# RBTree

潘小飞 20211016

目录

[RBTree 1](#_Toc85376601)

[1 二叉树 1](#_Toc85376602)

[2 平衡树 1](#_Toc85376603)

[3 AVL树 1](#_Toc85376604)

[4 2-3树 2](#_Toc85376605)

[4.1 创建规则 3](#_Toc85376606)

[5 2-3-4树 4](#_Toc85376607)

[5.1 插入操作 5](#_Toc85376608)

[6 B树 7](#_Toc85376609)

[7 红黑树 8](#_Toc85376610)

[7.1 红黑树与2-3-4树对比 8](#_Toc85376611)

[7.2 红黑树黑节点 9](#_Toc85376612)

[7.3 红黑树与2-3-4树节点转化 10](#_Toc85376613)

[7.4 红黑树与2-3树节点转化 10](#_Toc85376614)

[7.5 2-3树操作 12](#_Toc85376615)

[7.5.1 插入 12](#_Toc85376616)

[7.5.2 删除 14](#_Toc85376617)

[7.6 红黑树定义 17](#_Toc85376618)

[7.7 红黑树调整 17](#_Toc85376619)

[7.7.1 左旋 17](#_Toc85376620)

[7.7.2 右旋 18](#_Toc85376621)

[7.7.3 添加 19](#_Toc85376622)

[7.7.4 删除 19](#_Toc85376623)

[7.7.5 左倾红黑树插入 20](#_Toc85376624)

[7.7.6 左倾红黑树删除 23](#_Toc85376625)

# 二叉树

满足以下两个条件的树就是二叉树：

* 本身是有序树（若将树中每个结点的各子树看成是从左到右有次序的(即不能互换），则称该树为有序树(Ordered Tree)）。
* 树中包含的各个节点的度不能超过 2，即只能是 0、1 或者 2。

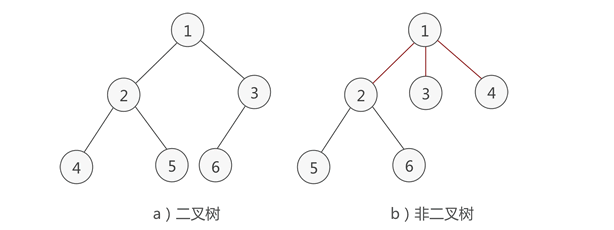


图 1‑1 二叉树

# 平衡树

平衡树（Balance Tree，BT）指的是，任意节点的子树的高度差都小于等于 1。  
常见的符合平衡树的有 AVL 树（二叉平衡搜索树），B 树（多路平衡搜索树，2-3 树，2-3-4 树中的一种），红黑树等。

# AVL树

AVL 树（由发明者 Adelson-Velsky 和 Landis 的首字母缩写命名），是指任意节点的两个子树的高度差不超过 1 的平衡树。又称自平衡二叉搜索树。  
AVL 树能解决上文二叉查找树中的右瘸子问题，例如，插入数据依次为 {1,2,3,4,5}（从小到大），则如下图所示：

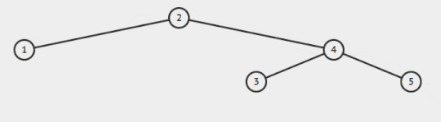


图 3‑1 AVL树

# 2-3树

2-3 树，是指每个具有子节点的节点（内部节点，internal node）要么有两个子节点和一个数据元素，要么有三个子节点和两个数据元素的自平衡的树，它的所有叶子节点都具有相同的高度。

简单点讲，2-3 树的非叶子节点都具有两个分叉或者三个分叉，所以，称作 2 叉-3 叉树更容易理解。

另外一种说法，具有两个子节点和一个数据元素的节点又称作 2 节点，具有三个子节点和两个数据元素的节点又称作 3 节点，所以，整颗树叫做 2-3 树。

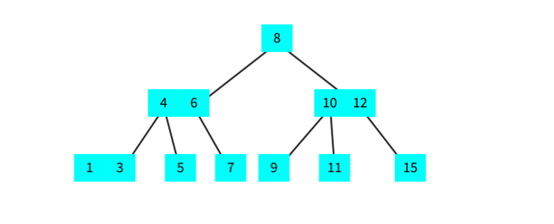


图 4‑1 2-3树

所有叶子点都在树的同一层，一样高：

* **性质 1：**满足二叉搜索树的性质。
* **性质 2：**节点可以存放一个或两个元素。
* **性质 3：**每个节点有两个或三个子节点。

## 创建规则

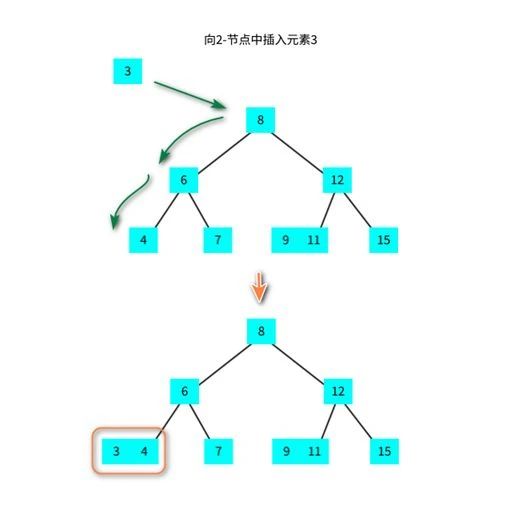


图 4‑2 插入元素3

向 2-节点中插入元素。

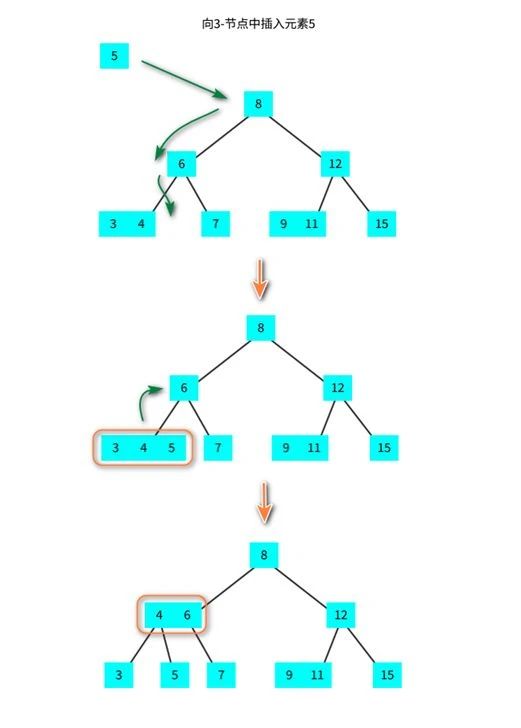


图 4‑3 插入元素5

向一颗只含有一个 3-节点的树中插入元素。

# 2-3-4树

含义如下：

* **2 节点：**包含两个子节点和一个数据元素。
* **3 节点：**包含三个子节点和一个数据元素。
* **4 节点：**包含四个子节点和一个数据元素。

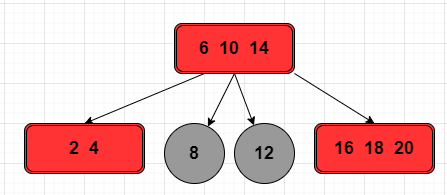


图 5‑1 2-3-4树

2-3-4 树，它的每个非叶子节点，要么是 2 节点，要么是 3 节点，要么是 4 节点，且可以自平衡，所以称作 2-3-4 树。

规则如下：

* **规则 1：**加入新节点时，不会往空的位置添加节点，而是添加到最后一个叶子节点上。
* **规则 2：**四节点可以被分解三个 2-节点组成的树，并且分解后新树的根节点需要向上和父节点融合。

