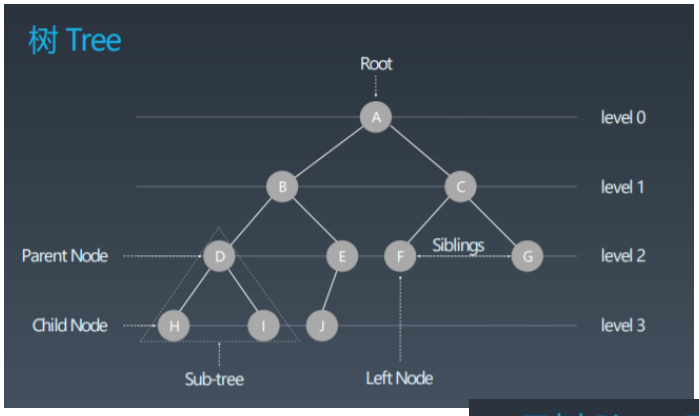
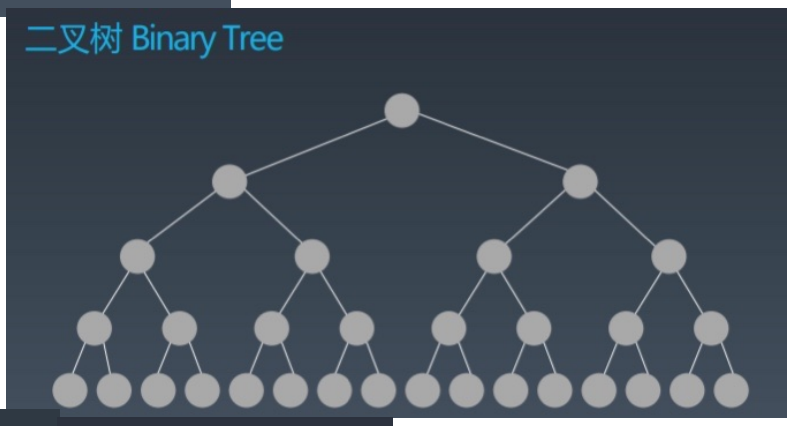
Review:





二叉树遍历

1. 前序(Pre-order)：根-左-右
2. 中序(In-order)：左-根-右 (有序)
3. 后序(Post-order)：左-右-根

示例代码

```
def preorder(self, root):

  if root:
    self traverse_path append(root val)
    self preorder(root left)
    self preorder(root right)

def inorder(self, root):

  if root:
    self inorder(root left)
    self traverse_path append(root val)
    self inorder(root right)

def postorder(self, root):

  if root:
    self postorder(root left)
    self postorder(root right)
    self traverse_path append(root val)
```

树和链表没有本质上的区别 (一个链表分出两个 next，就可以称之为树)

二叉搜索树 Binary Search Tree

时间复杂度
=
此树的深度

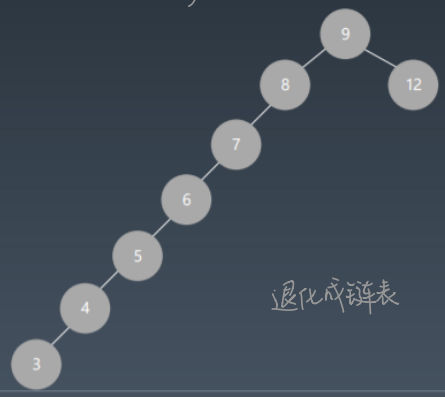
二叉搜索树 Binary Search Tree

二叉搜索树，也称二叉搜索树、有序二叉树（Ordered Binary Tree）、排序二叉树（Sorted Binary Tree），是指一棵空树或者具有下列性质的二叉树：

1. 左子树上所有结点的值均小于它的根结点的值；
2. 右子树上所有结点的值均大于它的根结点的值；
3. 以此类推：左、右子树也分别为二叉查找树。（这就是 重复性！）

中序遍历：升序排列

极端情况（二叉搜索树）



退化成链表

保证性能的关键

1. 保证二维维度！ —> 左右子树结点平衡（recursively）
2. Balanced 平衡二叉树
3. https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree

AVL Tree

AVL 树

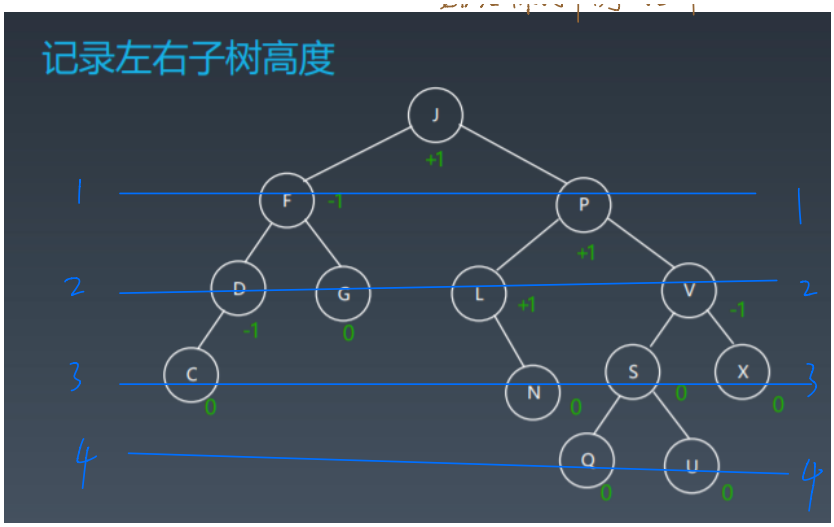
1. 发明者 G. M. Adelson-Velsky 和 Evgenii Landis
2. Balance Factor（平衡因子）：
   是它的左子树的高度减去它的右子树的高度（有时相反）。
   balance factor = {-1, 0, 1}
3. 通过旋转操作来进行平衡（四种）
4. https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree

查询时间复杂度 = 深度

查询二叉搜索树效率又与高度有关，

和结点个数没有关系

始终保持平衡因子都处在 -1，0，1



左子树高度：3

右子树高度：4

二、平衡因子 = 右子树的高度 − 左子树的高度

= 4 − 3

= 1

所有叶子结点，因为左右子树没有，所以高度就是0

F点：右边 − 左边

= 1 − 2

= −1

G点：0

D点：右边 − 左边

= 0 − 1

= −1

example 1

① 14 比 13 大 所以去右边

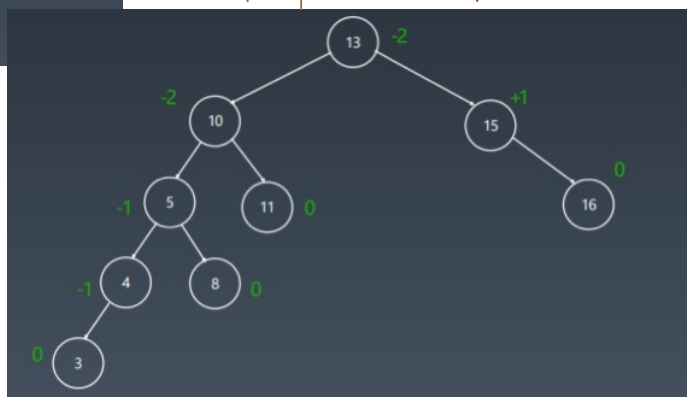
② 14 比 15 小 所以去左边

③ 调整平衡因子

example 2

① 往左走，走到4左下

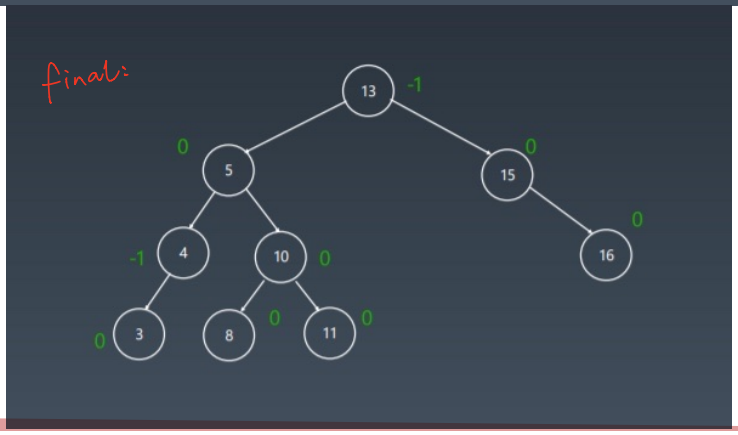
