Data Structure:
AVL Tree

Binary Search Tree

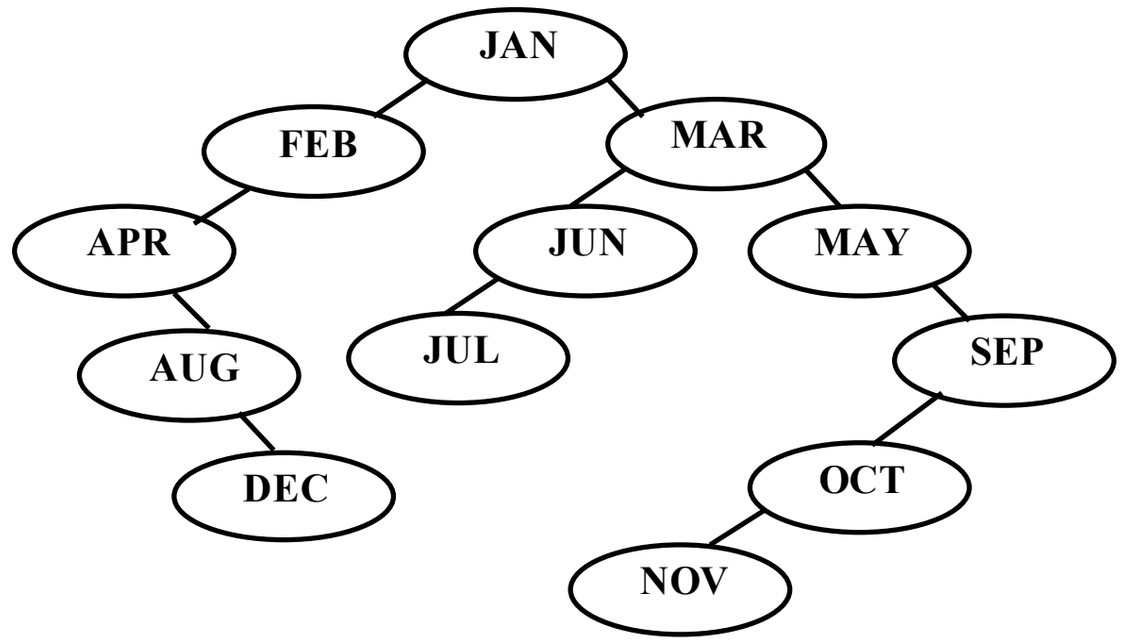
- for every node X in the tree,
  - the values of all the keys in its left subtree are smaller than the key value in X
  - the values of all the keys in its right subtree are larger than the key value in X

Binary Search Tree

Build binary search tree with Jan … Dec

Binary Search Tree

Build binary search tree with Jan … Dec

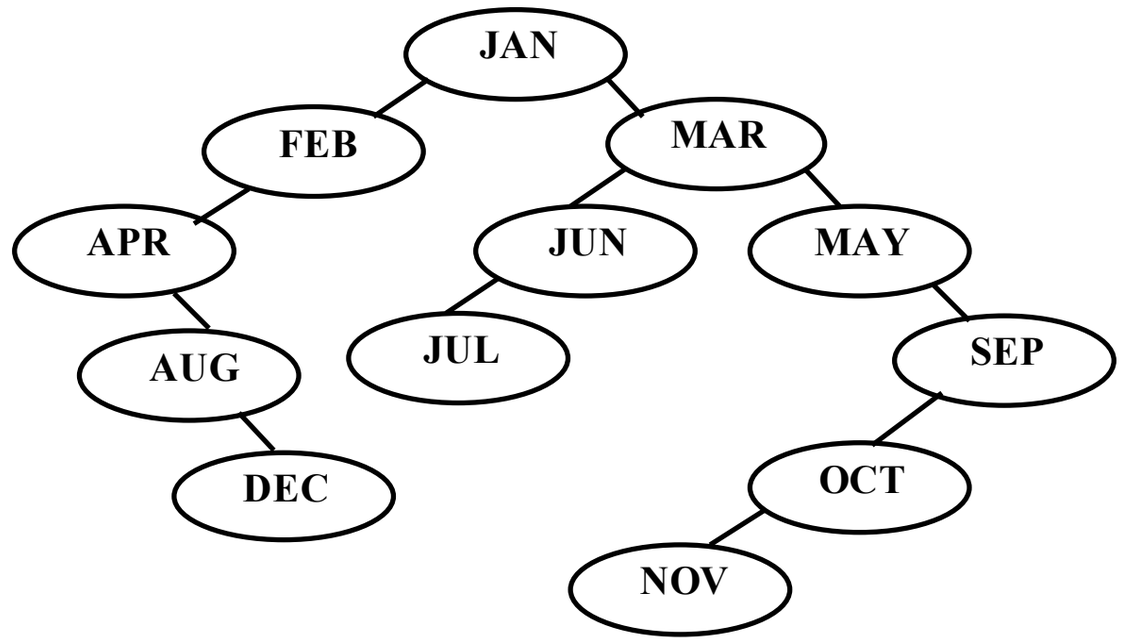


How many comparison do you need to search NOV?

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?

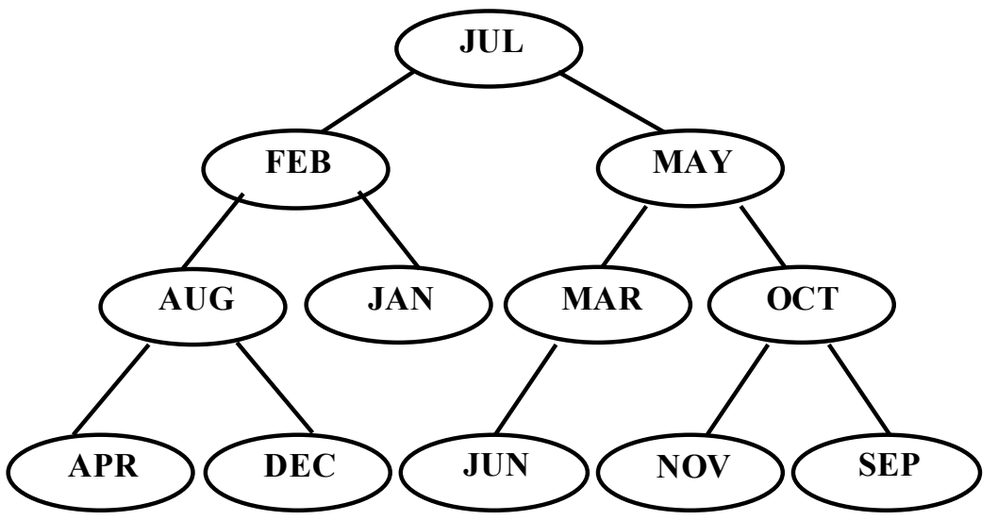
What is the average number of comparisons?

1 (JAN) + 2 (FEB) + 2 (MAR) + 3 (APR) + 3 (MAY) + 3 (JUN) +
4 (JUL) + 4 (AUG) + 4 (SEP) + 5 (OCT) + 6 (NOV) + 5 (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?

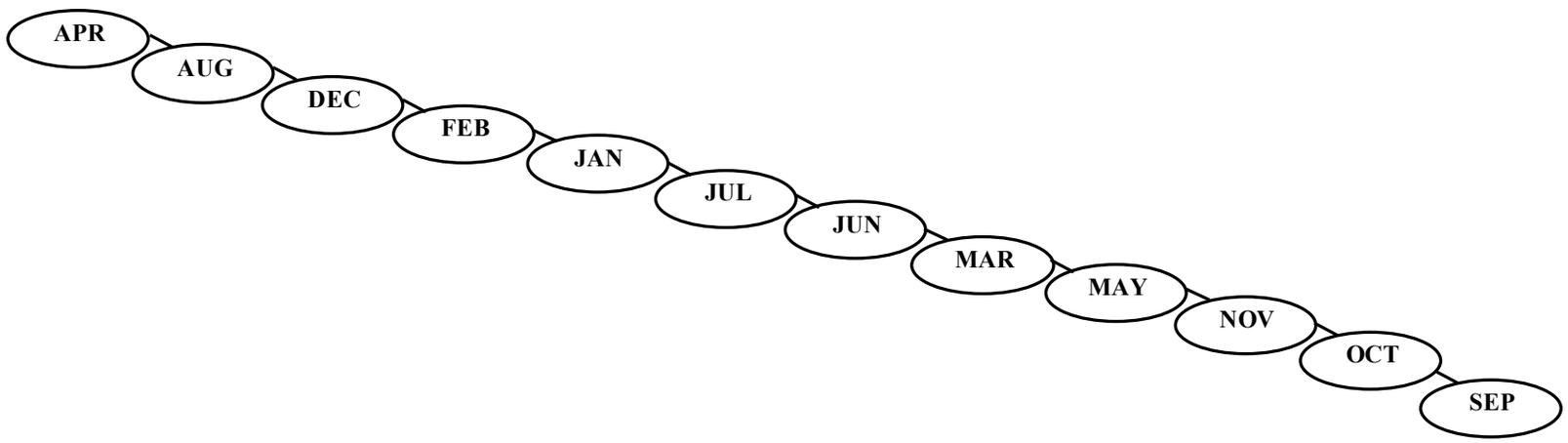
What is the average number of comparisons?

3 (JAN) + 2 (FEB) + 3 (MAR) + 4 (APR) + 2 (MAY) + 4 (JUN) +
1 (JUL) + 3 (AUG) + 4 (SEP) + 3 (OCT) + 4 (NOV) + 4 (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?

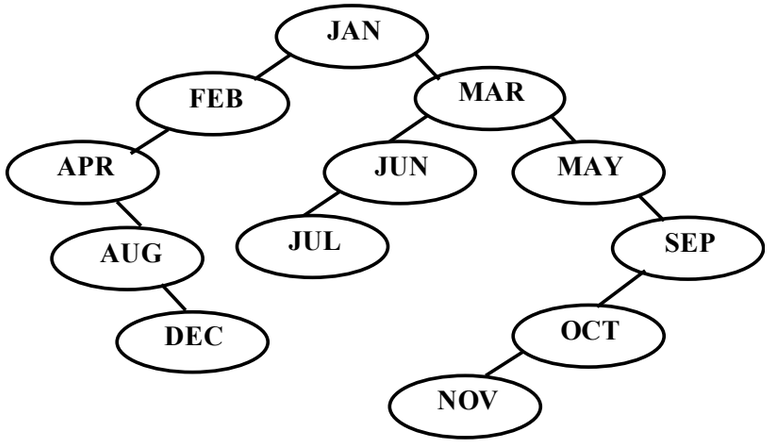
What is the average number of comparisons?

