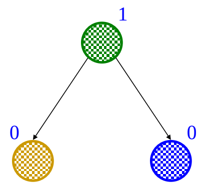
- Addressing one of our problems
- Single and Double Rotations
- AVL Tree Implementation

But, How Do We Stay Balanced?

Need: three volunteers proud of their very diverse height.

Beautiful Balance (SIMPLEST version)

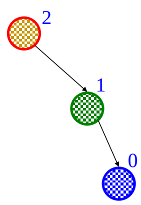
Insert(middle)
Insert(small)
Insert(tall)



But… BSTs are under-constrained in unfortunate ways; ours may not look like this.

Bad Case #1 (SIMPLEST version)

Insert(small)
Insert(middle)
Insert(tall)



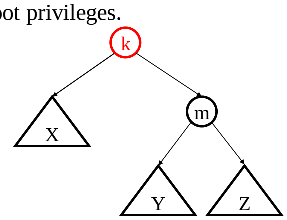
How do we fix the bad case?
How do we transition among different possible trees?

So you say you want a revolution?

Scenario: k is one of the 1%; we want a revolution to depose k and give m root privileges.

What do we do?

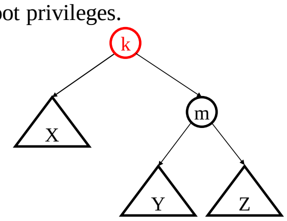
Well, first…



Well.. what do we know?

Scenario: k is one of the 1%; we want a revolution to depose k and give m root privileges.

What's everything we know about nodes k and m and subtrees X, Y, and Z?

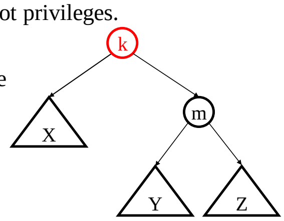


13

A real solution?

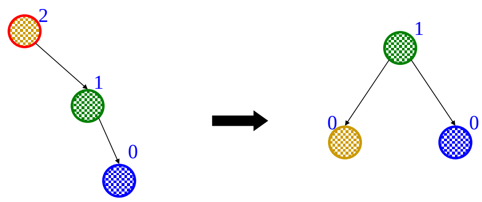
Scenario: k is one of the 1%; we want a revolution to depose k and give m root privileges.

Now, how can we make m the root?



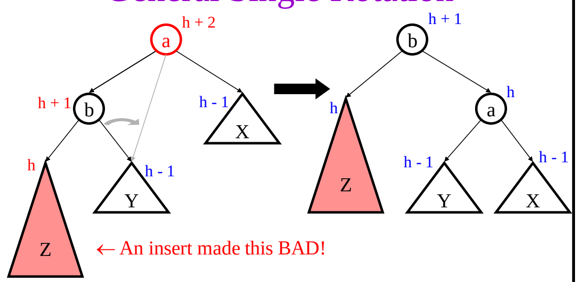
14

Single Rotation (SIMPLEST version)



15

General Single Rotation



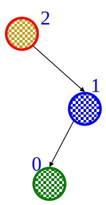
← An insert made this BAD!

- After rotation, subtree's height same as before insert!
- Height of all ancestors unchanged.

Why couldn't the bad insert be in X?
Why does it matter that ancestors stay the same?

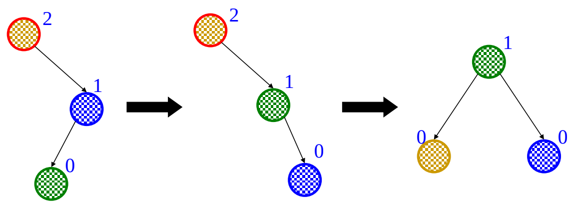
Bad Case #2 (SIMPLEST version)

Insert(small)
Insert(tall)
Insert(middle)



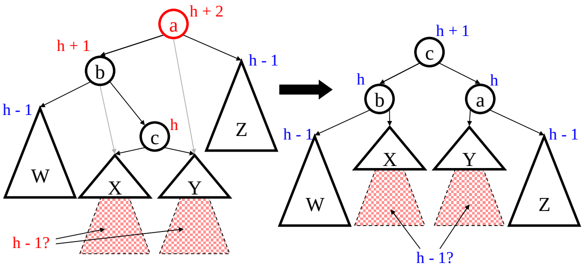
17

Double Rotation (SIMPLEST version)



18

General Double Rotation



- Height of subtree **still** the same as it was before insert!
- Height of all ancestors unchanged.

19

Today’s Outline

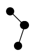
- Addressing one of our problems
- Single and Double Rotations
- AVL Tree Implementation

20

Insert Algorithm

- Find spot for value
- Hang new node
- Search back up for imbalance
- If there is an imbalance:


case #1: Perform single rotation and exit

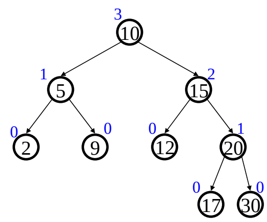

case #2: Perform double rotation and exit

Mirrored cases also possible

21

Easy Insert

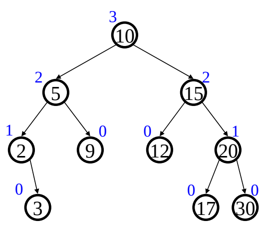
Insert(3)



