# 数据结构与算法（Python版）

## AVL树的Python实现

陈斌 北京大学 gischen@pku.edu.cn

# AVL树的Python实现



- 既然AVL平衡树确实能够改进BST树的性能，避免退化情形
- 我们来看看向AVL树插入一个新key，如何才能保持AVL树的平衡性质
- 首先，作为BST，新key必定以**叶节点**形式插入到AVL树中

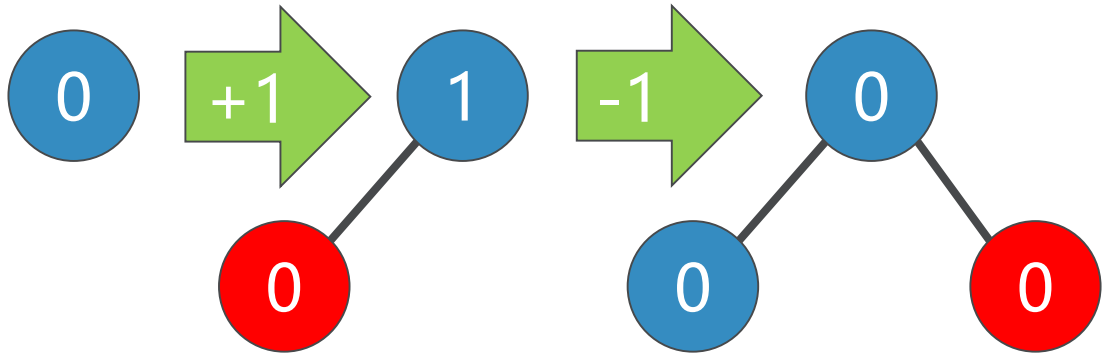
# AVL树的Python实现



- 叶节点的平衡因子是0，其本身无需重新平衡
- 但会影响其父节点的平衡因子：

  作为左子节点插入，则父节点平衡因子会增加1；

  作为右子节点插入，则父节点平衡因子会减少1。



# AVL树的Python实现

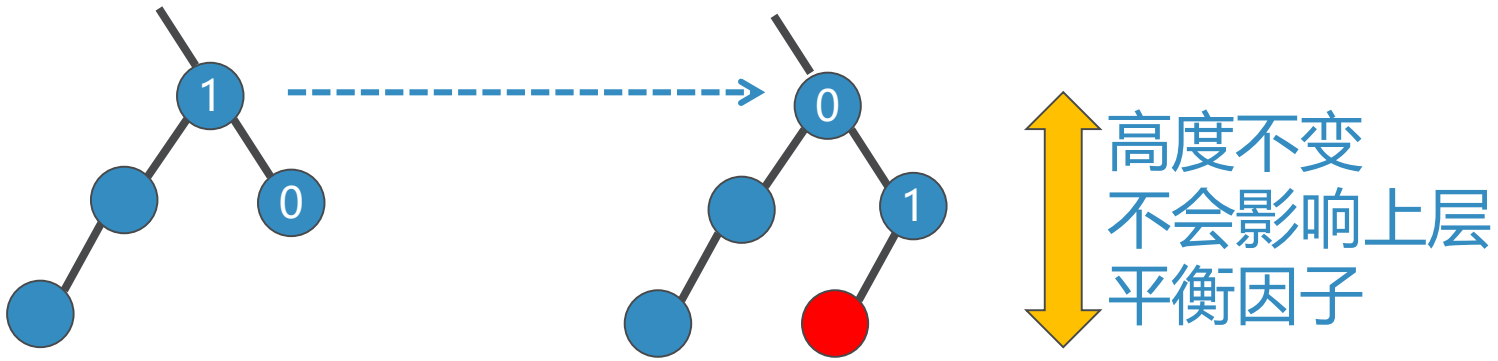


- **这种影响可能随着其父节点到根节点的路径一直传递上去，直到：**

  **传递到根节点为止；**

  **或者某个父节点平衡因子被调整到0，不再影响上层节点的平衡因子为止。**

  - （无论从-1或者1调整到0，都不会改变子树高度）



# AVL树的实现：put方法



- 重新定义_put方法即可

```
def _put(self,key,val,currentNode):
    if key < currentNode.key:
        if currentNode.hasLeftChild():
            self._put(key,val,currentNode.leftChild)
        else:
            currentNode.leftChild = TreeNode(key,val,parent=currentNode)
            self.updateBalance(currentNode.leftChild)
    else:
        if currentNode.hasRightChild():
            self._put(key,val,currentNode.rightChild)
        else:
            currentNode.rightChild = TreeNode(key,val,parent=currentNode)
            self.updateBalance(currentNode.rightChild)
```

