海量资源：666java.com

二叉搜索树 & 红黑树

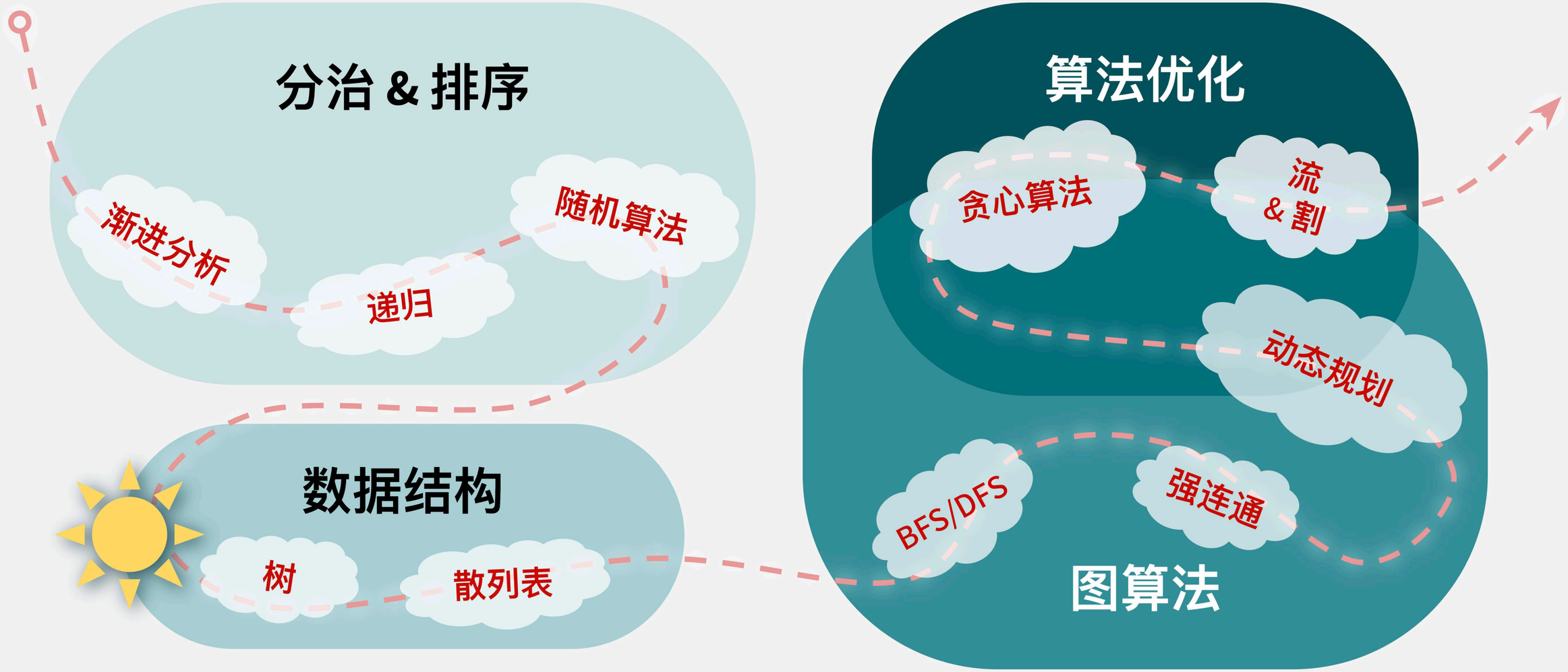
海量资源：666java.com

上次讲到

- 任何基于比较的排序的运行时间至少为 **Ω(n log n)**
- 采用非基于比较的计算模型可以在一定条件下实现线性时间排序
  - **计数排序**: 非常简单但不适用于输入可能值很多的情况
  - **基数排序**: 逐位进行计数排序, 当输入可能的最大值不是很大时可以在线性时间内完成

海量资源：666java.com

我们的征途



海量资源：666java.com

今天的内容

- 二叉搜索树
  - 支持哪些操作? 具有哪些性质?
- 平衡二叉树
  - 通过旋转实现平衡二叉树
  - 红黑树

海量资源：666java.com

二叉搜索树

什么是 BSTs, 如何操作二叉搜索树?

海量资源：666java.com

一维数据结构

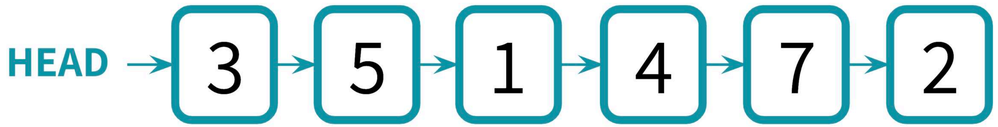
一维数据结构如何实现增删查?

排序数组

O(n) INSERT/DELETE: 找到插入/删除的位置(使用SEARCH), 移动一些数组内元素

O(log n) SEARCH: 使用二分查找寻找大小等于 A 的元素

链表



O(1) INSERT: 将新节点插入在表头处

O(n) SEARCH/DELETE: 由于链表无序, 需要先遍历链表再通过指针操作删除对应节点

海量资源：666java.com

二叉搜索树：意义

操作	排序数组	未排序链表
SEARCH	O(log(n))	O(n)
DELETE	O(n)	O(n)
INSERT	O(n)	O(1)

海量资源：666java.com

二叉搜索树：意义

操作	排序数组	未排序链表	BST (最坏情况)	BST (平衡)
SEARCH	O(log(n))	O(n)	O(n)	O(log(n))
DELETE	O(n)	O(n)	O(n)	O(log(n))
INSERT	O(n)	O(1)	O(n)	O(log(n))

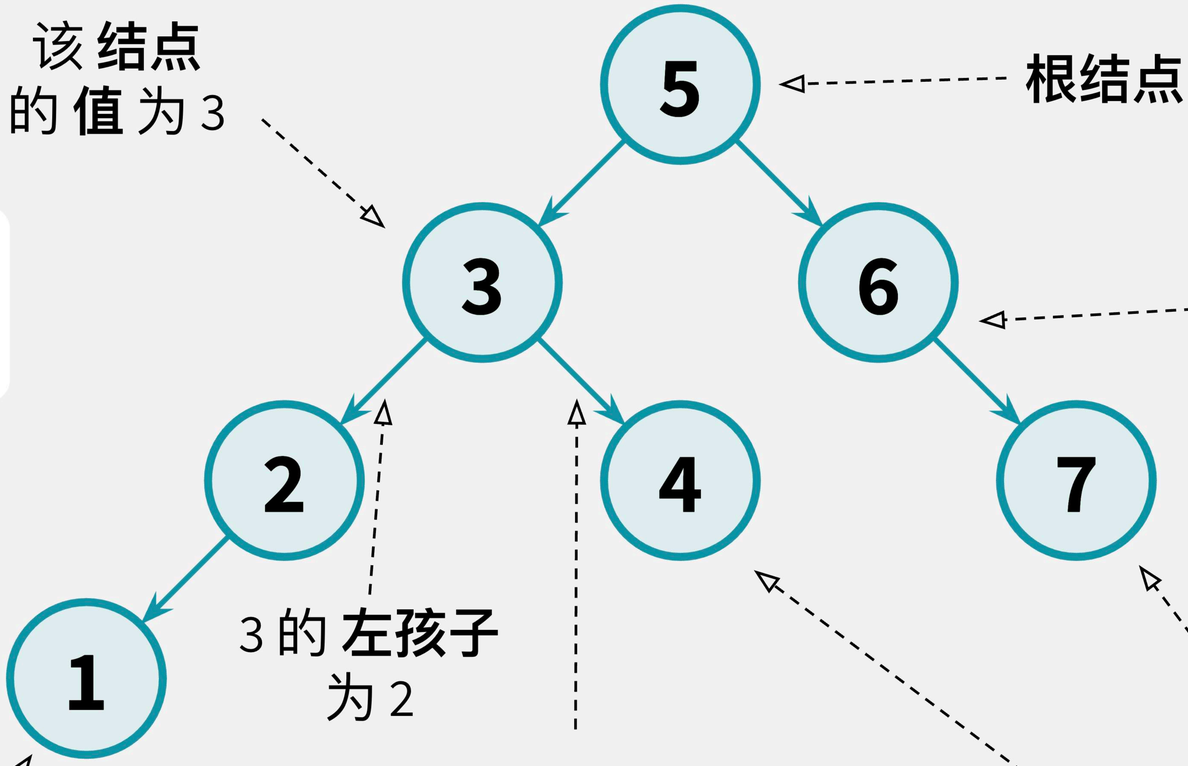
海量资源：666java.com

二叉树术语

每个结点有两个孩子

每个结点含有一个指针指向其左孩子和有孩子，有时也保存其父亲

结点 1 的祖先为 1, 2, 3, 5



该 **结点** 的 **值** 为 3

根结点

6 是 7 的 **父亲结点**

该二叉树的**高度**为 3

3 的 **左孩子** 为 2

3 的 **右孩子** 为 4

叶结点的左右孩子均为 **NIL**
(一般不画出来)

不含子节点的为 **叶结点**

海量资源：666java.com

二叉搜索树的性质

设 x 是二叉搜索树中的一个结点

如果 y 是 x 左子树中的一个结点，那么 $y.key \le x.key$

如果 y 是 x 右子树中的一个结点，那么 $y.key \ge x.key$

海量资源：666java.com

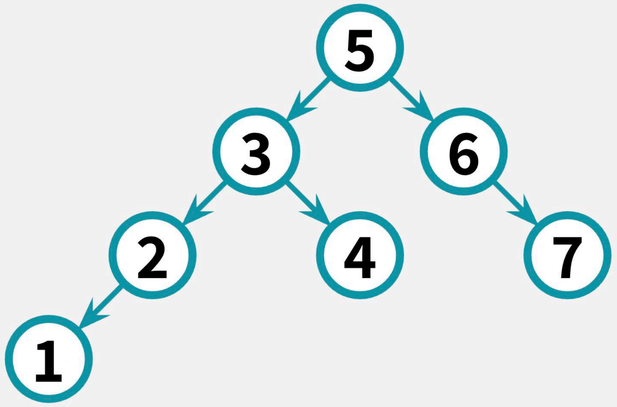
二叉搜索树的性质

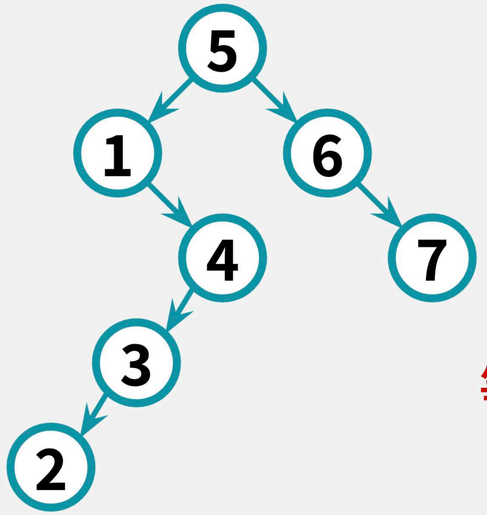
设 x 是二叉搜索树中的一个结点

如果 y 是 x 左子树中的一个结点，那么 y.key ≤ x.key

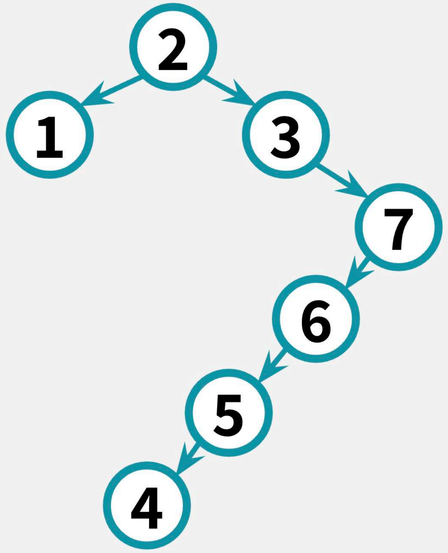
如果 y 是 x 右子树中的一个结点，那么 y.key ≥ x.key

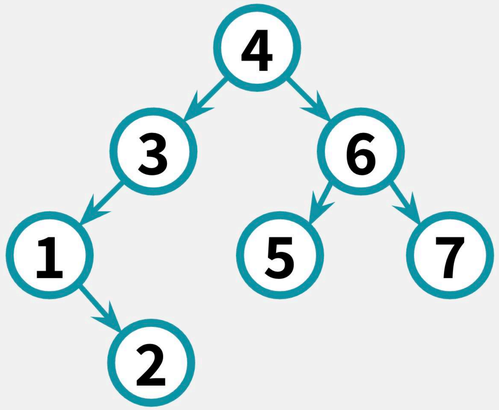
由右侧结点所构成的二叉搜索树:





等等...

