CS151 Intro to Data Structures

AVL Trees
Splay Trees

Announcements

HW8 (last homework) due Dec 15th

Outline

Review: Tree Rotation and AVL Trees

Splay Trees

Review: AVL Trees

A self-balancing binary search tree where the height difference between the left and right subtrees is at most 1 for every node.

Guarantees **O(log n)** time for insertion, deletion, and search by maintaining height balance.

Prevents degeneration into linked lists (common issue in unbalanced binary search trees).

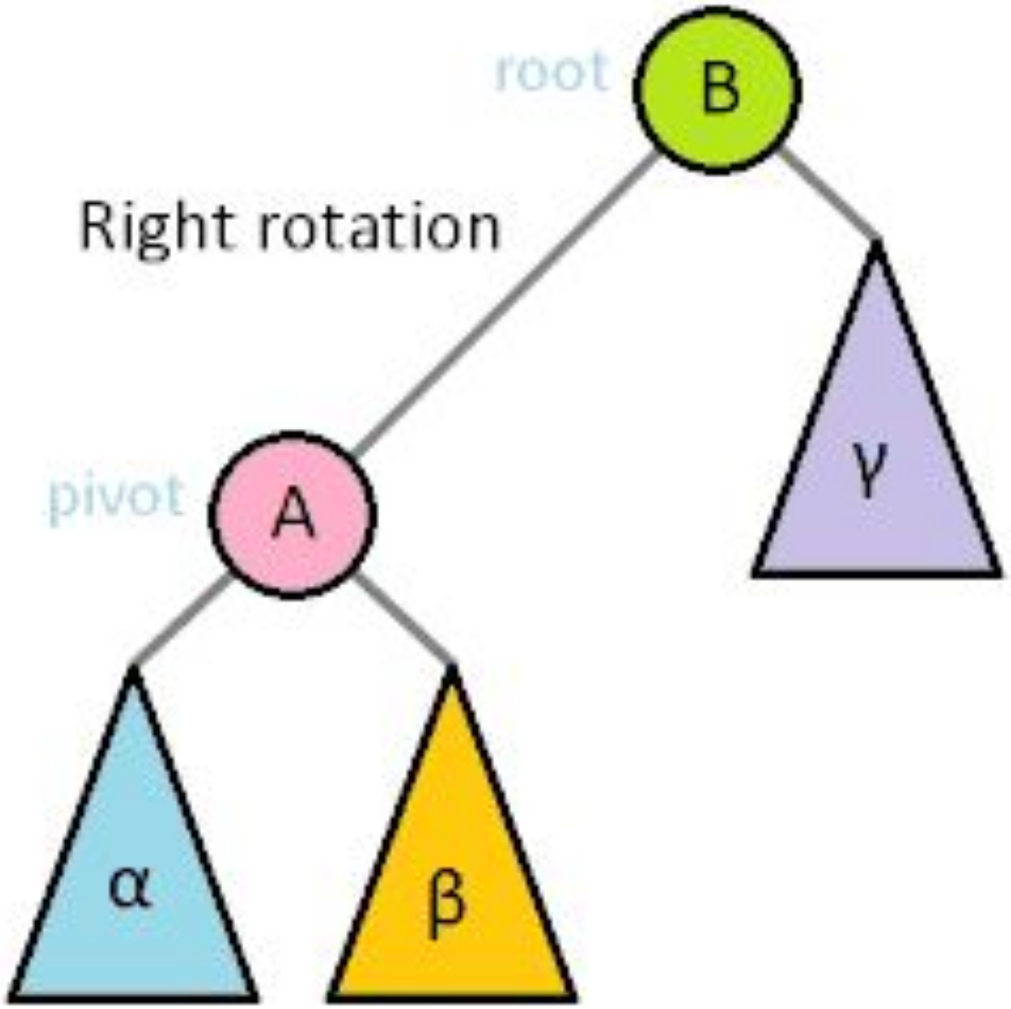
Rotations

Right rotation:

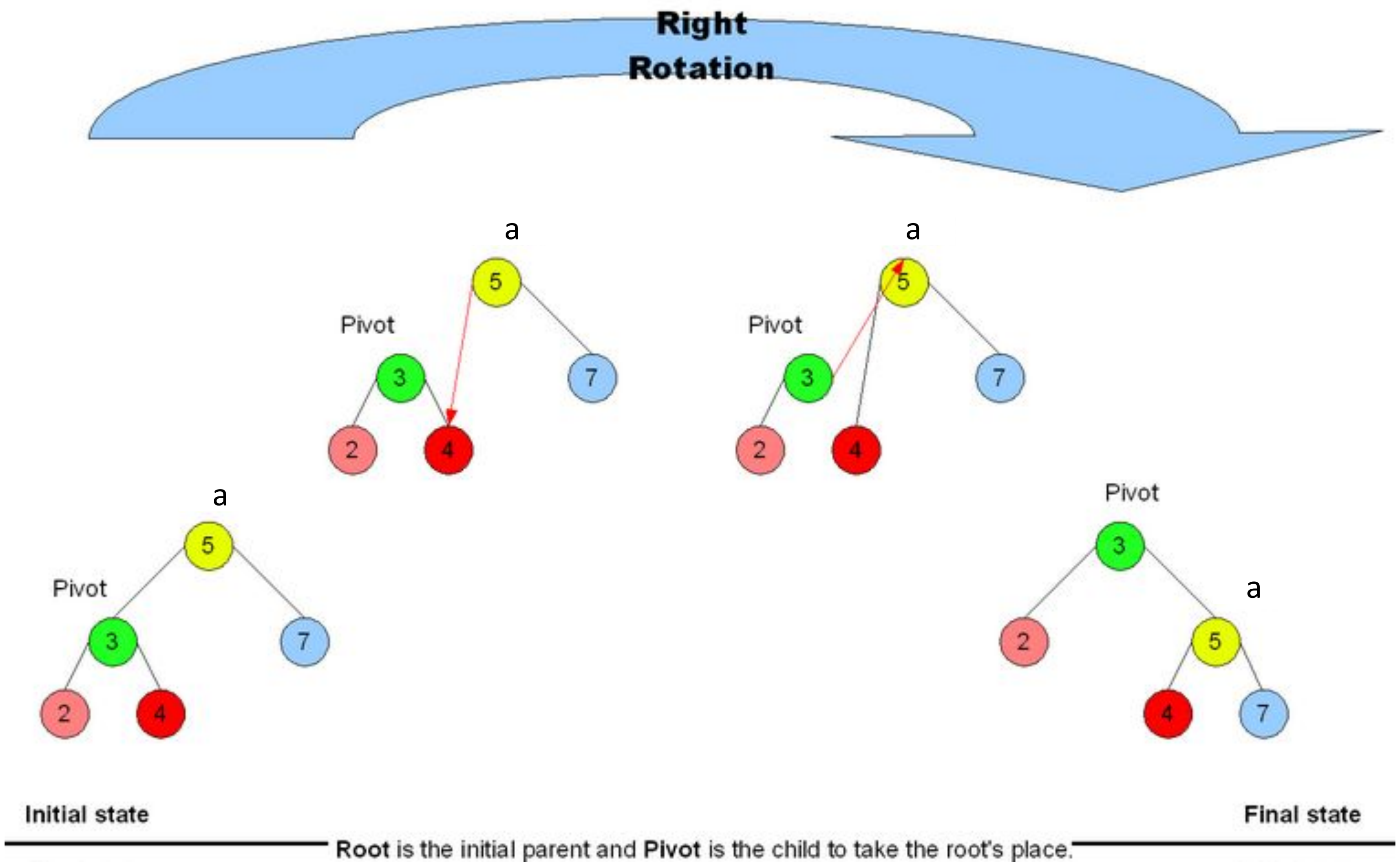
- Performed when left side is heavier
- left child becomes root

Left rotation:

