# AVL Trees — prototypes of Balanced Trees

In this lecture, unless otherwise mentioned, all trees will be *binary*. Moreover, the *insertion* and *deletion* algorithms will be the standard ones for binary search trees.

Binary search trees can suffer from becoming *unbalanced* after a sequence of adds and deletes.

Deletions on balanced trees can lead to left-heavy trees.

A major embarrassment is that when a binary search tree is constructed from an already sorted sequence of keys, we get a long skinny tree that is isomorphic to a linear list.

## Binary Search Tree Properties

A binary search tree has the following property: For each node

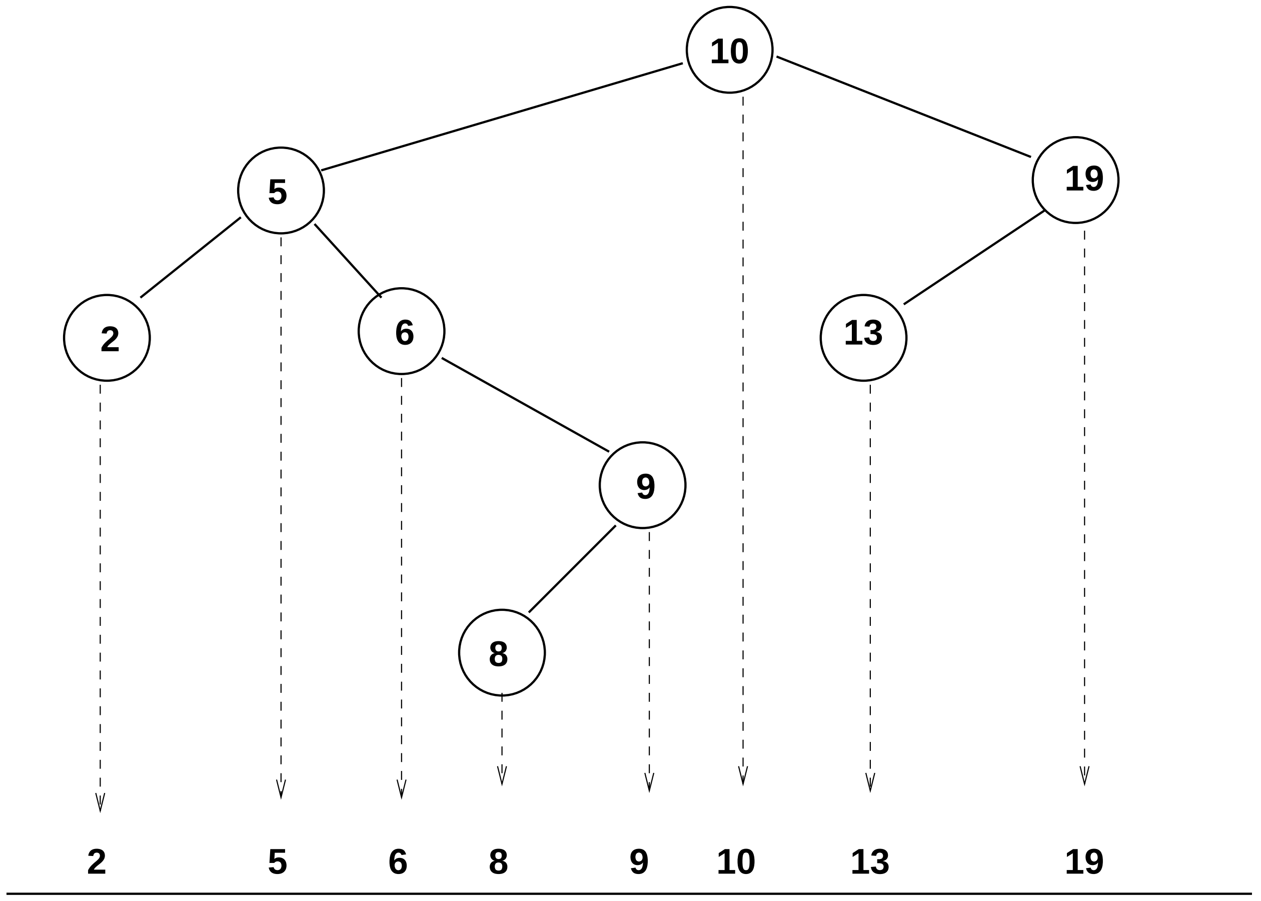
$$key(node.left.\{left, right\}^*) < key(node) \tag{1}$$

$$key(node.right\{left, right\}^*) > key(node) \tag{2}$$

From this one can show the following property:

*Inorder projection*: Inorder visit of a binary search tree in yields the sorted sequence of keys.

# Inorder projection visualized



**Inorder traversal : LeftSubTree − Node − RightSubTree**

## Balanced Search Trees

If there is a way to *maintain* or *restore* the balance in search trees, then we may be able to reduce worst case insert, delete and search times to $O(N\ log\ N)$.

This is possible. The currently popular (and efficient) ways to do this are:

- Red-Black Trees
- A-A Trees
- Splay Trees

However, all of them are *descendants* of *AVL* trees, and the underlying concepts and algorithms are best introduced via AVL trees.

