# 04-630
# Data Structures and Algorithms for Engineers

## Lecture 12: Height-Balanced Trees: AVL Trees

Theophilus A Benson

# Today!!

- Conclude Binary Search Tree lecture
  - BST Delete
  - Use cases for traversals

- AVL Tree ➔ Balanced BST
  - Motivation
  - Definition of Balancing (versus height)
  - Maintaining Balance: Rotation operations
  - Use cases for AVL over BST

- Get to Bus on time!!

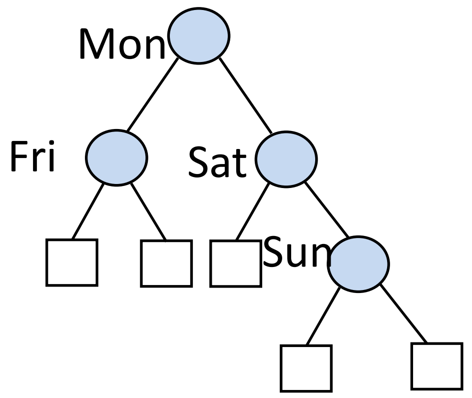
# Implementation of *Delete*($e$, $T$) – Step 1: finding the node e to delete

- If $T$ is not empty
  - if $e$ < element at root of $T$

    Delete $e$ from left child of $T$: *Delete(e, T(1))*
  - if *e* > element at root of $T$

    Delete *e* from right child of $T$: *Delete(e, T(2))*
  - if $e$ = element at root of $T$ and both children are empty

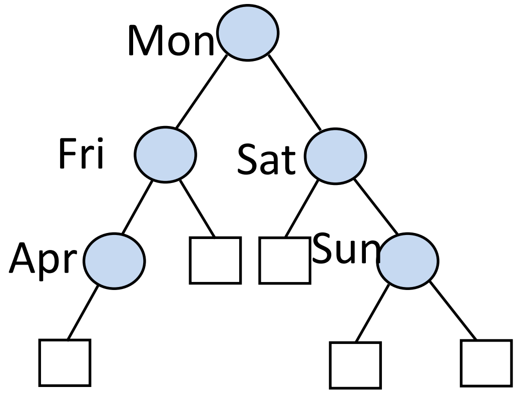
    Remove $T$



# Implementation of *Delete*(*e*, *T*) – Step 2: finding e's replacement

– if $e$ = element at root of $T$ and left child is empty

Replace $T$ with $T(2)$



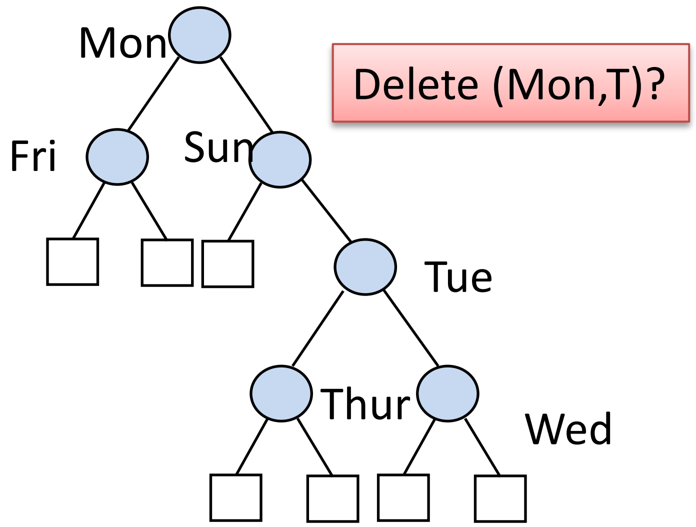

– if $e$ = element at root of $T$ and right child is empty

Replace $T$ with $T(1)$

– if $e$ = element at root of $T$ and neither child is empty

Replace $T$ with left-most node of $T(2)$ ← *"left-most node in right sub-tree!"*

# Deleting from a "BST"




- Goal: "we replace the node at $w$ with the lowest-valued element among the descendents of its right child"

- Which Traversal to use to find the replacement for Mon?
  - In-order: left -> root-> right
  - Post-order: left -> right->root
  - Pre-order: root -> left -> right

In-order of Tree @ Sun: Sun, Thur, Tue, Wed

Post-order of Tree @ Sun : Thurs, Wed, Tue, Sun

Delete one node

Delete one subtree

# Applications of BST Traversals

In-order

Process the nodes in order: Smallest to largest, e.g., find smallest node in a tree

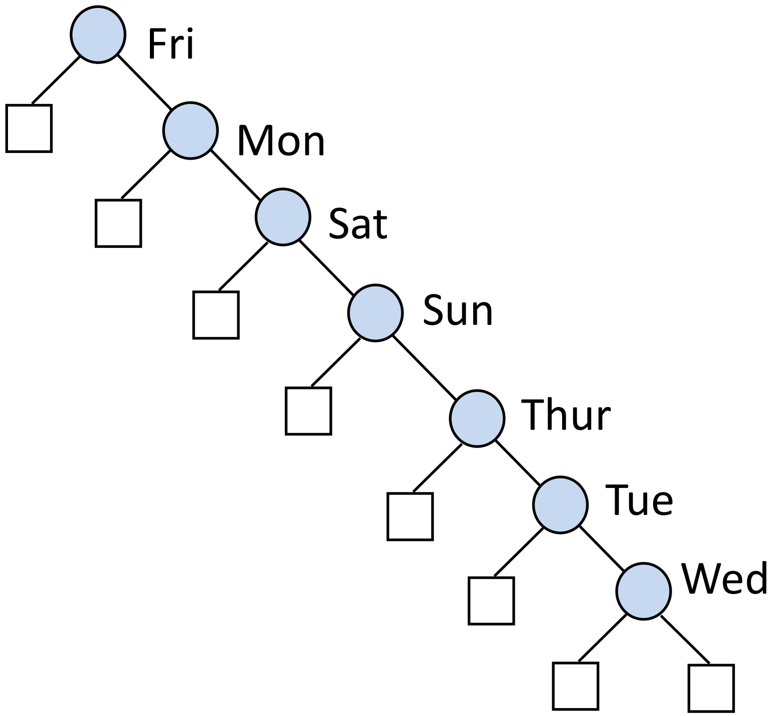
Post-order:

Process the leaves (dependencies) first, e.g., file system or memory management

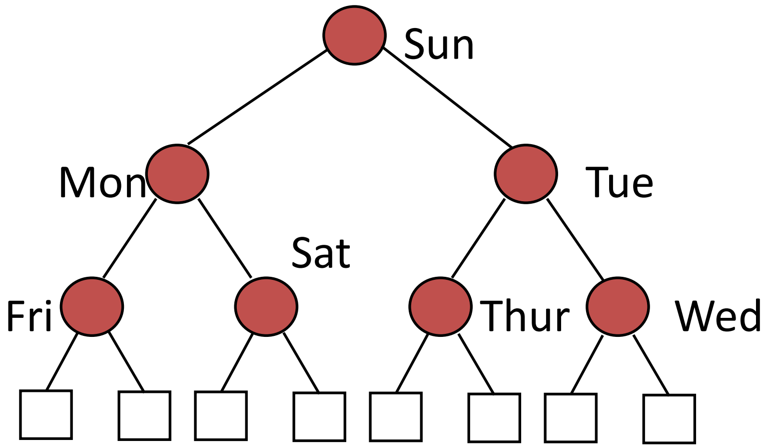
Pre-order

Process the root then propagate value to the leaves, e.g., DB search and indexing

# BST Time Complexity



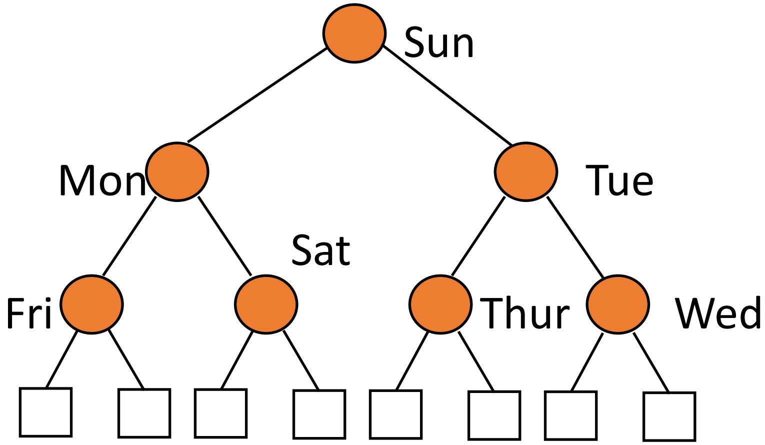

Insert: O(N)
Search: O(N)
Delete: O(N)




Insert: O(logN)
Search: O(LogN)
Delete: O(logN)

# Enter Adelson-Velsky Landis Trees
# AKA AVL
# AKA Balanced BST

# Height-balanced Trees




- Goal: the tree is as complete as possible and has minimal height for the number of nodes in the tree

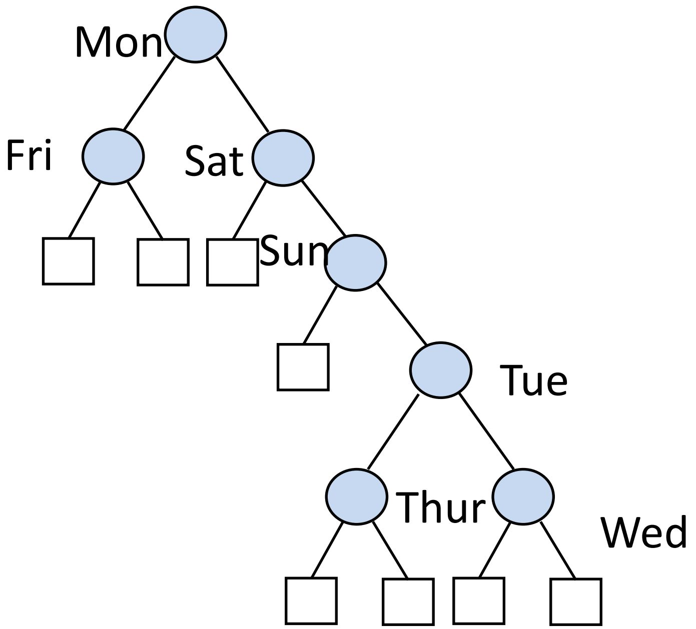
- Minimizes the # of probes to search the tree
  - # of probes == time

- balance is based on heights of subtrees.

- Insertions and deletions should be made such that the tree ***starts off*** and ***remains*** height-balanced.

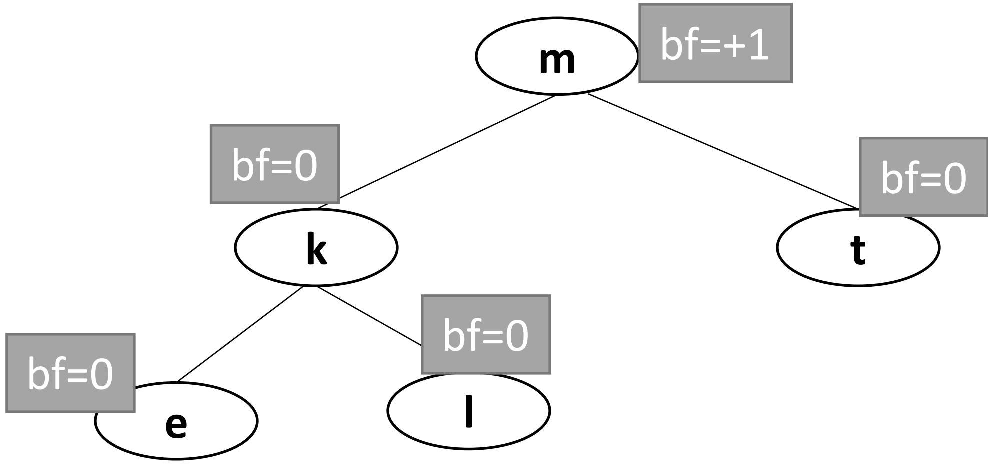
# AVL Trees: BSTs with a *balance condition*.

- **AVL tree:** a BST where the height of the left and right subtrees can differ by at most 1, i.e.:
  - $|height(T_L)-height(T_R)|\leq 1$.
  - Height information is kept for each node in the AVL tree.
- **balance factor:** height requirement of the subtrees
  - Must differs by at most 1.
  - maintained even after insertions/deletions.
  - Maintenance achieved through ***rotation***.
- AVL tree code == same code as BST with additional benefits
  - New structure to store height
  - Insert/delete need additional code to rotate



# Recall: Binary Tree basics

- Height Numbering
  - Number all external nodes 0
  - Number each internal node to be one more than the maximum of the numbers of its children
  - Then the number of the root is the height of $T$



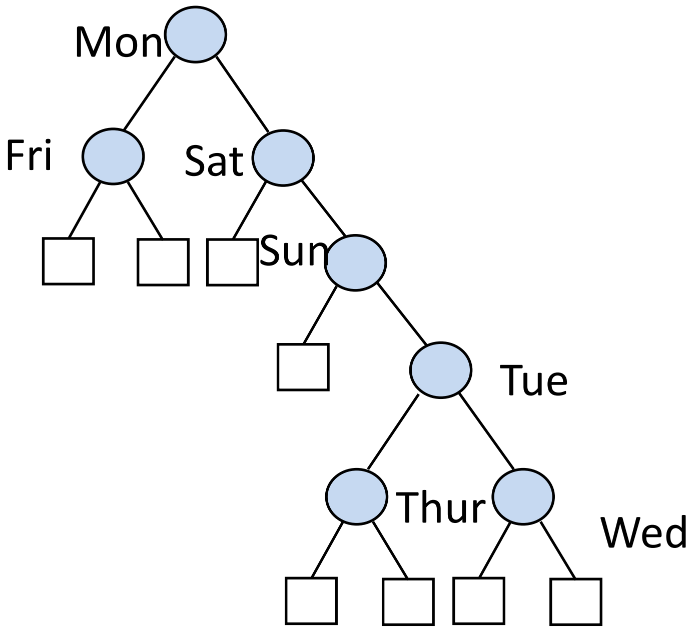

- The height of a node $u$ in $T$ is the height of the subtree rooted at $u$

- AVL : Balance Factor (bf) = |height($T_L$)-height($T_R$)|
  - Height(T*) returns -1 is subchild is empty

# AVL Trees: Definition

What is height of sun?
What is BF of sun?
- What is height(TL of sun)?
- What is height(TR of sun)?

1. An empty tree is height-balanced.
2. If T is a non-empty binary tree with left and right sub-trees $T_L$ and $T_R$, then T is height-balanced iff:
   a) $T_L$ and $T_R$ are height-balanced, and
   b) $|height(T_L)-height(T_R)|\leq 1$.
   c) Height(T*) returns -1 if is subchild is empty
3. Every sub-tree in a height-balanced tree is also height-balanced.



