Data Structure

http://smartlead.hallym.ac.kr

Instructor: Jin Kim
010-6267-8189(033-248-2318)
jinkim@hallym.ac.kr
Office Hours:

Non Linear Data Structure

- Data structure we will consider this semister:
  - Tree
  - Binary Search Tree
  - Graph
  - Weighted Graph
  - Sorting
  - Balanced Search Tree

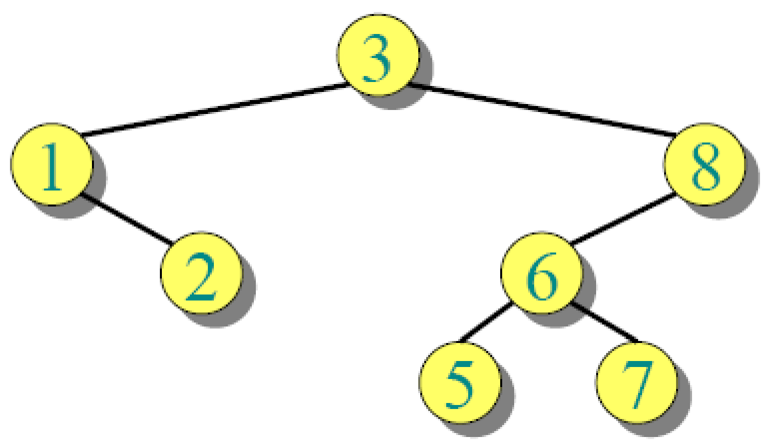
Balanced Search Trees
균형 탐색 트리

Balanced Search Trees

- Binary Search Tree(이진탐색트리)

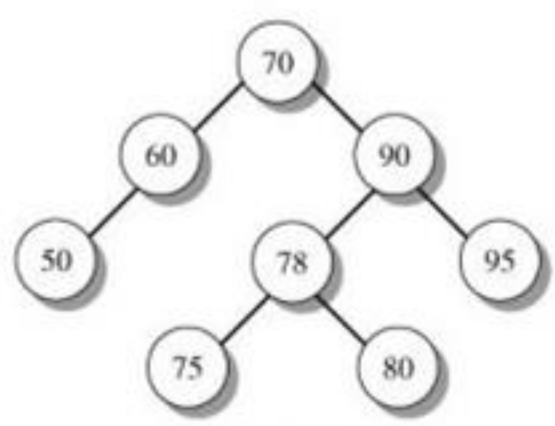
Balanced Search Trees(균형탐색트리)

- **Balanced search tree:** A search-tree data structure for which a height of O(lgn) is guaranteed when implementing a dynamic set of n items. (탐색시 탐색시간 O(lgn) 을 보장)
- Examples:
  - AVL Tree
  - 2-3-4 Tree
  - B Tree
  - Red-black Tree

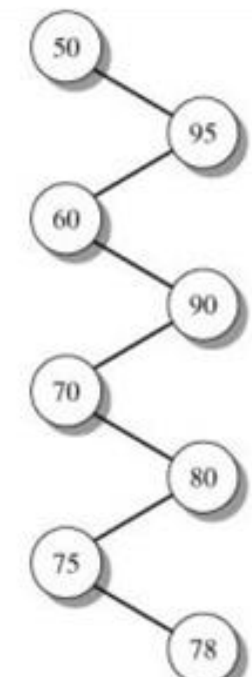


1.What is a Balanced Binary Search Tree?

- A balanced search tree is one where all the branches from the root have almost the same height.(균형탐색트리는 어느 단말에서도 루트까지지 높이가 거의같은 트리)



balanced



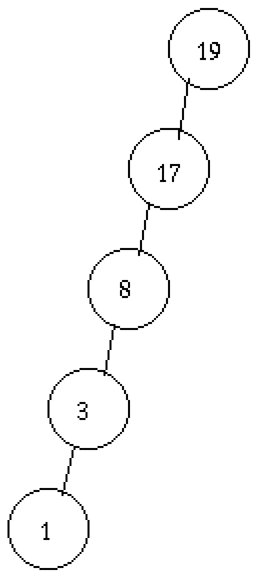
unbalanced

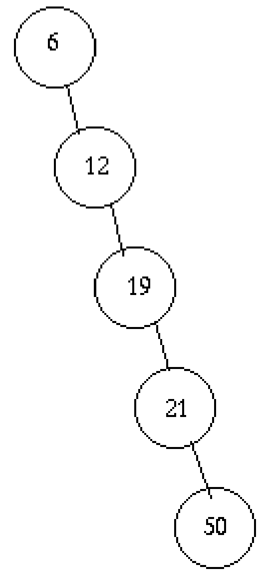
- As a tree becomes more unbalanced, search running time decreases from O(log n) to O(n)(불균형이진탐색트리는 탐색시간이 O(n))
  - because the tree shape turns into a list

- We want to keep the binary search tree balanced as nodes are added/removed, so searching/insertion remain fast.

Unbalanced Search Trees
(불균형탐색트리)

Skewd bst(경사이진트리)



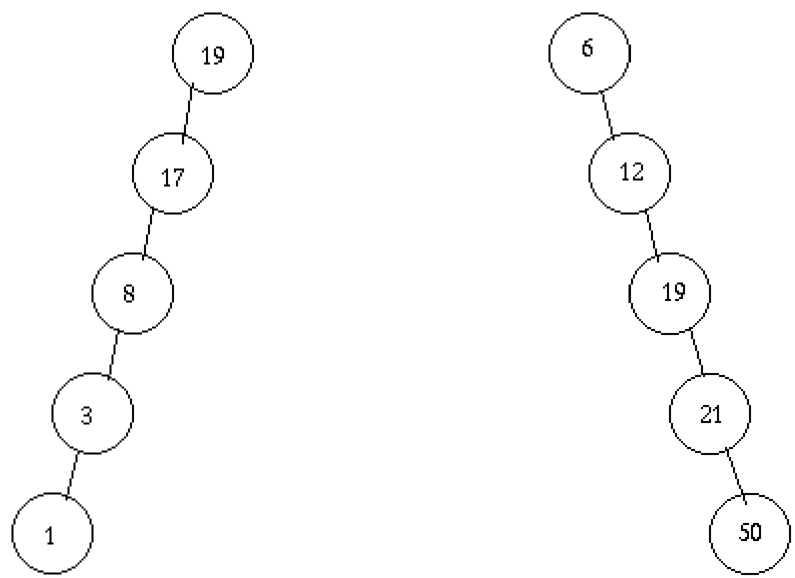


이진 탐색트리

❑ 이진 탐색 트리의 문제점

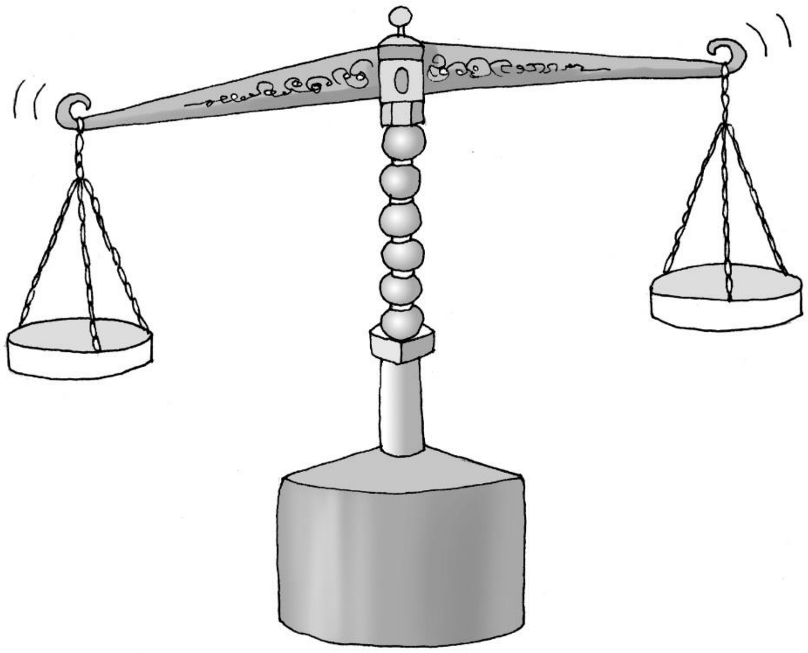
- 이진 탐색 트리에는 새로운 노드들이 무작위로 삽입/삭제가 됨
- 이때, 다음 그림과 같이 탐색 트리가 한 방향으로 기울어질 수 있음
- 이진 탐색 트리가 한 방향으로 기울어지면 비교횟수가 평균 n/2회로 증가하여 선형 탐색의 경우처럼 됨

➔ 이러한 문제를 해결하기 위해 균형 탐색트리(balanced search tree)가 사용



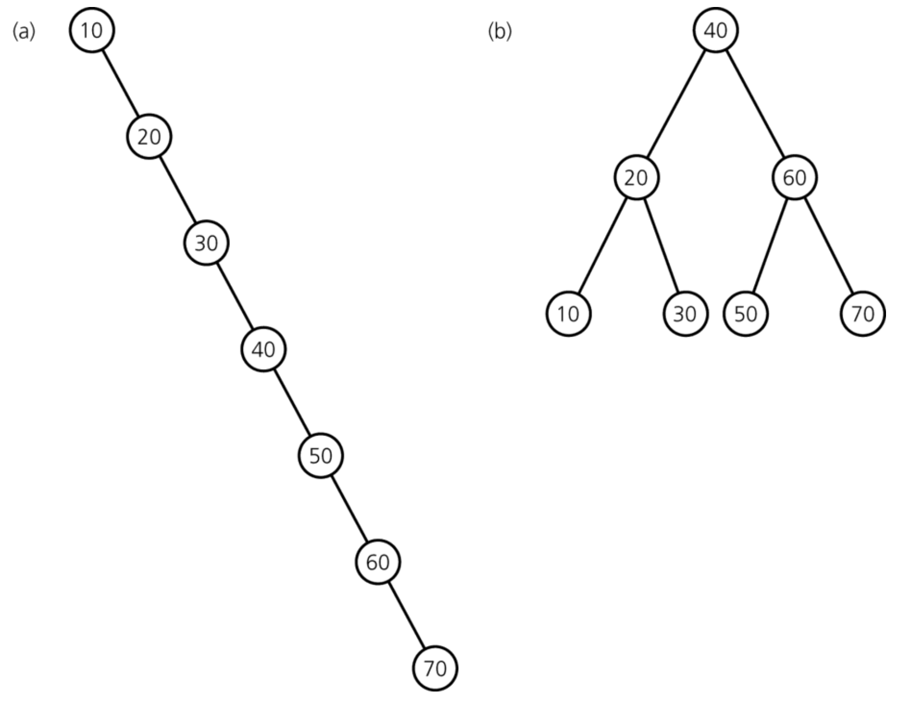
Balanced Search Trees

- Balanced(균형잡힌)



Why care about advanced implementations?

Same entries, different insertion sequence
(같은데이터, 다른 입력순서)



(a) Skewd bst 불균형 (b) complete bst

→ Not good! Would like to keep tree balanced.

순서

1 AVL 트리
2 스플레이 트리
3 2-3 트리
4 2-3-4 트리
5 레드-블랙 트리

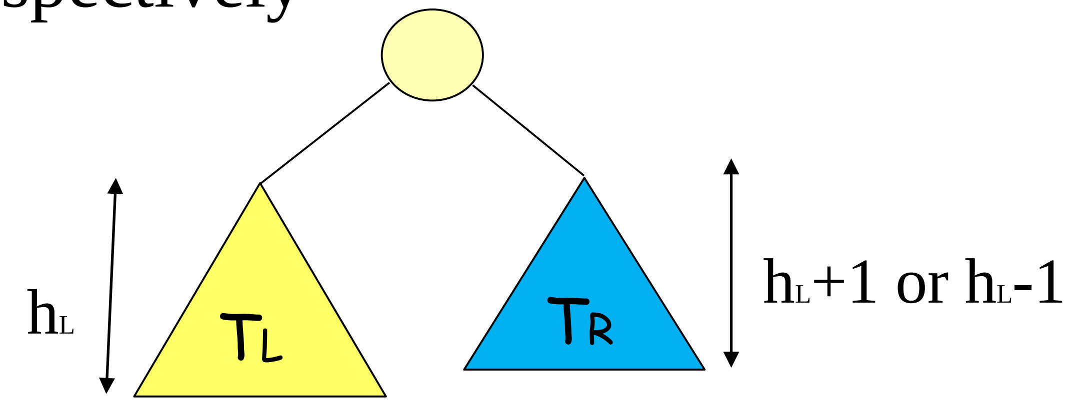
AVL Trees
(AVL 트리)

AVL Trees

- First-invented self-balancing binary search tree (최초의 균형탐색트리 시도)
- Named after its two inventors,
  1. G.M. Adelson-Velsky and
  2. E.M. Landis,
  - published it in their 1962 paper "An algorithm for the organization of information."

AVL Trees: Formal Definition

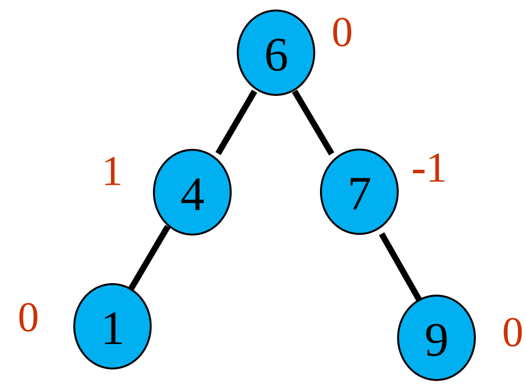
1. All empty trees are AVL-trees
2. If T is a non-empty binary search tree with T_L and T_R as its left and right sub-trees, then T is an AVL tree iff
   1. T_L and T_R are AVL trees
   2. $|h_L – h_R| <= 1$, where h_L and h_R are the heights of T_L and T_R respectively



AVL Trees

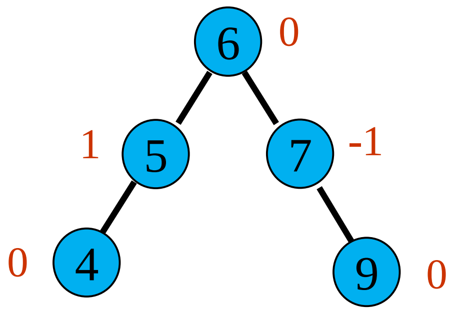
- AVL trees are height-balanced binary search trees
- Balance factor(균형인수) of a node = height(left subtree) - height(right subtree)
- An AVL tree can only have balance factors of –1, 0, or 1 at every node
- For every node, heights of left and right subtree differ by no more than 1

An AVL Tree

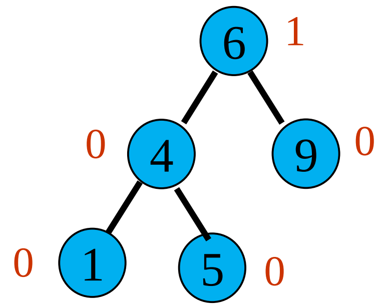


Red numbers
are Balance Factors

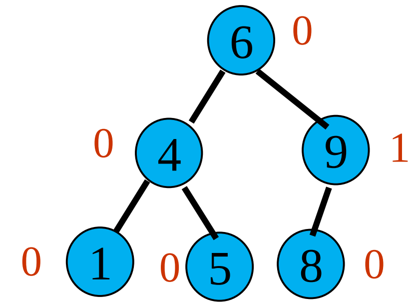
AVL Trees: Examples and Non-Examples



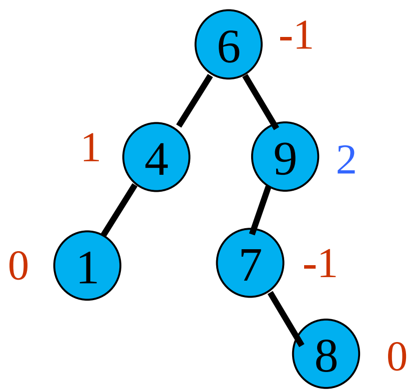
An AVL Tree



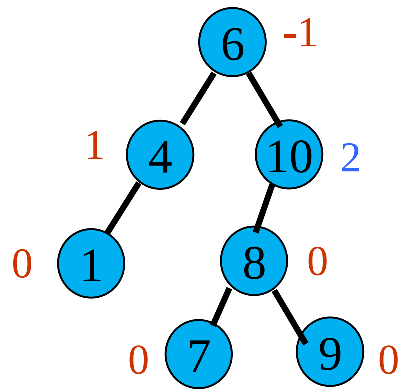
An AVL Tree



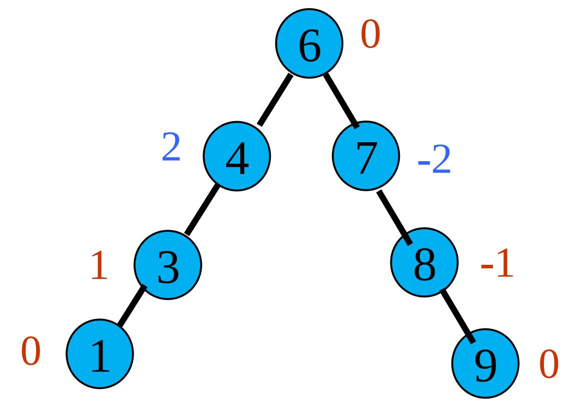
An AVL Tree



Non-AVL Tree



Non-AVL Tree

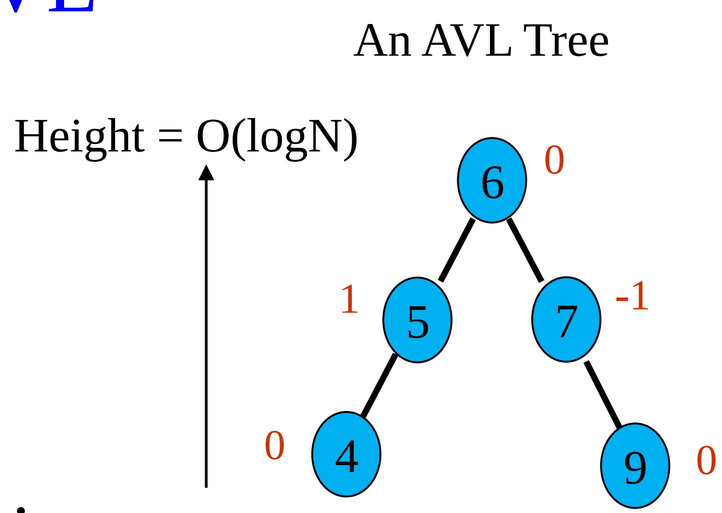


Non-AVL Tree

Red numbers are Balance Factors

Good News about AVL Trees

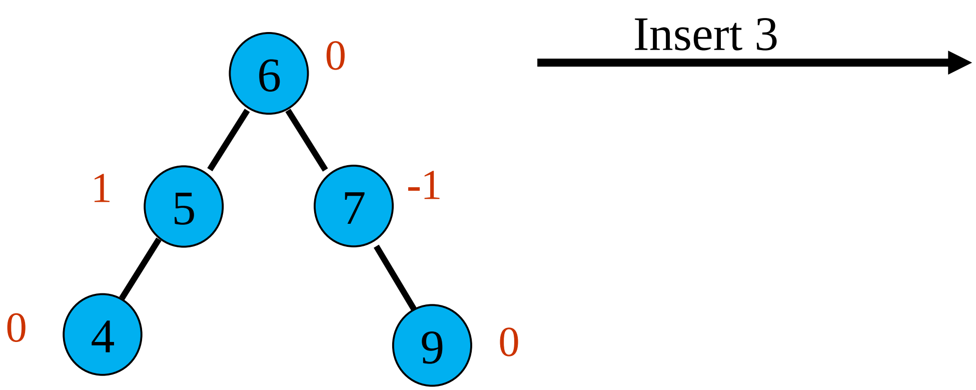
- Can prove: Height of an AVL tree of N nodes is always O(log N) (높이는 항상)
- How? Can show:
  - Height h = 1.44 log(N)
  - Prove using recurrence relation for minimum number of nodes S(h) in an AVL tree of height h:
    - S(h) = S(h-1) + S(h-2) + 1
  - Use Fibonacci numbers to get bound on S(h) bound on height h



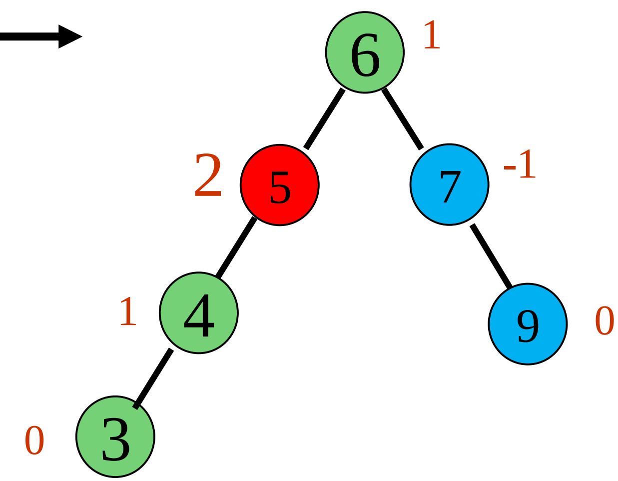
Red numbers
are Balance Factors

Good and Bad News about AVL Trees

- Good News:
  - Search takes O(h) = O(logN)
- Bad News
  - Insert and Delete may cause the tree to be unbalanced!



An AVL Tree

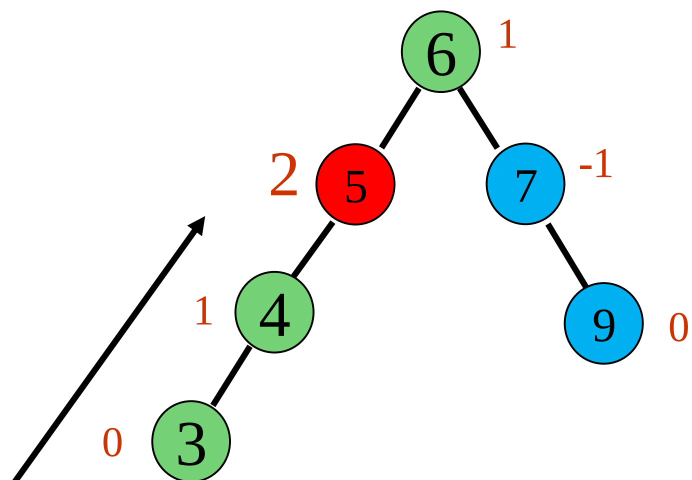


No longer an AVL Tree

Restoring Balance in an AVL Tree

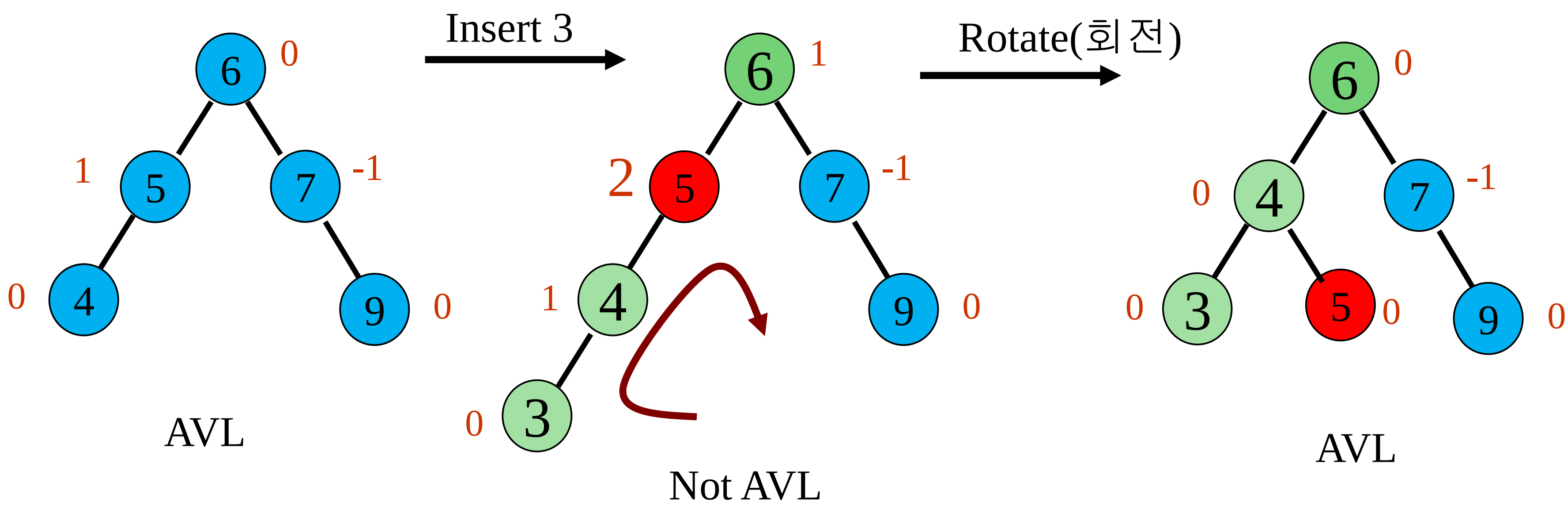
- Problem: Insert may cause balance factor to become 2 or –2 for some node on the path from root to insertion point(AVL트리에 원소삽입하면 AVL트리가 아니게 될 수 있음)

- Idea: After Inserting the new node
  1. Back up towards root updating balance factors along the access path
  2. If Balance Factor of a node = 2 or –2, adjust the tree by rotation around deepest such node.



Non- AVL Tree

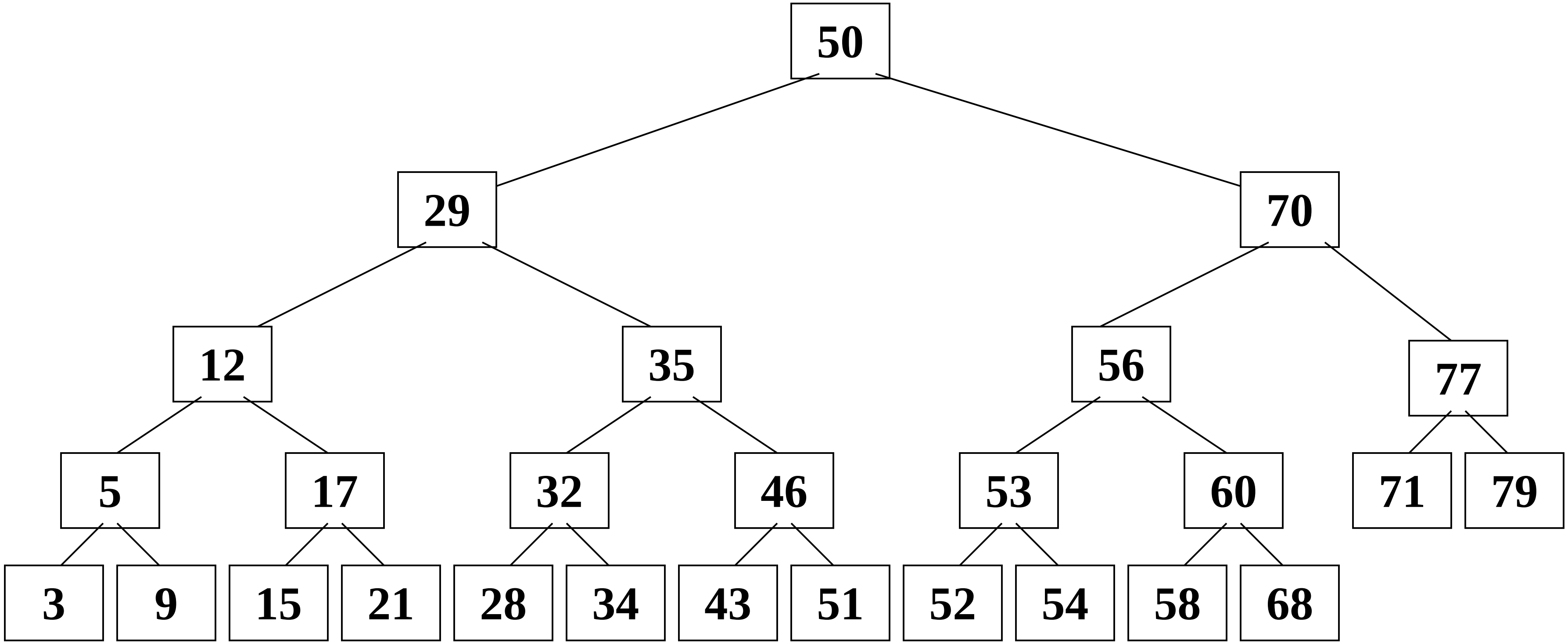
Restoring Balance: Example



- After Inserting the new node
  1. Back up towards root updating heights along the access path
  2. If Balance Factor of a node = 2 or –2, adjust the tree by rotation around deepest such node.

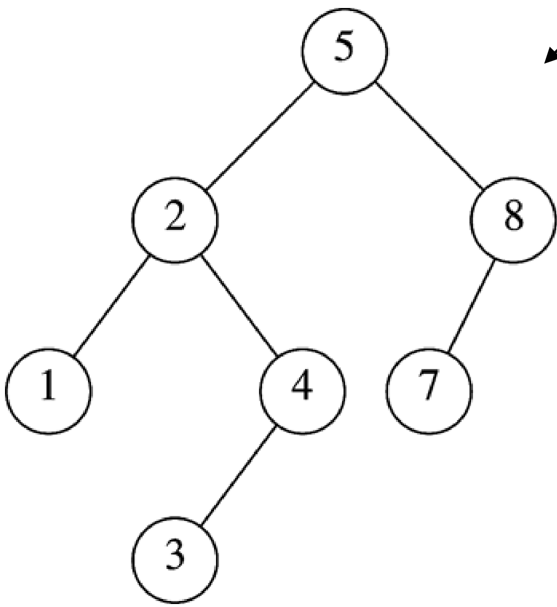
Question?

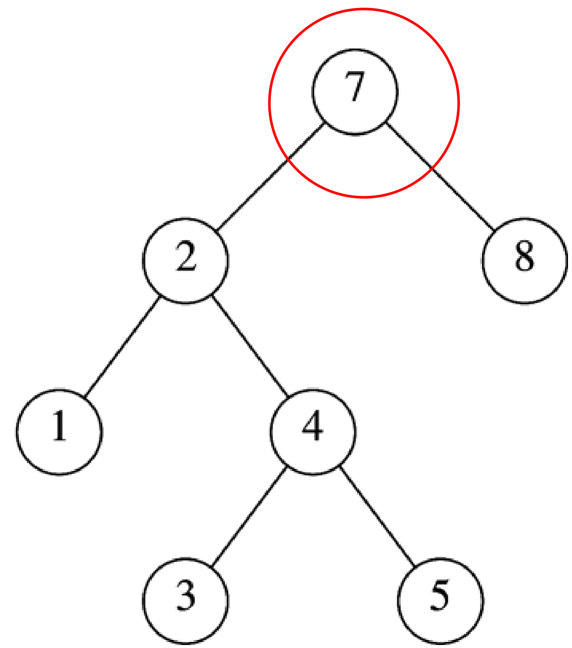
- Is this an AVL Tree?(yes!)



Which is an AVL Tree?

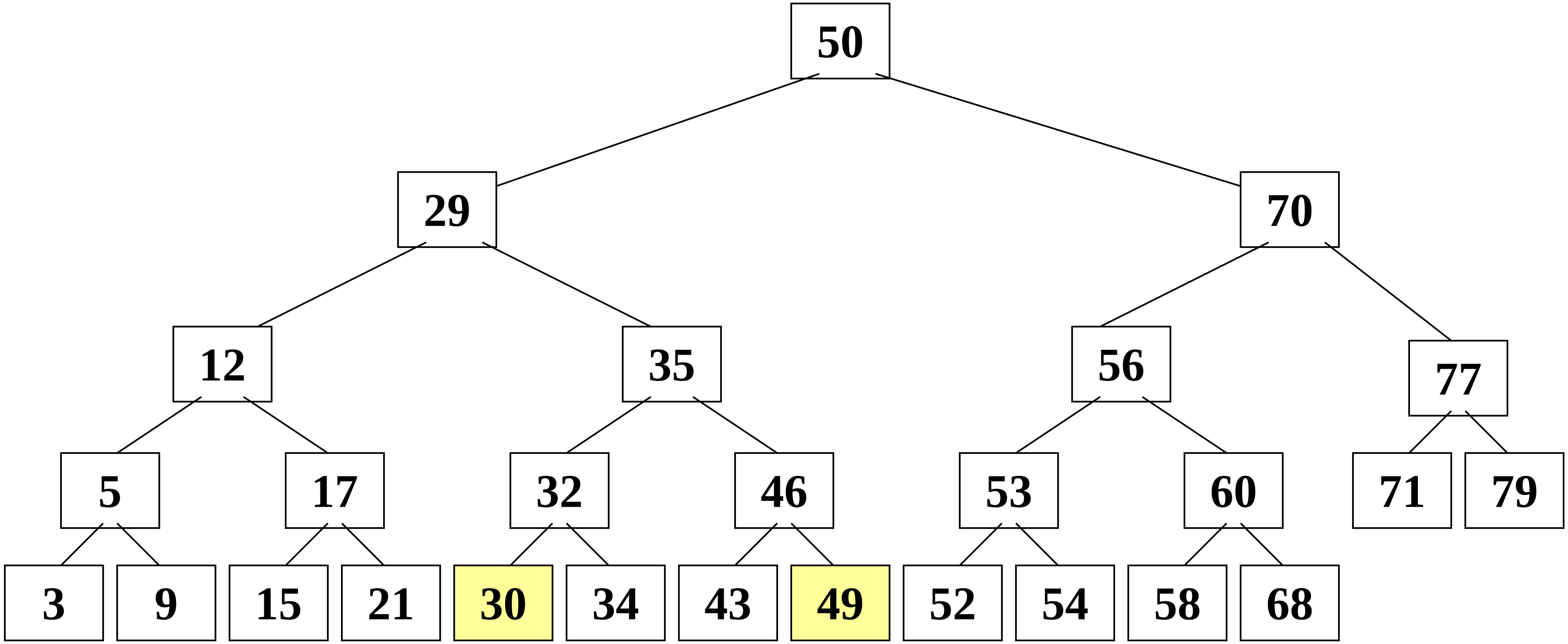
AVL tree





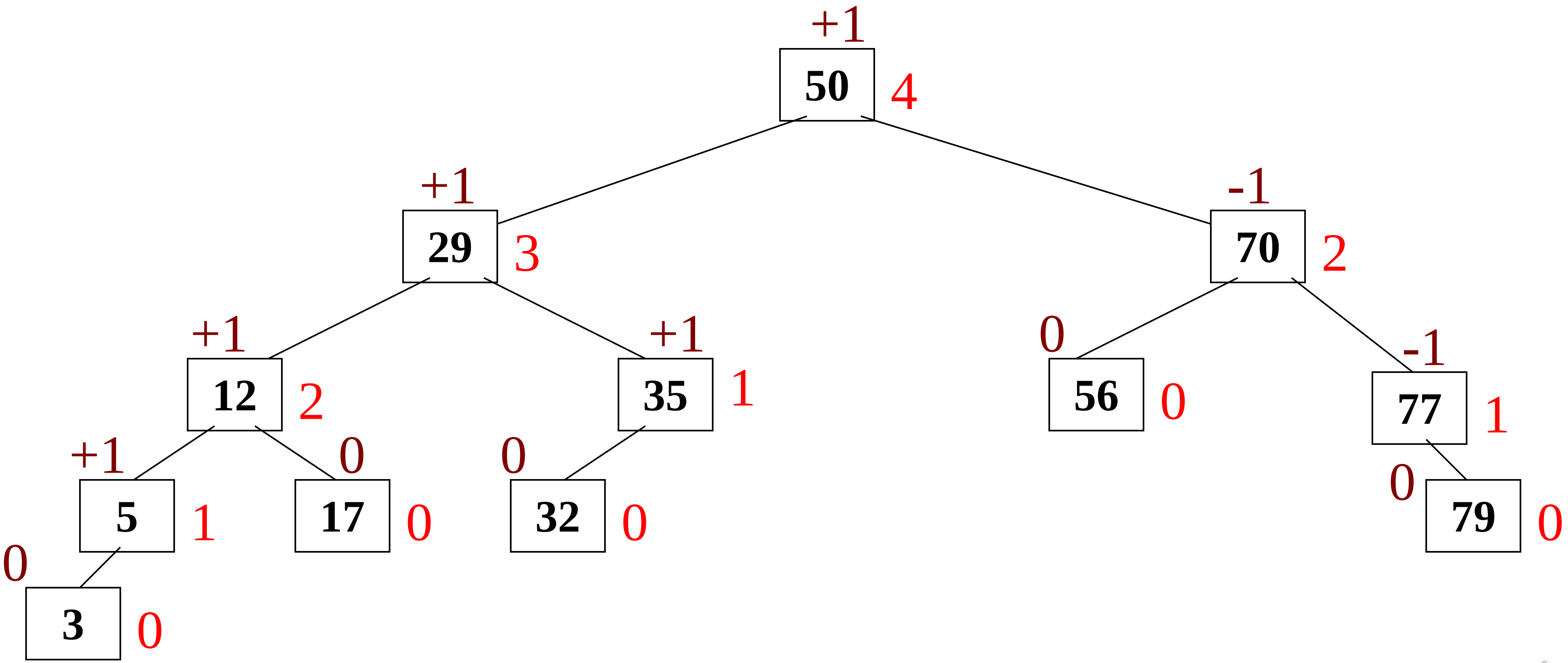
Question?

- Is this an AVL Tree?



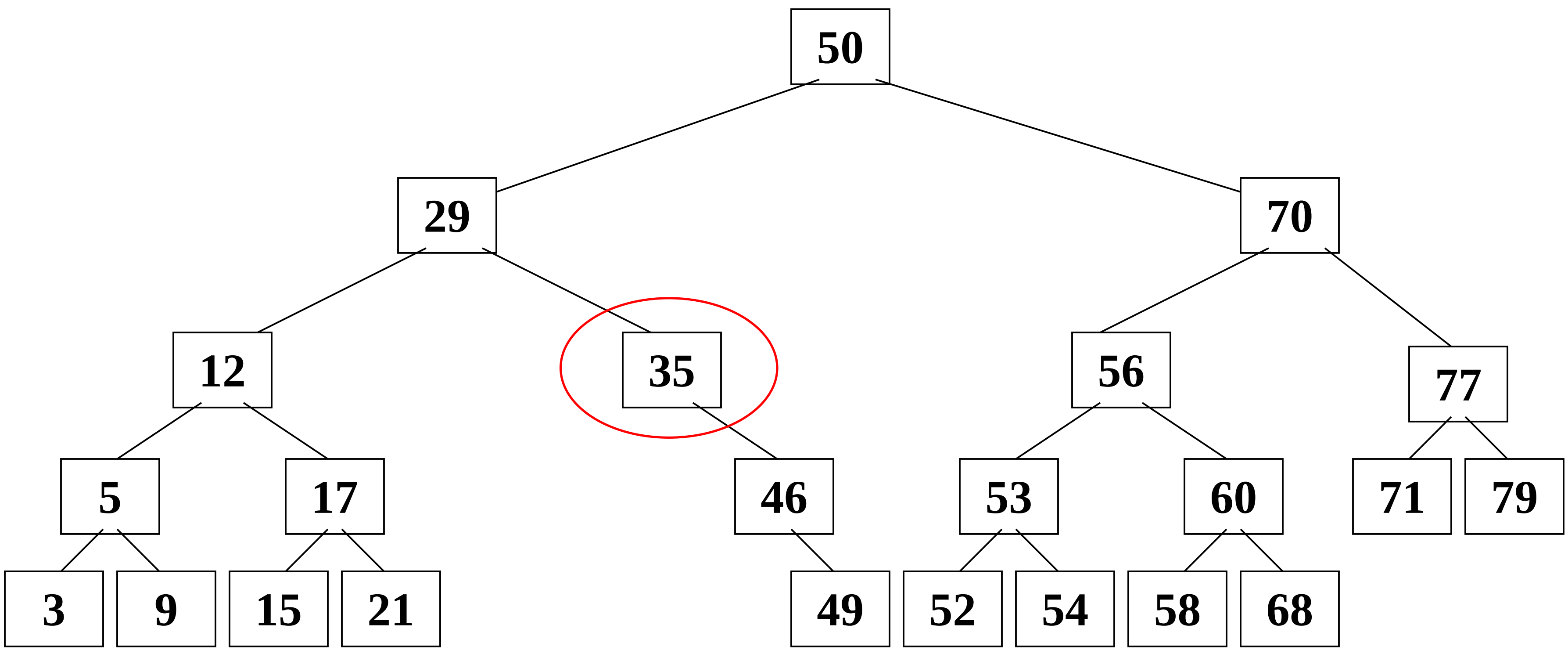
Question?

- Is this an AVL Tree?(yes!)



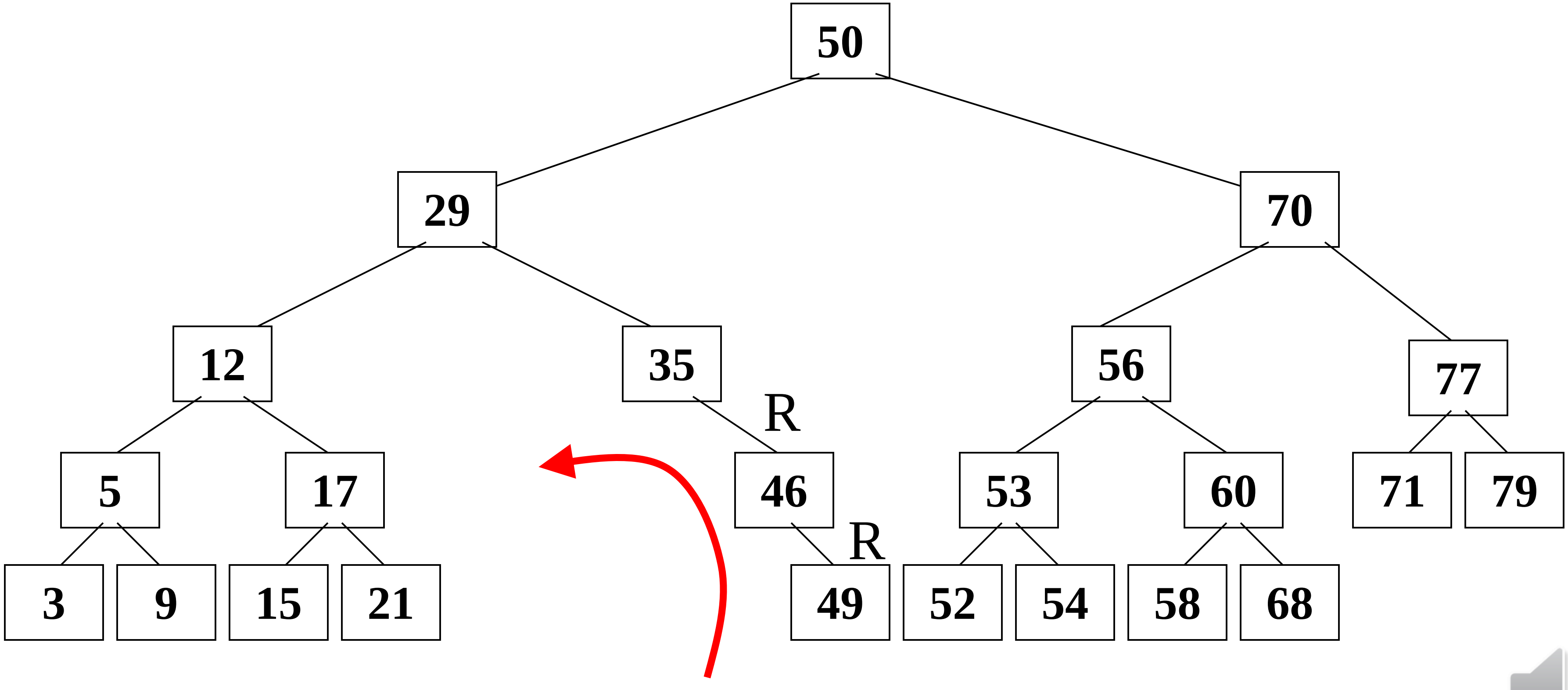
Question?

- Is this an AVL Tree?(No!)



