Balanced Binary Search Tree

- Worst case height of binary search tree: N-1
  - Insertion, deletion can be O(N) in the worst case
- We want a tree with small height
- Height of a binary tree with N node is at least O(log N)
  - Complete binary tree has height log N.
- Goal: keep the height of a binary search tree O(log N)
- Balanced binary search trees
  - Examples: AVL tree, red-black tree

AVL tree

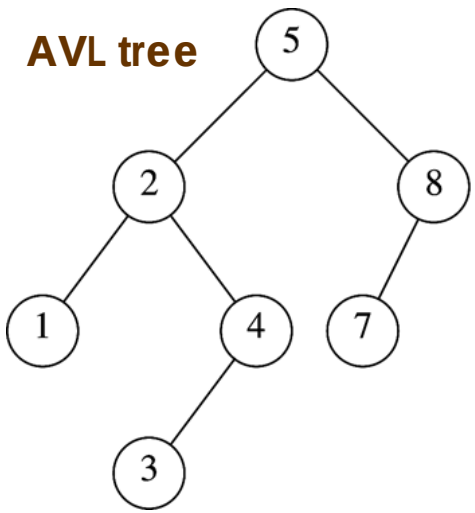
for each node, the height of the left and right subtrees can differ by at most d = 1.

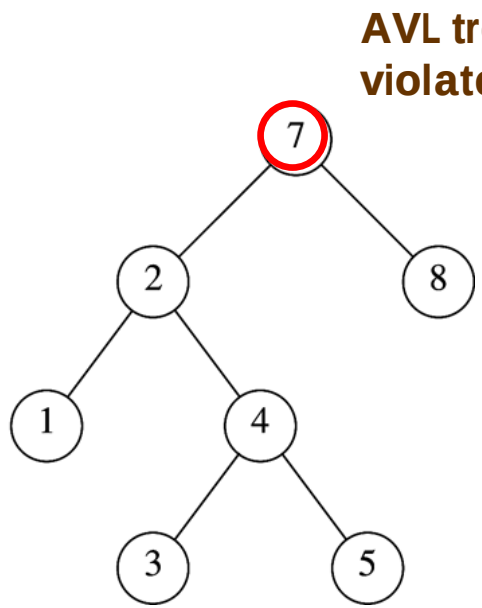
Maintaining a stricter condition (e.g above condition with d = 0) is difficult.

Note that our goal is to perform all the operations search, insert and delete in O(log N) time, including the operations involved in adjusting the tree to maintain the above balance condition.

