# AVL tree

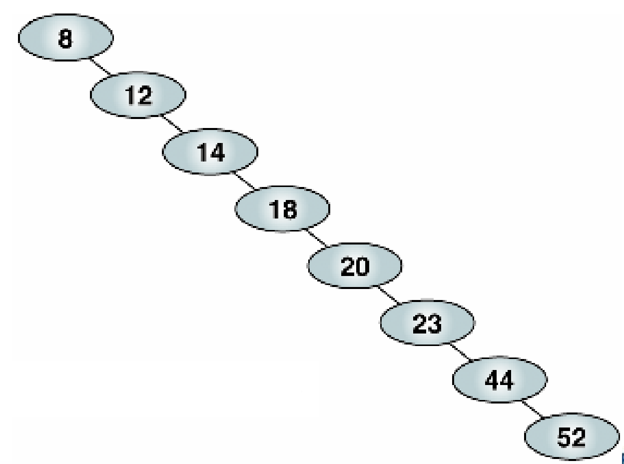
서승현 교수
Prof. Anes Seung-Hyun Seo
(seosh77@hanyang.ac.kr)

Division of Electrical Engineering
Hanyang University, ERICA Campus

# Balanced Search Tree

## ■ Motivation

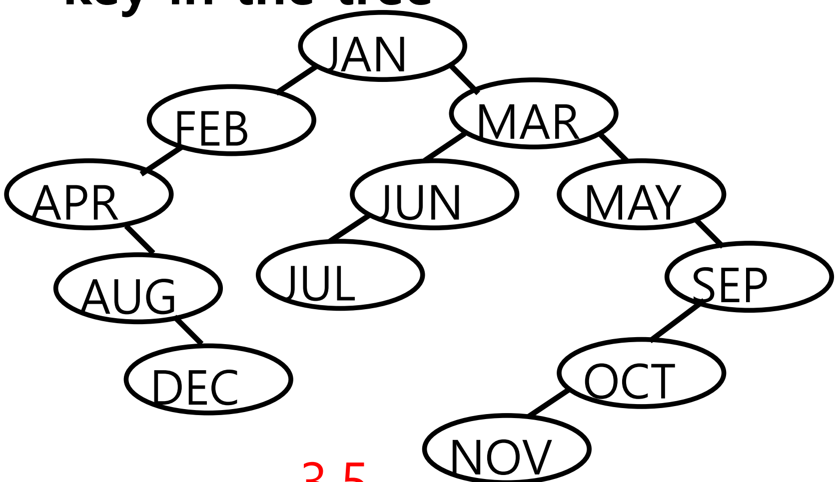
- Complexity of BST
  - Average case: O(log n)
    - Inserting sorted data
  - Worst case: O(n)
    - If BST is unbalanced, the efficiency is bad.
- Binary search tree is efficient when it is balanced.
- But, inserting/removing destroys the balanced structure of BST!




## ■ Remedy: balanced tree

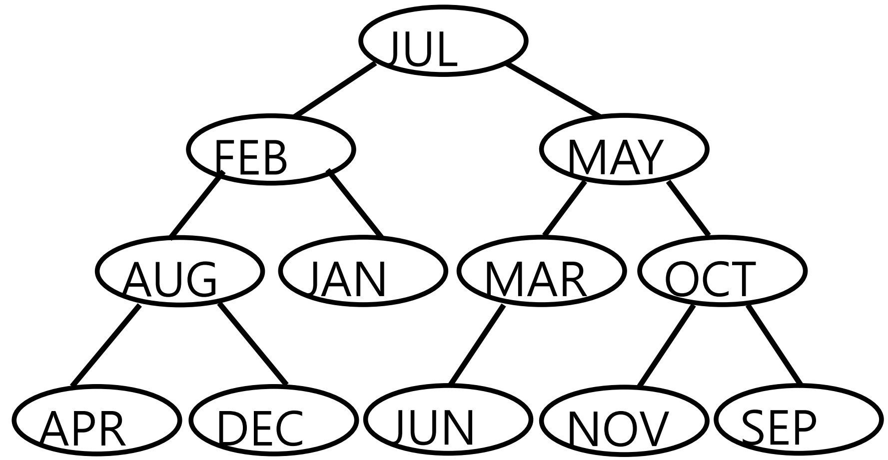
# Binary Search Tree

- **The maximum number of comparisons needed to search for any key in the tree**



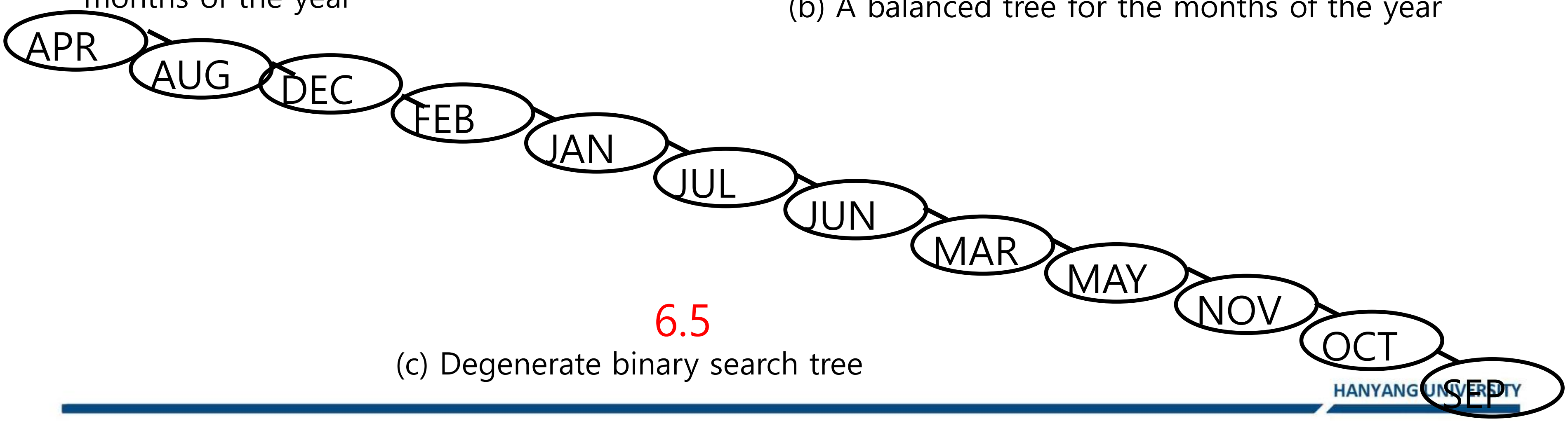

3.5

(a) Binary search tree obtained for the months of the year




3.1

(b) A balanced tree for the months of the year

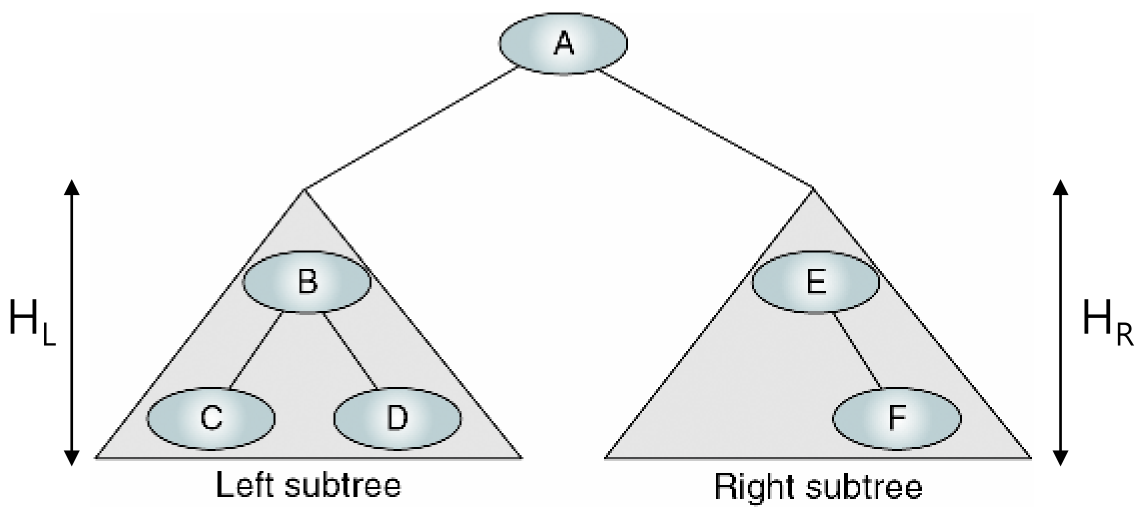



6.5

(c) Degenerate binary search tree

# Balance Factor of Binary Tree

- **Balance**
  - Balance factor (B) = $H_L - H_R$
  - Balanced binary tree: $|B| \le 1$



# Self-balancing BST

- **Automatically adjusts its structure to a balanced tree in inserting/removing**

- **Performance**
  - Worst case = O(log n)
    → better than that of plain BST = O(n)
    → better than that of hashing = O(n)

  - Average case = O(log n)
    → worse than that of hashing = O(1)

# AVL Tree

## The first self-balancing BST

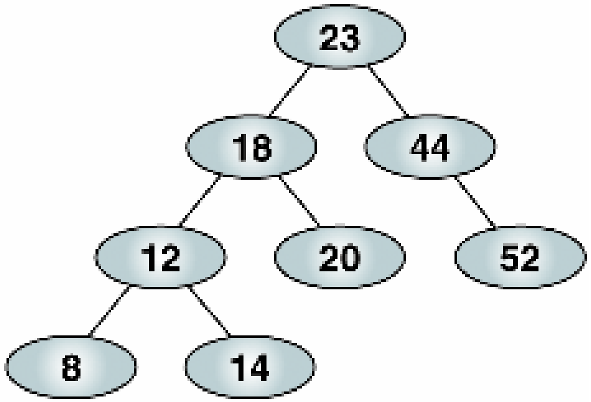
- Adelson-Velsky and Landis tree
- The structure is re-adjusted for each insert/remove
- Height differences of subtrees are less than one

