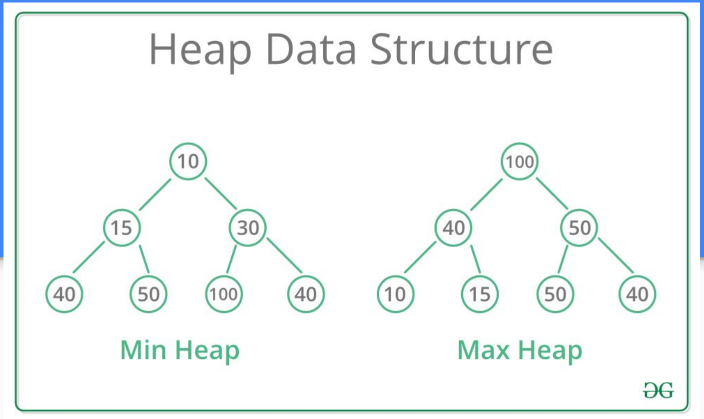
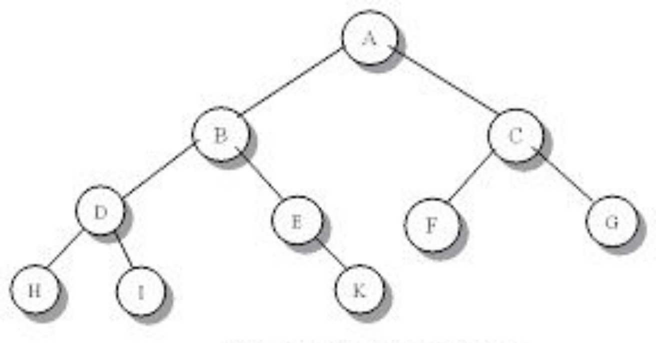
# Lab 10: BST and AVL

CSCI104

# REMEMBER: Heaps

- COMPLETE d-ary tree
  - All levels except the last are completely filled
  - All leaves in last level are to the left side
- Every parent is “better” than both of its children
- Min Heap: node is less than or equal to all children
- Max Heap: node is greater than or equal to all children

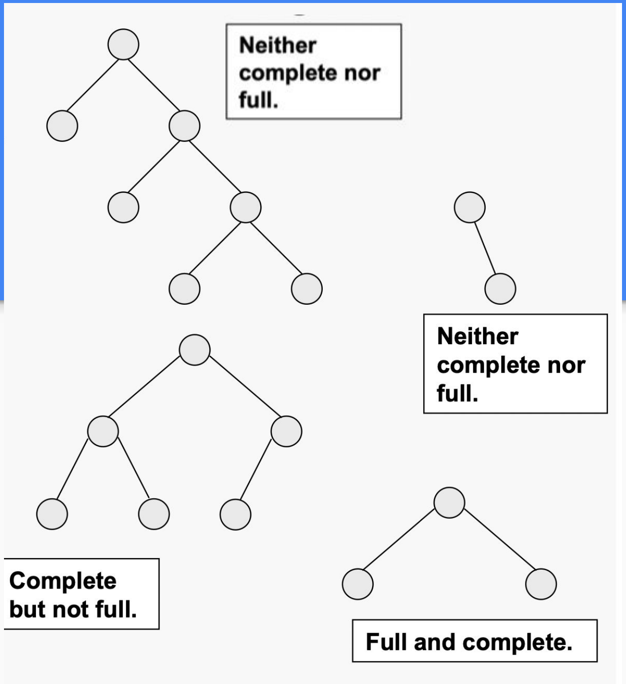



Could this be a heap??

# Binary Search Trees (BST)

- Not necessarily a complete or full tree
- Left children (left subtree) hold values LESS THAN or equal to parent's values
- Right children (right subtree) hold values GREATER THAN parent's value



