CSCI 136
Data Structures & Advanced Programming

Lecture 25

Fall 2018

Instructor: B^2

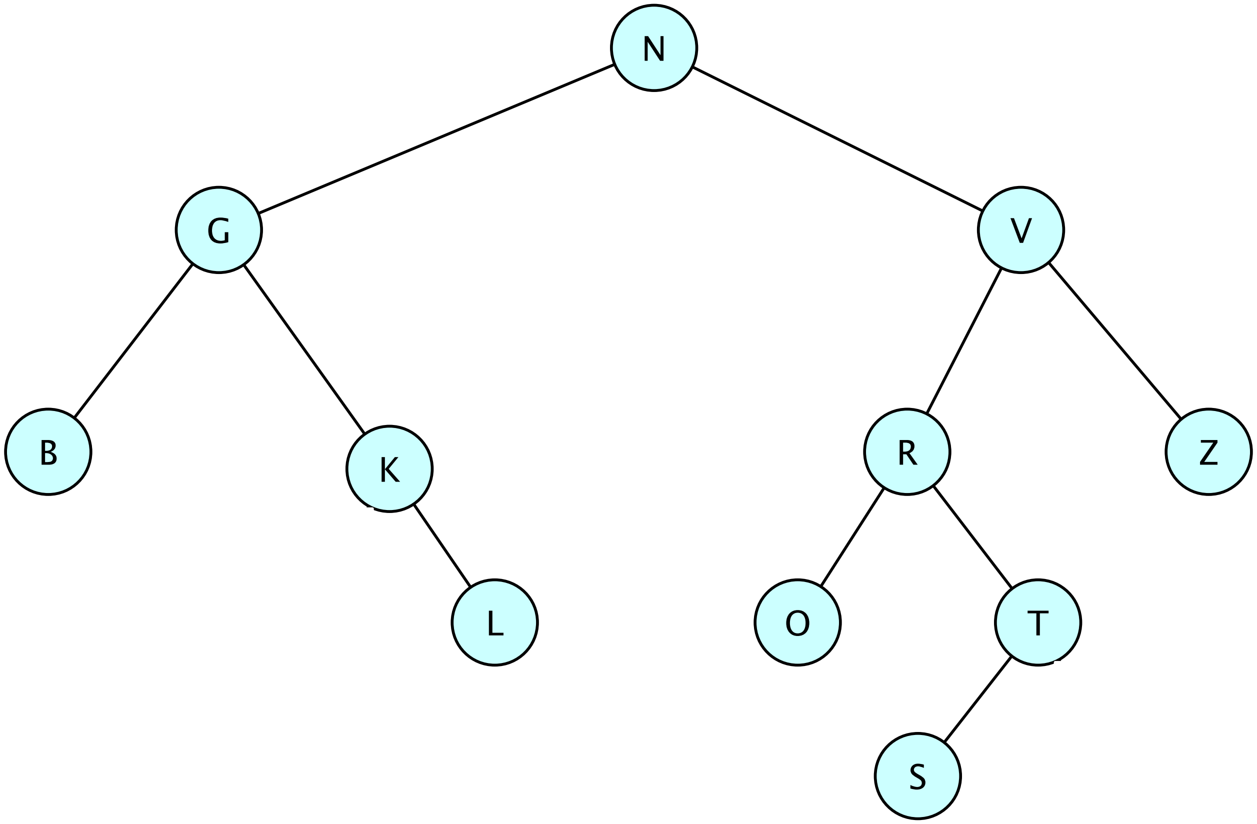
Last Time

- Binary search trees (Ch 14)
    - The *locate* method
    - Further Implementation

Today' s Outline

- Binary search trees (Ch 14)
  - Implementation wrap-up
- Tree balancing to maintain small height
  - AVL Trees
- Partial taxonomy of balanced tree species
  - Red-Black Trees
  - Splay Trees

Add: Repeated Nodes



Where would a new K be added?
A new V?

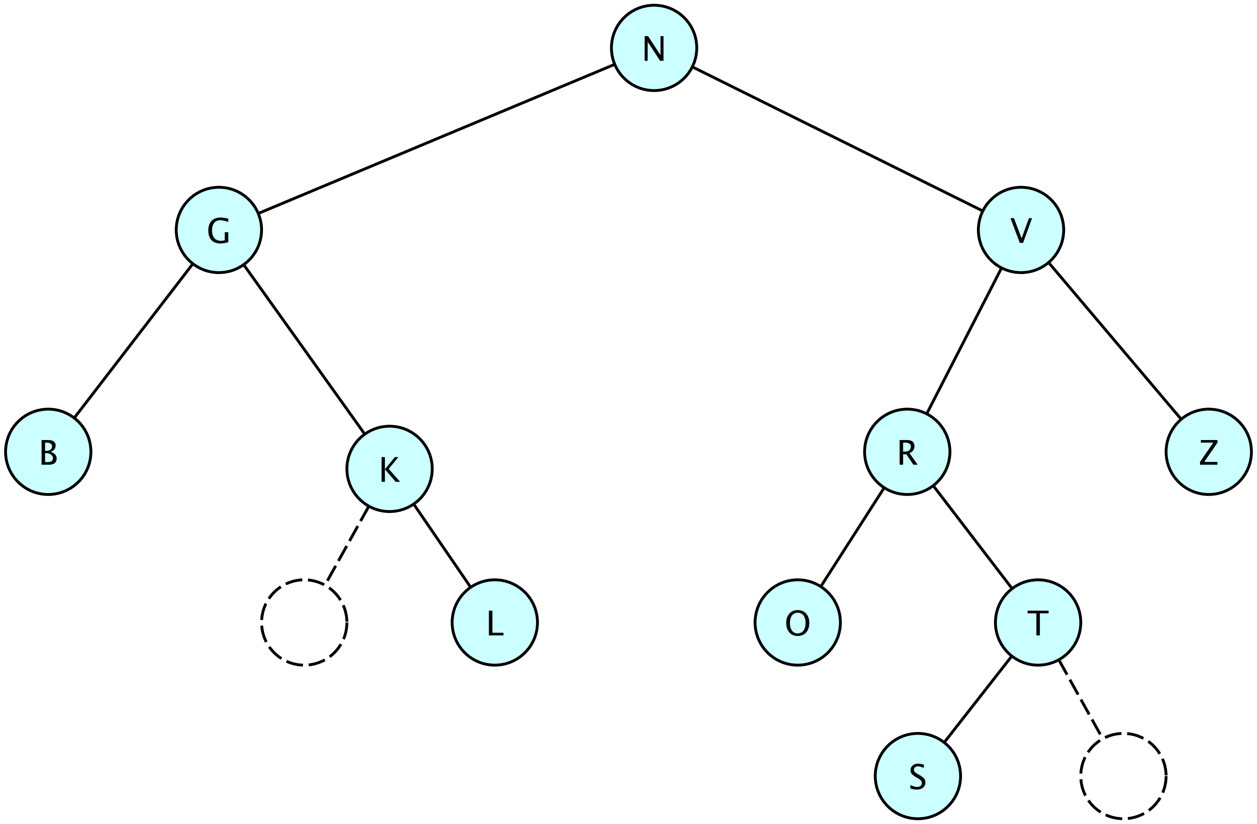
Add Duplicate to Predecessor

- If insertLocation has a left child then
  - Find insertLocation's predecessor
  - Add repeated node as right child of predecessor
  - Predecessor will be in insertLocation's left sub-tree
    - Do you believe me?

Corrected Version: add(E value)

```
BinaryTree<E> newNode = new BinaryTree<E>(value,EMPTY,EMPTY);
if (root.isEmpty()) root = newNode;
else {
    BinaryTree<E> insertLocation = locate(root,value);
    E nodeValue = insertLocation.value();
    if (ordering.compare(nodeValue,value) < 0)
        insertLocation.setRight(newNode);
    else
        if (insertLocation.left().isEmpty())
            insertLocation.setLeft(newNode);
        else
            // if value is in tree, we insert just before
            predecessor(insertLocation).setRight(newNode);
}
count++;
```

How to Find Predecessor



Where would a new K be added?
A new V?

