中国科学技术大学

算法分析与设计

主讲人： 庄连生

Email: { *lszhuang@ustc.edu.cn* }

***Fall* 2023，USTC**

第七讲
平衡树专题

- 二叉搜索树
- AVL树
- 红黑树及扩张
- B树/B+树/B*树

University of Science and Technology of China

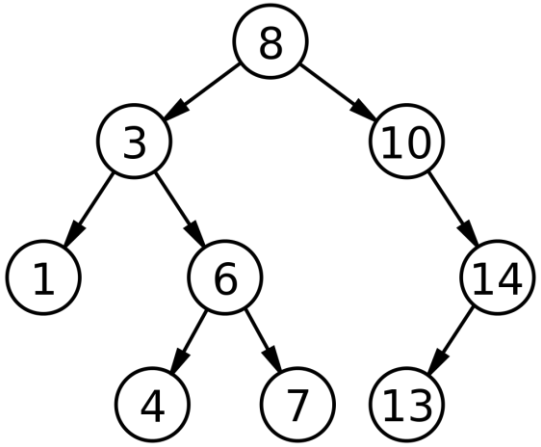
二叉搜索树-概念

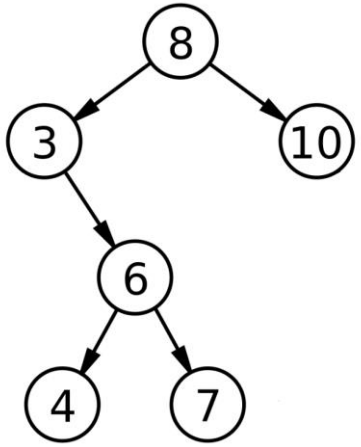
定义：二叉搜索树，也称为二叉查找树、有序二叉树或排序二叉树，是指一棵空树或者具有下列性质的二叉树。

① 若任意节点的左子树不空，则左子树上所有节点的值均小于它根节点的值；

② 若任意节点的右子树不空，则右子树上所有节点的值均大于或等于它根节点的值；

③ 任意节点的左、右子树也分别为二叉查找树。





二叉搜索树-概念

为什么使用二叉查找树这种数据结构?

	有序数组	链表	(平衡) 二叉查找树
搜索	$O(\log(n))$	$O(n)$	$O(\log(n))$
删除	$O(n)$	$O(n)$	$O(\log(n))$
插入	$O(n)$	$O(1)$	$O(\log(n))$

元素插入操作

基本步骤：

① 若T是空树，则将s所指节点作为根节点插入，返回；

② 若s->data小于T的根节点的数据域之值：

- 若T的左子树为空，则把s所指节点作为T的左子树插入，返回；
- 在T的左子树中递归寻找插入点；

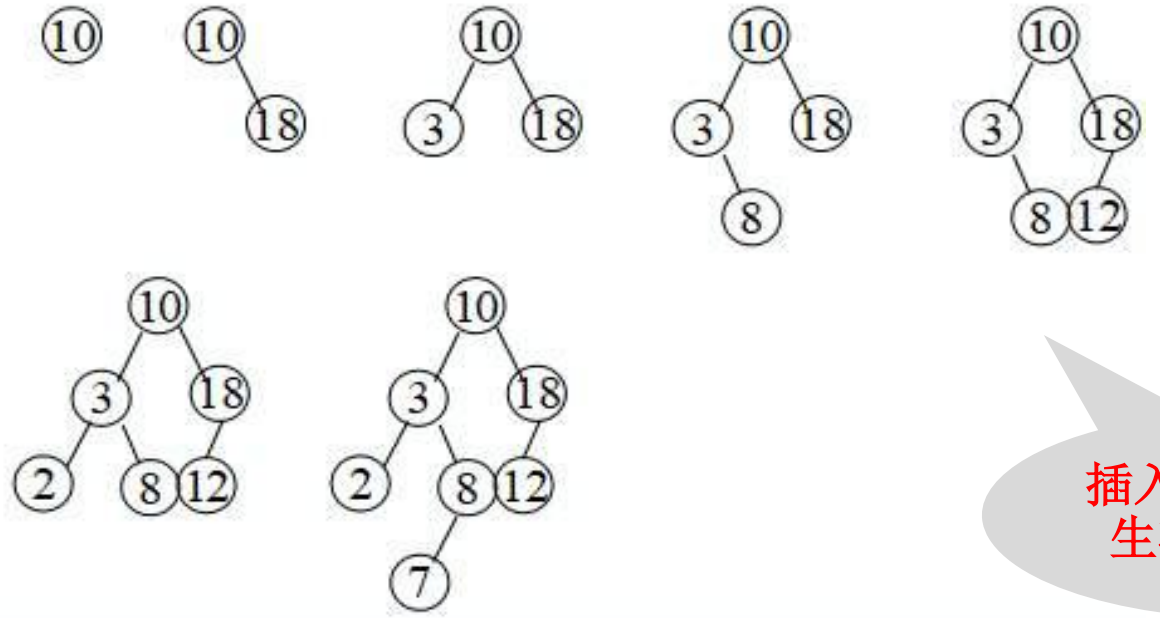
③ 若s->data大于等于T的根节点的数据域之值：

- 若T的右子树为空，则把s所指节点作为T的右子树插入，返回；
- 在T的右子树中递归寻找插入点。

元素插入操作

基本步骤：

{10，18，3，8，12，2，7}



插入操作一定发生在叶子节点

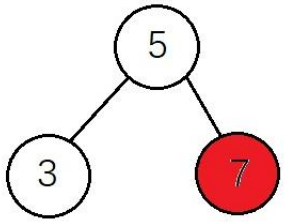
二叉搜索树的构造过程即为一系列元素的插入过程！

元素删除操作

基本步骤：

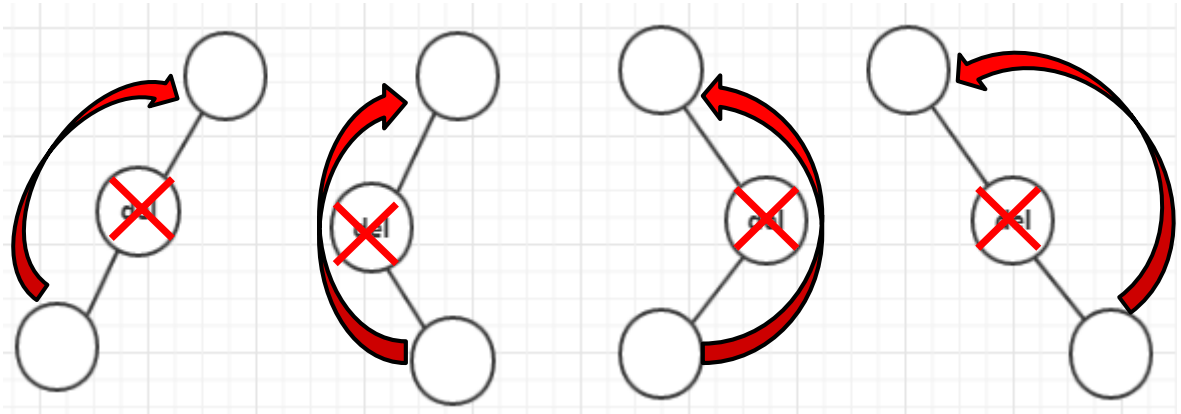
① **Case 1:** 删除叶子结点

- ✓ 直接删除



② **Case 2:** 删除带有一个子节点的节点

- ✓ 将待删除节点的左/右子树 赋值给 待删除节点的父节点的左/右子树



元素删除操作

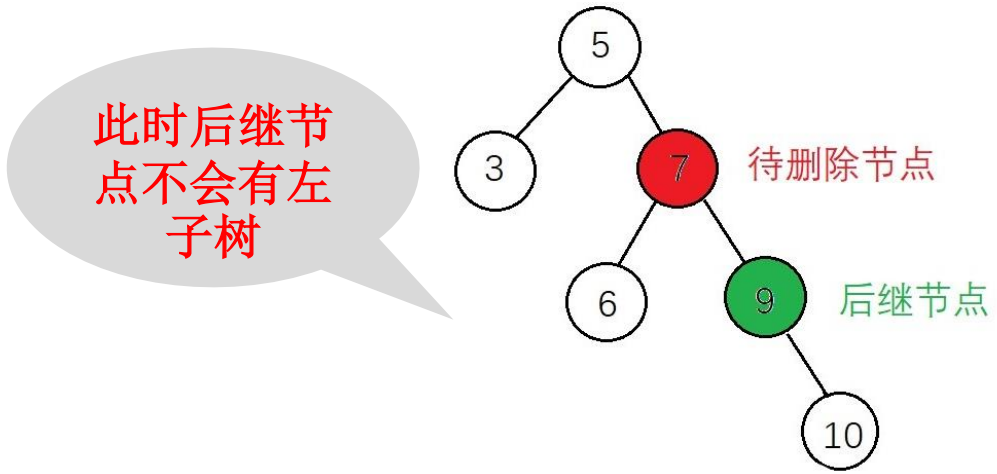
基本步骤：

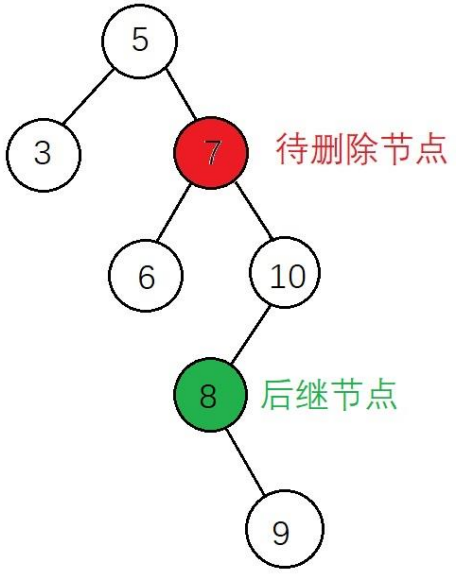
① **Case 3:** 删除带有两个子节点的节点

A. 首先需要找到待删除节点的后继节点和该后继节点的父节点；

B. 删除节点的后继节点一定是删除节点右子树的最左侧节点，我们将采用后继节点替代待删除节点，分为两种情况：

- ✓ 后继节点是待删除节点的子节点（左图）
- ✓ 后继节点不是待删除节点的子节点（右图）





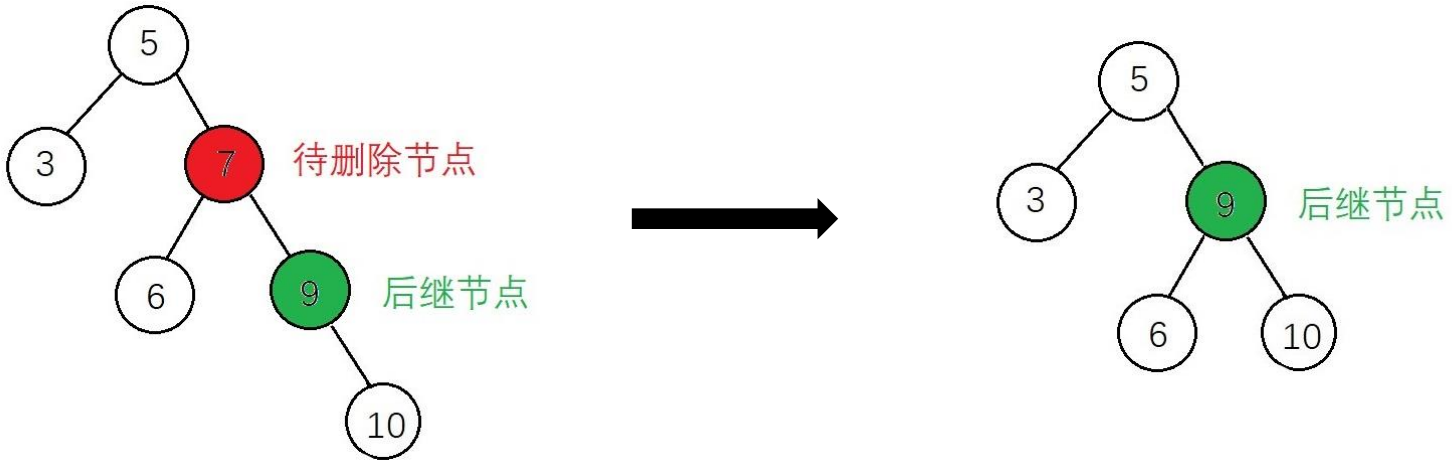
元素删除操作

基本步骤：

① **Case 3:** 删除带有两个子节点的节点

✓ 后继节点是待删除节点的子节点（左图）

在后继节点为待删除节点的子节点的前提下，该后继节点有右子树和没有右子树的操作是相同的，都是将 后继节点 替代 待删除节点，并将待删除节点的左子树赋值给后继节点的左子树；



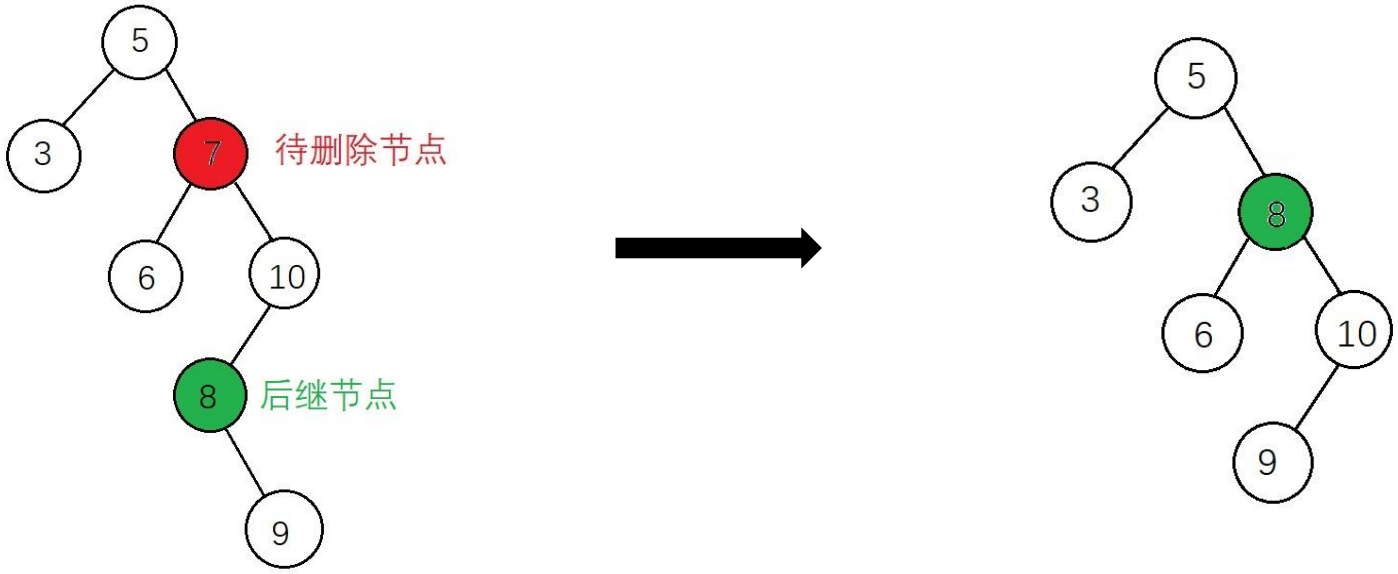
元素删除操作

基本步骤：

① **Case 3:** 删除带有两个子节点的节点

✓ 后继节点不是待删除节点的子节点（右图）

此时与上面的后继节点没有右子节点相比需要增加一个操作，需要将后继节点的右子树 赋值给 后继节点的父节点的左子树；



时间复杂度分析

构造二叉搜索树：

- 用一组数值建造一棵二叉搜索树的同时，也把这组数值进行了排序。其平均时间复杂度为$O(n\log n)$，最差时间复杂度为$O(n^2)$。
  - ✓ 例如，若该组数值已经是有序的（从小到大），则建造出来的二叉查找树的所有节点，都没有左子树。此时构造时间复杂度为$O(n^2)$。

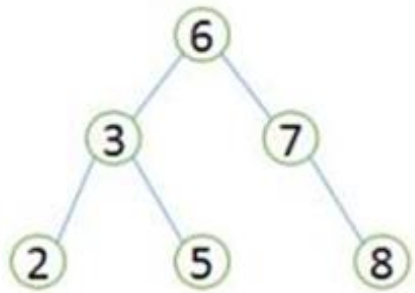
- 我们可以通过随机化建立二叉搜索树来尽量的避免这种情况，但是在进行了多次的操作之后，由于在删除时，我们总是选择将待删除节点的后继代替它本身，这样就会造成总是右边的节点数目减少，以至于树向左偏沉。这同时也会造成树的平衡性受到破坏，提高它的操作的时间复杂度。

时间复杂度分析

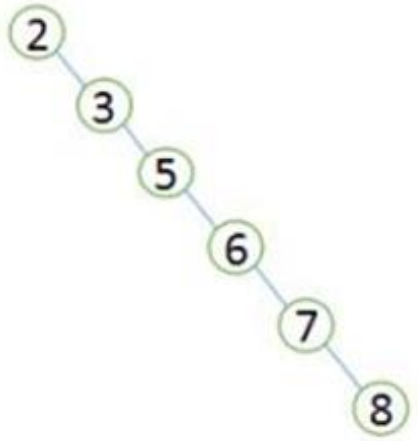
搜索二叉搜索树：

- 最坏情况下，当先后插入的关键字有序时，构成的二叉查找树蜕变为单支树，树的深度为元素个数n，其平均查找长度为$(n+1)/2$（和顺序查找相同）。

- 最好的情况是二叉查找树的形态和二分查找的判定树相同，其平均查找长度为$O(\log n)$。



最好情况



最差情况

删除元素：

- 从一棵二叉搜索树中删除一个元素时，已知寻找待删除元素所需平均时间为O(log n)，最差时间为$O(n)$。删除操作所需时间为$O(1)$。故而删除某个元素所需总时间平均时间复杂度为$O(\log n)$，最差时间复杂度为$O(n)$。

用大O符号表示的时间复杂度

算法	平均	最差
空间	$O(n)$	$O(n)$
搜索	$O(\log n)$	$O(n)$
插入	$O(\log n)$	$O(n)$
删除	$O(\log n)$	$O(n)$

第七讲
平衡树专题

- 二叉搜索树
- AVL树
- 红黑树及扩张
- B树

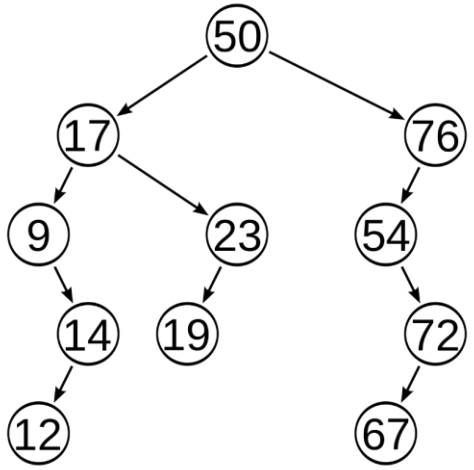
University of Science and Technology of China

AVL树--概念

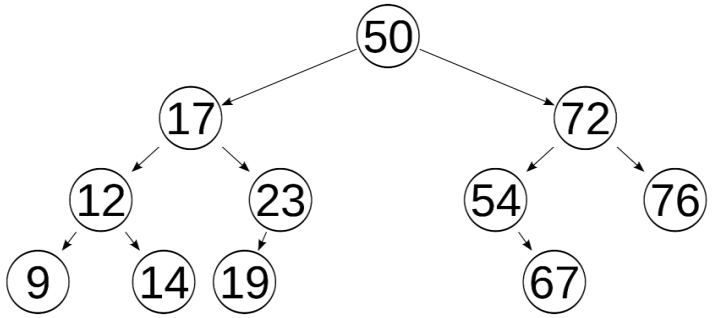
定义：

在计算机科学中，AVL树是最早被发明的自平衡二叉搜索树。AVL树得名于它的发明者G. M. Adelson-Velsky和 Evgenii Landis。它具有以下特点：

- ✓ 它是一棵空树或它的左右两个子树的高度差的绝对值不超过1；
- ✓ 左右两个子树都是一棵AVL树。



非AVL树的例子

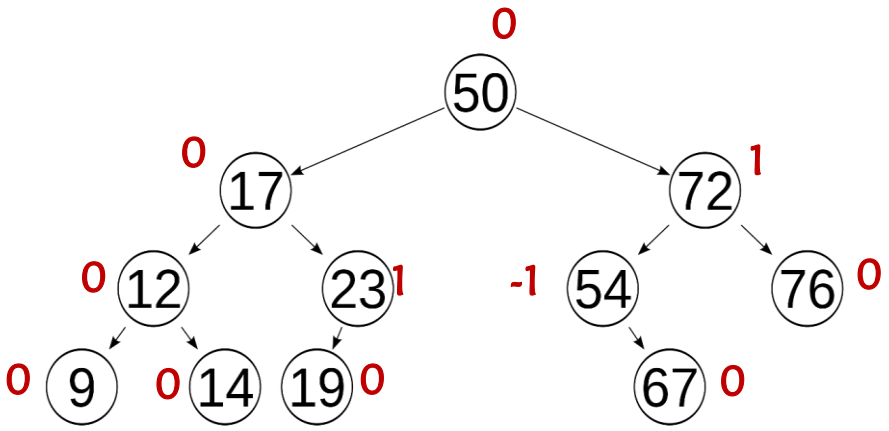


同一个树在高度平衡之后的样子

AVL树--概念

平衡因子： 节点的平衡因子是它的左子树的高度减去它的右子树的高度（有时相反）。

- ✓ 带有平衡因子1、0或 -1的节点被认为是平衡的。
- ✓ 带有平衡因子 -2或2的节点被认为是不平衡的，并需要重新平衡这个树。
- ✓ 平衡因子可以直接存储在每个节点中，或从可能存储在节点中的子树高度计算出来。



AVL树--概念

能够保持二叉搜索树性质的旋转操作：

- ✓ 左旋
- ✓ 右旋

```
T1, T2 and T3 are subtrees of the tree rooted with y (on left side)
or x (on right side)
                y                               x
               / \     Right Rotation          /  \
              x   T3   - - - - - - - >        T1   y
             / \       < - - - - - - -            / \
            T1  T2     Left Rotation            T2  T3
Keys in both of the above trees follow the following order
      keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.
```

AVL树—元素插入操作

AVL树插入操作 = 二叉搜索树插入操作 + 平衡旋转操作

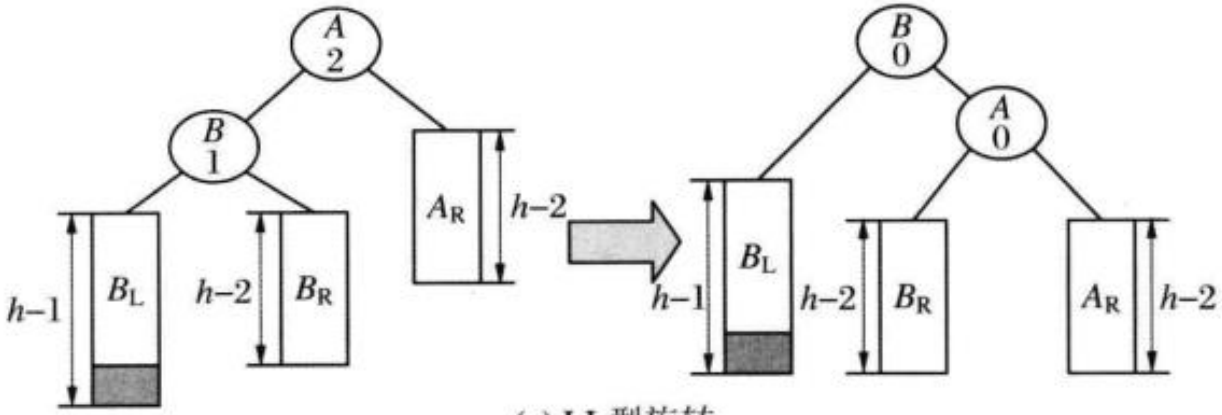
- 一个节点的插入可能会导致原本平衡的二叉树很多节点的平衡因子变成2或-2，此时我们发现只要调整离插入点最近的一个平衡因子不满足绝对值小于等于1的节点，就可以使二叉树重新处于平衡状态。我们称这个离插入节点最近的且平衡因子不满足绝对值小于1的祖先节点为最小子树根。

- 此时可以分为四种情况：
  ① Case 1：插入节点位于最小子树根的左孩子的左子树上
  ② Case 2：插入节点位于最小子树根的右孩子的右子树上
  ③ Case 3：插入节点位于最小子树根的左孩子的右子树上
  ④ Case 4：插入节点位于最小子树根的右孩子的左子树上

AVL树—元素插入操作

AVL树插入操作 = 二叉搜索树插入操作 + 平衡旋转操作

- 情况1：插入节点位于最小子树根的左孩子的左子树上：
  - ✓ 此时需要围绕最小子树根进行一次向右的旋转，称为LL型旋转

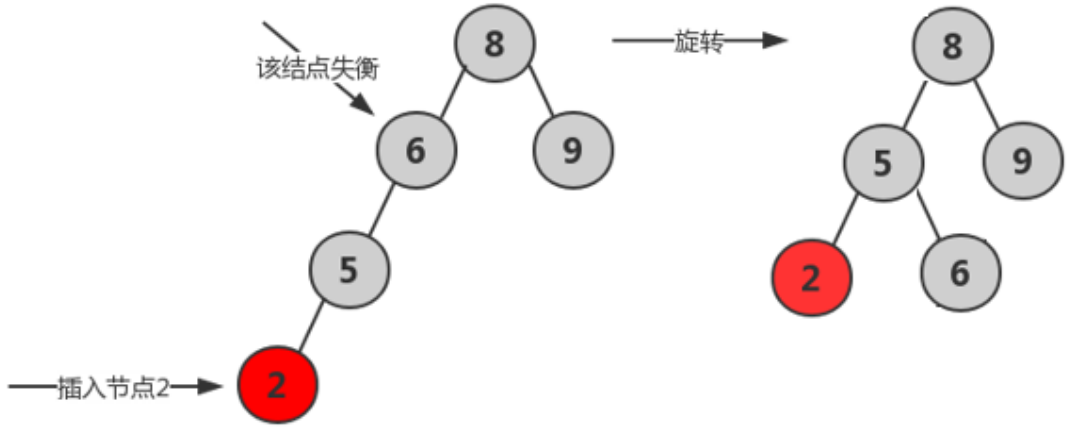


(a) LL型旋转

AVL树—元素插入操作

AVL树插入操作 = 二叉搜索树插入操作 + 平衡旋转操作

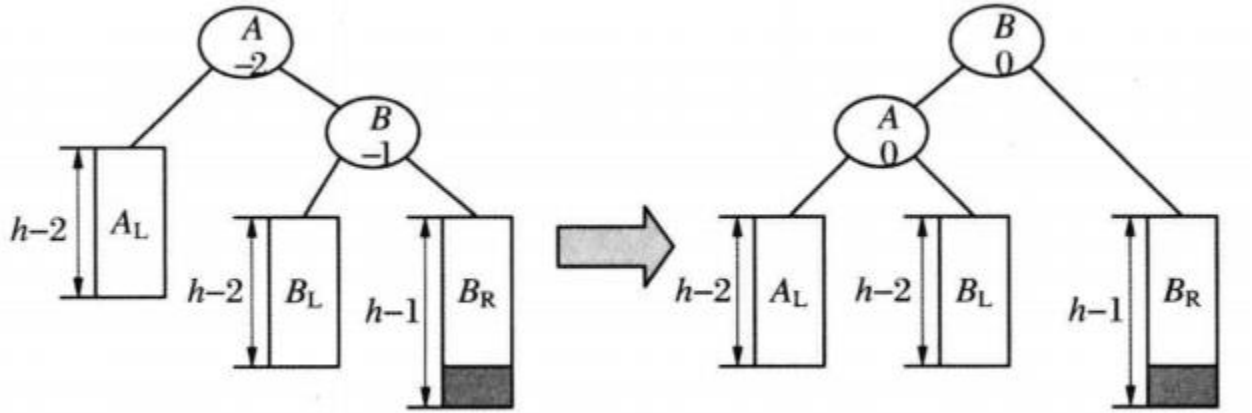
- 情况1：插入节点位于最小子树根的左孩子的左子树上：
  - ✓ 此时需要围绕最小子树根进行一次向右的旋转，称为LL型旋转