- Performed when right side is heavier
- right child becomes root

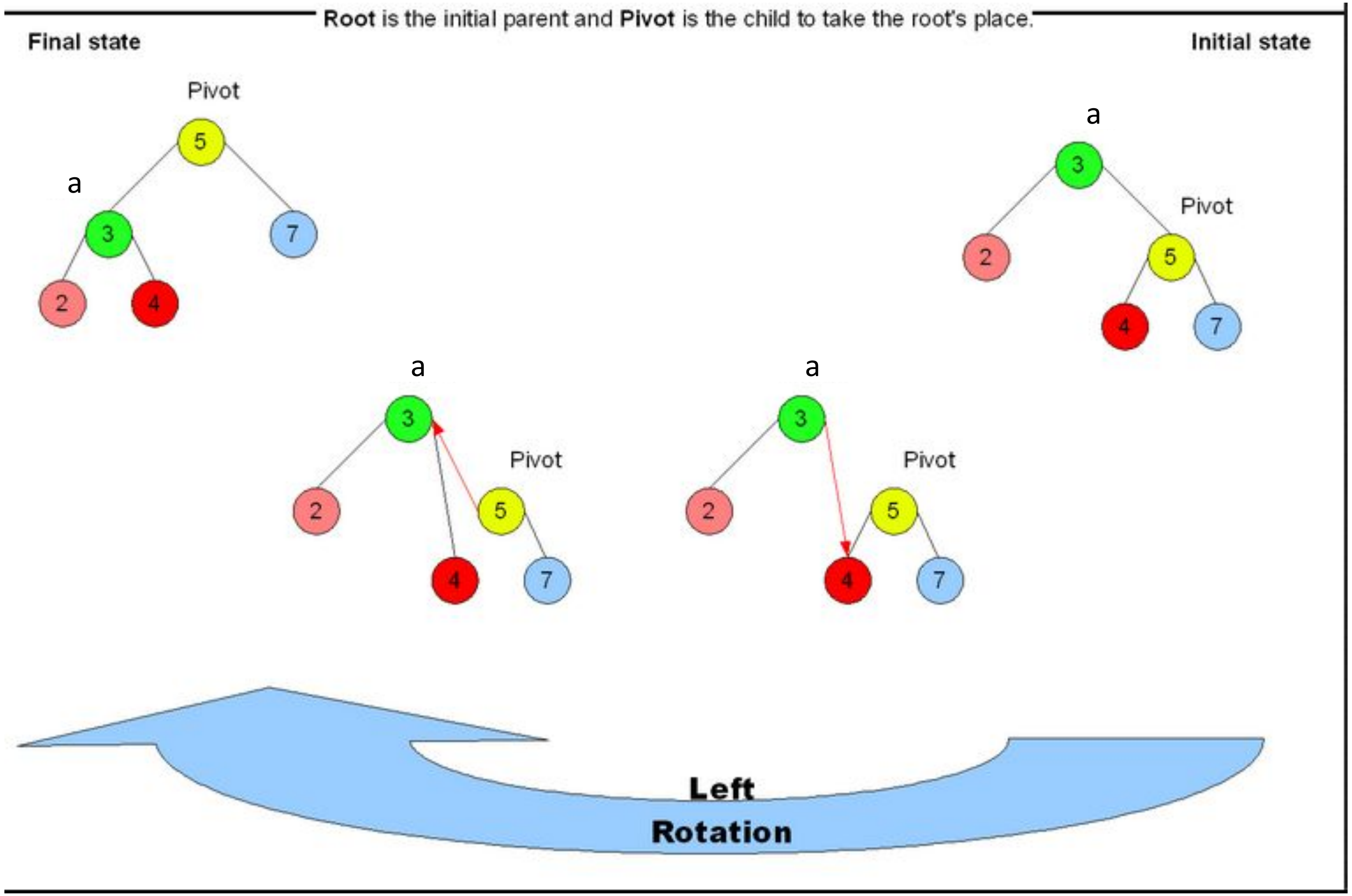


RotateRight Algorithm



1. `a.left = Pivot.right`
2. `Pivot.right = a`

RotateLeft Algorithm



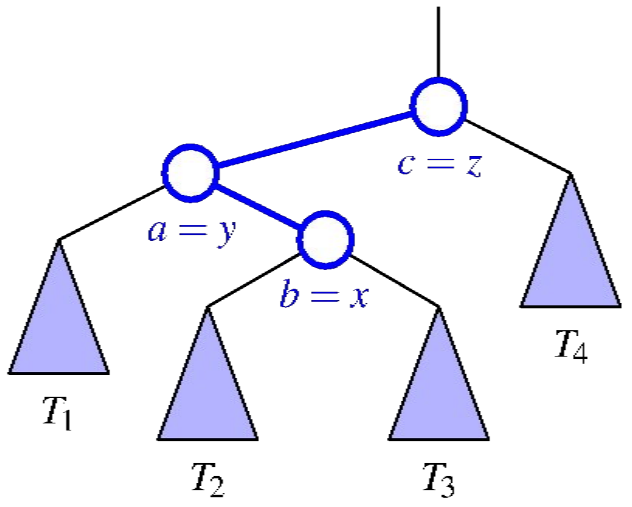
1. `a.right = Pivot.left`
2. `Pivot.left = a`

Double rotation

When do we need a double rotation?

Left subtree is too heavy on the right side

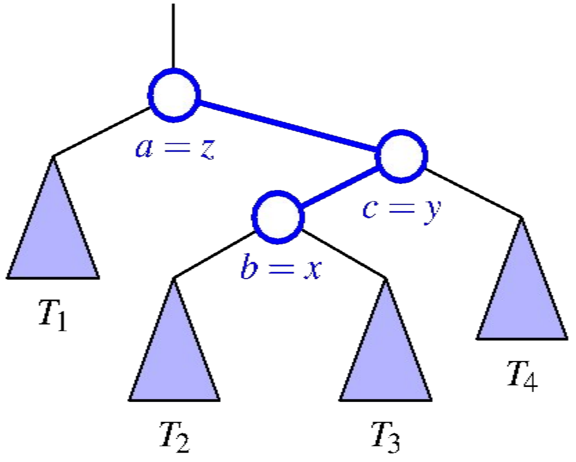
`rotateLeftRight`



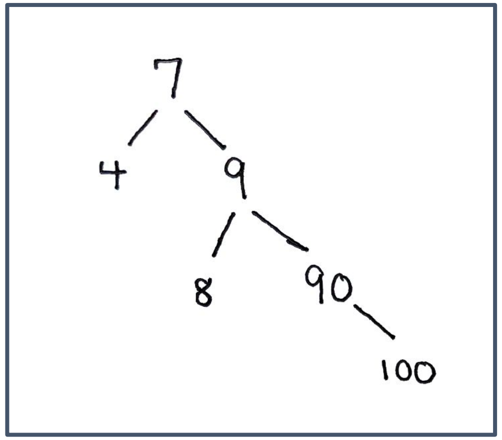
OR

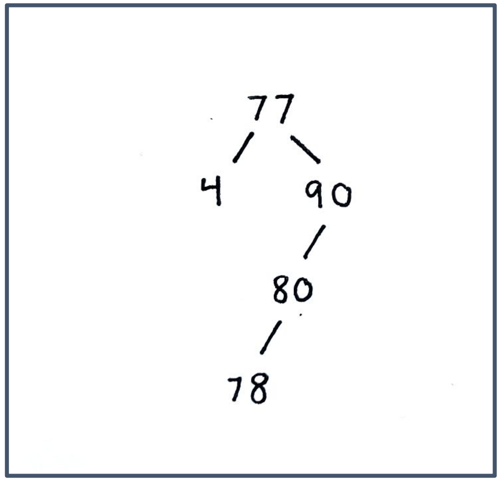
Right subtree is too heavy on the left side

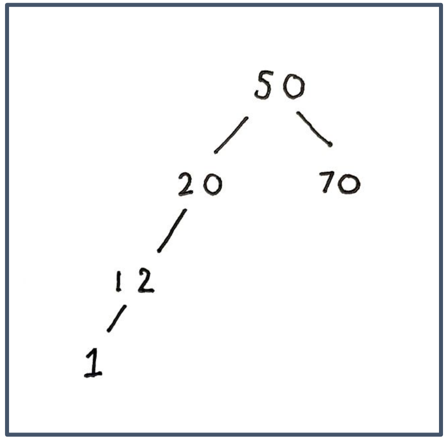
`rotateRightLeft`

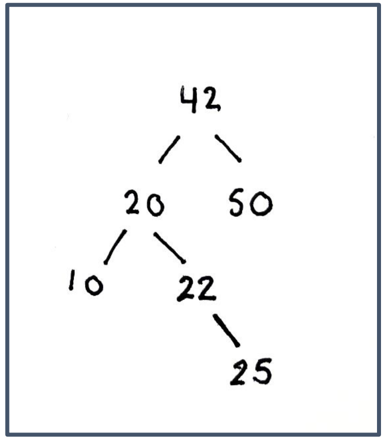


Examples - which way should I rotate?



