AVL tree (Adelson-Velskii-Landis)

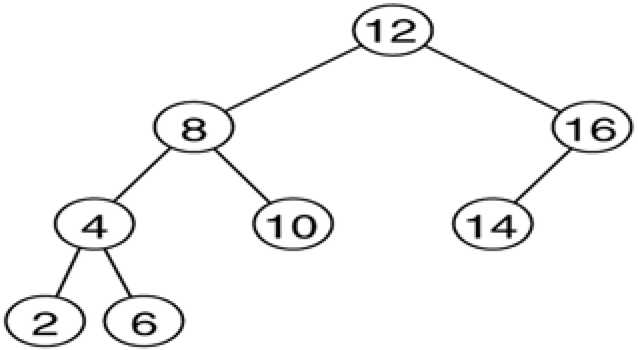
- AVL is named for its inventors: **A**del’son-**V**el’skii and **L**andis

Definition: An AVL tree is a **binary search tree**. For any node in the tree, the **height** of the left and right subtrees can differ by **at most 1**.

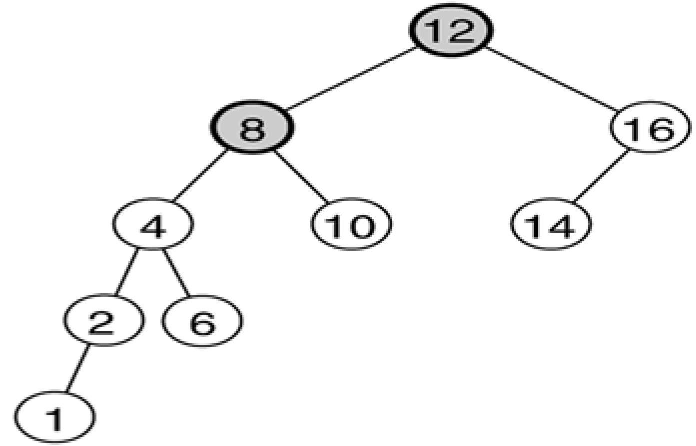
. The balance factor **bf(x) = height(left) – height(right)**

- bf(x) values **-1, 0, and 1** are allowed. **(AVL tree)**
- If bf(x) < -1 or bf(x) > 1 then tree is **NOT AVL tree**

- Take **O(log n)** time for searching, insertion, and deletion
- Search: same as BST, (delete and insertion breaks AVL tree)

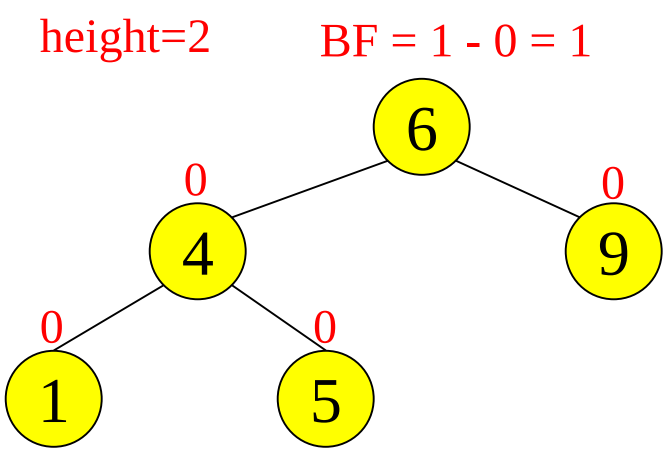


a) AVL tree,

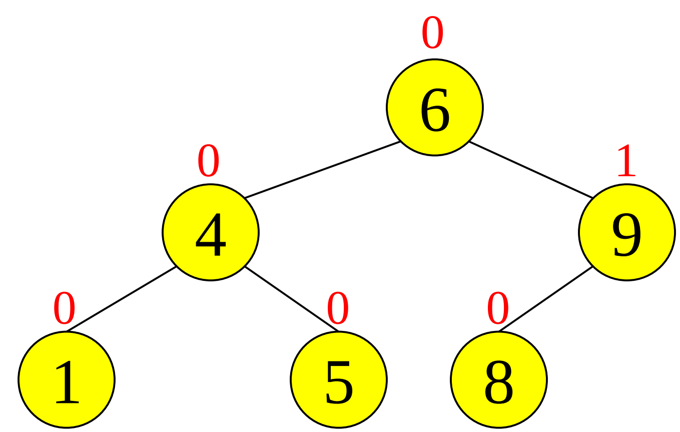


b) not an AVL tree

Tree A (AVL)

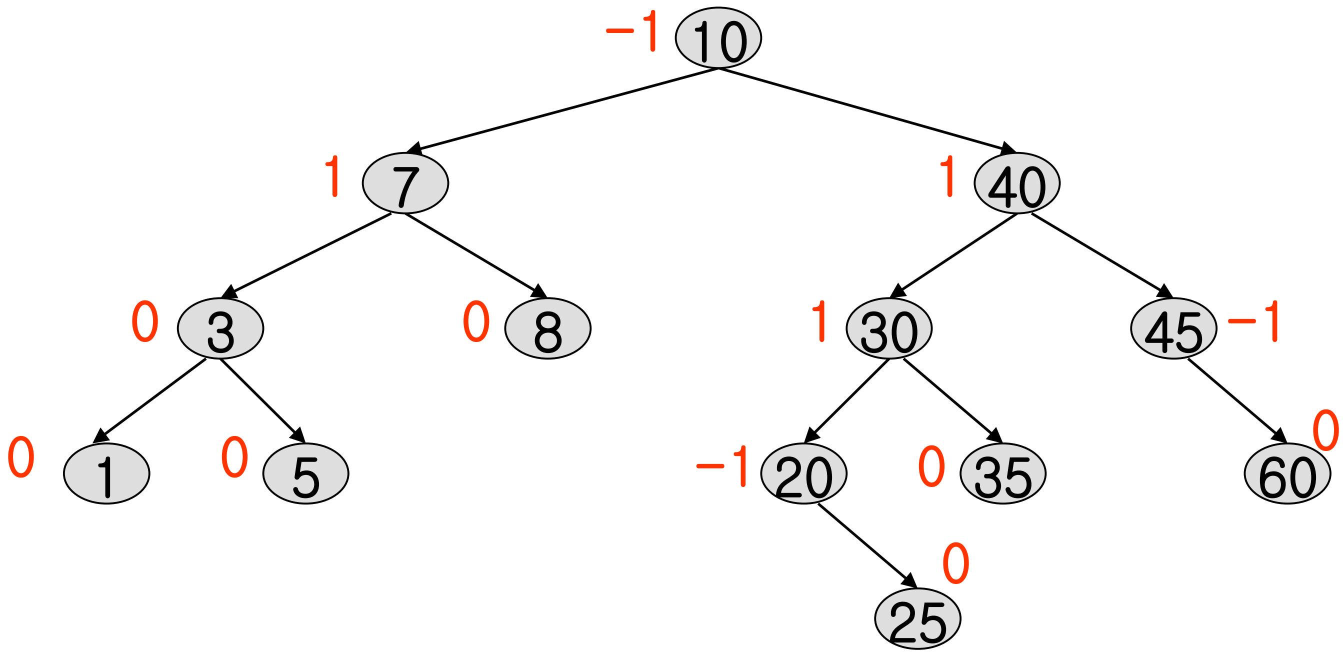


Tree B (AVL)



balance factor = $h_{left}-h_{right}$

AVL Tree with Balance Factors



- Is this an AVL tree?
- What is the balance factor for each node in this AVL tree?

- Insert (9) ➔ where is 9 going to be inserted?
- After insertion, is the tree still an AVL tree? (still balanced ?)

