第十一章 索引

张铭 主讲

采用教材：《数据结构与算法》，张铭，王腾蛟，赵海燕 编写
高等教育出版社，2008.6 （“十二五”国家级规划教材）

http://jpk.pku.edu.cn/course/sjjg/
https://www.icourse163.org/course/PKU-1002534001

第十一章 索引

- 基本概念
- 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 位索引技术
- 11.6 红黑树

第十一章 索引

- **课程目标**：排序、索引、**检索**
- **学习目的**：索引思想和技术
- **问题设置**：索引的意义？ **加快检索速度**
- **课程讨论要点**：多级索引，**B/B+树**，**红黑树**
- **拓展认知**：**AA树**，AVL，Splay

背景

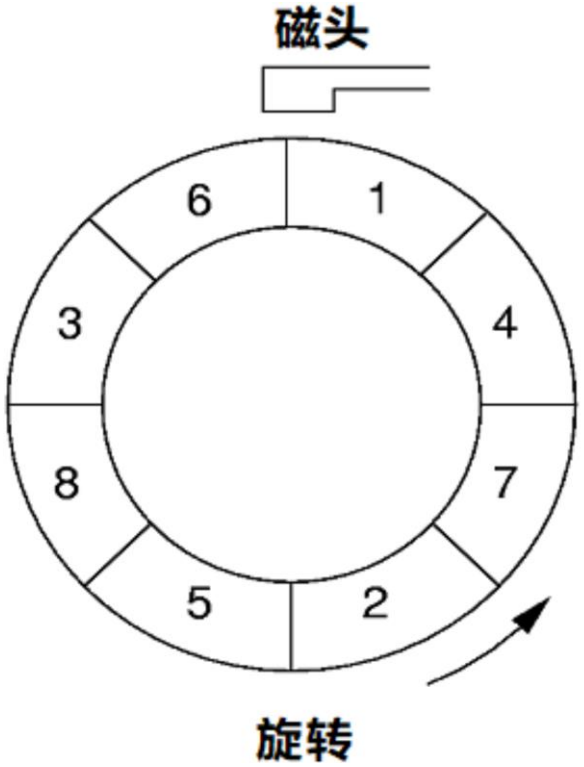
- “大数据”存放在大型数据库或者大型文件系统中
- 既要支持高效检索，又要支持动态变化
  - 随着数据的插入、删除和更新而动态调整
- 索引技术是支持高效数据组织管理的有效手段
  - 很多情况下进行了预排序
  - 索引为高效的检索服务

输入顺序文件

- **输入顺序文件(entry-sequenced file)**

  按照记录进入系统的顺序存储记录

  - 输入顺序文件相当于一个磁盘中未排序的线性表
  - 因此不支持高效率的检索



主码

- **主码(primary key)** 是数据库中的每条记录的 **唯一** 标识
  - 例如，公司职员信息的记录的主码可以是职员的身份证号码
  - 如果只有主码，不便于各种灵活检索

辅码

- **辅码(secondary key)**

  是数据库中可以出现重复值的码

- 辅码索引把一个辅码值与具有这个辅码值的每一条记录的主码值关联起来
  - 大多数检索都是利用辅码索引来完成的

索引

- **索引(indexing)** 是把一个关键码与它对应的数据记录的位置相关联的过程
  - (关键码，指针)对，即(key, pointer)
  - 指针指向主要数据库文件（即“主文件”）中的完整记录
- 索引文件(index file) 是用于记录这种联系的文件组织结构
- 索引技术是组织大型数据库的一种重要技术
  - 高效率的检索
  - 插入、更新、删除

索引文件

- 一个主文件可以有多个相关索引文件
  - 每个索引文件往往支持一个关键码字段
  - 不需要重新排列重排主文件
- 可以通过该索引文件高效访问记录中该关键码值

稠密索引 vs 稀疏索引

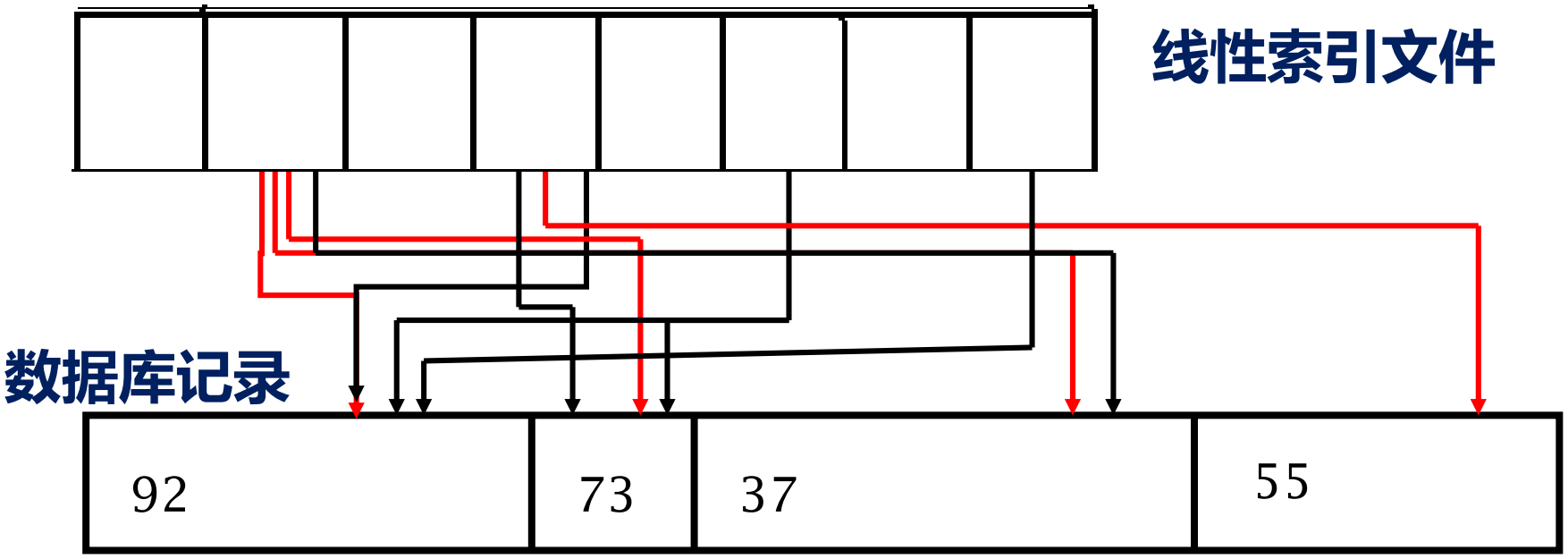
- 稠密索引：对 **每个** 记录建立一个索引项
  - 主文件不按照关键码的顺序排列
- 稀疏索引：对 **一组** 记录建立一个索引项
  - 记录按照关键码的顺序存放
  - 可以把记录分成多个组（块）
  - ➢索引指针指向的这一组记录在磁盘中的起始位置

11.1 线性索引

- 基本概念
- 线性索引的优点
- 线性索引的问题
- 二级线性索引

线性索引文件

- 按照关键码的顺序进行排序
- 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



线性索引的问题

- 线性索引太大，存储在磁盘块中
  - 在一次检索过程中可能多次访问几个磁盘块，从而影响检索的效率
  - 使用二级线性索引
- 更新线性索引
  - 在数据库中插入或者删除记录时

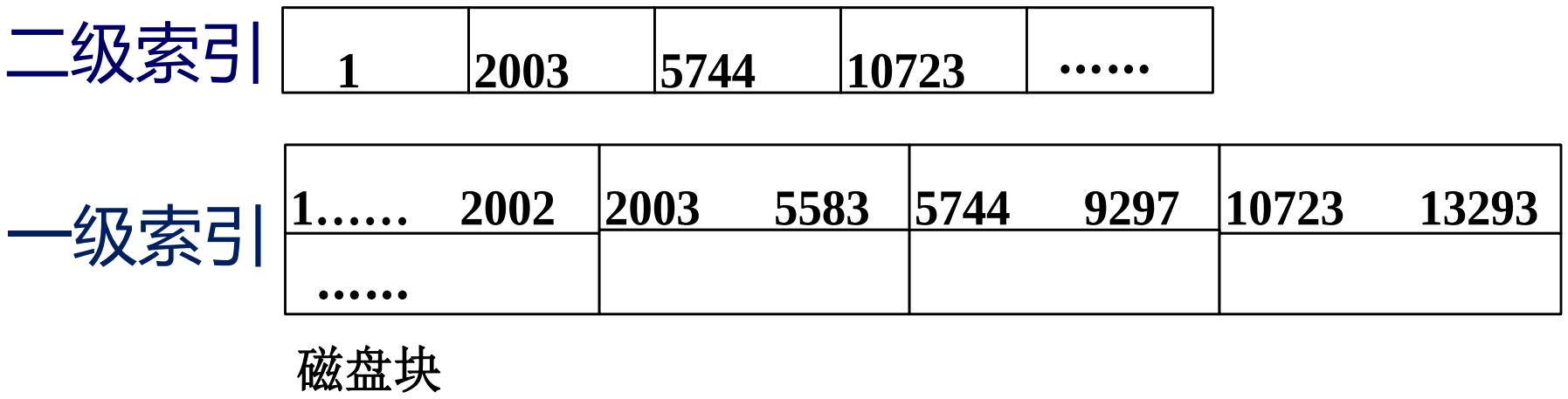
二级线性索引

- 例如，磁盘块的大小是 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 条这样的索引对

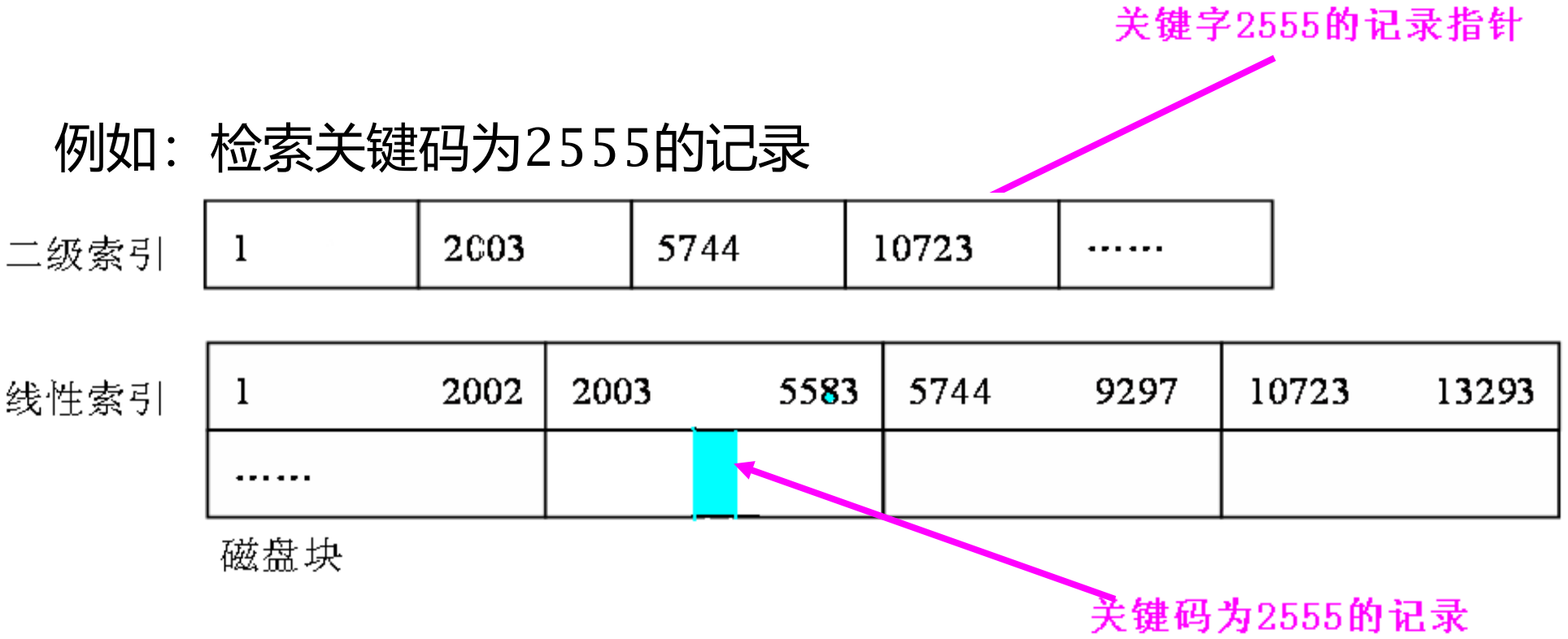
- 假设数据文件包含 10000 条记录
  - 稠密一级线性索引中包含 10000 条记录
    - 10000/128 = 78.1
    - 那么一级线性索引占用 79 个磁盘块
  - 相应地，二级线性索引文件中有 79 项索引对
  - 这个二级线性索引文件可以在一个磁盘块

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的值相同
- 指针指向相应磁盘块的起始位置



例如：检索关键码为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，找到所需要的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

