平衡树基础

胡船长

初航我带你，远航靠自己

本章题目

本章最大的练习题，就是：

学会每一个结构的代码细节

本期内容

一. （2）二叉排序树

二. （2）AVL 树

三. （4）红黑树

四. （4）B-树

一. 二叉排序树

二叉排序树

名称：二叉排序树、二叉搜索树

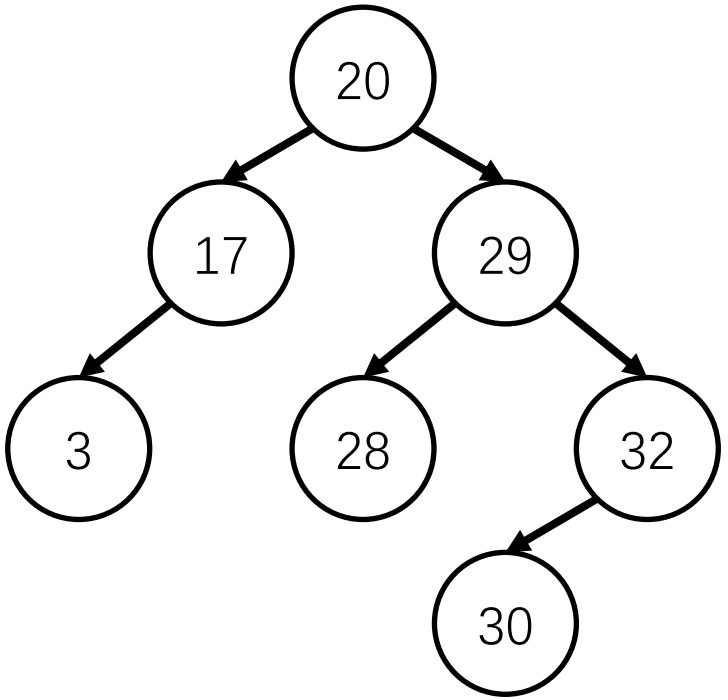
性质：

1、左子树 < 根节点

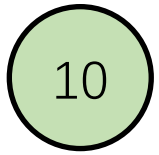
2、右子树 > 根节点

用途：

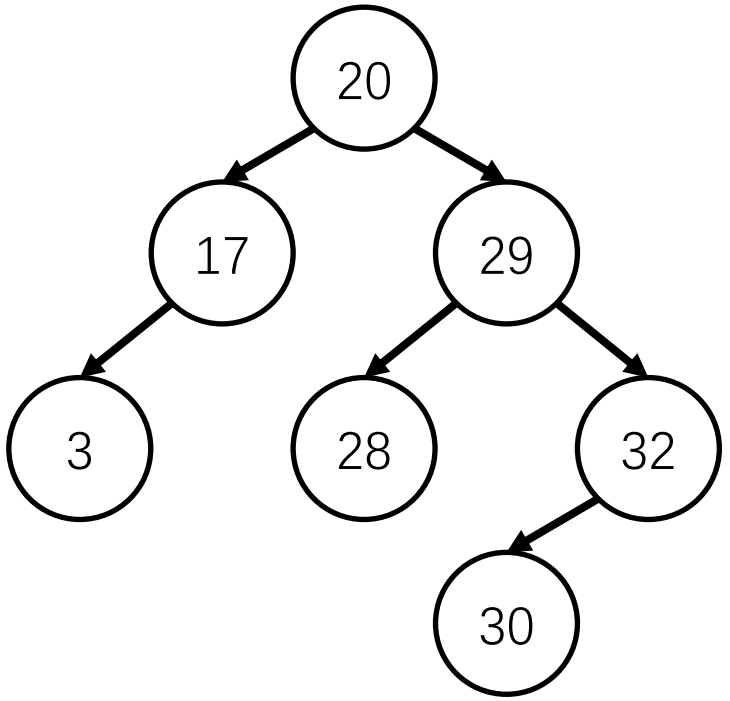
解决与排名相关的检索需求



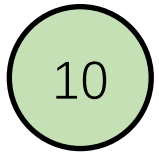
二叉排序树的插入



待插入：10

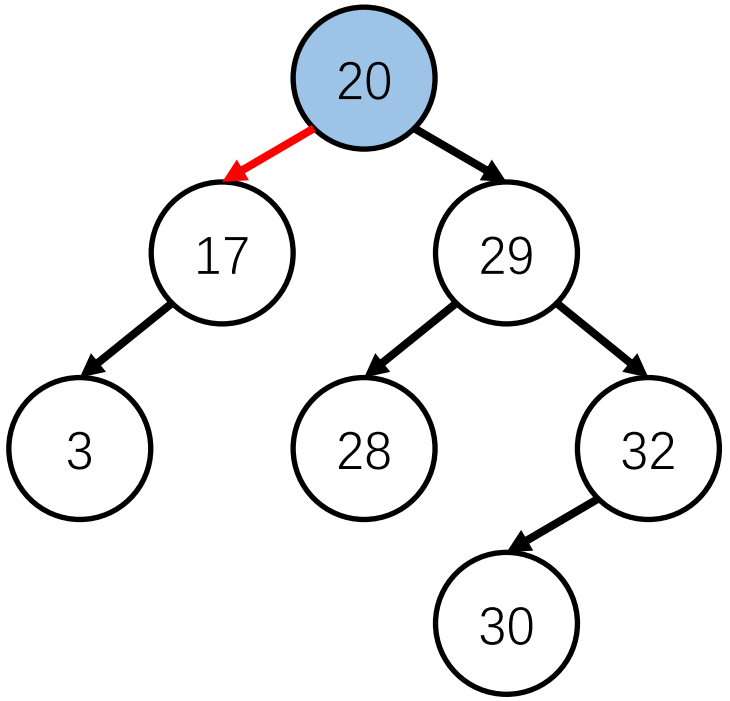


二叉排序树的插入

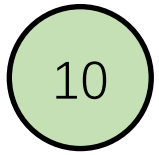


10 < 20

插入到左子树

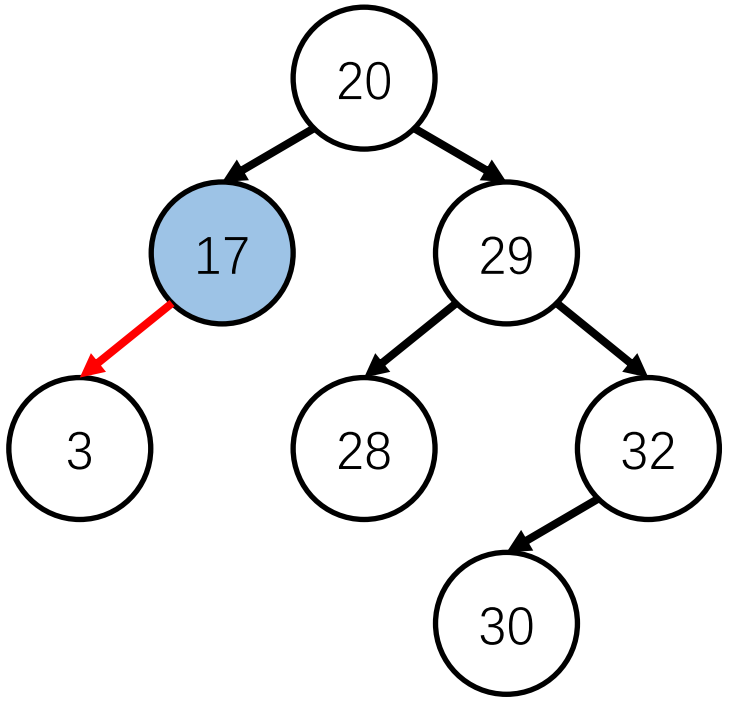


二叉排序树的插入

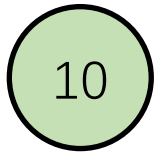


10 < 17

插入到左子树

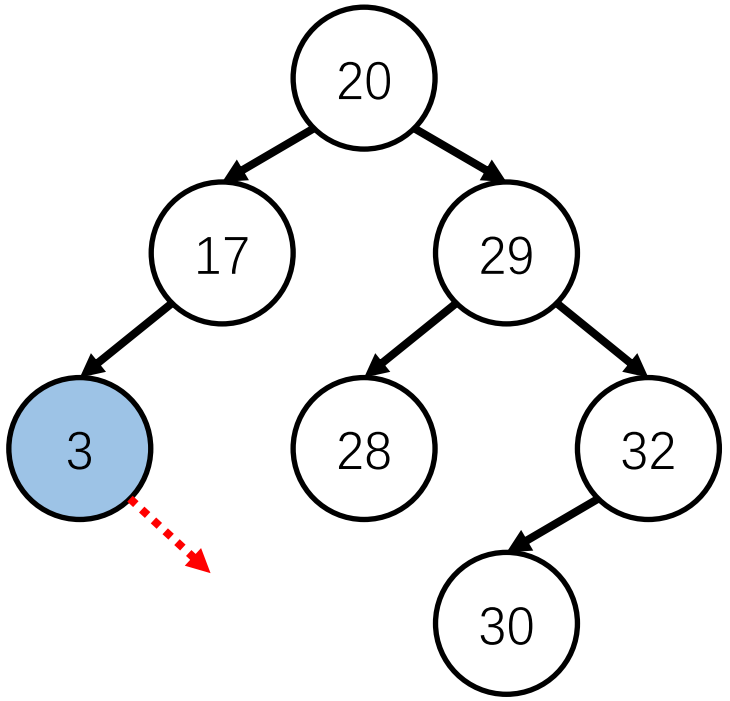


二叉排序树的插入



10 > 3

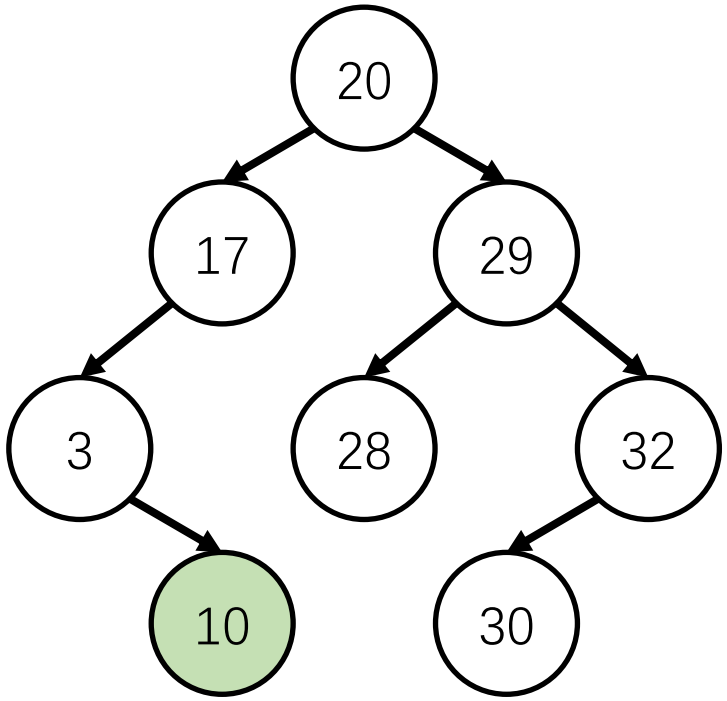
插入到右子树



二叉排序树的插入

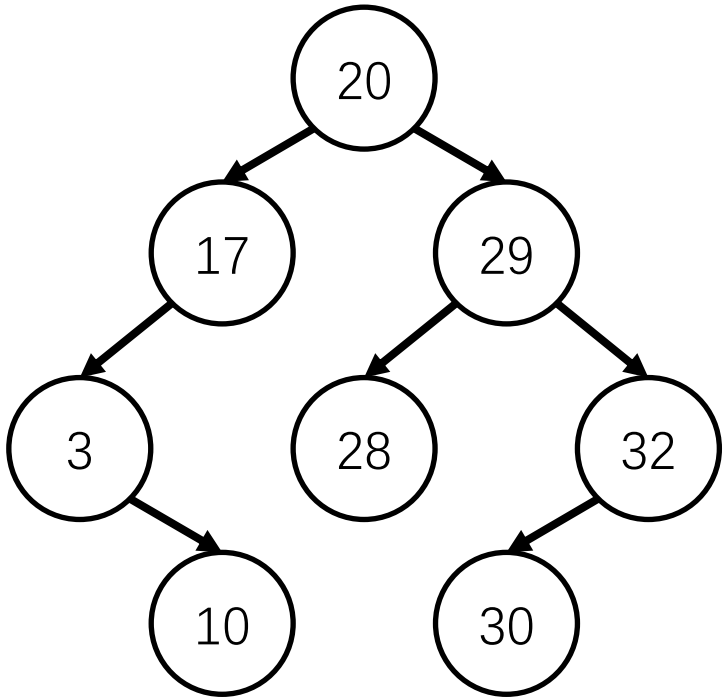
10 > 3

插入到右子树



二叉排序树的删除

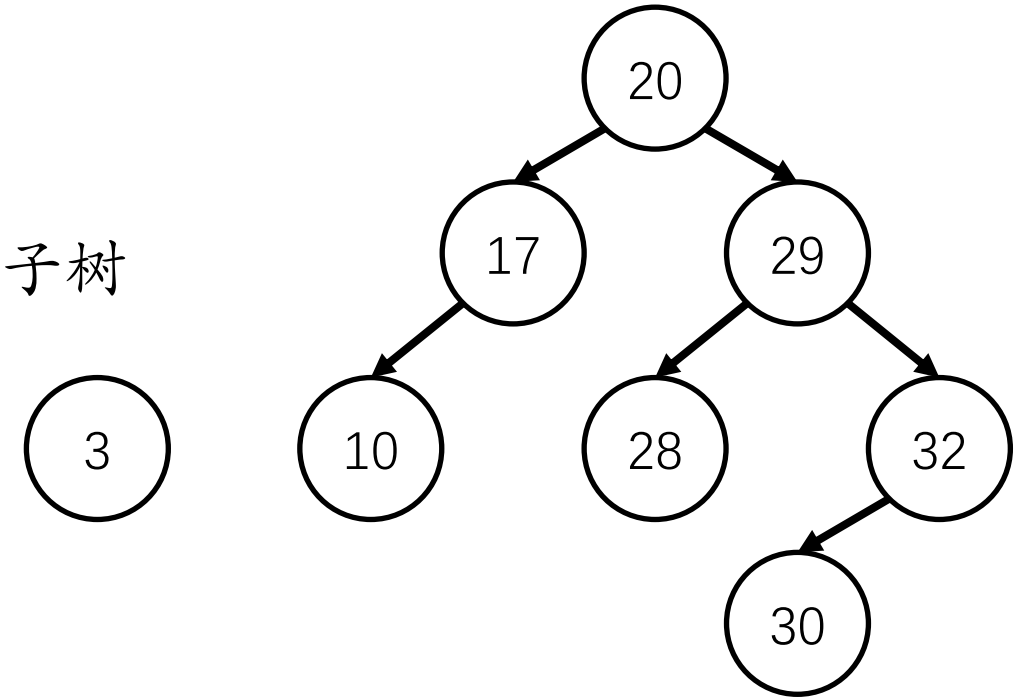
1、删除叶子节点
2、删除出度为1的节点
3、删除出度为2的节点



二叉排序树的删除

删除出度为1的节点

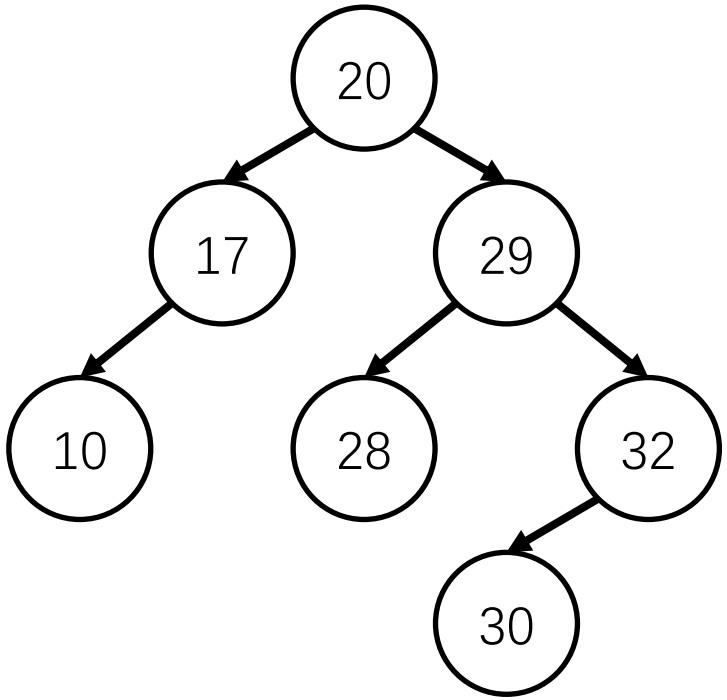
提升3的唯一子树



二叉排序树的删除

删除出度为1的节点

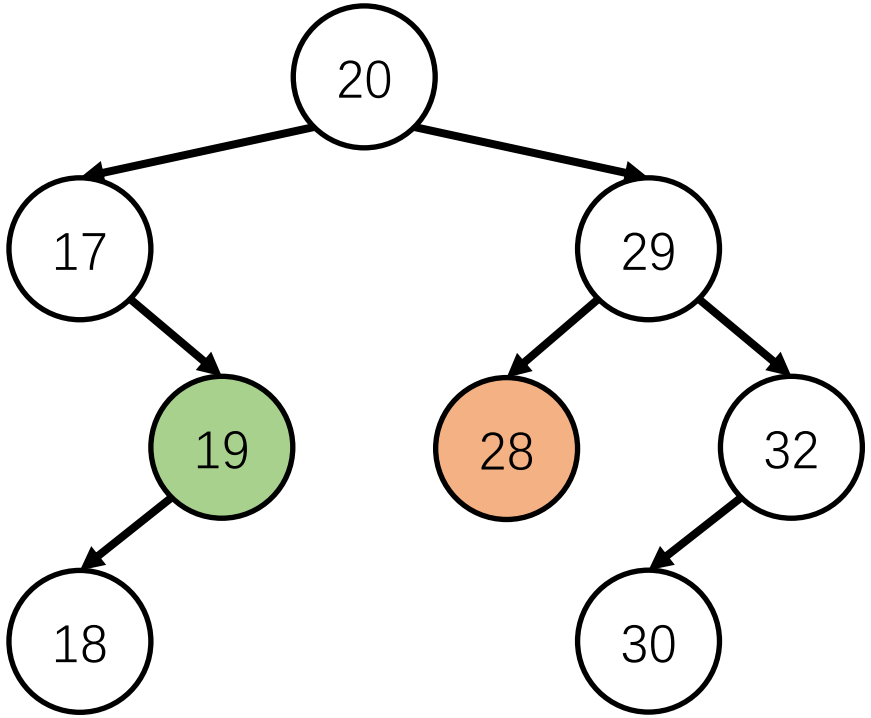
提升3的唯一子树



二叉排序树的删除

删除出度为2的节点

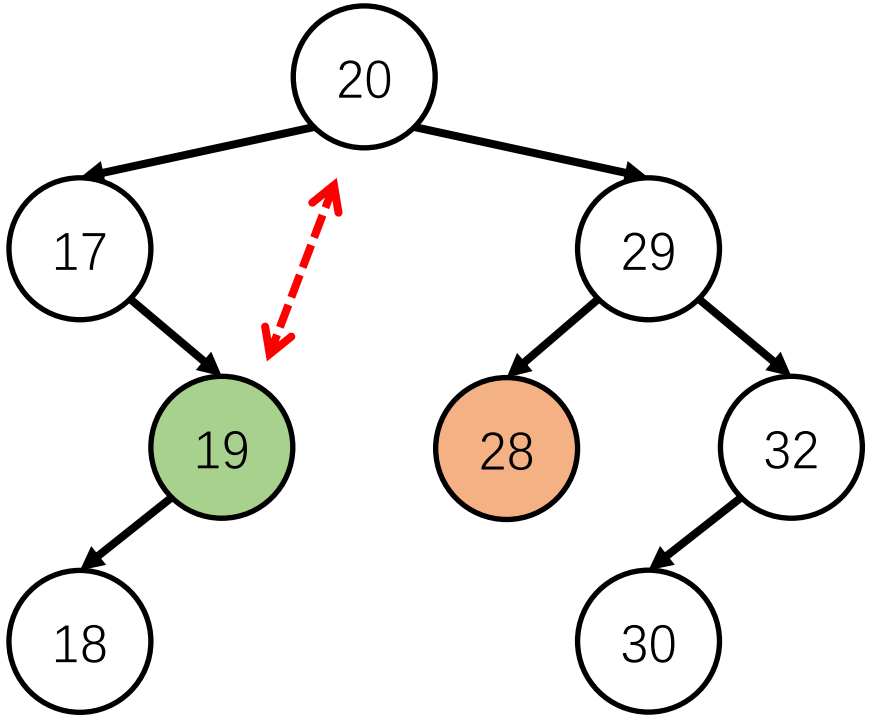
找到前驱或者后继替换后
转换为度为1的节点问题



二叉排序树的删除

删除出度为2的节点

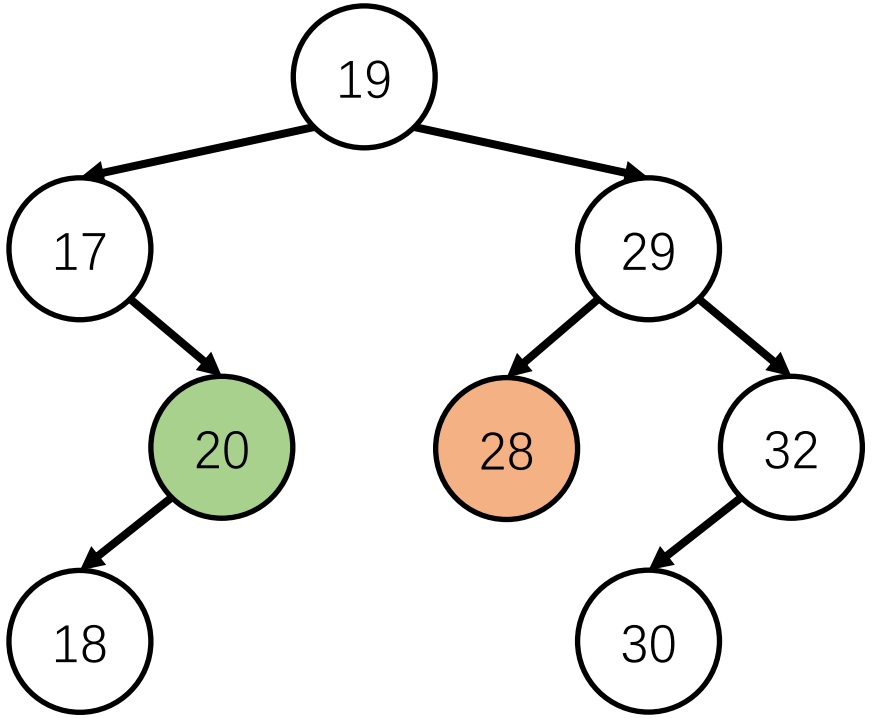
找到前驱或者后继替换后
转换为度为1的节点问题



二叉排序树的删除

删除出度为2的节点

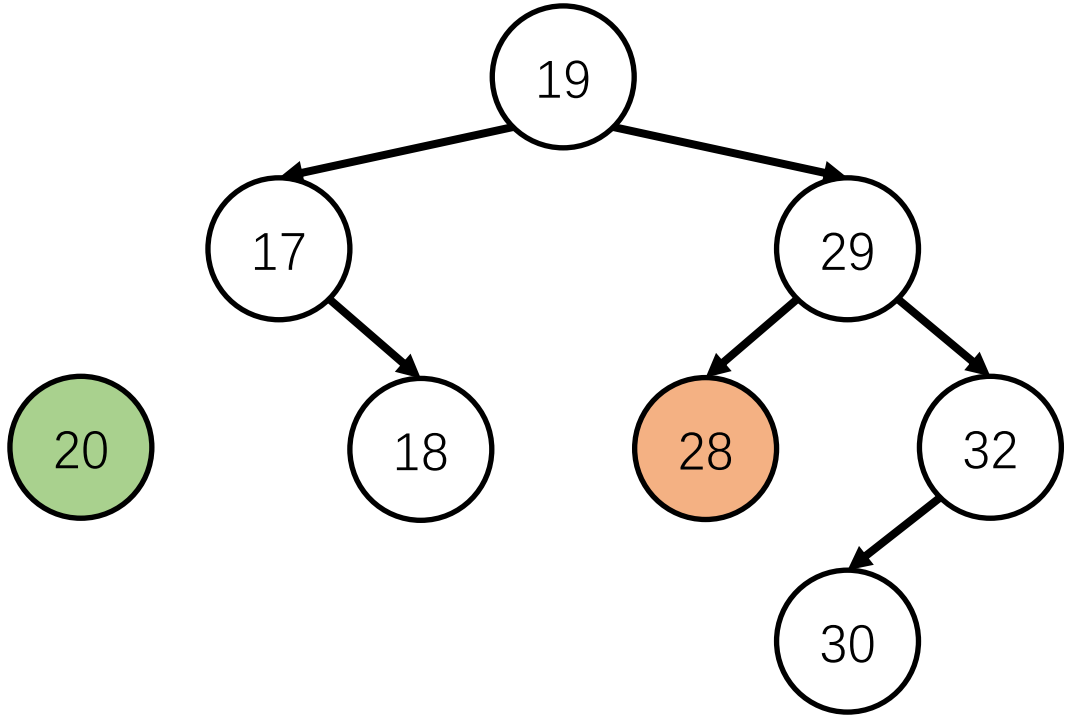
找到前驱或者后继替换后
转换为度为1的节点问题