调整因子

调整因子

# AVL树的实现：UpdateBalance方法



```
def updateBalance(self,node):
    if node.balanceFactor > 1 or node.balanceFactor < -1:
        self.rebalance(node)
        return
    if node.parent != None:
        if node.isLeftChild():
            node.parent.balanceFactor += 1
        elif node.isRightChild():
            node.parent.balanceFactor -= 1

        if node.parent.balanceFactor != 0:
            self.updateBalance(node.parent)
```

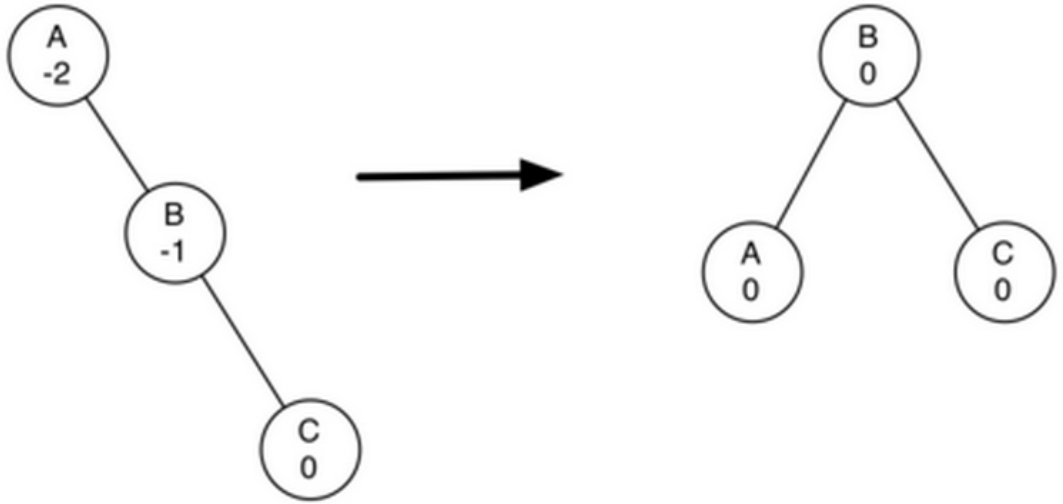
重新平衡

调整父节点因子

# AVL树的实现：rebalance重新平衡



- 主要手段：将不平衡的子树进行**旋转** rotation
  - 视“左重”或者“右重”进行不同方向的旋转
  - 同时更新相关父节点引用，更新旋转后被影响节点的平衡因子



