# CS151 Intro to Data Structures

Quick Sort
Balanced Binary Search Trees

# Announcements

HW7 and Lab9 due Friday 4/18

Lab9 Manual checkoff

# Outline

Warmup

Sorting:

1. QuickSort

Balanced BSTs!

# Which data structure would you use?

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.

# Quicksort

# Quicksort

- Divide and conquer
- **Divide:** select a *pivot* and create three sequences:
    a. L: stores elements less than the pivot
    b. E: stores elements equal to the pivot
    c. G: stores elements greater than the pivot
- **Conquer:** recursively sort L and G
- **Combine:** L + E + G is a sorted list

# Quick Sort

Sort [2, 6, 5, 3, 8, 7, 1, 0]

1. choose a pivot
2. swap pivot to the end of the array
3. Find two items:
   a. left which is larger than our pivot
   b. right which is smaller than our pivot

1. swap left and right
2. repeat 3 and 4 until right < left
3. swap left and pivot
4. Sort L E and R recursively

# Quick Sort - Choosing a pivot

What if we chose our pivot to be the smallest element?

We want a pivot that divides our list as evenly as possible.

Median-of-three: look at the first, middle, and last elems in the array, and pick the middle element.

# Quicksort runtime complexity

Bad pivot:

$O(n^2)$

Good pivot:

$O(n\log n)$

# Summary of Sorting Algorithms

| Algorithm | Time |
|---|---|
| selection-sort | |
| heap-sort | |
| merge-sort | |
| quick-sort | |

# Balanced Binary Trees

# What can go wrong?

Complexity?

**Search**

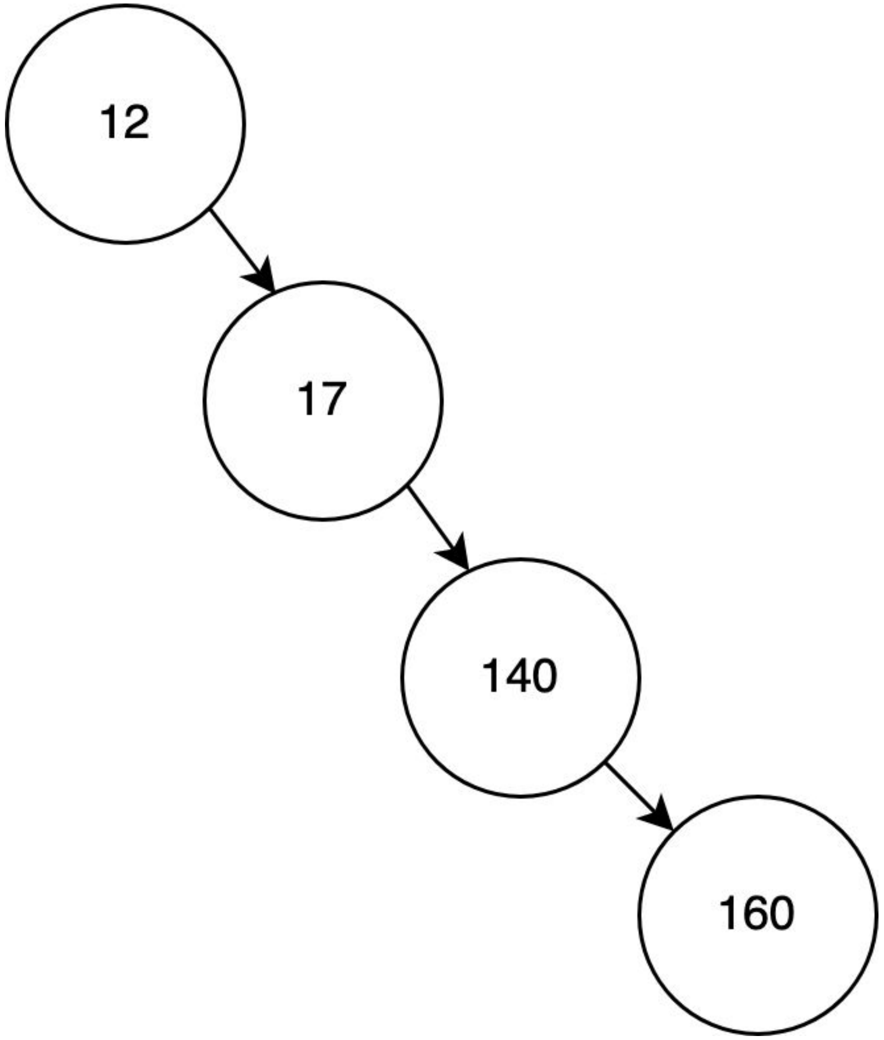
O(n)

**Insertion:**

O(n)

**Deletion:**

O(n)