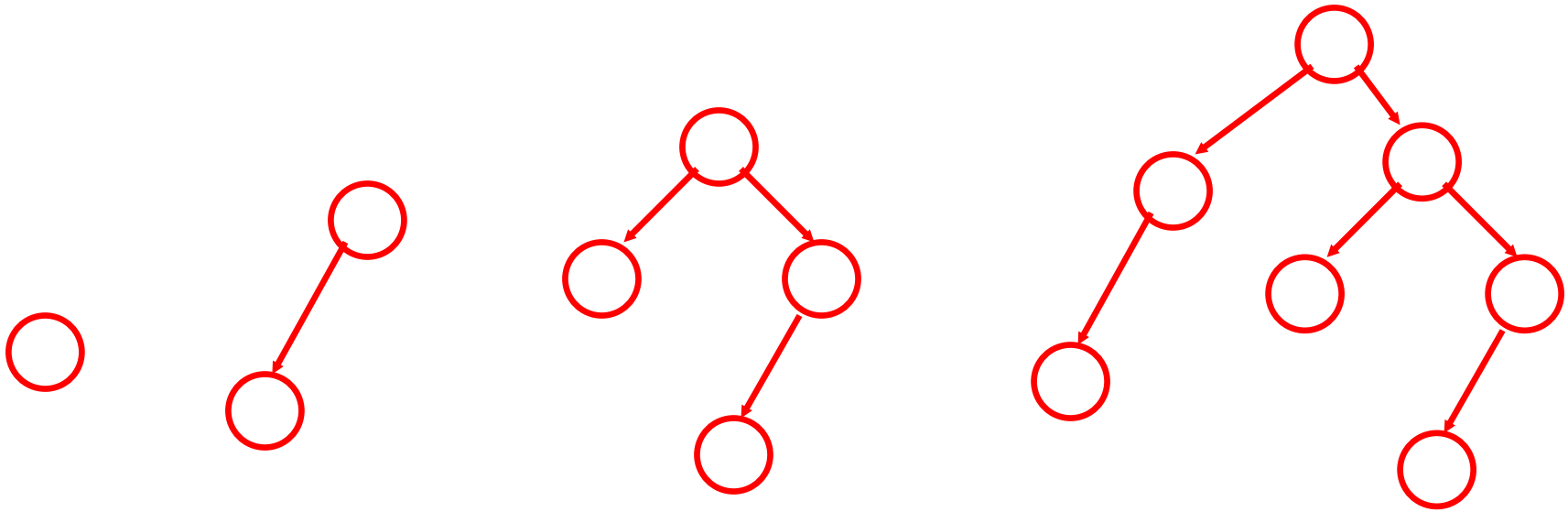
AVL Tree

- An AVL tree is a binary search tree in which
  - for *every* node in the tree, the height of the left and right subtrees differ by at most 1.
- Height of subtree: Max # of edges to a leaf
- Height of an empty subtree: -1
  - Height of one node: 0

