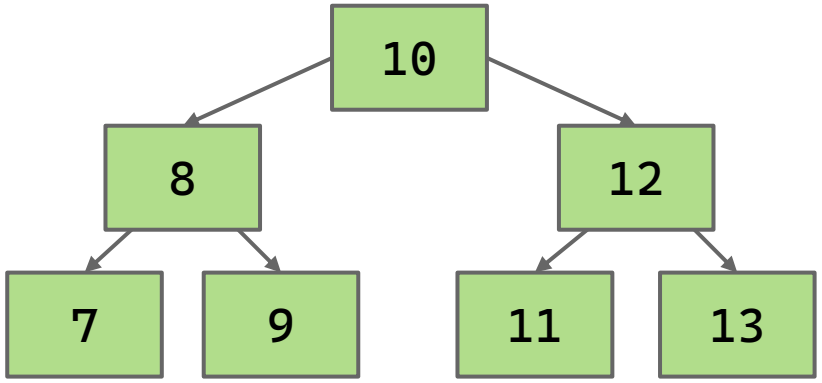
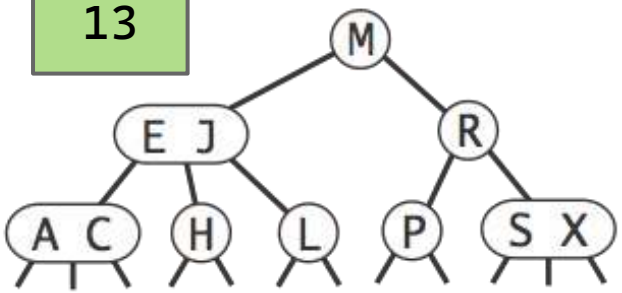
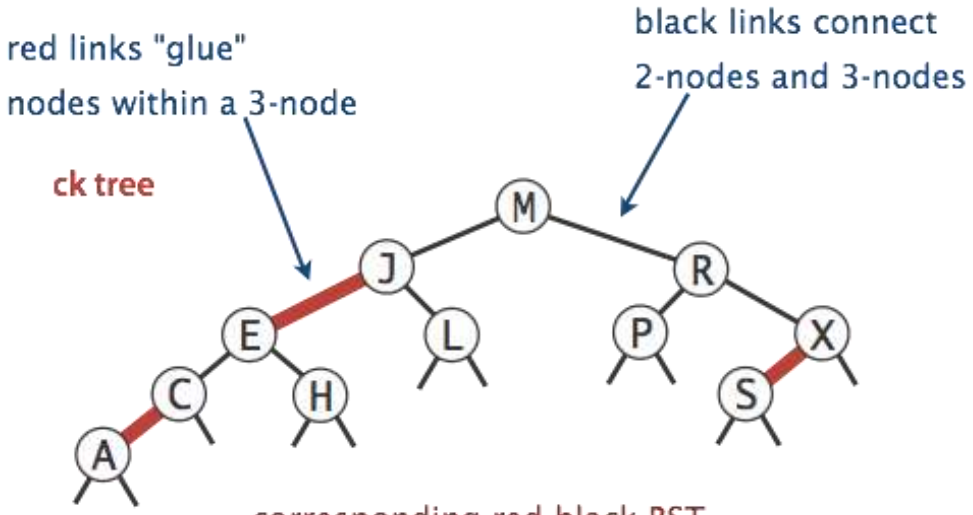


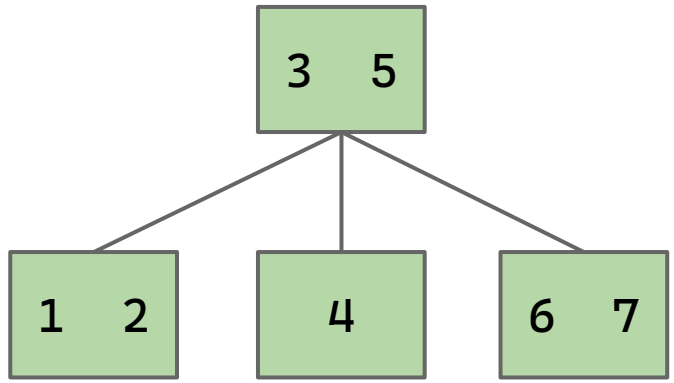


2-3 tree



corresponding red-black BST

BSTs, B-trees, AVL trees, Red-black trees



主讲人：七海Nana7mi

课程大纲：CS61B

课程说明：

- 课程内容基于 UC-Berkeley 的课程 CS61B-sp18 与 CS61B-fa23。可以理解为课程的汉化视频。
- 课程使用的编程语言为 Java。
- AI 语音模型来源 BiliBili 用户 Xz乔希。
- 七海也在学习中，有错误敬请指出！

- 第一期视频：【CS61B汉化】七海讲数据结构-二叉搜索树【七海Nana7mi】
- 第二期视频：【CS61B汉化】七海讲数据结构-平衡树&B树【七海Nana7mi】

本节课须知：

- 因为本课程实质为 CS61B 课程翻译。由于 CS61B 课程大纲没有涉及到 AVL树，但是又考虑到国内的教材非常喜欢把 AVL树 讲的看似“又长又难”所以这节课会适当的讲到AVL树，但不会是重点。
- 因为 CS61B 大纲设计，本节课讲的红黑树实际是左偏红黑树，一种简单的红黑树变体。
- 因为 CS61B 大纲设计，课程不会涉及到左偏红黑树删除操作。
- 本视频实质为课程，非科普，具有课程的严肃性。
- **这节课有很多问题环节，大家可以多暂停思考~**
  - **当背景颜色为 蓝色 / 紫色 时，表明这是一个问题环节。**
- 红黑树 AVL树 左偏红黑树 等其它数据结构与算法内容可见 https://oi-wiki.org/

B树太难实现了！

Lecture 3

B树太难实现了！

旋转

- 定义
- 用旋转维持平衡

AVL树

红黑树（LLRBs）

- 来自2-3树
- LLRB 性质
- LLRB 的插入操作
- LLRB 性能与实现

一个坏消息：

对于 B树 来说，它的实现实在是太太太麻烦了：

- 需要维护不同类型的结点
- 结点类型间需要不停的变化

```
1.public void put(Key key, Value val) {
2.   Node x = root;
3.   while (x.getTheCorrectChildKey(key) != null) {
4.      x = x.getTheCorrectChildKey();
5.      if (x.is4Node()) { x.split(); }
6.   }
7.   if (x.is2Node()) { x.make3Node(key, val); }
8.   if (x.is3Node()) { x.make4Node(key, val); }
9.}
```

“Beautiful algorithms are, unfortunately, not always the most useful.” – Knuth

旋转：定义

Lecture 3

B树太难实现了！

旋转

- 定义
- 用旋转维持平衡

AVL树

红黑树（LLRBs）

- 来自2-3树
- LLRB 性质
- LLRB 的插入操作
- LLRB 性能与实现

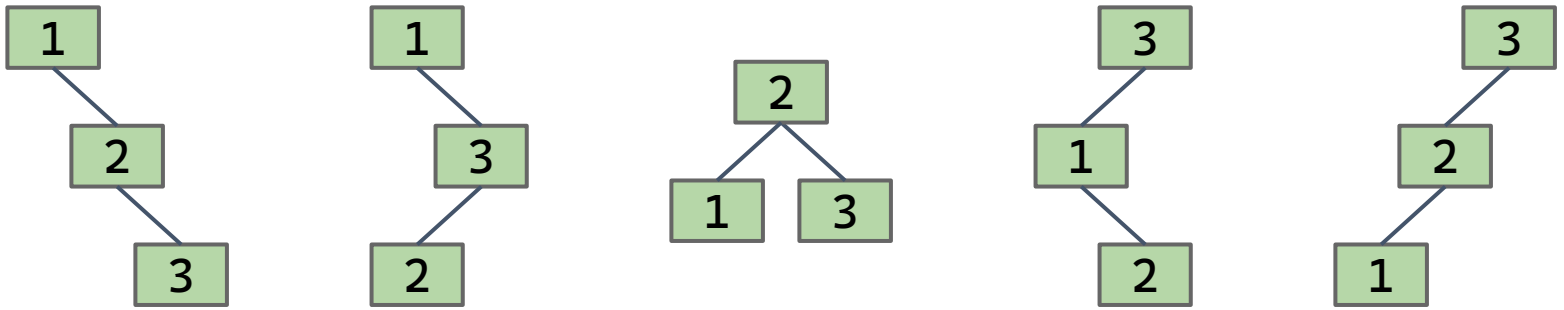
旋转操作

假设我们有一个二叉搜索树，包含三个元素：1，2，3。

根据插入的顺序不同，我们有 ？ 种可能的树的结构：

旋转操作

假设我们有一个二叉搜索树，包含三个元素：1，2，3。
根据插入的顺序不同，我们有 5 种可能的树的结构：

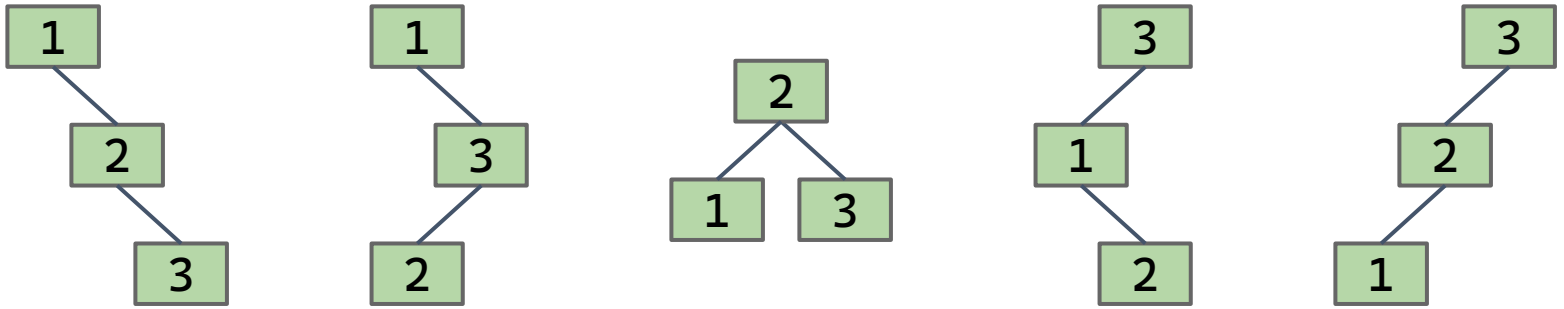


- 更广泛来说，BST 的结点数 N 和树可能的结构数量关系可以用**卡塔兰数列**表示。

旋转操作

假设我们有一个二叉搜索树，包含三个元素：1，2，3。

根据插入的顺序不同，我们有 5 种可能的树的结构：



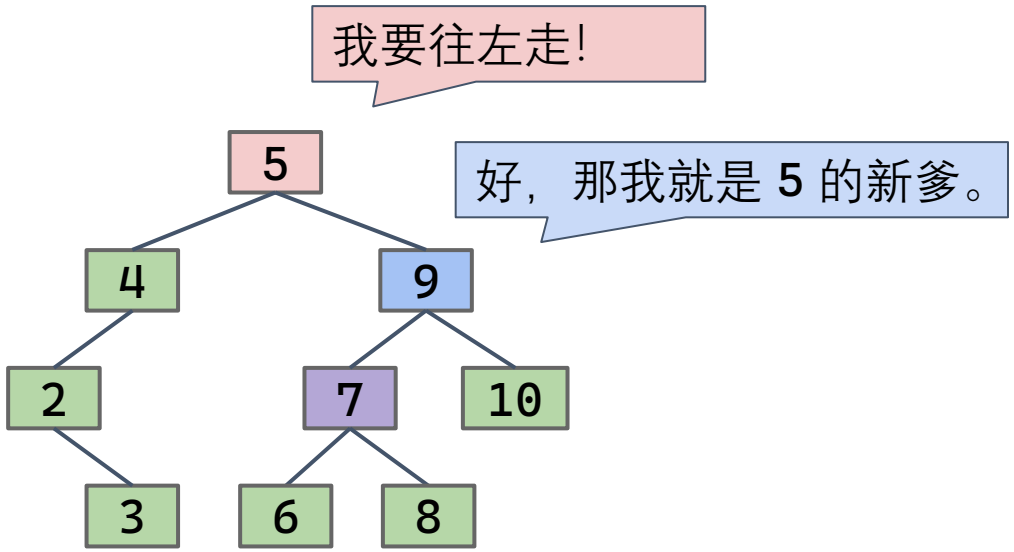
- 更广泛来说，BST 的结点数 N 和树可能的结构数量关系可以用**卡塔兰数列**表示。

已有一个 BST，我们可以用一系列不同的旋转操作让他成为另一种 BST 结构。

- 有论文证明有 n 个结点的不同形态的 BST 间转换可在 2n−6 个旋转操作内完成。（资料在简介）

旋转操作 – 旋转后仍是 BST

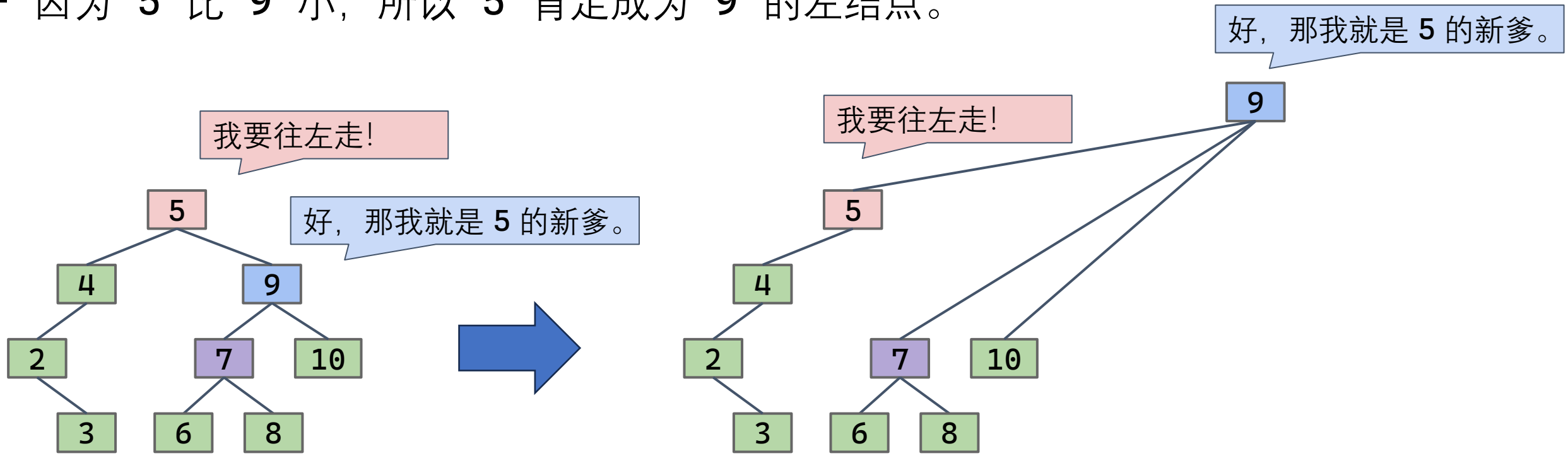
`rotateLeft(5)`：让 `5` 的**右结点**成为 `5` 的父结点



