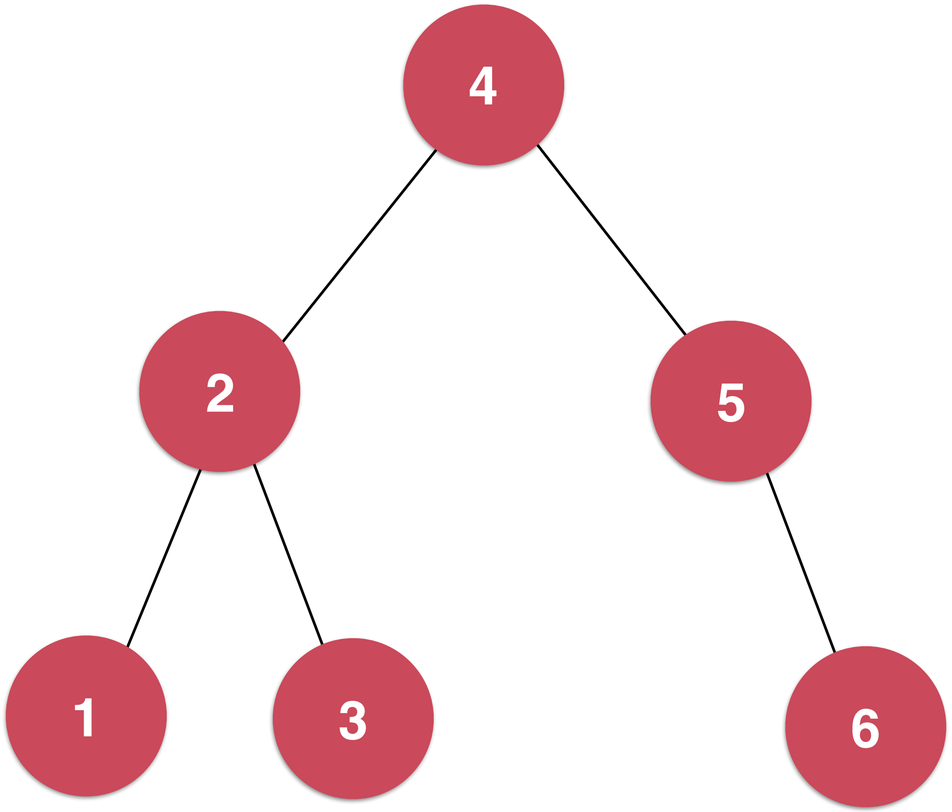
玩儿转数据结构

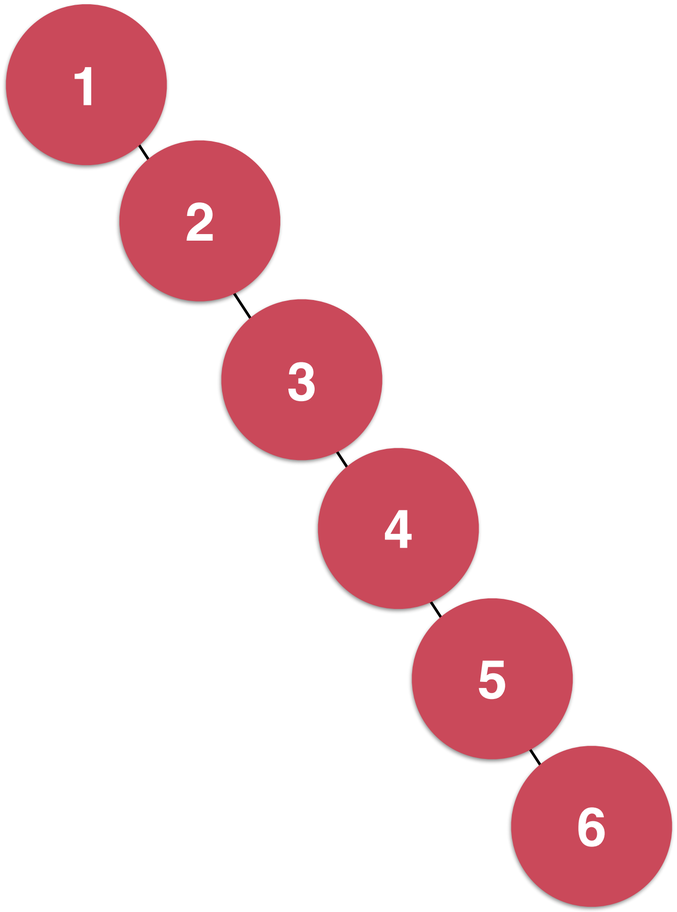
liuyubobobo

平衡二叉树与AVL树

回忆二分搜索树的问题

1，2，3，4，5，6





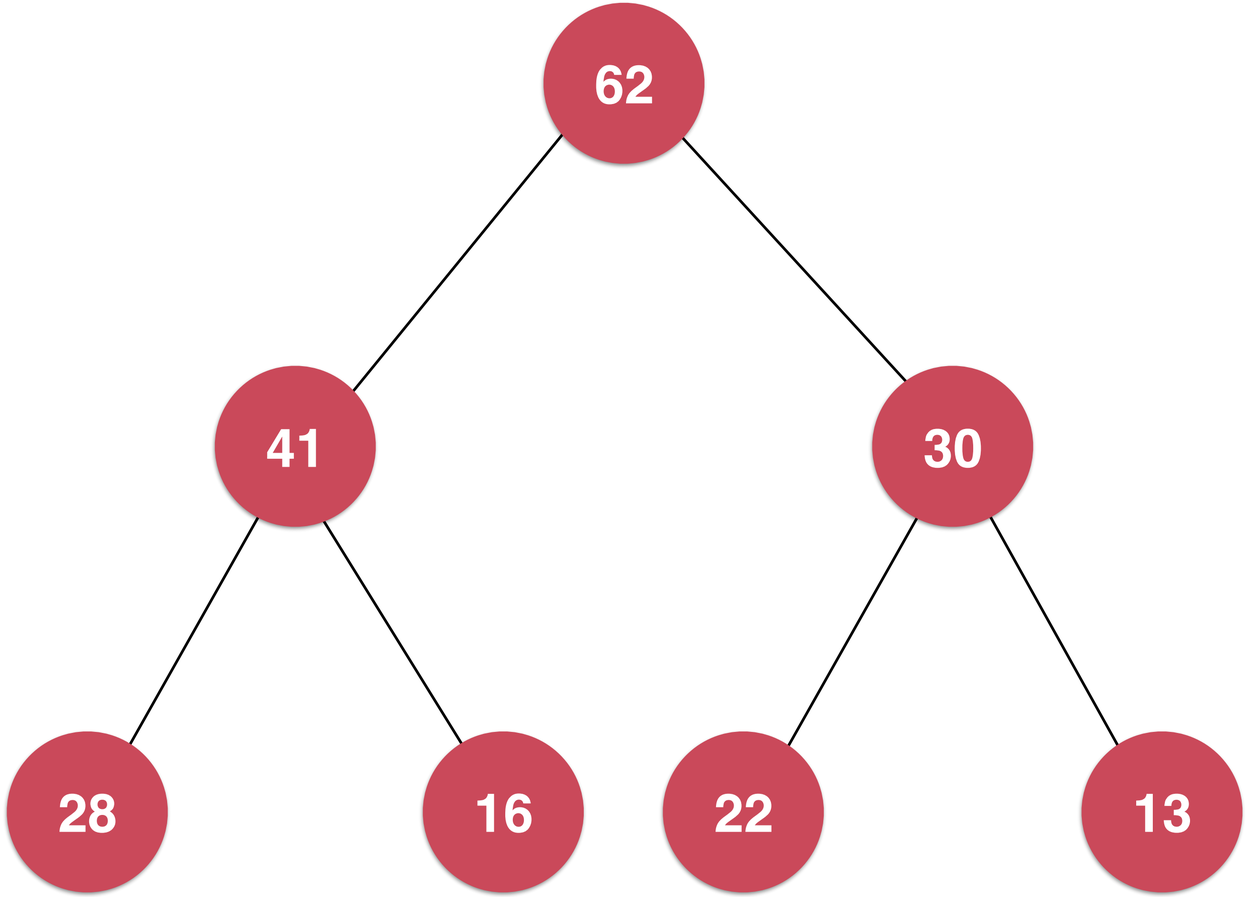
AVL 树

G. M. Adelson-Velsky 和 E. M. Landis

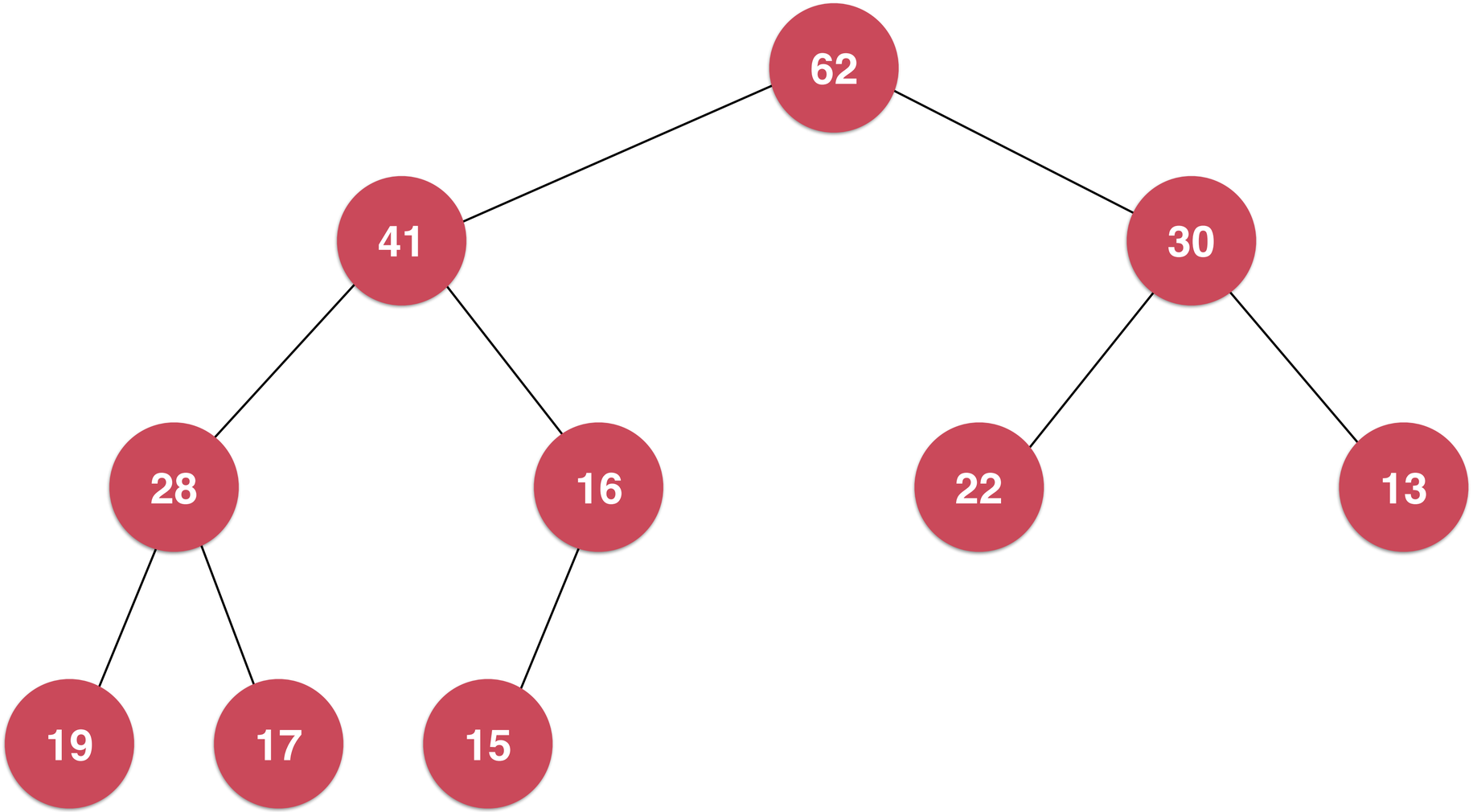
1962年的论文首次提出

最早的自平衡二分搜索树结构

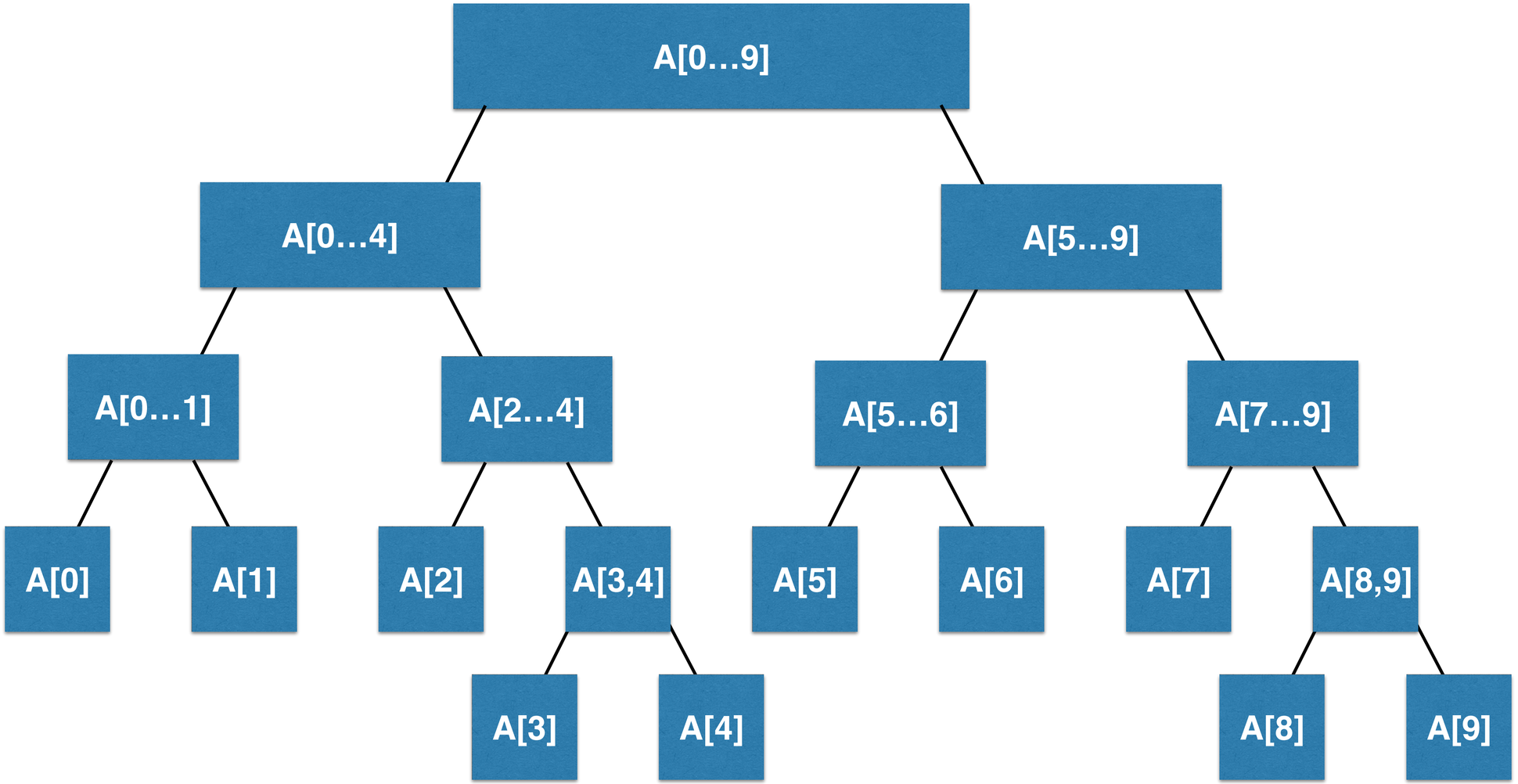
平衡二叉树



平衡二叉树



平衡二叉树



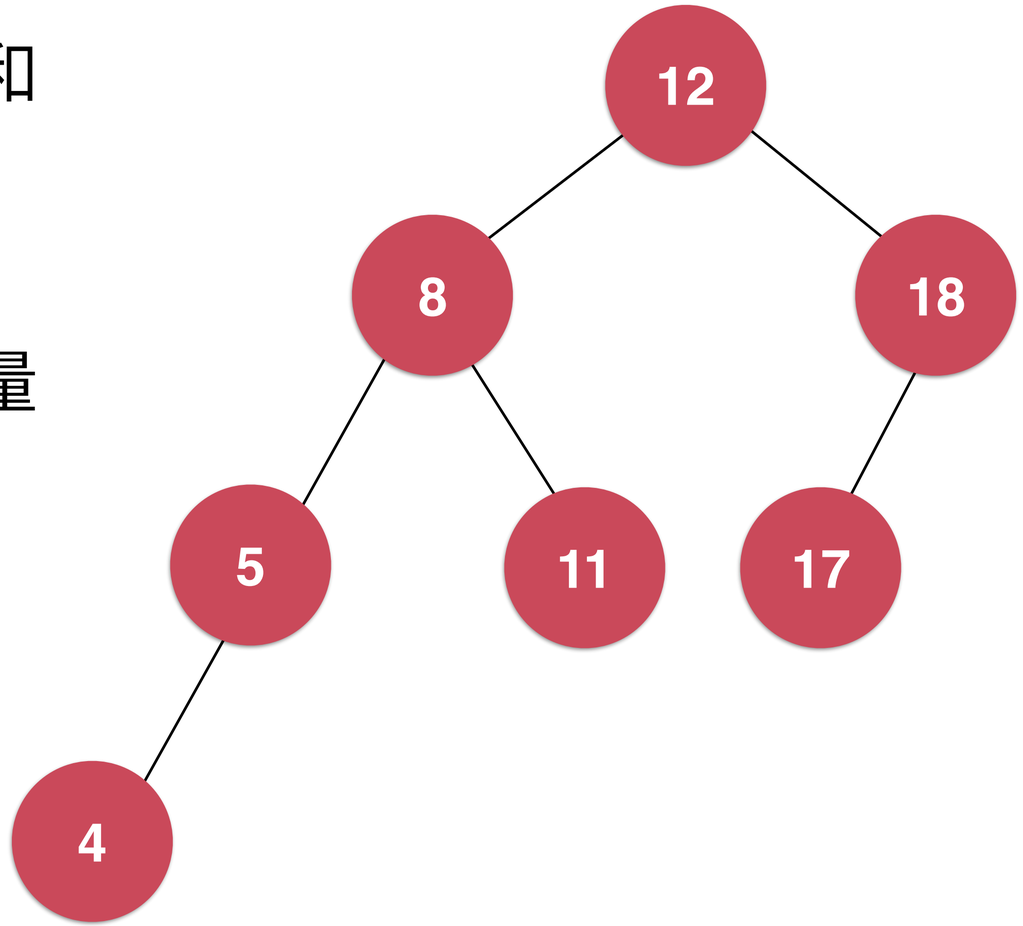
平衡二叉树

对于任意一个节点，左子树和右子树的高度差不能为超过1

平衡二叉树

对于任意一个节点，左子树和右子树的高度差不能为超过1