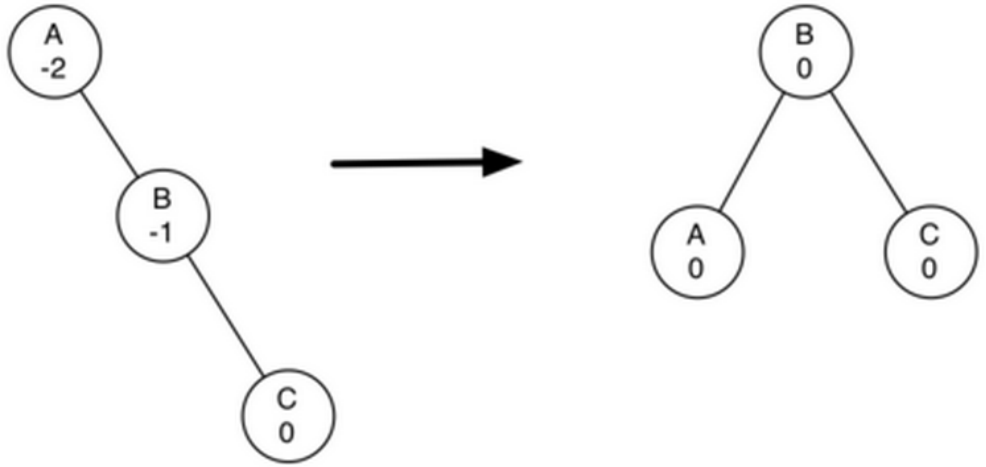
# AVL树的实现：rebalance重新平衡



- 如图，是一个“右重”子树A的左旋转（**并保持BST性质**）

  将右子节点B提升为子树的根，将旧根节点A作为新根节点B的左子节点

  如果新根节点B原来有左子节点，则将此节点设置为A的右子节点（A的右子节点一定有空）



# AVL树的实现：rebalance重新平衡

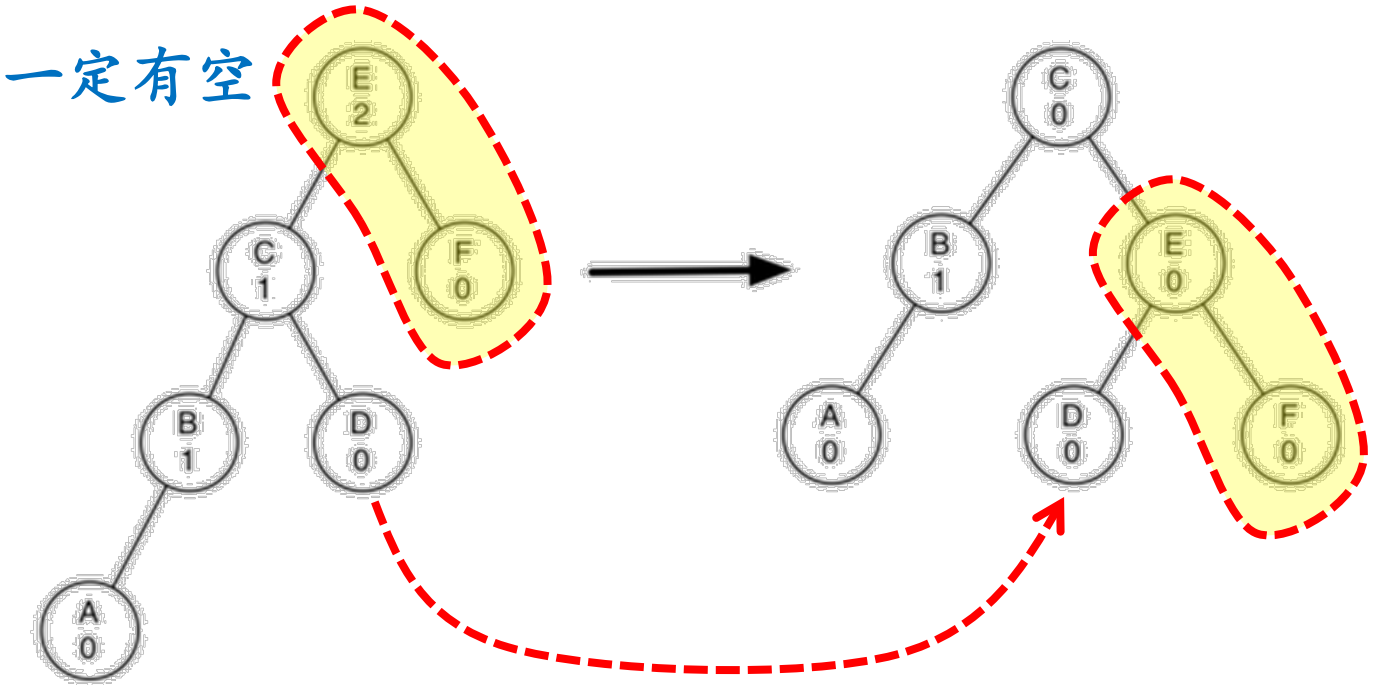


- **更复杂一些的情况：如图的“左重”子树右旋转**

  旋转后，新根节点将旧根节点作为右子节点，但是新根节点原来已有右子节点，需要将原有的右子节点重新定位！

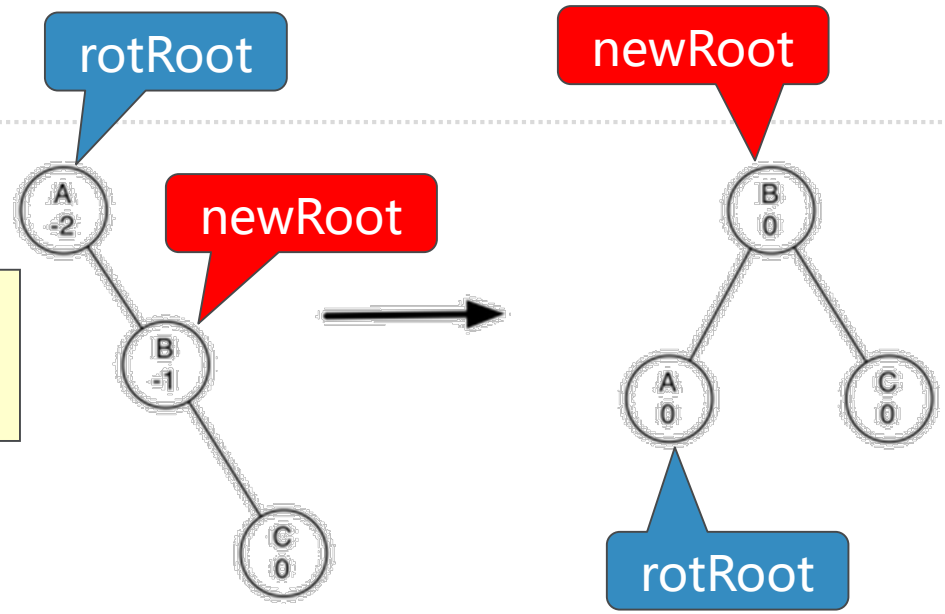
  原有的右子节点D改到旧根节点E的左子节点

  同样，E的左子节点在旋转后一定有空



# AVL树的实现：rotateLeft代码






```
def rotateLeft(self,rotRoot):
    newRoot = rotRoot.rightChild
    rotRoot.rightChild = newRoot.leftChild
    if newRoot.leftChild != None:
        newRoot.leftChild.parent = rotRoot
    newRoot.parent = rotRoot.parent
    if rotRoot.isRoot():
        self.root = newRoot
    else:
        if rotRoot.isLeftChild():
            rotRoot.parent.leftChild = newRoot
        else:
            rotRoot.parent.rightChild = newRoot
    newRoot.leftChild = rotRoot
    rotRoot.parent = newRoot
    rotRoot.balanceFactor = rotRoot.balanceFactor + \
                            1 - min(newRoot.balanceFactor, 0)
    newRoot.balanceFactor = newRoot.balanceFactor + \
                            1 + max(rotRoot.balanceFactor, 0)
```

