CSCI 136
Data Structures & Advanced Programming

Balanced Binary Search Trees

Balanced Binary Search Trees

Outline

- Tree balancing to maintain small height
  - AVL Trees
  - Red-Black Trees
  - Splay Trees

Binary Search Tree Summary

Binary search trees store comparable values and support

- add(E value)
- contains(E value)
- get(E value)
- remove(E value)

All of which run in O(h) time (h = tree height)

Can also support

- predecessor/successor methods
- Range query: Find all value V : A ≤ V ≤ B

Controlling Tree Height

- Can we design a binary search tree that is always "shallow"?
- Yes! In many ways. We'll Explore a few
- First Up : 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

One of the first balanced binary search tree structures

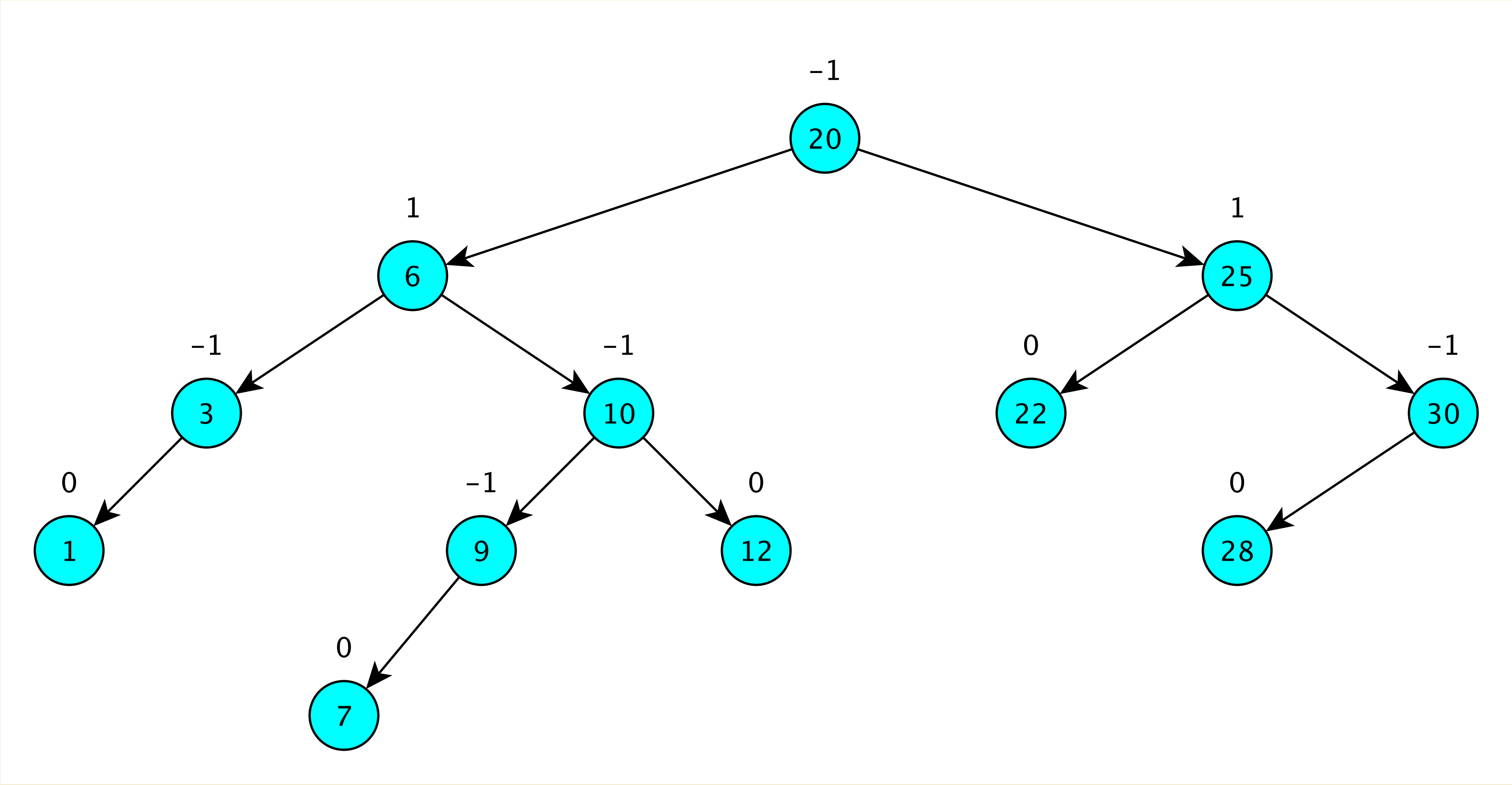
Definition: A binary search tree T is an AVL tree if

1. T is the empty tree, or
2. T has left and right sub-trees T_L and T_R such that
   a) The heights of T_L and T_R differ by at most 1, and
   b) T_L and T_R are AVL trees

Note

Recursive definition implies that height difference of at most 1 must hold at *every node!*

AVL Trees



Balance factor: Height of right subtree minus height of left subtree

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.

Alternate Definition: An AVL Tree is a binary tree in which every node is balanced.

- Tree stores balance factor at each node
- Updates balance factors during add/remove

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: $fib(h) \geq (^3/_2)^h$ if h ≥ 10
  - Also provable by induction!
- So $n \geq (^3/_2)^h$ and thus $\log_{3/2} n \geq h$ (for $h \geq 10$)
  - $\log_{3/2} n \geq h$ --- what's $\log_{3/2} n$???

AVL Trees have O(log n) Height

Recall the change of base rule for logs

$$\text{For any } a, b > 0,\ \log_a n = \frac{\log_b n}{\log_b a}$$

Therefore

$$\log_{3/2} n = \frac{\log_2 n}{\log_2(3/2)} = \frac{1}{\log_2(3/2)} \cdot \log_2 n = c \cdot \log_2 n$$

And so

$$\mathrm{h} \le \log_{3/2} n = c \cdot \log_2 n$$

So h is O(log n) as desired

We used Fibonacci numbers in a data structures proof

How Cool Is That?!

AVL Trees

If adding to an AVL tree creates an unbalanced node A, we rebalance the subtree with root A

This involves a constant-time restructuring of part of the tree (this is a claim!)

