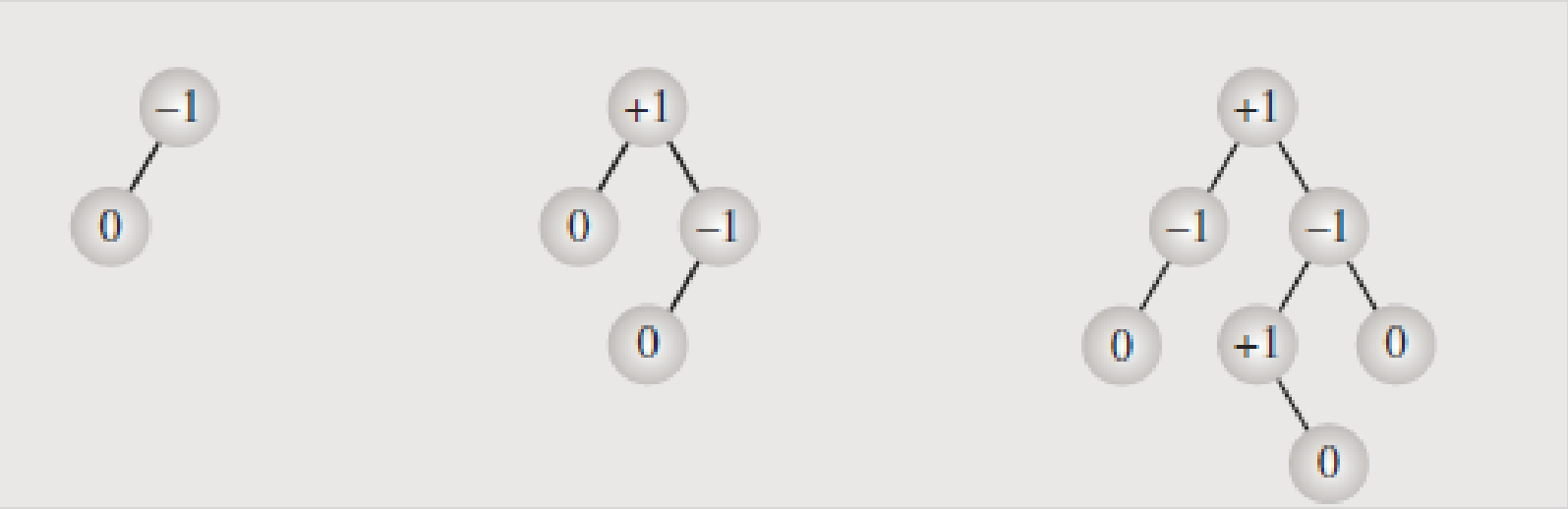
AVL TREES

Dr. Megha Ummat

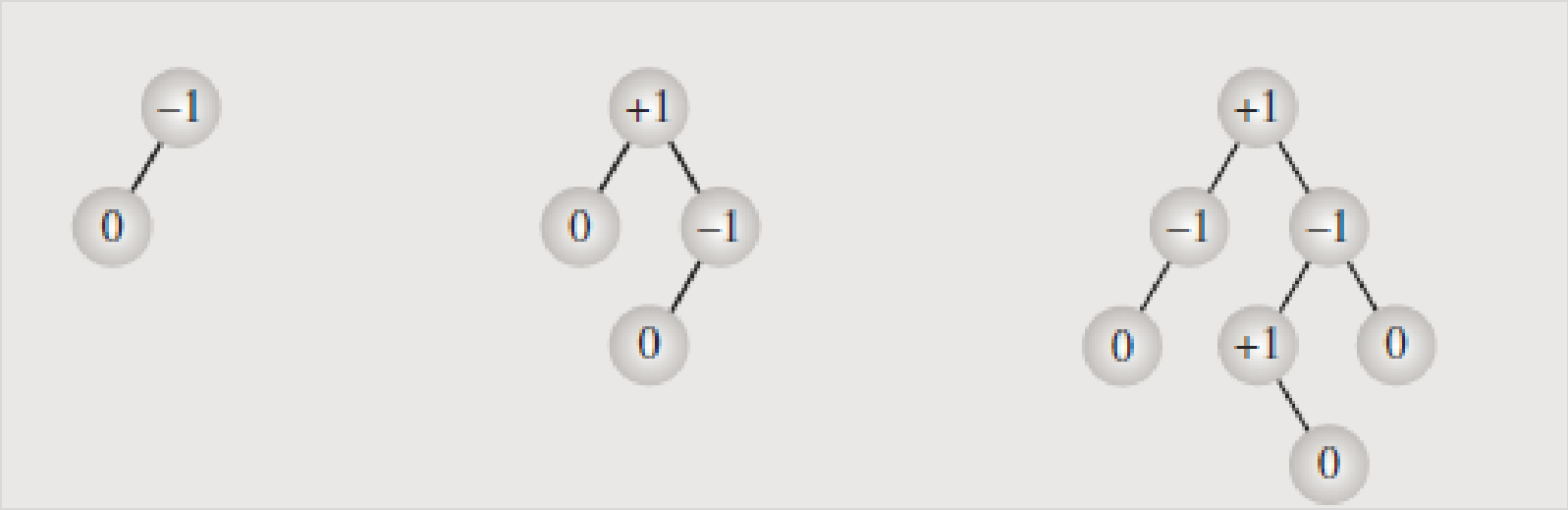
AVL Trees

- AVL Trees were proposed by Adel'son-Vel'skii and Landis, which is commemorated in the name of the tree.
- An *AVL tree* (originally called an *admissible tree*) is one in which the height of the left and right subtrees of every node differ by at most one.



AVL Trees

- Numbers in the nodes indicate the *balance factors* that are the differences between the heights of the left and right subtrees.
- A balance factor is the height of the right subtree minus the height of the left subtree. For an AVL tree, all balance factors should be +1, 0, or –1.



AVL Trees

- The definition of an AVL tree indicates that the minimum number of nodes in a tree is determined by the recurrence equation

$$AVL_h = AVL_{h-1} + AVL_{h-2} + 1$$

where $AVL_0 = 0$ and $AVL_1 = 1$ are the initial conditions.

- This formula leads to the following bounds on the height h of an AVL tree depending on the number of nodes n

$$\lg(n+1) \leq h < 1.44\lg(n+2) - 0.328$$

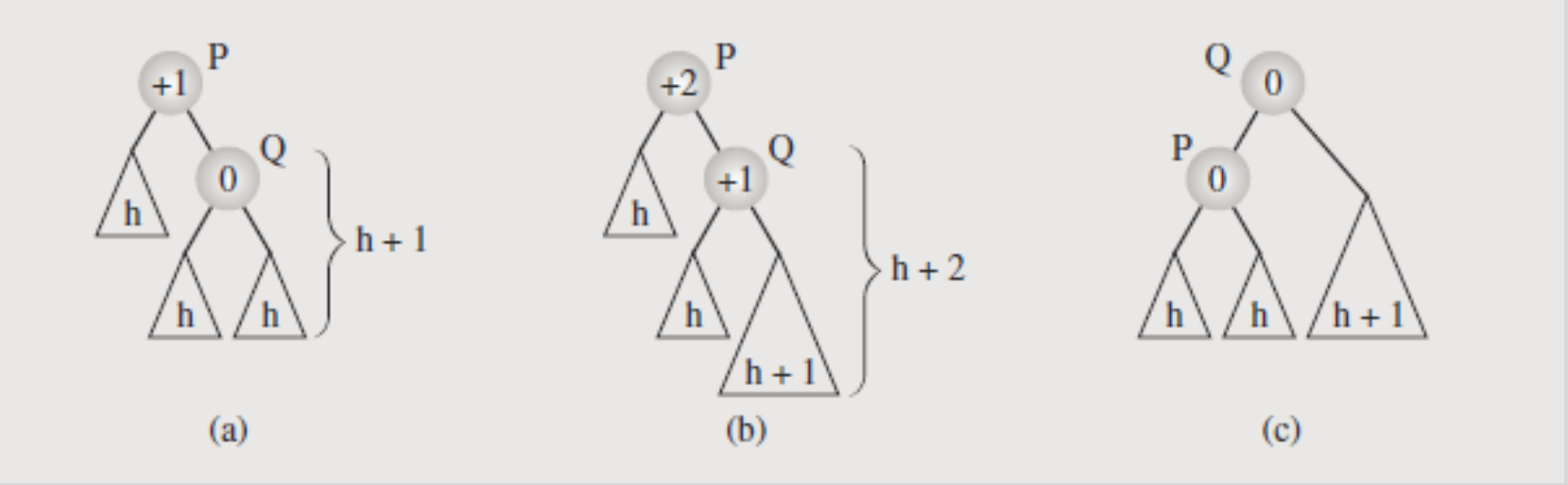
- Therefore, h is bounded by $O(\lg n)$; the worst case search requires $O(\lg n)$ comparisons.

AVL Tree Insertion

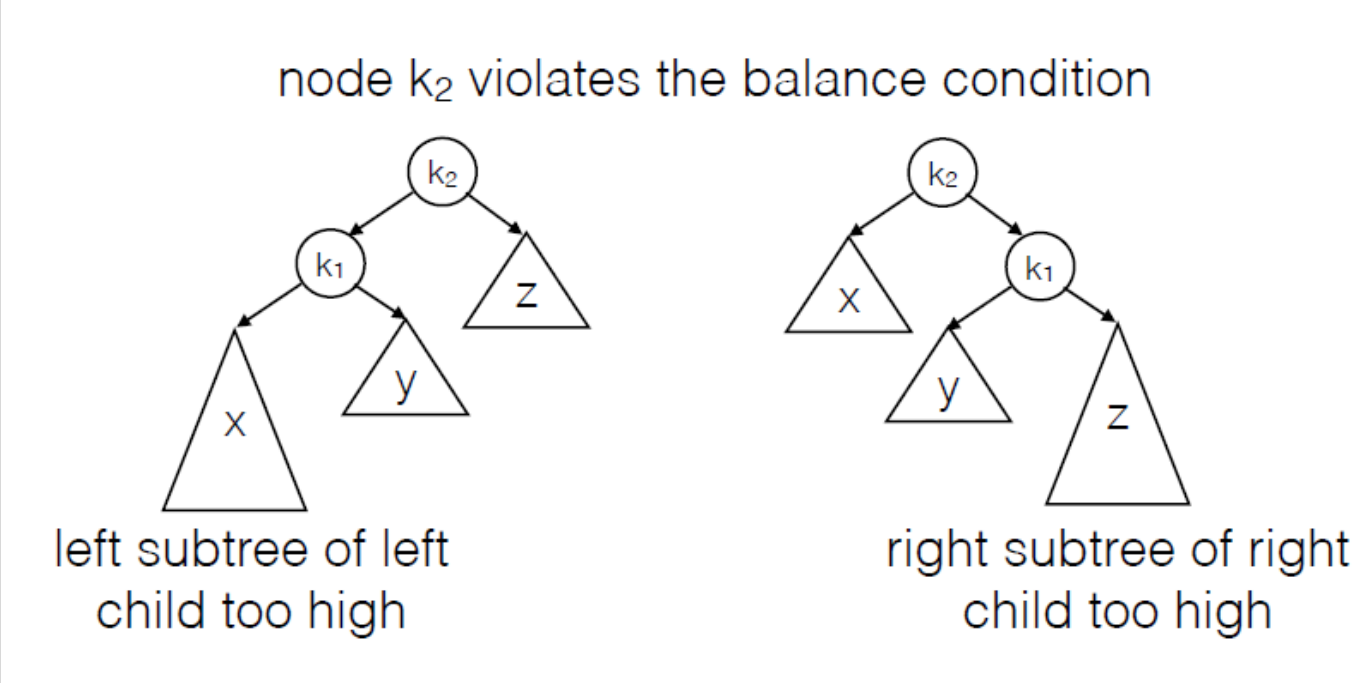
- If the balance factor of any node in an AVL tree becomes less than –1 or greater than 1, the tree has to be balanced.
- An AVL tree can become out of balance in four situations, but only two of them need to be analyzed; the remaining two are symmetrical.

