Trees and hierarchical orders, ordered trees, Search trees

AVL Trees

AVL Trees

An AVL tree is a binary search tree such that for any node in the tree, the height of the left and right subtrees can differ by at most 1.

- AVL is named by its inventor **Adelson-Velskii** and **Landis**
- The key advantage of using an AVL tree is that it takes O(logn) time to perform search, insertion and deletion operations in average case as well as worst case (because the height of the tree is limited to O(logn)).
- The structure of an AVL tree is same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the *BalanceFactor*.

AVL Trees

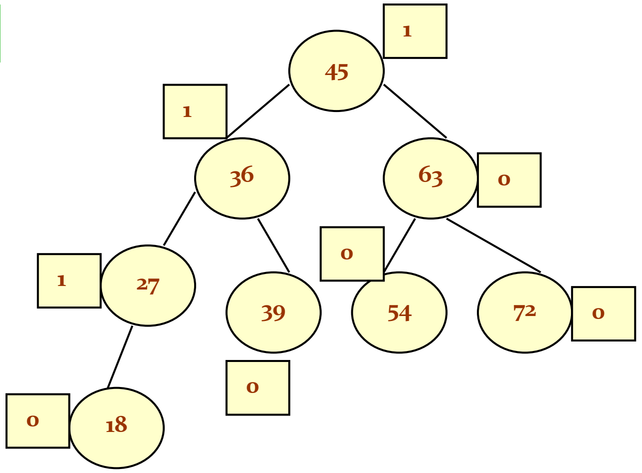
- The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree.

  Balance factor = Height (left sub-tree) – Height (right sub-tree)
- A binary search tree in which every node has a balance factor of -1, 0 or 1 is said to be height balanced.
- **A node with any other balance factor greater than |1| is considered to be unbalanced and requires rebalancing.**

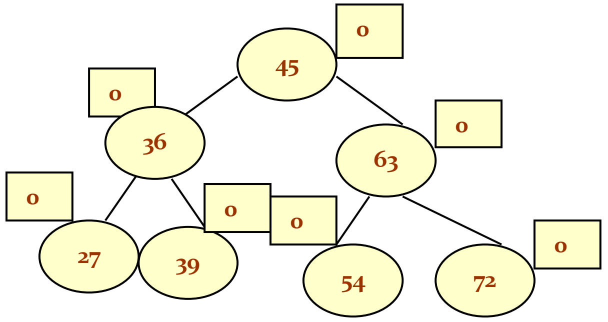
- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is called *Left-heavy tree*.
  - If the balance factor of a node is 0, then it means that the height of the left sub-tree is equal to the height of its right sub-tree.
  - If the balance factor of a node is -1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is called Right-*heavy tree*.

AVL Trees

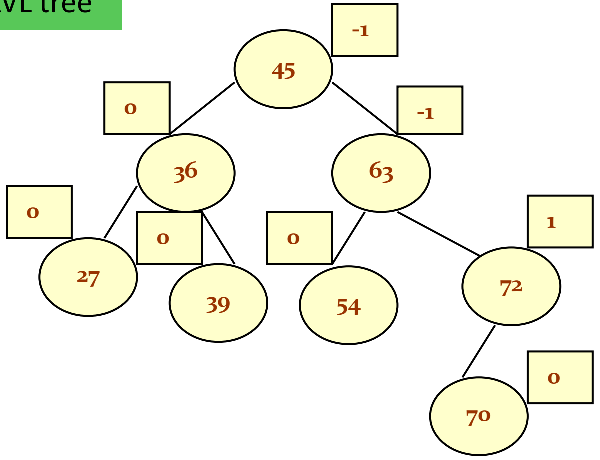
Left heavy AVL tree



Balanced AVL tree



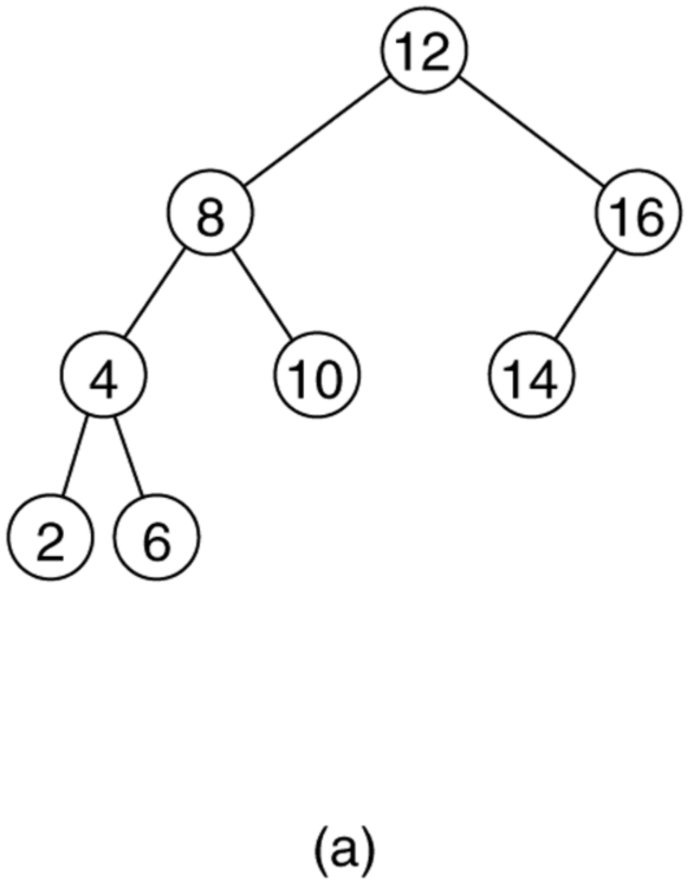
Right heavy AVL tree

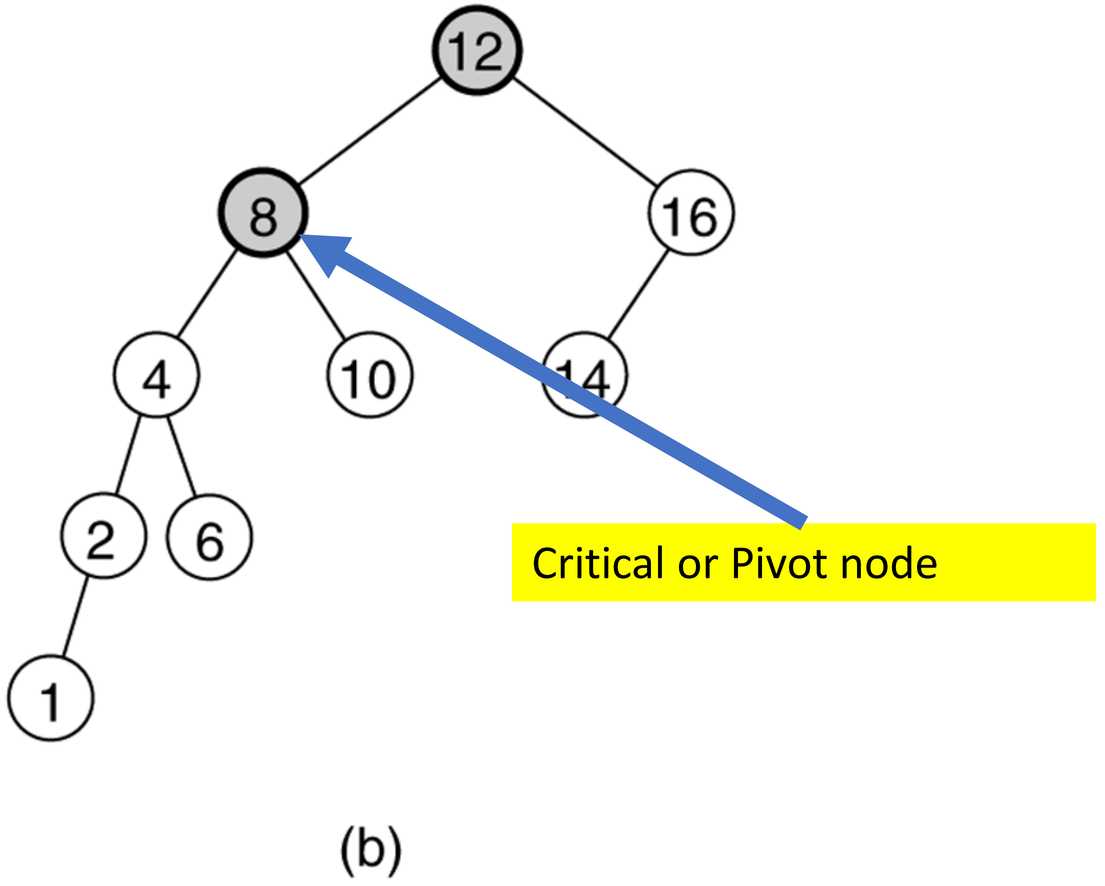


Searching for a Node in an AVL Tree

- Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree.
- Because of the height-balancing of the tree, the search operation takes O(log *n*) time to complete.
- Since the operation does not modify the structure of the tree, no special provisions need to be taken.