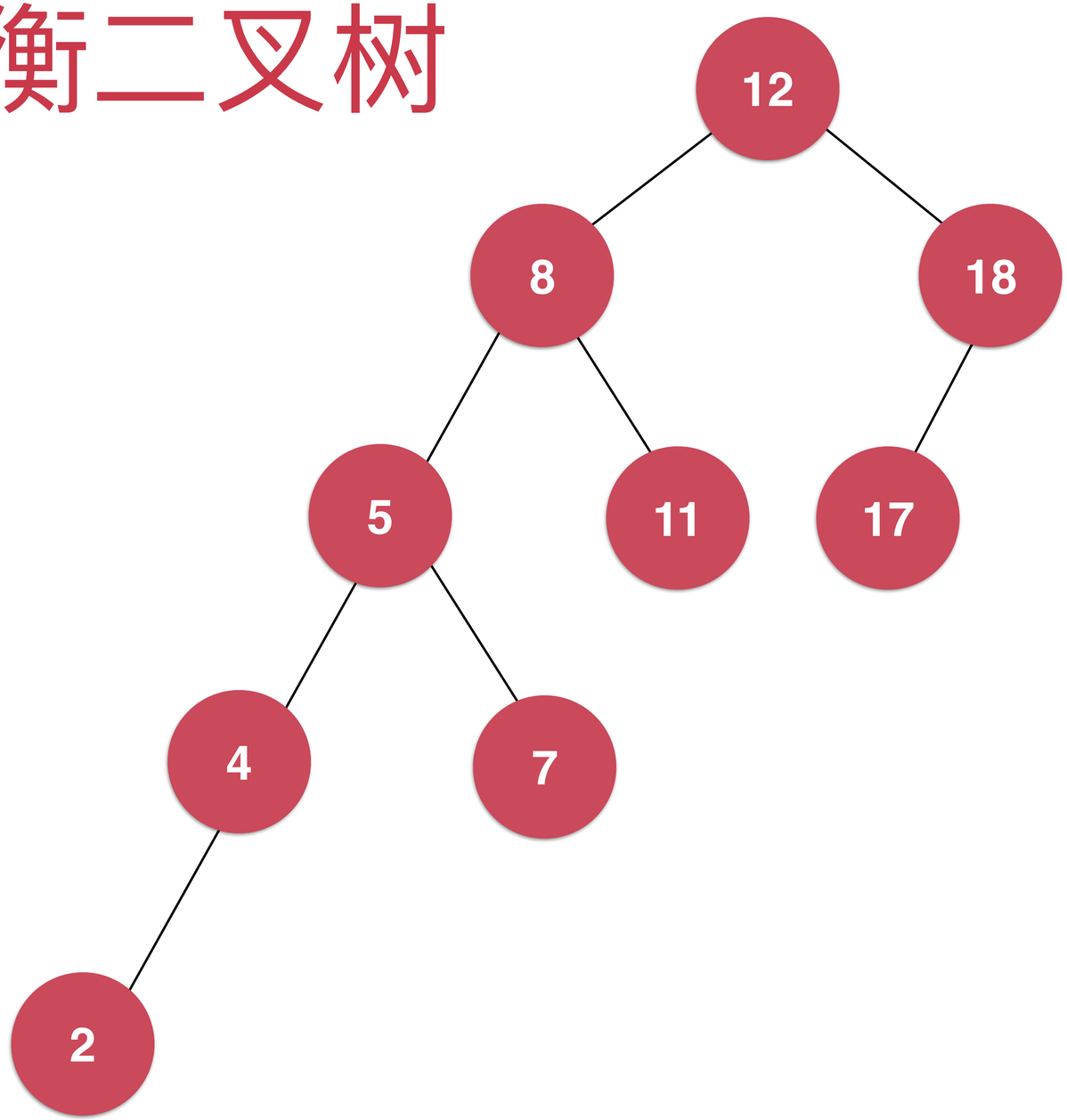
平衡二叉树的高度和节点数量之间的关系也是O(logn)的



平衡二叉树

对于任意一个节点，左子树和右子树的高度差不能为超过1

平衡二叉树的高度和节点数量之间的关系也是O(logn)的

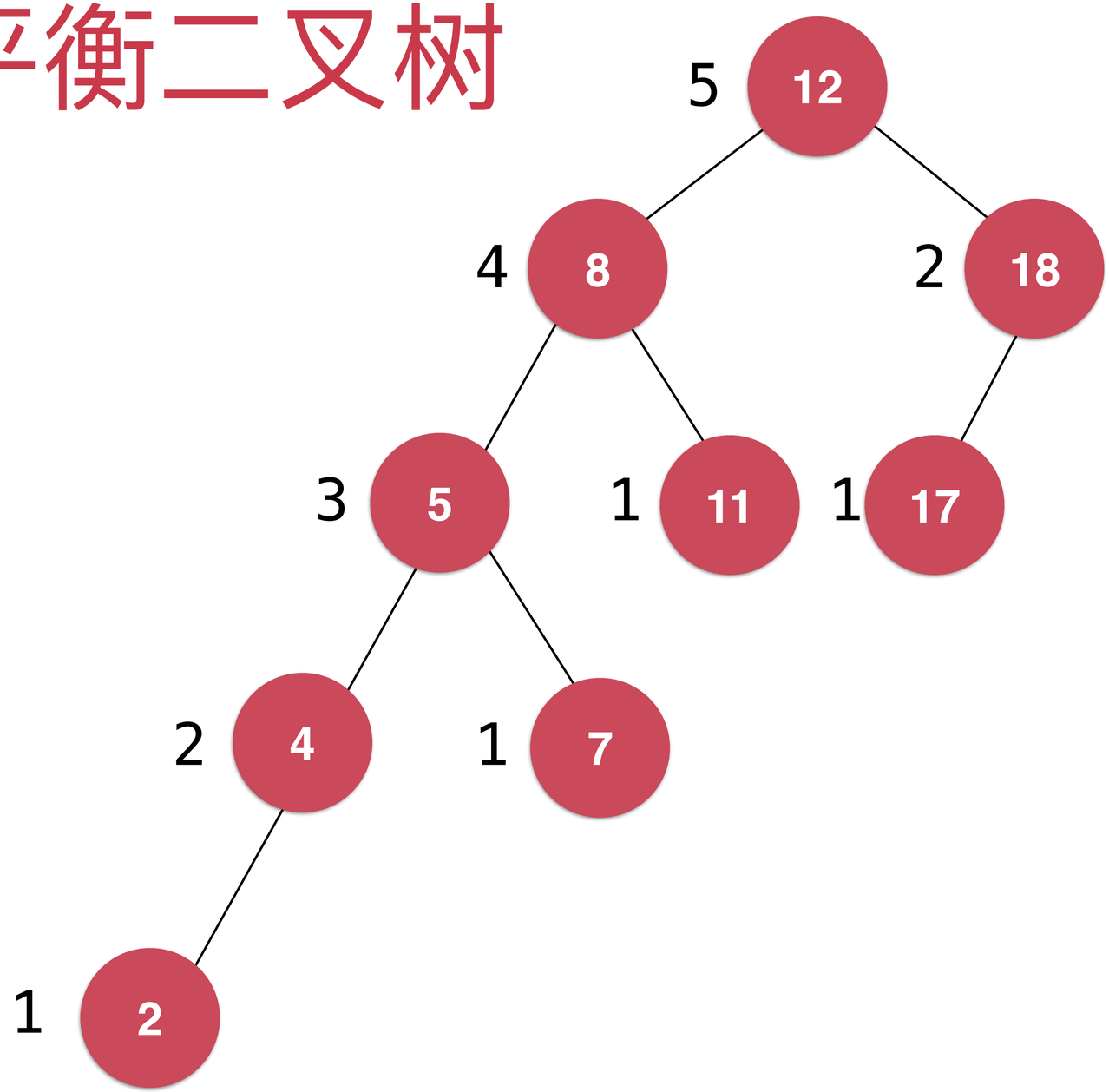


平衡二叉树

对于任意一个节点，左子树和右子树的高度差不能为超过1

平衡二叉树的高度和节点数量之间的关系也是O(logn)的

标注节点的高度



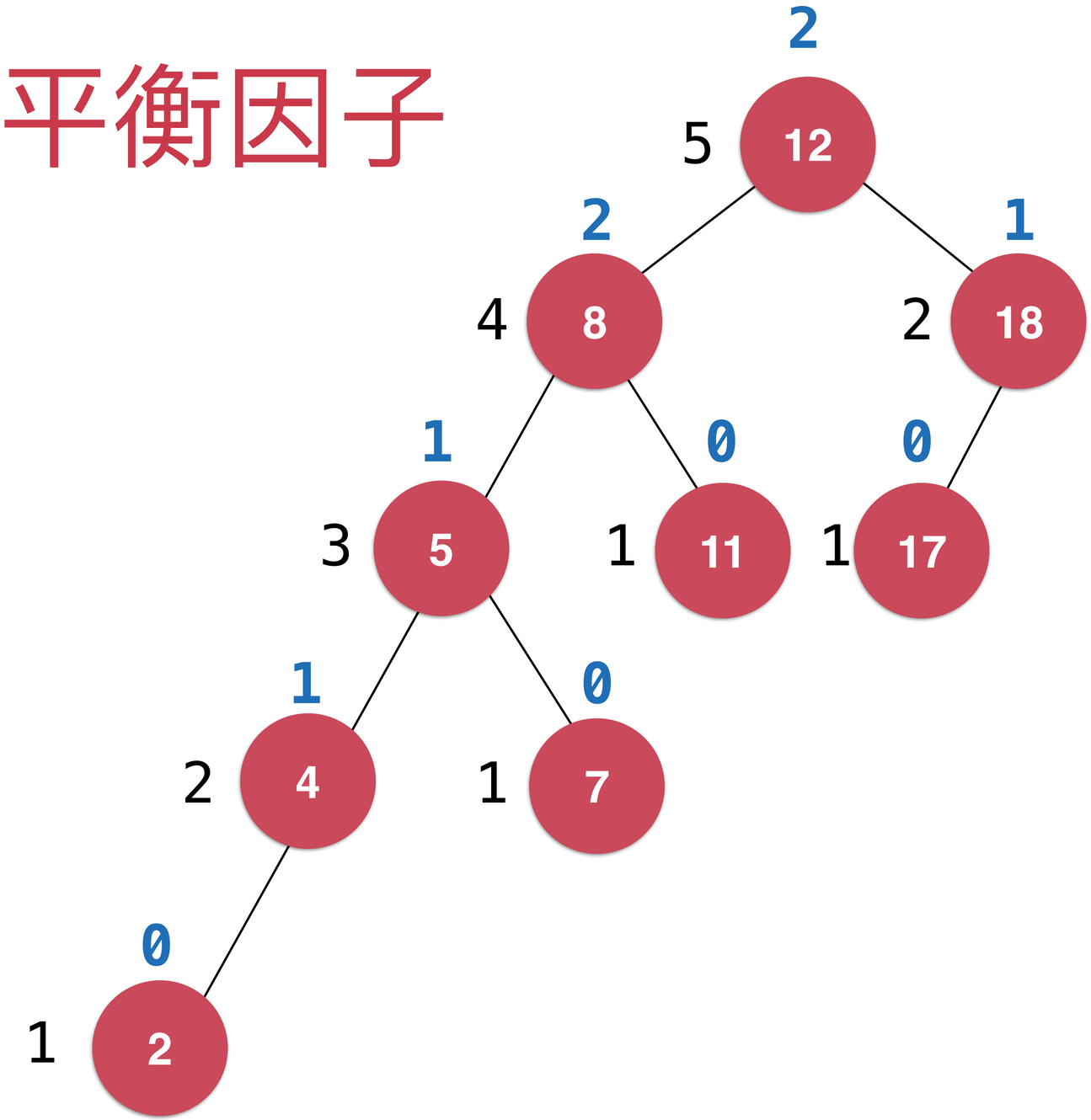
平衡因子

对于任意一个节点，左子树和右子树的高度差不能为超过1

平衡二叉树的高度和节点数量之间的关系也是O(logn)的

标注节点的高度

计算平衡因子



在二分搜索树中记录节点高度和计算平衡因子

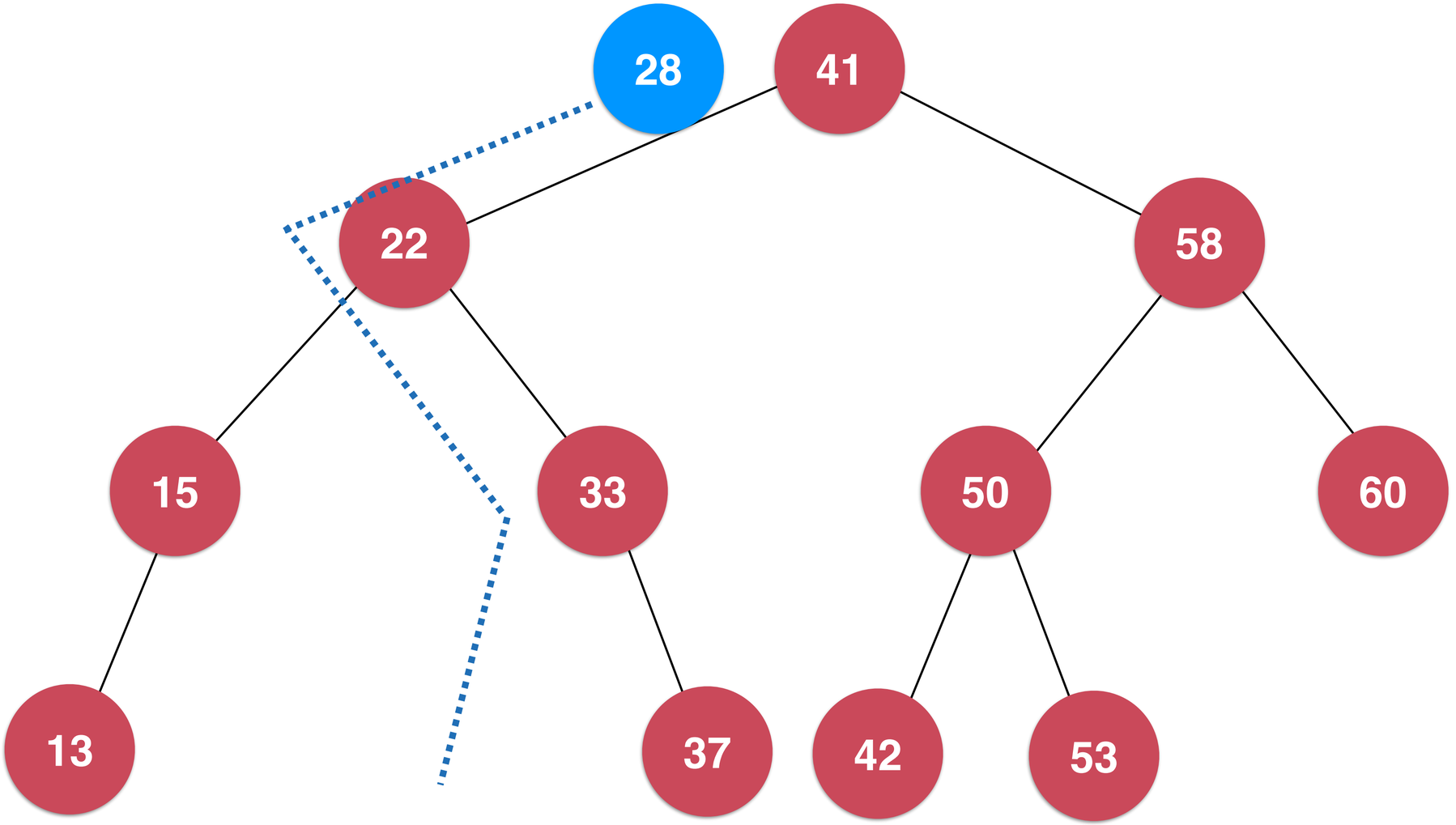
实践：计算高度和平衡因子

验证二分搜索树性质和平衡性

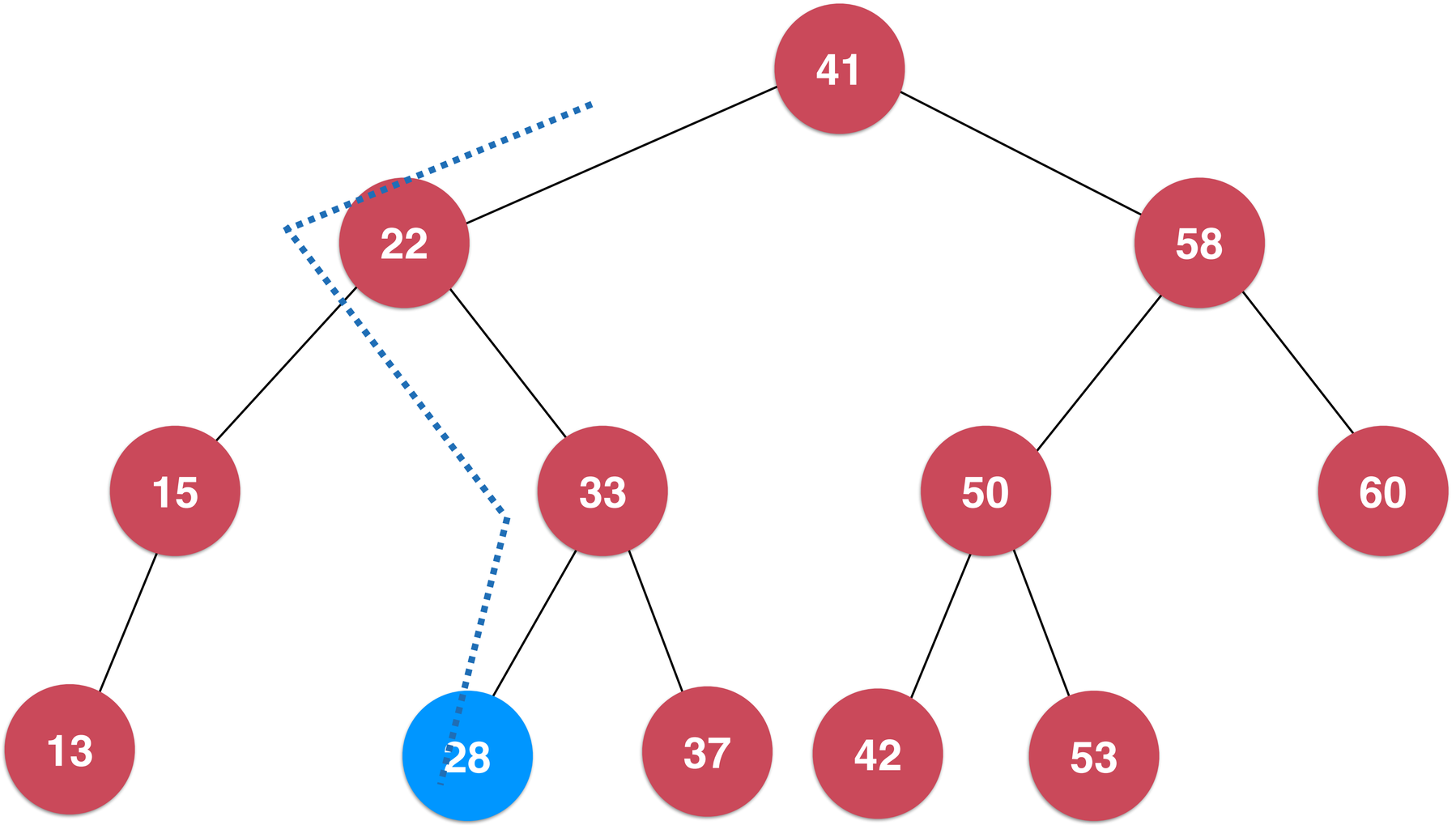
