CSCI 136
Data Structures & Advanced Programming

Lecture 25

Fall 2019

Instructor: Bill & Sam

Administrative Details

- Problem Set 3 due now
  - Hand in at the end of class
  - Late days are an option
- Lab 8 due Sunday (9 out right after)

Today

- Introduction to Binary Search Trees (BSTs)

Improving on OrderedVector

- The OrderedVector class provides O(log n) time searching for a group of n comparable objects
  - add() and remove(), though, take O(n) time in the worst case---and on average!
- Can we improve on those running times without sacrificing the O(log n) search time?
- Let’s find out....

Binary Trees and Orders

- Binary trees impose multiple orderings on their elements (pre-/in-/post-/level-orders)
- In particular, in-order traversal suggests a natural way to hold comparable items
  - For each node v in tree
    - All values in left subtree of v are at most v
    - All values in right subtree of v are at least v
- This leads us to...

Binary Search Trees

- Binary search trees maintain a *total* ordering among elements (assumes comparability)
- Definition: A BST T is either:
  - Empty
  - Has root r with subtrees T_L and T_R such that
    - All nodes in T_L have smaller value than r
    - All nodes in T_R have larger value than r
    - T_L and T_R are also BSTs

BST Observations

- The same data can be represented by many BST shapes
- Searching for a value in a BST takes time proportional to the height of the tree
  - Reminder: trees have height, nodes have depth
- Additions to a BST happen at nodes missing at least one child (*a constraint!*)
- Removing from a BST can involve *any* node

BST Operations

- BSTs will implement the OrderedStructure Interface
  - `add(E item)`
  - `contains(E item)`
  - `get(E item)`
  - `remove(E item)`
  - Runtime of above operations?
    - All O(h) – where h is the tree height
  - `iterator()`
    - `This will provide an in-order traversal`

BST Implementation

- The BST holds the following items
  - BinaryTree root: the root of the tree
  - BinaryTree EMPTY: a static empty BinaryTree
    - To use for all empty nodes of tree
  - int count: the number of nodes in the BST
  - Comparator<E> ordering: for comparing nodes
    - Note: E must implement Comparable
- Two constructors: One takes a Comparator
  - Other creates a NaturalComparatot

BST Implementation: locate

- Several methods search the tree
  - add, remove, contains
- We factor out common code: locate method
- *protected* locate(BinaryTree<E> *node*, E *v*)
  - Returns a BinaryTree<E> in the subtree with root *n* such that either
    - *n* has its value equal to *v*, or
    - *v* is not in this subtree and *n* is the node whose child *v* should be
- How would we implement locate()?

BST Implementation: locate

BinaryTree locate(BinaryTree root, E val)

if root's value equals val return root

child ← child of root whose subtree should

hold val

if child is emptry tree, return root

// val not in subtree based at root

else //keep looking

return locate(child, val)

BST Implementation: locate

- What about this line?

  child ← child of root whose subtree should hold value

- If the tree can have multiple nodes with same value, then we need to be careful
- Convention: During *add* operation, only move to right subtree if value to be added is *greater than* value at node
- We'll look at *add* later
- Let's look at *locate* now....

The code : locate

```
protected BinaryTree<E> locate(BinaryTree<E> root, E value) {
       E rootValue = root.value();
       BinaryTree<E> child;

       // found at root: done
       if (rootValue.equals(value)) return root;

       // look left if less-than, right if greater-than
       if (ordering.compare(rootValue,value) < 0)
           child = root.right();
       else
           child = root.left();

       // no child there: not in tree, return this node,
       // else keep searching
       if (child.isEmpty()) return root;
       else
           return locate(child, value);
}
```

Other core BST methods

- locate(v) returns either a node containing v or a node where v can be added as a child
- locate() is used by
  - `public boolean contains(E value)`
  - `public E get(E value)`
  - `public void add(E value)`
  - `Public void remove(E value)`
- Some of these also use another utility method
  - `protected BT predecessor(BT root)`
- Let's look at contains() first...

Contains

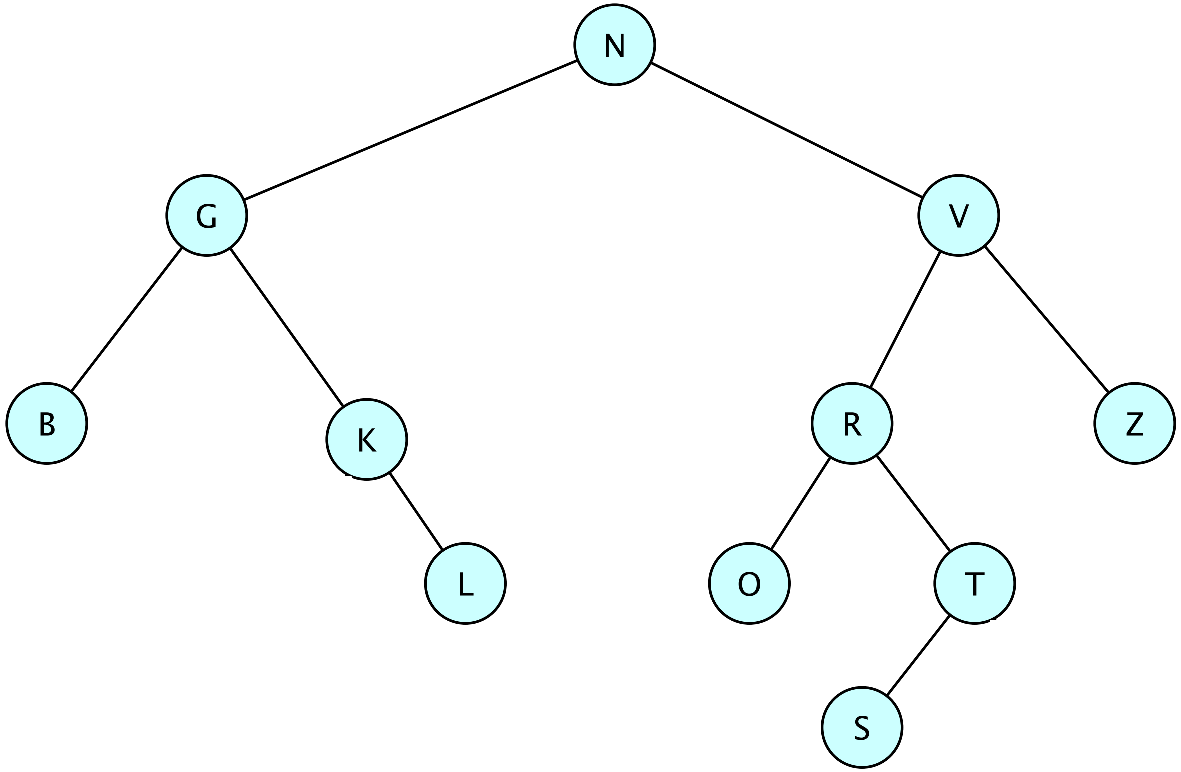
```
public boolean contains(E value){
   if (root.isEmpty()) return false;

   BinaryTree<E> possibleLocation = locate(root,value);

   return value.equals(possibleLocation.value());
}
```

First (Bad) Attempt: add(E value)

```
public void 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
                insertLocation.setLeft(newNode);
        }
        count++;
}
```

Problem: If repeated values are allowed, left subtree might not be empty when setLeft is called

