[CSED233-01] Data Structure
AVL Tree

Jaesik Park

Balanced Search Trees

- Memory-based search trees
  - Everything is in the main memory => fast, easy to implement, but... scale issue
  - Balanced BST
    - AVL (Adelson-Velskii & Landis) trees
    - Red-black trees
    - Splay trees, ...
  - Balanced multi-way search tree
    - 2-3, 2-3-4 trees (B-trees)

- Disk-based search trees
  - More scalable
  - Balanced multi-way search trees
    - B-trees (B+, B*)
    - Prefix B-trees

Balanced Trees

- Tree height is $O(\log n)$
  - No leaf is much further away from the root than any other leaf
  - To minimize the longest path from the root to any leaf node

- Why balanced?
  - Insert, Delete, and Member take $O(\log n)$ time

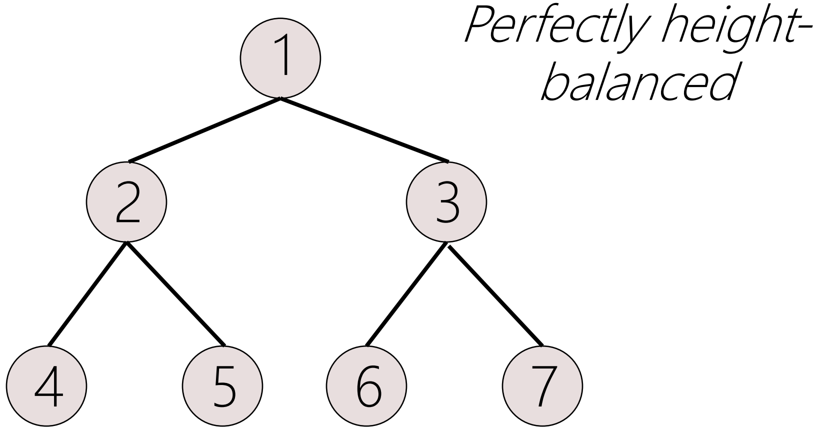
Data Structure	Worst case	Average
Binary Search Tree (BST)	$O(n)$	$O(\log n)$
Balanced Search Trees	$O(\log n)$	$O(\log n)$

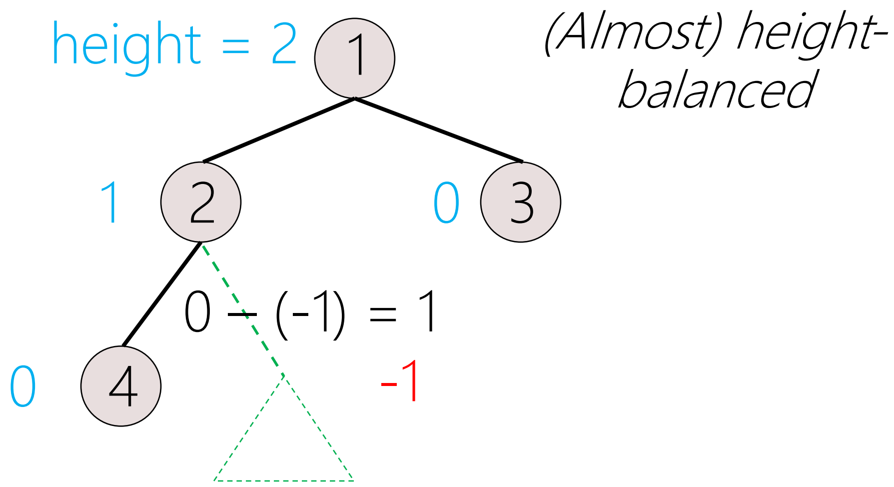
Height-Balanced

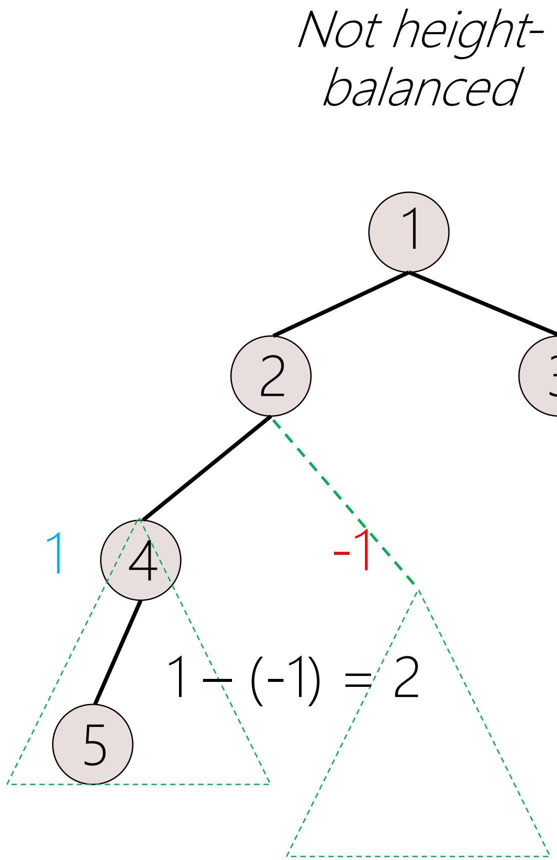
- Perfectly height-balanced
    - If the sub-trees of any node are of the same height, so that all leaf nodes are at the same level
    - Too tough condition to satisfy

- ('Almost' perfectly) height-balanced
    - For each node, the height of its sub-trees can differ by at most 1, & the sub-trees are also height-balanced
    - We can prove that height = $O(\log n)$ since it is an almost balanced tree

Example: Height-Balanced







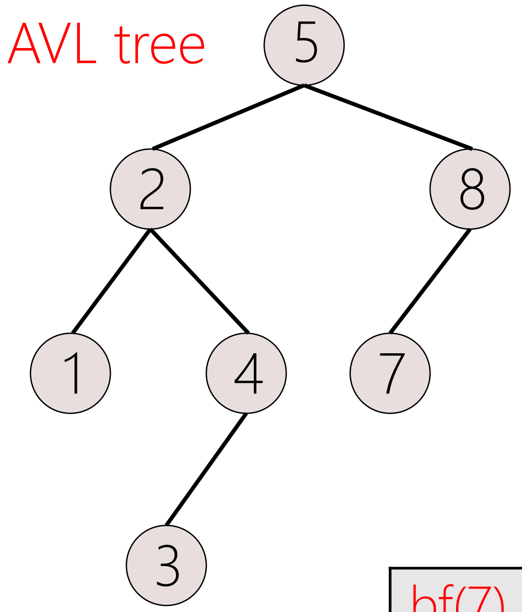
Height for non-existing subtree is -1

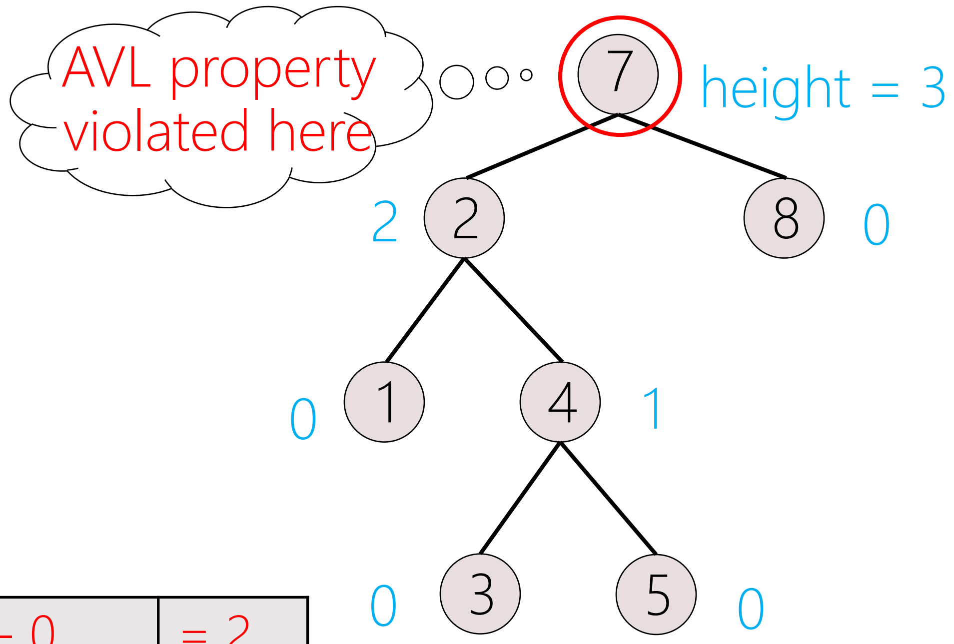
AVL Tree

- Height-balanced BST
  - For every node in the tree, the left & right sub-trees differ in height by at most 1
- Balance factor of node x
  - $bf(x)$ = height(left(x)) - height(right(x))
  - Note:
    - height(one node) = 0, height(*empty* tree) = -1
- AVL tree property
  - A valid AVL tree must satisfy: $-1 \leq bf(x) \leq 1$ for every node x

AVL Tree & Balance Factors

- An invalid AVL tree can be detected by bf(x)



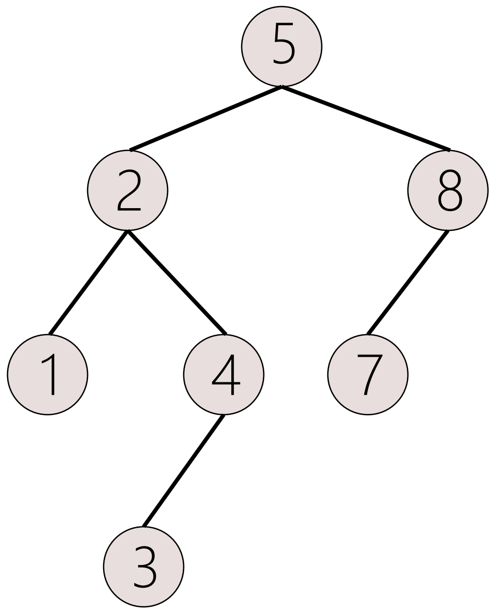


bf(7)	= 2 - 0	= 2
bf(2)	= 0 - 1	= -1
…	…..	…
bf(3)	= (-1) – (-1)	= 0
bf(5)	= (-1) – (-1)	= 0

Insertion in AVL tree

- Insertion can be done as in BST
  - But, it may cause violation of AVL tree property (height balanced)

