CS151 Intro to Data Structures

Balanced Search Trees, AVL Trees

Announcements

HW 7 and Lab9 (Hash Maps) due last night

LAST HOMEWORK - HW8 (AVL Trees) due Dec 15th

Outline

Warmup

Sorting review

Balanced BSTs

Choosing the Right Data Structure for Library Management

You are implementing a system to track and manage a library's collection of books. Each book has a unique ISBN number, and the system needs to efficiently support the following operations:

- **Add a book**: Insert a new book using its ISBN number as the key.
- **Remove a book**: Delete a book from the system by its ISBN number.
- **Find a book**: Retrieve details about a book by its ISBN number.
- **Get all books in sorted order**: Return a list of all books, sorted by their ISBN numbers.
- **Find the book with the closest higher ISBN**: Given an ISBN, find the next highest ISBN in the collection.

Design a data structure to efficiently support these operations. Justify your choice and explain the time complexity for each operation.

Binary Search Tree Review

What can go wrong?

Complexity?

Search

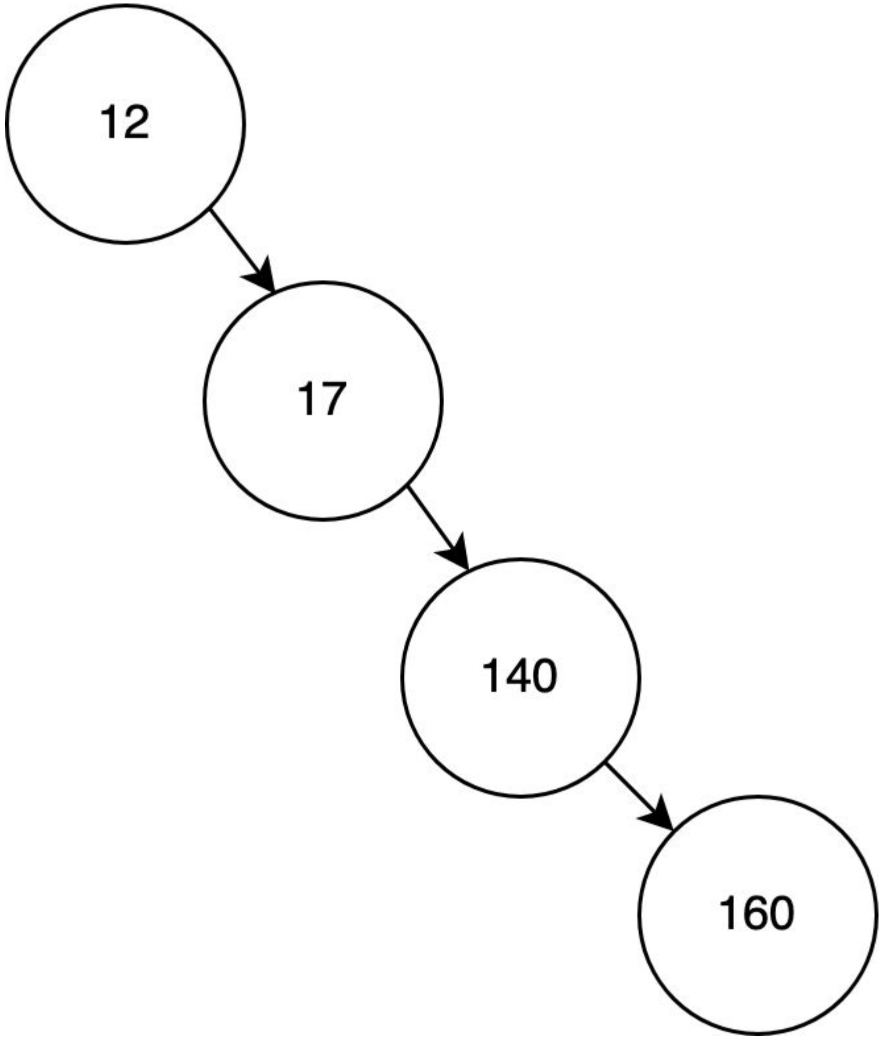
O(n)

Insertion:

O(n)

Deletion:

O(n)