## 插入操作

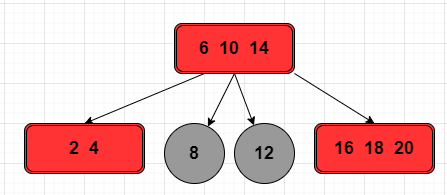


图 5‑2 原图

对于上图的 2-3-4 树，插入一个节点 17，由于规则 1，节点 17 不会加入节点 [16,18,20] 的子树，而是与该节点融合。

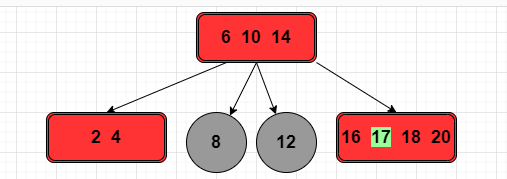


图 5‑3 插入17

由于规则 2，节点 [16,17,18,20] 是一个 4 节点，将该节点进行拆解成新的树，将 18 作为子树的根节点进行拆分。

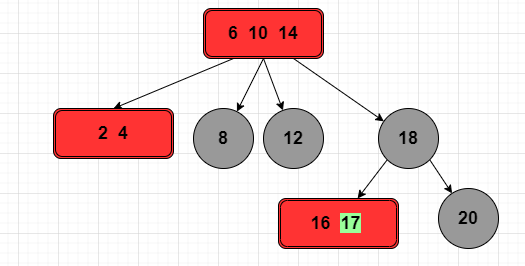


图 5‑4 拆解

此时树暂时失去了平衡，我们需要将拆分后的子树的根节点向上进行融合。

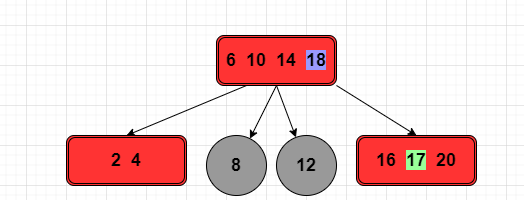


图 5‑5 融合

同理可得，由于规则 2，节点 [6,10,14,18] 是一个 4 节点，将该节点进行拆解成新的树，将 14 作为子树的根节点进行拆分，完成了 2-3-4 树的构建。

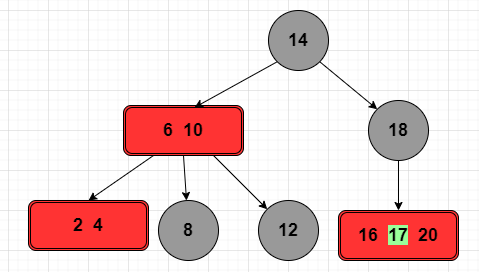


图 5‑6 调整

总结了下插入节点的过程，无非也就为了符合两条规则，那么，2-3 树，2-3-4 树都有了，那是不是也有 2-3-4-5 树，2-3-4-5--...-n 树的存在呢？

# B树

B 树，表示的是一类树，它允许一个节点可以有多于两个子节点，同时，也是自平衡的，叶子节点的高度都是相同的。

所以，为了更好地区分一颗 B 树到底属于哪一类树，我们给它一个新的属性：度（Degree）：一个节点能有多少箭头指向其他节点。

具有度为 3 的 B 树，表示一个节点最多有三个子节点，也就是 2-3 树的定义。具有度为 4 的 B 树，表示一个节点最多有四个子节点，也就是 2-3-4 树的定义。

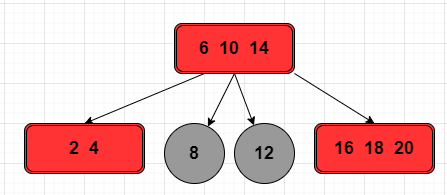


图 6‑1 B树

# 红黑树

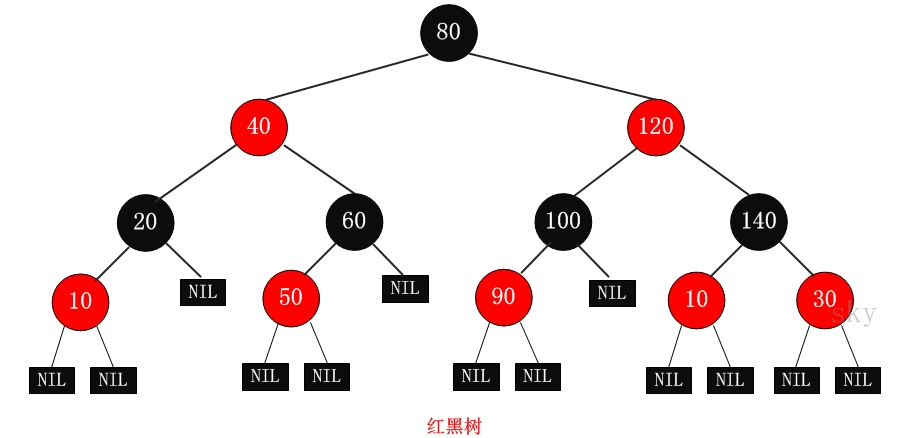


图 7‑1 红黑树

R-B Tree，全称是 Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。

红黑树的每个节点上都有存储位表示节点的颜色，可以是红（Red）或黑（Black）。

## 红黑树与2-3-4树对比

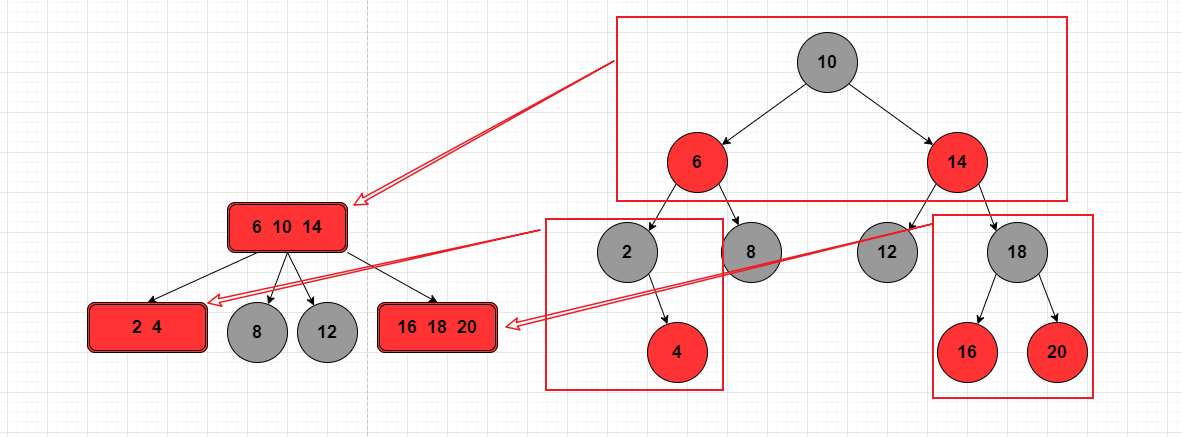


图 7‑2 红黑树与2-3-4树对比

该红黑树与上文讲到的 2-3-4 树对比，是否发现，红黑树就是一个 2-3-4 树：

* 每个节点或者是黑色，或者是红色。
* 根节点是黑色。
* 每个叶子节点（NIL）是黑色。注意：这里叶子节点，是指为空（NIL 或NULL）的叶子节点！
* 如果一个节点是红色的，则它的子节点必须是黑色的。由于红黑树的每个节点都是由 2-3-4 树转化而来的，从而红色节点不能连续两个出现，不然会出现 4 节点的情况，导致违反了规则 2。

