04-630
Data Structures and Algorithms for Engineers

Lecture 12: Height-Balanced Trees: AVL Trees

Height-balanced Trees

- The goal of height-balancing is to ensure that the tree is as complete as possible and that, consequently, it has minimal height for the number of nodes in the tree

- As a result, the number of probes it takes to search the tree (and the time it takes) is minimized.

- A tree can be balanced with respect to the heights of its subtrees.

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

Adelson-Velskii and Landis (AVL) tree

Recap: Complexity Analysis of AVL trees

- 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!).

AVL Trees(1/2)

- BSTs with a ***balance condition***.

- **AVL tree:** identical to a BST, except that for every node in the tree, the height of the left and right subtrees can differ by at most 1, i.e.:
  - $|\text{height}(T_L)\text{-height}(T_R)| \leq 1$.
  - Height information is kept for each node in the AVL tree.

AVL Trees(2/2)

- The requirement that the heights of the subtrees differs by at most 1 is called the **balance factor**.

- The balance factor must be maintained even after insertions.
  - Achieved through ***rotation***.

AVL Trees: Definition

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) $|height(T_L)-height(T_R)| \leq 1$.

3. Therefore, every sub-tree in a height-balanced tree is also height-balanced.

Recall: Binary Tree basics

- The height of T is defined recursively as

  0 if T is empty and

  $1 + max(height(T_L), height(T_R))$ otherwise,
  where T_L and T_R are the subtrees of the root

- The height of a tree is the length of a longest chain of descendants

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 Trees: Balance Factor

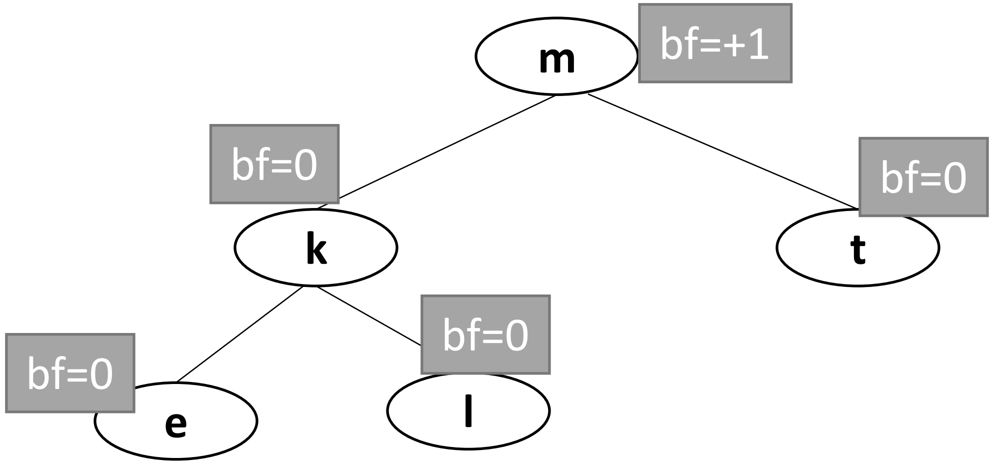
- Balance Factor $BF(T)$ of a node T in a binary tree is defined to be

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

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

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

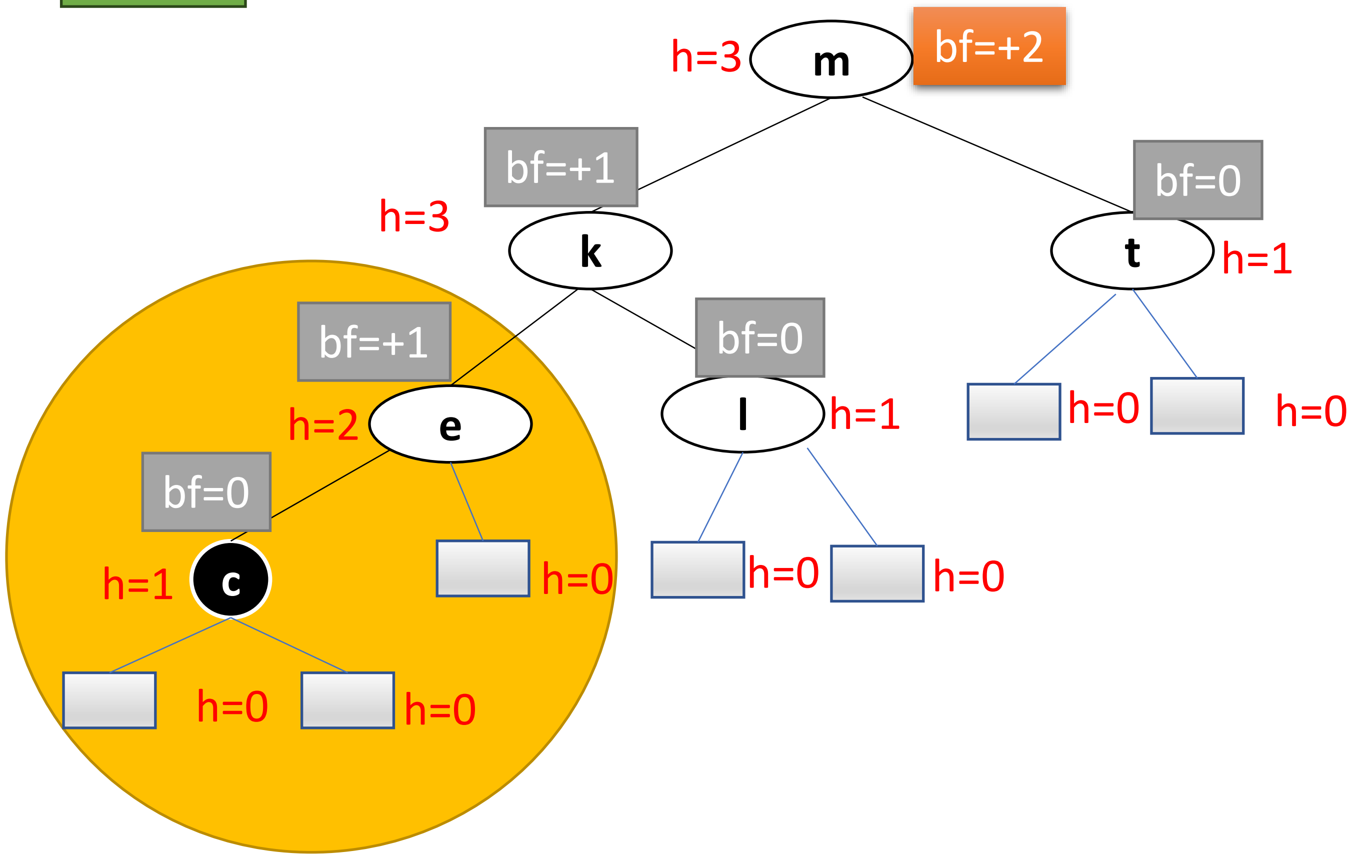
Example AVL tree



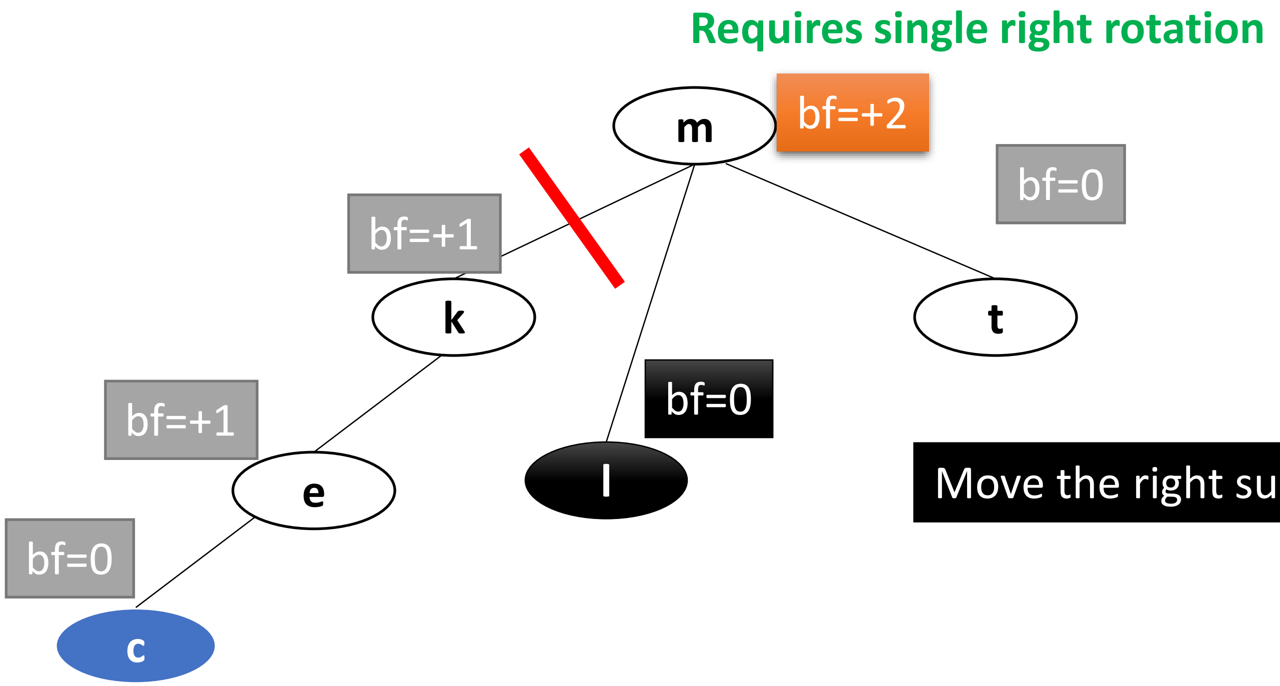
Example: Rebalancing after insertion

Insert c

Requires single right rotation



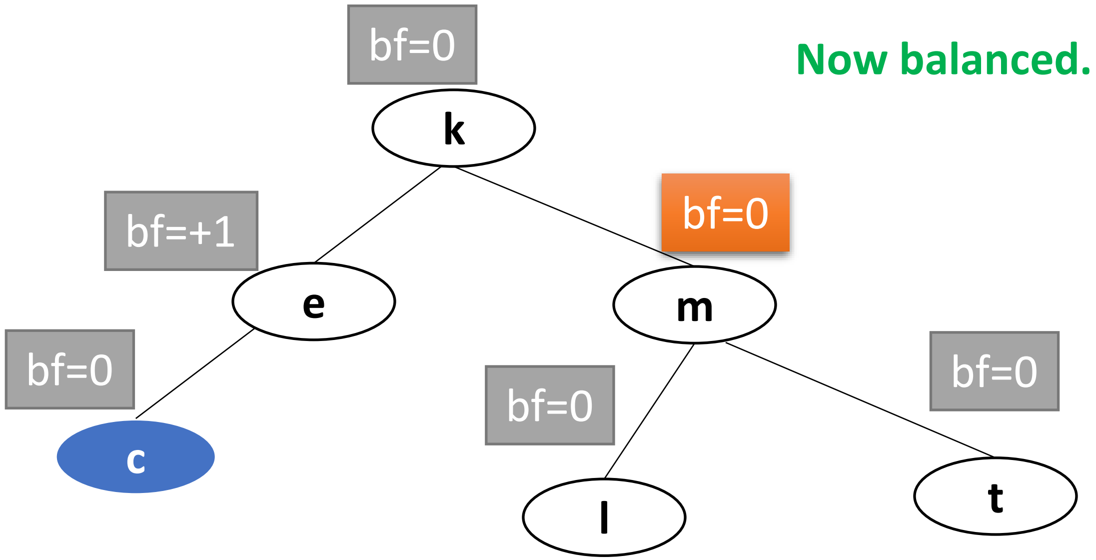
Rebalancing after insertion



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

Rebalancing after insertion

Make m the right child of k.



