

# VE281

Data Structures and Algorithms

## AVL Trees

# Announcement

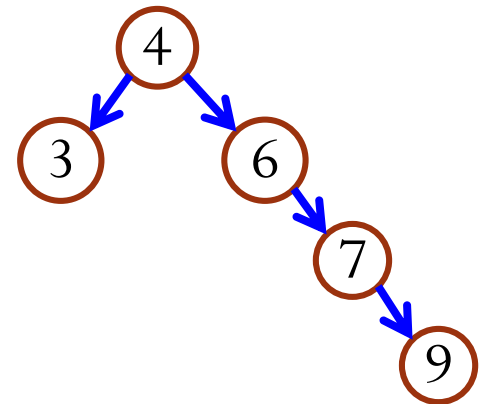
- Deadline of written assignment 4 extended to Nov. 8
  - See TA announcement for detailed submission deadline and place
- Will have a make-up lecture this Friday 2:00 – 3:40 pm
  - Classroom: East Middle Hall 2-101
- No lecture for the next two weeks (11/6~11/17)

# Outline

- Balanced Search Trees
  - AVL Trees
- AVL Tree Insertion
- Supporting Data Members and Functions of AVL Tree

# Motivation

- Given  $n$  nodes, the **average case** time complexities for search, insertion, and removal on BST are all  $O(\log n)$ .
- However, the **worst case** time complexities are still  $O(n)$ .
  - The reason is that a tree could become “**unbalanced**” after a number of insertions and removals.
- We want to maintain the tree as a “**balanced**” tree.



# Balanced Search Trees

- What are the requirements to call a tree a balanced tree?
- Would you require a tree to be perfect/complete to call it balanced?
  - No! They are too restrictive.



# Balanced Search Trees

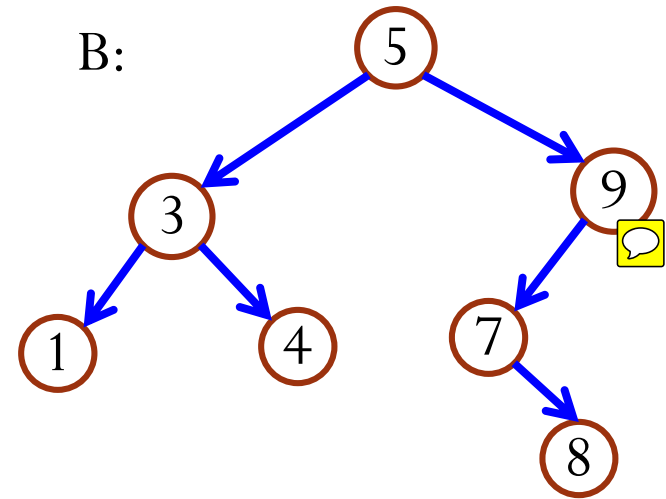
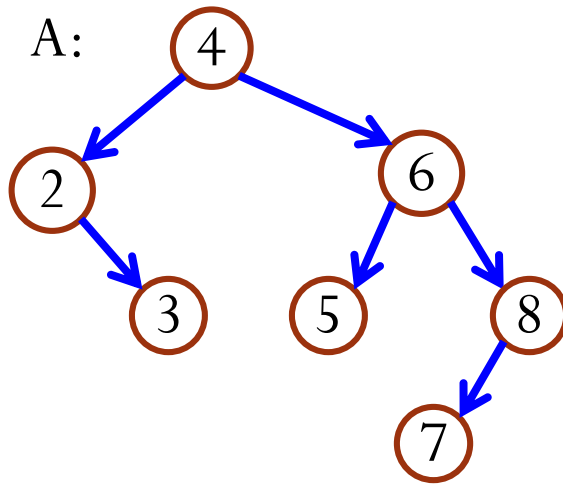
- We need another definition of “balanced condition.”
- We want the definition to satisfy the following two criteria:
  1. Height of a tree of  $n$  nodes =  $O(\log n)$ .
  2. Balance condition can be maintained **efficiently**:  $O(\log n)$  time to **rebalance** a tree.
- Several balanced search trees, each with its own balance condition
  - AVL trees
  - 2-3 trees
  - red-black trees

# AVL Trees

- Adelson-Velsky and Landis' trees
  - AVL tree is a **binary search tree**.
- AVL trees' balance condition:
  - An empty tree is **AVL balanced**.
  - A non-empty binary tree is **AVL balanced** if
    1. Both its left and right subtrees are AVL balanced, and
    2. The height of left and right subtrees differ by **at most 1**.

# AVL Trees

- Are the following trees AVL balanced?





# Properties of AVL Trees

- The height  $h$  of an AVL balanced tree with  $n$  internal nodes satisfies

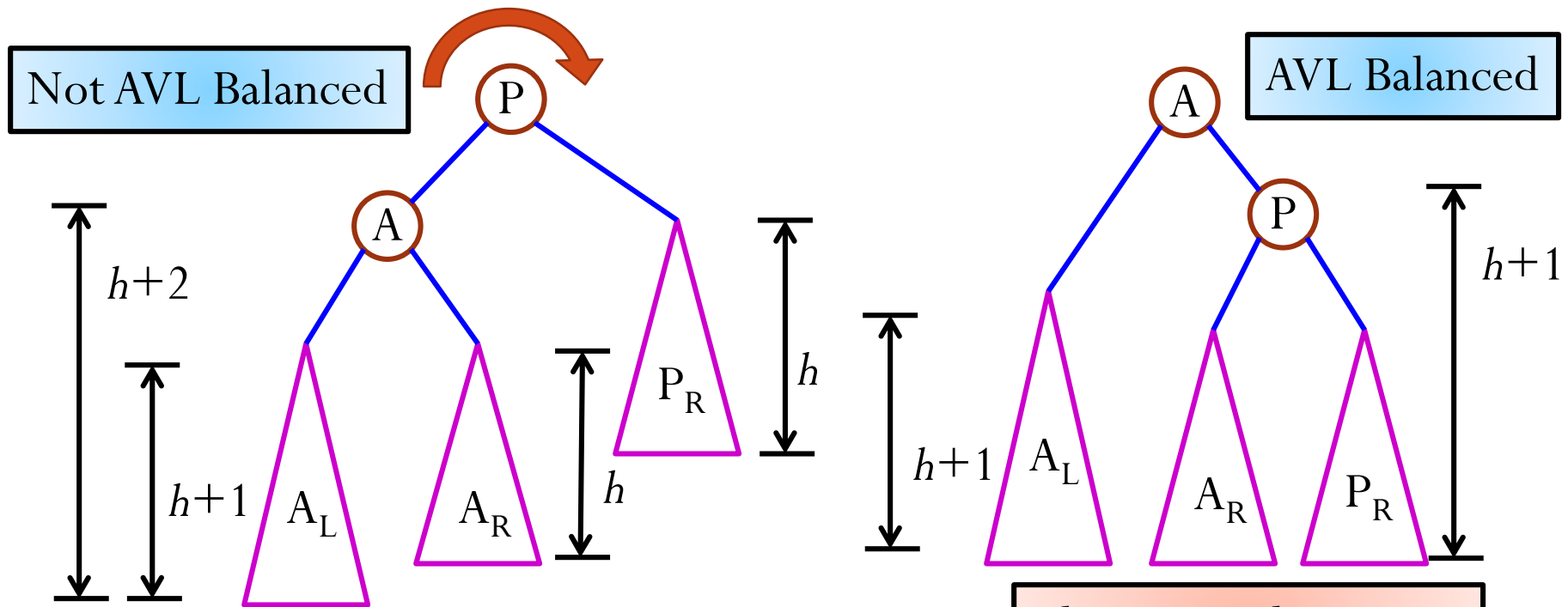
$$\log_2(n + 1) - 1 \leq h \leq 1.44 \log_2(n + 2)$$

- AVL trees satisfies the general “balanced condition” 1:
  - The height of a tree of  $n$  nodes is  $O(\log n)$ .
  - Search is guaranteed to always be  $O(\log n)$  time!
- We will also show that AVL trees satisfy the general “balance condition” 2:
  - Balance condition can be maintained **efficiently**.