而且红黑树的每一个黑节点都是 3 节点中的最中间的那个值，或者是 2 节点中其中一个值。

* 从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

## 红黑树黑节点

**原因：**红黑树这些黑色节点在 2-3-4 树中代表的是由 1 节点的一个 2-3-4 树，而 2-3-4 树是同一个子树的深度是相同的，平衡的，所以从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。

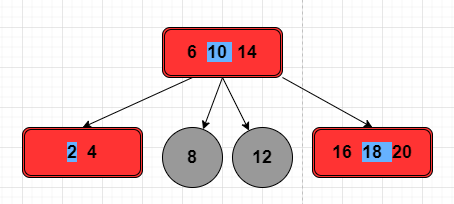


图 7‑3 红黑树黑节点

蓝色代表是黑色节点。

注意如下几点：

* 特性（3）中的叶子节点，是只为空（NIL 或 null）的节点。
* 特性（5），确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。
* 红黑树虽然本质上是一棵二叉查找树，但它在二叉查找树的基础上增加了着色和相关的性质使得红黑树相对平衡，从而保证了红黑树的查找、插入、删除的时间复杂度最坏为 O(log n)。

由上面的例子所示，我们只要把红黑树当做是 2-3-4 树来处理，并且对应的颜色进行改变或者进行左旋右旋的操作，即可达到使得红黑树平衡的目标。

## 红黑树与2-3-4树节点转化

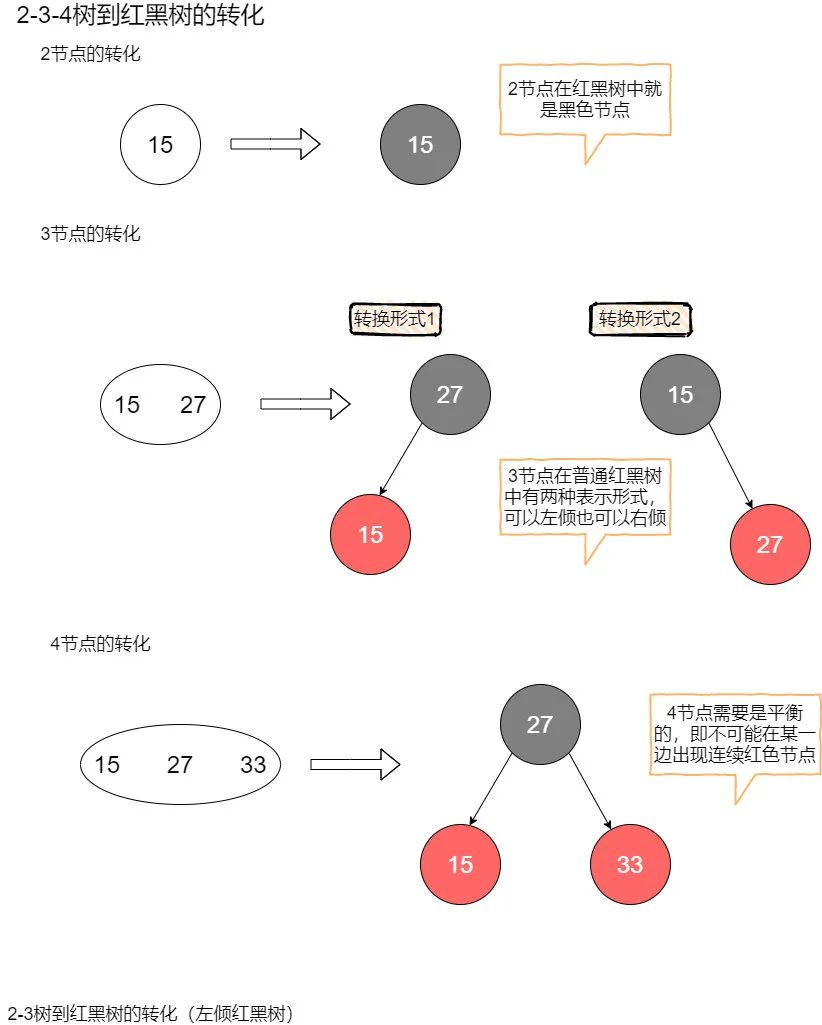


图 7‑4 节点转化

## 红黑树与2-3树节点转化

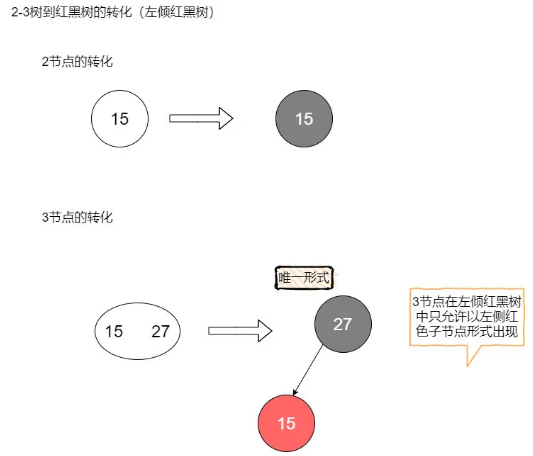


图 7‑5 节点转化

光看单个节点的转化可能还不够明显，我制作了一张红黑树转2-3树的示意图，很清晰地描绘了它们之间的关系。

只要把左倾红黑树中的红色节点顺时针方向旋转45°使其与黑父平行，然后再将它们看作一个整体，你就会发现，这不就是一颗2-3树吗？