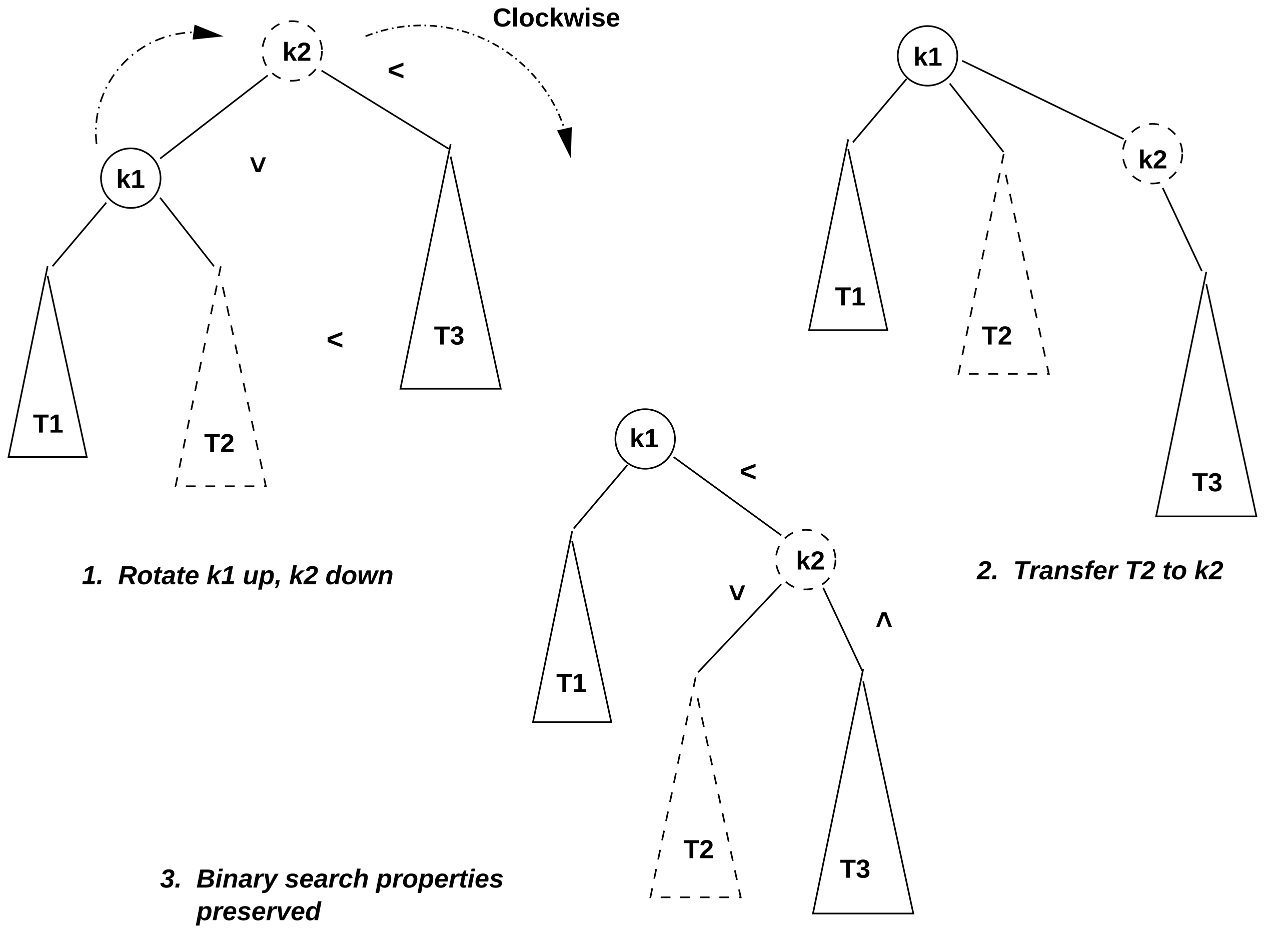
## Rotation and Subtree Transfer

The key ideas in all the re-balancing algorithms are the following:

- Allow a small slack in imbalance to postpone re-balancing
- In rebalancing, we may have to “rotate” a subtree, and transfer a subtree to another parent
- This may have to be repeated

To trigger off these actions, some count of imbalance has to be kept. It is preferable that these counts be “local”, so that updates are simple.