Add: Repeated Nodes



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

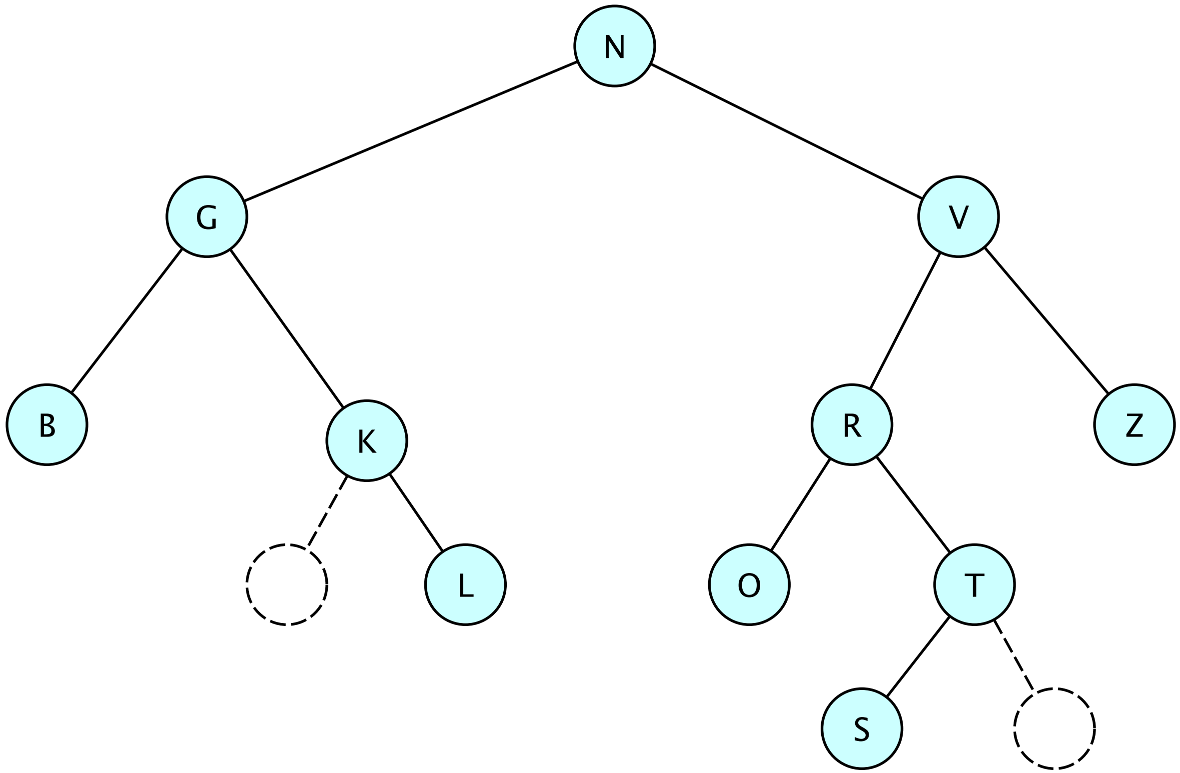
Add Duplicate to Predecessor

- If insertLocation has a left child then
  - Find insertLocation's predecessor
  - Predecessor: item stored immeditately "before" value in true
  - Add repeated node as right child of predecessor
  - If insertLocation has a left subtree that's where Predecessor will be
    - Rightmost item in the left subtree

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?

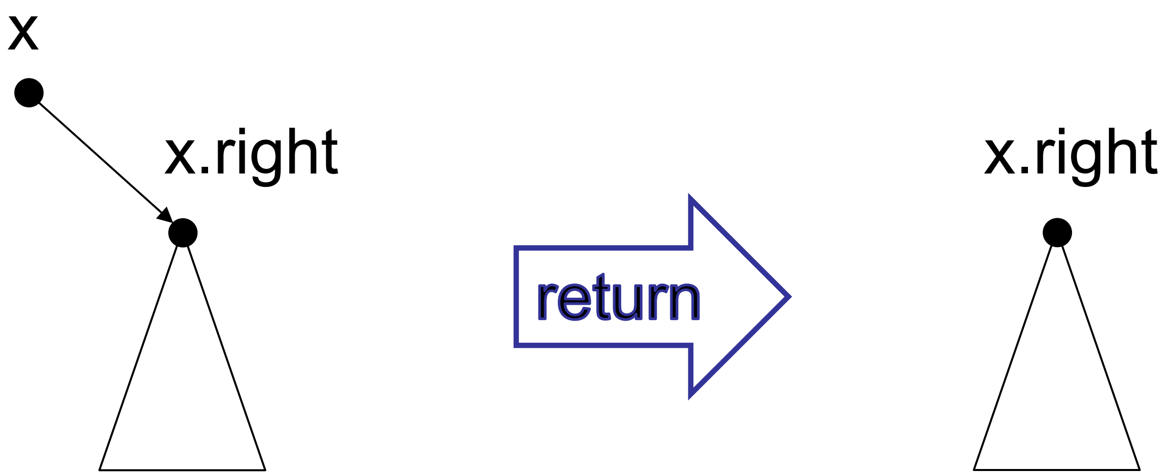
Predecessor

```
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;
}
```

