Unit #6: AVL Trees

CPSC 221: Algorithms and Data Structures

Will Evans and Jan Manuch

2016W1

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).

<a,b>

- range search

to find the first key a

to do in-order traversal until b

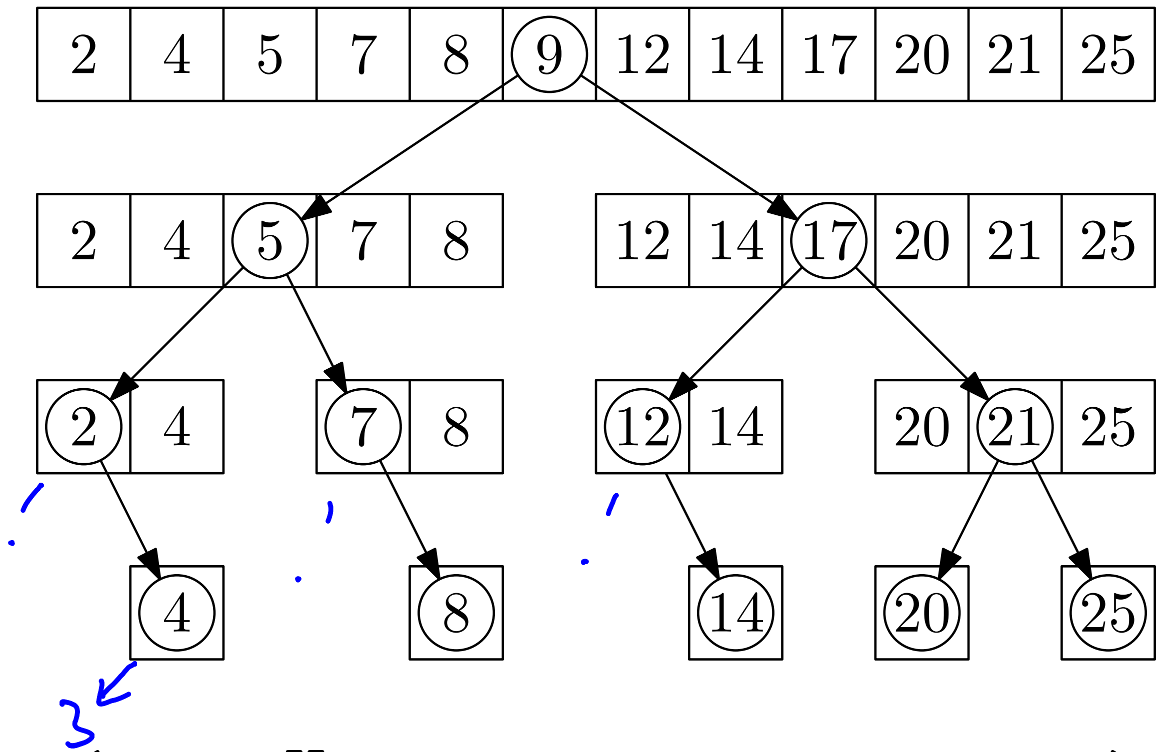
$\Theta(\log n + k)$

k = # nodes to output

Dictionary ADT Implementations

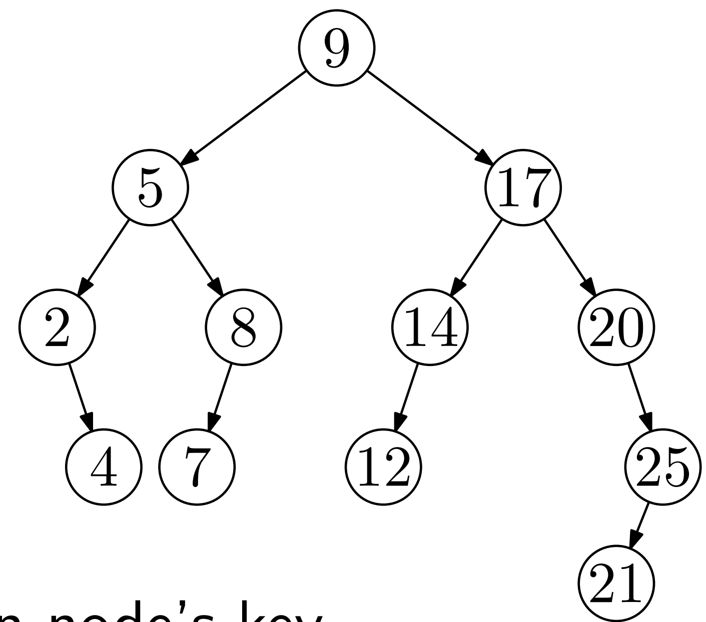
AVL Worst Case time	$\Theta(\log n)$ insert	$\Theta(\log n)$ find	$\Theta(\log n)$ delete (after find)
Linked list	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
(assuming no resize needed on insert) Unsorted array	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$ ← move last element to deleted position
Sorted array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Hash table	chain: $\Theta(1)$ open: $\Theta(n)$	$\Theta(n)$	$\Theta(1)$ chain: LL delete open: mark with tombstone

Binary Search in a Sorted List




```
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

Worst-case time for find(): $\Theta(4) = \Theta(n)$

Example:

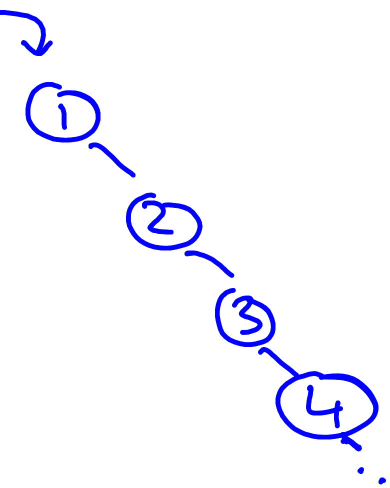
insert(1)

insert(2)

⋮

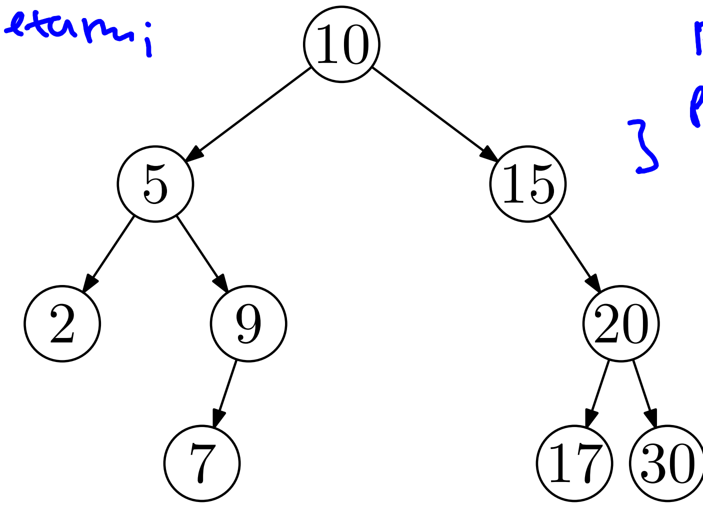
insert(n)

find(n+1)



In-, Pre-, Post-Order Traversal

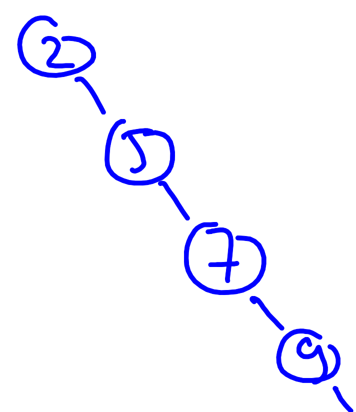
```
in(x) {
  if (x = NULL) return;
  in(x -> left);
  visit(x)
  in (x -> right;
}
```




