Data Structures & Algorithms (CS-212)

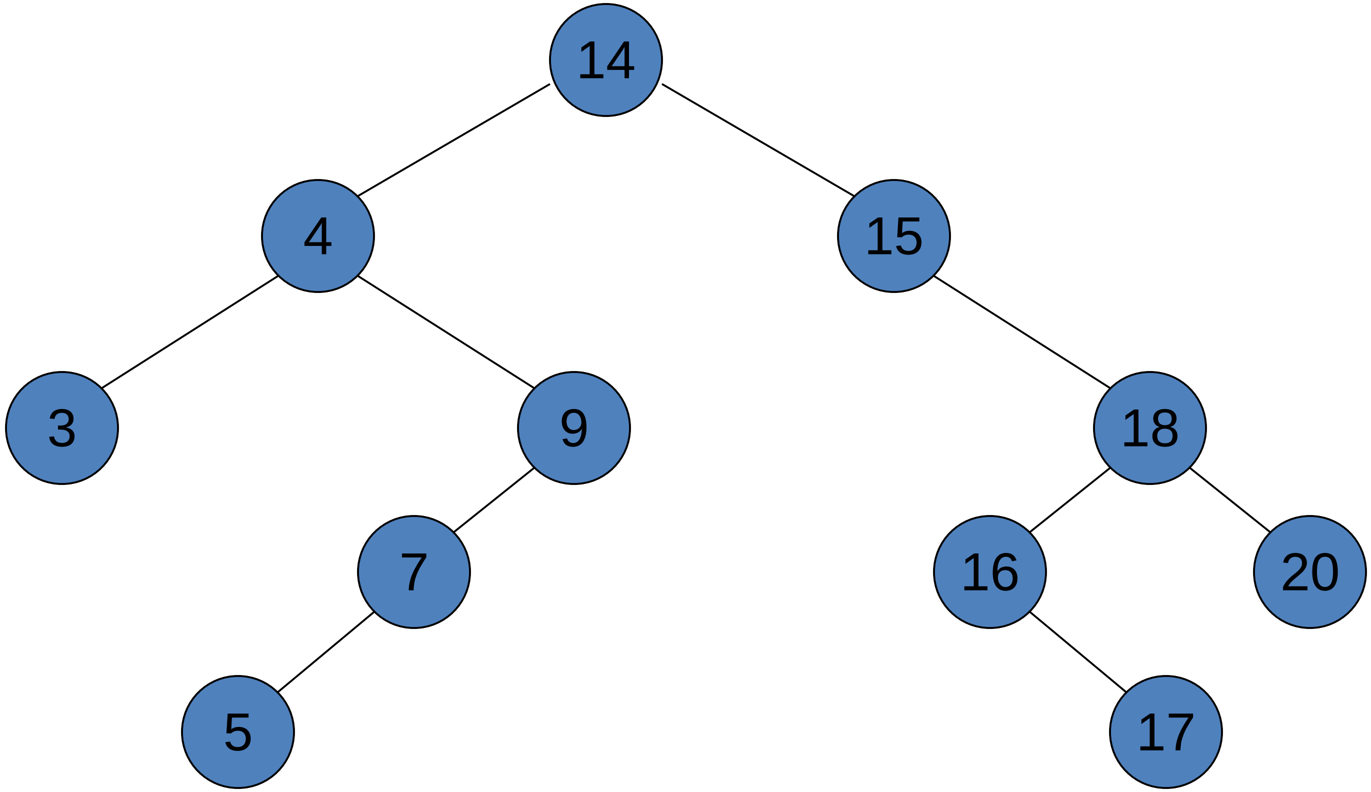
Week 11: Tree Balancing

Outline

- Introduction
- Insertion
- Deletion
- Implementation

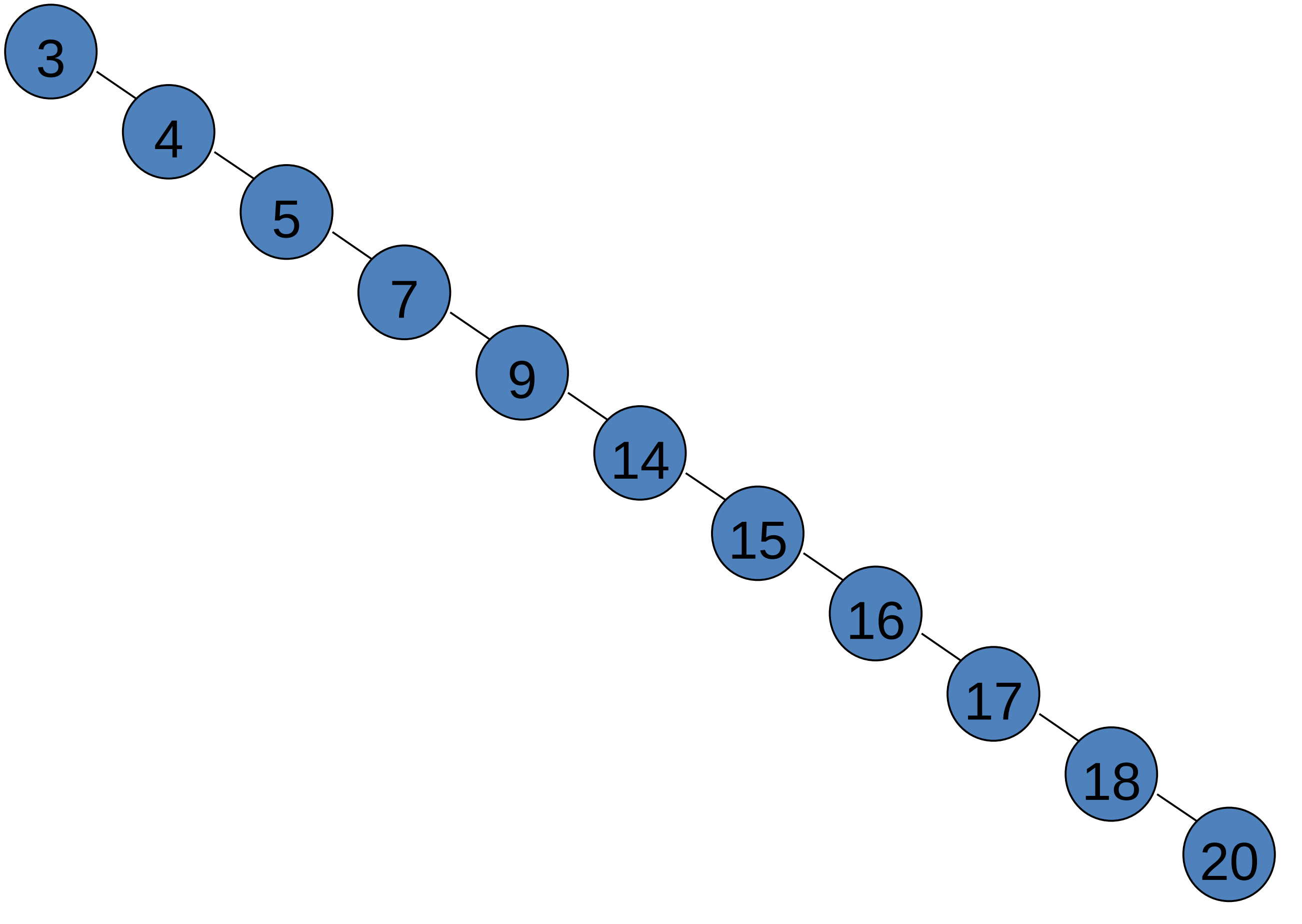
Degenerate Binary Search Tree

BST for 14, 15, 4, 9, 7, 18, 3, 5, 16, 20, 17



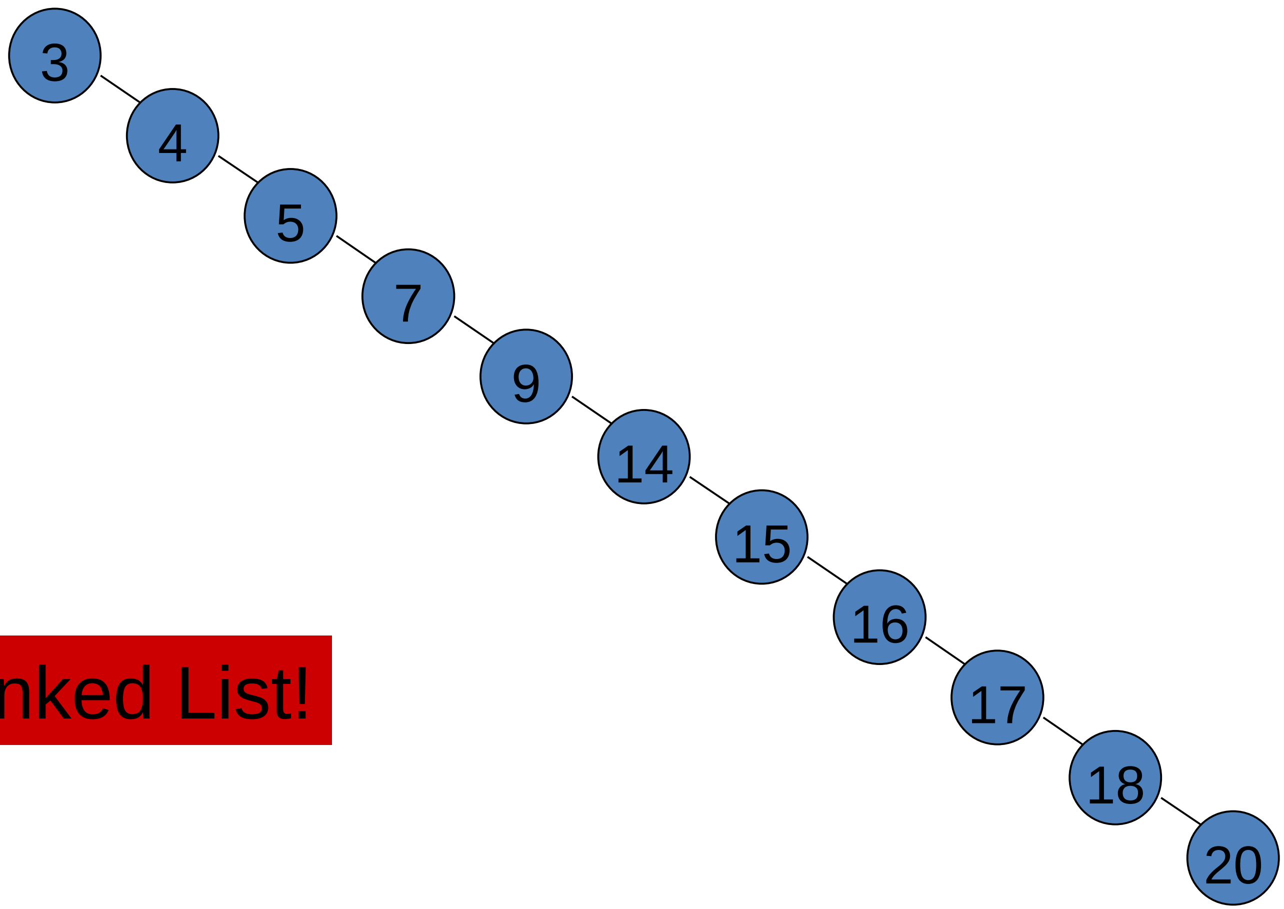
Degenerate Binary Search Tree

BST for 3 4 5 7 9 14 15 16 17 18 20



Degenerate Binary Search Tree

BST for 3 4 5 7 9 14 15 16 17 18 20

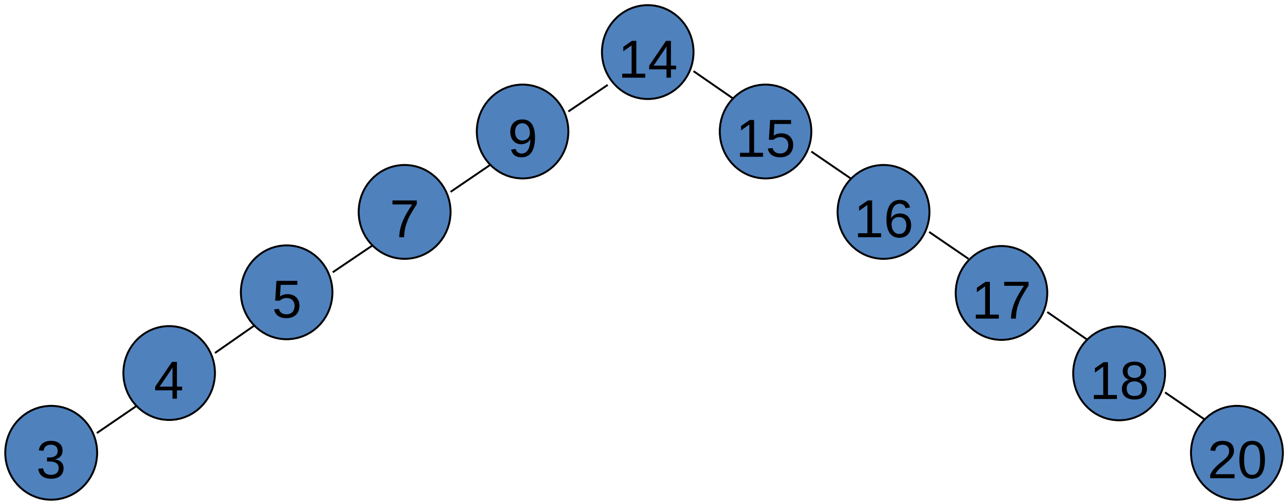


Linked List!

Balanced BST

- We should keep the tree *balanced*.
- One idea would be to have the left and right subtrees have the same height

Balanced BST



Does not force the tree to be *shallow*.

Balanced BST

- We could insist that every node must have left and right subtrees of same height.
- But this requires that the tree be a complete binary tree
- To do this, there must have ($2^{d+1} - 1$) data items, where *d* is the depth of the tree.
- This is too rigid a condition.

Approaches to balancing trees

- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - Self-adjusting

Balancing Binary Search Trees

- Many algorithms exist for keeping binary search trees balanced
  - Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
  - Splay trees and other self-adjusting trees
  - B-trees and other multiway search trees

The **AVL Tree** Data Structure

An **AVL(Adelson-Velskii and Landis) tree** is a *self-balancing* binary search tree.

Structural properties

1. Binary tree property (same as BST)
2. Order property (same as for BST)

3. Balance condition:
   balance of every node is between -1 and 1

   where **balance**(*node*) = height(*node*.left) – height(*node*.right)

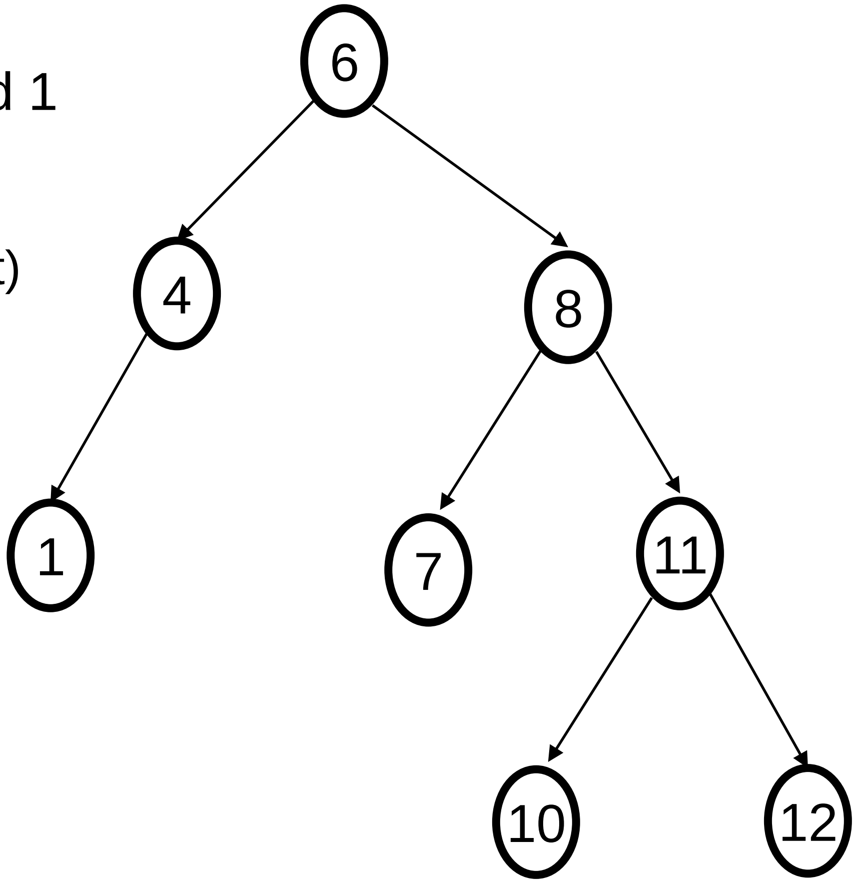
Balanced Binary Tree

- The *height* of a binary tree is the maximum level of its leaves (also called the depth).
- The balance of a node in a binary tree is defined as the height of its left subtree minus height of its right subtree.
- Here, for example, is a balanced tree. Each node has an indicated balance of 1, 0, or –1.