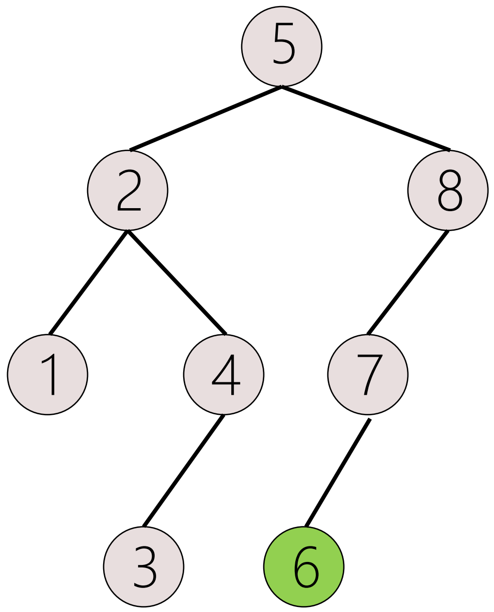
- Restore the AVL property if needed
  - *Insert*(6)



Insertion in AVL tree

- Insertion can be done as in BST
  - But, it may cause violation of AVL tree property (height balanced)

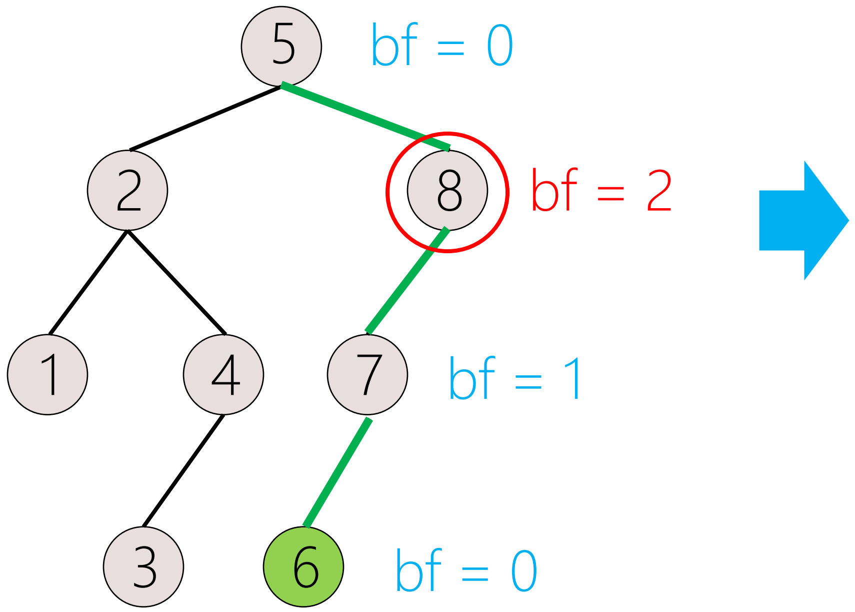
- Restore the AVL property if needed
  - After *Insert*(6)



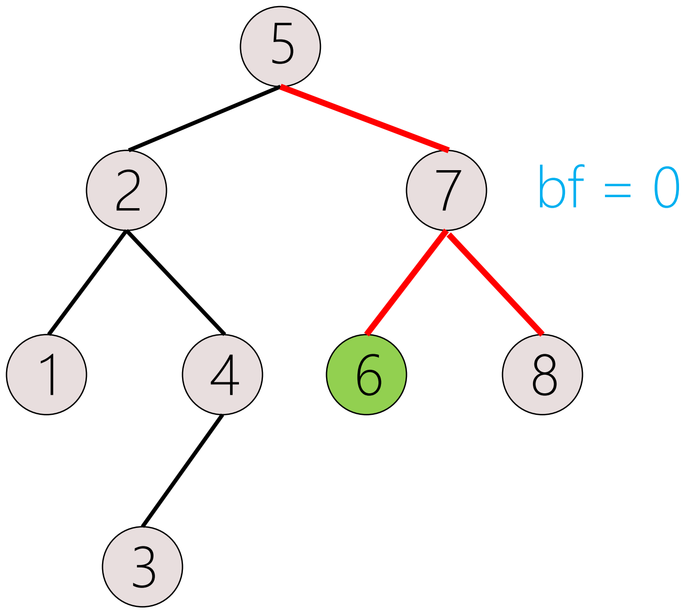
Insertion in AVL tree

- Insertion can be done as in BST
  - But, it may cause violation of AVL tree property (height balanced)

- Restore the AVL property if needed
  - After *Insert*(6), violated at 8

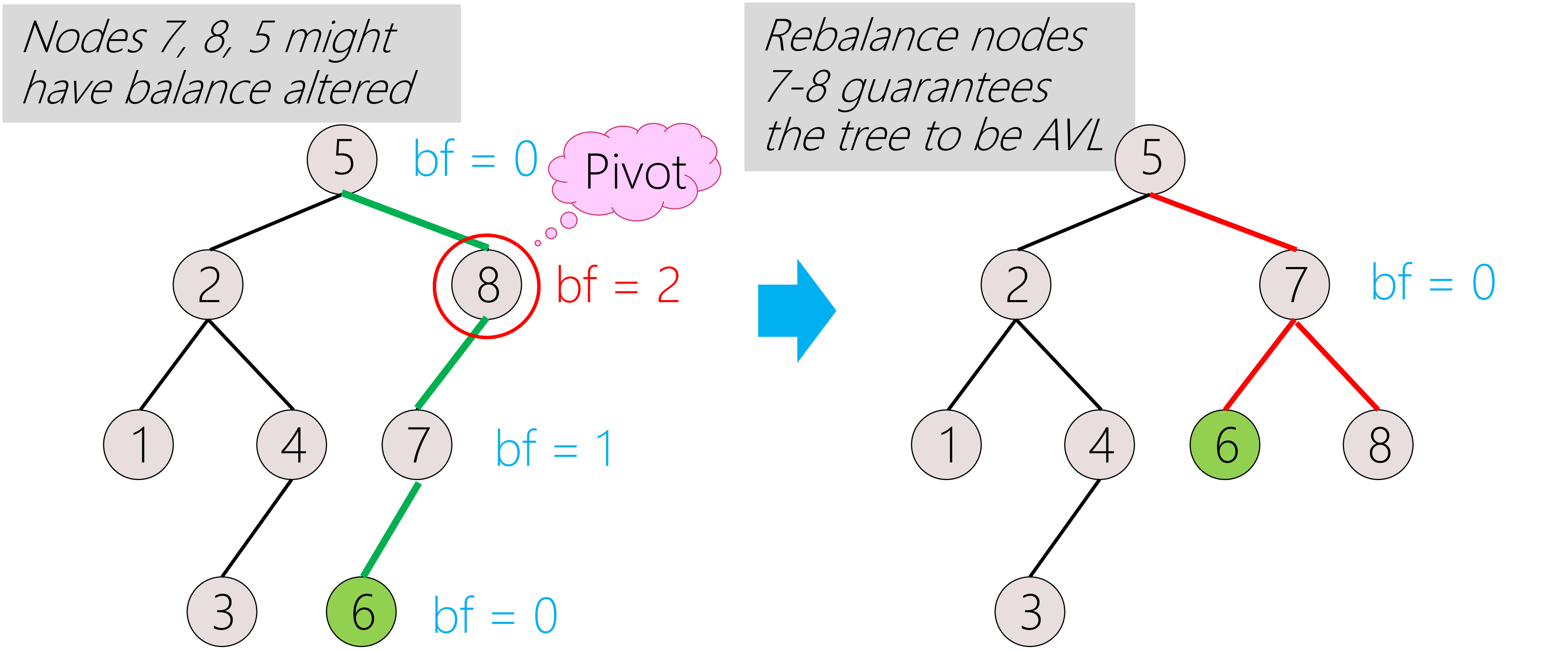


- Restore AVL property



Some Observations

- After insertion,
  - Only nodes along the path from insertion point to root might have their balance altered
  - Need to rebalance the tree at such the "deepest" node (called *pivot*)



Pivot Node & Rotation

- Pivot node (denoted by A)
  - The nearest ancestor of the newly inserted node that may go out of balance
    - The "deepest" unbalanced node from the root
  - bf(A) becomes $+2$ or -2 after insertion
- Restoring the balance of the pivot node
  - Restores the balance of the whole sub-tree & potentially all of the nodes that were affected by the insertion
  - This restructuring mechanism is called "rotation"

Imbalance Types

- Let *A* be the pivot node
  - The deepest node that violates the AVL property

- Only four possible types:
  - LL: insertion into left sub-tree of left child of *A*
    - The insertion took place in the *left sub-tree* of a node whose parent (*A*) was *left high*
  - RR: symmetric case of LL
  - LR: insertion into right sub-tree of left child of *A*
    - The insertion took place in the right sub-tree of a node whose parent (A) was *left high*
  - RL: symmetric case of LR

Tree Rotations: Two Types

- Single rotation (for LL or RR imbalance)
  - Requires only a single rotation on the pivot node as an axis

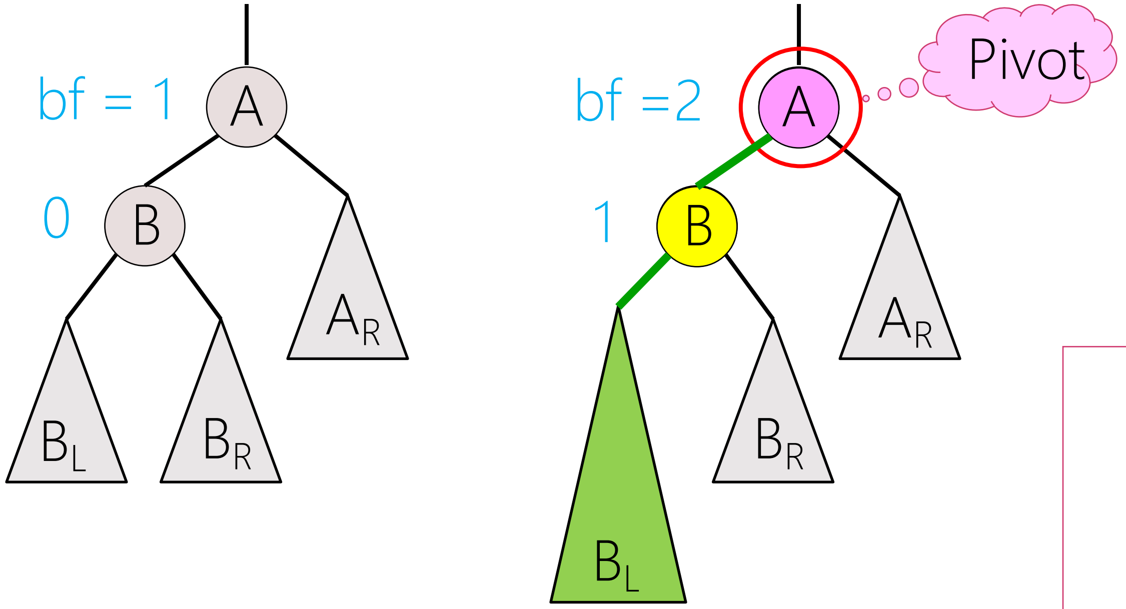
- Double rotation (for LR or RL imbalance)
  - Requires a prior rotation on the root of the affected sub-tree, then a rotation on the pivot node
    - LR = RR + LL
    - RL = LL + RR
  - The first rotation does not affect the degree of imbalance but convert it to a simple LL or RR