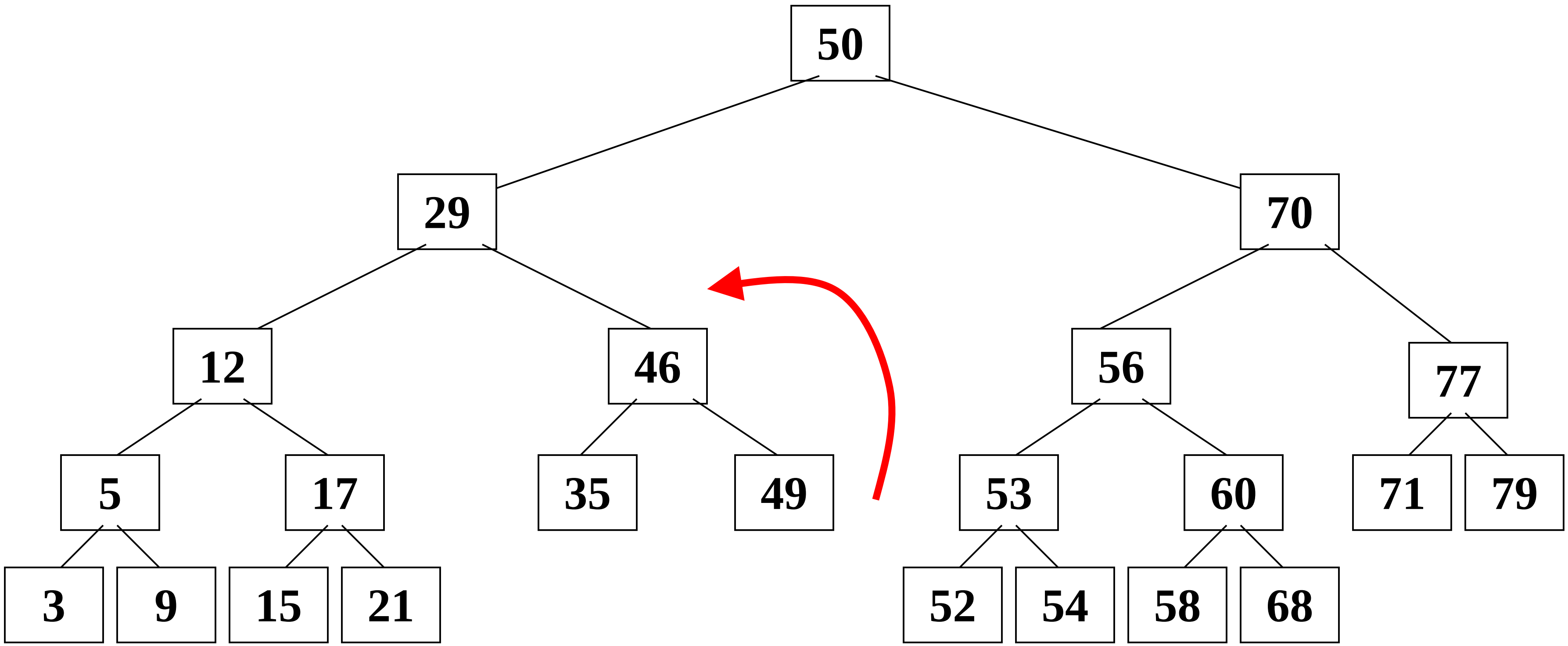
Question?

- No



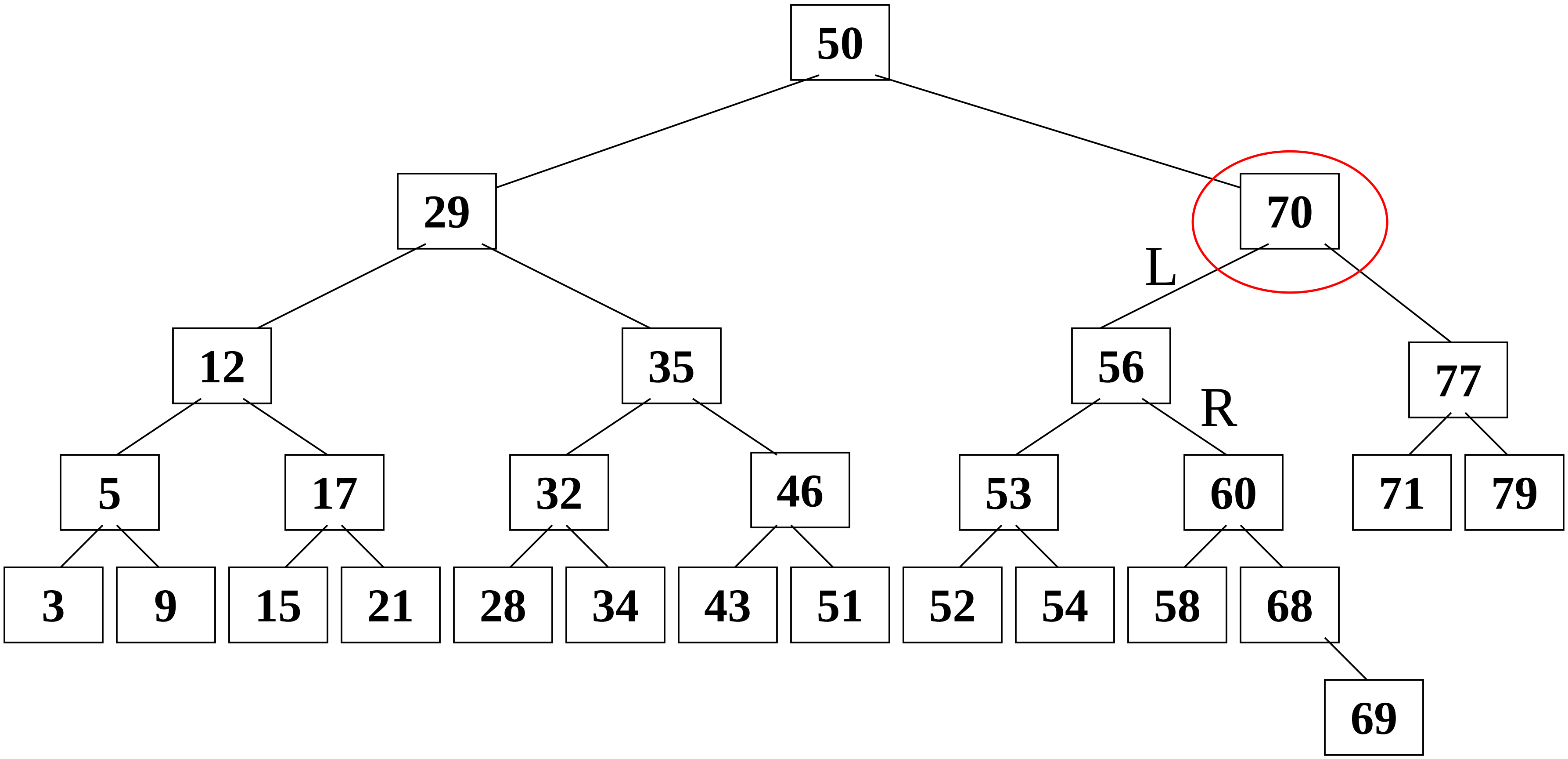
Question?

- Did this fix the problem?



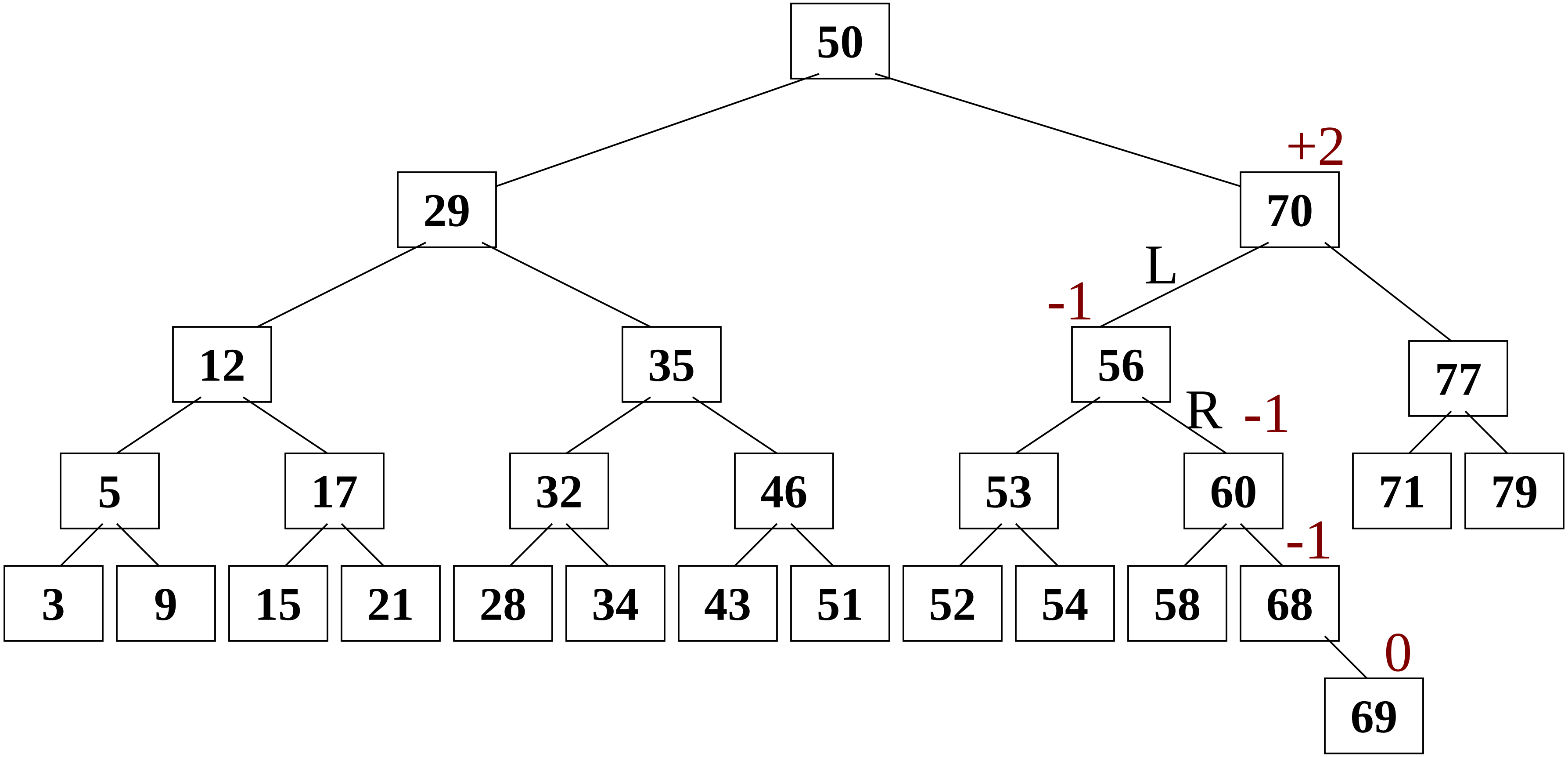
Question?

- Is this an AVL Tree?(No!)



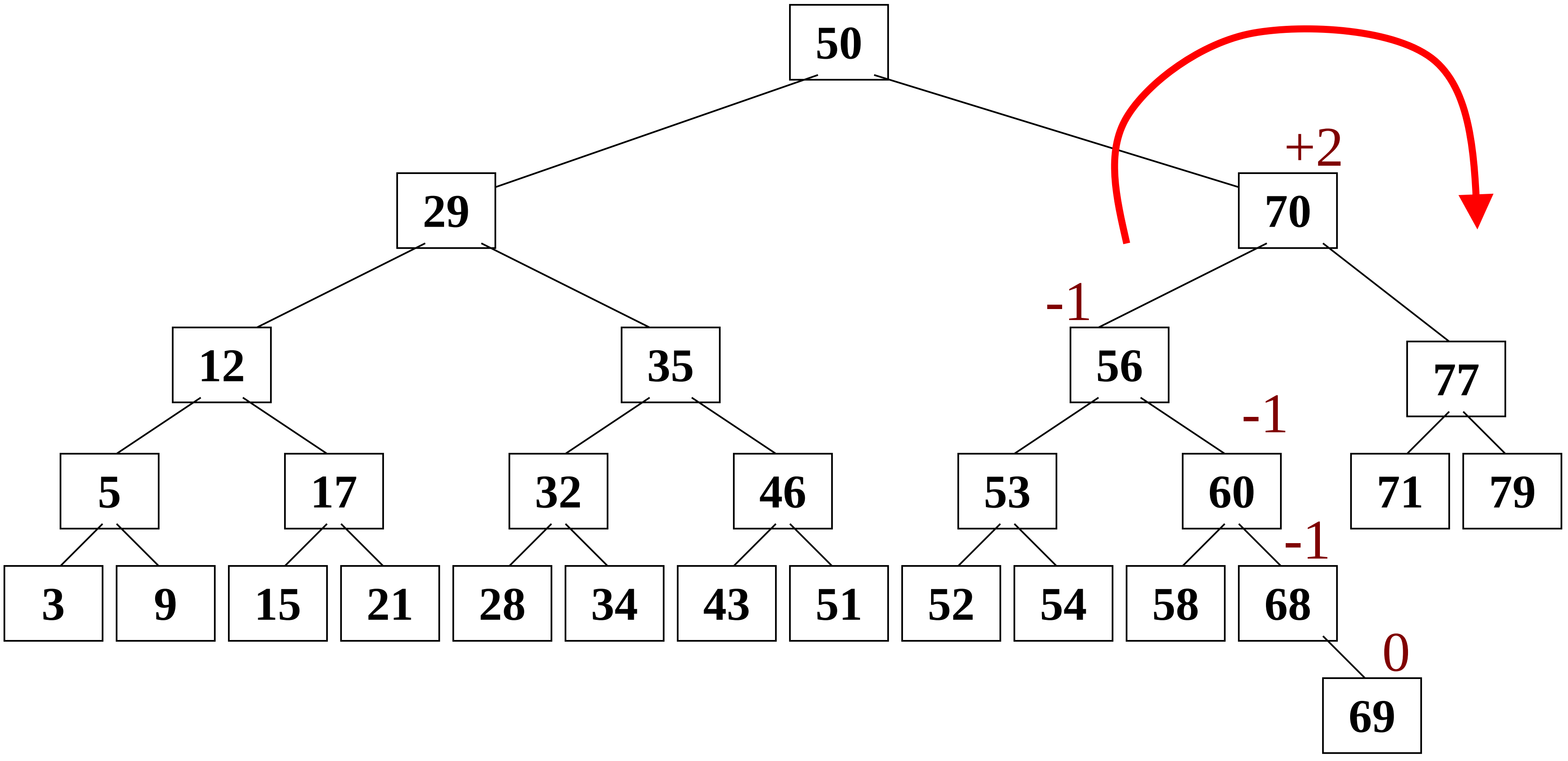
Question?

- Is this an AVL Tree?



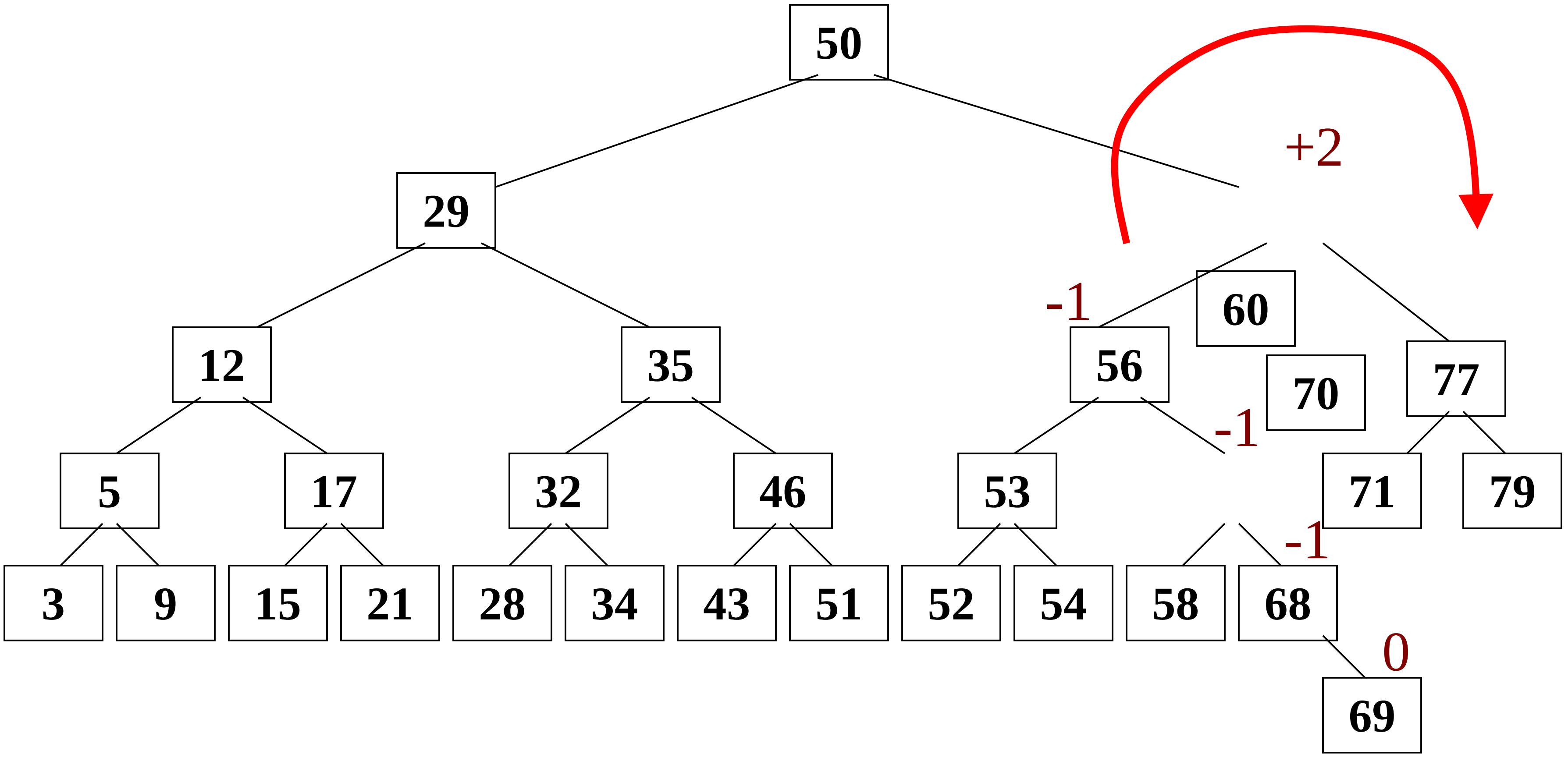
Question?

- Is this an AVL Tree?



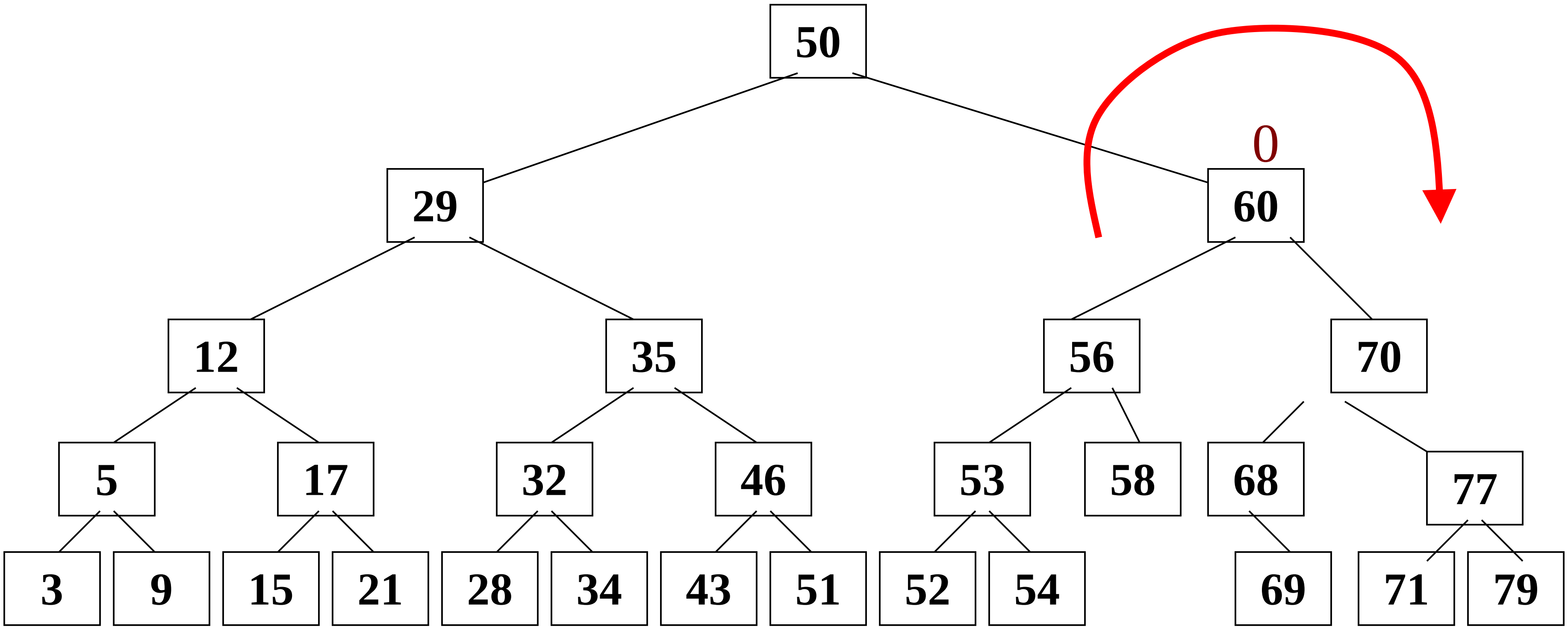
Question?

- Is this an AVL Tree?



Question?

- Is this an AVL Tree?



Correcting Imbalance(불균형 해소)

1. After every insertion
2. Check to see if an imbalance was created.
   - All you have to do backtrack(단말에서루트로 이동함) up the tree
3. If you find an imbalance, correct it.
4. As long as the original tree is an AVL tree, there are only 4 types of imbalances that can occur.

Insertions in AVL Trees

Let the node that needs rebalancing be α.

There are 4 cases:

Outside Cases (require single rotation) :

1. Insertion into left subtree of left child of α. (LL)
2. Insertion into right subtree of right child of α.(RR)

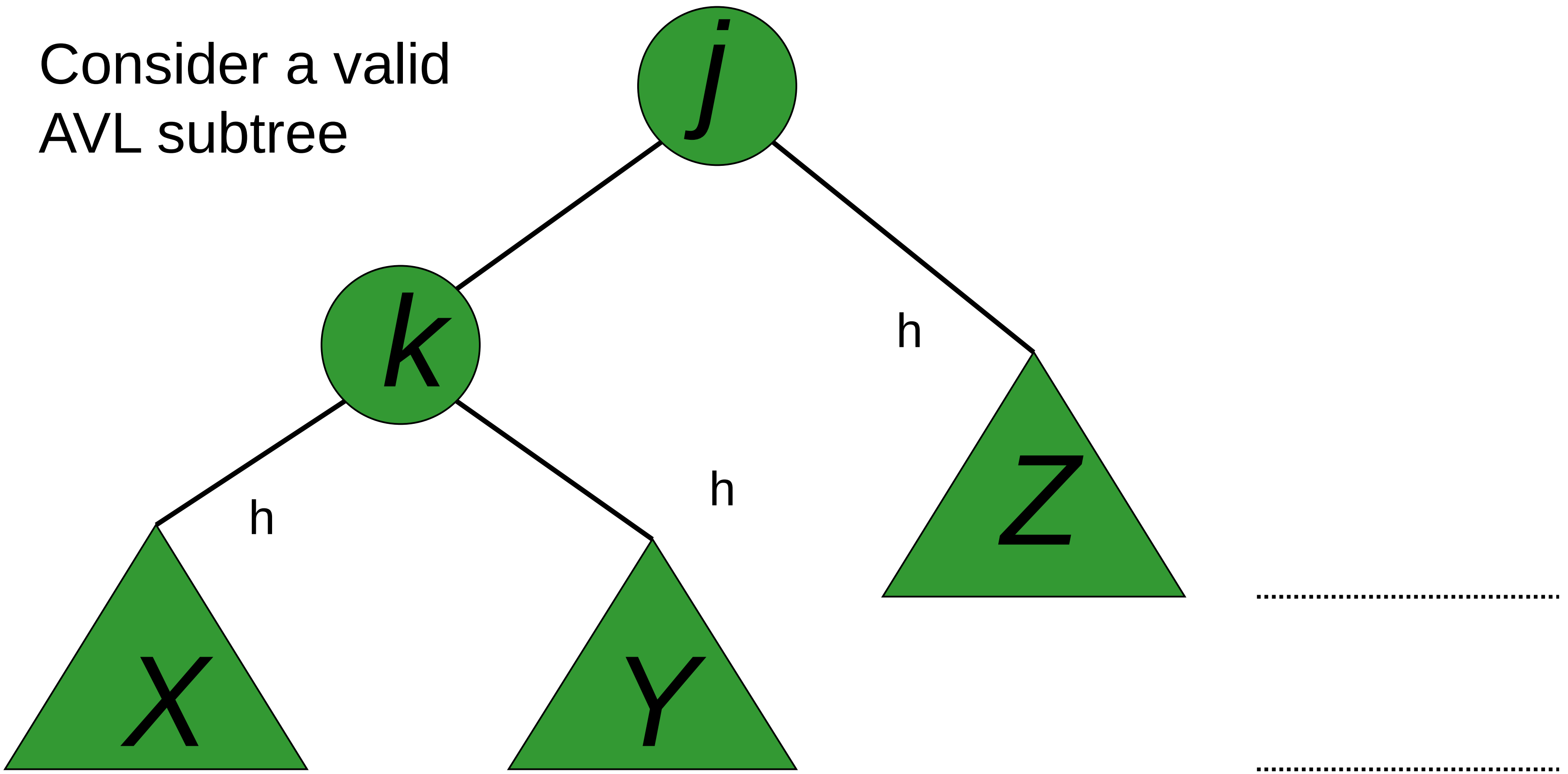
Inside Cases (require double rotation) :

3. Insertion into right subtree of left child of α.(RL)
4. Insertion into left subtree of right child of α.(LR)

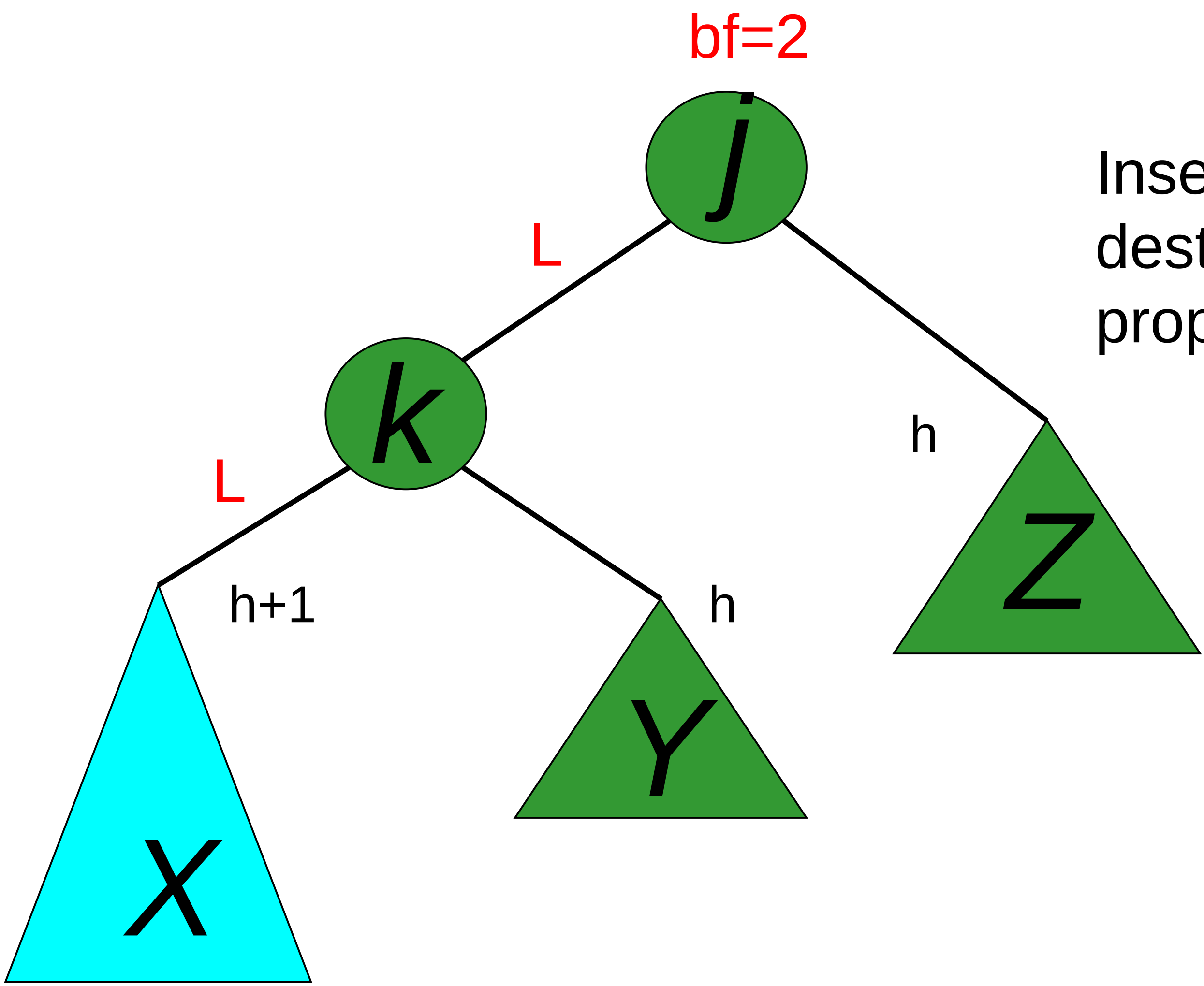
The rebalancing is performed through four separate rotation algorithms.

AVL Insertion: Left-Left

Consider a valid
AVL subtree

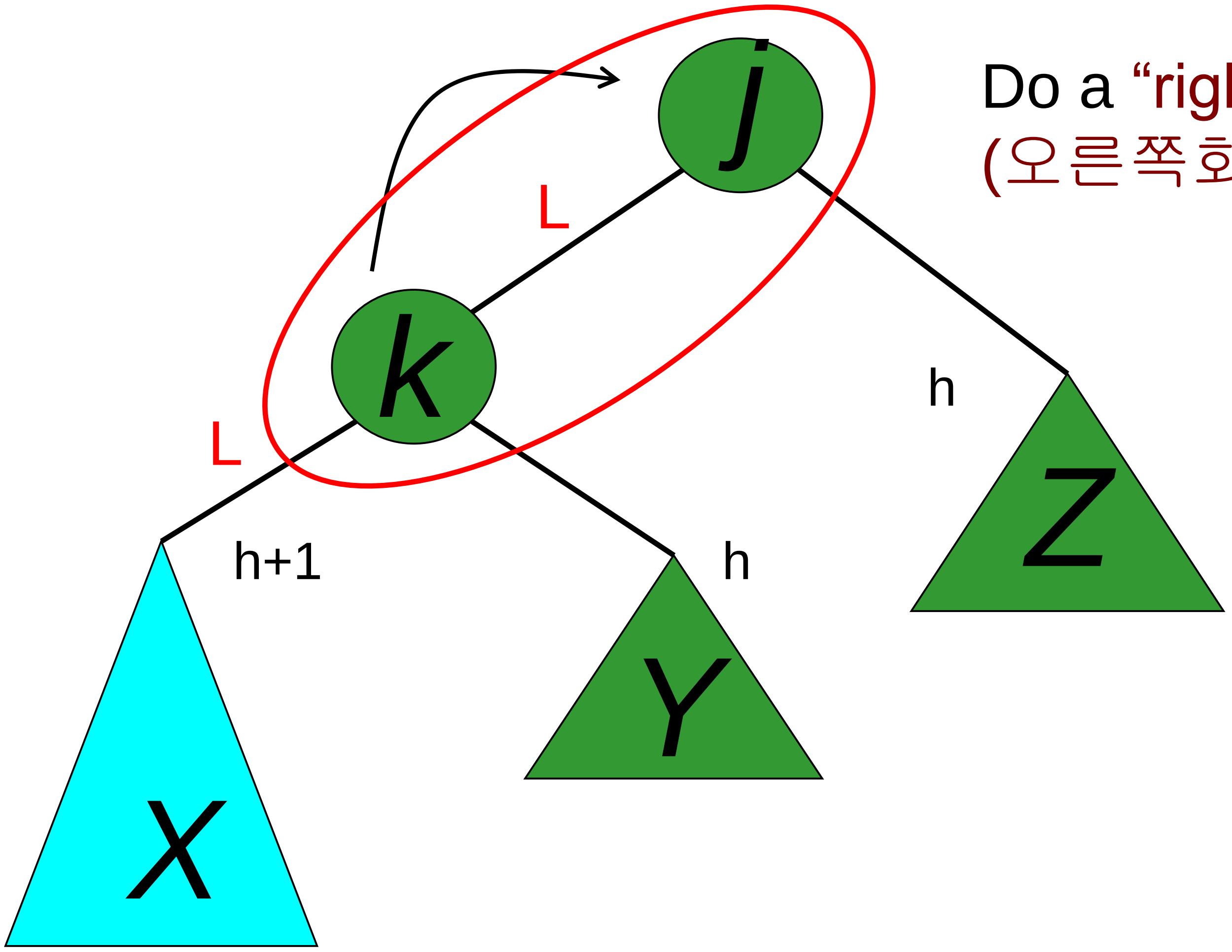


AVL Insertion: Outside Case



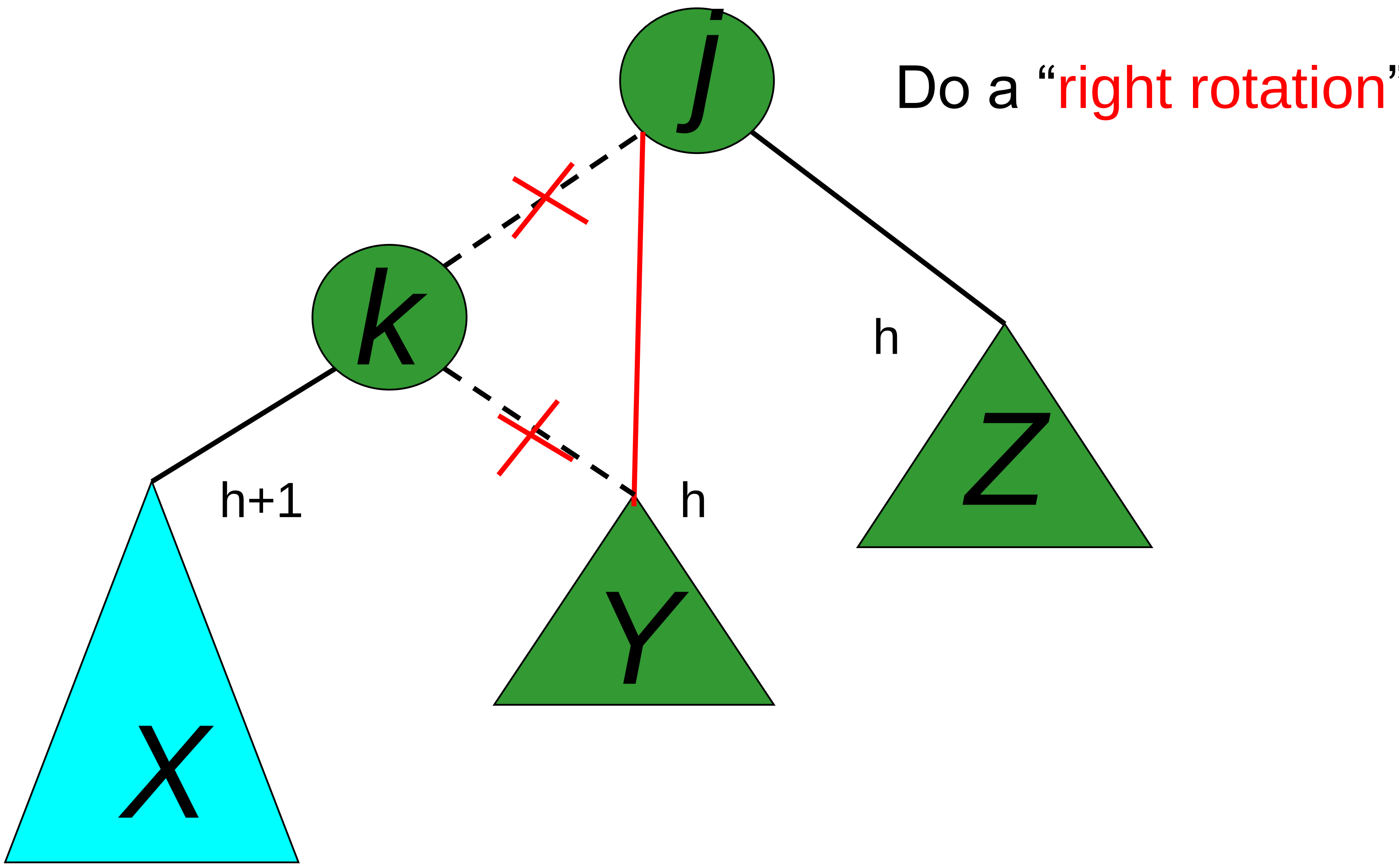
Inserting into X destroys the AVL property at node j

AVL Insertion: Outside Case

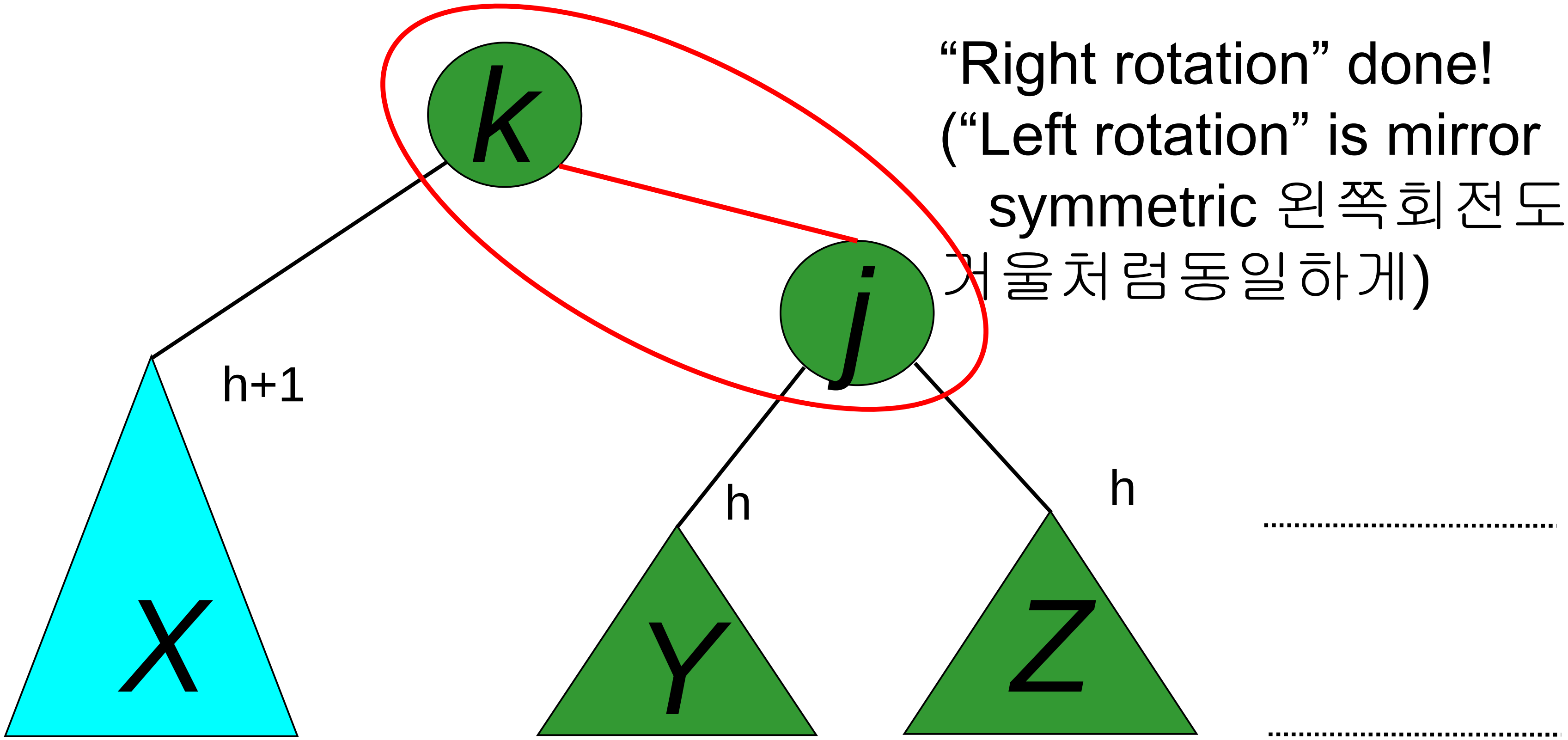


Do a “right rotation”
(오른쪽회전)

Single right rotation



Outside Case Completed

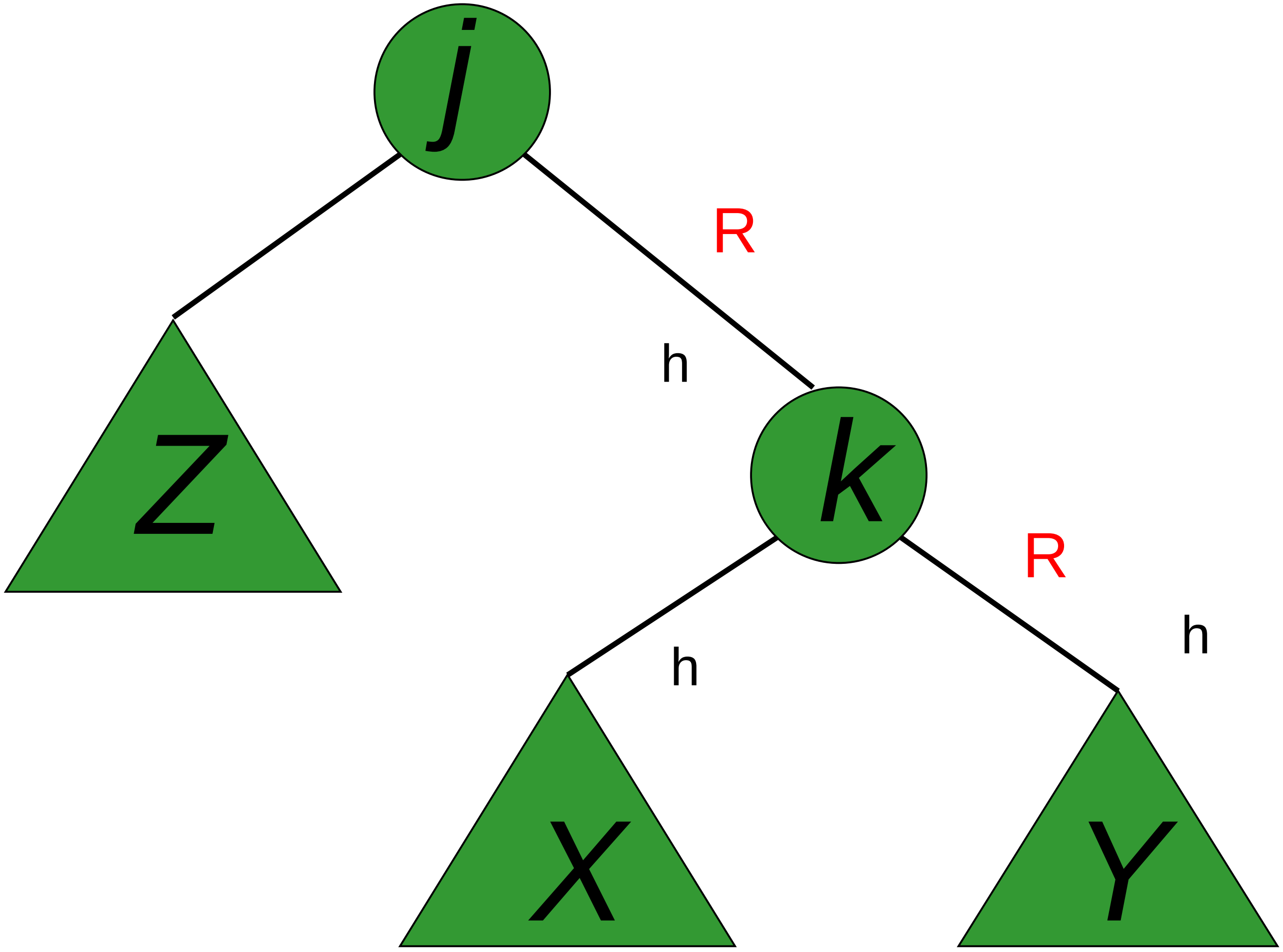


"Right rotation" done!
("Left rotation" is mirror symmetric 왼쪽회전도 거울처럼동일하게)

AVL property has been restored!