旋转操作 – 旋转后仍是 BST

`rotateLeft(5)`：让 5 的**右结点**成为 5 的父结点

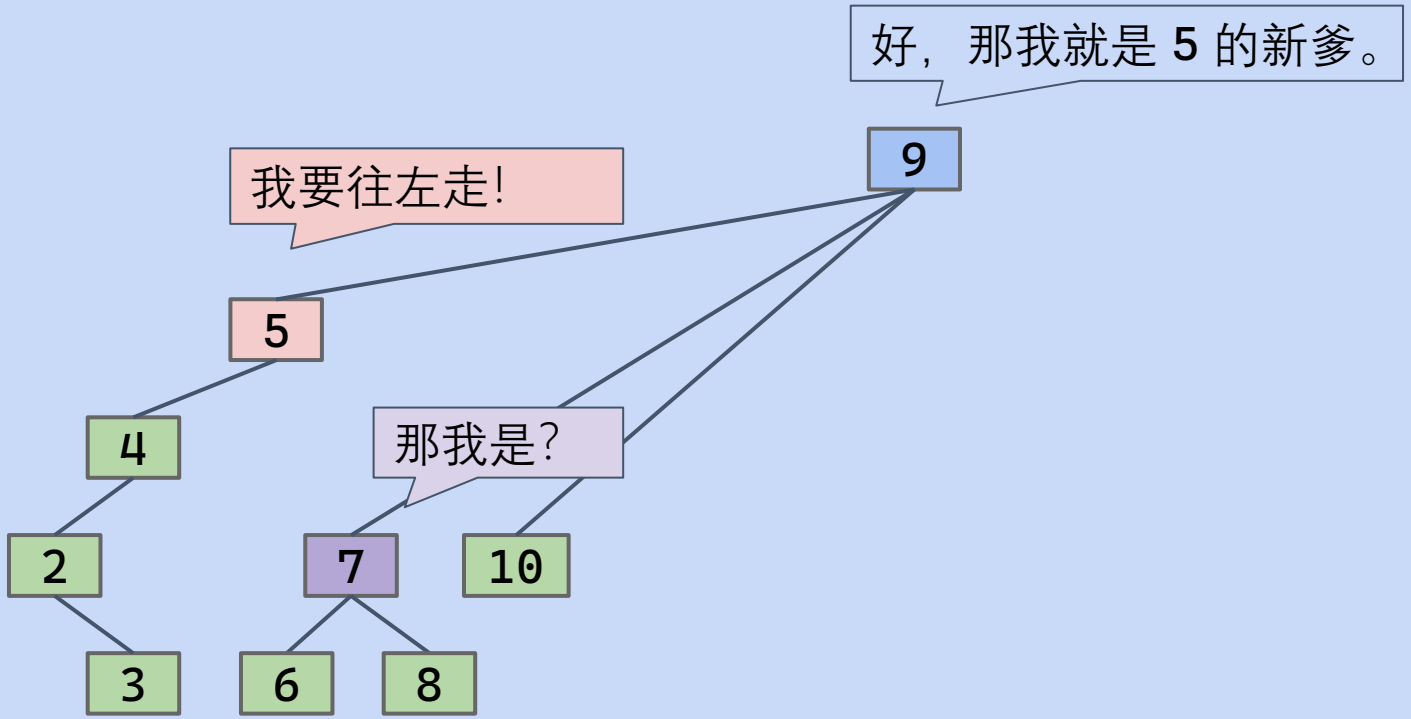
– 因为 5 比 9 小，所以 5 肯定成为 9 的左结点。

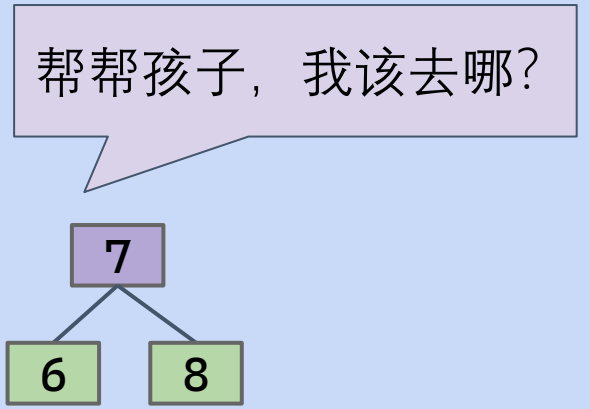


旋转操作 – 旋转后仍是 BST

`rotateLeft(5)`：让 5 的**右结点**成为 5 的父结点

– 因为 5 比 9 小，所以 5 肯定成为 9 的左结点。

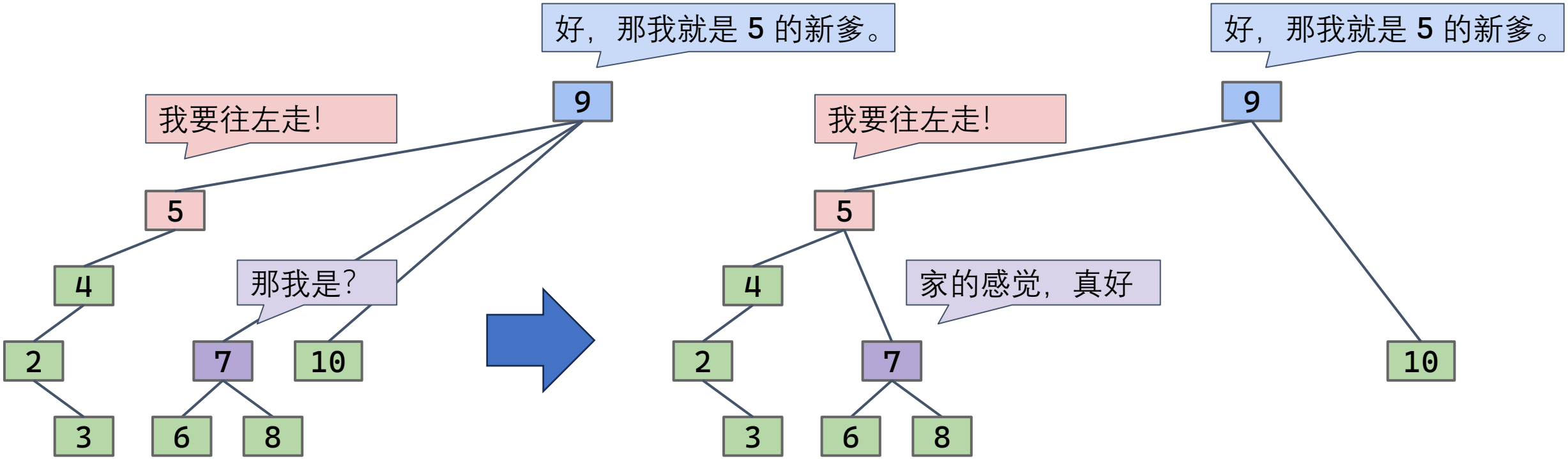




旋转操作 – 旋转后仍是 BST

`rotateLeft(5)`：让 5 的**右结点**成为 5 的父结点

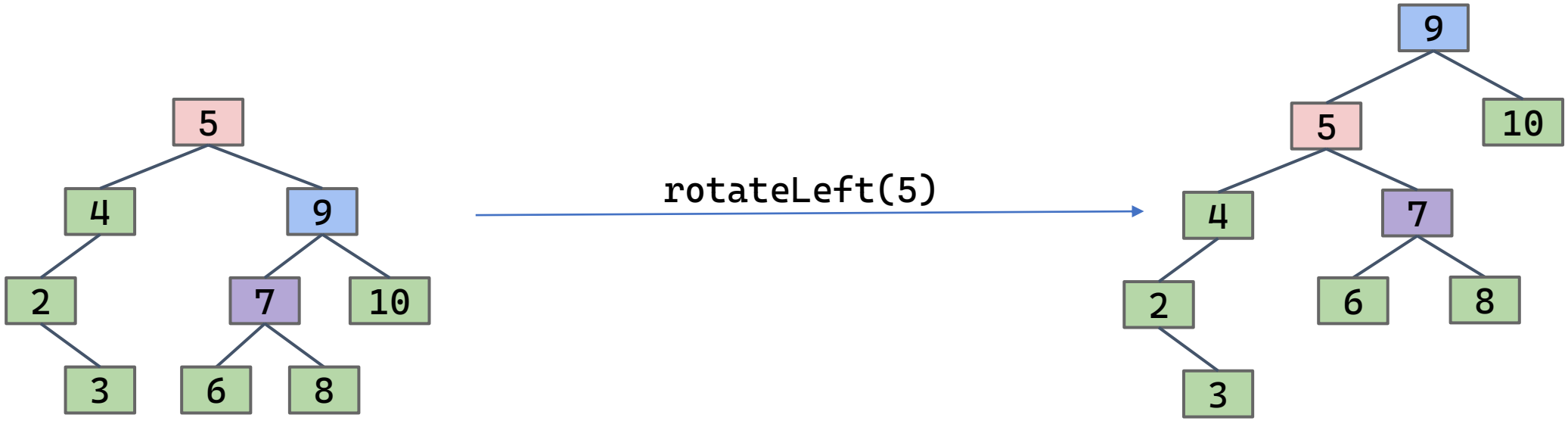
- 因为 5 比 9 小，所以 5 肯定成为 9 的左结点。
- 为了维持旋转后仍是 BST 的事实，原来 9 的左结点变成了 5 的右结点。



旋转操作 – 旋转后仍是 BST

`rotateLeft(5)`：让 `5` 的**右结点**成为 `5` 的父结点

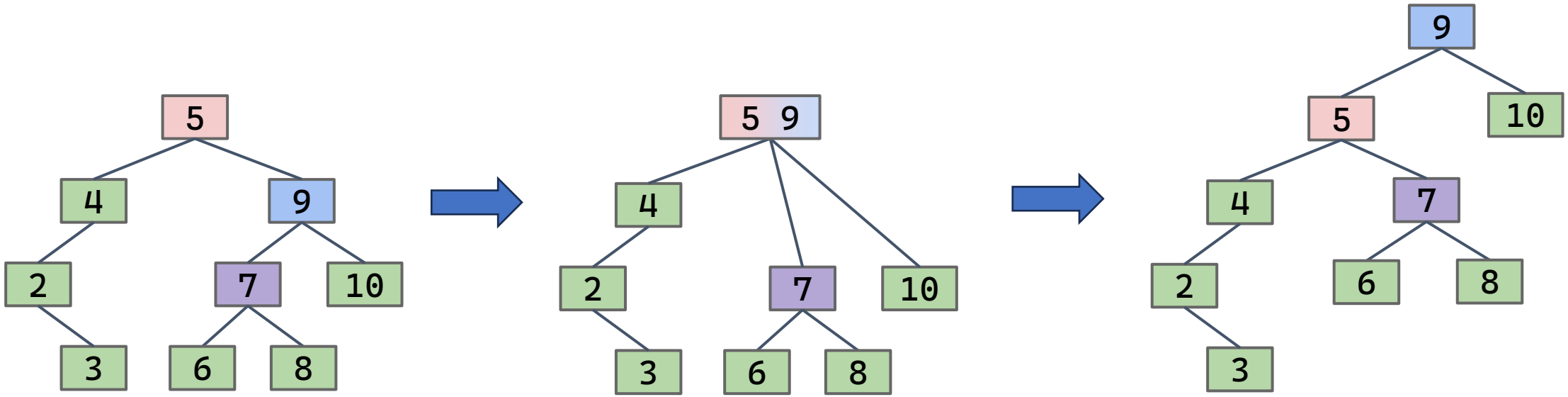
- 我们发现，这个旋转操作，增加了树的高度（`H += 1`）。
- 旋转后，新的树仍然是二叉搜索树。



旋转操作 – 另一种理解方法

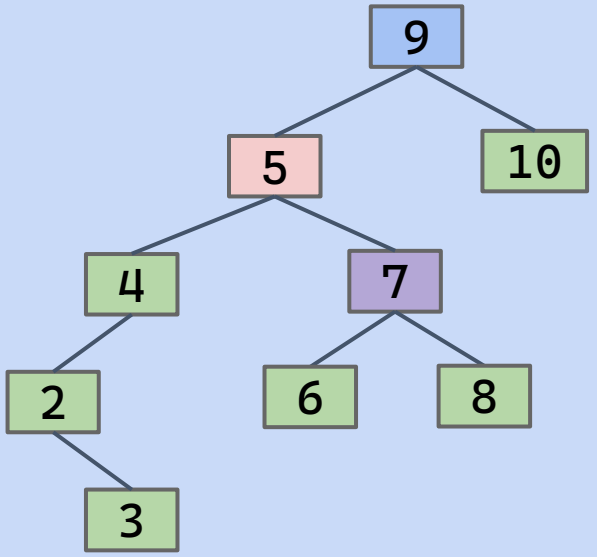
`rotateLeft(5)`：让 `5` 的**右结点**成为 `5` 的父结点

- 可以理解为：我们先合并了 `5` 和它的右结点 `9`。
- 然后再将 `5` 分裂下去。



旋转操作 – 练习

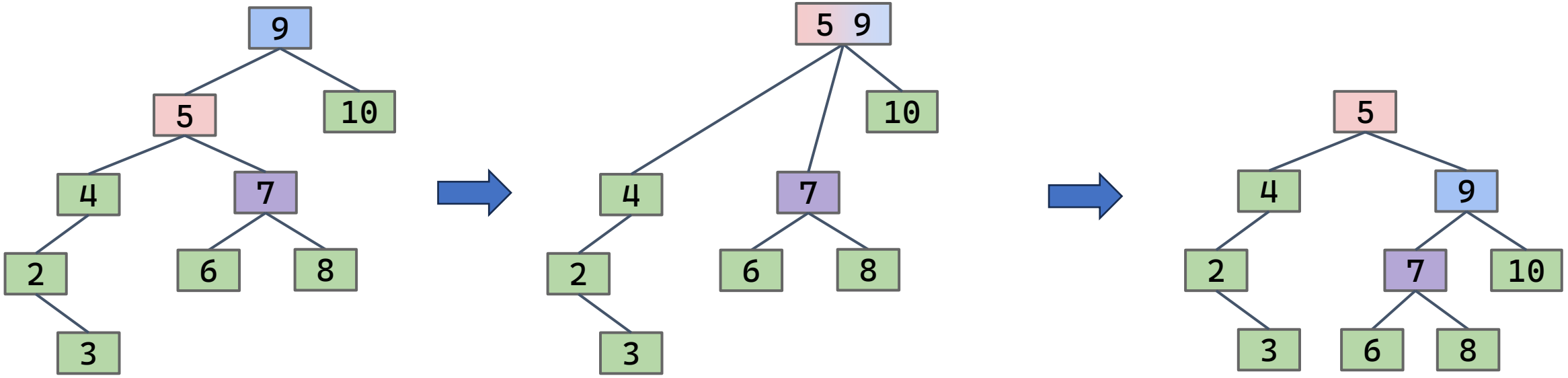
`rotateRight(9)`：让 9 的**左结点**成为 9 的父结点



旋转操作 – 练习

`rotateRight(9)`：让 `9` 的**左结点**成为 `9` 的父结点

–我们发现，这个旋转操作，缩小了树的高度（`H -= 1`）。



旋转：用旋转维持平衡

Lecture 3

B树太难实现了！

旋转

- 定义
- 用旋转维持平衡

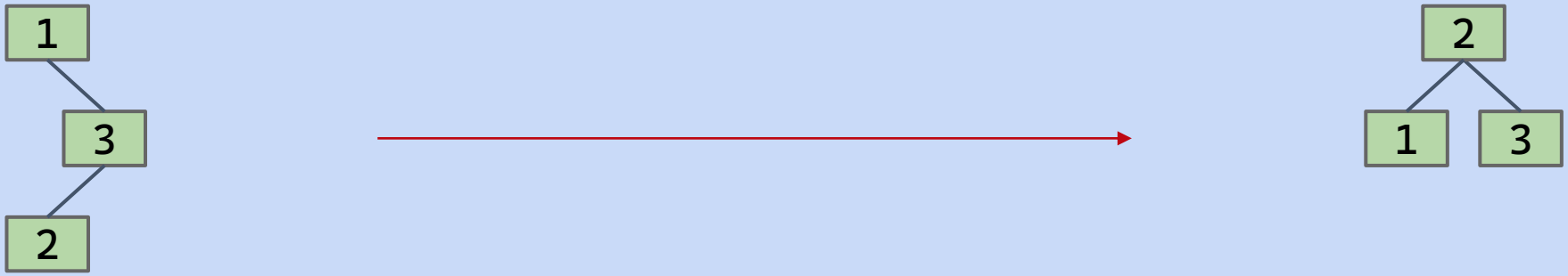
AVL树

红黑树（LLRBs）

- 来自2-3树
- LLRB 性质
- LLRB 的插入操作
- LLRB 性能与实现

用旋转维持平衡

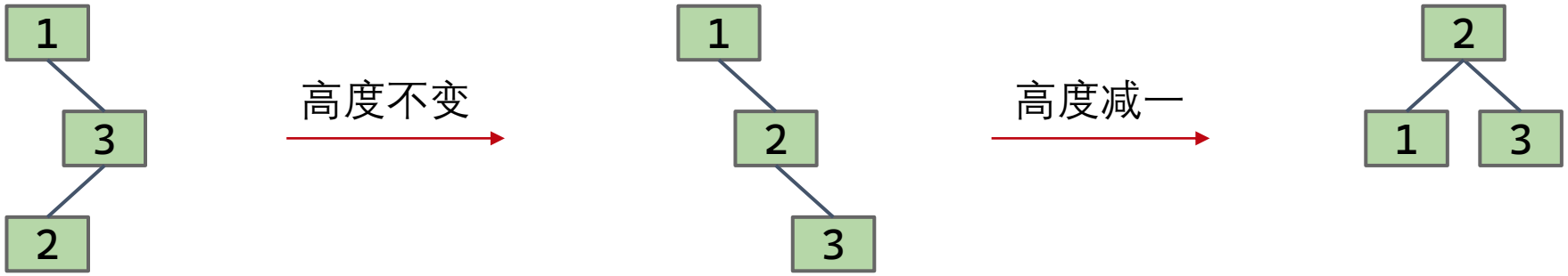
给出你认为对的一系列（对某结点左右）旋转操作，使下面这棵树（左图）平衡（像右图那样）。



用旋转维持平衡

给出你认为对的一系列（对某结点左右）旋转操作，使下面这棵树（左图）平衡（像右图那样）。

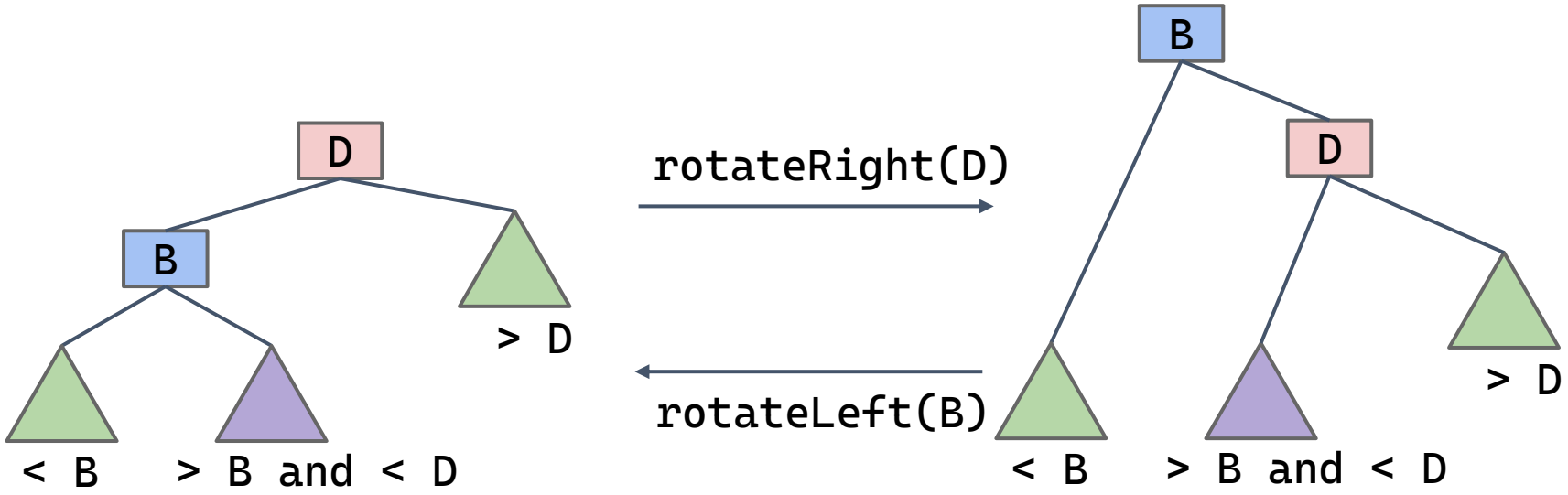
- 1. `rotateRight(3)`
- 2. `rotateLeft(1)`



用旋转维持平衡

旋转操作:

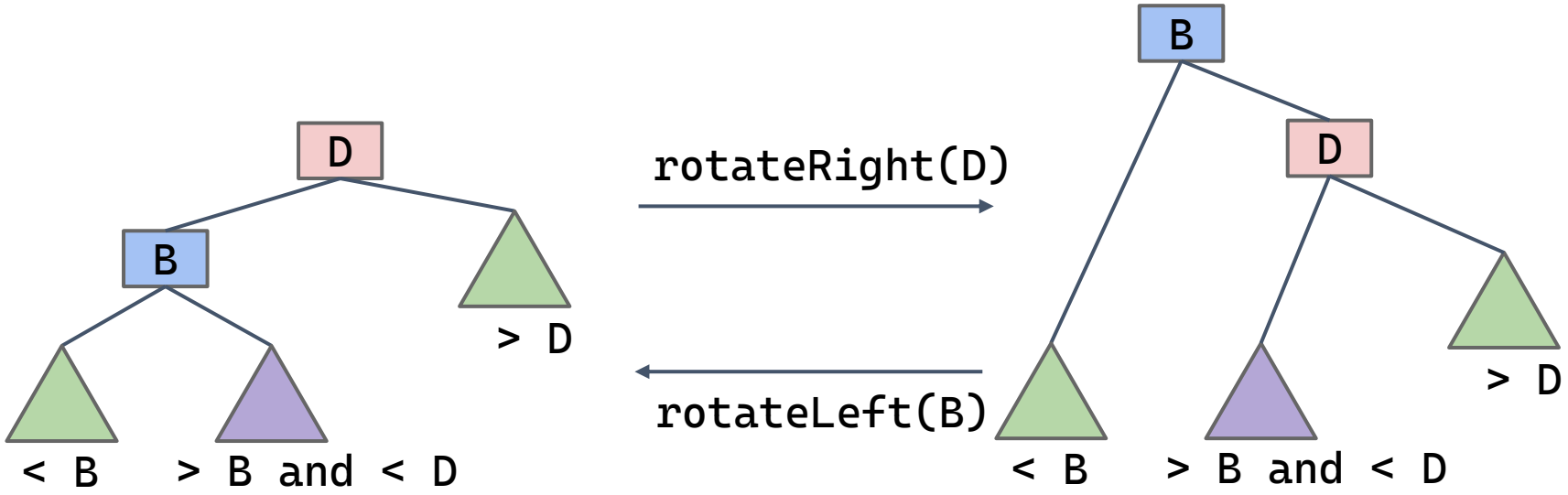
- 可以高度不变，可以降低高度，可以增加高度。
- 维持二叉搜索树的特征，旋转后仍是二叉搜索树。



用旋转维持平衡

旋转操作:

- 可以高度不变，可以降低高度，可以增加高度。
- 维持二叉搜索树的特征，旋转后仍是二叉搜索树。



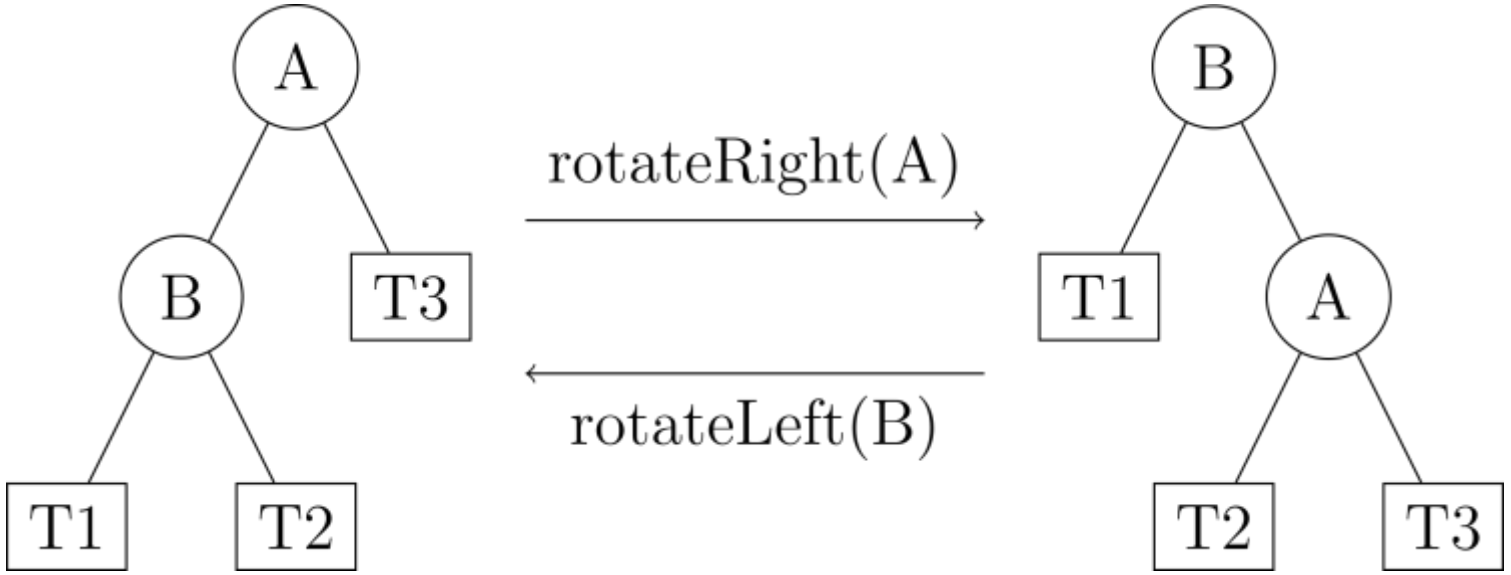
我们可以用旋转操作，在 O(N) 的时间复杂度下平衡一颗二叉搜索树

旋转操作代码实现

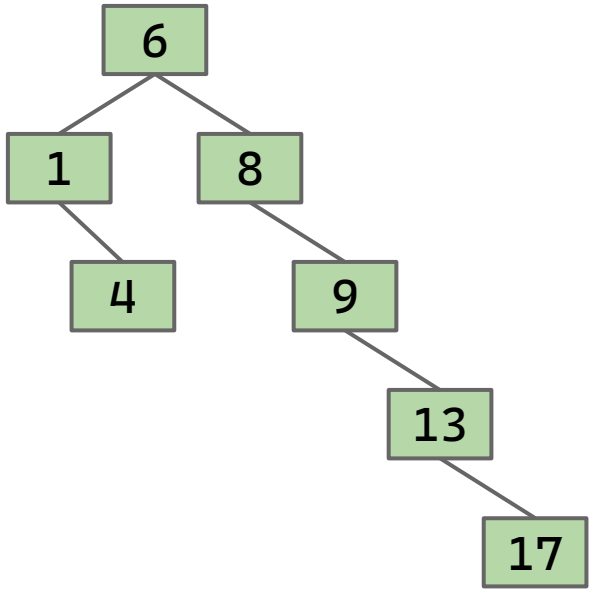
旋转操作只改变了三组结点关联，相当于对三组边进行循环置换一下，因此需要暂存一个结点再进行轮换更新。

对于右旋操作一般的更新顺序是：暂存 B 结点（新的根节点），让 A 的左孩子指向 B 的右子树 T2，再让 B 的右孩子指针指向 A，最后让 A 的父结点指向暂存的 B。

