



# 第11章 索引

邹磊

北京大学计算机科学技术

[zoulei@pku.edu.cn](mailto:zoulei@pku.edu.cn)

- **“大数据” 需存放在大型数据库中**
- **既要支持高效检索，又要动态灵活**
  - **随数据的插入、删除和更新而动态调整**
- **索引技术是支持高效数据组织管理的有效手段**

## ➤ 输入顺序文件

### ➡ 按照记录进入系统的顺序存储记录

✓ 结构相当于一个磁盘中未排序的线性表

### ➡ 不支持高效检索

➤ **主码( primary key )是数据库中的每条记录的唯一标识**

➡ **例如，身份证号码**

➡ **如果只有主码，不便于各种灵活检索**

- **辅码：数据库中可出现重复值的码（属性）**
  - ➡ 如姓名，性别等
- **辅码索引把一个辅码值与具有这个辅码值的多条记录的主码值关联起来**
  - ➡ 多数检索都是利用辅码索引完成的

## ➤ 索引

- ➡ 把关键码与它对应的记录位置关联起来的过程
- ➡ (关键码, 指针)对, 即(key, pointer)
- ➡ 指针指向主数据库文件（或“主文件”）中的完整记录

## ➤ 索引文件：用于记录这种联系的文件

## ➤ 索引技术是组织大型数据库的一项重要技术

- ➡ 高效率的检索
- ➡ 支持动态的数据插入、删除和更新

- 原始数据记录组成的文件称为**主文件**
- 索引数据组成的文件称为**索引文件**
- 一个主文件可能有多个相关索引文件
  - ➡ 每个索引文件往往支持一个索引字段
  - ➡ 不需要重新排列主文件
- 通过索引文件高效访问记录中的关键码

➤ **稠密索引**：对**每个记录**建立一个索引项

➡ **特点**：主文件不需按关键码次序排列

➤ **稀疏索引**：对**一组记录**建立一个索引项

➡ **特点**：主文件必须按照关键码次序存放

➡ 可以把记录分成多个组（块）








✓ 索引指针指向的这一组记录在磁盘中的起始位置



# 稠密索引示例



关键码 指针

8		$r_1$
20		$r_2$
35		$r_3$
40		$r_4$
52		$r_5$
56		$r_6$
61		$r_7$

索引表

关键码 其它数据项

8	...
20	...
52	...
35	...
40	...
61	...
56	...

文件

有序

无序或有序

# 稀疏索引示例



关键码 指针

8	
40	
56	
...	
...	
...	

索引表

有序

关键码 其它数据项

$r_1$	8	...
$r_2$	20	...
$r_3$	35	...
$r_4$	40	...
$r_5$	52	...
$r_6$	56	...
$r_7$	61	...

文件

有序

**11.1 线性索引**

**11.2 静态索引**

**11.3 倒排索引**

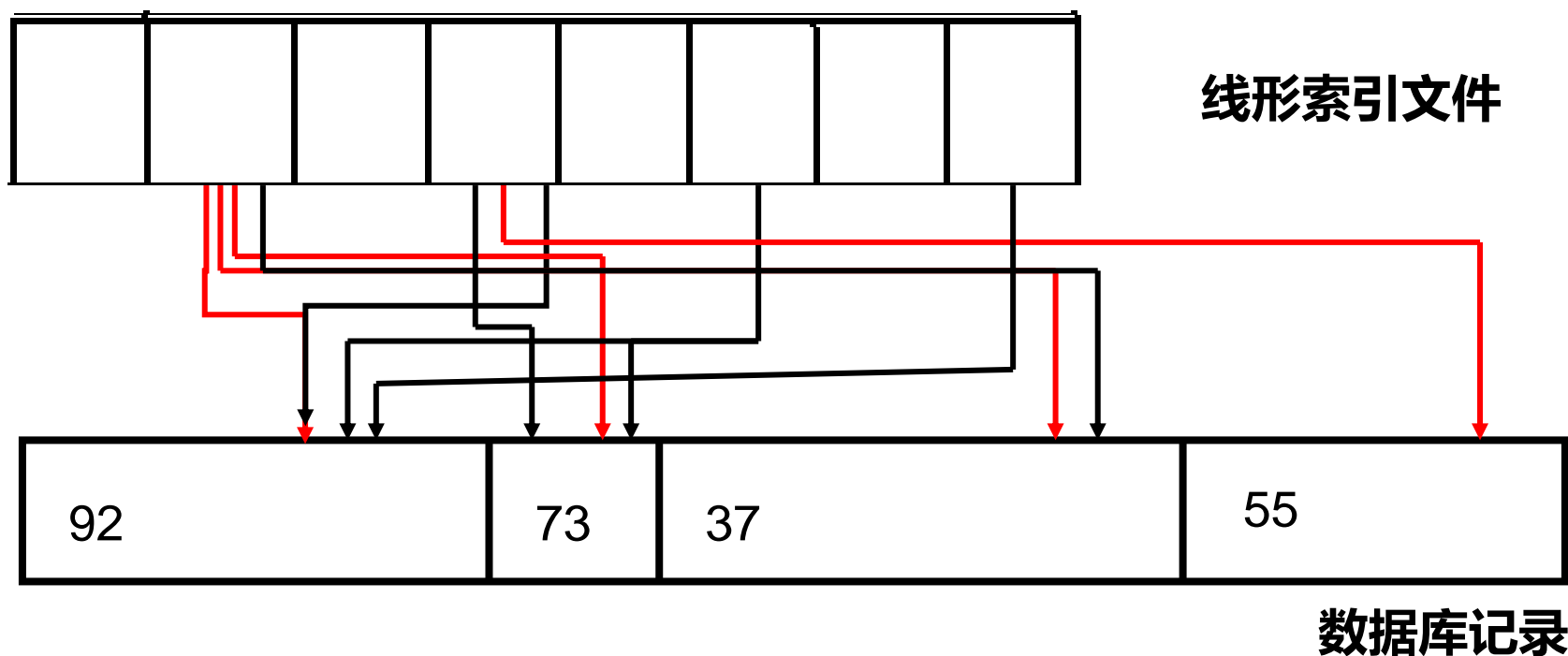
**11.4 动态索引**

**11.5 红黑树**

# 11.1 线性索引



- 按照索引码值的顺序进行排序的文件
- 索引文件中的指针指向存储在磁盘上的文件记录起始位置或者主索引中主码的起始位置



- 可对变长的数据库记录访问

- 支持对数据的高效检索

  - ➡ 二分检索

- 线性索引太大，存储在磁盘中

  - ➡ 一次检索可能多次访问磁盘，影响检索的效率

  - ➡ 使用二级线性索引

- 设磁盘块大小是1024字节，每对(关键码, 指针)索引对需要8个字节

- ➡  $1024 / 8 = 128$

- ➡ 每磁盘块可以存储128条索引对

- 假设数据文件包含10000条记录

- ➡ 稠密一级线性索引中包含10000条记录

- ✓  $10000/128 = 78.1$

- ✓ 那么一级线性索引占用79个磁盘块

- ➡ 相应地，二级线性索引文件中有79项索引对

- ➡ 这个二级线性索引文件可以在一个磁盘块

# 二级线性索引的例子



- 关键码与相应磁盘块中第一条记录的关键码值相同
- 指针指向相应磁盘块的起始位置

二级索引

1	2003	5744	10723	.....
---	------	------	-------	-------

一级索引

1.....	2002	2003	5583	5744	9297	10723	13293
.....							

磁盘块



## 例如：检索关键码为2555的记录

关键字2555的记录指针

二级索引

1	2003	5744	10723	.....
---	------	------	-------	-------

线性索引

1	2002	2003	5583	5744	9297	10723	13293
.....							

磁盘块

关键码为2555的记录

1. 二级线性索引文件读入内存
2. 二分法找关键码的值小于等于2555的最大关键码所在一级索引磁盘块地址——关键码为2003的记录
3. 根据记录2003中的地址指针找到其对应的一级线性索引文件的磁盘块，并把该块读入内存
4. 按照二分法对该块进行检索，找到所记录在磁盘上的位置
5. 最后把所需记录读入，完成检索操作



## 11.2 静态索引

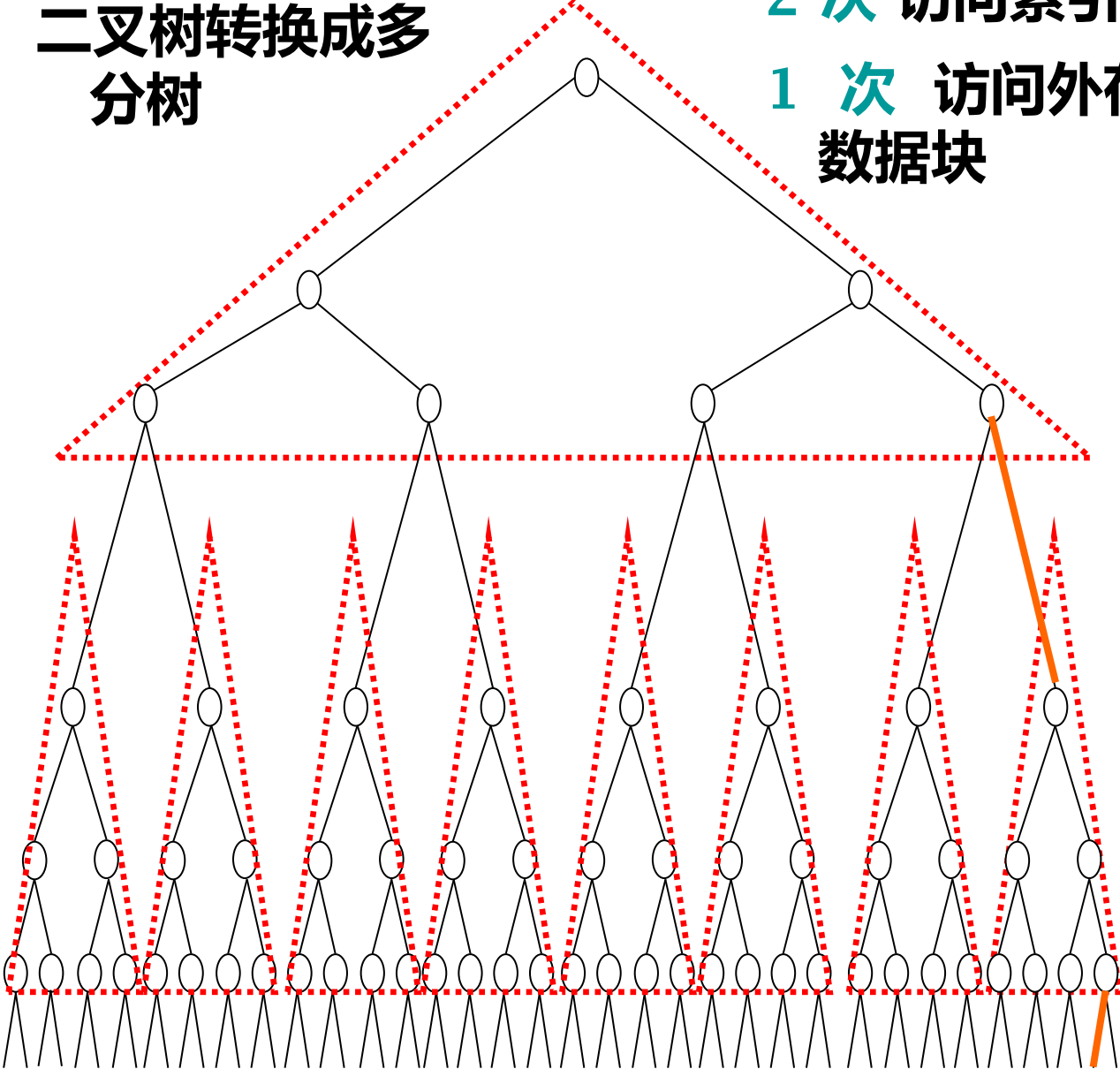
- 索引结构在文件创建时生成
- 一旦生成就固定下来，在系统运行(例如插入和删除记录)过程中，索引结构不做改变
- 只有当文件再组织时才允许改变索引结构

### 多分树

- 组织索引一般不用二叉树而采用多分树
- 大大减少访问外存的次数

二叉树转换成多  
分树

2 次 访问索引块  
1 次 访问外存  
数据块



- 在什么情况下需要组织二级线性索引?
- 多分树的阶（子结点的个数）应该怎么确定?

# 11.3 倒排索引



- 按属性值建立索引，索引表中的每一项包括
  - 一个属性值和具有该属性值的各关键码或者记录地址
    - ✓ 关键码对应的指针地址
- 由属性值来确定记录的位置，称之为**倒排索引**
- 带有倒排索引的文件称为**倒排文件**
- 分类
  - 基于属性的倒排
  - 对正文文件的倒排

# 11.3.1 基于属性的倒排



➤ 按属性值建立起来索引表，称倒排表

➡ (attr, ptrList)

✓ 属性值

✓ 记录指针：可以是关键码，或该记录的主文件地址

➤ 倒排文件：带有倒排索引的文件

# 教师数据库主表



EMP#	NAME	Department	Profession	Specialty	Address
0155	李宇	数学	教授	代数	C105
0421	刘阳	外语	助教	英语	E310
0208	赵亮	物理	助教	力学	C211
0211	张伟	物理	讲师	原子物理	D508
0132	王亮	数学	助教	几何	E220
0119	王卓	数学	讲师	代数	B102
0330	孙丽	计算机	教授	软件	A108
0455	刘珍	外语	讲师	法语	A225
0310	周兵	计算机	讲师	英语	B423
0341	何江	计算机	助教	计算机	F406
.....					

# 教师数据库倒排表



Department list	EMP#
数学 物理 计算机 外语	0155, 0132, 0119 0208, 0211 0330, 0310, 0341 0421, 0455
Profession list	EMP#
教授 讲师 助教	0155, 0330 0211, 0119, 0455, 0310 0421, 0208, 0132, 0341
Specialty list	EMP#
代数 几何 力学 原子物理 软件 英语 法语	0155, 0119 0132 0208 0211 0330, 0341 0421, 0310 0455

## ➤ 优点

- ➡ 支持基于属性的高效检索

## ➤ 缺点

- ➡ 花费了保存倒排表的存储代价
- ➡ 降低了更新运算的效率



# 11.3.2 对正文文件的倒排



## ➤ 正文索引(Text Indexing)

➡ 支持对文本内容的快速检索

## ➤ 方法

➡ 词索引(word index)

➡ 全文索引(full-text index)

## ➤ 基本思想

- ➡ 从正文中抽取出**关键词**，然后用这些关键词组成适合快速检索的数据结构

## ➤ 支持多种文本类型

- ➡ 适用于英文
- ➡ 中文等东方文字要经过“切词”处理

## ➤ 基本思想

- 正文看作一个长的字符串，记录每个字符串的开始位置，查询可以针对正文中的任何字符串进行
- 可以对**每个字符串**建立索引，从而使查询词不再限于关键词
- 索引结构将耗费更大的存储空间

## ➤ 一个已经排过序的关键词列表

### ➡ 其中每个关键词指向一个倒排表 (posting list)

✓ 指向该关键词出现文档集合

✓ 在文档中的位置

文档编号	文本内容
1	Pease porridge hot, please porridge cold,
2	Pease porridge in the pot,
3	Nine days old.
4	Some like it hot, some like it cold,
5	Some like it in the pot,
6	Nine days old.

文档编号	文本内容
1	Pease porridge hot, please porridge cold,
2	Pease porridge in the pot,
3	Nine days old.
4	Some like it hot, some like it cold,
5	Some like it in the pot,
6	Nine days old.

类似处理1中的其他词语

同理，处理2中的词语

依次处理所有文档

# 倒排索引



编号	词语	(文档编号, 位置)
1	cold	(1,6)
2	days	
3	hot	(1,3)
4	in	
5	it	
6	like	
7	nine	
8	old	
9	pease	(1,1) (1,4) (2,1)
10	porridge	(1,2) (1,5)
11	pot	
12	some	
13	the	

## 1. 对文档集中的所有文件都进行分割处理，把正文分成多条记录文档

➡ 切分正文记录取决于程序的需要

✓ 定长的块、段落、章节，甚至一组文档

## 2. 给每条记录赋一组关键词

- ▶ 以人工或自动的方式从记录中抽取关键词

- ▶ 停用词(Stopword) ,

  - ✓ 如 ‘the’、‘is’、‘at’、‘which’、‘on’等

- ▶ 抽词干(Stemming)

- ▶ 切词 (segmentation)

  - ✓ “Knowledge is power“, 可自然分割为 Knowledge/ is/ power

## 3. 建立正文倒排表、倒排文件

- ➡ 得到各个关键词的集合
- ➡ 对于每一个关键词得到其倒排表
- ➡ 然后把所有的倒排表存入文件
  - ✓ 记录每个关键词在文件中开始的位置（也可以记录每个关键词的出现次数）



- 第一步，在倒排文件中检索关键词
- 第二步，如果找到了关键词，那么获取文件中的对应的倒排表，并获取倒排表中的记录
- 通常使用另一个索引结构（字典）进一步对关键词表进行有效索引
  - Trie
  - 散列

- 高效检索，用于文本数据库系统
- 支持的检索类型有限
  - ➡ 检索词有限
    - ✓ 只能用索引文件中的关键词
  - ➡ 需要的空间代价往往很高（50%~300%）

# 11.4 动态索引



## ➤ 11.4.1 B树

## ➤ 11.4.2 B<sup>+</sup>树

## ➤ 动态索引结构

- ➡ 文件创建时生成
- ➡ 结构随着插入、删除等操作而改变

## ➤ 目的

- ➡ 保持最佳的检索性能

# 11.4.1 $m$ 阶B树的结构定义



➤  $m$  阶B-树是一棵  $m$  路查找树，或者为空，或者：

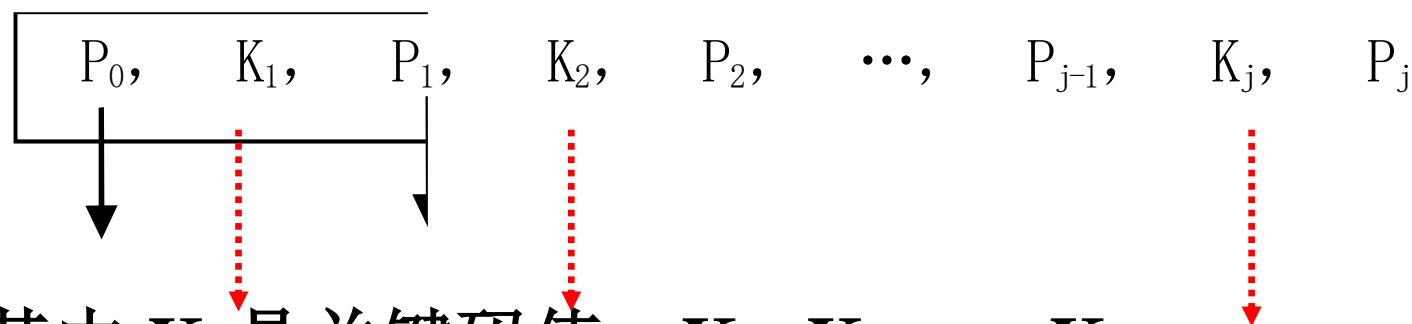
➡ **上界：** 树中每个结点至多有 $m$ 个子结点

➡ **下界：** 根结点至少有2棵子树，其它非叶结点至少有 $\lceil m/2 \rceil$ 棵子树

➡ 有 $k$ 棵子树的结点有 $k-1$ 个关键码

➡ 叶结点都位于同一层，有 $\lceil m/2 \rceil - 1$ 到 $m-1$ 个关键码

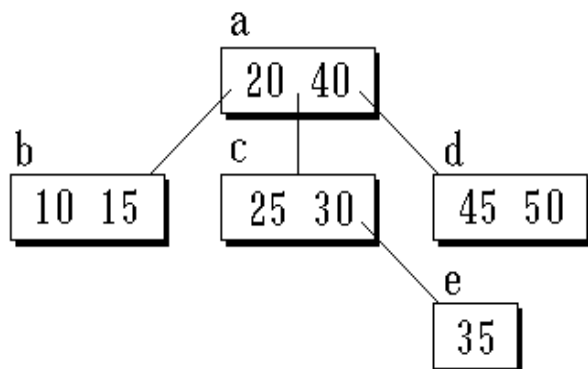
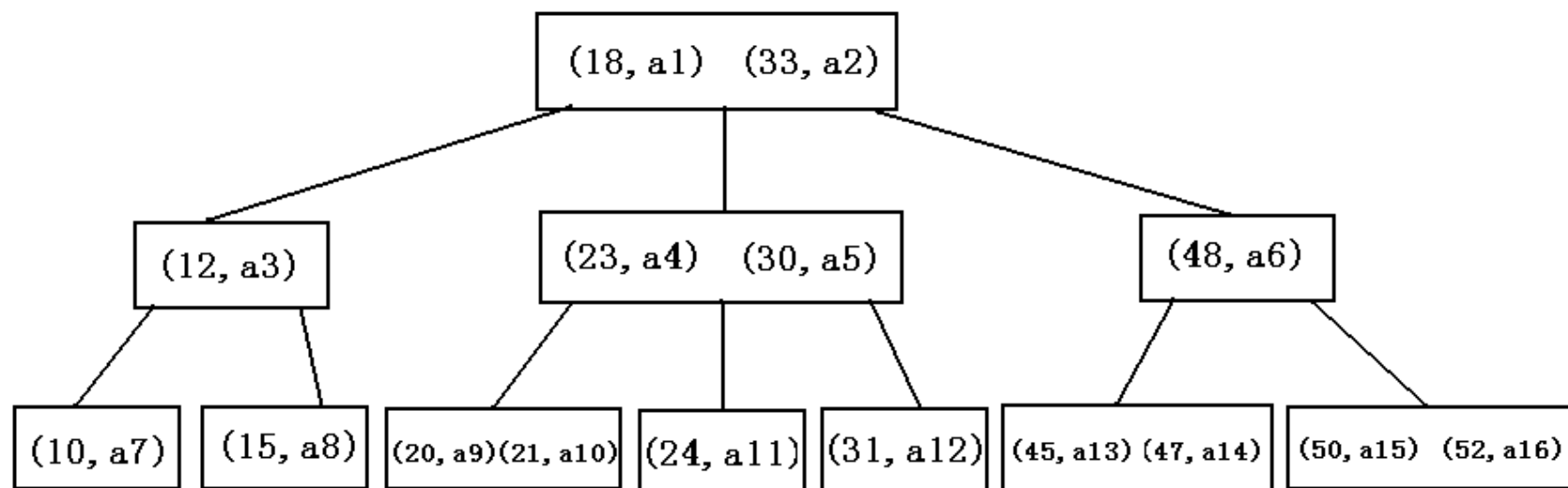
B 树的一个包含  $j$  个关键码,  $j+1$  个指针的结点的一般形式为:



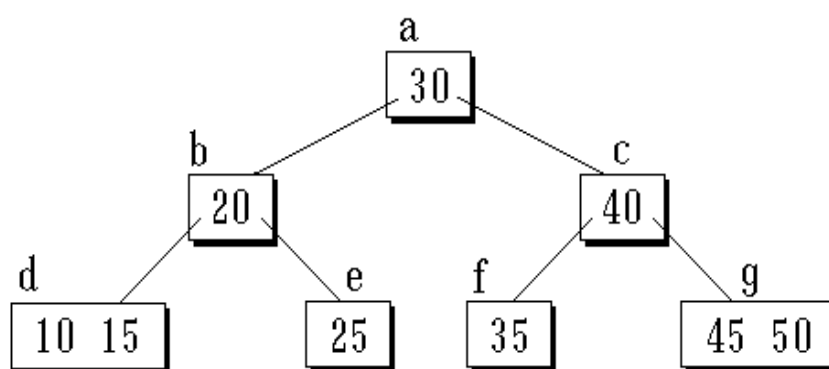
- 其中  $K_i$  是关键码值,  $K_1 < K_2 < \dots < K_j$ ,
- $P_i$  是指向包括  $K_i$  到  $K_{i+1}$  之间的关键码的子树的指针。

- 树高平衡、叶结点同层、关键码不重复
- 父结点关键码是子结点的分界
- 具有 $K$ 个子结点的结点包含 $k-1$ 个关键码
- 访问局部性原理
  - ➡ 值相近的记录放在相近的磁盘页中
- 节点关键码至少一定比例是满的
  - ➡ 改进空间利用率，减少检索、更新操作的I/O数

# 示例



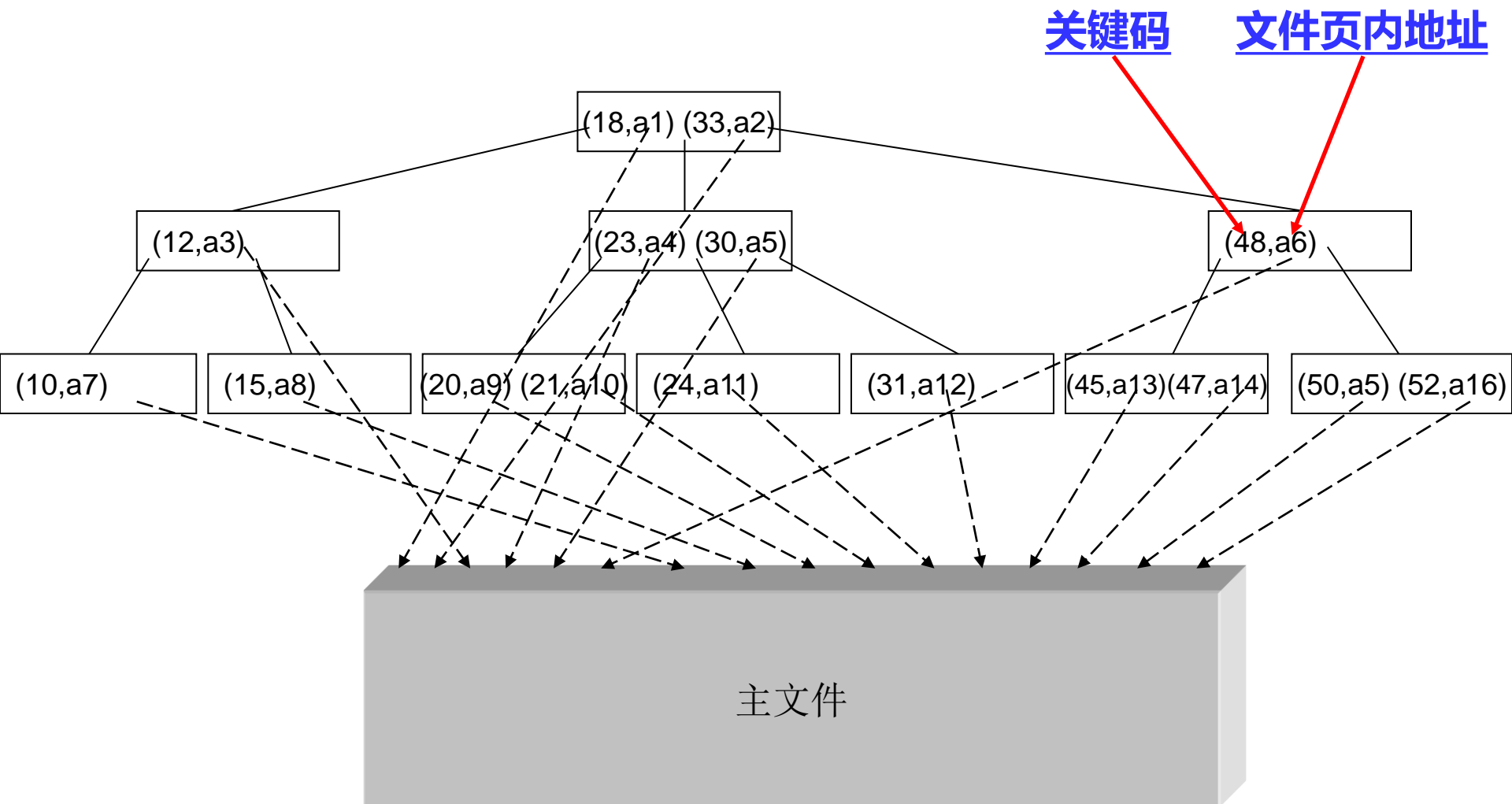
非B树



B树



# 每个关键码对应一个记录指针域



## ➤ 交替的两步过程

1. 读取根结点，在所包含的关键码 $K_1, \dots, K_j$ 中查找给定的关键码

✓ 找到则检索成功

2. 否则，确定要查的关键码值是在某个 $K_i$ 和 $K_{i+1}$ 之间，于是取 $p_i$ 所指向的结点继续查找

➤ 如果 $p_i$ 指向外部空结点，表示检索失败

- 设B树的高度为 $h$ 
  - ➡ 独根树高为1
- 在自顶向下检索到叶结点的过程中可能需要进行  $h$  次读盘
- 最多需要 $h+1$ 次访外

## ➤ 结构要调整，性质要保持（等高和阶）

- ➡ 1) 找到最底层插入
- ➡ 2) 若溢出，则结点分裂，中间关键码连同新指针插入父结点
- ➡ 3) 若父结点也溢出，则继续分裂
  - ✓ 分裂过程可能传达到根结点(则树升高一层)

- B树是从空树起，逐个插入关键码而生成的
- B树的每个内部结点的关键码个数都在 $[\lceil m/2 \rceil - 1, m-1]$  之间
- 插入在某个叶结点开始
  - ➡ 如果在关键码插入后结点中关键码数超出上界  $m-1$ ，则结点“分裂”
  - ➡ 否则可以直接插入
- 实现结点“分裂”的原则
  - ➡ 设结点  $A$  中已有  $m-1$  个关键码，当再插入一个关键码后结点的状态为

$$(m, A_0, K_1, A_1, K_2, A_2, \dots, K_m, A_m)$$

$$\text{其中 } K_i < K_{i+1}, 1 \leq i < m$$

- 分裂时从叶结点开始，往树根方向生长
- 结点p分裂成两个结点 p 和 q，它们包含的信息分别为：

➡ 结点 p

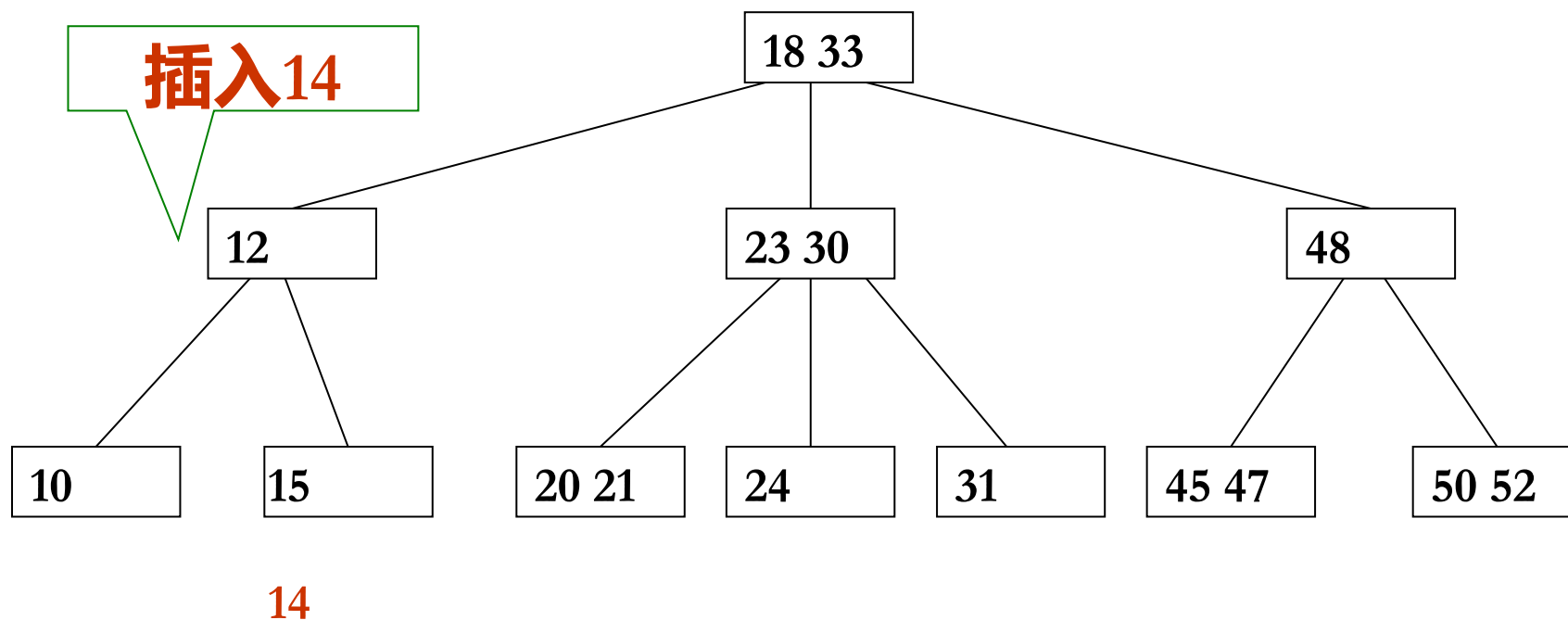
$$(\lceil m/2 \rceil - 1, A_0, K_1, A_1, \dots, K_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1})$$

➡ 结点 q

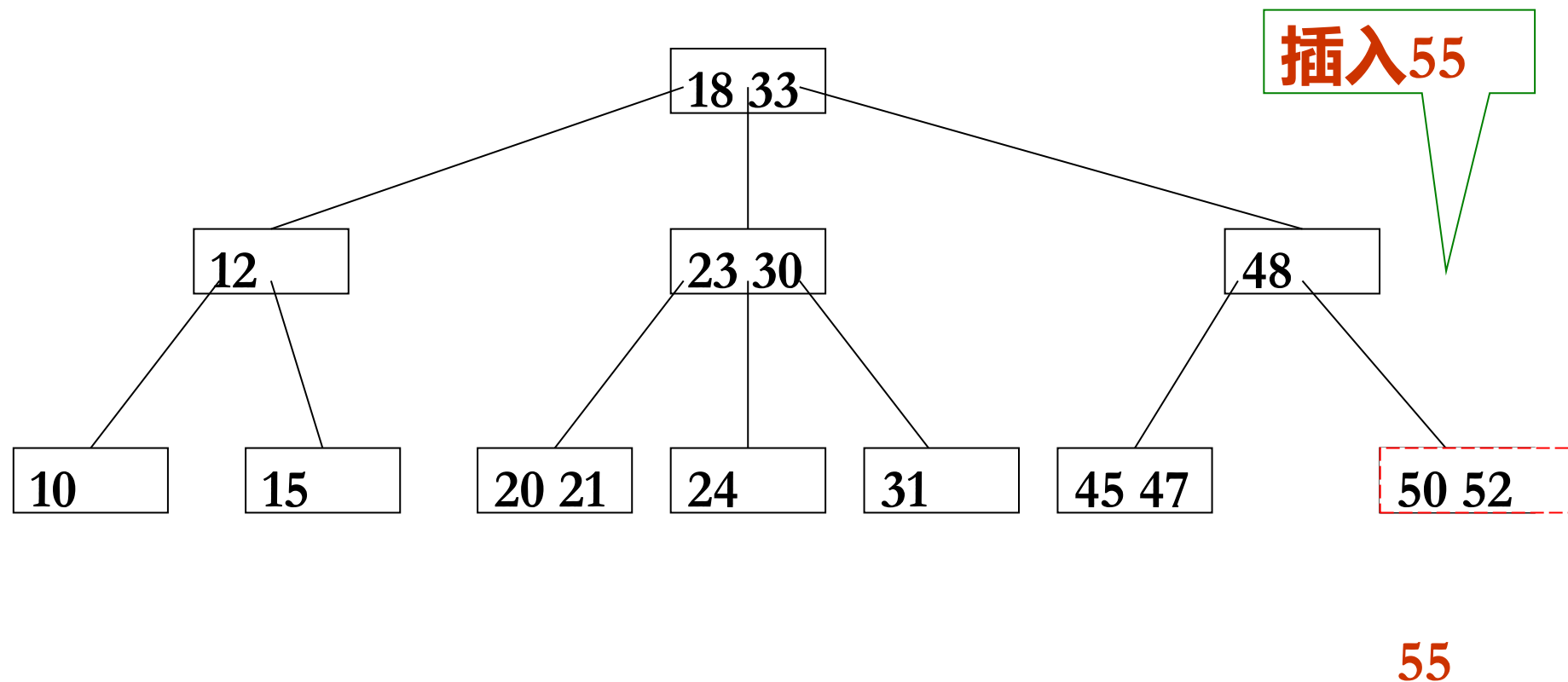
$$(m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, K_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}, \dots, K_m, A_m)$$

- 位于中间的关键码  $K_{\lceil m/2 \rceil}$  与指向新结点 q 的指针形成一个二元组  $(K_{\lceil m/2 \rceil}, q)$ ，插入到这两个结点的双亲结点中去