## The Single Rotation Idea

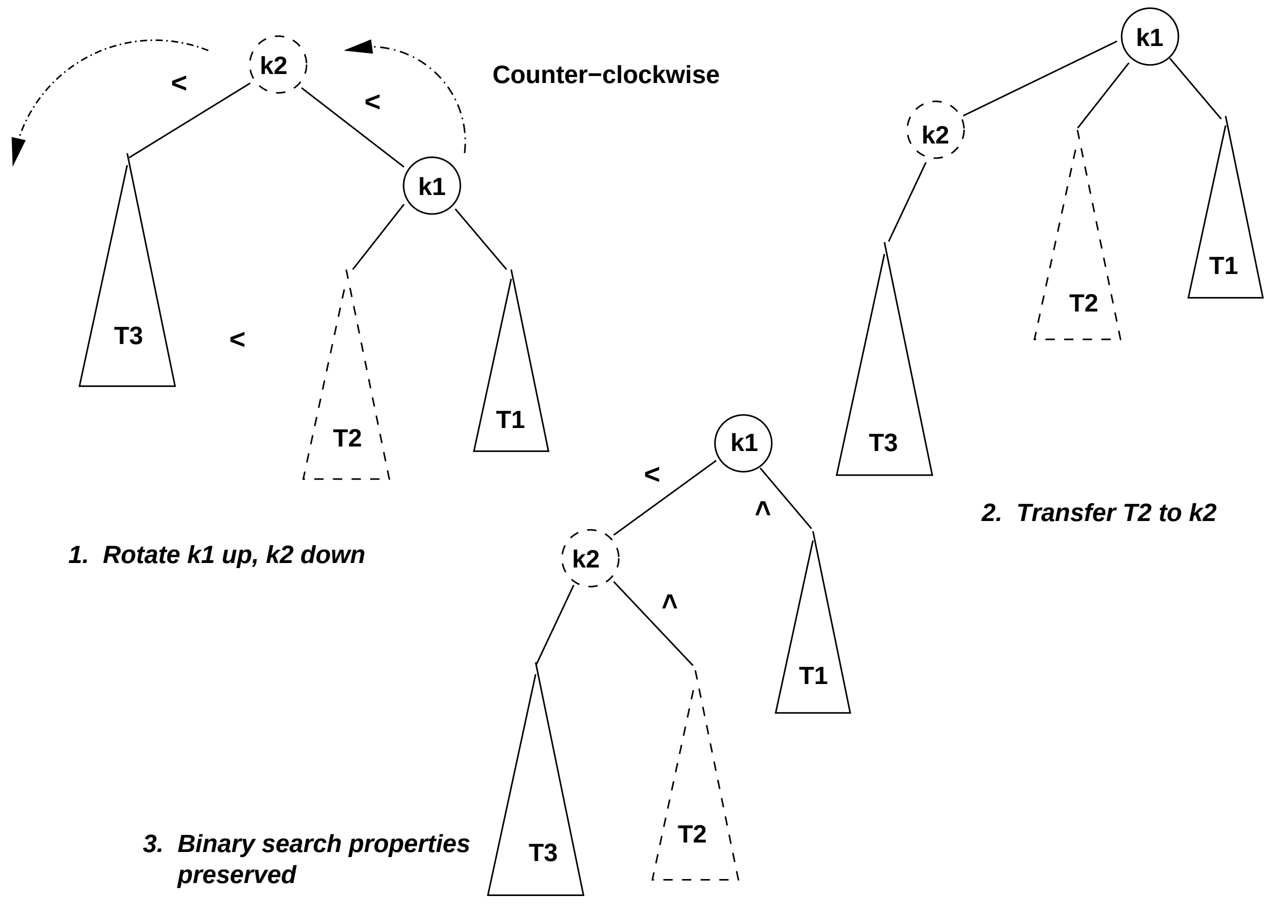


*1. Rotate k1 up, k2 down*

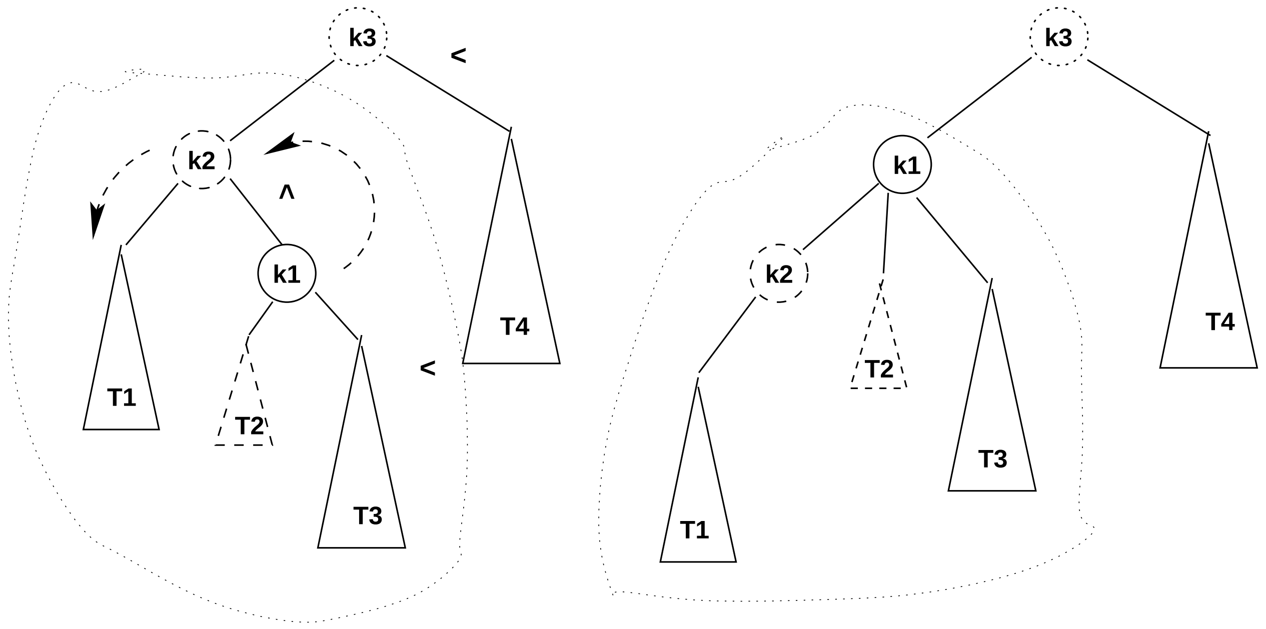
*2. Transfer T2 to k2*

*3. Binary search properties preserved*

## Dual of The Single Rotation Idea



## The Double Rotation Idea – Stage I



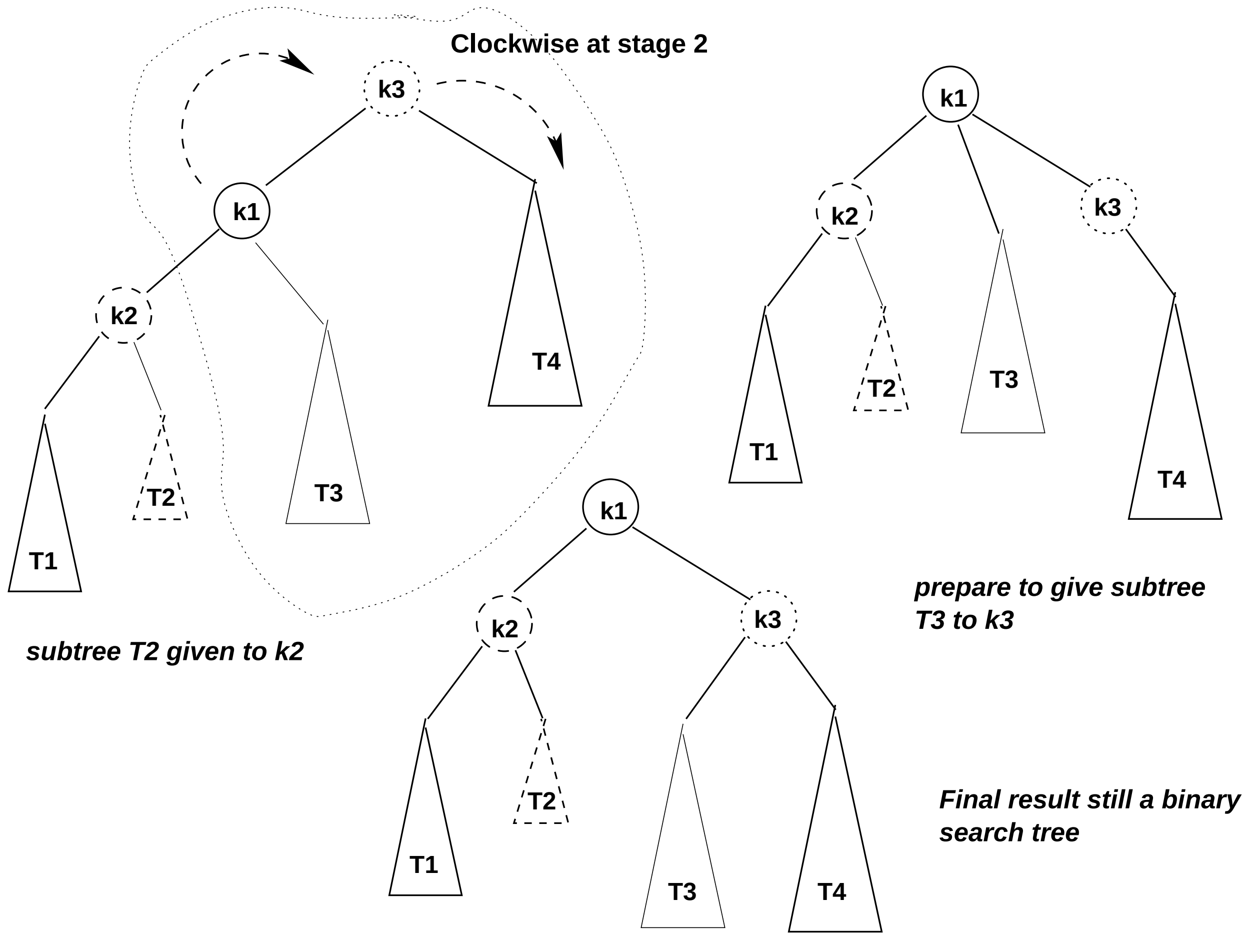
***Double Rotation –– stage 1: rotate grandchild k1 with child k2;***

***Child promoted to parent's role.***

***Goal: eventually, grandchild will go up to grandparent's role***

**Counter−clockwise at first stage**

## The Double Rotation Idea – Stage II



*Double Rotation, stage 2: rotating k1 and k3. Transfering subtree T3 to k3*

## Dual of The Double Rotation Idea

Exercise: You draw the pitures!