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

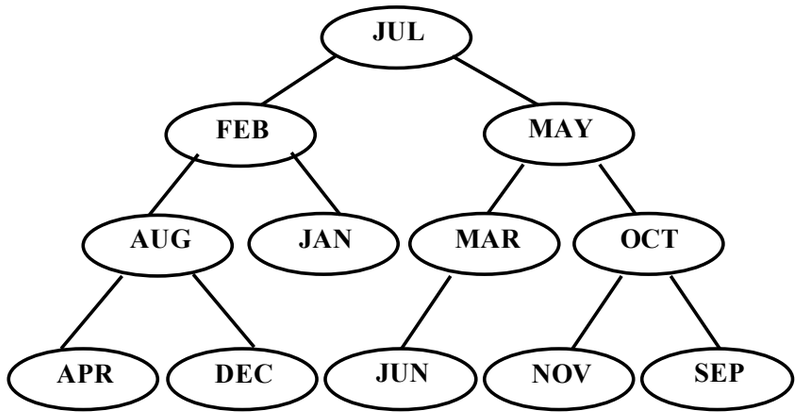
$78 / 12 = 6.5$

Binary Search Tree

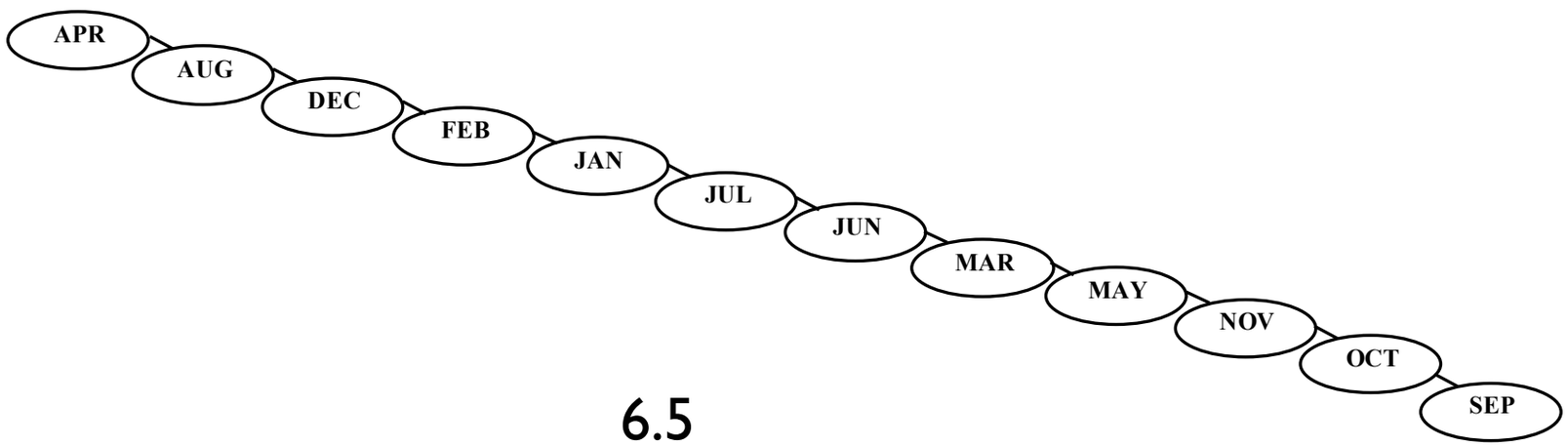
If equal probability, the average search and insertion time is O(logn)



3.5



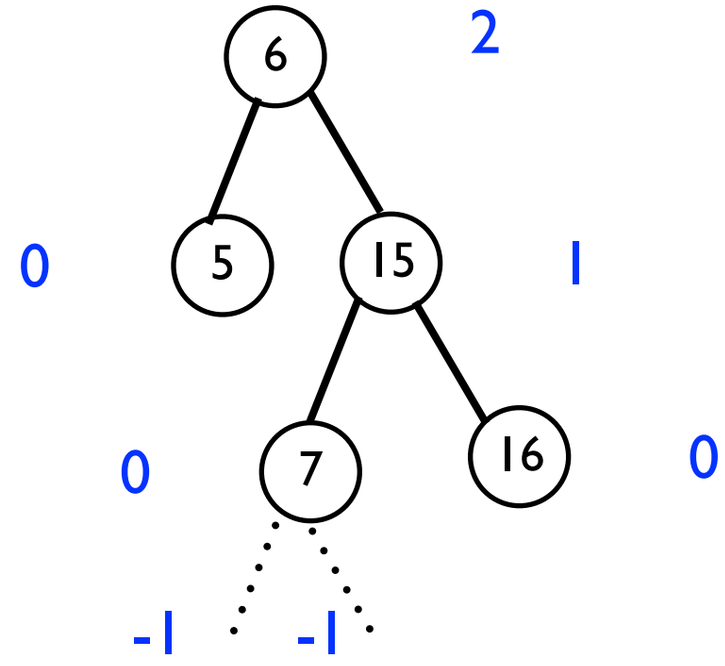
3.1



6.5

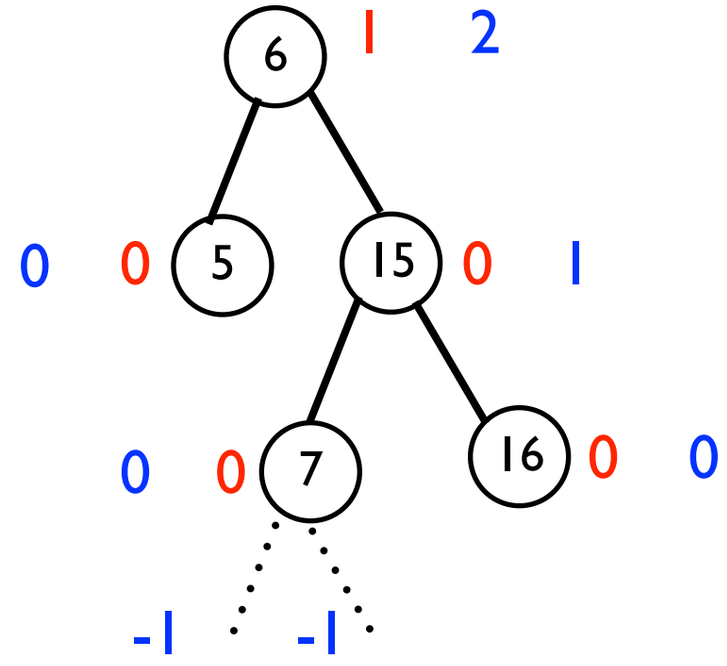
AVL (Adelson-Velskii and Landis) Tree

- 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



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

diff = $|h_L - h_R|$

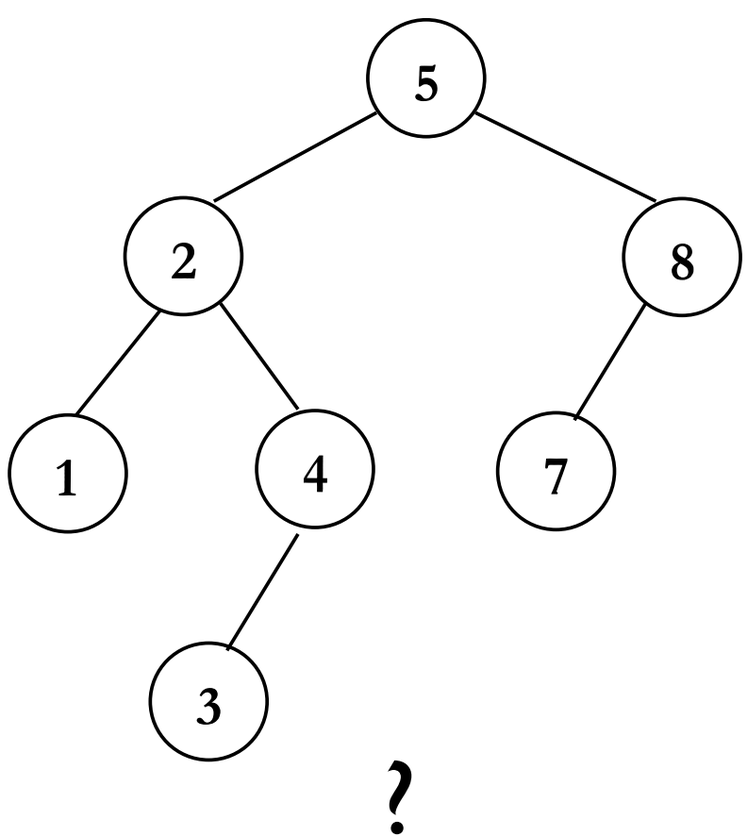
AVL (Adelson-Velskii and Landis) Tree

- 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

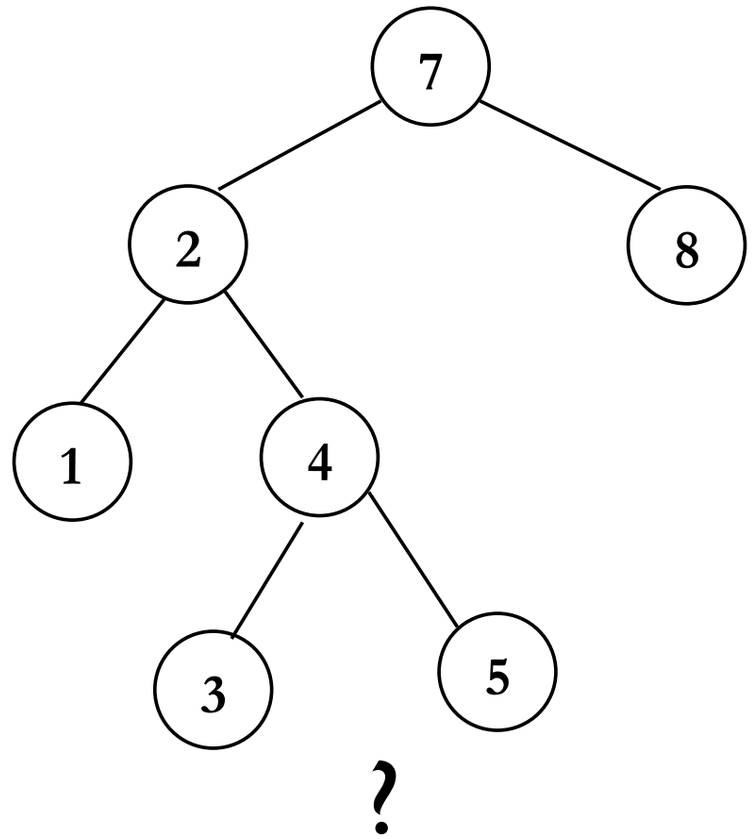
  (1) T_L and T_R are height-balanced and

  (2) $|h_L - h_R| <= 1$ where h_L and h_R are the heights of T_L and T_R

AVL (Adelson-Velskii and Landis) Tree

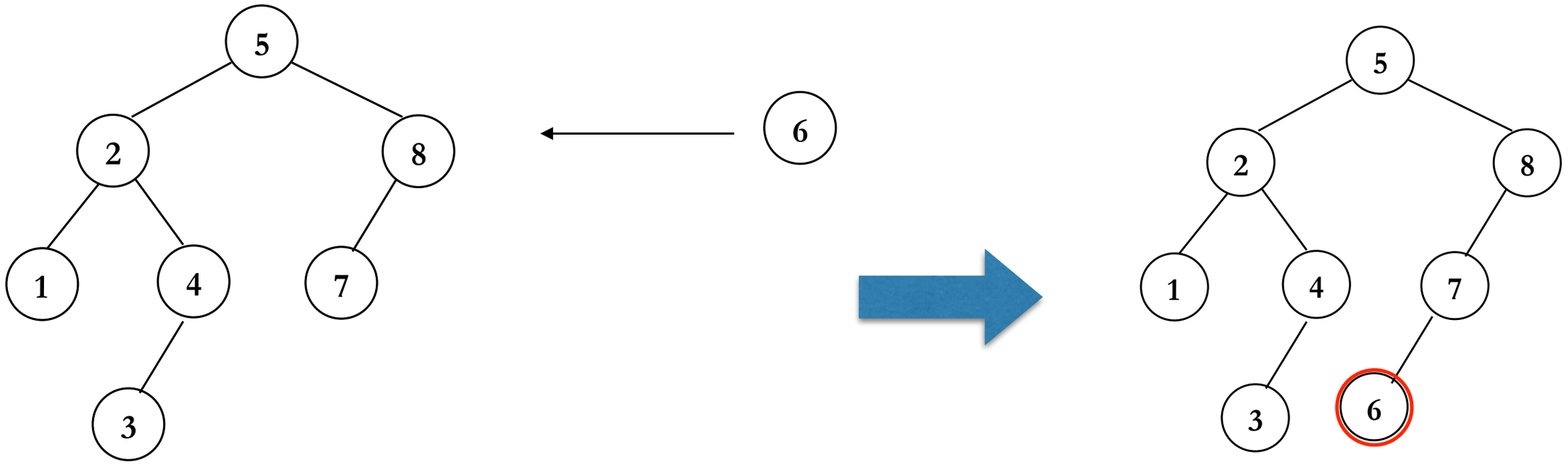


?

