Unit #6: AVL Trees

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveisi

2016W2

Skip slide 28

Annotated Slides from Ed's Class

Unit Outline

- Binary search trees
- Balance implies shallow (shallow is good)
- How to achieve balance
- Single and double rotations
- AVL tree implementation

Learning Goals

- Compare and contrast balanced/unbalanced trees.
- Describe and apply rotation to a BST to achieve a balanced tree.
- Recognize balanced binary search trees (among other tree types you recognize, e.g., heaps, general binary trees, general BSTs).

Dictionary ADT Implementations

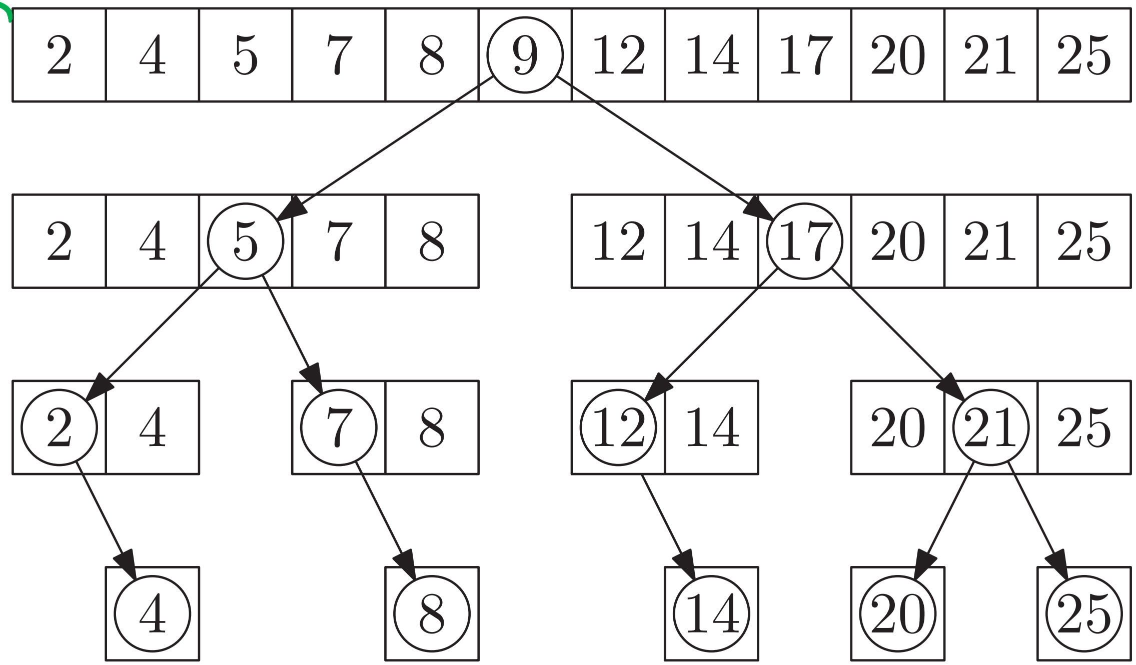
Worst Case time	insert	find	delete (after find)
Linked list	$\Theta(1)$	$\Theta(n)$	$O(1)$
Unsorted array	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Sorted array	$\Theta(n)$	$\Theta(\lg n)$	$\Theta(n)$
Hash table	chaining $\Theta(n)$ open addr. $\Theta(n)$	$\Theta(n)$ $\Theta(n)$	$\Theta(1)$ $\Theta(1)$ with tombstones

$\Theta(1)$ if inserting known-to-be unique values

does the key already exist?

Binary Search in a Sorted List

$O(\lg n)$




```
int bSearch(int A[], int key, int i, int j) {
  if (j < i) return -1;
  int m = (i + j) / 2;
  if (key < A[m]) return bSearch(A, key, i, m-1);
  else if (key > A[m]) return bSearch(A, key, m+1, j);
  else return m;
}
```

Binary Search Tree as Dictionary Data Structure

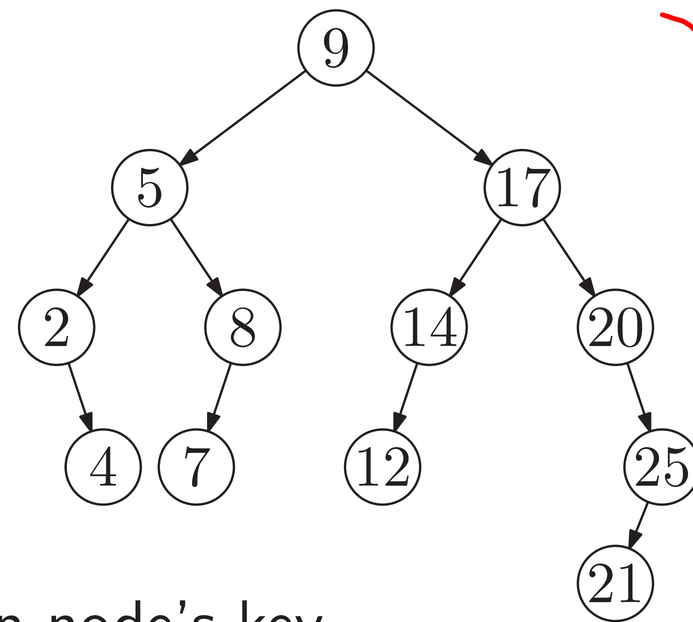
Binary tree property

- each node has ≤ 2 children

Search tree property

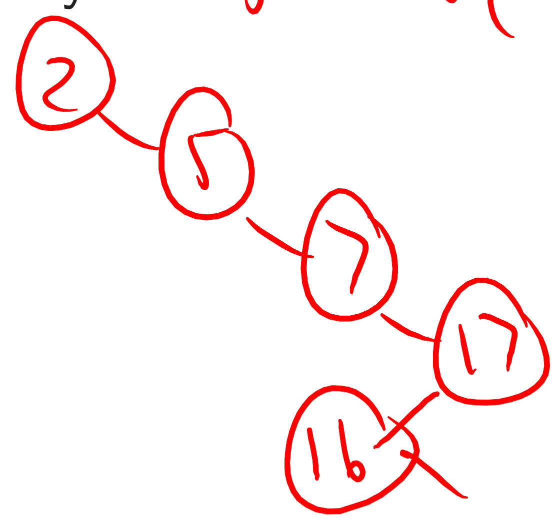
- all keys in left subtree smaller than node's key
- all keys in right subtree larger than node's key

Result: easy to find any given key



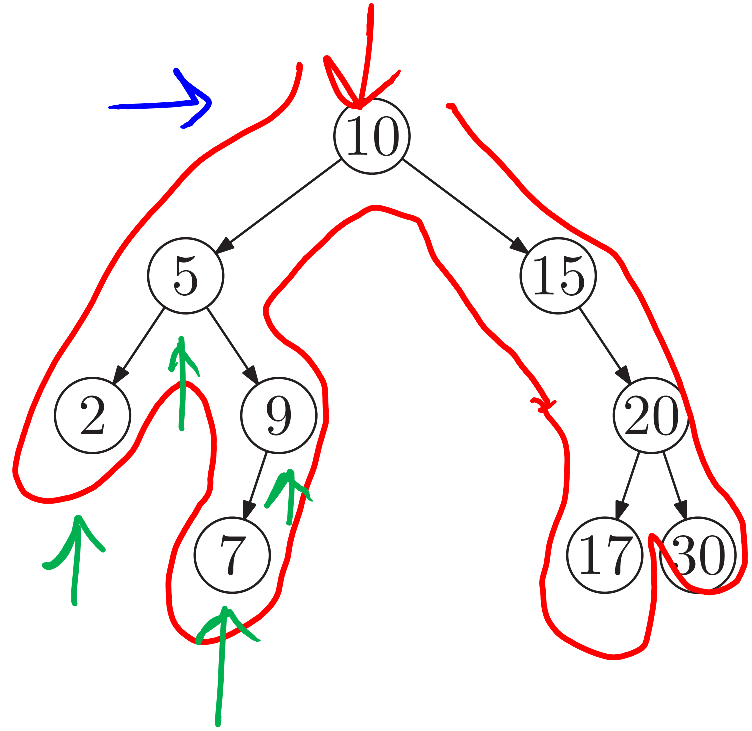
h=4 edges

worst case



$h = O(n)$

In-, Pre-, Post-Order Traversal



recursive
go L
visit
go R

6pm In-order: 2, 5, 7, 9, 10, 15, 17, 20, 30

9pm Pre-order: visit L R   10, 5, 2, 9, 7, 15, 20, 17, 30

3pm Post-order: L R visit   2, 7, 9, 5, 17, 30, 20, 15, 10

Beauty is Only $O(\log n)$ Deep

BST
insert 2, 3, 7, 11, 15

$2 \to 3 \to 7 \to 11 \to 15$

$O(n)$
search

Binary Search Trees are fast if they're shallow.
Know any shallow trees?