```
pre (x) {
  visit(x);
  pre (x -> left);
  pre (x -> right);
}
```

```
post(x) {
  post(x -> left);
  post (x -> right);
  visit(x);
}
```

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



Pre-order:

Post-order:

Questions to think about:

- does output of a traversal uniquely determine BST?
- how about if you know the output of two traversals (eg.: in & pre)?

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

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

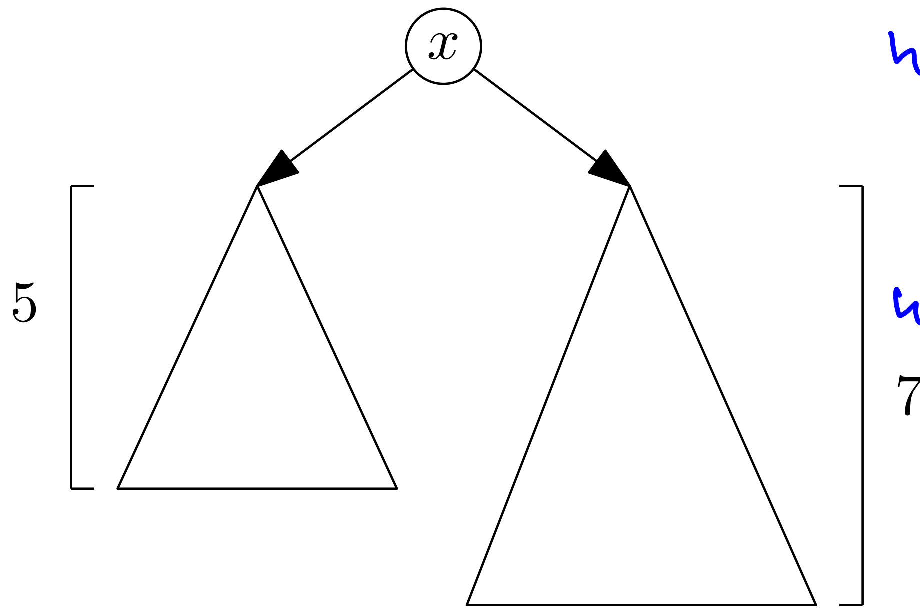
- perfectly complete
- perfectly complete except the last level (like a heap)
- anything else?

What matters here?

Siblings should have about the same height.

Balance

balance(x) = 5 − 7 = −2



height(x) = 1 + max $\begin{cases} \text{height}(x \to \text{left}) \\ \text{height}(x \to \text{right}) \end{cases}$