The rebalancing steps are called *tree rotations*

Tree rotations preserve binary search tree structure

AVL Trees

Suppose adding to an AVL tree creates some unbalanced nodes

- The only nodes that can become unbalanced are the ancestors of the newly added node
  - So they are all on the path from the new node back to the root
- Their balance factors can change by at most 1
- So there may be some nodes that now have balance factors ±2
- Let's consider the deepest such node, call it A
- All subtrees of A are AVL trees—balance factors 0 or ±1

AVL Trees

There are four cases for the location of the new node with respect to A

- It's in the *left* subtree of the *left* child of A
- It's in the *right* subtree of the *left* child of A
- It's in the *left* subtree of the *right* child of A
- It's in the *right* subtree of the *right* child of A

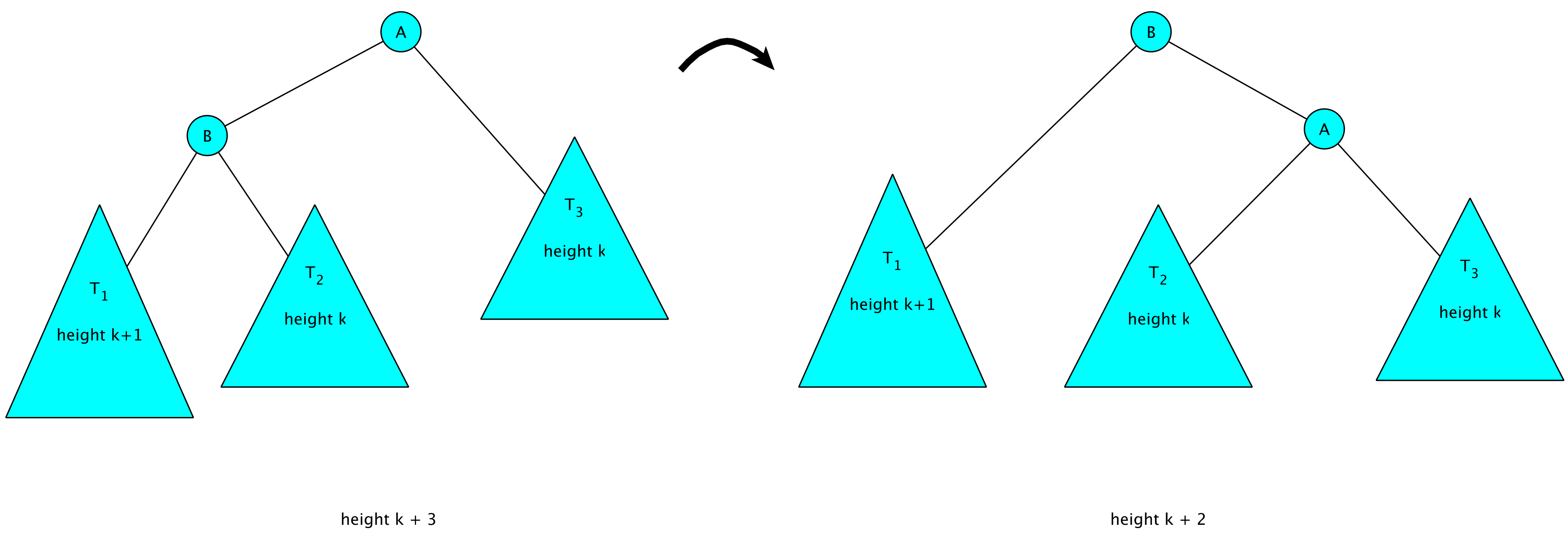
Let's consider the first two cases

- The other two are handled similarly

Single Right Rotation

Assume A is unbalanced but its subtrees are AVL…

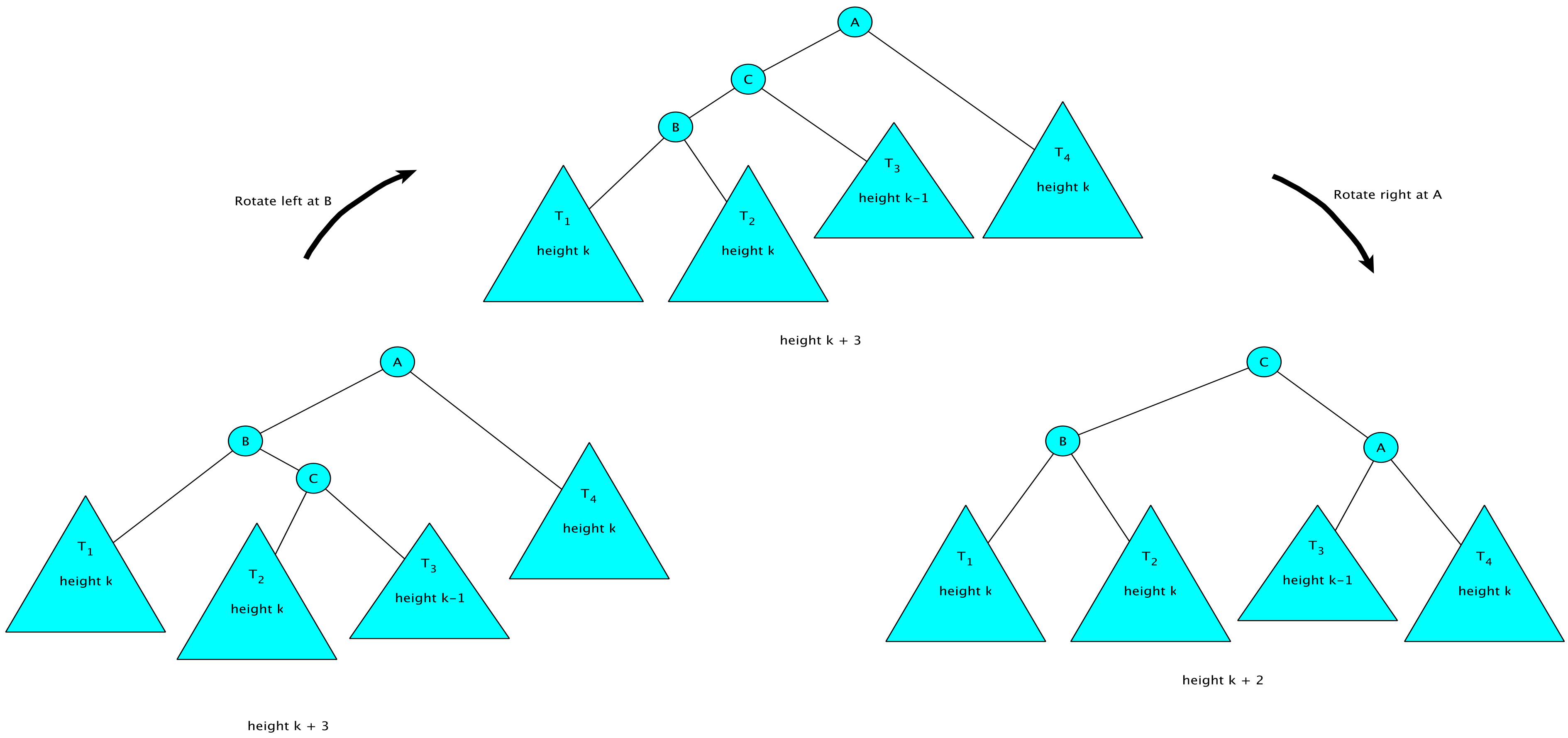
And that the new node is in the left subtree of B