## ➤ 3阶B树

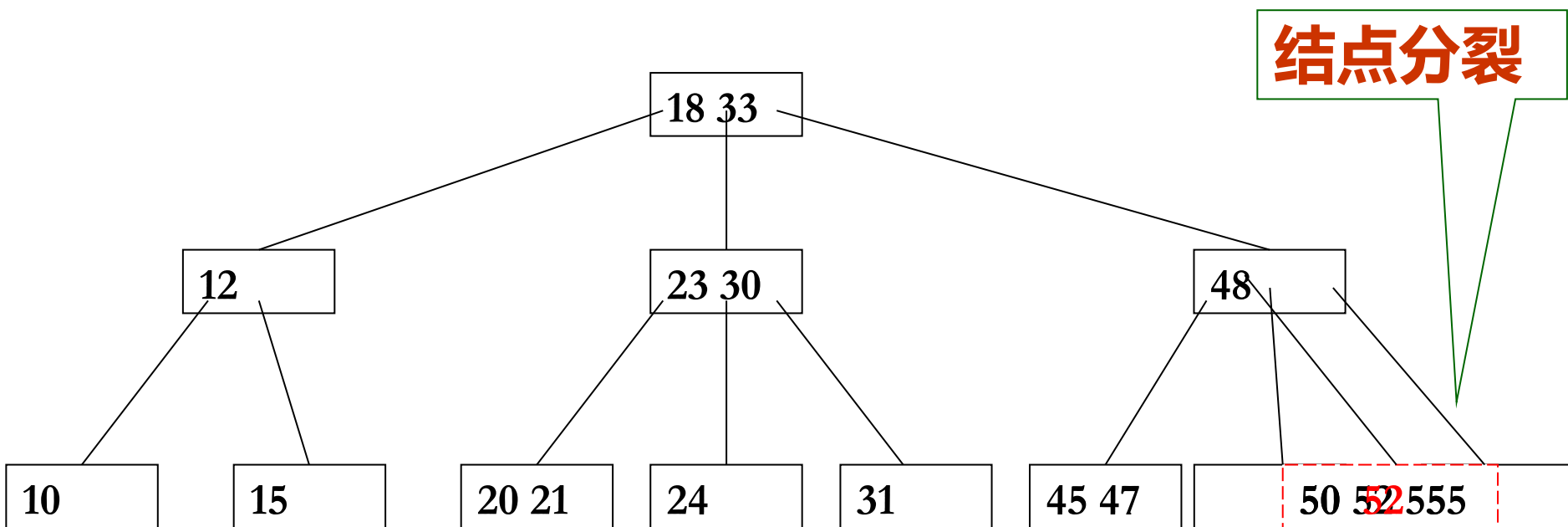


# m=3, 插入55

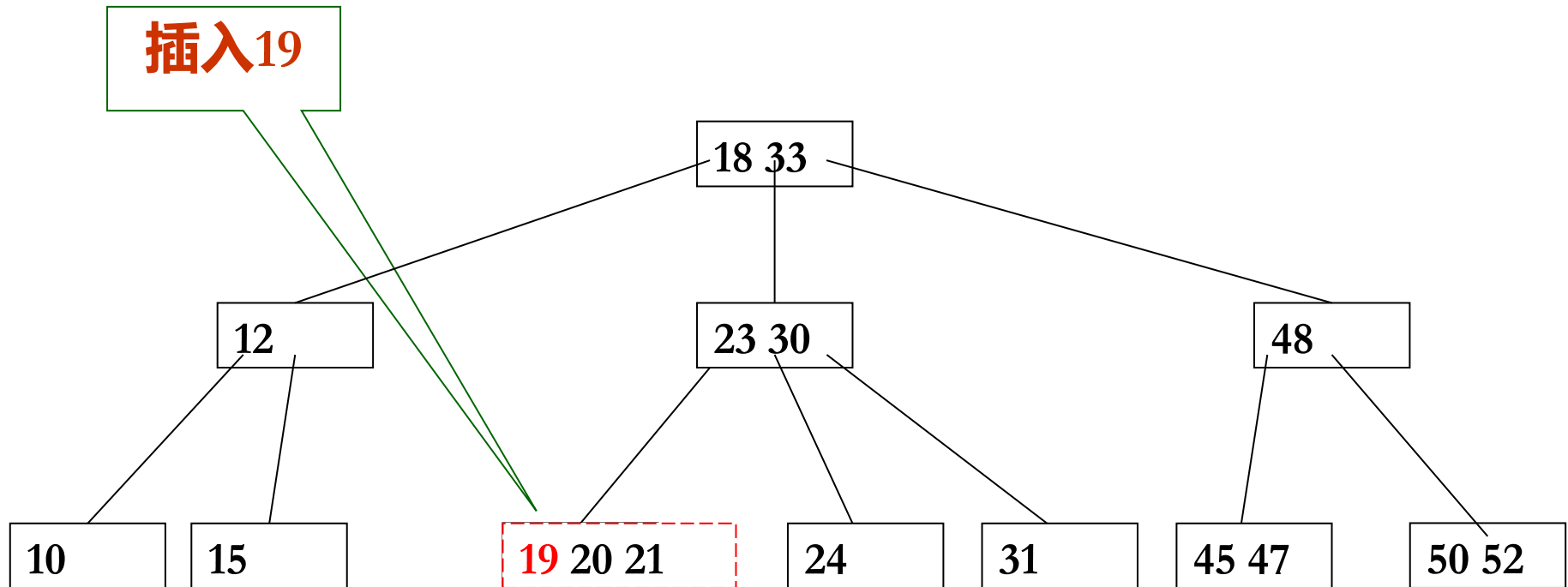




# $m=3$ , 叶结点分裂, 把52提升到父结点



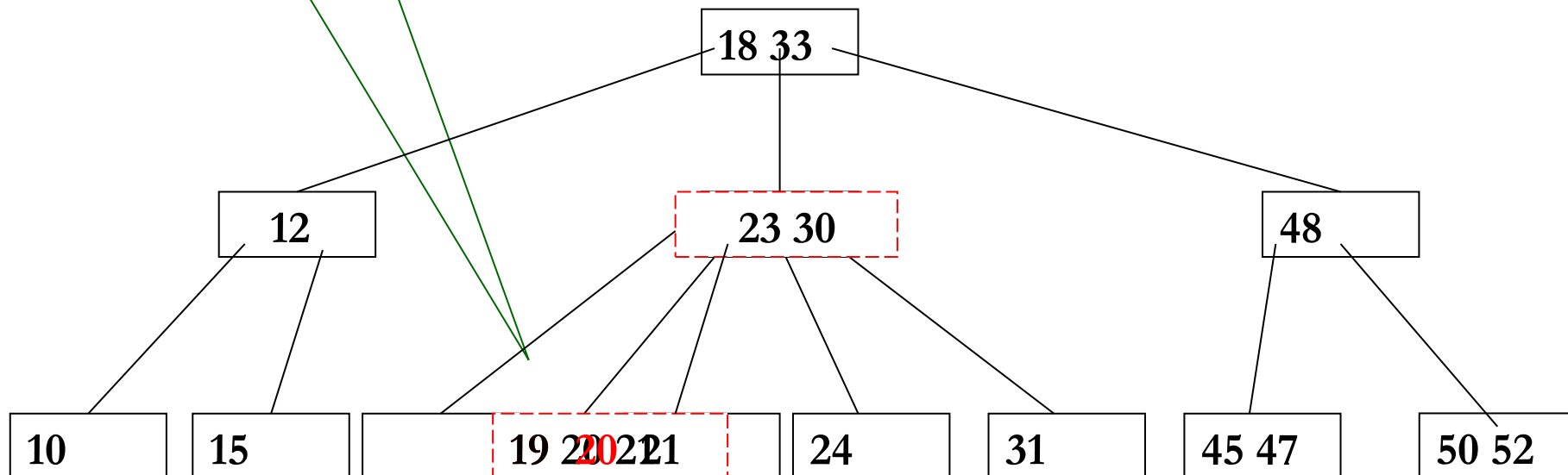
# 插入引起3阶B树根结点分裂的例子



$m=3$



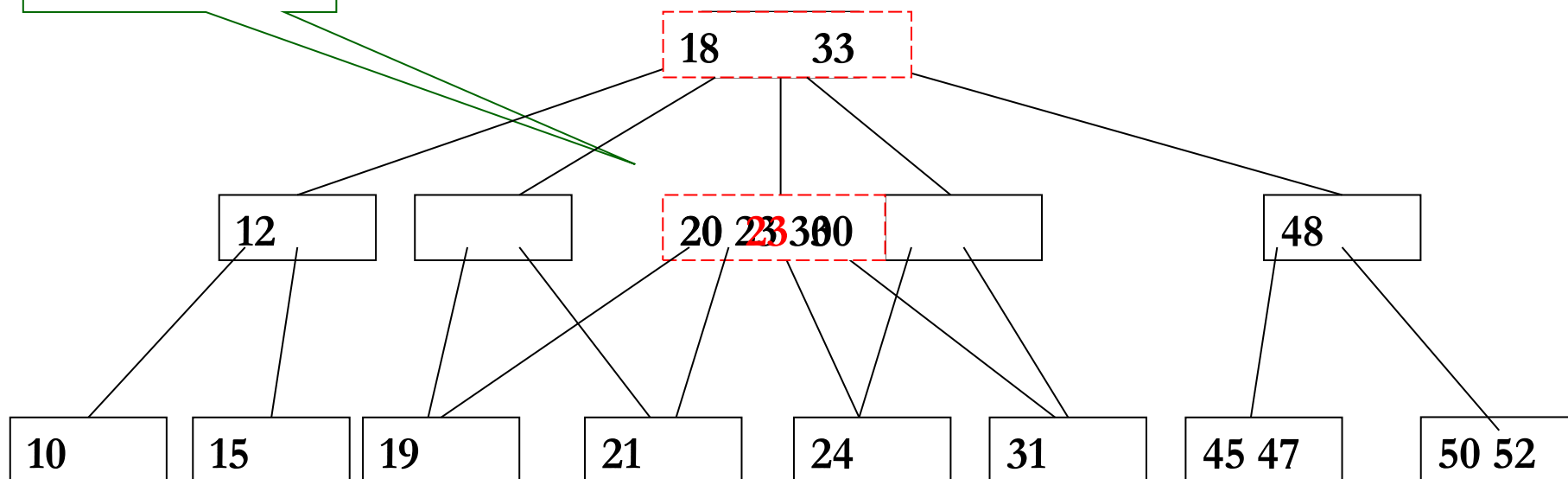
## 叶结点分裂



# m=3



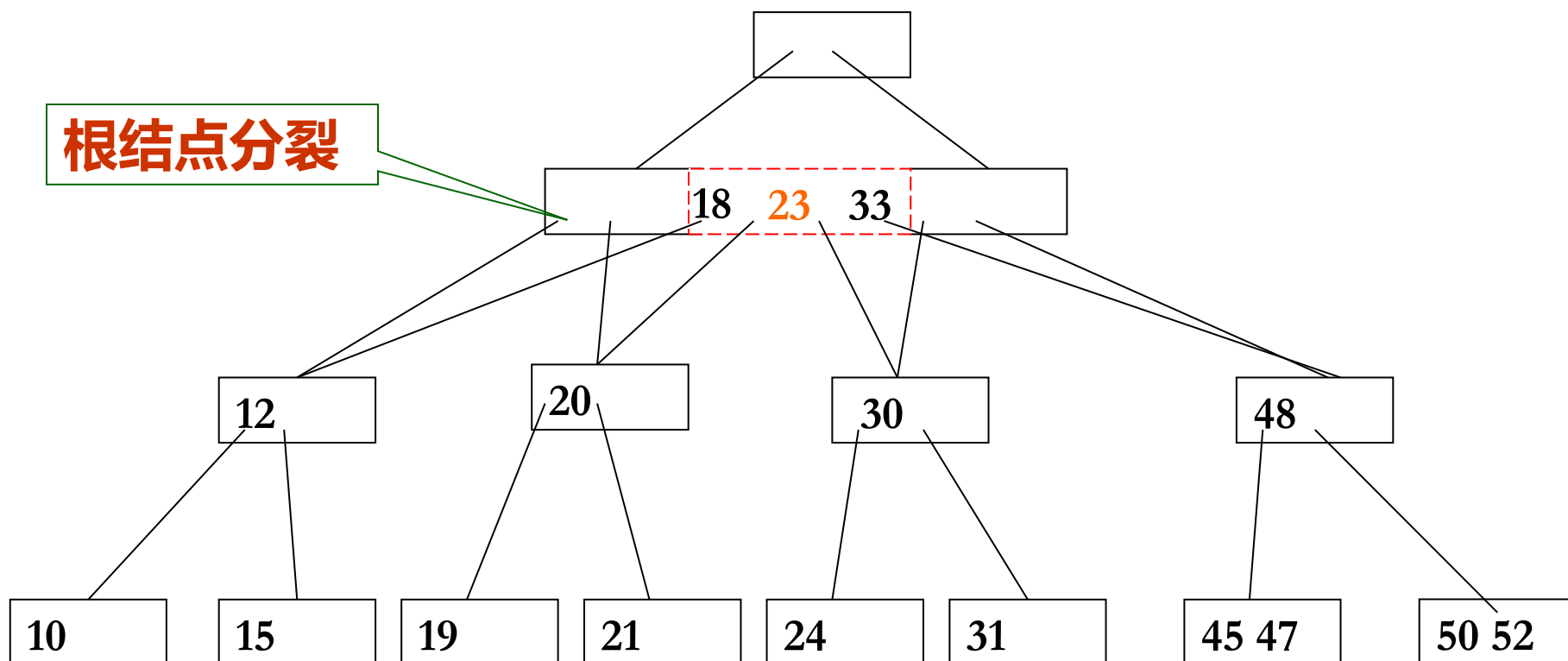
第二层结点  
分裂



$m=3$



根结点分裂



- **默认约定：**内存足够大，检索时读入的结点，在向上分裂时不必再从磁盘读入
- 读盘次数与查找相同
- 最少写盘次数：一次
  - 如不分裂，则仅将插入关键码所在结点写到外存
- 如考虑对主数据文件访问，则再加一次