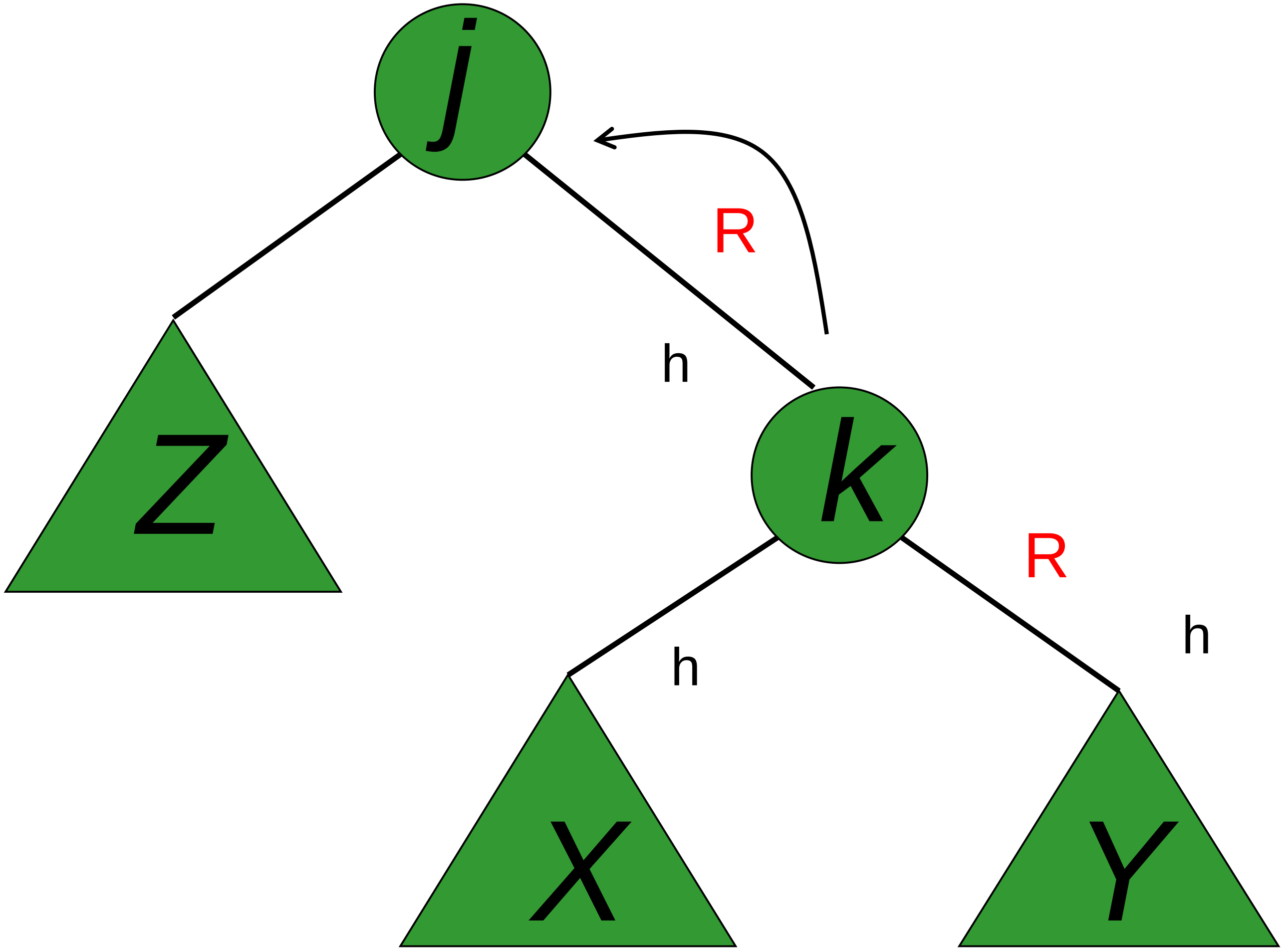
AVL Insertion: Right-Right

Exact same process as LL



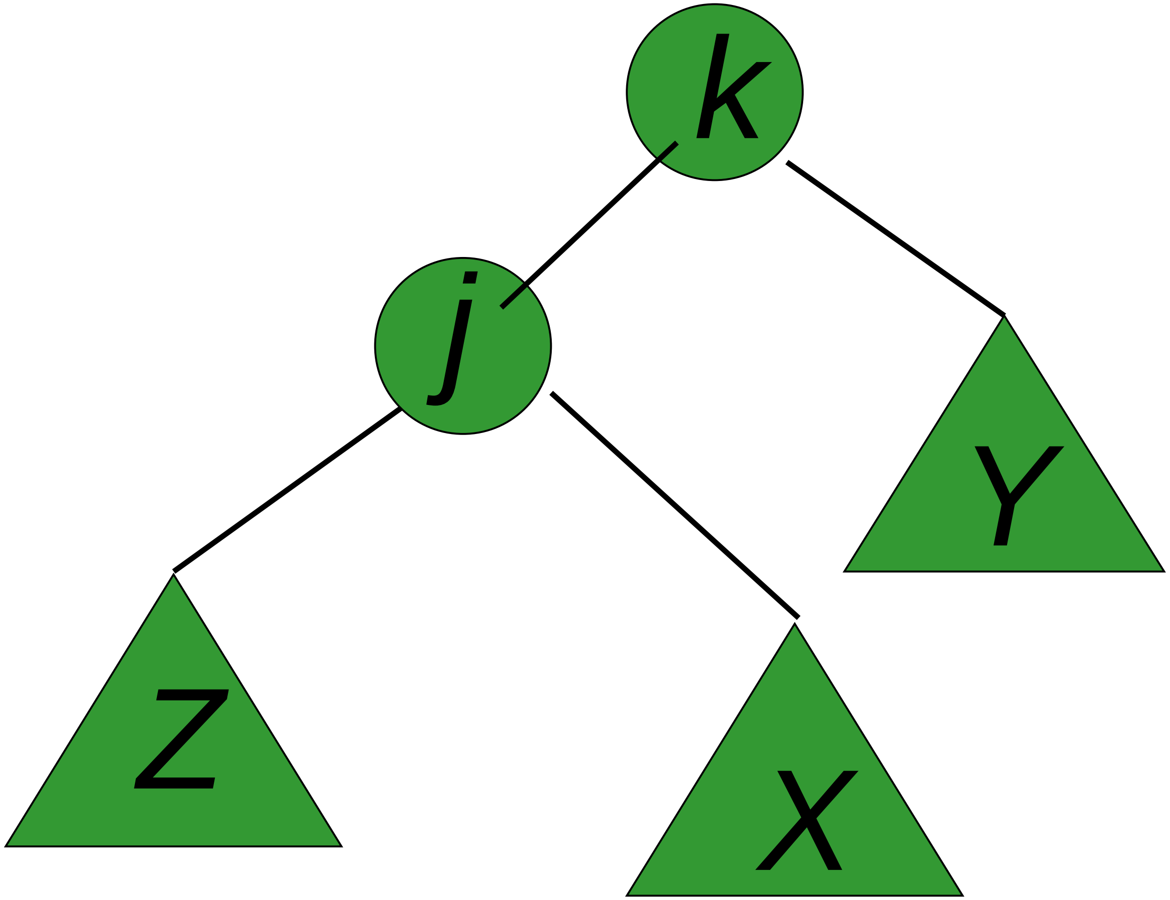
AVL Insertion: Right-Right

Exact same process as LL



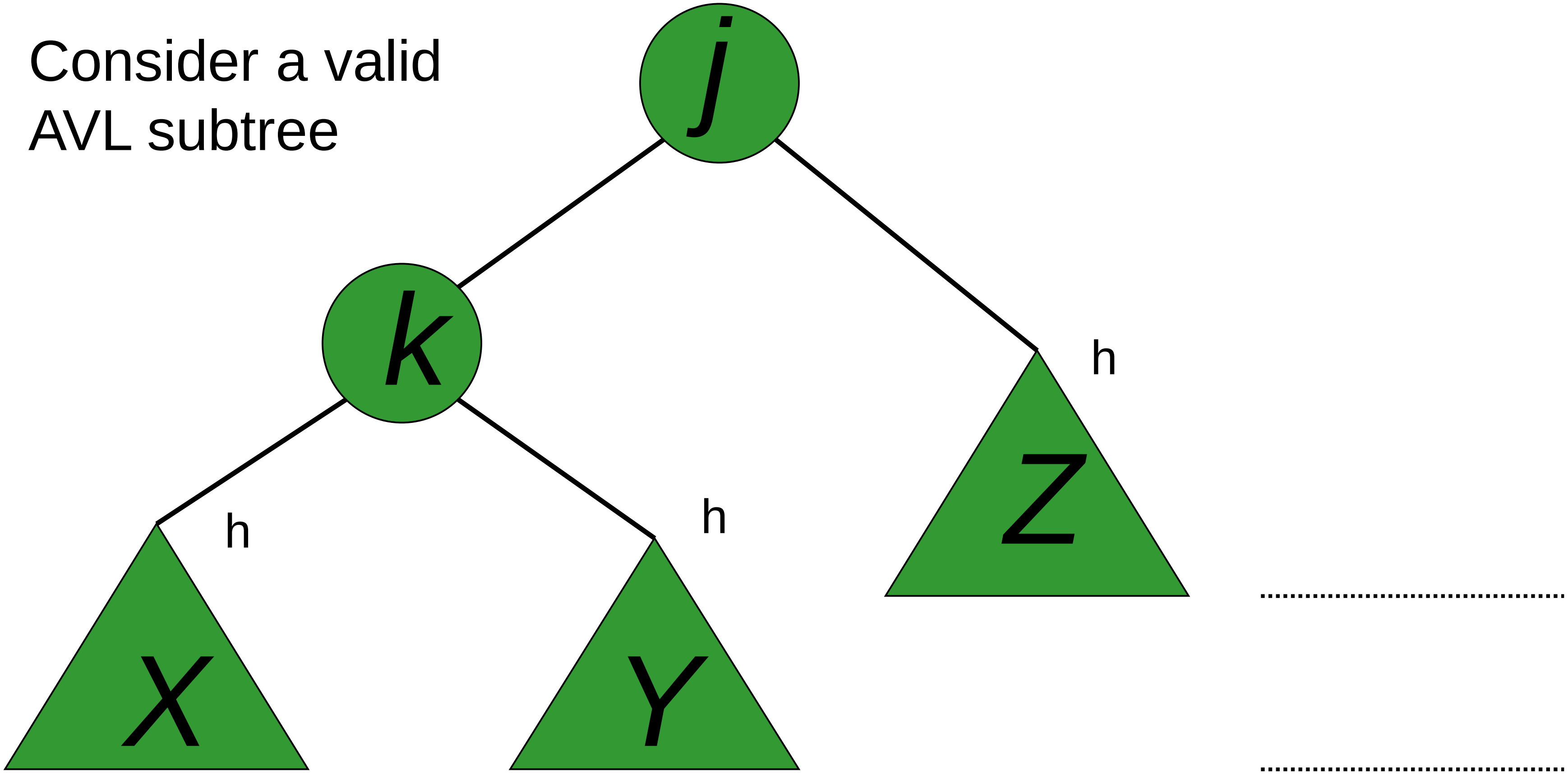
Single left rotation

Exact same process as LL



AVL Insertion: Inside Case

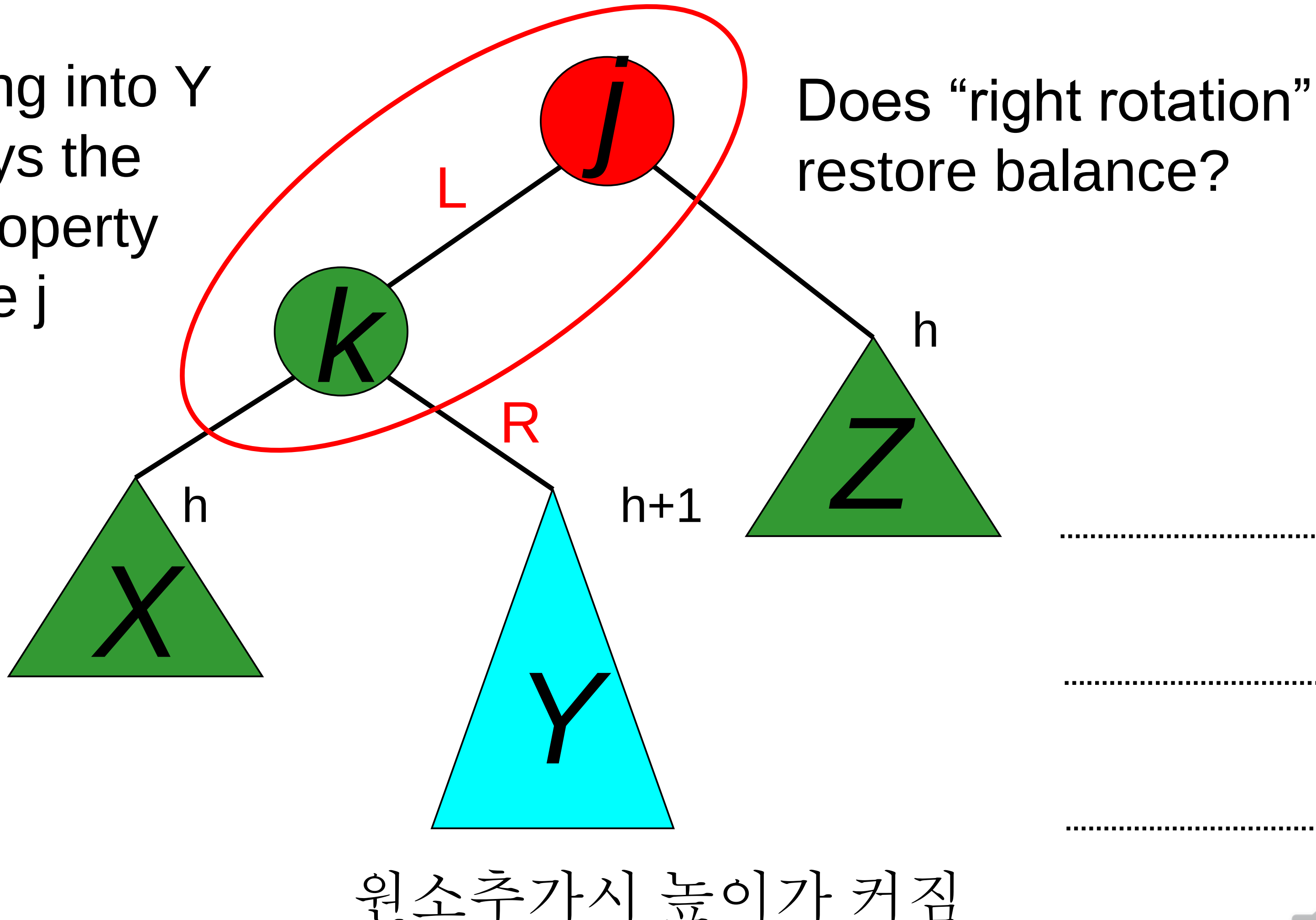
Consider a valid
AVL subtree



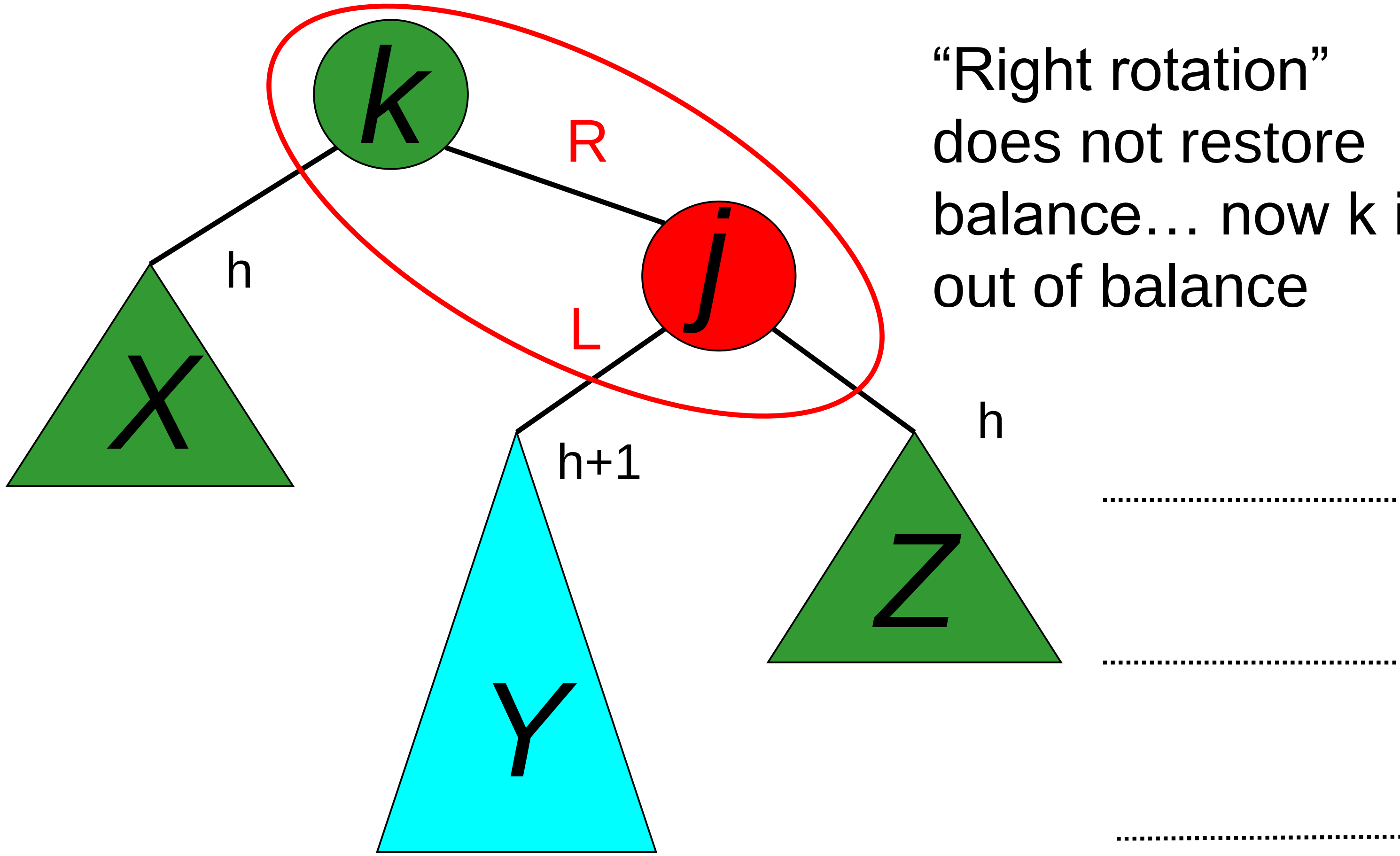
AVL Insertion: Left-Right

Inserting into Y destroys the AVL property at node j

Does "right rotation" restore balance?



AVL Insertion: Right-Left

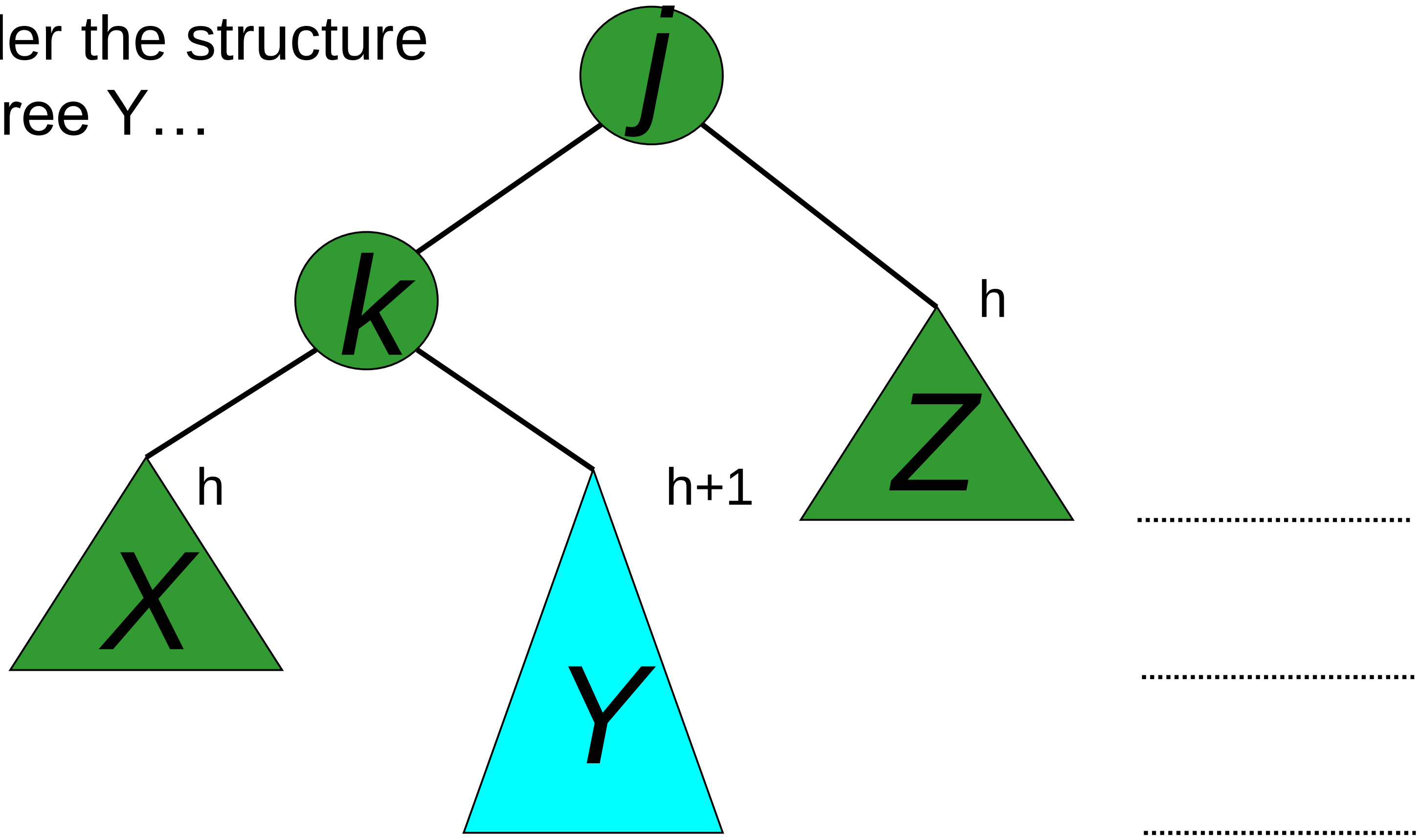


"Right rotation" does not restore balance… now k is out of balance

원소추가시 높이가 커짐

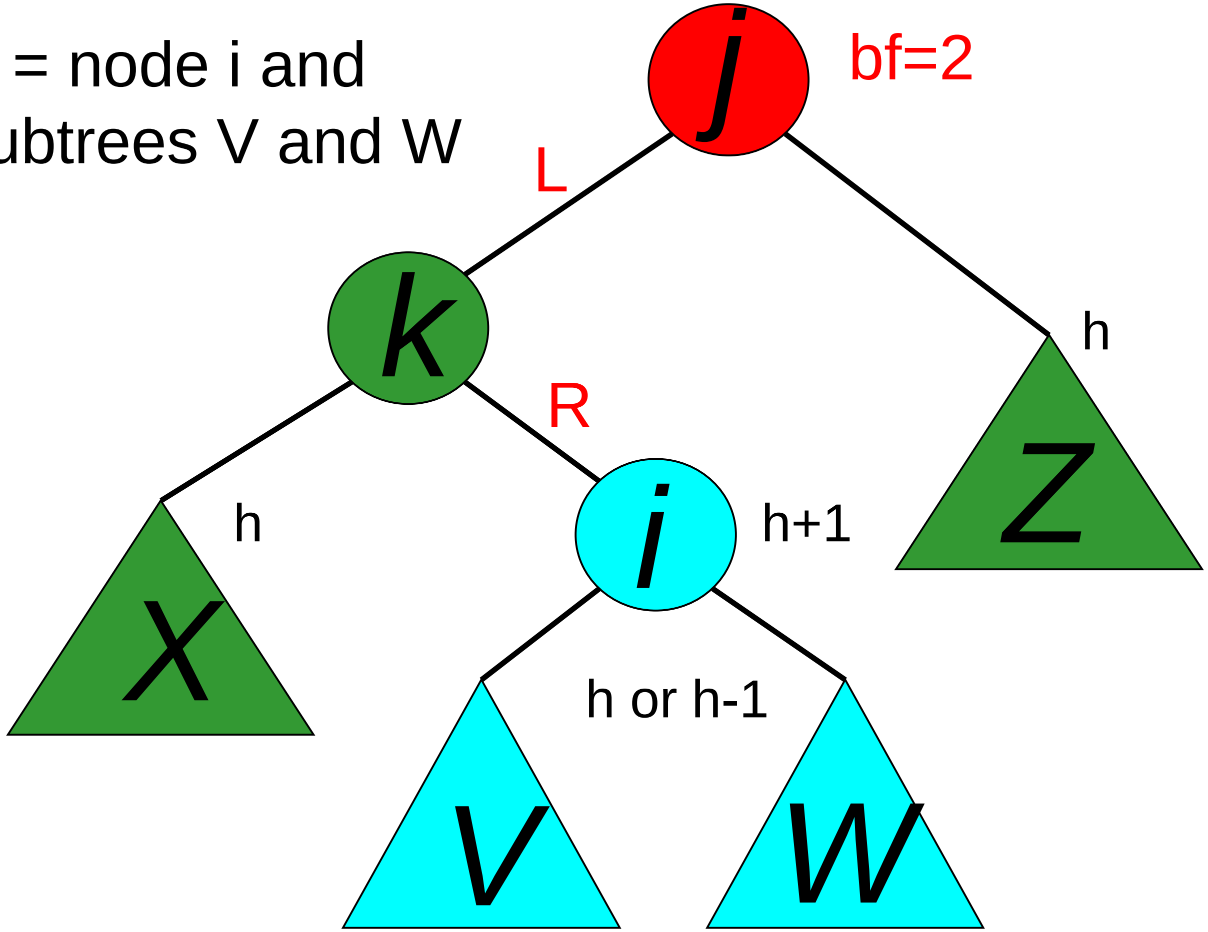
AVL Insertion: Double Rotation 이중회전

Consider the structure of subtree Y…

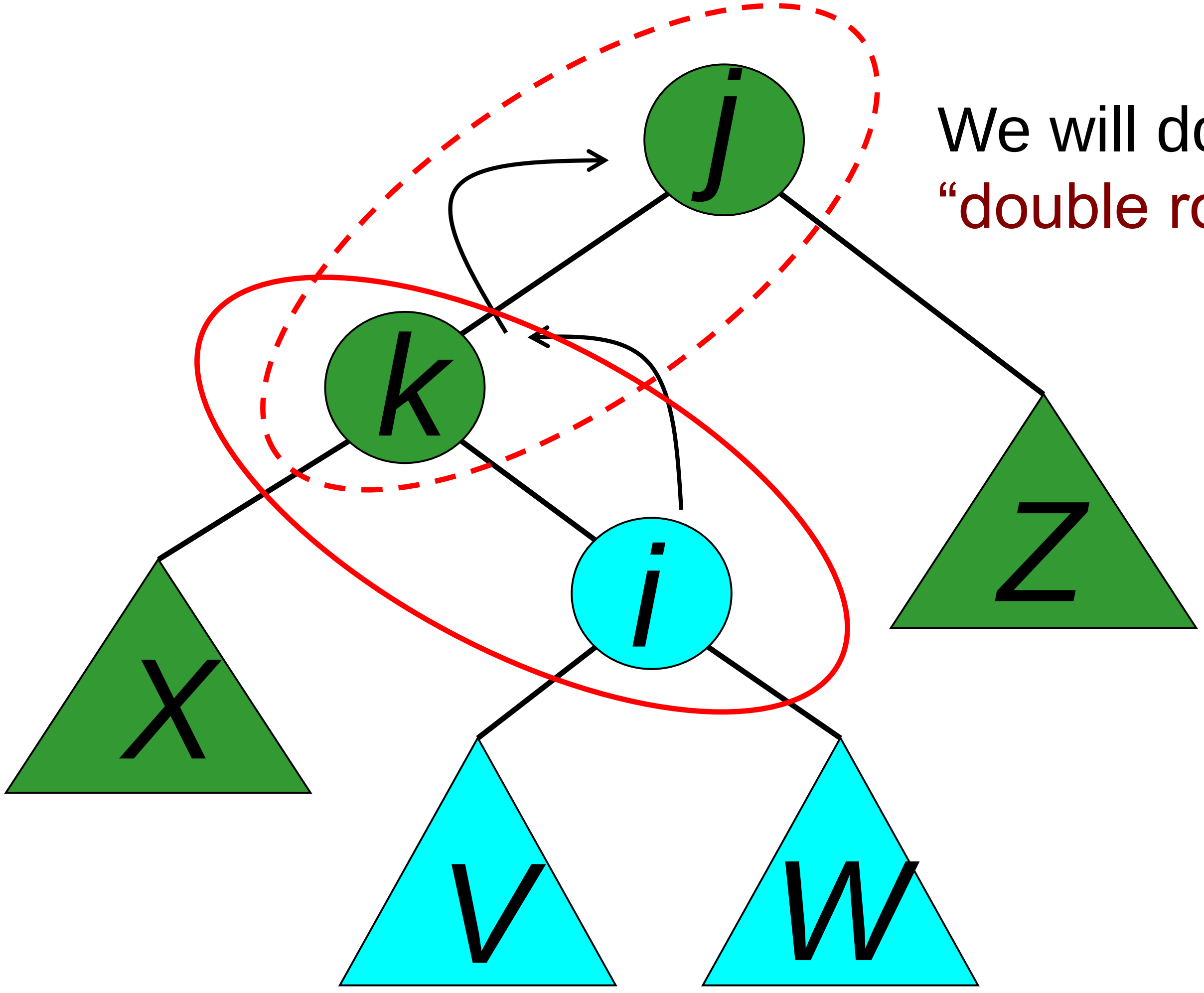


AVL Insertion: Inside Case

Y = node i and
subtrees V and W

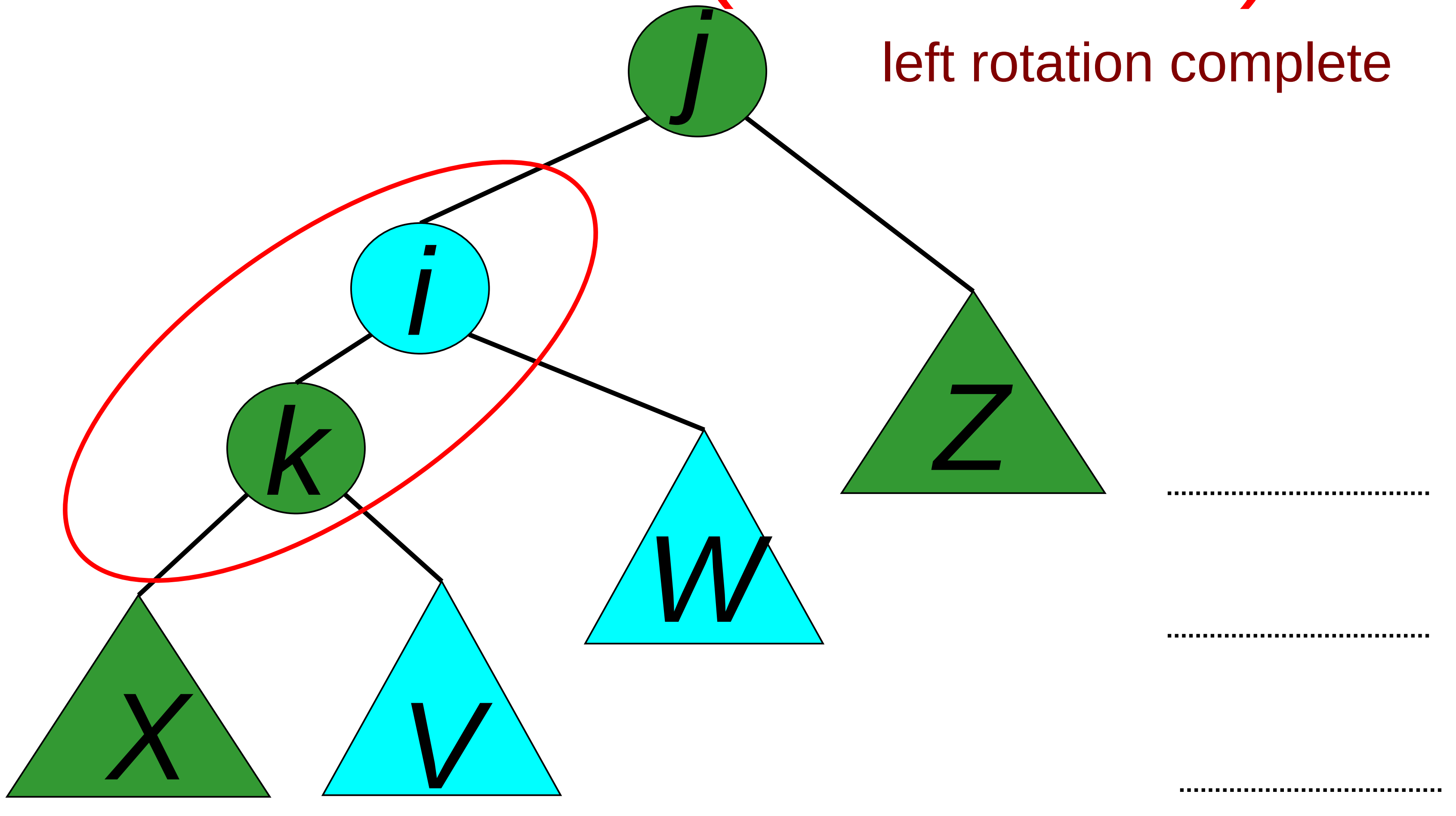


AVL Insertion: Inside Case

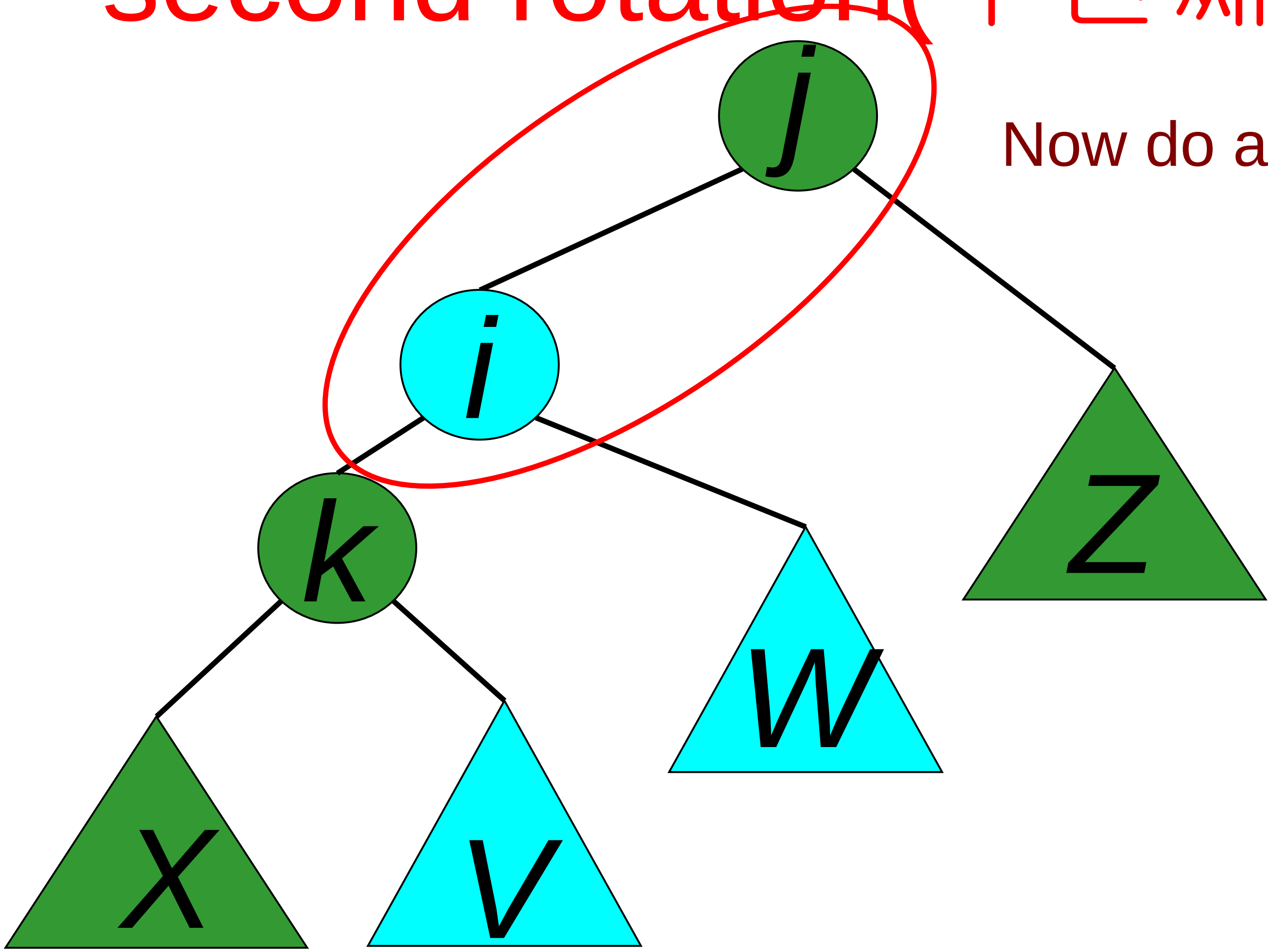


We will do a left-right "double rotation" . . .

Double rotation : first rotation(첫번째회전)



Double rotation : second rotation(두번째회전)



Now do a right rotation

....

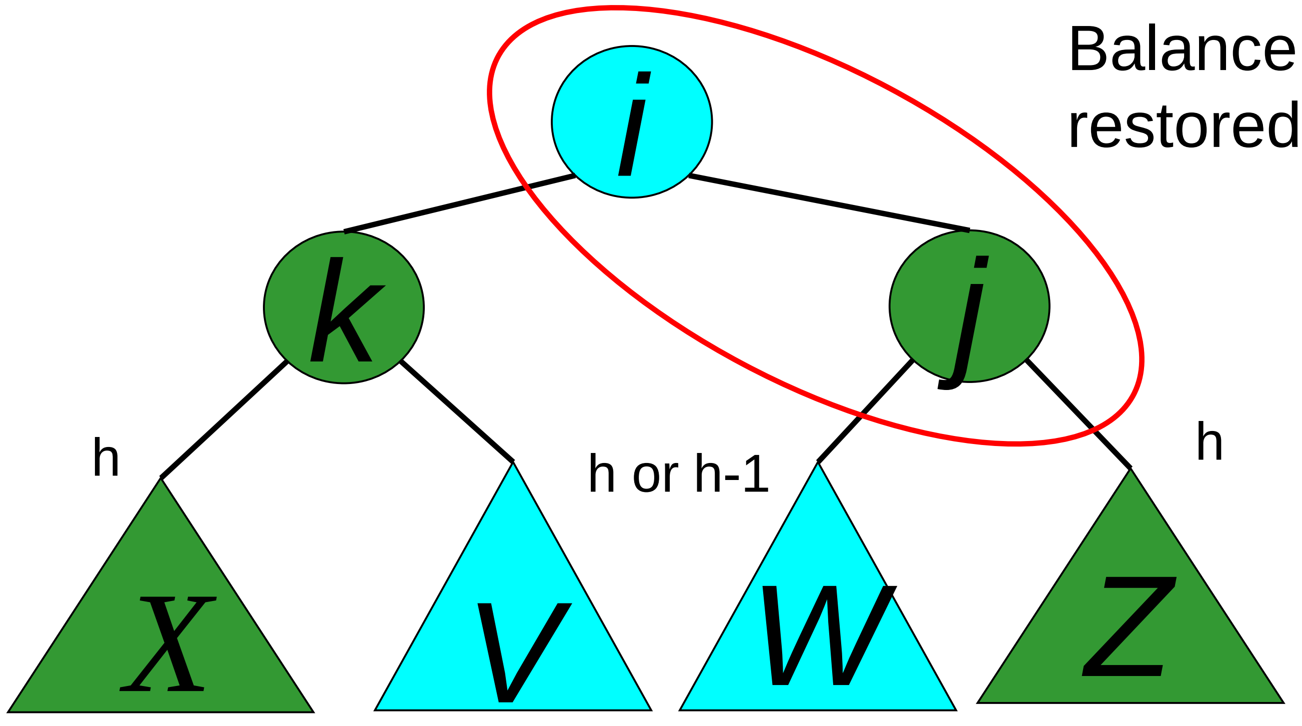
....

....

