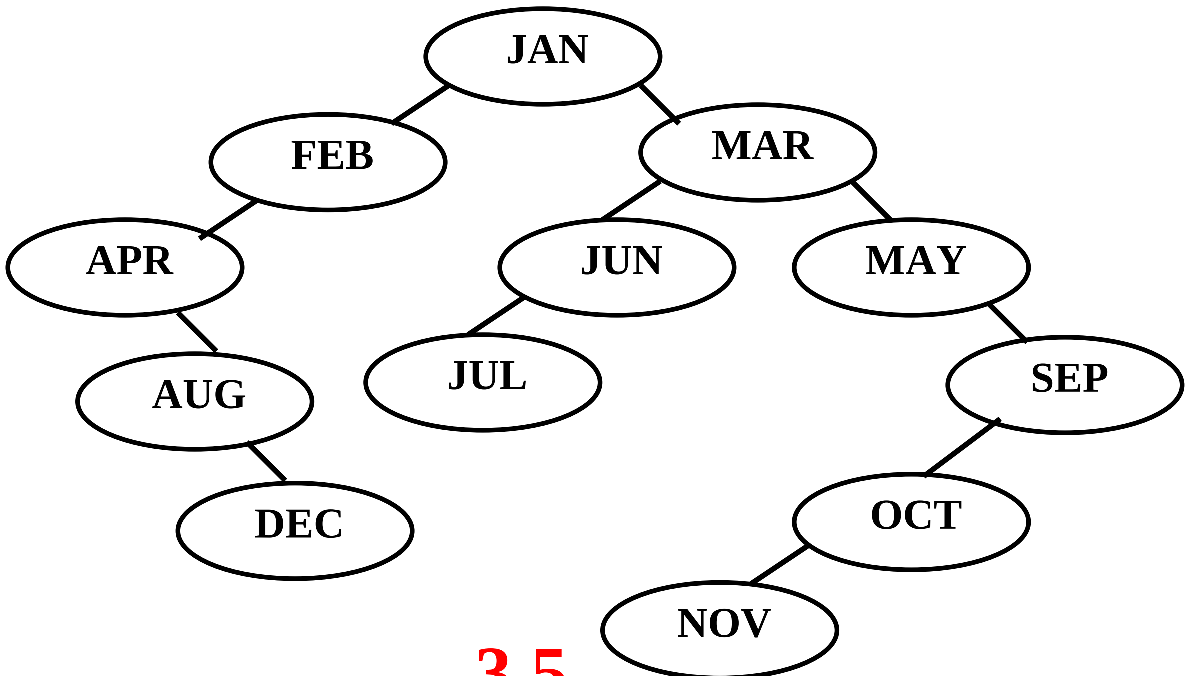
Data Structure:

Lecture 8. AVL Tree

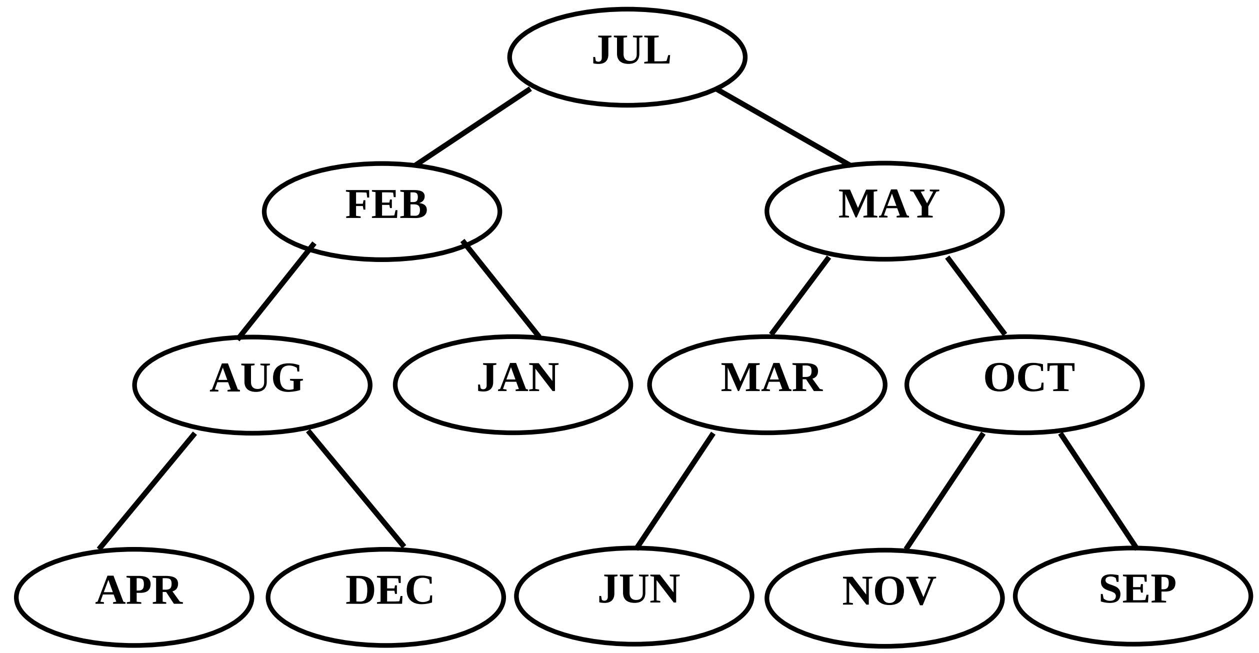
Binary Search Tree

- Average number of comparisons needed to search for any key



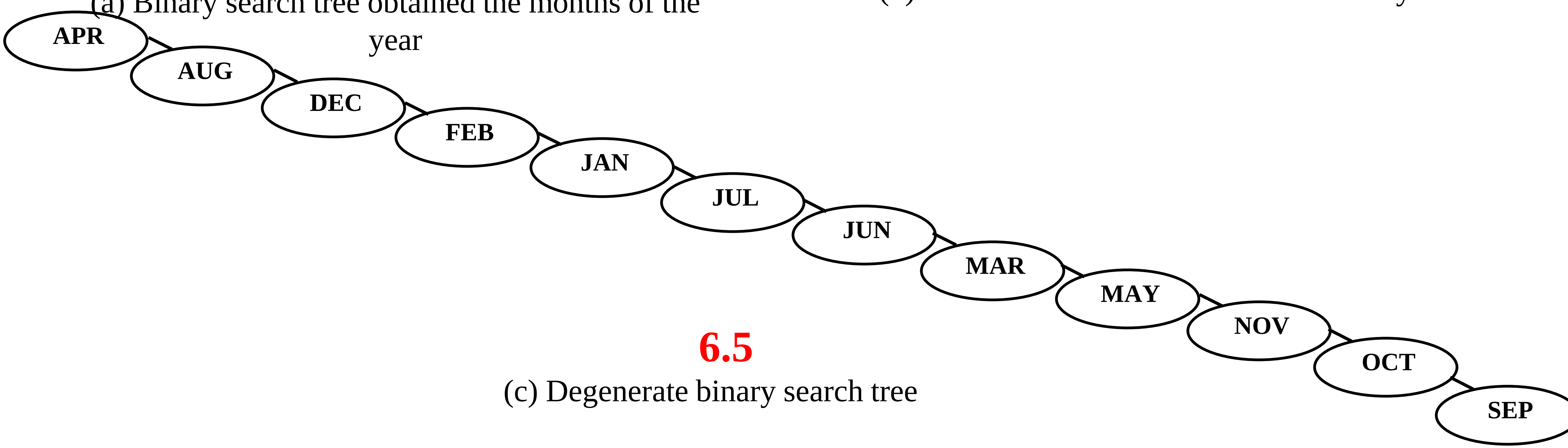
3.5

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



3.1

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

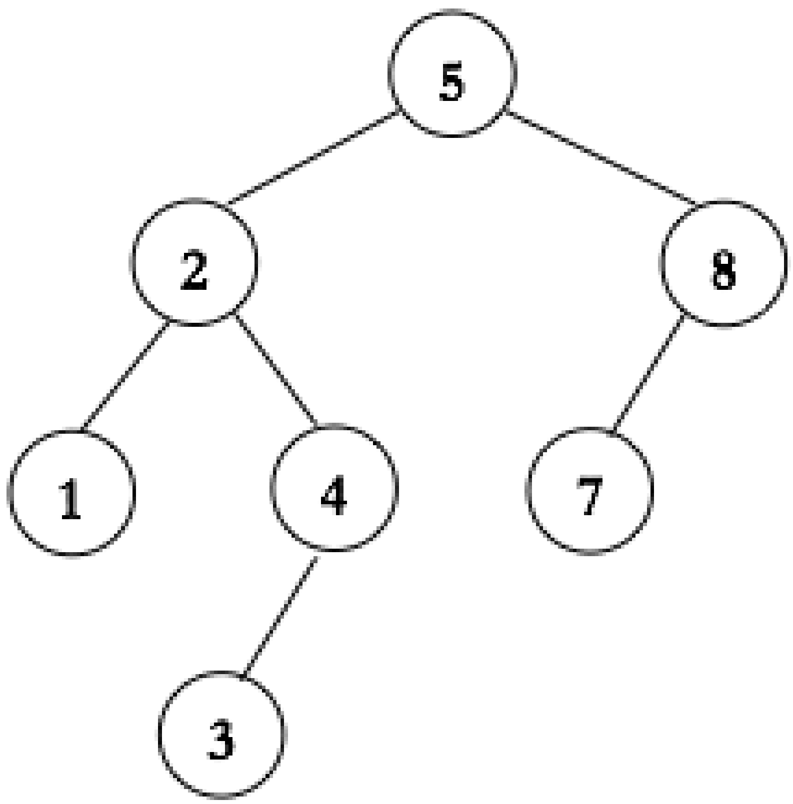


6.5

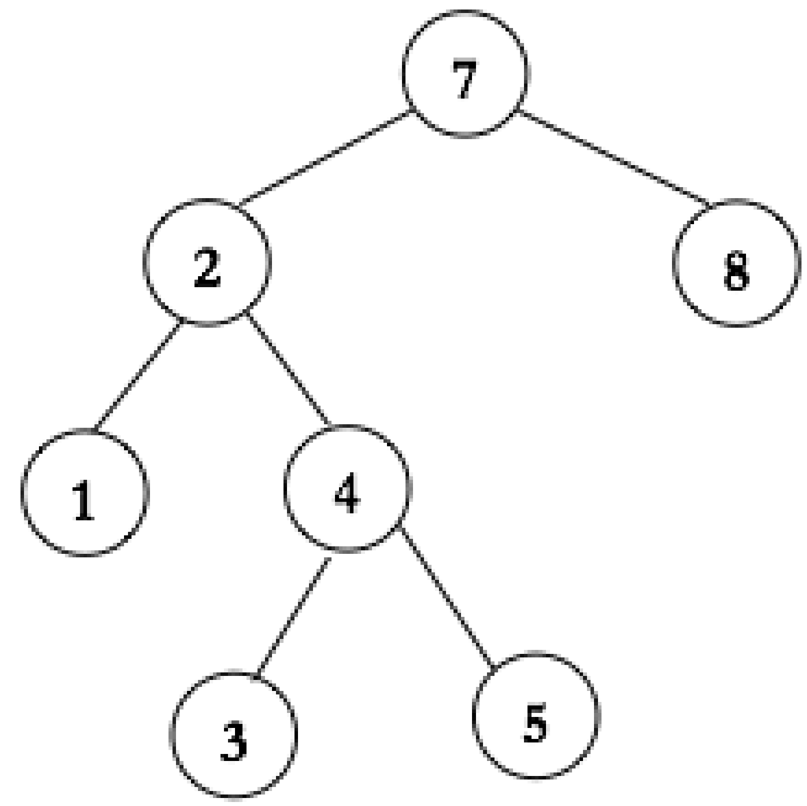
(c) Degenerate binary search tree

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)



?



?