Rebalancing

- After an insertion, when the balance factor of node A is –2 or 2, the node A is one of the following four imbalance types

* Outside cases (<u>Require single rotation</u> – LL and RR)

1. An insertion in the **left subtree** of **the left** child of X, (LL)
2. An insertion in the **right subtree** of the **right child** of X. (RR)

* Inside Cases (<u>Require double rotation</u> – RL and LR)

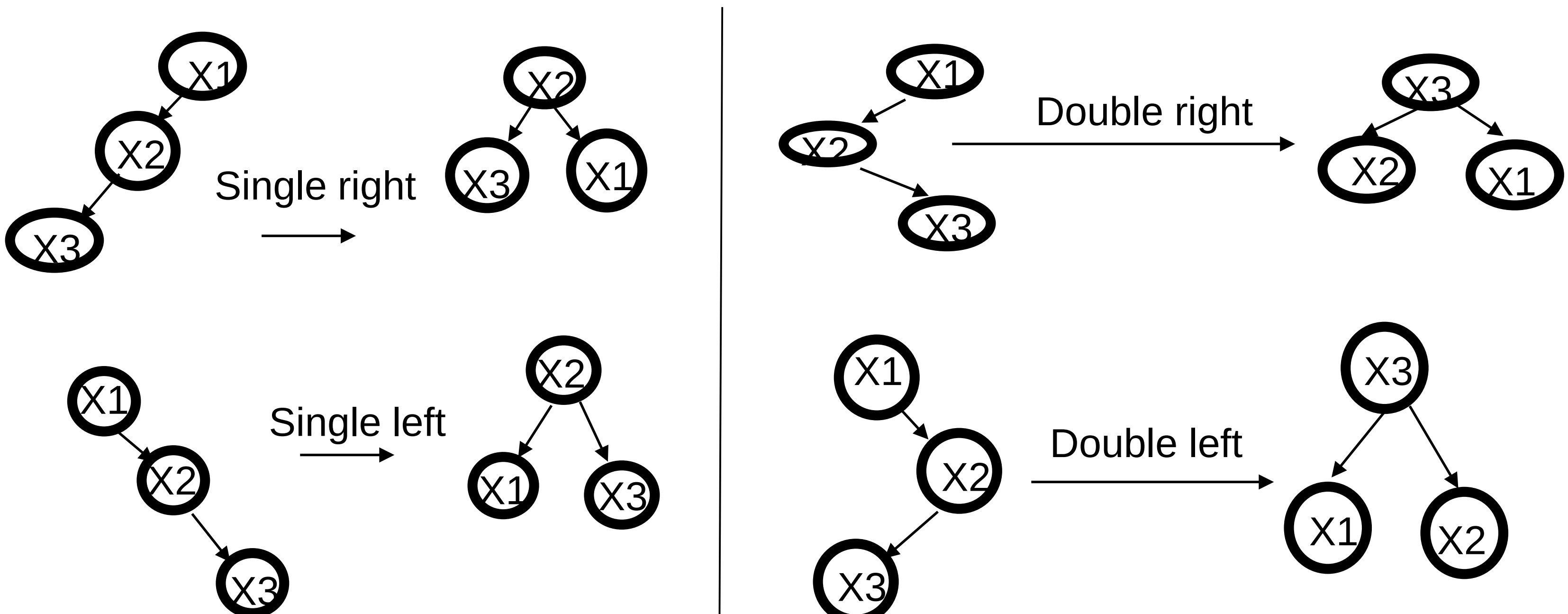
1. An insertion in the **right subtree** of the **left child** of X, (RL)
2. An insertion in the **left subtree** of the **right child** of X, (LR)

- Balance is restored by these *rotations*

AVL Balancing Operations: Rotations

Definition for Rotations

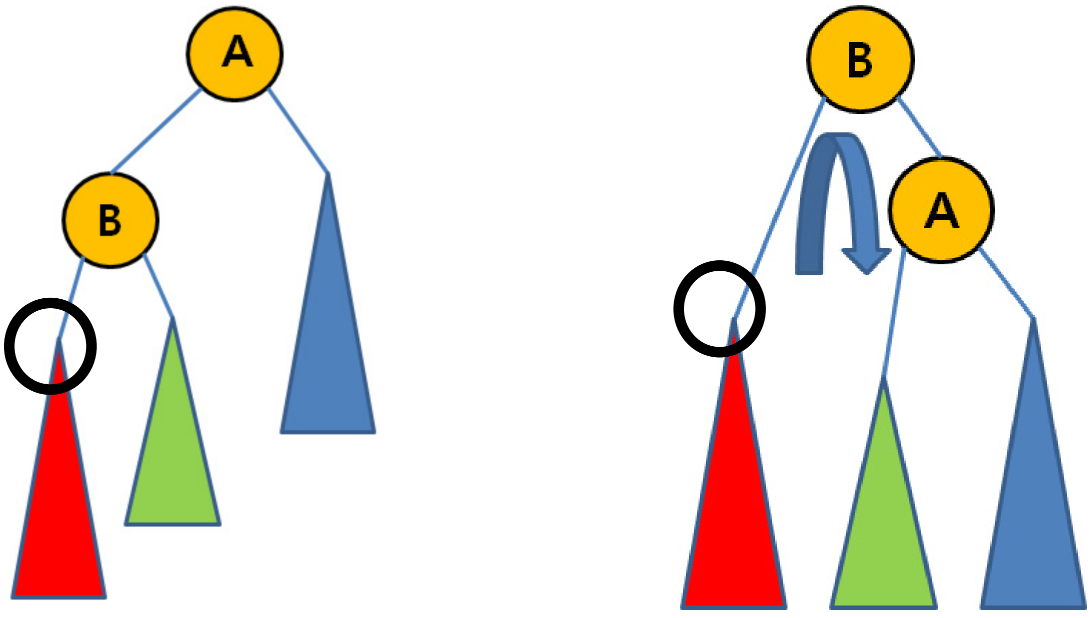
- To switch *children* and *parents* among two or three adjacent nodes to restore balance of a tree.
- A rotation may change the depth of some nodes, but does not change their relative ordering.



1) LL rotation (insertion in the **left subtree** of **the left** child of X)

* Right Rotation

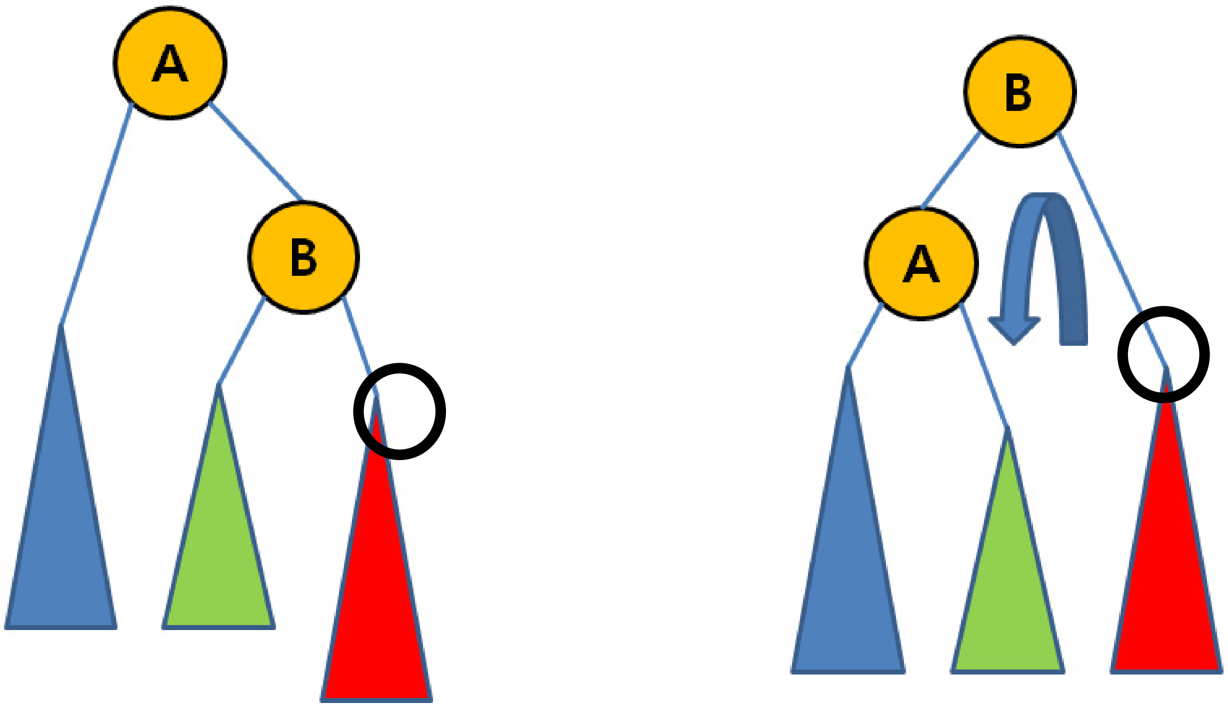
```
Node* rotateLL(Node *A)
{
    Node *B = A->left;
    A->left = B->right;
    B->right=A;
    A = B
}
```