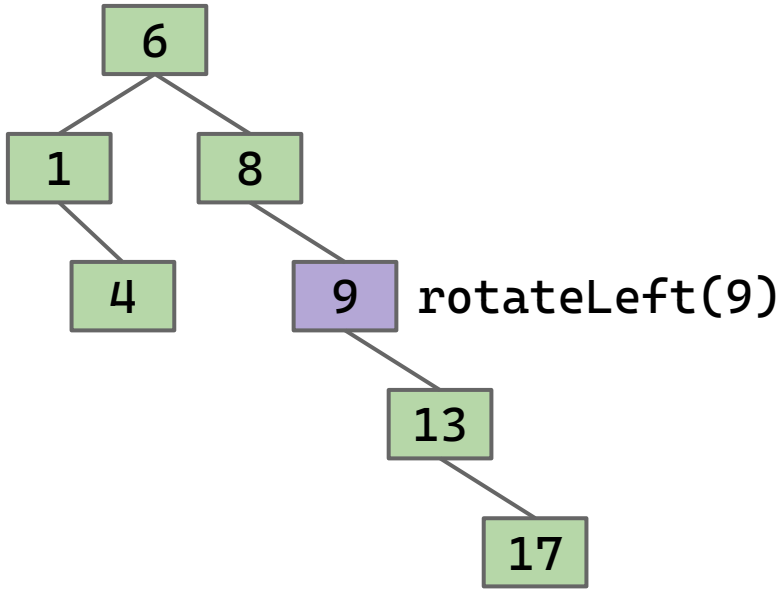
详细代码见 https://oiwiki.org/ds/bst/



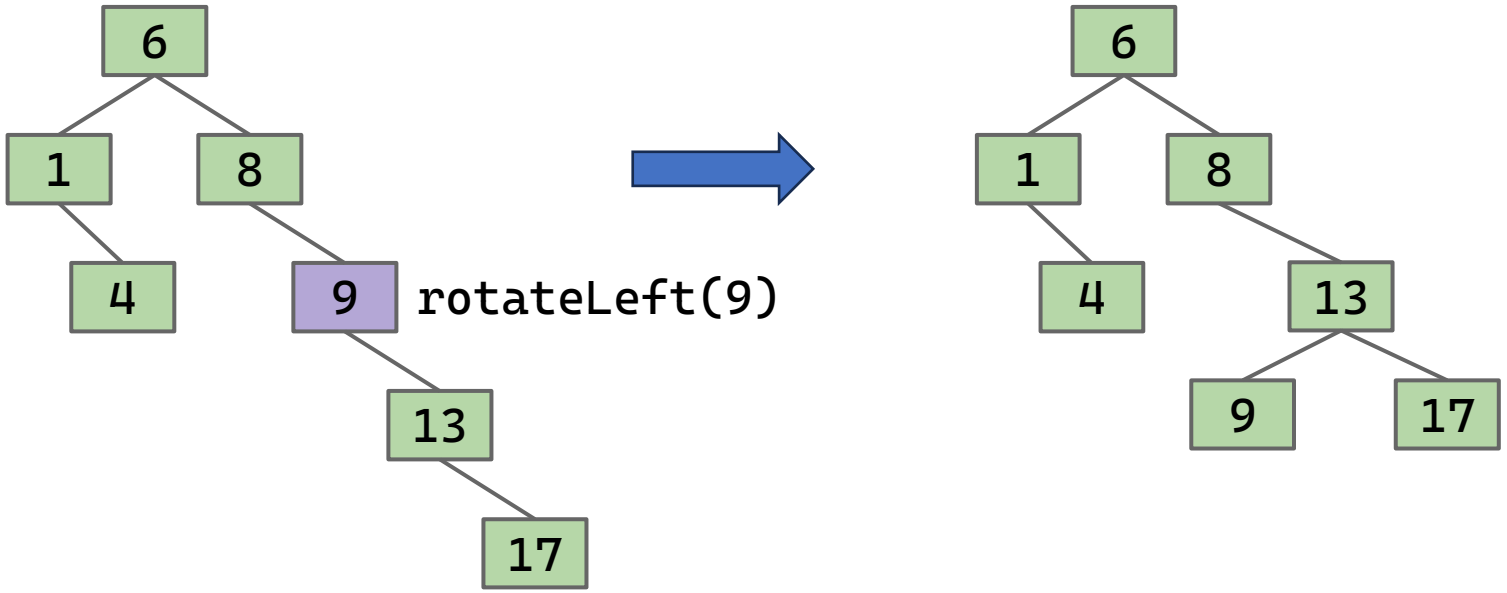
用旋转维持平衡



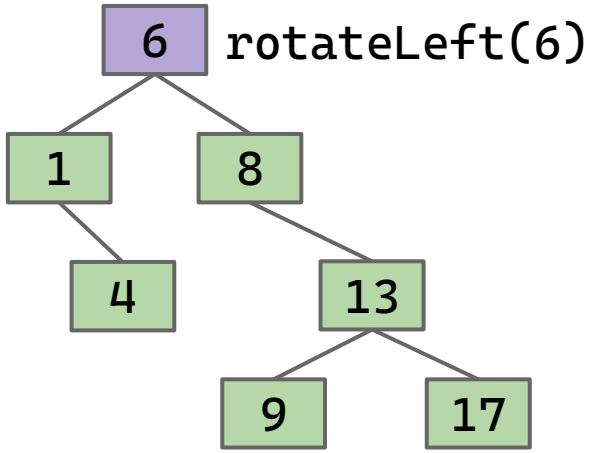
用旋转维持平衡



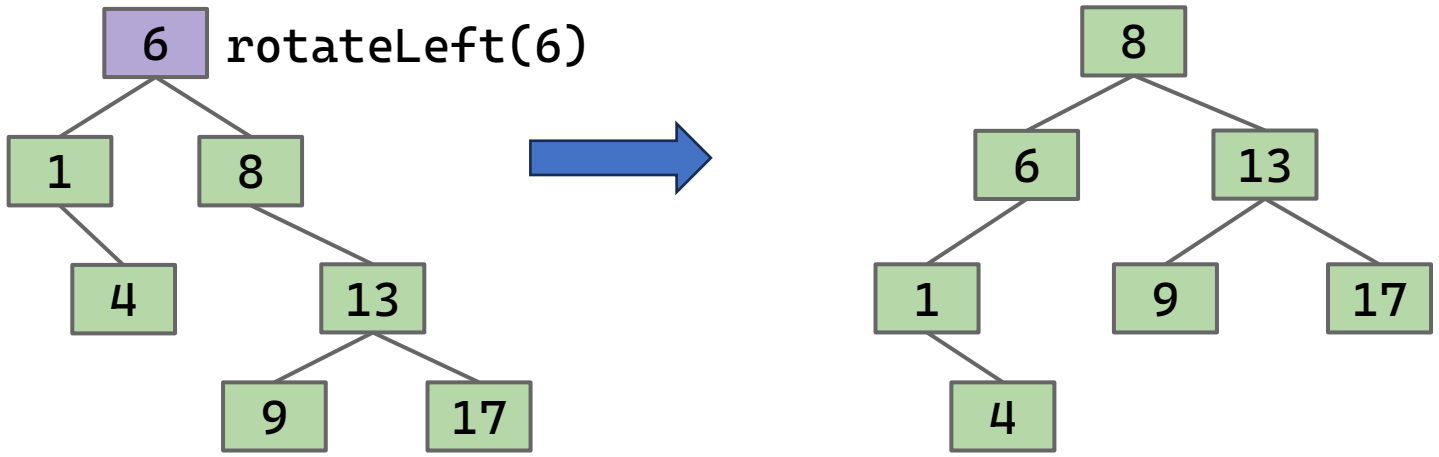
用旋转维持平衡



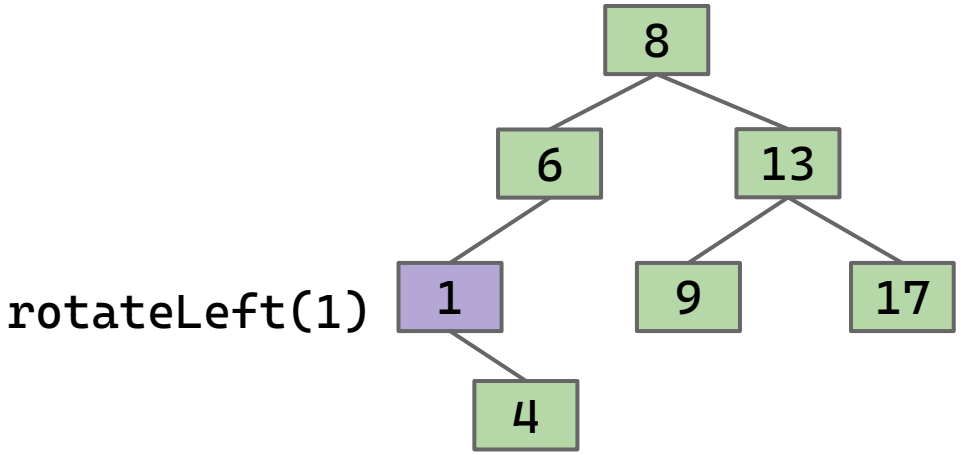
用旋转维持平衡



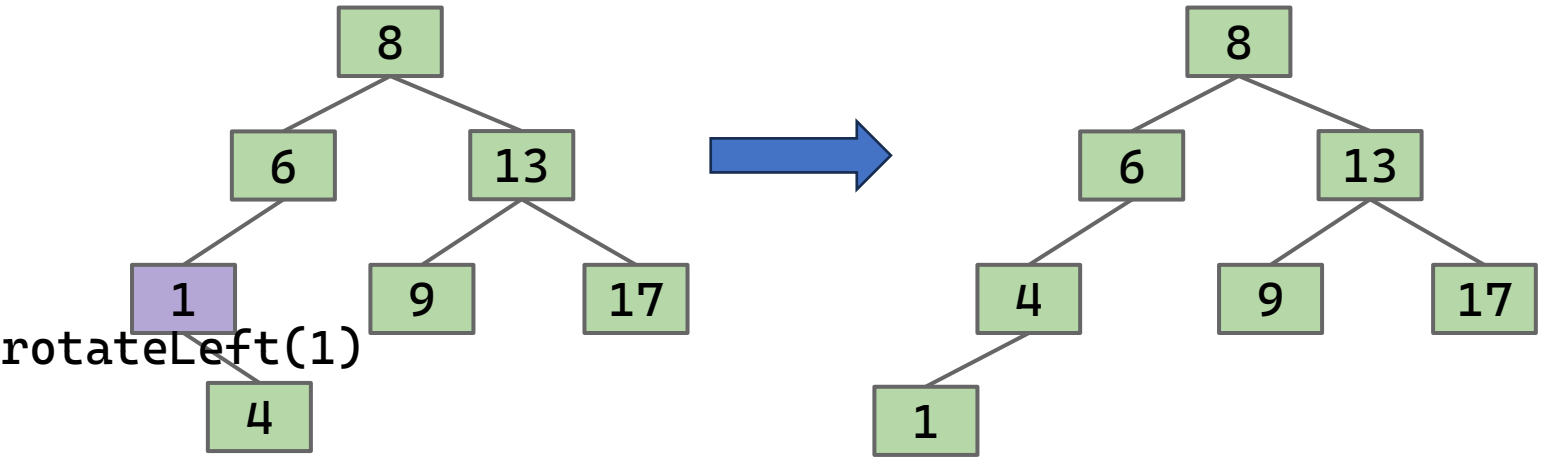
用旋转维持平衡



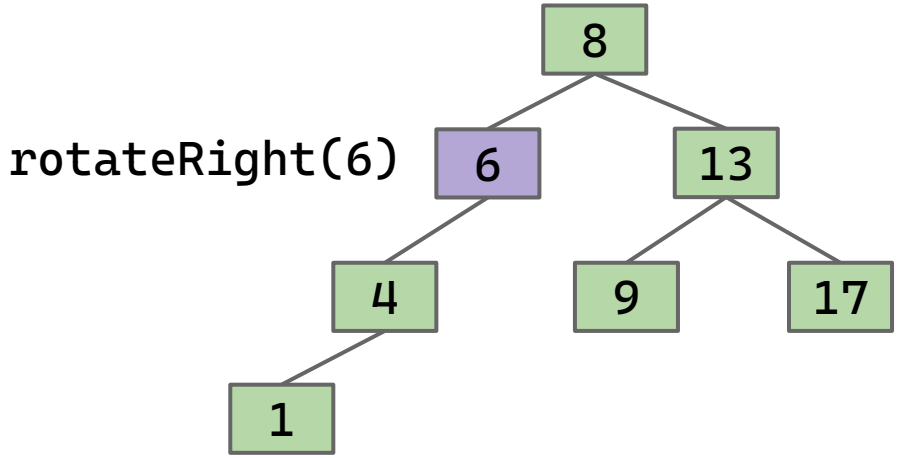
用旋转维持平衡



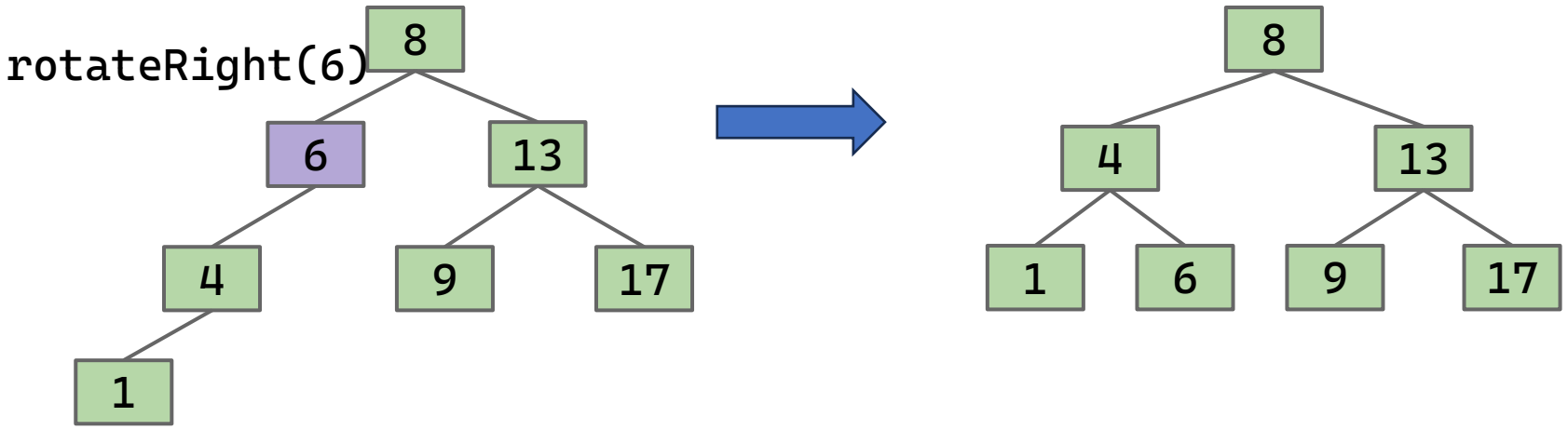
用旋转维持平衡



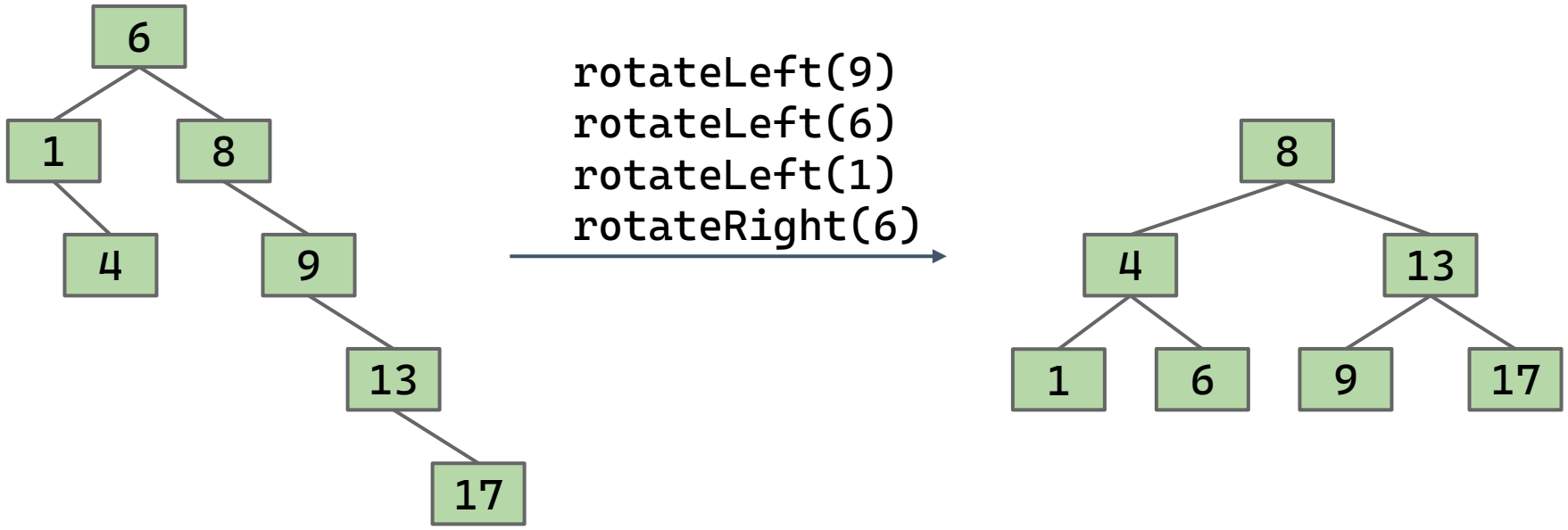
用旋转维持平衡



用旋转维持平衡



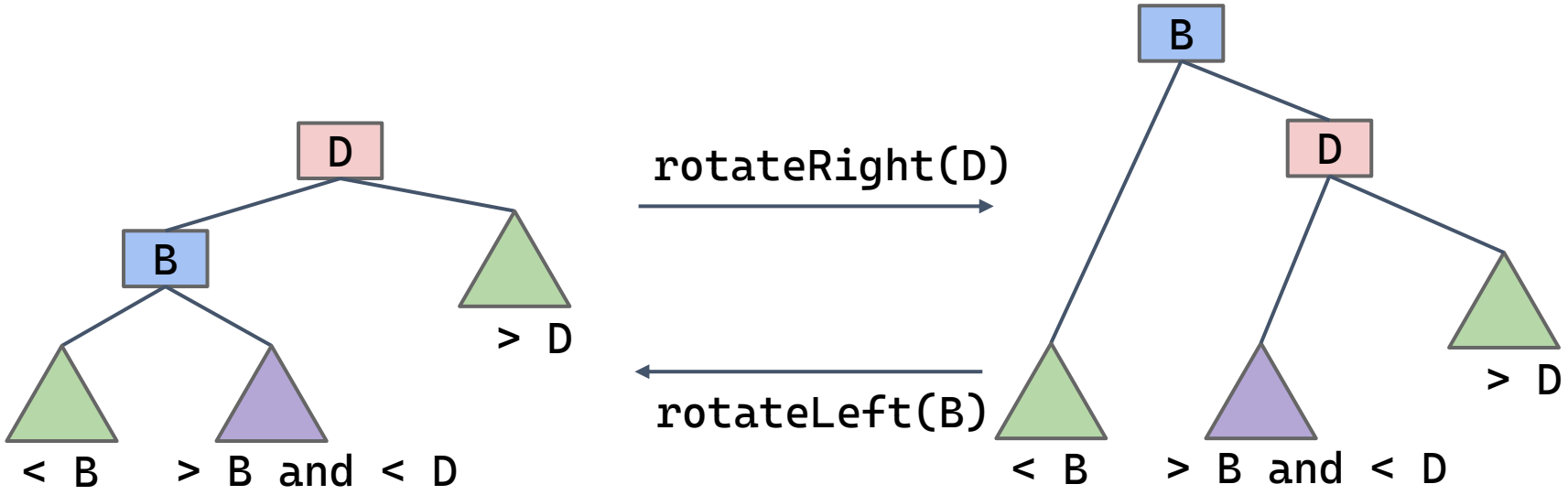
用旋转维持平衡


6
1
8
4
9
13
17
rotateLeft(9)
rotateLeft(6)
rotateLeft(1)
rotateRight(6)
8
4
13
1
6
9
17

用旋转维持平衡

旋转操作:

- 可以高度不变，可以降低高度，可以增加高度。
- 维持二叉搜索树的特征，旋转后仍是二叉搜索树。



已有一颗二叉搜索树，我们可以在 O(N) 的时间下平衡它。但我们更希望的是“自平衡”，即在插入或者删除的过程中，树自己会保持平衡...

AVL树

Lecture 3

B树太难实现了！

旋转

- 定义
- 用旋转维持平衡

AVL树

红黑树（LLRBs）

- 来自2-3树
- LLRB 性质
- LLRB 的插入操作
- LLRB 性能与实现

AVL树

AVL树，是一种自平衡的二叉搜索树。

- 空二叉树是一个 AVL树。
- 如果 T 是一棵 AVL树，那么其左右子树也是 AVL 树，
  并且 |h(ls) - h(rs)| <= 1 ，h 是其左右子树的高度。
- 树高为 O(log N)

AVL树 - 平衡因子

AVL树，是一种自平衡的二叉搜索树。

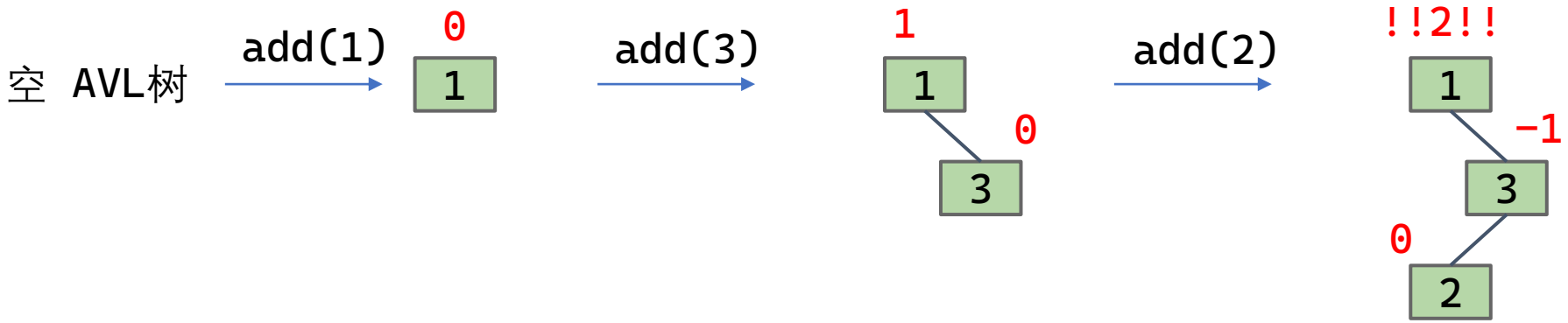
- 空二叉树是一个 AVL树。
- 如果 T 是一棵 AVL树，那么其左右子树也是 AVL 树，
  并且 |h(ls) - h(rs)| <= 1 ，h 是其左右子树的高度。
- 树高为 O(log N)

平衡因子 = 右子树的高度 - 左子树的高度

AVL树 – 平衡性的维持

- 如果 T 是一棵 AVL树，那么其左右子树也是 AVL 树，
  并且 |h(ls) – h(rs)| <= 1 ，h 是其左右子树的高度。
- 平衡因子 = 右子树的高度 – 左子树的高度

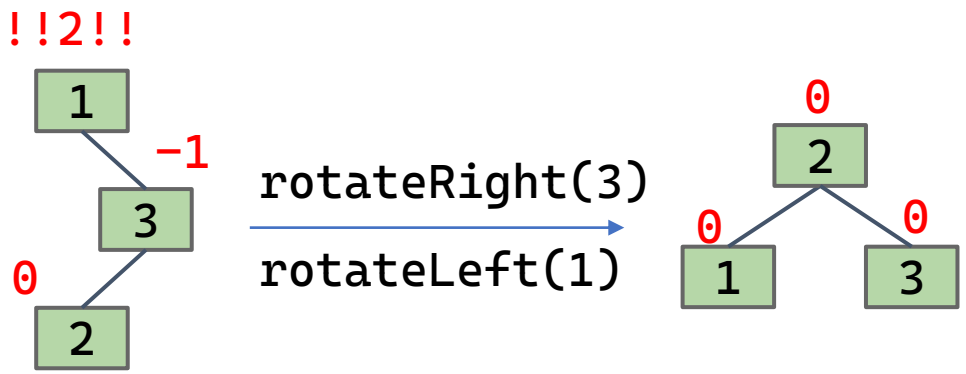
|平衡因子| > 1 -> 旋转!



AVL树 – 平衡性的维持

- 如果 T 是一棵 AVL树，那么其左右子树也是 AVL 树，
  并且 |h(ls) – h(rs)| <= 1 ，h 是其左右子树的高度。
- 平衡因子 = 右子树的高度 – 左子树的高度

|平衡因子| > 1 -> 旋转!



AVL树 – 平衡性的维持

– 插入：与 BST 中类似，先进行一次失败的查找来确定插入的位置，插入节点后根据平衡因子来决定是否需要调整。

– 删除：删除和 BST 类似，将结点与后继交换后再删除。删除会导致树高以及平衡因子变化，这时需要沿着被删除结点到根的路径来调整这种变化。

AVL树动态演示：https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

AVL树

由于 CS61B 没有对 AVL树 进行讲解，只是上课提了一嘴，所以本“汉化”课程对 AVL树 的介绍点到为止，实际已经完全讲到了 AVL树 的核心内容。

- Q: 你都说了 CS61B 没讲，为啥你非要讲?

- A: 因为笔者认为 AVL树 实际并不难，是很清晰易操作的平衡二叉树，只是国内一些教材写的十分冗杂混乱，导致很多同学在初学时感到十分不适，一头雾水。笔者在这里特地讲到 AVL树，就是为了消除大家的“恐惧”。

- Q: 那为啥你讲了这些，又突然停了不讲了?

- A: 因为笔者扪心自问没有能力可以用简洁，有启发性的文本向大家介绍清楚 AVL树 的各种细节。所以笔者止步于此，希望想知道更具体细节的大家可以带着目前在本视频学到的知识看自己课本中的 AVL树，或者其它的 AVL树 介绍视频，相信大家可以克服困难!

AVL树部分参考内容:

- https://oiwiki.org/ds/avl/
- https://www.icourse163.org/course/zju-93001

红黑树：来自2-3树

Lecture 3

B树太难实现了！

旋转

- 定义
- 用旋转维持平衡

AVL树

红黑树（LLRBs）

- 来自2-3树
- LLRB 性质
- LLRB 的插入操作
- LLRB 性能与实现

2-3树 -> BST

- AVL树：利用旋转操作自平衡的树。
- 2-3树（B树）：是一种在构建过程中就可以平衡的树，不需要用到旋转操作。

一个新的想法：

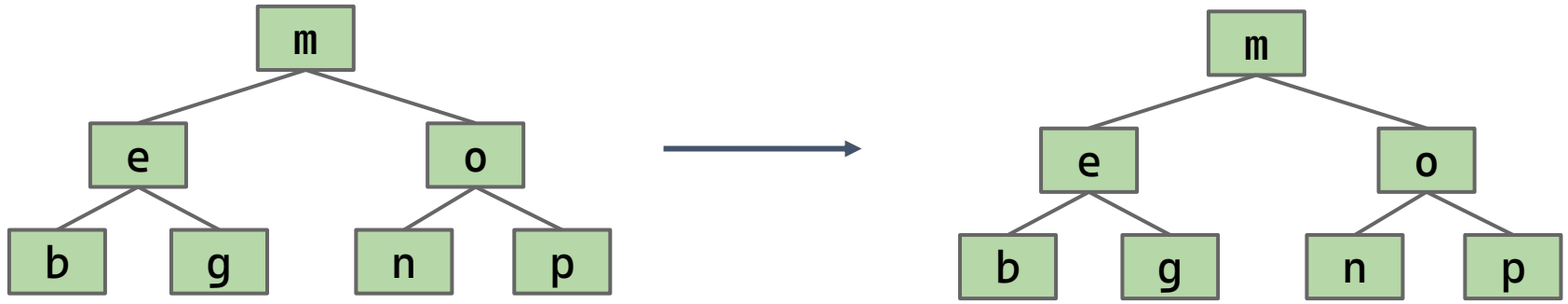
我们有没有办法用 BST 去表示一颗 2-3树，使得两者结构“等价”

- 因为 2-3树 平衡，我们的特殊的 BST 也平衡。

2–3树 –> BST

如果一颗 2–3树 全部都是只有一个元素的结点:

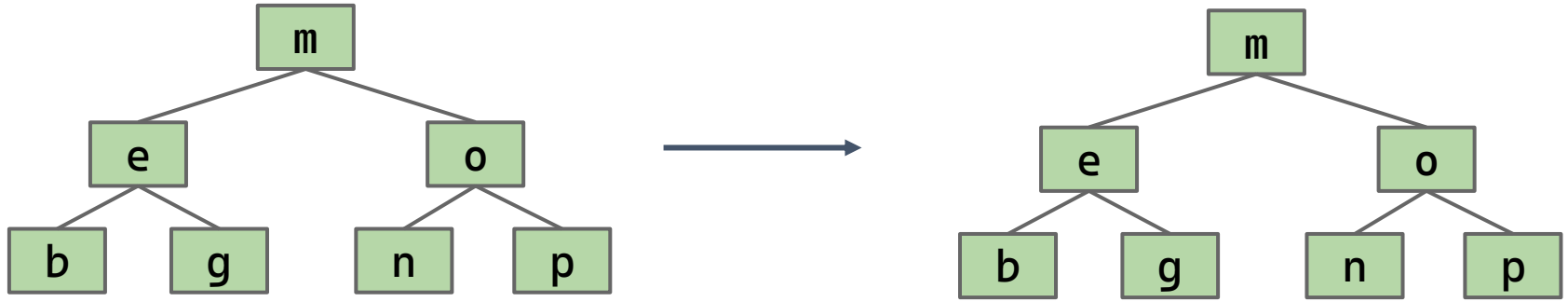
– 特殊 BST 很简单实现:



2-3树 -> BST

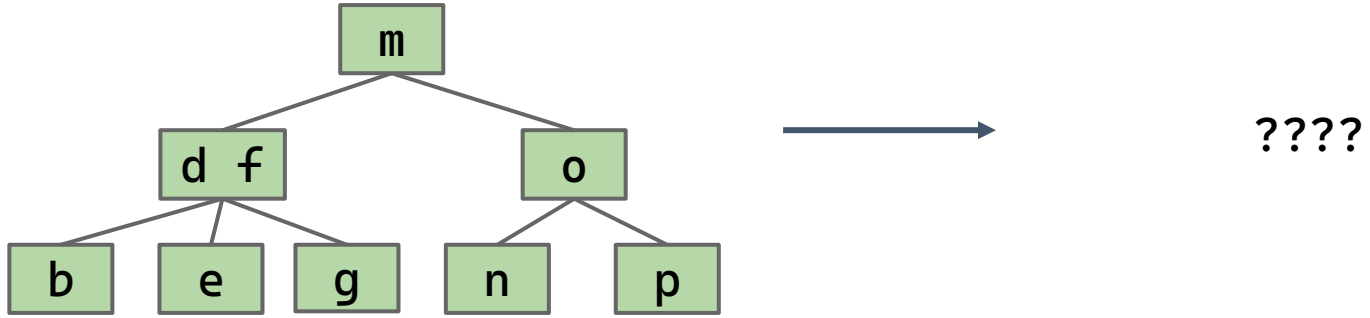
如果一颗 2-3树 全部都是只有一个元素的结点:

- 特殊 BST 很简单实现:



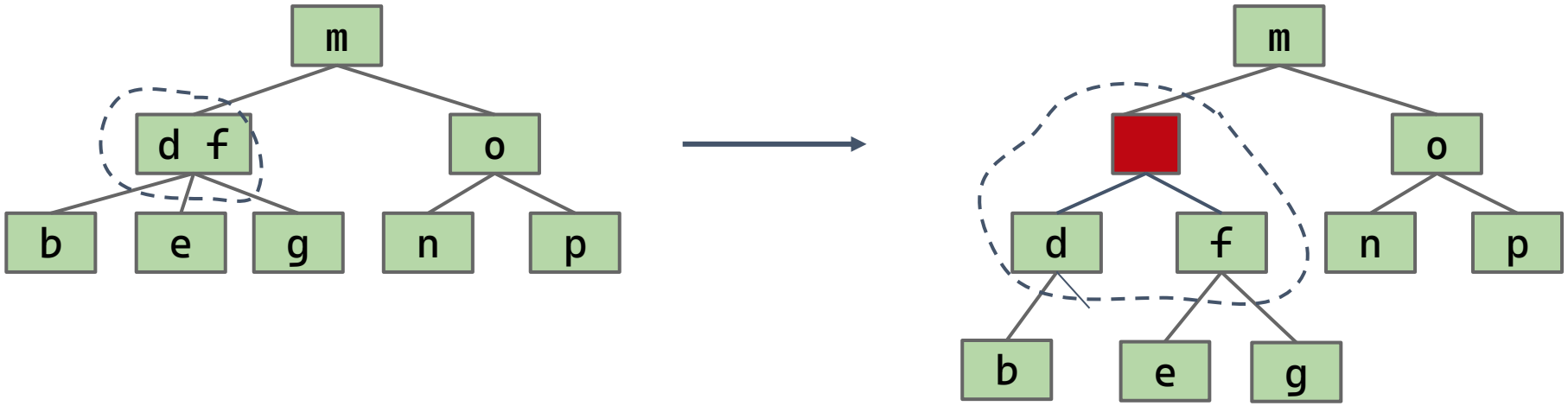
但如果 2-3树 中含有有两个元素的结点呢?

- 注意，二叉搜索树是一颗二叉树，一个结点最多有 2 个子结点!



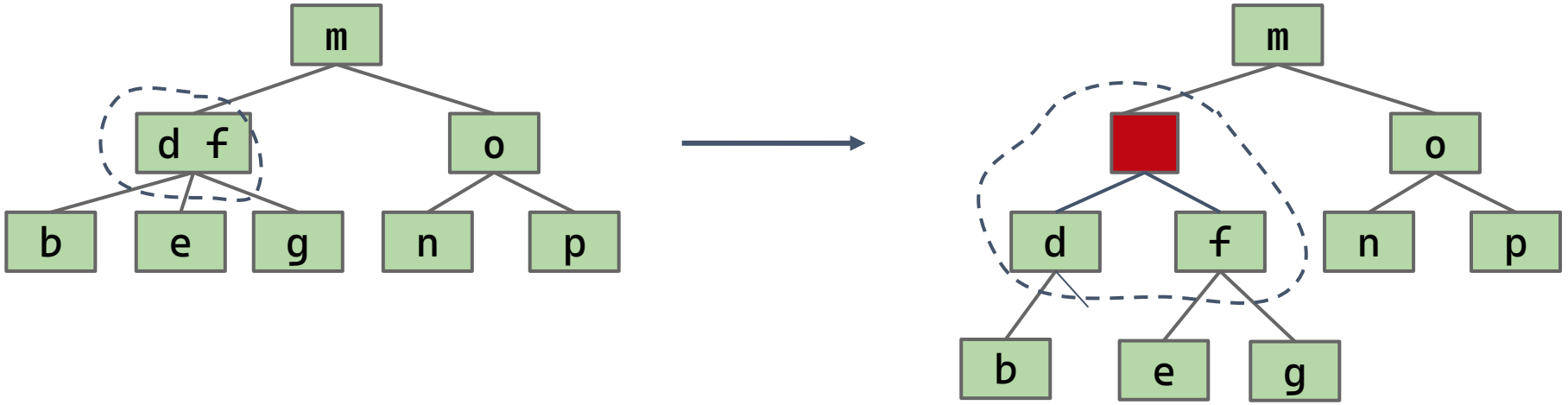
2-3树 -> BST - 解决过载结点问题

一种可能性：用一个“连接”结点

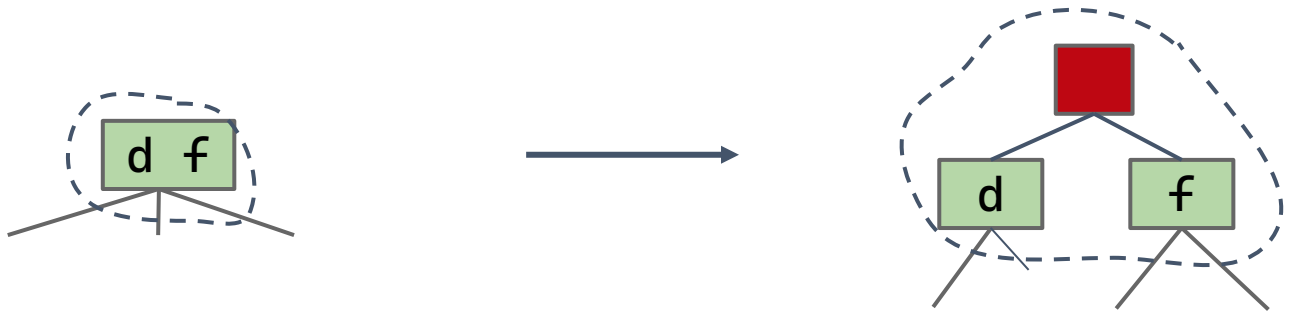


2-3树 -> BST - 解决过载结点问题

一种可能性：用一个“连接”结点

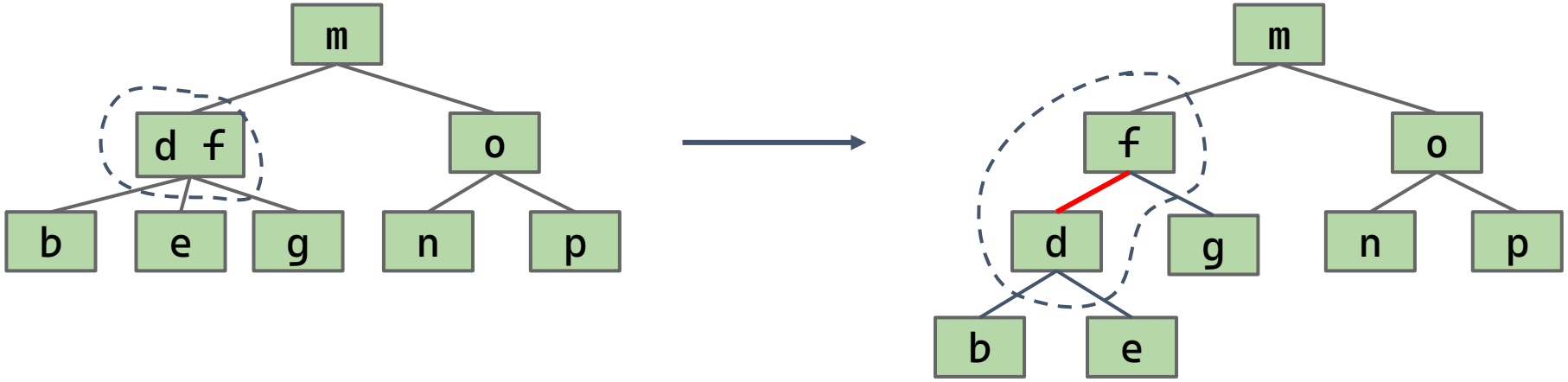


结果其实很不优雅，我们浪费了边数，代码会很丑陋



2-3树 -> BST - 解决过载结点问题

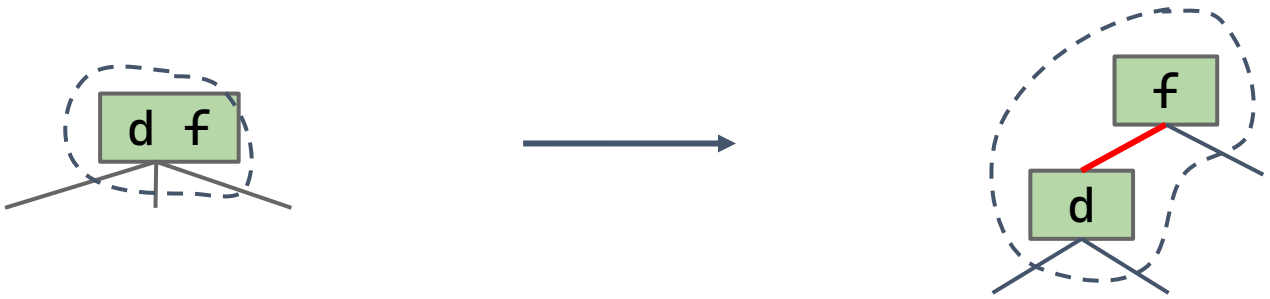
另一种可能性：用一个“连接”边 将较小的元素们**放在左边**



2-3树 -> BST - 解决过载结点问题

另一种可能性：用一个“连接”边 将较小的元素们**放在左边**

为了方便，我们把这些“连接边”标记为红色，对应的，其他边就是黑色边。

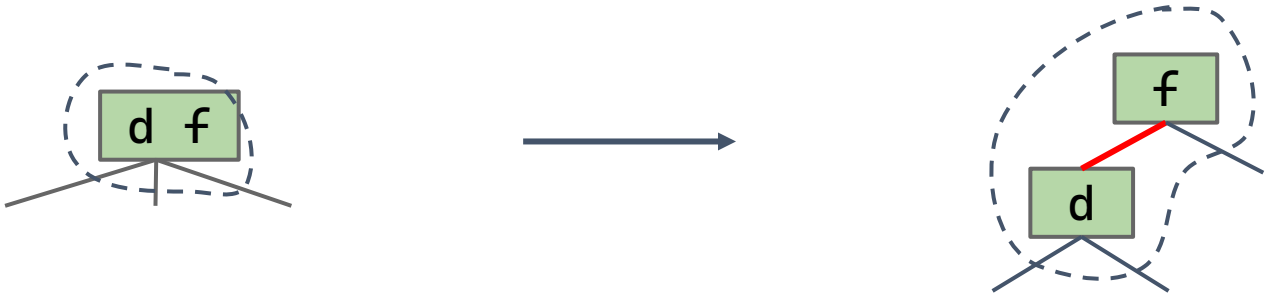


这种“连接边”的思想在实践中经常被用到（eg.java.util.TreeSet）

2-3树 -> BST - 解决过载结点问题

另一种可能性：用一个“连接”边 将较小的元素们**放在左边**

为了方便，我们把这些“连接边”标记为红色，对应的，其他边就是黑色边。

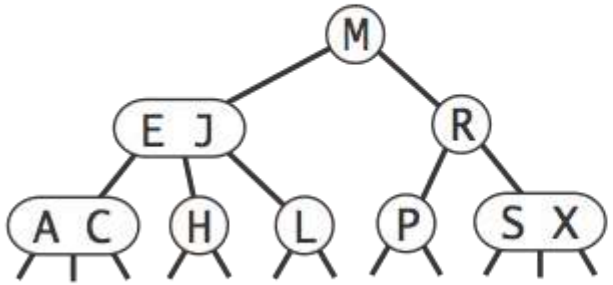


这种“连接边”的思想在实践中经常被用到（eg.java.util.TreeSet）

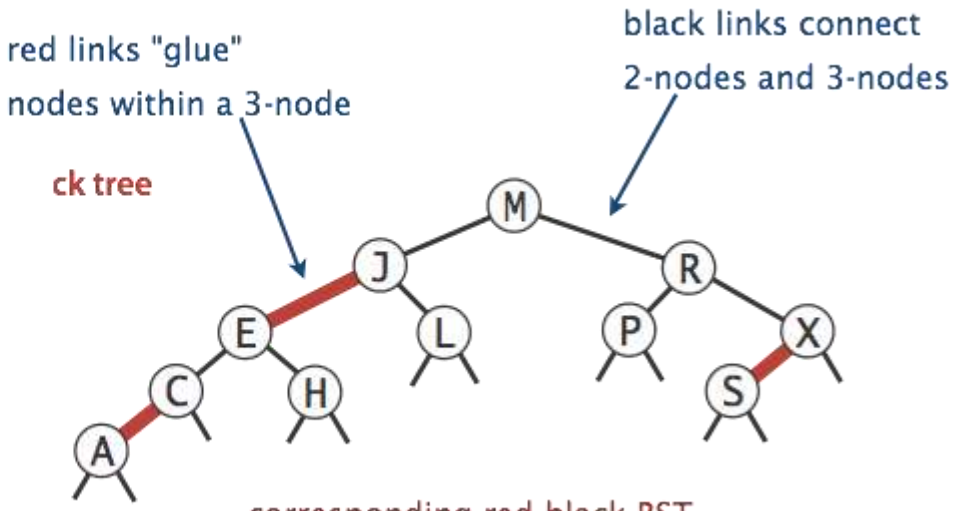
- 在实现中，我们也可以用结点的颜色表示 父结点指向这个结点的边 为红色。

2-3树 -> 左偏红黑树

用**左连接边**表示 2-3树 的 BST，通常称为左偏红黑树（Left Leaning Red Black Binary Search Tree），可以简写为 LLRB。



2-3 tree

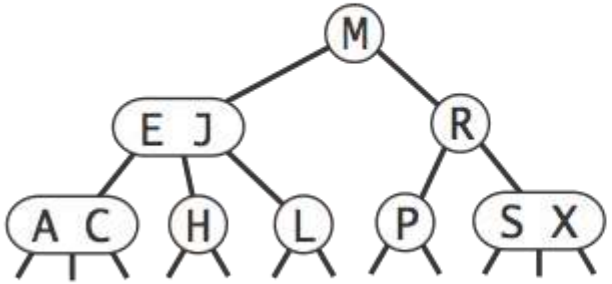


corresponding red-black BST

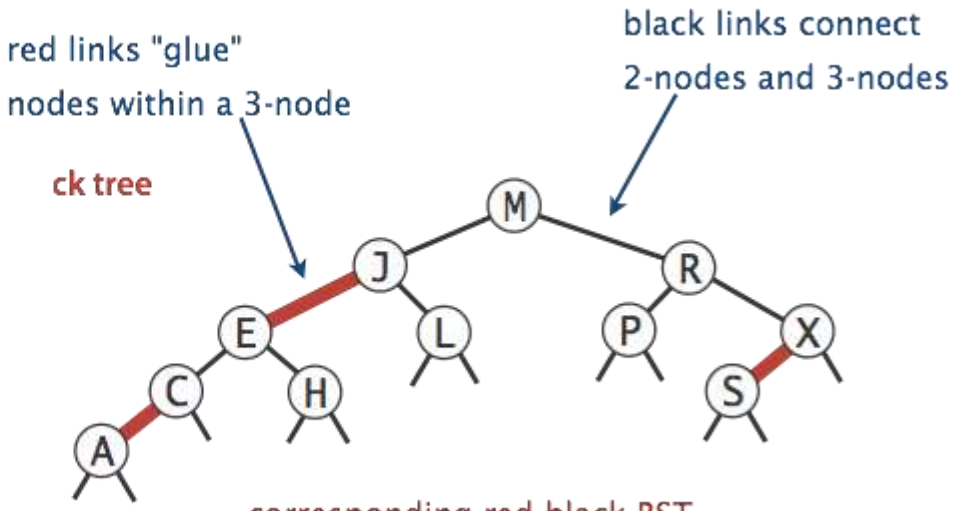
2-3树 -> 左偏红黑树

用**左连接边**表示 2-3树 的 BST，通常称为左偏红黑树（Left Leaning Red Black Binary Search Tree），可以简写为 LLRB。

- LLRB 就是一种二叉搜索树！他的搜索操作和普通的 BST 一模一样！
- LLRB 和 2-3树 有一一对应关系！
- 红色只是起到一种标注作用，不会对树的结构起到改变作用！