Case 1

- The first case, the result of inserting a node in the right subtree of the right child, is illustrated in Figure. The heights of the participating subtrees are indicated within these subtrees.
- In the AVL tree in Figure (a), a node is inserted somewhere in the right subtree of *Q* (Figure (b)), which disturbs the balance of the tree *P*.
- In this case, the problem can be easily rectified by rotating node *Q* about its parent *P* (Figure (c)) so that the balance factor of both *P* and *Q* becomes zero, which is even better than at the outset.

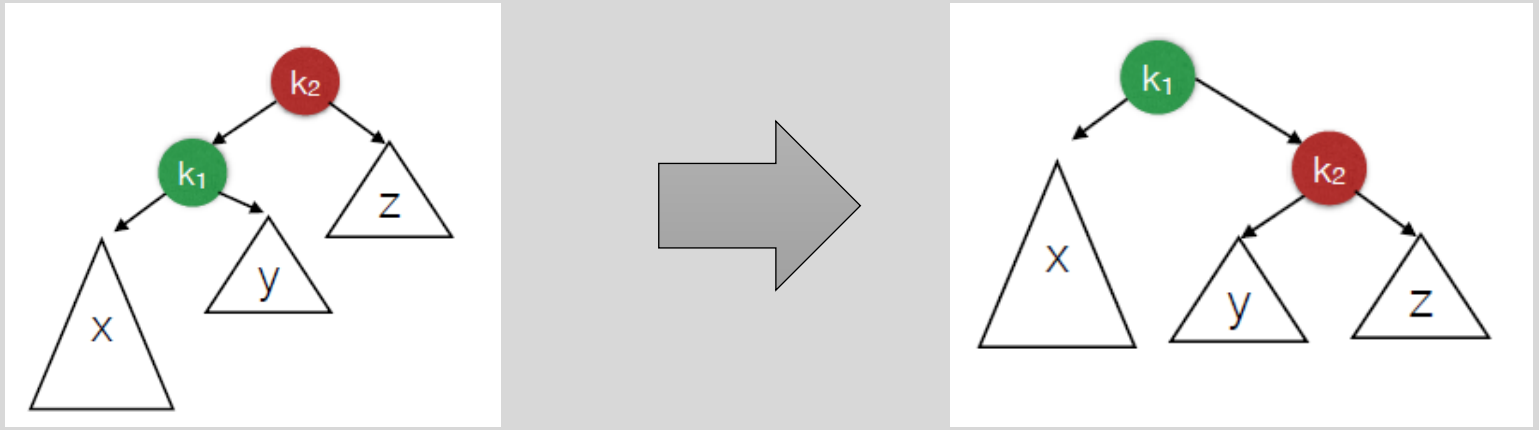


Rebalancing Trees with Single Rotation



Solution: Single Rotation

Single Rotation



- **Maintain BST property:**
- x is still left subtree of k*1.*
- z is still right subtree of *k2.*
- For all values *v* in *y*: k*1* < *v* < *k2* so y becomes new left subtree of *k2*.

Modify 3 references:

- k_2.left = k_1.right
- k_1.right = k_2
- parent(k_2).left = k_1 or parent(k_2).right = k_1

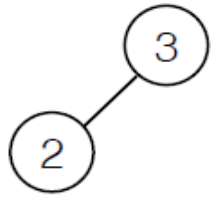
Example

Insert 3

insert(3)

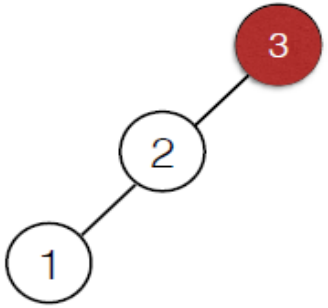
Insert 2

insert(3)
insert(2)



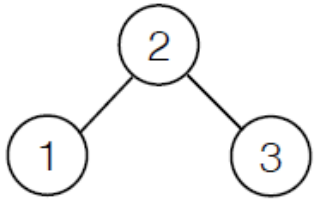
Insert 1

insert(3)
insert(2)
insert(1) rotate_left(3)



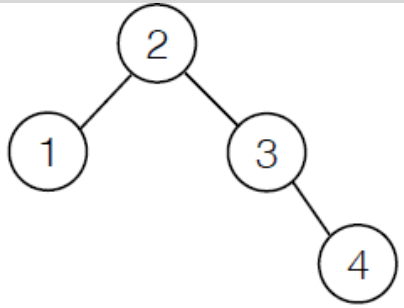
Example

insert(3)
insert(2)
insert(1) rotate_left(3)



Insert 4

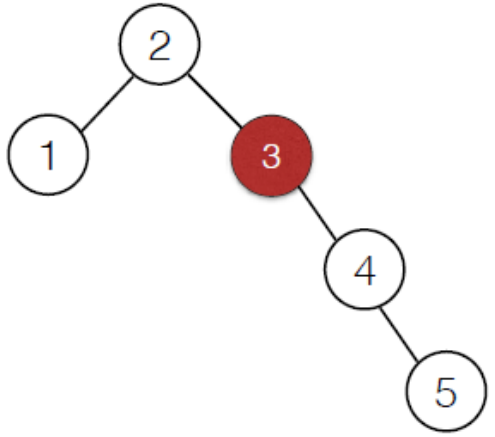
insert(3)
insert(2)
insert(1) rotate_left(3)
insert(4)



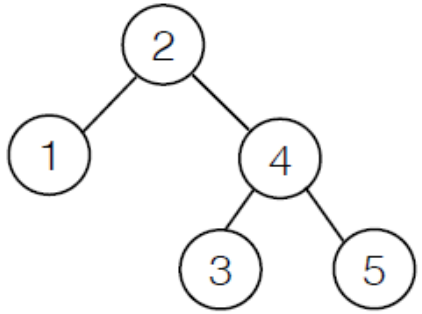
Example

Insert 5

insert(3)
insert(2)
insert(1) rotate_left(3)
insert(4)
insert(5) rotate_right(3)



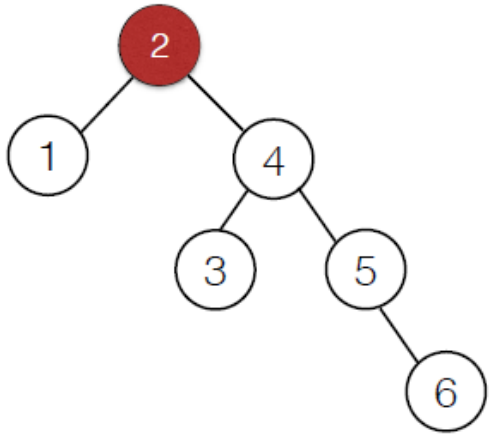
insert(3)
insert(2)
insert(1) rotate_left(3)
insert(4)
insert(5) rotate_right(3)



Example

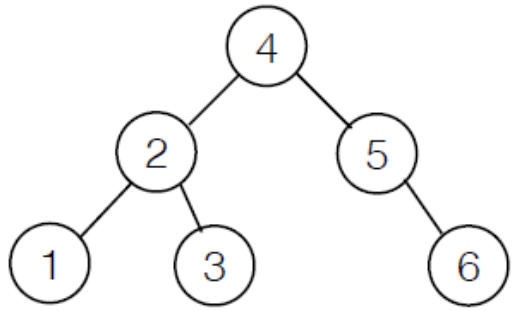
Insert 6

insert(3)
insert(2)
insert(1) rotate_left(3)
insert(4)
insert(5) rotate_right(3)
insert(6) rotate_right(2)



Example

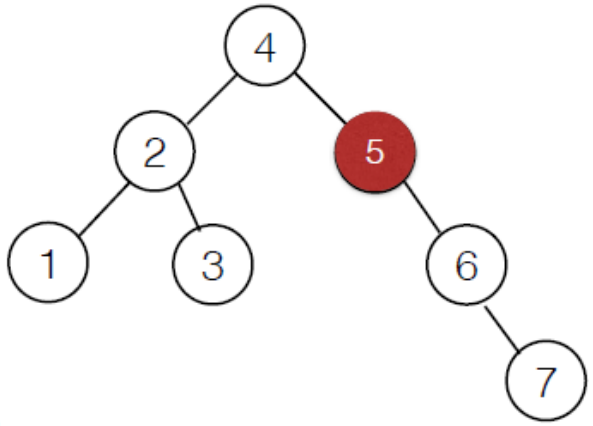
insert(3)
insert(2)
insert(1) rotate_left(3)
insert(4)
insert(5) rotate_right(3)
insert(6) rotate_right(2)



Example

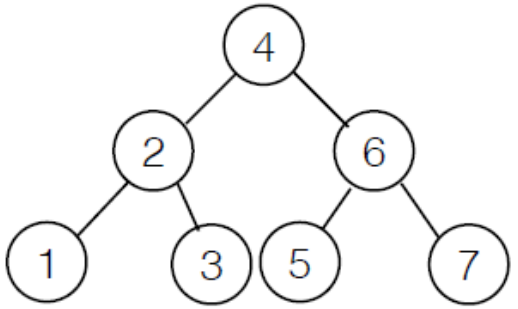
Insert 7

insert(3)
insert(2)
insert(1) rotate_left(3)
insert(4)
insert(5) rotate_right(3)
insert(6) rotate_right(2)
insert(7) rotate_right(5)



Example

insert(3)	
insert(2)	
insert(1)	rotate_left(3)
insert(4)	
insert(5)	rotate_right(3)
insert(6)	rotate_right(2)
insert(7)	rotate_right(5)



Balancing a Tree

- In the cases discussed, the tree *P* is considered a stand-alone tree. However, *P* can be part of a larger AVL tree; it can be a child of some other node in the tree.
- If a node is entered into the tree and the balance of *P* is disturbed and then restored, does extra work need to be done to the predecessor(s) of *P?* Fortunately not.
- Changes made to the subtree *P* are sufficient to restore the balance of the entire AVL tree.
- The problem is in finding a node *P* for which the balance factor becomes unacceptable after a node has been inserted into the tree.

Balancing a Tree

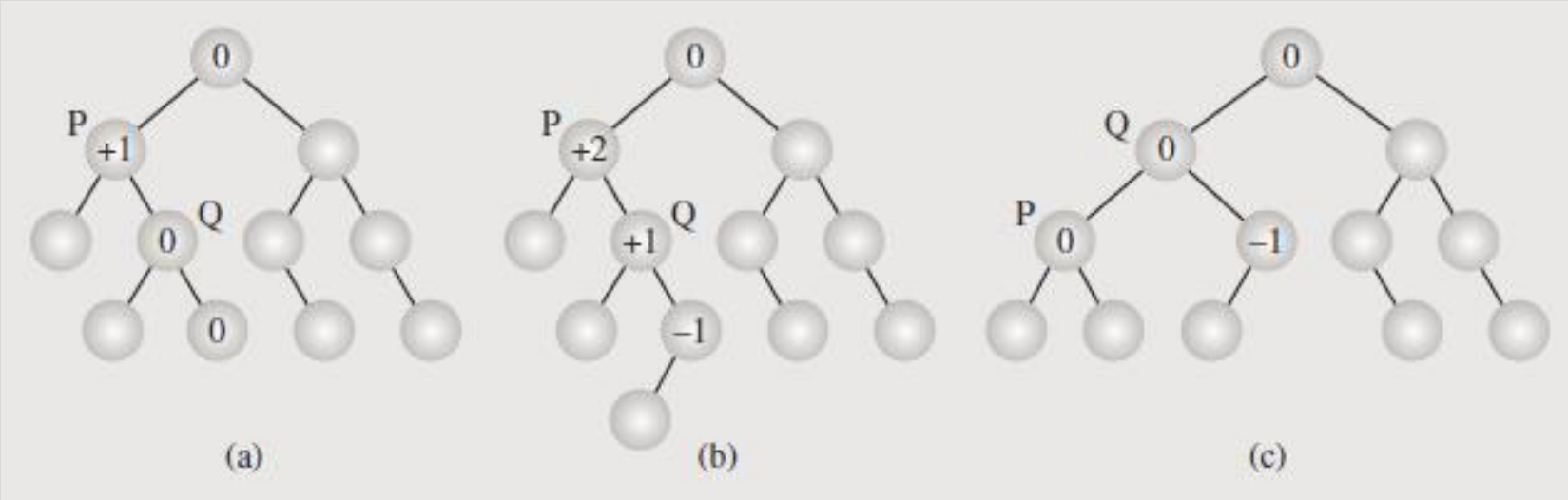
- This node can be detected by moving up toward the root of the tree from the position in which the new node has been inserted and by updating the balance factors of the nodes encountered.
- Then, if a node with a ±1 balance factor is encountered, the balance factor may be changed to ±2, and the first node whose balance factor is changed in this way becomes the root *P* of a subtree for which the balance has to be restored.
- Note that the balance factors do not have to be updated above this node because they remain the same.