Observation: These rotations all preserve the binary search tree property, so inorder projection still works. They do NOT assume that the tree is balanced.

*Checking rotations*: To see if a proposed rotation is legitimate, check for the inorder projection property — this is an invariant. Verify that it is so for all the rotations so far described.

## AVL Trees

*What is an AVL tree?*

**Definition**: The *height* of a tree is the maximum length of paths from root to leaves.

**Definition**: A binary search tree is an AVL tree if the height of the left and right subtrees of any node differs by at most one.

## Simple AVL Tree properties

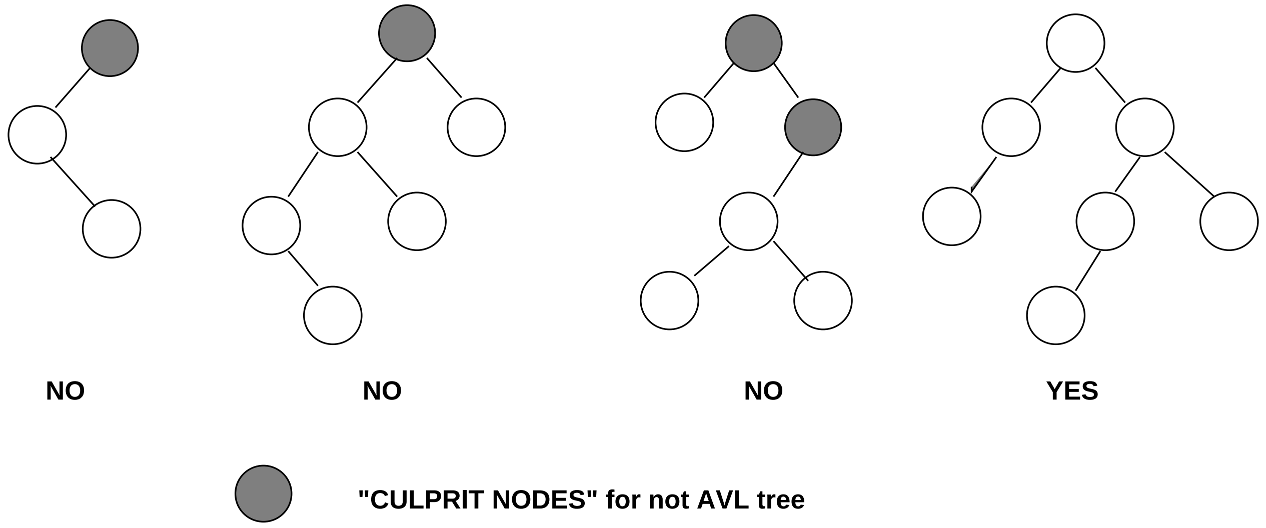
If the height of a given node's left subtree is m, then the height of this node's right subtree must be neither lower than m-1 nor higher than m+1.

Any subtree of an AVL tree is also an AVL tree.

An empty binary search tree and a binary search tree consisting of exactly one node are AVL trees.

**Convention**: Call AVL trees *balanced*, non-AVL trees *unbalanced*.

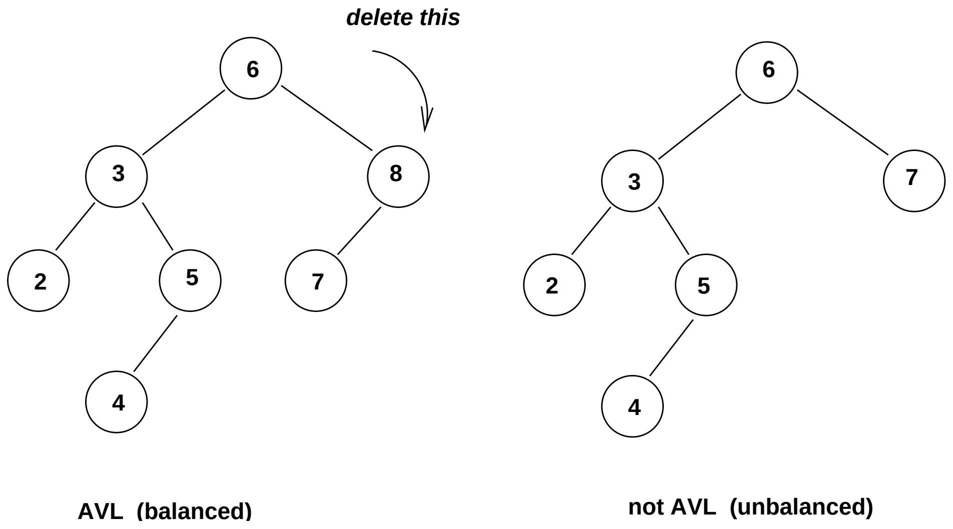
## Examples and Counter-examples



## Insertion and Deletion in AVL Trees

Insertion and Deletion are done as if the AVL tree is a standard binary search tree.