Two binary search trees: (a) an AVL tree; (b) not an AVL tree (unbalanced nodes are darkened)





Properties

- The depth of a typical node in an AVL tree is very close to the optimal *log N*.
- Consequently, all searching operations in an AVL tree have logarithmic worst-case bounds.
- An update (insert or remove) in an AVL tree could destroy the balance. It must then be rebalanced before the operation can be considered complete.
- After an insertion, only nodes that are on the path from the insertion point to the root can have their balances altered.

Inserting a Node in an AVL Tree

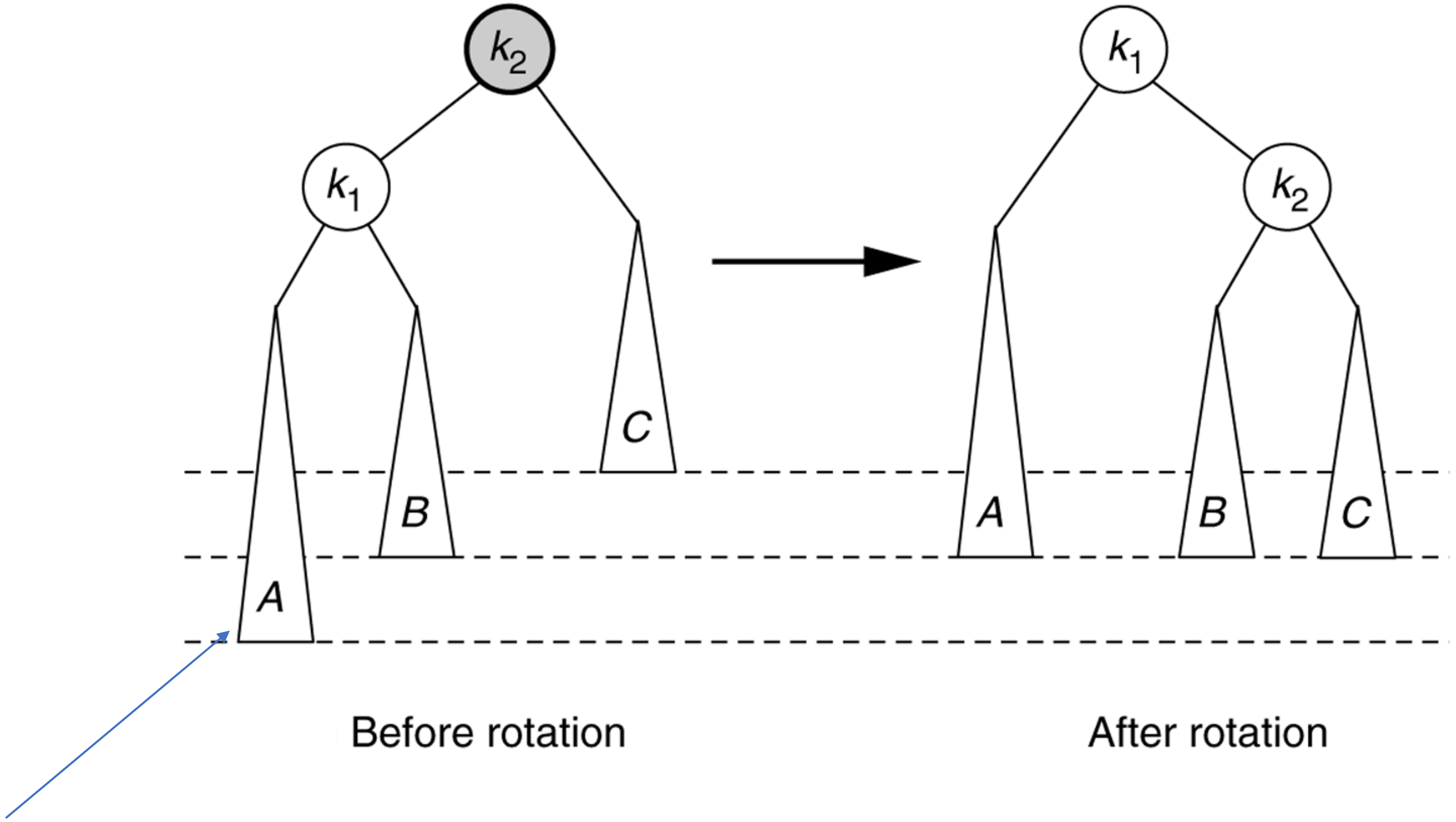
- Since an AVL tree is also a variant of binary search tree, insertion is also done in the same way as it is done in case of a binary search tree.
- Like in binary search tree, the new node is always inserted as the leaf node. But the step of insertion is usually followed by an additional step of rotation.
- **Rotation is done to restore the balance of the tree. However, if insertion of the new node does not disturb the balance factor, that is, if the balance factor of every node is still -1, 0 or 1, then rotations are not needed.**

Rotations to Balance AVL Trees

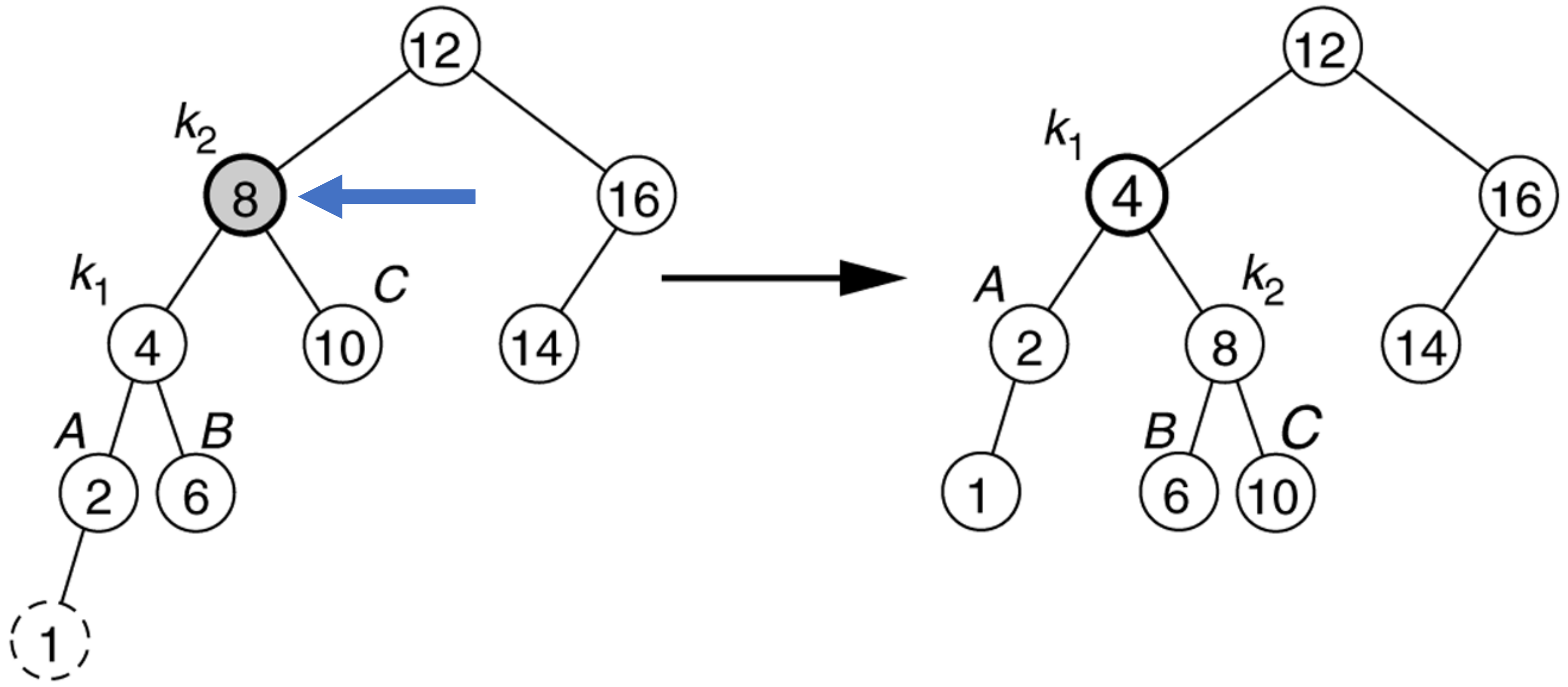
- To perform rotation, our first work is to find the *critical node*. Critical node is the nearest ancestor node on the path from the root to the inserted node whose balance factor is neither -1, 0 nor 1.
- The second task is to determine which type of rotation has to be done.
- There are four types of rebalancing rotations and their application depends on the position of the inserted node with reference to the critical node.