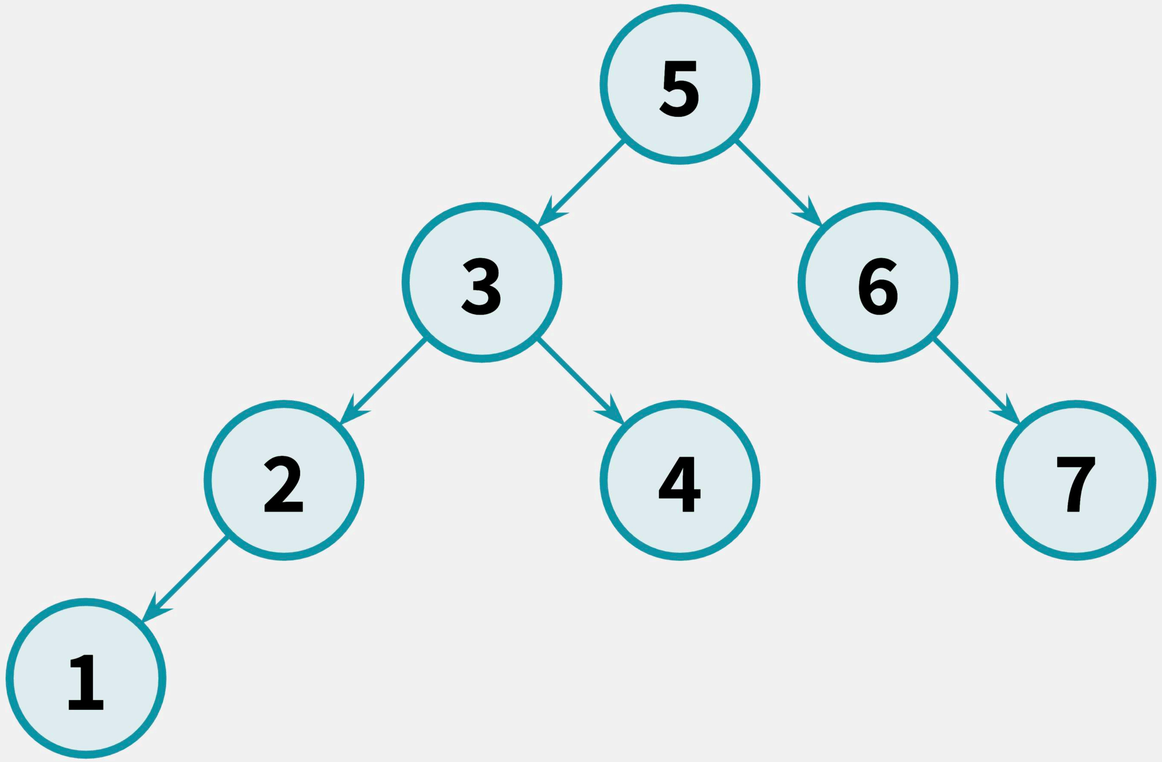




海量资源：666java.com

BST: SEARCH

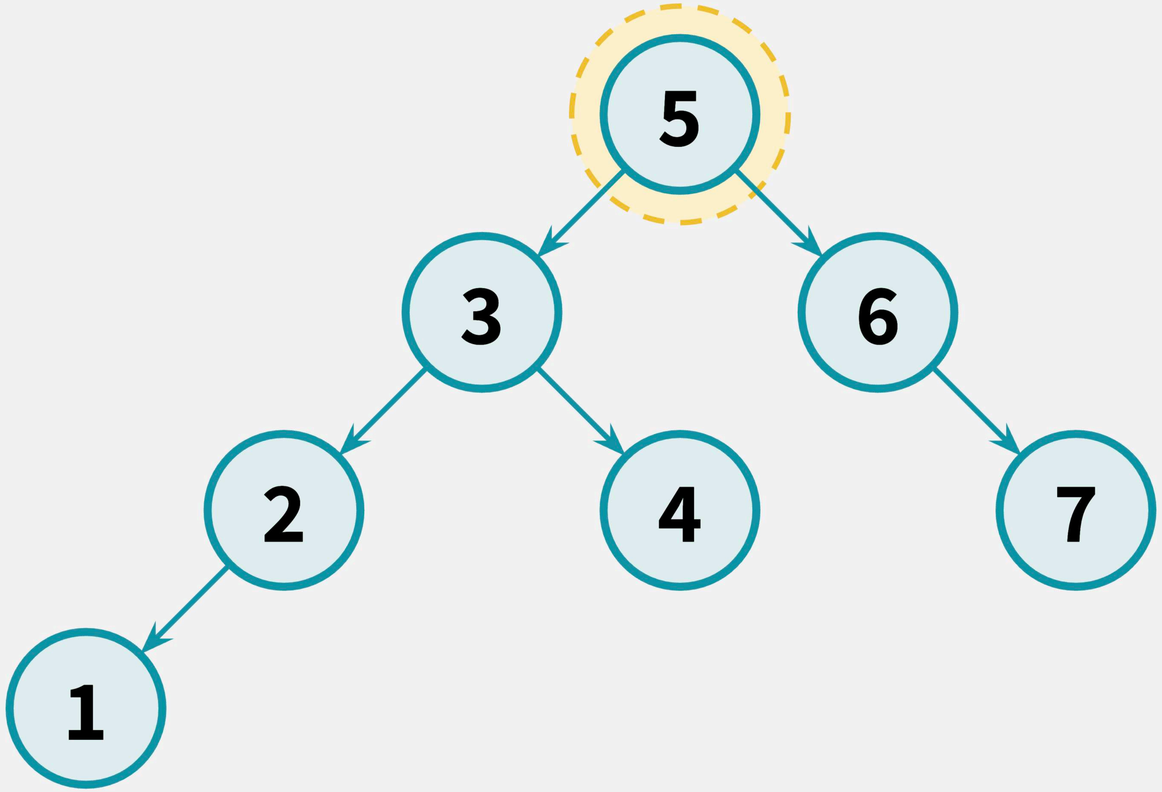
查找 4



海量资源：666java.com

BST: SEARCH

查找 4

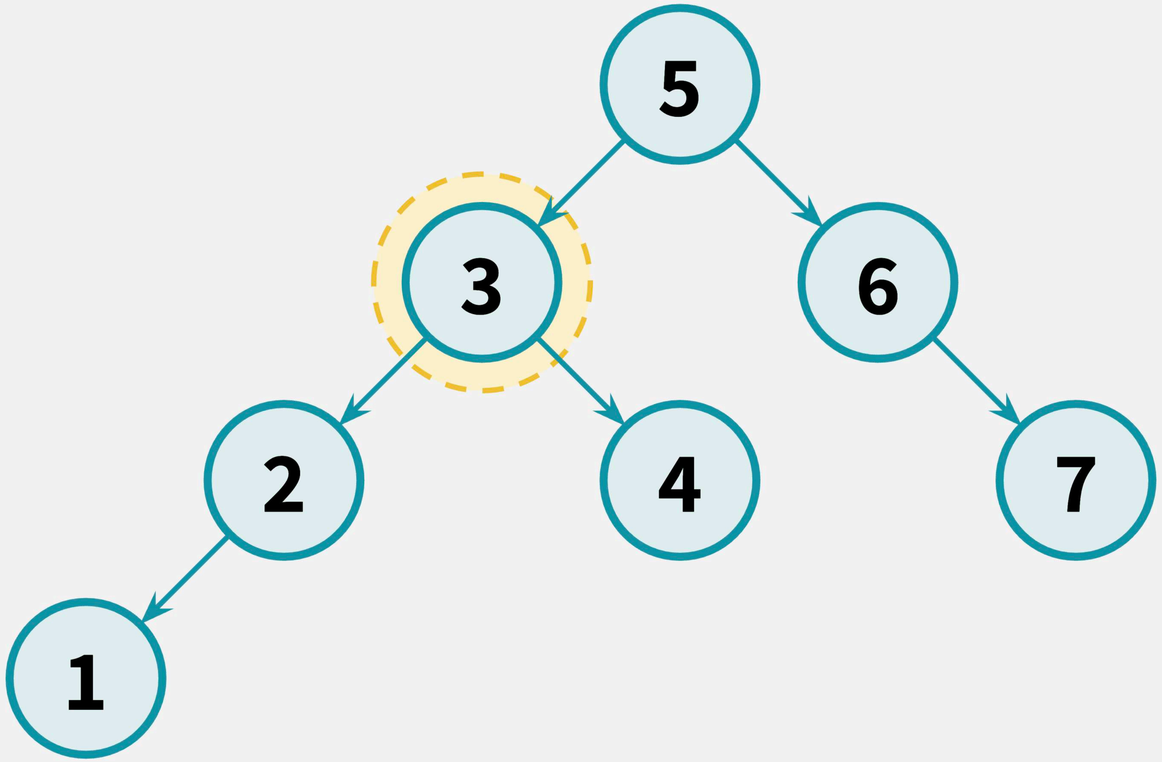


比较 **4** 和 **root**:
4 更小→ 向左

海量资源：666java.com

BST: SEARCH

查找 4



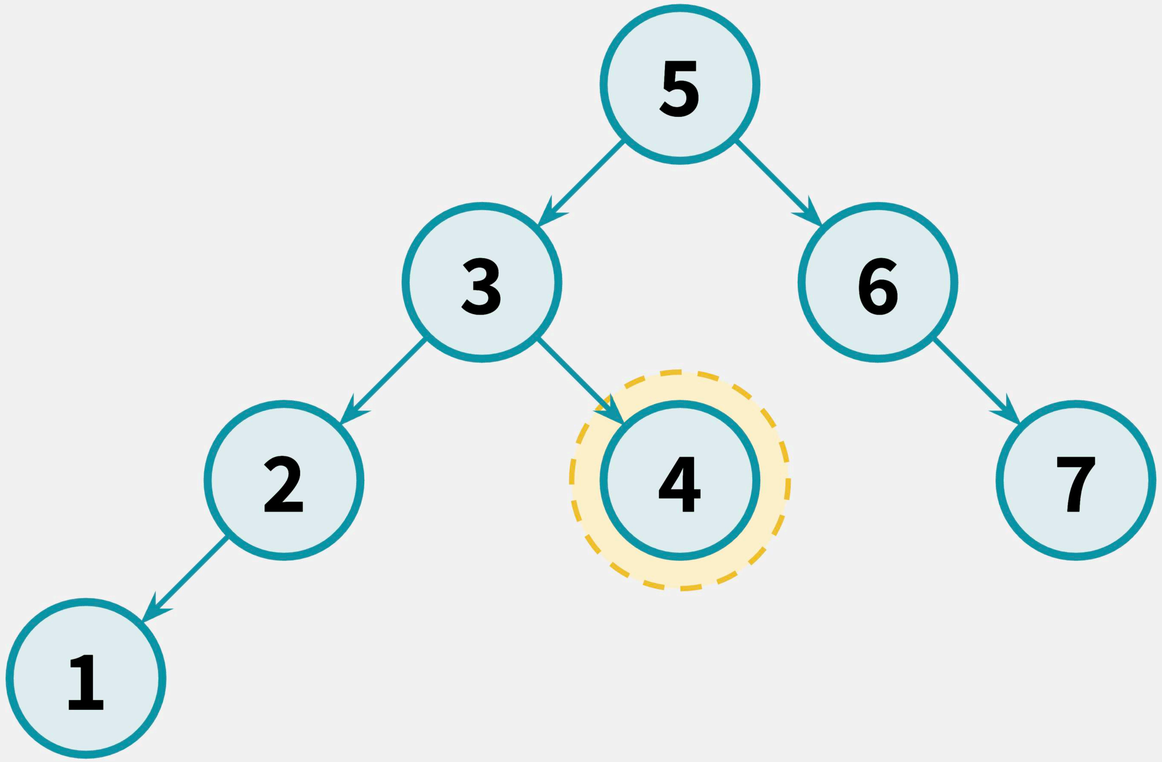
比较 **4** 和 **root**:
4 更小→ 向左

比较 **4** 和 **3**:
4 更大→ 向右

海量资源：666java.com

BST: SEARCH

查找 4



比较 **4** 和 **root**:
4 更小→ 向左

比较 **4** 和 **3**:
4 更大→ 向右

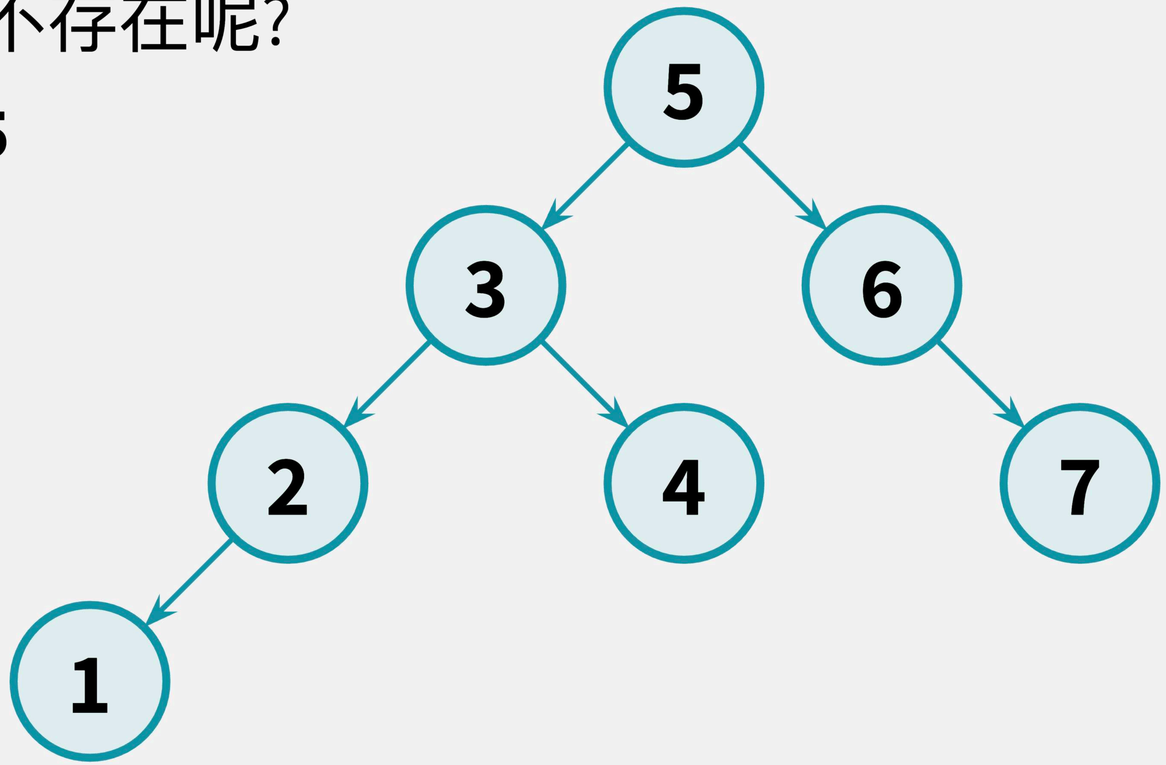
比较 **4** 和 **4**:
4 = 4 → 查找完成

海量资源：666java.com

BST: SEARCH

如果要查找的值不存在呢?

查找 4.5

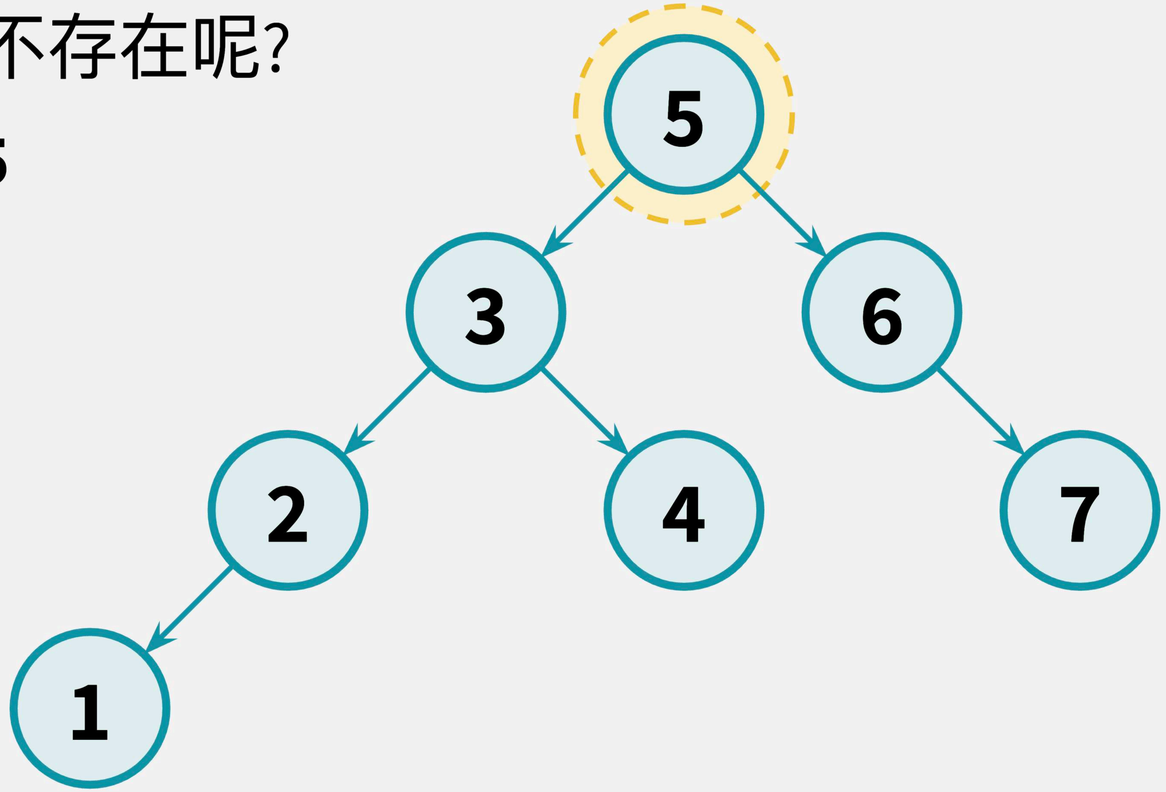


海量资源：666java.com

BST: SEARCH

如果要查找的值不存在呢?

查找 4.5



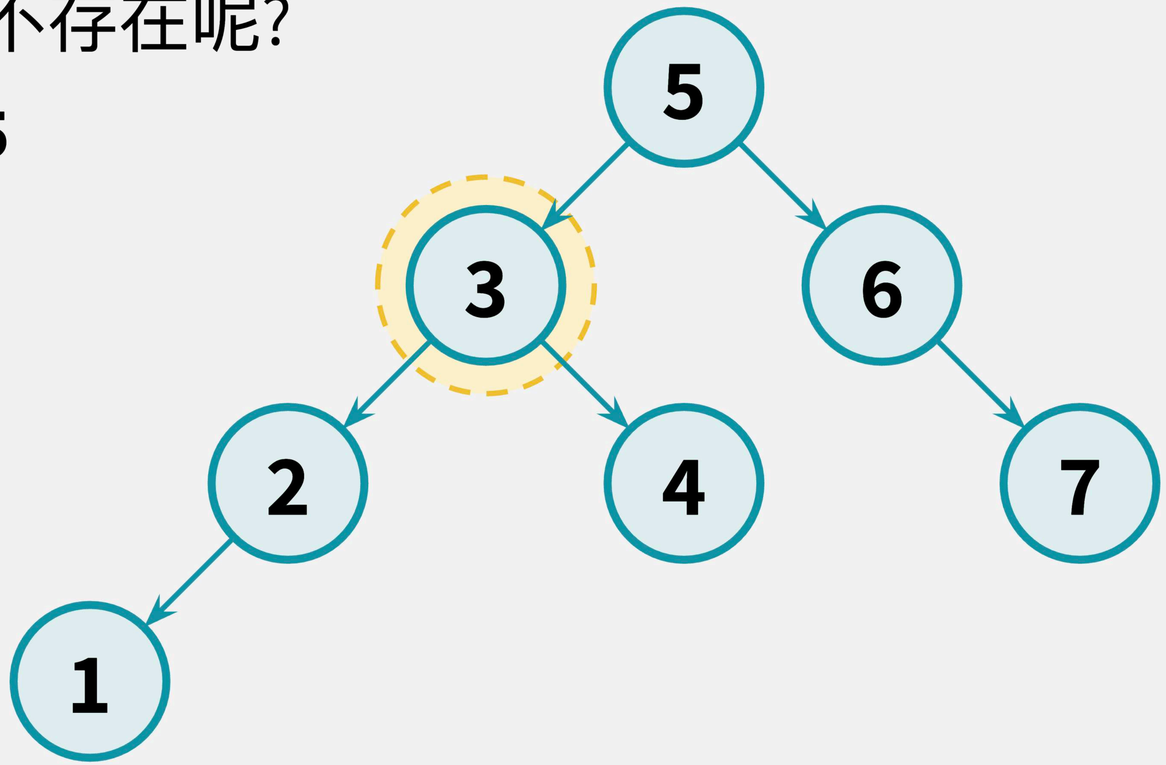
比较 **4.5** 和 **root**:
4.5 更小 → 向左

海量资源：666java.com

BST: SEARCH

如果要查找的值不存在呢?

查找 4.5



比较 **4.5** 和 **root**:
4.5 更小 → 向左

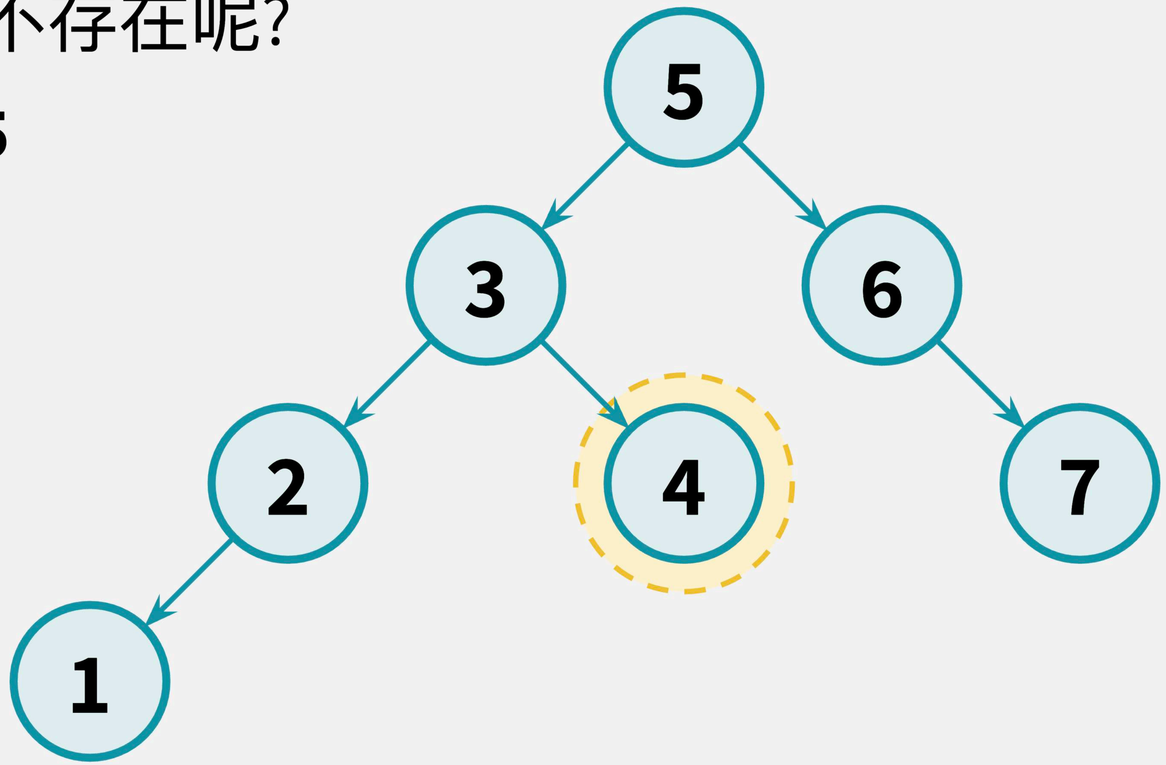
比较 **4.5** 和 **3**:
4.5 更大→ 向右

海量资源：666java.com

BST: SEARCH

如果要查找的值不存在呢?

查找 4.5



比较 **4.5** 和 **root**:
4.5 更小 → 向左

比较 **4.5** 和 **3**:
4.5 更大→ 向右

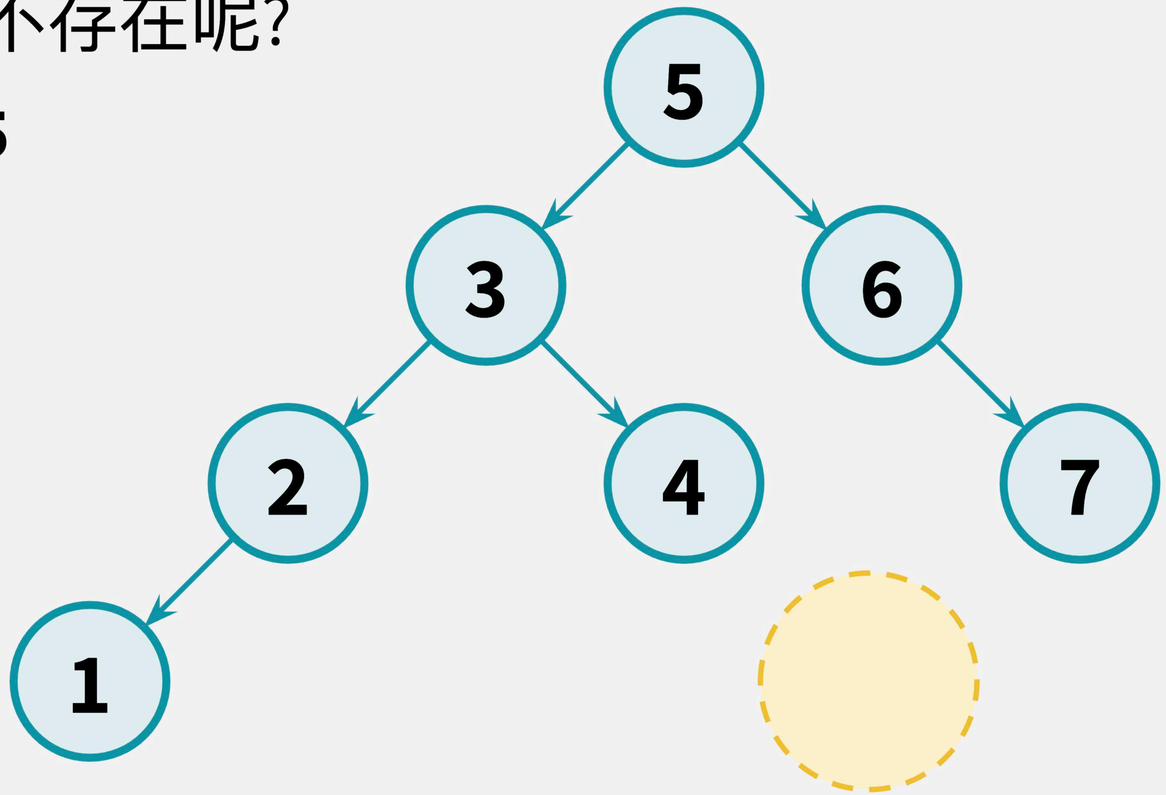
比较 **4.5** 和 **4**:
4.5 更大 → 向右

海量资源：666java.com

BST: SEARCH

如果要查找的值不存在呢?