2-3 tree



corresponding red-black BST

红黑树：LLRB 性质

Lecture 3

B树太难实现了！

旋转

- 定义
- 用旋转维持平衡

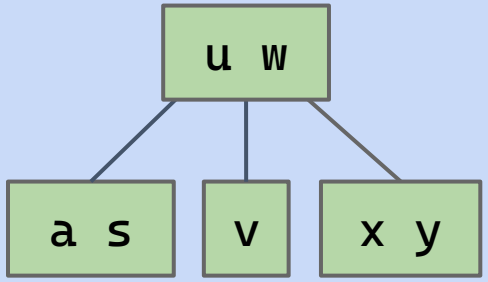
AVL树

红黑树（LLRBs）

- 来自2-3树
- LLRB 性质
- LLRB 的插入操作
- LLRB 性能与实现

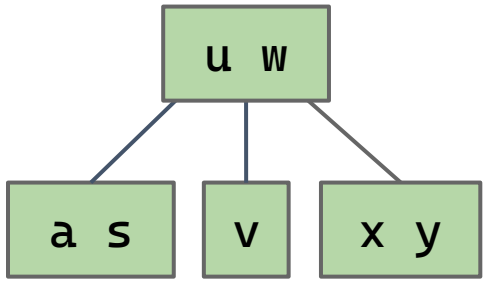
左偏红黑树

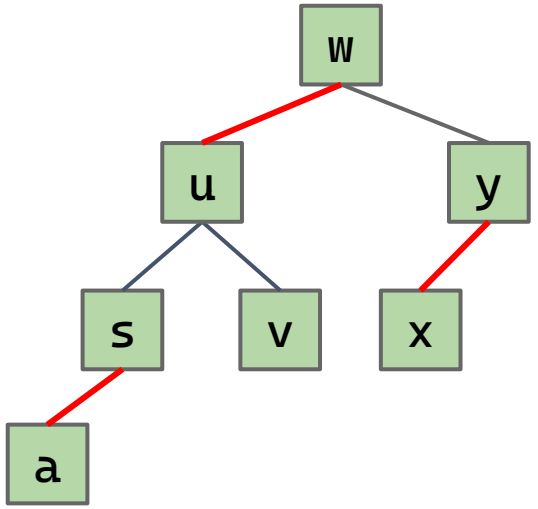
画出与下面 2-3树 对应的左偏红黑树:



左偏红黑树

画出与下面 2-3树 对应的左偏红黑树:

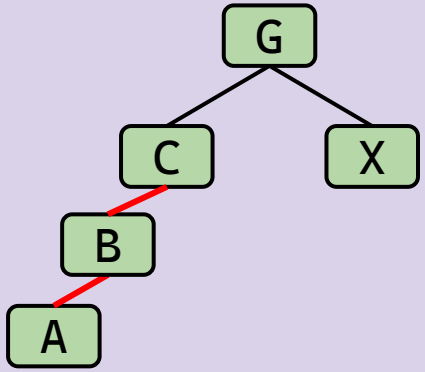


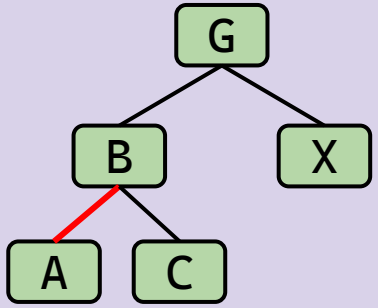


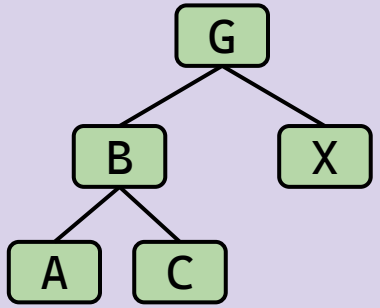
难题 #1 !

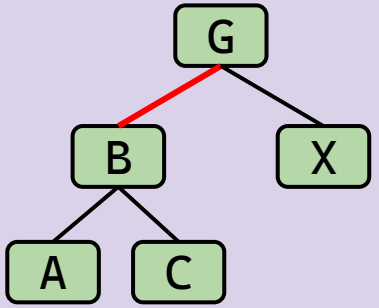
下面的左偏红黑树中，哪个是合法的左偏红黑树?

提示：与合法的 2-3树 有一一对应关系。





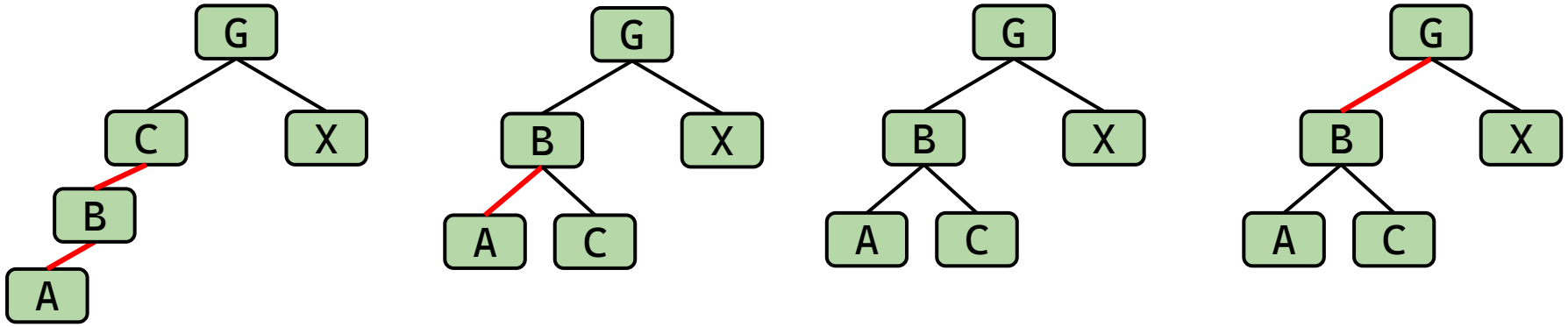




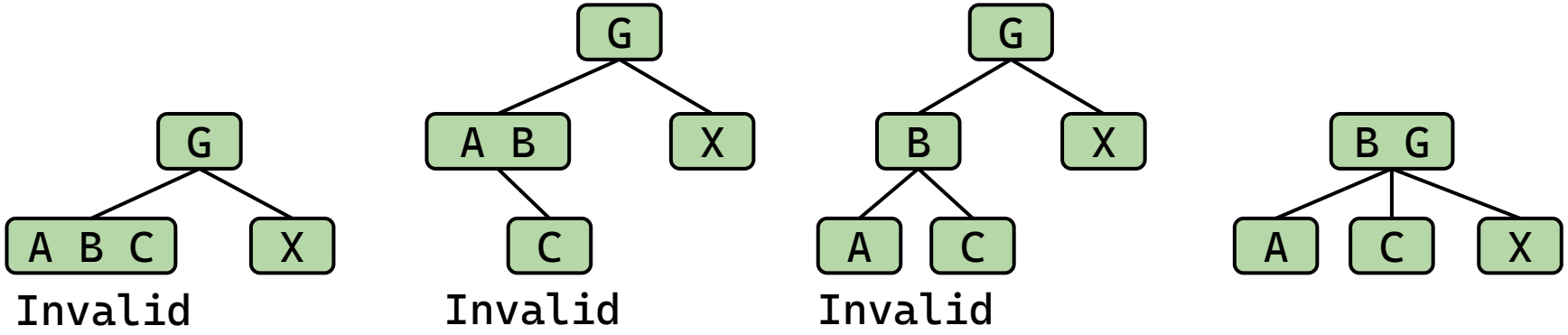
难题 #1 ！

下面的左偏红黑树中，哪个是合法的左偏红黑树？

提示：与合法的 2-3树 有一一对应关系。

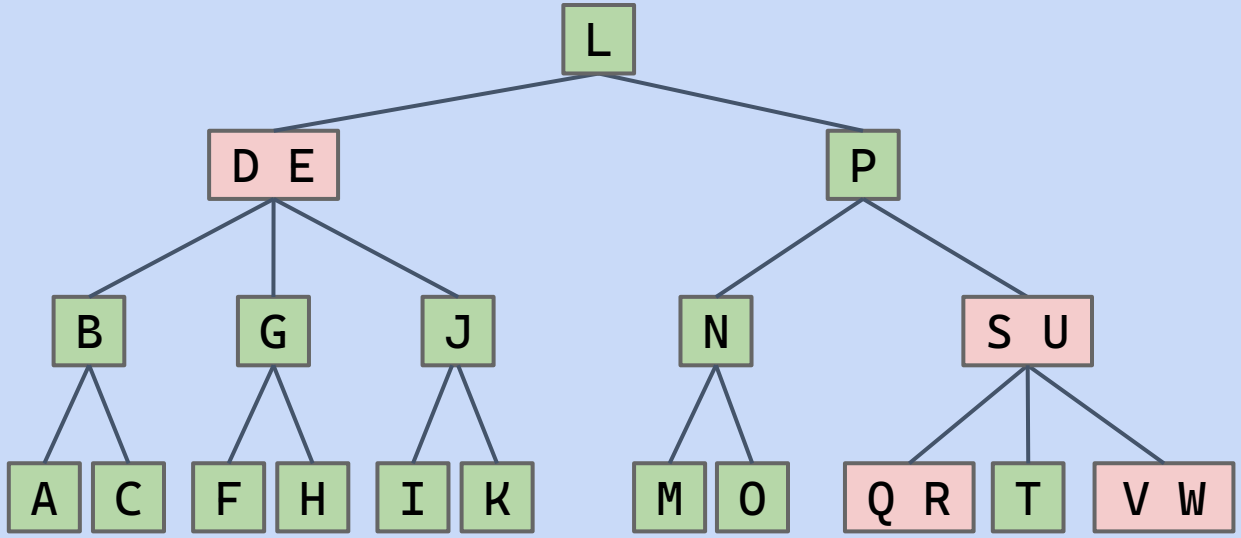


Equivalent 2-3



练习

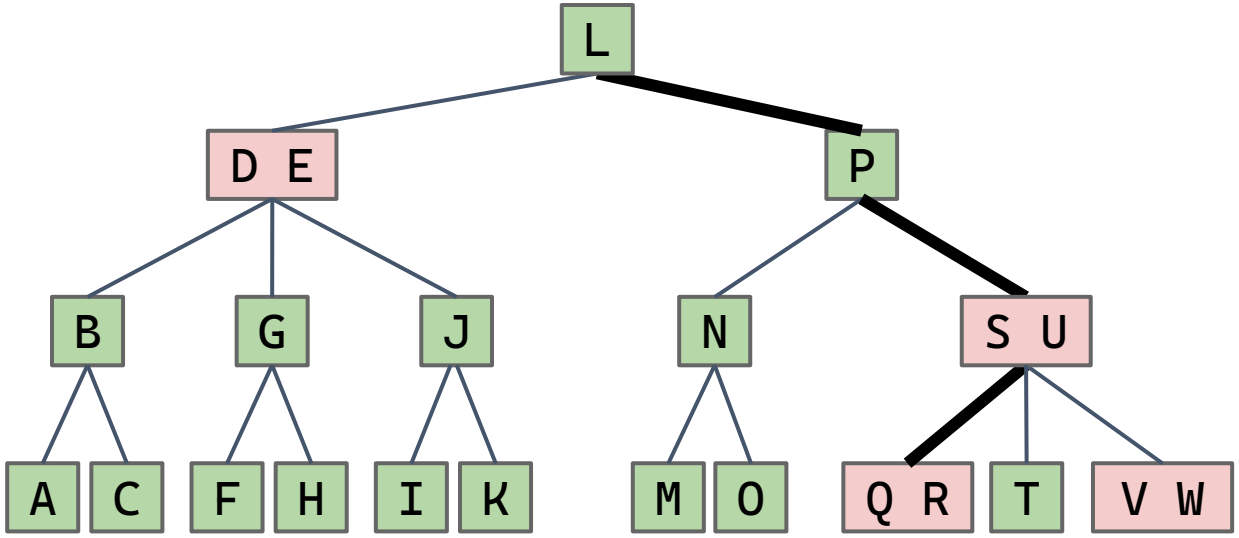
下面这棵 2-3树 对应的左偏红黑树的高度是多少?



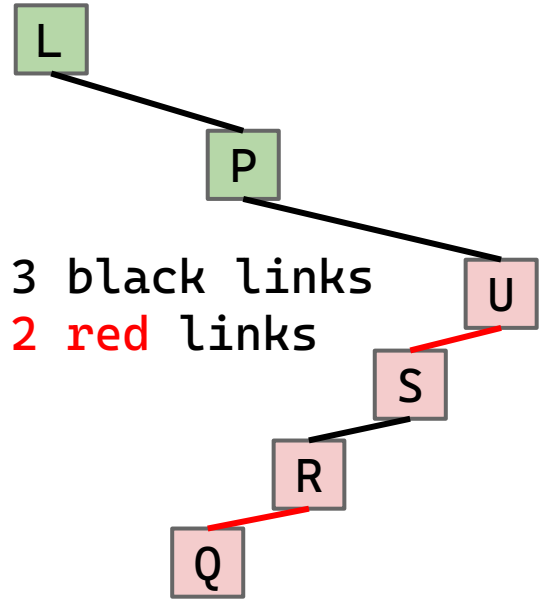
练习

下面这棵 2-3树 对应的左偏红黑树的高度是多少?

- 高度 = 3(black) + 2(red) = 5



Dark line shows longest path (3 links).



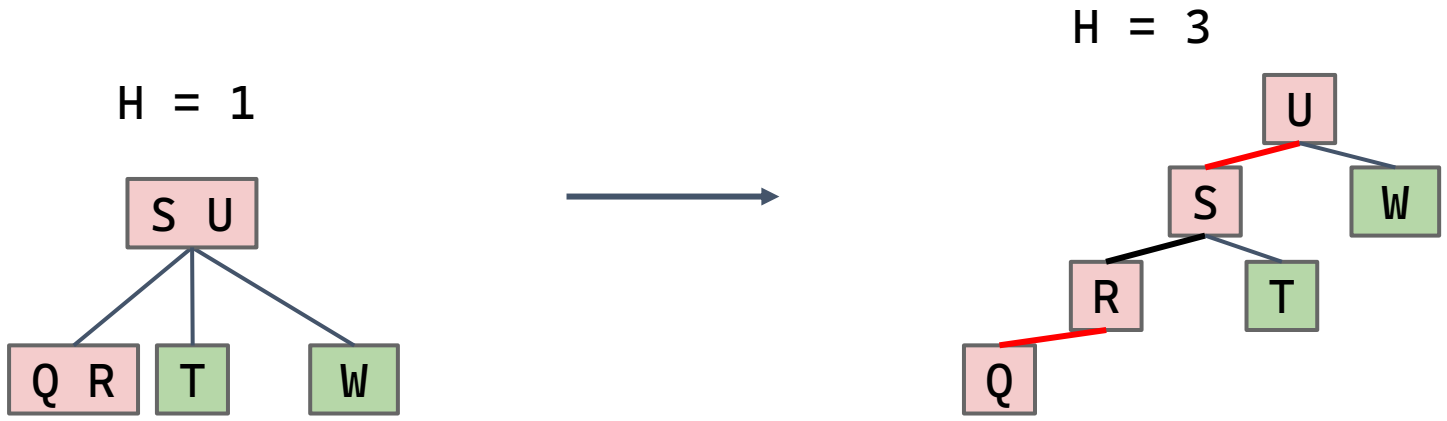
难题 #2 !

假设一棵 2-3树 高度为 H，它对应的左偏红黑树最大高是多少?

难题 #2 ！

假设一棵 2-3树 高度为 H，它对应的左偏红黑树最大高是多少？

- Hmax = H(black) + (H + 1)(red) = 2H + 1

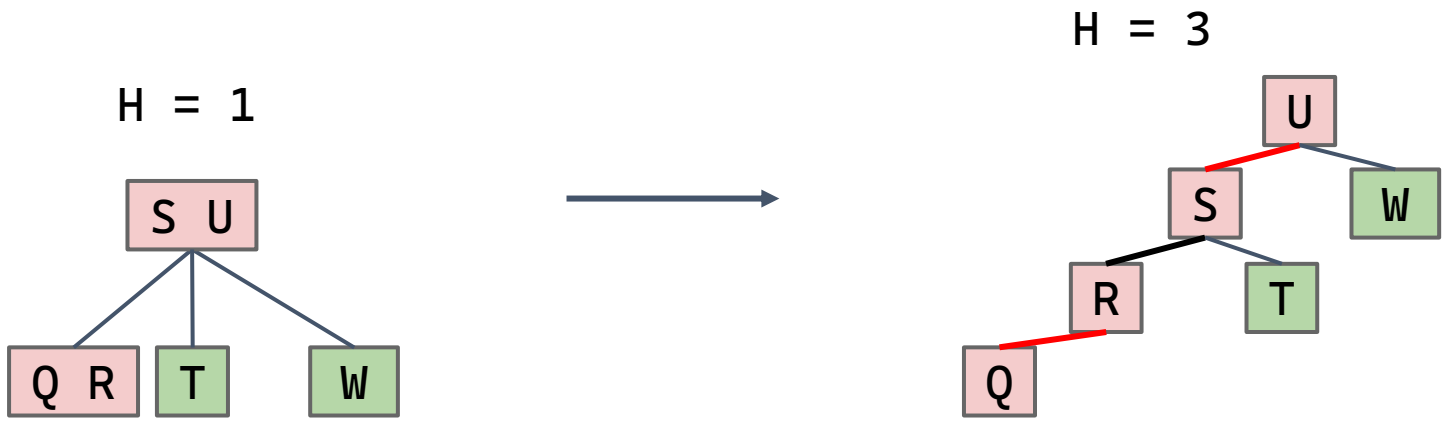


难题 #2 !

假设一棵 2-3树 高度为 H，它对应的左偏红黑树最大高是多少?

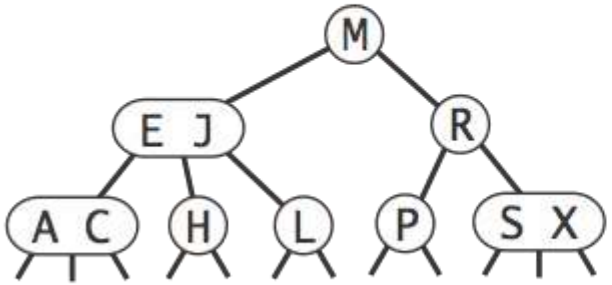
- Hmax = H(black) + (H + 1)(red) = 2H + 1

因为 2-3树 的高度是 O(log N)，而与其对应的同样元素数量的 LLRB 高度约等于 2-3树 高度的两倍，所以 LLRB 的高度也是 O(log N)!