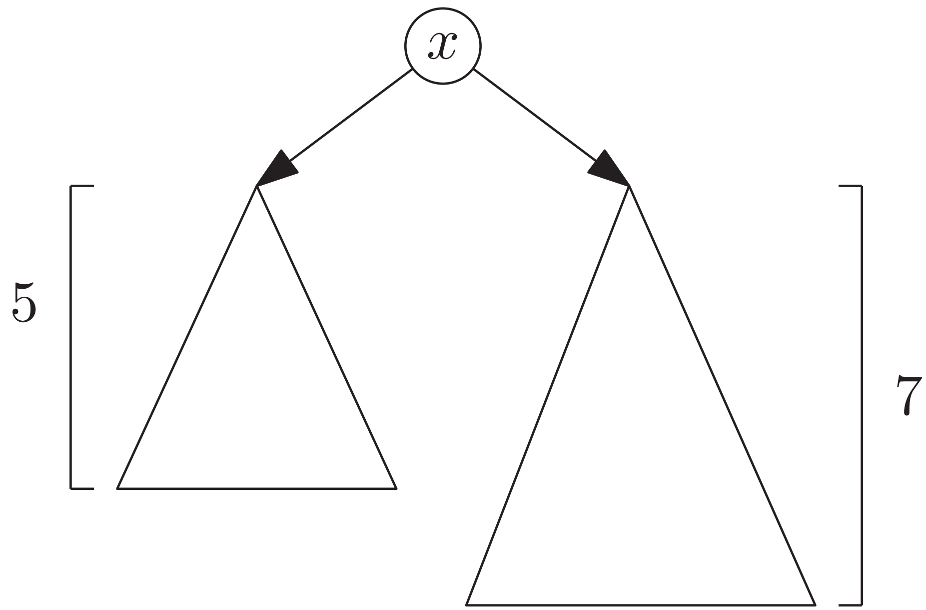
- perfectly complete
- perfectly complete except the last level (like a heap)
- anything else?

What matters here?

Siblings should have about the same height.

nodes that have the same parent

Balance



goal: want $O(\lg n)$ search

$\text{balance}(x) = \text{height}(x.\text{left}) - \text{height}(x.\text{right})$ $= 5 - 7 = -2$

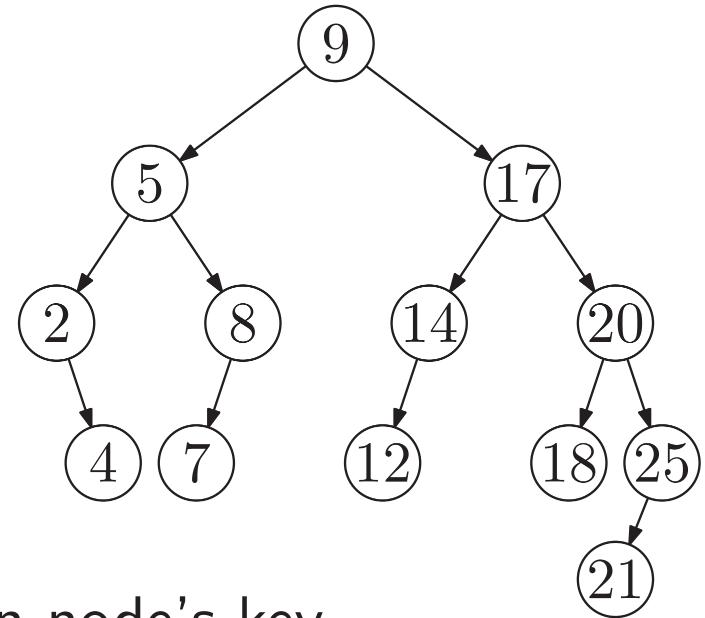
$\text{height}(\texttt{NULL}) = -1.$

If for all nodes x,

- $\text{balance}(x) = 0$ then perfectly balanced.
- $|\text{balance}(x)|$ is small then balanced enough.
- $-1 \le \text{balance}(x) \le 1$ then tree height $\le c \lg n$ where $c < 2$.

close to 1

AVL (Adelson-Velsky and Landis) Tree



Binary tree property

- each node has ≤ 2 children

Search tree property

- all keys in left subtree smaller than node's key
- all keys in right subtree larger than node's key

Balance property

- For all nodes x, $-1 \leq \text{balance}(x) \leq 1$

Result: height is $\Theta(\log n)$.

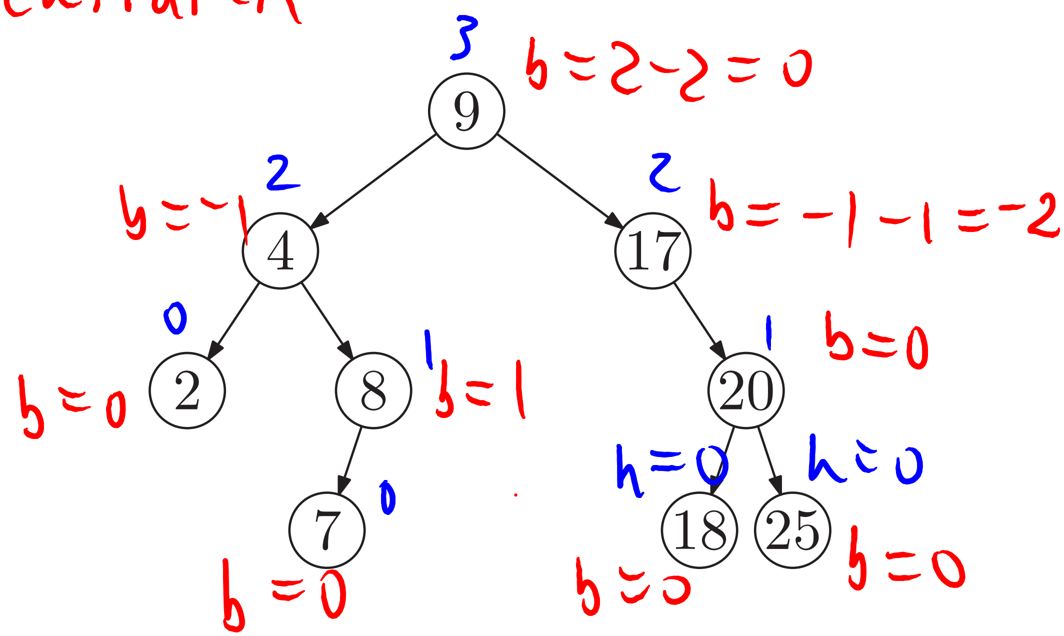
Is this an AVL tree? No

balance $= L_h - R_h$

heights of children

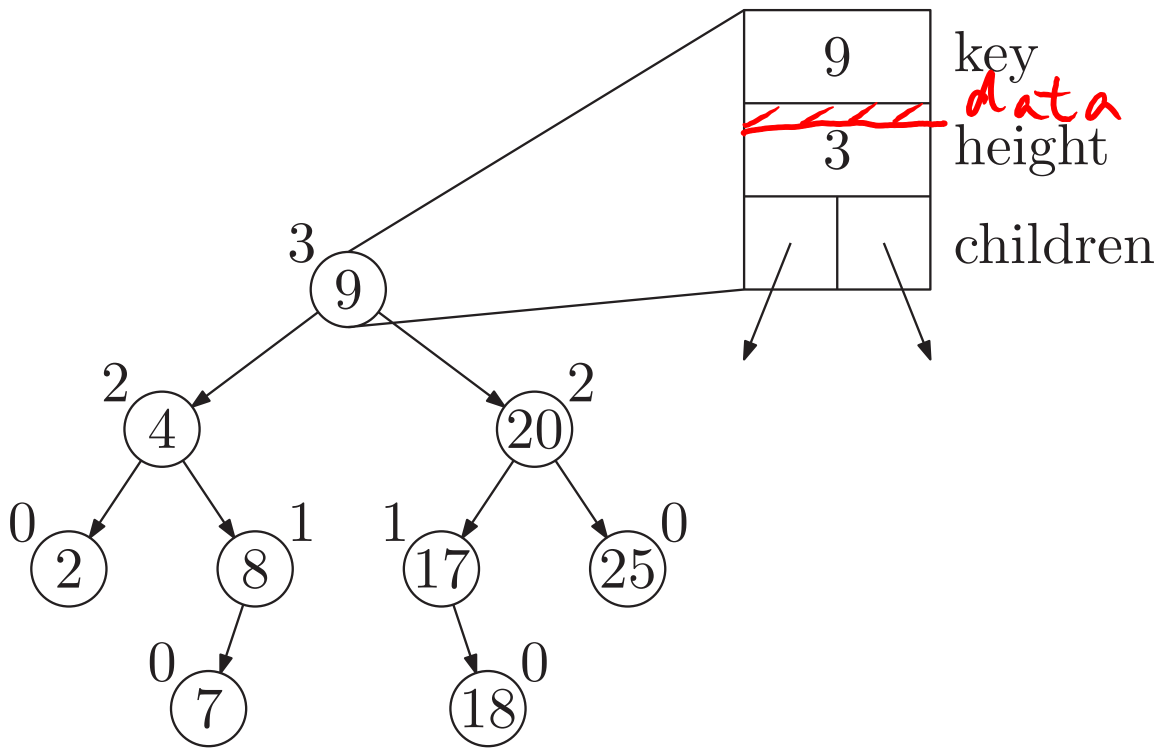
height(NULL) $= -1$

height = path length (edges from node to leaf level)