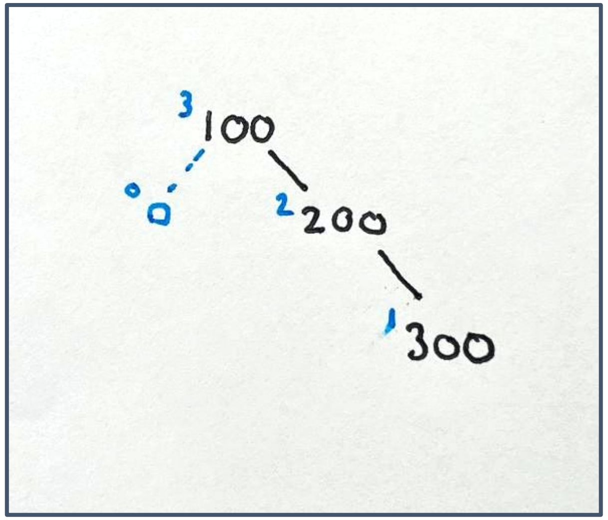


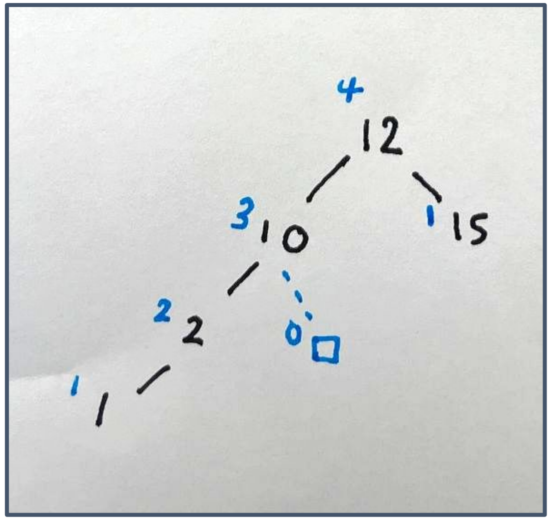


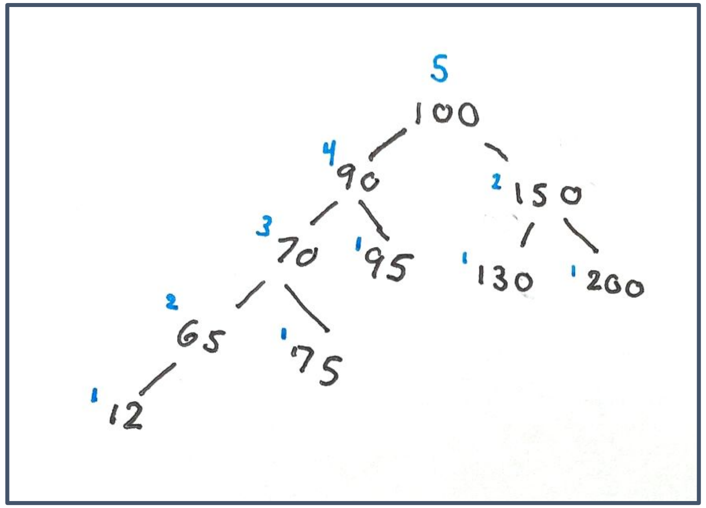


rotateLeft rotateRightLeft rotateRight rotateLeftRight

Which node do we “rebalance over”?