# AVL Trees Operations

- Search, insertion, and removal all work exactly the same as with BST.
- However, after each insertion or removal, we must check whether the tree is still **AVL balanced**.
  - If not, we need to “**re-balance**” the tree.

# Re-Balance the Tree via Rotation

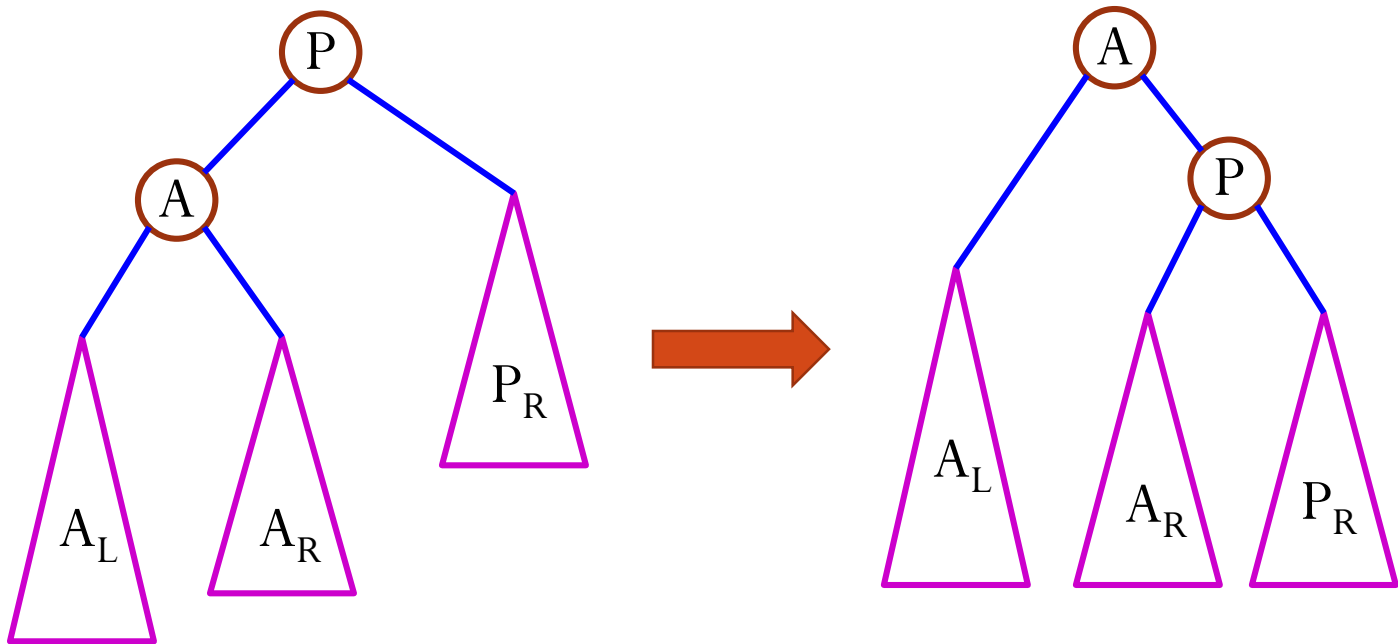


The BST ordering on keys are preserved.

- The rotation operation:
  - Interchange the role of **a parent** and **one of its children**, while still preserving the BST ordering on the keys.

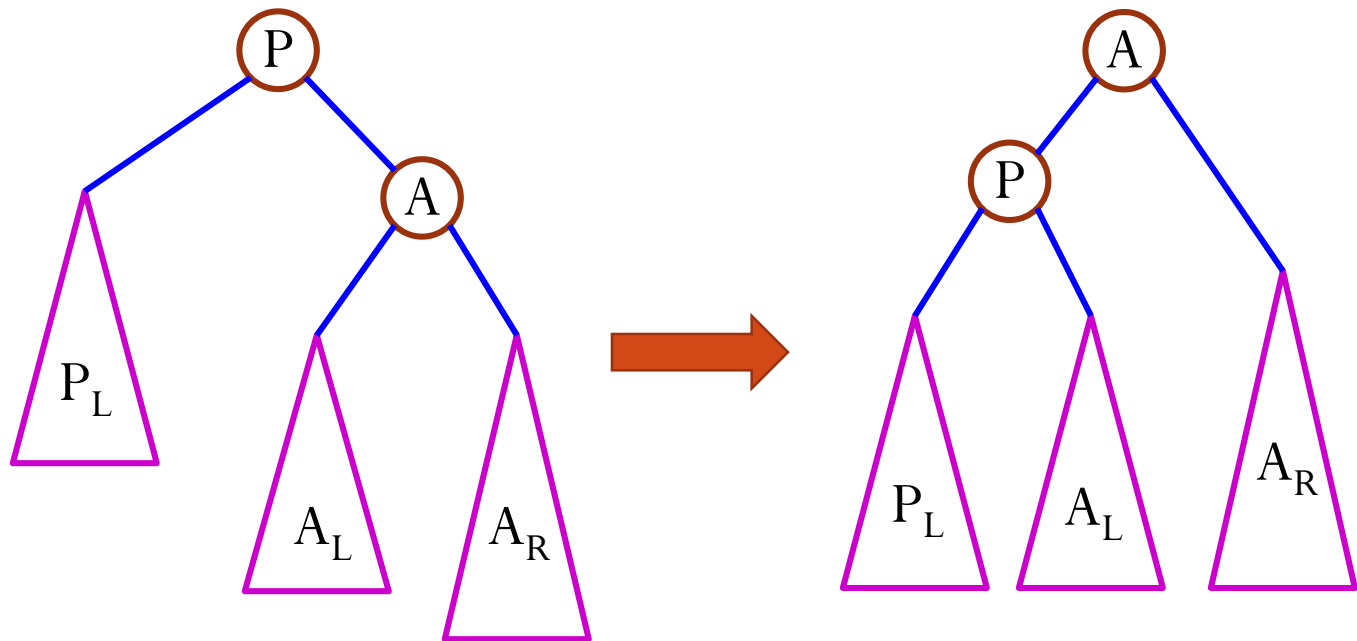
# Right Rotation

1. The right link of the **left child** becomes the left link of the **parent**.
2. **Parent** becomes right child of the **old left child**.



# Left Rotation

- The left link of the **right child** becomes the right link of the **parent**.
- **Parent** becomes left child of the **old right child**.

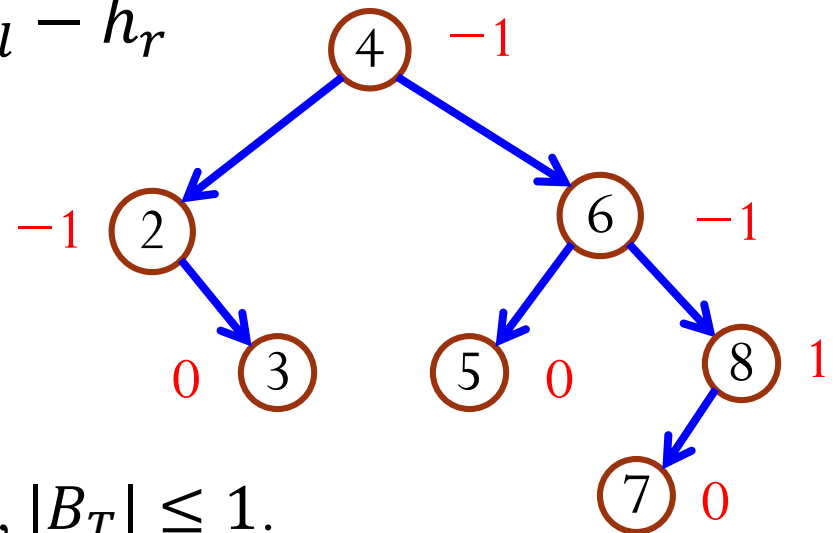


# Balance Factor

- Let  $T_l$  and  $T_r$  be the left and right subtrees of a tree rooted at node  $T$ .
- Let  $h_l$  be the height of  $T_l$  and  $h_r$  be the height of  $T_r$ .
- Define the **balance factor** ( $B_T$ ) of node  $T$  as

$$B_T = h_l - h_r$$

## Balance Factor Example



- AVL tree's balance condition:
  - For **every node**  $T$  in the tree,  $|B_T| \leq 1$ .