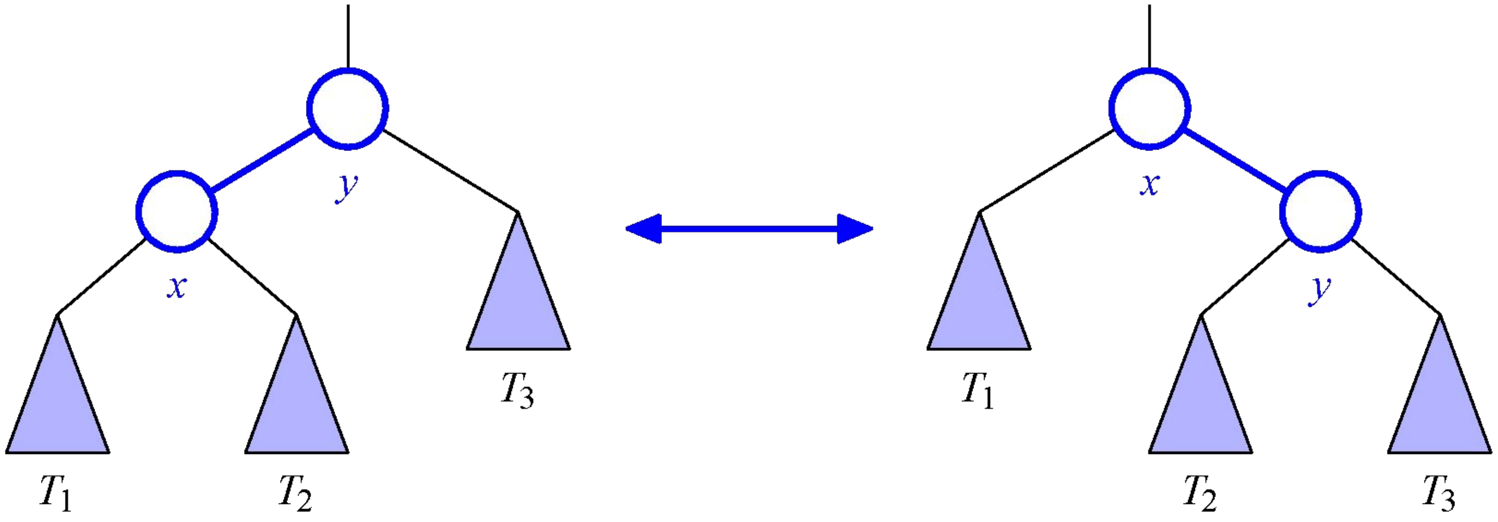
# Balanced Binary Trees

- 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

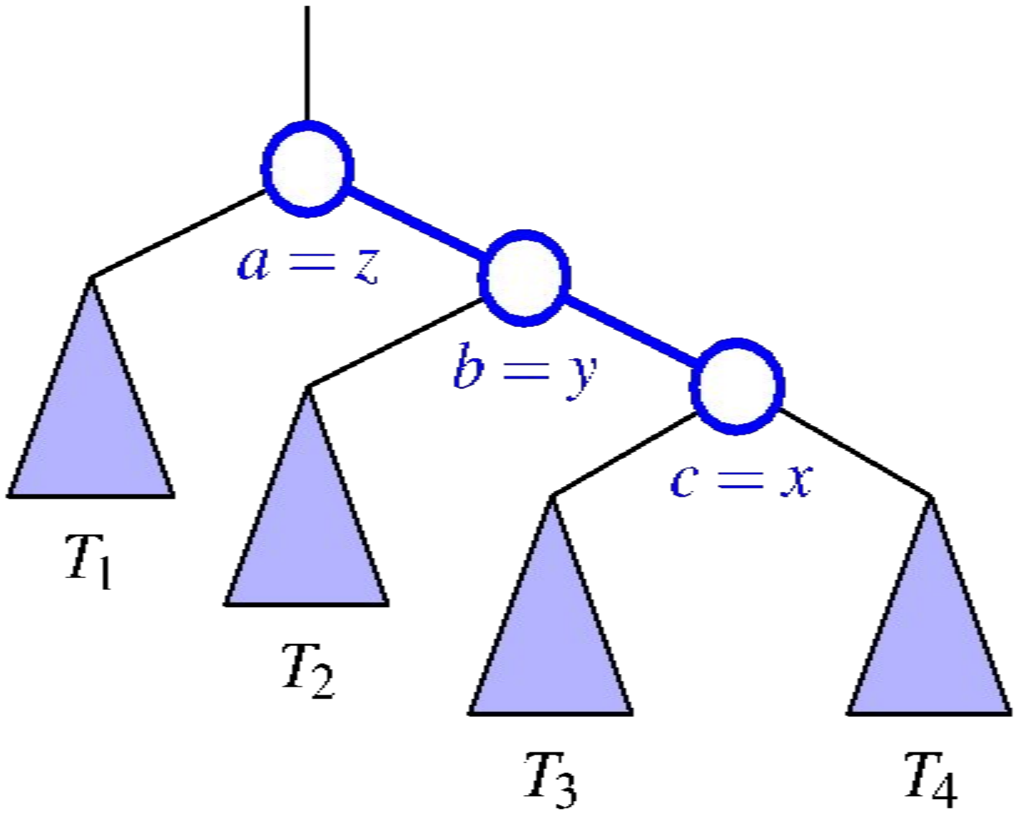


# Rotation Operation

- Used to maintain balance

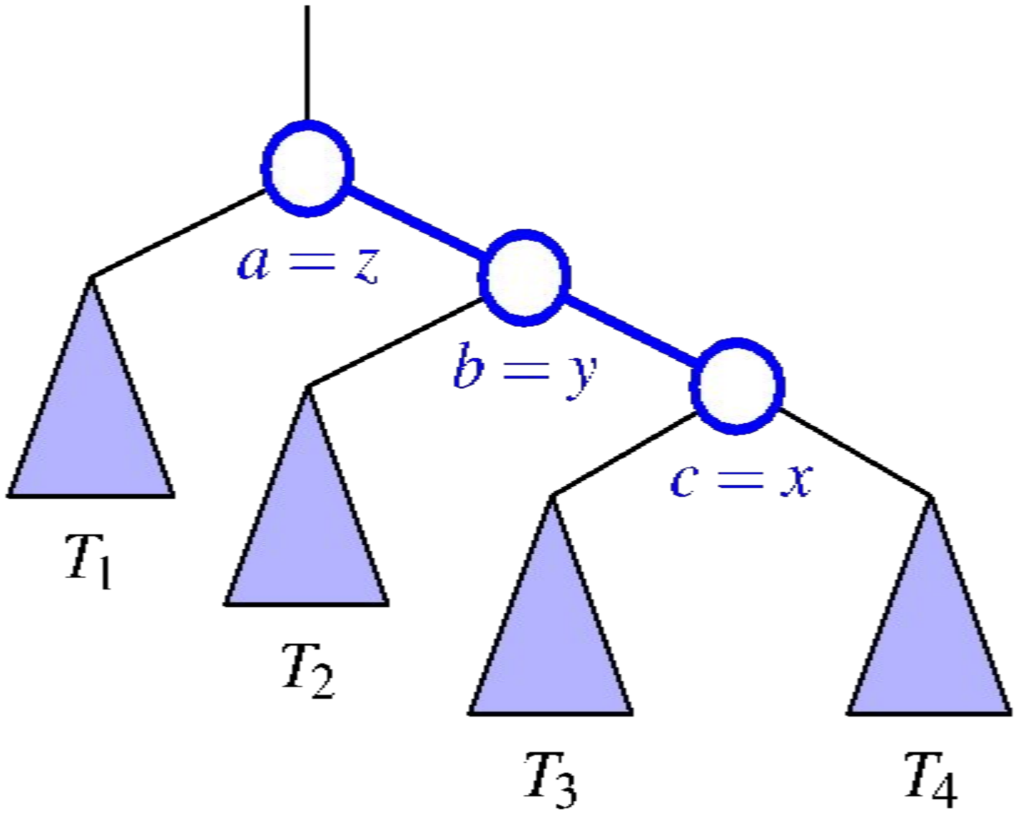
- When should **rotate** be invoked?
  - Difference of heights of left and right subtrees at any node is > 1