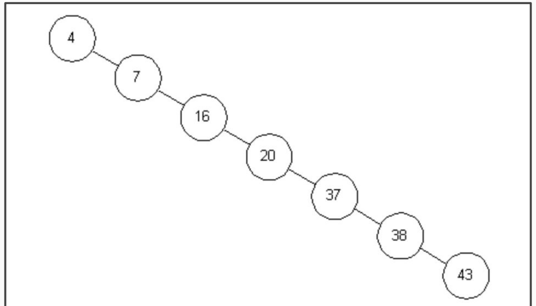




Figure 3-3: An Unbalanced Binary Search Tree

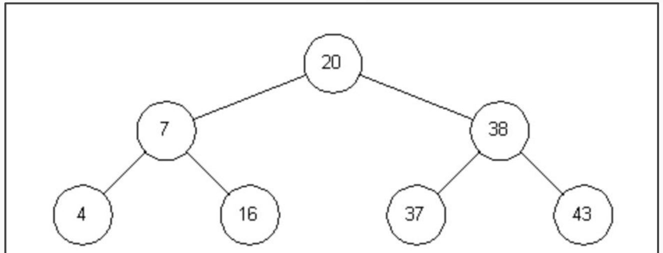



Figure 3-2: A Binary Search Tree

# Traversals: Pre-Order, In-Order, Post-Order

- All traverals operate on EVERY node eventually–just in different orders
    - "Pre" : visit the parent "pre-" (before) visiting left and right sub-trees.
    - "In" : visit the parent "in"-between visiting left and right sub-trees.
    - "Post": visit the parent "post-" (after) visiting left and right sub-trees.

### Pre-Order Traversal

```
// Operate on current node
// Recurse left
// Recurse right
// return
```

### In-Order Traversal

```
// Recurse left
// Operate on current node
// Recurse right
// return
```

### Post-Order Traversal

```
// Recurse left
// Recurse right
// Operate on current node
// return
```

# Traversals in C++

For a BST, what is special about operating on elements using an in-order traversal? If we were printing integers using this traversal, what would the output look like?

```
void pre_order(Node* node) {
    if (node == nullptr) return;
    print(node);
    pre_order(node->left);
    pre_order(node->right);
}

void in_order(Node* node) {
    if (node == nullptr) return;
    in_order(node->left);
    print(node);
    in_order(node->right);
}

void post_order(Node* node) {
    if (node == nullptr) return;
    post_order(node->left);
    post_order(node->right);
    print(node);
}
```

# Why BSTs? SEARCHING!

- Enable (potentially) faster searching
- Why do we say potentially? What is an example where the search is slow, even if it’s a valid BST?

# Why BSTs? SEARCHING!

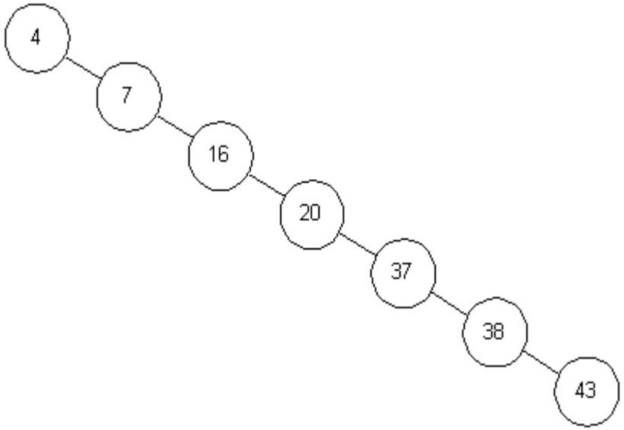



Figure 3-3: An Unbalanced Binary Search Tree

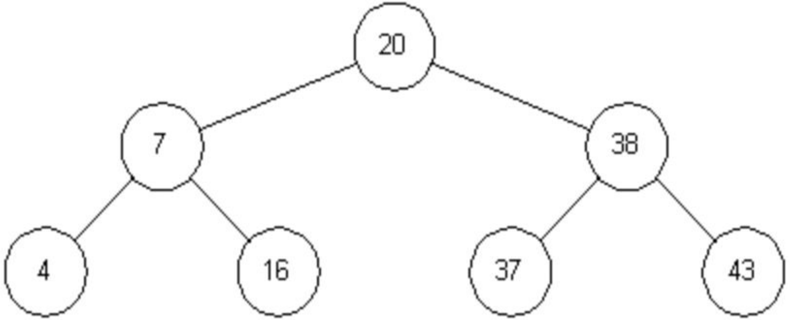



Figure 3-2: A Binary Search Tree

Slower search: O(n)
Basically like a linked list

Faster search: O(logn)

# Search Function

- Can do it iteratively or recursively

To search for key `X` in a BST, we compare *X* to the current node.