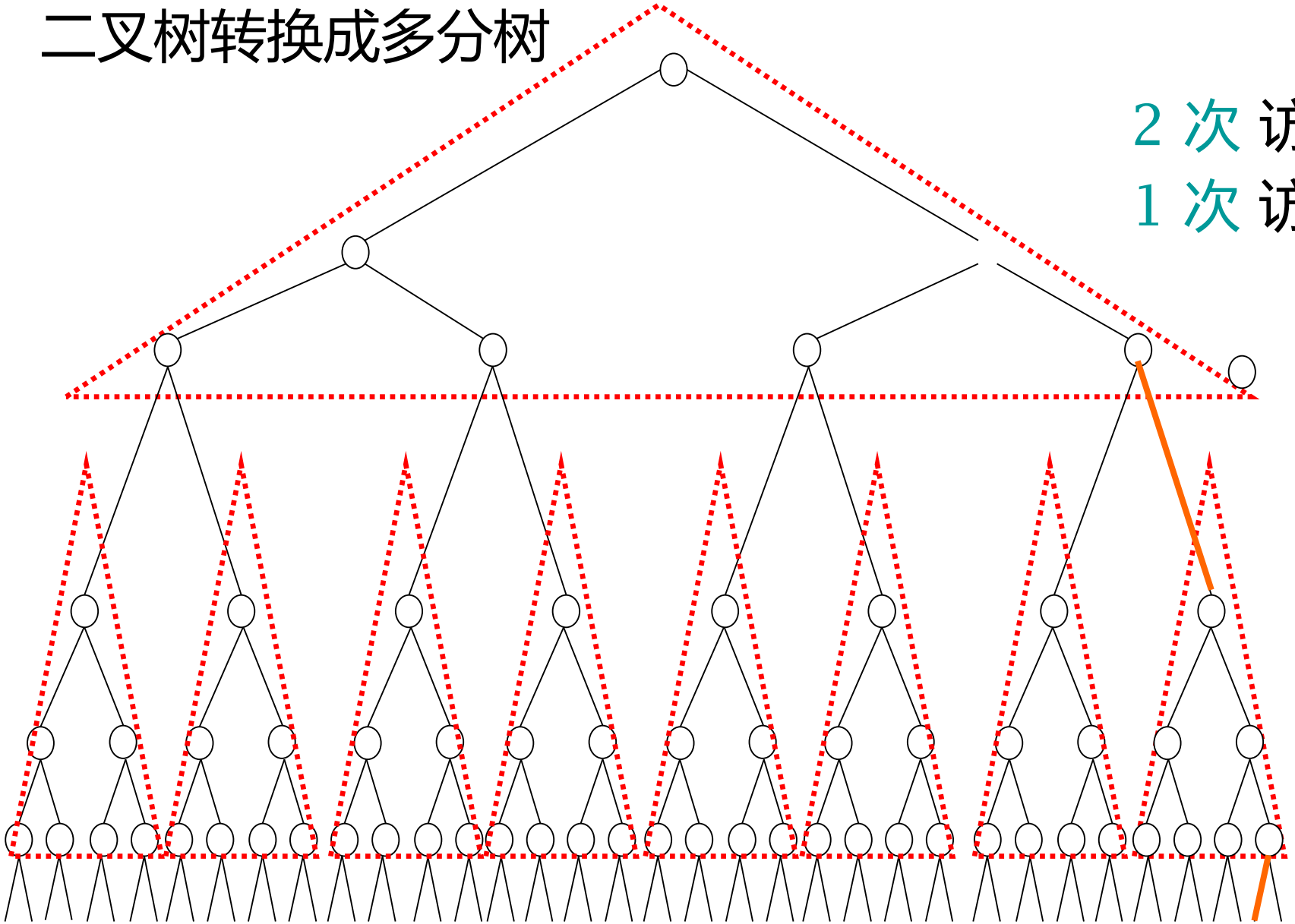
静态索引

- 索引结构在文件创建、初始装入记录时生成
- 一旦生成就固定下来，在系统运行(例如插入和删除记录)过程中索引结构并不改变
- 只有当文件再组织时才允许改变索引结构

多分树

- 静态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

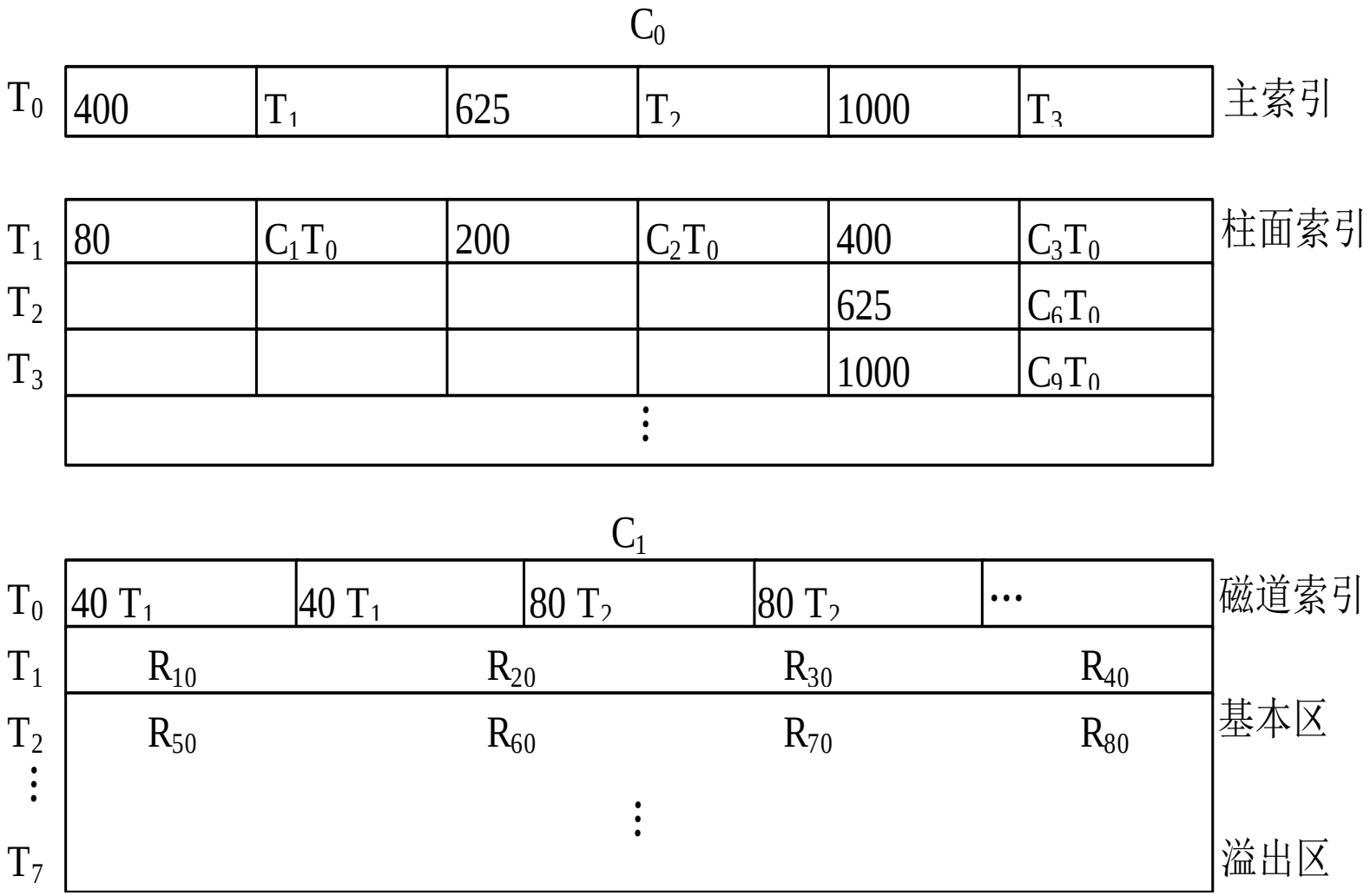


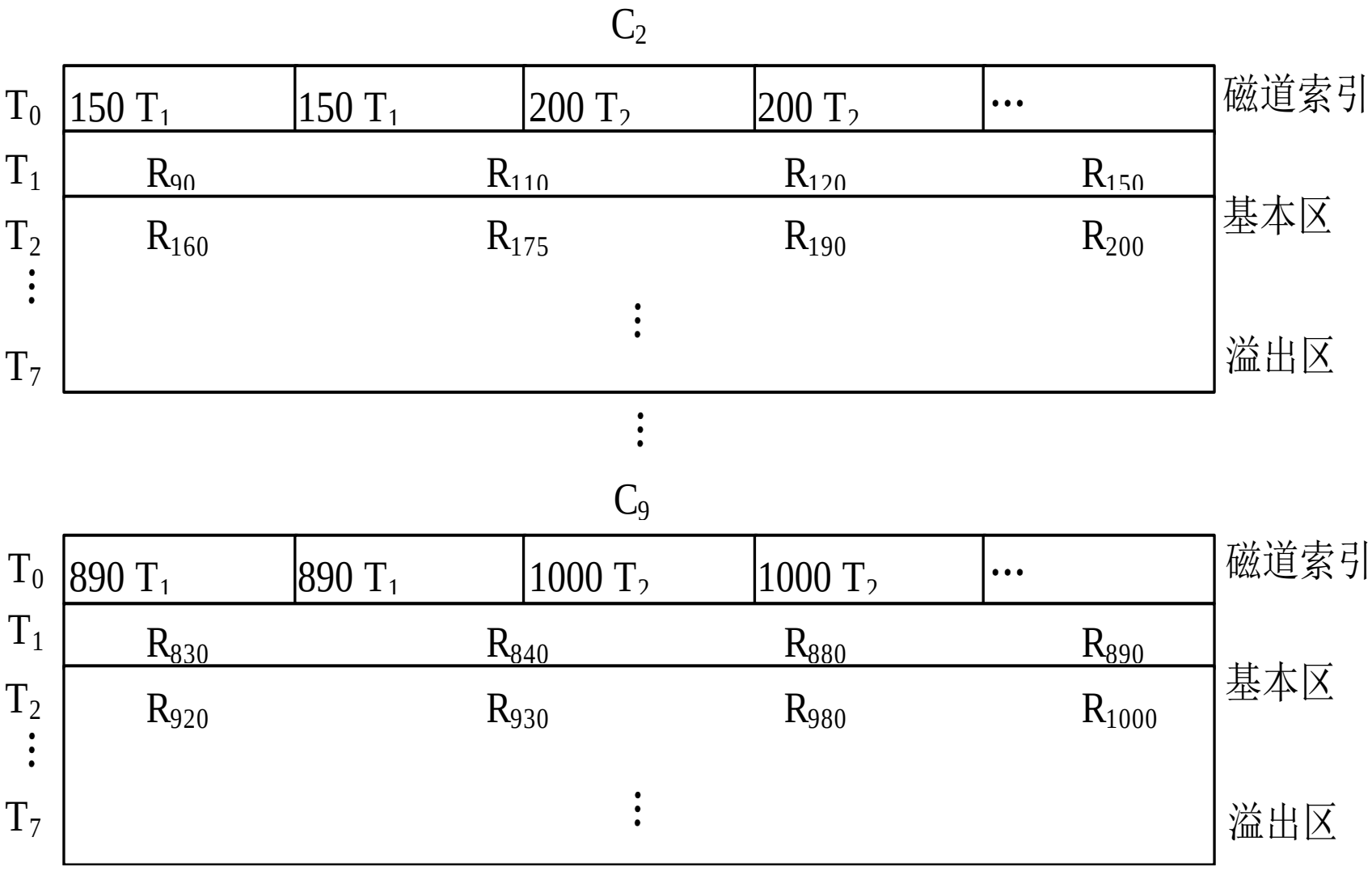
2 次 访问索引块

1 次 访问外存数据块

ISAM

- 基于多分树的 ISAM (Index Sequential Access Method)
  - 为磁盘存取而设计
  - 结构采用多级索引
    - 主索引
    - 柱面索引
    - 磁道索引
- 在采用基于 B^+ 树的 VSAM (Virtual Storage Access Method) 技术之前，IBM 公司曾广泛地采用 ISAM 技术


C0
T0
400
T1
625
T2
1000
T3
主索引
T1
80
C1T0
200
C2T0
400
C3T0
T2
625
C6T0
T3
1000
C9T0
柱面索引
C1
T0
40 T1
40 T1
80 T2
80 T2
…
磁道索引
T1
R10
R20
R30
R40
T2
R50
R60
R70
R80
基本区
T7
溢出区


C2
T0
150 T1
150 T1
200 T2
200 T2
…
磁道索引
T1
R90
R110
R120
R150
T2
R160
R175
R190
R200
基本区
T7
溢出区
C9
T0
890 T1
890 T1
1000 T2
1000 T2
…
磁道索引
T1
R830
R840
R880
R890
T2
R920
R930
R980
R1000
基本区
T7
溢出区

思考

- 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?

主要内容

- 基本概念
- 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 位索引技术
- 11.6 红黑树

11.3 倒排索引（Inverted Index）

- 11.3.1 基于属性的倒排
  - 要求检索结构中某个或若干个属性满足一定条件的结点（不是按关键码的值检索）
  - 按照属性建立索引

- 11.3.2 对正文文件的倒排
  - 以文中的词（word）为索引项建立的索引

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
……					

11.3.1 基于属性的倒排

- 对某属性按属性值建立索引表，称倒排表
- “属性－指针”对 (attr，ptrList)
  - （属性值，具有该属性值的各记录指针）
  - 指针可以是关键码，或该记录的主文件地址
- 颠覆主文件的顺序，因而称为倒排索引
- 属性往往是离散型的
  - 对于连续型的索引，往往用B树
- 倒排文件：带有倒排索引的文件

11.3.1 基于属性的倒排

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

优缺点

- 优点：
  - 能够对于基于属性的检索进行较高效率的处理
- 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) 处理的就是“建立一个数据结构以提供对文本内容的快速检索”
- 方法
  - 词索引 (word index)
  - 全文索引 (full-text index)

词索引

- 基本思想：
  - 把正文看作由符号和词所组成的集合，从正文中抽取出关键词，然后用这些关键词组成一些适合快速检索的数据结构。
- 适用于多种文本类型，特别是那些可以很容易就解析成一组词的集合的文本
  - 适用于英文
  - 中文等东方文字要经过“切词”处理

全文索引

- 基本思想：
  - 把正文看作一个长的字符串
  - 在数据结构中记录的是子字符串的开始位置
  - 查询就可以针对正文中的任何子字符串
- 可以对每一个字符建立索引，从而使查询词不再限于关键词
- 需要更大的空间

倒排文件使用最广泛的是词索引

- 词索引使用 最广泛
- 一个已经排过序的关键词的列表
  - 其中每个关键词指向一个倒排(posting list)
    - 指向该关键词出现文档集合
    - 在文档中的位置

倒排索引建立示例

正文文件：由6个文档组成，每个文档都是长字符串

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.