Example #1: Is this an AVL Tree?

Balance Condition:
balance of every node is between -1 and 1

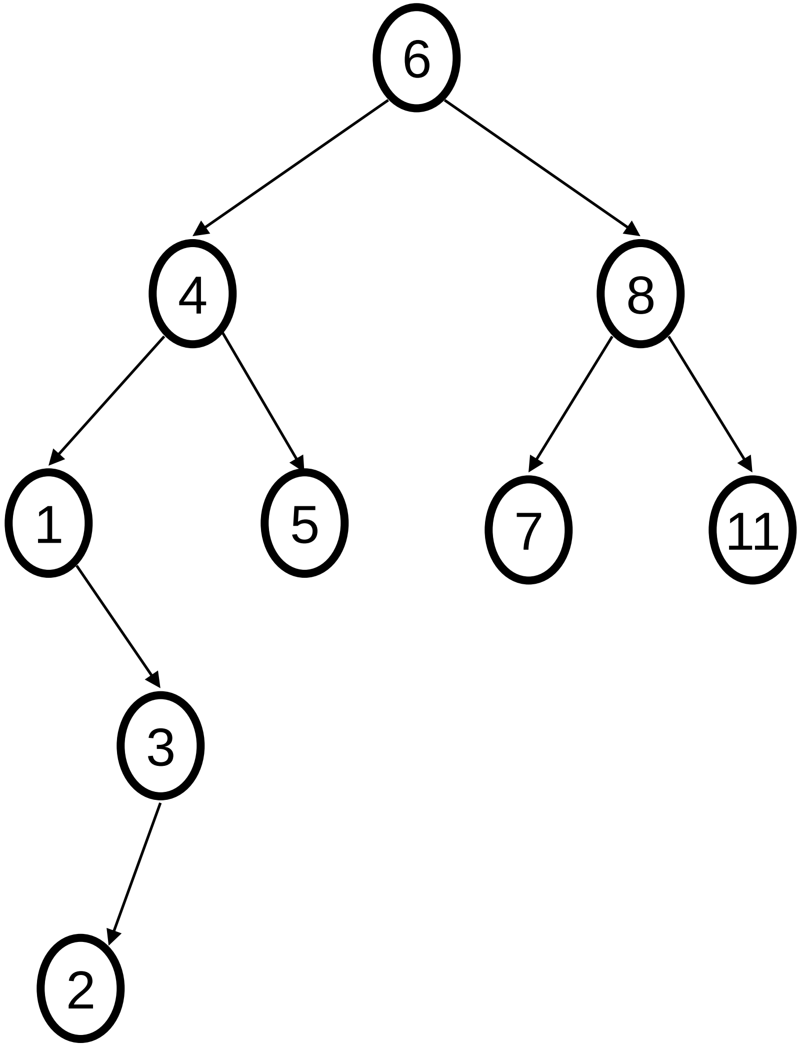
where balance(*node*) =
height(*node*.left) – height(*node*.right)



Example #2: Is this an AVL Tree?

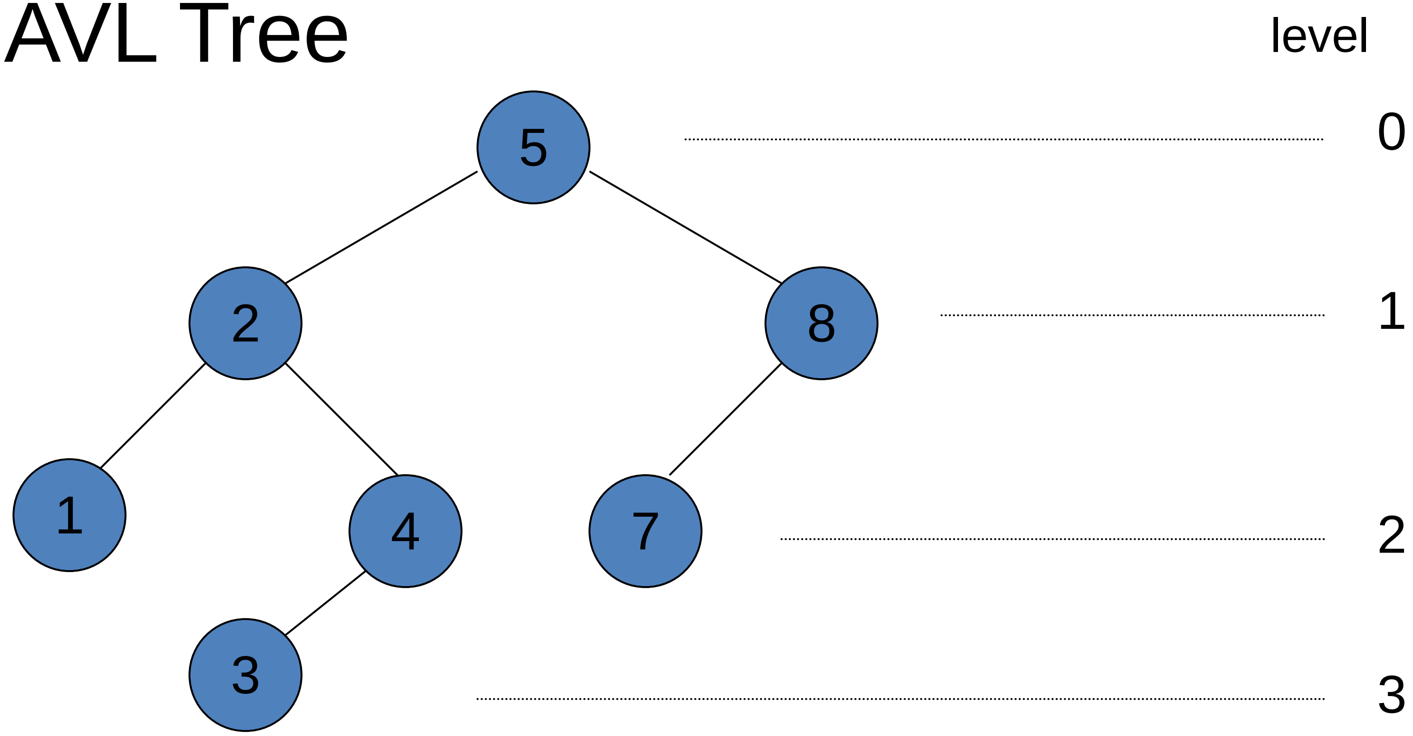
Balance Condition:
balance of every node is between -1 and 1

where balance(*node*) =
height(*node*.left) – height(*node*.right)