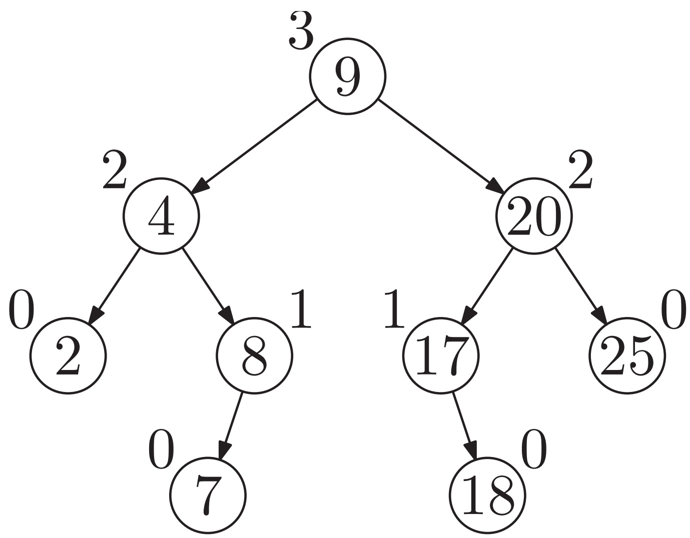
No, because of node 17; it's out of balance.

An AVL Tree



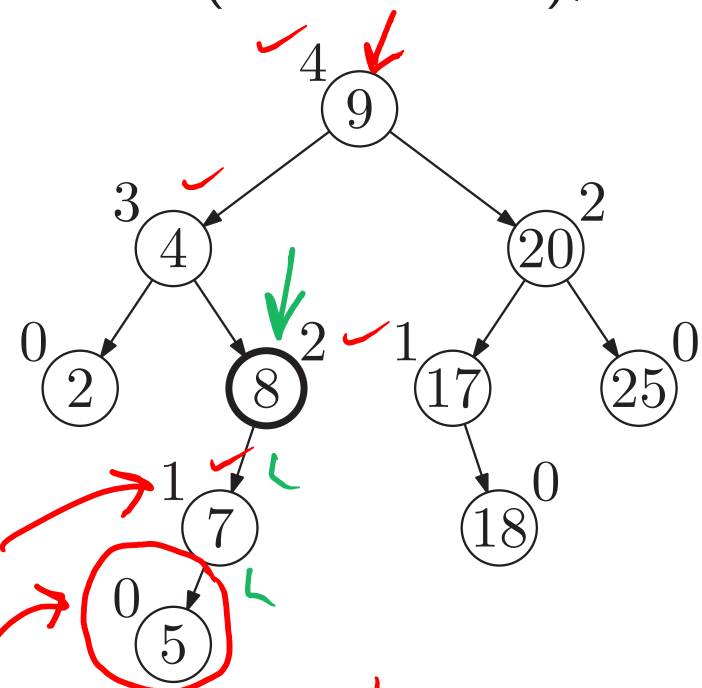
How Do We Stay Balanced?

Suppose we start with a balanced search tree (an AVL tree),



and insert 5

5 ⟹



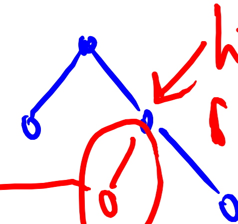
heights changed up to the root

It's no longer an AVL tree. What can we do?

– but in general, stop when ancestor's height doesn't change (or if you invoke a rotation – later)

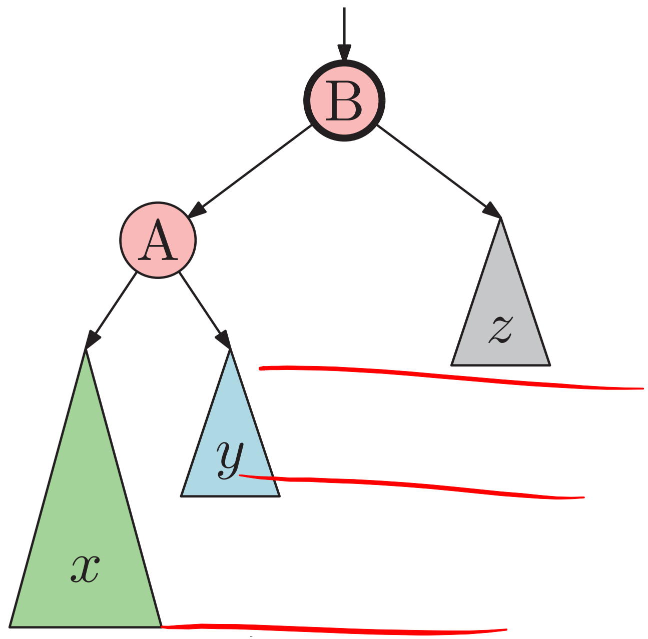
ROTATE!

