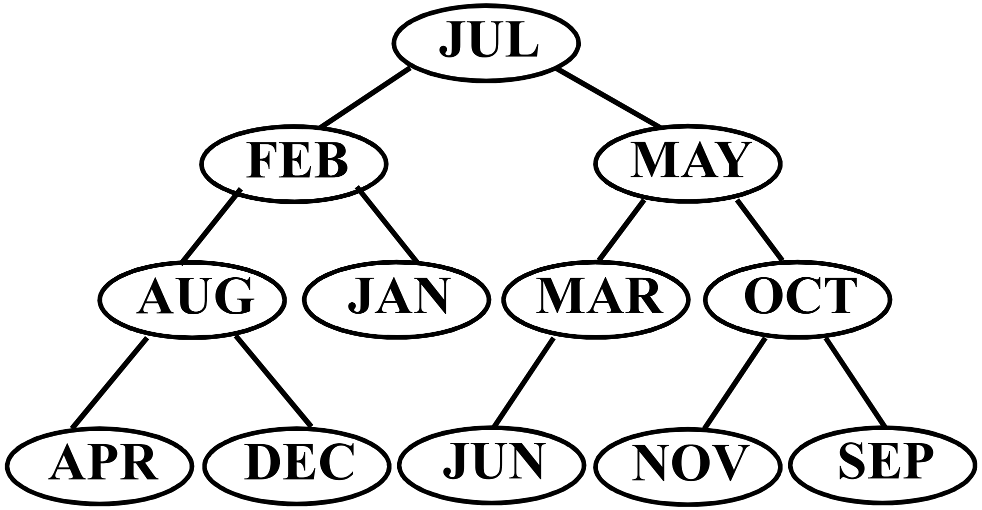
AVL Trees

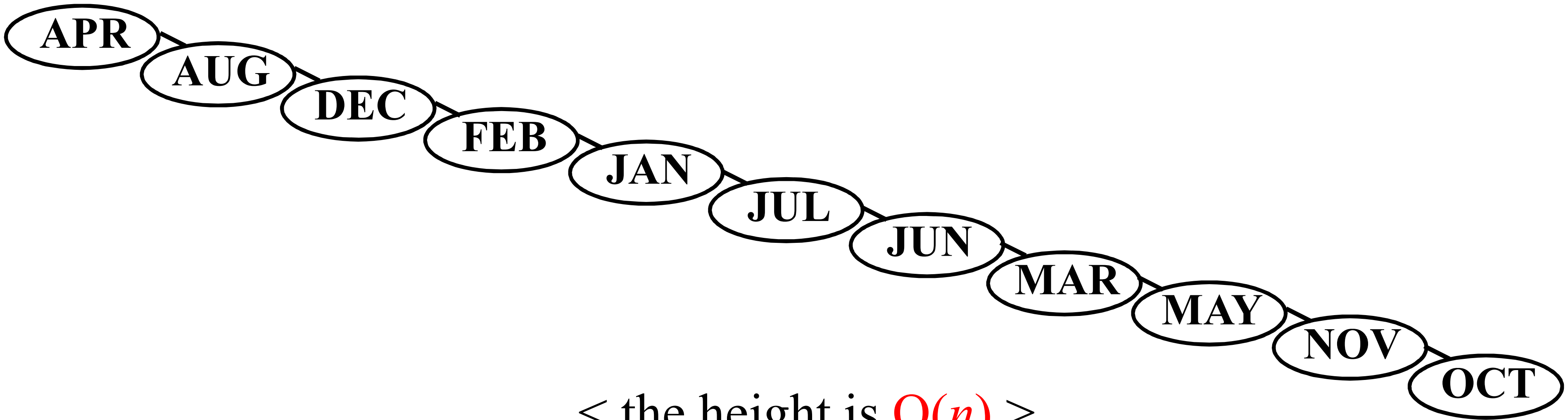
Prof. Ki-Hoon Lee
Dept. of Computer Engineering
Kwangwoon University

Motivation

- We have seen that the efficiency of many important operations on trees is related to the height of the tree
  - For example, searching, inserting, and deleting in a BST are all $O(height)$
- Number of nodes n & height h
  - $\log_2(n+1) \le h \le n$
- For efficiency's sake, we would like to guarantee that $h = O(\log n)$



< the height is O(log n) >



< the height is O(n) >

AVL Trees

- A height-balanced binary search tree invented by Adelson-Velskii and Landis
- The height of an AVL tree that has n nodes is at most $1.44 \log_2 (n+2)$
- The height of every n node binary tree is at least $\log_2 (n+1)$

$$\log_2 (n+1) \leq \text{height} \leq 1.44 \log_2 (n+2)$$

Definition of an AVL 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, then T is *height balanced* iff
  - (i) T_L and T_R are height balanced and
  - (ii) $|h_L - h_R| \leq 1$

