

Data Structure:

AVL Tree

Binary Search Tree

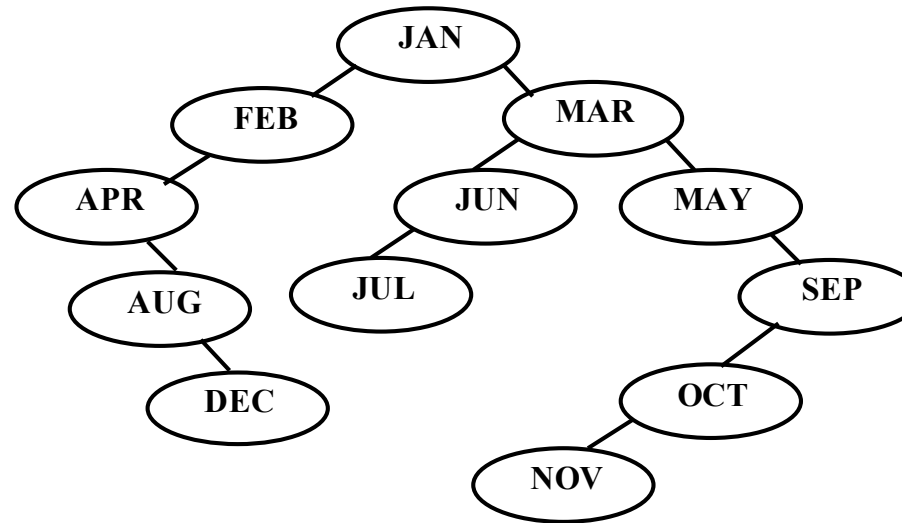
- for every node X in the tree,
 - the values of all the **keys in its left subtree are smaller** than the key value in X
 - the values of all the **keys in its right subtree are larger** than the key value in X

Binary Search Tree

Build binary search tree with Jan ... Dec

Binary Search Tree

Build binary search tree with Jan ... Dec

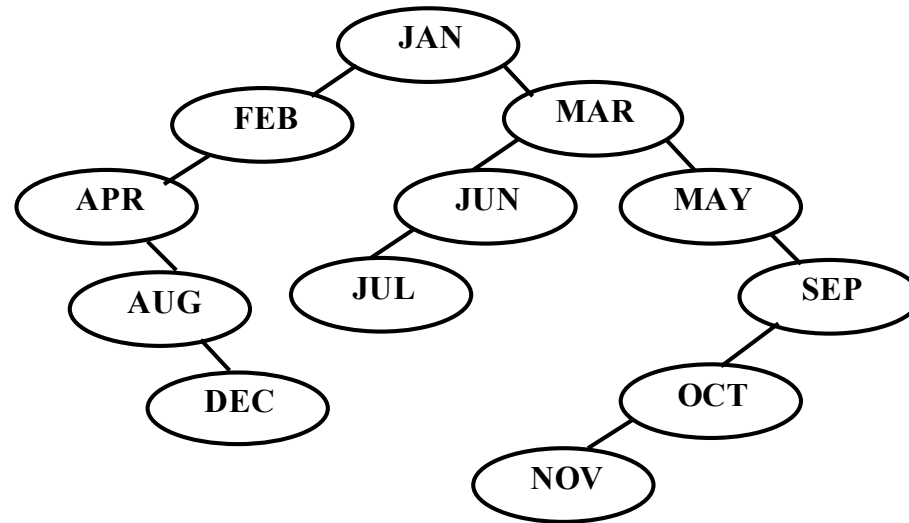


How many comparison do you need to search NOV?

what is the average number of comparisons?

Binary Search Tree

Build binary search tree with Jan ... Dec



How many comparison do you need to search NOV?

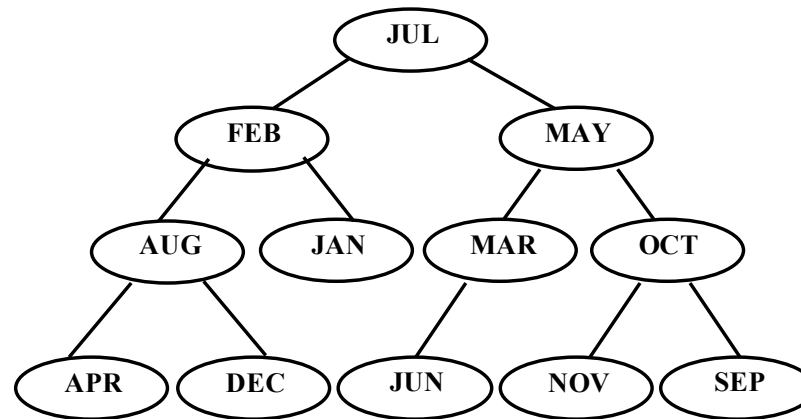
What is the average number of comparisons?

$$1 \text{ (JAN)} + 2 \text{ (FEB)} + 2 \text{ (MAR)} + 3 \text{ (APR)} + 3 \text{ (MAY)} + 3 \text{ (JUN)} + 4 \text{ (JUL)} + 4 \text{ (AUG)} + 4 \text{ (SEP)} + 5 \text{ (OCT)} + 6 \text{ (NOV)} + 5 \text{ (DEC)} = 42$$

$$42 / 12 = 3.5$$

Binary Search Tree

Insert JUL, FEB, MAY, AUG, DEC, MAR, OCT, APR, JAN, JUN, SEP, and NOV



What is the maximum number of comparison?

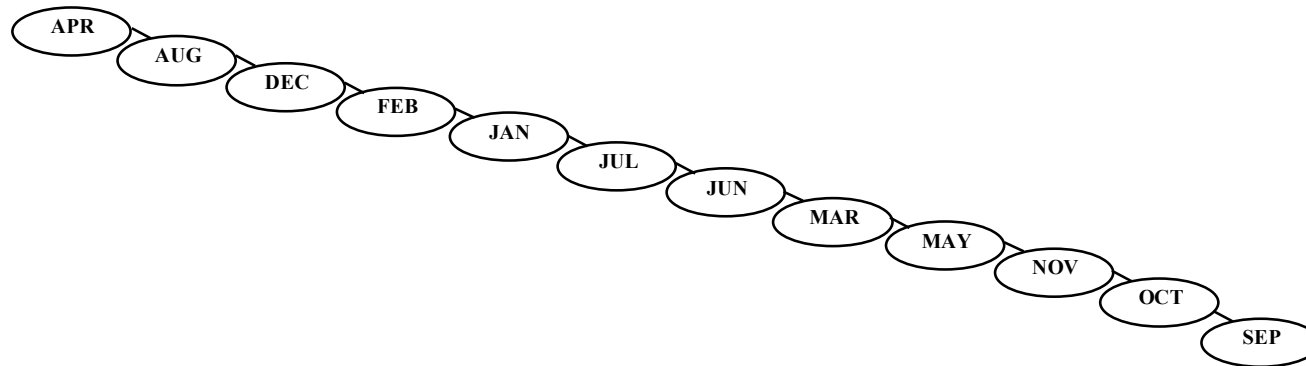
What is the average number of comparisons?

$$3 \text{ (JAN)} + 2 \text{ (FEB)} + 3 \text{ (MAR)} + 4 \text{ (APR)} + 2 \text{ (MAY)} + 4 \text{ (JUN)} + 1 \text{ (JUL)} + 3 \text{ (AUG)} + 4 \text{ (SEP)} + 3 \text{ (OCT)} + 4 \text{ (NOV)} + 4 \text{ (DEC)} = 37$$

$$37 / 12 = 3.1$$

Binary Search Tree

What if you insert the key in lexicographical order?



What is the maximum number of comparison?

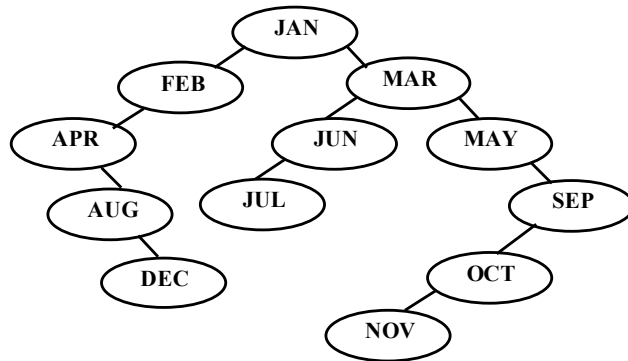
What is the average number of comparisons?

$$1 + 2 + \dots + 12 = 78$$

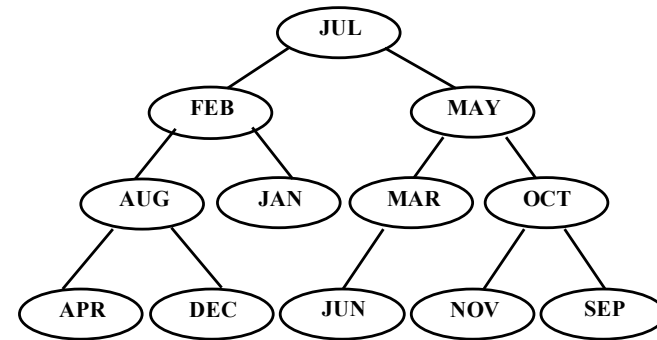
$$78 / 12 = 6.5$$

Binary Search Tree

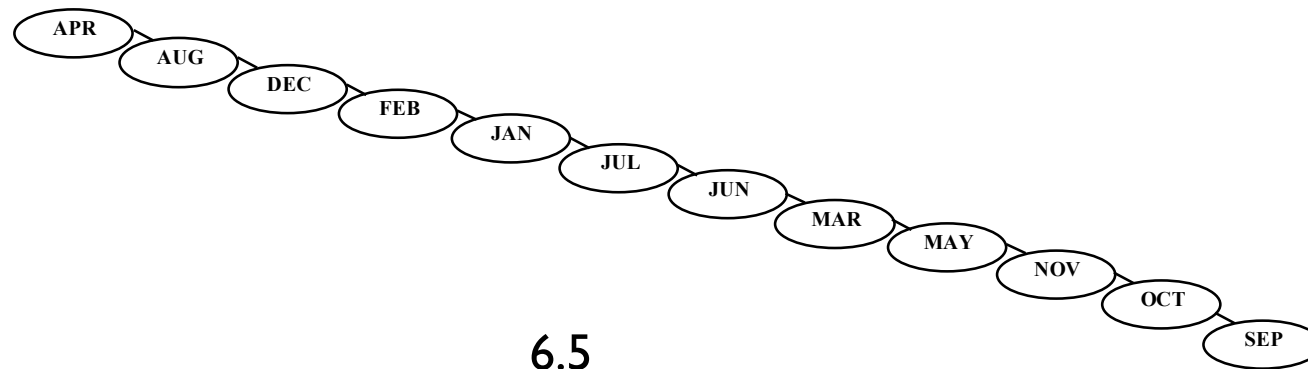
If equal probability, the average search and insertion time is $O(\log n)$