insert



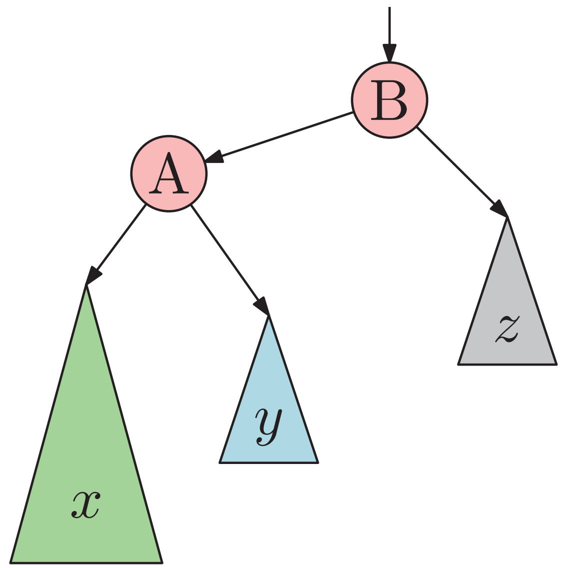
h=1 still

Rotation Animation

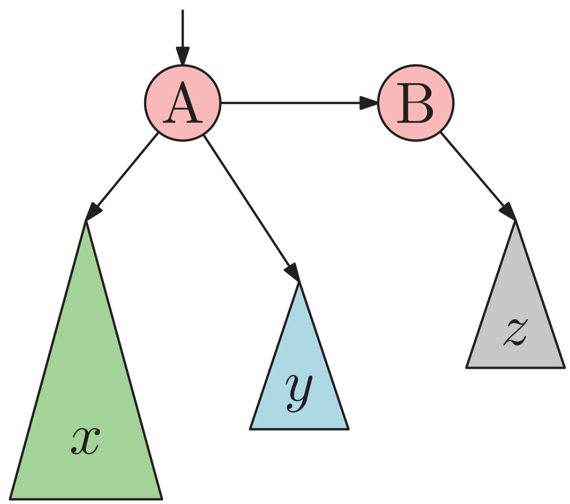


Note that y subtree's values are $\geq A$ & $\leq B$

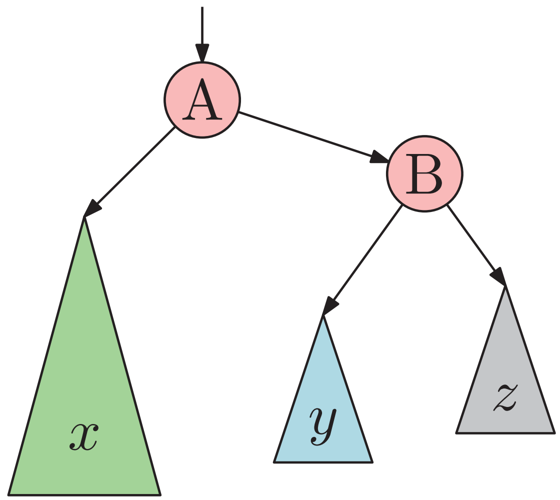
Rotation Animation



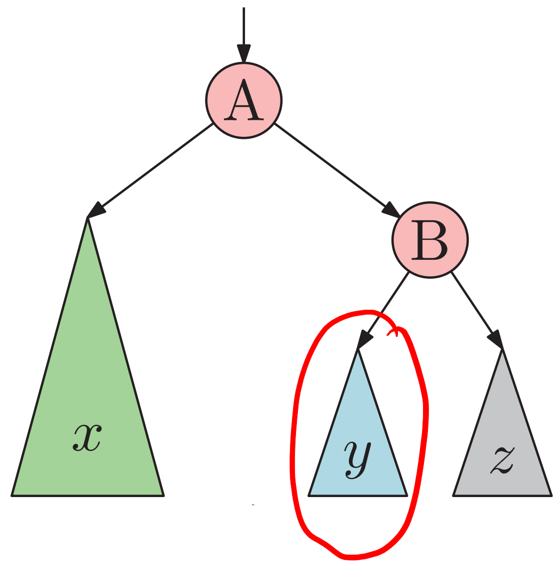
Rotation Animation



Rotation Animation

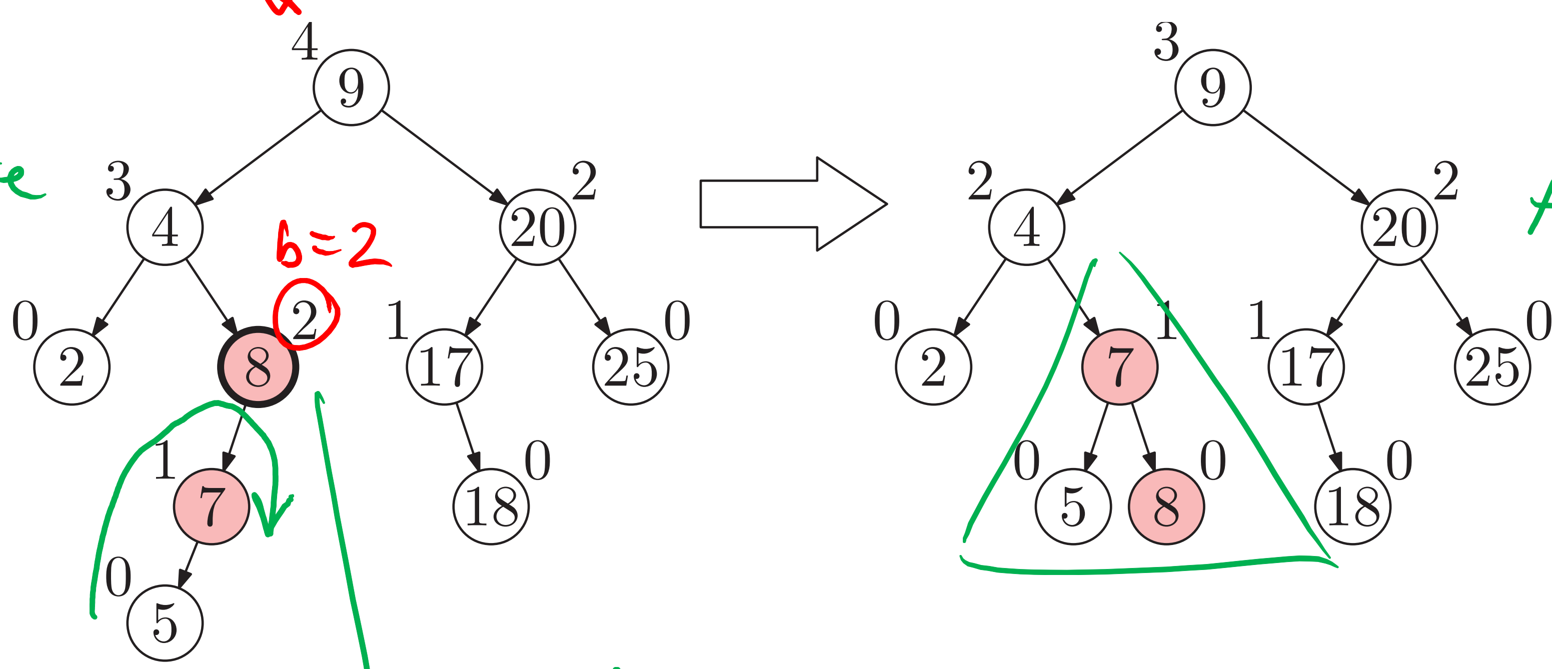


Rotation Animation



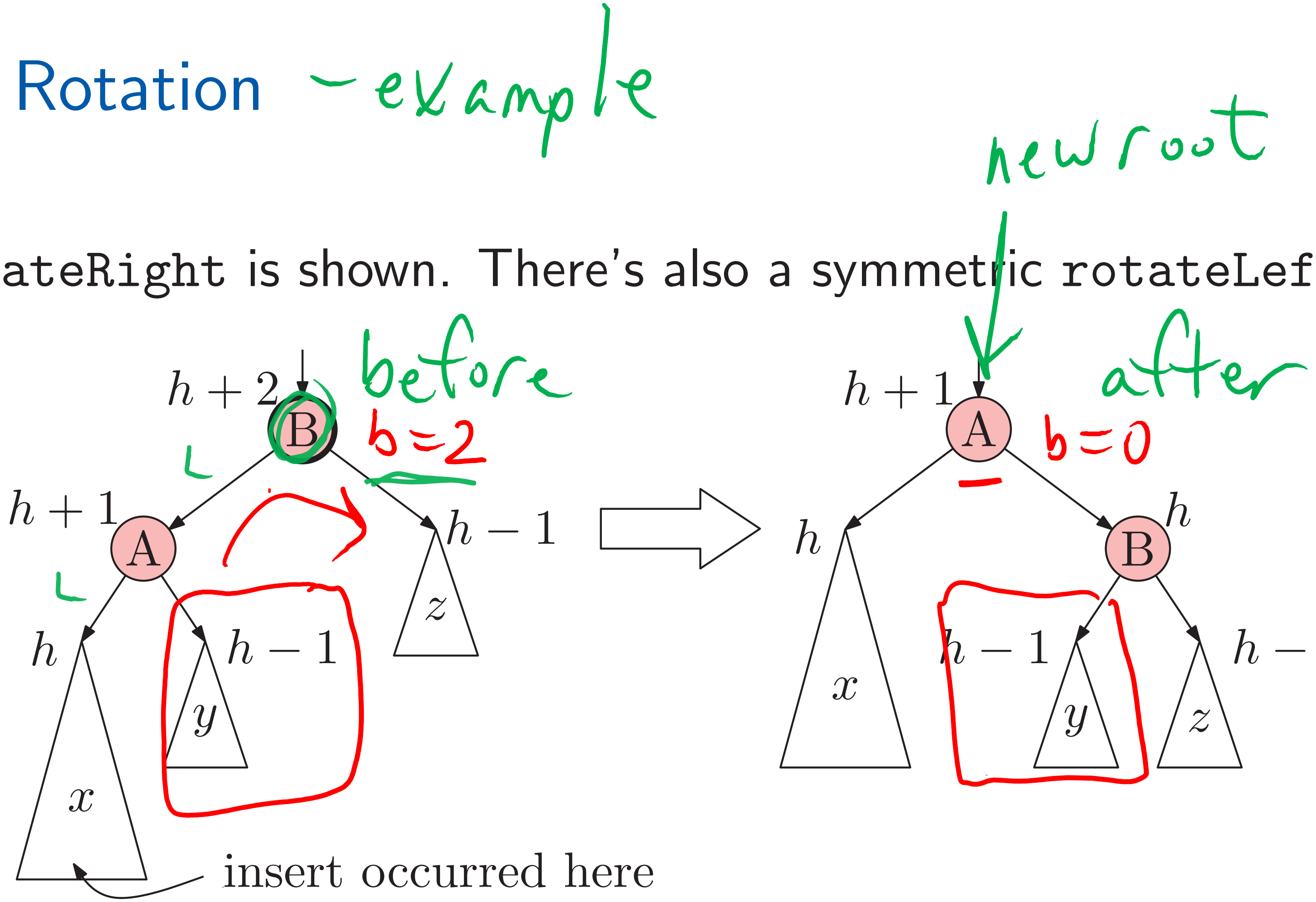
y is still between A & B

Single Rotation — an example of one of 4 different kinds of rotation



Single Rotation - example

`rotateRight` is shown. There's also a symmetric `rotateLeft`.



