Algorithms and Data Structures

Andrzej Pisarski

Plan of the lecture

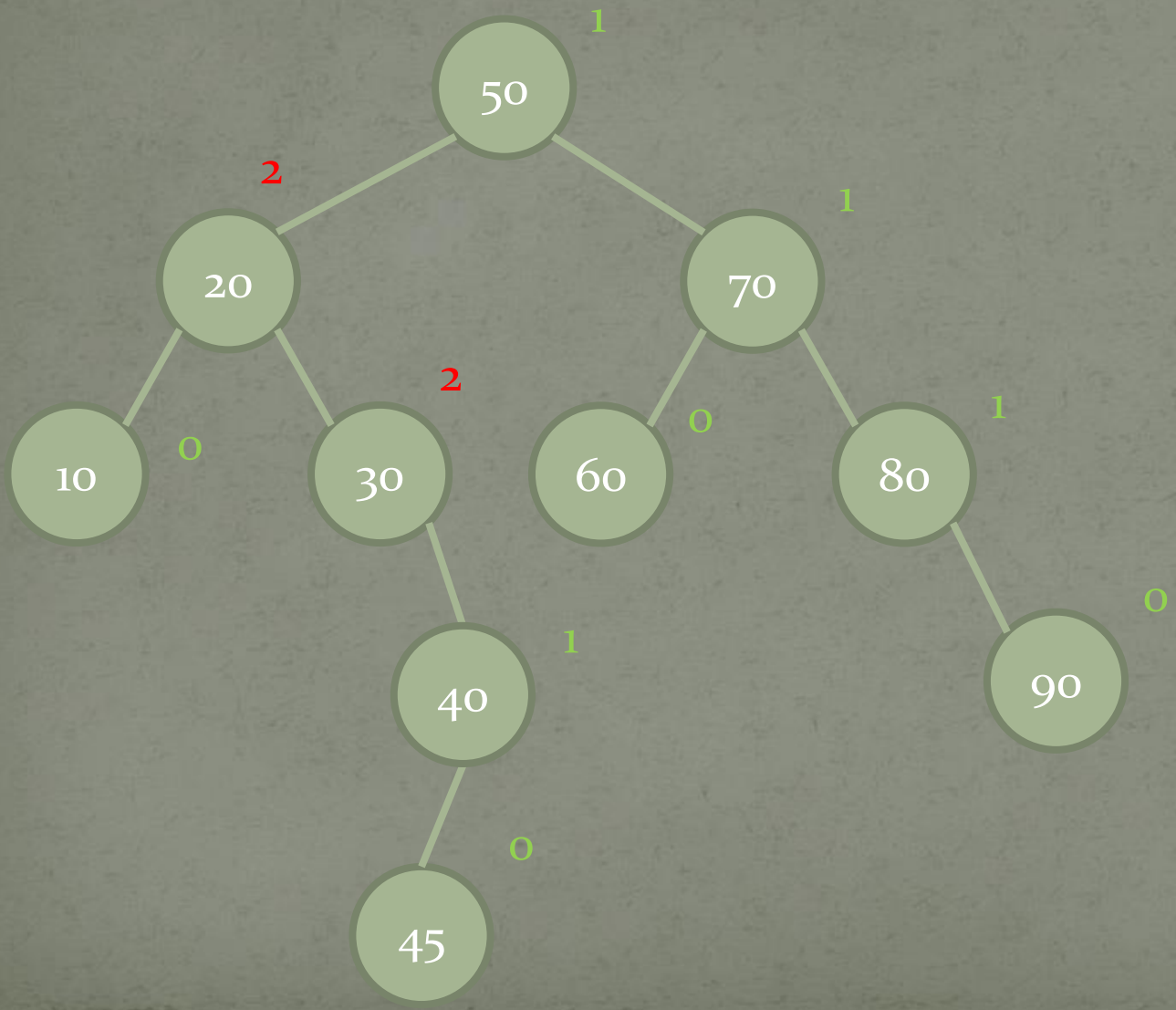
- AVL Tree [Georgy Adelson Velsky + Jewgienij Landis (1962)]
  - Rotation: Left, Right.
  - Insertion (1 or 2 operations of rotation)
- Red-Black Tree (RBT)
  - Insertion

AVL Tree

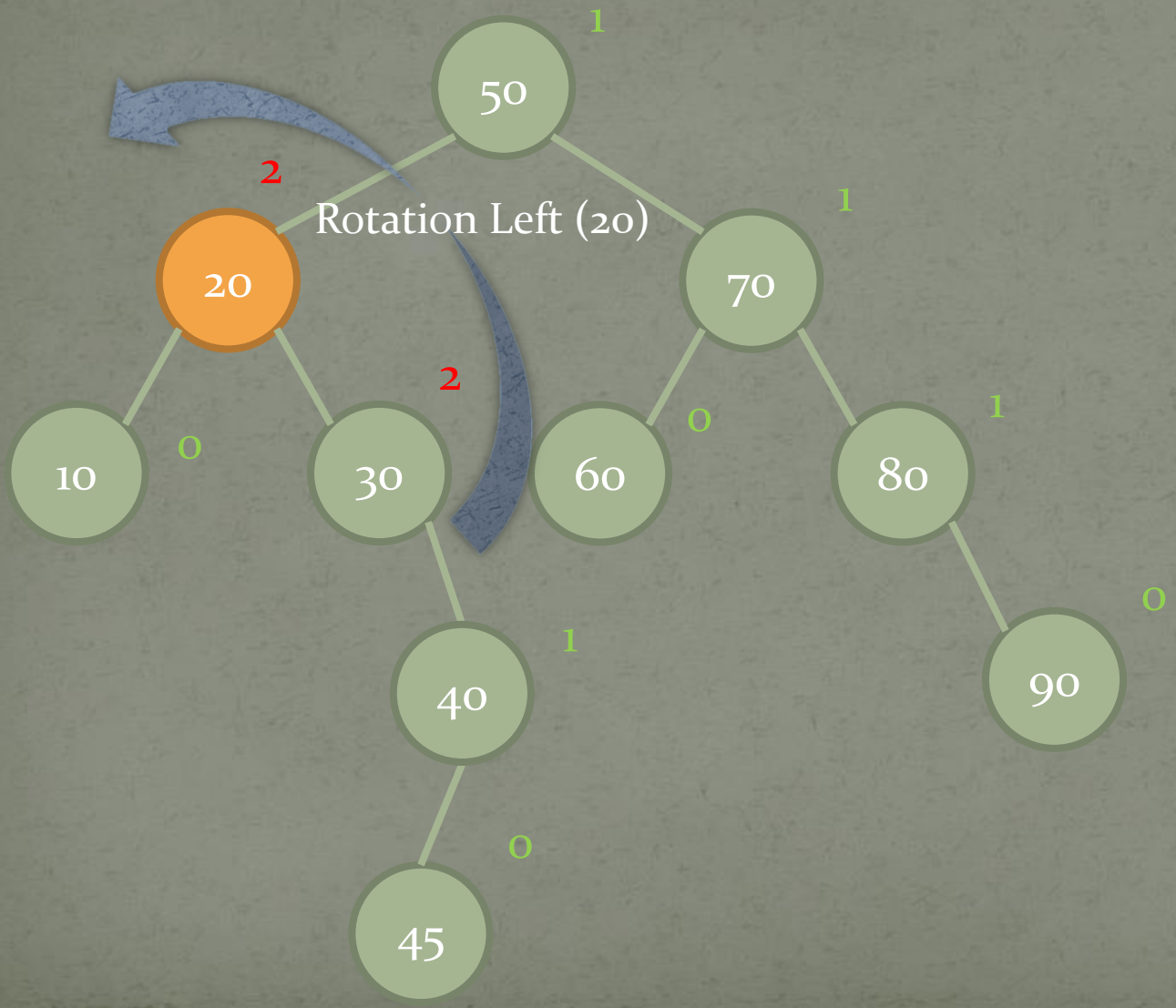
```
class Node{
    public:
        int iData;              /// data item (key)
        int weight;             /// it take -1, 0, 1 for well-balanced tree
        Node* pLeftChild;
        Node* pRightChild;
        Node* pParent;

        //constructor
        Node(int id=0, int iw=0, Node* pL=NULL, Node* pR=NULL, Node* pP=NULL)
        : iData(id), weight(iw), pLeftChild(pL), pRightChild(pR), pParent(pP)
        {  }
};
```