仅有两个节点需要调整因子

# AVL树的实现：如何调整平衡因子



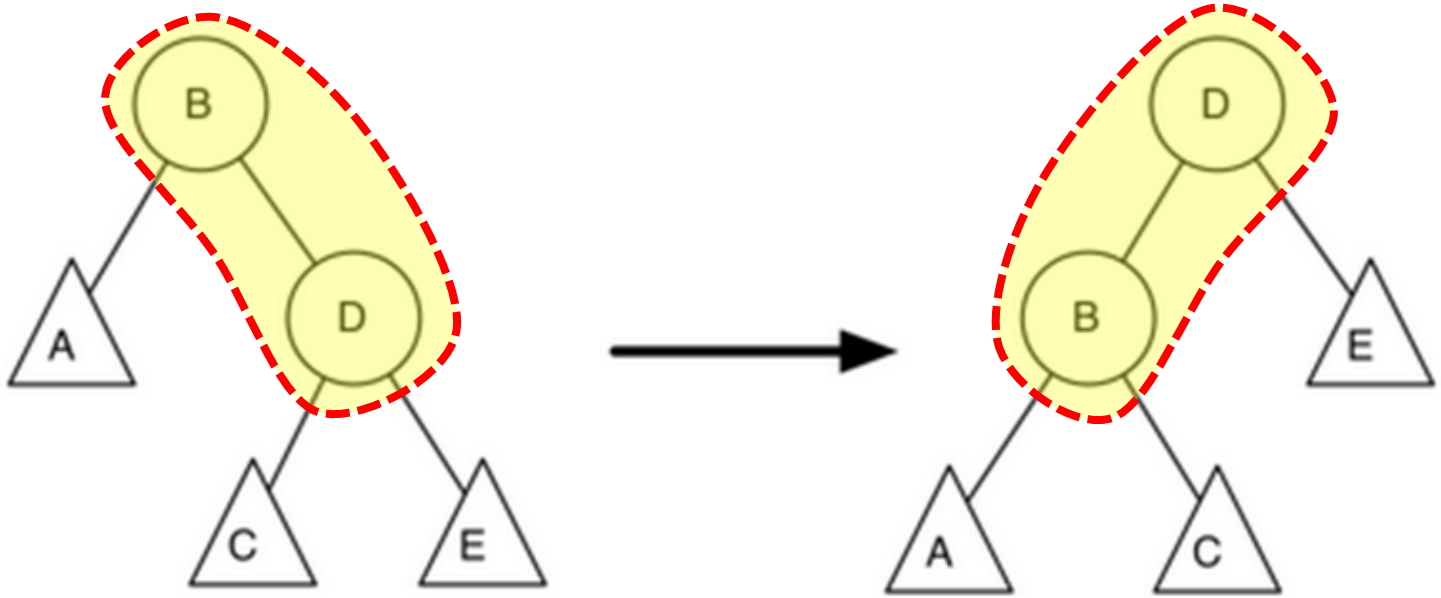
## ❖ 看看左旋转对平衡因子的影响

保持了次序ABCDE

ACE的平衡因子不变

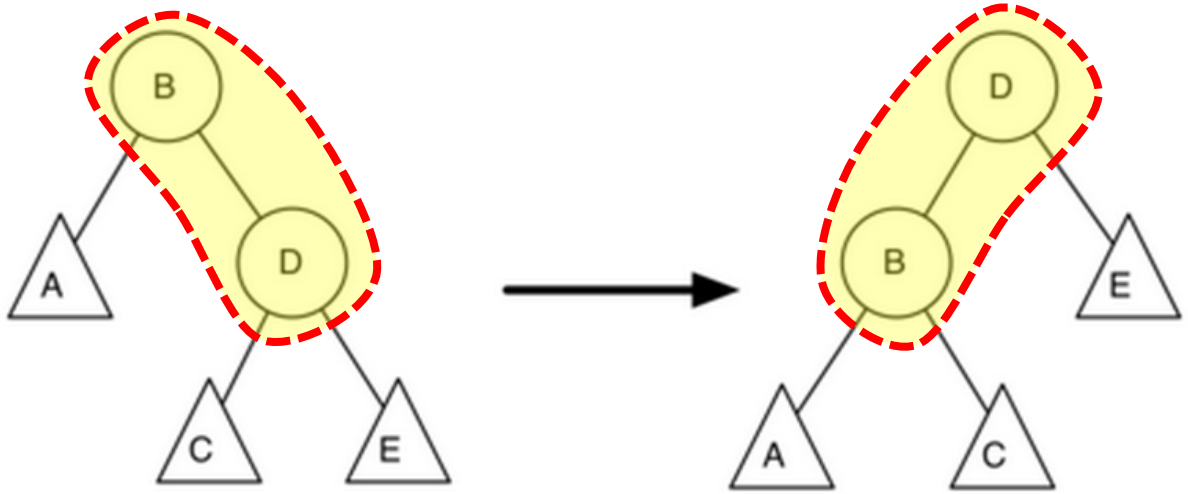
- hA/hC/hE不变

主要看BD新旧关系



# AVL树的实现：如何调整平衡因子






- 我们来看看B的变化

新B= hA- hC

旧B= hA- 旧hD

而：

旧hD= 1+ max(hC, hE)，所以旧B= hA- (1+ max(hC, hE))

新B- 旧B= 1+ max(hC, hE)- hC

新B= 旧B+ 1+ max(hC, hE)- hC；把hC移进max函数里就有

新B= 旧B+ 1+ max(0, -旧D) <==> 新B= 旧B+ 1- min(0, 旧D)

```
rotRoot.balanceFactor = rotRoot.balanceFactor + \
                        1 - min(newRoot.balanceFactor, 0)
```