3.5



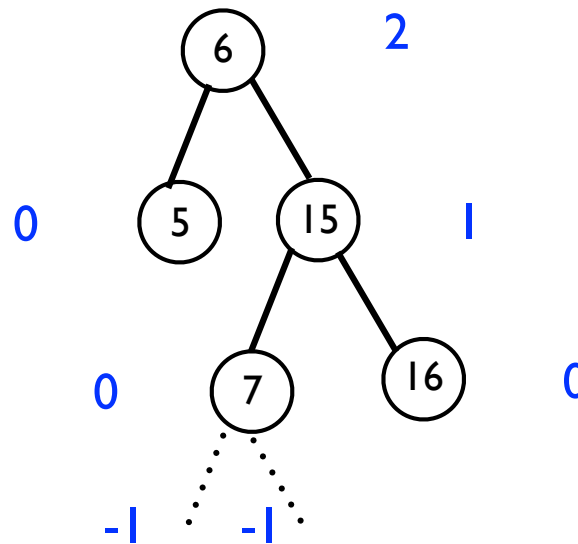
3.1



6.5

AVL (Adelson-Velskii and Landis) Tree

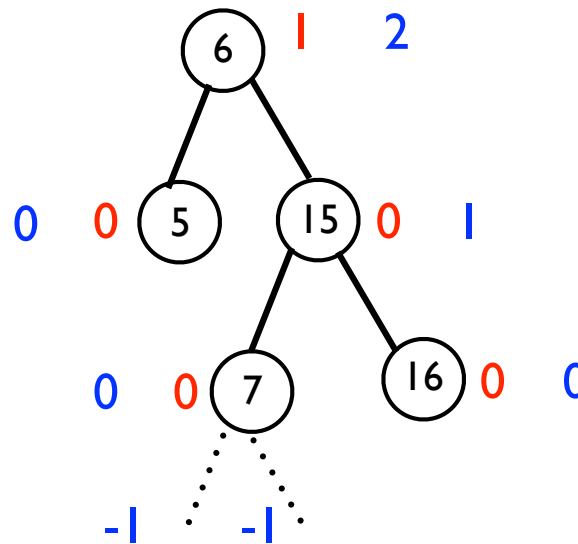
- binary search tree
- for every node in the tree, the **heights** of its left subtree and right subtree differ by **at most 1**.
 - the height of a null subtree is -1
 - the height of a subtree with one node is 0



h = height of the subtree

AVL (Adelson-Velskii and Landis) Tree

- binary search tree
- for every node in the tree, the **heights** of its left subtree and right subtree differ by **at most 1**.
 - the height of a null subtree is -1
 - the height of a subtree with one node is 0



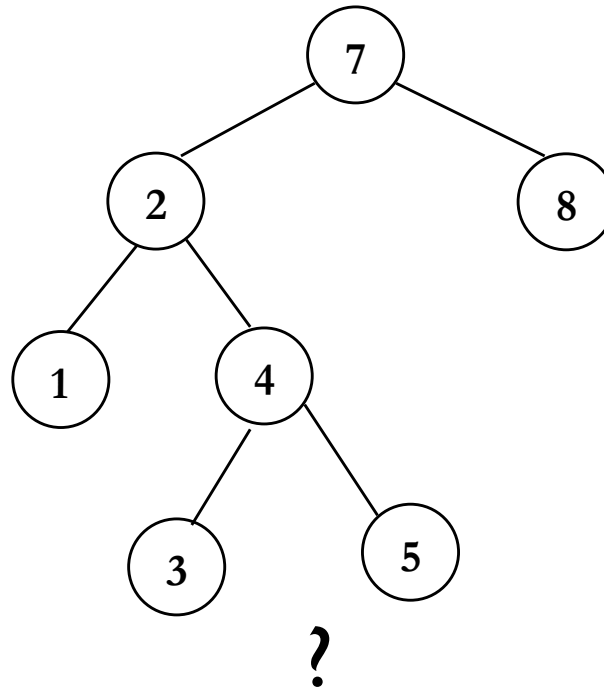
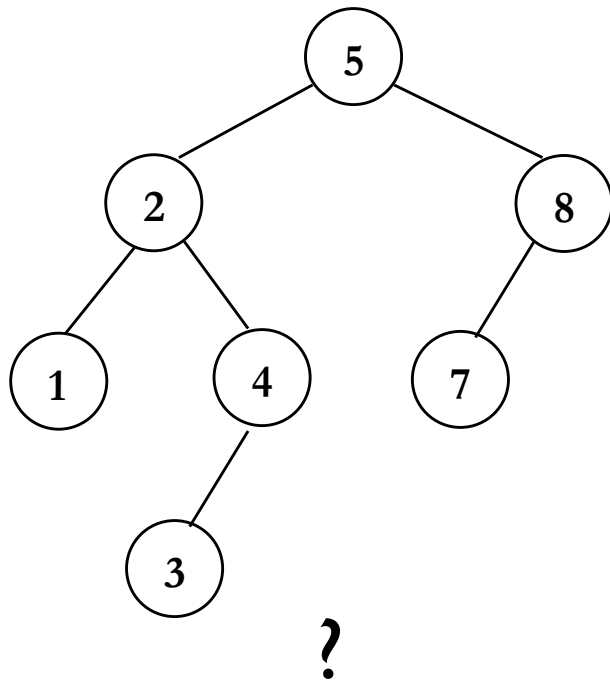
h = height of the subtree

$$\text{diff} = |h_L - h_R|$$

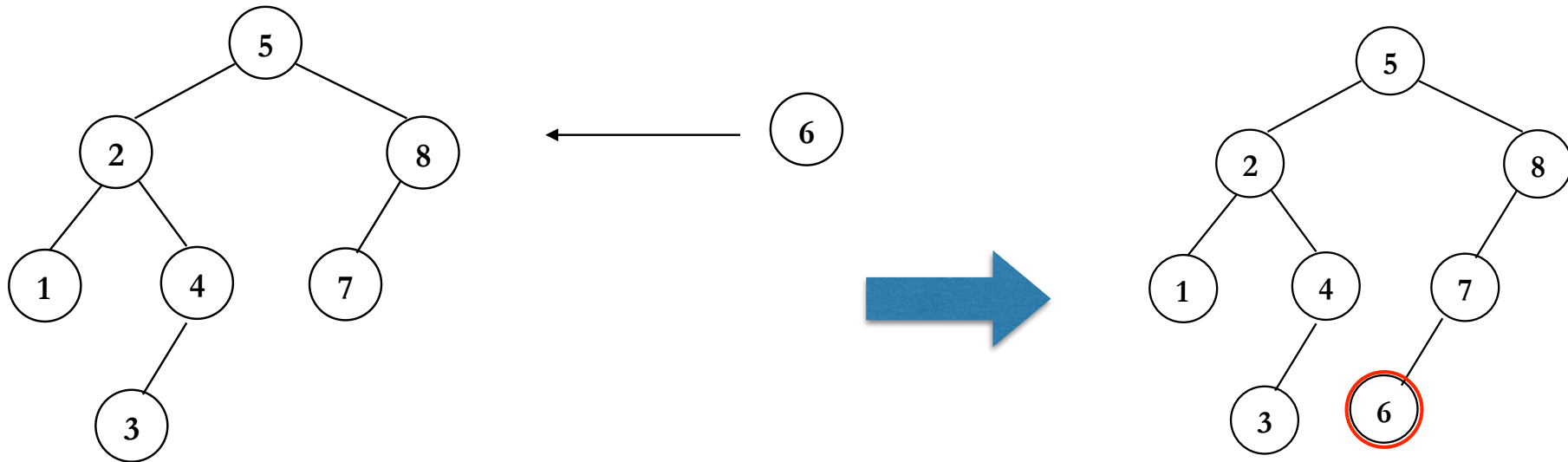
AVL (Adelson-Velskii and Landis) Tree

- An empty tree is height-balanced
- If T is a nonempty binary tree with T_L and T_R as its left and right subtree, T is height-balanced iff
 - (1) T_L and T_R are height-balanced and
 - (2) $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R

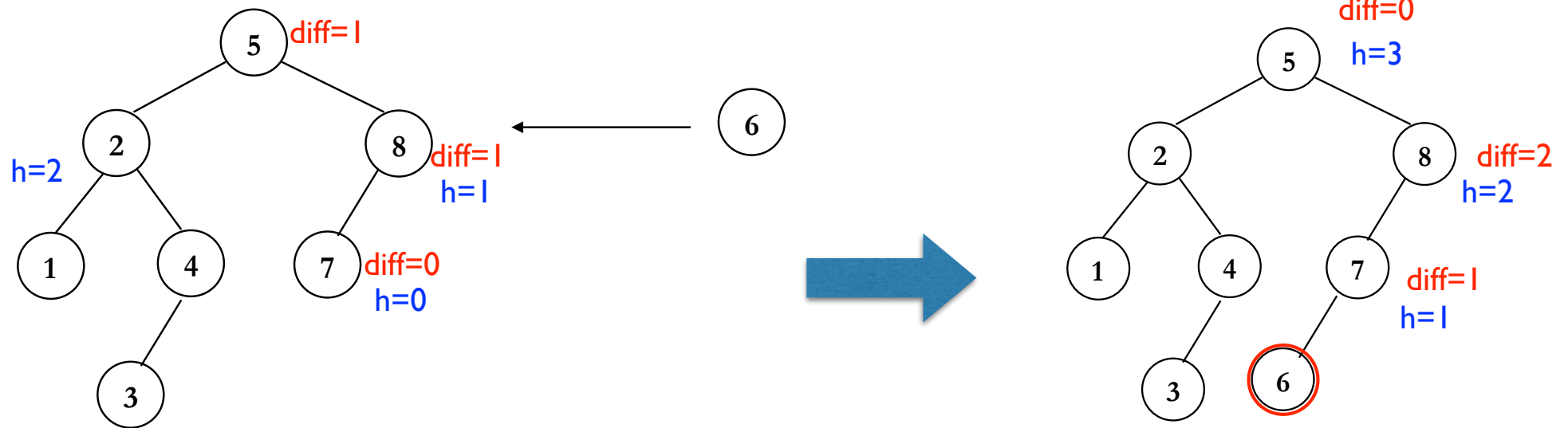
AVL (Adelson-Velskii and Landis) Tree



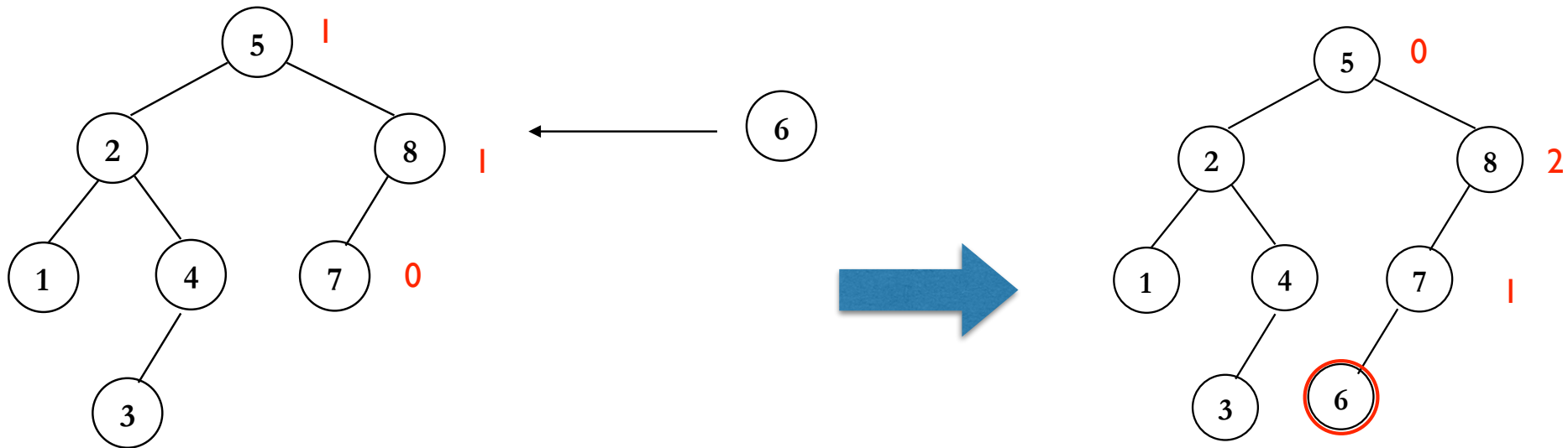
AVL Tree: insertion



AVL Tree: insertion



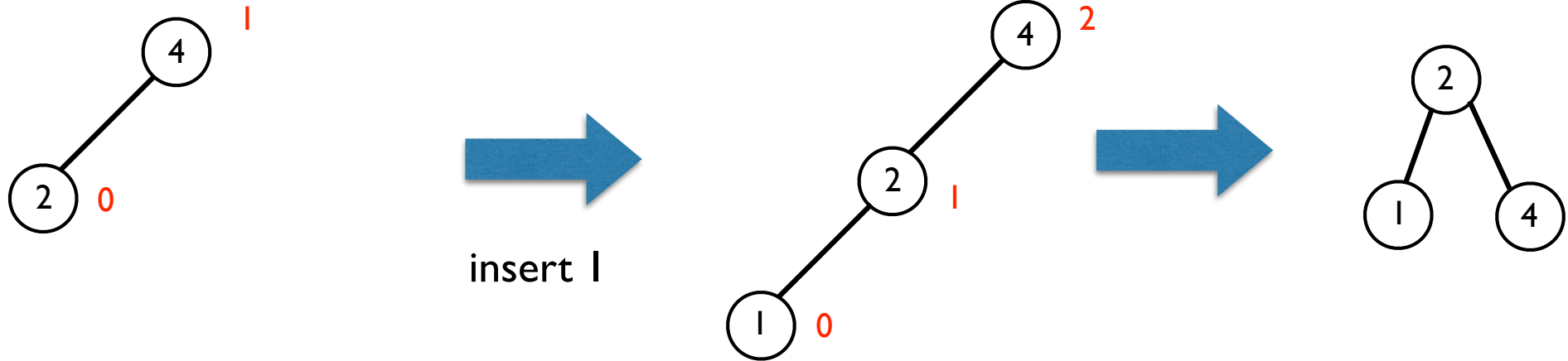
AVL Tree: insertion



- case I: an insertion into the **left subtree of the left child**
 - ▶ **single rotation**