Algorithm for Updating Balance Factors

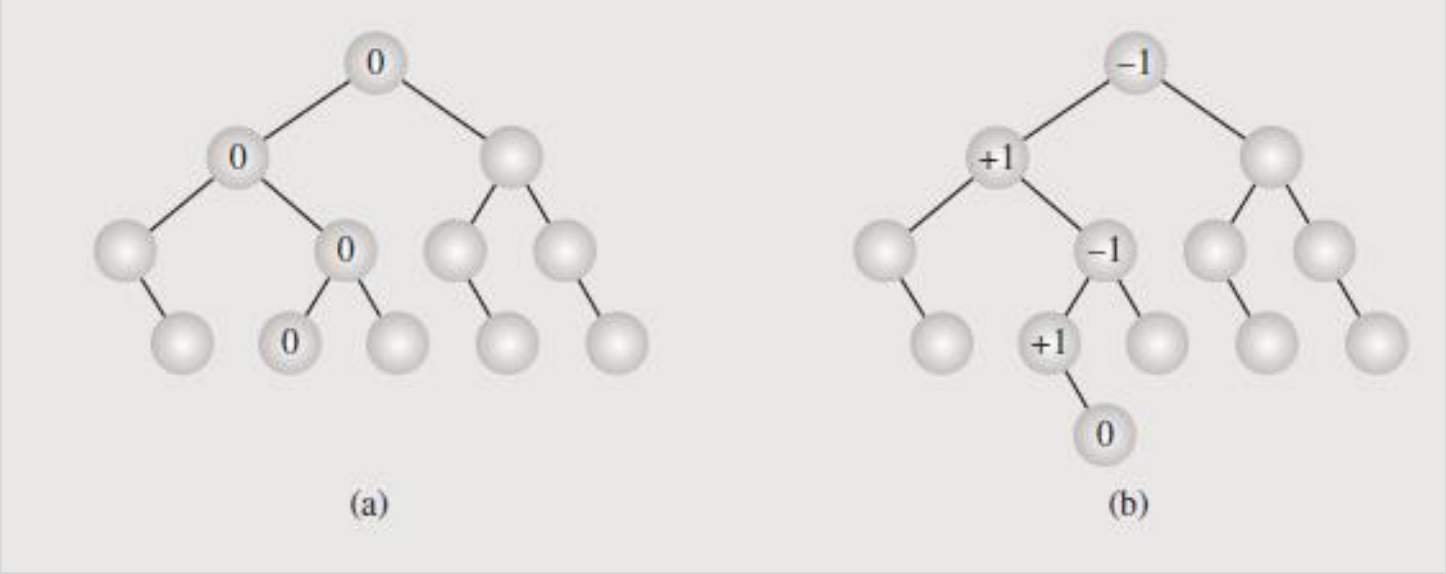
```
updateBalanceFactors()
   Q = the node just inserted;
   P = parent of Q;
   if Q is the left child of P
         P->balanceFactor--;
   else P->balanceFactor++;
   while P is not the root and P->balanceFactor ≠ ±2
        Q = P;
        P = P's parent;
        if Q->balanceFactor is 0
              return;
        if Q is the left child of P
              P->balanceFactor--;
        else P->balanceFactor++;
   if P->balanceFactor is ±2
        rebalance the subtree rooted at P;
```

Algorithm for Updating Balance Factors

- In Figure (a), a path is marked with one balance factor equal to +1.
- Insertion of a new node at the end of this path results in an unbalanced tree (Figure (b)), and the balance is restored by one left rotation (Figure (c)).

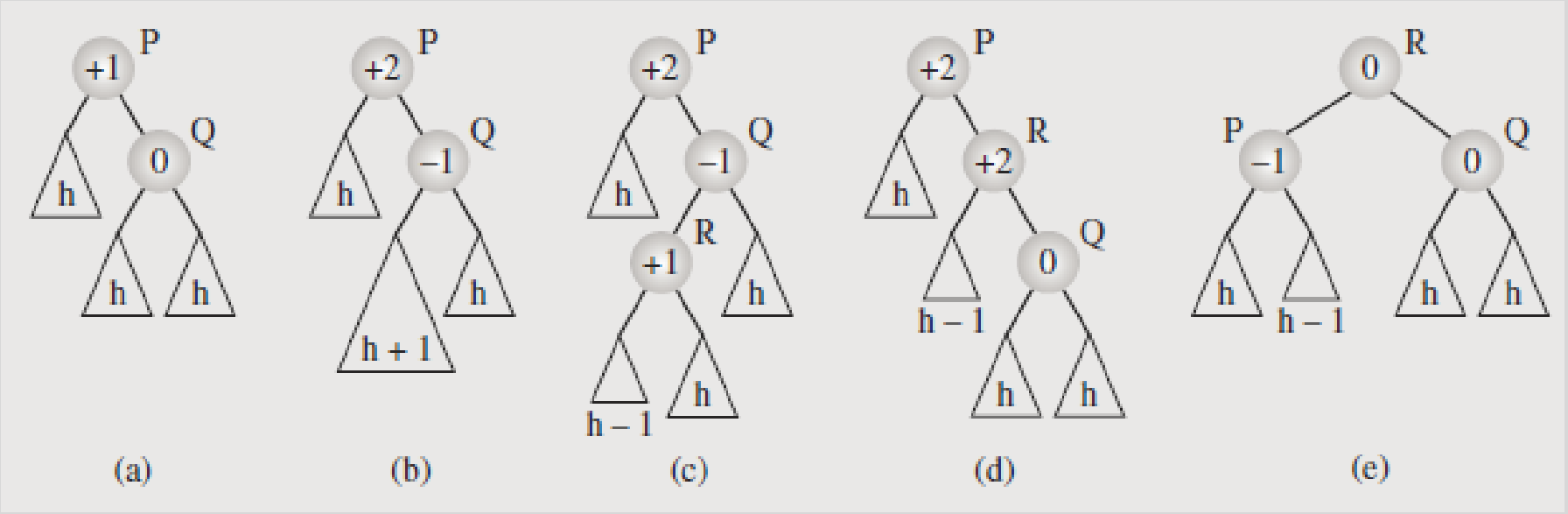


- If the balance factors on the path from the newly inserted node to the root of the tree are all zero, all of them have to be updated, but no rotation is needed for any of the encountered nodes.
- In Figure (a), the AVL tree has a path of all zero balance factors. After a node has been appended to the end of this path (Figure (b)), no changes are made in the tree except for updating the balance factors of all nodes along this path.

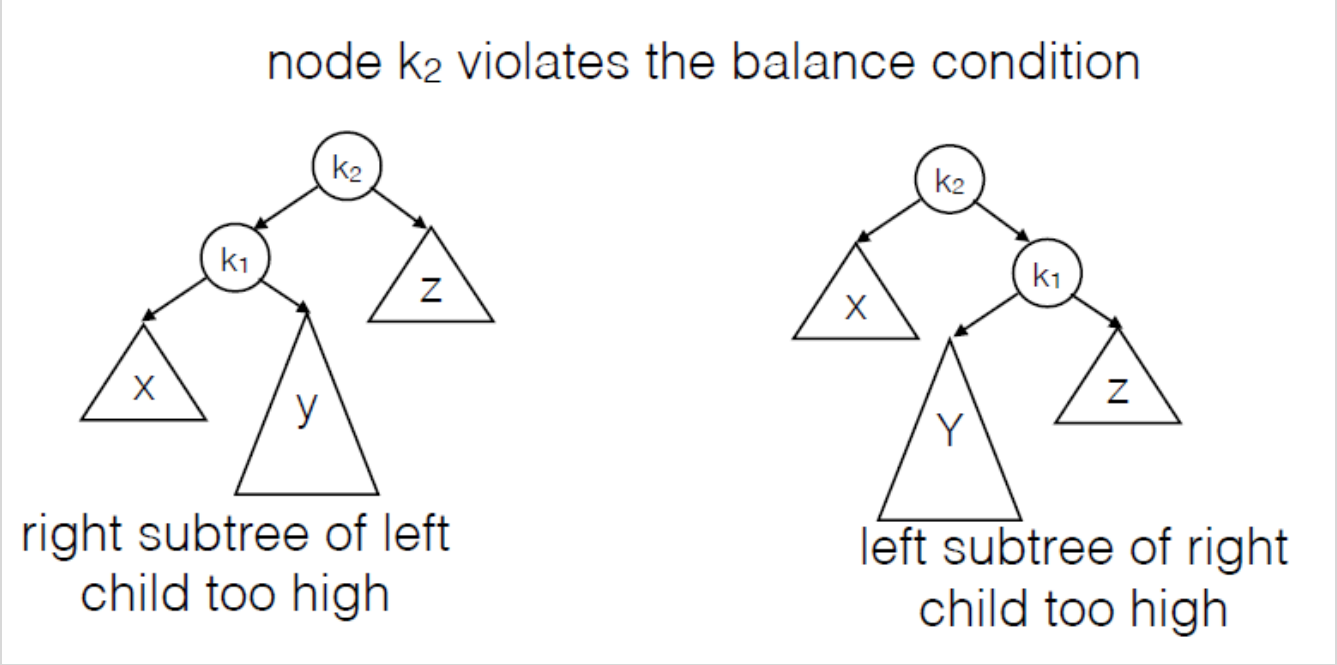


Case 2

- The second case, the result of inserting a node in the left subtree of the right child.
- A node is inserted into the tree in Figure (a); the resulting tree is shown in Figure (b) and in more detail in Figure (c). Note that *R*'s balance factor can also be –1.
- To bring the tree back into balance, a double rotation is performed. The balance of the tree *P* is restored by rotating *R* about node *Q* (Figure (d)) and then by rotating *R* again, this time about node *P* (Figure (e)).

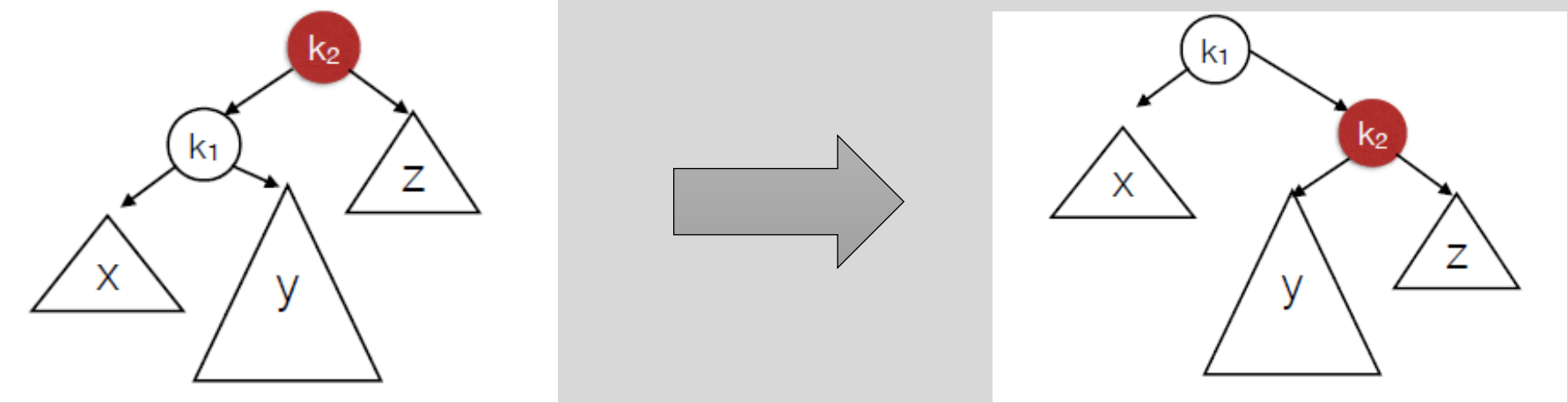


Rebalancing Trees with Double Rotation



Solution: Double Rotation

Single Rotation Does not work

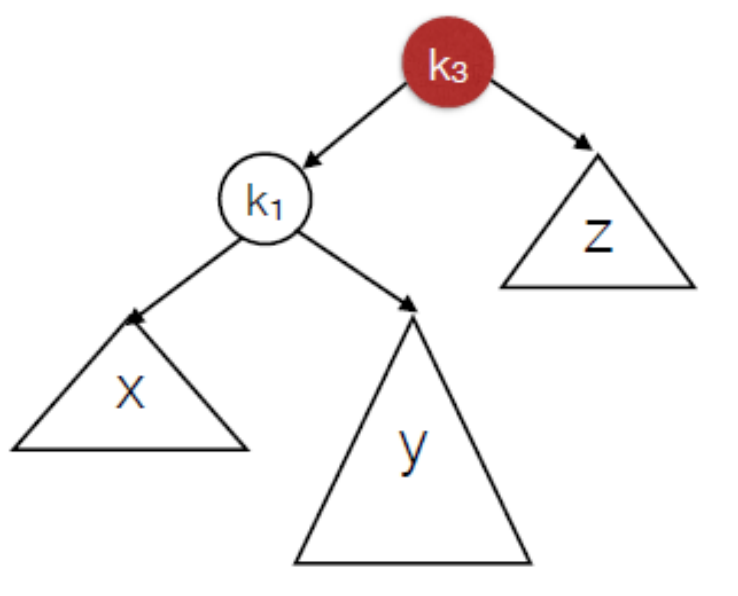


Result is not an AVL tree.