- ***LL rotation:***
  - **the new node is inserted in the left sub-tree of the left child of the critical node**
- ***RR rotation:***
  - **the new node is inserted in the right sub-tree of the right child of the critical node**
- ***LR rotation:***
  - **the new node is inserted in the right sub-tree of the left child of the critical node**
- ***RL rotation:***
  - **the new node is inserted in the left sub-tree of the right child of the critical node**

➢***LL rotation:*** **the new node is inserted in the left sub-tree of the left child of the critical node**



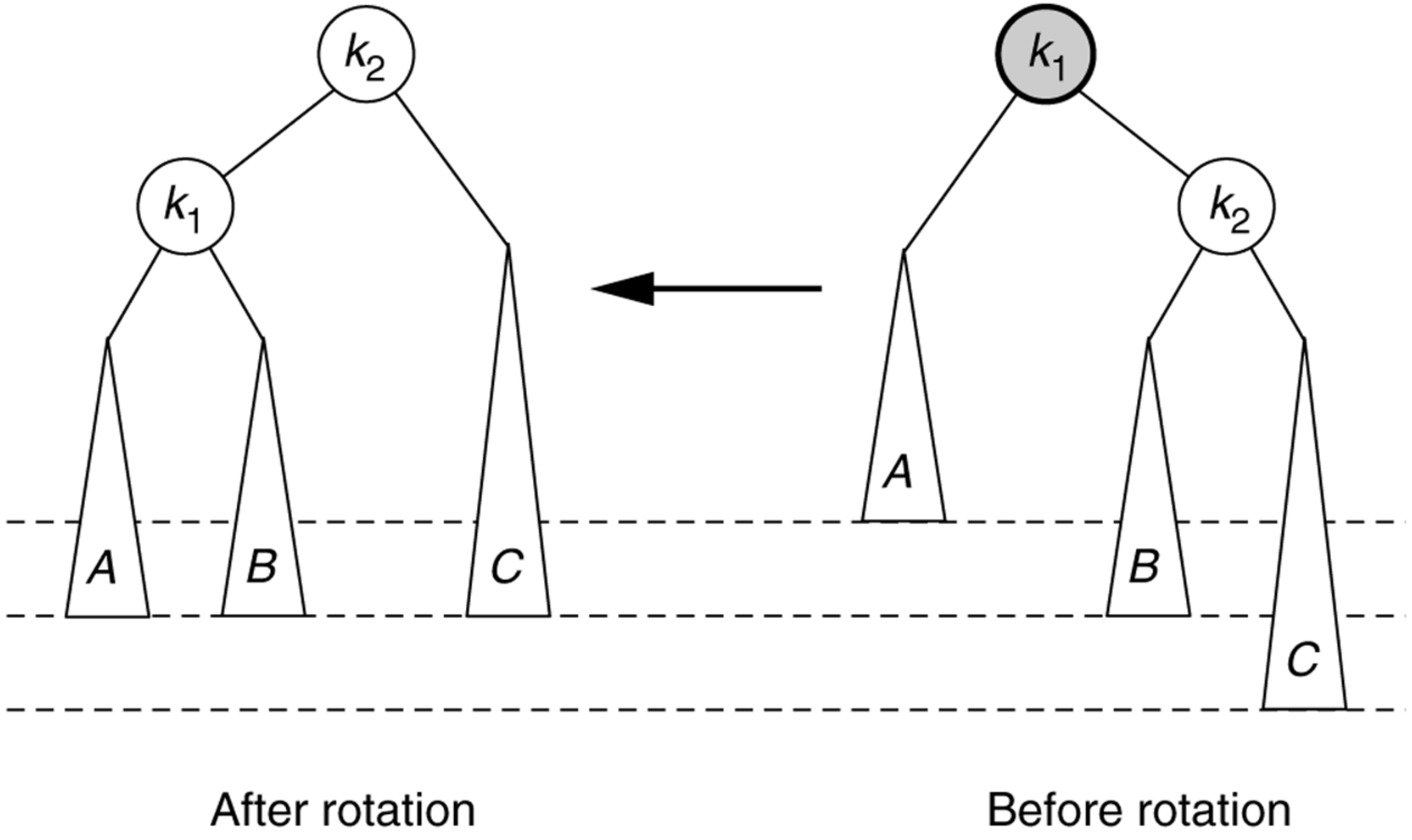
➢***LL rotation:*** **the new node is inserted in the left sub-tree of the left child of the critical node [Example]**



Before rotation

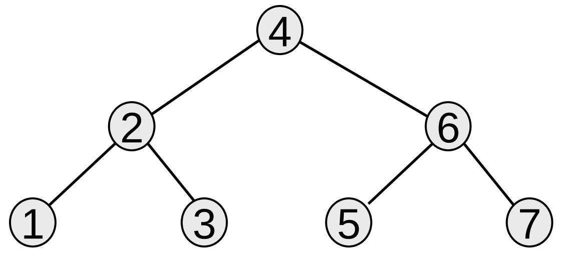
After rotation

➢***RR rotation:*** **the new node is inserted in the right sub-tree of the right child of the critical node**



➢*RR rotation:* the new node is inserted in the right sub-tree of the right child of the critical node [Example]

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.
- Answer:

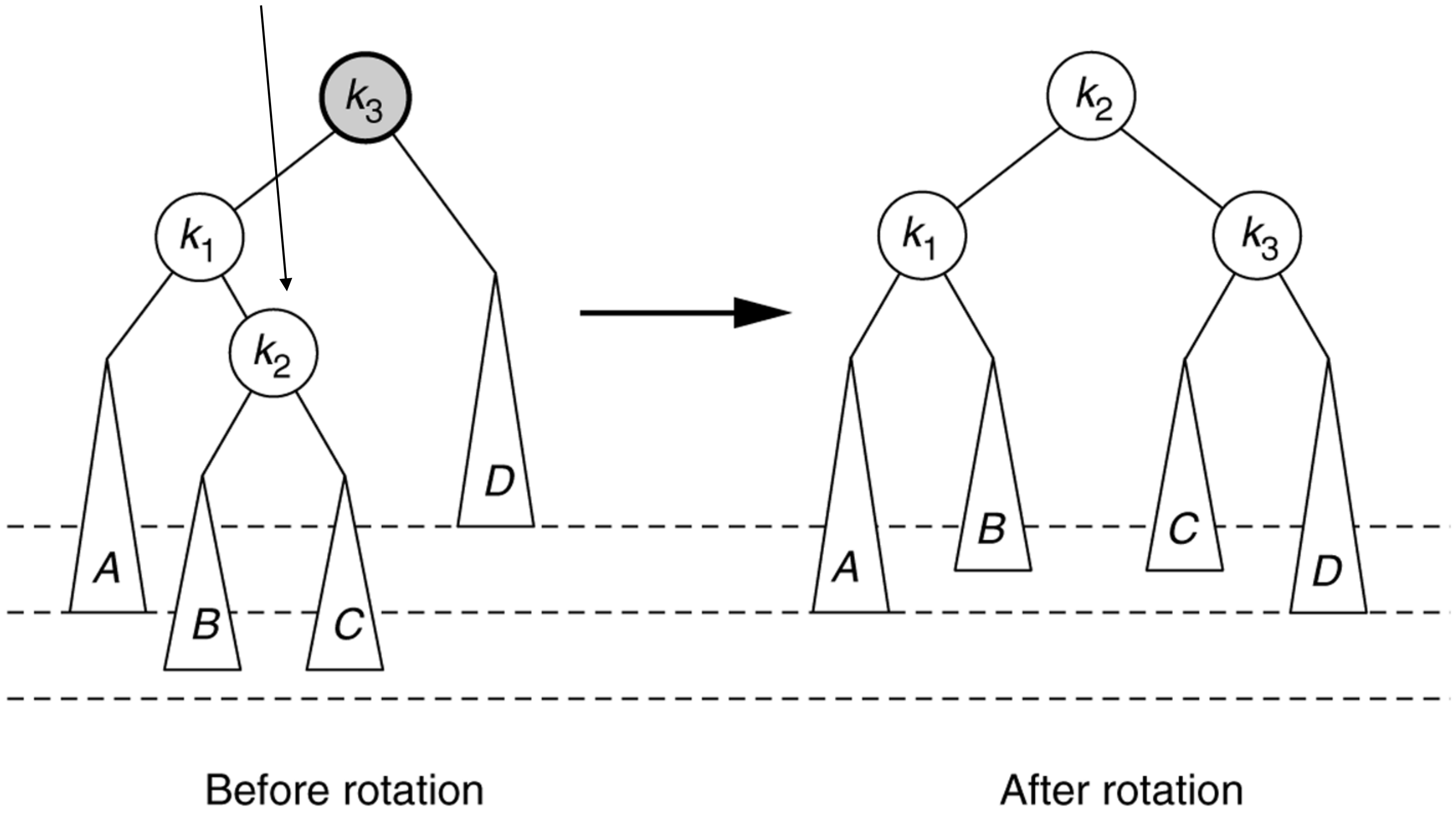


Analysis

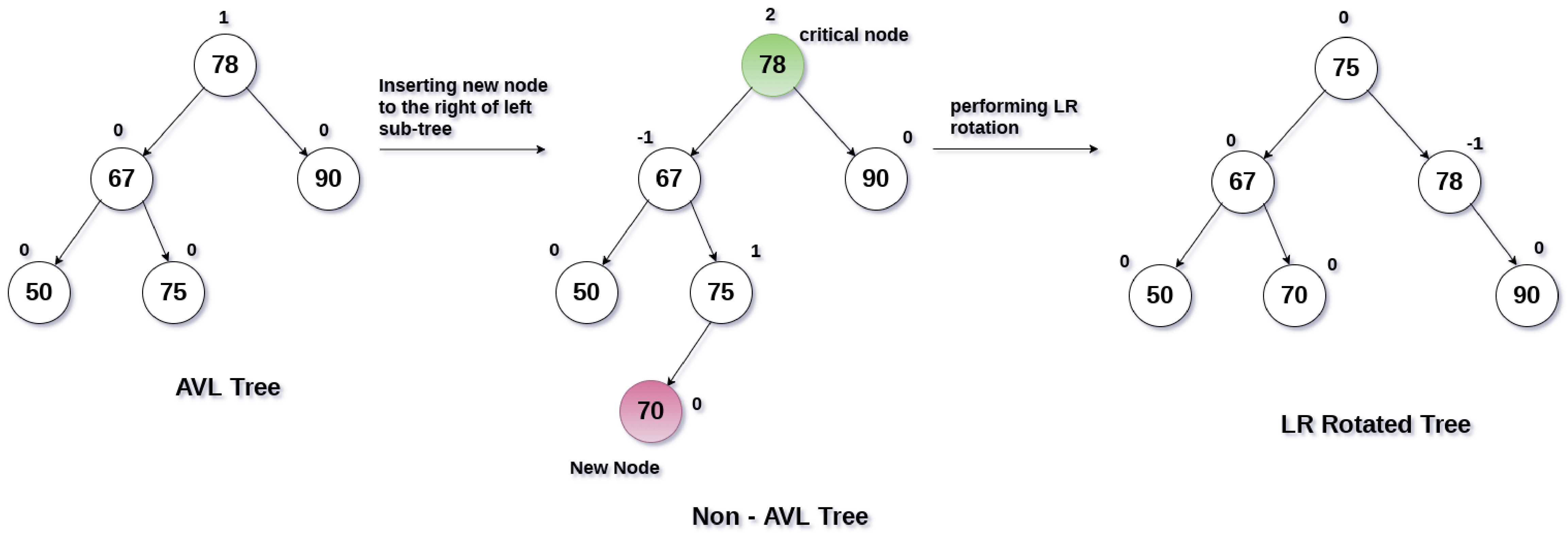
- One rotation suffices to fix cases LL and RR.
- Single rotation preserves the original height:
  - The new height of the entire subtree is exactly the same as the height of the original subtree before the insertion.
- Therefore, it is **enough** to do rotation only at the first node, where imbalance exists, on the path from inserted node to root.
- Thus, the rotation takes O(1) time.
- Hence insertion is O(logN)

➢***LR rotation:*** **the new node is inserted in the right sub-tree of the left child of the critical node**

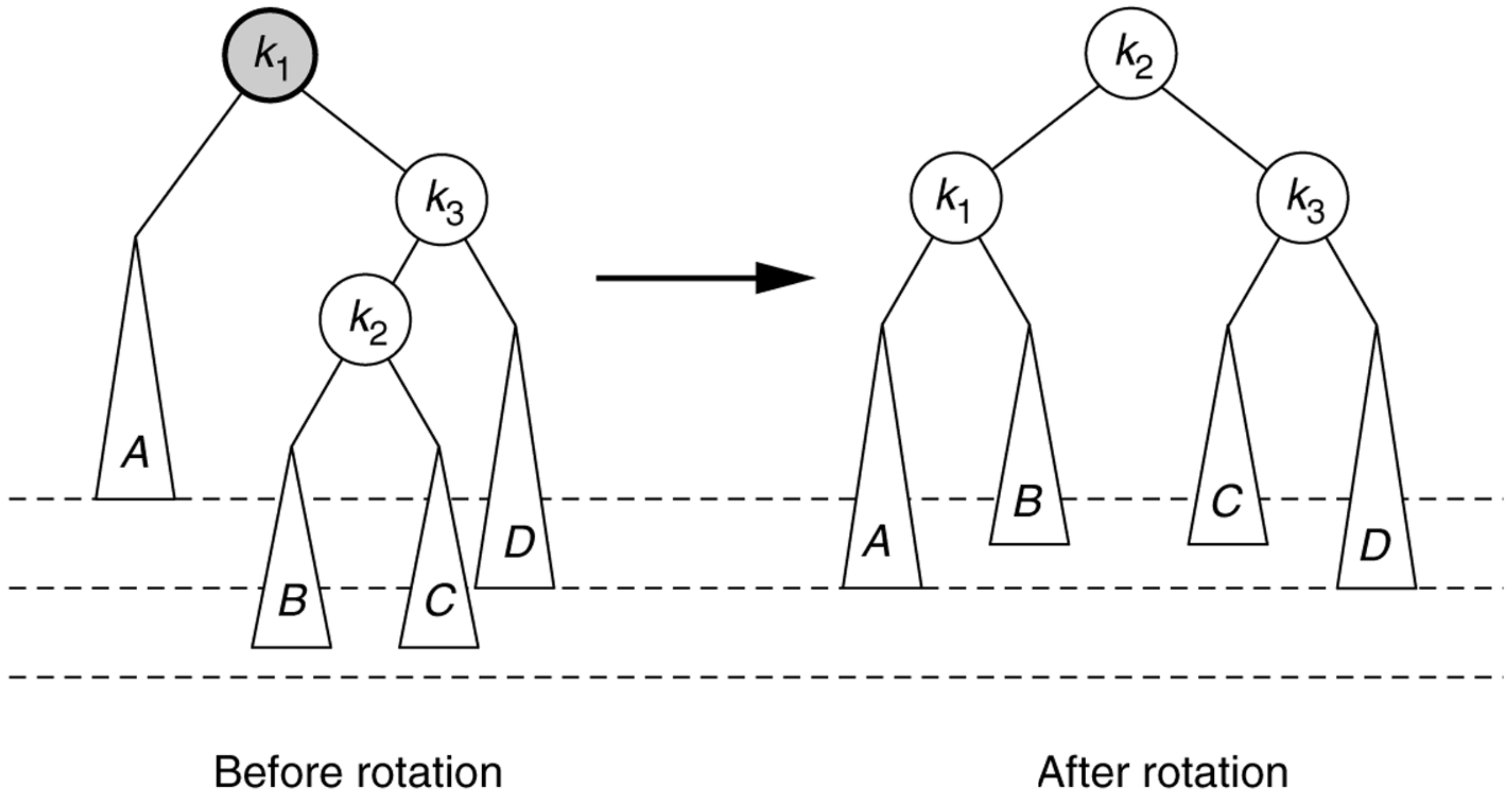
Lift this up:
first rotate left between (k_1,k_2),
then rotate right between (k_3,k_2)