Double rotation : second rotation

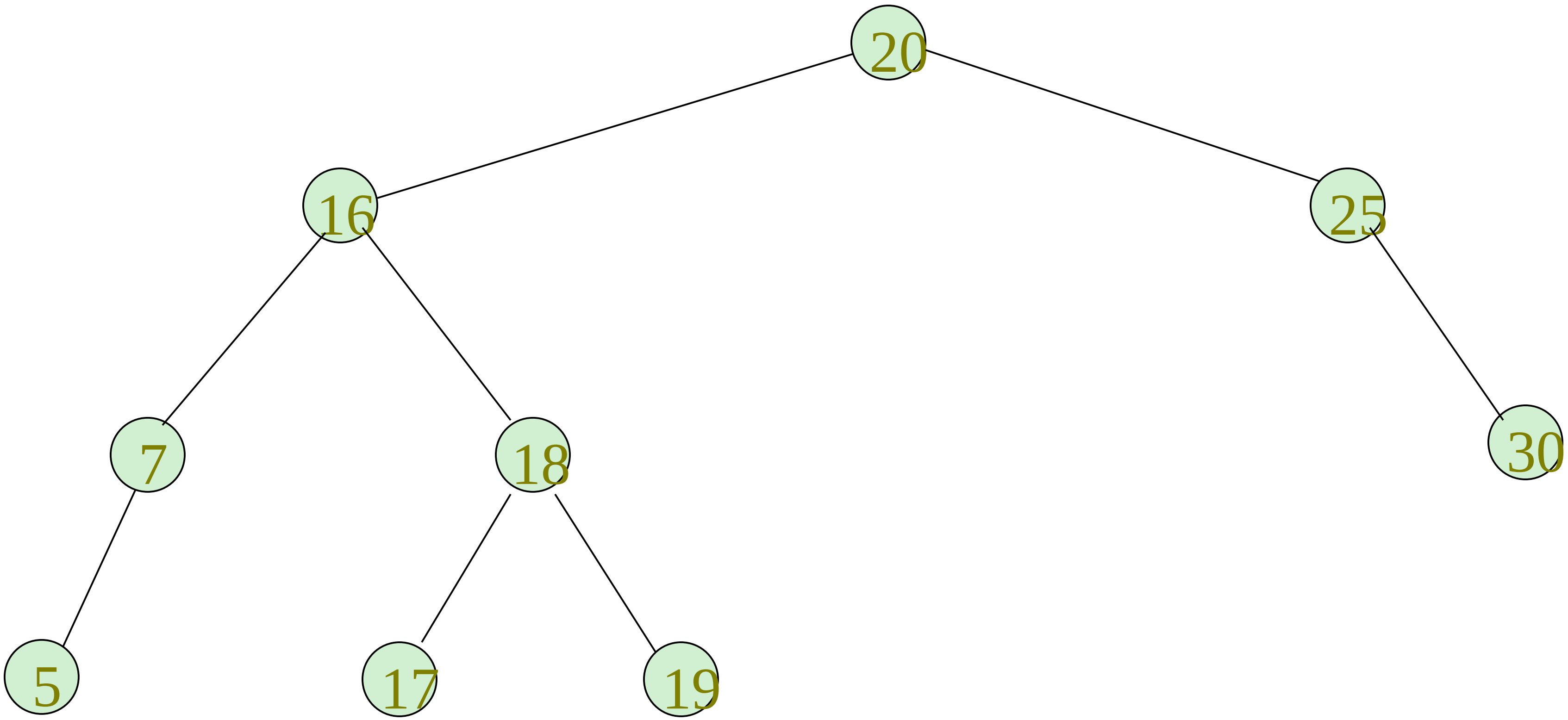
right rotation complete

Balance has been restored



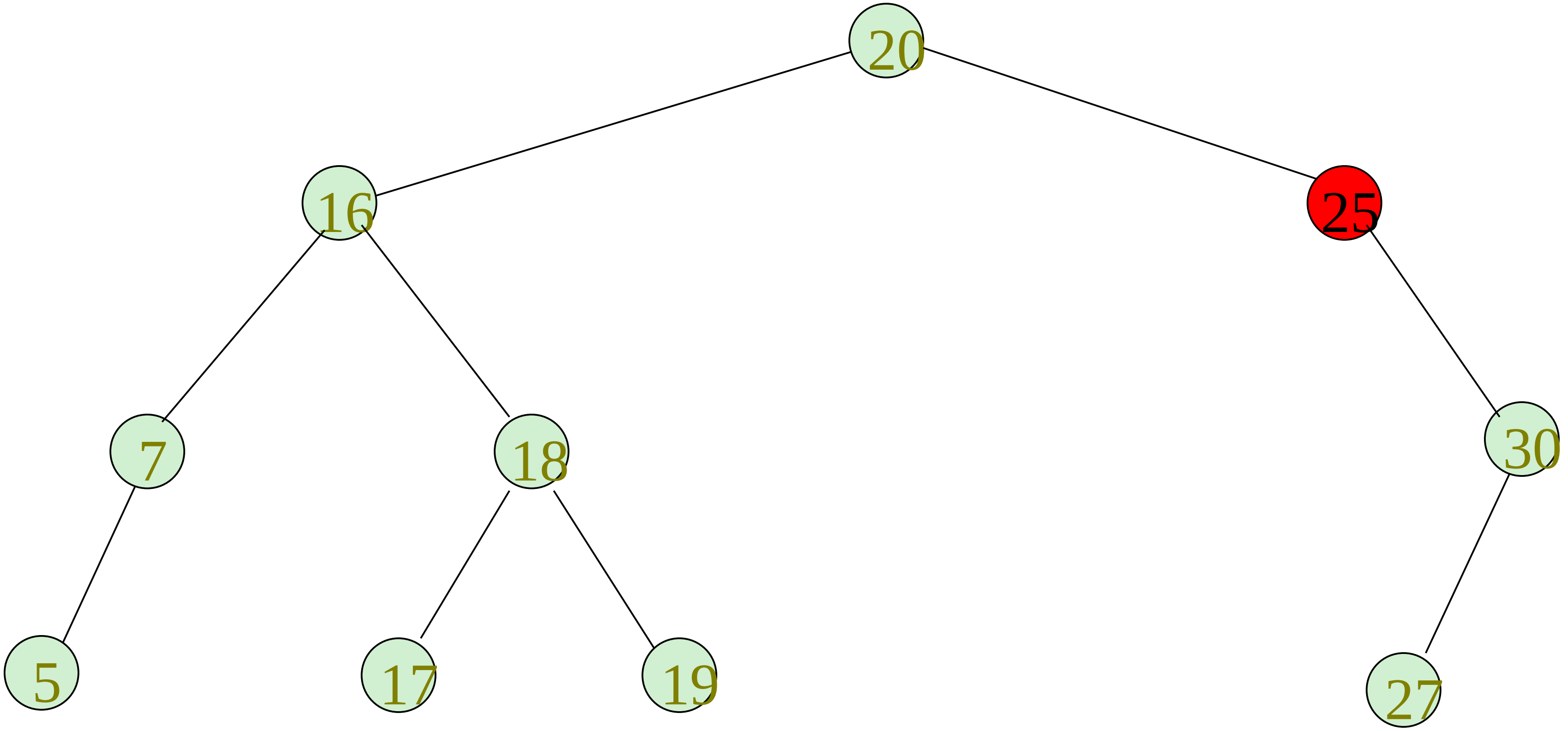
AVL Trees
(Adelson – Velskii – Landis)

AVL tree: yes



AVL Trees

AVL tree: No

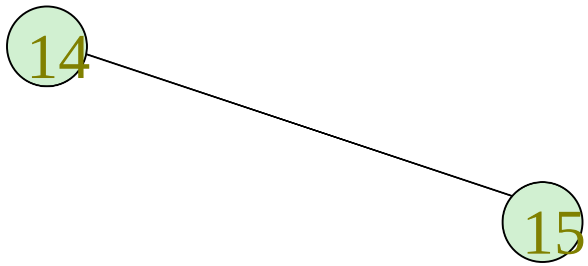


Example 1

AVL Tree Rotations

Single rotations: insert 14, 15, 16, 13, 12, 11, 10

- First insert 14 and 15:

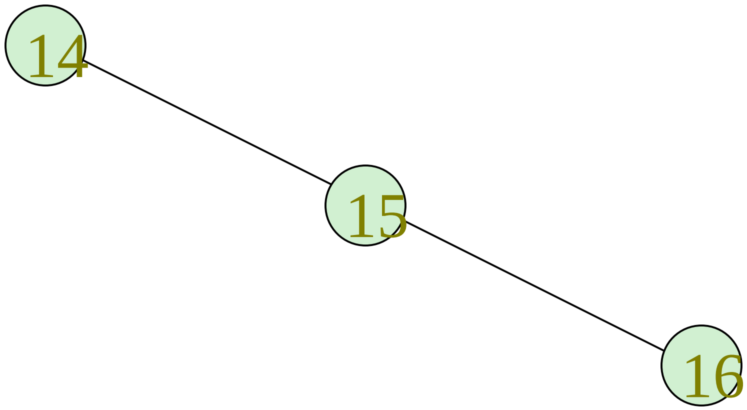


- Now insert 16.

AVL Tree Rotations

Single rotations:

- Inserting 16 causes AVL violation:

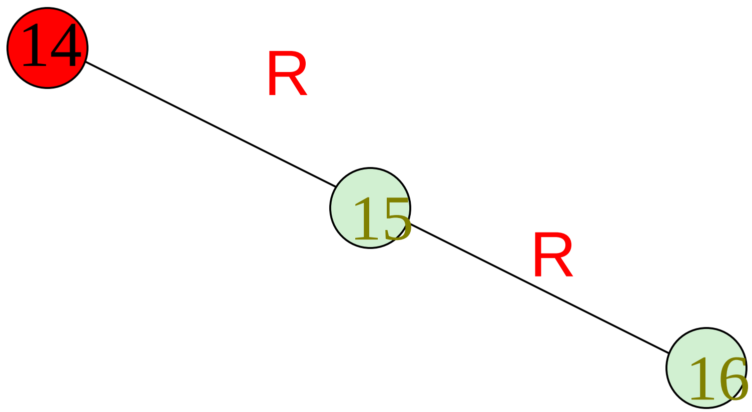


- Need to rotate.

AVL Tree Rotations

Single rotations:

- Inserting 16 causes AVL violation:

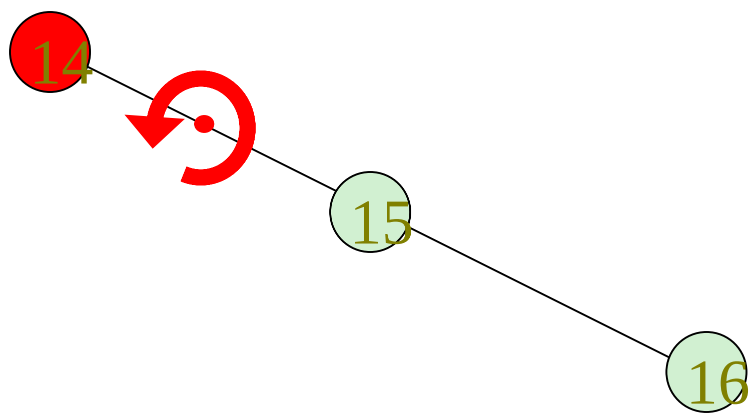


- Need to rotate.

AVL Tree Rotations

Single rotations:

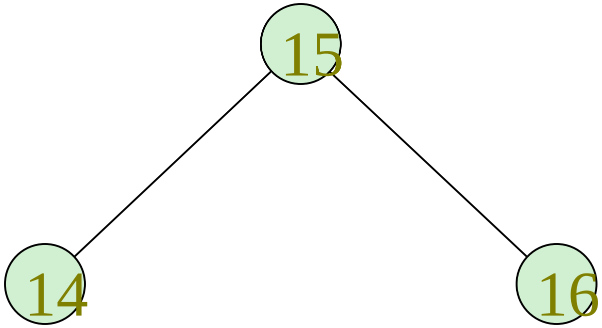
- Rotation type:



AVL Tree Rotations

Single rotations:

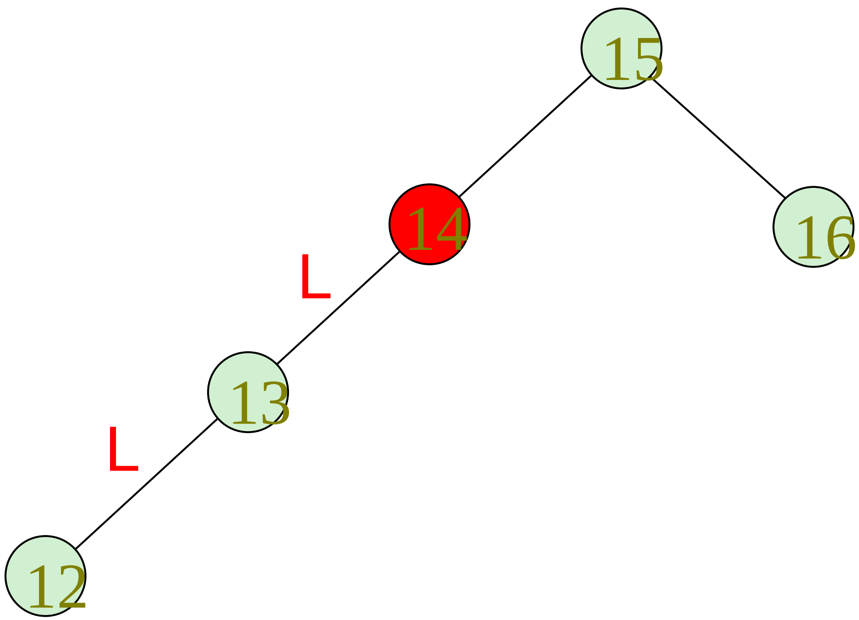
- Rotation restores AVL balance:



AVL Tree Rotations

Single rotations:

- Now insert 13 and 12:

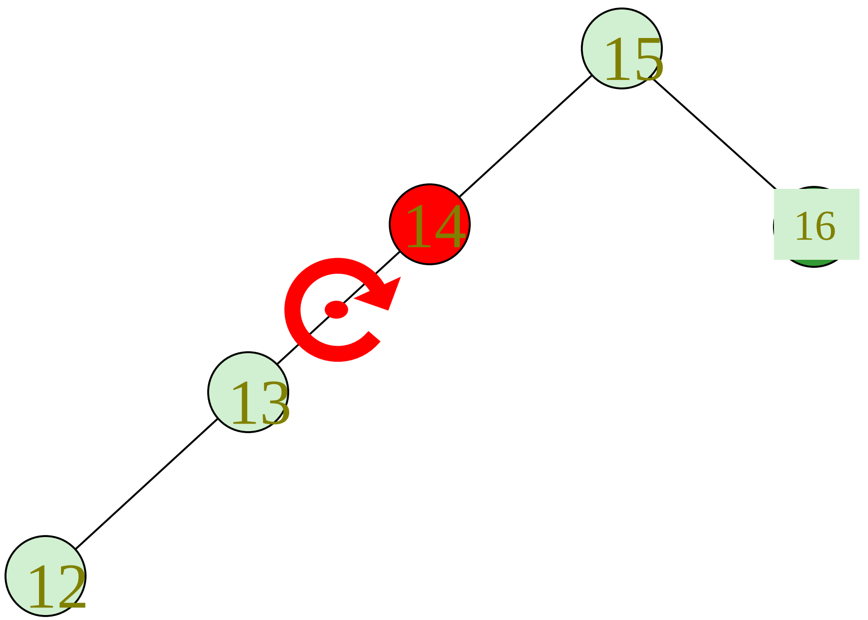


- AVL violation - need to rotate.

AVL Tree Rotations

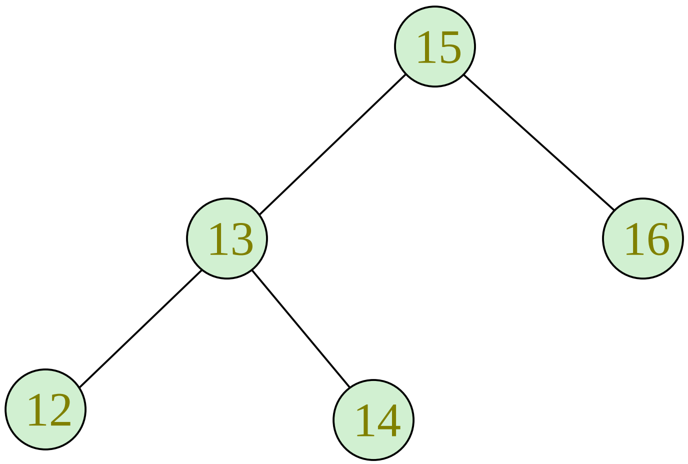
Single rotations:

- Rotation type:



AVL Tree Rotations

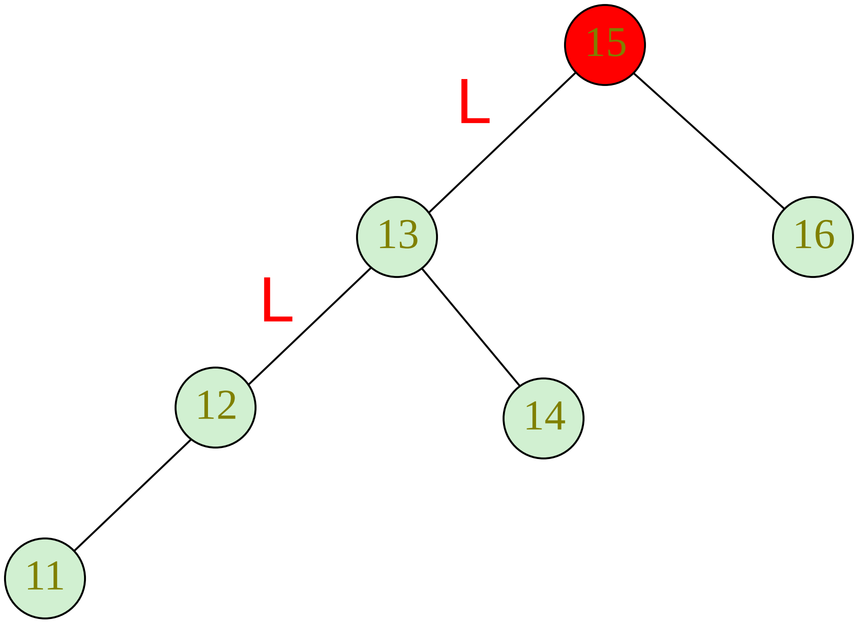
Single rotations:



- Now insert 11.

AVL Tree Rotations

Single rotations:

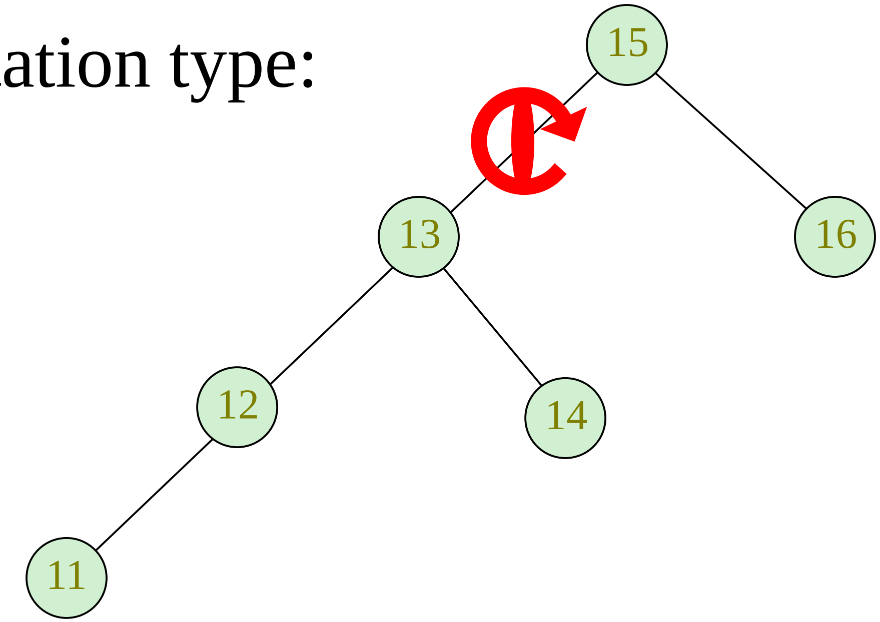


- AVL violation – need to rotate

AVL Tree Rotations

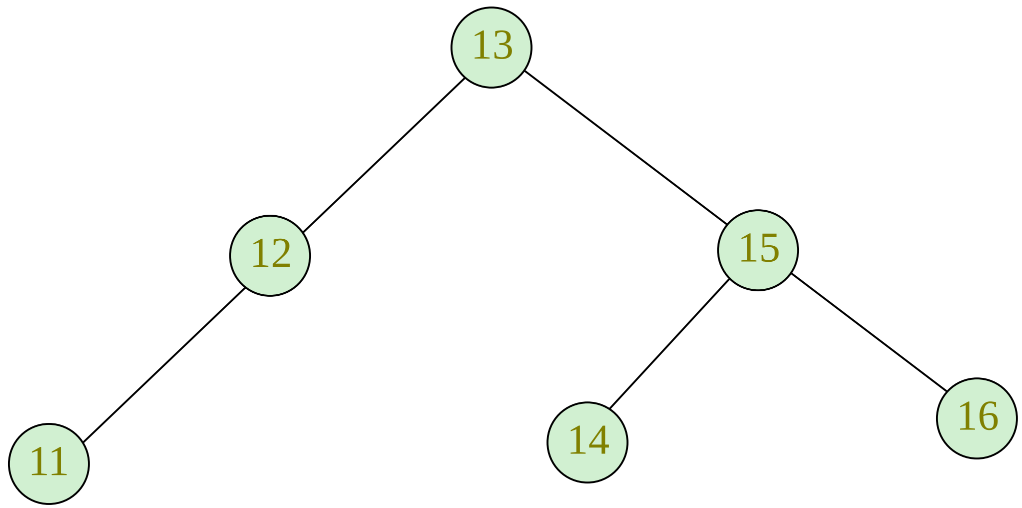
Single rotations:

- Rotation type:



AVL Tree Rotations

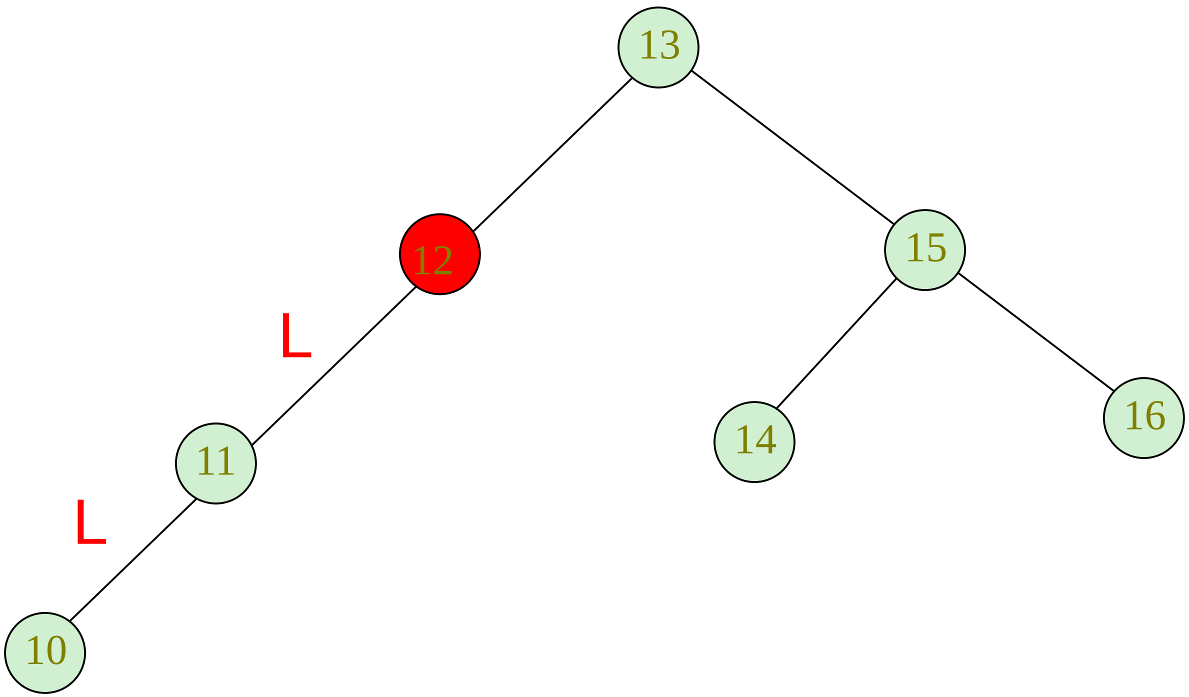
Single rotations:



- Now insert 10.

AVL Tree Rotations

Single rotations:

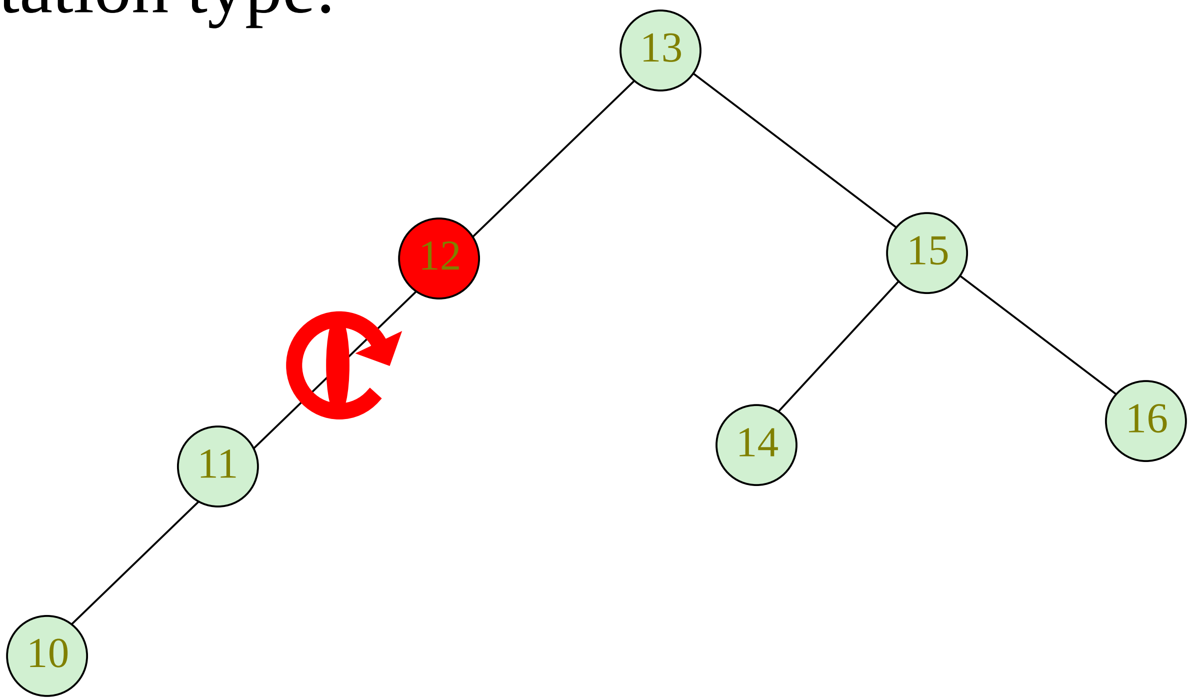


- AVL violation – need to rotate

AVL Tree Rotations

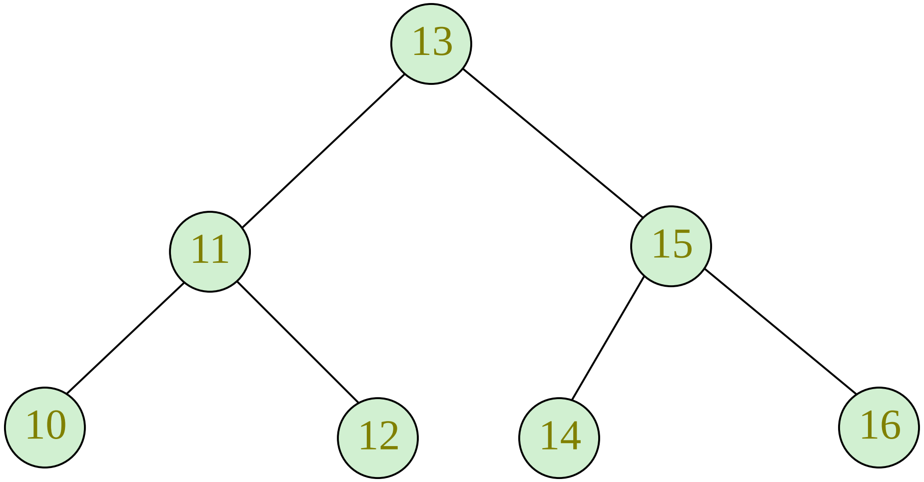
Single rotations:

- Rotation type:



AVL Tree Rotations

Single rotations:



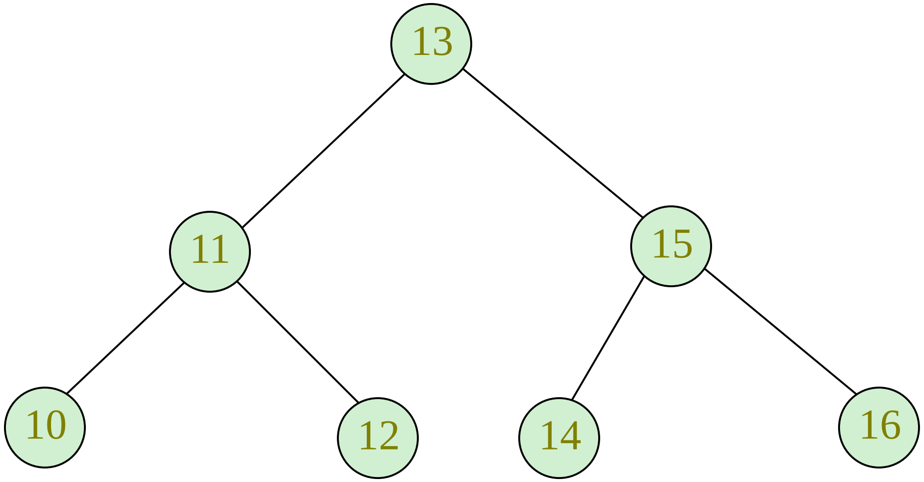
- AVL balance restored.

Continue

AVL Tree Rotations

Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

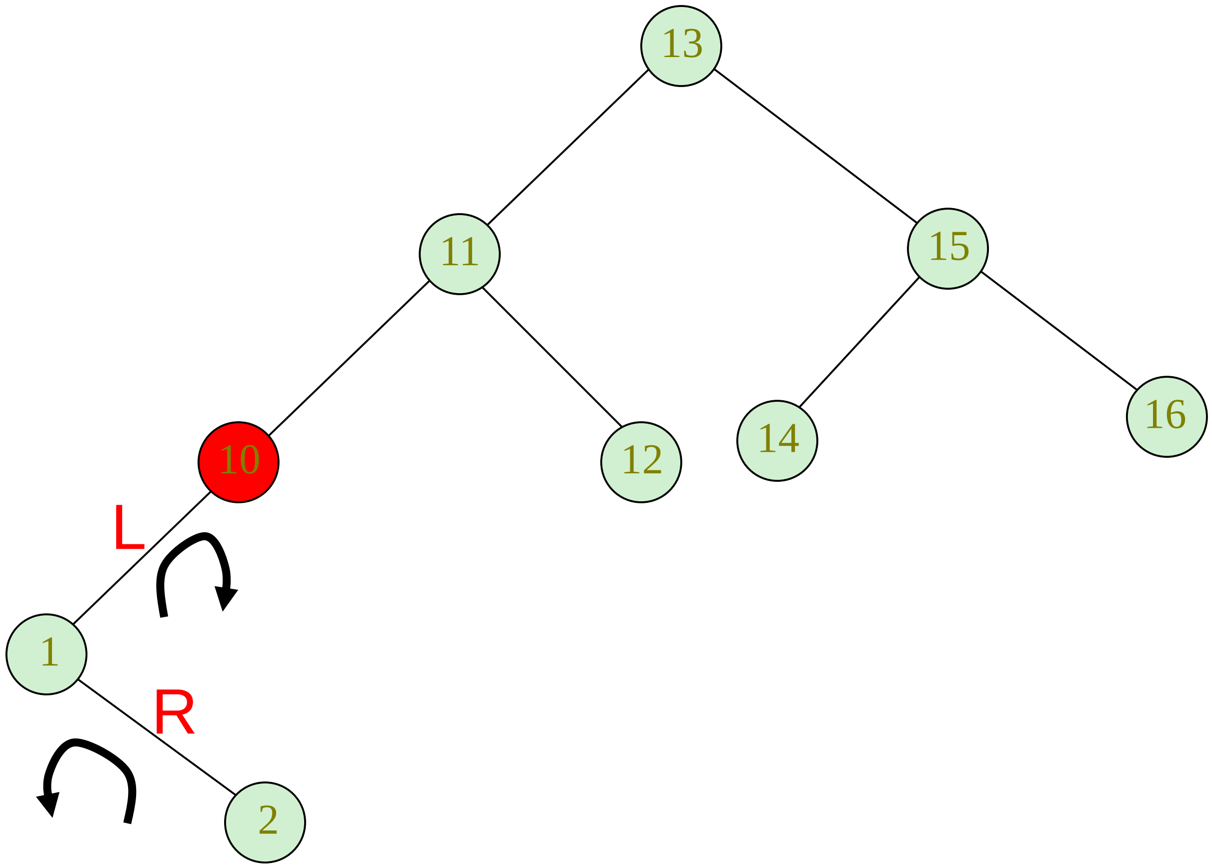
- First insert 1 and 2:



AVL Tree Rotations

Double rotations:

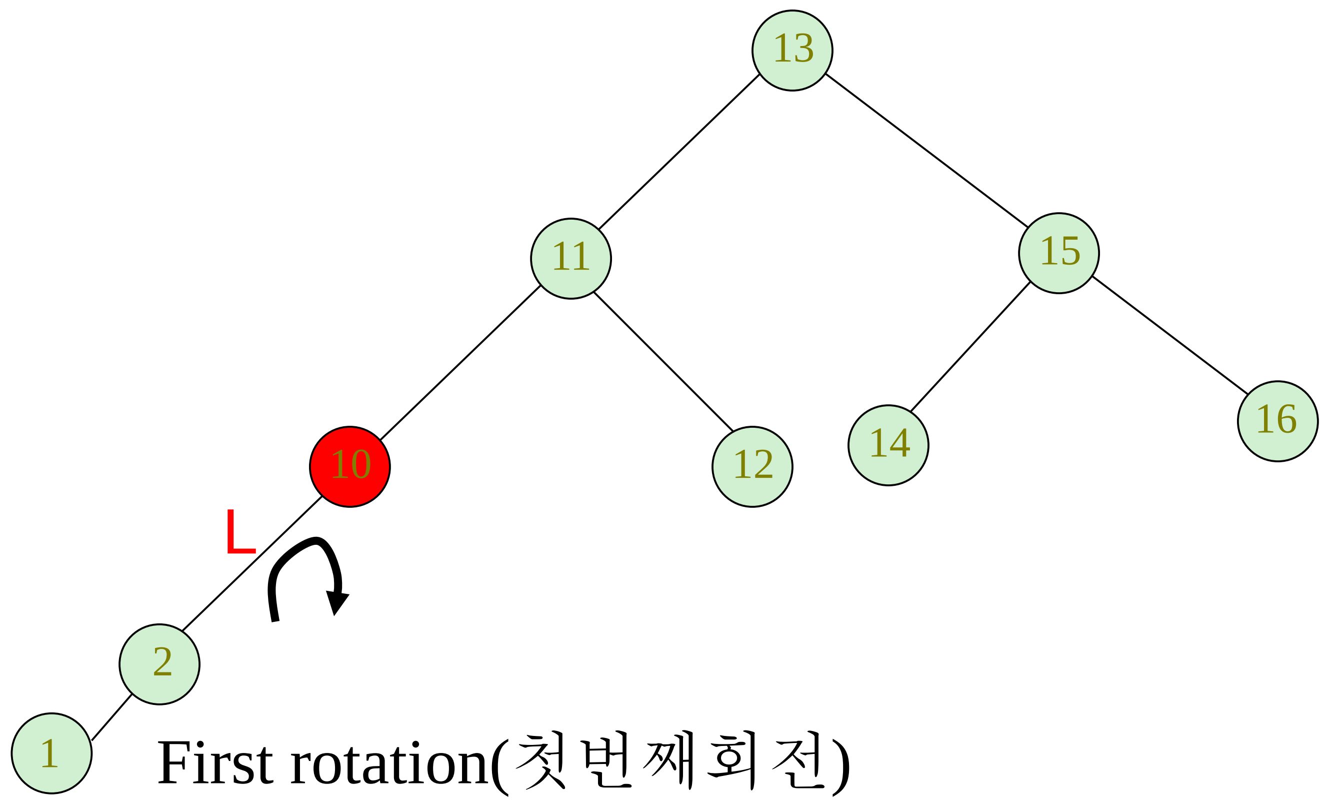
- AVL violation - rotate



AVL Tree Rotations

Double rotations:

- AVL violation - rotate

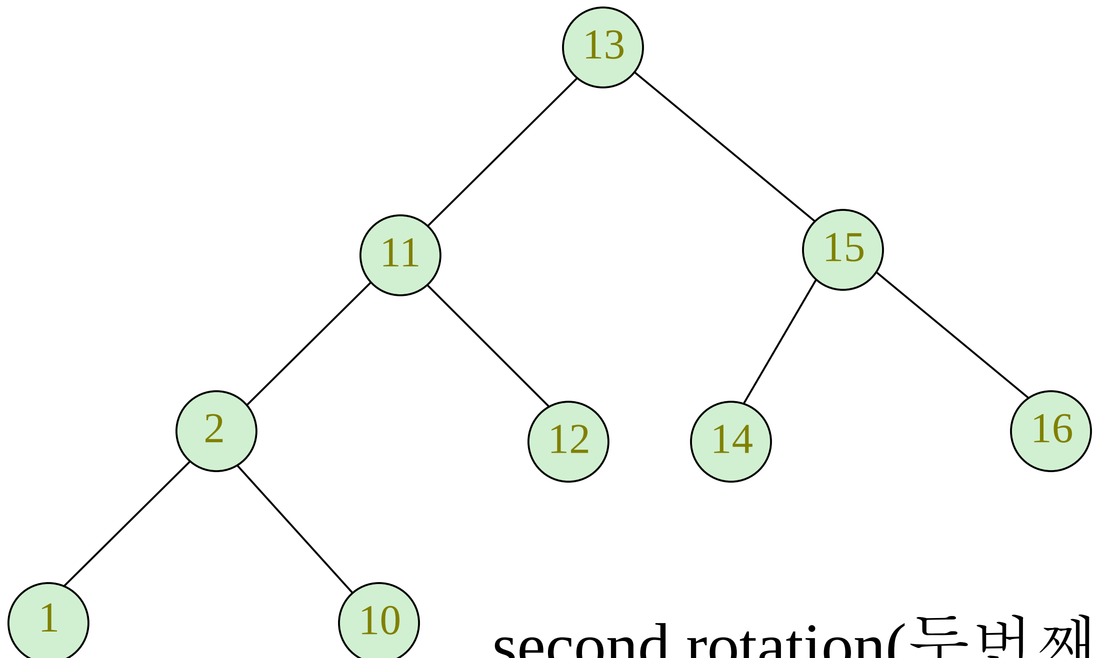


First rotation(첫번째회전)

AVL Tree Rotations

Double rotations:

- AVL balance restored:



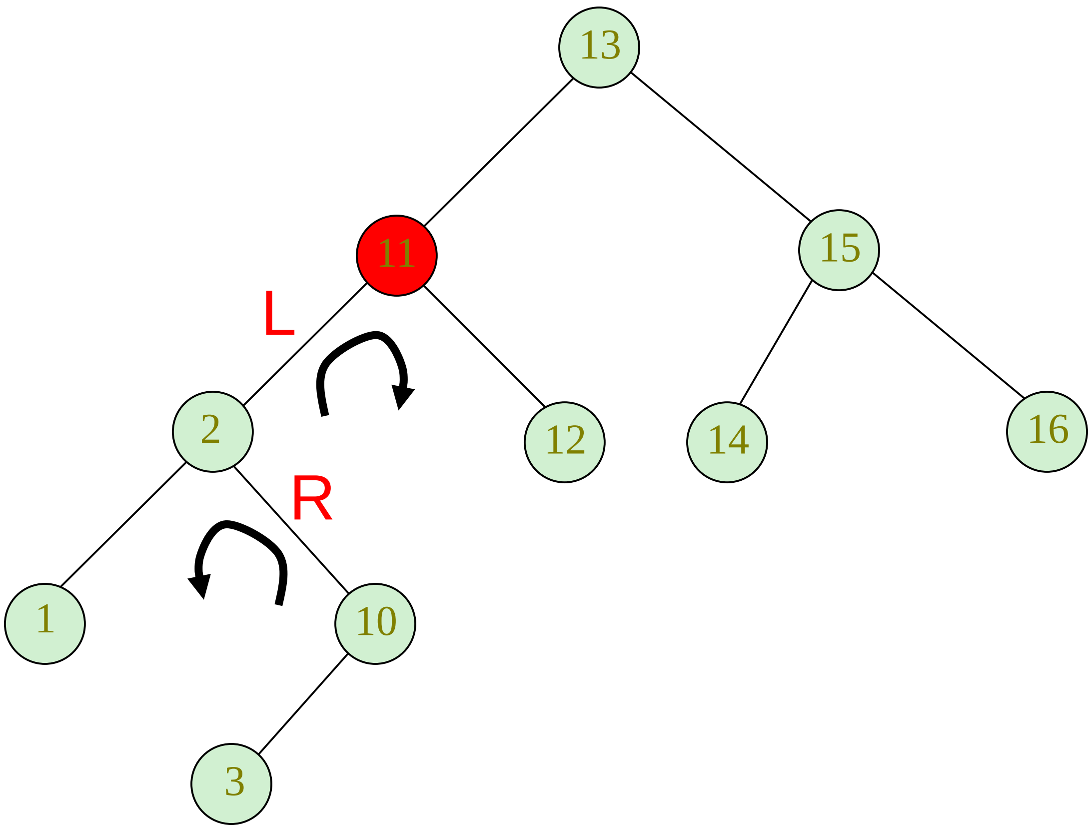
second rotation(두번째회전)

- Now insert 3.

AVL Tree Rotations

Double rotations:

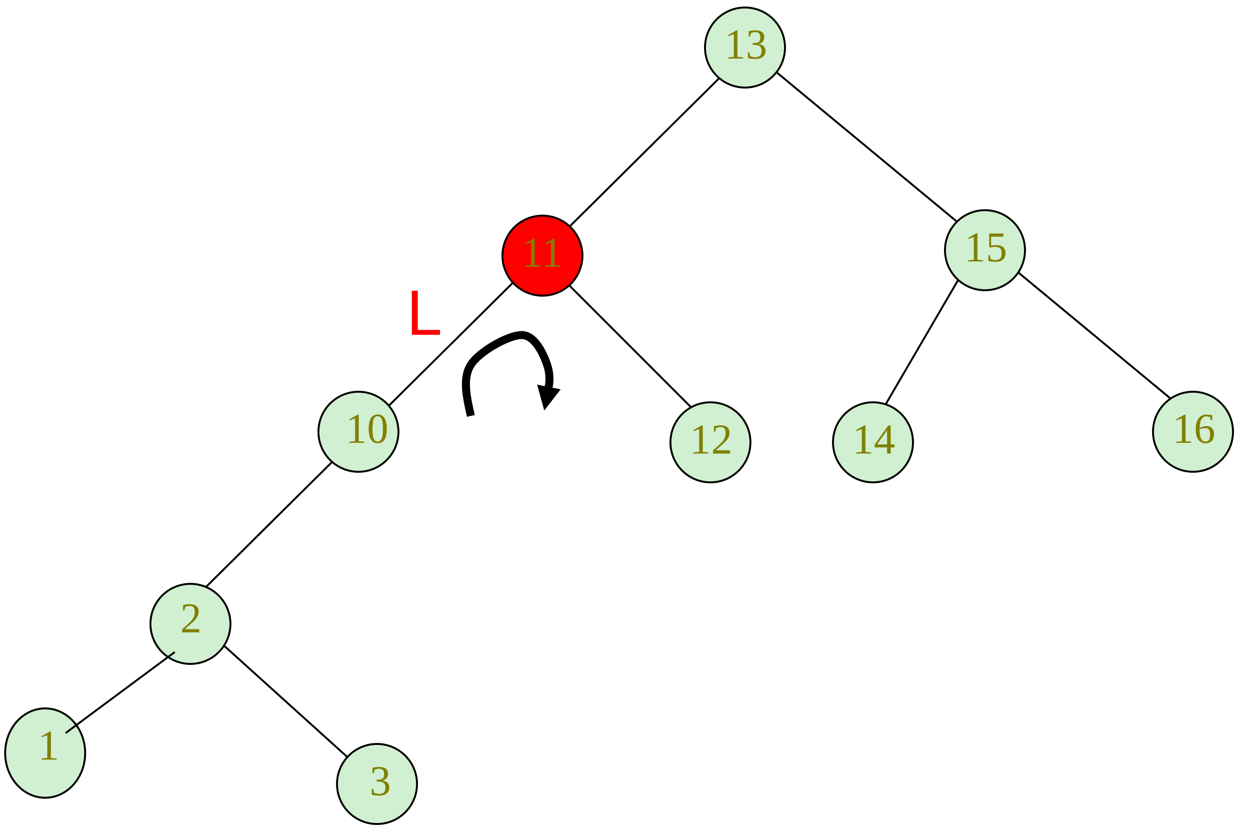
- AVL violation – rotate:



AVL Tree Rotations

Double rotations:

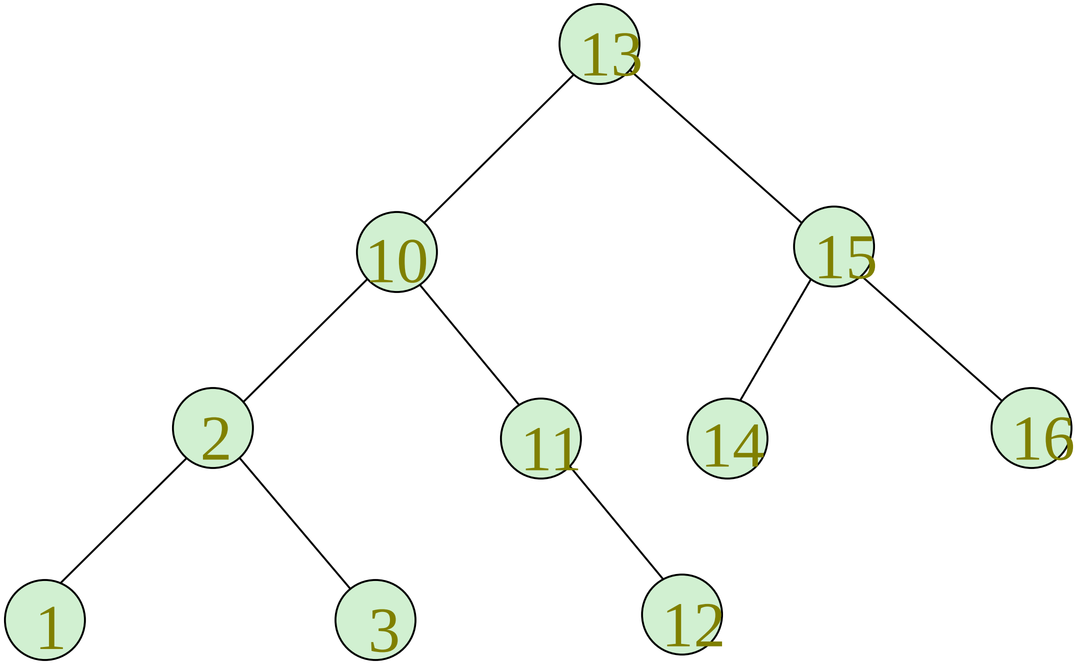
- AVL violation – rotate:



AVL Tree Rotations

Double rotations:

- AVL balance restored:

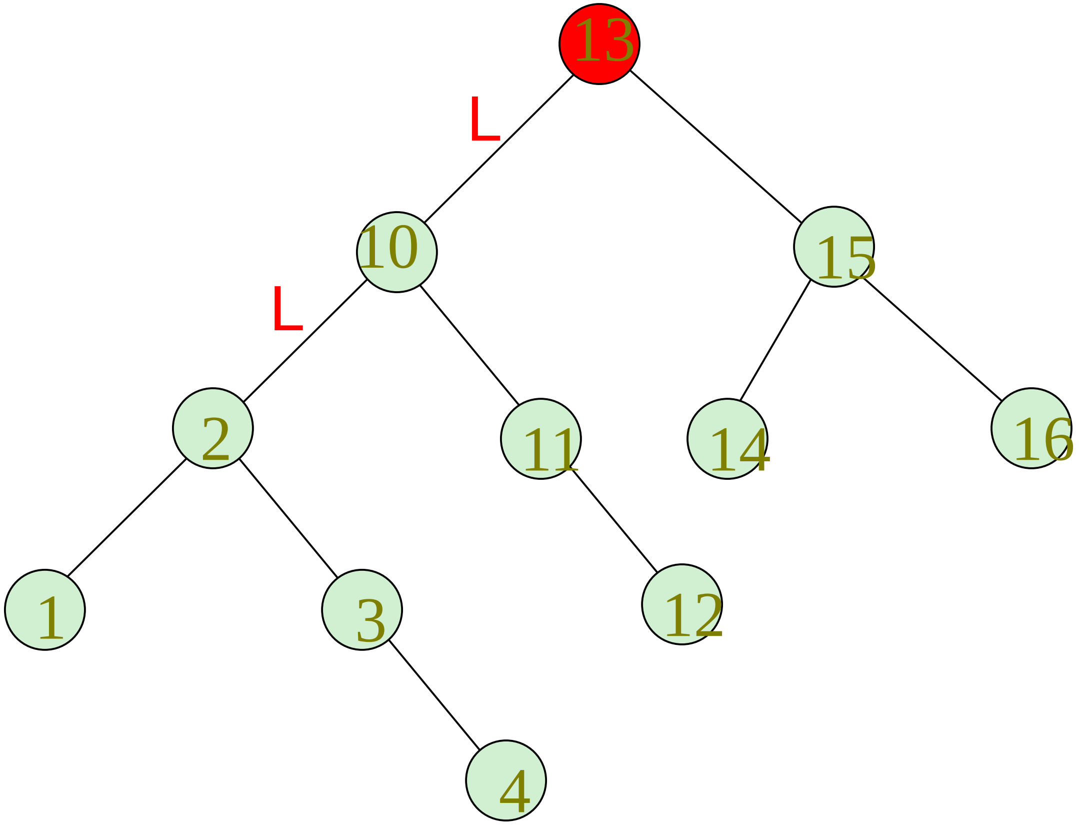


- Now insert 4.