实践：验证二分搜索树性质和平衡性

AVL树的左旋转和右旋转

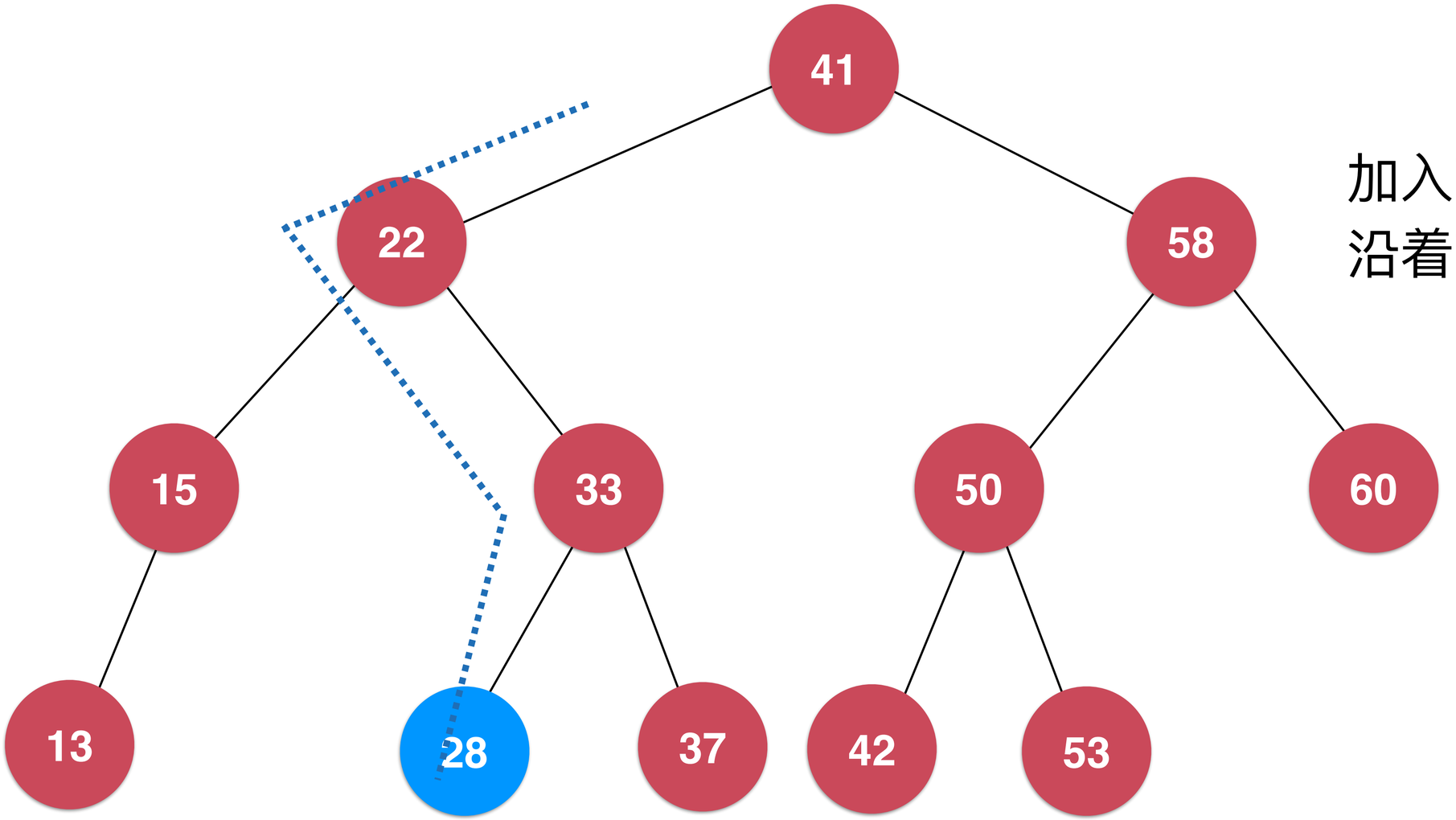
在什么时候维护平衡



在什么时候维护平衡



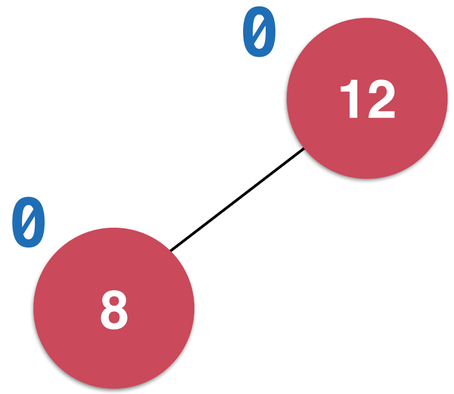
在什么时候维护平衡



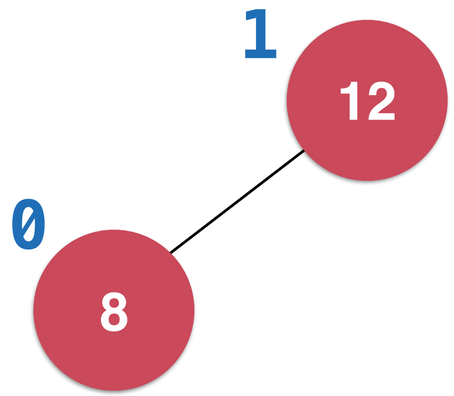
加入节点后，
沿着节点向上维护平衡性

在什么时候维护平衡?

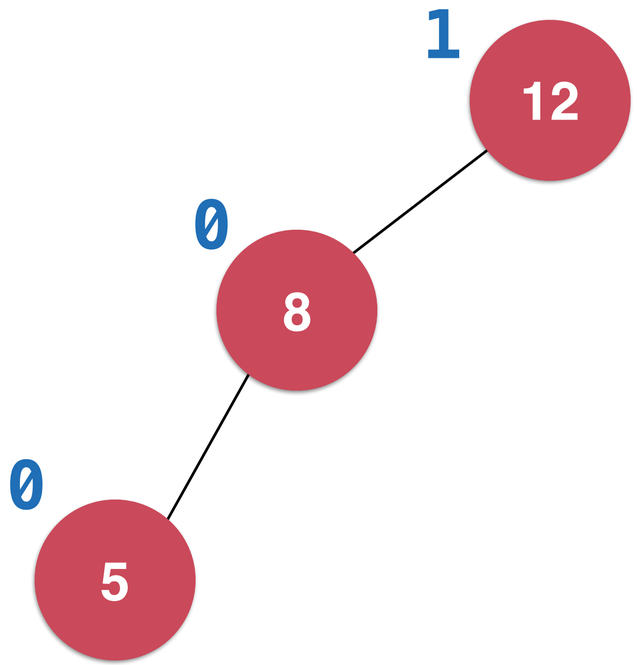
在什么时候维护平衡?



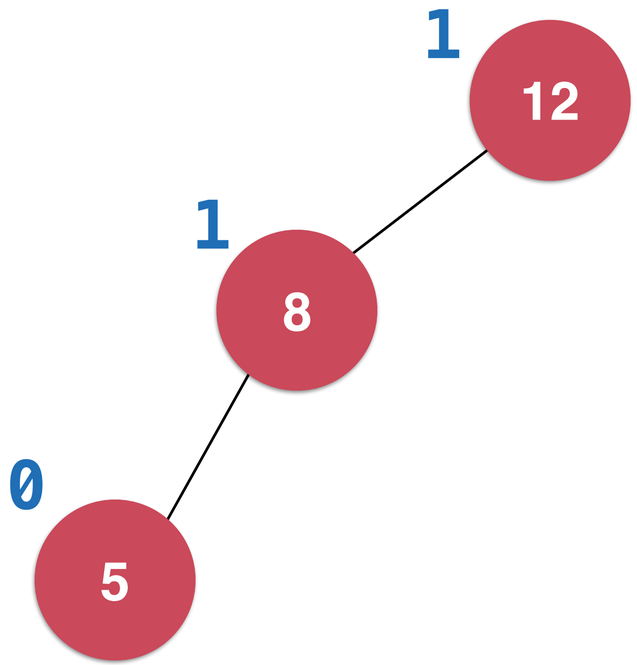
在什么时候维护平衡?



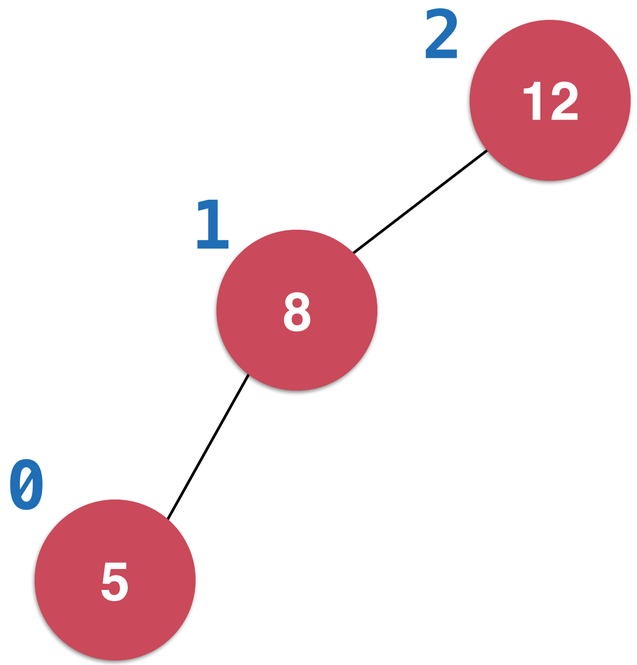
在什么时候维护平衡?



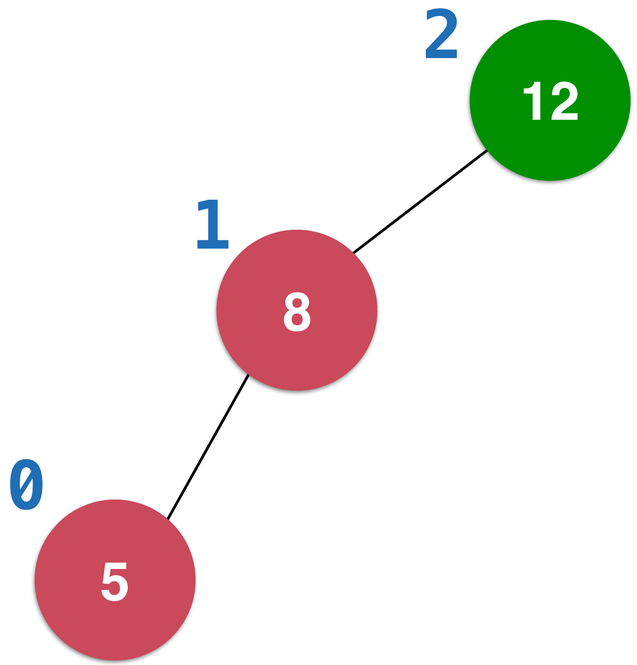
在什么时候维护平衡?



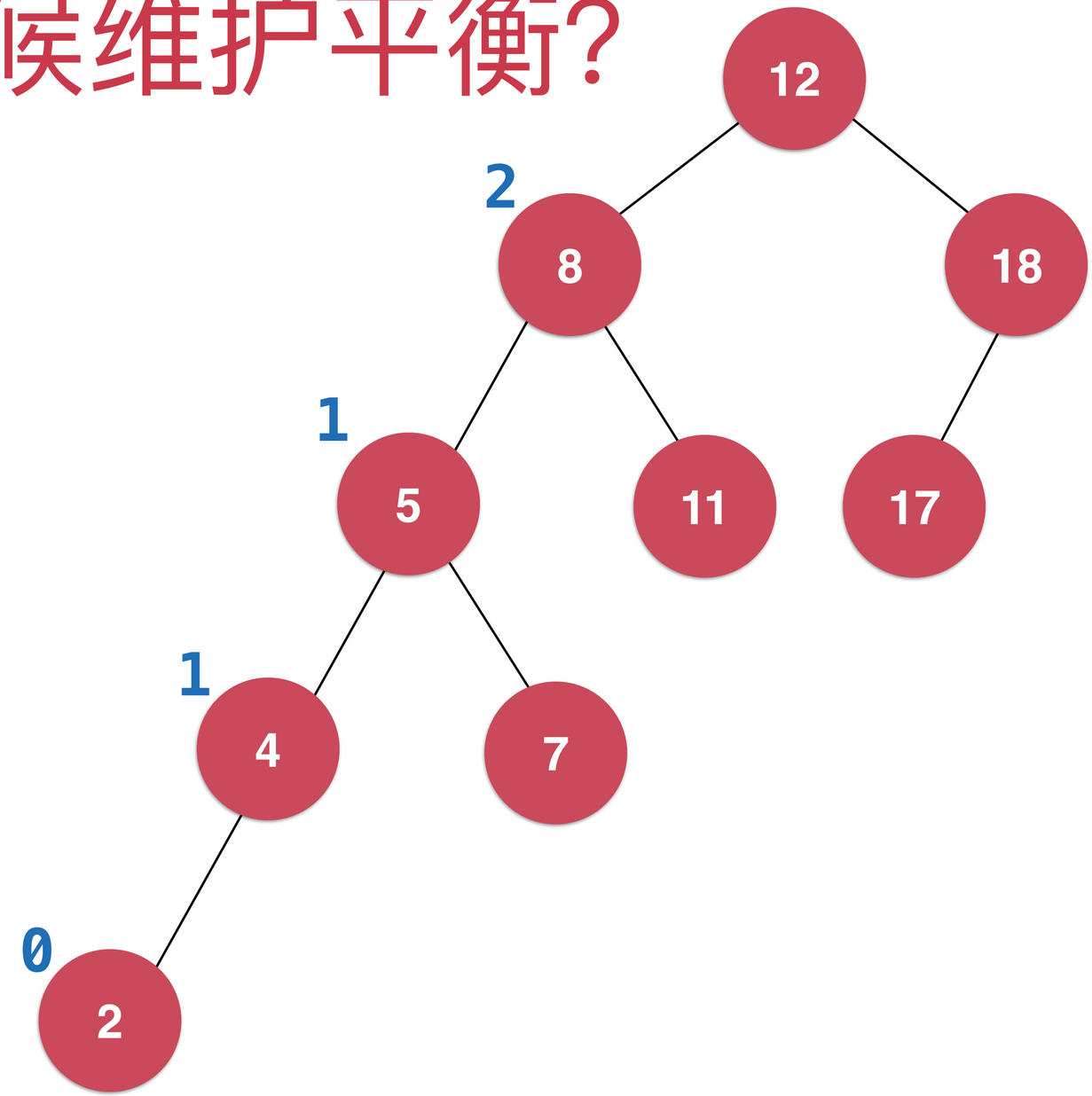
在什么时候维护平衡?