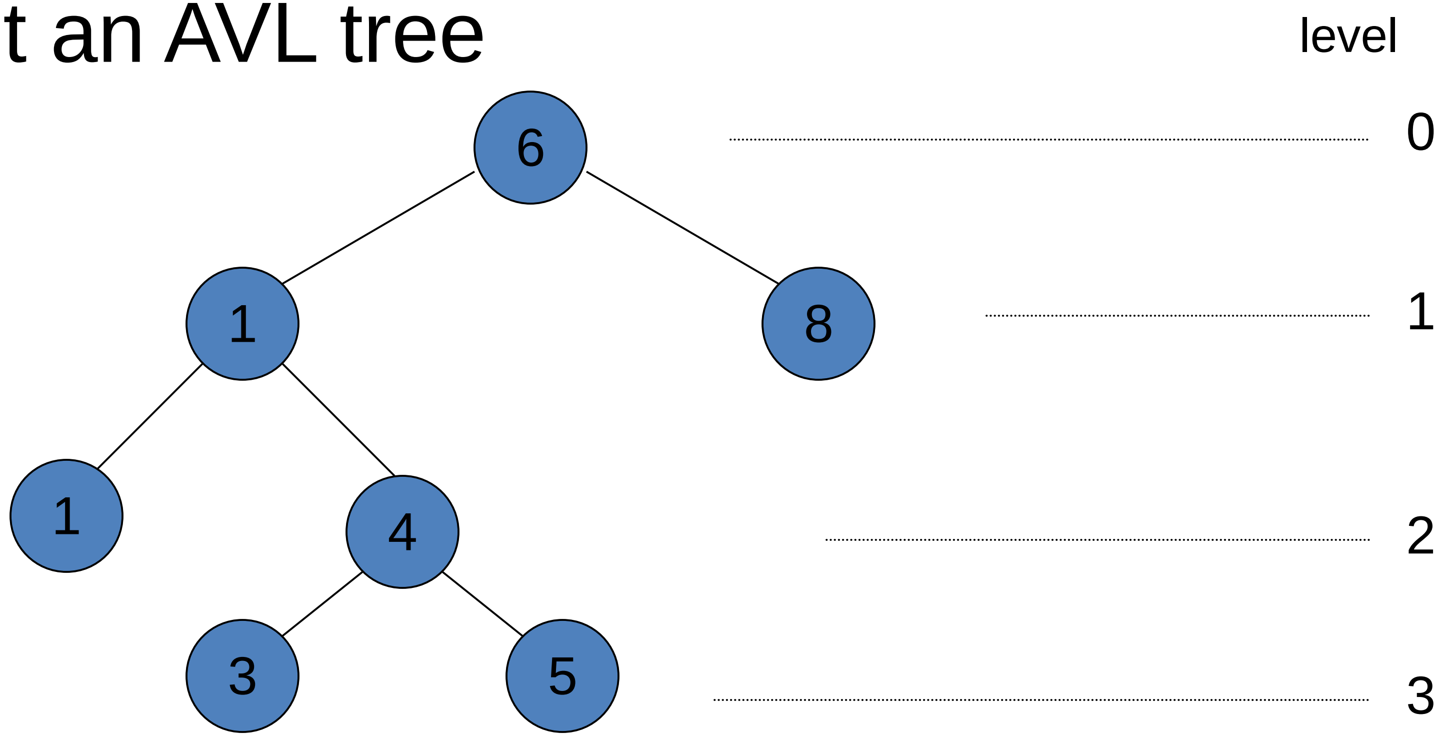
AVL Tree

- An AVL Tree

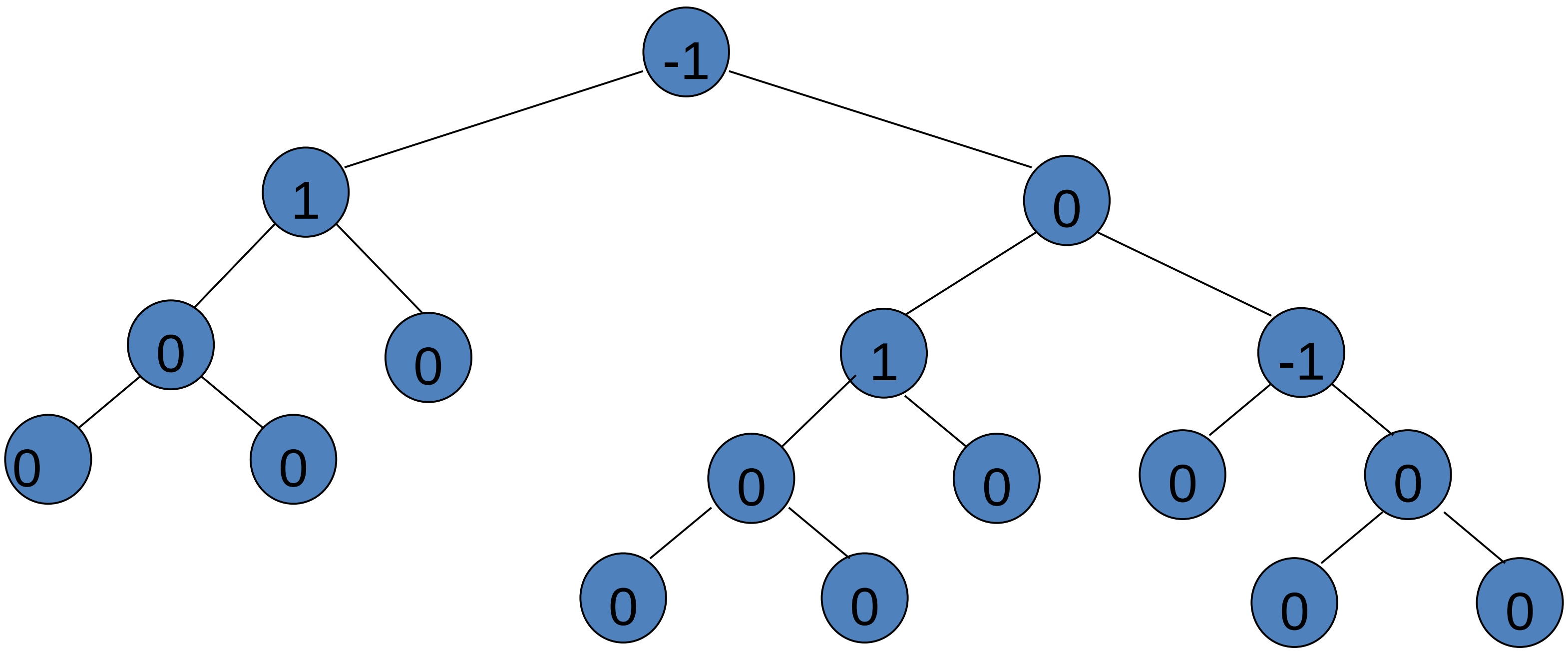


AVL Tree

- Not an AVL tree



Balanced Binary Tree

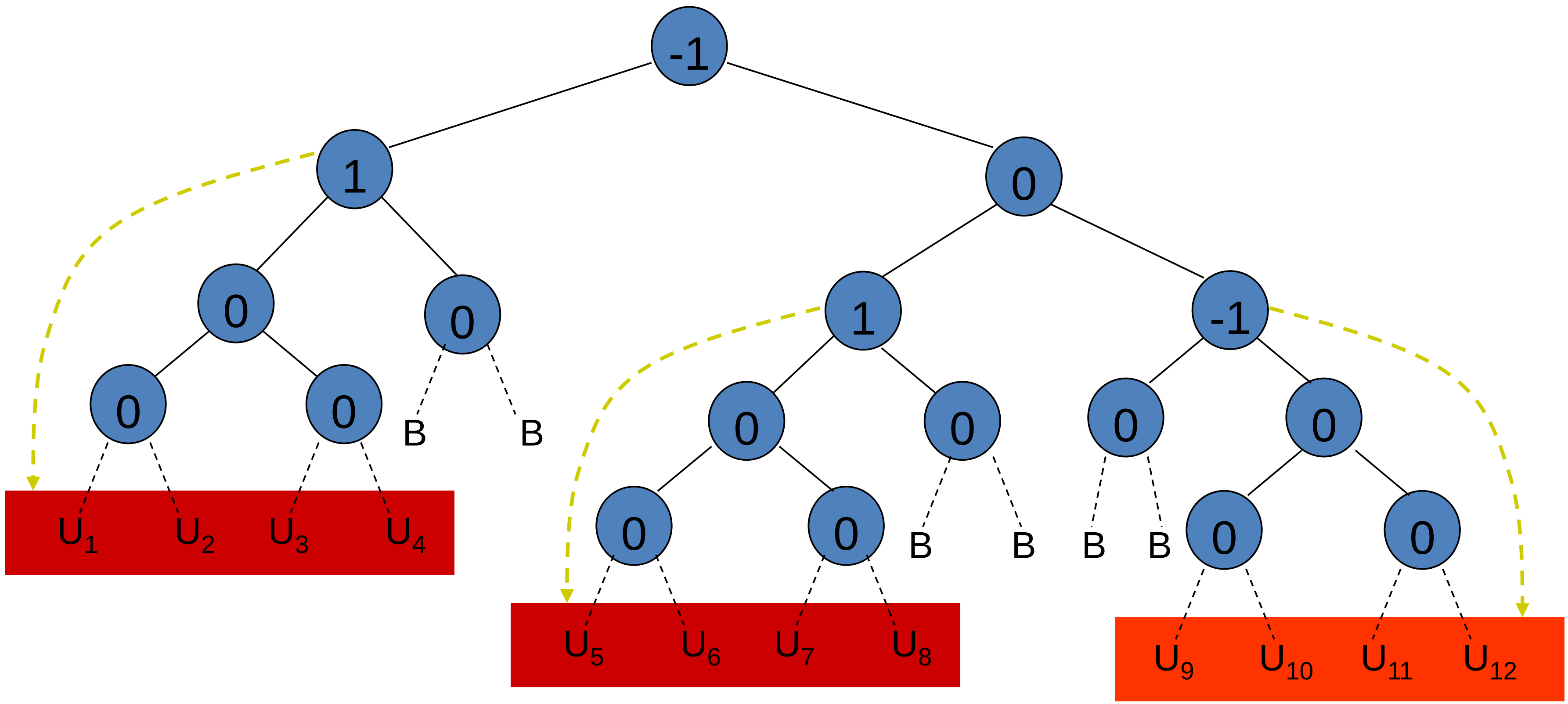


Balanced Binary Tree

- Tree becomes unbalanced only if the newly inserted node
  - is a left descendant of a node that previously had a balance of 1 (U_1 to U_8),
  - or is a descendant of a node that previously had a balance of –1 (U_9 to U_{12})

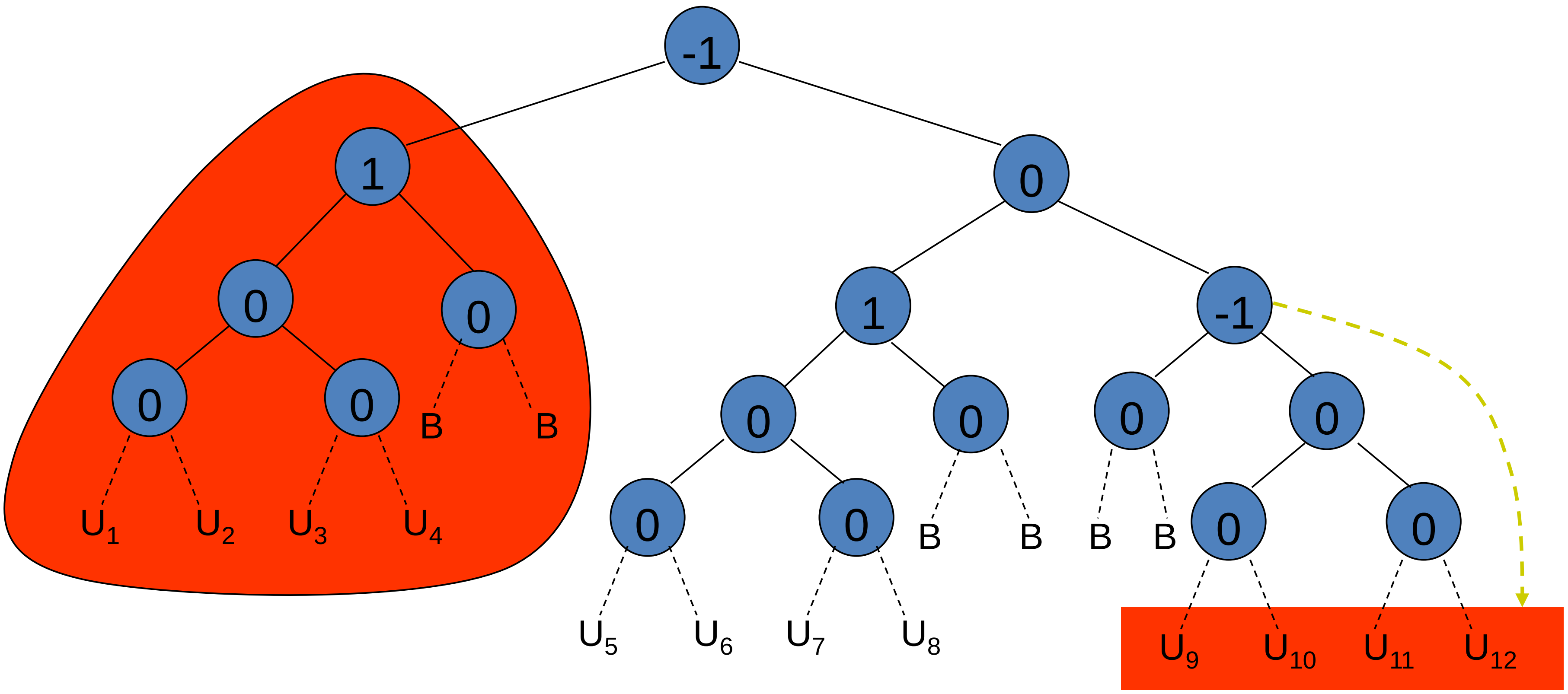
Balanced Binary Tree

Insertions and effect on balance

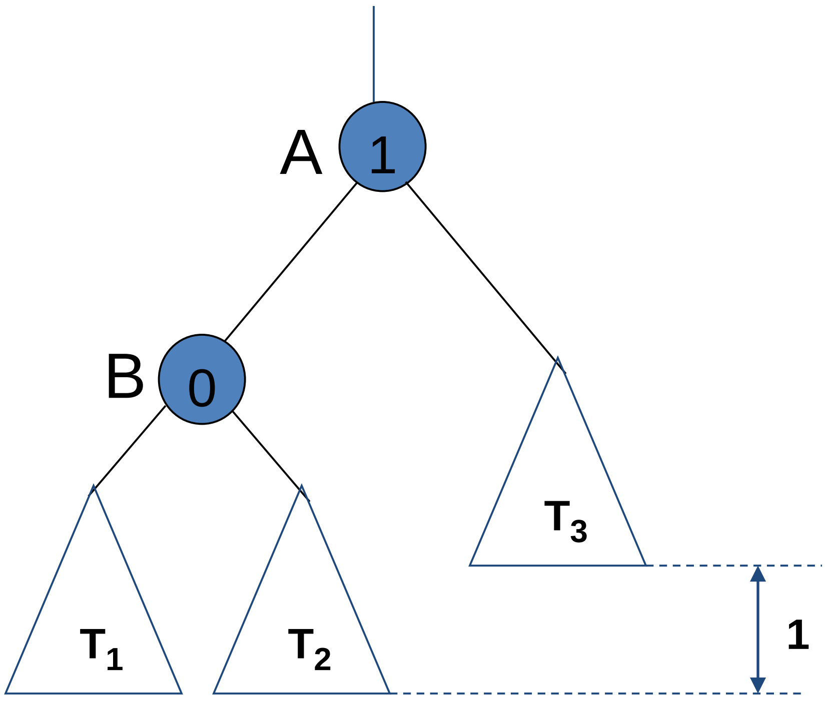


Balanced Binary Tree

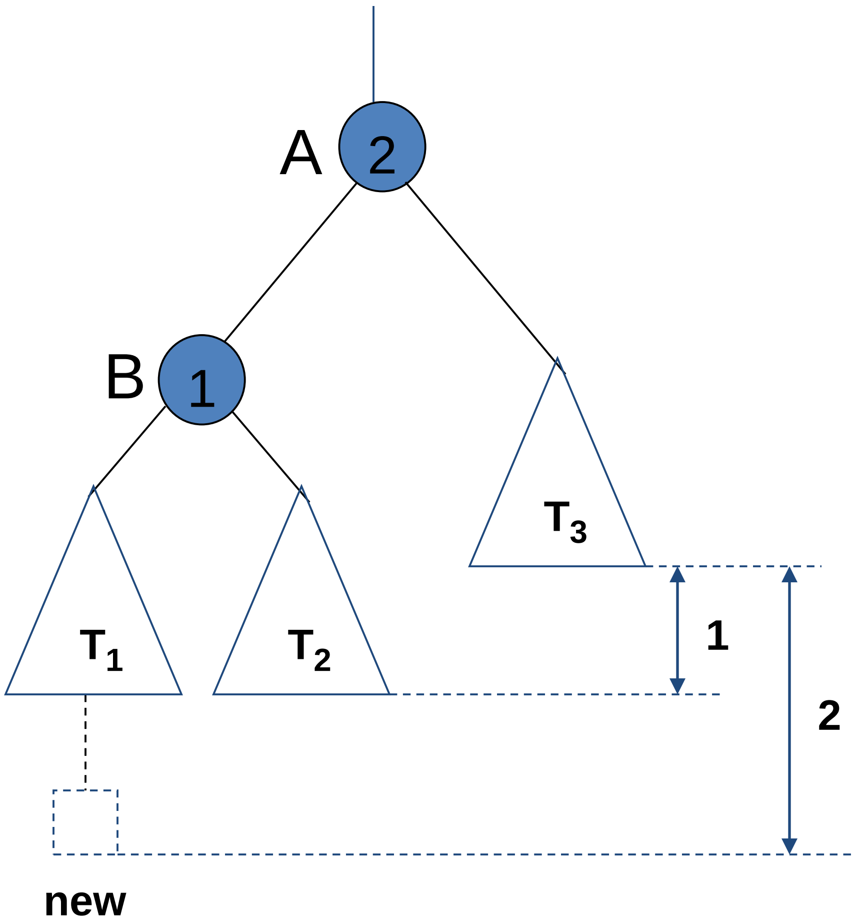
Consider the case of node that was previously 1