Insertion Algorithm

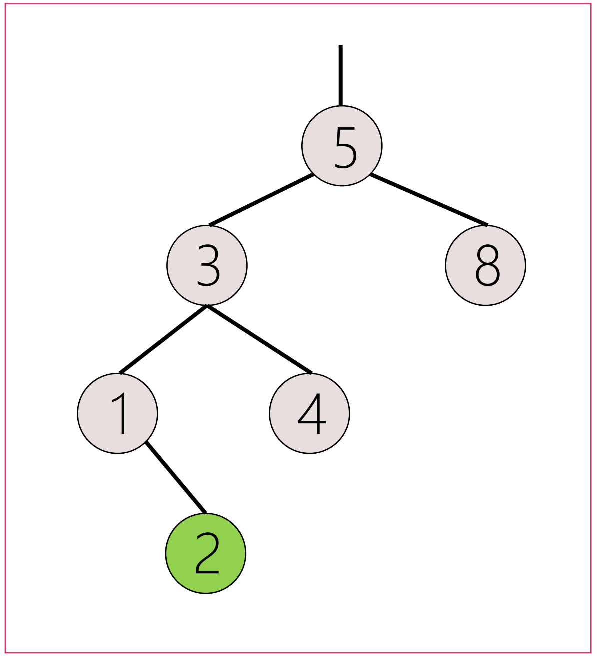
- First, insert a new key as in BST
- Then, trace the path from the new leaf towards the root:
  - For each node x, check if $-1 \le bf(x) \le 1$
    - If yes, proceed to parent(x)
    - If no, rotate the tree on node x, and then finish (early termination)
      - That's the pivot

- Note:
  - Once we perform a rotation on node x, we don't need to do any rotation on any ancestor of x
  - Rotation may change x, so connect the resulting tree to parent(x)
  - Update the height of nodes involved in rotations

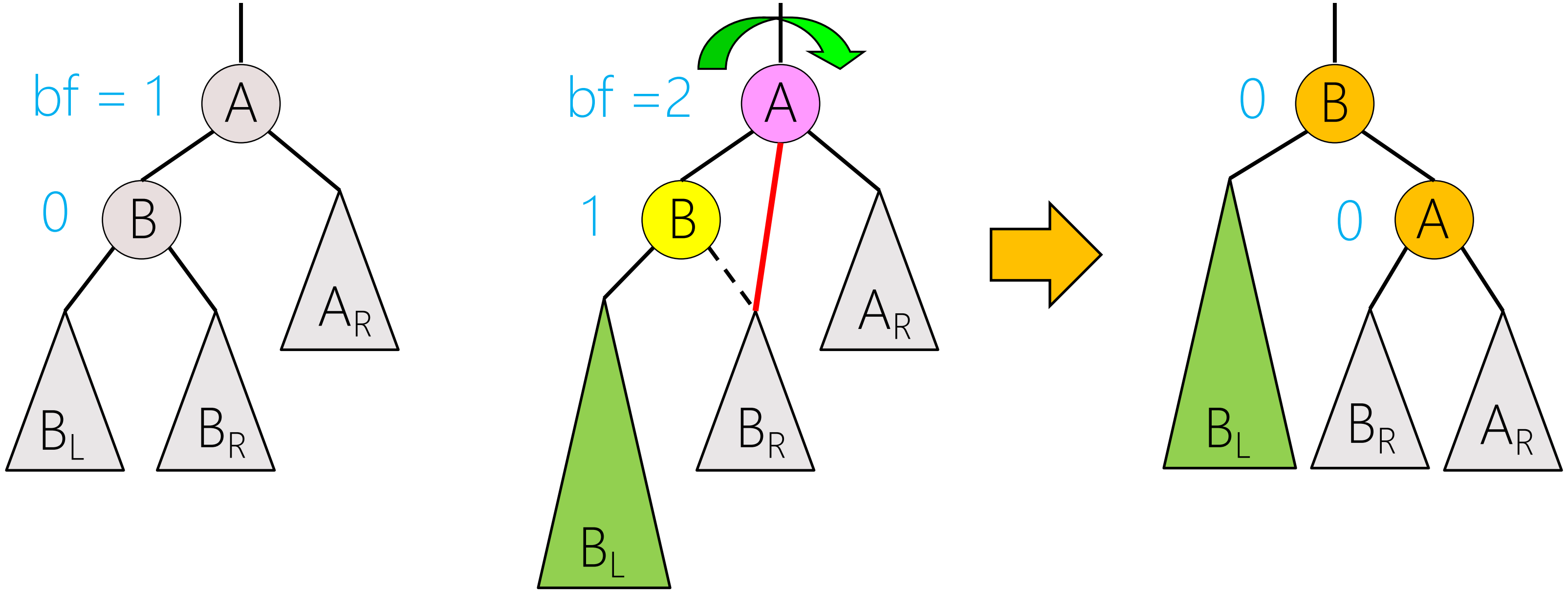
Single Rotation: LL Case



Before insertion *After insertion*



Single Rotation: LL Case (Right Rotation)

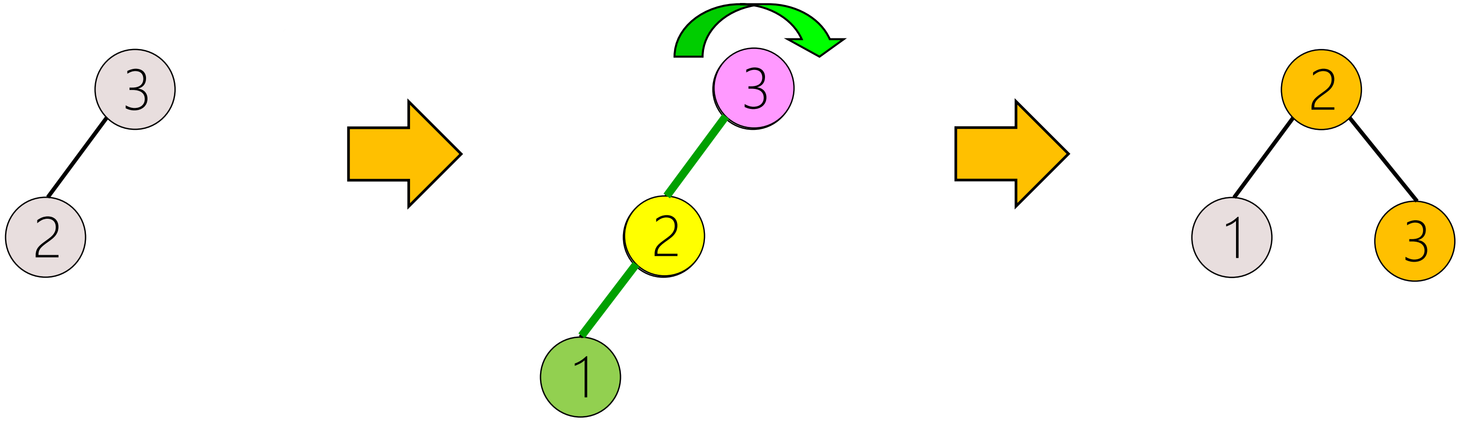


Before insertion *After insertion* *After LL (Right) rotation*

- A single rotation takes O(1) time
  - Insertion takes O(Height of AVL tree) time
- RR case is symmetric to LL case