② 两个结节平衡因子不再是{-1, 0, 1}了



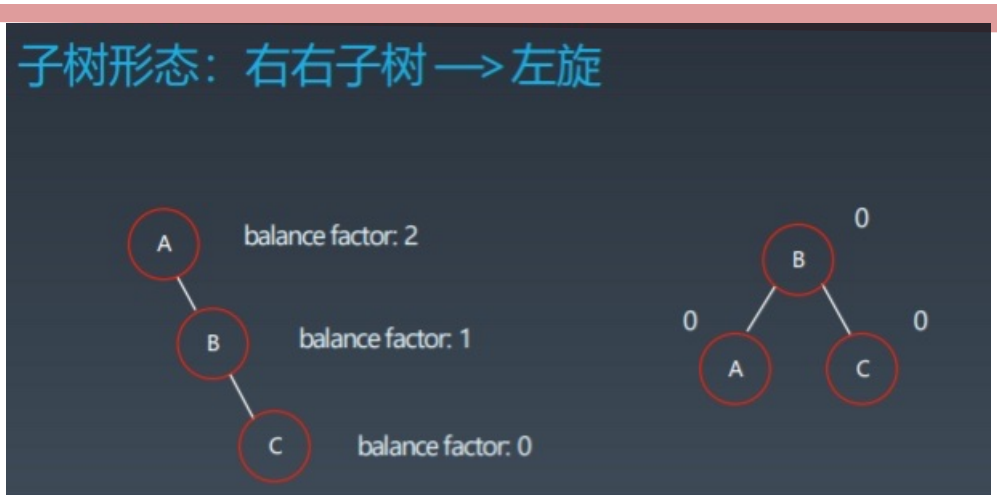
③ 旋转操作

旋转操作

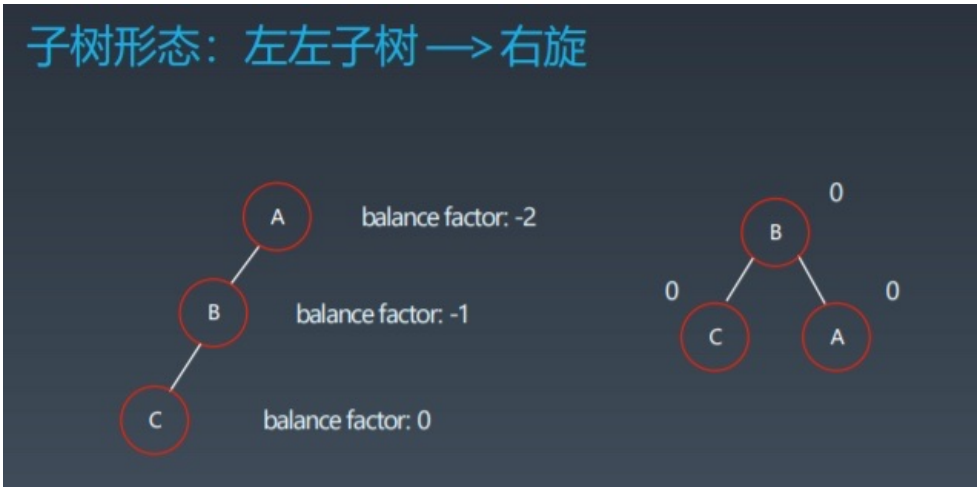
1. 左旋
2. 右旋
3. 左右旋
4. 右左旋



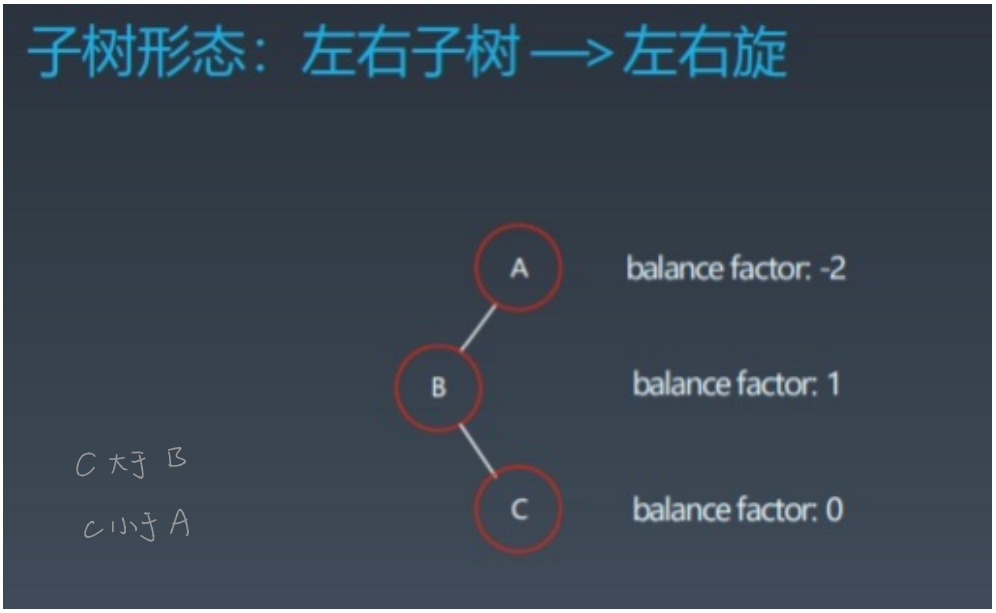
子树形态：右右子树 —> 左旋



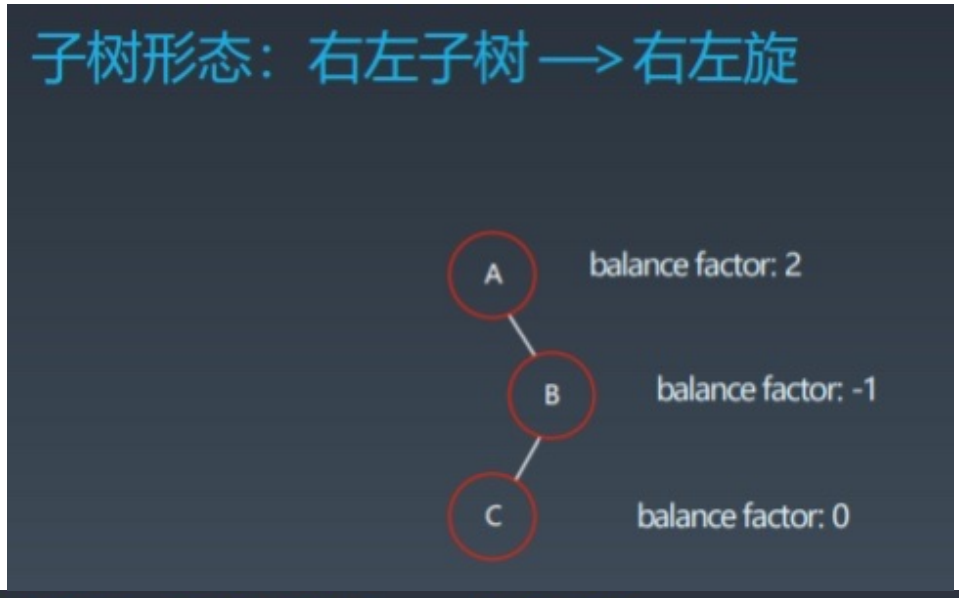
子树形态：左左子树 —> 右旋

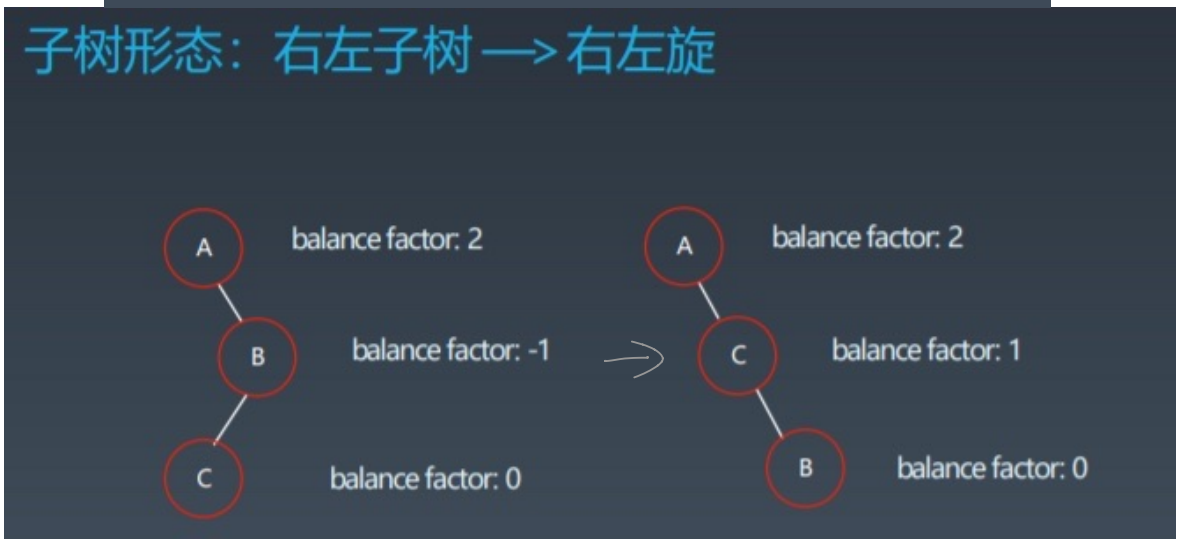


子树形态：左右子树 —> 左右旋

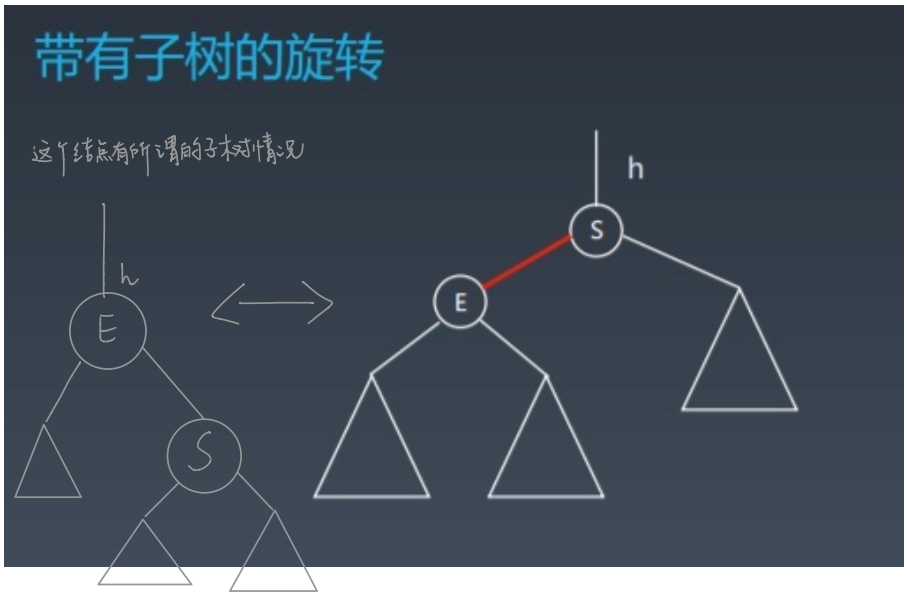