AVL树—元素插入操作

AVL树插入操作 = 二叉搜索树插入操作 + 平衡旋转操作

- 情况2：插入节点位于最小子树根的右孩子的右子树上；
  - ✓ 这种情况与第一种情况对称，此时需要围绕最小子树根进行一次向左的旋转，称为RR型旋转

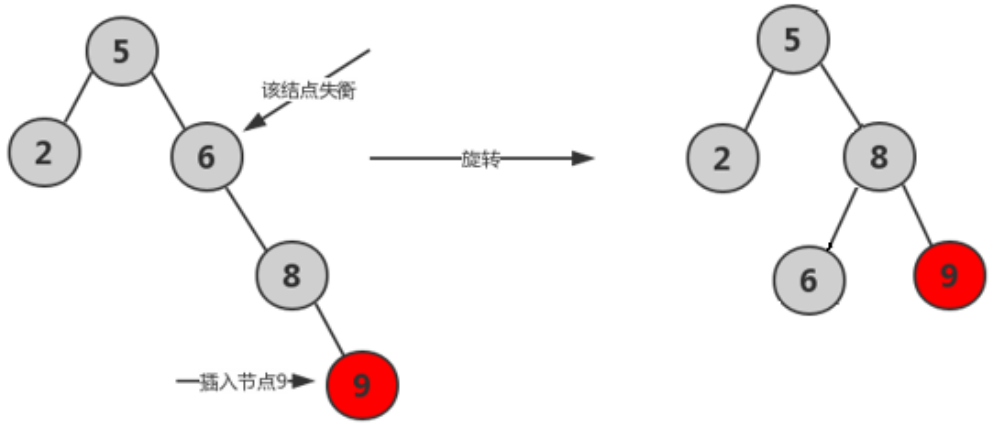


(b) RR型旋转

AVL树—元素插入操作

AVL树插入操作 = 二叉搜索树插入操作 + 平衡旋转操作

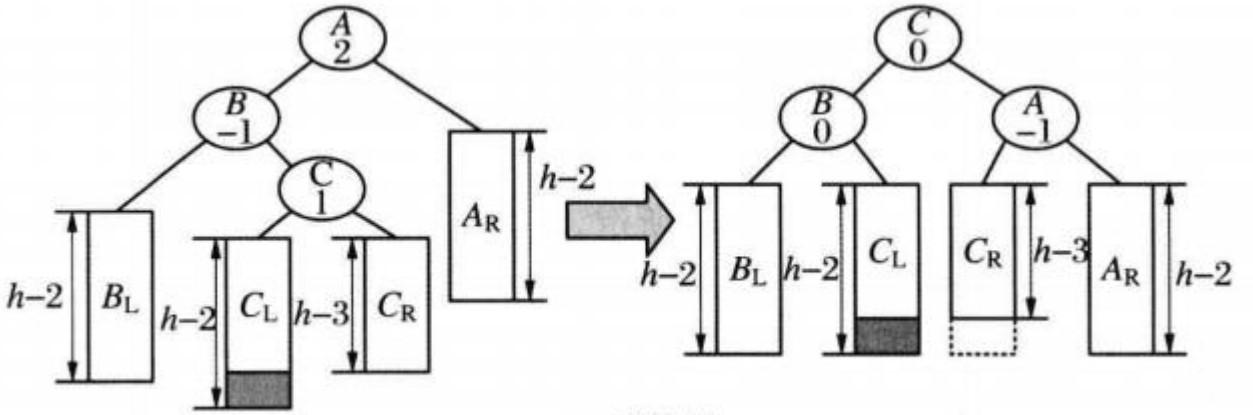
- **情况2**：插入节点位于最小子树根的右孩子的右子树上；
  - ✓ 这种情况与第一种情况对称，此时需要围绕最小子树根进行一次向左的旋转，称为RR型旋转。



AVL树—元素插入操作

AVL树插入操作 = 二叉搜索树插入操作 + 平衡旋转操作

- 情况3：插入节点位于最小子树根的左孩子的右子树上；
  - ✓ 这时需要进行两次旋转，先围绕左孩子做一次向左的旋转，再围绕子树根做一次向右的旋转，称为LR旋转

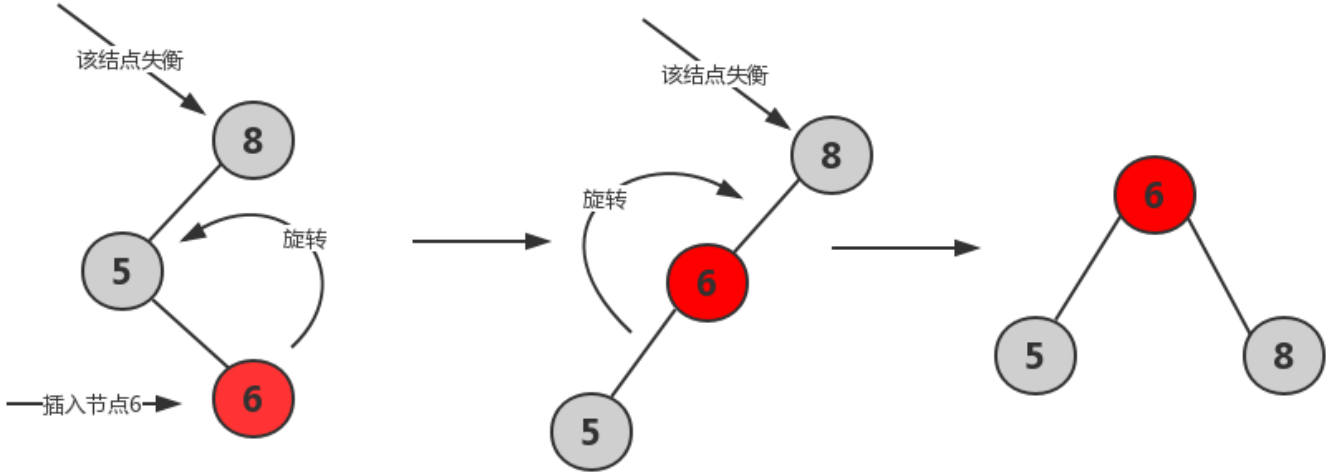


(c) LR型旋转

AVL树—元素插入操作

AVL树插入操作 = 二叉搜索树插入操作 + 平衡旋转操作

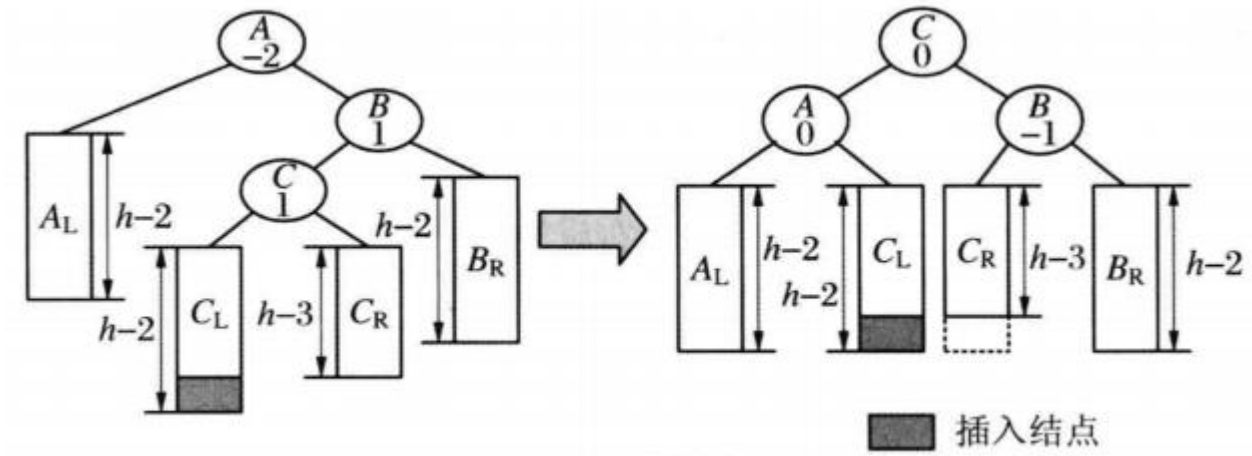
- 情况3：插入节点位于最小子树根的左孩子的右子树上；
  - ✓ 这时需要进行两次旋转，先围绕左孩子做一次向左的旋转，再围绕子树根做一次向右的旋转，称为LR旋转



AVL树—元素插入操作

AVL树插入操作 = 二叉搜索树插入操作 + 平衡旋转操作

- 情况4：插入节点位于最小子树根的右孩子的左子树上；
  - ✓ 这种情况与第三种情况对称，需要进行两次旋转，先围绕右孩子做一次向右的旋转，再围绕子树根做一次向左的旋转，称为RL旋转

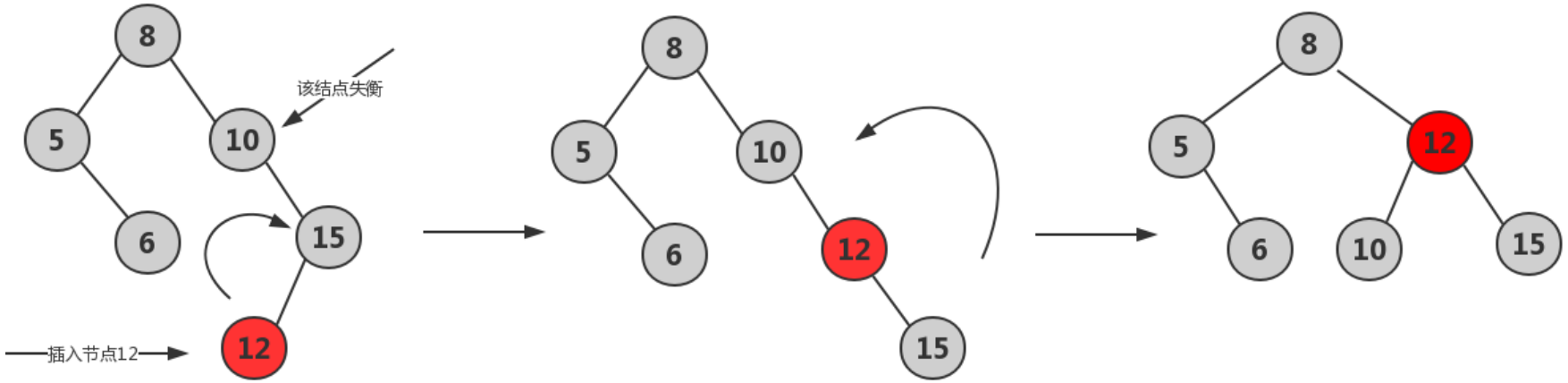


(d) RL型旋转

AVL树—元素插入操作

AVL树插入操作 = 二叉搜索树插入操作 + 平衡旋转操作

- 情况4：插入节点位于最小子树根的右孩子的左子树上；
  - ✓ 这种情况与第三种情况对称，需要进行两次旋转，先围绕右孩子做一次向右的旋转，再围绕子树根做一次向左的旋转，称为RL旋转



AVL树—元素删除操作

AVL树删除操作 = 二叉搜索树删除操作 + (多次)平衡旋转操作

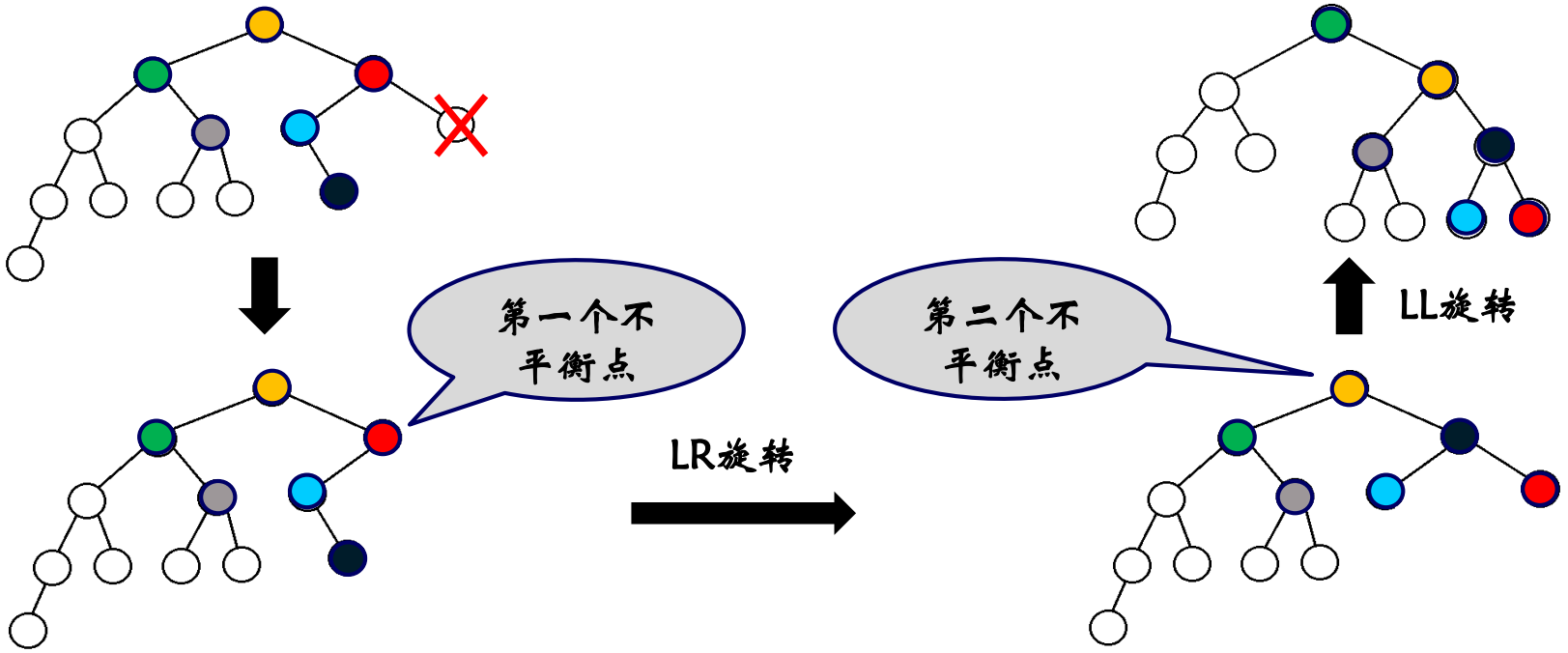
- 一个节点的删除可能会导致原本平衡的二叉树很多节点的平衡因子变成2或-2，此时与插入不同的是，删除操作的平衡旋转操作会稍微复杂一点，可能会需要多次的旋转操作；
- **原因：**
  - ✓ 平衡旋转操作会导致子树高度减1，从而可能引发高层父节点不平衡。
  - ✓ 插入操作不会引起上述问题的原因是：插入所引发的不平衡，是子树高增加导致的，会与平衡旋转所产生的子树高减1相抵消。不会引发更高层的祖先节点不平衡。

- 在执行二叉搜索树的删除操作之后，从被删除节点的父节点、或者是被删除节点的后继节点的父节点开始(取决于被删除节点是否为叶子节点)，沿着树向上寻找不平衡节点并执行平衡旋转操作直到根节点。

AVL树—元素删除操作

AVL树删除操作 = 二叉搜索树删除操作 + (多次)平衡旋转操作

- 例子：



AVL树—时间复杂度分析

构造AVL树

- 用一组数值建造一棵AVL树时，其平均时间复杂度为$O(n\log n)$，最差时间复杂度也为$O(n\log n)$。
- 因为：
  - ✓ 由于AVL树始终保持平衡，每插入一个元素，插入点的平均查找时间为$O(k\log k)$，平衡旋转操作的时间为$O(1)$，其中k为当前树中元素个数。故而插入n个元素构成AVL树时，我们所需的时间为$O(n\log n)$。

搜索AVL树

- AVL树的搜索可以像普通二叉搜索树一样的进行。平均时间复杂度总是$O(\log n)$，因为AVL树总是保持平衡的！

AVL树—时间复杂度分析

删除元素

- 删除AVL树中一个元素时，时间复杂度总是$O(\log n)$。
- 因为：
  - ✓ 由于AVL树始终保持平衡，寻找待删除元素的时间为$O(\log n)$，平衡旋转操作的时间为$O(\log n)$（因为可能有多次旋转），故删除某个元素所需的时间为$O(\log n)$。

用大O符号表示的时间复杂度

算法	平均	最差
空间	O(n)	O(n)
搜索	O(log n)	O(log n)
插入	O(log n)	O(log n)
删除	O(log n)	O(log n)

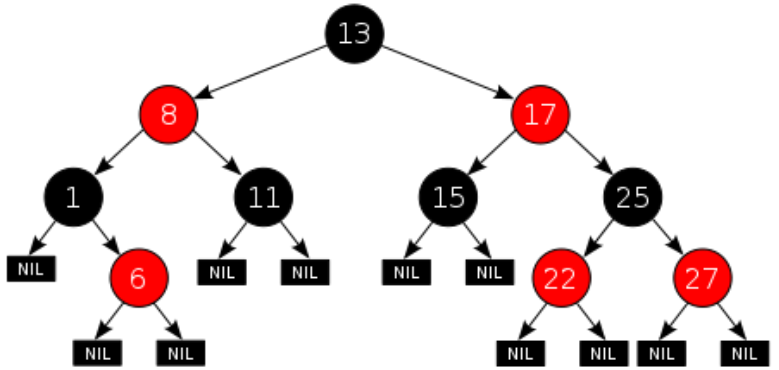
第七讲
平衡树专题

- 二叉搜索树
- AVL树
- 红黑树及扩张
- B树/B+树/B*树

University of Science and Technology of China

红黑树--概念

红黑树和AVL树一样也是一种自平衡二叉搜索树。它现代的名字源于Leo J. Guibas和Robert Sedgewick于1978年写的一篇论文。红黑树的结构复杂，但它的操作有着良好的最坏情况运行时间，并且在实践中高效。

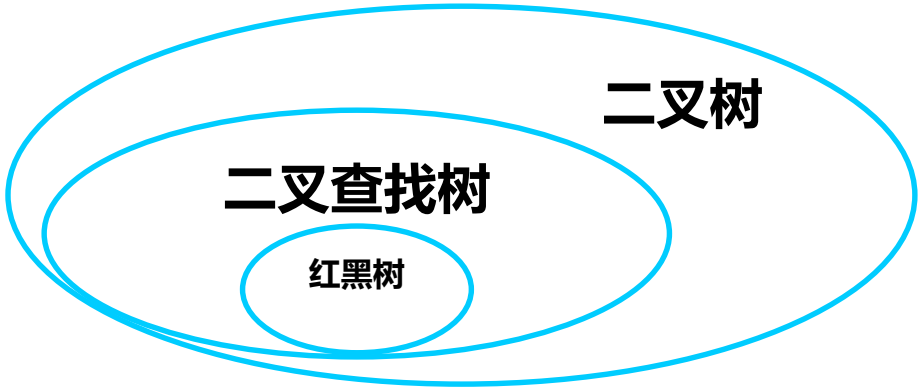


注:

在很多树数据结构的表示中，一个节点有可能只有一个子节点，而叶子节点包含数据。用这种范例表示红黑树是可能的，但是这会改变一些性质并使算法复杂。为此，本文中我们使用"nil叶子"或"空（null）叶子"，如上图所示，它不包含数据而只充当树在此结束的指示。

红黑树—概念

红黑树、二叉查找树、二叉树关系：



- 二叉树性质：
  - ✓ 每个节点最多有两个子树；
- 二叉查找树：
  - ✓ 每个节点左孩子的Key小于它自身的Key；
  - ✓ 每个节点有孩子的Key大于它自身的Key；
- 红黑树：
  - ✓ 对二叉搜索树节点上增加一个存储位表示节点的颜色(红色或黑色)，通过对任何一条从根到叶子的路径上各个节点着色方式的限制，确保没有一条路径会比其他路径长出两倍，从而接近平衡。

红黑树—概念

红黑树的定义如下：

① 每个节点必须为红色或黑色；

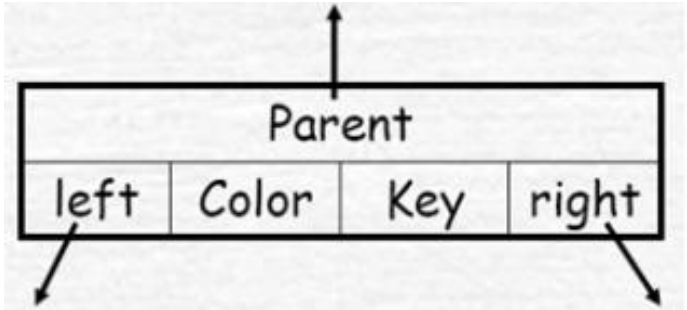
② 根为黑色；

③ 所有叶子节点都是黑色（叶子节点是NIL节点）；

④ 若节点为红，则其两个子节点必为黑色；

⑤ 对每个节点，从该节点到其每个叶子节点的所有简单路径都包含相同数目的黑节点。

节点结构：



红黑树—概念

红黑树与AVL树相比的优点：

① 红黑树不追求“完全平衡”，即不像AVL树那样要求节点的左右孩子高度差小于等于1。它只要求部分达到平衡。提出了为节点增加颜色，红黑树用非严格的平衡来换取增删节点时候旋转次数的降低，任何不平衡都会在三次旋转之内解决，而AVL树是严格平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转次数比红黑树要多。

② 就插入节点导致树失衡的情况，AVL和RB-Tree都是最多两次树旋转来实现复衡rebalance，旋转的量级是$O(1)$。

③ 删除节点导致失衡，最坏情况下AVL需要维护从被删除节点到根节点root这条路径上所有节点的平衡，旋转的量级为$O(logN)$，而RB-Tree最多只需要旋转3次实现复衡，只需$O(1)$，所以说RB-Tree删除节点的rebalance的效率更高，开销更小！

红黑树—概念

黑高的定义：

从某个节点x出发(不包括该节点)到达一个叶子节点的任意一条路径上，黑色节点的个数称为该节点x的黑高度，用$bh(x)$表示。

红黑树高度的定义：

红黑树的黑高度定义为其根节点的黑高度，记为$bh(root[T])$。

引理13.1

一棵n个内节点的红黑树的高度至多为$2\lg(n+1)$

红黑树—概念

引理13.1： **一棵n个内节点的红黑树的高度至多为$2lg(n+1)$**

证明： ① 先证对任何以x为根的子树其内节点数 $\geq 2^{bh(x)}-1$

当$bh(x)=0$时，x就是$nil[T]$，故$2^{bh(x)}-1=2^0-1=0$ 即为0个内节点，正确。

假设对x的左右孩子命题正确，则：

∵ x的左右孩子的黑高或为$bh(x)$或为$bh(x)-1$

∴ x的内节点数 = 左孩子内节点数+右孩子内节点数+1

$$\geq \left(2^{bh(x)-1}-1\right)+\left(2^{bh(x)-1}-1\right)+1$$

$$=\left(2^{bh(x)}-1\right)$$

红黑树—概念

引理13.1： **一棵n个内节点的红黑树的高度至多为$2lg(n+1)$**

证明： ① 先证对任何以x为根的子树其内节点数 $\geq 2^{bh(x)}-1$

当$bh(x)=0$时，x就是$nil[T]$，故$2^{bh(x)}-1=2^0-1=0$ 即为0个内节点，正确。

假设对x的左右孩子命题正确，则：

∵ x的左右孩子的黑高或为$bh(x)$或为$bh(x)-1$

∴ x的内节点数 = 左孩子内节点数+右孩子内节点数+1

$$\geq \left(2^{bh(x)-1}-1\right)+\left(2^{bh(x)-1}-1\right)+1$$

$$=\left(2^{bh(x)}-1\right)$$

红黑树—概念

引理13.1： **一棵n个内节点的红黑树的高度至多为$2lg(n+1)$**

证明： ② 证明$\boldsymbol{bh(root[T]) \geq h/2}$, 其中$h$为红黑树的高。

∵ 红节点的孩子必为黑

∴ 红节点的层数$< h/2$

∴ $bh(root[T]) \geq h/2$

③ 证明最后结论

∵ 红黑树有n个内节点

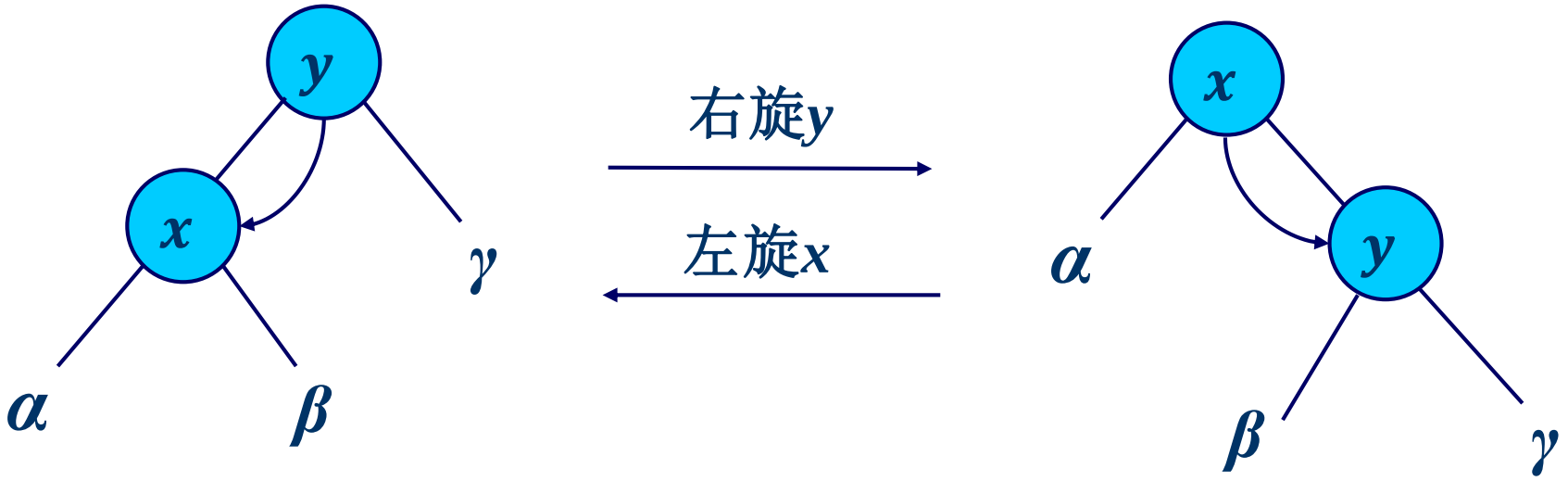
由 ① 知：$n \geq 2^{bh(root[T])} - 1 \geq 2^{\frac{h}{2}} - 1$