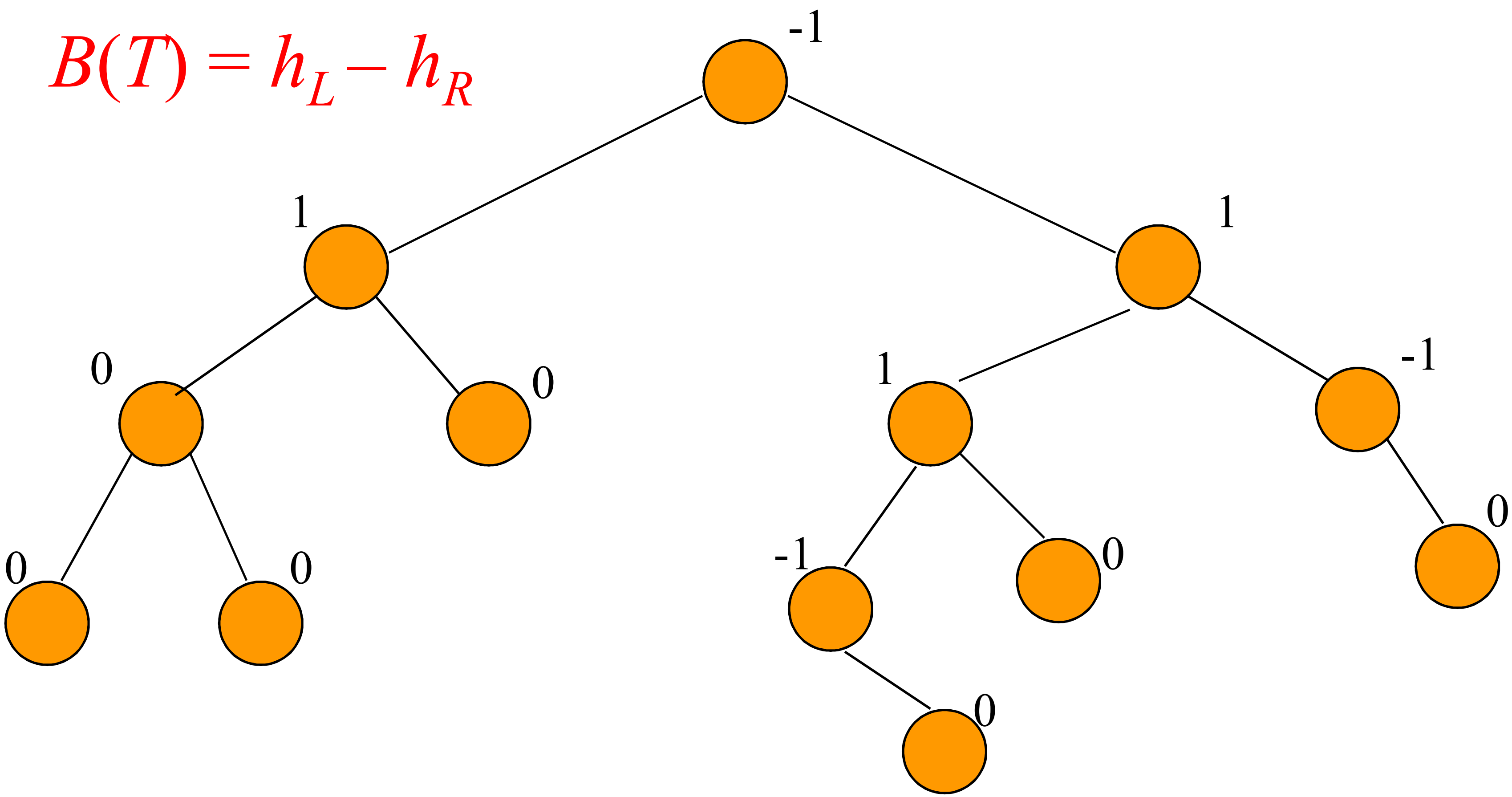
  where h_L and h_R are the heights of T_L and T_R respectively.

Balance Factor

- The *balance factor*, $BF(T)$, of a node T in a binary tree is defined to be $h_L - h_R$ where h_L and h_R are the heights of the left and right subtrees of T
- For any node T in a height-balanced tree, $BF(T) = -1, 0, \text{or } 1$

Balance Factors

$$B(T) = h_L - h_R$$



Insert

- Following insert, retrace path towards root and adjust balance factors as needed
- Stop when you reach a node whose balance factor becomes 0, 2, or –2, or when you reach the root
- The new tree is not an AVL tree only if you reach a node whose balance factor is either 2 or –2
- In this case, we say the tree has become unbalanced

A-Node

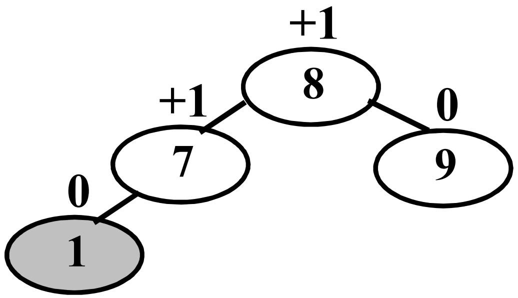
- Let *A* be the nearest ancestor of the newly inserted node whose balance factor becomes $+2$ or -2
- Before the insertion, the balance factors of all nodes on the path from *A* to the new insertion point must have been 0
  - Why?