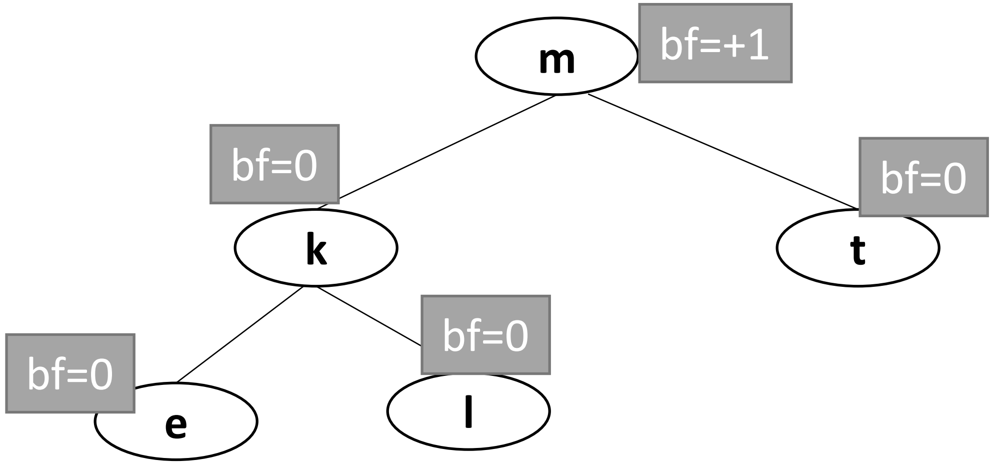
# AVL Trees: Balance Factor

- Balance Factor $BF(T)$ of a node $T$ in a binary tree is defined to be

  $height(T_L)$ - $height(T_R)$

  where $T_L$ and $T_R$ are the left and right subtrees of $T$

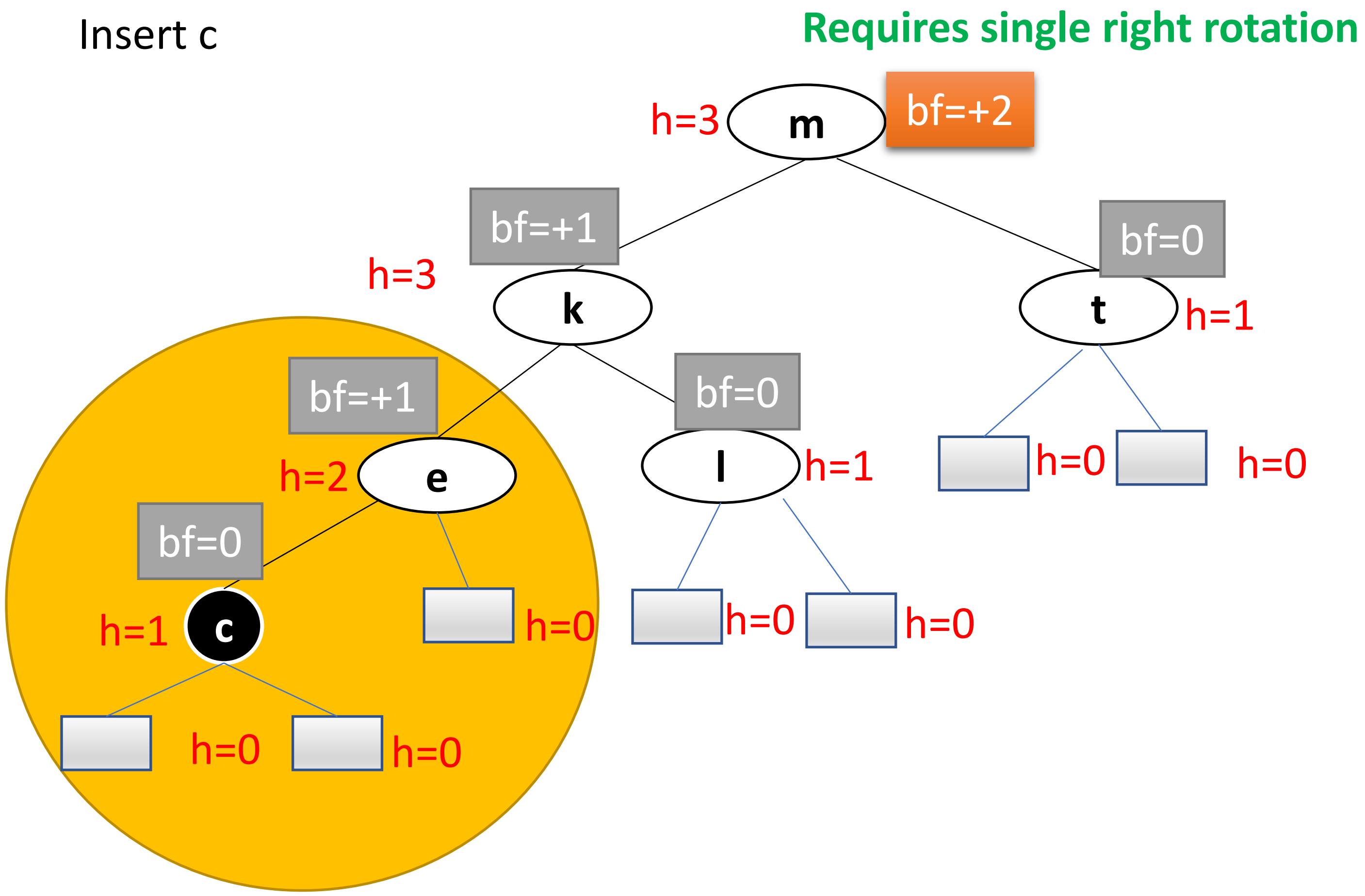
- For any node $T$ in an AVL tree $BF(T) = -1, 0, +1$



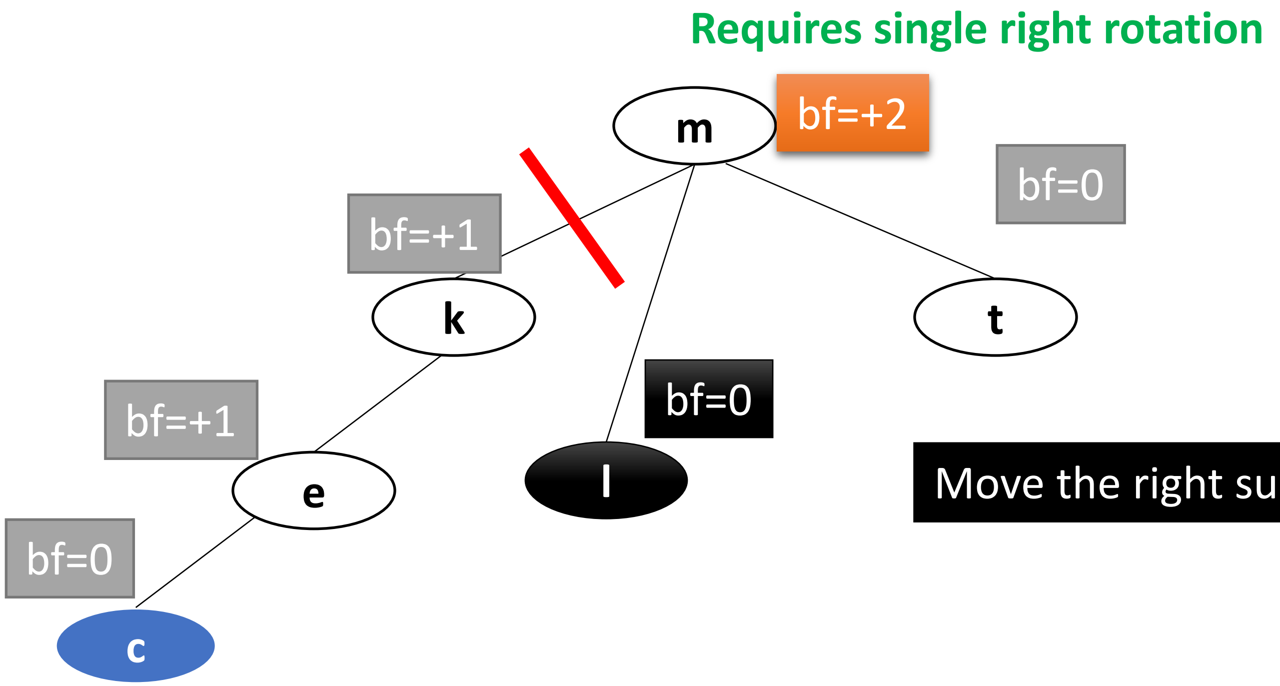

Why is bf of t 0?

# Example: Rebalancing after insertion

Insert c



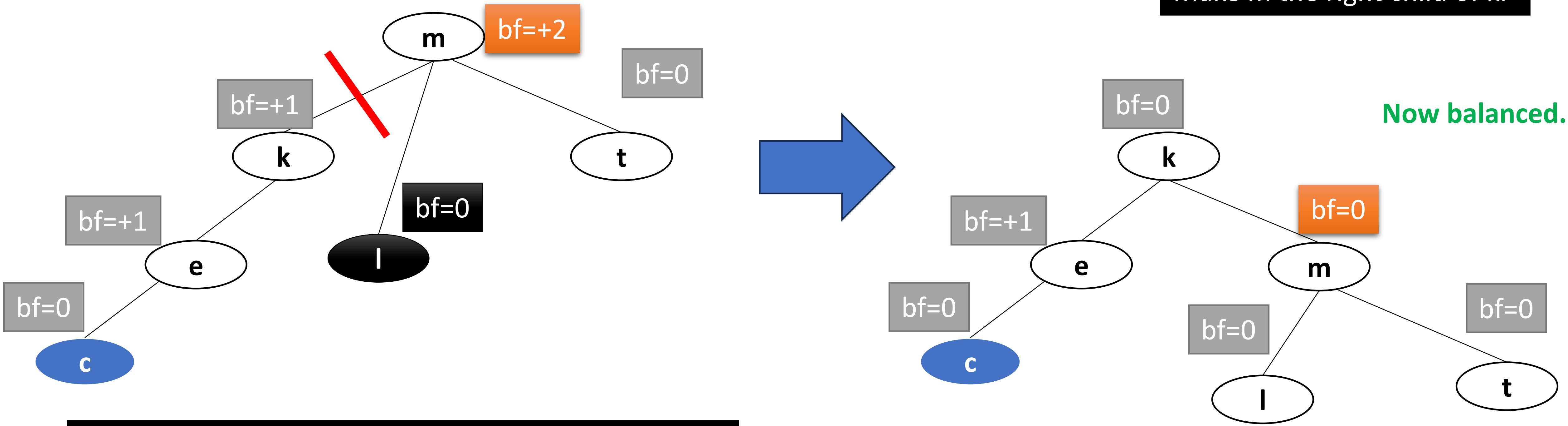
# Rebalancing after insertion




Move the right sub-tree of k to the left subtree of m.

# Rebalancing after insertion

Make m the right child of k.

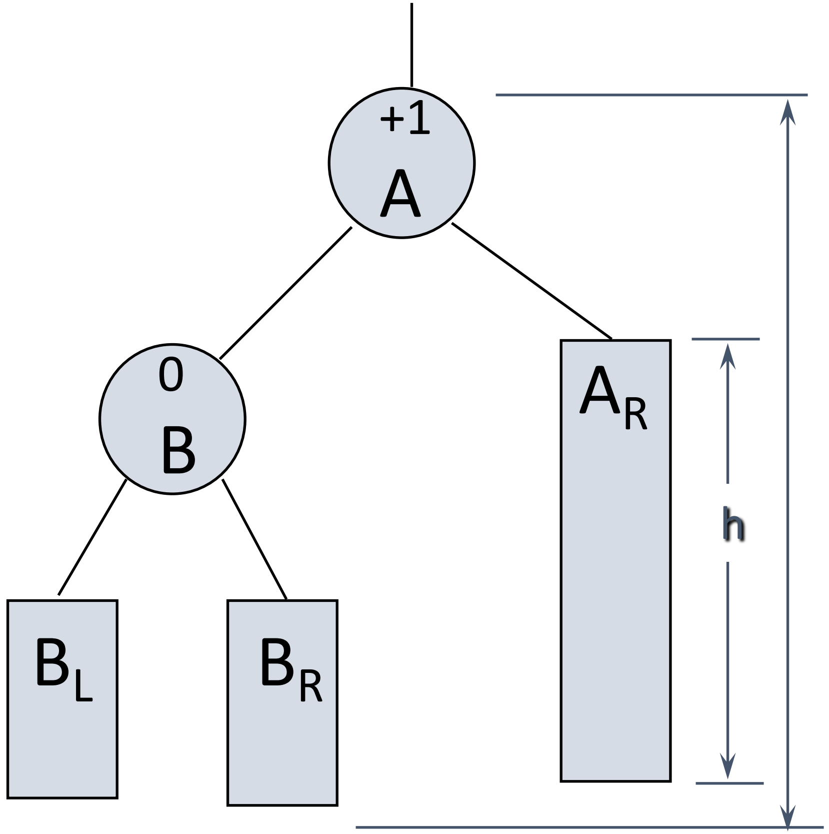



Now balanced.

Move the right sub-tree of k to the left subtree of m.