∴ $h <= 2\log(n+1)$

引理13.1 证毕。

红黑树—旋转操作

在插入和删除元素时，为保持红黑树的性质，我们同样需要对红黑树进行相应的旋转操作。其原理与AVL树中所介绍的左旋与右旋相同。



注：在旋转过程中保持二叉搜索性质不变：$\alpha \le x \le \beta \le y \le \gamma$。

红黑树—元素插入操作

算法步骤：

① 将节点z按照BST树（二叉搜索树）规则插入红黑树中，z为叶子节点；

② 将节点z涂红；

③ 调整（涂色、旋转操作）使满足红黑树性质。

```
1   RB-INSERT(T, z):
2       y ← nil              '''y用于记录：当前扫描节点的双亲节点'''
3       x ← T.root           '''从根节点开始扫描'''
4       while x ≠ T.nil '''查找插入位置'''
5           do y ← x
6           if z.key < x.key     '''z 插入x的左子树'''
7               then x ← x.left
8           else x ← x.right     '''z 插入x的右子树'''
9       z.p ← y                  '''y 是z的双亲'''
10      if y == nil[T]           '''z 插入空树的情况'''
11          then T.root ← z      '''z 作为根插入'''
12      else if z.key < y.key
13          then y.left ← z
14      else y.right ← z
15      z.left ← T.nil
16      z.right ← T.nil
17      z.color ← RED
18      RB-INSERT-FIXUP(T, z)
```

RB-INSERT-FIXUP用于实现对红黑树进行调整，以保证红黑树的性质

红黑树—元素插入操作

颜色调整思路：

① 通过旋转和改变颜色，自下而上调整（z进行上溯），使树满足红黑树的性质。

插入新节点z后违反红黑树性质的情况：

① z作为红节点，其两个孩子(nil)为黑，不违反性质1, 3, 5

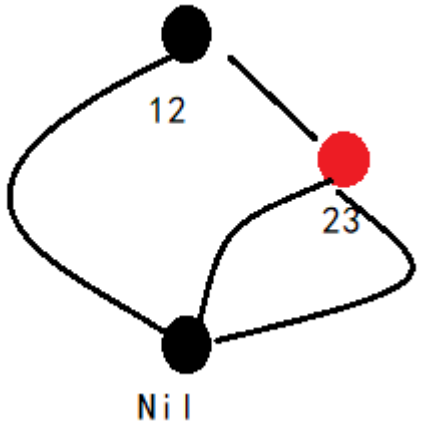
② 可能违反性质2：z如果作为根插入

③ 可能违反性质4：如果z的双亲$p[z]$为红色时

红黑树—元素插入操作

调整步骤：

1. 若作为根插入，则将其涂黑即可；
2. 若z不是根，则$p[z]$存在：
   ① 若$p[z]$为黑，则无需调整；



插入23，符合红黑树的基本性质，无需做出调整

红黑树—元素插入操作

调整步骤：

1. 若作为根插入，则将其涂黑即可；
2. 若z不是根，则$p[z]$存在：
   - ① 若$p[z]$为黑，则无需调整；
   - ② 若$p[z]$为红，违反性质4，则需要调整；

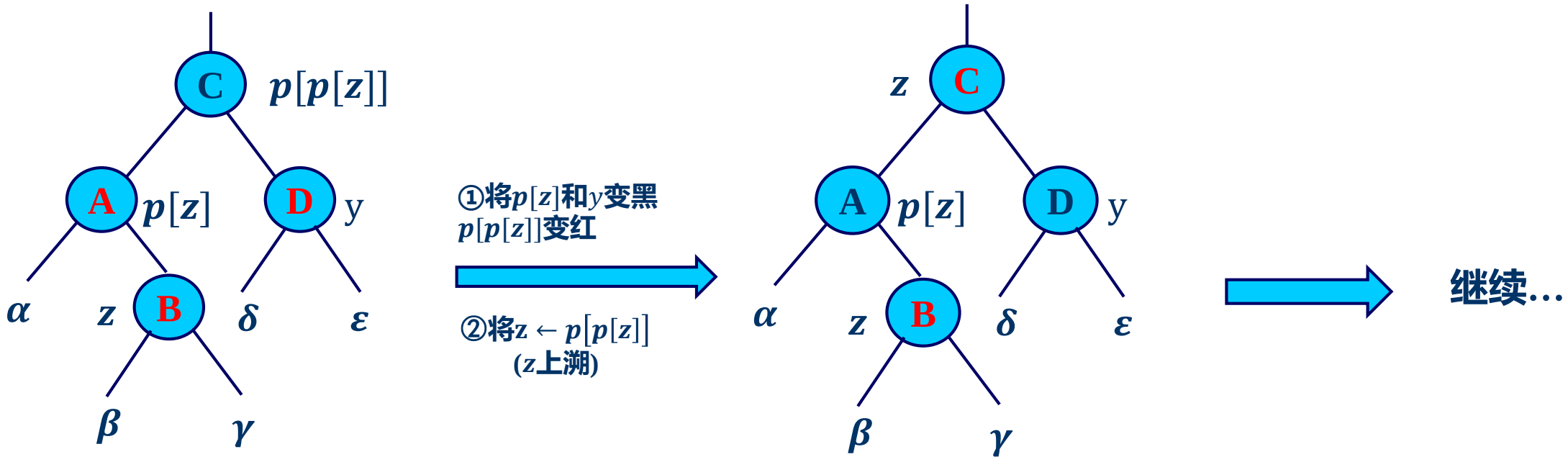
     此时因为$p[z]$为红，故不为根，故$p[p[z]]$存在且必为黑

     此时我们可以分为六种情形去处理：
     - ✓ **case1~3**为z的双亲$p[z]$是其祖父$p[p[z]]$的左孩子
     - ✓ **case4~6**为z的双亲$p[z]$是其祖父$p[p[z]]$的右孩子

case1~3与case4~6对称，这里以case1~3为例进行讲解。

红黑树—元素插入操作

case 1: z的叔叔是红色



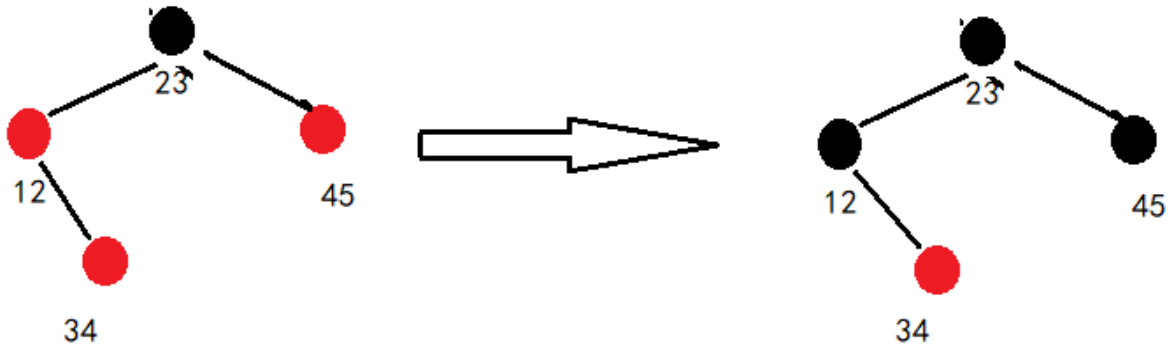
① 变换后，z上溯到祖先节点$p[p[z]]$，新的z可能仍然会违反性质4，我们需要递归调整(新的z可能是case1、case2、case3中的一种)。

② 如果z上溯到根，我们将根直接涂黑，此时树高增加一，调整终止。

红黑树—元素插入操作

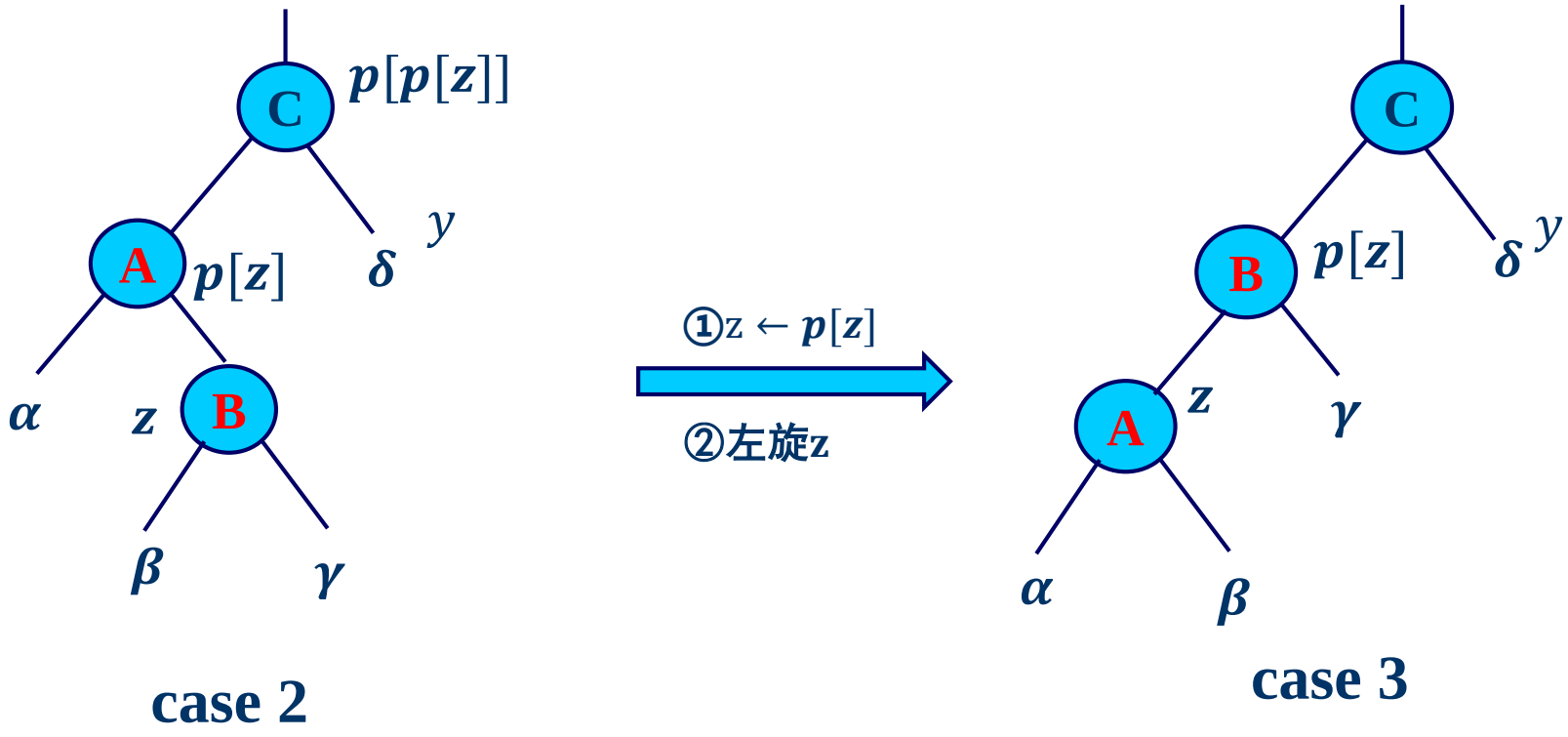
case 1: z的叔叔是红色

插入34，不满足性质4(红色节点一定有两个黑色子节点)，所以将34的父节点和叔叔节点 涂成黑色，祖父节点变成红色，但23是根，必须为黑色，所以如下图所示 23，12，45节点颜色为黑色。



红黑树—元素插入操作

case 2: z的叔叔是黑色，其z是双亲$p[z]$的右孩子



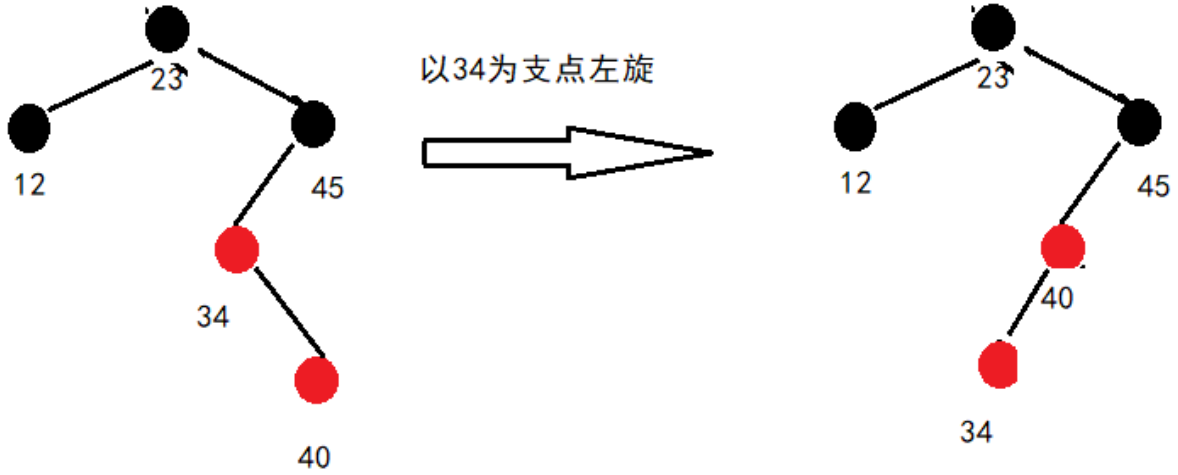
case 2　　case 3

① Case 2可以通过左旋转化为Case 3去解决。

红黑树—元素插入操作

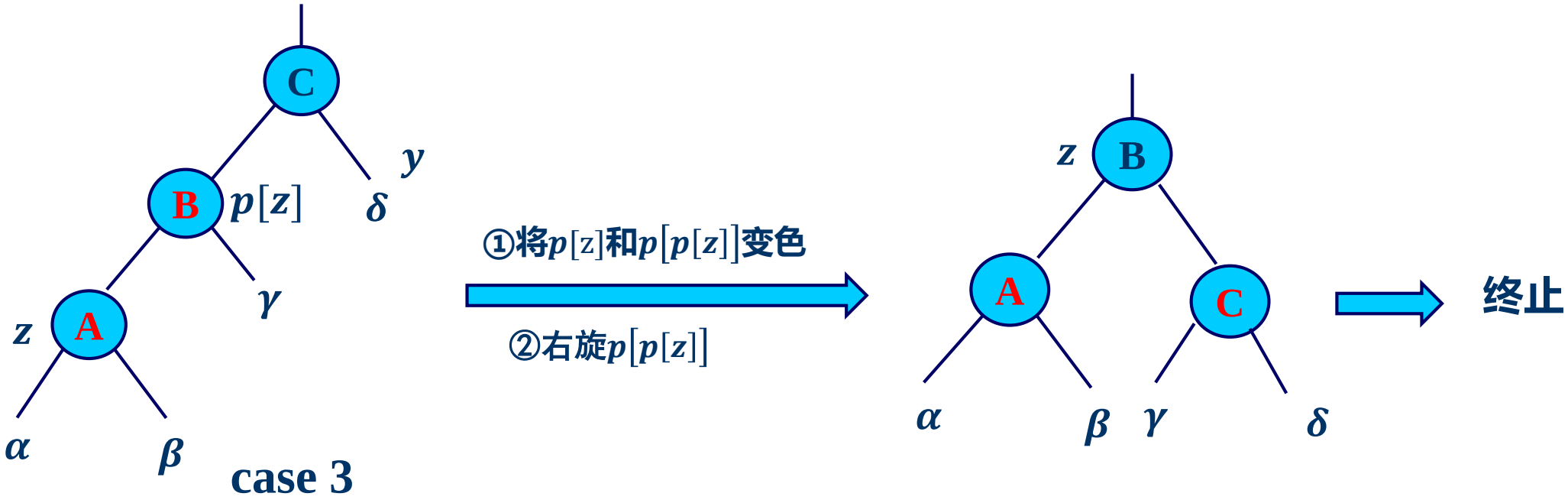
case 2: z的叔叔是黑色，其z是双亲$p[z]$的右孩子

插入40，不满足性质4，此时40的叔节点为NIL，是黑色，且40为父节点的右孩子，故为Case2。所以执行左旋操作将其转换为Case3



红黑树—元素插入操作

case 3: z的叔叔是黑色，其z是双亲$p[z]$的左孩子

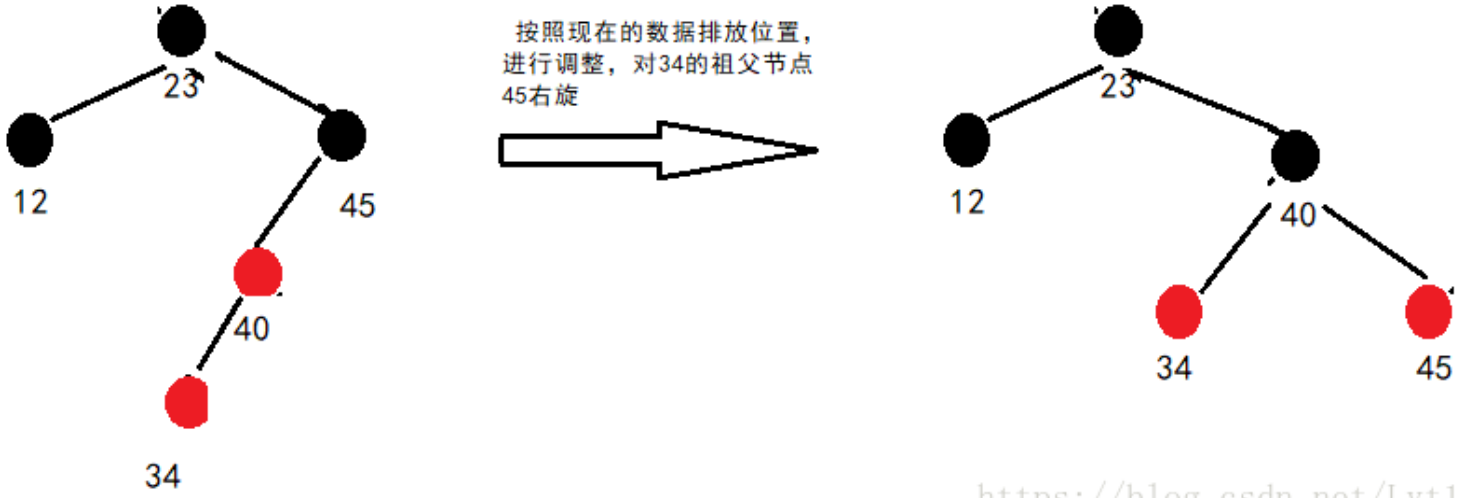


① Case3可以通过一次右旋，并且改变$p[z]$和$p[p[z]]$的颜色，从而终止调整操作。

红黑树—元素插入操作

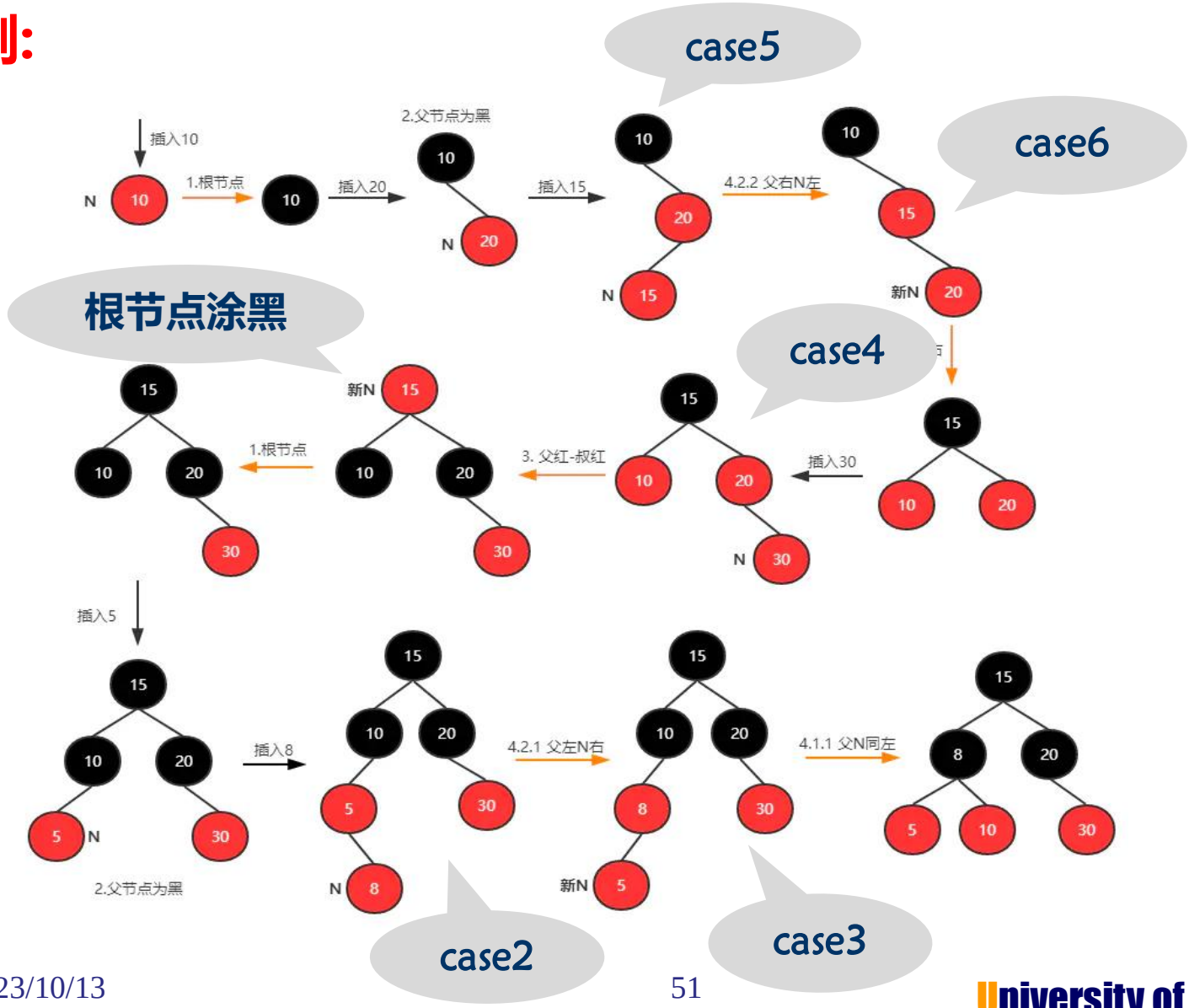
case 3: z的叔叔是黑色，其z是双亲$p[z]$的左孩子

下图左为case2例子中调整后的情况，属于case3，此时我们执行右旋操作，并且改变40与45的颜色。此时此时新z（40）颜色为黑，不再违反红黑树性质，调整终止。



红黑树—元素插入操作

示例:



红黑树—元素插入操作

伪代码:

```
RBInseertFixup( T, z ){
1    while (color[p[z]] = red ) do {
2        //若z为根或者p[z]为黑，无需要调整不进入本循环
3        if (p[z] = left[p[p[z]]]) then {              //case 1,2,3
4            y ← right[p[p[z]]];                       //y是z的叔叔
5            if (color[y] = red) then {                //case 1
6                color[y] = black; color[p[z]] = black;
7                color[p[p[z]]] = red; z ← p[p[z]];
8            } //case 1结束
9            else {                                    //case 2 or case 3  y为黑
10                   if(z = right[p[z]]) then {        //case 2
11                       z ← p[z];
12                       leftRotate(T, z)
13                   }                                 //以下为case 3
14                   color[p[z]] = black; color[p[p[z]]] = red;
15                   rightRotate(T, p[p[z]]);
16           }                                         //case 1结束
17       }                                             //case 2,3结束
18       else                                          //case 4~6 与上面对称
19       {… …}
20   } //endwhile
21   color[root[t]] ← black;
}
```

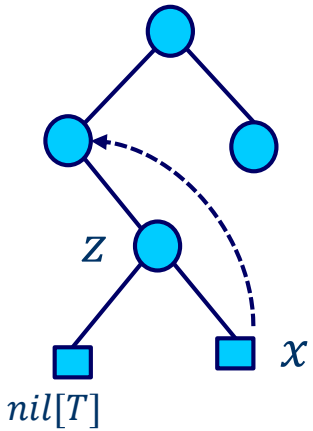
算法的时间复杂度:

- 调整算法时间：$O(\log n)$
- 整个插入算法的时间：$O(\log n)$
- 调整算法中至多使用两个旋转

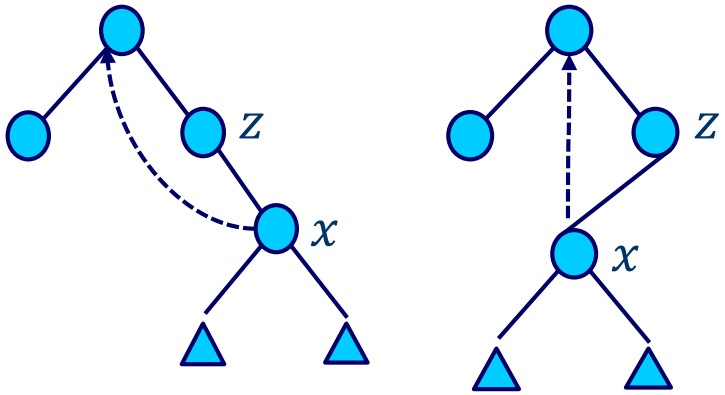
红黑树—元素删除操作

回顾BST树删除节点操作：

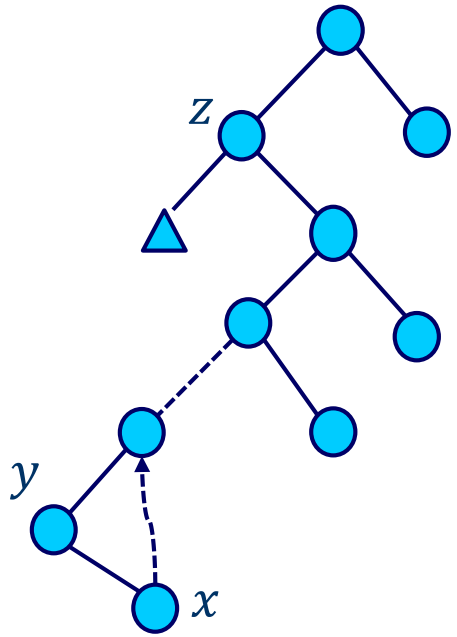
Case 1: z为叶子



Case 2: z只有一个孩子（非空）



Case 3: z的两个孩子非空



删除方法：

case 1：删除z，连接x。这里x是z的中序后继；

case 2：删除z，连接x。这里x是z的中序后继；

case 3：首先，找z的中序后继，即找z的右子树中最左下节点y；

然后，删除y，将y的内容copy到z，再将y的右子树连到$p[y]$左下。

红黑树—元素删除操作

红黑树删除节点代码：

```
RB-DELETE(T, z):
    if left[z] = nil[T] or right[z] = nil[T]     '''case1,2'''
        y ← z    '''后面进行物理删除y'''
    else         '''z的两子树均非空,case3'''
        y ← TREE-SUCCESSOR(z)   '''y是z的中序后继'''
    ''' 此时，y统一是x的双亲结点且是要删除节点
        x是待连接到p[y]的节点，以下要确定x'''
    if left[y] ≠ nil[T] '''本if语句综合了case1,2,3的x'''
        x ← left[y]
    else
        x ← right[y]
    '''以下处理：用x取代y与y的双亲连接'''
    p[x] ← p[y]
    if p[y] = nil[T]     '''y是根'''
        root[T] ← x      '''根指针指向x'''
    else     '''非根'''
        if y = left[p[y]]    '''y是双亲的左孩子'''
            left[p[y]] ← x
        else right[p[y]] ← x

    if y ≠ z                '''case3'''
        key[z] ← key[y]   '''y的内容拷贝到z'''
    if color[y] = BLACK
        RB-DELETE-FIXUP(T, x)   '''调整算法'''
    return y
```