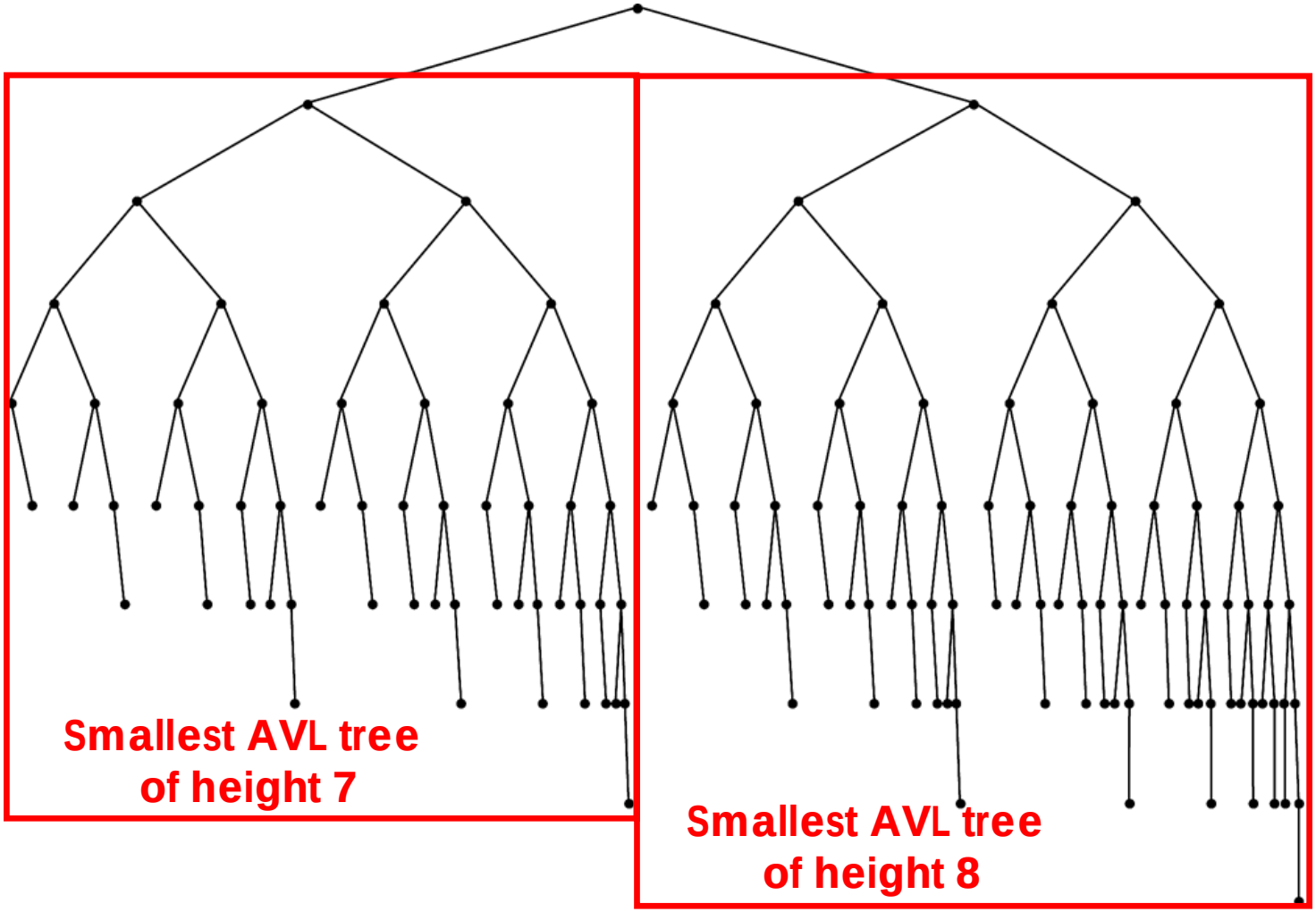




AVL Tree with Minimum Number of Nodes



$N_0 = 1$ $N_1 = 2$ $N_2 = 4$ $N_3 = N_1 + N_2 + 1 = 7$


Smallest AVL tree
of height 7
Smallest AVL tree
of height 8

Height of AVL Tree with N nodes

- Denote N_h the minimum number of nodes in an AVL tree of height h

- $N_0 = 1$, $N_1 = 2$ (base)
  $N_h = N_{h-1} + N_{h-2} + 1$ (recursive relation)

- $N > N_h = N_{h-1} + N_{h-2} + 1$
  $> 2\ N_{h-2} > 4\ N_{h-4} > \ldots > 2^i\ N_{h-2i}$

Height of AVL Tree with N nodes

- If h is even, let i=h/2–1. The equation becomes

$$N > 2^{h/2-1}N_2 \Rightarrow N > 2^{h/2-1}x4 \Rightarrow h=O(\log N)$$

- If h is odd, let i=(h-1)/2. The equation becomes

$$N > 2^{(h-1)/2}N_1 \Rightarrow N > 2^{(h-1)/2}x2 \Rightarrow h=O(\log N)$$

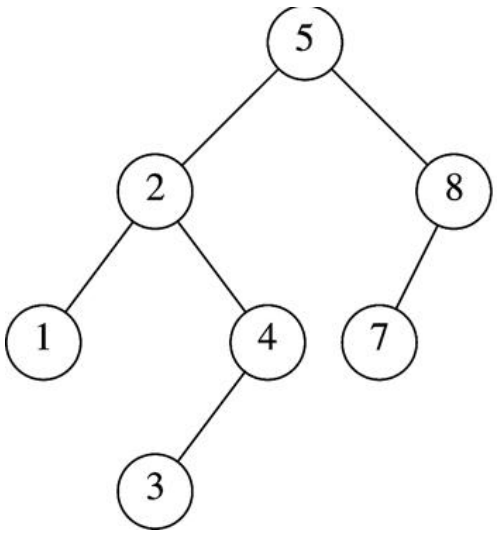
- Recall the cost of operations search, insert and delete is O(h).

- cost of searching = O(log N). (Just use the search algorithm for the binary search tree.)

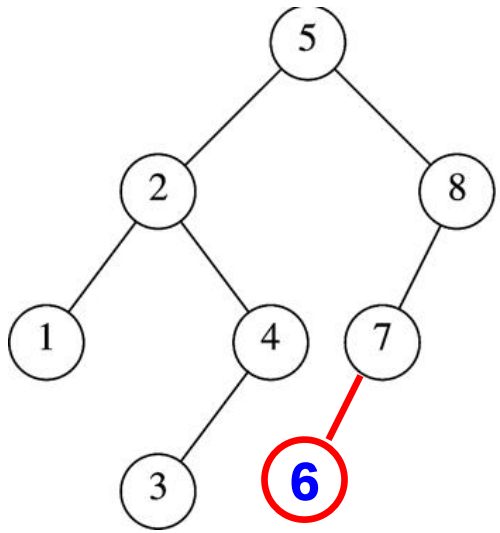
- But insert and delete are more complicated.