Balanced Binary Trees

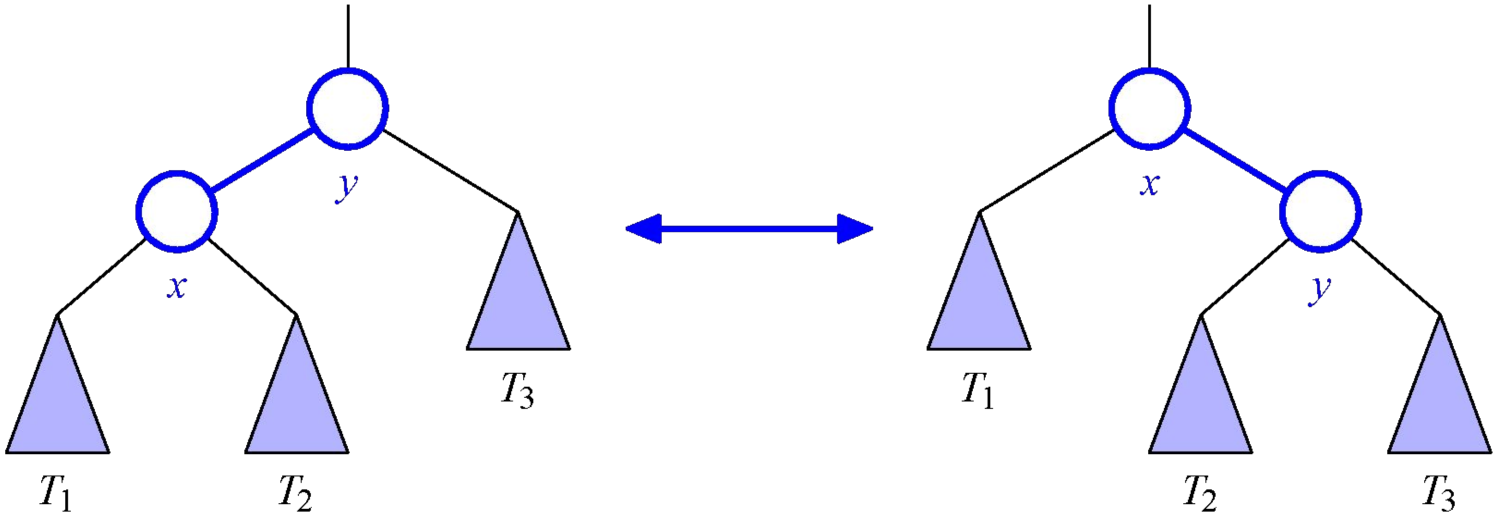
Balanced Binary Trees

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

Rotation Operation

Move a child above its parent and relink subtrees

Maintains BST order

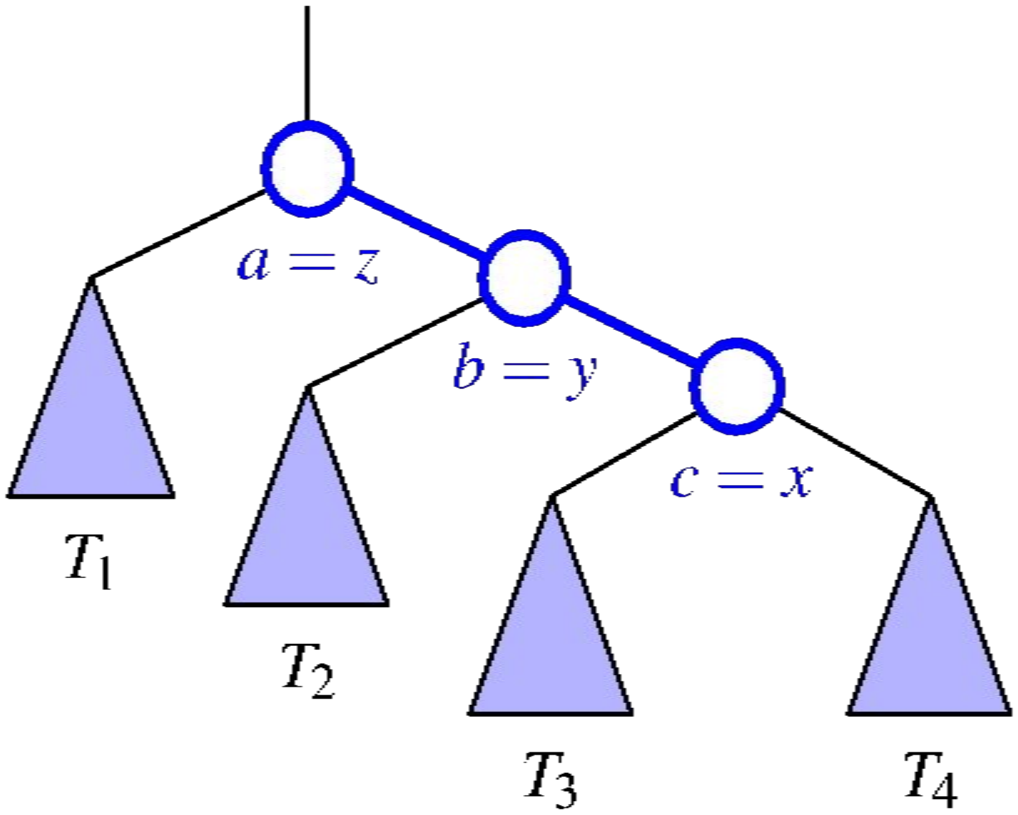


Rotation Operation

- Used to maintain balance

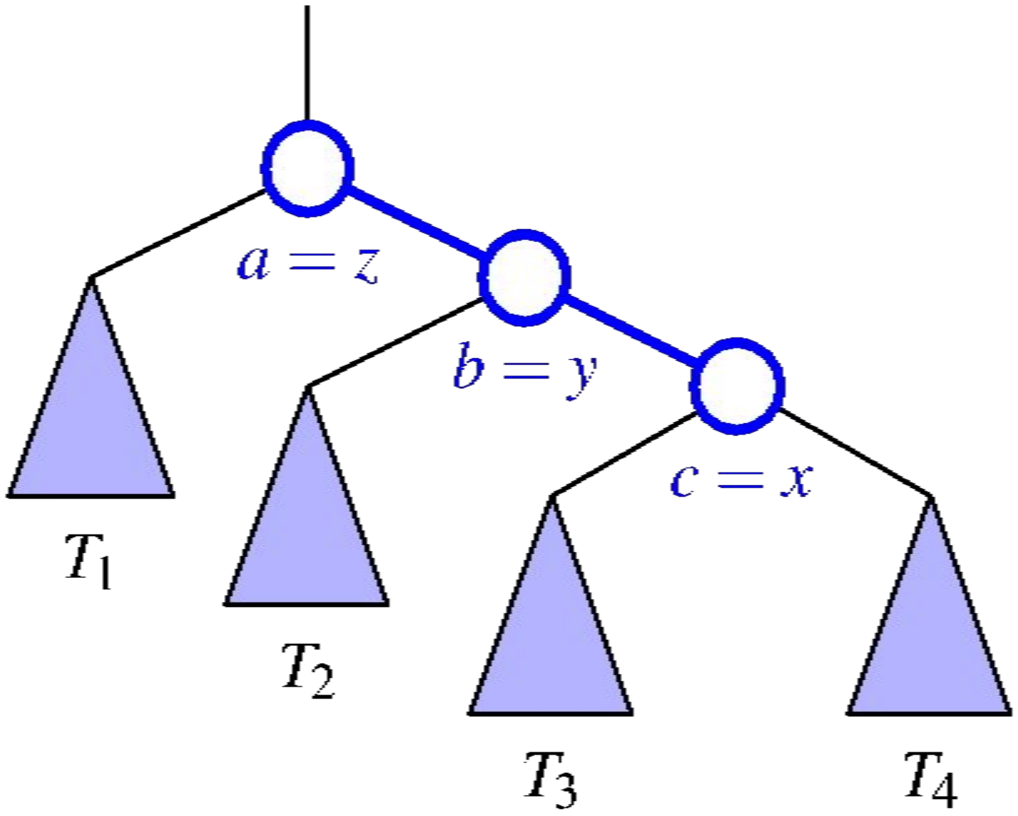
- When should **rotate** be invoked?
  - Difference of heights of left and right subtrees at any node is > 1

Rotation Operation



- Assume heights of subtrees are equal
  - h(T1) = h(T2) = h(T3) = h(T4)
- What is the height of the entire tree?
  - h(T3) + 2
- What is the height of the left subtree of a?
  - h(T1)
- What is the height of the right subtree of a?
  - h(T4) + 2
- Is this tree balanced?

Rotation Operation

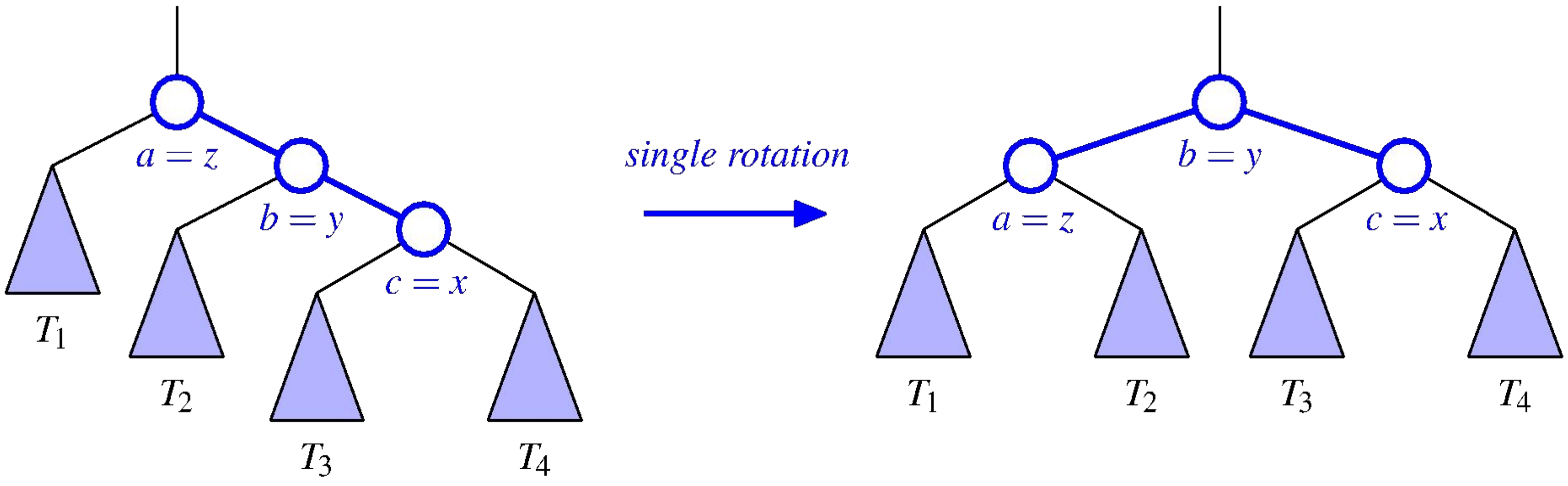


Right subtree is too large!

How can we rotate to fix this?

What should we make the root?

Single Rotation (around z)



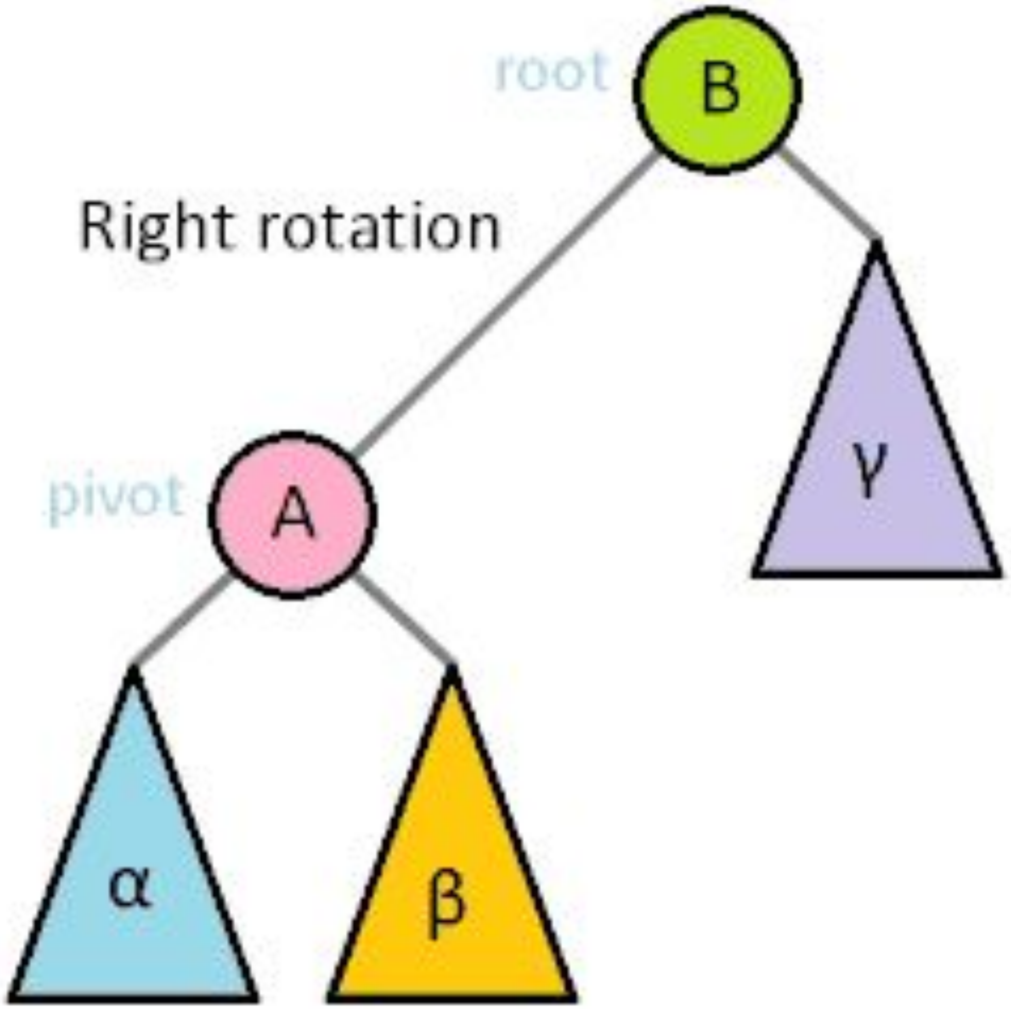
Rotations

Right rotation:

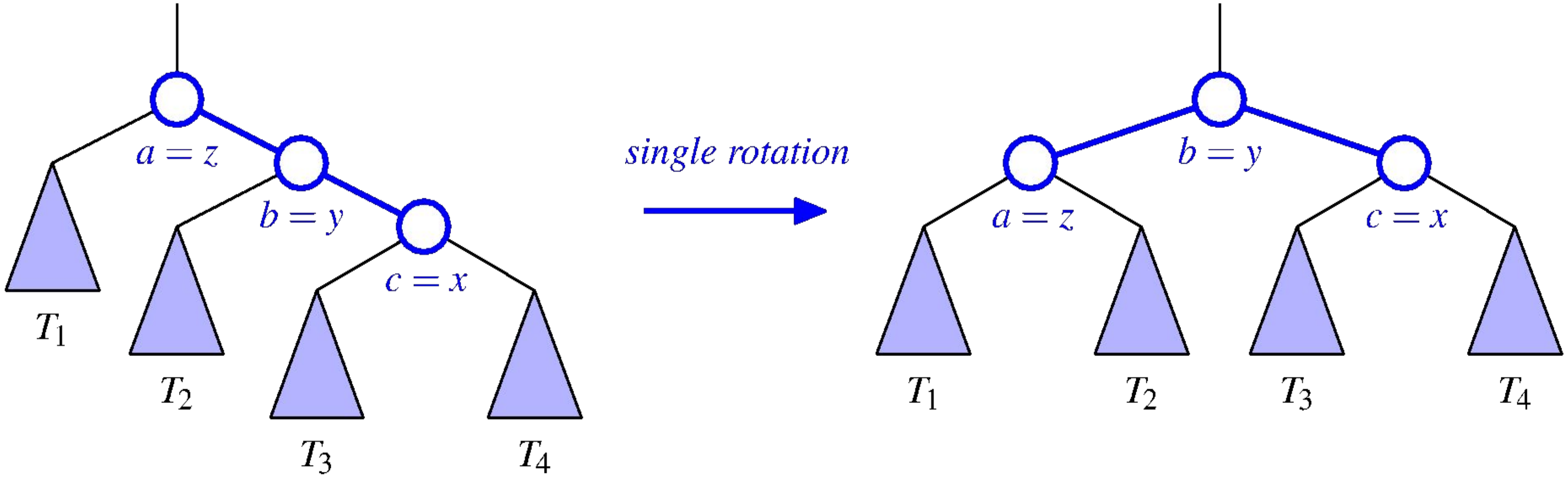
- Performed when left side is heavier
- left child becomes root

