CSE 332: Data Abstractions

Lecture 7: AVL Trees

Ruth Anderson

Winter 2013

Announcements

- **Project 2** – posted!
- **Homework 2** – due Friday Jan 25th at ***beginning*** of class, see clarifications posted

Today

- Dictionaries
  - AVL Trees

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 ≤ 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

Note: height of a null tree is -1, height of tree with a single node is 0

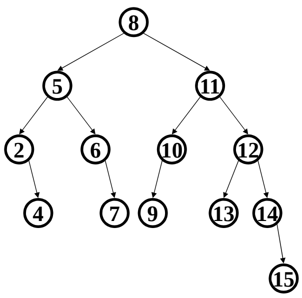
The AVL Tree Data Structure

Structural properties

1. Binary tree property (0,1, or 2 children)
2. Heights of left and right subtrees of *every node* **differ by at most 1**

Result:

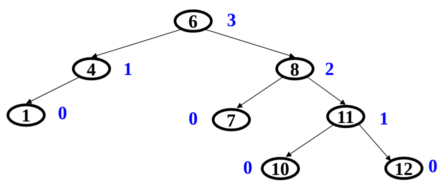
Worst case depth of any node is: O(log n)



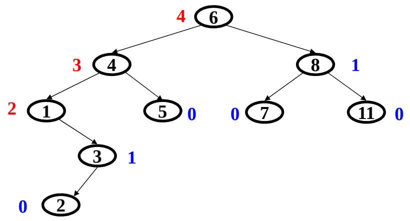
Ordering property

– Same as for BST

An AVL tree?



An AVL tree?



Height of an AVL Tree?

Using the AVL balance property, we can determine the minimum number of nodes in an AVL tree of height *h*

Let **S**(*h*) be the minimum # of nodes in an AVL tree of height *h*, then:

$$\mathbf{S}(h) = \mathbf{S}(h-1) + \mathbf{S}(h-2) + 1$$

where $\mathbf{S}(-1) = 0$ and $\mathbf{S}(0) = 1$

Solution of Recurrence: $\mathbf{S}(h) \approx 1.62^h$

Let **S**(*h*) be the <u>minimum</u> # of nodes in an AVL tree of height *h*, then:

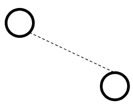
$$\mathbf{S}(h) = \mathbf{S}(h-1) + \mathbf{S}(h-2) + 1$$

where $\mathbf{S}(-1)=0$ and $\mathbf{S}(0)=1$

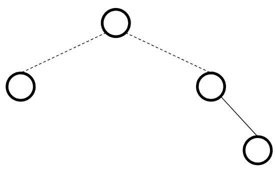
h	***S(h)***

Minimal AVL Tree (height = 0)

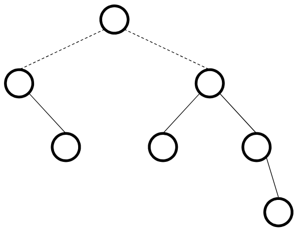
Minimal AVL Tree (height = 1)



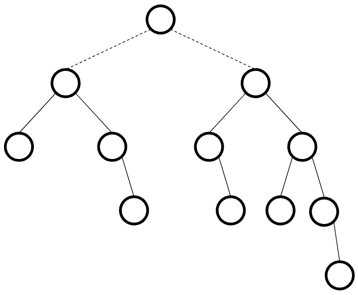
Minimal AVL Tree (height = 2)



Minimal AVL Tree (height = 3)



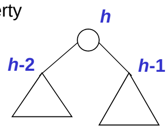
Minimal AVL Tree (height = 4)



The shallowness bound

Let $S(h)$ = the minimum number of nodes in an AVL tree of height h

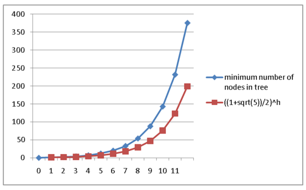
- If we can prove that $S(h)$ grows exponentially in h, then a tree with n nodes has a logarithmic height

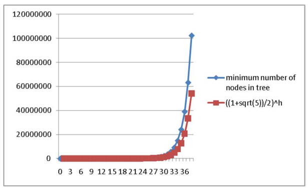


- Step 1: Define $S(h)$ inductively using AVL property
  - $S(-1)=0$, $S(0)=1$, $S(1)=2$
  - *For* $h \geq 1$, $S(h) = 1+S(h-1)+S(h-2)$
- Step 2: Show this recurrence grows really fast
  - Similar to Fibonacci numbers
  - Can prove for all h, $S(h) > \phi^h - 1$ where ϕ is the golden ratio, $(1+\sqrt{5})/2$, about 1.62
  - Growing faster than 1.6^h is "plenty exponential"