Imbalance Types

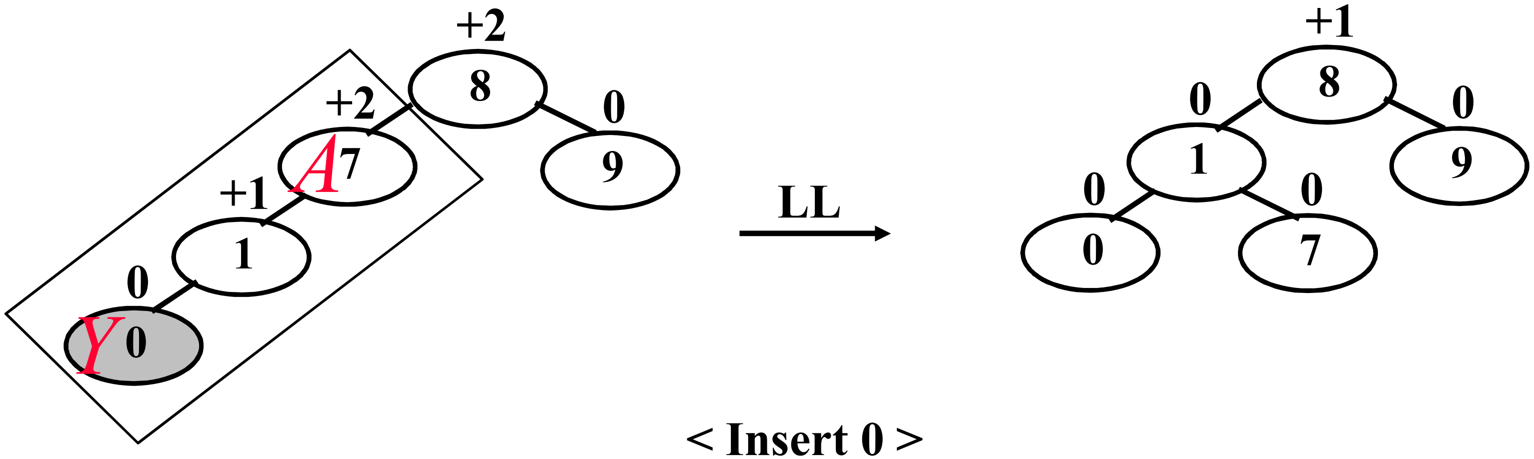
- RR: newly inserted node *Y* is in the right subtree of the right subtree of *A*
- LL: … left subtree of left subtree of *A*
- RL: … left subtree of right subtree of *A*
- LR: … right subtree of left subtree of *A*

- *LL and RR are symmetric, as are LR and RL*

LL Rotation

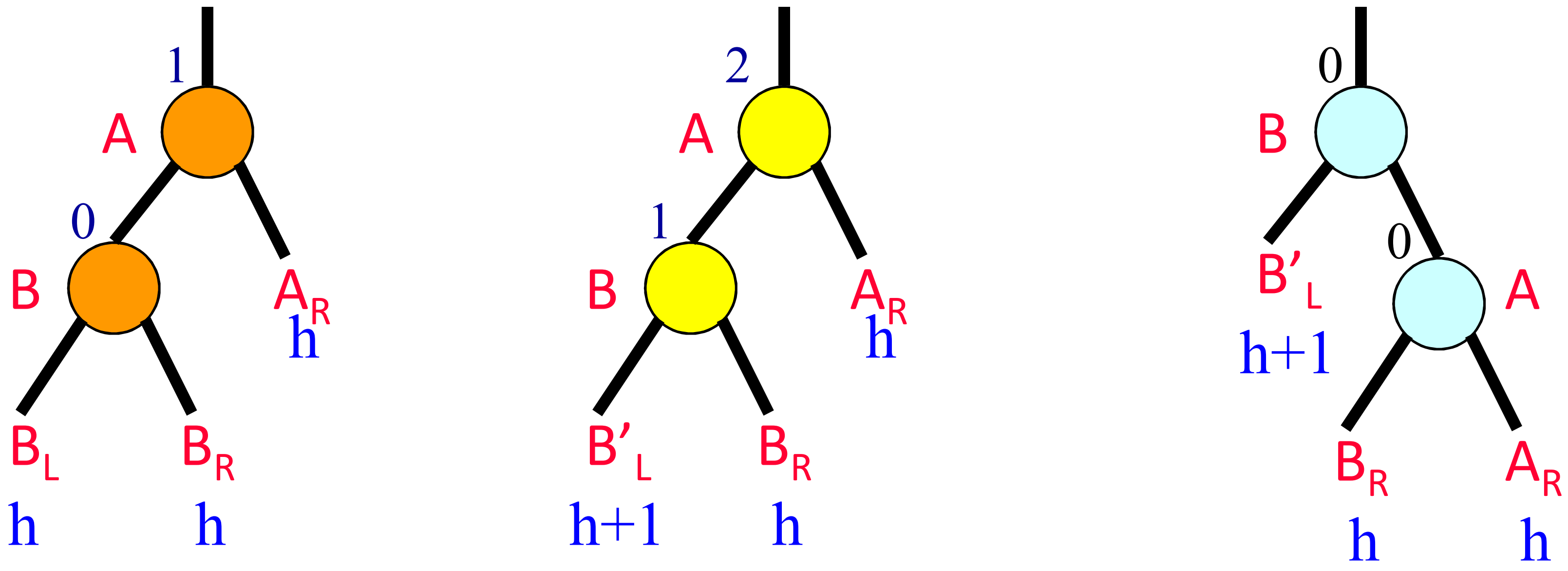


< Before >



< Insert 0 >

LL Rotation (cont.)