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	

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

建立正文倒排文件

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

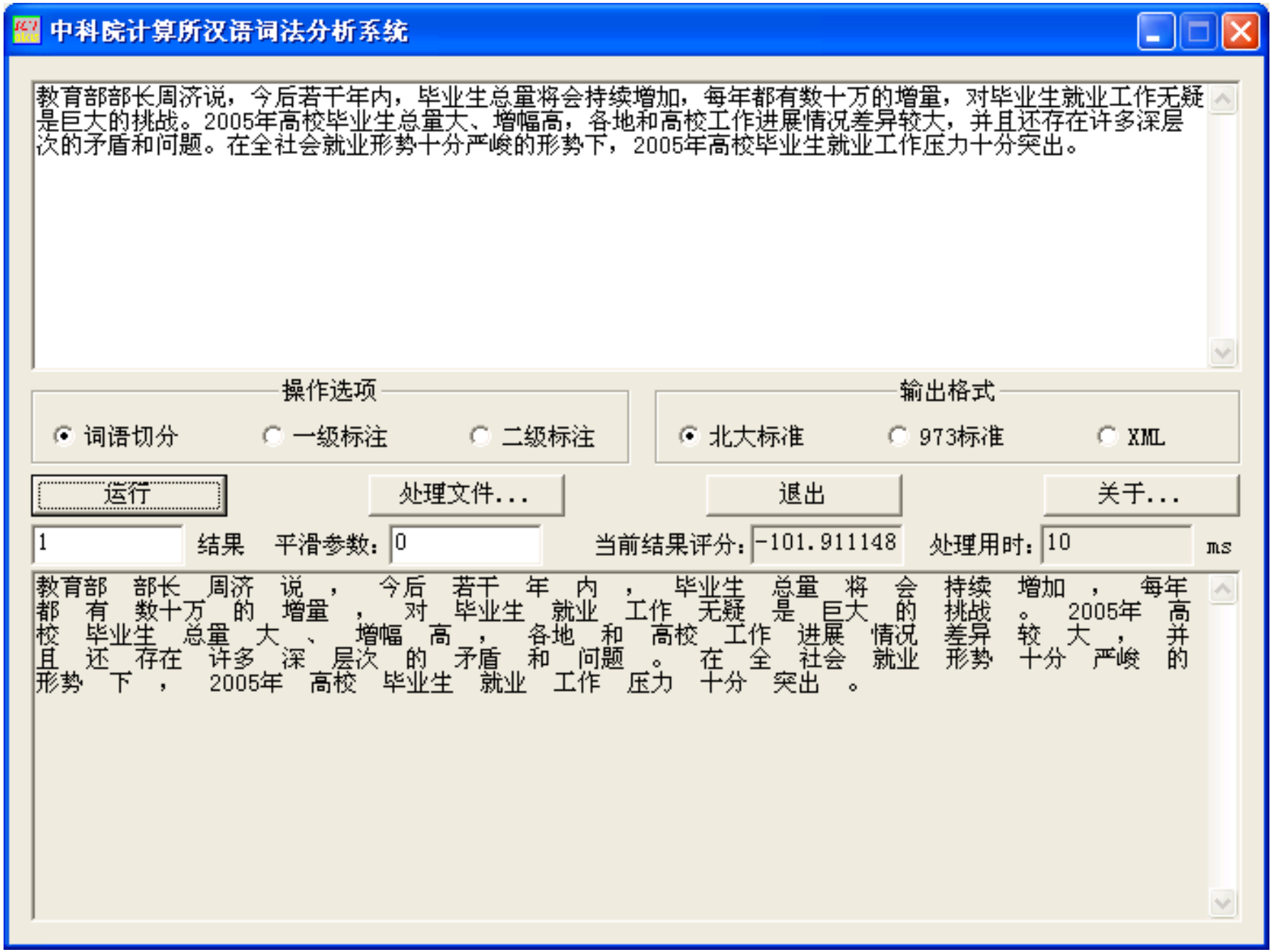
切分正文记录取决于程序的需要

定长的块、段落、章节，甚至一组文档

建立正文倒排文件（续1）

2. 给每条记录赋一组关键词

以人工或者自动的方式从记录中抽取关键词
停用词(Stopword)
抽词干(Stemming)
切词（Segmentation)


中科院计算所汉语词法分析系统
教育部部长周济说，今后若干年内，毕业生总量将会持续增加，每年都有数十万的增量，对毕业生就业工作无疑是巨大的挑战。2005年高校毕业生总量大、增幅高，各地和高校工作进展情况差异较大，并且还存在许多深层次的矛盾和问题。在全社会就业形势十分严峻的形势下，2005年高校毕业生就业工作压力十分突出。
操作选项
词语切分
一级标注
二级标注
输出格式
北大标准
973标准
XML
运行
处理文件...
退出
关于...
1
结果
平滑参数:
0
当前结果评分:
-101.911148
处理用时:
10
ms
教育部 部长 周济 说 ， 今后 若干 年 内 ， 毕业生 总量 将 会 持续 增加 ， 每年 都 有 数十万 的 增量 ， 对 毕业生 就业 工作 无疑 是 巨大 的 挑战 。 2005年 高校 毕业生 总量 大 、 增幅 高 ， 各地 和 高校 工作 进展 情况 差异 较 大 ， 并且 还 存在 许多 深 层次 的 矛盾 和 问题 。 在 全 社会 就业 形势 十分 严峻 的 形势 下 ， 2005年 高校 毕业生 就业 工作 压力 十分 突出 。

建立正文倒排文件（续2）

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

对关键词的检索

- 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

倒排文件优劣

- 高效检索，用于文本数据库系统
  - 20万条数据，用MSFTESQL
  - LIKE花费4秒
  - FREETEXT或者CONTAINS基本是毫秒级
- 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）
  - 倒排文件中的索引效率可能不高
  - 需要的空间代价往往很高

思考

- 怎样有效地组织属性倒排索引表?

- 一个关键词如果在同一个文本中多次出现，它在倒排文件中的索引项是否能进行合并?

主要内容

- 基本概念
- 11.1 线性索引
- 11.2 静态索引
- 11.3 倒排索引
- 11.4 动态索引
  - 11.4.1 B 树
  - 11.4.2 B 树的性能分析
  - 11.4.3 B^+ 树
  - 11.4.4 B 树、B^+ 树索引性能的比较
- 11.5 位索引技术
- 11.6 红黑树

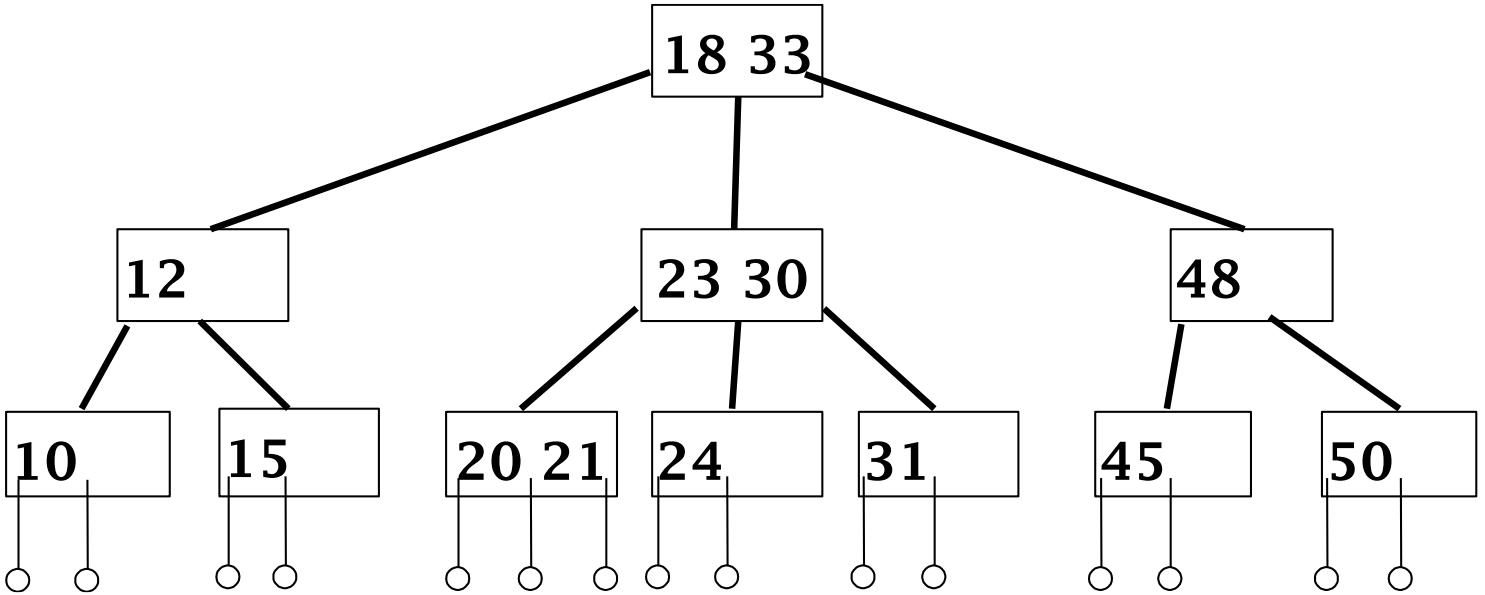
基本概念

- 动态索引结构
  - 索引结构本身也可能发生改变
  - 在系统运行过程中插入或删除记录时
- 目的
  - 保持较好的性能
    - 例如较高的 检索 效率

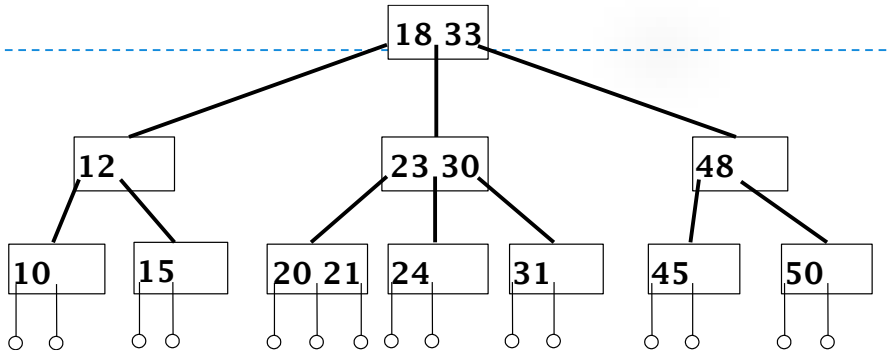
11.4.1 B 树

- 一种平衡的多分树 (Balanced Tree)

3 阶 B 树　2-3 树



m 阶 B 树的结构定义



(1) 每个结点至多有 m 个子结点

(2) 除根结点和叶结点外，其它每个结点至少有$\lceil \frac{m}{2} \rceil$个子结点

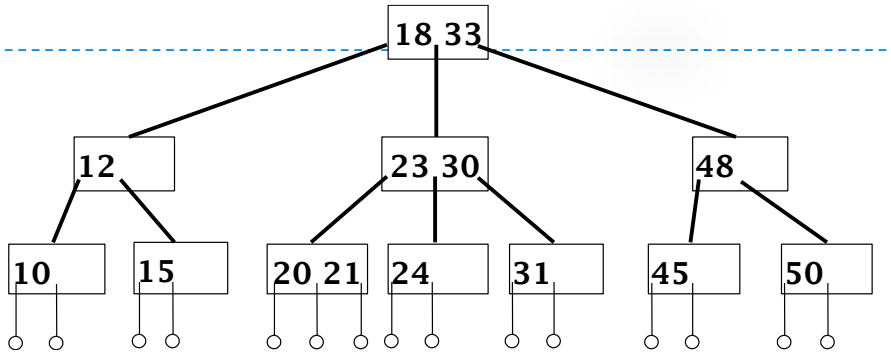
(3) 根结点至少有两个子结点

- 唯一例外的是根结点就是叶结点时没有子结点
- 此时 B 树只包含一个结点

(4) 所有的叶结点在同一层

(5) 有 k 个子结点的非根结点恰好包含 k-1 个关键码

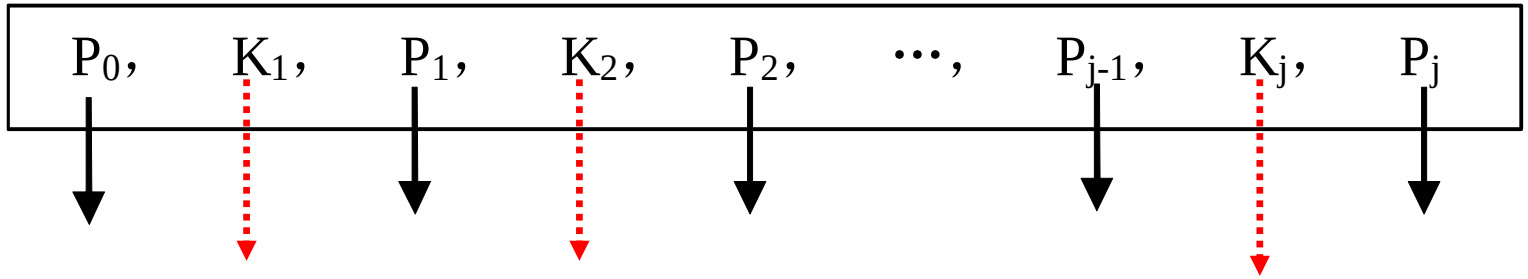
B 树的性质



- 树高平衡，所有叶结点都在同一层
- 关键码没有重复，父结点中的关键码是其子结点的分界
- B 树把（值接近）相关记录放在同一个磁盘页中，从而利用了访问局部性原理
- B 树保证树中至少有一定比例的结点是满的
  - 这样能够改进空间的利用率
  - 减少检索和更新操作的磁盘读取数目