Note: Heights *must be* as labeled in figure!

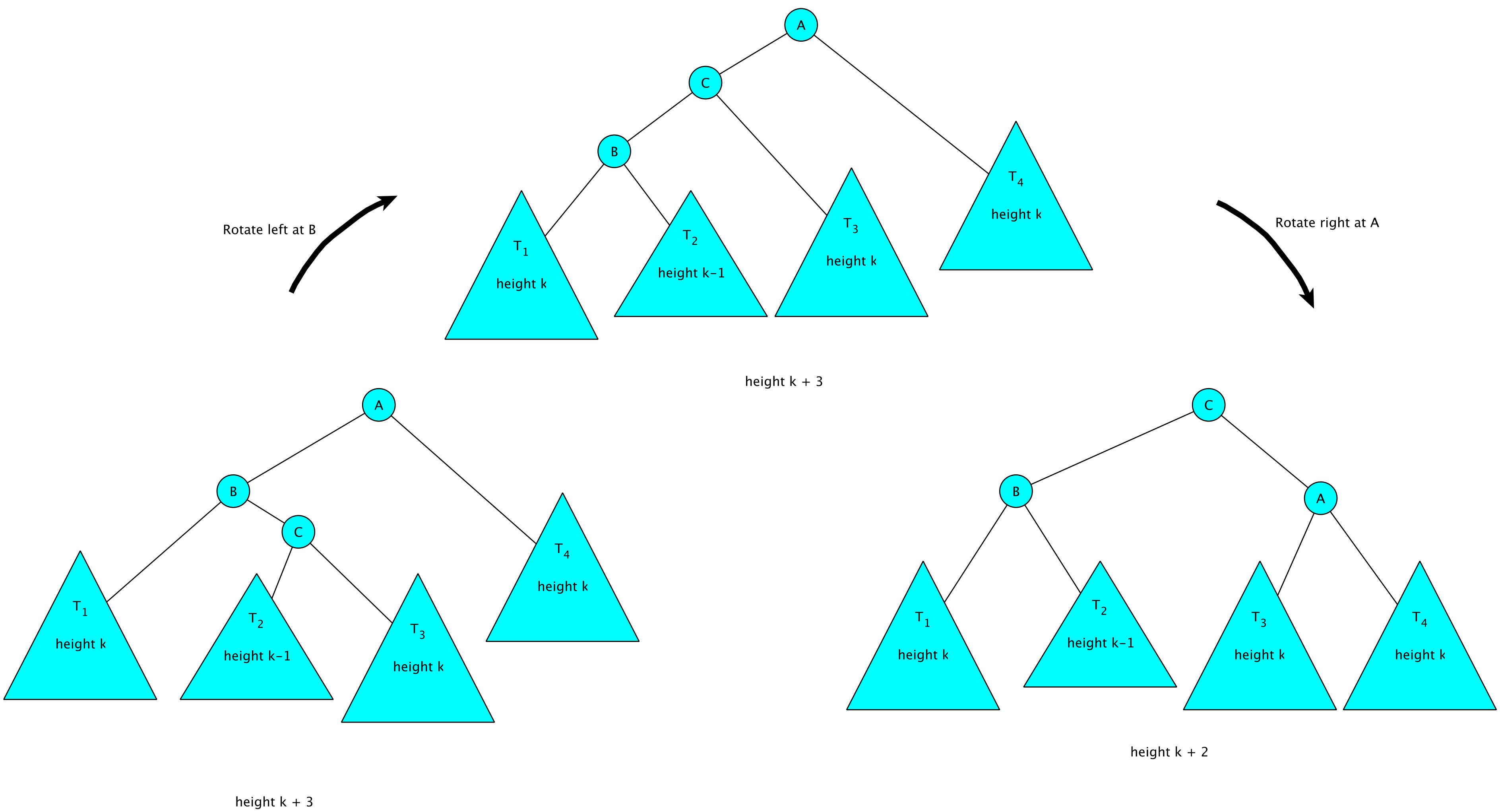
Double Rotation I

Now assume the new node is in the right subtree of B



Note: T_2 and T_3 might be switched!