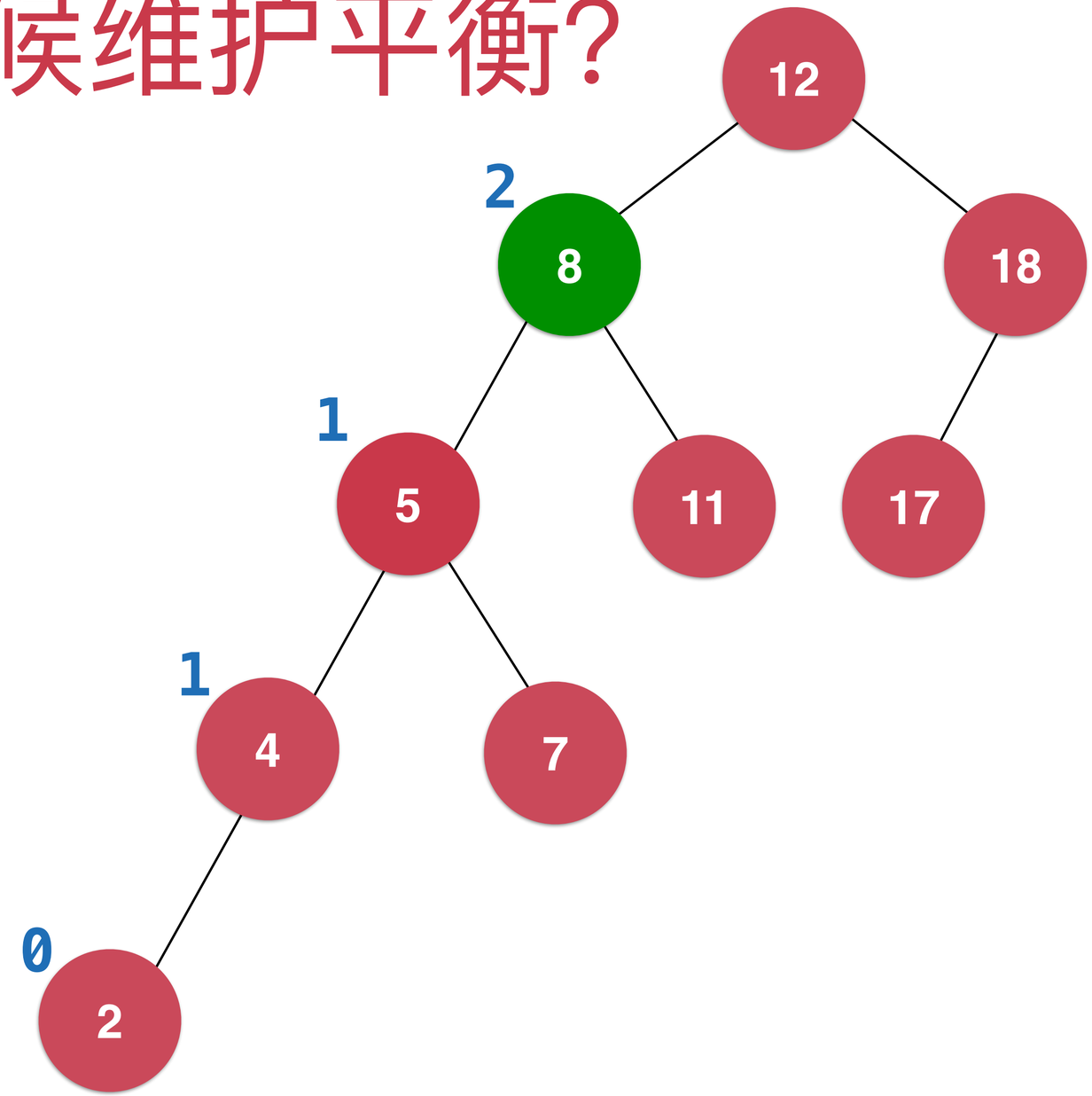
在什么时候维护平衡?



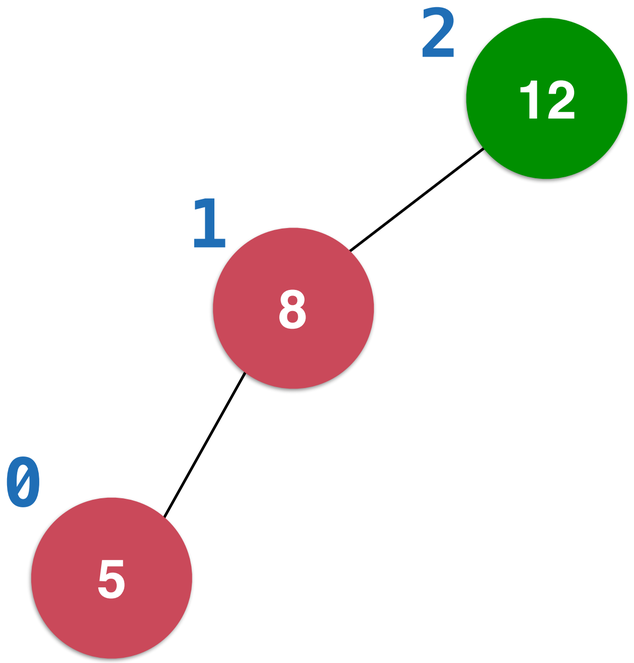
在什么时候维护平衡?

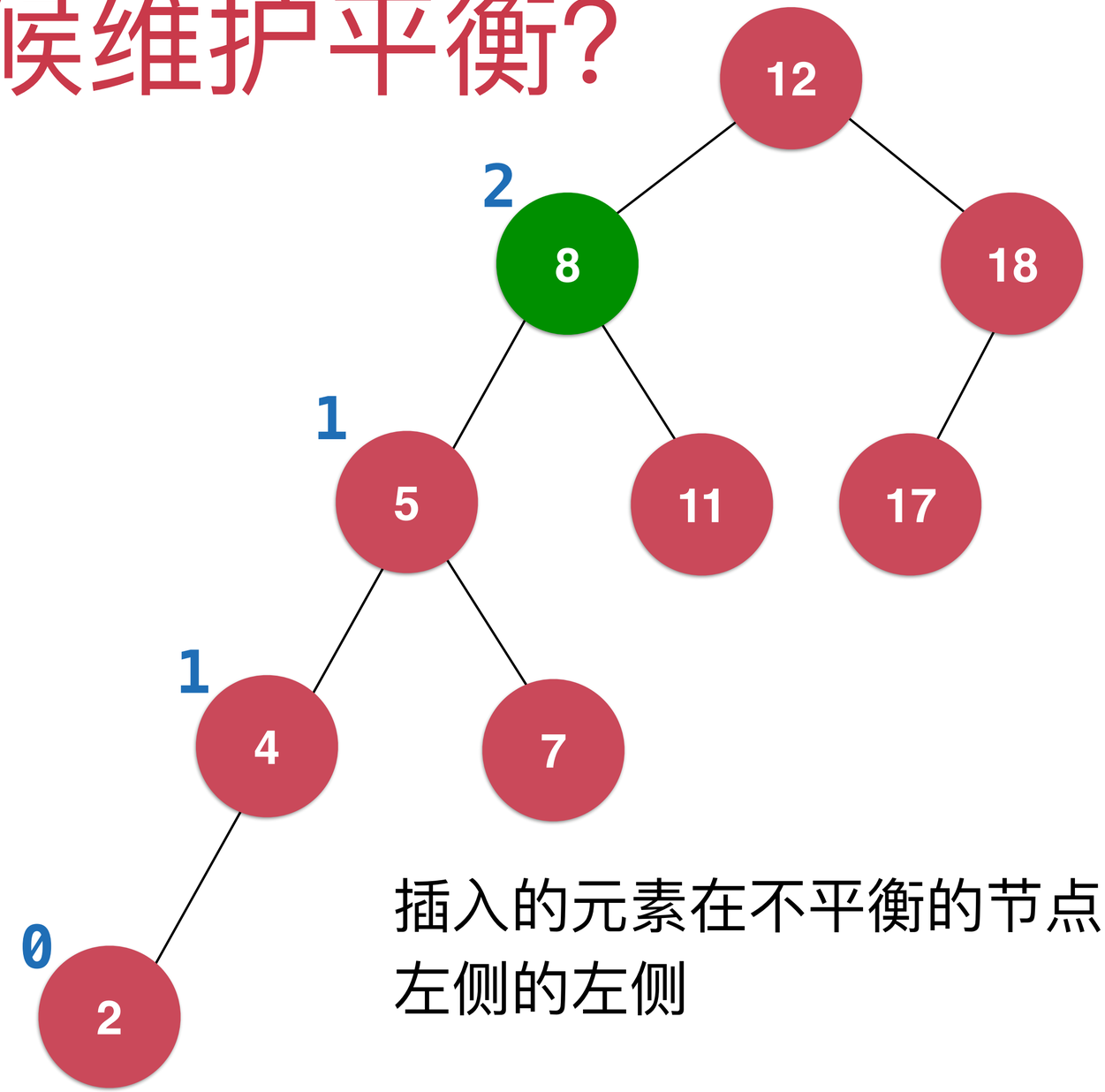


在什么时候维护平衡?



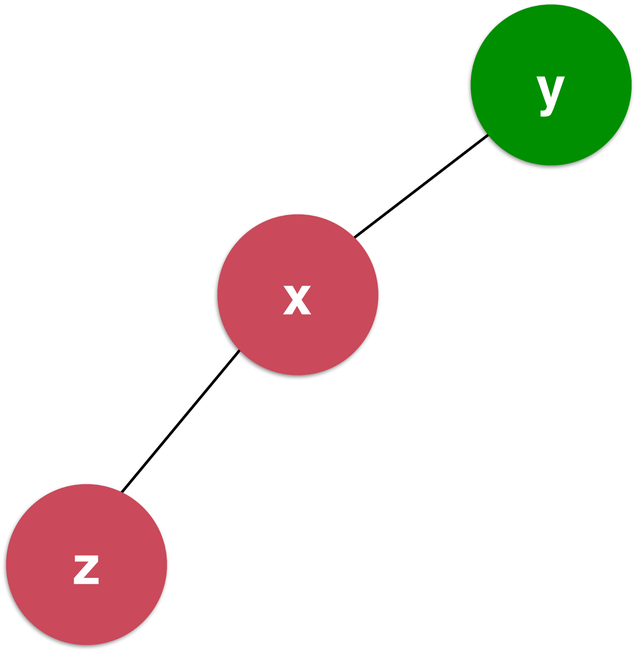
在什么时候维护平衡?





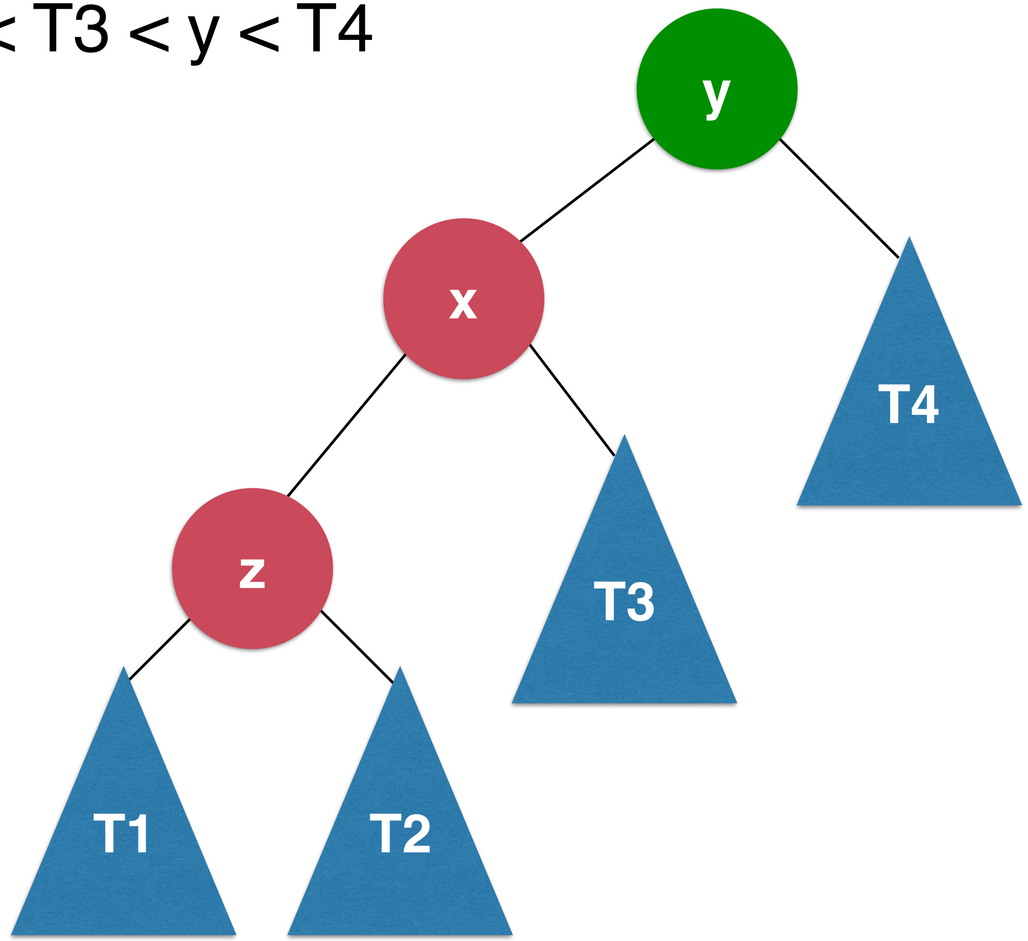
插入的元素在不平衡的节点的左侧的左侧

右旋转



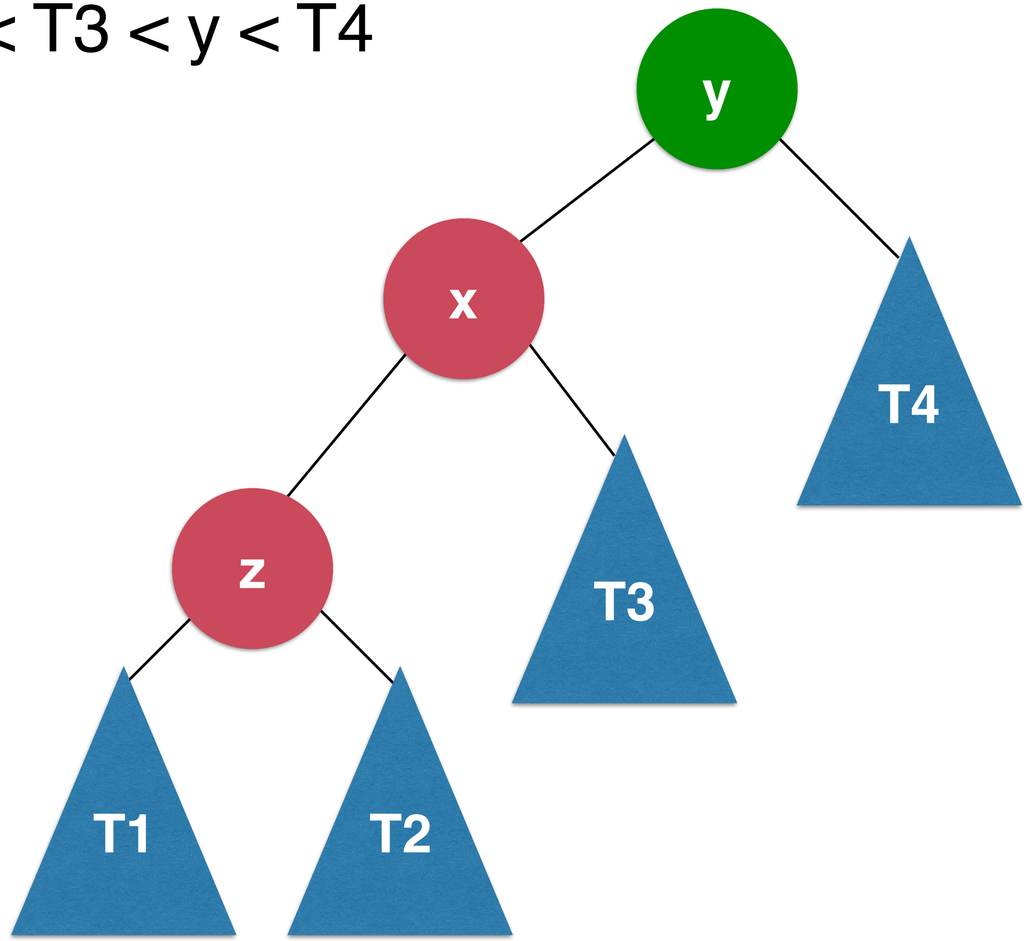
右旋转

$T1 < z < T2 < x < T3 < y < T4$



右旋转

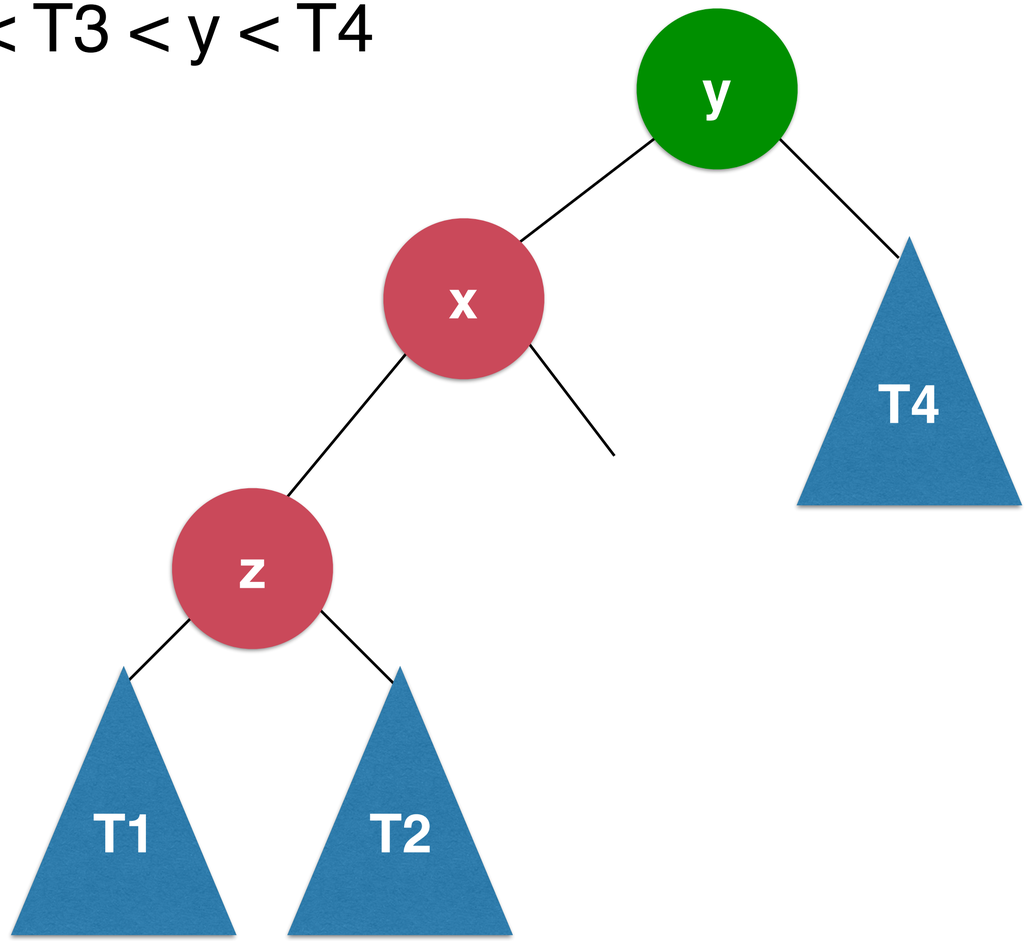
T1 < z < T2 < x < T3 < y < T4




```
x.right = y
```