Insertion in AVL Tree

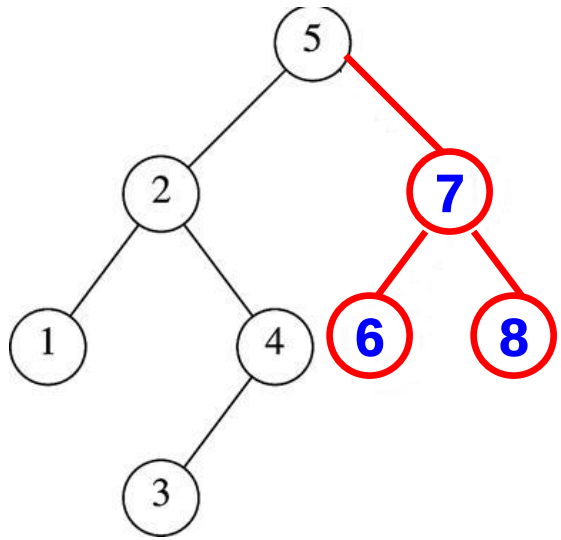
- Basically follows insertion strategy of binary search tree
  - But may cause violation of AVL tree property
- Restore the destroyed balance condition if needed



Original AVL tree



Insert 6
Property violated



Restore AVL property

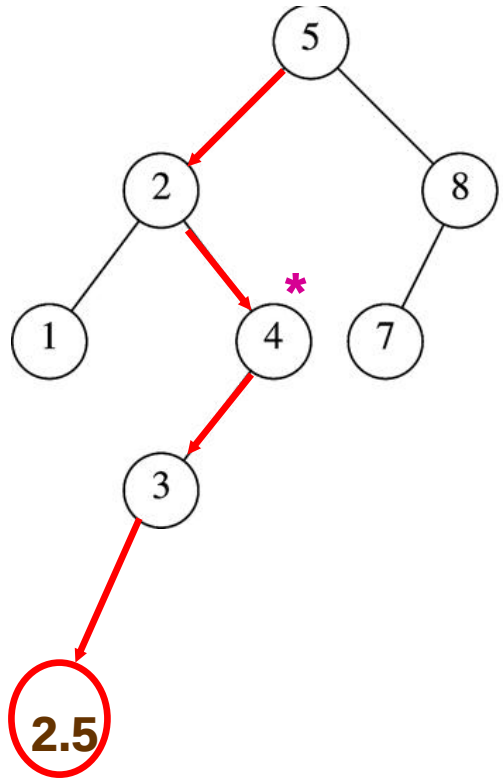
Some Observations

- After an insertion (using the insertion algorithm for a binary search tree), only nodes that are on the path from the insertion point to the root might have their balance altered.
  - Because only those nodes have their subtrees altered
- So it's enough to adjust the balance on these nodes.
- We will see that AVL tree property can be restored by performing a balancing operation at a single node.

Node at which balancing must be done

- The node of smallest depth on the path from to the newly inserted node.
- We will call this node pivot.

Example:



Suppose we inserted key is 2.5

Pivot is the node containing 4.

This is the lowest depth node in the path where the AVL tree property is violated.

Different Cases for Rebalance

- Let α be the pivot node
  - Case 1: inserted node is in the left subtree of the left child of α (LL-rotation)
  - Case 2: inserted node is in the right subtree of the left child of α (RL-rotation)
  - Case 3: inserted node is in the left subtree of the right child of α (LR-rotation)
  - Case 4: inserted node is in the right subtree of the right child of α (RR-rotation)
- Cases 3 & 4 are mirror images of cases 1 & 2 so we will focus on 1 & 2.