## ➤ 删除的关键码不在叶结点层

- ➡ 先把此关键码与它在B树里的后继对换位置  
，然后再删除该关键码

## ➤ 删除的关键码在叶结点层

➡ 删除后关键码个数不小于 $\lceil m/2 \rceil - 1$

✓ 直接删除

➡ 删除后关键码个数小于 $\lceil m/2 \rceil - 1$

✓ 如果兄弟结点关键码个数大于 $\lceil m/2 \rceil - 1$

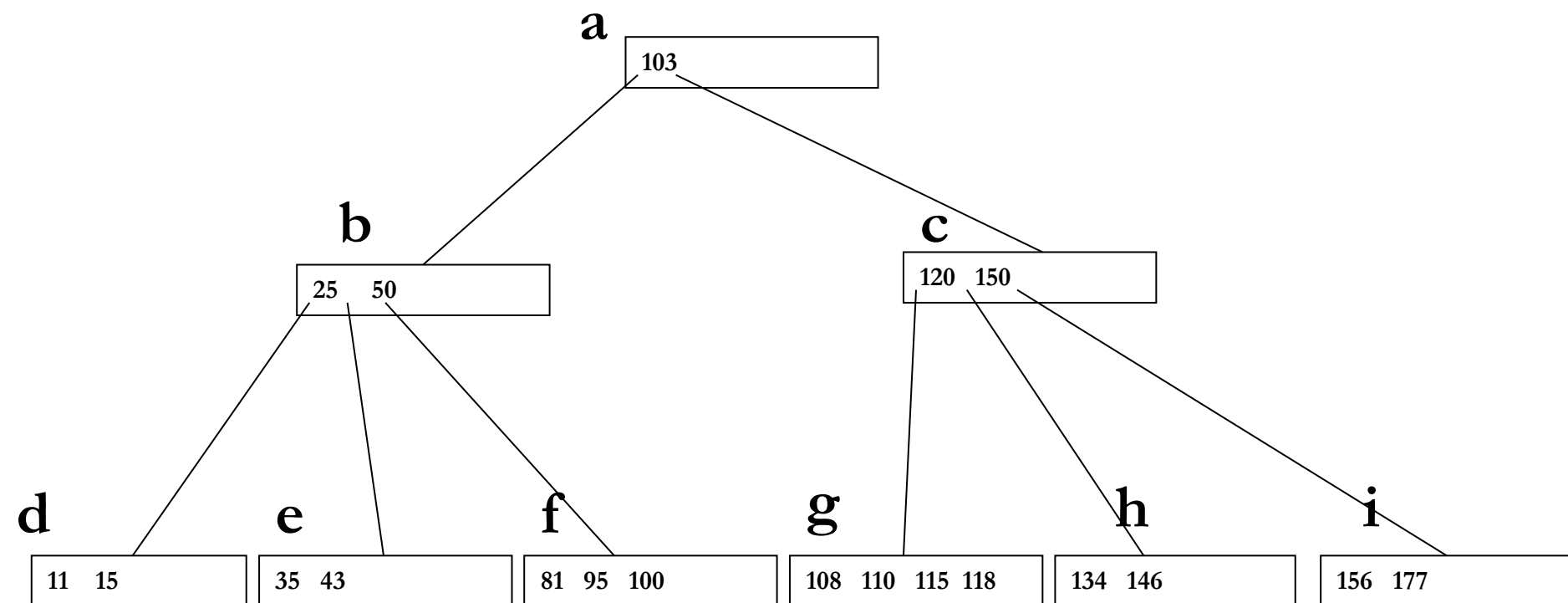
– “借”若干关键码（父结点中的分界关键码要做调整）

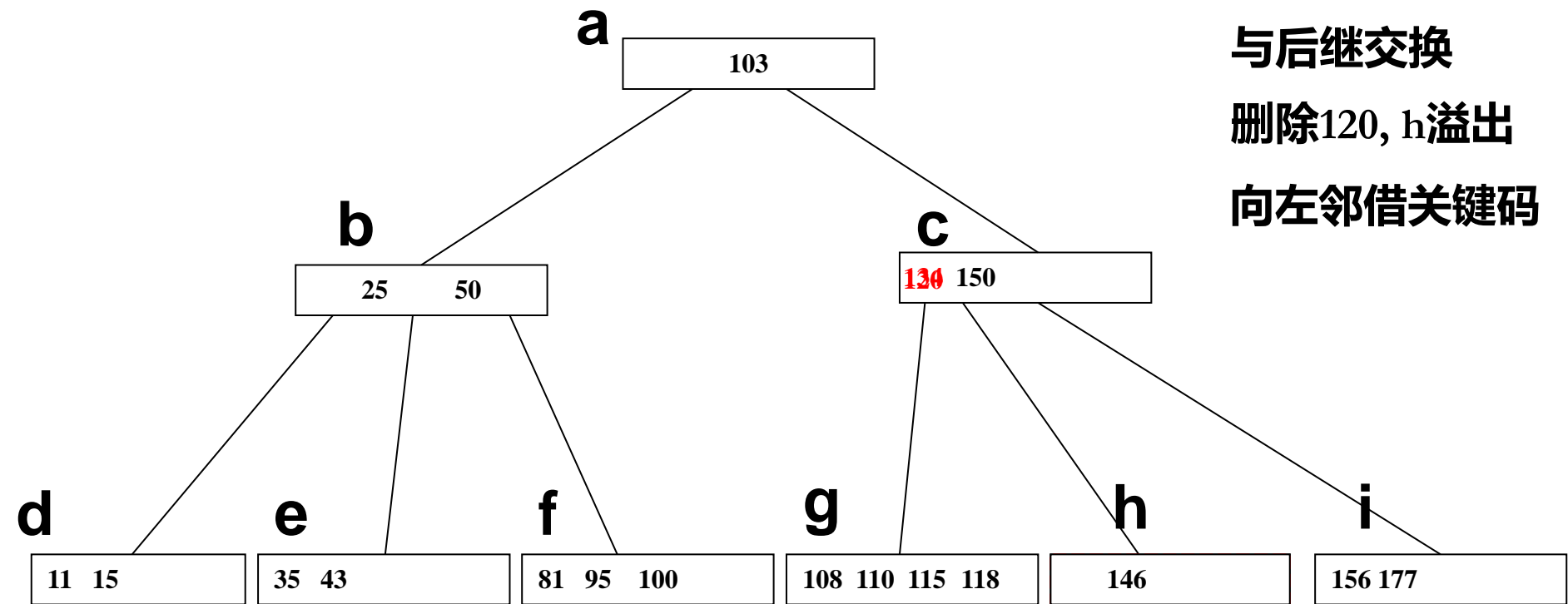
✓ 如果兄弟结点关键码个数等于 $\lceil m/2 \rceil - 1$

– 合并



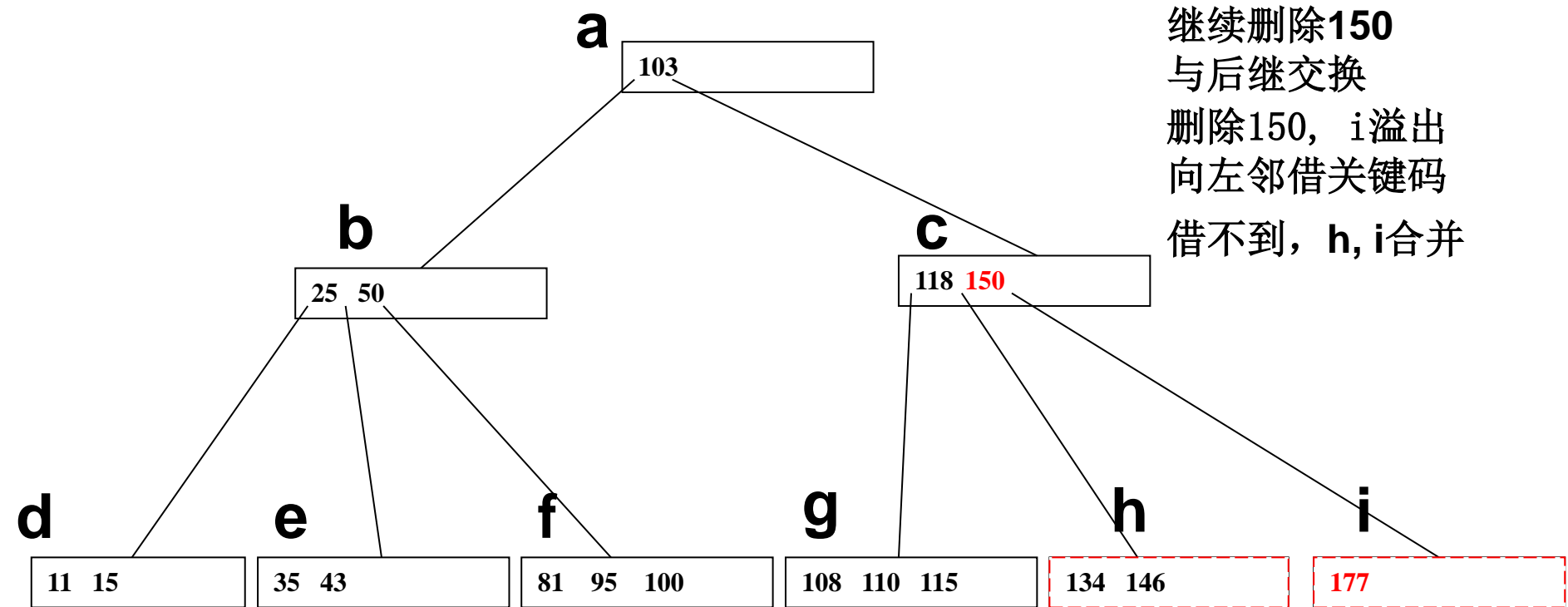
# 5阶B树删除示例

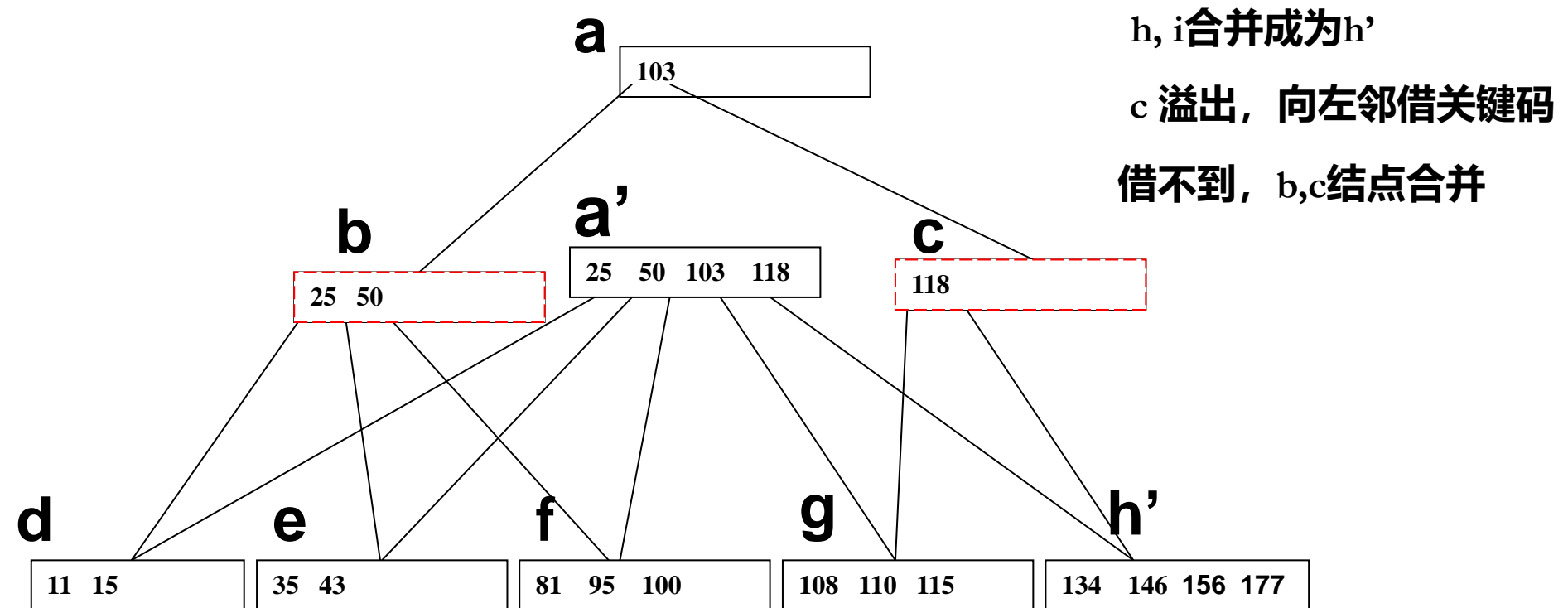




与后继交换  
删除120, h溢出  
向左邻借关键码

删除120，先与后继交换，删后下溢出，向左邻借关键码





➤ **B树的一种变形**

➤ **在叶结点上存储信息的树**

➡ **所有关键码均在叶结点**

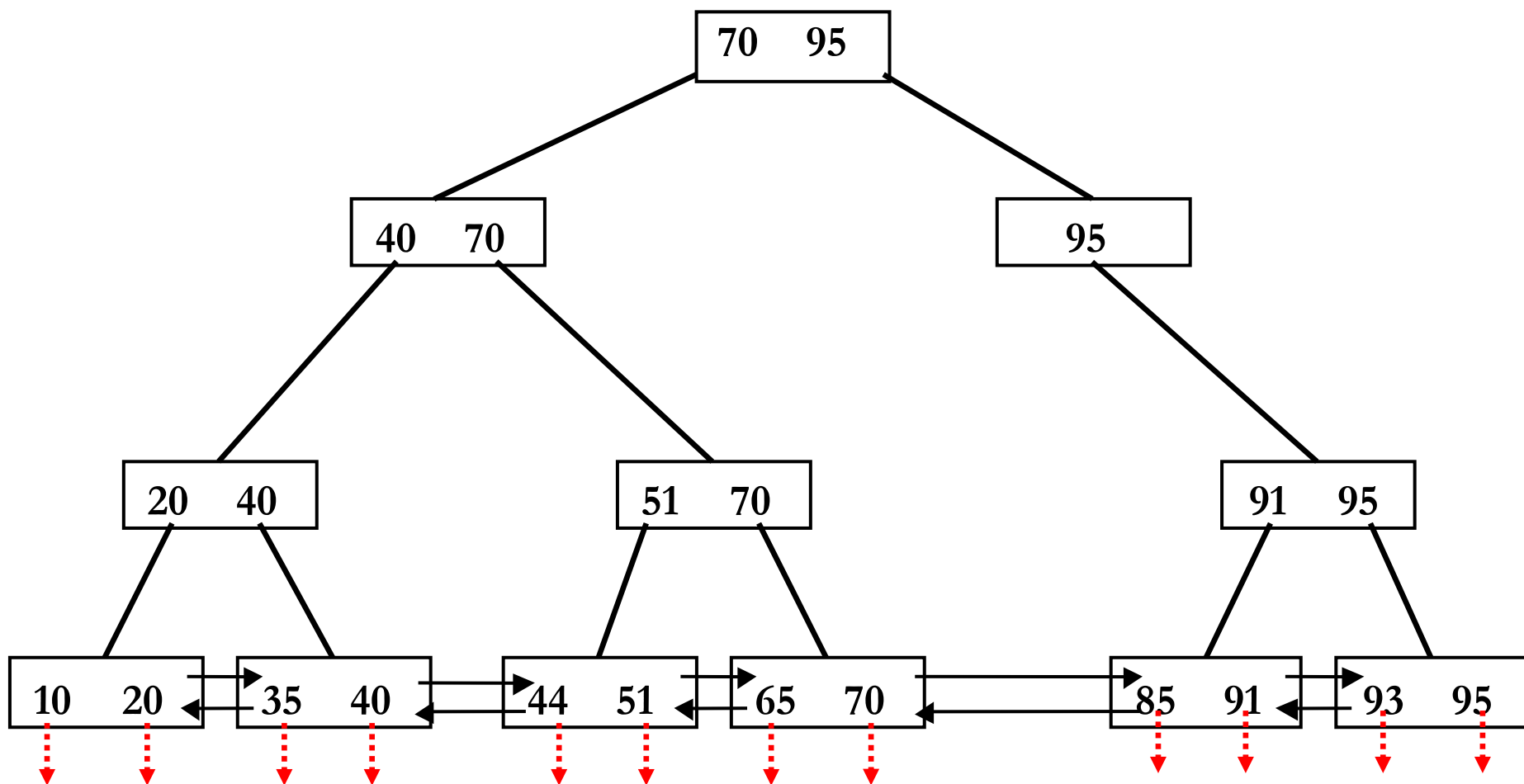
➡ **各层结点中的关键码均是下一层相应结点  
中最大关键码（或最小关键码）的复写**

# B<sup>+</sup>树的结构定义 (m阶)



- 每个结点至少有  $\lceil m/2 \rceil$  个子结点 (根除外) ,  
至多有m个子结点
- 根结点至少有两个子结点
  - ➡ 根为空, 或者独根情况除外
- 有k个子结点的结点必有k个关键码。

# 2阶B+树的例子



# B<sup>+</sup>树和B树的差异



- B<sup>+</sup>树中 $n$ 棵子树的结点中含有 $n$ 个关键码；而B树中 $n$ 棵子树的结点中含有 $n-1$ 个关键码
- B<sup>+</sup>树叶子结点包含了完整的索引的信息，而B树所有结点共同构成全部索引信息
- B<sup>+</sup>树所有的非叶结点可以看成是高层索引，结点中仅含有其子树中最大（或最小）关键码



## ➤ 查找到叶结点层

- ➡ 在上层已找到待查的关键码，并不停止
- ➡ 继续沿指针向下查到叶结点层的这个关键码

## ➤ B<sup>+</sup>树叶结点一般链接起来，形成一个双链表

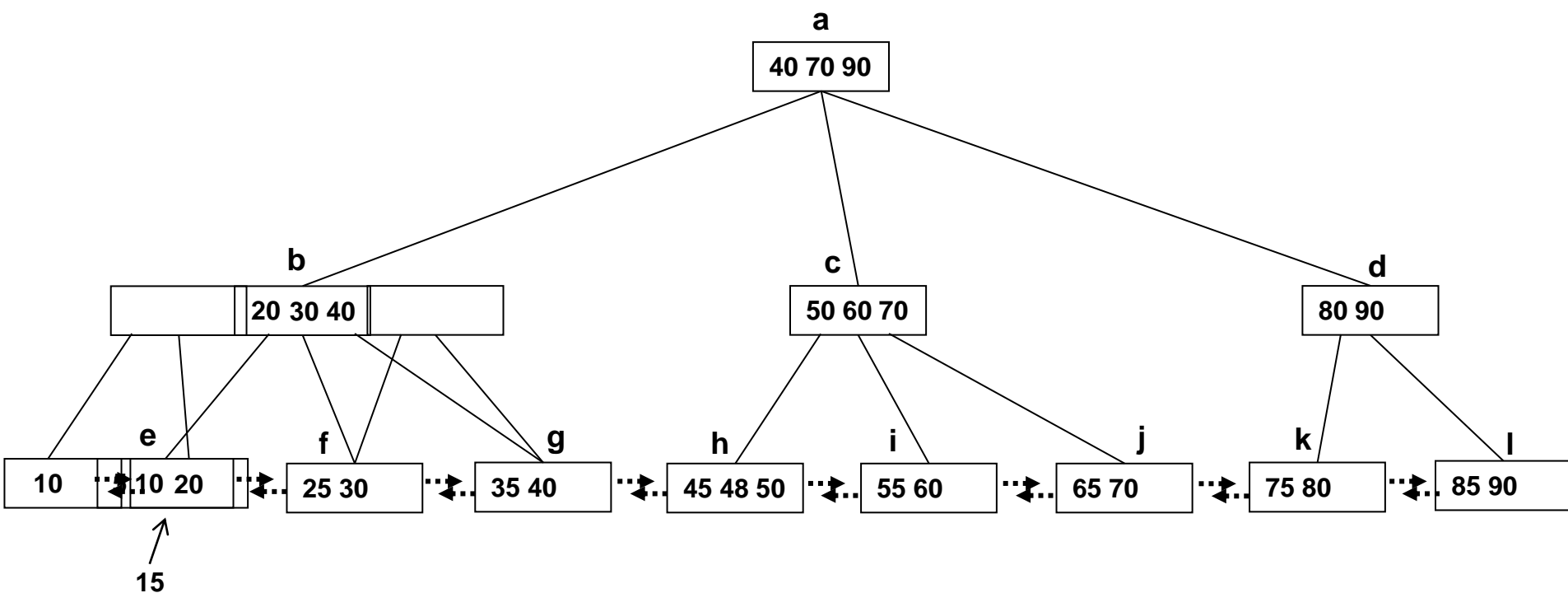
- ➡ 适合顺序检索（范围检索）
- ➡ 实际应用更广

**需要的话，每一层结点也可以顺序链接**

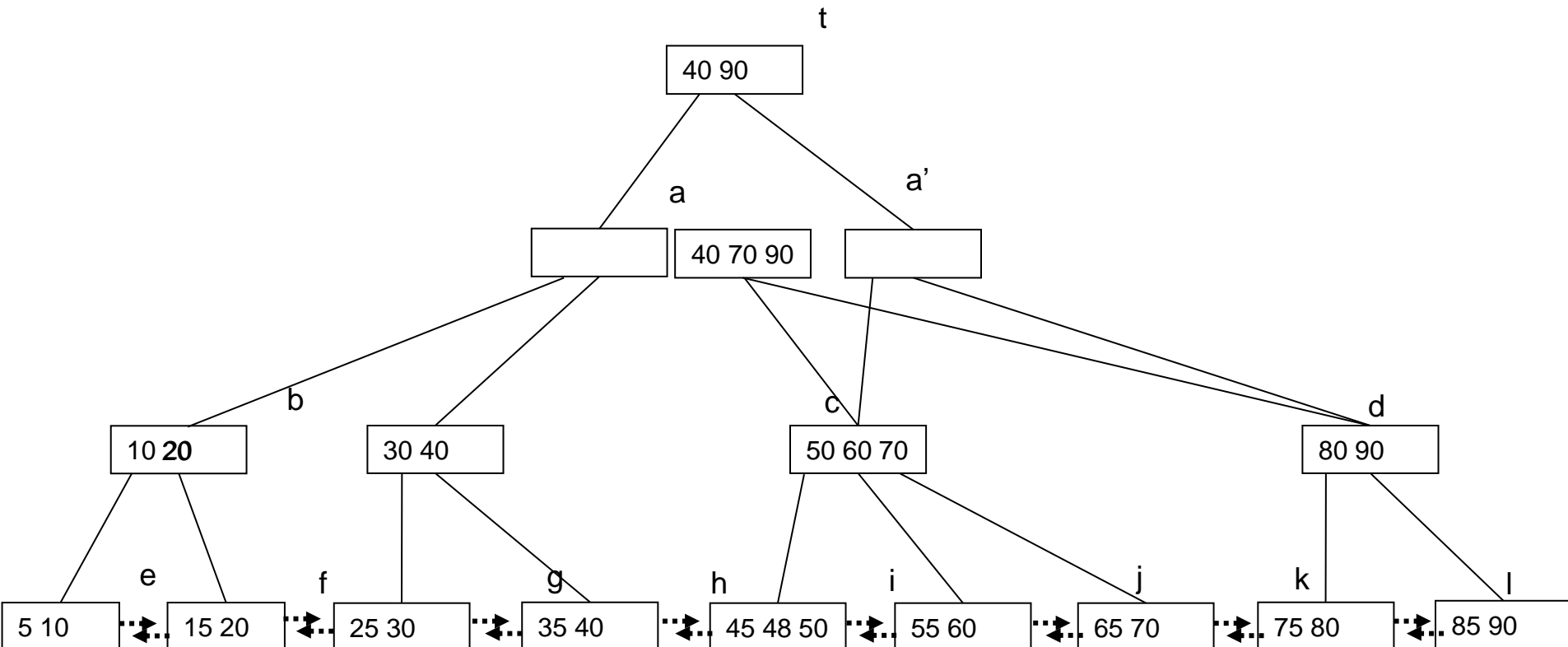
## ➤ 插入——分裂

- ➡ **过程和B树类似**，分裂为左  $\lceil m/2 \rceil$  和右  $\lfloor m/2 \rfloor$
- ➡ **注意保证上一层结点中有这两个结点的最大关键码（或最小关键码）**

# 3阶B+树

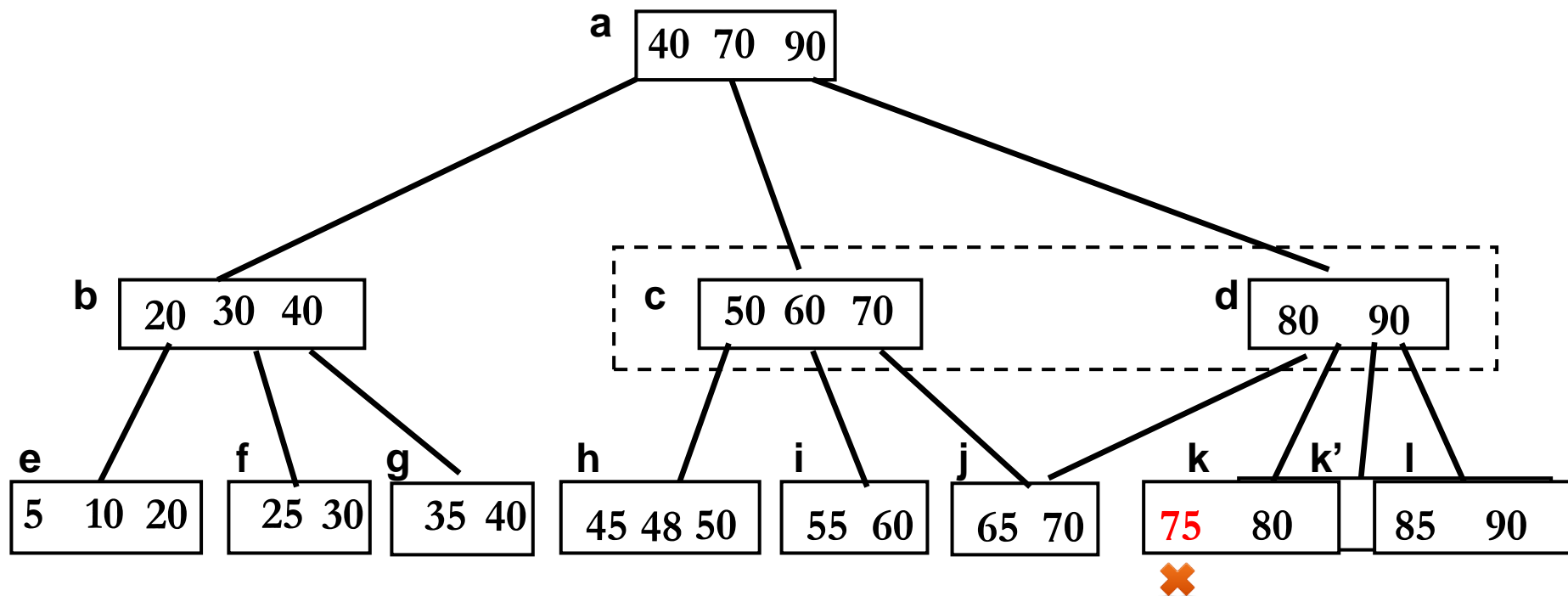


# 3阶B+树中插入15后，树高+1

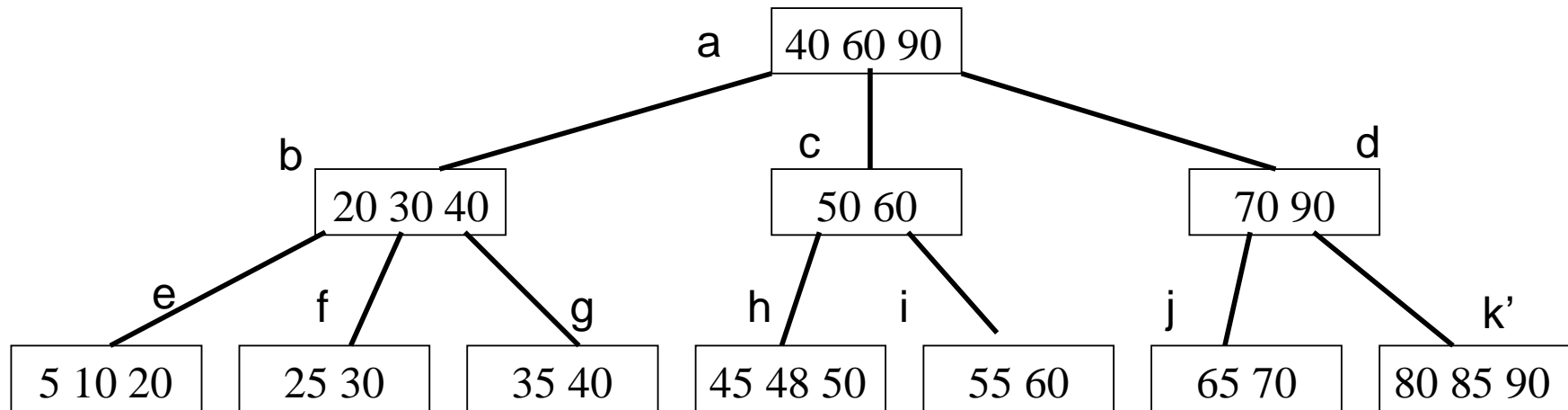


- 当删除结点后，关键码个数越下界  $\lceil m/2 \rceil$ ，与左右兄弟进行调整、合并处理，与B树类似
- 关键码在叶结点层删除后，其在上层的复本**可以保留**，做为一个“分界关键码”存在
  - ➡ 也可以替换为新的最大关键码（或最小关键码）

# 在3阶B+树中删除结点75



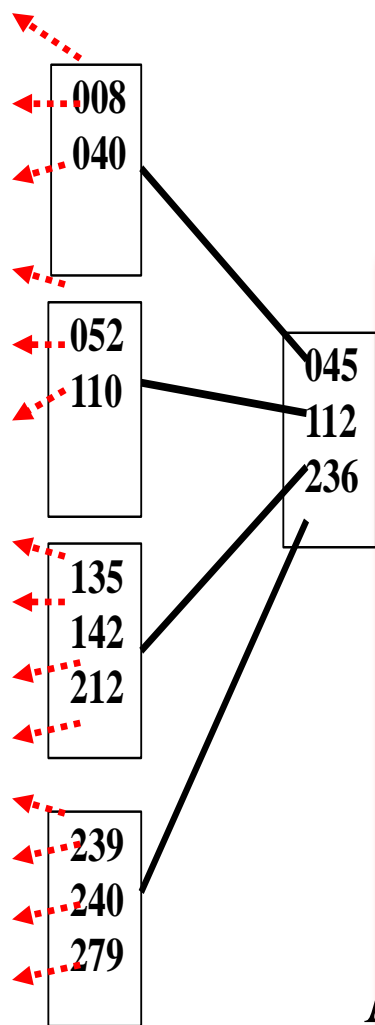
- 沿a、d、k查找，找到叶结点
- 在k中删去75，发生下溢出，剩余关键码80与右邻l结点合并为新k' (80, 85, 90)
- 父结点d中原分界码80删除
  - d结点下溢出
  - 借左邻c的关键码，c和d的关键码平分
  - 父结点a中的分界码70修改为60