Before we prove it

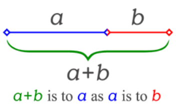
- Good intuition from plots comparing:
  - $S(h)$ computed directly from the definition
  - $((1+\sqrt{5})/2)^h$
- $S(h)$ is always bigger, up to trees with huge numbers of nodes
  - Graphs aren't proofs, so let's prove it





The Golden Ratio

$$\phi = \frac{1+\sqrt{5}}{2} \approx 1.62$$



This is a special number

- Aside: Since the Renaissance, many artists and architects have proportioned their work (e.g., length:height) to approximate the ***golden ratio***: If `(a+b)/a = a/b`, then `a = φb`
- We will need one special arithmetic fact about ϕ :

$$\begin{aligned}
\phi^2 &= ((1+5^{1/2})/2)^2 \\
&= (1 + 2*5^{1/2} + 5)/4 \\
&= (6 + 2*5^{1/2})/4 \\
&= (3 + 5^{1/2})/2 \\
&= 1 + (1 + 5^{1/2})/2 \\
&= 1 + \phi
\end{aligned}$$

The proof

$S(-1)=0$, $S(0)=1$, $S(1)=2$
For $h \geq 1$, $S(h) = 1+S(h-1)+S(h-2)$

Theorem: For all $h \geq 0$, $S(h) > \phi^h - 1$
Proof: By induction on h
Base cases:

$S(0) = 1 > \phi^0 - 1 = 0$ $\quad$ $S(1) = 2 > \phi^1 - 1 \approx 0.62$

Inductive case ($k > 1$):

Show $S(k+1) > \phi^{k+1} - 1$ assuming $S(k) > \phi^k - 1$ and $S(k-1) > \phi^{k-1} - 1$

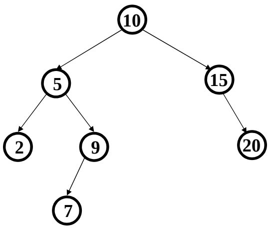
$S(k+1) = 1 + S(k) + S(k-1)$	by definition of S
$> 1 + \phi^k - 1 + \phi^{k-1} - 1$	by induction
$= \phi^k + \phi^{k-1} - 1$	by arithmetic (1-1=0)
$= \phi^{k-1} (\phi + 1) - 1$	by arithmetic (factor ϕ^{k-1})
$= \phi^{k-1} \phi^2 - 1$	by special property of ϕ
$= \phi^{k+1} - 1$	by arithmetic (add exponents)

Good news

Proof means that if we have an AVL tree, then **`find`** is *O*(**`log`** *n*)

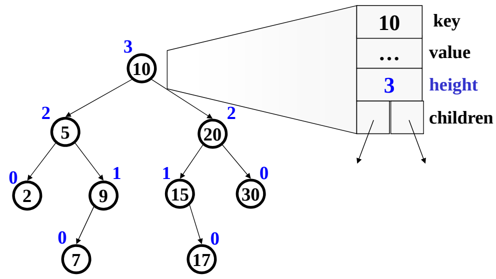
But as we **insert** and **delete** elements, we need to:

1. Track balance
2. Detect imbalance
3. Restore balance



Is this tree AVL balanced?
How about after **`insert(30)`**?

An AVL Tree



AVL tree operations

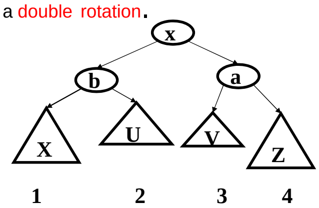
- **AVL `find`**:
  - Same as BST **`find`**
- **AVL `insert`**:
  - First BST **`insert`**, *then* check balance and potentially "fix" the AVL tree
  - Four different imbalance cases
- **AVL `delete`**:
  - The "easy way" is lazy deletion
  - Otherwise, like insert we do the deletion and then have several imbalance cases

AVL tree insert

Let *x* be the node where an imbalance occurs.
Four cases to consider. The insertion is in the

1. left subtree of the left child of *x*.
2. right subtree of the left child of *x*.
3. left subtree of the right child of *x*.
4. right subtree of the right child of *x*.