height((5)) = 0

NULL NULL

So we need to set

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

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

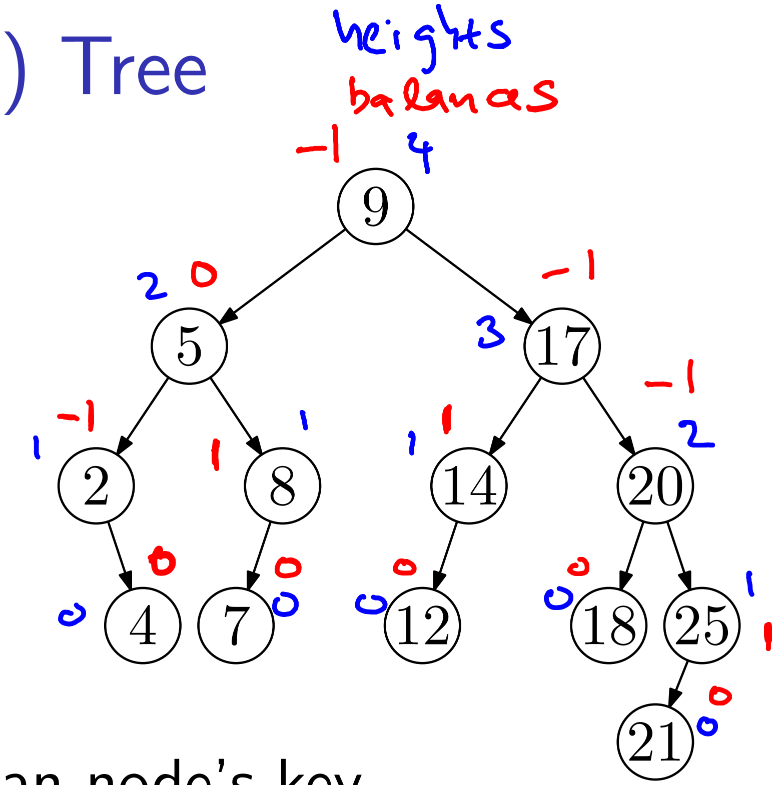
If for all nodes x,

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

1.4405

$\in \Theta(\lg n)$

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

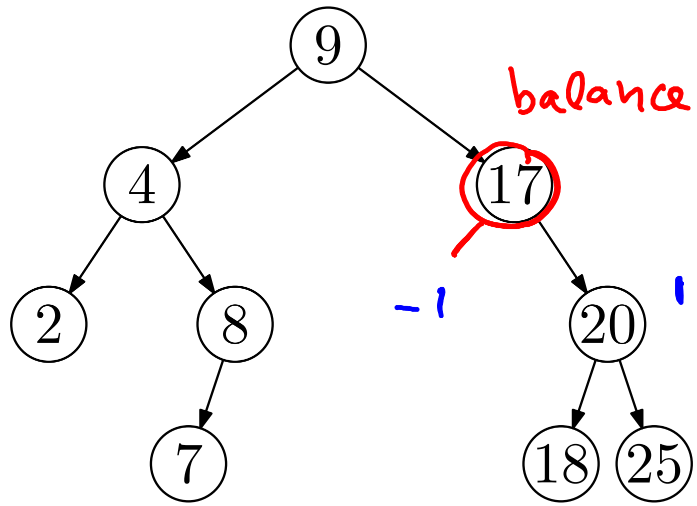
- For all nodes x, $-1 \leq \text{balance}(x) \leq 1$

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

Is this an AVL tree?

NO!

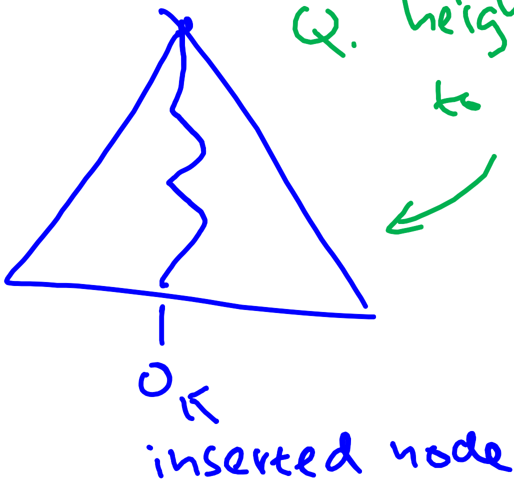
balance(17) = -2



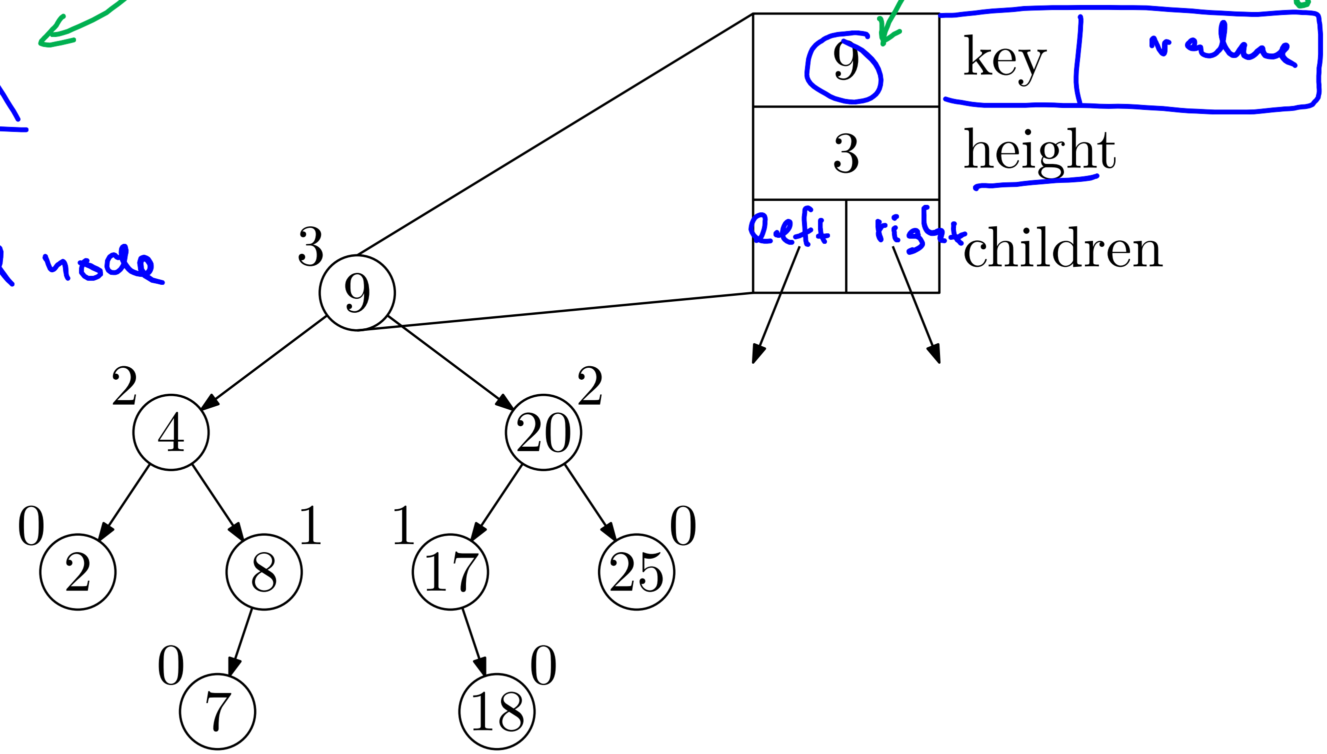
← heights

An AVL Tree

Q. heights of which nodes need to be recalculated?

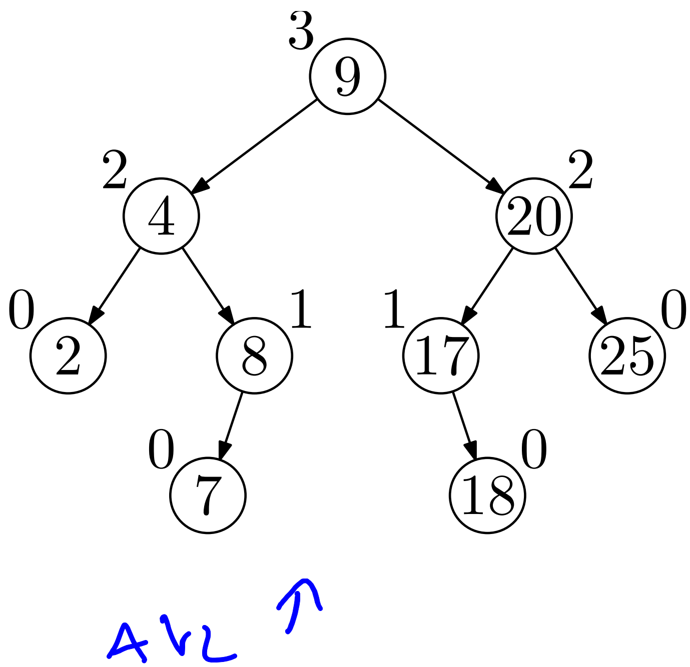


doesn't need to be an integer