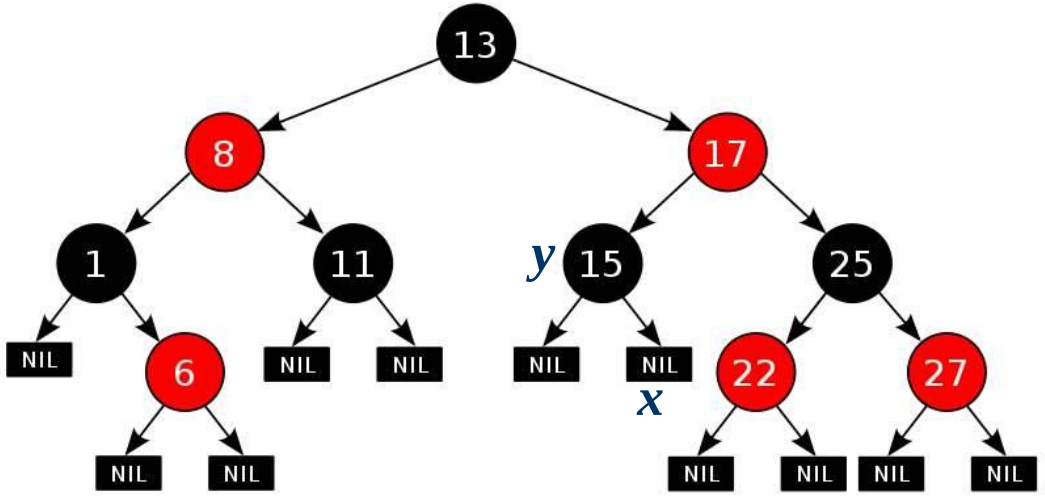
RB-DELETE-FIXUP用于实现对红黑树进行调整，以保证红黑树的性质

红黑树—元素删除操作

删除元素后的调整分析

① 删除操作总是在只有一边有孩子的节点或者没有孩子的节点(两个孩子指向nil[T])上进行的，不会在一个有两个孩子的节点上进行删除操作。因为如果要删除的元素左右孩子均非空，我们会找到它的后继节点，并用后继的值代替它，然后转而去删除后继所在节点。

如右图，如果我们要删除根节点13，那么我们会找到它的后继节点，将后继节点的值15赋值给根节点，转而去删除后继节点。



红黑树—元素删除操作

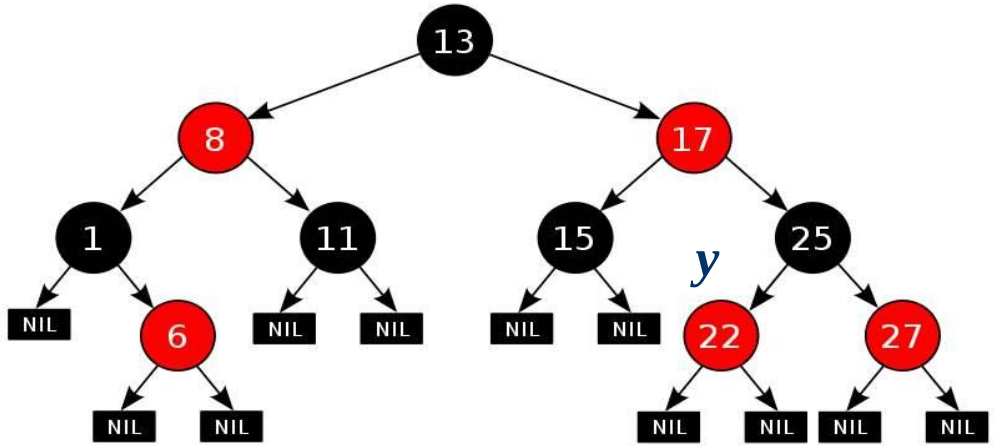
删除元素后的调整分析

② 如果删除一个红色的节点，红黑树的性质不会被破坏，因为：

A. 树中各节点的黑高度都没有变化

B. 不存在两个相邻的红色节点

C. 根仍是黑色的，因为如果y是红的，就不会是根



红黑树—元素删除操作

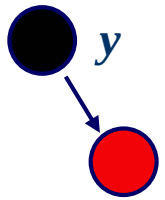
删除元素后的调整分析

③ 如果删除的是黑色节点会出现如下问题：

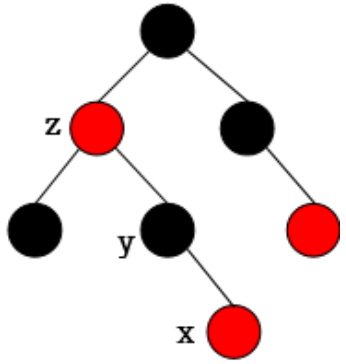
A. 若删除了根且一个红色节点做了新的根，性质1将被破坏

B. 若该节点唯一非空子节点和其父节点都是红色的，则违反了性质4

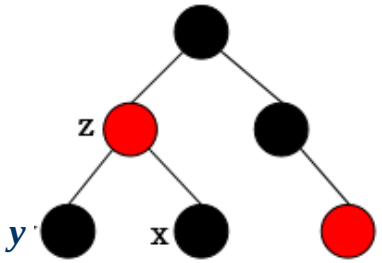
C. 删除该节点将导致先前包含y的任何路径上黑节点的个数减少1个，破坏性质5



(A)



(B)

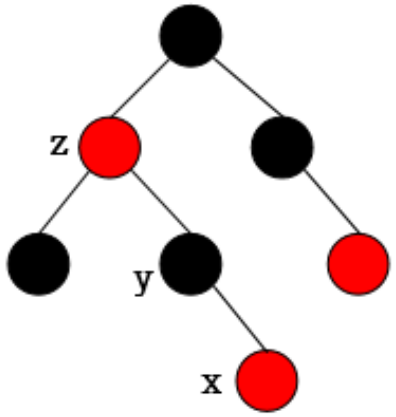


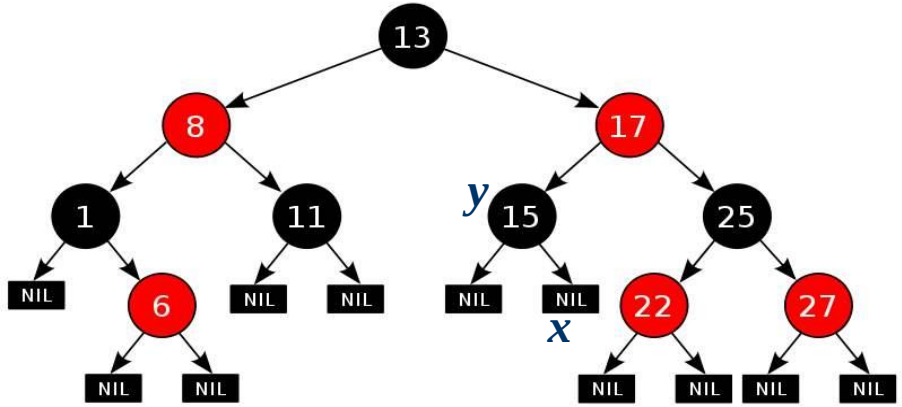
(C)

红黑树—元素删除操作

讨论：

已知y为待删除节点且为黑色，x是y的右孩子，则x或是y的唯一孩子或是$nil[T]$。可想象将y的黑色涂到x上，于是：



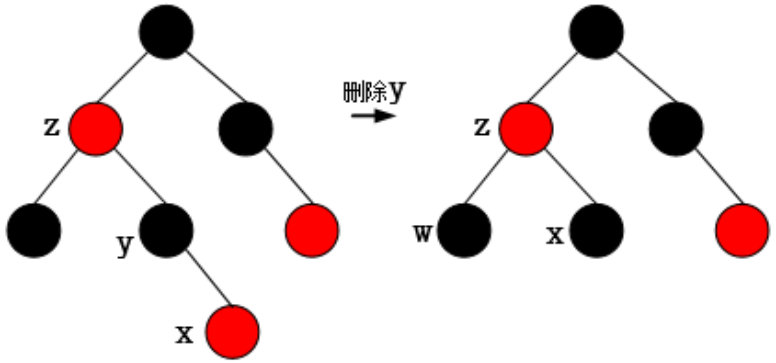


红黑树—元素删除操作

讨论：

已知y为待删除节点且为黑色，x是y的右孩子，则x或是y的唯一孩子或是$nil[T]$。可想象将y的黑色涂到x上，于是：

① 若x为红，只要将其涂黑，调整即可终止。

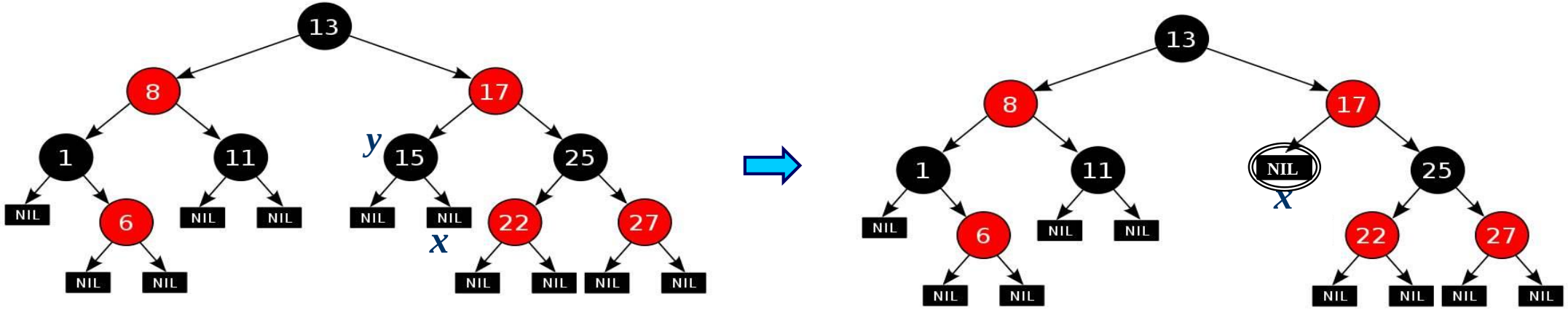


红黑树—元素删除操作

讨论：

已知y为待删除节点且为黑色，x是y的右孩子，则x或是y的唯一孩子或是$nil[T]$。可想象将y的黑色涂到x上，于是：

② 若x为黑，将y的黑色涂上之后，x是一个双黑色节点，违反性质1。



红黑树—元素删除操作

讨论：

已知y为待删除节点，x是y的右孩子，则x或是y的唯一孩子或是$nil[T]$。可想象将y的黑色涂到x上，于是：

② 若x为黑，将y的黑色涂上之后，x是一个双黑色节点，违反性质1。

处理步骤：

A. 若x为根，直接移除多余的一层黑色(树高减一)，终止；

B. 若x原为红，将y的黑色涂到x上，终止；

C. 若x非根节点，且为黑色，则x为双黑。通过**变色**、**旋转**使多余黑色**向上**传播，直到某个**红色节点**或传到**根**。

红黑树—元素删除操作

讨论：

已知y为待删除节点，x是y的右孩子，则x或是y的唯一孩子或是$nil[T]$。可想象将y的黑色涂到x上，于是：

② 若x为黑，将y的黑色涂上之后，x是一个双黑色节点，违反性质1：

A. 若x为根，直接移除多余的一层黑色(树高减一)，终止；

B. 若x原为红，将y的黑色涂到x上，终止；

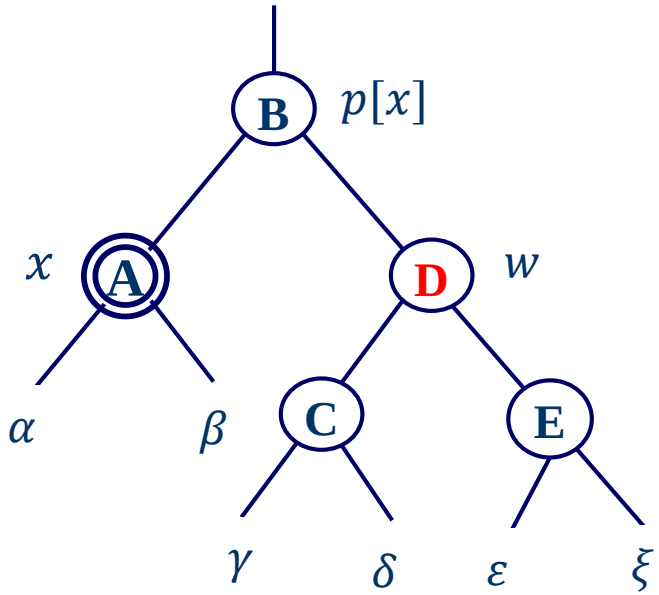
C. 若x非根节点，且为黑色，则x为双黑。通过**变色**、**旋转**使多余黑色**向上**传播，直到某个**红色节点**或传到**根**。

可被分为8种情况

- ✓ Case1~Case4为x是$p[x]$的左子树
- ✓ Case5~Case8为x是$p[x]$的右子树

红黑树—元素删除操作

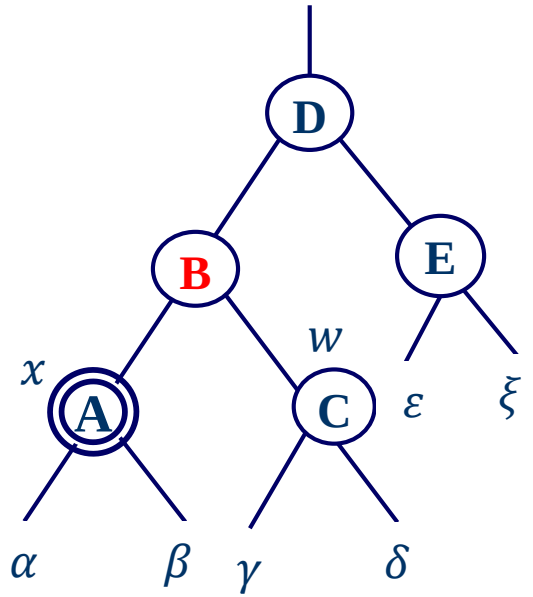
Case 1：x的兄弟节点是红色的



∵ w是红，　∴$p[x]$必黑

① w变黑，$p[x]$变红

② $p[x]$左旋，w指向x的新兄弟



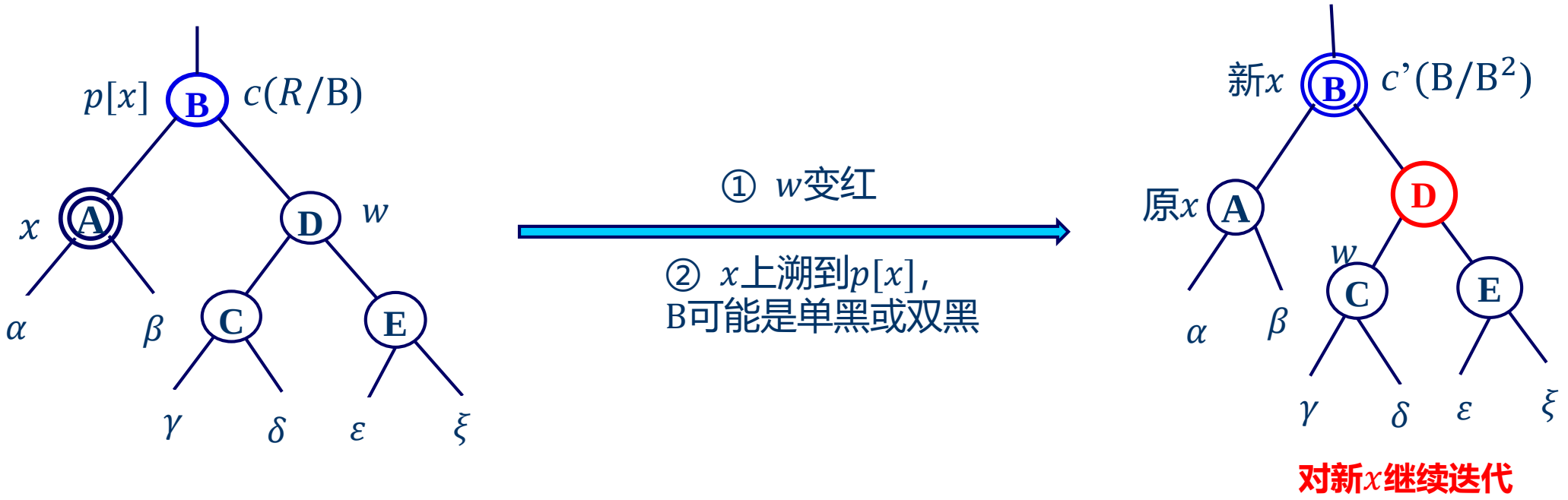
Case 2, 3, 4

理由：若w为红，则w的孩子必为黑，通过上述旋转操作可以使得x的新兄弟为黑，从而将Case 1转变为Case2，3，4中的一种去解决。

注意：为了旋转后不违反红黑树的性质，$p[x]$需变成红色，w需变成黑色

红黑树—元素删除操作

Case 2：x的兄弟节点w是黑色，且w的两个孩子均为黑色

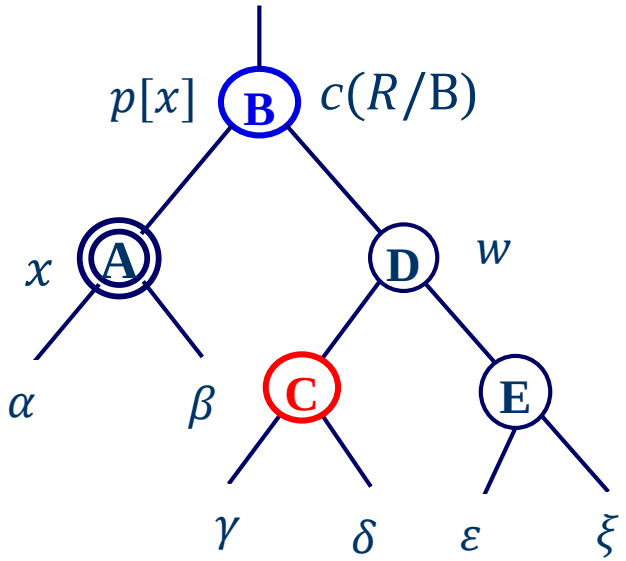


对新x继续迭代

w的孩子都是黑色情况下，将w变红不违反性质4，但此时w这一侧分支少一层黑色。为了不违反性质5，将x的一层黑色上移至$p[x]$，此时$p[x]$成为了新x（可能为单黑或双黑），我们可以对新x继续进行调整操作。

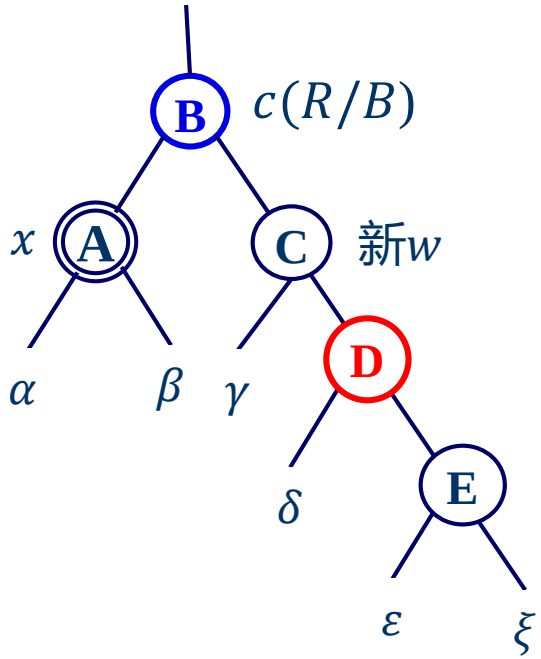
红黑树—元素删除操作

Case 3：x的兄弟节点w是黑色，且w的右孩子为黑色，左孩子为红色



① w左孩子变黑，w变红

② w右旋，w指向x的新兄弟

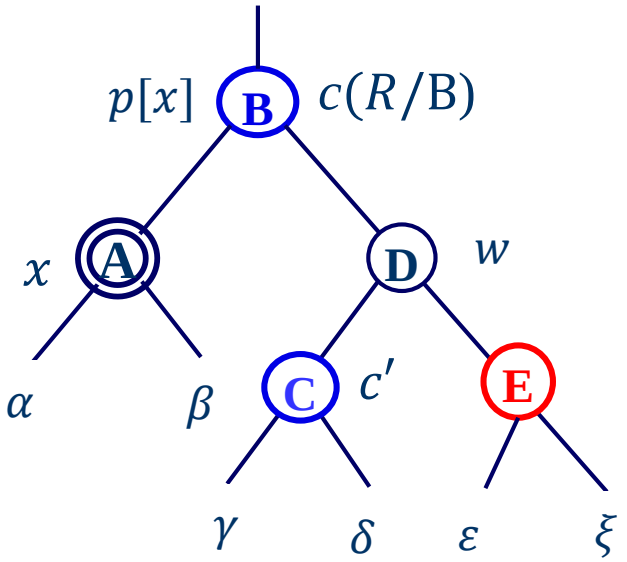


Case 4

Case3 的调整目标是将***Case3*** 转变成***Case4***去解决。

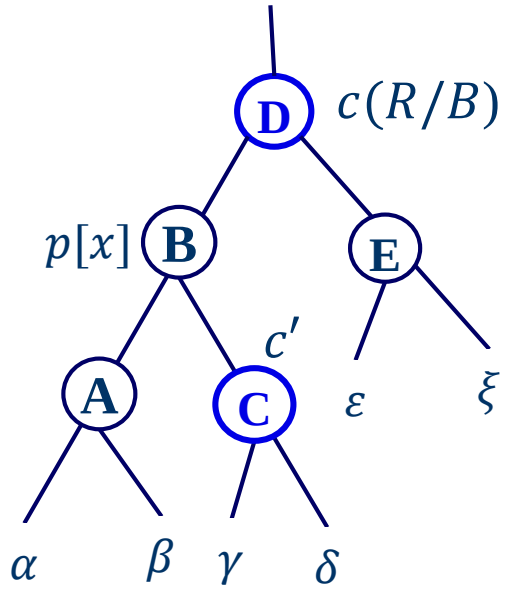
红黑树—元素删除操作

Case 4：x的兄弟节点w是黑色的，且w的右孩子为红色（左孩子为黑或红）



① $p[x]$的颜色c涂到w上

② $p[x]$涂黑，w右孩子变黑，$p[x]$左旋



终止

Case4可终结调整操作，可以理解为：

① B做一个左旋操作并且与D互换颜色，这个操作存在的问题有：B的左侧分支多了一层黑色，D的右侧分支少了一层黑色。这两个问题将在②中得到解决；

② 我们可以直接丢掉A（双黑）的一层黑色。对于E而言，可以直接将其变成黑色，调整结束！

红黑树—元素删除操作

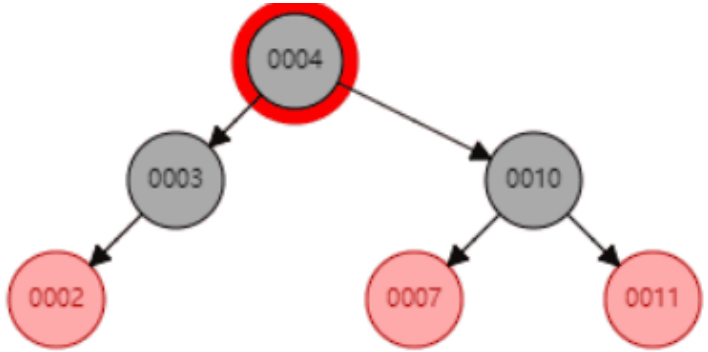
伪代码：

```
1 while x ≠ root[T] and color[x] = BLACK
2     do if x = left[p[x]]
3           then w ← right[p[x]]
4                if color[w] = RED
5                   then color[w] ← BLACK                          ▸ Case 1
6                        color[p[x]] ← RED                         ▸ Case 1
7                        LEFT-ROTATE(T, p[x])                      ▸ Case 1
8                        w ← right[p[x]]                           ▸ Case 1
9                if color[left[w]] = BLACK and color[right[w]] = BLACK
10                  then color[w] ← RED                            ▸ Case 2
11                       x ← p[x]                                  ▸ Case 2
12                  else if color[right[w]] = BLACK
13                          then color[left[w]] ← BLACK            ▸ Case 3
14                               color[w] ← RED                    ▸ Case 3
15                               RIGHT-ROTATE(T, w)                ▸ Case 3
16                               w ← right[p[x]]                   ▸ Case 3
17                       color[w] ← color[p[x]]                    ▸ Case 4
18                       color[p[x]] ← BLACK                       ▸ Case 4
19                       color[right[w]] ← BLACK                   ▸ Case 4
20                       LEFT-ROTATE(T, p[x])                      ▸ Case 4
21                       x ← root[T]                               ▸ Case 4
22       else (same as then clause with "right" and "left" exchanged)
23 color[x] ← BLACK
```

红黑树—元素删除操作

可以利用下面这个网站可视化红黑树操作过程，协助理解算法。

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

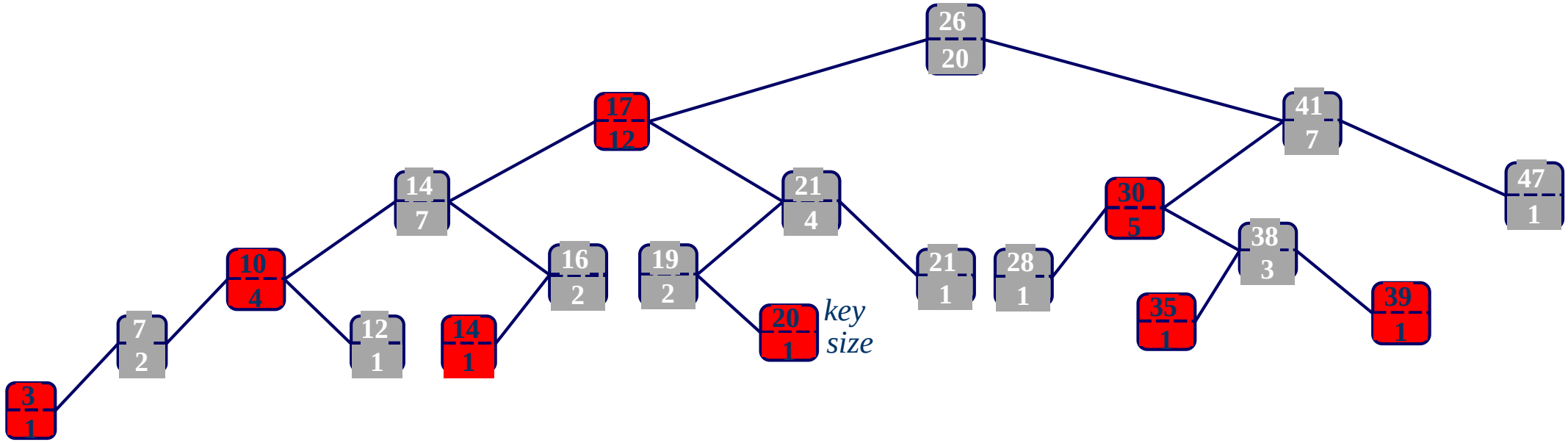


红黑树扩张—顺序统计树

顺序统计树定义：

$OS(Order-Statistic)$树是一棵扩充的红黑树，在每个节点上扩充一个域$size[x]$而得到的。$size[x]$域表示以x为根的子树中内部节点的总数（包括x），即子树大小。

$$size[x]=\begin{cases}0 & if\ \ x=nil[T]\\ size[left[x]]+size[right[x]]+1 & other\end{cases}$$