二叉排序树的删除

删除出度为2的节点

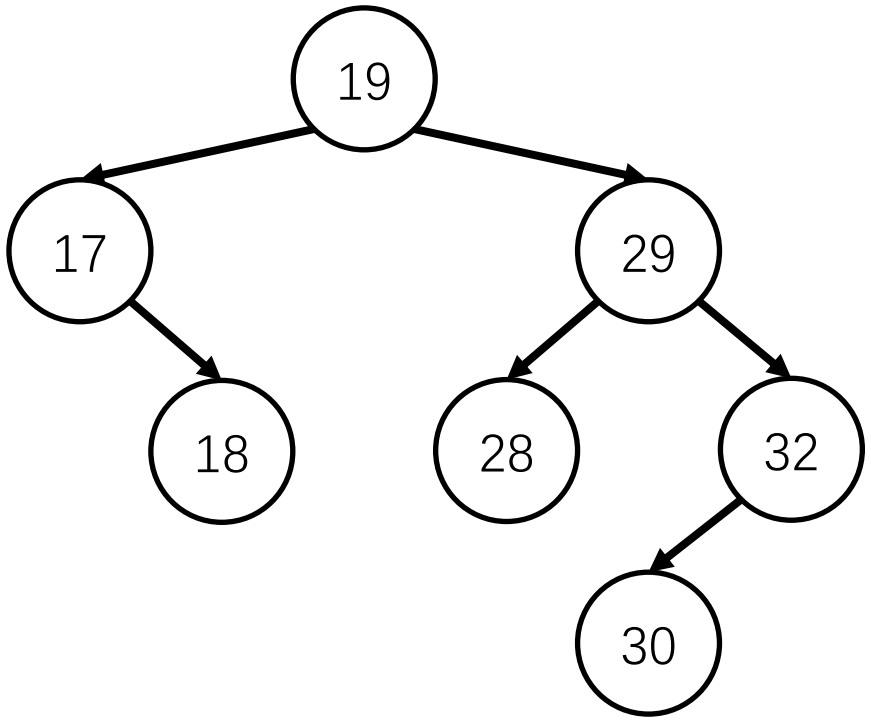
找到前驱或者后继替换后
转换为度为1的节点问题

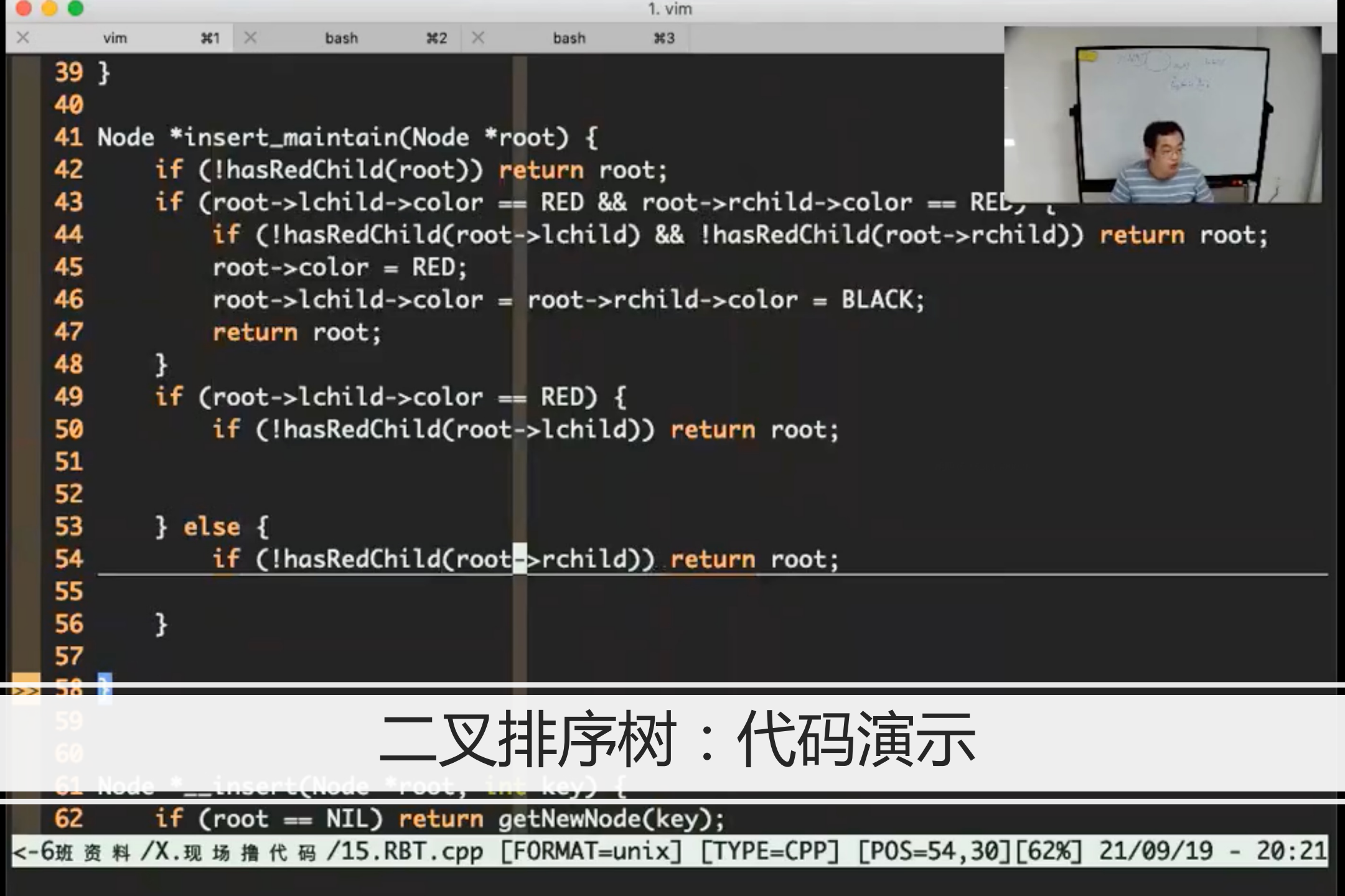


二叉排序树的删除

删除出度为2的节点

找到前驱或者后继替换后
转换为度为1的节点问题




1. vim
vim ⌘1
bash ⌘2
bash ⌘3
39 }
40
41 Node *insert_maintain(Node *root) {
42 if (!hasRedChild(root)) return root;
43 if (root->lchild->color == RED && root->rchild->color == RED) {
44 if (!hasRedChild(root->lchild) && !hasRedChild(root->rchild)) return root;
45 root->color = RED;
46 root->lchild->color = root->rchild->color = BLACK;
47 return root;
48 }
49 if (root->lchild->color == RED) {
50 if (!hasRedChild(root->lchild)) return root;
51
52
53 } else {
54 if (!hasRedChild(root->rchild)) return root;
55
56 }
57
58 }
59
60
61 Node *__insert(Node *root, int key) {
62 if (root == NIL) return getNewNode(key);
<-6班 资 料 /X.现 场 撸 代 码 /15.RBT.cpp [FORMAT=unix] [TYPE=CPP] [POS=54,30][62%] 21/09/19 - 20:21

二叉排序树：代码演示

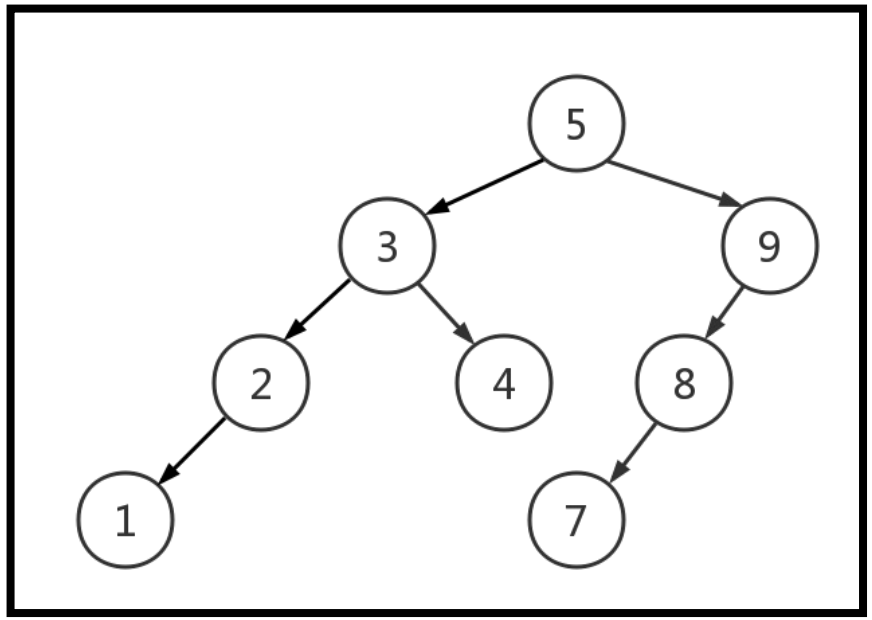
二. AVL 树

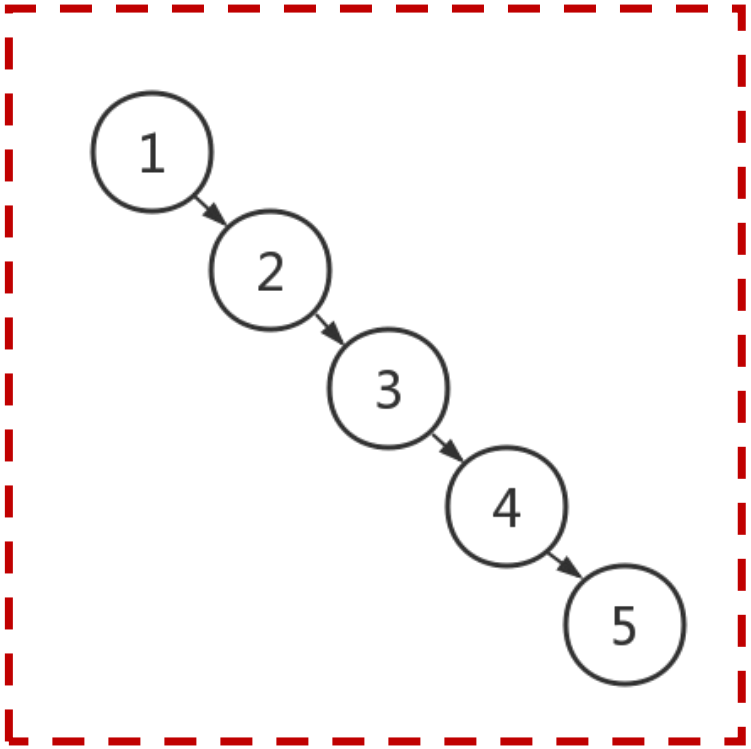
随堂练习-1

请按照如下顺序插入数字，画出对应的二叉搜索树

```
1 : [ 5 9 8 3 2 4 1 7 ]
2 : [ 1 2 3 4 5 ]
```

随堂练习-1





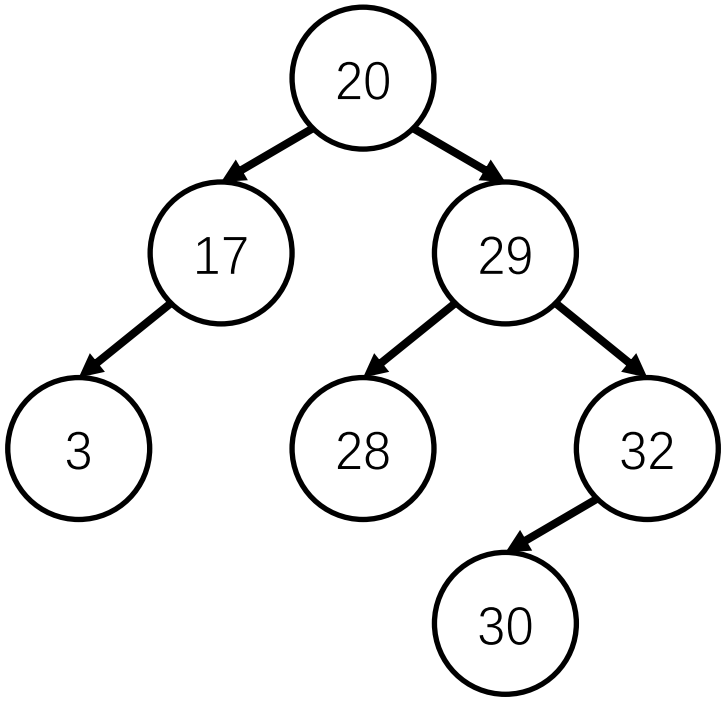
AVL 树

名称：AVL 树
发明者：
G.M. **A**delson-**V**elsky
E.M. **L**andis
年代：1962 年（61 岁）

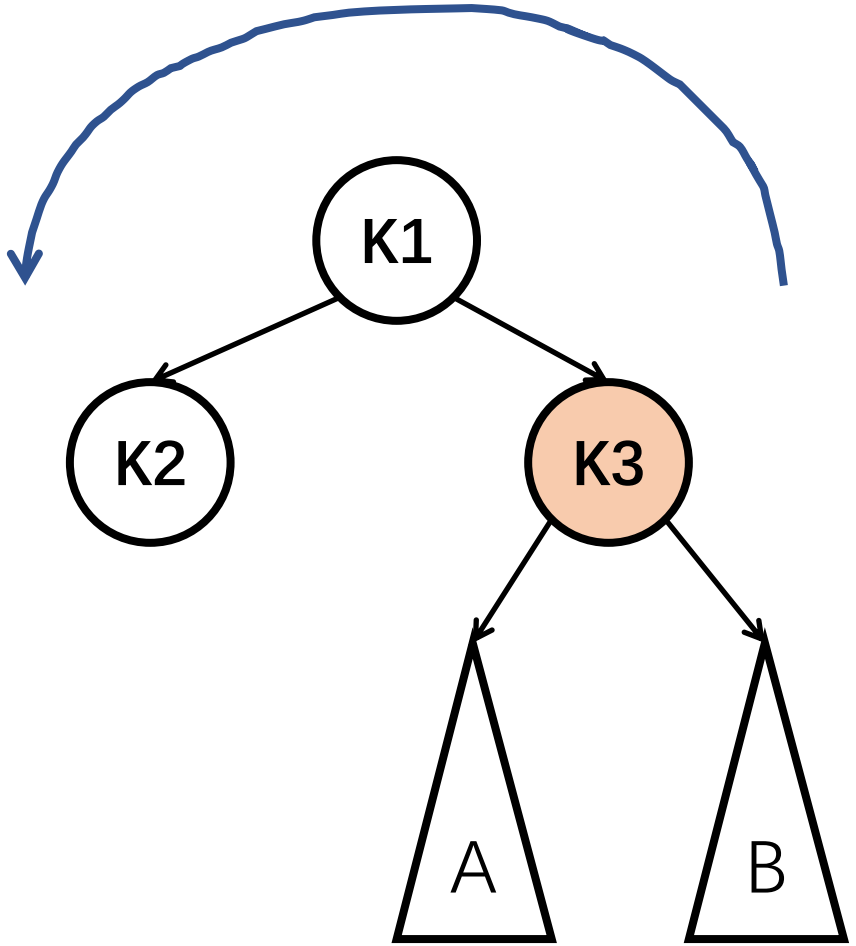
性质：

```
| H(left) - H(right) | <= 1
```