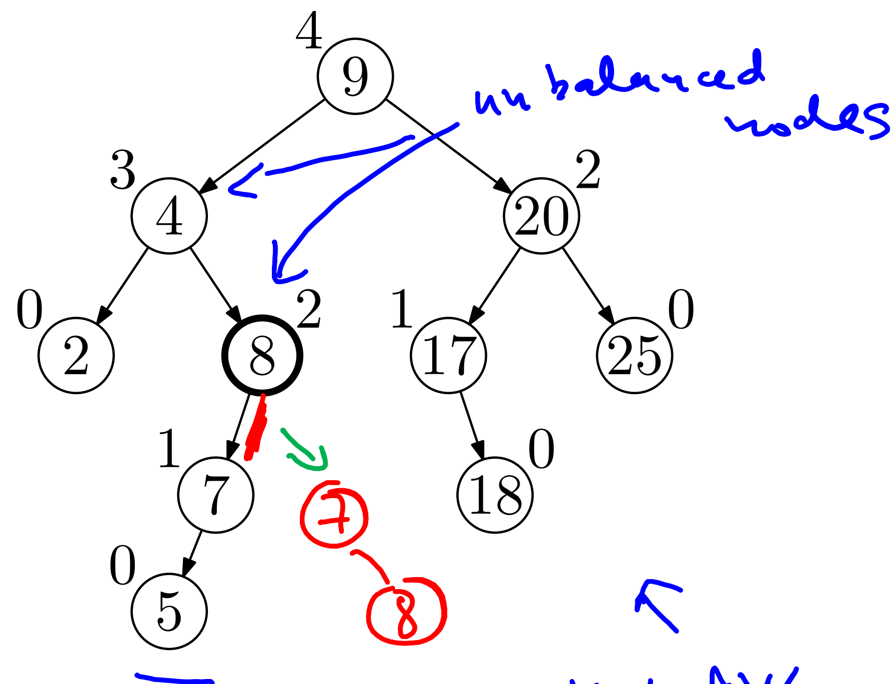


How Do We Stay Balanced?

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



and insert 5

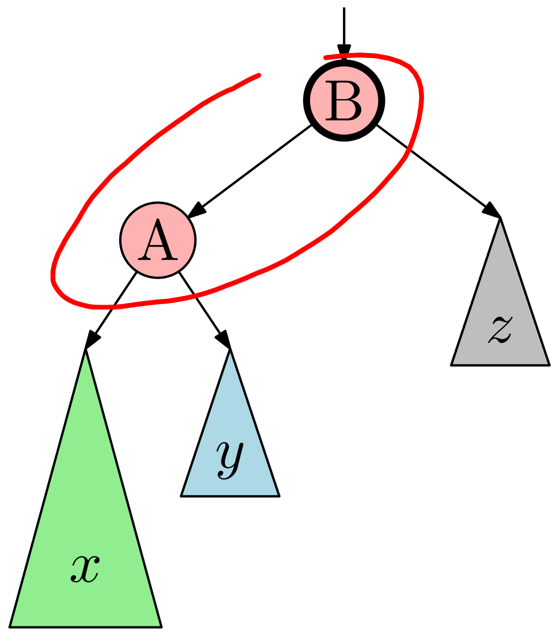


It's no longer an AVL tree. What can we do?

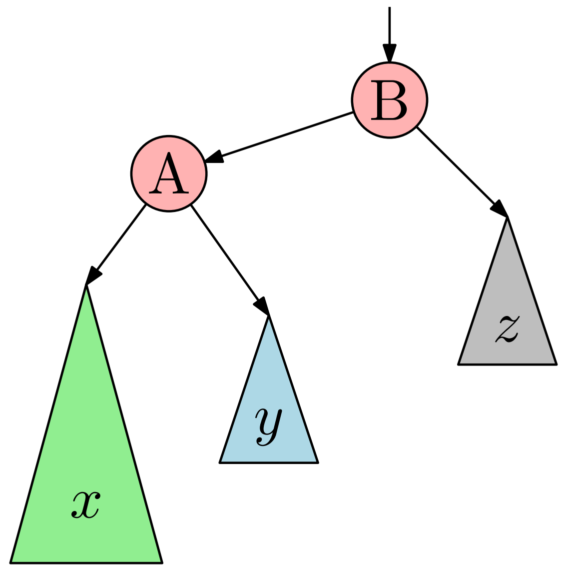
right rotate

ROTATE!

Rotation Animation



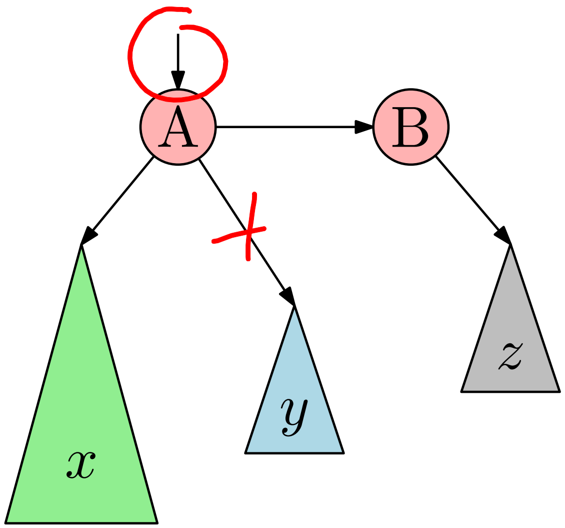
Rotation Animation



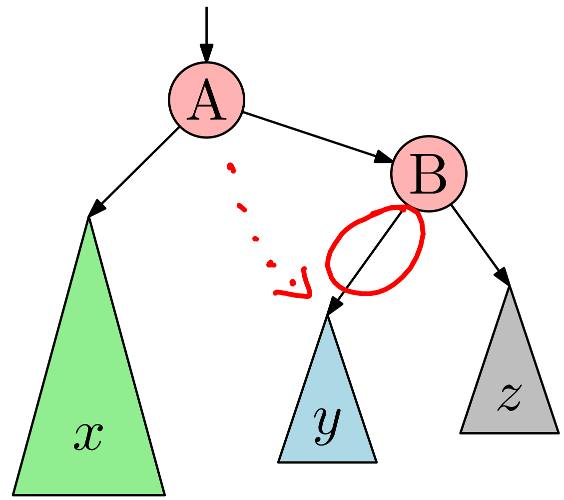
A < B

A < y < B

Rotation Animation



Rotation Animation



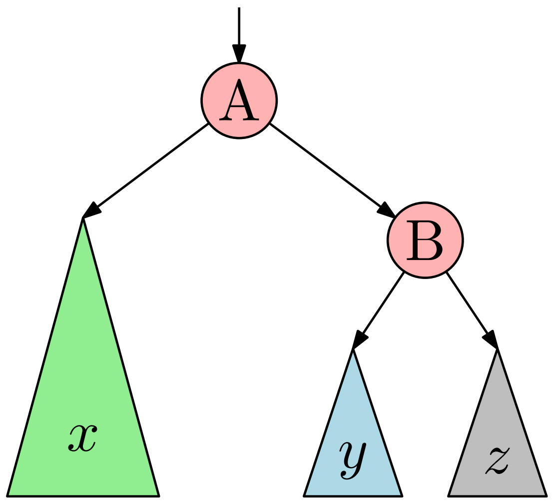
$A < B$

$A < y < B$

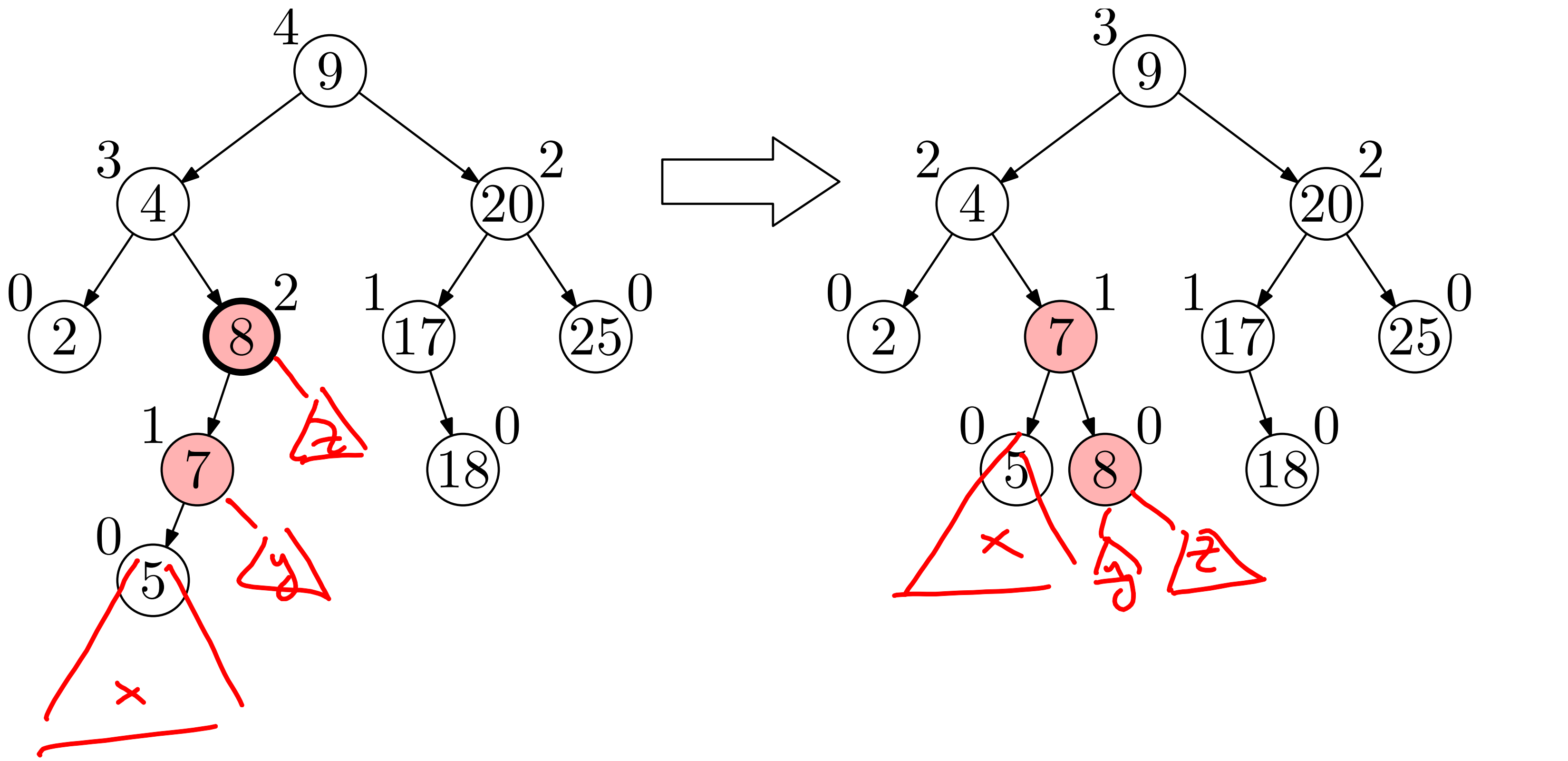
still

binary search tree

Rotation Animation



Single Rotation



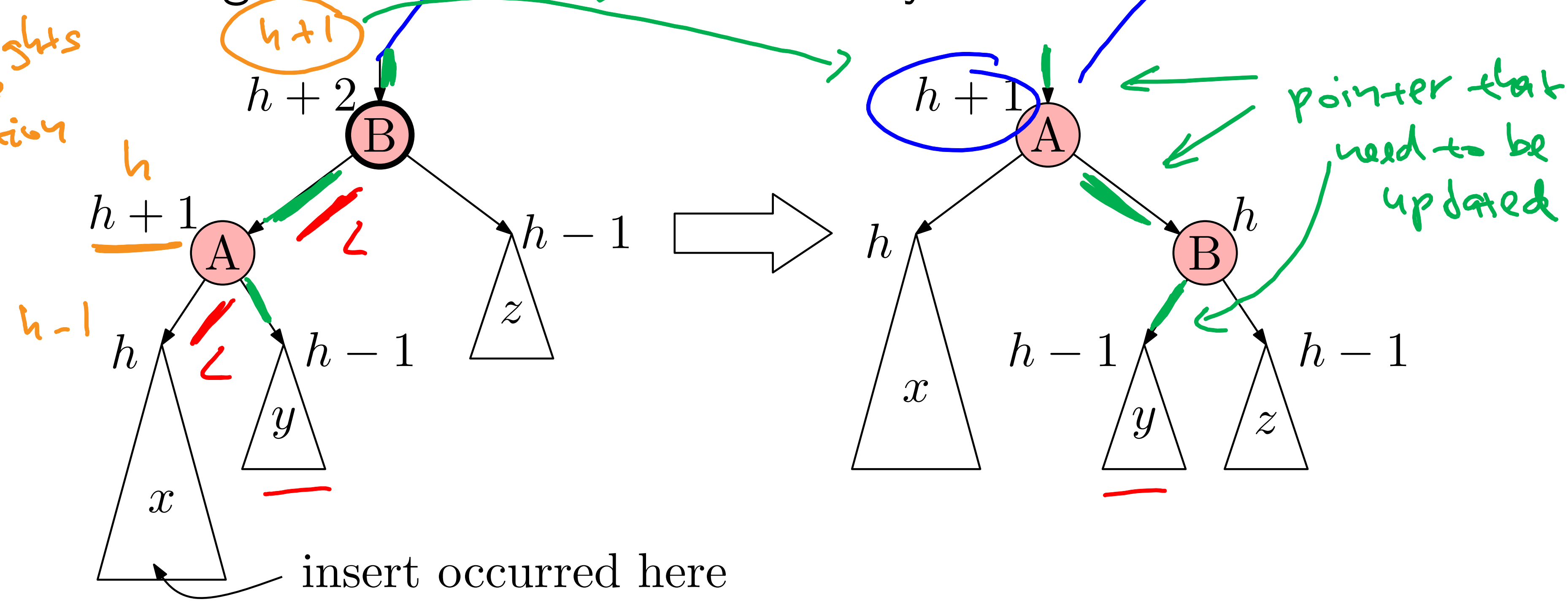
Single Rotation

(LL case)

height didn't change
⇒ no need to check above

