CS151 Intro to Data Structures

AVL Trees
Splay Trees

Announcements

April 29: No lab. Extra credit opportunity instead

HW7 due Sunday
HW8 released tonight - due May 9th

Outline

Review: Tree Rotation and AVL Trees

Splay Trees

Red-Black Trees

Runtime Complexity

Runtime Complexity of rotation?

- O(1)

Constant time... we’re just updating links

Balanced Binary Trees

- Difference of heights of left and right subtrees at any node is at most 1
- Add an operation to BSTs to maintain balance:
  - **Rotation**

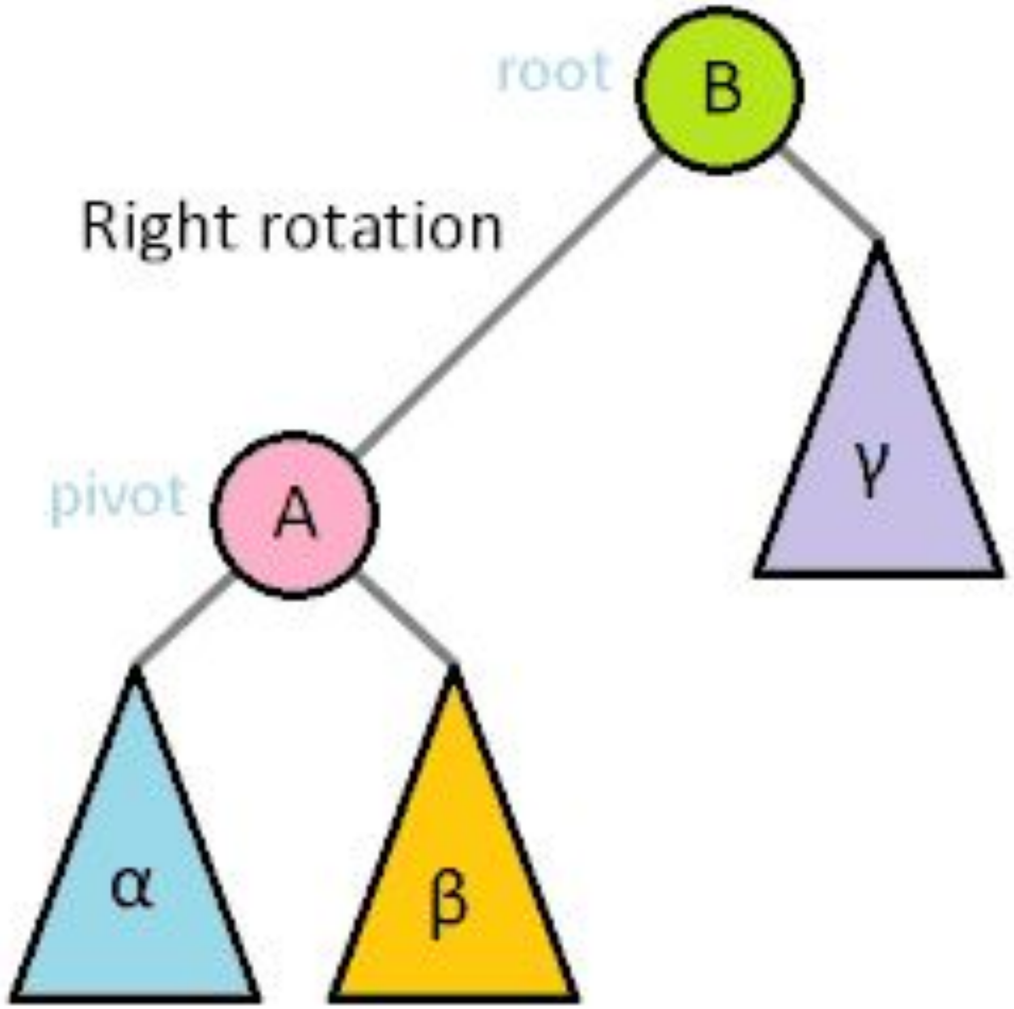
Rotations

Right rotation:

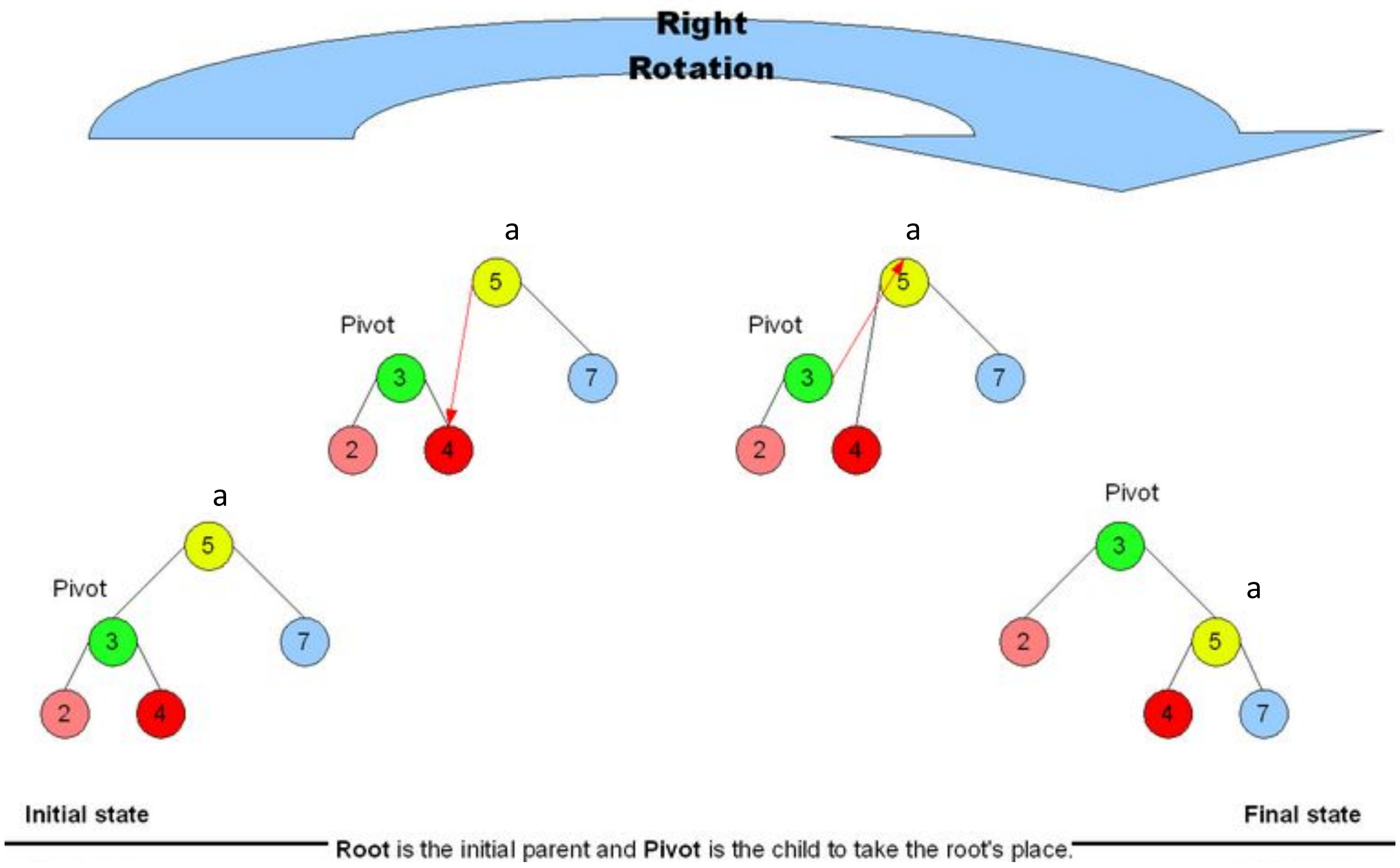
- Performed when left side is heavier
- left child becomes root

Left rotation:

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



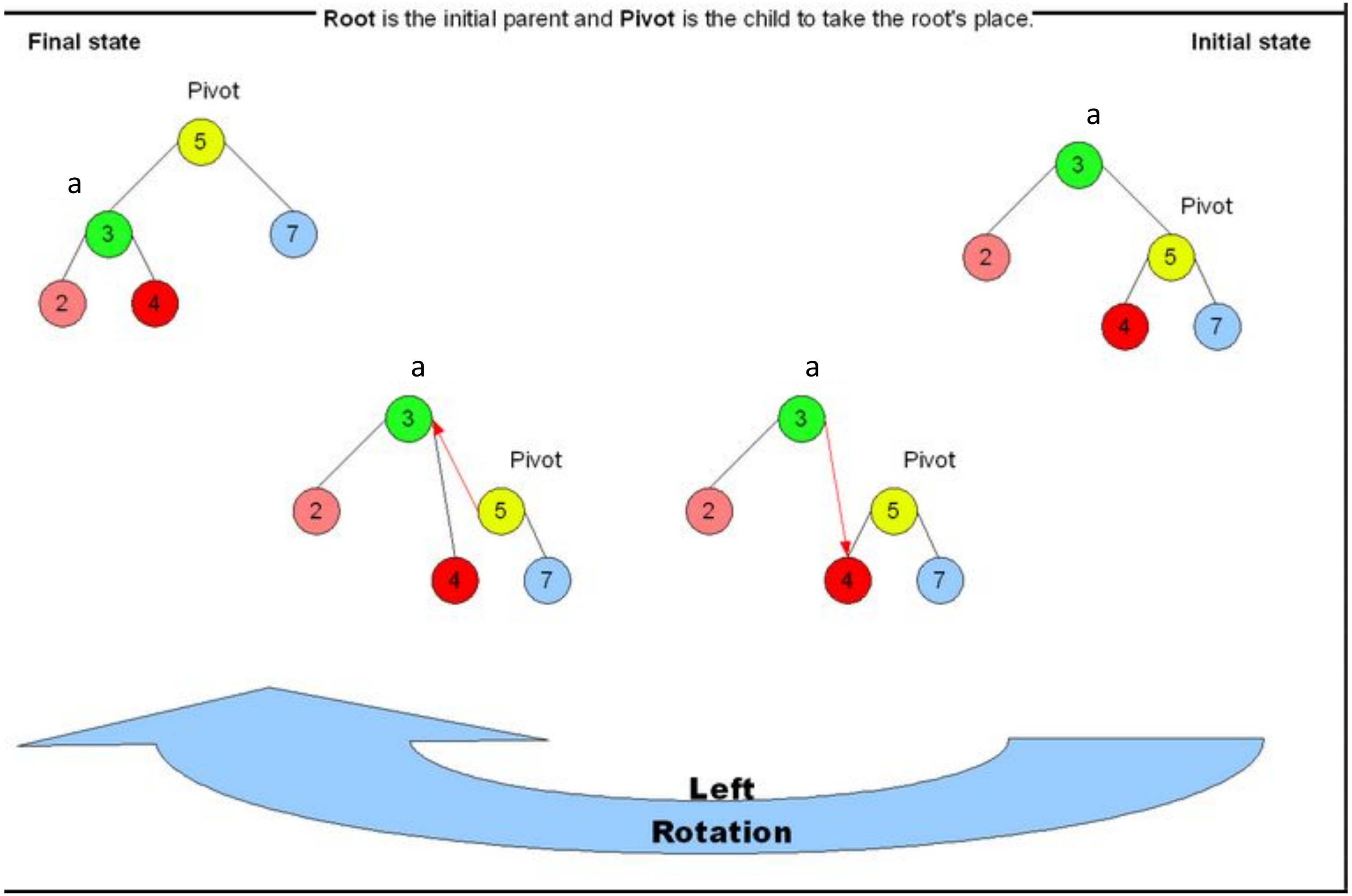
RotateRight Algorithm



1. `a.left = Pivot.right`

2. `Pivot.right = a`

RotateLeft Algorithm



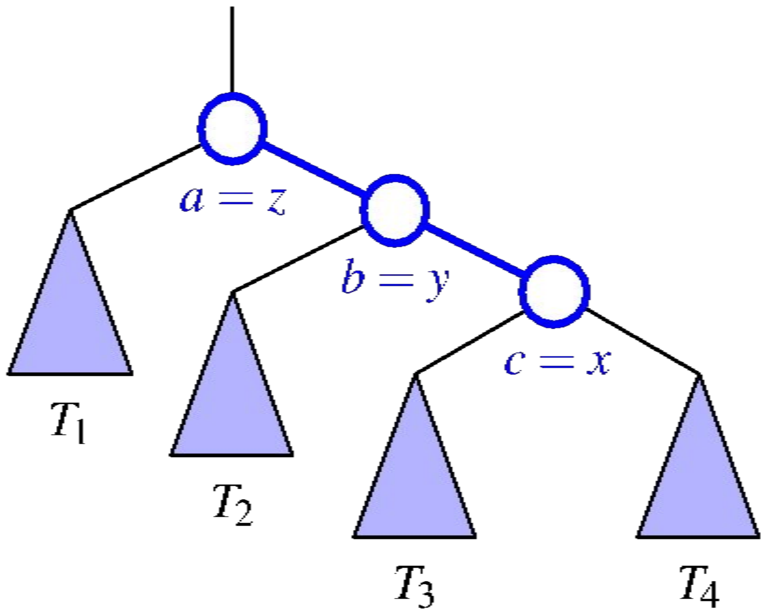
1. `Root.right = Pivot.left`

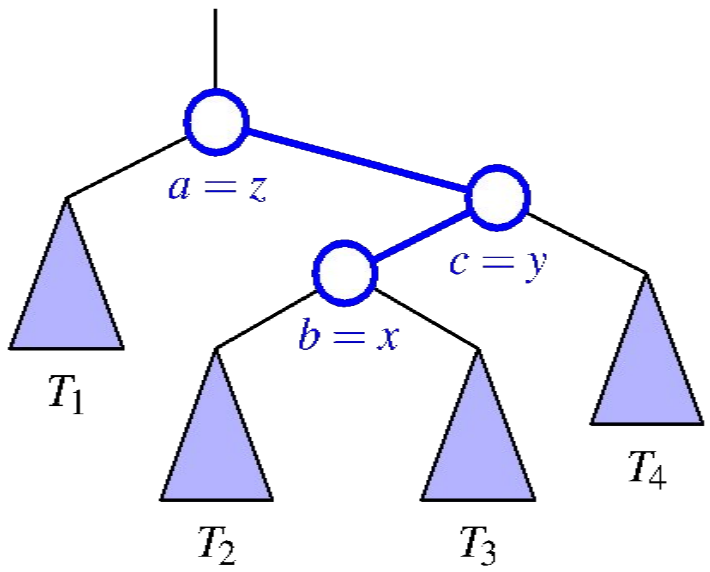
2. `Pivot.left = root`

Double Rotation

Sometimes a single rotation is not enough to restore balance

Double Rotation

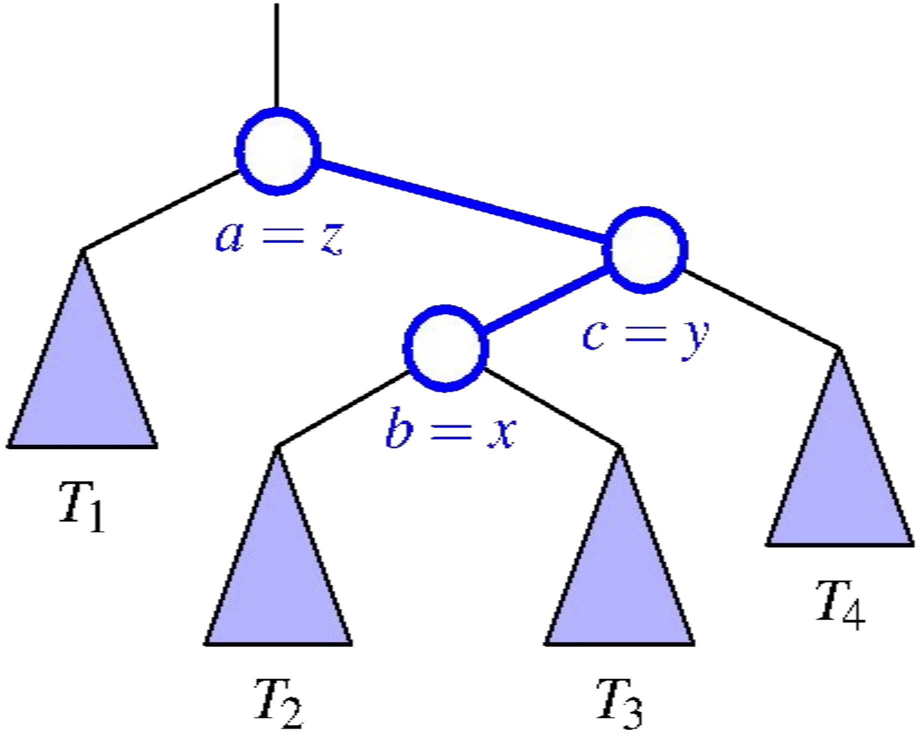




Right child of a is too heavy.. because
Right subtree of b is too heavy..
Single Left rotation on the root needed

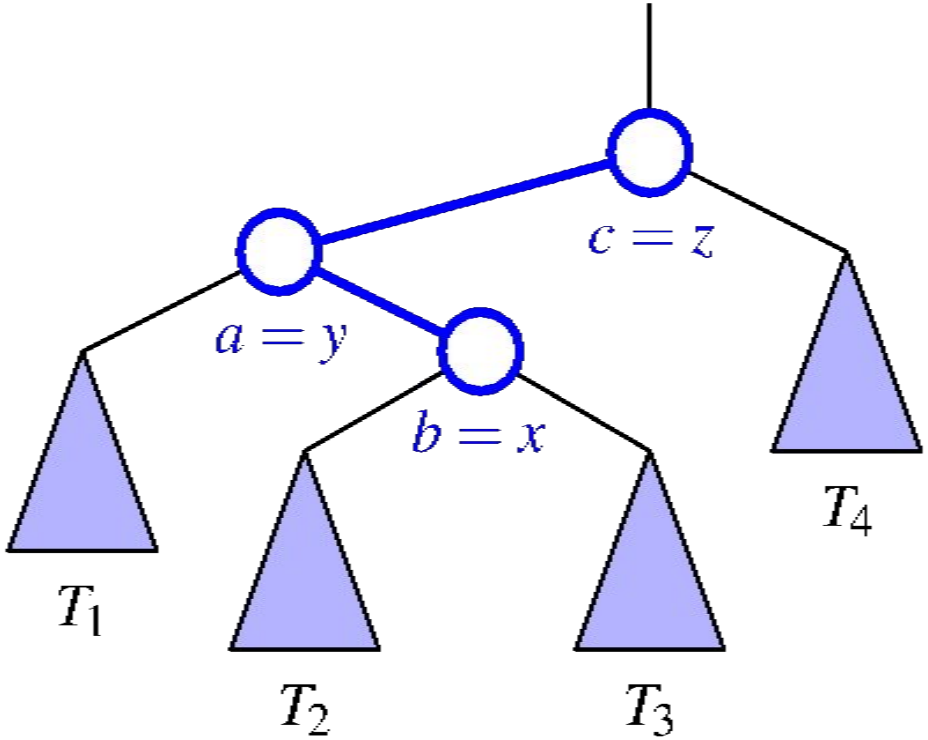
Right child of a is too heavy... because
Left subtree of c is too heavy
Is a single rotation enough?

Double Rotation

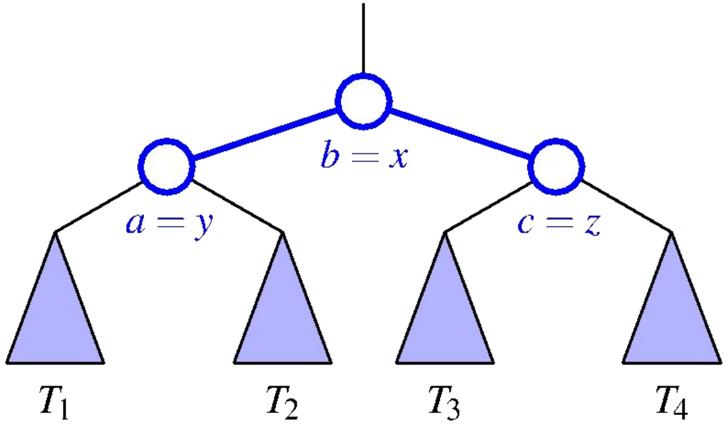


1. **Rotate Right** at c because right subtree of root is too heavy
2. **Rotate Left** at the root (a)

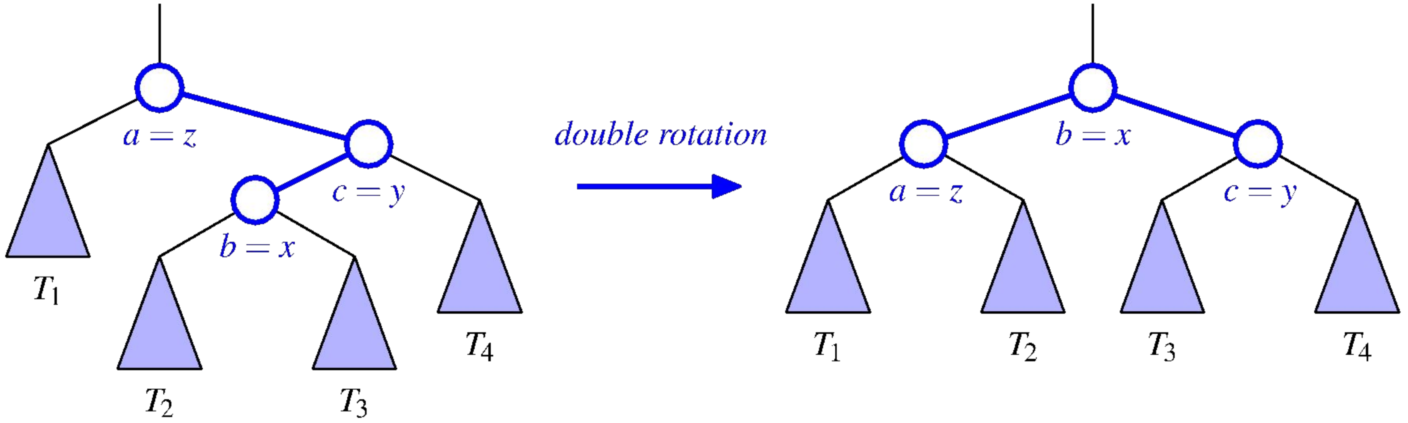
Double Rotation Example 2:



1. **Rotate Left** at `a` because right subtree of root is too heavy
2. **Rotate right** at the root (c)

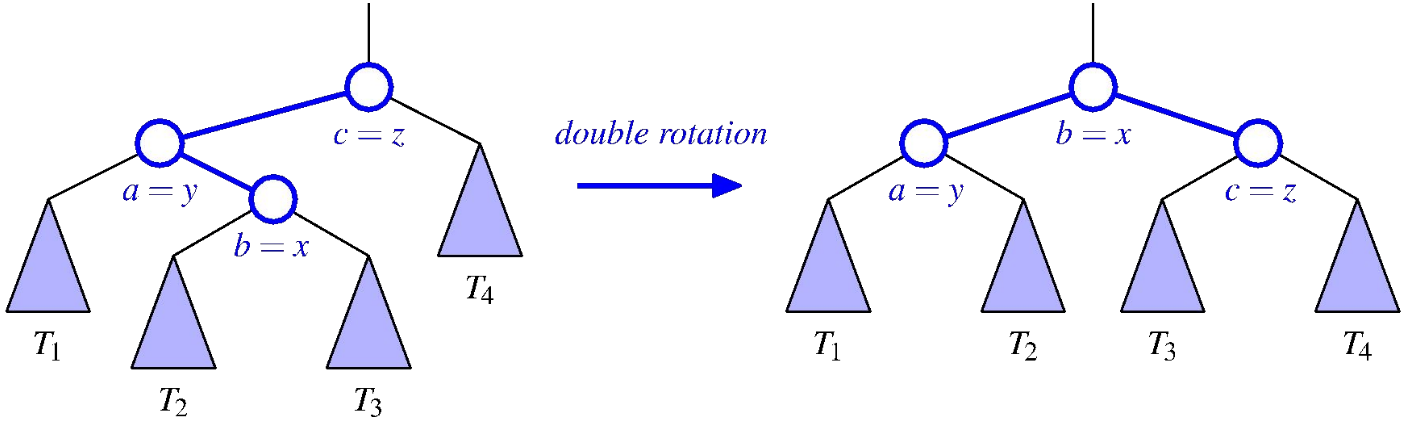


Double Rotations



Right subtree is too heavy because of **left** subtree of c

1. Rotate Right about c
2. Rotate Left about a

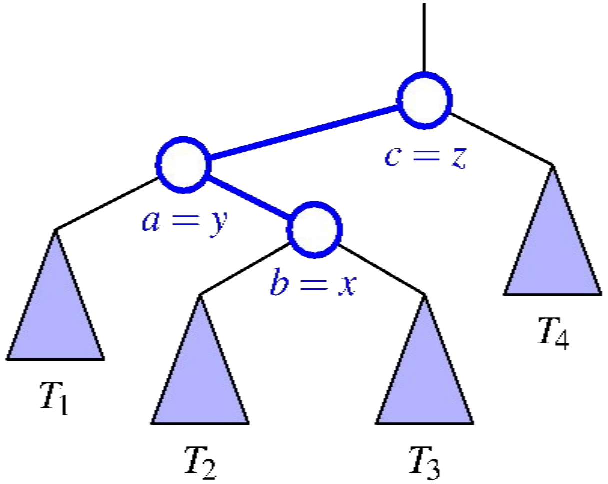


Left subtree is too heavy because of **right** subtree of a

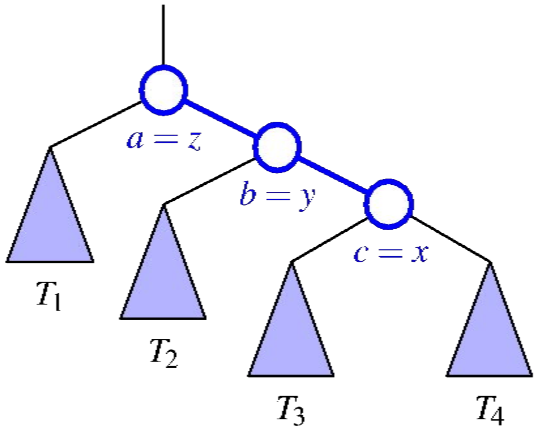
1. Rotate Left about a
2. Rotate Right about c

Double Rotation

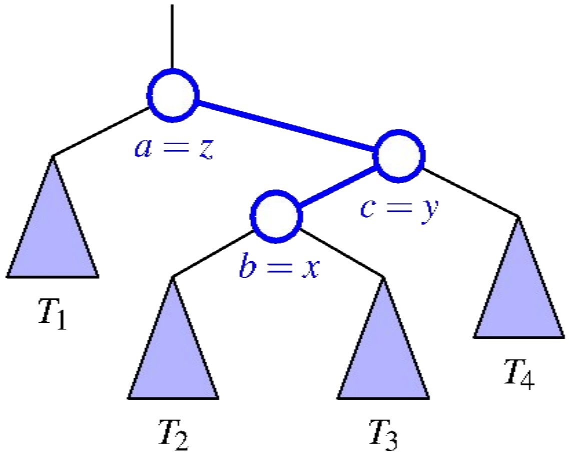
When do we need a double rotation vs a single rotation?



Double rotation



Single rotation



Double rotation

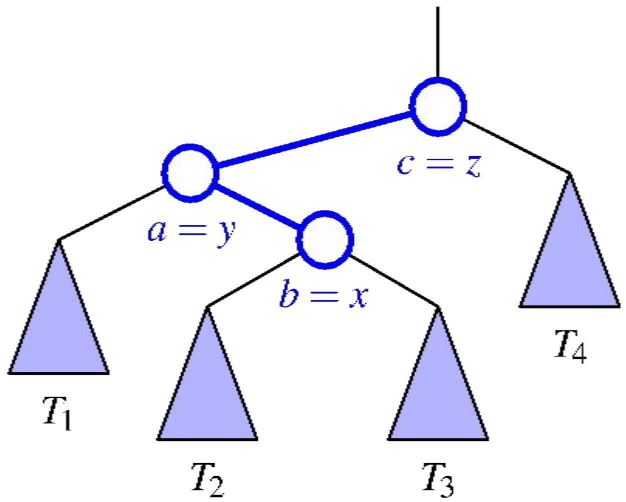
Look for zig-zag pattern!

Double rotation

When do we need a double rotation?

Left subtree is too heavy on the right side

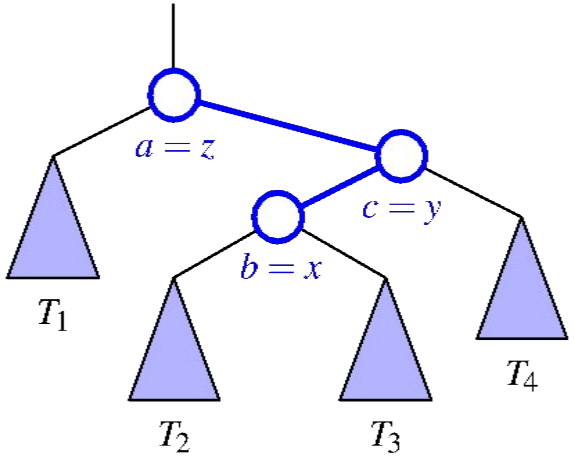
`rotateLeftRight`



OR

Right subtree is too heavy on the left side

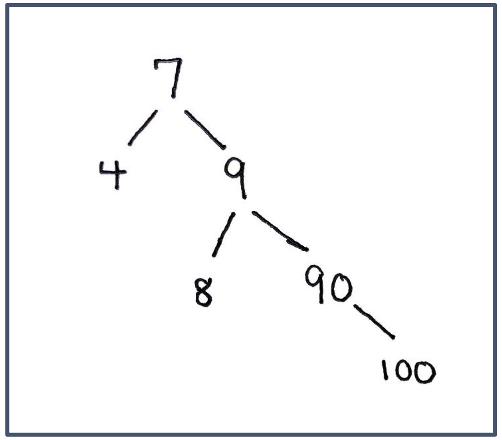
`rotateRightLeft`

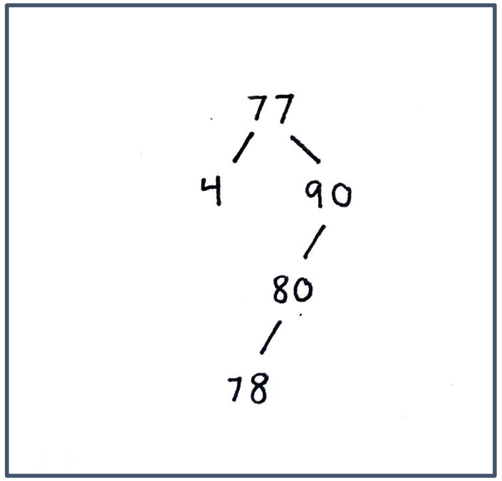


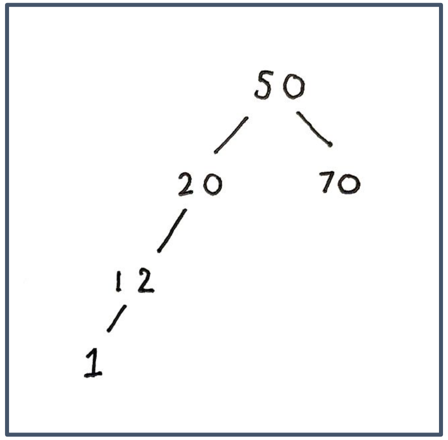
Double Rotation Code

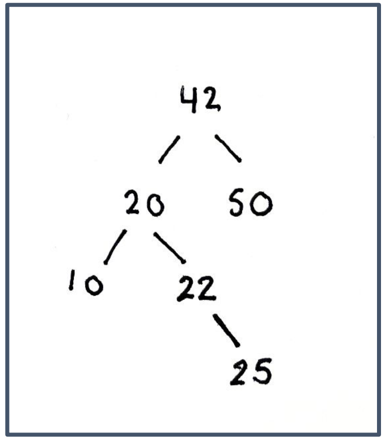
```
def rotateLeftRight(n)
    n.left = rotateLeft(n.left);
    n = rotateRight(n);

def rotateRightLeft(n)
    n.right = rotateRight(n.right);
    n = rotateLeft(n);
```

Examples - which way should I rotate?









rotateLeft

rotateRightLeft

rotateRight

rotateLeftRight

AVL Trees

AVL Trees

- "*self balancing* binary search tree"
- For every internal node, **the heights of the two children differ by at most 1**
- does rotations upon insert/removal if necessary