# B树的性能分析

## ➤ 包含N个关键码的B树

➡ 有 $N+1$ 个外部空指针



假设空指针数为 $l$ , 节点数为 $n$ , 关键码个数为 $N$ 。

则边的总数为:  $l+n-1$

假设 $n$ 个节点关键码个数分别为 $x_1+x_2+\dots+x_n=N$ , 由于每个节点的射出的边数比关键码个数多1, 则总的边数为

$(x_1+1)+(x_2+1)+\dots+(x_n+1)$ , 合并 $n$ 个1, 则有

$$x_1+x_2+\dots+x_n+n=N+n$$

联立方程:  $l+n-1=n+N$ , 消掉 $n$ , 则有 $l=N+1$ , 即有 $N+1$ 个外部空指针。

李振科。



# B树的性能分析

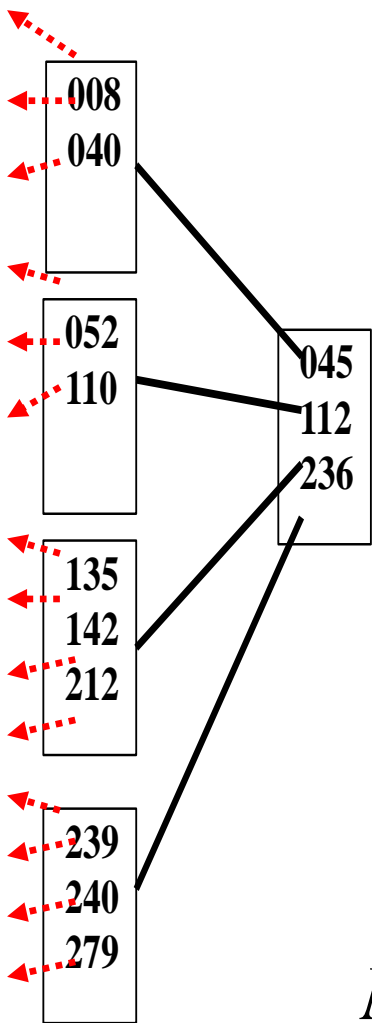
## ➤ 包含N个关键码的B树

- 有 $N+1$ 个外部空指针
- 假设外部指针在第 $k$ 层

## ➤ 各层的结点数目

- 第0层根至少1个结点，第1层至少2个结点
- 第2层至少  $2 \cdot \lceil m/2 \rceil$  个结点
- 第 $k$ 层至少  $2 \cdot \lceil m/2 \rceil^{k-1}$  个结点

$$N + 1 \geq 2 \cdot \lceil m/2 \rceil^{k-1}, \quad k \leq 1 + \log_{\lceil m/2 \rceil} \left( \frac{N+1}{2} \right)$$



## ➤ 检索效率

### ➡ 存取次数

$$k \leq 1 + \log_{\left\lfloor \frac{m}{2} \right\rfloor} \left( \frac{N+1}{2} \right)$$

➤  $N=1,999,998$ ,  $m=199$ 时, 一次检索最多4层。

➤ 设关键码数为 $N$ ，内部结点数为 $p$

➤ 当根包含1个关键码时，除根外的所有内部结点都包含 $\lceil m/2 \rceil - 1$ 时，B树包含的关键码的个数最少

$$N \geq 1 + (\lceil m/2 \rceil - 1)(p - 1)$$

即

$$p - 1 \leq \frac{N - 1}{\lceil m/2 \rceil - 1}$$

➤ 除第1个结点外，剩余的 $p-1$ 个结点都是分裂而来的，则每插入一个关键码平均分裂结点个数为

$$s = \frac{p - 1}{N - 1} \leq \frac{N - 1}{(\lceil m/2 \rceil - 1) \cdot (N - 1)} = \frac{1}{\lceil m/2 \rceil - 1}$$

# 要求能够根据B树的特点进行分析



➤ 在一棵含有1023个关键字的4阶B树中进行查找（不读取数据主文件），至多读盘\_\_\_\_次。

A. 7

B. 8

C. 10

D. 9

$$k \leq 1 + \log_{\left\lfloor \frac{m}{2} \right\rfloor} \left( \frac{N+1}{2} \right)$$

**至少读盘次数该怎么算？**

# 11.5 红黑树 (Red-Black tree)

## ➤ 11.5.1 定义

## ➤ 11.5.2 性质

## ➤ 11.5.3 结点插入算法

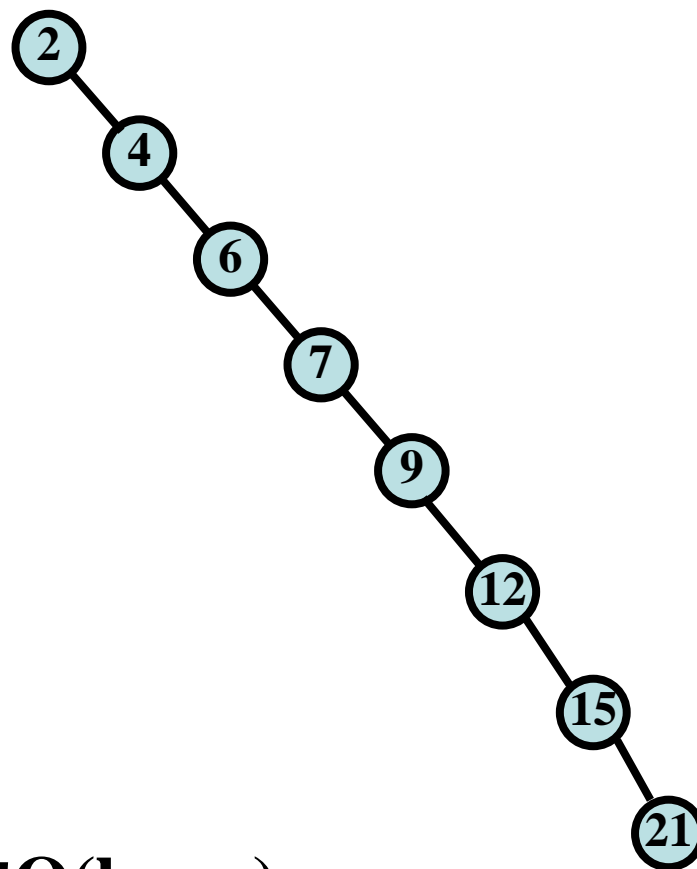
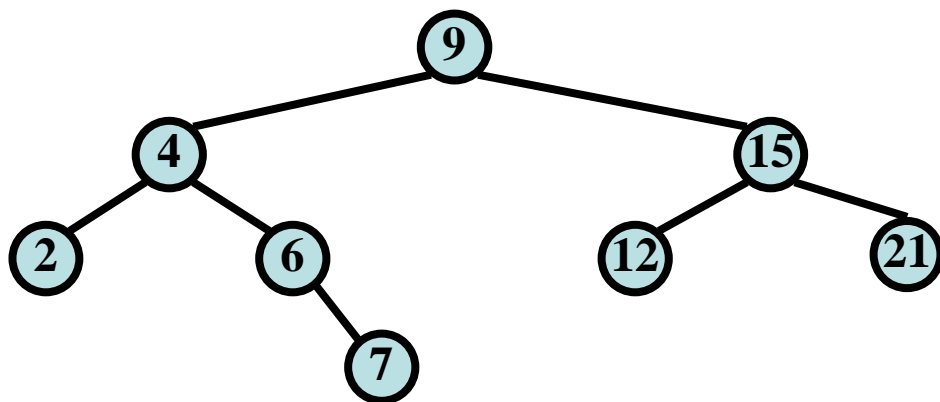
## ➤ 11.5.4 结点删除算法

# BST树的平衡问题



➤ 输入：9,4,2,6,7,15,12,21

➤ 输入：2,4, 6,7, 9, 12,15, 21



➤ 希望保持理想状况

➤ 插入、删除、查找时间代价为 $O(\log n)$

## ➤ 满足下列条件的**BST树** (要满足BST的约束条件)

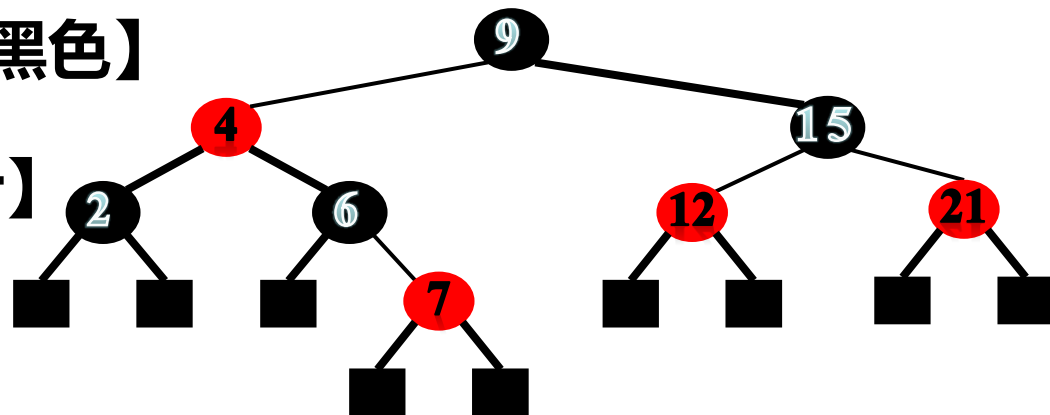
1. 颜色两态: 【红色 or 黑色】

2. 树根为黑: 【根和树叶】

3. 树叶也为黑

4. 红红限制: 父子节点不允许红红连续

5. 路径上黑结点数目相同: 任意结点到我每个叶结点包含相同数目的黑结点。



是一种扩充的BST树

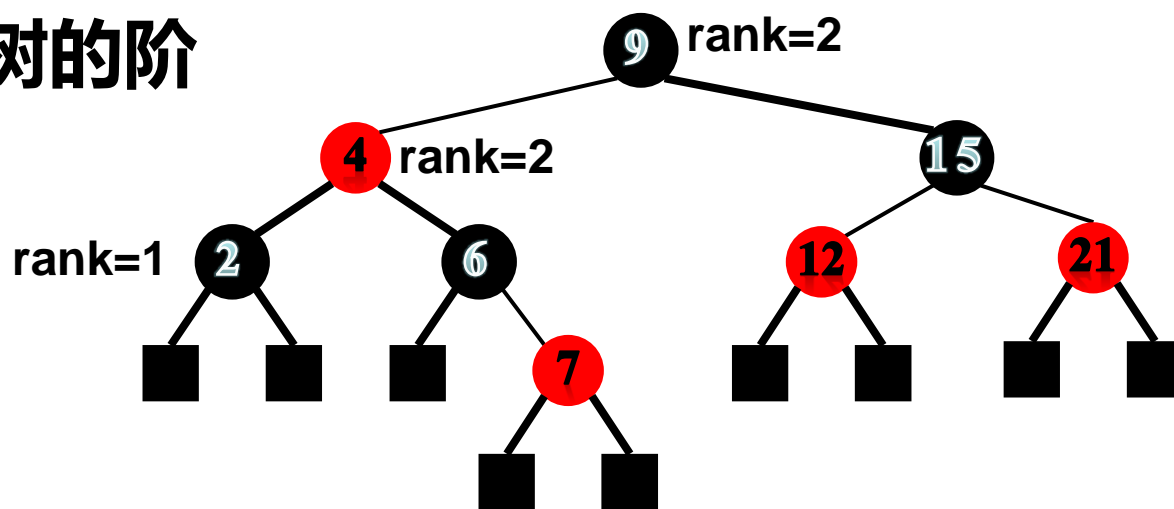
## ➤ 结点X的阶 (Rank, 也称 “黑色高度” )

➡ 从该结点到外部结点的黑色结点数量

➡ 不包括X结点本身, 包括叶结点

## ➤ 叶结点的阶是0

## ➤ 根的阶称为该树的阶



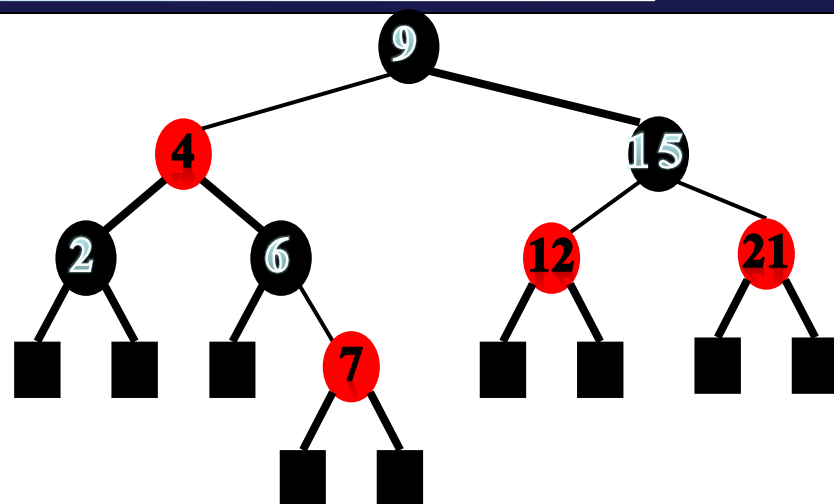


# 11.6.2 红黑树的性质



性质1：红黑树是满二叉树

➡ 空树叶也看作结点



性质2：k阶红黑树路径长度

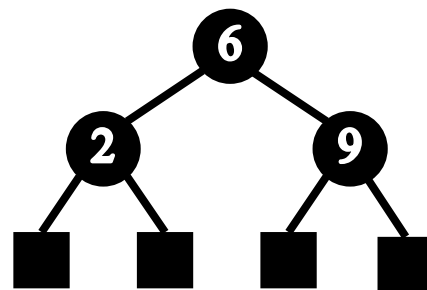
➡ 从根到叶的简单路径长度，即介于  $[k, 2k]$

➡ 或者说树高介于  $[k+1, 2k+1]$  之间

性质3：k阶红黑树内部结点数

➡ 最少时是一棵完全满二叉树

➡ 内部结点数最少是  $2^k - 1$



**性质4:  $n$ 个内部结点红黑树最大高度:  $2 \cdot \log_2 (n+1) + 1$**

证明:

设红黑树的阶为 $k$ , 高为 $h$ 。

由性质 (2) 得  $h \leq 2k + 1$

✓ 则  $k \geq (h-1) / 2$

由性质 (3) 得  $n \geq 2^k - 1$

✓ 即  $n \geq 2^{(h-1)/2} - 1$

可得出  $h \leq 2 \log_2 (n+1) + 1$

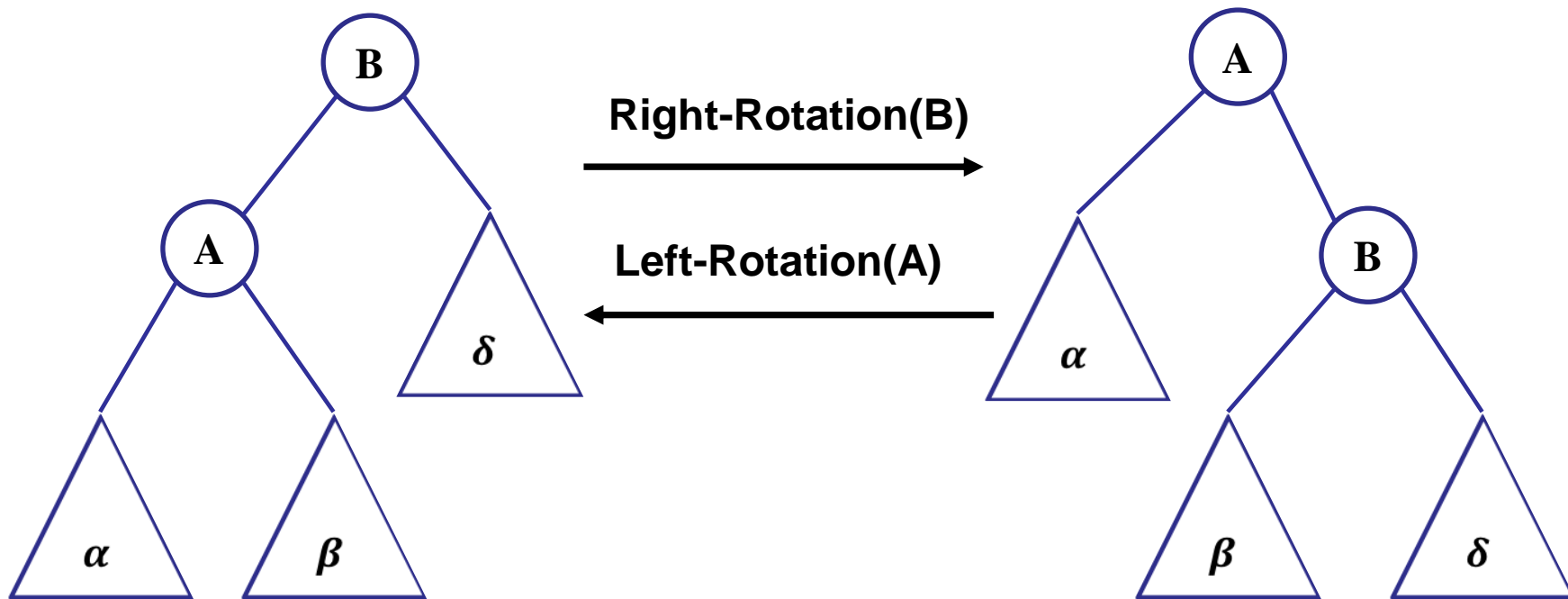
★推论：红黑树下列的搜索操作的复杂度为 $O(\log n)$ ， $n$ 为结点的数目。

操作：Search(key)/Min/Max/Successor/  
Predecessor

# 红黑树---基本操作



基本操作：旋转



LR/RR 都保留了二叉搜索树的特性

$$\forall a \in \alpha, b \in \beta, c \in \delta$$

$$a < A < b < B < c$$

# 红黑树---插入



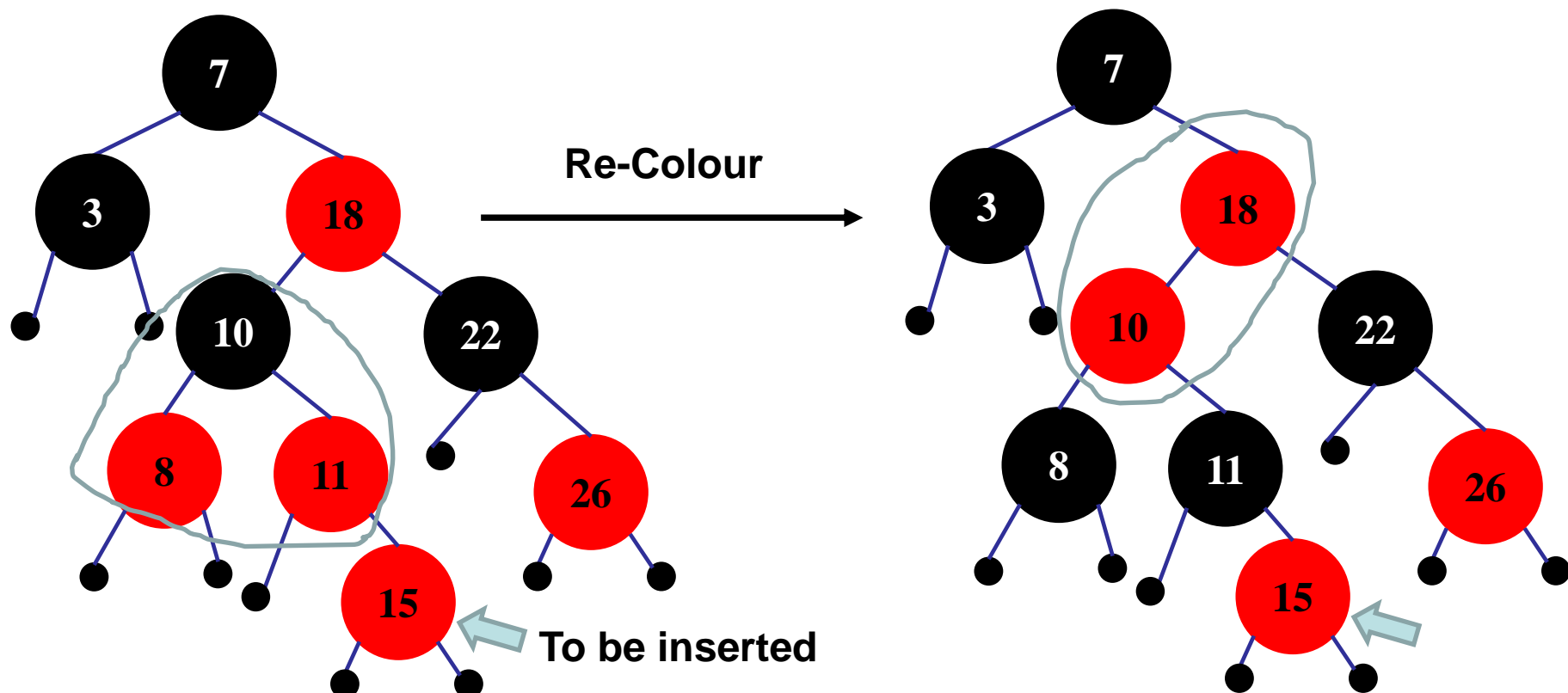
## RBTree-Insert(x)