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

• heights before insertion



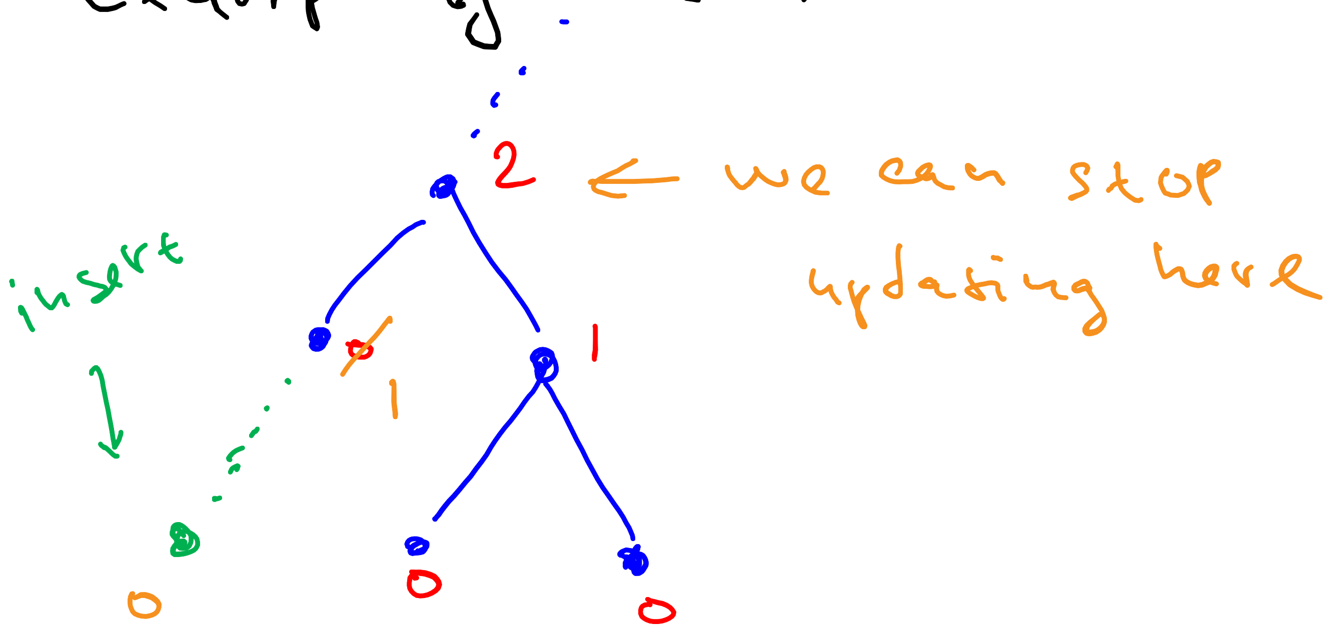
After rotation, subtree's height is the same as before insert.

So heights of ancestors don't change.

So? Increase heights of ancestors of inserted node until:

1) reach the root
2) node's height doesn't change
3) reach a node with imbalance ⇒ rotation & stop!

Example of case 2):

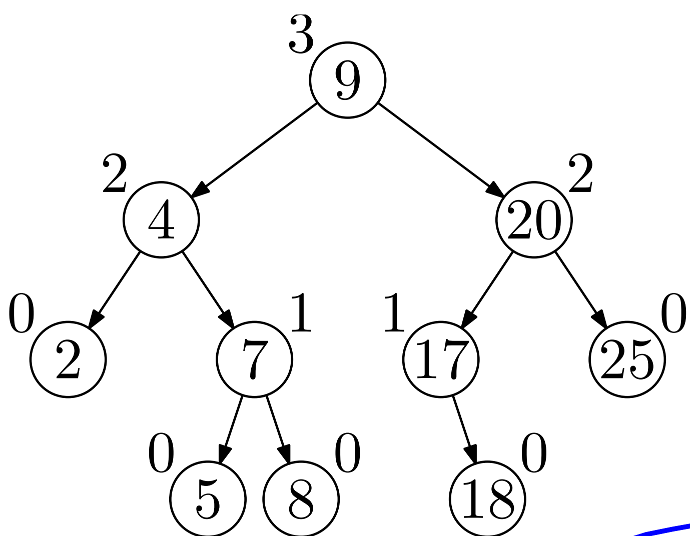


in red: original height

in orange: new height

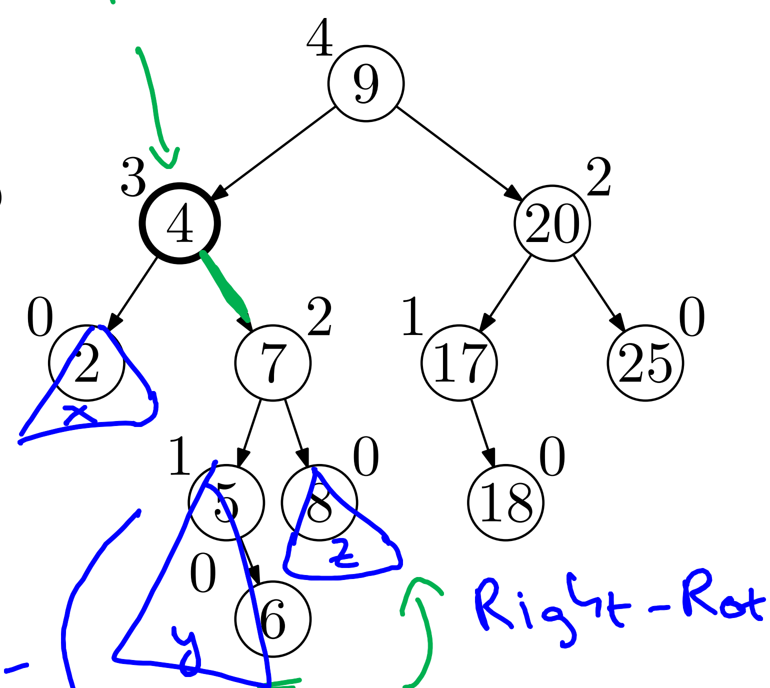
Double Rotation

Start with



and insert 6

imbalanced node

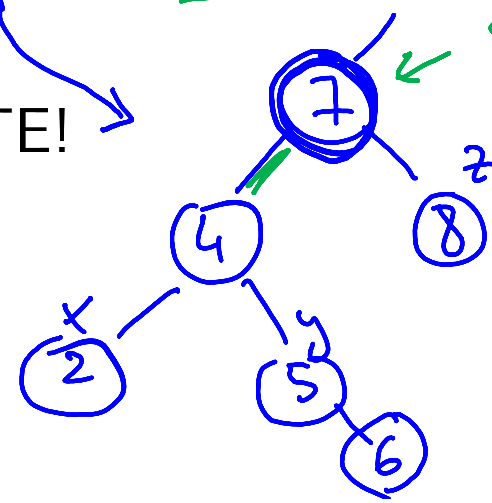


Right-Rot

A single rotation won't fix this.

Left-Rot

still imbalanced

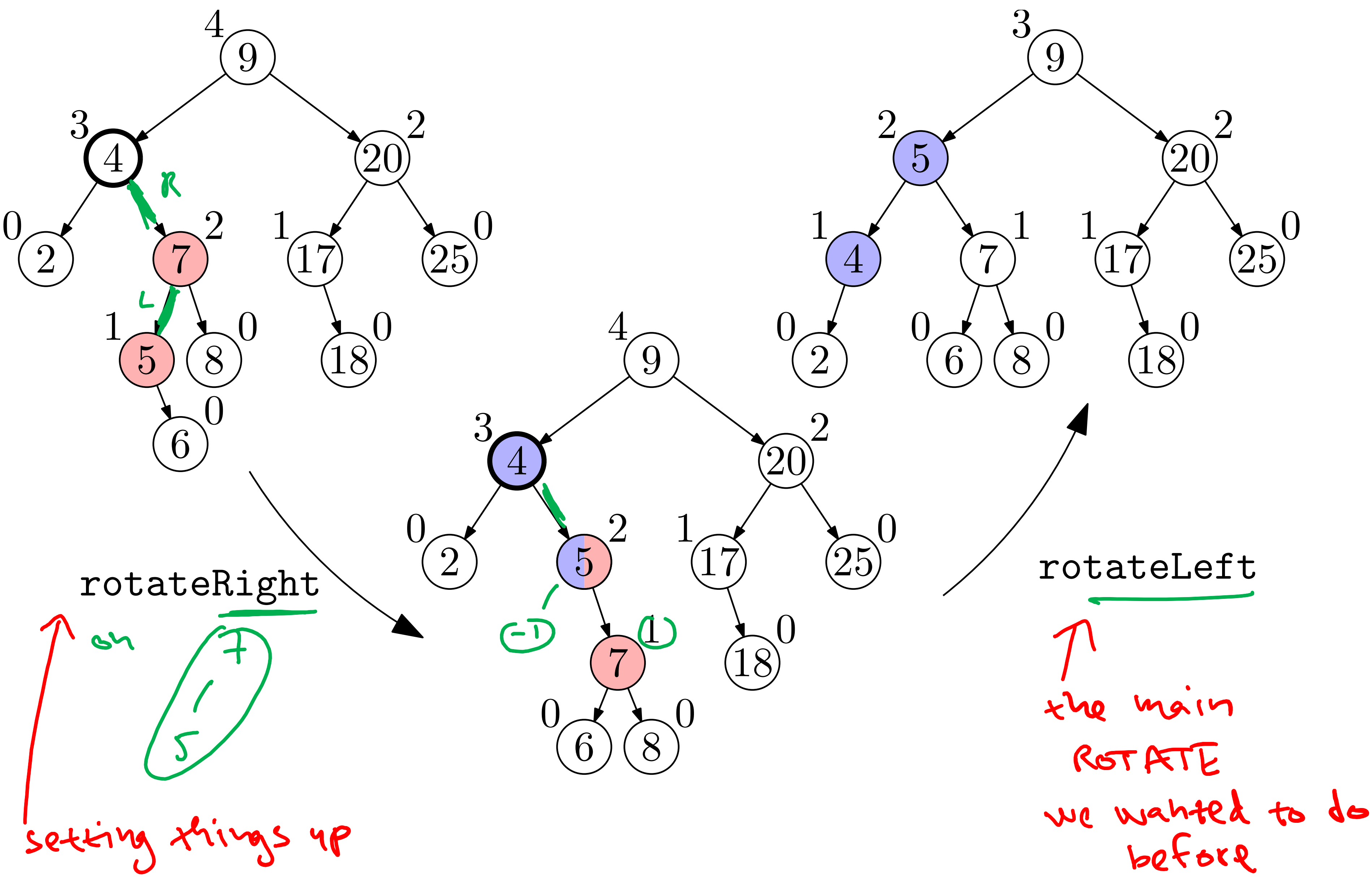


DOUBLE ROTATE!

WE NEED

(next slide)

Double Rotation Left