AVL Height

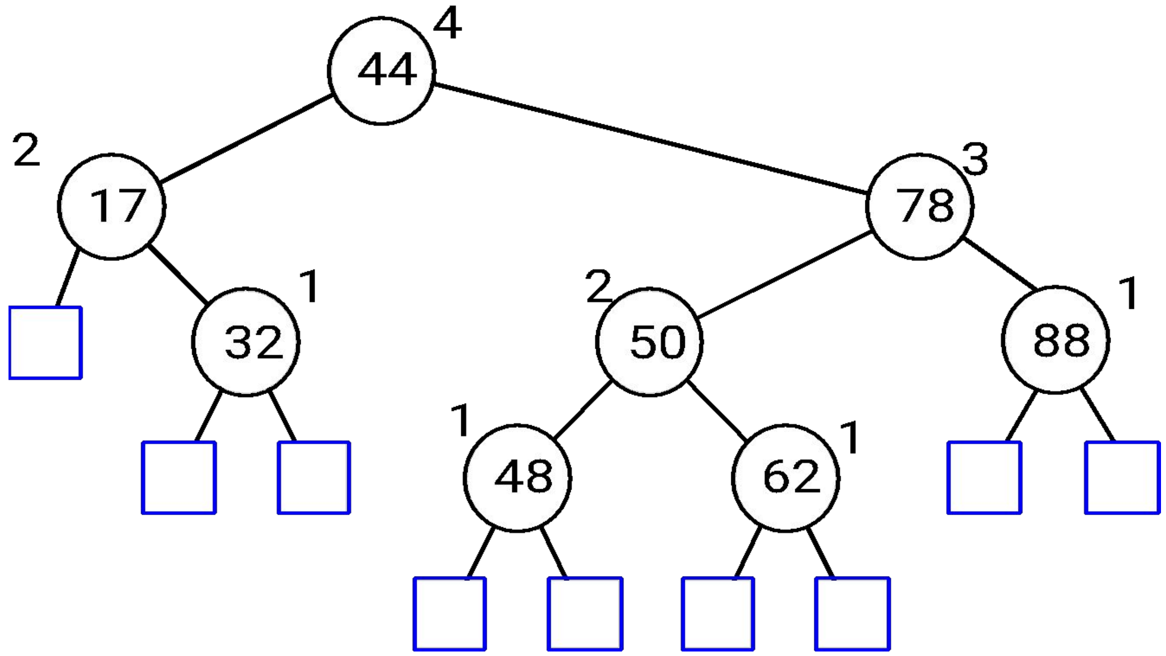
- We keep track of the height of each node as a field for quick access

- The height of an AVL tree is logn
    - Always balanced

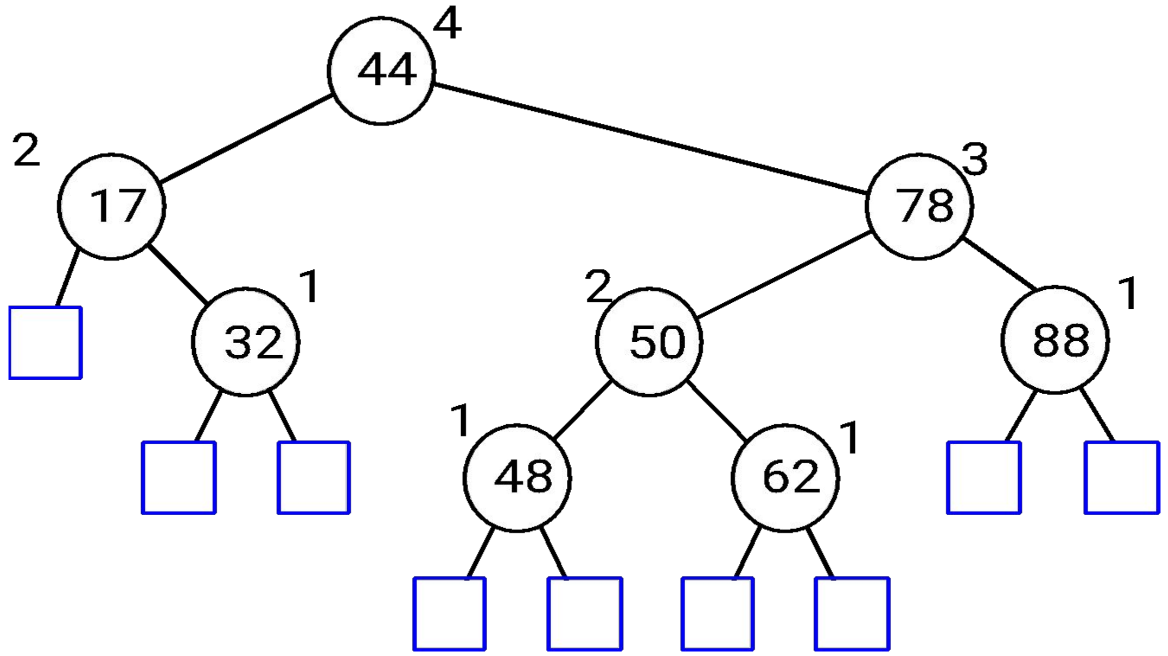
Insertion

AVL Tree Example

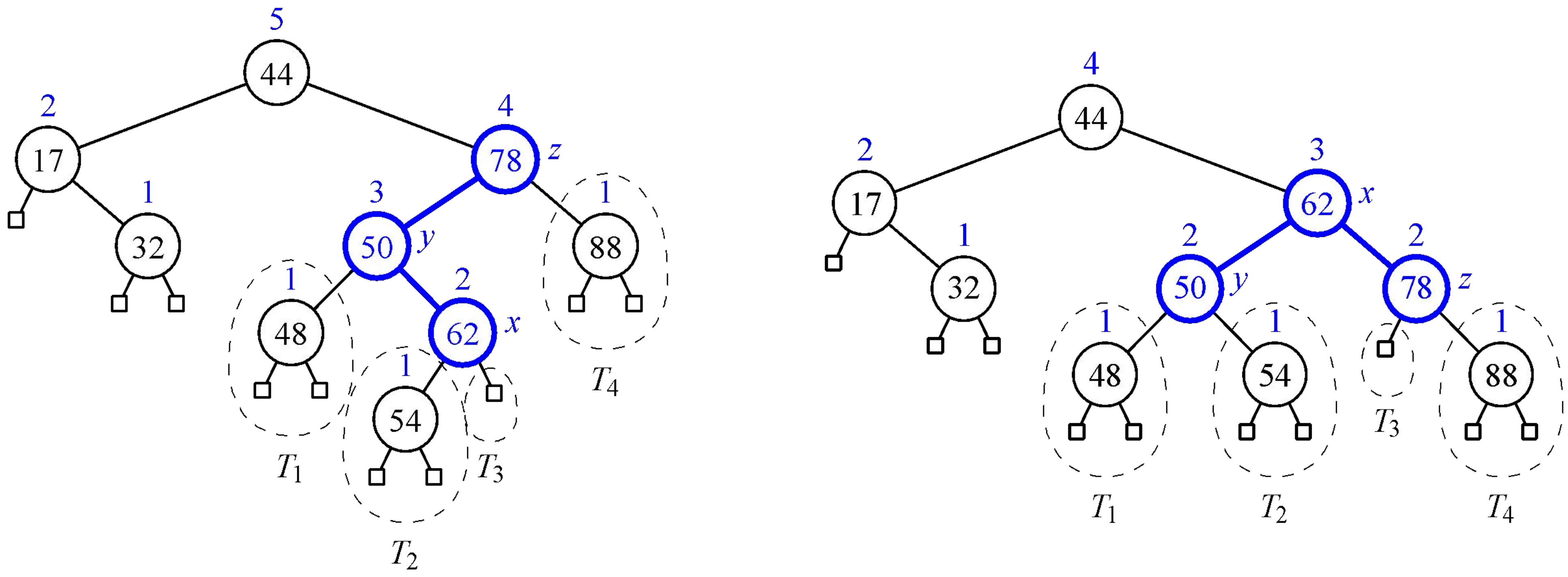
- leaves are sentinels and have height 0



Insert 54



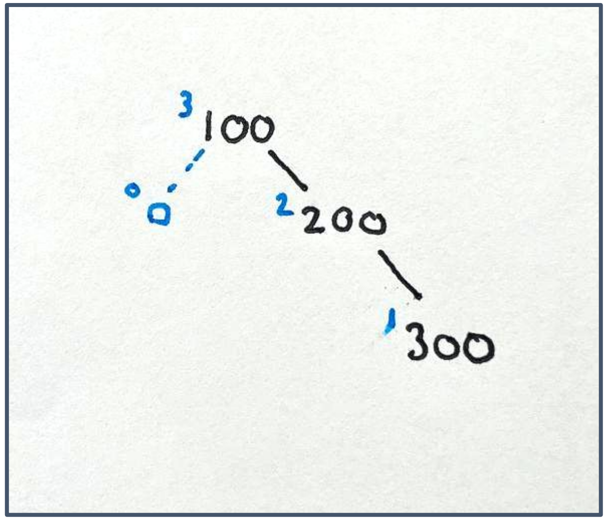
Insertion (54)

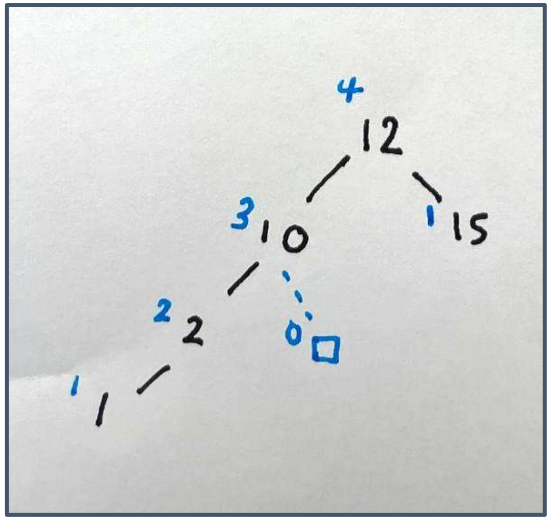


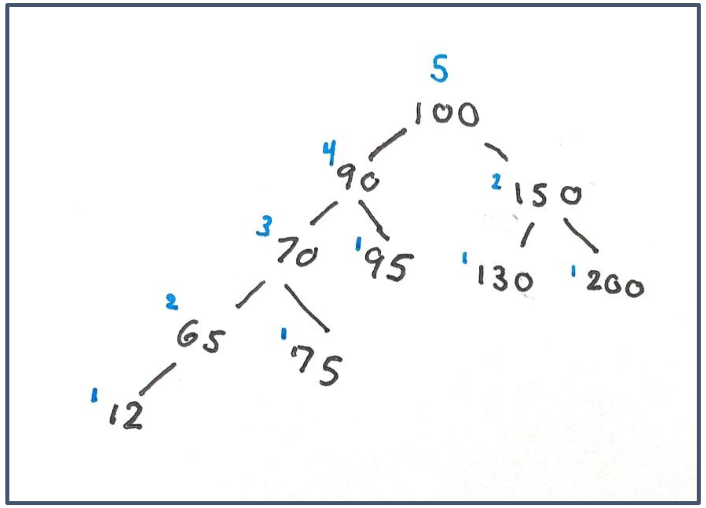
New node always has height 1

Parent may change height

Which node do we “rebalance over”?







lowest subtree with diff(heights) > 1

Exercise

- Create an AVL tree by inserting the nodes in this order:
  - M, N, O, L, K, Q, P, H, I, A

AVL Animation

Rebalance Algorithm

```
If left.height > right.height + 1:
    if (left.right.height > left.left.height)  //double rotate
        rotateLeftRight(n)
    else:
        rotateRight(n)

else if right.height > left.height + 1:
    if (right.left.height > right.right.height) //double rotate
        rotateRightLeft(n)
    else:
        rotateLeft(n)
```