Left rotation:

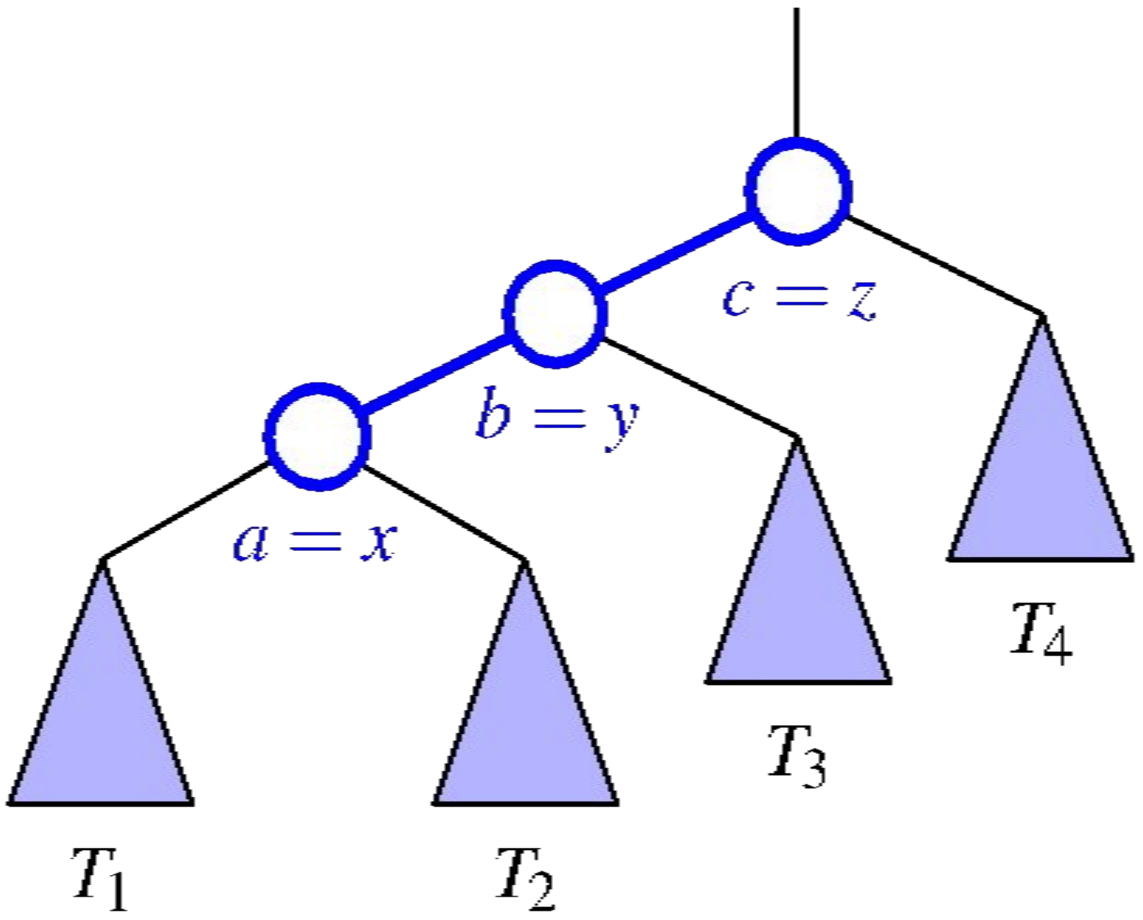
- Performed when right side is heavier
- right child becomes root



Left or Right rotation?



Example 2:

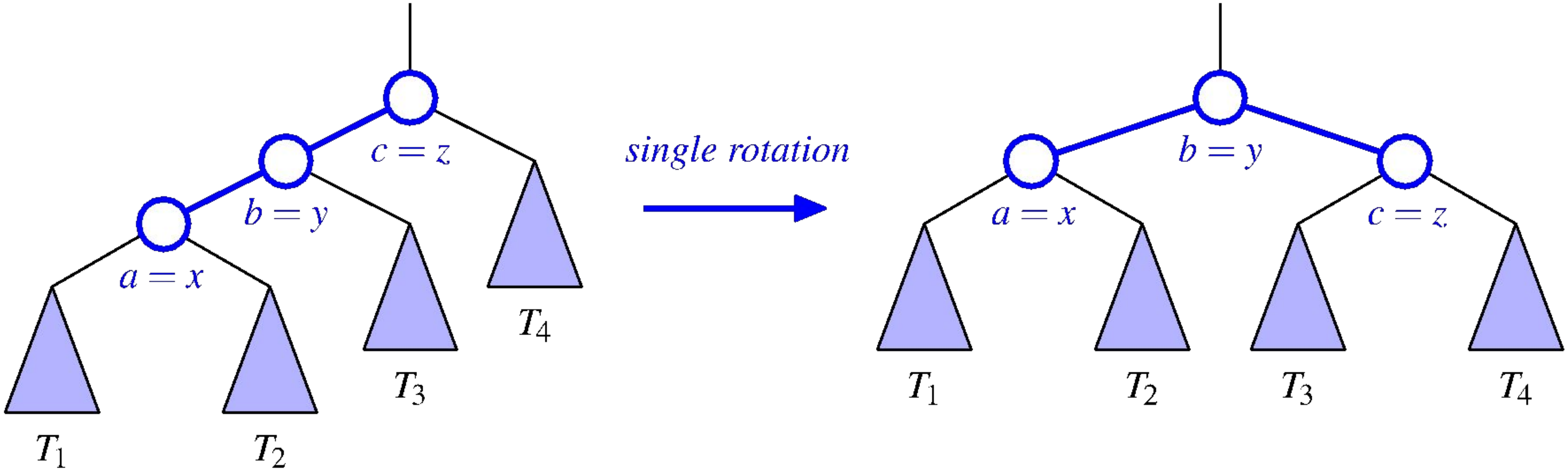


Should we do a left or right rotation?

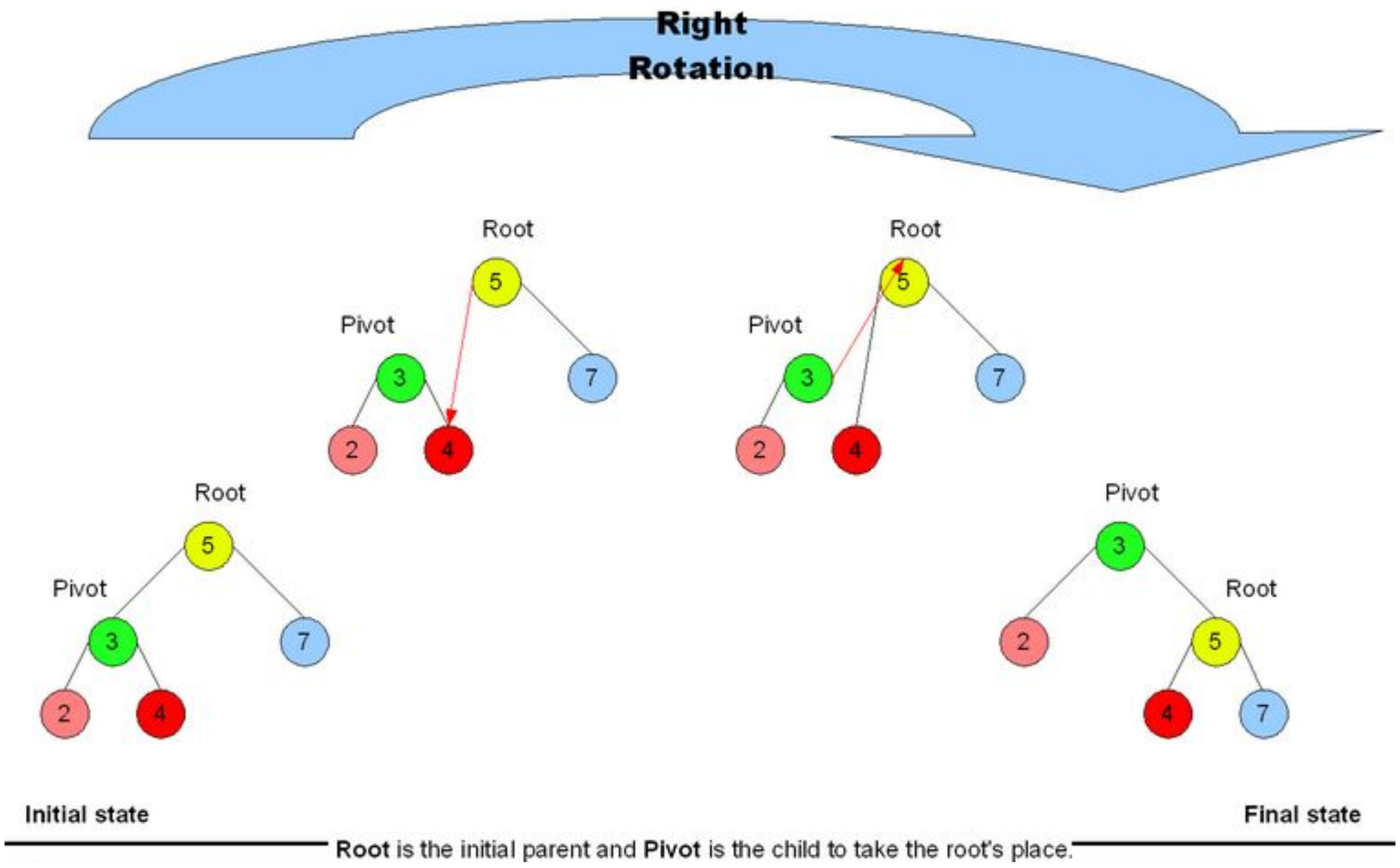
What will become the root?