Rebalancing cases

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

- 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)

AVL Trees

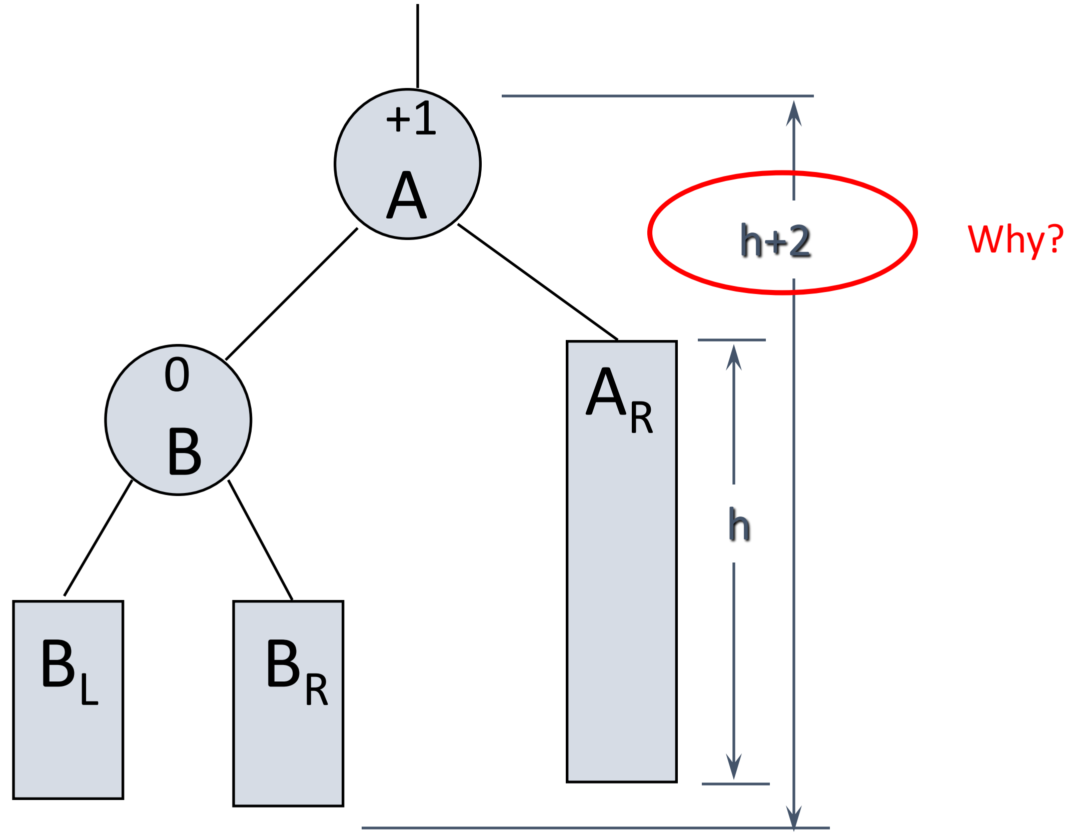
- LL: Y is inserted in the
  Left subtree of the Left subtree of A
  - LL: the path from A to Y
  - Left subtree then Left subtree
- LR: Y is inserted in the
  Right subtree of the Left subtree of A
  - LR: the path from A to Y
  - Left subtree then Right subtree

AVL Trees

- 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

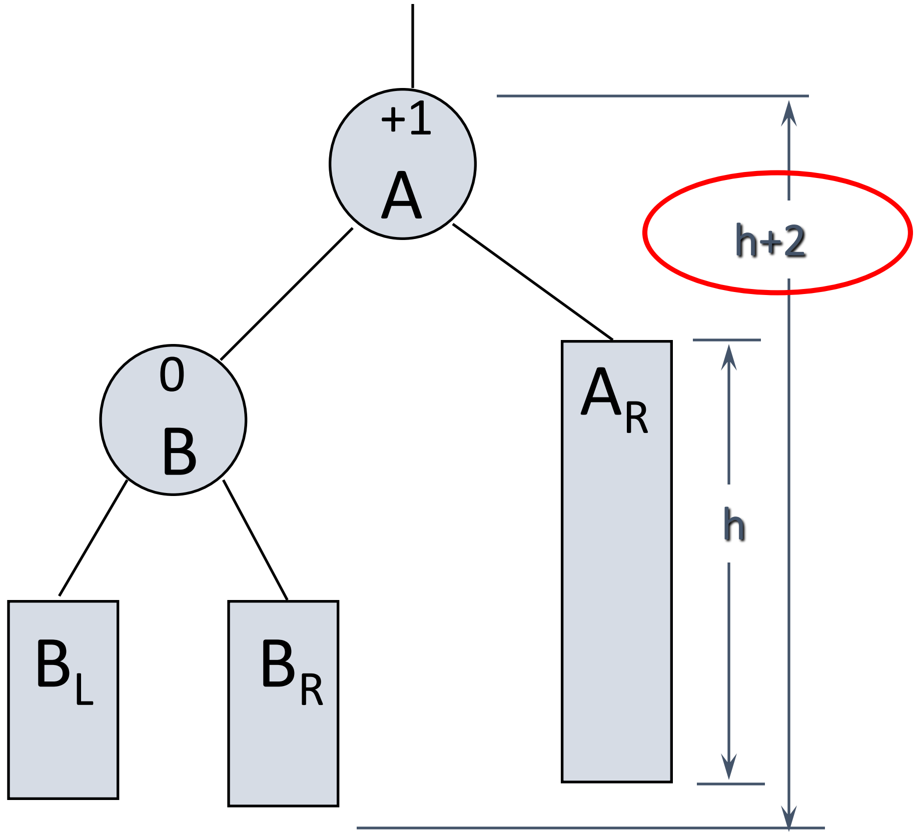
AVL Trees

Balanced Subtree



AVL Trees

Balanced Subtree



Why?