右旋转

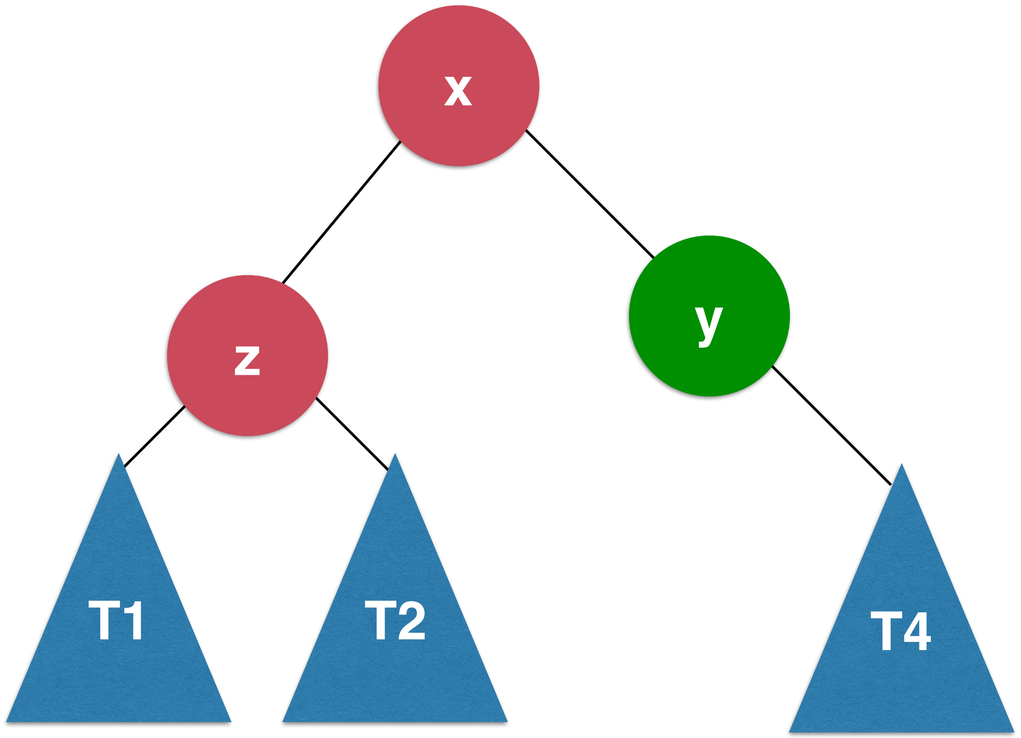
T1 < z < T2 < x < T3 < y < T4




```
x.right = y
```

右旋转

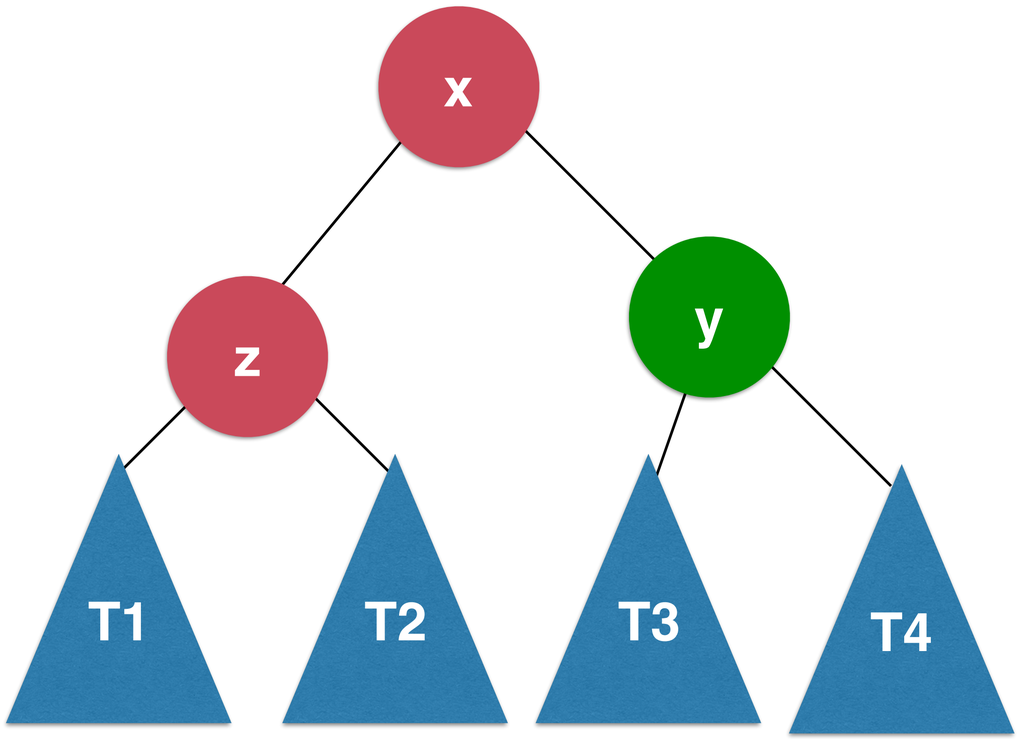
$T1 < z < T2 < x < T3 < y < T4$




```
x.right = y
y.left = T3
```

右旋转

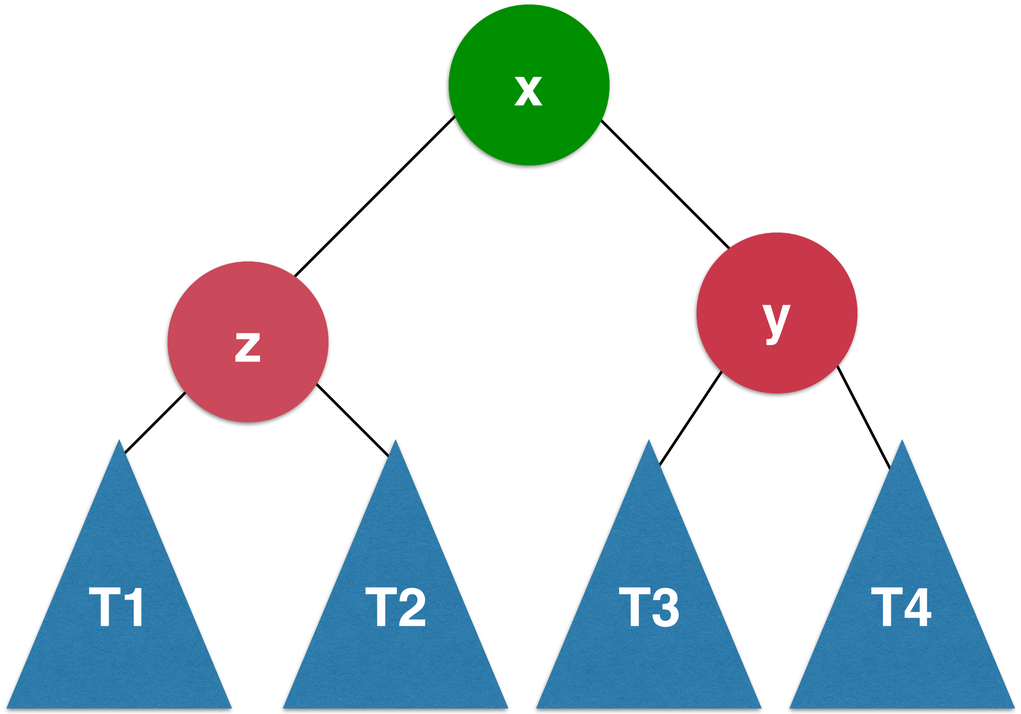
$T1 < z < T2 < x < T3 < y < T4$




```
x.right = y
y.left = T3
```

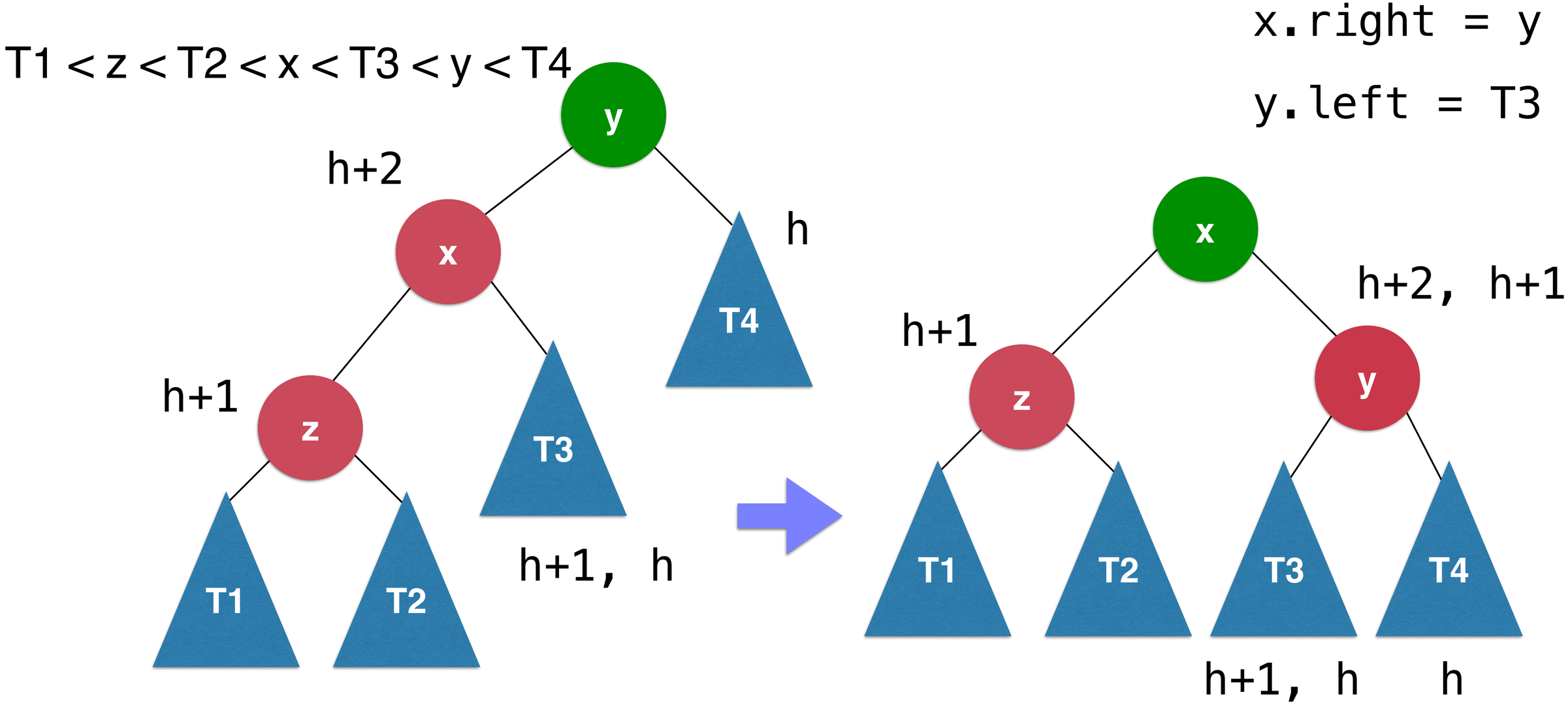
右旋转

T1 < z < T2 < x < T3 < y < T4




```
x.right = y
y.left = T3
```

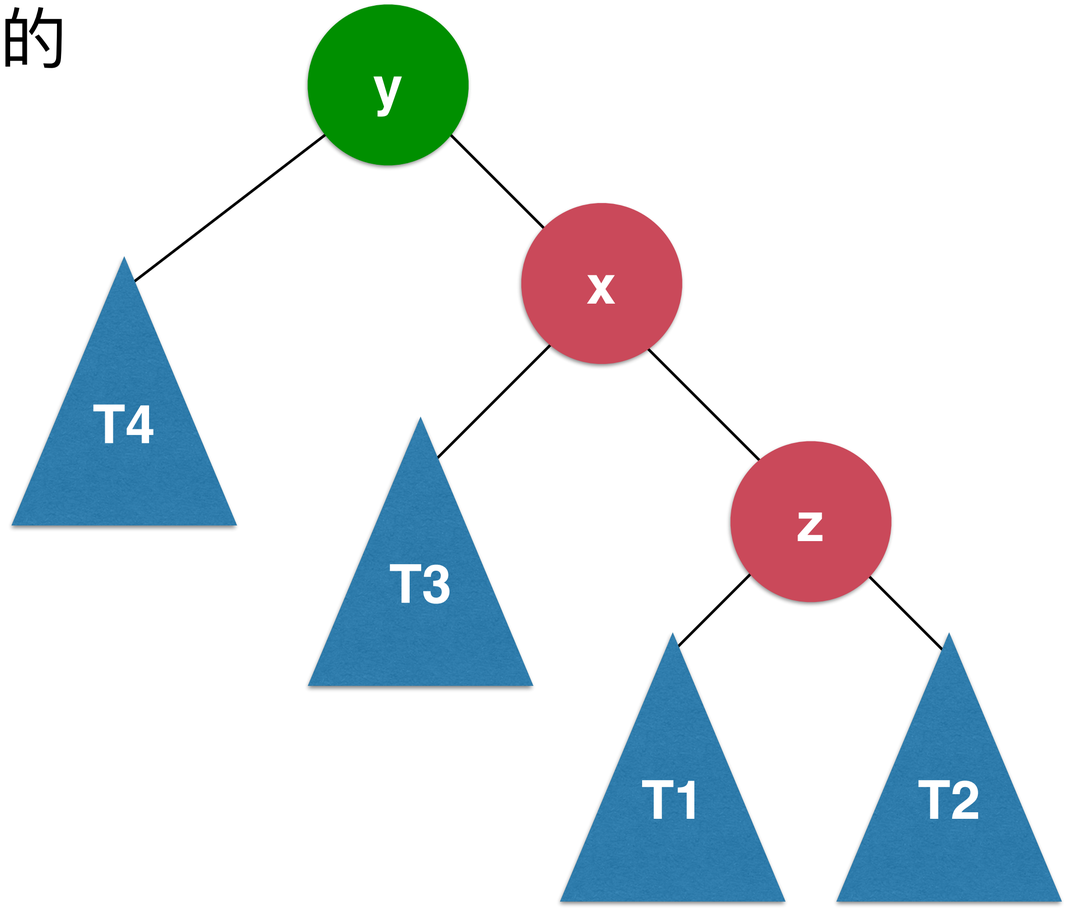
右旋转


T1 < z < T2 < x < T3 < y < T4
y
h+2
x
h
T4
h+1
z
T3
T1
T2
h+1, h
x.right = y
y.left = T3
x
h+2, h+1
h+1
z
y
T1
T2
T3
T4
h+1, h
h