B 树的结点结构

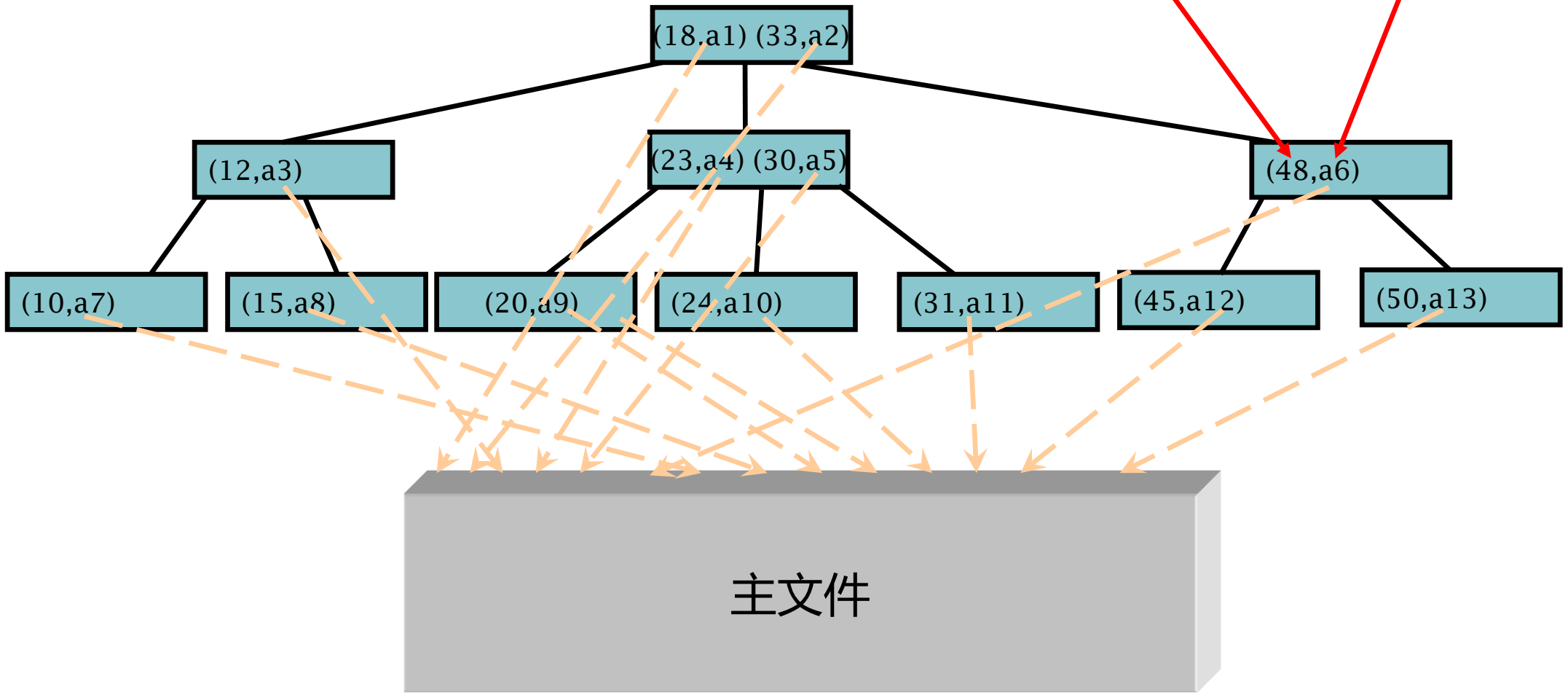
B 树的一个包含 j 个关键码，j+1 个指针的结点的一般形式为：



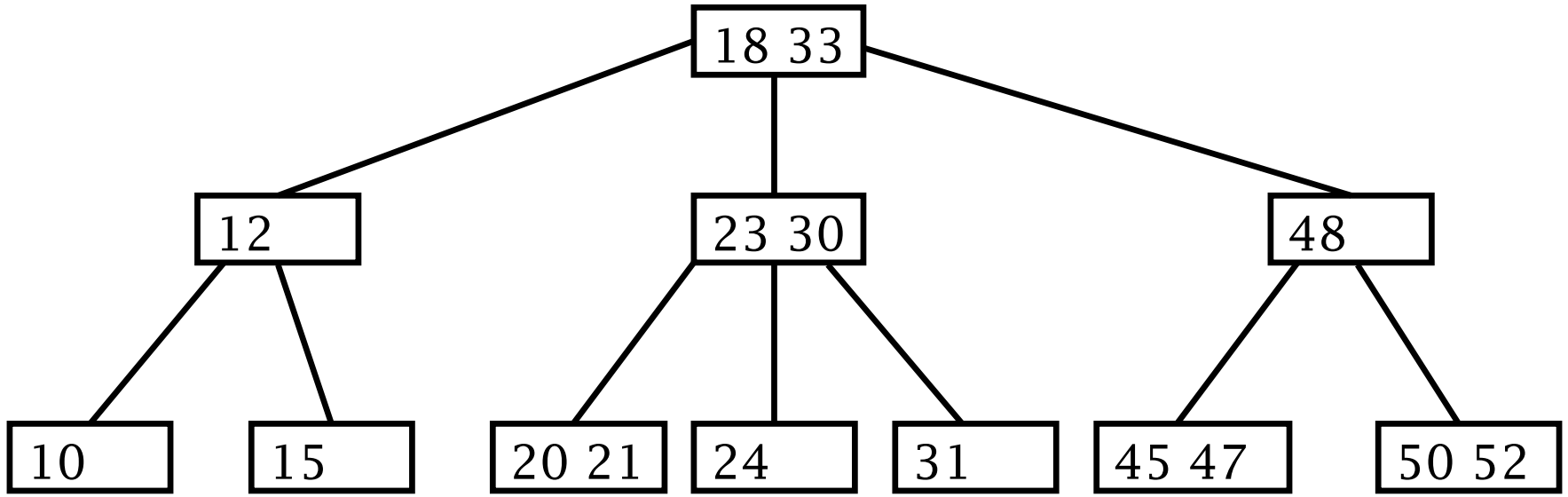
- 其中 K_i 是关键码值，$K_1<K_2<\ldots<K_j$，
- P_i 是指向包括 K_i 到 K_{i+1} 之间的关键码的子树的指针。
- 还有指针吗?

B 树隐含指针

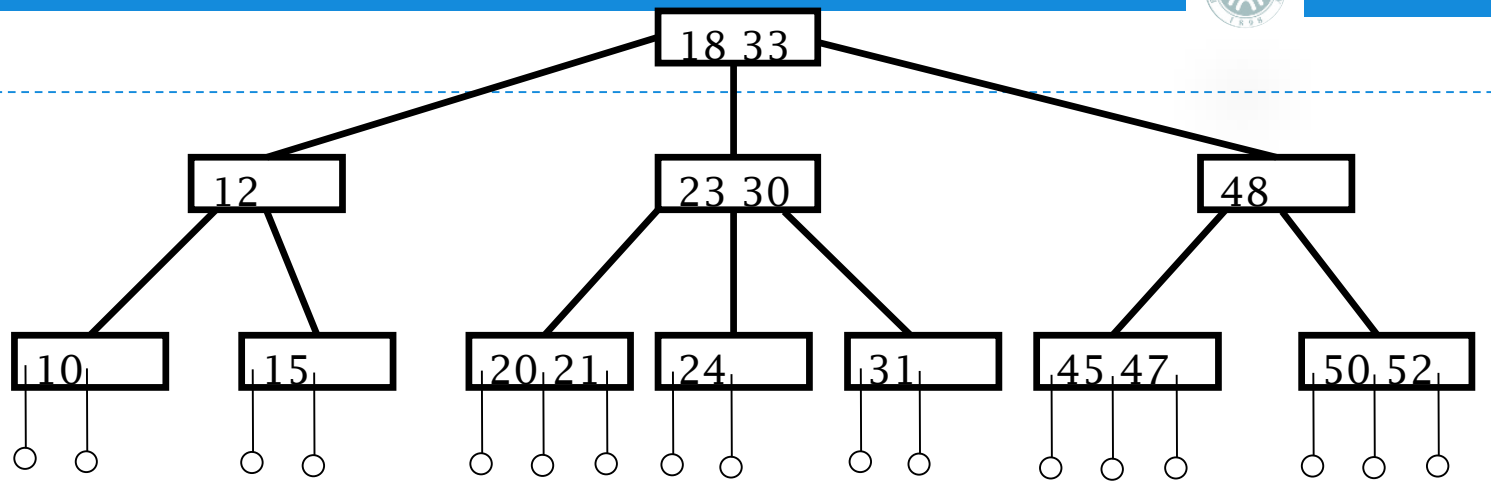
（关键码，文件页内地址）



2-3 树 = 3 阶 B 树（阶应该 ≥ 3）



B 树的查找



- 交替的两步过程
  - 1. 把根结点读出来，在根结点所包含的关键码 K_1，…，K_j 中查找给定的关键码值
    - 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

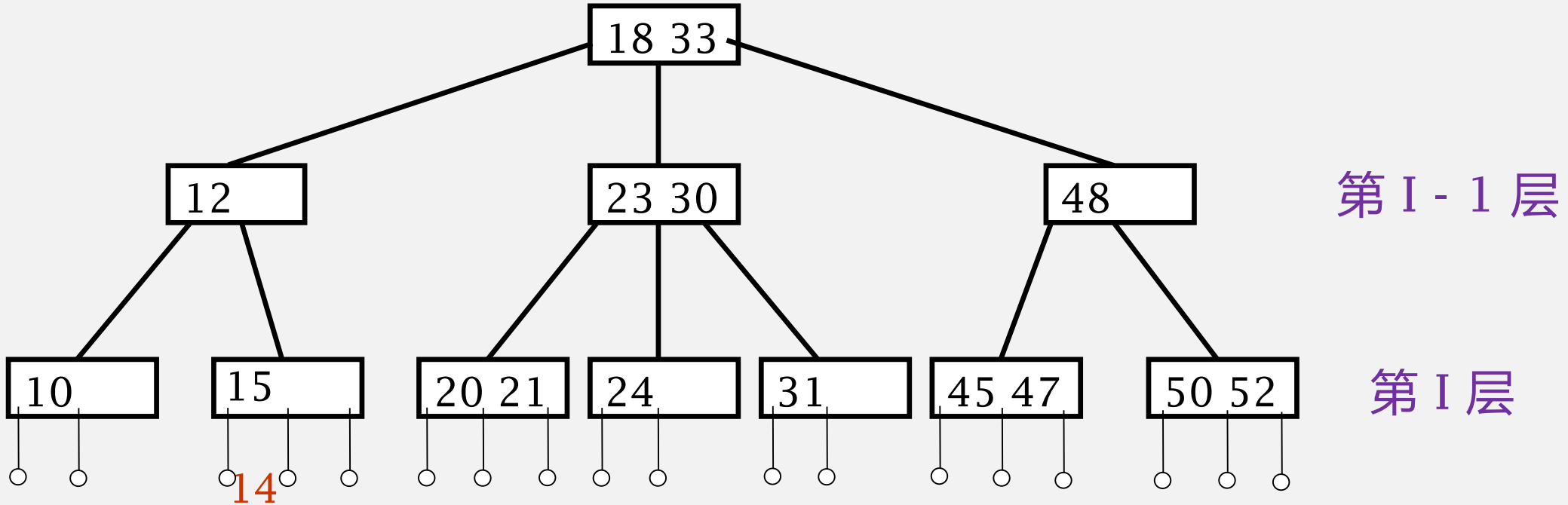
B 树插入

- 注意保持性质，特别是等高和阶的限制
  - 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 树的插入