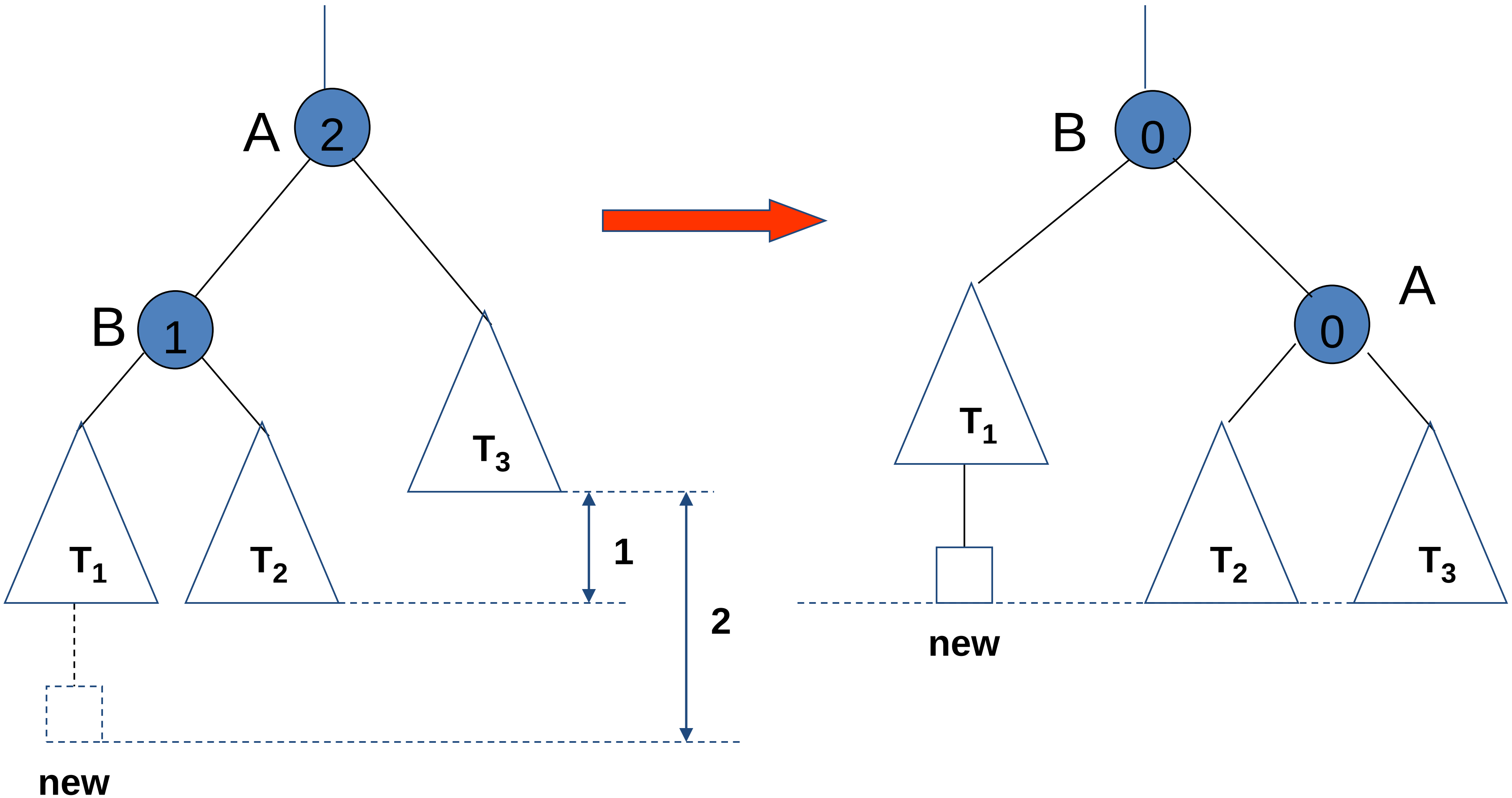
Inserting New Node in AVL Tree



Inserting New Node in AVL Tree



Inserting New Node in AVL Tree

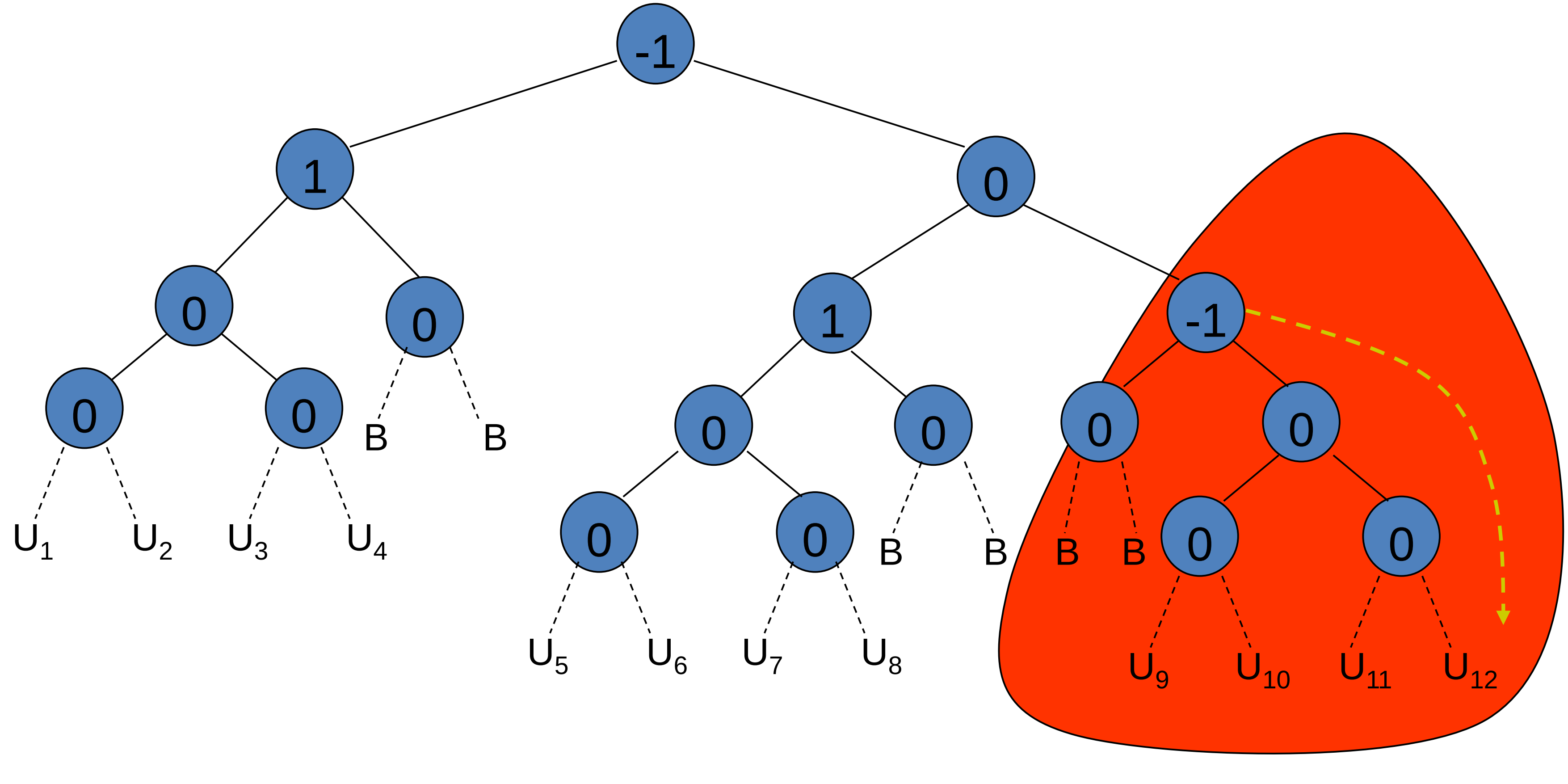


Inorder: T_1 B T_2 A T_3

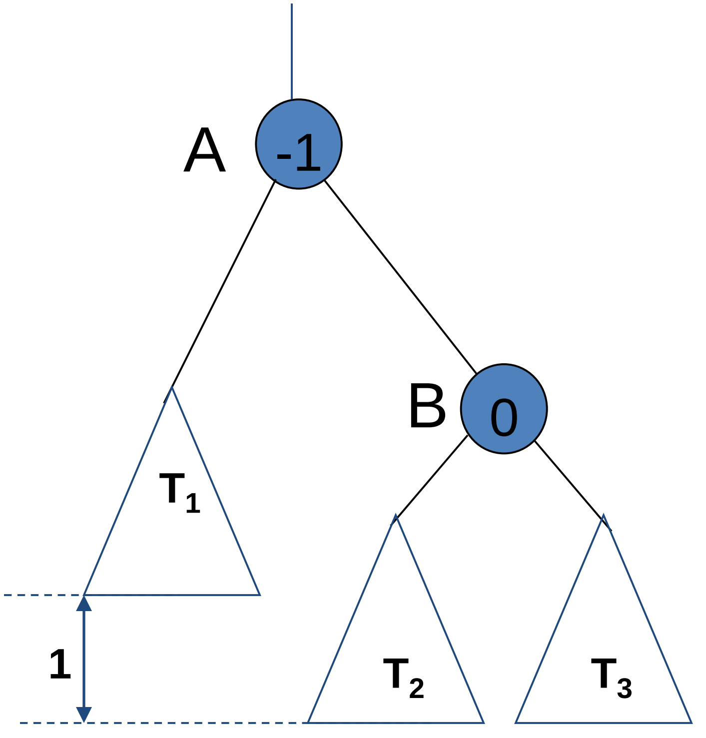
Inorder: T_1 B T_2 A T_3

Balanced Binary Tree

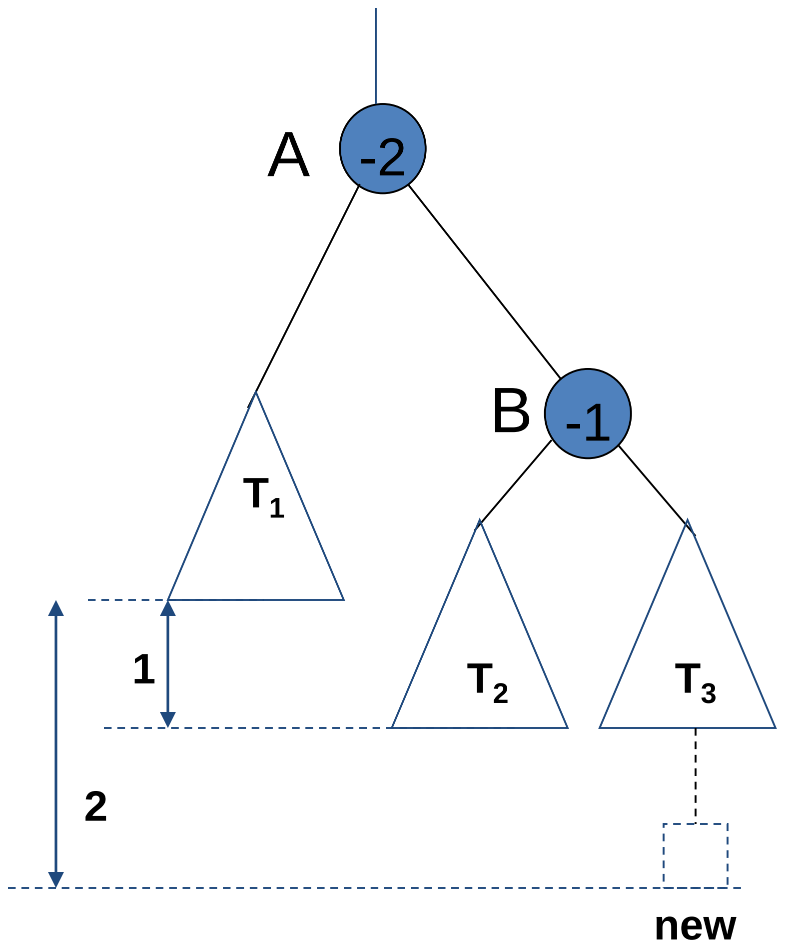
Consider the case of node that was previously 1



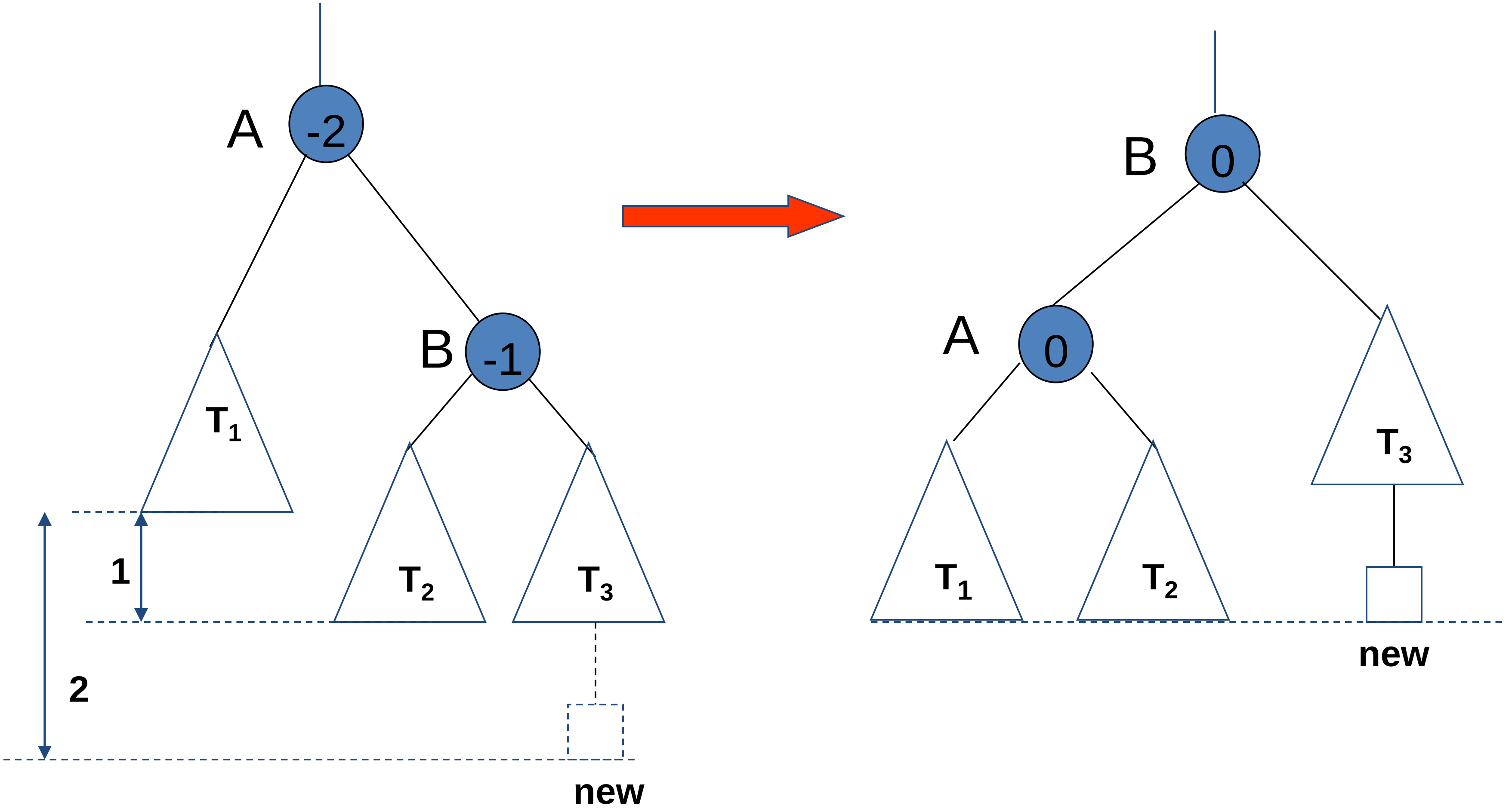
Inserting New Node in AVL Tree



Inserting New Node in AVL Tree



Inserting New Node in AVL Tree



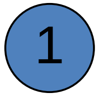
INSERTION

AVL Tree Building Example

- Let us work through an example that inserts numbers in a balanced search tree.
- We will check the balance after each insert and rebalance if necessary using rotations.

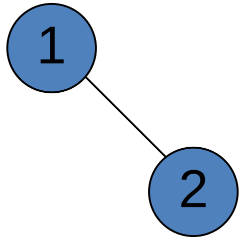
AVL Tree Building Example

Insert(1)



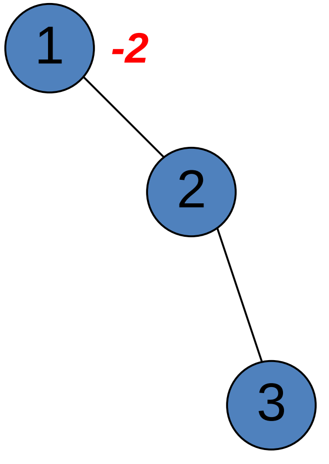
AVL Tree Building Example

Insert(2)



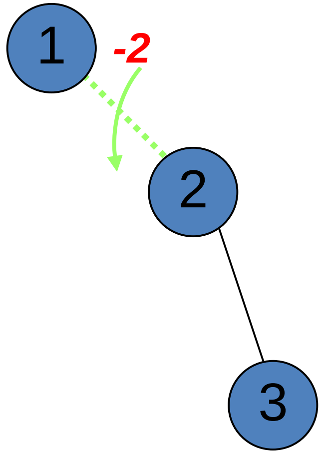
AVL Tree Building Example

Insert(3) single left rotation



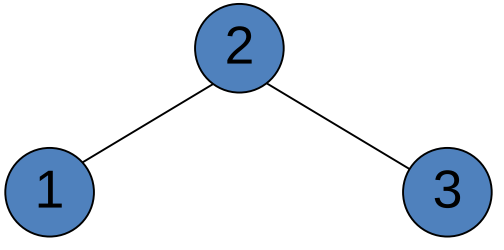
AVL Tree Building Example

Insert(3) single left rotation



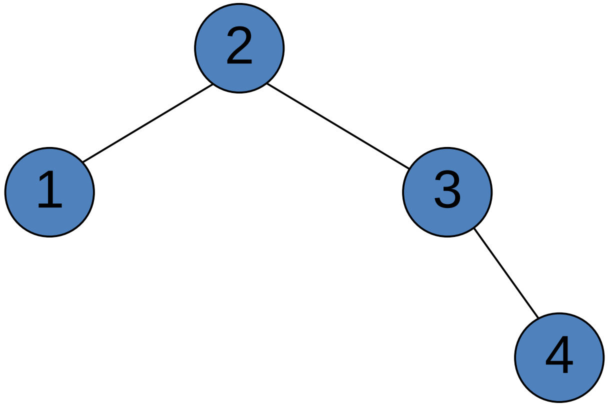
AVL Tree Building Example

Insert(3)



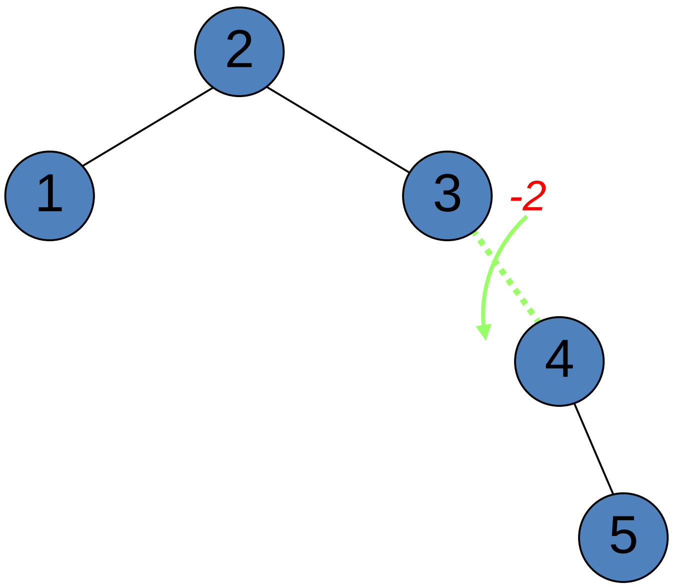
AVL Tree Building Example

Insert(4)



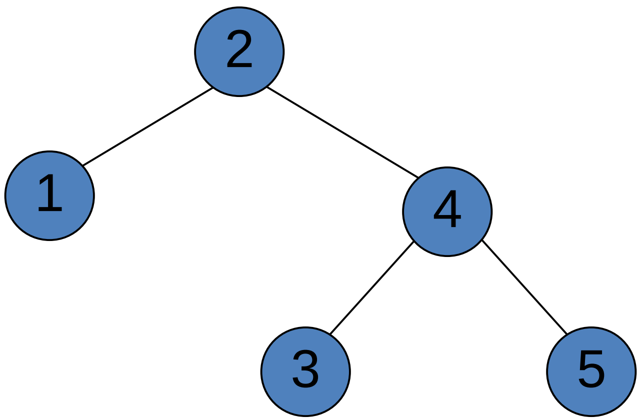
AVL Tree Building Example

Insert(5)