查找 4.5



比较 **4.5** 和 **root**:
4.5 更小 → 向左

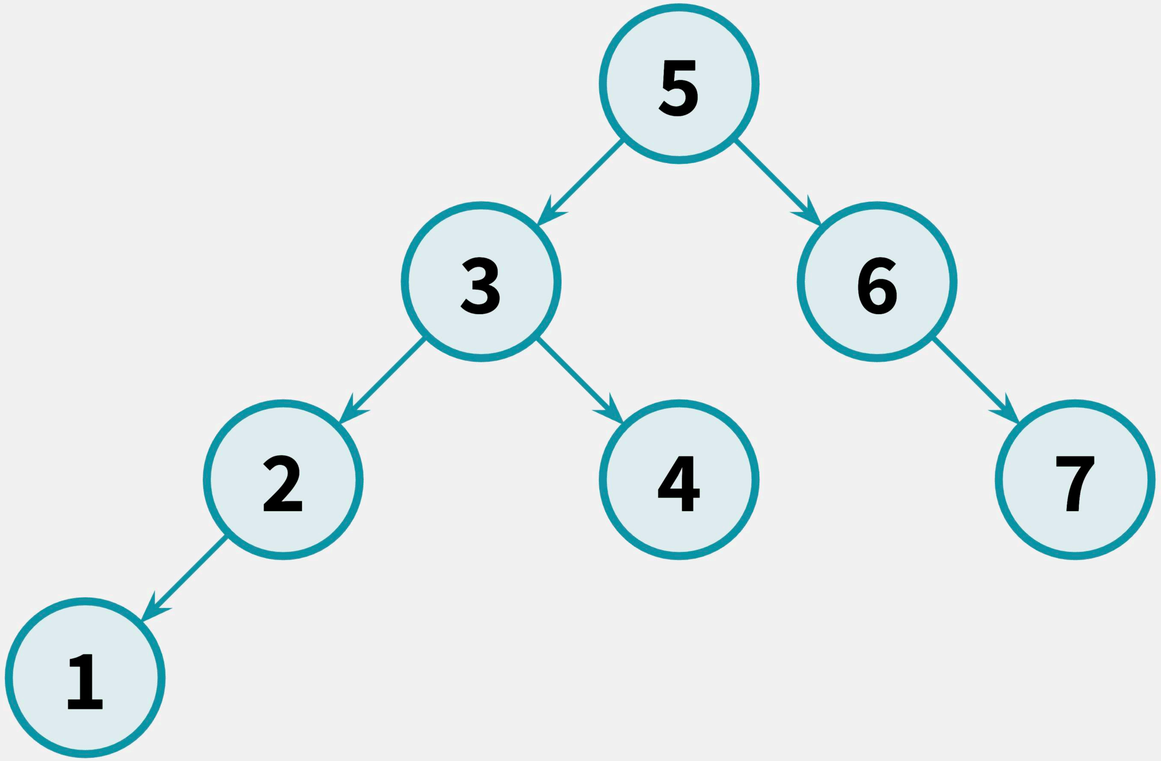
比较 **4.5** 和 **3**:
4.5 更大→ 向右

比较 **4.5** 和 **4**:
4.5 更大 → 向右

遇了 **NIL**, 查找结束

海量资源：666java.com

BST: INSERT

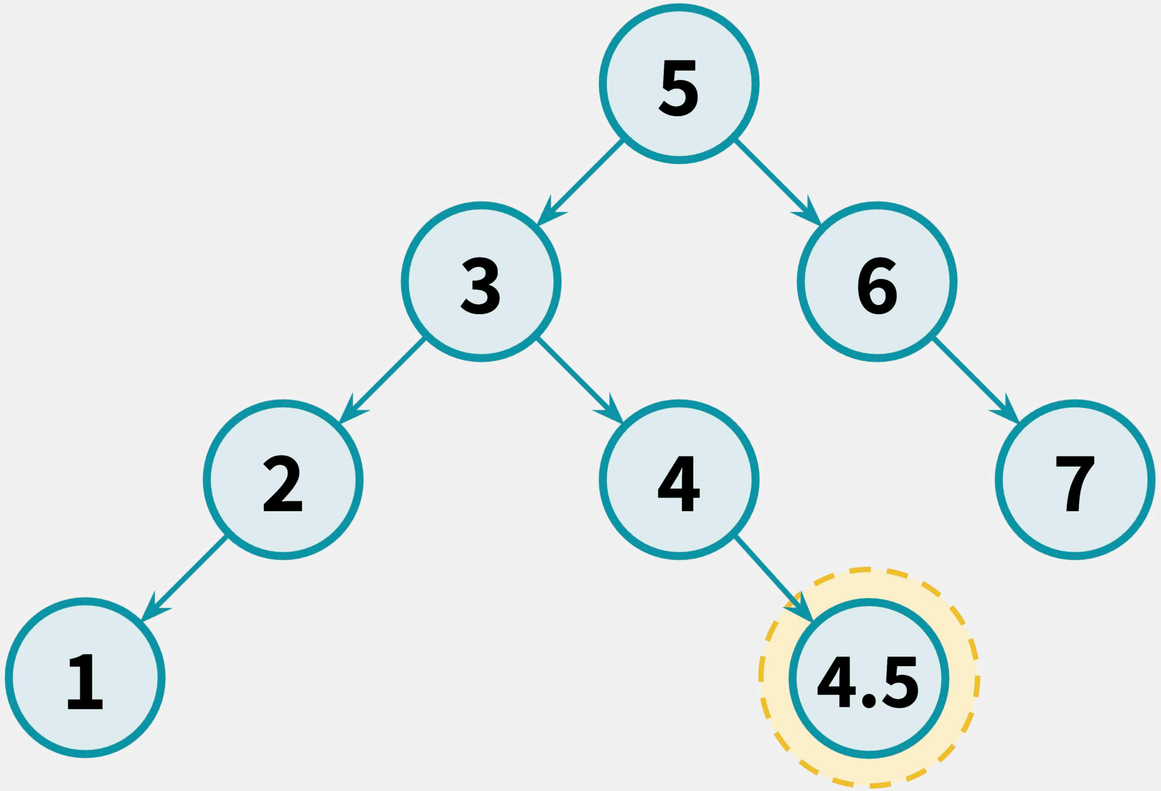



```
INSERT(root, key):
    x = SEARCH(root, key)
    node = new node with key
    if key < x.key:
        x.left = node
    if key > x.key:
        x.right = node
    if key = x.key:
        return
```

海量资源：666java.com

BST: INSERT

例: **插入 4.5**

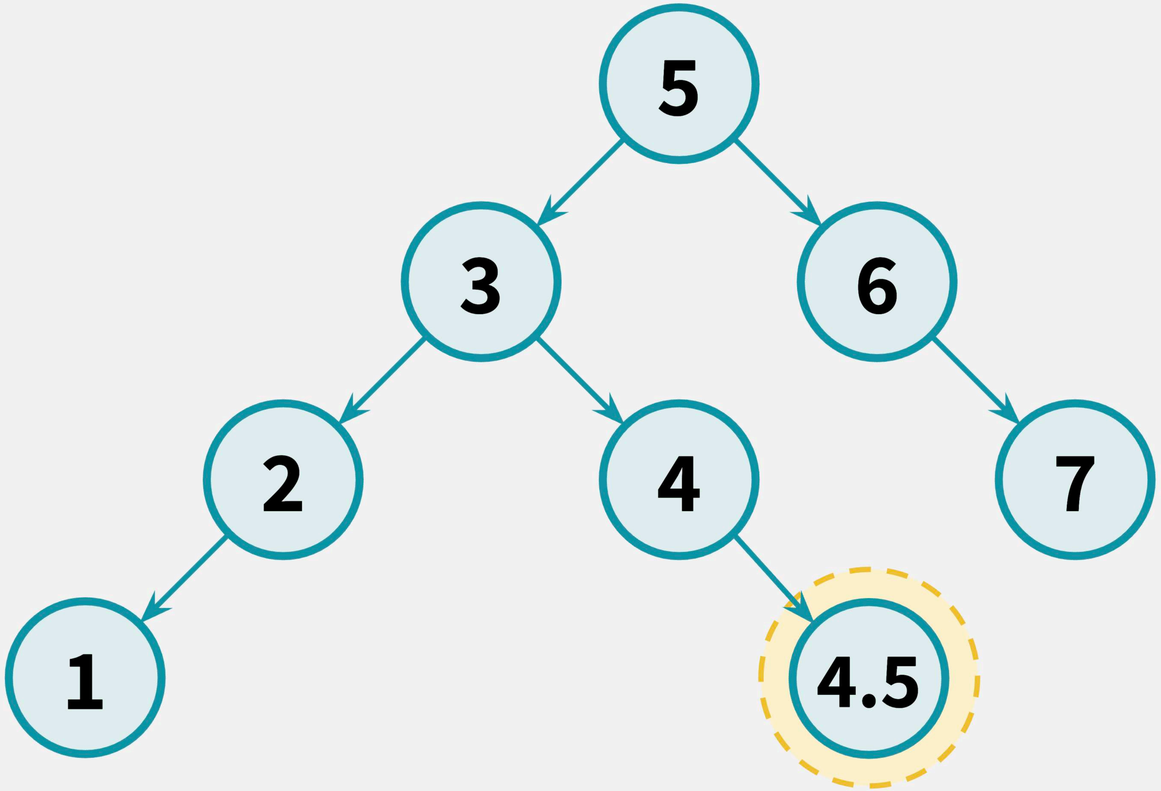



```
INSERT(root, key):
    x = SEARCH(root, key)
    node = new node with key
    if key < x.key:
        x.left = node
    if key > x.key:
        x.right = node
    if key = x.key:
        return
```

海量资源：666java.com

BST: INSERT

例: **插入 4.5**



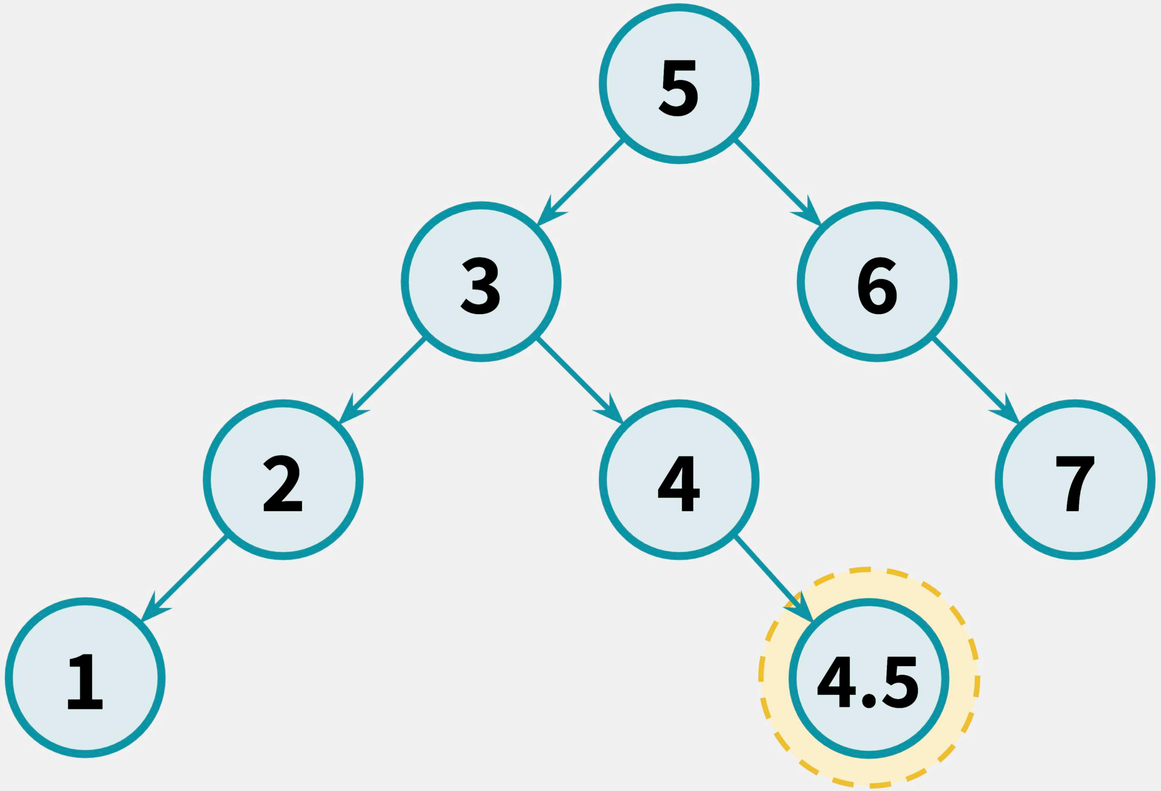

```
INSERT(root, key):
    x = SEARCH(root, key)
    node = new node with key
    if key < x.key:
        x.left = node
    if key > x.key:
        x.right = node
    if key = x.key:
        return
```

插入的运行时间?

海量资源：666java.com

BST: INSERT

例: **插入 4.5**

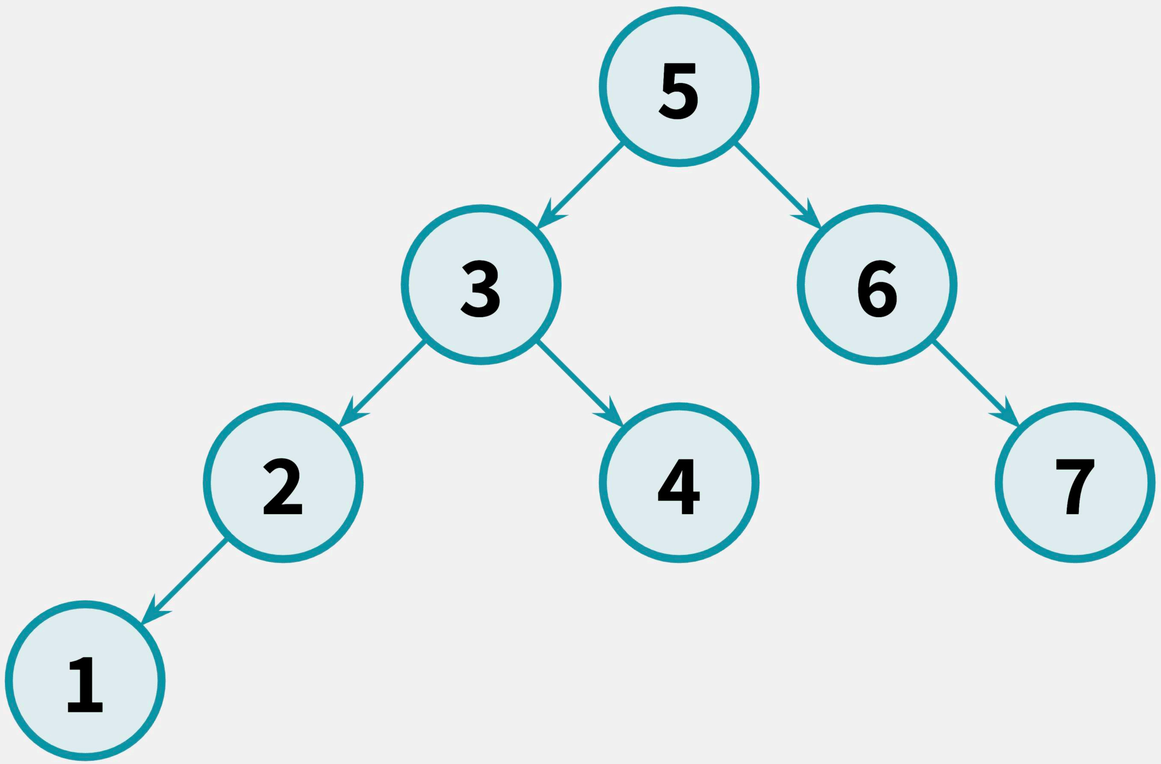



```
INSERT(root, key):
    x = SEARCH(root, key)
    node = new node with key
    if key < x.key:
        x.left = node
    if key > x.key:
        x.right = node
    if key = x.key:
        return
```

插入 的运行时间 = **查找** 的运行时间= **O(高度)**

海量资源：666java.com

BST: DELETE

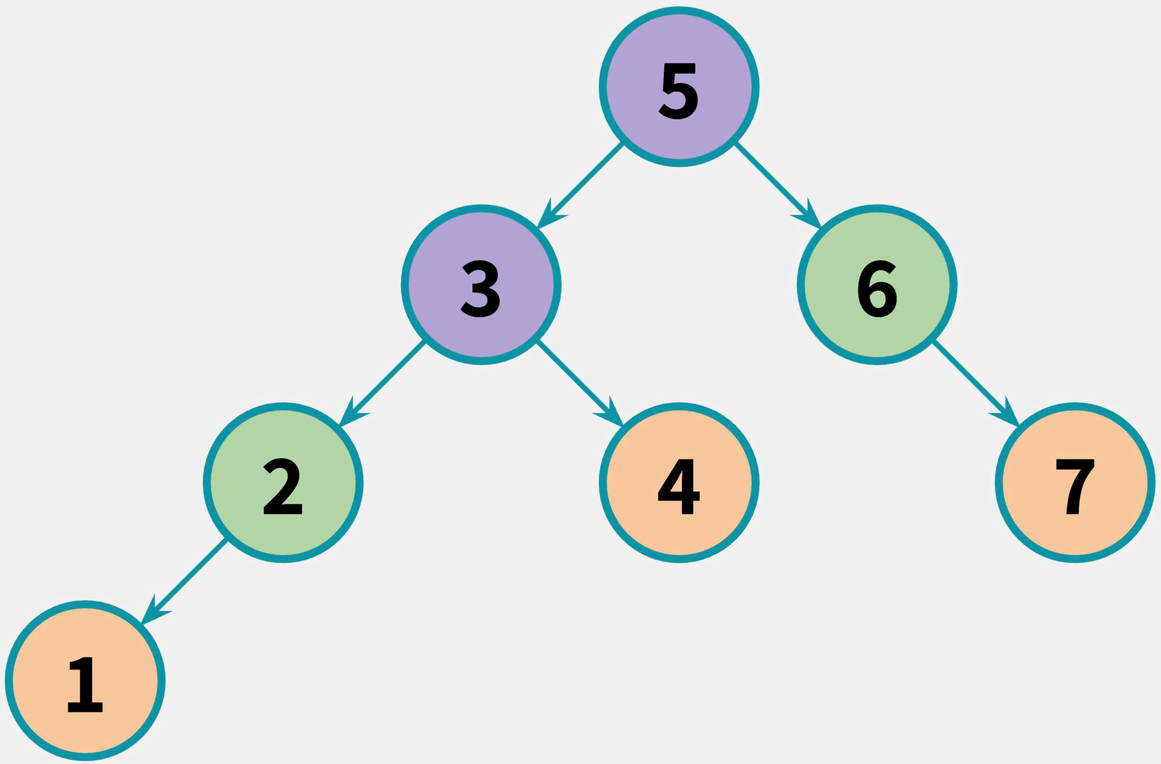



```
DELETE(root, key):
    x = SEARCH(root, key)
    if key = x.key:
        ...delete x...
```

BST 的删除要复杂一些... 需要分三种情况讨论

海量资源：666java.com

BST: DELETE

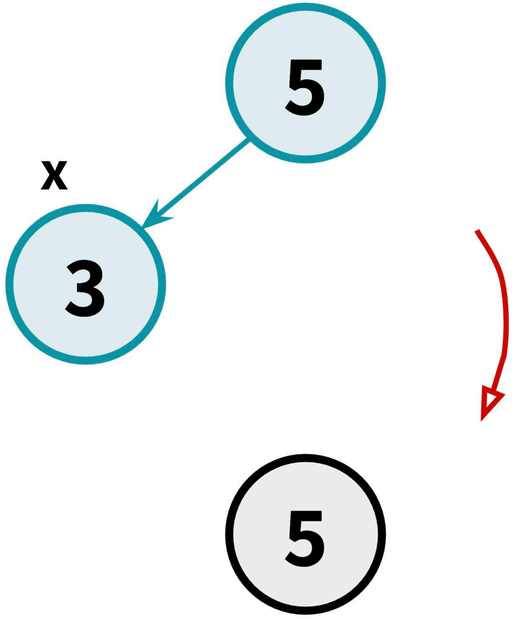



```
DELETE(root, key):
    x = SEARCH(root, key)
    if key = x.key:
        CASE 1: x is a leaf
        CASE 2: x has 1 child
        CASE 3: x has 2 children
```