子树形态：左右子树 —> 左右旋
A
balance factor: -2
C
balance factor: -1
B
balance factor: 0
0
C
0
0
B
A

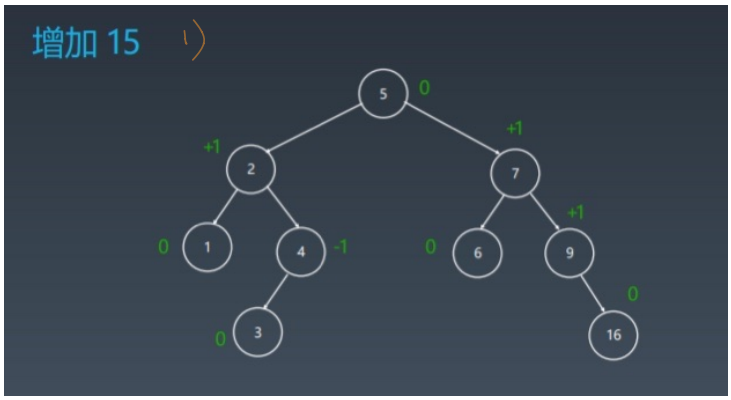

子树形态：右左子树 —> 右左旋
A
balance factor: 2
B
balance factor: -1
C
balance factor: 0


子树形态：右左子树 —> 右左旋
A
balance factor: 2
B
balance factor: -1
C
balance factor: 0
A
balance factor: 2
C
balance factor: 1
B
balance factor: 0