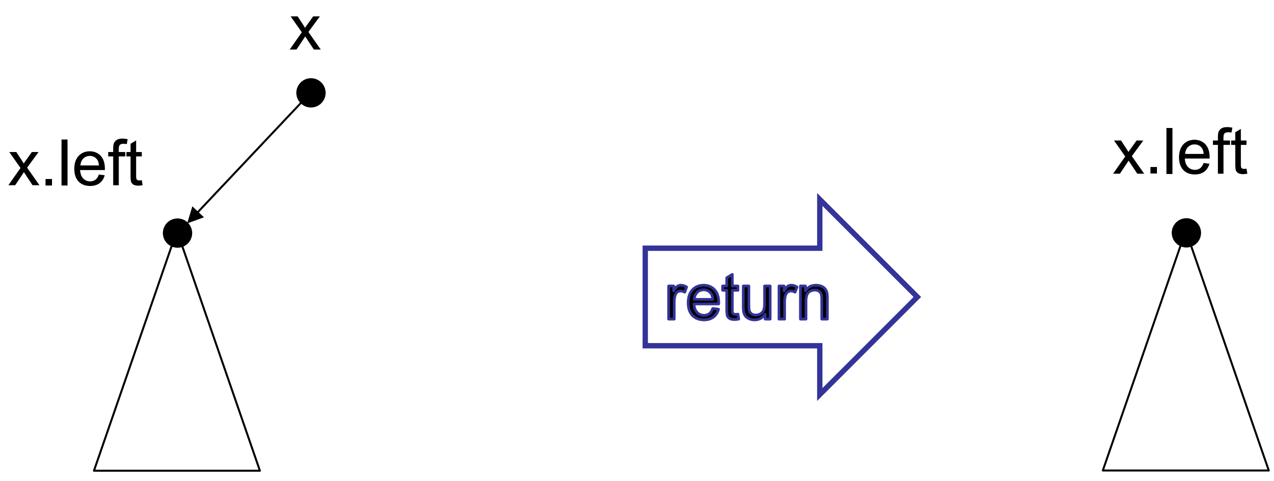
Removal

- 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)`

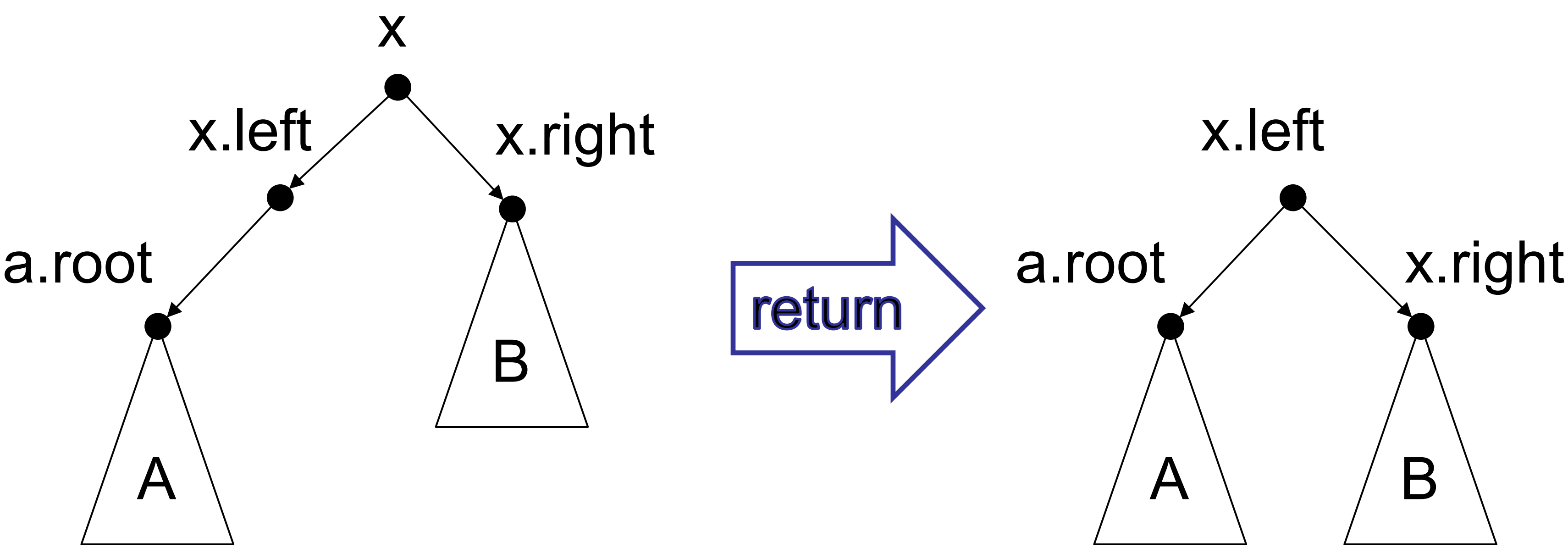
Case 1: No left binary tree



Case 2: No right binary tree



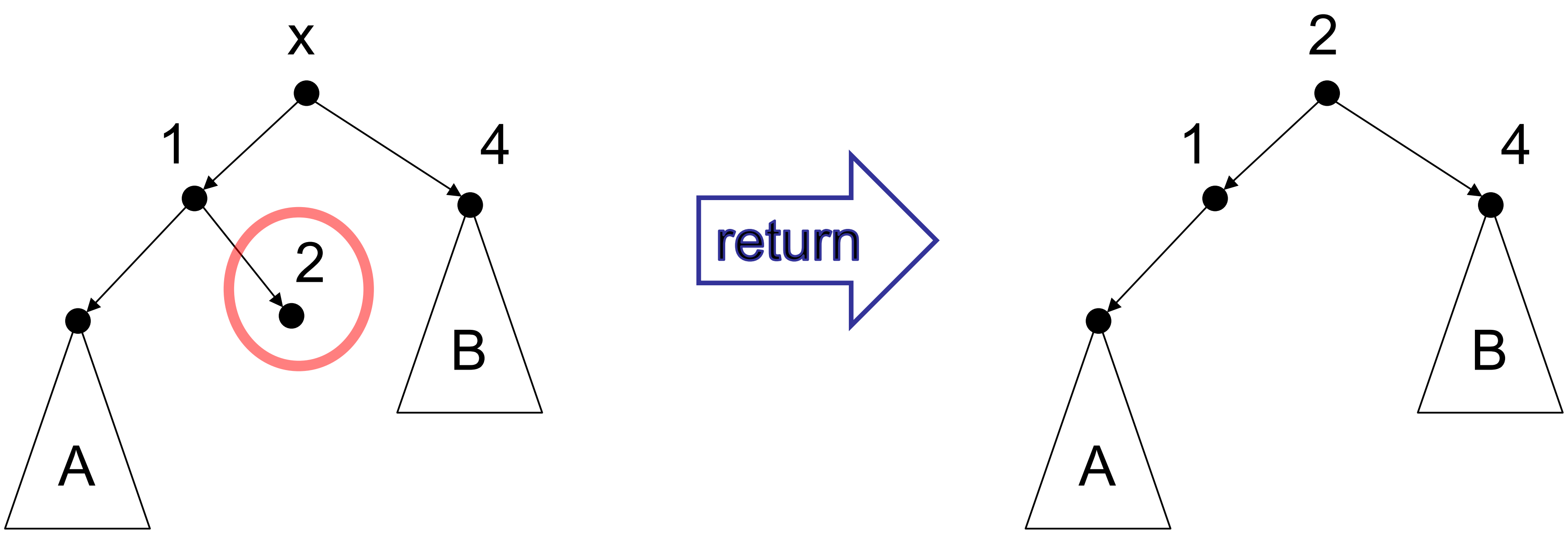
Case 3: Left has no right subtree



Case 4: General Case

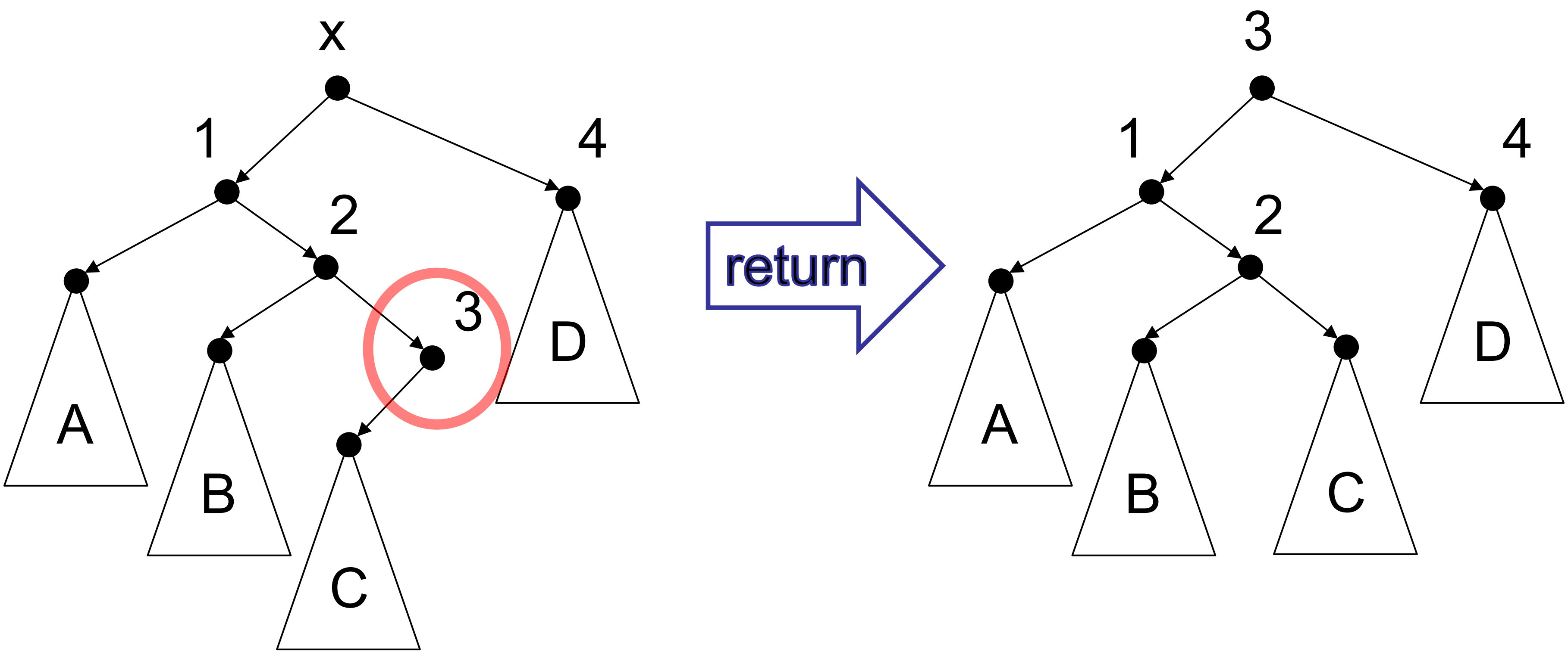
- Consider BST requirements:
  - Left subtree must be <= root
  - Right subtree must be > root
- Strategy: replace the root with the largest value that is less than or equal to it
  - predecessor(root) : rightmost left descendant
- This may require reattaching the predecessor's left subtree!

Case 4: General Case



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

Case 4: General Case



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"