Double Rotation (RL case)

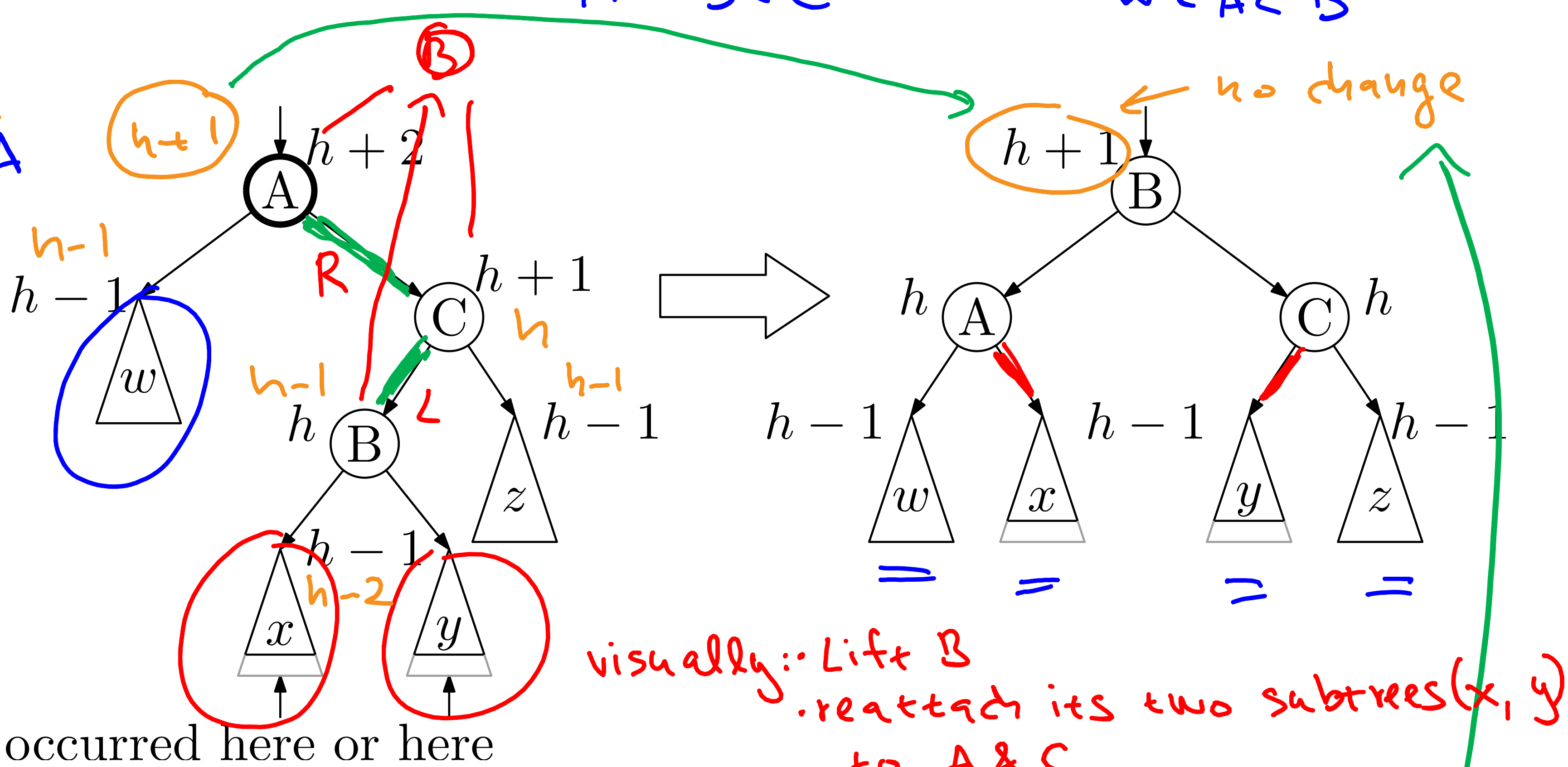
doubleRotateLeft is shown. There's also a symmetric doubleRotateRight.

A < B < C

w < A < B

heights before insertion

w < A



visually: · Lift B
· reattach its two subtrees (x, y) to A & C

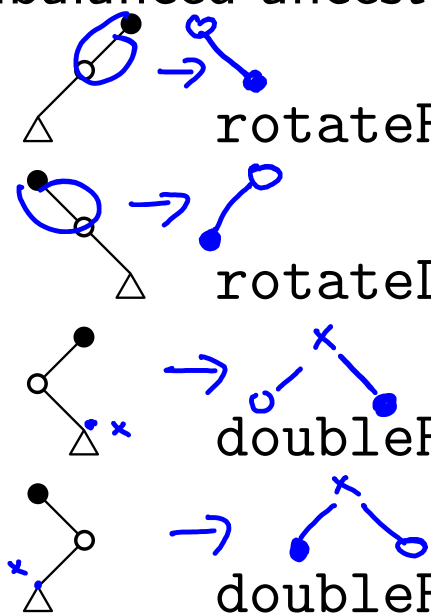
Either x or y increased to height $h - 1$ after insert.
After rotation, subtree's height is the same as before insert.
So height of ancestors doesn't change.

Insert Algorithm

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

● inbalanced node

△ insertion in this subtree.



Case LL: `rotateRight`

Case RR: `rotateLeft`

Case LR: `doubleRotateRight`

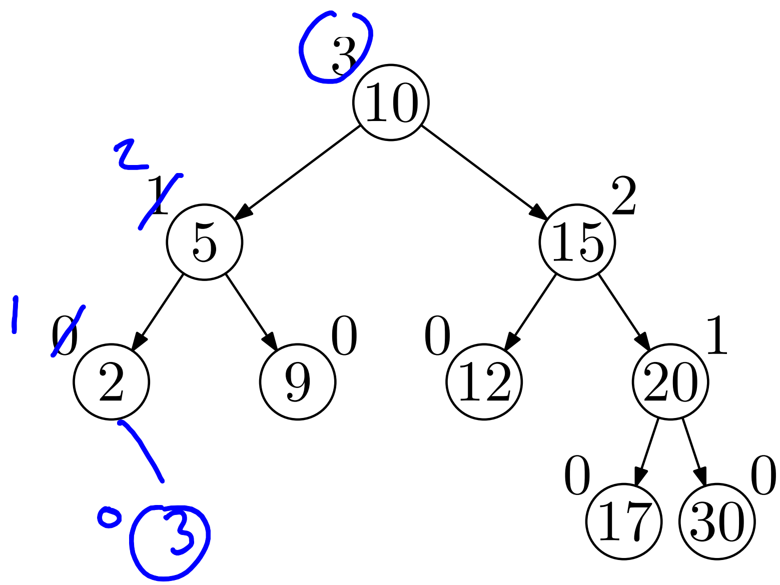
Case RL: `doubleRotateLeft`

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