- 外部空结点（即失败检索）处在第 I 层的 B 树，插入的关键码总是在第 I-1 层

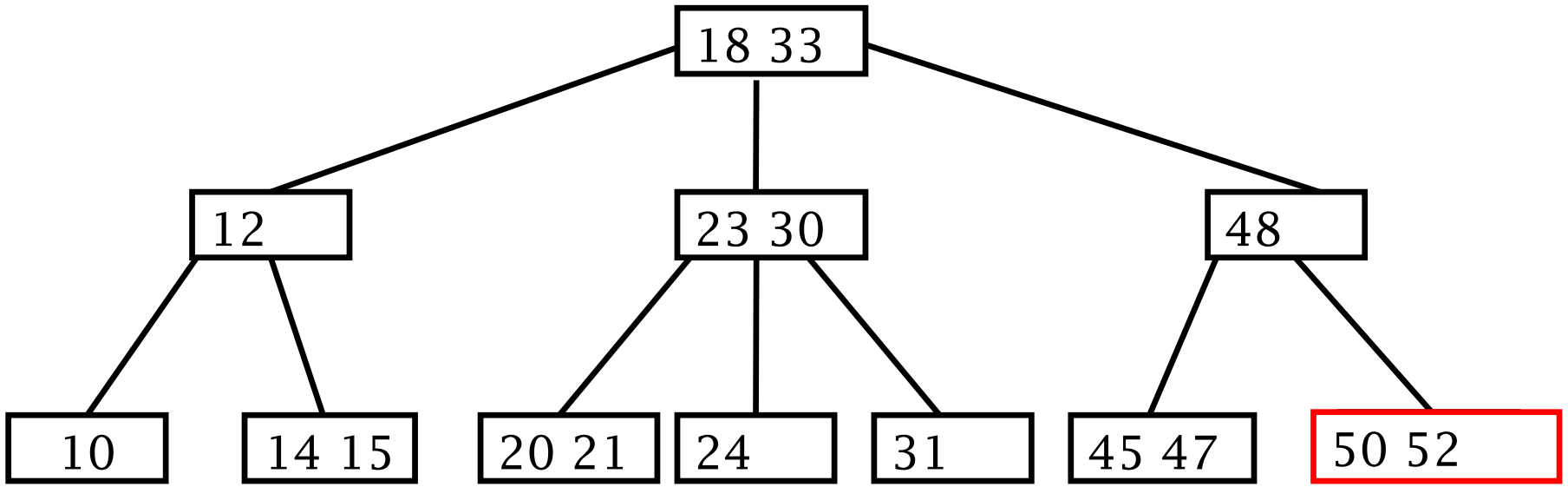
3阶B树 插入14



第 I - 1 层

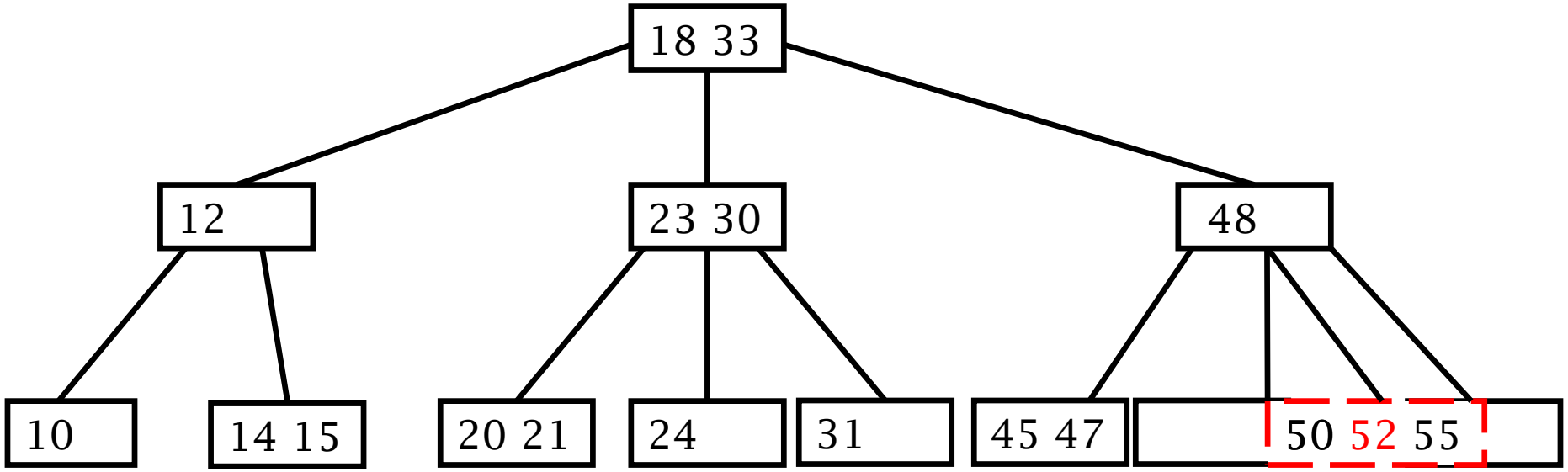
第 I 层

m=3，插入 55

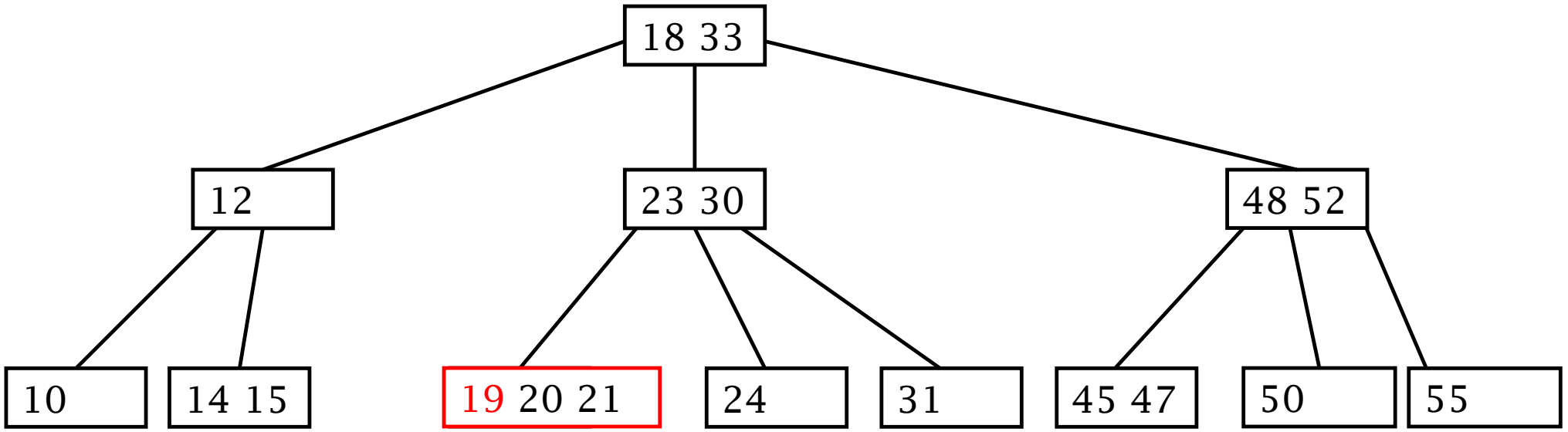


55

m=3，叶结点分裂，把 52 提升到父结点

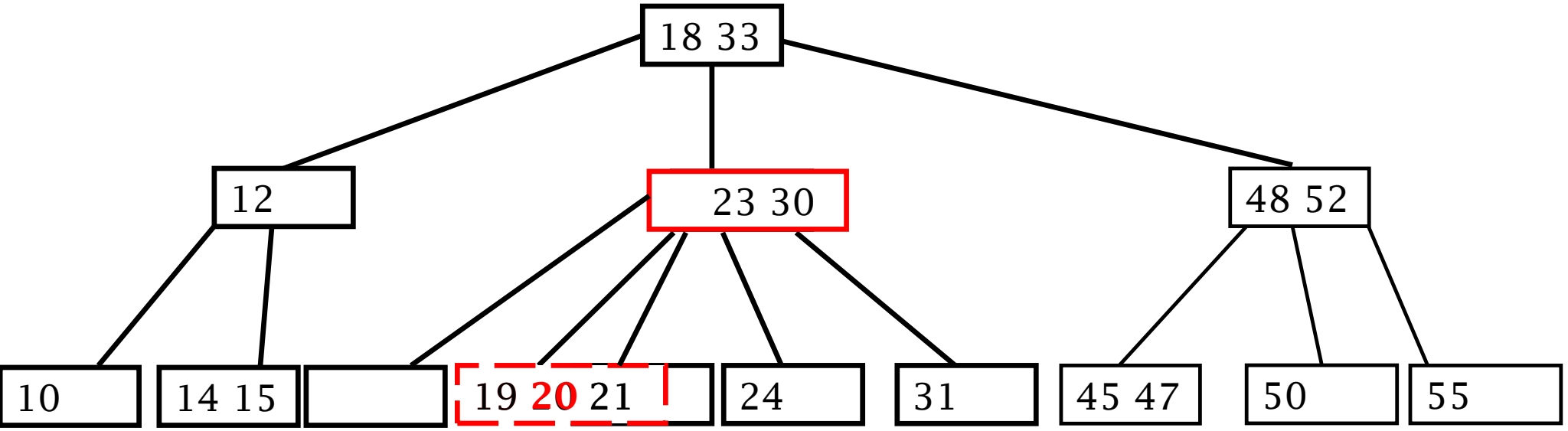


插入 19, 引起 3 阶 B 树根结点分裂

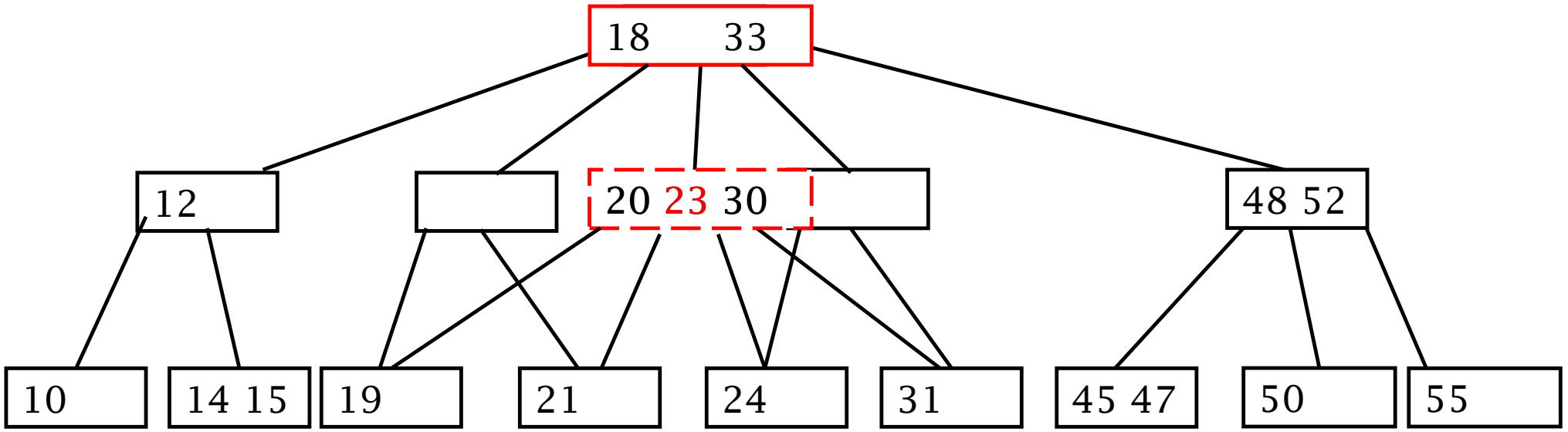


19

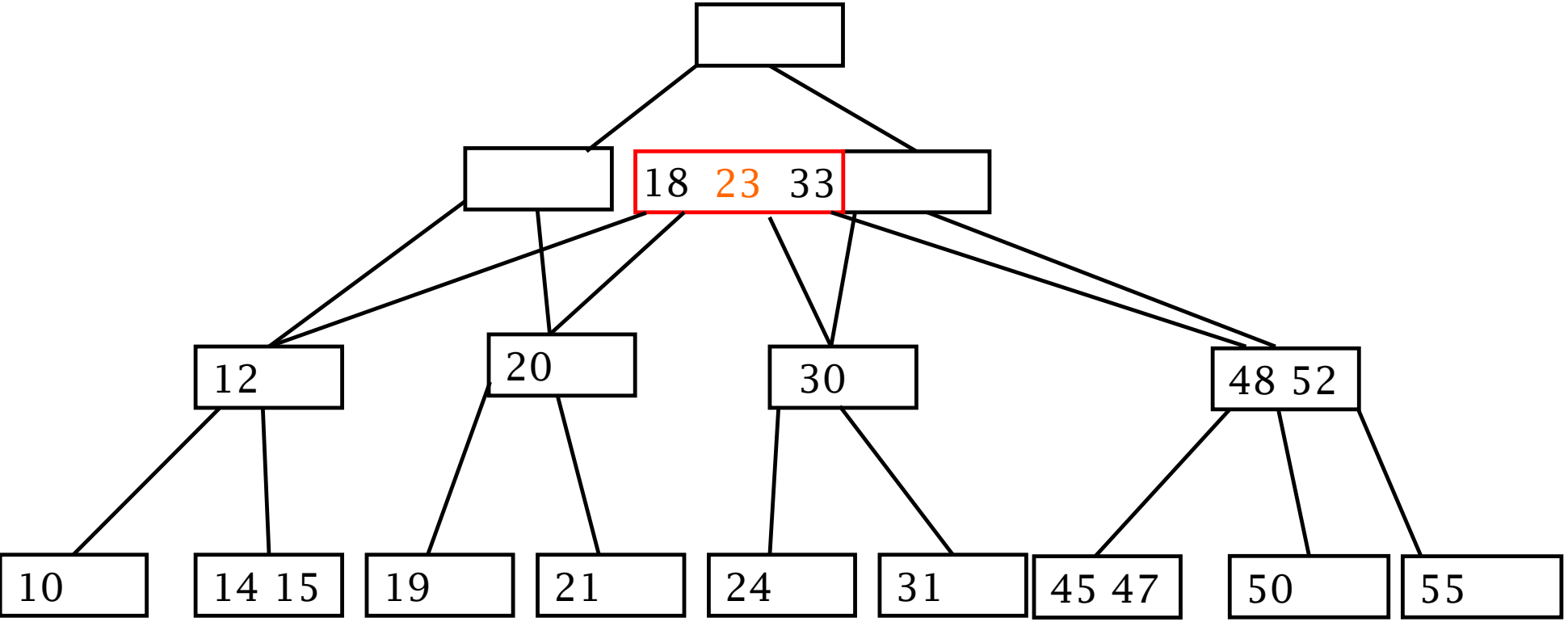
m=3，叶结点分裂



m=3，第二层结点分裂

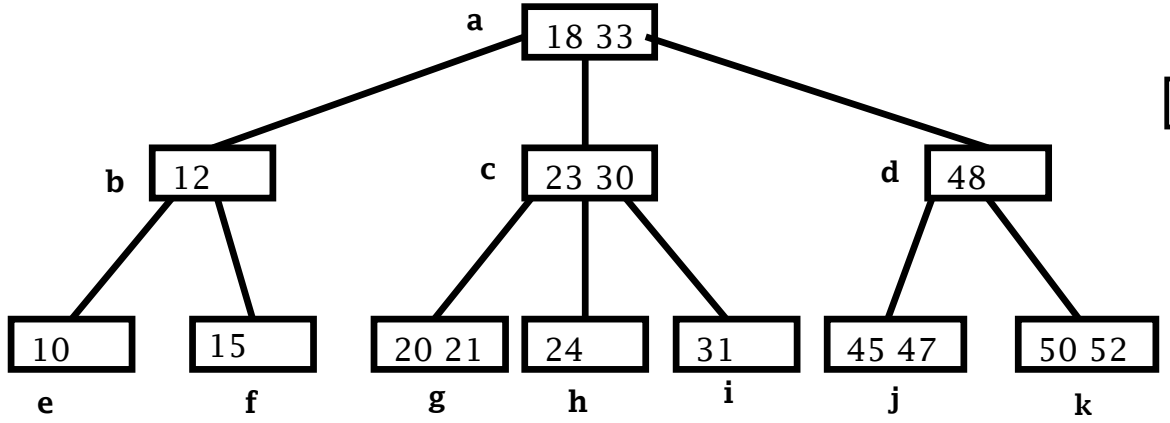


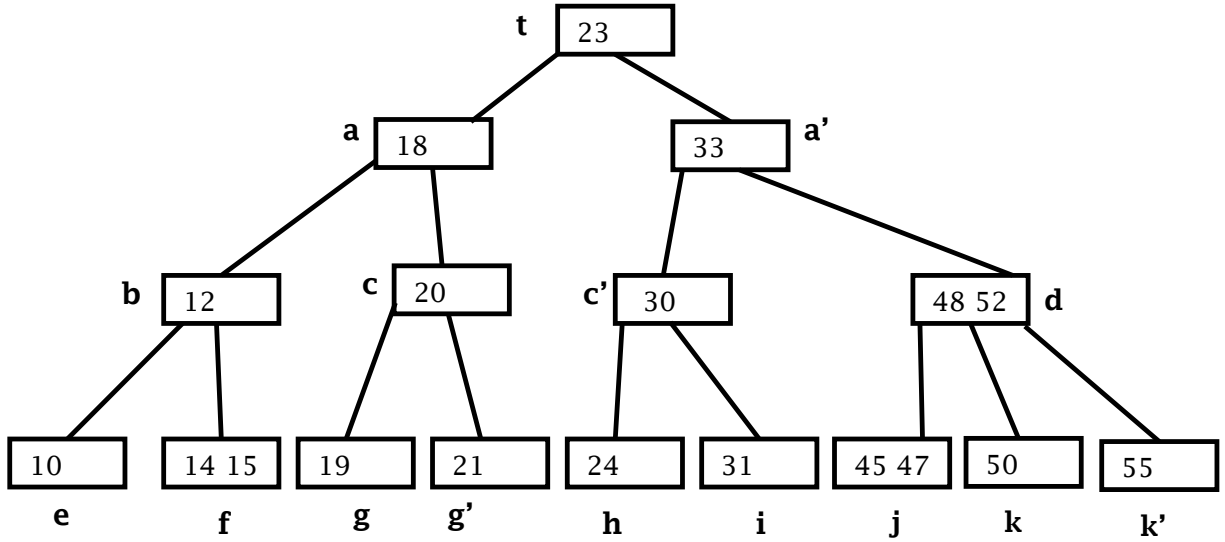
m=3，根结点分裂



B 树操作的访外次数

- 连续插入14、55、19，假设访问过的都缓存
- 读盘7次（a,b,f; d,k; c,g)
- 写盘11次（f; k,k’,d; g,g’,c,c’,a,a’,t)