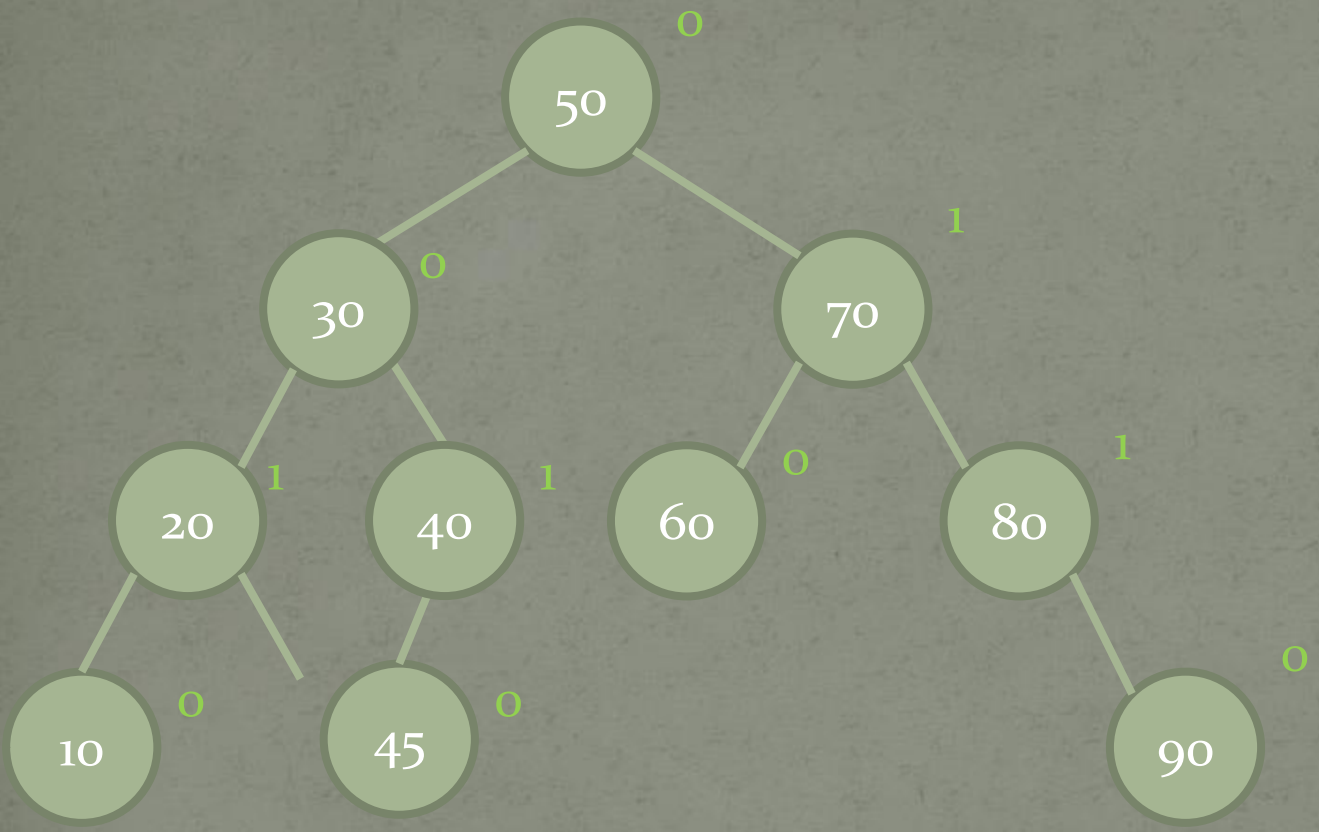
AVL Tree



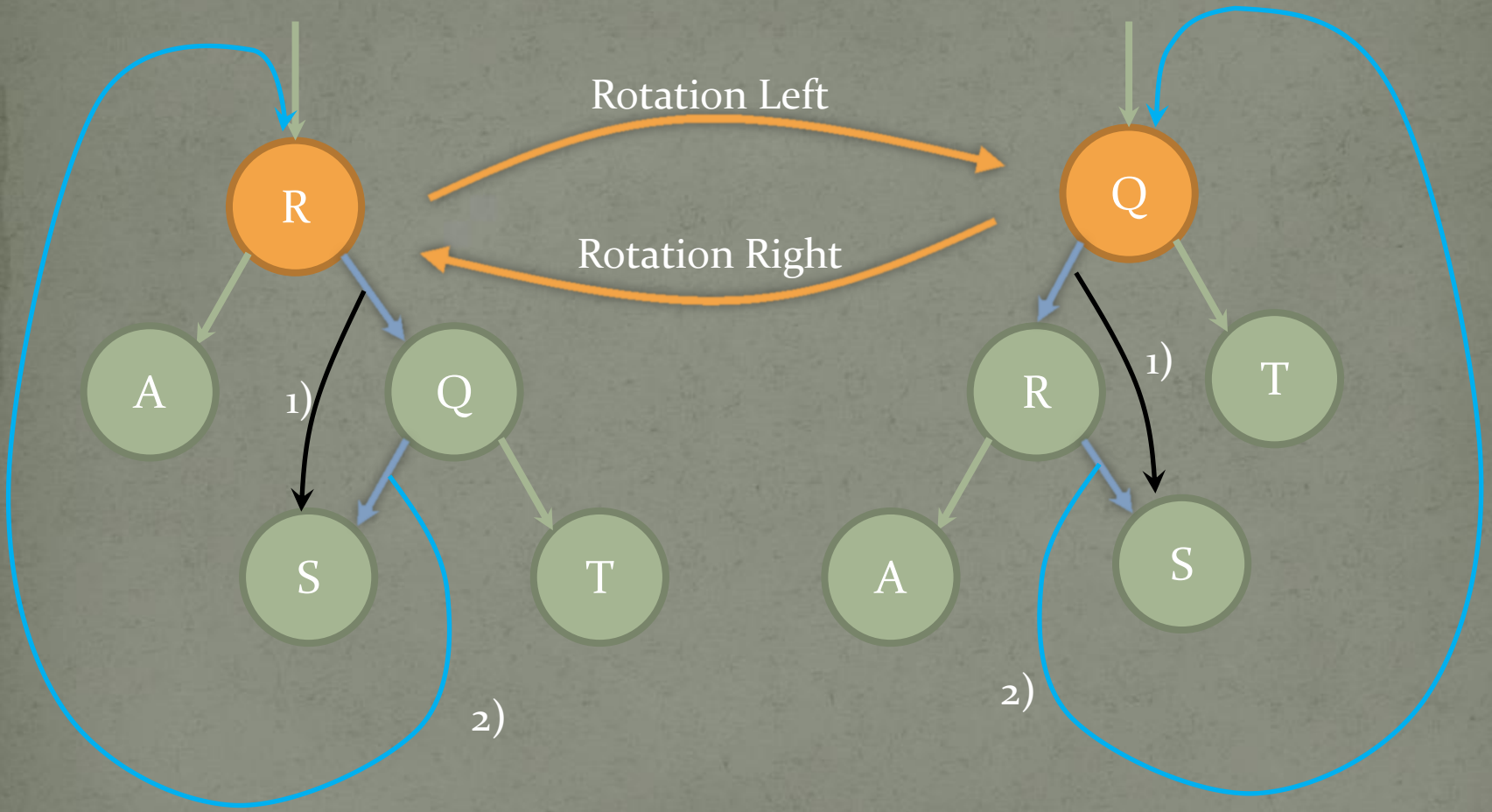
AVL Tree



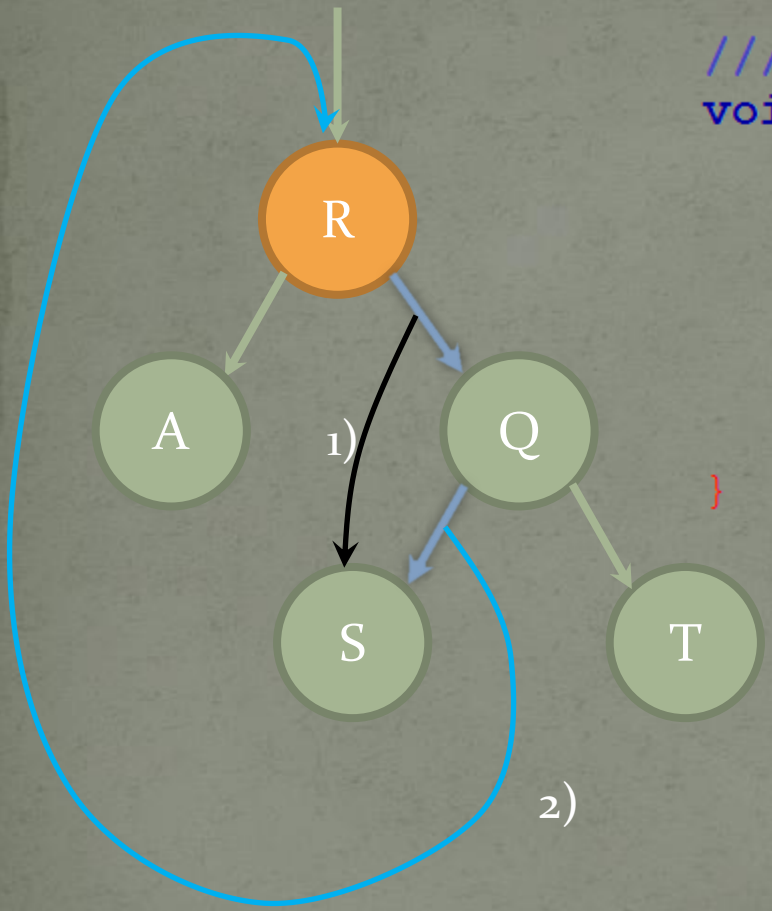
AVL Tree



AVL Tree: Rotation



AVL Tree: Rotation



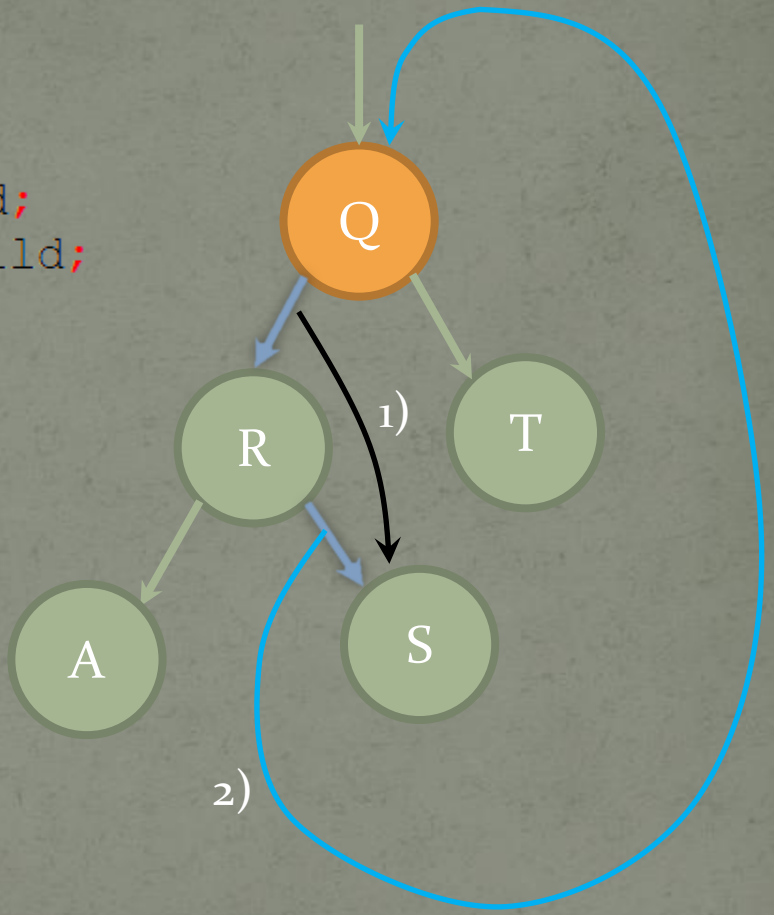

```
///Rotation left
void RLeft(Node* pN){
    Node* pR = pN -> pRightChild;
    Node* pRL = pR -> pLeftChild;
    pN -> pRighChild = pRL;

    pR -> pLeftChild = pN;
    pN = pR;
}
```

AVL Tree: Rotation

```
///Rotation right
void RRight(Node* pN){
    Node* pL = pN -> pLeftChild;
    Node* pLR = pL -> pRightChild;
    pN -> pLeftChild = pLR;

    pL -> pRightChild = pN;
    pN = pL;
}
```



AVL Tree: Insertion

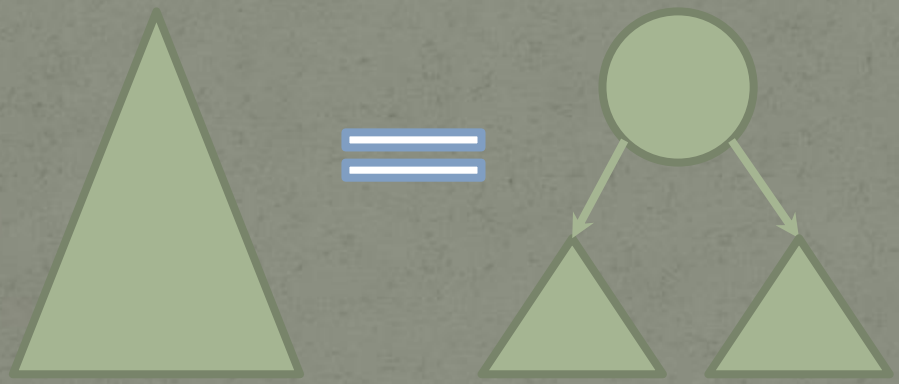
1) Insertion is performed like in Binary Search Tree (BST).