# 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

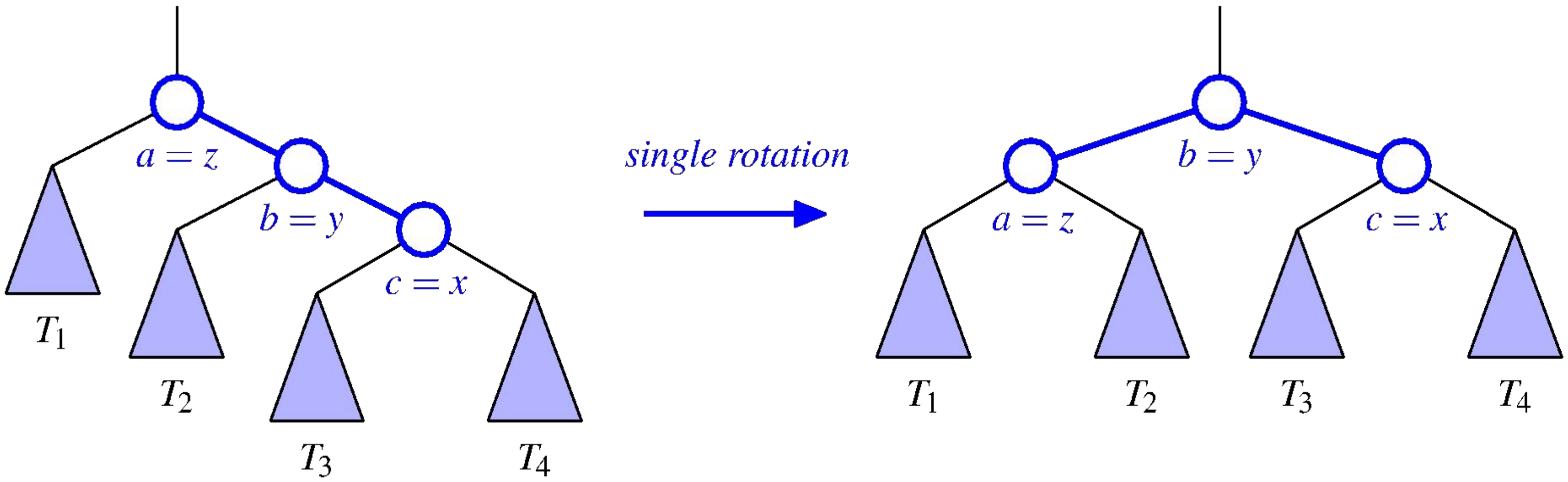



Right subtree is too large!

How can we rotate to fix this?

What should we make the root?

# Single Rotation (around $z$)



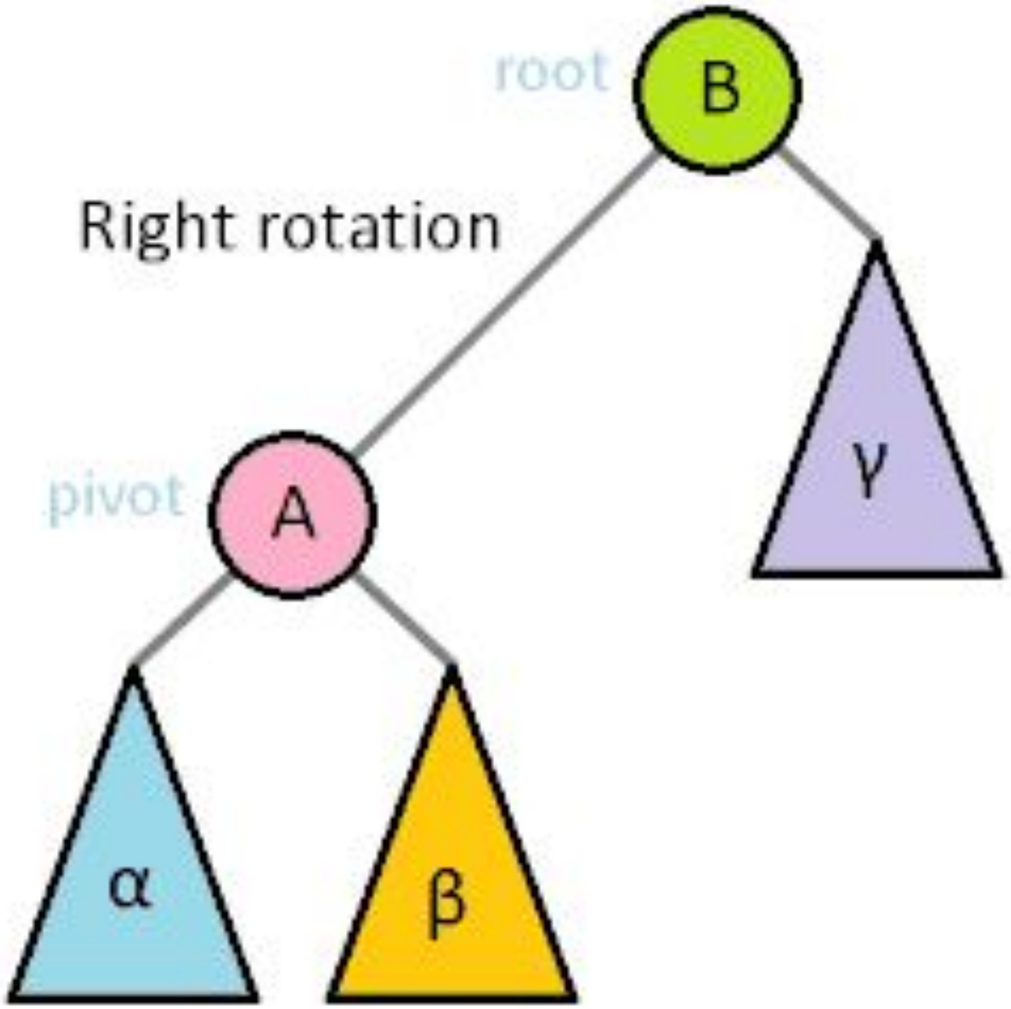
# Rotations

Right rotation:

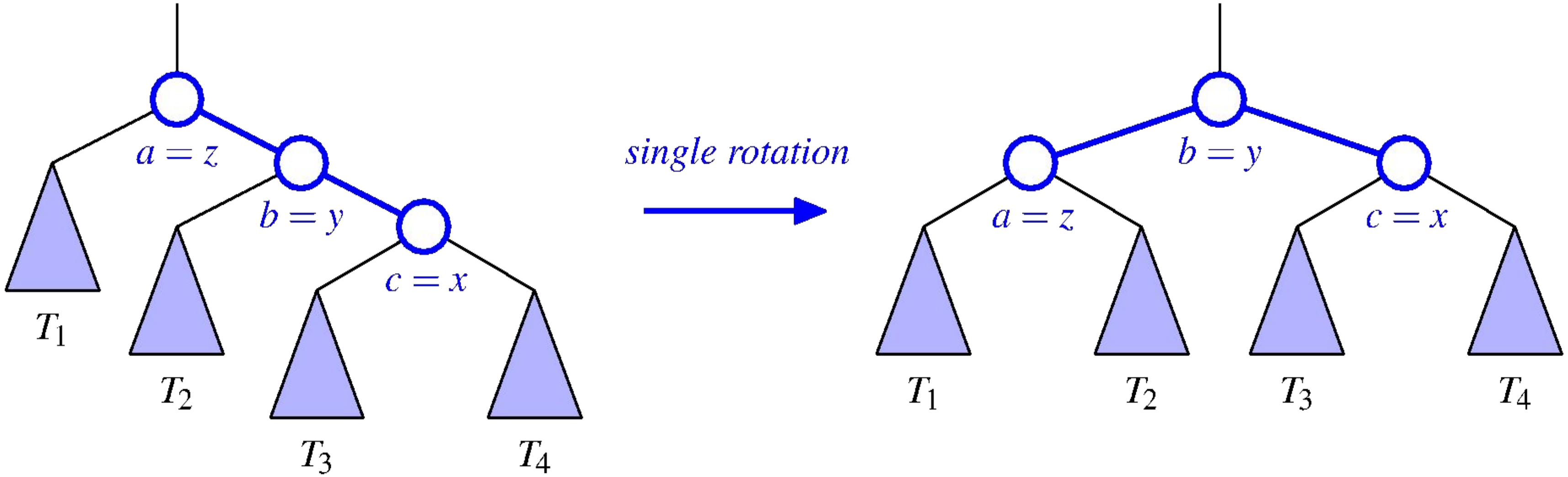
- Performed when left side is heavier
- left child becomes root