- Now k_1 is violates the balance condition.
- Problem: Tree y cannot move and it is too high.

Double Rotation

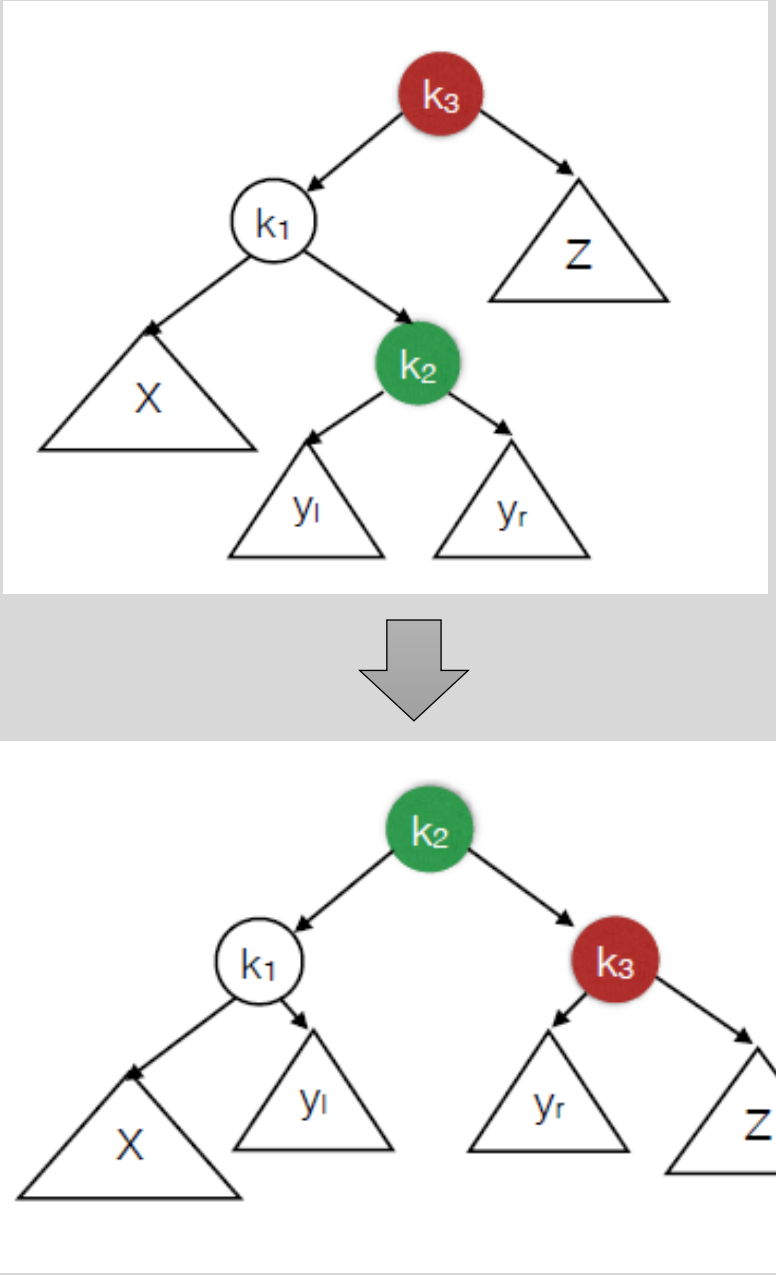
- y is non-empty (imbalance due to insertion into y or deletion from z)
- So we can view y as a root and two sub-trees.



Double Rotation

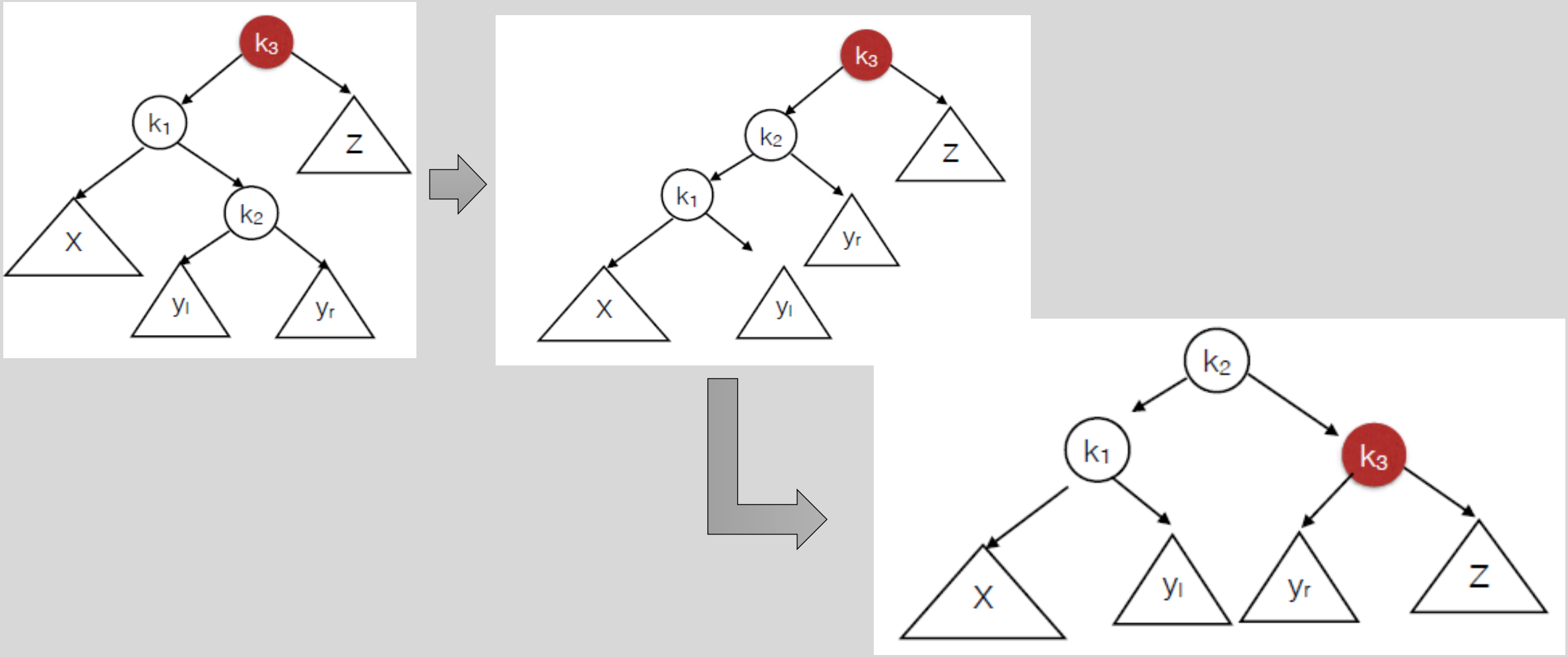
Maintain BST property:

- x is still left subtree of k_1.
- z is still right subtree of k_3.
- For all values v in y_l: $k_1 < v < k_2$
- So y_l becomes new right subtree of k_1.
- For all values w in y_r: $k_2 < w < k_3$
- So y_r becomes new left subtree of k_3.



Double Rotation

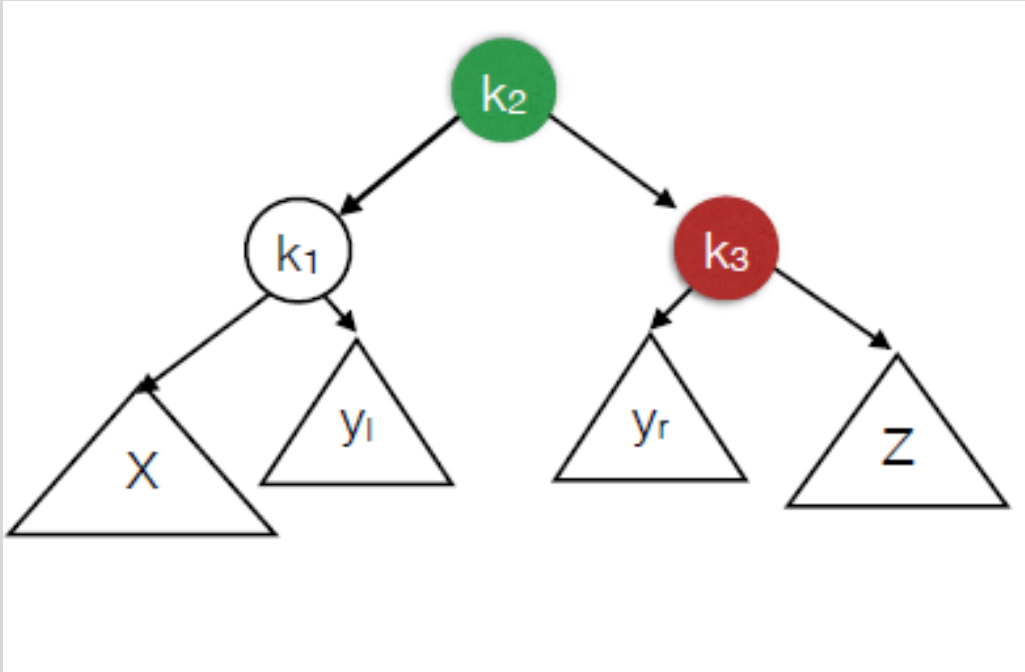
- These are actually two single rotations: First at k1, then at k3.