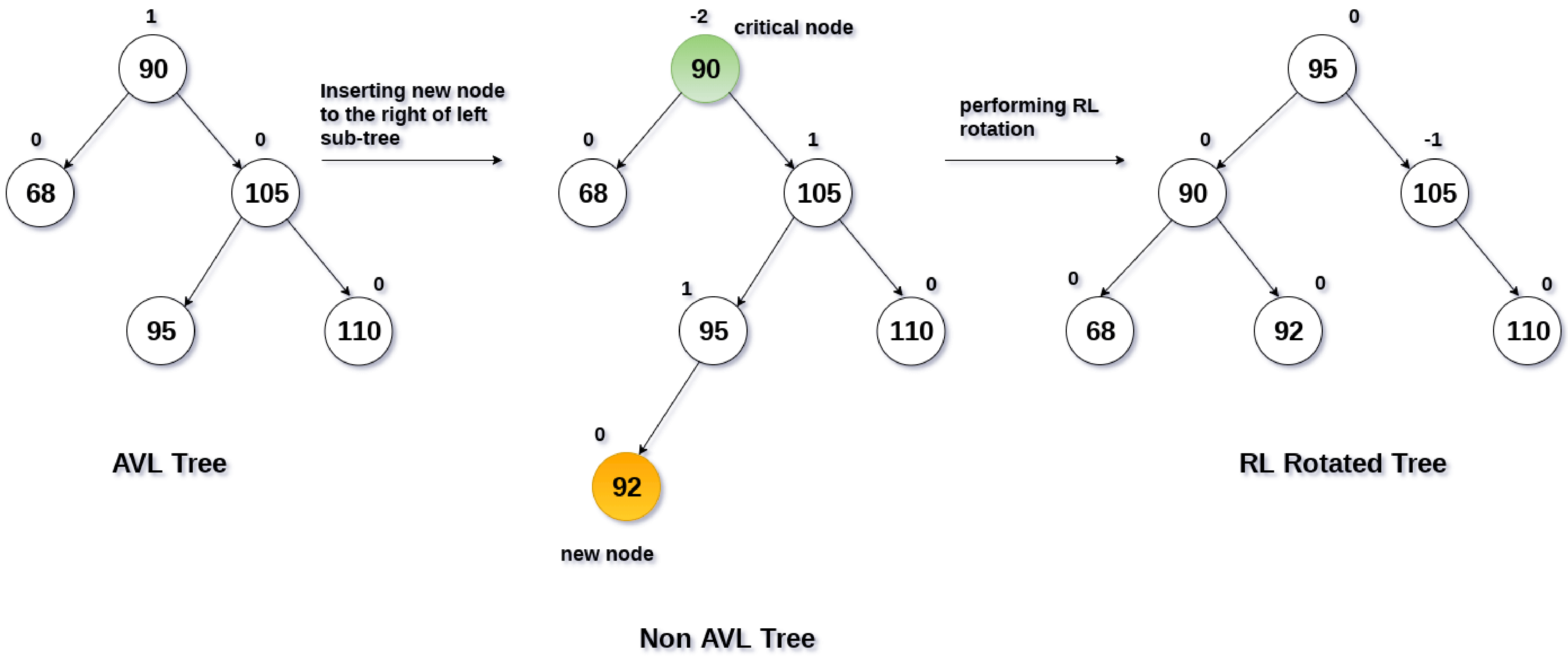
➢*LR rotation:* the new node is inserted in the right sub-tree of the left child of the critical node



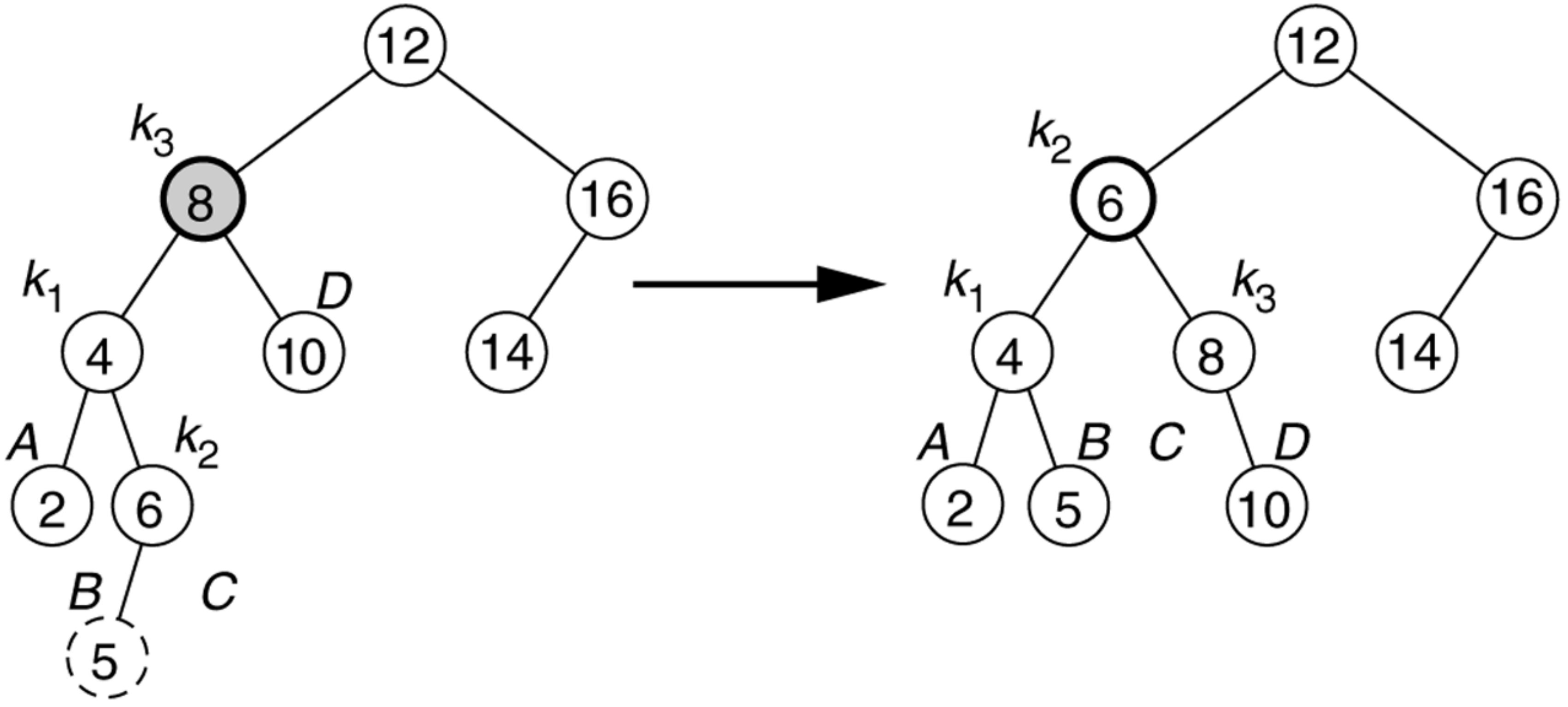
➢ ***RL rotation:*** **the new node is inserted in the left sub-tree of the right child of the critical node**



➢*RL rotation:* the new node is inserted in the left sub-tree of the right child of the critical node



insertion of 5.