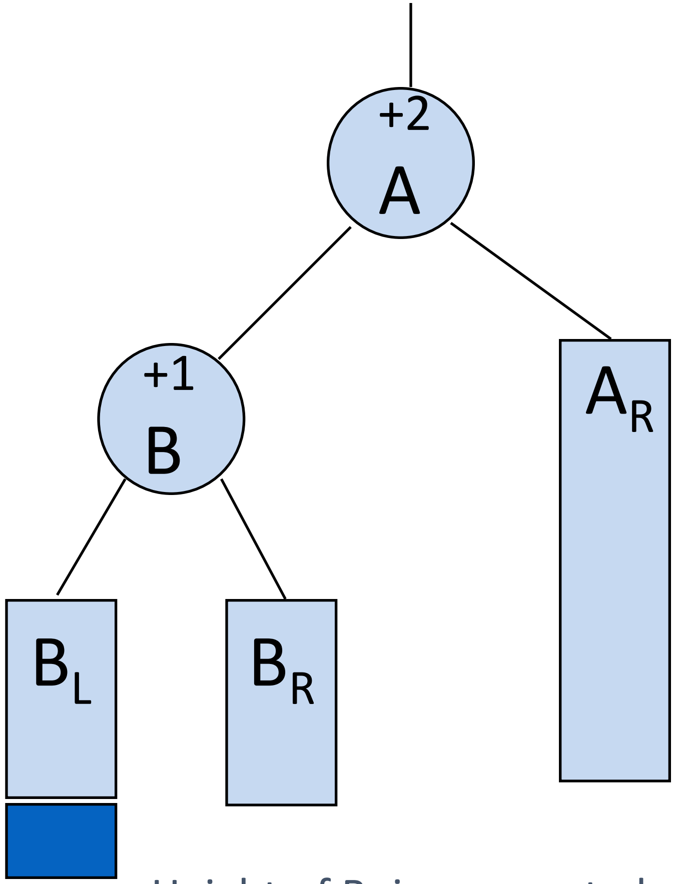
Hint:
balance factor
of A is +1

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

Can only mean height(T_1)=h+1

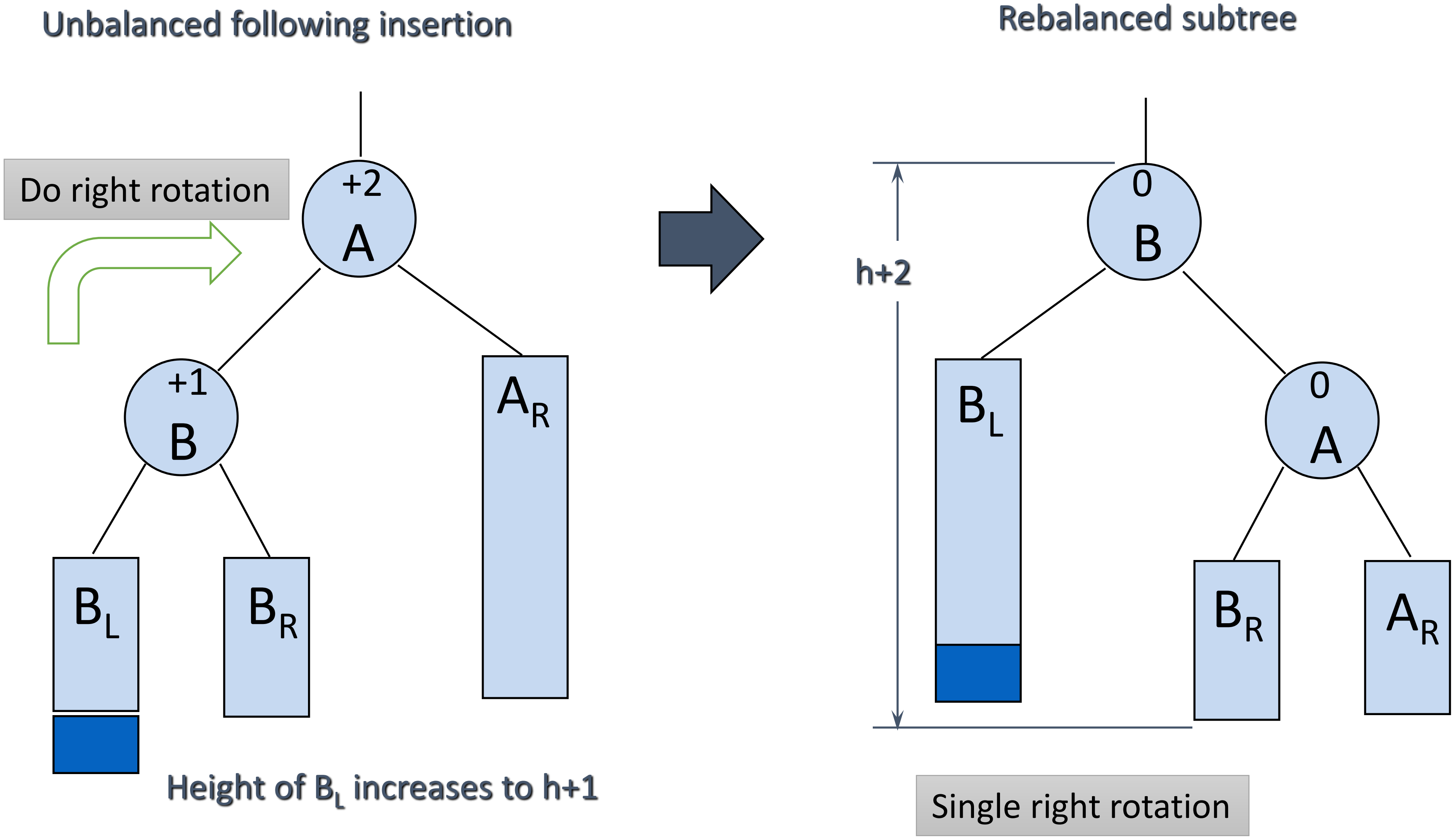
AVL Trees

Unbalanced following insertion



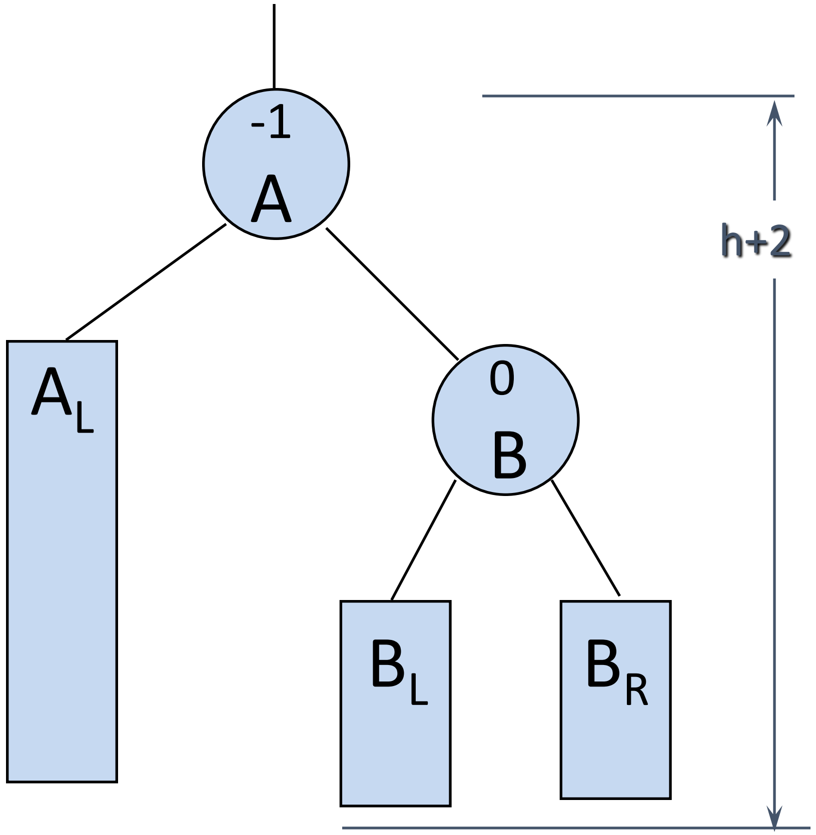
Height of B_L increases to h+1

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



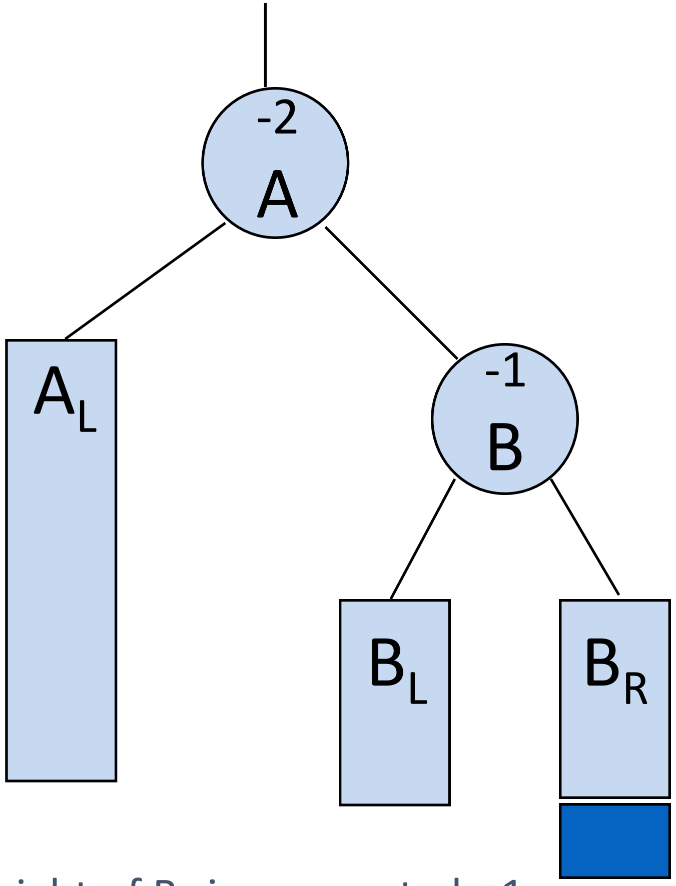