Idea: Cases 1 & 4 are solved by a single rotation.
Cases 2 & 3 are solved by a double rotation.



Insert: detect potential imbalance

1. Insert the new node as in a BST (a new leaf)
2. For each node on the path from the root to the new leaf, the insertion may (or may not) have changed the node's height
3. So after recursive insertion in a subtree, detect height imbalance and perform a *rotation* to restore balance at that node

All the action is in defining the correct rotations to restore balance

Fact that makes it a bit easier:

- There must be a deepest element that is imbalanced after the insert (all descendants still balanced)
- After rebalancing this deepest node, every node is balanced
- So at most one node needs to be rebalanced

Case #1 Example

Insert(6)
Insert(3)
Insert(1)

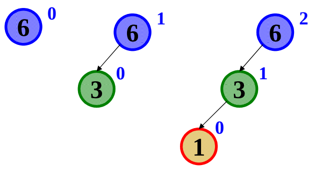
Case #1: Example

Insert(6)
Insert(3)
Insert(1)

Third insertion violates balance property

- happens to be at the root

What is the only way to fix this?

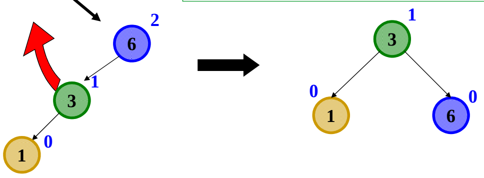


Fix: Apply “Single Rotation”

- *Single rotation:* The basic operation we’ll use to rebalance
  - Move child of unbalanced node into parent position
  - Parent becomes the “other” child (always okay in a BST!)
  - Other subtrees move in only way BST allows (next slide)

Single Rotation:
1. Rotate between self and child

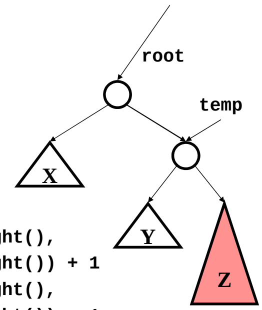
AVL Property violated here



RotateRight brings up the right child

Single Rotation Code

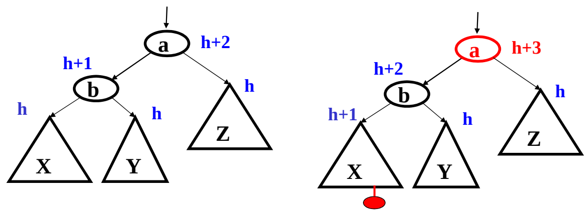
```
void RotateRight(Node root) {
  Node temp = root.right
  root.right = temp.left
  temp.left = root
  root.height = max(root.right.height(),
                    root.left.height()) + 1
  temp.height = max(temp.right.height(),
                    temp.left.height()) + 1
  root = temp
}
```



The example generalized

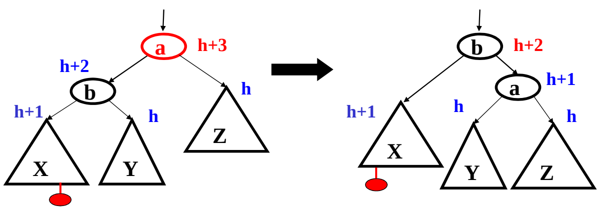
Notational note:
Oval: a node in the tree
Triangle: a subtree

- Node imbalanced due to insertion *somewhere* in **left-left grandchild** increasing height
  - 1 of 4 possible imbalance causes (other three coming)
- First we did the insertion, which would make ***a*** imbalanced



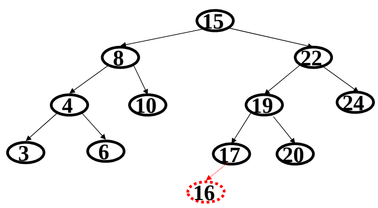
The general left-left case

- Node imbalanced due to insertion *somewhere* in **left-left grandchild** increasing height
  - 1 of 4 possible imbalance causes (other three coming)
- So we rotate at ***a***, using BST facts: X < b < Y < a < Z