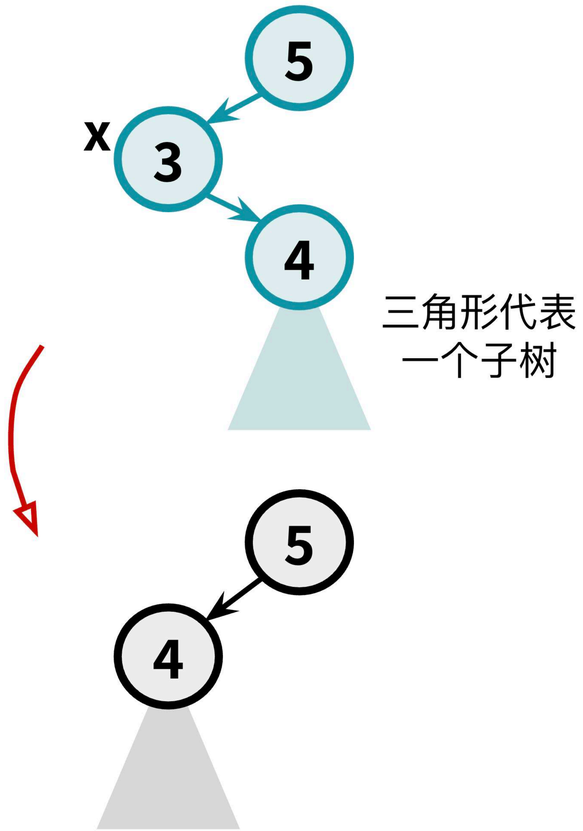
海量资源：666java.com

BST: DELETE

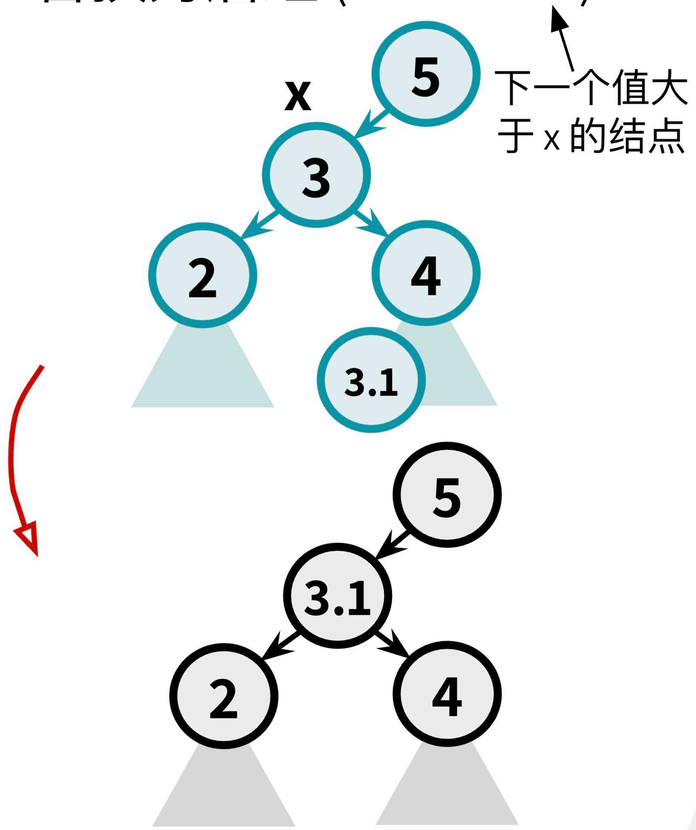
CASE 1: x is a leaf
直接删除 x



CASE 2: x has 1 child
替换为子树



CASE 3: x has 2 children
替换为后继 (successor)



海量资源：666java.com

BST: DELETE

CASE 1: x is a leaf

CASE 2: x has 1 child

CASE 3:

操作确保 BST 的性质

如何寻找后继 (successor)?

在子树 3.right 中查找 3

如何在原 BST 中删除后继?

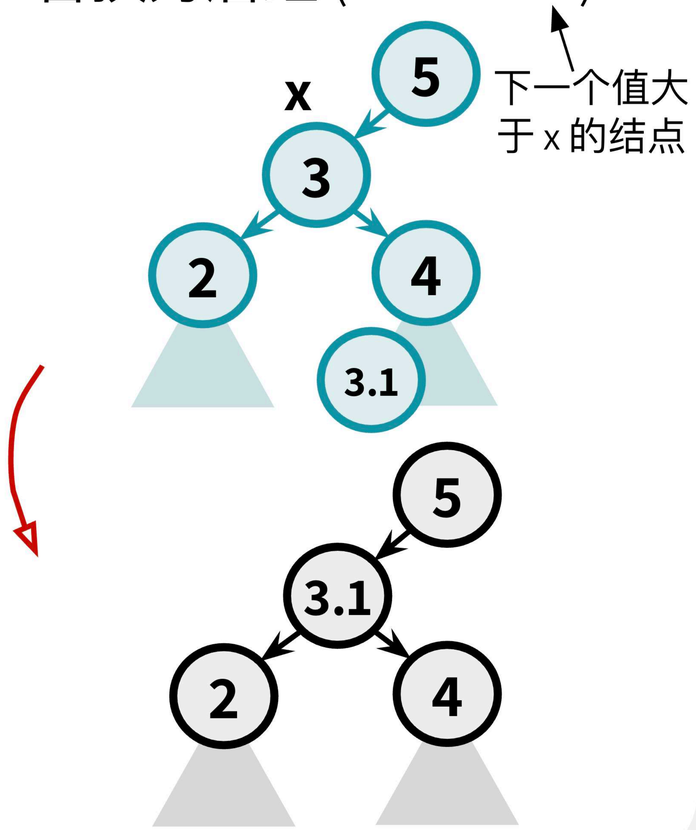
如果 [3.1] 有 0 或 1 个孩子, 应用 CASE 1 或 2.

如果 [3.1] 有两个孩子?

不可能. 否则就不是后继.

CASE 3: x has 2 children

替换为后继 (successor)

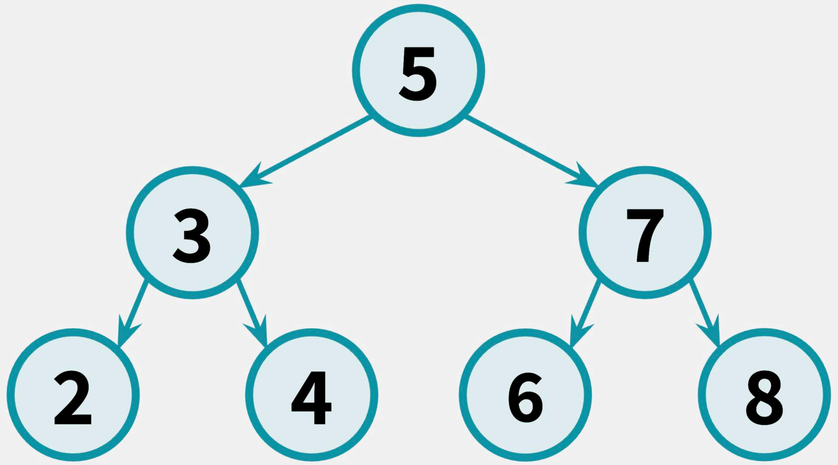


海量资源：666java.com

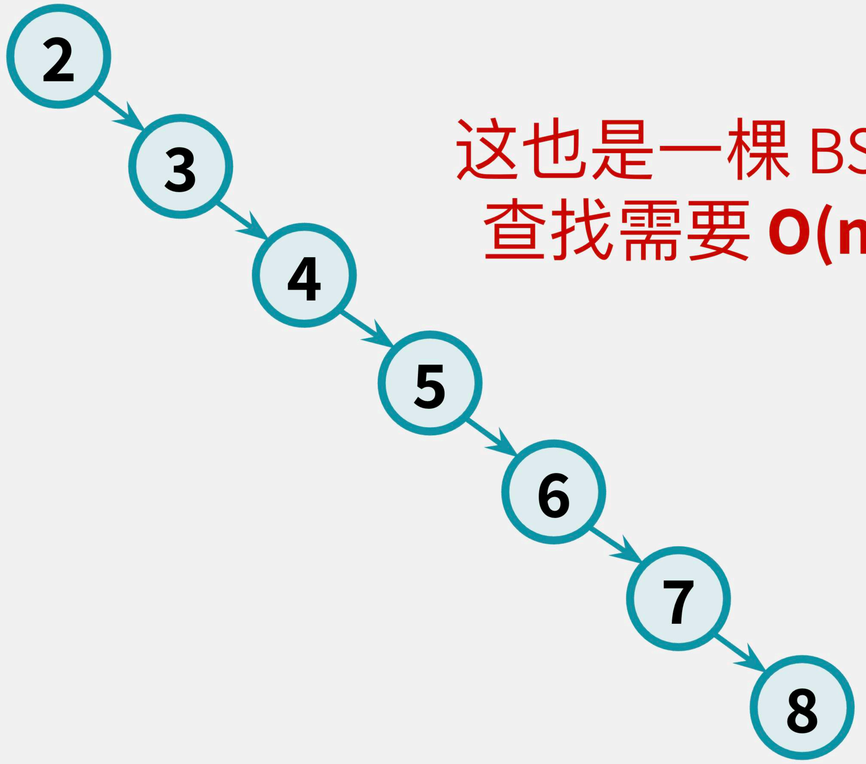
查找/插入/删除 的运行时间

插入 和 **删除** 均调用 **查找** (剩余操作时间为 O(1))

查找 的运行时间 = **O(高度)**



有时查找的运行时间为 **O(log n)**



这也是一棵 BST, 但是
查找需要 **O(n)** 时间

海量资源：666java.com

查找/插入/删除 的运行时间

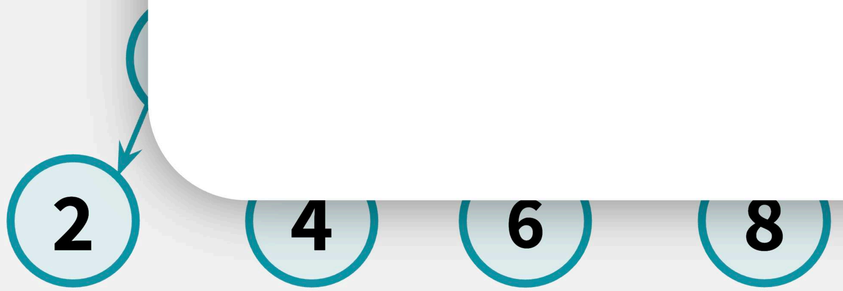
插入 和 **删除** 均调用 **查找** (剩余操作时间为 O(1))

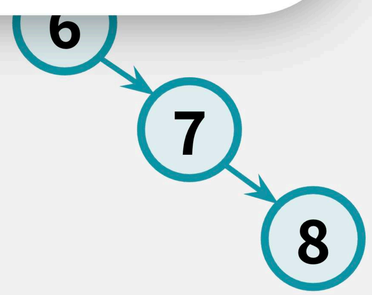
查找 的运行时间 = O(高度)

我们希望快速查找/插入/删除但是高度有时会达到 O(n)
我们希望树是平衡的… 下面介绍

平衡二叉树

BST, 但是
n) 时间





有时查找的运行时间为 **O(log n)**

海量资源：666java.com

平衡二叉树

海量资源：666java.com

平衡二叉树

平衡二叉树有多种实现方式

(例如 红黑树, AVL 树, B-树, 2-3-4 树)

最早被发明的自平衡二叉查找树，有时也被直接称为平衡二叉树

海量资源：666java.com

平衡二叉树

平衡二叉树有多种实现方式
(例如 红黑树, AVL 树, B-树, 2-3-4 树)

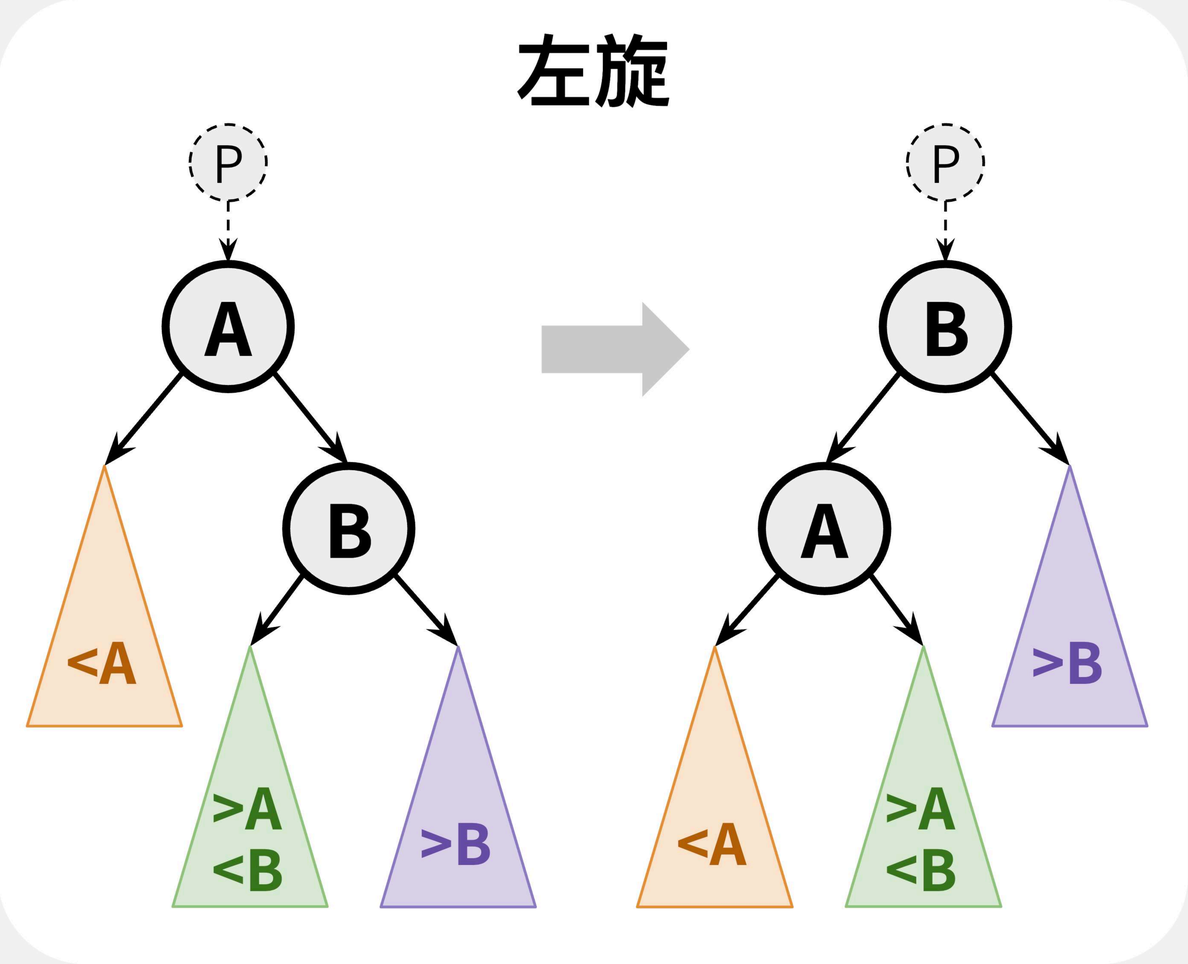
维护二叉搜索树平衡的基本操作操作:

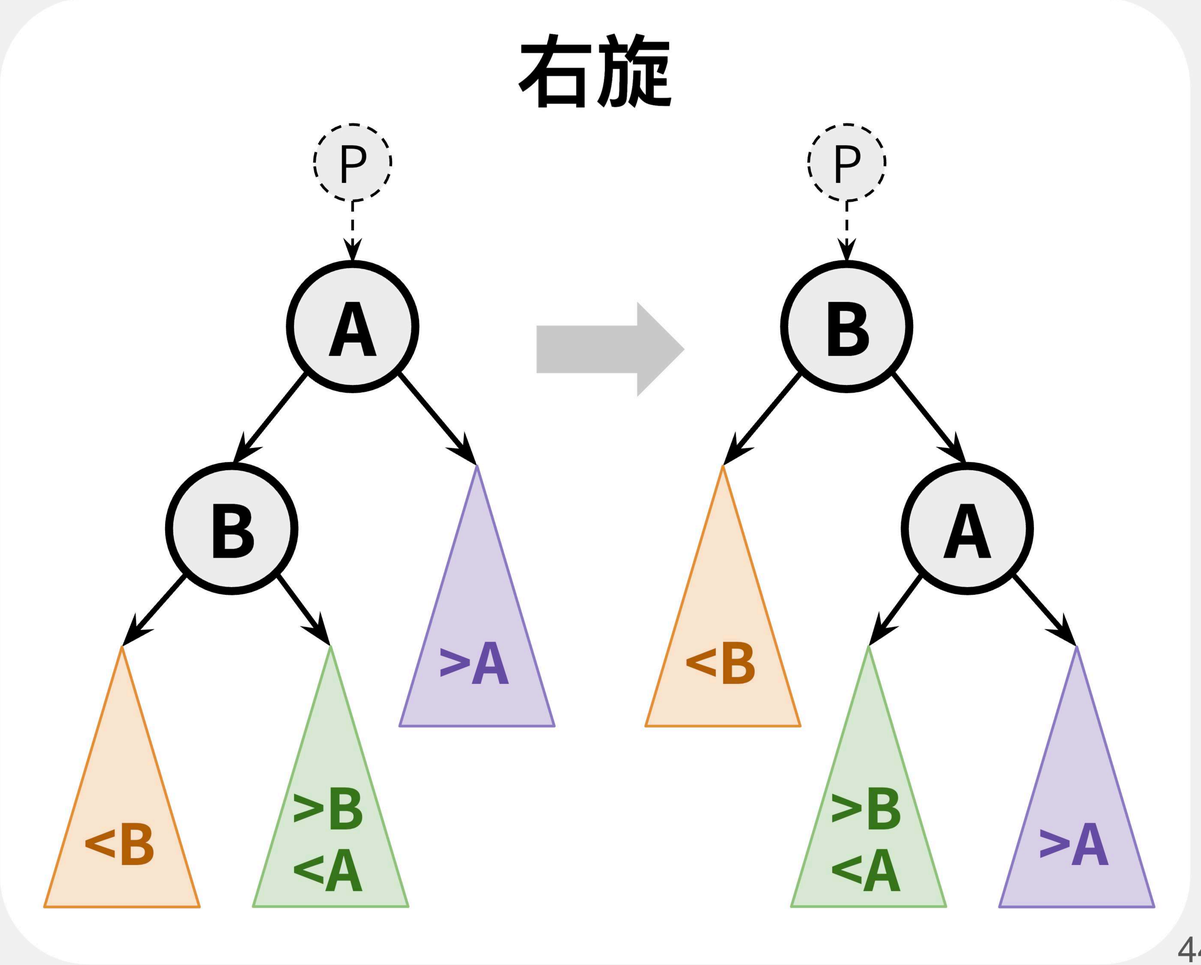
旋转

海量资源：666java.com

旋转

思路: 在 O(1) 时间内重新平衡当前结点的子树来保持整棵 BST 的平衡

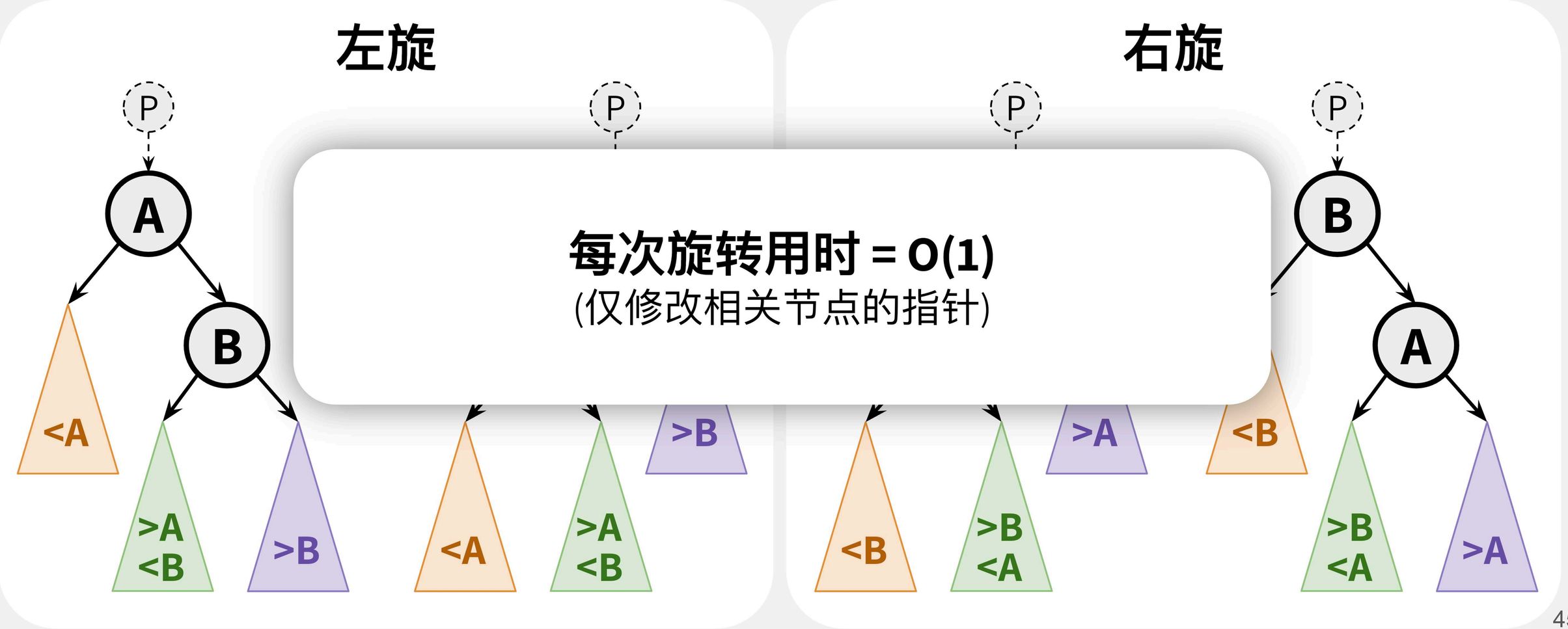




海量资源：666java.com

旋转

思路: 在 O(1) 时间内重新平衡当前结点的子树来保持整棵 BST 的平衡



每次旋转用时 = O(1)
(仅修改相关节点的指针)

海量资源：666java.com

旋转

什么时候需要旋转操作?