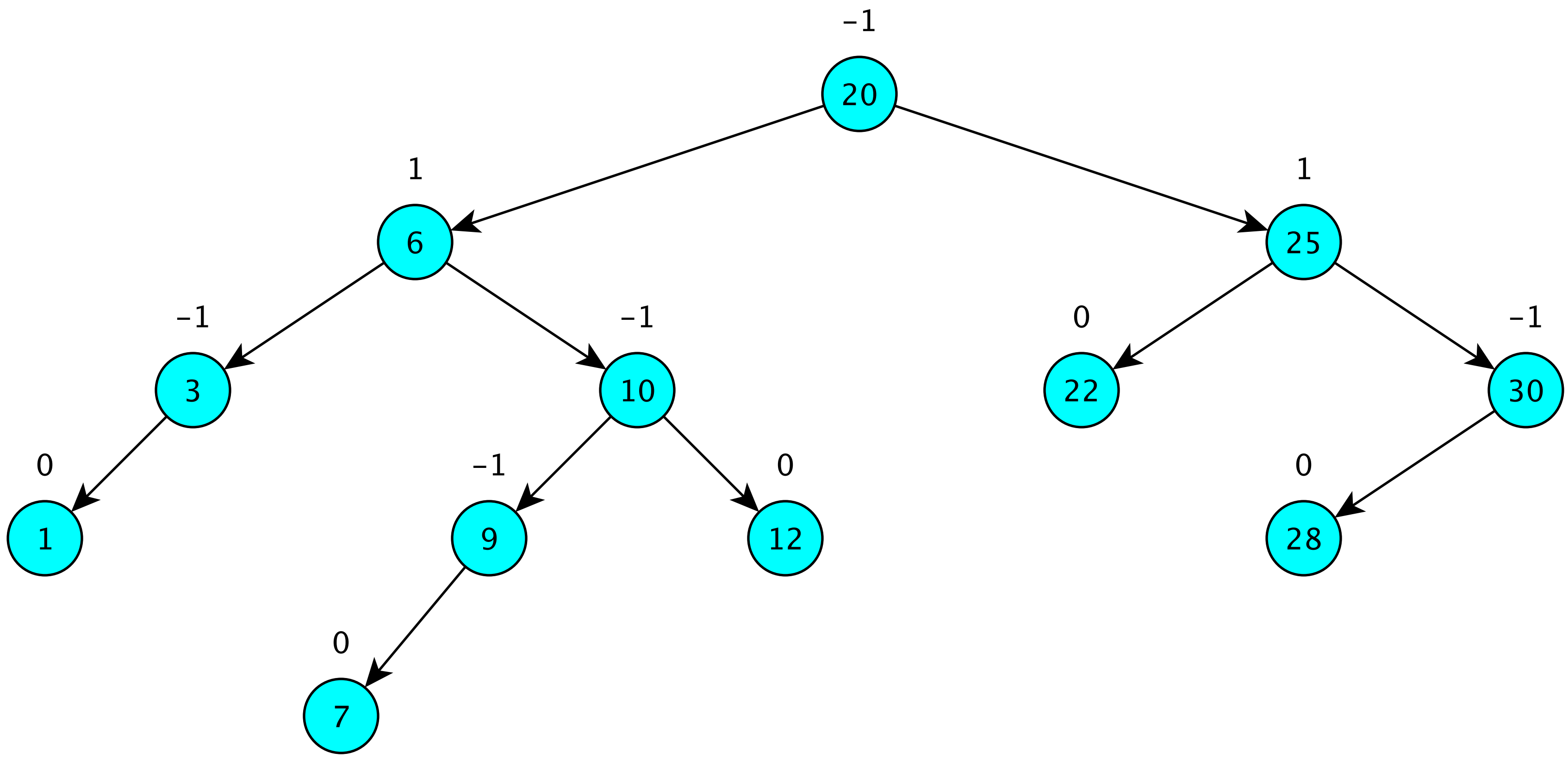
AVL Trees

One of the first balanced binary tree structures

Definition: A binary 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

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.
- Alternate 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 (classic 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)

We used Fibonacci numbers in a data structures proof!!!

AVL Trees

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

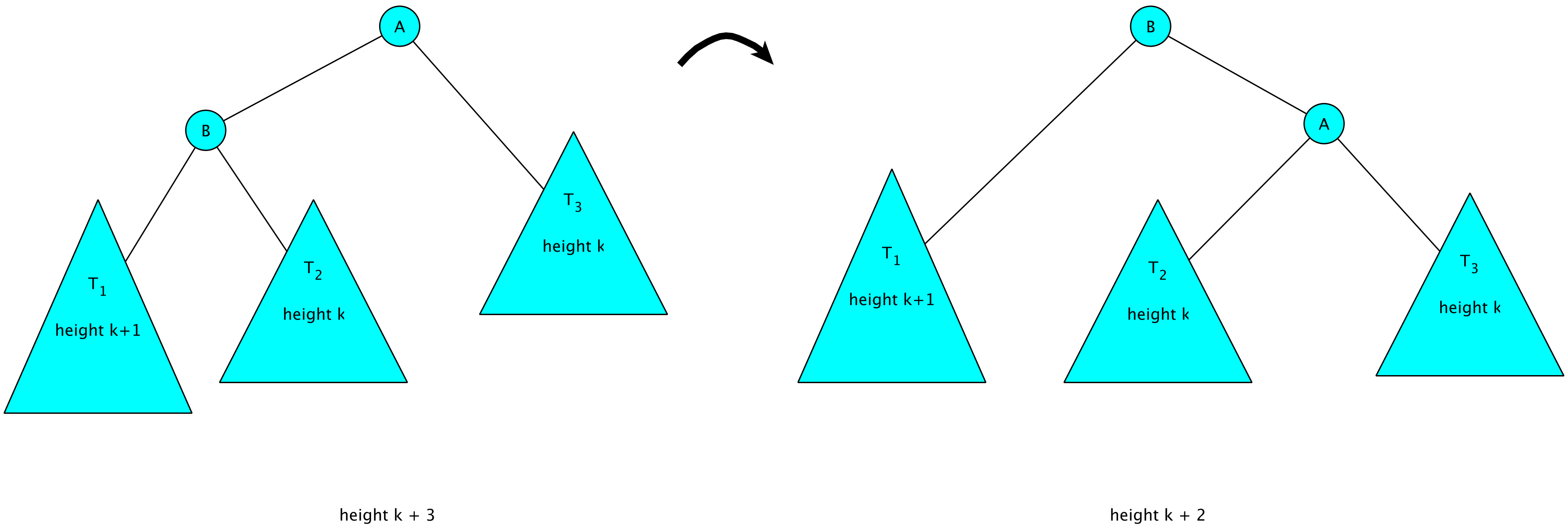
This involves a constant-time restructuring of part of the tree with root NA

The rebalancing steps are called *tree rotations*

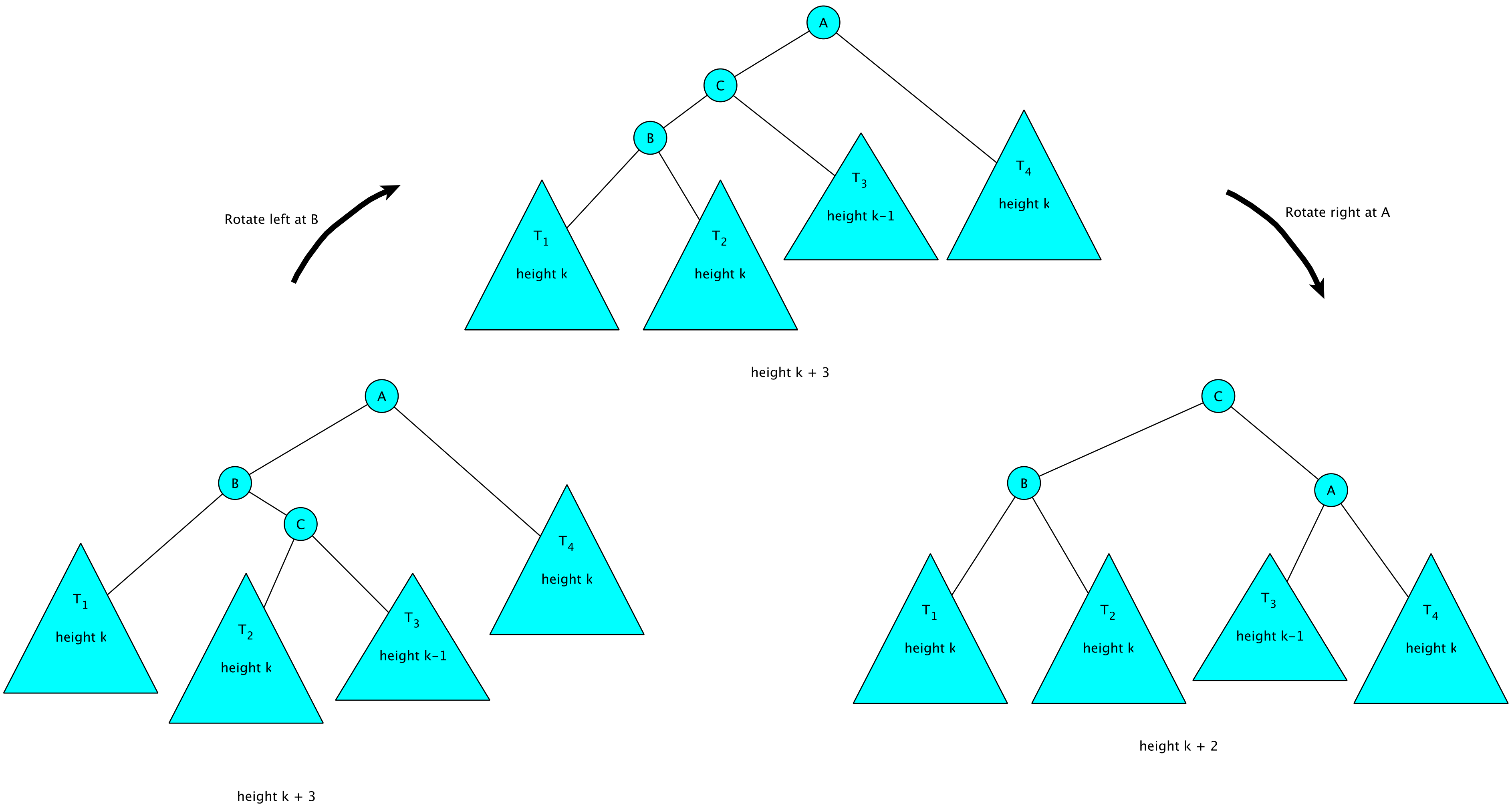
Tree rotations preserve binary search tree structure