2) Going back to the root (in the opposite direction: from the inserted node to the root) every node must be checked and updated if necessary:

weight = 0 -> left and right subtree has this same height (balanced)
weight = -1 or 1 -> is balanced,
weight = -2 or 2 -> is not balanced ; there is a need to perform oparations:

- one rotation in case of:
  - insertion of new node to right subtree of the right node,
  - insertion of new node to left subtree of the left node,
- two rotation in case of:
  - insertion of new node to left subtree of the right node,
  - insertion of new node to right subtree of the left node.

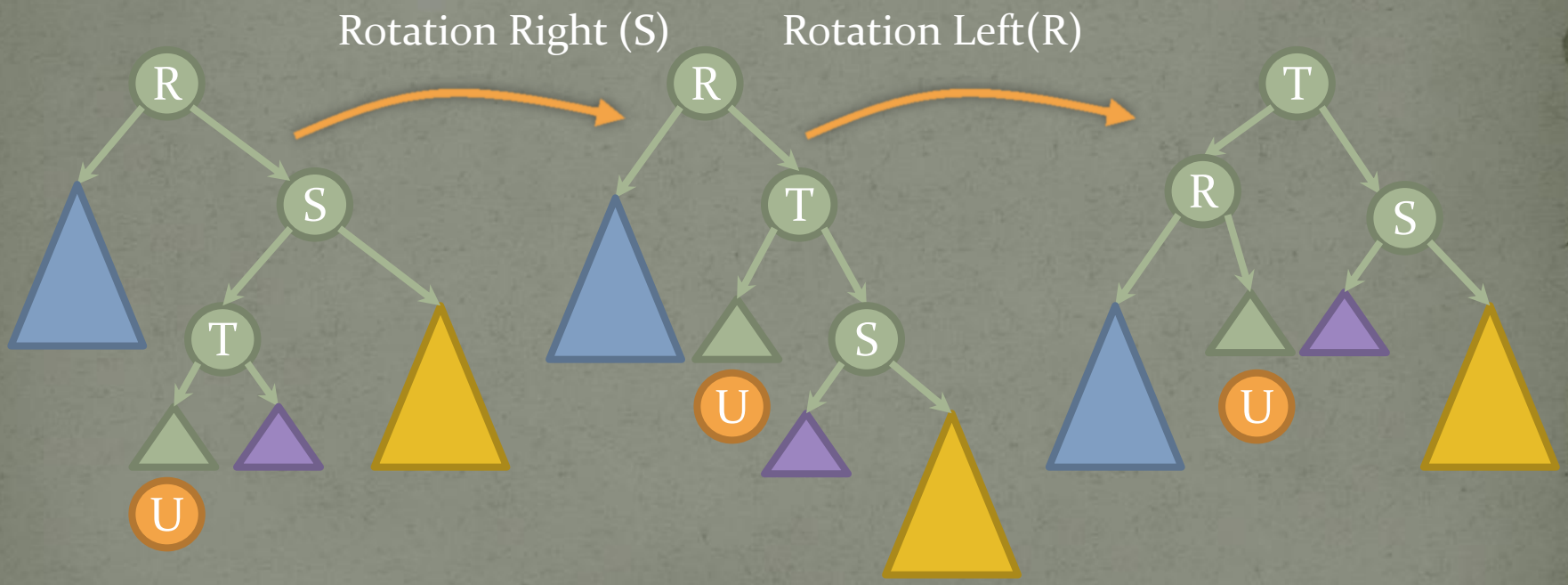
AVL Tree: Insertion

- two rotation in case of:
  - insertion of new node to left subtree of the right node