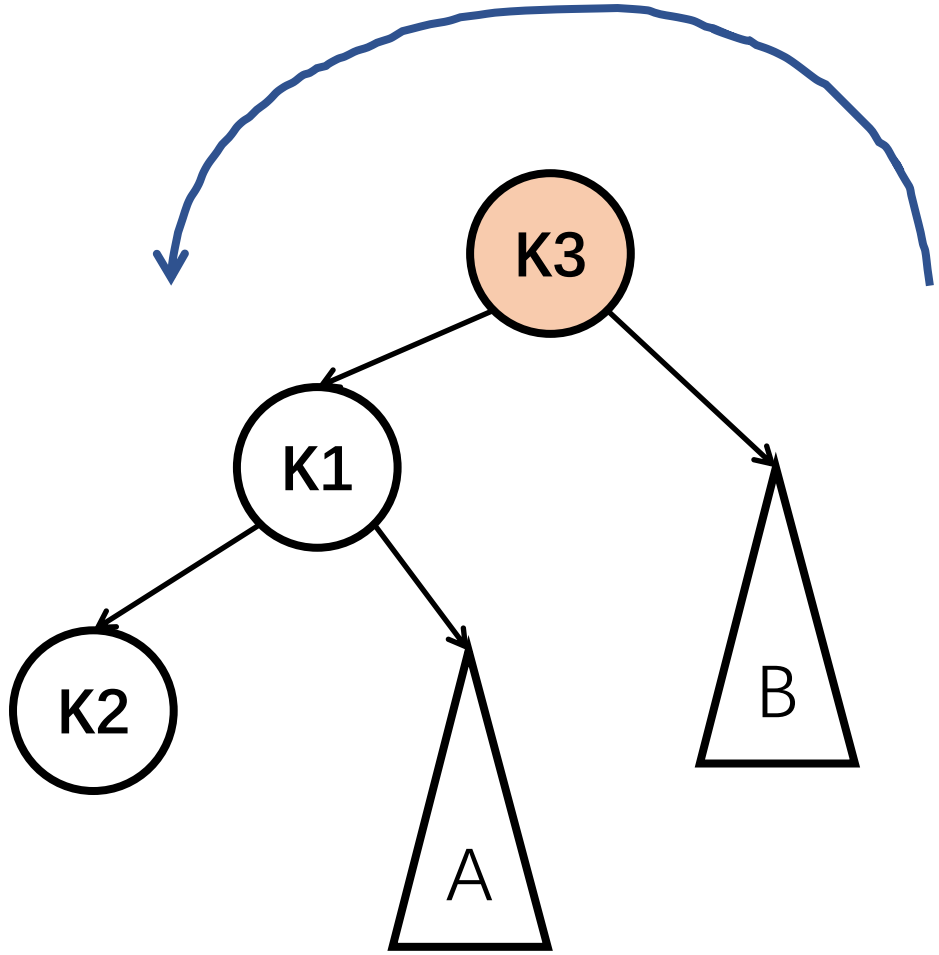
优点：
由于对每个节点的左右子树的树高做了限制，所以整棵树不会退化成一个链表



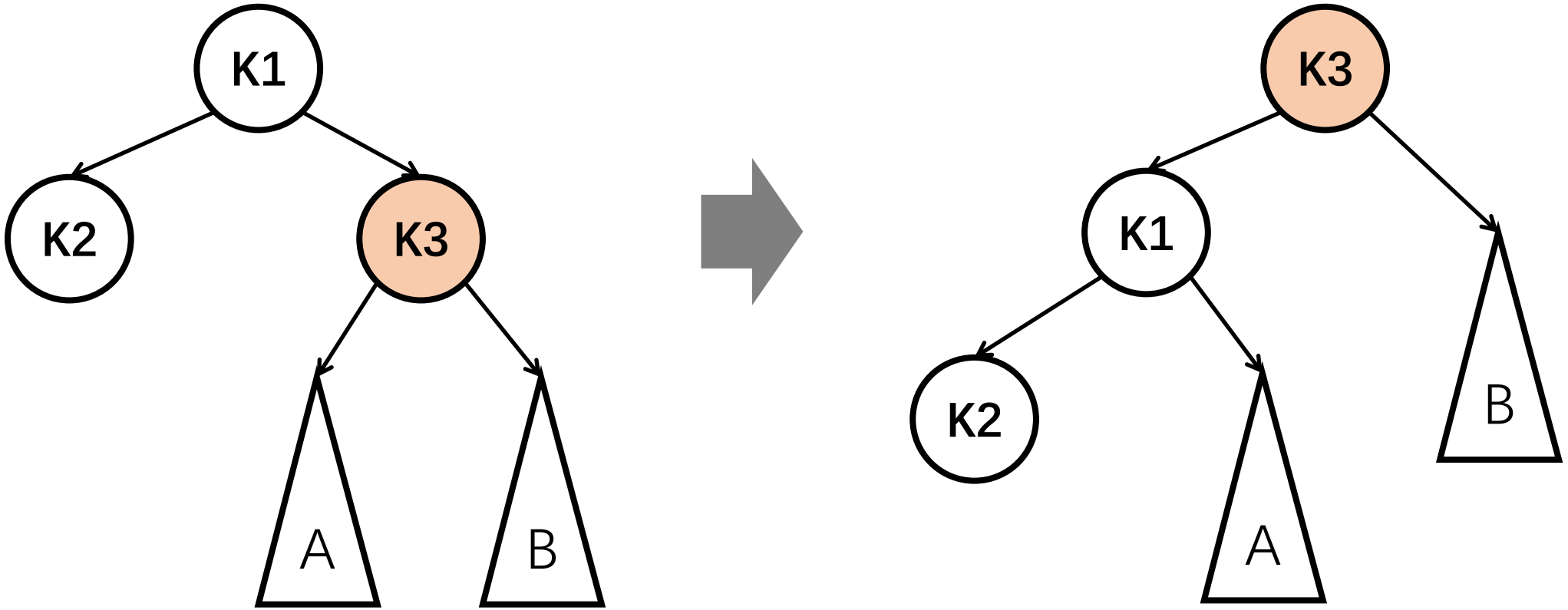
AVL 树-左旋



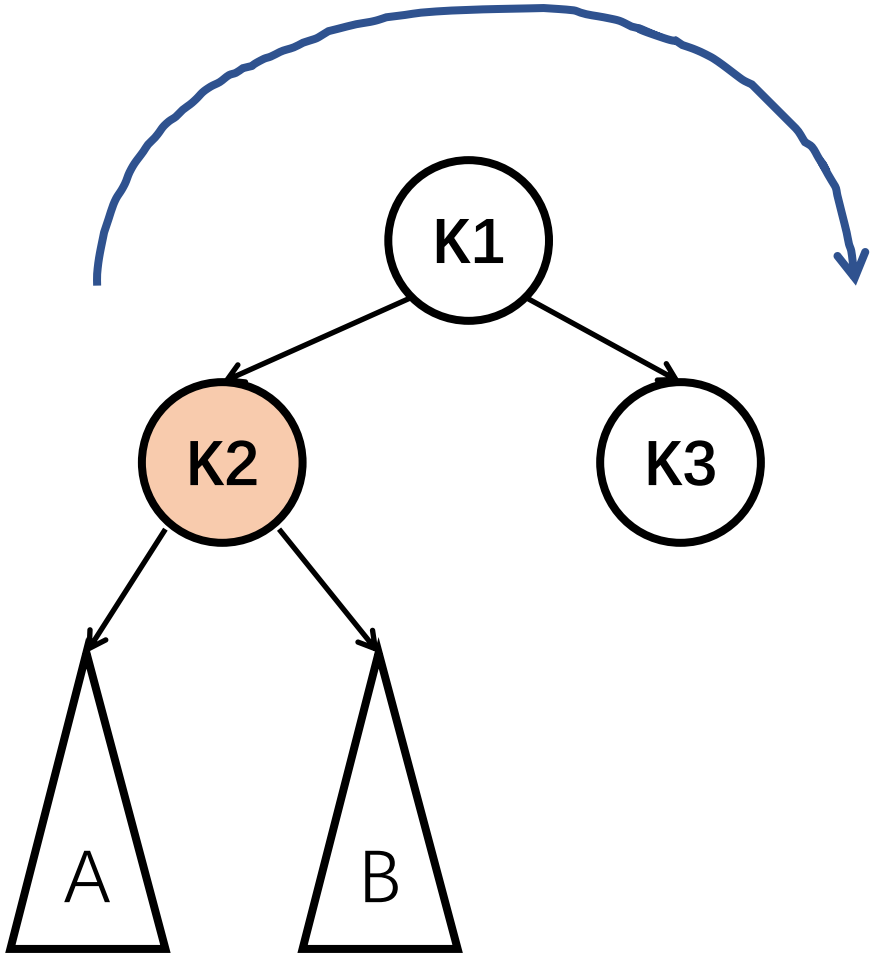
AVL 树-左旋



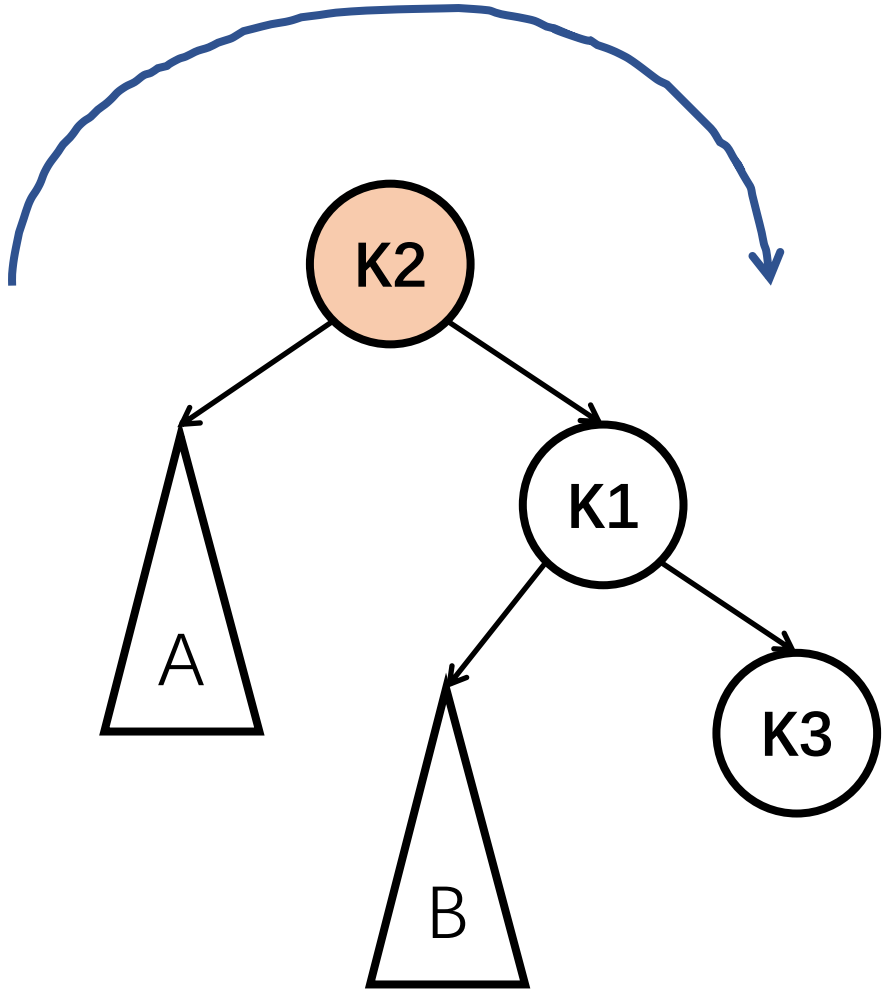
AVL 树-左旋



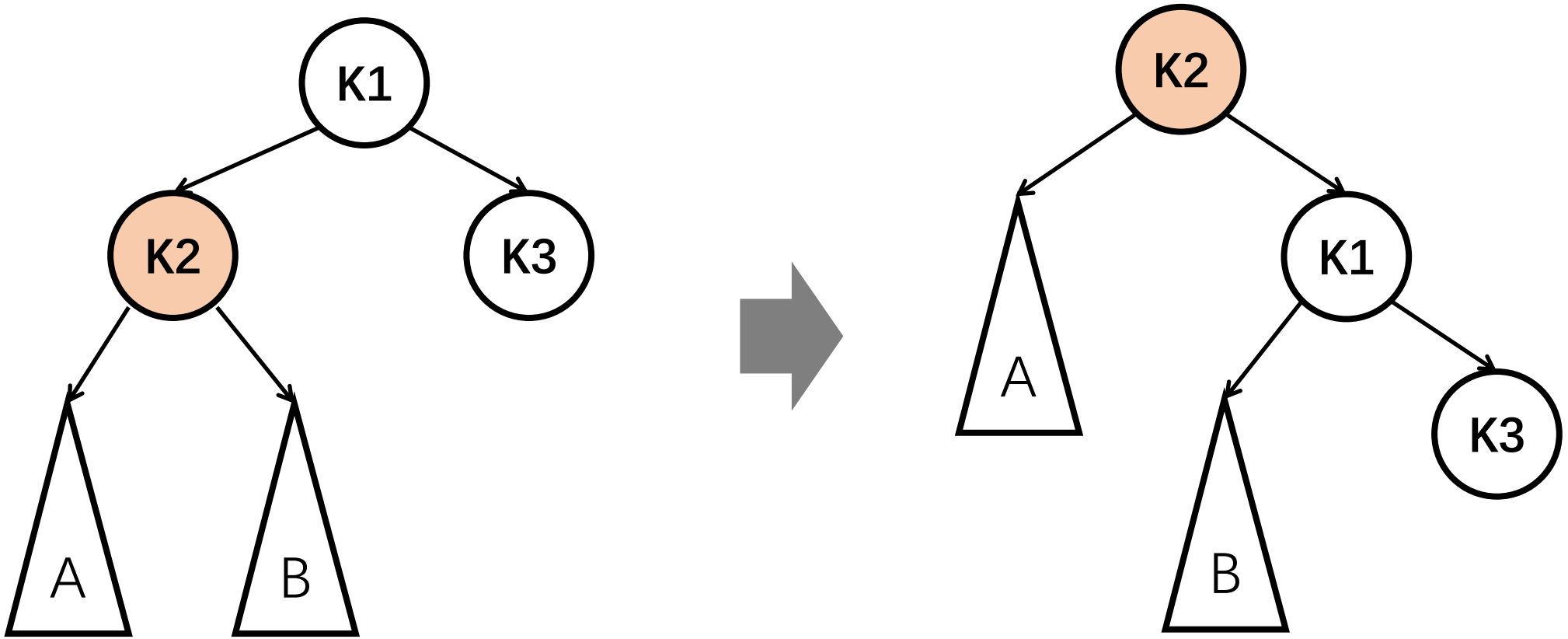
AVL 树-右旋



AVL 树-右旋

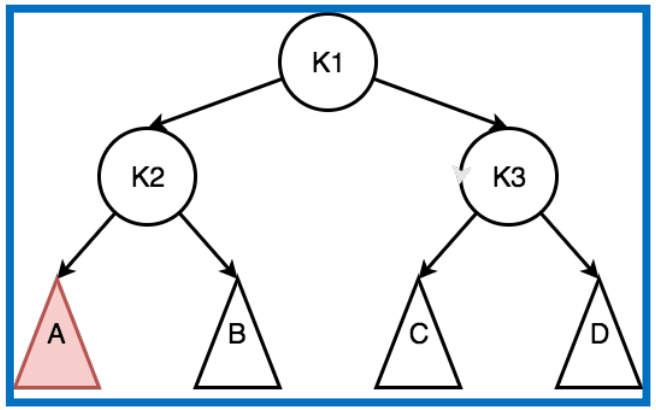


AVL 树-右旋

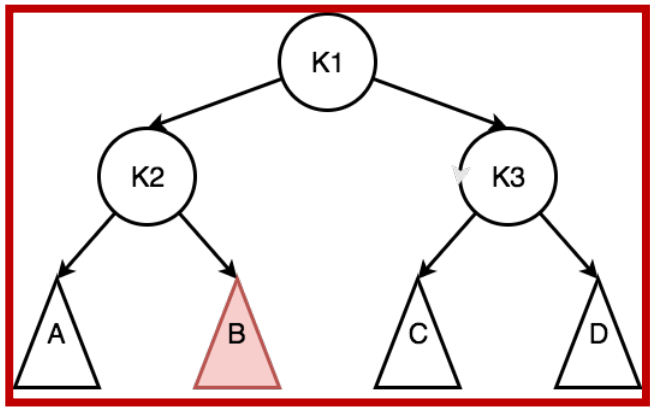


AVL 树-失衡类型

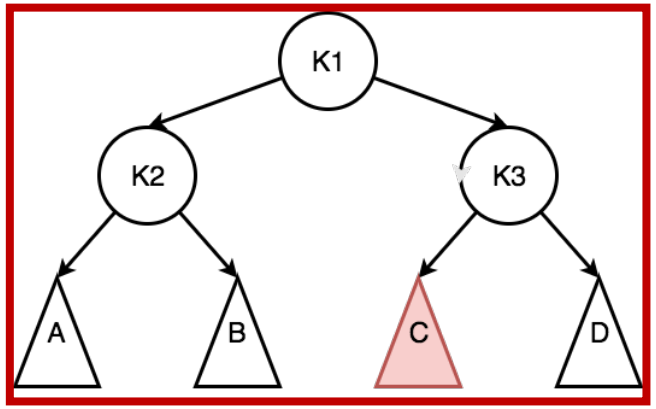
LL 型



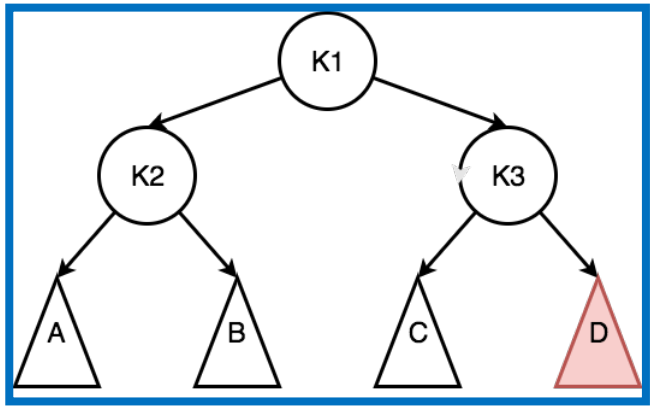
LR 型



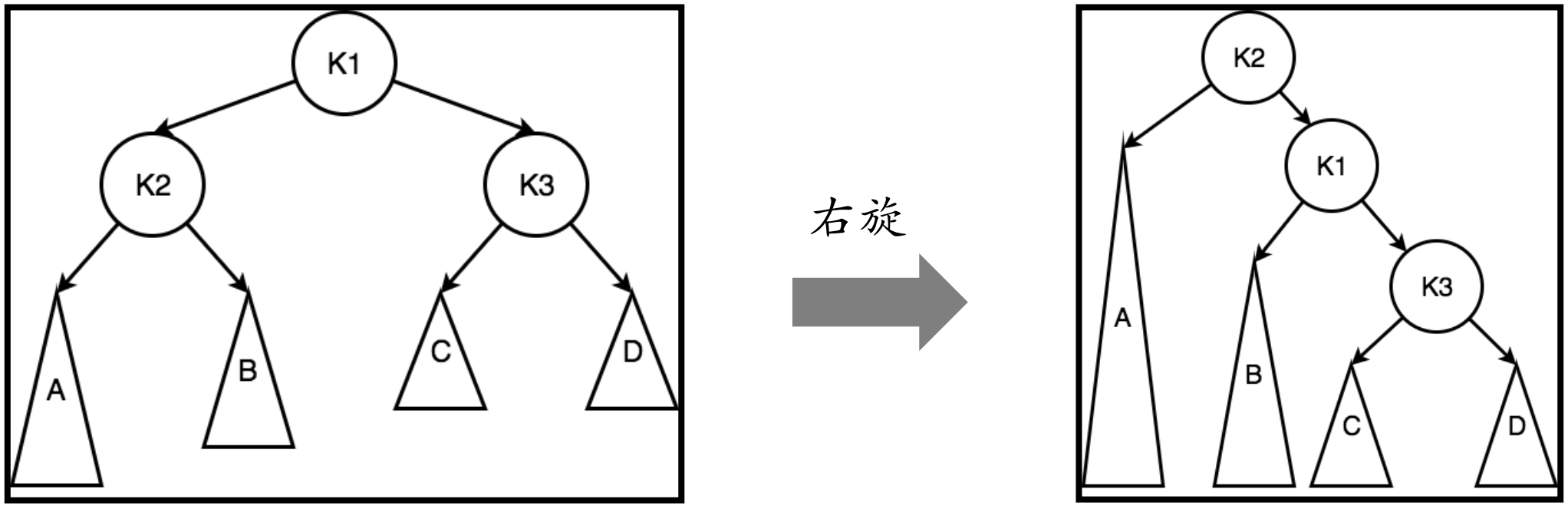
RL 型



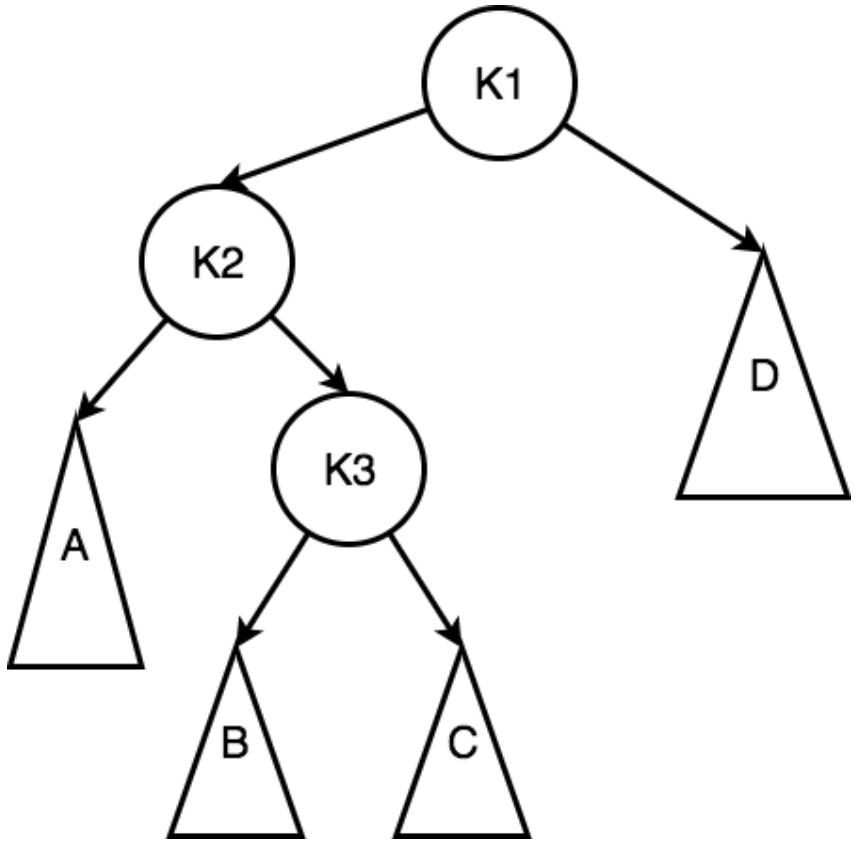
RR 型



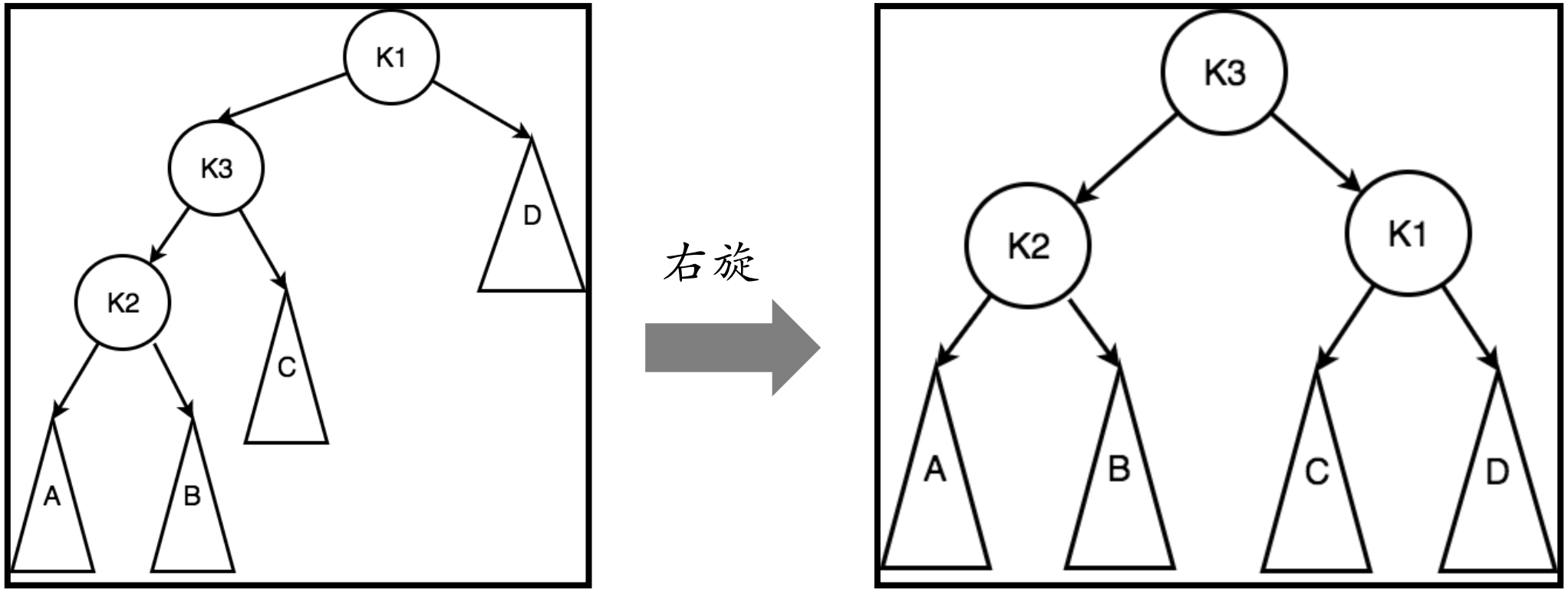
AVL 树-LL 型



AVL 树-LR 型



AVL 树-LR 型-先左旋

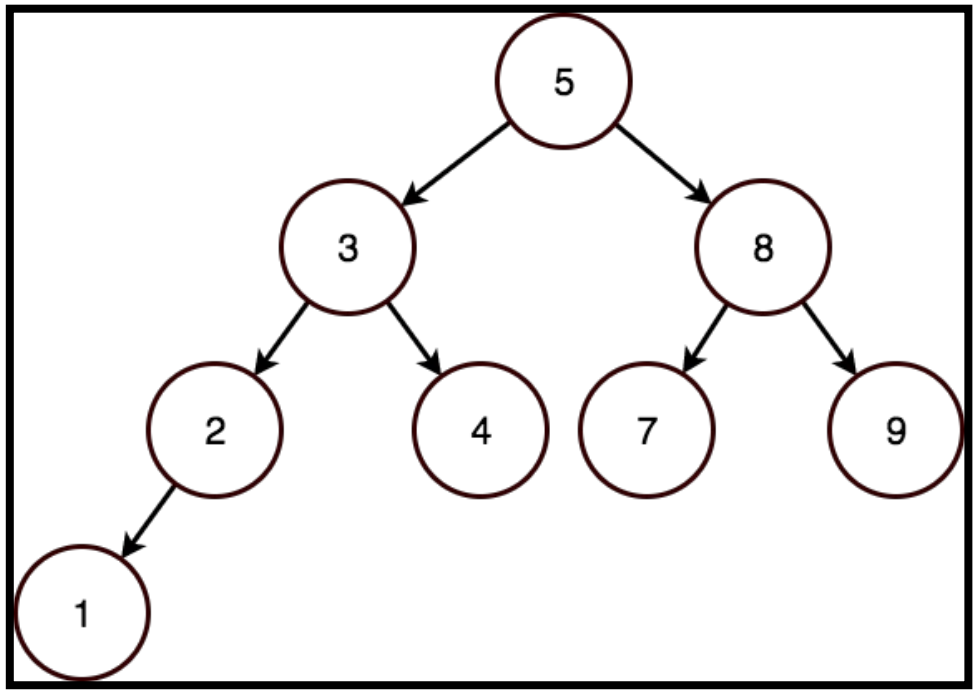


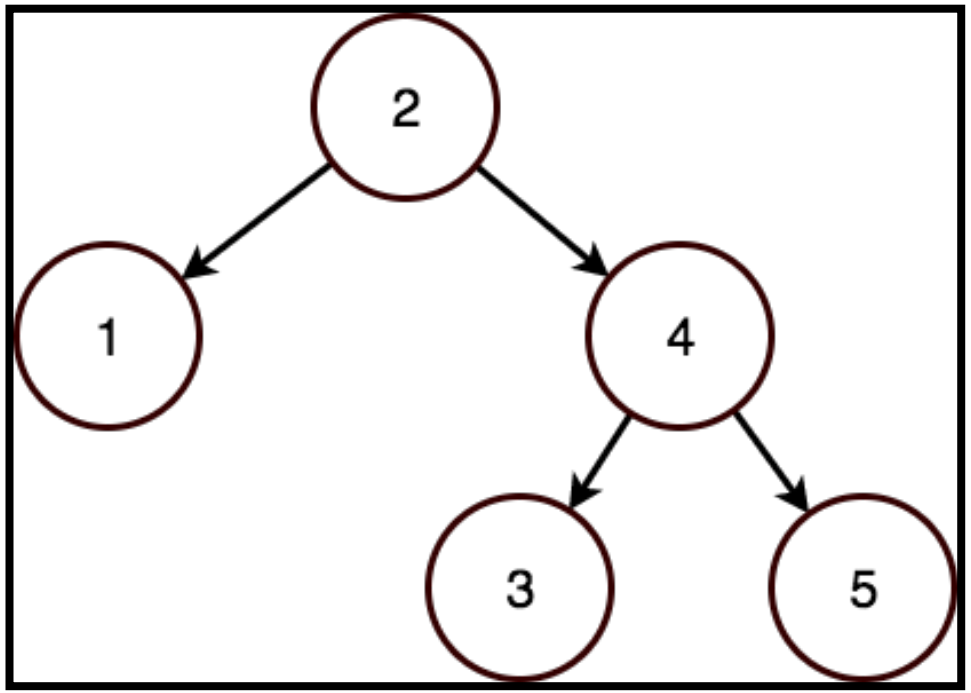
随堂练习-2

请按照如下顺序插入数字，画出对应的 AVL 树

```
1 : [ 5 9 8 3 2 4 1 7 ]
2 : [ 1 2 3 4 5 ]
```

随堂练习-2



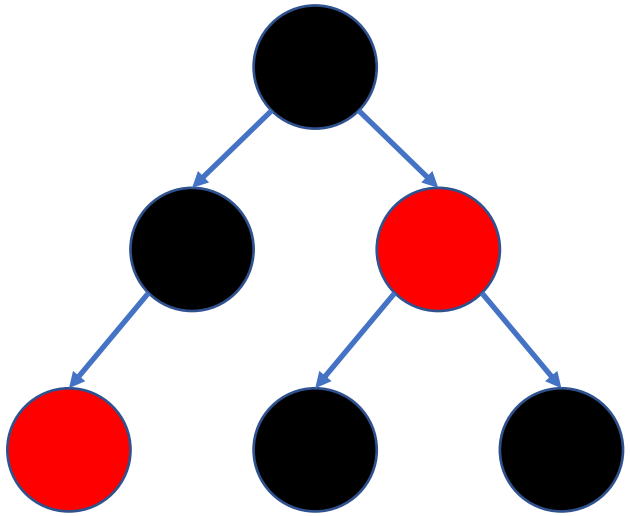



1. vim
vim ⌘1
bash ⌘2
bash ⌘3
39 }
40
41 Node *insert_maintain(Node *root) {
42 if (!hasRedChild(root)) return root;
43 if (root->lchild->color == RED && root->rchild->color == RED) {
44 if (!hasRedChild(root->lchild) && !hasRedChild(root->rchild)) return root;
45 root->color = RED;
46 root->lchild->color = root->rchild->color = BLACK;
47 return root;
48 }
49 if (root->lchild->color == RED) {
50 if (!hasRedChild(root->lchild)) return root;
51
52
53 } else {
54 if (!hasRedChild(root->rchild)) return root;
55
56 }
57
58 }
59
60
61 Node *__insert(Node *root, int key) {
62 if (root == NIL) return getNewNode(key);
<-6班 资 料 /X.现 场 撸 代 码 /15.RBT.cpp [FORMAT=unix] [TYPE=CPP] [POS=54,30][62%] 21/09/19 - 20:21

AVL树：代码演示

三. 红黑树

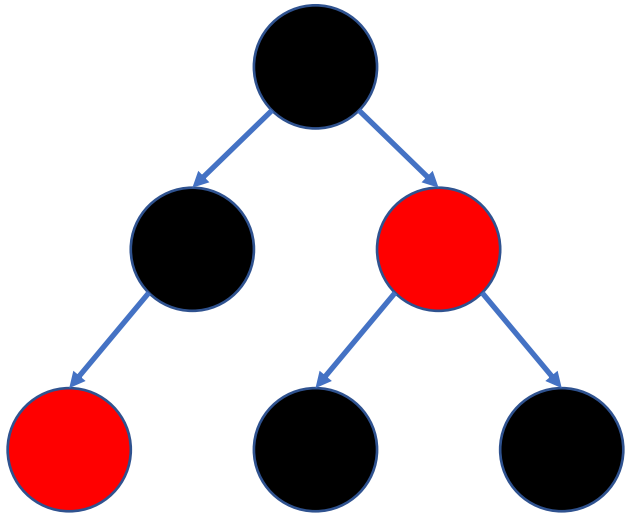
红黑树的平衡条件



1. 每个节点非黑即红
2. 根节点是黑色
3. 叶节点（NIL）是黑色
4. 如果一个节点是红色，则它的两个子节点都是黑色的
5. 从根节点出发到所有叶节点路径上，黑色节点数量相同