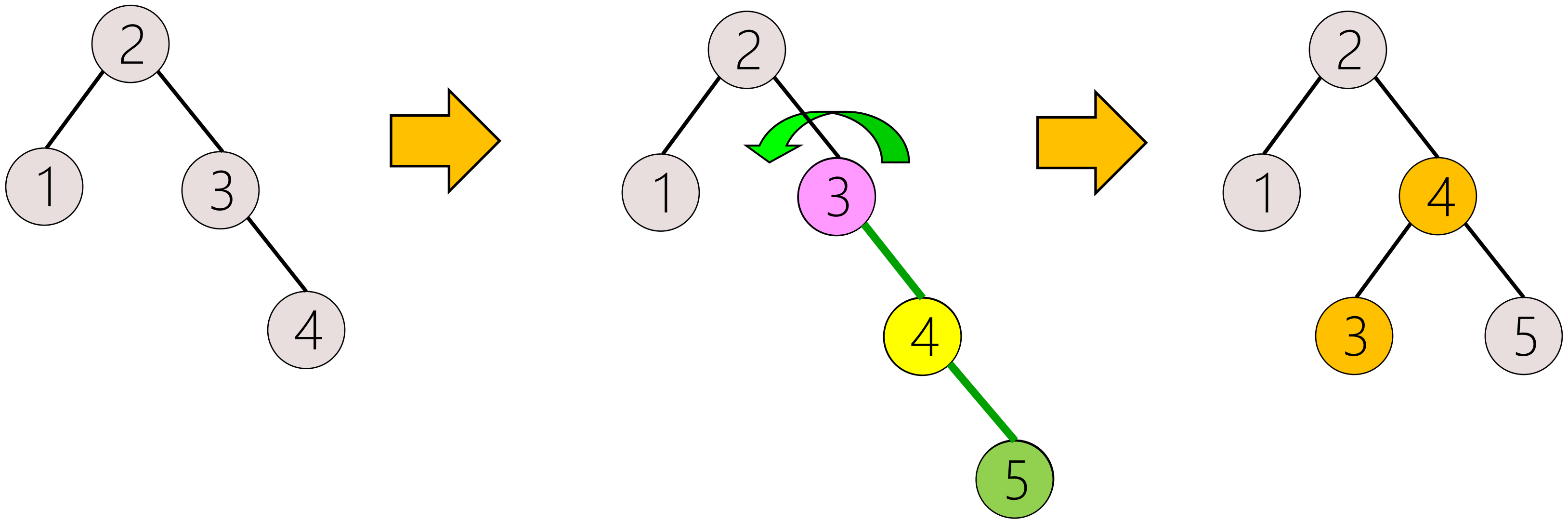
Single Rotation: Example

- Insert 3, 2
- Insert 1
  - Violation at node 3
  - Right (LL) rotation



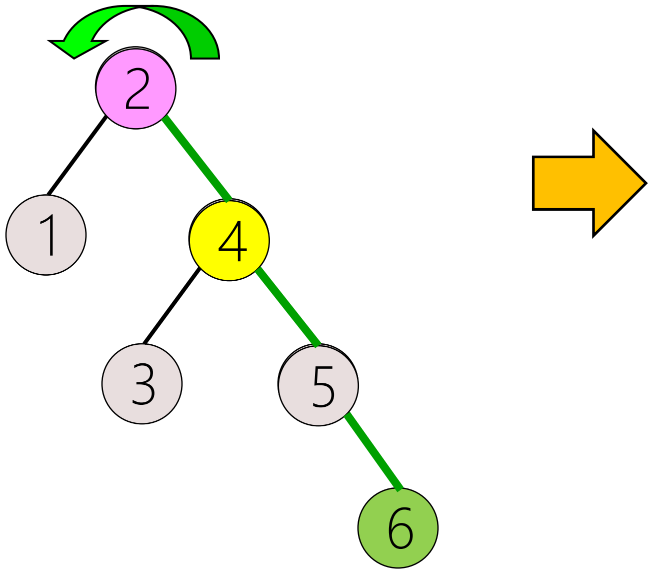
Single Rotation: Example

- Insert 4
- Insert 5
  - Violation at node 3
  - Left (RR) rotation

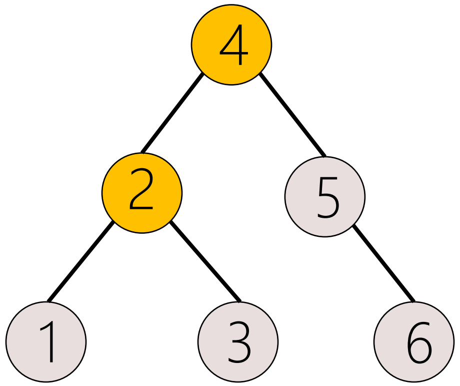


Single Rotation: Example

- Insert 6
  - Violation at node 2

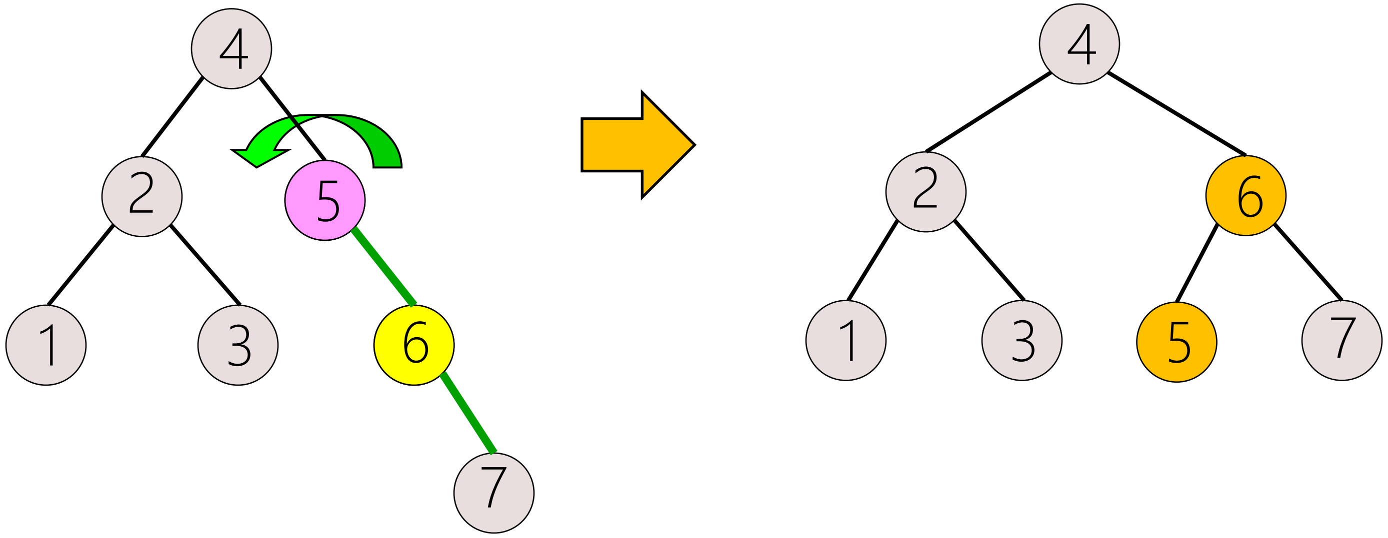


- Left (RR) rotation



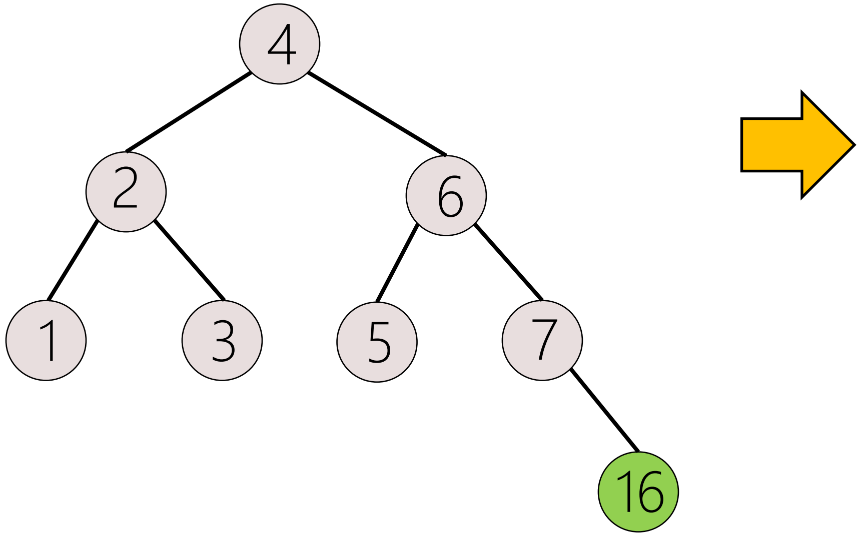
Single Rotation: Example

- Insert 7
  - Violation at node 5
  - Left (RR) rotation

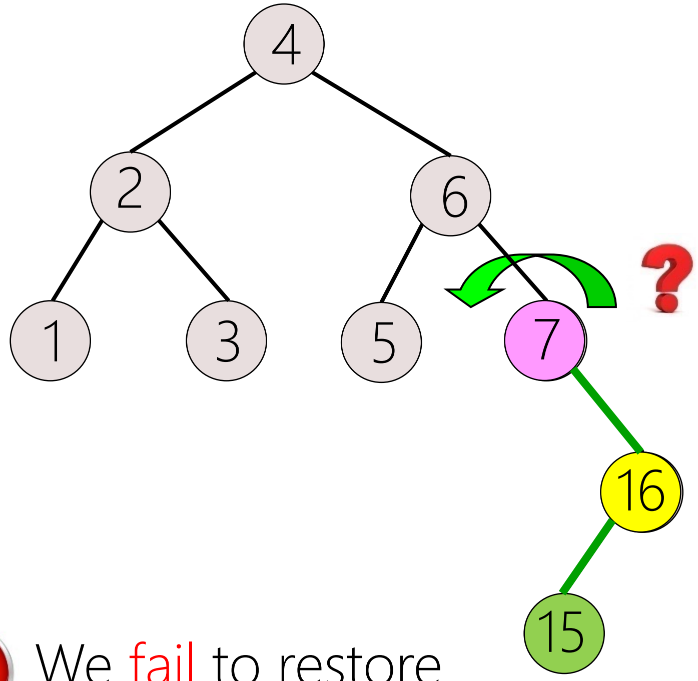


Single Rotation: Example