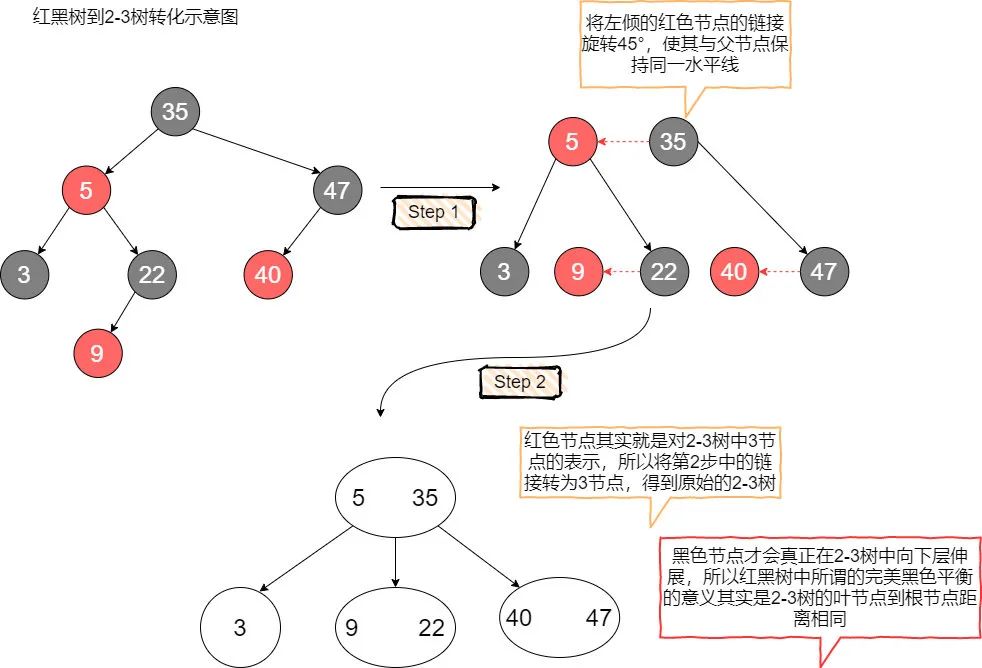


图 7‑6 节点对比

## 2-3树操作

我们在了解红黑树的插入删除操作之前，需要先了解2-3树的插入删除操作，这样才能理解红黑树中染色和旋转背后的意义。

### 插入

让我们来看一下对于2-3树的插入。我们的插入操作需要遵循一个**原则**：先将这个元素尝试性地放在**已经存在的节点中**，如果要存放的节点是2节点，那么插入后会变成3节点，如果要存放的节点是3节点，那么插入后会变成4节点（**临时**）。然后，我们对可能生成的临时4节点进行分裂处理，使得临时4节点消失。

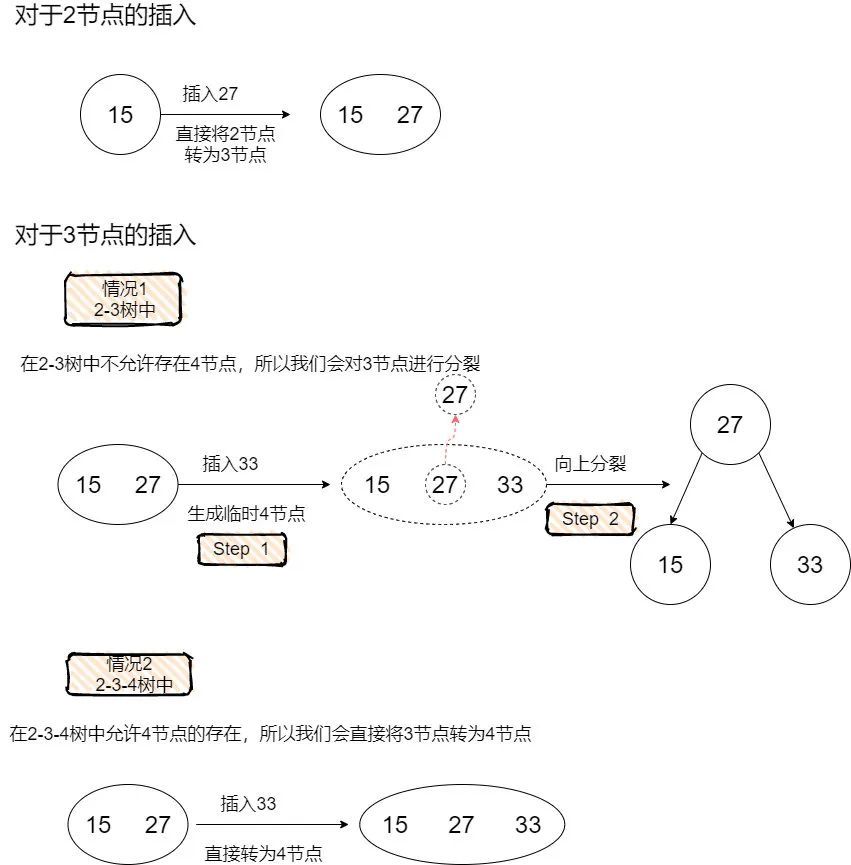


图 7‑7 插入操作

如果需要在2-3-4树中向4节点内插入元素，那么会引发如**下图所示**的分裂过程

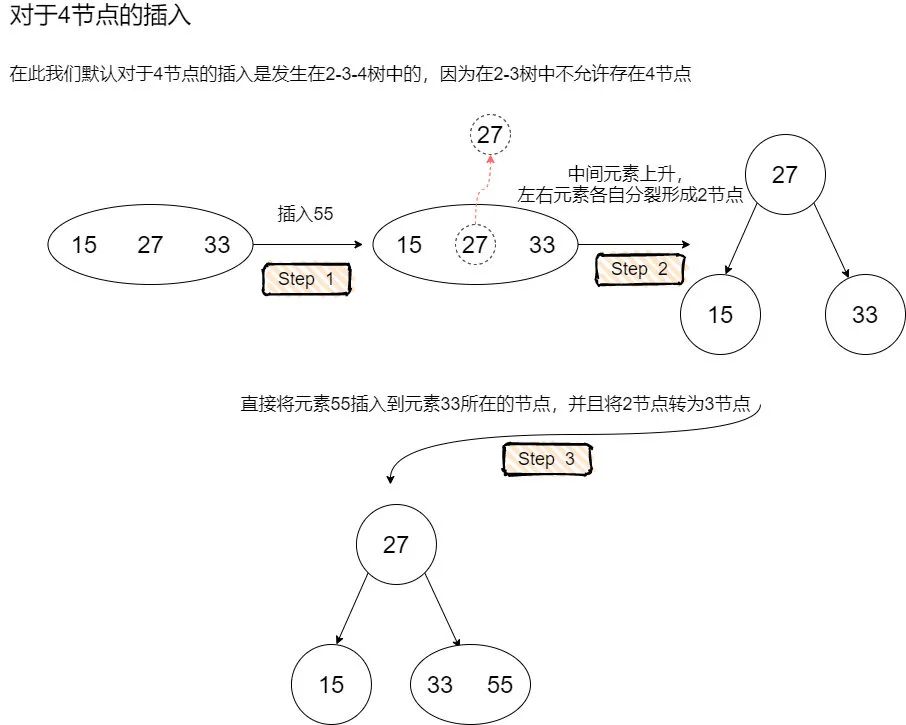


图 7‑8 4节点插入操作

事实上，这正对应了**红黑树在插入的时候一定会把待插入节点涂成红色**，因为红色节点的意义是**与父节点进行关联**，形成概念模型2-3树中的3节点或者临时4节点。

而红黑树之所以需要在插入后进行调整，正是因为可能存在着**概念模型中的临时4节点**（反应在红黑树中是双红的情况）。

试想在2-3树中如果待插入节点是个2节点，那么反应在红黑树中，不正好对应着黑色父节点吗，在黑色父节点下面增加一个红色儿子，确实不会违背红黑树的任何规则，这也对应着我们向2-3树中的2节点插入一个元素，只需要简单的把2节点变成3节点。

### 删除

接下来让我们来看一下对于2-3树的删除。对于2-3树的删除我们主要要考虑待删除元素在2节点这种情况，因为如果待删除元素在3节点，那么可以直接将这个元素删除，而不会破坏2-3树的任何性质（删除这个元素不会引起高度的变化）。