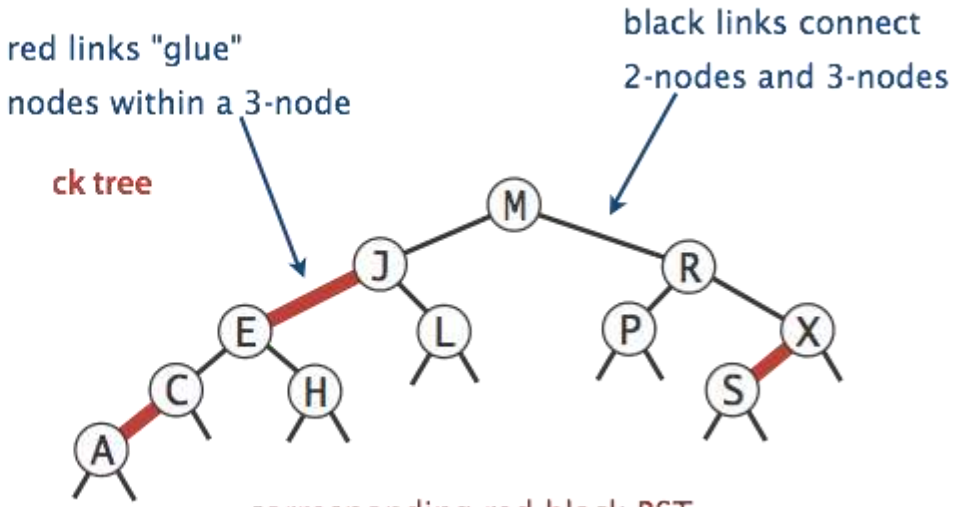


左偏红黑树 – 性质

- 不可能有结点有两个红色边。
  - 如果有结点有两个红色边，那它对应的 2-3树 就会有一个三元素结点，这是不行的。
- 从根结点到每个叶结点的所有简单路径上都包含相同数目的黑色边。
  - 对应了 2-3树 的一个性质：所有叶子结点都在同一层。



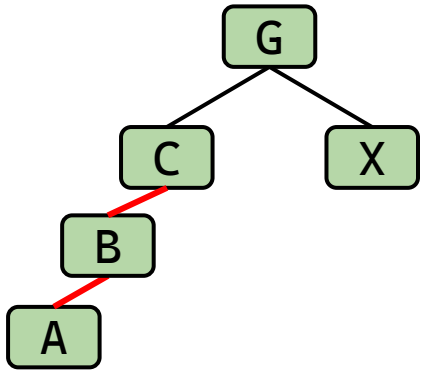
2-3 tree



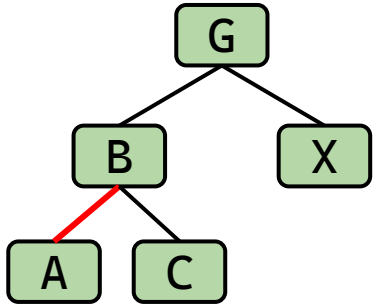
corresponding red-black BST

难题 #1 ！

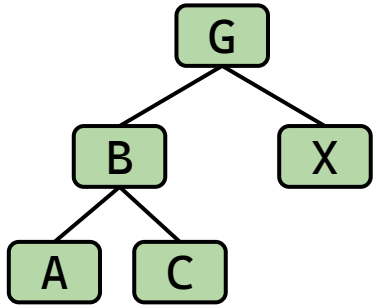
下面的左偏红黑树中，哪个是合法的左偏红黑树？



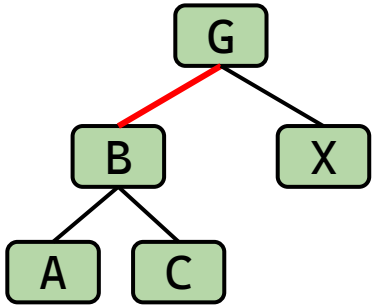
Invalid, B has two red links.



Invalid, not black balanced.



Invalid, not black balanced.



Valid

红黑树：
LLRB 的插入操作

Lecture 3

B树太难实现了！

旋转

- 定义
- 用旋转维持平衡

AVL树

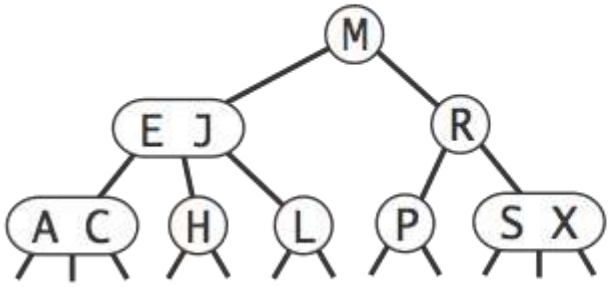
红黑树（LLRBs）

- 来自2-3树
- LLRB 性质
- LLRB 的插入操作
- LLRB 性能与实现

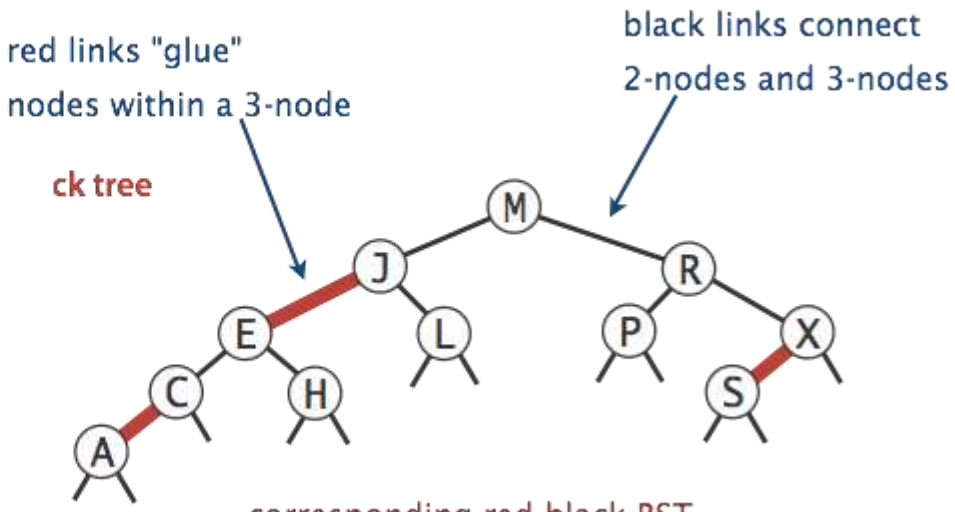
LLRB 从哪里来?

一个重要的问题：LLRB 从哪里来?

- 先构建一棵 2-3树，再将 2-3树 转换为 LLRB?



2-3 tree

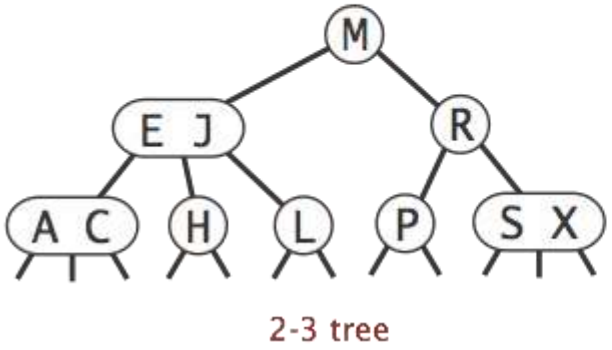


corresponding red-black BST

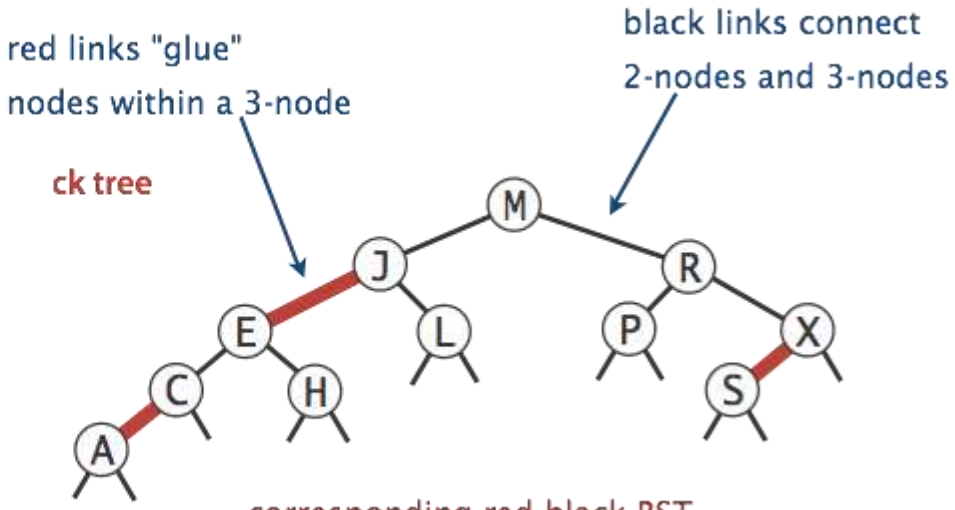
LLRB 从哪里来?

一个重要的问题：LLRB 从哪里来？

- ~~先构建一棵 2-3树，再将 2-3树 转换为 LLRB？~~
- 我们用如下策略实现 LLRB 的插入：
  - 像插入一般的 BST 一样先进行插入。
  - 用旋转操作保证 LLRB 与 2-3树 的一一对应关系（也就是在维持平衡）。



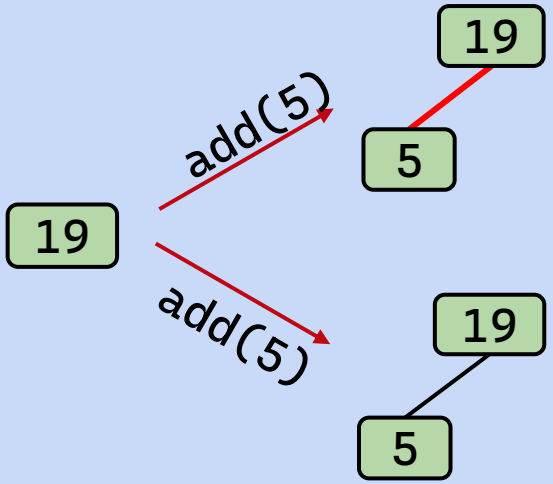
2-3 tree



corresponding red-black BST

Design Task #1: Insertion Color

我们的插入应该用什么颜色的边？红？黑？

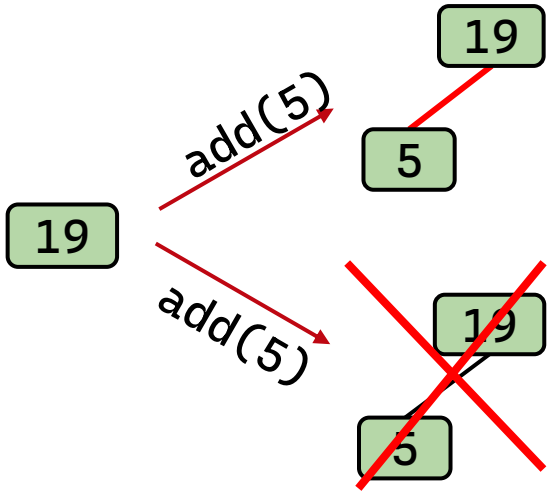


LLRB World

World 2-3

Design Task #1: Insertion Color

我们的插入应该用什么颜色的边？红？黑？

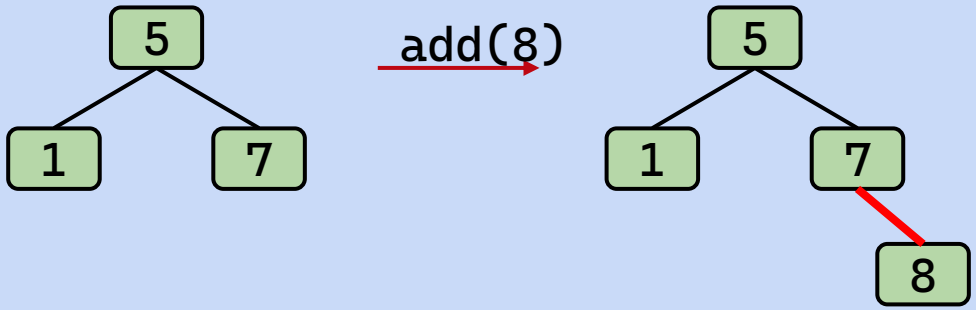


LLRB World

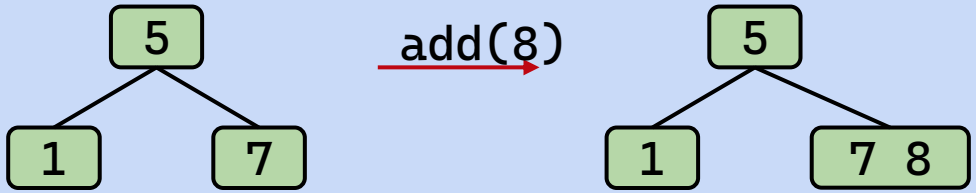
World 2-3

Design Task #2: Insertion on the Right

假设我们加入的元素处于右结点，按照规则，我们应该使用红边。但这样违反了“左偏”的规则，该怎么办呢？



LLRB World

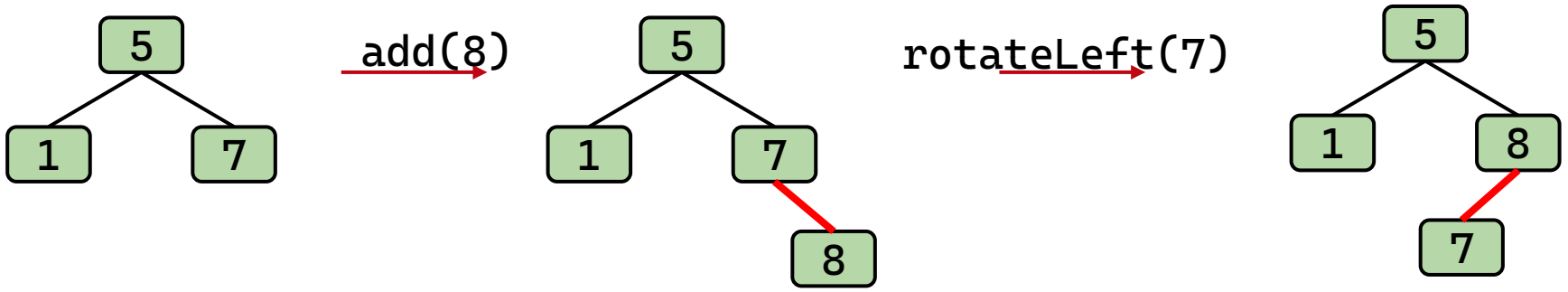


World 2-3

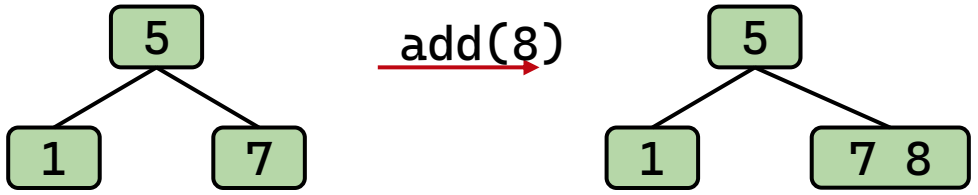
Design Task #2: Insertion on the Right

假设我们加入的元素处于右结点，按照规则，我们应该使用红边。但这样违反了“左偏”的规则，该怎么办呢？

- 别忘了我们可以用旋转操作来维持一一对应关系



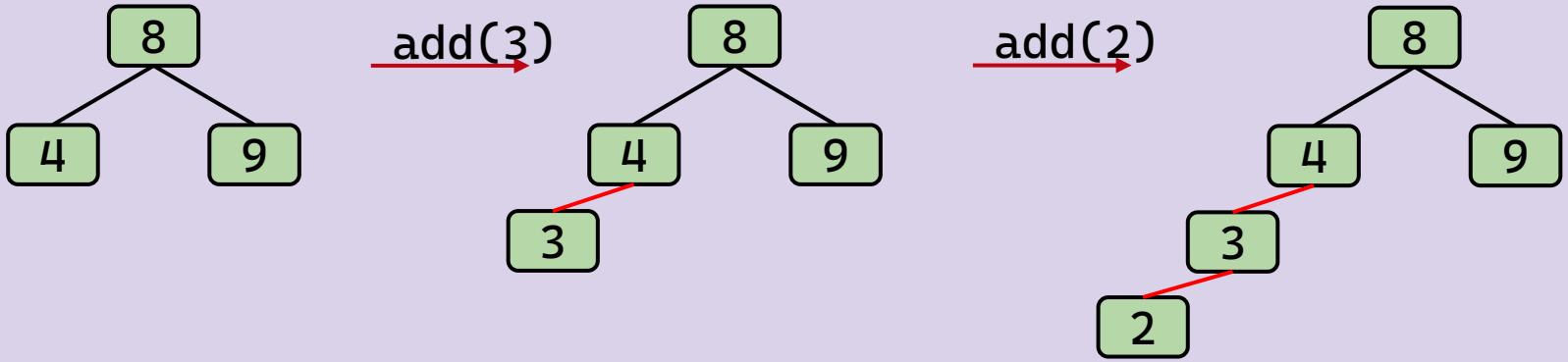
LLRB World



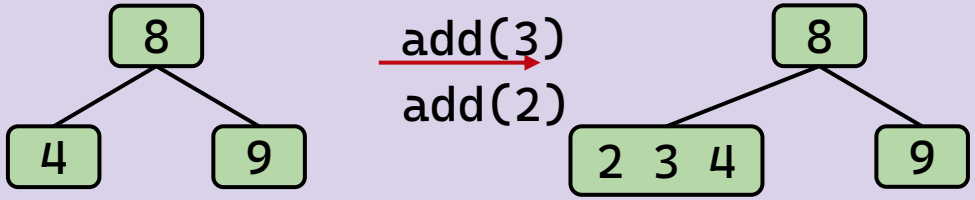
World 2-3

Design Task #3: Double Insertion on the Left

假设我们加入的元素造成了如下两个相邻红边出现的情况，该怎么办呢？



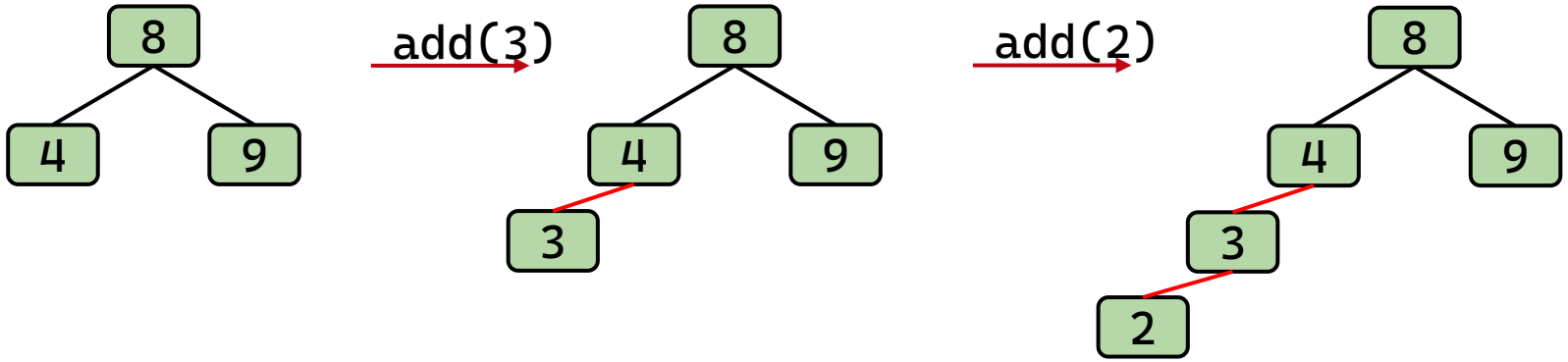
LLRB World



World 2-3

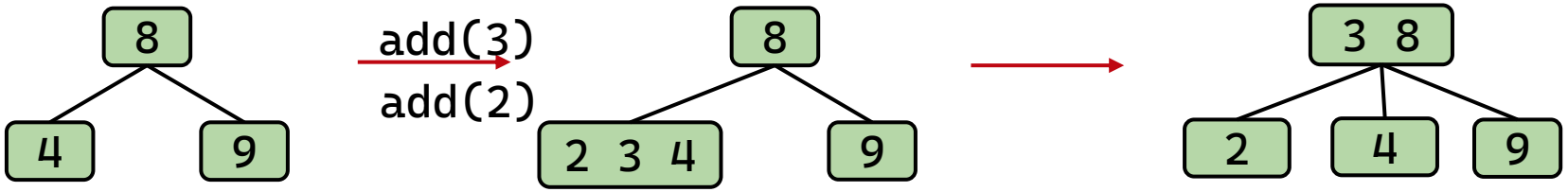
Design Task #3: Double Insertion on the Left

假设我们加入的元素造成了如下两个相邻红边出现的情况，该怎么办呢?