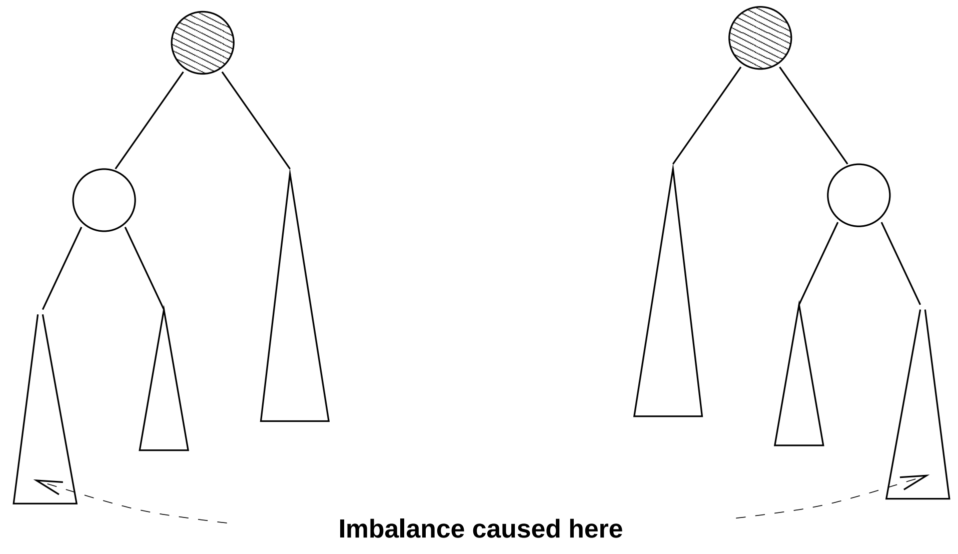
After insertion/deletion, the AVL tree may become *unbalanced*.



To restore it to the AVL condition, we do either one or two *rotations*.

## When to do One Rotation

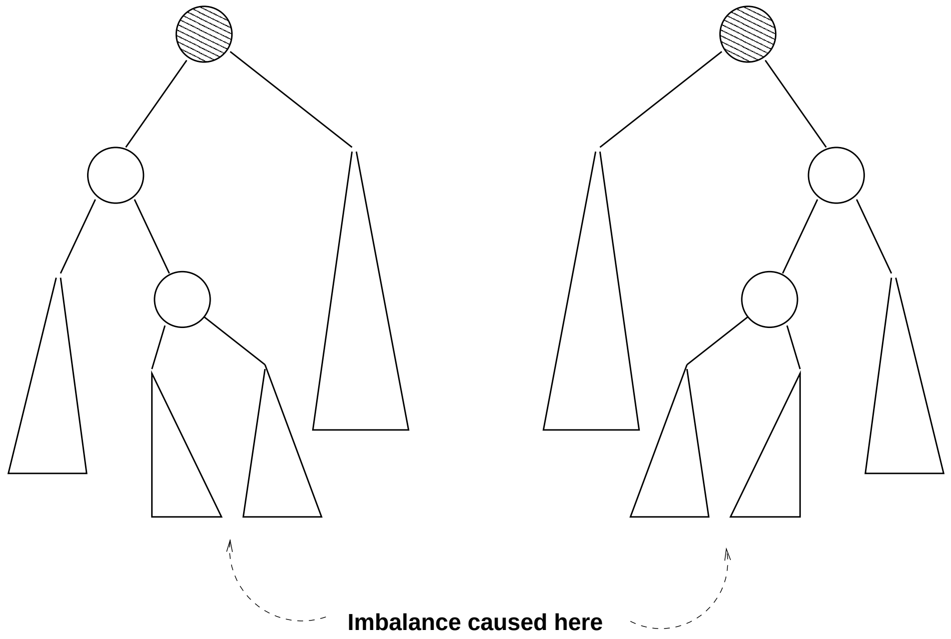
Do this in the case when the insertion or deletion causes an imbalance on the *outside*, i.e., relative to the *lowest* culprit node, the imbalance is in the *left subtree of its left child*, or the *right subtree of its right child.*