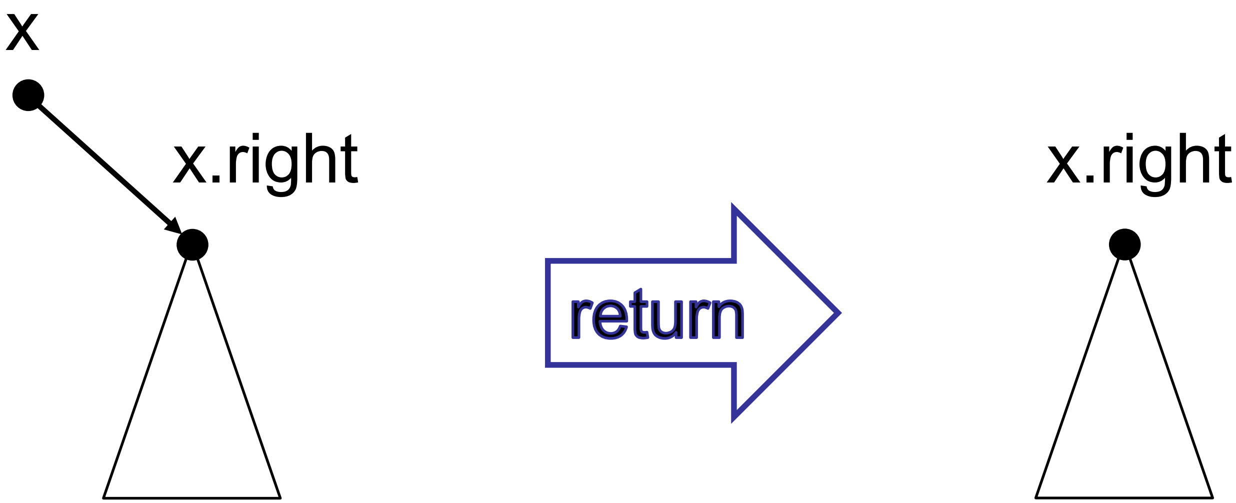
Predecessor

```
protected BinaryTree<E> predecessor(BinaryTree<E> root) {
   Assert.pre(!root.isEmpty(), ”Root has predecessor");
   Assert.pre(!root.left().isEmpty(),"Root has left child.");

   BinaryTree<E> result = root.left();

   while (!result.right().isEmpty())
      result = result.right();

   return result;
}
```

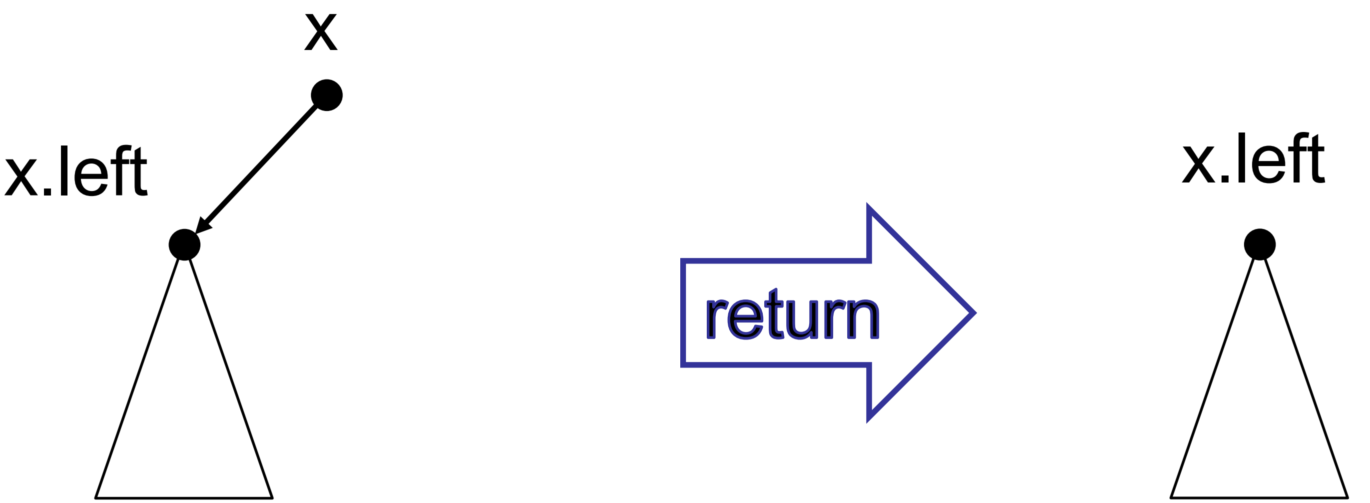
Removal

- Removing the root is a (not so) special case
- Let’s figure that out first
  - If we can remove the root, we can remove any element in a BST in the same way
    - Do you believe me?
- We need to implement:
  - `public E remove(E item)`
  - `protected BT removeTop(BT top)`