实践：右旋转

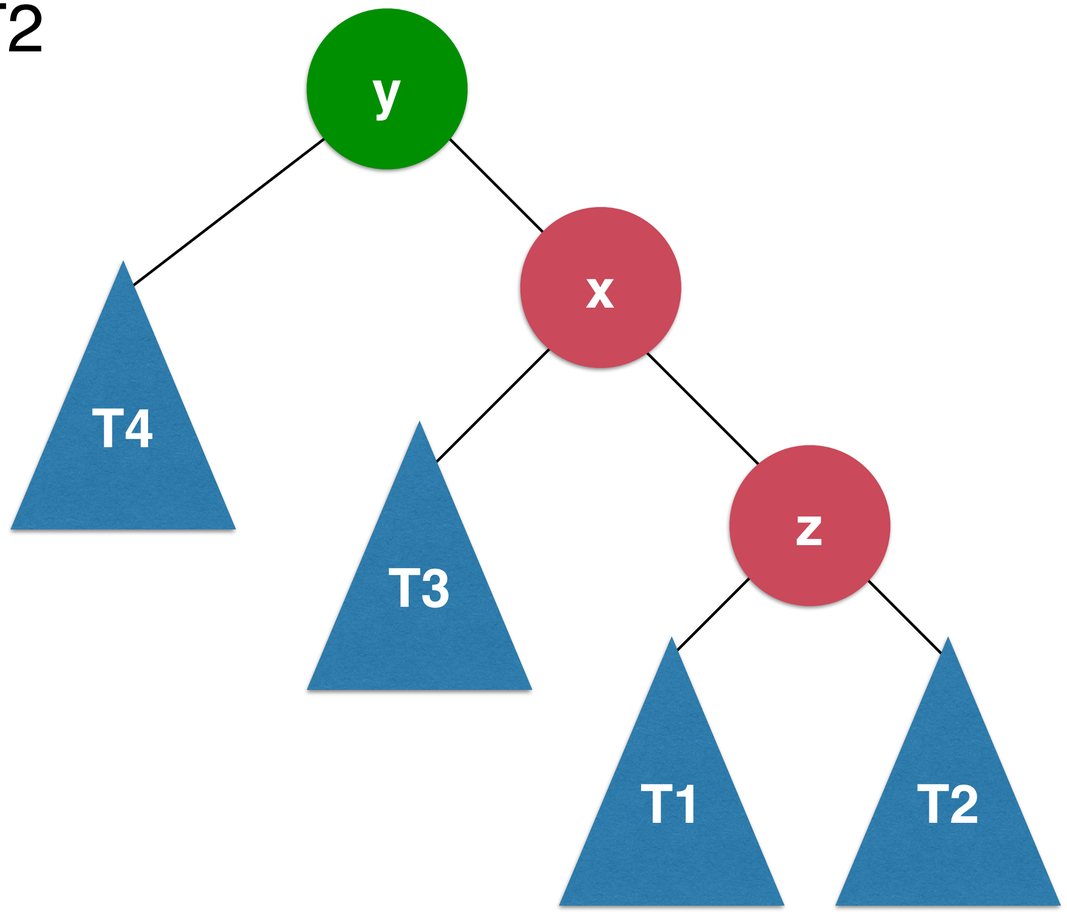
左旋转

插入的元素在不平衡的节点的
右侧的右侧



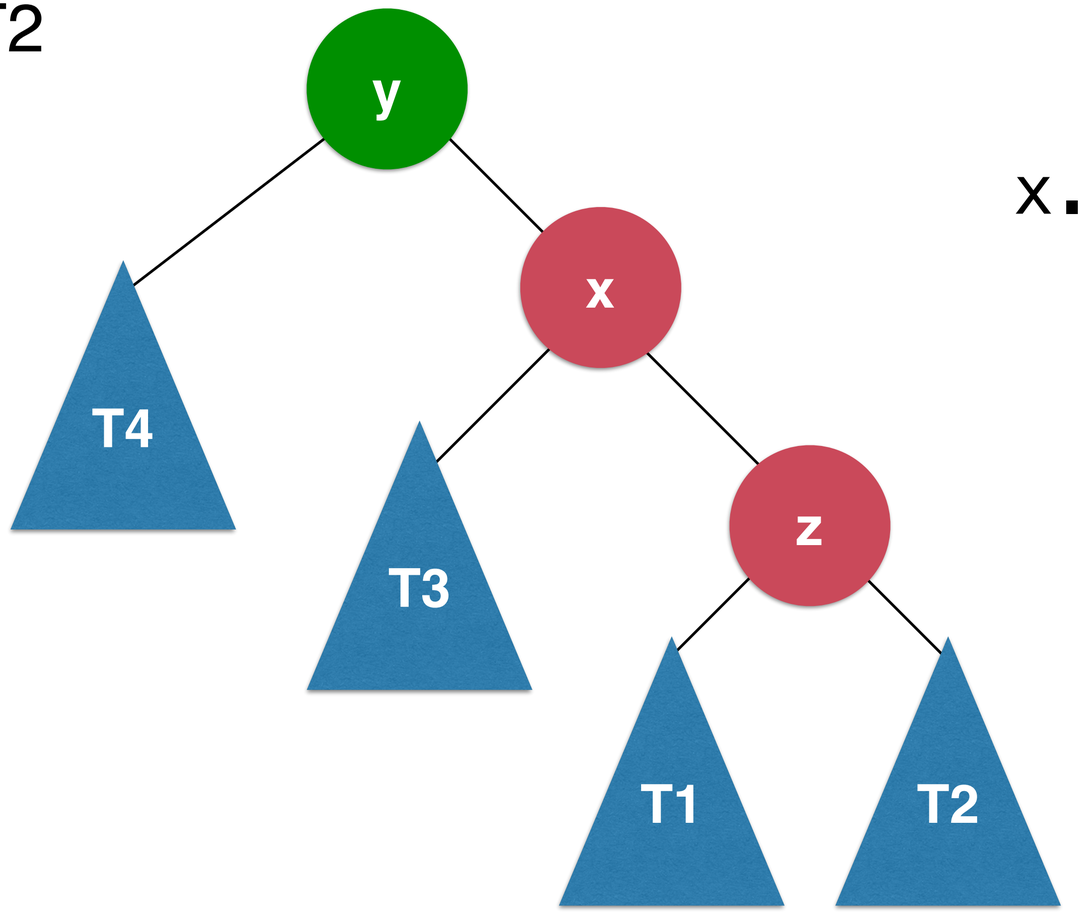
左旋转

T4 < y < T3 < x < T1 < z < T2



左旋转

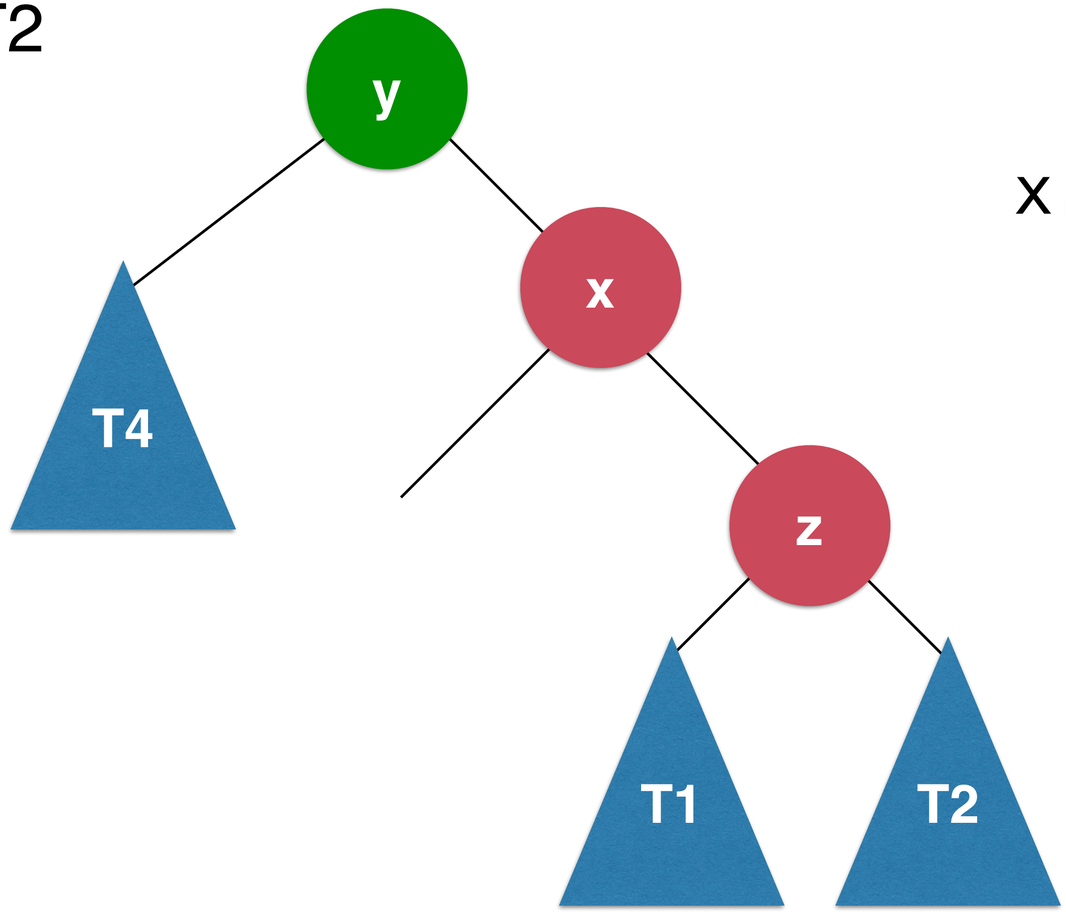
T4 < y < T3 < x < T1 < z < T2




```
x.left = y
```

左旋转

T4 < y < T3 < x < T1 < z < T2

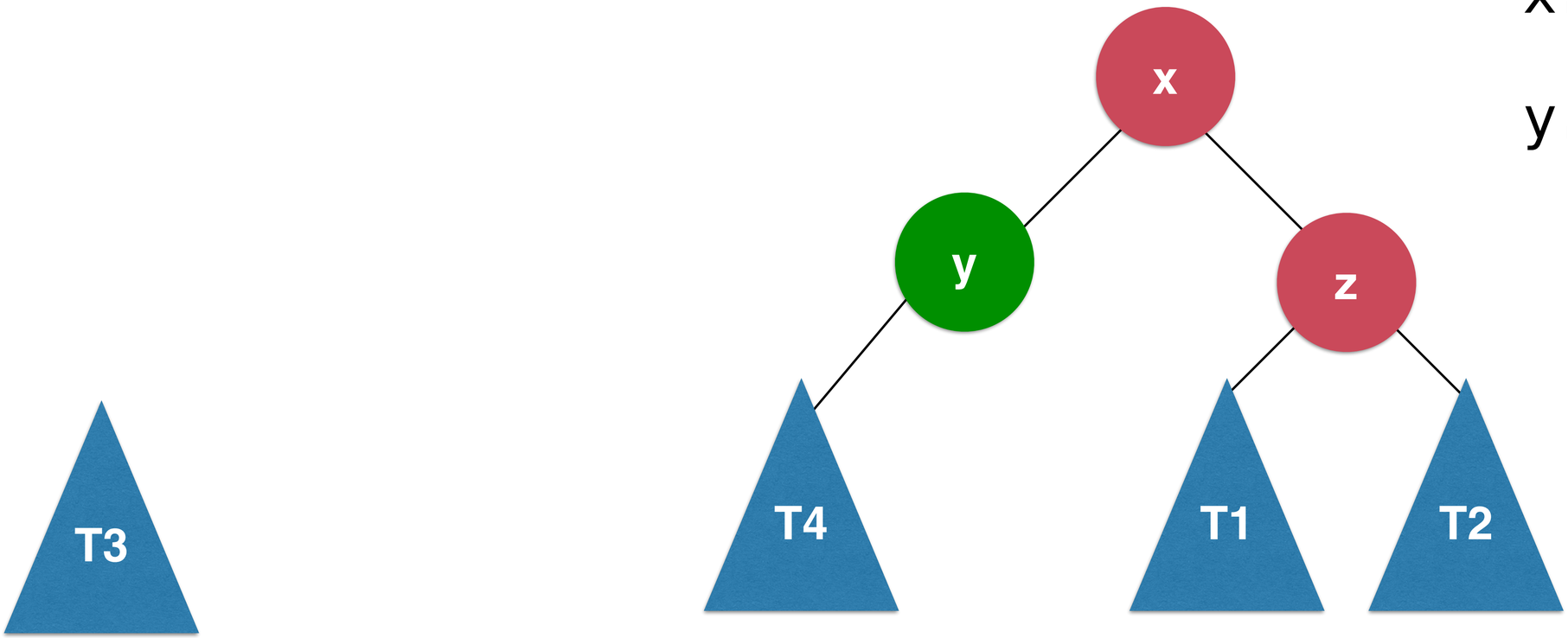



```
x.left = y
```

左旋转

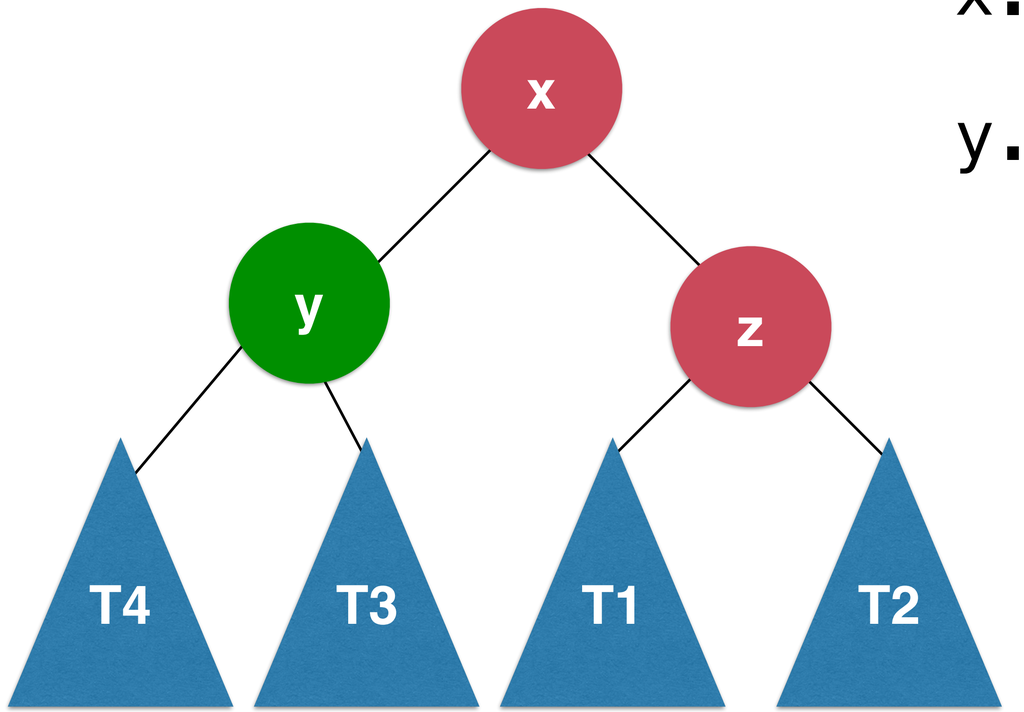
$T4 < y < T3 < x < T1 < z < T2$

```
x.left = y

y.right = T3
```



左旋转

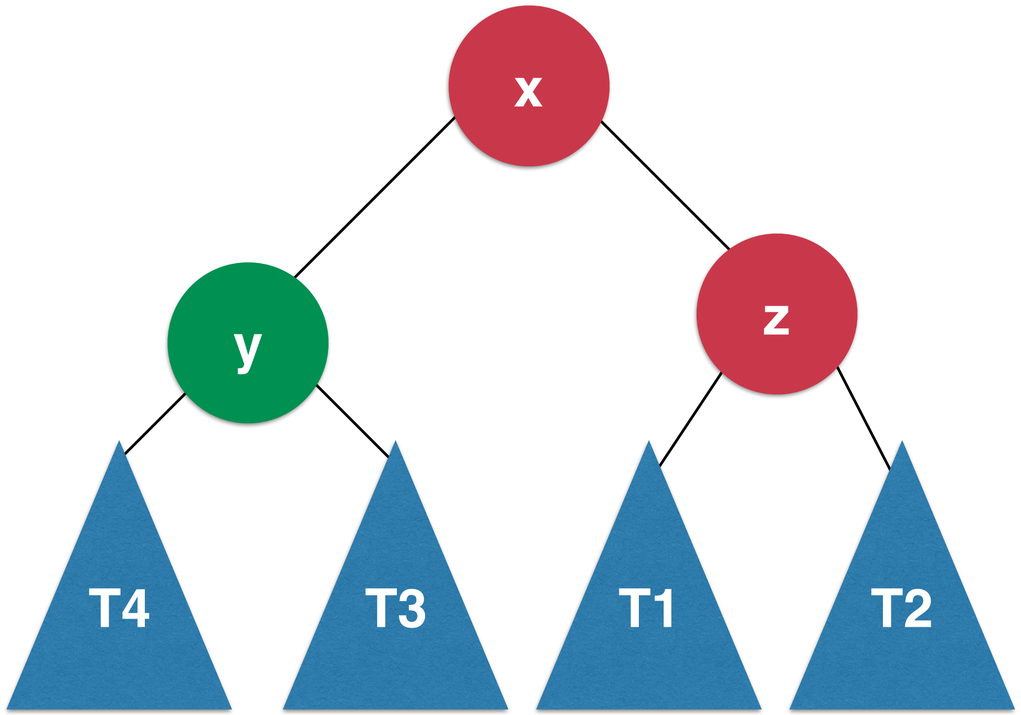
T4 < y < T3 < x < T1 < z < T2




```
x.left = y

y.right = T3
```

左旋转

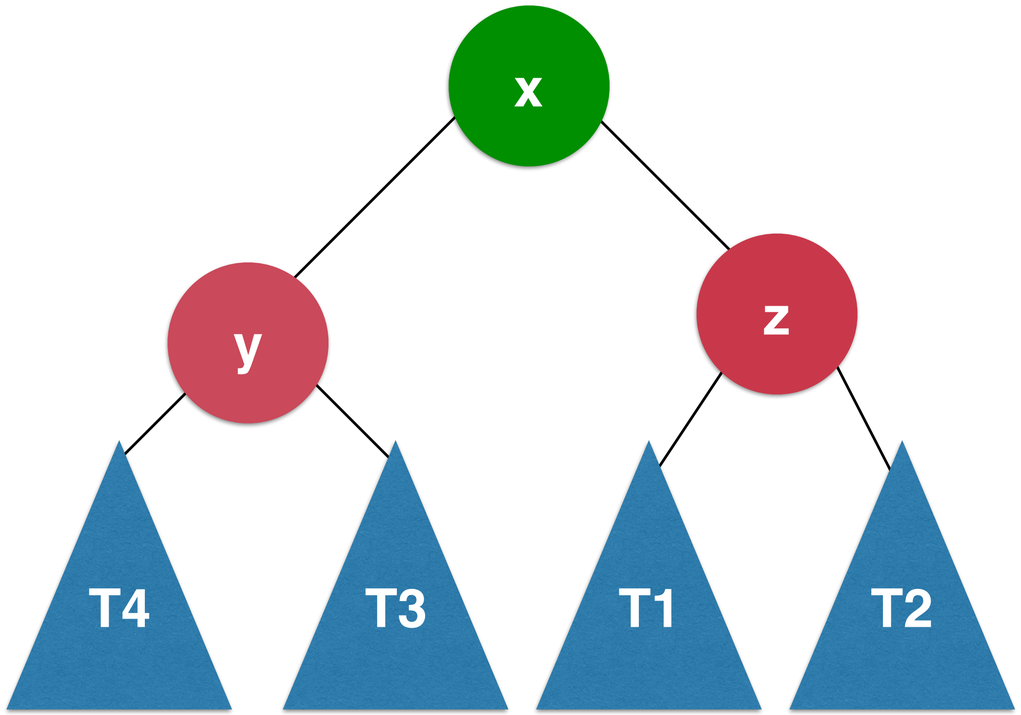
T4 < y < T3 < x < T1 < z < T2




```
x.left = y

y.right = T3
```