The OUTSIDE case

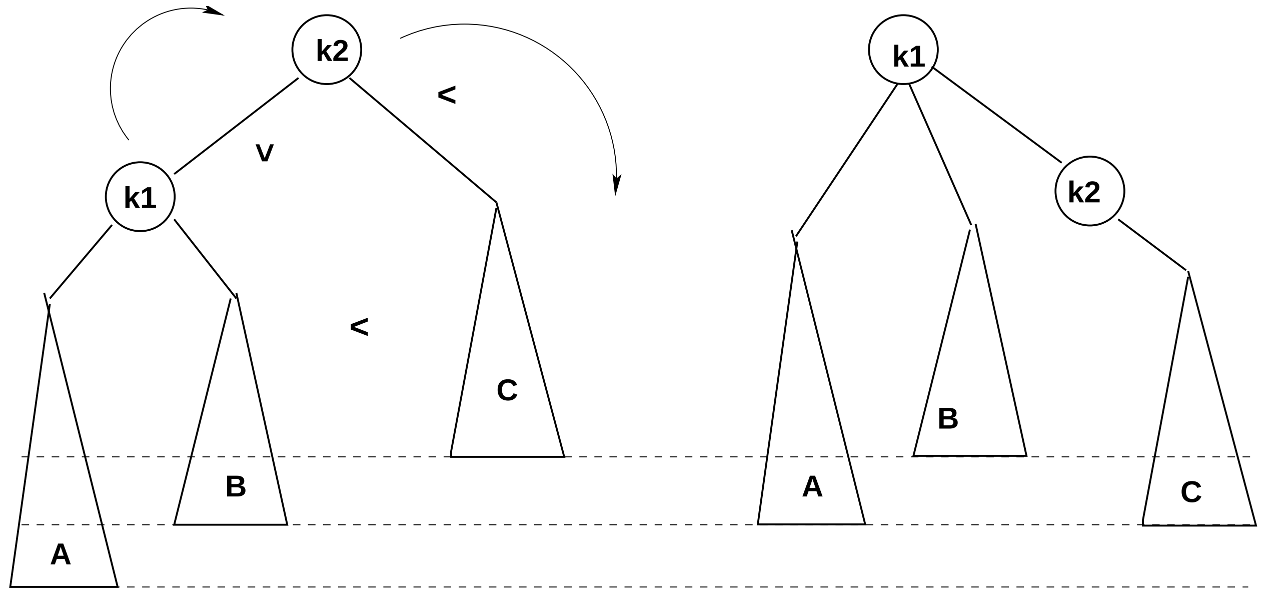
## When to do Two Rotations

Do this in the case when the insertion or deletion causes an imbalance on the *inside*, i.e., relative to the lowest culprit node, the imbalance is in the *right subtree of its left child*, or the *left subtree of its right child*.