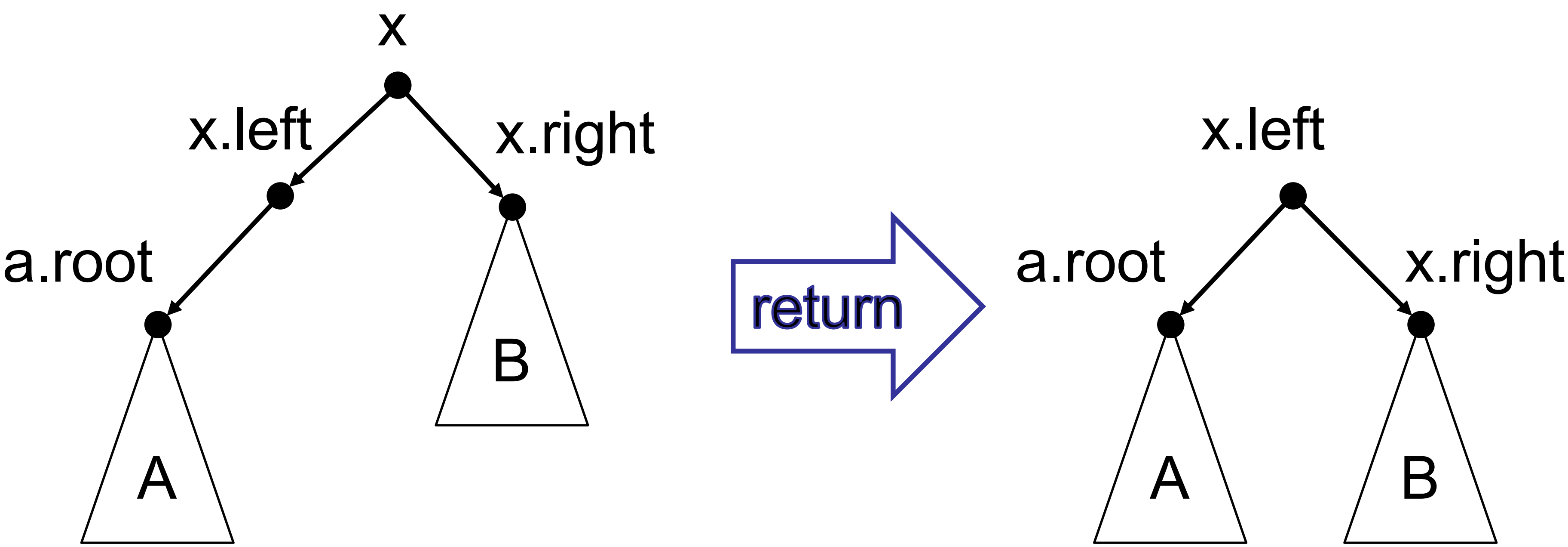
Case 1: No left binary tree



Case 2: No right binary tree



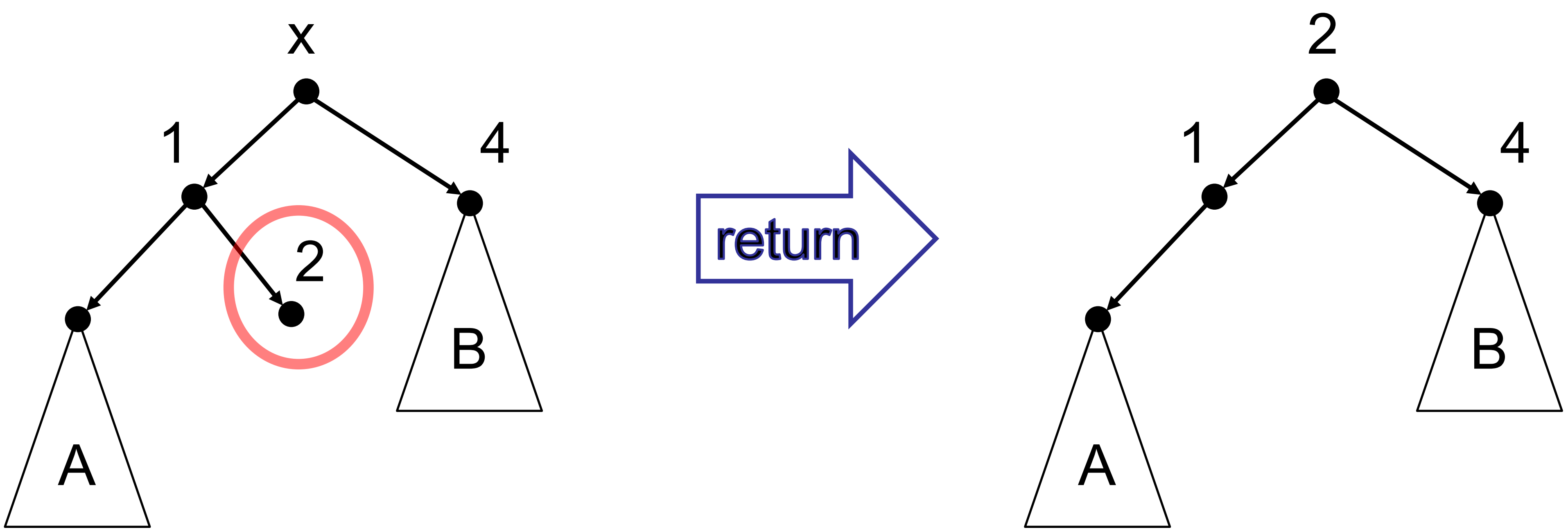
Case 3: Left has no right subtree



Case 4: General Case (HARD!)

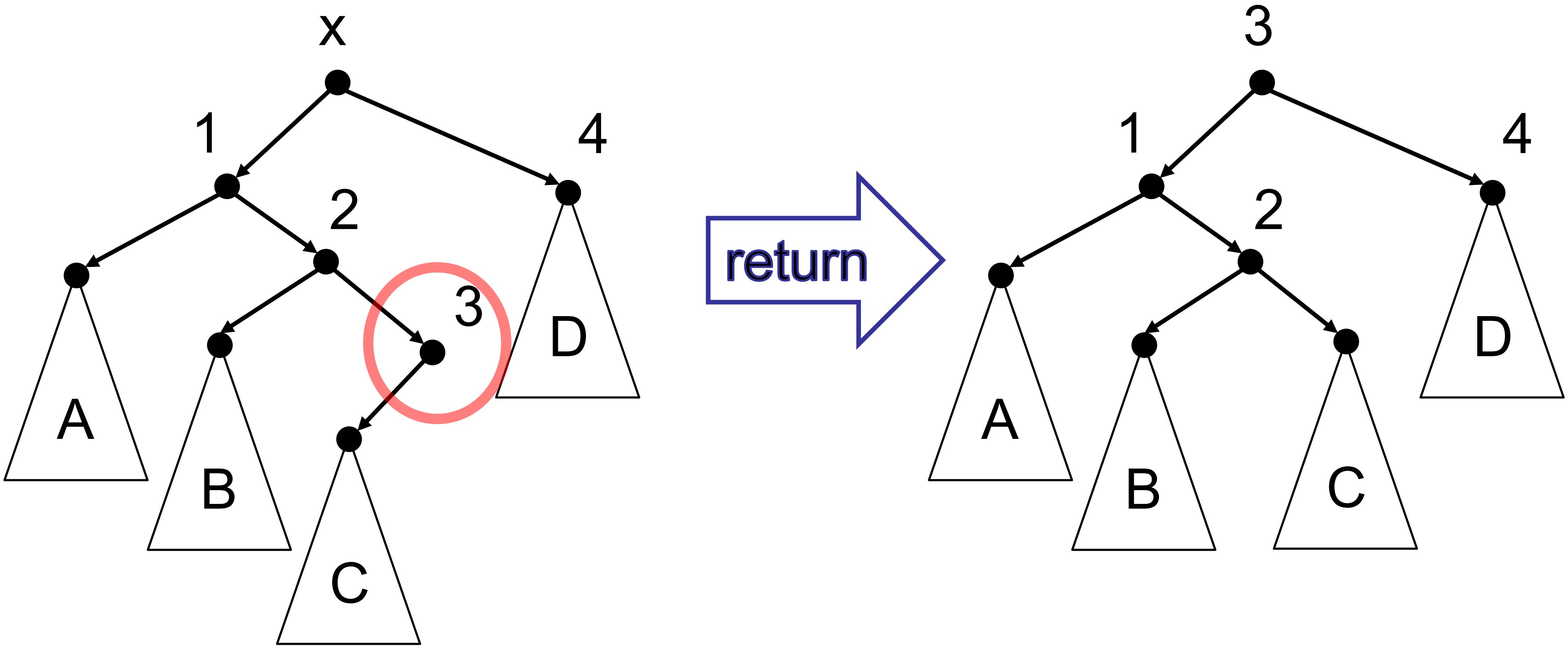
- Consider BST requirements:
  - Left subtree must be <= root
  - Right subtree must be > root
- Strategy: replace the root with the largest value that is less than or equal to it
  - predecessor(root) : rightmost left descendant
- This may require reattaching the predecessor's left subtree!

Case 4: General Case (HARD!)



Replace root with predecessor(root), then patch up the remaining tree

Case 4: General Case (HARD!)