下面介绍一个著名的自平衡 BST:

红黑树

海量资源：666java.com

红黑树

一种非常著名的平衡二叉树

海量资源：666java.com

红黑树

红黑树 Red-Black Tree (RB tree) 是一棵 **BST** 且拥有以下性质:

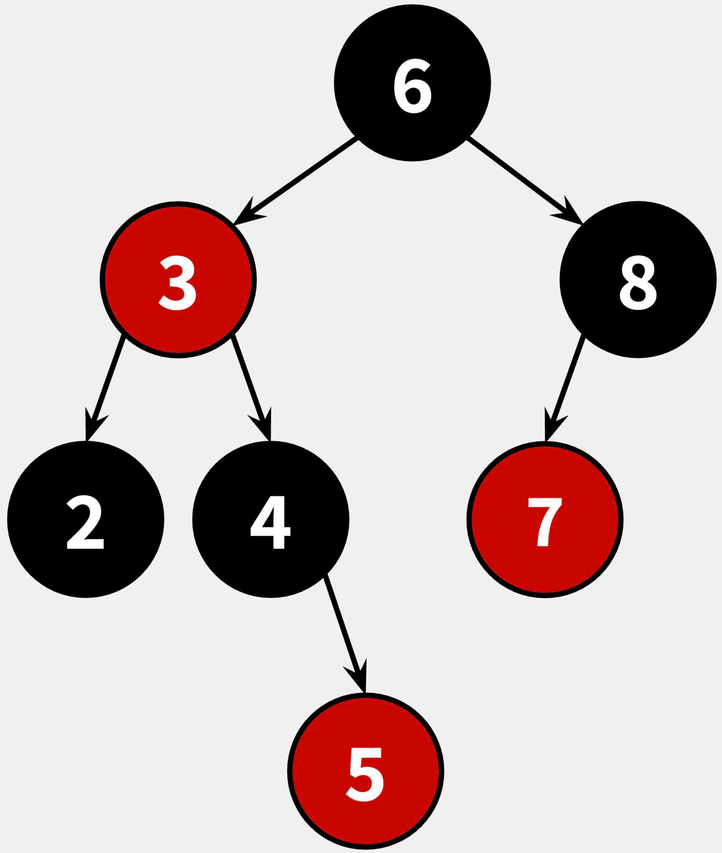
1. 每个结点是 **红色** 或 **黑色**
2. 根结点是 **黑色**
3. **红色** 的子节点不能是 **红色**
4. 每条 root-NIL 路径中包含的**黑色**结点数量相同

海量资源：666java.com

红黑树

红黑树 Red-Black Tree (RB tree) 是一棵 **BST** 且拥有以下性质:

1. 每个结点是 **红色** 或 **黑色**
2. 根结点是 **黑色**
3. **红色** 的子节点不能是 **红色**
4. 每条 root-NIL 路径中包含的**黑色**结点数量相同

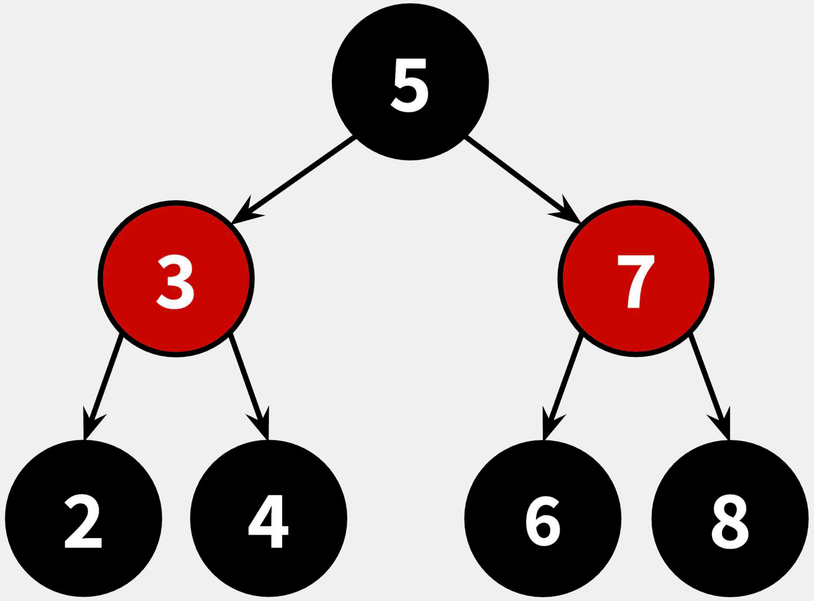


海量资源：666java.com

红黑树

红黑树 Red-Black Tree (RB tree) 是一棵 **BST** 且拥有以下性质:

1. 每个结点是 **红色** 或 **黑色**
2. 根结点是 **黑色**
3. **红色** 的子节点不能是 **红色**
4. 每条 root-NIL 路径中包含的**黑色**结点数量相同

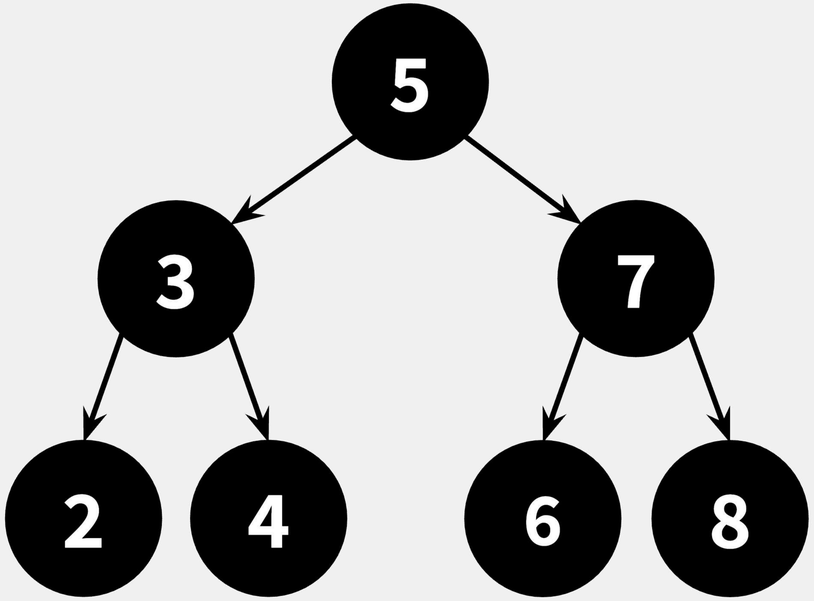


海量资源：666java.com

红黑树

红黑树 Red-Black Tree (RB tree) 是一棵 **BST** 且拥有以下性质:

1. 每个结点是 **红色** 或 **黑色**
2. 根结点是 **黑色**
3. **红色** 的子节点不能是 **红色**
4. 每条 root-NIL 路径中包含的**黑色**结点数量相同

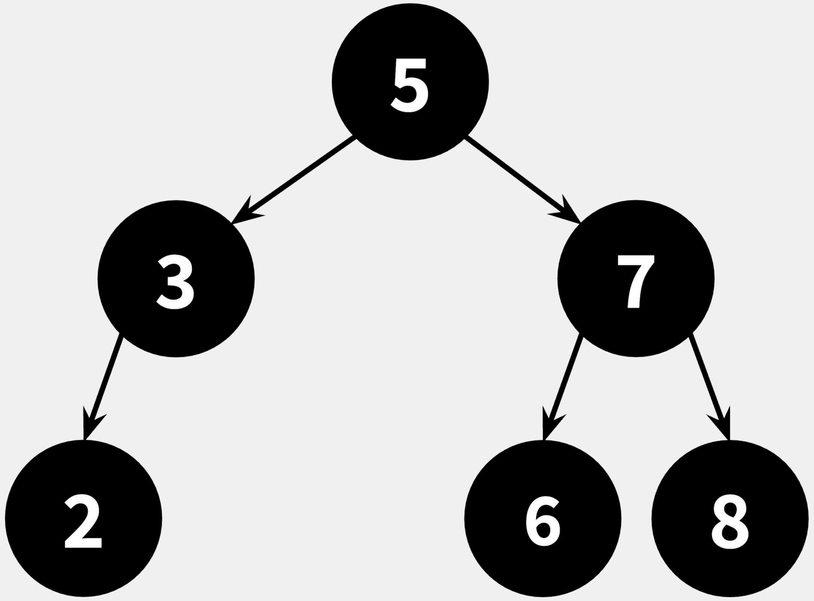


海量资源：666java.com

红黑树

红黑树 Red-Black Tree (RB tree) 是一棵 **BST** 且拥有以下性质:

1. 每个结点是 **红色** 或 **黑色**
2. 根结点是 **黑色**
3. **红色** 的子节点不能是 **红色**
4. 每条 root-NIL 路径中包含的**黑色**结点数量相同

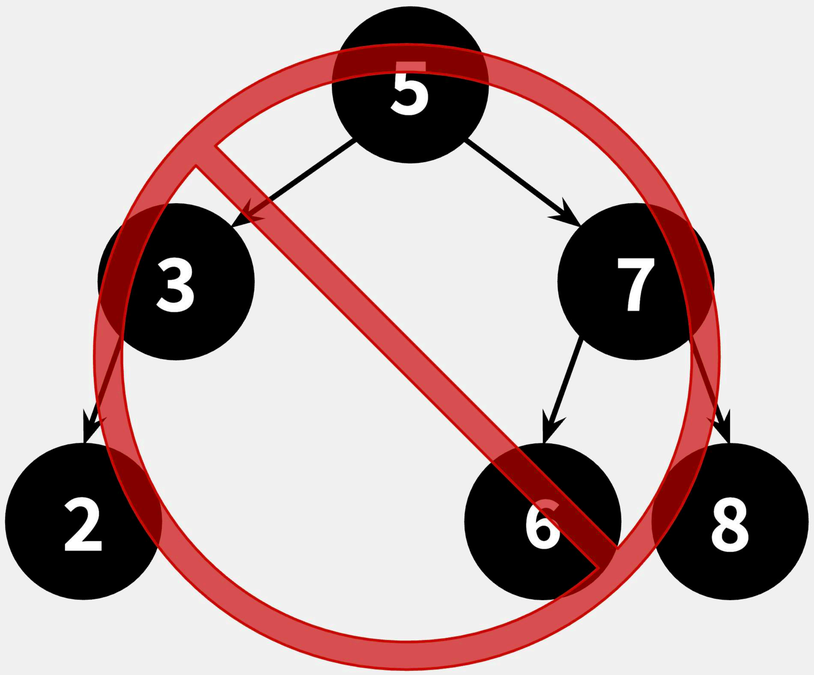


海量资源：666java.com

红黑树

红黑树 Red-Black Tree (RB tree) 是一棵 **BST** 且拥有以下性质:

1. 每个结点是 **红色** 或 **黑色**
2. 根结点是 **黑色**
3. **红色** 的子节点不能是 **红色**
4. 每条 root-NIL 路径中包含的**黑色**结点数量相同

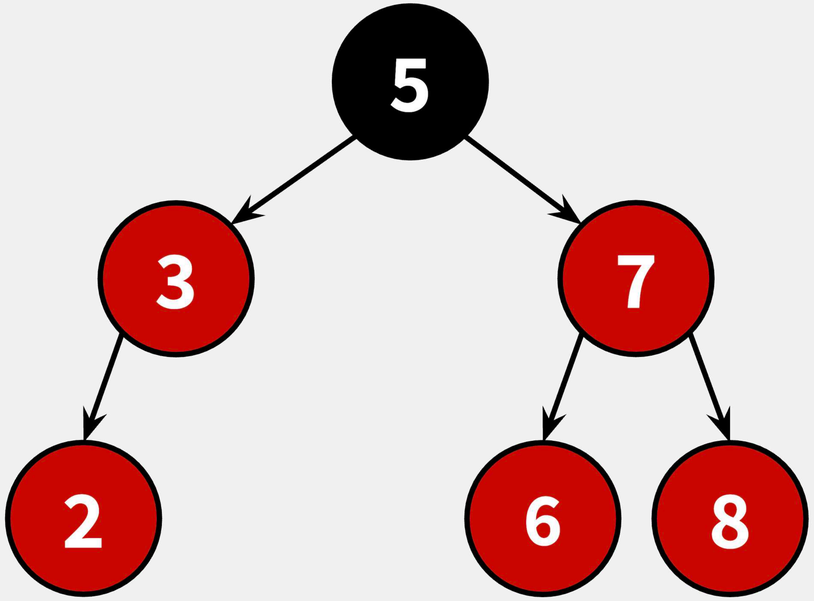


海量资源：666java.com

红黑树

红黑树 Red-Black Tree (RB tree) 是一棵 **BST** 且拥有以下性质:

1. 每个结点是 **红色** 或 **黑色**
2. 根结点是 **黑色**
3. **红色** 的子节点不能是 **红色**
4. 每条 root-NIL 路径中包含的**黑色**结点数量相同

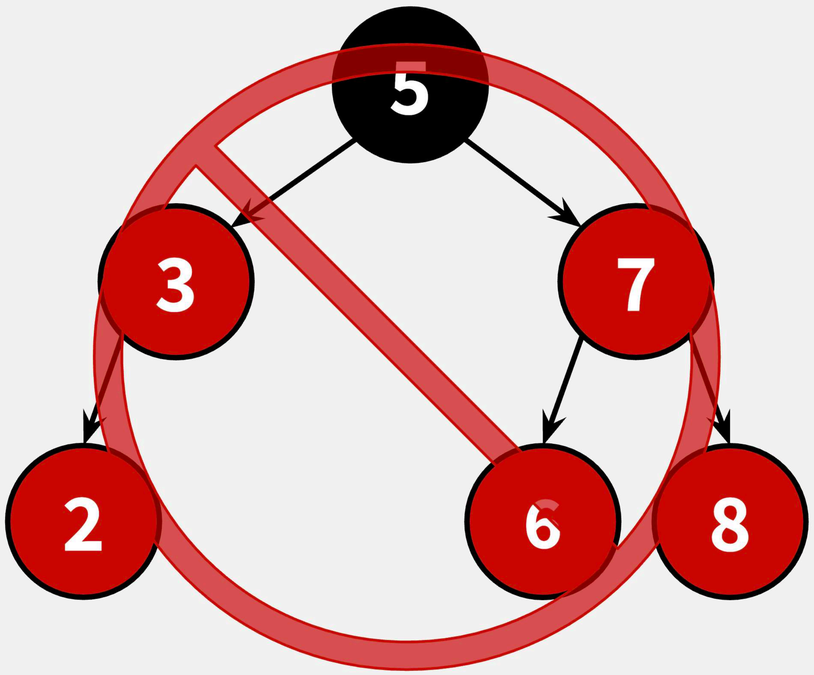


海量资源：666java.com

红黑树

红黑树 Red-Black Tree (RB tree) 是一棵 **BST** 且拥有以下性质:

1. 每个结点是 **红色** 或 **黑色**
2. 根结点是 **黑色**
3. **红色** 的子节点不能是 **红色**
4. 每条 root-NIL 路径中包含的**黑色**结点数量相同



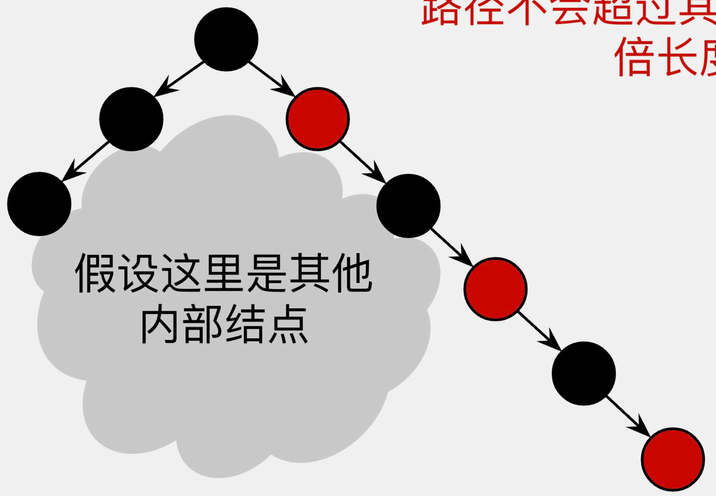
海量资源：666java.com

为什么要这样规定?

1. 每个结点是 **红色** 或 **黑色**
2. 根结点是 **黑色**
3. **红色** 的子节点不能是 **红色**
4. 每条 root-NIL 路径中包含的**黑色**结点数量相同

3 & 4 保证了包含红色结点的路径不会超过其他路径的两倍长度

可以通过结点着色来体现树的平衡情况:
黑色 结点在树中的分布是近似平衡的
红色 结点会“拉长”所在路径但不会太多



海量资源：666java.com

为什么要这样规定?

1. 每个结点是 **红色** 或 **黑色**
2. 根结点是 **黑色**
3. **红色** 的子节点不能是 **红色**
4. 每条 root-NIL 路径中包含的**黑色**结点数量相同

定理: 任何包含 **n** 个结点的红黑树的高度为 **O(log n)**

海量资源：666java.com

为什么要这样规定?