The INSIDE case

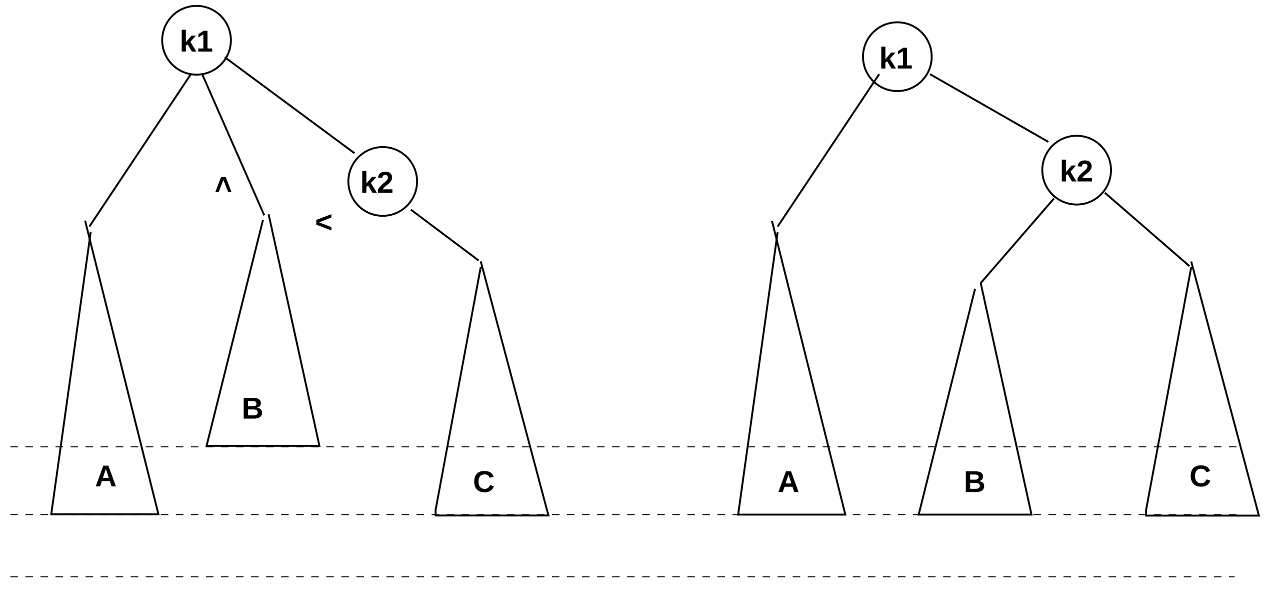
## Single Rotation Details, Stage I



***Single rotation; stage 1 −− "pick up k1, let k2 drop"***

***k1 goes up 1 level, k2 drops 1 level; subtree levels as shown***

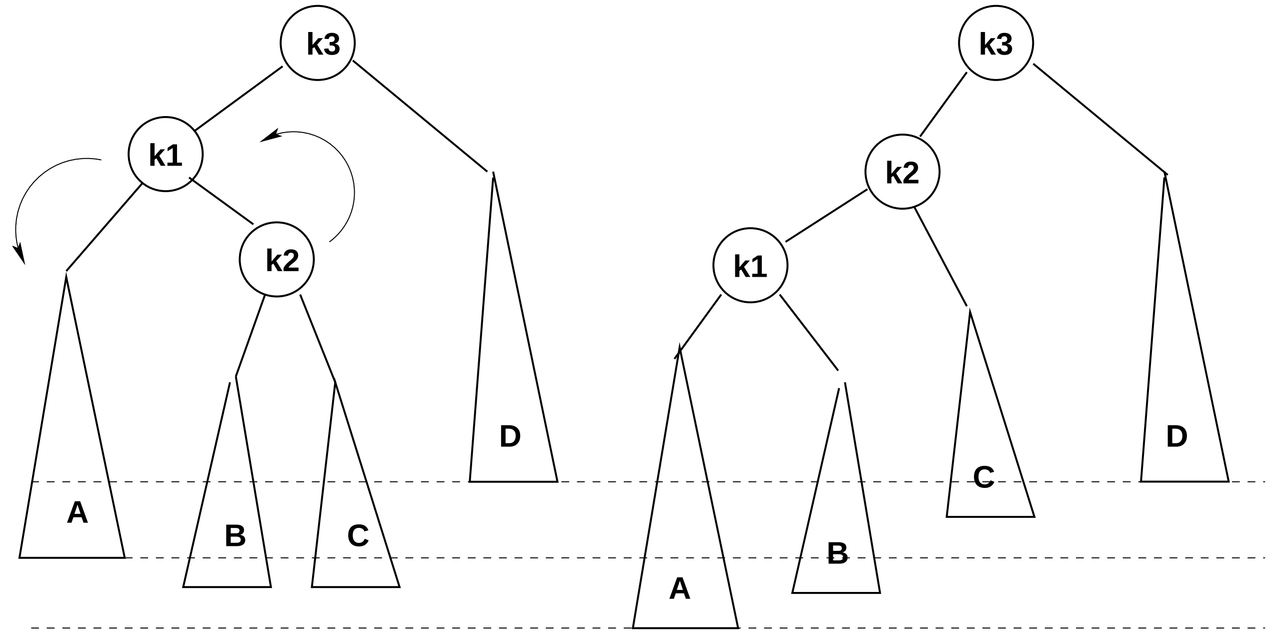
## Single Rotation, Final



***Single rotation, stage 2;***
***transfer subtree B to k2; when done all subtrees***
***are at the same depth***

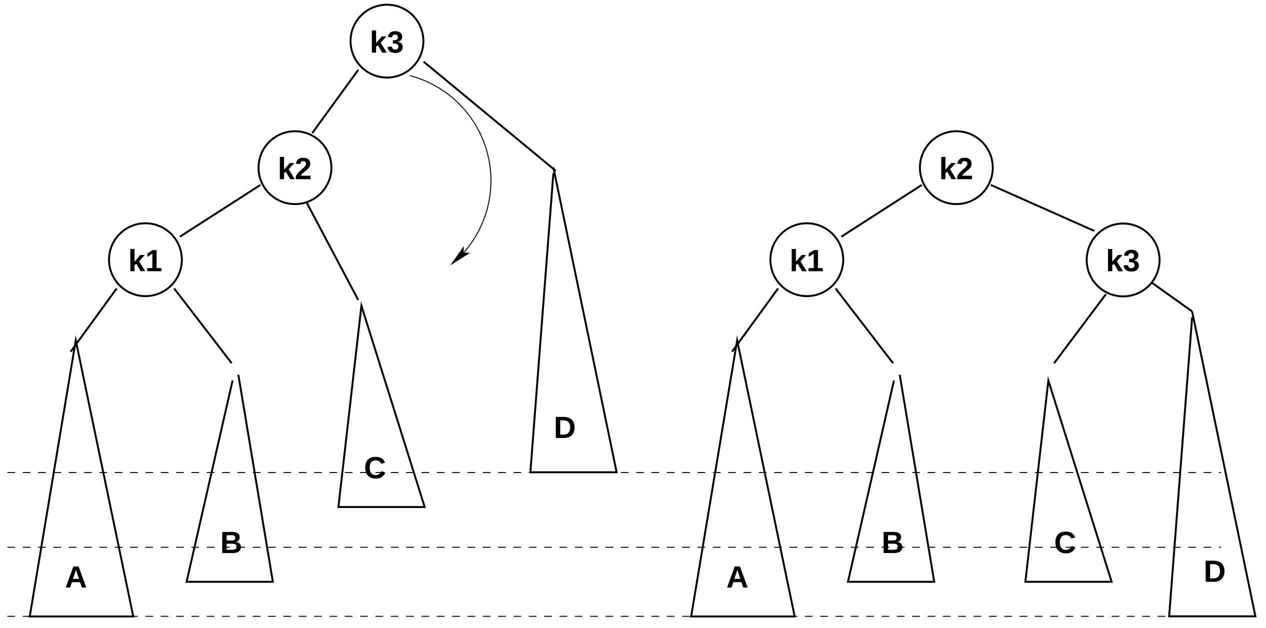
Figure 1: Check the re-balance!

## Double Rotation Details, Stage I



***[One of B or C is at bottom depth]***
***Double rotation; stage 1: lift k2, drop k1***
***Transfer subtree B to k1***

## Double Rotation, Final



***Double rotation; stage 2 –– drop k3 below k2;***
***Transfer subtree C to k3***

Figure 2: Check the re-balance!