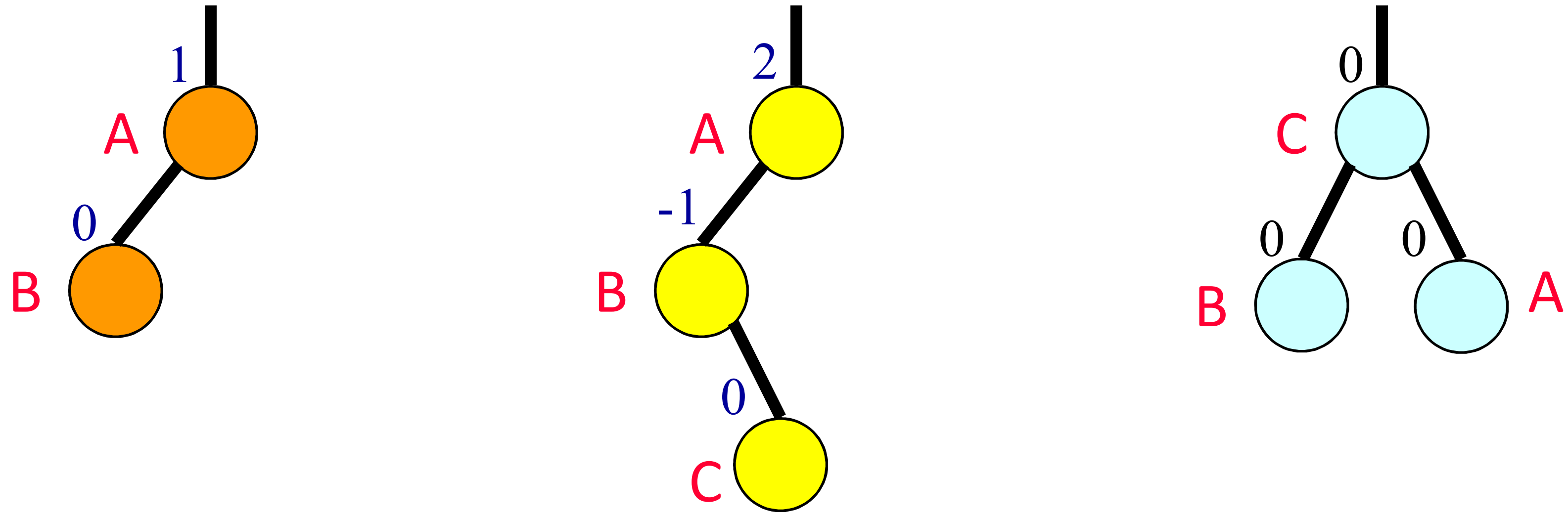


Before insertion. After insertion. After rotation.

- Subtree height is unchanged.
- No further adjustments to be done.

LR Rotation (case 1)

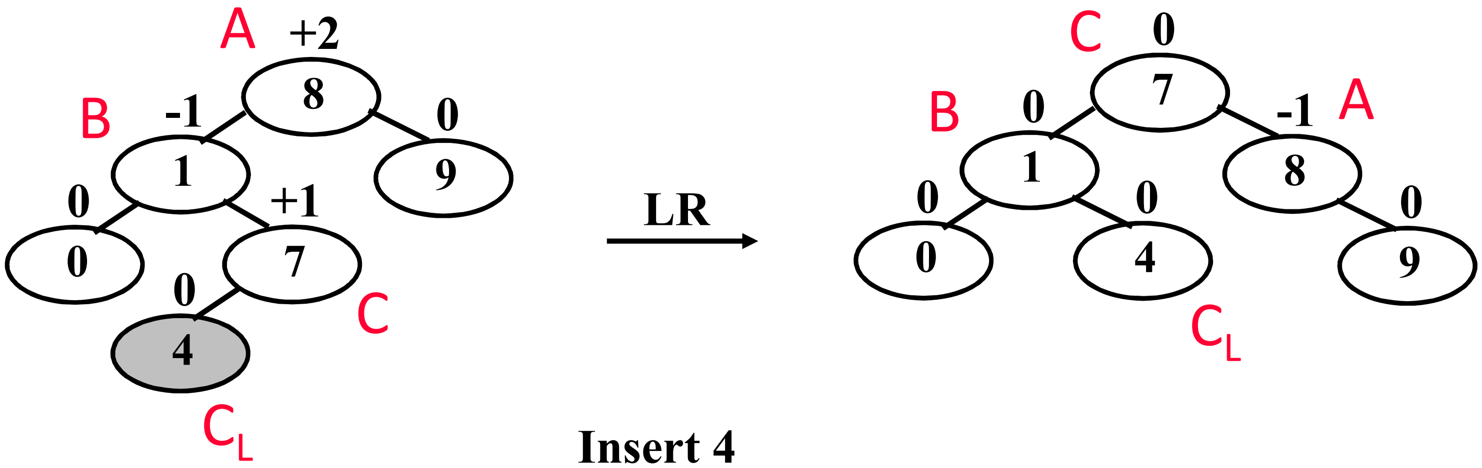
- *B* is a leaf prior to the insert



Before insertion. After insertion. After rotation.

LR Rotation (case 2)

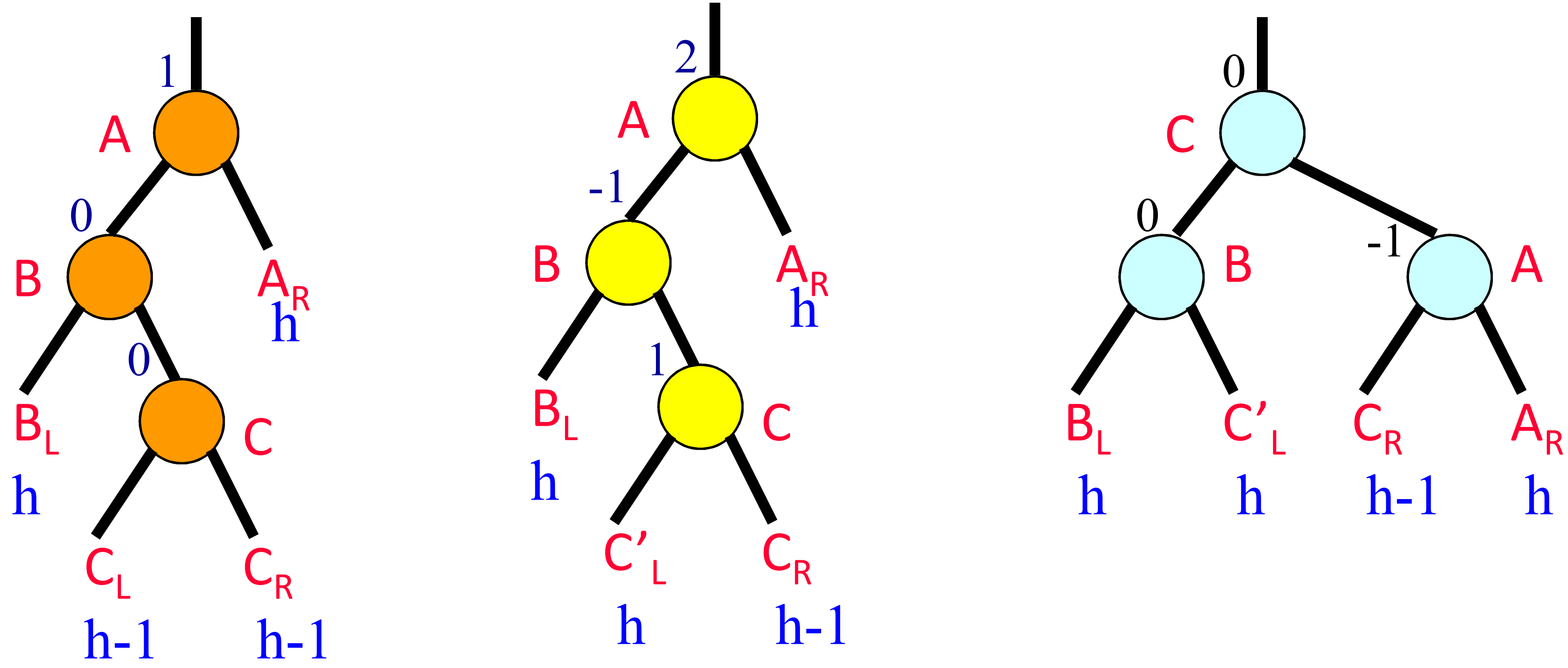
- *B* is not a leaf prior to the insert, and the insert takes place in the left subtree of *C*