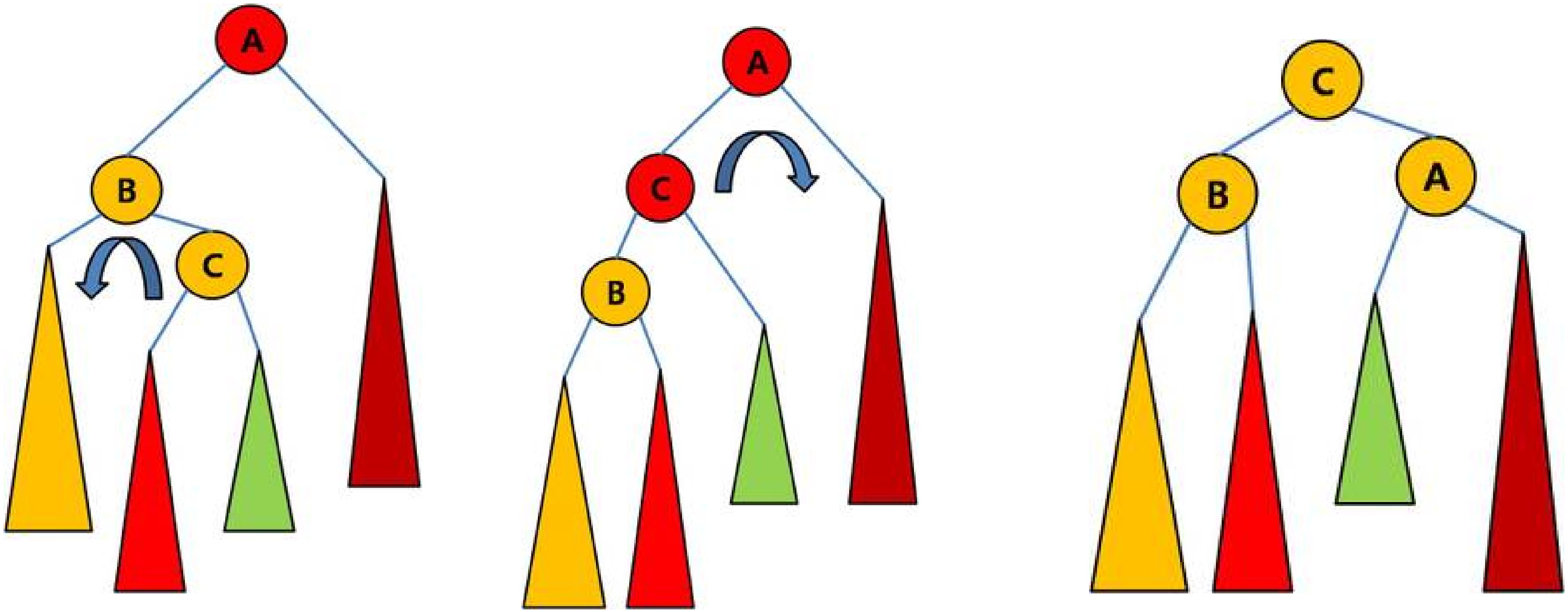
2) RR rotation (insertion in the **right subtree** of the **right child** of X)

```
Node* rotateRR(Node *A)
{
        Node *B = A->right;
        A->right = B->left;
        B->left = A;
        return B;
}
```

* Left Rotation

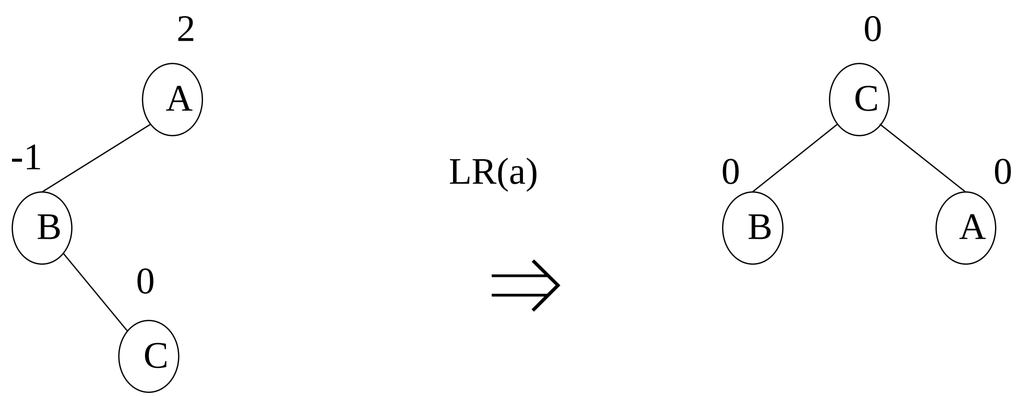


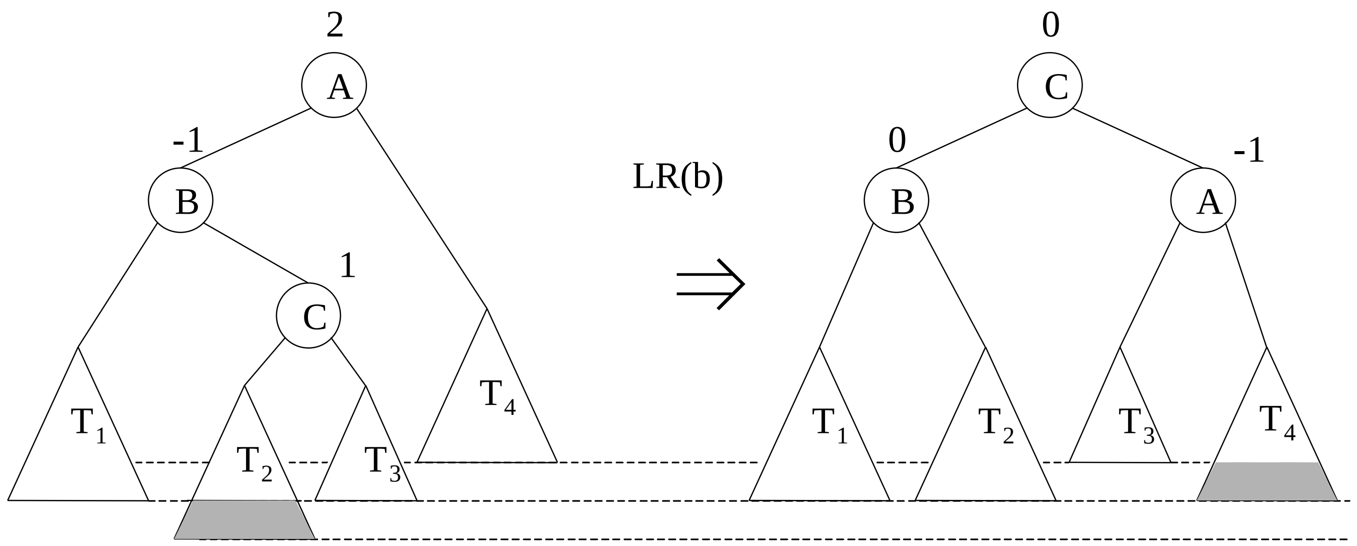
3) LR rotation (insertion in the **left subtree** of the **right child** of X)