# Outline

- Balanced Search Trees
  - AVL Trees
- AVL Tree Insertion
- Supporting Data Members and Functions of AVL Tree

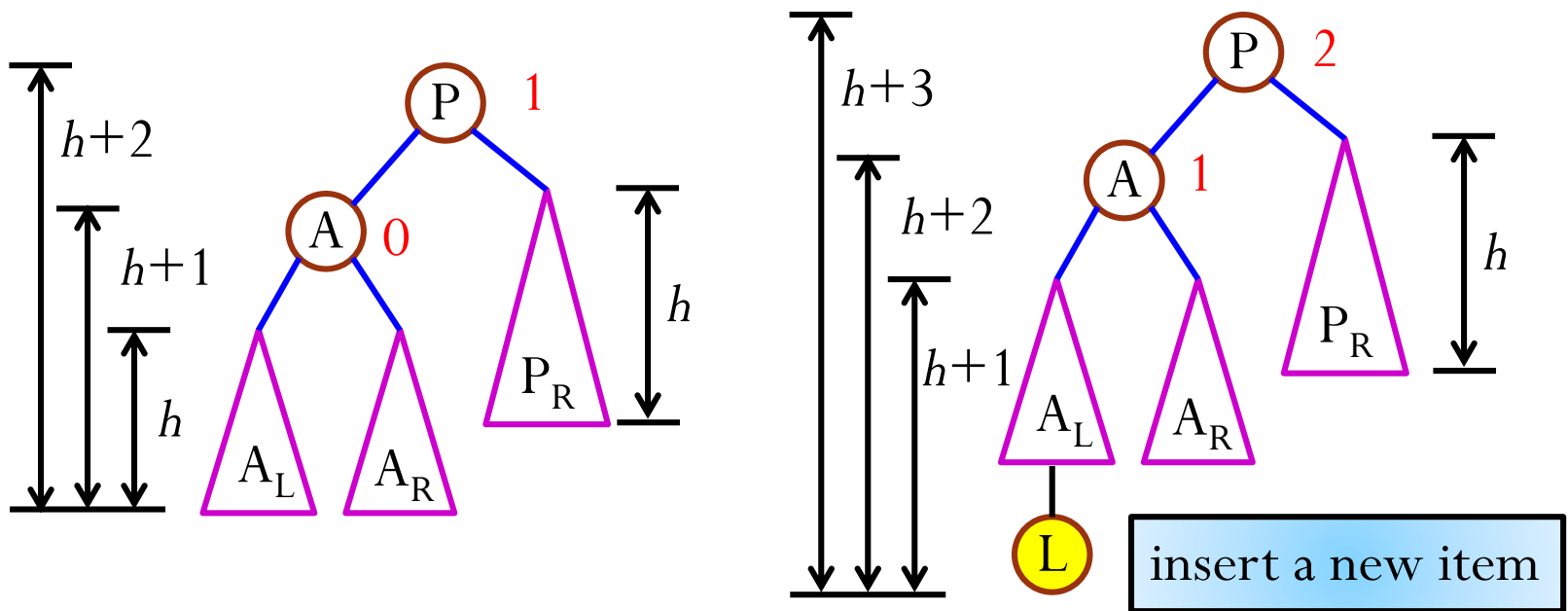
# Insertion

- Inserting an item in a tree affects potentially the heights of all of the nodes along the **access path**, i.e., the path from the root to that leaf.
- When an item is inserted in a tree, the height of any node on the access path may increase by one.
- To ensure the resulting tree is still AVL balanced, the heights of all the nodes along the access path must be **recomputed** and the AVL balance condition must be **checked**.
  - Sometimes, increasing the height by one does not violate the AVL balance condition.
  - In other cases, the AVL balance condition is violated.
  - We will fix **the first unbalanced node** in the access path **from the leaf**.



# Breaking AVL Balance Condition

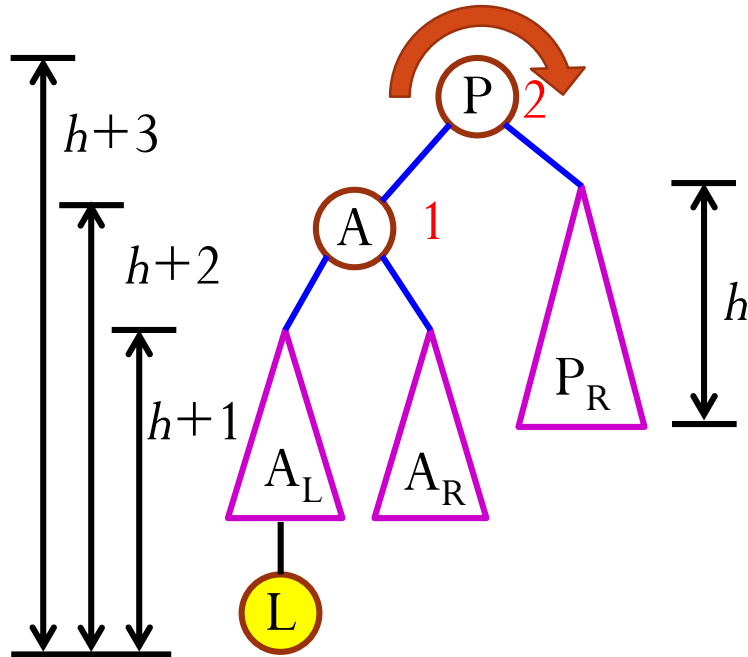
## Left-Left Insertion



**Left-left insertion:** the first two edges in the insertion path from node  $P$  both go to the left.

# Restoring AVL Balance Condition

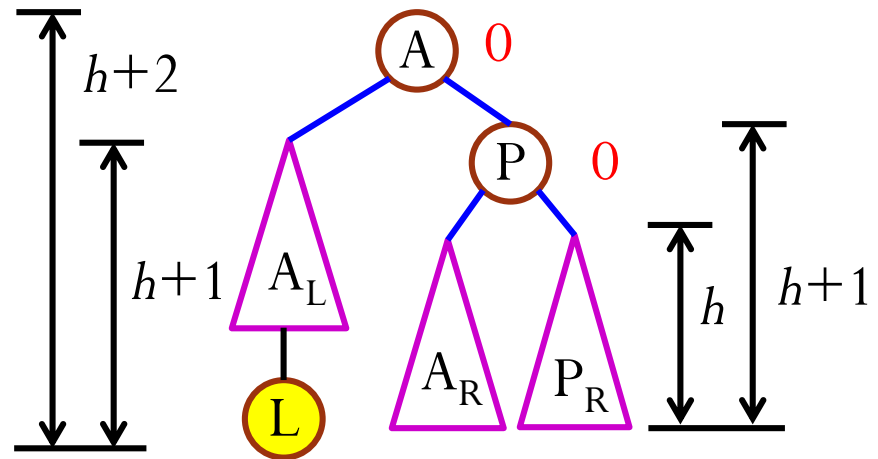
## Left-Left Rotation



How to restore AVL balance?