Double Rotation

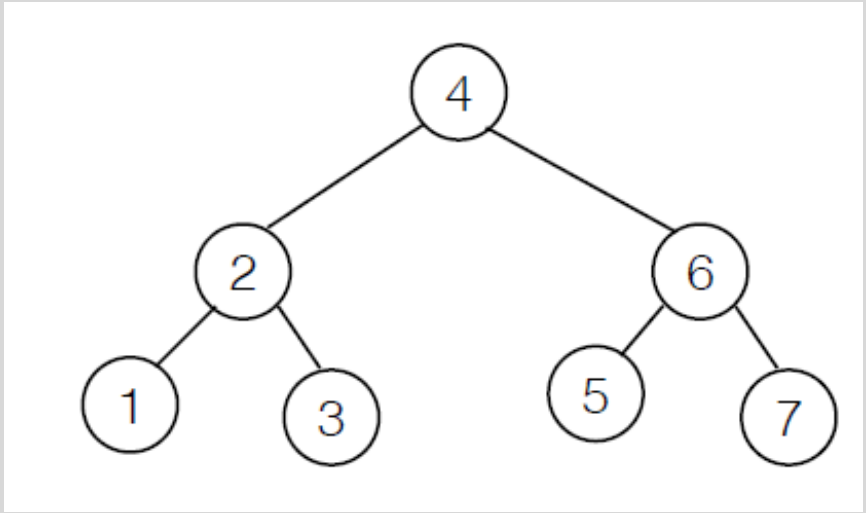
Modify 5 references:

- parent(k_3).left = k_2 or parent(k_3).right = k_2
- k_2.left = k_1
- k_2.right = k_3
- k_1.right = root(y_l)
- k_3.left = root(y_r)



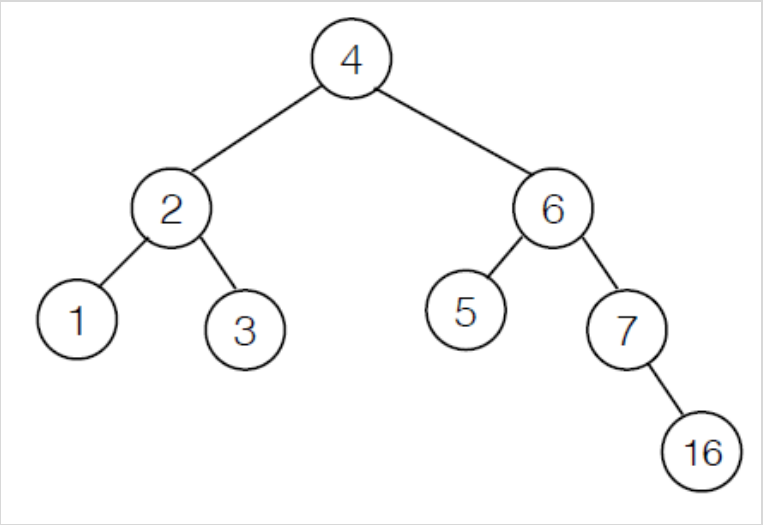
Double Rotation Example

Insert 16



Double Rotation Example

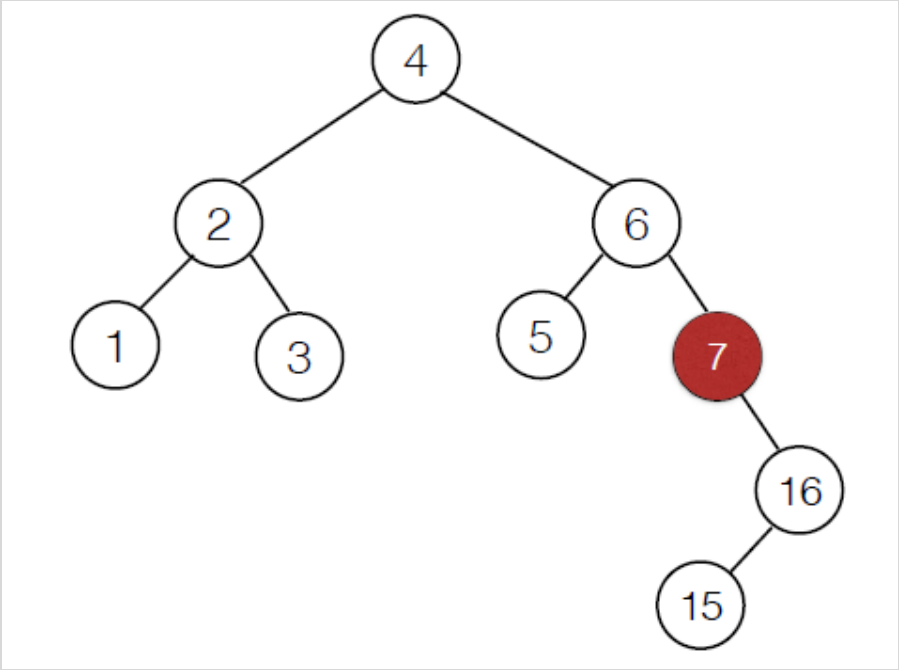
Insert 16



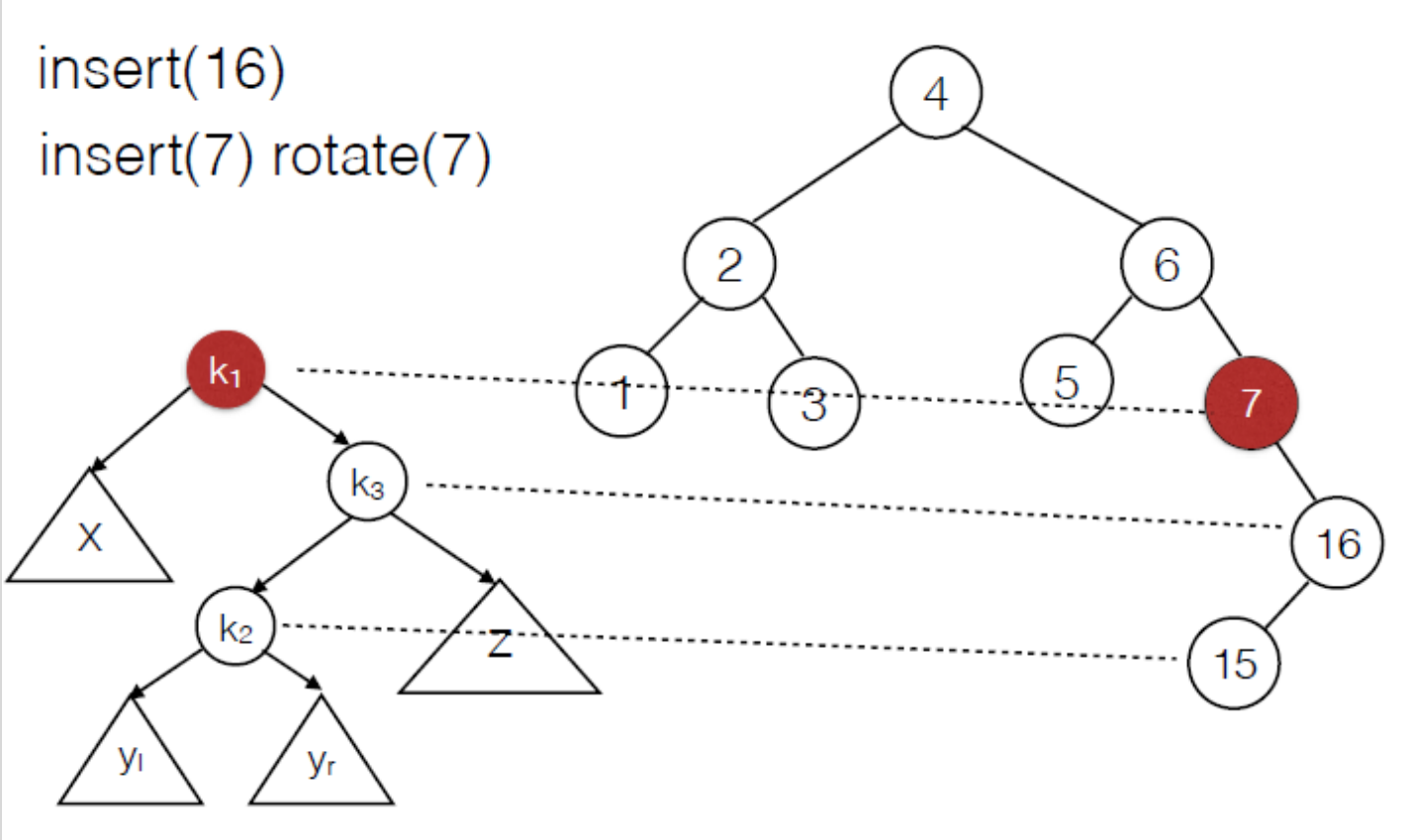
Double Rotation Example

Insert 15

Rotate 7



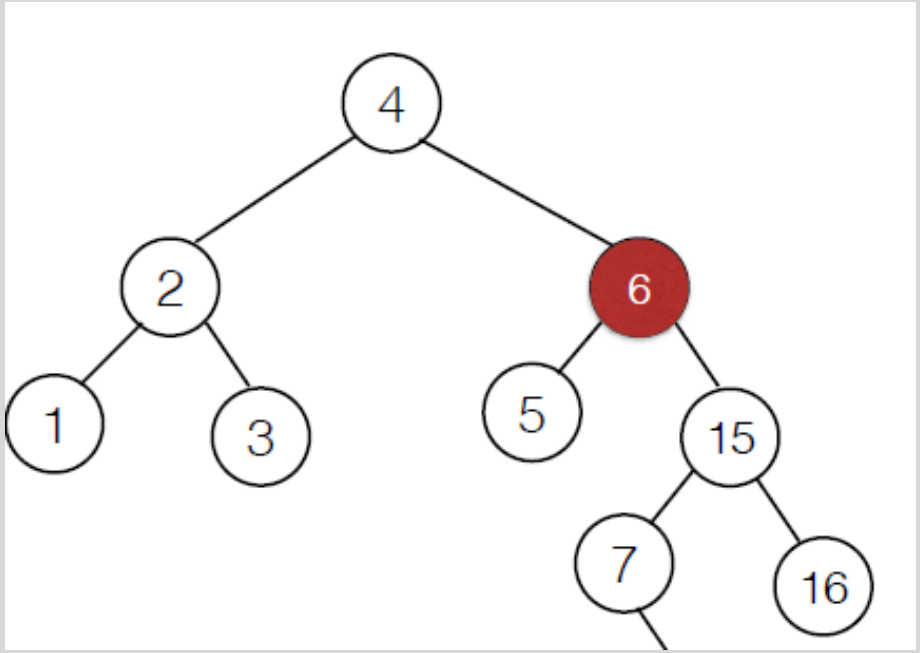
Double Rotation Example



k_1 and k_3 designations in the figure are interchanged in the figure. Correct it.

