# CS151 Intro to Data Structures

Balanced Binary Search Trees

# Announcements

HW8 and Lab 10 released

- start with the lab (autograded)
- Due December 11th

Final Exam: Monday 12/15 9:30am-12:30am (Park 159)

Office hours at 4pm

# Outline

1. Warmup

2. 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.

# Balanced Binary Trees

# Properties of a BST

1. Binary: each node has at most 2 children

2. At each node with value k
   a. Left subtree contains only nodes with value lesser than k
   b. Right subtree contains only nodes with value greater than k
   c. Both subtrees are binary search trees

# What can go wrong?

Complexity?

**Search**

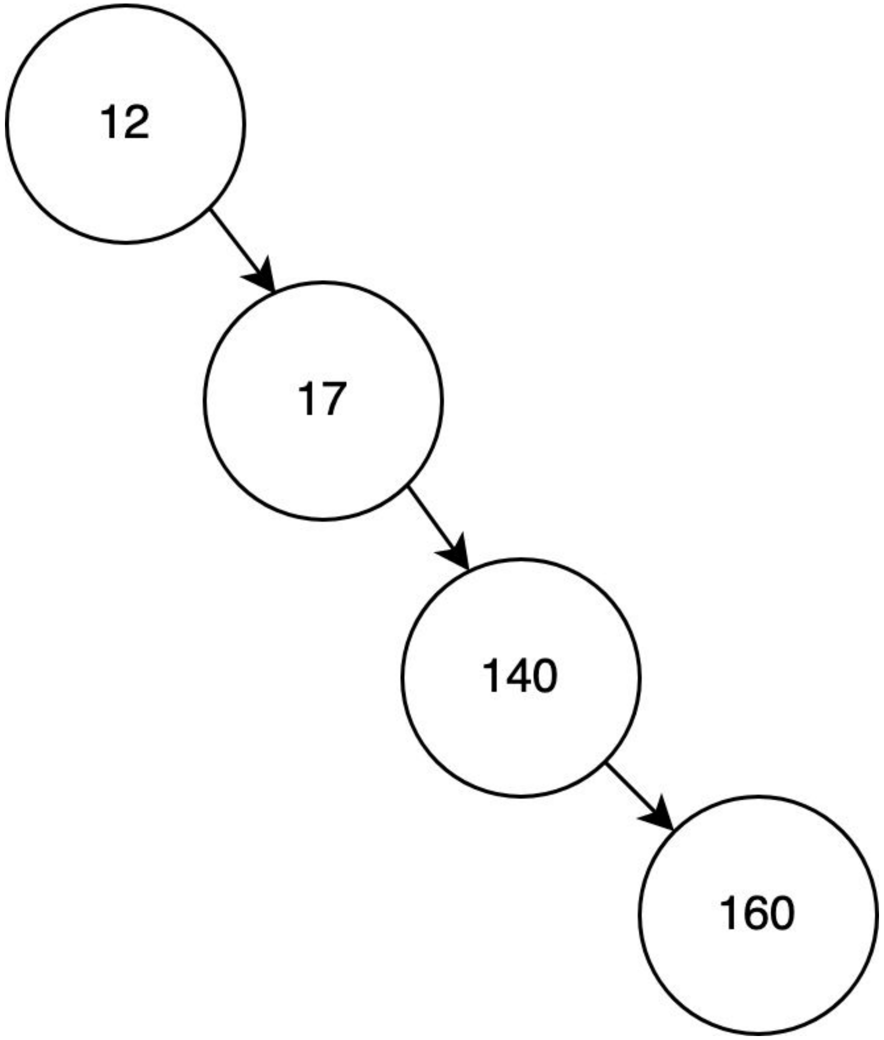
O(n)

**Insertion:**

O(n)

**Deletion:**

O(n)