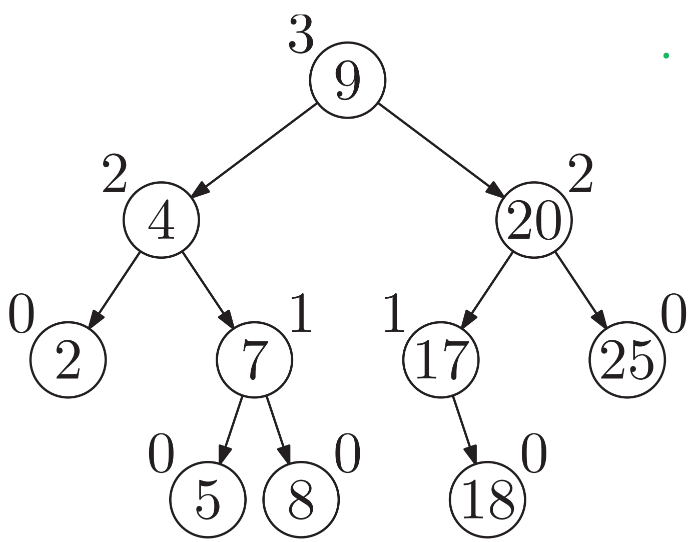
After rotation, subtree's height is the same as before insert.
So heights of ancestors don't change. from before the Insert.
So?

confusing but compare slide 12 to 15's RHS to see this
- heights are the same

Double Rotation

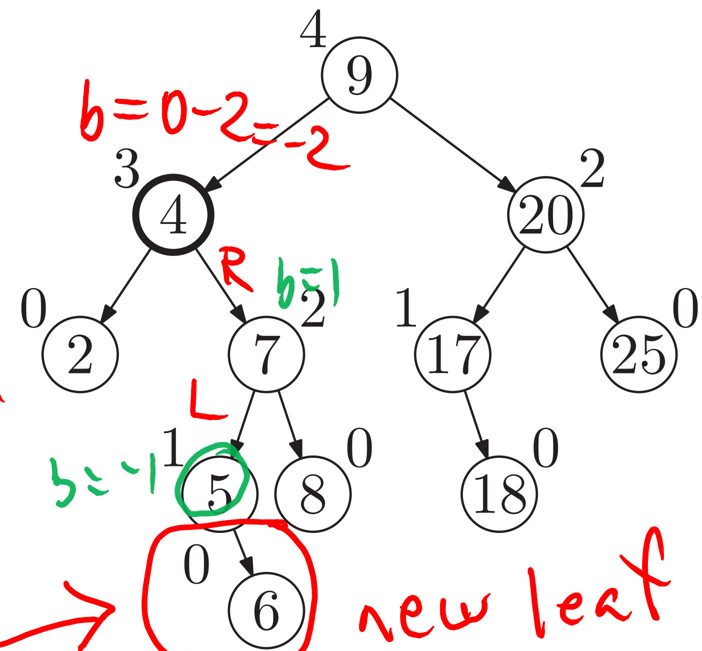
LL
LR
RL
RR

Start with



and insert 6

becomes unbalanced



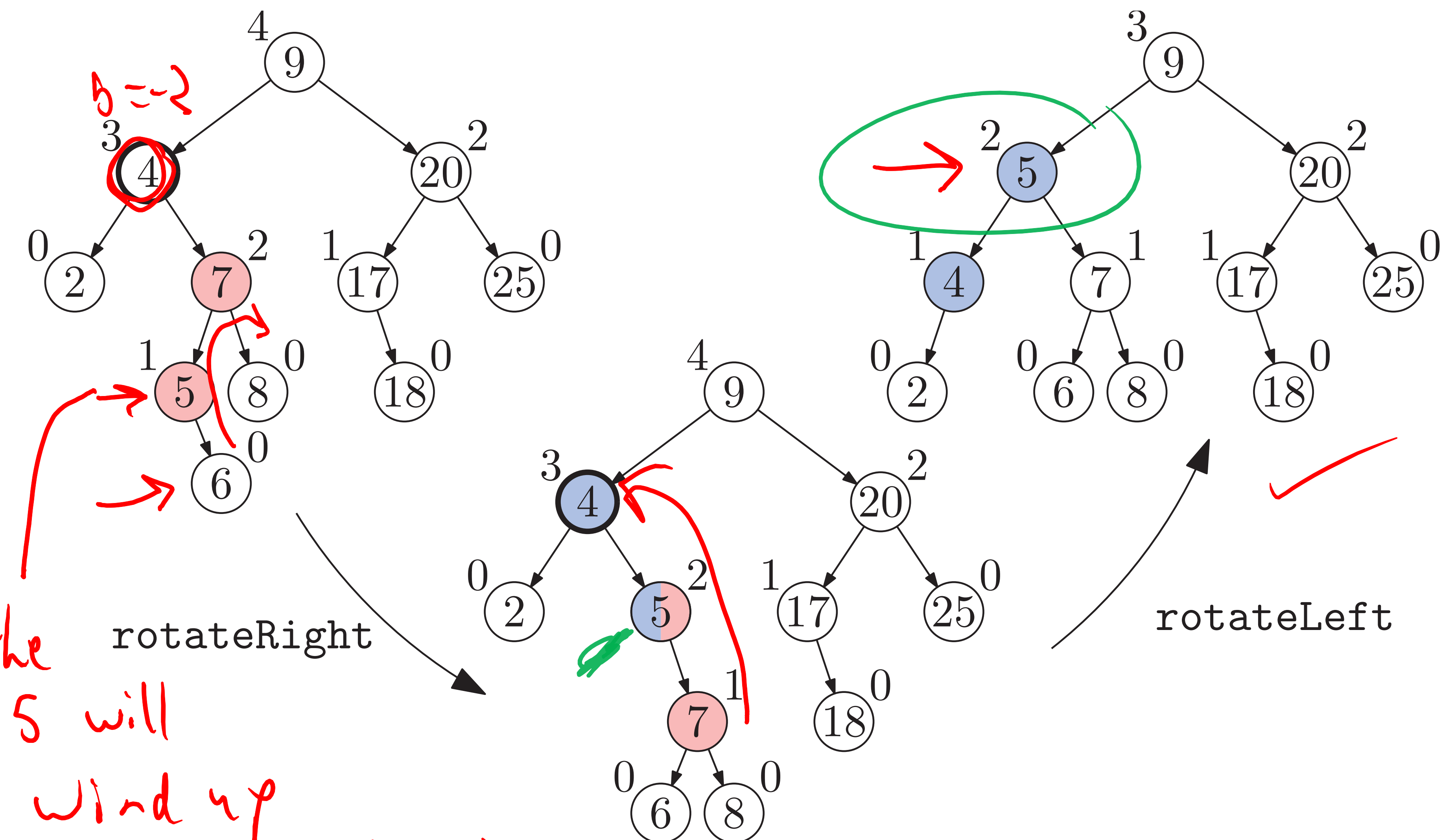
new leaf
- go up to find first unbalanced ancestor 4

A single rotation won't fix this.

R-L means
(L-R too)

DOUBLE ROTATE!

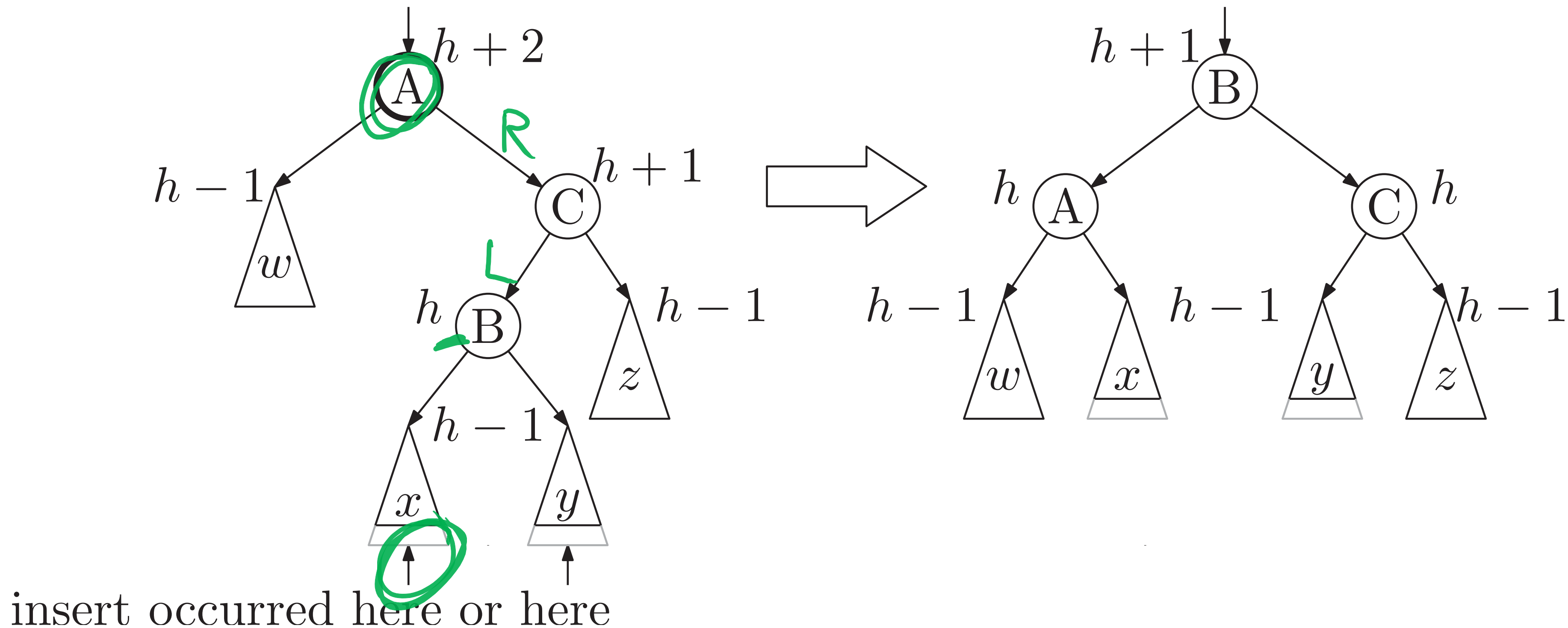
Double Rotation



the 5 will wind up at the unbalanced node's location

Double Rotation

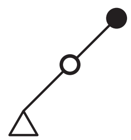
`doubleRotateLeft` is shown. There's also a symmetric `doubleRotateRight`.