AVL Trees

Balanced Subtree



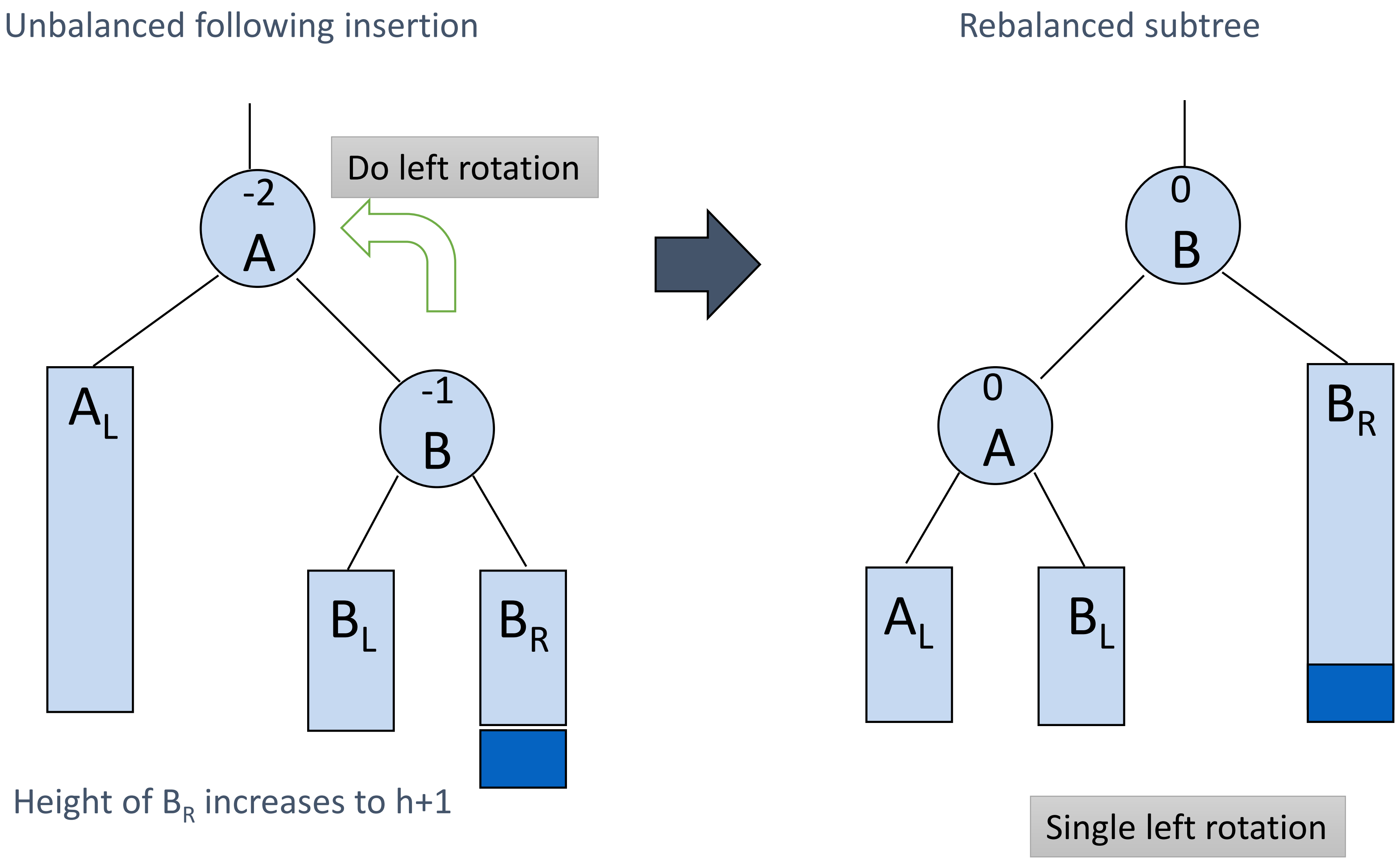
AVL Trees

Unbalanced following insertion



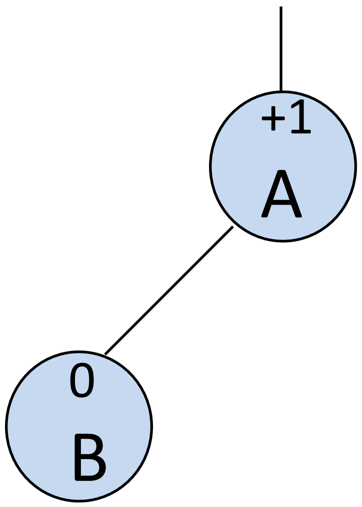
Height of B_R increases to h+1

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



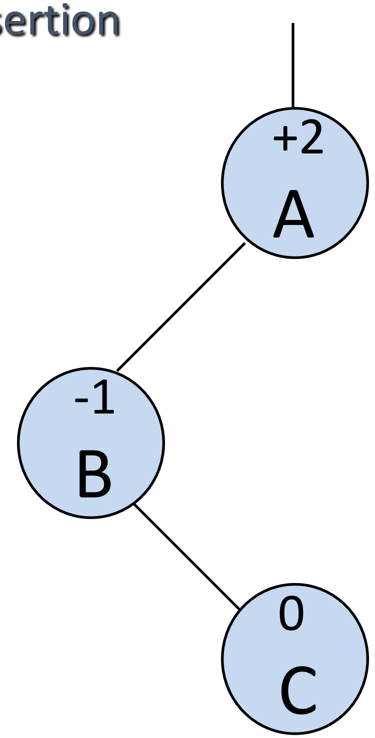
AVL Trees

Balanced Subtree

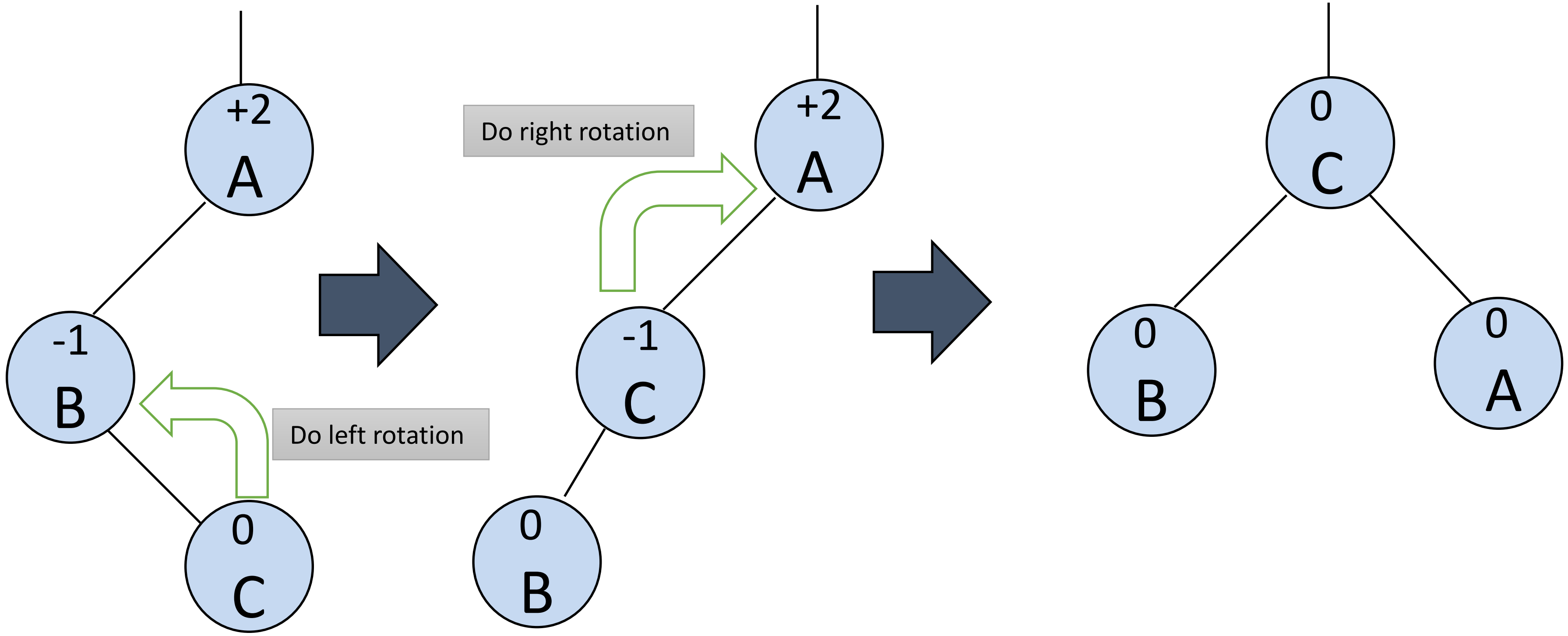


AVL Trees