# AVL Trees

- All re-balancing operations are carried out with respect to the closest ancestor of the new node having balance factor +2 or -2
- Let's refer to the node inserted as Y
- Let's refer to the nearest ancestor having balance factor +2 or -2 as A
- There are 4 types of re-balancing operations (called rotations)
  - LL
  - RR(symmetric with LL)
  - LR
  - RL (symmetric with LR)



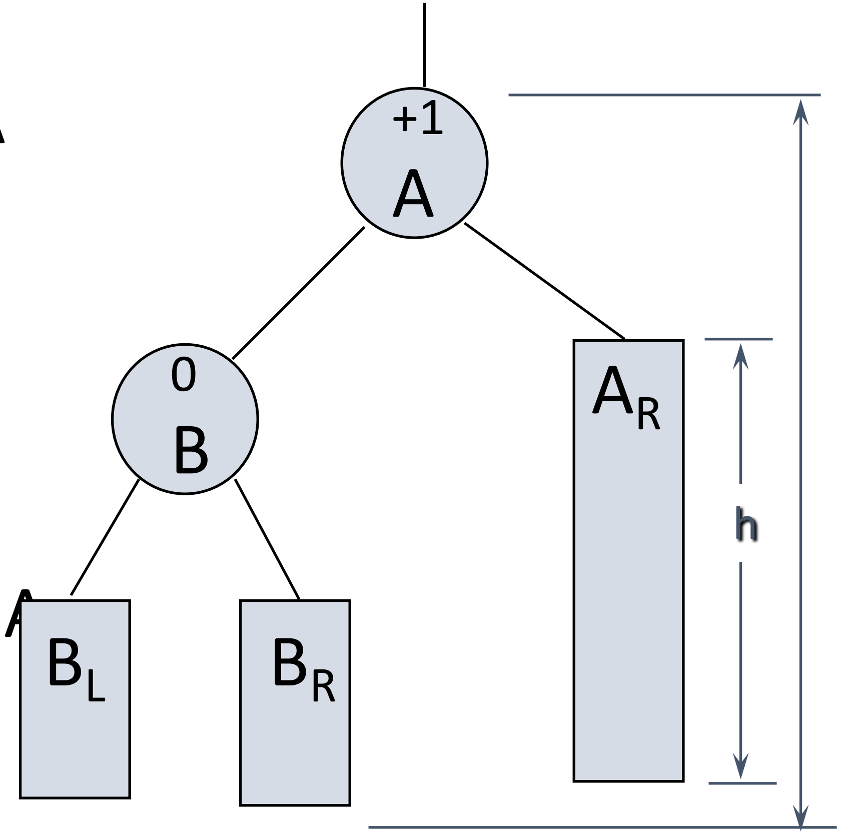

Why is balance factor of A is +1

height($T_1$)-height($T_2$)=+1

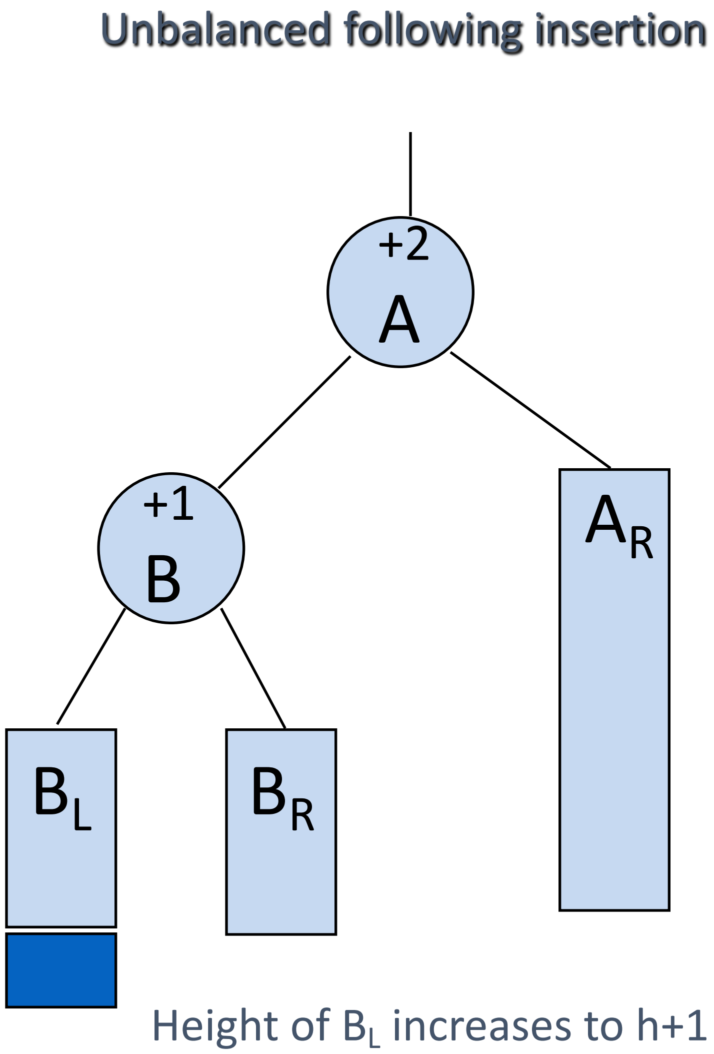
Can only mean height($T_1$)=h+1

# AVL Trees

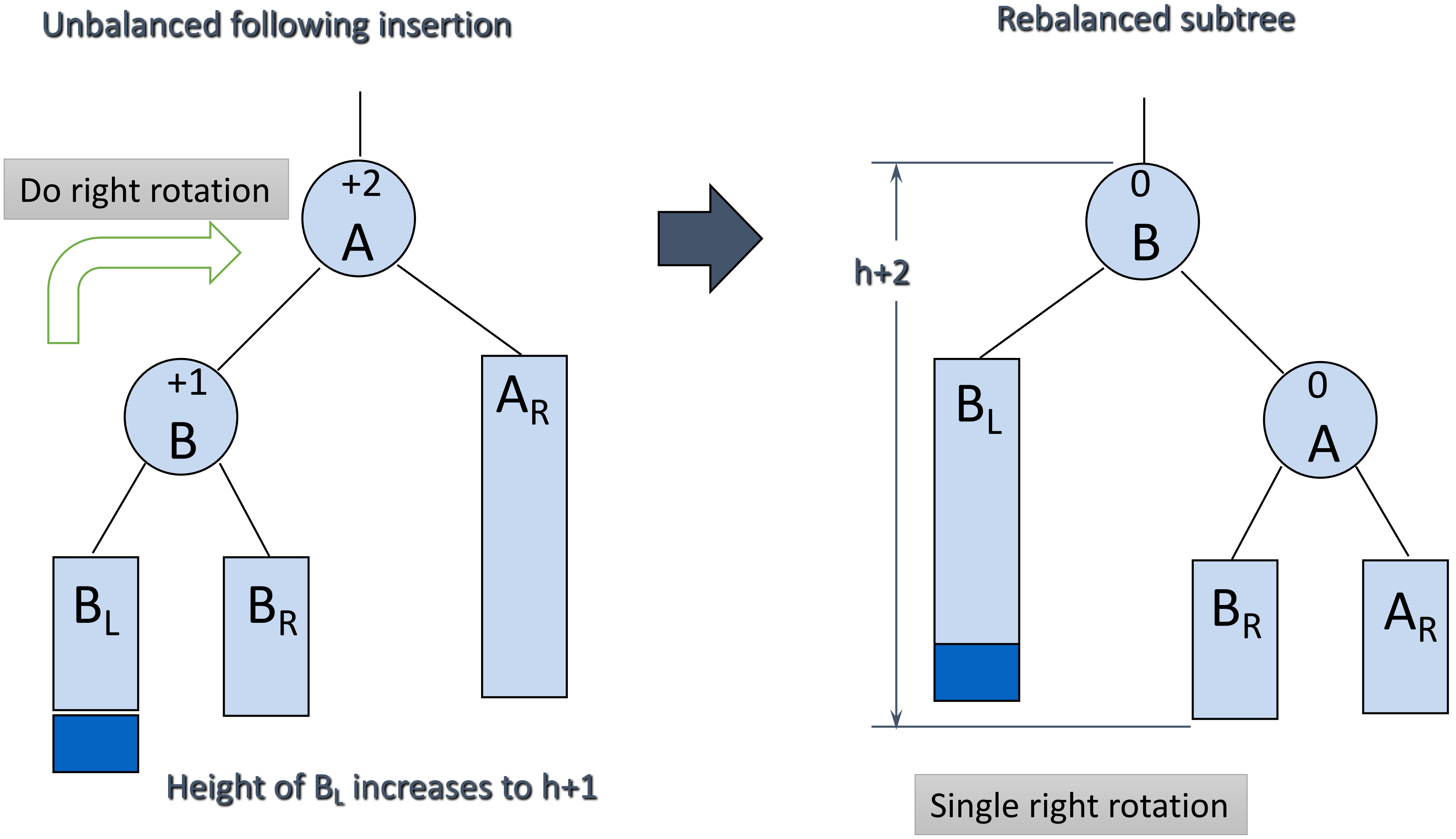
- LL: Y is inserted in the Left subtree of the Left subtree of A
  - LL: the path from A to Y
  - Left subtree then Left subtree
- LR: Y is inserted in the Right subtree of the Left subtree of A
  - LR: the path from A to Y
  - Left subtree then Right subtree



# AVL Trees


Unbalanced following insertion
+2
A
+1
B
$A_R$
$B_L$
$B_R$
Height of $B_L$ increases to h+1

# AVL Trees - LL rotation(Outside case- Case 1)



# AVL Trees

- RR: Y is inserted in the
  Right subtree of the Right subtree of A

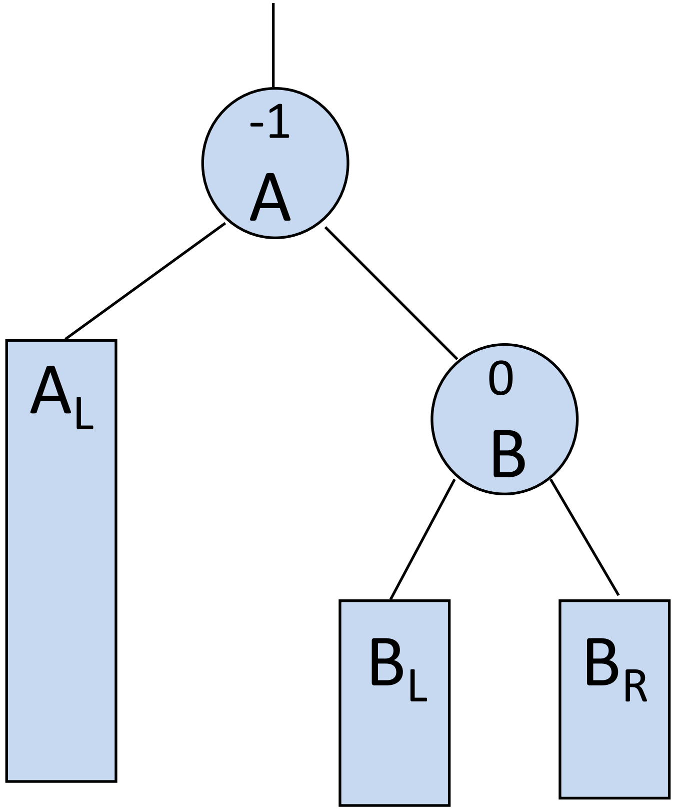
  - RR: the path from A to Y
  - Right subtree then Right subtree