左旋转

T4 < y < T3 < x < T1 < z < T2



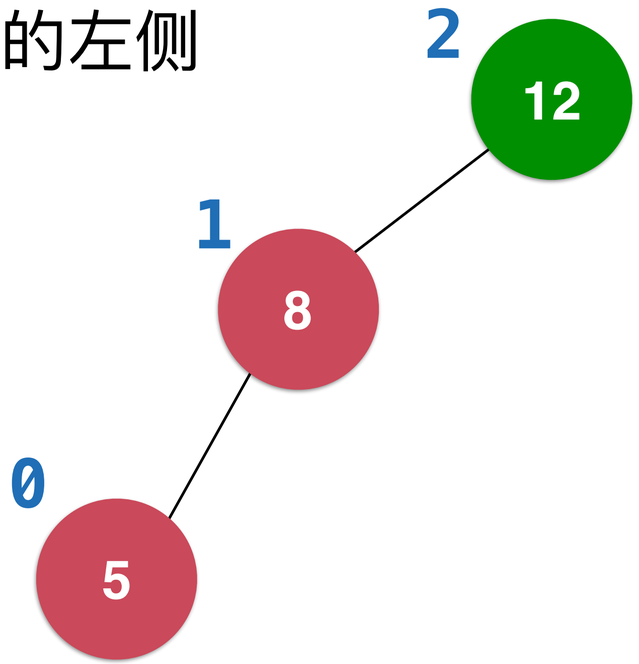

```
x.left = y

y.right = T3
```

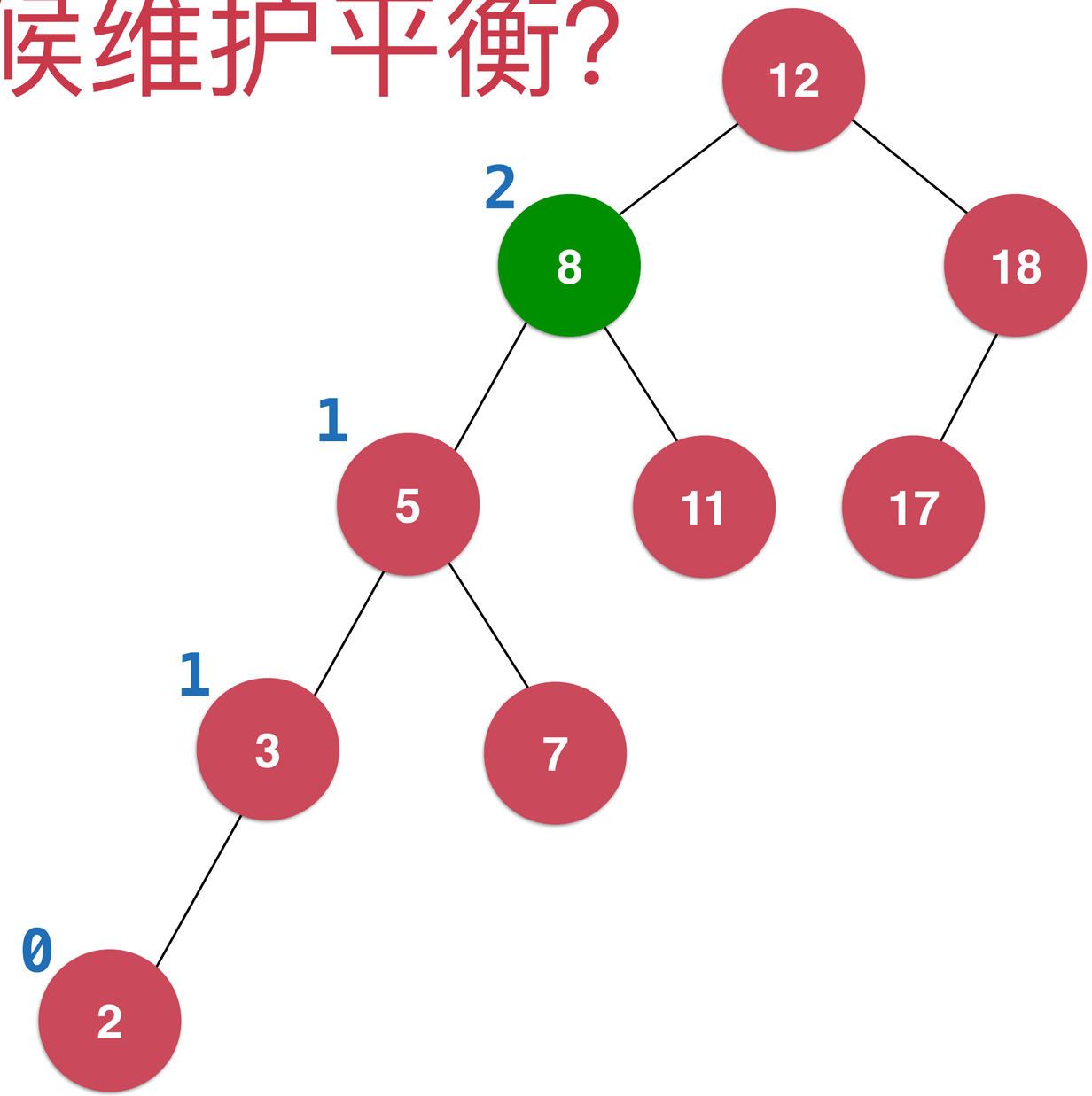
实践：左旋转

LR和RL

在什么时候维护平衡?

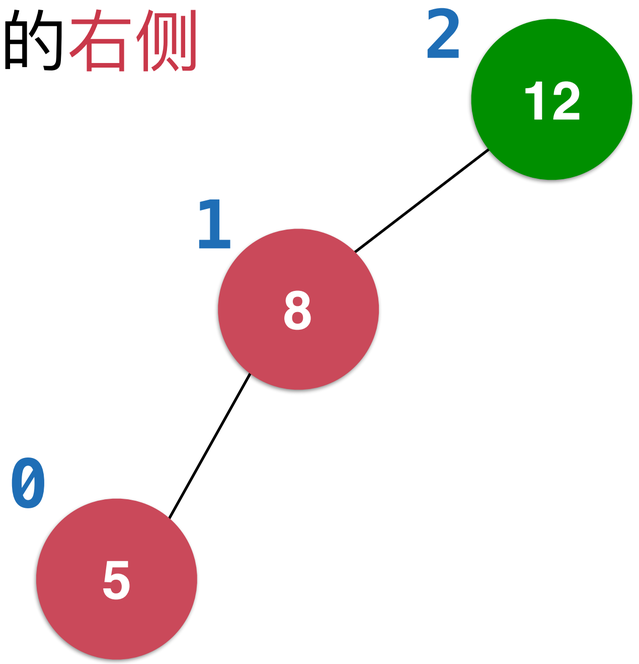
插入的元素在不平衡的
节点的左侧的左侧

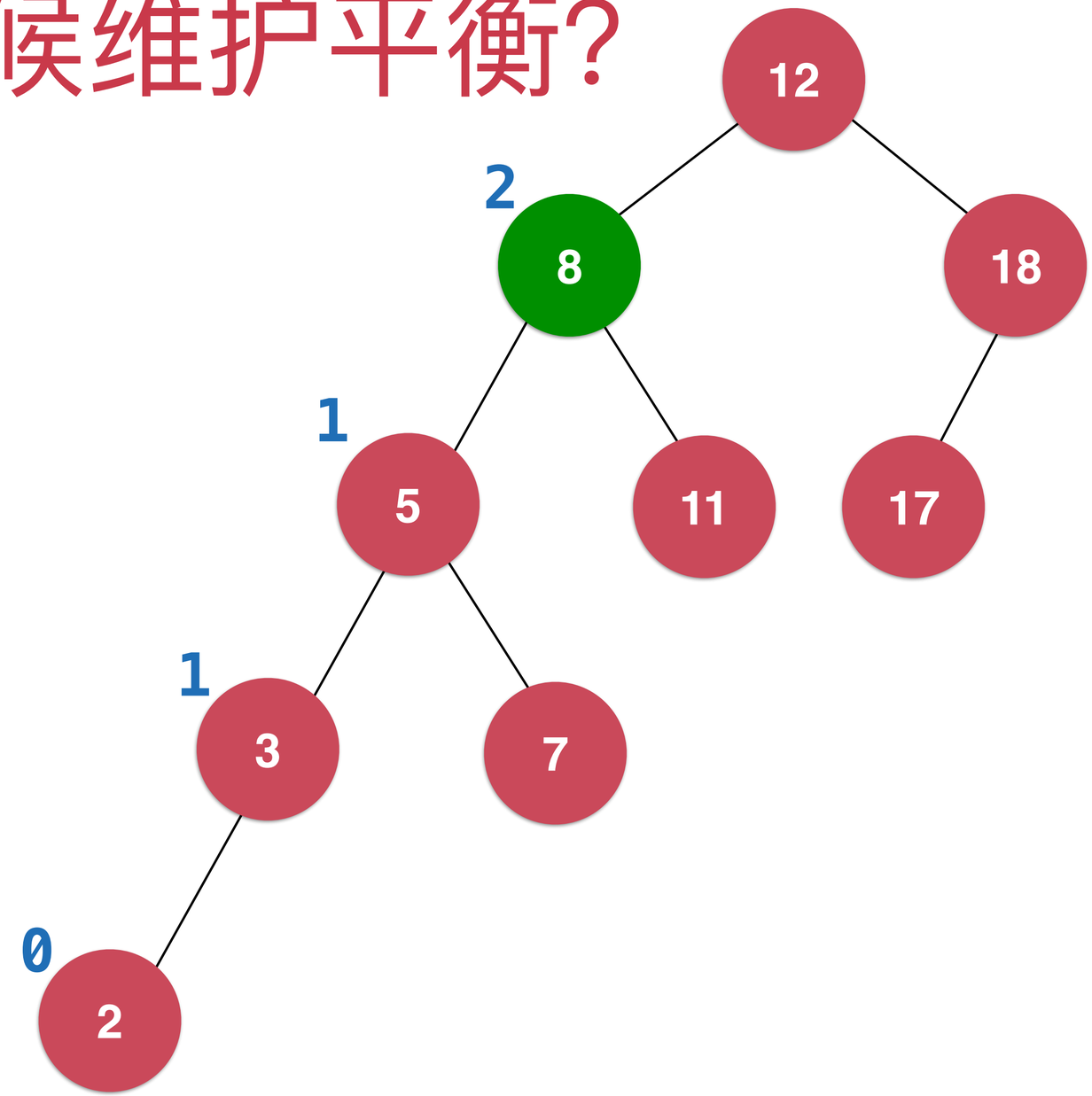




在什么时候维护平衡?

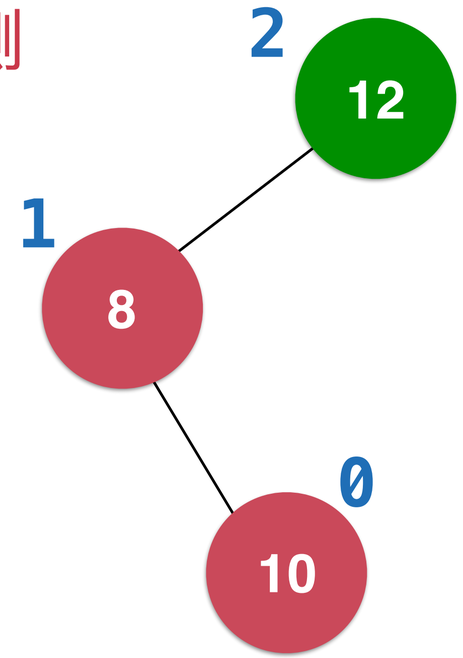
插入的元素在不平衡的
节点的左侧的右侧

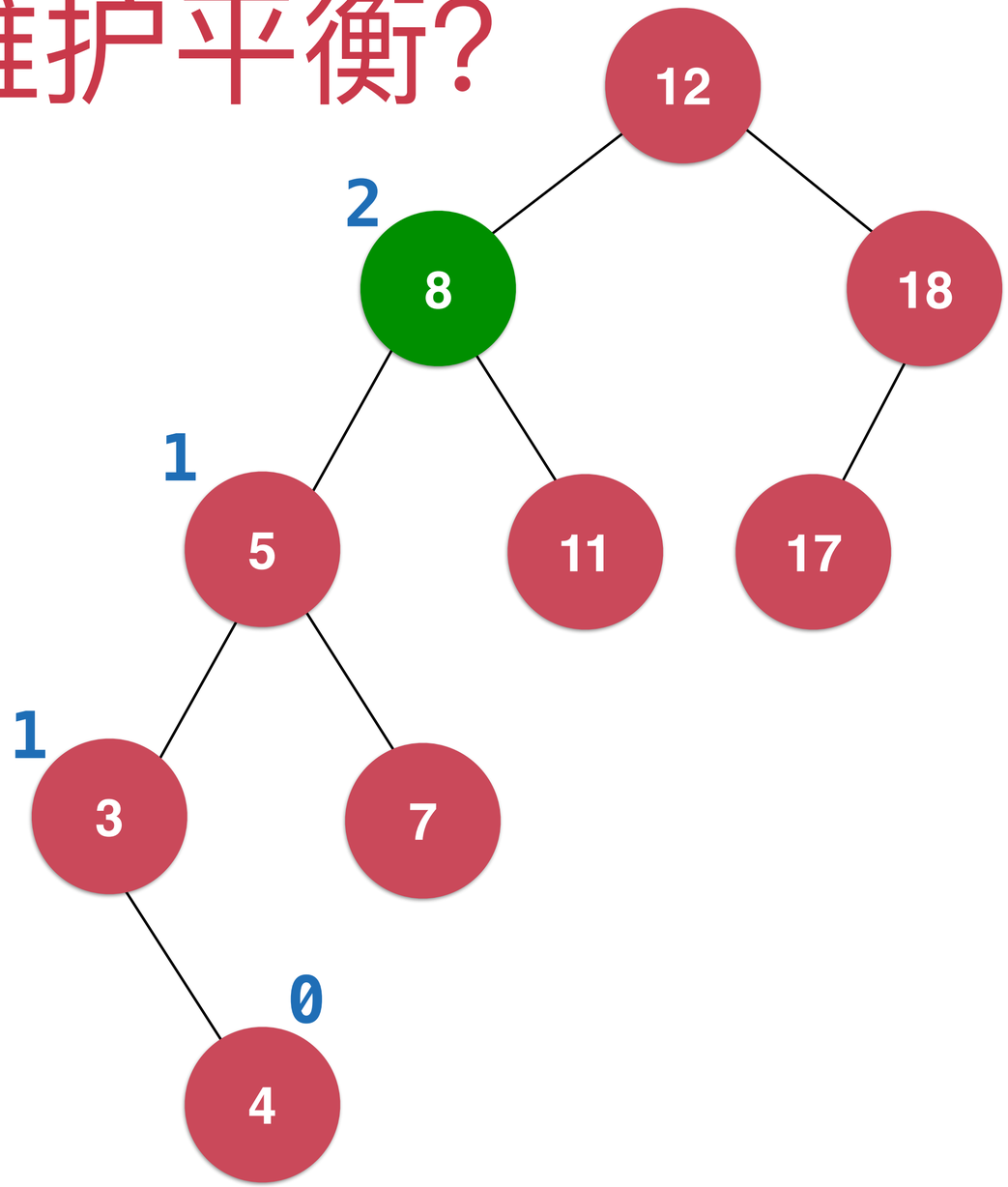




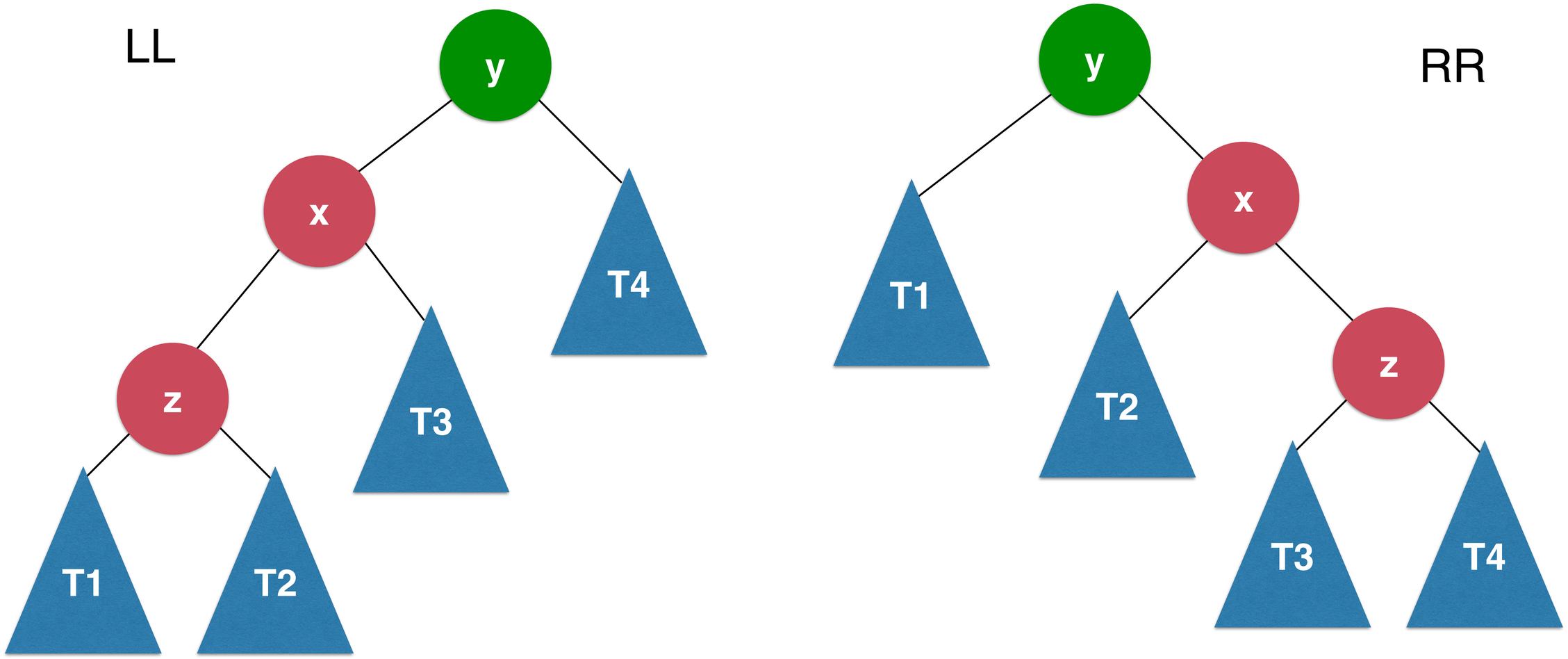
在什么时候维护平衡?

