CSE 326: Data Structures

Topic #7: AVL Trees

Ashish Sabharwal

Autumn, 2003

Today's Outline

- Quiz #2
- Note: Chapter 4 has quite a few corrections!
  See errata.
- **Balance** in Binary Search Trees
- **AVL Trees**

2

Balanced BST

Observation
- BST: the shallower the better!
- For a BST with *n* nodes
  - Average height is $\Theta(\log n)$
  - Worst case height is $\Theta(n)$
- Simple cases such as insert(1, 2, 3, ..., n) lead to the worst case scenario

Solution: Require a **Balance Condition** that
1. ensures depth is $\Theta(\log n)$ – strong enough!
2. is easy to maintain – not too strong!

3

Potential Balance Conditions

1. Left and right subtrees of the root have equal number of nodes

2. Left and right subtrees of the root have equal *height*

4

Potential Balance Conditions

3. Left and right subtrees of *every node* have equal number of nodes

4. Left and right subtrees of *every node* have equal *height*

5

The AVL Balance Condition

Left and right subtrees of every node have heights differing by at most 1

Define: **balance**(x) = height(x.left) – height(x.right)

AVL property: **$-1 \leq$ balance(x) ≤ 1, for every node x**

- Ensures small depth
  - Will prove this by showing that an AVL tree of height h must have a lot of (i.e. $\Theta(2^h)$) nodes
- Easy to maintain
  - Using single and double rotations

6

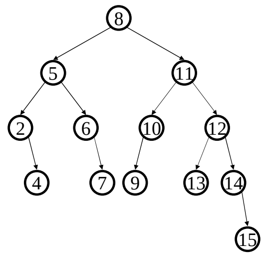
The AVL Tree Data Structure

Structural properties
1. Binary tree property
2. Balance property: balance of every node is between -1 and 1

Result:

Worst case depth is $\Theta(\log n)$

Ordering property
– Same as for BST



7

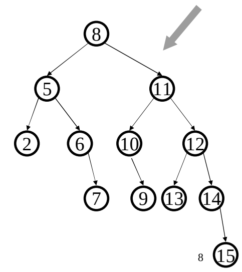
Proving Shallowness Bound

Let **S**(h) be the min # of nodes in an AVL tree of height h

Claim: **S**(h) = **S**(h-1) + **S**(h-2) + 1

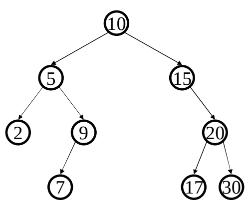
Solution of recurrence: **S**(h) = $\Theta(2^h)$ (like Fibonacci numbers)

AVL tree of height h=4 with the min # of nodes



8

Testing the Balance Property



We need to be able to:

1.

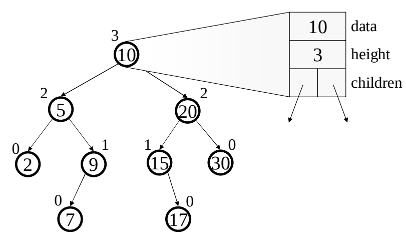
2.

3.

`NULL`s have height `-1`

9

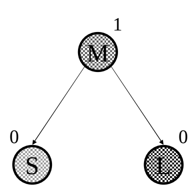
An AVL Tree



10

Beautiful Balance

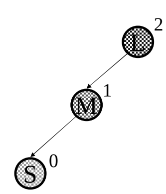
Insert(mid)
Insert(small)
Insert(large)



11

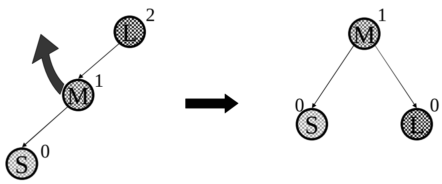
Bad Case #1

Insert(large)
Insert(mid)
Insert(small)