- RL: Y is inserted in the
  Left subtree of the Right subtree of A

  - RL: the path from A to Y
  - Right subtree then Left subtree

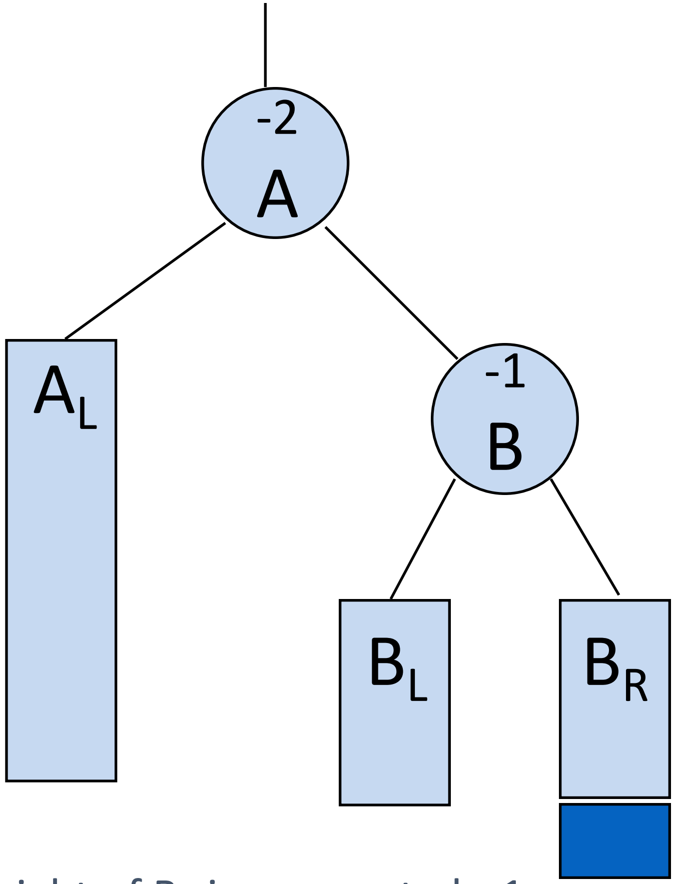
# AVL Trees

Balanced Subtree



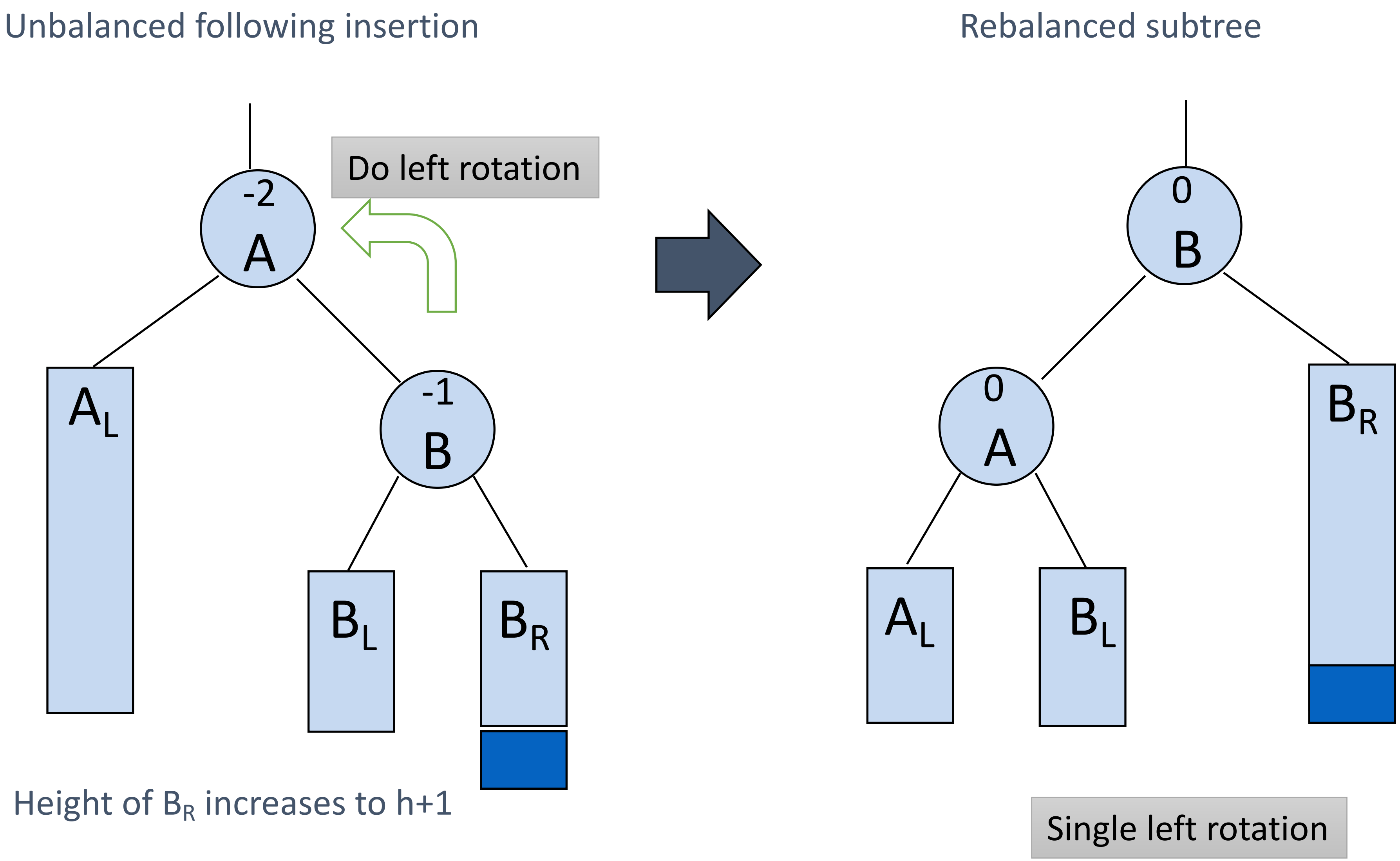
# AVL Trees

Unbalanced following insertion



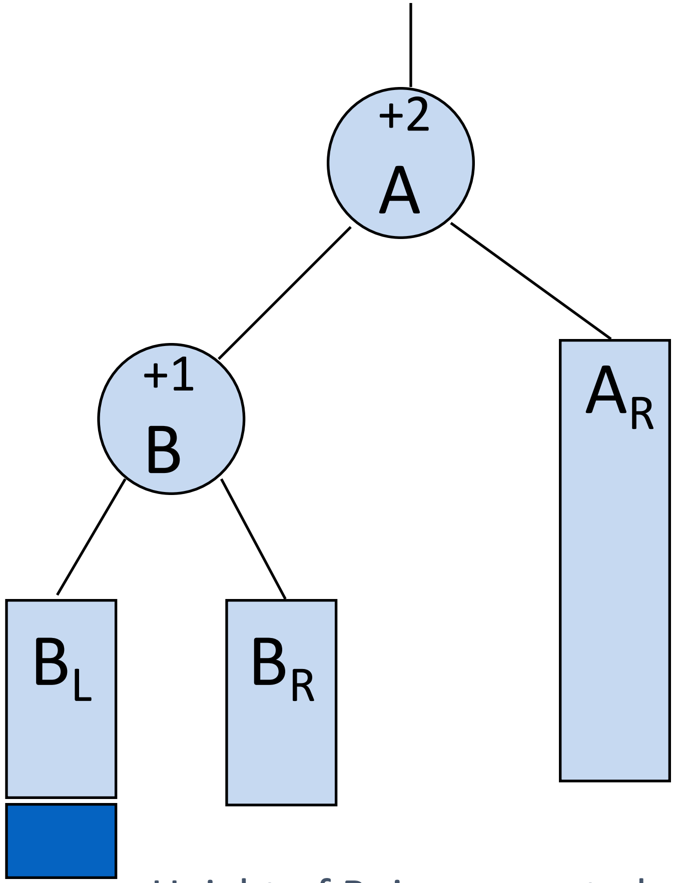

Height of $B_R$ increases to h+1

# AVL Trees - RR Rotation(Outside case- Case 2)



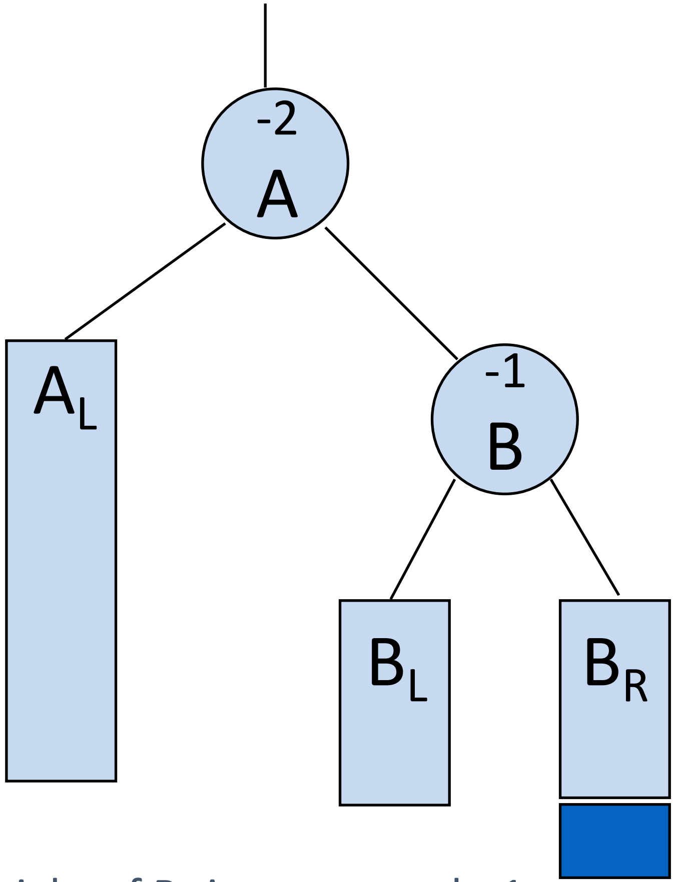
# Type 1 (+2: left imbalance → Rotate right) Versus Type 2 (-2: right imbalance → rotate left)

Unbalanced following insertion




Height of $B_L$ increases to h+1

Unbalanced following insertion




Height of $B_R$ increases to h+1

# Rebalancing cases: 4 types of rotations always focused on unbalanced root and two of its descendants

Consider **k** to be the node to be rebalanced.

Inserting into the left sub-tree of the left child of k [Case1--LL]

**Requires single right rotation**

Inserting into the right sub-tree of the right child of k[Case 2-RR]

**Requires single left rotation**

Outside cases

Inserting into the right sub-tree of the left child of k.[Case 3-LR]

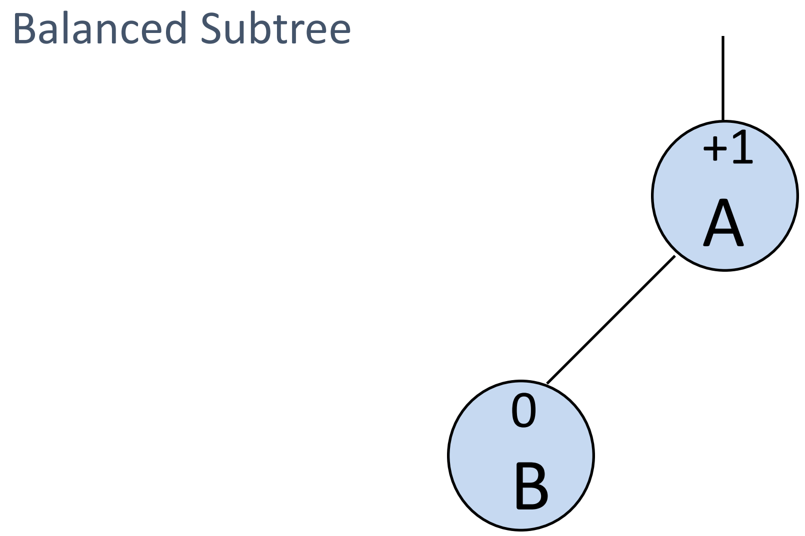
**Do a double rotation: left then right.**

Inserting into the left sub-tree of the right child of k.[Case 4-RL]

**Do a double rotation:  right then left**

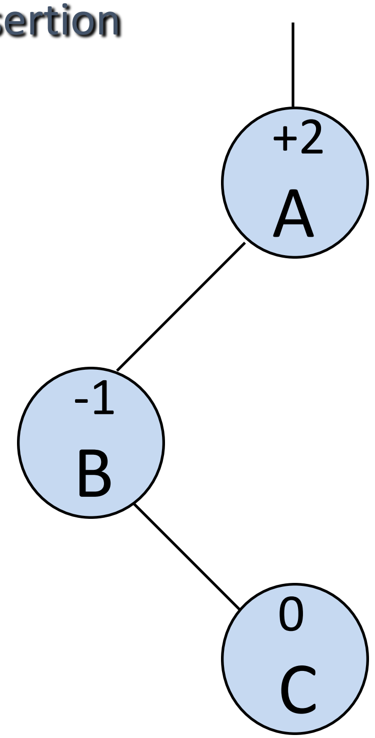
Inside cases

# AVL Trees



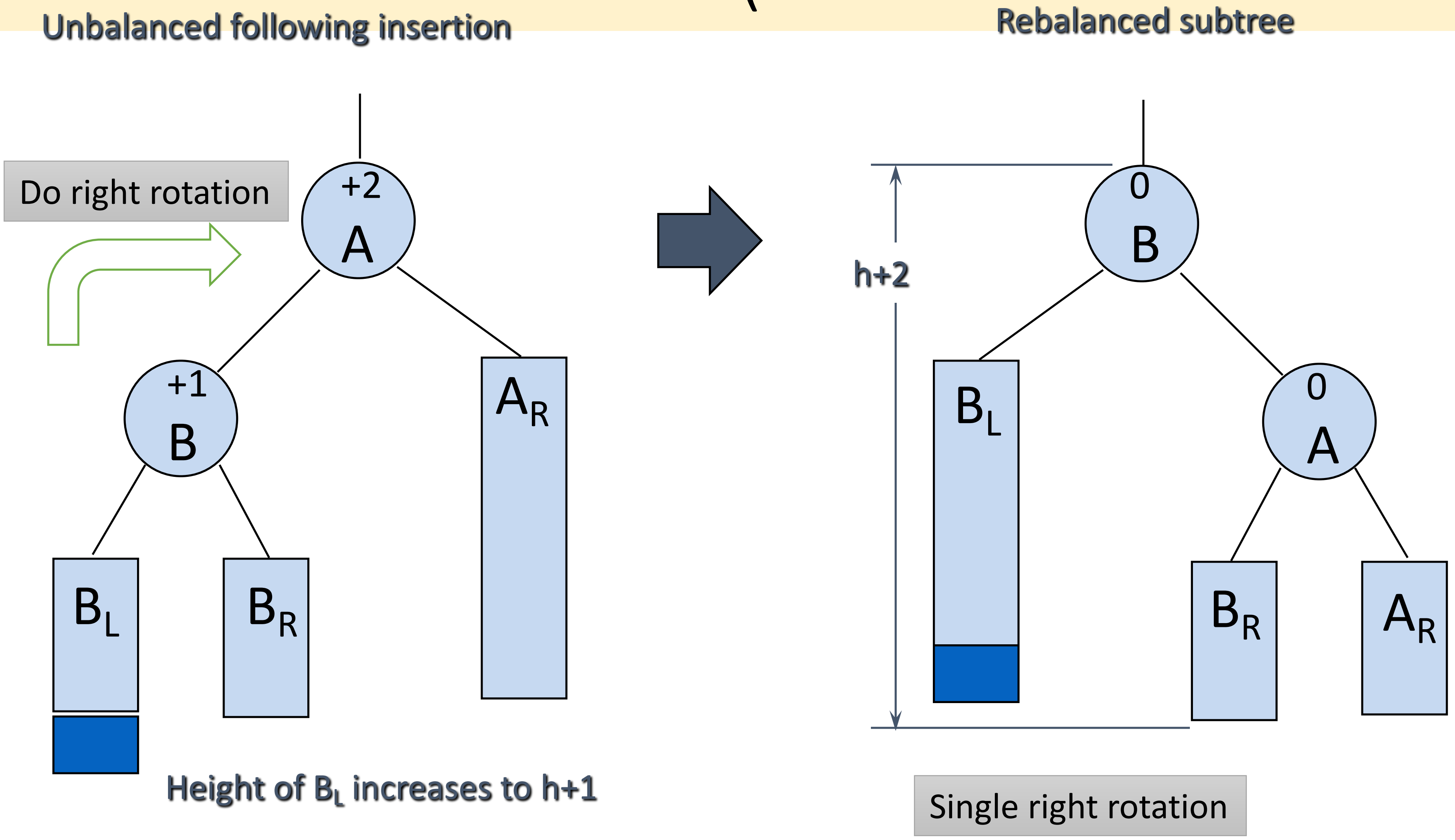
# AVL Trees

Unbalanced following insertion

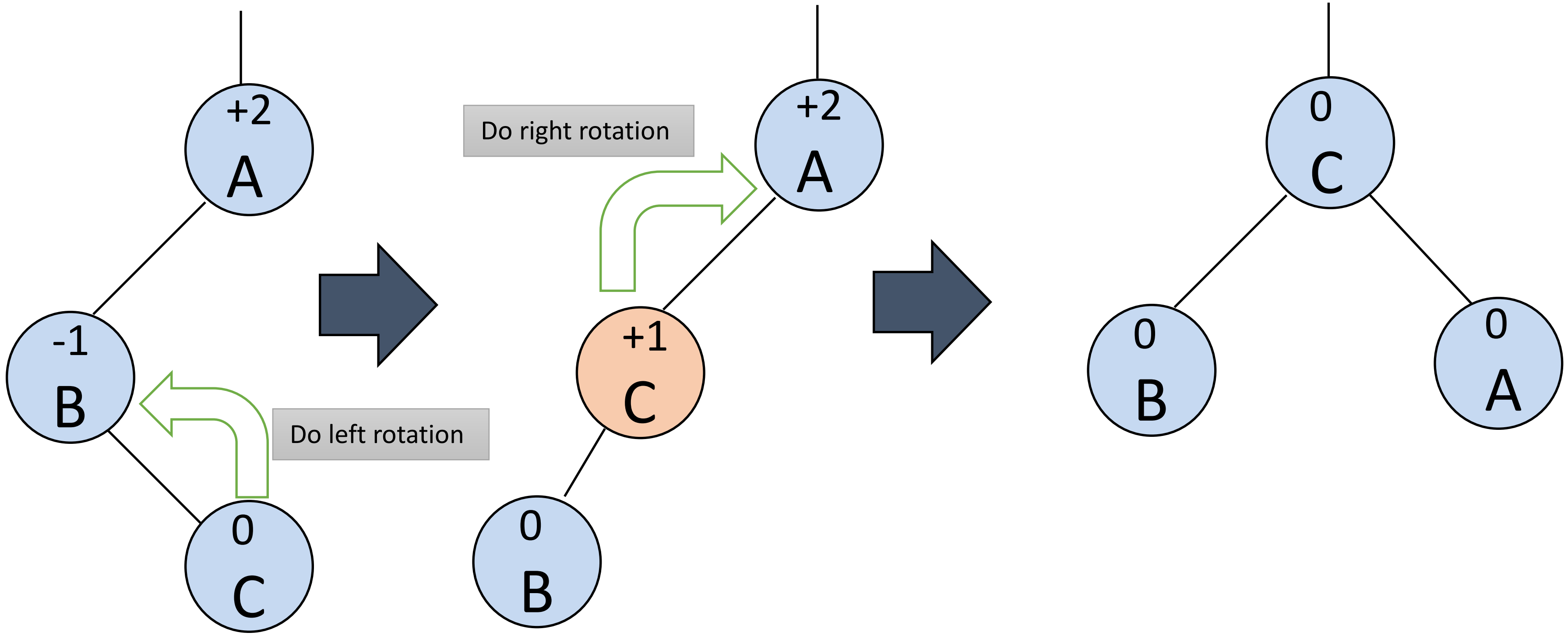



+2 Why can't we do a right rotation?