插入的元素在不平衡的
节点的左侧的右侧



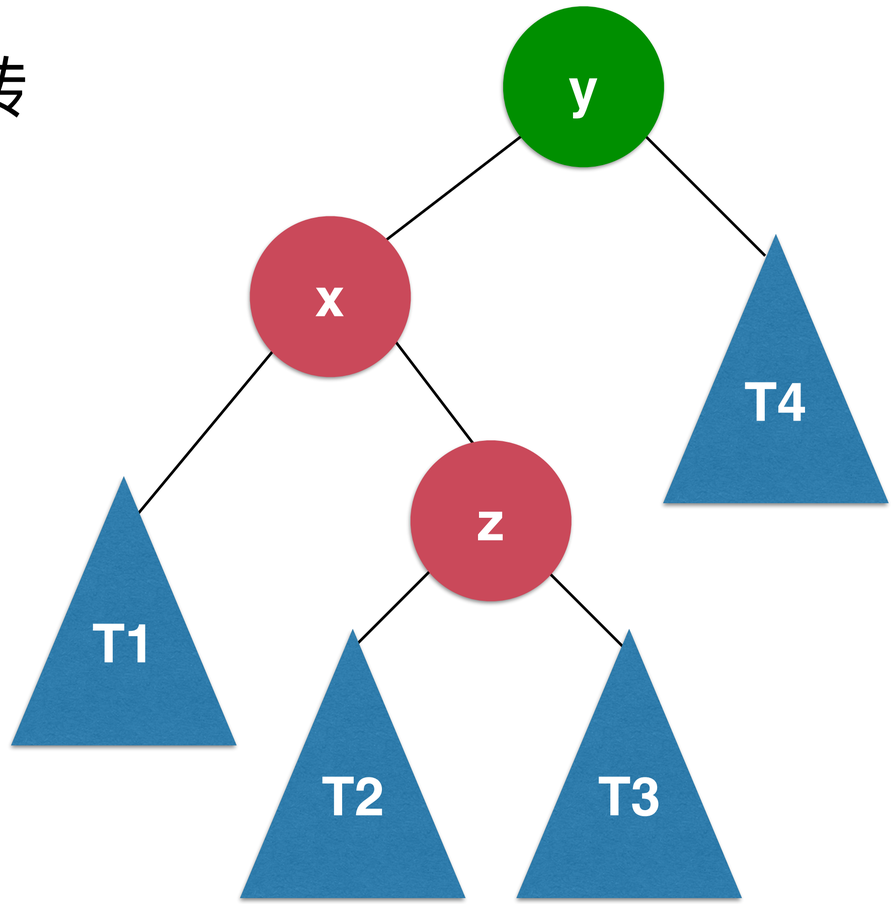


LL 和 RR



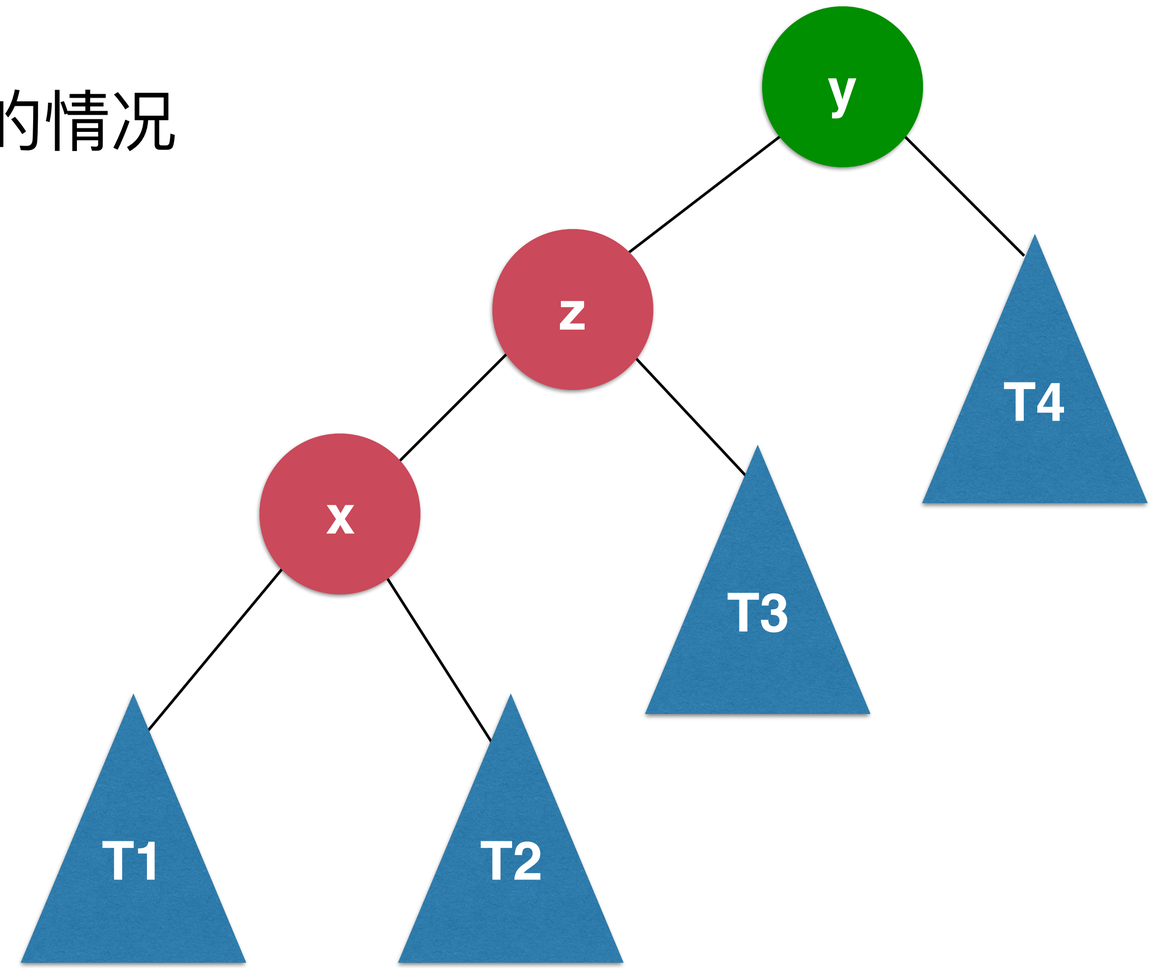
LR

首先对x进行左旋转



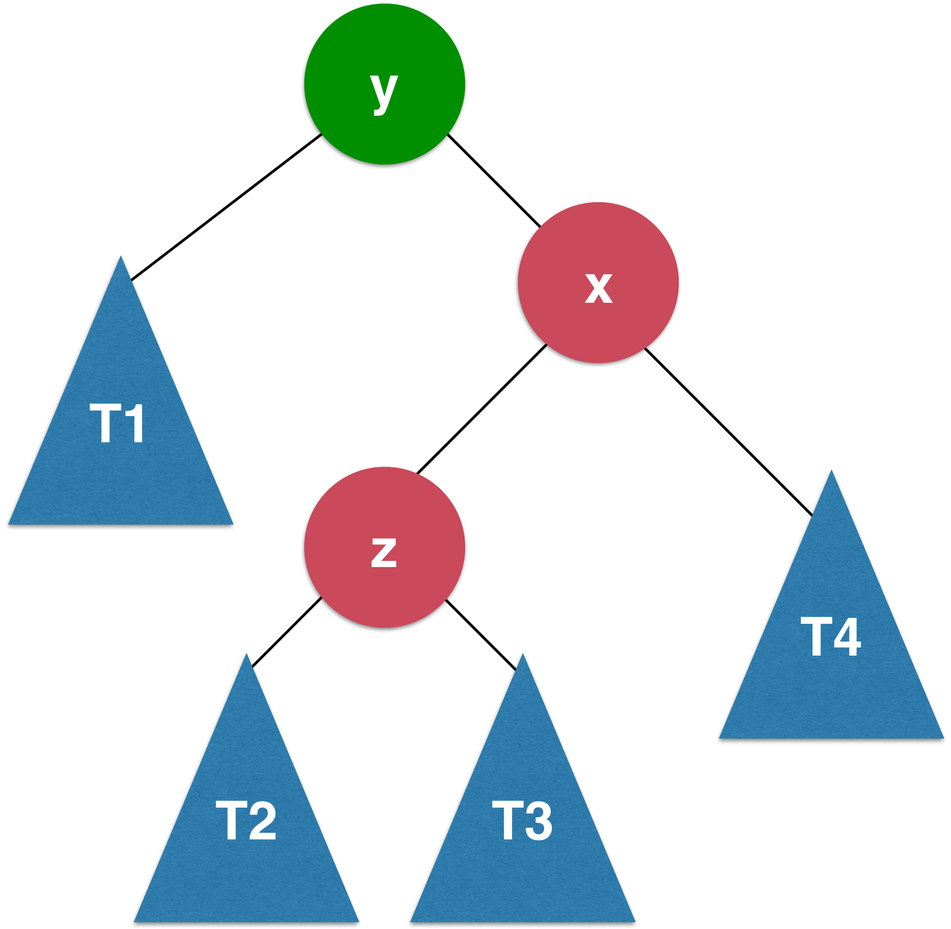
LR

转化为了LL的情况



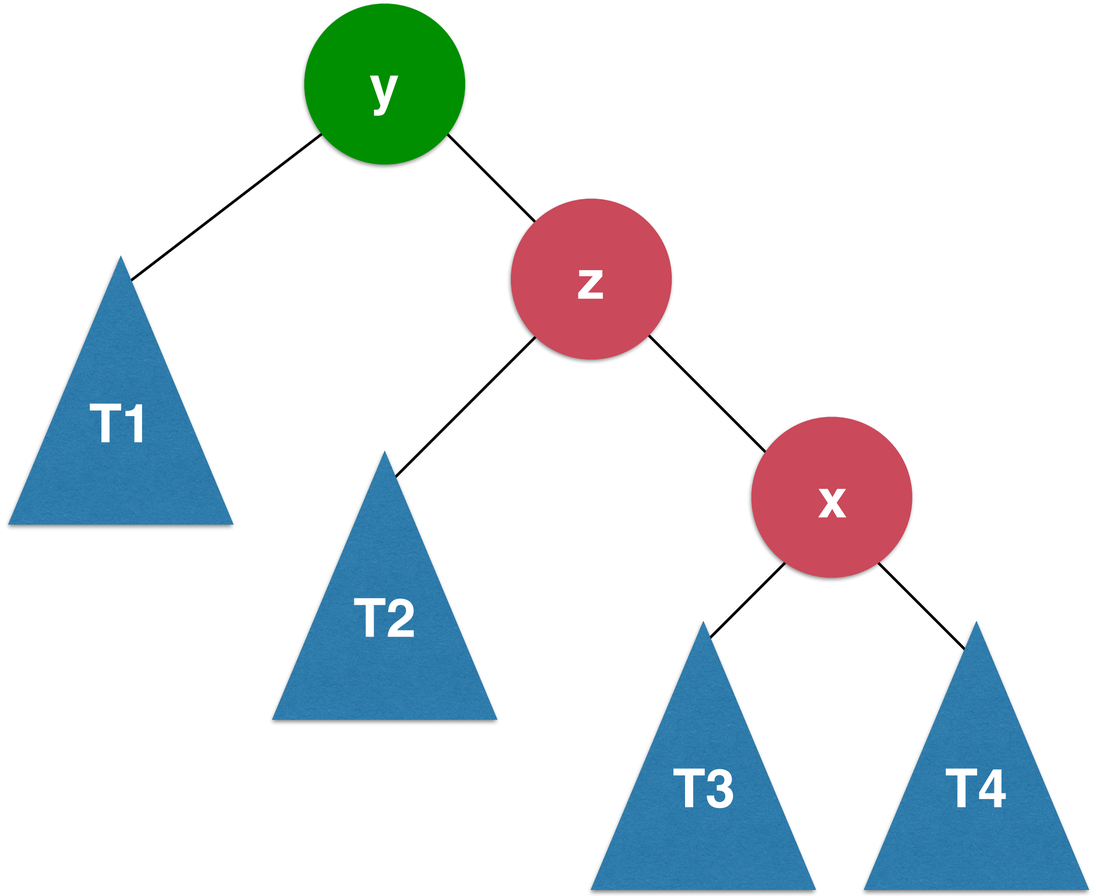
RL

首先对x进行右旋转



RL

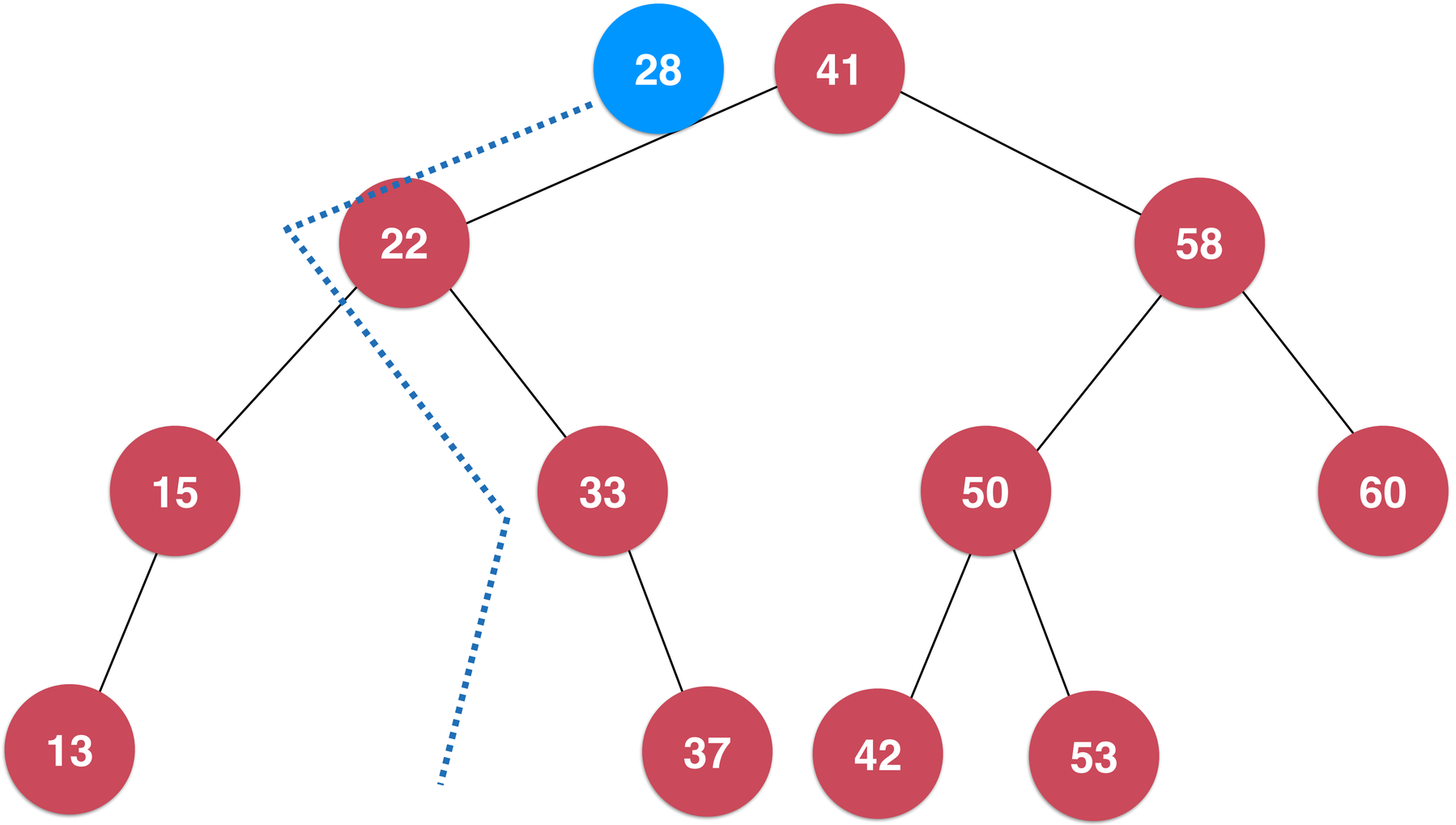
转化成了RR的情况



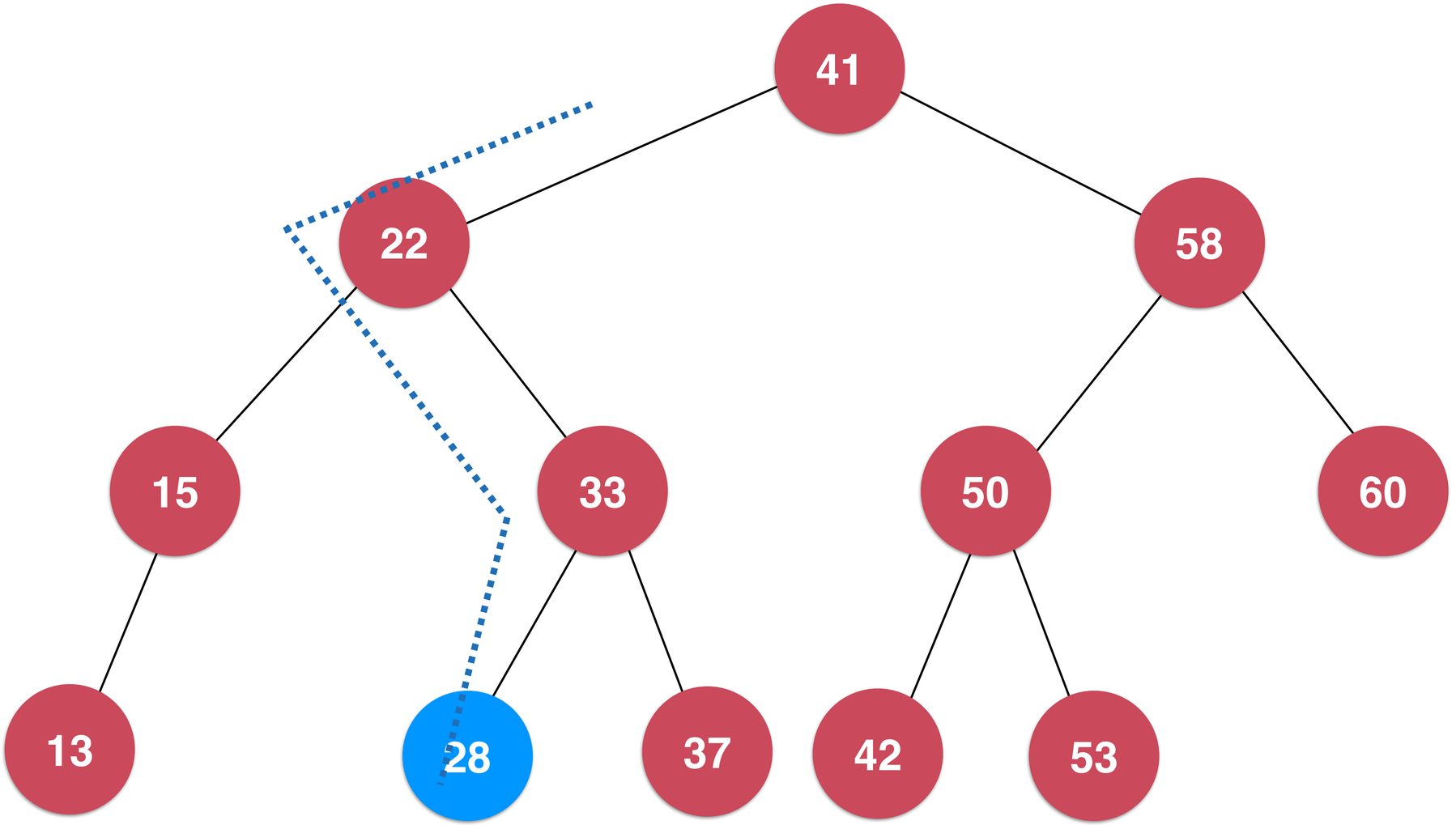
实践：处理LR和RL的情况

AVL树的删除

在什么时候维护平衡



在什么时候维护平衡



实践：AVL树的删除

更多AVL树的相关问题

基于AVL树的set和map

实践：基于AVL树的set和map

AVL树的优化

AVL树的局限性

AVL 树

其他

欢迎大家关注我的个人公众号：是不是很酷

玩儿转数据结构

liuyubobobo