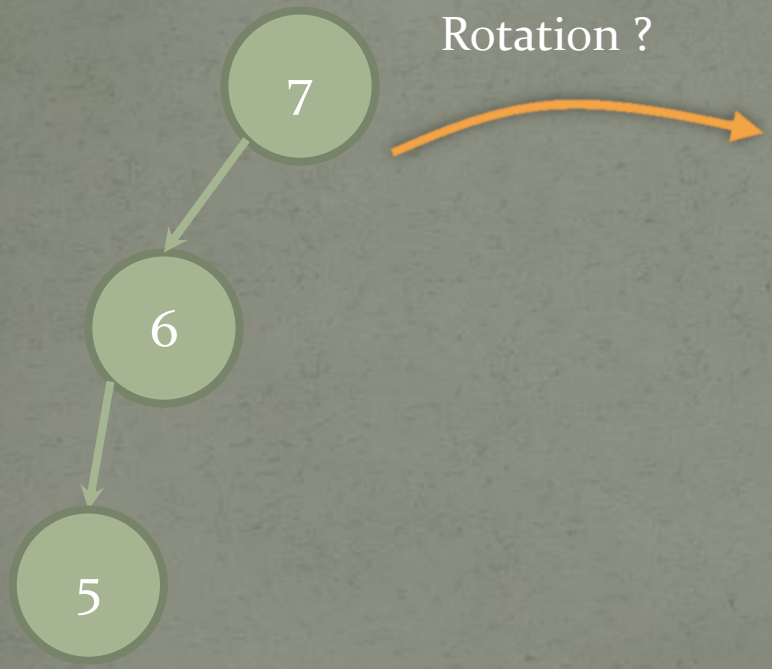
AVL Tree: Insertion

- two rotation in case of:
  - insertion of new node to left subtree of the right node