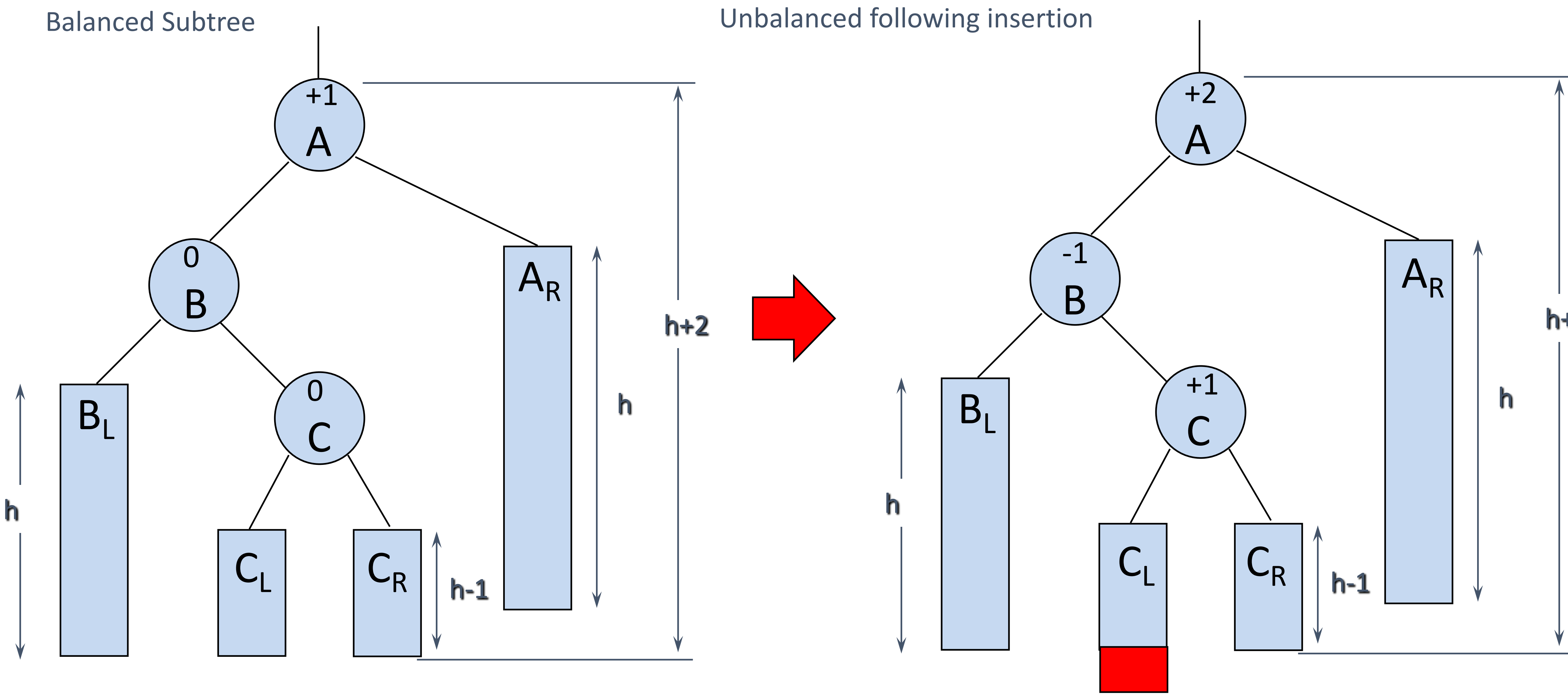
# Recall:
# AVL Trees - LL rotation(Outside case- Case 1)



# AVL Trees - LR rotation (a)- Inside Case- Case 3

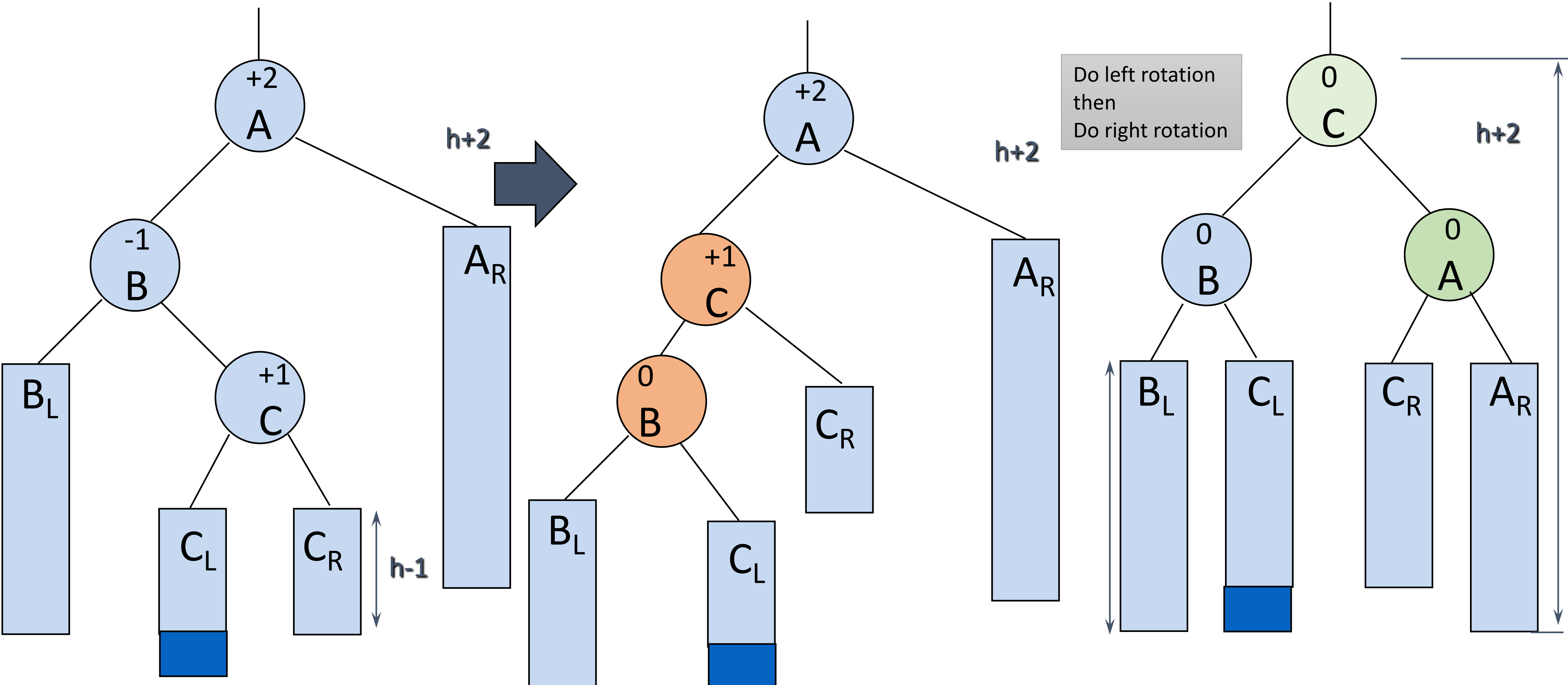


# AVL Trees

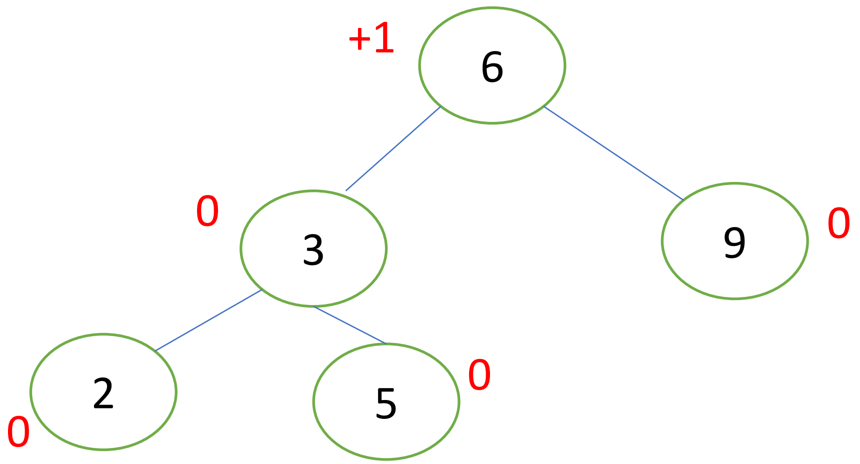


Balanced Subtree
+1
A
0
B
0
C
$B_L$
$C_L$
$C_R$
$A_R$
h
h-1
h
h+2
Unbalanced following insertion
+2
A
-1
B
+1
C
$B_L$
$C_L$
$C_R$
$A_R$
h
h-1
h