Unbalanced following insertion

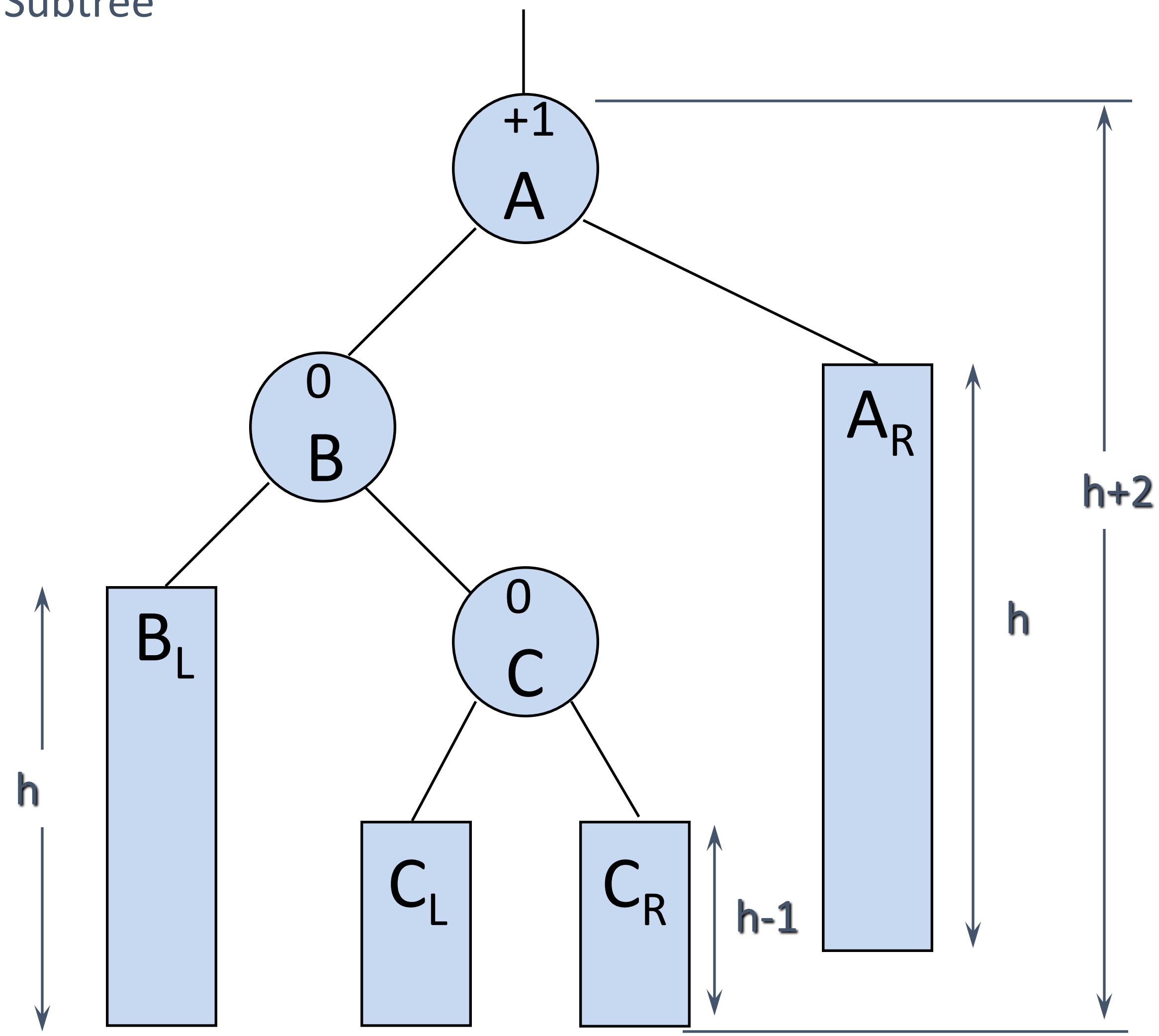


AVL Trees - LR rotation (a)- Inside Case- Case 3



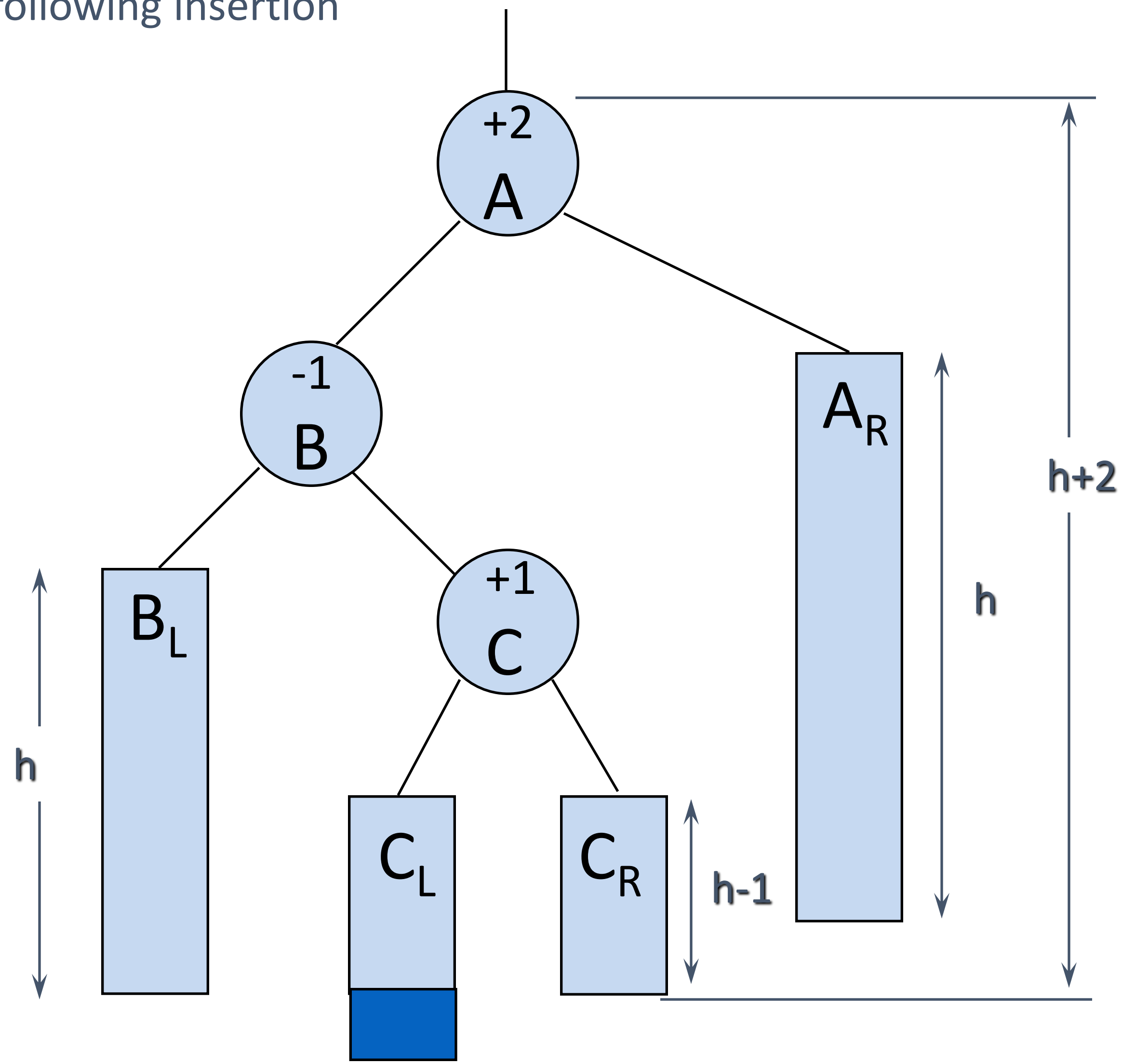
AVL Trees

Balanced Subtree

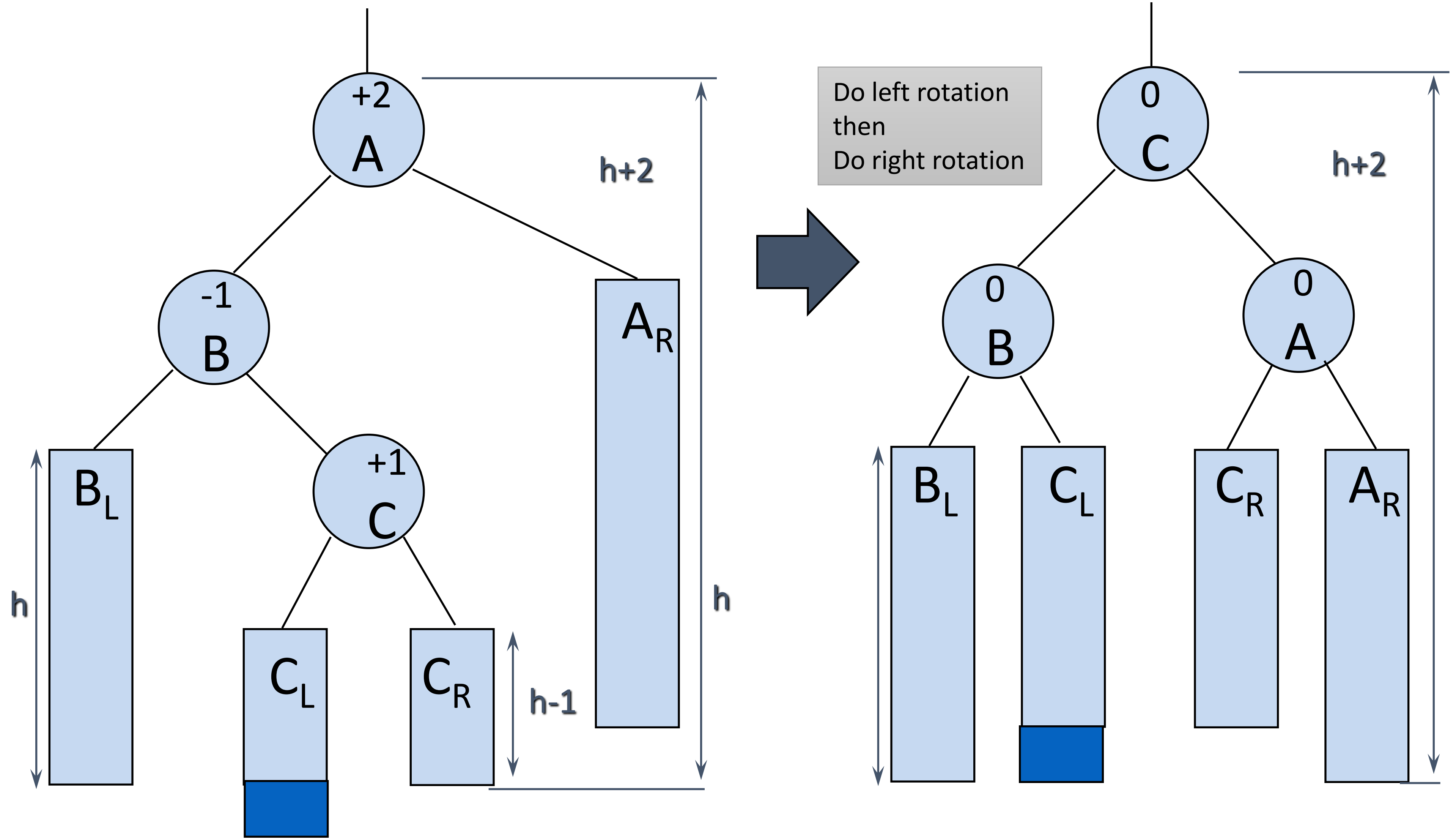


AVL Trees

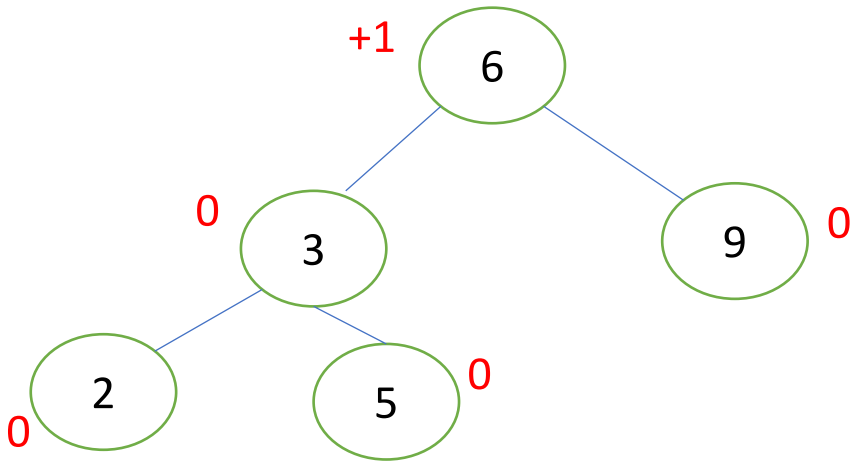
Unbalanced following insertion



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

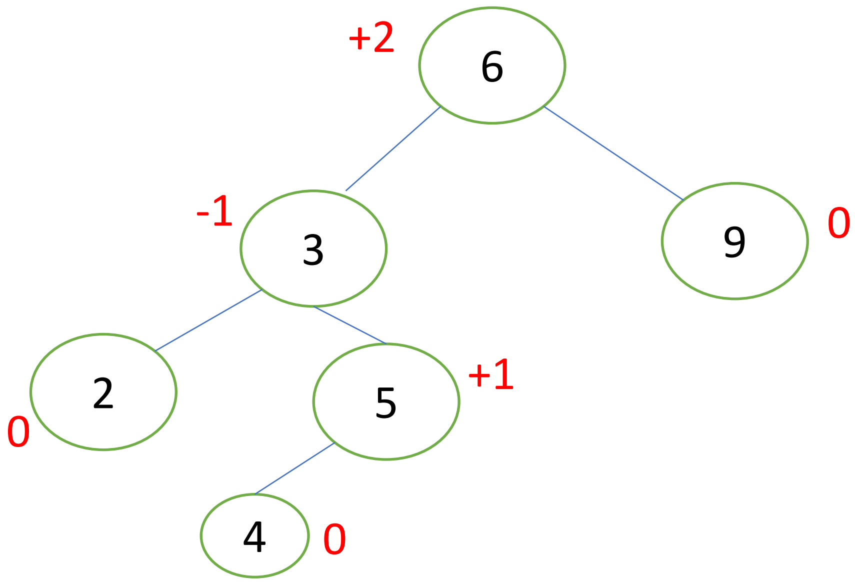