当待删除元素在2节点的时候，由于删除这个元素会导致2节点失去自己**唯一的元素**，引发2节点自身的删除，会使得树中某条路径的高度发生变化，树变得**不平衡**。

因此我们有两种方案去解决这个问题：

* 第一种方案，先删除这个2节点，然后对树进行平衡调整。
* 第二种方案，我们想办法让这个被删除的元素不可能出现在2节点中。

本文选择第二种方案，我们在搜索到这个节点的路径中，不断地判断当前节点是否为2节点，如果是，就从它的兄弟节点或者它的父节点借一个元素，使得当前节点由2节点成为一个3节点或者一个临时4节点（视具体情况而定，在后面的红黑树部分会详细介绍）。

这种操作会产生一种结果：**除非当前节点是根节点，否则当前节点的父节点一定是一个非2节点**（因为搜索的路径是自上而下，父节点已经进行过了这种操作，所以不可能是2节点），那么我们可以保证到达叶子节点的时候，也能顺利的从父节点或者兄弟节点处借到元素，使得自己成为非2节点。从而能够直接删除某个元素（现在这个元素不在2节点中了）。

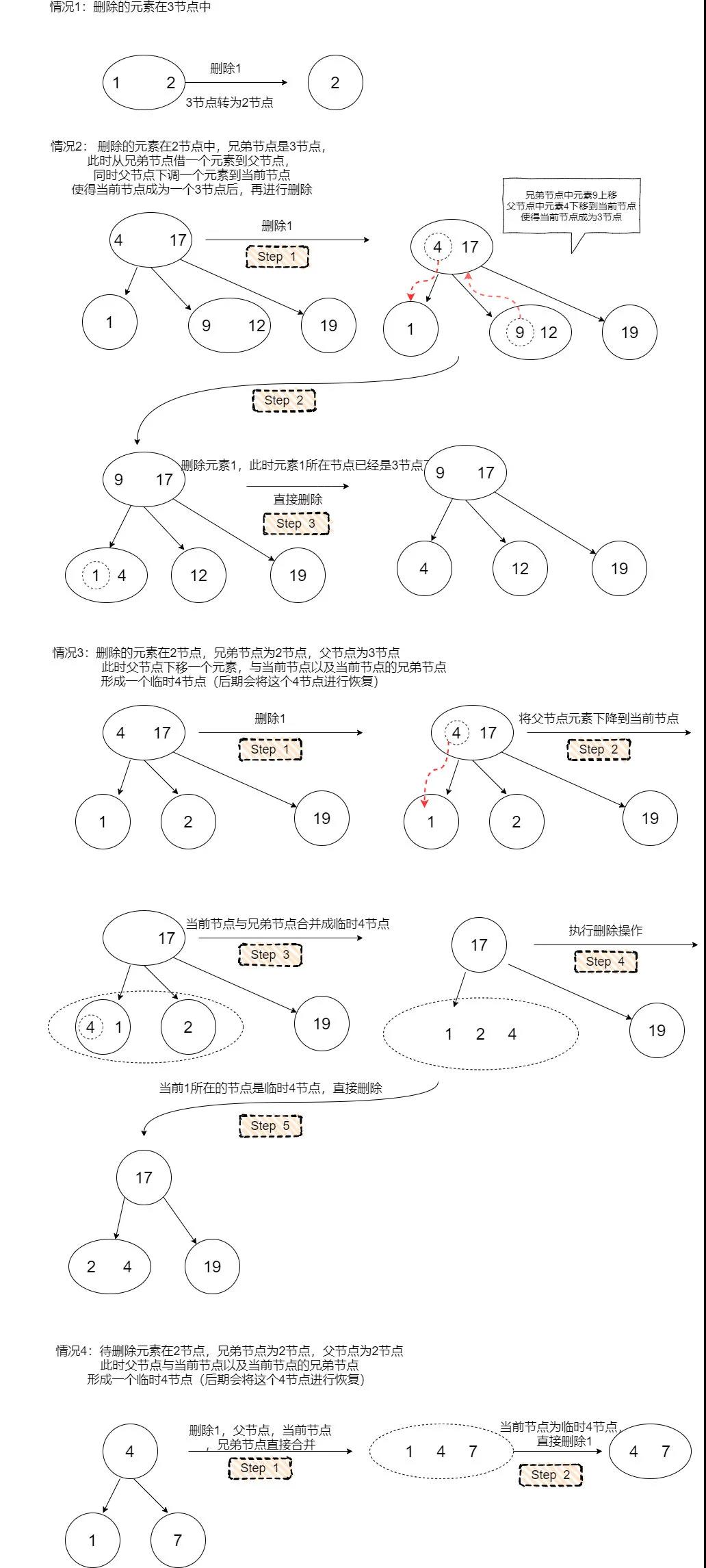


图 7‑9 删除操作

## 红黑树定义

1.节点颜色有红色和黑色

【2-3树到红黑树的转化已经解释过】

2.根节点必为黑色

【2-3树中如果根节点为2节点，那么它本来就对应红黑树中黑节点；如果根节点为3节点，也可以用黑色节点表示较大的那个元素，然后较小的元素作为左倾红节点存在于红黑树中】

3.所有叶子节点都是黑色

【此处提到的叶子其实是空链接，因篇幅问题不便全部画出】

4.任意节点到叶子节点经过的黑色节点数目相同

【红黑树中的红节点是和黑色父节点绑定的，在2-3树中本来就是同一层的，只有黑色节点才会在2-3树中真正贡献高度，由于2-3树的任一节点到空链接距离相同，因此反应在红黑树中就是黑色完美平衡】

5.不会有连续的红色节点

【2-3树中本来就规定没有4节点，2-3-4树中虽然有4节点，但是要求在红黑树中体现为一黑色节点带两红色儿子，分布左右，所以也不会有连续红节点】

相信在你的视角中，红黑树已经不再是这五条僵硬的定义了，它背后正浮现着一颗2-3树概念模型。虽然你已经有了这样的认识，但是红黑树作为真正的实现模型，我们还是要回到这个实现本身来探究它的一系列操作。