Let's draw what it will look like after rotation

Example 2: Rotate Right



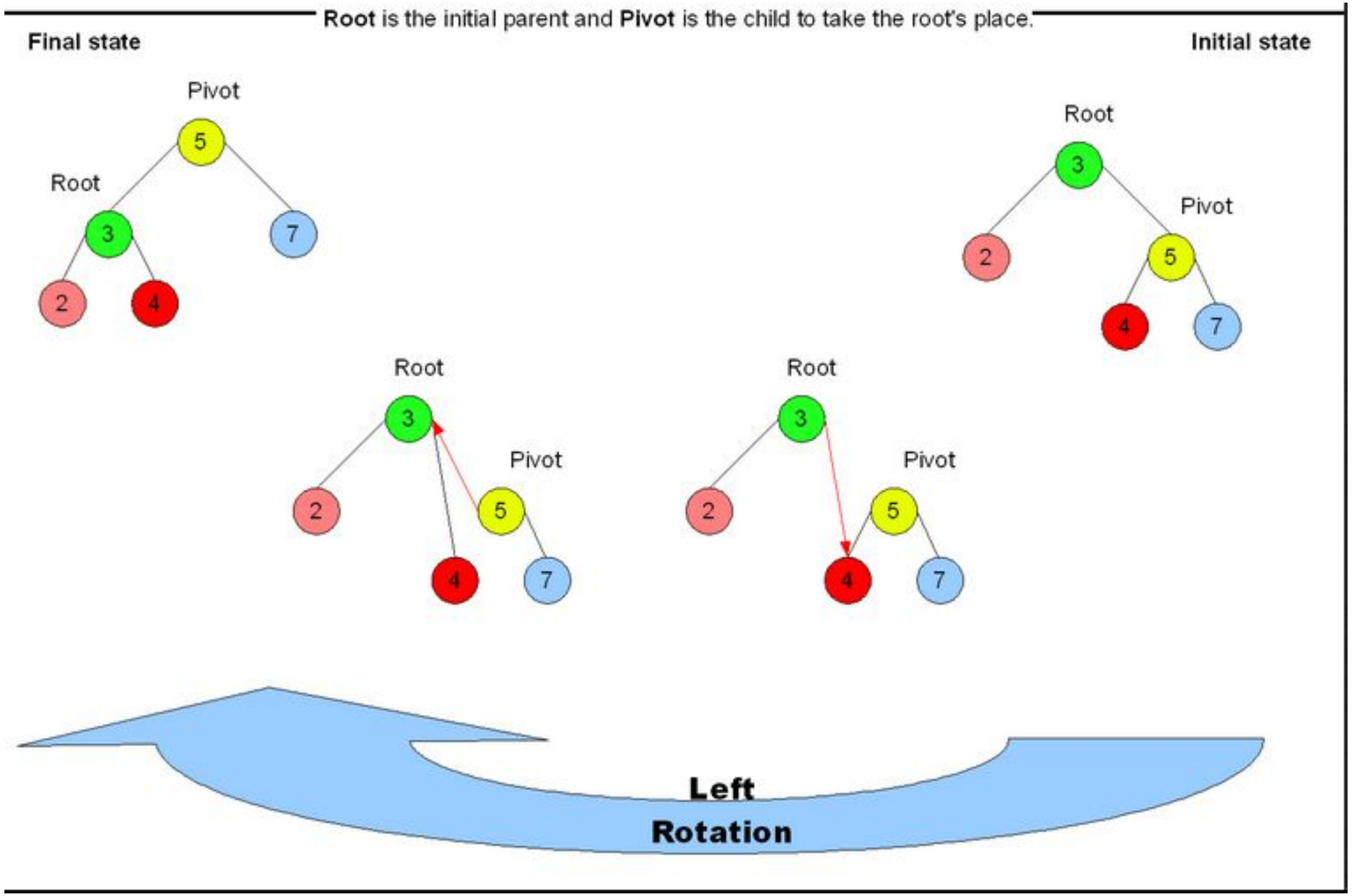
RotateRight Algorithm



1. `Root.left = Pivot.right`

1. `Pivot.right = root`

RotateLeft Algorithm



1. `Root.right = Pivot.left`

1. `Pivot.left = root`

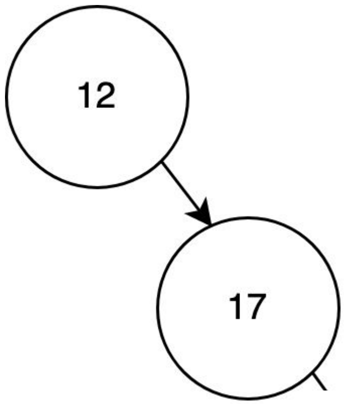
Example:

1. What is the height of the right and left subtrees?

2. Is this tree balanced?

3. Insert 140. Now, revisit questions (1) and (2)

4. Rotate? Which one?



Runtime Complexity

Runtime Complexity of rotation?

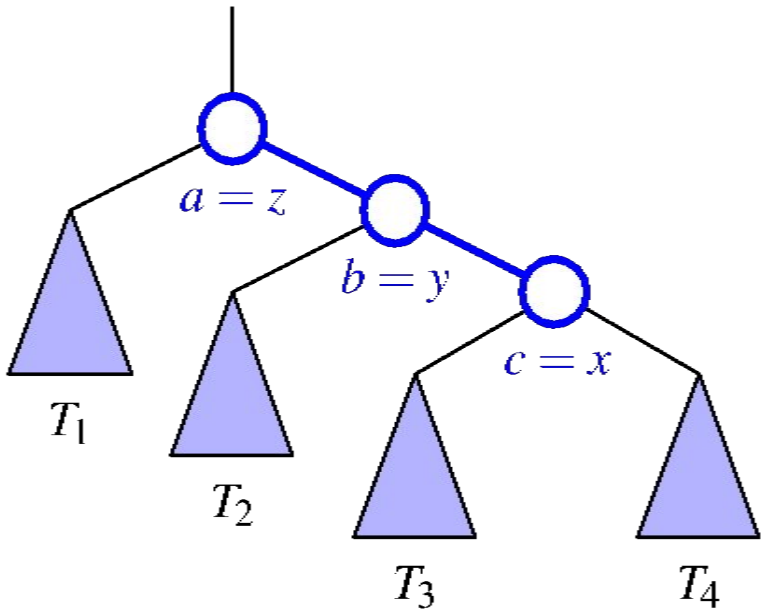
- O(1)

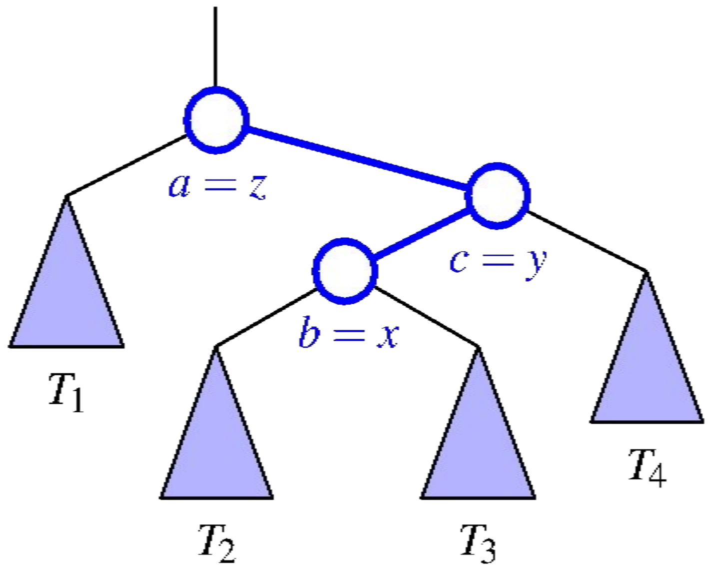
Constant time... we’re just updating links

Double Rotation

Sometimes a single rotation is not enough to restore balance

Double Rotation

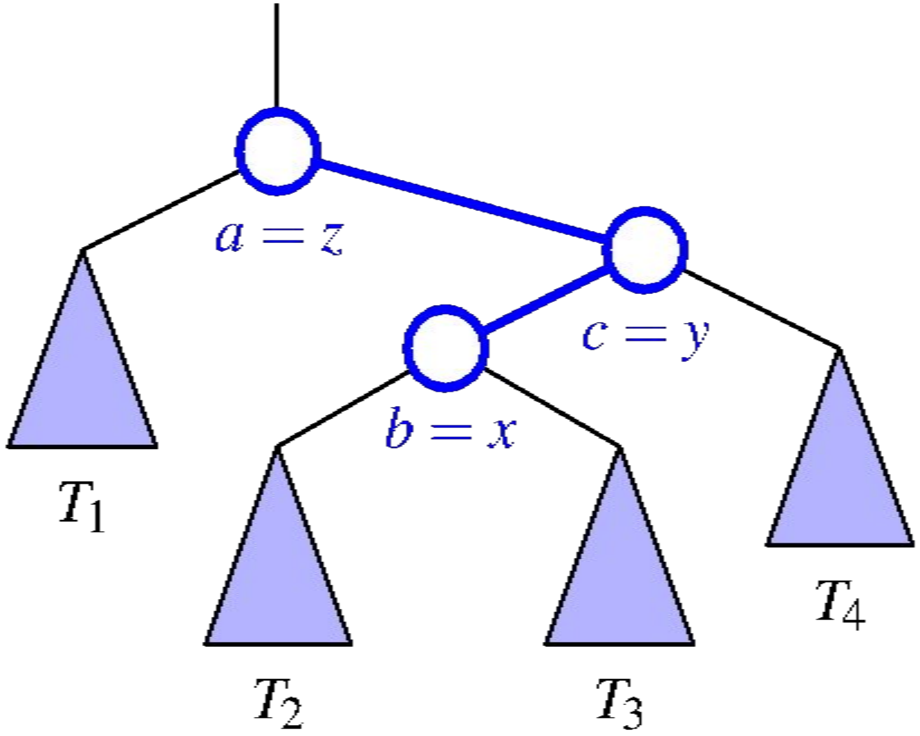




Right child of a is too heavy.. because
Right subtree of b is too heavy..
Single Left rotation on the root needed