AVL Tree Rotations

Double rotations:

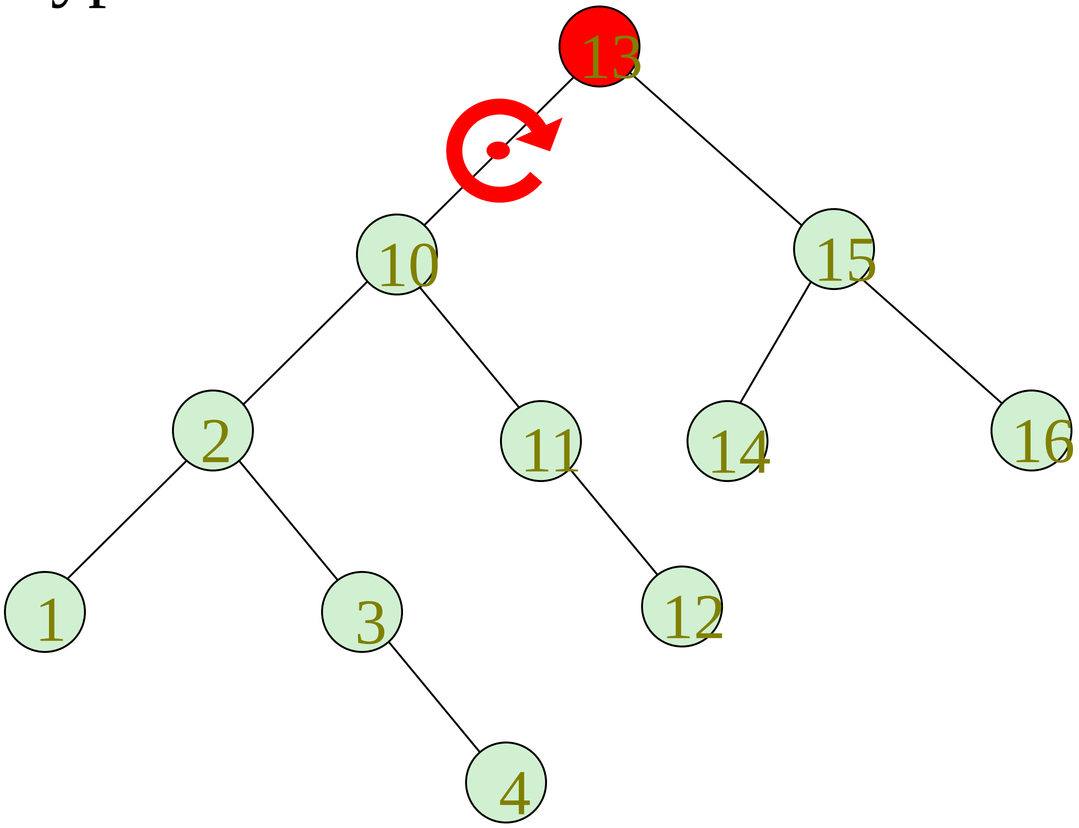
- AVL violation - rotate



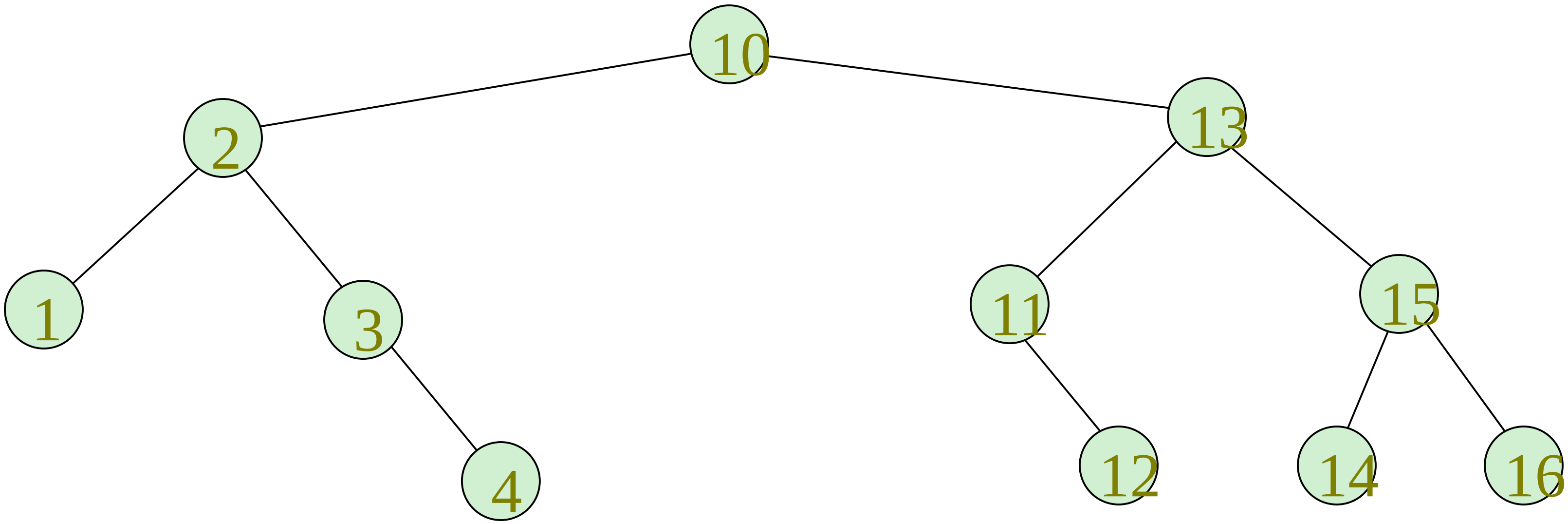
AVL Tree Rotations

Double rotations:

- Rotation type:



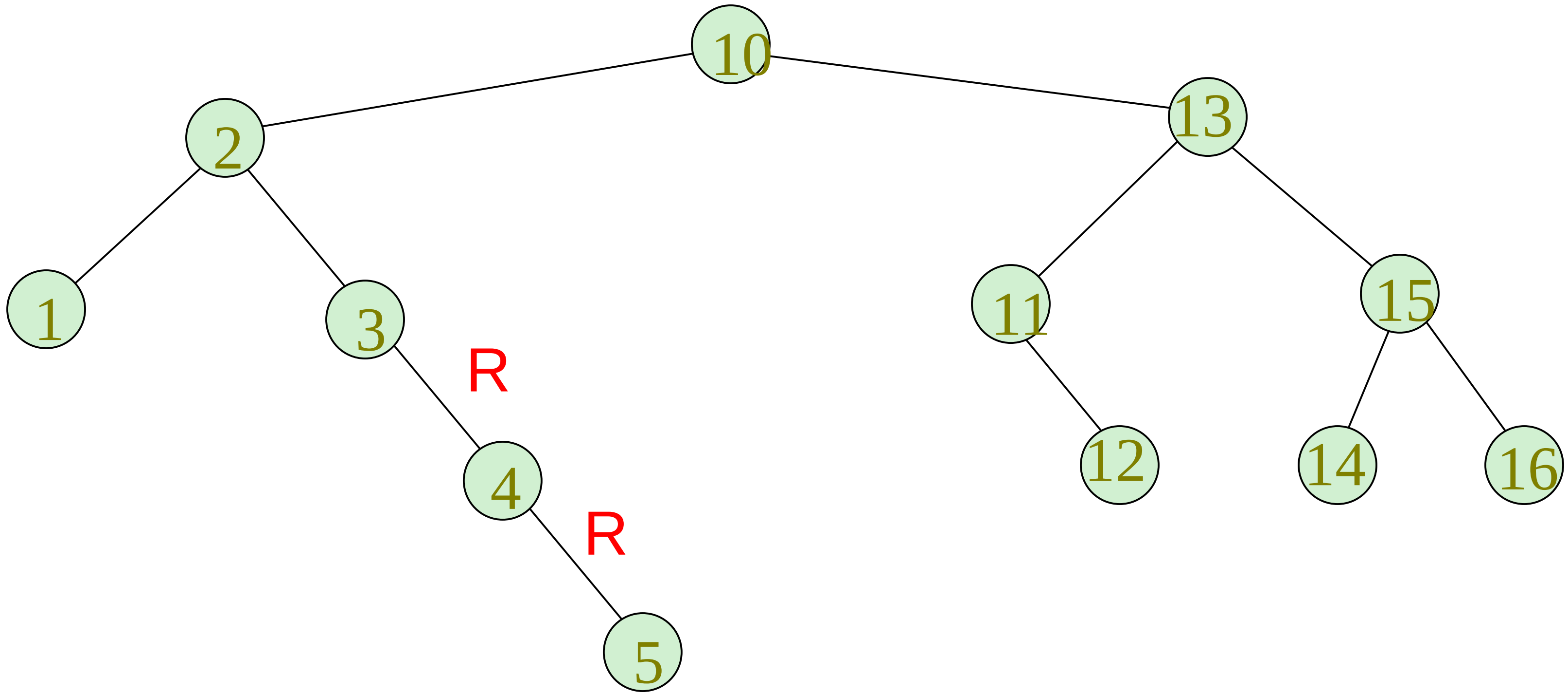
AVL Tree Rotations



- Now insert 5.

AVL Tree Rotations

Double rotations:

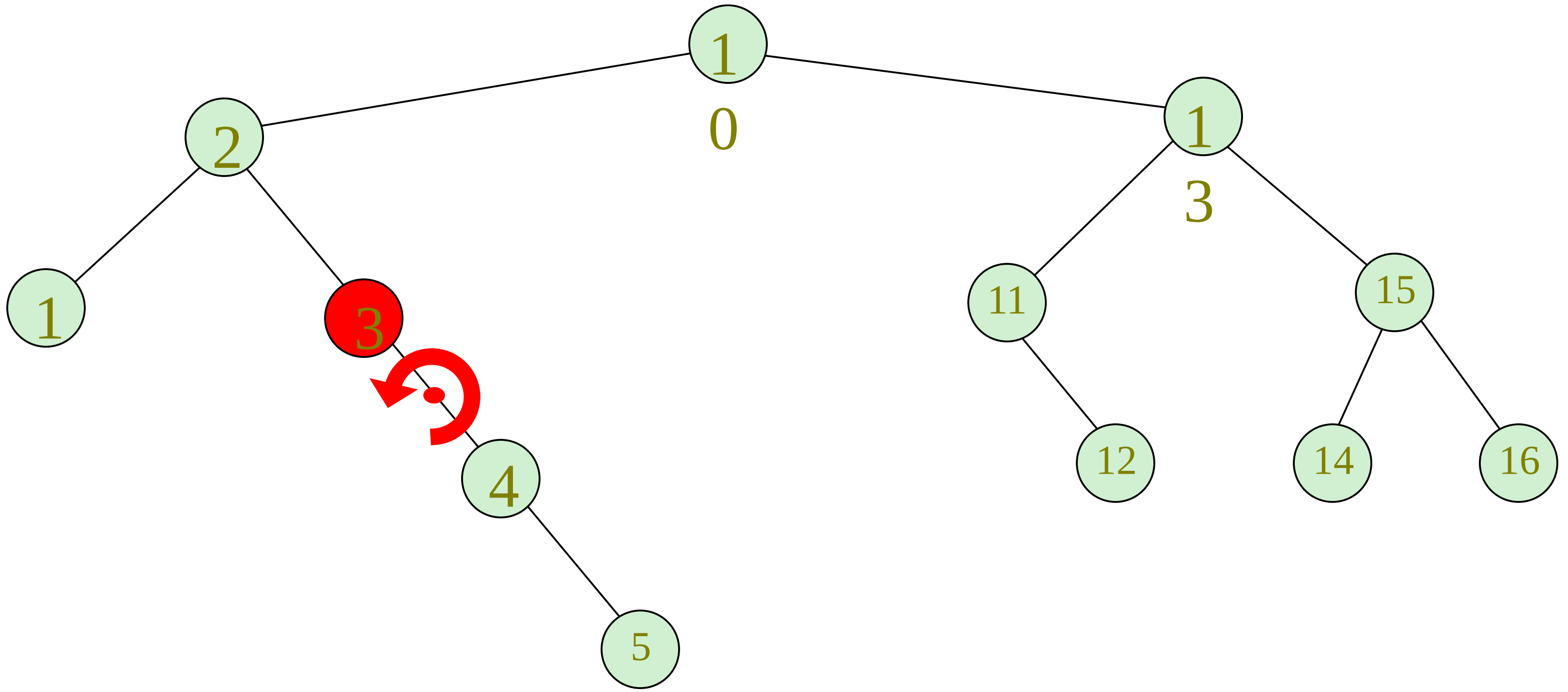


- AVL violation – rotate.

AVL Tree Rotations

Single rotations:

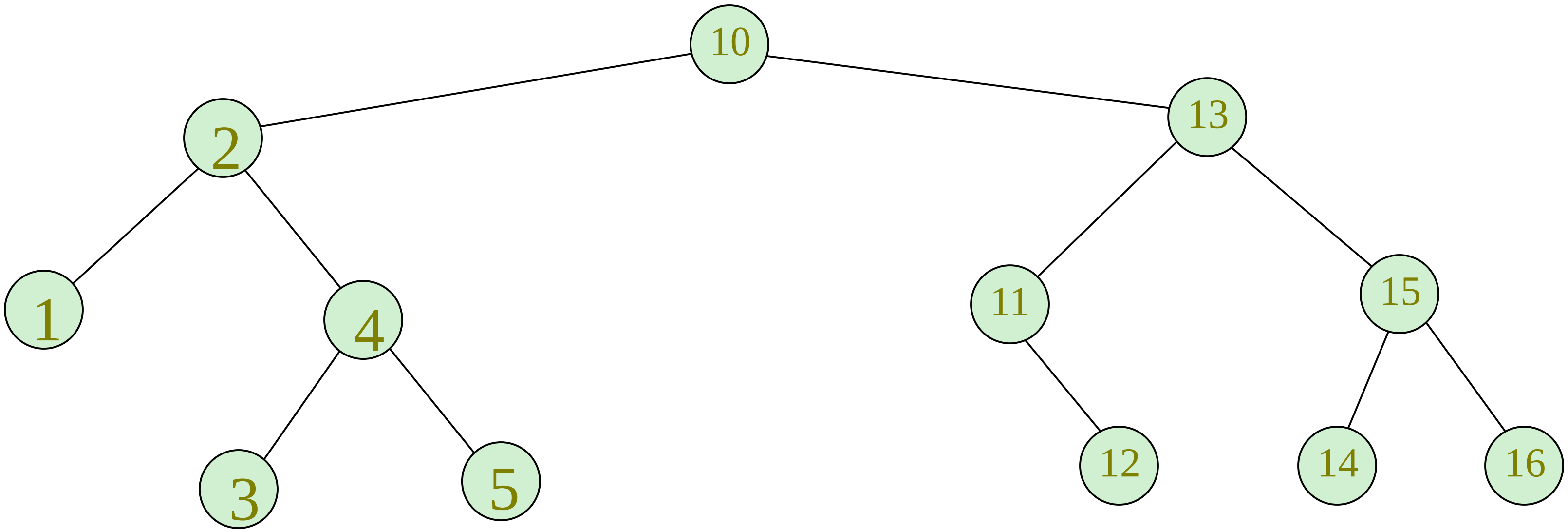
- Rotation type:



AVL Tree Rotations

Single rotations:

- AVL balance restored:

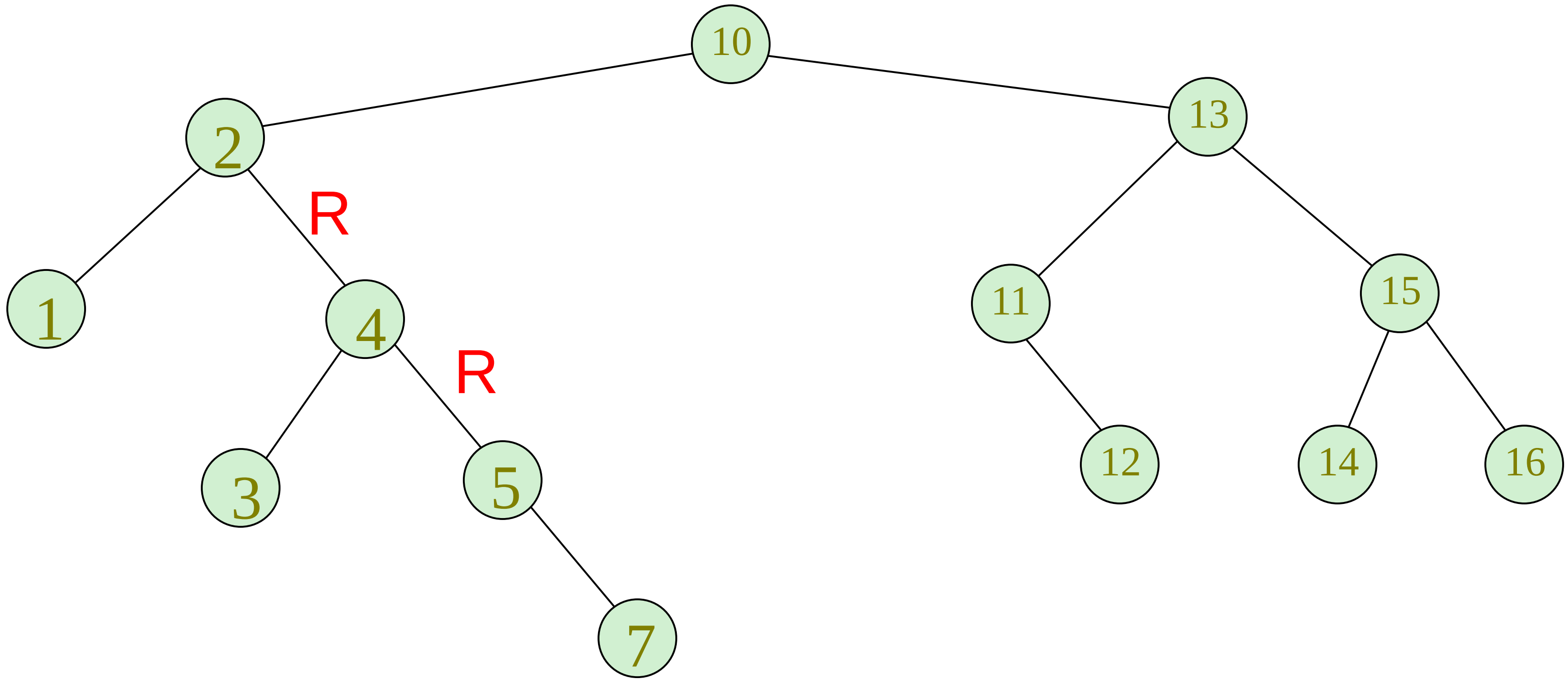


- Now insert 7.

AVL Tree Rotations

Single rotations:

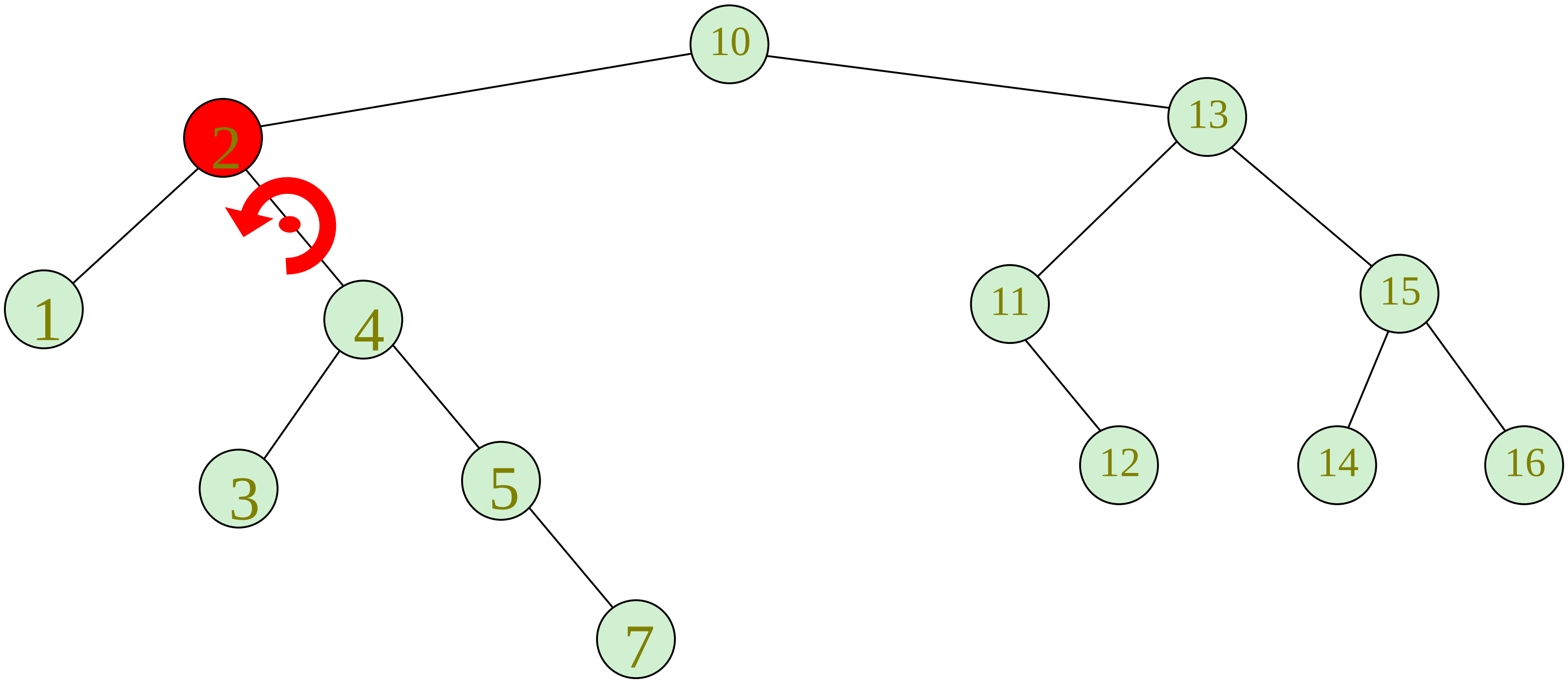
- AVL violation – rotate.



AVL Tree Rotations

Single rotations:

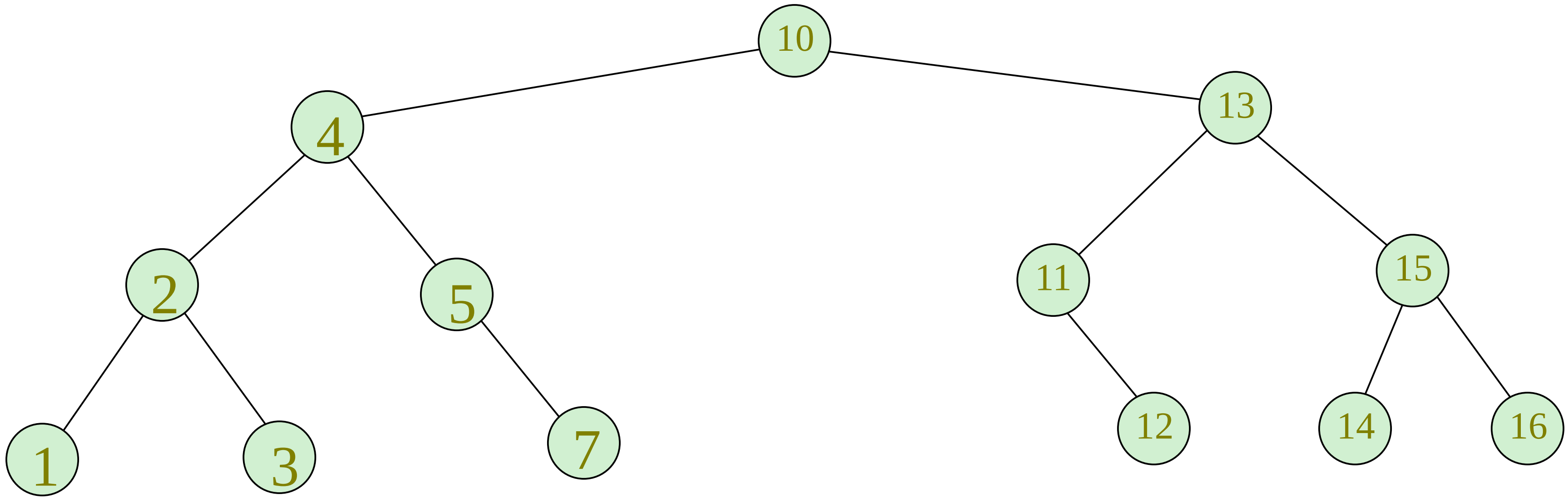
- Rotation type:



AVL Tree Rotations

Double rotations:

- AVL balance restored.

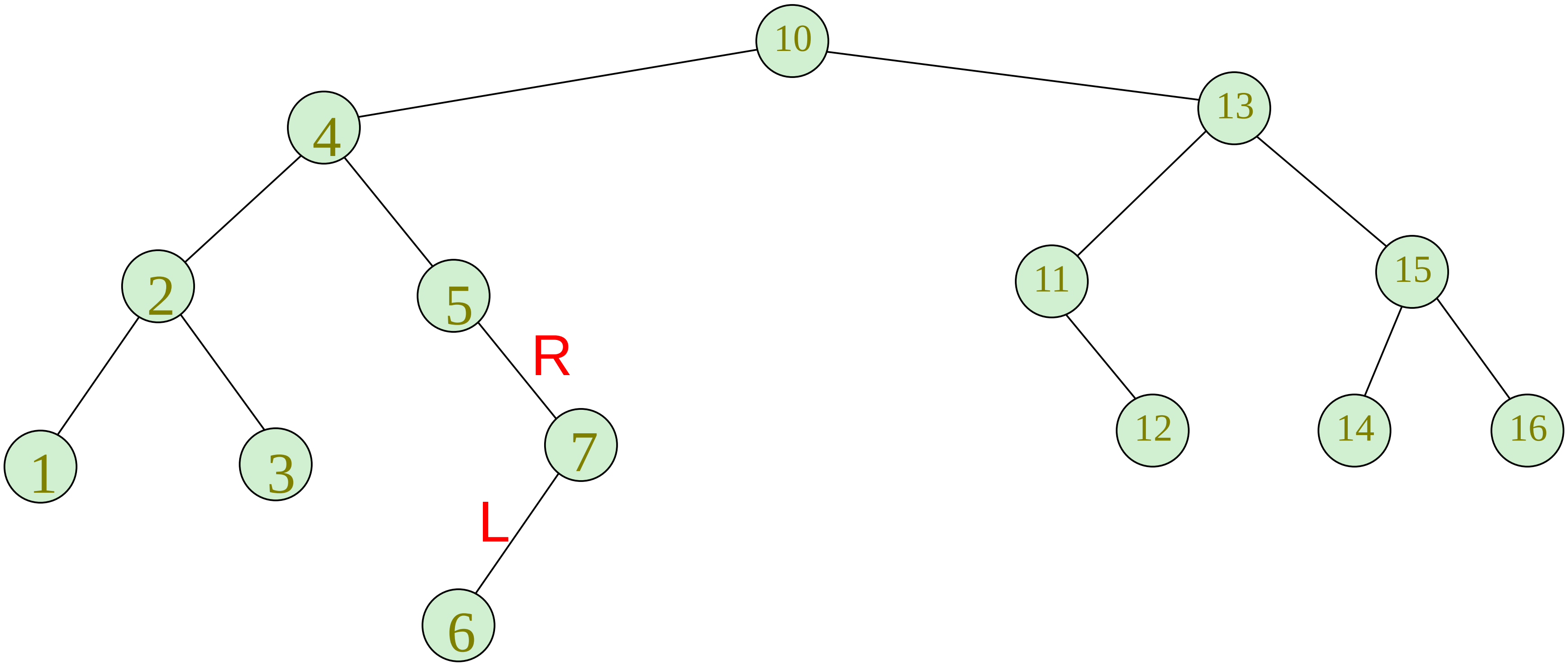


- Now insert 6.

AVL Tree Rotations

Double rotations:

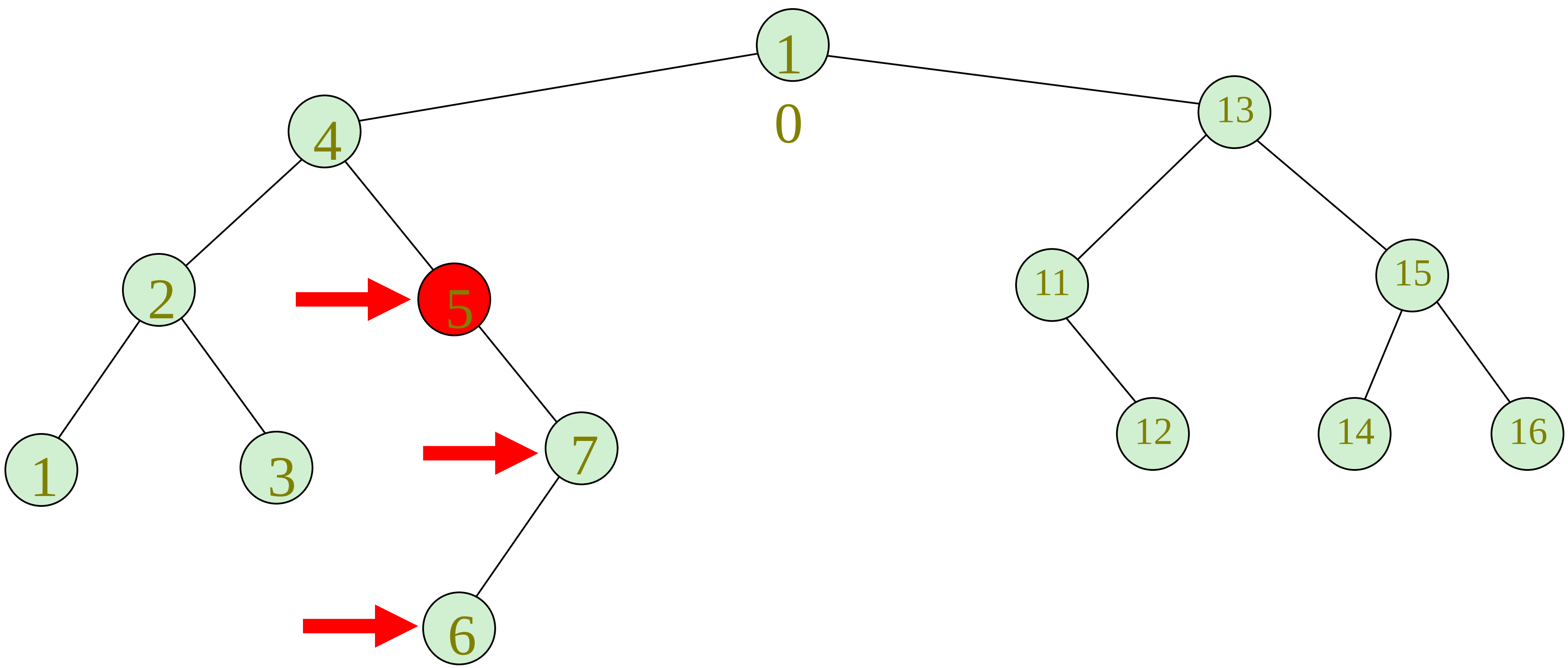
- AVL violation - rotate.



AVL Tree Rotations

Double rotations:

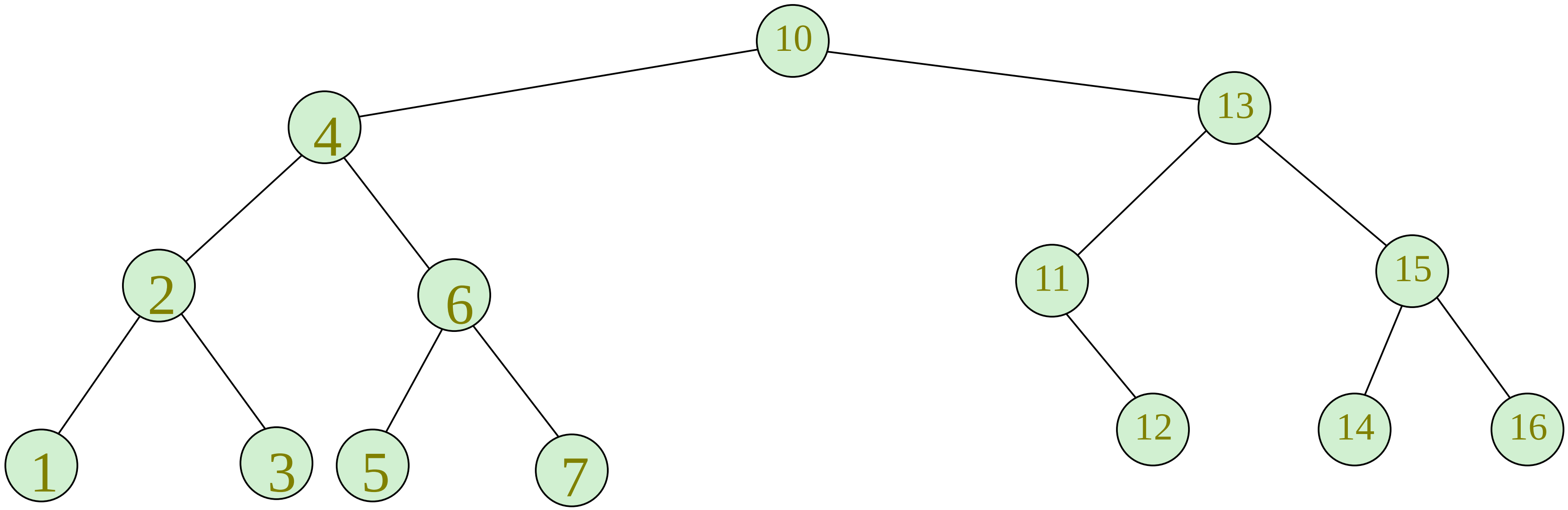
- Rotation type:



AVL Tree Rotations

Double rotations:

- AVL balance restored.

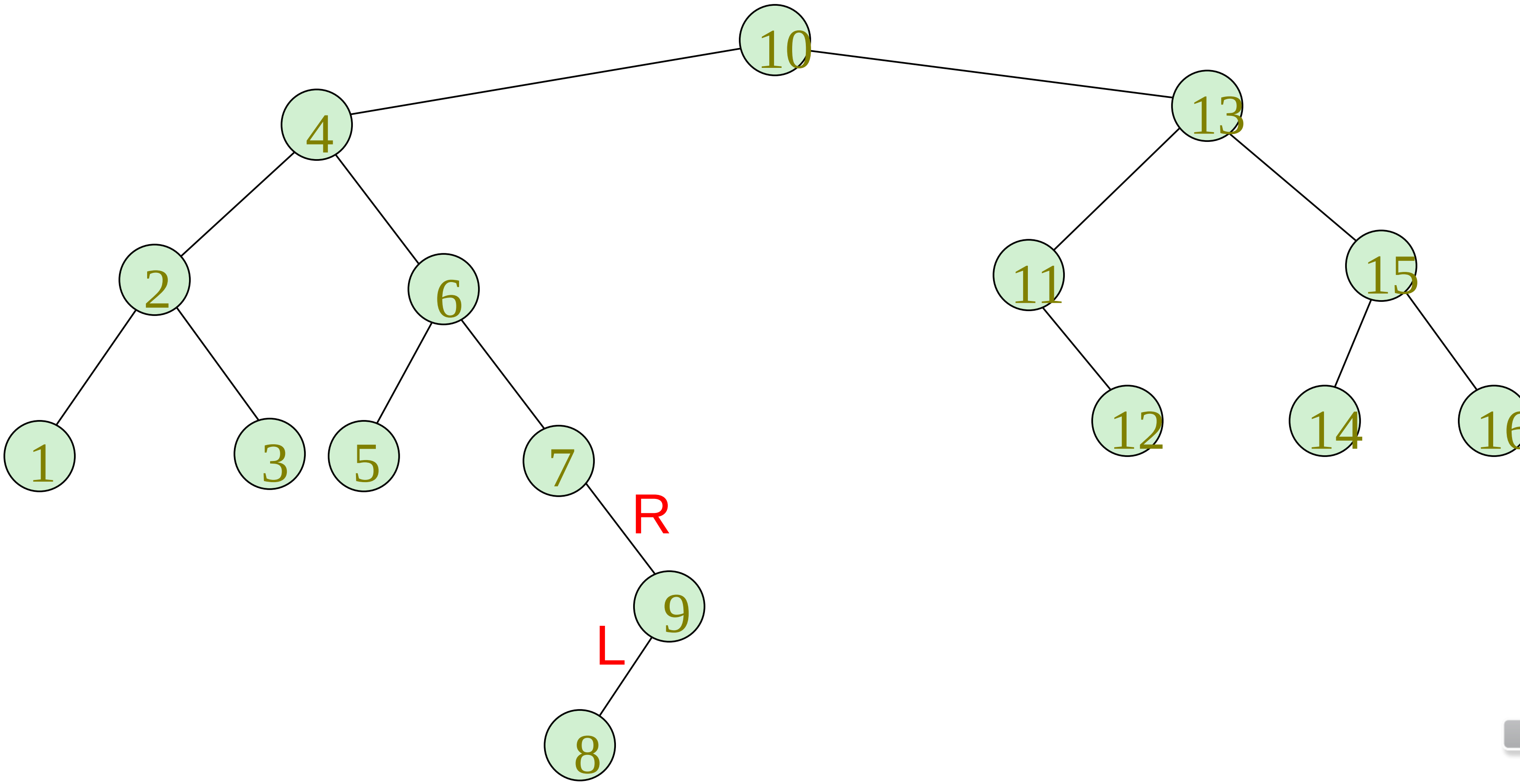


- Now insert 9 and 8.

AVL Tree Rotations

Double rotations:

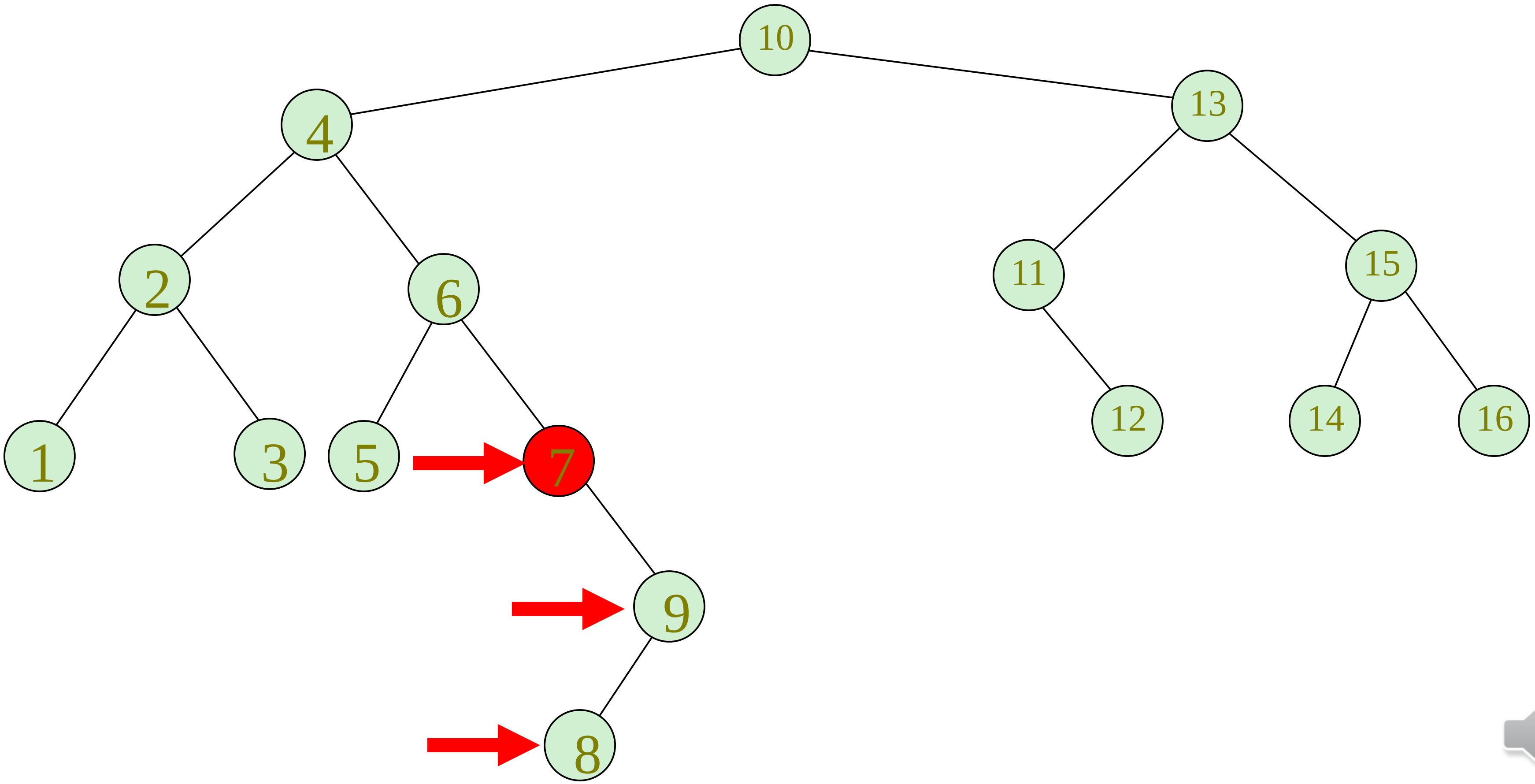
- AVL violation - rotate.