Single Right Rotation

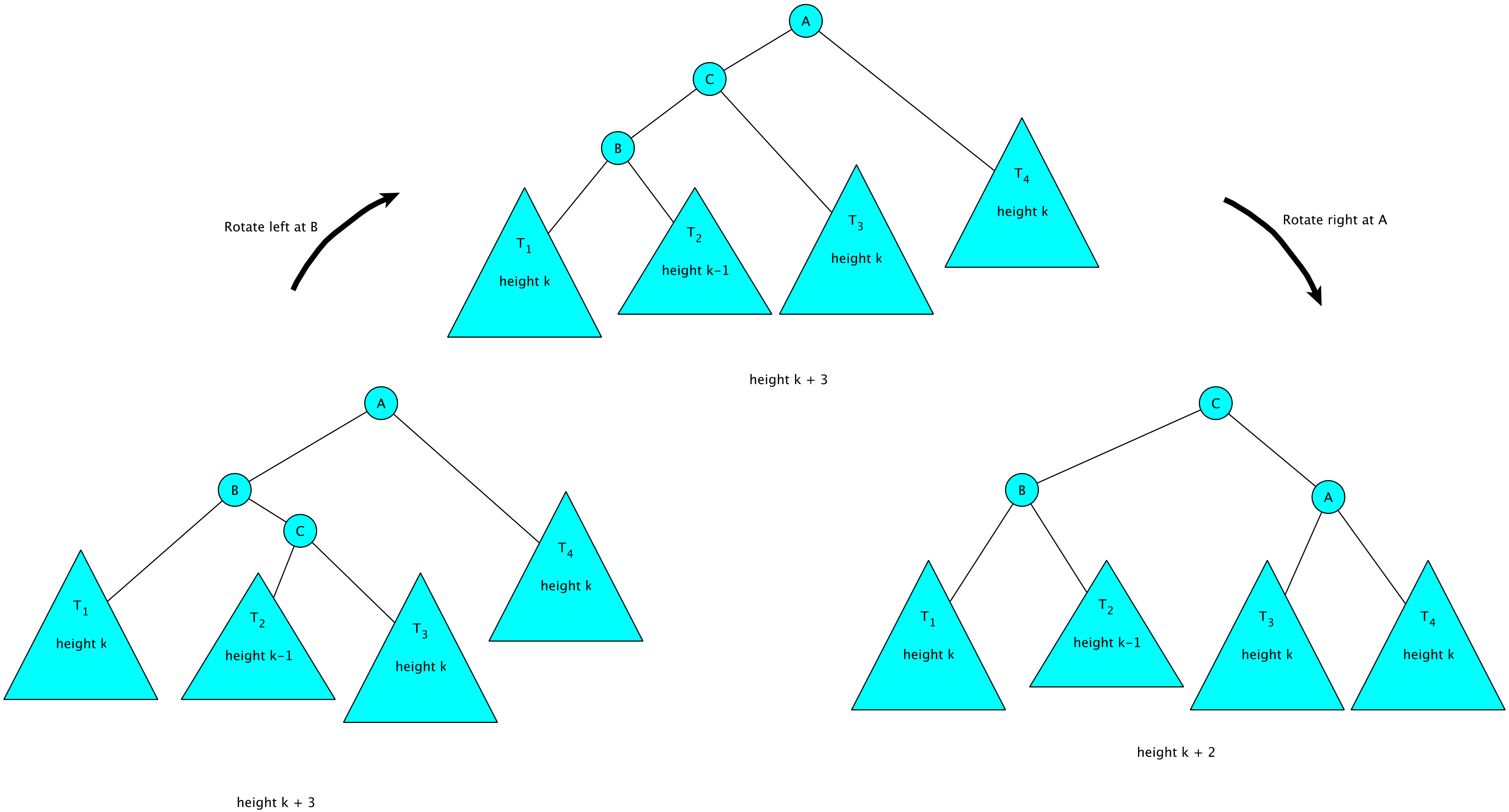
Assume A is unbalanced but its subtrees are AVL…



Double Rotation I



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)

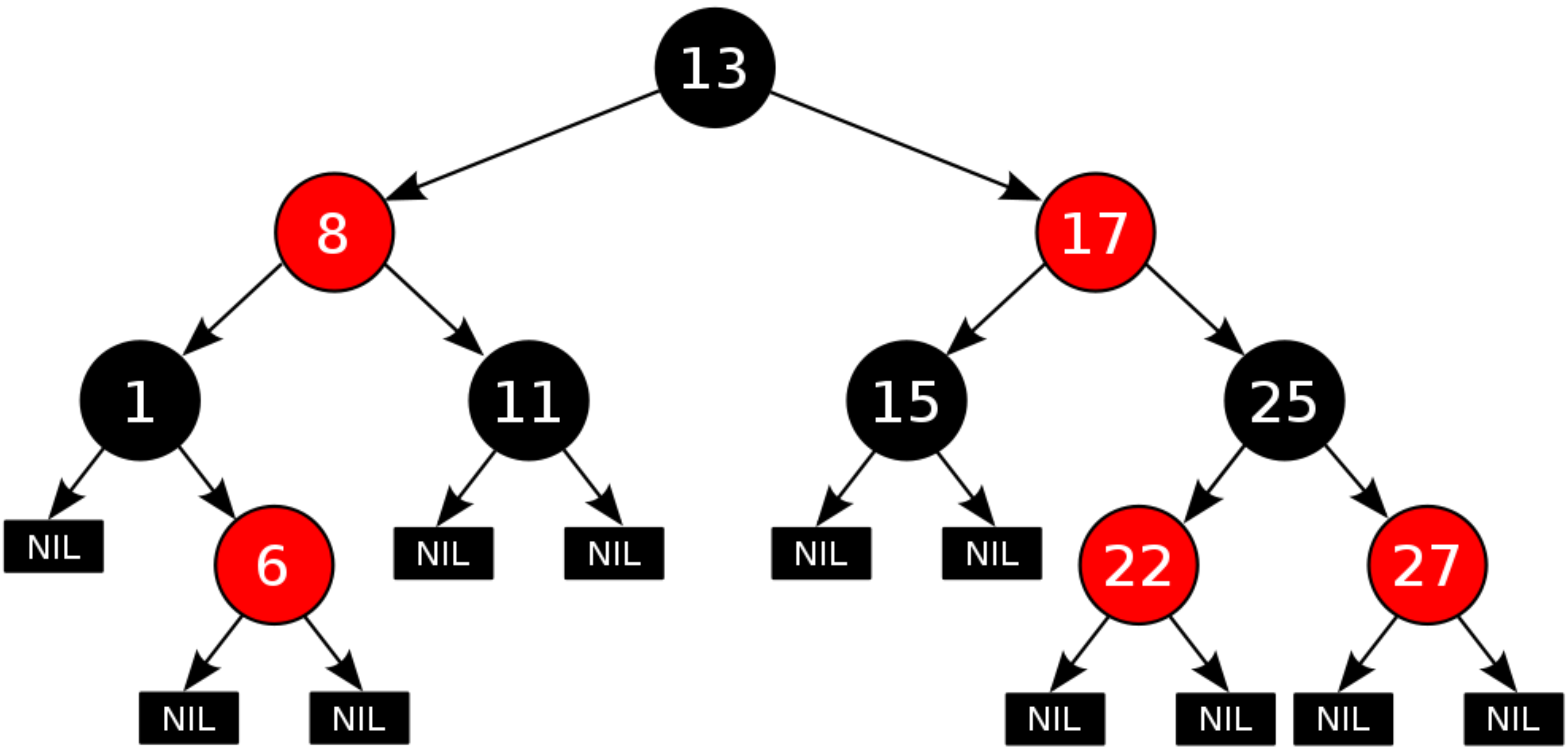
AVL Trees: One of Many

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)