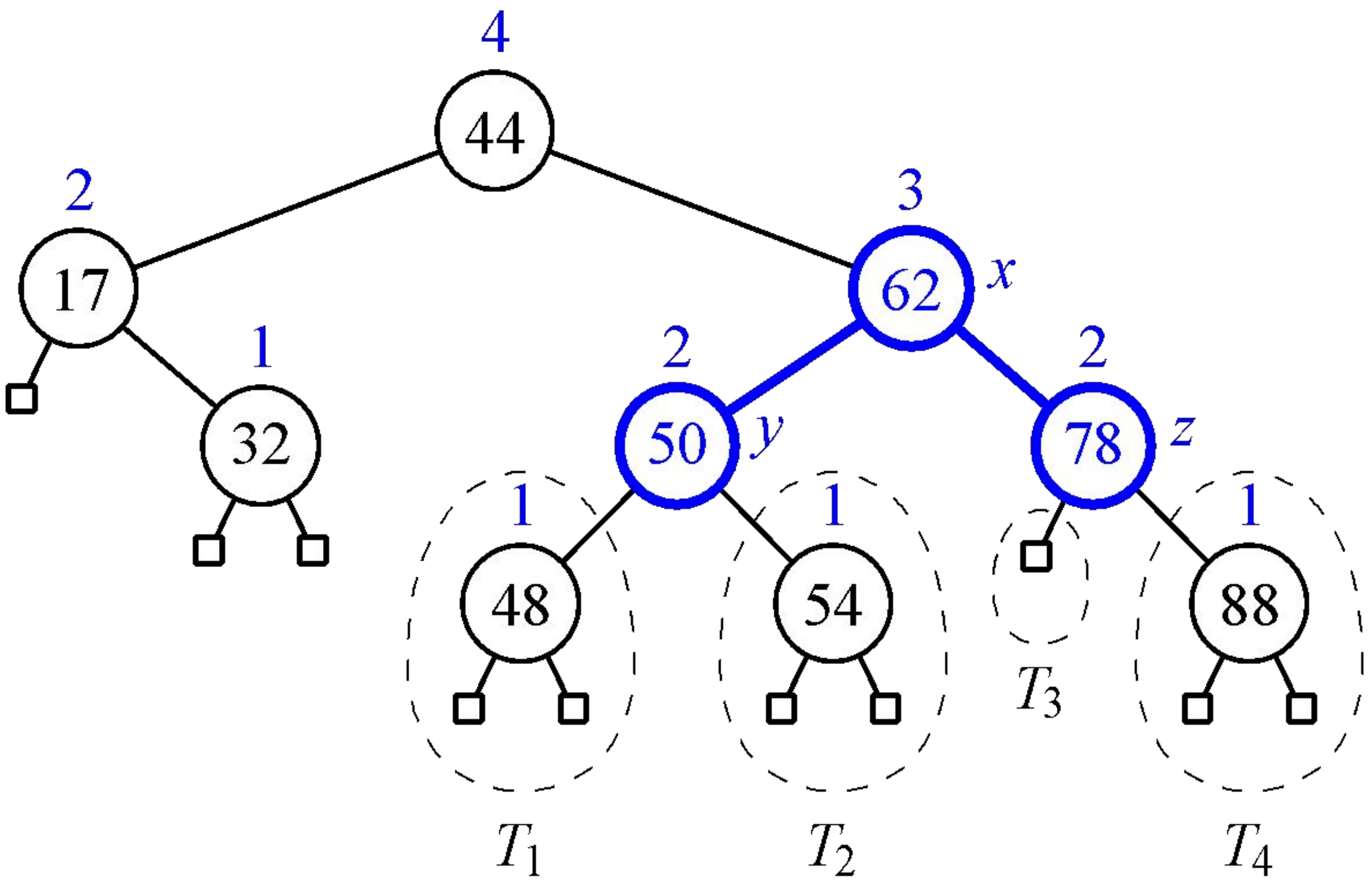




lowest subtree with diff(heights) > 1

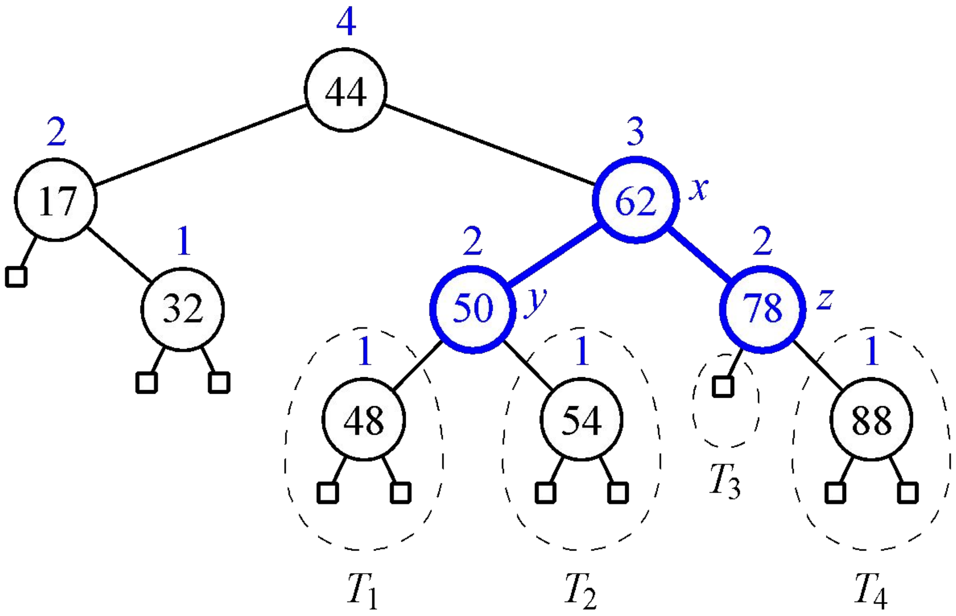
AVL Deletion

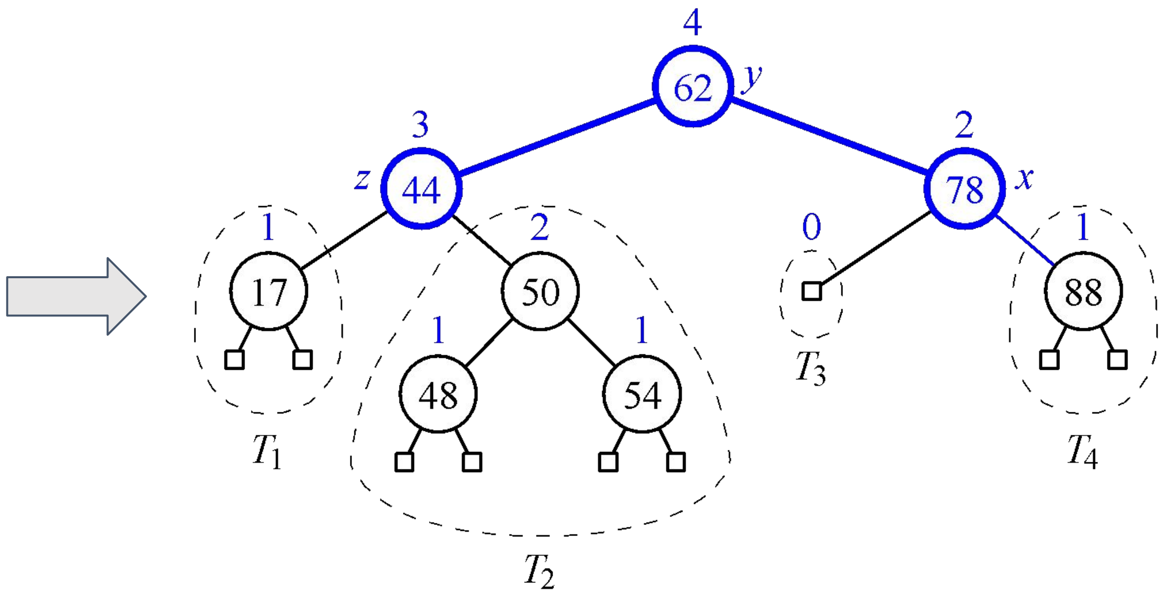
Delete Example 1: 32



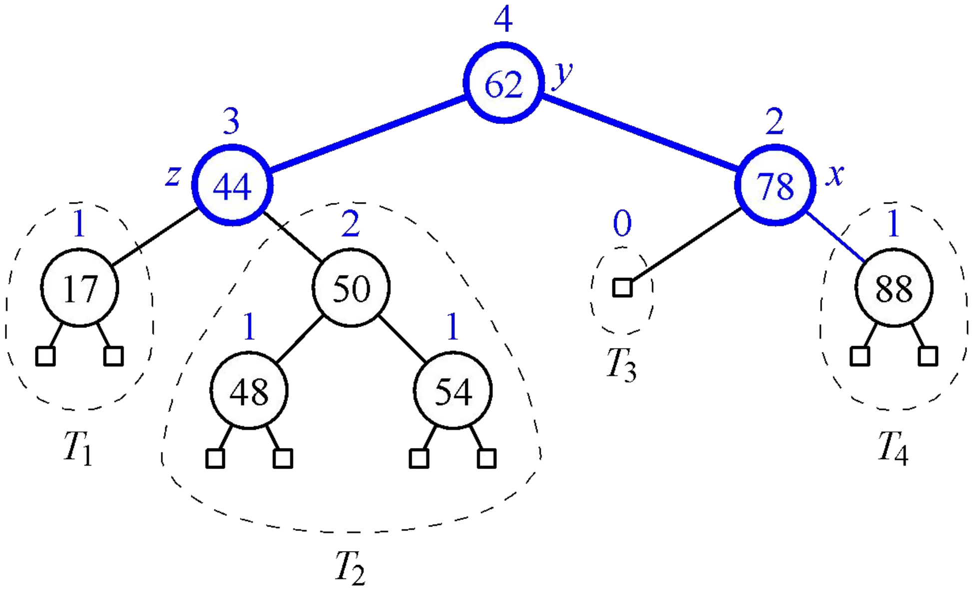
Delete Example 1: 32

rotateLeft



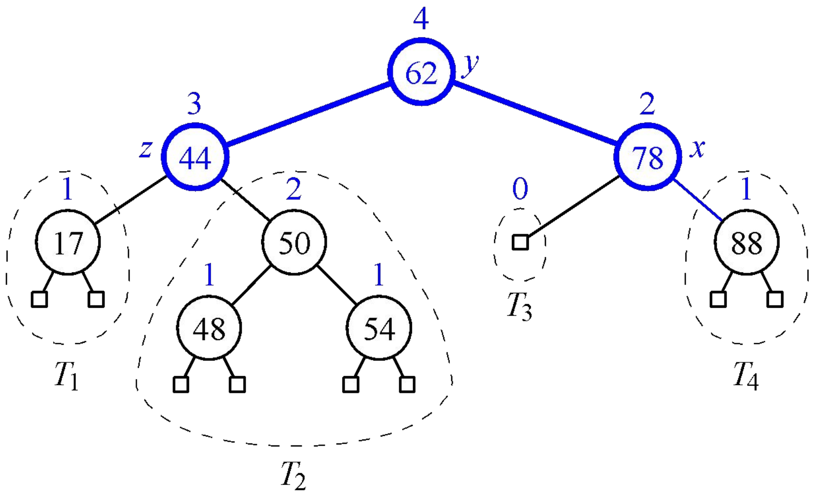


Delete Example 2: 88

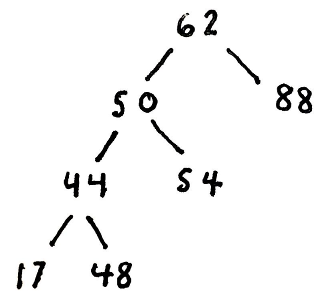


Delete Example 2: 88

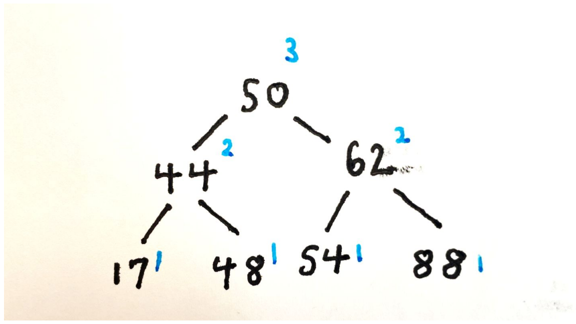
rotateLeftRight



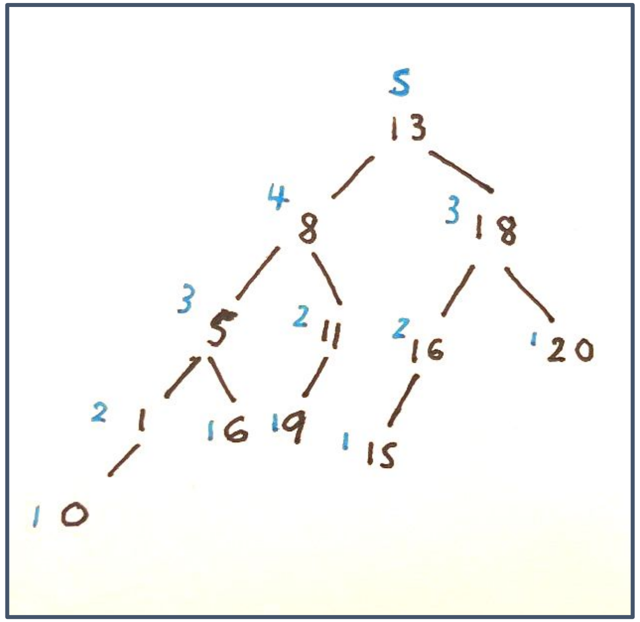
rotateLeft



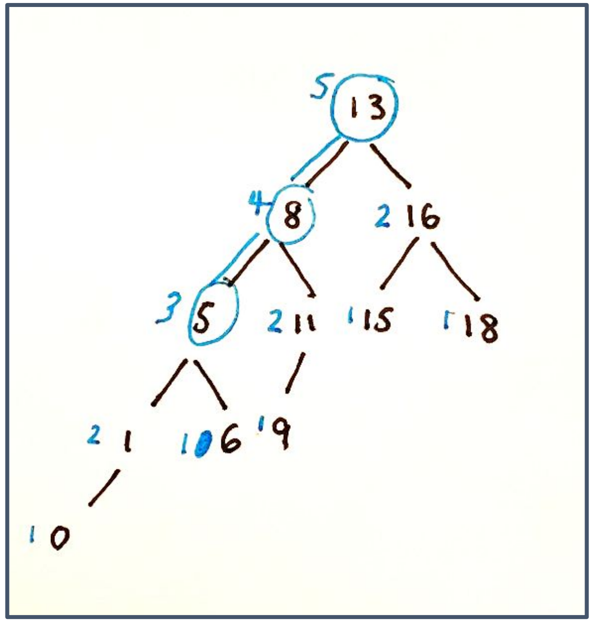
rotateRight



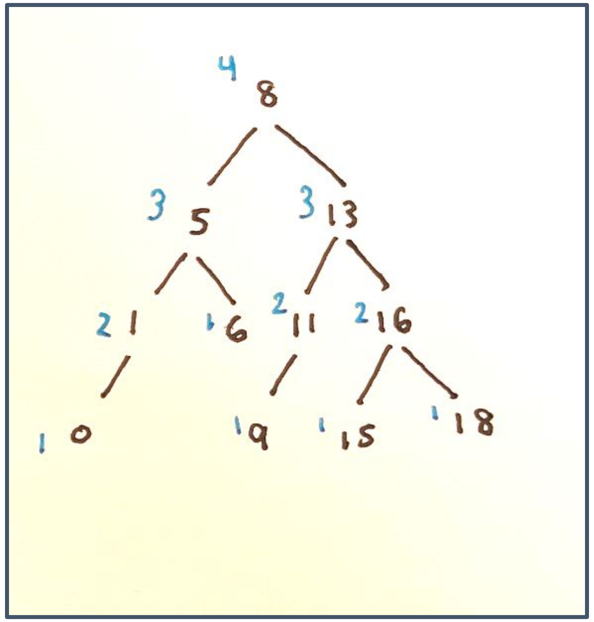
Delete Example 3: 20



rotateRight



rotateRight



Cascading rotations!