**红黑树在插入的时候一定会把待插入节点涂成红色。**

## 红黑树调整

树的旋转分为左旋和右旋，下面借助图来介绍一下左旋和右旋这两种操作。

### 左旋

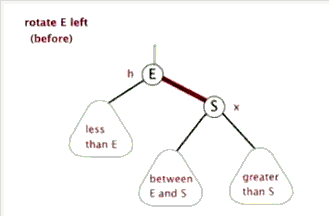


图 7‑10 旋转前

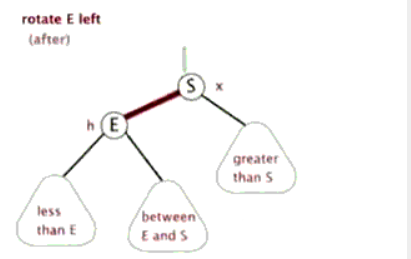


图 7‑11 左旋

如上图所示，当在某个目标结点 E 上，做左旋操作时，我们假设它的右孩子 S 不是 NIL。

左旋以 S 到 E 之间的链为“支轴”进行，它使 S 成为该子树的新根，而 S 的左孩子则成为 E 的右孩子。

### 右旋

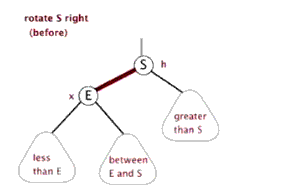


图 7‑12 旋转前

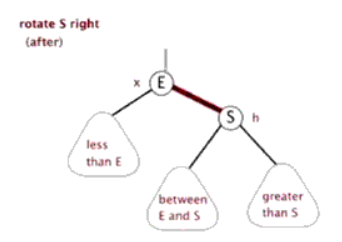


图 7‑13 右旋

同左旋类似，当在某个目标结点 S 上，做右旋操作时，我们假设它的右孩子 S 不是 NIL。

左旋以 S 到 E 之间的链为“支轴”进行，它使 S 成为该子树的新根，而 S 的左孩子则成为 E 的右孩子。

### 添加

对二叉树进行平衡调整是很重要的一个环节，无论是AVL还是红黑树，它们本质上都是希望尽可能保证这颗二叉查找树中的元素尽量均衡的分布在树的两侧。

当我们向一颗二叉查找树中插入一个元素Y的时候，我们会一直与树中的节点进行大小**比较**，如果Y小于当前元素，就往左走，如果Y大于当前元素，就往右走，直到达到叶子节点，这个时候我们可以把Y插入这颗二叉查找树了。

由于这次的插入动作，整棵树可能会发生一些不平衡，因此我们需要在插入后进行一次**平衡调整**，使得整棵树恢复到平衡的状态（具体如何调整，要看树是AVL还是红黑树亦或是其他的平衡树）。

### 删除

二叉查找树的删除是一个很有意思的问题，不同于插入的是，待删除的元素并不能保证一定出现在树中的叶子节点。这将带来一个棘手的情景，即我们需要从树的中间部分取走一个元素，而且在取走后还需要经过调整来使得**整颗树满足平衡**的性质。从树的中间部分直接取走一个节点的场景实在是太多，也牵扯到了太多相关的节点，这种操作很难实现。

好在有人提出了一个观点，我们对查找树中一个节点的删除，其实可以不必真的改动这个节点的位置。由于查找树的特殊性质，将某个元素节点删除后，它有两个最佳替代者，分别是有序序列中的**前驱元素和后继元素**。

我们还是以一个包含元素1~10的二叉查找树为例，如果我们希望删除5所在的节点，那么让4或者6替代它的位置都是可行的。作为前驱元素的4，会存放在5所在节点的左子树的最右侧；作为后继元素的6，会存放在5所在节点的右子树的最左侧。

现在我们又让问题简化了，也就是说，删除某个节点的时候，我们先找到它的前驱元素或者后继元素（随便选一个），将它的前驱元素直接填到待删除的节点，然后再把它的前驱元素或者后继元素删除。

这个时候问题就转化成了在二叉查找树中删除一个没有左子树的节点（或者是一个没有右子树的节点），我们只需要将这个节点删除再进行对应的平衡调整即可（虽然还是需要调平，但是比直接在树中层删除一个同时具备左右儿子的节点要容易很多）。

注意，此处并没有强调是**针对**红黑树的操作，因为红黑树和AVL都是**二叉查找树**，它们都适用这个方法。

### 左倾红黑树插入

**添加就要添加红色节点，红色节点不影响黑色节点高度。**

如下图对于左倾红黑树的插入一共有三种可能的情况。

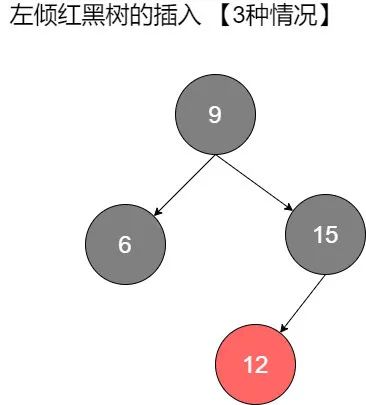


图 7‑14 插入前

#### 情况一：

待插入元素比黑父大，插在了黑父的右边，而黑父左边是红色儿子。这种情况会导致在红黑树中出现右倾红节点。

注意，这种情况对应着2-3树中出现了**临时4节点**，我们在2-3树中的处理是将这个临时4节点分裂，左右元素各自形成一个2节点，中间元素**上升**到上层跟父节点结合。所以，我们在红黑树中的动作是，将原本红色的左右儿子染黑（左右分裂），将黑父染红（等待上升结合）。

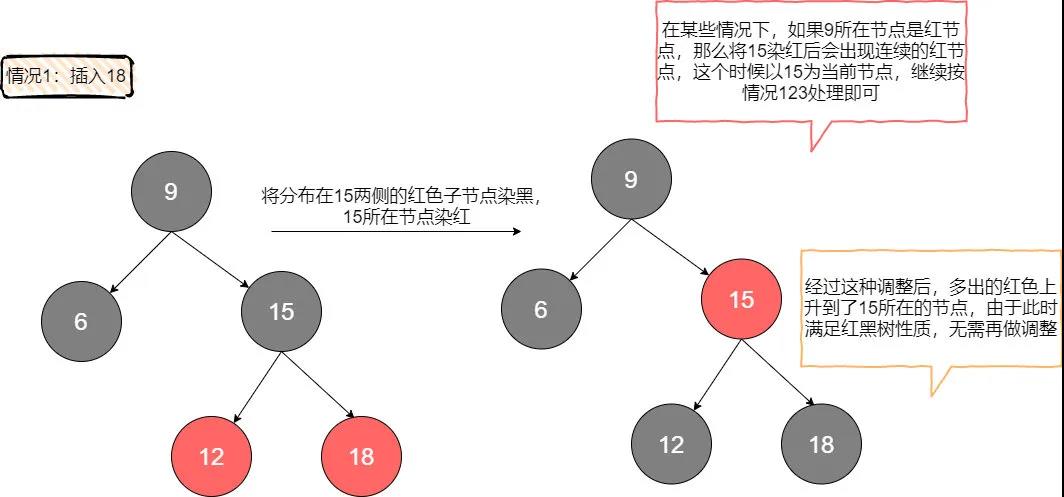


图 7‑15 插入后与左子节点同时反色（后左旋或其他处理）

#### 情况二：

待插入元素比红父小，且红父自身就是左倾。听起来有点绕，看图就会明白，其实就是说红父和待插入元素**同时靠在了左边**，形成了连续的红节点。

这种情况我们需要用两步来调整。由于我们插入的是红色节点，其实不会破坏黑色完美平衡，所以要注意的是在旋转和染色的过程种继续保持这种**完美黑色平衡**。

首先对红父的父亲进行一次右旋，这次右旋不会破坏黑色平衡，但是也没有解决连续红色的问题。

接下来将12所在节点与15所在节点交换颜色，这样的目的是为了消除连续红色，并且这个操作依旧维持了黑色平衡。现在我们已经得到了情况1的场景，直接按**情况1**处理即可。