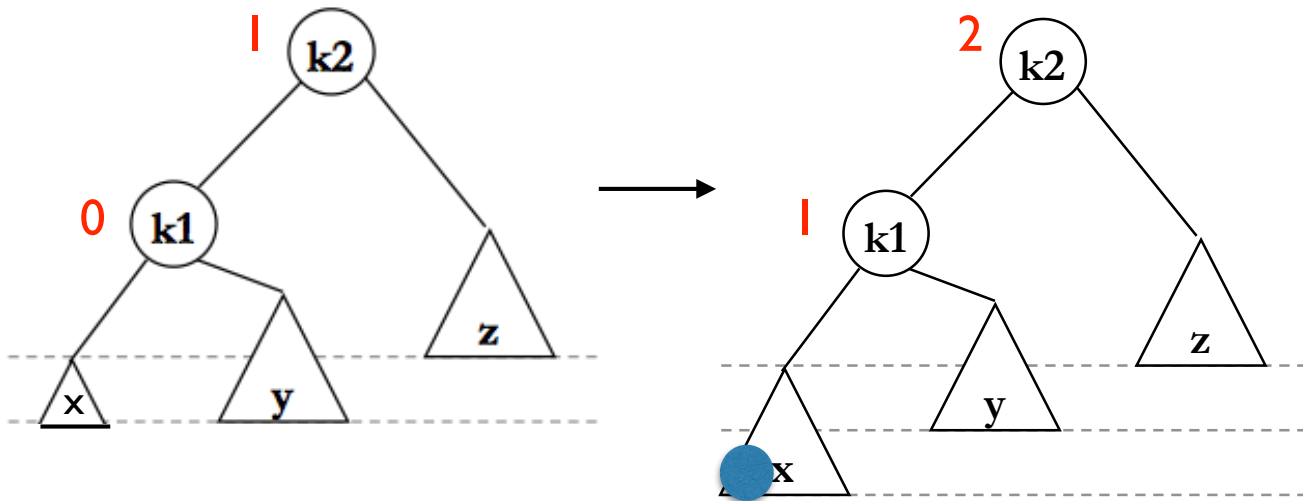
AVL Tree: insertion

- case I: an insertion into the **left subtree of the left child**
 - ▶ **single rotation**



AVL Tree: insertion

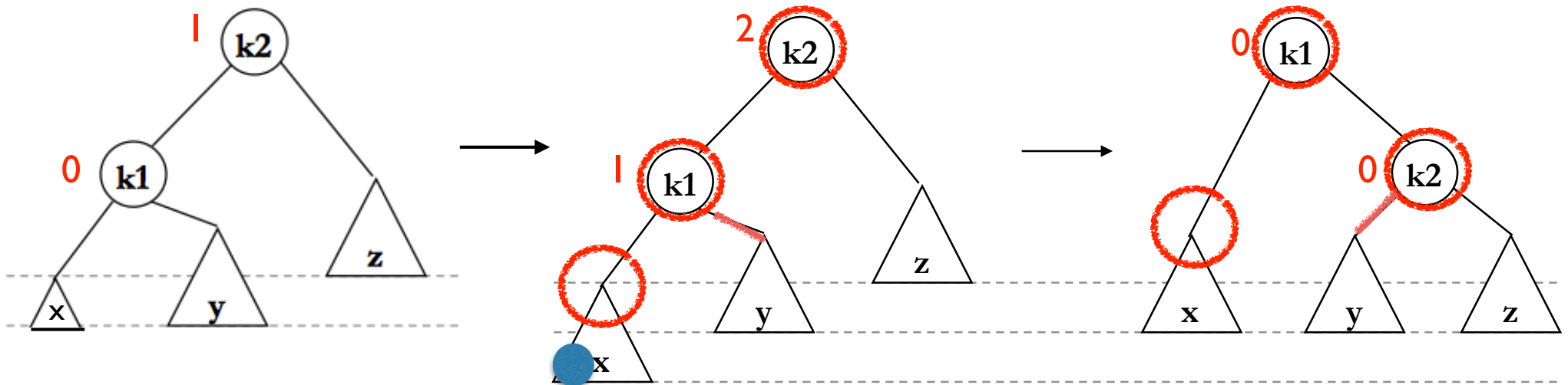
- case I: an insertion into the **left subtree of the left child**
 - ▶ **single rotation**



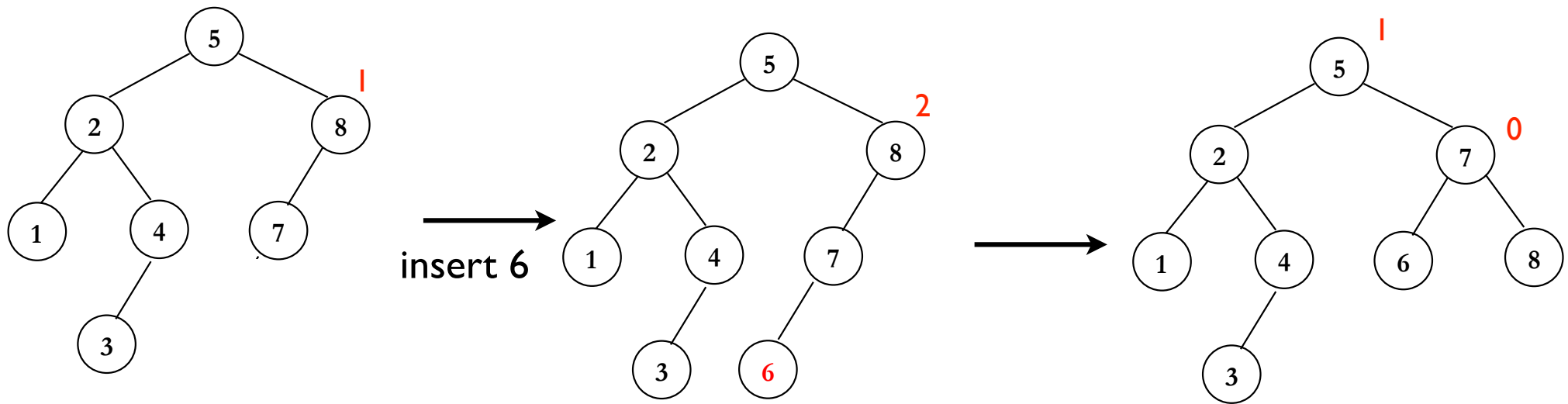
결과 tree의 height는 insertion 이전 tree의 height와 같음. 따라서 k_2 가 더 큰 tree 안에 embed 되어 있어도 다른 변경은 필요 없음.

AVL Tree: insertion

- case I: an insertion into the **left subtree of the left child**
 - ▶ **single rotation**

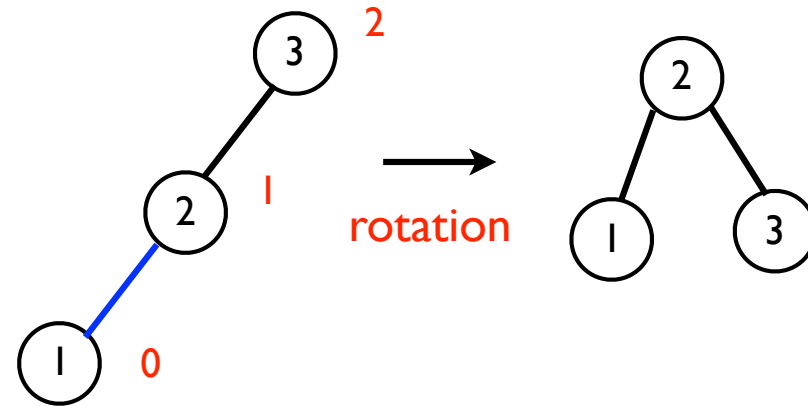
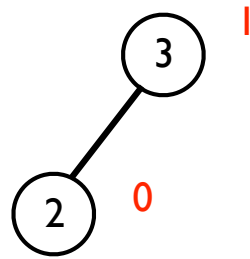