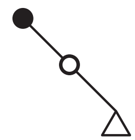


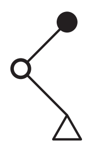
Either x or y increased to height $h - 1$ after insert.
After rotation, subtree's height is the same as before insert.
So height of ancestors doesn't change.

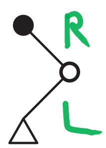
Insert Algorithm

1. Find location for new key.
2. Add new leaf node with new key.
3. Go up tree from new leaf searching for imbalance.
4. At lowest unbalanced ancestor:

Case LL:  `rotateRight`

Case RR:  `rotateLeft`

Case LR:  `doubleRotateRight` L then R

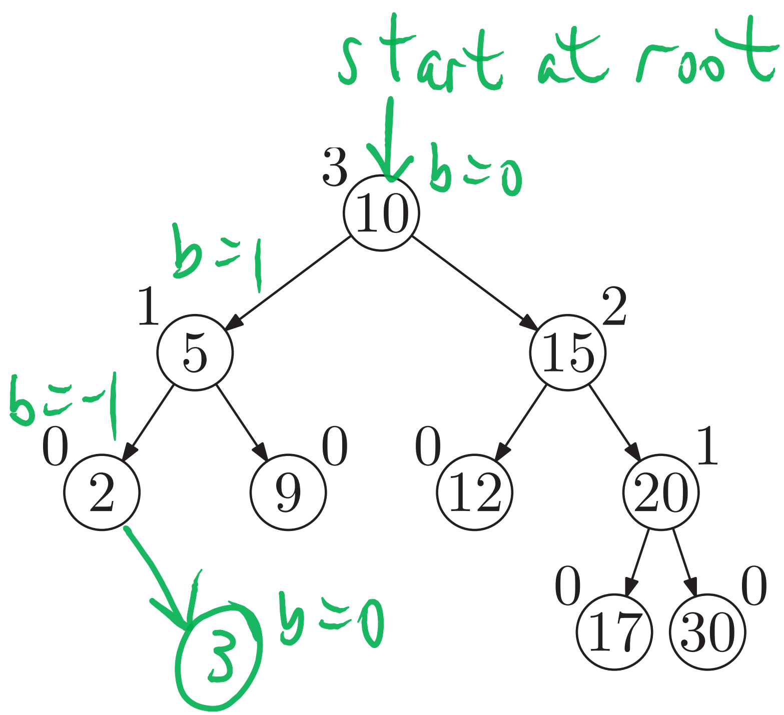
Case RL: 

`doubleRotateLeft` R then L

The case names are the first two steps on the path from the unbalanced ancestor to the new leaf.

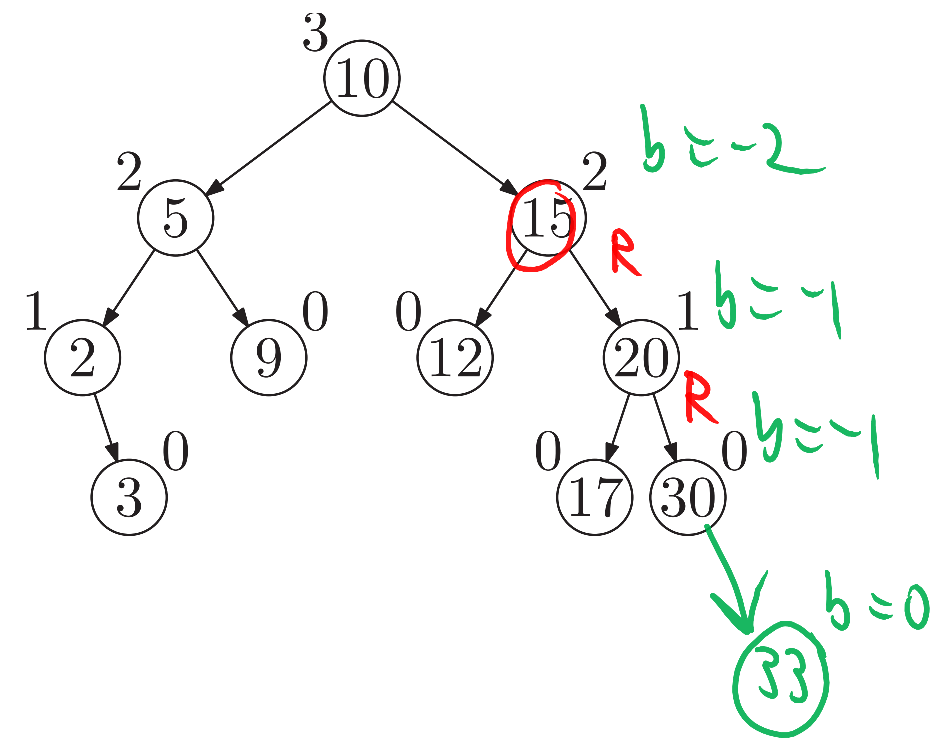
Insert: No Imbalance

Insert(3)