Double Rotation Example

Insert 14

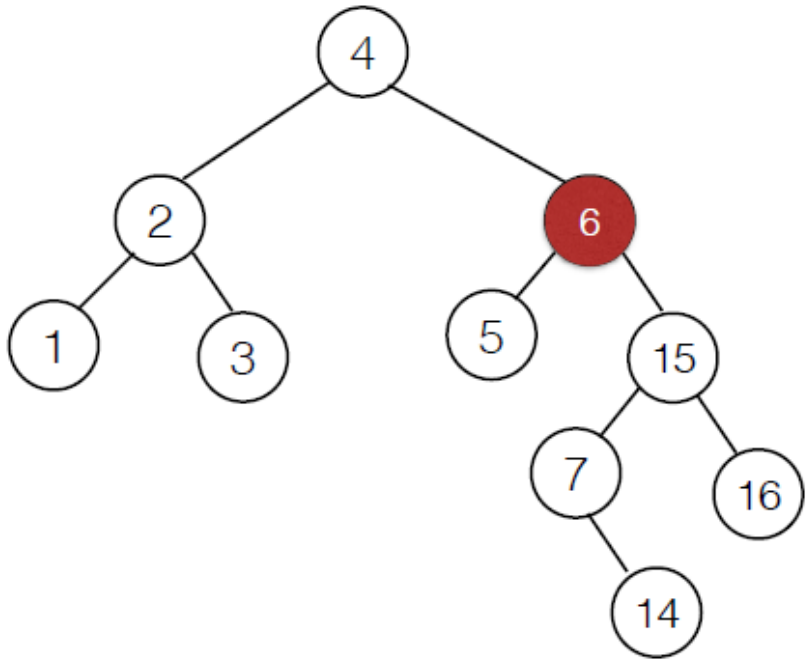


Double Rotation Example

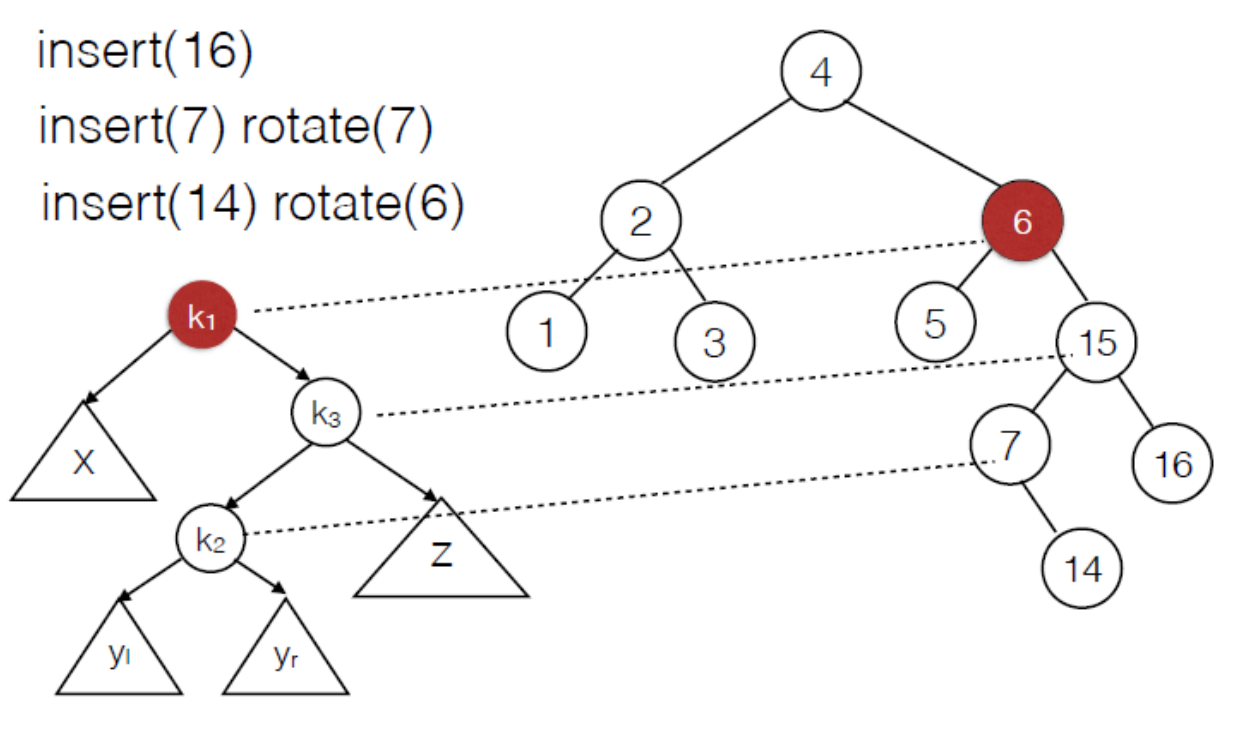
insert(16)

insert(7) rotate(7)

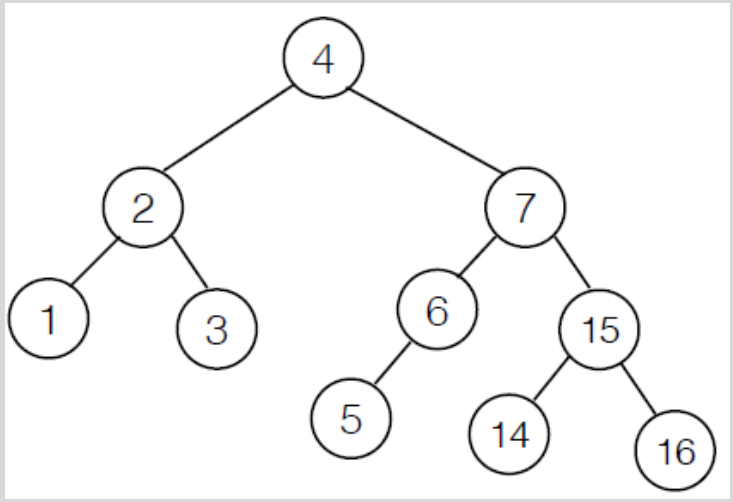
insert(14) rotate(6)



Double Rotation Example



Double Rotation Example



Deletion

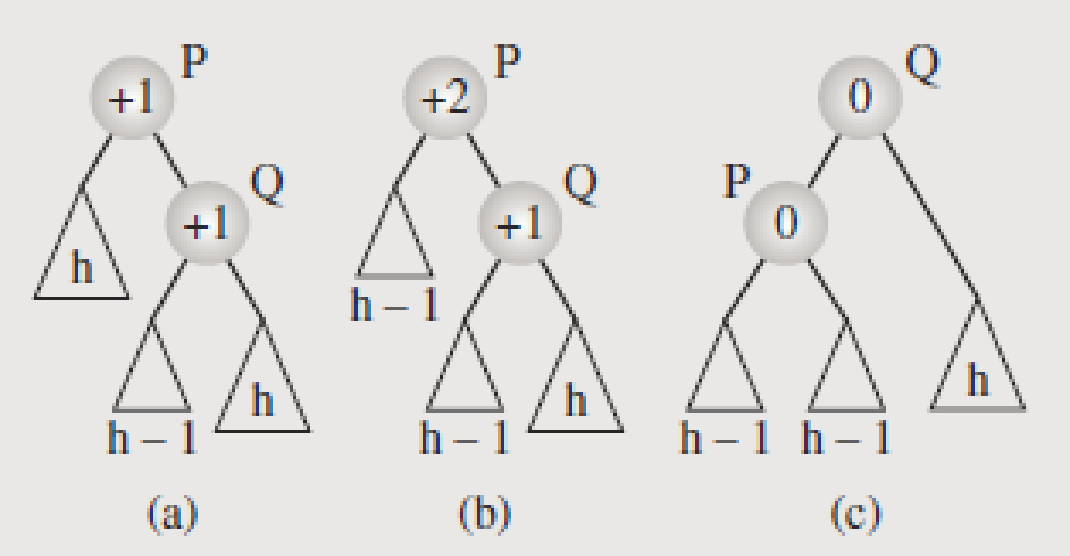
- Deletion may be more time-consuming than insertion. First, we apply deleteByCopying() to delete a node.
- After a node has been deleted from the tree, balance factors are updated from the parent of the deleted node up to the root.
- For each node in this path whose balance factor becomes ±2, a single or double rotation has to be performed to restore the balance of the tree.
- Importantly, the rebalancing does not stop after the first node *P* is found for which the balance factor would become ±2, as is the case with insertion.

Deletion

- This also means that deletion leads to at most $O(\lg n)$ rotations, because in the worst case, every node on the path from the deleted node to the root may require rebalancing.
- Deletion of a node does not have to necessitate an immediate rotation because it may improve the balance factor of its parent (by changing it from ±1 to 0).
- But it may also worsen the balance factor for the grandparent (by changing it from ±1 to ±2).
- We illustrate only those cases that require immediate rotation. There are four such cases (plus four symmetric cases). In each of these cases, we assume that the left child of node *P* is deleted.

Deletion: Case 1

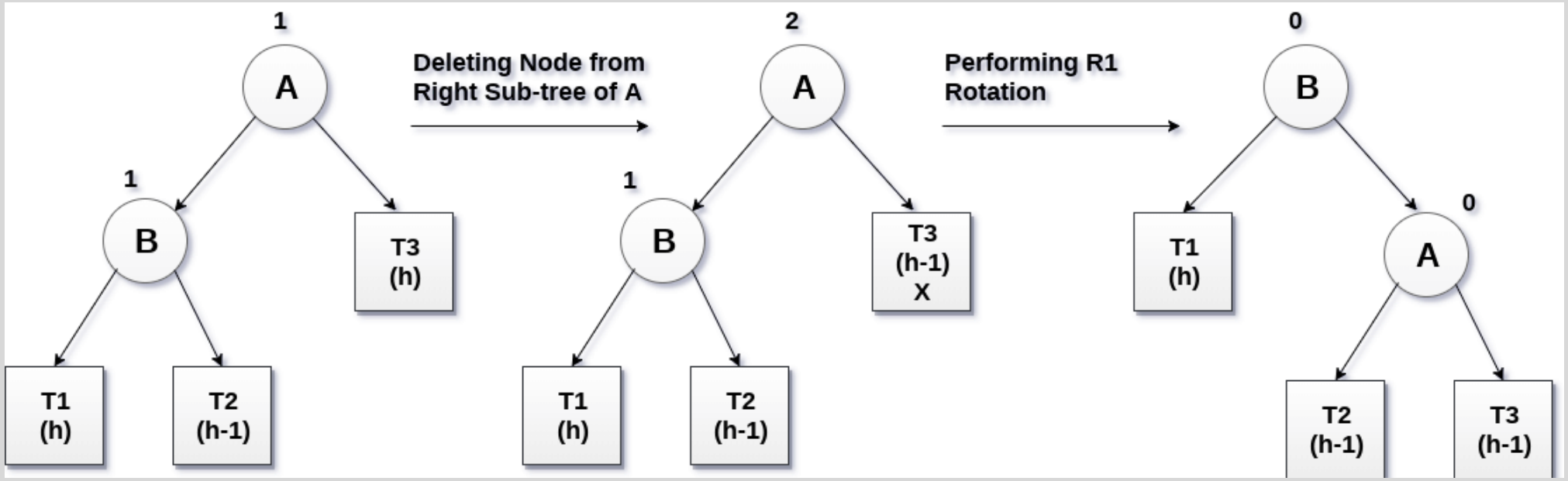
- In the first case, the tree in Figure (a) turns, after deleting a node, into the tree in Figure (b).
- The tree is rebalanced by rotating *Q* about *P* (Figure (c)).
- Case: Balance Factor of Q is +1.



Case 1 for Right Subtree

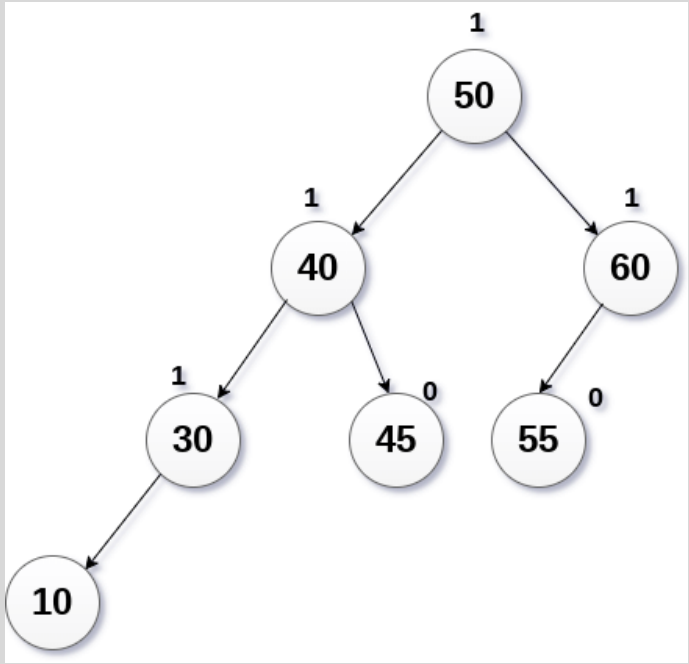
For recognizing deletion cases, compute balance factor as (Left subtree – Right subtree).

- R1 Rotation (Node B has balance factor 1)
- R1 Rotation is to be performed if the balance factor of Node B is 1. In R1 rotation, the critical node A is moved to its right having sub-trees T2 and T3 as its left and right child respectively. T1 is to be placed as the left sub-tree of the node B.