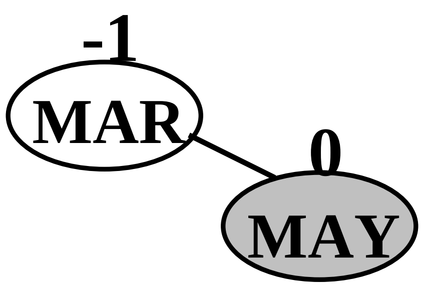
AVL tree

- Balance factor
  - the difference in heights between the left and right subtrees of that node
  - $BF(T) = h_L - h_R$
  - $BF(T) = -1, 0, 1$ for any node T

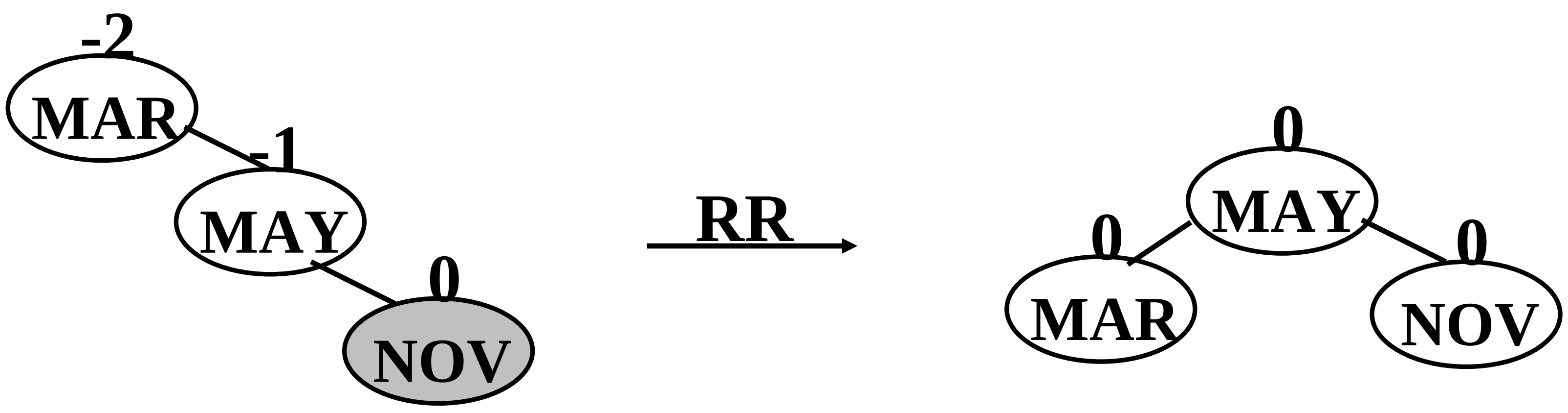
Height-balanced tree (1/5)

- Insertion in following order
  - MAR, MAY, NOV, AUG, APR, JAN, DEC, JUL, FEB, JUN OCT, SEP

(a) insert MARCH

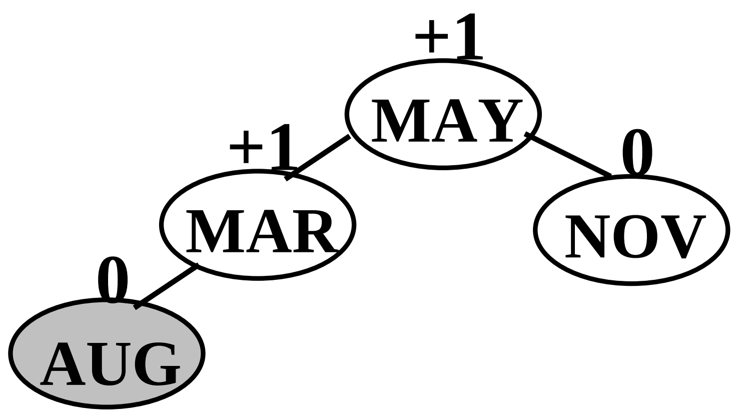


(b) insert MAY

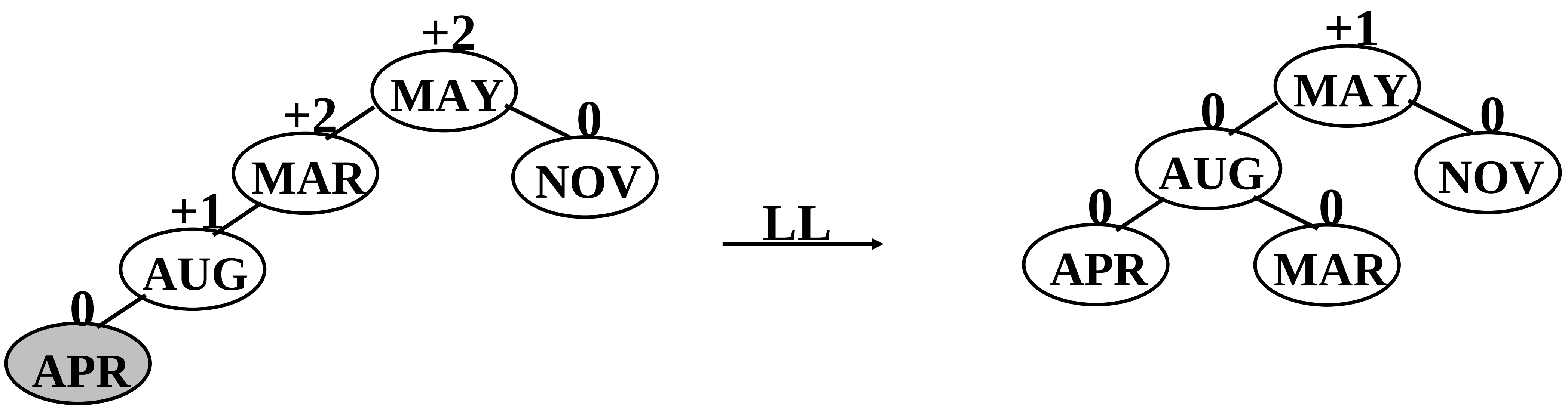


(c) insert NOV

Balanced tree(2/5)

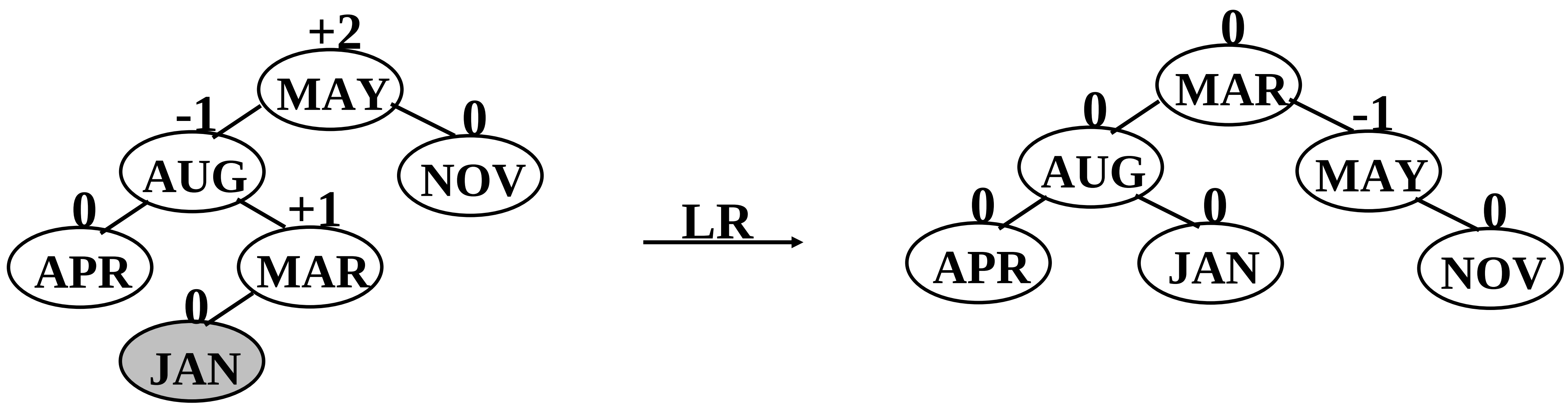


(d) insert AUGUST

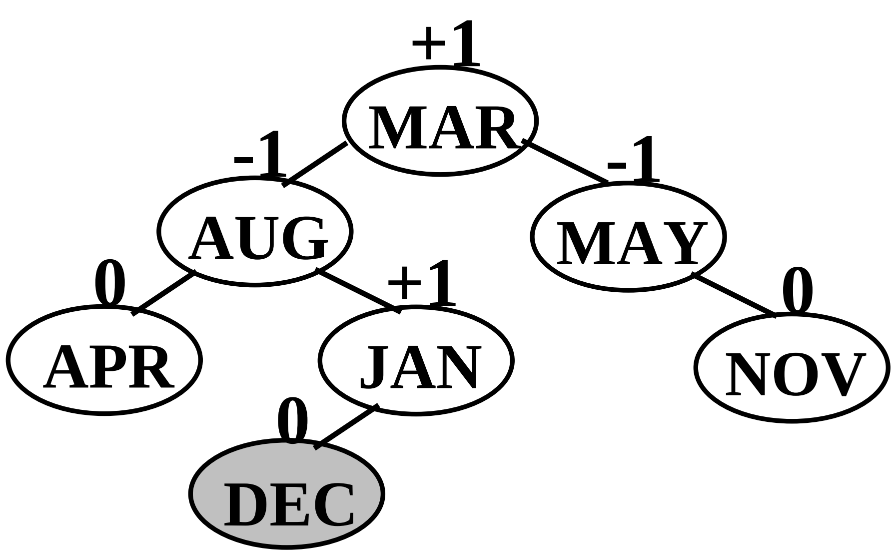


(e) insert APRIL

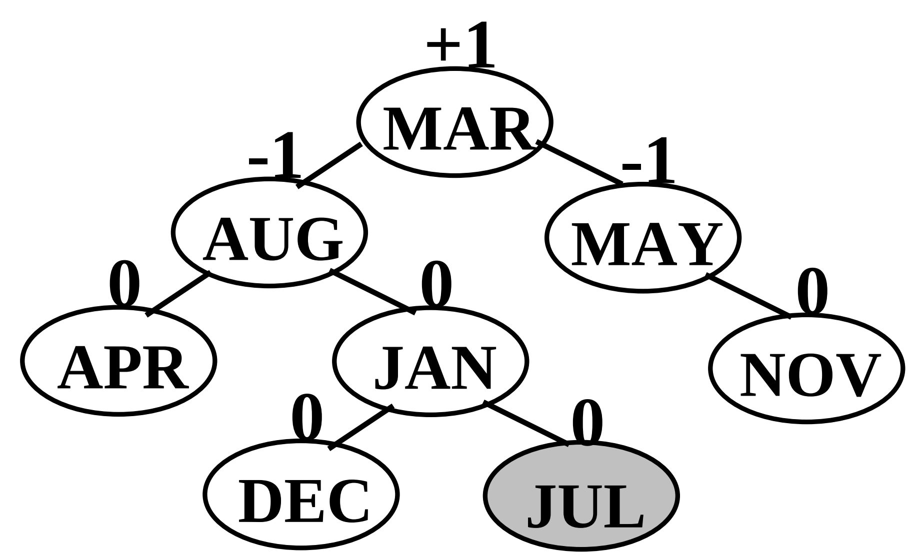
Balanced tree(3/5)