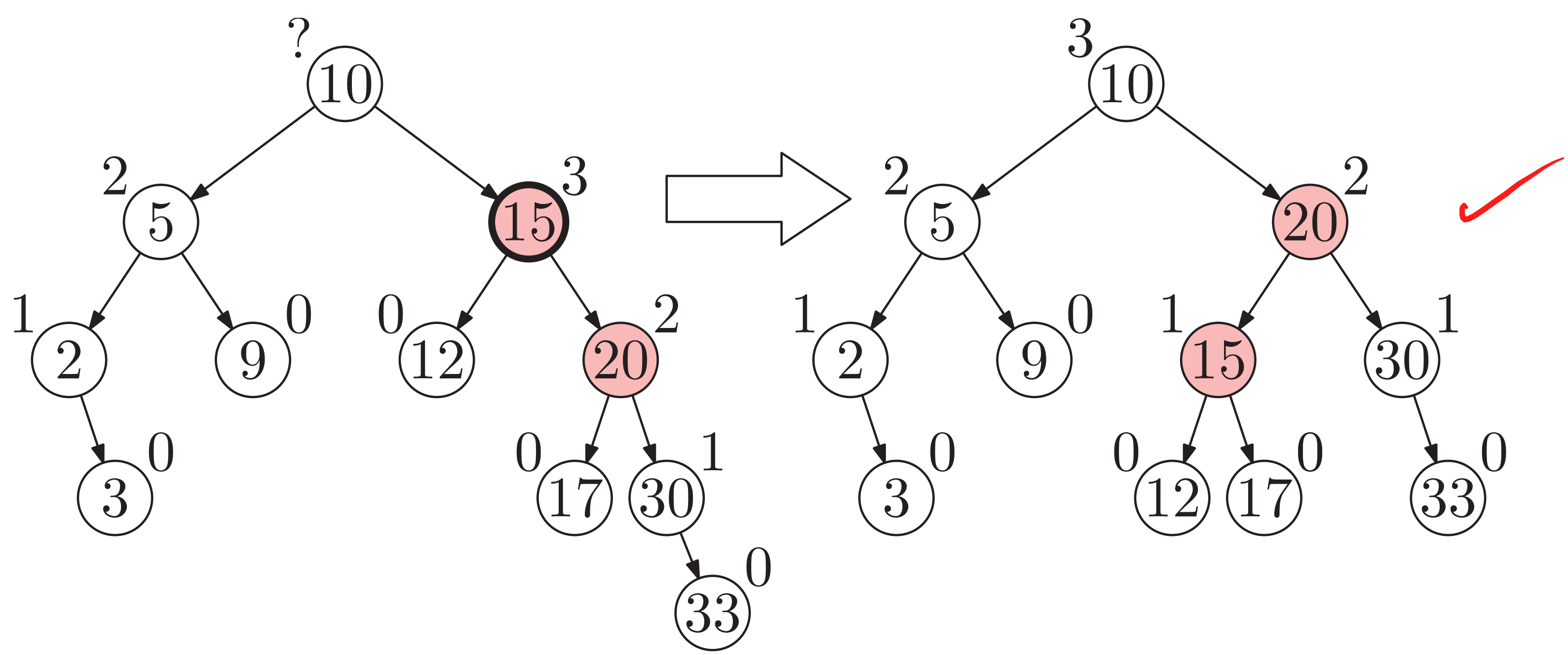


Insert: Imbalance Case RR

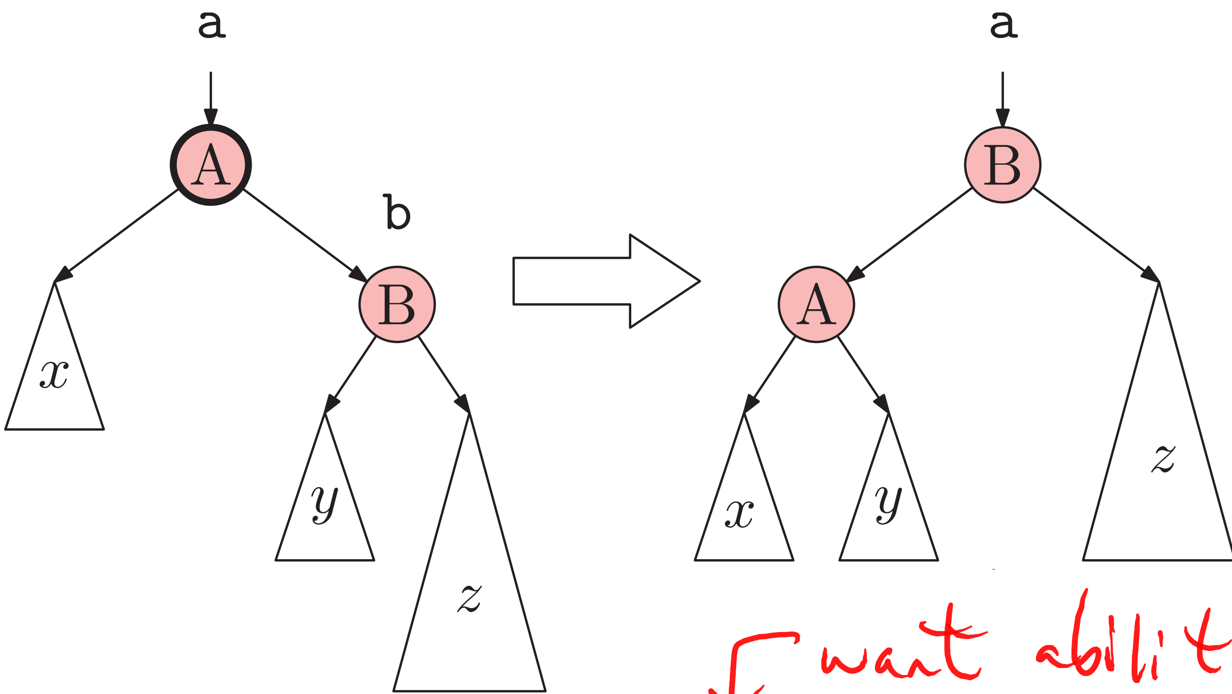
Insert(33)



Case RR: `rotateLeft`



Single Rotation Code



want ability to change the root of this (sub)tree

```
void rotateLeft( Node *& a ) {
  Node * b = a->right;
  a->right = b->left;
  b->left = a;
  updateHeight(a);
  updateHeight(b);
  a = b;
}
```

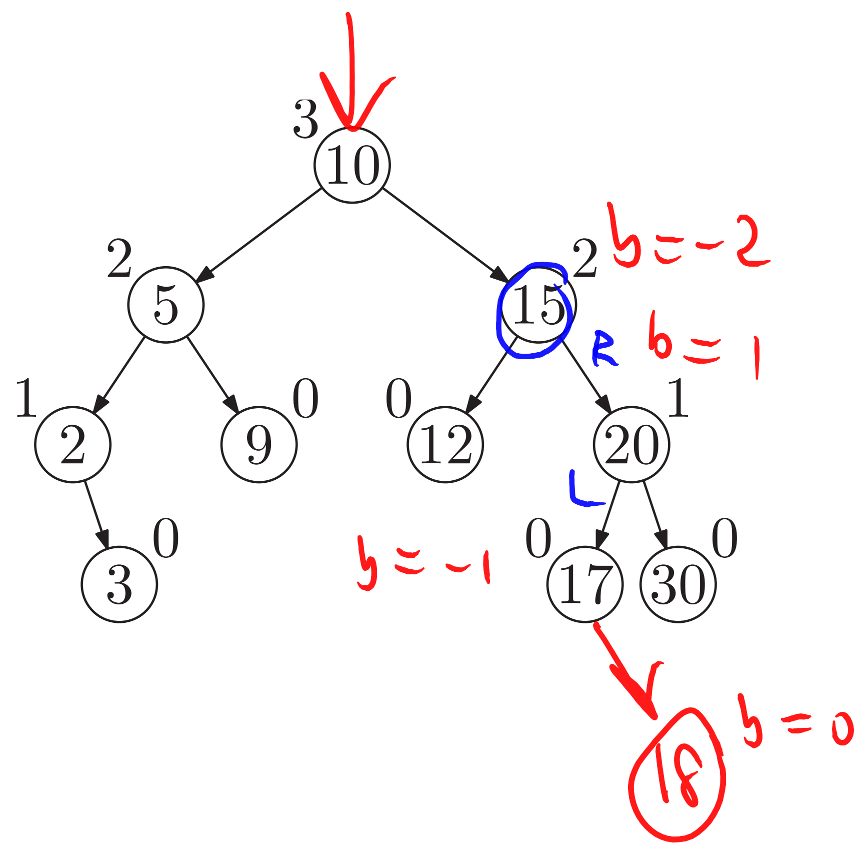
manipulate pointers like on the diagrams above and on previous pages