红黑树扩张—顺序统计树

选择问题： 在以x为根的子树中 查找第i个最小元素；

主要步骤：

① 计算以x为根的子树中节点x的排序r（=左子树节点个数+当前节点）；

② 如果$i = r$，则x就是待查找的最小元素；

③ 如果$i < r$，则第i小元素就在x的左子树中，到左子树中递归查找第i小元素；

④ 如果$r > r$，则第i小元素就在x的右子树中，到右子树中递归查找第$i - r$小元素。

```
OS-Select(x, i)
{
1    r ← sizeof[left[x]] + 1;
2    if i=r
3       then return r;
4    elseif i<r
5       then return OS-Select( left[x], i )
6    else return OS-Select( right[x], i - r)
}
```

红黑树扩张—顺序统计树

求秩问题： 在OS树中，给定元素x求其在中序遍历得到的线性序列中x的位置；

主要步骤：

① 计算在以x为根的子树中，x的秩r；

② 如果x是根，则返回r；

③ 如果x是双亲的左孩子，则x在以$p[x]$为根的子树中的秩是r；

④ 如果x是双亲的右孩子，则x在以$p[x]$为根的子树中的秩是$r + sizeof\left[left[p[x]]\right] + 1$；

⑤ x上移至$p[x]$；

⑥ 重复③④⑤直至情况②成立时终止。

```
OS-Rank(T, x)
{
1    r ← sizeof[left[x]] + 1;
2    y ← x ;
3    while y ≠ root[T]
4         do if  y = right[p[y]]
5                  then r ← r + size[left[p[y]]] + 1
6         y ← p[y];
7    return r;
}
```

时间复杂度： $O(\lg n)$

红黑树扩张—顺序统计树

元素插入：

Phase 1：从根向下插入新节点，将搜索路径上所经历的每个节点的**size + 1**，新节点的**size**置为1；

- 附加成本：$O(\lg n)$

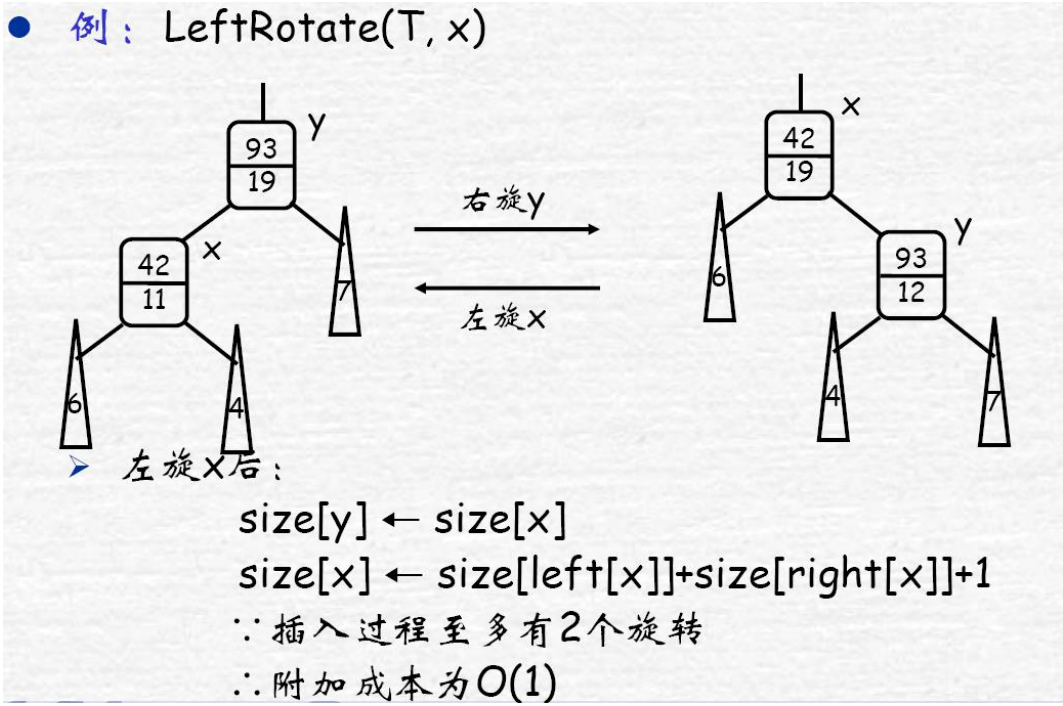
Phase 2：采用变色和旋转方法，从叶子向上调整；

- 变色不改变$size$；
- 旋转可能改变$size$：

  ∵ 旋转是局部操作，

  又，只有轴上两个节点的$size$可能违反定义。

  ∴ 只需在旋转操作后对违反节点的$size$进行修改。
- 附加成本：旋转为$O(1)$，总成本为$O(\lg n)$；



红黑树扩张—顺序统计树

元素删除：

Phase 1：物理上删除y，在删除y时从y上溯至根，将所经历的节点的$size$均减1；

- 附加成本：$O(\lg n)$

Phase 2：采用变色和旋转方法，从叶子向上调整；

- 变色不改变$size$；
- 旋转可能改变$size$，至多有3个旋转；
- 附加成本：$O(\lg n)$；

重点说明：

上面介绍的插入和删除均是有效维护，有效维护保证扩充前后的基本操作的渐近时间不变。

红黑树扩张—基本步骤

扩张目的： **设计新的操作、加速已有的操作**

扩张步骤：

① 选择基础数据结构

② 确定要在基础数据结构中添加附加信息

③ 可以有效地维护附加信息

④ 设计新的操作。

顺序统计树：

① 选择红黑树作为基本数据结构；

② 在红黑树上增加size域；

③ 有效性证明；

④ 设计OS-Select、OS-Rank操作。

关键点： **如何保证可以有效维护附加信息？**

红黑树扩张—基本步骤

扩张目的： **设计新的操作、加速已有的操作**

扩张步骤：

① 选择基础数据结构

② 确定要在基础数据结构中添加附加信息

③ 可以有效地维护附加信息

④ 设计新的操作。

顺序统计树：

① 选择红黑树作为基本数据结构；

② 在红黑树上增加size域；

③ 有效性证明；

④ 设计OS-Select、OS-Rank操作。

定理14.1： 假设f是红黑树T的n个节点上扩充域。对$\forall x \in T$，假设x的f域的内容能够仅通过节点x、$left[x]$、$right[x]$的信息（包括f域）的计算就可以得到，则：扩充树上的插入和删除维护操作（包括对f域的维护）不改变原有的渐近时间$O(\lg n)$。

红黑树扩张—区间树

问题描述

一个事件占用一段连续时间，因此每个事件可以用一个区间来表示。我们经常要查询一个由时间区间数据构成的数据库，以找出特定的区间内发生了什么事件。如何设计有效的数据结构来维护这样一个区间数据库?

应用场景： 课程查询、活动查询。

区间树是解决上述问题的一个非常有效的数据结构，通过扩张红黑树以支持由区间构成的动态集合上的操作。

红黑树扩张—区间树

基本概念：

① 区间：表示占用一段连续时间的事件；

② 闭区间：实数的有序对$[t_1, t_2]$，$t_1 \leq t2$；

③ 区间的对象表示：区间$[t_1, t_2]$可以用对象i表示，有两个属性：$low[i] = t_1$，$high[i] = t_2$；

④ 区间重叠：$i \cap i' \neq \emptyset \Leftrightarrow (low[i] \leq high[i'])\ and\ (low[i'] \leq high[i])$

区间树支持下列操作：

① INTERVAL-INSERT(T, x)：将包含区间域int的元素x插入到区间树T中；

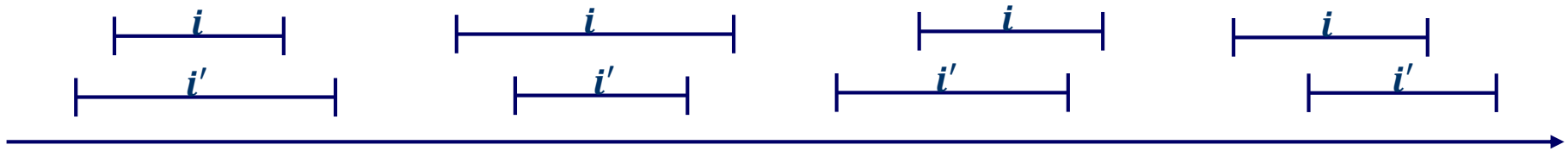
② INTERVAL-DELETE(T, x)：从区间树T中删除元素x；

③ INTERVAL-SEARCH(T, i)：返回一个指向区间树T中元素x的指针，使$int[x]$与i重叠；若集合中无此元素存在，则返回$nil[T]$。

红黑树扩张—区间树

区间三分法：

(a) i和i'重叠：



(b) i在i'前：$high[i] < low[i']$

(c) i'在i前：$high[i'] < low[i]$

红黑树扩张—区间树

扩充步骤：

① 基本结构

以红黑树为基础，对$\forall x \in T$, x包含区间$int[x]$的信息(低点和高点)，$key = low[int[x]]$。

② 附加信息

$$\max[x] = \max(high[int[x]], \max[left[x]], \max[right[x]])$$

③ 有效维护附加信息

由定理14.1及max的定义可以证明附加信息可以有效维护。

④ 开发新操作

查找与给定区间重叠的区间

节点x

红黑树扩张—区间树

查找算法INTERVAL-SEARCH(T, i)

Step 1: $x \leftarrow root[T]$; //从根开始查找

Step 2: 若$x \neq nil[T]$且i与$int[x]$不重叠

if x的左子树非空且左子树中最大高点≥ $low[i]$ then

$x \leftarrow left[x]$; //到x的左子树中继续查找

else //左子树必查不到，到右子树查

$x \leftarrow right[x]$;

Step 3: 返回x; //$x = nil$ 或者 i和x重叠

关键点：

如何理解所寻找路径的安全性？如果存在着重叠区间，怎么保证一定可以找到？

红黑树扩张—区间树

定理14.2 **INTERVAL-SEARCH(T, i)**的任意一次执行，或者返回一个其区间与i重叠的结点，或者返回$nil[T]$，此时树T中没有任何结点的区间与i重叠。

证明思路：

Case 1：若算法从x搜索到左子树，则左子树中包含一个与i重叠的区间或在x的右子树中没有与i重叠的区间；

Case 2：若从x搜索到右子树时，则在左子树中不会有与i重叠的区间。

只要证明Case 1和Case 2成立即可。

红黑树扩张—区间树

定理14.2 **INTERVAL-SEARCH(T, i)**的任意一次执行，或者返回一个其区间与i重叠的结点，或者返回$nil[T]$，此时树T中没有任何结点的区间与i重叠。

证明：

Case 2：若从x搜索到右子树时，则在左子树中不会有与i重叠的区间。

① 下面先证明Case 2成立。

根据算法可知，走右分支条件是$left[x] = nil \ \ or \max[left[x]] < low[i]$。

- 若左子树为空，则左子树不含有与i重叠的区间；
- 若左子树非空，由于有$\max[left[x]] < low[i]$

  ∵ $\max[left[x]]$是左子树中最大高点

  ∴ 左子树中的区间不可能与i重叠

红黑树扩张—区间树

定理14.2 **INTERVAL-SEARCH(T, i)**的任意一次执行，或者返回一个其区间与i重叠的结点，或者返回$nil[T]$，此时树T中没有任何结点的区间与i重叠。

证明：

Case 1：若算法从x搜索到左子树，则左子树中包含一个与i重叠的区间或在x的右子树中没有与i重叠的区间；

② 下面再证明Case 1成立。

- 若左子树包含与i重叠的区间，则走左分支正确；
- 若左子树无区间与i重叠，下证x的右子树中也无区间与i重叠。

$\because \max[left[x]] \geq low[i]$　　$\therefore$ 存在区间$i' \in \{x$左子树的区间$\}$，使得 $high[i'] = \max[left[x]] \geq low[i]$

$\Rightarrow$ 只有i'和i重叠或i在i'前

$\because$ 左子树无区间与i重叠　　$\therefore$ 只有i在i'前，即$high[i] < low[i']$ --------- (1)

$\because$ 对任何区间$i" \in \{x$右子树的区间$\}$， 由BST性质有$low[i'] \leq low[i"]$，　$\therefore high[i] < low[i"]$ ----- (2)

$\Rightarrow$ 结合(1)和（2）可以判断i和$i"$不重叠。

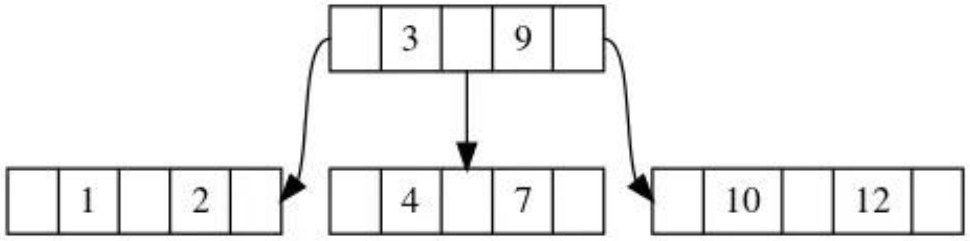
第七讲
平衡树专题

- 二叉搜索树
- AVL树
- 红黑树及扩张
- B树/B+树/B*树

University of Science and Technology of China

B树-概念

- 在计算机科学中，B树是一种自平衡的树，能够保持数据有序。这种数据结构能够让查找数据、顺序访问、插入数据及删除的动作，都在对数时间内完成。
- B树概括来说是一个一般化的二叉查找树，一个节点可以拥有2个以上的子节点。



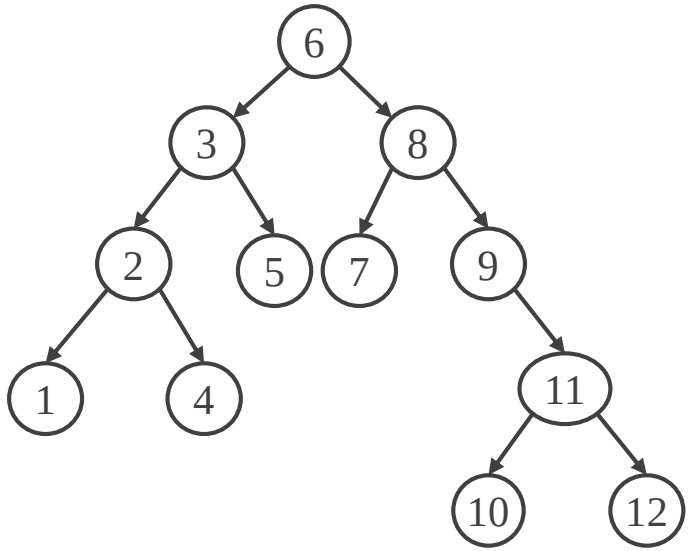
- 在B树当中有一个非常巧妙的设计，就是每一个节点的孩子个数是元素的数量+1，和二叉搜索树一样，B树种结点元素存在大小顺序的关联。因此，可以使用二分查找进行高效搜索。

如图，根节点有两个元素3和9，并且有3个孩子节点，刚好对应了3个区间。分别是小于3的，在3和9中间的以及大于9的，那么根据我们要查找的元素的大小，我们很容易判断究竟应该选择哪一个分支。

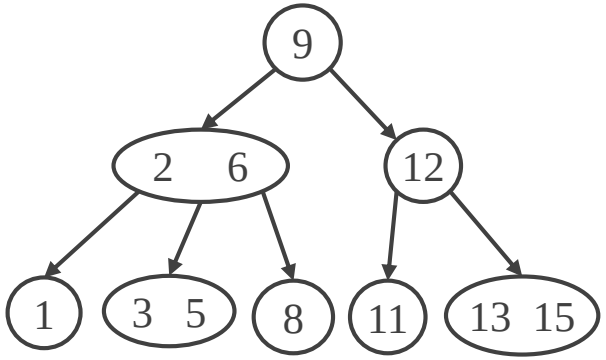
B树-概念

B树是为了弥补不同的存储级别之间的访问速度上的巨大差异，实现高效 I/O。

平衡二叉树的查找效率是非常高的，并可以通过降低树的深度来提高查找的效率。但是当数据量非常大，树的存储的元素数量是有限的，这样会导致二叉查找树结构由于树的深度过大而造成磁盘I/O读写过于频繁，进而导致查询效率低下。另外数据量过大会导致内存空间不够容纳平衡二叉树所有结点的情况。B树是解决这个问题的很好的结构。



相同情况下，B树相比二叉树有更小的树高



B树-概念

应用场景：

① **B树主要用在各大存储文件系统和数据库系统当中**。在这些场景下数据总量很大，我们不可能将它们都存储在内存当中。所以为了解决这个问题，我们会在树节点当中存储孩子节点的在磁盘上的地址。在需要访问的时候通过磁盘加载将孩子节点的信息读取到内存当中。也就是说在数据库当中我们遍历树的时候也伴随着磁盘读取。

② 磁盘的随机读写是非常耗时的。显然，树的深度越大，磁盘读写的次数也就越多，那么带来的IO开销也就越大。所以为了优化这个问题，才设计出了B树。由于B树每个节点存储的数据和孩子节点数都大于2，所以和二叉搜索树相比，它的树深要明显小得多。因此读写磁盘的次数也更少，带来的IO开销也就越小。这也是它**适合用在文件引擎以及数据库引擎上**的原因。

B树-概念

(M阶) B树的定义：：

① 每一个节点最多有M个子节点

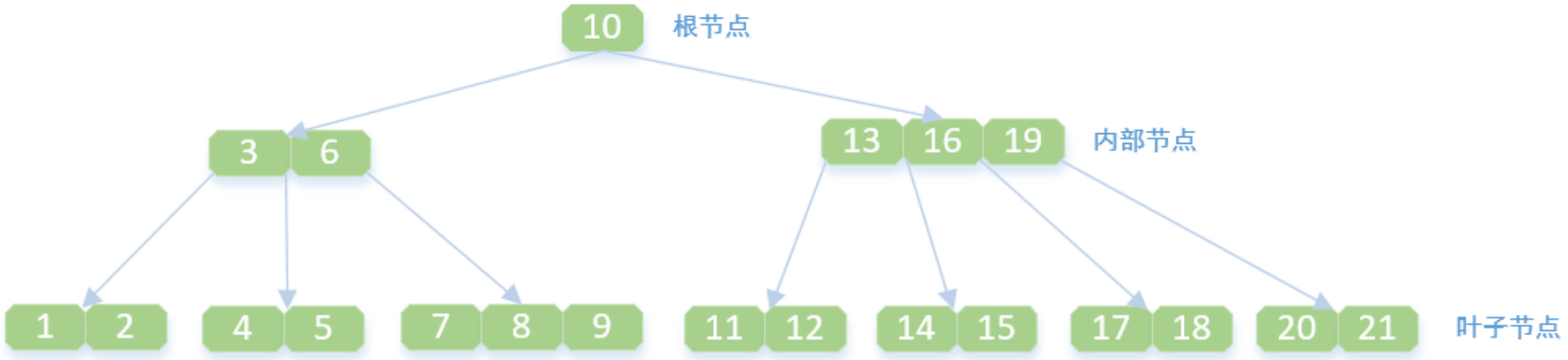
② 每一个内节点(除根节点)最少有$\lceil M/2 \rceil$个子节点

③ 根节点若非叶子节点，那么它至少拥有两个子节点（也就是至少拥有1个关键字）

④ 有 k 个子节点的非叶子节点拥有 $k - 1$ 个键

⑤ 叶子节点最少拥有$\lceil M/2 \rceil - 1$个键

⑥ 所有叶子节点属于同一层



一颗四阶B树

B树-概念

B树的阶：

- ✓ **B树中一个节点的子节点数目的最大值**
- ✓ 用M表示，假如最大值为4，则为4阶，如图，所有节点中，节点[13,16,19]拥有的子节点数目最多，四个子节点（灰色节点），所以可以定义上面的图片为4阶B树。

根节点

- ✓ **特征：**根节点拥有的子节点数量的上限和内部节点相同，如果根节点不是树中唯一节点的话，至少有俩个子节点（不然就变成单支了）。
- ✓ 在M阶B树中（根节点非树中唯一节点），那么子节点的数目满足关系式$2 \leq m \leq M$，包含的元素数量满足$1 \leq k \leq m-1$。

B树-概念

内部节点:

- ✓ **特征：**内部节点是除叶子节点和根节点之外的所有节点，拥有父节点和子节点；
- ✓ M阶B树中内部节点元素数量k必须满足$\lceil \frac{M}{2} \rceil - 1 \leq k \leq M - 1$，其子节点数量为$k + 1$。

叶子节点

- ✓ 最后一层都为叶子节点，叶子节点对元素的数量有相同的限制，但是没有子节点，也没有指向子节点的指针。
- ✓ **特征：**在M阶B树中叶子节点的元素数量k符合$\frac{M}{2} - 1 \leq k \leq M - 1$。

B树-概念

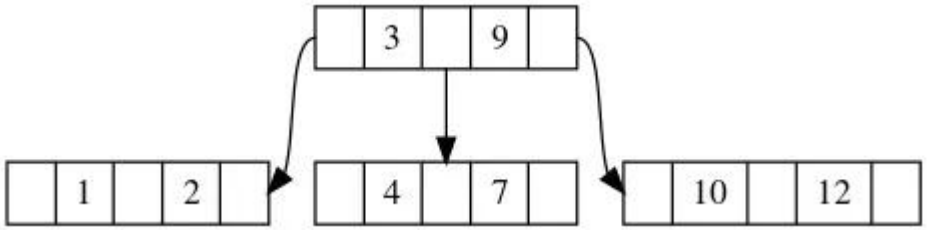
B树的特点：

- 根节点拥有的子节点数量的上限和内部节点相同，但是没有下限。例如，当整个树中的元素数量小于$[M/2]-1$时，根节点是唯一的节点并且没有任何子节点。
- 叶子节点对元素的数量有相同的限制，但是没有子节点，也没有指向子节点的指针。
- 内部节点是除叶子节点和根节点之外的所有节点。它们通常被表示为一组有序的元素和指向子节点的指针。元素的数量总是比子节点指针的数量少一。

B树操作 – 元素搜索

搜索步骤：

- 根据要查找的关键码key，在根节点的关键码集合中进行顺序或二分法检索，若$key = k_i$，则检索成功；否则，key一定在某 k_i 和 k_{i+1} 之间，用一个指针在所指节点继续查找，重复上述检索过程，直到检索成功；或指针为空，则检索失败。



此处可利用二分查找

欲搜索7，首先于根节点中找到第一个大于等于7的位置，这个位置的元素是9不等于7，说明当前节点没有7，我们需要继续往子树递归查找。由于子树对应元素分割出来的区间，所以我们可以确定如果7存在子树当中，只会出现在9前面的子树中，所以我们往9前面的子树，也就是node.children[1]的子树方向递归。

B树操作 – 元素插入

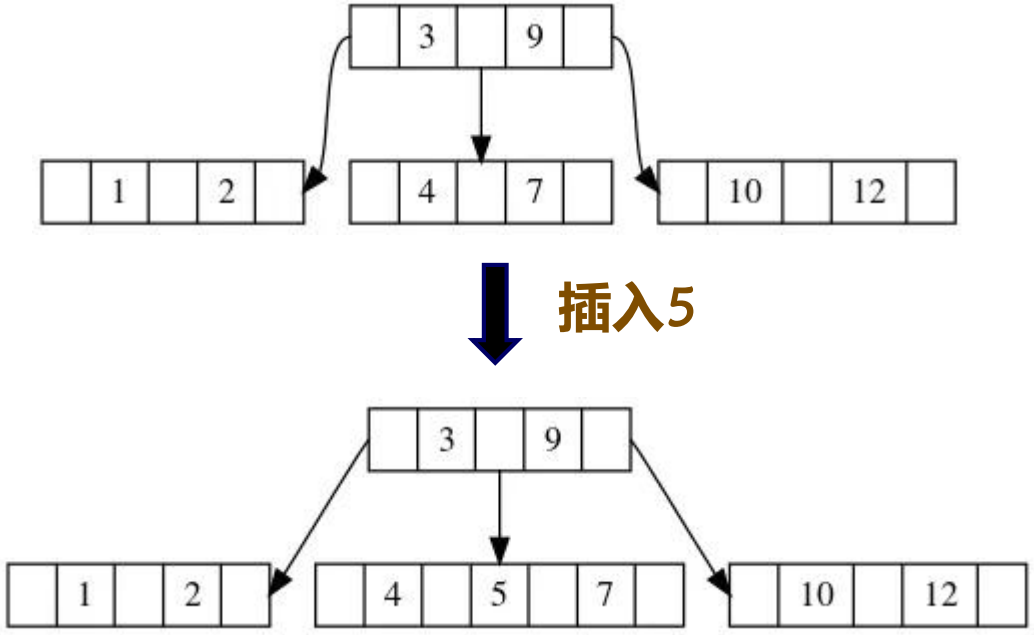
B树元素所有的插入操作只发生在叶子节点。

步骤：

- 如果叶子节点拥有的元素数量小于最大值（$M-1$），那么有空间容纳新的元素。将新元素插入到这一节点，且保持节点中元素有序。
- 否则的话这一节点已经满了，将它平均地分裂成两个节点：
  ① 从叶子节点的元素和新的元素中选择出中位数；
  ② 小于这一中位数的元素放入左边节点，大于这一中位数的元素放入右边节点，中位数作为分隔值；
  ③ 分隔值被插入到父节点中，这可能会造成父节点分裂，分裂父节点时可能又会使它的父节点分裂，以此类推。如果没有父节点（这一节点是根节点），就创建一个新的根节点（增加了树的高度）。