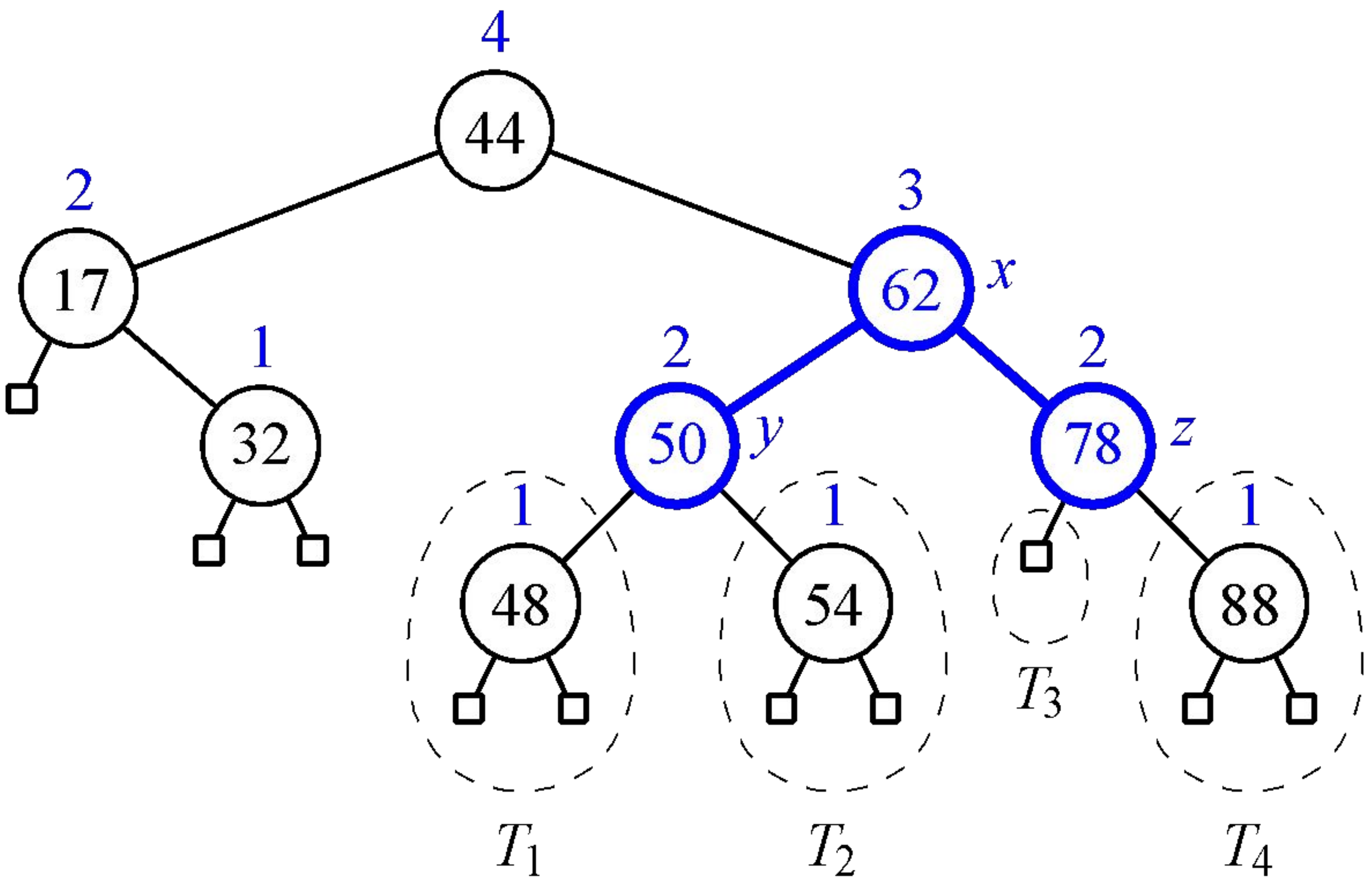
Runtime Complexity:

Insertion (plus rotation)