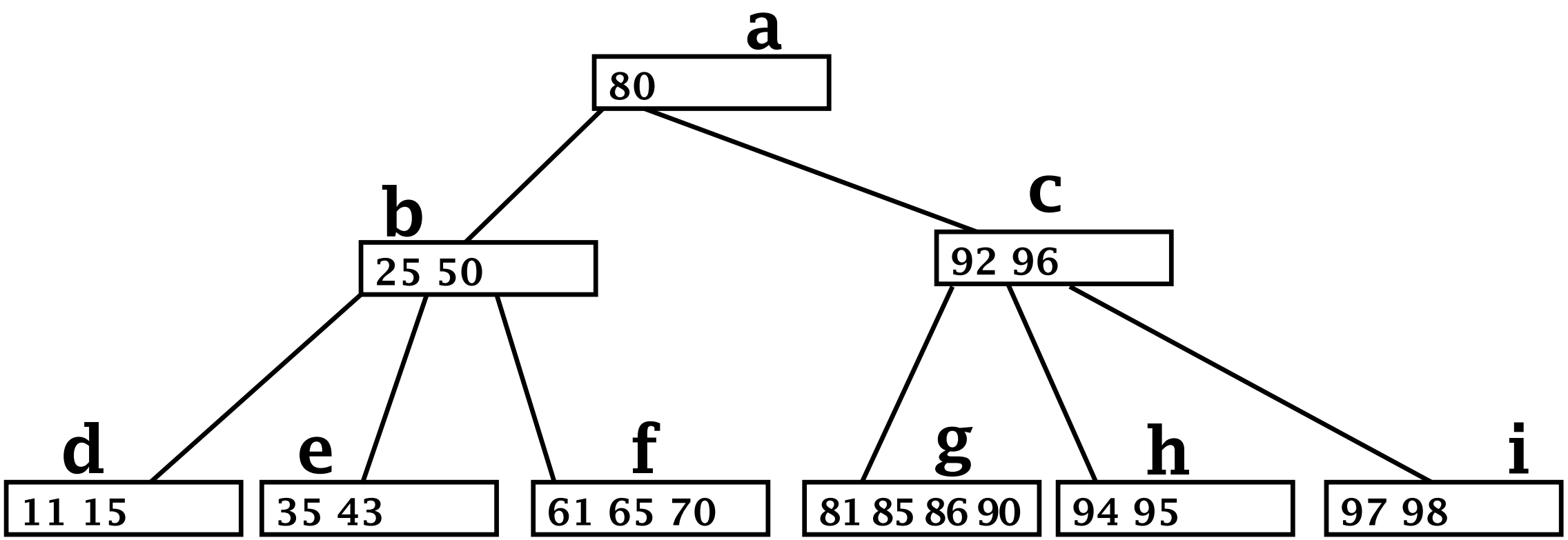




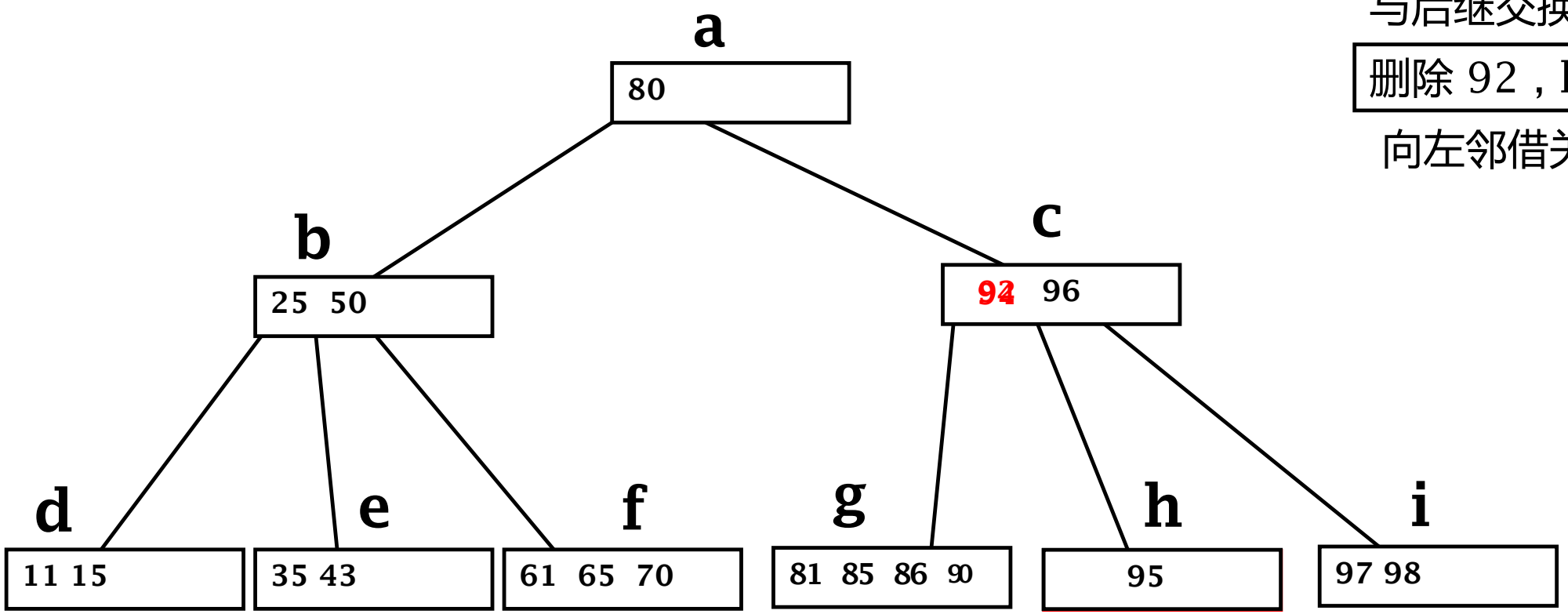
B 树的删除

- 删除的关键码不在叶结点层，跟叶中后继对换
- 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
  - 关键码个数小于 $\lceil m/2 \rceil - 1$
    - 如果兄弟结点关键码个数不等于 $\lceil m/2 \rceil - 1$
      - 从兄弟结点移若干个关键码到该结点中来(父结点中的一个关键码要做相应变化)
    - 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