# AVL Trees - LR rotation (b)-- Inside Case- Case 3

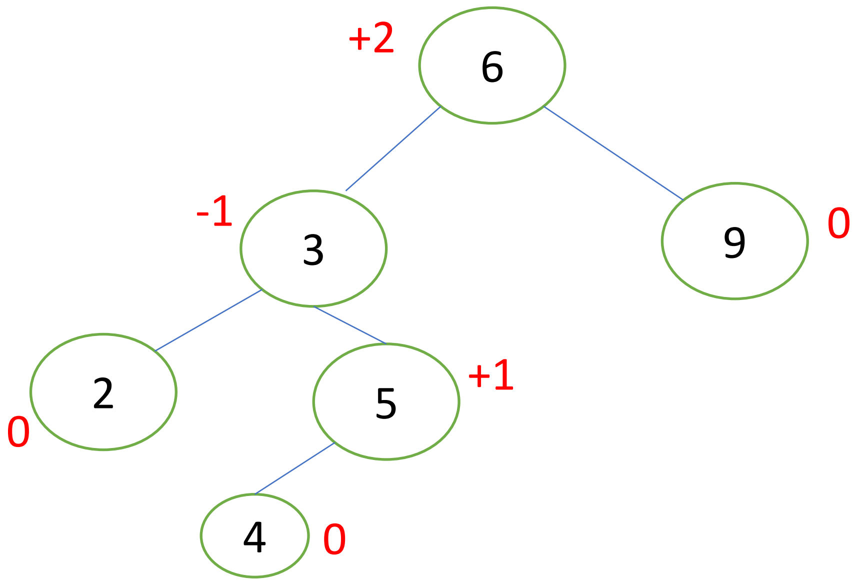


# Example-Case 3

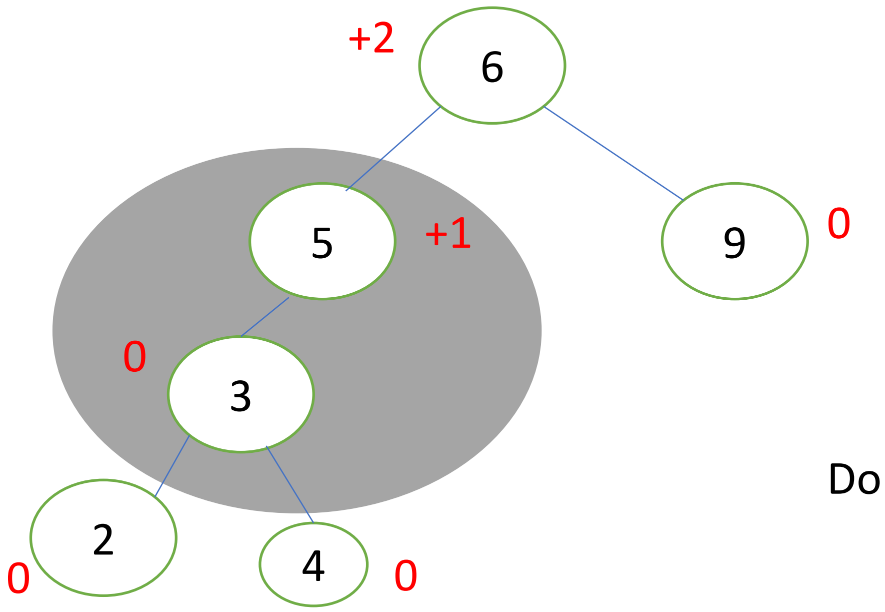


# Example-Case 3

Insert 4

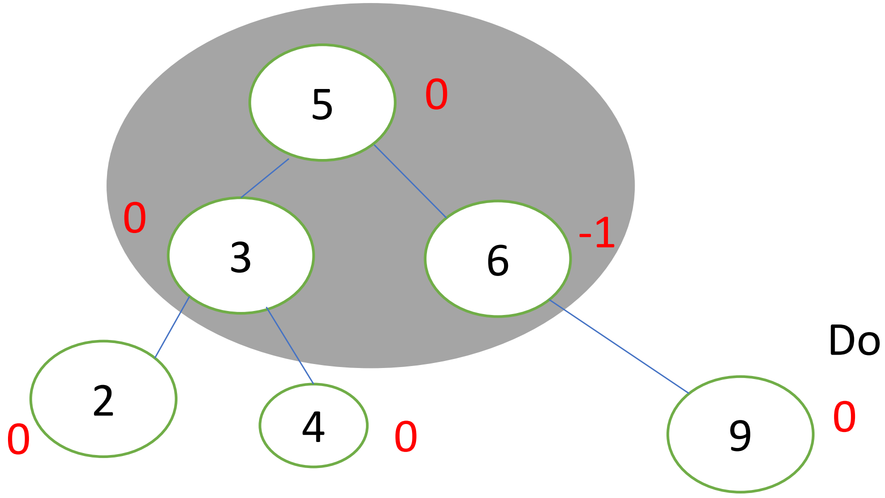


# Example-Case 3




Do left rotation

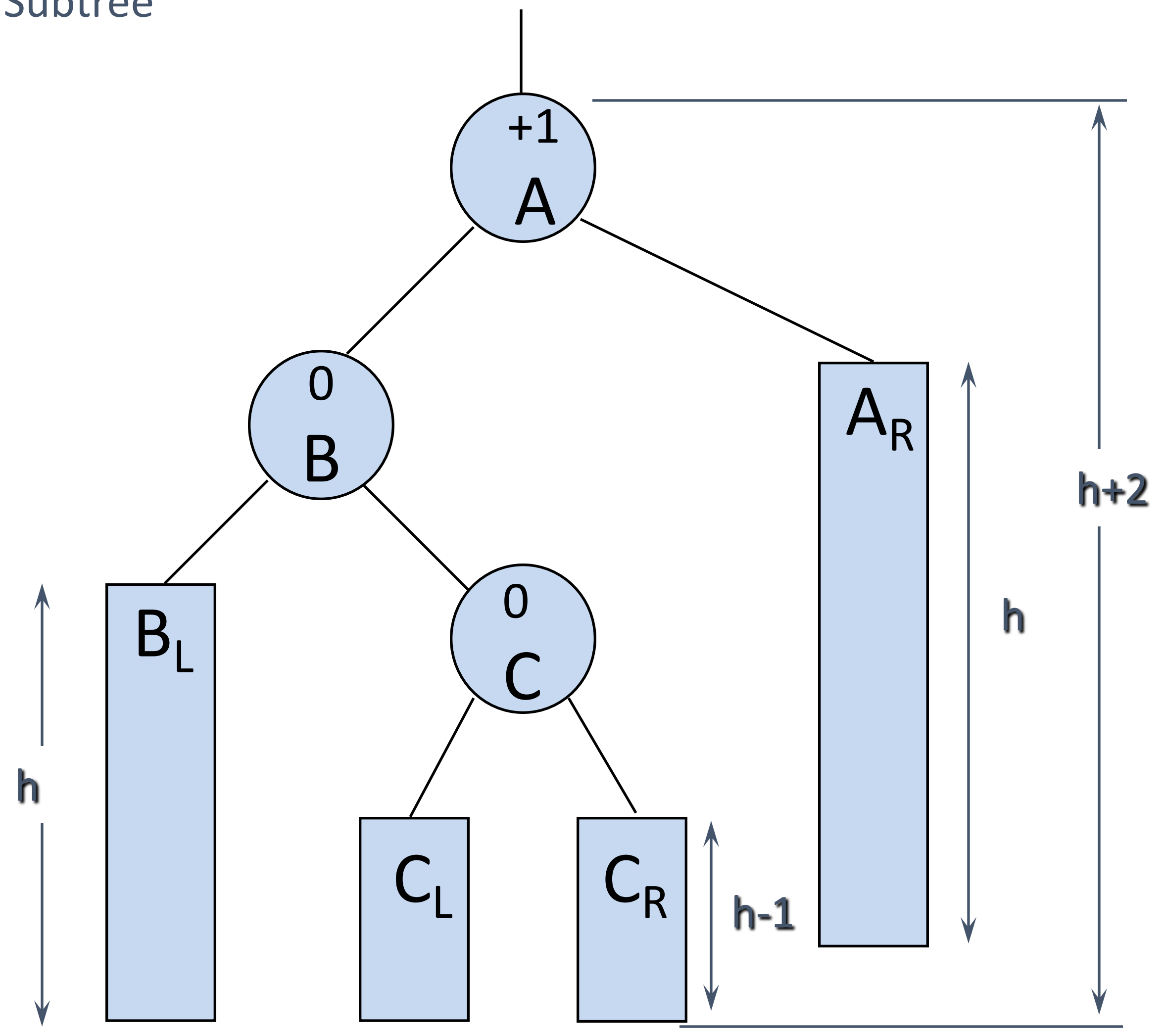
# Example-Case 3




Do right rotation

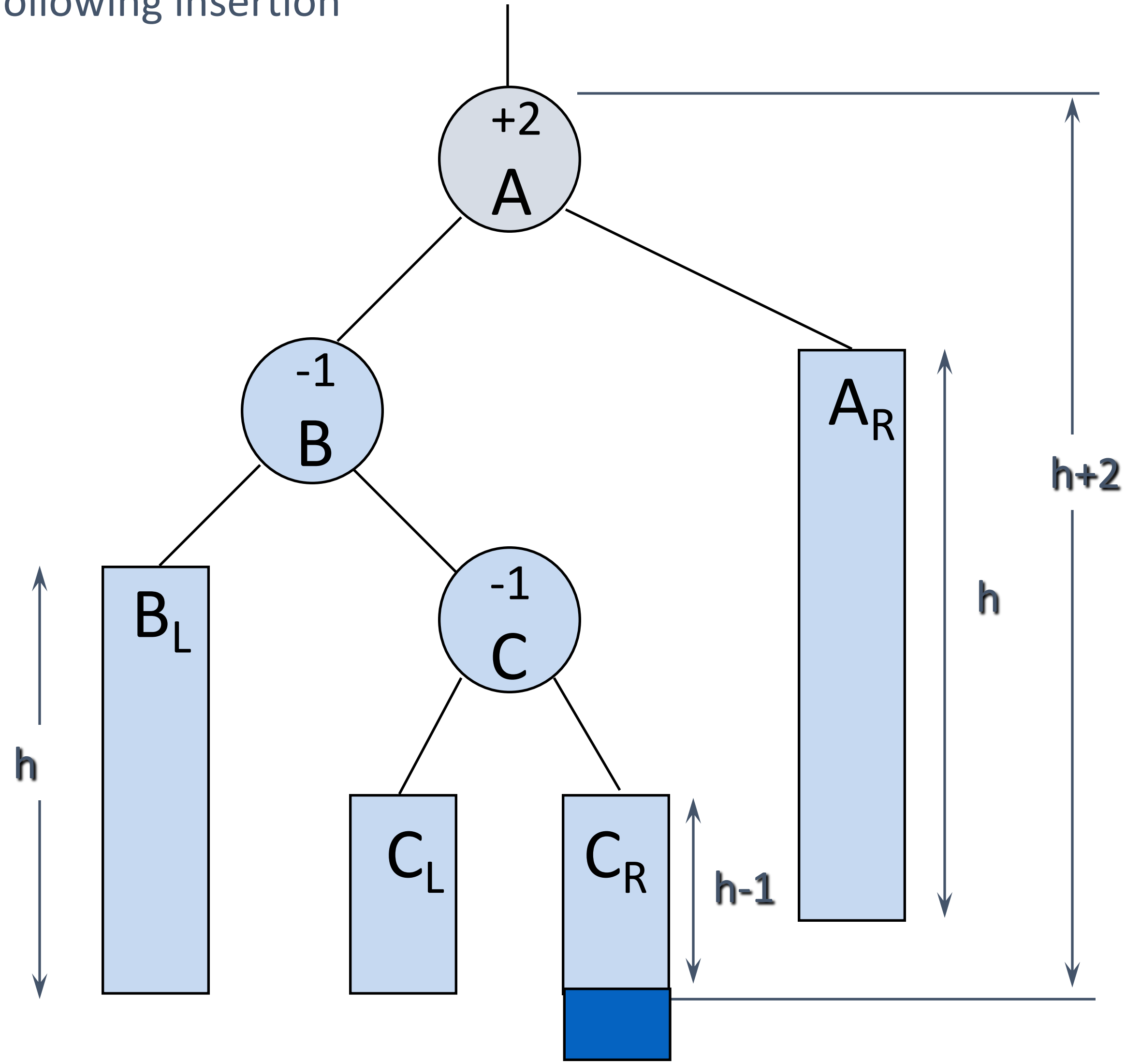
# AVL Trees

Balanced Subtree

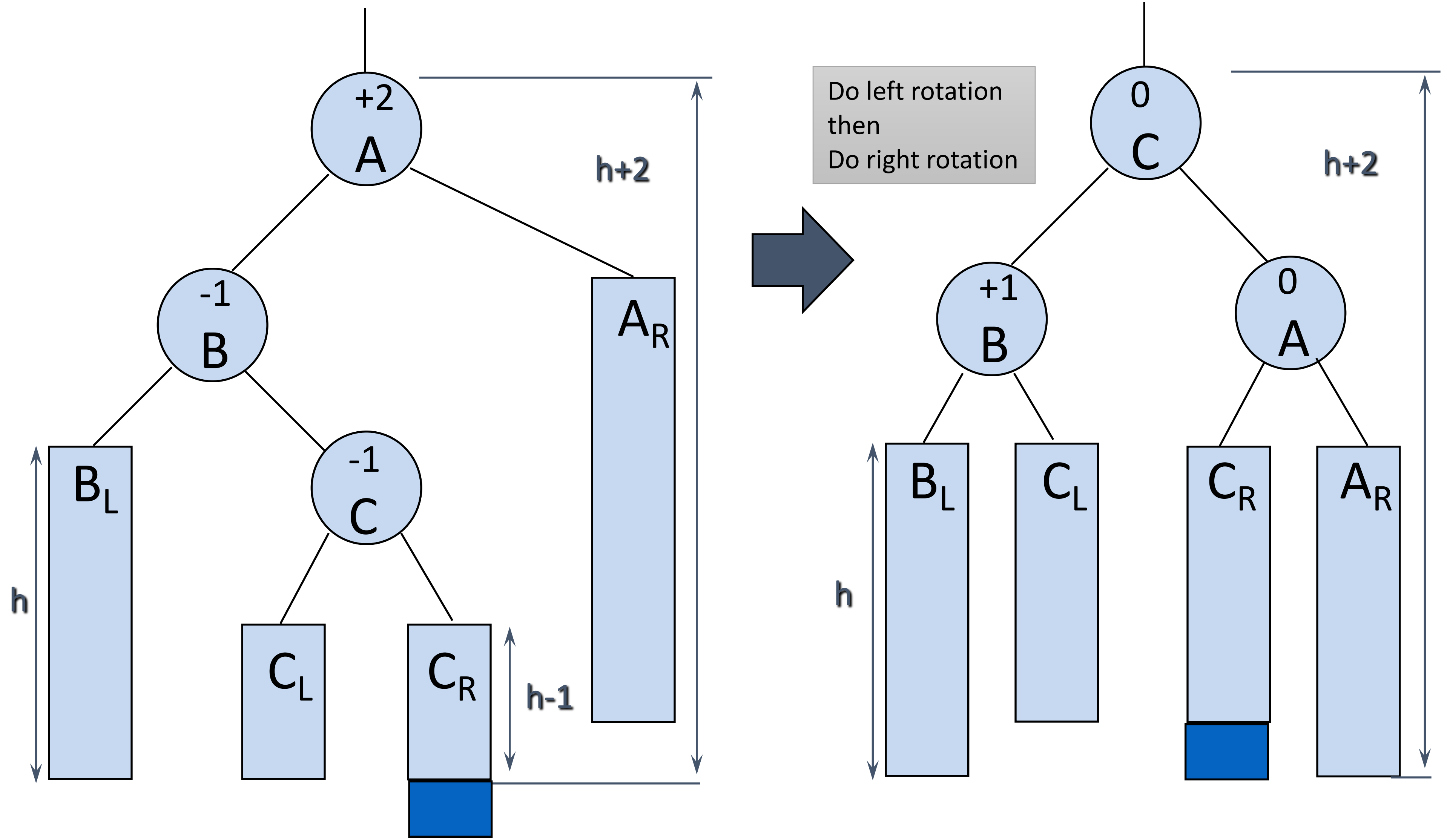


# AVL Trees

Unbalanced following insertion



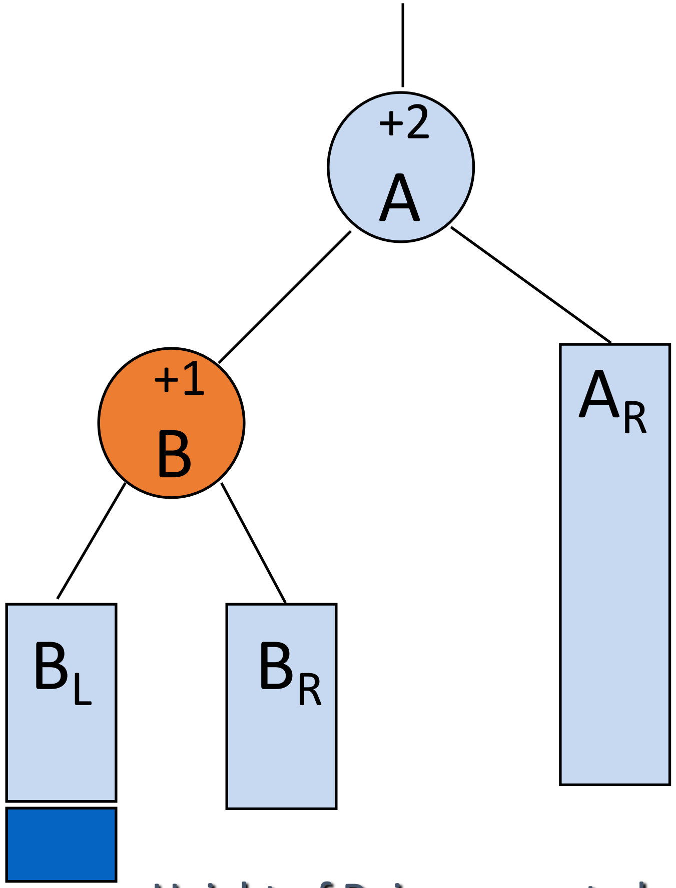
# AVL Trees - LR rotation (c) - Inside Case- Case 3



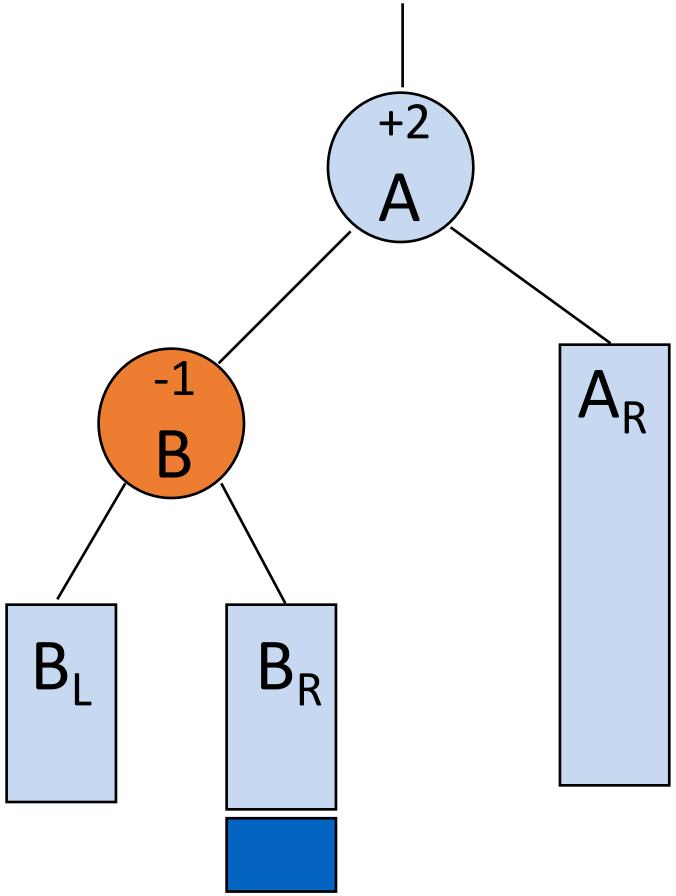
# Insertion into AVL Tree: Algorithm

- **Step 1**:Insert node as per BST.
- **Step 2**: Update the balance factor of each node.
- **Step 3**: If balance condition is violated, then
    - Perform rotations as per case.
    1. If BF(node) = +2 and BF(node -> left-child) = +1, perform LL rotation.[Case 1]
    2. If BF(node) = -2 and BF(node -> right-child) = -1, perform RR rotation. [Case 2]
    3. If BF(node) = -2 and BF(node -> right-child) = +1, perform RL rotation. [Case 4]
    4. If BF(node) = +2 and BF(node -> left-child) = -1, perform LR rotation. [Case 3]