AVL Tree: insertion



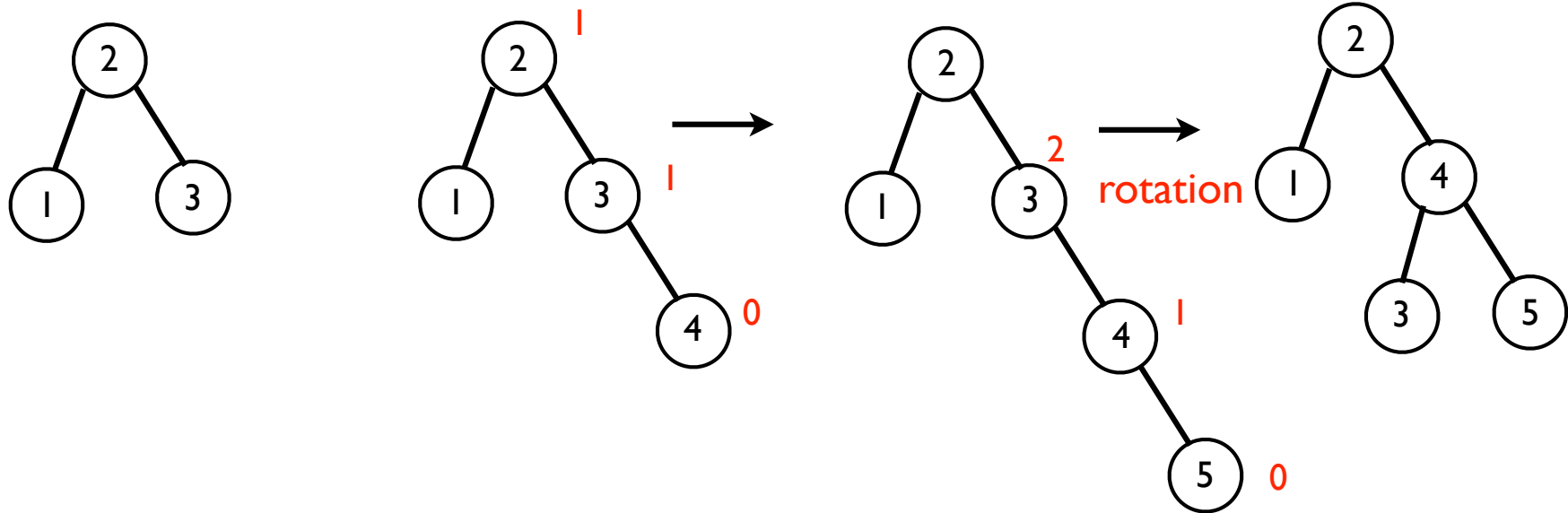
AVL Tree: insertion

insert 3, 2, 1, 4, 5, 6



AVL Tree: insertion

insert 3, 2, 1, 4, 5, 6

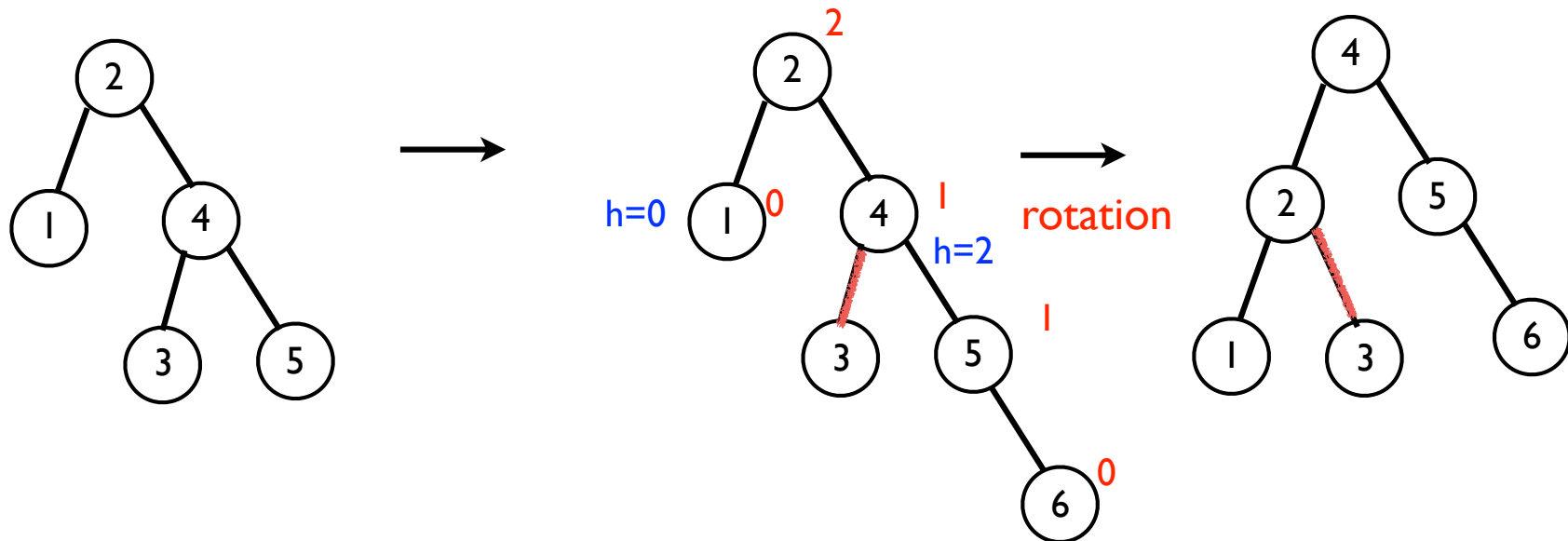


■ case 2: an insertion into the **right subtree of the right child** of the node A

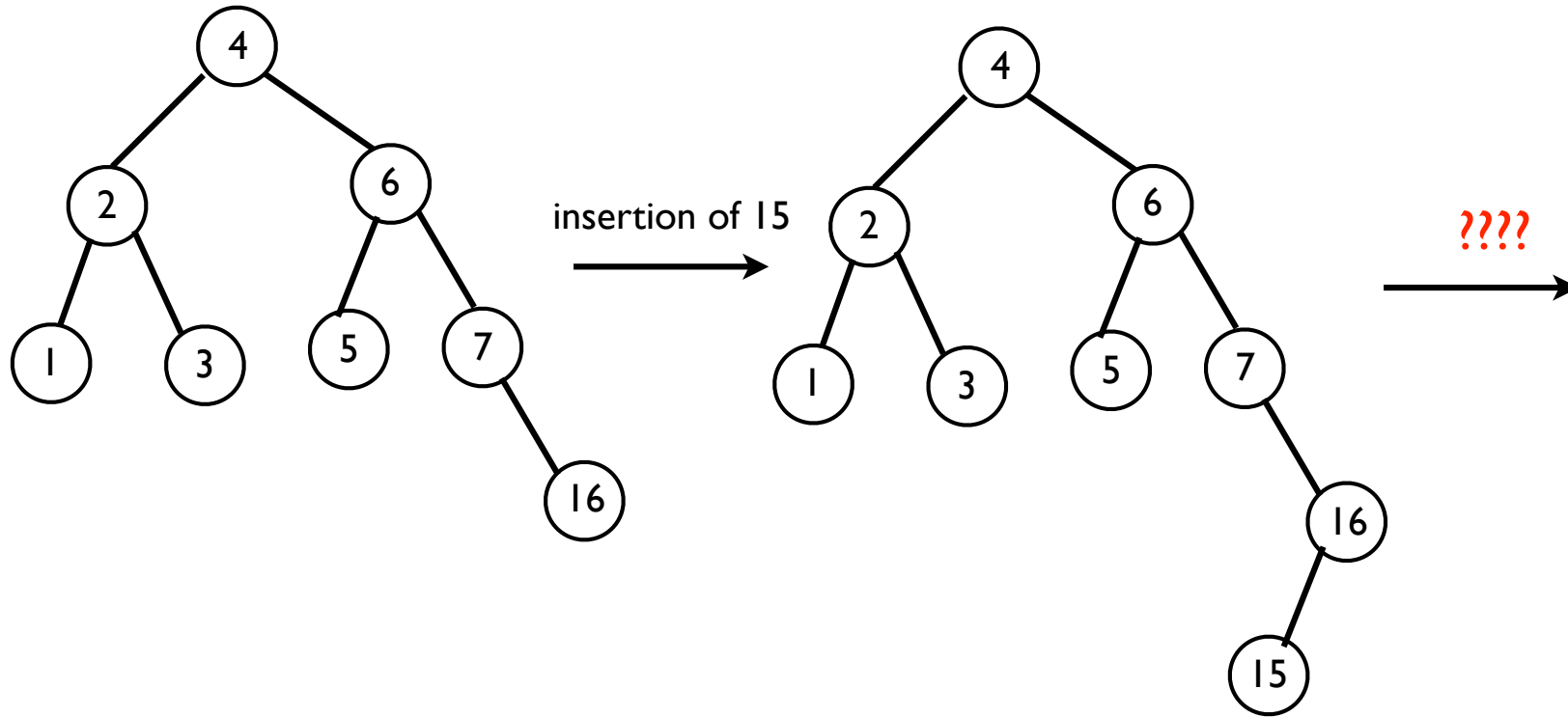
► single rotation

AVL Tree: insertion

insert 3, 2, 1, 4, 5, 6

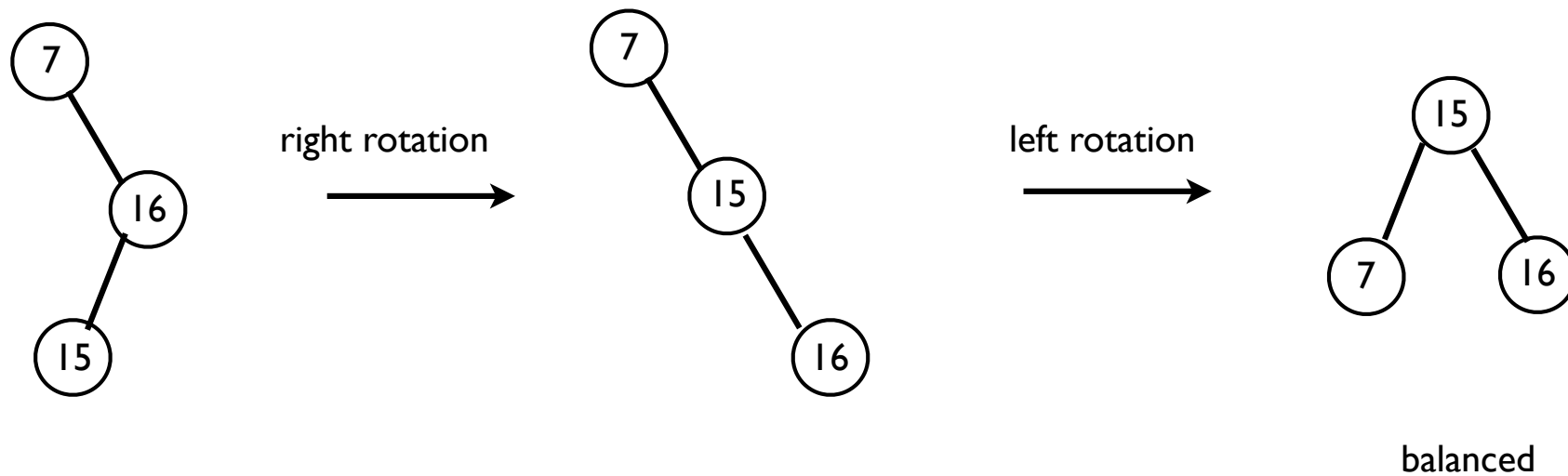


AVL Tree: insertion

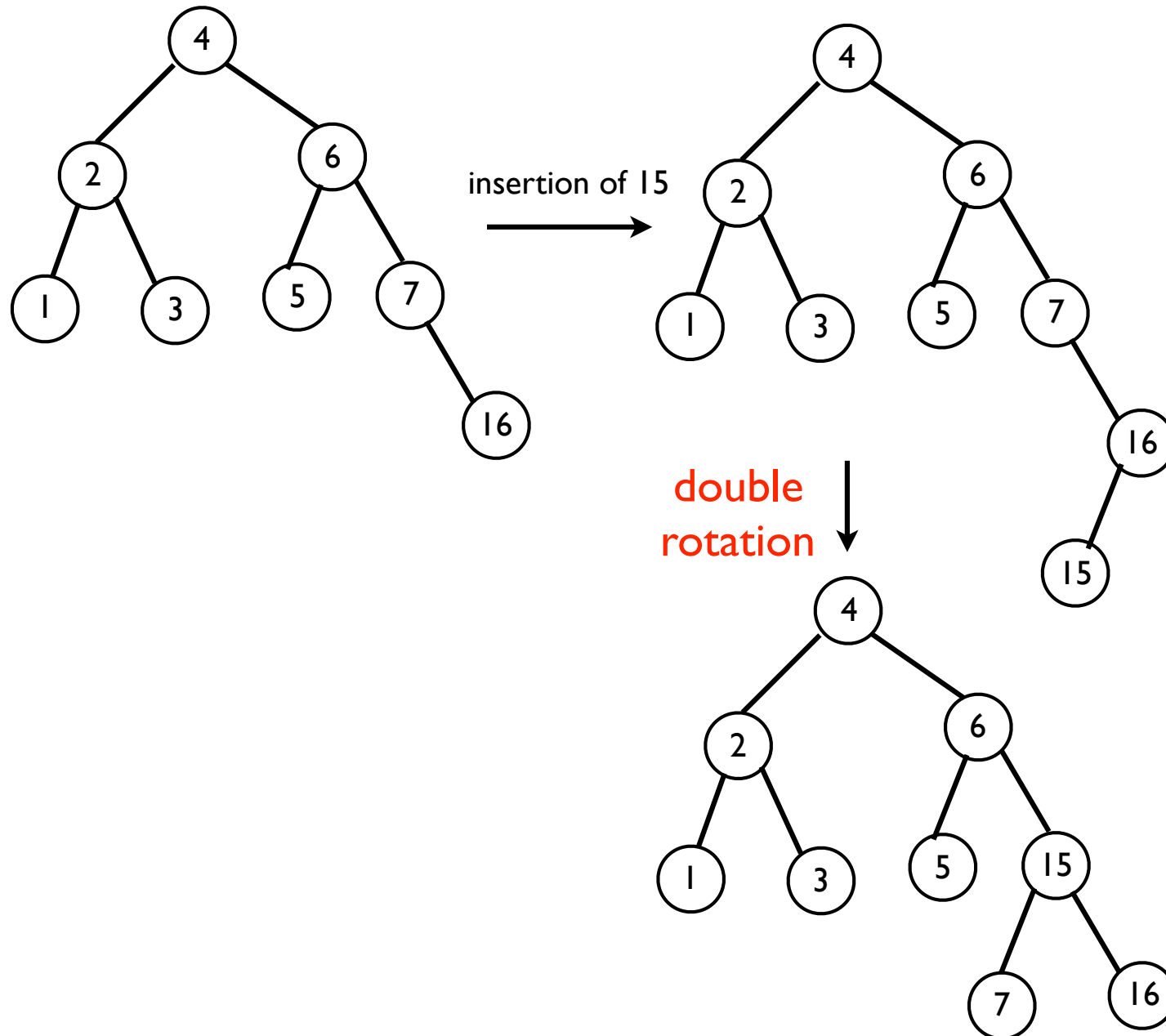


AVL Tree: insertion

- case 3: an insertion into the **left** subtree of the **right** child of the unbalanced node
 - ▶ **double rotation**