Double Rotation II



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)

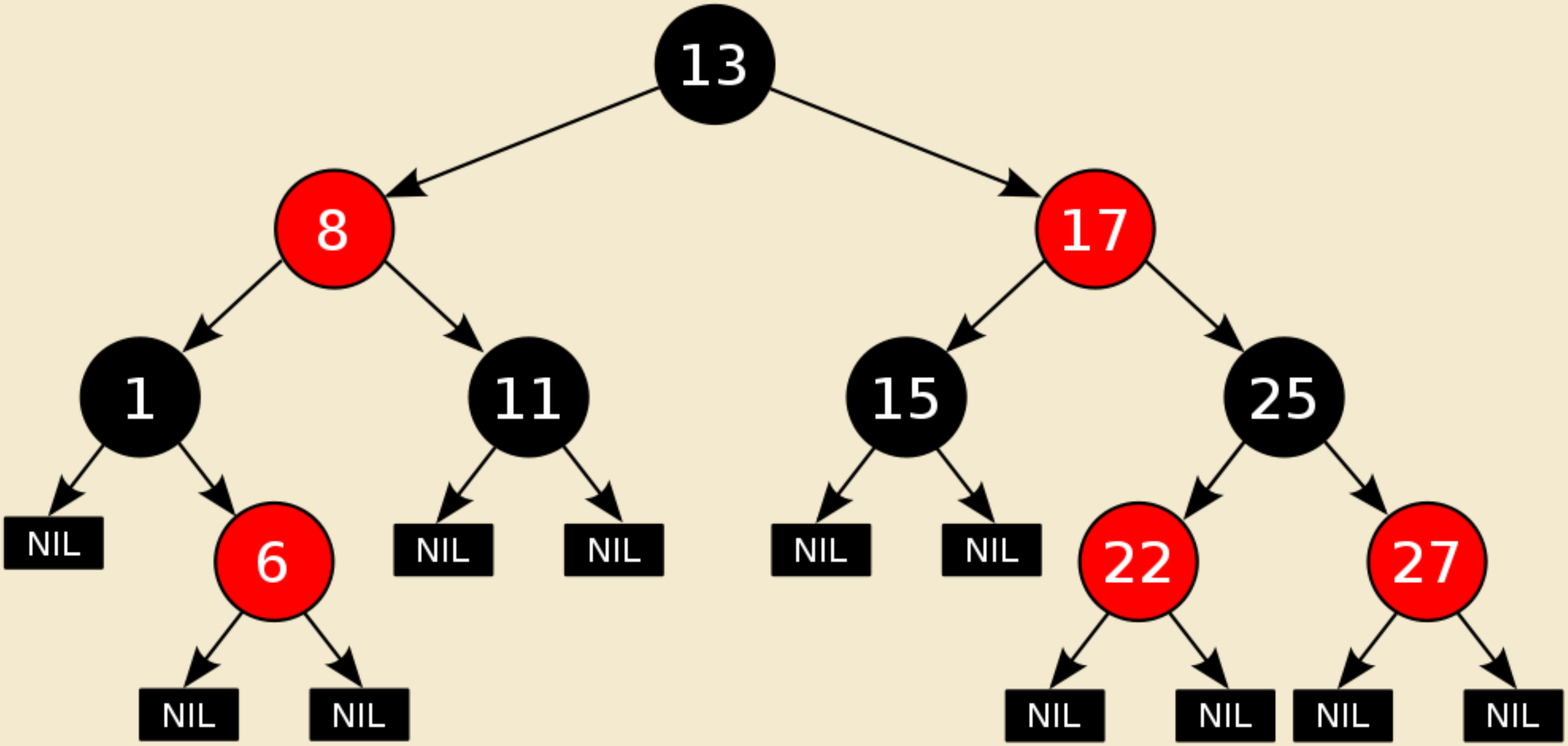
AVL Trees: One of Many

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)



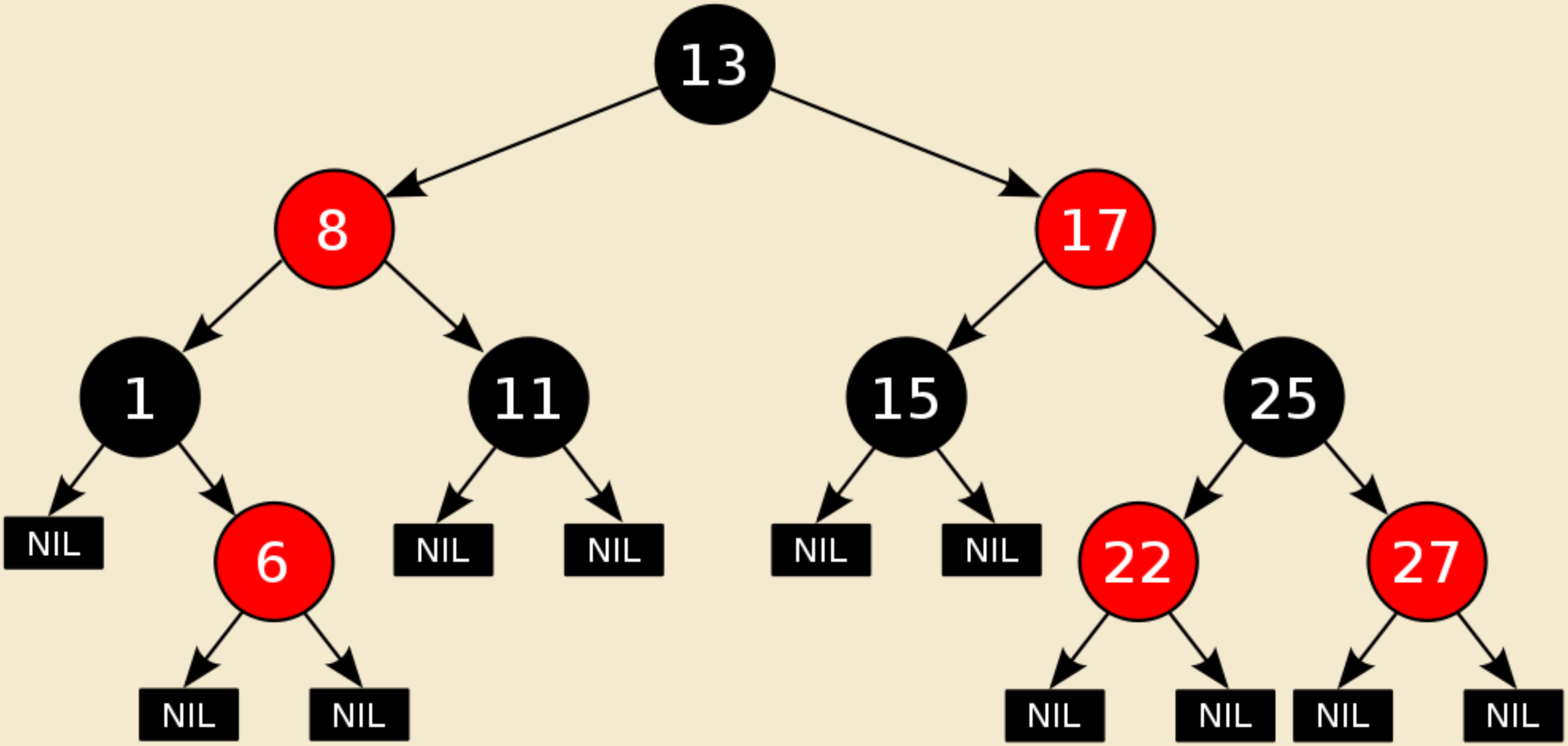
Red-Black Trees

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 empty nodes to be the leaves of the tree
  - Children of red nodes are black
  - All paths from a given node to its 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