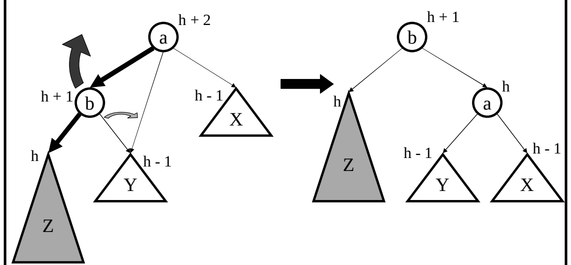
12

Fix: Apply Single Rotation



13

General Single Rotation

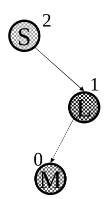


- Are the blue and red heights the only possibilities? Yes!
- Height of this whole tree same as it was before insert!
- Height of all ancestors unchanged. So?

14

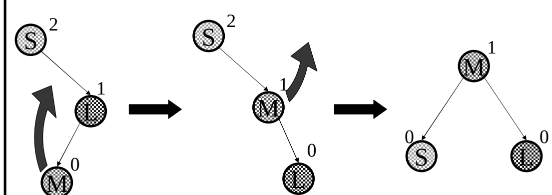
Bad Case #2

Insert(small)
Insert(large)
Insert(mid)



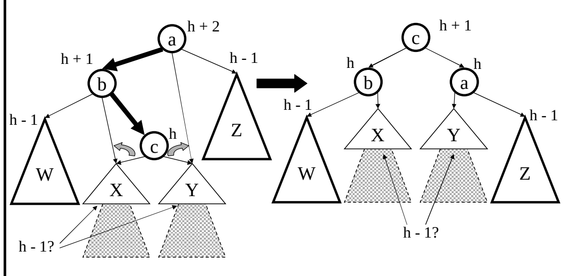
15

Fix: Apply Double Rotation



16

General Double Rotation

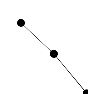


- Are the blue and red heights the only possibilities? Yes!
- Height of subtree **still** the same as it was before insert!

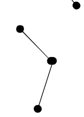
17

Great! Now we can fix imbalance!

- Single rotation for the "zig-zig" case



- Double rotation for the "zig-zag" case



Both rotations keep the subtree height unchanged.
Hence only one rotation is sufficient!

18

So what does AVL mean anyway??

Let's vote!!

- Automatically Virtually Leveled
- Architecture for inVisible Leveling (the "in" is inVisible)
- All Very Low
- Absolut Vodka Logarithms
- Amazingly Vexing Letters

19

AVL Tree Operations

- Find(*x*)
- Insert(*x*)
- Delete(*x*)

} Θ(log *n*)

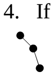
- buildTree

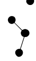
} Θ(*n* log *n*)

20

Insertion into AVL tree

1. Find spot for new key
2. Hang new node there with this key
3. Search back up the path for imbalance
4. If there is an imbalance:


case #1: Perform single rotation and exit

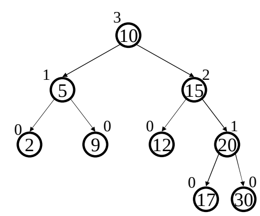

case #2: Perform double rotation and exit

Should we loop to fix all problems?

21

Easy Insert

Insert(3)

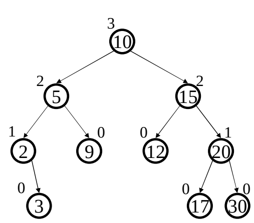


Unbalanced?

22

Hard Insert (Bad Case #1)

Insert(33)



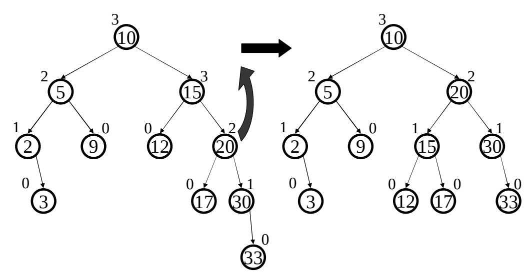
Unbalanced?

How to fix?

How did we know?