Left rotation:

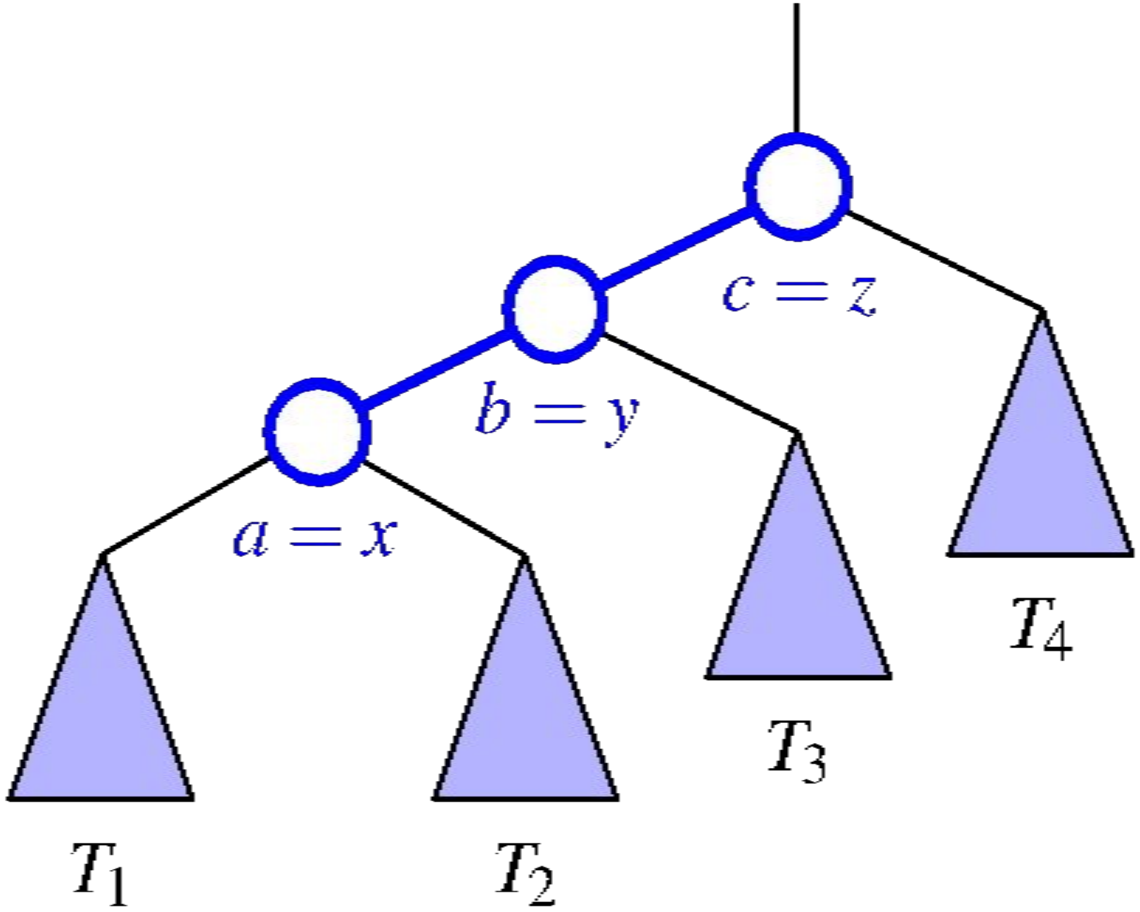
- Performed when right side is heavier
- right child becomes root



# Left or Right rotation?



# Example 2:

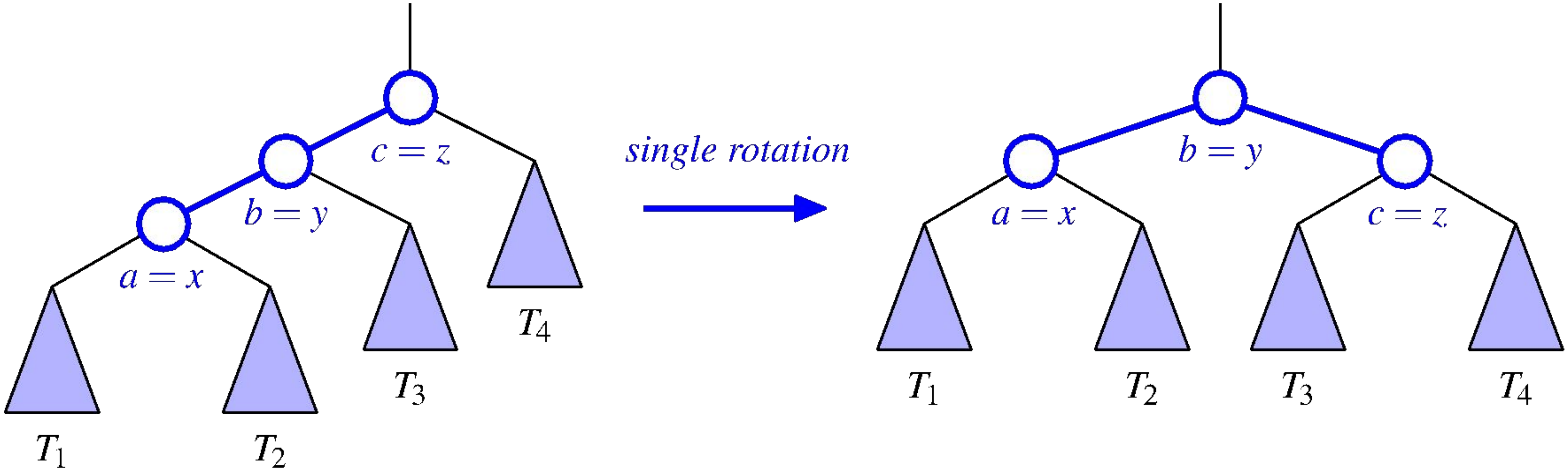



Should we do a left or right rotation?