记录它左右子树深度，同时保证任何时候任何一个节点，它的左右子树的深度差不超过绝对值1

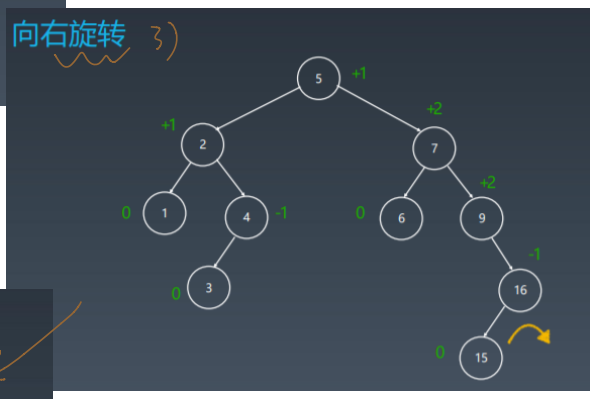


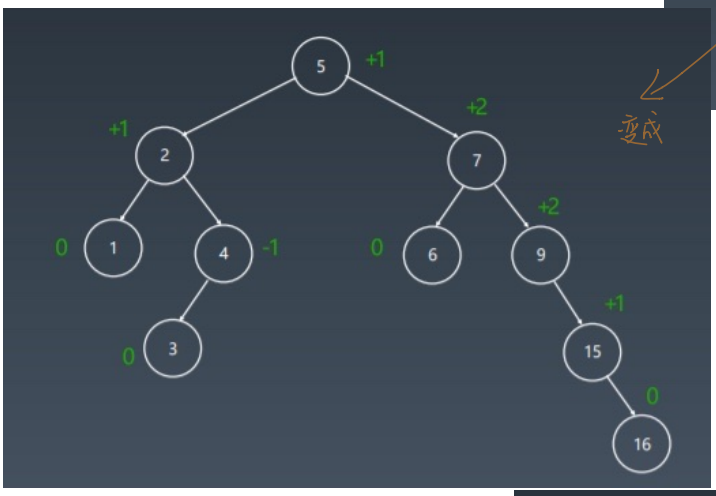
example 3



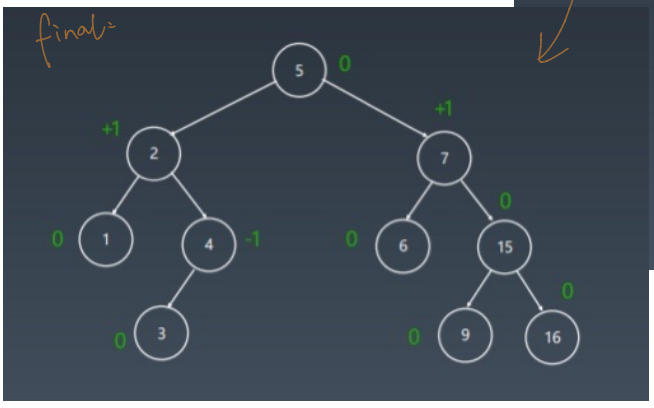

增加 15
2)
5
+1
2
+1
7
+2
1
0
4
-1
6
0
9
+2
3
0
16
-1
15
0

(右左子树)