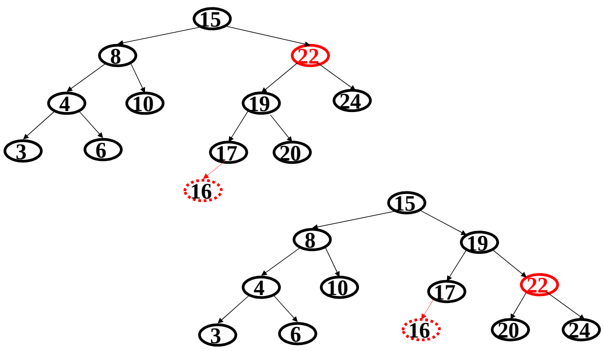


- A single rotation restores balance at the node
  - To same height as before insertion (so ancestors now balanced)

Another example: `insert(16)`

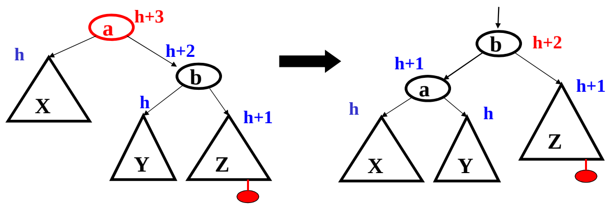


Another example: `insert(16)`



The general right-right case

- Mirror image to left-left case, so you rotate the other way
  - Exact same concept, but need different code



Case #3 Example

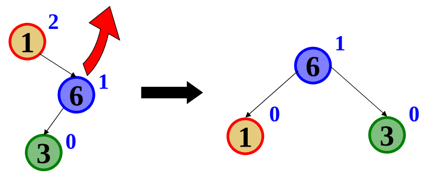
Insert(1)
Insert(6)
Insert(3)

Two cases to go

Unfortunately, single rotations are not enough for insertions in the left-right subtree or the right-left subtree

Simple example: `insert`(1), `insert`(6), `insert`(3)

- First wrong idea: single rotation like we did for left-left

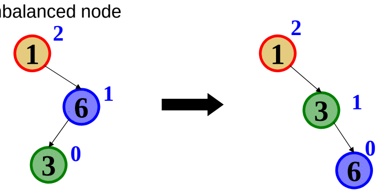


Two cases to go

Unfortunately, single rotations are not enough for insertions in the left-right subtree or the right-left subtree

Simple example: `insert`(1), `insert`(6), `insert`(3)

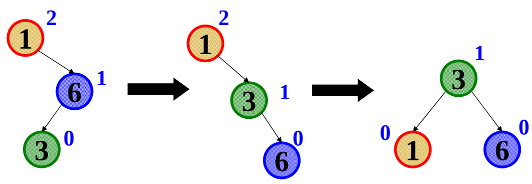
- Second wrong idea: single rotation on the child of the unbalanced node



Sometimes two wrongs make a right ☺

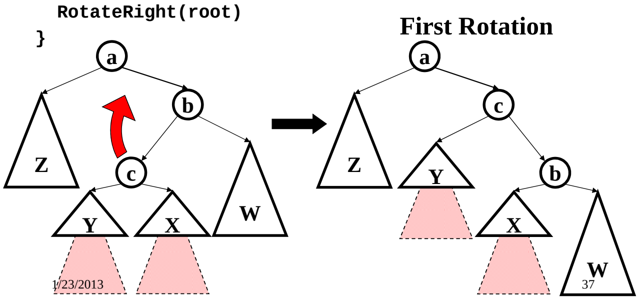
- First idea violated the BST property
- Second idea didn't fix balance
- But if we do both single rotations, starting with the second, it works! (And not just for this example.)

Double rotation:
1. **Rotate problematic child and grandchild**
2. **Then rotate between self and new child**

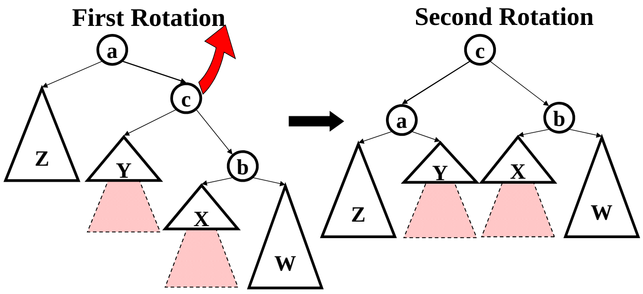


Double Rotation Code

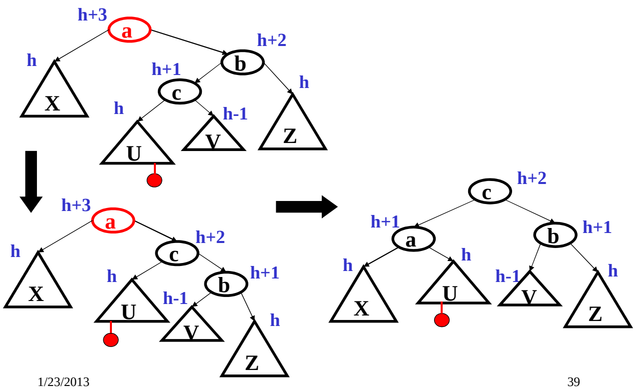
```
void DoubleRotateRight(Node root) {
  RotateLeft(root.right)
  RotateRight(root)
}
```