(f) insert JANUARY

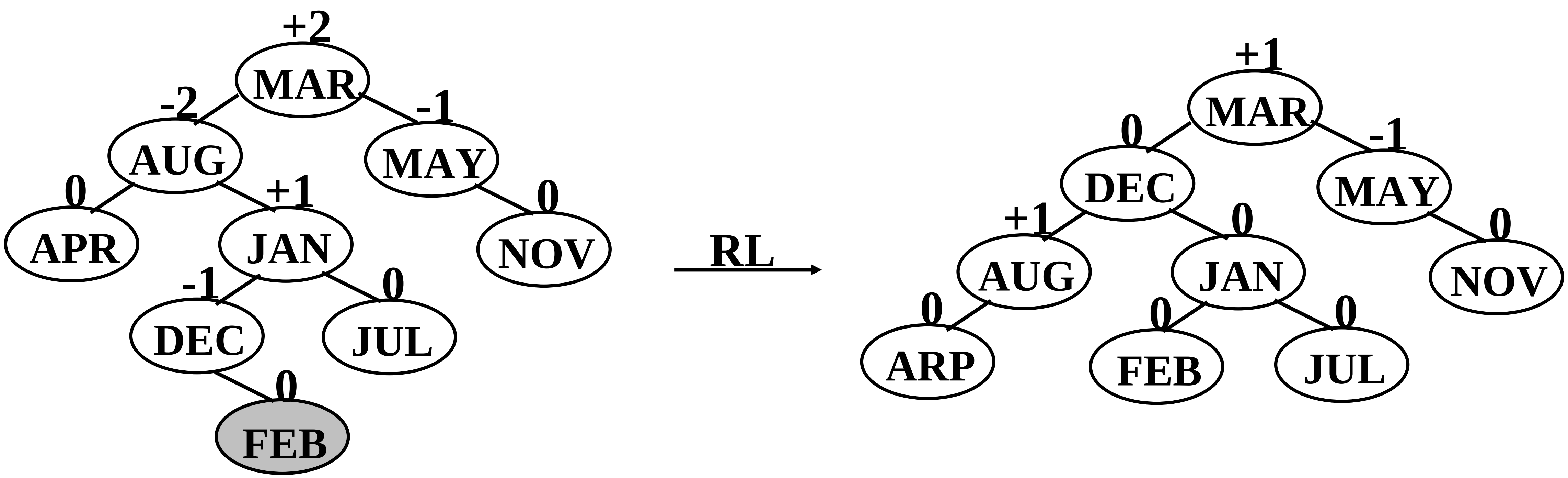


(g) insert DECEMBER

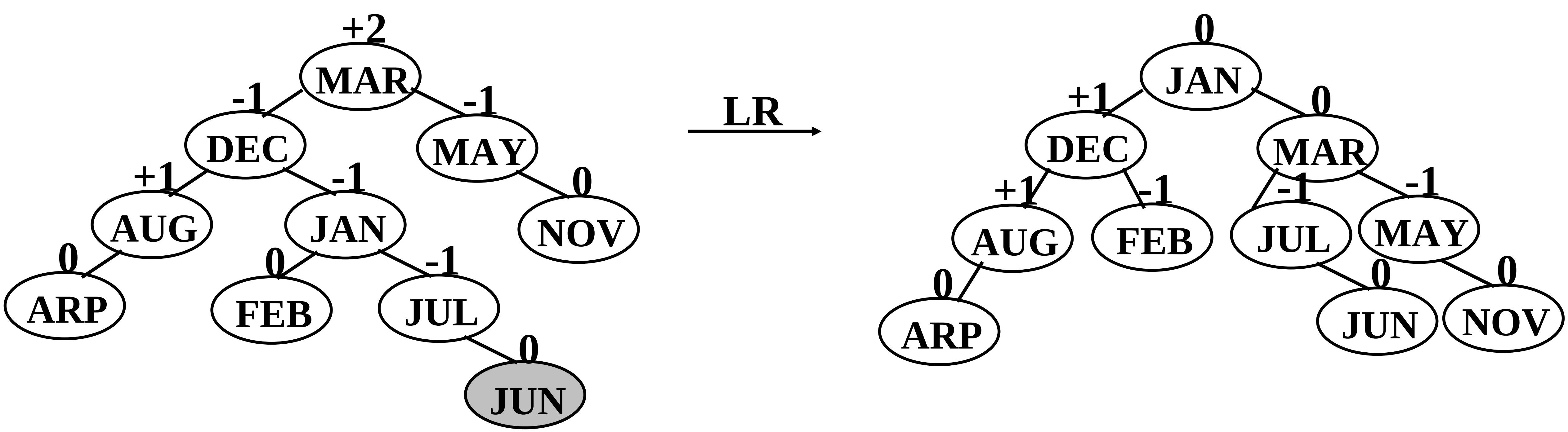


(h) insert JULY

Balanced tree(4/5)

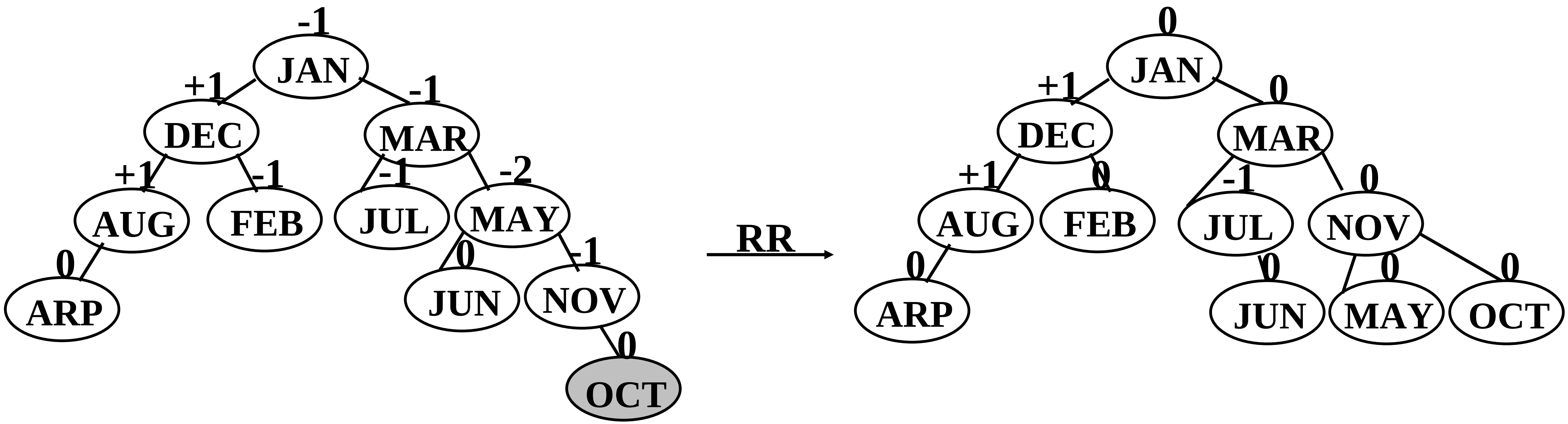


(i) insert FEBRUARY

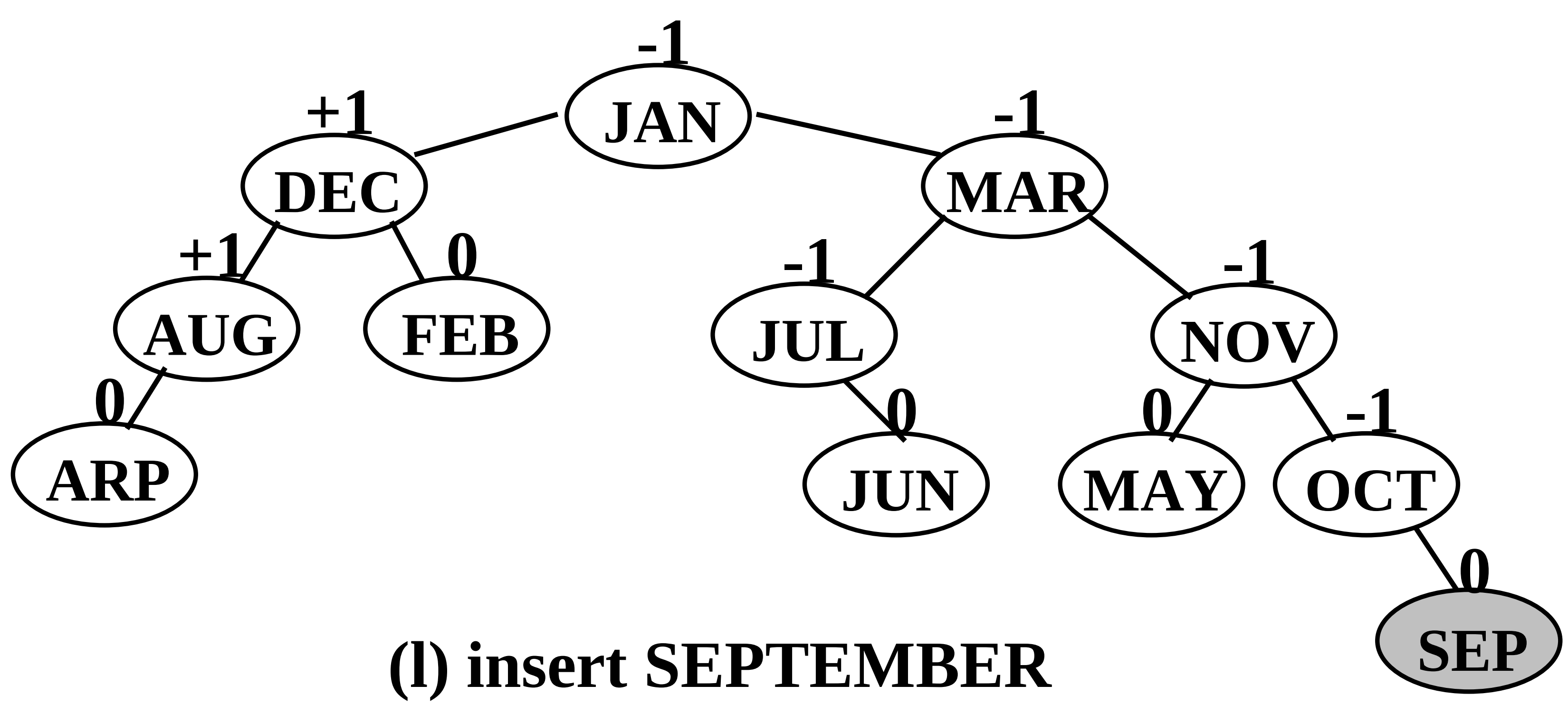


(j) insert JUNE

Balanced tree(5/5)