Insert 4

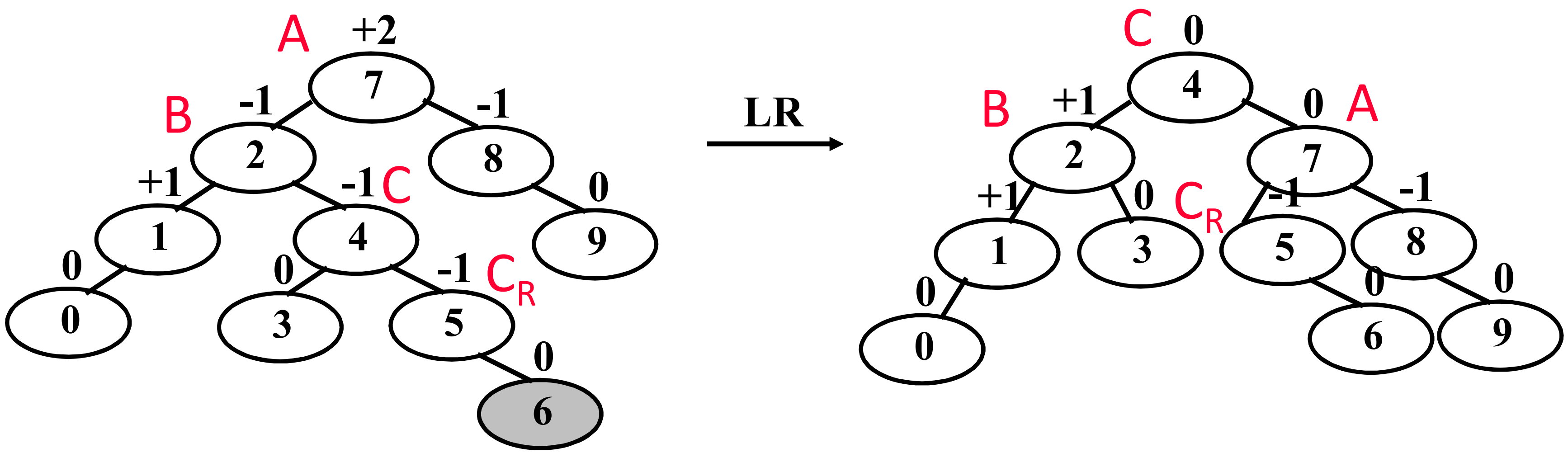
LR Rotation (case 2)

- *B* is not a leaf prior to the insert, and the insert takes place in the left subtree of *C*



LR Rotation (case 3)