5 阶 B 树删除示例



与后继交换

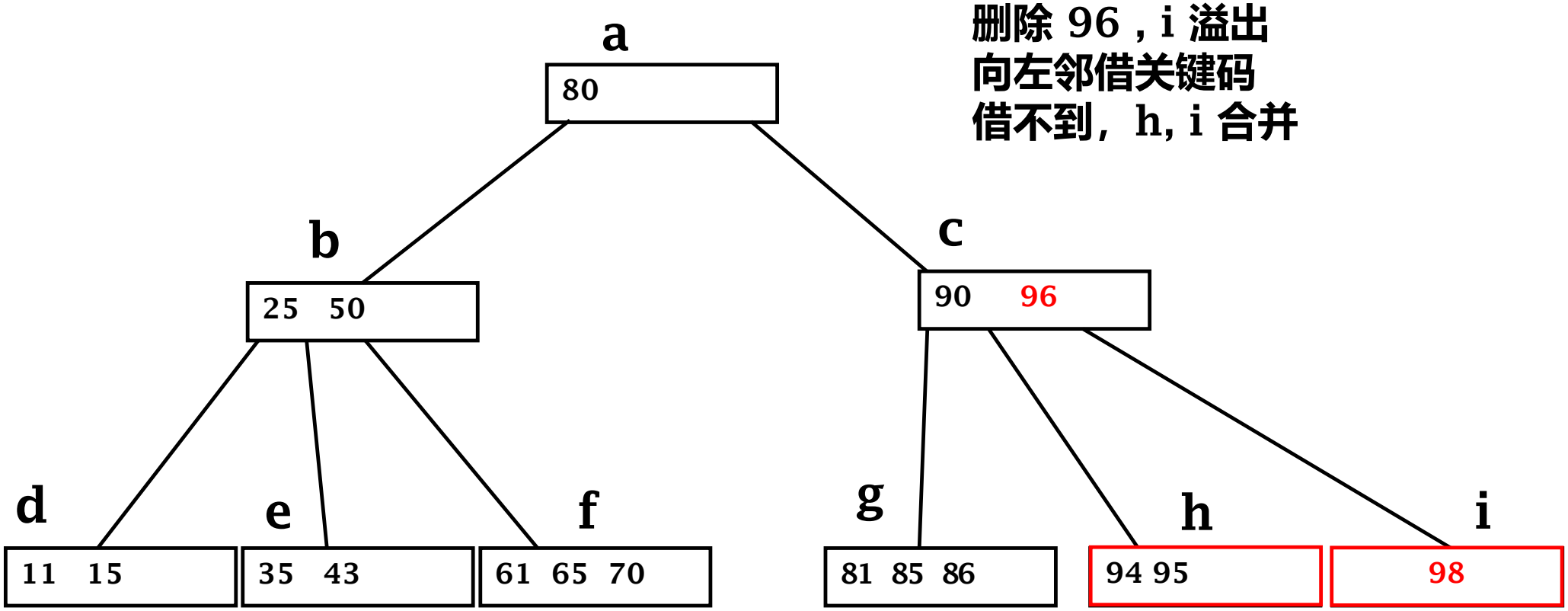
删除 92，h 溢出

向左邻借关键码

删除92，先与后继交换，删后下溢出，向左邻借关键码

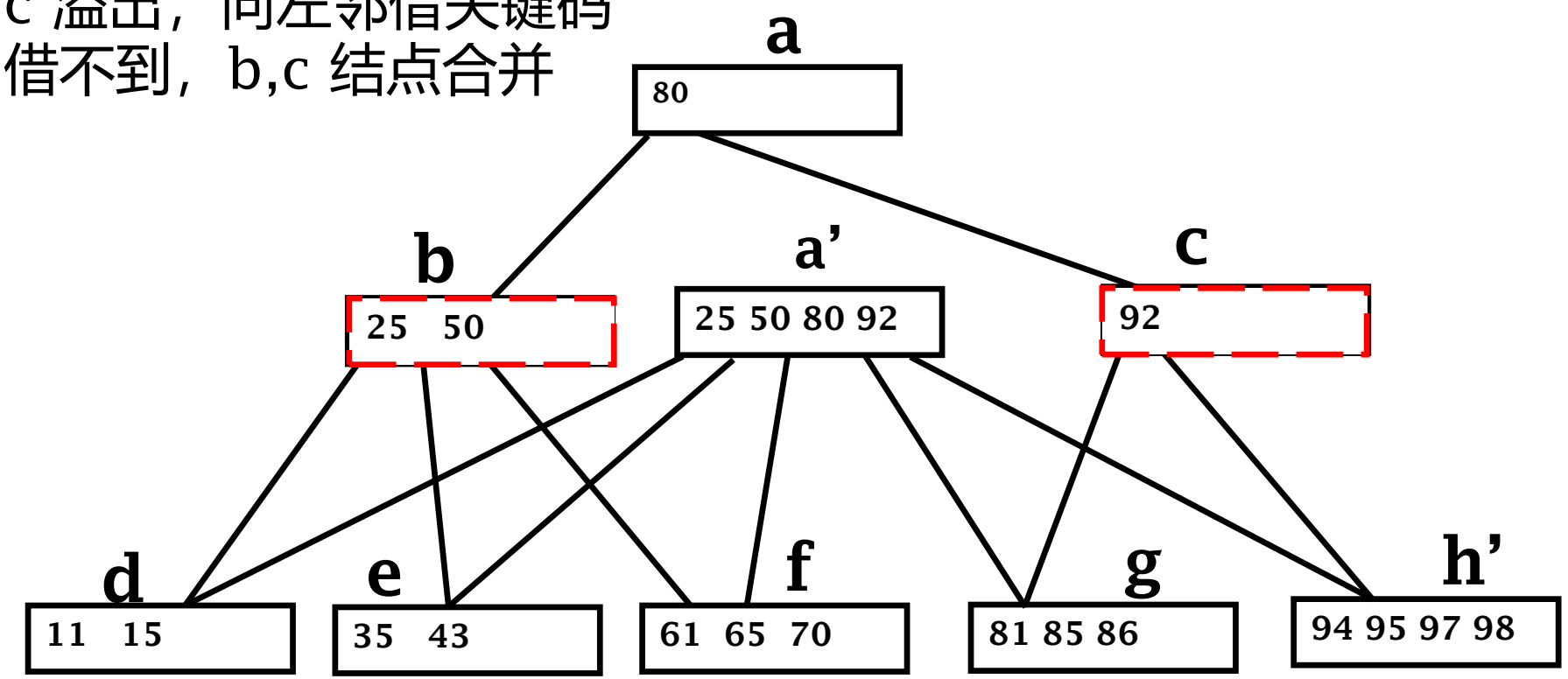
5 阶 B 树删除示例

继续删除 96
与后继交换
删除 96，i 溢出
向左邻借关键码
借不到，h, i 合并

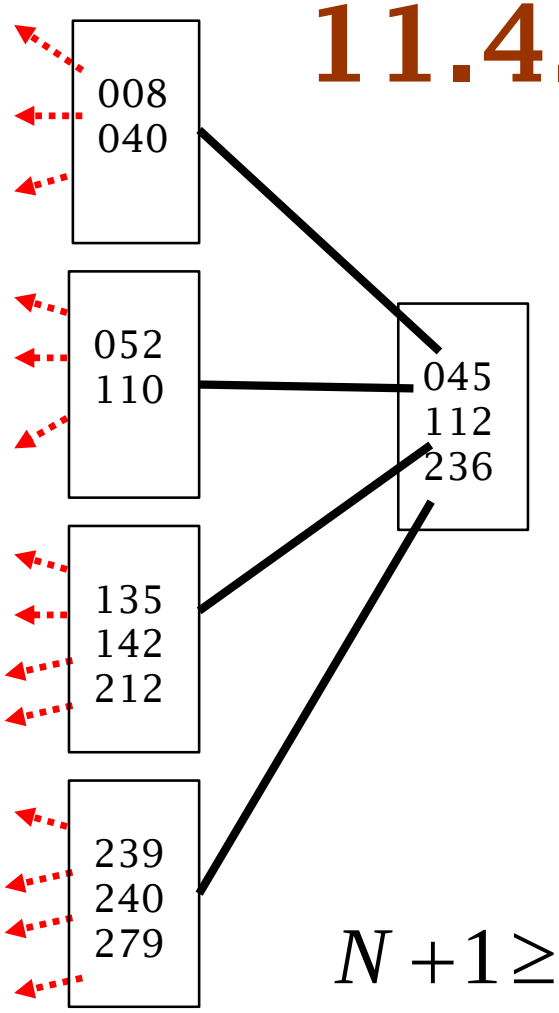


5 阶 B 树删除示例

h, i 合并成为 h’
c 溢出，向左邻借关键码
借不到，b,c 结点合并



11.4.2 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 层为根，第一层至少两个结点，
  - 第二层至少 $2 \cdot \lceil \frac{m}{2} \rceil$ 个结点，
  - 第 k 层至少 $2 \cdot \lceil \frac{m}{2} \rceil^{k-1}$ 个结点，

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

示例

- N=1,999,998，m=199 时

$$k \le 1 + \log_{\lceil m/2 \rceil}(\frac{N+1}{2})$$

  - k=4
  - 一次检索最多 4 层

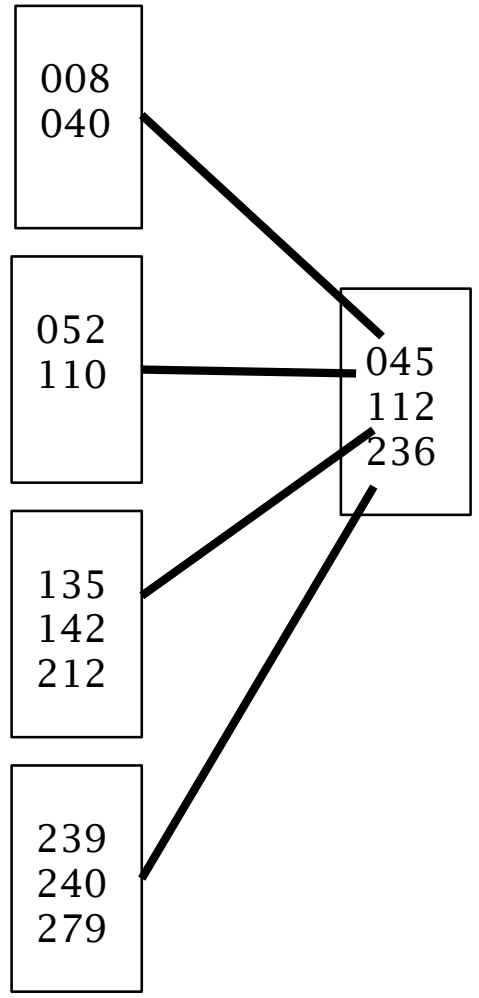
结点分裂次数

- 设关键码数为 N (空指针 N+1)，内部结点数为 p

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

- 最差情况下每插入一个结点都经过分裂(除第一个)，即 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 $p-1 \leq \frac{N-1}{\lceil m/2 \rceil -1}$

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



思考

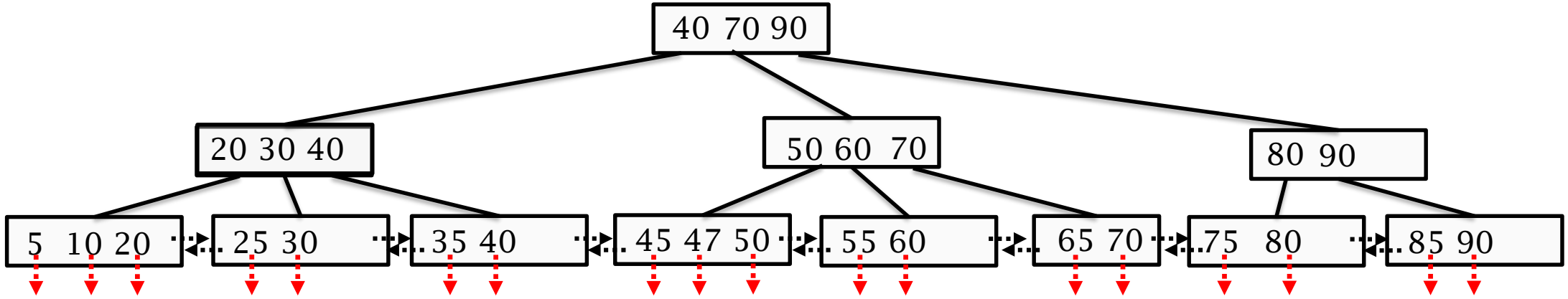
- 1. 是否存在符合定义的 2 阶 B 树？是否有实用价值？为什么？
- 2. B 树删除时使用先借用再合并的方法，为何在插入的时候不使用先送给兄弟结点再考虑分裂的方法？
- 3. B 树的定义中关于度数的定义为从$\left\lceil \frac{m}{2} \right\rceil$到$m$之间，是否可以调整为其他范围？

主要内容

- 11.1 线性索引
- 11.2 静态索引
- 11.3 倒排索引
- 11.4 动态索引
  - 11.4.1 B 树
  - 11.4.2 B 树的性能分析
  - 11.4.3 B+ 树
  - 11.4.4 B 树、B+ 树索引性能的比较
- 11.5 位索引技术
- 11.6 红黑树

11.4.3 B+ 树

- 是B 树的一种变形，在叶结点上存储信息
  - 所有的关键码均出现在叶结点上
  - 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+ 树的结构定义

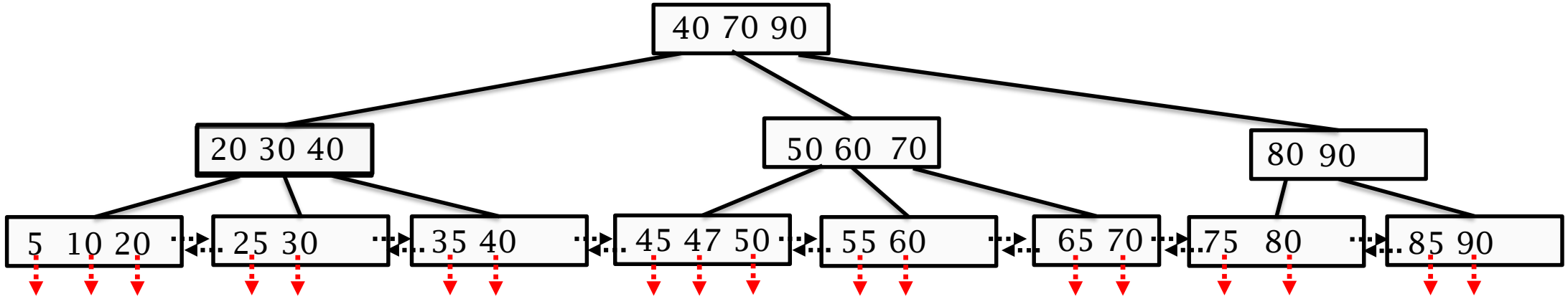
m 阶 B+ 树的结构定义如下：

(1) 每个结点至多有 m 个子结点

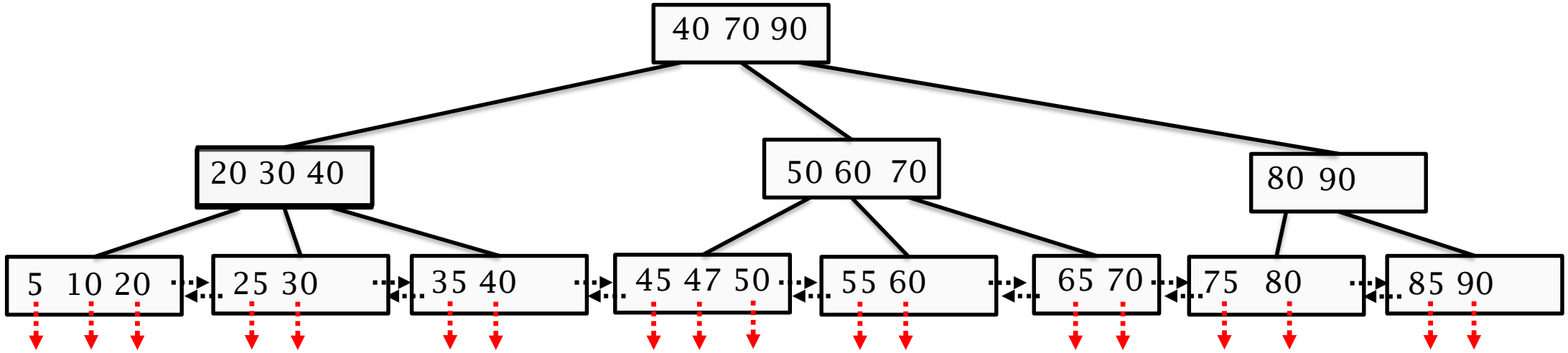
(2) 每个结点(除根外)至少有 $\lceil m/2 \rceil$ 个子结点

(3) 根结点至少有两个子结点

(4) 有 k 个子结点 的结点必有 k 个关键码



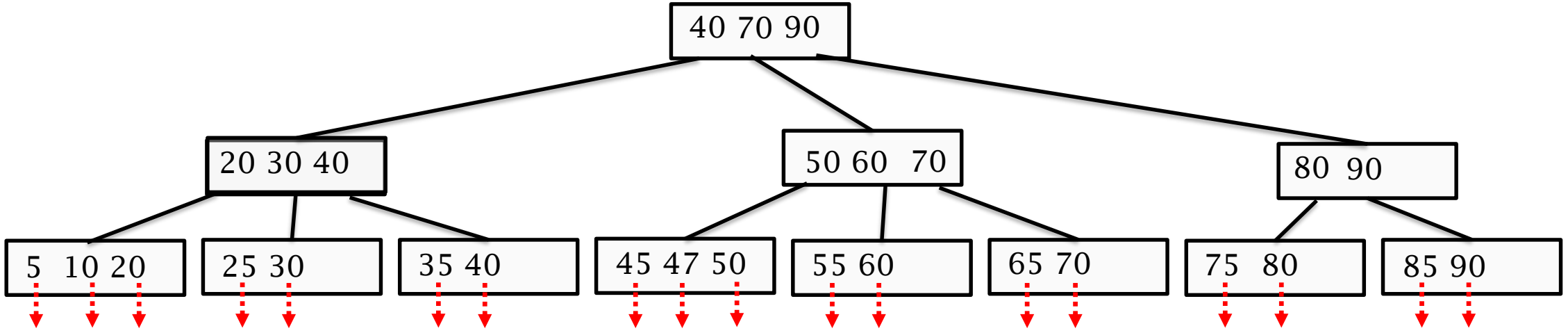
3 阶 B+ 树的例子（一般阶 ≥ 3）



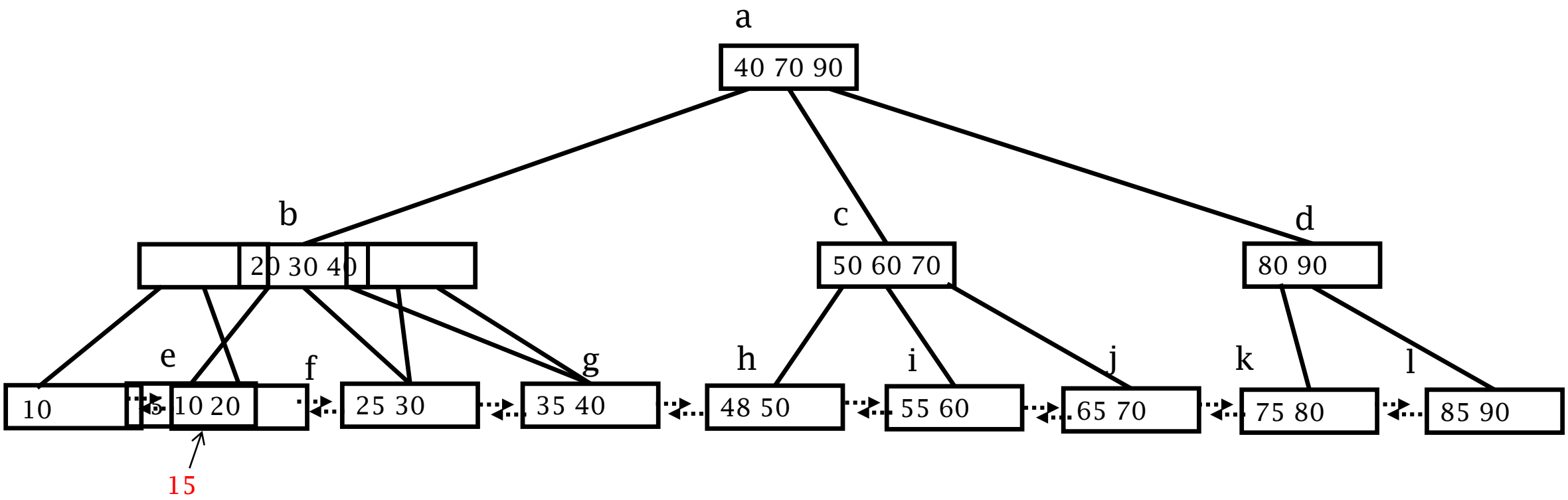
- B+ 树的查找
  - 在上层已找到待查的关键码，并不停止
  - 而是继续沿指针向下一直查到叶结点层的这个关键码