# BF = +2: Is it Case 1 or Case 3?

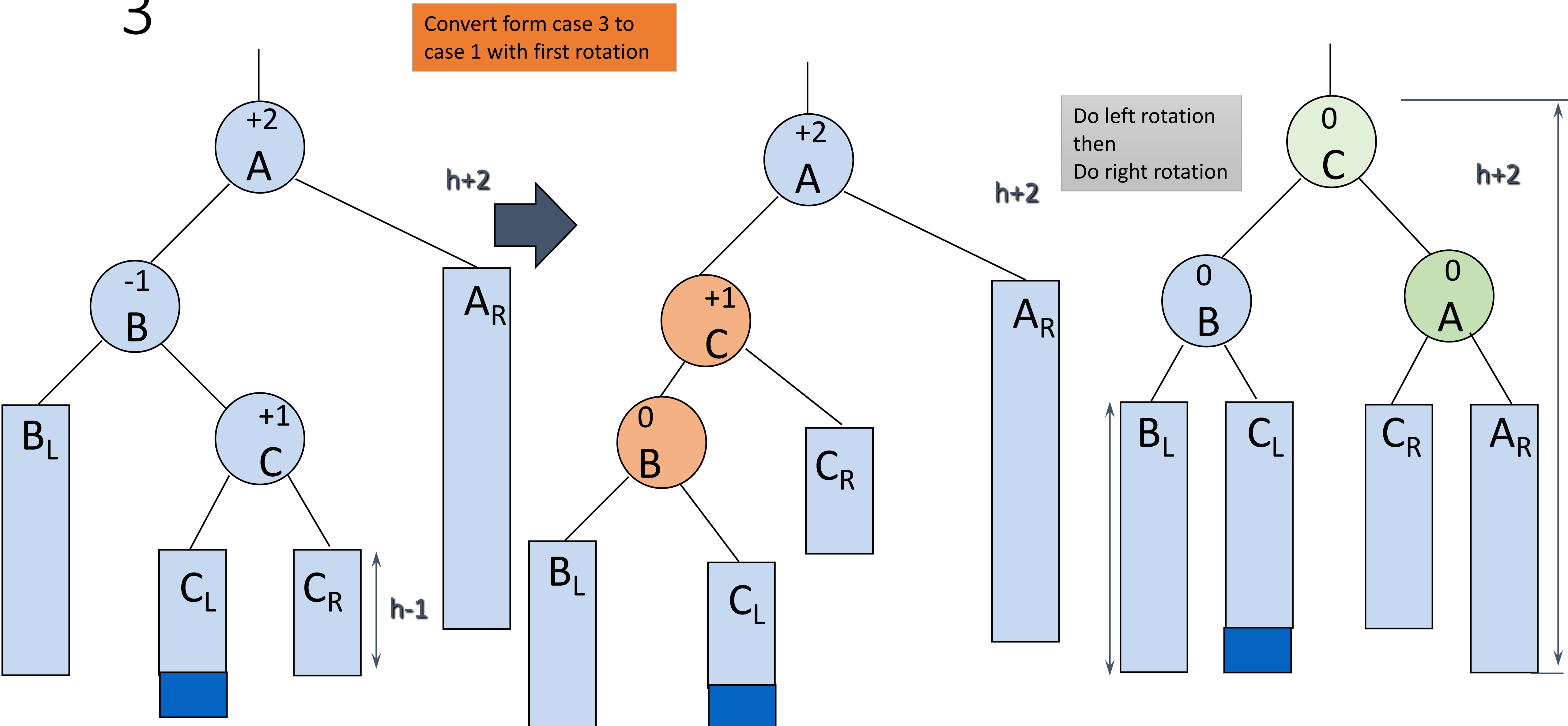



Height of $B_L$ increases to h+1



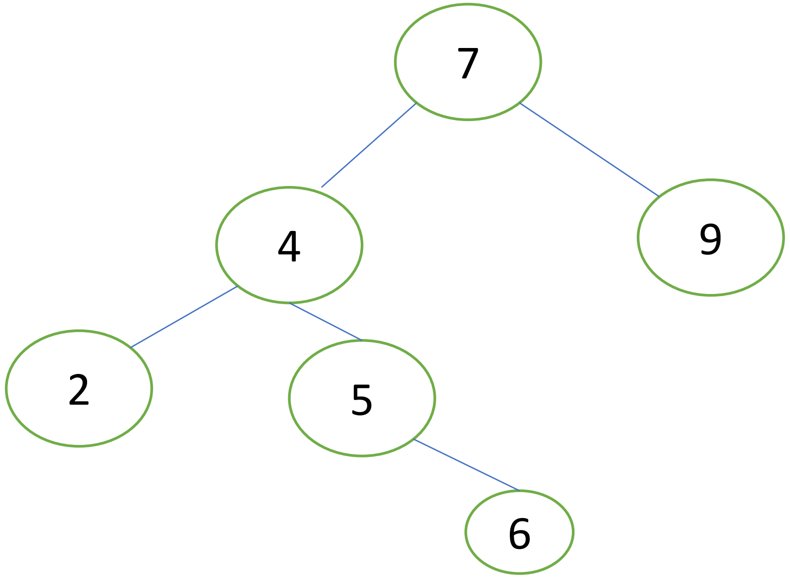

Height of $B_L$ increases to h+1

# AVL Trees - LR rotation (b)-- Inside Case- Case 3



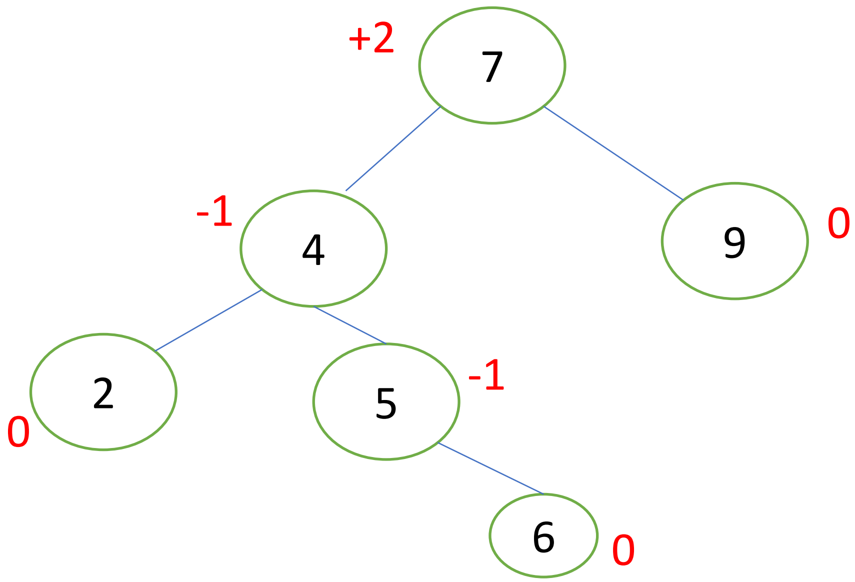
# Another Example-Case 1 or 3??

Insert 6



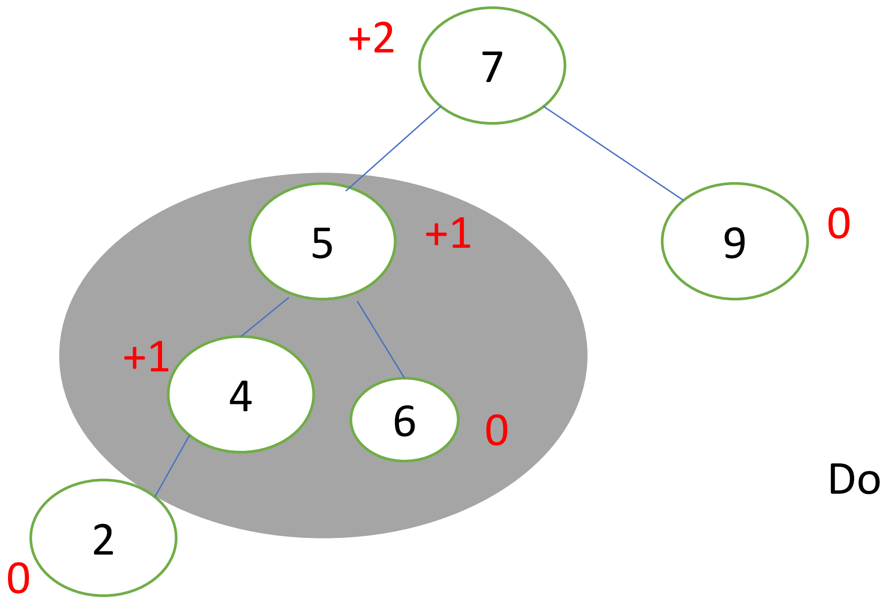
# Another Example-Case 1 or 3??

Insert 6



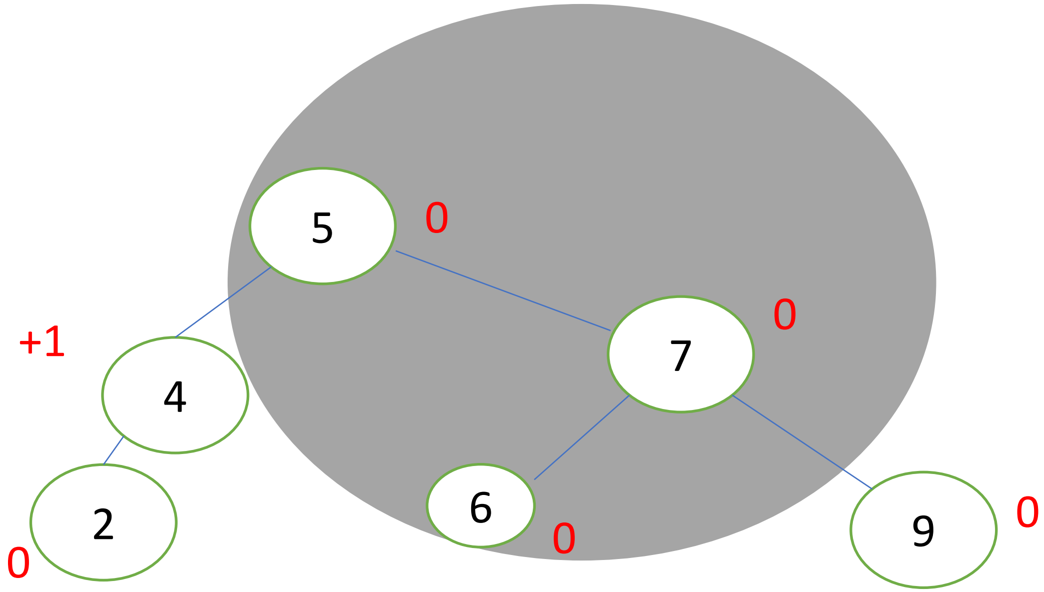

1. If BF(node) = +2 and BF(node -> left-child) = +1, perform LL rotation.[Case 1]
2. If BF(node) = -2 and BF(node -> right-child) = -1, perform RR rotation. [Case 2]
3. If BF(node) = -2 and BF(node -> right-child) = +1, perform RL rotation. [Case 4]
4. **If BF(node) = +2 and BF(node -> left-child) = -1, perform LR rotation. [Case 3]**

# Another Example-Case 3




Do left rotation

# Another Example-Case 3



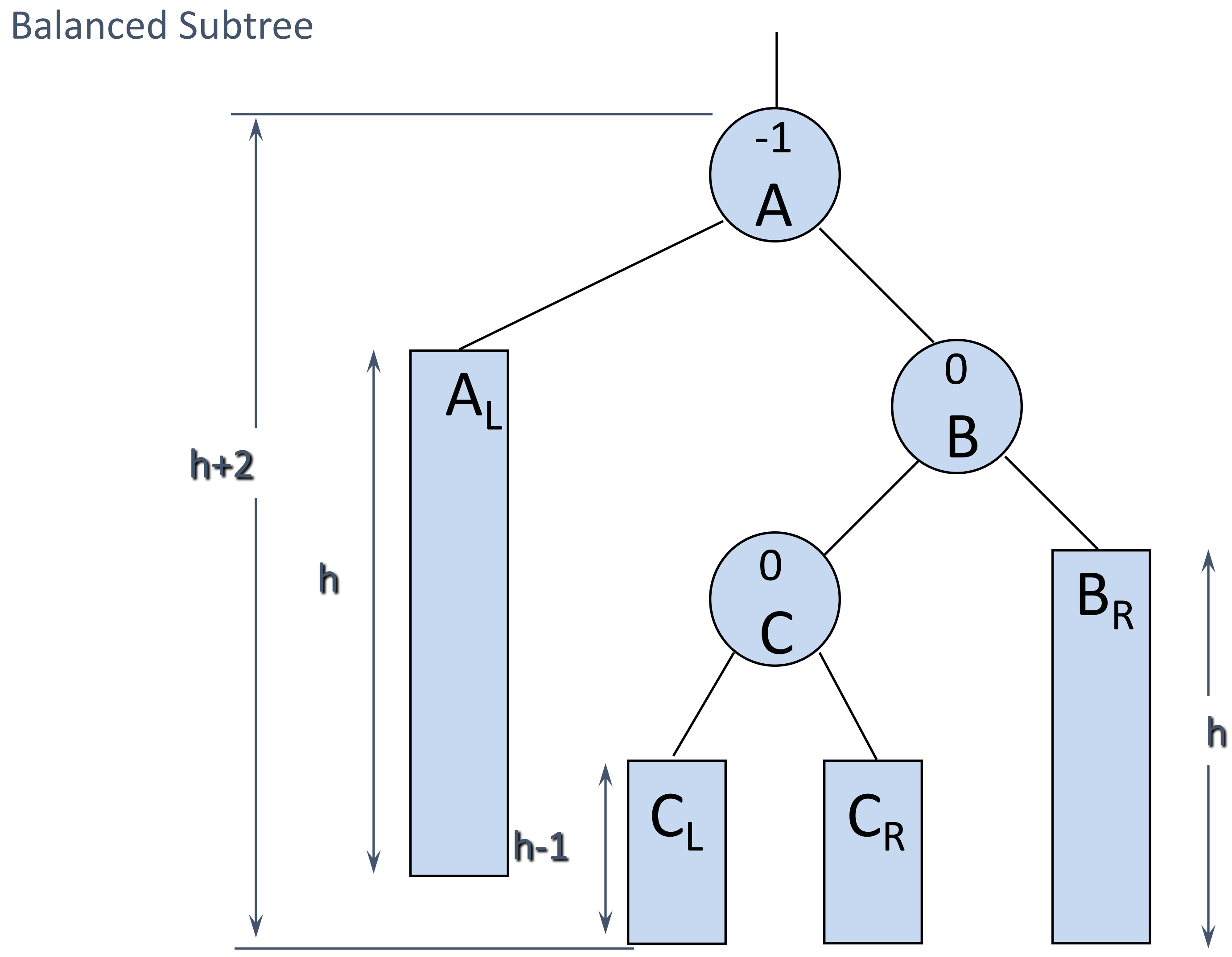

Do right rotation

# Insertion into AVL Tree: Algorithm

- **Step 1**:Insert node as per BST.
- **Step 2**: Update the balance factor of each node.
- **Step 3**: If balance condition is violated, then
  - Perform rotations as per case.