- Insert 16

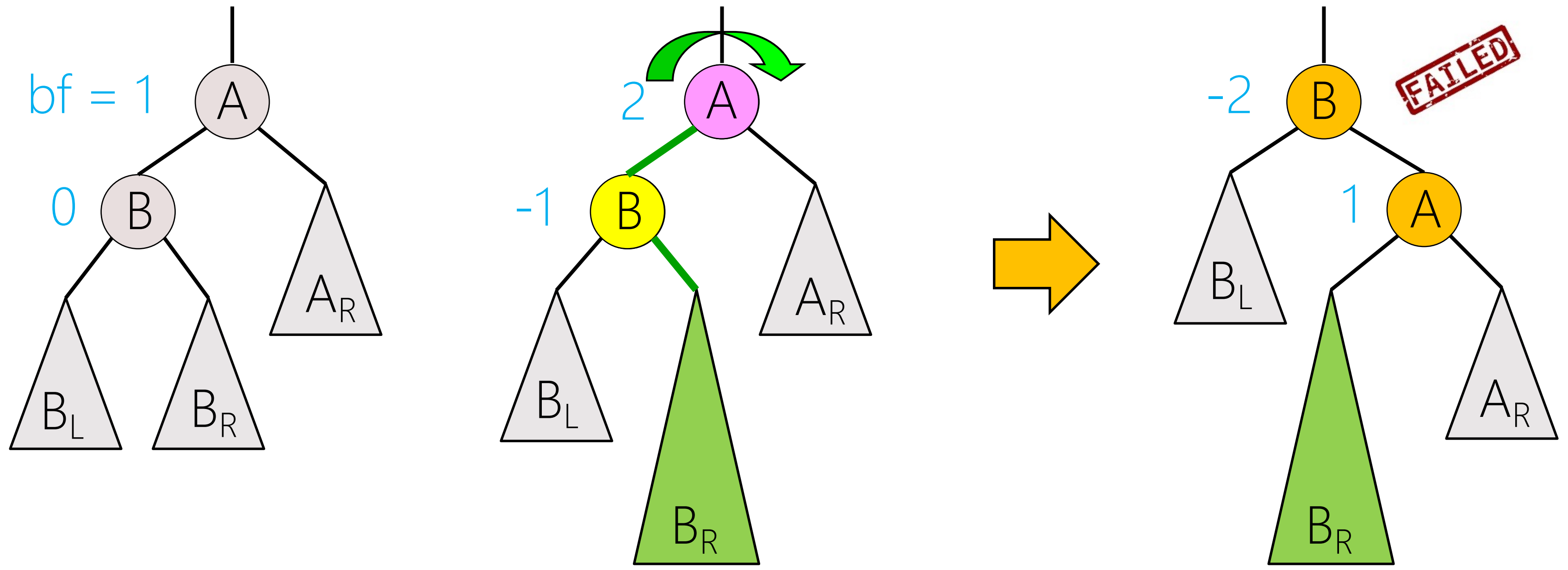


- Insert 15
  - Violation at node 7



We fail to restore the balance by single rotation

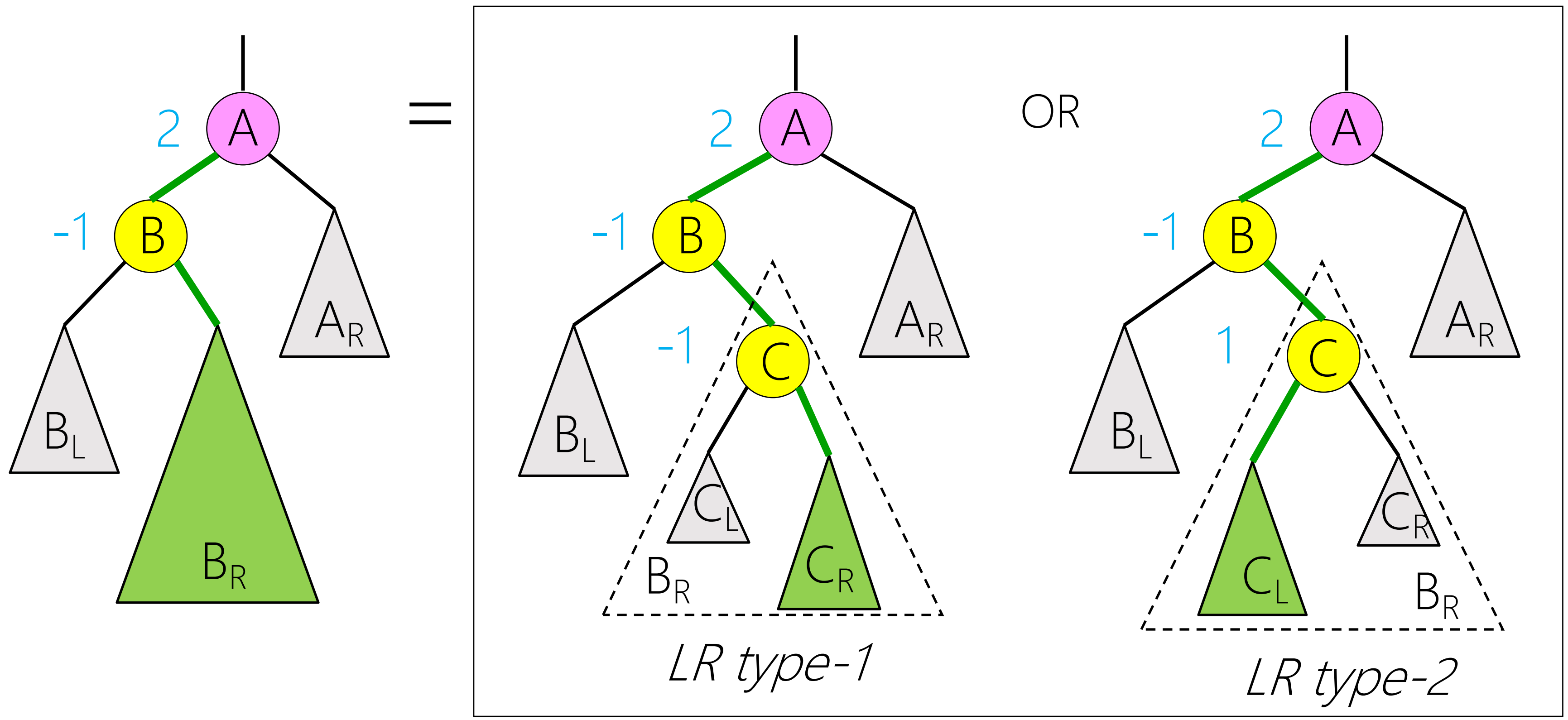
Double Rotation: LR case



Before insertion *After insertion (LR type)* *After right rotation*

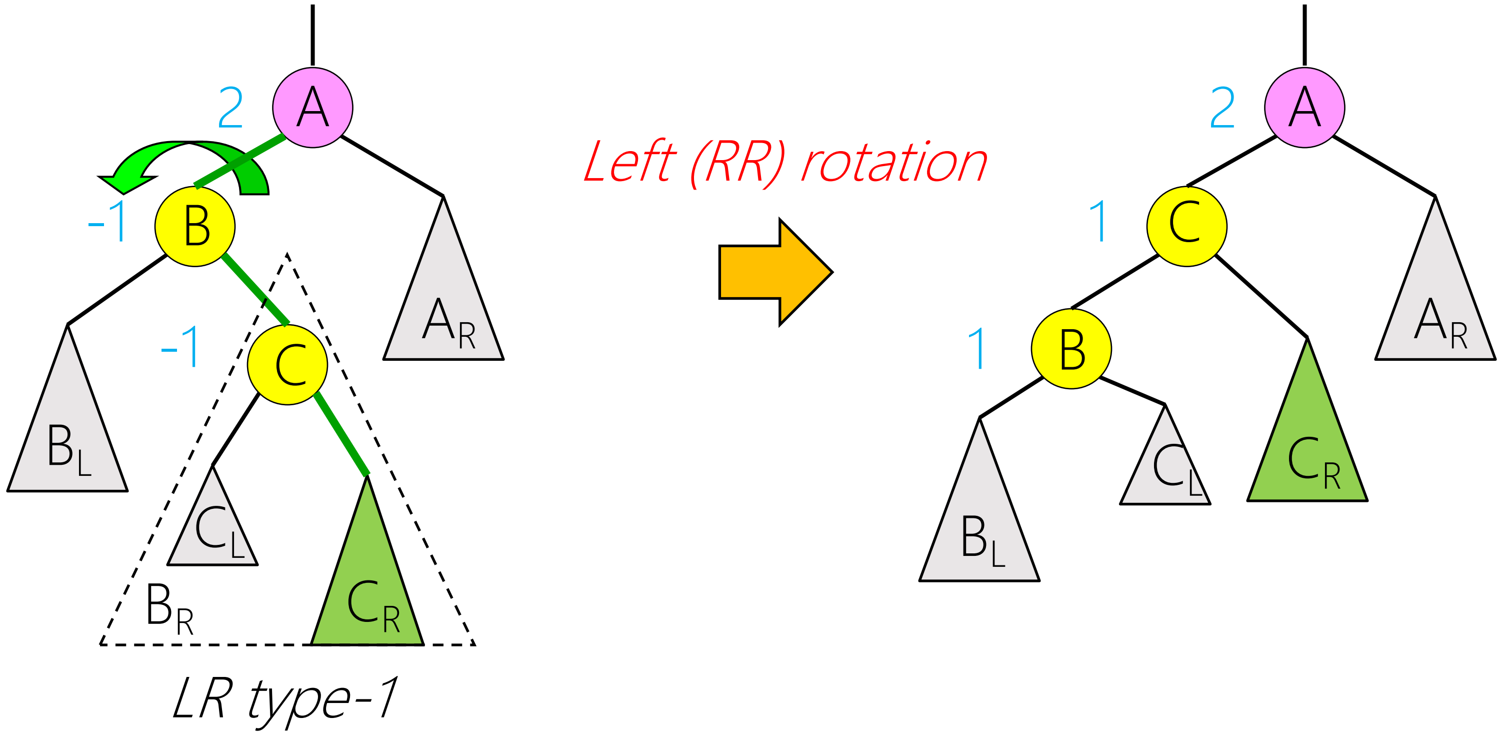
Single rotation only fails to restore the balance

Double Rotation: LR case-1 & case-2



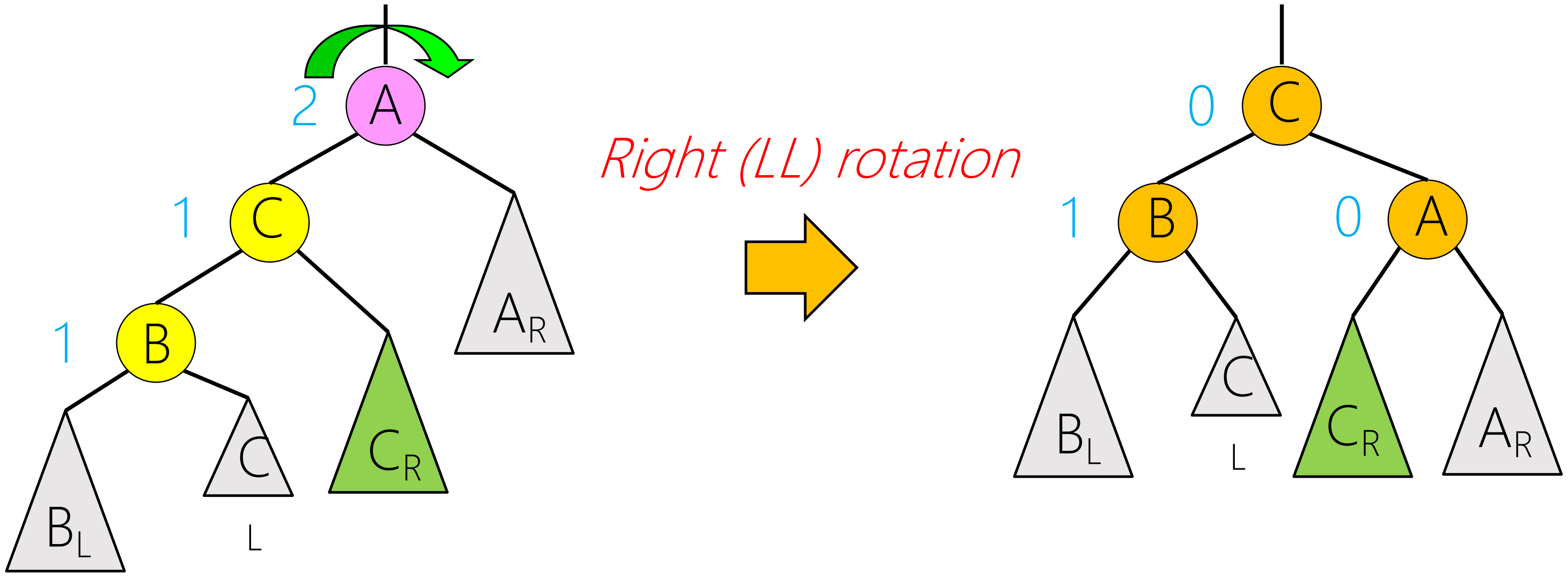
- Further examine the sub-tree B_R
  - A new node is inserted either in C_L or C_R

Double Rotation (LR): First Rotation (RR)



- First, do a single left rotation on node B as an axis (as in RR case)