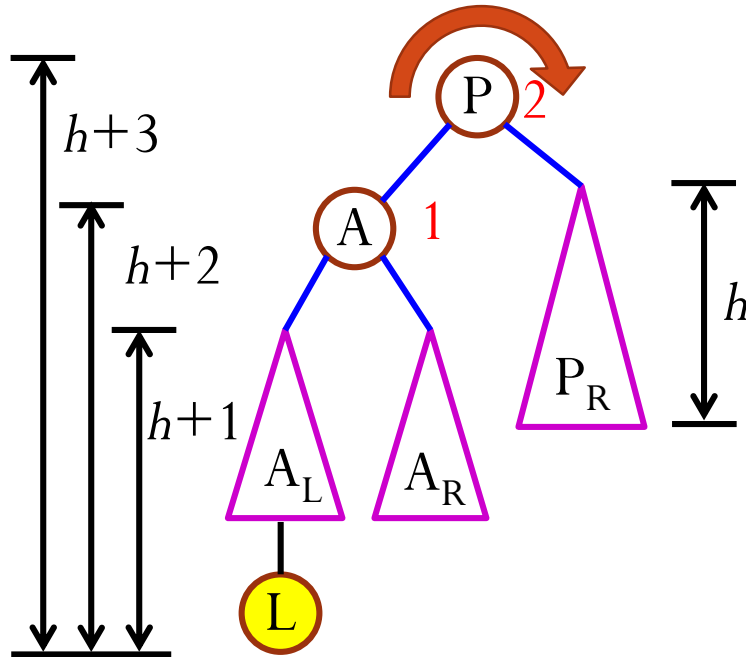
Do a right rotation at node P.

The rotation is also called **left-left (LL) rotation**.



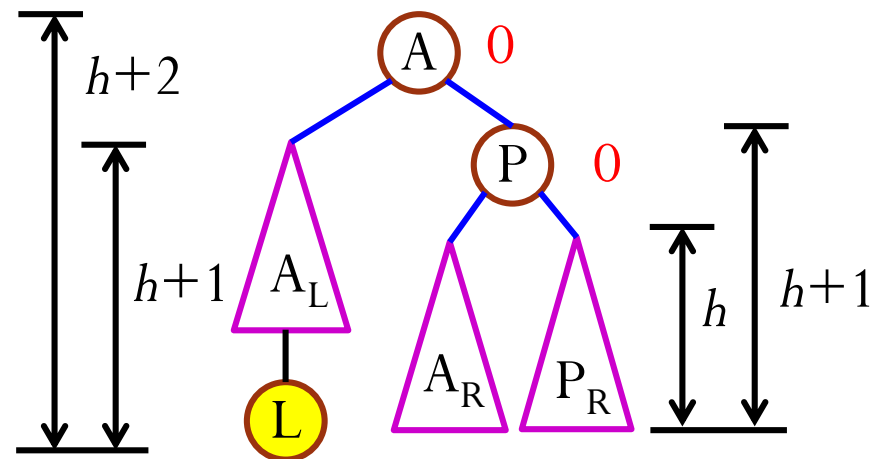
# Restoring AVL Balance Condition

## Left-Left Rotation



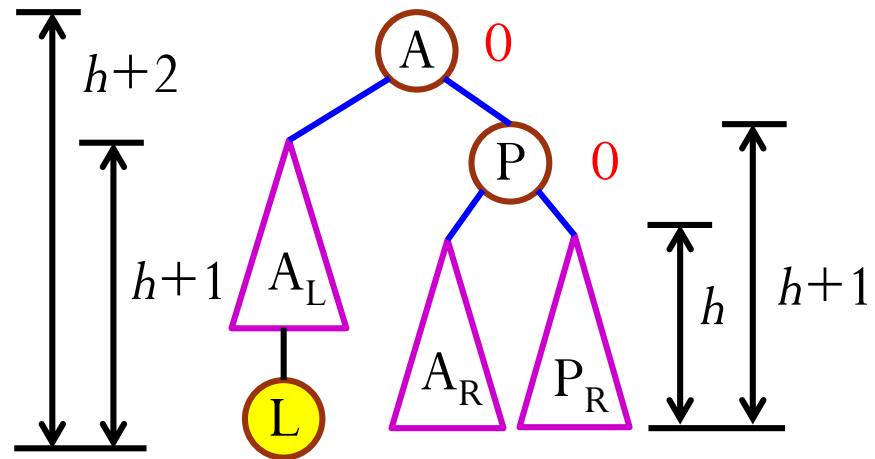
An **LL rotation** is called for when the node becomes unbalanced with a **positive** balance factor and the left subtree of the node also has a **positive** balance factor.

The rotation is also called **left-left (LL) rotation**.



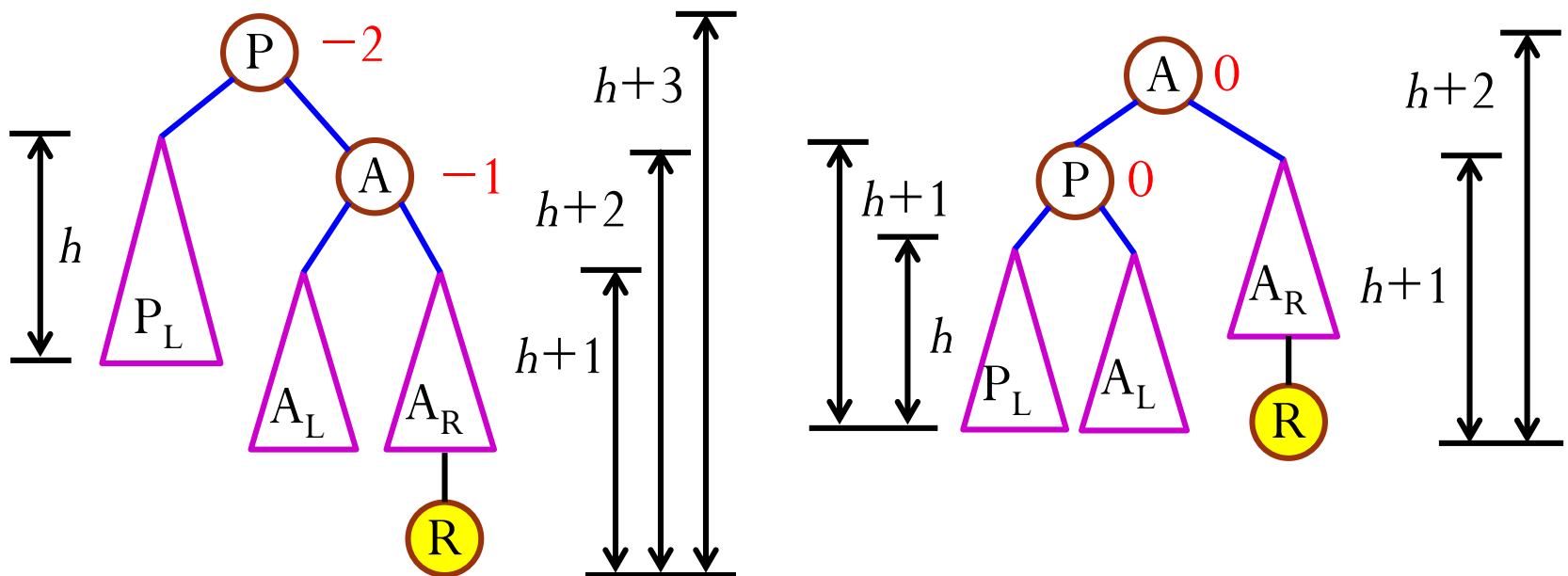
# Properties of Left-Left Rotation

- The ordering property of BST is kept.
- Both nodes A and P have balance factor of 0.
- The height of the tree **after the rotation** is the same as the height of the tree before insertion.