## Tree adjustment

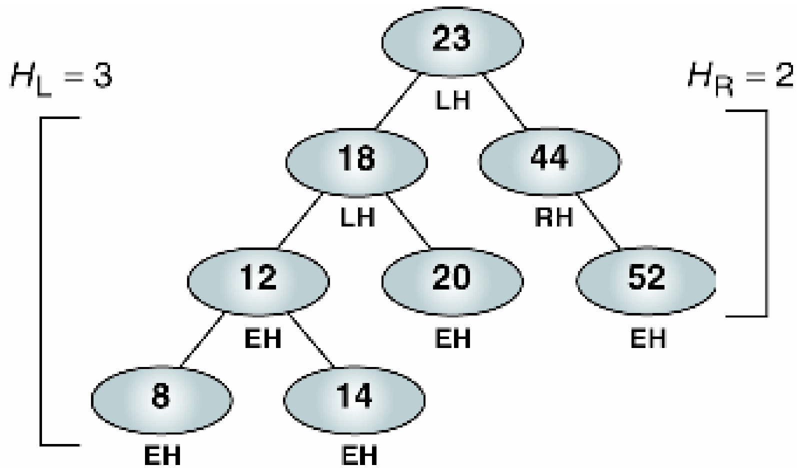
- Based on tree rotations
- Tree rotations take O(log n) for each insert/remove

# AVL Tree

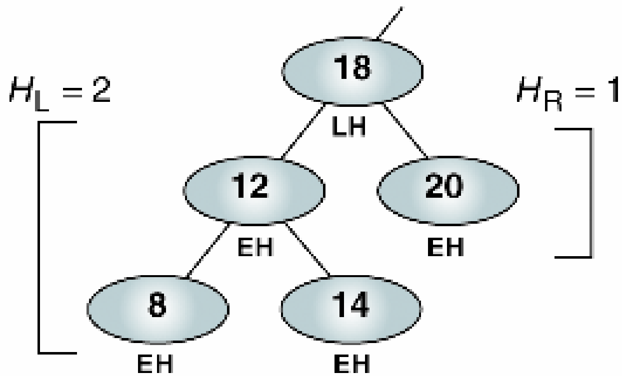
- **AVL Tree: a binary search tree in which the heights of the subtrees differ by no more than 1**
  - $|H_L - H_R| \leq 1$ for all nodes
    
    Cf. $H_L - H_R$ is called **balance factor**
    - An AVL is a **balanced tree**
  - Invented by G.M **A**delson-**V**elskii and E.M. **L**andis, 1962.



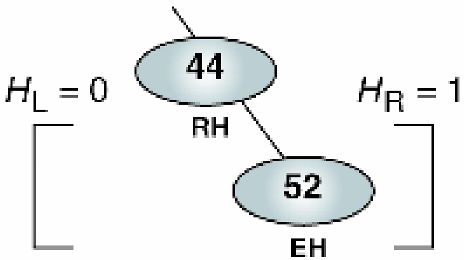
# AVL Tree




(a) Tree 23 appears balanced: $H_L - H_R = 1$



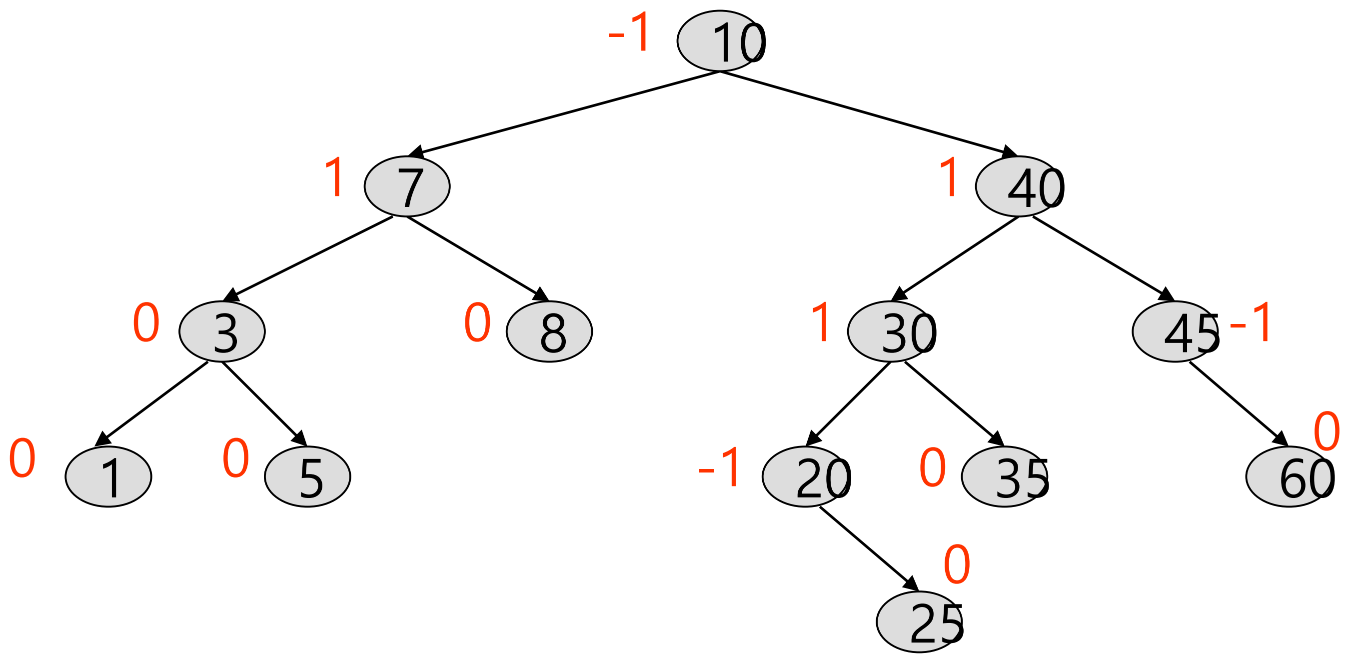

(b) Subtree 18 appears balanced: $H_L - H_R = 1$




(c) Subtree 44 is balanced: $| H_L - H_R | = 1$

# AVL Tree

- **Ex) Is it AVL tree?**



# AVL Tree

## Definition of height-balanced tree

- An empty tree is height-balanced.
- If T is a nonempty binary tree with $T_L$ and $T_R$ as its left and right subtrees respectively, then T is height-balanced iff
  - $T_L$ and $T_R$ are height-balanced.
  - $|h_L - h_R| \le 1$ where $h_L$ and $h_R$ are the heights of $T_L$ and $T_R$, respectively
  - $T_L$ : left subtree, $T_R$ : right subtree

## Balance factor, BF(T) of a node T in a binary tree

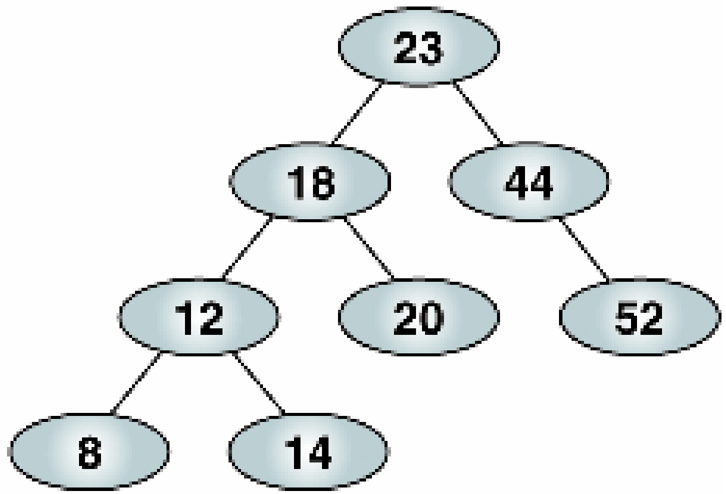
- Difference between the height of the left and right subtrees of T
- $BF(T) = h_L - h_R$
- For any node T in an AVL tree, BF(T)= -1, 0, 1