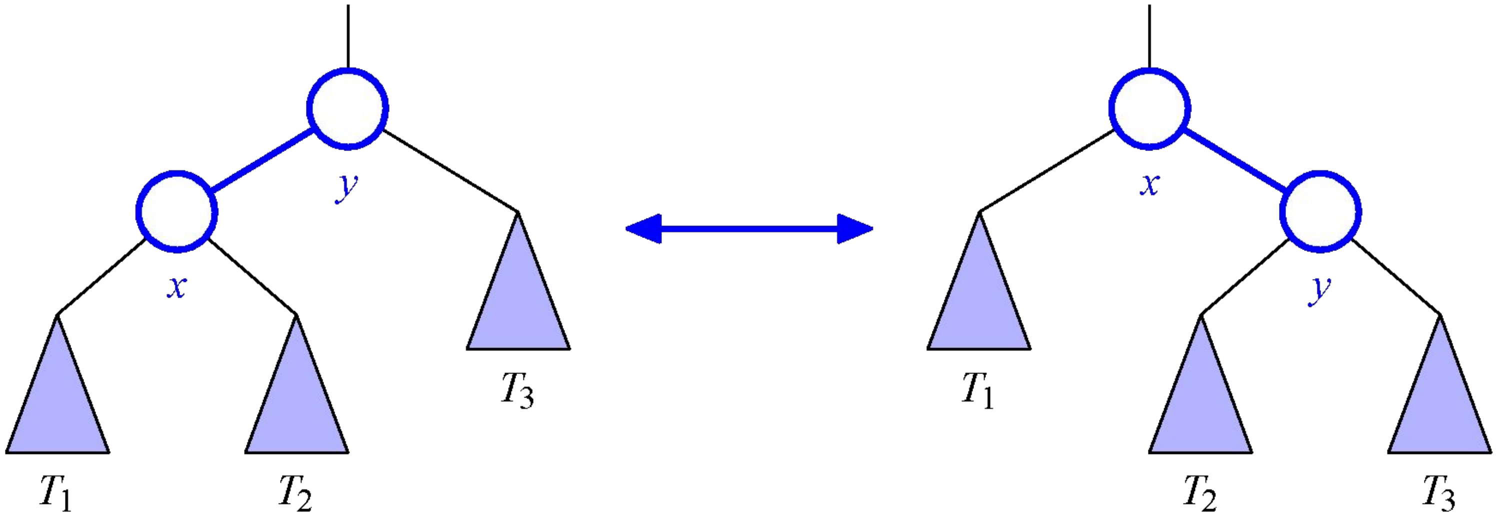
# Balanced Binary Trees

- 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

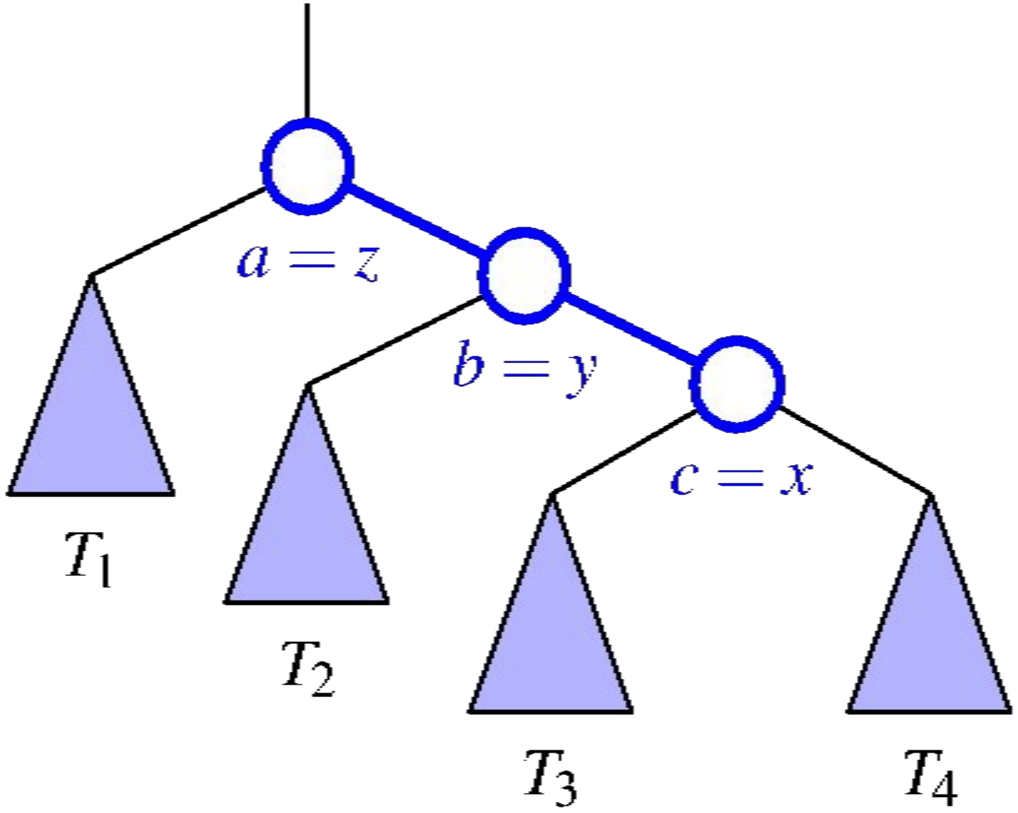


# Rotation Operation

- Used to maintain balance

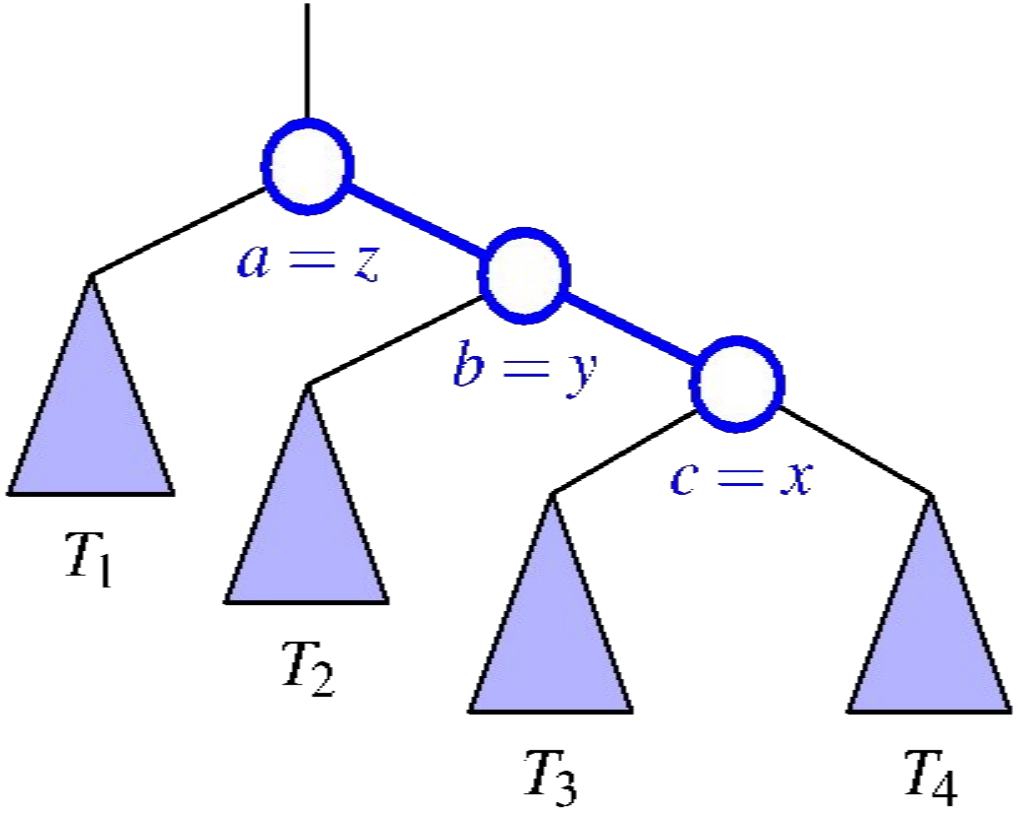
- When should **rotate** be invoked?
  - Difference of heights of left and right subtrees at any node is > 1

# 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

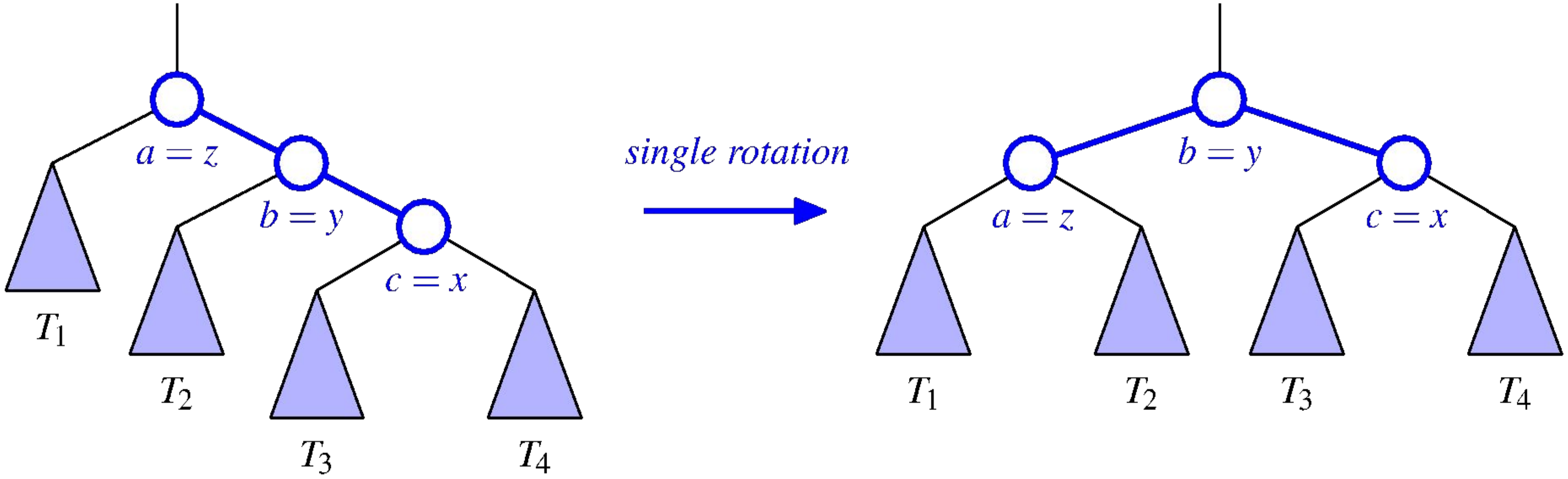


Right subtree is too large!