红黑树的平衡条件

问题1：红黑树中，最长路径和最短路径长度的关系？



红黑树的平衡条件

问题1：红黑树中，组长路径和最短路径长度的关系？

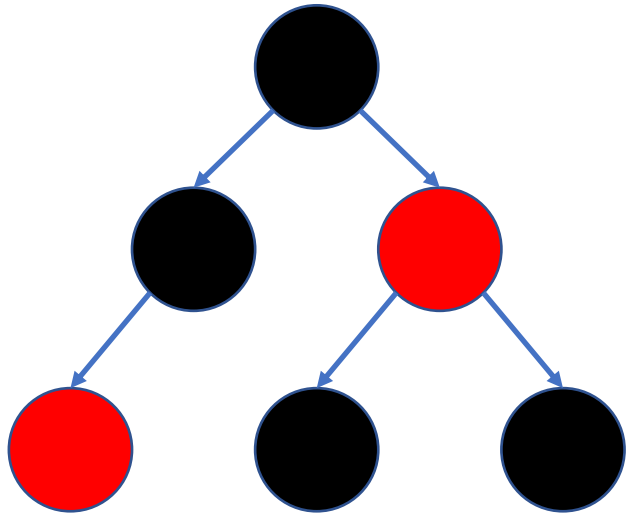
答：

根据平衡条件第4、5两点

最短路径，都是黑色

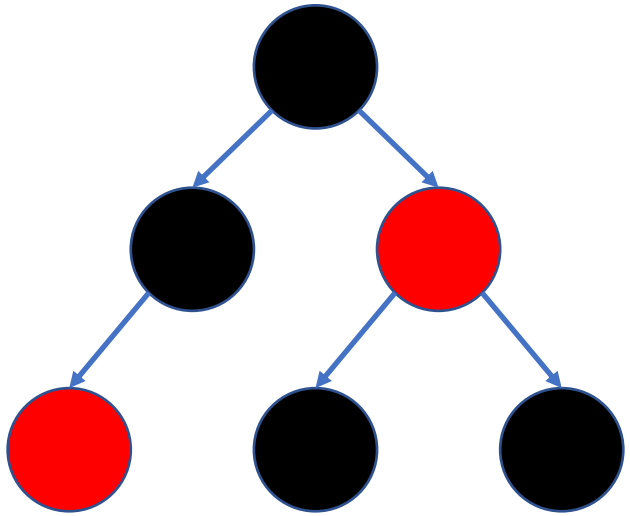
最长路径，红黑相间

最长是最短的两倍



红黑树的平衡条件

问题2：怎么理解条件 3 中的 NIL 节点



红黑树的平衡条件

问题2：怎么理解条件 3 中的 NIL 节点

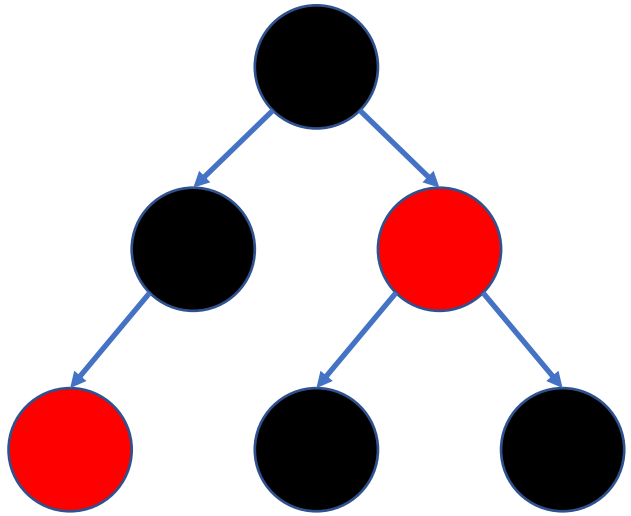
答：

就像文章中的标点符号，

虽然它不属于内容的部分，

平时你也不会注意他，

可要是真没有，就会很麻烦。

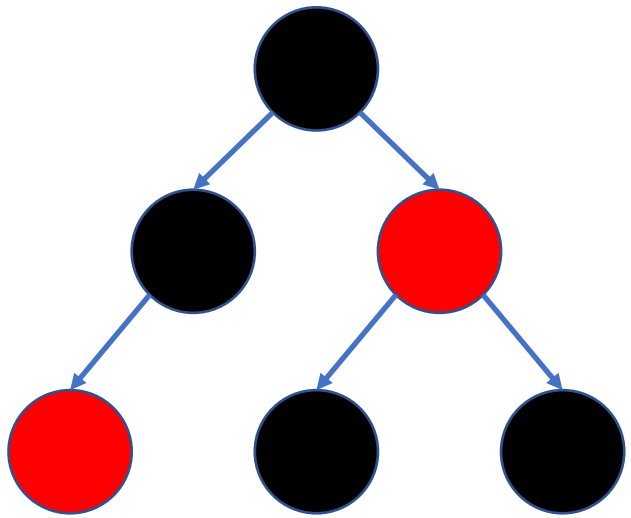


平衡调整终极法门

插入调整站在 祖父节点 看

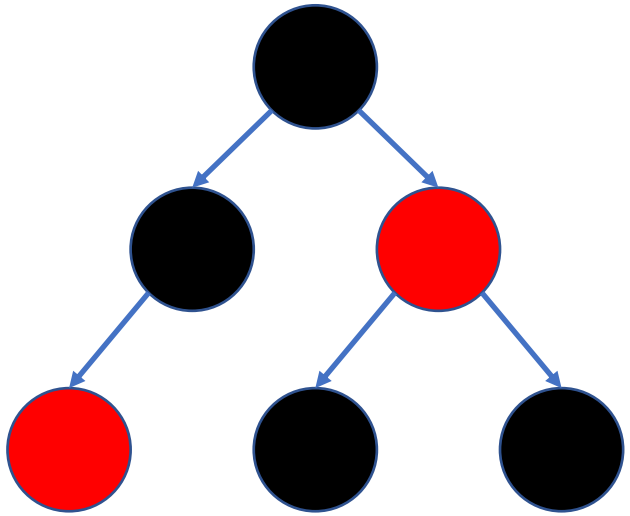
删除调整站在 父节点 看

插入和删除的情况处理一共五种



插入调整的发生场景

问题3：新插入的节点是什么颜色的？红色，黑色？



插入调整的发生场景

问题3：新插入的节点是什么颜色的？红色，黑色？

答：

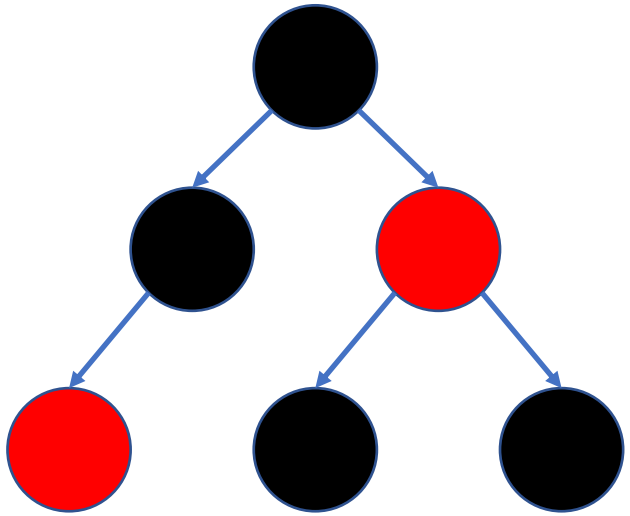
红色，因为

插入黑色一定引发失衡，

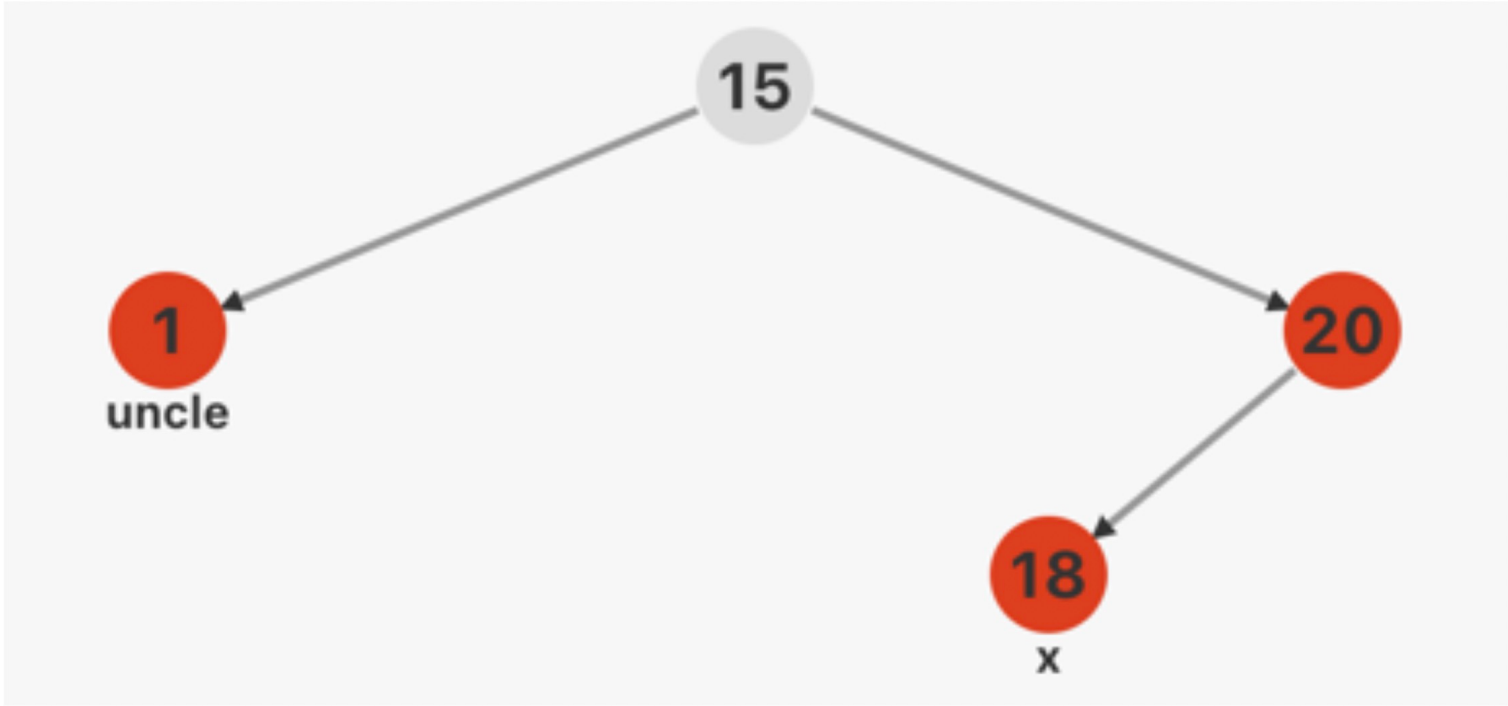
插入红色不一定引发失衡，

一个必死，一个有概率活，

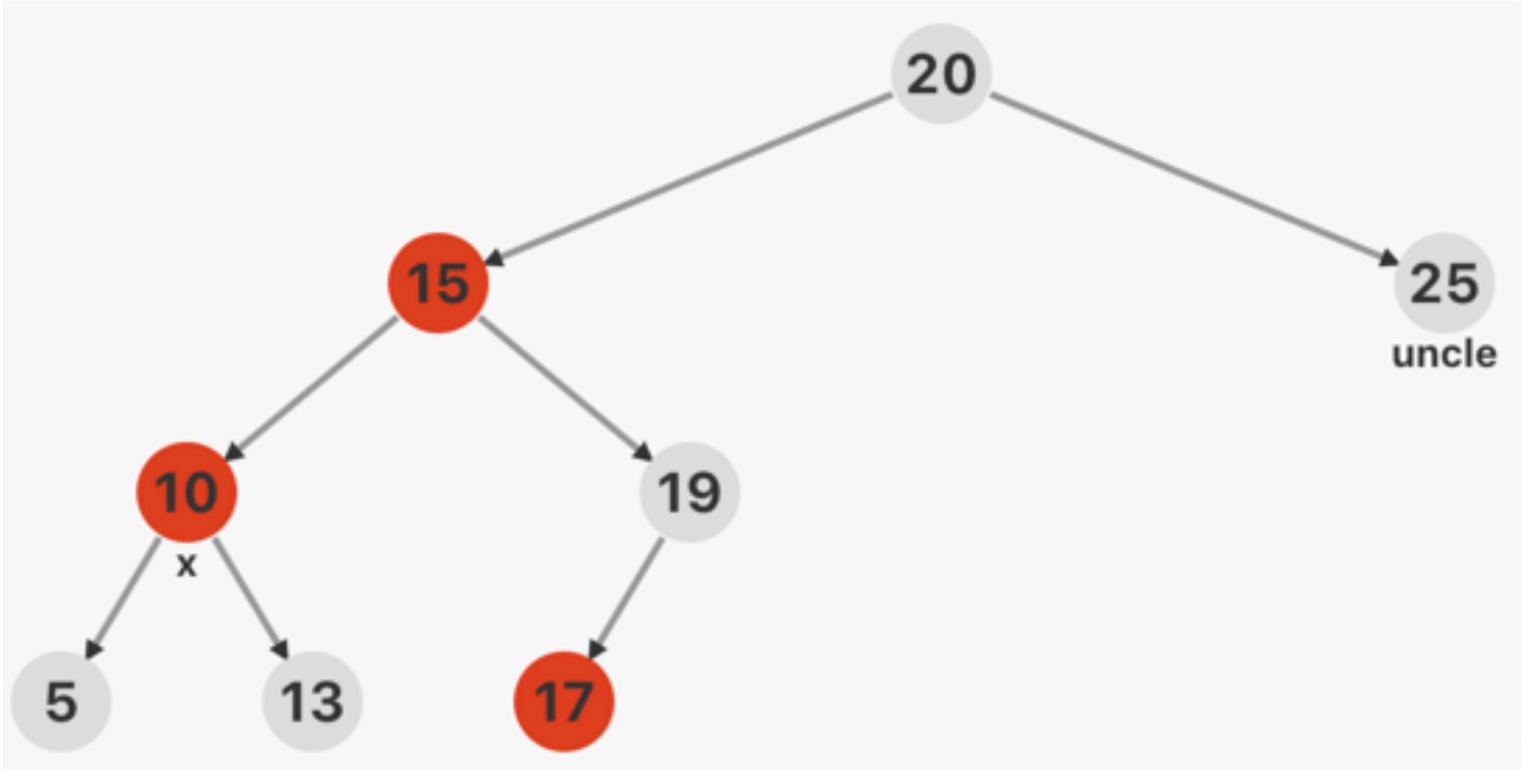
正常人，都会选后者。



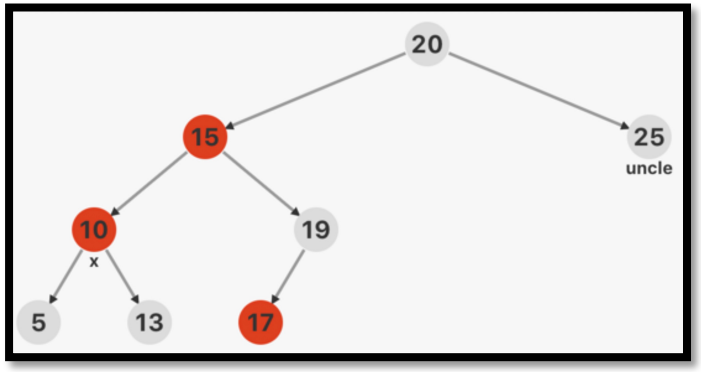
插入调整情况一

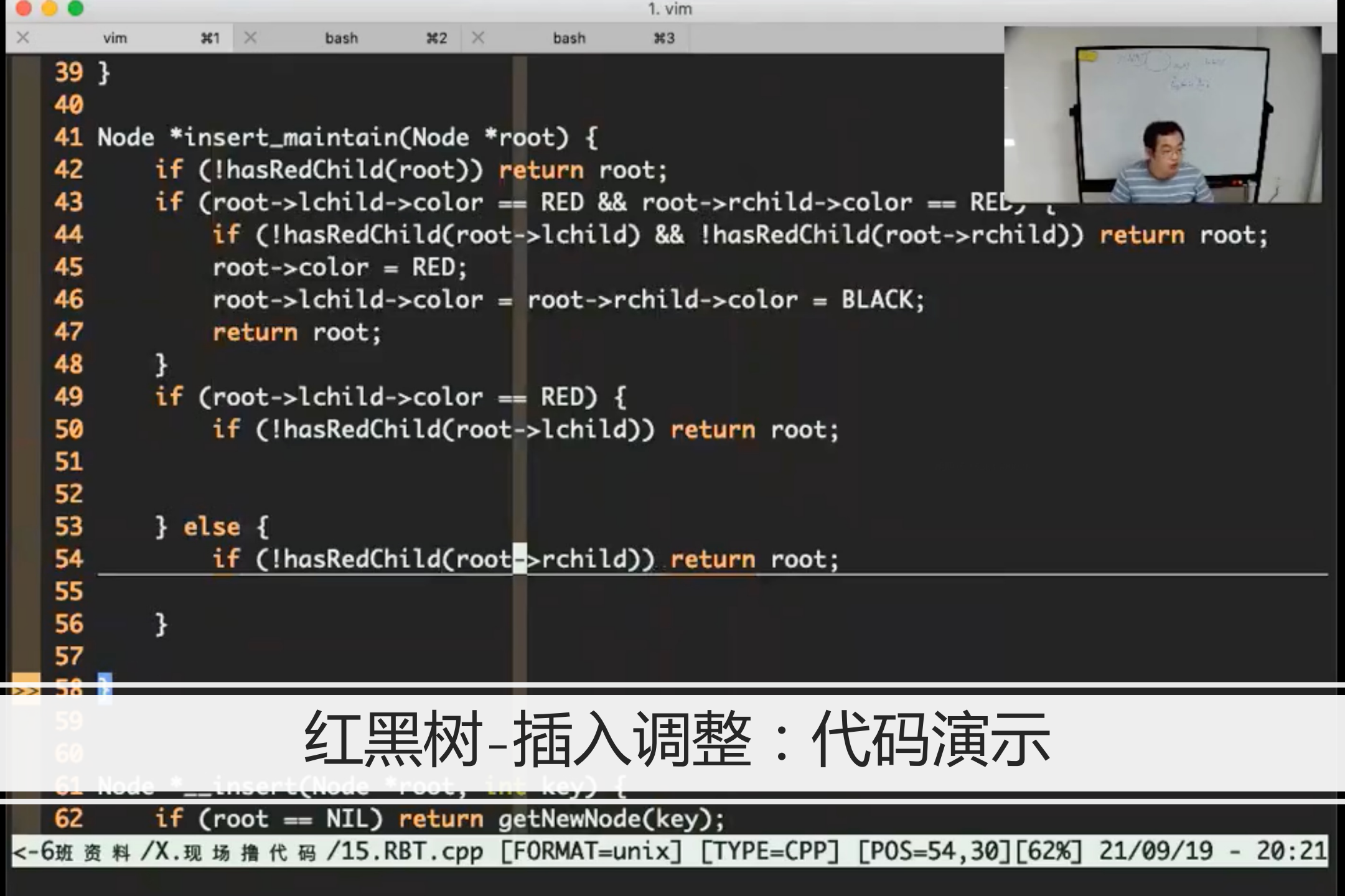


插入调整
情况二



插入调整情况二

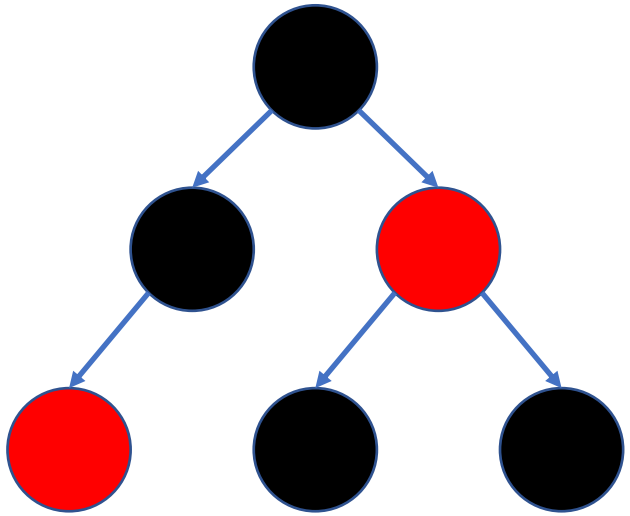




红黑树-插入调整：代码演示

删除调整的发生场景

问题4：删除什么样的节点，会引发红黑树的失衡?



删除调整的发生场景

问题4：删除什么样的节点，会引发红黑树的失衡?