# Insertion/Deletion in AVL Tree

- **Insertion or deletion may break the balance of AVL tree.**

  Ex) Inserting 15

  Deleting 52

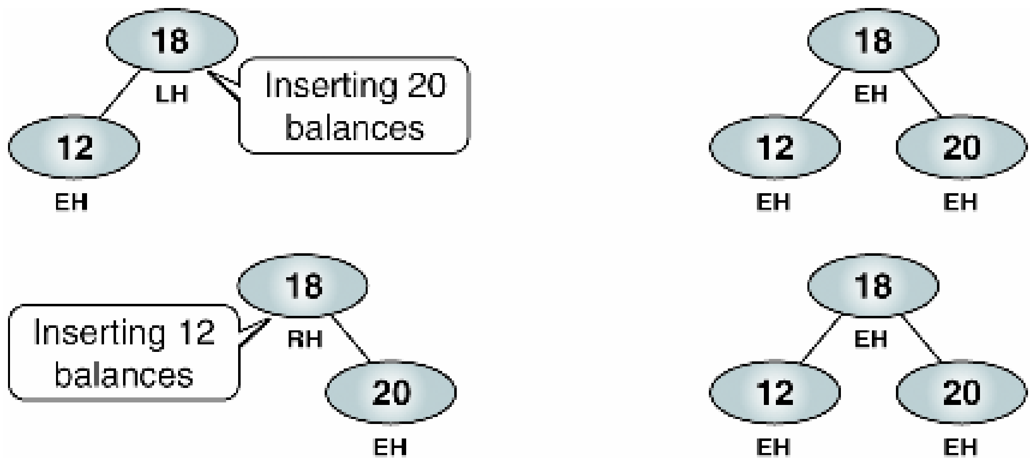



- **Remedy: Rebalance the tree after insertion or deletion**

# Insertion into AVL Trees

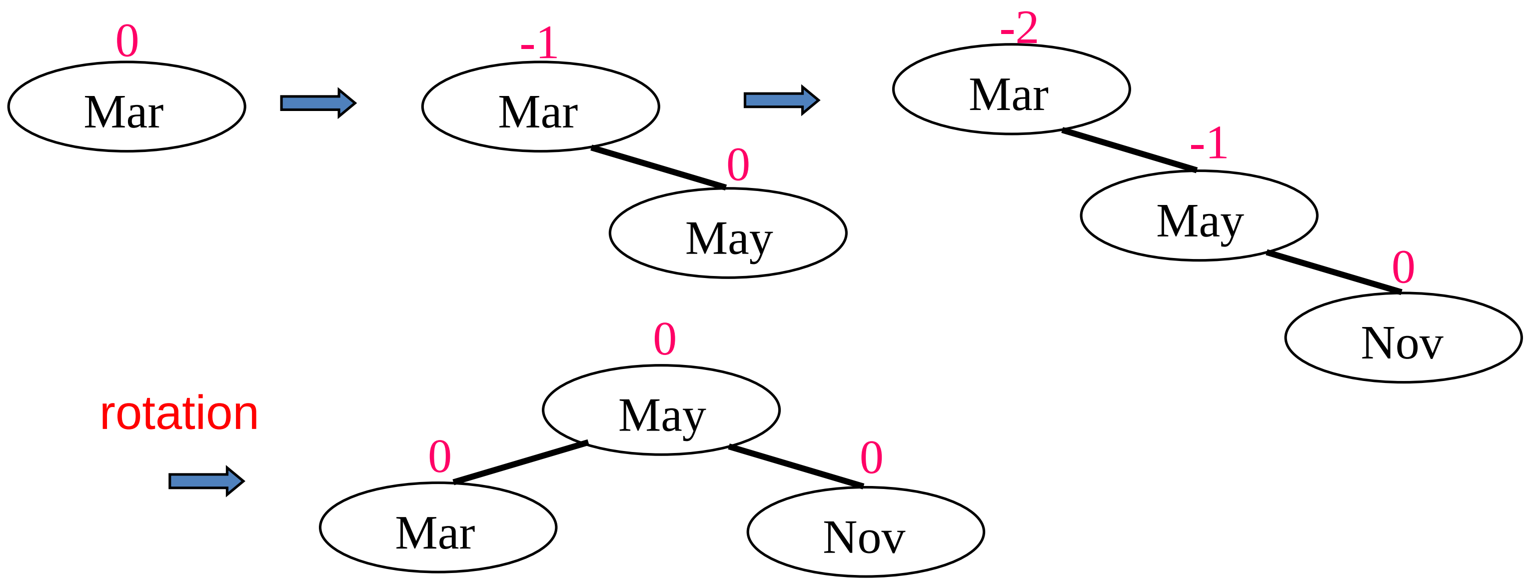
## ■ Inserting a node into an AVL tree

1. Connect the new node to a leaf node with the same algorithm with BST insertion
2. Check the balance of each node
   - If the new node increased the balance
3. If an unbalanced node is found, rebalance it and then continue up the tree
   - Not all insertion breaks the balance



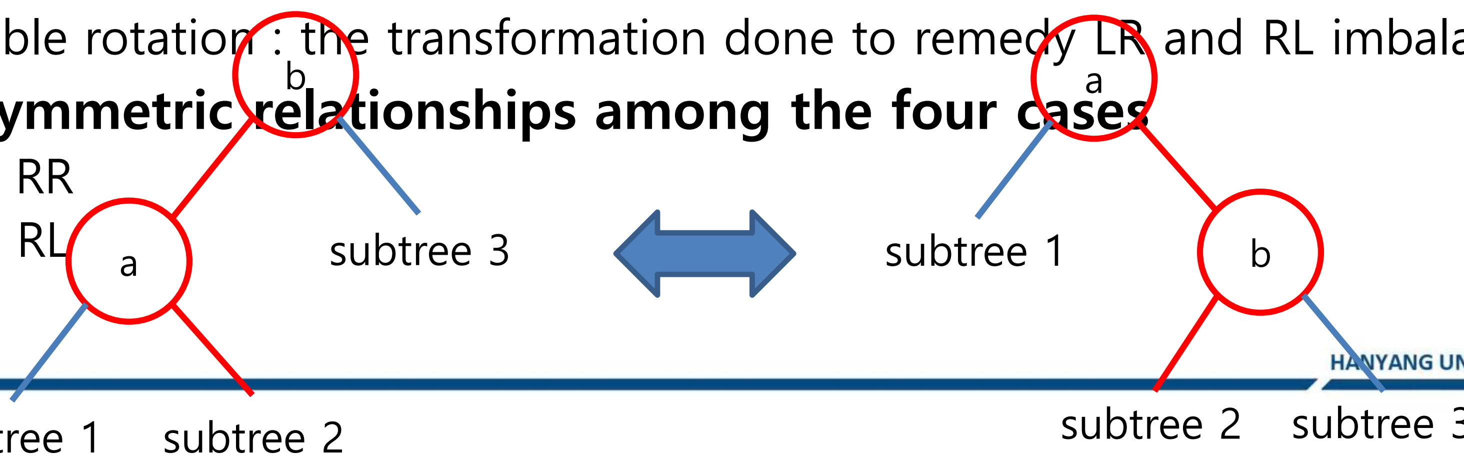
# Rotation

- **Rotation: switching children and parents among two or three adjacent nodes to restore balance of a tree.**
  - A rotation may change the depth of some nodes, but do not change their relative ordering.

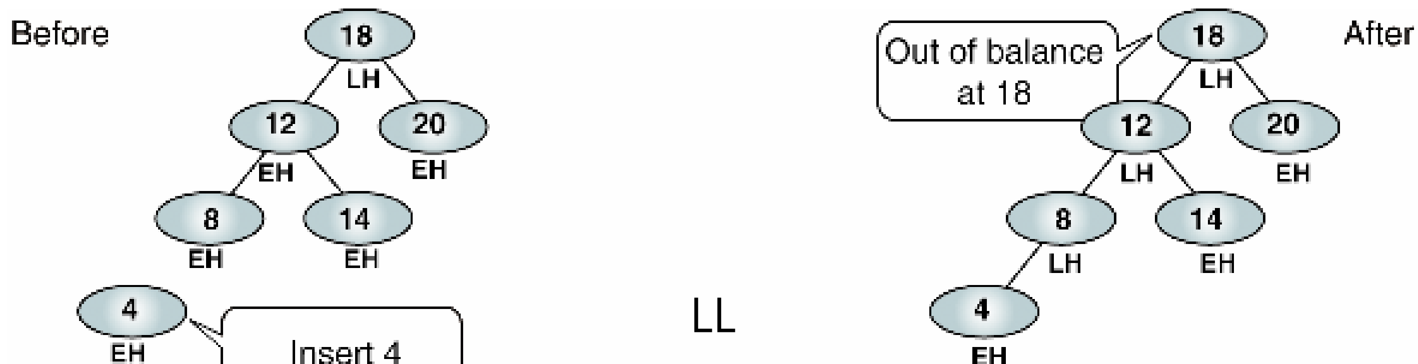


# Rotation

- **The rotations are characterized by the nearest ancestor, 'A' of the inserted node 'Y' whose balance factor becomes ±2.**
  - LL(Left of Left, Left-Left): new node Y is inserted in the left subtree of the left subtree of A
  - LR(Right of Left, Left-Right): Y is inserted in the right subtree of the left subtree of A
  - RR(Right of Right, Right-Right): Y is inserted in the right subtree of the right subtree of A
  - RL(Left of Right, Right-Left): Y is inserted in the left subtree of the right subtree of A
- The rotations
  - Single rotation : the transformation done to remedy LL and RR imbalances
  - Double rotation : the transformation done to remedy LR and RL imbalances
- **Note symmetric relationships among the four cases**
  - LL – RR
  - LR - RL

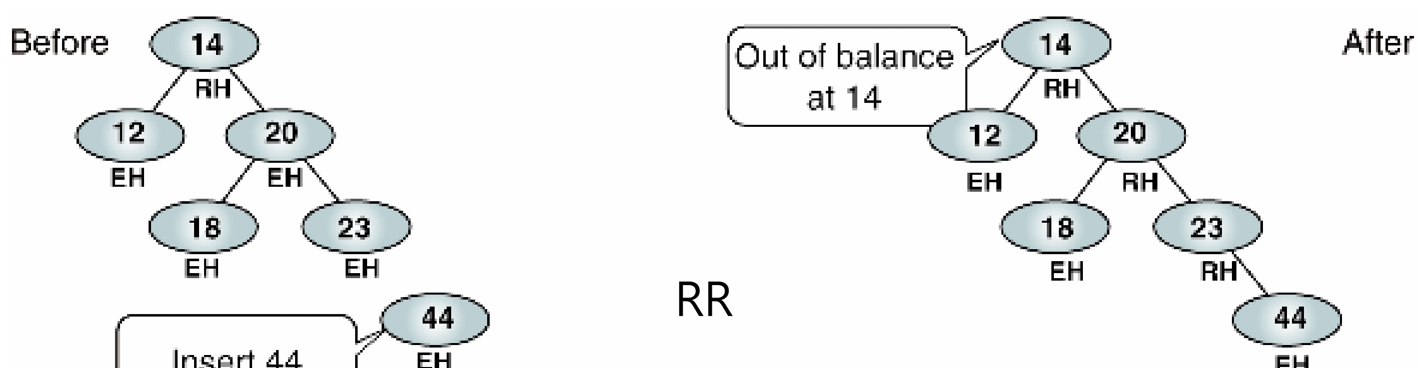


# Insertion into AVL Tree




LL

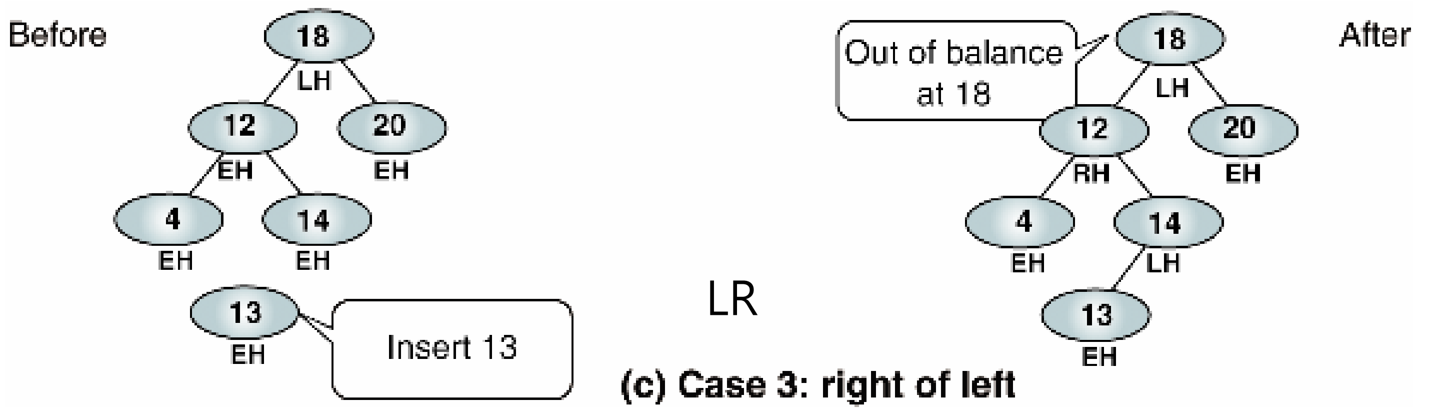
(a) Case 1:



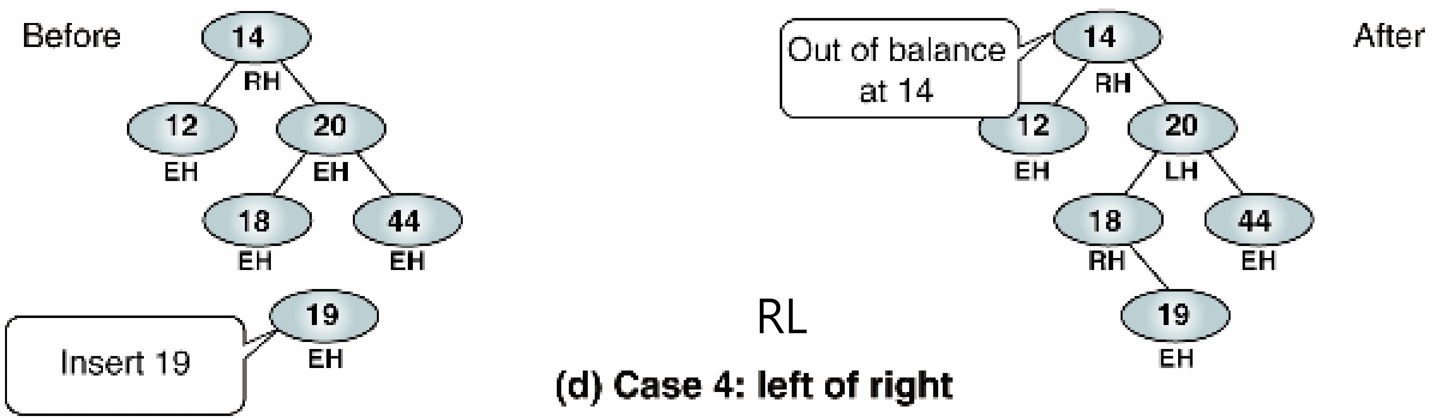

RR

(b) Case 2: right of right

# Insertion into AVL Tree




(c) Case 3: right of left



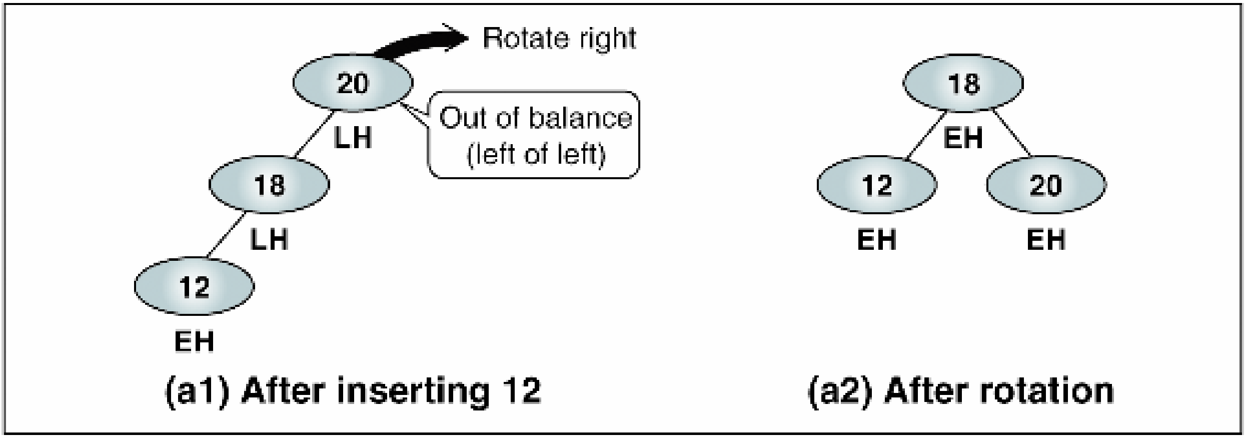

(d) Case 4: left of right

# Left Balancing

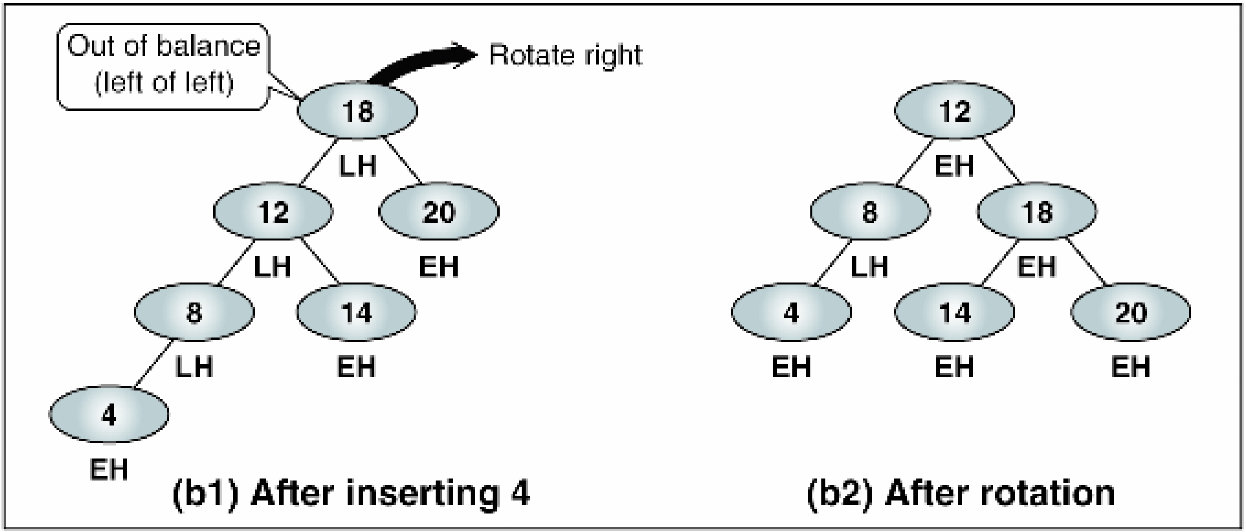
- **To rebalance left high tree, it should be right rotated**

- **Types of right rotations**
  - LL (left of left, Left-Left) rotation: right rotation (r)
    - Simple right rotation
      - Left subtree has no right child
    - Complex right rotation
      - Left subtree has a right child
  - LR (right of left, Left-Right) rotation: left rotation (x) + right rotation (r)
    - Simple double rotation right
    - Complex double rotation right

# Left Balancing: LL Rotation



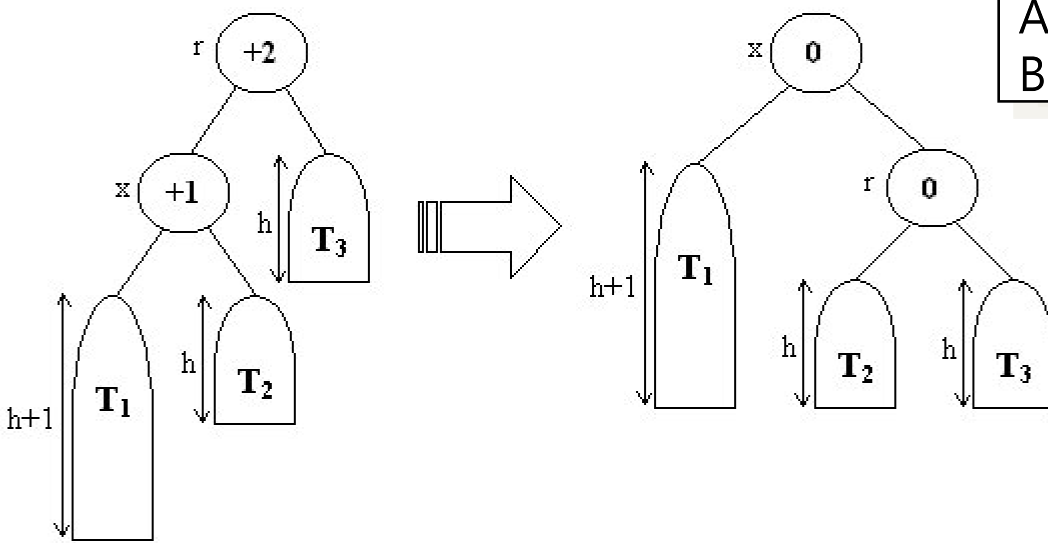

(a) Simple right rotation




(b) Complex right rotation

# Left Balancing: LL Rotation (Complex Right Rotation)

- x becomes new root
- r and r's right subtree become x's right subtree
- right subtree of x becomes r's left subtree



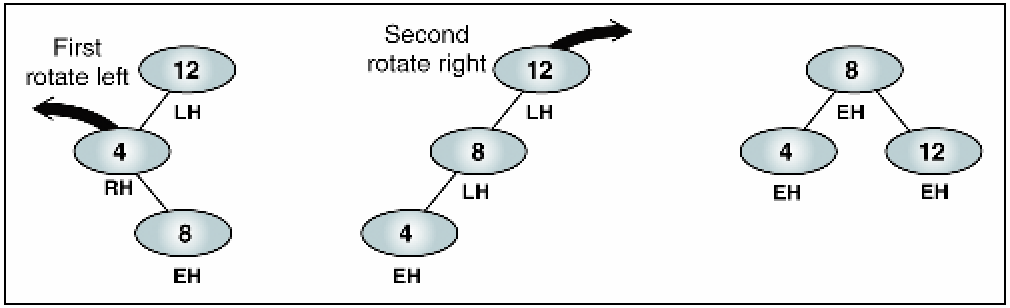

After rotation,
BF(x) = 0, BF(r) = 0

* r: nearest ancestor of the new node whose BF(r) is +2 or -2

# Left Balancing: LR Rotation

- **Rebalancing left: double rotation**
  - Firstly rotate the left subtree (x) to the left
  - Then, rotate the root (r) to the right, making the left node the new root.