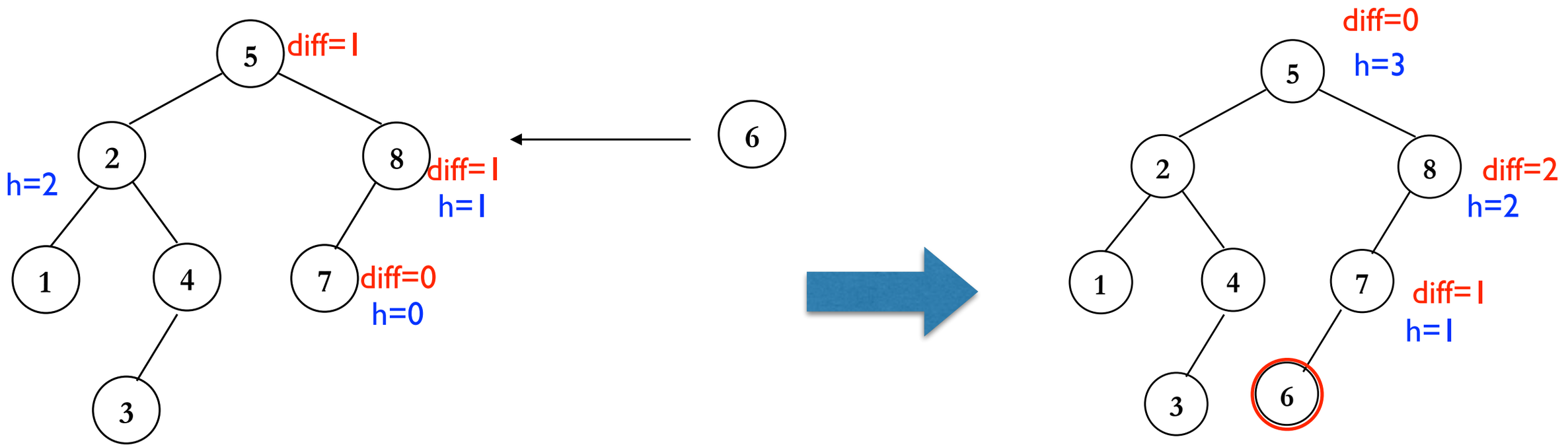


?

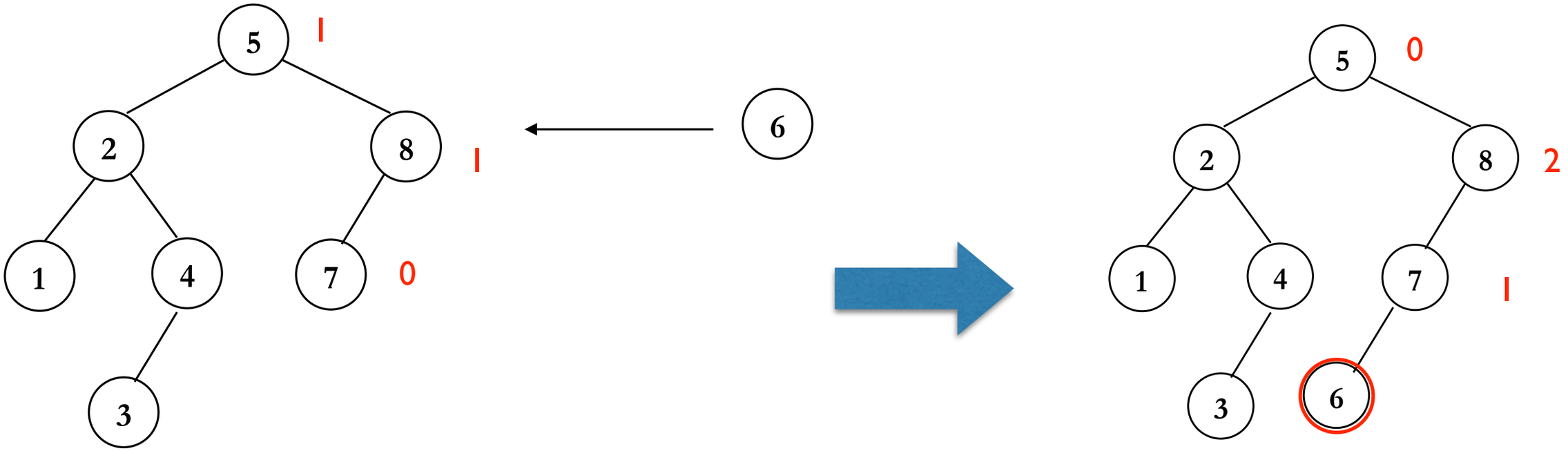
AVL Tree: insertion



AVL Tree: insertion



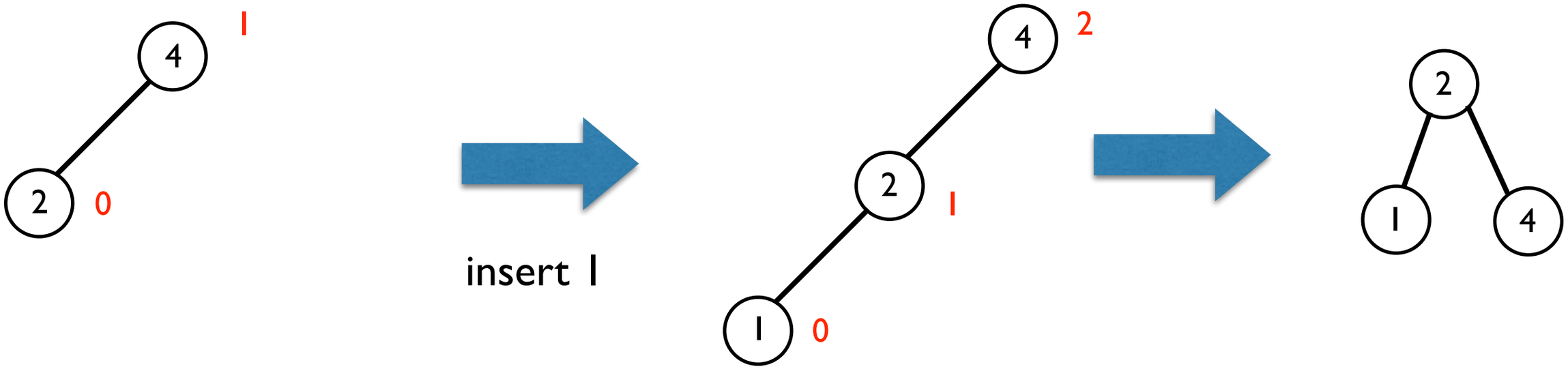
AVL Tree: insertion



- case 1: an insertion into the left subtree of the left child
  - single rotation

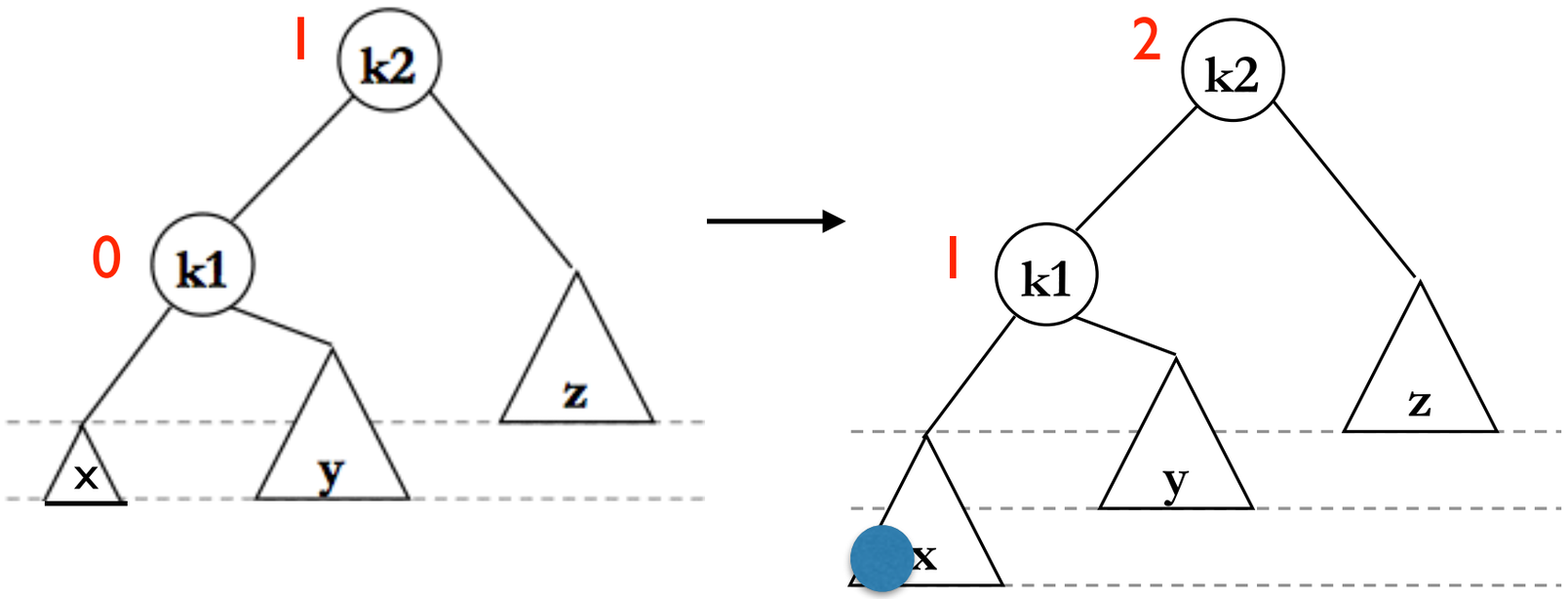
AVL Tree: insertion

- case 1: an insertion into the left subtree of the left child
  - single rotation



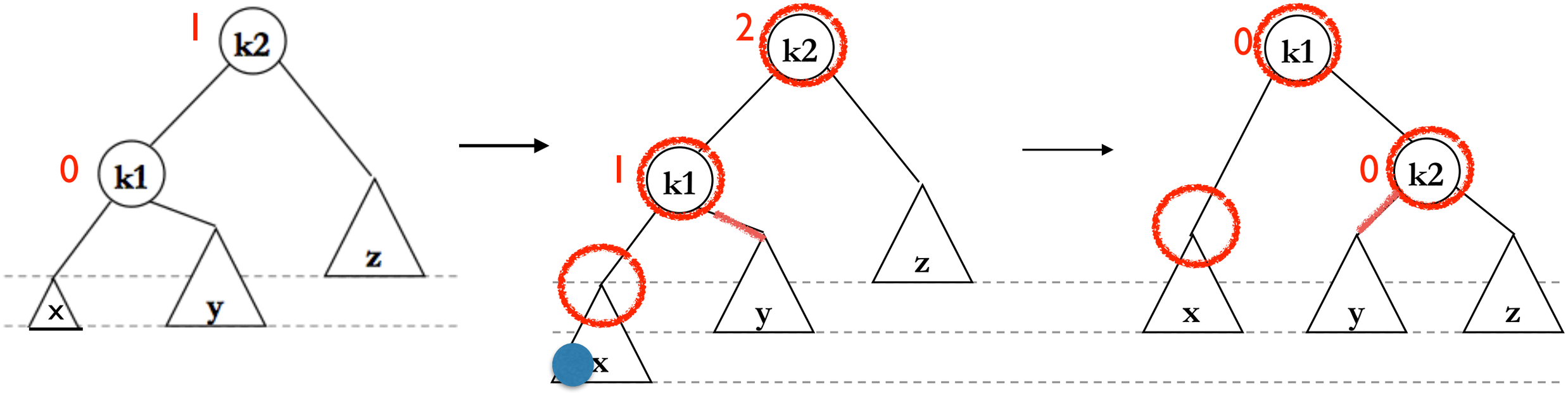
AVL Tree: insertion

- case 1: an insertion into the left subtree of the left child
  - single rotation