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

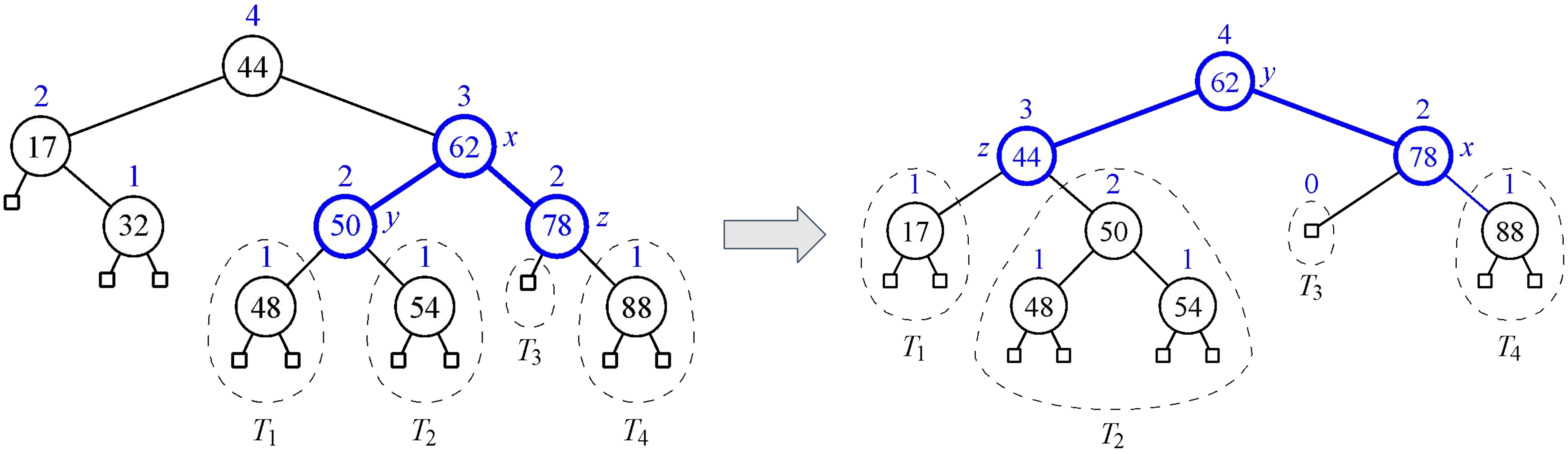
Deletion

Delete Example 1: 32

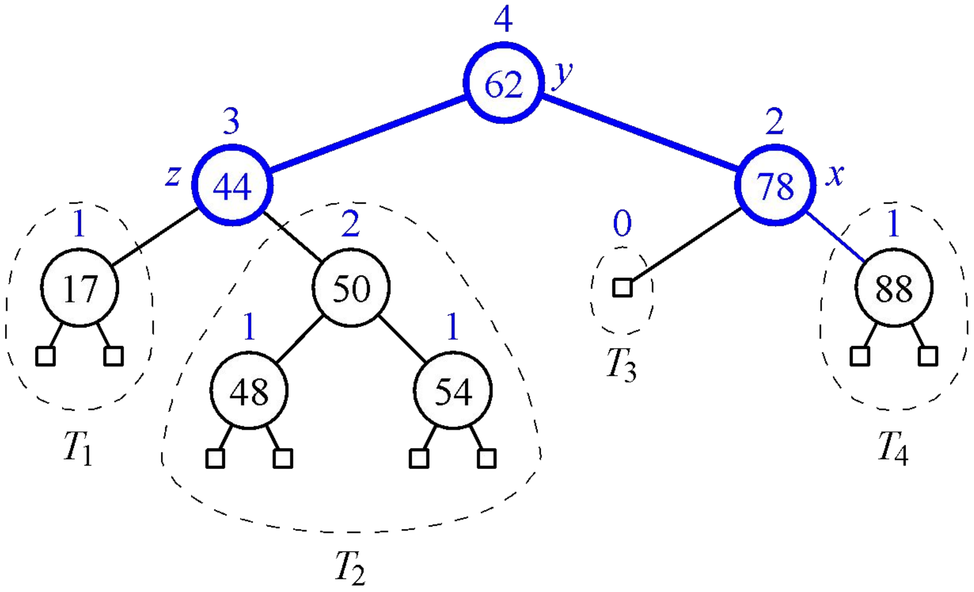


Delete Example 1: 32

rotateLeft

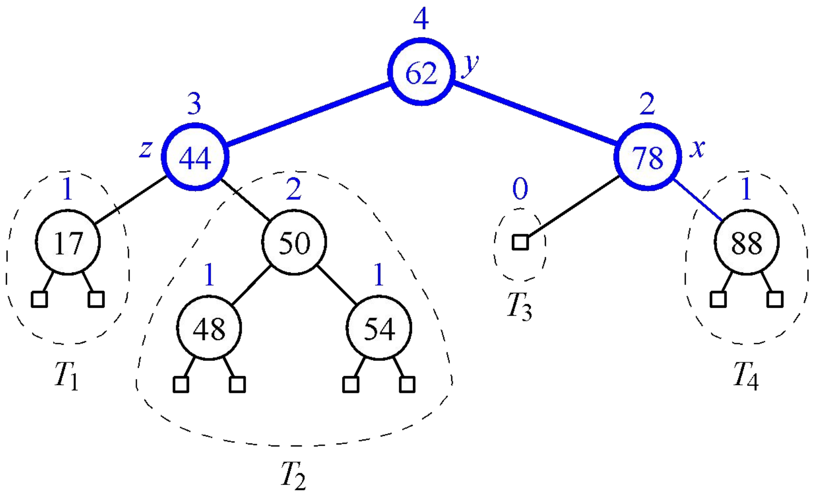


Delete Example 2: 88

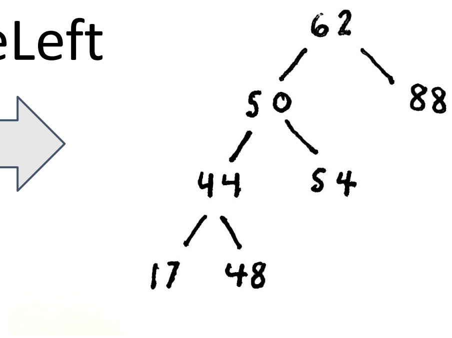


Delete Example 2: 88

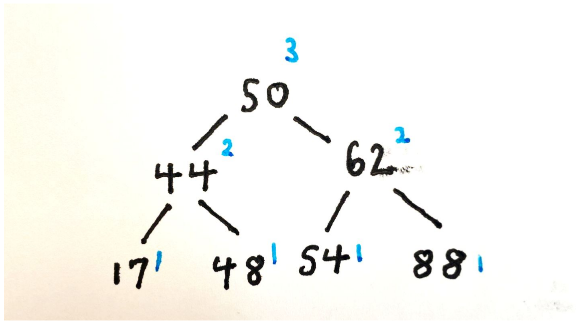
rotateLeftRight



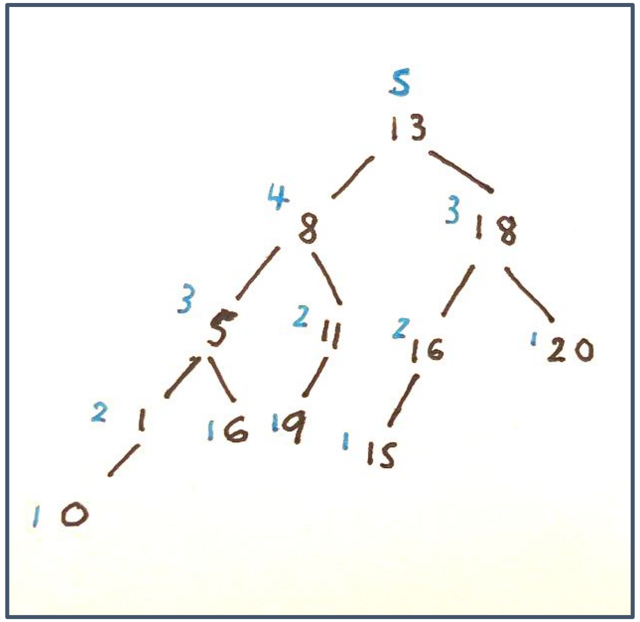
rotateLeft



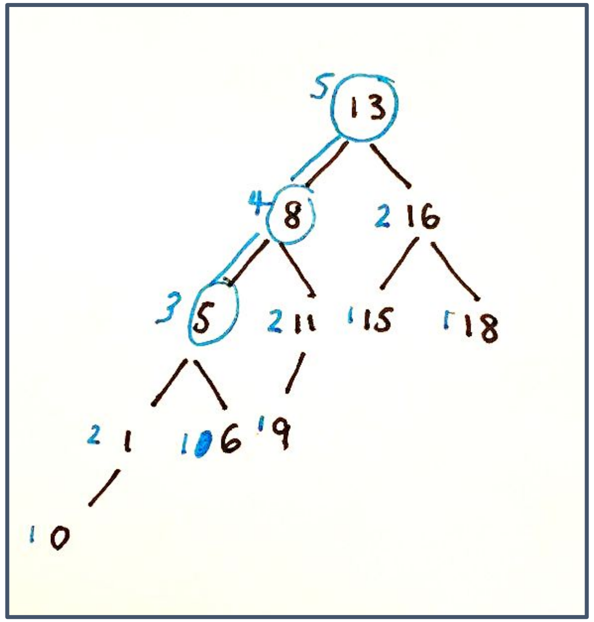
rotateRight



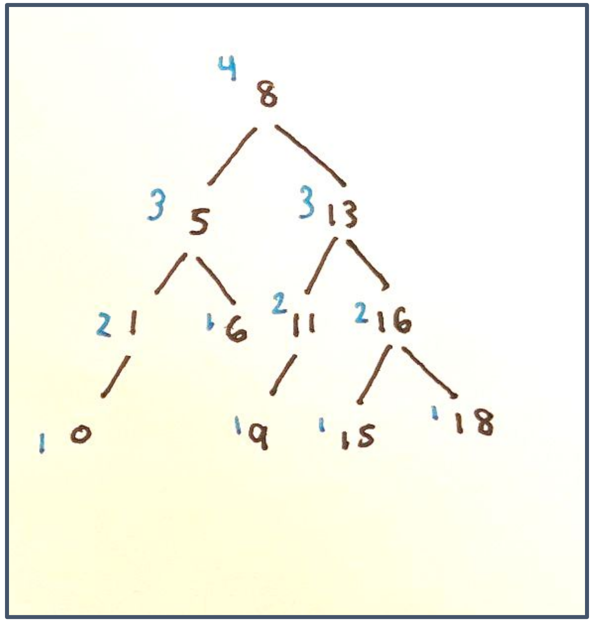
Delete Example 3: 20



rotateRight



rotateRight



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

Performance of BSTs

Runtime complexity:

search?

- BST:
  - O(n)
- AVL:
  - O(logn)

Performance of BSTs

Runtime complexity:

insert?

- BST:
  - O(n)
- AVL:
  - O(logn)

Performance of BSTs

Runtime complexity:

remove?
- BST:
  - O(n)
- AVL:
  - O(logn)

Splay Trees

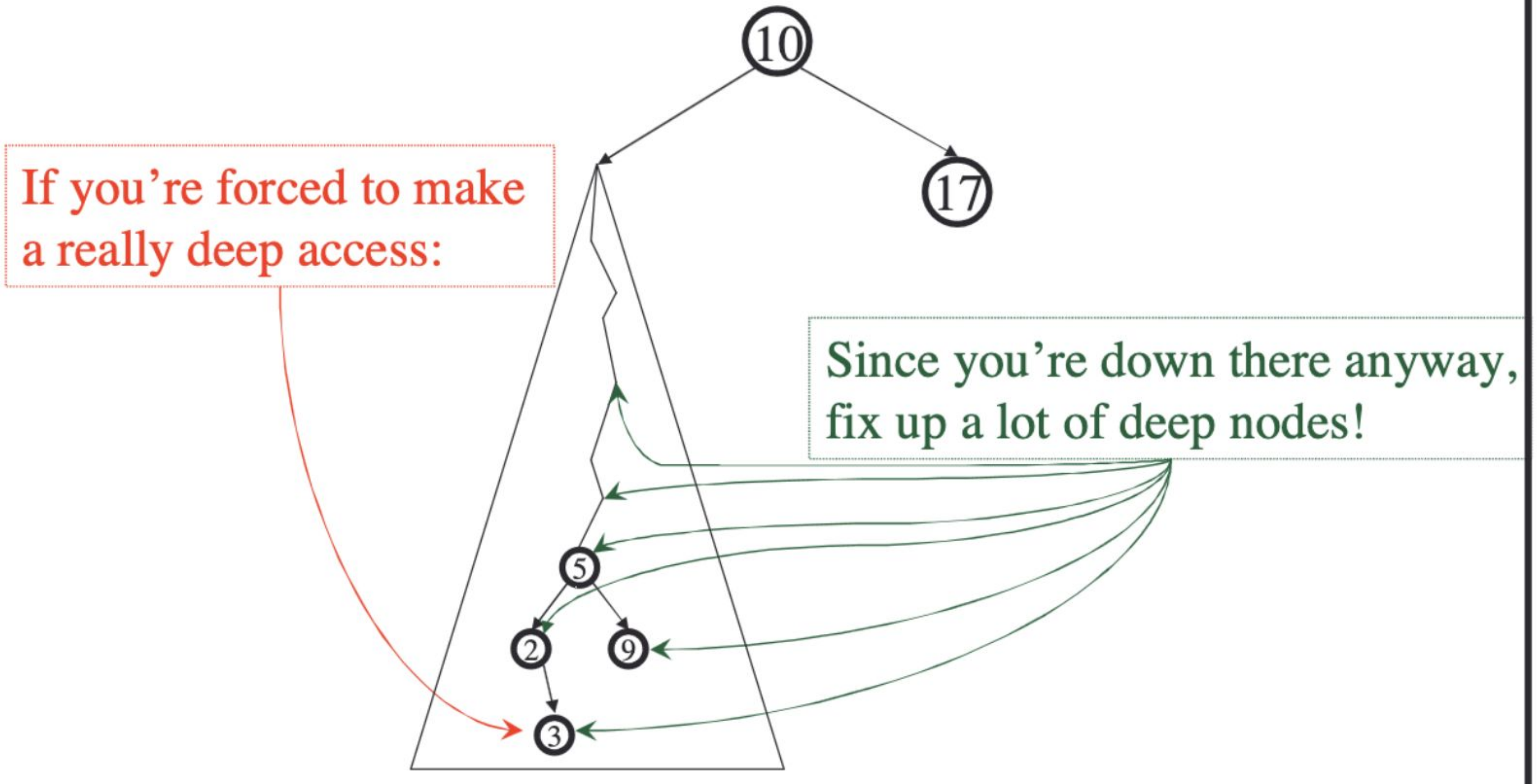
Splay Trees

- 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*"

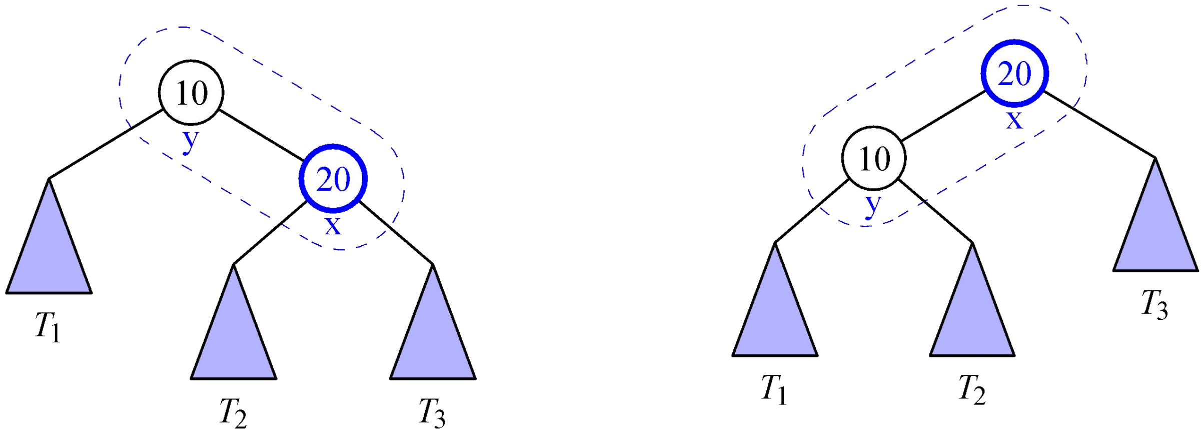
The Splay Tree Idea



Splaying

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

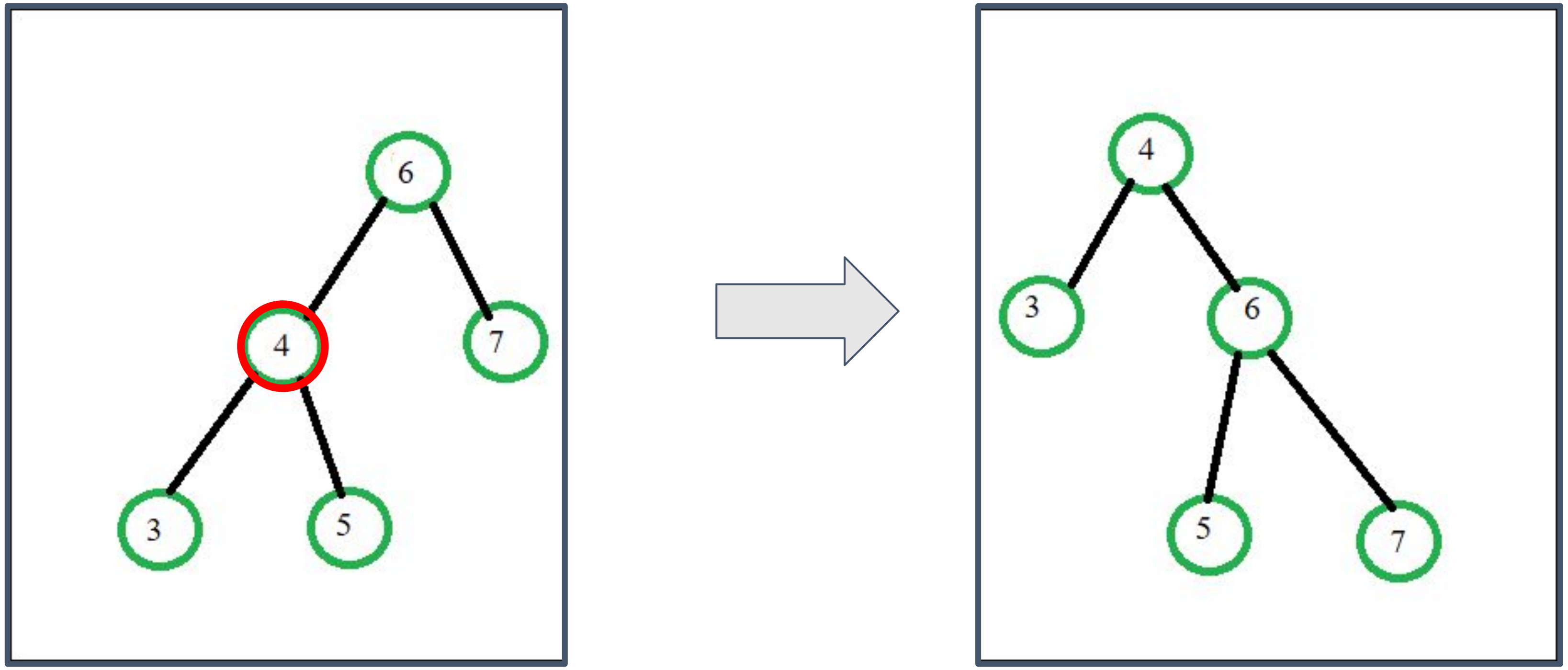
- **zig**
  a. accessed node becomes root of subtree
  b. parent becomes child