B树操作 – 元素插入

例子： 向如下阶为4的B树中依次插入元素{5,6}

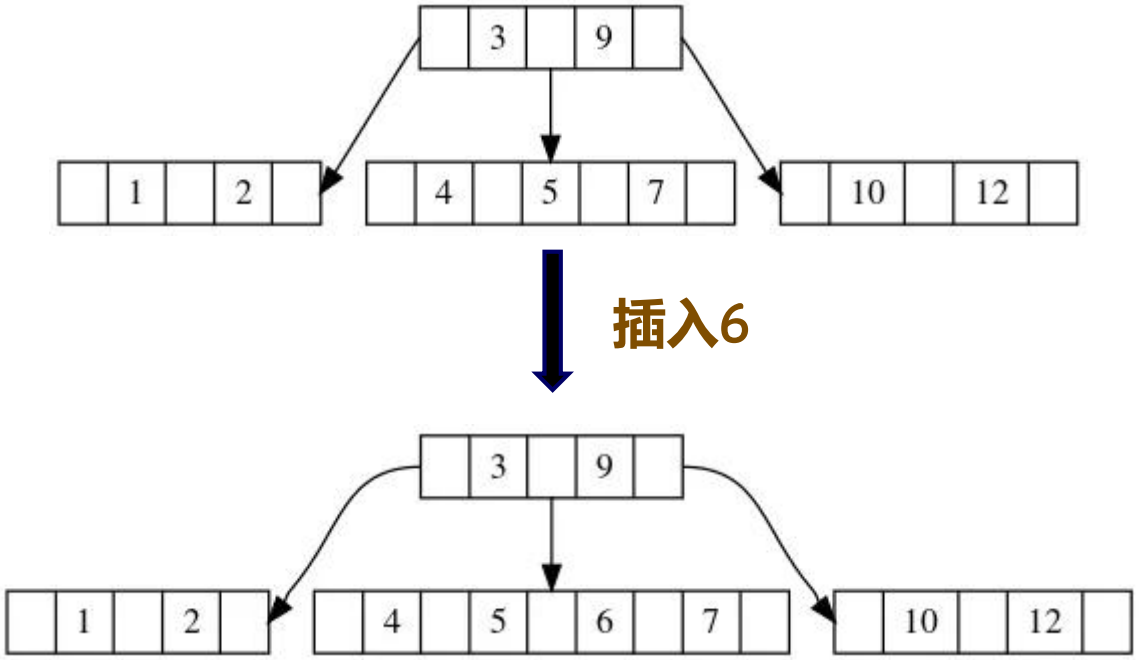


插入元素5后，这时候中间叶子节点的元素数量达到3，

这时尚未违反B树的性质。

B树操作 – 元素插入

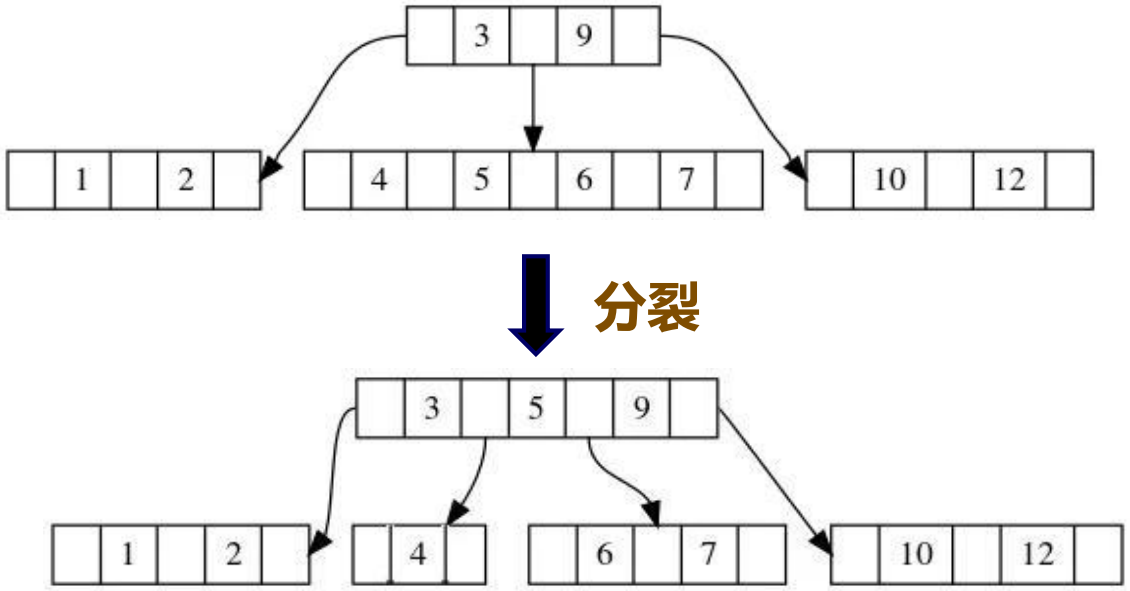
例子： 向如下阶为4的B树中依次插入元素{5,6}



这时叶子节点在连续插入两个元素之后数量大于等于M，那么我们需要将它一分为二，将中间节点上传给父节点。

B树操作 – 元素插入

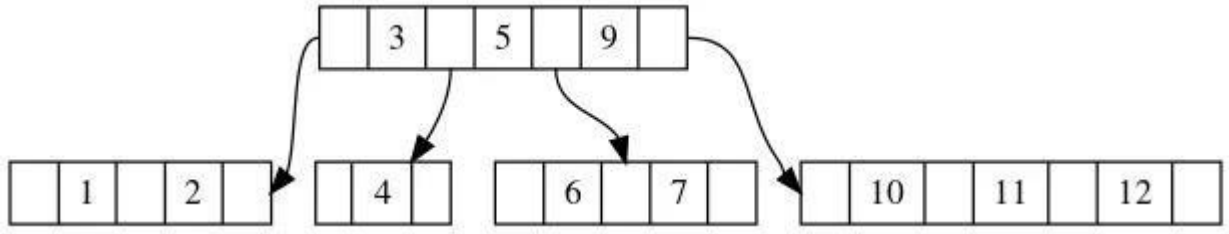
例子： 向如下阶为4的B树中依次插入元素{5,6}



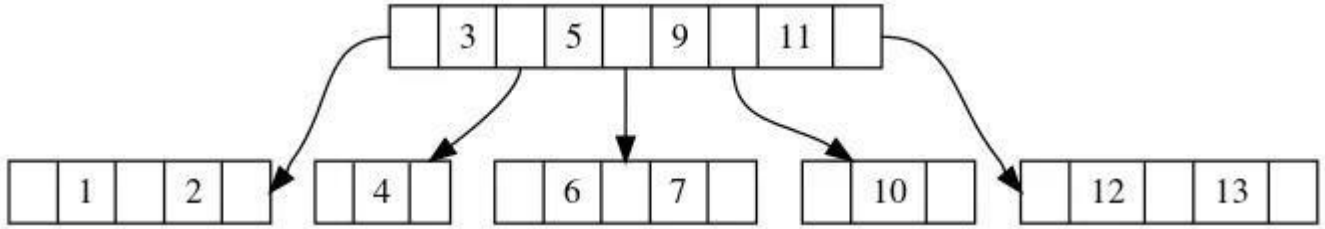
于是经过这个调整之后，父节点当中增加了一个元素5，也增加了一个分支，保证了B树继续合法。

B树操作 – 元素插入

例子： 向如下阶为4的B树中依次插入元素{13}

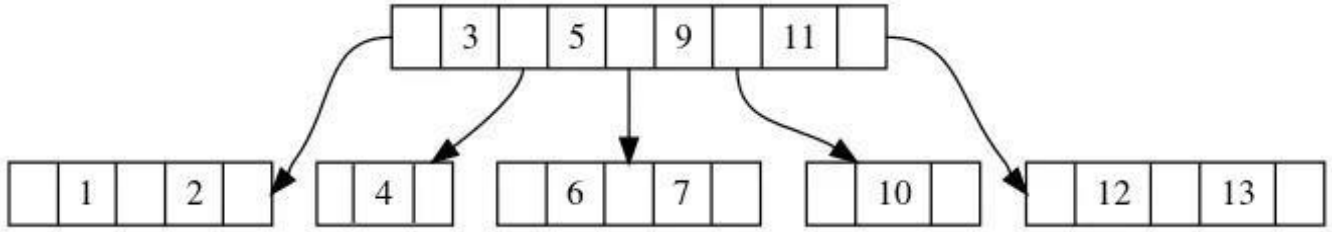


我们往其中插入13，会导致最后一个叶子节点数量超过限制，于是我们分裂节点，将中间元素11上传到父节点：

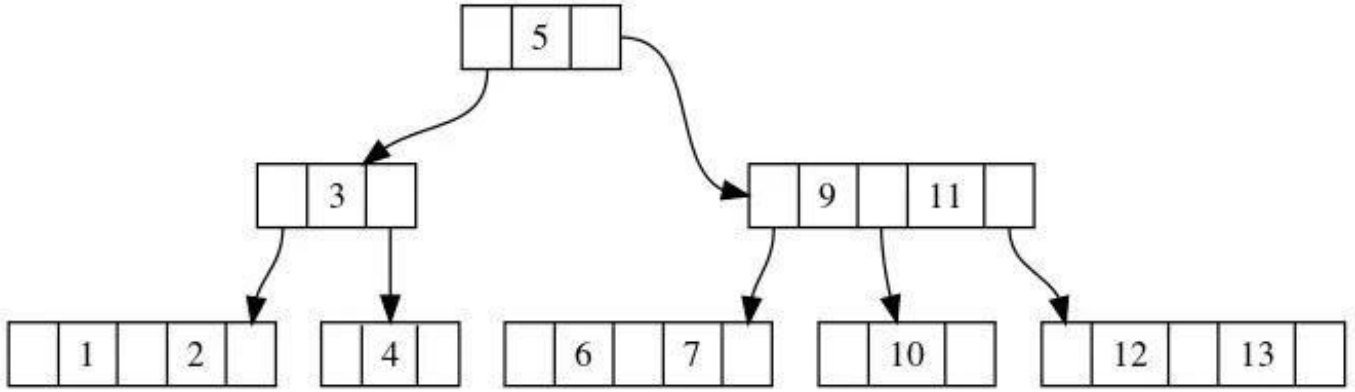


B树操作 – 元素插入

例子： 向如下阶为4的B树中依次插入元素{13}

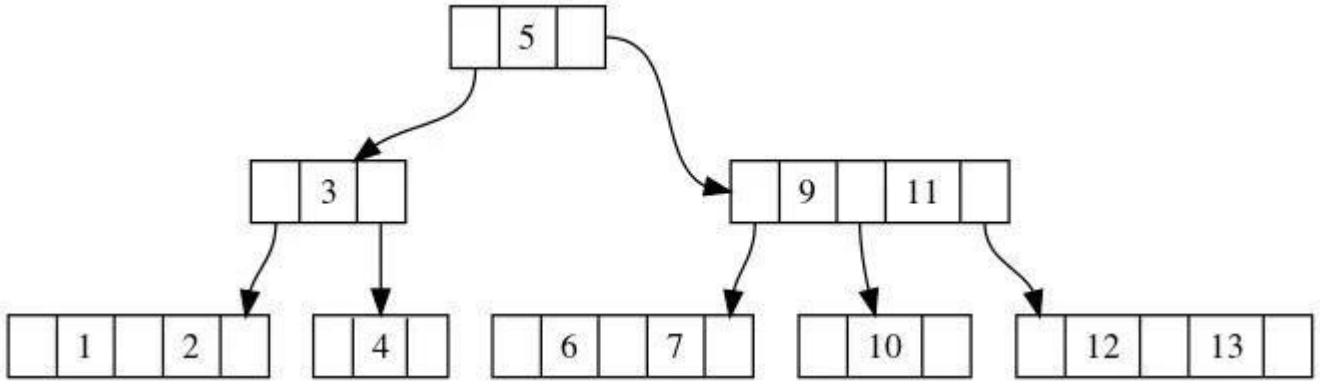


但是由于上传父节点也可能引起元素数量超过限制，所以我们要向上递归判断是否需要分裂节点的操作。此时父节点当中元素数量大于等于M，需要继续分裂：



B树操作 – 元素插入

例子： 向如下阶为4的B树中依次插入元素{13}

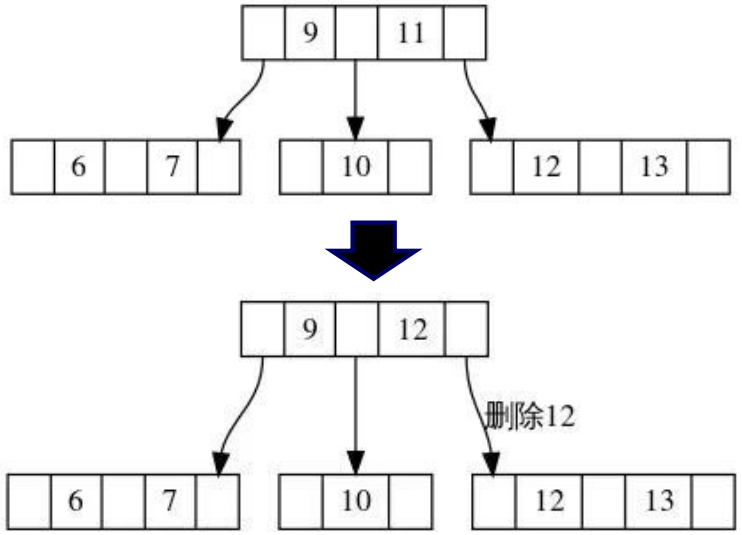


分裂产生新生成的节点由于高度更高，代替了原本的根节点，成为了新的根节点。并且原来根节点的子树也发生了拆分，分别分配给新根节点的两个子树。也就是说我们在拆分节点的时候，除了要拆分keys之外，也需要拆分children。

B树操作 – 元素删除

所有的删除都可以转化成删除叶子节点的问题。

- 如果要删除的key位于非叶子结点上，则用后继key覆盖要删除的key，然后在删除该后继key。非叶子节点中元素的后继key一定位于叶子结点上。

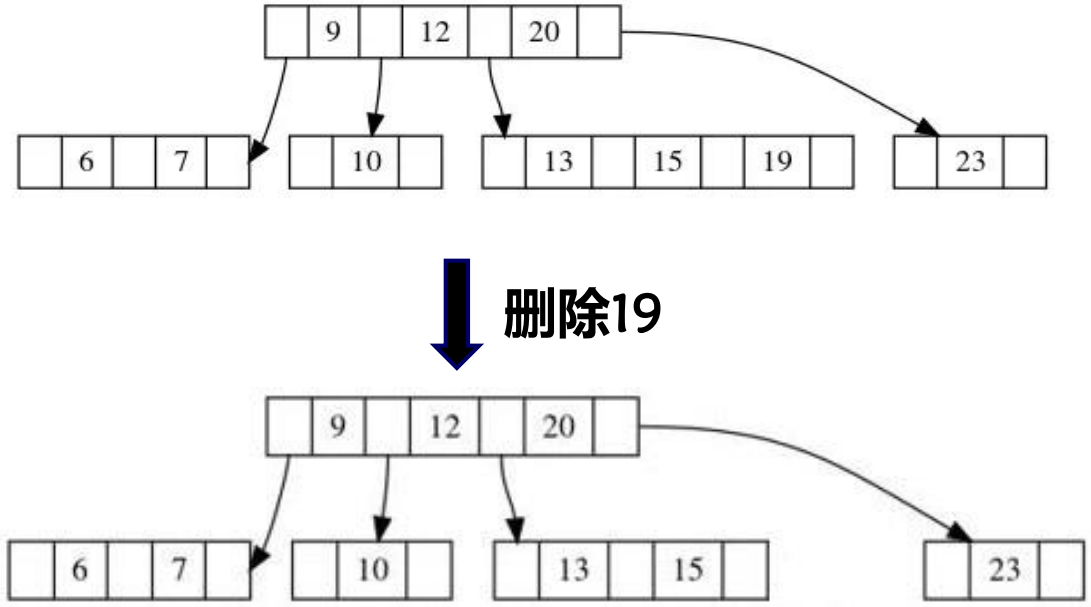


- B树叶子节点上的元素应是$K-1$个（$\left\lceil \frac{M}{2} \right\rceil \le K \le M$），删除叶子节点中的$key$可能会违反B树的如下性质：**叶子节点最少拥有$\lceil M/2 \rceil - 1$个键。**

B树操作 – 元素删除

所有的删除都可以转化成删除叶子节点的问题。

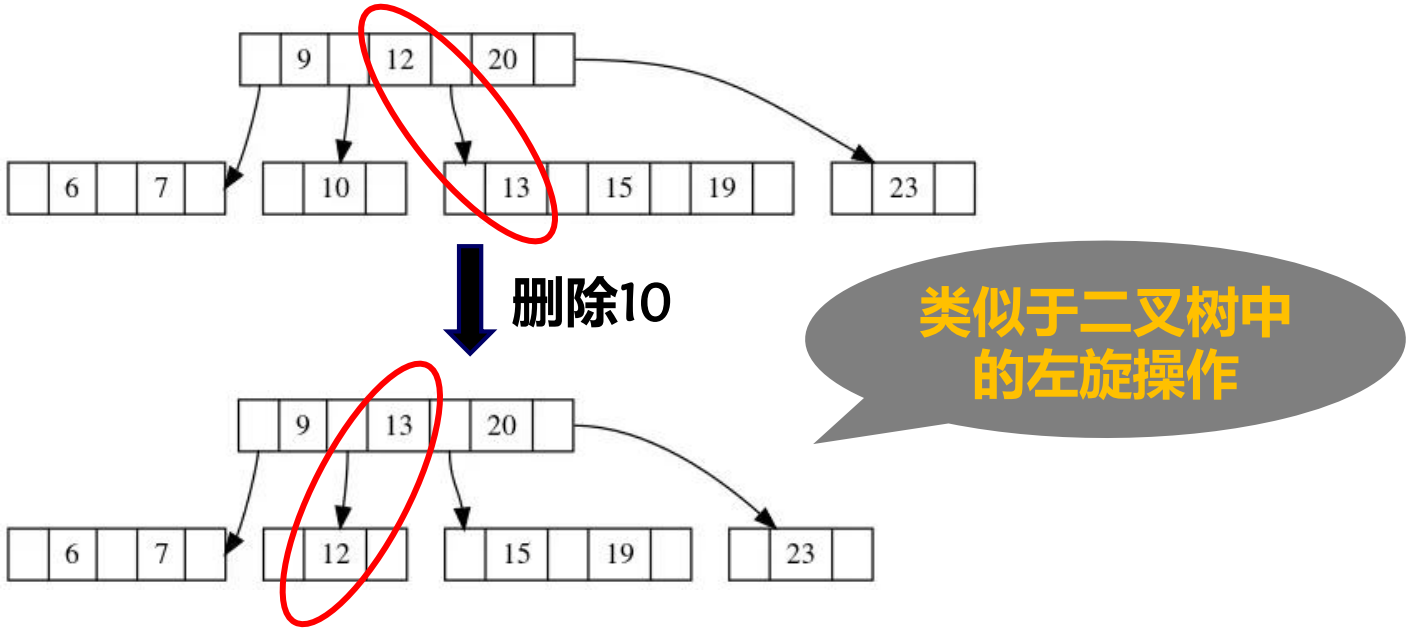
假设叶子节点当中元素数量很多，我们删除一个仍然可以保证它是合法的。这种情况很简单，直接删除即可，比如在如下B树中删除19。



B树操作 – 元素删除

所有的删除都可以转化成删除叶子节点的问题。

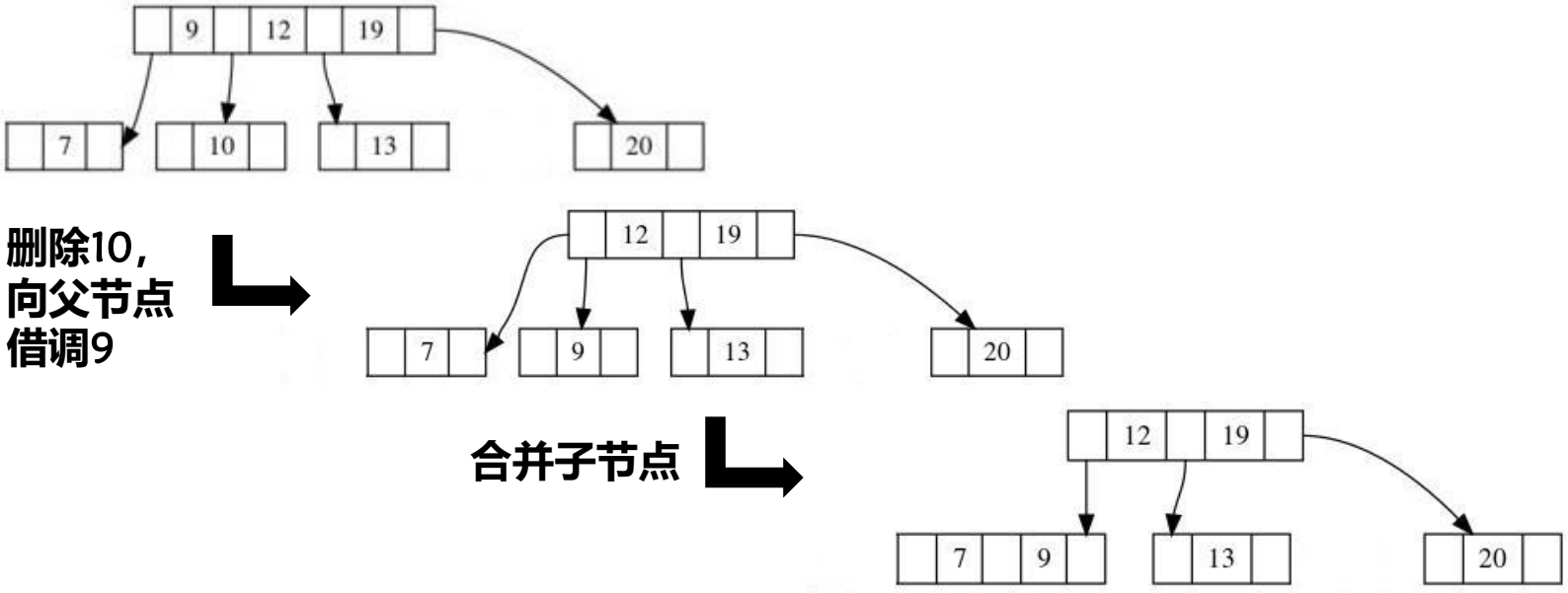
但如果我们删除的是10，由于节点10只有一个元素，如果删除了，那么就会破坏节点的最小元素数量的限制。这时我们可以从(左右)兄弟节点抽调key来使B树保持性质。



B树操作 – 元素删除

所有的删除都可以转化成删除叶子节点的问题。

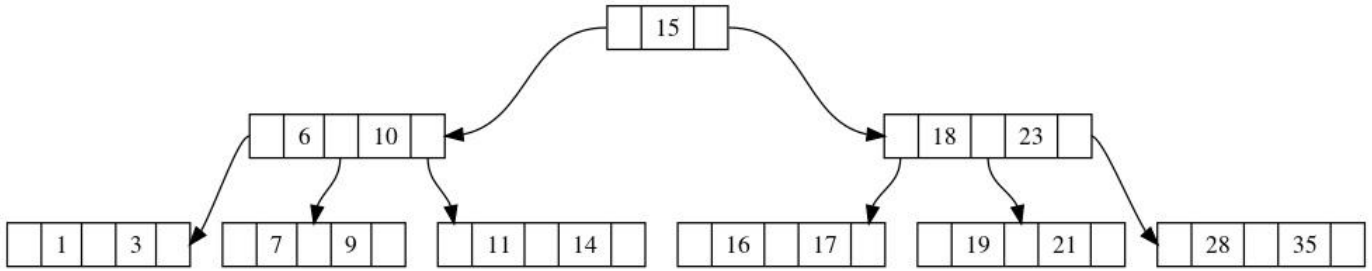
如果兄弟节点自身也是勉强达到最少key数目条件，显然是借不了的。这种时候只能和父亲节点借调。如果父亲节点稍稍富裕，给出了一个元素之后还是能满足条件，那么就从父亲节点出。但是这里需要注意，父亲节点给出去了一个元素，那么它的子树数量也应该随之减少，不然也会违反B树的性质。为此，可以通过合并两个子树来实现。



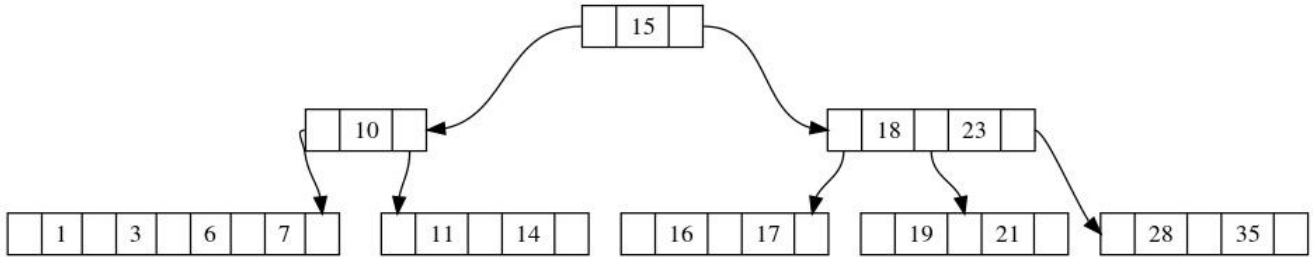
B树操作 – 元素删除

所有的删除都可以转化成删除叶子节点的问题。

- 若父节点减少了一个元素之后也不满足B树的条件，则需要递归借调节点的操作，让父节点去和它的兄弟节点以及父节点借调元素。在极端情况下，这很有可能导致树的高度发生变化：



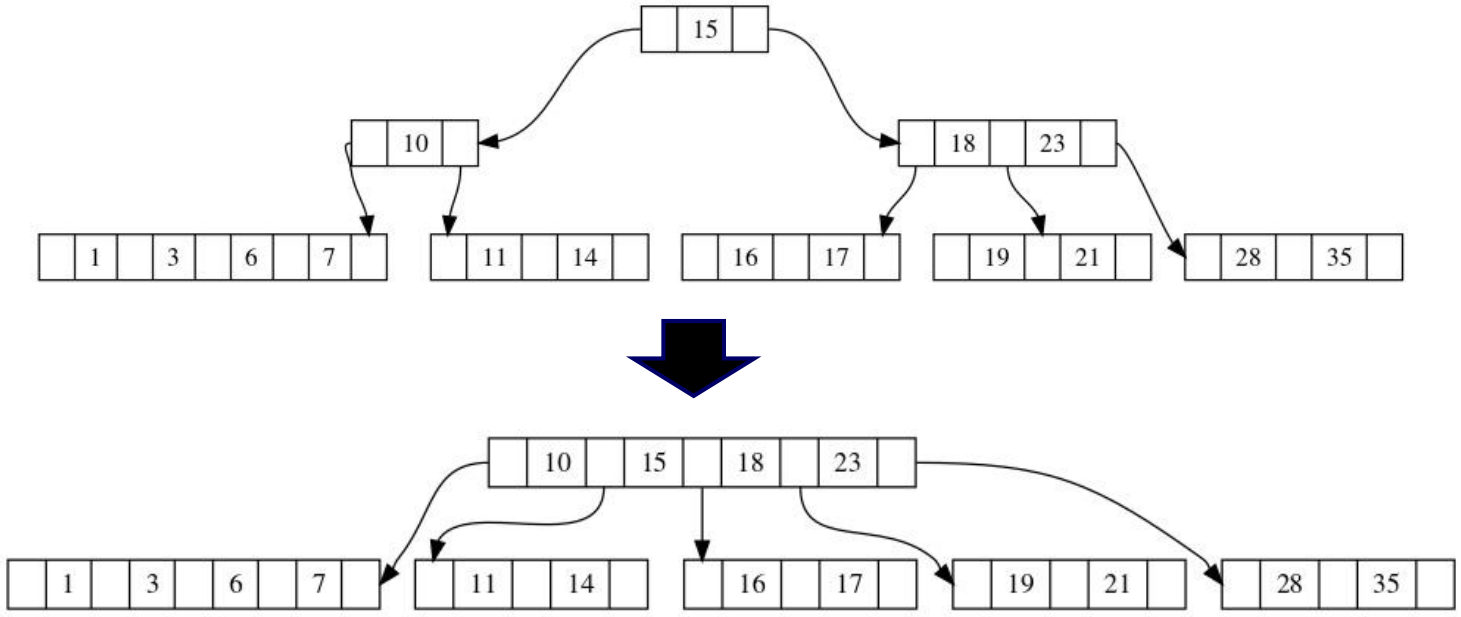
- 上图是一个阶数为5的B树，如果我们删除9，根据刚才所说的，我们会跟父亲节点借元素6，并且和[1, 3]子树合并，得到：