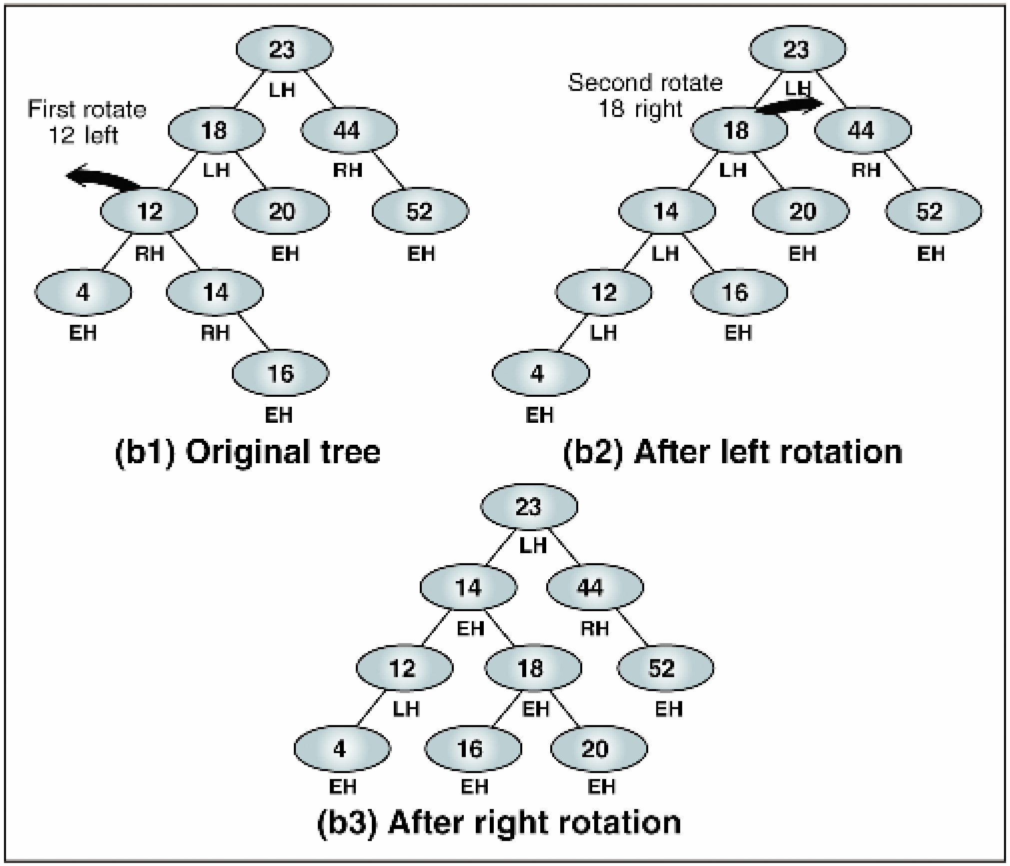
- **Simple double rotation right**




(a) Simple double rotation right

# Left Balancing: LR Rotation

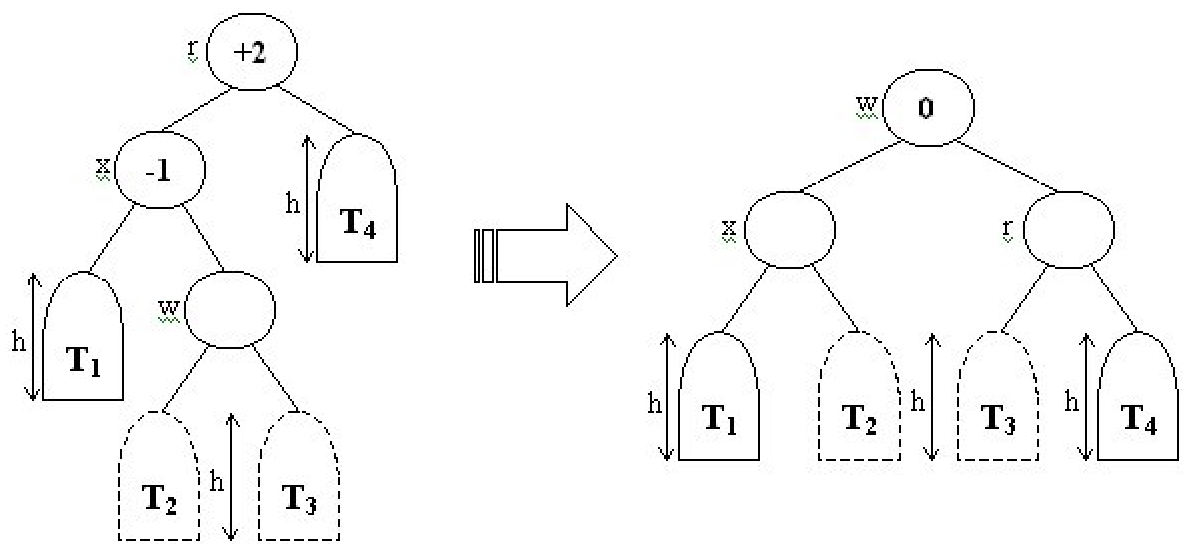
- Complex double rotation right




(b) Complex double rotation right

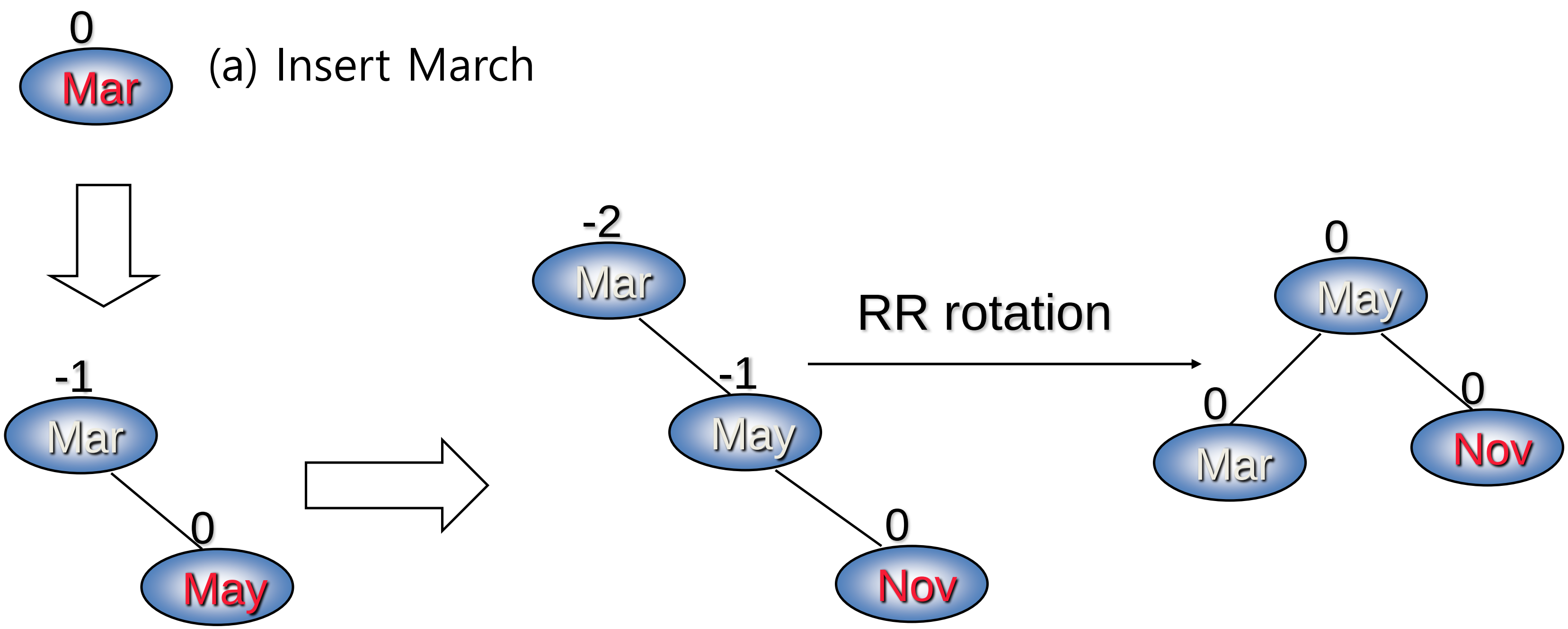
# Left Balancing: LR Rotation (Complex double rotation right)

- w becomes new root
- r and r's right subtree become right subtree of w
- Left subtree of w becomes right subtree of x
- Right subtree of w becomes left subtree of r



# AVL Insertion - Example

- **Insertion: Mar, May, Nov, Aug, Apr, Jan, Dec, July, Feb, June, Oct, Sept**
  - Insertion-> unbalancing -> rebalancing