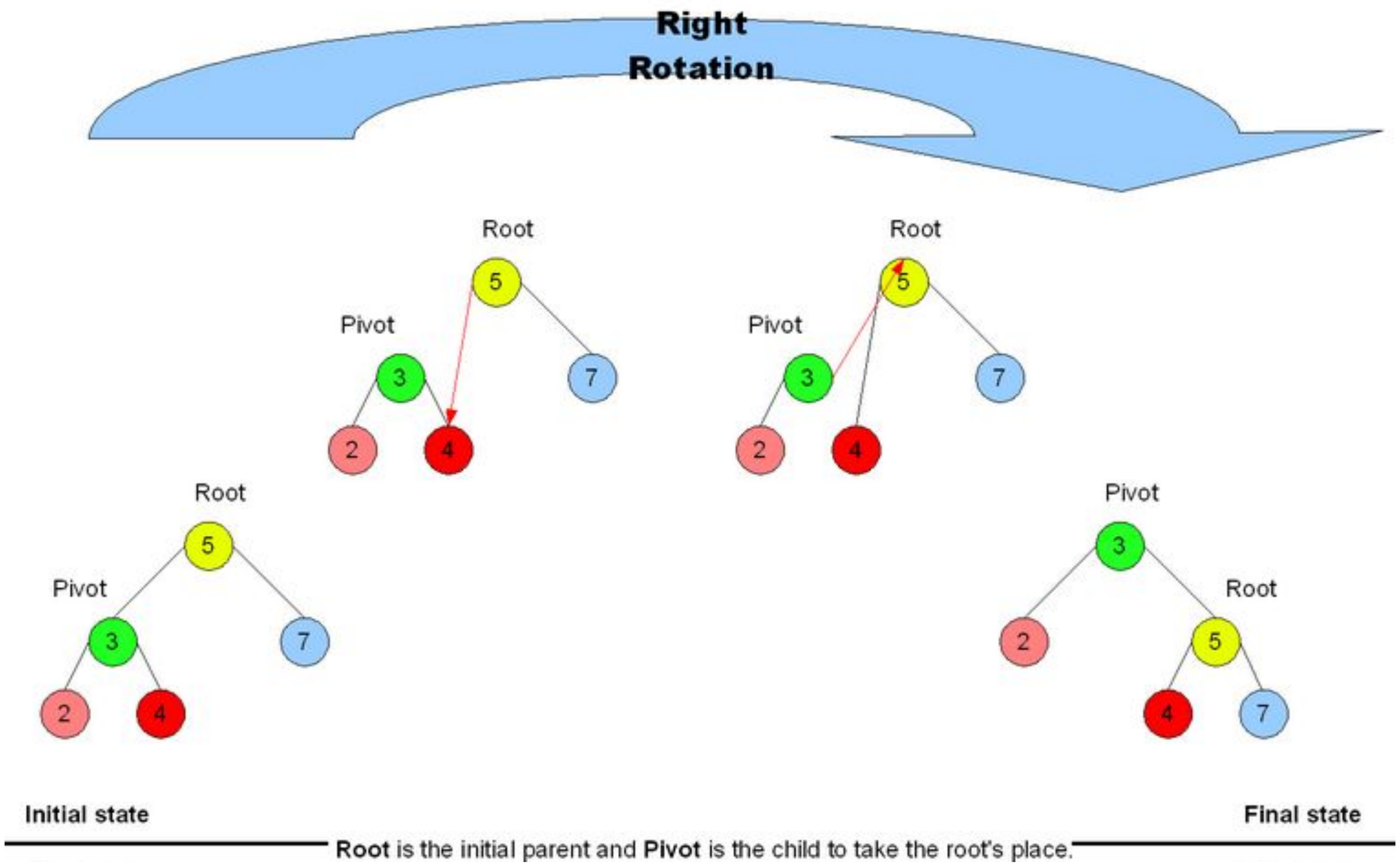
What will become the root?

Let’s draw what it will look like after rotation

# Example 2: Rotate Right



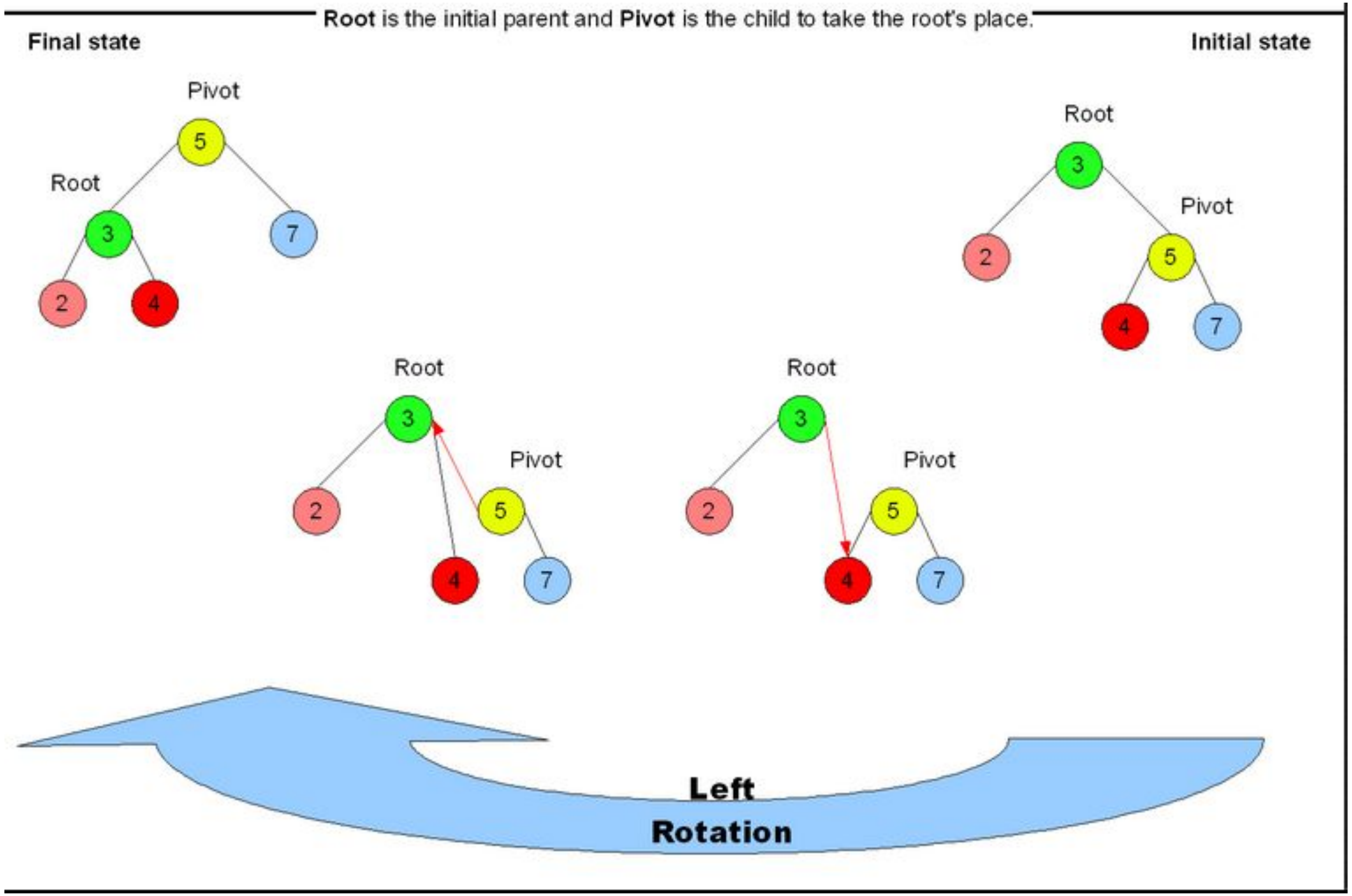
# RotateRight Algorithm




1. `Root.left = Pivot.right`

1. `Pivot.right = root`

# RotateLeft Algorithm




1. `Root.right = Pivot.left`

1. `Pivot.left = root`

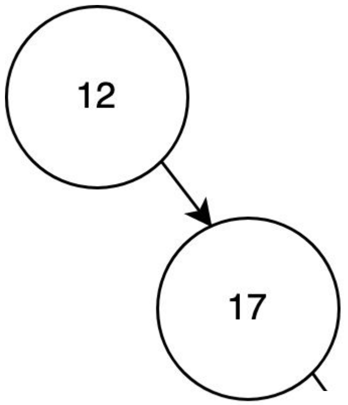
# Example:

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

1. Is this tree balanced?

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

1. Rotate? Which one?



# Runtime Complexity

Runtime Complexity of rotation?