Example-Case 3

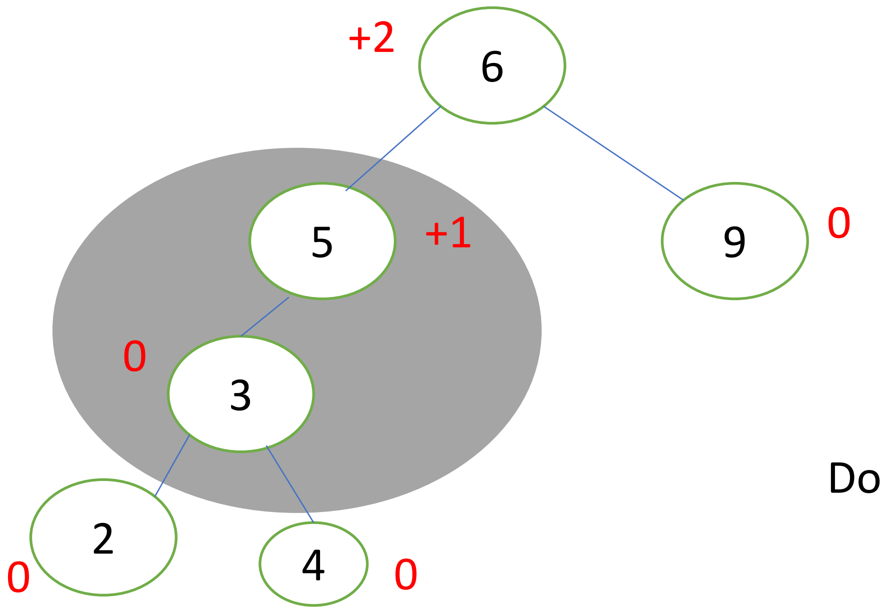


Example-Case 3

Insert 4

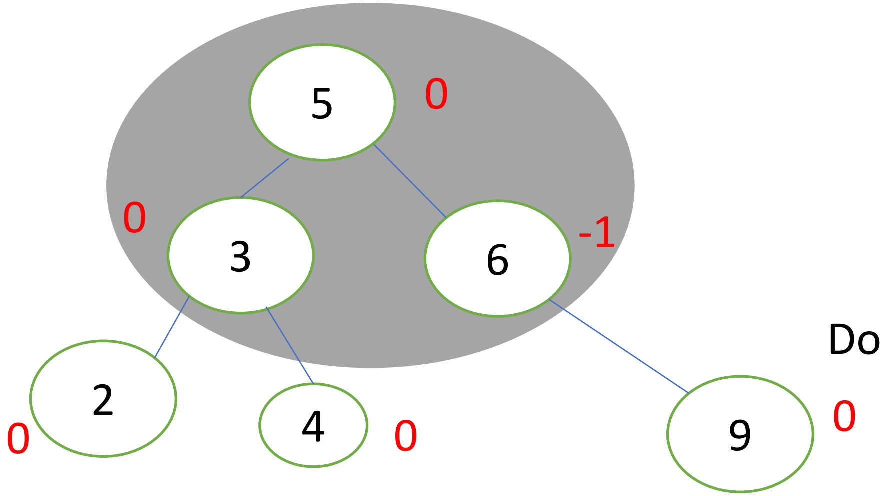


Example-Case 3



Do left rotation

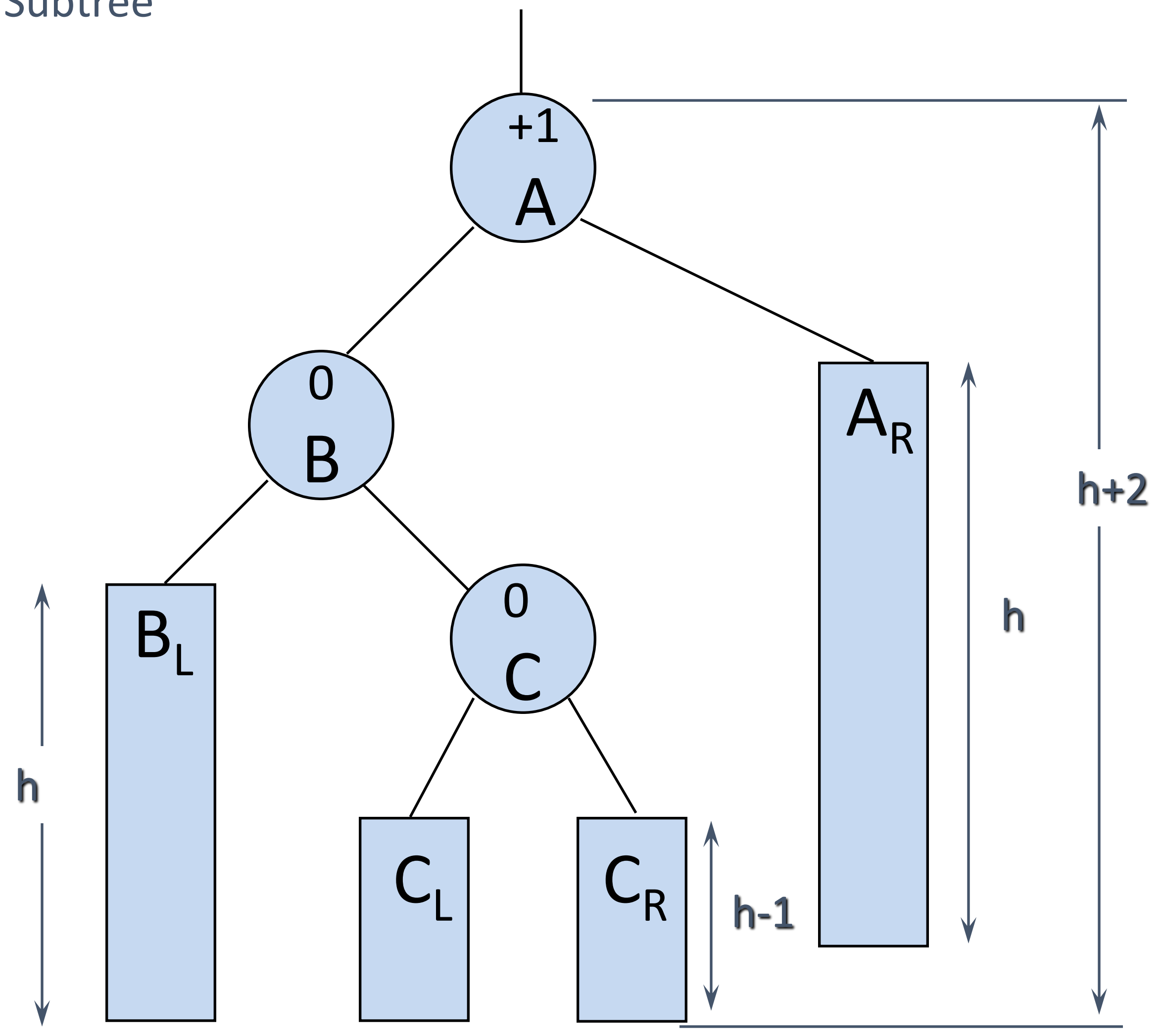
Example-Case 3



Do right rotation

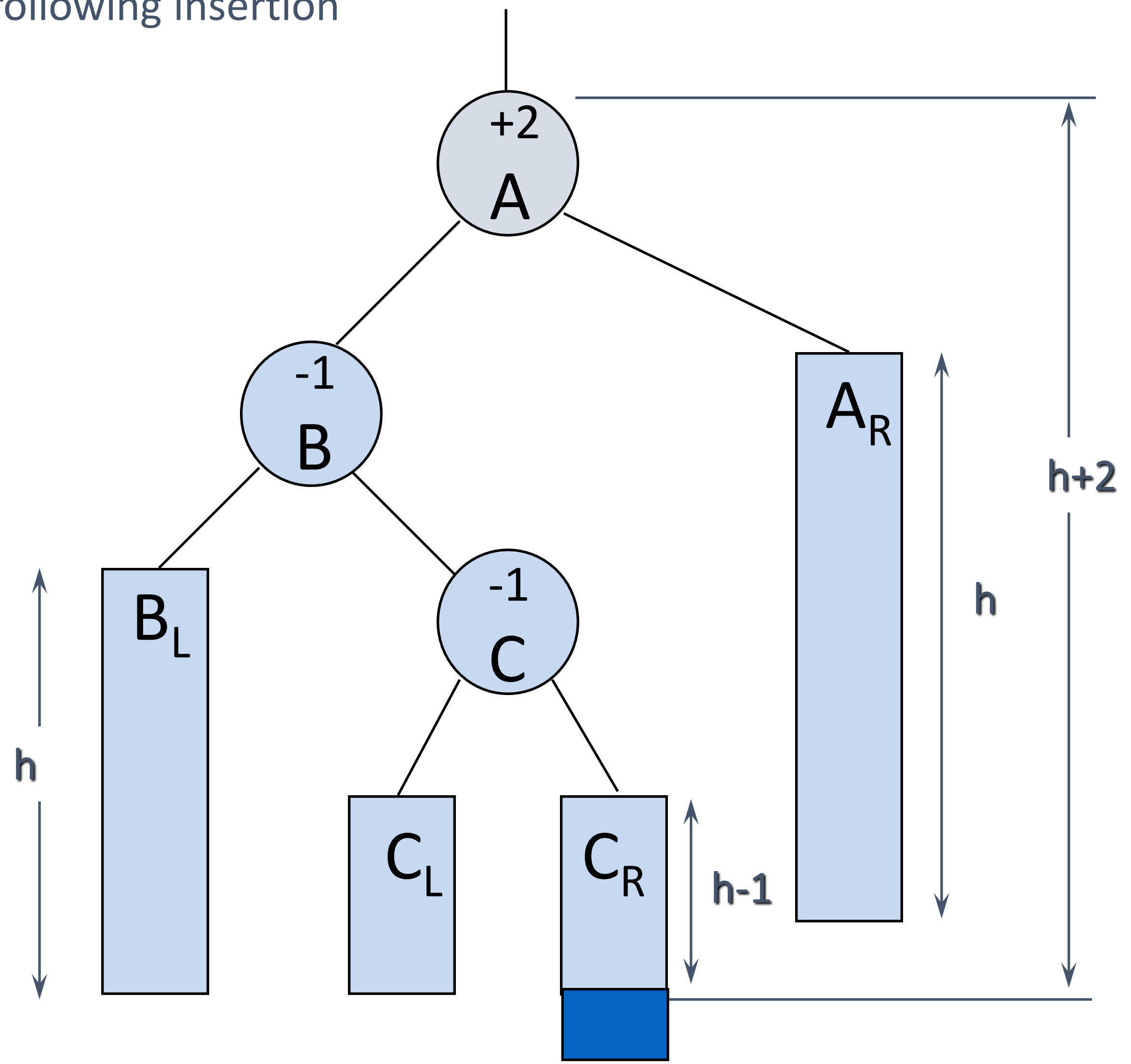
AVL Trees

Balanced Subtree

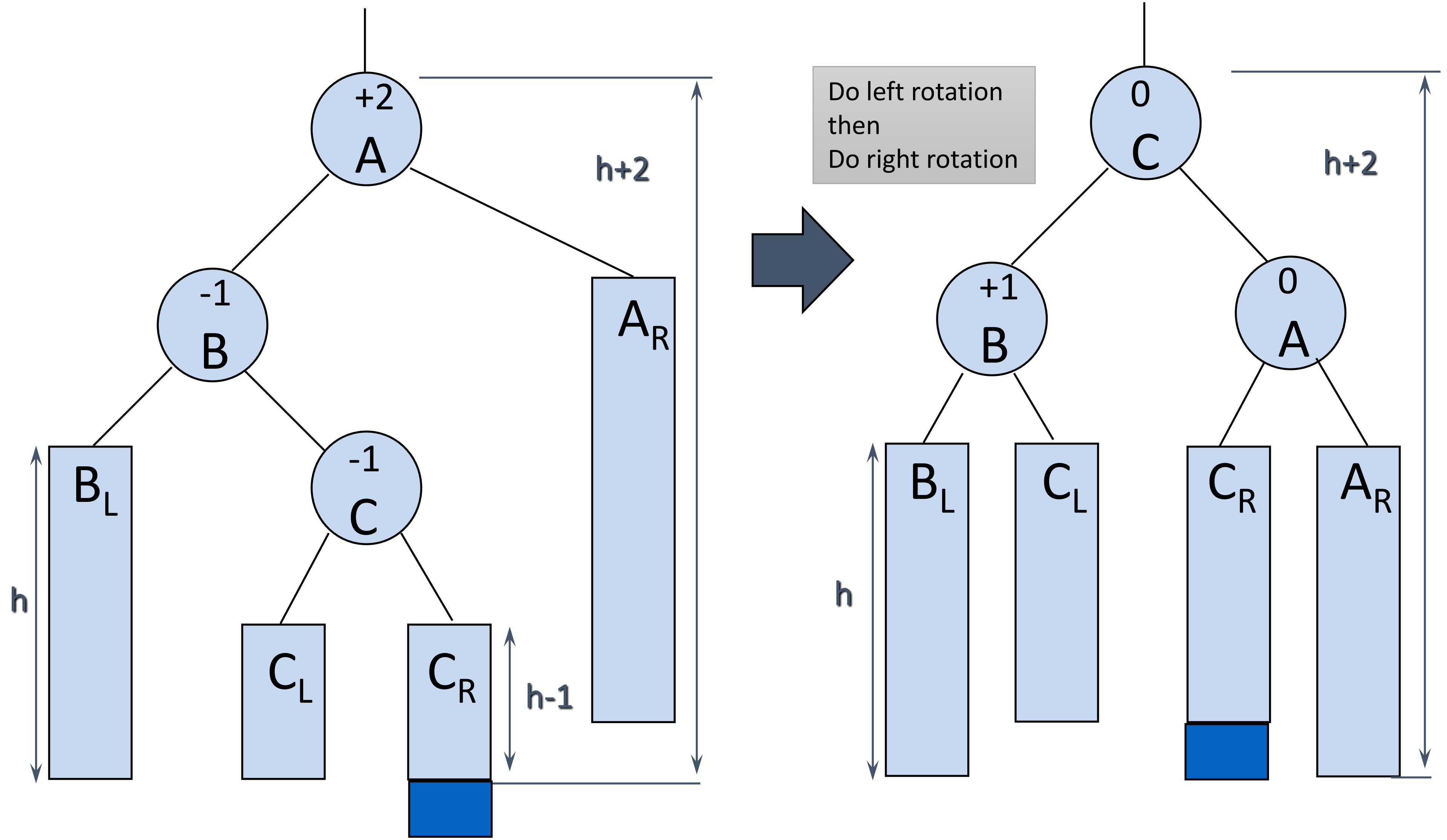


AVL Trees