Rotations

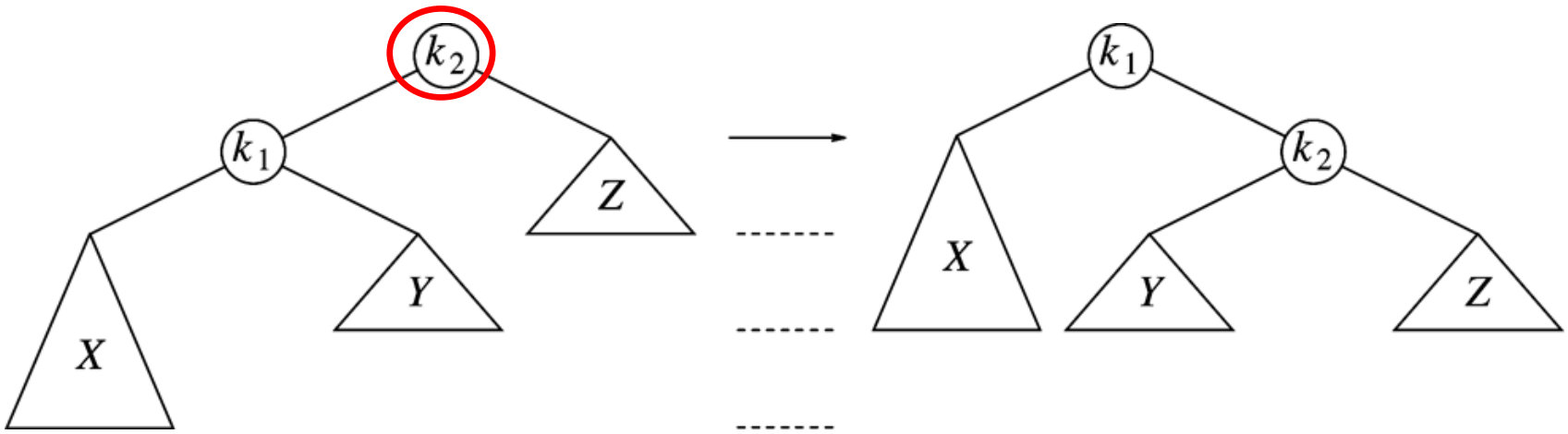
- Rebalance of AVL tree are done with simple modification to tree, known as rotation
- Insertion occurs on the “outside” (i.e., left-left or right-right) is fixed by single rotation of the tree
- Insertion occurs on the “inside” (i.e., left-right or right-left) is fixed by double rotation of the tree

Insertion Algorithm (outline)

- First, insert the new key as a new leaf just as in ordinary binary search tree
- Then trace the path from the new leaf towards the root. For each node x encountered, check if heights of left(x) and right(x) differ by at most 1.
  - If yes, proceed to parent(x)
  - If not, restructure by doing either a single rotation or a double rotation
- Note: once we perform a rotation at a node x, we won't need to perform any rotation at any ancestor of x.

Single Rotation to Fix Case 1(LL-rotation)