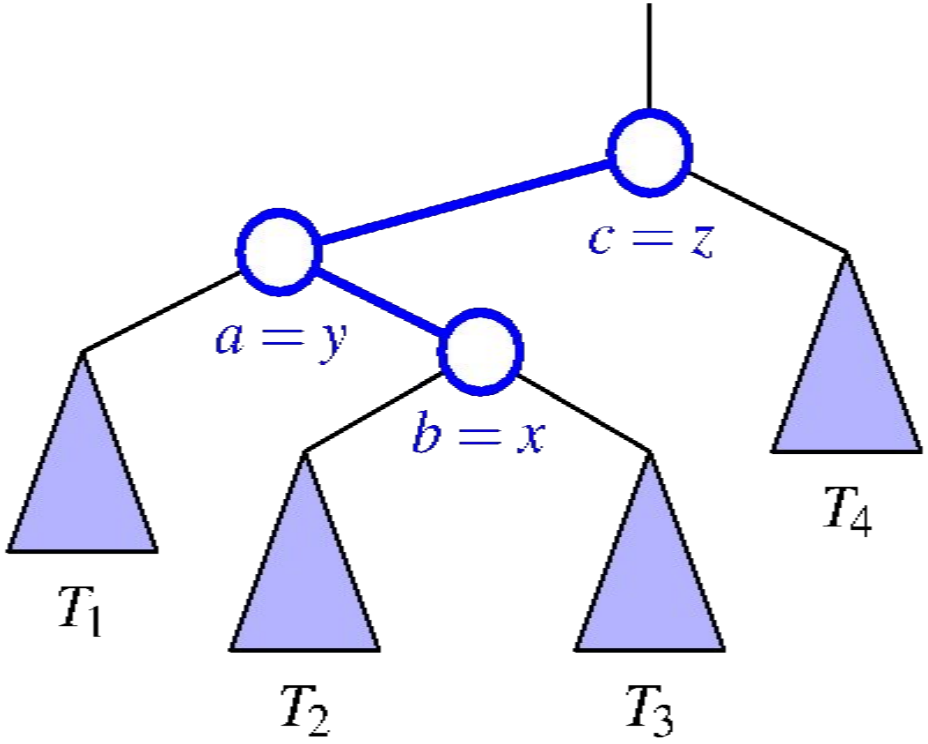
Right child of a is too heavy... because
Left subtree of c is too heavy
Is a single rotation enough?

Double Rotation

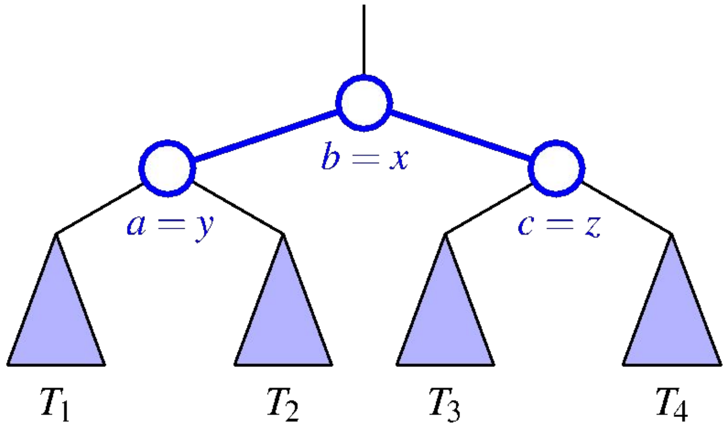


1. **Rotate Right** at c because right subtree of root is too heavy
2. **Rotate Left** at the root (a)

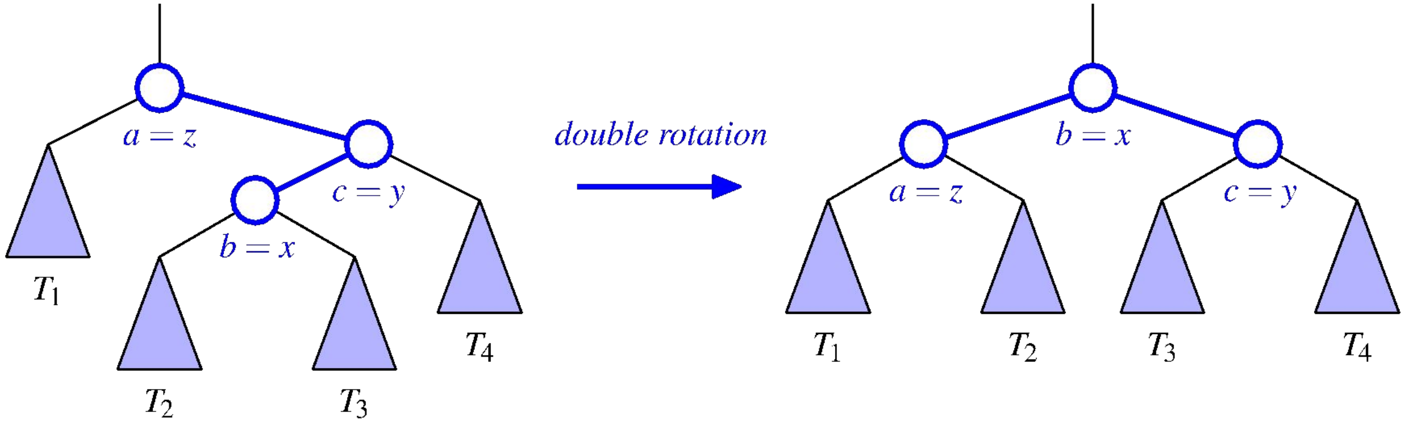
Double Rotation Example 2:



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

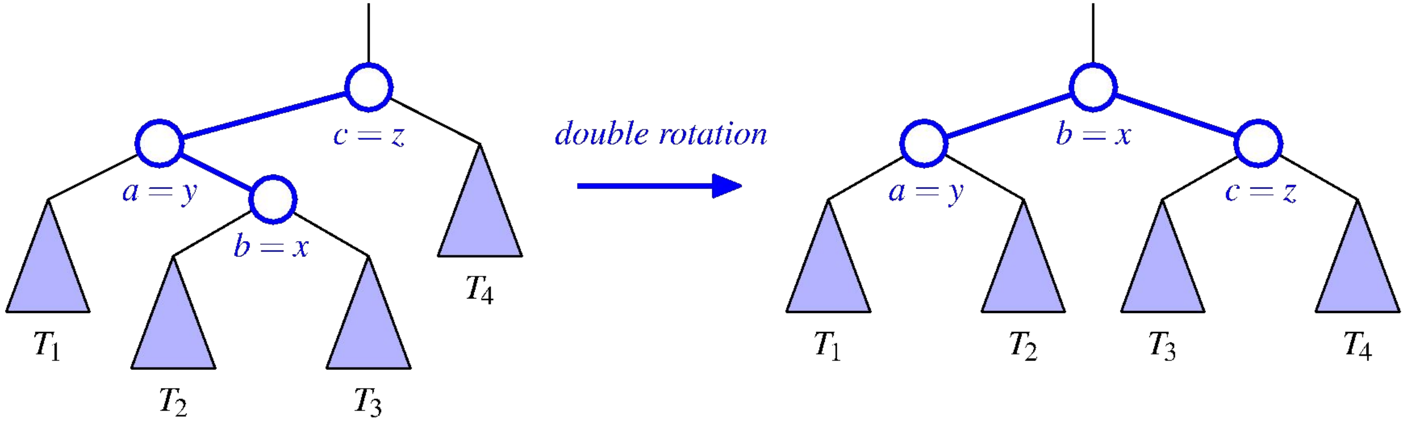


Double Rotations



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

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

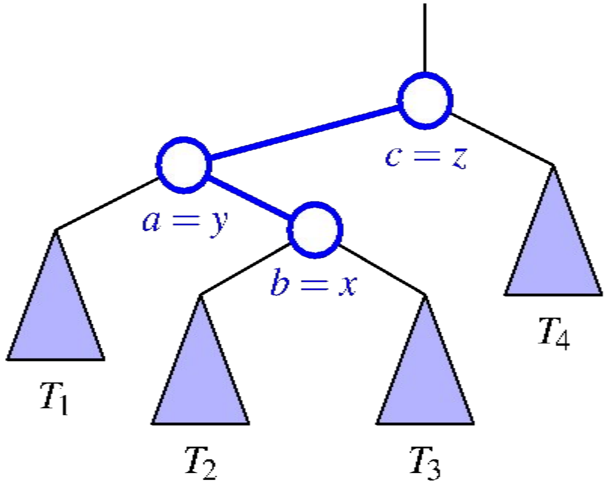


Left subtree is too heavy because of **right** subtree of a

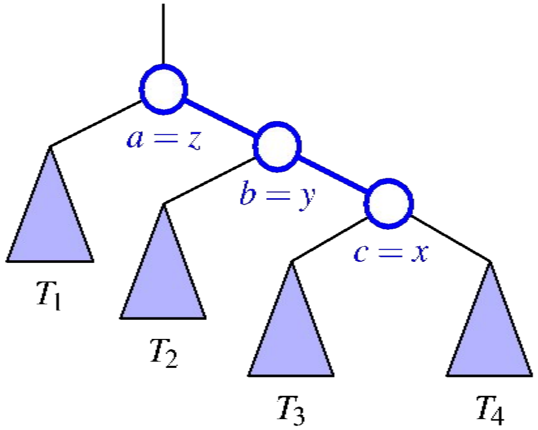
1. Rotate Left about a
2. Rotate Right about c

Double Rotation

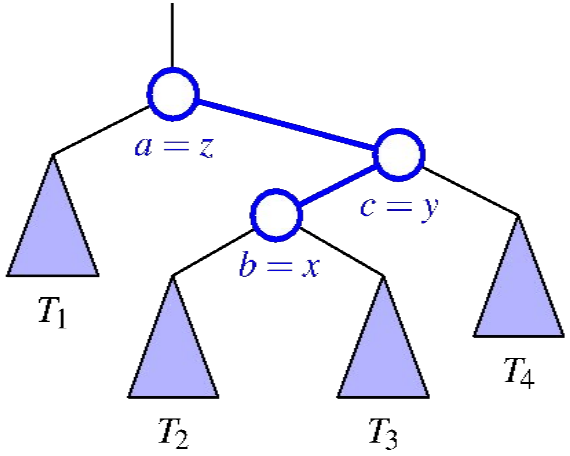
When do we need a double rotation vs a single rotation?