LLRB World

对于 2-3树 来说，是时候分裂了:



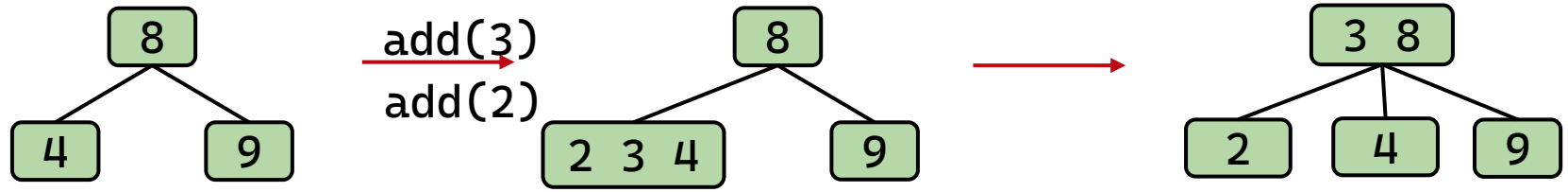
World 2-3

Design Task #3: Double Insertion on the Left

假设我们加入的元素造成了如下两个相邻红边出现的情况，该怎么办呢?

LLRB World

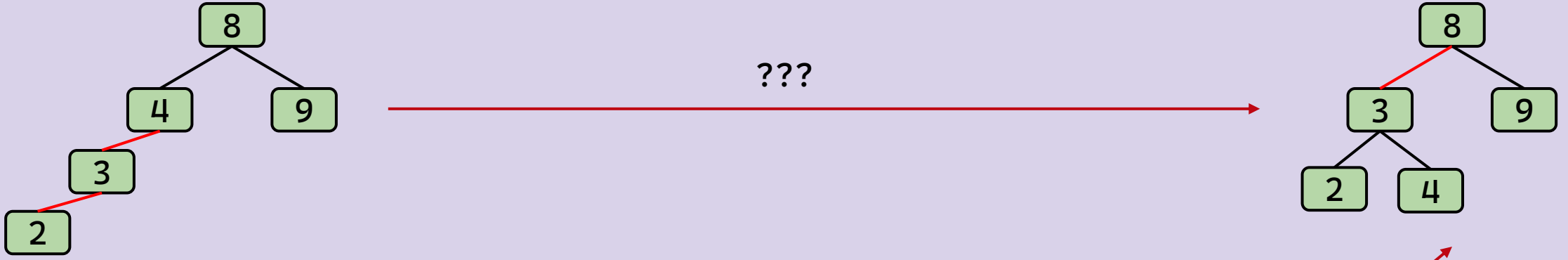
对于 2-3树 来说，是时候分裂了:



World 2-3

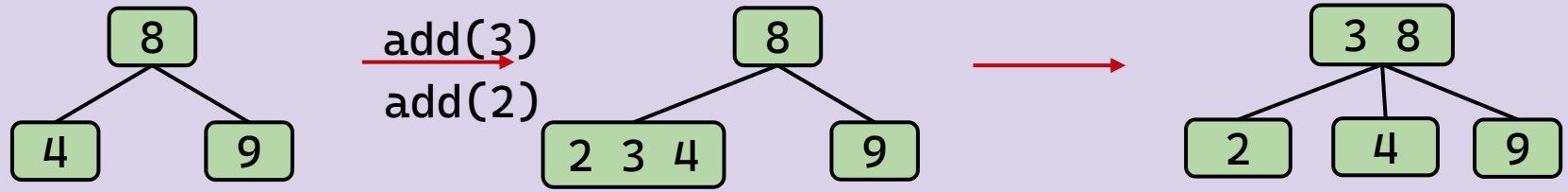
Design Task #3: Double Insertion on the Left

假设我们加入的元素造成了如下两个相邻红边出现的情况，该怎么办呢?



LLRB World

对于 2-3树 来说，是时候分裂了:

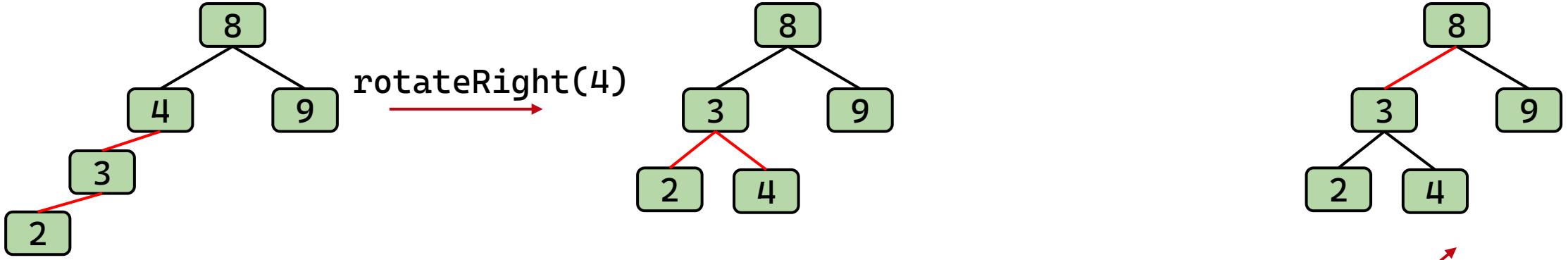


World 2-3

Design Task #3: Double Insertion on the Left

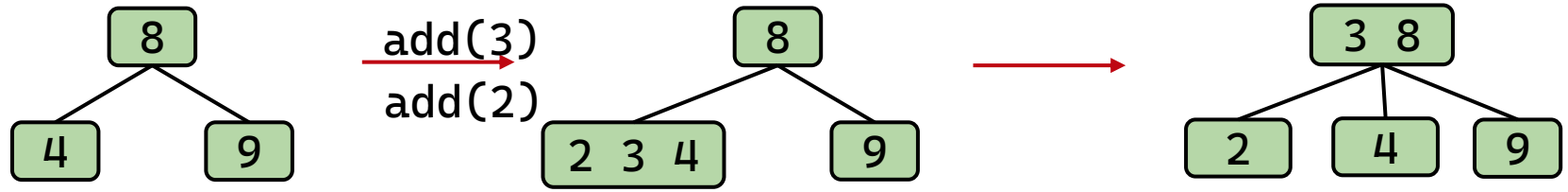
假设我们加入的元素造成了如下两个相邻红边出现的情况，该怎么办呢?

- 旋转，使 3 有两个红色的子边



LLRB World

对于 2-3树 来说，是时候分裂了:

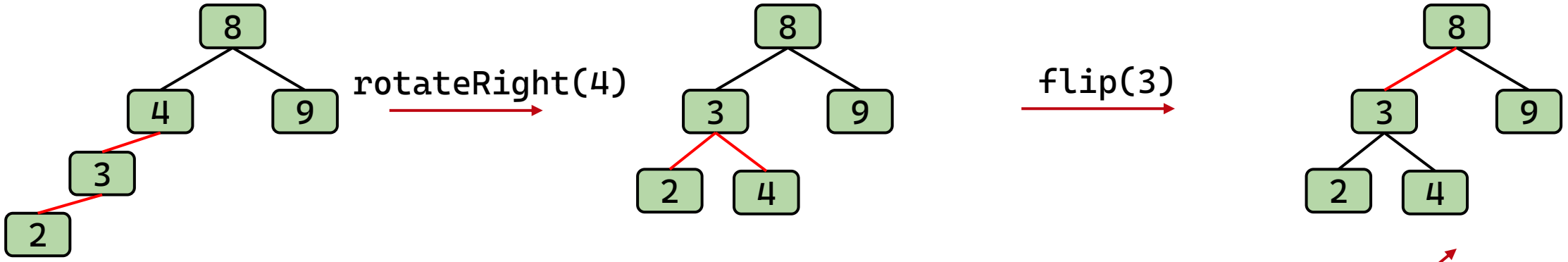


World 2-3

Design Task #3: Double Insertion on the Left

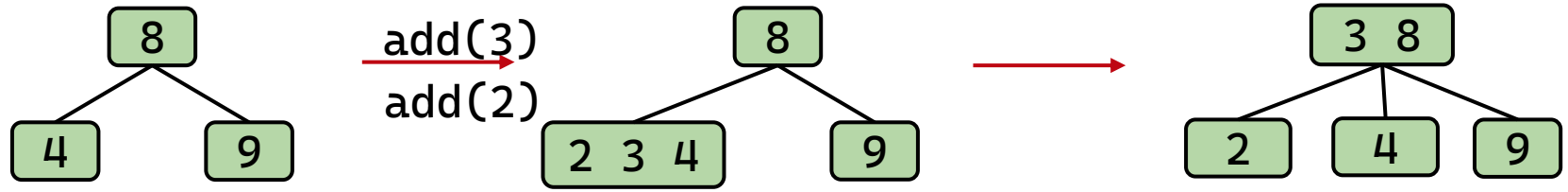
假设我们加入的元素造成了如下两个相邻红边出现的情况，该怎么办呢?

- 旋转，使 3 有两个红色的子边
- 颜色翻转



LLRB World

对于 2-3树 来说，是时候分裂了:



World 2-3

这就是全部插入规则了！

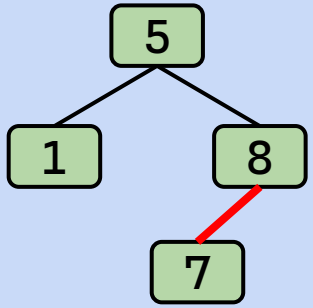
在插入时，只需遵守下面四个规则即可：

- 1．插入时，总是用红色边。
- 2．如果插入在了右结点：左旋相应结点即可
- 3．如果插入后产生相邻的两个红边：右旋相应结点，产生了一个结点有两个红色子边的情况。
- 4．如果一个结点有两个红色子边：颜色翻转即可。

插入时，可能产生连锁反应，颜色翻转可能导致红色边出现在右边，没关系，我们见招拆招，利用规则二进行正确的左旋即可：

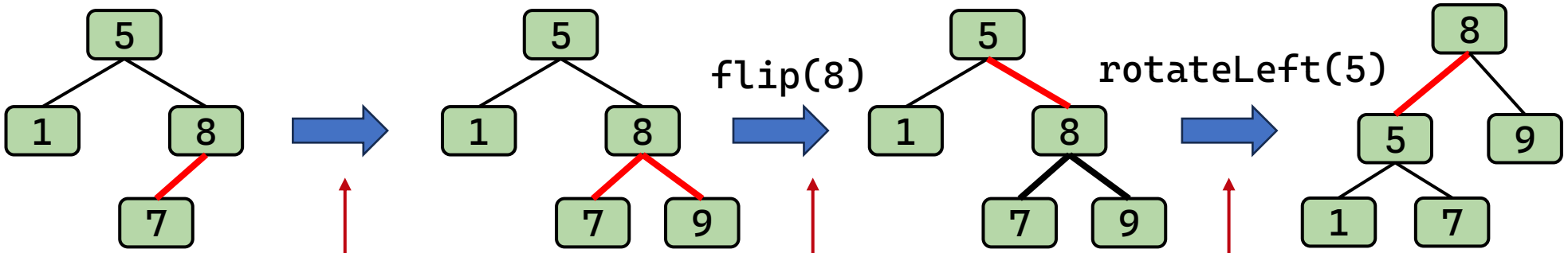
练习插入规则:

在如下 LLRB 中插入 9 。画出插入后的 LLRB。



练习插入规则：

在如下 LLRB 中插入 9 。画出插入后的 LLRB。



1. 插入时，总是用红色边。

4. 如果一个结点有两个子红边：颜色翻转即可。

2. 如果插入在了右结点：左旋相应结点即可

红黑树：
LLRB 性能与实现

Lecture 3

B树太难实现了！

旋转

- 定义
- 用旋转维持平衡

AVL树

红黑树（LLRBs）

- 来自2-3树
- LLRB 性质
- LLRB 的插入操作
- LLRB 性能与实现

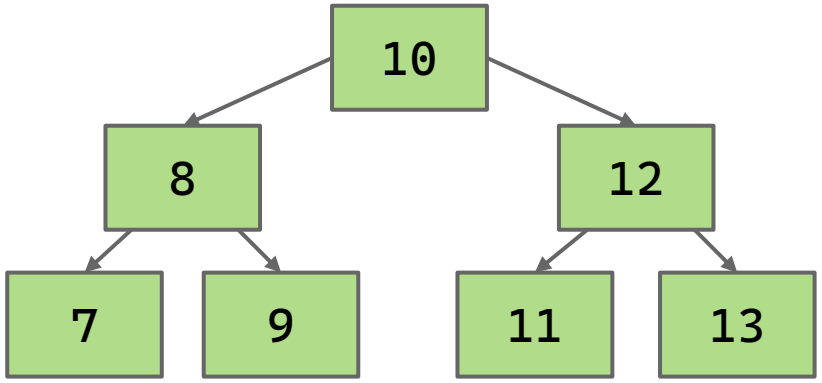
LLRB 的性能

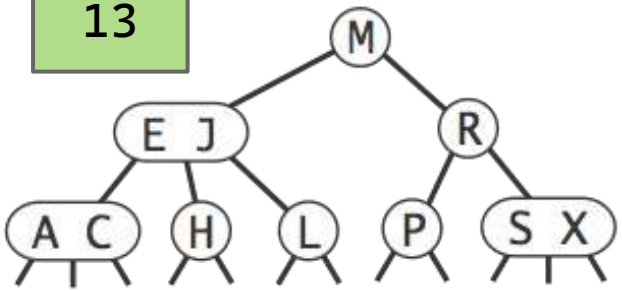
我们此前已经证明过，LLRB 的高度也是 O(log N)。

- 搜索操作和一般的 BST 相同，时间复杂度为 O(log N)。
- 插入操作也是 O(log N)：
  - O(log N) to add the new node.
  - O(log N) to do rotation and color flip operations.
- 我们不会讲到删除操作，删除操作的时间复杂度同样是 O(log N)。
  - 可以在 https://xjtu.men/oi/ds/llrbt 了解删除操作。

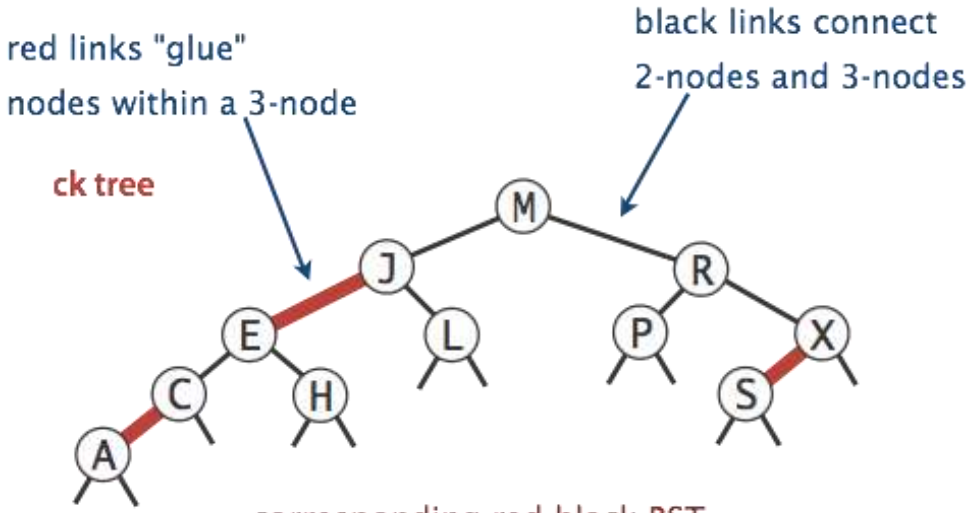
LLRB 的实现

```
1.private Node put(Node h, Key key, Value val) {
2.    if (h == null) { return new Node(key, val, RED); }
3.    int cmp = key.compareTo(h.key);
4.    if (cmp < 0)      { h.left  = put(h.left,  key, val); }
5.    else if (cmp > 0) { h.right = put(h.right, key, val); }
6.    else              { h.val   = val;                    }
7.
8.    if (isRed(h.right) && !isRed(h.left))      { h = rotateLeft(h);  }
9.    if (isRed(h.left)  &&  isRed(h.left.left)) { h = rotateRight(h); }
10.   if (isRed(h.left)  &&  isRed(h.right))     { flipColors(h);      }
11.   return h;
12.}
```





2-3 tree



corresponding red-black BST

谢谢大家

主讲人：七海Nana7mi

课程大纲：CS61B