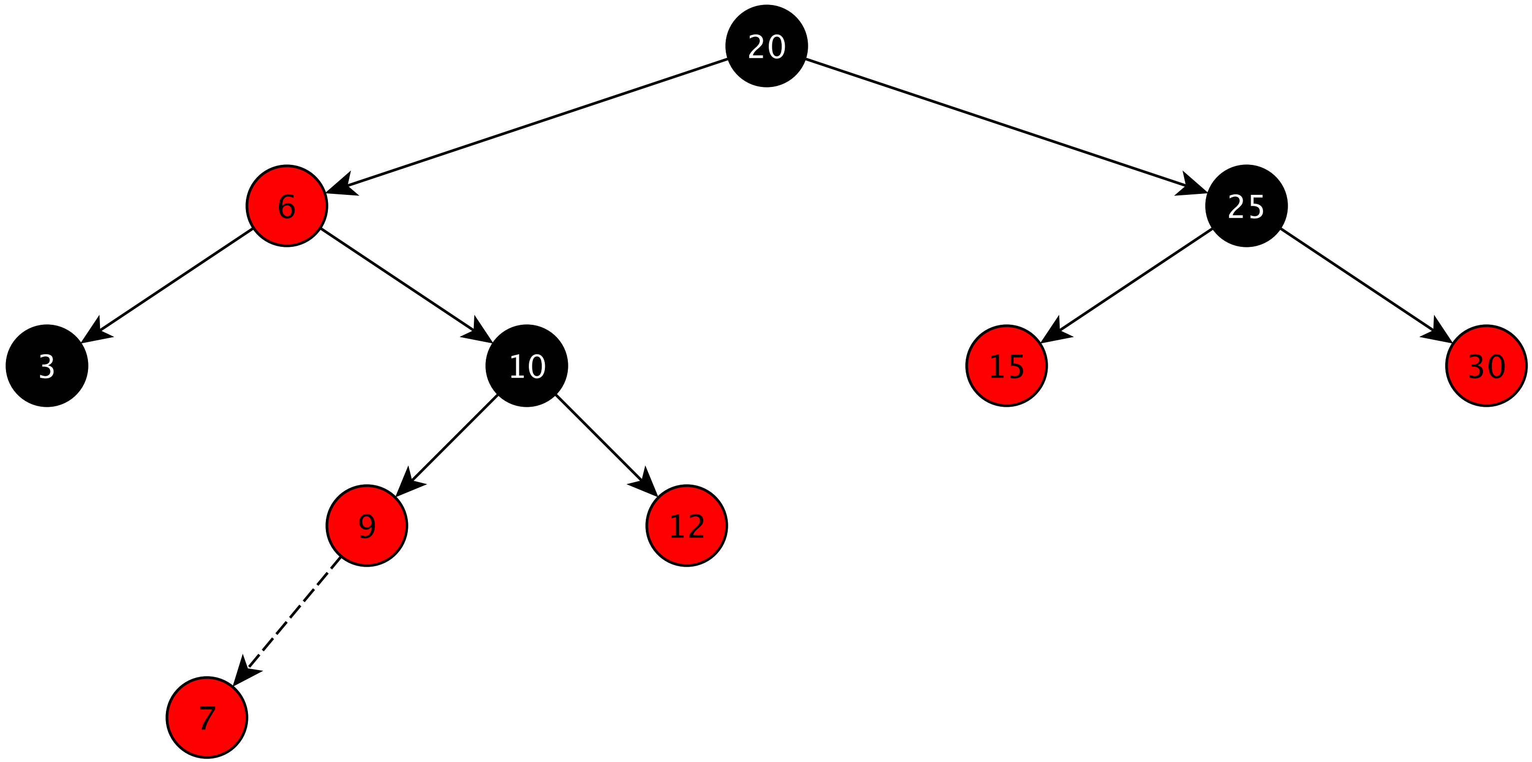
Theorem: A Red-Black tree with n *internal* nodes has height satisfying $h \leq 2\log(n+1)$

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: Each internal node has 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 $\log_2(n+1) \geq h$