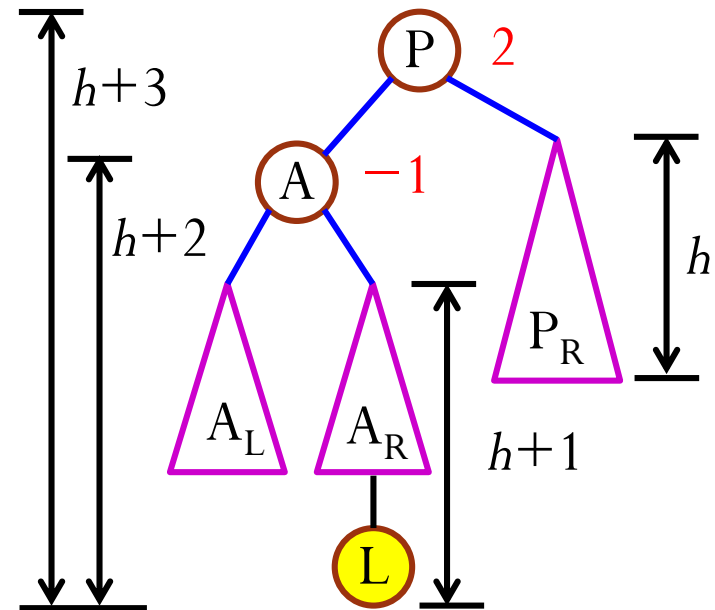
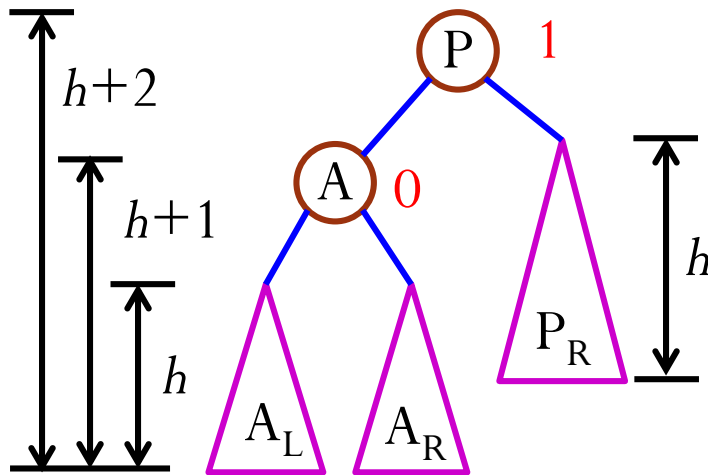
# Right-Right (RR) Rotation

- Symmetric to left-left rotation.
- An RR rotation is called for when the node becomes unbalanced with a **negative** balance factor and the right subtree of the node also has a **negative** balance factor.



# Breaking AVL Balance Condition

## Left-Right Insertion

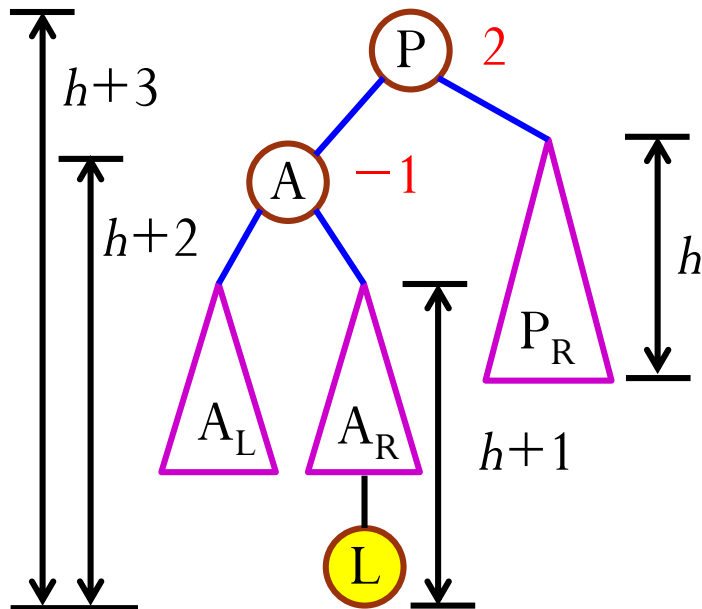


insert a new item

**Left-right insertion:** the first edge in the insertion path goes to the left and the second edge goes to the right.

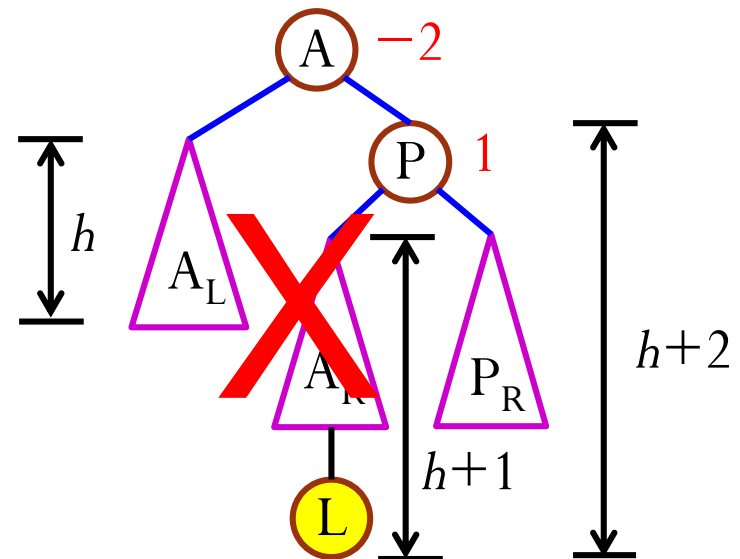
# Restoring AVL Balance Condition

## Left-Right Insertion

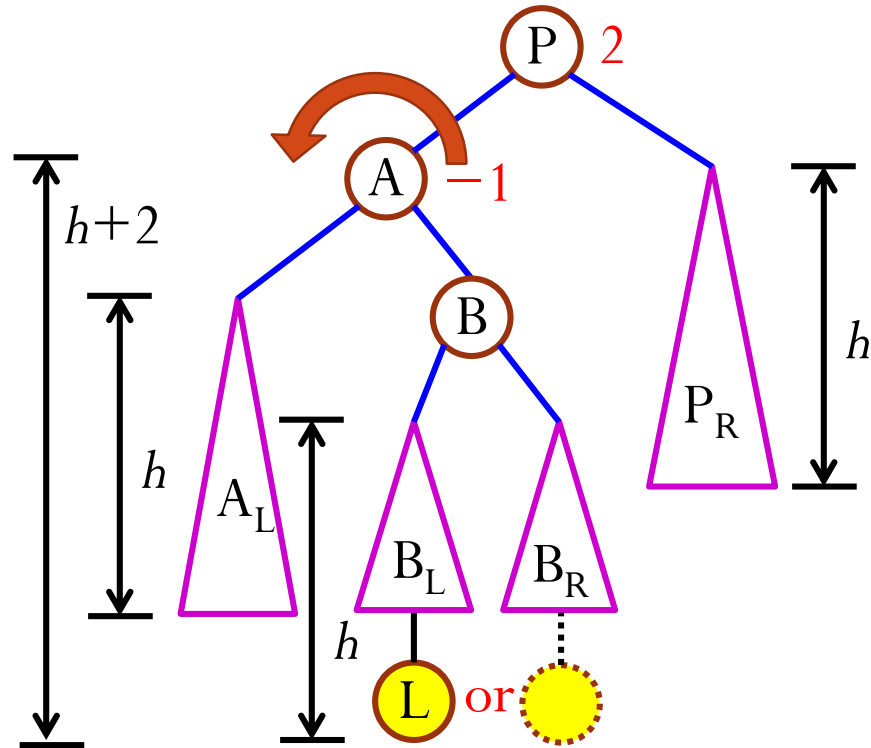


How to restore AVL balance?