How can we rotate to fix this?

What should we make the root?

# Single Rotation (around $z$)



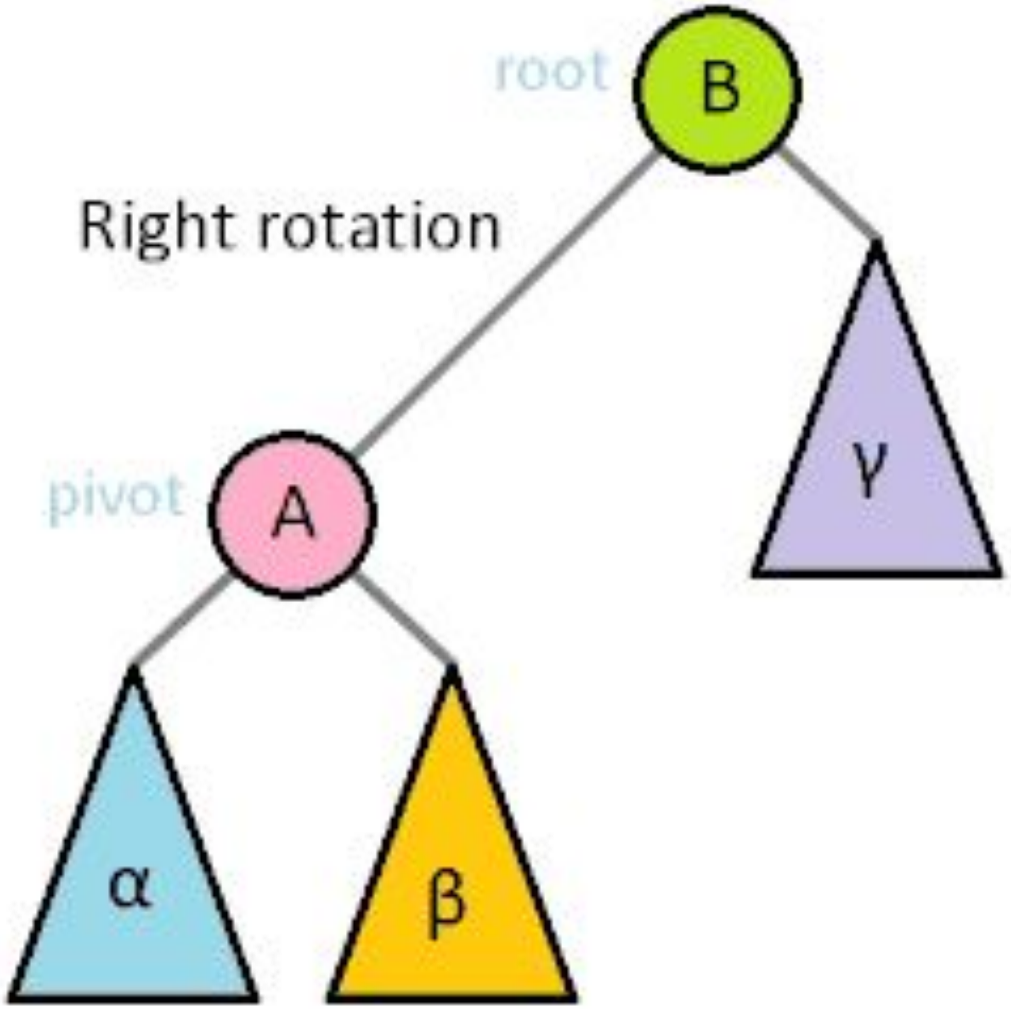
# Rotations

Right rotation:

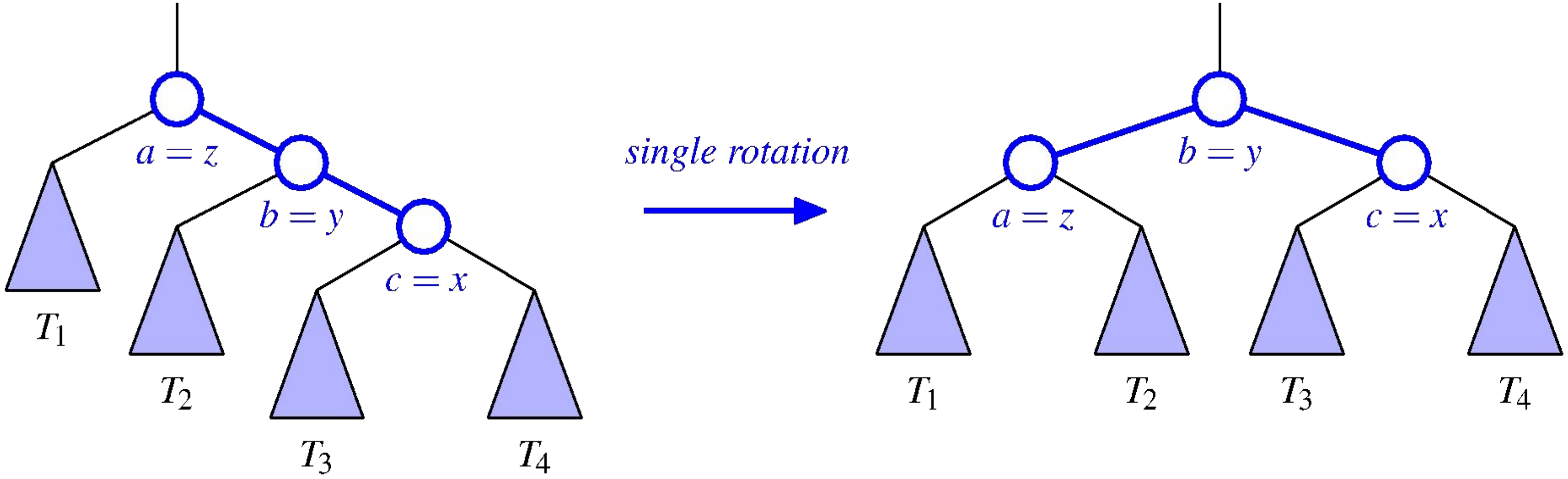
- Performed when left side is heavier
- left child becomes root

Left rotation:

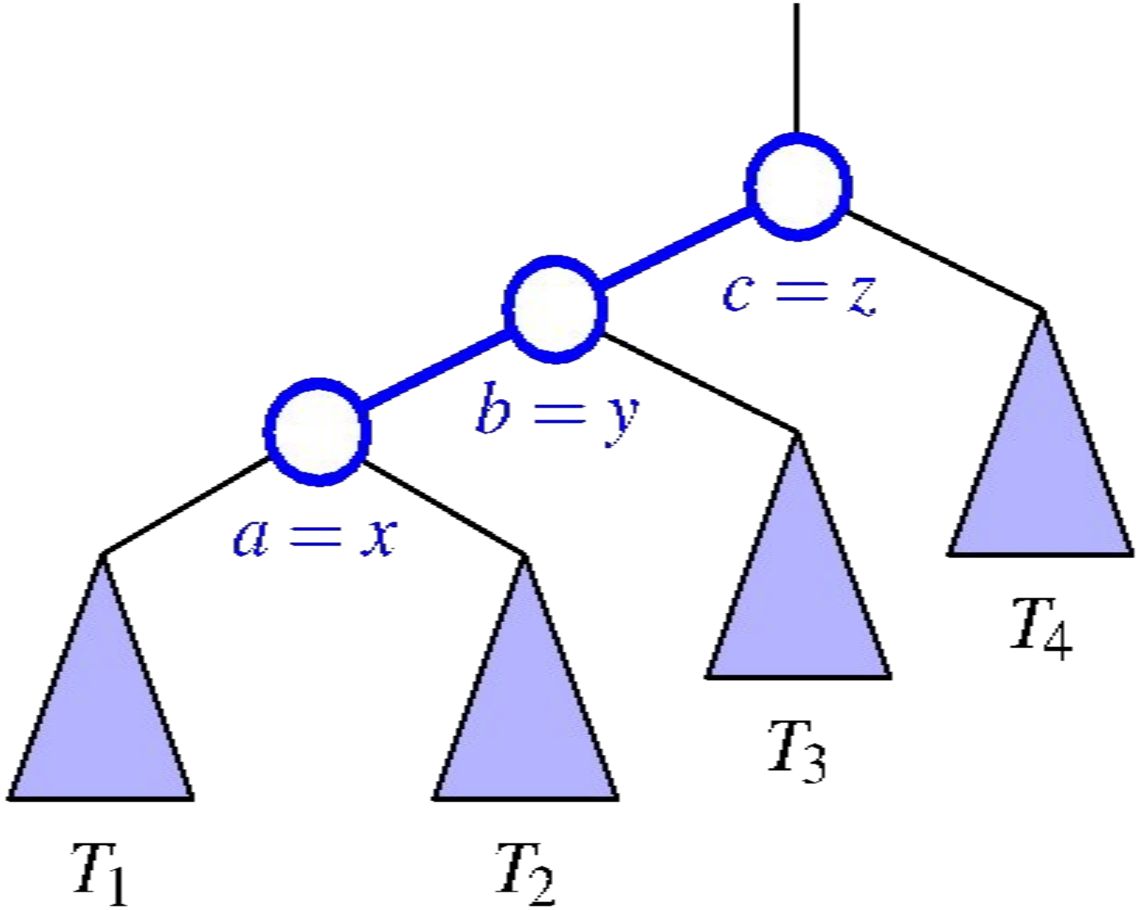
- Performed when right side is heavier
- right child becomes root



# Left or Right rotation?



# Example 2:

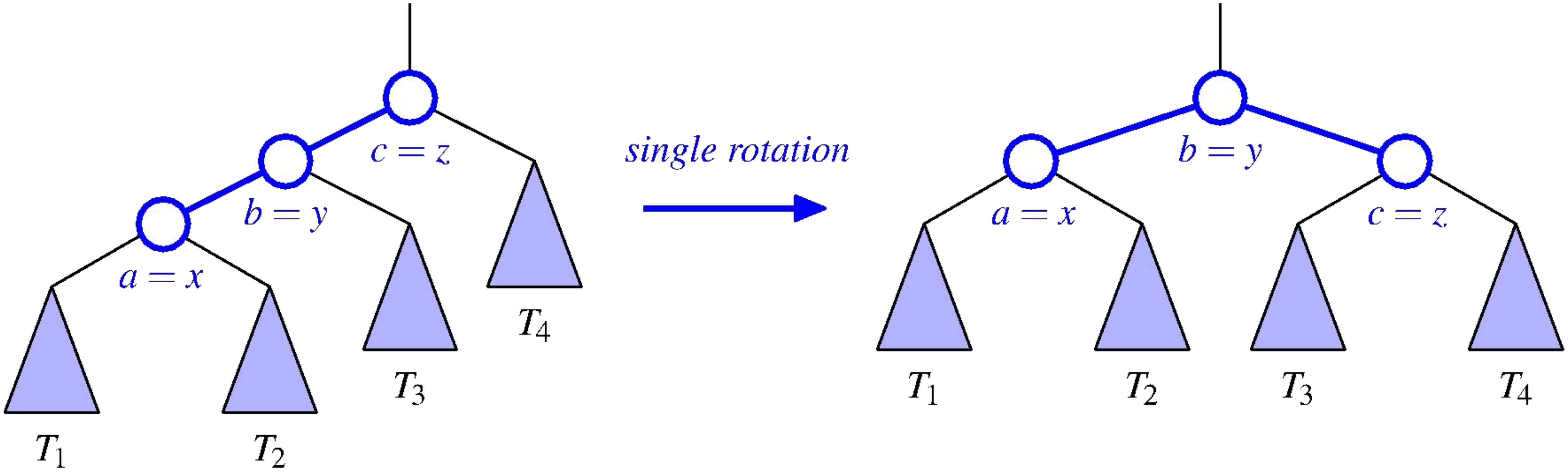


Should we do a left or right rotation?