Delete Example 3: 20

- Deletion can cause more than one rotation

- Worst case requires O(logn) rotations
  - deleting from a deepest leaf node and rotating each subtree up to the root

Removal

Runtime Complexity?

a. search + find node to rebalance + rotate
b. O(logn) + O(logn) + O(1) = **O(logn)**

Still O(logn) even though we may need multiple rotations?

Why?

-> Even though we may need to find multiple nodes to rebalance we only traverse the height of the tree once

Splay Trees

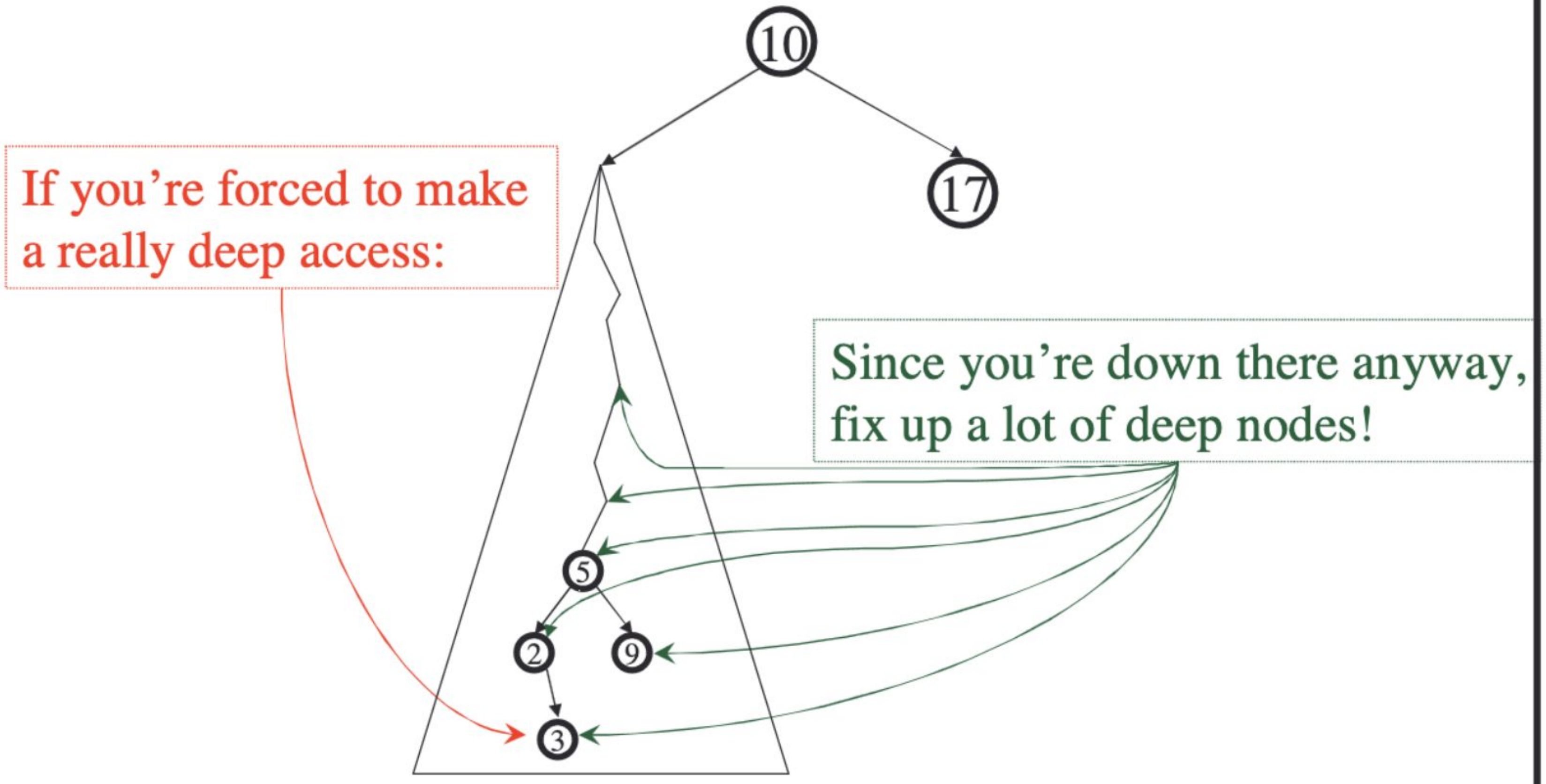
Splay Trees

- No enforcement on height

- Instead, exploits *principle of locality*
  - items that have been recently accessed are more likely to be accessed again in the near future

- “Move to root” operation
  - When a node is accessed (searched, inserted, or deleted), it becomes the root of the tree by performing a series of rotations called "*splays*"