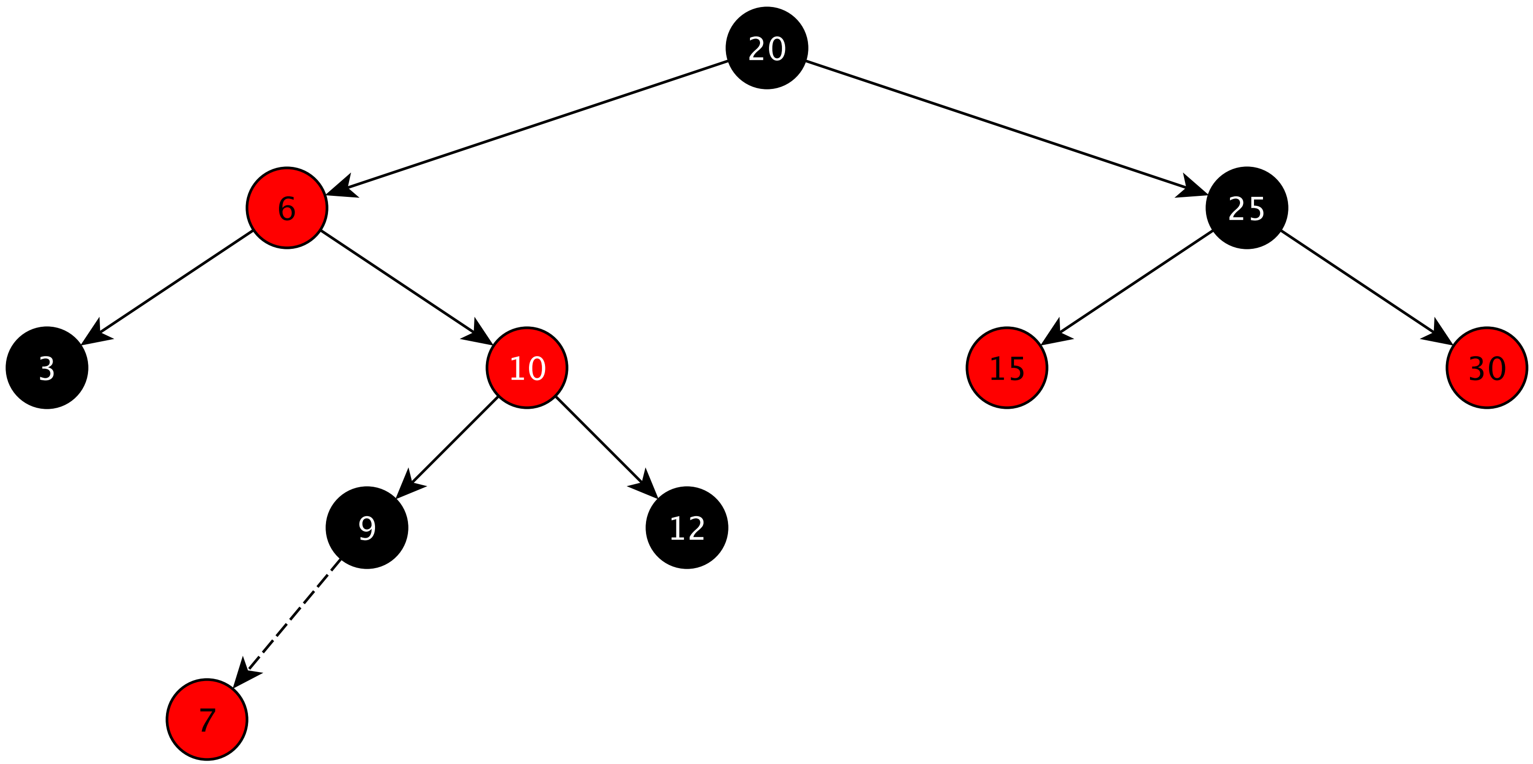
Corollary: R-B trees with n nodes have height O(log n)

Red-Black Tree Insertion



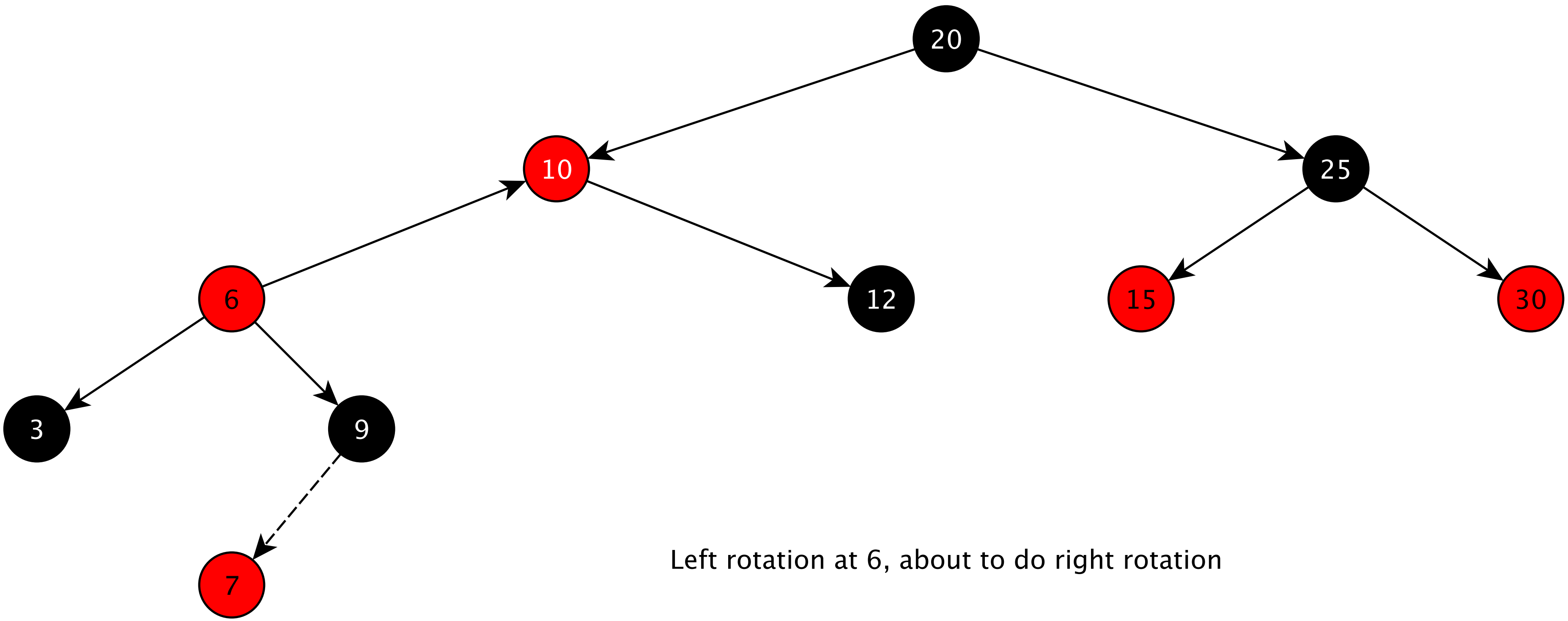
Black empty leaves not drawn. 7 just added Black-height still 2.