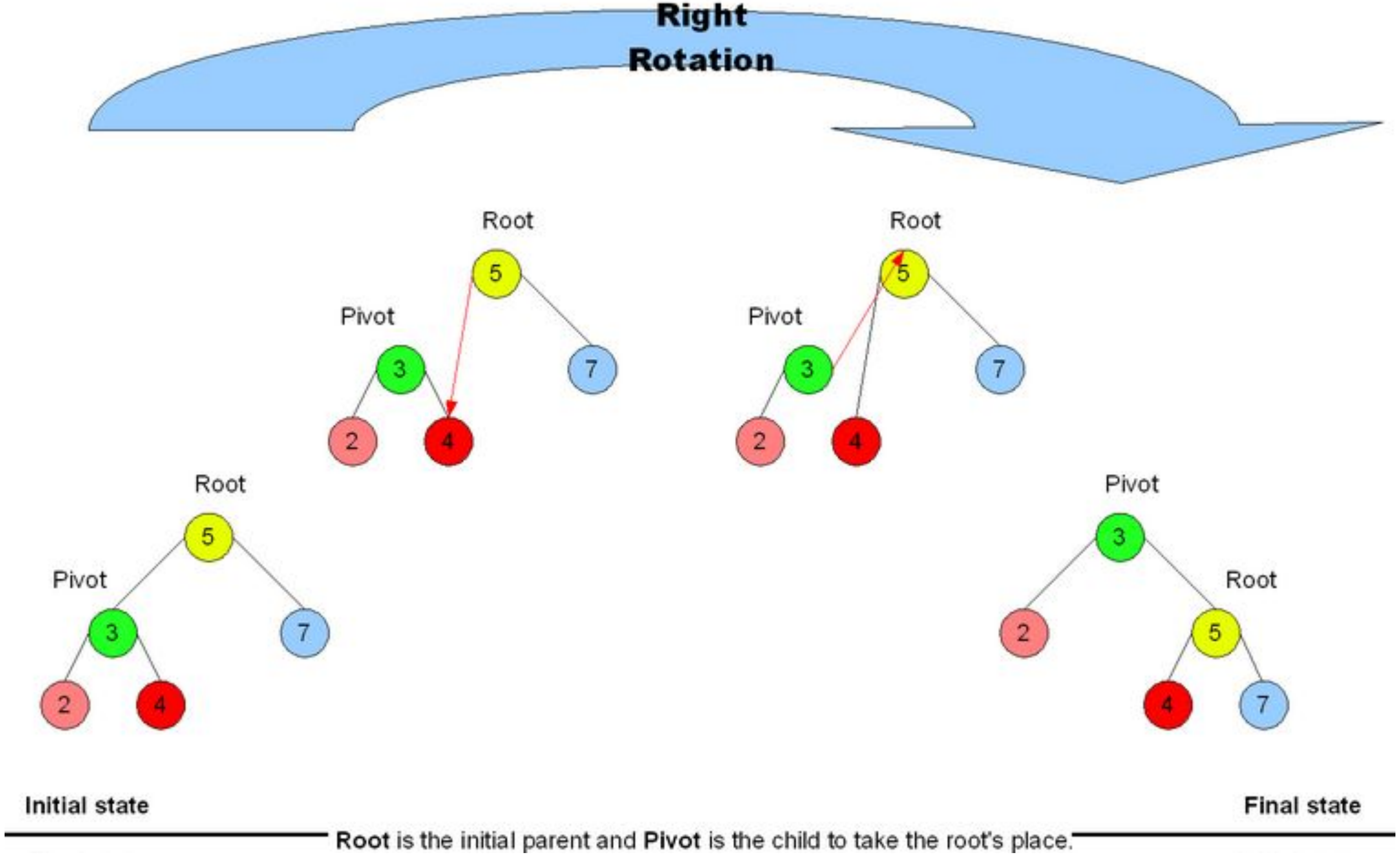
What will become the root?

Let’s draw what it will look like after rotation

# Example 2: Rotate Right



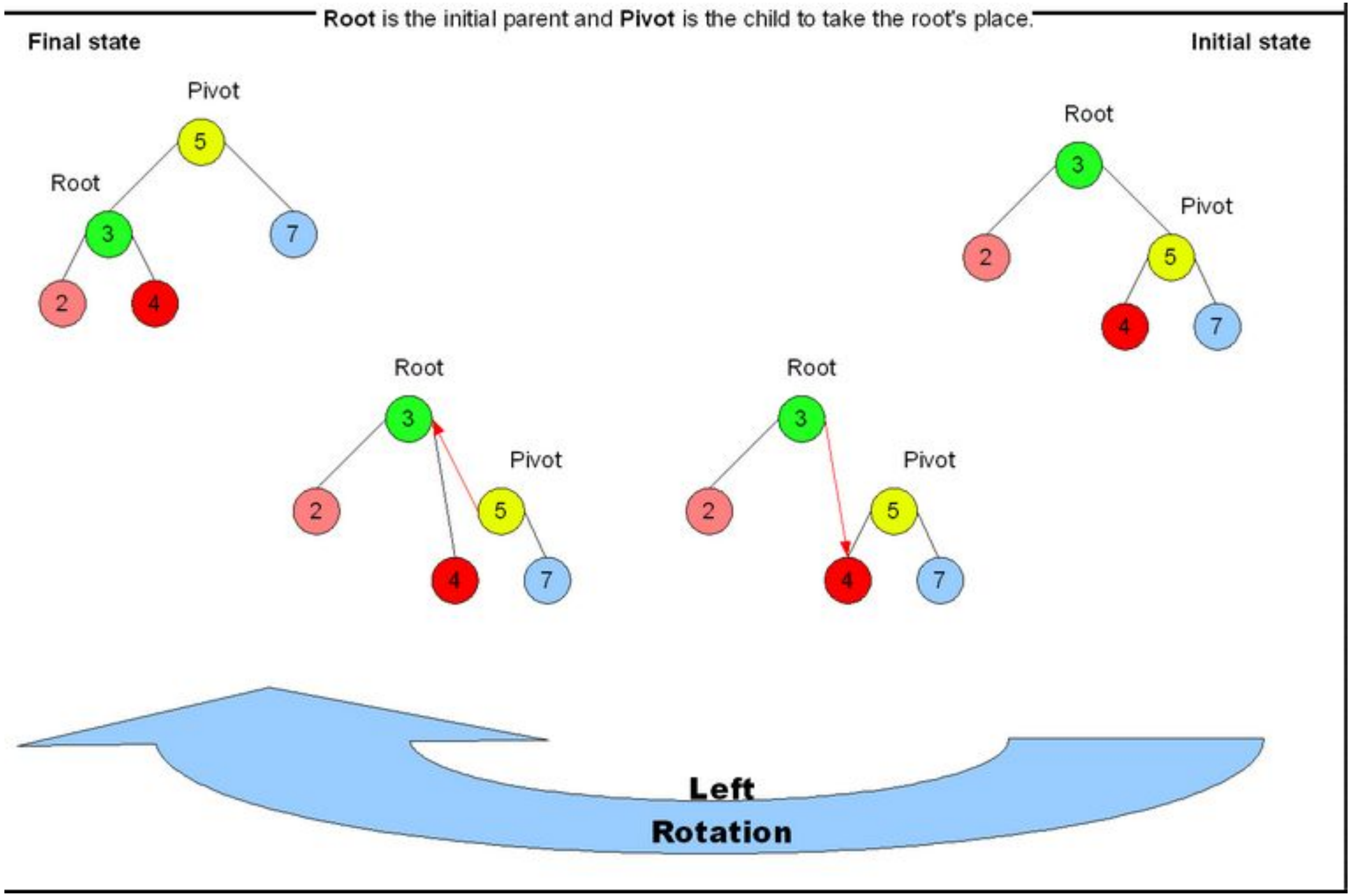
# RotateRight Algorithm



**Root** is the initial parent and **Pivot** is the child to take the root's place.

```
1. Root.left =
   Pivot.right

2. Pivot.right =
   root
```

# RotateLeft Algorithm



```
1. Root.right =
   Pivot.left

2. Pivot.left =
   root
```

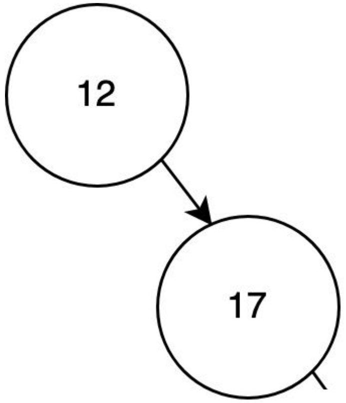
# Example:

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?