a->height = 1 + max{height(a->left), height(a->right)}

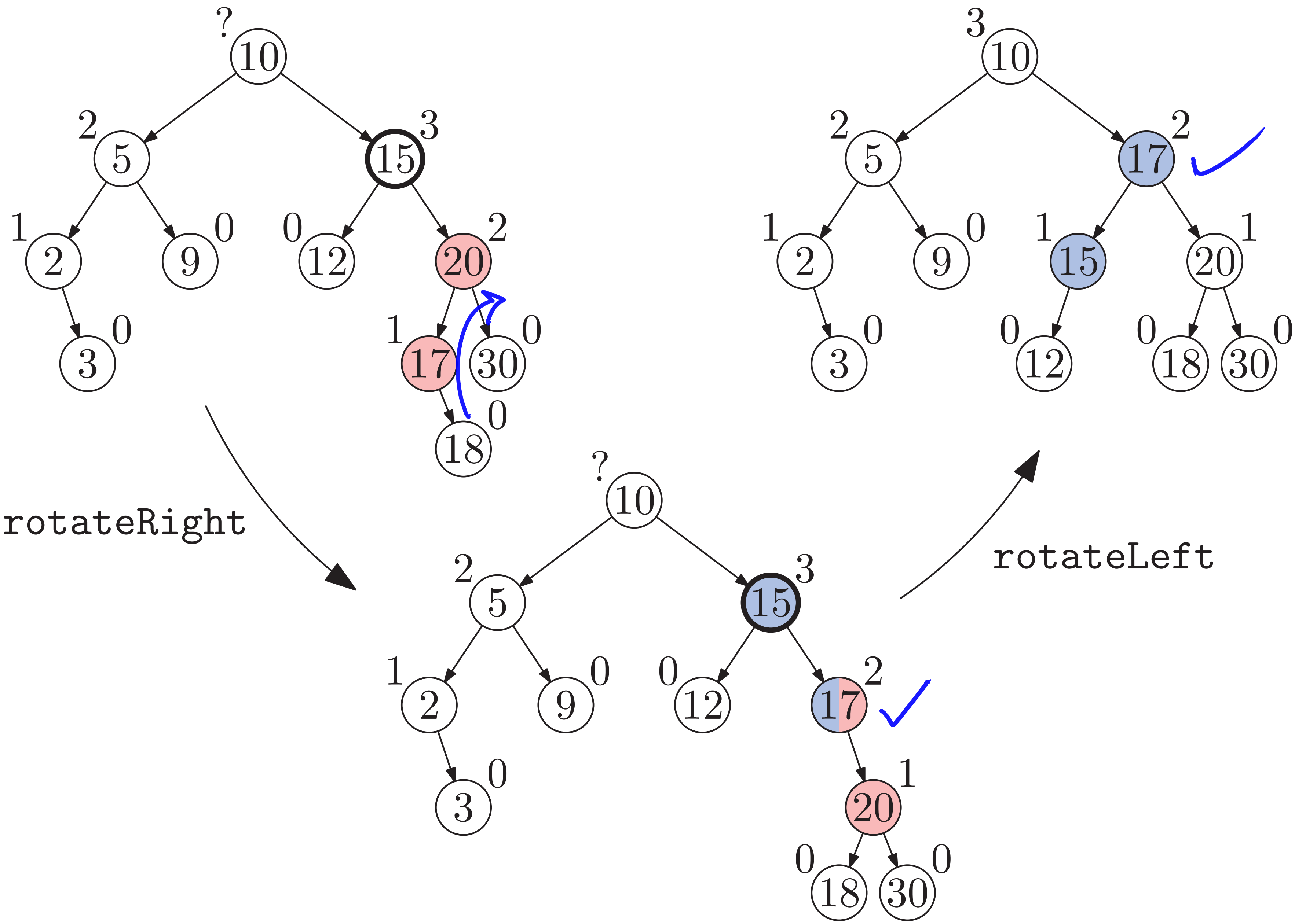
set new parent

Insert: Imbalance Case RL

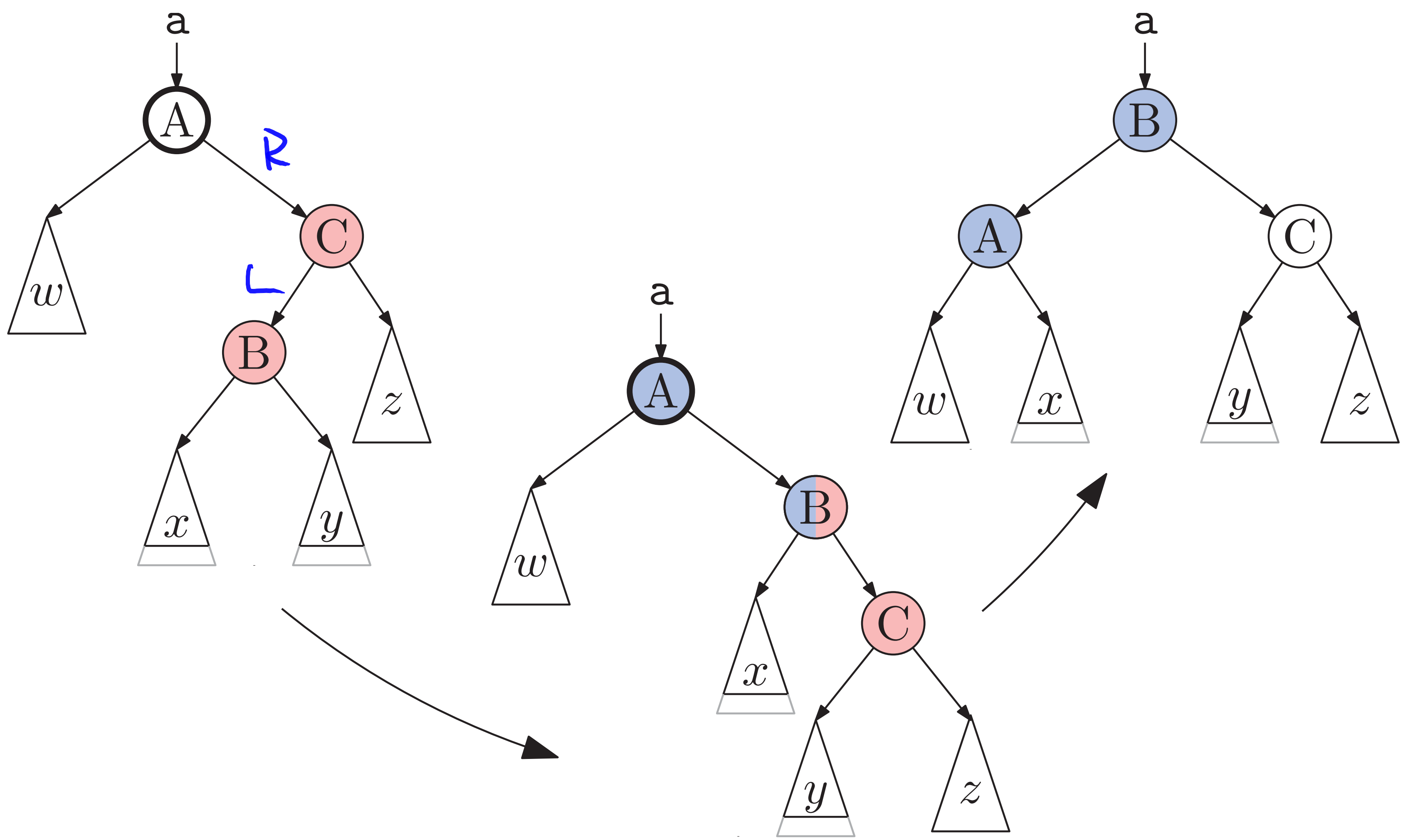
Insert(18)



Case RL: doubleRotateLeft



Double Rotation Code

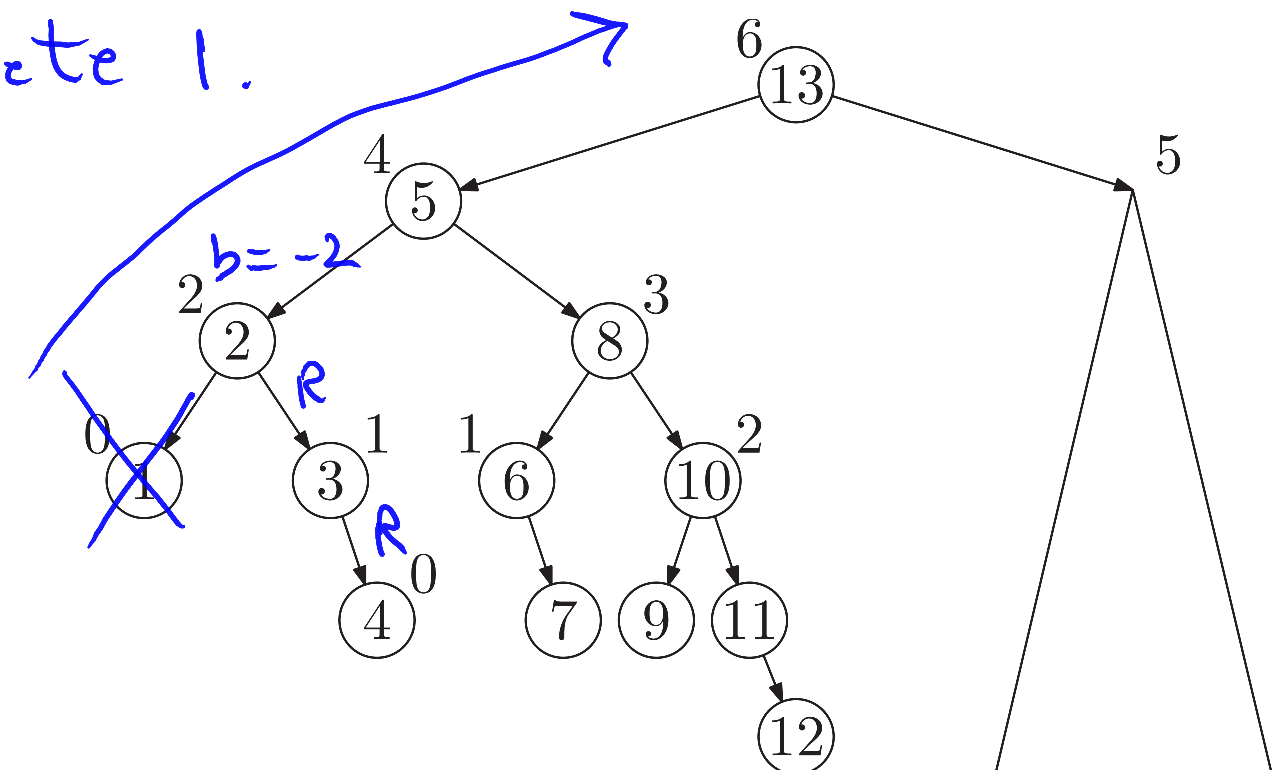



```
void doubleRotateLeft( Node *& a ) {
  rotateRight(a->right);
  rotateLeft(a);
}
```

Delete

1. Delete as for general binary search tree. (This way we reduce the problem to deleting a node with 0 or 1 child.)
2. Go up tree from deleted node searching for imbalance (and fixing heights).
3. Fix **all** unbalanced ancestors (bottom-up)

Delete 1.



Thinking about AVL trees

Observations

- AVL trees are binary search trees that allow only slight imbalance
- Worst-case $O(\log n)$ time for find, insert, and delete
- Elements (even siblings) may be scattered in memory

Realities

- For large data sets, disk accesses dominate runtime

Could we have perfect balance if we relax binary tree restriction?