基本思路: ----- **Tree\_Insert(X)** // 利用普通二叉搜索树的方法

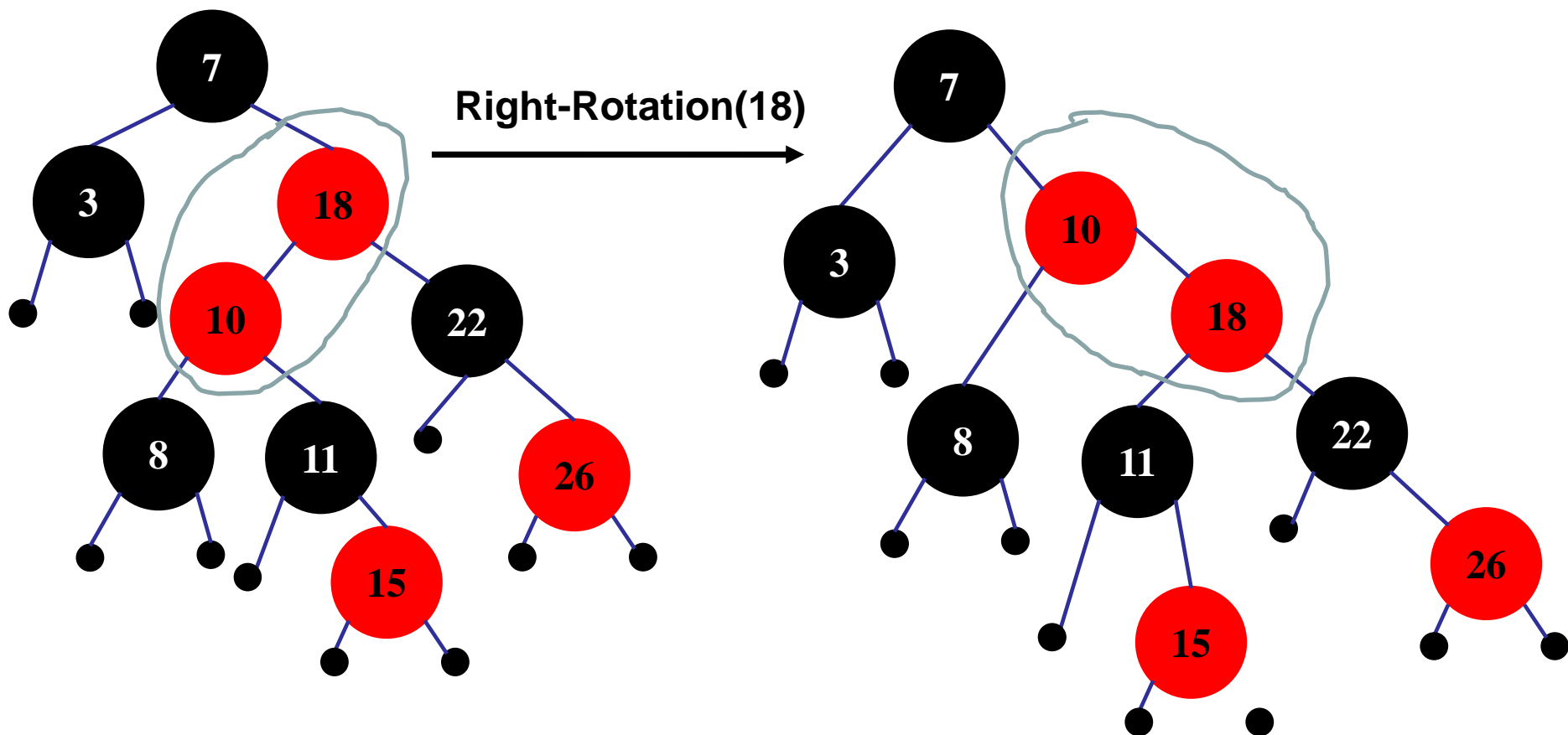
----- 将插入的X设置为“红色”

带来的问题: X的父亲也可能为红色, 造成红-红冲突  
(违反了红黑树的第4条性质)

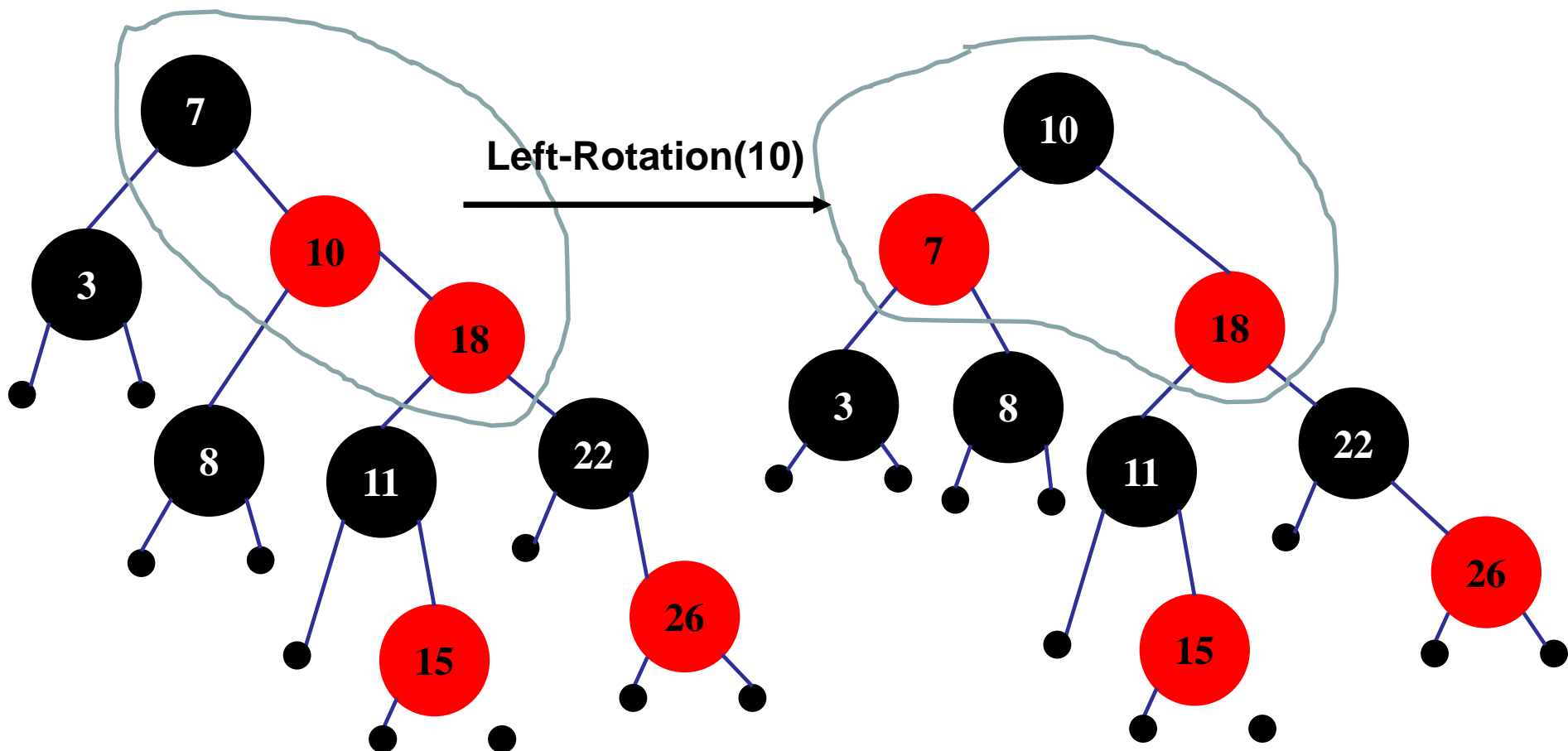
----- 通过Rotation和Re-colour来调整



# 红黑树---插入



# 红黑树---插入



# 红黑树---插入



**RB\_Insert(T,X)**

**Tree\_Insert(T,X) // BST 插入算法**

**Colour(x) ← RED**

**While  $X \neq \text{root} \wedge \text{Colour}[X] == \text{RED} \wedge \text{Colour}[P[X]] == \text{RED}$**

**Do If  $P[X] == \text{left}[P[P[x]]]$  // (CASE-A)**

**then  $y \leftarrow \text{Right}[P[P[x]]]$**

**if  $\text{Colour}[y] == \text{RED}$**

**then <Case 1>**

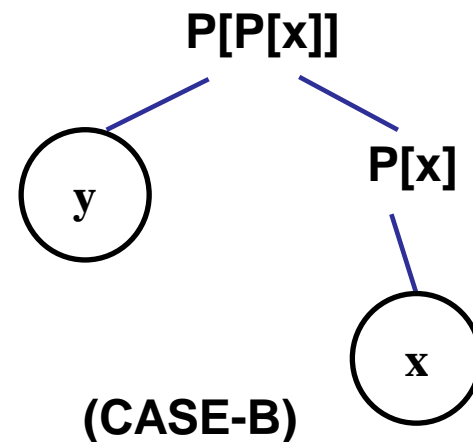
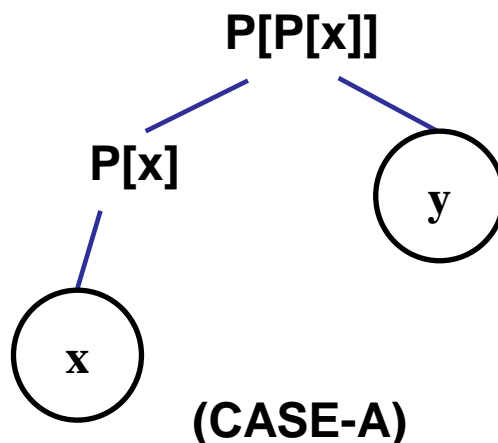
**else if  $X == \text{right}[P[X]]$**

**then <Case 2>**

**else <Case 3>**

**Else // (CASE-B)**

**Colour[root] ← BLACK**



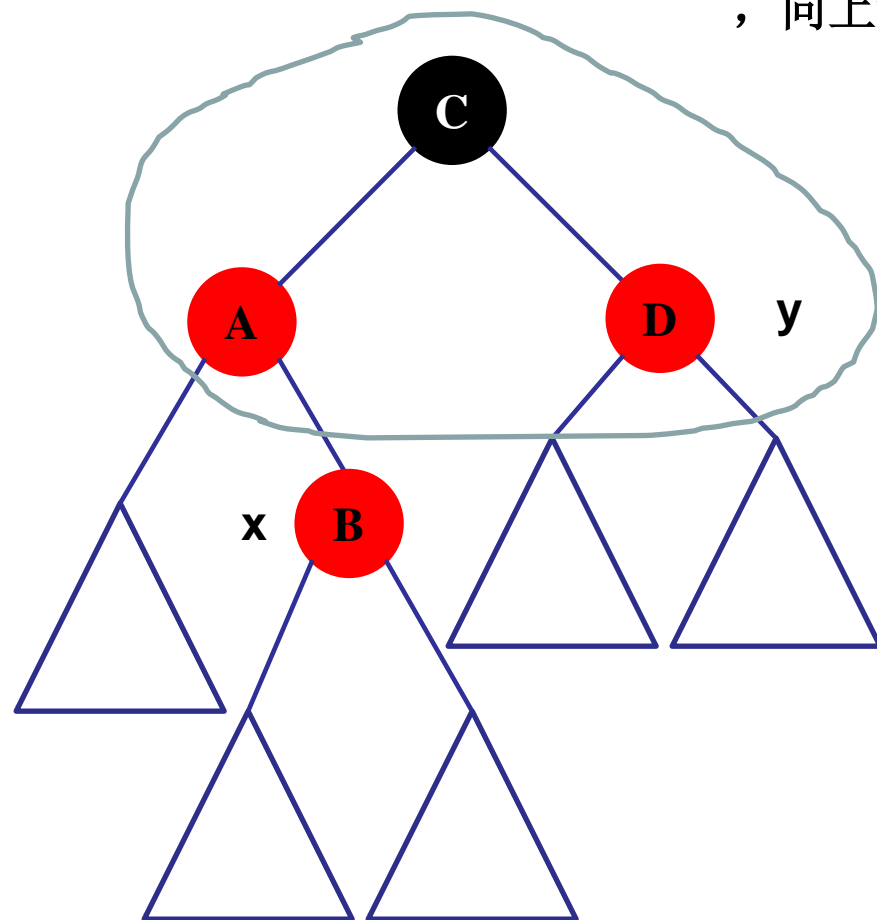


# 红黑树---插入

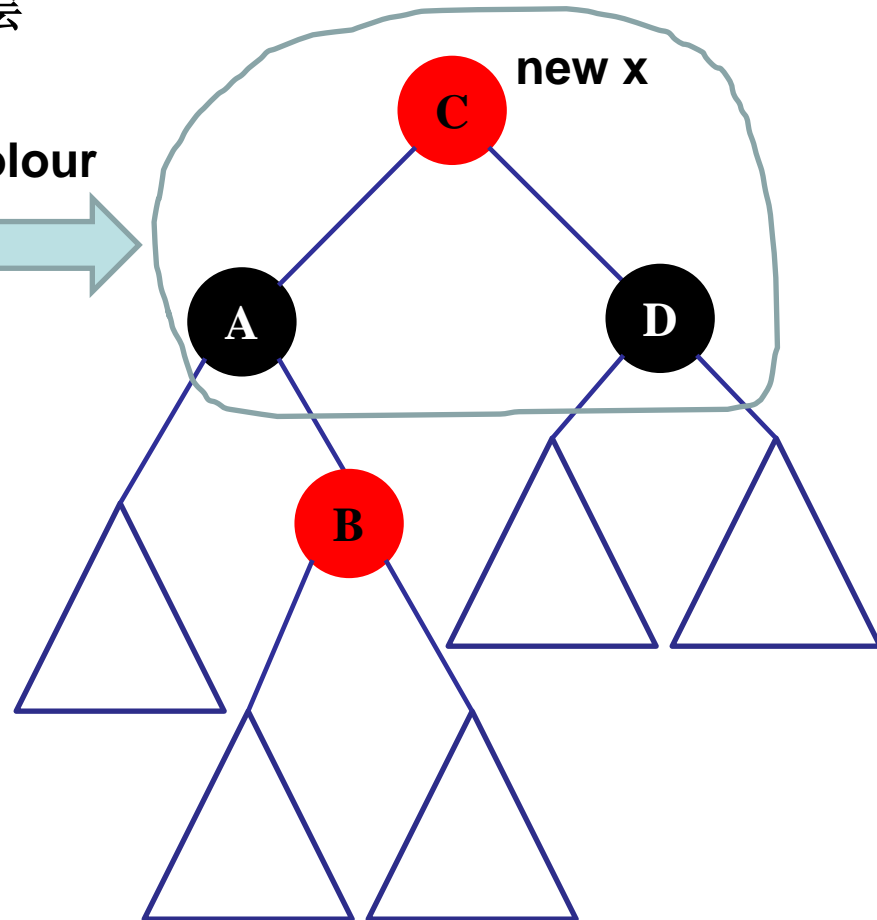


(CASE-1)

c变红，如果c不是root  
，向上迭代2层



Re-colour



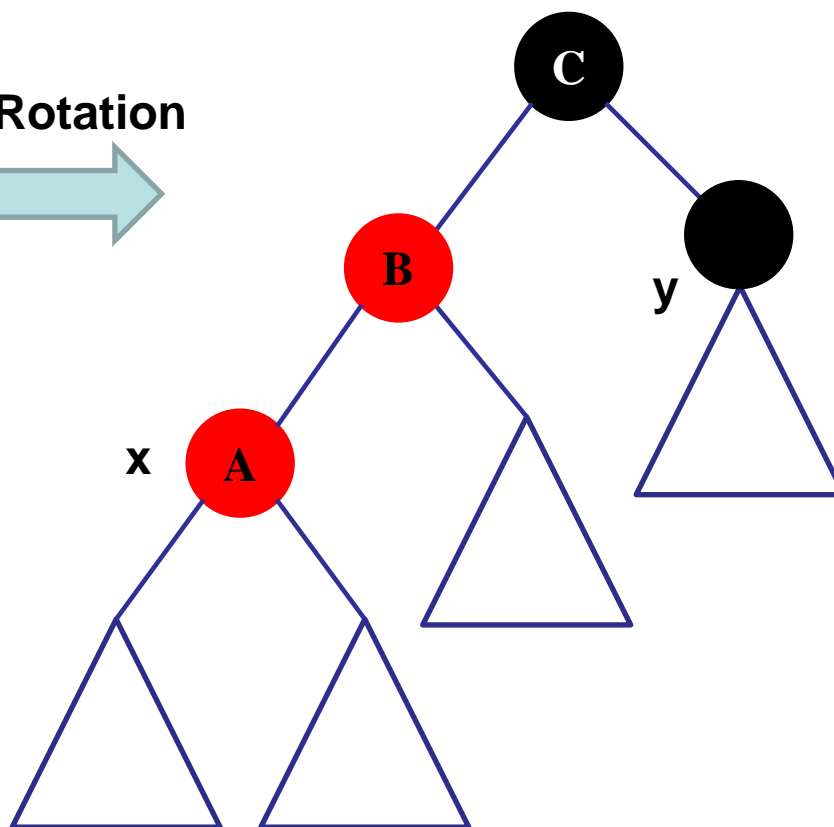
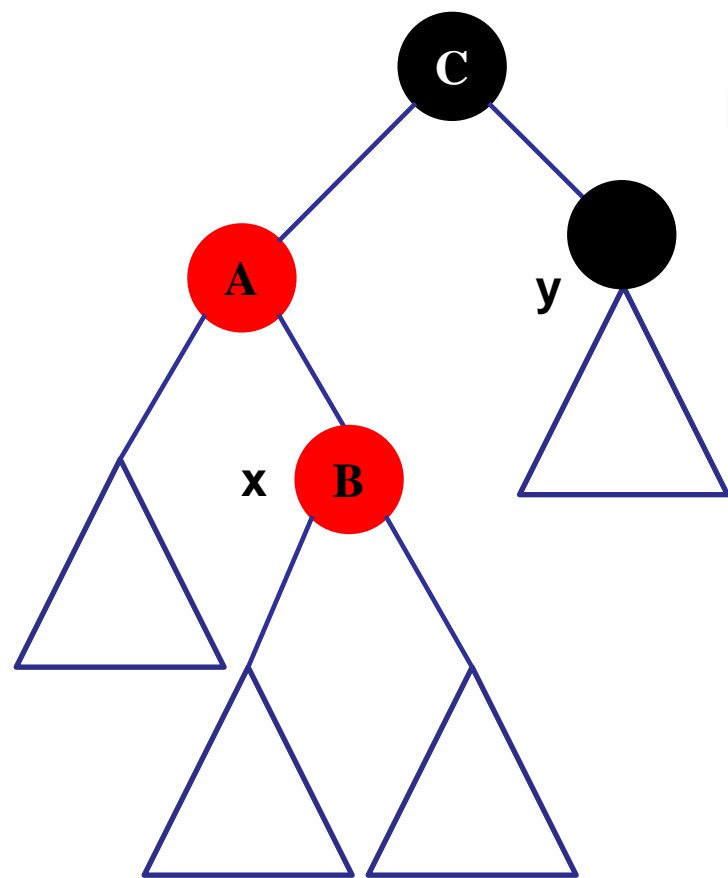
# 红黑树---插入



(CASE-2)