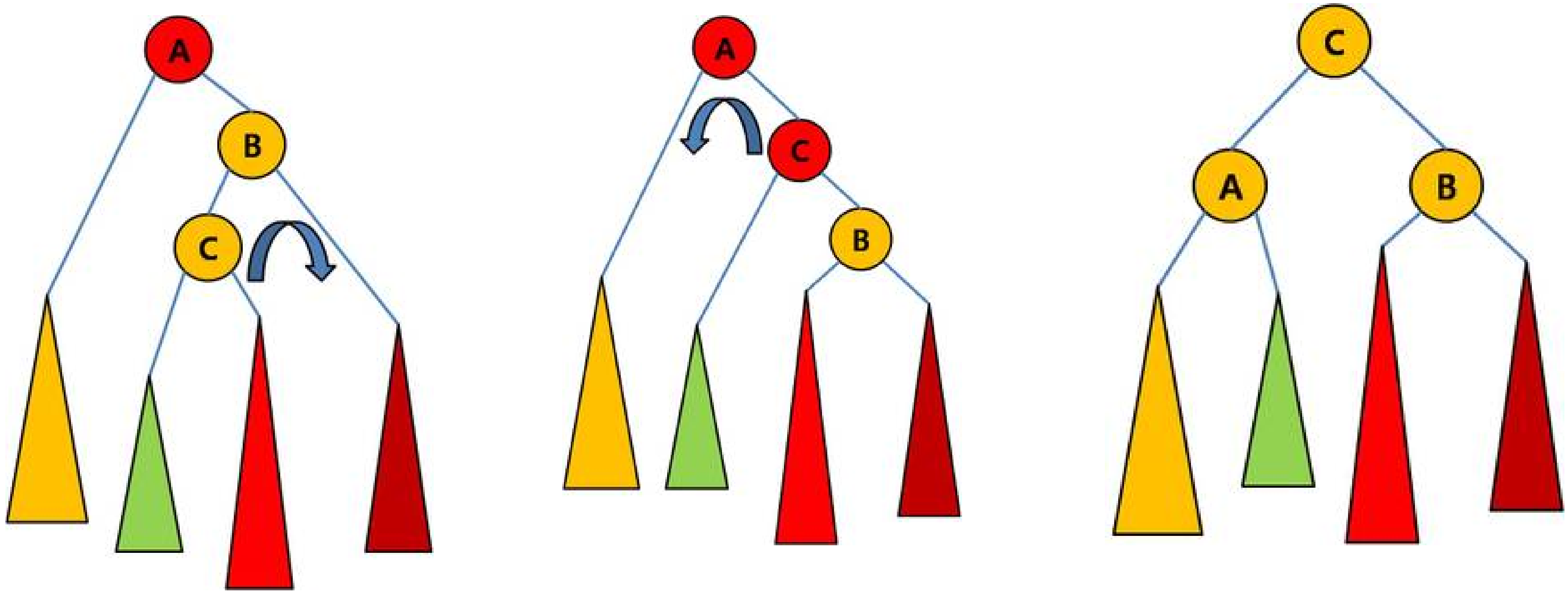

```
Node* rotateLR(Node *A)
{
        Node *B = A->left;
        A->left = rotateRR(B);
        return rotateLL(A);
}
```

LR rotation



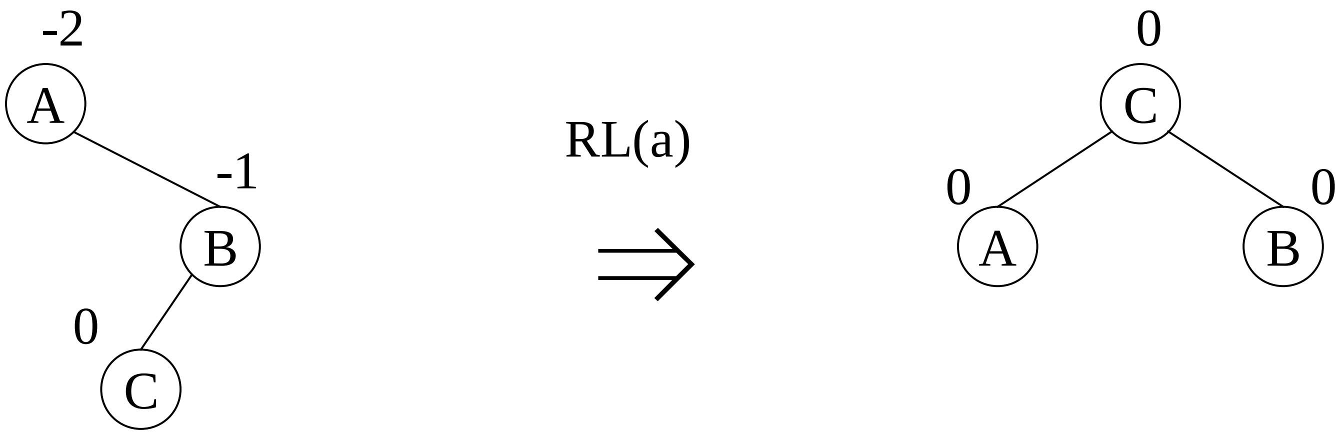


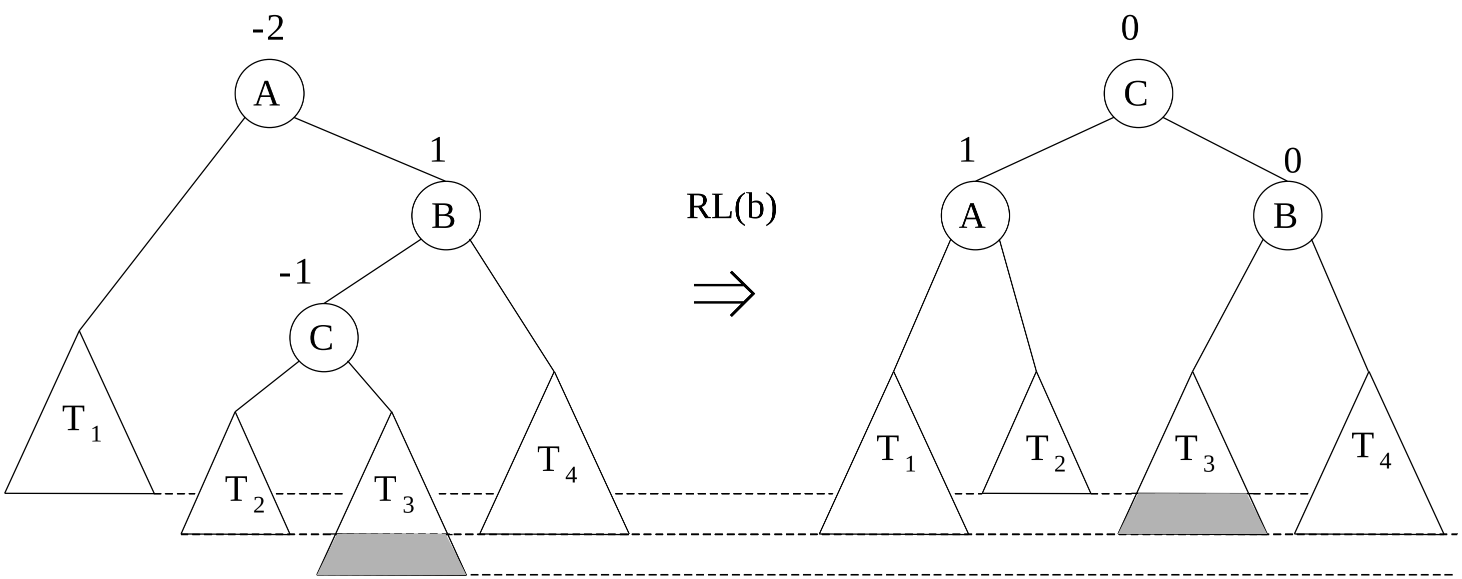
4) RL rotation (insertion in the **right subtree** of the **left child** of X)




```
Node* rotateRL(Node *A)
{
	Node *B = A->right;
	A->right = rotateLL(B);
	return rotateRR(A);
}
```