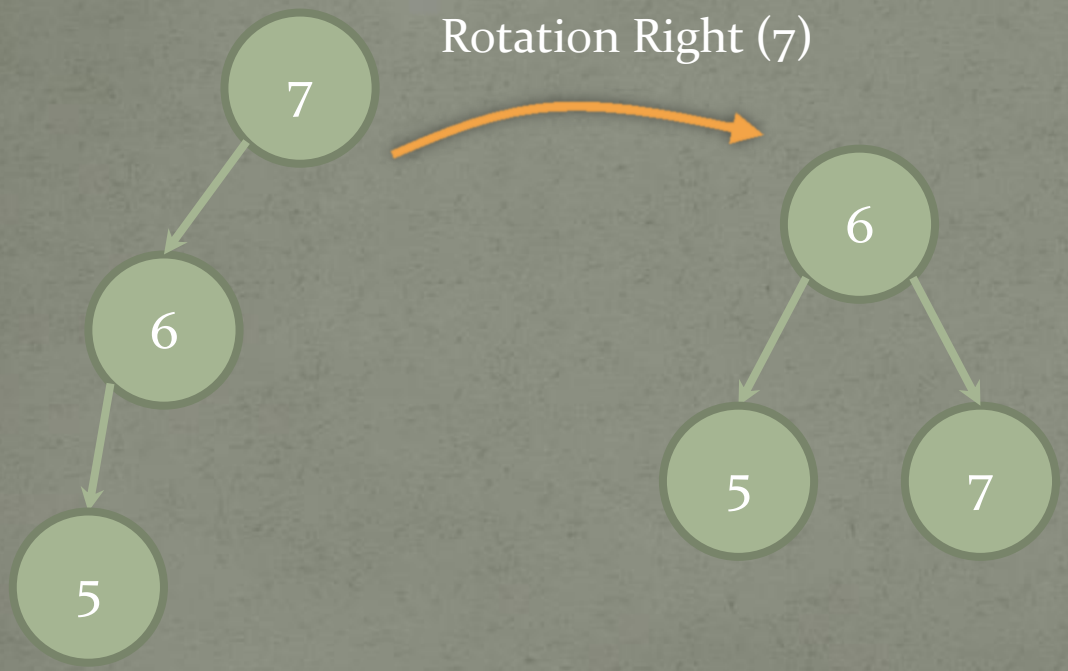
AVL Tree: Insertion

- Example 1:
  - insertion: 7, 6, 5



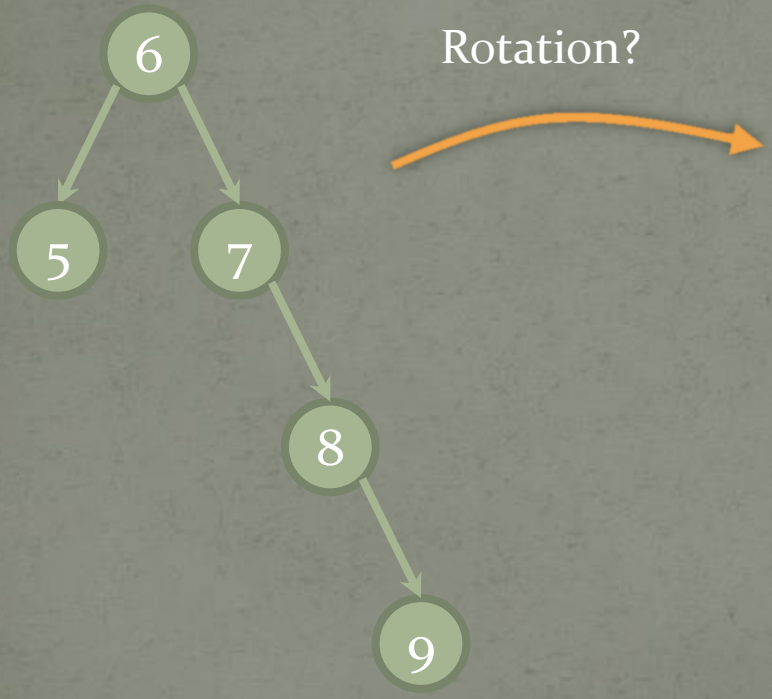
AVL Tree: Insertion

- Example 1:
  - insertion: 7, 6, 5



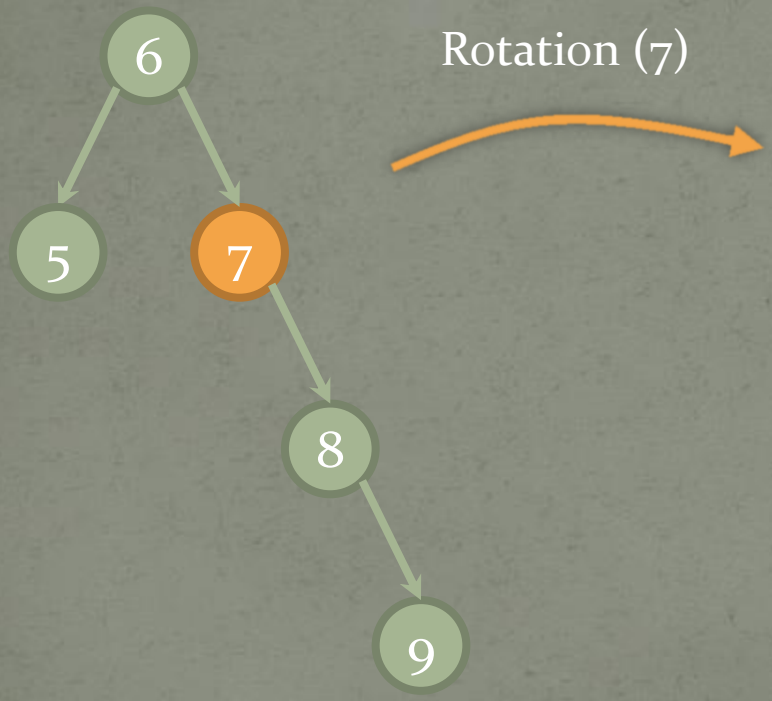
AVL Tree: Insertion

- Example 2:
  - insertion: 8, 9



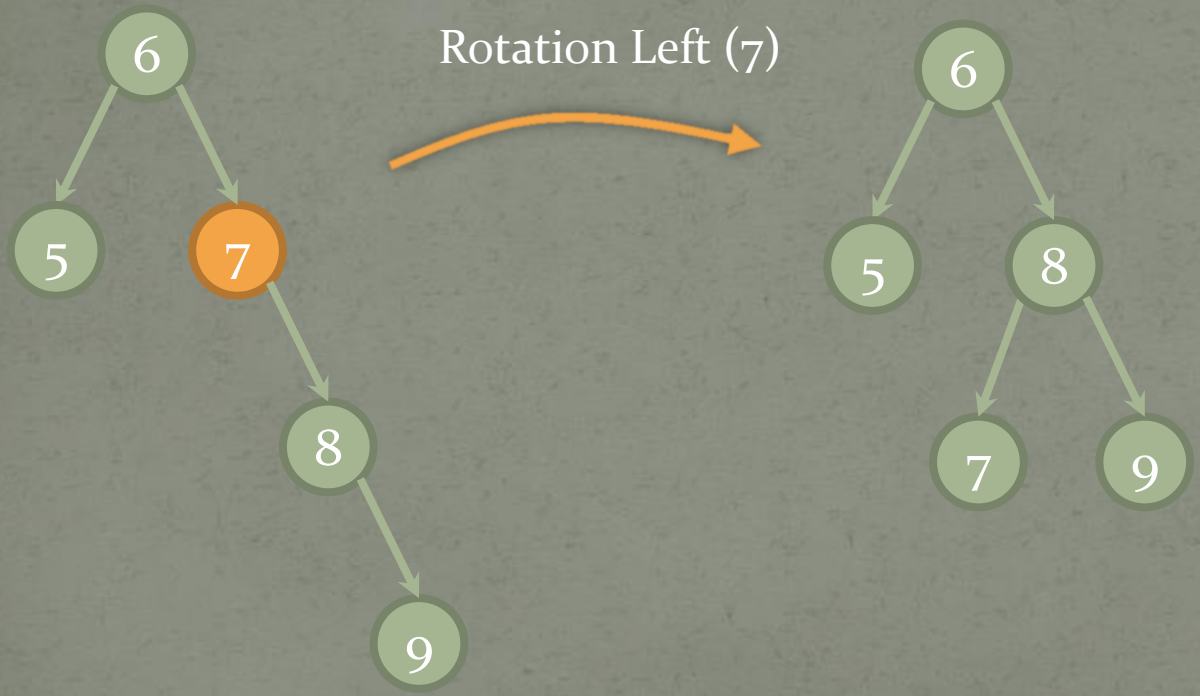
AVL Tree: Insertion

- Example 2:
  - insertion: 8, 9



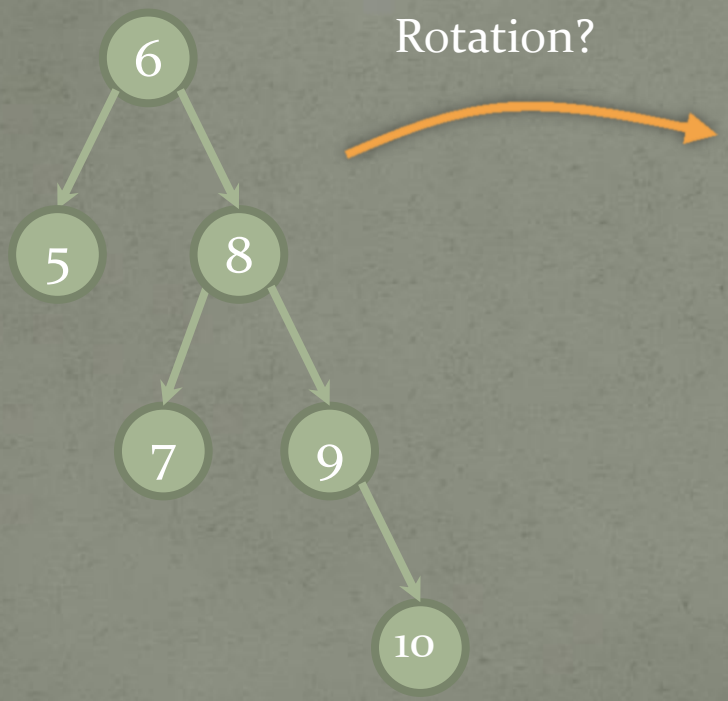
AVL Tree: Insertion

- Example 2:
  - insertion: 8, 9



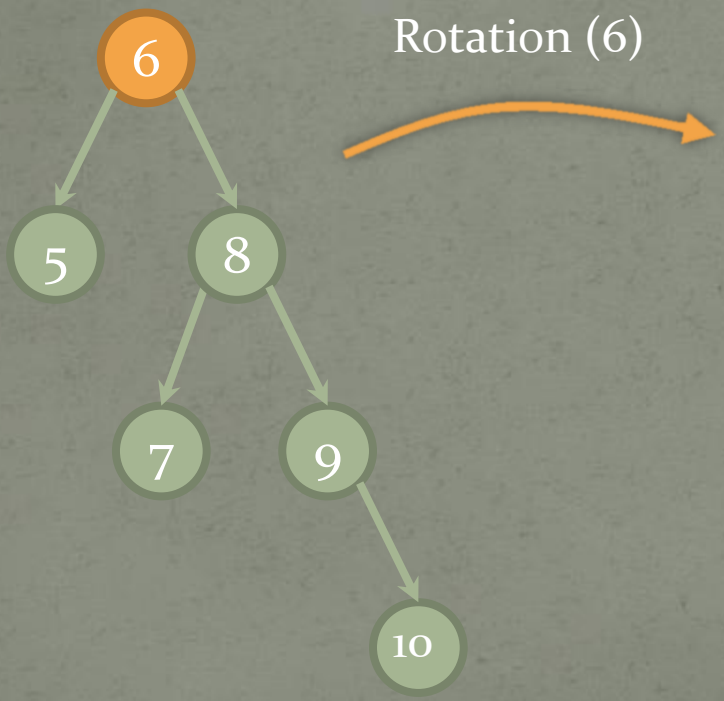