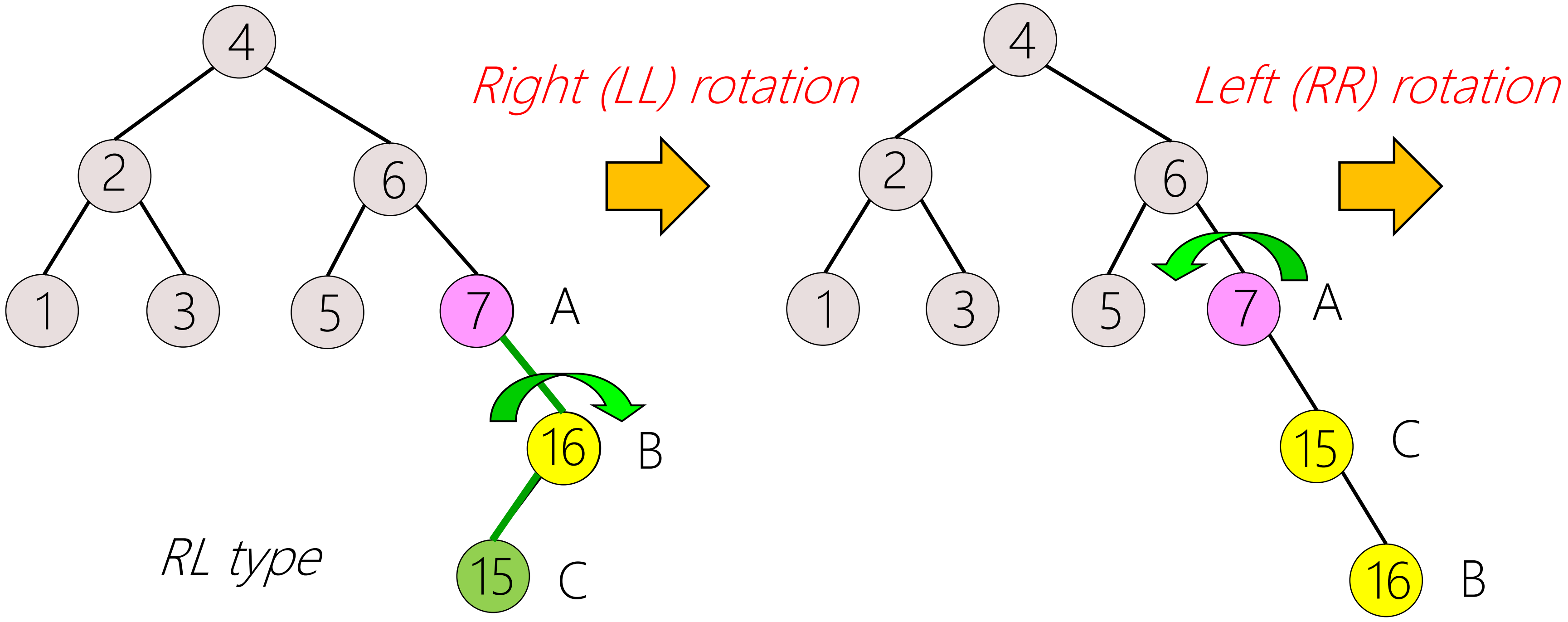
Double Rotation (LR): Second Rotation (LL)



- Then, do a single right rotation at node *A* (as in LL case)
  - Consequently LR = RR + LL : double rotation
- RL case is symmetric to LR case (RL = LL + RR)

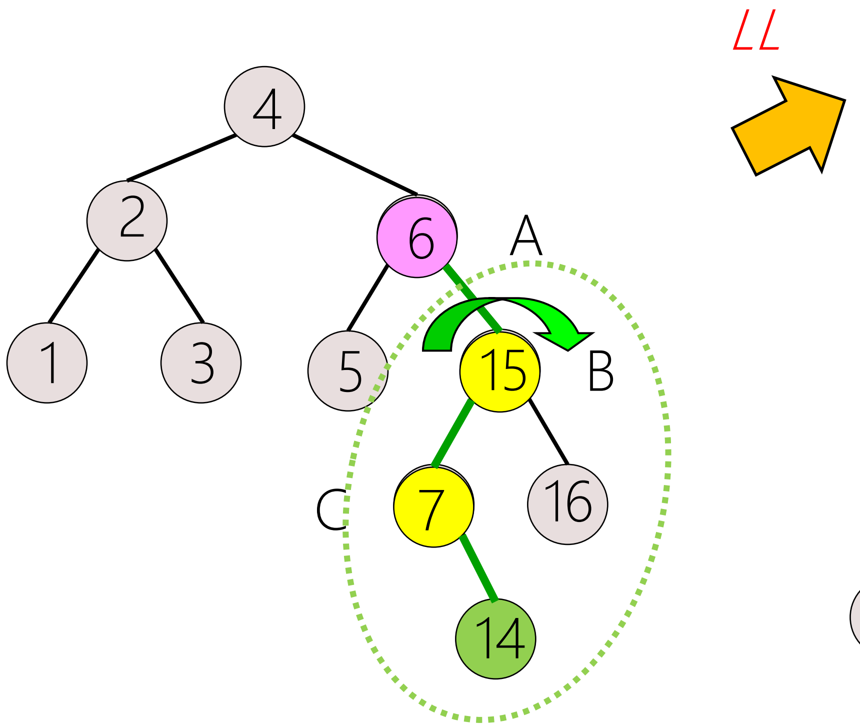
Double Rotation: Example

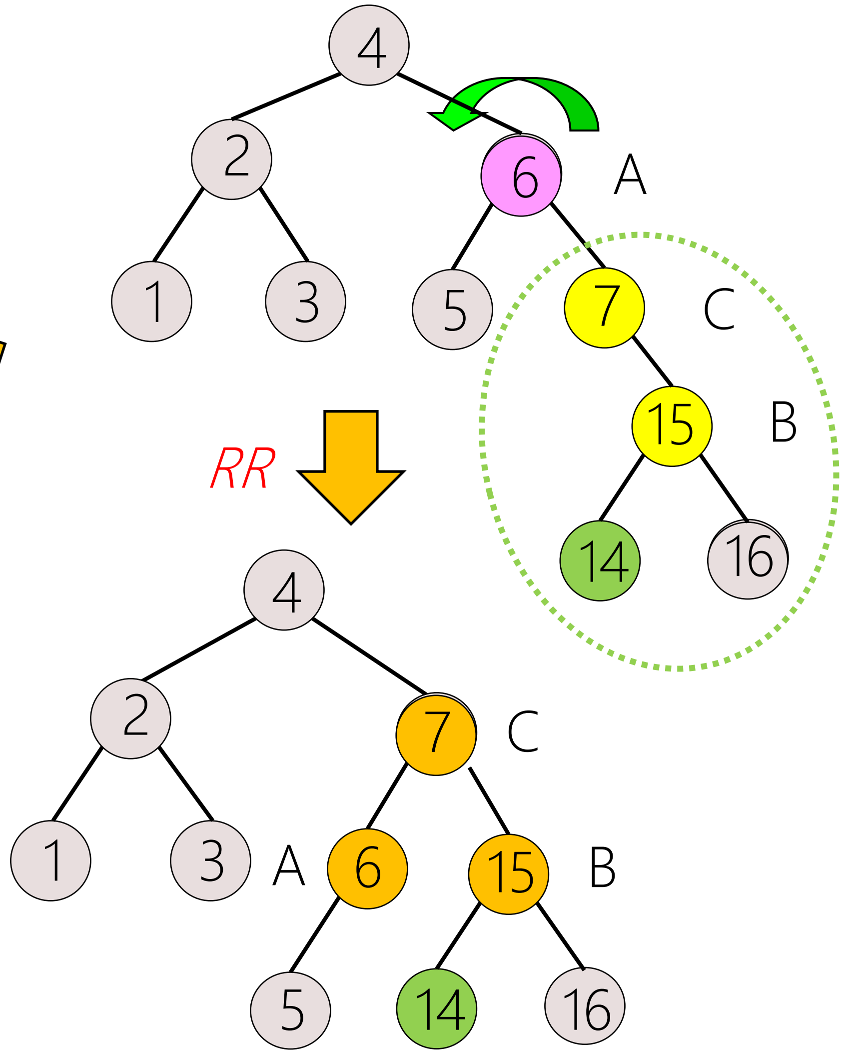
- Restart our example
  - We've inserted 3, 2, 1, 4, 5, 6, 7, 16, but we failed to insert 15
  - We'll insert 15 again



Single Rotation: Example

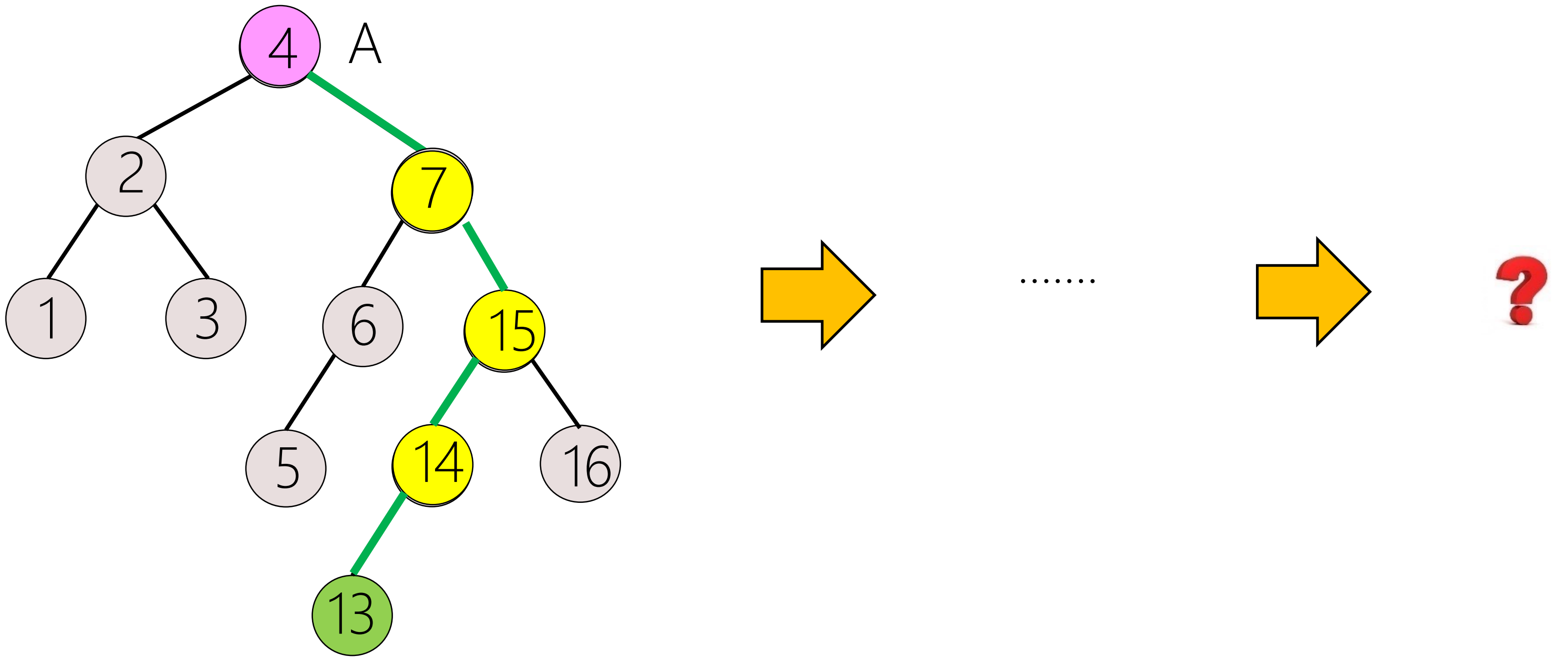
- Insert 14
  - Violation at node 6
  - RL rotation (= LL + RR)





Single Rotation: Example

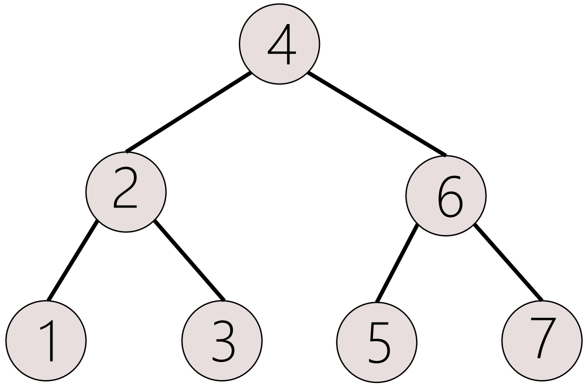
- Insert 13, 12, 11, 10, 8, 9
  - Violation at node 4
- What's the final results?