Replace root with predecessor(root),
then patch up the remaining tree

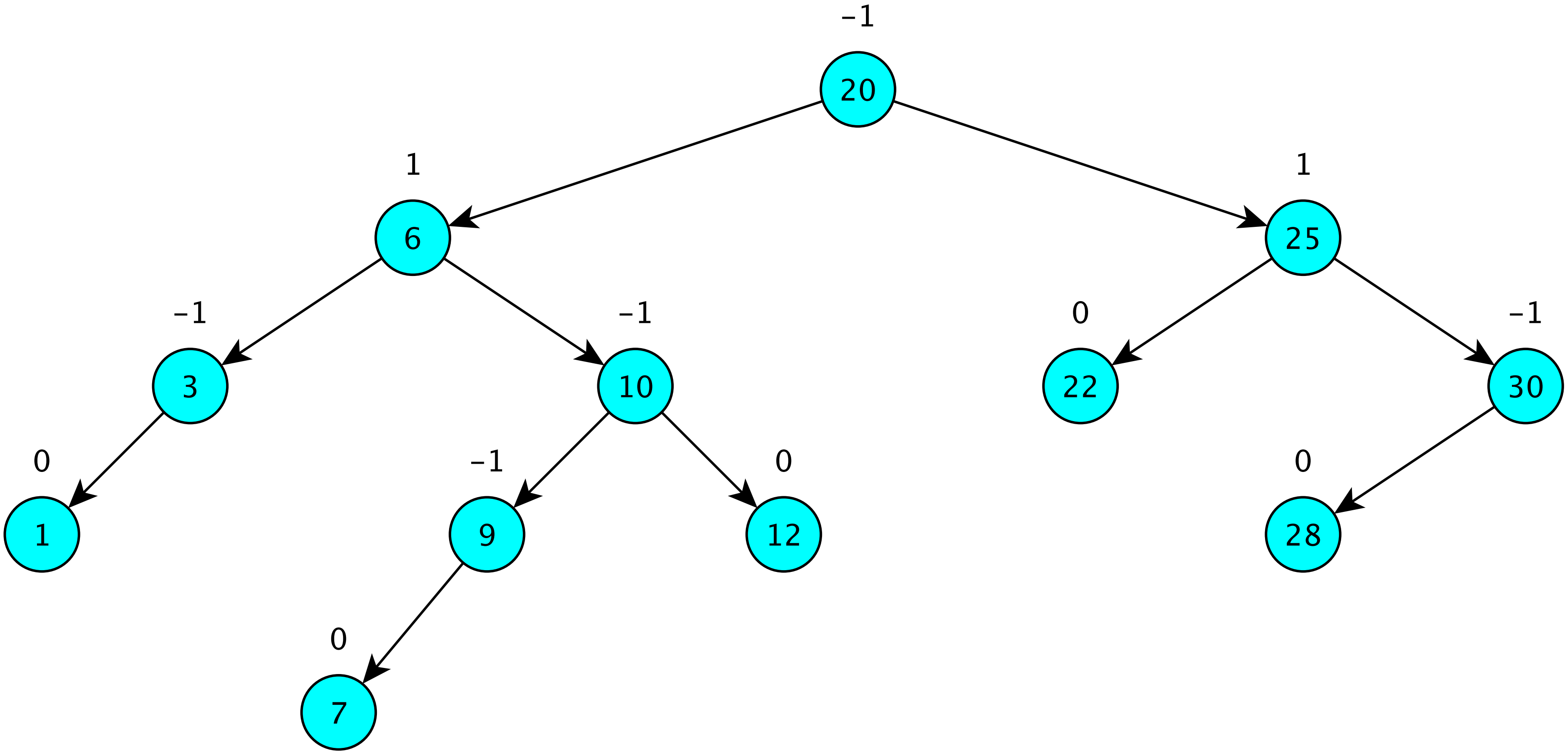
RemoveTop(topNode)

```
Detach left and right sub-trees from root (i.e. topNode)
If either left or right is empty, return the other one
If left has no right child
    make right the right child of left then return left
Otherwise find largest node C  in left
    // C is the right child of its own parent P
    // C is the predecessor of right (ignoring topNode)
Detach C from P; make C's left child the right child of P
Make C new root with left and right as its sub-trees
```

But What About Height?

- Can we design a binary search tree that is always “shallow”?
- Yes! In many ways. Here’s one
- AVL trees
  - Named after its two inventors, G.M. Adelson-Velsky and E.M. Landis, who published a paper about AVL trees in 1962 called "An algorithm for the organization of information"

AVL Trees



AVL Trees

- Balance Factor of a binary tree node:
  - height of right subtree minus height of left subtree.
  - A node with balance factor 1, 0, or -1 is considered *balanced.*
  - A node with any other balance factor is considered unbalanced and requires rebalancing the tree.
- Definition: An *AVL* Tree is a binary tree in which every node is balanced.

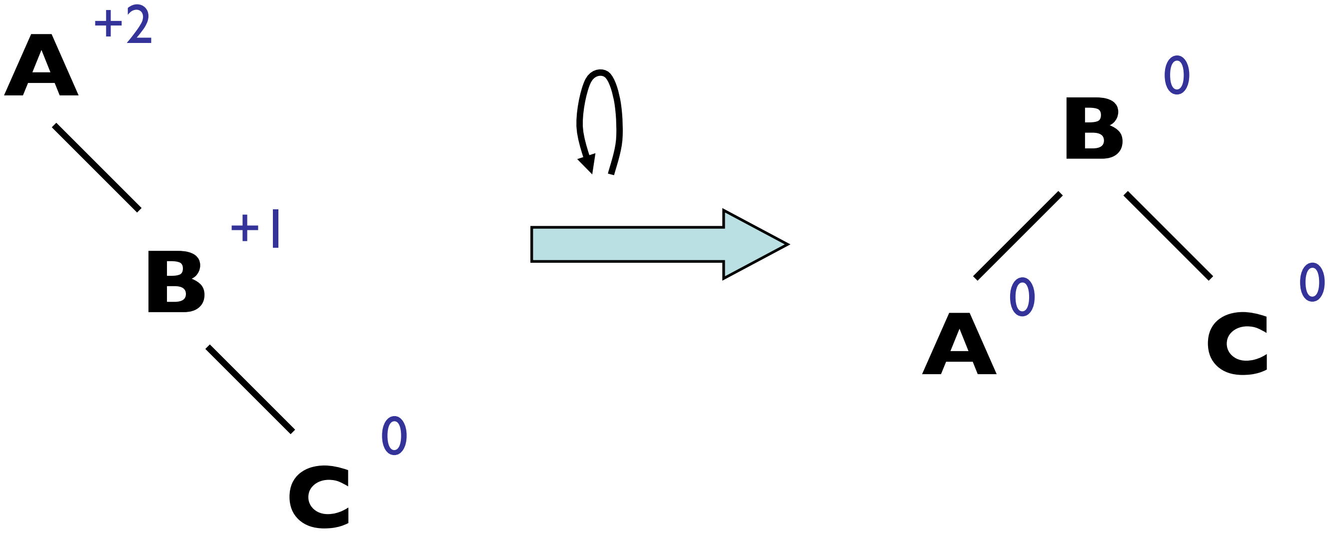
AVL Trees have O(log n) Height

Theorem: An AVL tree on n nodes has height O(log n)

Proof idea

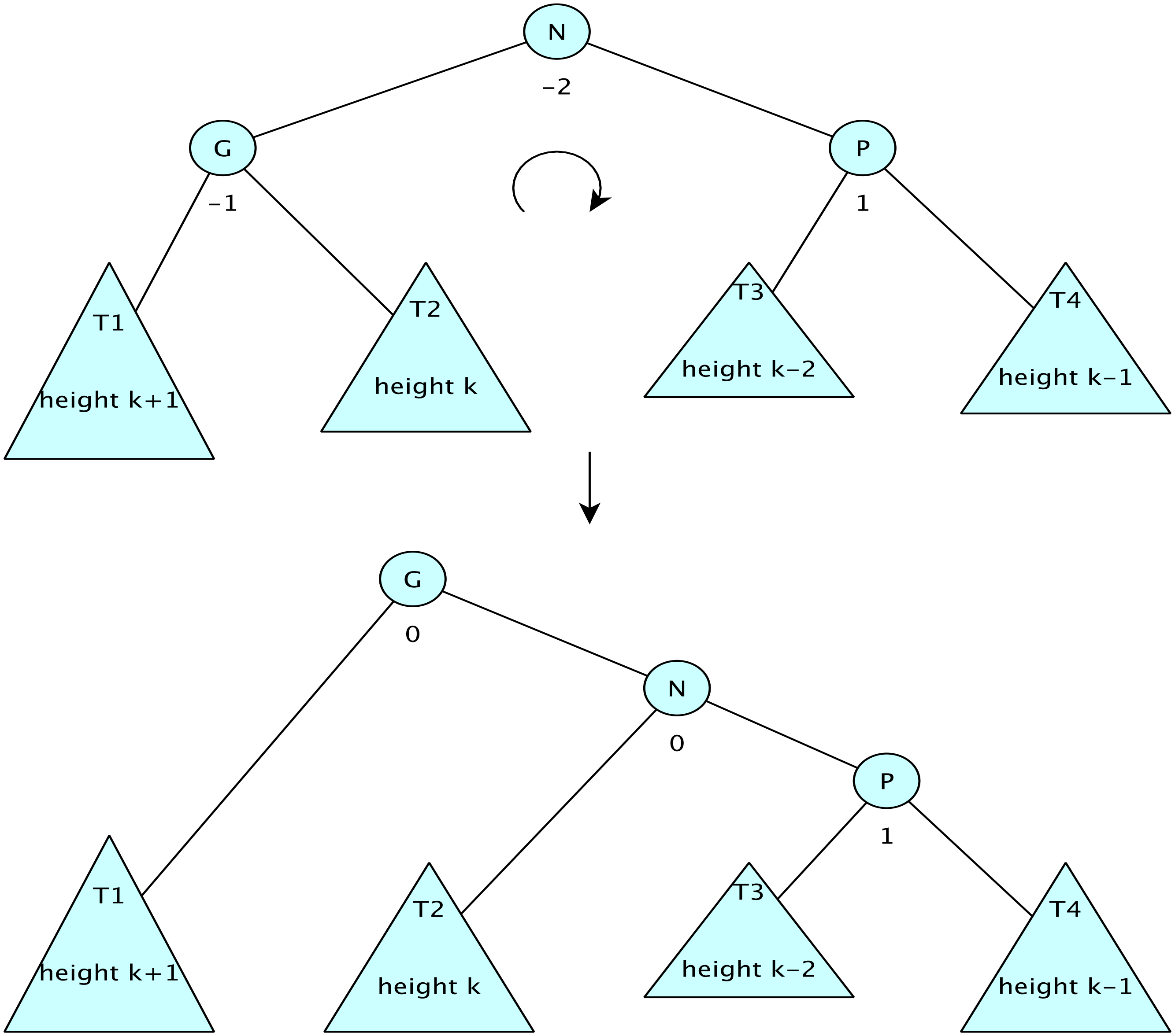
- Show that an AVL tree of height h has at least fib(h) nodes (easy induction proof---try it!)
- Recall (HW): $fib(h) \geq \left({}^{3}\!/_{2}\right)^{h}$ if h ≥ 10
- So $n \geq \left({}^{3}\!/_{2}\right)^{h}$ and thus $\log_{3/2} n \geq h$
  - Recall that for any $a, b > 0$, $\log_a n = \frac{\log_b n}{\log_b a}$
  - So $\log_a n$ and $\log_b n$ are Big-O of one another
- So h is O(log n)