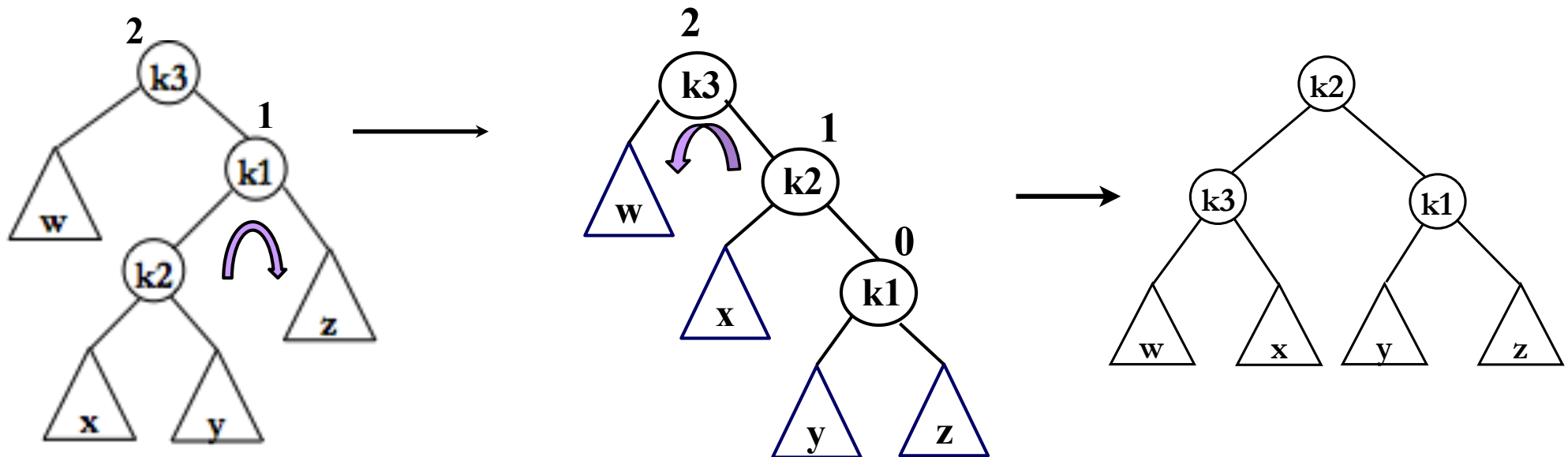


AVL Tree: insertion



AVL Tree: insertion

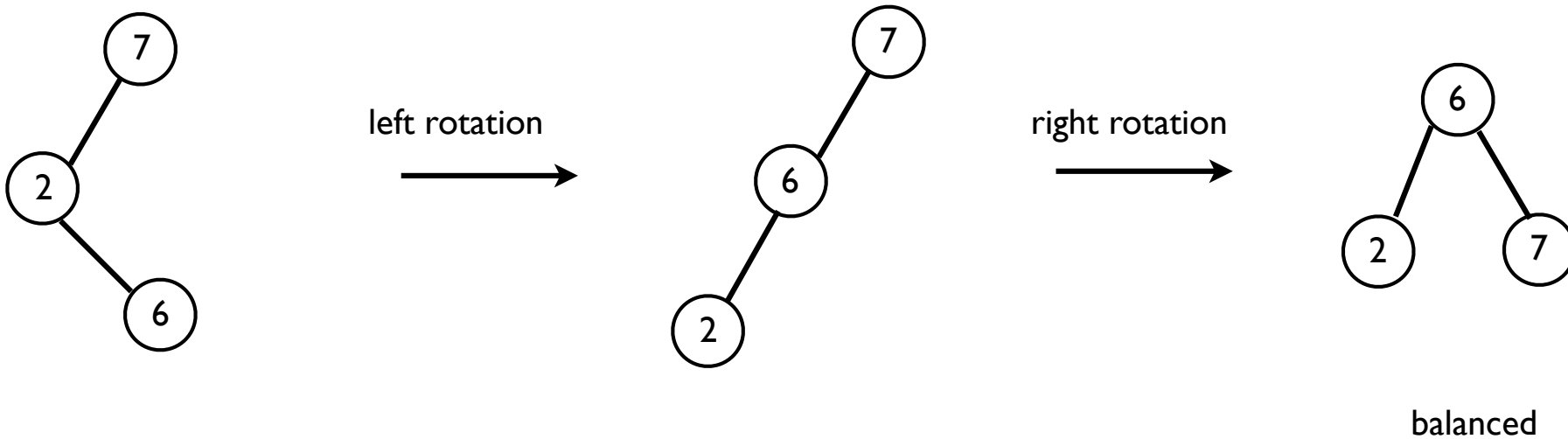
- case 3: an insertion into the **left** subtree of the **right** child of the unbalanced node
 - ▶ **double rotation**



right-left double rotation

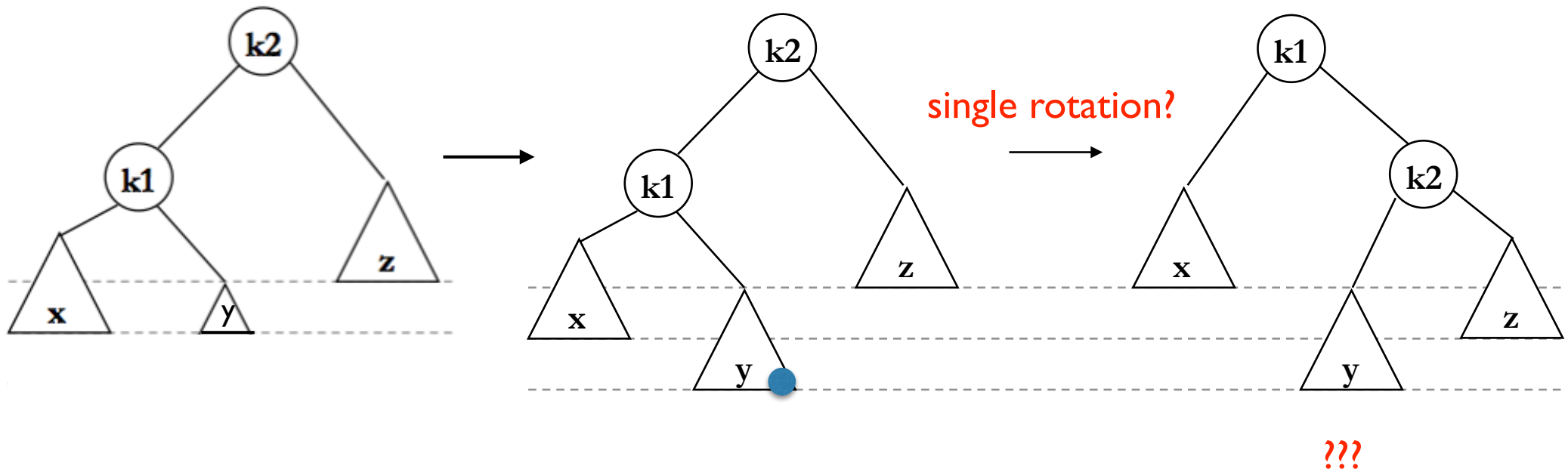
AVL Tree: insertion

- case 4: an insertion into the **right** subtree of the **left** child of the unbalanced node
 - ▶ **double rotation**



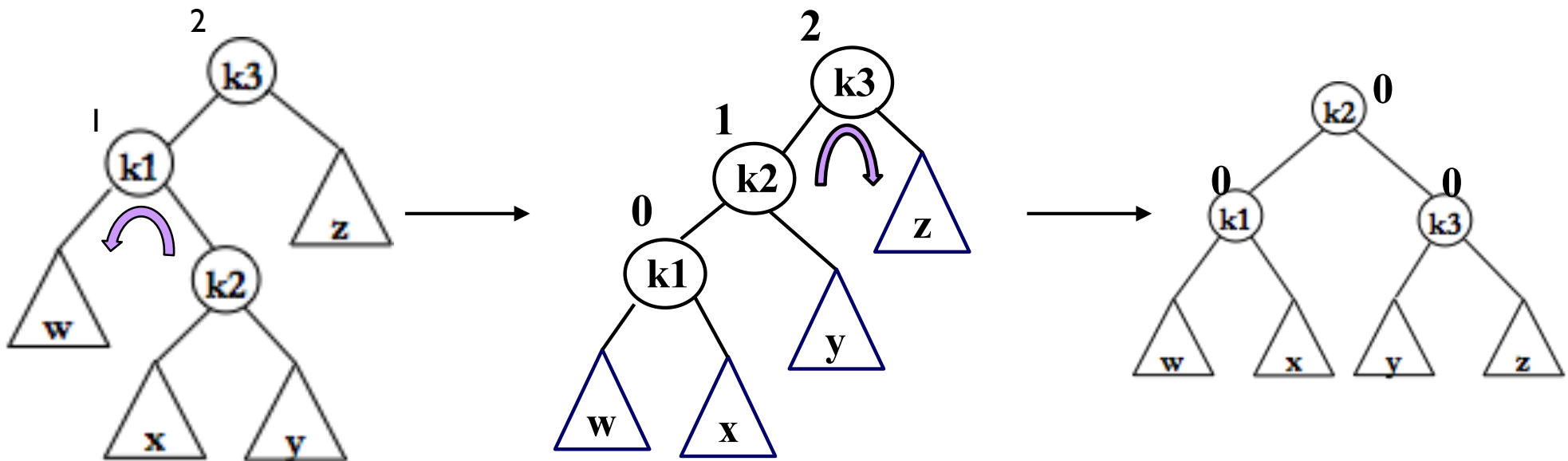
AVL Tree: insertion

- case 4: an insertion into the **right** subtree of the **left** child of the unbalanced node
 - ▶ **double rotation**



AVL Tree: insertion

- case 4: an insertion into the **right** subtree of the **left** child of the unbalanced node
 - ▶ **double rotation**