Red-Black Tree Insertion



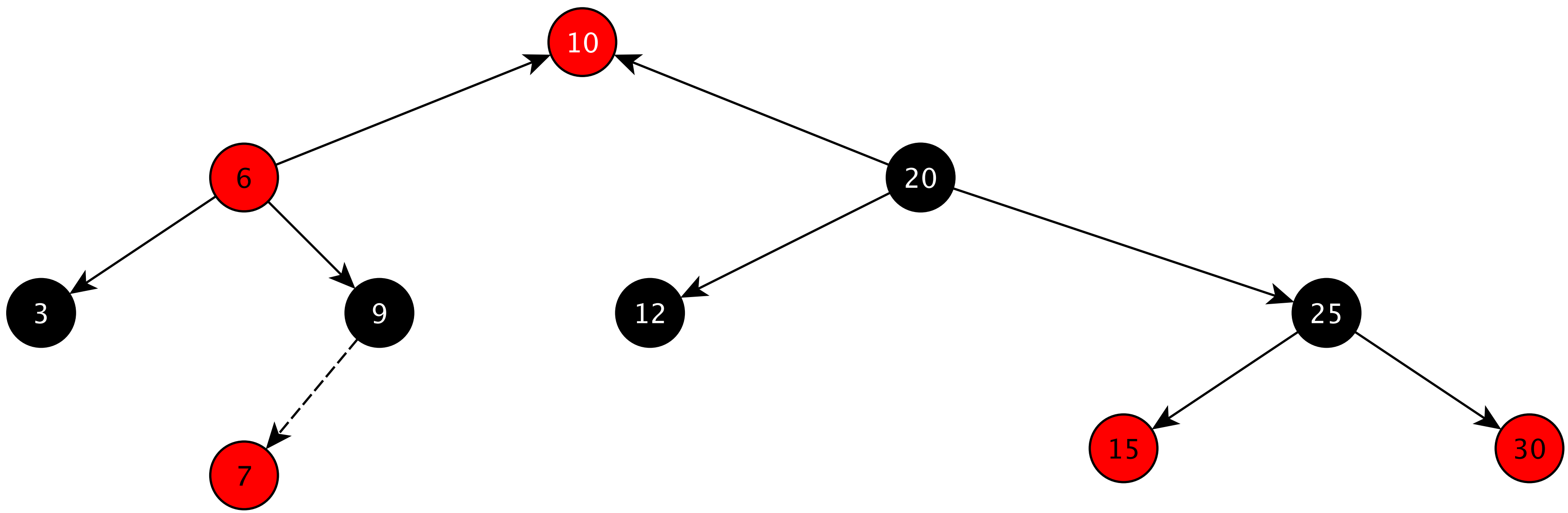
Black height still 2, color violation moved up

Red-Black Tree Insertion



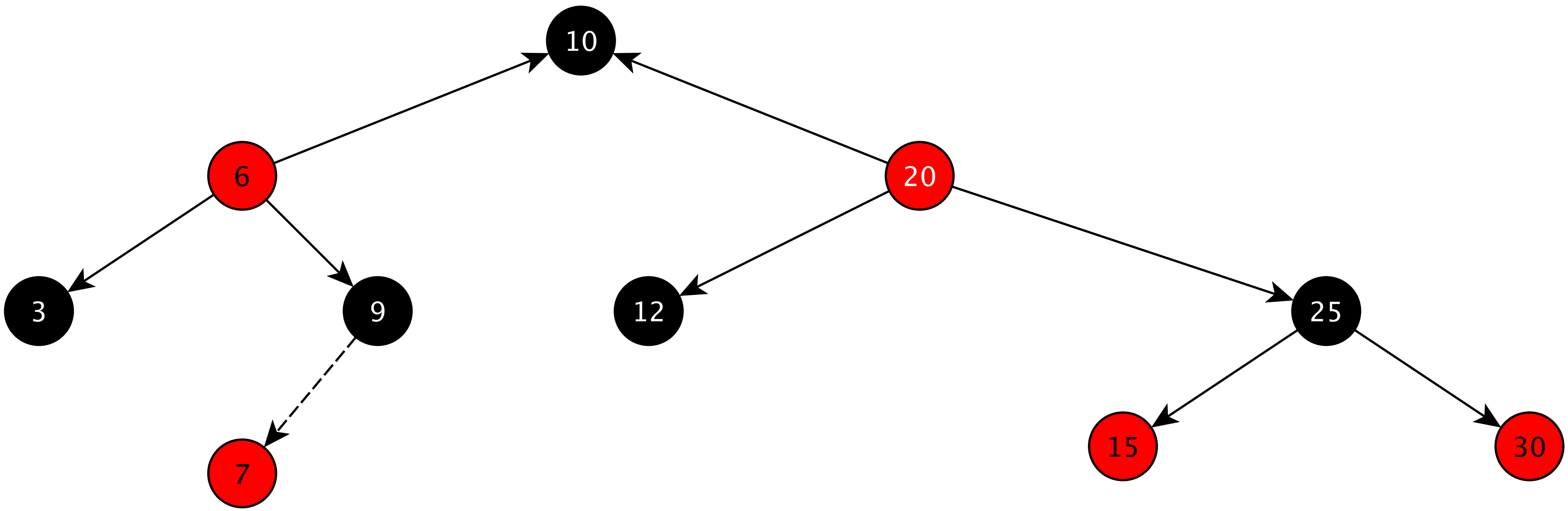
Left rotation at 6, about to do right rotation

Red-Black Tree Insertion



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

Red-Black Tree Insertion



Color conditions restored, black-height restored.

Splay Trees

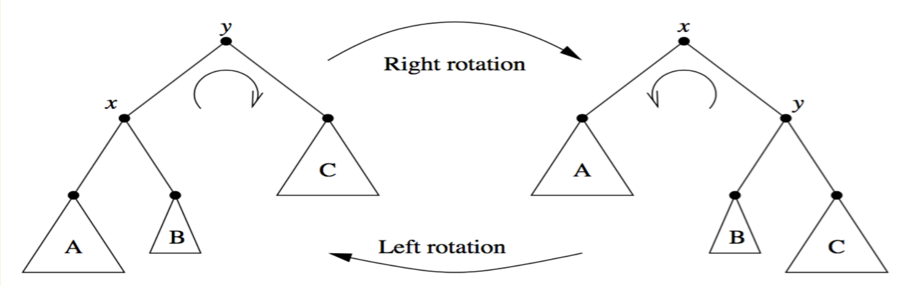
Splay trees are self-adjusting binary trees

- Each time a node is accessed, it is moved to root position via rotations
- No guarantee of balance (or shallow height)
- But good *amortized* performance