Single Rotation

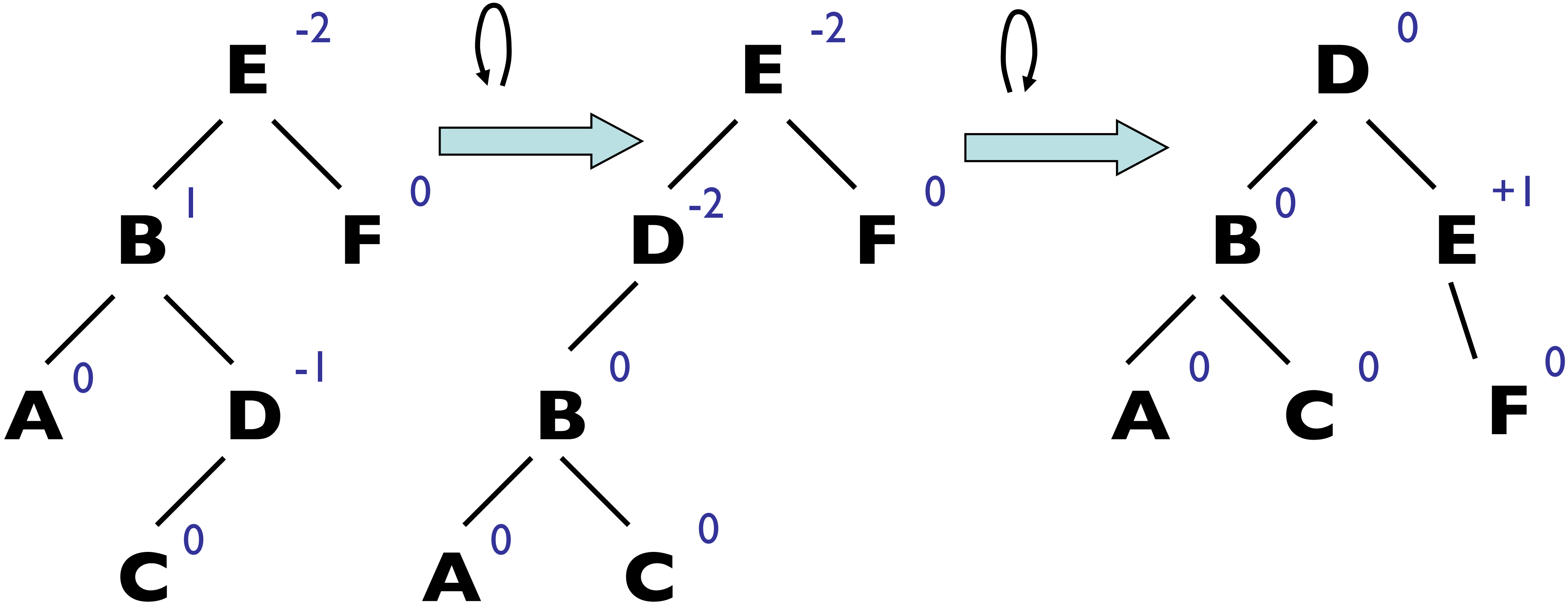


Unbalanced trees can be rotated to achieve balance.

Single Right Rotation



Double Rotation



AVL Tree Facts

- A tree that is AVL except at root, where root balance factor equals ±2 can be rebalanced with at most 2 rotations
- add(v) requires at most O(log n) balance factor changes and one (single or double) rotation to restore AVL structure
- remove(v) requires at most O(log n) balance factor changes and (single or double) rotations to restore AVL structure
- An AVL tree on n nodes has height O(log n)