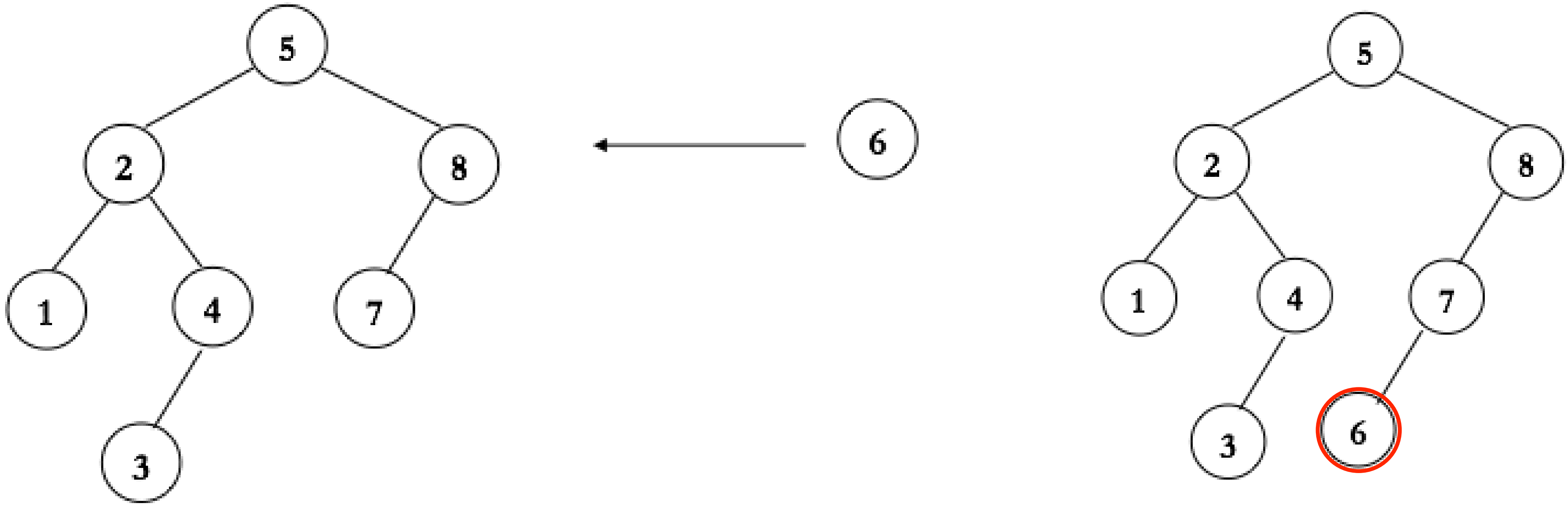


(k) insert OCTOBER



(l) insert SEPTEMBER

AVL Tree: insertion



The node “A” needs to be rebalanced, when

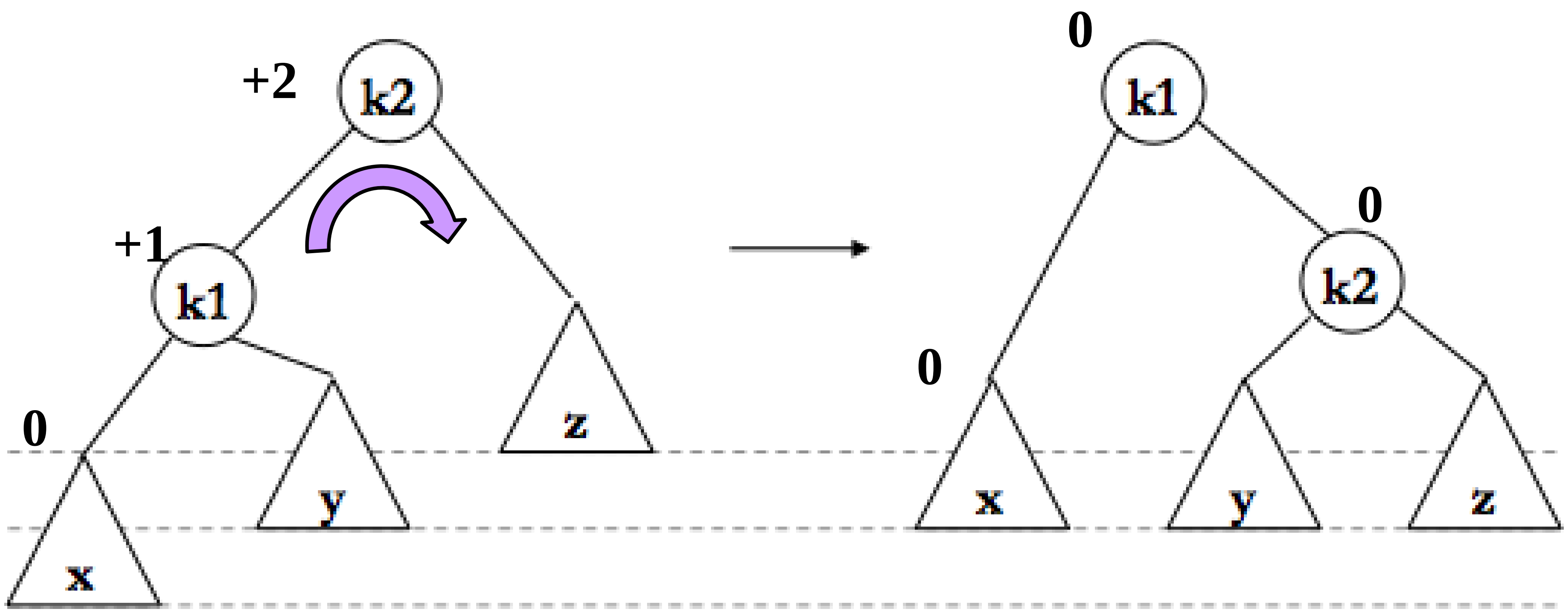
- case 1: an insertion into the left subtree of the left child of the node A
  - single rotation
- case 2: an insertion into the right subtree of the left child of the node A
  - double rotation
- case 3: an insertion into the left subtree of the right child of the node A
  - double rotation
- case 4: an insertion into the right subtree of the right child of the node A
  - single rotation

AVL tree

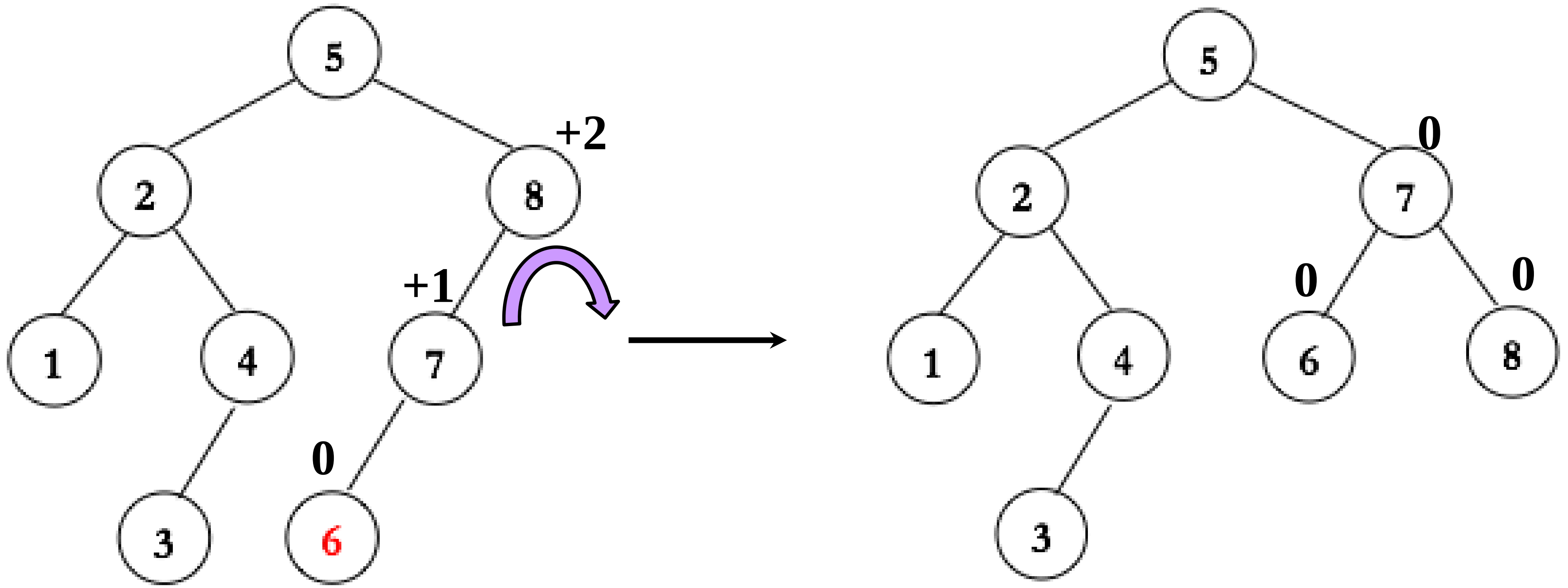
- Unbalanced tree with respect to the closest parent A (that had a BF of +2 or -2) of the new node Y
  - LL type : new node Y is inserted in the left subtree of the left subtree of A
  - LR type : Y is inserted in the right subtree of the left subtree of A
  - RL type : Y is inserted in the left subtree of the right subtree of A
  - RR type : Y is inserted in the right subtree of the right subtree of A
- Rotation carried out with respect to the closest parent A (that had a BF of +2 or -2) of the new node Y
- Single rotation : transformation to remedy LL and RR inbalance
  - LL rotation for LL type : right rotation from A to Y
  - RR rotaion for RR type : left rotation from A to Y
- Double rotation : transformation to remedy LR and RL inbalance
  - LR rotation for LR type : left rotation(RR rotation) → right rotation (LL rotation)
  - RL rotation for RL type : right rotation(LL rotation) → left rotation (RR rotation)

AVL Tree: insertion

- case 1: an insertion into the left subtree of the left child of the unbalanced node
  - LL rotation for LL type : single rotation (right rotation)