Double rotation



Single rotation



Double rotation

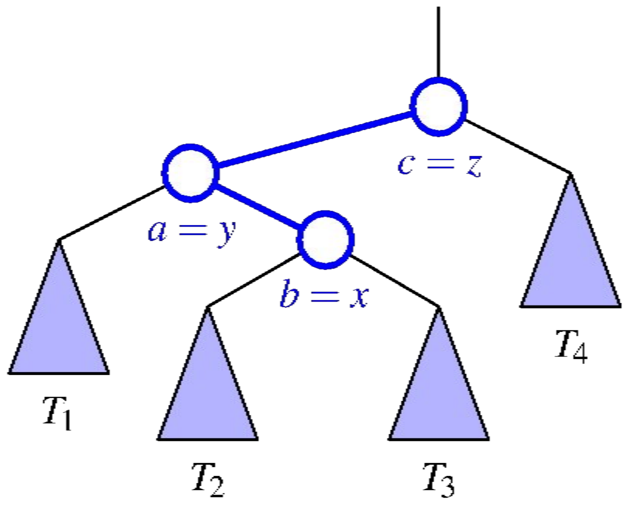
Look for zig-zag pattern!

Double rotation

When do we need a double rotation?

Left subtree is too heavy on the right side

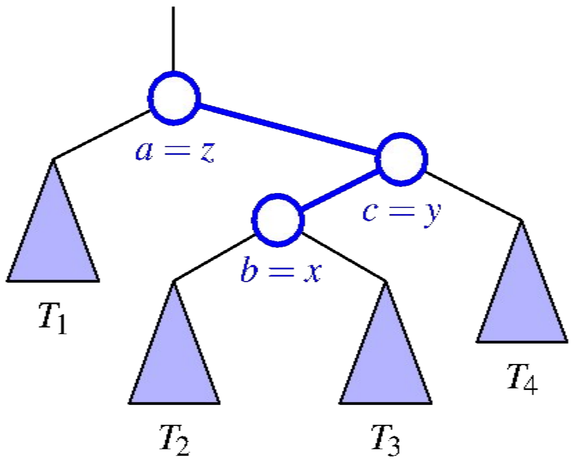
`rotateLeftRight`



OR

Right subtree is too heavy on the left side

`rotateRightLeft`

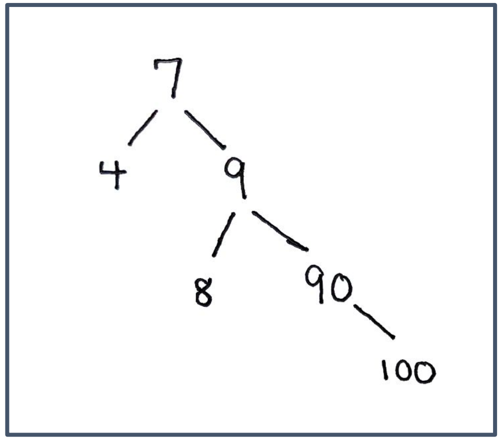


Double Rotation Code

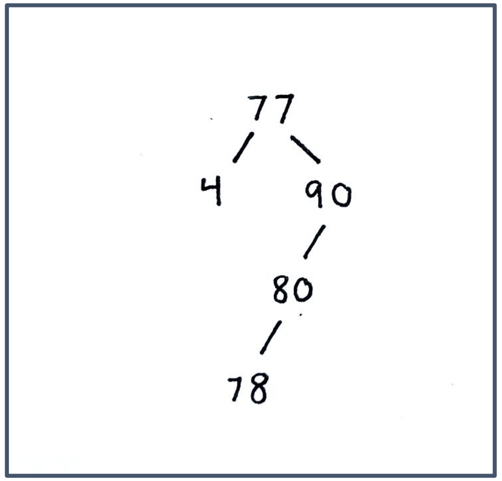
```
def rotateLeftRight(n)
    n.left = rotateLeft(n.left);
    n = rotateRight(n);

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

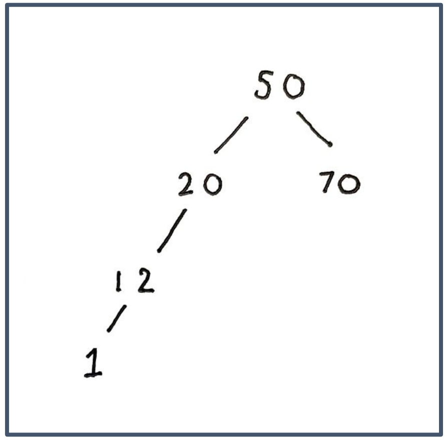
Examples - which way should I rotate?



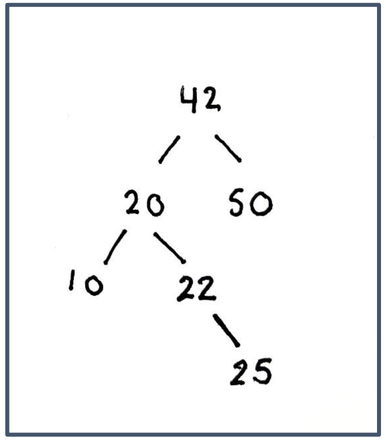
rotateLeft



rotateRightLeft



rotateRight



rotateLeftRight

Summary: Tree rotation

- Can rotate to left or right
- Used to restore balance in height
- Rotation maintains BST order
- Runtime complexity of rotation?
  - O(1)

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

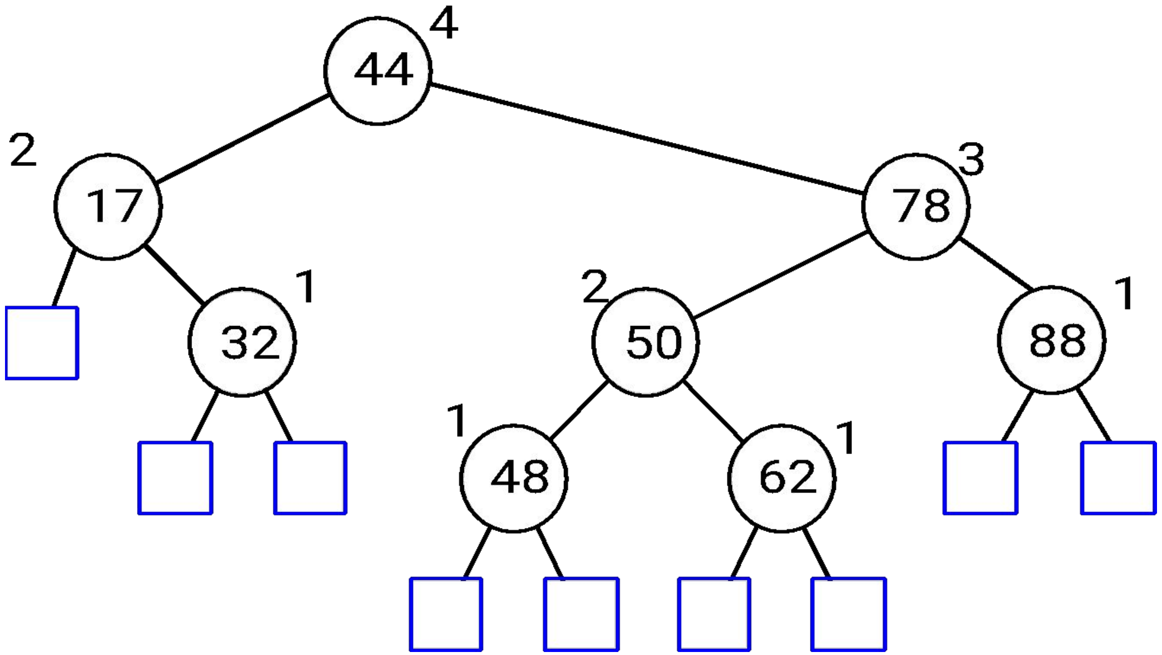
AVL Height

- We keep track of the height of each node as a field for quick access
  - height of a leaf is 1

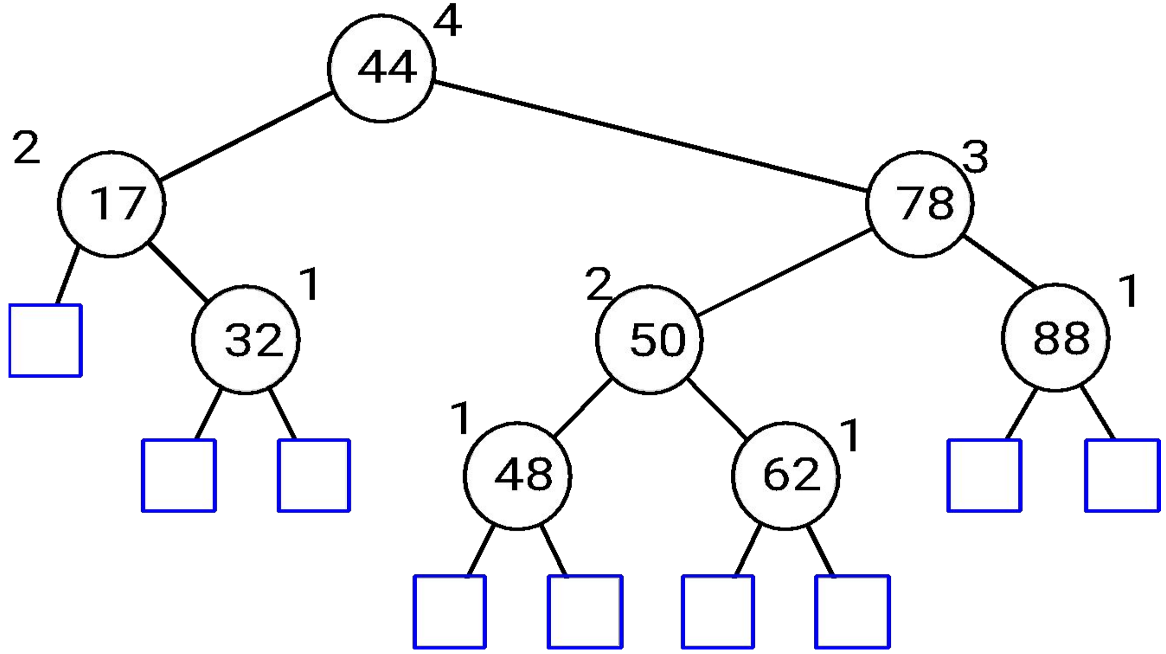
- The height of an AVL tree is logn
  - Always balanced

Insertion

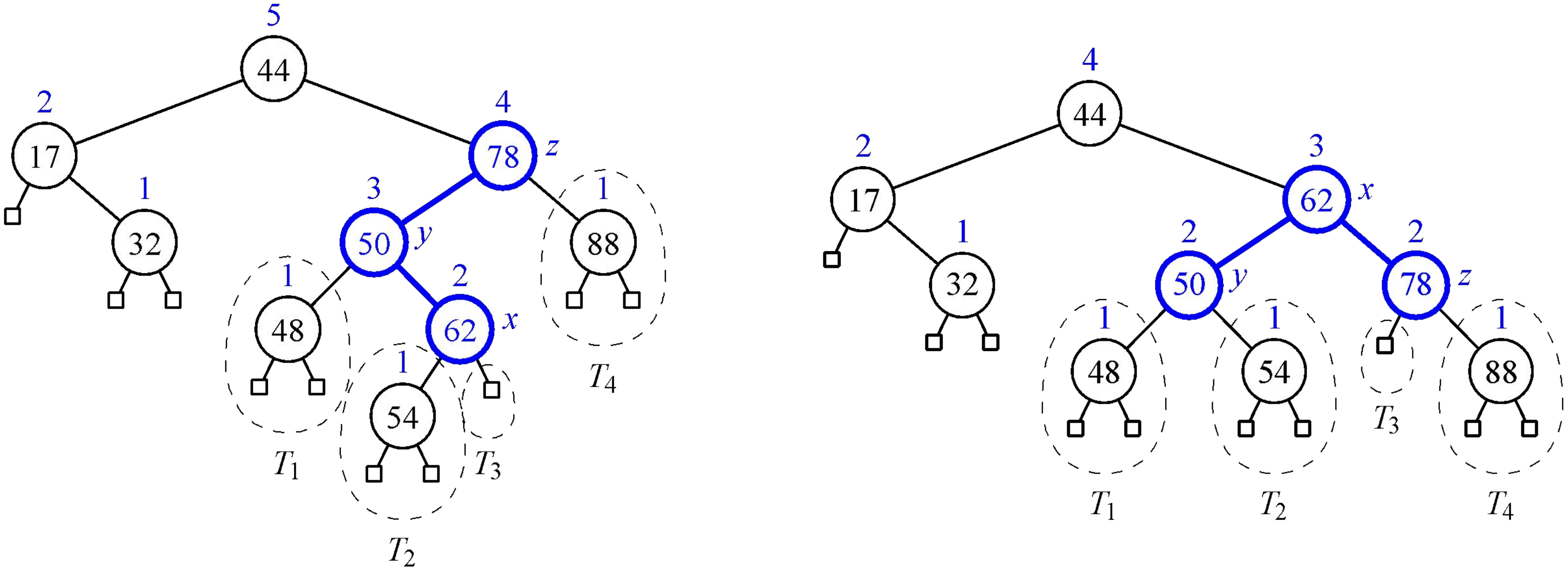
AVL Tree Example



Insert 54



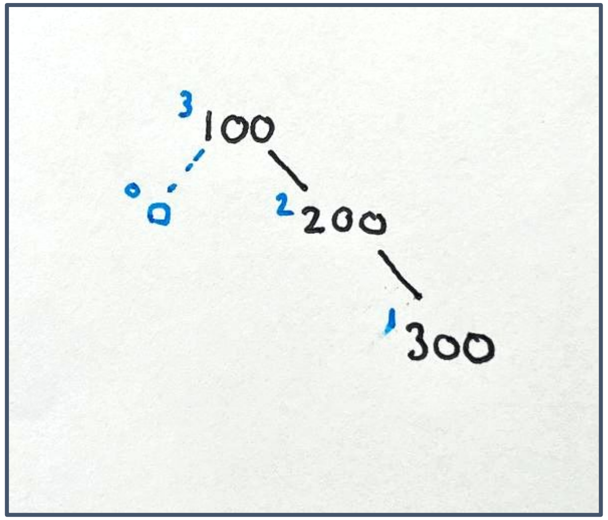
Insertion (54)

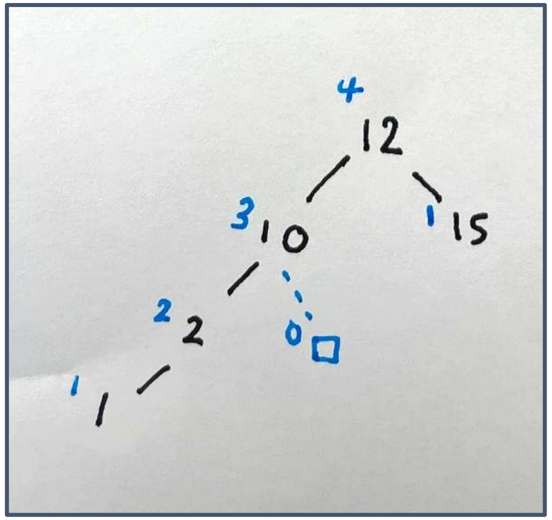


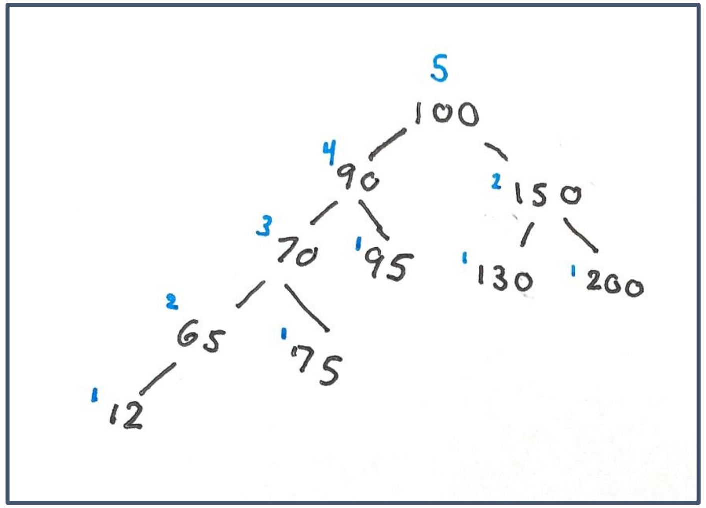
New node always has height 1

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)
```