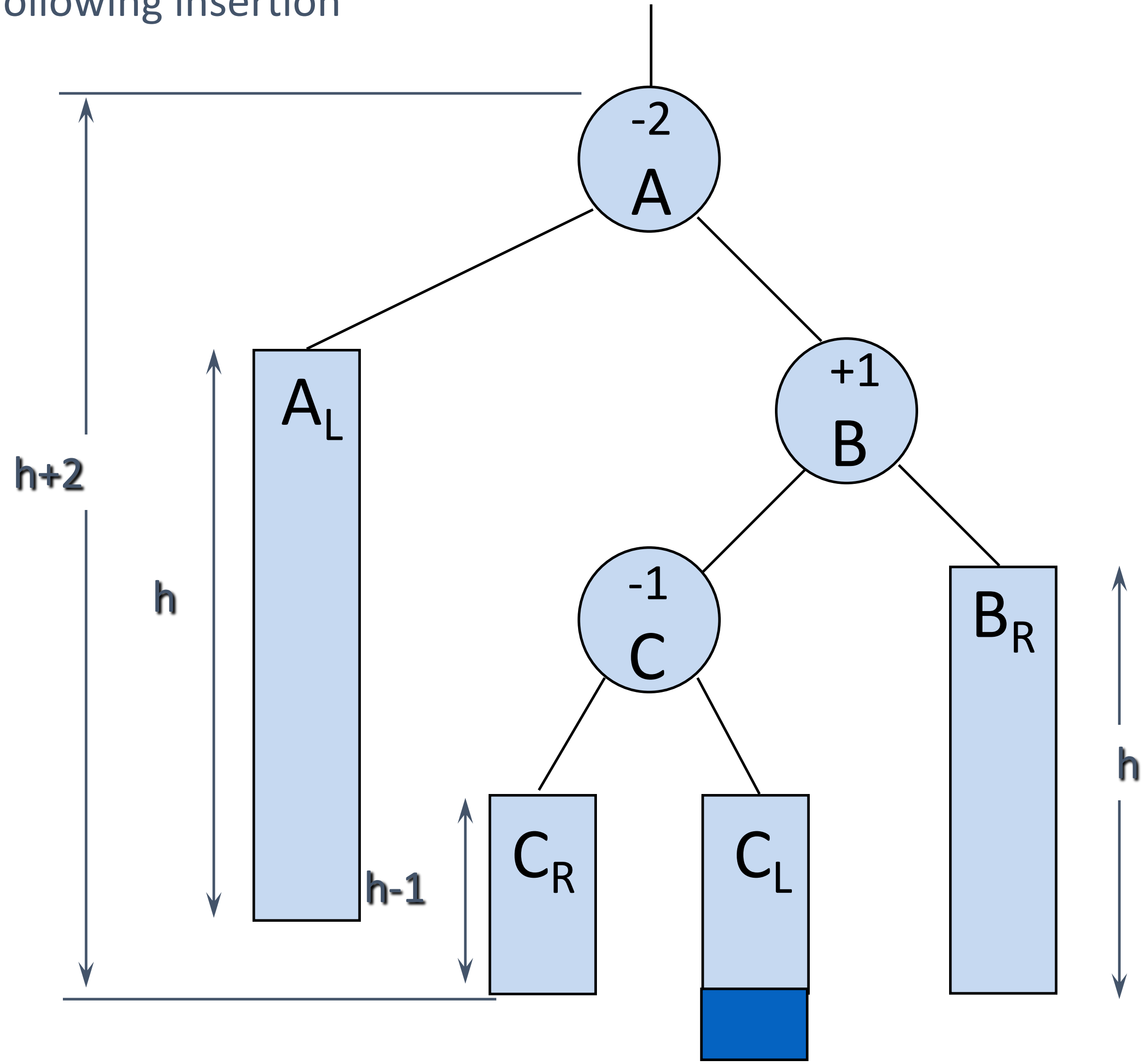
  1. If BF(node) = +2 and BF(node -> left-child) = +1, perform LL rotation.[Case 1]
  2. If BF(node) = -2 and BF(node -> right-child) = -1, perform RR rotation. [Case 2]
  3. **If BF(node) = -2 and BF(node -> right-child) = +1, perform RL rotation. [Case 4]**
  4. If BF(node) = +2 and BF(node -> left-child) = -1, perform LR rotation. [Case 3]

# AVL Trees

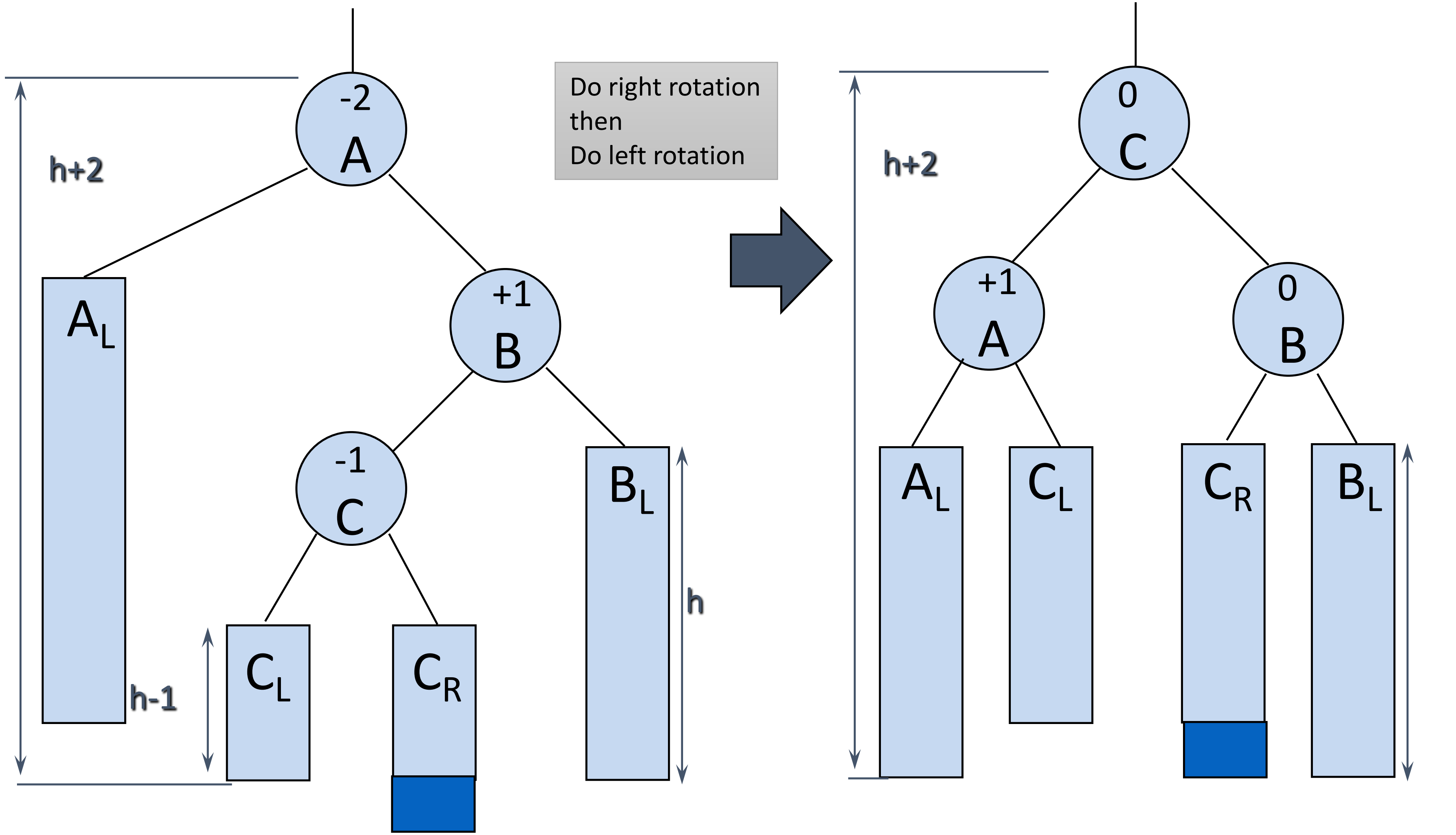


# AVL Trees

Unbalanced following insertion



# AVL Trees - RL rotation- Inside Case- Case 4



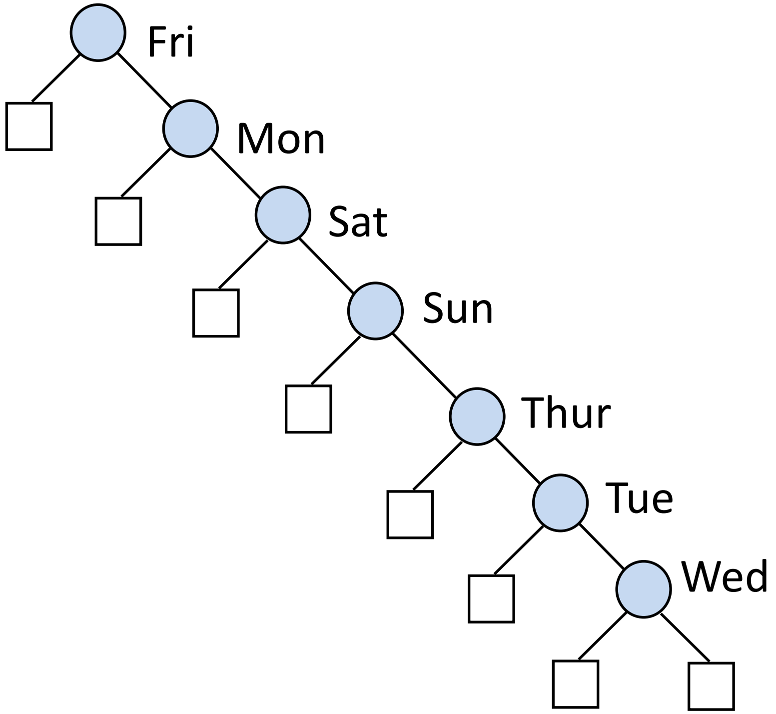
# Deletion in AVL tree: algorithm

- **Step 1:** Find the element in the tree.
- **Step 2:** Delete the node, as per the BST Deletion.
- **Step 3:** Two cases are possible:-
  - **Case 1:** Deleting from the right subtree.
    - 1A. If BF(node) = +2 and BF(node -> left-child) = +1, perform LL rotation.
    - 1B. If BF(node) = +2 and BF(node -> left-child) = -1, perform LR rotation.
    - 1C. If BF(node) = +2 and BF(node -> left-child) = 0, perform LL rotation.
  - **Case 2**: Deleting from left subtree.
    - 2A. If BF(node) = -2 and BF(node -> right-child) = -1, perform RR rotation.
    - 2B. If BF(node) = -2 and BF(node -> right-child) = +1, perform RL rotation.
    - 2C. If BF(node) = -2 and BF(node -> right-child) = 0, perform RR rotation.

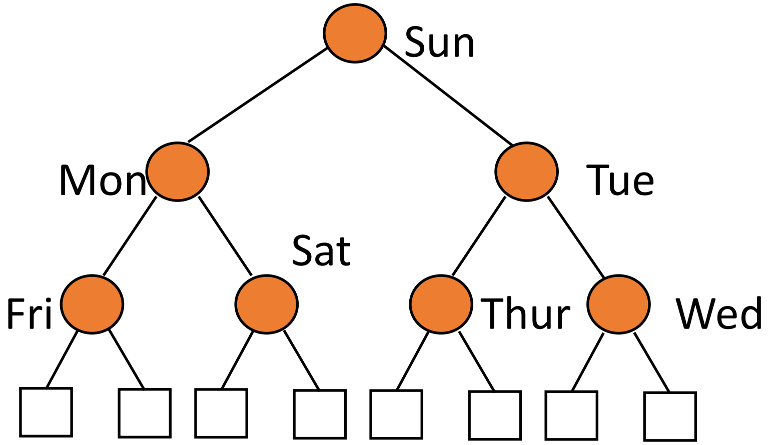
# Comments on complexity

- The re-balancing rotation only costs O(1).
- Insertion/deletion/searching in AVL trees:
  - all take O(log n) in the best, average and worst cases!
- Contrast with BST, where the best and average case is O(log n) but the worst case is O(n) (the worst case being when the BST is effectively a linked list!).

# BST Time Complexity: motivating Balance Trees AKA AVL (**Adelson-Velskii and Landis) Trees**




Insert: O(N)
Search: O(N)
Delete: O(N)




Insert: O(logN)
Search: O(LogN)
Delete: O(logN)

# Applications of AVL trees

- In general, AVL trees can be applied in cases characterized by the following conditions:
  - Fewer insertions and deletions. Why?
  - Faster search is needed.
  - Sorted or nearly sorted input data.

- For example, AVL are used in:
  - Sorting of in-memory collections e.g., sets and dictionaries.
  - In applications that require improved searching, including database applications where there are fewer insertions and deletions.
    - Indexes large records in a database to improve search.