AVL Tree: insertion

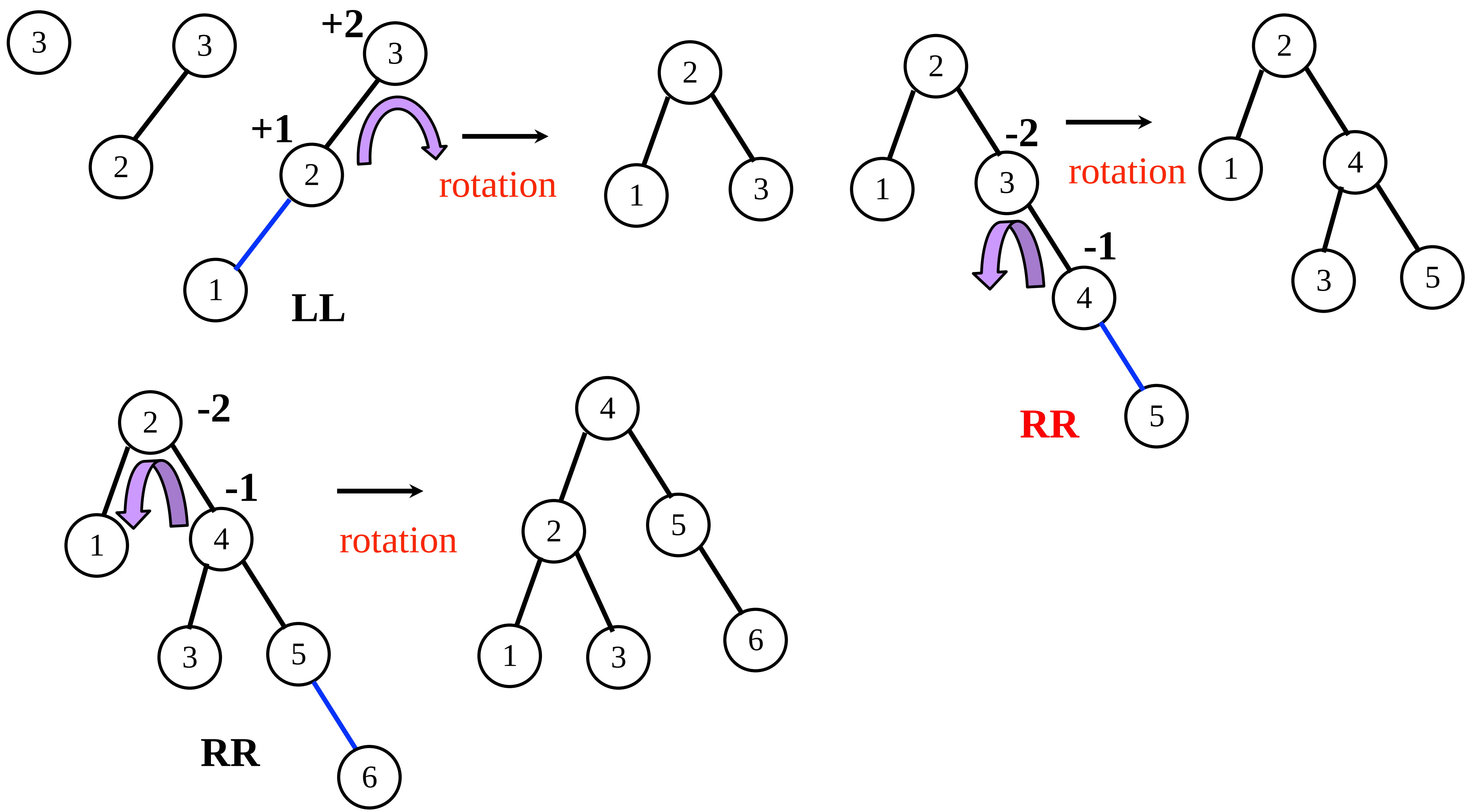


AVL Tree: insertion

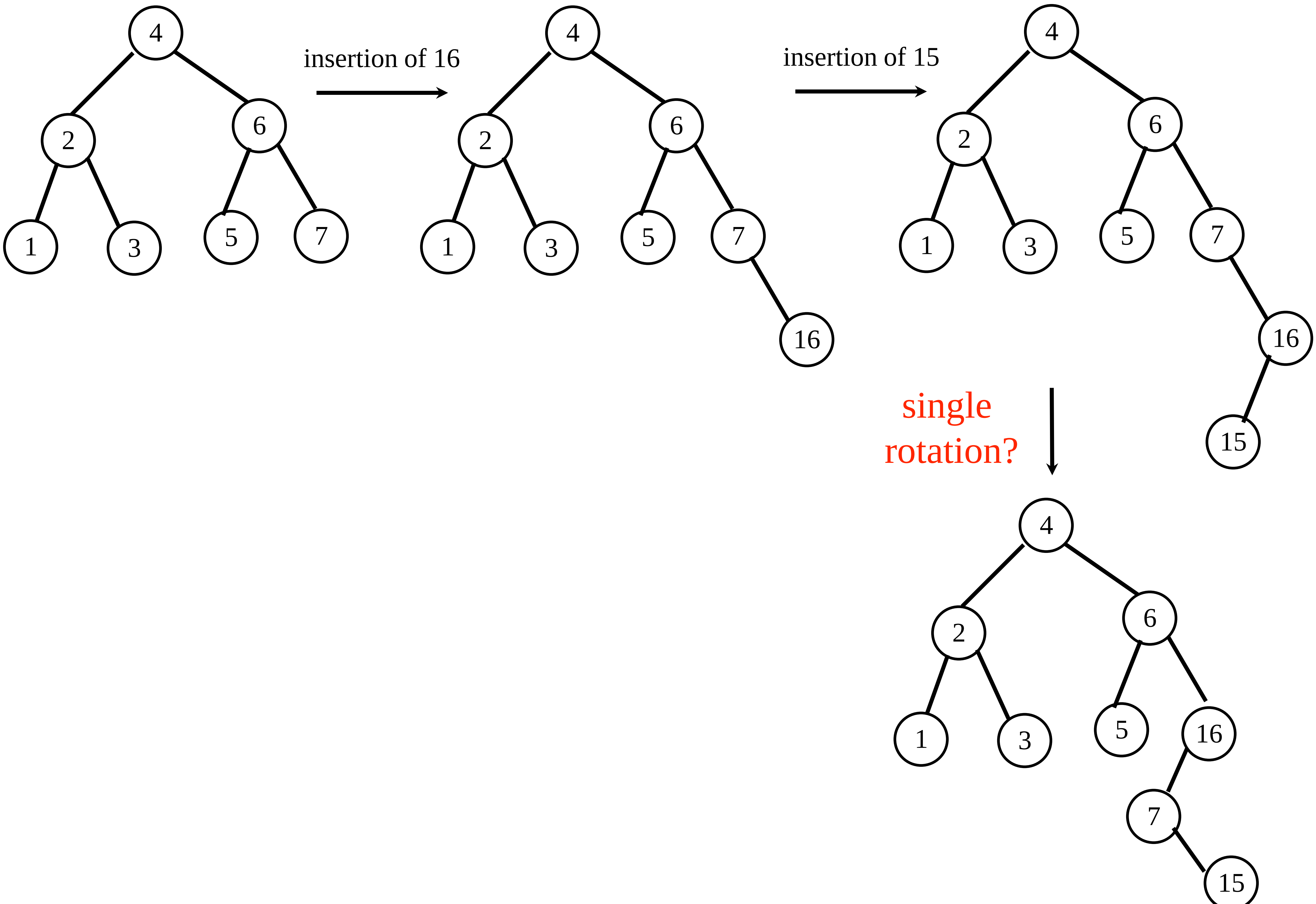
insert 3, 2, 1, 4, 5, 6

AVL Tree: insertion

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

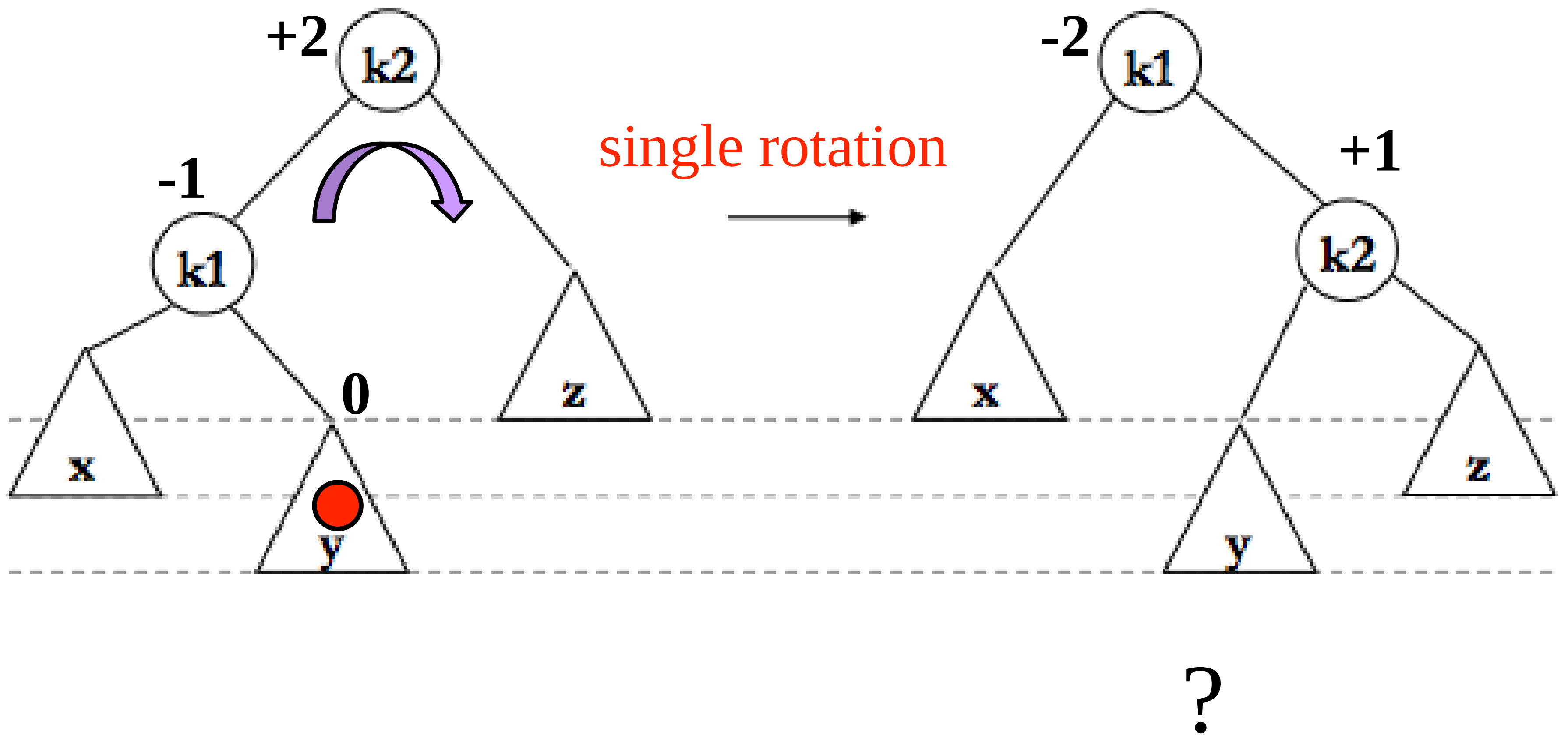


AVL Tree: insertion



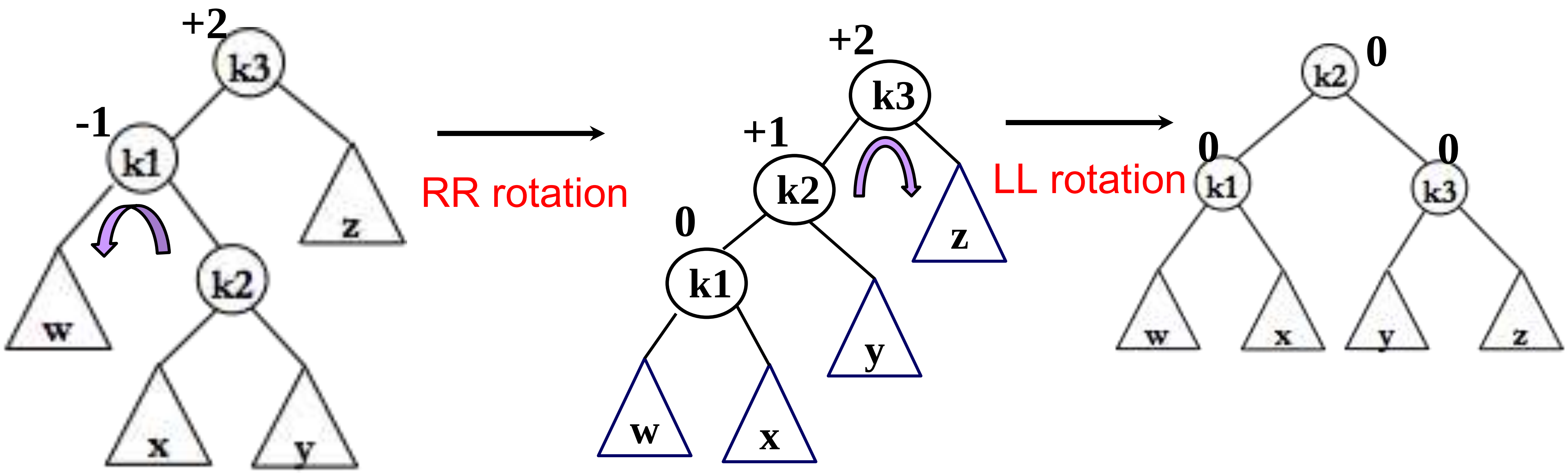
AVL Tree: insertion

- case 2: an insertion into the right subtree of the left child of the unbalanced node
  - LR rotation for LR type: double rotation (Left rotation → Right rotation)



AVL Tree: insertion

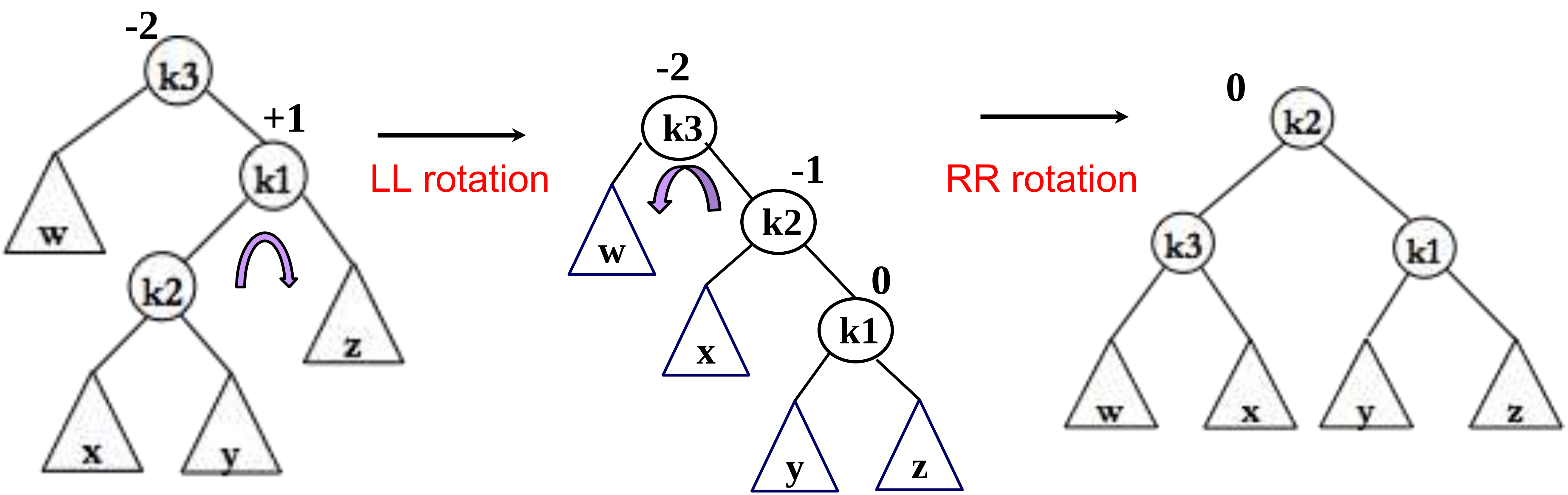
- case 2: an insertion into the right subtree of the left child of the unbalanced node
  - LR rotation for LR type: double rotation (Left rotation → Right rotation)