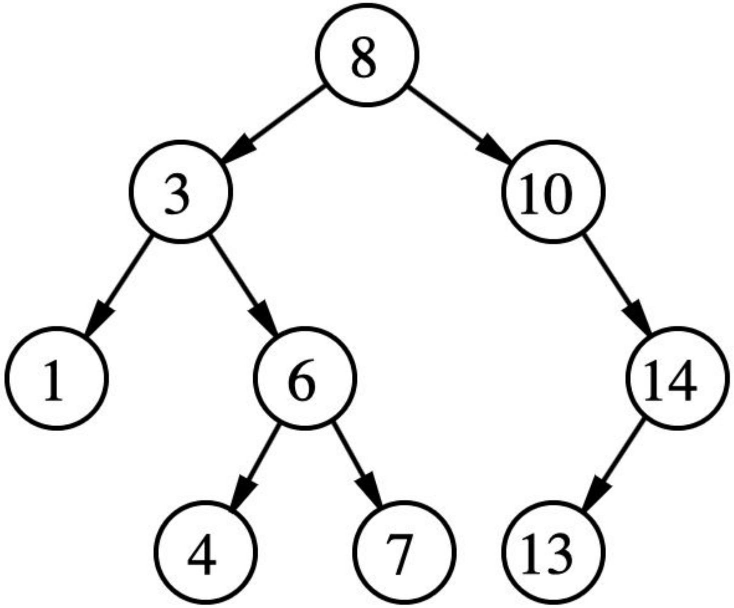
- If the current node is null, `X` must not reside in the tree.
- If `X` is equal to the current node, simply return the current node.
- If it is less than the current node, we check the left subtree.
- Else, it must be greater than the current node, so we check the right subtree.

Or, in code:

```
// Finds the node with value == val inside the bst. Returns nullptr if not found
Node* find(Node* root, int val) {
    if (root == nullptr) return nullptr;
    if (root->val == val) return root;
    if (root->val > val) return find(root->left, val);
    return find(root->right, val);
}
```

Recursive example

# Search Example




Operation: `find(6)` // We begin at the root

Let's walk through this:

- Current node = 8, 6 < 8, therefore go left.
- Current node = 3, 6 > 3, therefore go right.
- Current node = 6, 6 = 6, we've found the node.

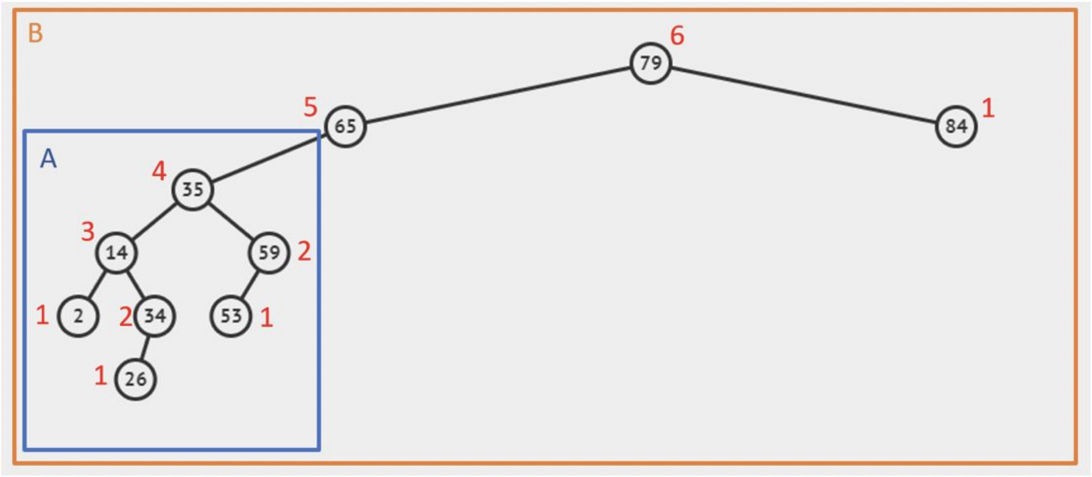
Operation: `find(0)` // We begin at the root

Let's walk through this one too:

- Current node = 8, 0 < 8, therefore go left.
- Current node = 3, 0 < 3, therefore go left.
- Current node = 1, 0 < 1, therefore go left.
- Current node = null. 0 is not in the tree.