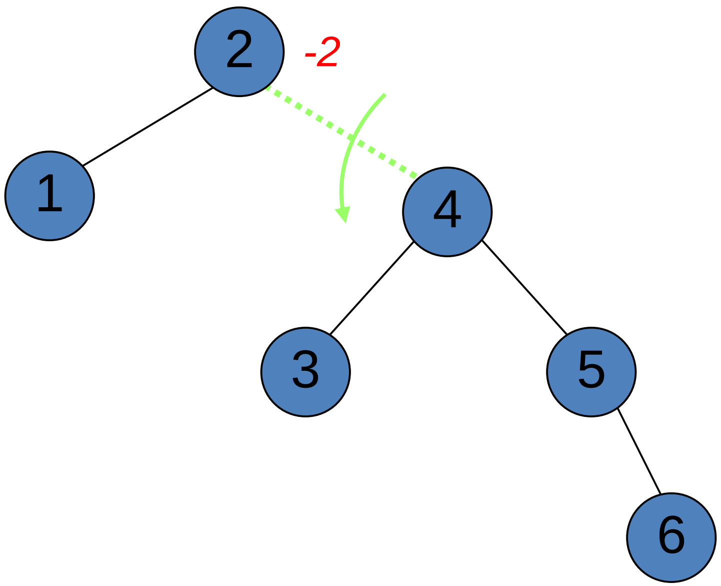
AVL Tree Building Example

Insert(5)



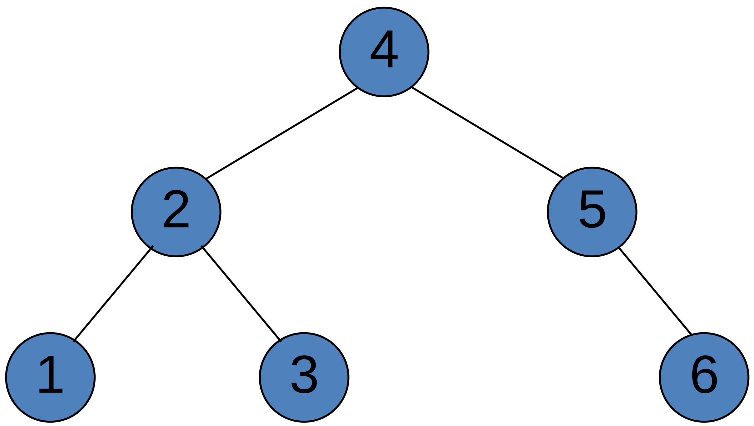
AVL Tree Building Example

Insert(6)



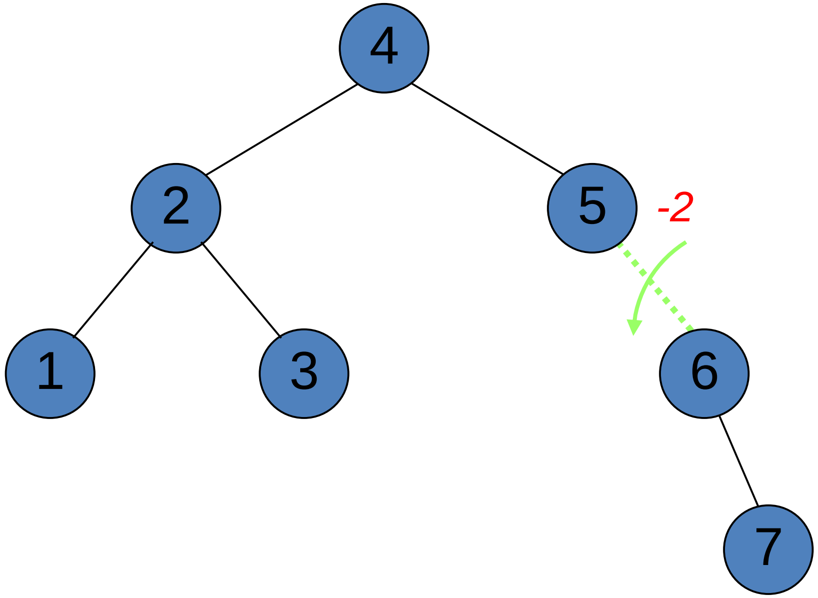
AVL Tree Building Example

Insert(6)



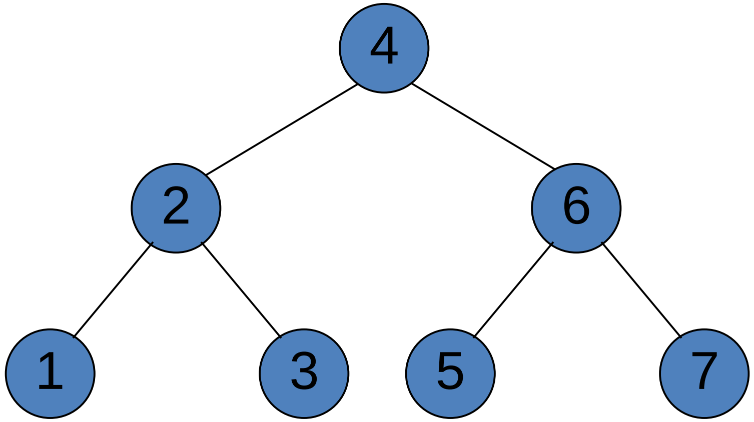
AVL Tree Building Example

Insert(7)



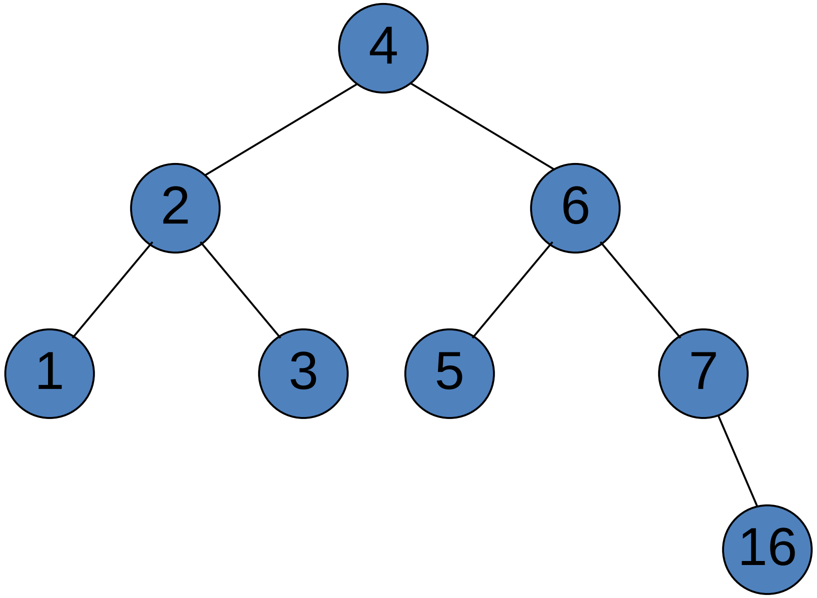
AVL Tree Building Example

Insert(7)



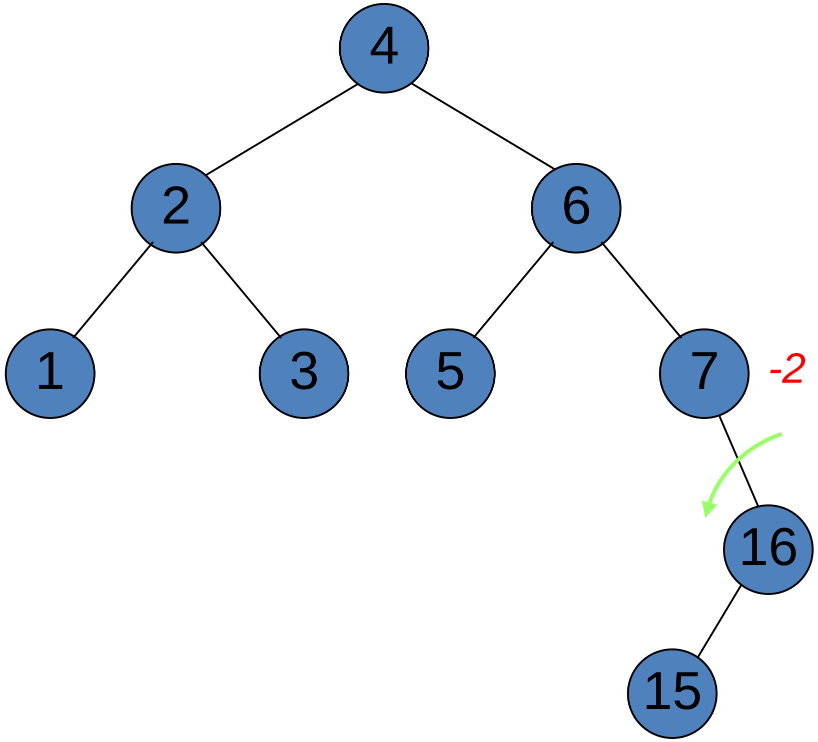
AVL Tree Building Example

Insert(16)



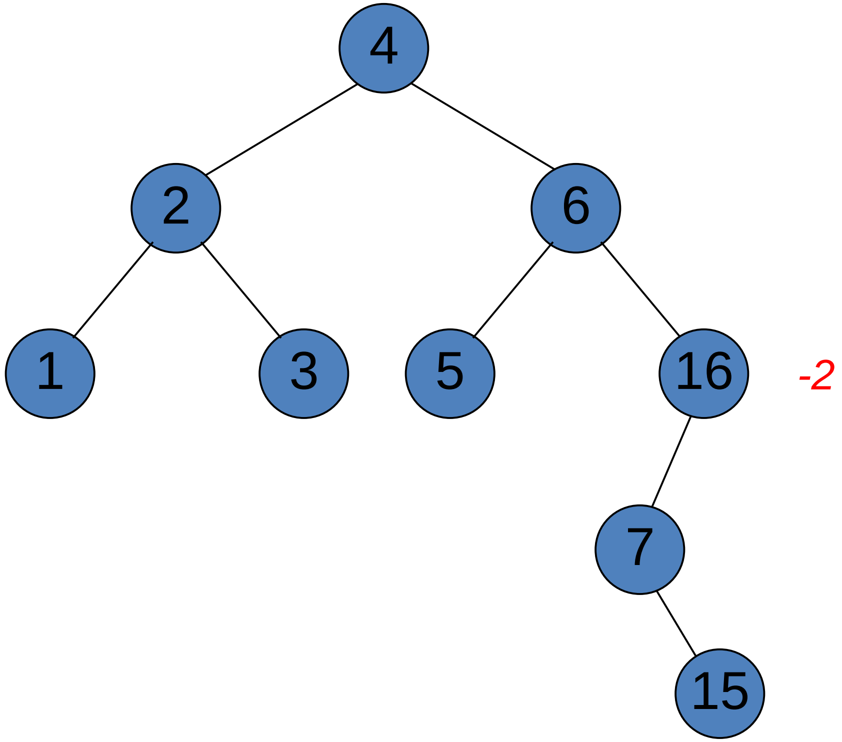
AVL Tree Building Example

Insert(15)



AVL Tree Building Example

Insert(15)



Cases for Rotation

- Single rotation does not seem to restore the balance.
- The problem is the node 15 is in an inner subtree that is too deep.
- Let us revisit the rotations.

Cases for Rotation

- Let us call the node that must be rebalanced X.
- Since any node has at most two children, and a height imbalance requires that X’s two subtrees differ by two (or –2), the violation will occur in four cases:

Cases for Rotation

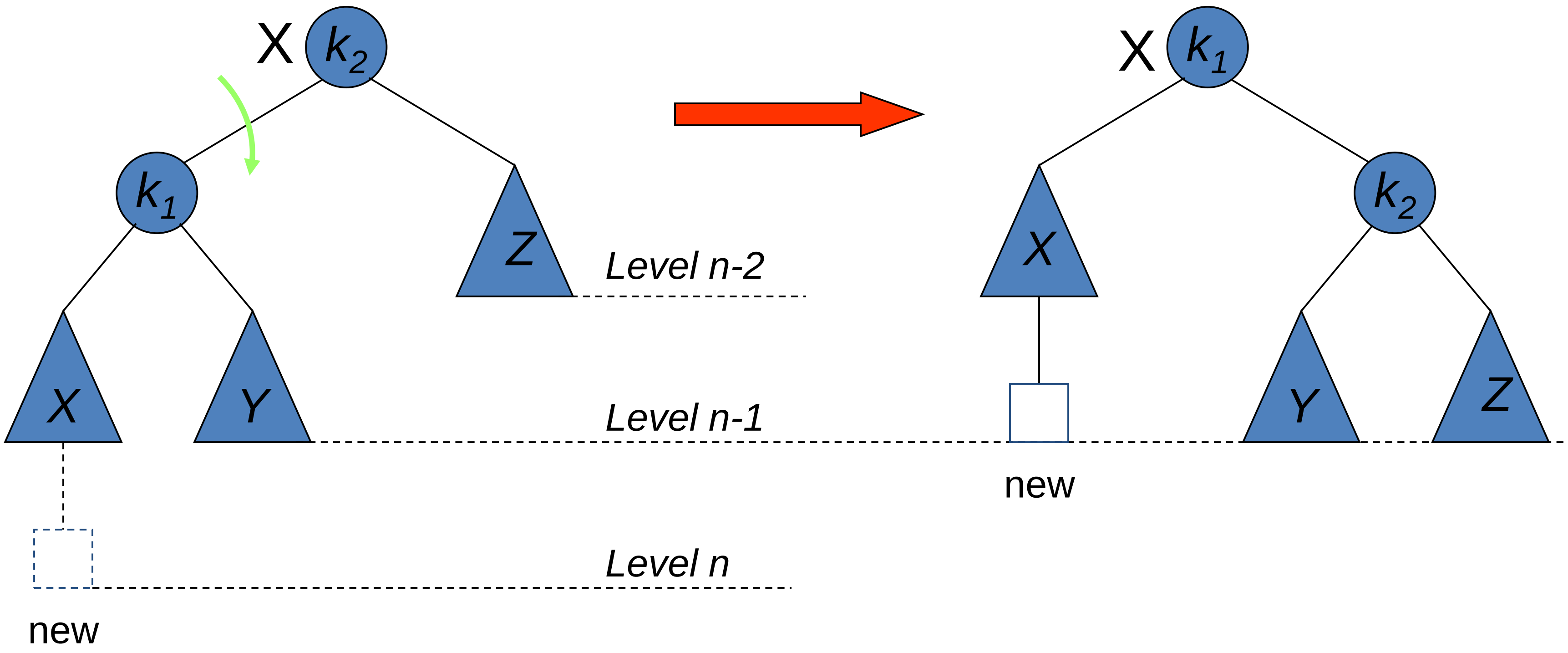
1. An insertion into left subtree of the left child of X.
2. An insertion into right subtree of the left child of X.
3. An insertion into left subtree of the right child of X.
4. An insertion into right subtree of the right child of X.

Cases for Rotation

- The insertion occurs on the “outside” (i.e., left-left or right-right) in cases 1 and 4
- Single rotation can fix the balance in cases 1 and 4.
- Insertion occurs on the “inside” in cases 2 and 3 which single rotation cannot fix.