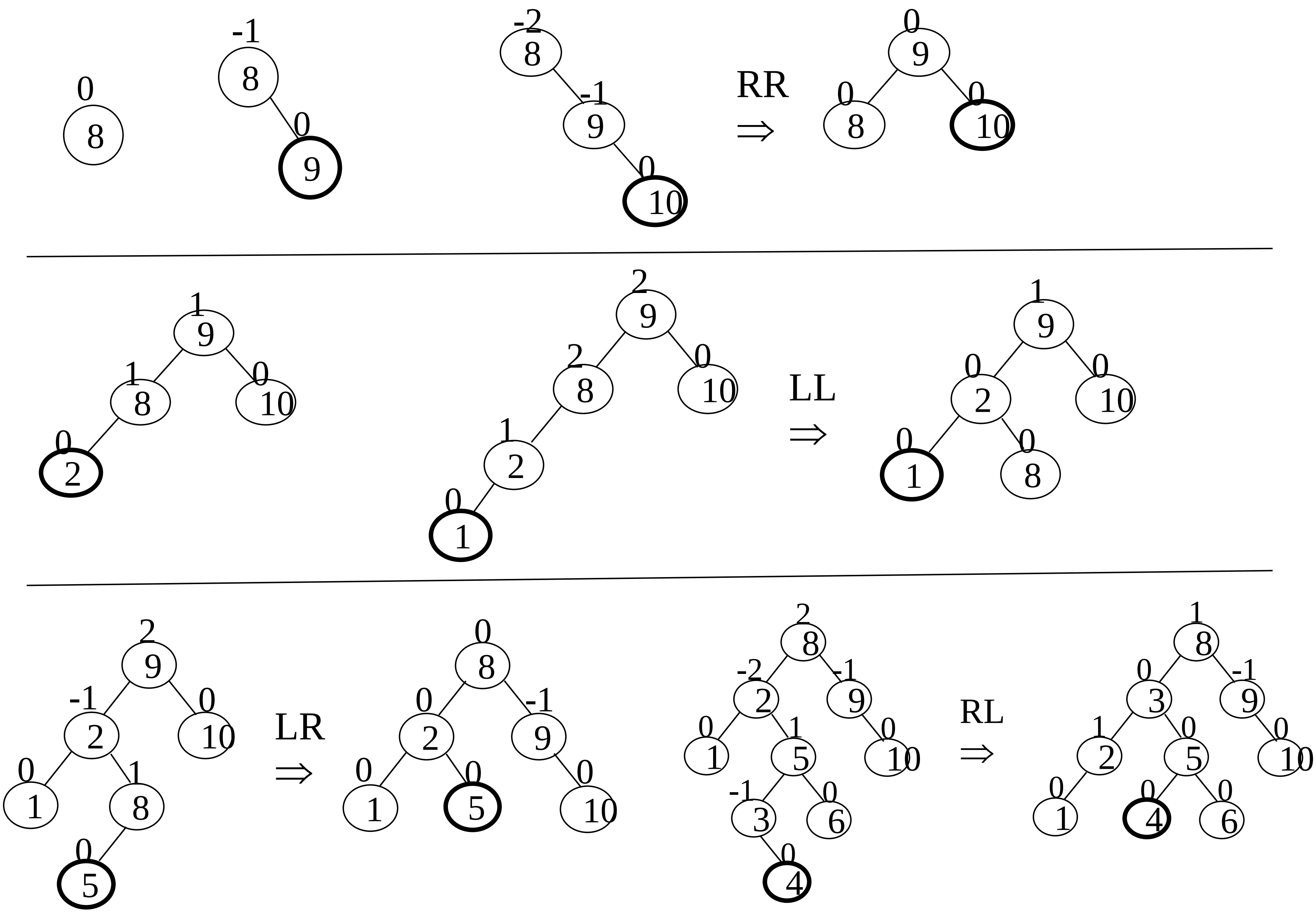
RL rotation

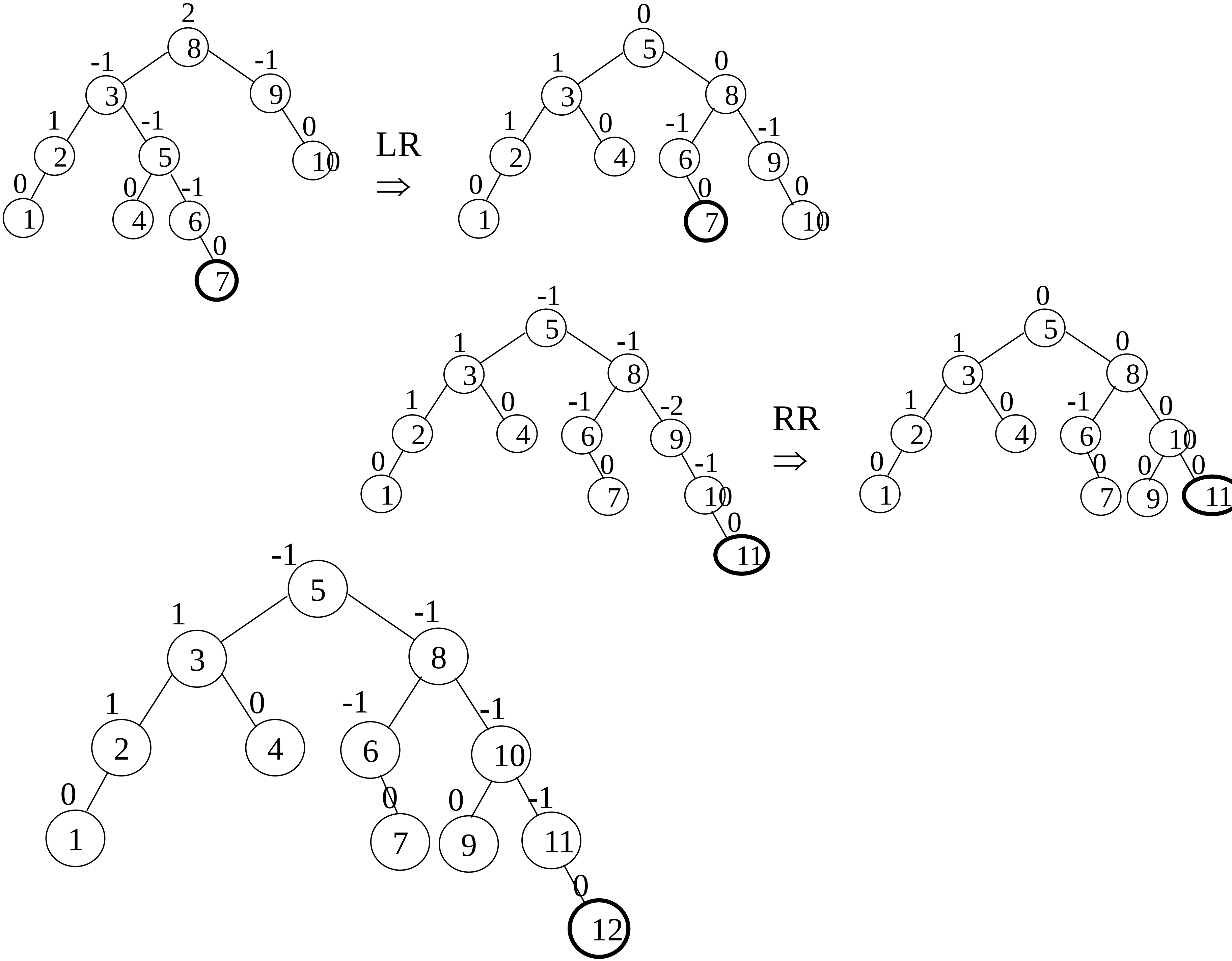




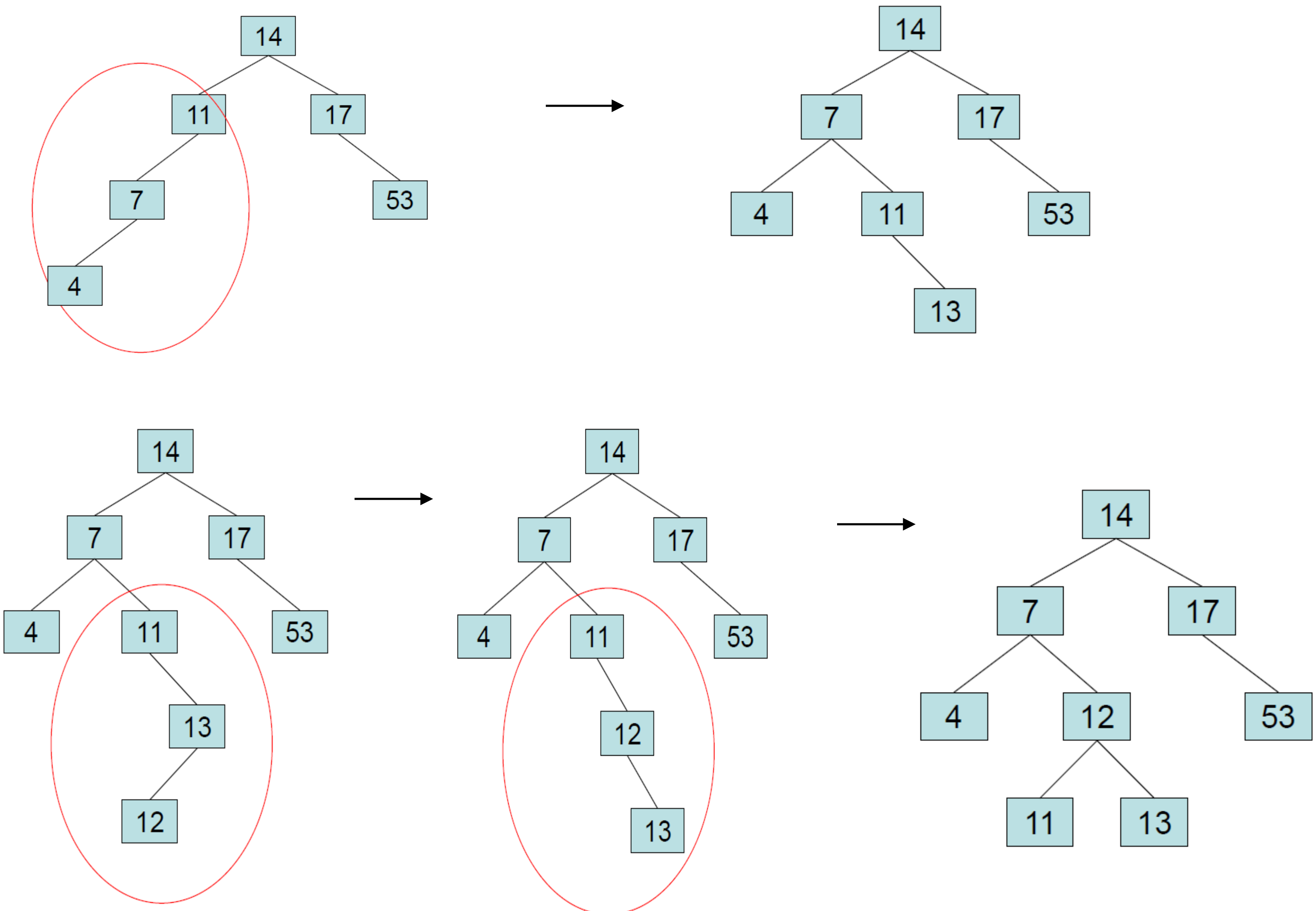
example

Insert (8, 9, 10, 2, 1, 5, 3, 6, 4,7,11,12) and build AVL tree



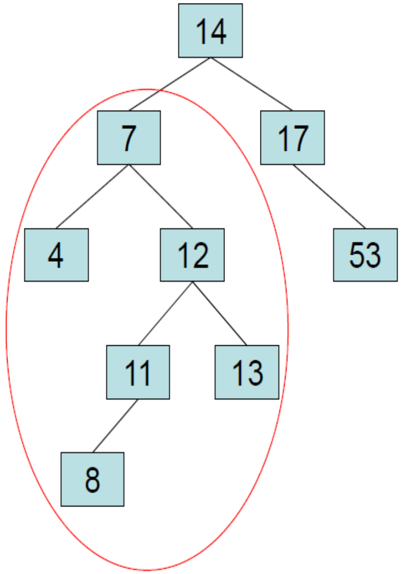

LR
⇒
RR
⇒

AVL Tree Example 1: Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree

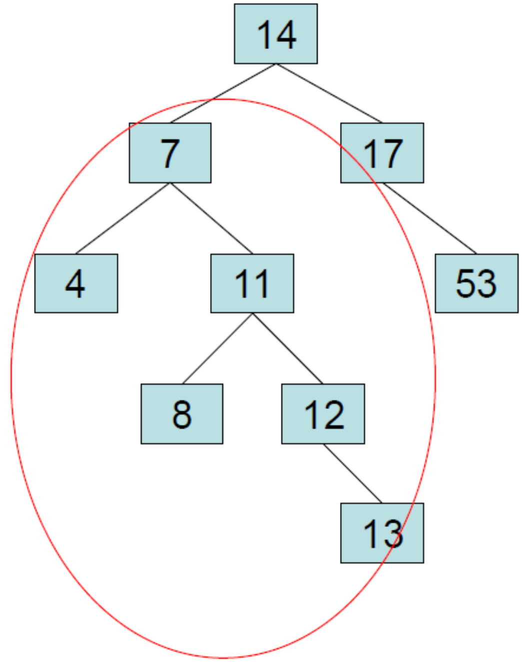


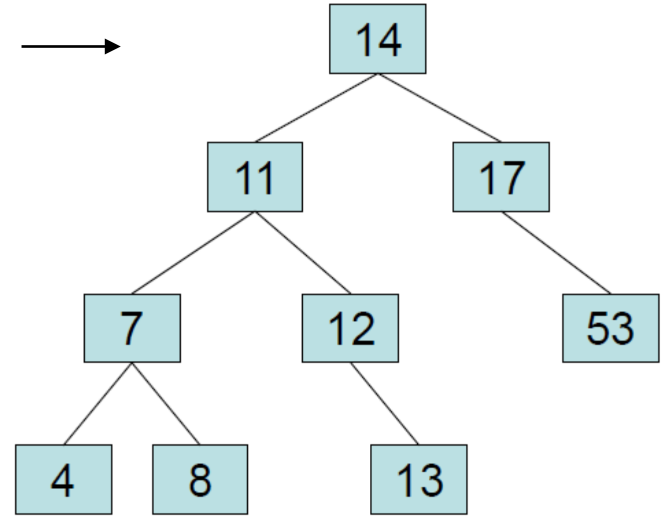
insert 12

AVL tree is balanced

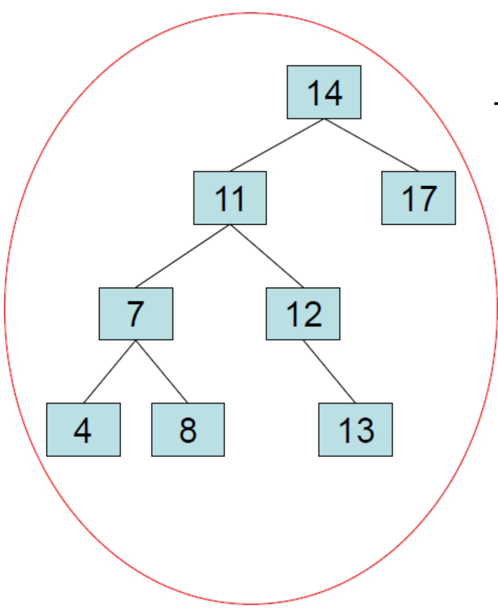


insert 8

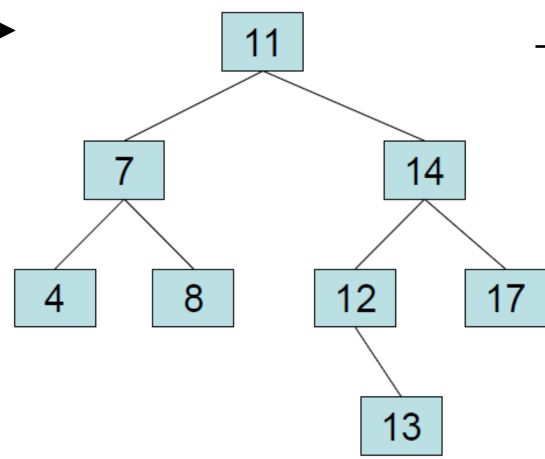




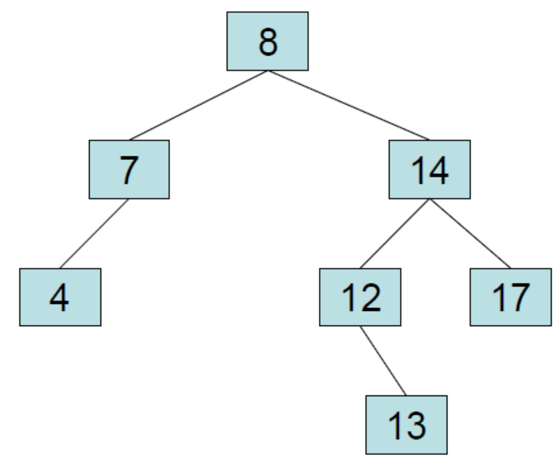
AVL tree is balanced



remove 53

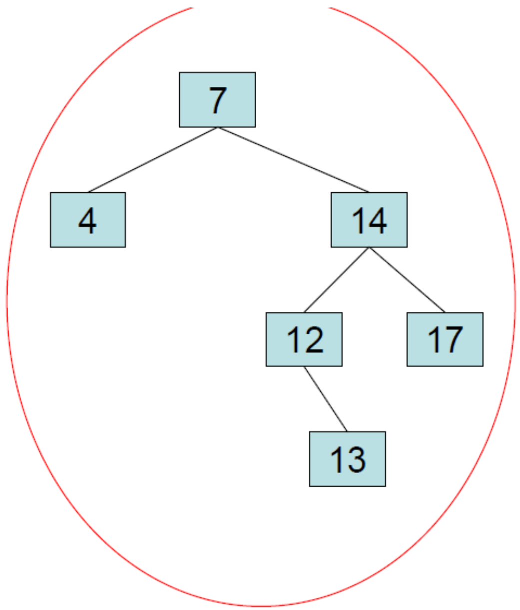


AVL tree is balanced