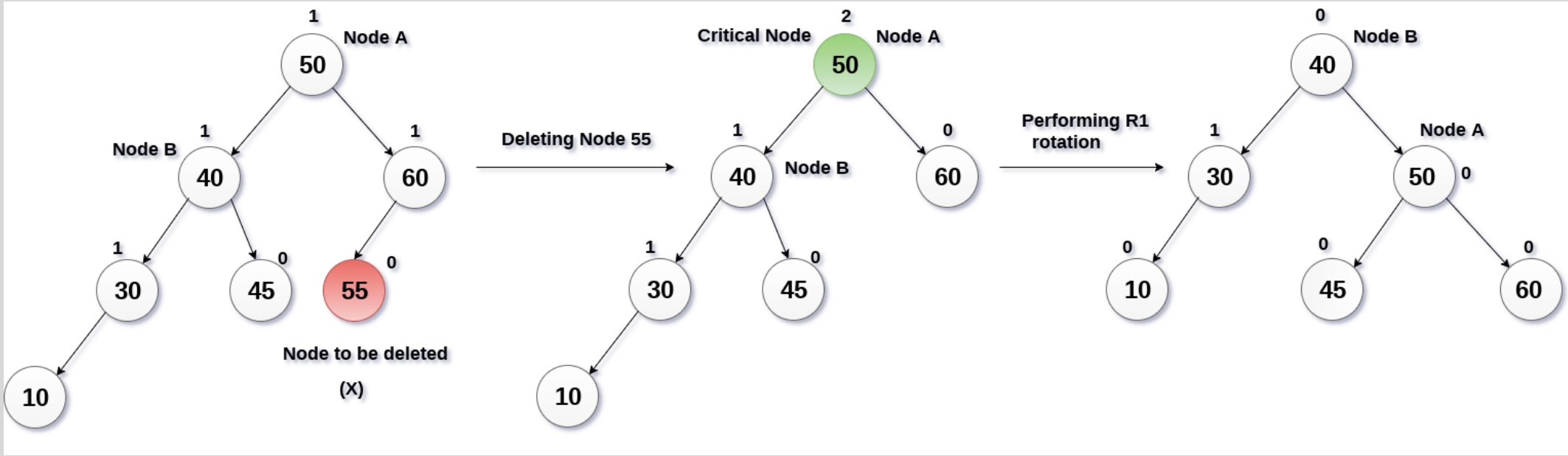


Example

Delete Node 55.

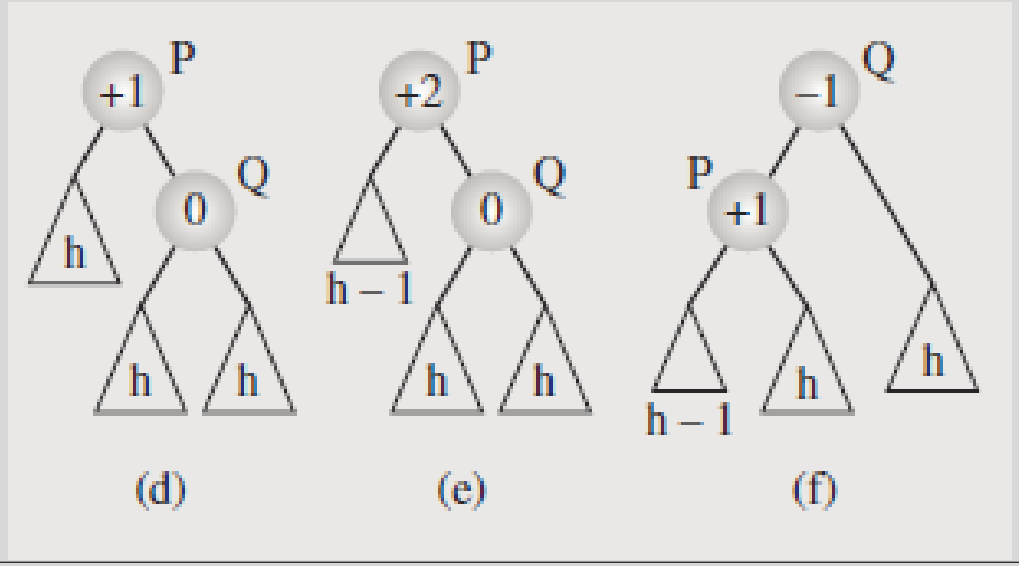


Example


1
Node A
50
Node B
1
1
40
60
1
0
0
30
45
55
10
Node to be deleted
(X)
Deleting Node 55
2
Critical Node
Node A
50
1
0
40
Node B
60
1
0
30
45
10
Performing R1
rotation
0
Node B
40
1
Node A
30
50
0
0
0
0
10
45
60

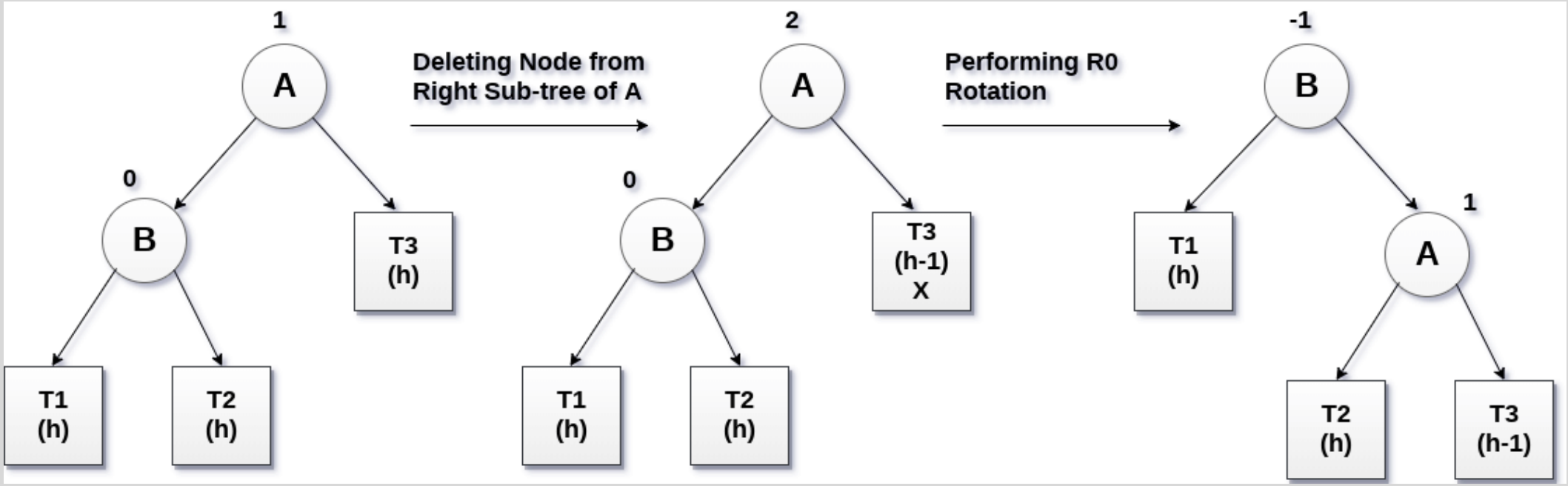
Deletion: Case 2

- Case: Balance Factor of Q is 0.
- In the second case, *P* has a balance factor equal to +1, and its right subtree *Q* has a balance factor equal to 0 (Figure (d)). After deleting a node in the left subtree of *P* (Figure (e)), the tree is rebalanced by the same rotation as in the first case (Figure (f)).
- In this way, cases one and two can be processed together in an implementation after checking that the balance factor of *Q* is +1 or 0.



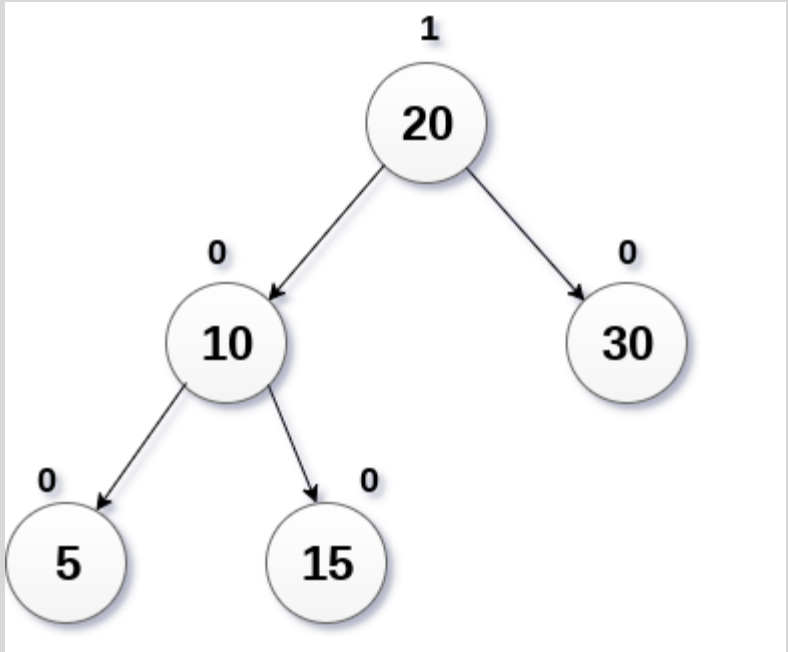
Case 2 for Right Subtree

- R0 rotation (Node B has balance factor 0)
- If the node B has 0 balance factor, and the balance factor of node A disturbed upon deleting the node X, then the tree will be rebalanced by rotating tree using R0 rotation. The critical node A is moved to its right and the node B becomes the root of the tree with T1 as its left sub-tree. The sub-trees T2 and T3 becomes the left and right sub-tree of the node A.

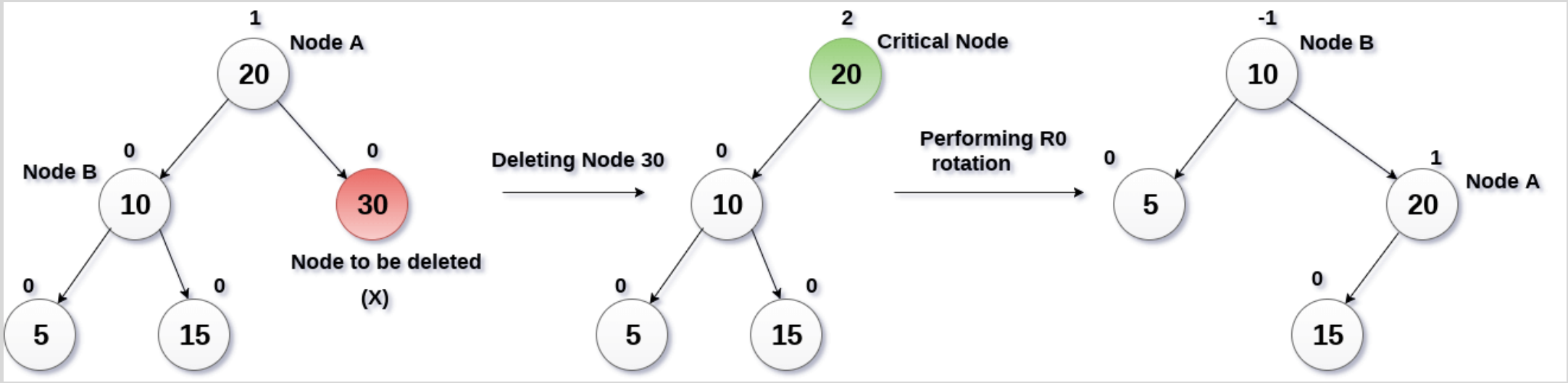


Example

Delete Node 30.