Left-right double rotation

AVL Tree: insertion

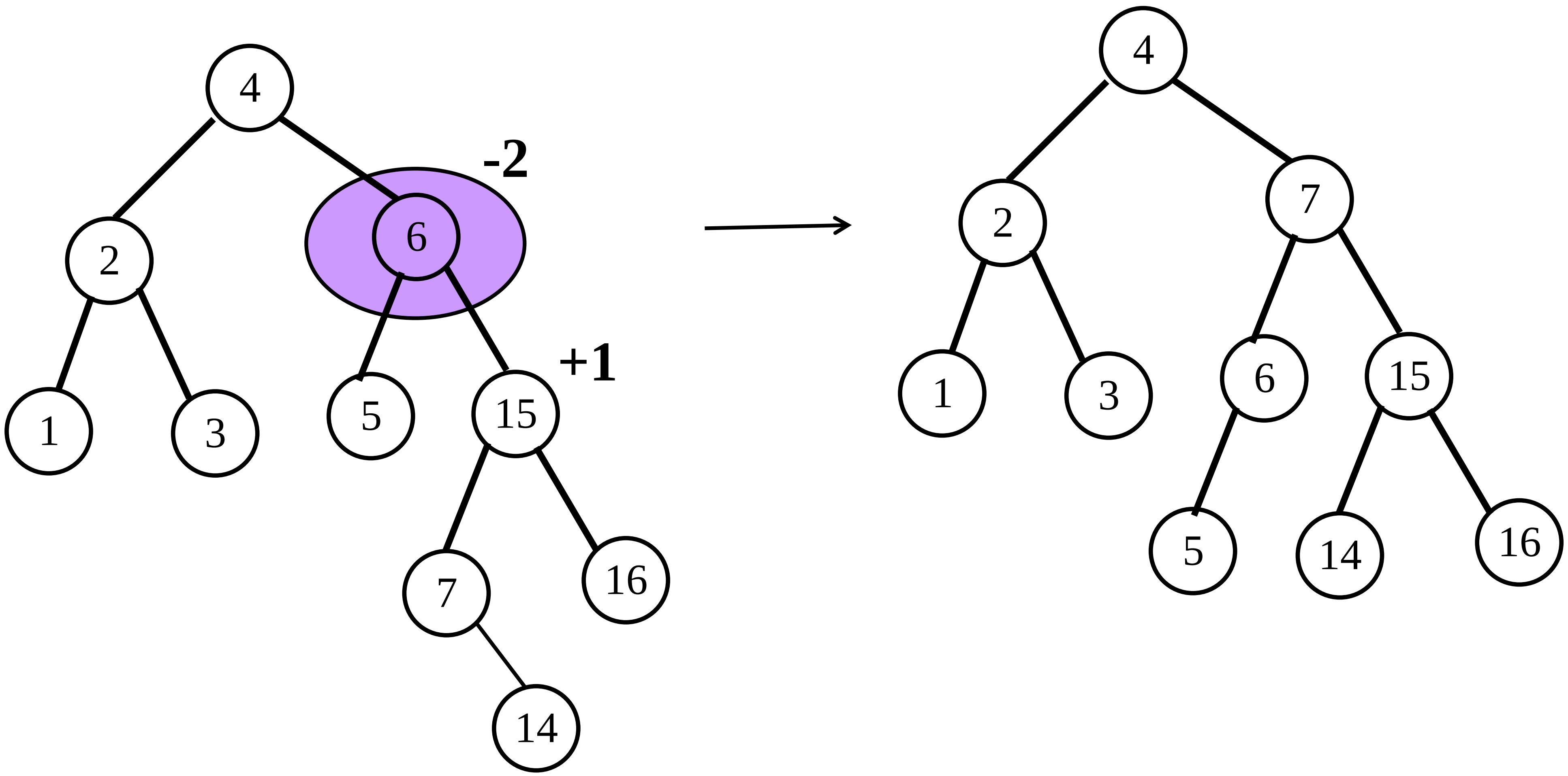
- case 3: an insertion into the left subtree of the right child of the unbalanced node
  - RL rotation for RL type: double rotation (Right rotation → Left rotation)



right-left double rotation

AVL Tree: insertion

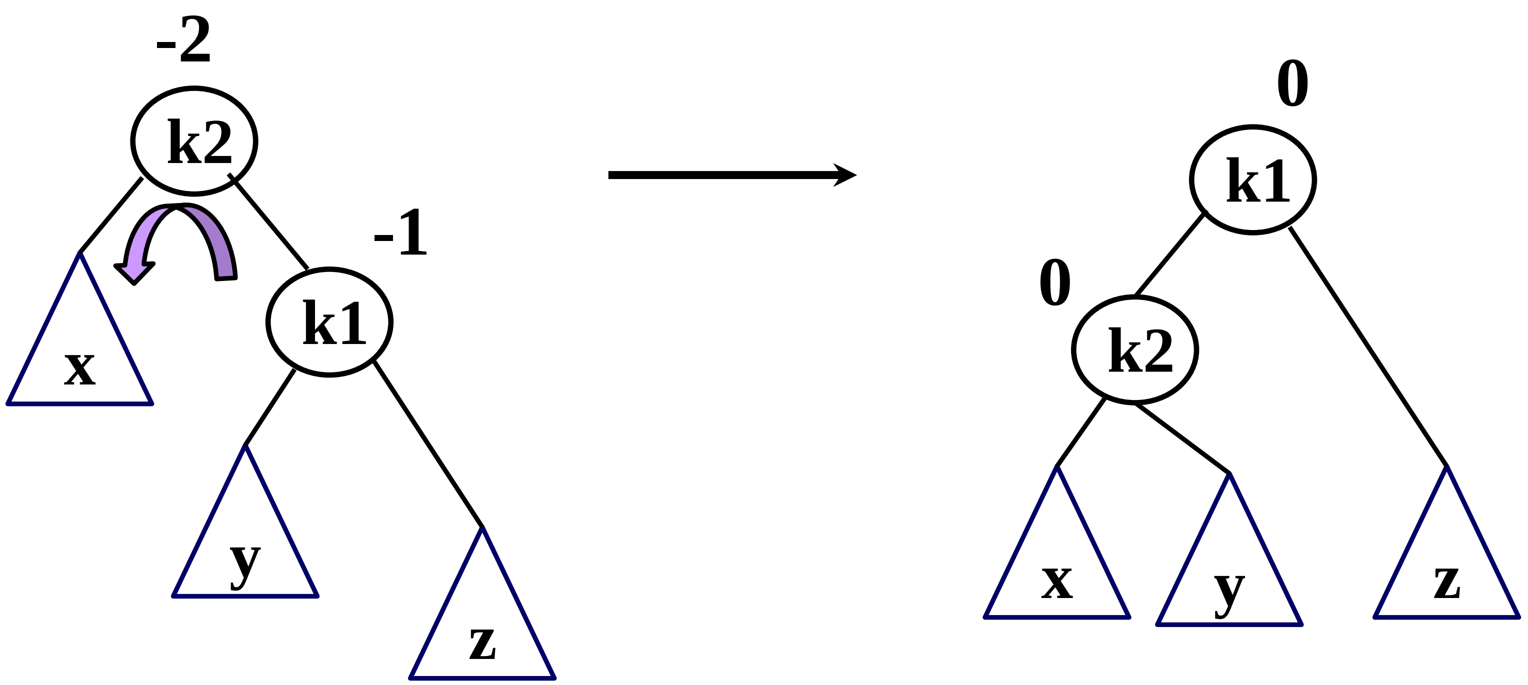
insert 14



RL type

AVL Tree: insertion

- case 4: an insertion into the right subtree of the right child of the unbalanced node
  - RR rotation for RR type : single rotation (left rotation)



Goto Slide 15

AVL Tree: exercise

Single rotation

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

Double rotation

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

Another exercise

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

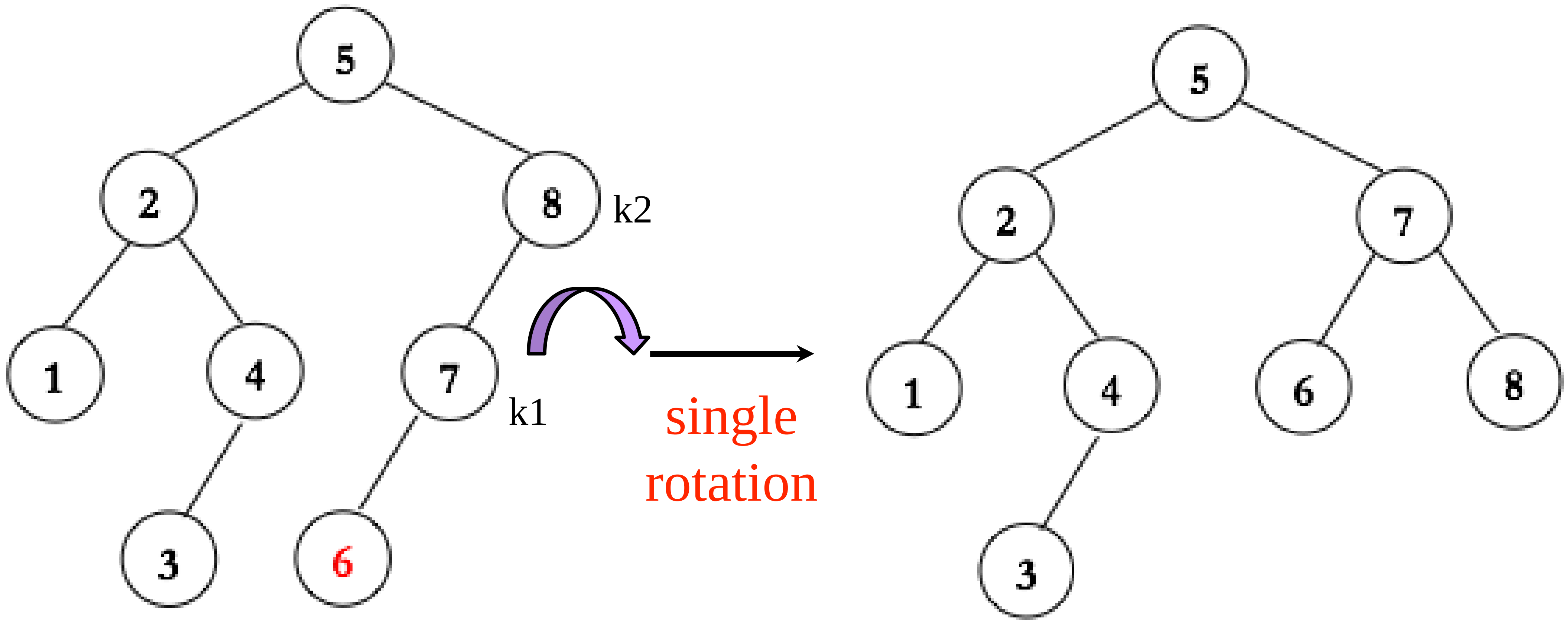
Insert sequence: 3, 2, 1, 4, 5, 6, 7, 16, 15, 14, 13, 12, 11, 10, 8, 9

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

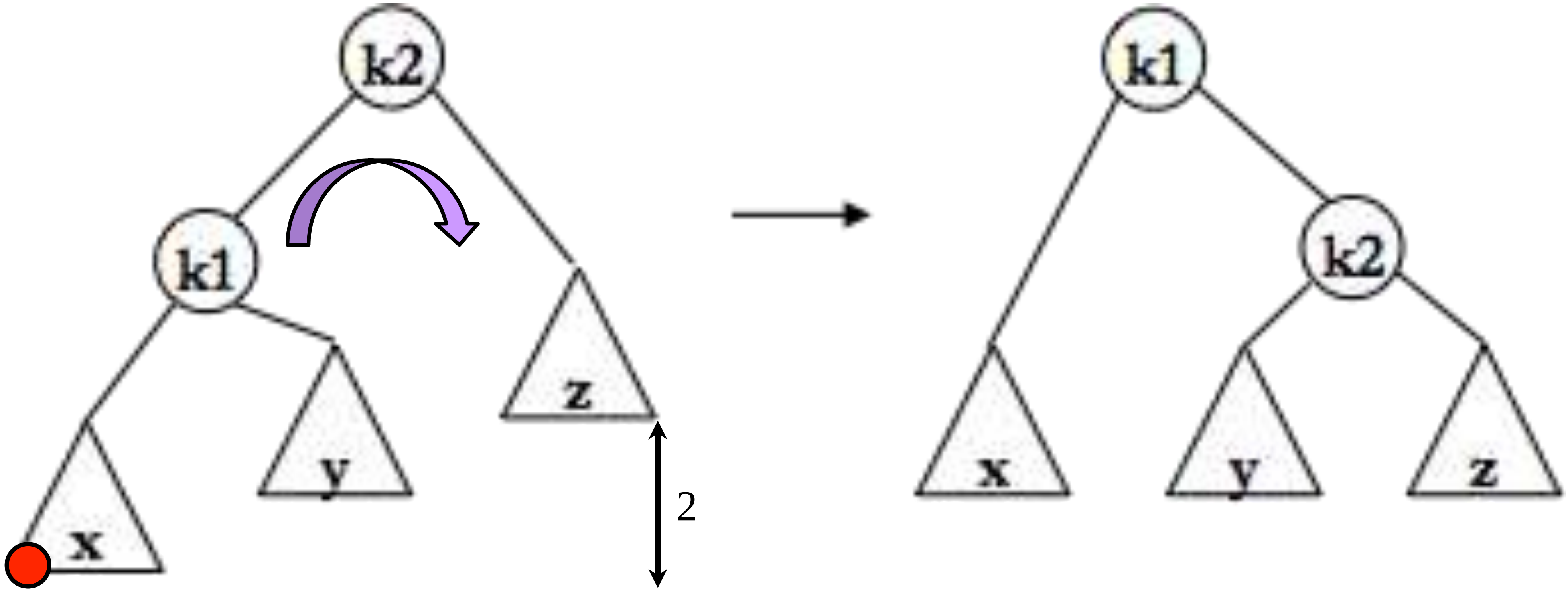
LL rotation for LL type: in case of k1 has no right child