# Balanced Binary Tree

- **Height-balancing property: heights of each subtree differ by no more than 1**
- Avoids the slower search times!
- Keeps the height of the tree log(n)




A is balance, B is not

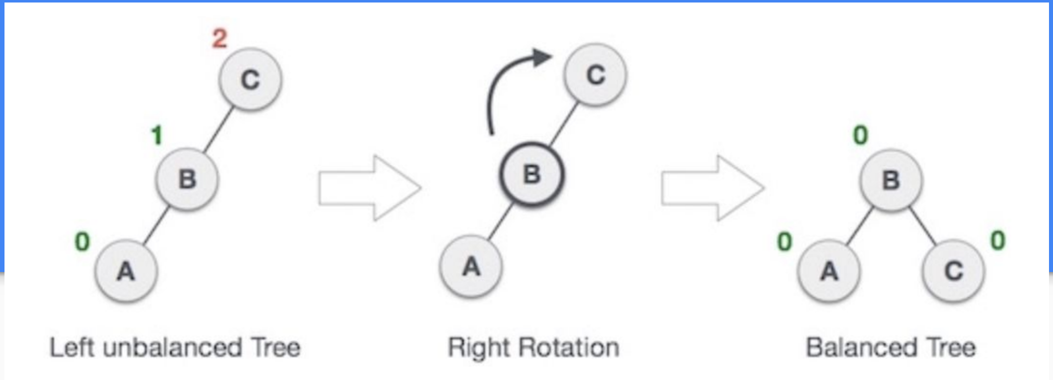
# Maintaining BST Property

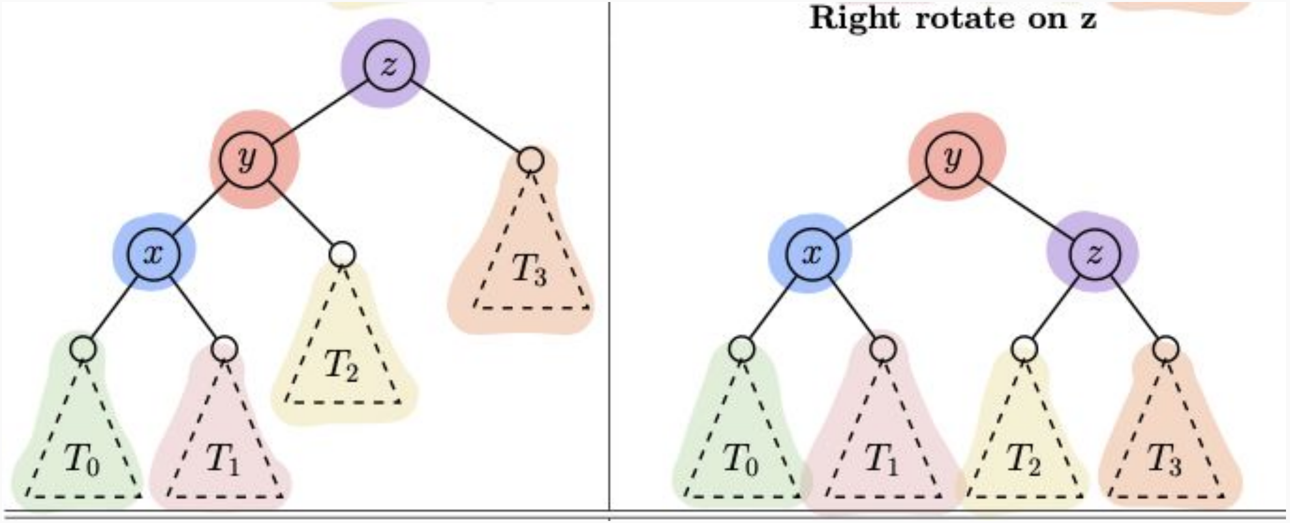
- REMEMBER: BST Property = left subtree node keys less than parent's and right subtree node keys greater than parent's
- Maintained by **smart** insertion and deletion
- Insert function
    - Traverse the tree based on key to be inserted
    - Insert once you encounter a situation where you cannot traverse further
- Remove function
    - Need to choose which node to promote
    - If node you want to remove has 0 children: just remove it
    - If node you want to remove has 1 child: promote the child of the node
    - If node you want to remove has 2 children: swap with its predecessor OR successor