AVL Tree: Insertion

- Example 3:
  - insertion: 10



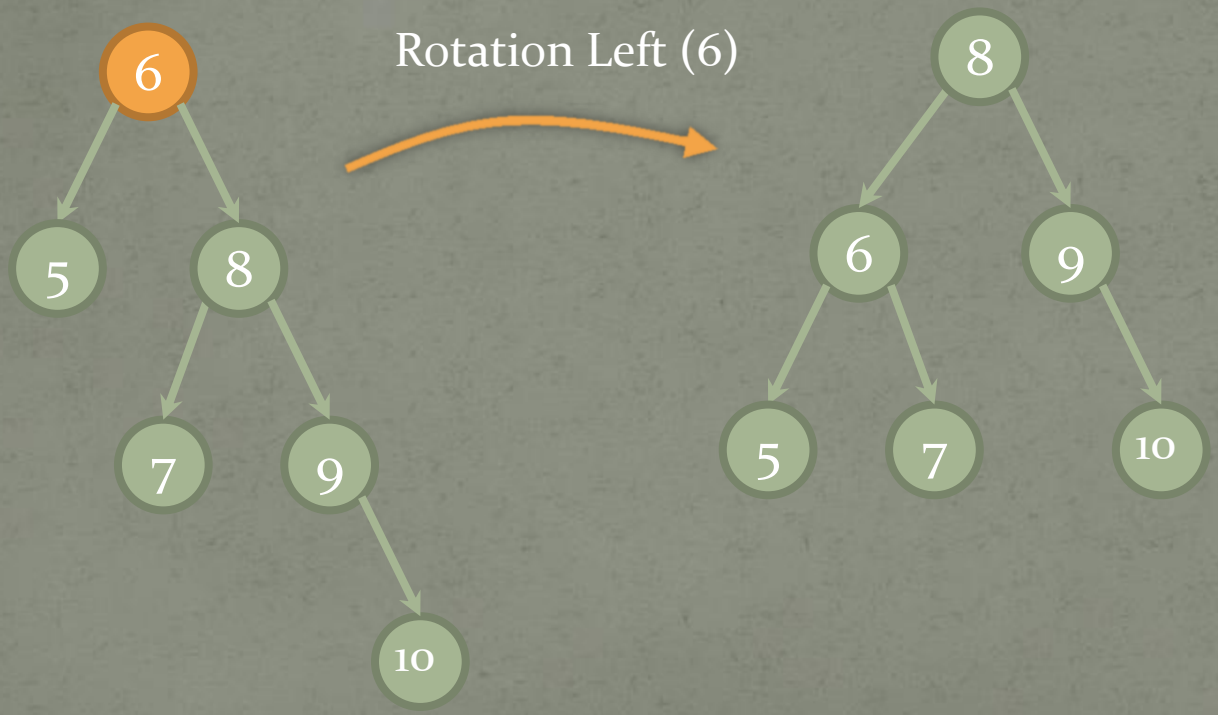
AVL Tree: Insertion

- Example 3:
  - insertion: 10



AVL Tree: Insertion

- Example 3:
  - insertion: 10



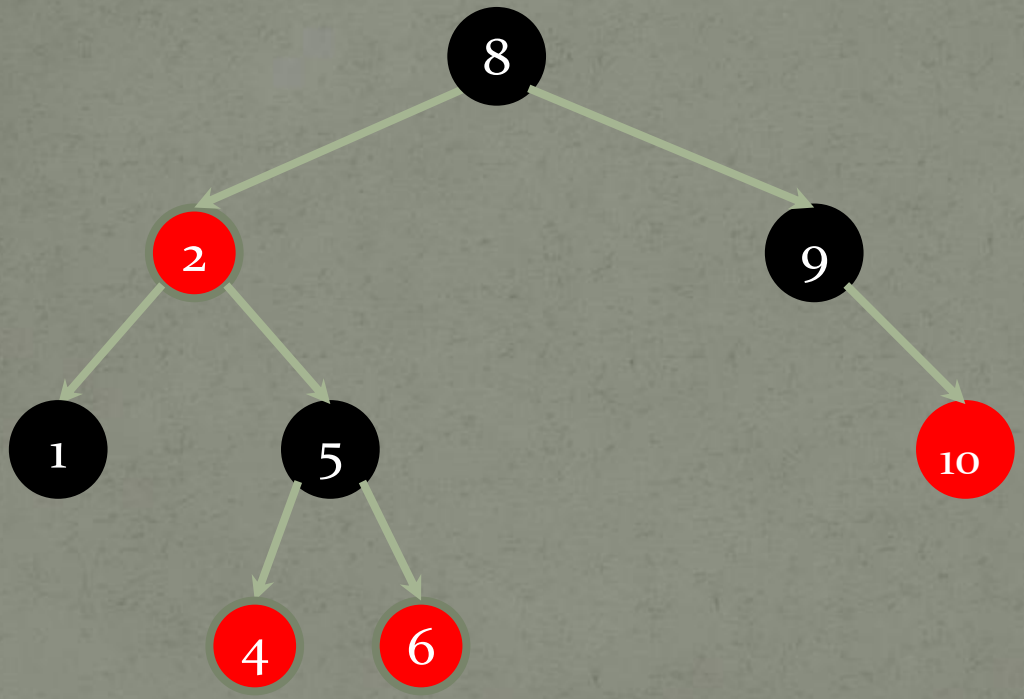
Red-Black Tree

Node insertion or delation:

1. Each node is black or red,
2. The root is always black ,
3. Each leaf is always black ,
4. Children of red node must be black,
5. Each path from the root to leaf must contain the same number of black nodes