K1->Right = K2;

AVL Tree: insertion

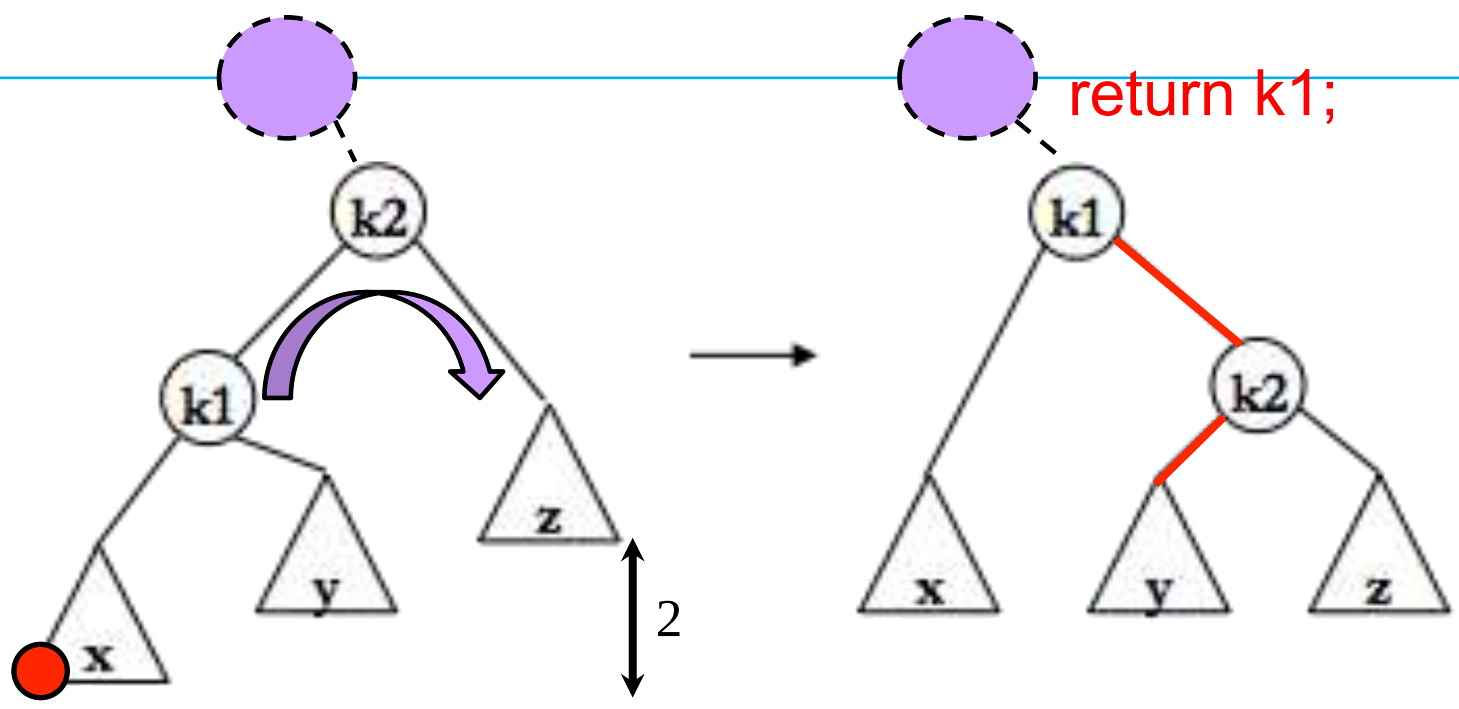
LL rotation for LL type: in case of k1 has right child




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

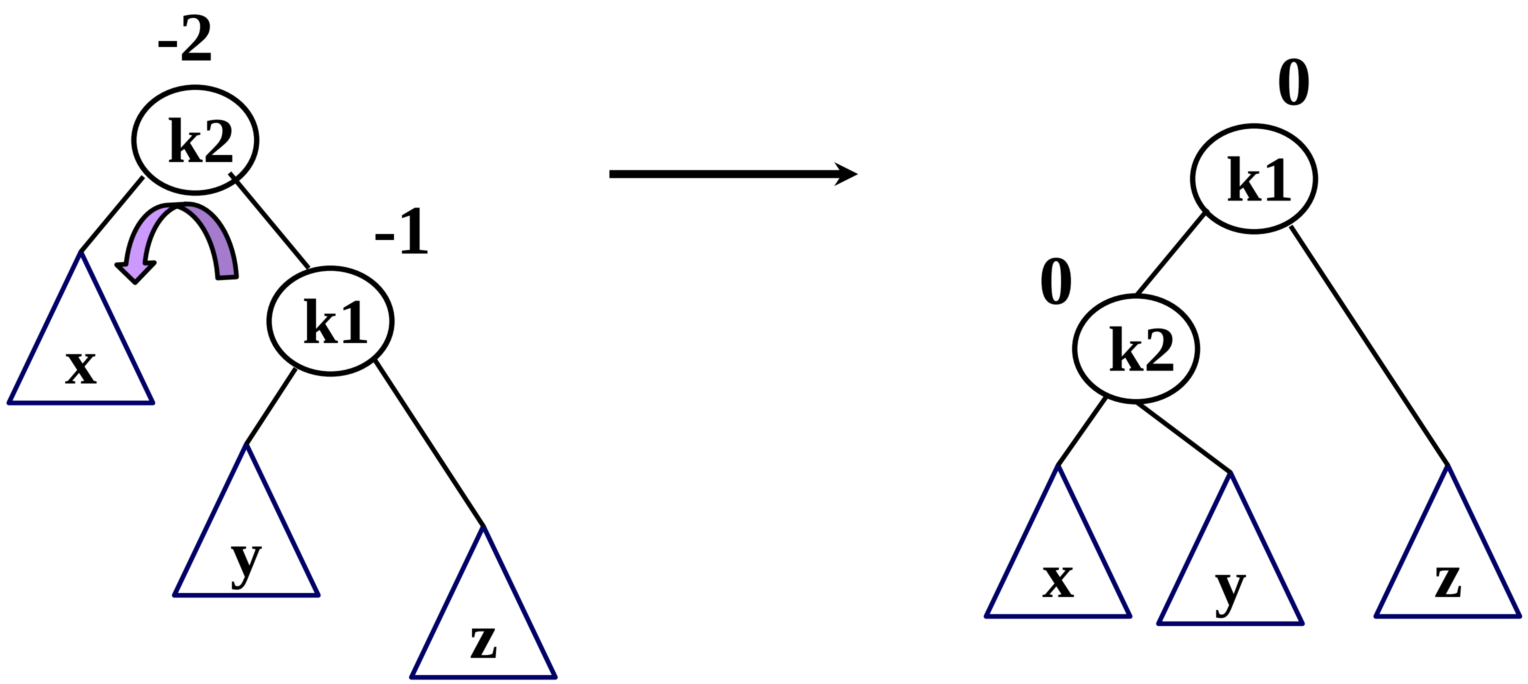
AVL Tree: insertion

```
Position  SingleRotateWithLeft( Position K2 )  /* LL rotation for LL type */
{
    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: insertion

RR rotation for RR type:

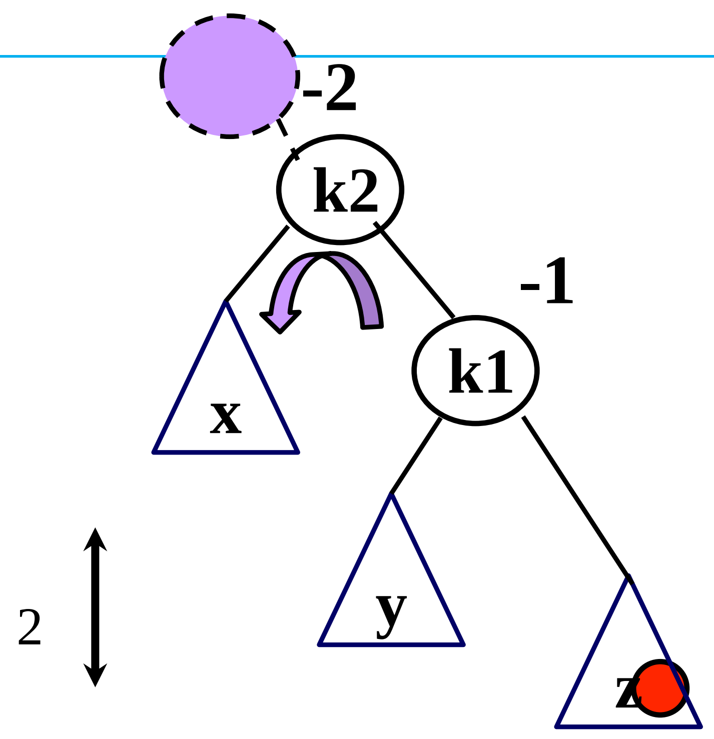


?

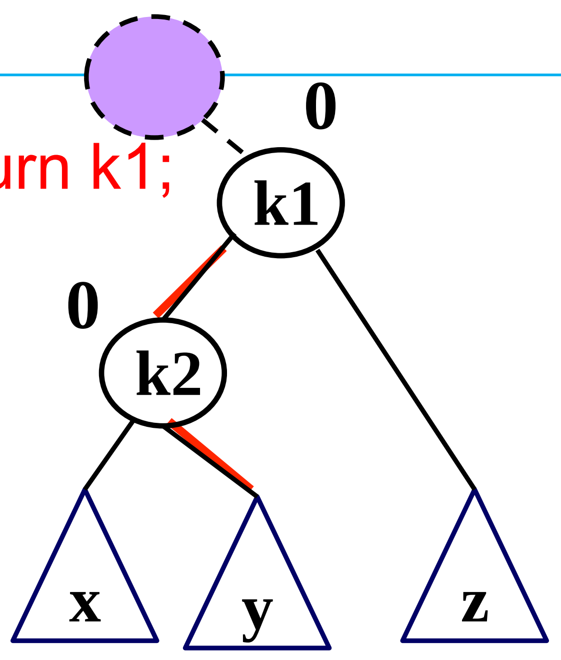
```
K2->Right = K1->Left;
K1->Left = K2;
```

AVL Tree: insertion