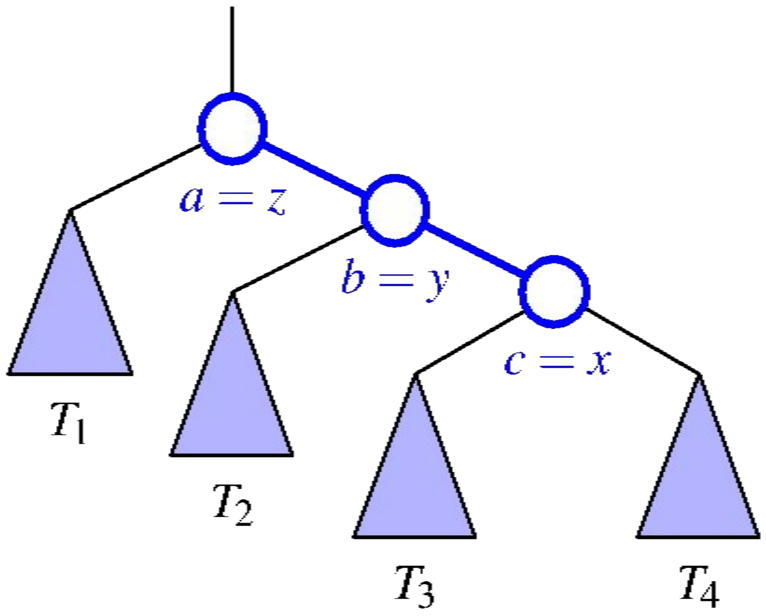
- O(1)

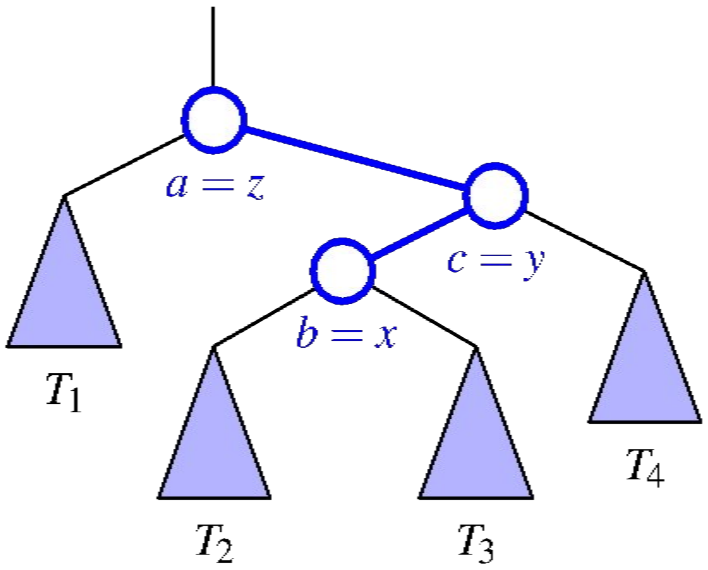
Constant time... we’re just updating links

# Double Rotation

Sometimes a single rotation is not enough to restore balance

# Double Rotation

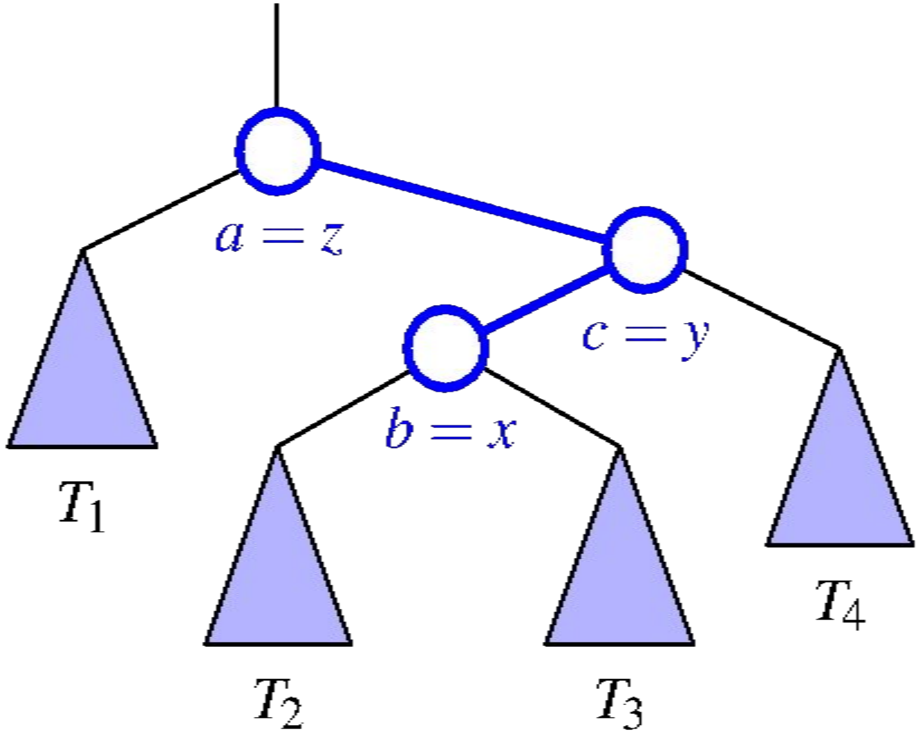



**Right** child of a is too heavy.. because
**Right subtree** of b is too heavy..
Single Left rotation on the root needed

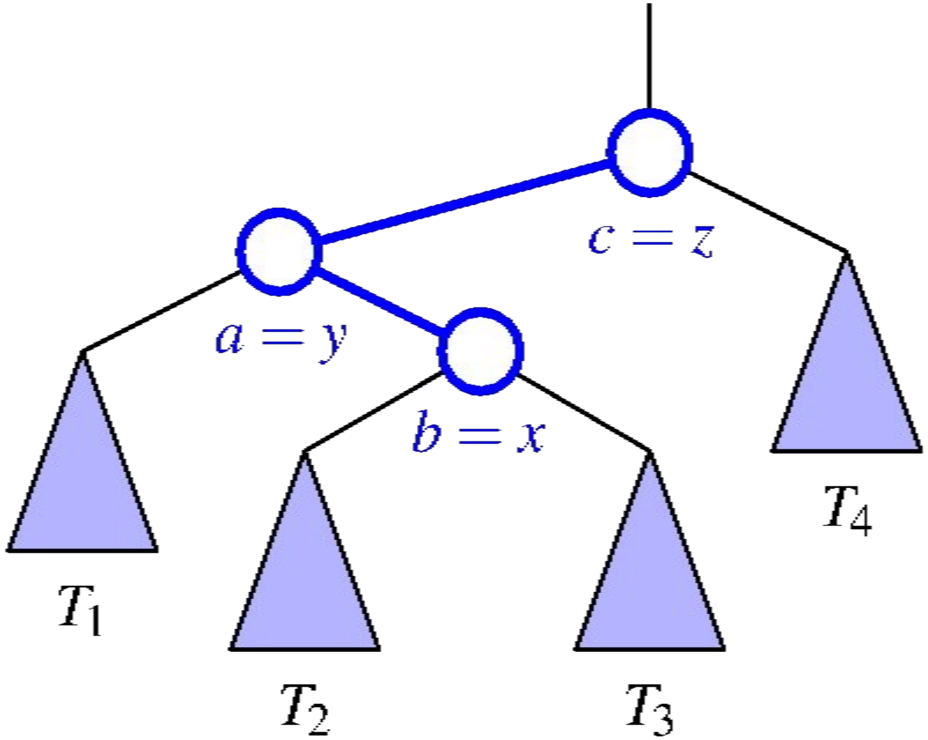
**Right** child of a is too heavy... because
**Left subtree** of c is too heavy
**Is a single rotation enough?**