AVL Tree Rotations

Double rotations:

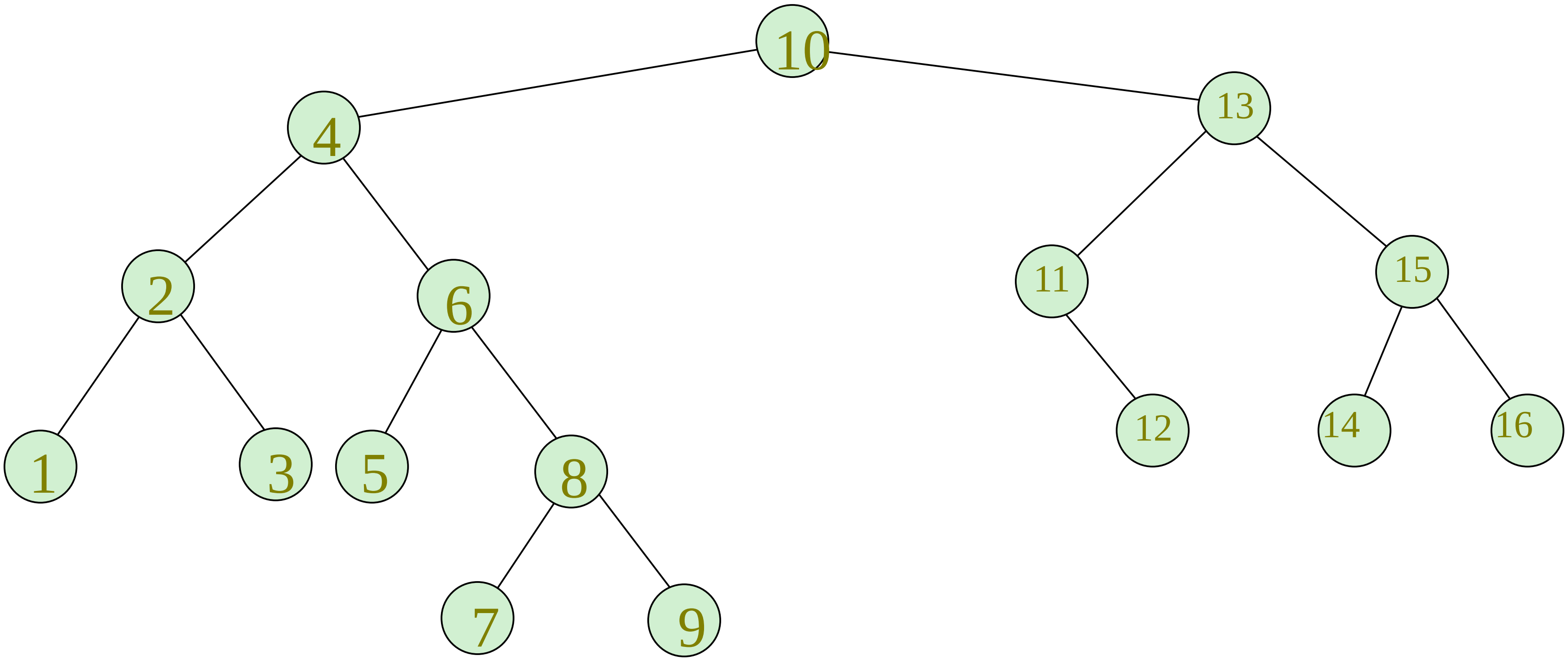
- Rotation type:



AVL Tree Rotations

Final tree:

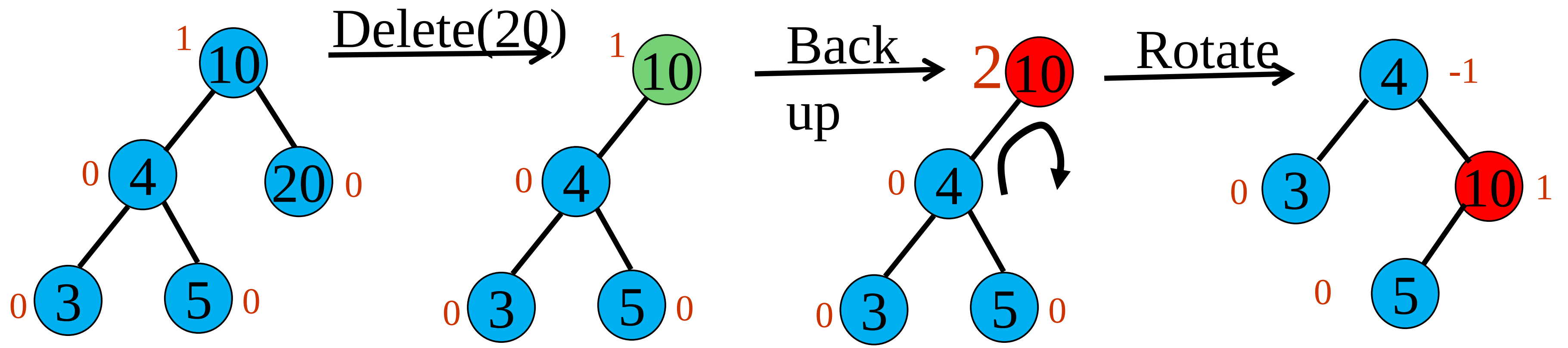
- Tree is almost perfectly balanced



Deletion(삭제)

- Deletion is similar to insertion
- First do regular BST deletion keeping track of the nodes on the path to the deleted node
- After the node is deleted, simply backup the tree(이진트리재구성) and update balance factors
  - If an imbalance is detected, do the appropriate rotation to restore the AVL tree property
  - You may have to do more than one rotation as you backup the tree

Deletion Example (1)



Initial AVLTree

Tree after deletion of 20

Idenfitied 10 as the pivot

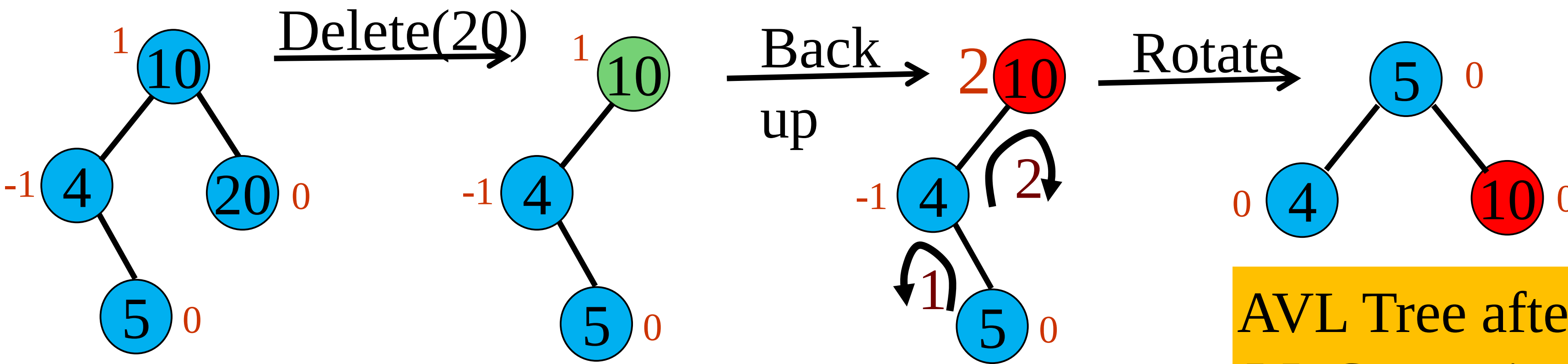
AVL Tree after LL Correction

Now, backup the tree updating balance factors

Classify the type of imbalance

- LL Imbalance:
  - bf of P(10) is 2
  - bf of L(4) is 0 or 1

Deletion Example (2)



Initial AVLTree

Tree after deletion of 20

Now, backup the tree updating balance factors

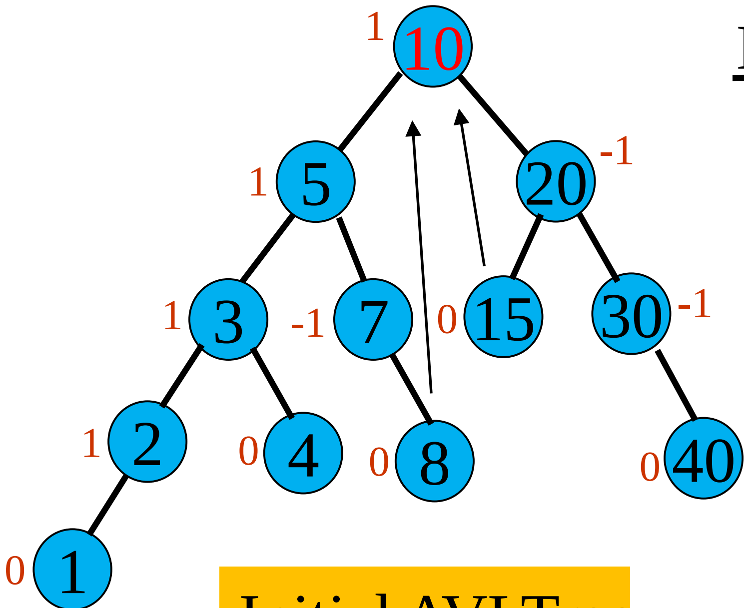
Idenfitied 10 as the pivot

Classify the type of imbalance

- LR Imbalance:
  - bf of P(10) is 2
  - bf of L(4) is -1

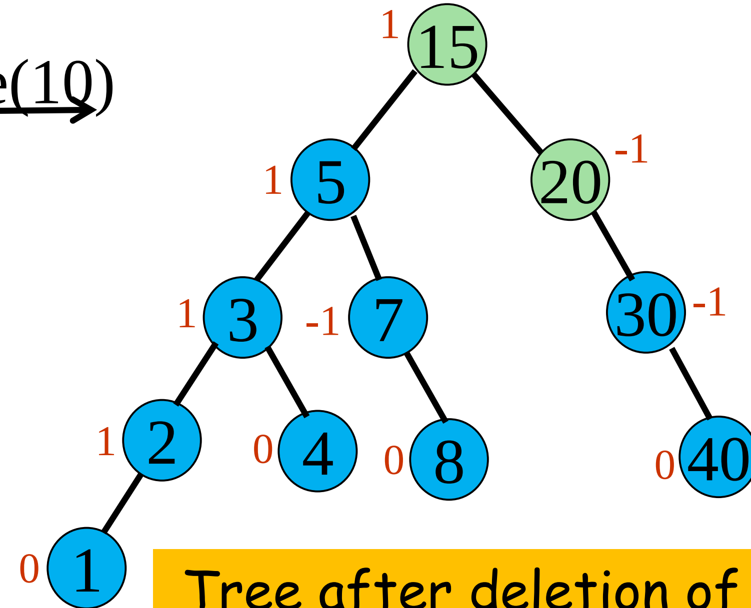
AVL Tree after LL Correction

Deletion Example (3)



Initial AVLTree

Delete(10)

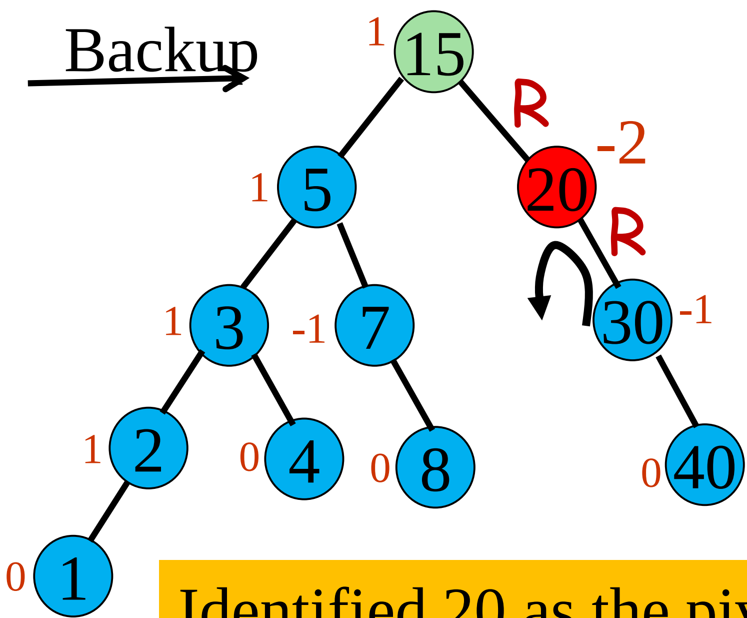


Tree after deletion of 10

We have copied the successor of 10 to root and deleted 15

Now, backup the tree updating balance factors

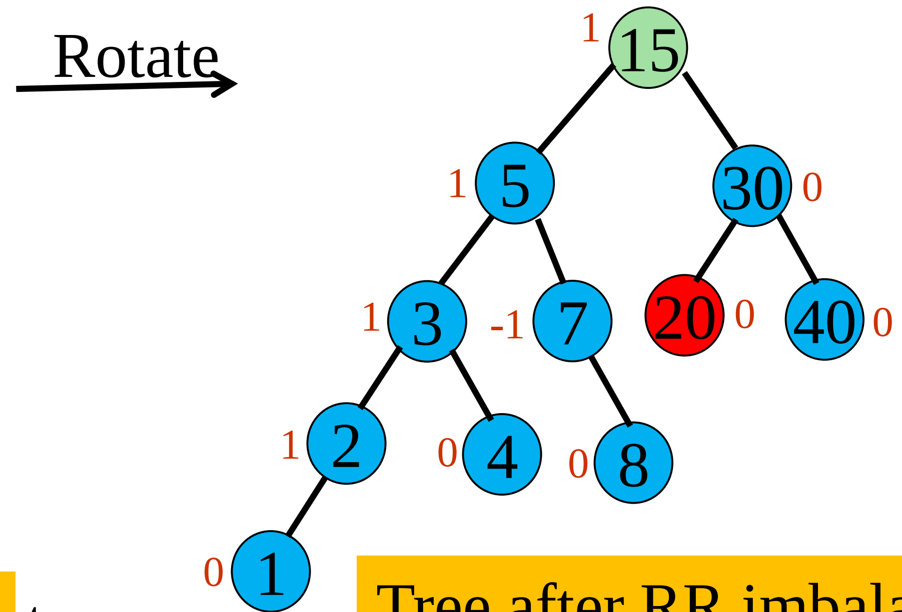
Deletion Example (3) - continued



Identified 20 as the pivot

Classify the type of imbalance

- RR Imbalance:
  - bf of P(20) is -2
  - bf of R(30) is 0 or -1



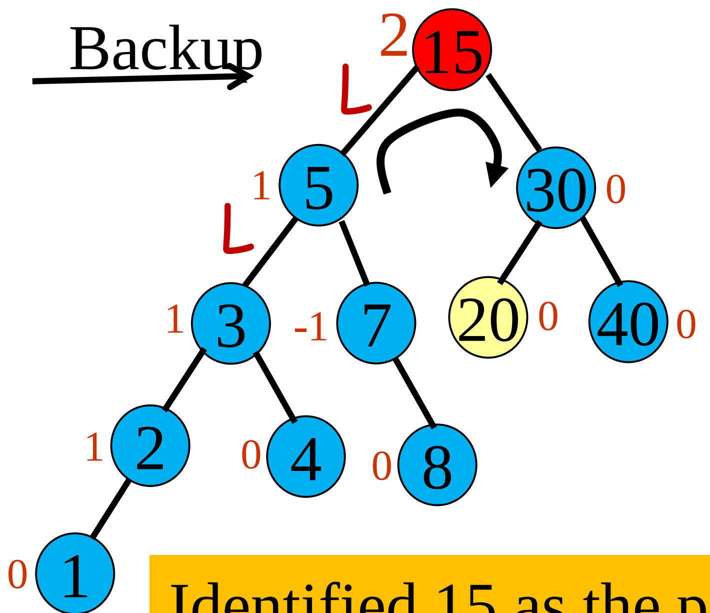
Tree after RR imbalance is corrected

Is this an AVL tree?

Continue backing up the tree updating balance factors

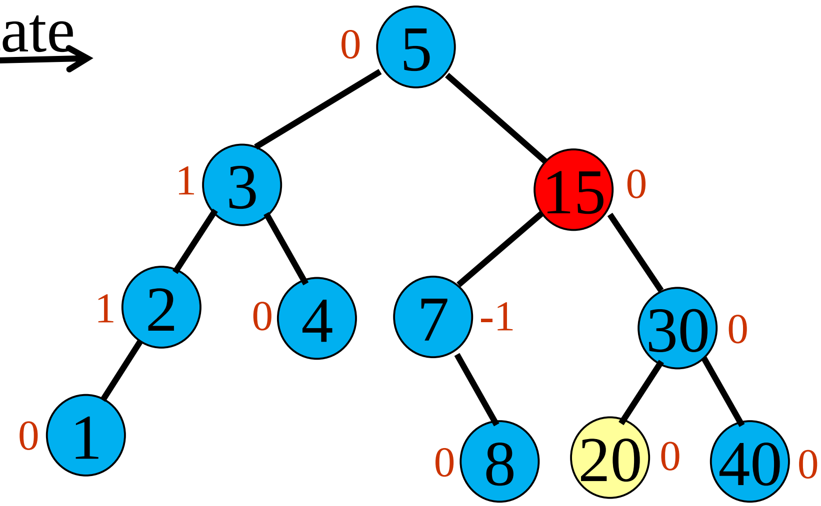
Deletion Example (3) - continued

Backup



Identified 15 as the pivot

Rotate



Final Tree

Classify the type of imbalance

- LL Imbalance:
  - bf of P(15) is 2
  - bf of L(5) is 0 or 1

Example 2

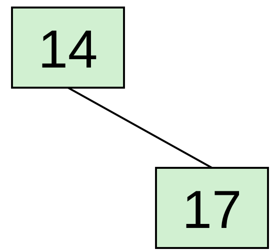
AVL Tree Example:

- **Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree**

14

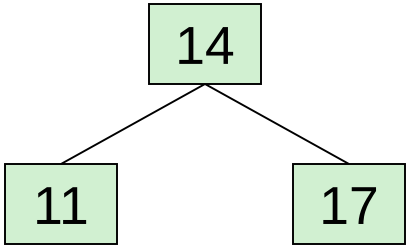
AVL Tree Example:

- **Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree**



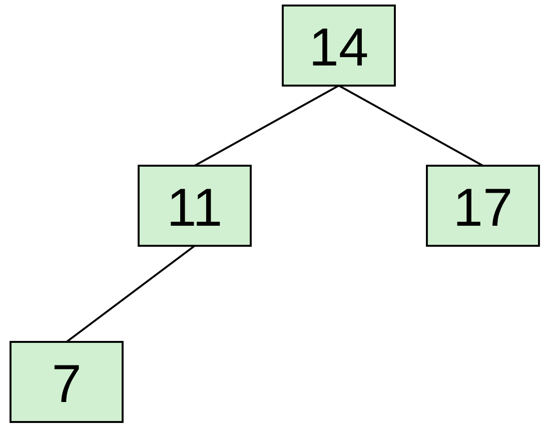
AVL Tree Example:

- Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree



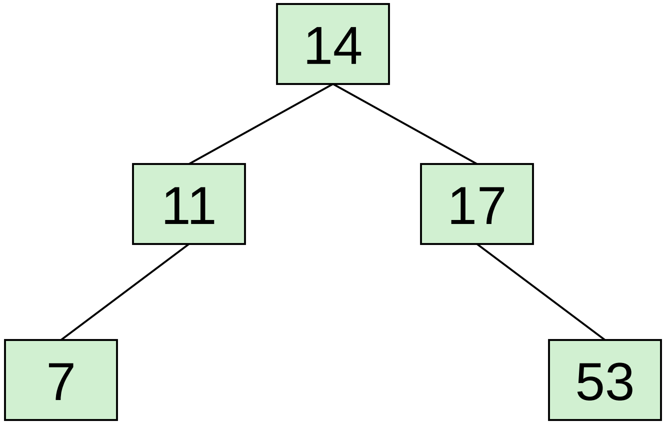
AVL Tree Example:

- Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree



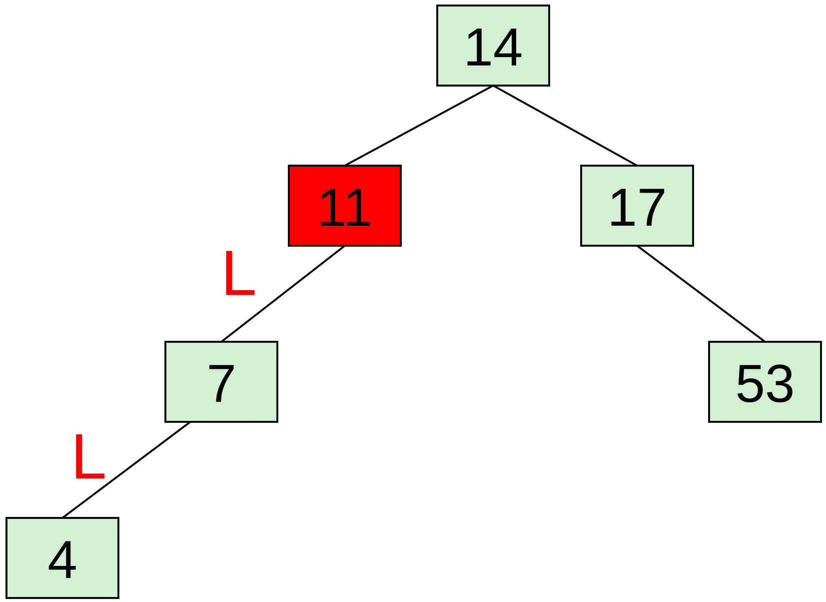
AVL Tree Example:

- **Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree**



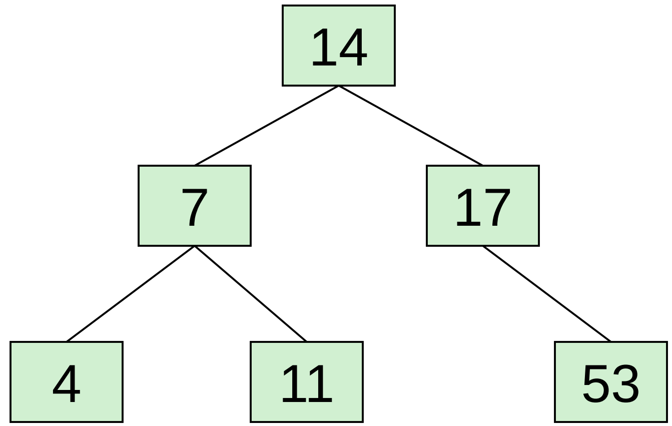
AVL Tree Example:

- **Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree**



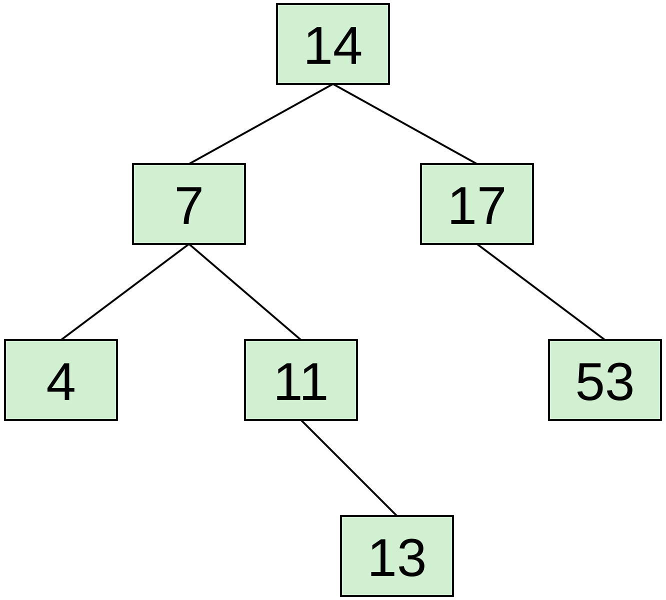
AVL Tree Example:

- **Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree**



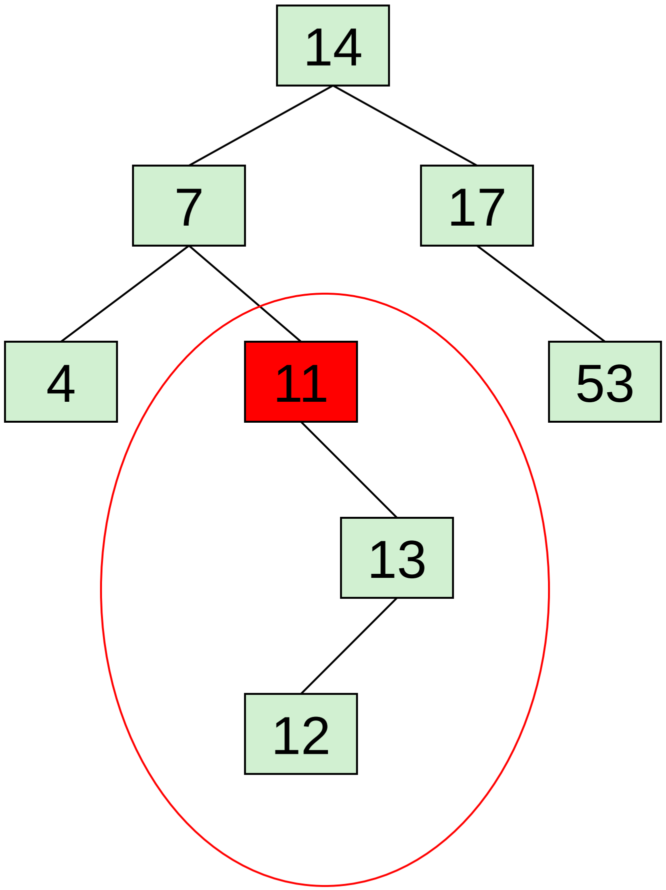
AVL Tree Example:

- Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree



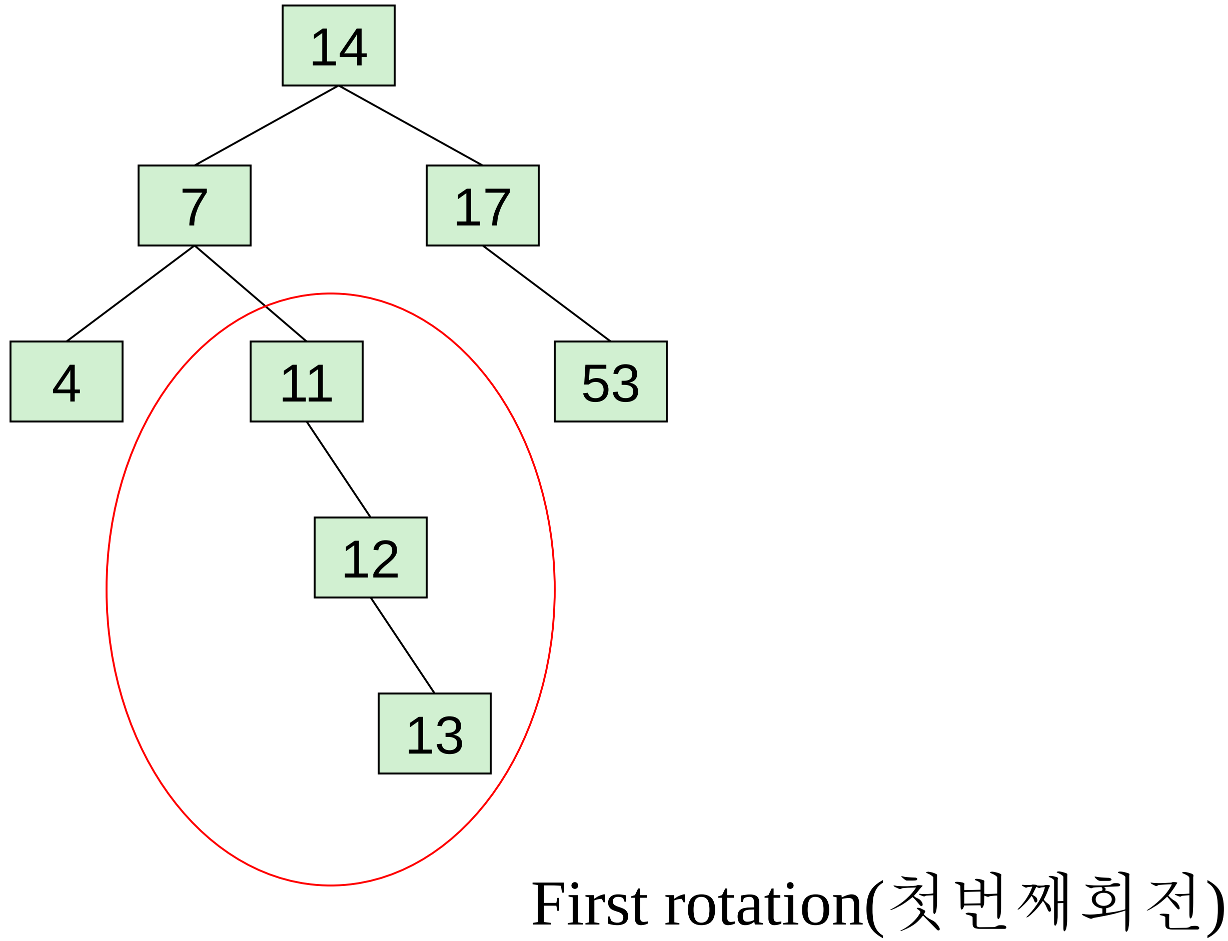
AVL Tree Example:

- **Now insert 12**



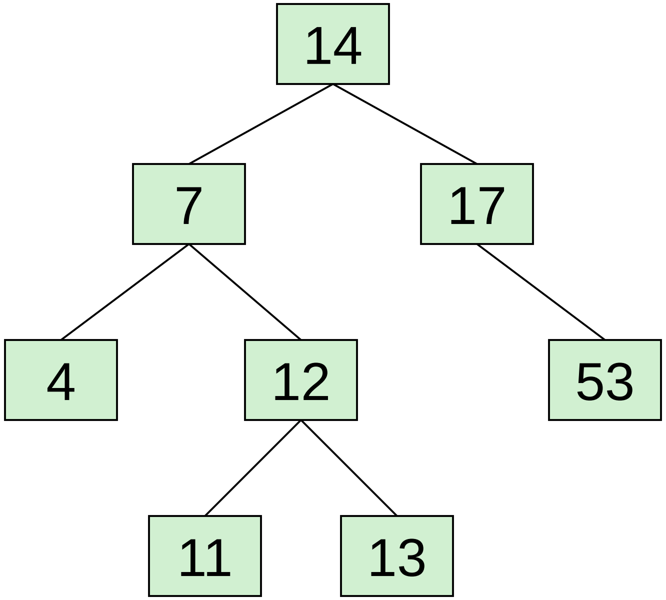
AVL Tree Example:

- **Now insert 12**



AVL Tree Example:

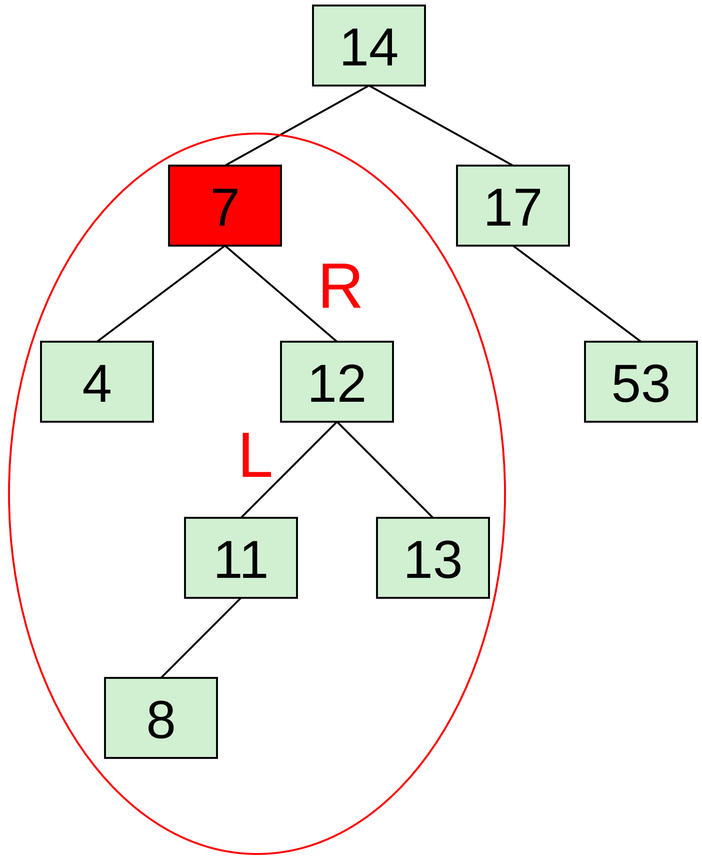
- **Now the AVL tree is balanced.**



Second rotation(두번째회전)

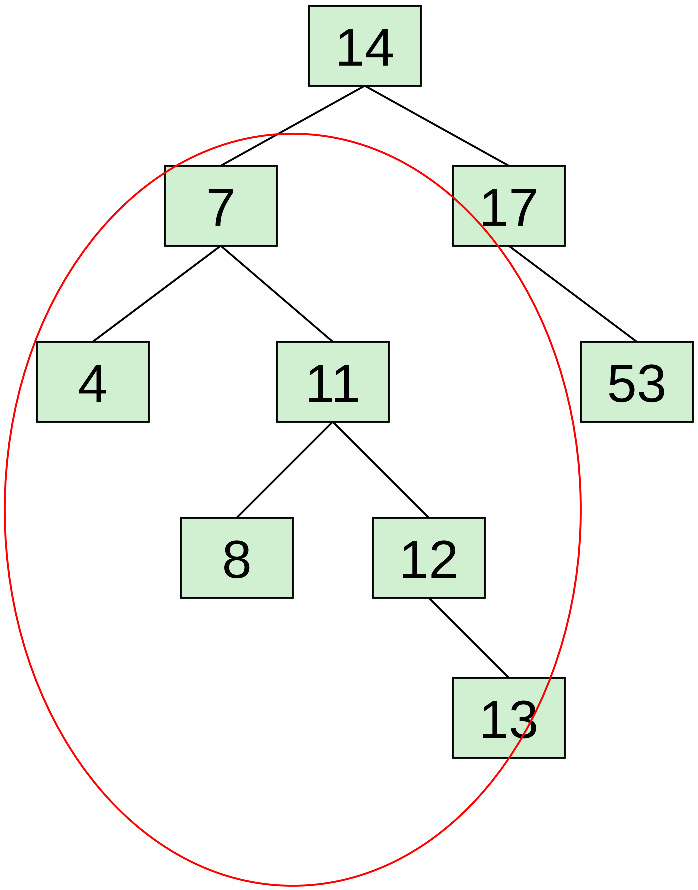
AVL Tree Example:

- **Now insert 8**



AVL Tree Example:

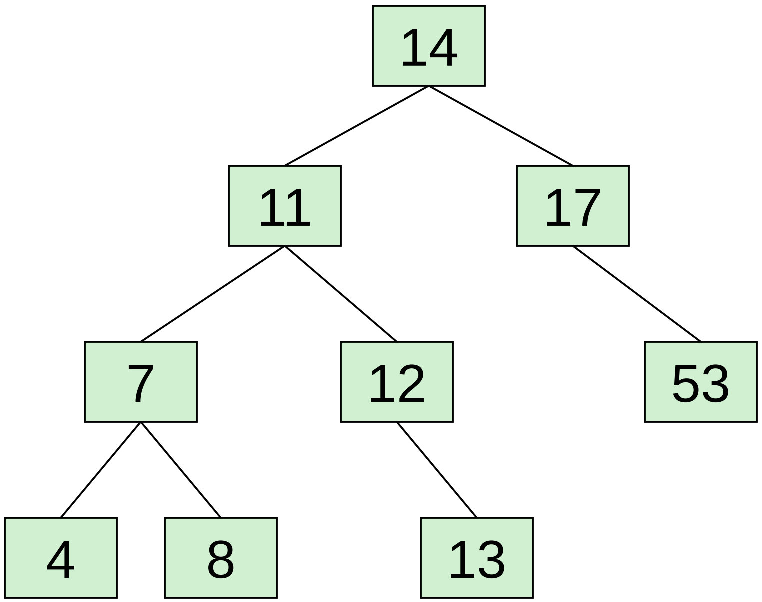
- **Now insert 8**



First rotation(첫번째회전)

AVL Tree Example:

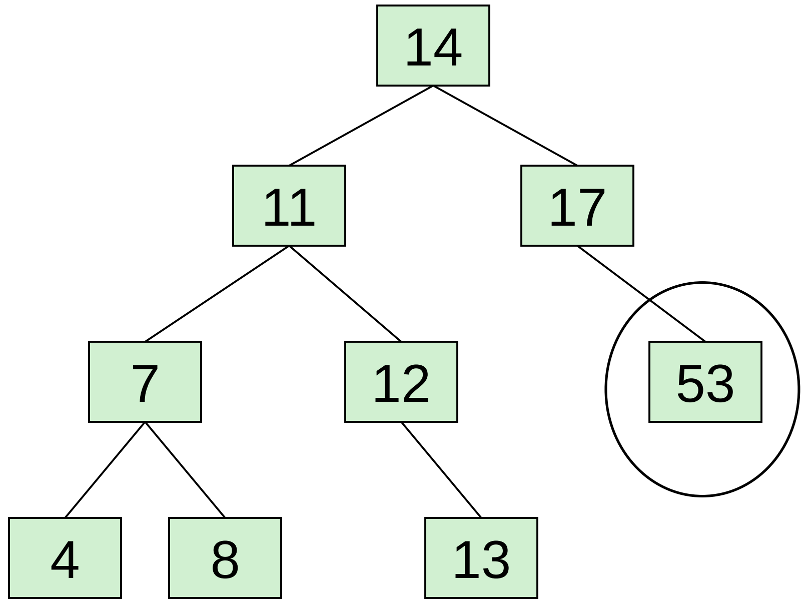
- **Now the AVL tree is balanced.**



Second rotation(두번째회전)

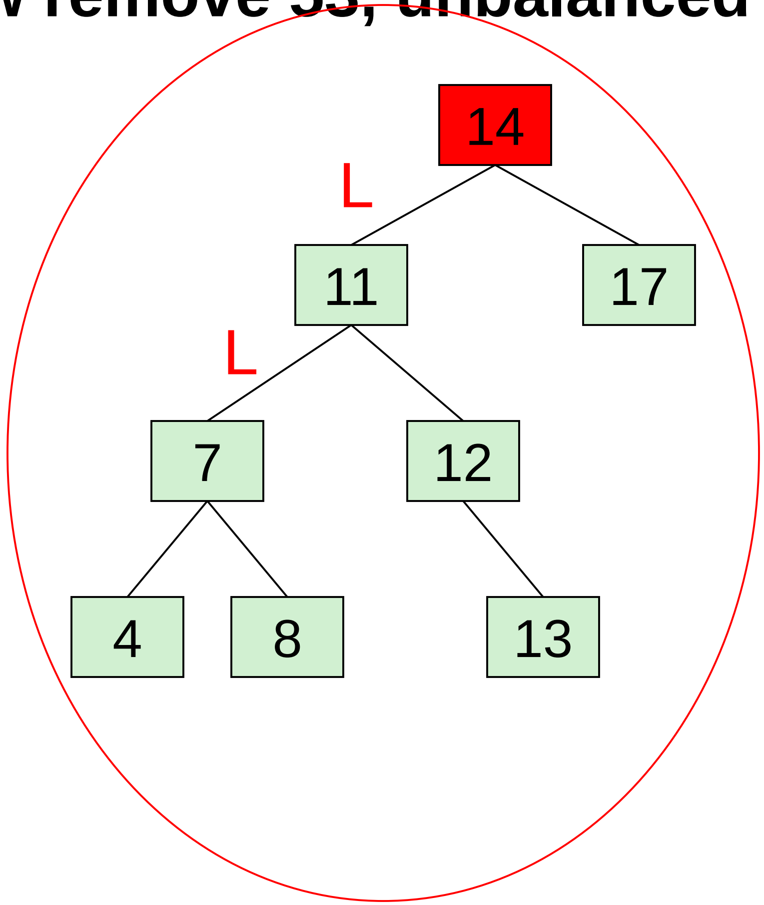
AVL Tree Example:

- **Now remove 53(53제거)**



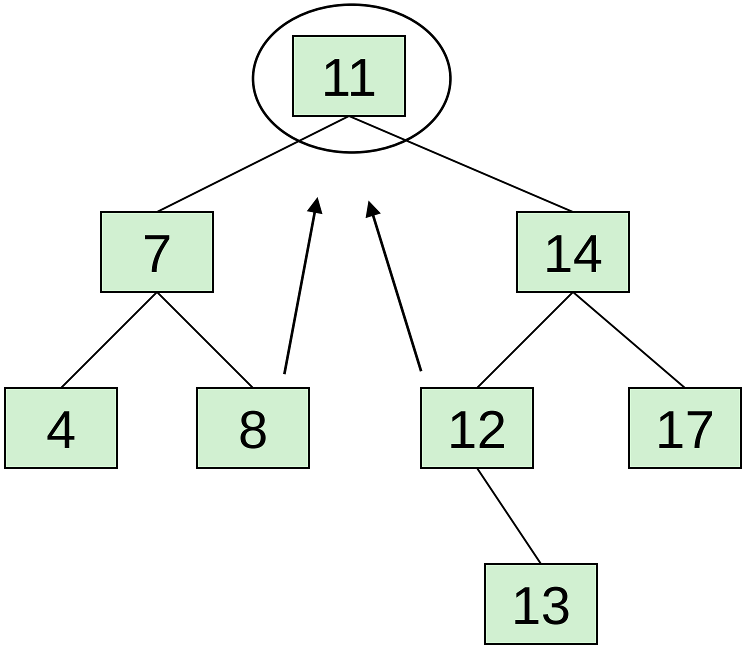
AVL Tree Example:

- **Now remove 53, unbalanced**



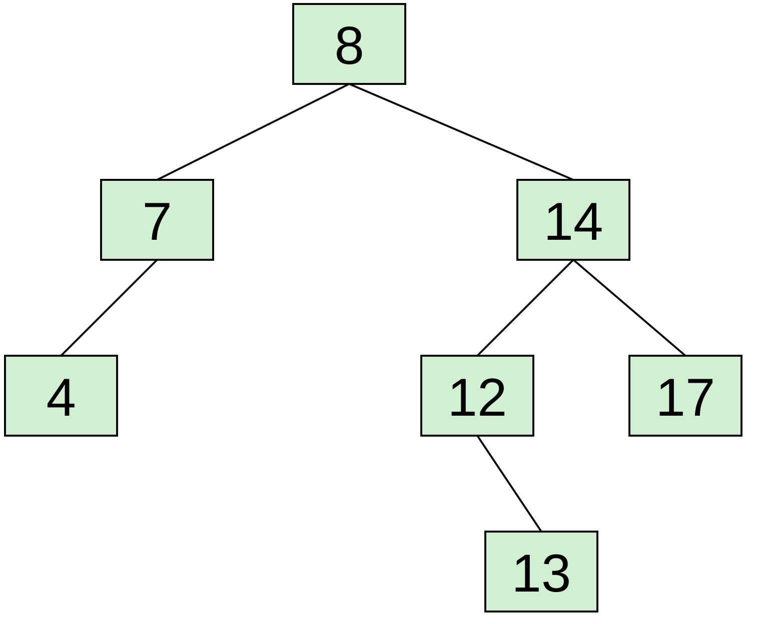
AVL Tree Example:

- **Balanced!**   **Remove 11**



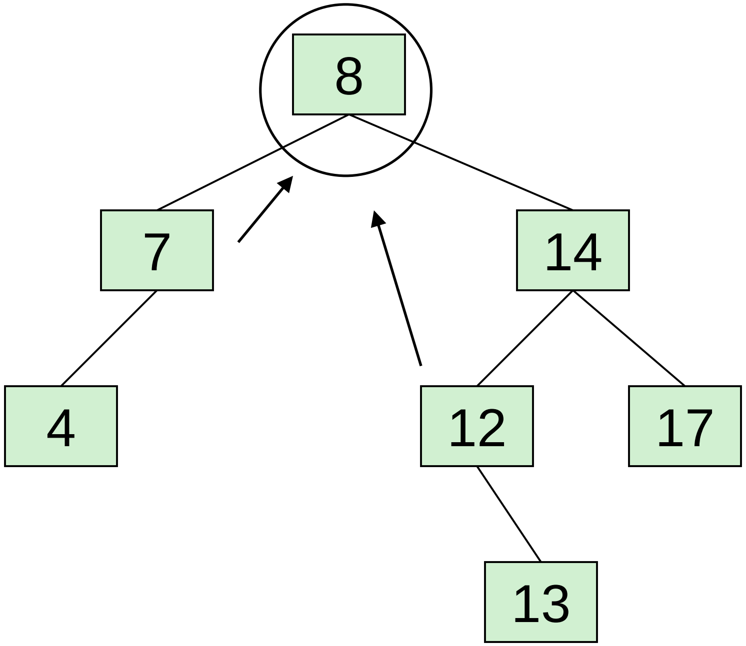
AVL Tree Example:

- **Remove 11, replace it with the largest in its left branch**



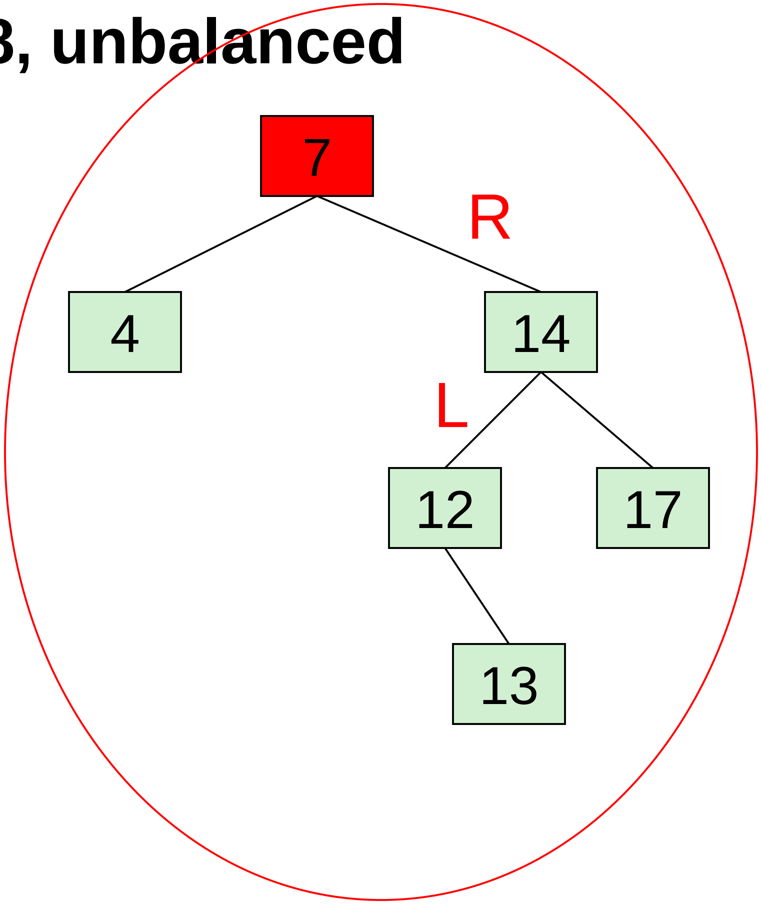
AVL Tree Example:

- **Remove 8, replace it with the largest in its left branch**



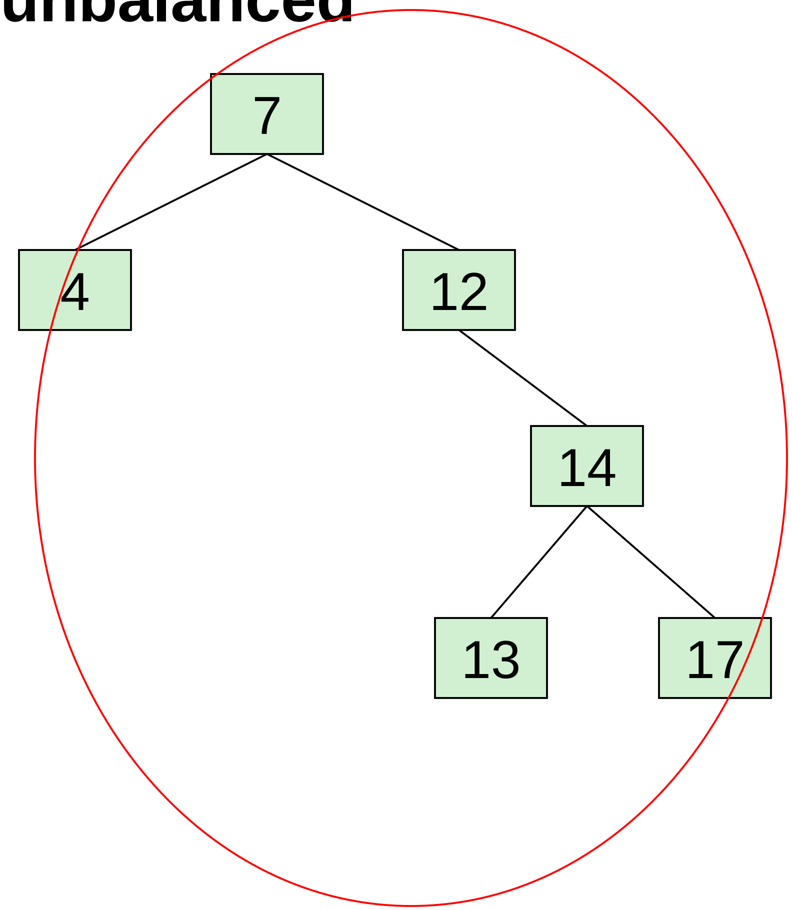
AVL Tree Example:

- **Remove 8, unbalanced**



AVL Tree Example:

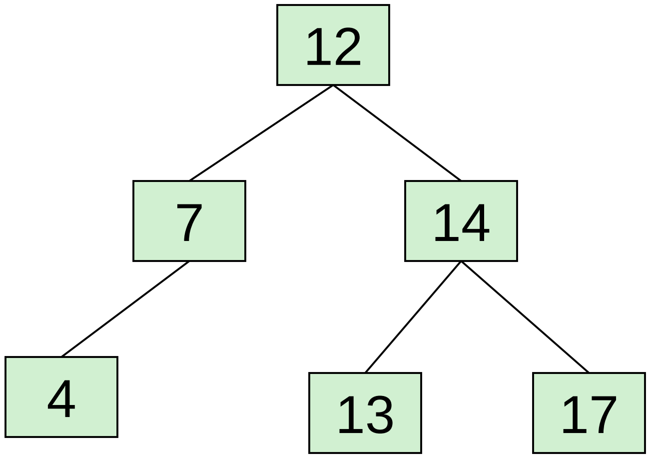
- **Remove 8, unbalanced**



First rotation(첫번째회전)

AVL Tree Example:

- **Balanced!!**



Second rotation(두번째회전)

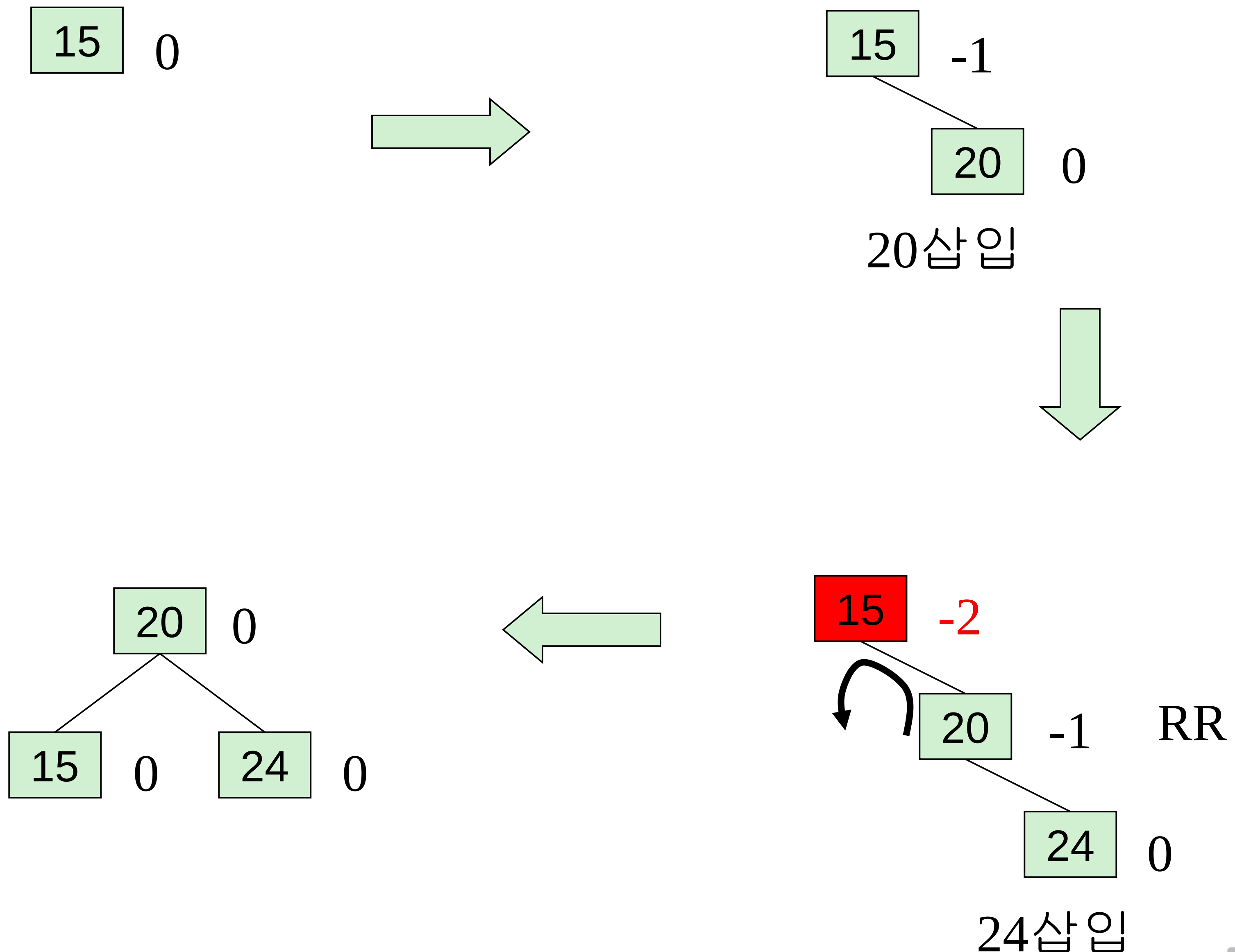
Example 3

Example

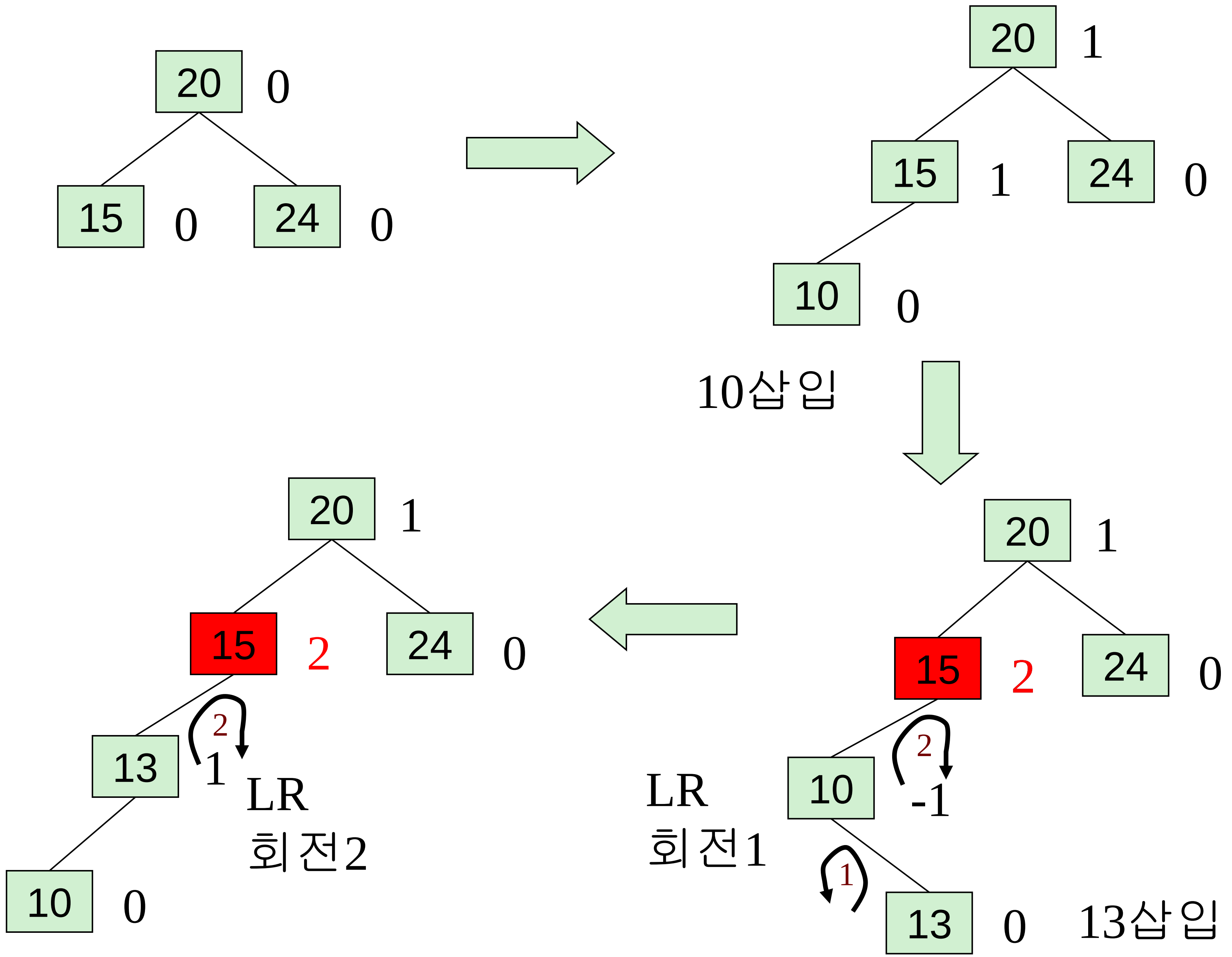
- Build an AVL tree with the following values:

  15, 20, 24, 10, 13, 7, 30, 36, 25

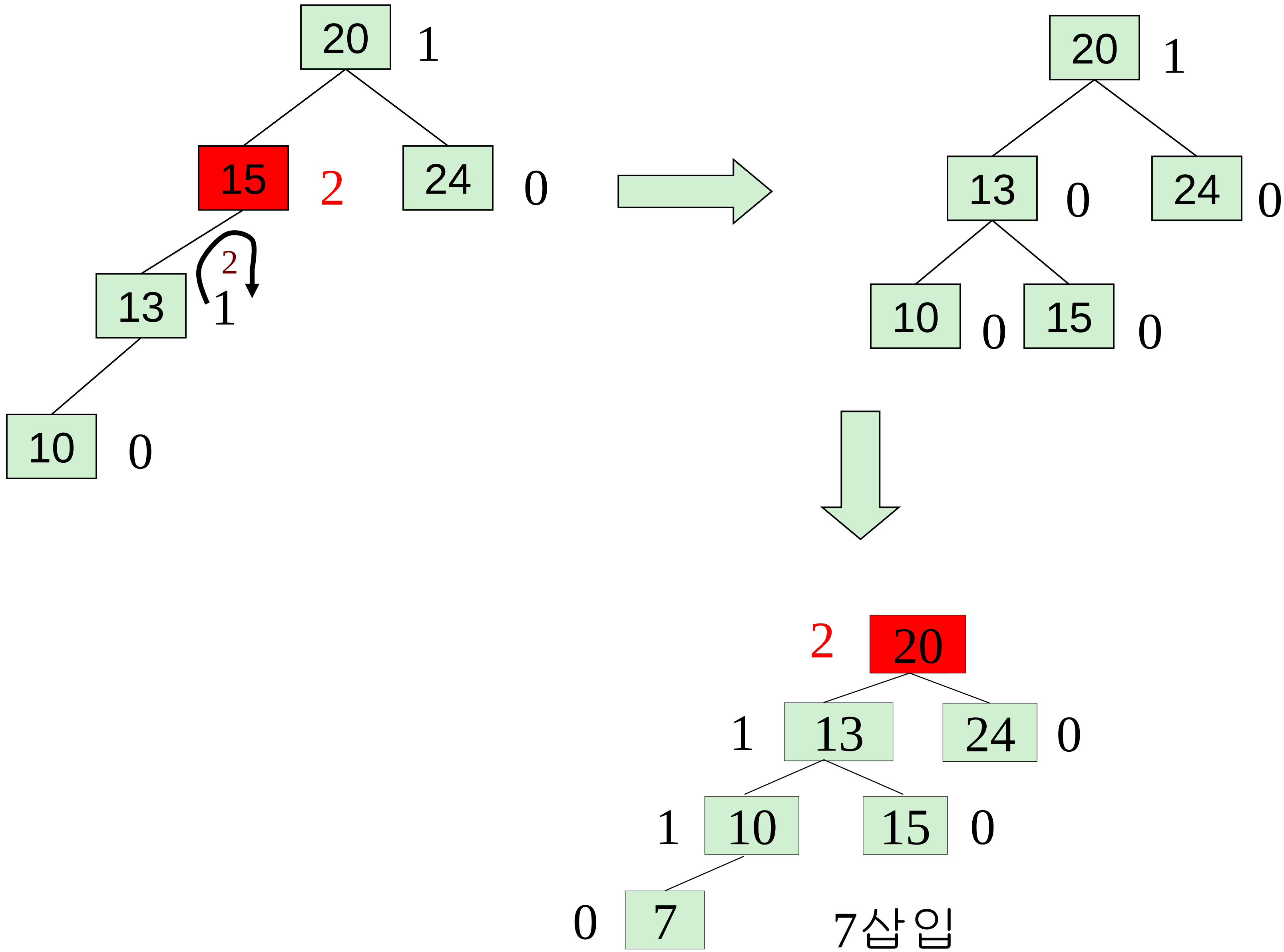
15, 20, 24, 10, 13, 7, 30, 36, 25



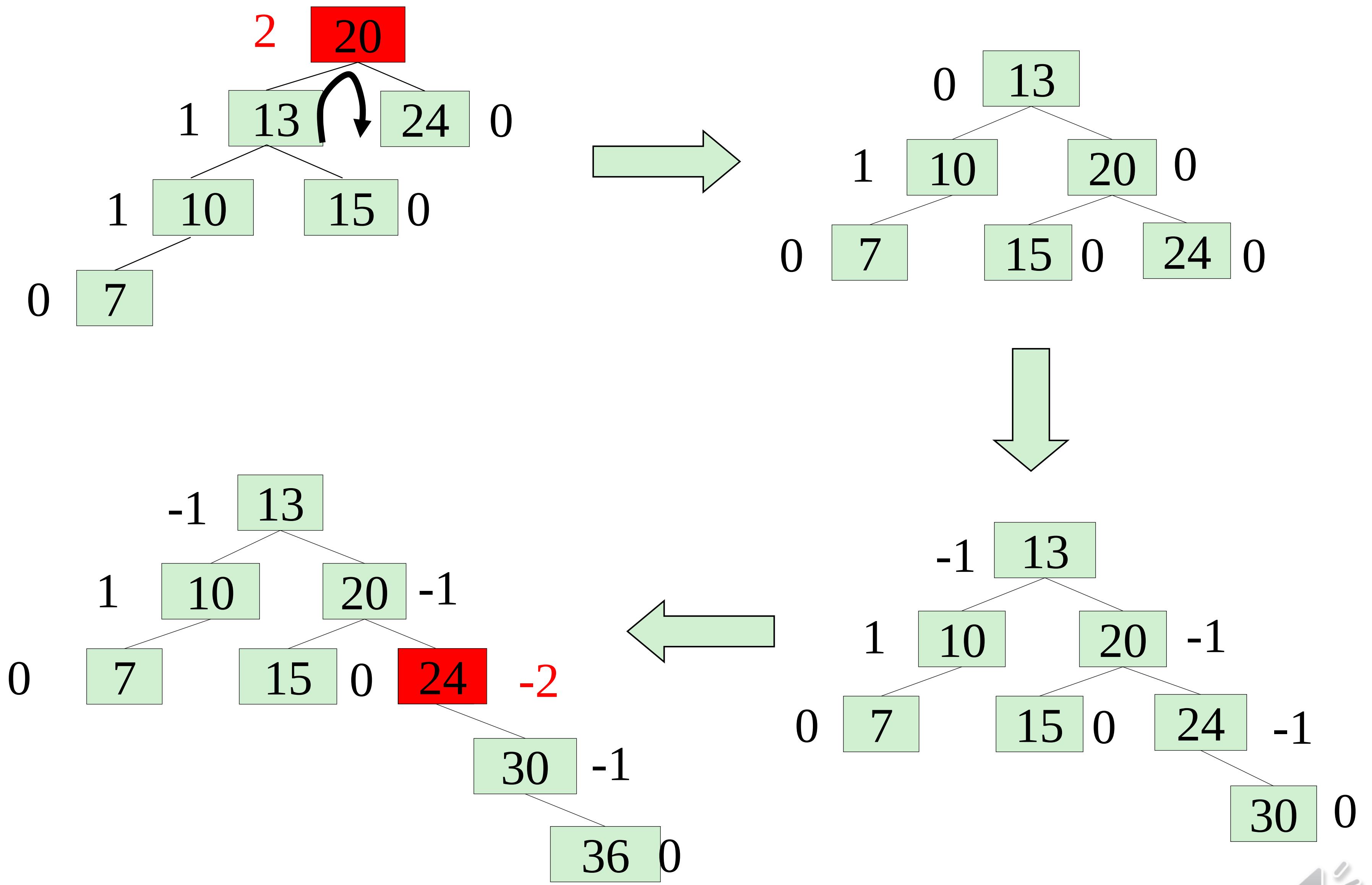
15, 20, 24, 10, 13, 7, 30, 36, 25



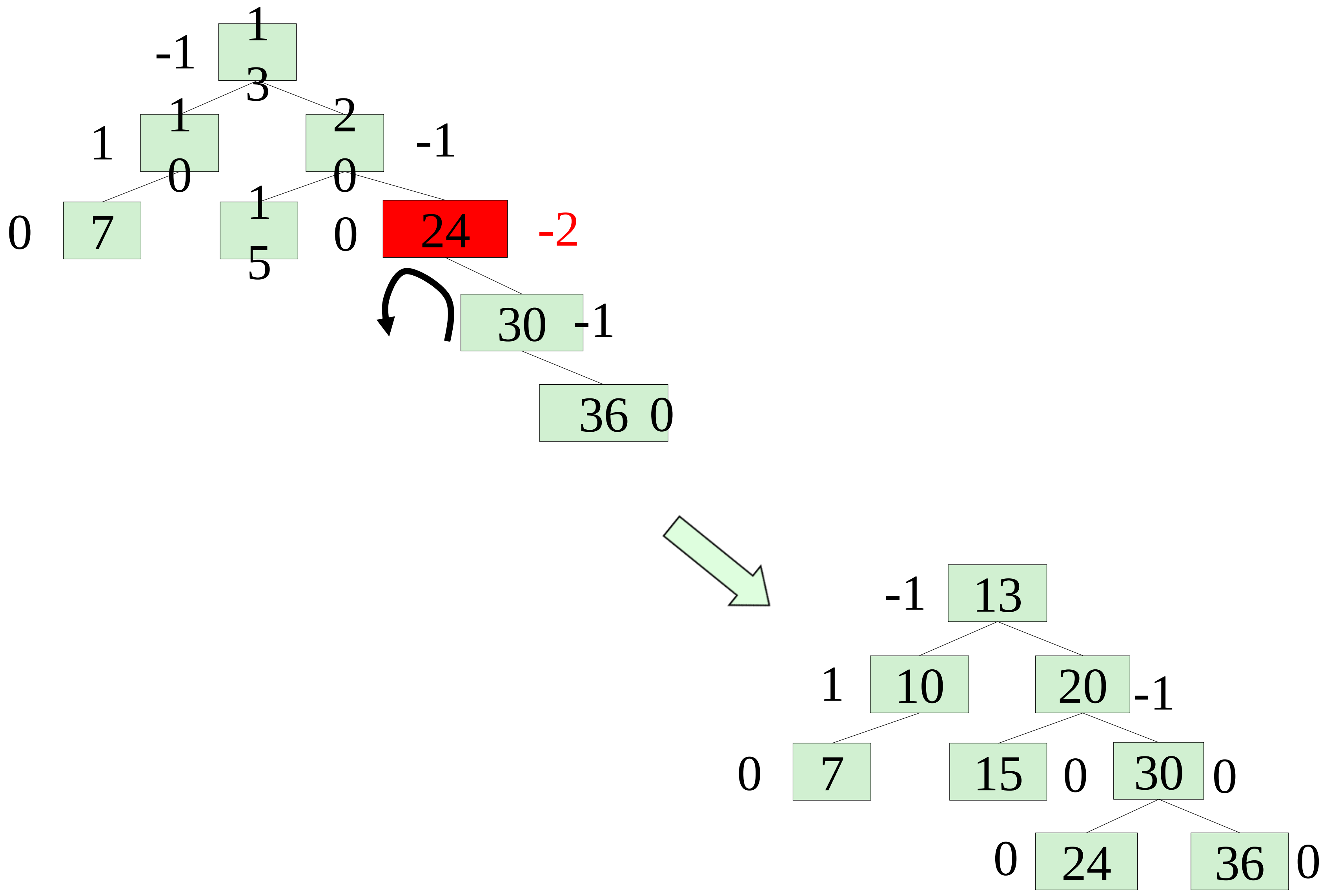
15, 20, 24, 10, 13, 7, 30, 36, 25



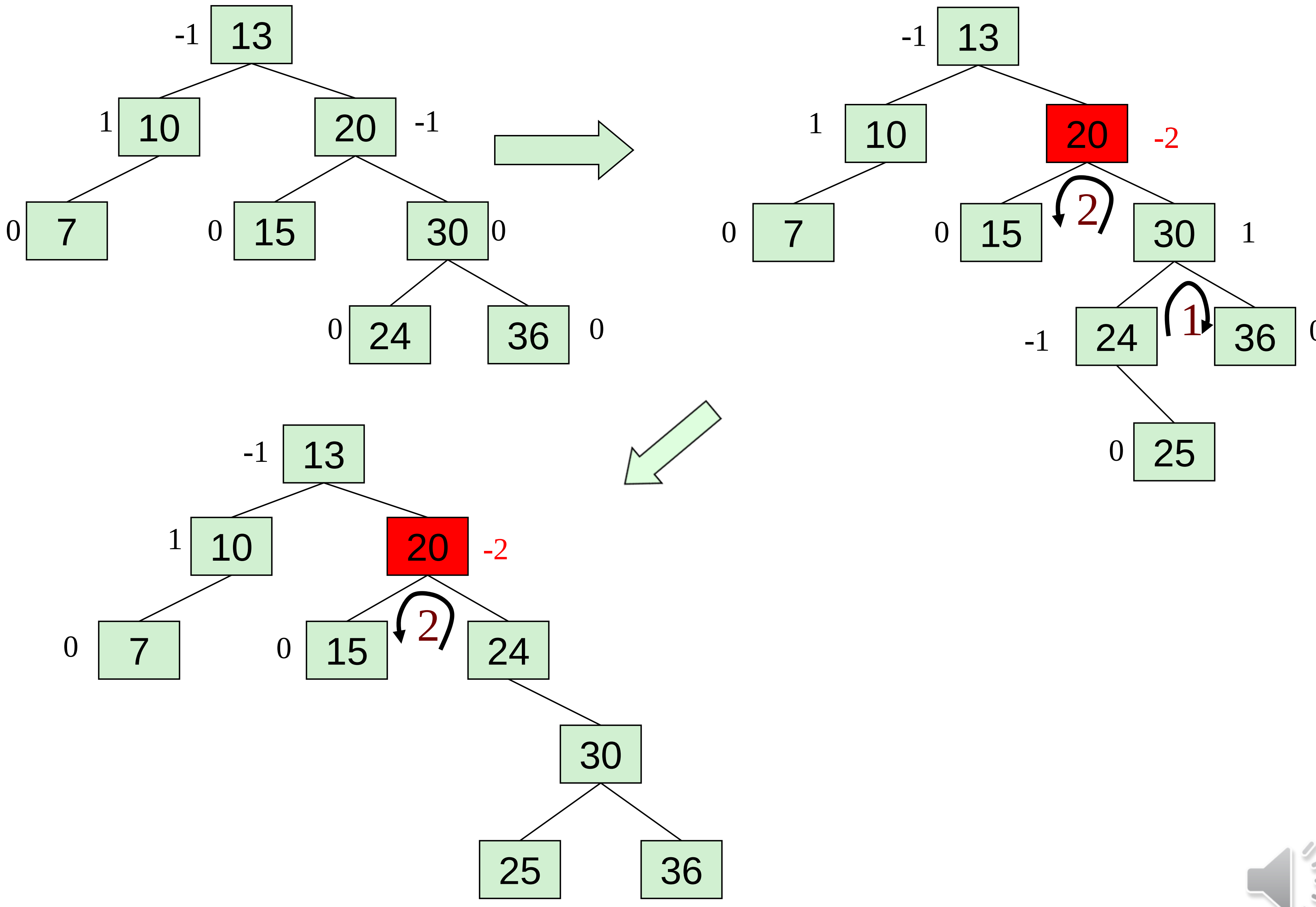
15, 20, 24, 10, 13, 7, 30, 36, 25



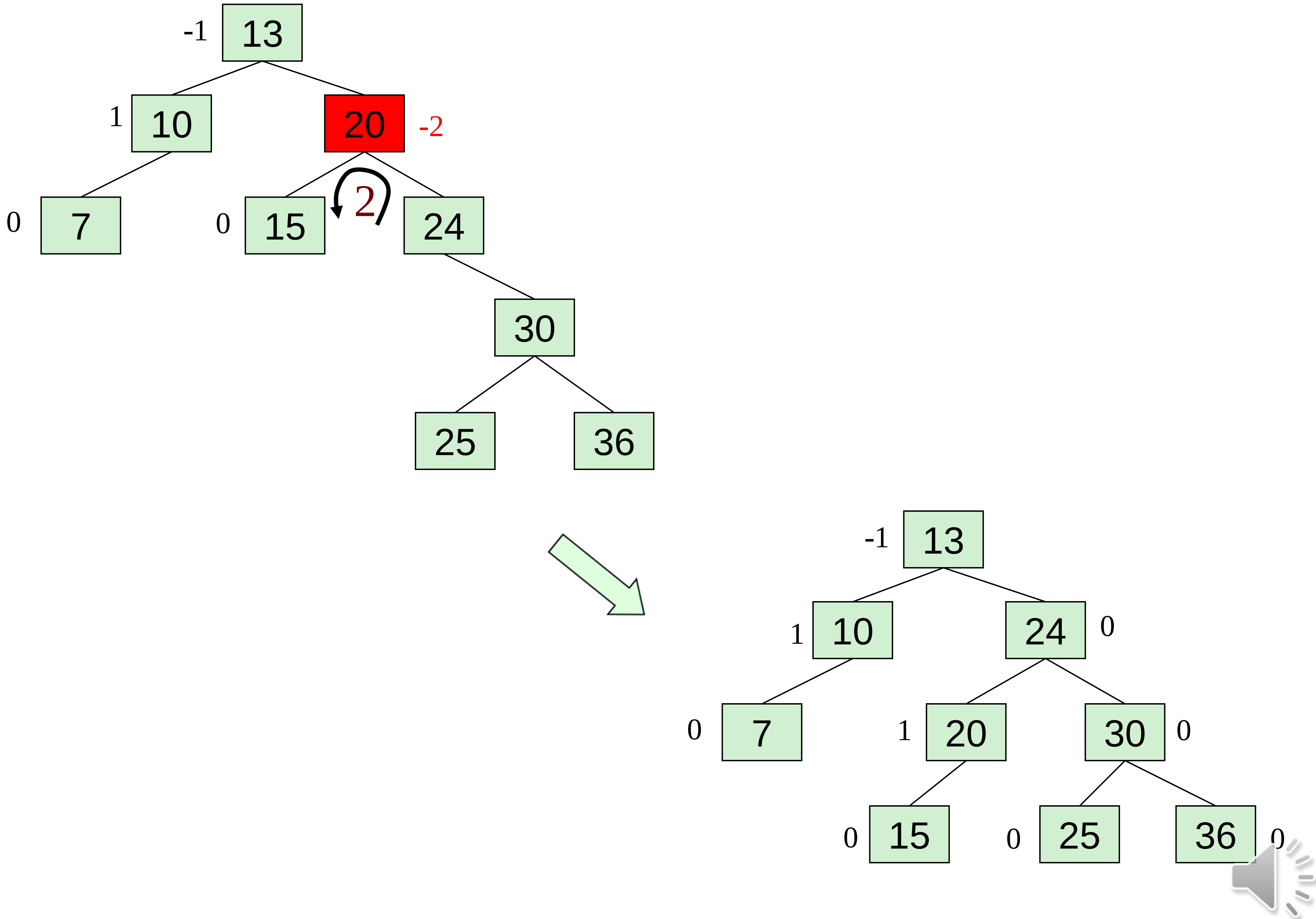
15, 20, 24, 10, 13, 7, 30, 36, 25



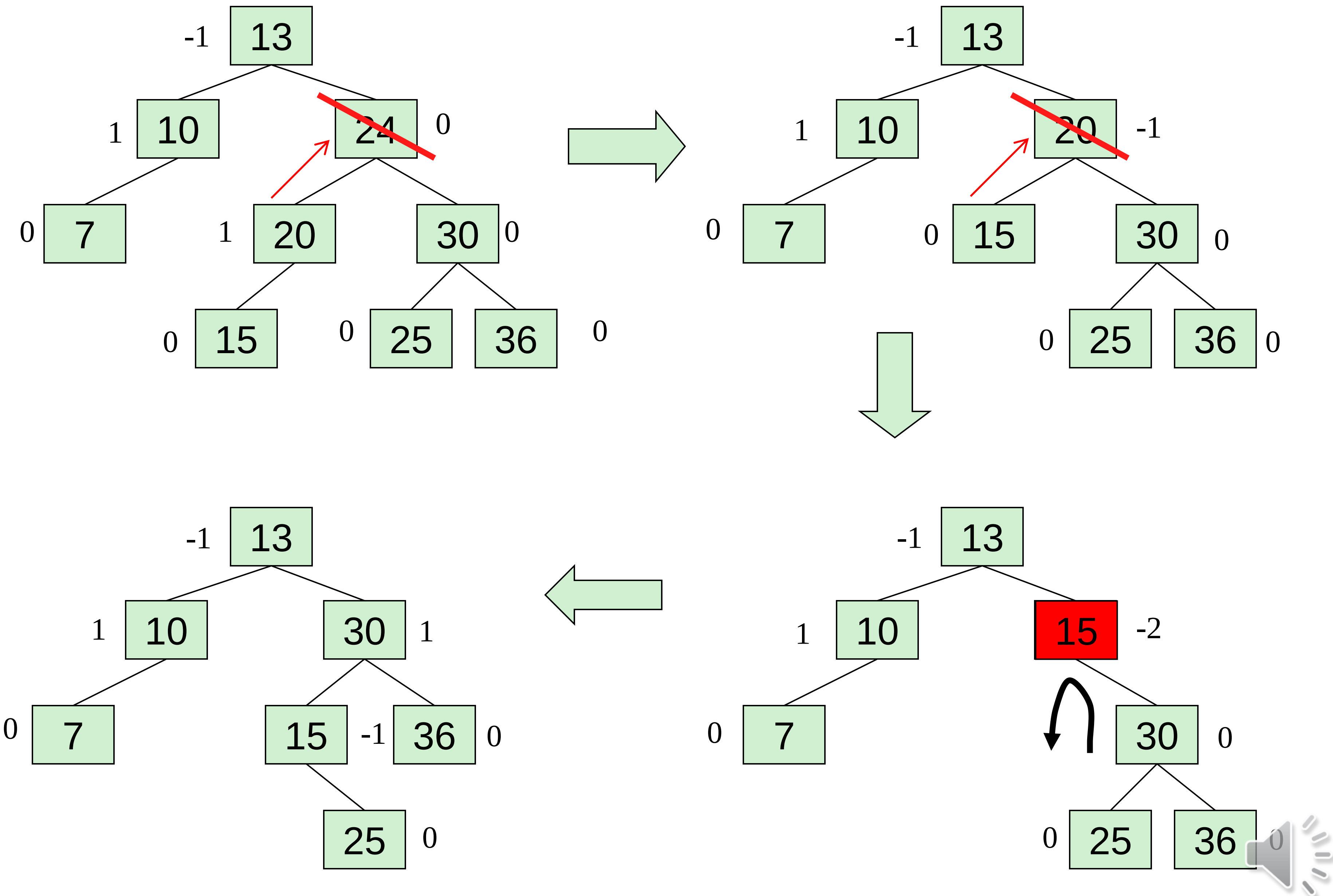
15, 20, 24, 10, 13, 7, 30, 36, 25



15, 20, 24, 10, 13, 7, 30, 36, 25



Remove 24 and 20 from the AVL tree.



Search (Find)

- Since AVL Tree is a BST(이진탐색트리), search algorithm is the same as BST search and runs in guaranteed O(logn) time

Pros and Cons of AVL Trees
장점/단점

Arguments for AVL trees: 삽입,삭제,탐색 모두 O(log N) 보장

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 & debug프로그램하기어려움; more space for balance factor.
2. Asymptotically faster but rebalancing costs time. 균형유지비용 공간소모
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
4. May be OK to have O(N) for a single operation if total run time for many consecutive operations is fast (e.g. Splay trees).

Splay Trees
(스플레이 트리)

splay : 벌리다. 펼치다.

Motivation for Splay Trees

Problems with AVL Trees

- extra storage/complexity for height fields
- ugly delete code

Solution: splay trees

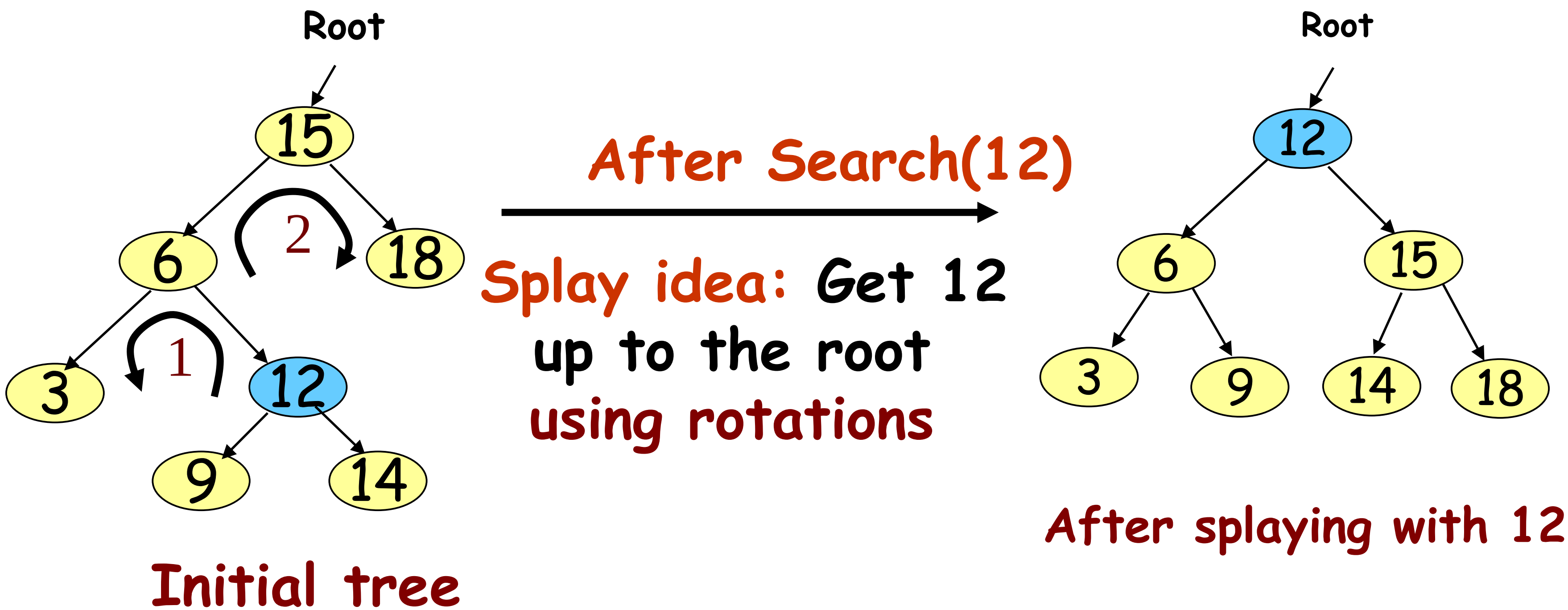
- blind adjusting version of AVL trees
- amortized time for all operations is O(log n)
- worst case time is O(n)
- insert/find always rotates node *to the root*!

Splay Trees

- Splay trees are binary search trees (BSTs) that:
  - Are not perfectly balanced all the time (완전균형이 아님)
  - Allow search and insertion operations to try to balance the tree so that future operations may run faster (삽입,삭제원소를 루트로 가져와 다음 탐색이 빠르도록 함)

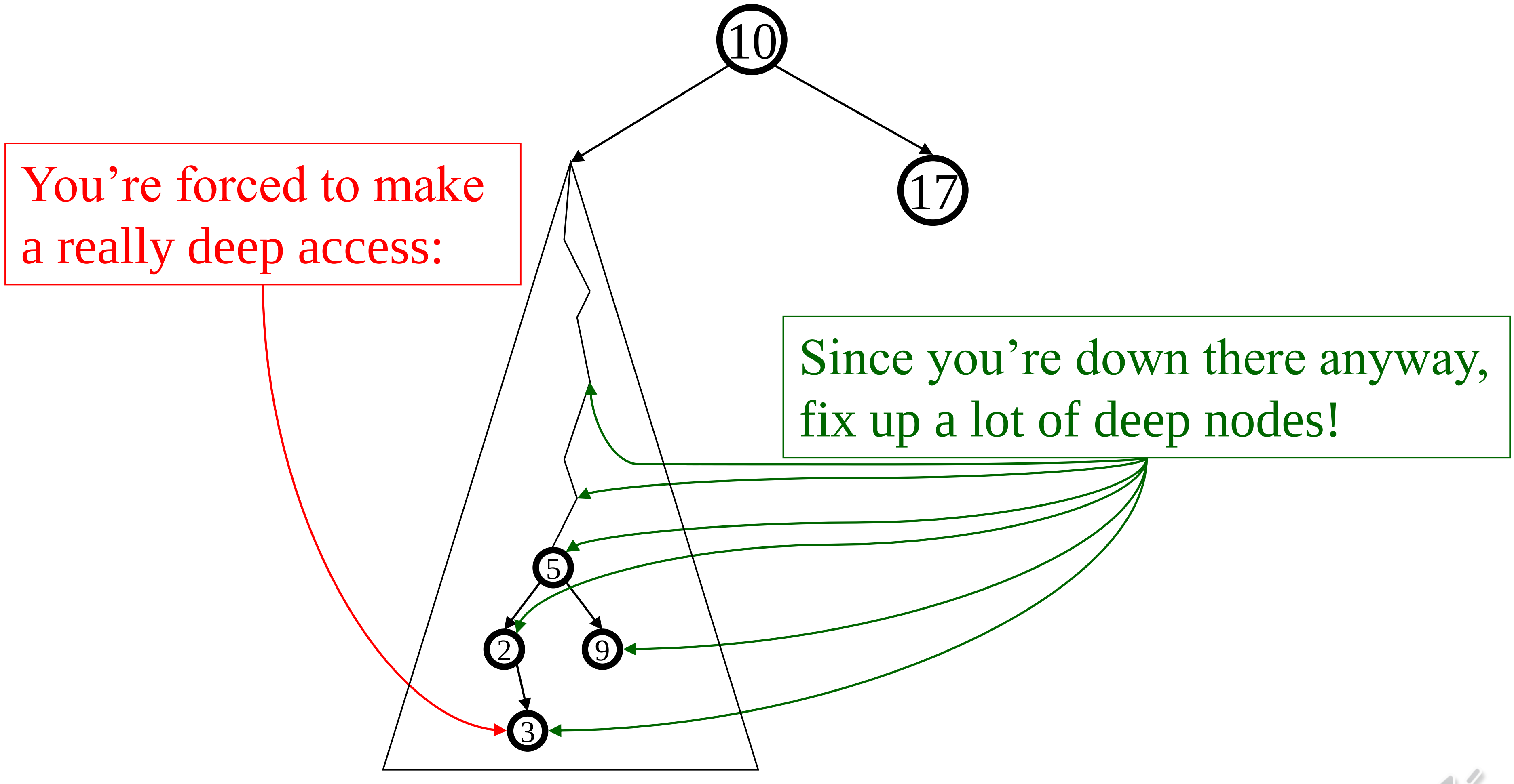
- Based on the heuristic:
  - If X is accessed once, it is likely to be accessed again. (한번 탐색되었던 원소는 다시 탐색되기 쉽다는 가정을 기반)
  - After node X is accessed, perform "splaying" operations to bring X up to the root of the tree.
  - Do this in a way that leaves the tree more or less balanced as a whole.