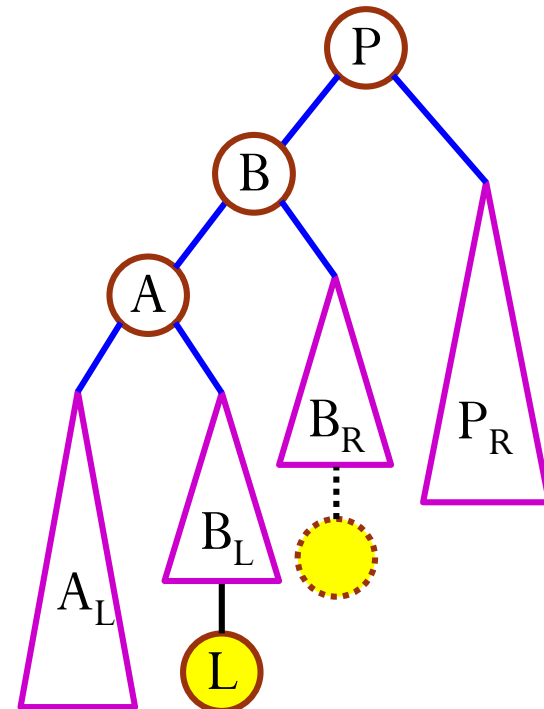
A right rotation at node  $P$  does not work!



# Left-Right (LR) Rotation

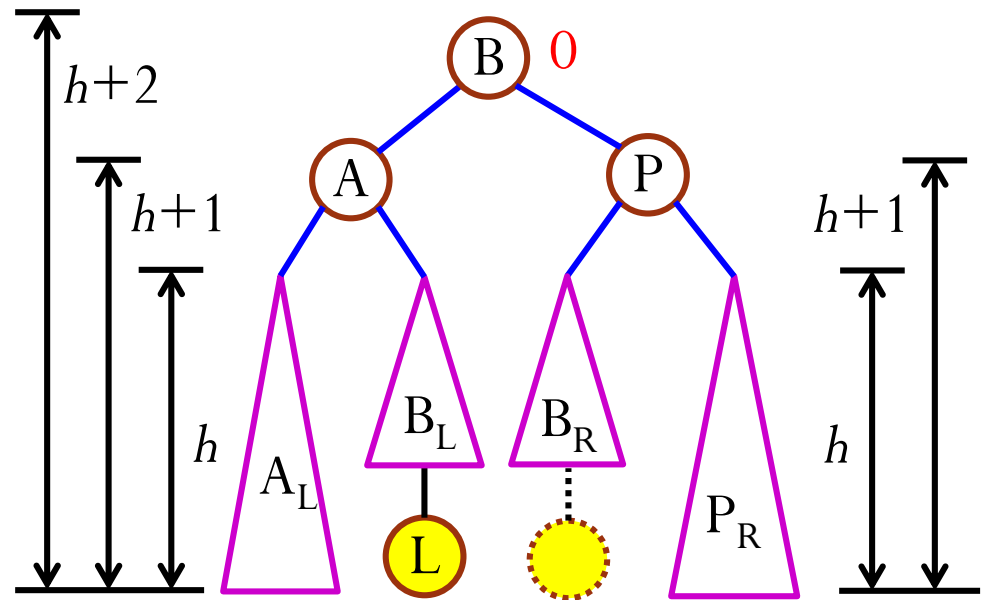
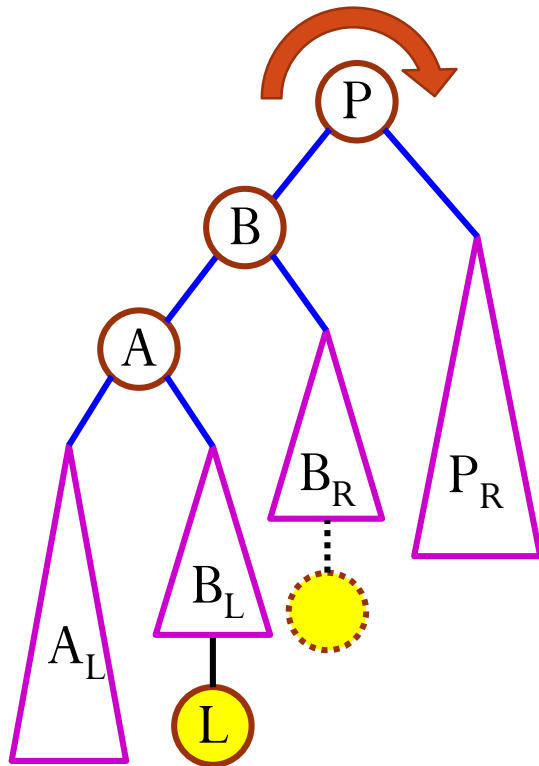


A **double rotation** to re-balance:  
Do a **left** rotation on node  $A$ ;  
then a **right** rotation on node  $P$   
(next slide).

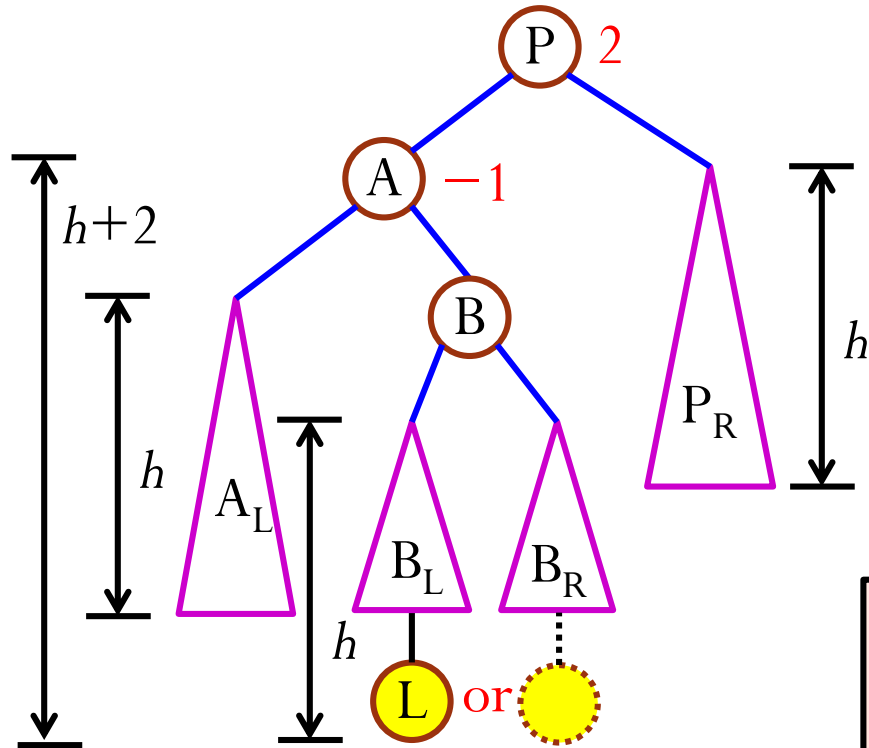




# Left-Right (LR) Rotation



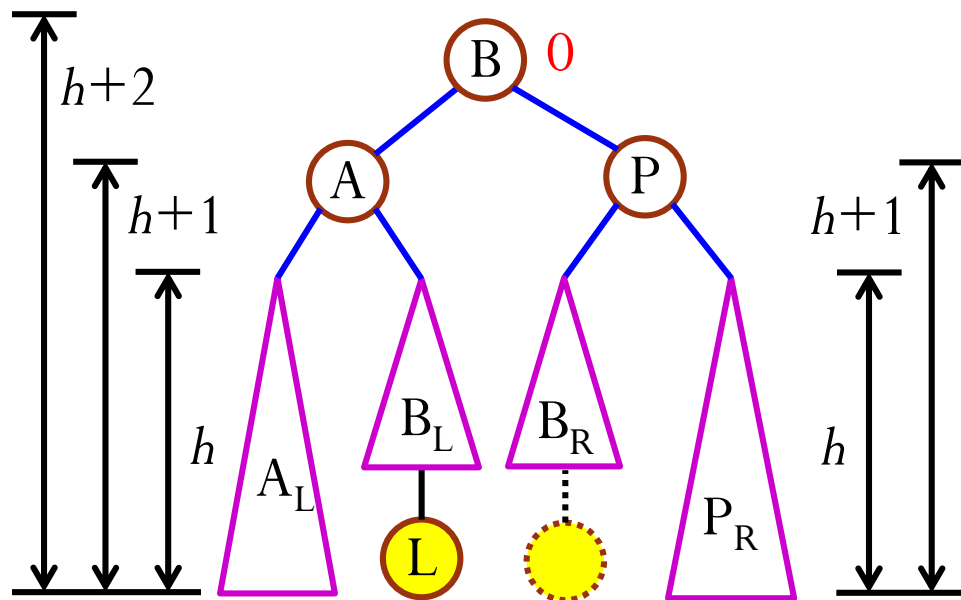
# Left-Right (LR) Rotation



An **LR rotation** is called for when the node becomes unbalanced with a **positive** balance factor but the left subtree of the node has a **negative** balance factor.