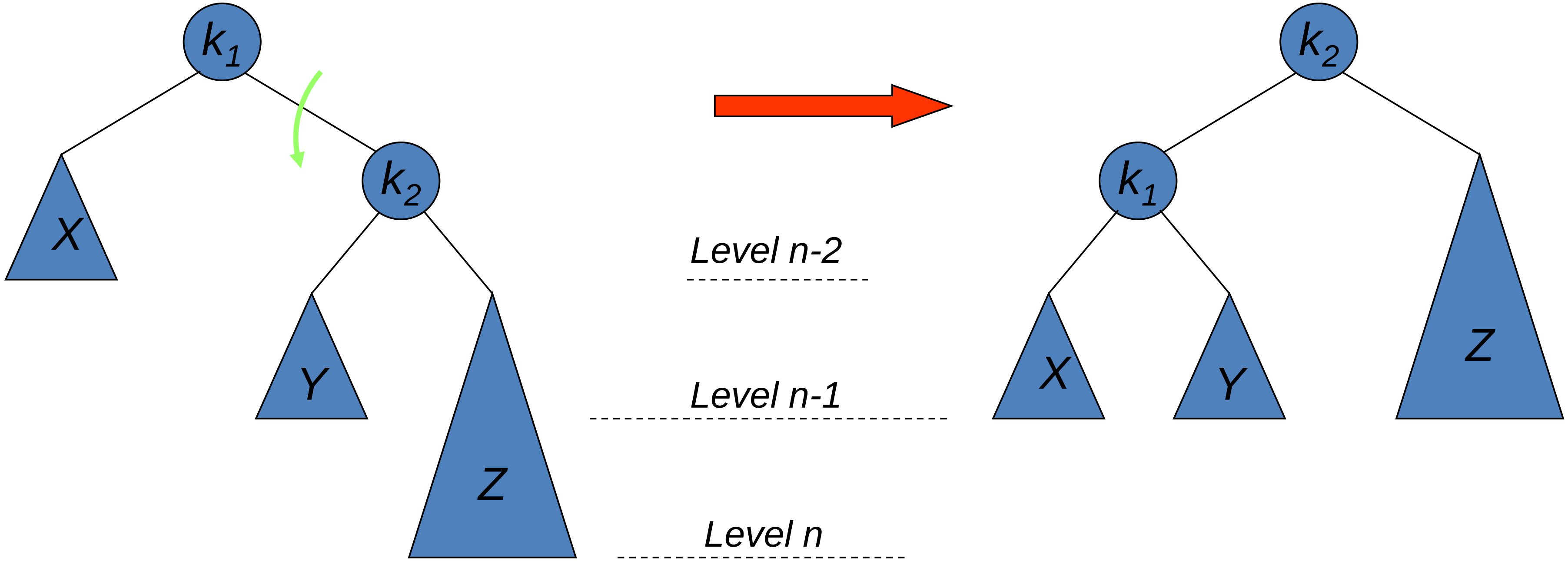
Cases for Rotation

- Single right rotation to fix case 1.



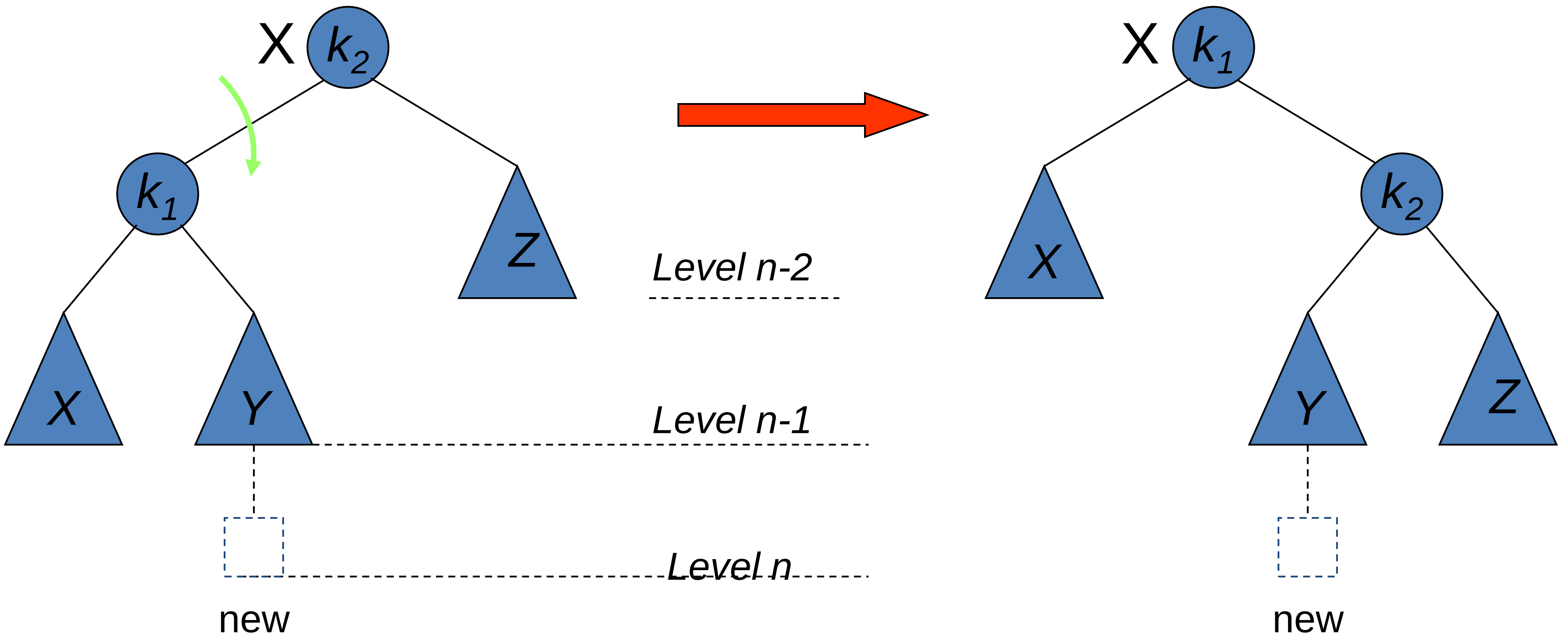
Cases for Rotation

- Single left rotation to fix case 4.

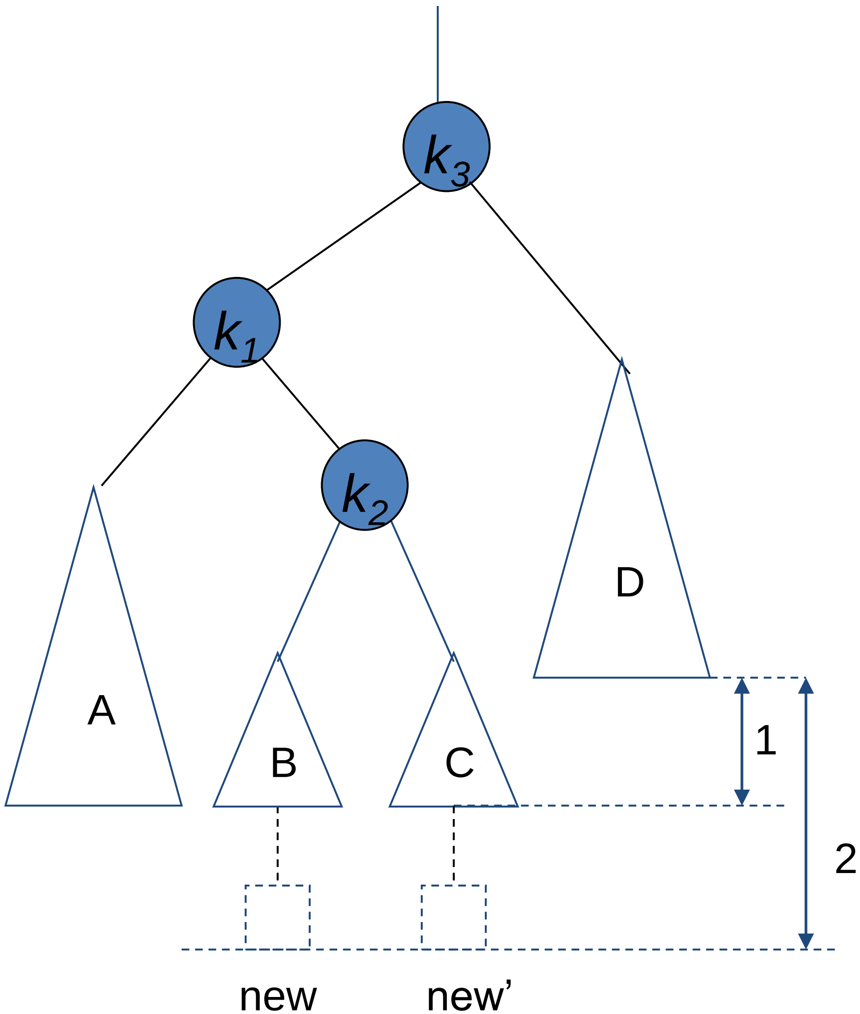


Cases for Rotation

- Single right rotation *fails* to fix case 2.



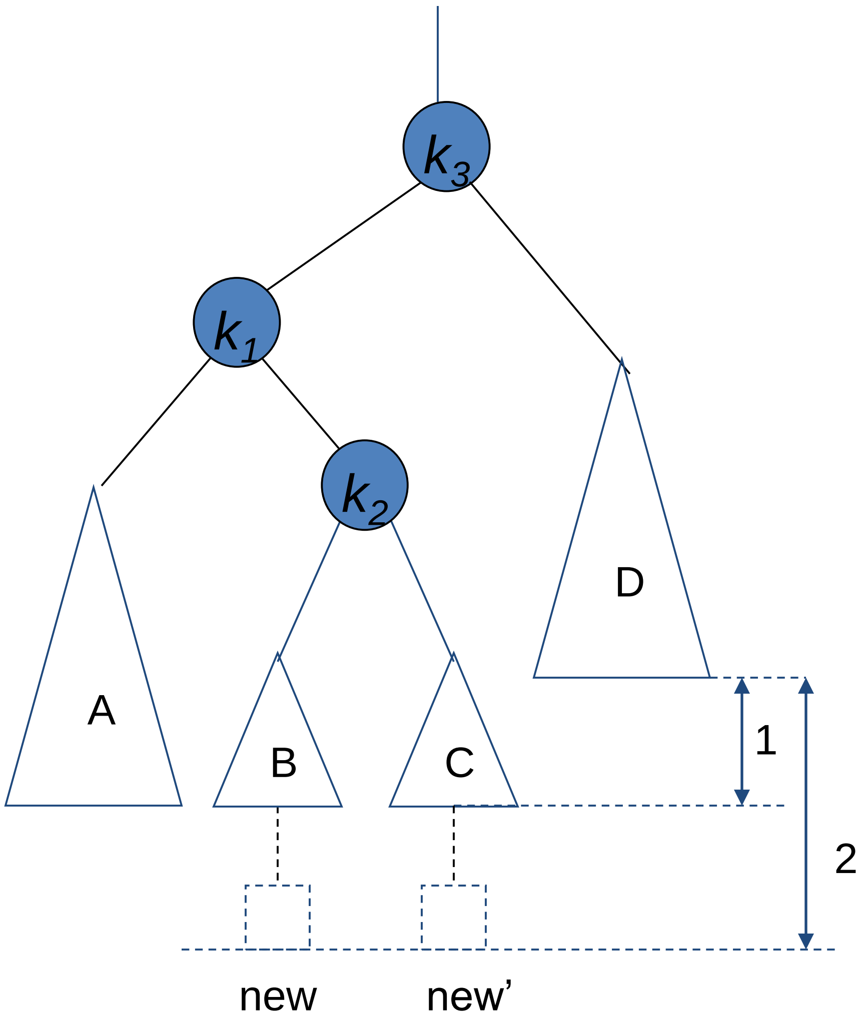
Cases for Rotation



New node inserted a either of the two spots

- Exactly one of tree *B* or *C* is two levels deeper than *D*; we are not sure which one.
- Good thing: it does not matter.
- To rebalance, k_3 cannot be left as the root.

Cases for Rotation

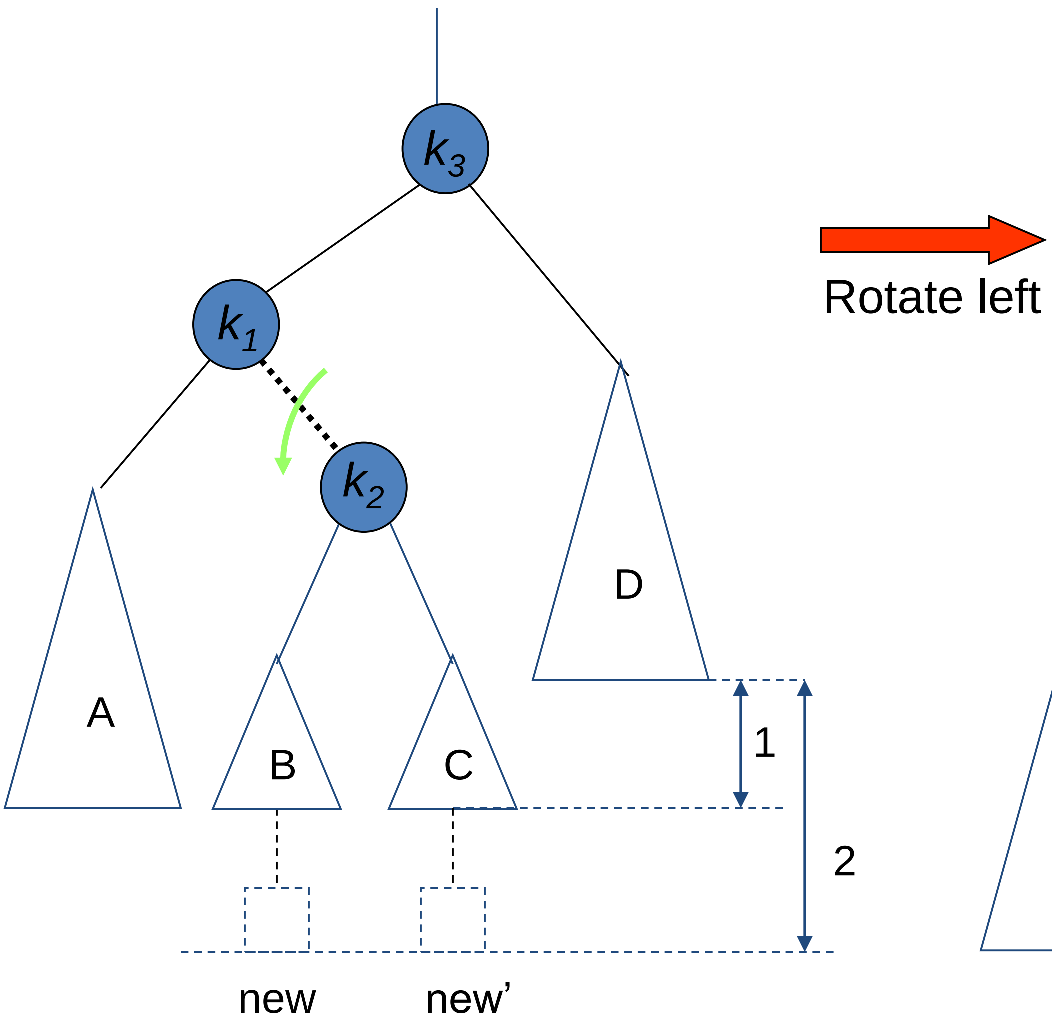


New node inserted a either of the two spots

- A rotation between k_3 and k_1 (k_3 was k_2 then) was shown to not work.
- The only alternative is to place k_2 as the new root.
- This forces k_1 to be k_2's left child and k_3 to be its right child.