B树操作 – 元素删除

所有的删除都可以转化成删除叶子节点的问题。

但是这会导致父节点破坏了B树的最低元素要求，所以我们需要递归维护父节点，让父节点去重复借元素的步骤，我们可以发现对于10所在节点来说，它没有富裕的兄弟节点，只能继续向父节点借调，这会再一次导致合并的发生：



B树 – 时间复杂度分析

元素搜索

- B树根节点均存在最多/最少元素个数限制，且叶子节点都在同一层，故可以轻易得到B树的树高为$O(log_M N)$。
- 在B树中搜索一个元素的最坏情况就是在叶子节点处找到该元素或是该元素不存在。得到B树的高度之后我们便可以知道最坏情况下元素搜索的时间复杂度为$O(\log_M N) * O(log_2 M)$。其中$O(log_2 M)$为在一个节点中二分搜索所需的时间。

元素插入

- 与元素搜索相同，找到插入位置所需时间为：$O(log_M N) * O(log_2 M)$
- 为保持B树性质，可能的调整操作：$O(log_M N)$

综上，元素插入操作的时间复杂度为$O(\log_M N) * O(\log_2 M)$，其中：$M$为B树的阶数。

B树 – 时间复杂度分析

元素删除

- 找到待删除元素位置所需时间为：$O(\log_M N) * O(\log_2 M)$
- 将待删除元素转嫁到叶子节点所需时间：$O(\log_M N)$
- 删除后为保持B树性质，可能的调整操作：$O(\log_M N)$

综上，元素删除操作的时间复杂度为$O(log_M N) * O(log_2 M)$, 其中M为B树的阶数。

阶数M为常数，B树的搜索、插入、删除操作的时间复杂度均为：$O(log_M N)$

B+树-概念

(M阶) B+树的定义：

① 每个节点最多有M个子节点；

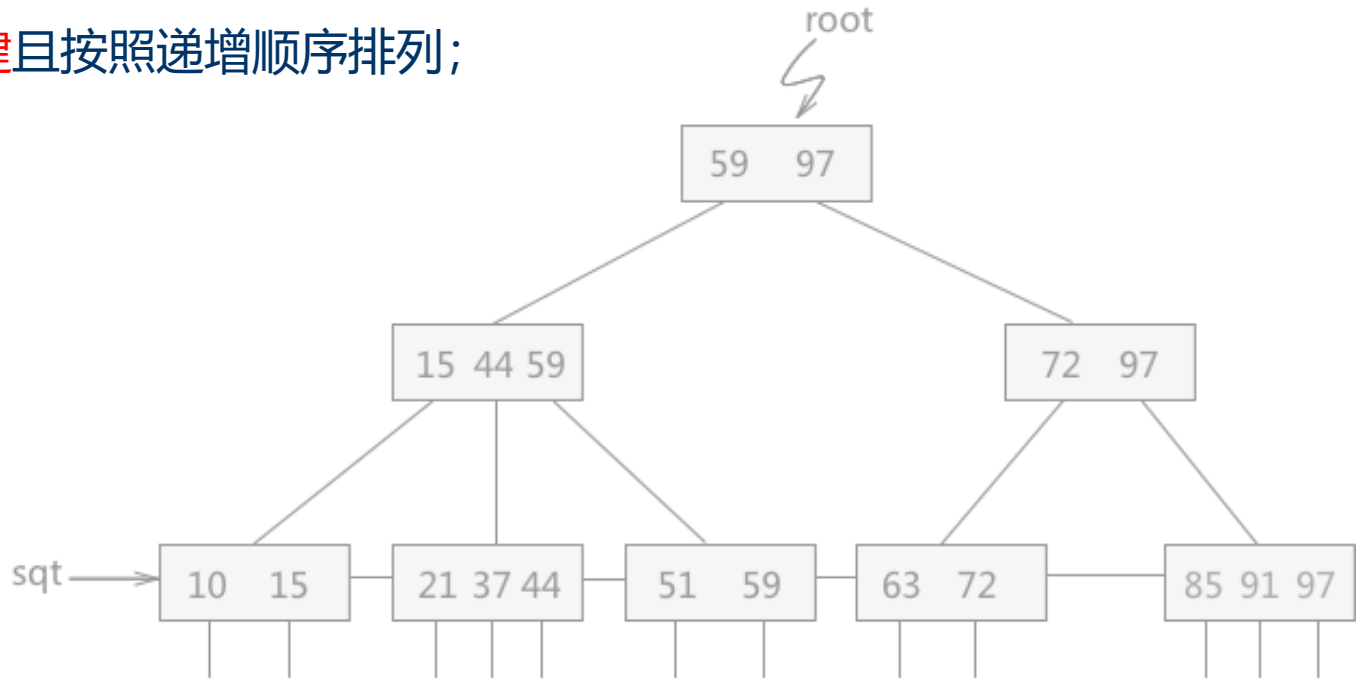
② 每一个非叶子节点(除根节点)最少有[M/2]个子节点；

③ 根节点若非叶子节点，那么它至少拥有两个子节点；

④ 有 k 个子节点的非叶子节点拥有 k 个键且按照递增顺序排列；

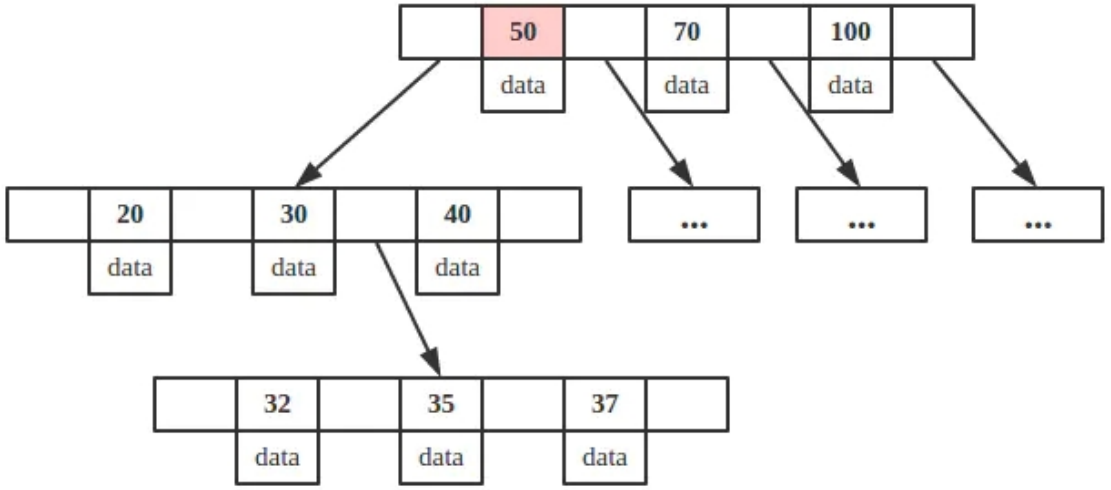
⑤ 叶子节点最少拥有[M/2]个键；

⑥ 所有叶子节点属于同一层。

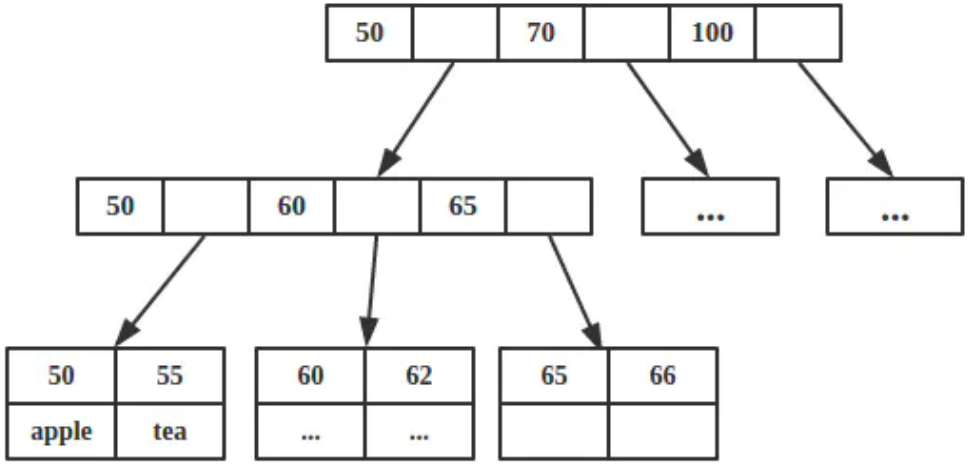


B+树和B树的差异

（M阶）B+树	（M阶）B树
有M颗子树的节点中含有M个关键码	有M颗子树的节点中含有$M-1$个关键码
所有叶子结点中包含了完整的索引信息，包括指向含有这些关键字记录的指针，中间节点每个元素不保存数据，只用来索引	B树中非叶子节点的关键码与叶子结点的关键码均不重复，它们共同构成全部的索引信息
所有的非叶子节点可以看成是高层索引， 结点中仅含有其子树根结点中最大（或最小）关键字	B 树的非叶子节点包含需要查找的有效信息



B-树



B+树

B+树的优势

B+树的磁盘读写代价更低

B+树的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了；

B+树查询效率更加稳定

由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当；

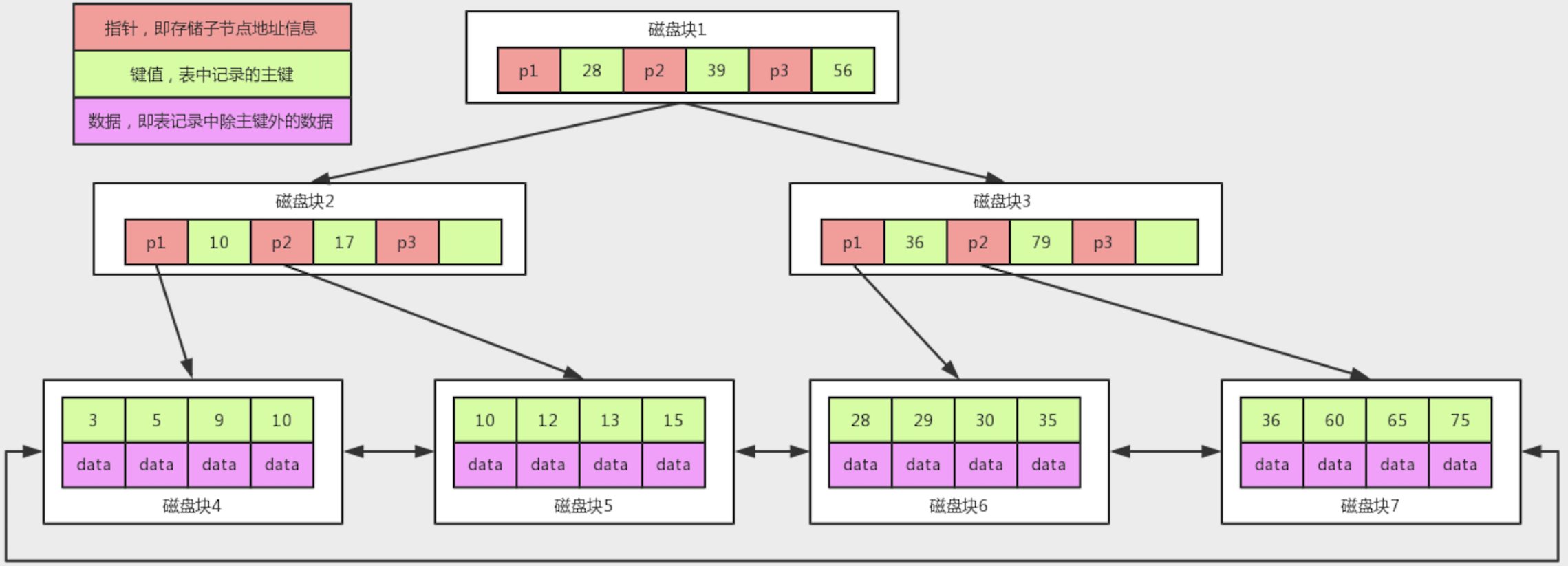
B+树便于范围查询（最重要的原因，范围查找是数据库的常态）

B树在提高了IO性能的同时并没有解决元素遍历的我效率低下的问题，正是为了解决这个问题，B+树应用而生。B+树只需要去遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作或者说效率太低。

B树的范围查找用的是中序遍历，而B+树用的是在链表上遍历。

B+树比B树更适合做数据库索引

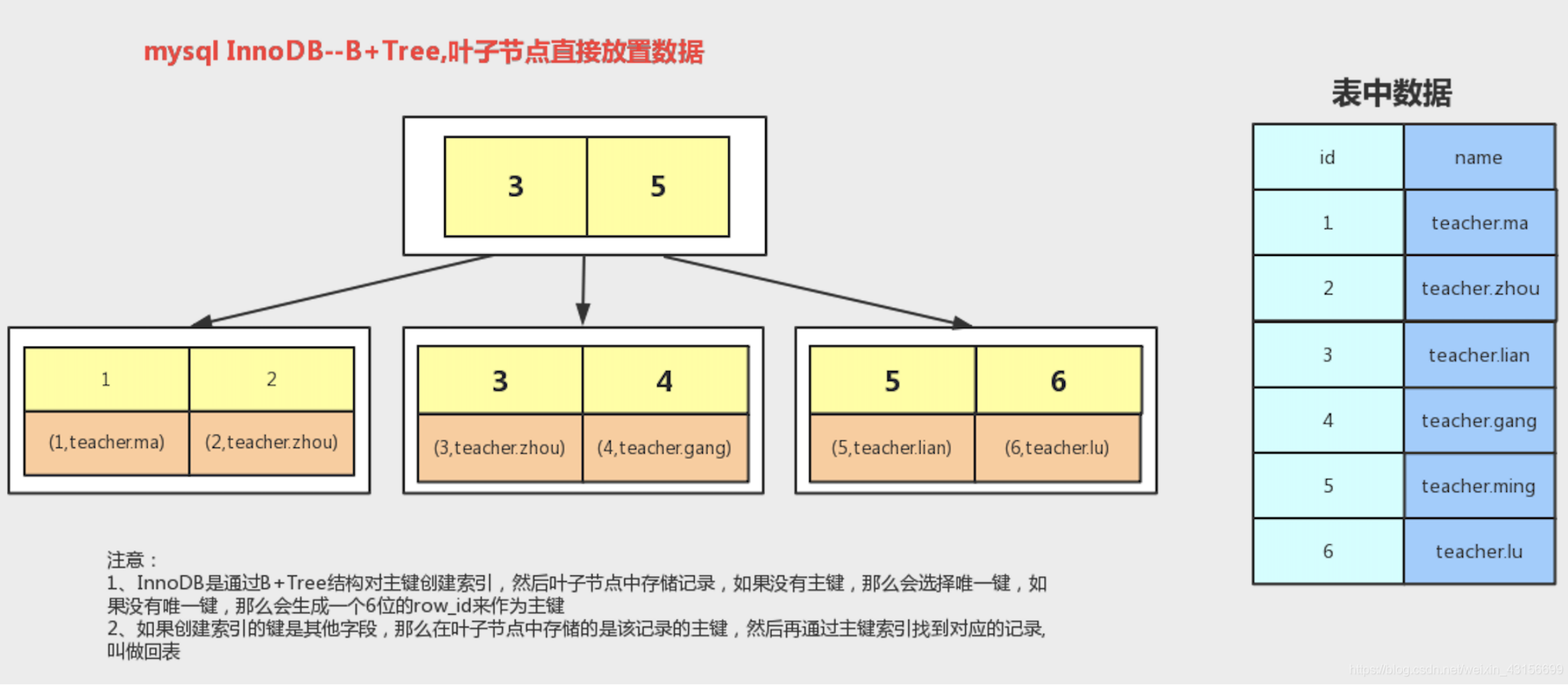
B+树：



注意：在B+Tree上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（即数据节点）之间是一种链式环结构。因此可以对 B+Tree 进行两种查找运算：一种是对于主键的范围查找和分页查找，另一种是从根节点开始，进行随机查找。

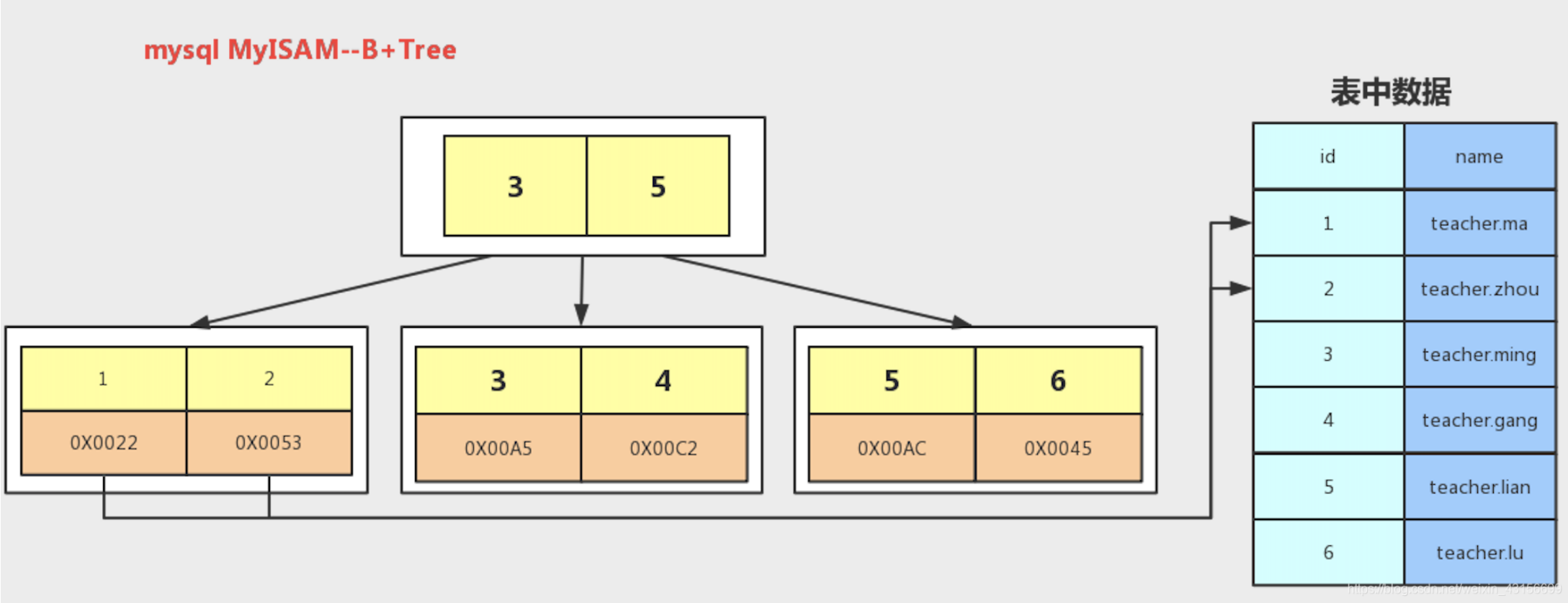
B+树比B树更适合做数据库索引

MySQL之InnoDB:



B+树比B树更适合做数据库索引

MySQL之MyISAM:



B+树操作—元素查找

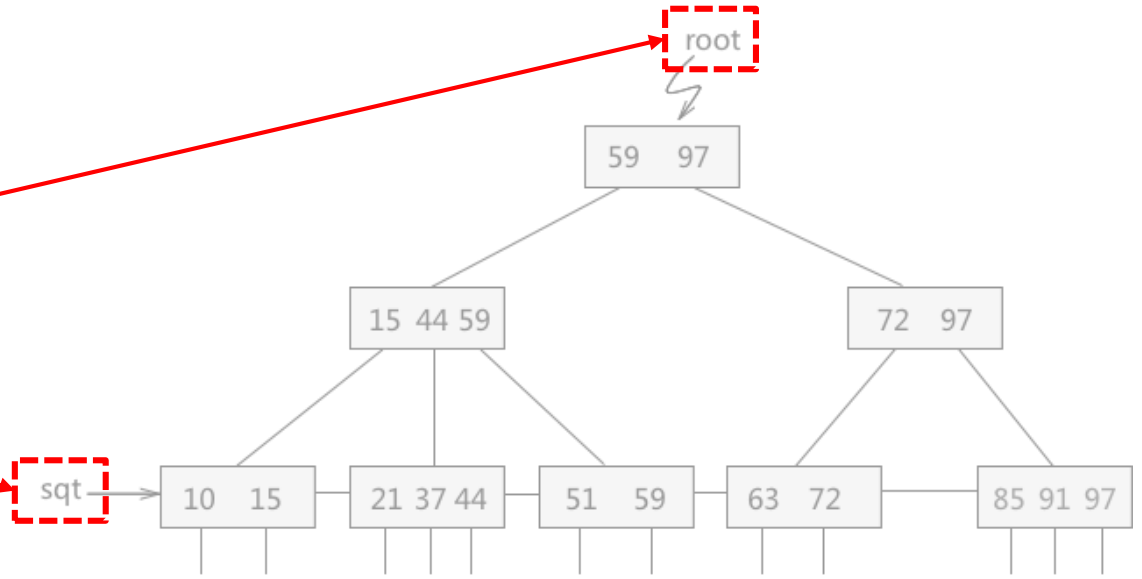
B+树搜索和B树搜索类似：

- 在 B+树中，所有非终端结点都相当于是终端结点的索引，而所有的关键字都存放在终端结点中。
- 所有在从根结点出发做查找操作时，**若在内部节点中找到检索的关键码时，检索并不会结束，要继续找到B+树的叶子结点为止。**
- B+树的查找操作，无论查找成功与否，每次查找操作都是走了一条从根结点到叶子结点的路径。

两种查找运算：

✓ 从根结点开始的二分查找；

✓ 从sqt链表开始的顺序查找。



B+树操作--元素插入

关键思路：

① 插入的操作全部都在叶子结点上进行，且不能破坏关键字自小而大的顺序；

② 由于 B+树中各结点中存储的关键字的个数有明确的范围，做插入操作可能会出现结点中关键字个数超过阶数的情况，此时需要将该结点进行“分裂”。

插入操作会出现3种情况：

① **情况1：**若被插入关键字所在的结点，其含有关键字数目小于阶数M，则直接插入结束；

② **情况2：**若被插入关键字所在的结点，其含有关键字数目等于阶数 M ，则需要将该结点分裂为两个结点，一个结点包含$[M/2]$，另一个结点包含$[M/2]$。同时将$[M/2]$的关键字上移至其双亲结点。假设其双亲结点中包含的关键字个数小于M，则插入操作完成；

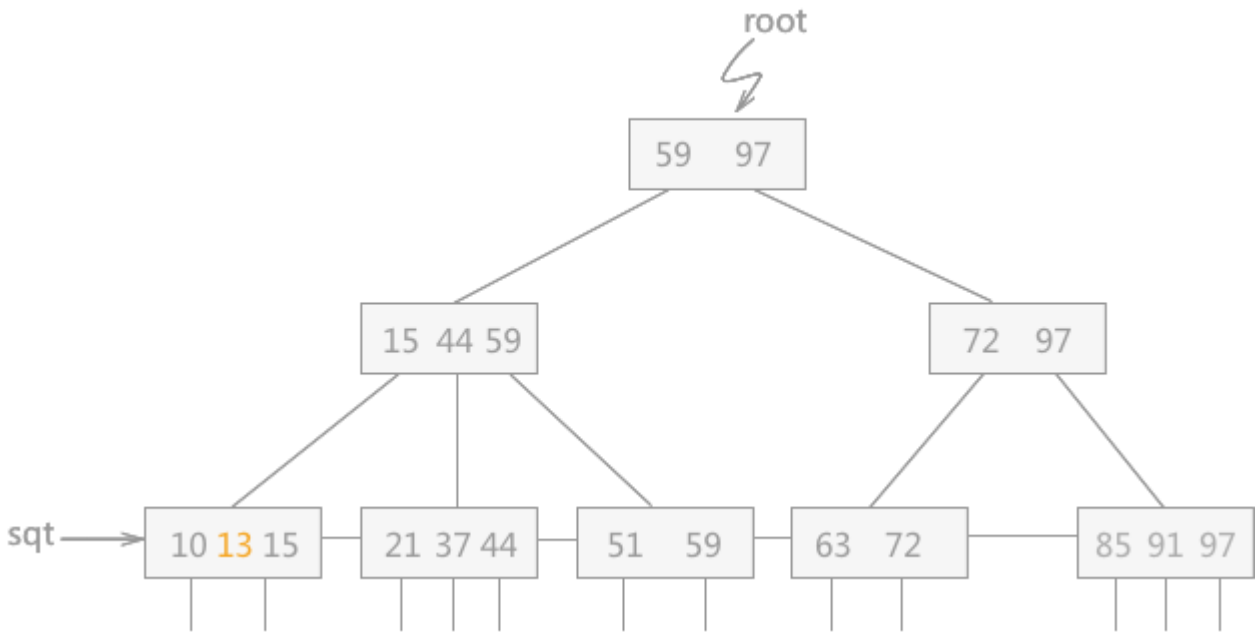
③ **情况3：**在第 2 情况中，如果上移操作导致其双亲结点中关键字个数大于 M ，则应继续分裂其双亲结点。

B+树操作--元素插入

插入操作的3 种情况：

- **情况1：**若被插入关键字所在的结点，其含有关键字数目小于阶数M，则直接插入结束；

插入关键字13后

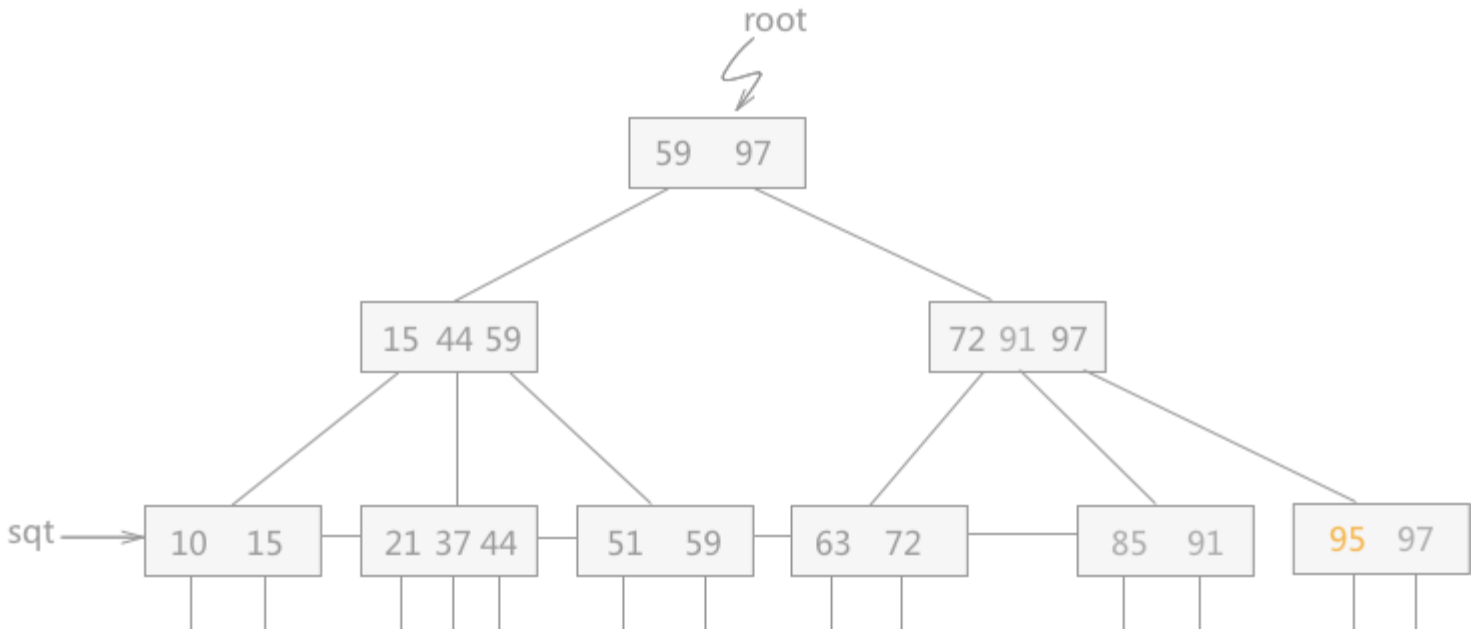


B+树操作--元素插入

插入操作的3 种情况：

- **情况2：**若被插入关键字所在的结点，其含有关键字数目等于阶数M，则需要将该结点分裂为两个结点，一个结点包含$\lfloor M/2 \rfloor$，另一个结点包含$\lceil M/2 \rceil$。同时，将$\lceil M/2 \rceil$的关键字上移至其双亲结点。假设其双亲结点中包含的关键字个数小于M，则插入操作完成。