Exercise

- Insert 3, 2, 1, 4, 5, 6, 7 into an AVL tree:

- Answer:
  - We need 4 rotations when insert 1, 5, 6, 7



Time Complexity: AVL Insertion

- Time complexity of each step of AVL insertion
  - BST insertion
    - O(log *n*)
  - Checking & rotation for all ancestors of the inserted node
    - O(log *n*)
    - Checking & rotation (if needed) for a single node
      - To update the balance factor of a single node = O(1)
      - Rotation = O(1)
- Overall time complexity of insertion is: O(log *n*)

Review: Deletion in BST

- There are 3 cases of deletion from
  - *Leaf* node
  - *Degree-1* node (with one child)
  - *Degree-2* node (with two children)
    - The smallest key must be in a *leaf* or *degree-1* node

- In all cases, the last (deepest) deleted node is a
  - *leaf* node or *degree-1* node

Deletion from AVL Tree

- First, perform BST deletion
- Then, trace the path from the last deleted node towards the root
  - For each node x, check if AVL property ($-1 \le bf(x) \le 1$)
    - If yes
      - Proceed to parent(x)
    - If no,
      - Do an appropriate *rotation* at node x
      - Details (in next slide)
  - Continue to trace the path until we reach the root

What Rotation (after BST Deletion)?

- If AVL property is violated at node x
  - *If h(x.left) = h(x) - 1 (that is, h(x.right) = h(x) - 3)*
    - *If h(x.left.left) = h(x) – 2* → LL rotation(x)
    - *Else If h(x.left.right) = h(x) – 2* → LR rotation(x)
  - *If h(x.right) = h(x) - 1 (that is, h(x.left) = h(x) - 3)*
    - *If h(x.right.right) = h(x) – 2* → RR rotation(x)
    - *Else If h(x.right.left) = h(x) – 2* → RL rotation(x)

Note:

- The above is the same as in the insertion case

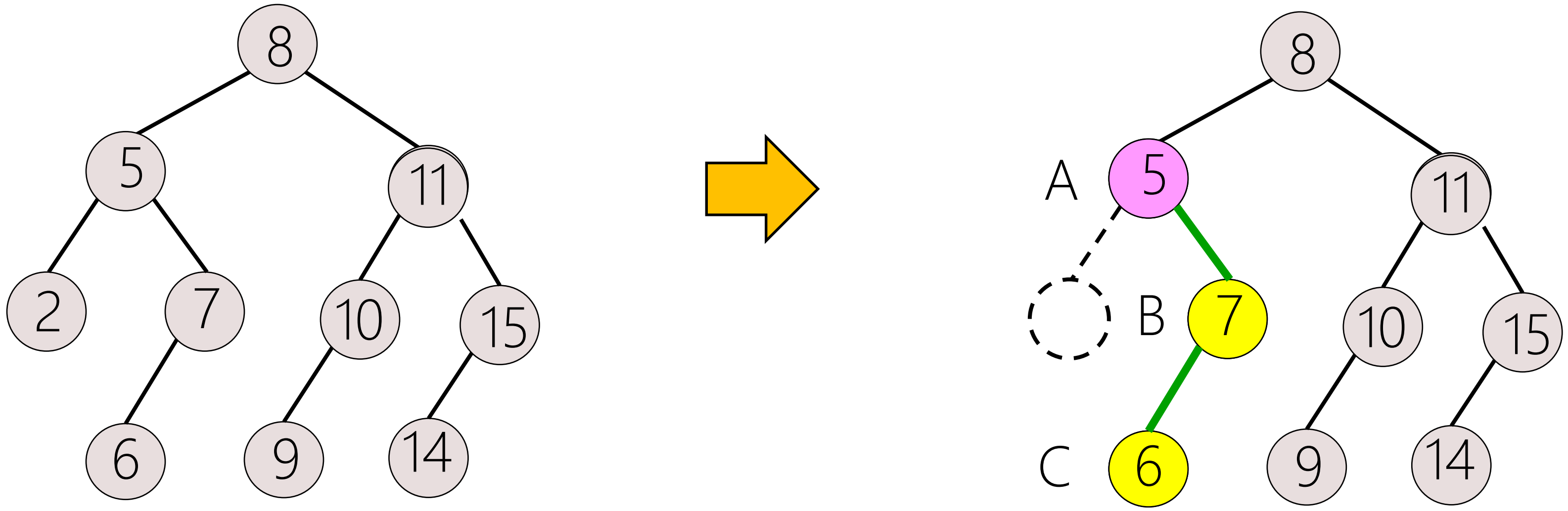
What Rotation?: Interpretation

- For a given pivot (*A*),
  - Let *B* be the child of *A* with the longer height
  - Let *C* be the child of *B* with the longer height
    - If two children of *B* are of the same height, then *C* must be chosen in favor of single rotation

- Determine the type of imbalance based on *A-B-C*
    - *LL, LR, RR, or RL*

Delete (from Leaf): Example

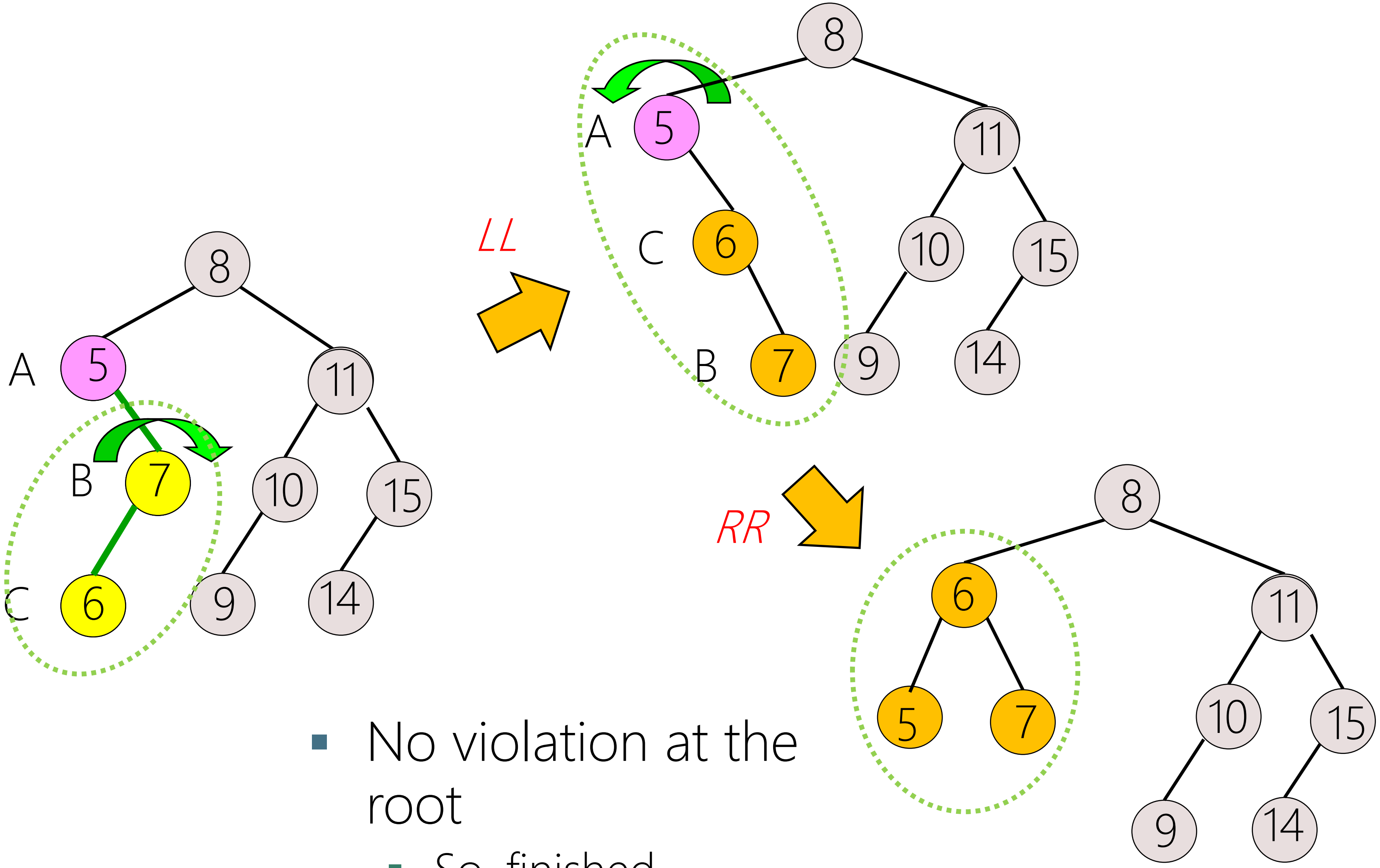
- Delete 2 (leaf node)
  - Violation at node 5