Cases for Rotation

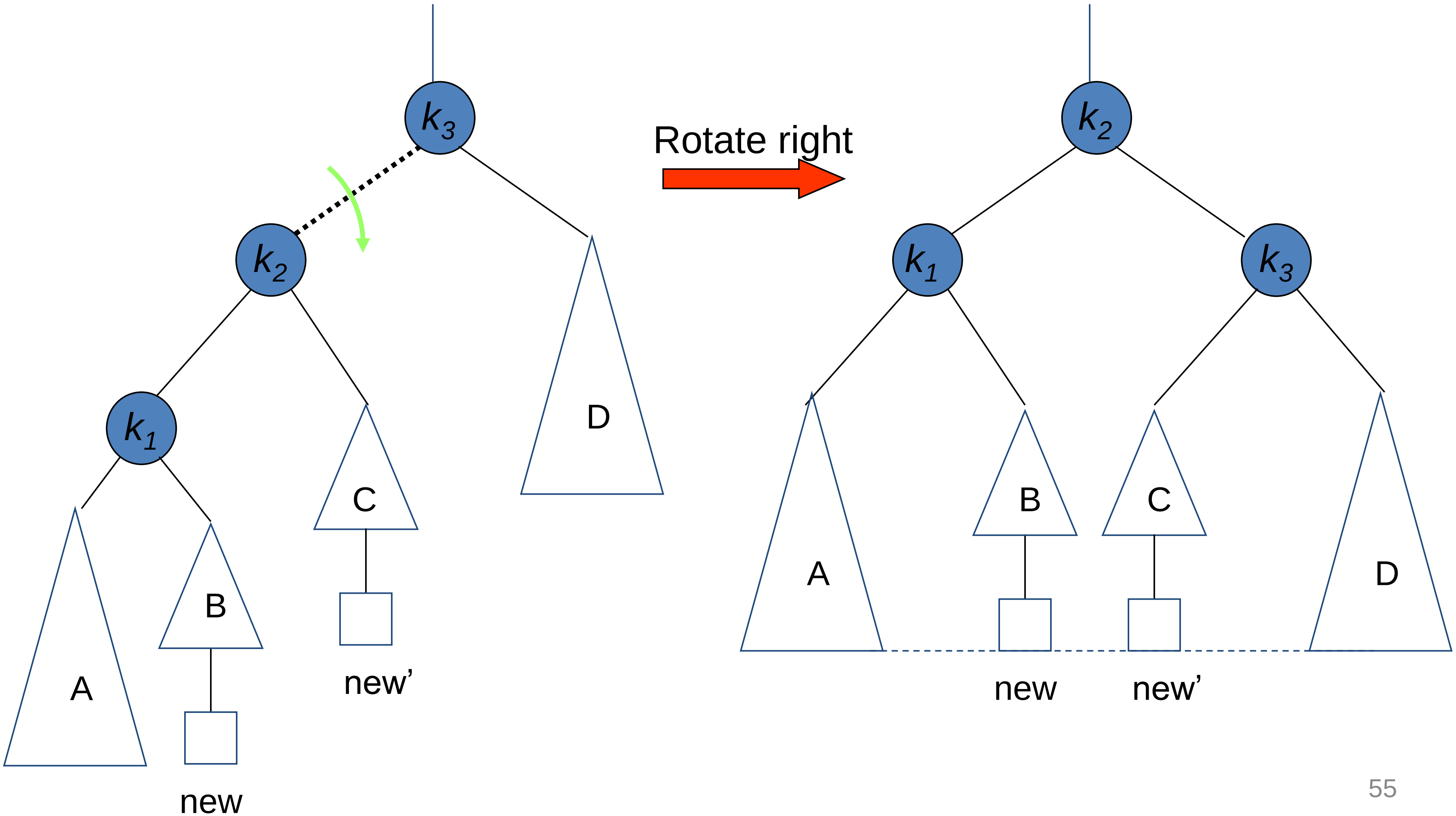
- Left-right *double* rotation to fix case 2.



New node inserted a either of the two spots

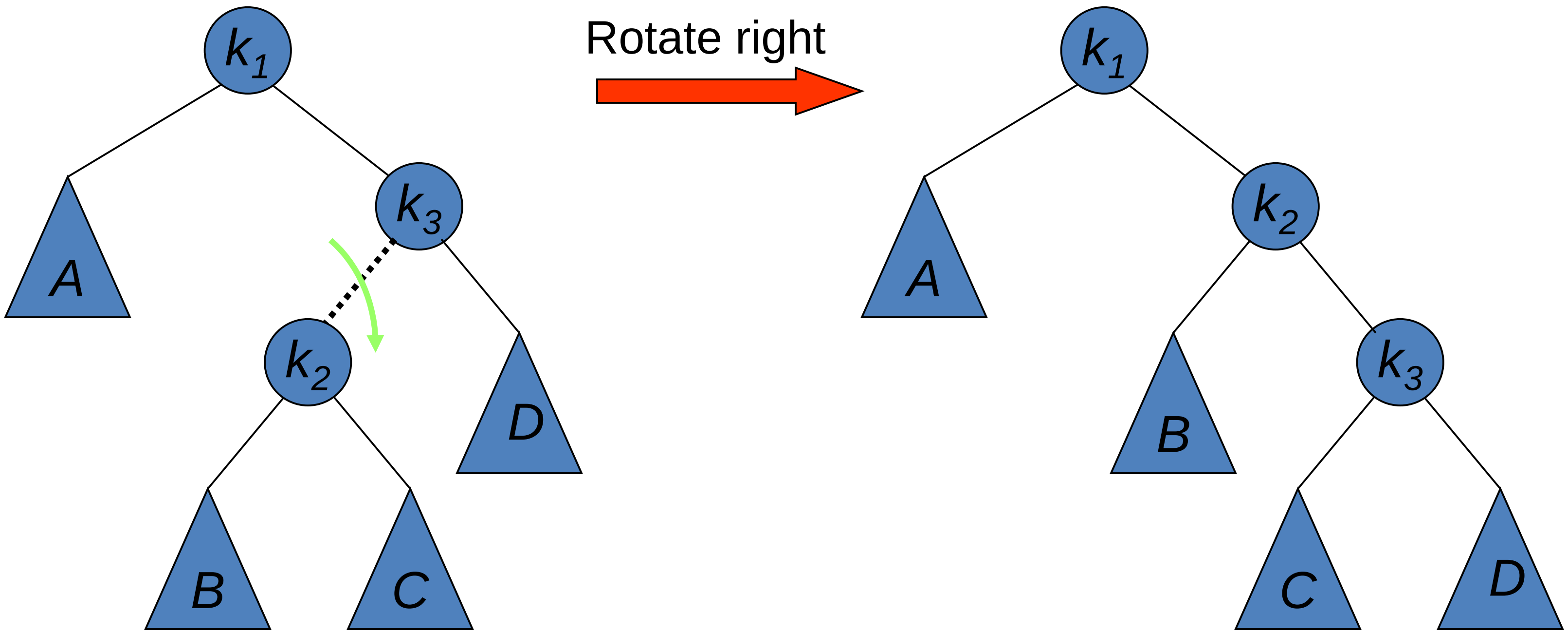
Cases for Rotation

- Left-right *double* rotation to fix case 2.



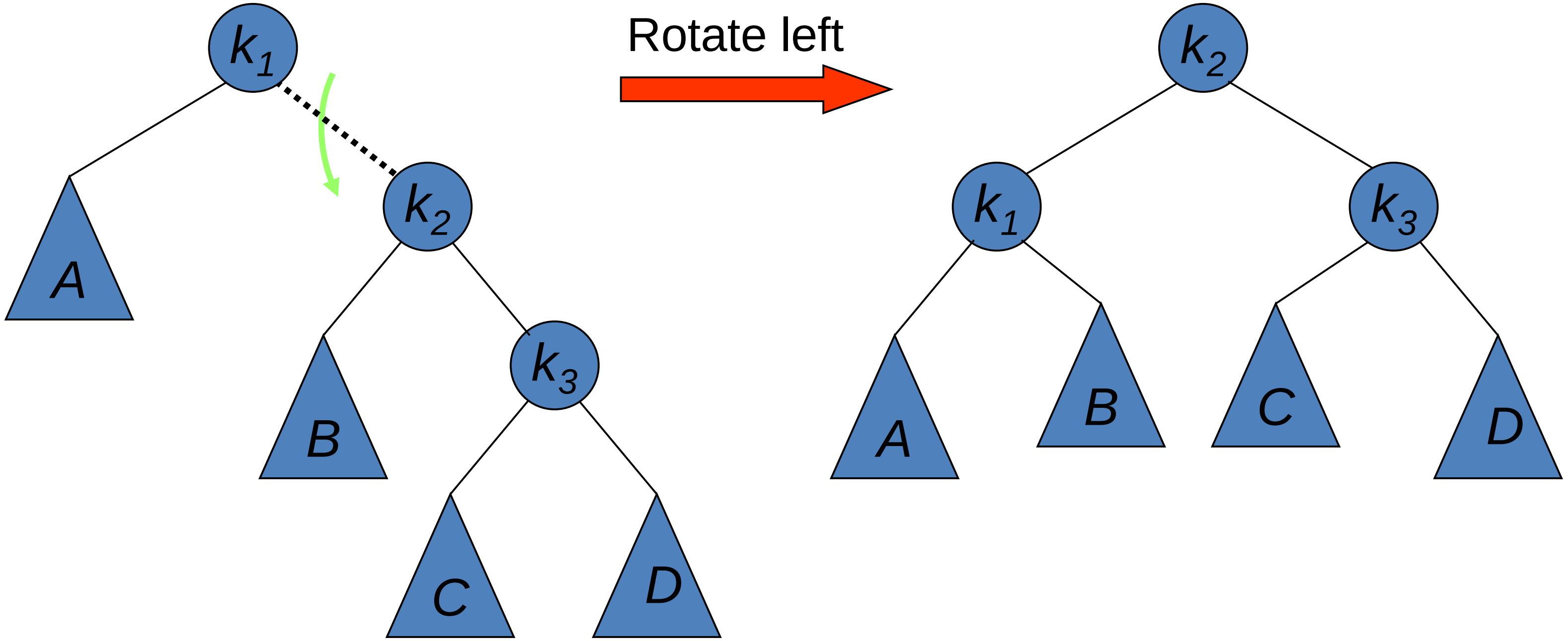
Cases for Rotation

- Right-left *double* rotation to fix case 3.



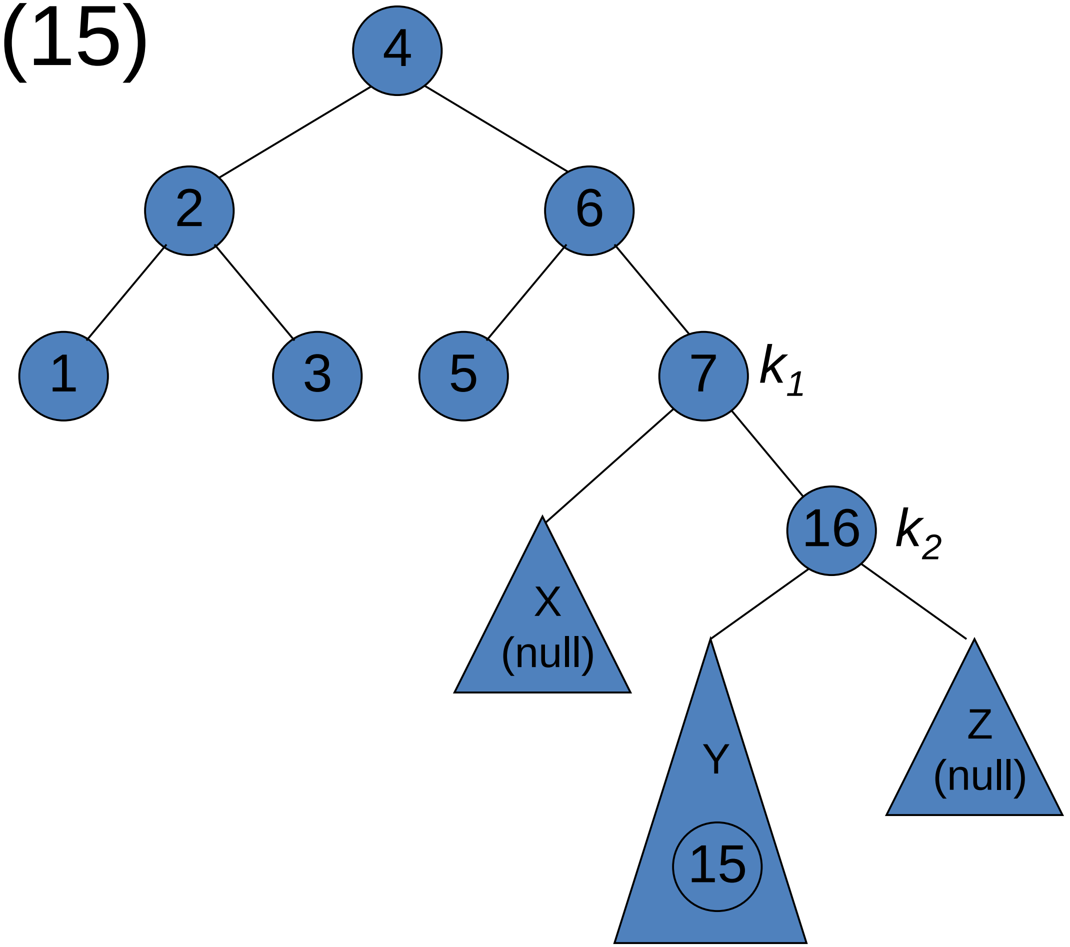
Cases for Rotation

- Right-left *double* rotation to fix case 3.



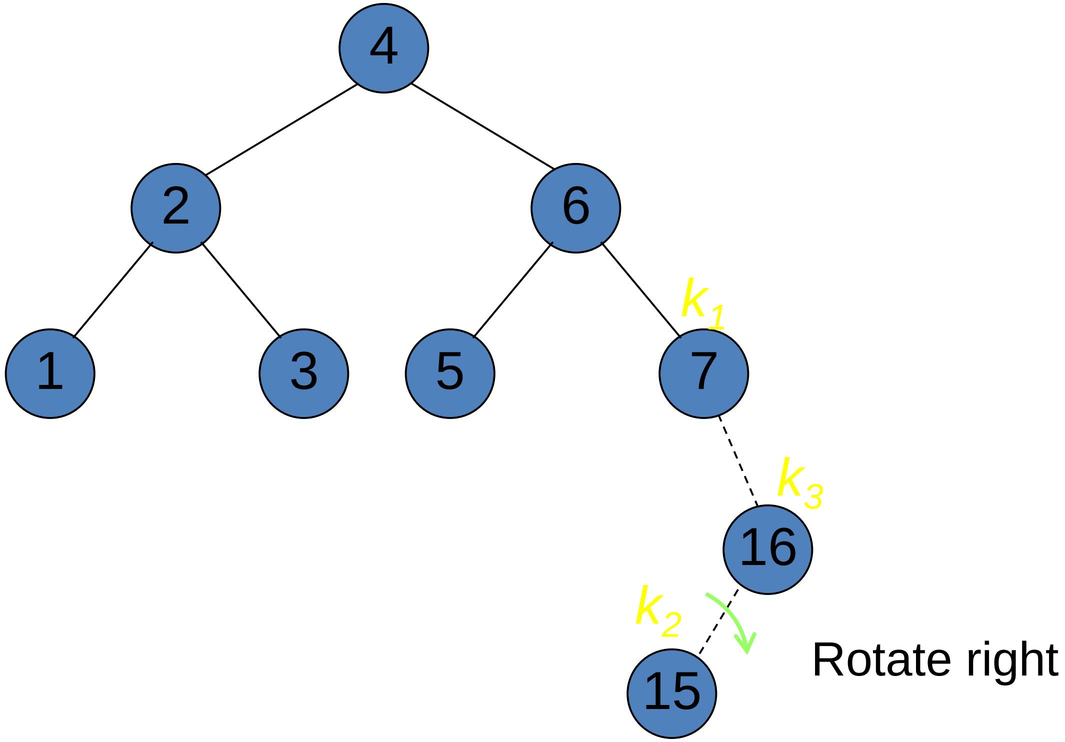
AVL Tree Building Example

Insert(15)