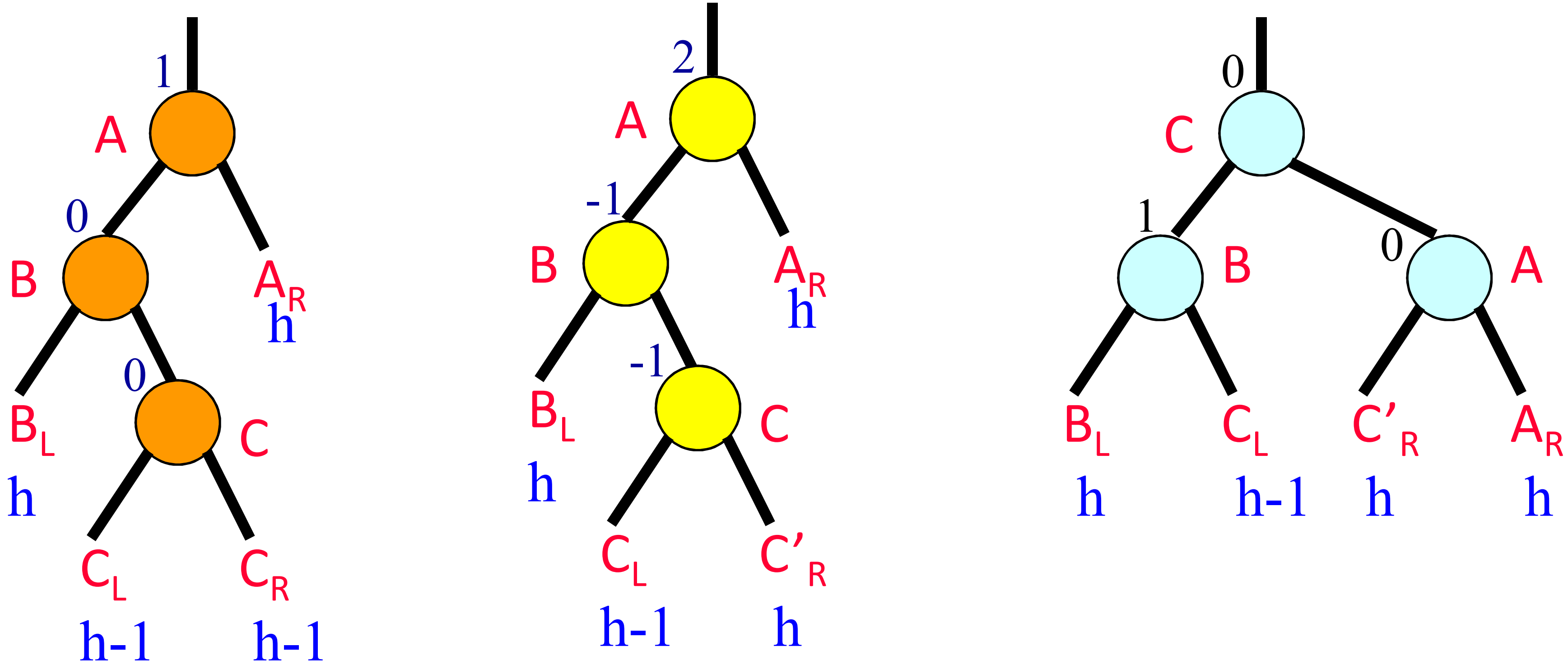
- *B* is not a leaf prior to the insert, and the insert takes place in the right subtree of *C*



Insert 6

LR Rotation (case 3)

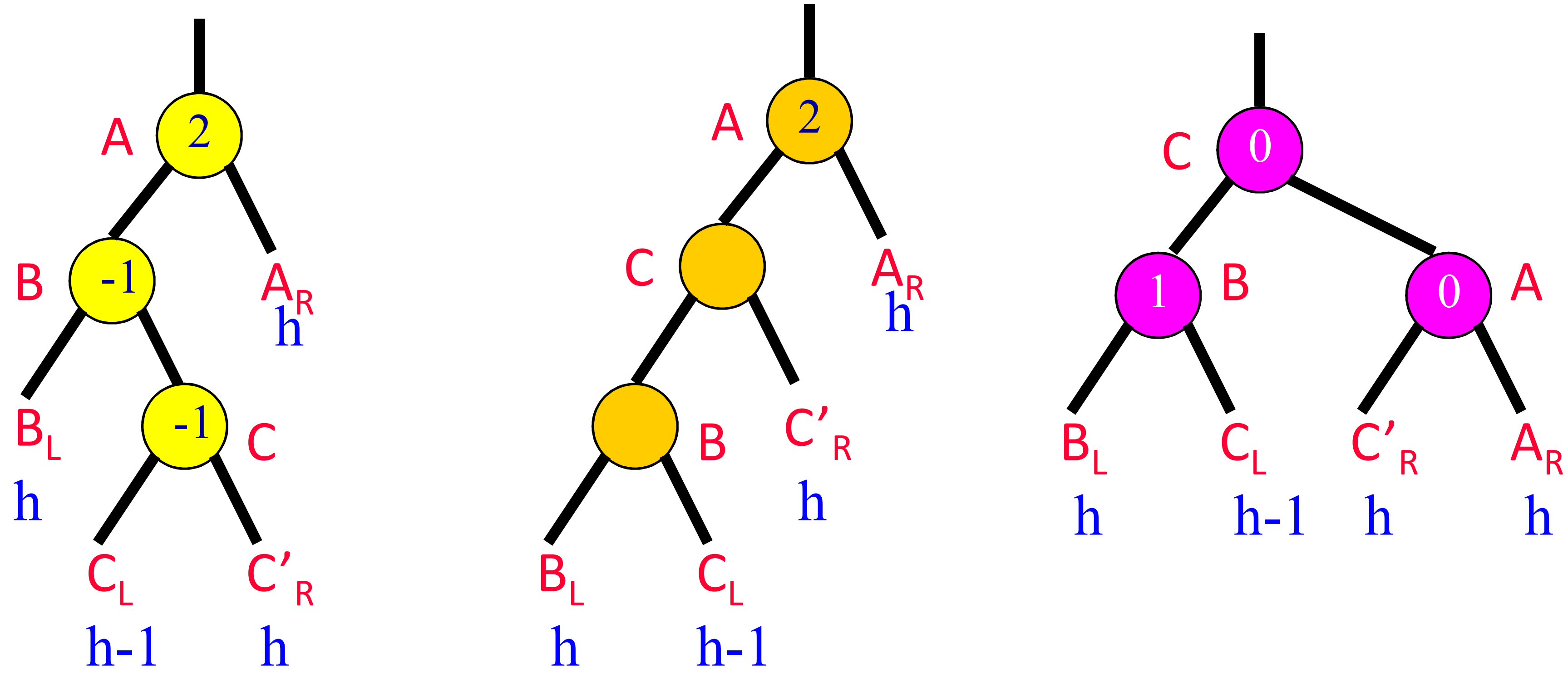
- *B* is not a leaf prior to the insert, and the insert takes place in the right subtree of *C*



Single & Double Rotations

- Single
  - LL and RR
- Double
  - LR and RL
  - LR is RR followed by LL
  - RL is LL followed by RR

LR Is RR + LL



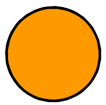
After insertion. After RR rotation. After LL rotation.