答：

删除度为 0 的黑色节点的时候，

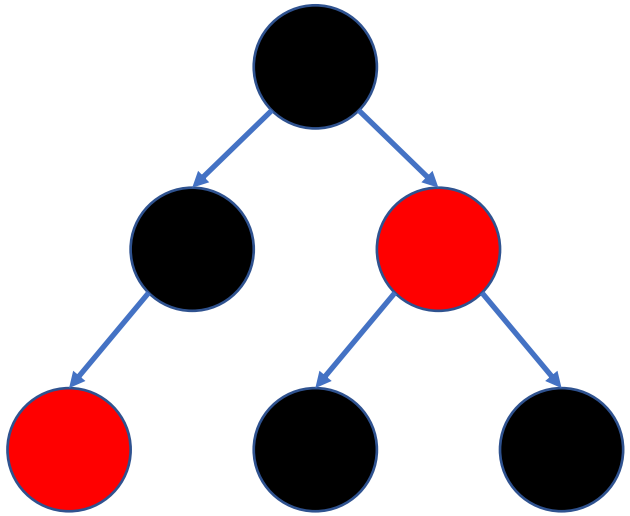
会引发红黑树的失衡。

无处安放的 1 个黑，导致

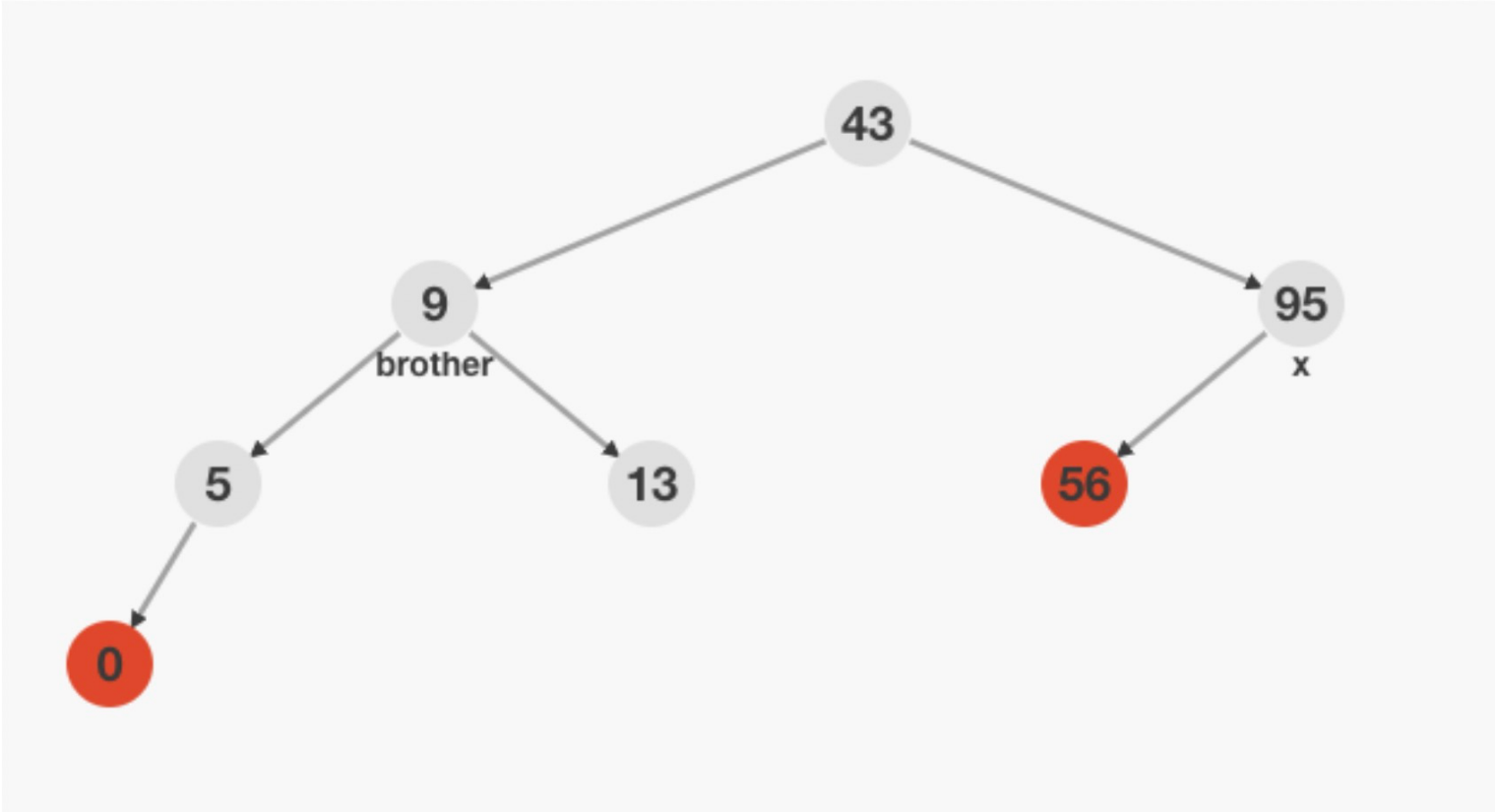
NIL 成了背锅侠。

从此以后， NIL 彻底黑化。

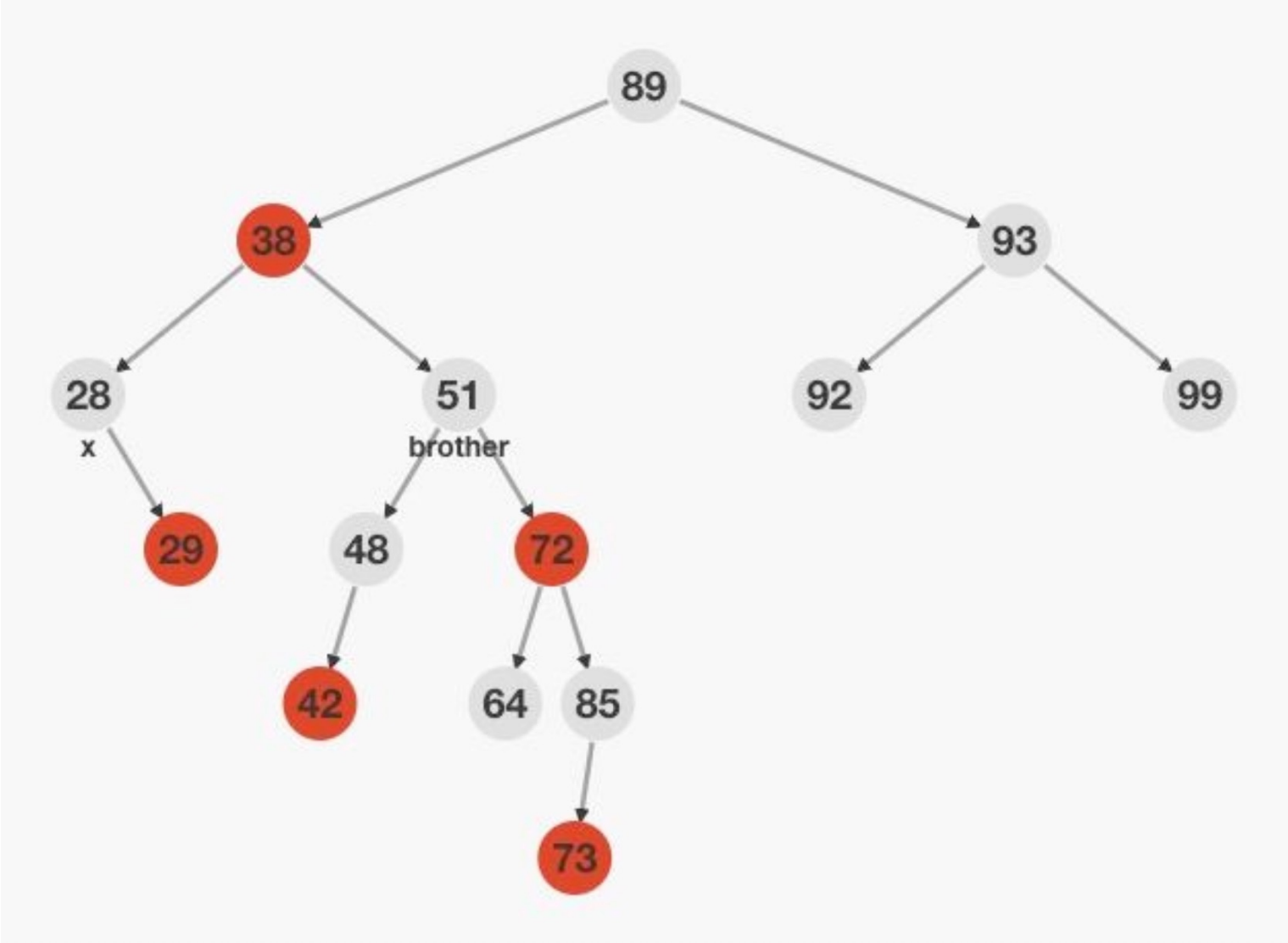
这就是【双重黑】的诞生过程。



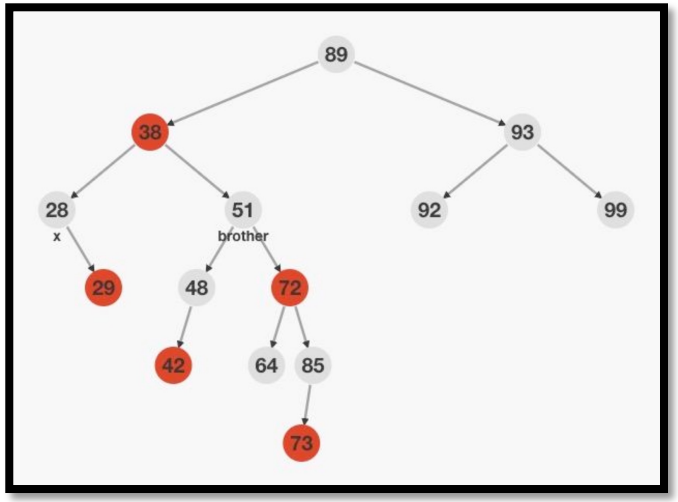
删除调整情况一



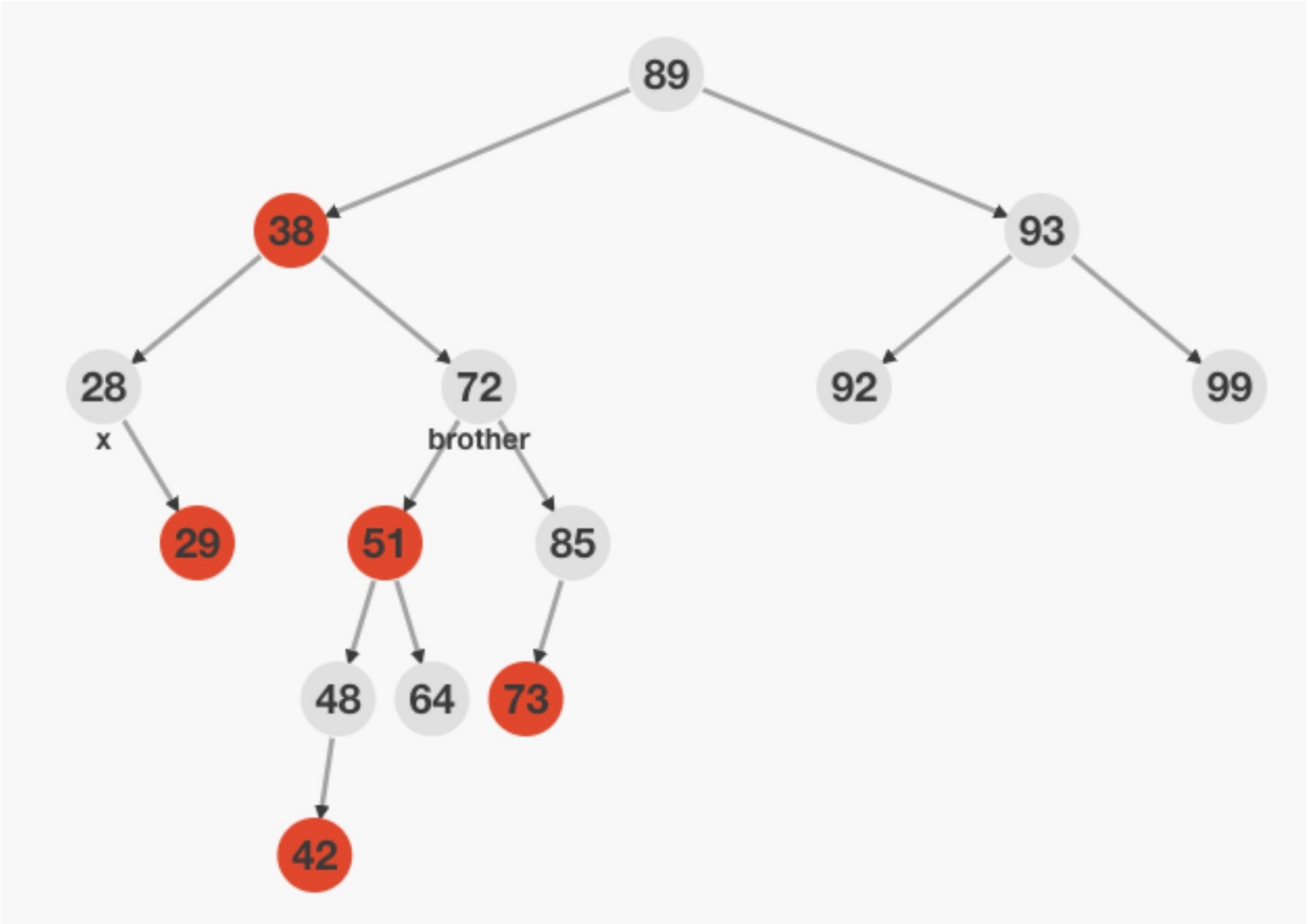
删除调整情况二



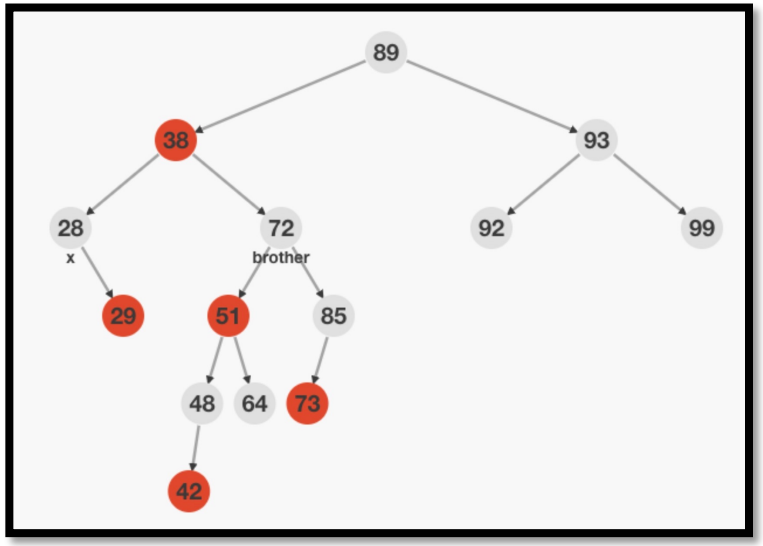
删除调整情况二



删除调整
情况三



删除调整
情况三



红黑树-删除调整：代码演示

四. B-树

B 树 – 结构定义

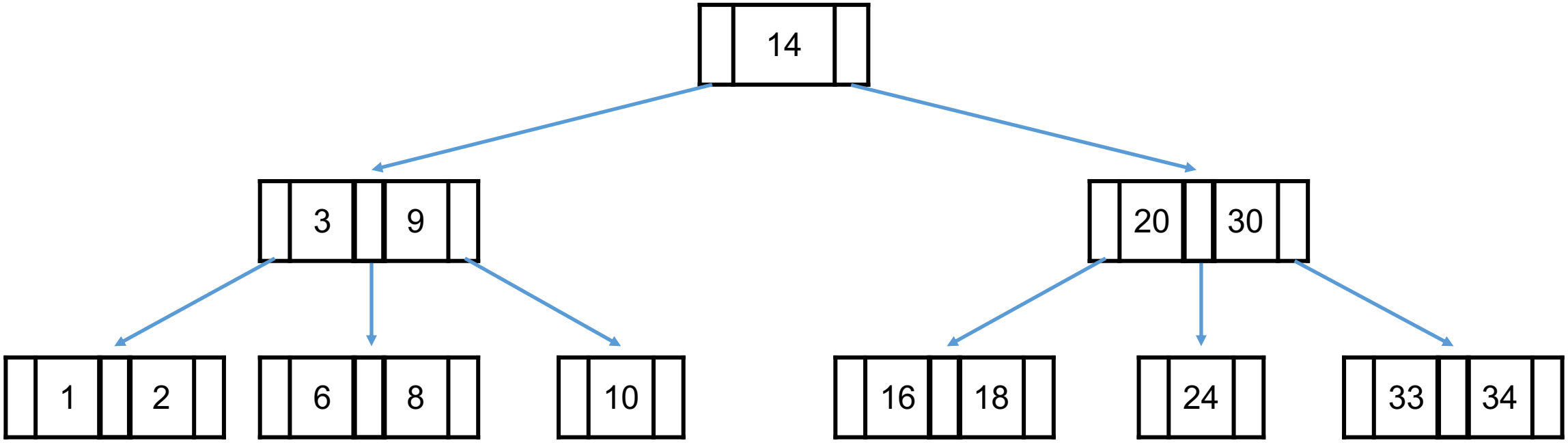
一棵 m 阶 B 树，需要满足下列特性：

1. 树中每个节点，最多含有 m 棵子树
2. 若根节点不是叶子节点，则至少有 2 棵子树
3. 除根结点之外的所有非终端结点至少有[m/2]棵子树
4. 如果一个结点有n-1个关键字，则该结点有n个分支，且这n-1个关键字按照递增顺序排列
5. 每个结点的结构为：（ n，A_0，K_1，A_1，K_2，A_2，… ，K_n，A_n ）
6. 非根结点中关键字的个数 n，满足：[m/2]-1 ≤ n ≤ m-1
7. 所有叶子节点处在同一层

B 树 – 结构定义

m 阶 B 树的性质解读：

1. B 树中只有根节点没办法满足拥有至少 [m/2] 棵子树的条件，其他节点均能满足
2. B 树是一种高度平衡的树形结构，比 AVL 树结构上更优美



1. 元素插入

2. 插入调整

3. 元素删除

4. 删除调整

1. 元素插入

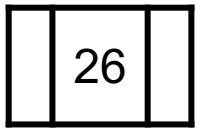
2. 插入调整

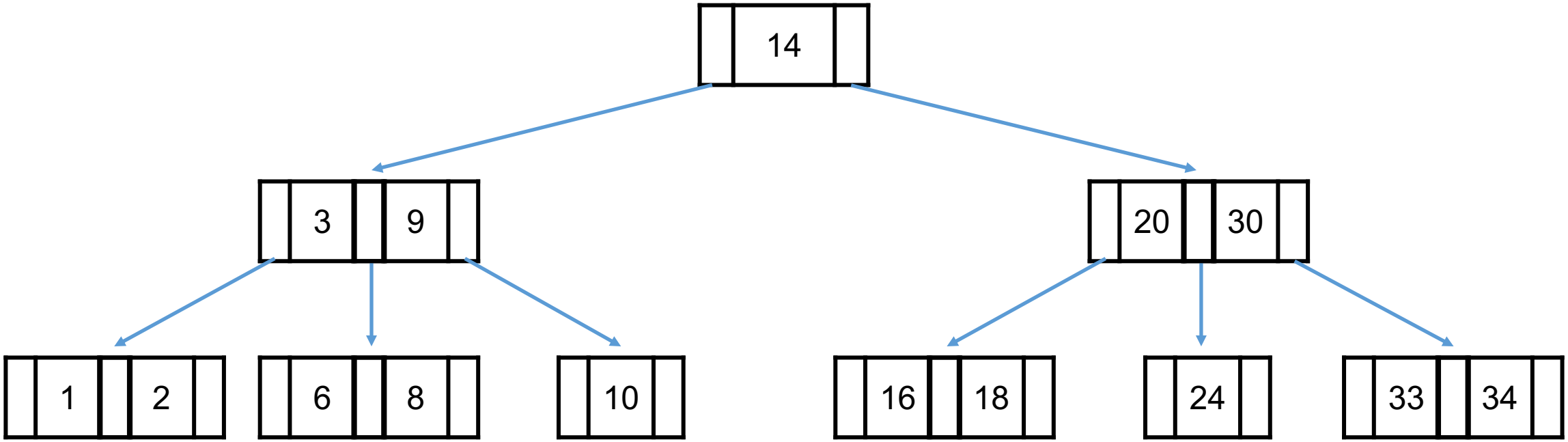
3. 元素删除

4. 删除调整

B 树 – 元素插入

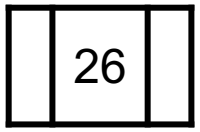
插入：

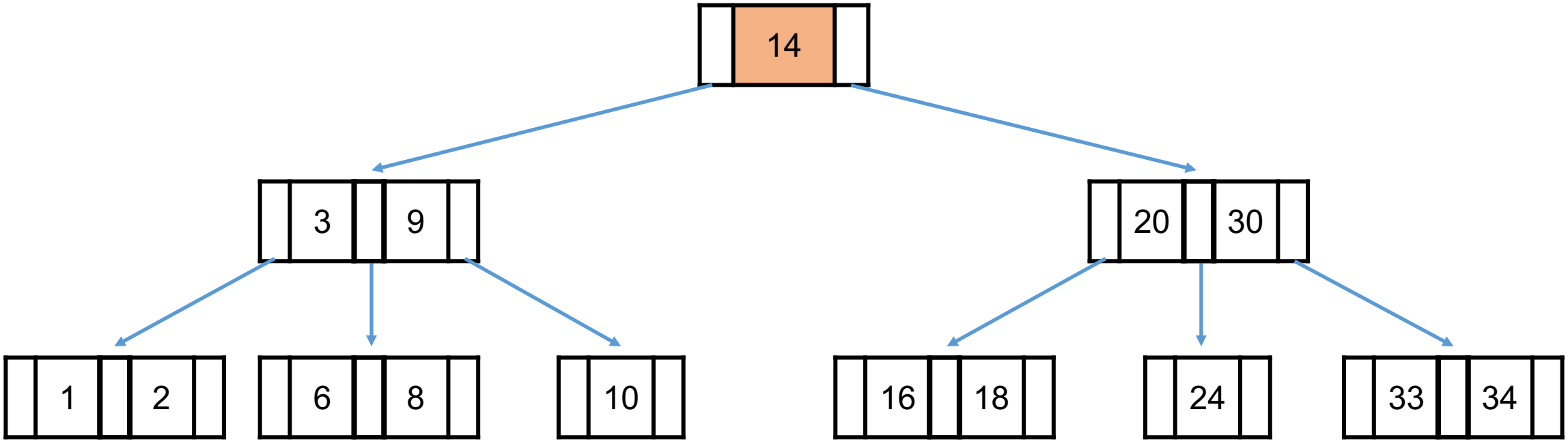




B 树 – 元素插入

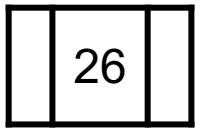
插入：

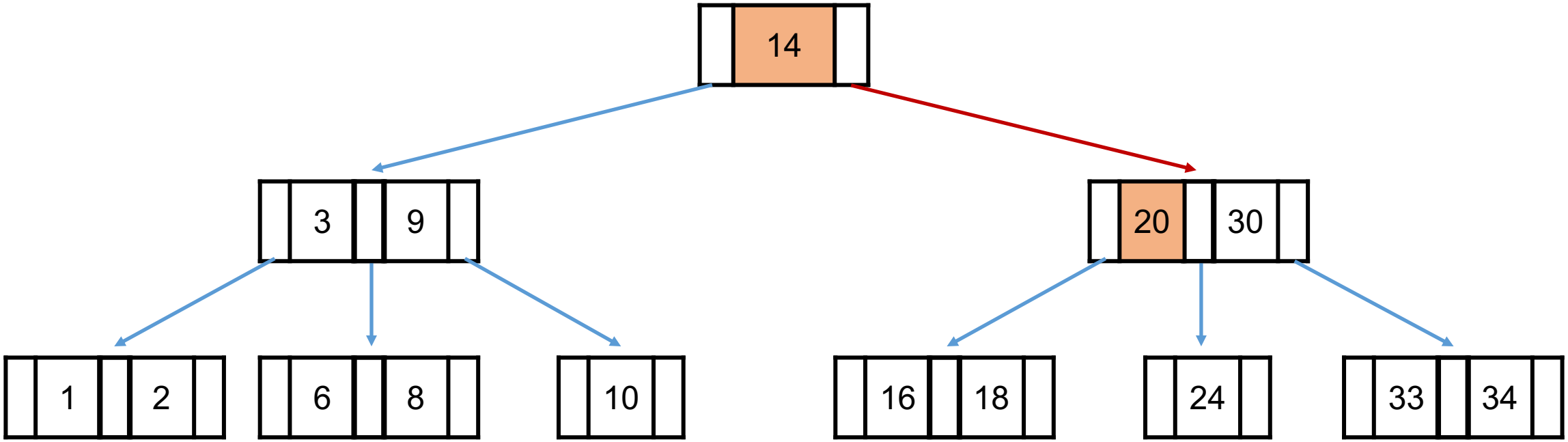




B 树 – 元素插入

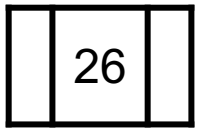
插入：

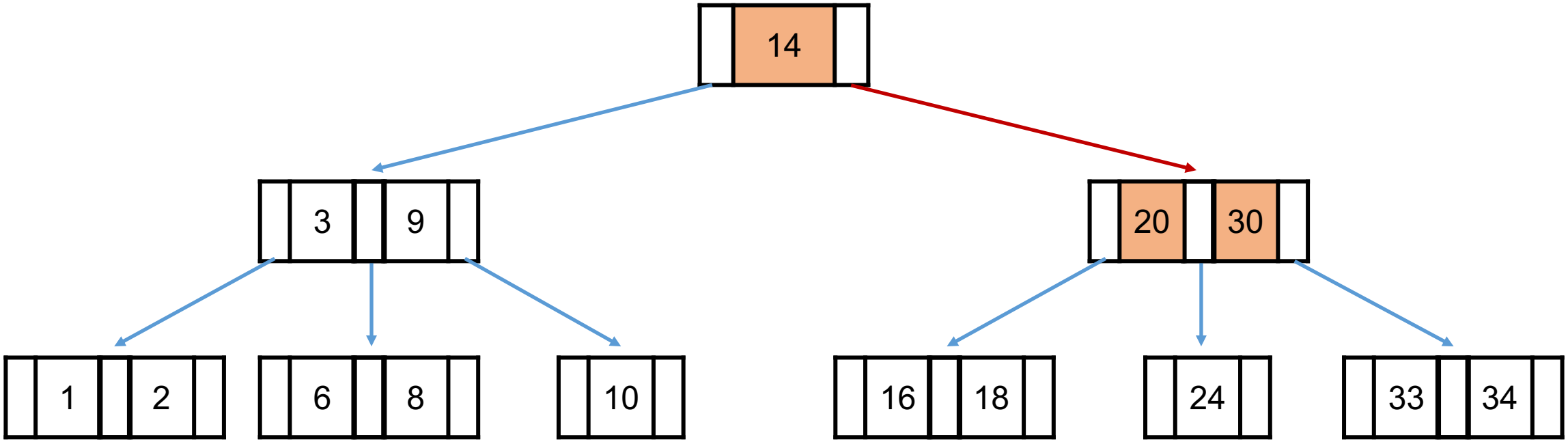




B 树 – 元素插入

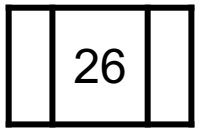
插入：

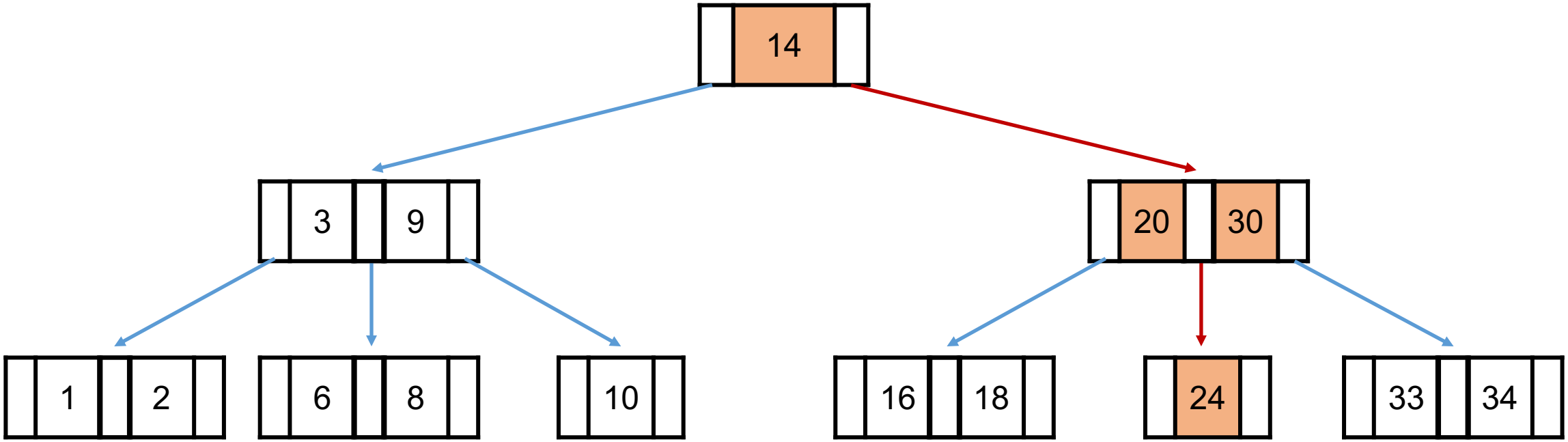




B 树 – 元素插入

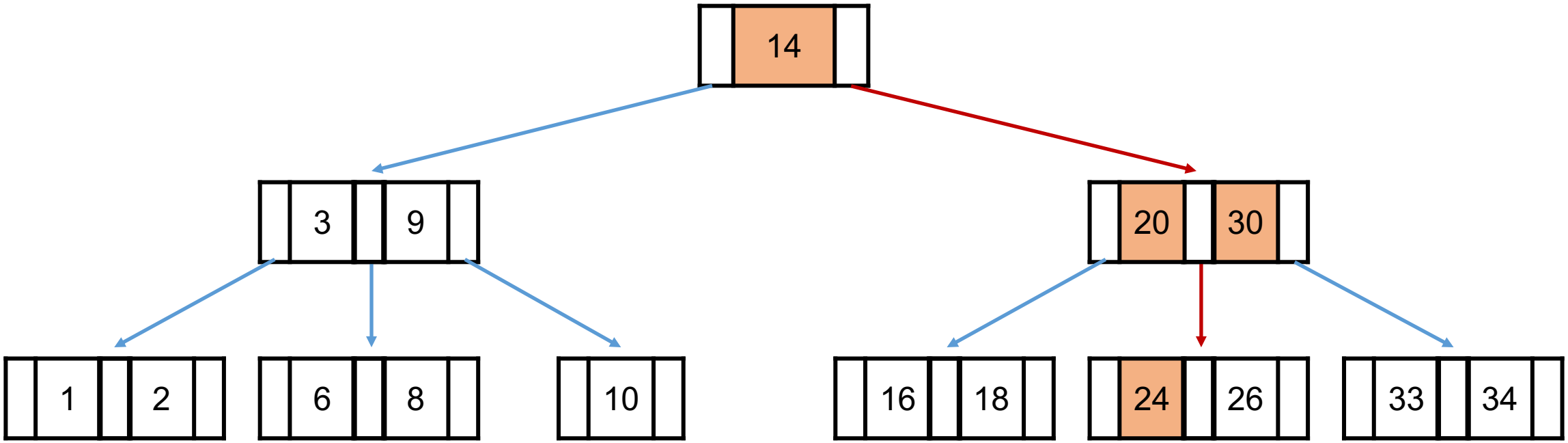
插入：



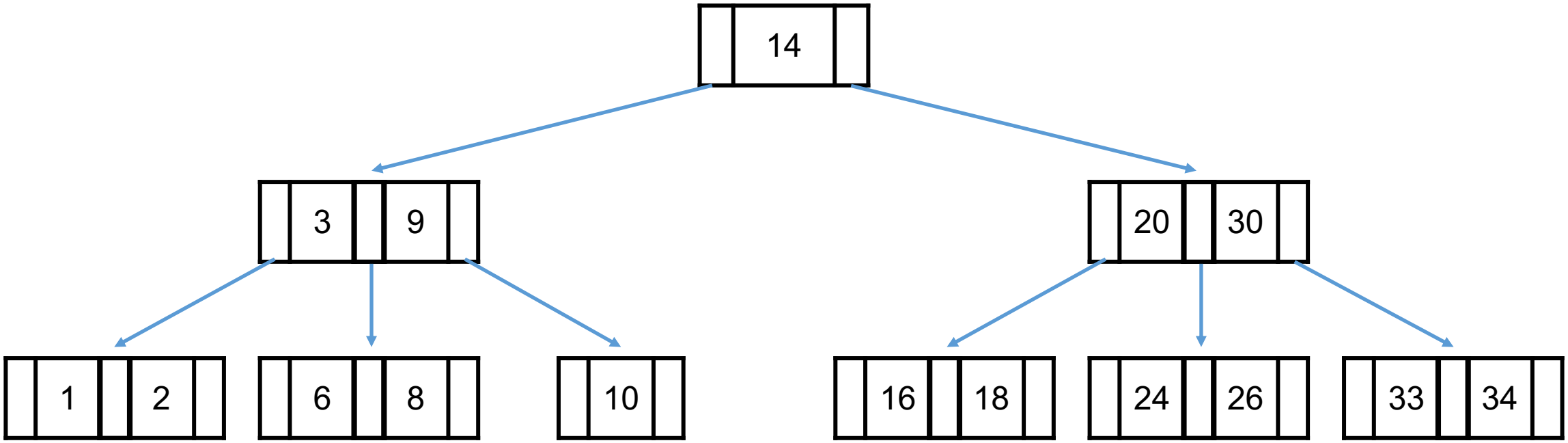


B 树 – 元素插入

插入：



B 树 – 元素插入



~~1. 元素插入~~

2. 插入调整

3. 元素删除

4. 删除调整

~~1. 元素插入~~

2. 插入调整

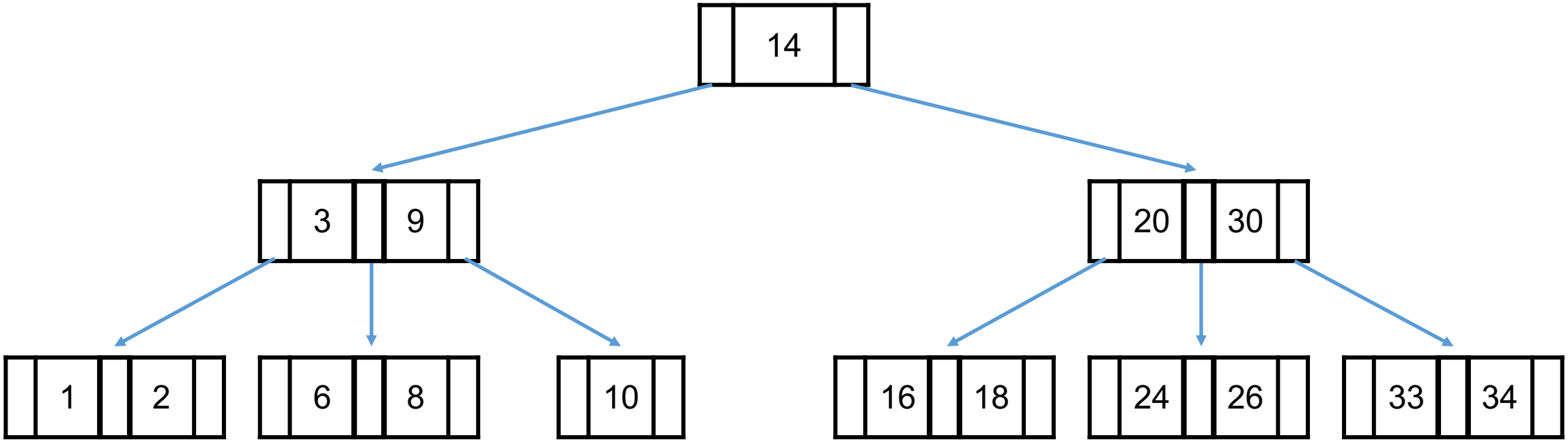
3. 元素删除

4. 删除调整

B 树 – 插入调整（上溢）

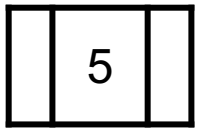
m 阶 B 树的插入调整:

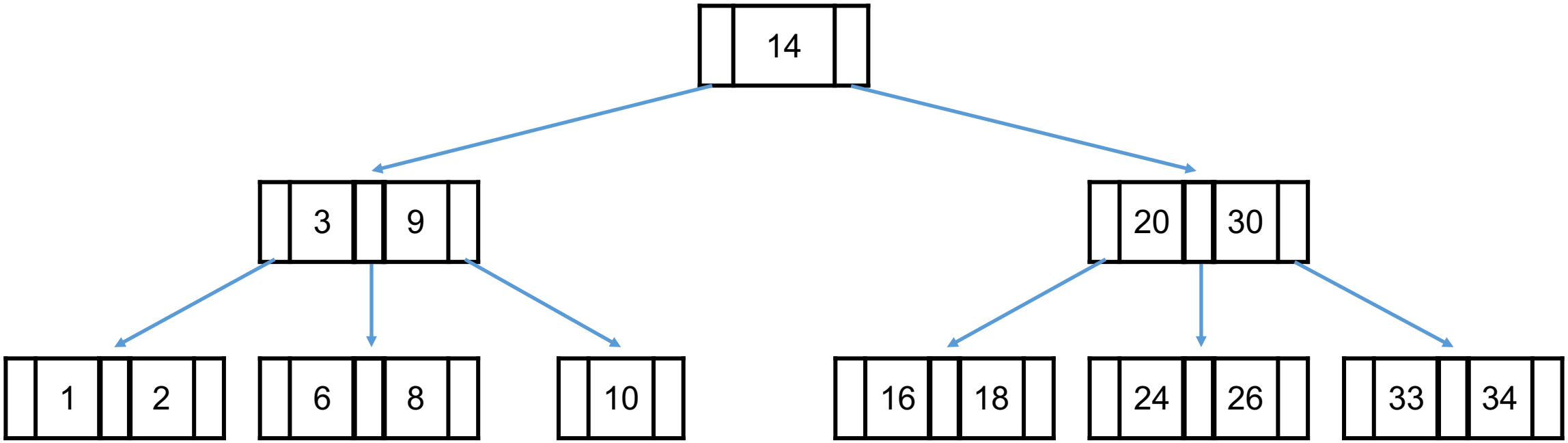
1. 插入调整站在父节点处理，发生在节点关键字数量达到 m 时
2. 插入调整的核心操作是: 节点分裂