before after

Splaying - Zig

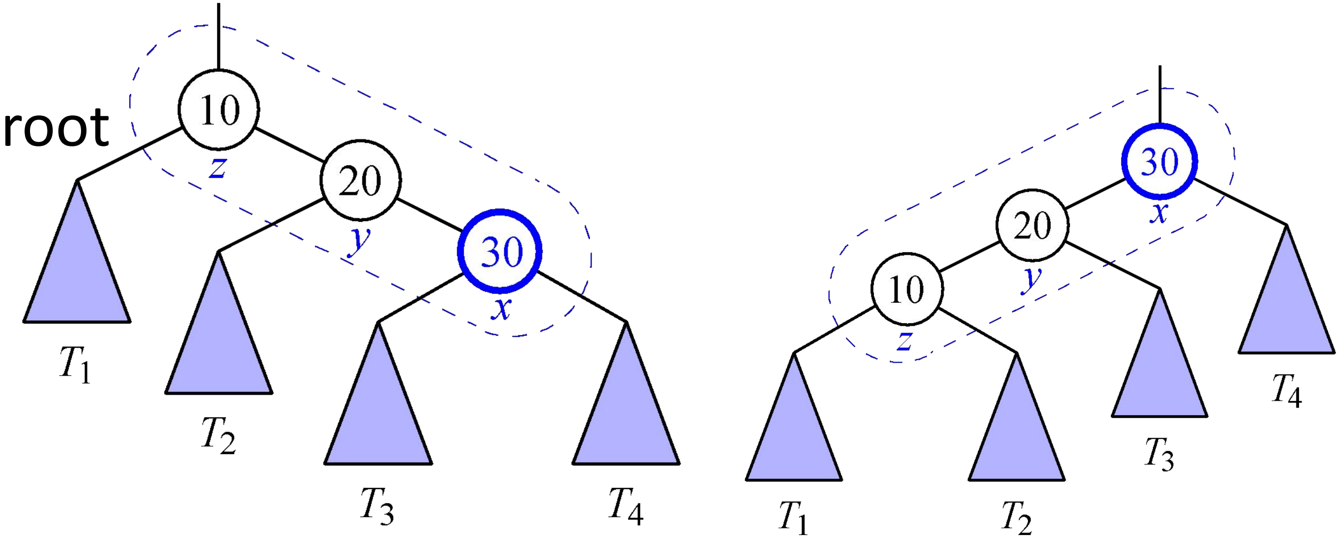
- **zig**
  a. accessed node becomes root of subtree
  b. parent becomes child



Splaying: Zig-Zig

zig-zig:

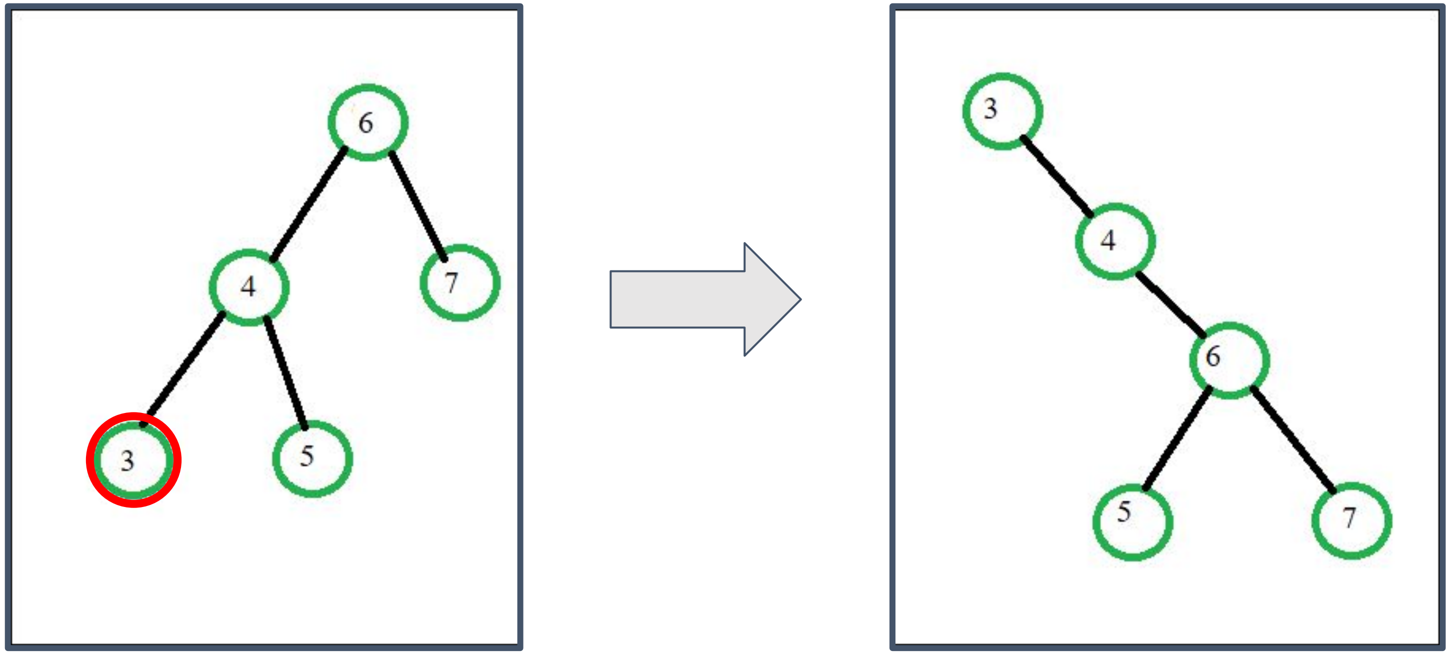
- step 1: **zig**
  a. accessed node's parent (y) becomes root
  b. parent becomes child
- step 2: **zig**
  a. accessed node (x) becomes root
  b. parent becomes child



before after

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

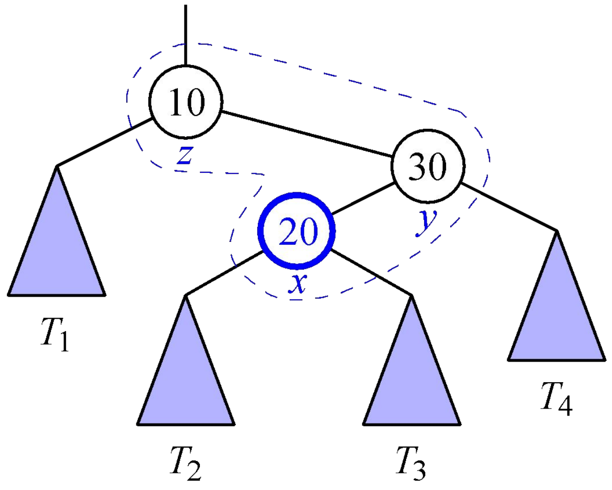


Splaying: Zig-Zag

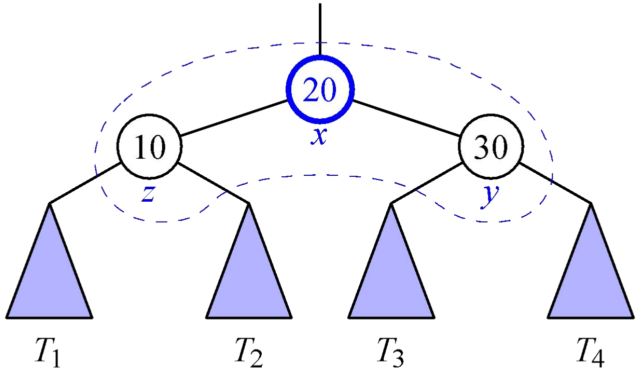
zig-zag:

- 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**

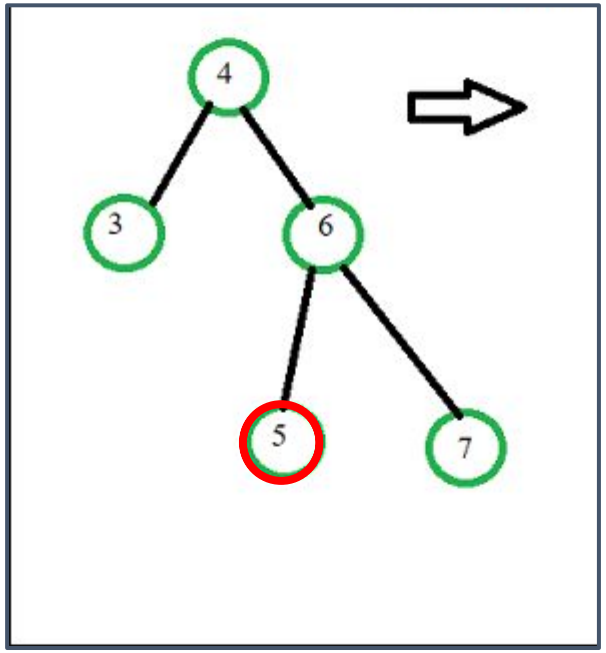


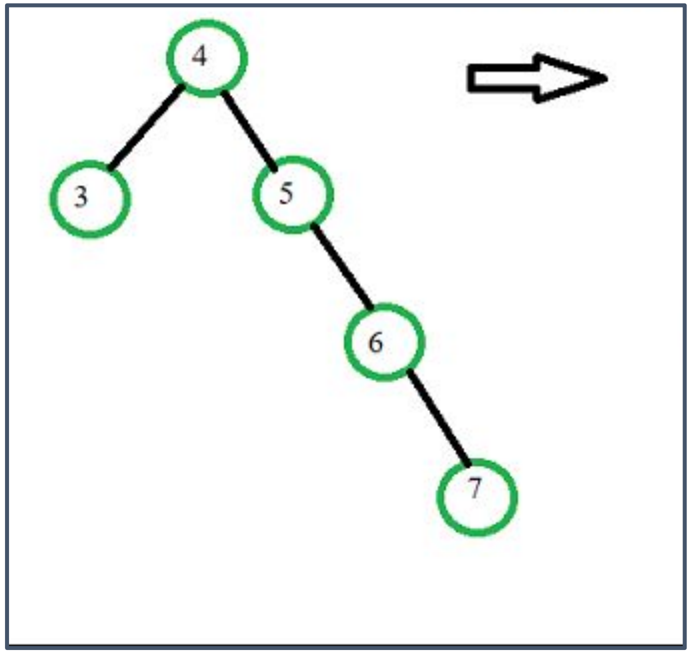
before

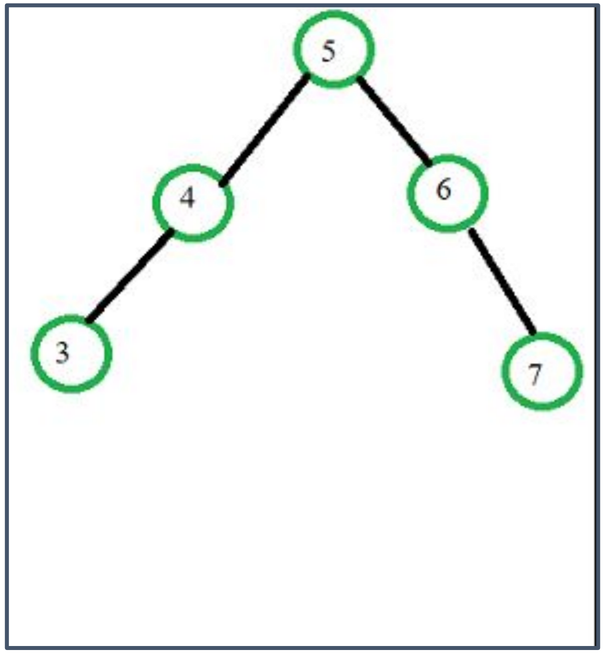


after

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

2. **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

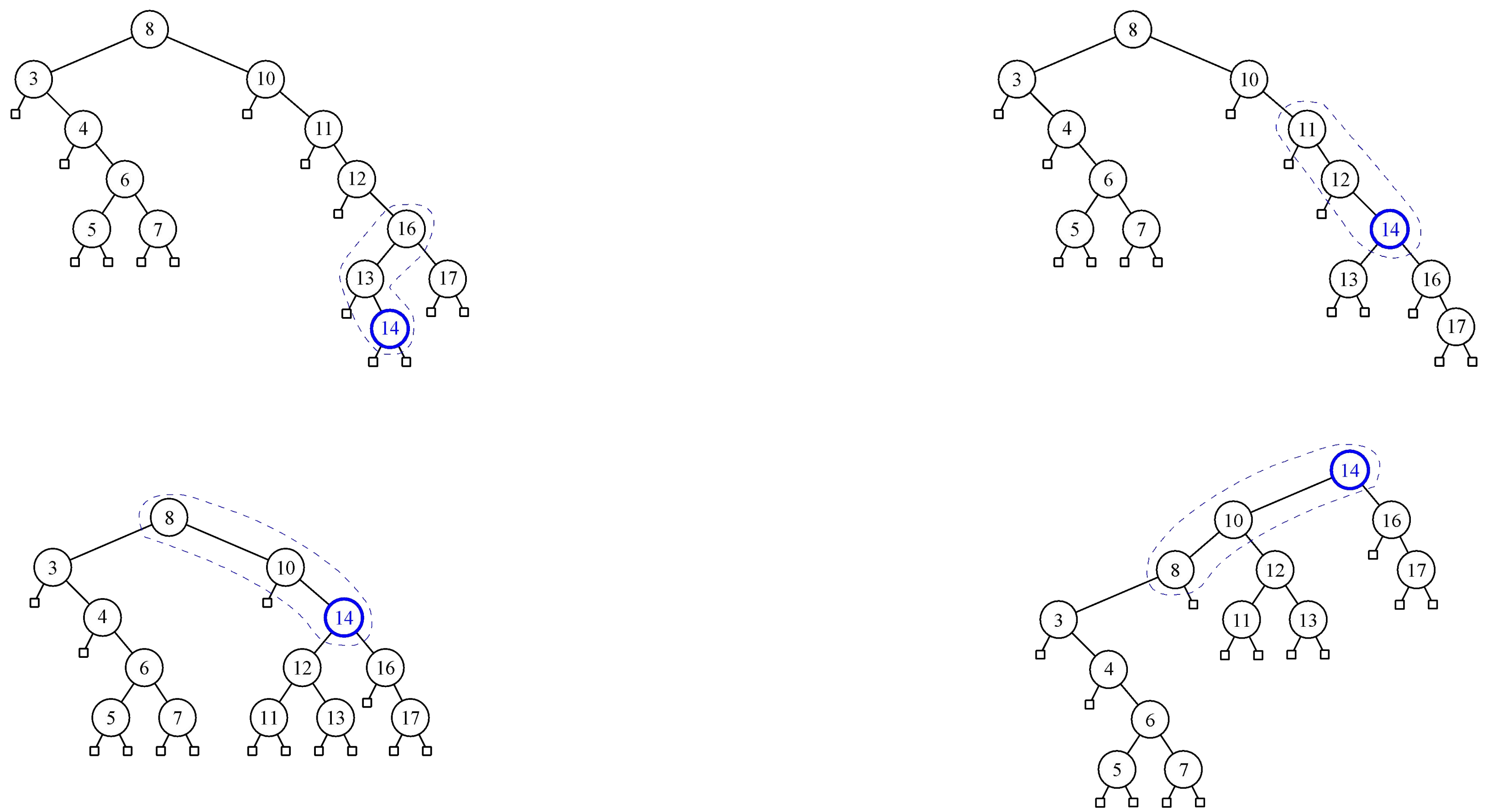
3. **Zig-Zag:** one of x and y is a right child and the other is a left child
   a. Analogous to double rotations in AVLs

Splaying

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

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

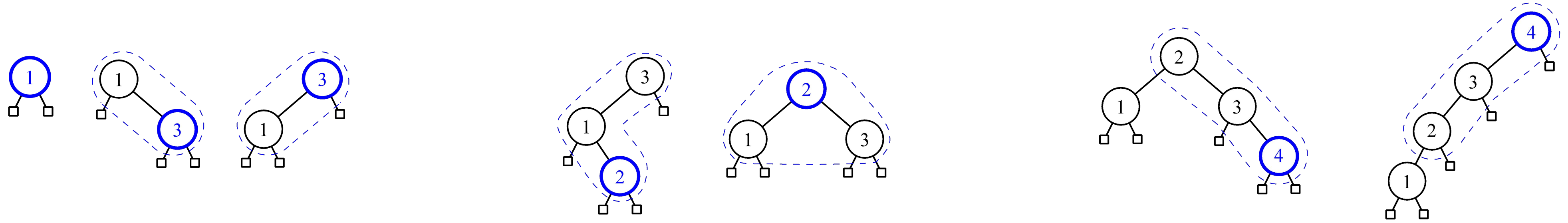
Example - insert(14)



When/what to Splay

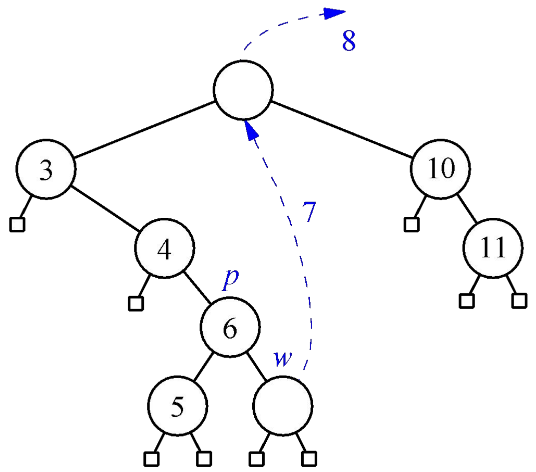
on search for x: if x is found, splay x. else splay x's parent

on insert x: splay x after insertion

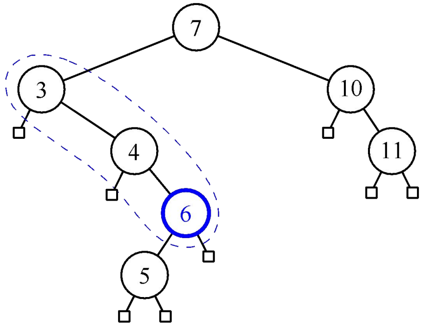


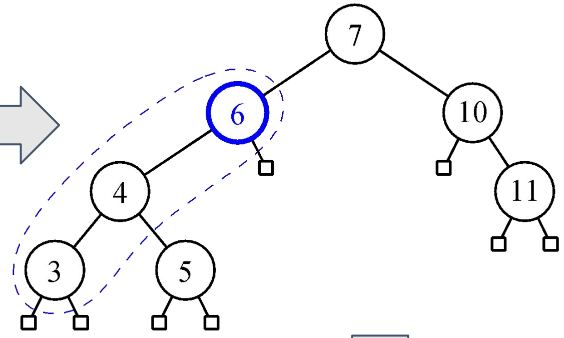
on remove x: splay parent of removed leaf node

Deletion: remove(8)

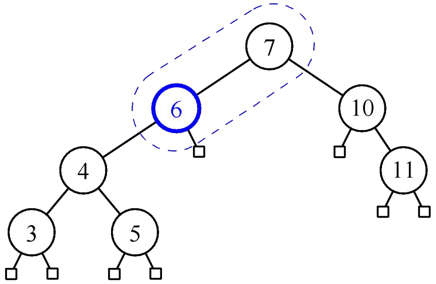


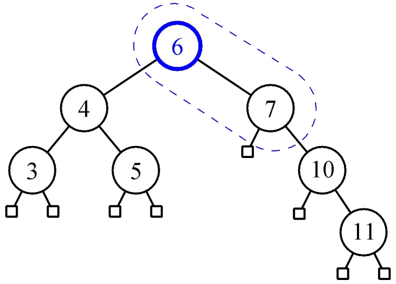
splay 6 (parent of removed node)





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





Analysis of Splaying

Runtime of restructuring operations:

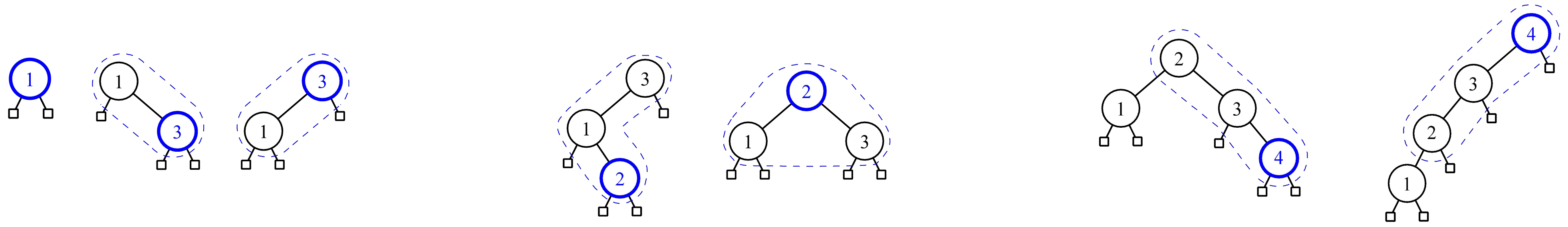
1. zig
   a. O(1)
2. zig-zig
   a. O(1)
3. zig-zag
   a. O(1)

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)