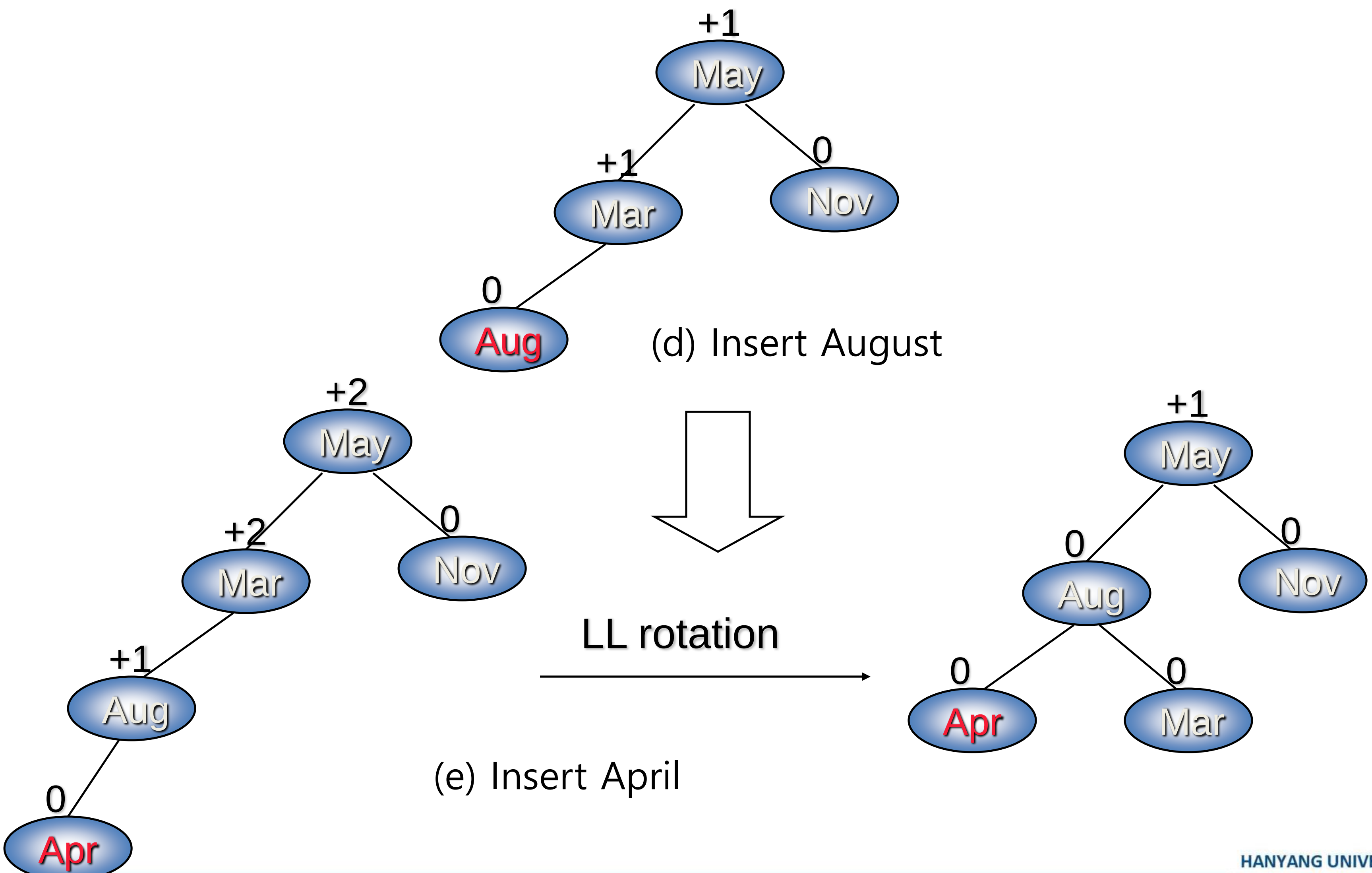



(a) Insert March

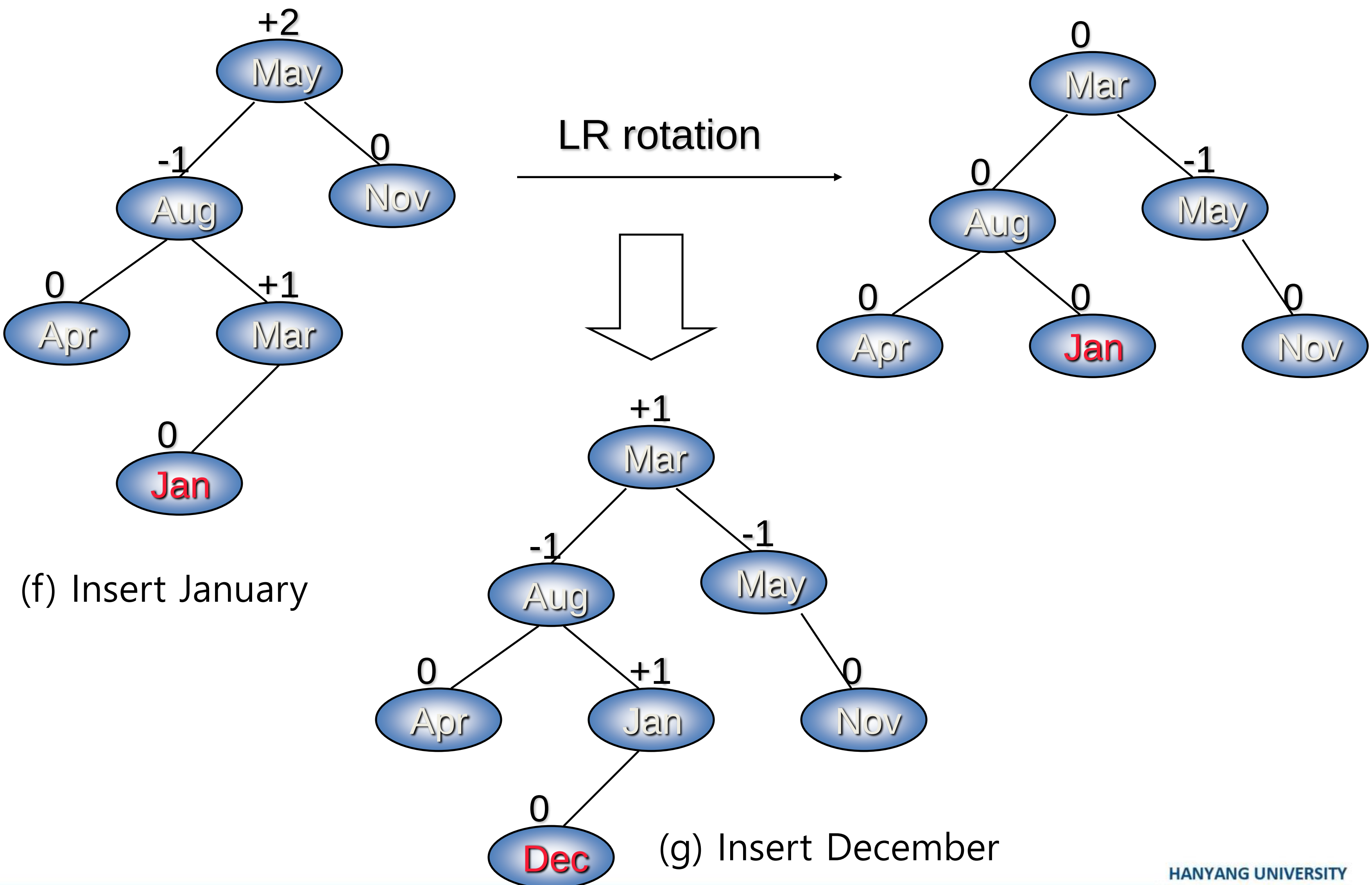
(b) Insert May

(c) Insert Nov

# AVL Insertion - Example



# AVL Insertion - Example

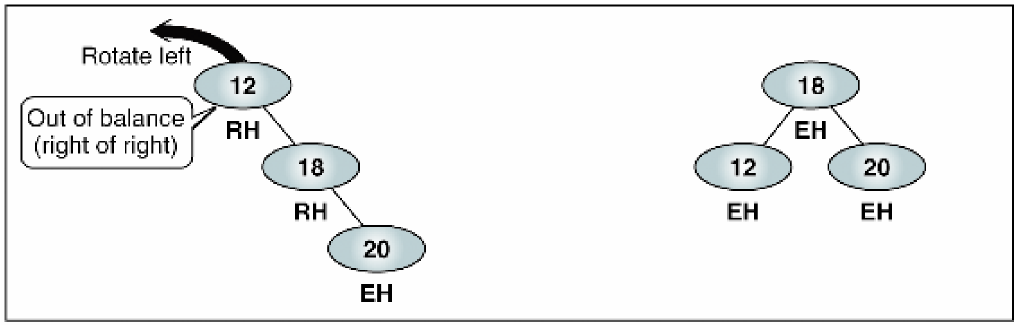



(f) Insert January

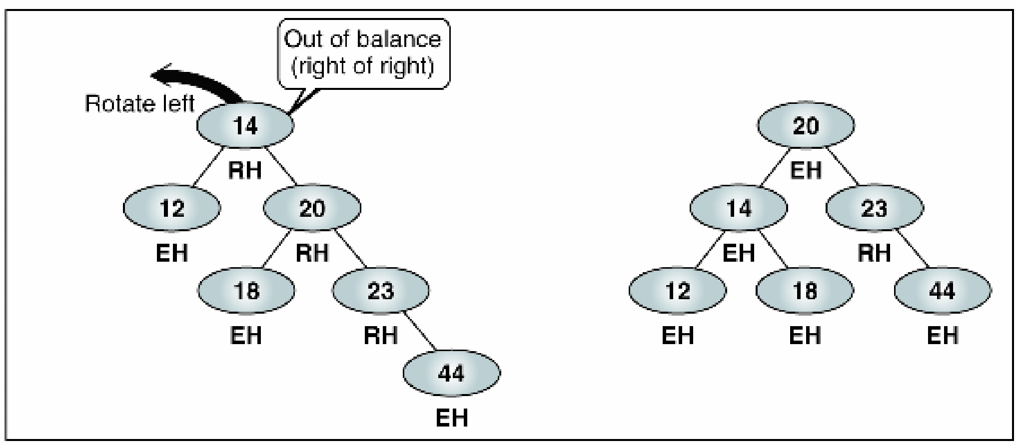
(g) Insert December

# Right Balancing

- Basically, right balancing makes symmetric with left balancing



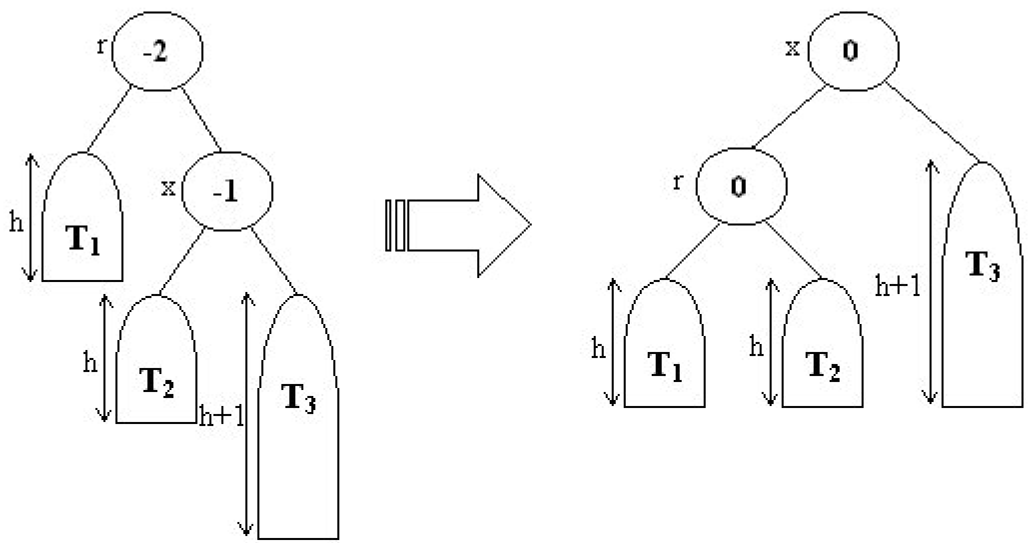

(a) Simple left rotation




(b) Complex left rotation

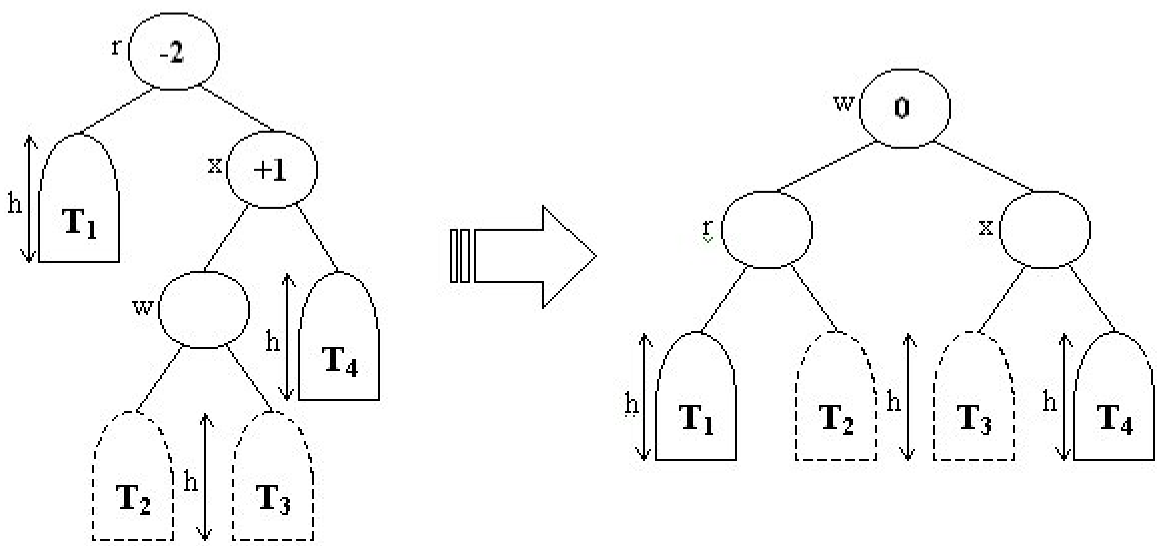
# Right Balancing: RR Rotation

- x becomes new root
- r and r's left subtree become x's left subtree
- Left subtree of x becomes r's right subtree



# Right Balancing: RL Rotation

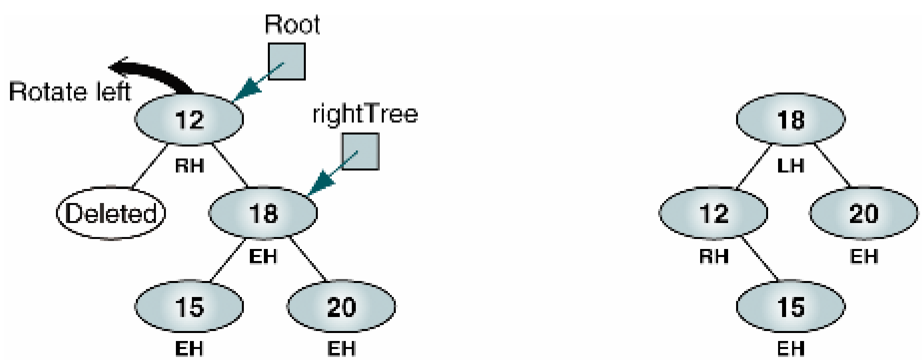
- w becomes new root
- r and r's left subtree becomes w's left subtree