B 树 – 插入调整（上溢）

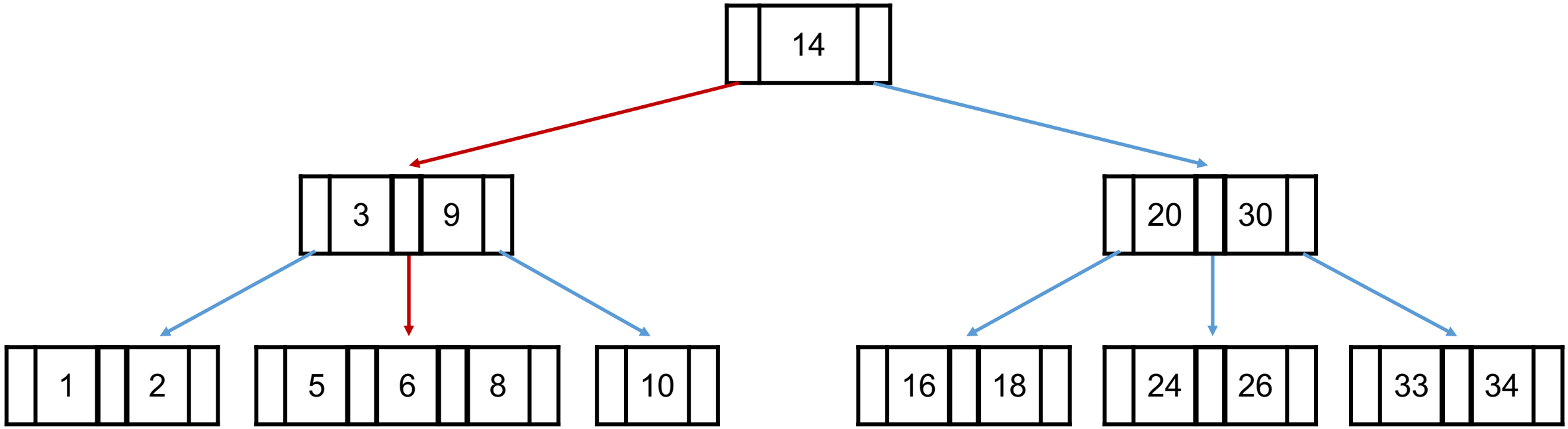
插入：





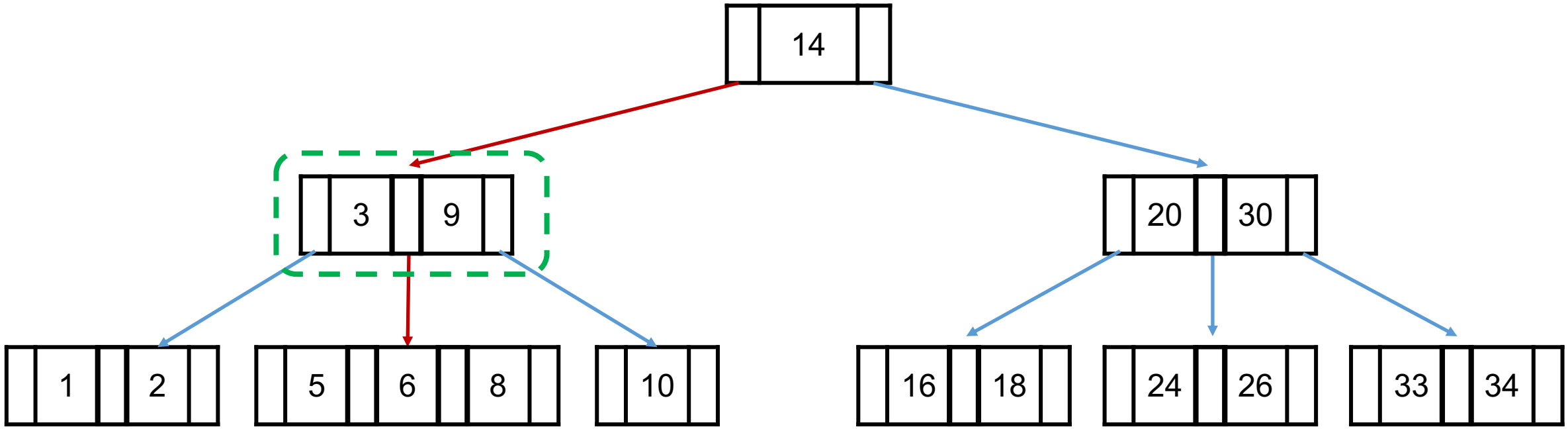
B 树 – 插入调整（上溢）

插入：



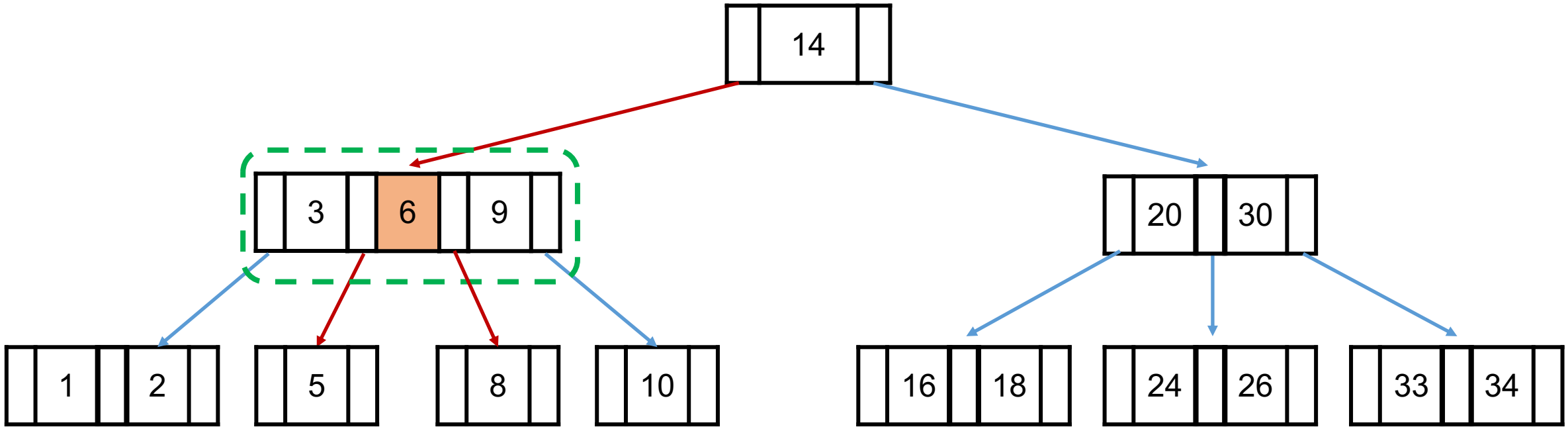
B 树 – 插入调整（上溢）

站在父节点处理



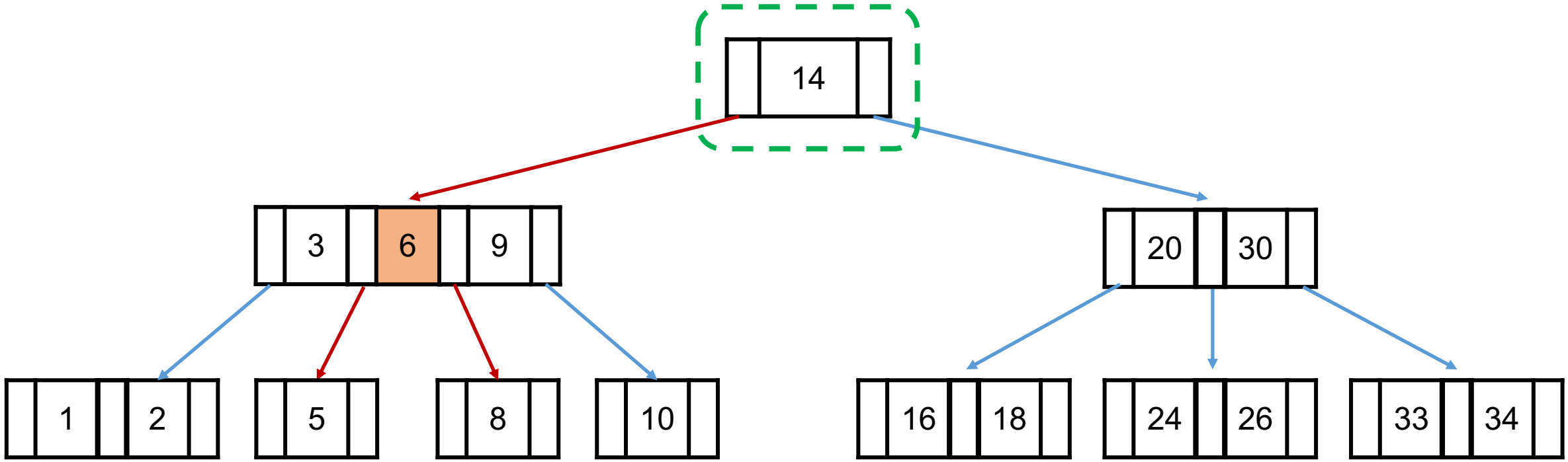
B 树 – 插入调整（上溢）

节点分裂



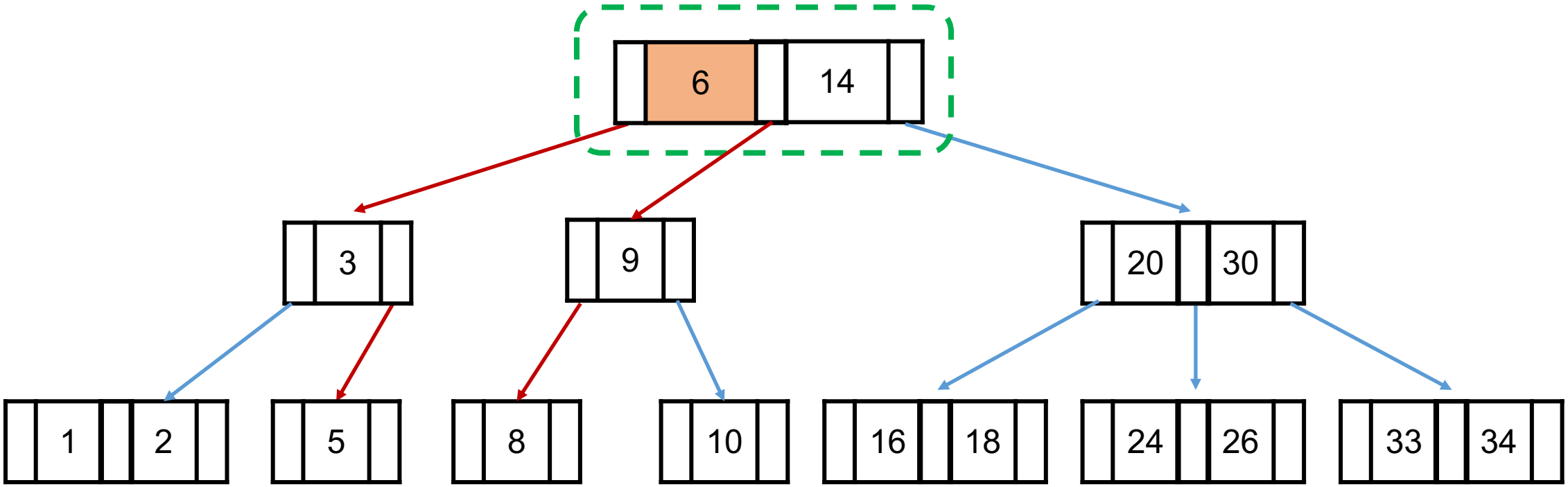
B 树 – 插入调整（上溢）

站在父节点处理

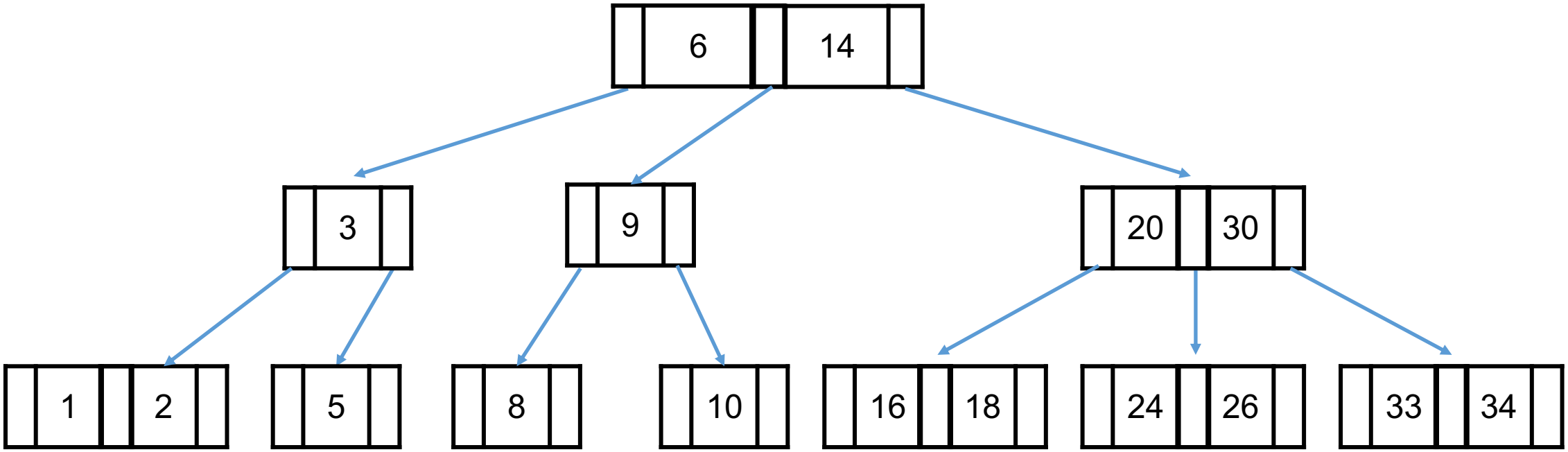


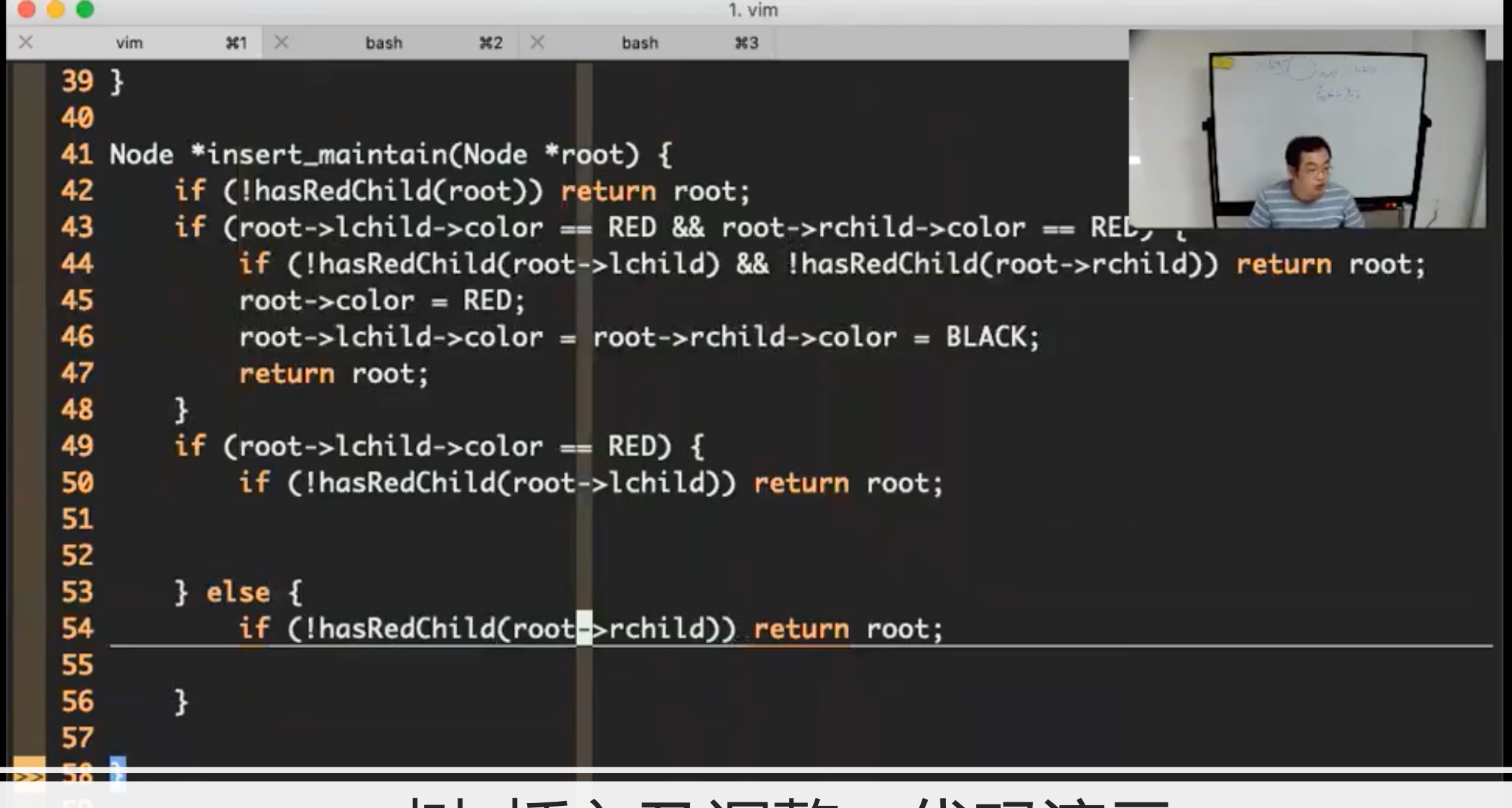
B 树 – 插入调整（上溢）

节点分裂



B 树 – 插入调整（上溢）





B 树-插入及调整：代码演示

~~1. 元素插入~~

~~2. 插入调整~~

3. 元素删除

4. 删除调整

~~1. 元素插入~~

~~2. 插入调整~~

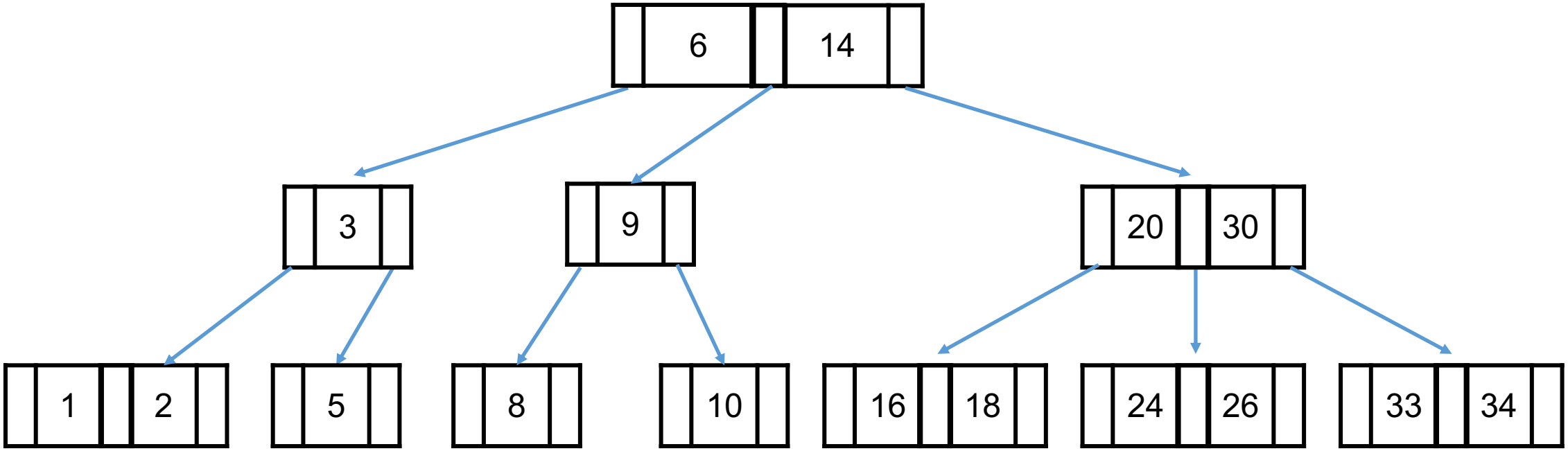
3. 元素删除

4. 删除调整

B 树 – 元素删除(1)

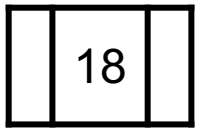
m 阶 B 树的元素删除:

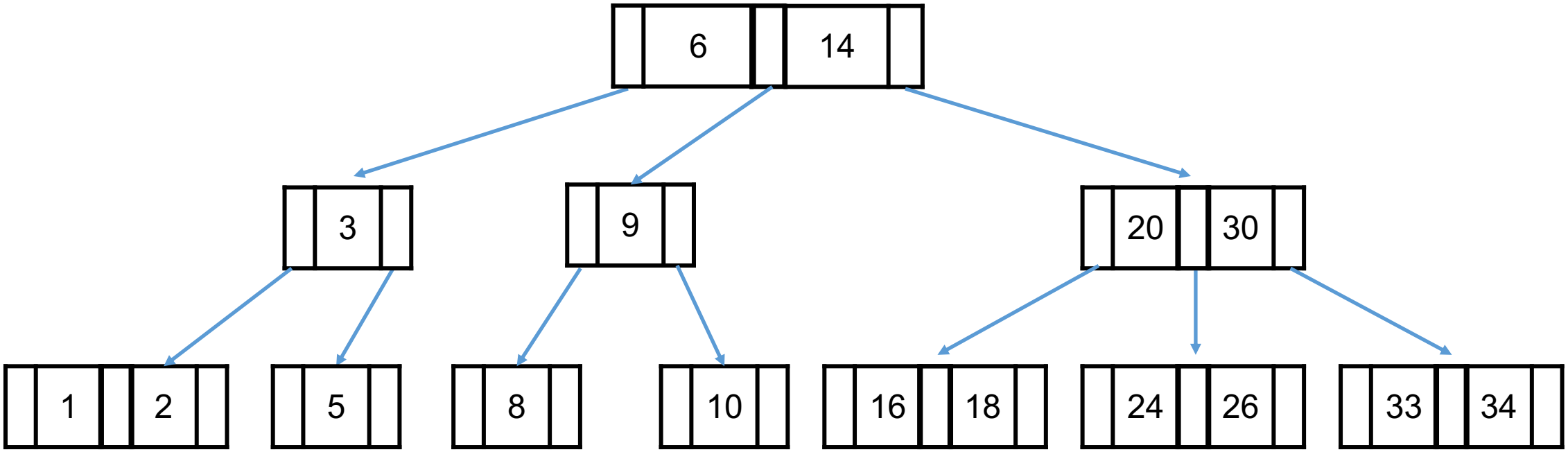
1. 终端节点直接删除
2. 非终端节点，与(前驱/后继)交换，再删除



B 树 – 元素删除(1)

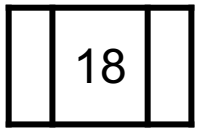
删除：

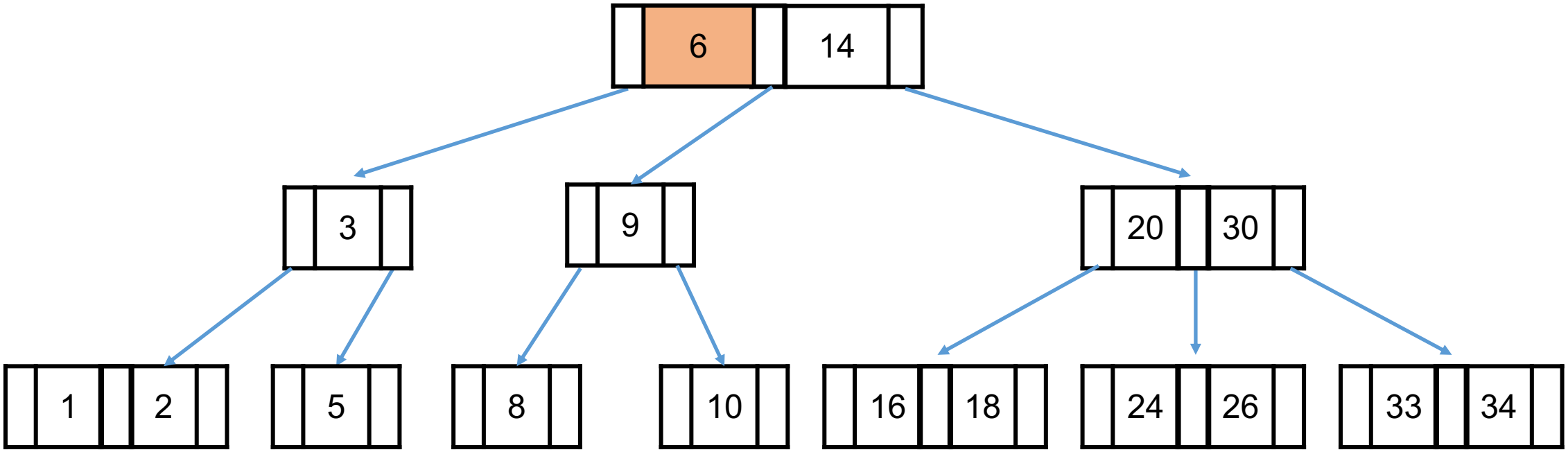




B 树 – 元素删除(1)