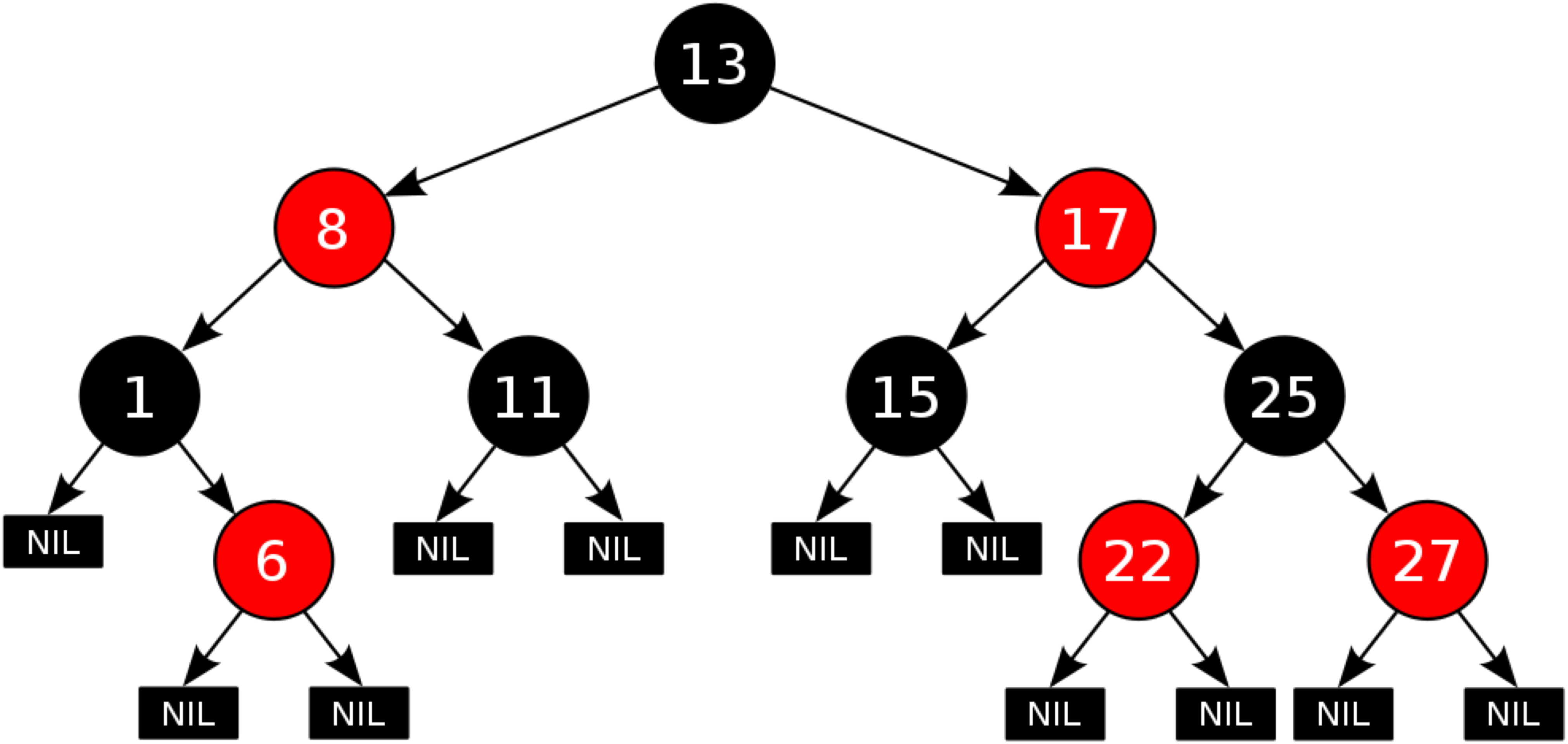
AVL Trees: One of Many

There are many strategies for tree balancing to preserve O(log n) height, including

- AVL Trees: guaranteed O(log n) height
- Red-black trees: guaranteed O(log n) height
- B-trees (not binary): guaranteed O(log n) height
  - 2-3 trees, 2-3-4 trees, red-black 2-3-4 trees, ...
- Splay trees: *Amortized* O(log n) time operations
- Randomized trees: O(log n) expected height

A Red-Black Tree

(from Wikipedia.org)



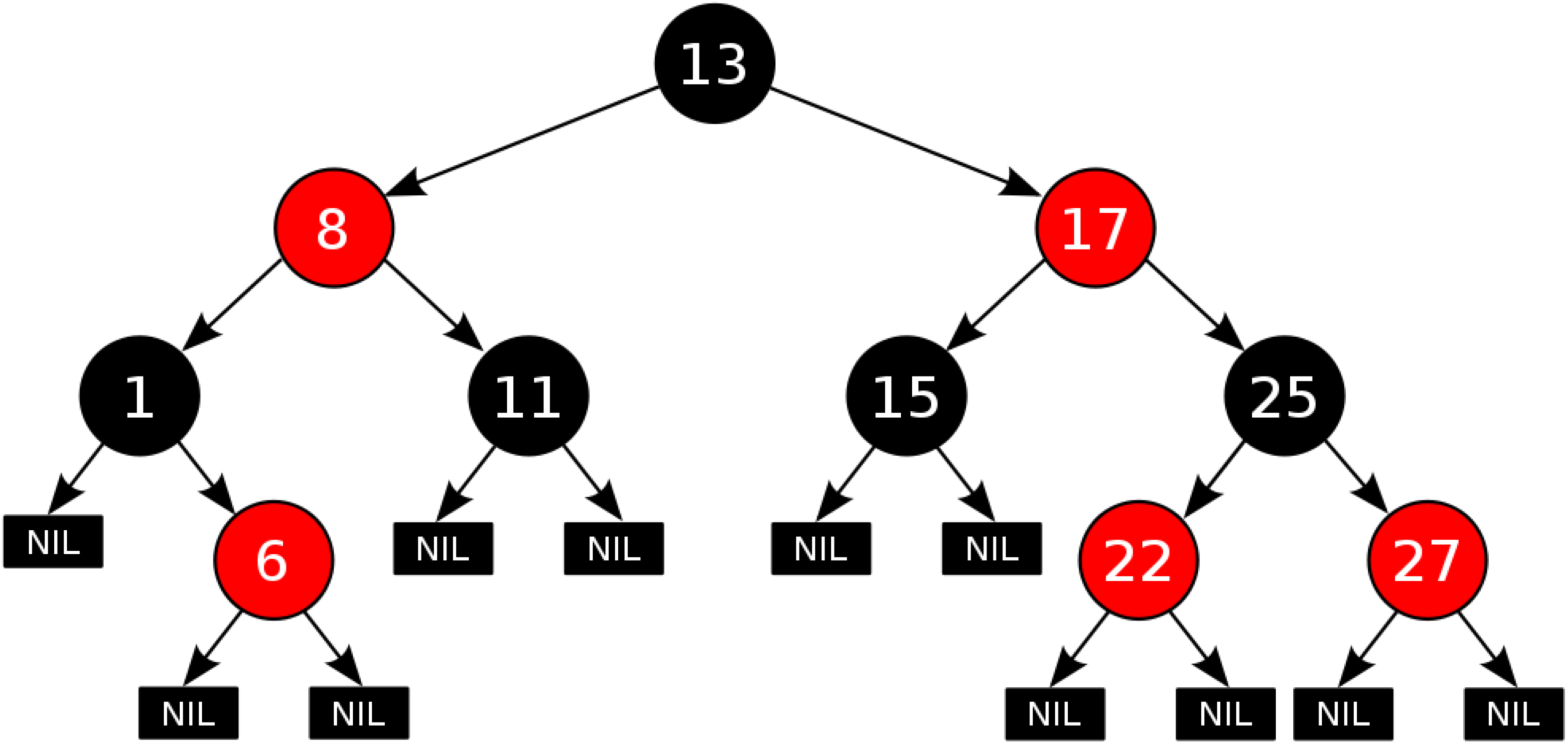
Red-Black Trees

Red-Black trees, like AVL, guarantee shallowness

- Each node is colored *red* or *black*
- Coloring satisfies these rules
  - All empty trees are black
    - We consider them to be the leaves of the tree
  - Children of red nodes are black
  - All paths from a given node to it's descendent leaves have the *same number* of black nodes
    - This is called the *black height* of the node

A Red-Black Tree

(from Wikipedia.org)



Red-Black Trees

The coloring rules lead to the following result

Proposition: No leaf has depth more than twice that of any other leaf.

This in turn can be used to show

Theorem: A Red-Black tree with n internal nodes has height satisfying $h \leq 2\log(n+1)$

- Note: The tree will have *exactly* n+1 (empty) leaves
  - since each internal node has two children