# Self-Balancing BSTs

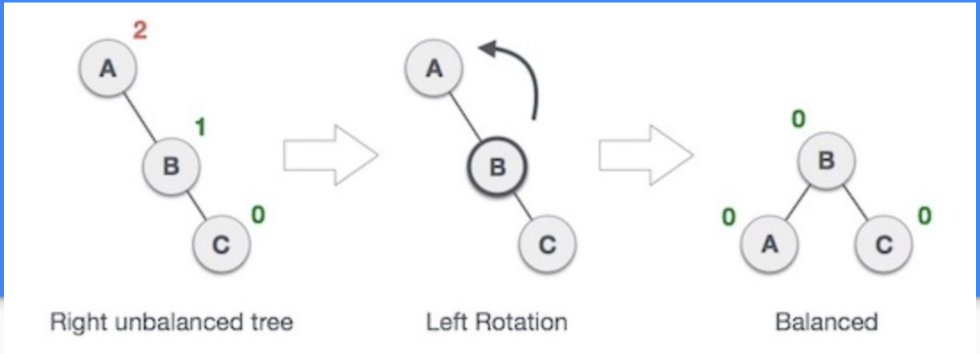
- We will be focusing on AVL trees
- You keep the tree balanced even after insertions or deletions
- This involves using rotations!
    - Foundation of AVL trees