插入关键字95后

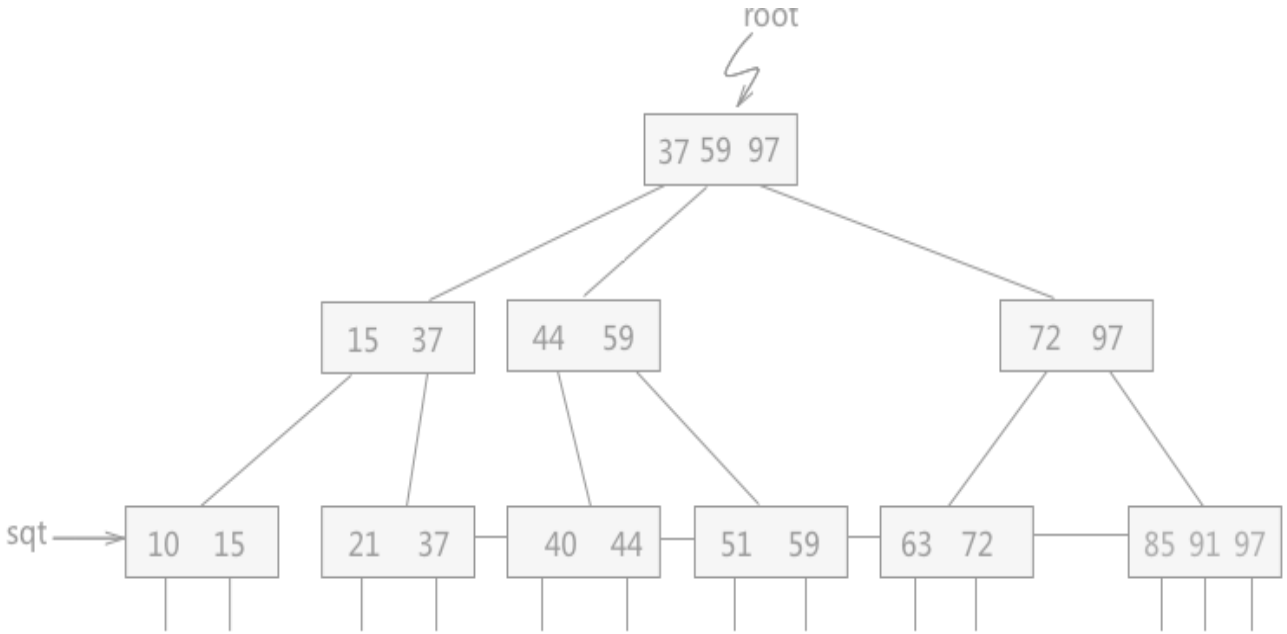


B+树操作--元素插入

插入操作的3 种情况：

- **情况3：**在第 2 情况中，如果上移操作导致其双亲结点中关键字个数大于M，则应继续分裂其双亲结点。

插入关键字40

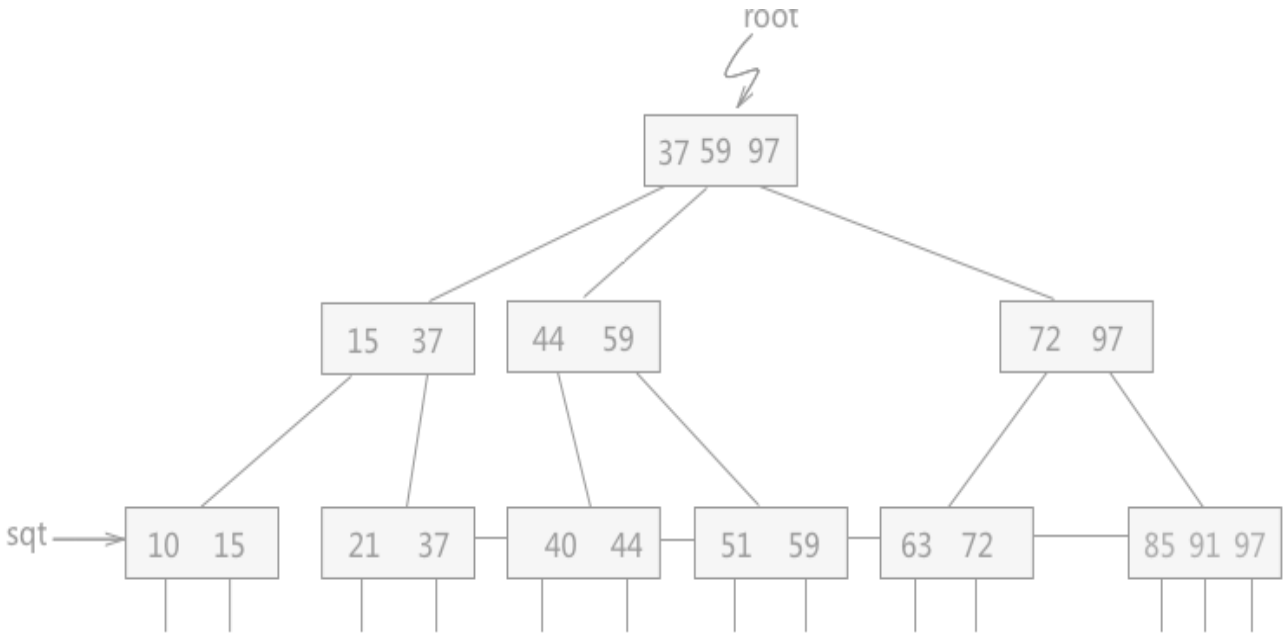


B+树操作--元素插入

插入操作的3 种情况：

- **情况3：**在第 2 情况中，如果上移操作导致其双亲结点中关键字个数大于M，则应继续分裂其双亲结点。

插入关键字40



如果插入的关键字比当前结点中的最大值还大，破坏了B+树中从根结点到当前结点的所有索引值，此时需要及时修正后，再做其他操作。例如，在图 1 的 B+树种插入关键字 100，由于其值比 97 还大，插入之后，从根结点到该结点经过的所有结点中的所有值都要由 97 改为 100。改完之后再做分裂操作。

B+树操作—元素删除

核心思路：

仅在叶节点删除关键码。若因为删除操作使得节点中关键码数少于 $\lceil M/2 \rceil$时，则需要调整或者和兄弟节点合并。合并的过程和B树类似，区别是父节点中作为分界的关键码不放入合并后的节点中。

主要步骤：

① 删除该关键字，如果不破坏 B+树本身的性质，直接完成操作；

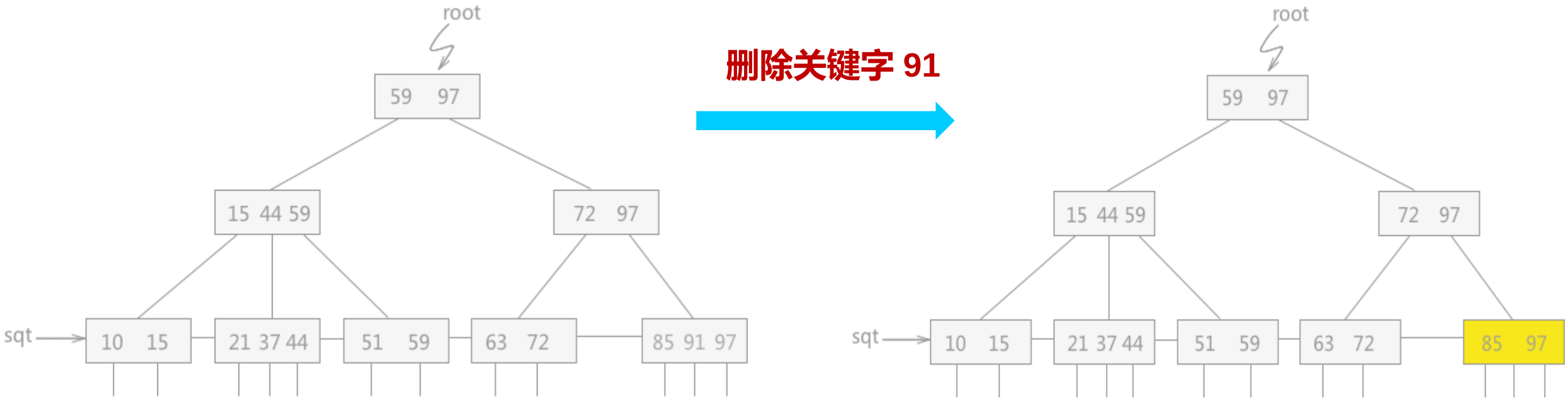
② 如果删除操作导致其该结点中最大（或最小）值改变，则应相应改动其父结点中的索引值；

③ 在删除关键字后，如果导致其结点中关键字个数不足，有两种方法：一种是向兄弟结点去借，另外一种是同兄弟结点合并。（注意这两种方式有时需要更改其父结点中的索引值。）

B+树操作—元素删除

删除操作的5种情况：

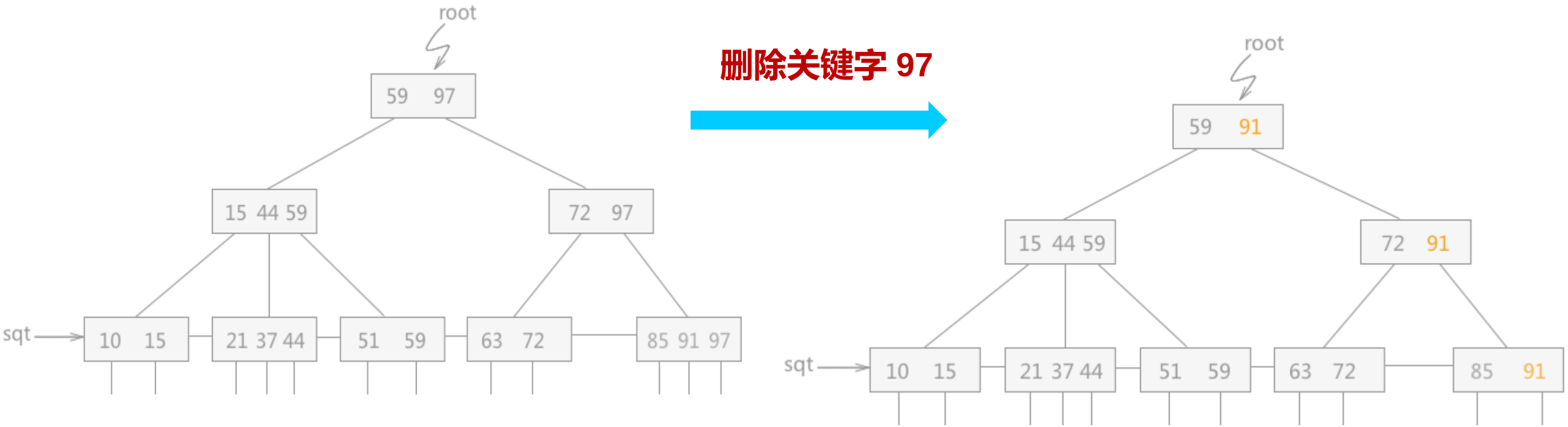
- **情况1：**找到存储有该关键字所在的结点时，由于该结点中关键字个数大于$\lceil M/2 \rceil$，做删除操作不会破坏 B+树，则可以直接删除。



B+树操作—元素删除

删除操作的5种情况：

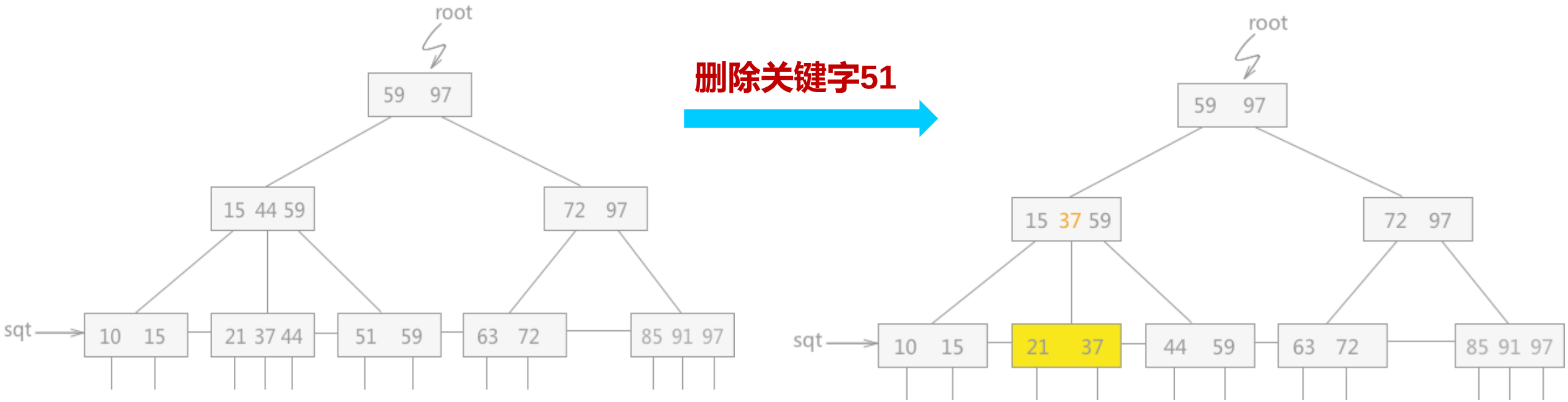
- **情况2：**当删除某结点中最大或者最小的关键字，就会涉及到更改其双亲结点一直到根结点中所有索引值的更改。



B+树操作—元素删除

删除操作的5种情况：

- **情况3：**当删除该关键字，导致当前结点中关键字个数小于 $\lceil M/2 \rceil$ ，若其兄弟结点中含有多余的关键字，可以从兄弟结点中借关键字完成删除操作。

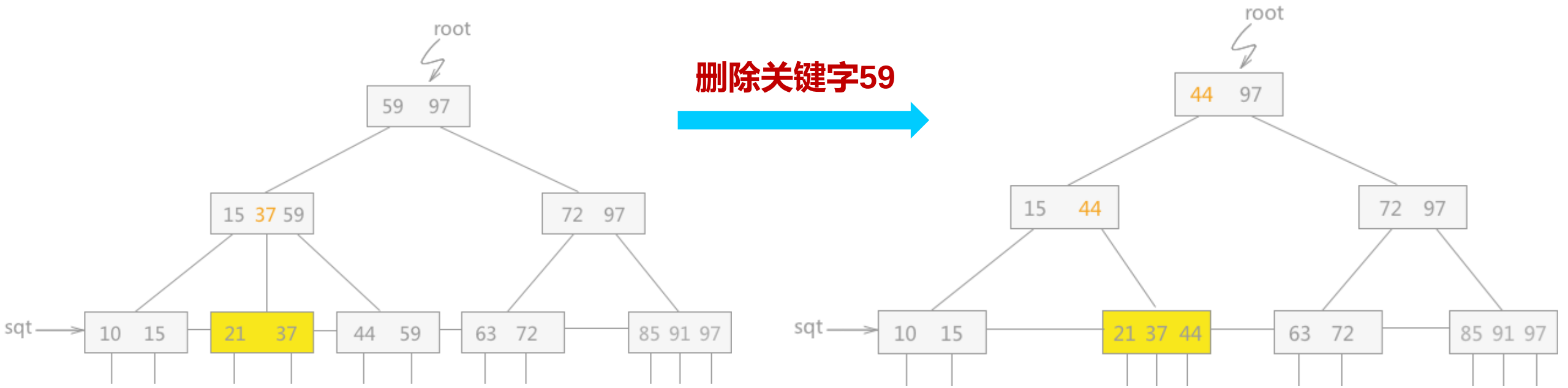


在左图B+树中删除关键字 51，由于其兄弟结点中含有 3 个关键字，所以可以选择借一个关键字同时修改双亲结点中的索引值。

B+树操作—元素删除

删除操作的5种情况：

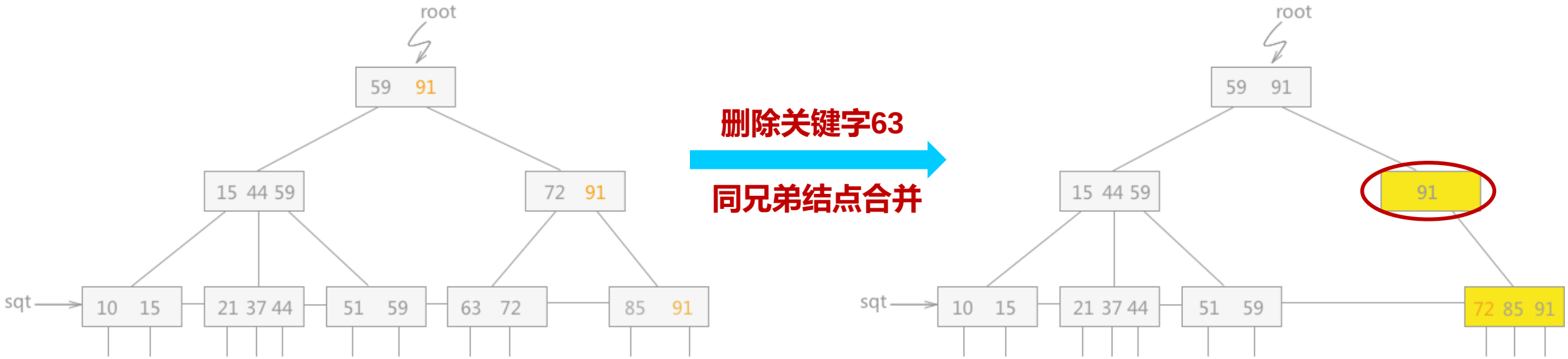
- **情况4：**第 3 种情况中，如果其兄弟结点没有多余的关键字，则需要同其兄弟结点进行合并。



B+树操作—元素删除

删除操作的5种情况：

- **情况5**：当进行合并时，可能会产生因合并使其双亲结点破坏 B+树的结构，需要依照以上规律处理其双亲结点。

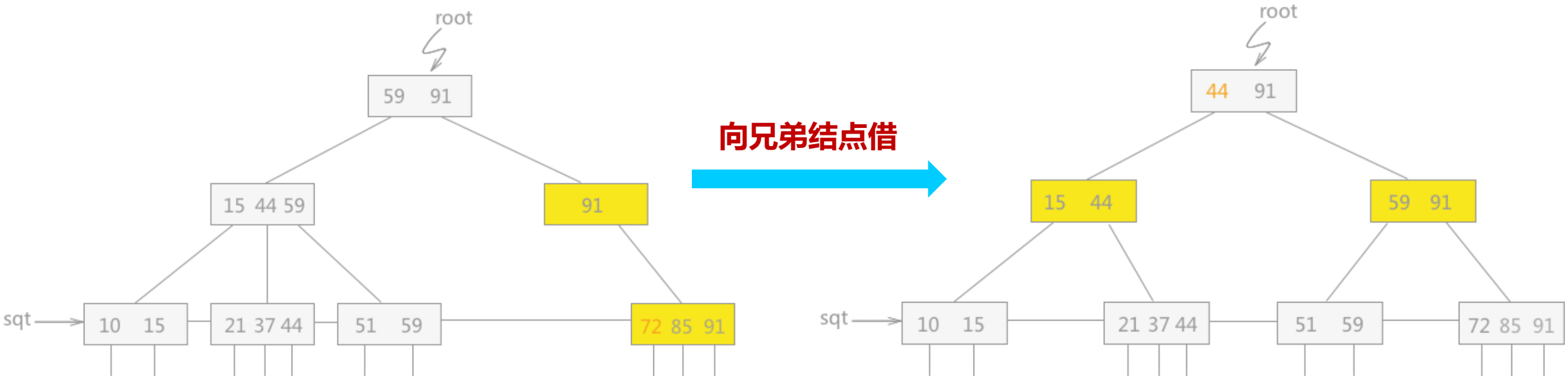


删除关键字63后只剩关键字72，且兄弟结点只有2个关键字，无法实现借的操作，只能进行合并。但是，合并后如右图所示，此时父节点只有1个关键字，不满足3阶B+树性质。

B+树操作—元素删除

删除操作的5种情况：

- **情况5：**当进行合并时，可能会产生因合并使其双亲结点破坏 B+树的结构，需要依照以上规律处理其双亲结点。



合并后由于双亲结点中只有一个关键字，其兄弟结点中有 3 个关键字，故可以通过借的操作来满足B+树的性质。最终结果如右图所示。

B*树-概念

(M阶) B*树的特点：

① 是B+树的变体，在B+树的非根和非叶子结点再增加指向兄弟的指针；

② B*树定义了非叶子结点关键字个数至少为$(2/3)*M$，即块的最低使用率为2/3（代替B+树的1/2）；

③ B*树分配新结点的概率比B+树要低，空间使用率更高。

B+树的分裂

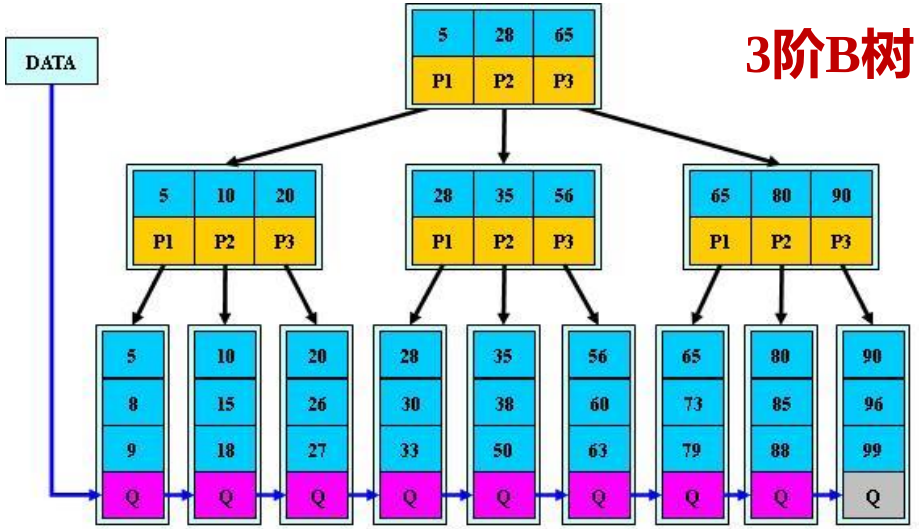
- 当一个结点满时，分配一个新的结点，并将原结点中1/2的数据复制到新结点，最后在父结点中增加新结点的指针；B+树的分裂只影响原结点和父结点，而不会影响兄弟结点，所以它不需要指向兄弟的指针；

B*树的分裂

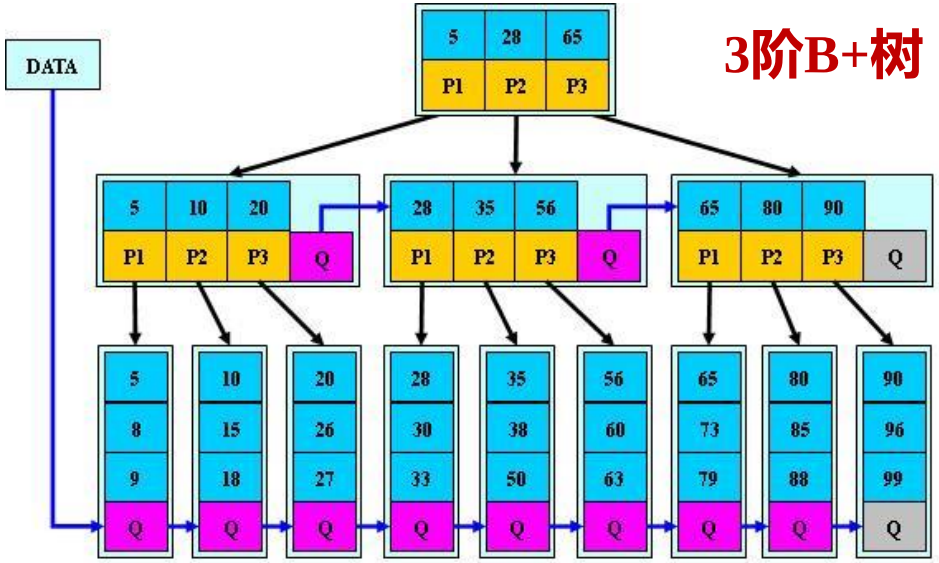
- 当一个结点满时，如果它的下一个兄弟结点未满，那么将一部分数据移到兄弟结点中，再在原结点插入关键字，最后修改父结点中兄弟结点的关键字（因为兄弟结点的关键字范围改变了）；如果兄弟也满了，则在原结点与兄弟结点之间增加新结点，并各复制1/3的数据到新结点，最后在父结点增加新结点的指针；

B*树-概念

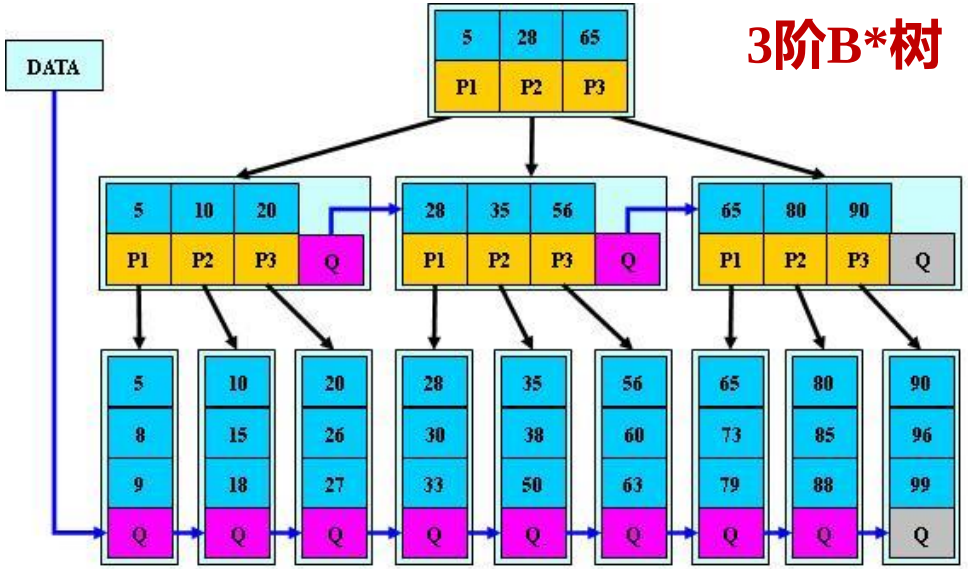
例子：



3阶B树



3阶B+树



3阶B*树

THANKS

敬　请　指　正