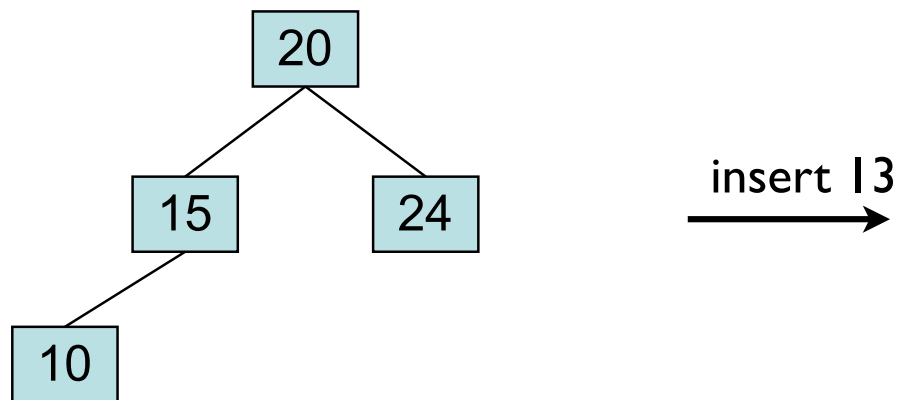
Left-right double rotation

AVL Tree: insertion

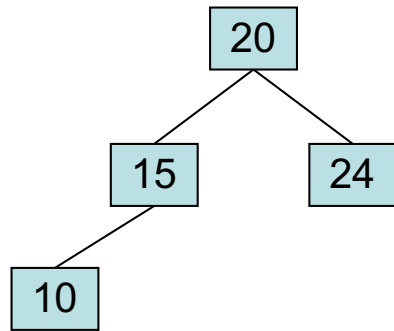
The node “A” ($|h_L - h_R| > 1$) needs to be rebalanced, when

- case 1: an insertion into the **left subtree of the left child** of the node A
 - ▶ single rotation (LL)
- case 2: an insertion into the **right subtree of the right child** of the node A
 - ▶ single rotation (RR)
- case 3: an insertion into the **right subtree of the left child** of the node A
 - ▶ double rotation (LR)
- case 4: an insertion into the **left subtree of the right child** of the node A
 - ▶ double rotation (RL)