The Splay Tree Idea

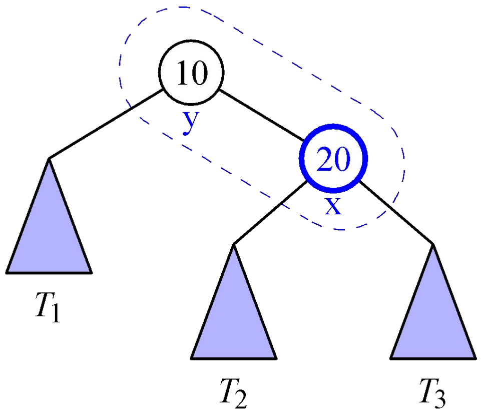


1

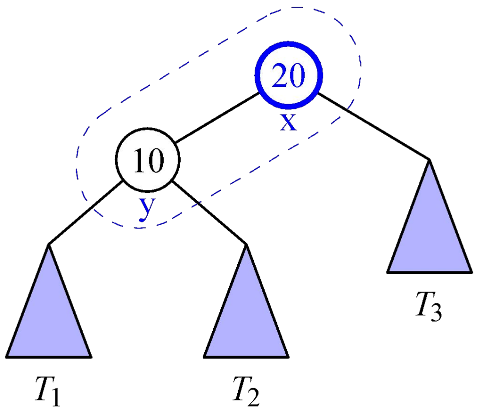
Splaying

- Move to root operation requires a **zig / zag** restructuring

- **zig**
  a. accessed node becomes root of subtree
  b. parent becomes child



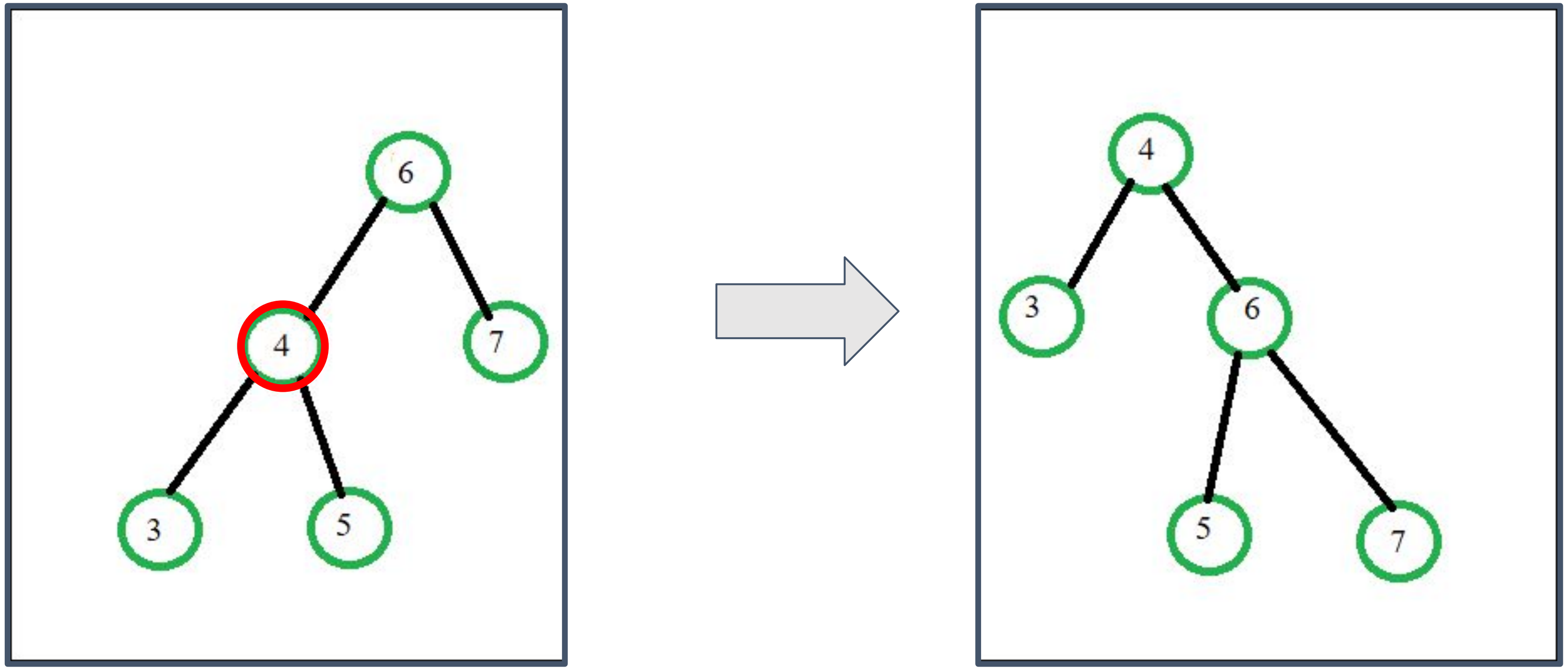
before



after

Splaying - Zig

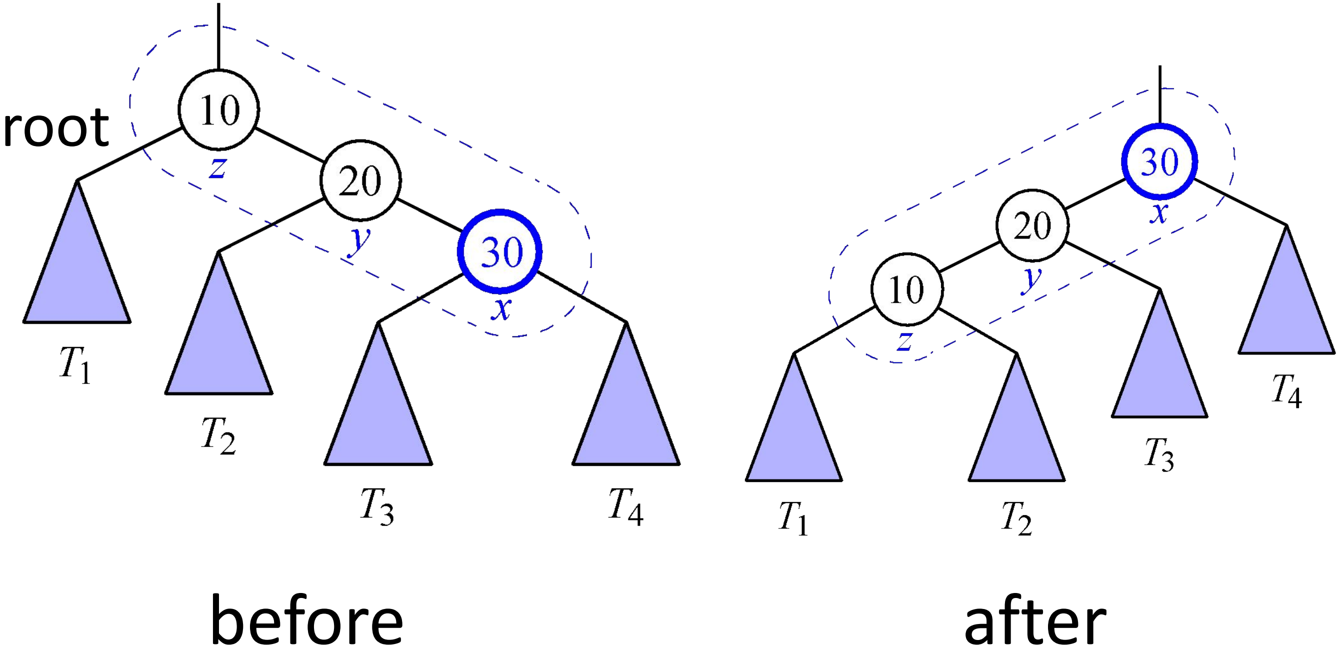
- **zig**
  a. accessed node becomes root of subtree
  b. parent becomes child



Splaying: Zig-Zig

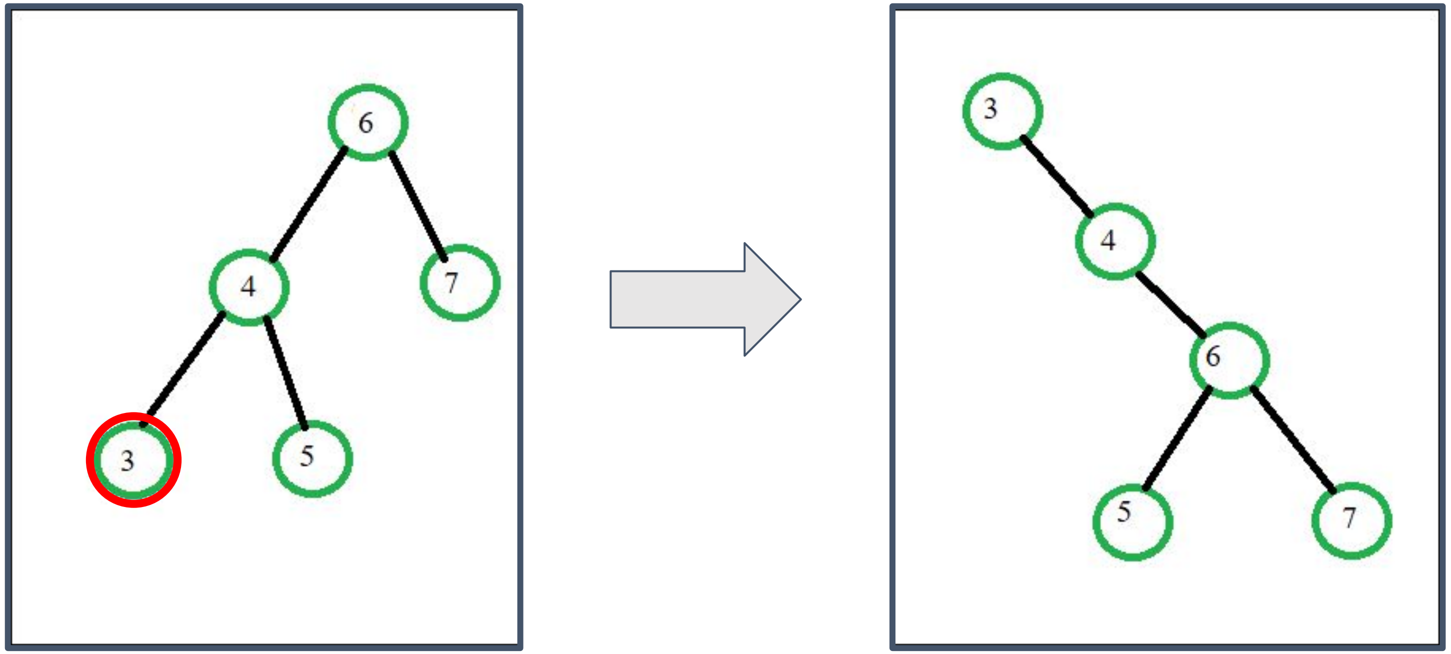
zig-zig:

- step 1: **zig**
  a. accessed node's parent (y) becomes root
  b. parent becomes child
- step 2: **zig**
  a. accessed node (x) becomes root
  b. parent becomes child



zig-zig:

- step 1: **zig**
  a. accessed node's parent (4) becomes root
  b. parent becomes child
- step 2: **zig**
  a. accessed node (3) becomes root
  b. parent becomes child

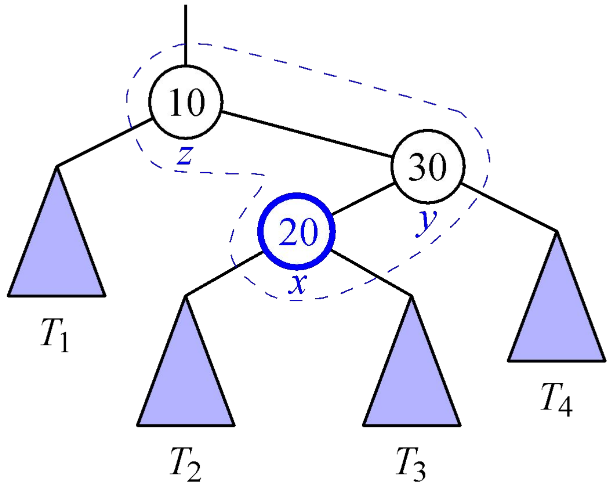


Splaying: Zig-Zag

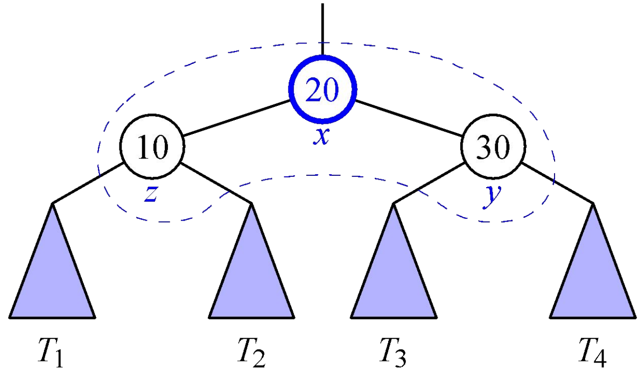
zig-zag:

- step 1: **zig**
  a. accessed node (x) becomes root of subtree
  b. parent becomes child
- step 2: **zag**
  a. accessed node (x) becomes root of tree
  b. parent becomes child

Called zig-zag because the second step is a rotation in the **opposite direction**

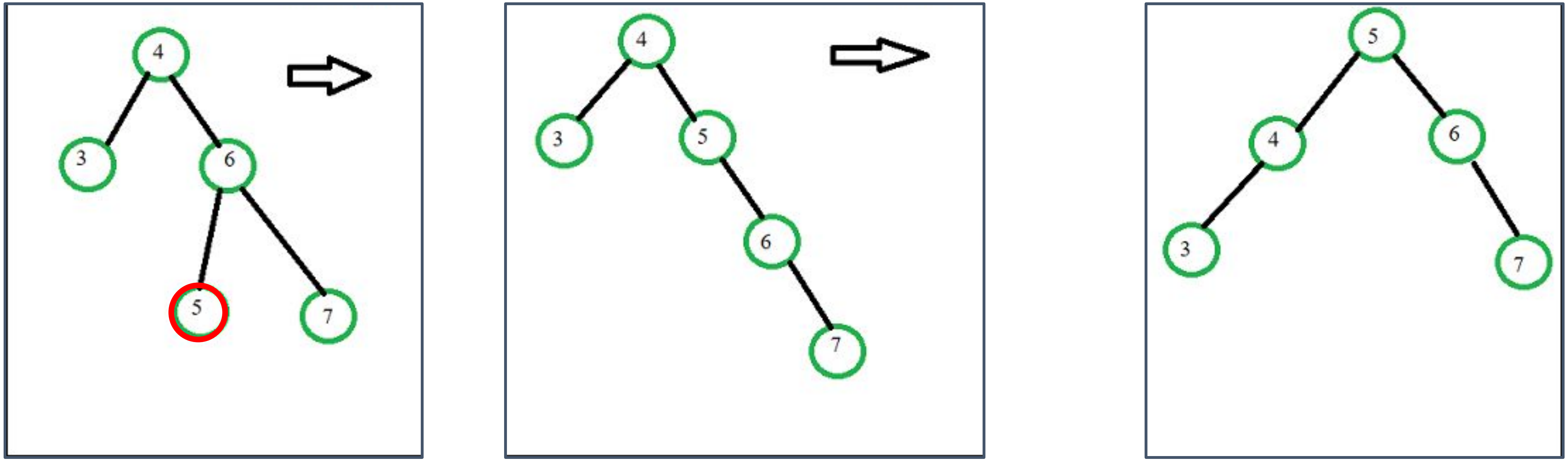


before



after

Splaying: Zig-Zag



Splaying: Zig-Zag and Zig-Zig

- Analogous to a double rotation in AVLs

- Zig-Zag
  - Two rotations in opposite directions

- Zig-Zig
  - Two rotations in the same direction

Which Transformation to Perform

1. **Zig**: accessed node does not have a grandparent. Only one rotation required

1. **Zig-Zig**: accessed node and its parent are both children on the same side
   a. x is the left child of y and y is the left child of z OR
   b. x is the right child of y and y is the right child of z

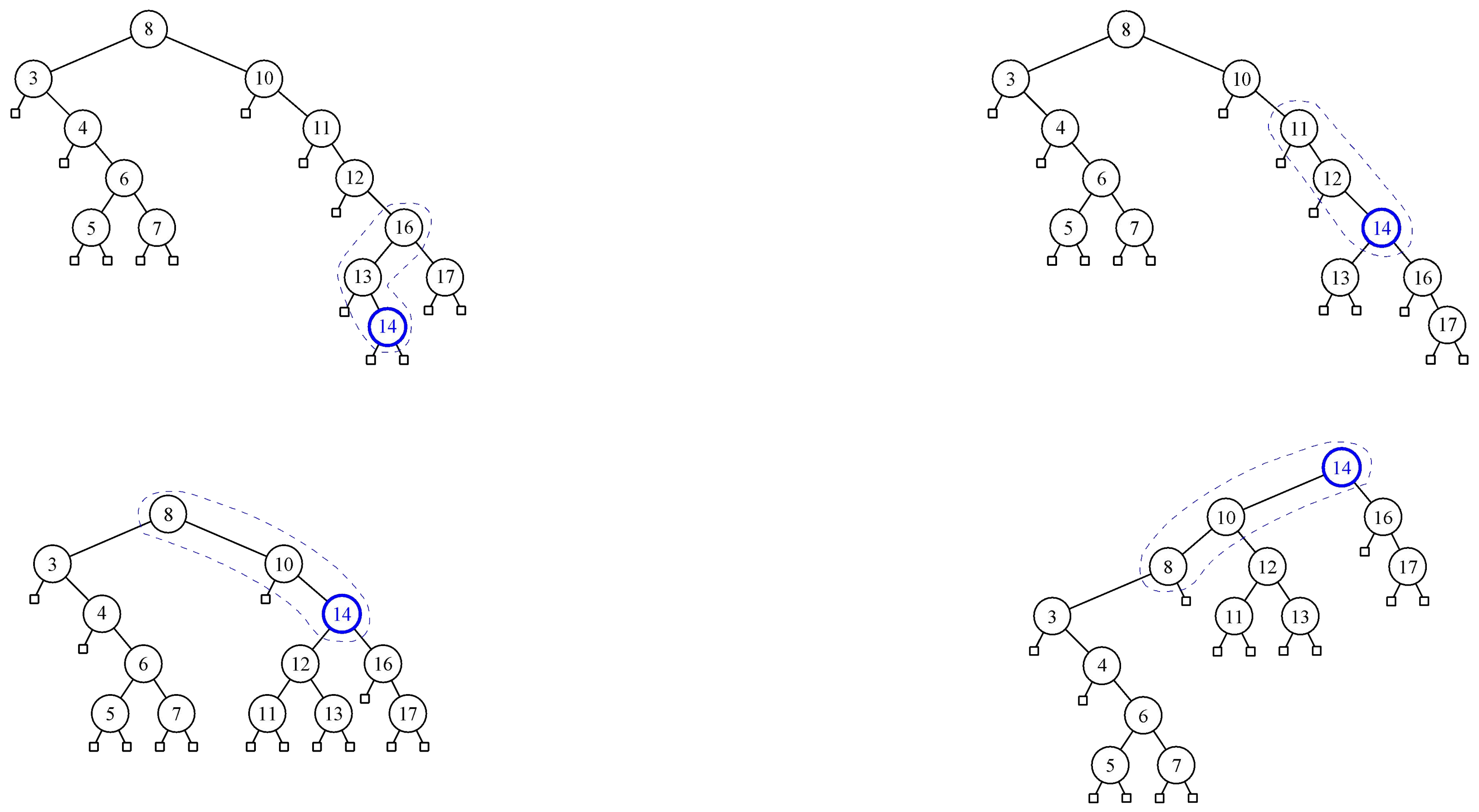
1. **Zig-Zag:** one of x and y is a right child and the other is a left child
   b. Analogous to double rotations in AVLs

Splaying

Repeating restructurings until the accessed node x is at the root of the tree.