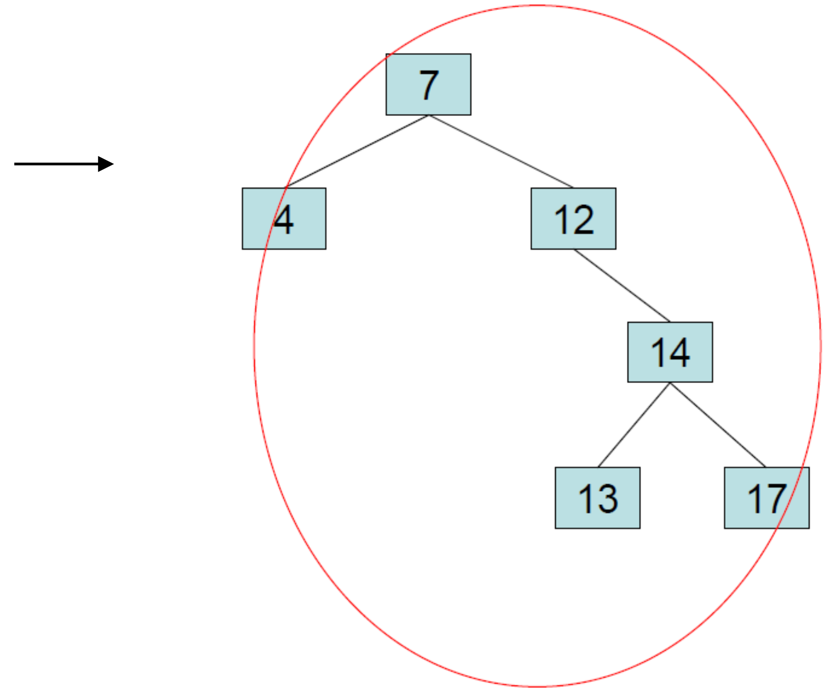


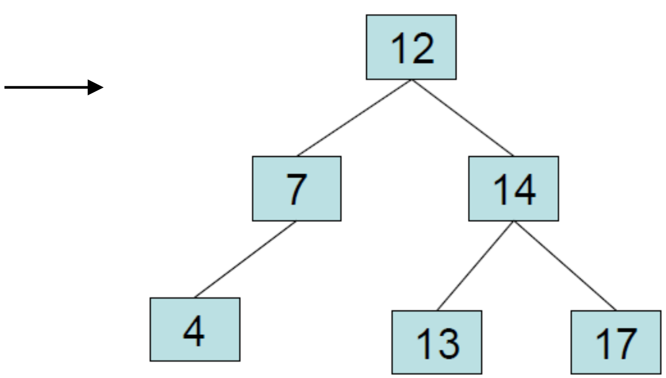
Remove 11

replace it with largest in its left branch



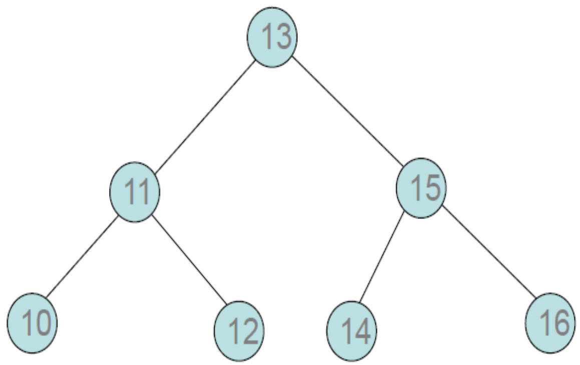
Remove 8, unbalanced



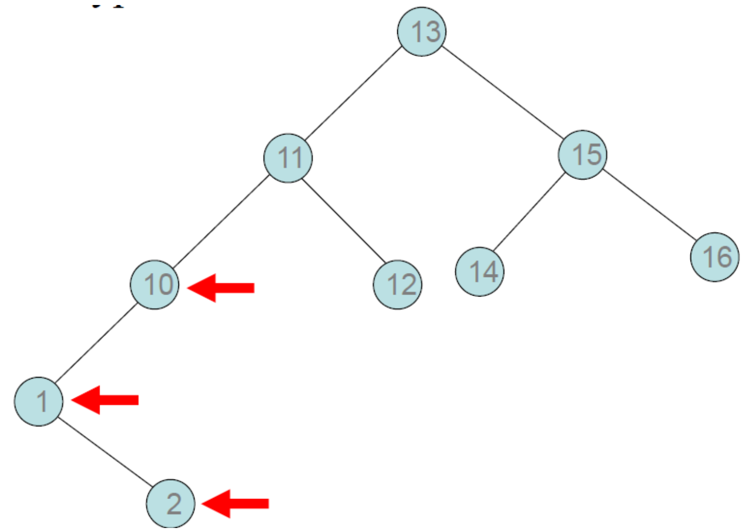


AVL Tree Double Rotations

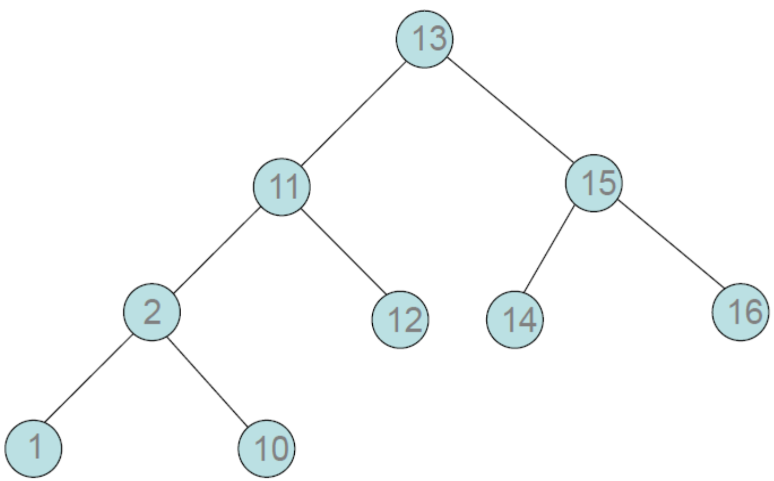
Insert 1, 2, 3, 4, 5, 7, 6, 9, 8