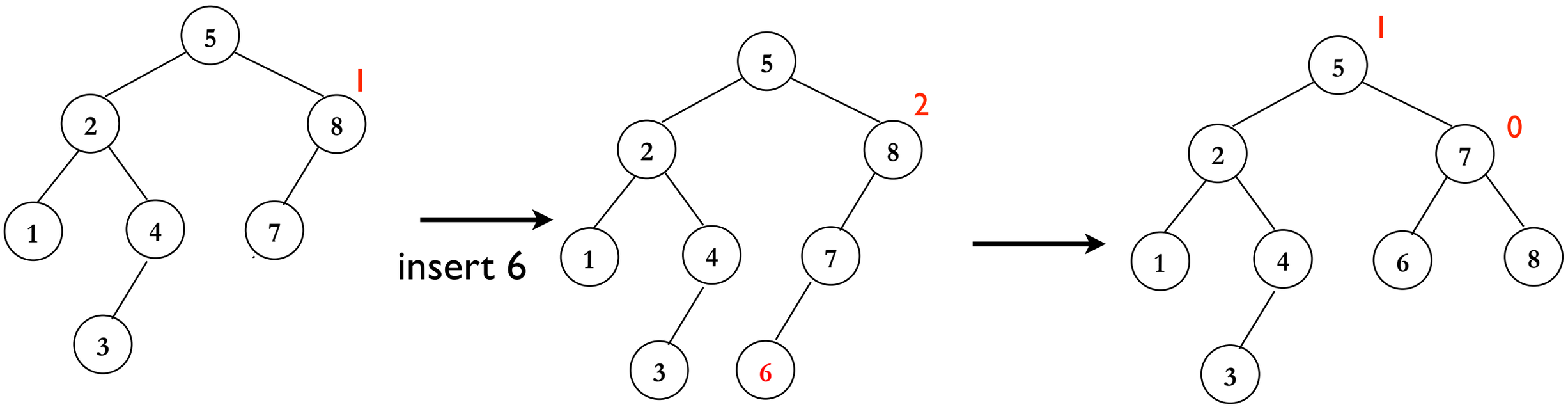


AVL Tree: insertion

- case 1: an insertion into the left subtree of the left child
  - single rotation

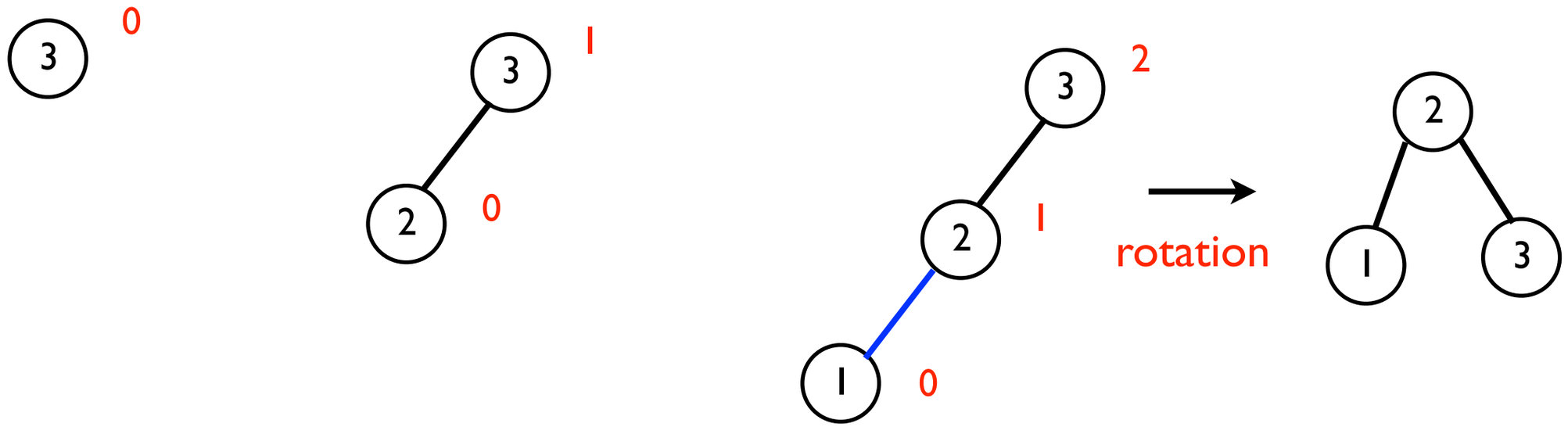


AVL Tree: insertion



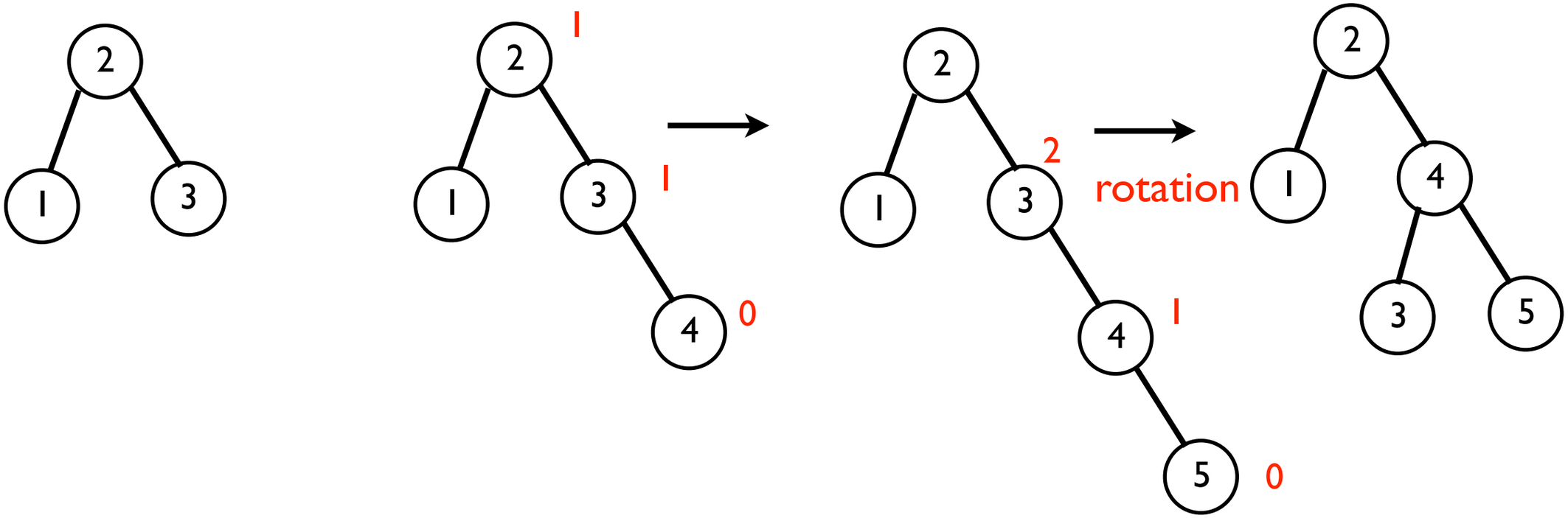
AVL Tree: insertion

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



AVL Tree: insertion

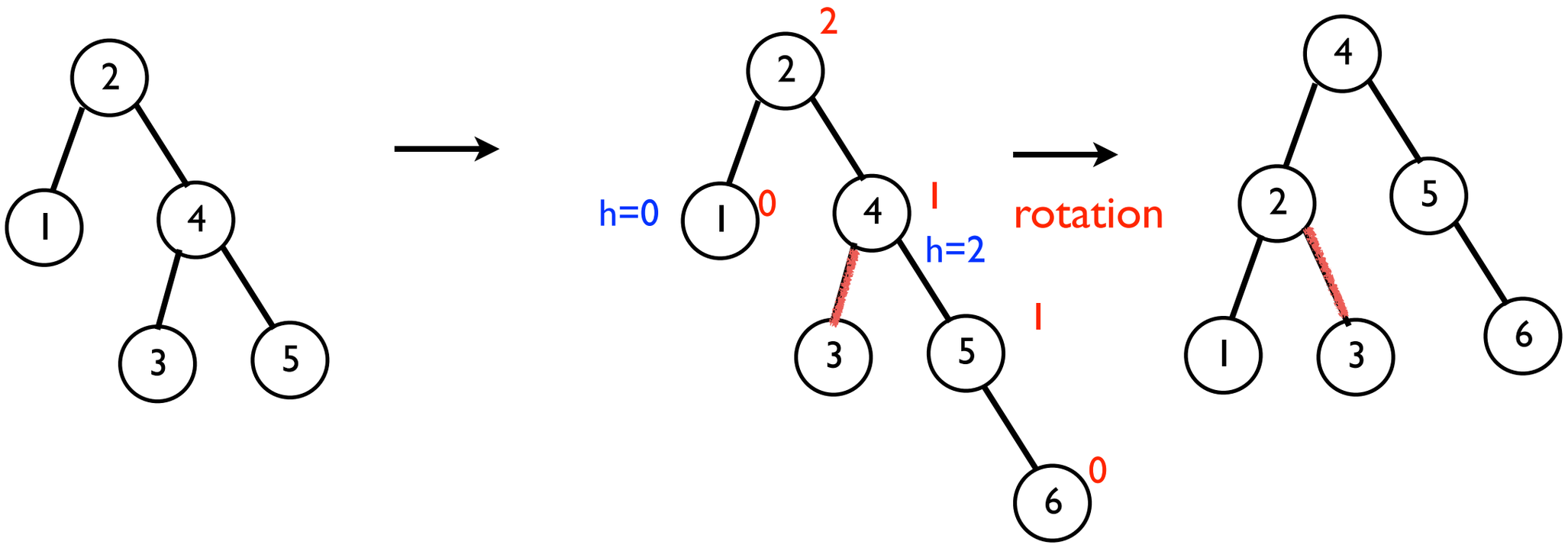
insert 3, 2, 1, 4, 5, 6



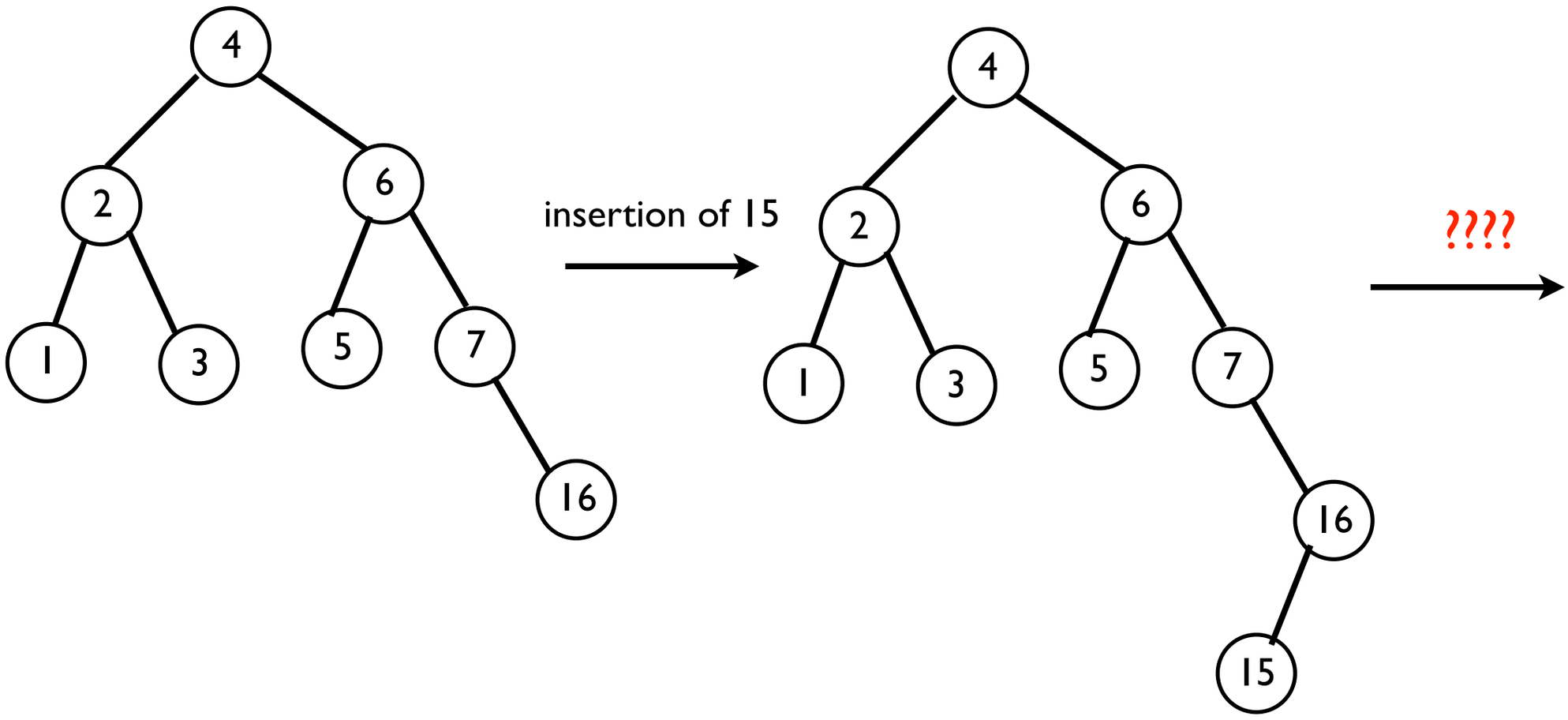
- case 2: an insertion into the right subtree of the right child of the node A
  - single rotation