22

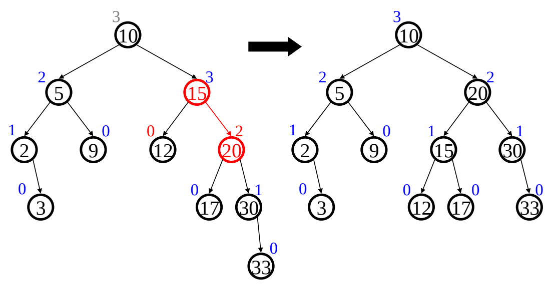
Hard Insert (Bad Case #1)

Insert(33)



23

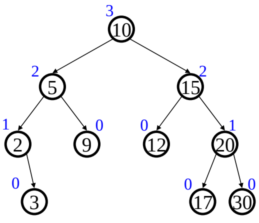
Single Rotation



24

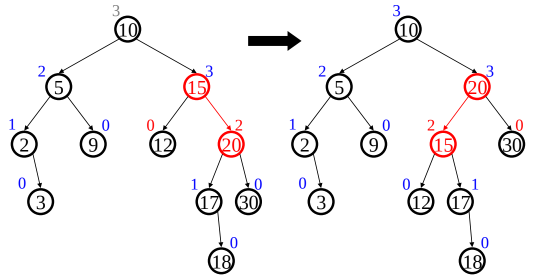
Hard Insert (Bad Case #2)

Insert(18)



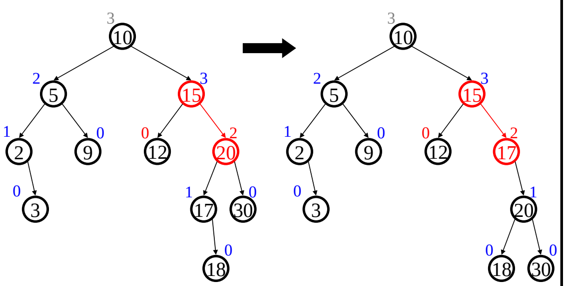
25

Single Rotation (oops!)



26

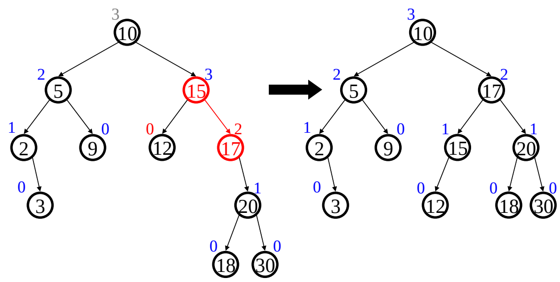
Double Rotation (Step #1)



Look familiar?

27

Double Rotation (Step #2)



28

AVL Algorithm Revisited

- Recursive

1. Search downward for spot
2. Insert node
3. Unwind stack, correcting heights
   a. If imbalance #1, single rotate
   b. If imbalance #2, double rotate

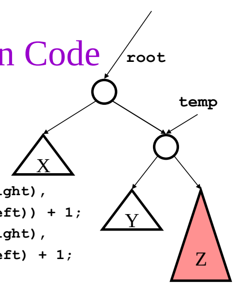
- Iterative

1. Search downward for spot, stacking parent nodes
2. Insert node
3. Unwind stack, correcting heights
   a. If imbalance #1, single rotate and exit
   b. If imbalance #2, double rotate and exit

29

Single Rotation Code

```
void RotateLeft(Node *& root) {
  Node * temp = root->right;
  root->right = temp->left;
  temp->left = root;
  root->height = max(height(root->right),
                     height(root->left)) + 1;
  temp->height = max(height(temp->right),
                     height(temp->left) + 1;
  root = temp;
}
```

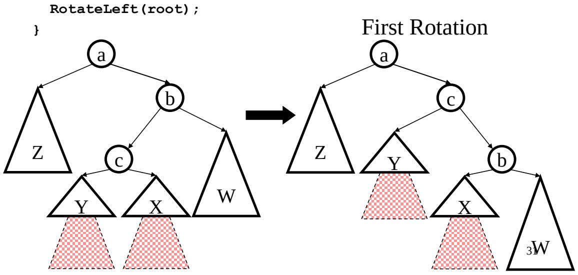


(The “height” function returns -1 for a NULL subtree or the “height” field of a non-NULL subtree.)

30

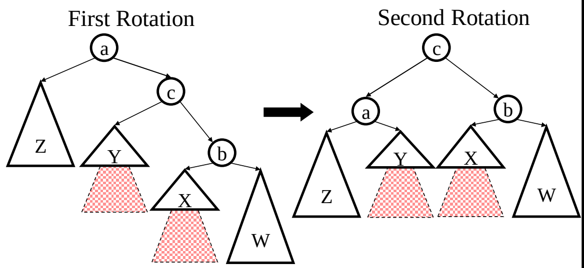
Double Rotation Code

```
void DoubleRotateLeft(Node *& root) {
  RotateRight(root->right);
  RotateLeft(root);
}
```



31

Double Rotation Completed



32

AVL

- Automatically Virtually Leveled
- Architecture for inVisible Leveling (the "in" is inVisible)
- All Very Low
- Articulating Various Lines
- Amortizing? Very Lousy!
- Absolut Vodka Logarithms
- Amazingly Vexing Letters

Adelson-Velskii Landis

33

To Do

- Posted readings on priority queues and sorts, but de-emphasizing heaps

34

Coming Up

- Quick Jump Back to Priority Queue ADT
  - A very familiar data structure gives us O(lg n) performance!
- On to sorts
- Then forward again!

35