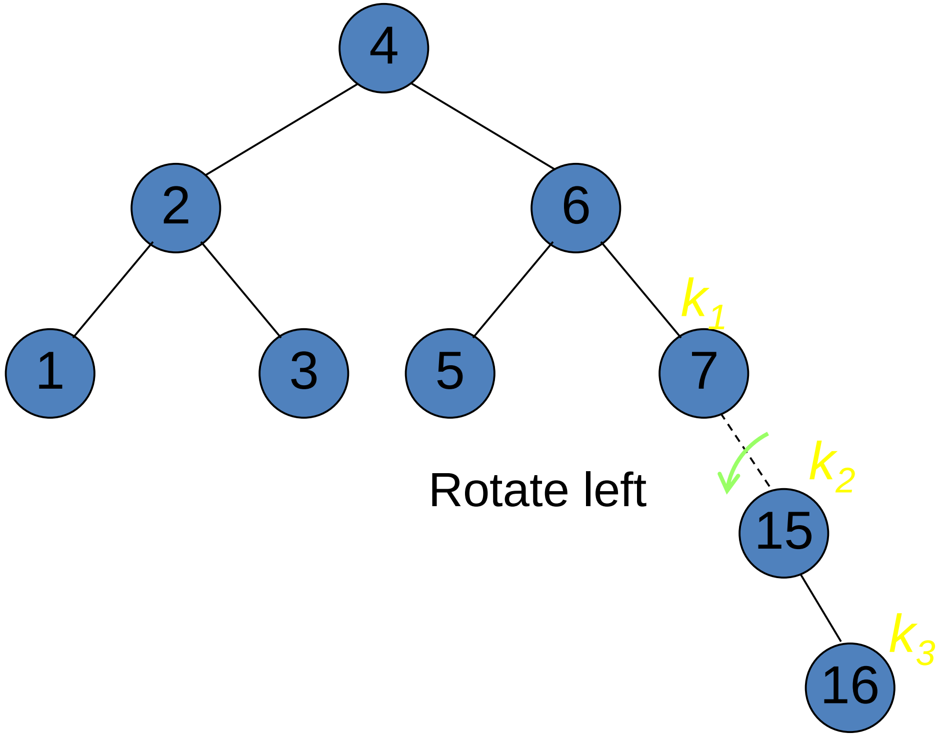
AVL Tree Building Example

Insert(15) right-left double rotation



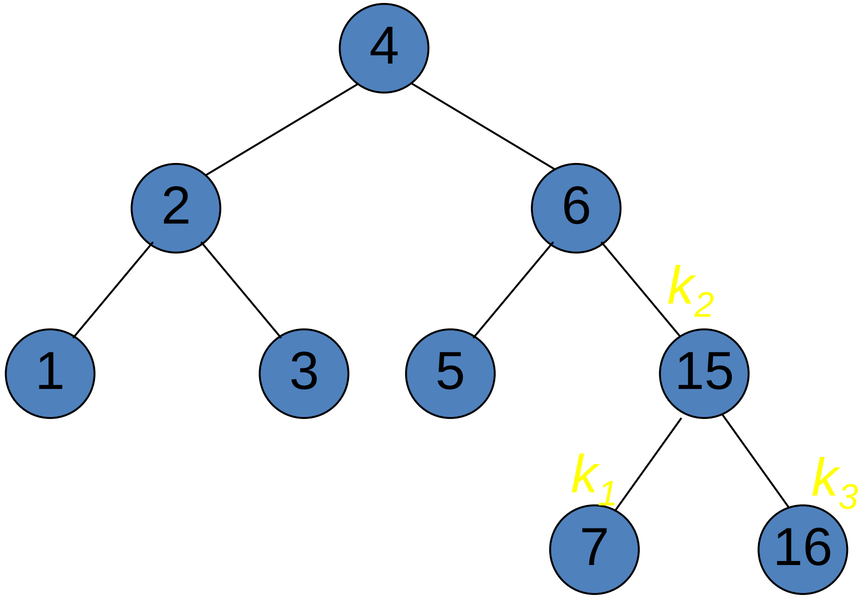
AVL Tree Building Example

Insert(15) right-left double rotation



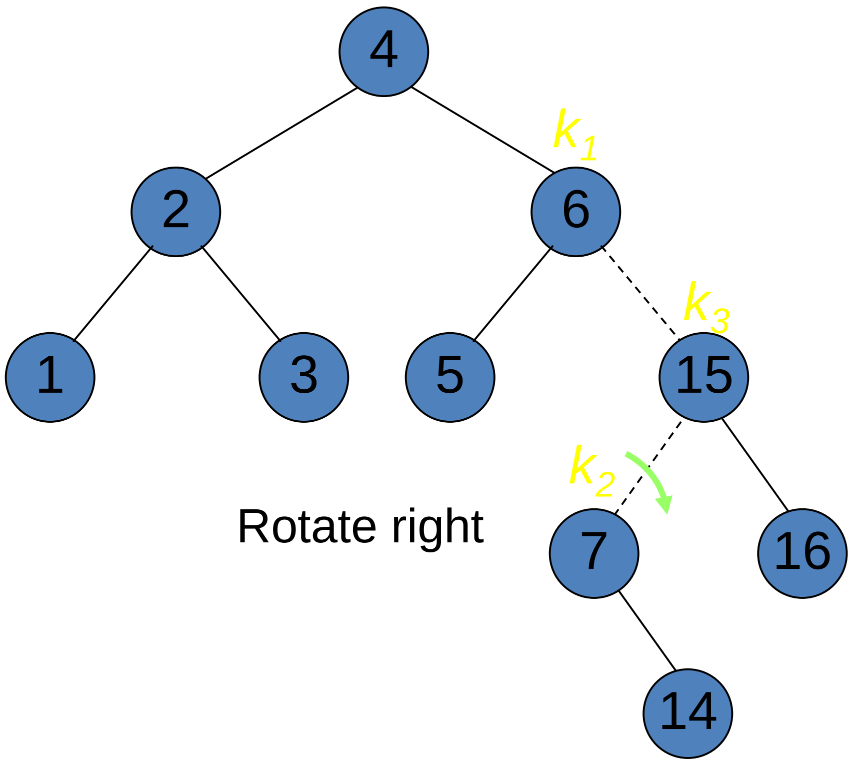
AVL Tree Building Example

Insert(15) right-left double rotation



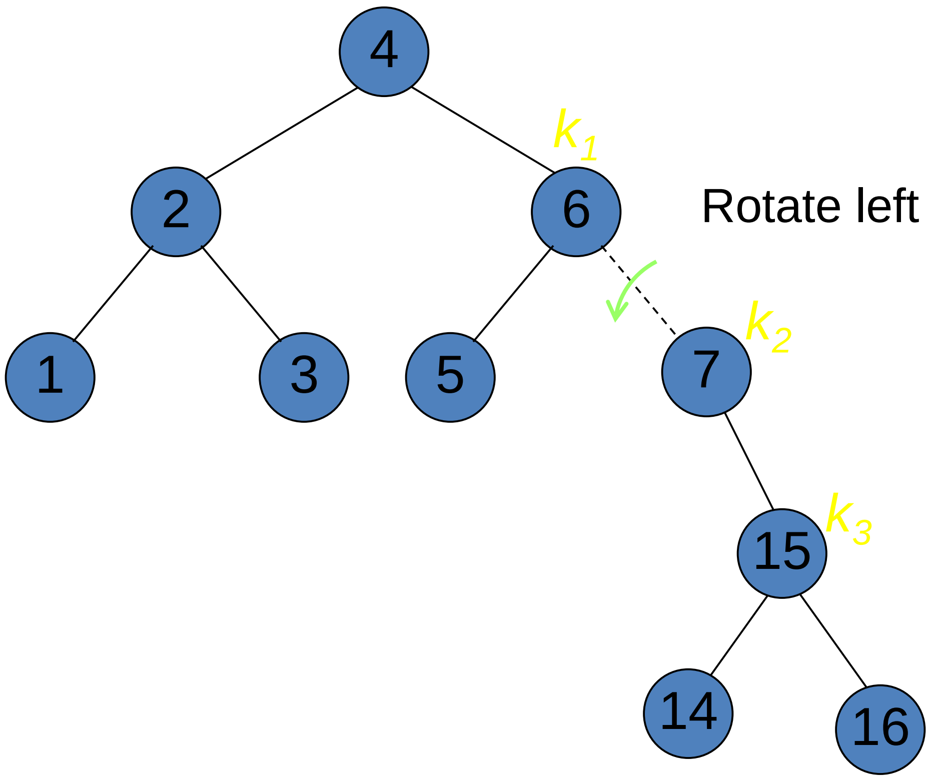
AVL Tree Building Example

Insert(14): right-left double rotation



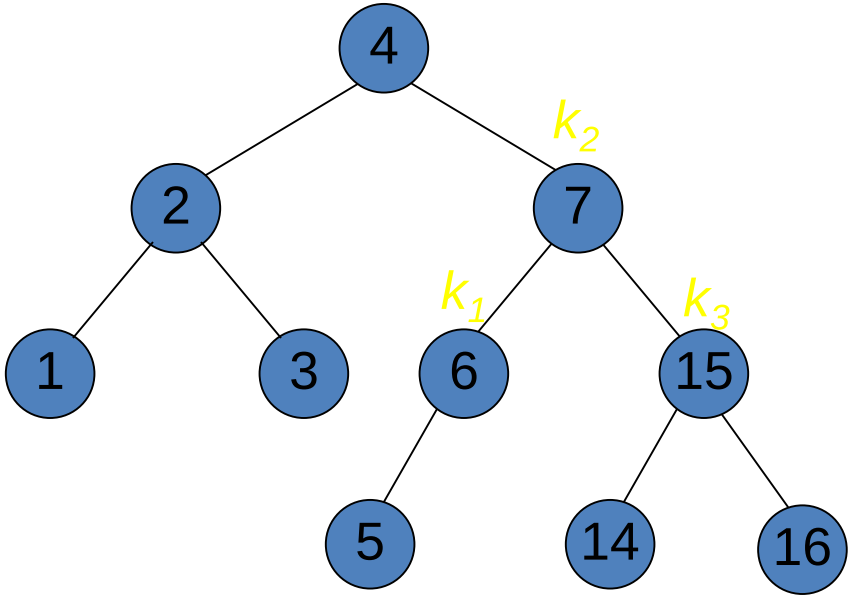
AVL Tree Building Example

Insert(14): right-left double rotation



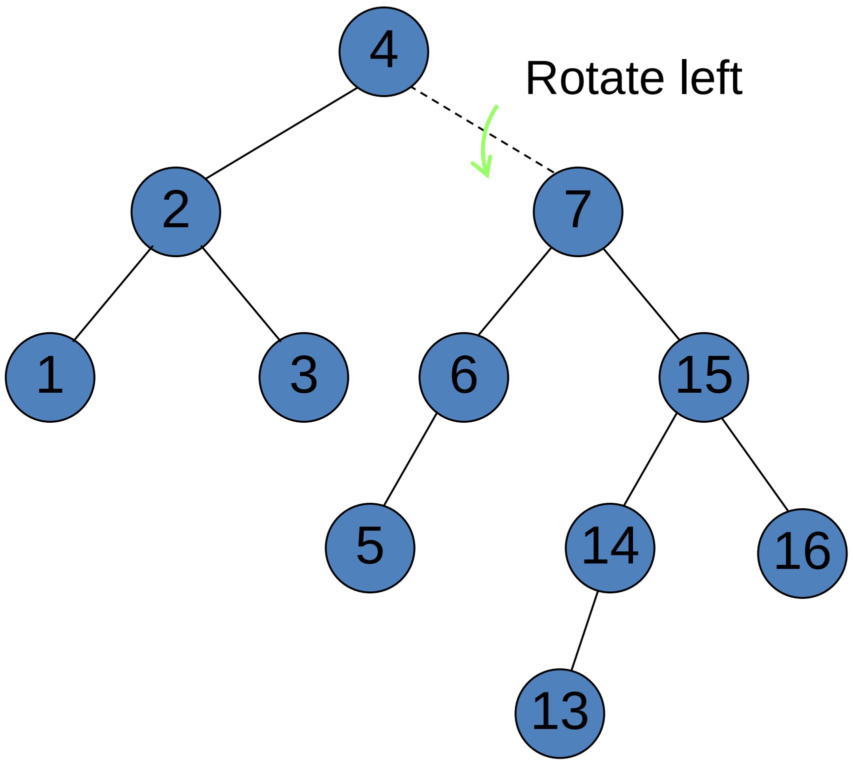
AVL Tree Building Example

Insert(14): right-left double rotation



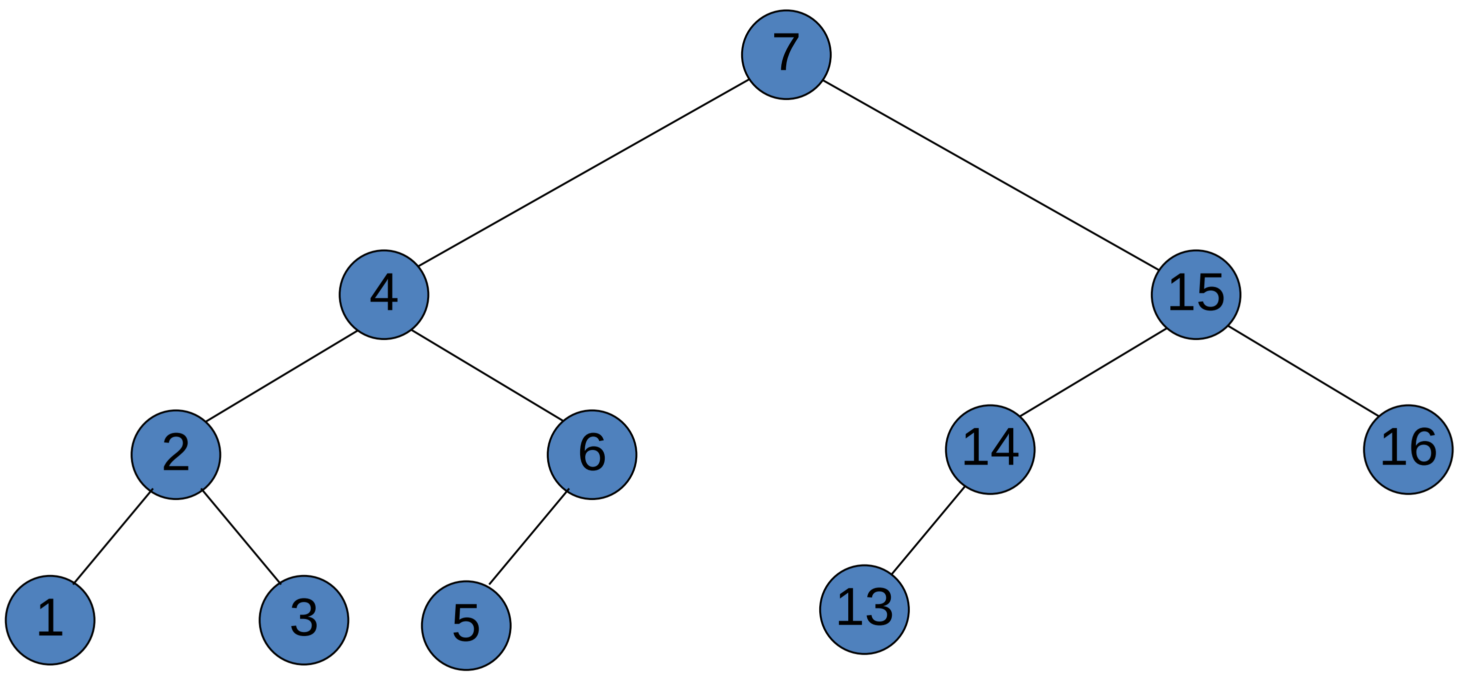
AVL Tree Building Example

Insert(13): single rotation



AVL Tree Building Example

Insert(13): single rotation



DELETION