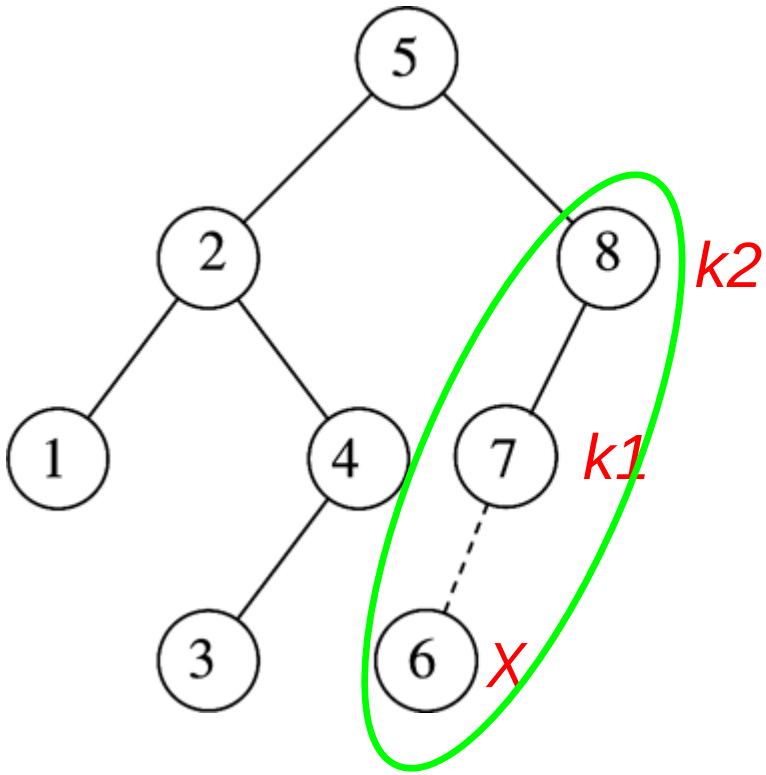
k2 is the pivot node

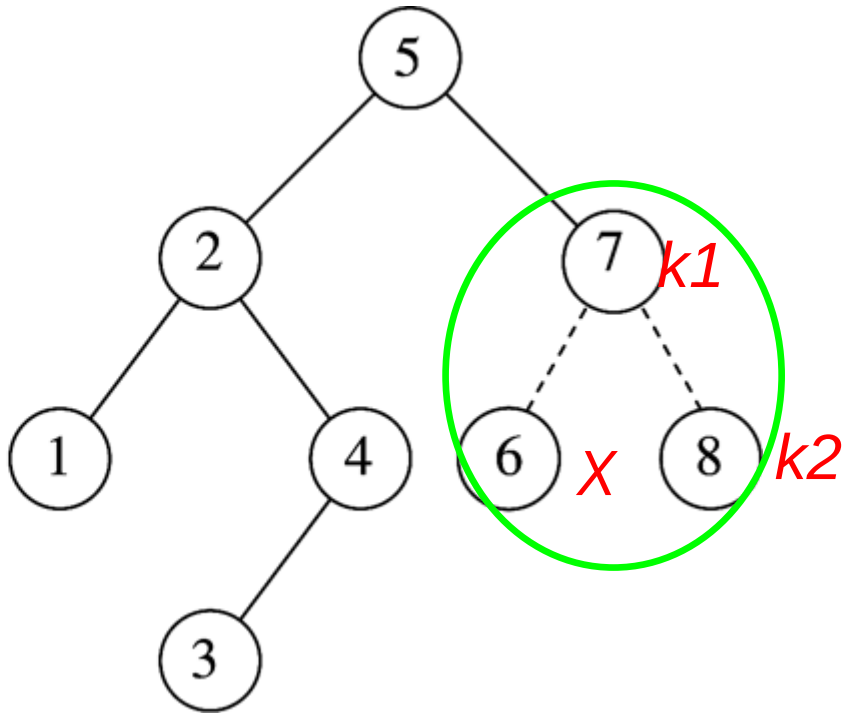


Questions:

- Can Y have the same height as the new X?
- Can Y have the same height as Z?

Single Rotation Case 1 Example





Informal proof/argument for the LL-case

We need to show the following:

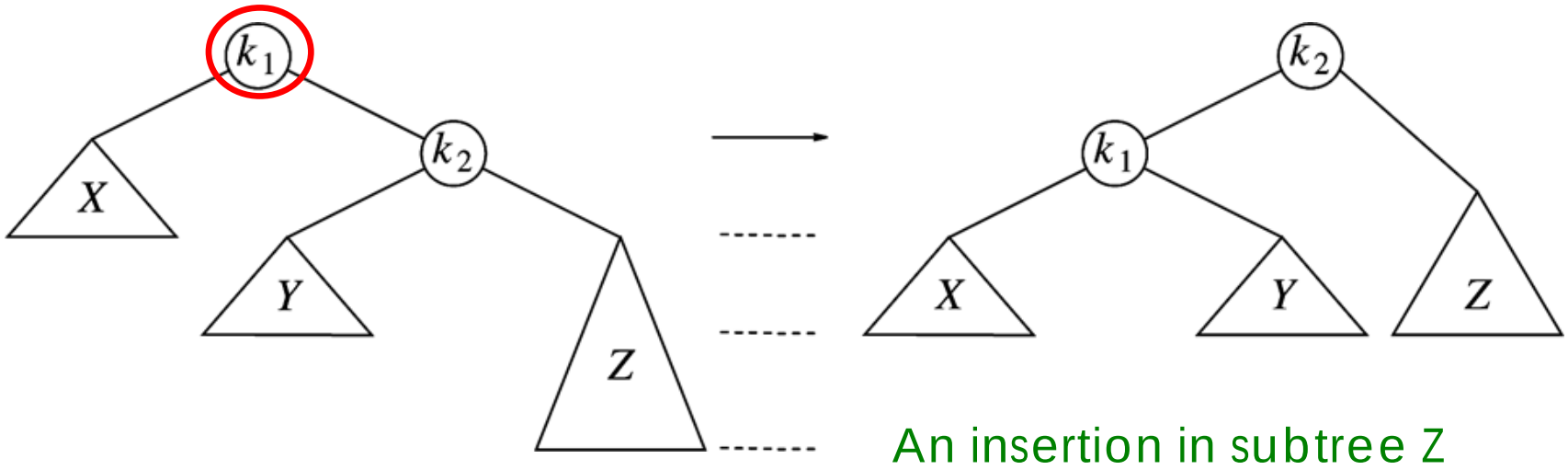
- After the rotation, the balance condition at the pivot node is restored and at all the nodes in the subtree rooted at the pivot.

- After the rotation, the balance condition is restored at all the ancestors of the pivot.

- We will prove both these informally.

Single Rotation to Fix Case 4 (RR-rotation)