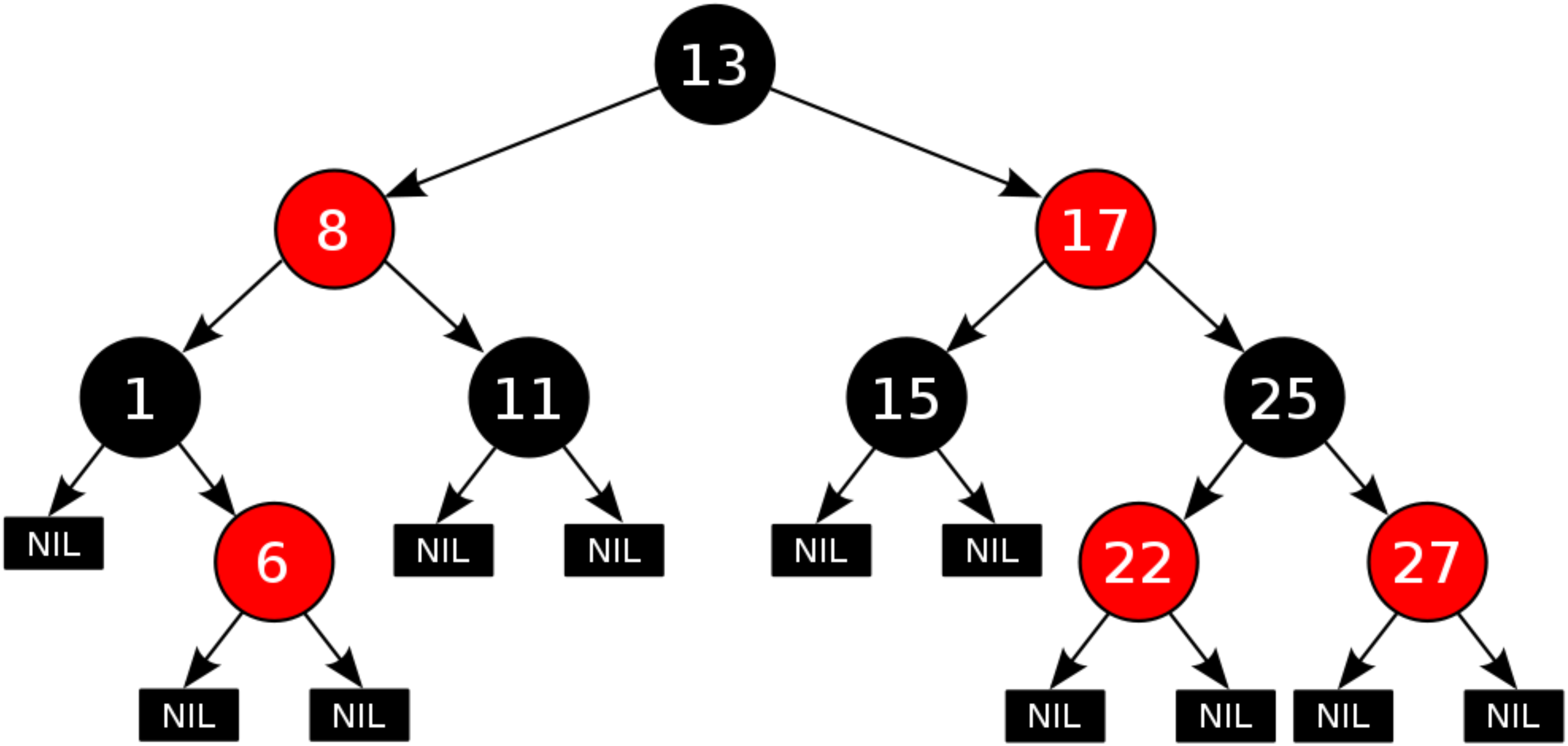
Red-Black Trees

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 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

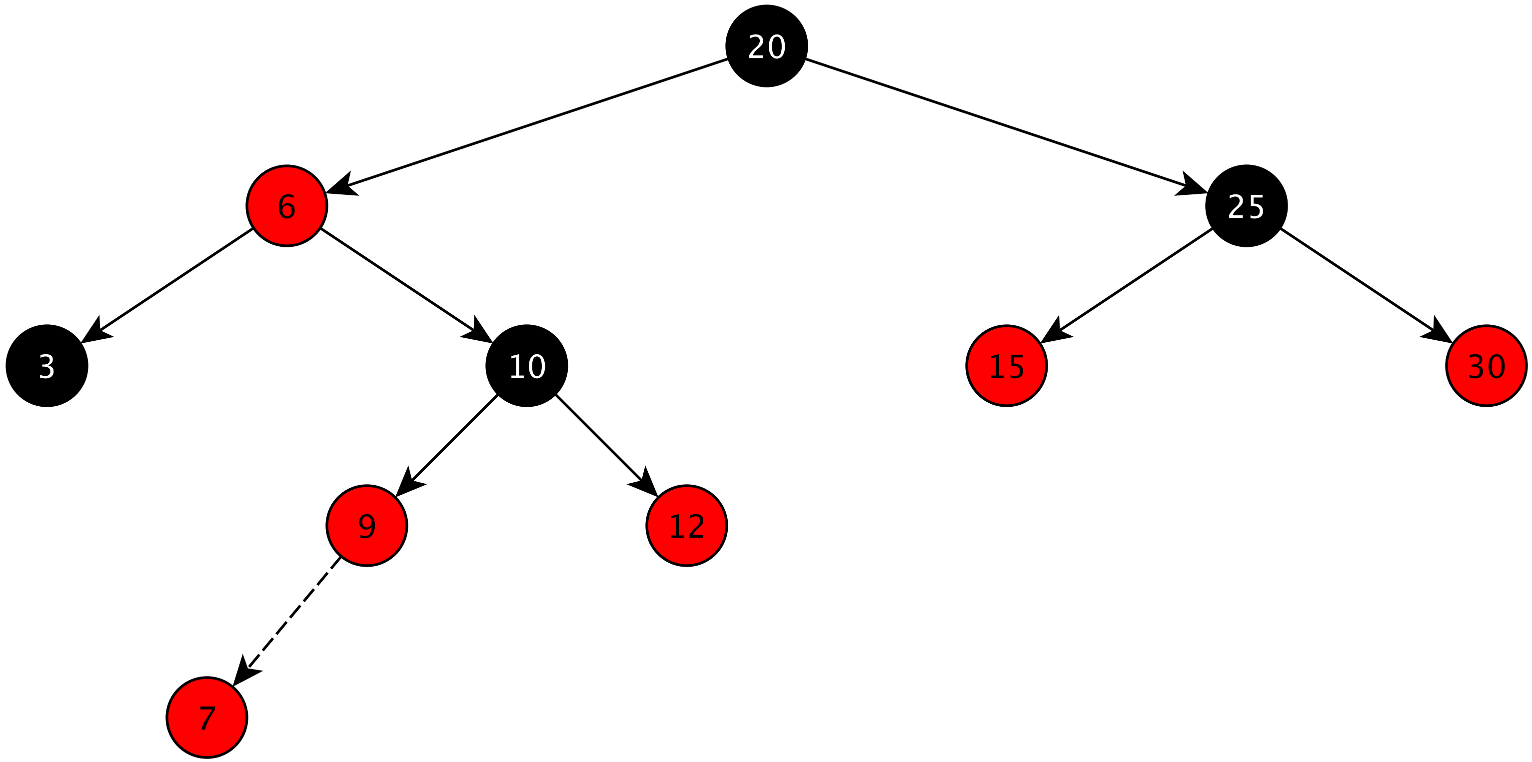
Theorem: A Red-Black tree with n *internal* nodes has height satisfying $h \leq 2\log(n+1)$