Rotation Frequency

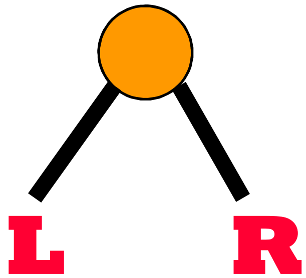
- Insert random numbers.
  - No rotation … 53.4% (approx).
  - LL/RR … 23.3% (approx).
  - LR/RL … 23.2% (approx).

Proof of Upper Bound on Height

- Let N_h = min # of nodes in an AVL tree whose height is h
- $N_0 = 0$
- $N_1 = 1$



N_h, $h > 1$



- Both **L** and **R** are AVL trees.
- Worst case
  - The height of one is h-1.
  - The height of the other is h-2.
- The subtree whose height is h-1 has N_{h-1} nodes.
- The subtree whose height is h-2 has N_{h-2} nodes.
- So, $N_h = N_{h-1} + N_{h-2} + 1$.

Fibonacci Number Theory

- $F_0 = 0$, $F_1 = 1$.
- $F_i = F_{i-1} + F_{i-2}$, $i > 1$.
- $N_0 = 0$, $N_1 = 1$.
- $N_h = N_{h-1} + N_{h-2} + 1$, $i > 1$.
- $N_h = F_{h+2} - 1$ (proof by induction)
- $F_{h+2} \approx \phi^{h+2}/\sqrt{5}$ where $\phi = (1 + \sqrt{5})/2$
- $h \approx \log_\phi(\sqrt{5}(N_h + 1)) - 2 \approx 1.44 \log(N_h + 1) - 0.33$
- insertion time $= O(\log N_h)$

Class Definition of AVL Node

```
template <class K, class E> class AVL;

template <class K, class E>
class AvlNode {
    friend class AVL<K, E>;
public:
    AvlNode(const K& k, const E& e)
    {
        key = k; element = e; bf = 0; leftChild = rightChild = NULL;
    }
private:
    K key;
    E element;
    int bf;
    AvlNode<K, E> *leftChild, *rightChild;
};
```

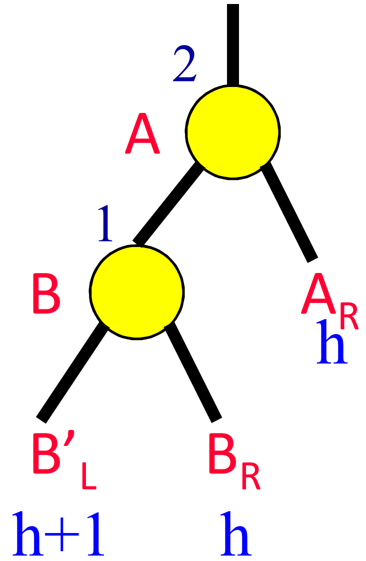
Class Definition of AVL Tree

```
template <class K, class E>
class AVL {
public:
    AVL() : root(NULL) {};
    void Insert(const K&, const E&);
    // ...

private:
    AvlNode<K, E> *root;
};
```

Insert

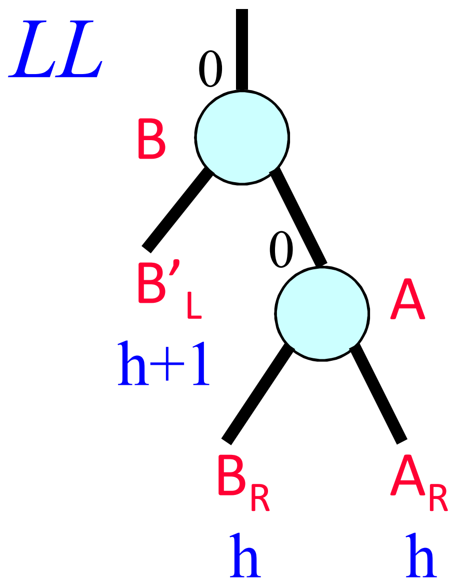
```
template <class K, class E>
void AVL<K, E>::Insert(const K& k, const E& e)
{
    if (root == NULL) { // special case: empty tree
        root = new AvlNode<K, E>(k, e);
        return;
    }

    // Phase 1: Locate insertion point for e
    AvlNode<K, E>   *a = root,   // most recent node with bf = +-1
                    *pa = NULL, // parent of a
                    *p = root,   // p moves through the tree
                    *pp = NULL, // parent of p
                    *rootSub = NULL;

    while (p != NULL) {
        if (p->bf != 0) { a = p; pa = pp; }
        if (k < p->key) { pp = p; p = p->leftChild; } // take left branch
        else if (k > p->key) { pp = p; p = p->rightChild; } // take right branch
        else { p->element = e; return; } // k is already in tree. update e
    } // end of while
```

```
// Phase 2: Insert and rebalance. k is not in the tree and
// may be inserted as the appropriate child of pp.
AvlNode<K, E> *y = new AvlNode<K, E>(k, e);
if (k < pp->key) pp->leftChild = y; // insert as left child
else pp->rightChild = y; // insert as right child

// Adjust balance factors of nodes on path from a to pp. By the definition
// of a, all nodes on this path presently have a balance factor of 0. Their new
// balance factor will be +-1. d=+1 implies that k is inserted in the left subtree
// of a. d=-1 implies that k is inserted in the right subtree of a.
int d;
AvlNode<K, E>   *b, // child of a
                *c; // child of b

if (k > a->key) { b = p = a->rightChild; d = -1; }
else { b = p = a->leftChild; d = 1; }

while (p != y) {
    if (k > p->key) { // height of right increases by 1
        p->bf = -1; p = p->rightChild;
    }
    else { // height of left increases by 1
        p->bf = 1; p = p->leftChild;
    }
}
```