$x$ 的叔父是黑， $x$ 是 $P[x]$ 的右儿子；

Left-Rotation

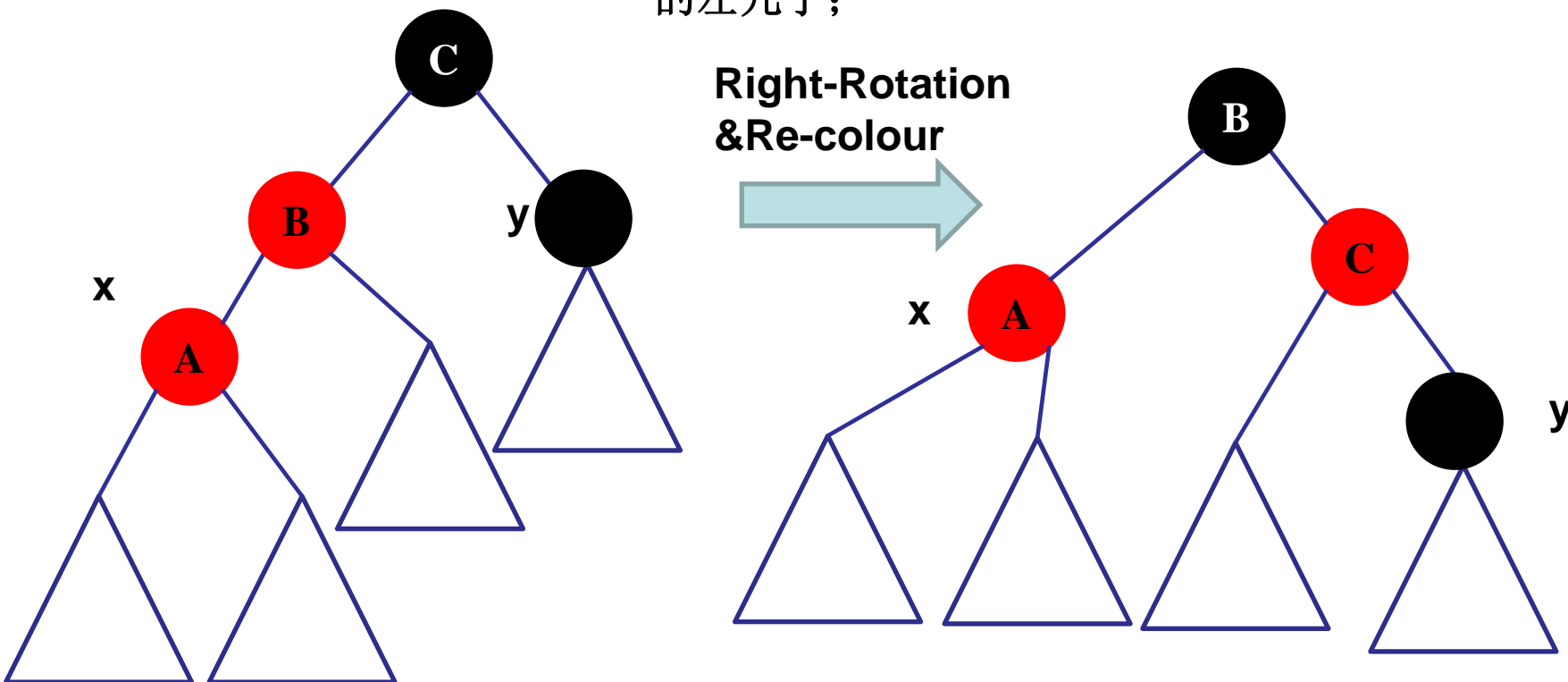


# 红黑树---插入



(CASE-3)

$x$ 的叔父是黑， $x$ 是 $P[x]$ 的左儿子；

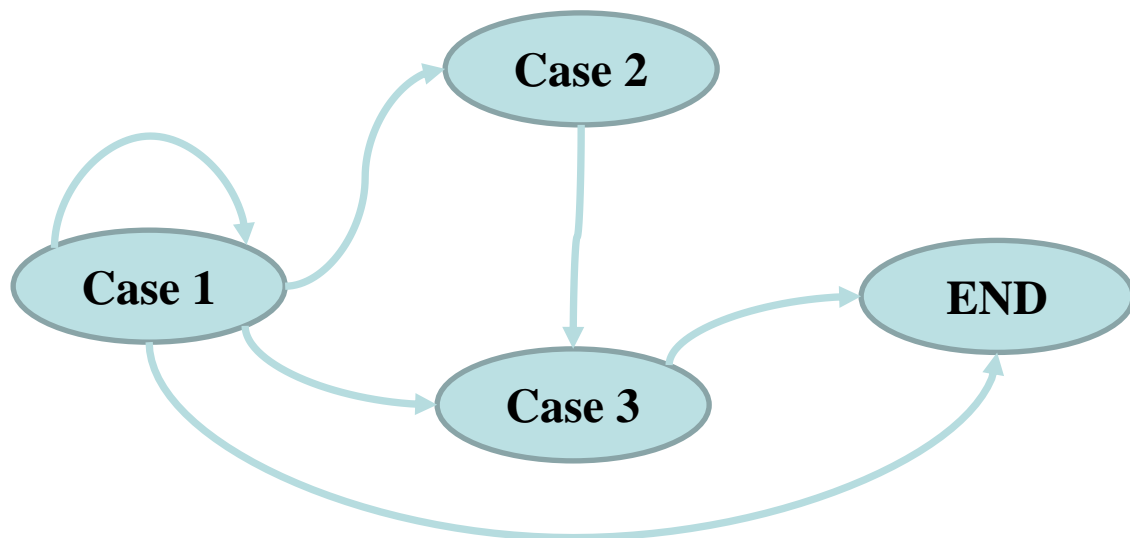


插入算法停止在此步

# 红黑树---插入



为什么红黑树的插入算法是 $O(\log(n))$ 复杂度， $n$ 为树中节点的数目。

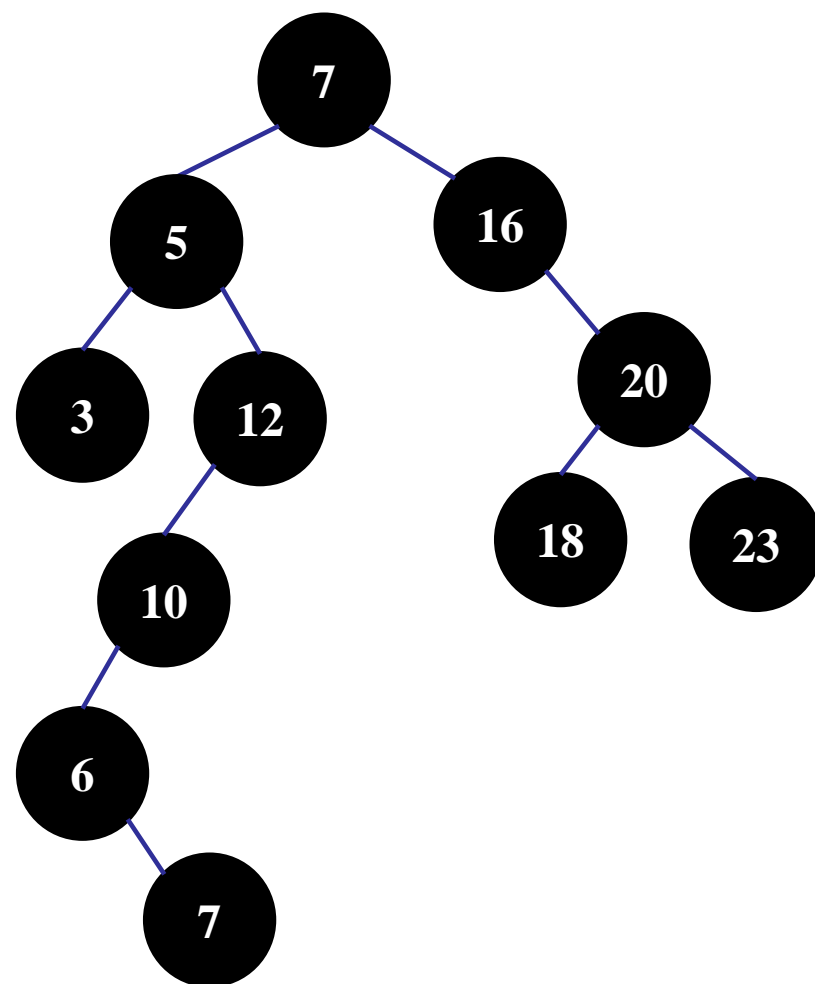
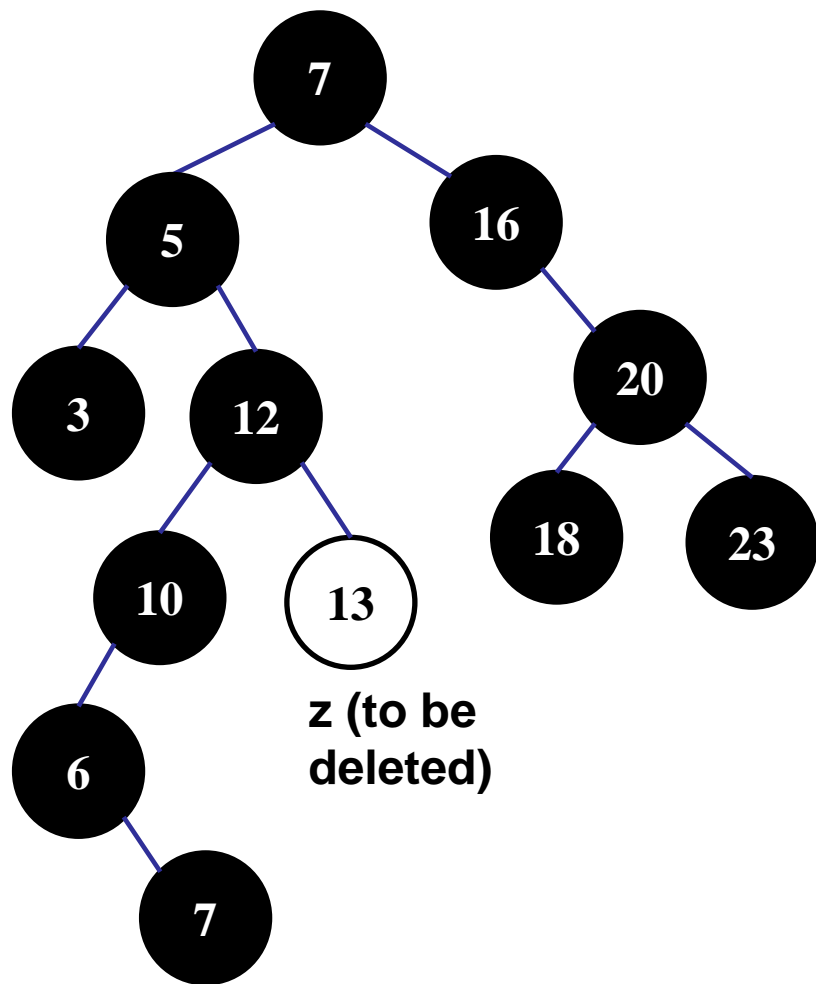


每次**Case 1**可以将待处理的 $x$ 向上推2层，所以**Case 1**最多执行 $O(\log(n))$ 次。

# 二叉搜索树(BST)的删除



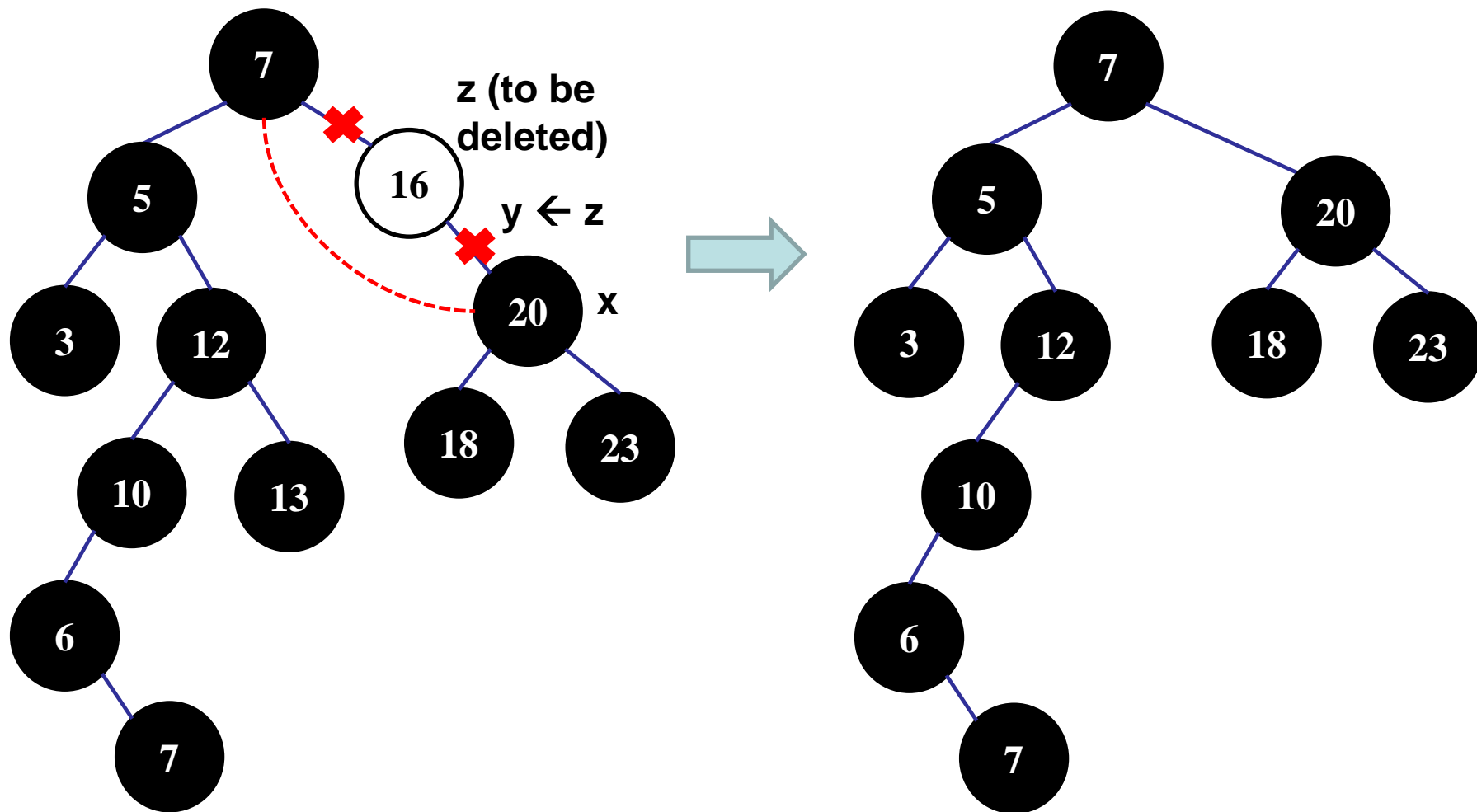
Case (a): z 没有儿子



# 二叉搜索树(BST)的删除



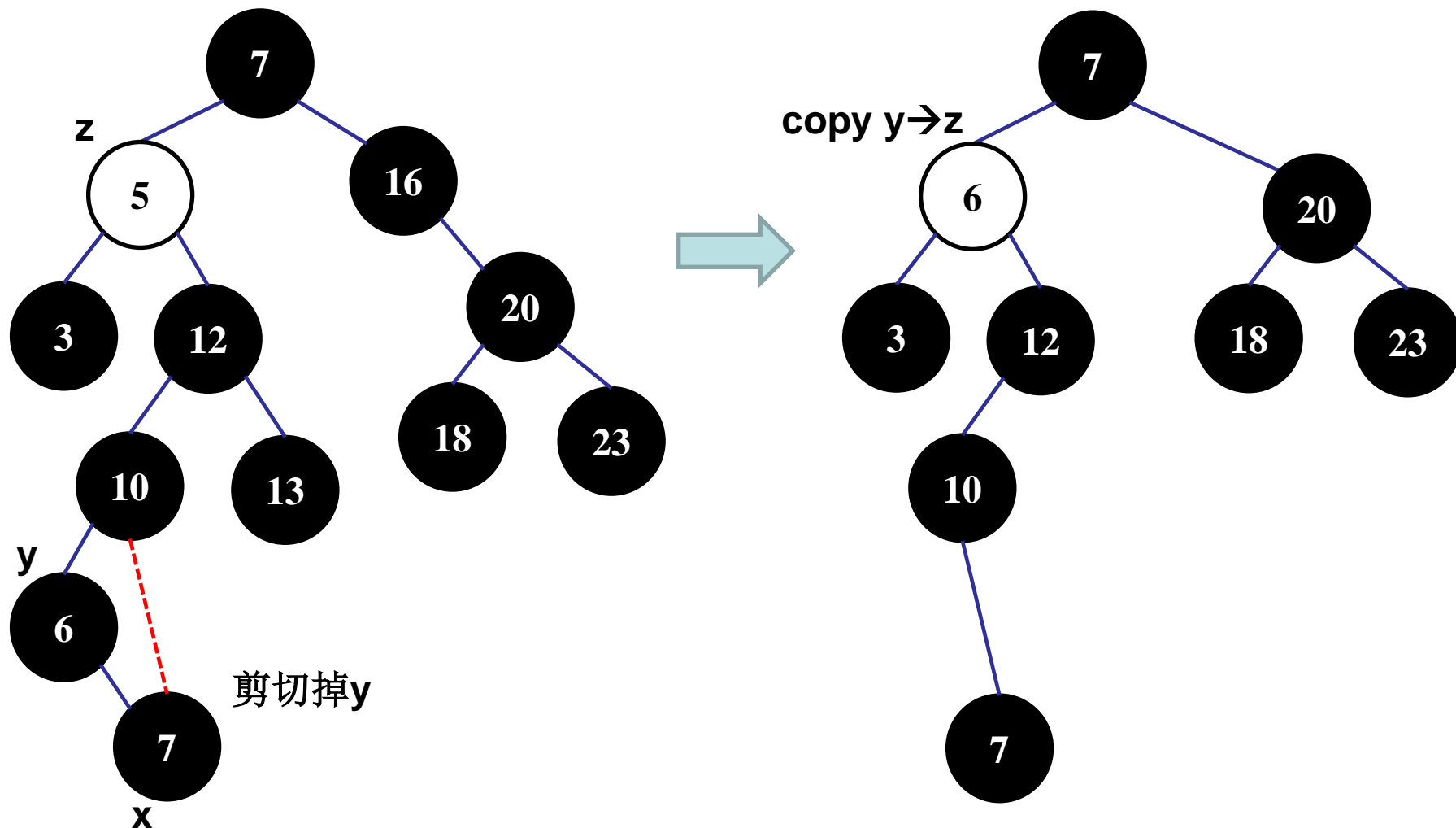
Case (a): z 有一个儿子



# 二叉搜索树(BST)的删除



Case (a):  $z$  有两个儿子



# 二叉搜索树(BST)的删除



RB\_Delete(T, z)

  If Left[z]==NULL or Right(z)==NULL

    then  $y \leftarrow z$

  else  $y \leftarrow \text{Tree\_Successor}(z)$  // z 有两个儿子的时候

  If Left[y]!=NULL

    then  $x \leftarrow \text{Left}[y]$

  else  $x \leftarrow \text{Right}[y]$

$P[x] \leftarrow P[y]$

  If  $P[y] == \text{NULL}$

    then  $\text{root}[T] \leftarrow x$

  else if  $y = \text{left}[p[y]]$

    then  $\text{left}[p[y]] \leftarrow x$

    else  $\text{right}[p[y]] \leftarrow x$

  If  $y \neq z$  // z 没有空子节点的情况，即z 有两个儿子的时候

    then  $\text{key}[z] \leftarrow \text{key}[y]$  // copy data y to z

  If  $\text{colour}[y] == \text{BLACK}$

    then RB\_DELETE\_FixUP(T,x)

  return y



注:

If  $\text{Colour}[y] == \text{RED}$ , 不用做任何调整

Why?