RL type of imbalance

Delete (from Leaf): Example

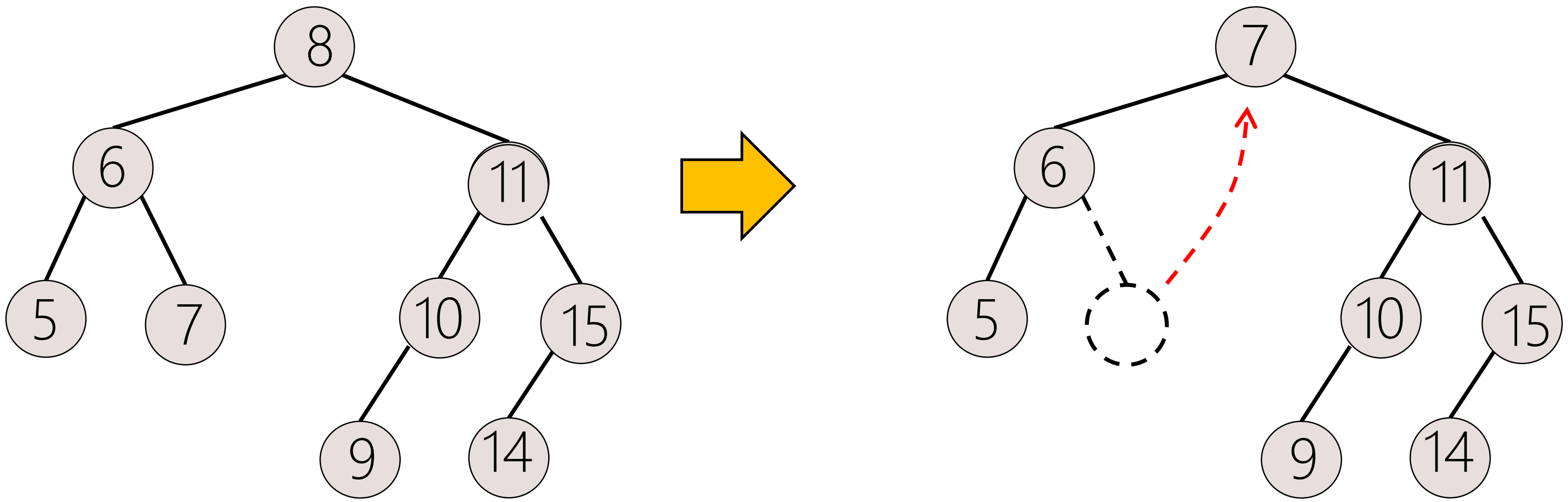
- RL rotation (= LL + RR)



- No violation at the root
  - So, finished

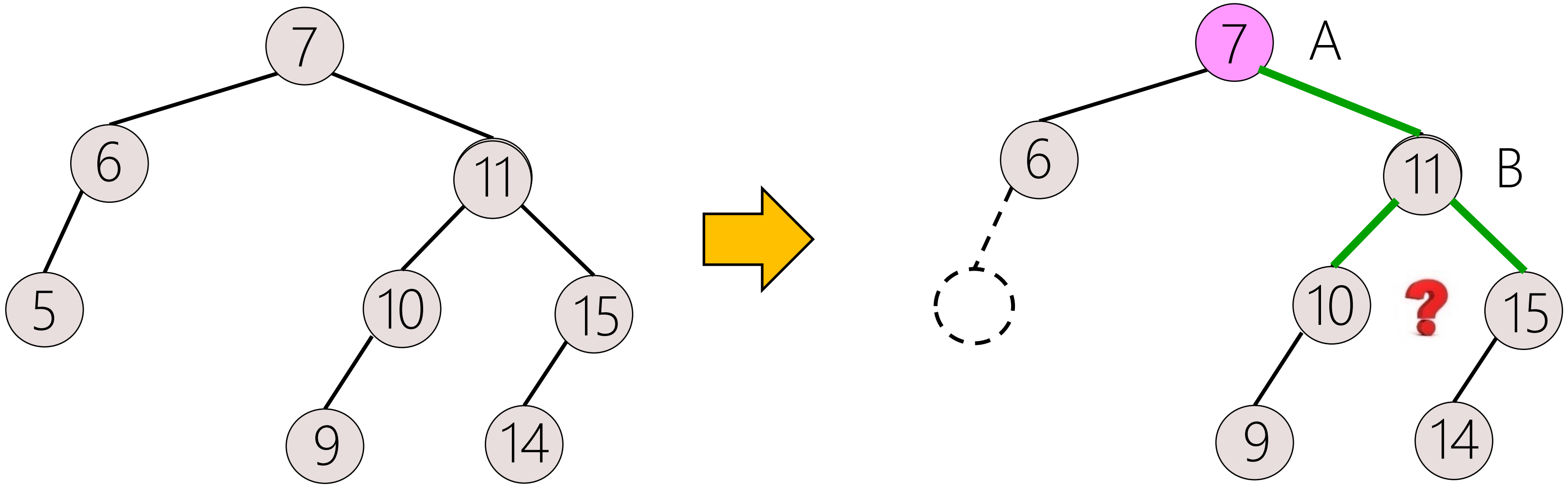
Delete (from Deg-2 Node): Example

- Delete 8 (degree-2 node)
  - No violation at node 6, 7
  - So, finished



Delete: Children of the Same Height

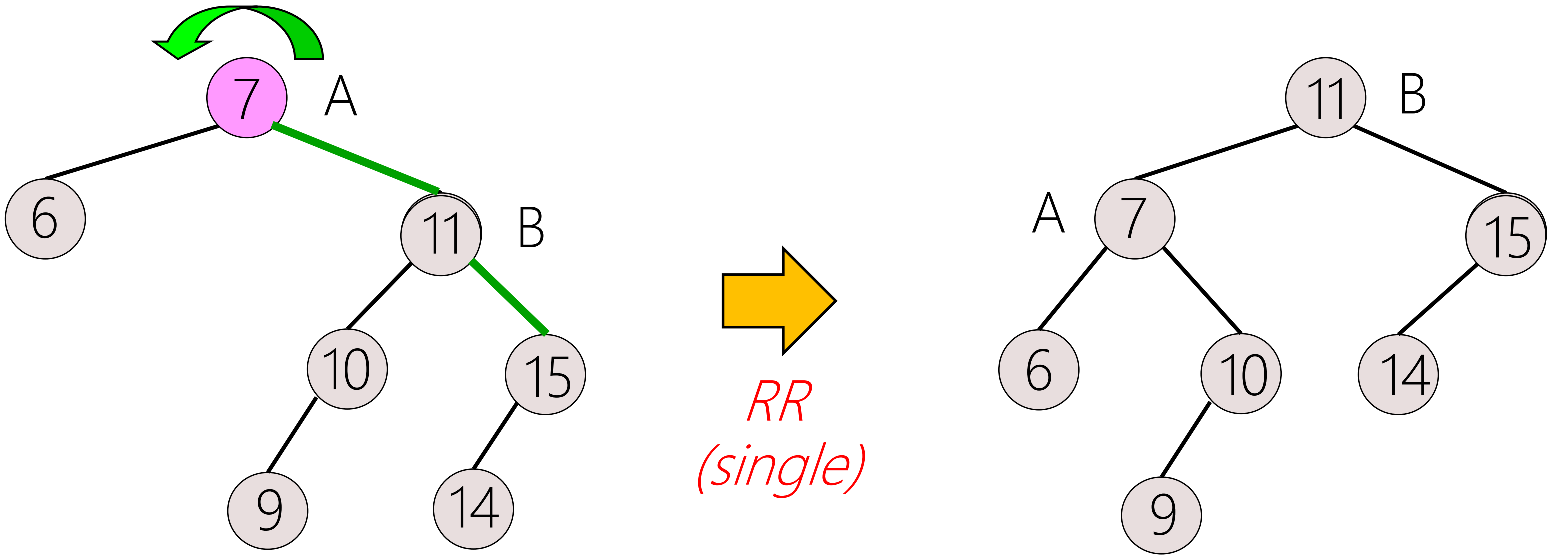
- Delete 5 (leaf node)
  - Violation at node 7



- Which type, RR or RL?(when two subtrees of B are of the same height)

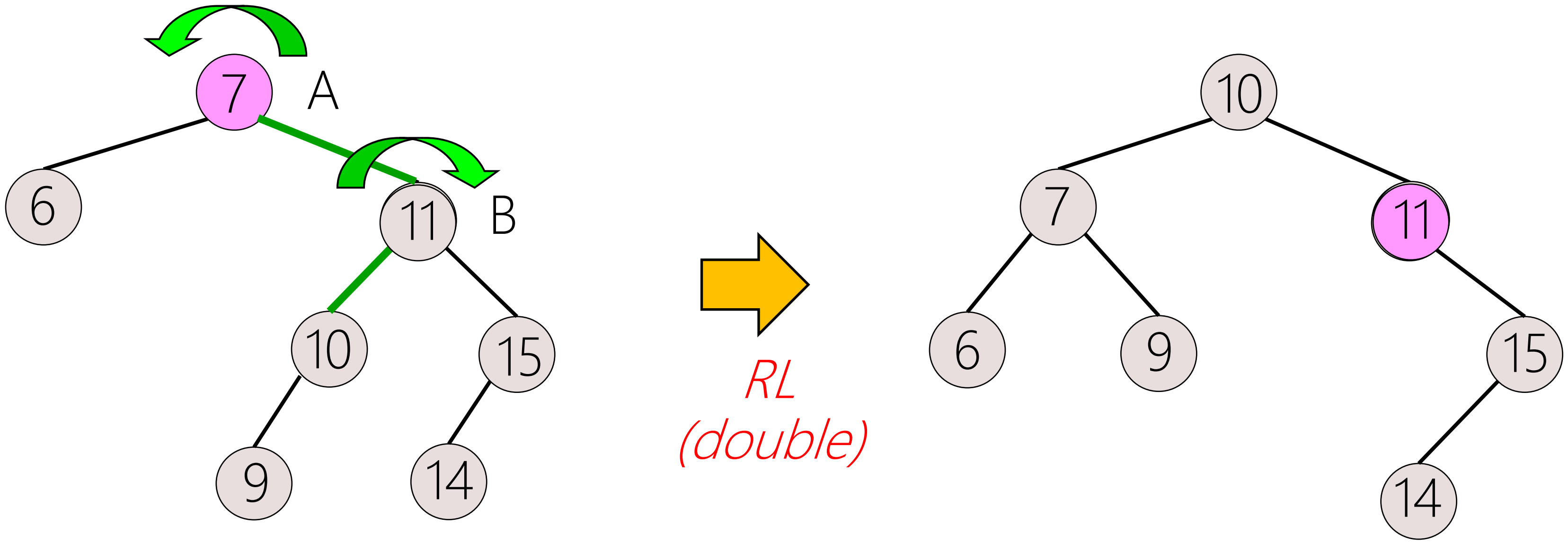
Delete: Children of the Same Height (1)

- When RR type (single rotation) is chosen:



Delete: Children of the Same Height (2)

- When RL type (double rotation) is chosen:



- Still violation at node 11

➔ Fail

Delete: Children of the Same Height (3)

- As a consequence
  - If two sub-trees of B are of the same height
  - Then a single rotation must be chosen

References

- Further reading list and references
  - https://en.wikipedia.org/wiki/AVL_tree
  - https://www.geeksforgeeks.org/avl-tree-set-1-insertion/

- Slide credit
  - Jaesik Park
  - Seung-Hwan Baek
  - Jong-Hyeok Lee