# Double Rotation

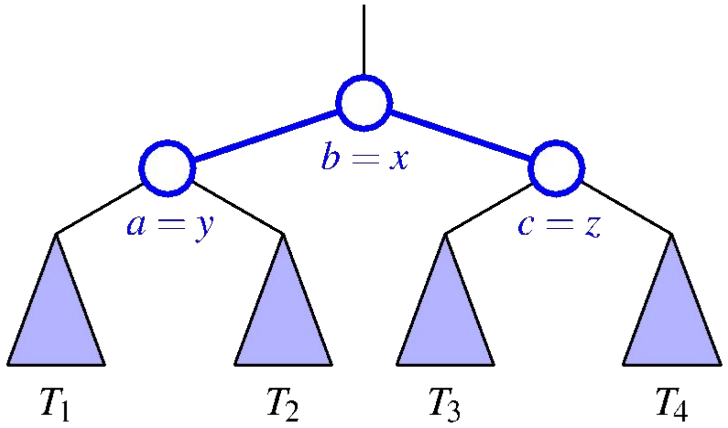



1. **Rotate Right** at c because right subtree of root is too heavy
2. **Rotate Left** at the root (a)

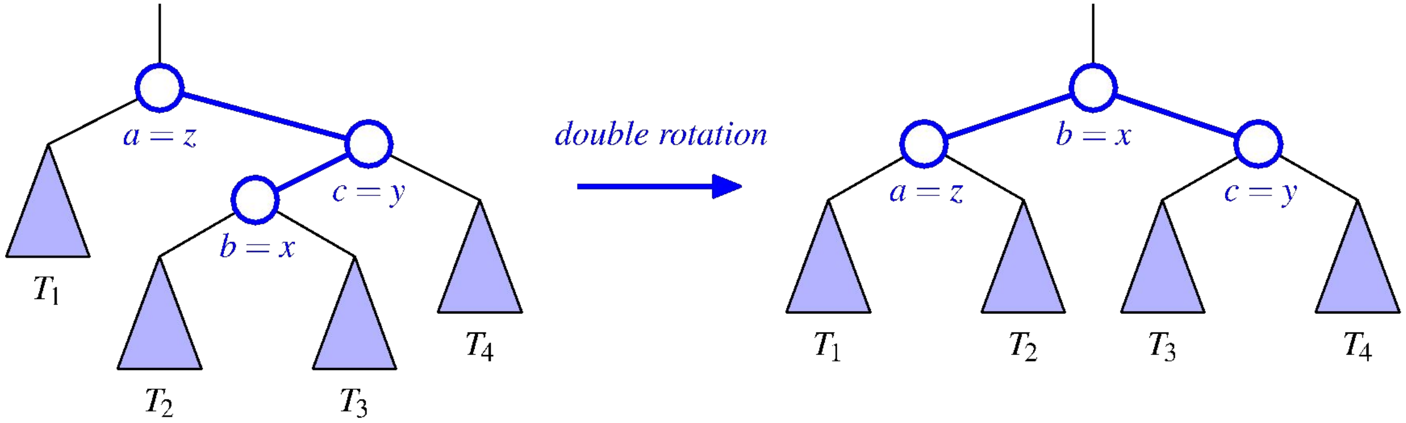
# Double Rotation Example 2:




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

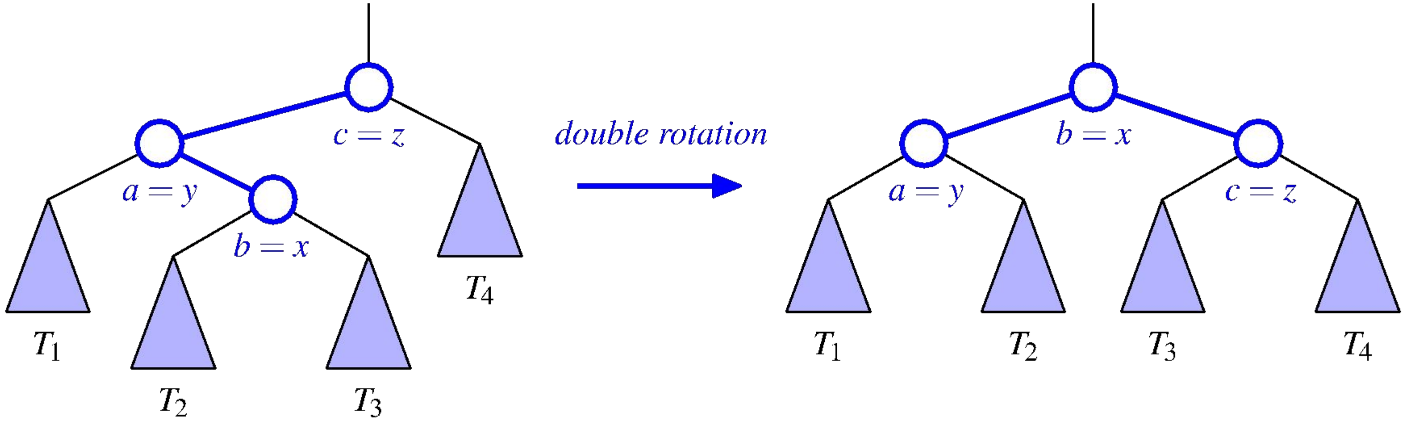


# Double Rotations




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

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

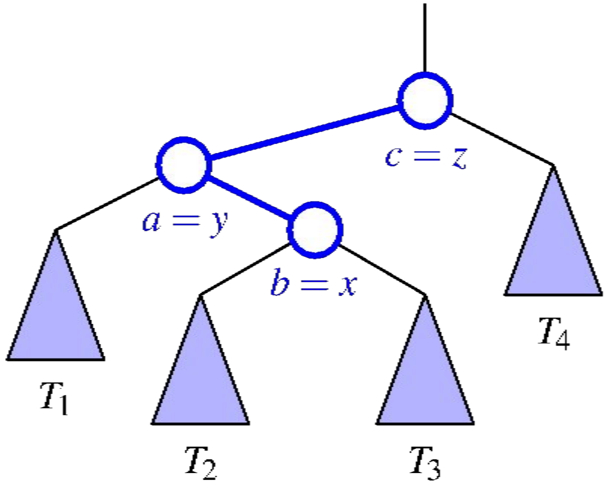



**Left** subtree is too heavy because of **right** subtree of a

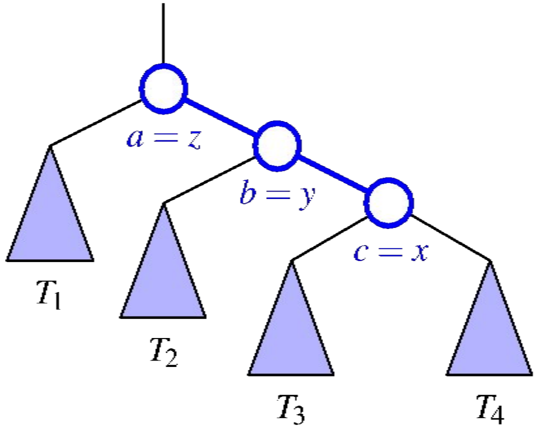
1. Rotate Left about a
2. Rotate Right about c