1. 任何结点的 **Black\_height** 没有改变;
2. 不会产生红-红冲突
3.  $y$  此时不可能为 **root** (**root** 一定是黑色的)

If  $\text{Colour}[y] == \text{BLACK}$ , 会引起下面的问题

1. if  $y$  是 **root**,  $y$  的一个红色儿子结点成为了新的 **root**, 违反了性质2
2. If  $x$  和  $P[y]$  (现在是  $P[x]$  了) 都是红色的, 违反了性质4
3. 删除黑色点  $y$ , 使得以前含有  $y$  的路径都少了一个黑色结点 (即 **Black\_height** 减少了)

Idea: 把  $y$  的“黑色”, 传递给它的儿子 (即  $x$ )

→ 新的问题是:  $x$  是“双黑” (**Double Black**), 或者成为 (**RED-BLACK**) 结点, 违反了性质1

目标:

循环以下步骤，从而达到以下目标

- (1) **x**指向一个**Red-Black**结点，此时只用把这个点的颜色改成**Black**就可以了；
- (2) **x**指向一个**root**结点，直接把此结点的颜色设为**BLACK**；
- (3) 其他情况，需要继续调整。

目标:

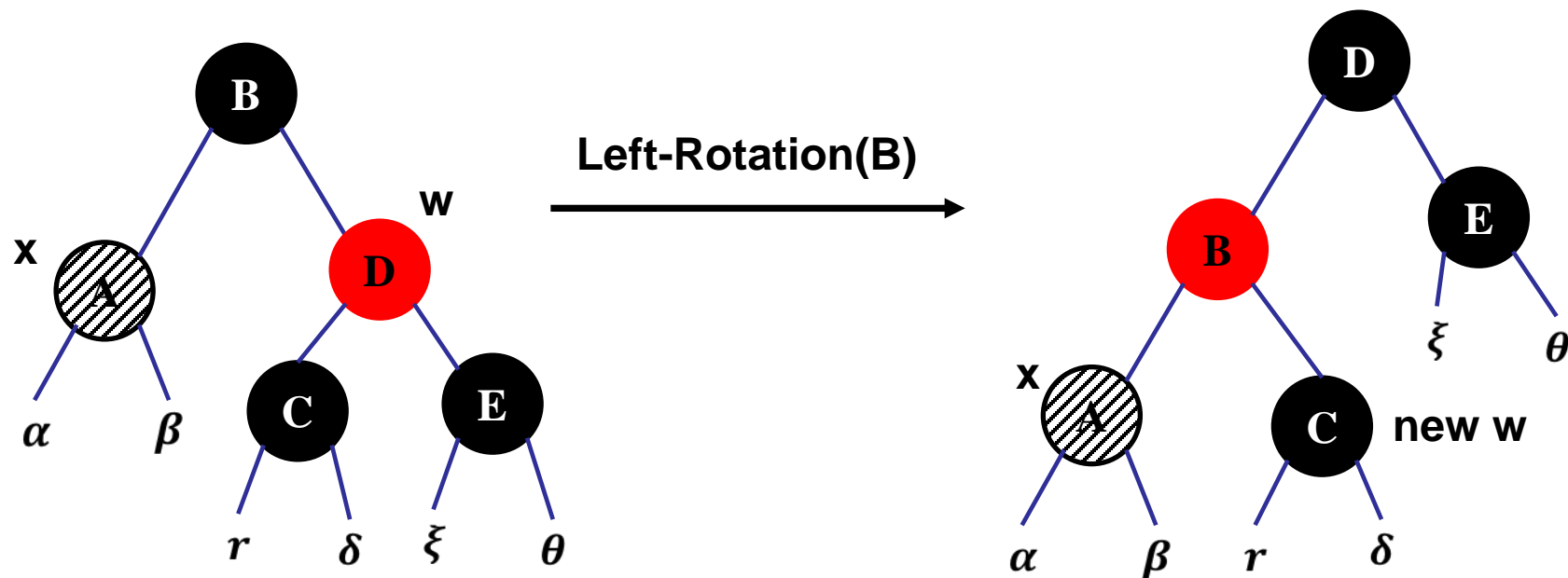
循环以下步骤，从而达到以下目标

- (1) **x**指向一个**Red-Black**结点，此时只用把这个点的颜色改成**Black**就可以了；
- (2) **x**指向一个**root**结点，直接把此结点的颜色设为**BLACK**；
- (3) 其他情况，需要继续调整。

# 红黑树的删除



Case 1:  $x$  的兄弟  $w$  为红

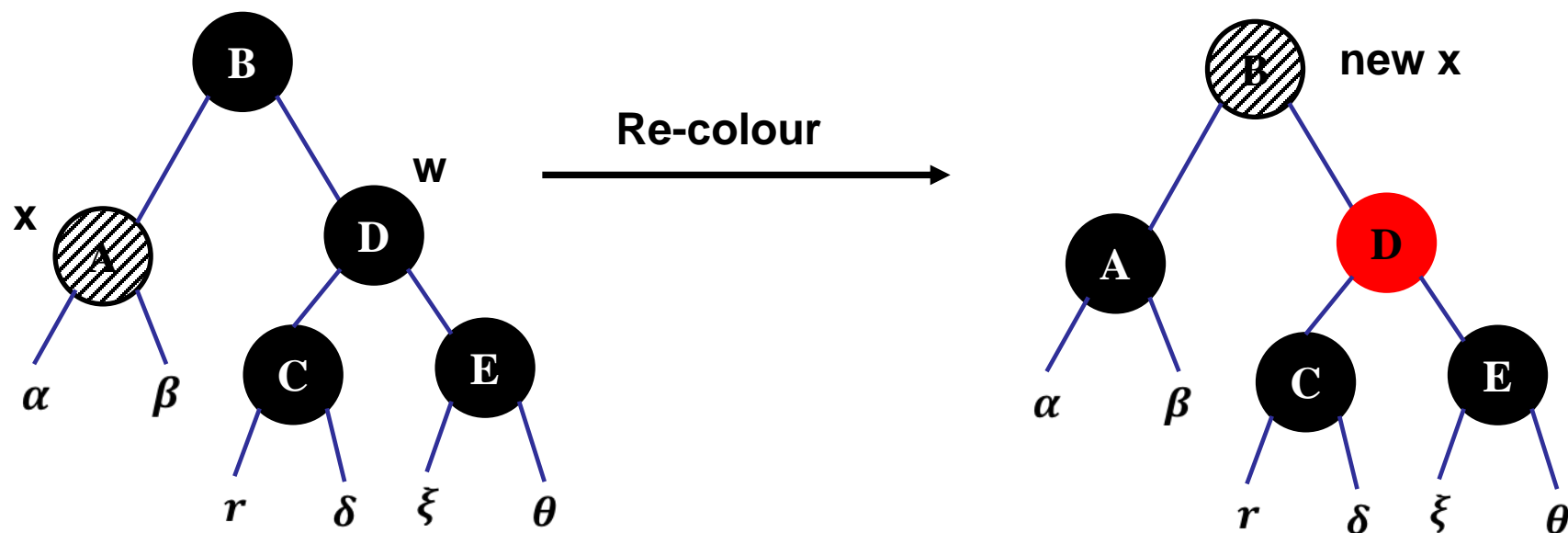


Case 1  $\rightarrow$  Cases 2, 3, 4

# 红黑树的删除



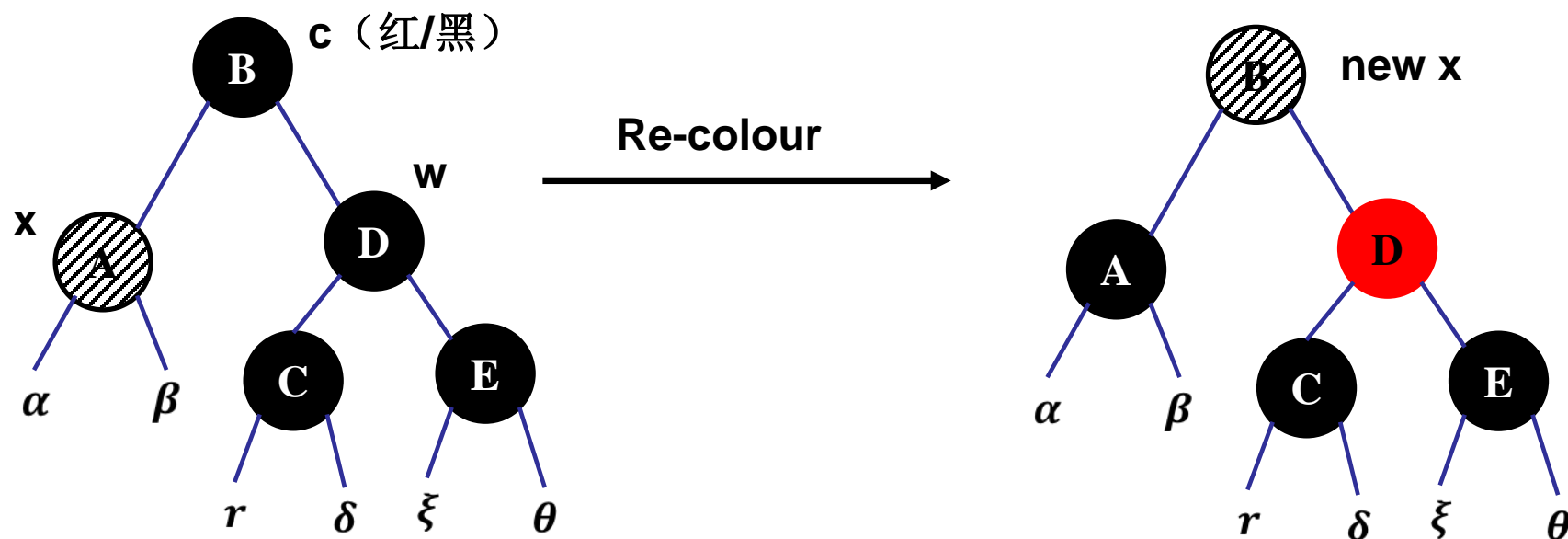
Case 2:  $x$ 的兄弟 $w$ 为黑，并且 $w$ 的两个儿子也为黑



# 红黑树的删除



**Case 2:**  $x$  的兄弟  $w$  为黑，并且  $w$  的两个儿子也为黑



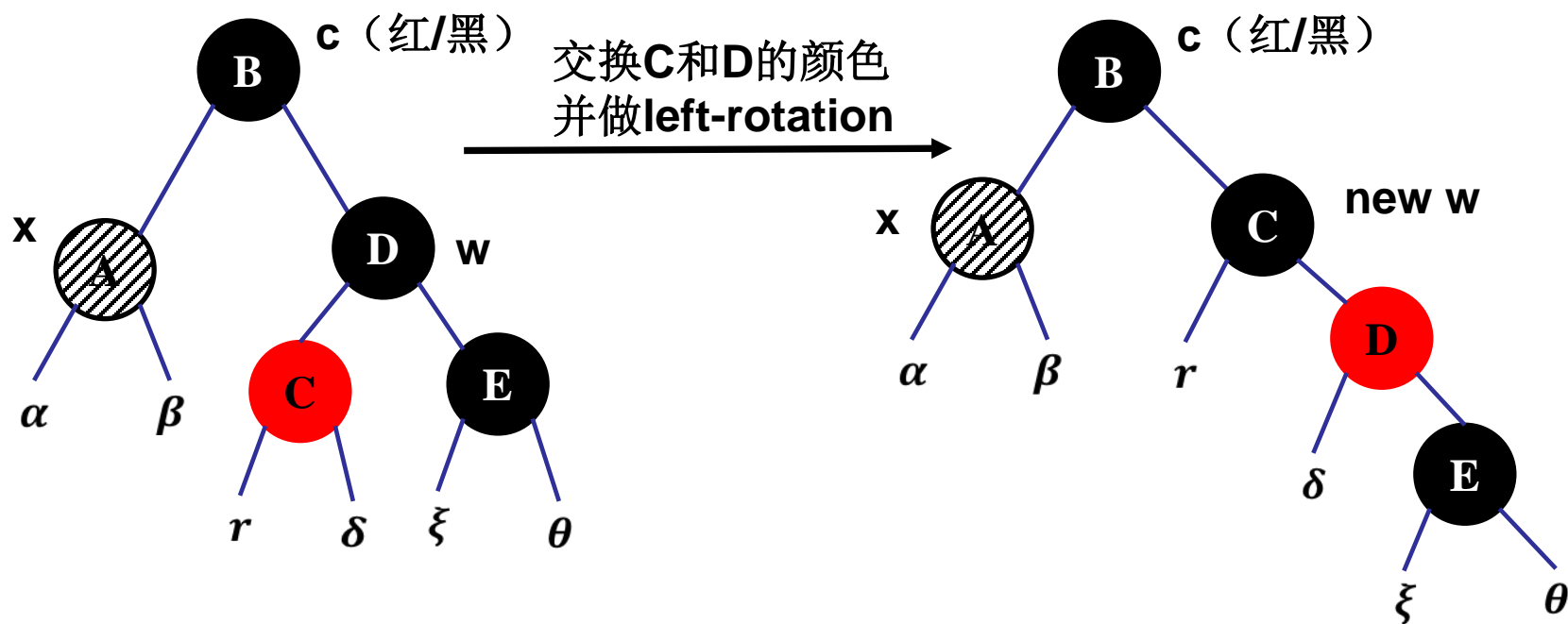
此时，如果原来的  $c$  指向的点为红色的，此时它变为 **red-black**，循环结束，最后把这个点变成黑色的

如果  $c$  指向的点为黑色的，此时它变为 **double-black**， $c$  变为 **new x**，迭代上述操作（高度提升1层）

# 红黑树的删除



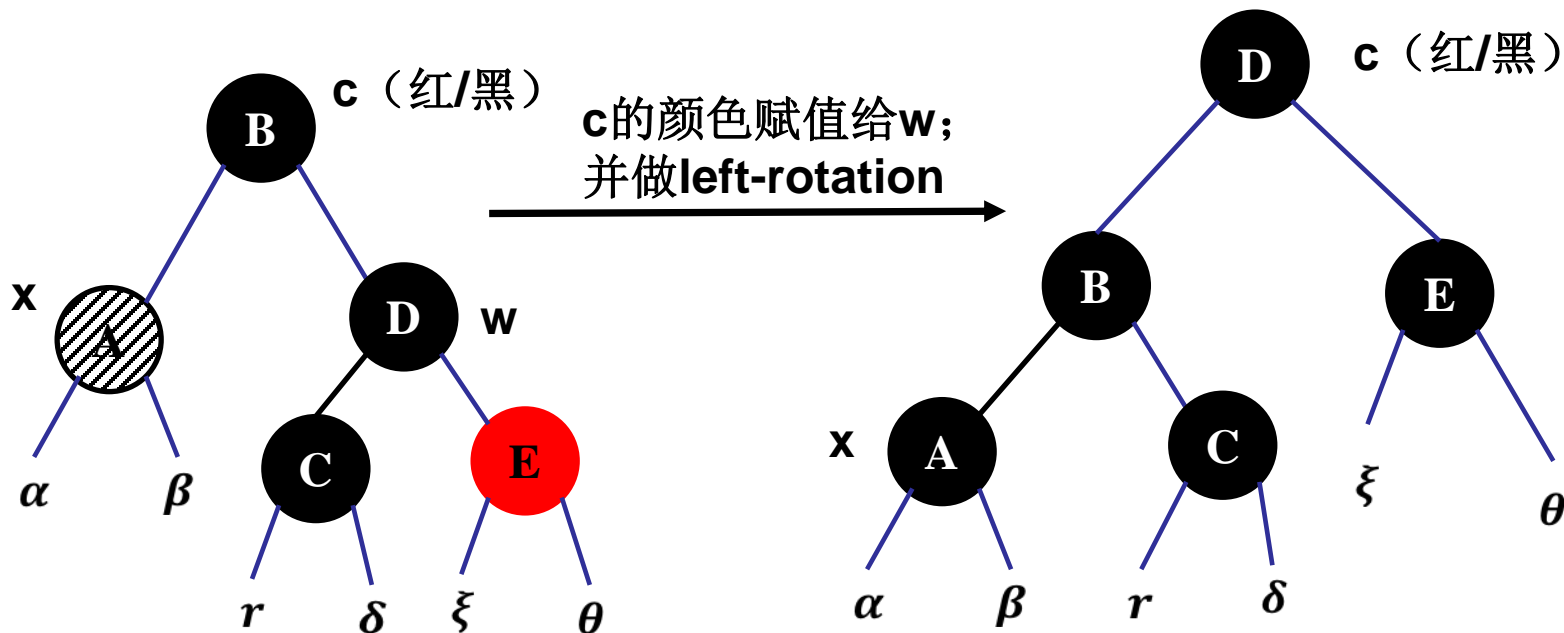
**Case 3:**  $x$ 的兄弟 $w$ 为黑，并且 $w$ 的左儿子为红， $w$ 的右儿子为黑。



# 红黑树的删除



**Case 4:**  $x$ 的兄弟 $w$ 为黑，并且 $w$ 的右儿子为红（无论 $w$ 左儿子颜色）



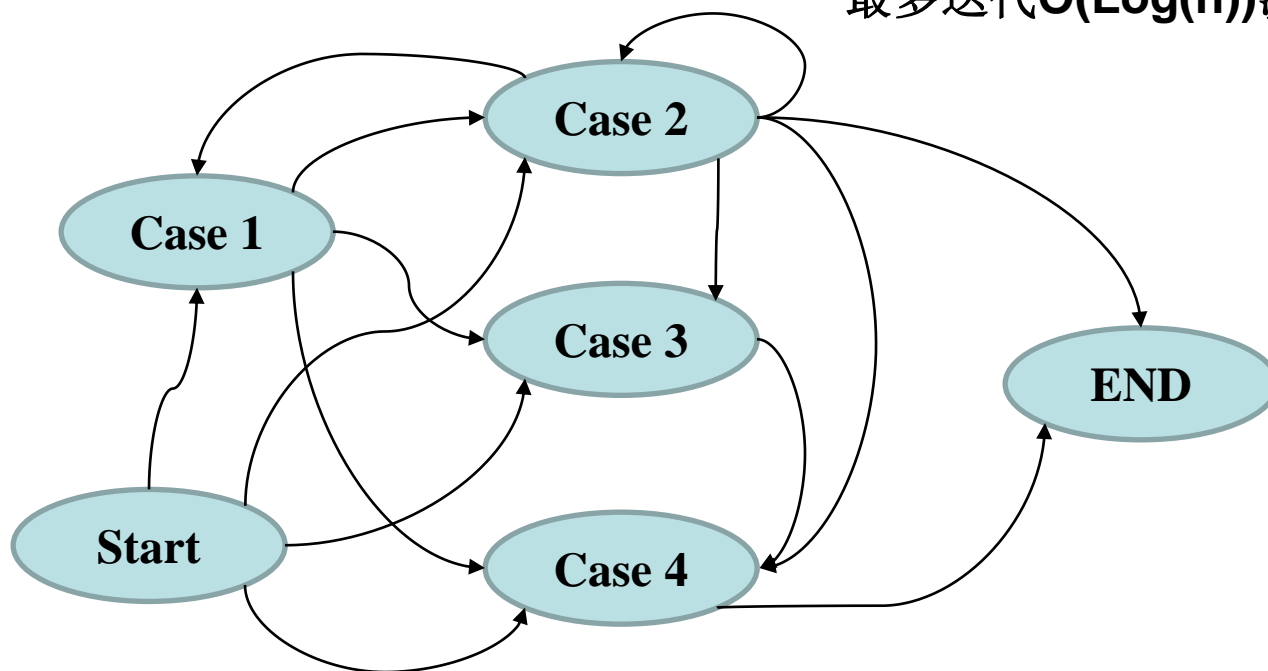
```
Color[w] ← Color[P[x]]
Color[P[x]] ← BLACK
Color[right[w]] ← BLACK
LEFT-Rotate(P[x])
x ← root[T] // 终止循环
```



# 红黑树的删除



最多迭代 $O(\log(n))$ 次（树高）



如果**Case 2**是由**Case 1**转化得到的，则  
**Case 2**一定一次就停止了  
(Why? new x (B)一定原为红色的)

# 红黑树的删除



```
RB_DELETE_FIXUP(T,x)
  WHILE x!=root(T) AND color[x]==BLACK
    Do if x==left[P[x]]
      then w←right[p[x]]
        if colour[w]==RED
          <CASE_1>
            else if colour[left[w]]==BLACK AND
colour[right[w]]==BLACK
              then <CASE_2>
                else if colour[left[w]]==RED AND
colour[right[w]]==BLACK
                  then <CASE_3>
                    else <CASE_4>
                      else (same as with “right” “left” exchange)
```

## ➤ RB-Tree局部平衡

- ➡ 统计性能好于AVL树

- ➡ 且增删记录算法性能好、易实现

## ➤ C++ STL的set、multiset、map、multimap都应用了红黑树的变体

A decorative graphic on the left side of the slide, consisting of overlapping green, blue, and yellow squares with a black crosshair.

# 再见…

---

**联系信息:**

**电子邮件:** [zoulei@pku.edu.cn](mailto:zoulei@pku.edu.cn)

**电 话:** 82529643