Red-Black Trees

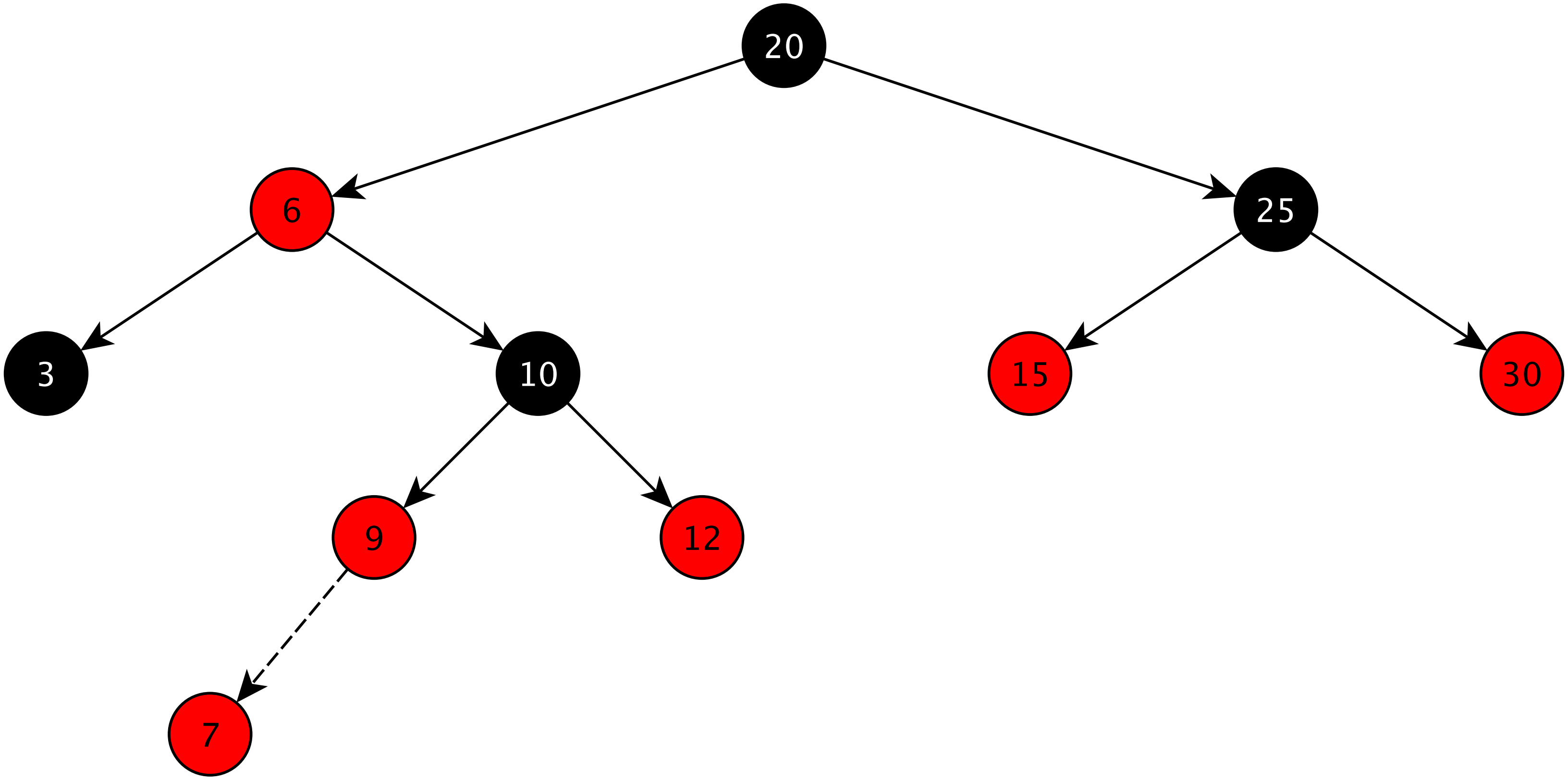
Theorem: A Red-Black tree with n *internal* nodes has height satisfying $h \leq 2\log(n+1)$

Proof sketch: Note: we count empty tree nodes!

- If root is red, recolor it black.
- Now merge red children into (black) parents
  - Now n’ **≤ n nodes and height h’ ≥ h/2**
- New tree has all children with degree 2, 3, or 4
  - All leaves have depth *exactly* h’ and there are n+1 leaves
    - So $n+1 \geq 2^{h'}$, so $\log_2(n+1) \geq h' \geq \frac{h}{2}$
- Thus $2\log_2(n+1) \geq h$

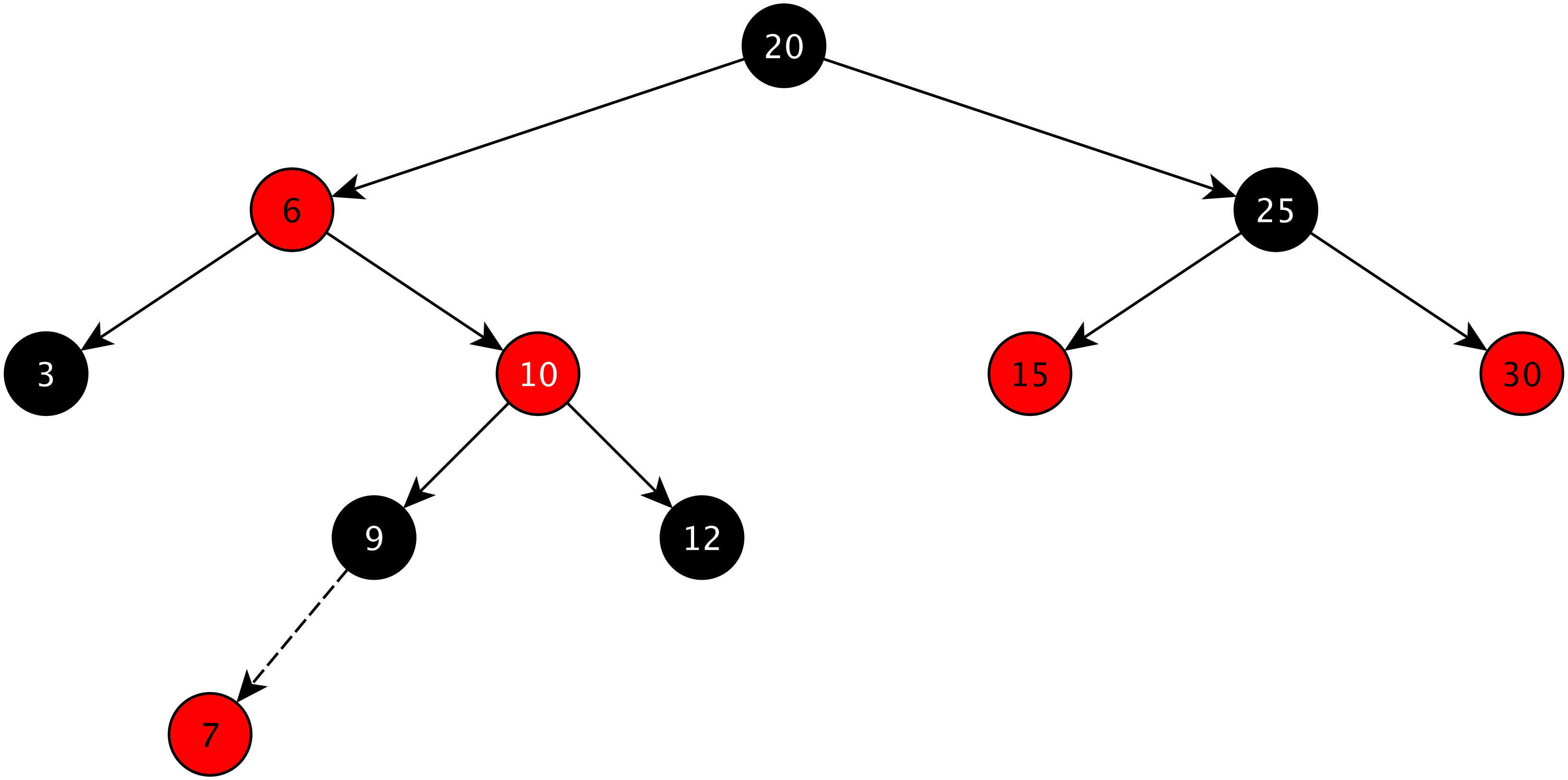
Corollary: R-B trees with n nodes have height O(log n)