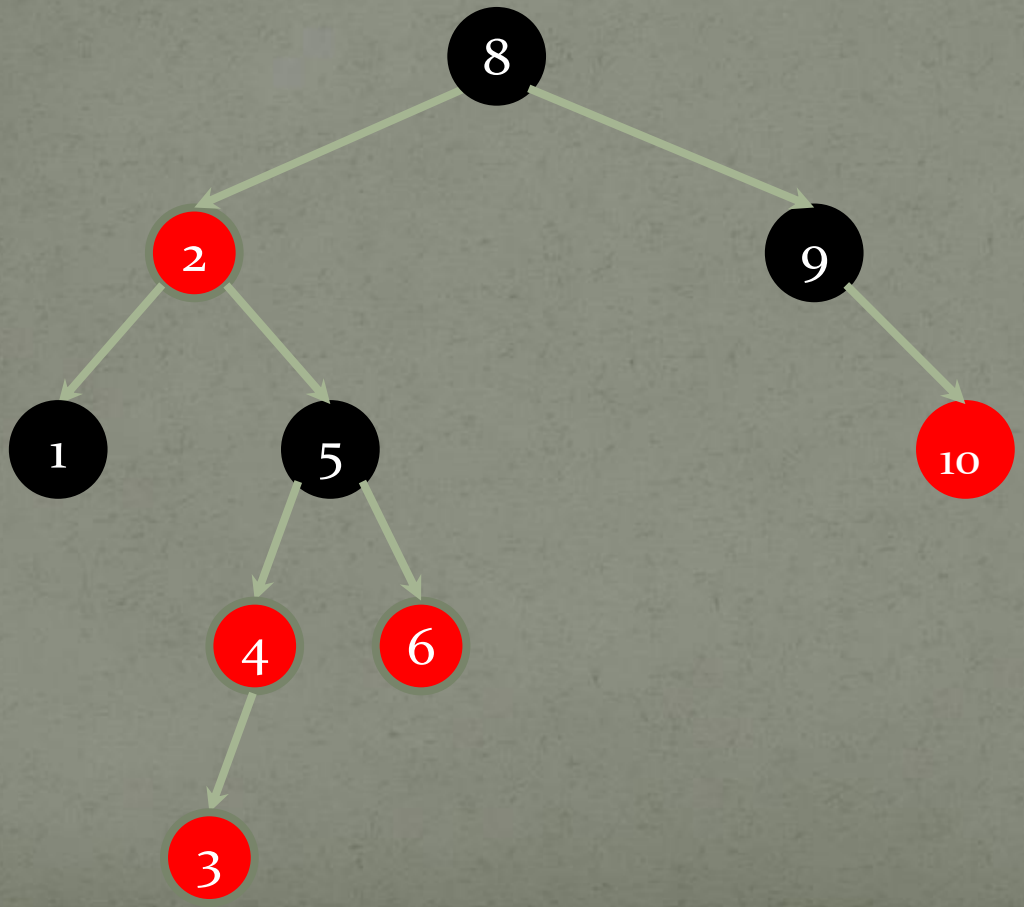
Red-Black Tree: insertion

Case 1: node insertion with key = 3



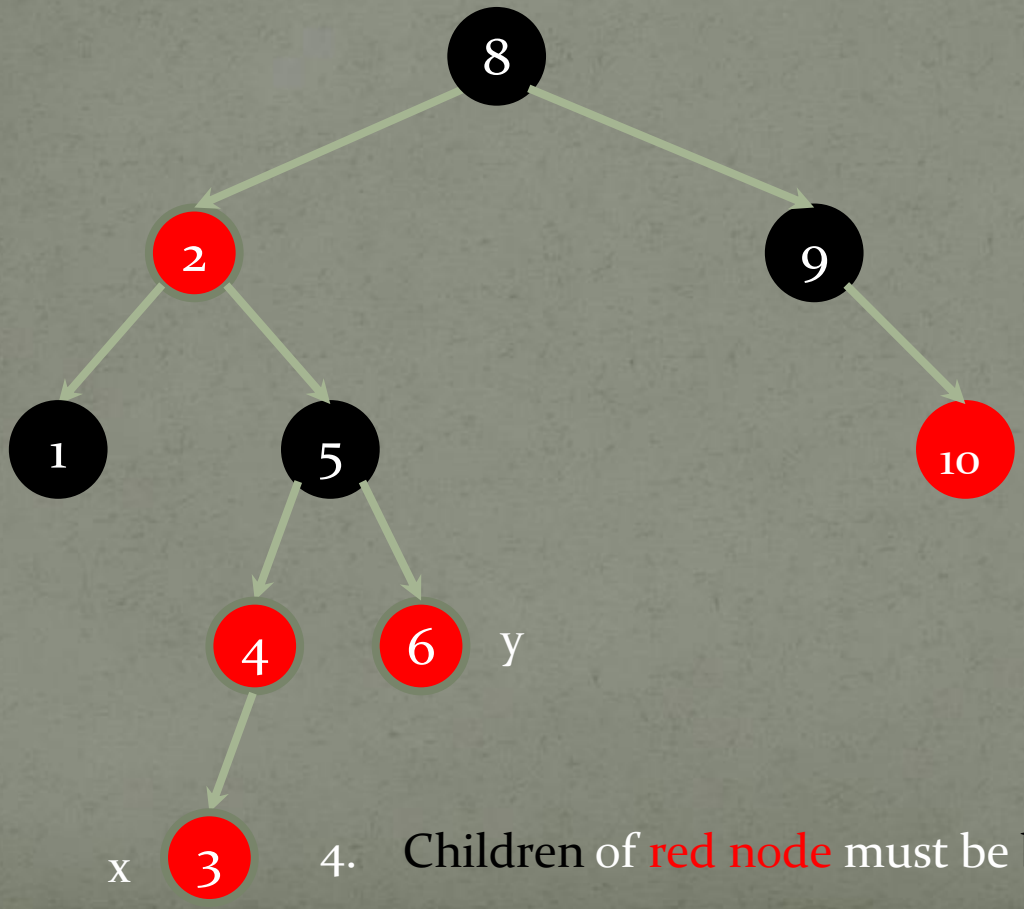
Red-Black Tree: insertion

Case 1: node insertion with key = 3



Red-Black Tree: insertion

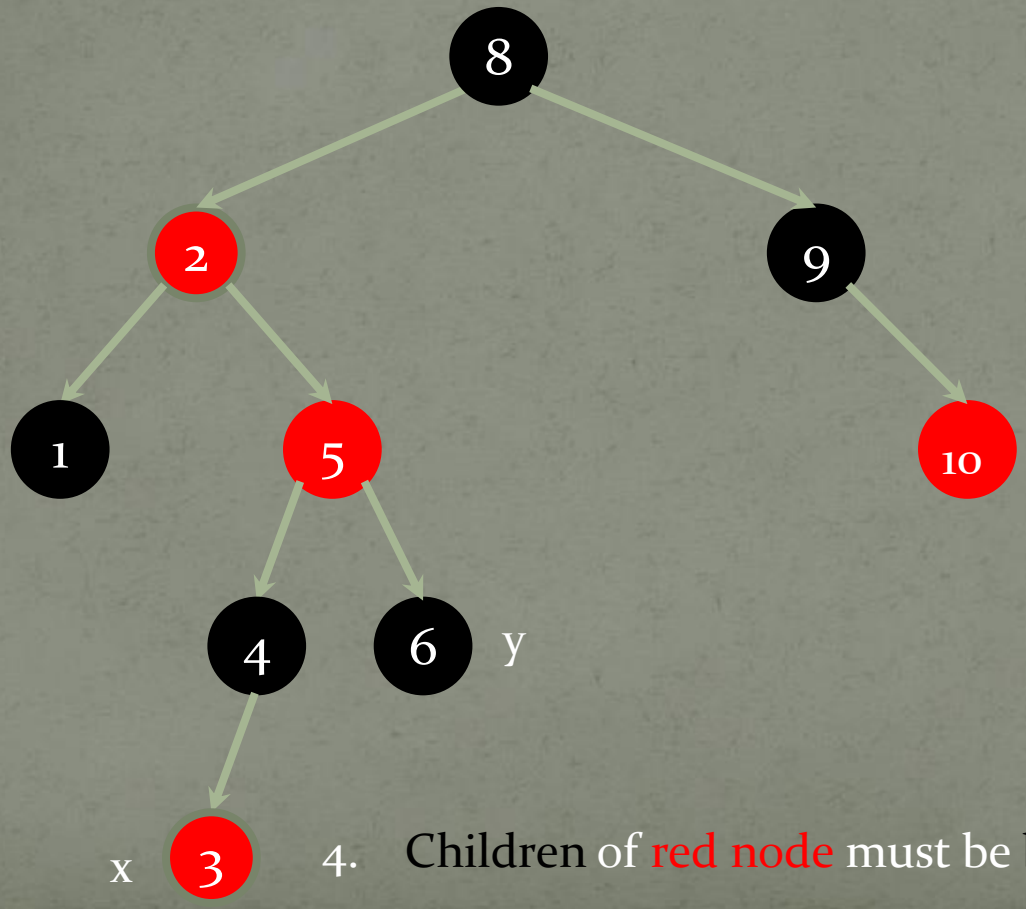
Case 1: node insertion with key = 3



4. Children of red node must be black

Red-Black Tree: insertion

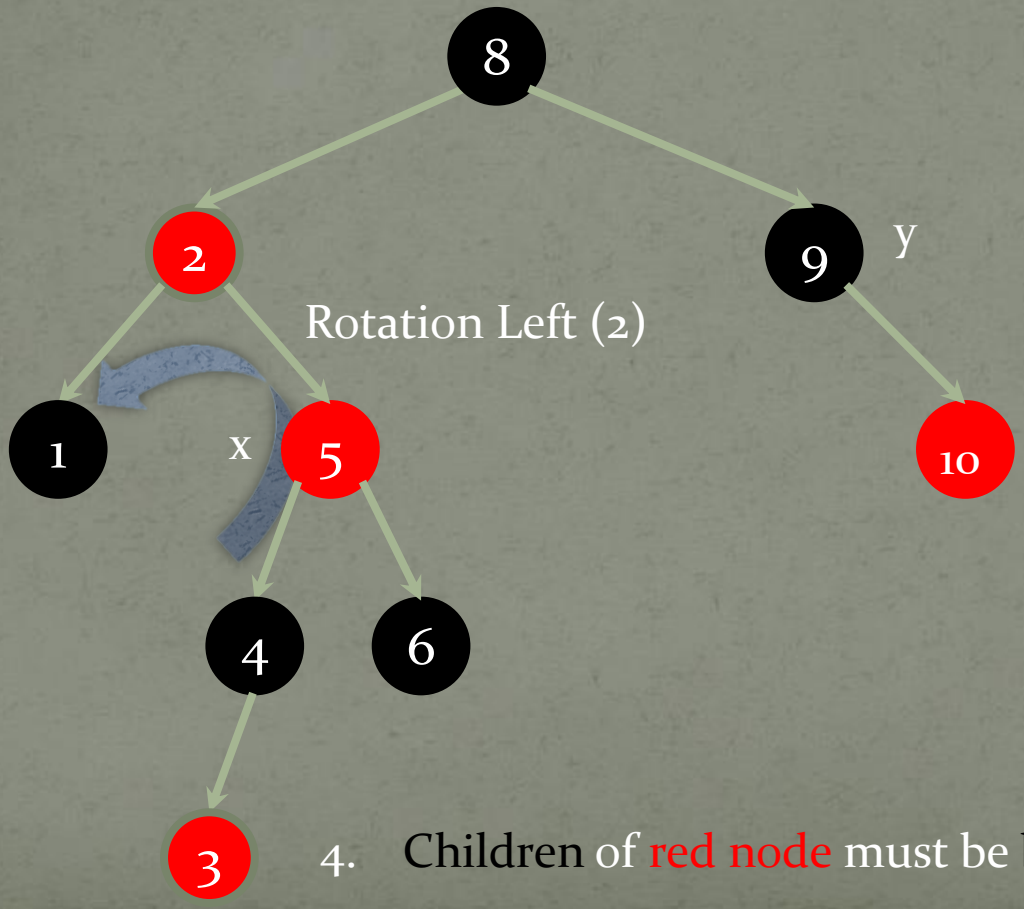
Case 2: node insertion with key = 3



4. Children of red node must be black

Red-Black Tree: insertion

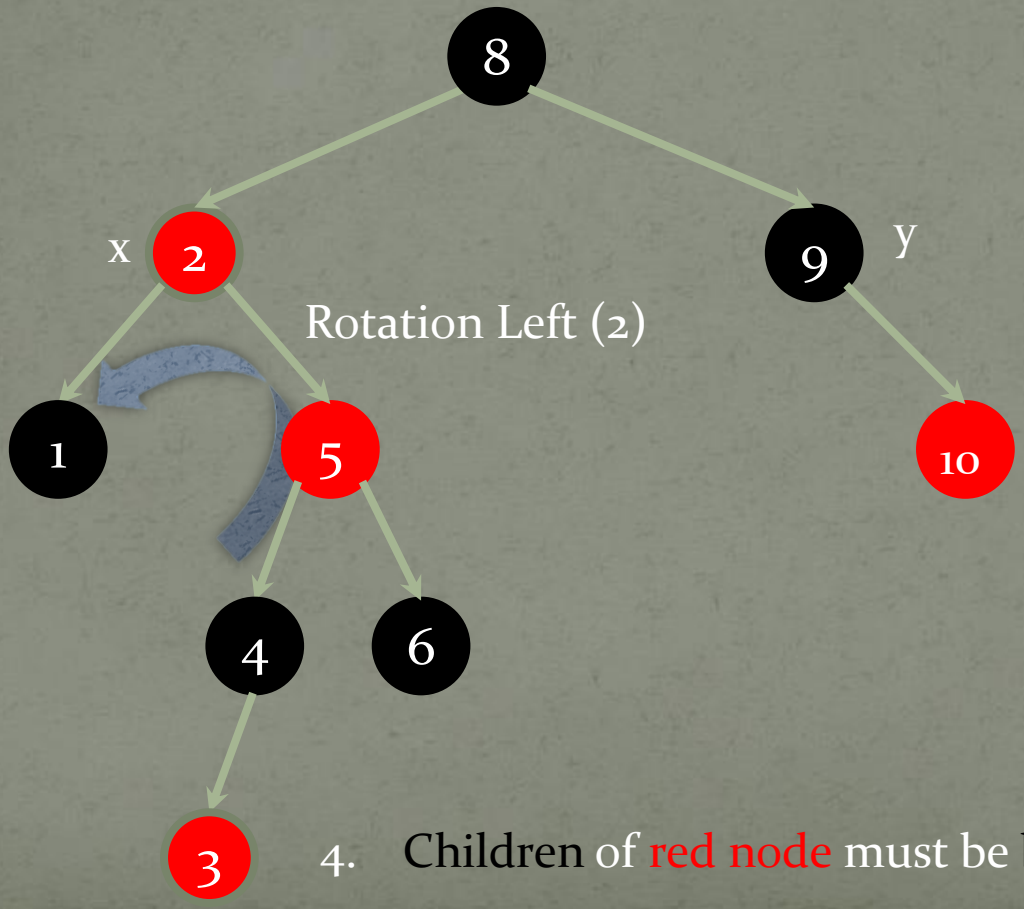