AVL Tree: insertion

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

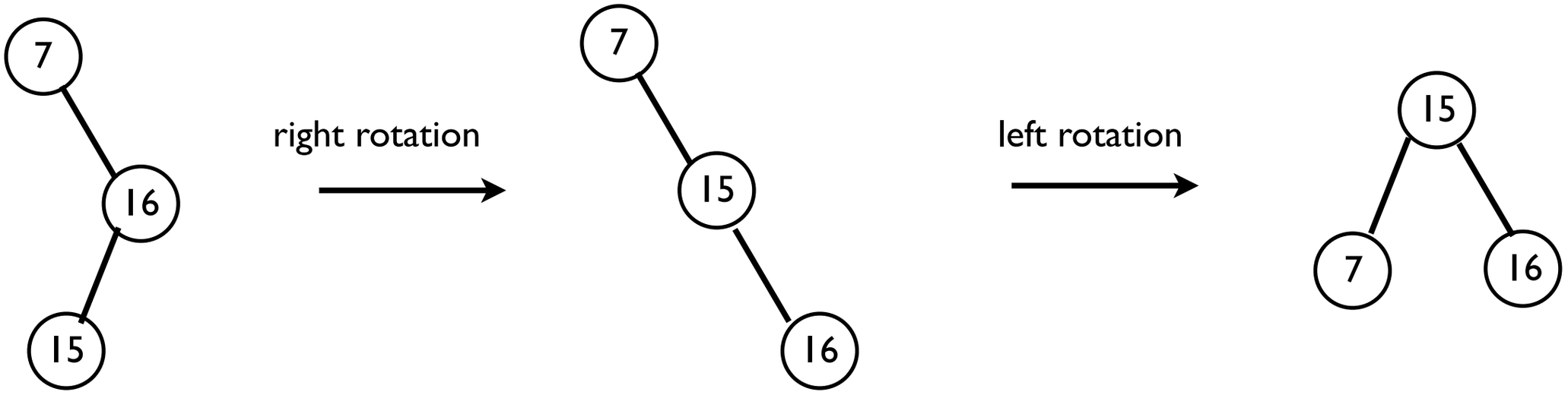


AVL Tree: insertion



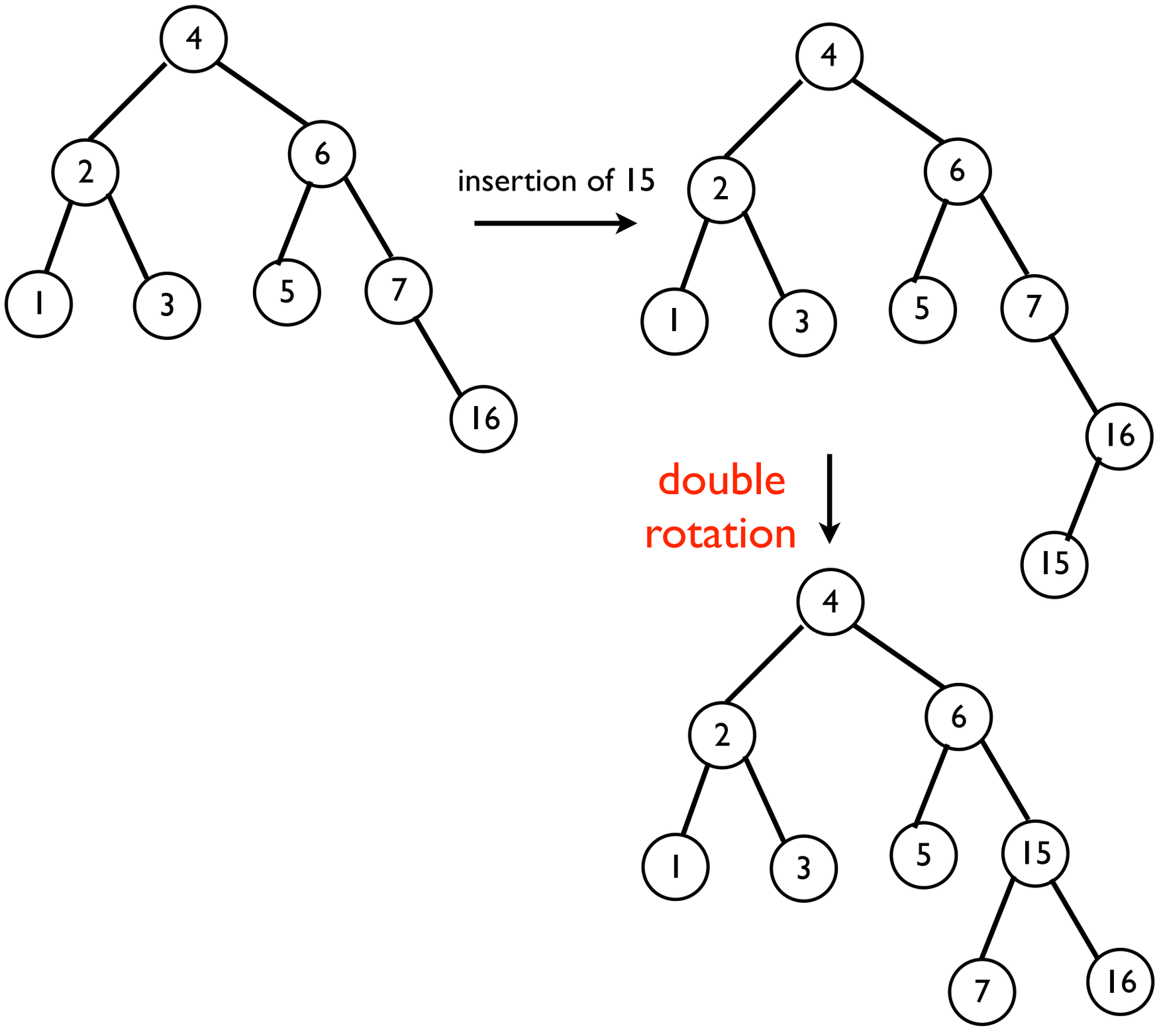
AVL Tree: insertion

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



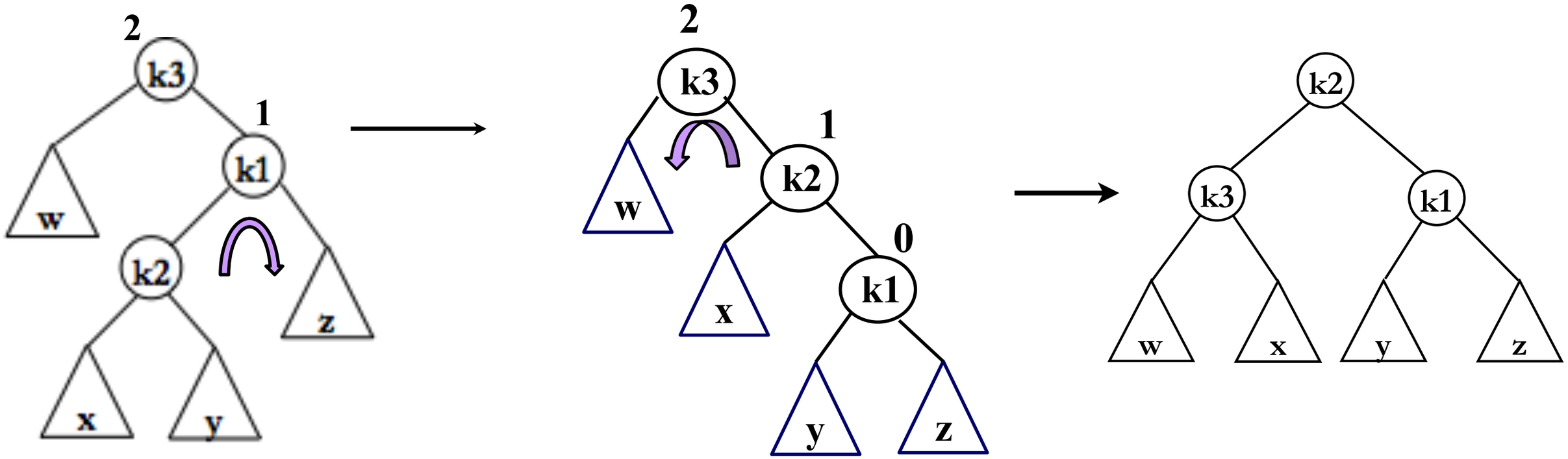
balanced

AVL Tree: insertion



AVL Tree: insertion

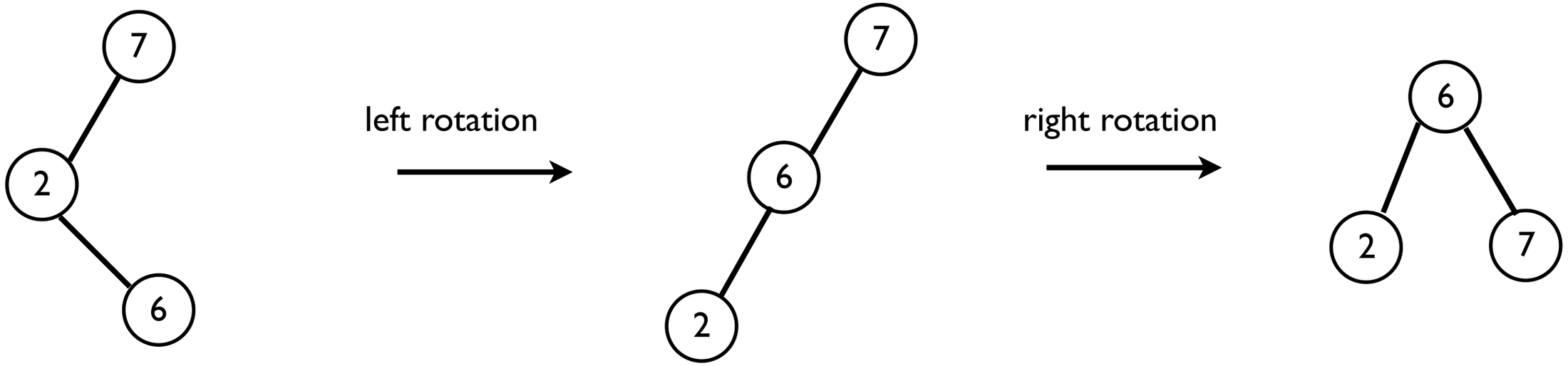
- case 3: an insertion into the left subtree of the right child of the unbalanced node
  - double rotation