Red-Black Tree Insertion



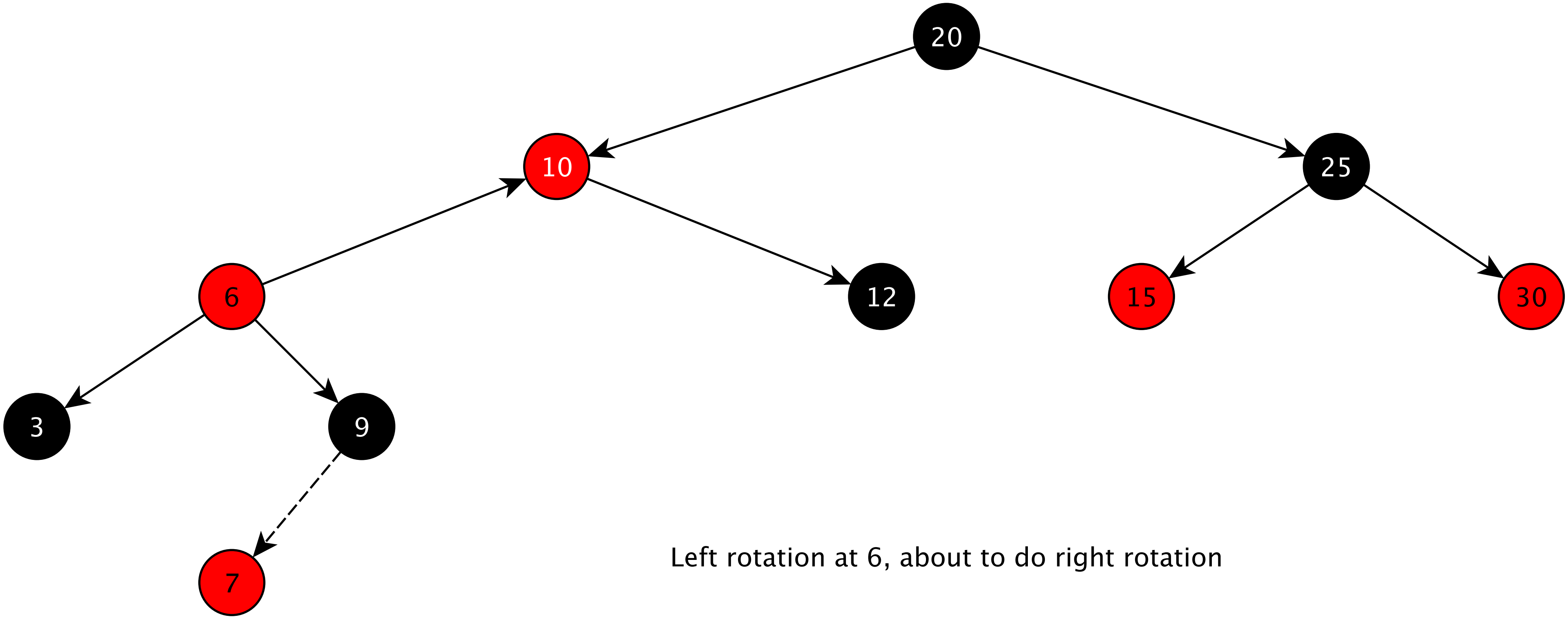
Black empty leaves not drawn. 7 just added Black-height still 2.

Red-Black Tree Insertion



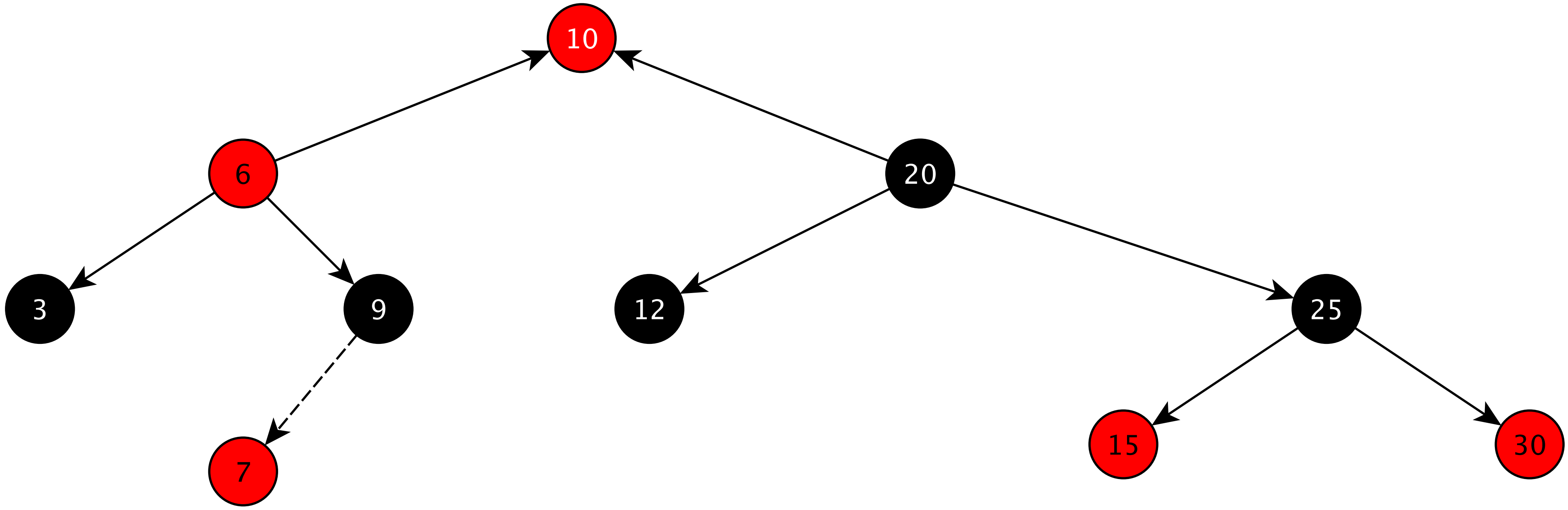
Black height still 2, color violation moved up

Red-Black Tree Insertion



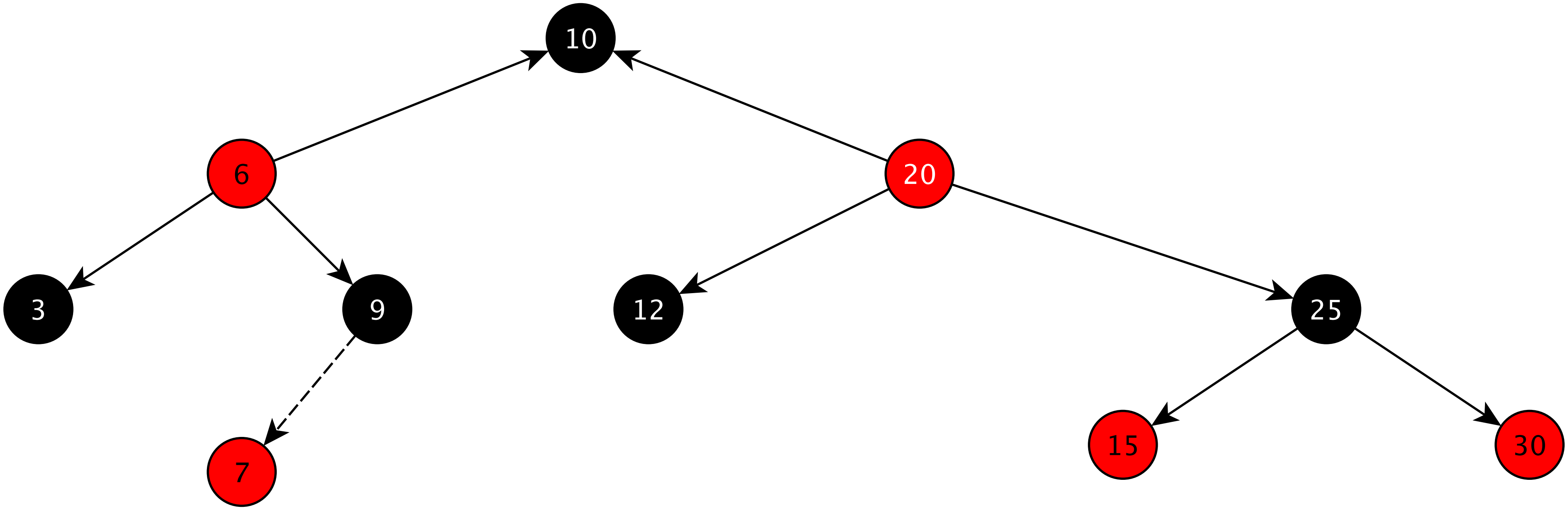
Left rotation at 6, about to do right rotation

Red-Black Tree Insertion



Right rotation at 20, black height broken, need to recolor

Red-Black Tree Insertion



Color conditions restored, black-height restored.