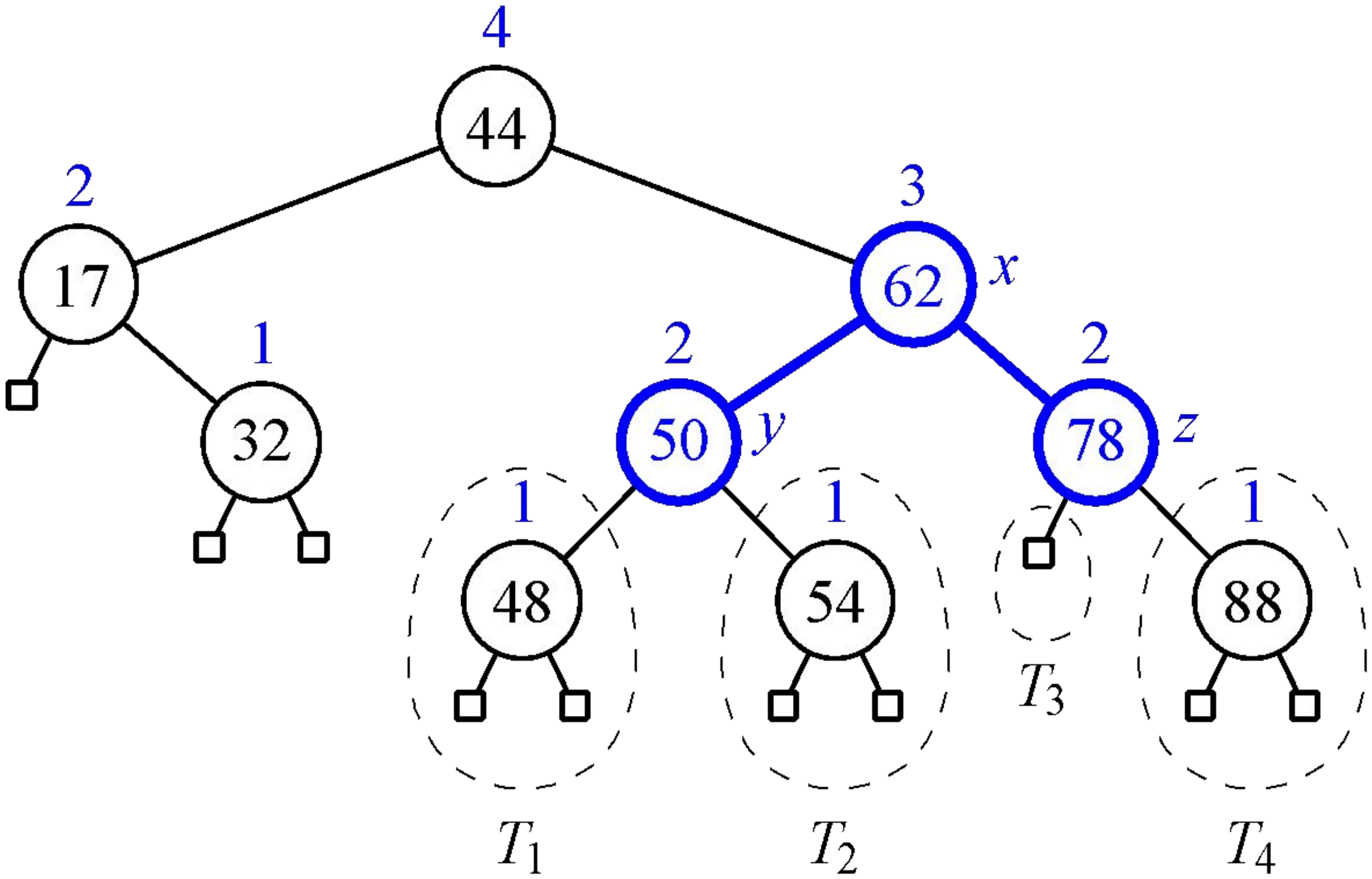
Runtime Complexity:

Insertion (plus rotation)