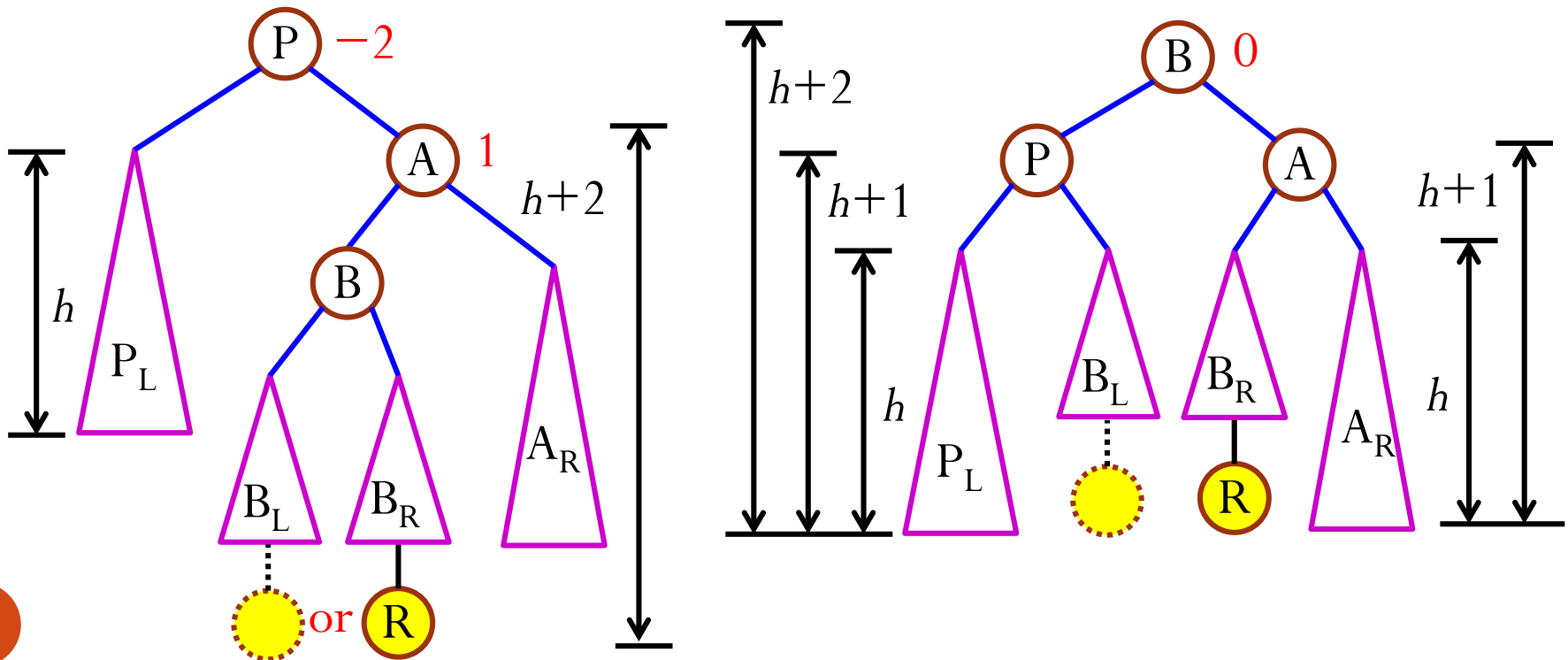
# Properties of Left-Right Rotation

- The ordering property of BST is kept.
- Node B has a balance factor of 0.
- The height of the tree **after the rotation** is the same as the height of the tree before insertion.



# Right-Left (RL) Rotation

- Symmetric to left-right rotation; also a double rotation.
- An **RL rotation** is called for when the node becomes unbalanced with a **negative** balance factor but the right subtree of the node has a **positive** balance factor.



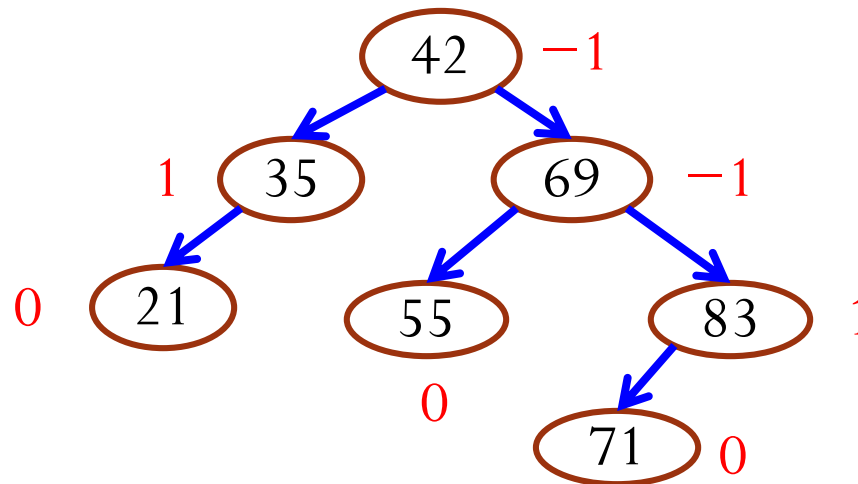
# Rotation Summary

- When an AVL tree becomes unbalanced, there are four cases to consider depending on the **direction** of the first two edges on the insertion path from the **unbalanced node**:
  - Left-left                      LL Rotation                      }
  - Right-right                      RR Rotation                      } single rotation
  - Left-right                      LR Rotation                      }
  - Right-left                      RL Rotation                      } double rotation

Note: We fix **the first unbalanced node** in the access path **from the leaf**.