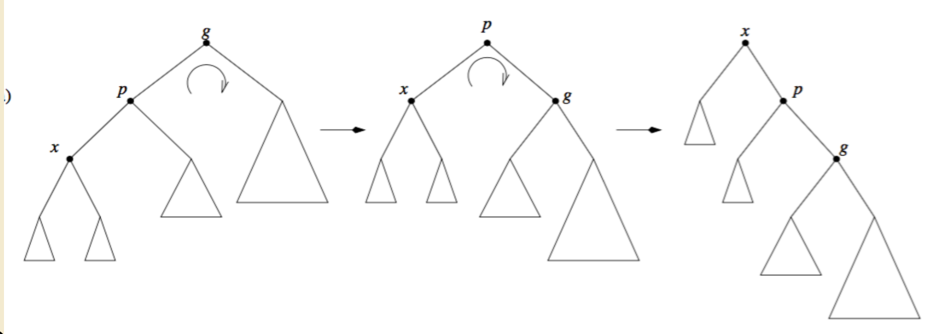
Theorem: Any set of m operations (add, remove, contains, get) on an n-node splay tree take at most O(m log n) time.

Splay Tree Rotations

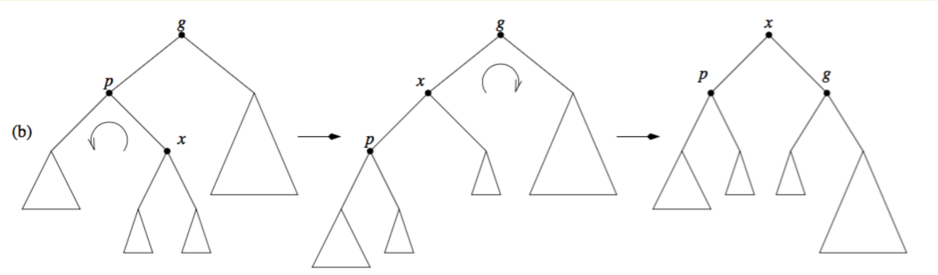
Right Zig Rotation (left version too)



Right Zig-Zig Rotation (left version too)



Right Zig-Zag Rotation (left version too)



Splay Tree Iterator

(aka Unintended Consequence)

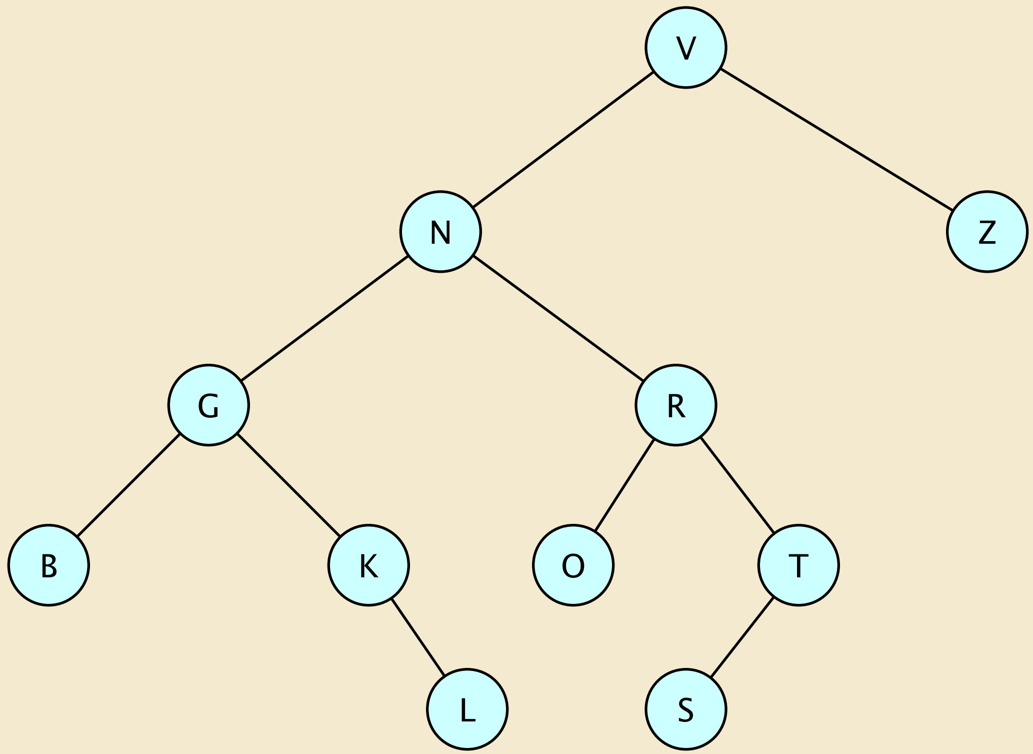
- Recall: The iterator for an in-order traversal of a Binary(Search)Tree employed a stack that contained the path from the root of the tree to the next node to be served by the iterator
- For each of our balanced binary tree implementations
  - Add and remove methods change the shape of the tree
  - This means they break any iterators that are currently traversing the tree.
- This reinforces our dictum

  Don't change a structure while iterating over it
- However, splay trees introduce a new wrinkle….

Splay Tree Iterator

- Even *contains* method changes splay tree shape!
- Solution: Remove the stack from the iterator
- Observation: Given location of current node (node whose value is next to be returned), we can compute it's (in-order)successor in *next()*: It is either
  - The left-most leaf of the right child of current, or
  - The closest "left-ancestor" of current
    - Ancestor whose left child is also an ancestor of current

Finding the Next Node



If current = N, next = O : left-most leaf of right child of N
If current = L, next = N : closest left-ancestor of L

Splay Tree Iterator

But, for reset to work, we also need be able to find the root of the tree!

- Idea: Hold a single “reference” node
  - Any node of the tree will do
  - To reset the iterator
    - Walk up the tree from the reference node to the root
    - Use the root to reset the iterator
- The splay tree iterator can now survive tree reshaping!
  - Although behavior after add/remove is still unpredictable

Summary & Observations

Many variants of the binary search tree structure exist

- They take different approaches to improving the effectiveness of the structure
  - AVL, RedBlack, and other variants ensure that the tree height is O(log n)
  - Splay Trees provide O(log n) *amortized* performance per operation
  - Randomization (not discussed here) can be used to guarantee O(log n) *average* performance
  - This presentation has just scratched the surface