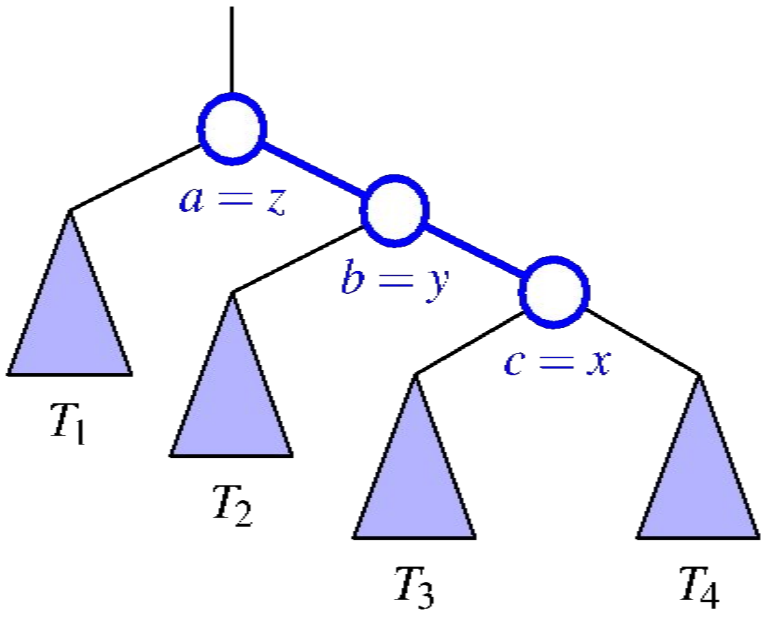
- O(1)

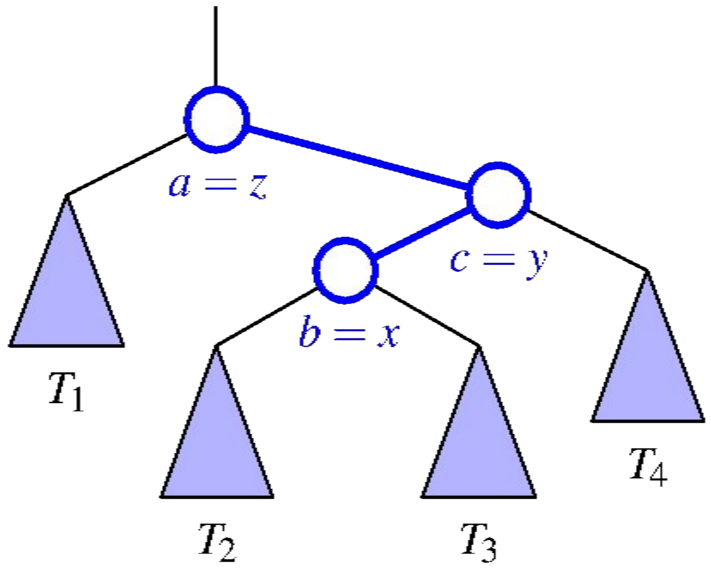
Constant time... we’re just updating links

# Double Rotation

Sometimes a single rotation is not enough to restore balance

# Double Rotation

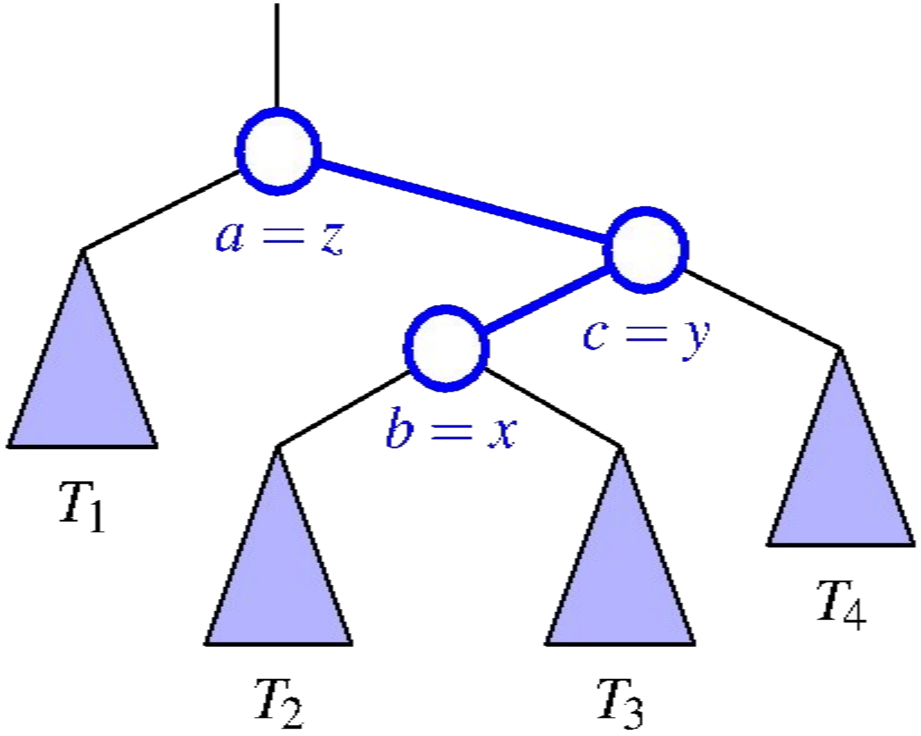




**Right** child of a is too heavy.. because
**Right subtree** of b is too heavy..
Single Left rotation on the root needed

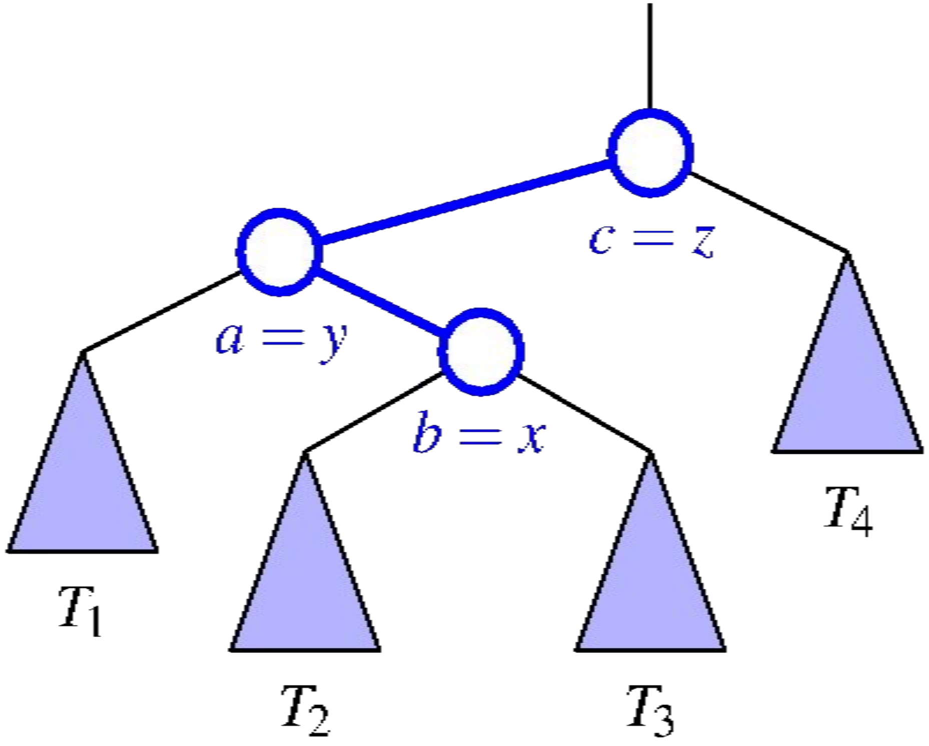
**Right** child of a is too heavy... because
**Left subtree** of c is too heavy
**Is a single rotation enough?**