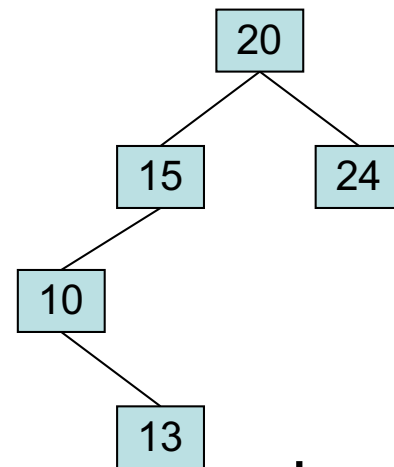
AVL Tree: insertion



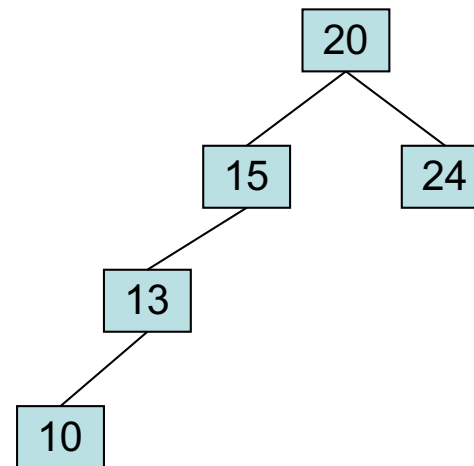
AVL Tree: insertion



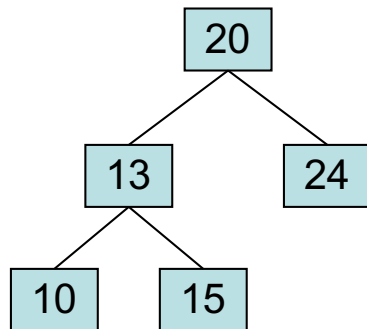
insert 13
→



↓
left rotation

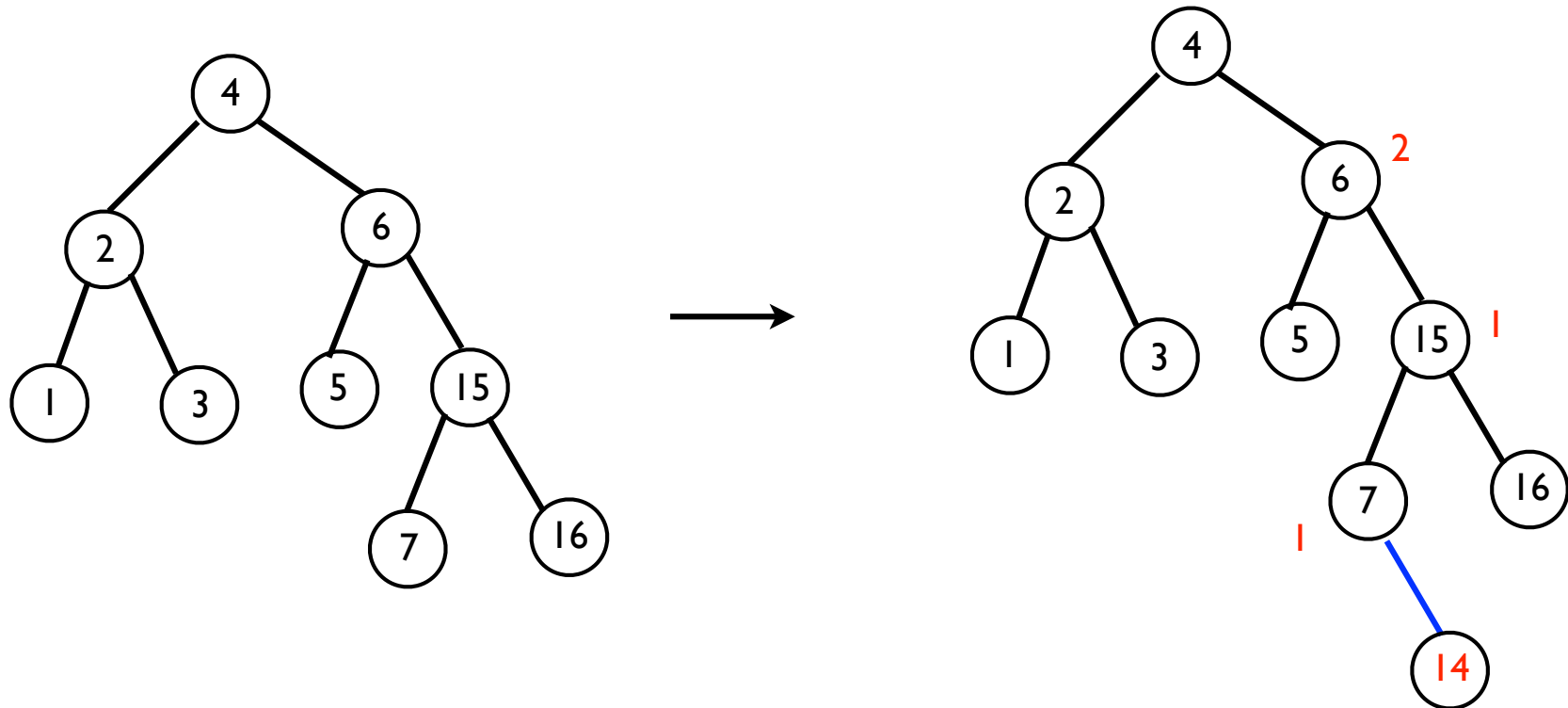


←
right rotation

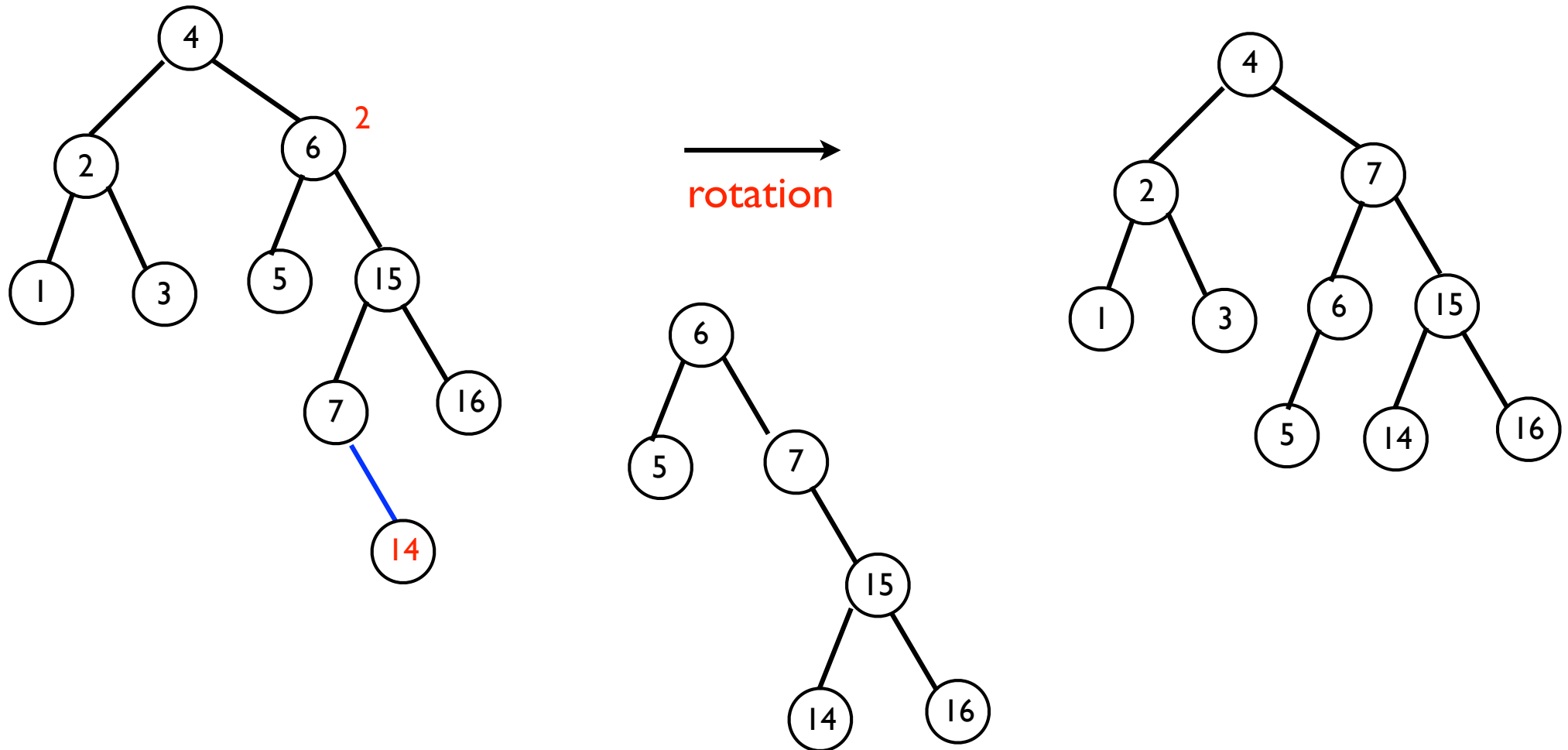


AVL Tree: insertion

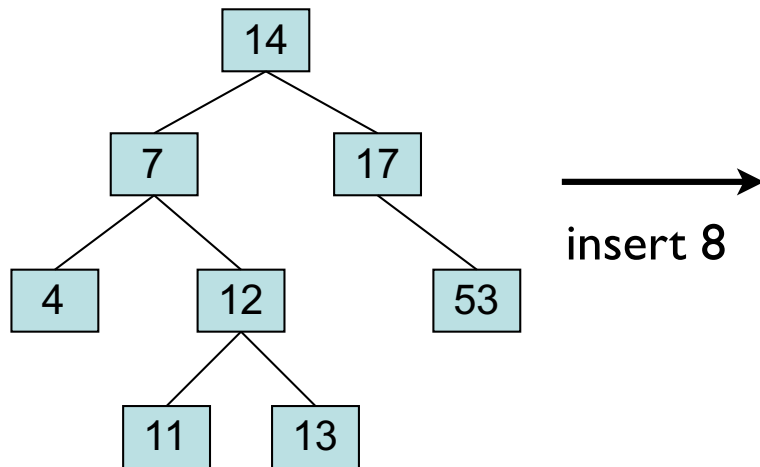
insert 14



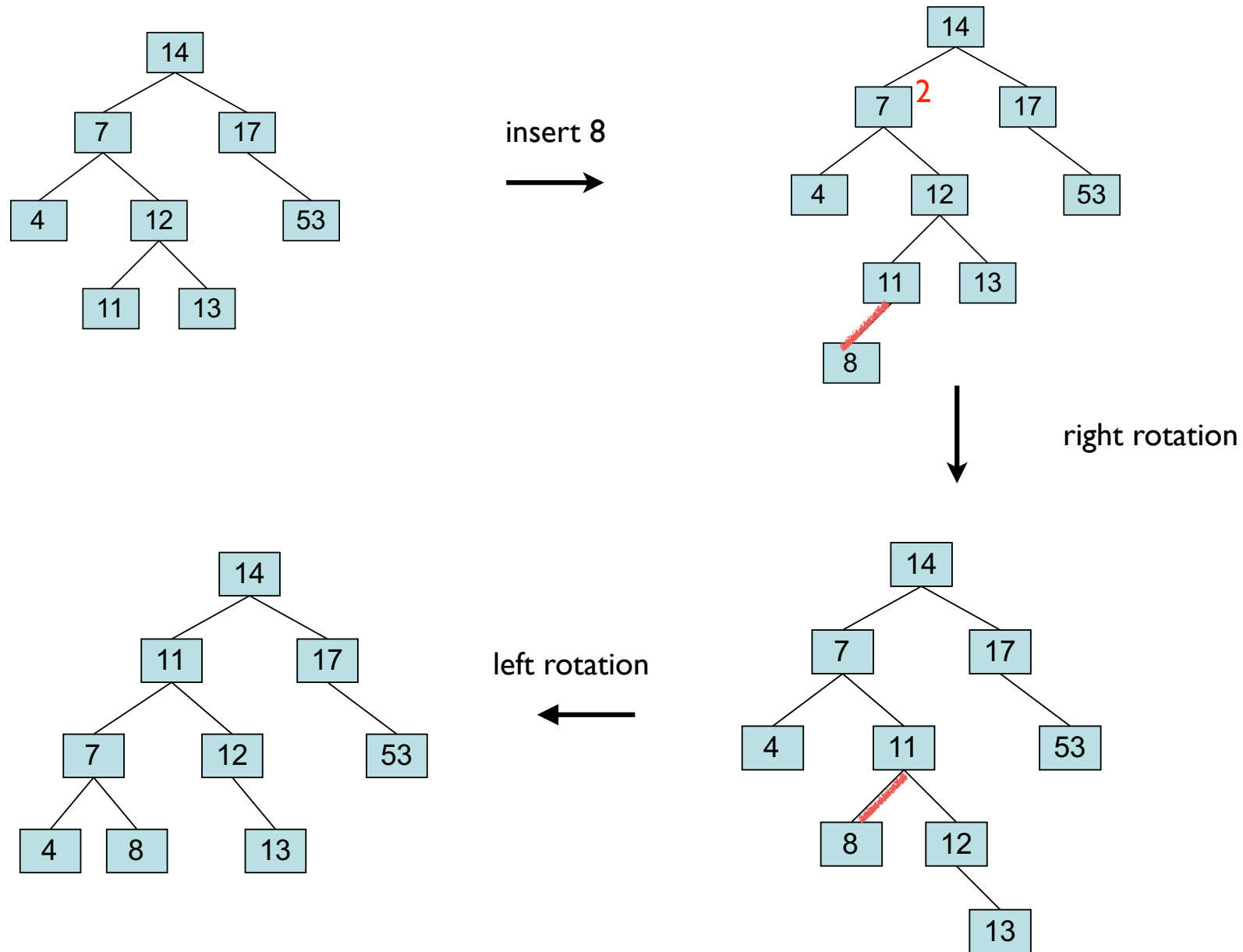
AVL Tree: insertion



AVL Tree: insertion



AVL Tree: insertion



AVL Tree: exercise

Insert sequence: 2, 1, 4, 5, 9, 3, 6, 7

Insert sequence: 3, 2, 1, 4, 5, 6, 7, 16, 15, 14, 13, 12, 11, 10, 8, 9

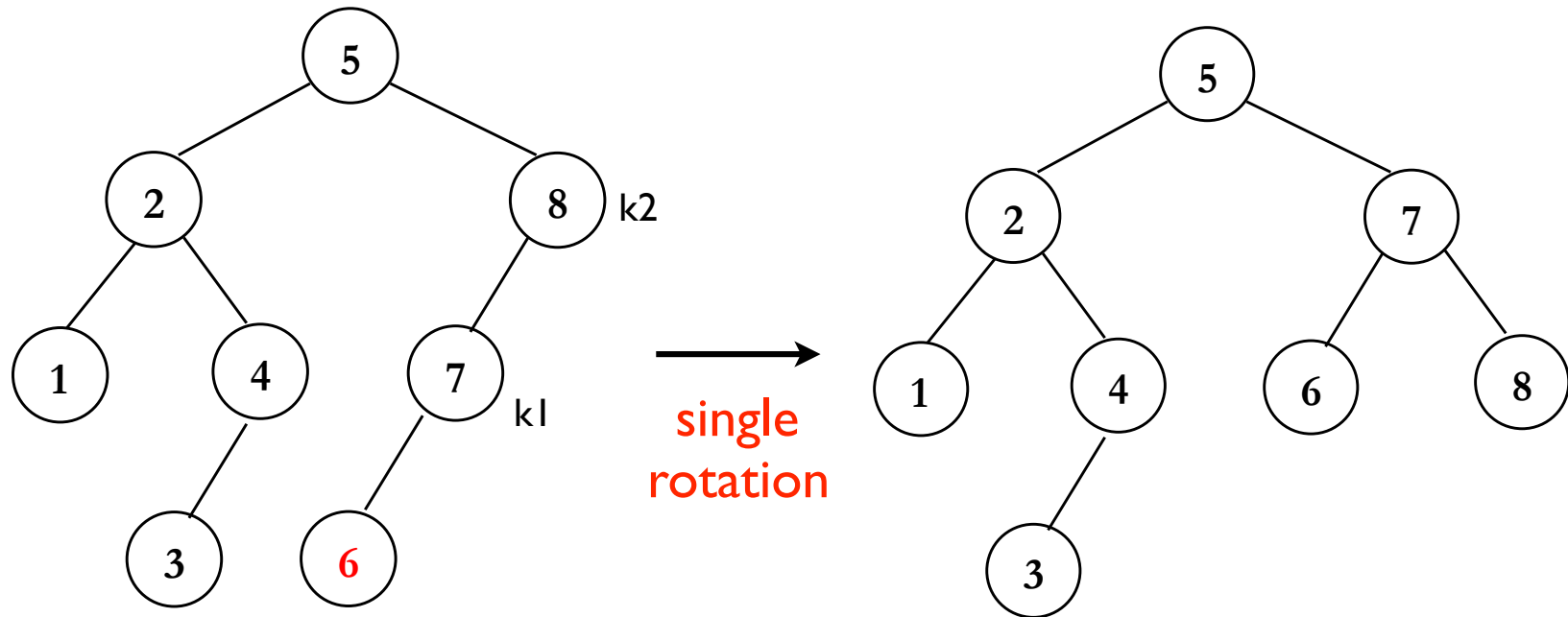
AVL Tree

```
struct AVLNode;  
typedef struct AVLNode *Position;  
typedef struct AVLNode *AVLTree;
```

```
struct AVLNode  
{  
    ElementType Element;  
    AVLTree Left;  
    AVLTree Right;  
    int Height;  
}
```

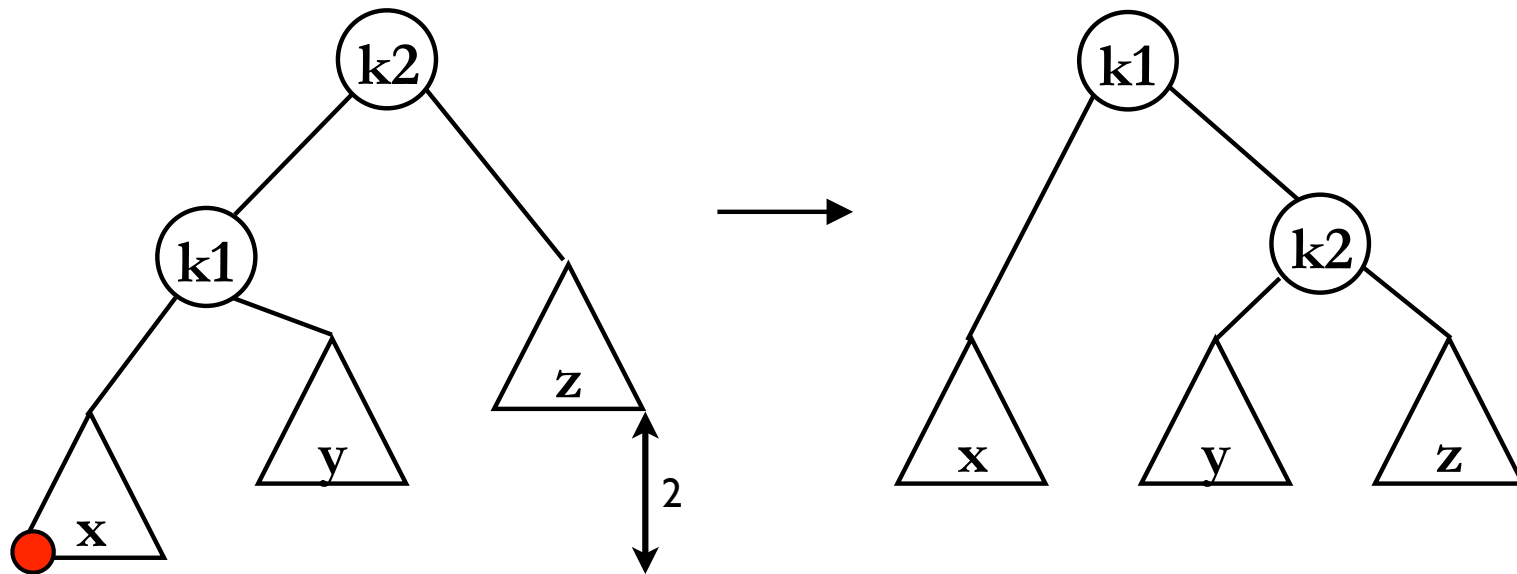
```
int Height(Position P)  
{  
    if (P == NULL)  
        return -1;  
    else  
        return P->Height;  
}
```

AVL Tree: insertion



K1->Right = K2;

AVL Tree: insertion



$K2 \rightarrow \text{Left} = K1 \rightarrow \text{Right};$
 $K1 \rightarrow \text{Right} = K2;$

AVL Tree: rotation

```
Position SingleRotateWithLeft( Position K2 ) /* LL */
```

```
{
```

```
    Position K1;
```

```
    K1 = K2->Left;
```

```
    K2->Left = K1->Right;      /* Y */
```