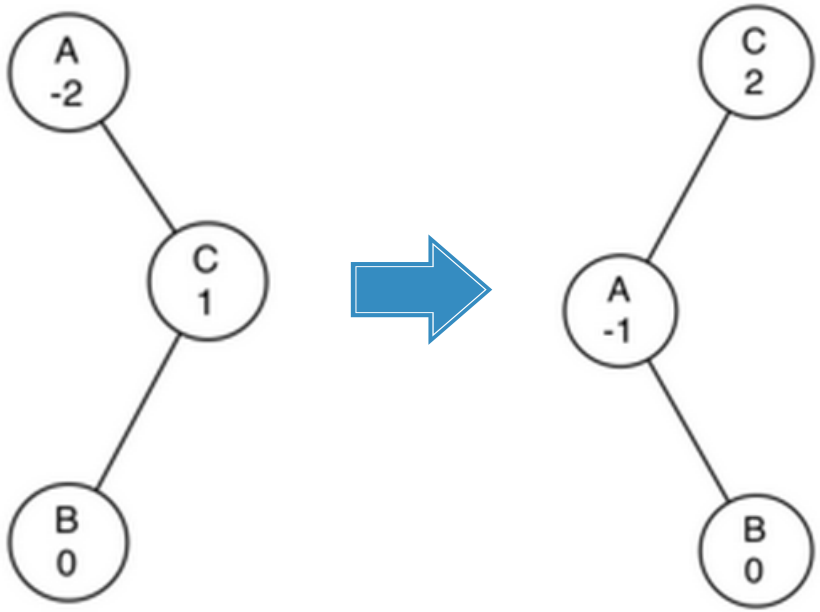
# AVL树的实现：更复杂的情形



- 下图的“右重”子树，单纯的左旋转无法实现平衡

  左旋转后变成“左重”了

  “左重”再右旋转，还回到“右重”



# AVL树的实现：更复杂的情形



- 所以，在左旋转之前检查右子节点的因子

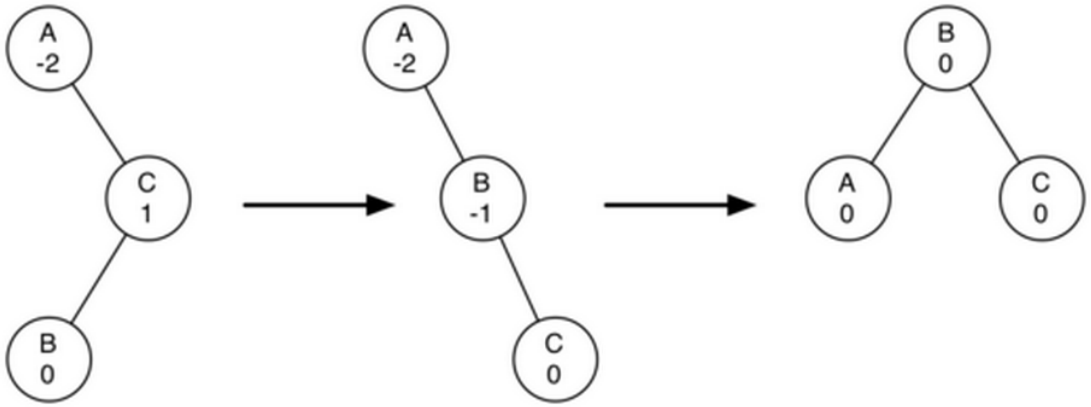
  如果右子节点“左重”的话，先对它进行右旋转

  再实施原来的左旋转

- 同样，在右旋转之前检查左子节点的因子

  如果左子节点“右重”的话，先对它进行左旋转

  再实施原来的右旋转



# AVL树的实现：rebalance代码



```
def rebalance(self,node):
    if node.balanceFactor < 0:
        if node.rightChild.balanceFactor > 0:
            # Do an LR Rotation
            self.rotateRight(node.rightChild)
            self.rotateLeft(node)
        else:
            # single left
            self.rotateLeft(node)
    elif node.balanceFactor > 0:
        if node.leftChild.balanceFactor < 0:
            # Do an RL Rotation
            self.rotateLeft(node.leftChild)
            self.rotateRight(node)
        else:
            # single right
            self.rotateRight(node)
```

右重需要左旋

右子节点左重
先右旋

左重需要右旋

左子节点右重
先左旋

# AVL树的实现：结语



- 经过复杂的put方法，AVL树始终维持平衡，get方法也始终保持O(log n)高性能

  不过，put方法的代价有多大？

- 将AVL树的put方法分为两个部分：

  需要插入的新节点是叶节点，更新其所有父节点和祖先节点的代价最多为O(log n)

  如果插入的新节点引发了不平衡，重新平衡最多需要2次旋转，但旋转的代价与问题规模无关，是常数O(1)

  所以整个put方法的时间复杂度还是O(log n)