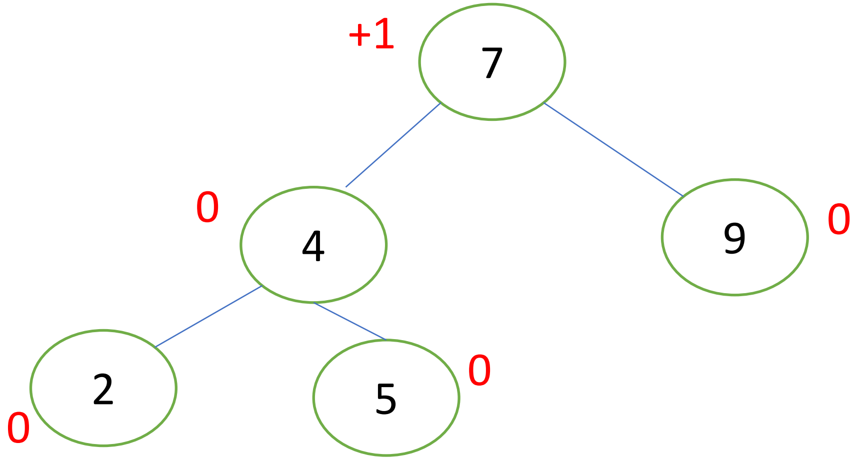
Unbalanced following insertion



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

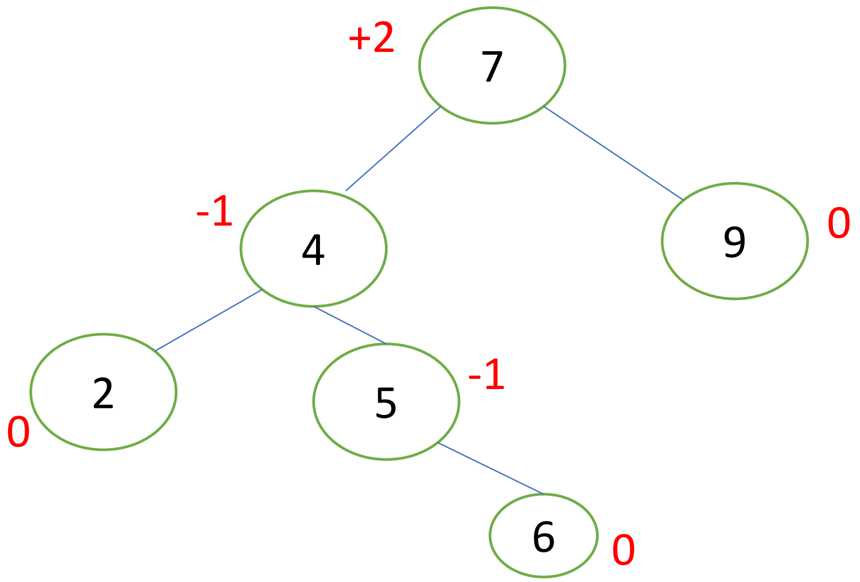


Another Example-Case 3

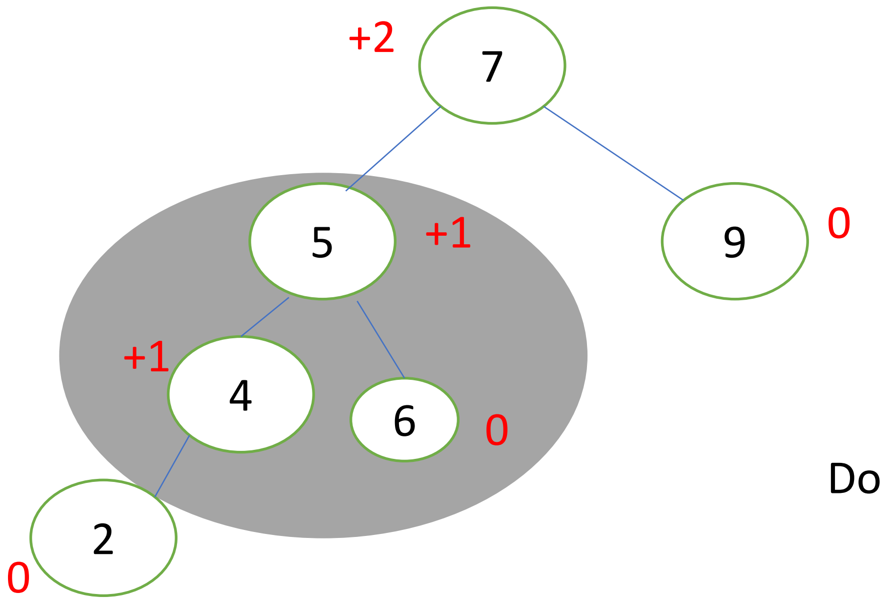


Example-Case 3

Insert 6

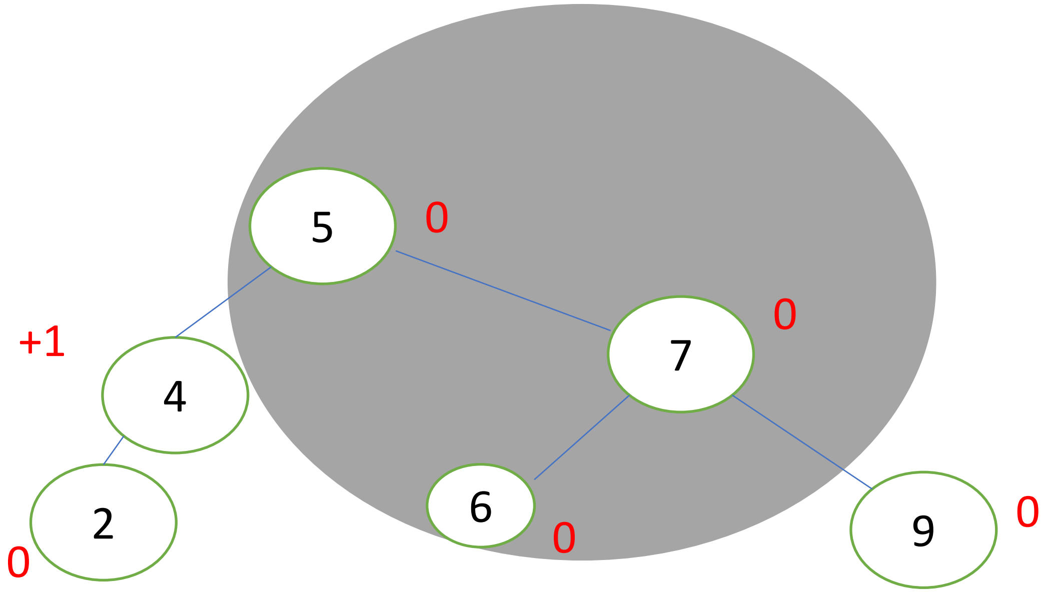


Another Example-Case 3



Do left rotation

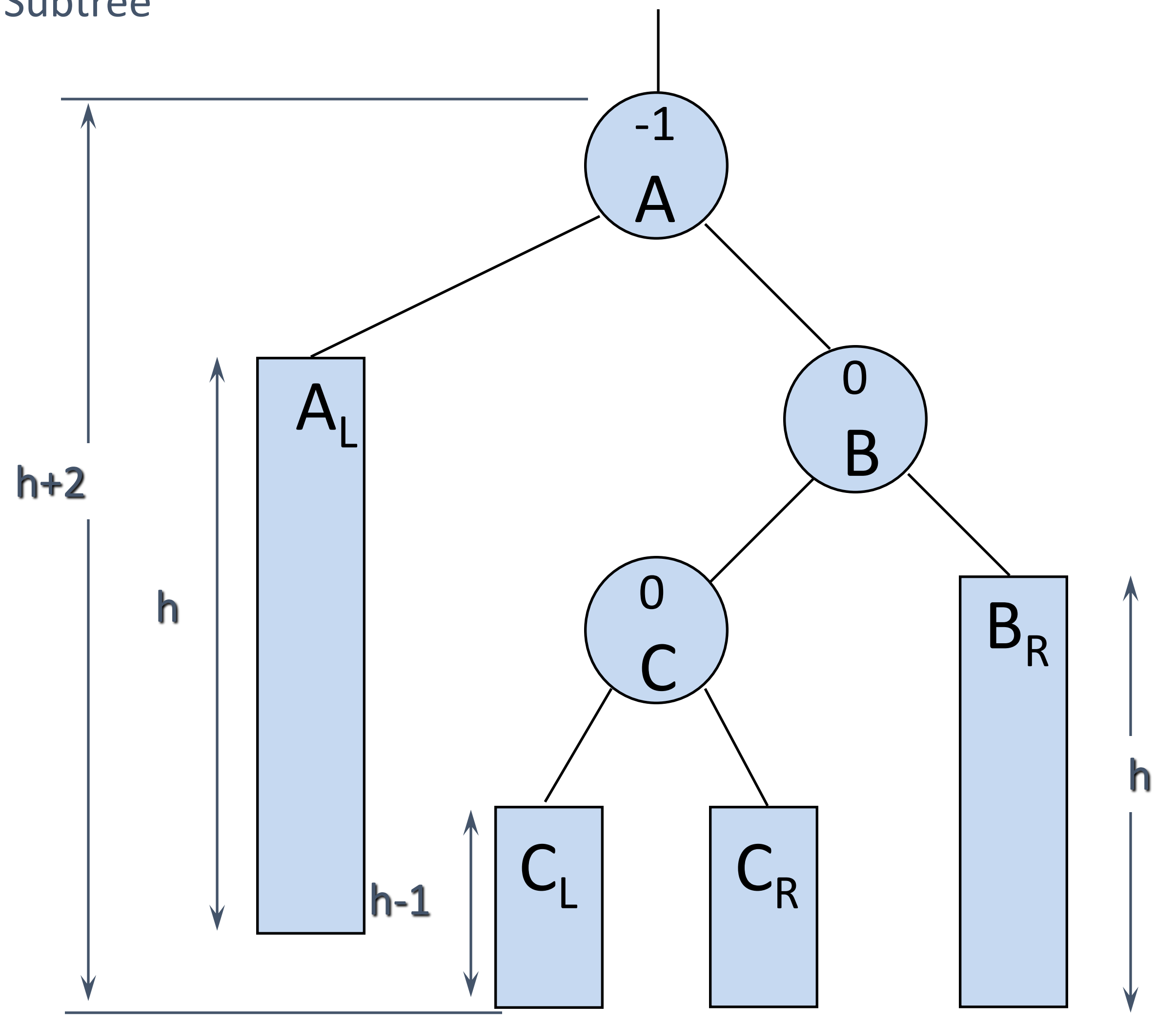
Another Example-Case 3