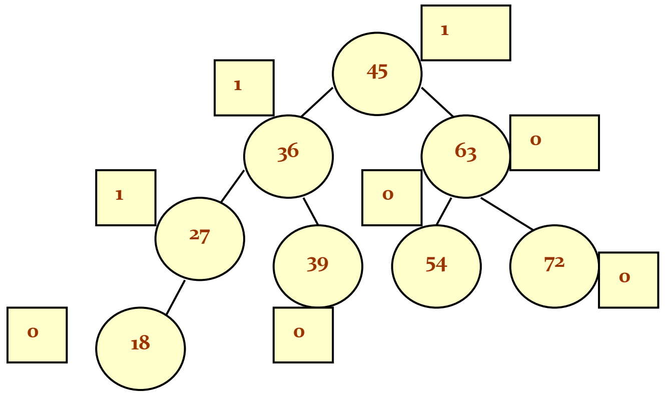


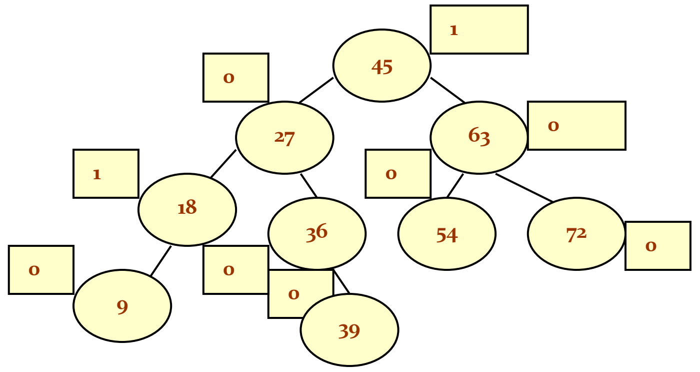
Before rotation

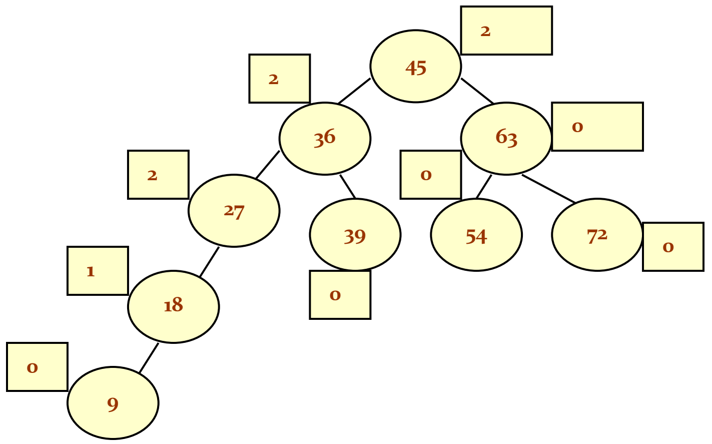
After rotation

Rotations to Balance AVL Trees

Example: Consider the AVL tree given below and insert 9 into it.



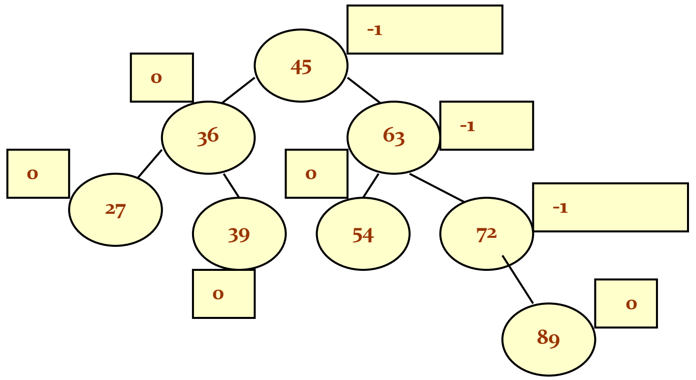


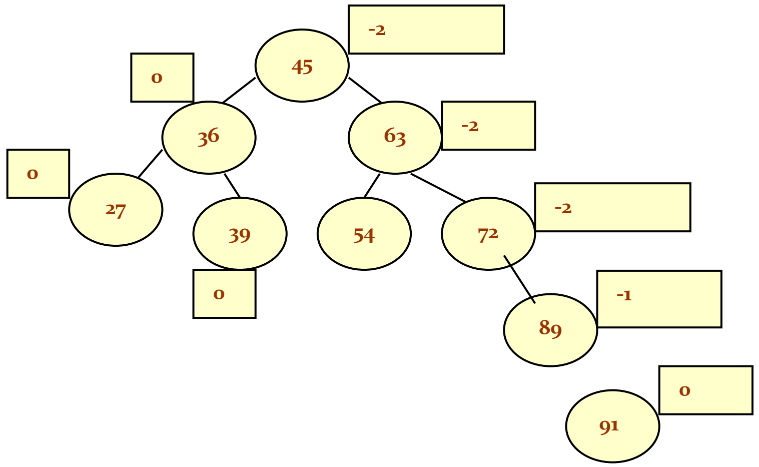


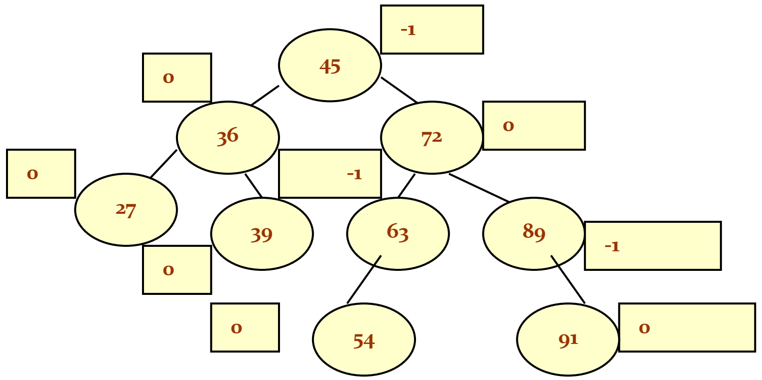
The tree is balanced using LL rotation

Rotations to Balance AVL Trees

Example: Consider the AVL tree given below and insert 91 into it.



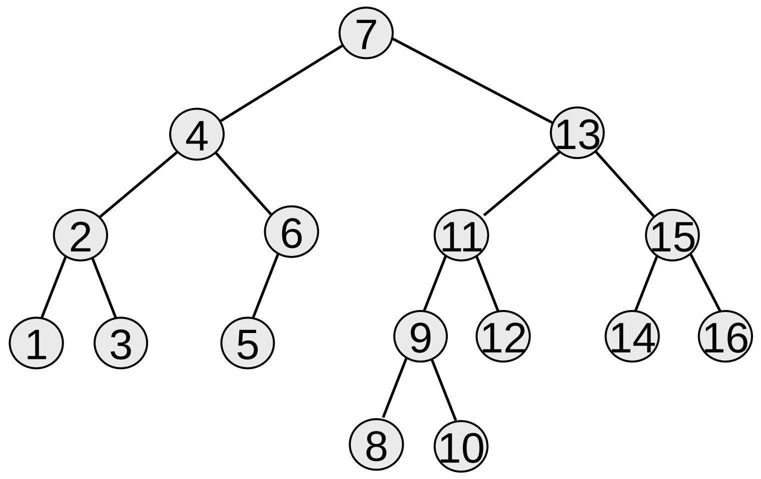




The tree is balanced using RR rotation

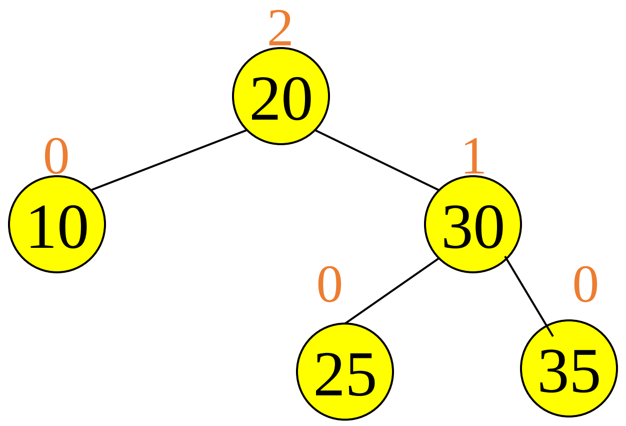
Example

- Insert 16, 15, 14, 13, 12, 11, 10, and 8, and 9 to the previous tree obtained in the previous single rotation example.
- Answer:



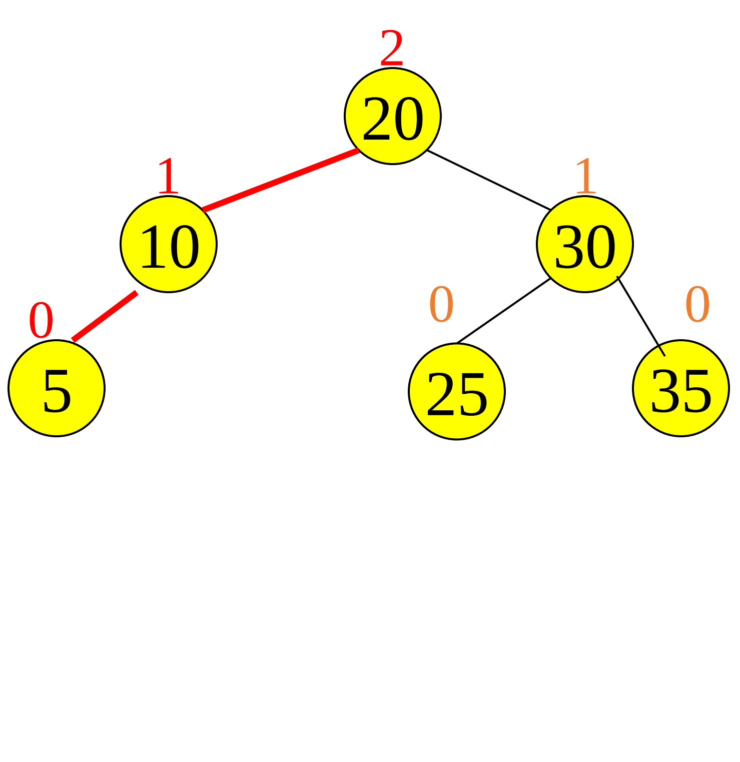
Example

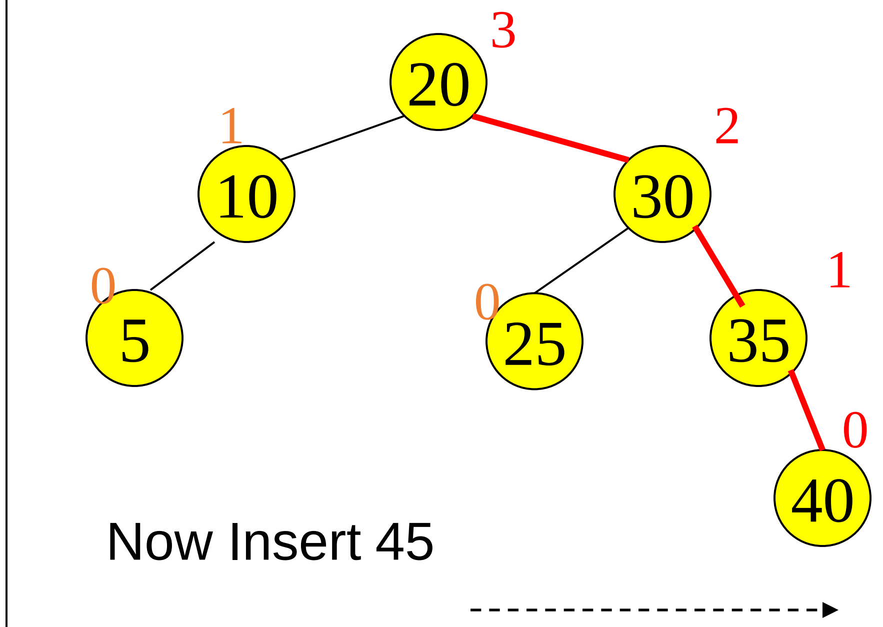
Example of Insertions in an AVL Tree



Insert 5, 40

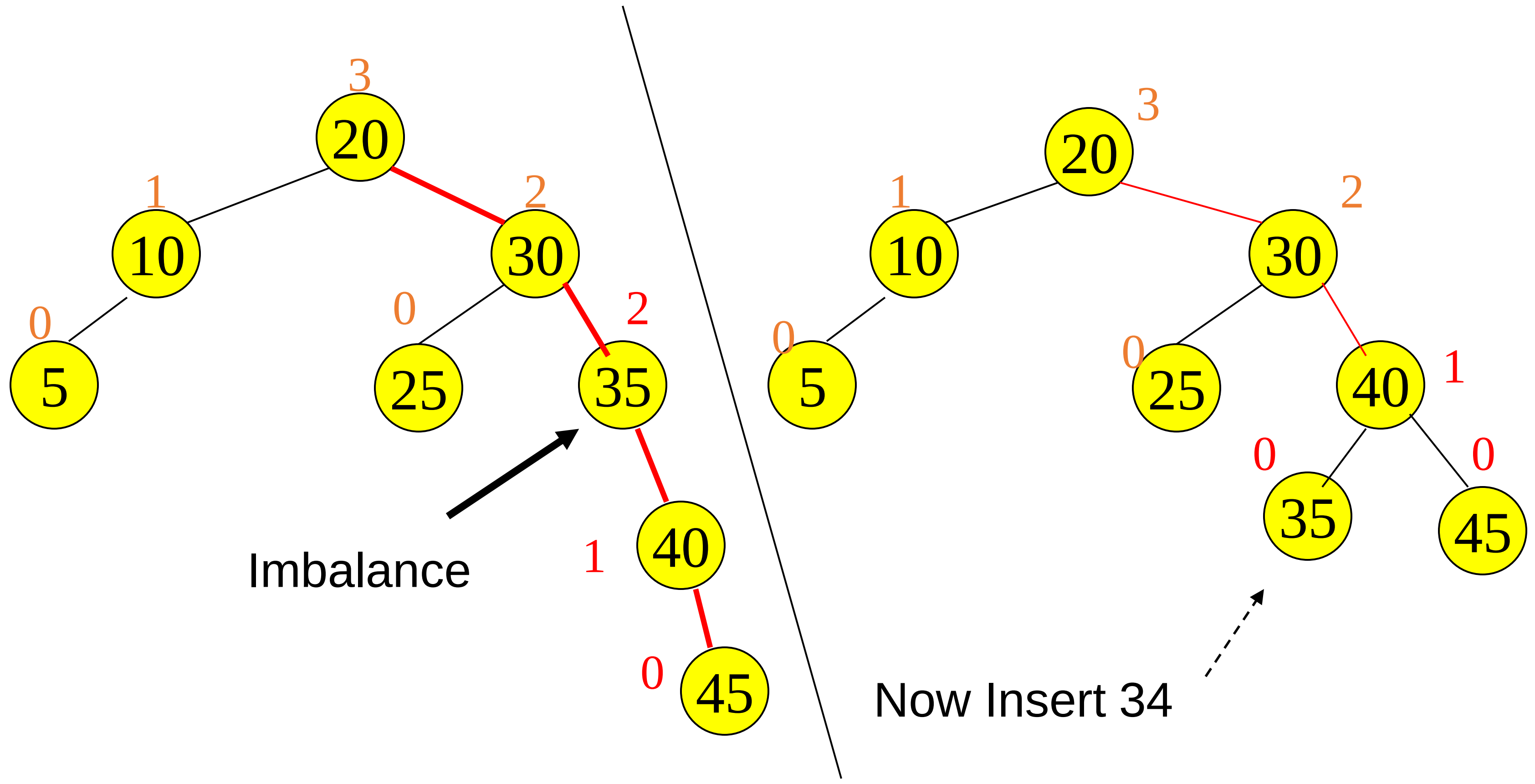
Example of Insertions in an AVL Tree



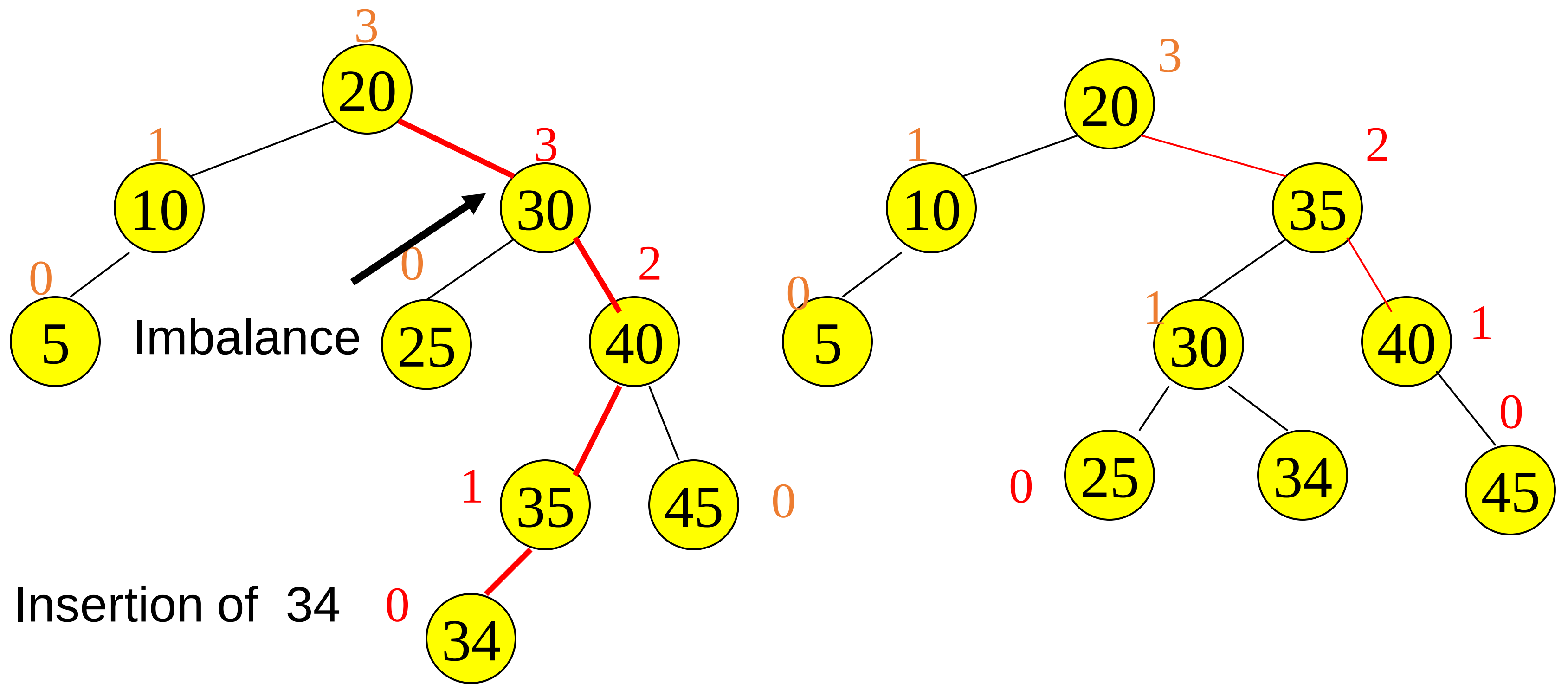


Now Insert 45

Single rotation (outside case)



Double rotation (inside case)



Pros and Cons of AVL Trees

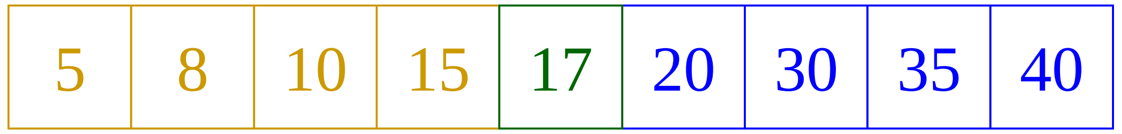
Arguments for AVL trees:

1. Search is O(log N) since AVL trees are always balanced.
2. Insertion and deletions are also O(logn)
3. The height balancing adds no more than a constant factor to the speed of insertion.

Arguments against using AVL trees:

1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).

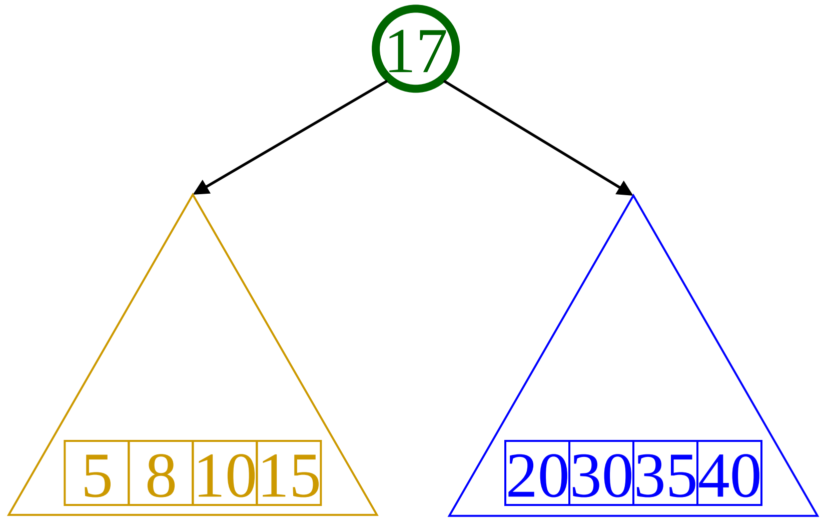
AVL BuildTree



Divide & Conquer

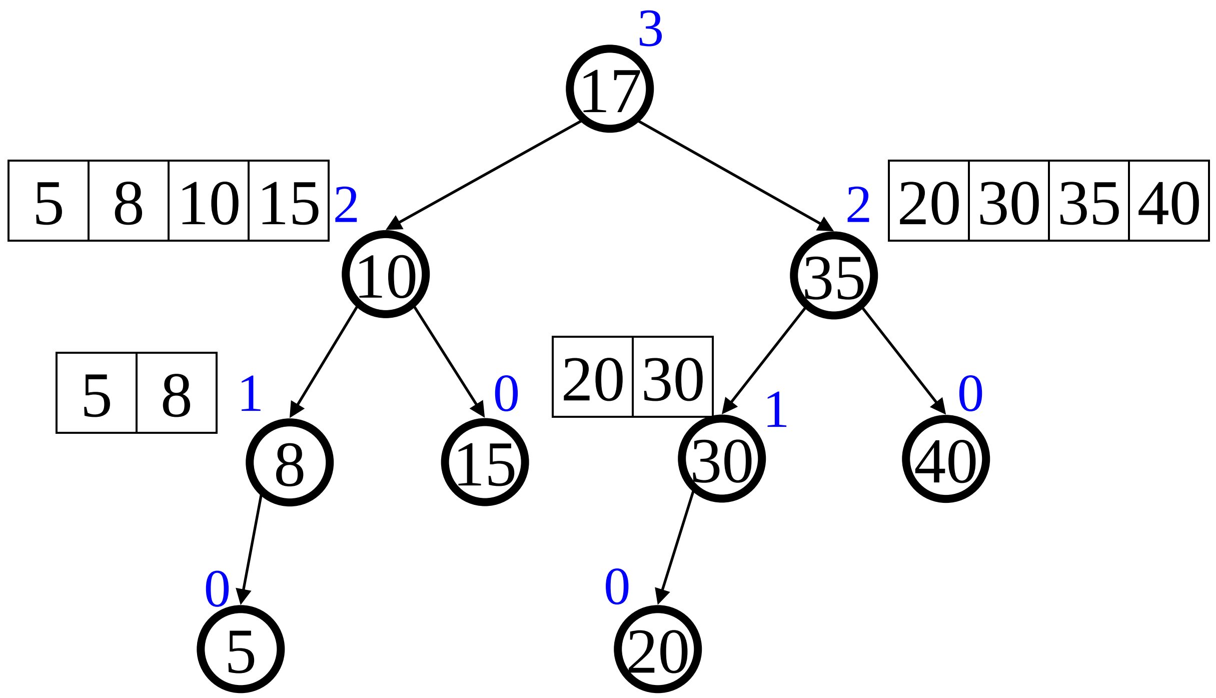
- Divide the problem into parts
- Solve each part recursively
- Merge the parts into a general solution

How long does
divide & conquer take?



BuildTree Example

5	8	10	15	17	20	30	35	40



BuildTree Analysis (Approximate)

```
T(1) = 1
T(n) = 2T(n/2) + 1
T(n) = 2(2T(n/4)+1) + 1
T(n) = 4T(n/4) + 2 + 1
T(n) = 4(2T(n/8)+1) + 2 + 1
T(n) = 8T(n/8) + 4 + 2 + 1
```

$\frac{1}{2}\sum_{i=1}^{k} 2^i$

```
T(n) = 2^kT(n/2^k) +
```

let $2^k = n$, log n = k

$\frac{1}{2}\sum_{i=1}^{\log n} 2^i$

```
T(n) = nT(1) +
T(n) = Θ(n)
```