1. 每个结点是 **红色** 或 **黑色**
2. 根结点是 **黑色**
3. **红色** 的子节点不能是 **红色**
4. 每条 root-NIL 路径中包含的**黑色**结点数量相同

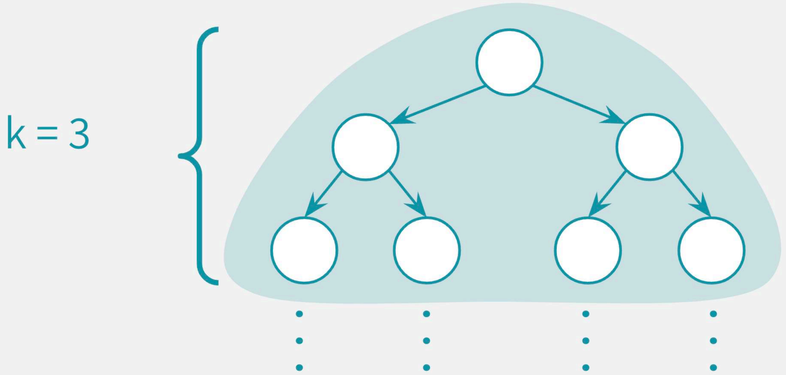
定理: 任何包含 **n** 个结点的红黑树的高度为 **O(log n)**

思路: 任何包含 **n** 个结点的红黑树的高度 $\leq 2 \cdot \log_2(n+1)$

海量资源：666java.com

红黑树高度满足 O(log n)

假设每条 root-NIL 路径上包含结 ≥ **k** 个结点. 则红黑树内必存在一棵高度为 **k - 1** 的平衡二叉子树.



阴影部分包含多少个结点?

$\mathbf{2^k - 1}$

由于整棵红黑树的结点数为 **n**:

$n \geq 2^k - 1$

即 $\mathbf{k \leq \log_2(n+1)}$

红黑树性质

1. 每个结点是 **红色** 或 **黑色**
2. 根结点是 **黑色**
3. **红色** 的子节点不能是 **红色**
4. 每条 root-NIL 路径中包含的**黑色**结点数量相同

如果所有路径长度均 $> \log_2(n+1)$, 则 k 无法满足 $\leq \log_2(n+1)$

也就是说一定存在这样的 root-NIL 路径, 其包含结点数 $\leq \log_2(n+1)$

根据定义, 该路径上的黑色结点数 $\leq \log_2(n+1)$

由性质 4: *每一条* root-NIL 路径上的黑色结点数均 $\leq \log_2(n+1)$

由性质 3: *每一条* root-NIL 路径上的结点总数 $\leq 2 \cdot \log_2(n+1)$

所以, 红黑树的高度至多为 $2 \cdot \log_2(n+1)$, 即 **任意红黑树的高度满足 O(log n)**

海量资源：666java.com

红黑树高度满足 O(log n)

红黑树的高度满足 O(log n).

红黑树上 **查找** 的运行时间 = **O(高度)**
= O(log n)

海量资源：666java.com

红黑树高度满足 O(log n)

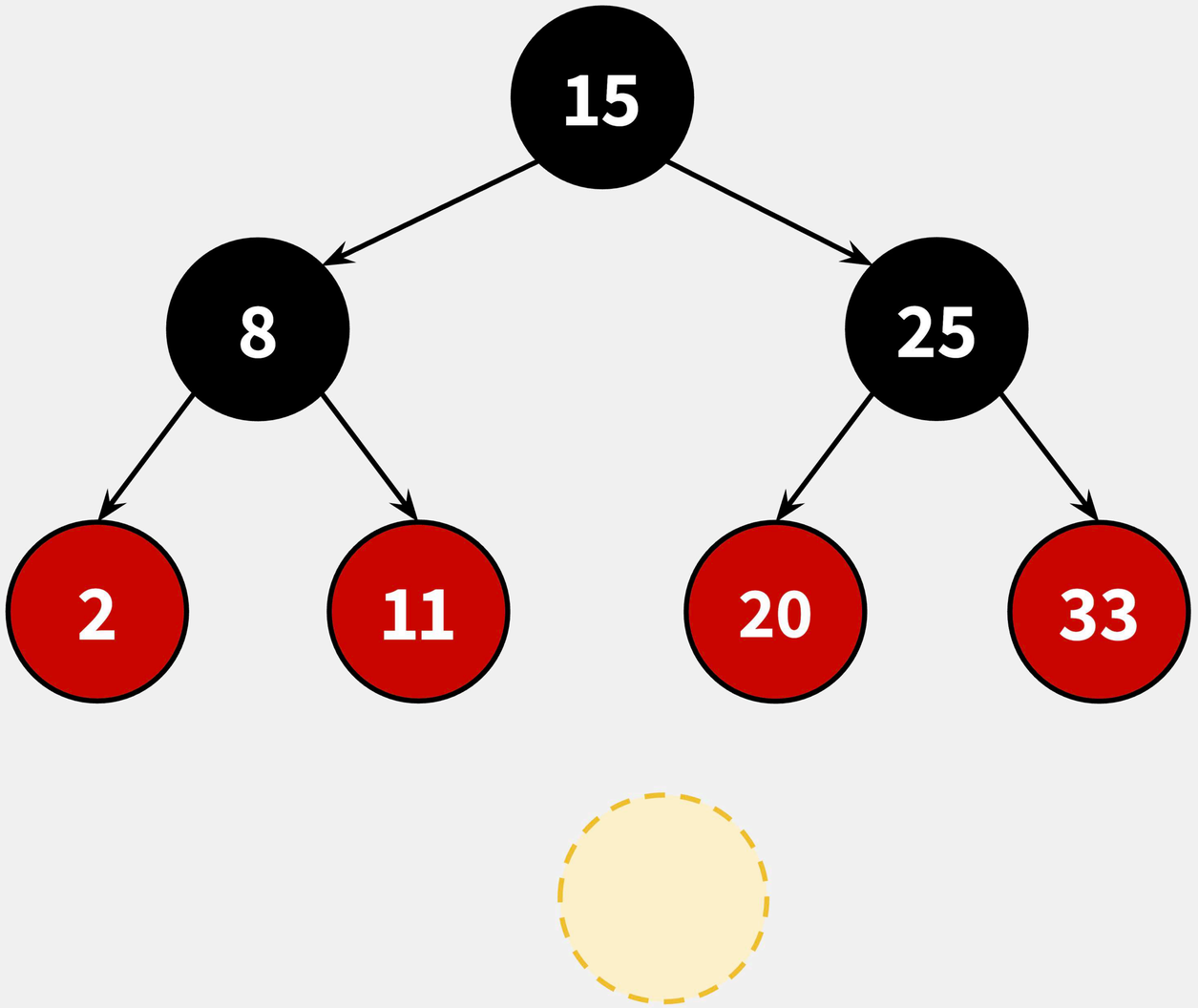
红黑树的高度满足 O(log n).

红黑树上 **查找** 的运行时间 = **O(高度)**
= O(log n)

插入/删除?
由于插入/删除操作修改了红黑树中的结点, 我们必须保证操作完成后红黑树依然保持其性质...

海量资源：666java.com

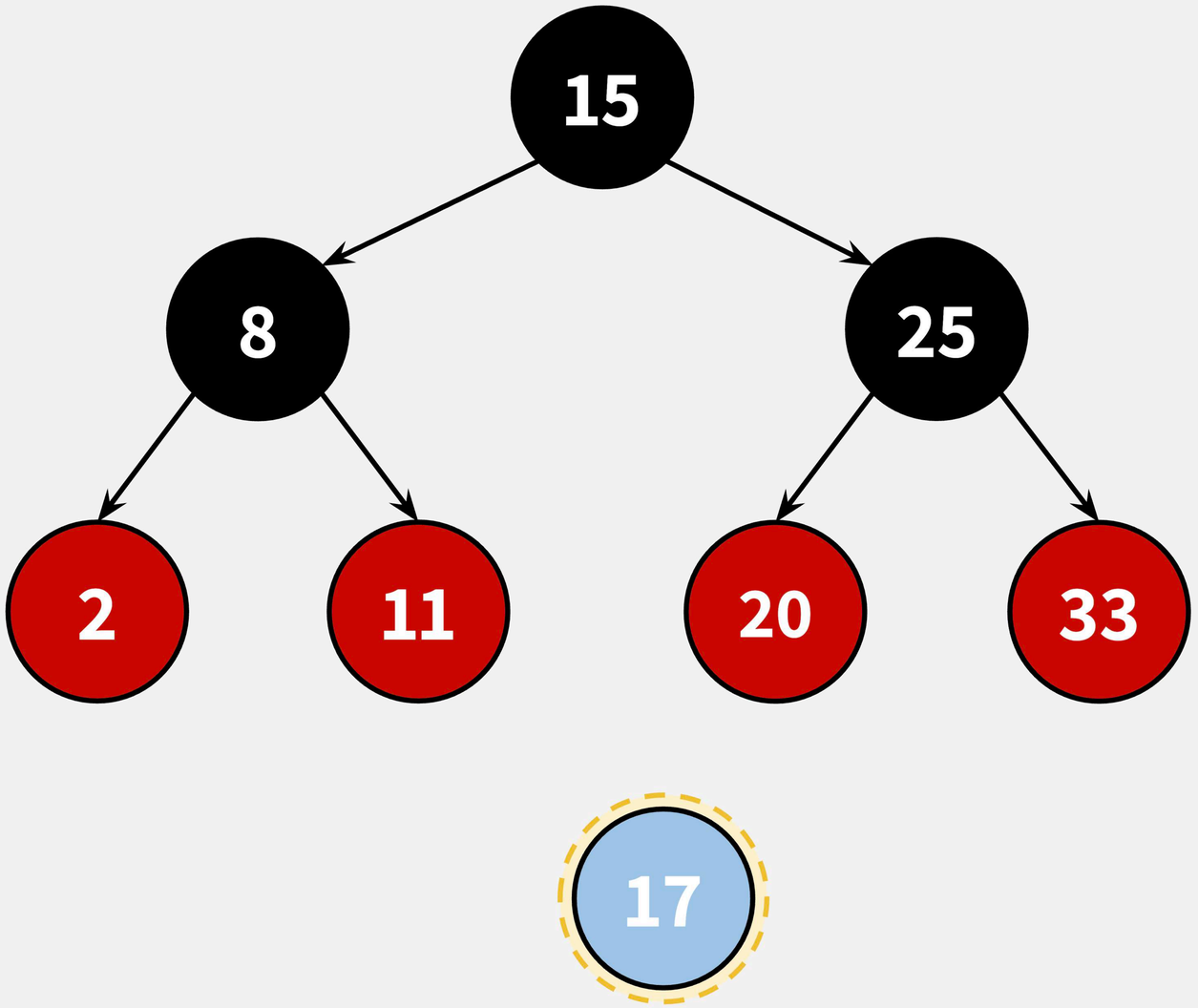
插入操作



例: 插入 17

海量资源：666java.com

插入操作



例: 插入 17

需要哪些操作?

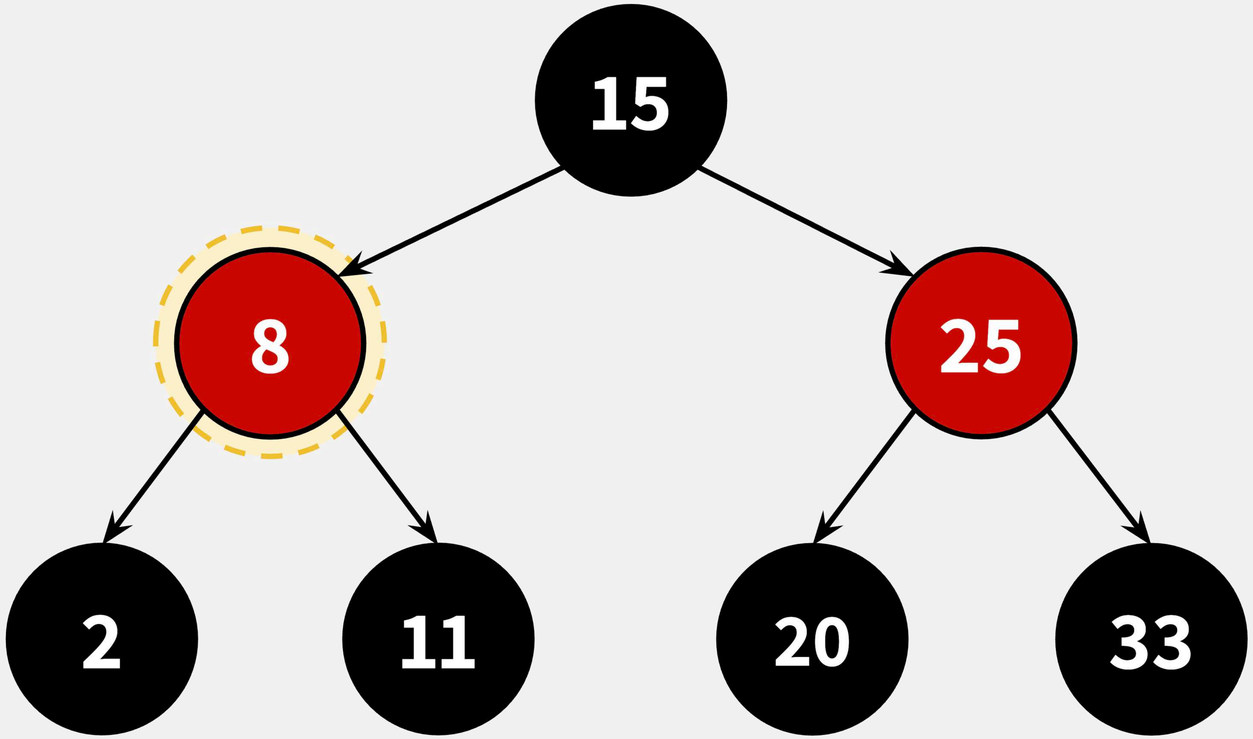
新结点应该是 **红色** 还是 **黑色**?

其他结点的颜色是否需要调整?

如果之后插入 16?

海量资源：666java.com

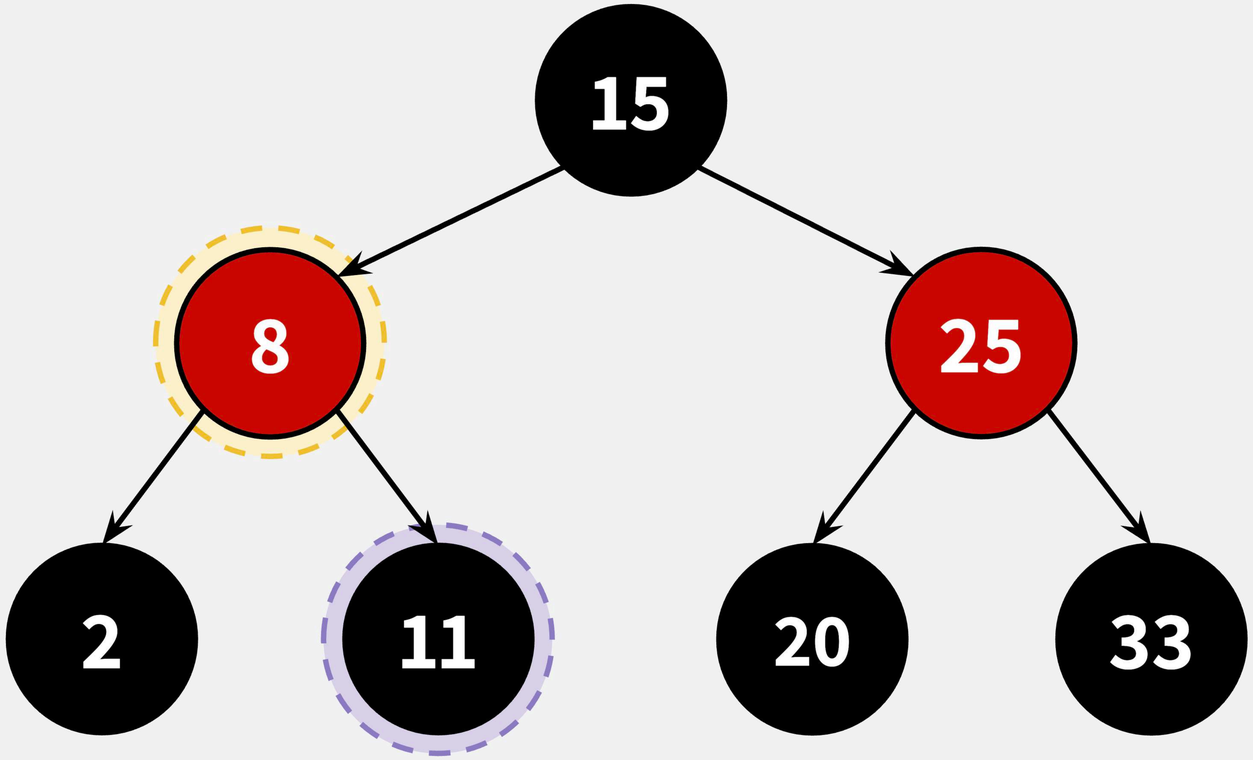
删除操作



例: 删除 8

海量资源：666java.com

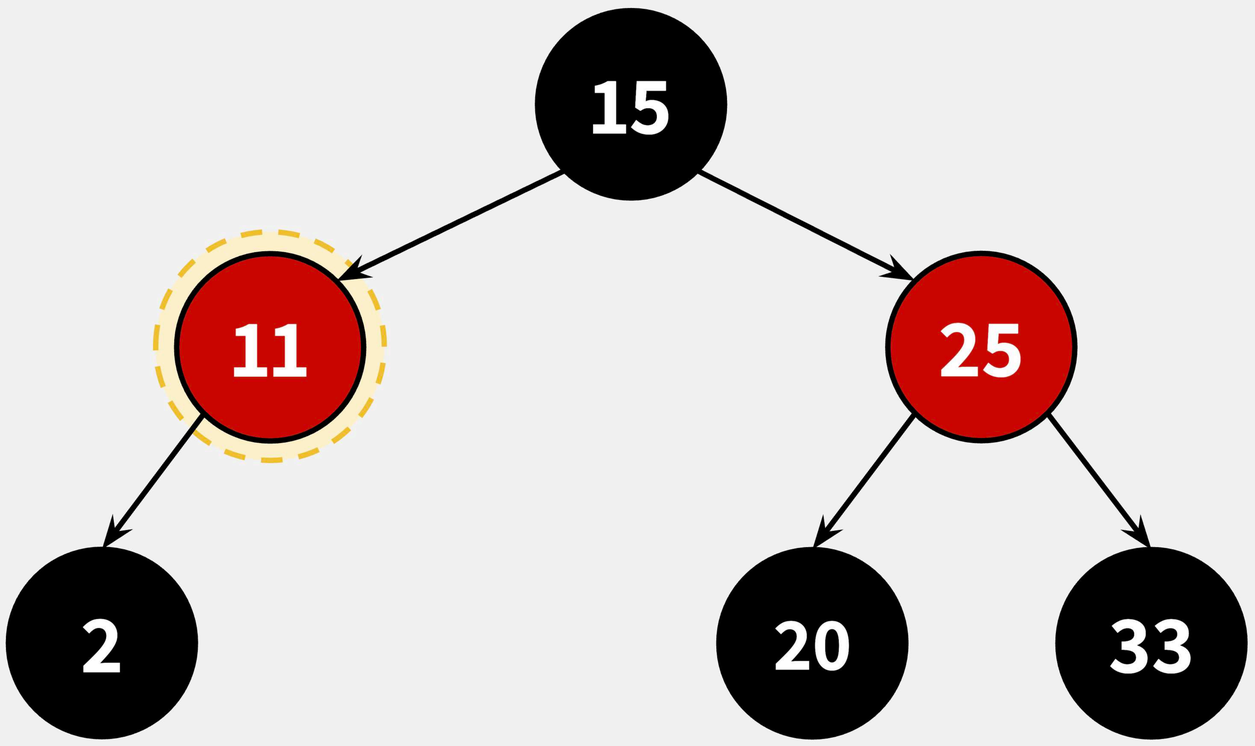
删除操作



例: 删除 8
(替换为其后继)

海量资源：666java.com

删除操作



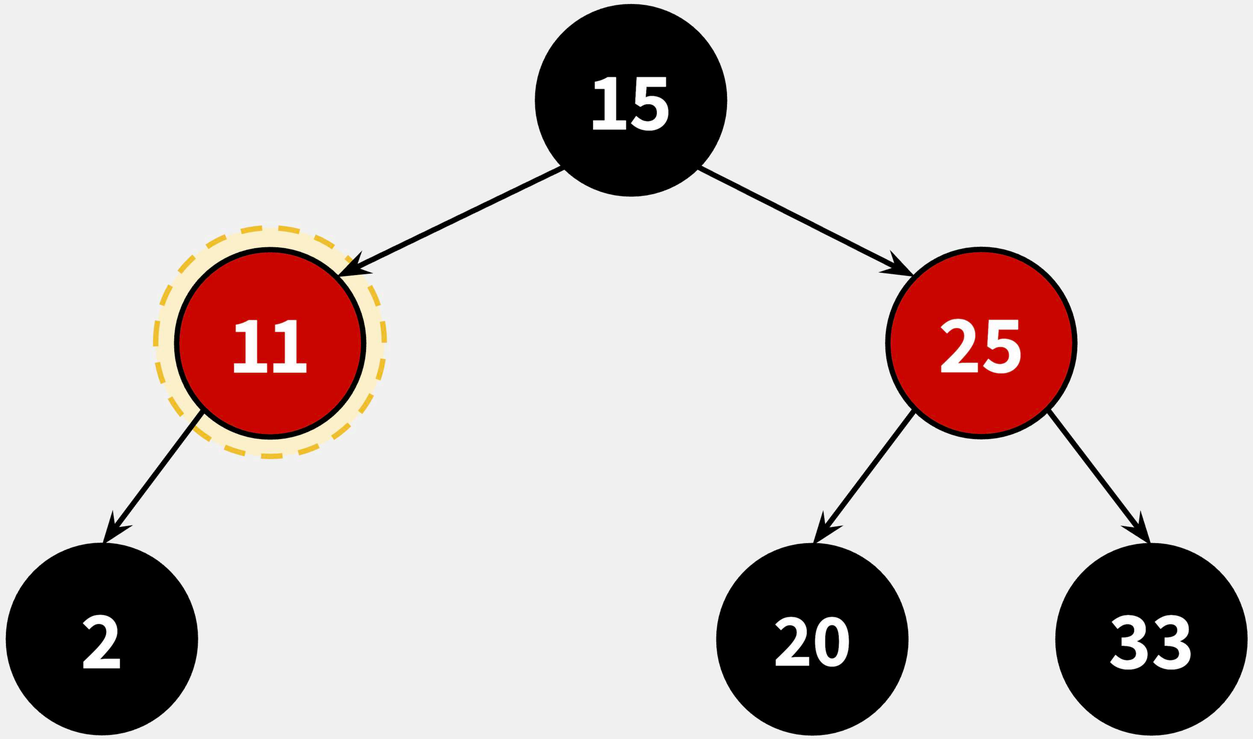
例: 删除 8
(替换为其后继)

破坏了性质 4
(每条 root-NIL 路径中包含的**黑色**结点数量相同)

如何修改?