# Double Rotation

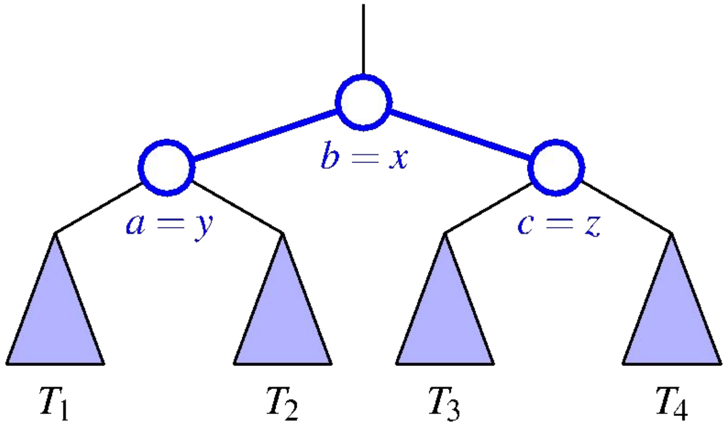


1. **Rotate Right** at c because right subtree of root is too heavy
2. **Rotate Left** at the root (a)

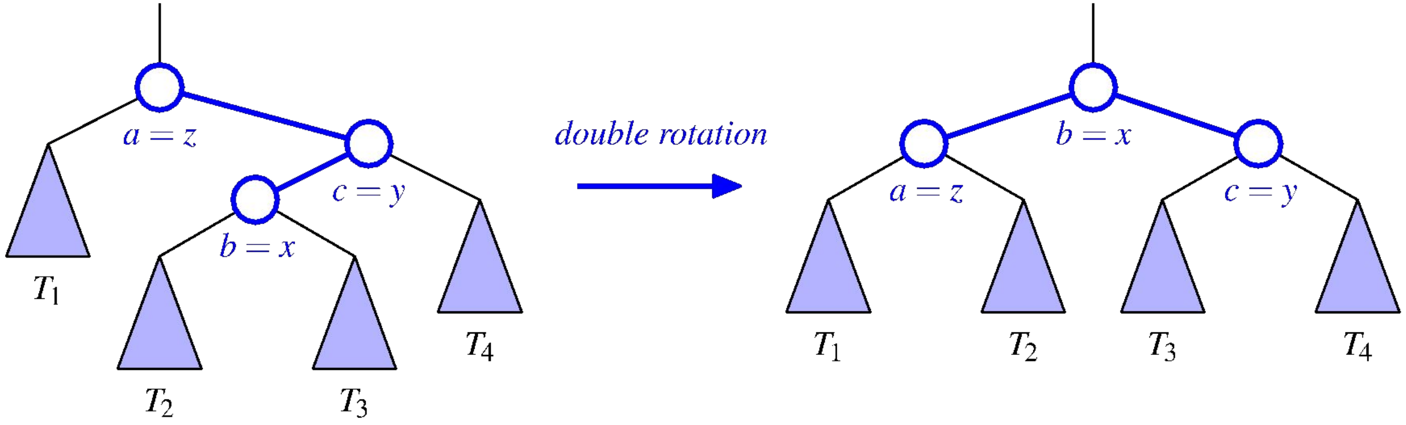
# Double Rotation Example 2:



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

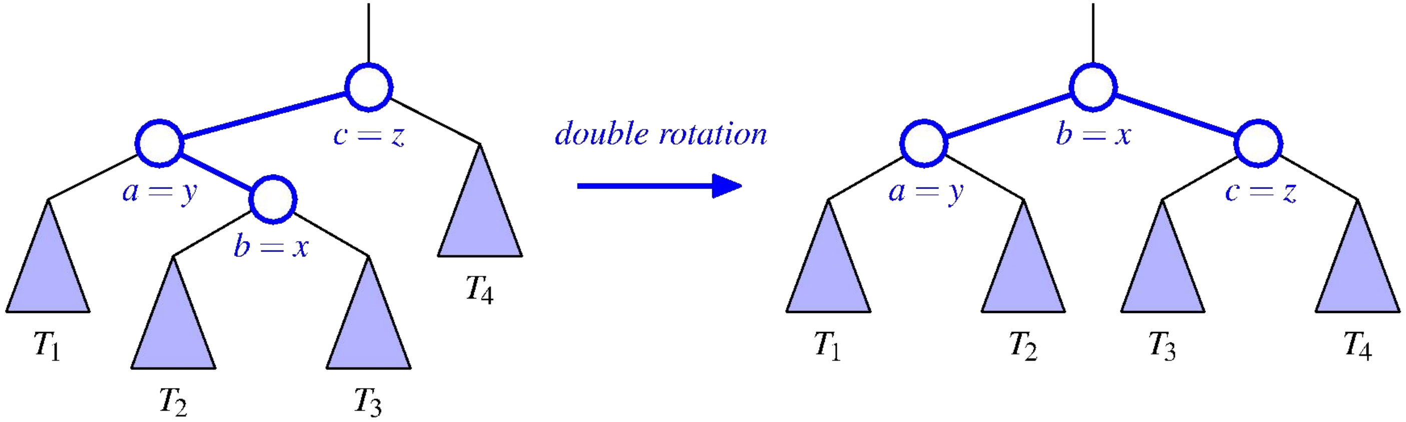


# Double Rotations



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

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

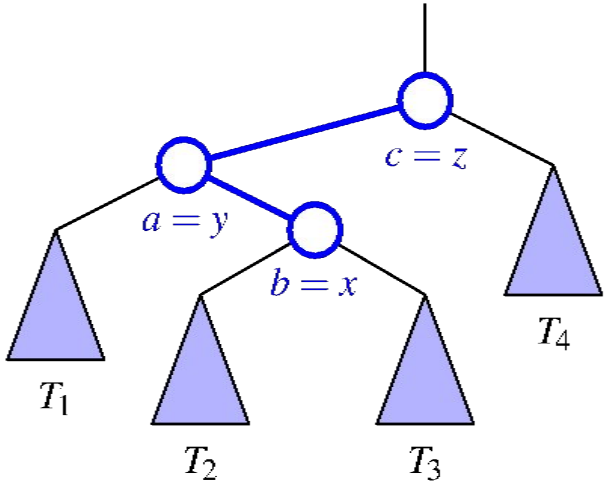


**Left** subtree is too heavy because of **right** subtree of a

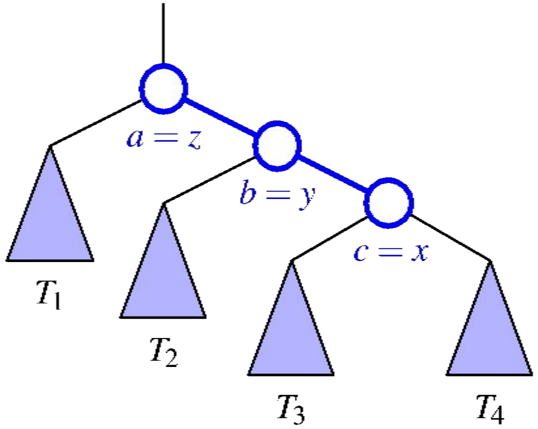
1. Rotate Left about a
2. Rotate Right about c