Do right rotation

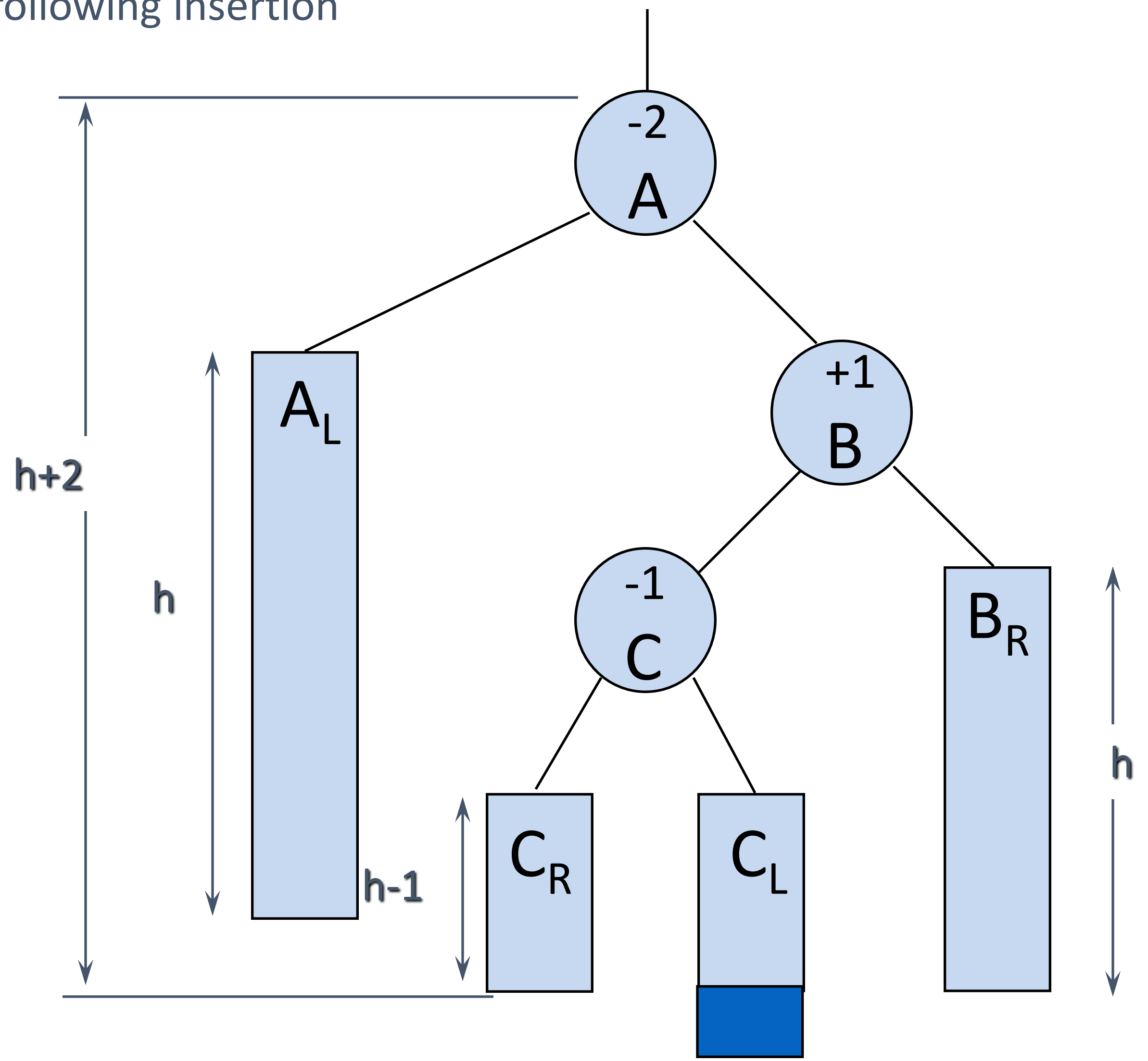
AVL Trees

Balanced Subtree

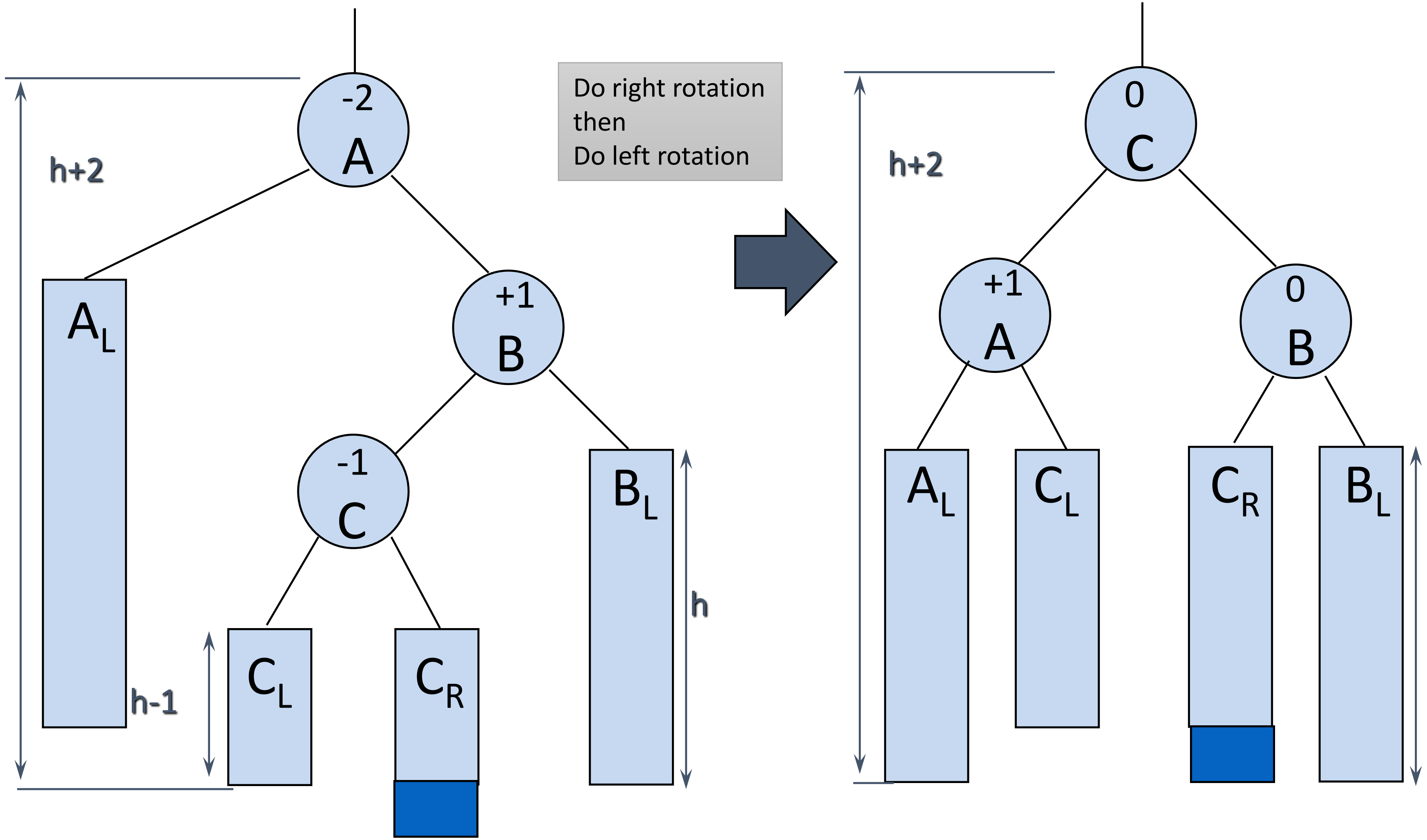


AVL Trees

Unbalanced following insertion



AVL Trees - RL rotation- Inside Case- Case 4



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]

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!).

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.