right-left double notation

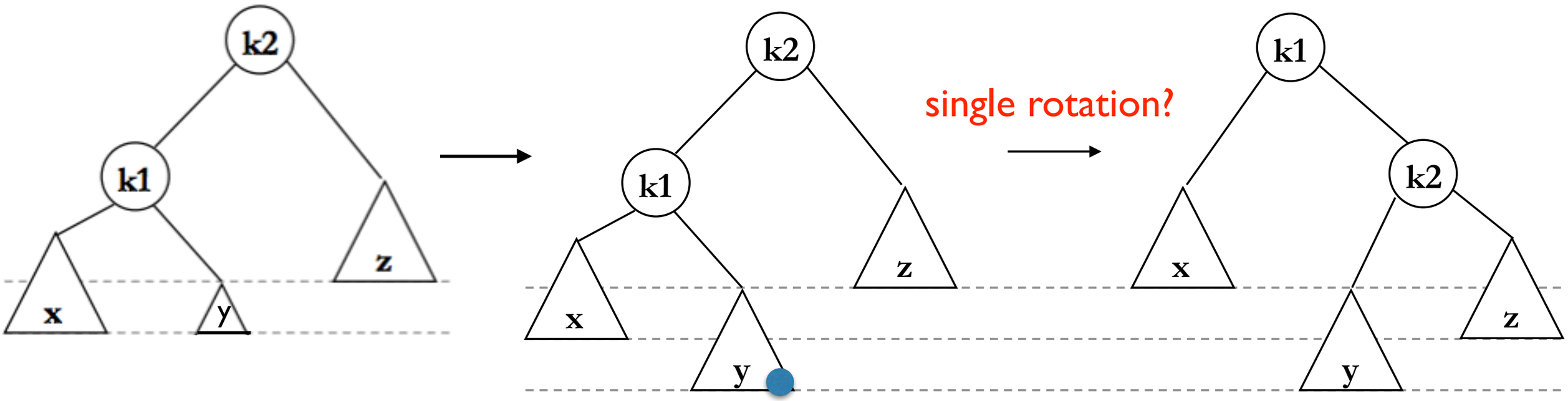
AVL Tree: insertion

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



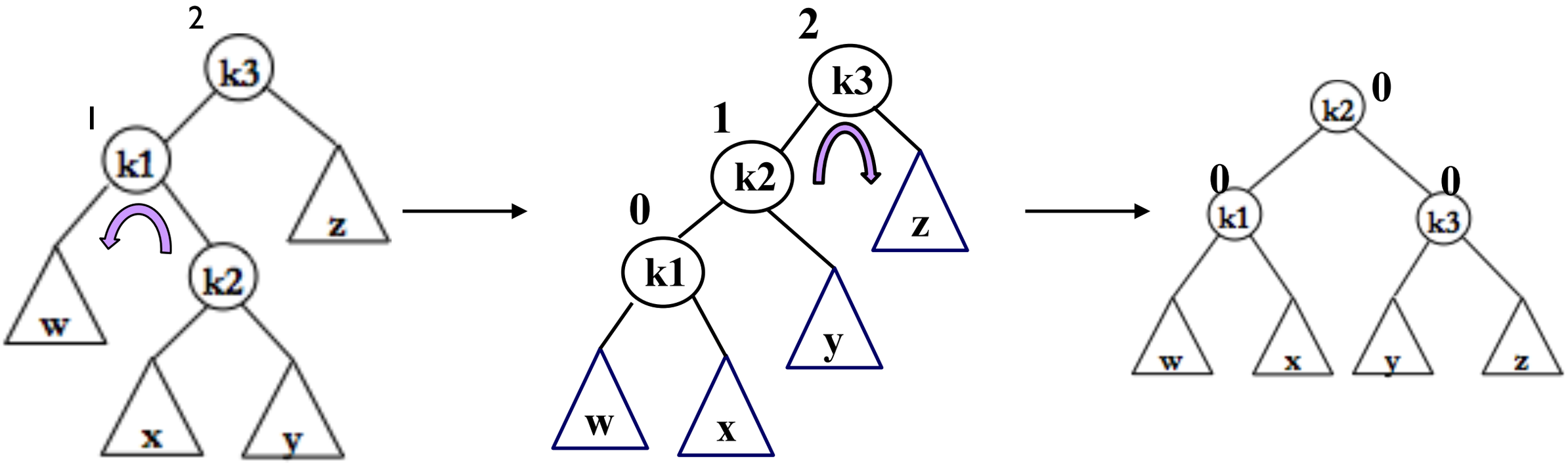
AVL Tree: insertion

- 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



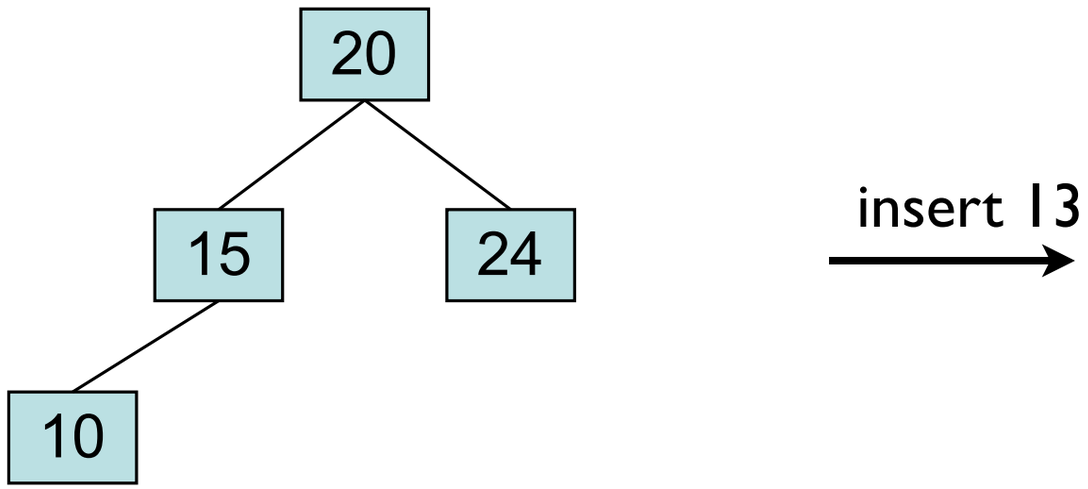
Left-right double notation

AVL Tree: insertion

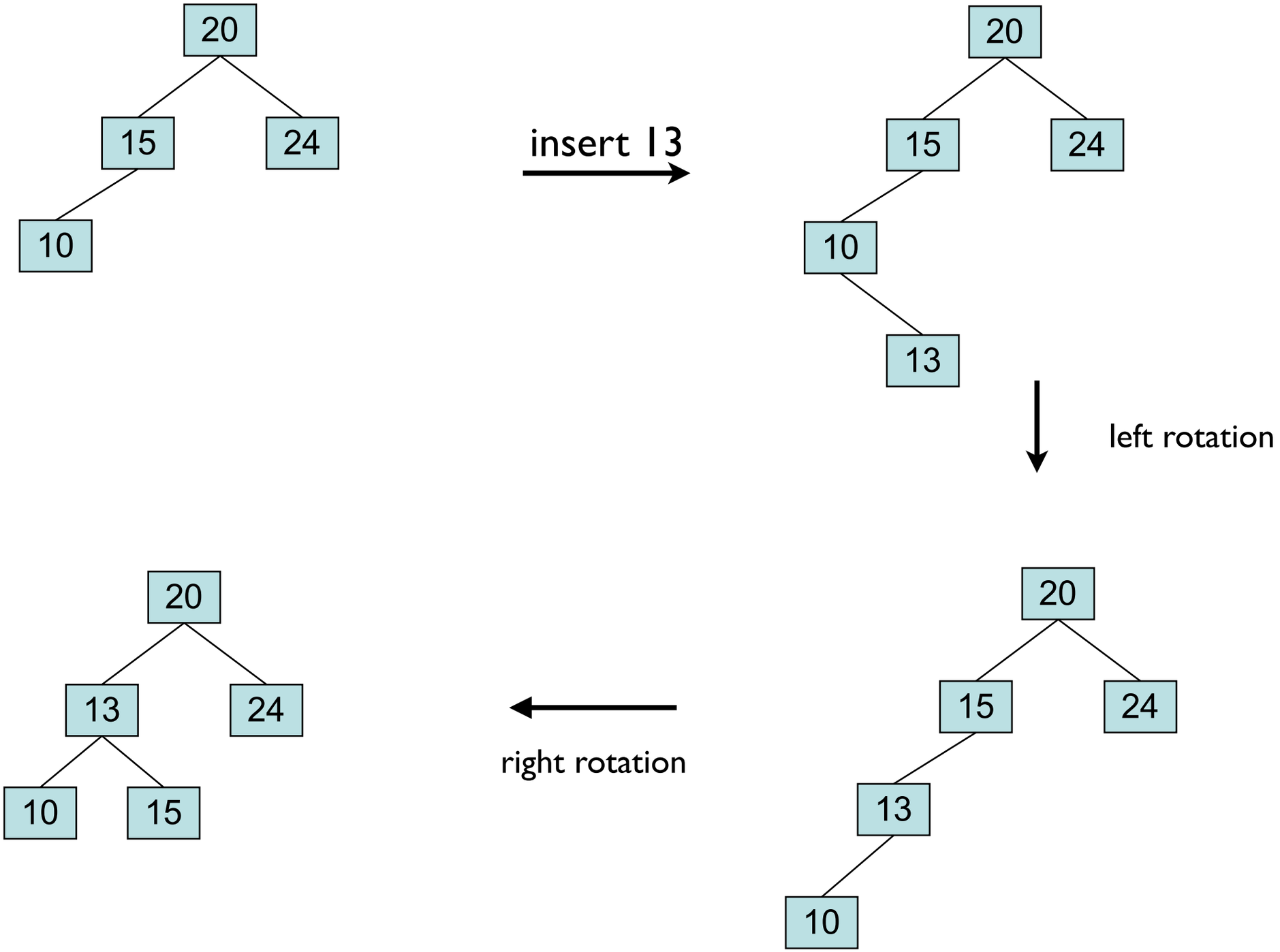
The node "A" ($|h_L - h_R|>1$) needs to be rebalanced, when