- Right subtree of w becomes left subtree of x
- Left subtree of w becomes right subtree of r



# Deletion from AVL Tree

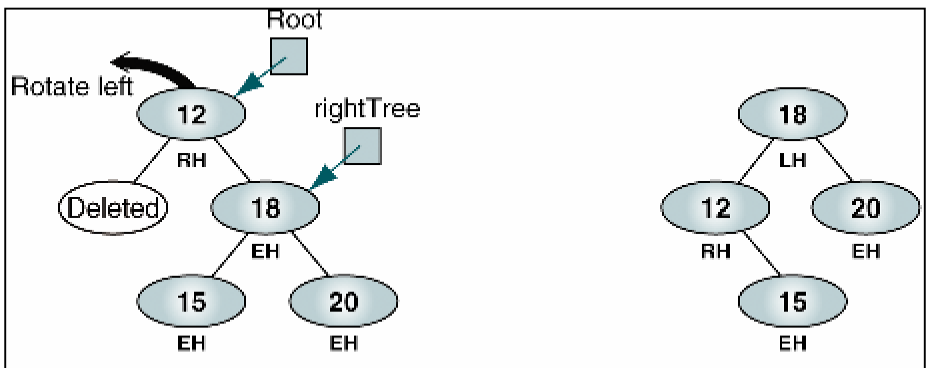
## ■ Deleting a node from AVL tree

1. Delete a node by deletion algorithm of binary search tree
   - Deletion take place at a leaf node.
2. If tree is unbalanced, rebalance it

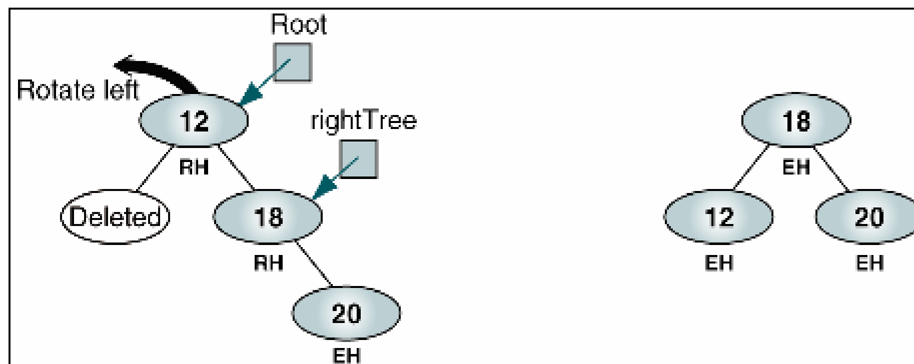


# Rebalancing After Deletion

- Left balancing and right balancing are symmetric
- If unbalancing was caused by deletion from left subtree rebalance should be take place at the right subtree, and vice versa

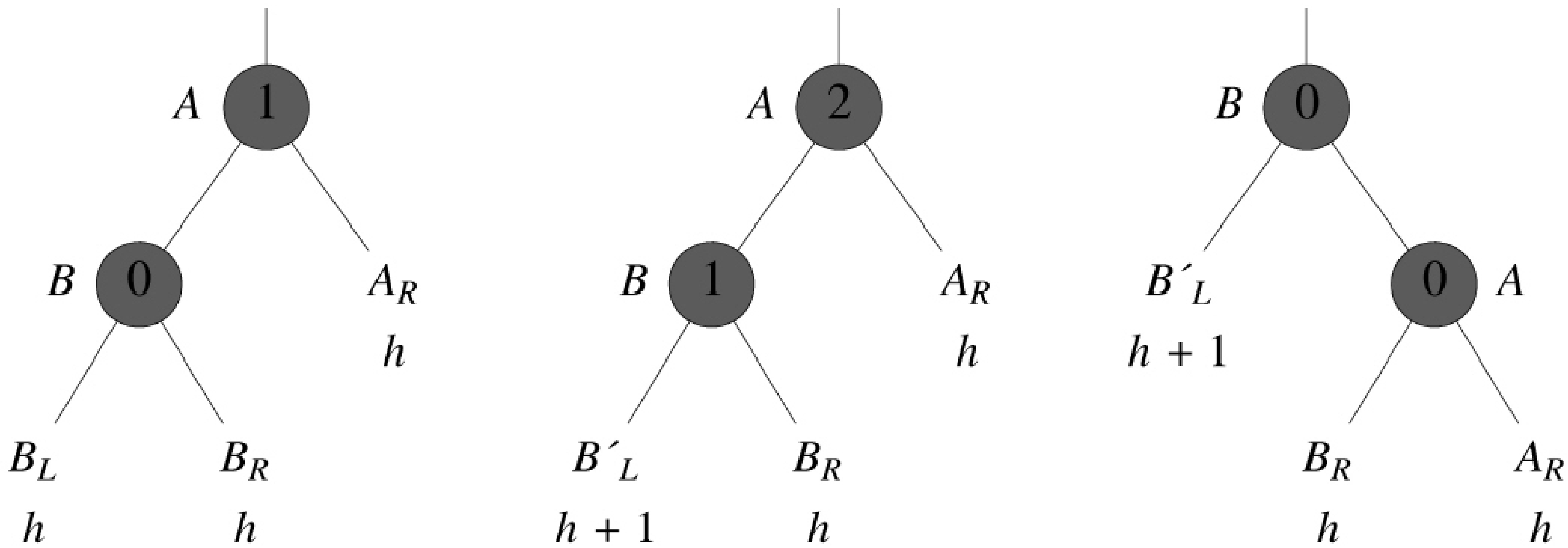


(a) Right subtree is even balanced



(b) Right subtree is not even balanced

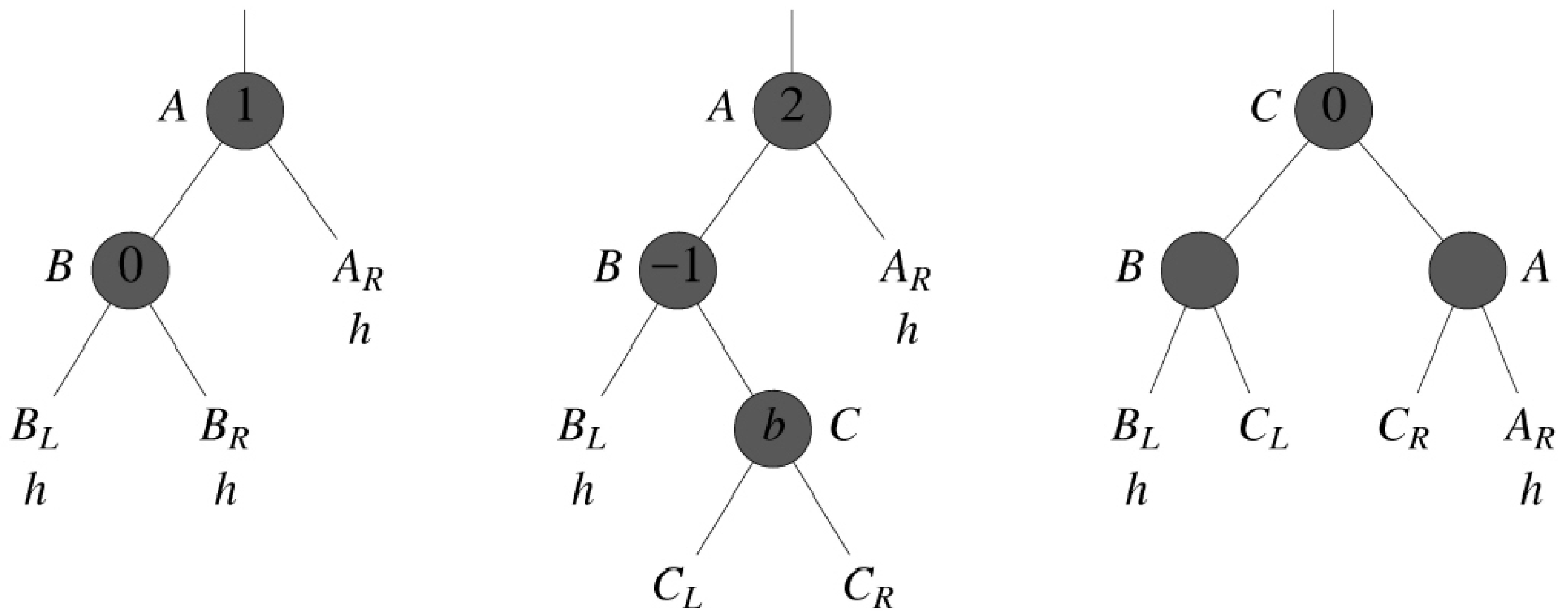
# Insertion into an AVL tree- An LL Rotation