向右旋转
3)
5
+1
2
+1
7
+2
1
0
4
-1
6
0
9
+2
3
0
16
-1
15
0


变成
5
+1
2
+1
7
+2
1
0
4
-1
6
0
9
+2
3
0
15
+1
16
0


向左旋转
4)
5
+1
2
+1
7
+2
1
0
4
-1
6
0
9
+2
3
0
15
+1
16
0


final:
5
0
2
+1
7
+1
1
0
4
-1
6
0
15
0
3
0
9
0
16
0

AVL 总结

1. 平衡二叉搜索树
2. 每个结点存 balance factor = {-1, 0, 1}
3. 四种旋转操作

不足：结点需要存储额外信息、且调整次数频繁

且存储的信息是 int 型的 → 到会要 3 个值来表示

Red-black Tree

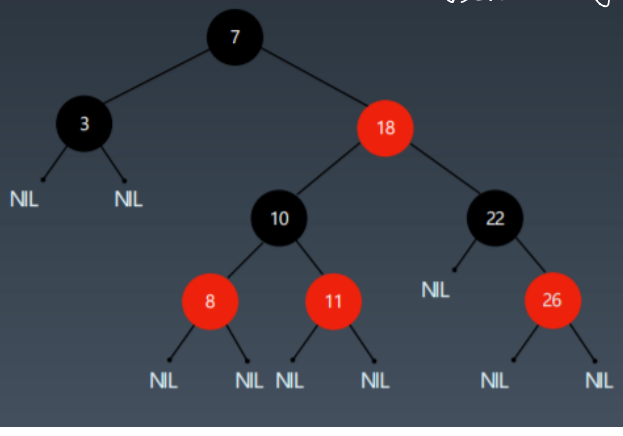
红黑树是一种近似平衡的二叉搜索树（Binary Search Tree），它能够确保任何一个结点的左右子树的高度差小于两倍。具体来说，红黑树是满足如下条件的二叉搜索树：

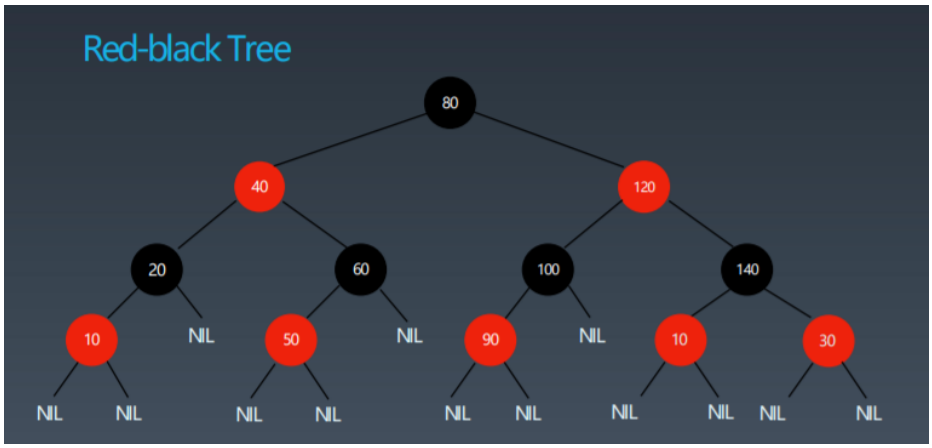
5 个特点

- 每个结点要么是红色，要么是黑色
- 根结点是黑色
- 每个叶结点（NIL结点，空结点）是黑色的。
- 不能有相邻接的两个红色结点
- 从任一结点到其每个叶子的所有路径都包含相同数目的黑色结点。

Red-black Tree

时间复杂度：$O(\log(n))$





关键性质：

从根到叶子的最长的可能路径不多于最短的可能路径的两倍长

对比

- AVL trees provide faster lookups than Red Black Trees because they are more strictly balanced.
- Red Black Trees provide faster insertion and removal operations than AVL trees as fewer rotations are done due to relatively relaxed balancing.
- AVL trees store balance factors or heights with each node, thus requires storage for an integer per node whereas Red Black Tree requires only 1 bit of information per node.
- Red Black Trees are used in most of the language libraries like map, multimap, multisetin C++ whereas AVL trees are used in databases where faster retrievals are required.

- 读/查找性能 AVL更好
- 红黑树 提供了更快的插入和删除的操作 (∵ AVL旋转操作更多)
- AVL 额外的信息要存 factor 和 height 更多一点，需要用更多的内存 附加在每个结点里面来存这些额外的信息
- 红黑树 要的信息非常少，它只要一个 bit 就是来存 0 和 1 表示黑或是红，所以它对额外空间的消耗更小

Database 一般用 AVL