23

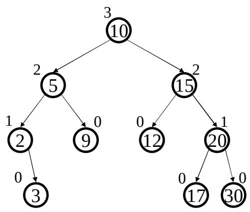
Single Rotation



24

Hard Insert (Bad Case #2)

Insert(18)



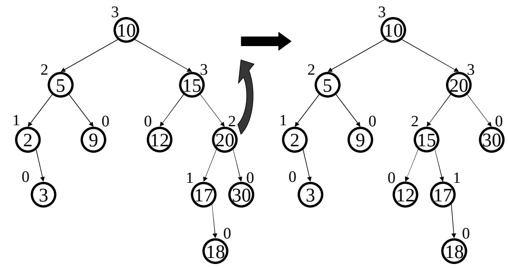
Unbalanced?

How to fix?

How did we know?

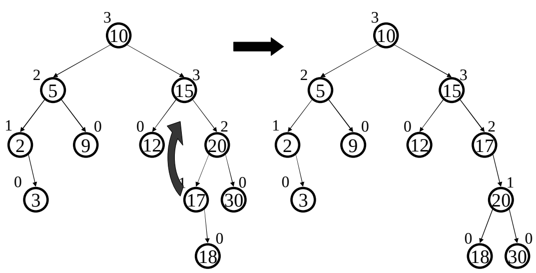
25

Single Rotation (oops!)



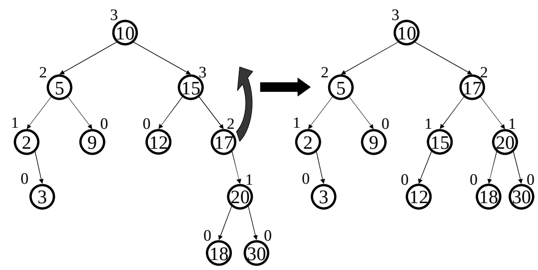
26

Double Rotation (Step #1)



27

Double Rotation (Step #2)



28

AVL Insert Algorithm Revisited

Recursive

```
1. Search downward for
   spot
2. Insert node
3. On the way back,
   correct 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
```

Why use a stack?

29

Deletion in AVL Tree

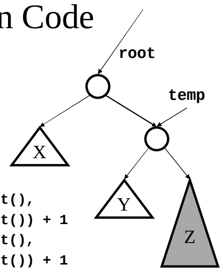
Recall deletion in BST:

- What's the order change in the tree?
  - Can this affect balance?
- What's the structural change?
  - Can this affect balance?

30

Single Rotation Code

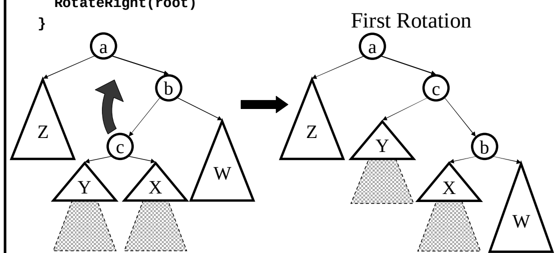
```
void RotateRight(Node root) {
  Node temp = root.right
  root.right = temp.left
  temp.left = root
  root.height = max(root.right.height(),
                    root.left.height()) + 1
  temp.height = max(temp.right.height(),
                    temp.left.height()) + 1
  root = temp
}
```



31

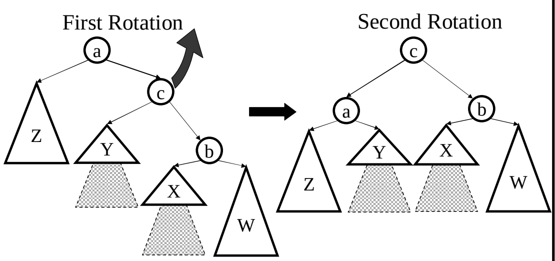
Double Rotation Code

```
void DoubleRotateRight(Node root) {
  RotateLeft(root.right)
  RotateRight(root)
}
```



32

Double Rotation Completed



33

To Do

- Written homework #1
- Read chapter 4

34