(a) Before insertion (b) After inserting into $B_L$ (c) After LL rotation

Balance factors are inside nodes
Subtree heights are below subtree name

# Insertion into an AVL tree- An LR Rotation




(a) Before insertion (b) After inserting into $B_R$ (c) After LR rotation

b = 0 ⇒ bf(B) = bf(A) = 0 after rotation
b = 1 ⇒ bf(B) = 0 and bf(A) = -1 after rotation
b = -1 ⇒ bf(B) = 1 and bf(A) = 0 after rotation

# Insertion into an AVL tree

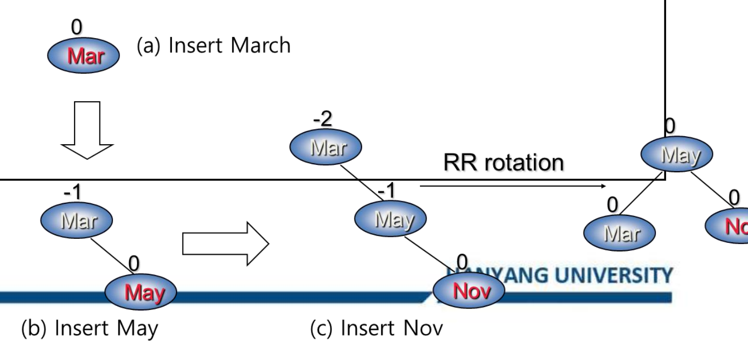
```
typedef struct {
            int key;
            } element;

typedef struct treeNode *treePointer;
           struct treeNode {
           treePointer leftChild;
           element     data;
           short int    bf;
           treePointer rightChild;
           };
```

# Insertion into an AVL tree

```
void avlInsert(treePointer *parent, element x, int *unbalanced)
{
        if(!*parent) { /*insert element into null tree */
                *unbalanced = TRUE;
                MALLOC(*parent, sizeof(treeNode));
                (*parent)->leftChild = (*parent)->rightChild = NULL;
                (*parent)->bf = 0; (*parent)->data = x;
        }
        else if(x.key < (*parent)->data.key) {
                avlInsert(&(*parent)->leftChild, x, unbalanced);
                if(*unbalanced)
                /* left branch has grown higher */
                        switch((*parent)->bf) {
                                case -1: (*parent)->bf = 0;
                                         *unbalanced = FALSE; break;
                                case 0: (*parent)->bf = 1; break;
                                case 1: leftRotation(parent, unbalanced);
                        }
        }
```

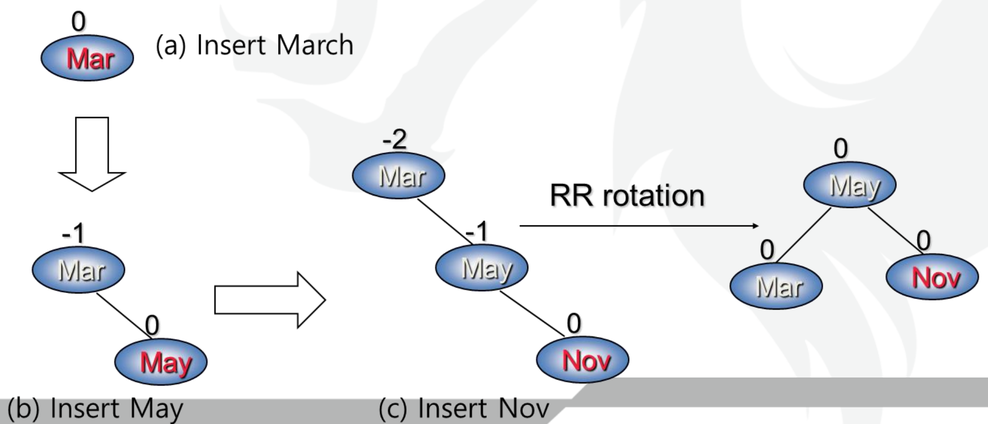
# Insertion into an AVL tree

```
        else if(x.key > (*parent)->data.key) {
                avlInsert(*(*parent)->rightChild, x, unbalanced);
                if(*unbalanced)
                  /*right branch has grown higher */
                        switch((*parent)->bf) {
                                case 1: (*parent)->bf = 0;
                                        *unbalanced = FALSE; break;
                                case 0: (*parent)->bf = -1; break;
                                case -1: rightRotation(parent, unbalanced);
                        }
        }
        else {
                *unbalanced = FALSE;
                printf("The key is already in the tree");
        }
}
```



# AVL트리

```
void rightRotation(treePointer *parent, int *unbalanced) {
        treePointer grandChild, child;
        child = (*parent)->rightChild;
        if (child->bf == -1) {
                /* RR Rotation */
                (*parent)->rightChild = child->leftChild;
                child->leftChild = *parent;
                (*parent)->bf = 0;
                (*parent) = child;
        }
        else {
                /* RL Rotataion */
                grandChild = child->leftChild;
                child->leftChild = grandChild->rightChild;
                grandChild->rightChild = child;
                (*parent)->rightChild = grandChild->leftChild;
                grandChild->leftChild = *parent;
```

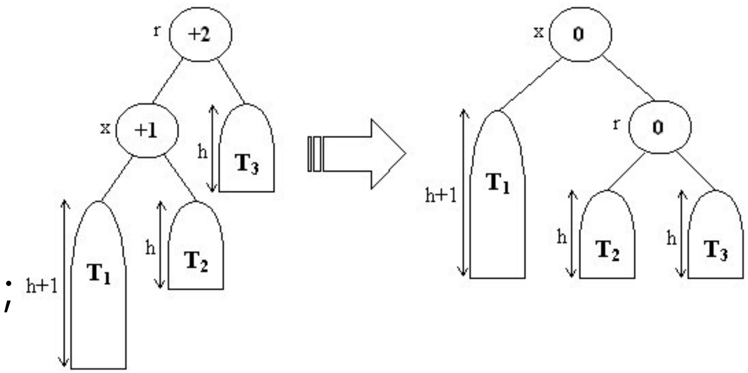


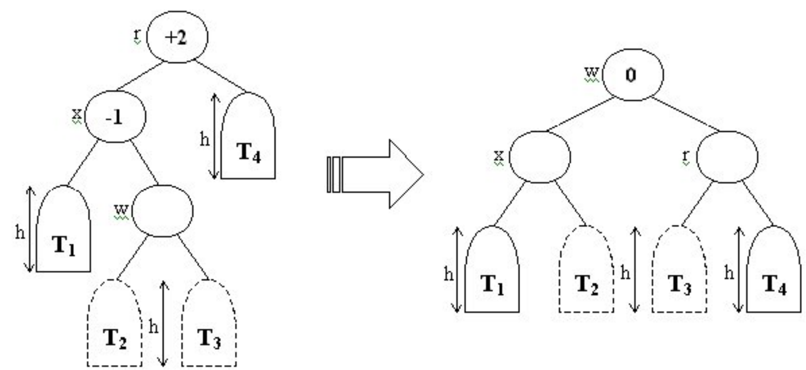
# AVL트리

```
                switch (grandChild->bf) {
                case -1:
                        (*parent)->bf = 1;
                        child->bf = 0;
                        break;
                case 0:
                        (*parent)->bf = child->bf = 0;
                        break;
                case 1:
                        (*parent)->bf = 0;
                        child->bf = -1;
                        break;
                }
                *parent = grandChild;
        }
        (*parent)->bf = 0;
        *unbalanced = FALSE;
}
```

# Left Rotation Function

```
void leftRotation(treePointer *parent, int *unbalanced)
{
        treePointer grandChild, child;
        child = (*parent)->leftChild;  x
        if(child->bf == 1) {
                /* LL rotation */                T2
r의 왼쪽        (*parent)->leftChild = child->rightChild;
x의 오른쪽      child->rightChild = *parent;  r
                (*parent)->bf = 0;
                (*parent) = child;  x가 루트가 됨
        }
        else {
                /*LR rotation */
                grandChild = child->rightChild;  w
x의 오른쪽      child->rightChild = grandChild->leftChild;  T2
w의 왼쪽        grandChild->leftChild = child;  x
r의 왼쪽        (*parent)->leftChild = grandChild->rightChild;  T3
```






```
                grandChild->rightChild= *parent;
```

# Left Rotation Function

```
            switch(grandChild->bf) {
                    case 1: (*parent)->bf = -1;
                             child->bf = 0;  break;
                    case 0: (*parent)->bf = child->bf = 0;
                             break;
                    case -1: (*parent)->bf = 0;
                              child->bf = 1;
            }
            *parent = grandChild;
      }
      (*parent)->bf = 0;
      *unbalanced = FALSE;
}
```

# Comparison of various structures

| Operation | Sequential list | Linked list | AVL tree |
|---|---|---|---|
| Search for element with key k | $O(\log n)$ | $O(n)$ | $O(\log n)$ |
| Search for jth item | $O(1)$ | $O(j)$ | $O(\log n)$ |
| Delete element with key k | $O(n)$ | $O(1)^1$ | $O(\log n)$ |
| Delete jth element | $O(n - j)$ | $O(j)$ | $O(\log n)$ |
| Insert | $O(n)$ | $O(1)^2$ | $O(\log n)$ |
| Output in order | $O(n)$ | $O(n)$ | $O(n)$ |

1. Doubly linked list and position of K known
2. Position for insertion known