- 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

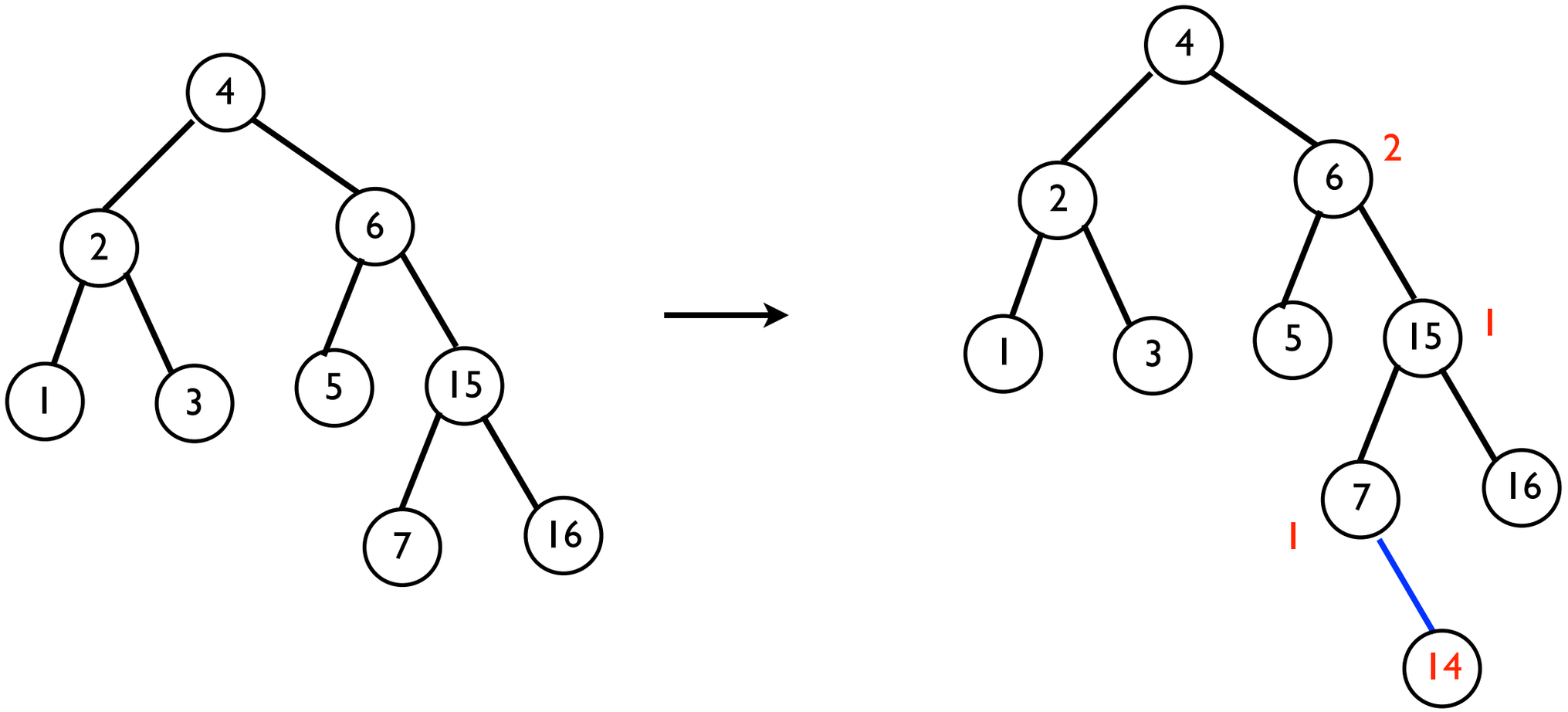


AVL Tree: insertion

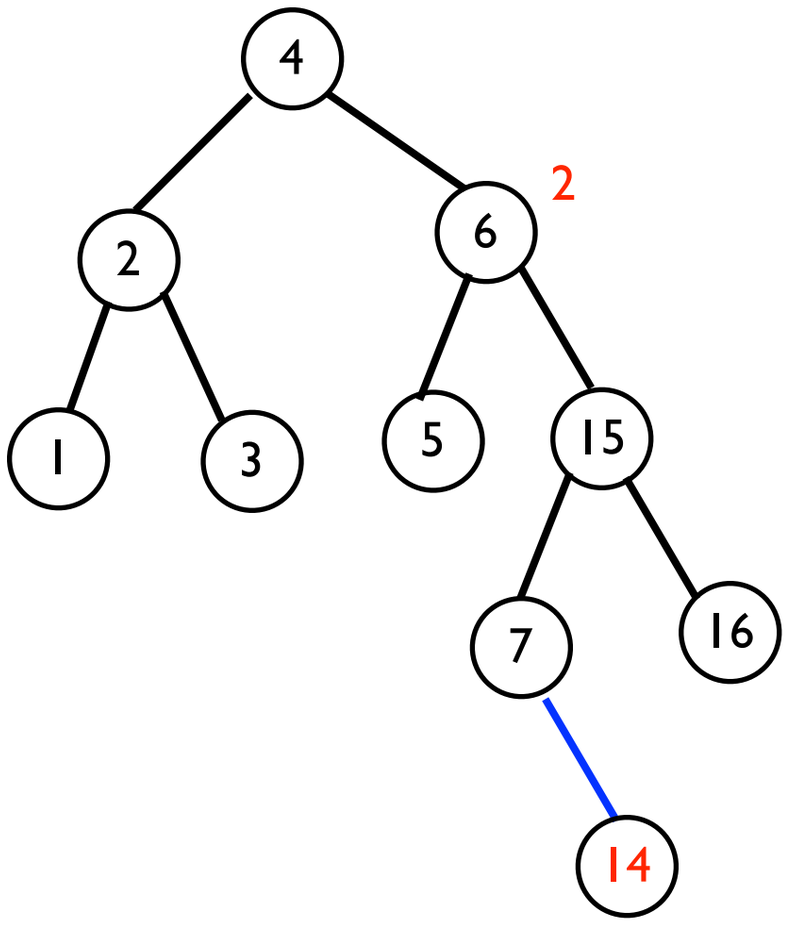


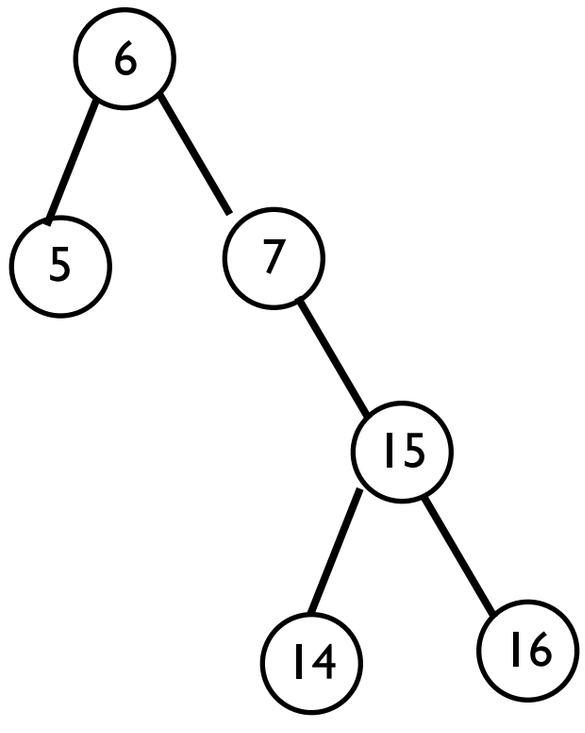
AVL Tree: insertion

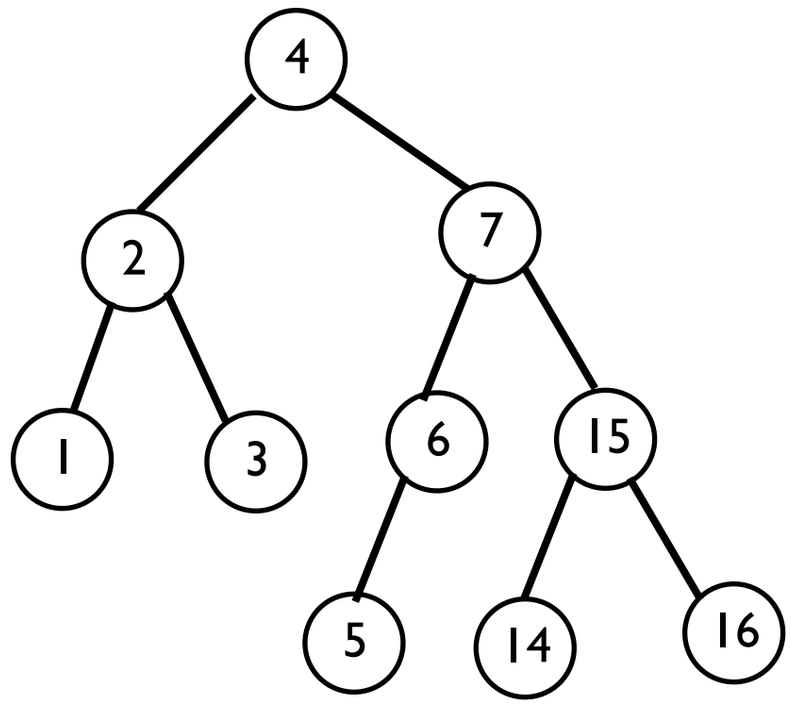
insert 14



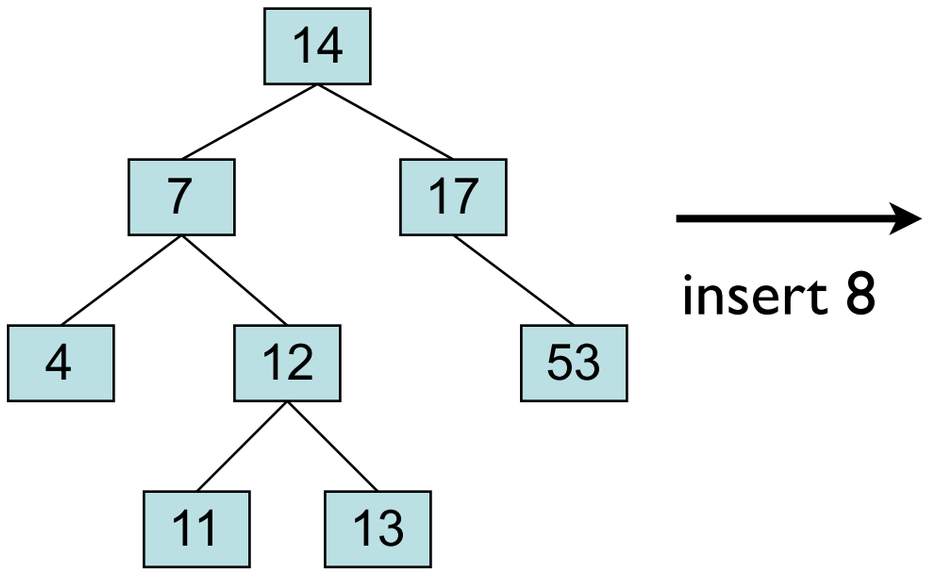
AVL Tree: insertion



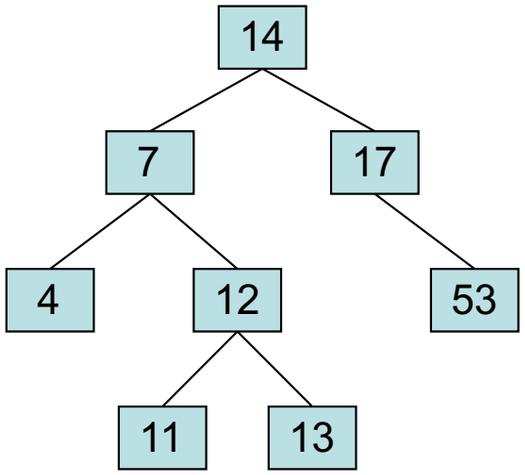




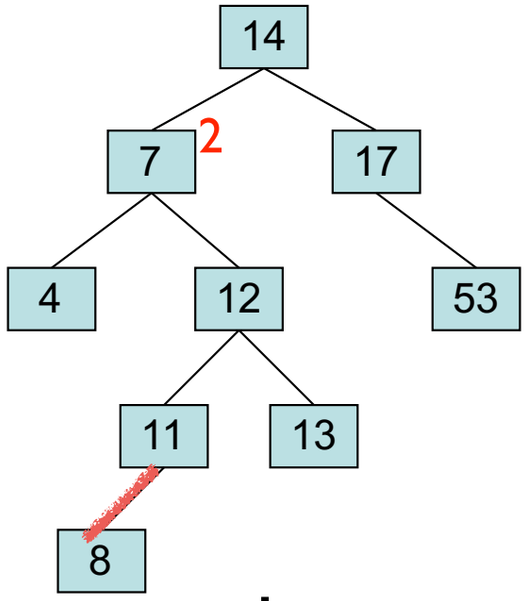
AVL Tree: insertion



AVL Tree: insertion

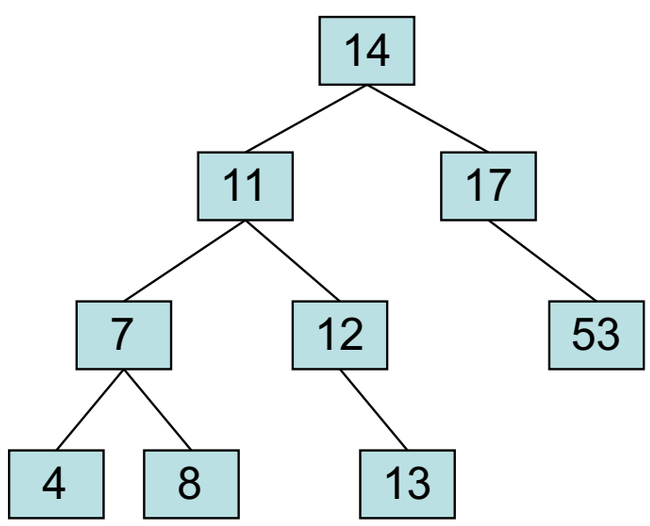


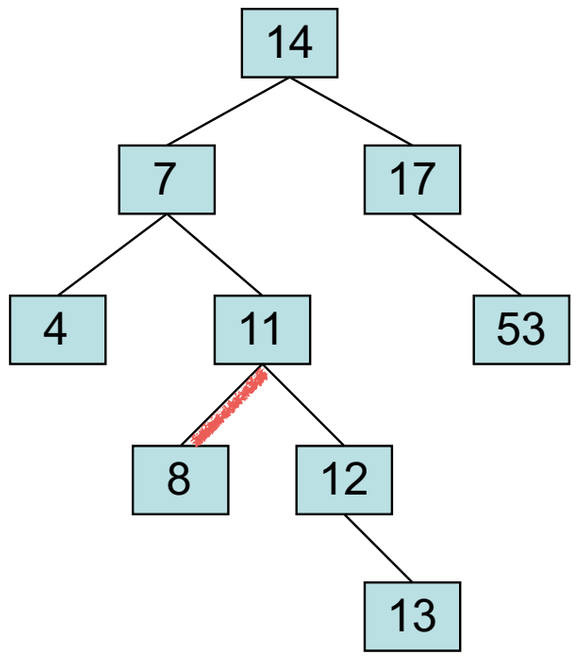
insert 8



right rotation

left rotation