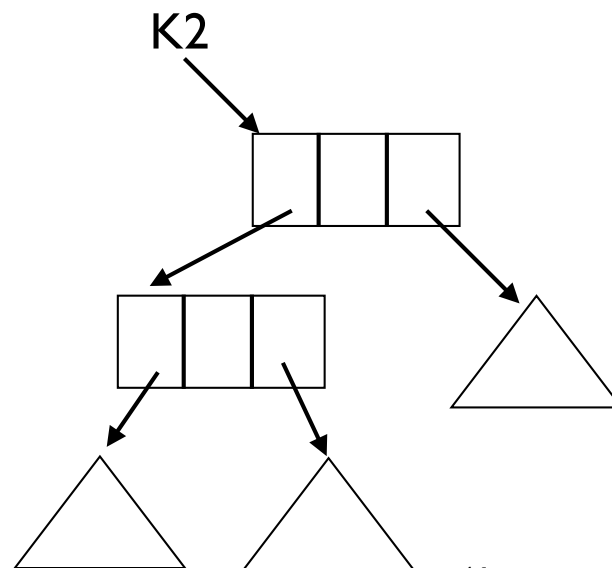
```
    K1->Right = K2;
```

```
    K2->Height = Max( Height( K2->Left ), Height( K2->Right ) ) + 1;
```

```
    K1->Height = Max( Height( K1->Left ), K2->Height ) + 1;
```

```
    return K1;                /* New root */
```

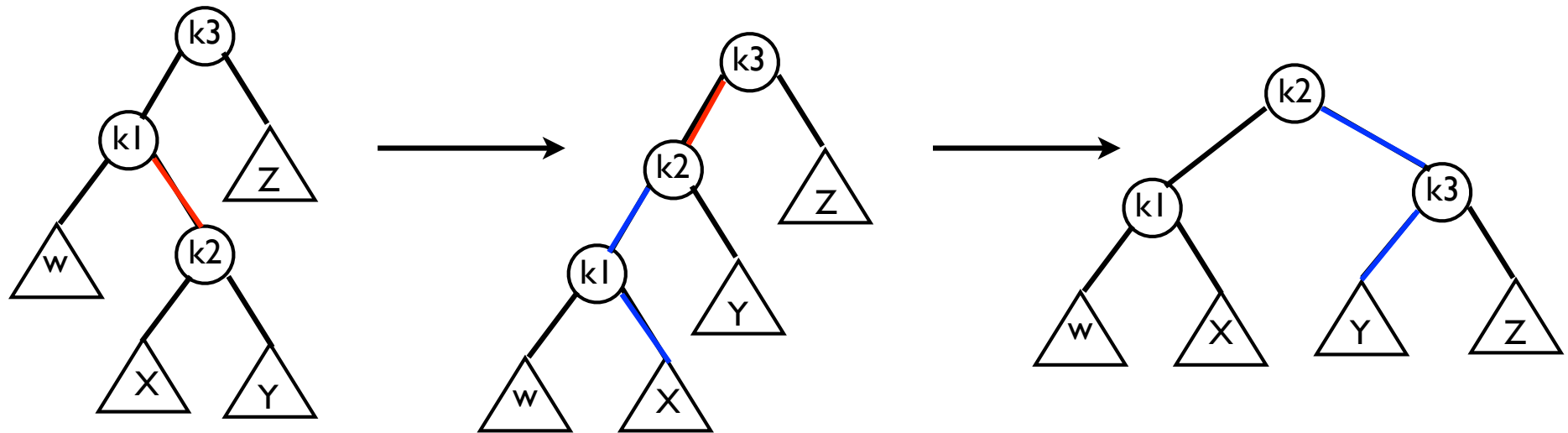
```
}
```



AVL Tree: rotation

```
static Position DoubleRotateWithLeft ( Position K3 ) /*LR */
{
    /* rotate between K1 and K2 */
    K3->Left = SingleRotateWithRight( K3->Left ); /* k2 */

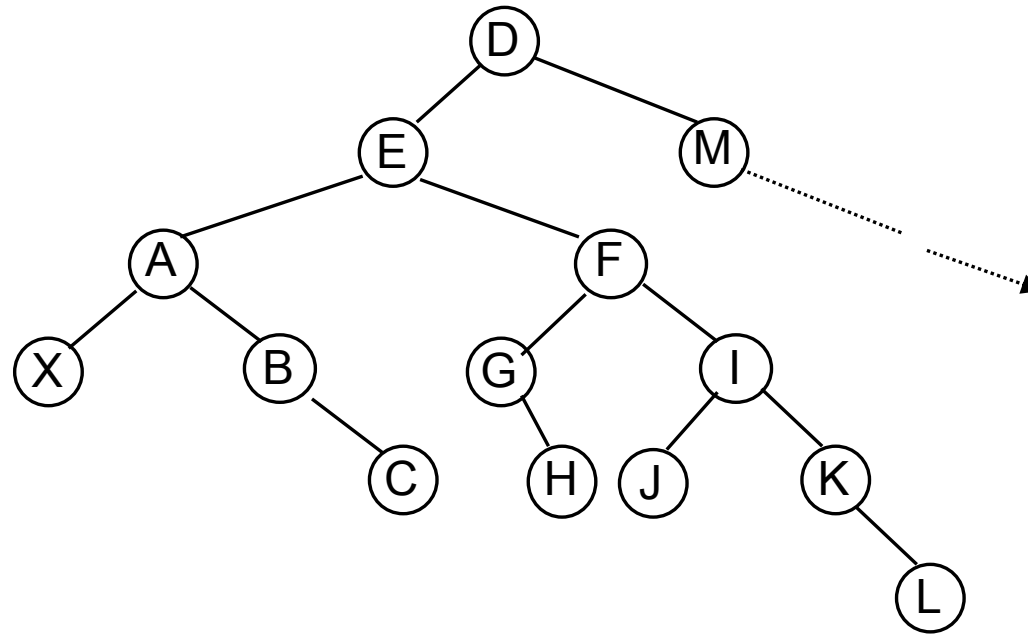
    /* rotate between K3 and K2 */
    return SingleRotateWithLeft( K3 ); /* K2 */
}
```



AVL Tree: insertion

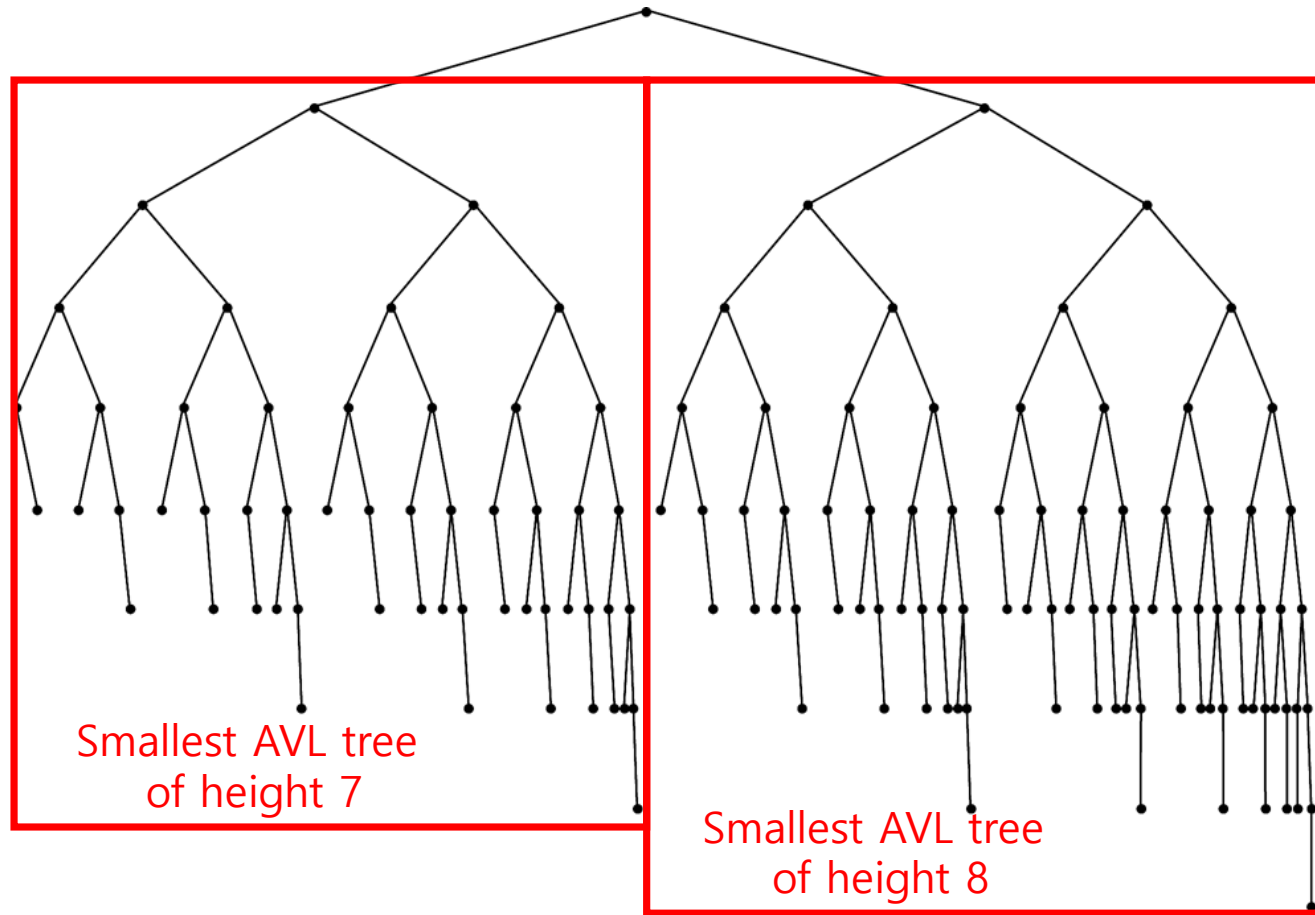
```
AVLTree Insert( ElementType X, AVLTree T ) {  
  
    if( T == NULL ) {                                     /* found the right place for insertion*/  
        T = malloc( sizeof( struct AVLNode ) );  
        if( T == NULL )  
            FatalError( "Out of space!!!" );  
        else {  
            T->Element = X; T->Height = 0;  
            T->Left = T->Right = NULL;  
        }  
    } else if ( X < T->Element ) {  
        T->Left = Insert( X, T->Left );                  /* BST*/  
        if( Height( T->Left ) - Height( T->Right ) == 2 )  
            if( X < T->Left->Element )  
                T = SingleRotateWithLeft( T );  
            else  
                T = DoubleRotateWithLeft( T );  
    } else if( X > T->Element ) {  
        T->Right = Insert( X, T->Right );  
        if( Height( T->Right ) - Height( T->Left ) == 2 )  
            if( X > T->Right->Element )  
                T = SingleRotateWithRight( T );  
            else  
                T = DoubleRotateWithRight( T );  
    }  
    T->Height = Max( Height( T->Left ), Height( T->Right ) ) + 1;  
    return T;  
}
```

AVL Tree: deletion



- Restructuring occurs only once by insertion, why?
- How should we apply rotations for deletion of X in the AVL tree shown above, for example?
- How many restructuring do we need for deletion?

Smallest AVL Tree of height 9



Height of AVL Tree

Denote N_h the minimum number of nodes in an AVL tree of height h

$$N_0 = 1, N_1 = 2 \text{ (base)}$$

$$N_h = N_{h-1} + N_{h-2} + 1 \text{ (recursive definition)}$$

$$N > N_h = N_{h-1} + N_{h-2} + 1$$

$$> 2 \times N_{h-2} > 4 \times N_{h-4} > \dots > 2^i \times N_{h-2i}$$

If h is even, let $i = h/2 - 1$.

The equation becomes $N > 2^{h/2-1} N_2, N > 2^{h/2-1} \times 4. h = O(\log N)$

If h is odd, let $i = (h - 1)/2$.

The equation becomes $N > 2^{(h-1)/2} N_1, N > 2^{(h-1)/2} \times 2. h = O(\log N)$

Thus, many operations (i.e. searching) on an AVL tree will take $O(\log N)$ time.