海量资源：666java.com

删除操作



例: 删除 8
(替换为其后继)

破坏了性质 4
(每条 root-NIL 路径中包含的
黑色结点数量相同)

如何修改?

删除后的调整操作非常复杂. 详见 算法导论 13.4

海量资源：666java.com

红黑树中的插入操作

插入新元素的同时保持红黑树的性质

海量资源：666java.com

插入操作

算法思路

像二叉搜索树那样插入新结点，再进行调整

调整 = 为了维护红黑树的性质进行结点颜色的修改及结点旋转

海量资源：666java.com

插入操作

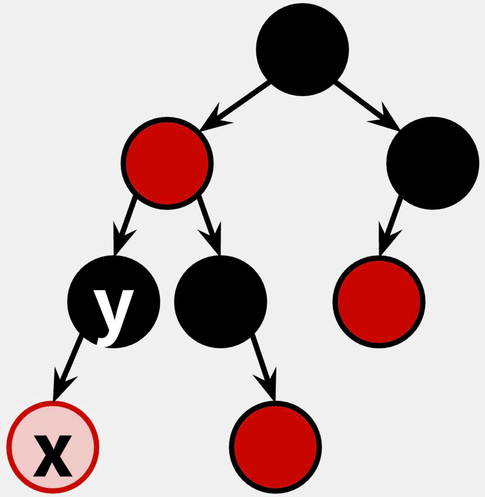
算法思路

像二叉搜索树那样插入新结点, 再进行调整

调整 = 为了维护红黑树的性质进行结点颜色的修改及结点旋转

INSERT(x):

- 插入 **x** (**x** 成为了新的叶结点)
- 将 **x** 标为 **红色**
- 如果 **x** 的父结点 **y** 为 **黑色**, 则插入完成



海量资源：666java.com

插入操作

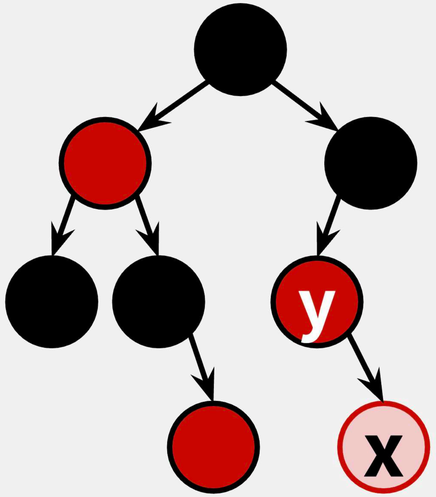
算法思路

像二叉搜索树那样插入新结点, 再进行调整

调整 = 为了维护红黑树的性质进行结点颜色的修改及结点旋转

INSERT(x):

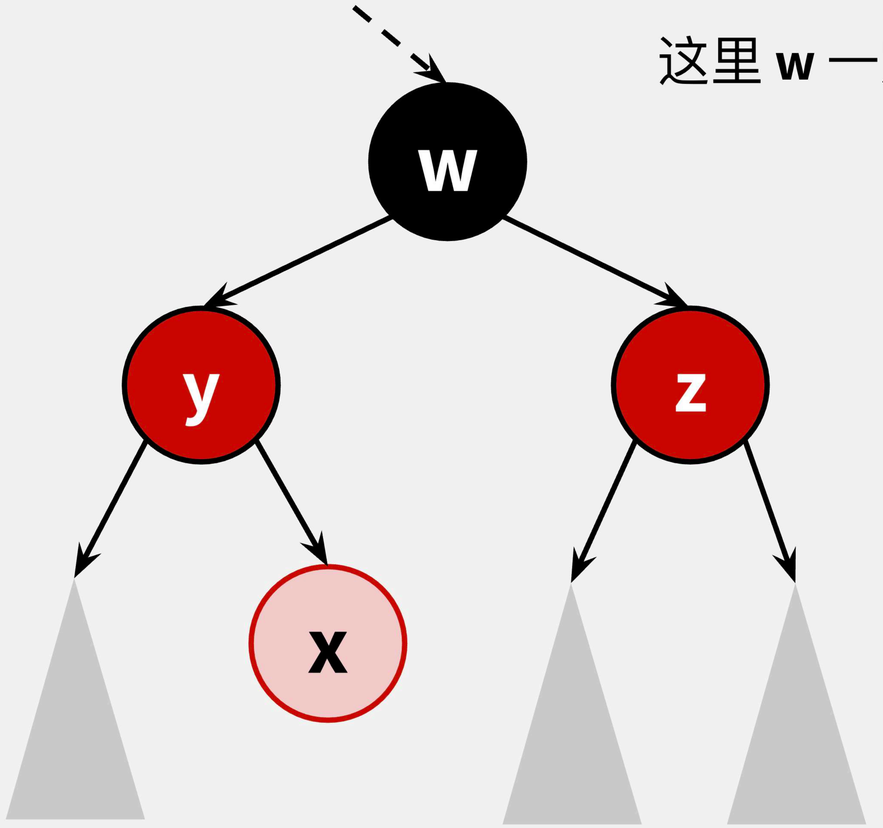
- 插入 **x** (**x** 成为了新的叶结点)
- 将 **x** 标为 **红色**
- 如果 **x** 的父结点 **y** 为 **黑色**, 则插入完成
- 如果 **y** 为 **红色**, 则需要调整



海量资源：666java.com

插入后调整: CASE 1

CASE 1: 父结点 **y** 为 **红色**, 且 “叔结点” **z** 也为 **红色**



这里 **w** 一定是 **黑色**，因为不存在两个连续的 **红色** 结点

海量资源：666java.com

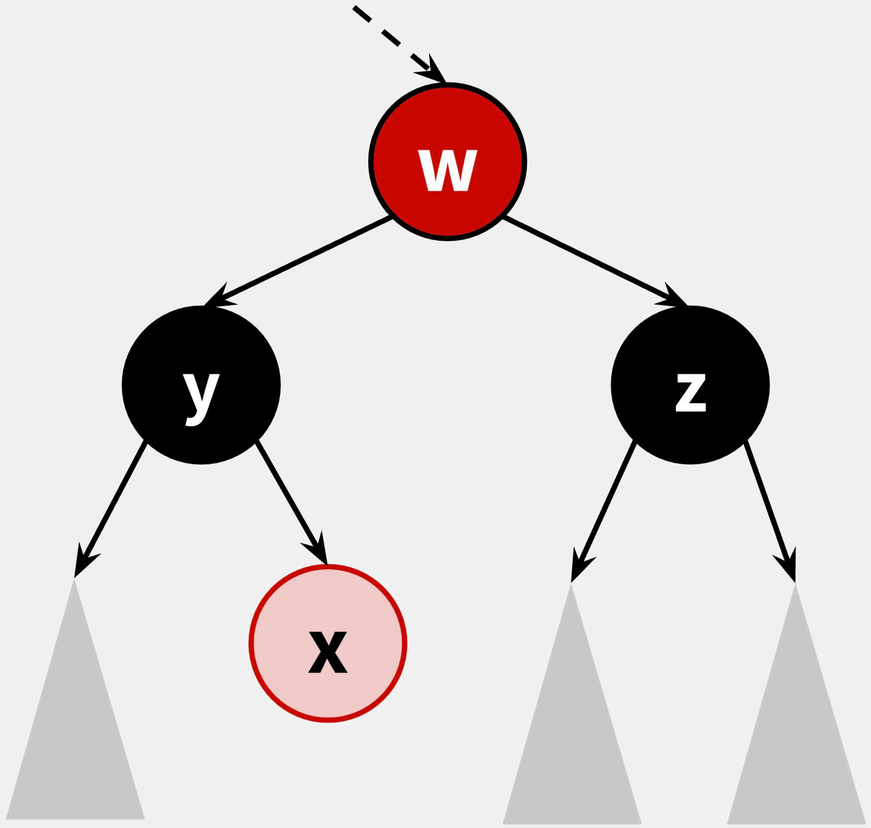
插入后调整: CASE 1

CASE 1: 父结点 **y** 为 **红色**, 且 “叔结点” **z** 也为 **红色**

重新着色

将 **w** 修改为 **红色** &
将 **y** 和 **z** 修改为 **黑色**

用时 O(1)



海量资源：666java.com

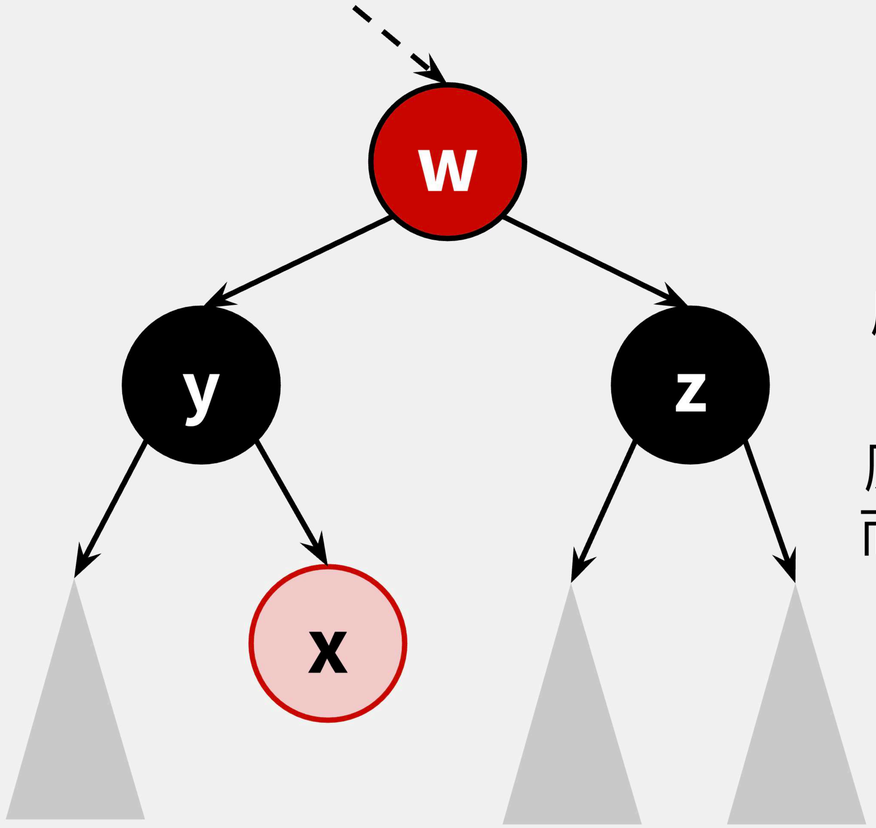
插入后调整: CASE 1

CASE 1: 父结点 **y** 为 **红色**, 且 “叔结点” **z** 也为 **红色**

重新着色

将 **w** 修改为 **红色** &
将 **y** 和 **z** 修改为 **黑色**

用时 O(1)



性质 4 依然得以保持

所有包含 **w**, **y**, **z** 的 root-NIL 路径都会经过 **w**.

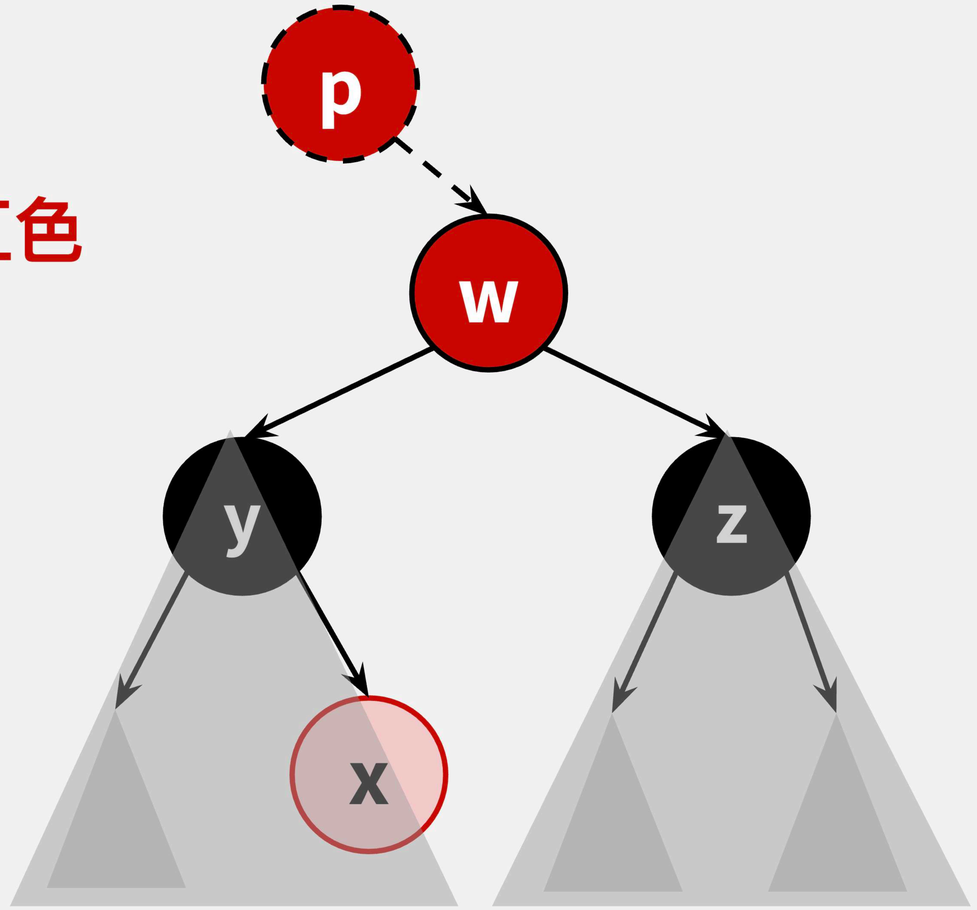
原本这里贡献 1 个 **黑色** 结点 (**w**), 而现在依然仅贡献 1 个 **黑色** 结点 (**y** 或 **z**).

海量资源：666java.com

插入后调整: CASE 1

CASE 1: 父结点 **y** 为 **红色**, 且 “叔结点” **z** 也为 **红色**

如果 **w** 的父结点为 **红色** 呢?

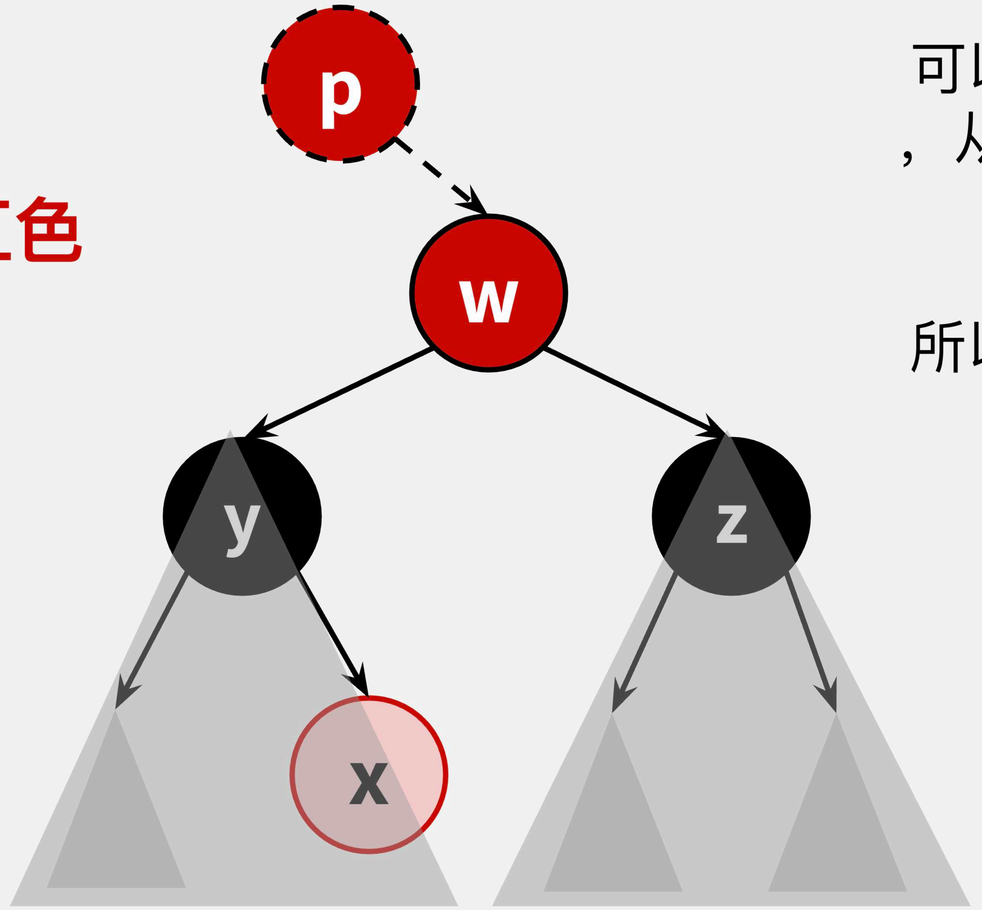


海量资源：666java.com

插入后调整: CASE 1

CASE 1: 父结点 **y** 为 **红色**, 且 “叔结点” **z** 也为 **红色**

如果 **w** 的父结点为 **红色** 呢?