```
Position  SingleRotateWithRight( Position K2 )  /* RR rotation for RR type */
{
}
```

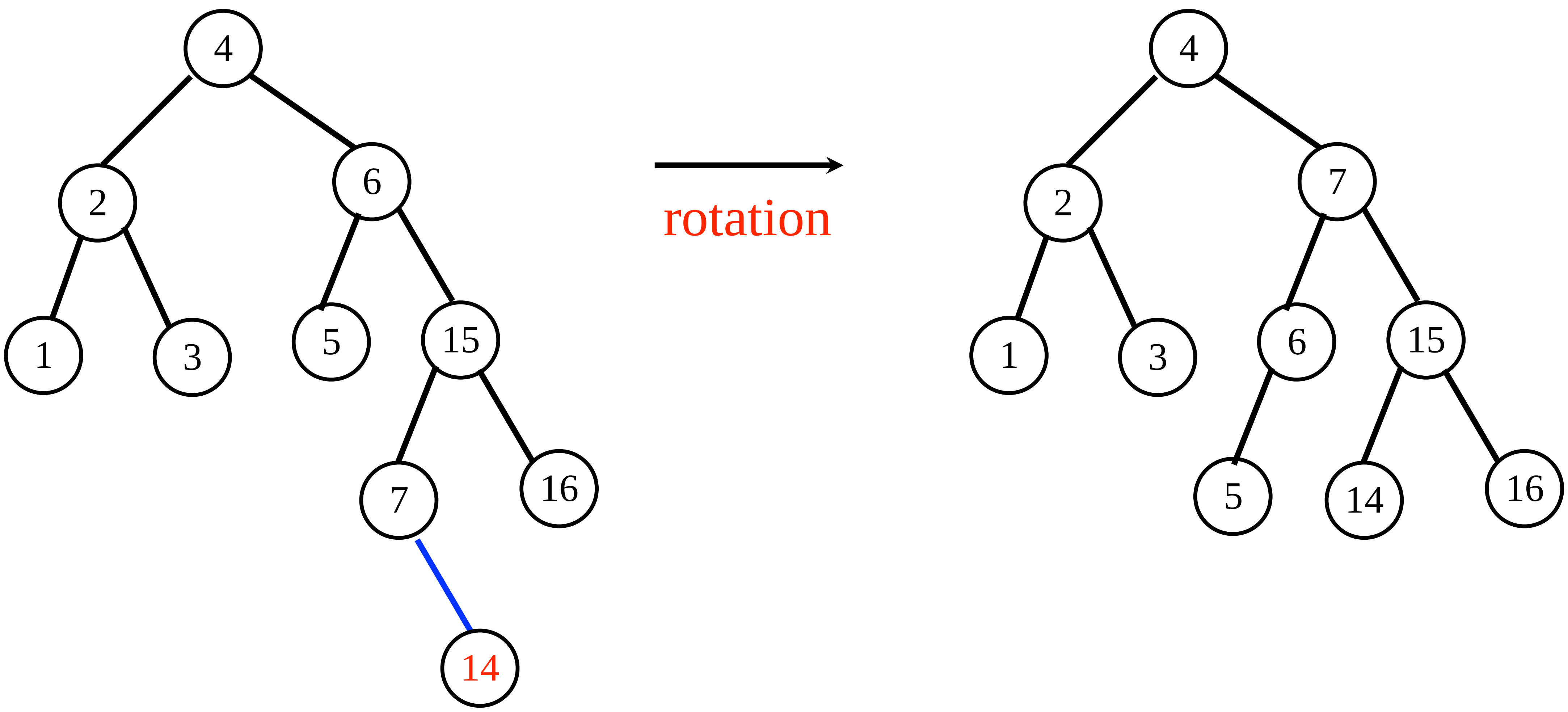


return k1;



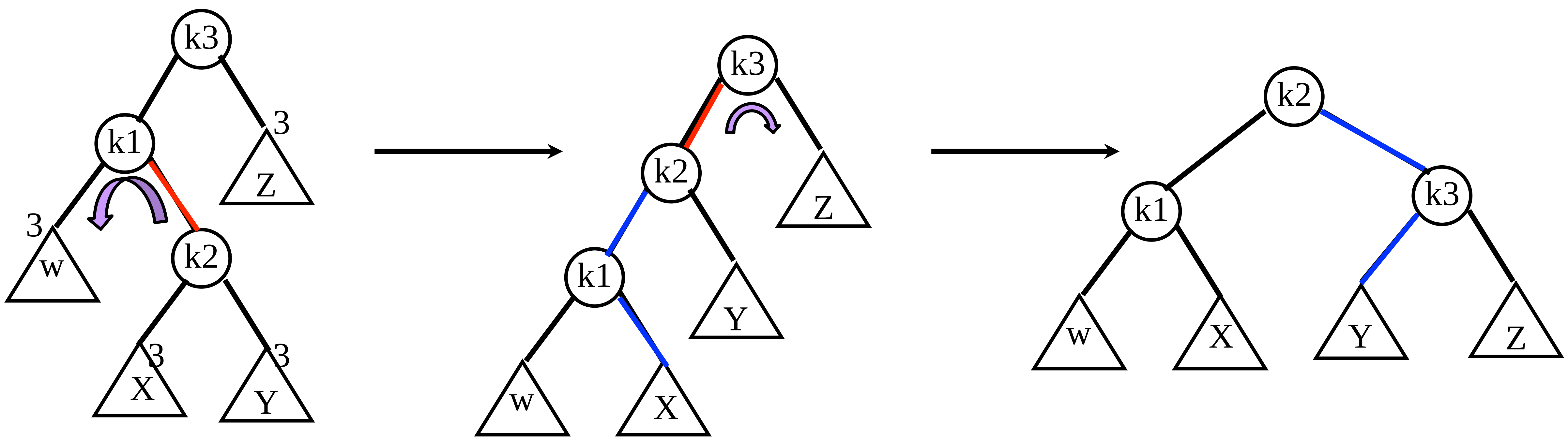
AVL Tree: insertion

RL rotation for RL type:



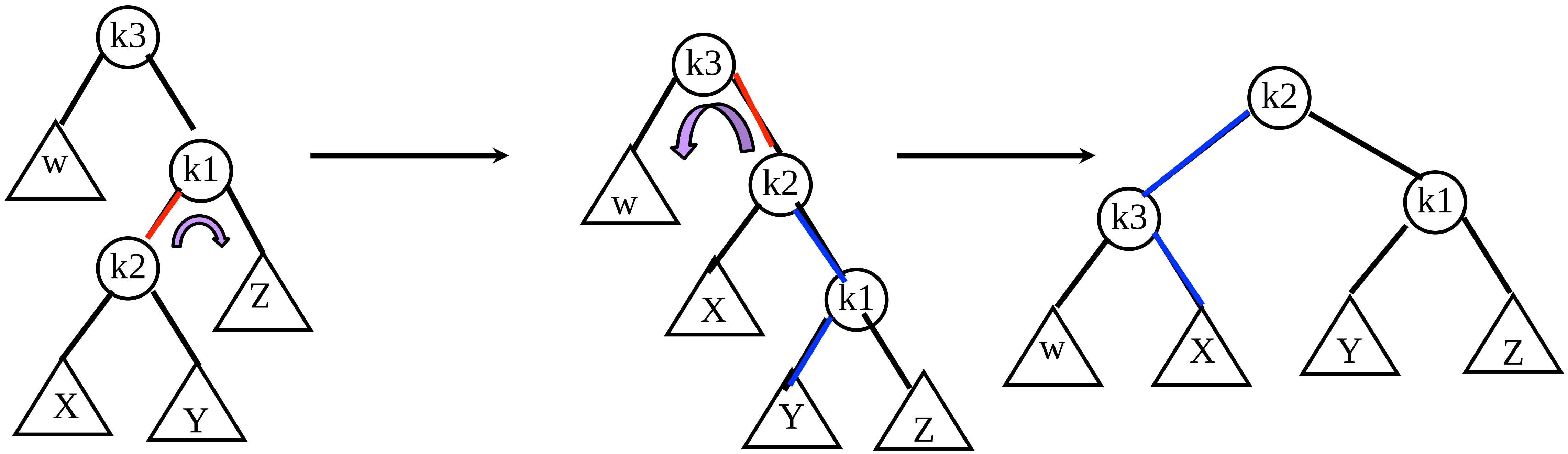
AVL Tree

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

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



AVL Tree

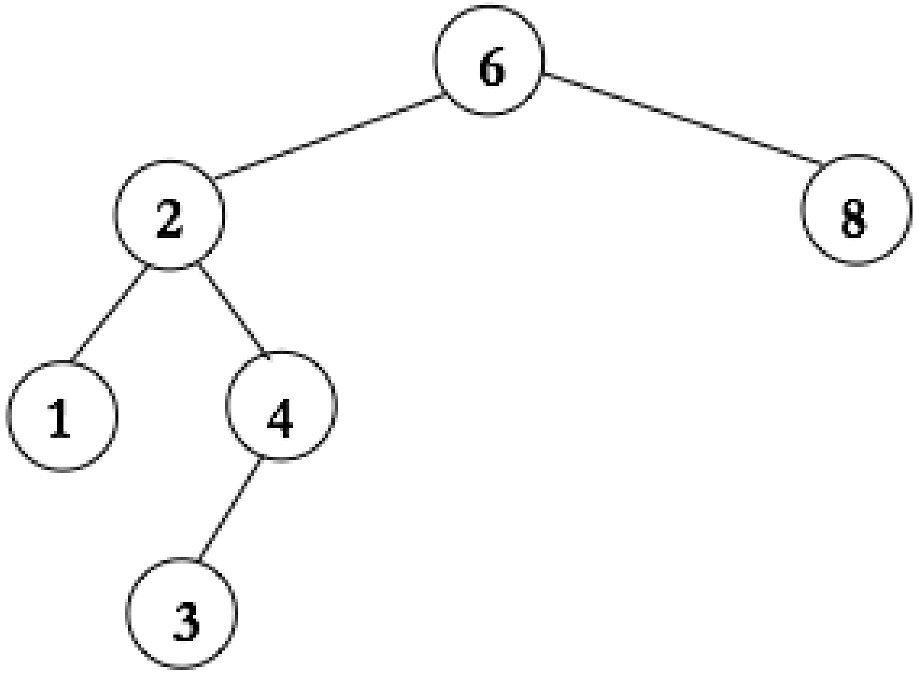
```
static Position DoubleRotateWithRight  ( Position K3 ) /* RL rotation for RL type*/
{



}
```

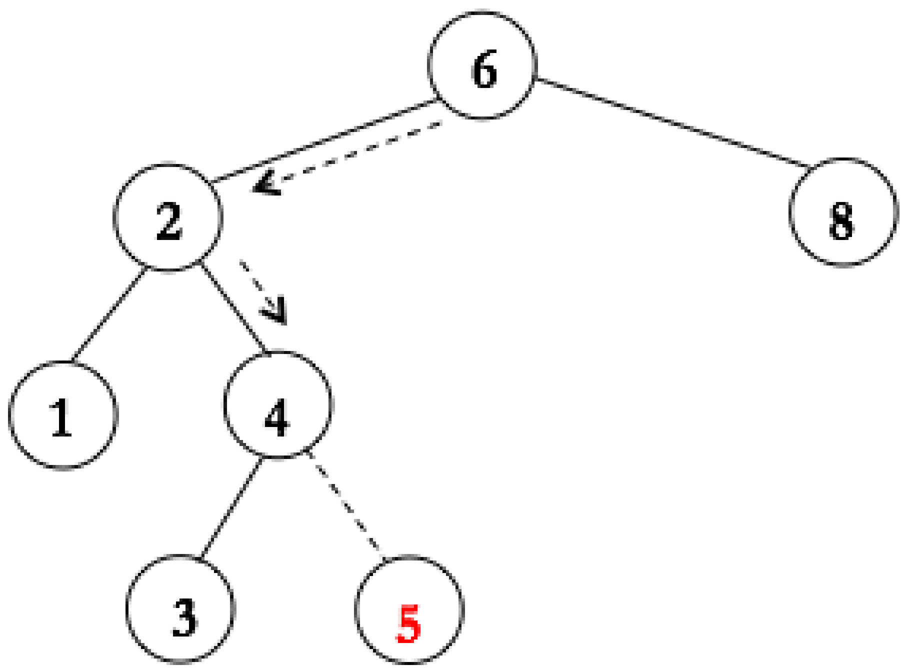


```
AVLTree Insert( ElementType X, AVLTree T )   {

     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 ); /* LL type */
               else
                    T = DoubleRotateWithLeft( T ); /* LR type */
     } else if( X > T->Element )   {
        T->Right = Insert( X, T->Right ); /* BST*/
        if( Height( T->Right ) – Height( T->Left ) == 2 )
             if( X > T->Right->Element )
                T = SingleRotateWithRight( T );  /* RR  type */
             else
                T = DoubleRotateWithRight( T ); /* RL type */
             }
     }
     T->Height = Max( Height( T->Left ), Height( T->Right ) ) + 1;
        return T;
}
```

vs. Binary Search Tree: Insert

insertion of 5





vs. Binary Search Tree: Insert

```
SearchTree  Insert ( ElementType X, SearchTree T )
{
    if( T == NULL ) {

        T = malloc( sizeof( struct TreeNode ) );
        if( T == NULL )
            FatalError( "Out of space!!!" );
        else
        {
            T->Element = X;
            T->Left = T->Right = NULL;
        }
    } else if( X < T->Element ) {
        T->Left = Insert( X, T->Left );

    } else if( X > T->Element )
       T->Right = Insert( X, T->Right );

    /* Else X is in the tree already; we'll do nothing */

    return T;  /* Do not forget this line! */
}
```

AVL Tree

rebuild the trees below