# Double Rotation

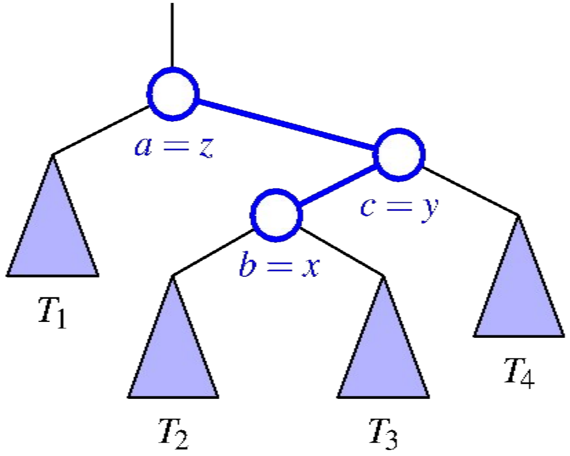
When do we need a double rotation vs a single rotation?



Double rotation



Single rotation



Double rotation

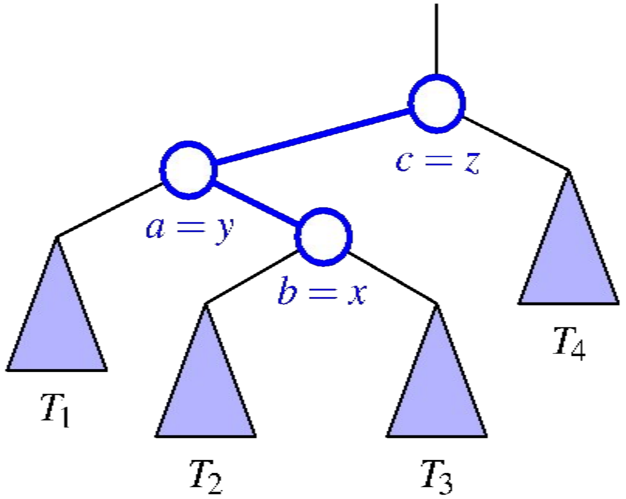
Look for zig-zag pattern!

# Double rotation

When do we need a double rotation?

Left subtree is too heavy on the right side

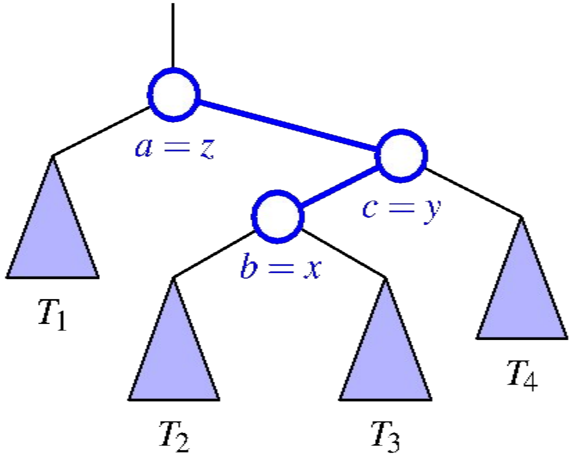
`rotateLeftRight`



OR

Right subtree is too heavy on the left side

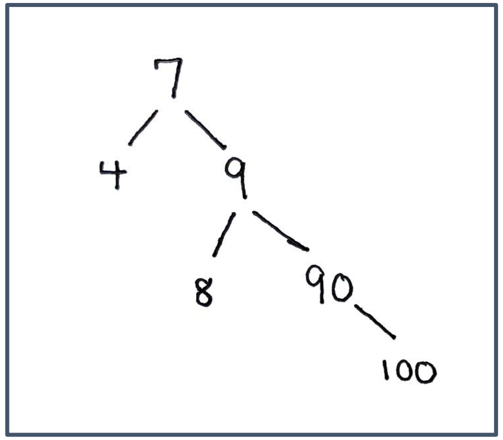
`rotateRightLeft`

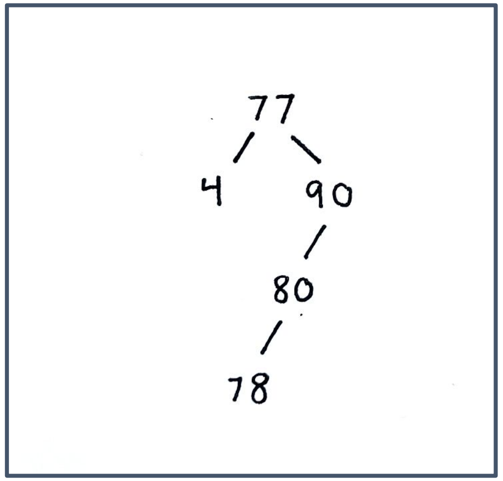


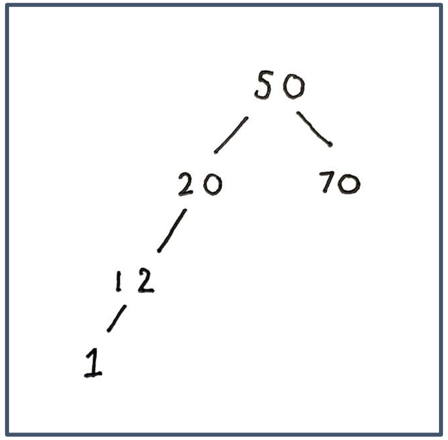
# Double Rotation Code

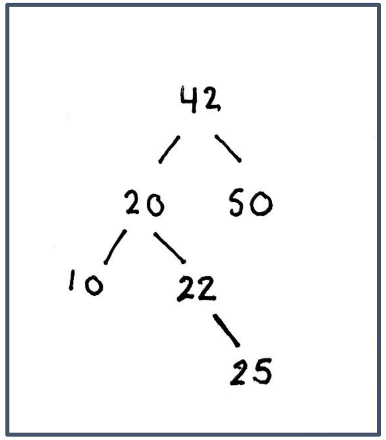
```
def rotateLeftRight(n)
    n.left = rotateLeft(n.left);
    n = rotateRight(n);

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)

# Summary

Start HW8 and Lab10!

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