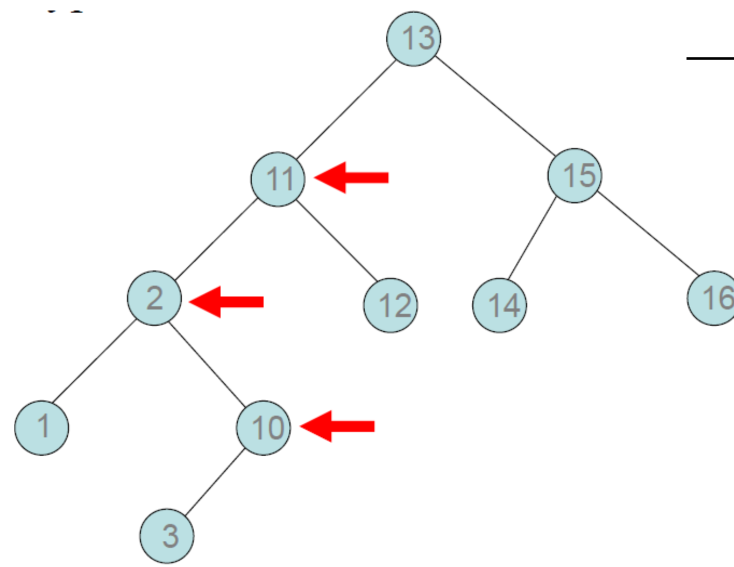
Original Tree

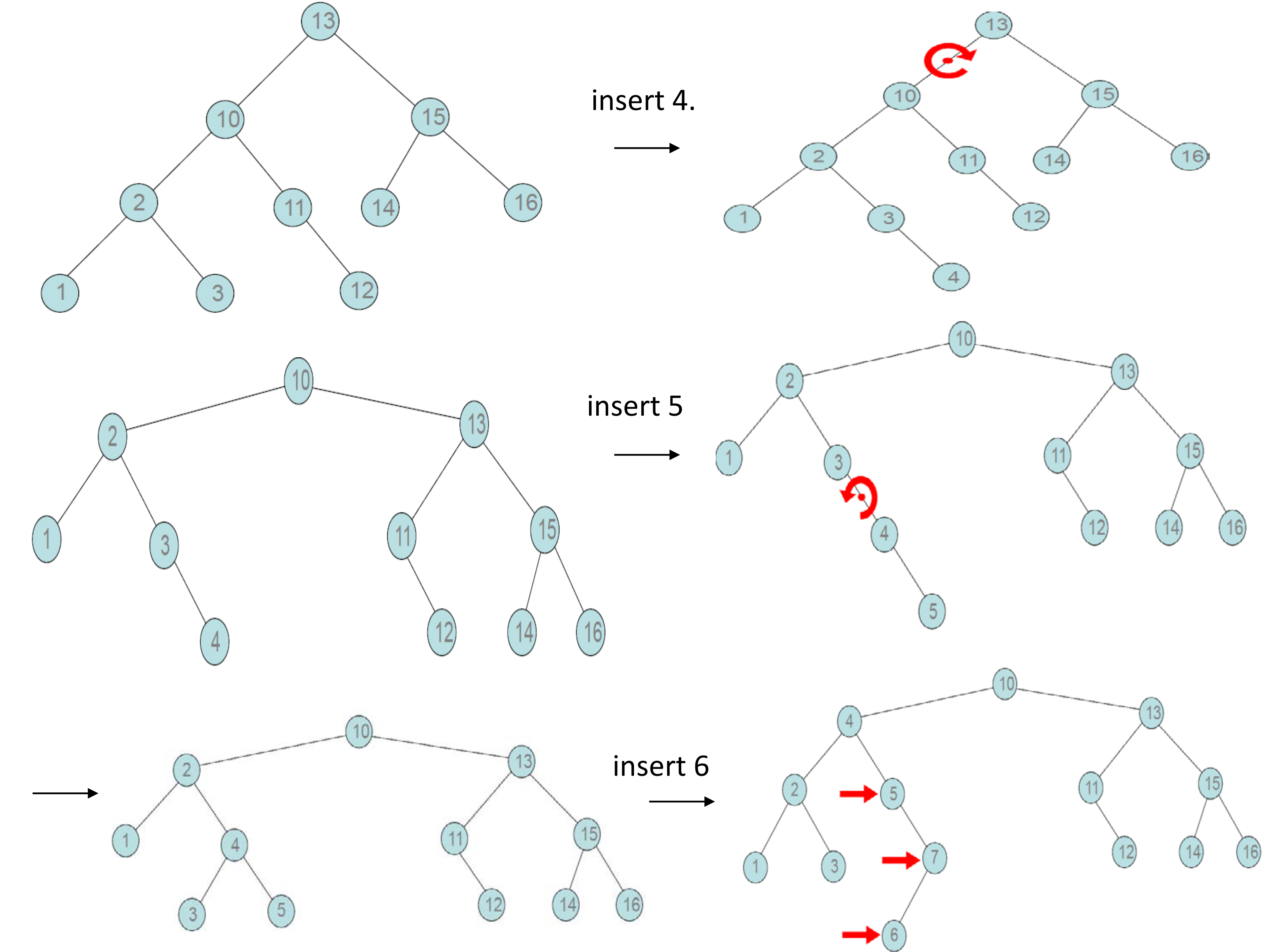


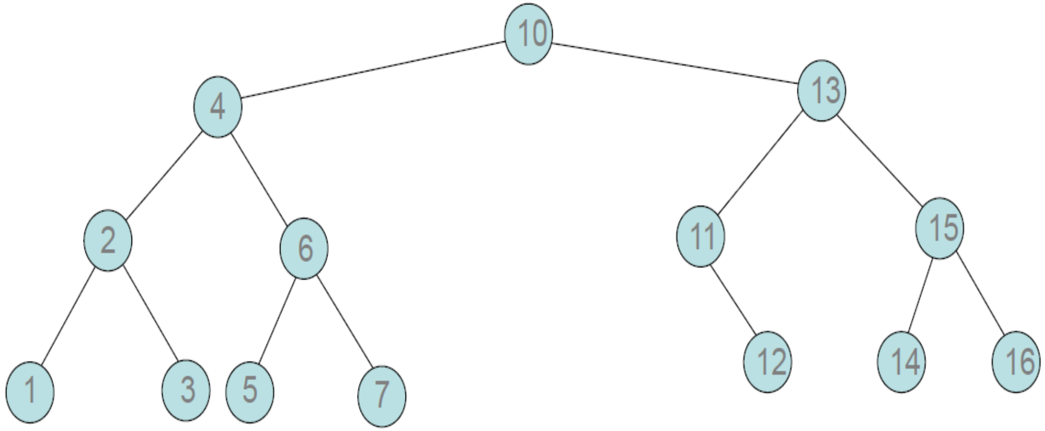
insert 1 and 2:



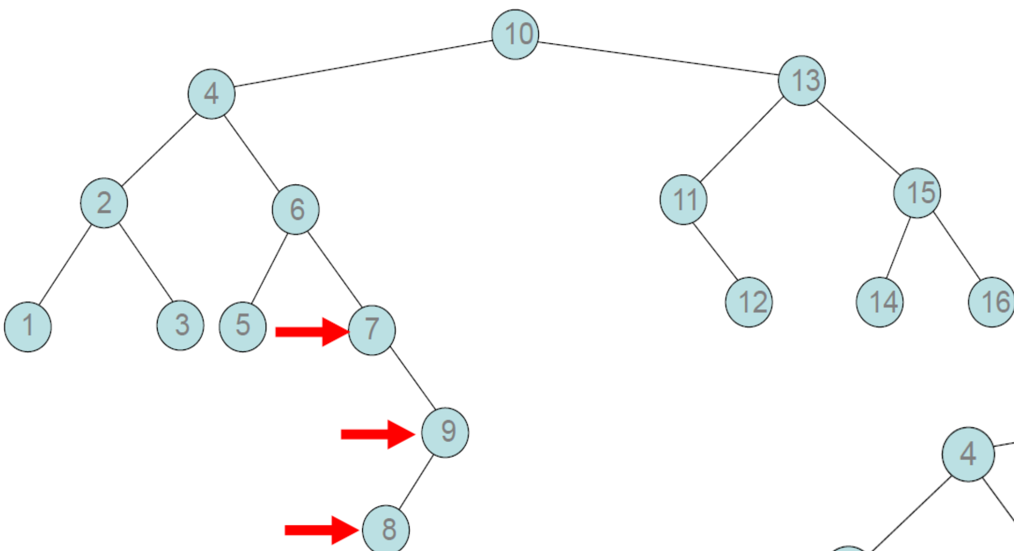
insert 3.

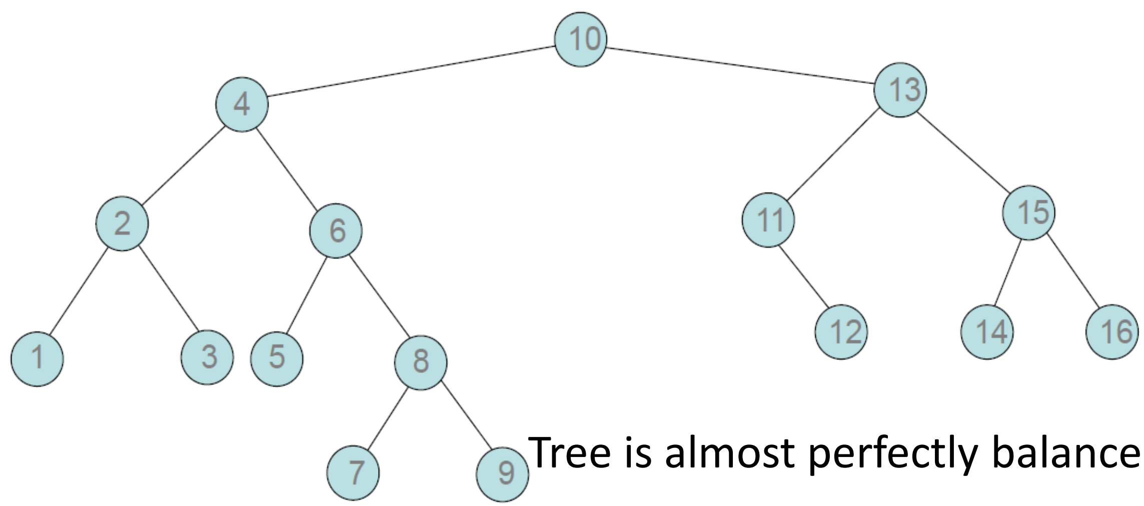



insert 4.
insert 5
insert 6



insert 9 and 8.





Tree is almost perfectly balanced