worst-case time complexity: $\Theta(H) = \Theta(\log n)$ because it's balanced!

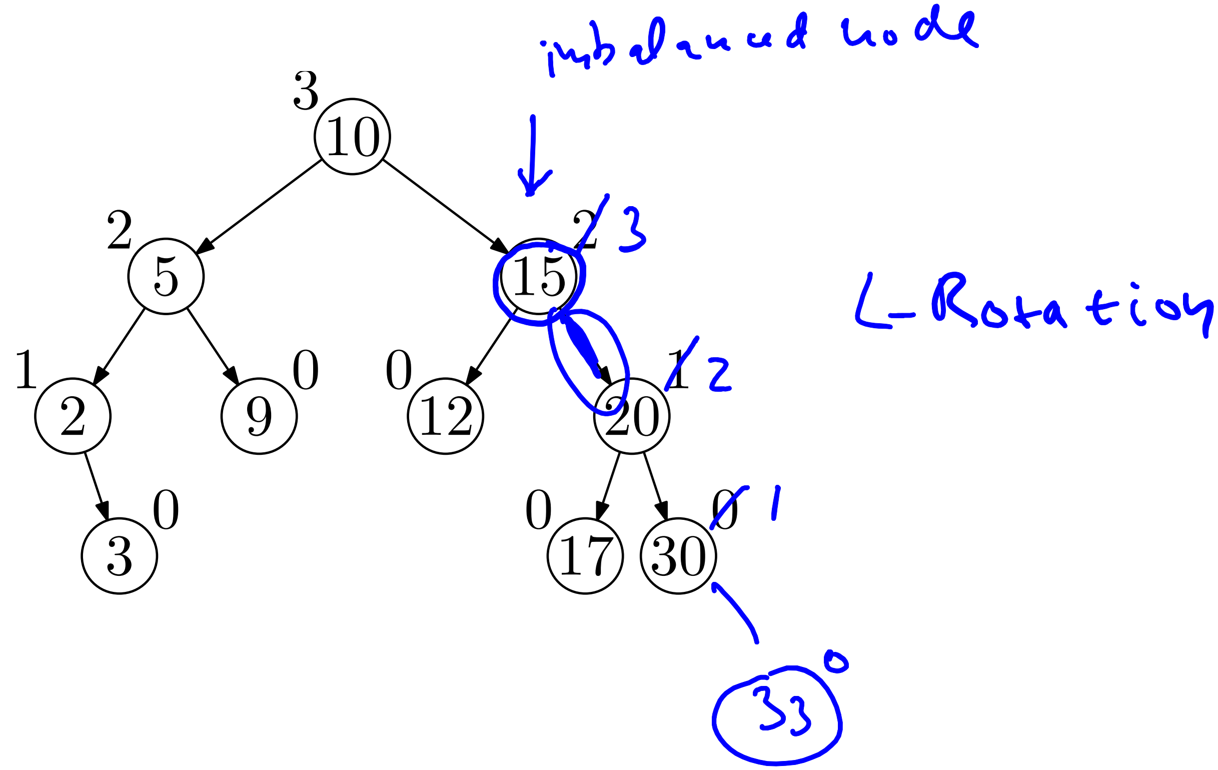
Insert: No Imbalance

Insert(3)

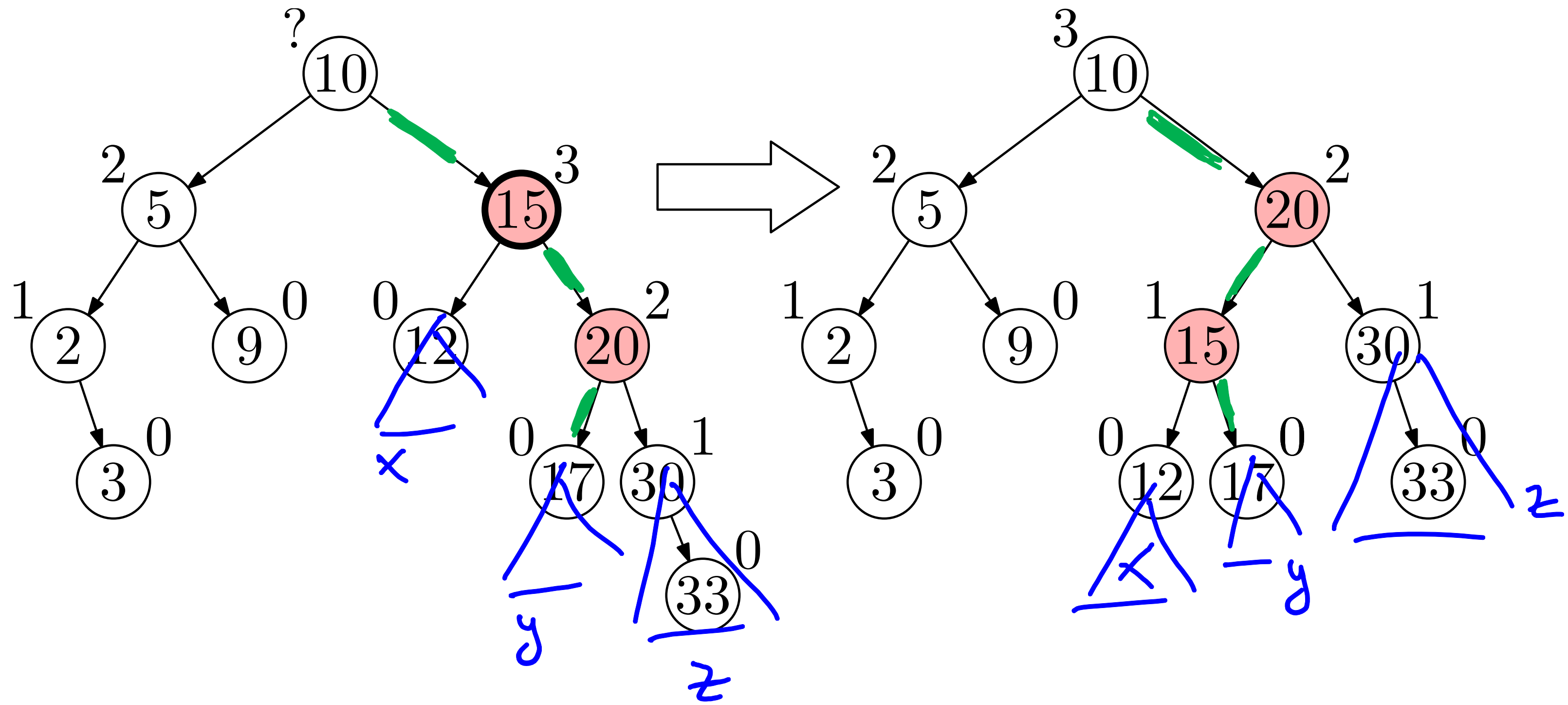


Insert: Imbalance Case RR

Insert(33)

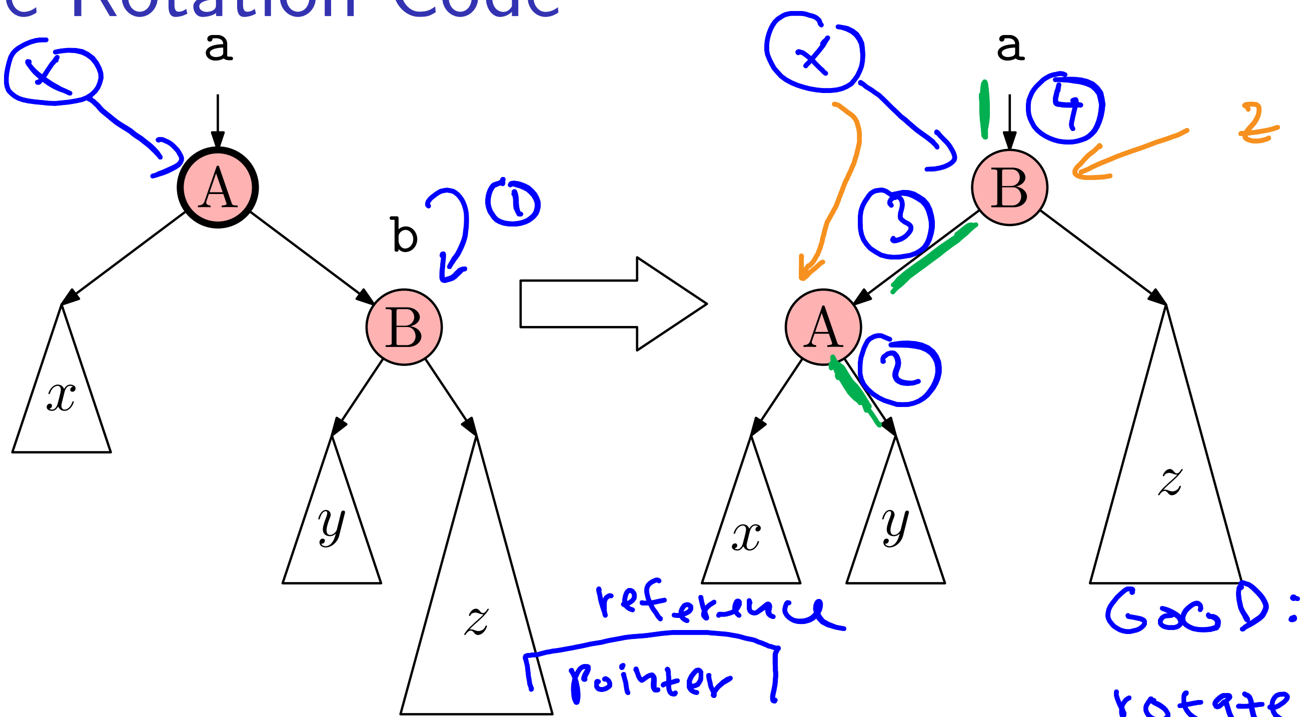


Case RR: `rotateLeft`



green: changed pointers

Single Rotation Code



reference

pointer

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

GOOD:

rotateLeft(x->right);

BAD:

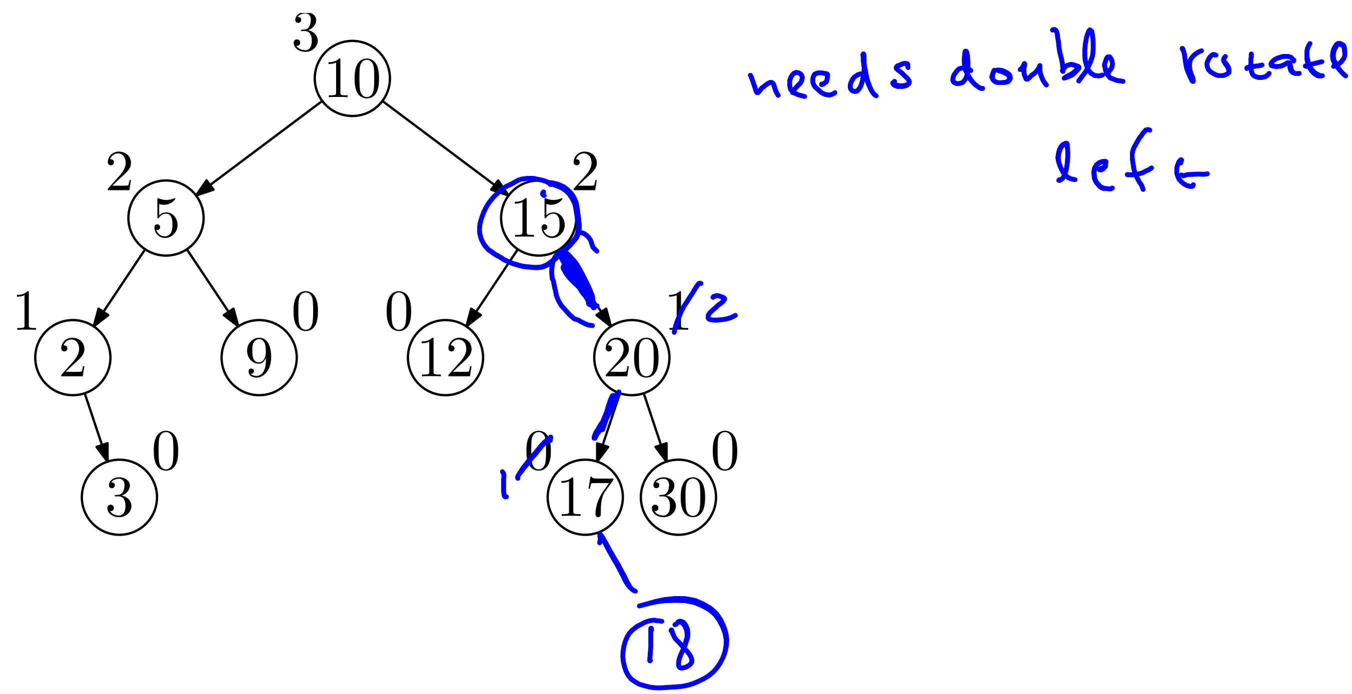
Node * z = x->right;

rotateLeft(z);

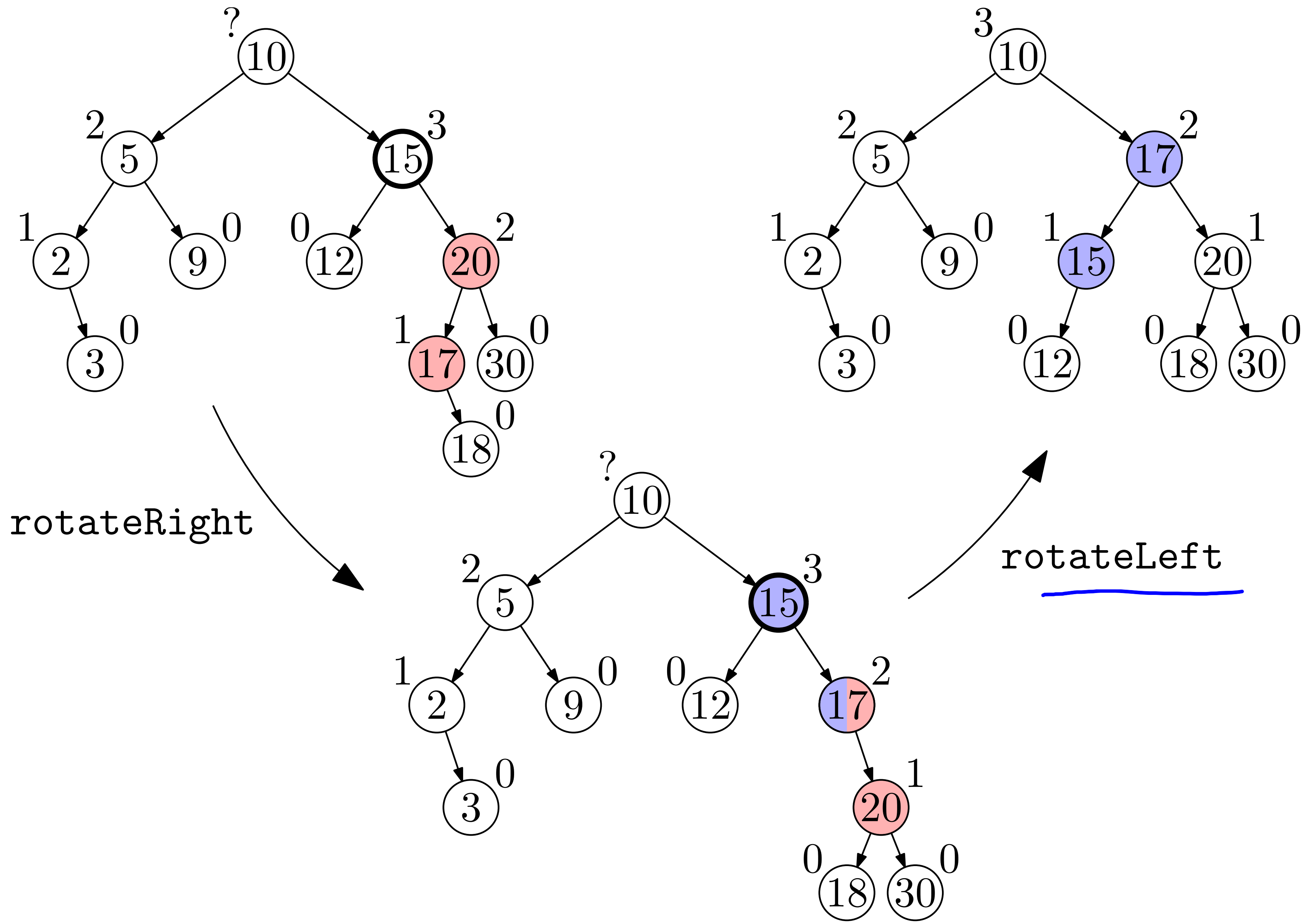
updates z, but does not update x->right (so it still points to A)

Insert: Imbalance Case RL

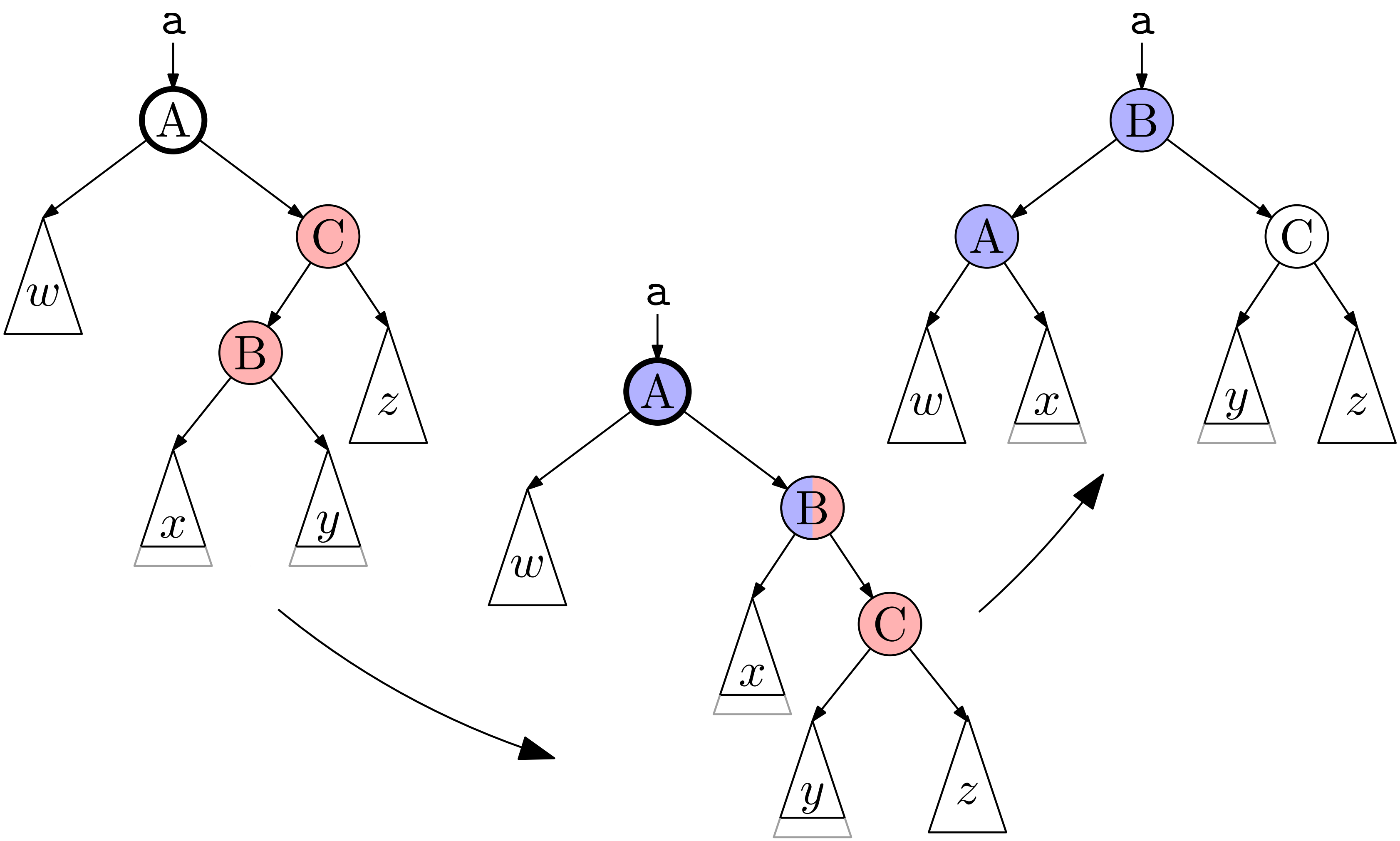
Insert(18)



Case RL: `doubleRotateLeft`



Double Rotation Code




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