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 $2\log_2(n+1) \geq h$

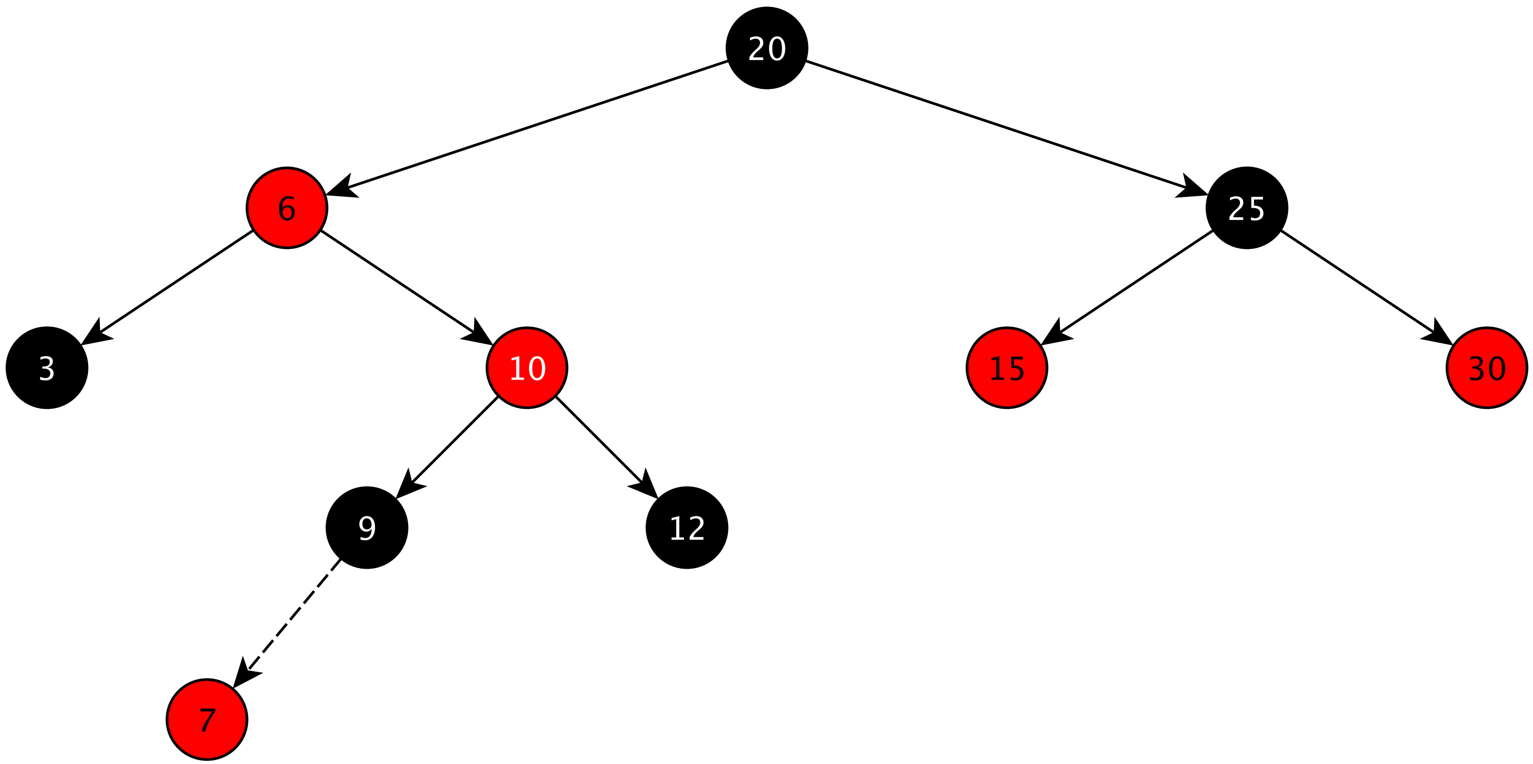
Corollary: R-B trees with n nodes have height O(log n)

Red-Black Tree Insertion



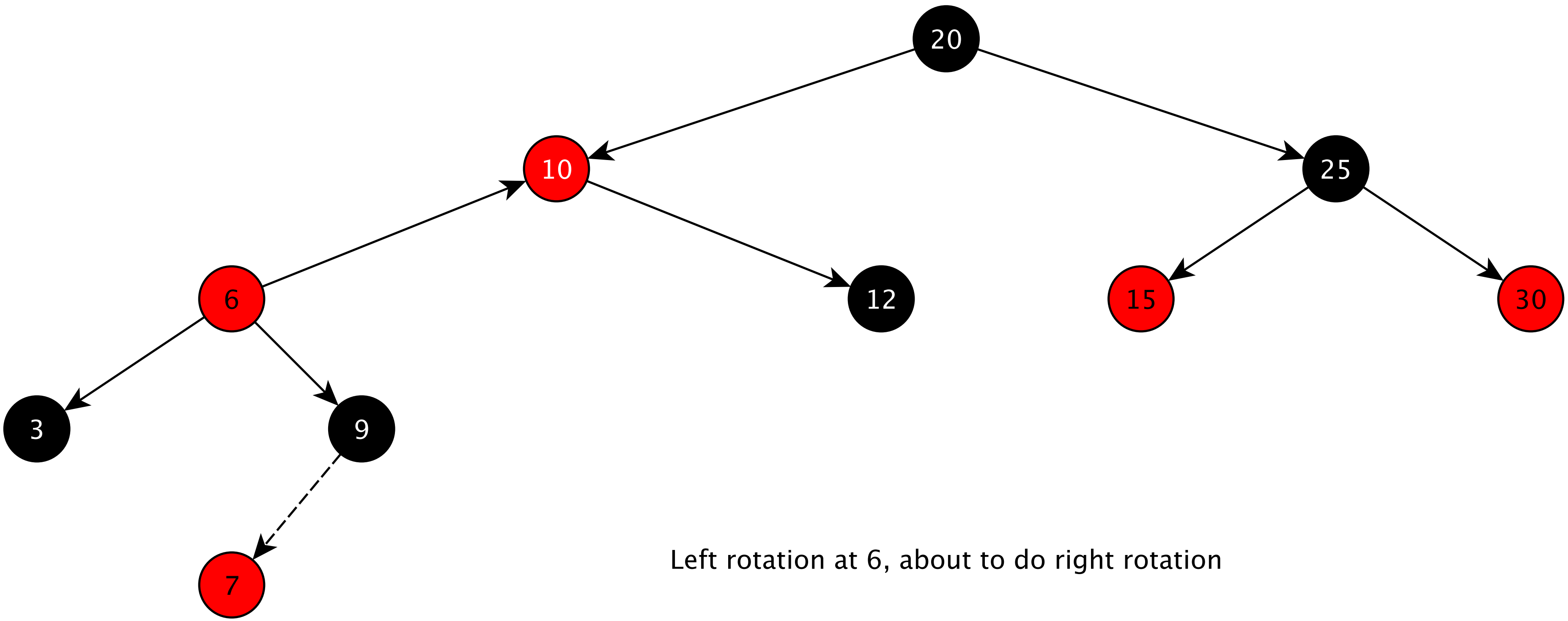
Black empty leaves not drawn. 7 just added Black-height still 2.

Red-Black Tree Insertion



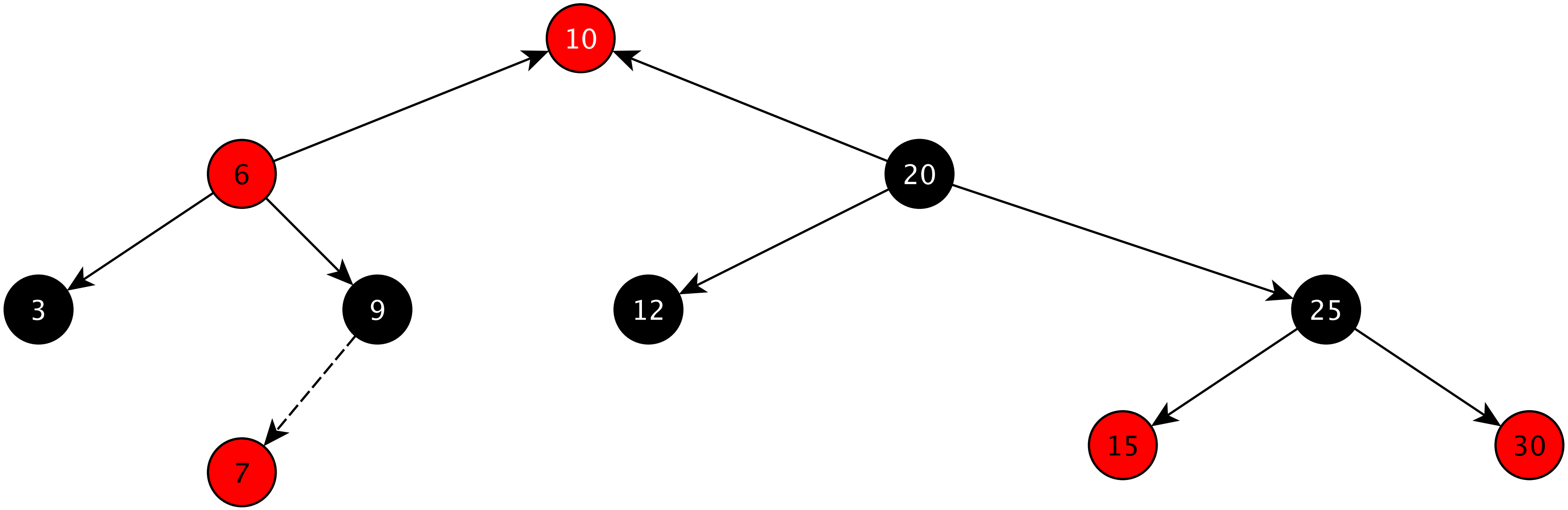
Black height still 2, color violation moved up