Case 2: node insertion with key = 3



4. Children of red node must be black

Red-Black Tree: insertion

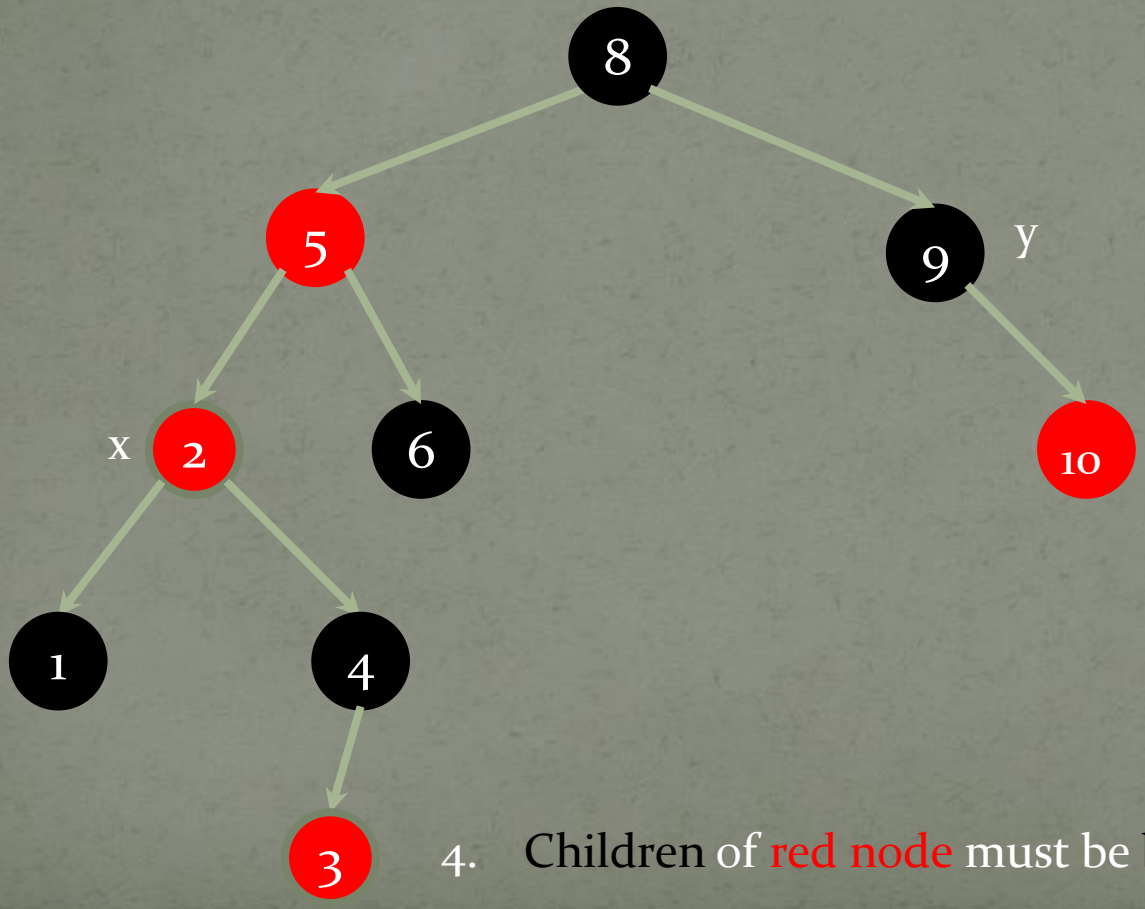
Case 2: node insertion with key = 3



4. Children of red node must be black

Red-Black Tree: insertion

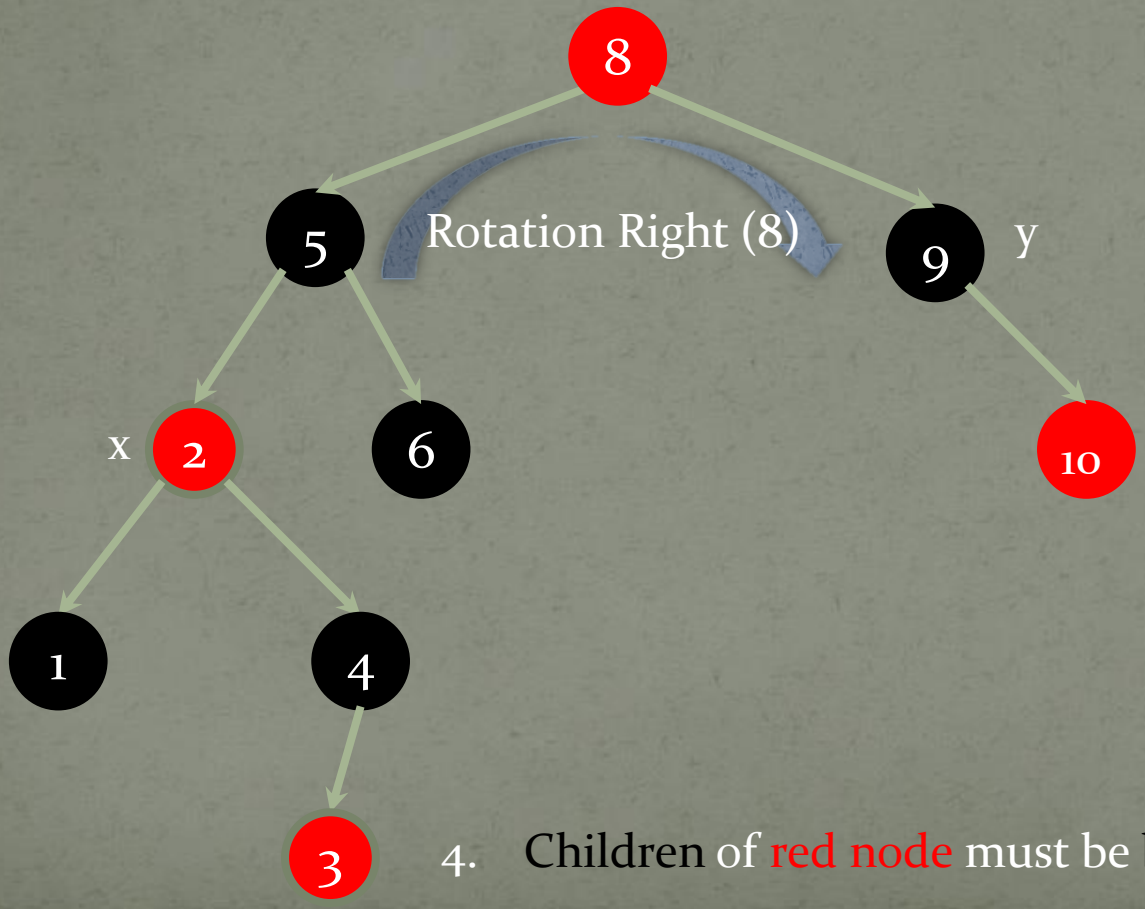
Case 2: node insertion with key = 3



4. Children of red node must be black

Red-Black Tree: insertion

Case 3: node insertion with key = 3



4. Children of red node must be black

Red-Black Tree: insertion

Case 3: node insertion with key = 3