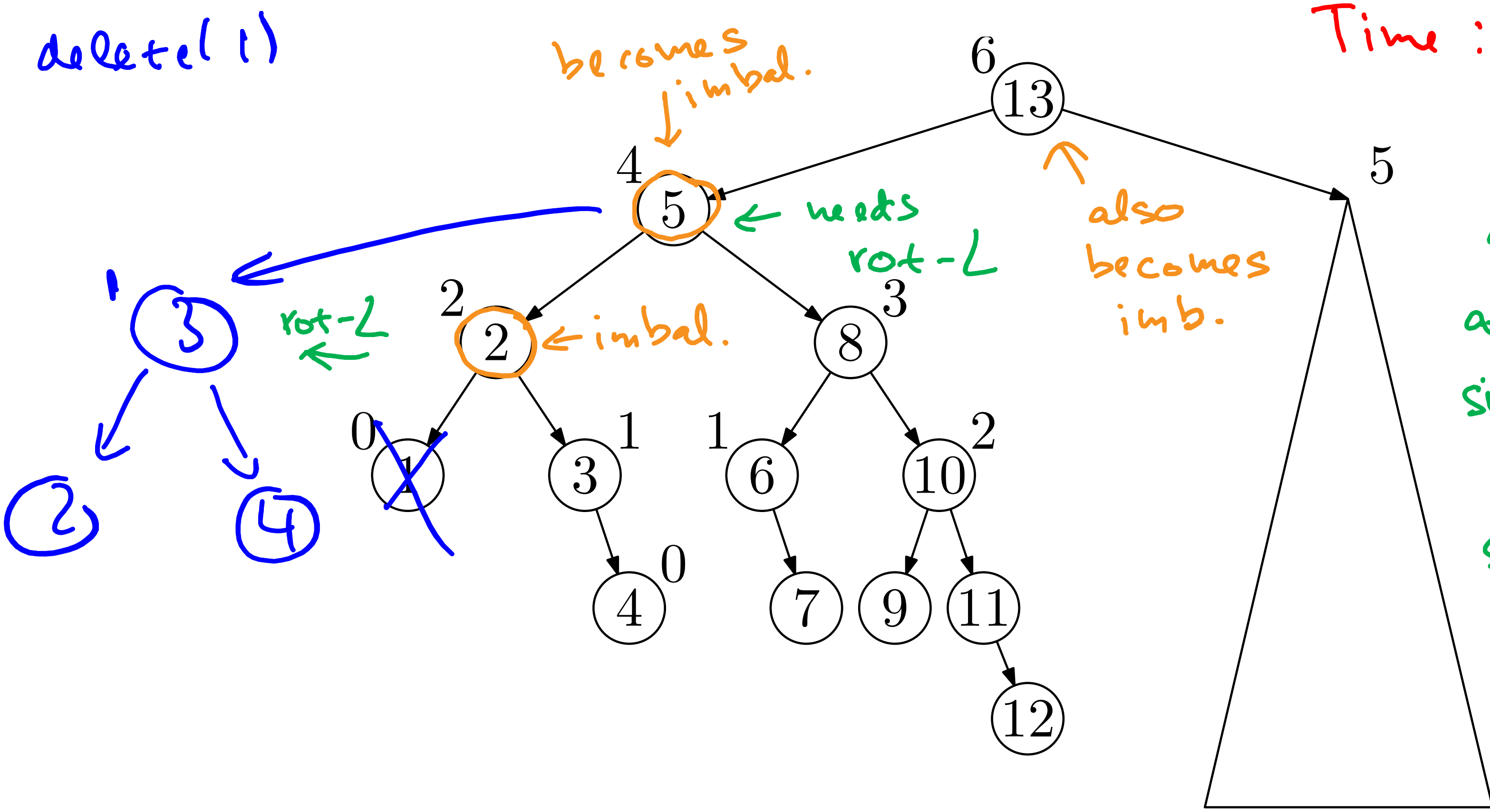
important: otherwise parent of A will not update its left/right pointer to B

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)

Worst-case Time: $\Theta(\log n)$

delete(1)

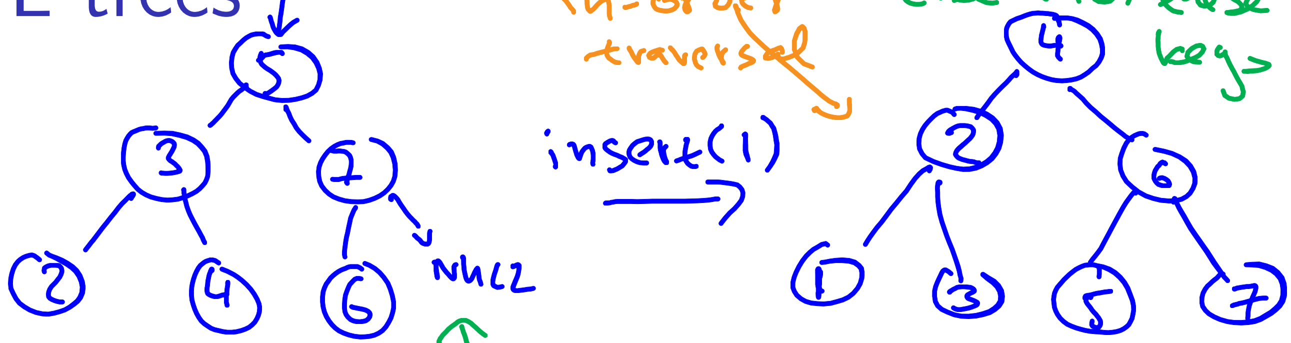


at each level at most one single/double rotation, so constant time per level

Thinking about AVL trees

filled by in-order traversal

the only nearly complete tree with these keys



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

each left & right pointer has changed! we need to update every node!

Realities

SQL server

- For large data sets, disk accesses dominate runtime

let's try nearly complete

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

nearly complete trees would require $\Omega(n)$ operations for insert.