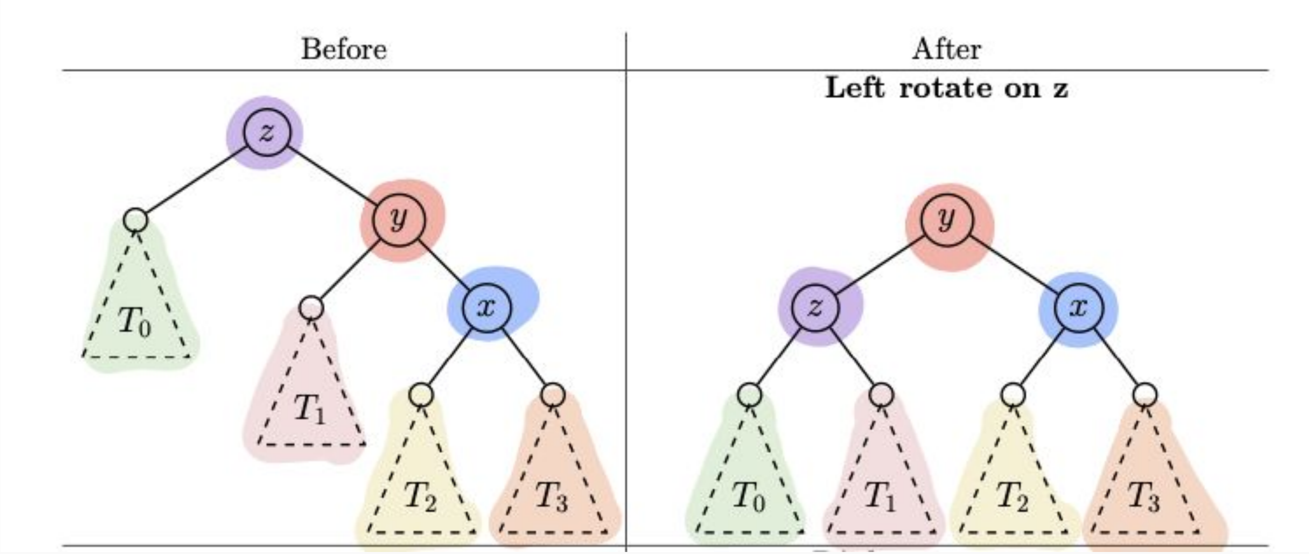
# Single Rotations RIGHT



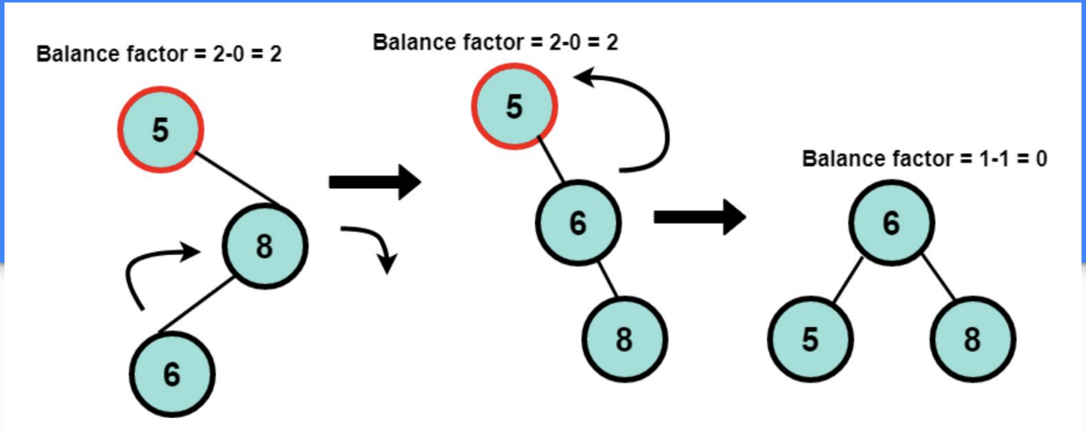



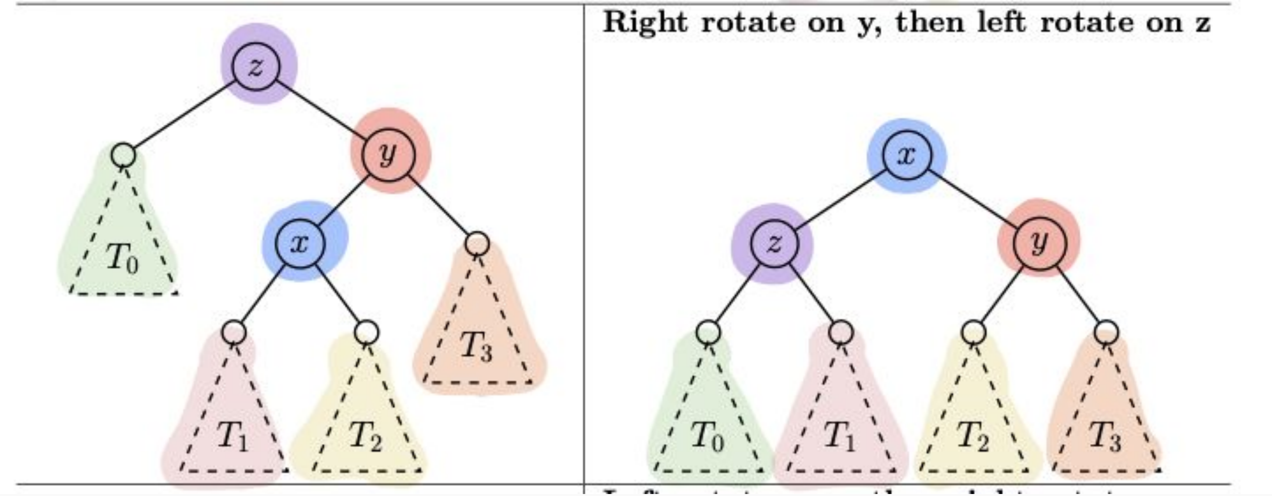
# Single Rotations LEFT



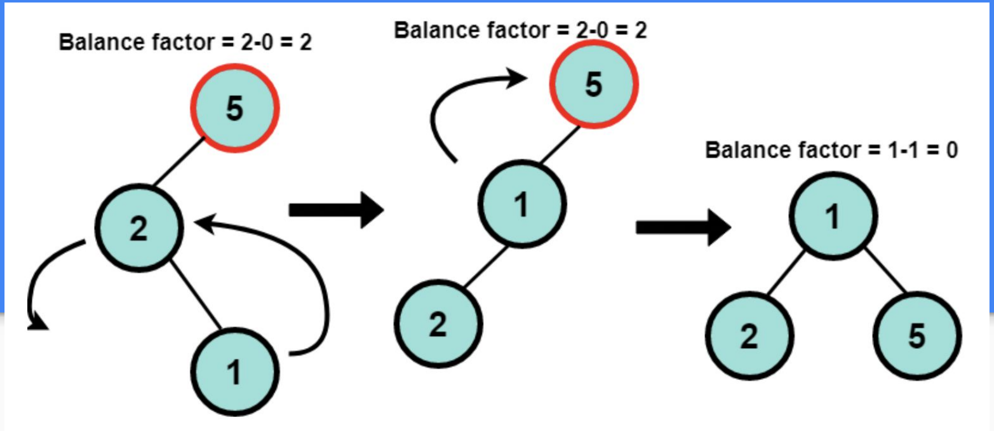



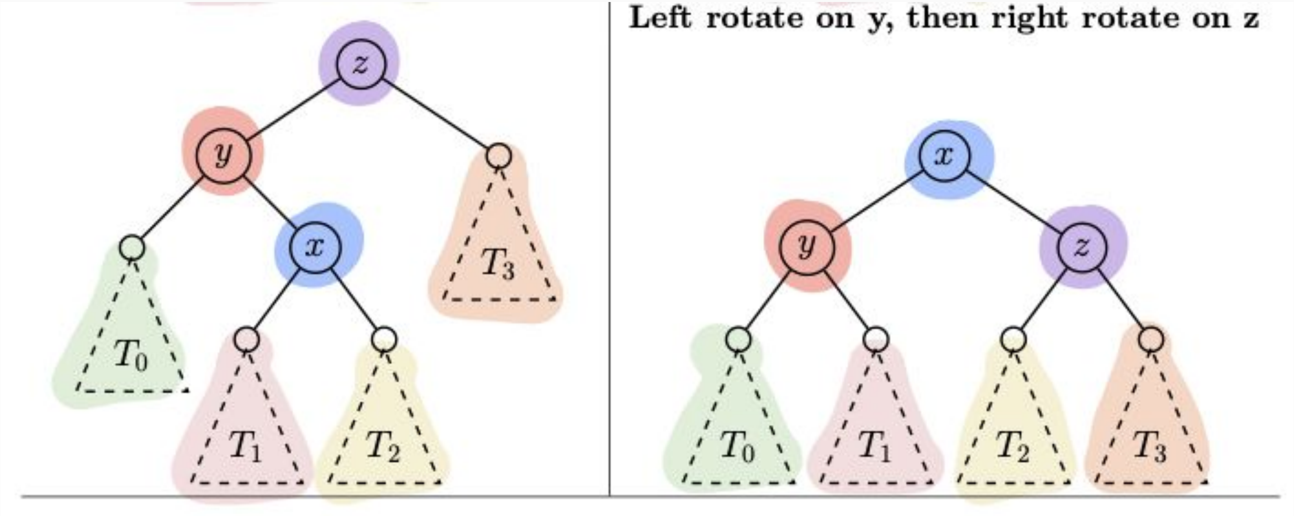
# Double Rotations
# RIGHT LEFT



# Double Rotations
# LEFT RIGHT



# AVL Insert and Remove

- Insert
  - Insert as you would in a BST
  - Fix the tree if it is unbalanced after inserting the node (ROTATION)
    - Need at most 1 rotation (either a single or double rotation)
- Remove
  - Remove as you would in a BST
  - Keep traversing up the tree and fixing tree if unbalanced (ROTATIONS)
    - You may need multiple rotations to fully fix the tree

# The Lab

- Exercise 1
  - Draw or type out operations on tree
- Exercise 2
  - Write isBalanced function
  - **NOT NEEDED TO GET CHECKED OFF**
  - This function is also part of the PA
  - We encourage you to use a single traversal for this function
  - Cannot work in groups, we will not be going over this solution

Initial Tree

```
                 13
          +---------+---+
          10            15
      +---+---+         +---+
      5       11            16
  +---+---+
  4       8
```

Insert 14

Insert 3

Remove 3

Remove 4

Write a function to determine whether a binary tree is height-balanced or not.

- A binary tree in which the depth of the two subtrees of every node never differs by more than 1.

```
bool isBalanced(Node *root)
```