# Double Rotation

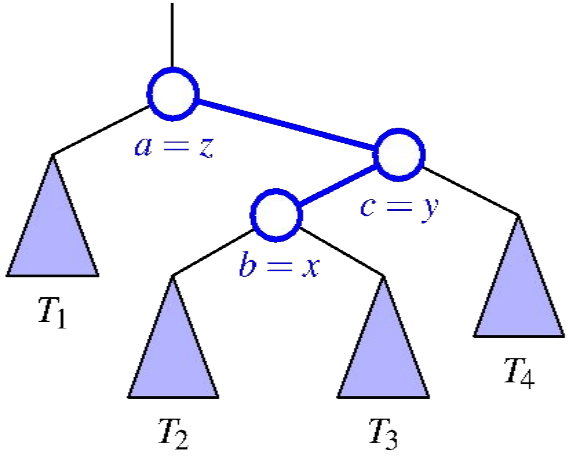
When do we need a double rotation vs a single rotation?




Double rotation




Single rotation




Double rotation

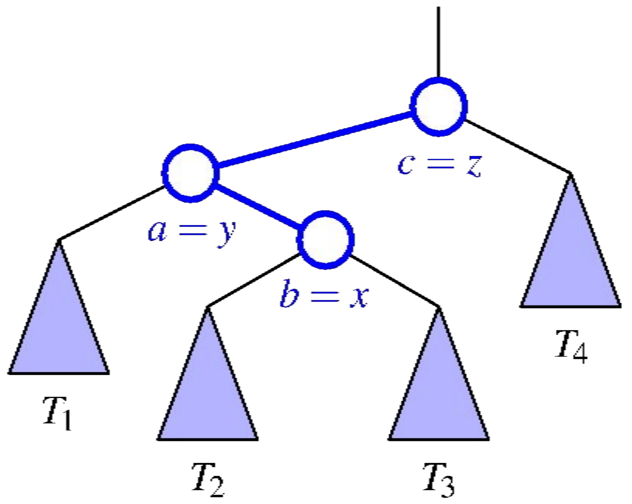
Look for zig-zag pattern!

# Double rotation

When do we need a double rotation?

Left subtree is too heavy on the right side

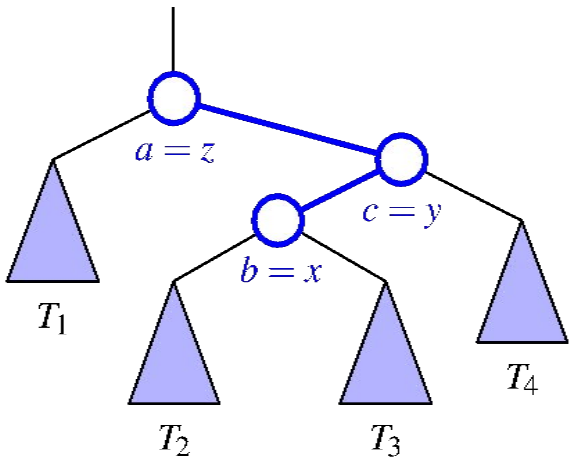
`rotateLeftRight`




OR

Right subtree is too heavy on the left side

`rotateRightLeft`

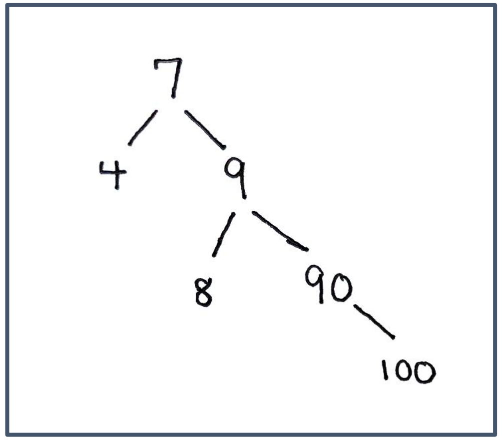


# Double Rotation Code

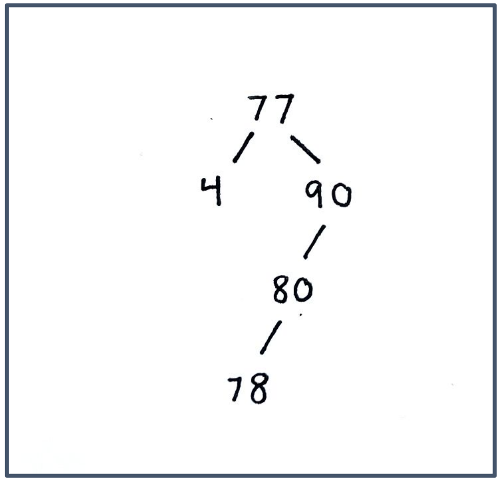
```
def rotateLeftRight(n)
    n.left = rotateLeft(n.left);
    n = rotateRight(n);

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

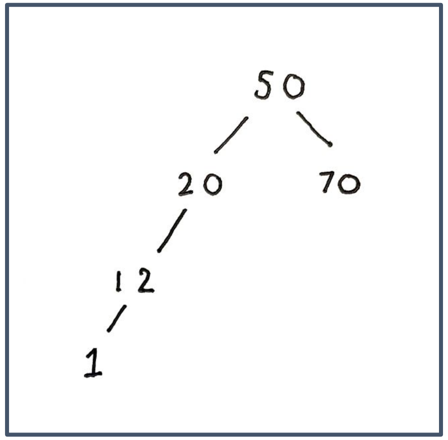
# Examples - which way should I rotate?



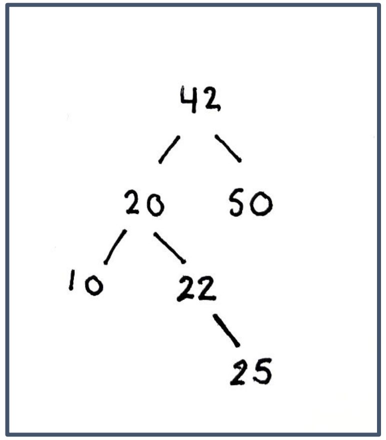

rotateLeft




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)

# Summary

- Quicksort and Mergesort are **recursive O(nlogn)** sorting algorithms

- In quicksort, good pivots are important in achieving O(nlogn) runtime complexity

- HashTable + Quicksort homework due Friday!

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