Red-Black Tree Insertion



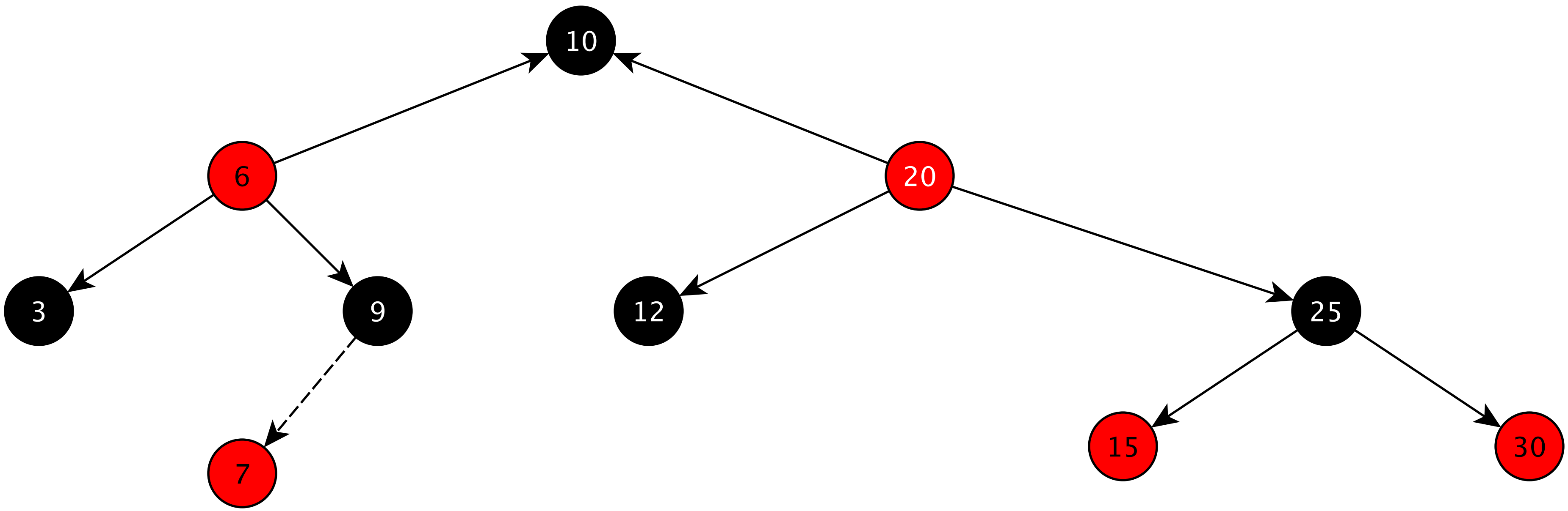
Left rotation at 6, about to do right rotation

Red-Black Tree Insertion



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

Red-Black Tree Insertion



Color conditions restored, black-height restored.

Splay Trees

Splay trees are self-adjusting binary trees

- Each time a node is accessed, it is moved to root position via rotations
- No metadata at all. Just rotate up each element you access

Splay Trees

Splay trees are self-adjusting binary trees

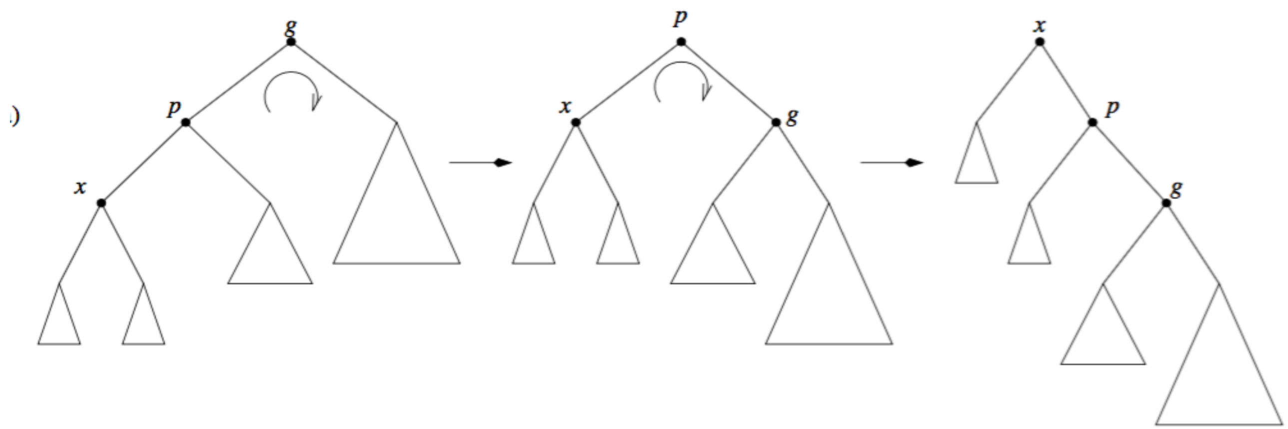
- Each time a node is accessed, it is moved to root position via rotations
- No guarantee of balance (or shallow height)
- But good *amortized* performance

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

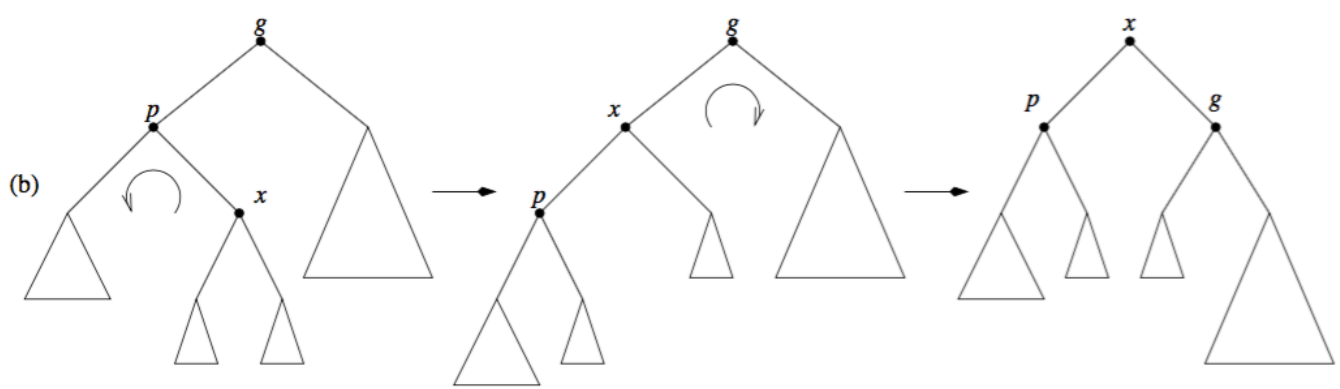
- As good as an AVL or Red-Black Tree!

Splay Tree Rotations

Right Zig-Zig Rotation (left version too)



Right Zig-Zag Rotation (left version too)



Dynamic Optimality

- Conjecture: For any sequence of access operations, if the best possible Binary Search Tree takes X operations, then a splay tree takes O(X) operations

- Essentially: keeping no metadata, and with no knowledge of the future, splay trees do as well as a perfect tree that knows the whole sequence in advance

Dynamic Optimality

- Conjecture: For any sequence of access operations, if the best possible Binary Search Tree takes X operations, then a splay tree takes O(X) operations

- One consequence would be: splay trees can handle stack or queue operations in O(1) average operations like a DLL

Dynamic Optimality

- Open since 1985
- Recent progress [Levy Tarjan 2019]: if a splay tree's performance only improves when we remove operations, then the splay tree is dynamically optimal

Dynamic Optimality

- Some really cool math in this area