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

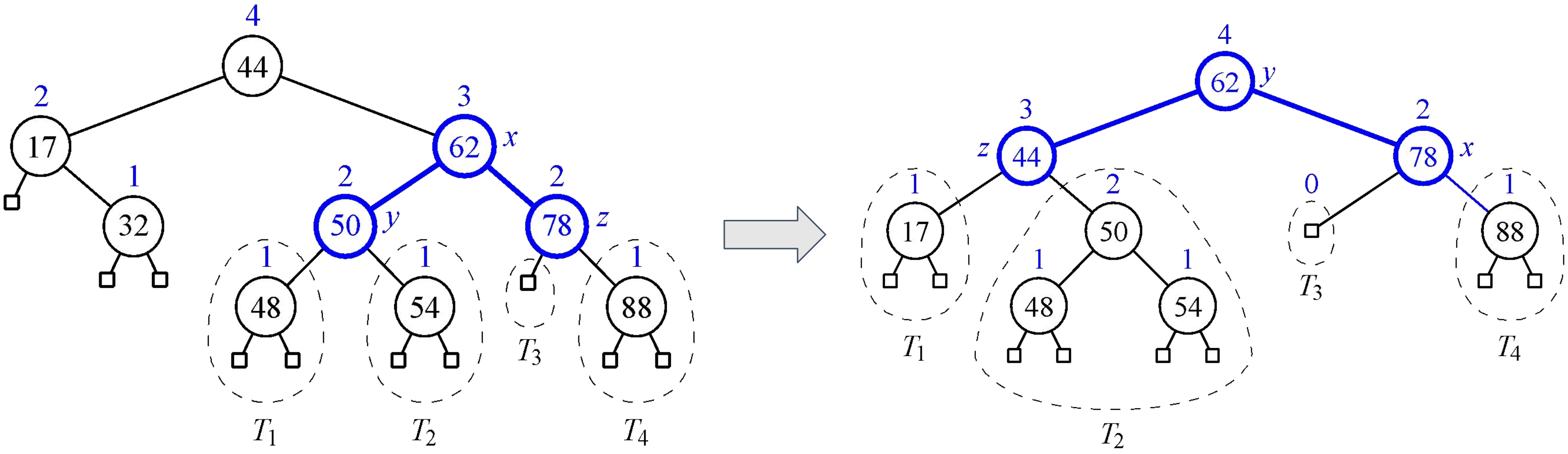
Deletion

Delete Example 1: 32

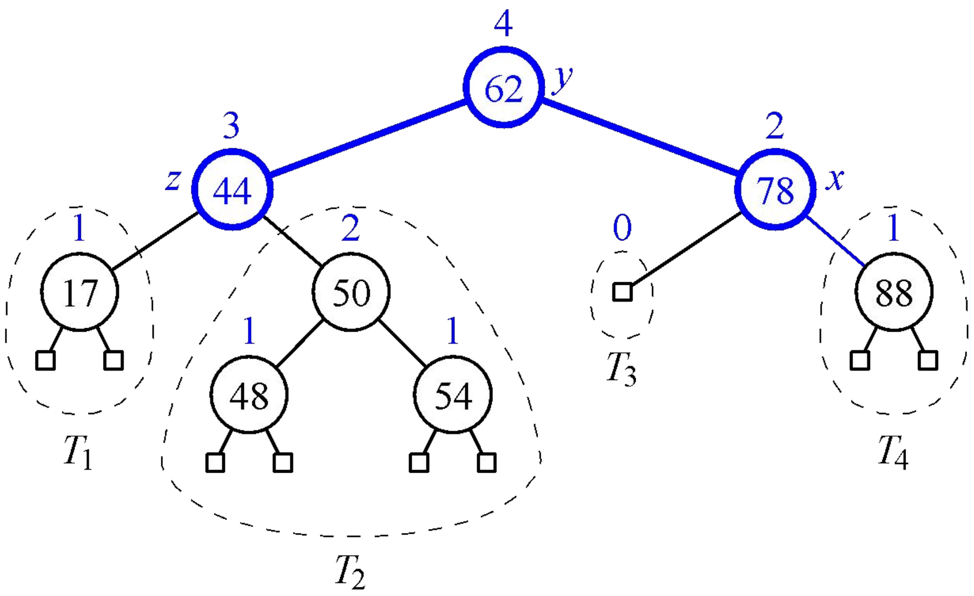


Delete Example 1: 32

rotateLeft

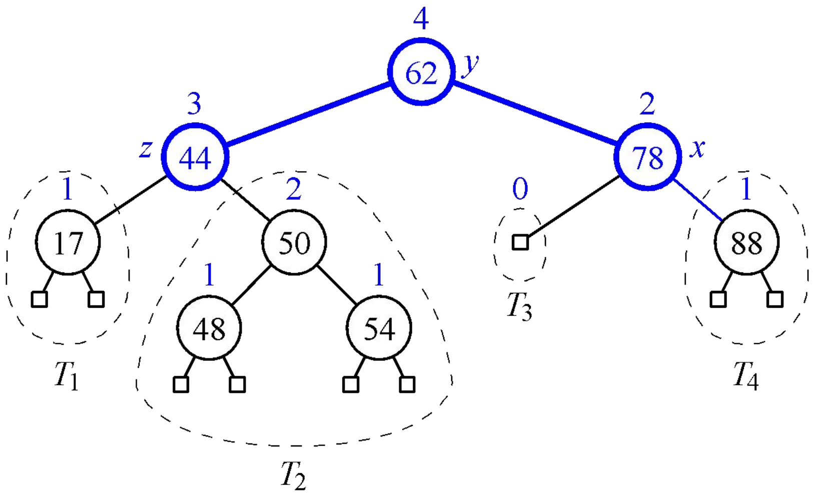


Delete Example 2: 78

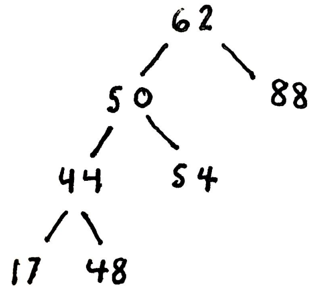


Delete Example 2: 78

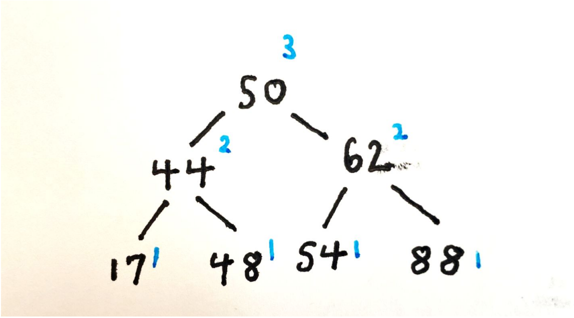
rotateLeftRight



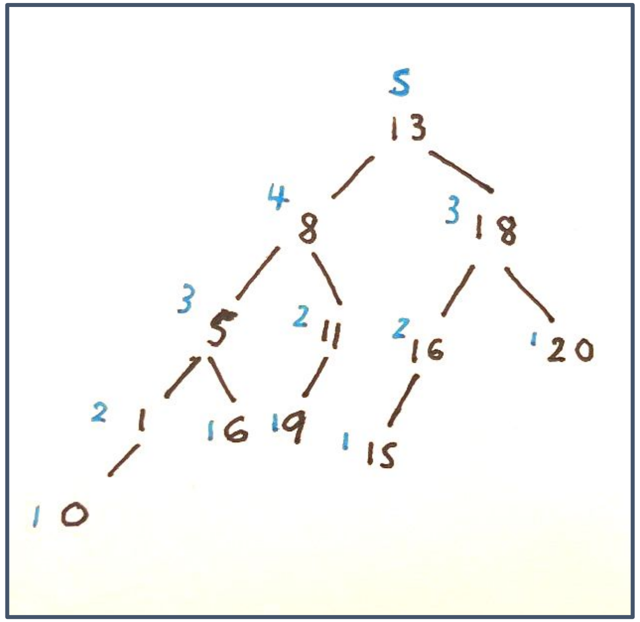
rotateLeft



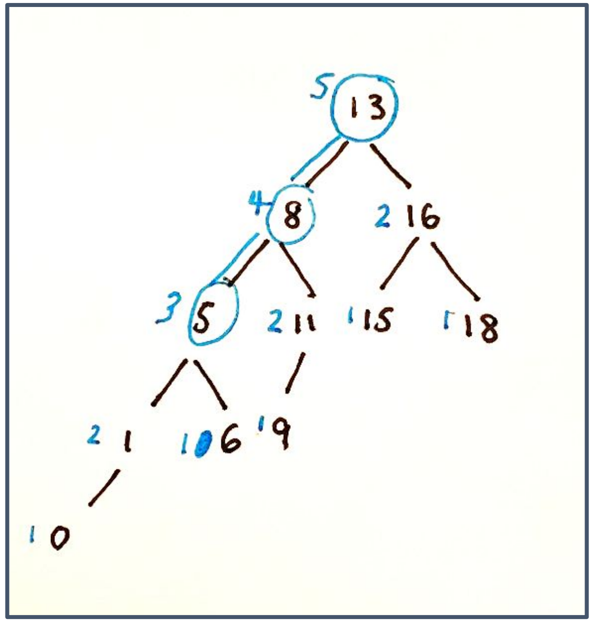
rotateRight



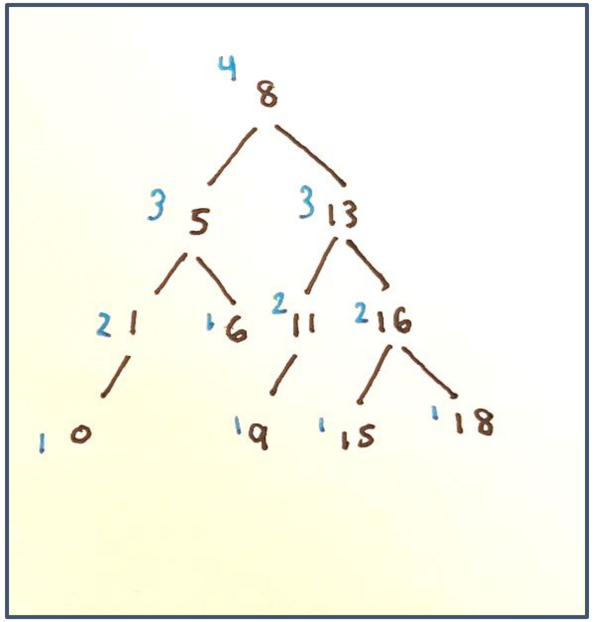
Delete Example 3: 20



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:
  - $O(\log n)$

Summary

AVL Trees:

- BST with a rotate operation which maintains tree balance
- O(logn) operations

Rotations:

- double rotation needed when
  - Left subtree is too heavy on the right side OR
  - Right subtree is too heavy on the left side (zig-zag pattern)

Rotations are constant time