# Exercises

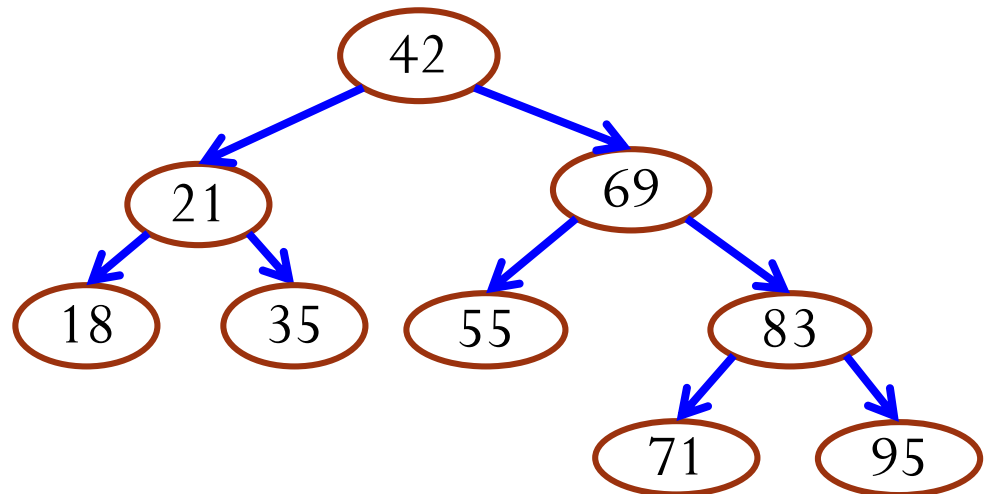
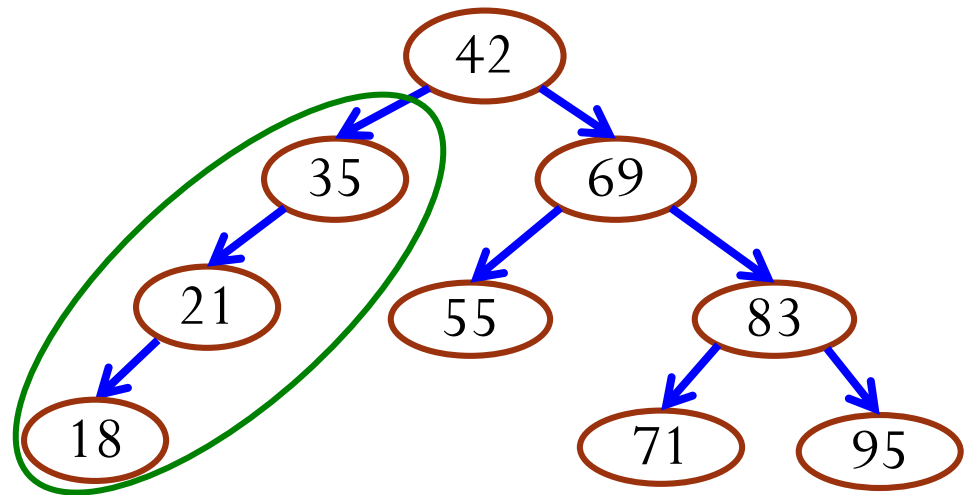
- Insert into an empty BST: 42, 35, 69, 21, 55, 83, 71.
  - Compute the balance factors.
  - Is the tree AVL balanced?



- Insert 95, 18, 75?

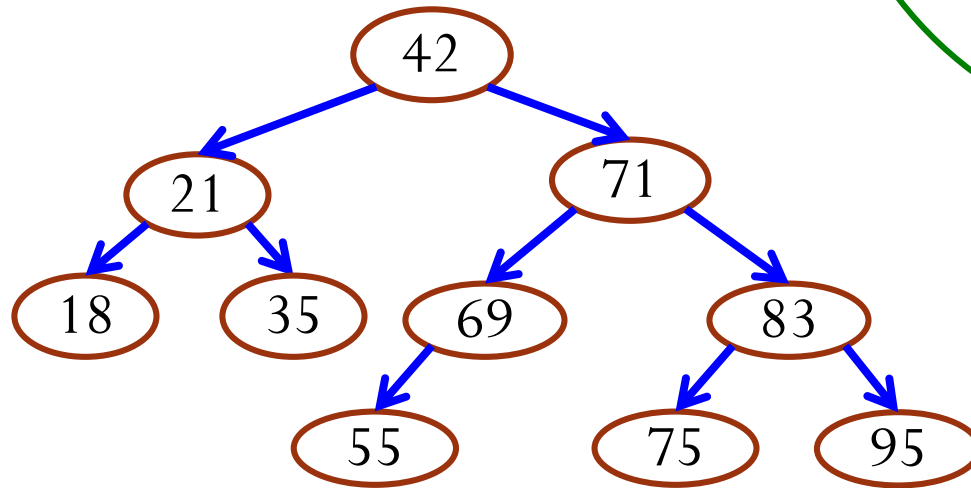
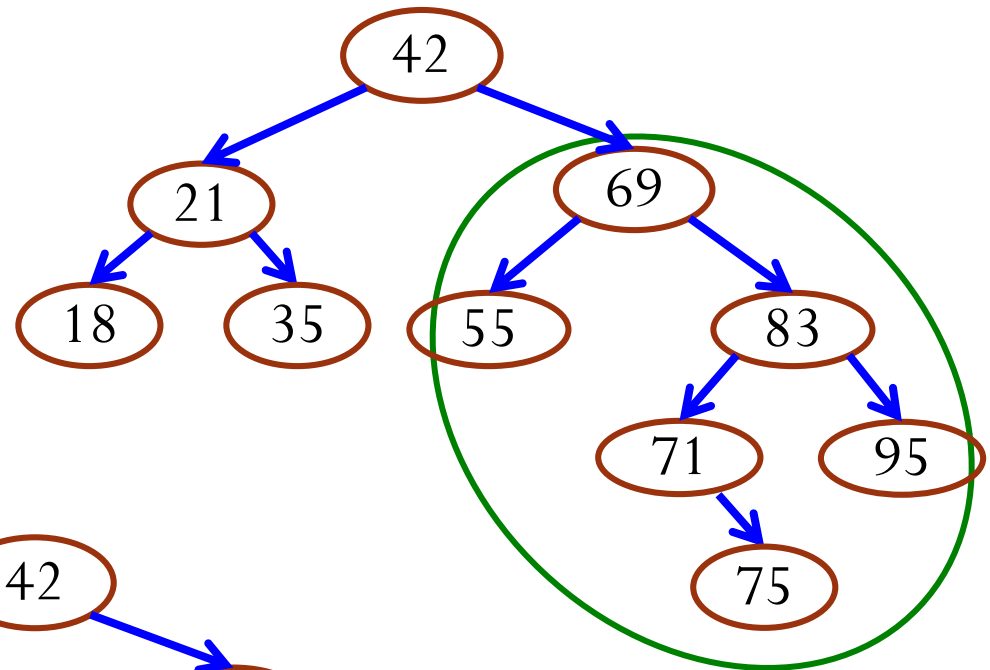
# Exercises

- Insert 95, 18



# Exercises

- Insert 75





# The Number of Rotations Required

- When an AVL tree **becomes unbalanced after an insertion**, **exactly one** single or double rotation is required to balance the tree.
  - Before the insertion, the tree is balanced.
  - Only nodes on the access path of the insertion can be unbalanced. All other nodes are balanced.
  - We rotate at the first unbalanced node **from the leaf**.
  - By the properties of rotation, the height of the node after rotation is the same as that before insertion.
  - All ancestors of that node on the access path should now be balanced.

# Outline

- Balanced Search Trees
  - AVL Trees
- AVL Tree Insertion
- Supporting Data Members and Functions of AVL Tree

# AVL Trees

## Supporting Data Members and Functions

```
struct node {  
    Item item;  
    int height;  
    node *left;  
    node *right;  
};
```

```
int Height(node *n) {  
    if(!n) return -1;  
    return n->height;  
}
```

```
void AdjustHeight(node *n) {  
    if(!n) return;  
    n->height = max( Height(n->left),  
                    Height(n->right) ) + 1;  
}
```

```
int BalFactor(node *n) {  
    if(!n) return 0;  
    return (Height(n->left) -  
            Height(n->right));  
}
```

# AVL Trees

## Supporting Functions

```
void LLRotation(node *&n) ;  
void RRRotation(node *&n) ;  
void LRRotation(node *&n) ;  
void RLRotation(node *&n) ;
```

```
void Balance(node *&n) {  
    if(BalFactor(n) > 1) {  
        if(BalFactor(n->left) > 0) LLRotation(n) ;  
        else LRRotation(n) ;  
    }  
    else if(BalFactor(n) < -1) {  
        if(BalFactor(n->right) < 0) RRRotation(n) ;  
        else RLRotation(n) ;  
    }  
}
```

# AVL Trees

## Changes to Insertion

```
void insert(node *&root, Item item)
{
    if(root == NULL) {
        root = new node(item);
        return;
    }
    if(item.key < root->item.key)
        insert(root->left, item);
    else if(item.key > root->item.key)
        insert(root->right, item);

    AdjustHeight(root);
    Balance(root);
}
```

# Removal

- First remove node as with BST
- Then update the balance factors of those ancestors in the access path and rebalance as needed.