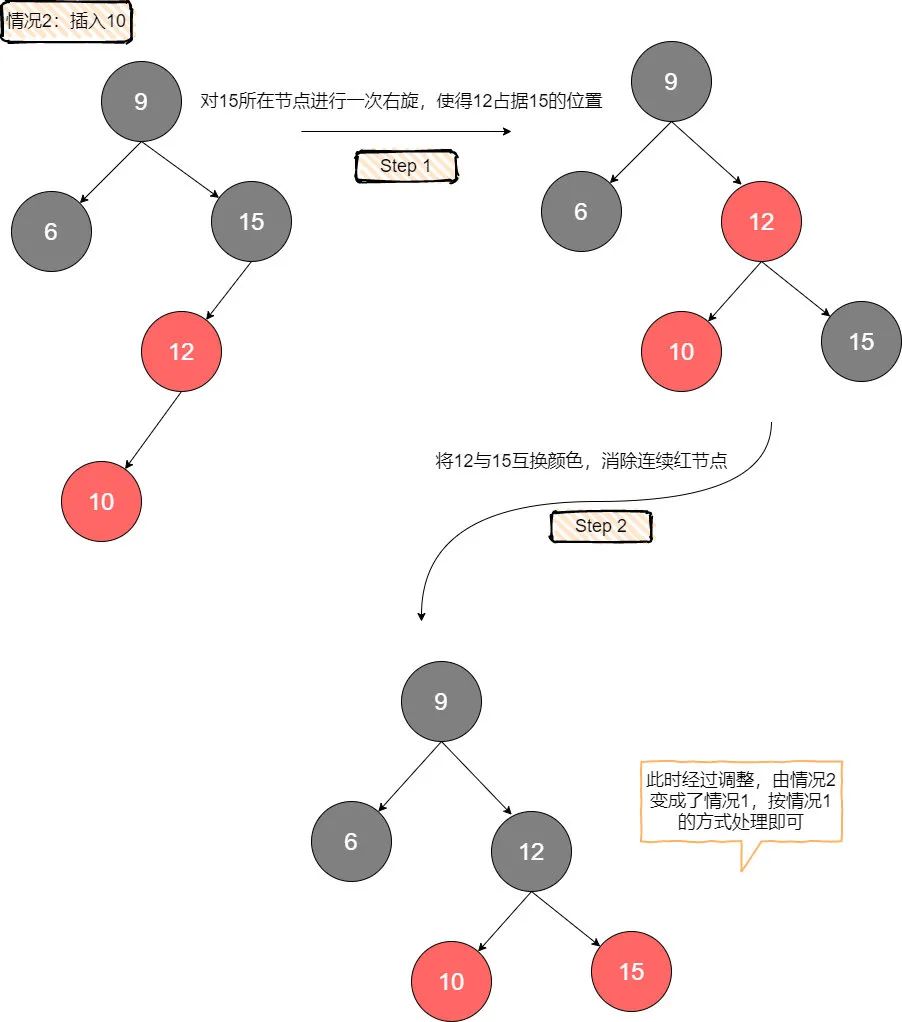


图 7‑16 插入后右旋对掉色后按情况一处理

#### 情况三：

待插入元素比红父大，且红父自身就是左倾。

也就是说插入的这个节点形成了一个右倾的红色节点，对右倾的处理很简单，将红父进行一次左旋，就能使得右倾红节点变为左倾，现在出现了连续的左倾红节点，直接按情况2处理即可。

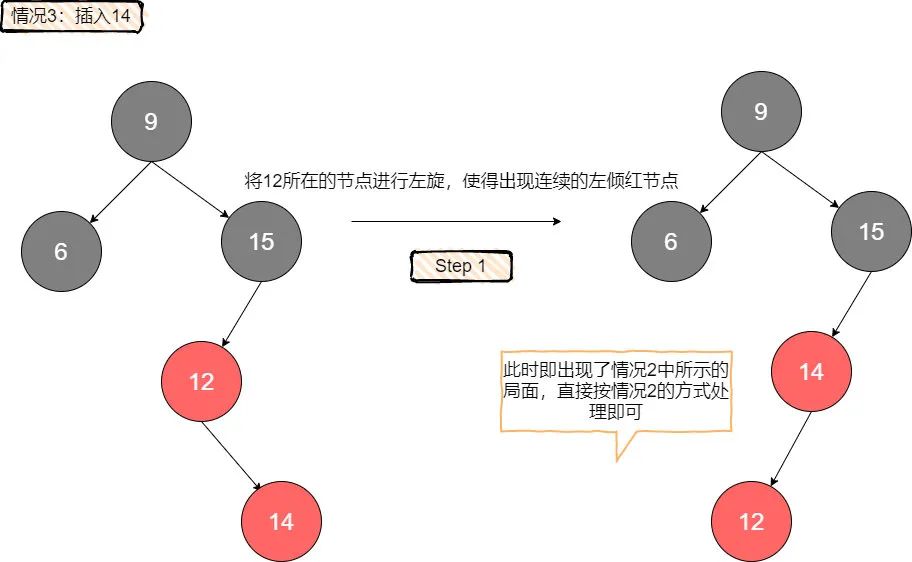


图 7‑17 插入后左旋后按情况二处理

在插入时，可以体会到左倾红黑树对于左倾的限制带来的好处，因为在原树符合红黑树定义的情况下，如果父亲是红的，那么它**一定左倾**，同时也不用考虑可能存在的右倾兄弟（如果有，那说明**原树不满足红黑树定义**）。

这种限制消除了很多需要考虑的场景，让插入变得更加简单。

### 左倾红黑树删除

**删除需要删红色节点，因为不影响黑色节点高度。**

左倾红黑树的删除需要借鉴上文中提到的**二叉查找树通用的删除策略**，当我们要删除某个节点的时候选择它的前驱节点或者后继节点元素来**替代**它，转而删除它的前驱/后继节点。

但从策略上来看需要把对整体进行调整，把**需要删除的前驱/后继节点变为红色节点**。推荐用后继节点进行替代，因为是左倾的，是红色节点的概率大。

在这个例子中，我选择用后继节点来替代被删除节点。

假设我们需要删除的节点它的右子树如图所示，那么对该节点的删除实际上转为了对2的删除。

我们从当前的根节点出发，利于2-3树中预合并的策略逐层对红黑树进行调整。具体的做法是，每次都保证当前的节点是2-3树中的非2节点，如果当前节点已经是非2节点，那么直接跳过；如果当前节点是2节点，那么根据兄弟节点的状况来进行调整：

* 如果兄弟是2节点，那么从父节点借一个元素给当前节点，然后与兄弟节点一起形成一个临时4节点。
* 如果兄弟是非2节点，那么兄弟上升一个元素到父节点，同时父节点下降一个元素到当前节点，使得当前节点成为一个3节点。