可以将 **w** 看成“新插入”的结点，从而产生递归调用. 这种情况只会发生 O(log n) 次

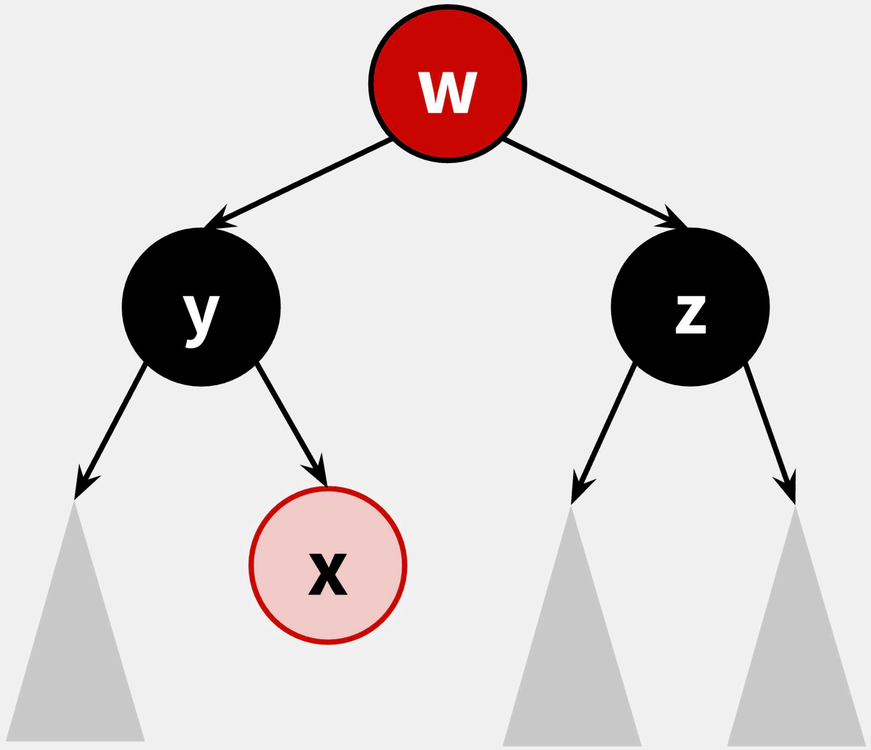
所以这种情况下的插入操作耗时 O(log n)

海量资源：666java.com

插入后调整: CASE 1

CASE 1: 父结点 **y** 为 **红色**, 且 “叔结点” **z** 也为 **红色**

那如果 **w** 是根结点呢?
根结点不能为 **红色**!

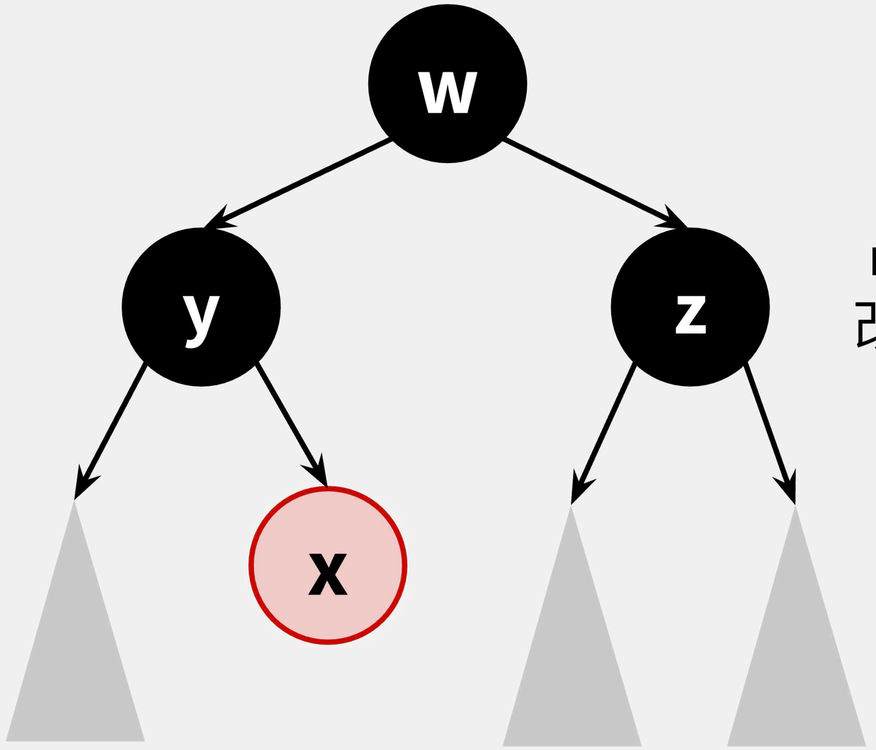


海量资源：666java.com

插入后调整: CASE 1

CASE 1: 父结点 **y** 为 **红色**, 且 “叔结点” **z** 也为 **红色**

那如果 **w** 是根结点呢?
根结点不能为 **红色**!



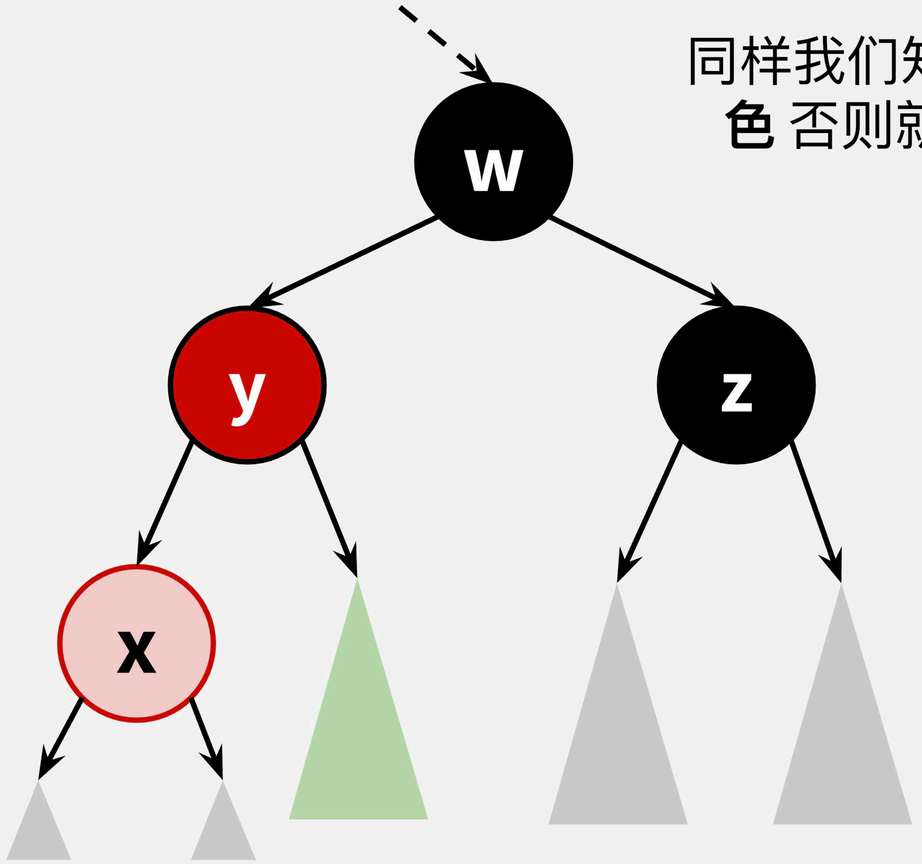
直接将根结点标成 **黑色**

如果 **w** 是根结点, 则 **w** 在每条 root-NIL 路径上均出现. 所以修改 **w** 为 **黑色** 仅在每条路径上新添加了一个黑色结点
性质 4 依然成立

海量资源：666java.com

插入后调整: CASE 2

CASE 2: 父结点 **y** 为 **红色**, 且 “叔结点” **z** 为 **黑** (或 NIL)



同样我们知道 **w** (x 的祖父结点) 一定是 **黑色** 否则就会出现两个连续的 **红色** 结点

海量资源：666java.com

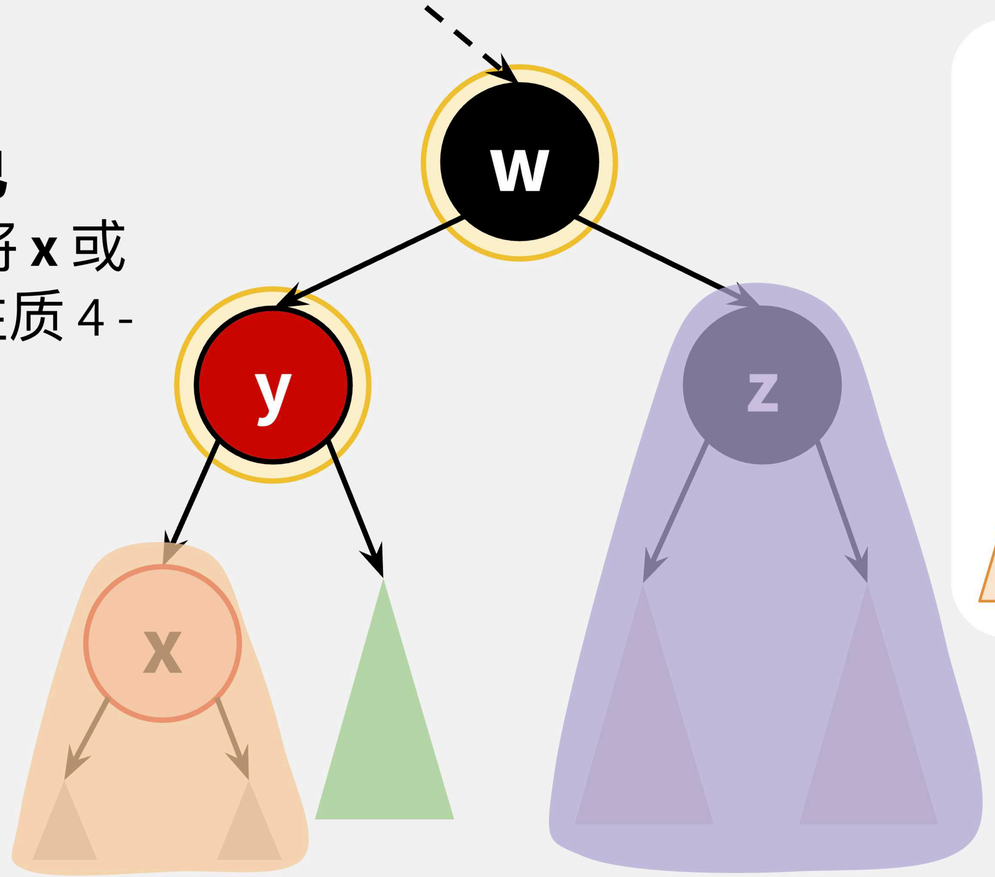
插入后调整: CASE 2

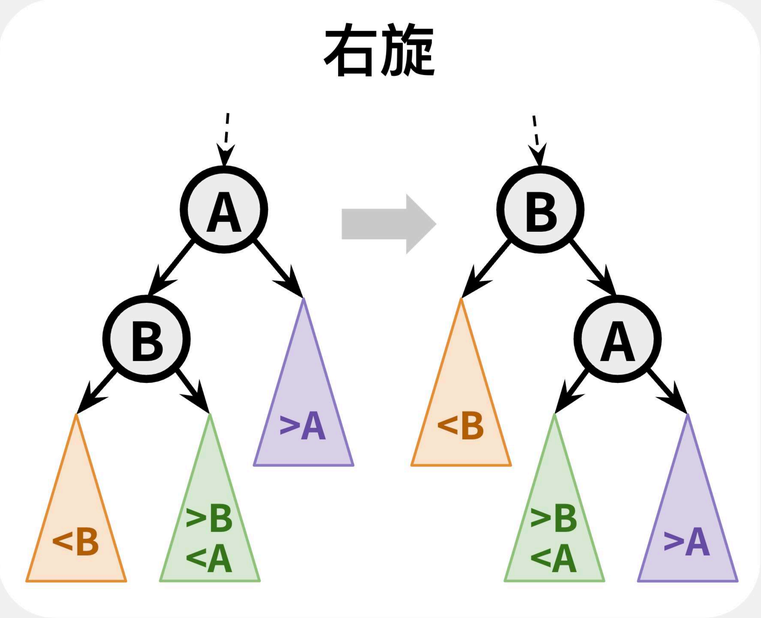
CASE 2: 父结点 **y** 为 **红色**, 且 “叔结点” **z** 为 **黑** (或 NIL)

旋转 & 重新着色

首先需要旋转, 不能直接将 **x** 或 **y** 标为 **黑色**! 那样会破坏性质 4 - 为什么?

旋转 + 重新着色 O(1)





海量资源：666java.com

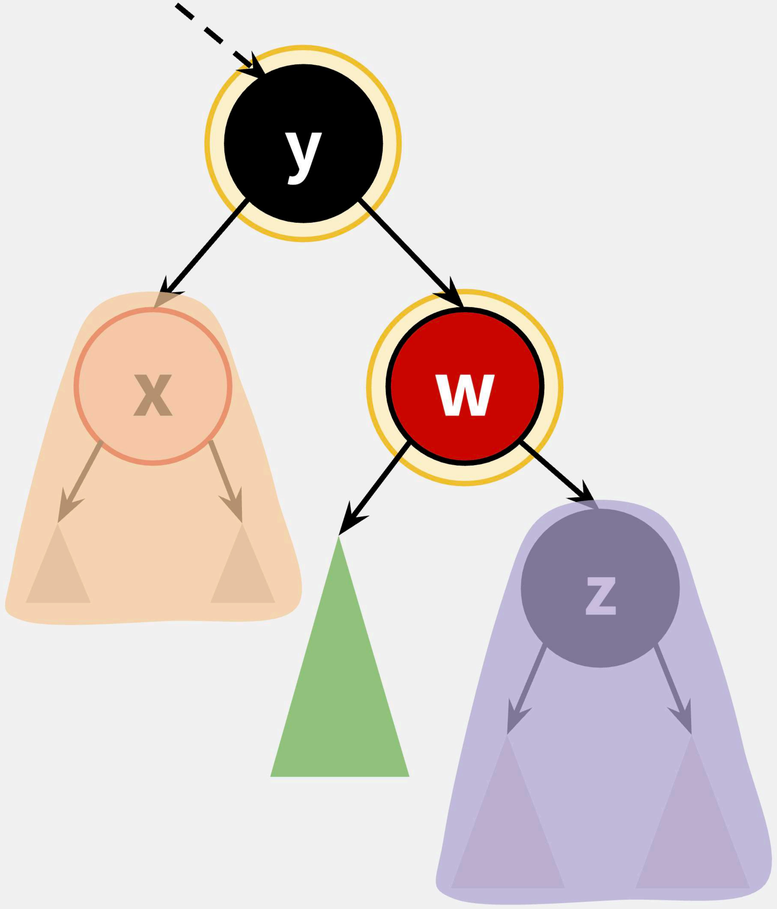
插入后调整: CASE 2

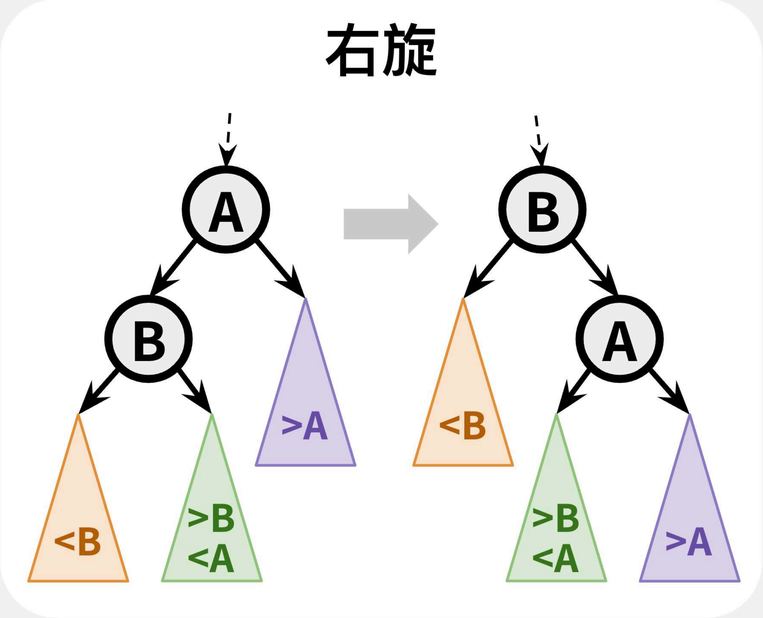
CASE 2: 父结点 **y** 为 **红色**, 且 “叔结点” **z** 为 **黑** (或 NIL)

旋转 & 重新着色

首先需要旋转, 不能直接将 **x** 或 **y** 标为 **黑色**! 那样会破坏性质 4 - 为什么?

旋转 + 重新着色 O(1)





海量资源：666java.com

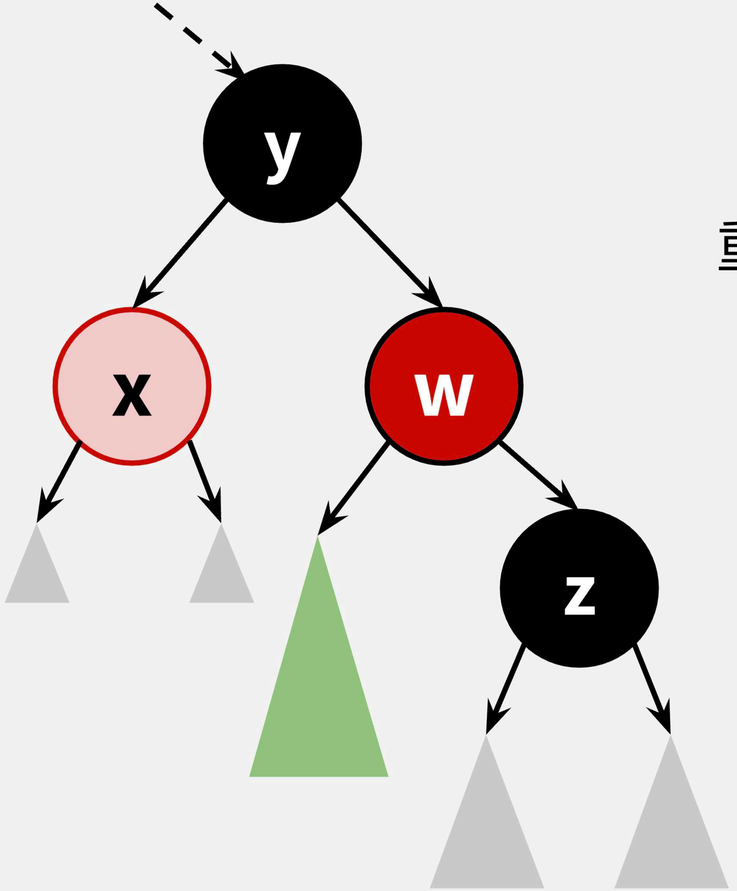
插入后调整: CASE 2

CASE 2: 父结点 **y** 为 **红色**, 且 “叔结点” **z** 为 **黑** (或 NIL)

旋转 & 重新着色

首先需要旋转, 不能直接将 **x** 或 **y** 标为 **黑色**! 那样会破坏性质 4 - 为什么?

旋转 + 重新着色 O(1)



重新着色后, 性质 4 得以维持, 并且不存在需要递归调用的情况
插入与调整完成

海量资源：666java.com

插入操作

算法思路

像二叉搜索树那样插入新结点, 再进行调整

调整 = 为了维护红黑树的性质进行结点颜色的修改及结点旋转

不建议花过多的时间来记忆红黑树的插入和删除操作

但是你应该了解并理解

- 红黑树的性质
- 为什么这些性质能保持红黑树是平衡的?

海量资源：666java.com

红黑树总结

红黑树 上 查找, 插入, 删除

的运行时间为 **O(log n)**

因为红黑树的高度一定不超过 2·log(n+1)

实际应用中, 如果需要用到 BST 来解决现实中的问题, 你应该考虑使用
自平衡的 BST 例如 Red-Black Trees
非平衡的 BST 最坏运行时间可以达到 O(n)

海量资源：666java.com

红黑树总结

操作	排序数组	未排序链表	BST (最坏情况)	BST (平衡)
SEARCH	O(log(n))	O(n)	O(n)	O(log(n))
DELETE	O(n)	O(n)	O(n)	O(log(n))
INSERT	O(n)	O(1)	O(n)	O(log(n))

海量资源：666java.com

总结

- 学习了 **二叉搜索树**
- BST 有时不是平衡的, 非平衡时运行时间会很慢 (O(n)) … 所以我们介绍了 **平衡二叉树**
  - 通过 BST 中的 *旋转* 来实现平衡
  - 平衡二叉树的一个例子是 **红黑树**

海量资源：666java.com

接下来

- 哈希 (Hashing)