Double Rotation Completed

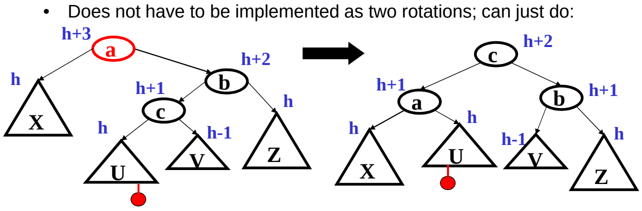


The general right-left case



Comments

- Like in the left-left and right-right cases, the height of the subtree after rebalancing is the same as before the insert
  - So no ancestor in the tree will need rebalancing
- Does not have to be implemented as two rotations; can just do:

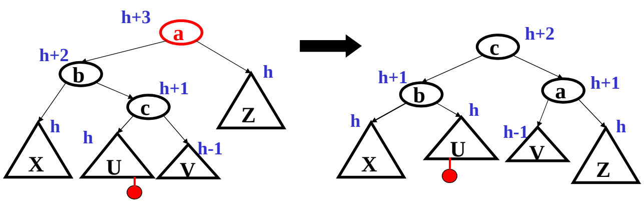


Easier to remember than you may think:

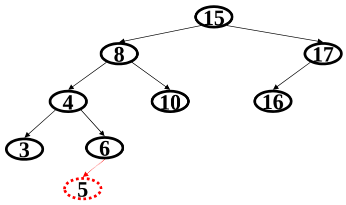
Move c to grandparent's position and then put a, b, X, U, V, and Z in the only legal positions for a BST

The last case: left-right

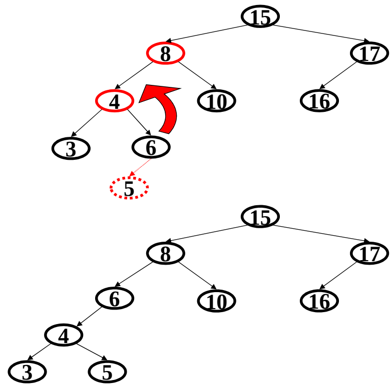
- Mirror image of right-left
  - Again, no new concepts, only new code to write



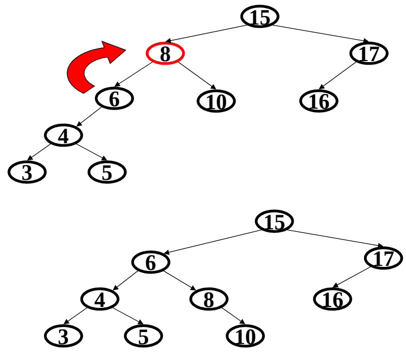
Insert 5



Double rotation, step 1



Double rotation, step 2



Insert, summarized

- Insert as in a BST
- Check back up path for imbalance, which will be 1 of 4 cases:
  - node's left-left grandchild is too tall
  - node's left-right grandchild is too tall
  - node's right-left grandchild is too tall
  - node's right-right grandchild is too tall
- *Only one case occurs* because tree was balanced before insert
- After the appropriate single or double rotation, the smallest-unbalanced subtree has the same height as before the insertion
  - So all ancestors are now balanced

Now efficiency

- Worst-case complexity of **`find`**: __________
  - Tree is balanced
- Worst-case complexity of **`insert`**: __________
  - Tree starts balanced
  - A rotation is *O*(1) and there's an *O*(**`log`** *n*) path to root
  - (Same complexity even without one-rotation-is-enough fact)
  - Tree ends balanced
- Worst-case complexity of **`buildTree`**: _________
- **`delete?`** (see 3 ed. Weiss) requires more rotations: _________
- Lazy deletion? ________________