k1 violates AVL tree balance condition

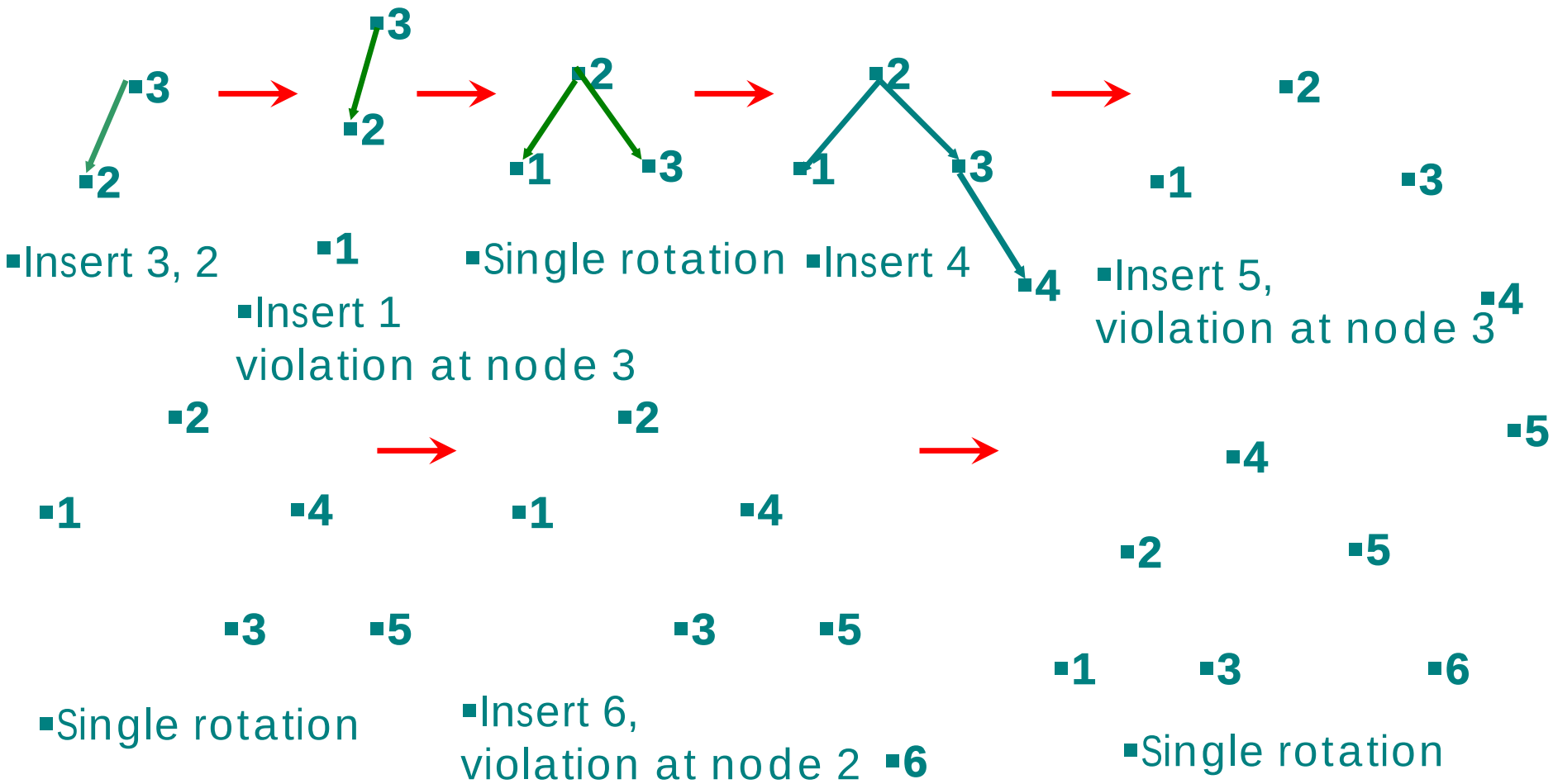


An insertion in subtree Z

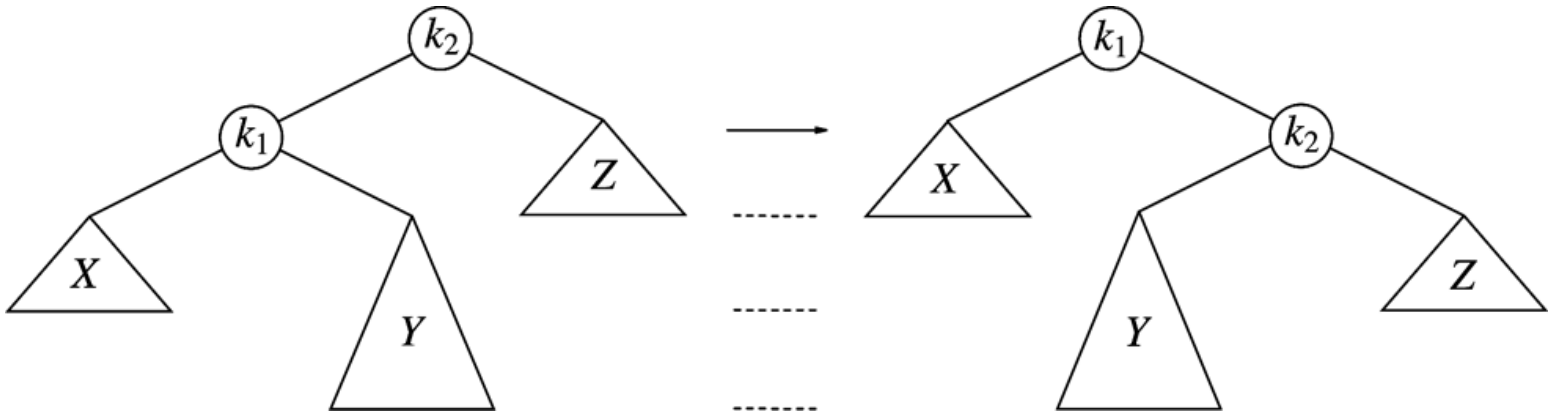
- Case 4 is a symmetric case to case 1
- Insertion takes O(Height of AVL Tree) time, Single rotation takes O(1) time

Single Rotation Example

- Sequentially insert 3, 2, 1, 4, 5, 6 to an AVL Tree



Single Rotation Fails to fix Case 2&3



Case 2: violation in k2 because of insertion in subtree Y

Single rotation result

- Single rotation fails to fix case 2&3
- Take case 2 as an example (case 3 is a symmetry to it)
  - The problem is subtree Y is too deep
  - Single rotation doesn't make it any less deep