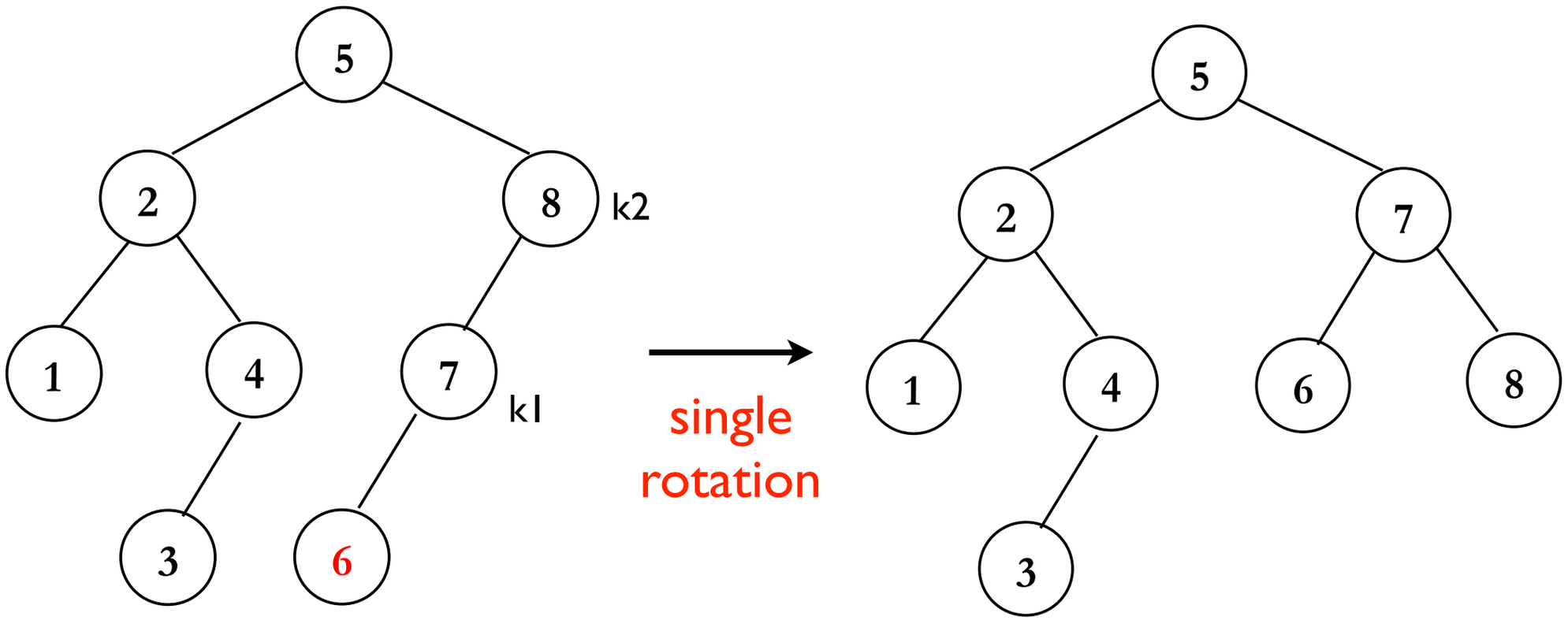
AVL Tree: exercise

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

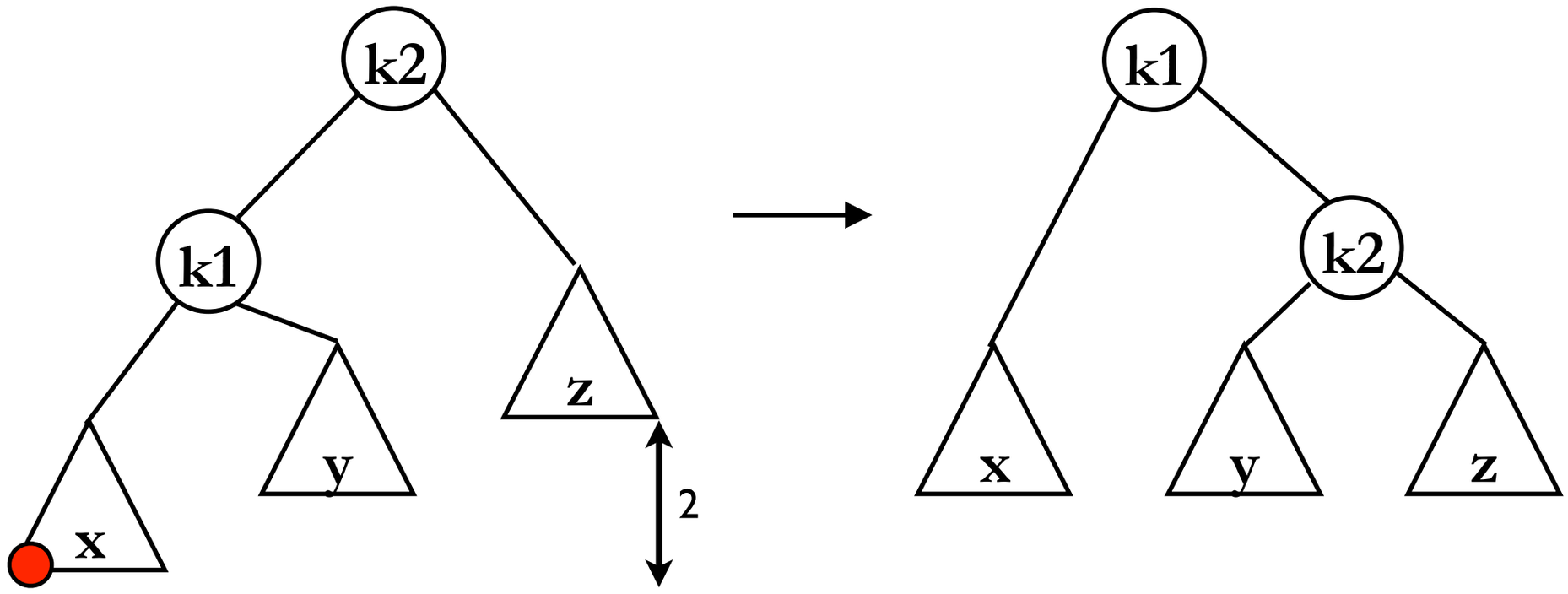
```
struct AVLNode;
typedef struct AVLNode *Position;
typedef struct AVLNode *AVLTree;

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

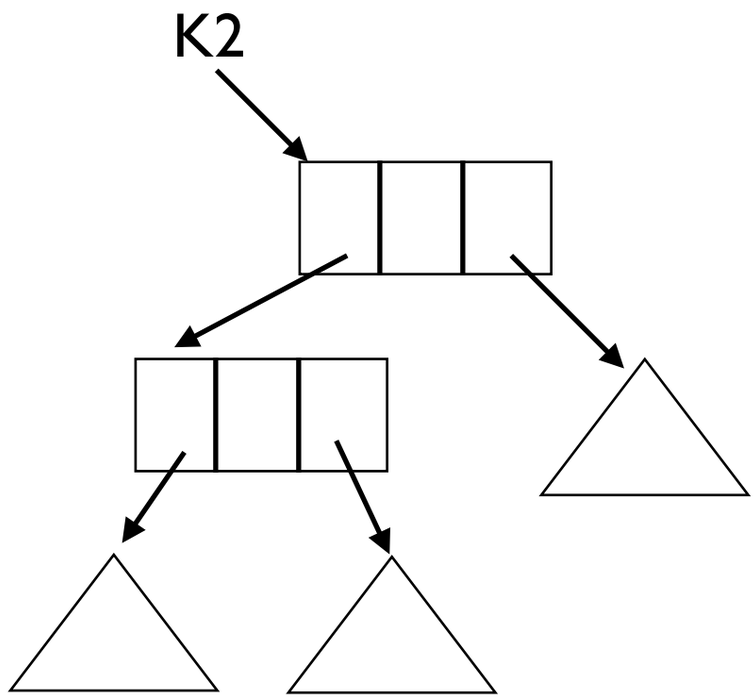


K2->Left = K1->Right;

K1->Right = K2;

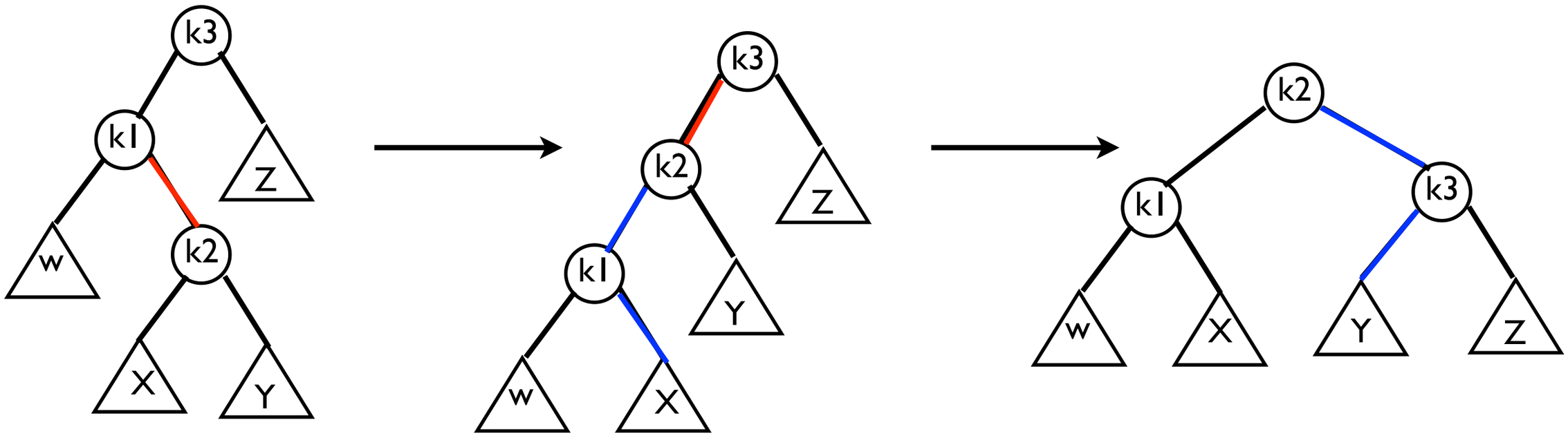
AVL Tree: insertion

```
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

```
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

```
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;
}
```