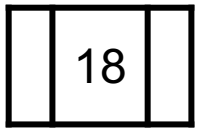
删除：

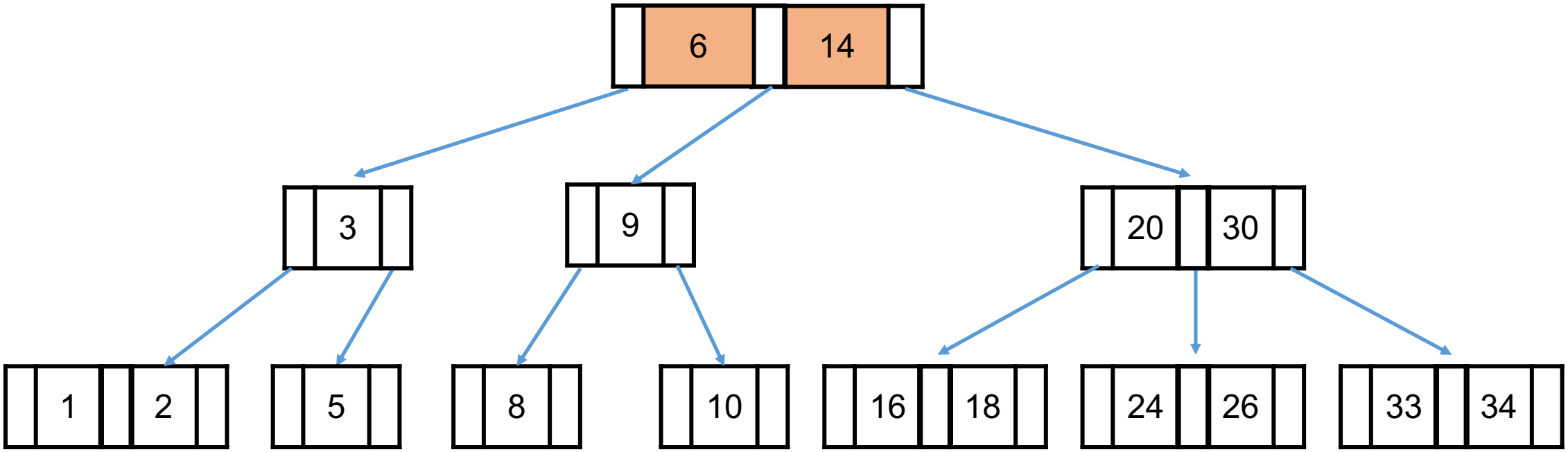




B 树 – 元素删除(1)

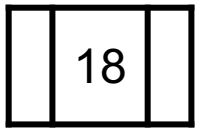
删除：

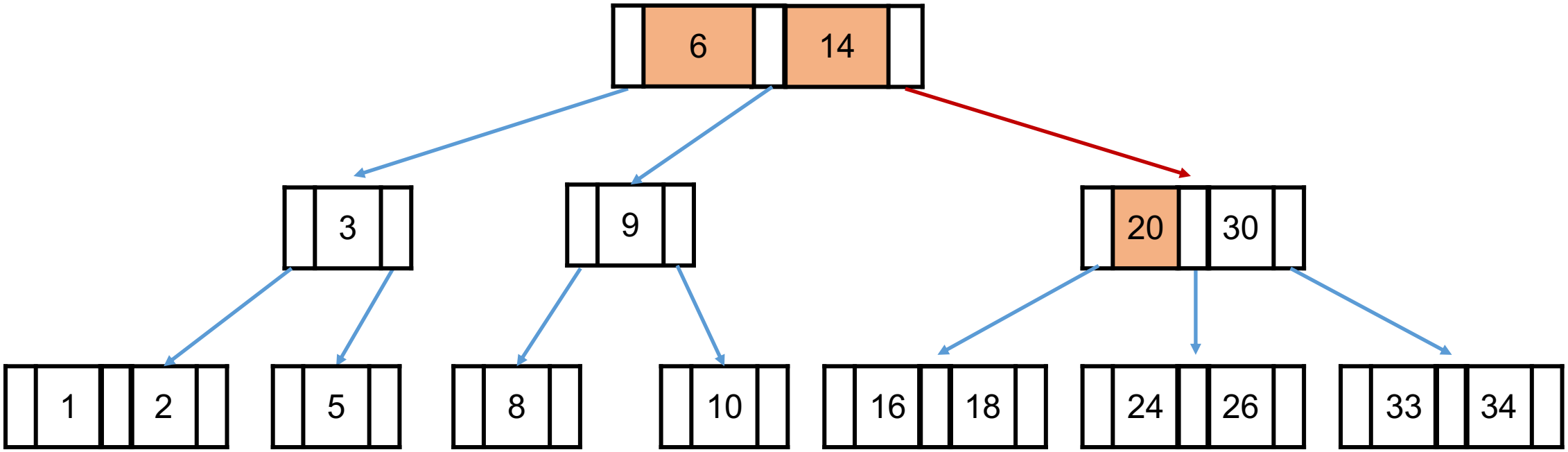




B 树 – 元素删除(1)

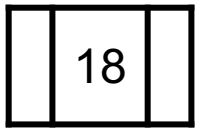
删除：

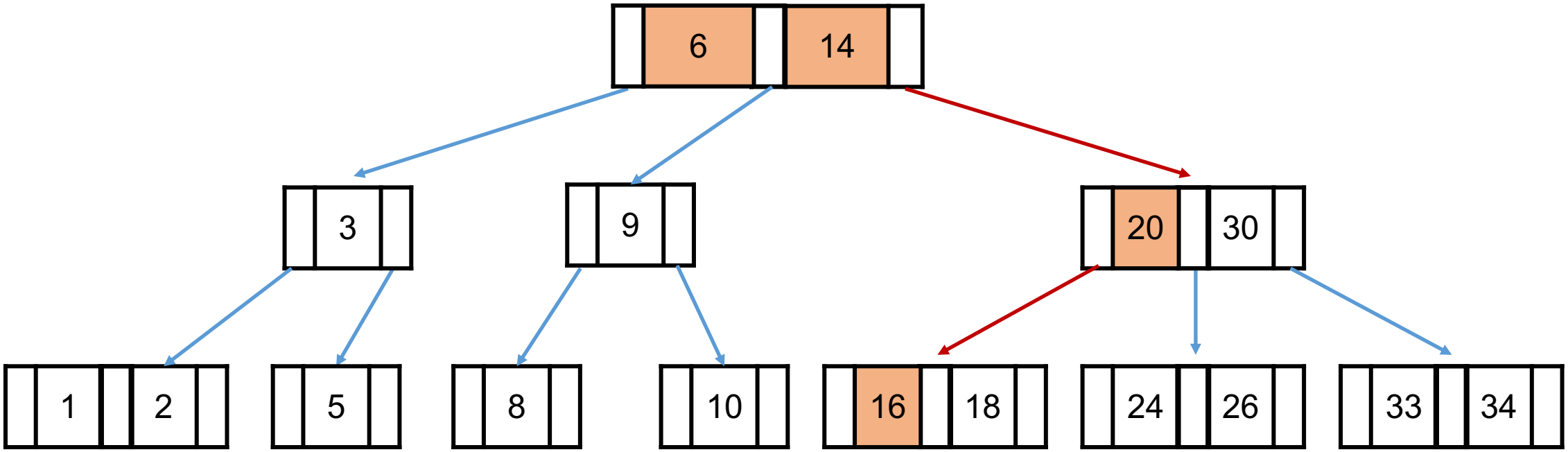




B 树 – 元素删除(1)

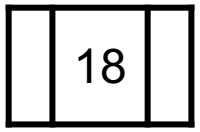
删除：

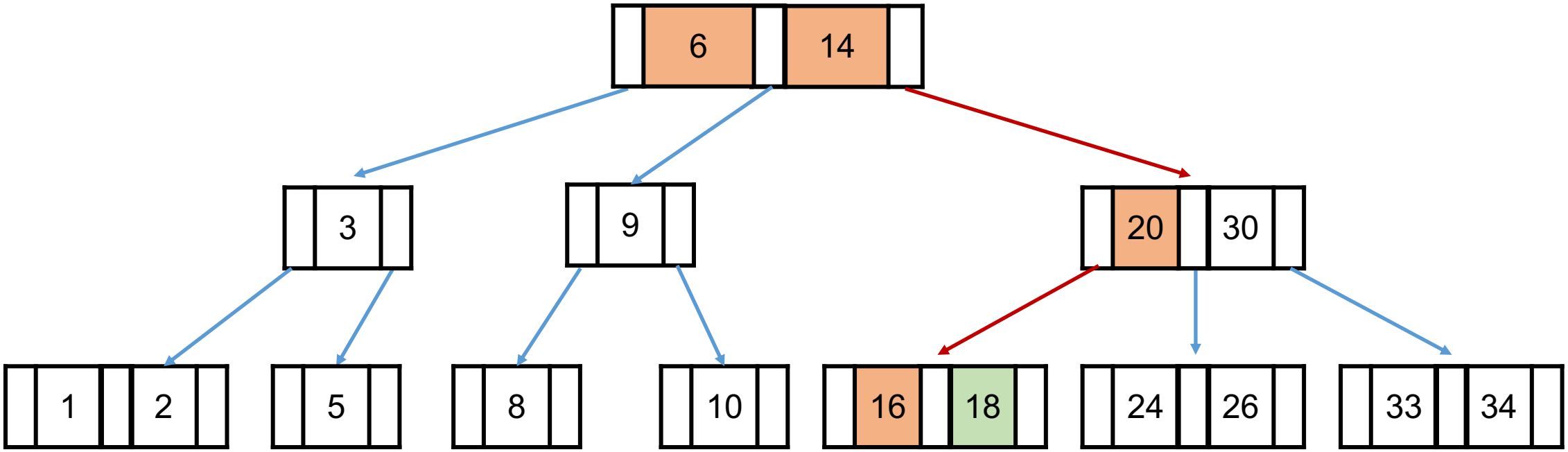




B 树 – 元素删除(1)

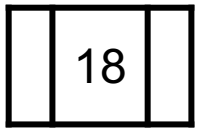
删除：

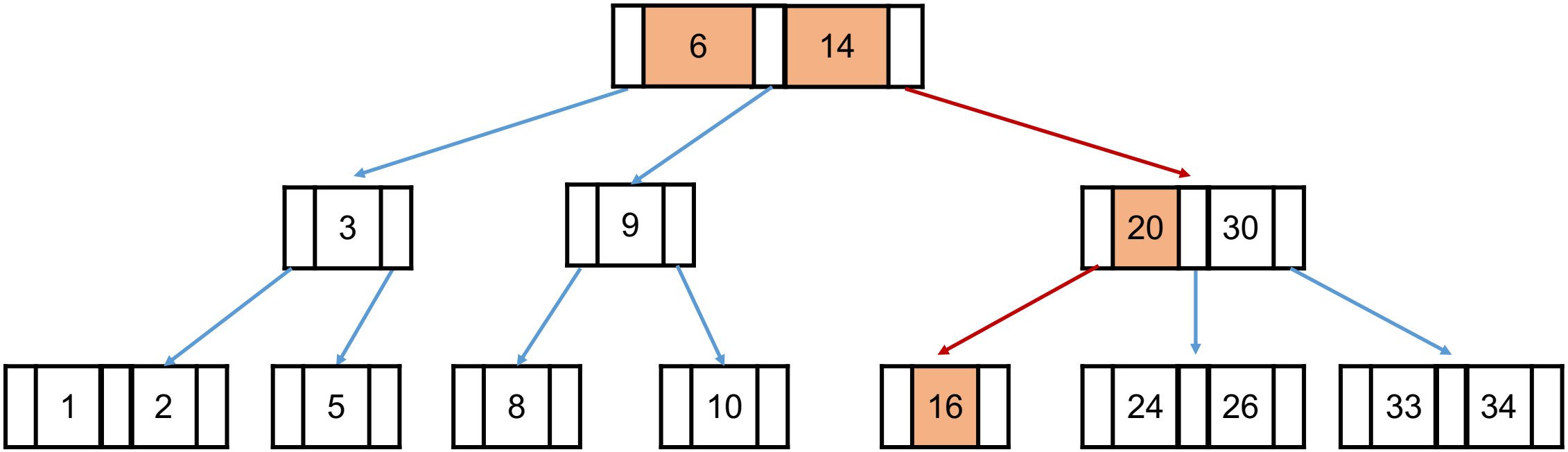




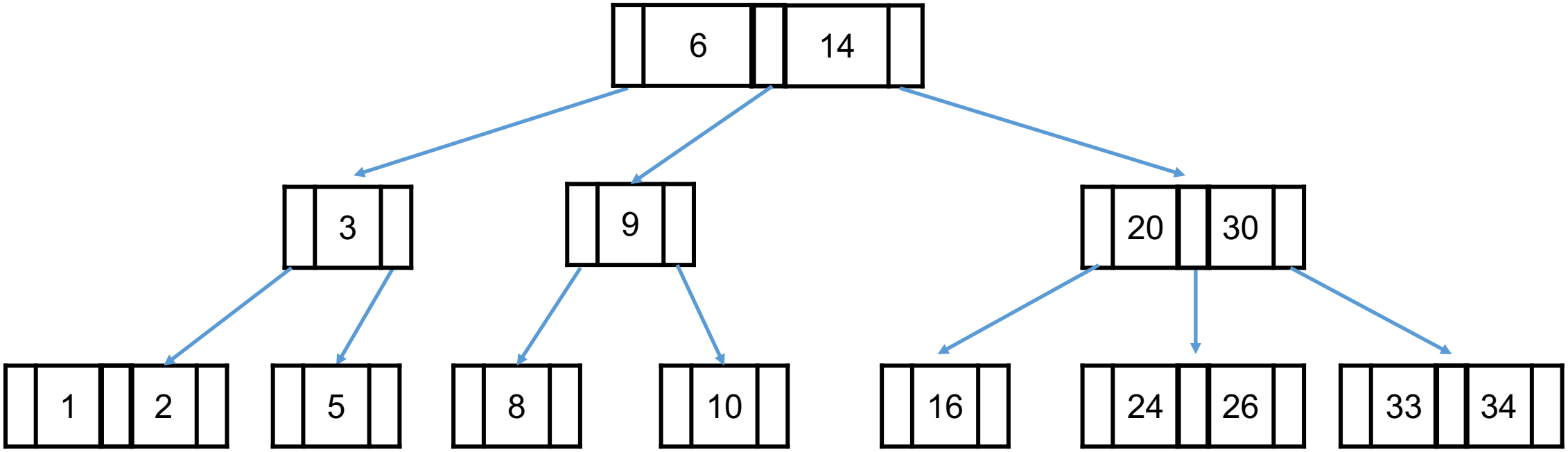
B 树 – 元素删除(1)

删除：



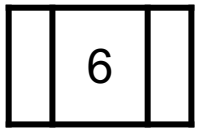


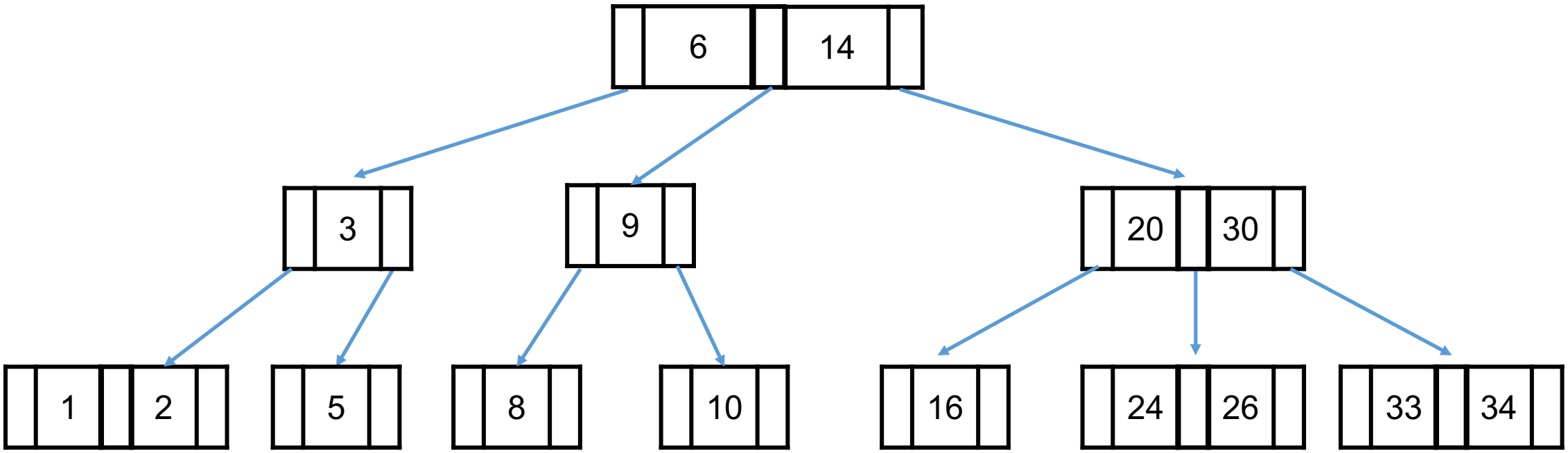
B 树 – 元素删除(1)



B 树 – 元素删除(2)

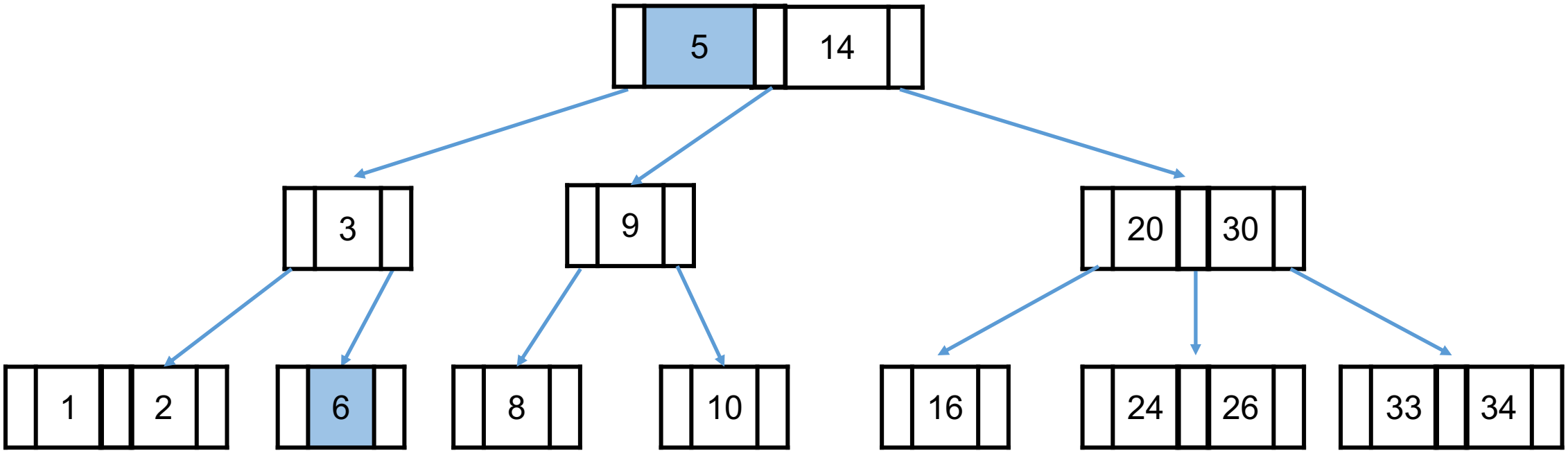
删除：





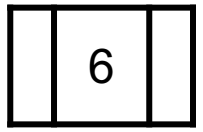
B 树 – 元素删除(2)

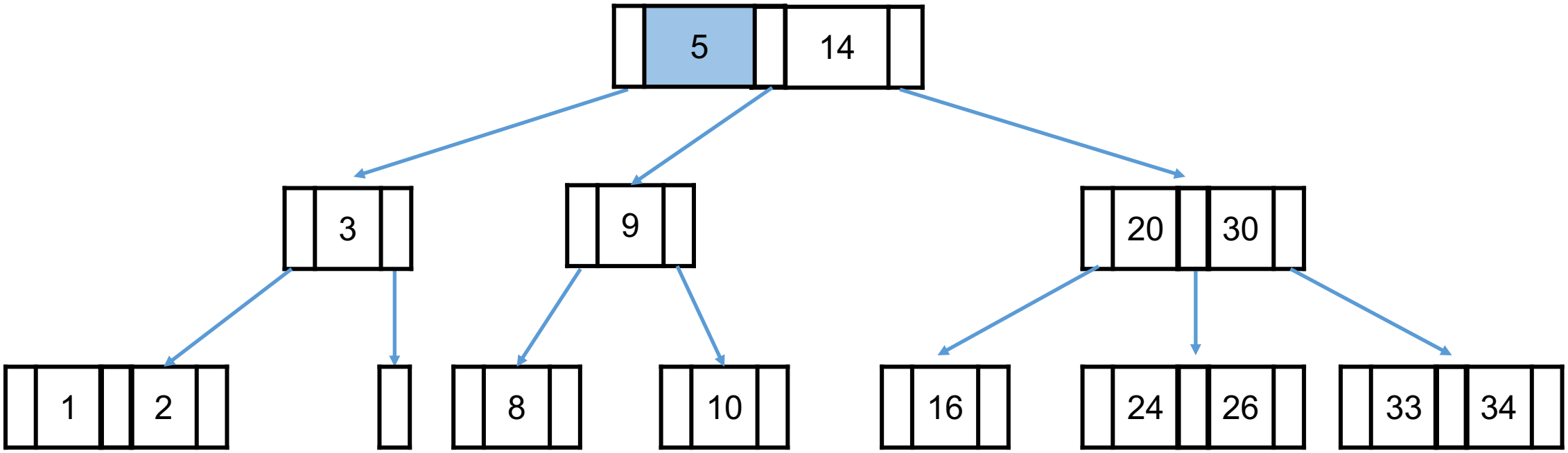
交换 6 和 5



B 树 – 元素删除(2)

删除：





~~1. 元素插入~~

~~2. 插入调整~~

~~3. 元素删除~~

4. 删除调整

~~1. 元素插入~~

~~2. 插入调整~~

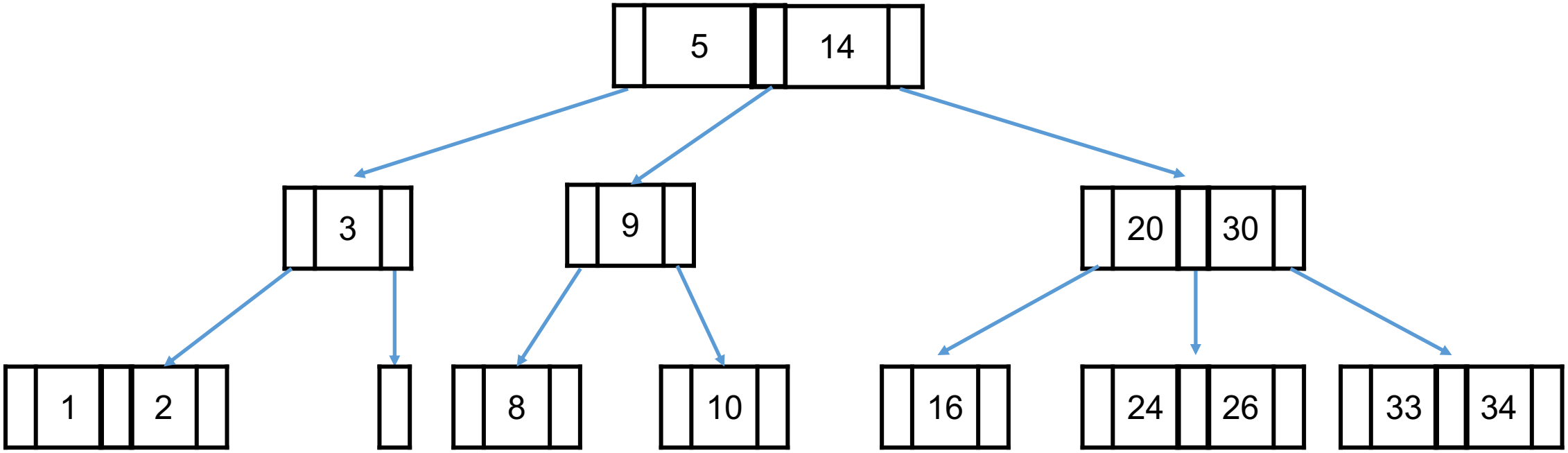
~~3. 元素删除~~

4. 删除调整

B 树 – 删除调整（下溢）

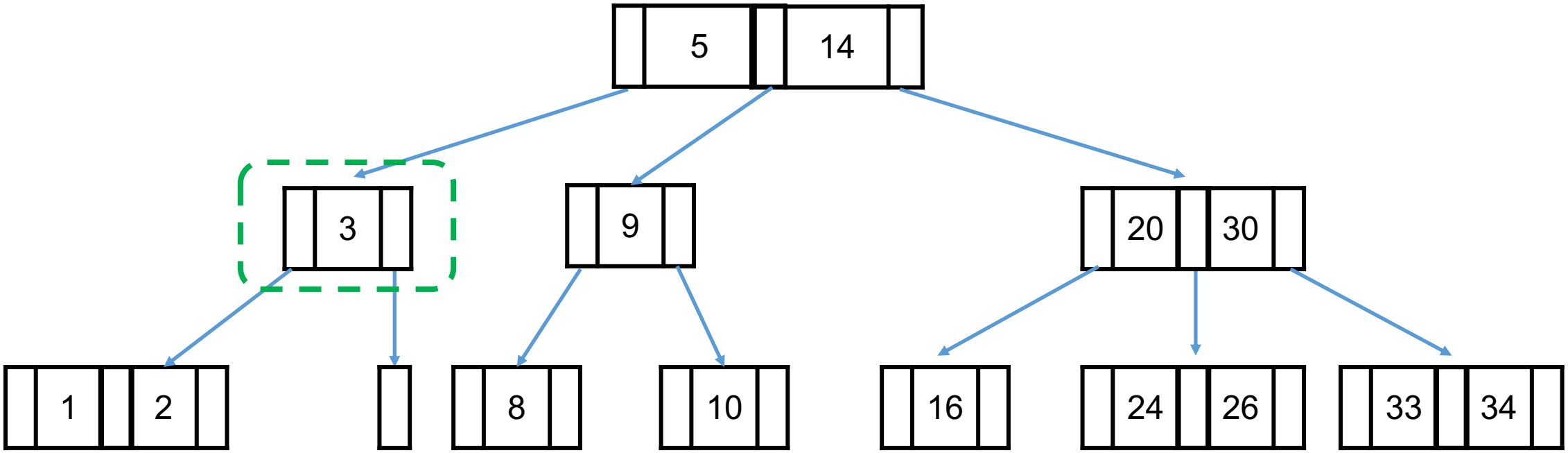
m 阶 B 树的删除调整:

1. 删除调整站在父节点处理，发生在节点关键字数量为 [m/2]-2 时
2. 删除调整的核心操作是: 左旋，右旋与合并



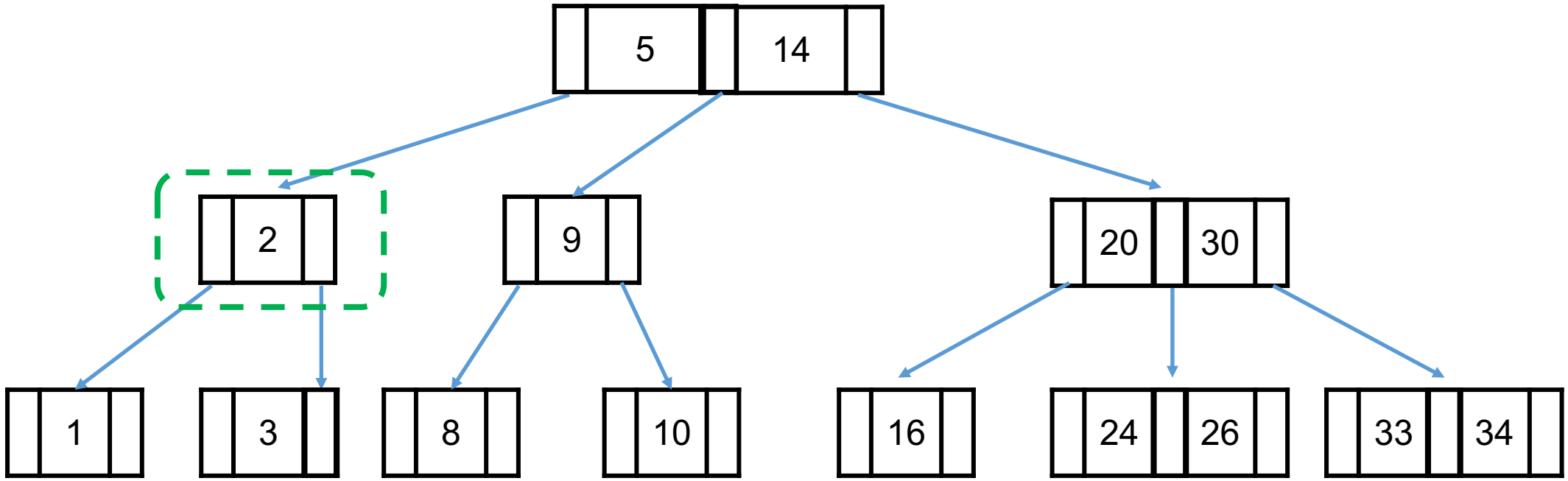
B 树 – 删除调整（下溢）

站在父节点处理

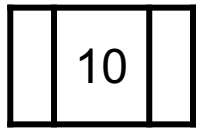


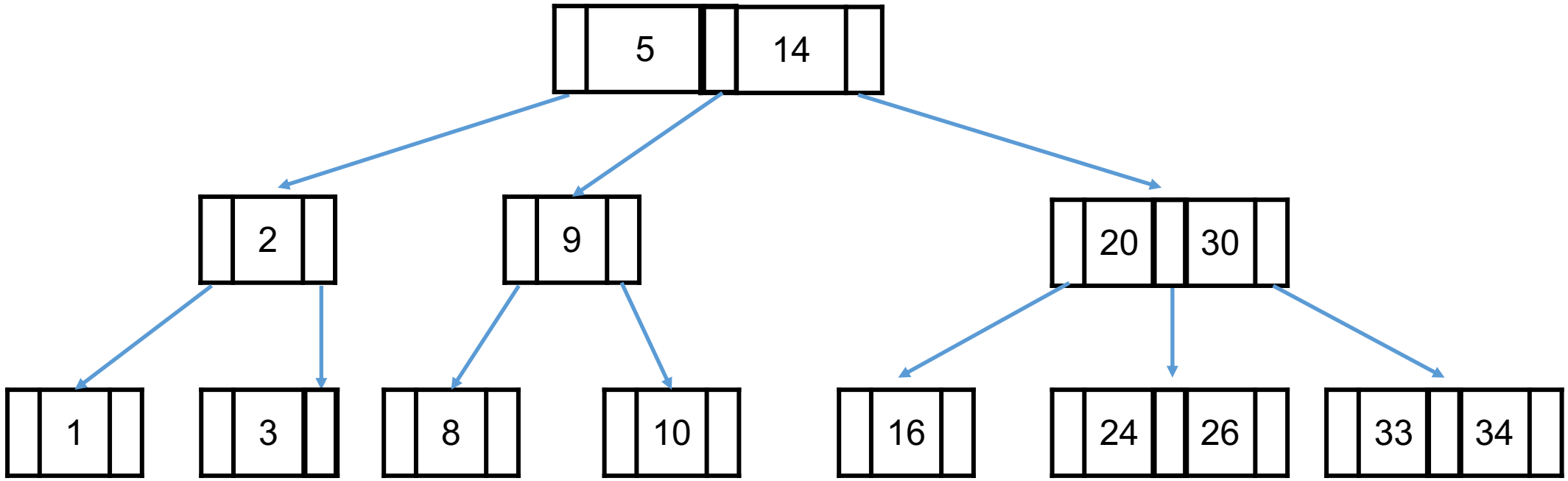
B 树 – 删除调整（下溢）

右旋处理

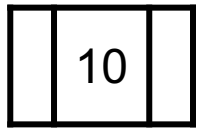


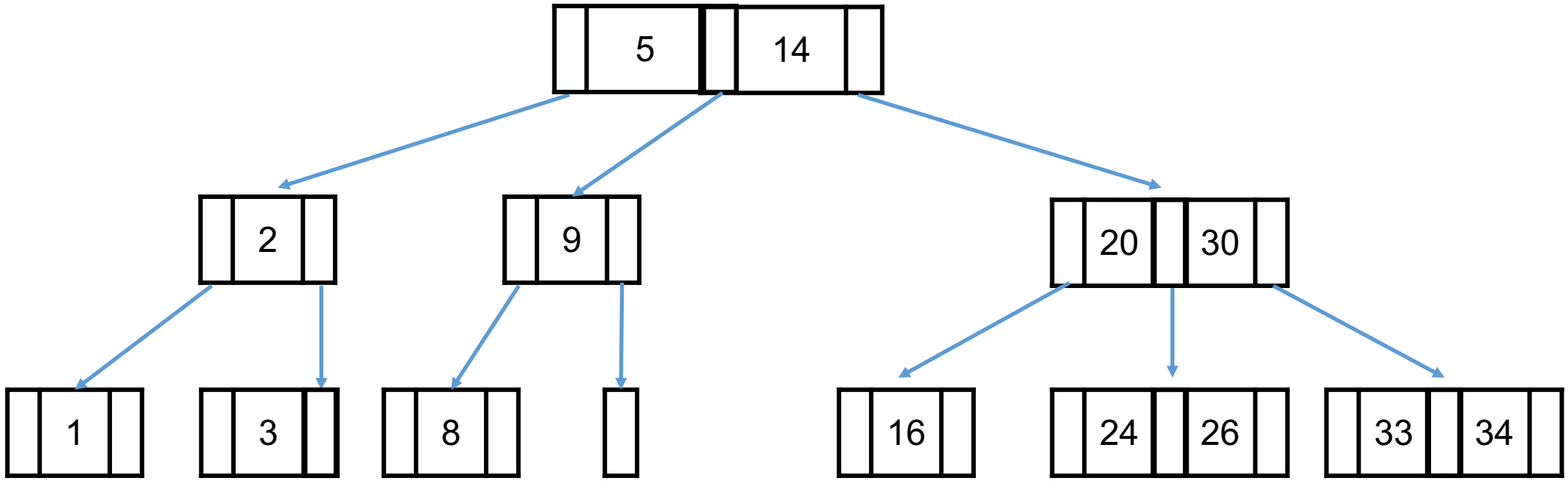
B 树 – 删除调整（下溢）

删除： 



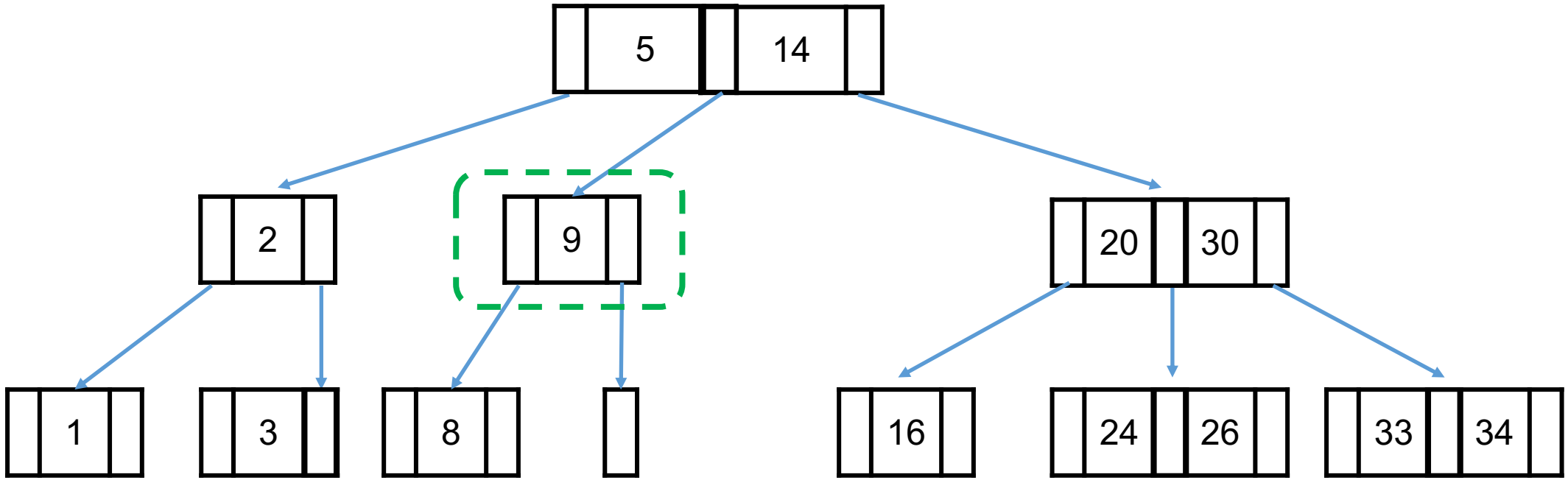
B 树 – 删除调整（下溢）

删除： 



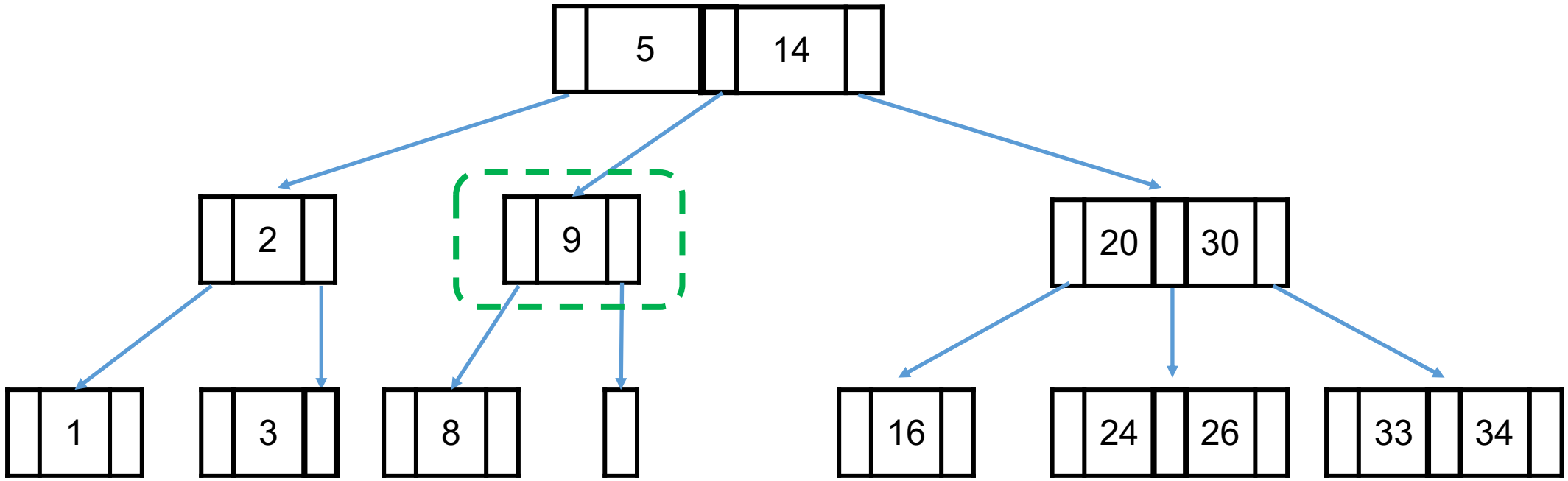
B 树 – 删除调整（下溢）

站在父节点处理



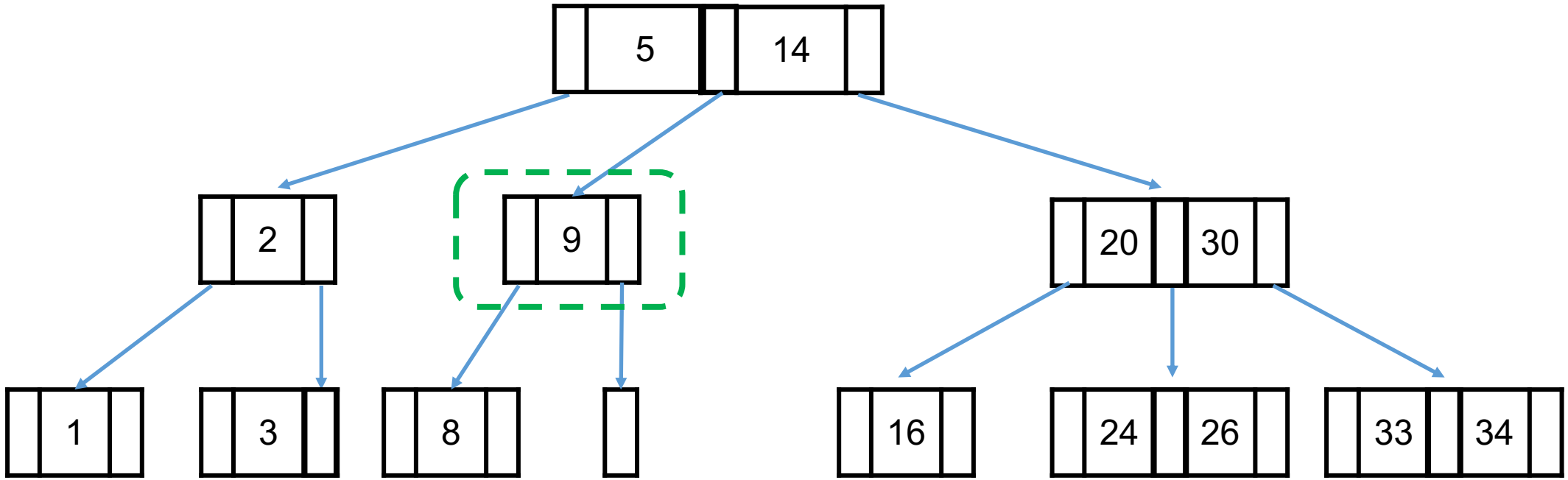
B 树 – 删除调整（下溢）

无法左旋和右旋



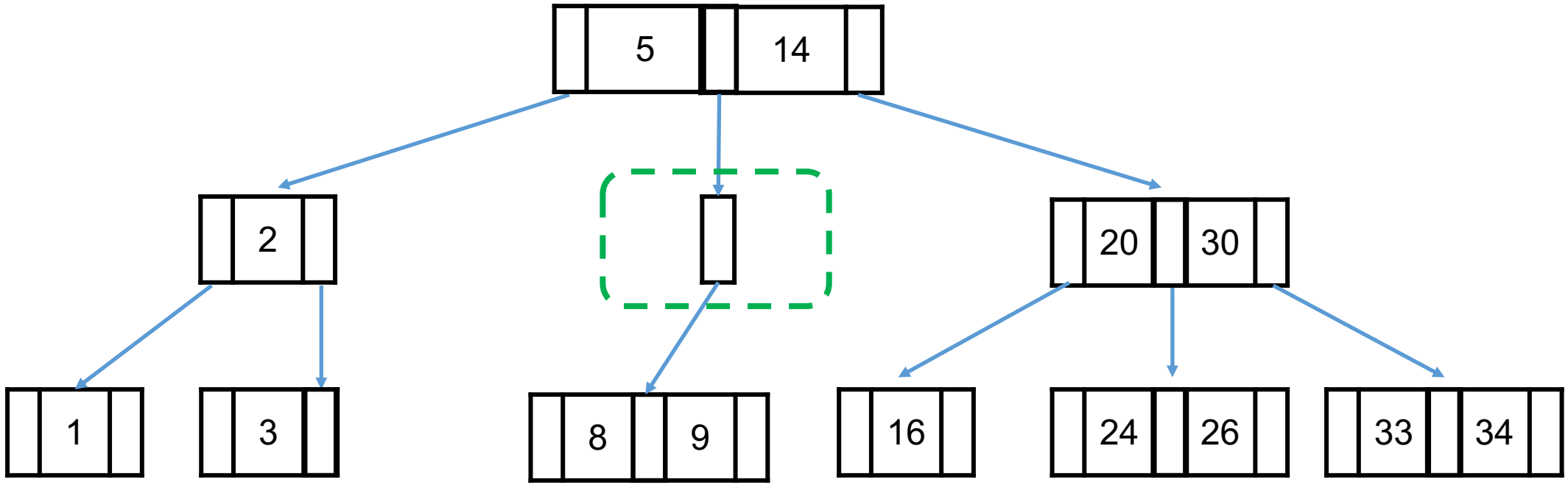
B 树 – 删除调整（下溢）

合并



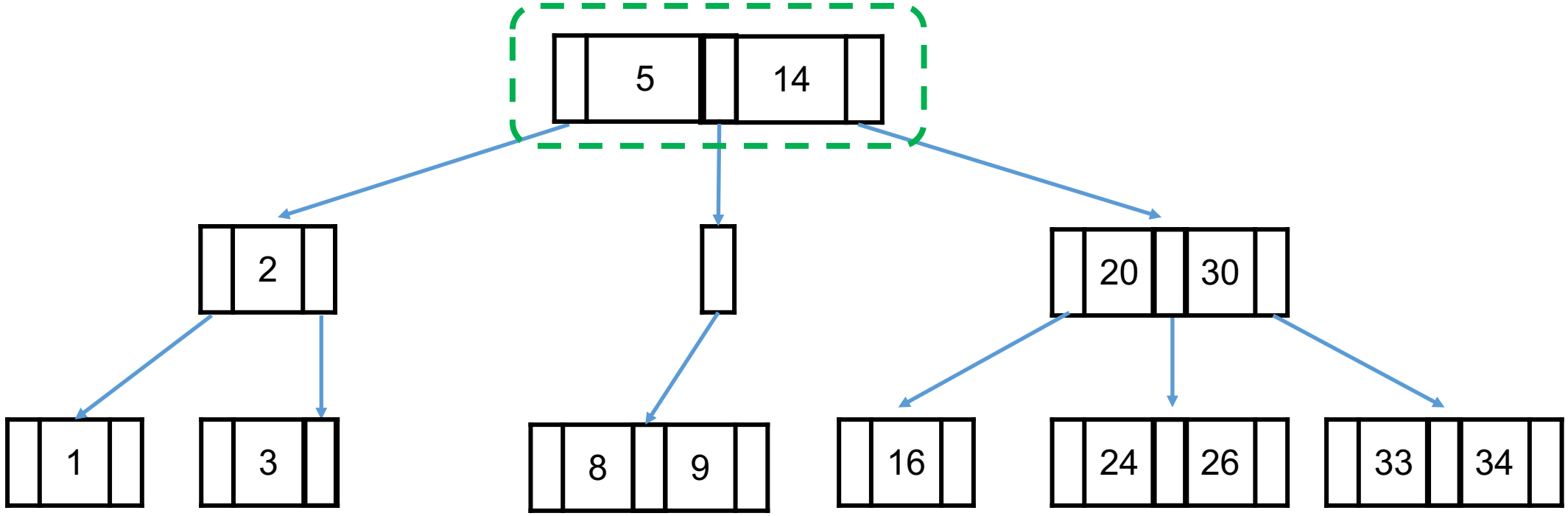
B 树 – 删除调整（下溢）

合并



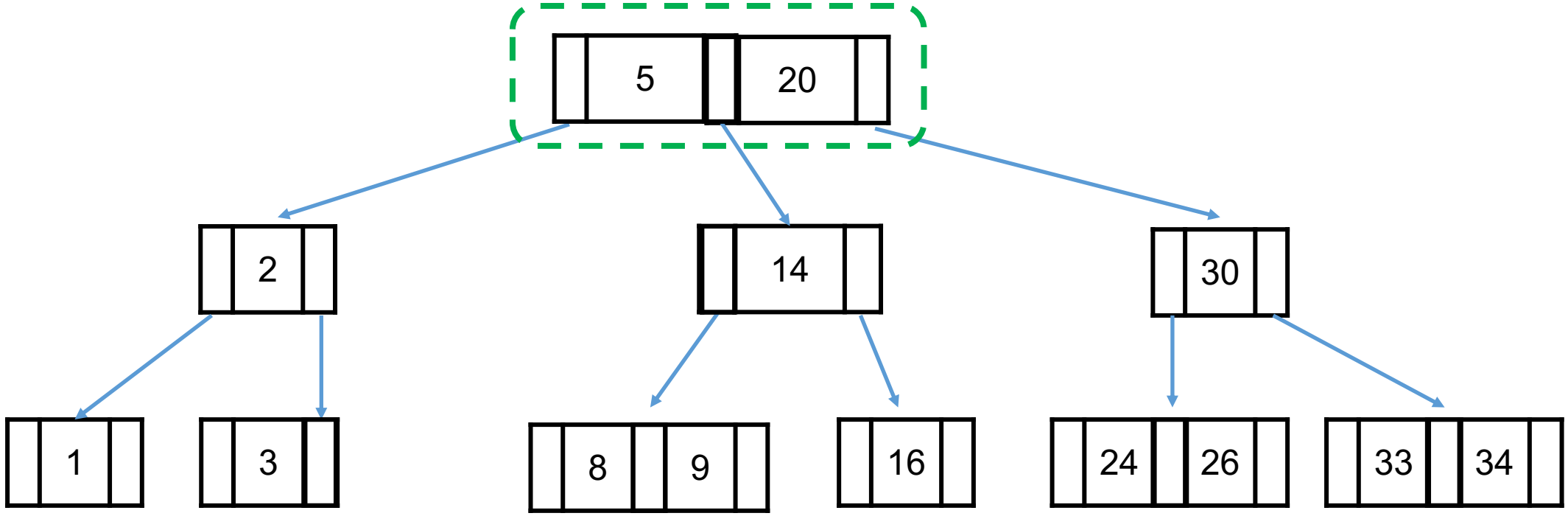
B 树 – 删除调整（下溢）

站在父节点处理



B 树 – 删除调整（下溢）

左旋



1. 元素插入

2. 插入调整

3. 元素删除

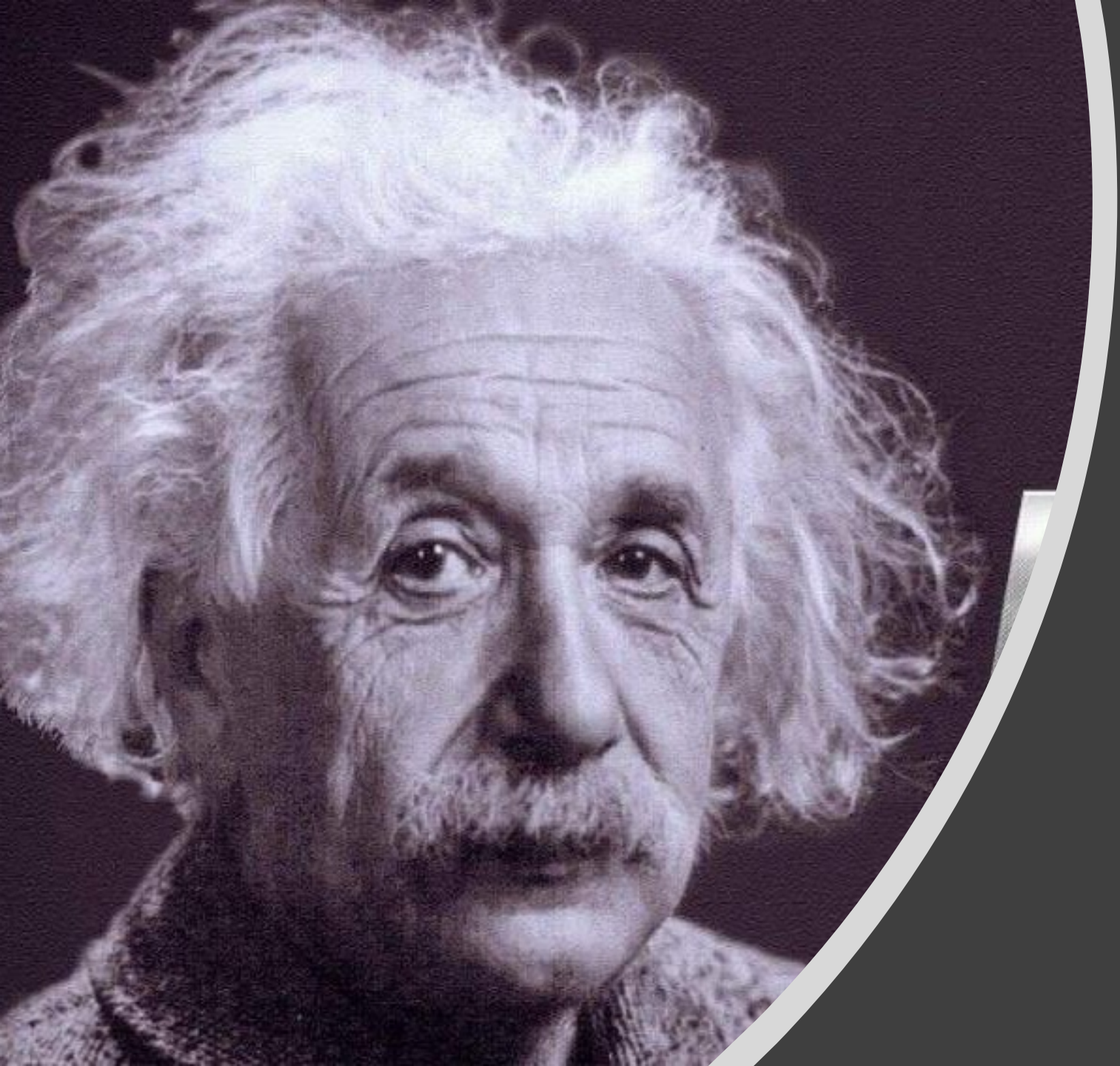
4. 删除调整

B 树 – 总结

m 阶 B 树，定义与操作总结：

1. 树中每个节点，最多含有 m 棵子树
2. 根节点中关键字数量范围 $1 \leq n \leq m-1$
3. 非根结点中关键字数量范围 $\lceil m/2 \rceil - 1 \leq n \leq m-1$
4. 插入调整为了解决『上溢』节点，关键字数量为 m 的节点
5. 删除调整为了解决『下溢』节点，关键字数量为 $\lceil m/2 \rceil - 2$ 的节点
6. 插入调整的核心操作是：节点分裂
7. 删除调整的核心操作是：左旋，右旋与合并

B 树-删除及调整：代码演示



为什么
会出一样的题目？

《船说：算法与数据结构》
第8章-平衡树基础