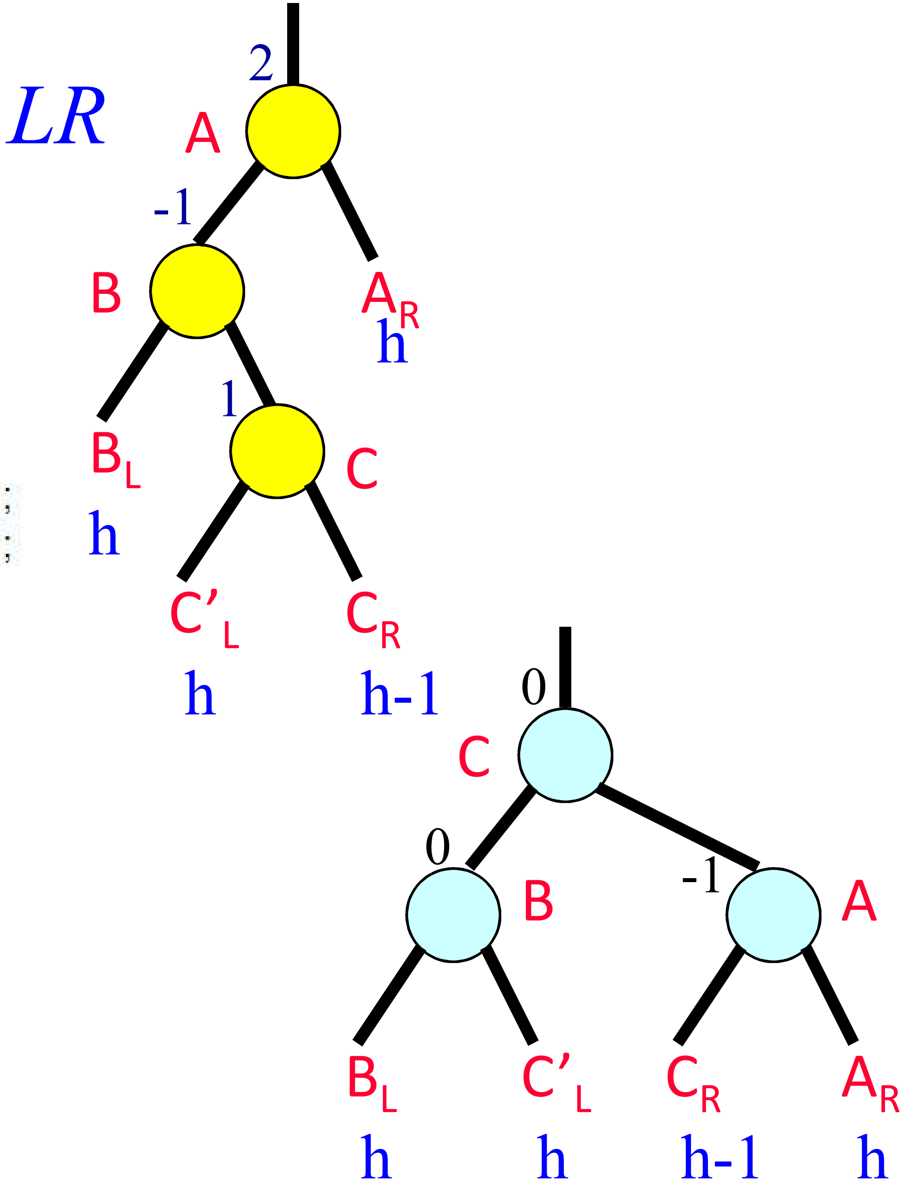


```
// Is tree unbalanced?
if (a->bf == 0 || a->bf + d == 0) { // tree still balanced
    a->bf += d; return;
}
```

```
// tree unbalanced, determine rotation type
if (d == 1) {// left imbalance
    if (b->bf == 1) {// rotation type LL
        a->leftChild = b->rightChild;
        b->rightChild = a; a->bf = 0; b->bf = 0;
        rootSub = b; // b is the new root of the subtree
    }
```



```
else { // rotation type LR
    c = b->rightChild;
    b->rightChild = c->leftChild;
    a->leftChild = c->rightChild;
    c->leftChild = b;
    c->rightChild = a;
    switch (c->bf) {
    case 0:
        b->bf = 0; a->bf = 0;
        break;
    case 1:
        a->bf = -1; b->bf = 0;
        break;
    case -1:
        b->bf = 1; a->bf = 0;
        break;
    }
    c->bf = 0; rootSub = c; // c is the new root of the subtree
} // end of LR
} // end of left imbalance
```



```
else { // right imbalance: this is symmetric to left imbalance
    if (b->bf == -1) {// rotation type RR
        a->rightChild = b->leftChild;
        b->leftChild = a; a->bf = 0; b->bf = 0;
        rootSub = b; // b is the new root of the subtree
    }
    else { // rotation type RL
        c = b->leftChild;
        b->leftChild = c->rightChild;
        a->rightChild = c->leftChild;
        c->rightChild = b;
        c->leftChild = a;
        switch (c->bf) {
        case 0:
            b->bf = 0; a->bf = 0;
            break;
        case 1:
            b->bf = -1; a->bf = 0;
            break;
        case -1:
            a->bf = 1; b->bf = 0;
            break;
        }
        c->bf = 0; rootSub = c; // c is the new root of the subtree
    } // end of RL
}
```

```
    // Subtree with root b has been rebalanced.
    if (pa == NULL) root = rootSub;
    else if (a == pa->leftChild) pa->leftChild = rootSub;
    else pa->rightChild = rootSub;
    return;
} // end of AVL::Insert
```