这样的策略能够保证最后走到待删除节点的时候，它一定是一个非2节点，我们可以直接将其元素删除。

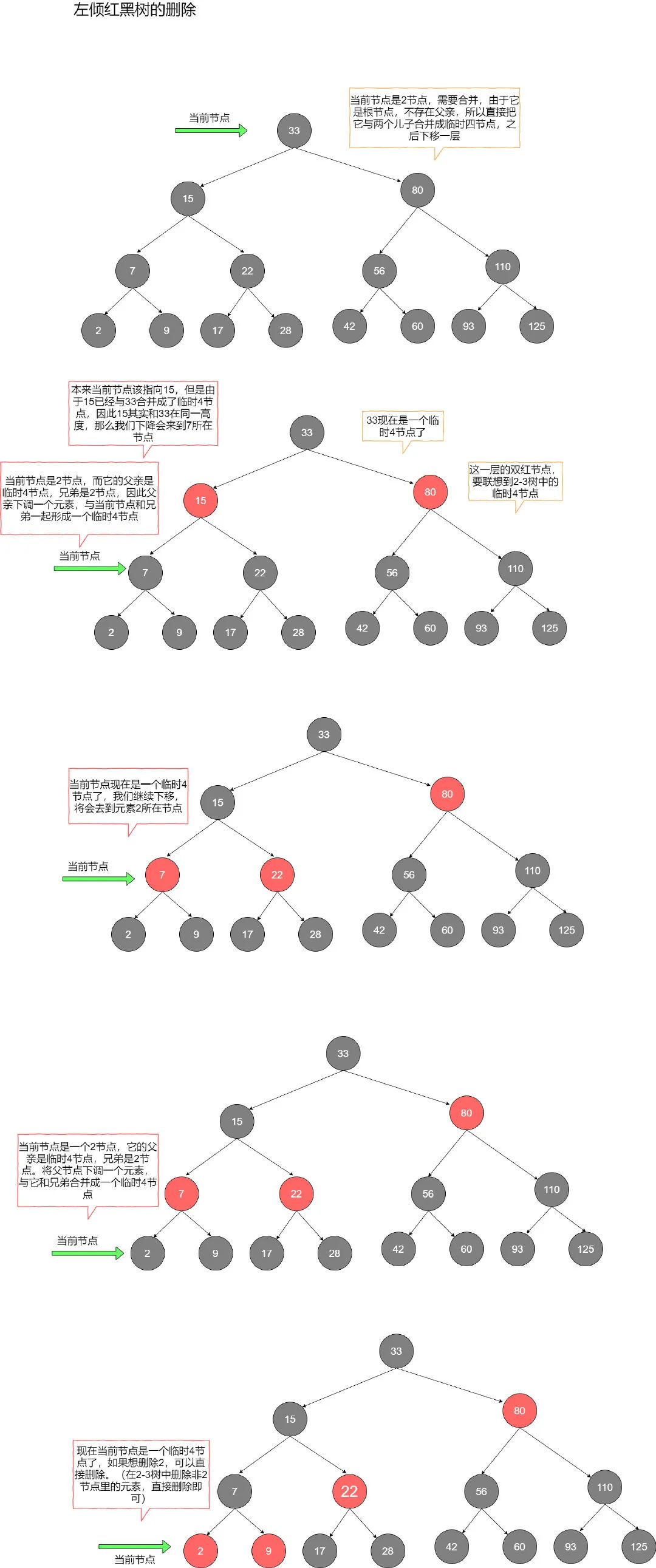


图 7‑18 删除操作

接下来要考虑的是**修复工作**，由于红黑树定义的限制，我们在调整的过程中出现了一些本不该存在的**红色右倾节点**（因为生成了概念模型中的临时4节点），于是我们顺着搜索的方向向上**回溯**，如果遇到当前节点具备右倾的红色儿子，那么对当前节点进行一次左旋，这时原本的右儿子会来到当前节点的位置，然后将右儿子与当前节点交换颜色，我们就将右倾红节点修复成了左倾红节点，同时我们并没有破坏黑色节点的平衡。

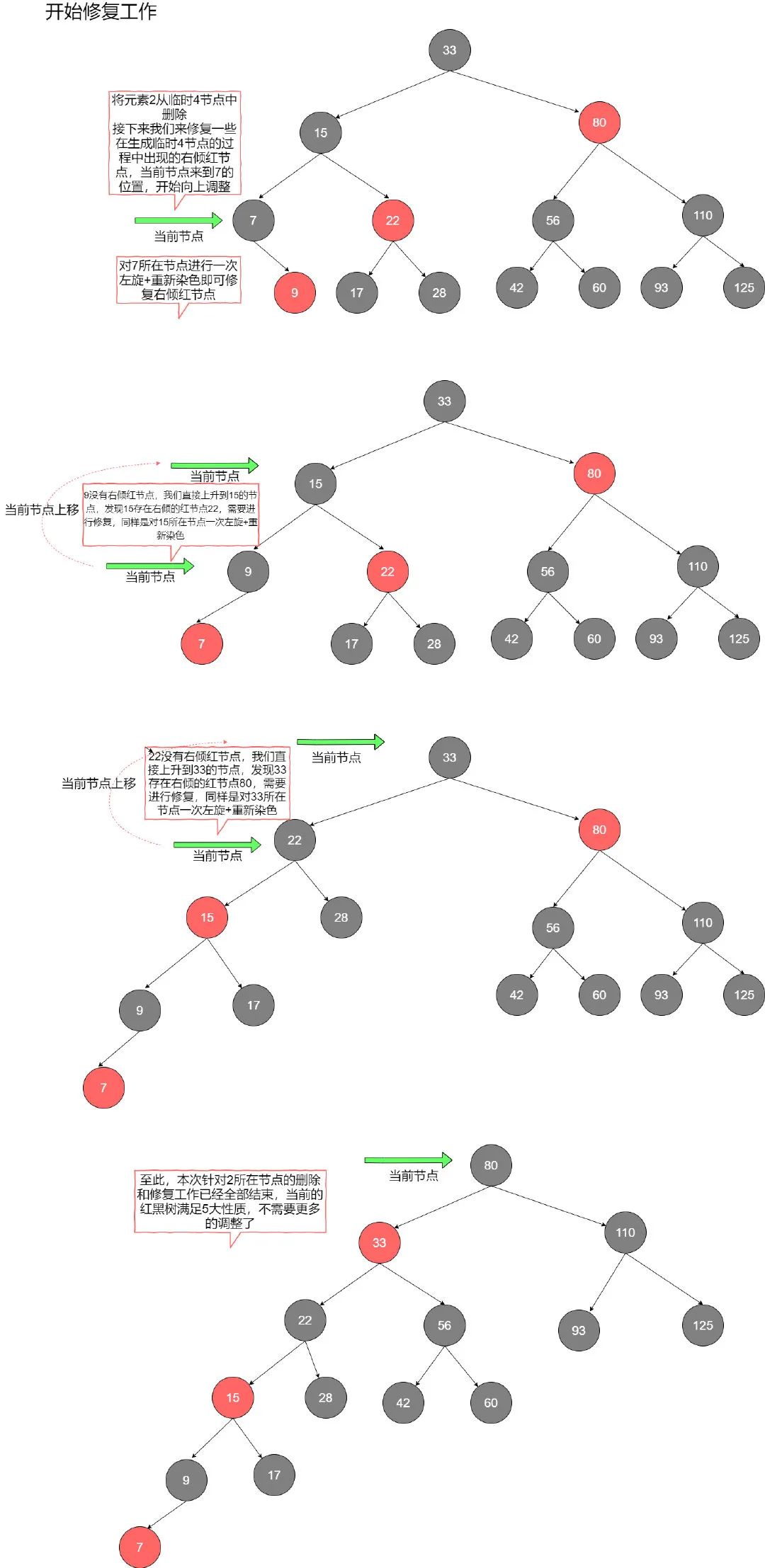


图 7‑19 节点调整

右倾转左倾是一个很基本的操作，我们以35，44为例，你既可以将35作为黑节点，44作为右倾红色儿子；也可以将44作为黑节点，35作为左倾红儿子。事实上我们对于右倾的修复就是换了一种**树形**而已。一路回溯到当前根节点，直至路径中不再包含**任何**的红色右倾节点，至此修复工作全部完成。