Motivating Example



- Not only splaying with 12 makes the tree balanced, subsequent accesses for 12 will take O(1) time.
- Active (recently accessed) nodes will move towards the root and inactive nodes will slowly move further from the root

Splay Tree Idea



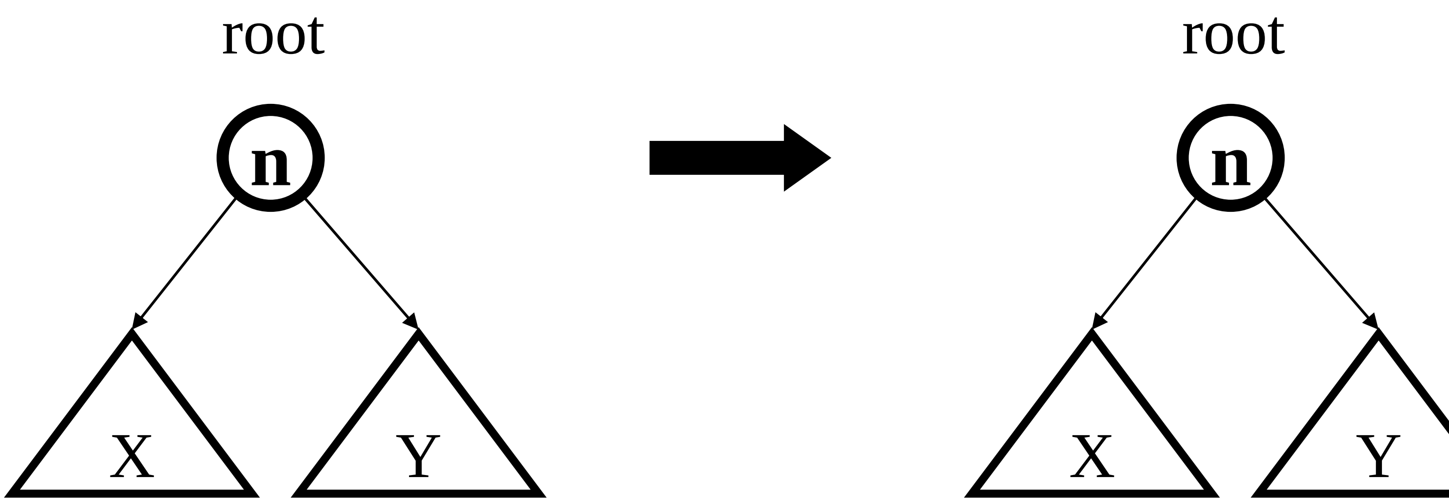
Splaying Cases

Node being accessed (n) is:

- Root
- Child of root
- Has both parent (p) and grandparent (g)
  - Zig-z**i**g pattern: g → p → n is left-left or right-right
  - Zig-z**a**g pattern: g → p → n is left-right or right-left

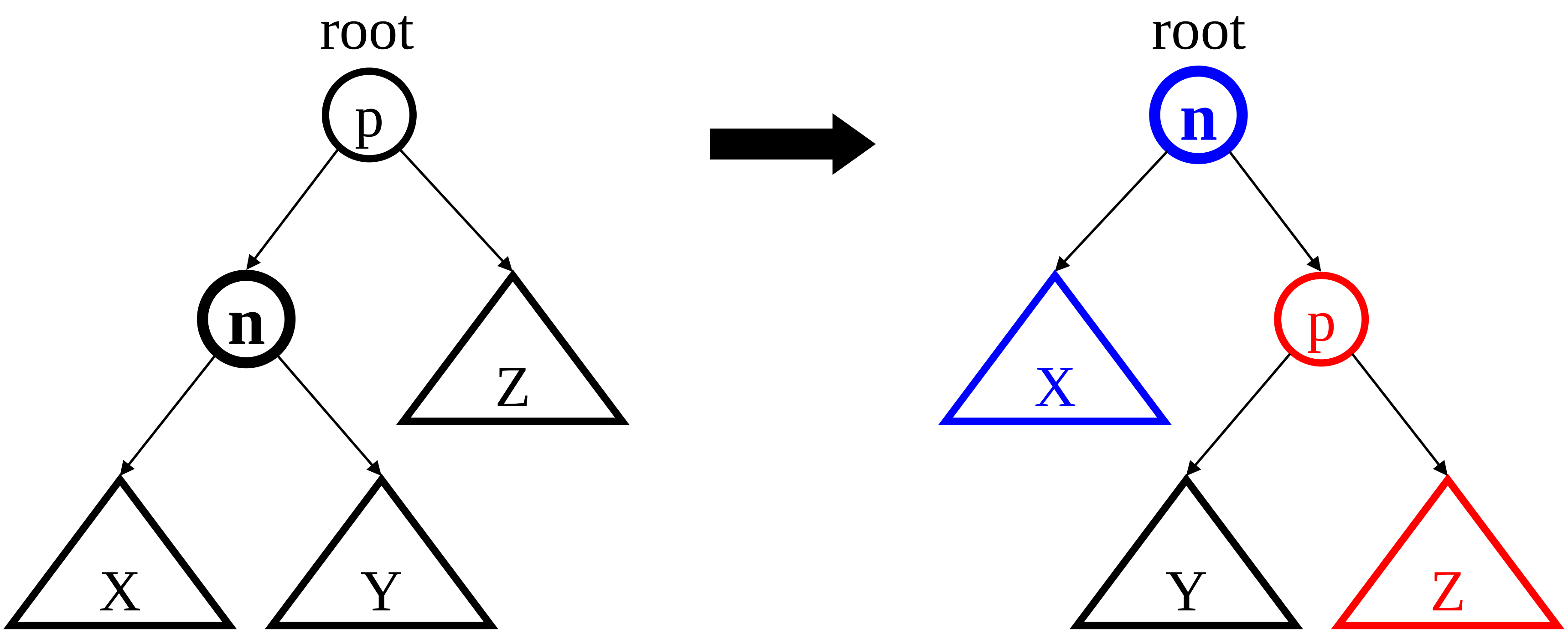
Access root:

Do nothing (that was easy!)



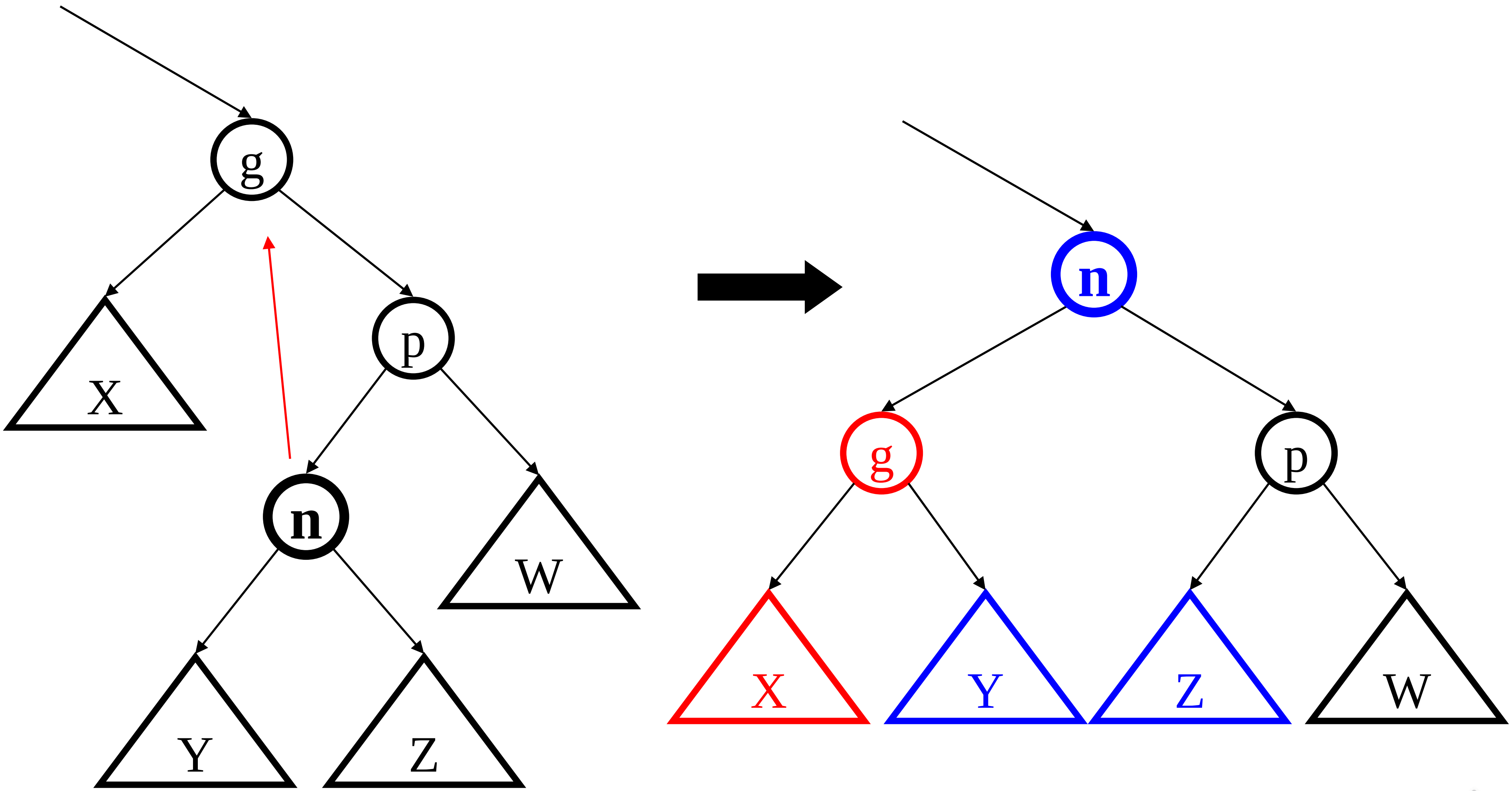
Access child of root:

Zig (AVL single rotation)



Access (LR, RL) grandchild:

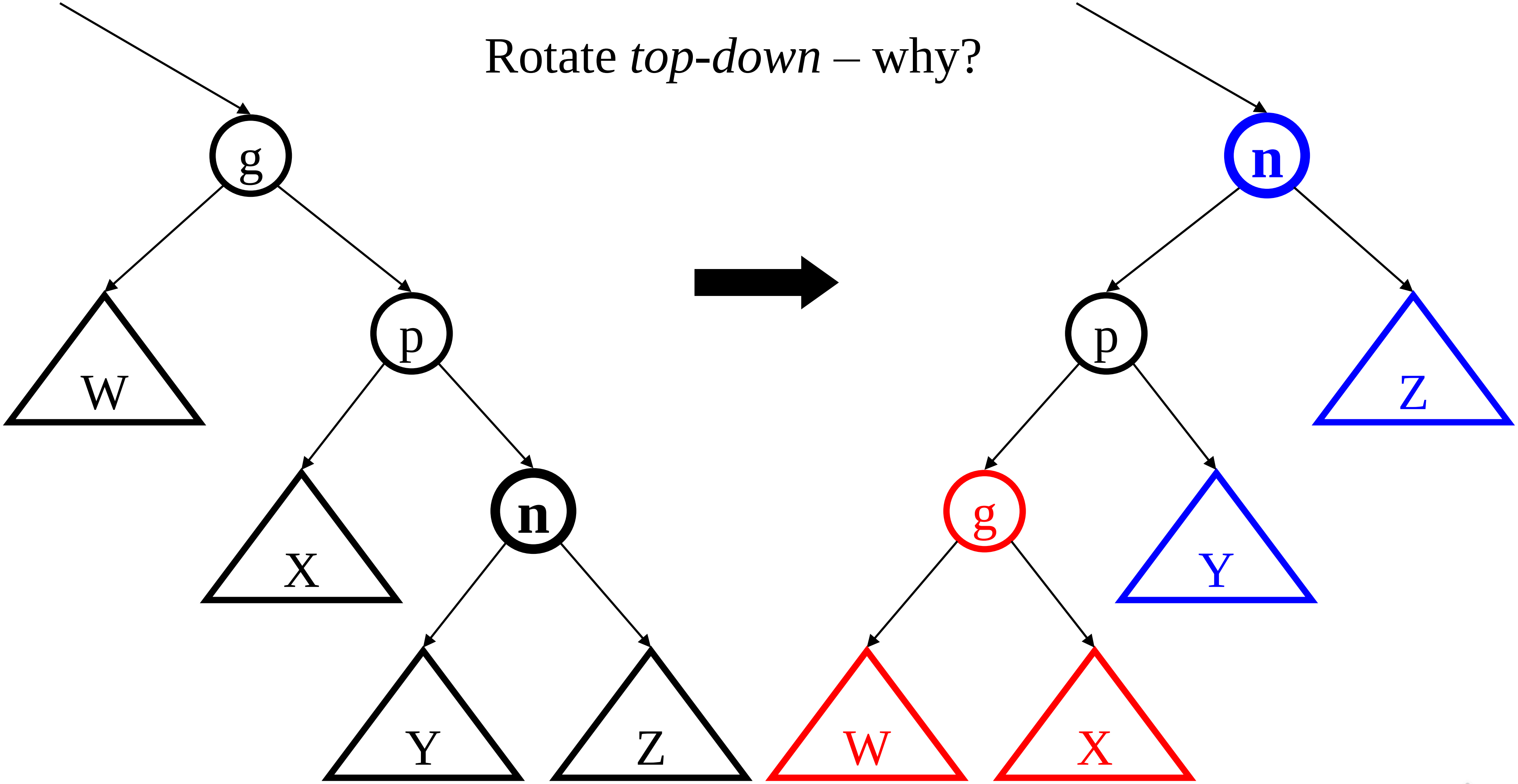
Zig-Zag (AVL double rotation)



Access (LL, RR) grandchild:

Zig-Zig

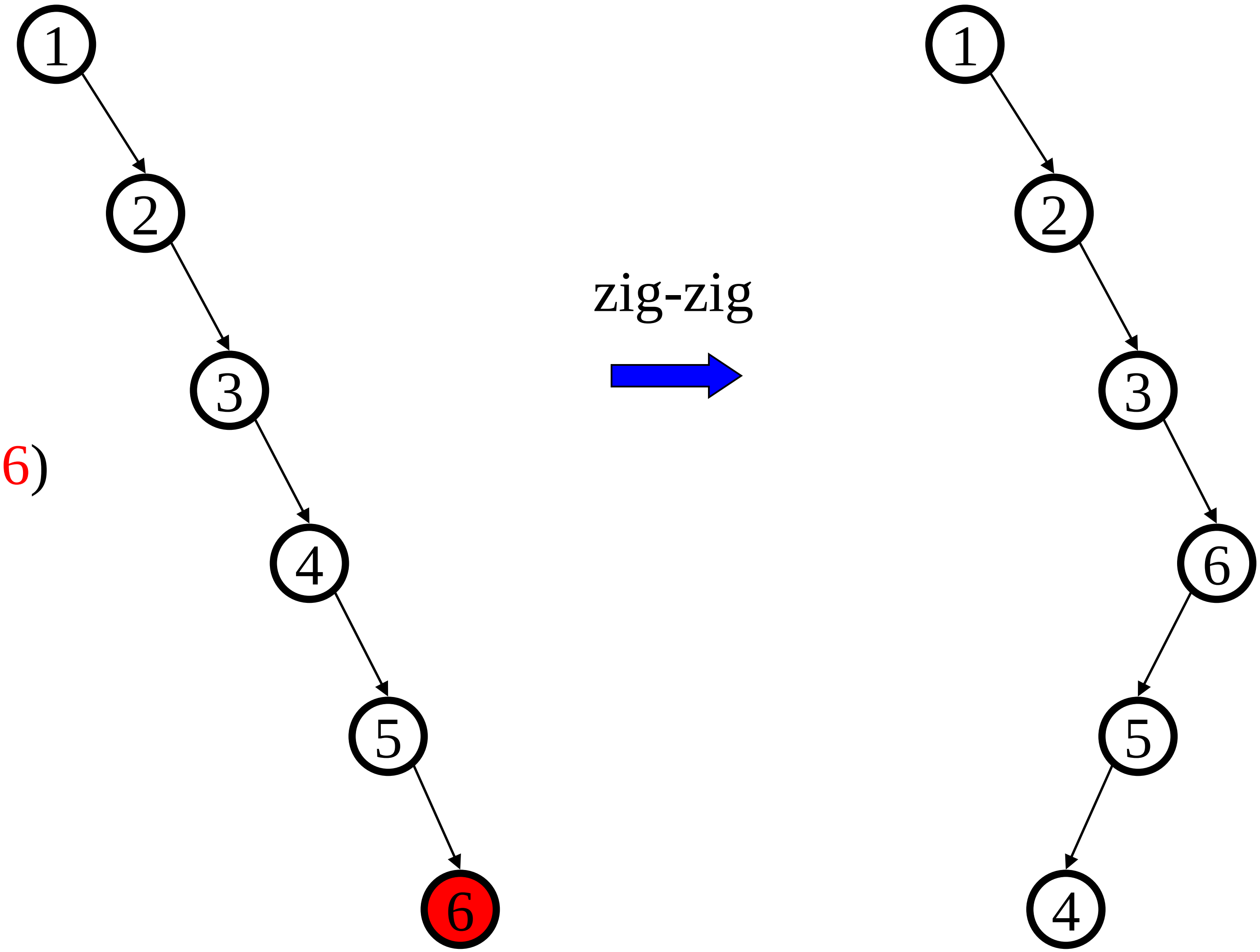
Rotate *top-down* – why?



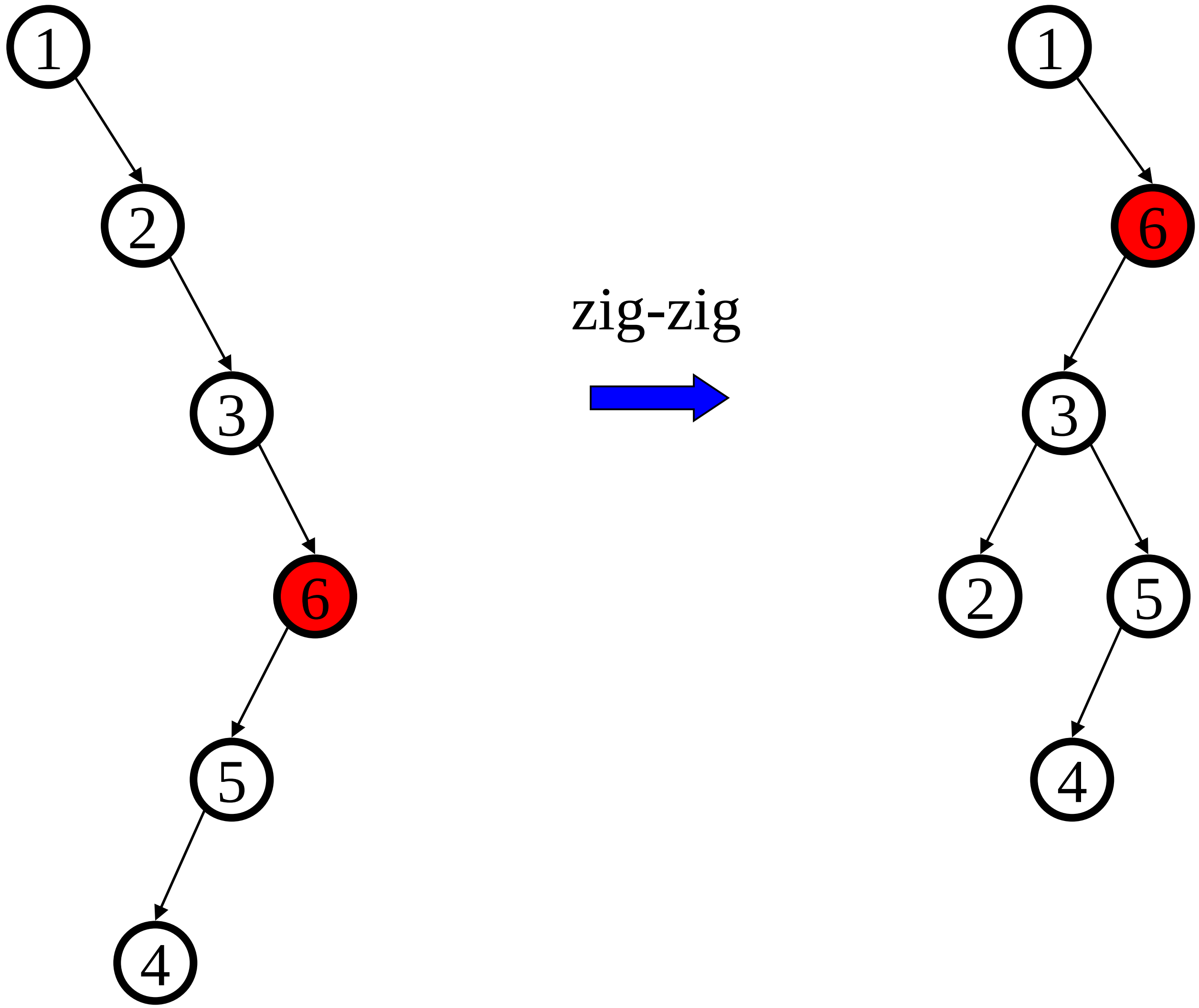
Splaying Example:

Find(6)

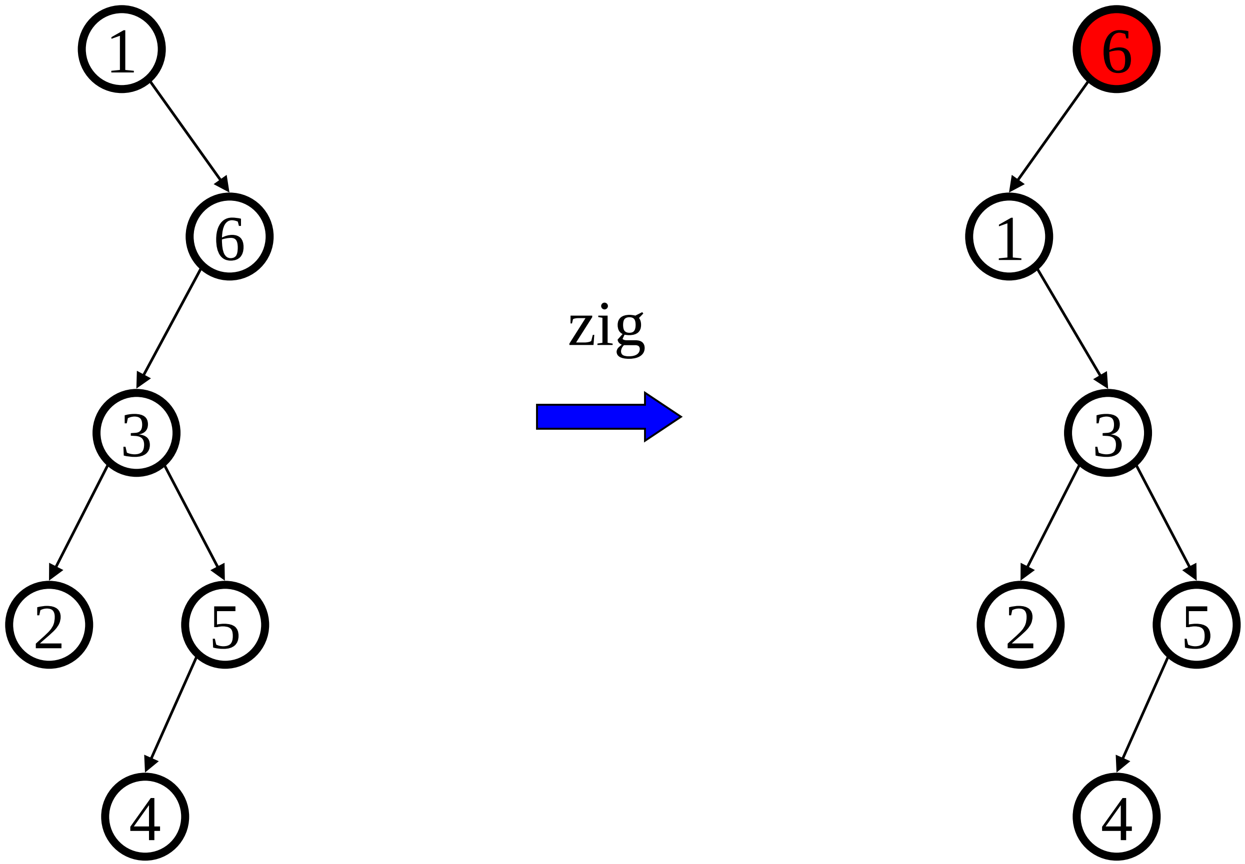
Find(6)



… still splaying …



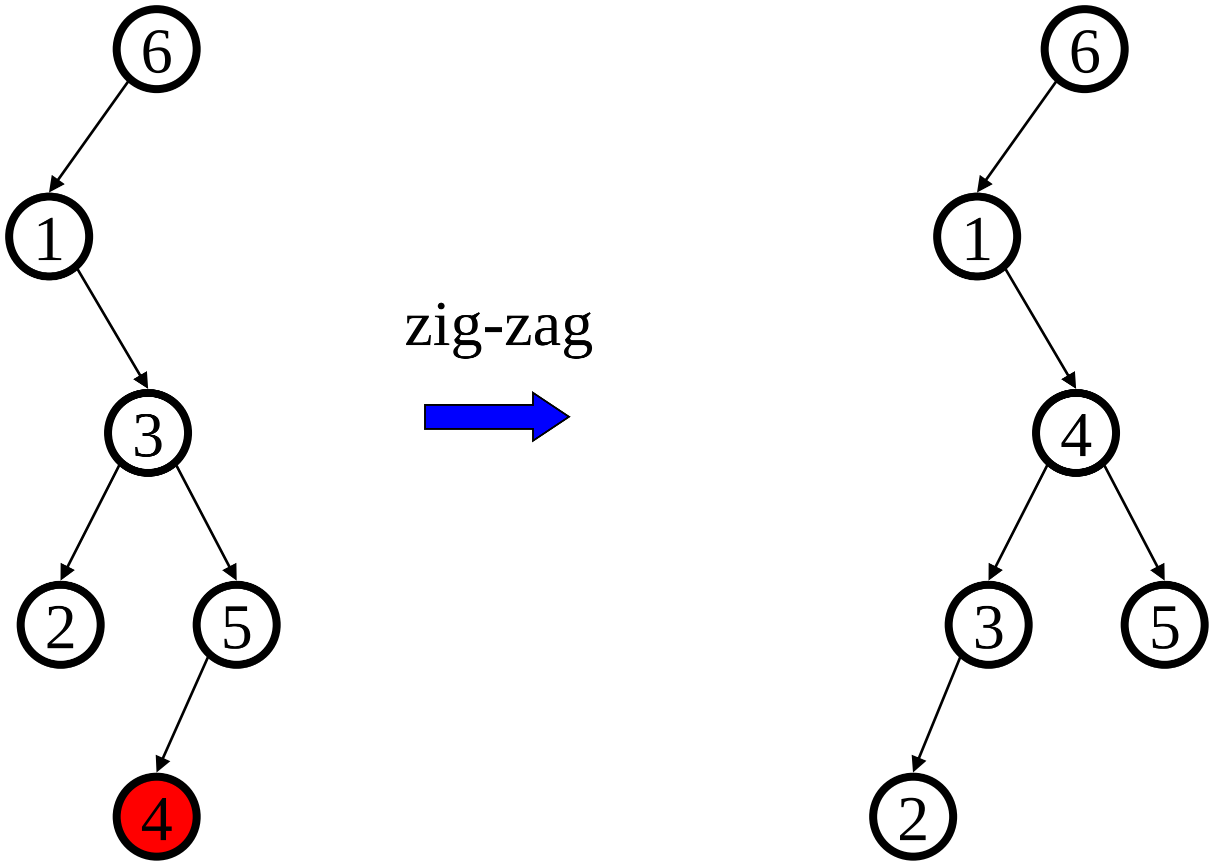
… 6 splayed out!



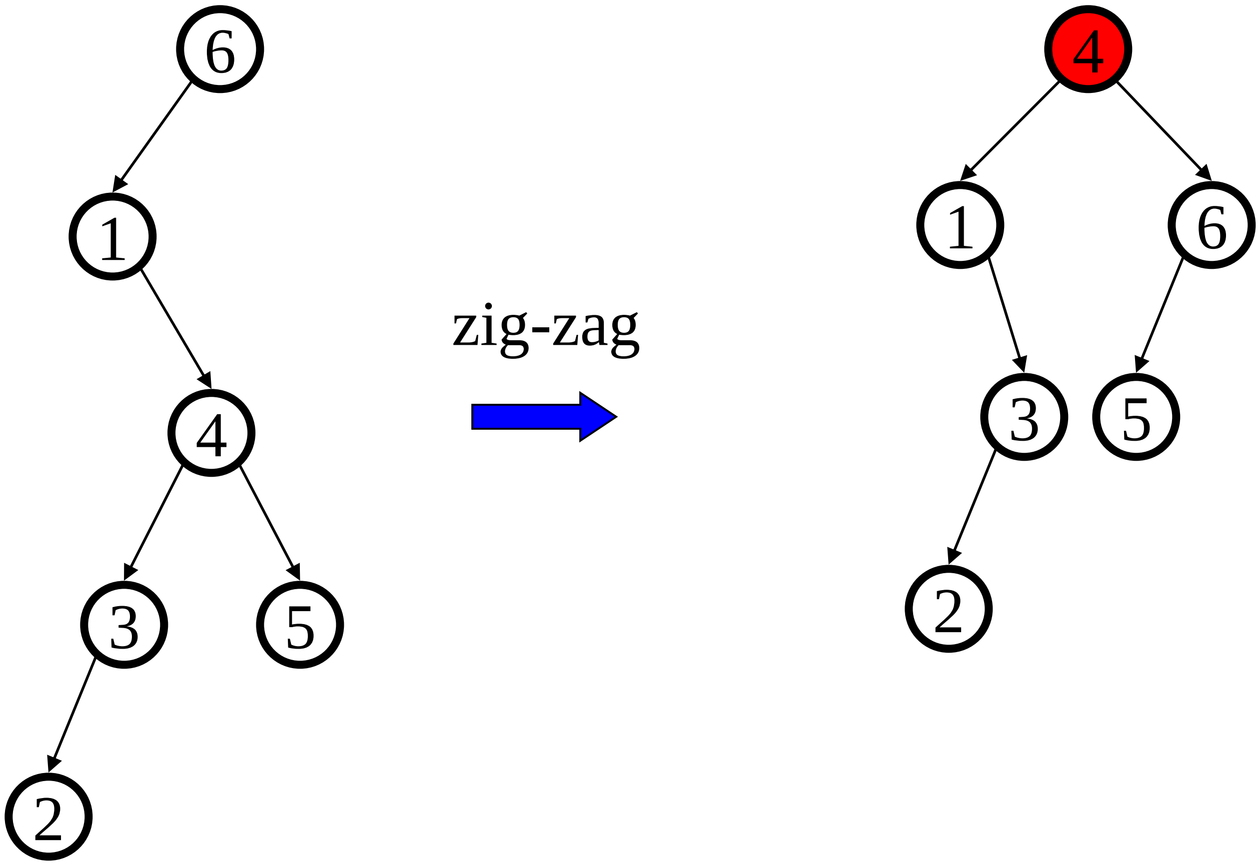
Splay it Again, Sam!

Find (4)

Find(4)



… 4 splayed out!



Why Splaying Helps

- If a node *n* on the access path is at depth *d* before the splay, it's at about depth *d/2* after the splay
  - Exceptions are the root, the child of the root, and the node splayed

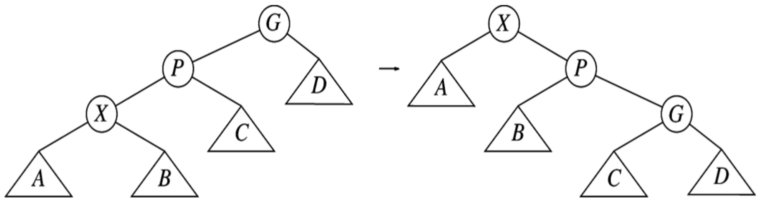
- Overall, nodes which are below nodes on the access path tend to move closer to the root

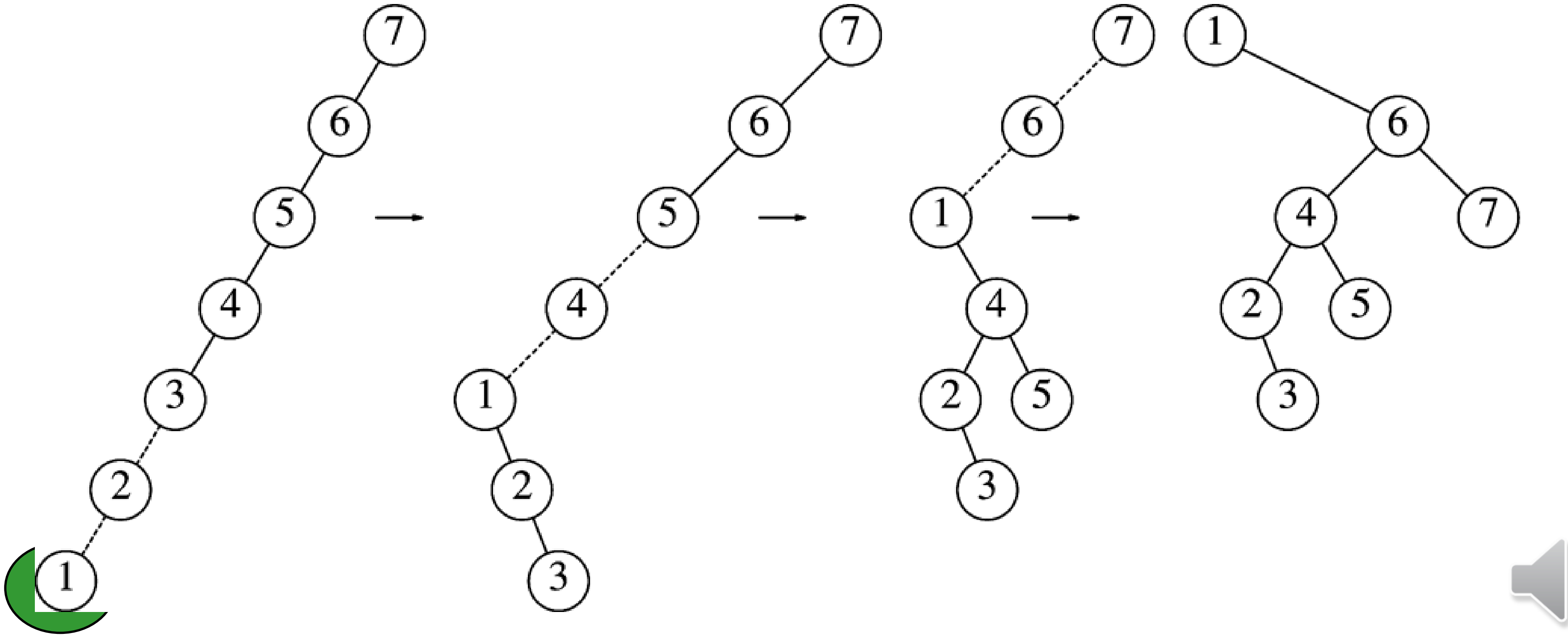
- Splaying gets amortized O(log n) performance. (Maybe not now, but soon, and for the rest of the operations.)

Splay Tree: Splaying

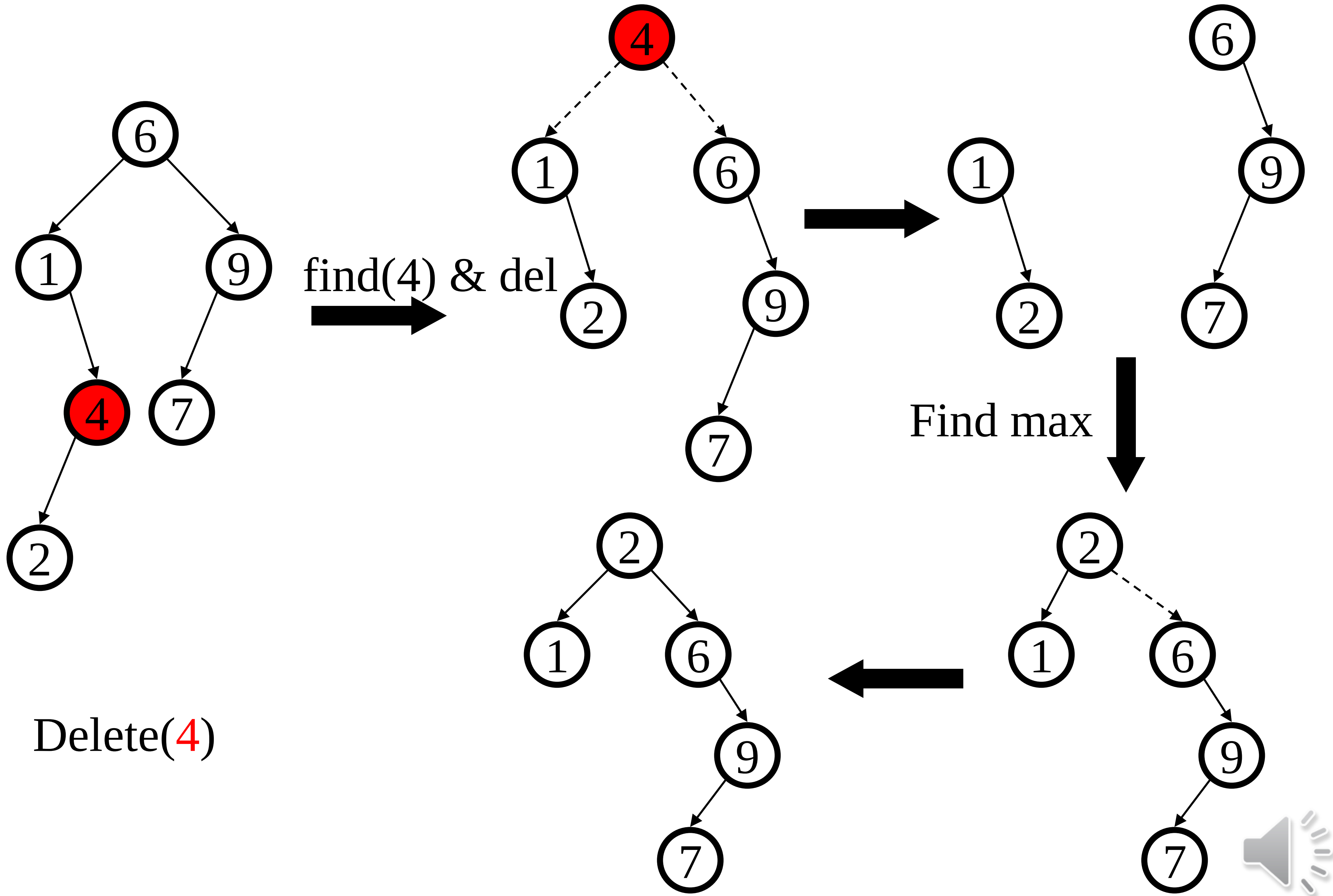
- Work out an Example: Insert node 1 in the following Splay tree

CASE: Zig-zig



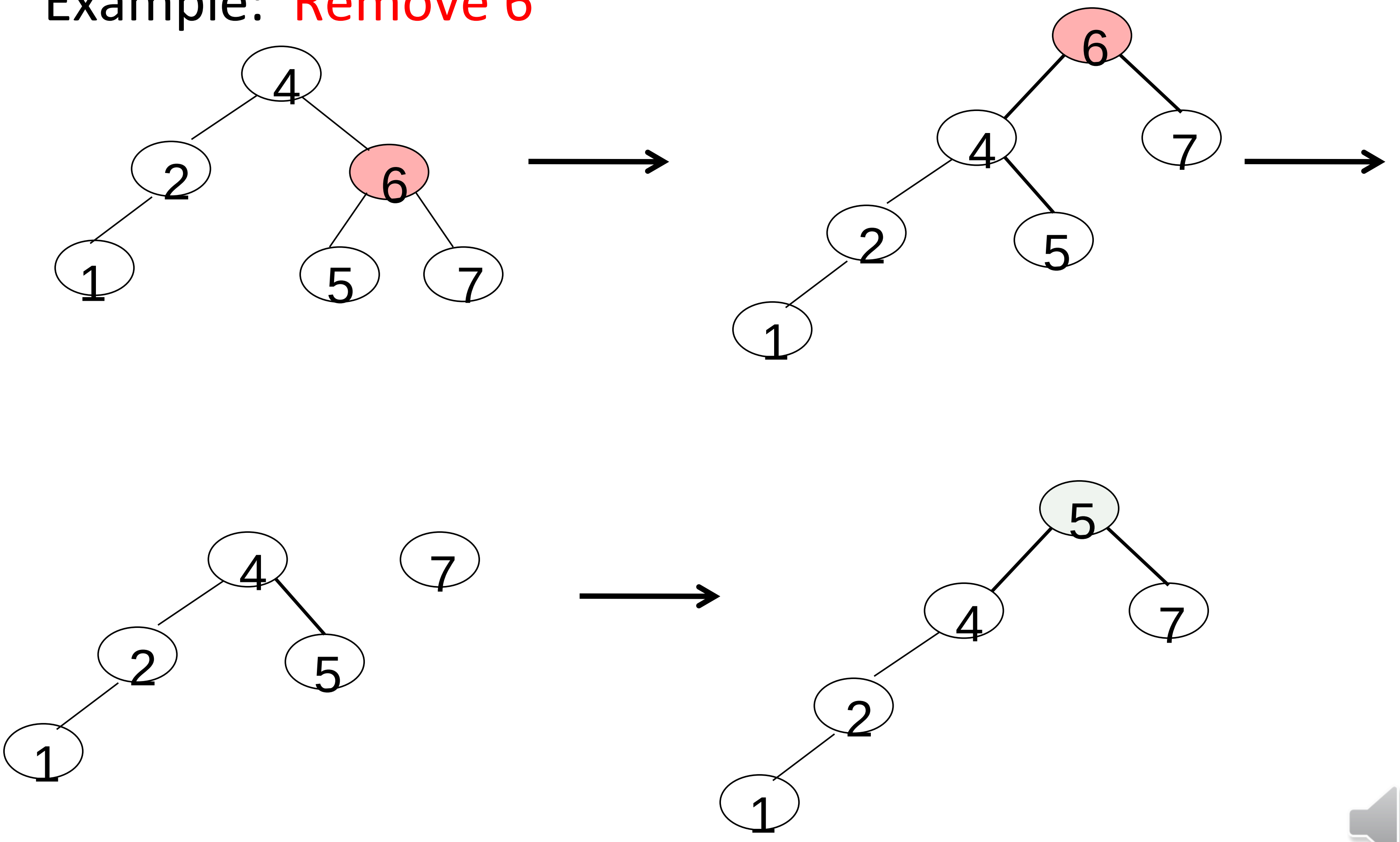


Delete Example



Splay Tree: Remove

- Example: Remove 6



Do it yourself exercise

- Insert the keys 1, 2, …, 7 in that order into an empty splay tree.(1~7까지 차례로 원소를 삽입하면 어떤 스플레이트리가 만들어지는가?)
- What happens when you access “3”?(Final exam)

스플레이 트리

- 스플레이 트리의 시간 복잡도
  - 각 연산(탐색, 삽입, 삭제, 조인, 분할)은 O(*logn*) 상환 시간에 수행할 수 있음
  - 상환 시간(amortized time)
    - 일련의 연산 수행에서 시간이 많이 걸리는 연산의 시간을 적게 걸리는 연산에 전가시킨 뒤의 시간
    - 개개 연산의 최악의 경우에 걸리는 시간이 짧아짐
  - m번의 삽입, 삭제 연산을 수행 → O(m*logn*) 상환 시간

Summary of Splay Trees

- Examples suggest that splaying causes tree to get balanced.

- Result of Analysis: Any sequence of M operations on a splay tree of size N takes O(M log N) time. So, the amortized running time for one operation is O(log N).

- This guarantees that even if the depths of some nodes get very large, you cannot get a long sequence of O(N) searches because each search operation causes a rebalance.

- Without splaying, total time could be O(MN).

감사합니다.