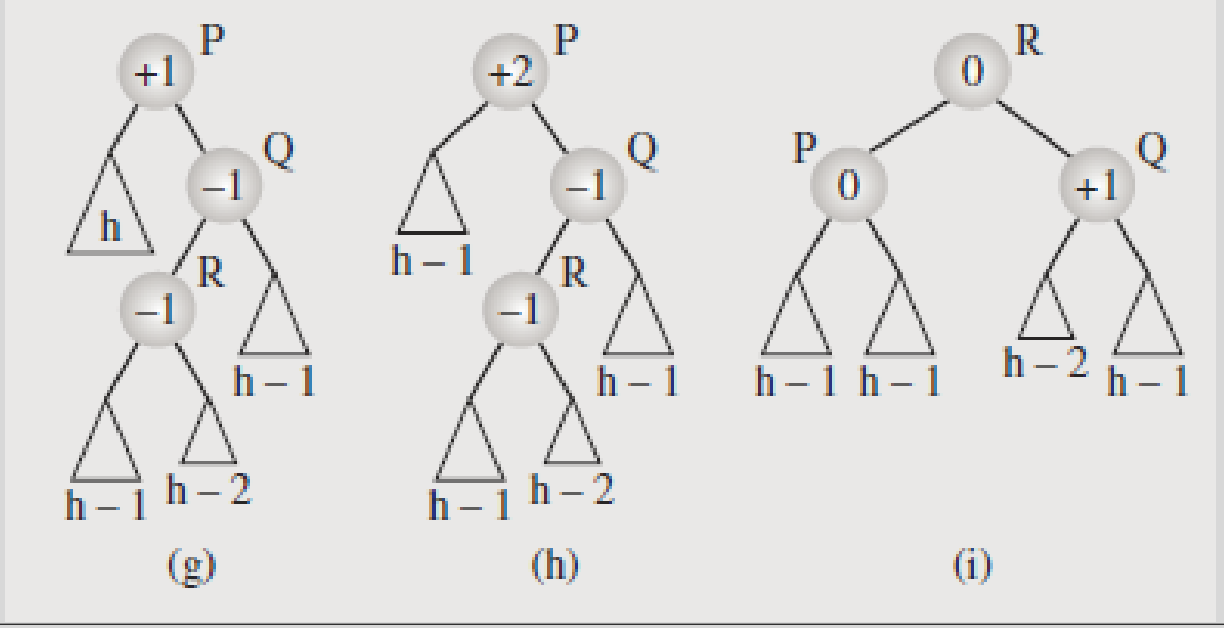


Example


1
Node A
20
Node B
0
10
0
30
Node to be deleted
(X)
0
5
0
15
Deleting Node 30
2
Critical Node
20
0
10
0
5
0
15
Performing R0
rotation
-1
Node B
10
0
5
1
Node A
20
0
15

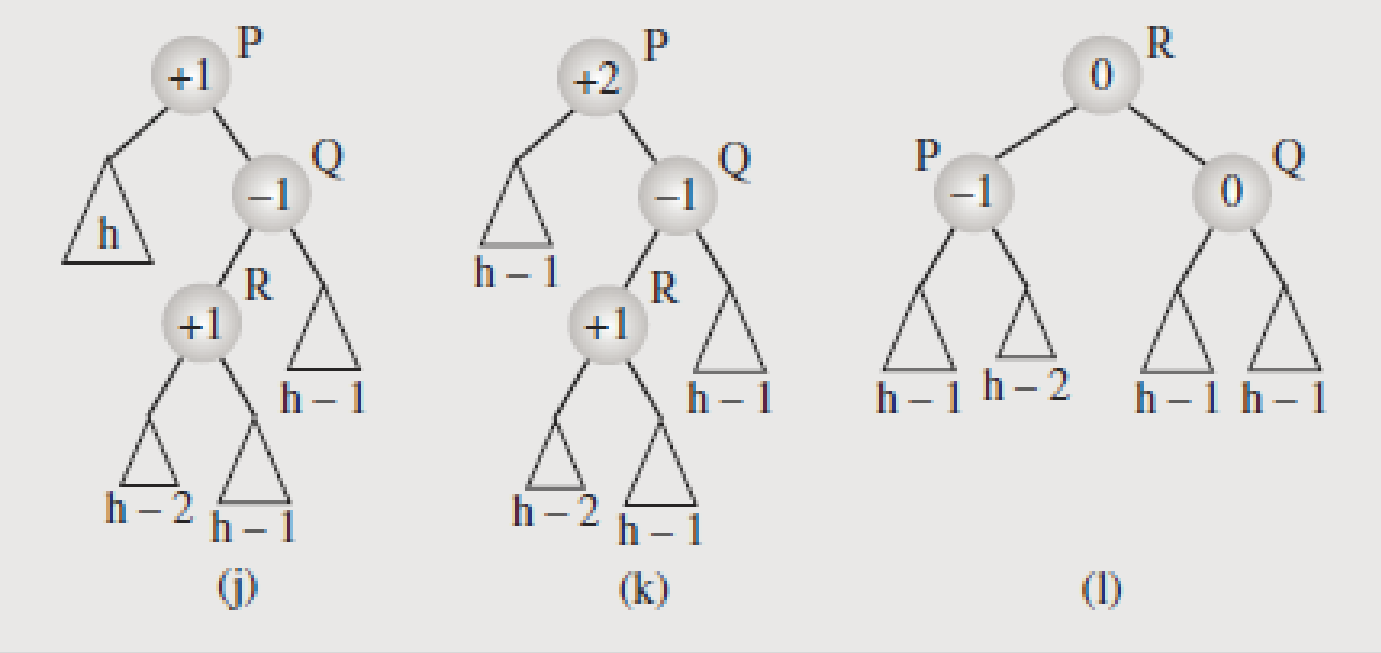
Deletion: Case 3

- If *Q* is –1, we have two other cases, which are more complex.
- In the third case, the left subtree *R* of *Q* has a balance factor equal to –1 (Figure (g)).
- To rebalance the tree, first *R* is rotated about *Q* and then about *P* (Figures h–i).



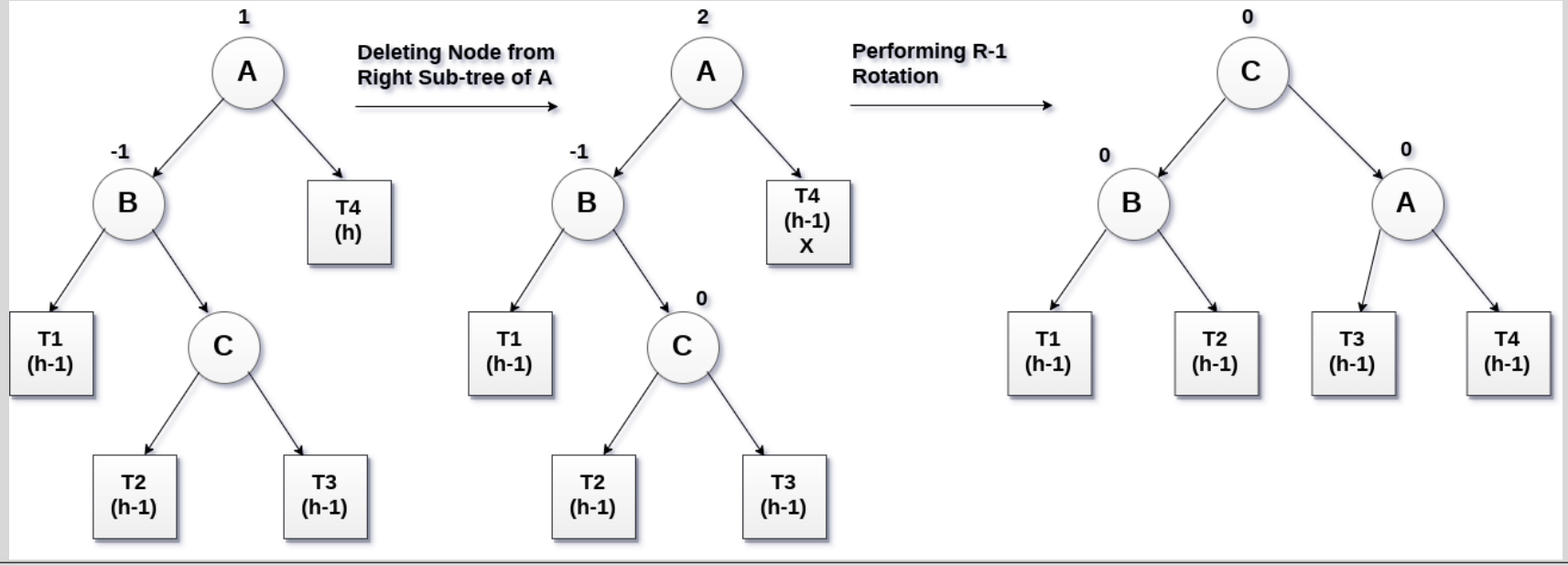
Deletion: Case 4

- The fourth case differs from the third in that *R*'s balance factor equals +1 (Figure (j)), in which case the same two rotations are needed to restore the balance factor of *P* (Figures 6.45k–l).
- Cases three and four can be processed together in a program processing AVL trees.



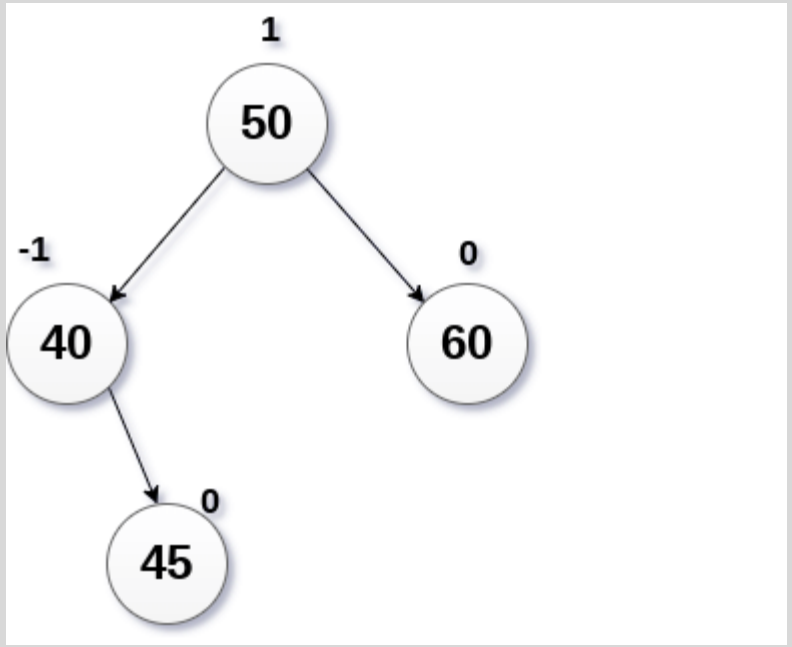
Case 3 & 4 for Right Subtree

- R-1 Rotation (Node B has balance factor -1)
- R-1 rotation is to be performed if the node B has balance factor -1. In this case, the node C, which is the right child of node B, becomes the root node of the tree with B and A as its left and right children respectively (LR Rotation).
- The sub-trees T1, T2 becomes the left and right sub-trees of B whereas, T3, T4 become the left and right sub-trees of A.

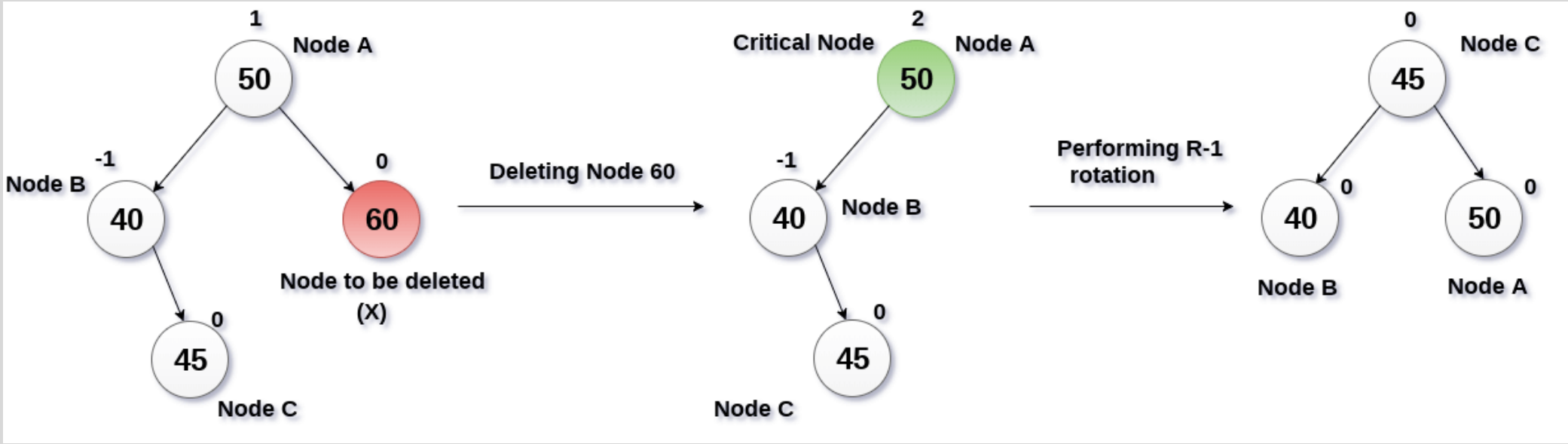


Example

Delete Node 60.



Example


1
Node A
50
-1
Node B
40
0
60
Node to be deleted
(X)
0
45
Node C
Deleting Node 60
2
Critical Node
Node A
50
-1
40
Node B
0
45
Node C
Performing R-1
rotation
0
Node C
45
0
40
0
50
Node B
Node A