Now efficiency

- Worst-case complexity of **`find`**: *O*(**`log`** *n*)
  - Tree is balanced
- Worst-case complexity of **`insert`**: *O*(**`log`** *n*)
  - Tree starts balanced
  - A rotation is *O*(1) and there's an *O*(**`log`** *n*) path to root
  - (Same complexity even without one-rotation-is-enough fact)
  - Tree ends balanced
- Worst-case complexity of **`buildTree`**: *O*(*n* **`log`** *n*)
- **`delete?`** (see 3 ed. Weiss) requires more rotations: *O*(**`log`** *n*)

Pros and Cons of AVL Trees

Arguments for AVL trees:

1. All operations logarithmic worst-case because trees are *always* balanced
2. Height balancing adds no more than a constant factor to the speed of **`insert`** and **`delete`**

Arguments against AVL trees:

1. Difficult to program & debug
2. More space for height field
3. Asymptotically faster but rebalancing takes a little time
4. Most large searches are done in database-like systems on disk and use other structures (e.g., B-trees, our next data structure)

More Examples…

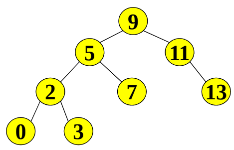
Insert into an AVL tree: a b e c d

Student Activity

Single and Double Rotations:

Inserting what integer values would cause the tree to need a:

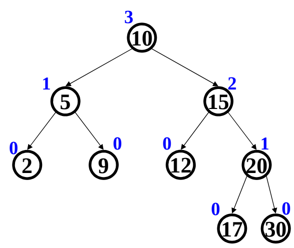
1. **single rotation?**
2. **double rotation?**
3. **no rotation?**



Student Activity

Easy Insert

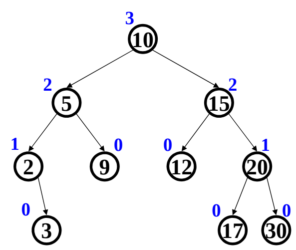
Insert(3)



Unbalanced?

Hard Insert

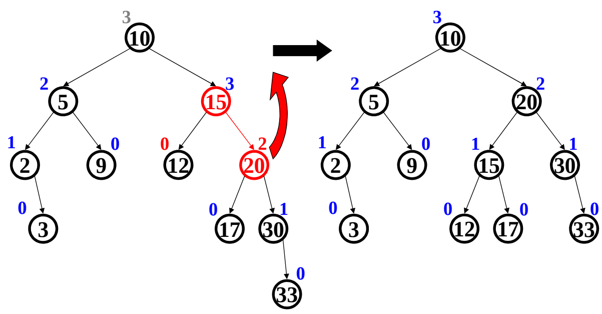
Insert(33)



Unbalanced?

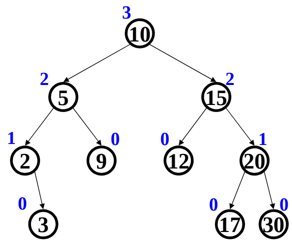
How to fix?

Single Rotation



Hard Insert

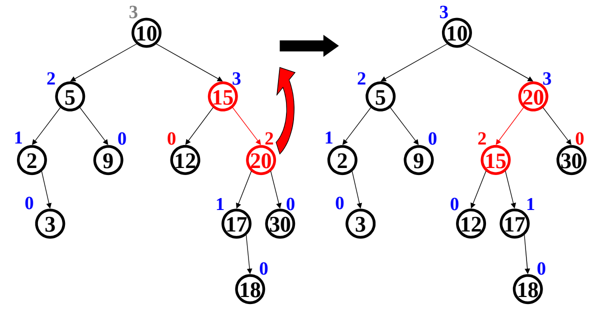
Insert(18)



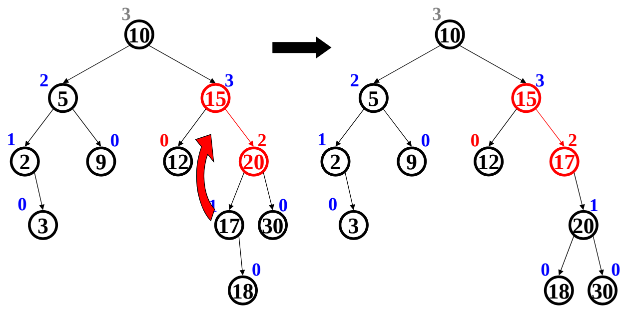
Unbalanced?

How to fix?

Single Rotation (oops!)



Double Rotation (Step #1)



Double Rotation (Step #2)