Series of zig, zig-zig, and zig-zag rotations

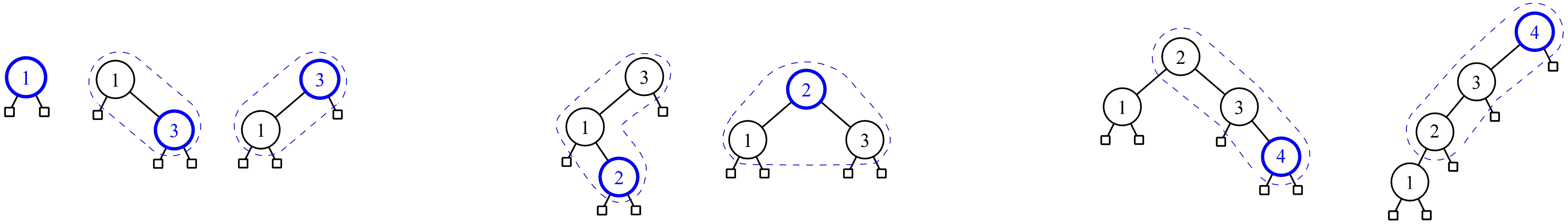
Example - insert(14)



When/what to Splay

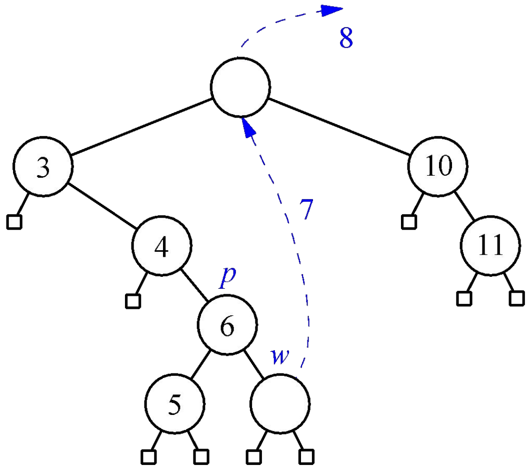
on search for x: if x is found, splay x. else splay x's parent

on insert x: splay x after insertion



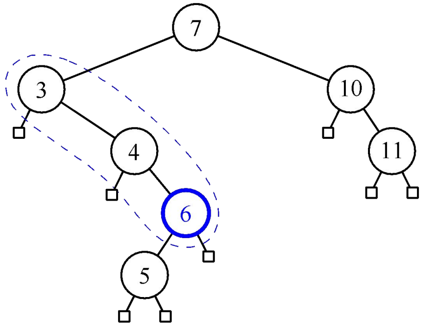
on remove x: splay parent of removed leaf node

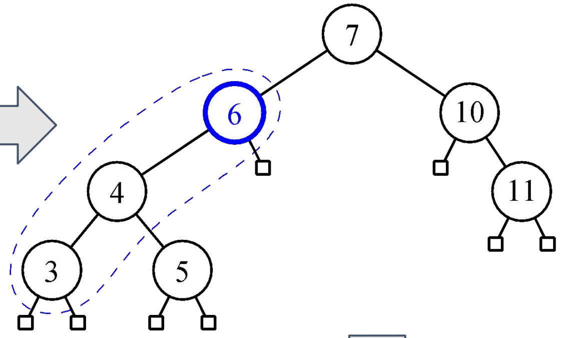
Deletion: remove(8)

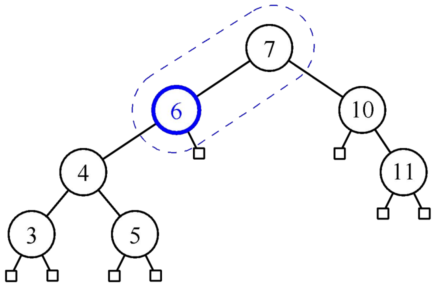


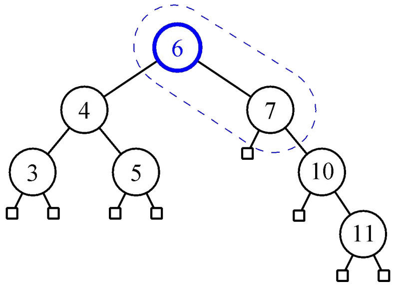
remove 8 and replace it with 7 (largest node on left)

splay 6 (parent of removed node)









Analysis of Splaying

Runtime of restructuring operations:

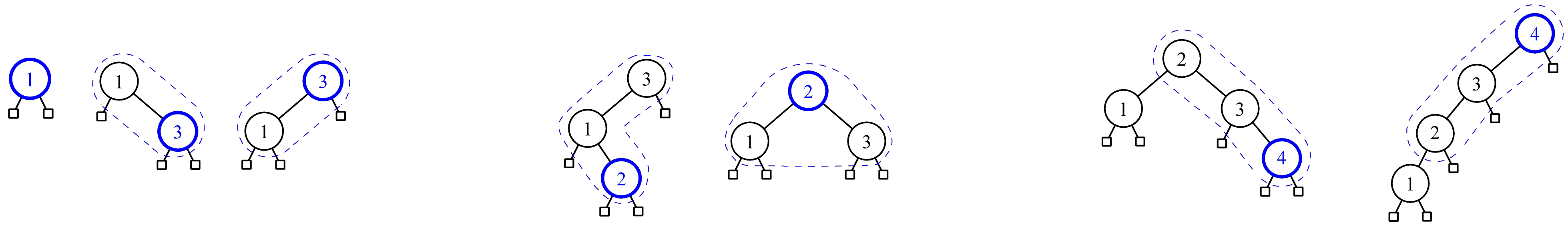
1. zig
   a. O(1)
2. zig-zig
   a. O(1)
3. zig-zag
   a. O(1)

Analysis of Splaying

Splay trees do rotations after every operation (including search)

Each rotation is constant time..

What is the max number of rotations we may need to perform?



insert(0)

Analysis of Splaying

Each rotation is constant time..

What is the max number of rotations we may need to perform?

O(n)

Analysis of Splaying

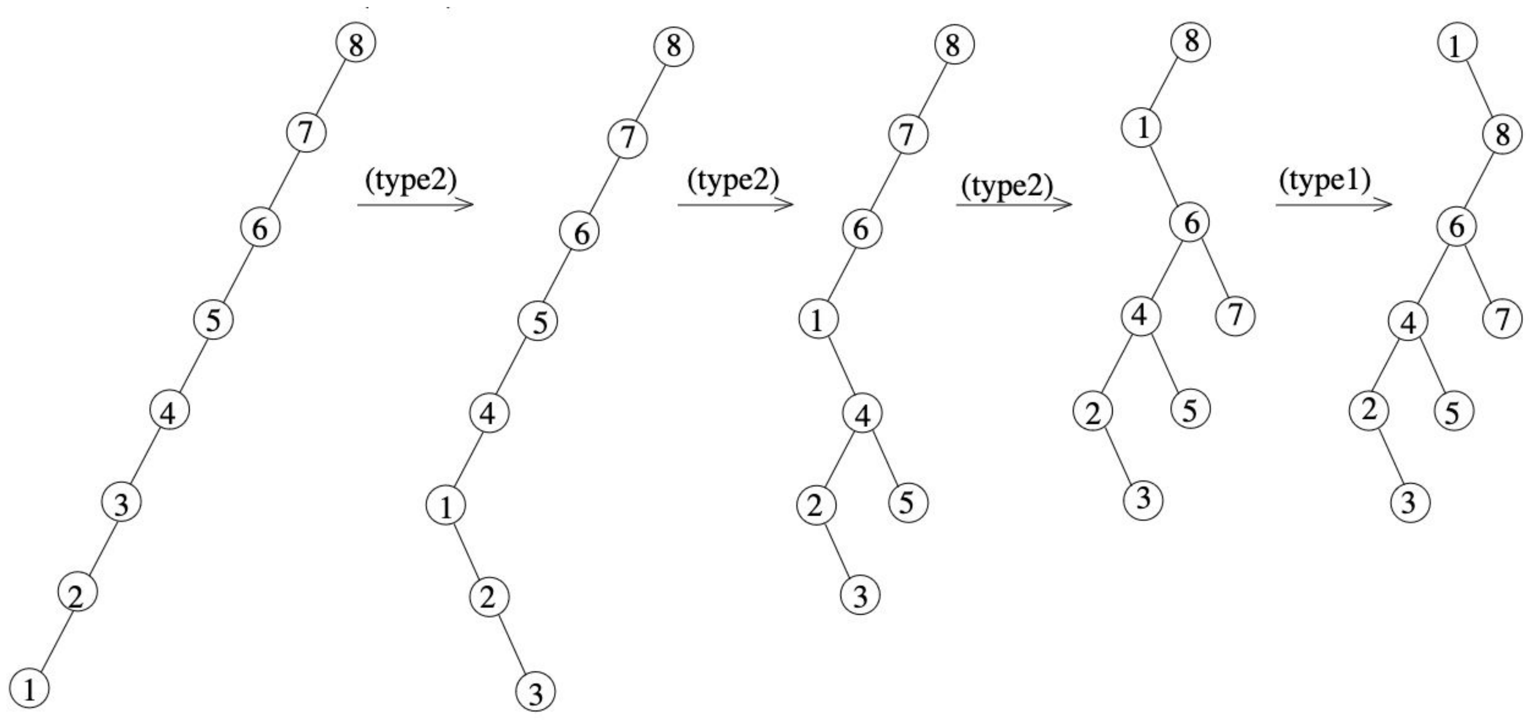
Worst case:

- Search:
  - O(n)
- Remove:
  - O(n)
- Insert:
  - O(n)

Analysis of Splaying

High cost operations often balance the tree

Amortized: O(logn)



Check your understanding

Which is more efficient? Splay tree or AVL

Inserting the numbers {100, 200, 300, 400, 500} in increasing order and removing them in decreasing order

Summary

1. AVL Trees - maintains logn height by rotating after any operations that cause imbalance

2. Splay Trees - moves accessed nodes to the root. Worst case O(n) operations, but amortized O(logn)