B+ 树的插入

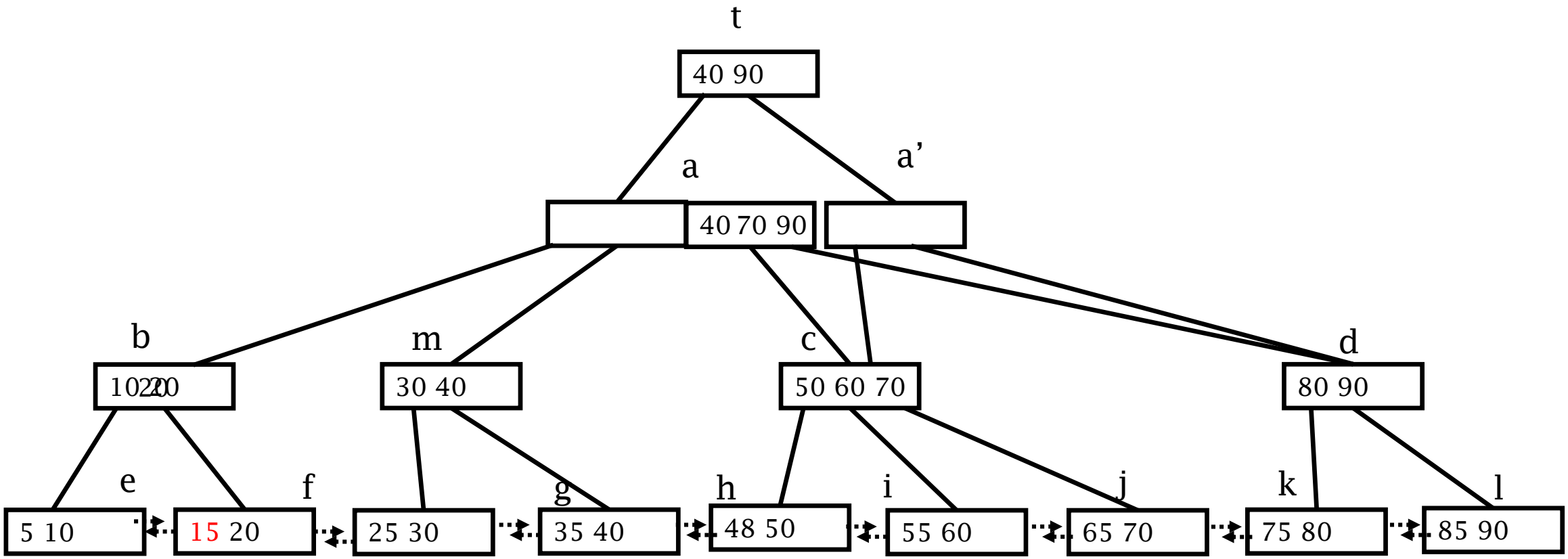
- 插入——分裂
  - 过程和 B 树类似
  - 注意保证上一层结点中有这两个结点的最大关键码（或最小关键码）



3 阶 B+ 树插入15

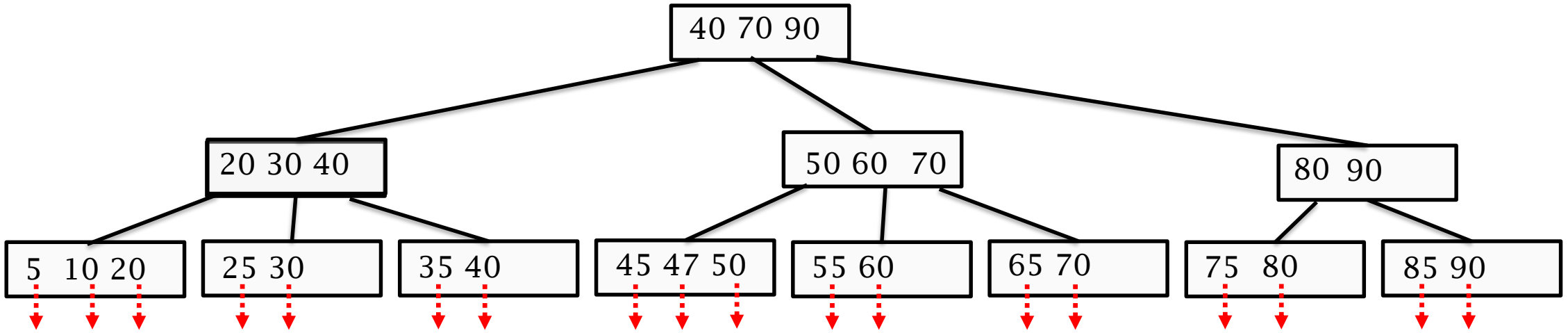

a
40 70 90
b
20 30 40
c
50 60 70
d
80 90
e
f
g
h
i
j
k
l
10
10 20
25 30
35 40
48 50
55 60
65 70
75 80
85 90
15

在上图 3 阶 B+ 树中插入 15 后，树增高一层



B+ 树的删除

- 当关键码下溢出时，与左或右兄弟进行调整（甚至合并）
- 关键码在叶结点层删除后，其在上层的复本可以保留，作为一个“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）



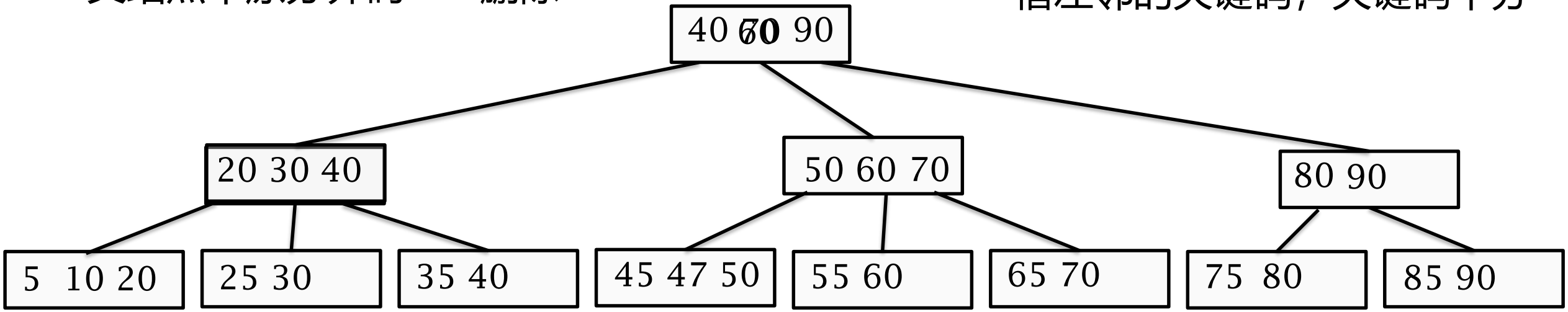
11.4.3 B+ 树

在 3 阶 B+ 树中删除 75

删去 75，发生下溢出，剩余关键码 80 与右邻结点合并为新结点（80，85，90）
父结点中原分界码 80 删除

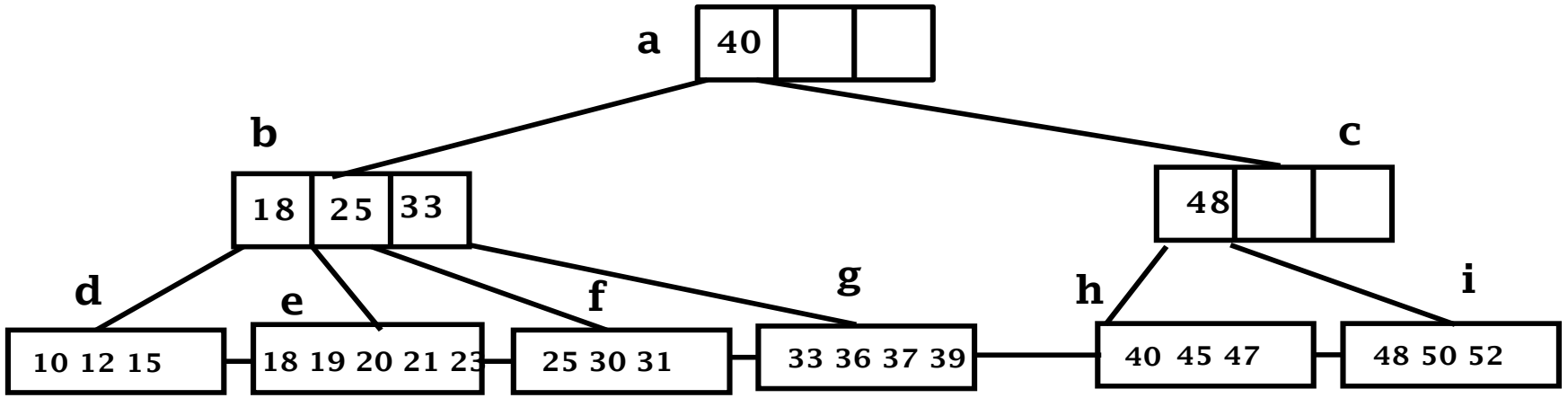
根结点中的分界码70修改为60

父结点下溢出
借左邻的关键码，关键码平分

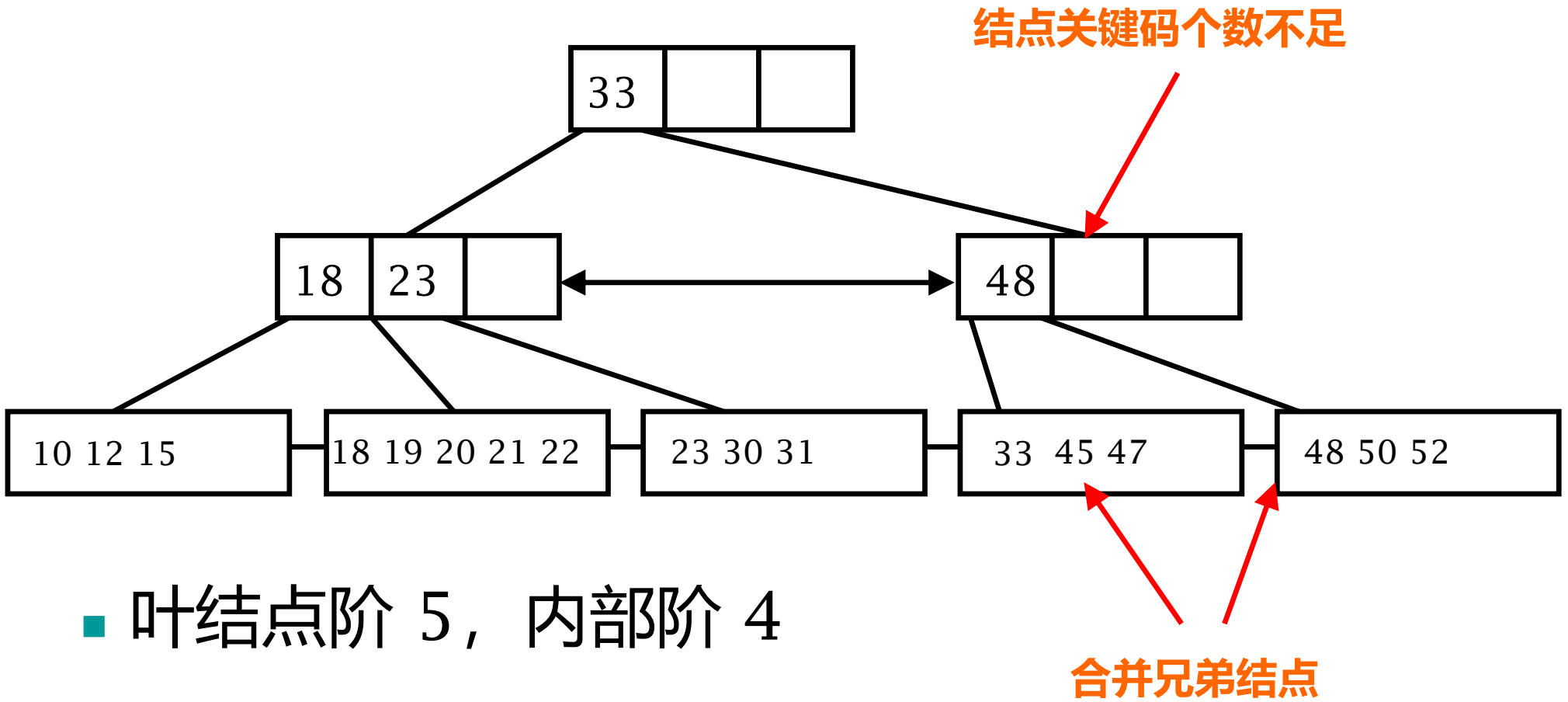


另一种 B+ 树

- 叶结点中关键码数目与非叶的不同
  - 内部非叶结点构成 B 树
  - 叶的阶与 B+ 树一致
  - 例如，叶结点阶 5，内部阶 4

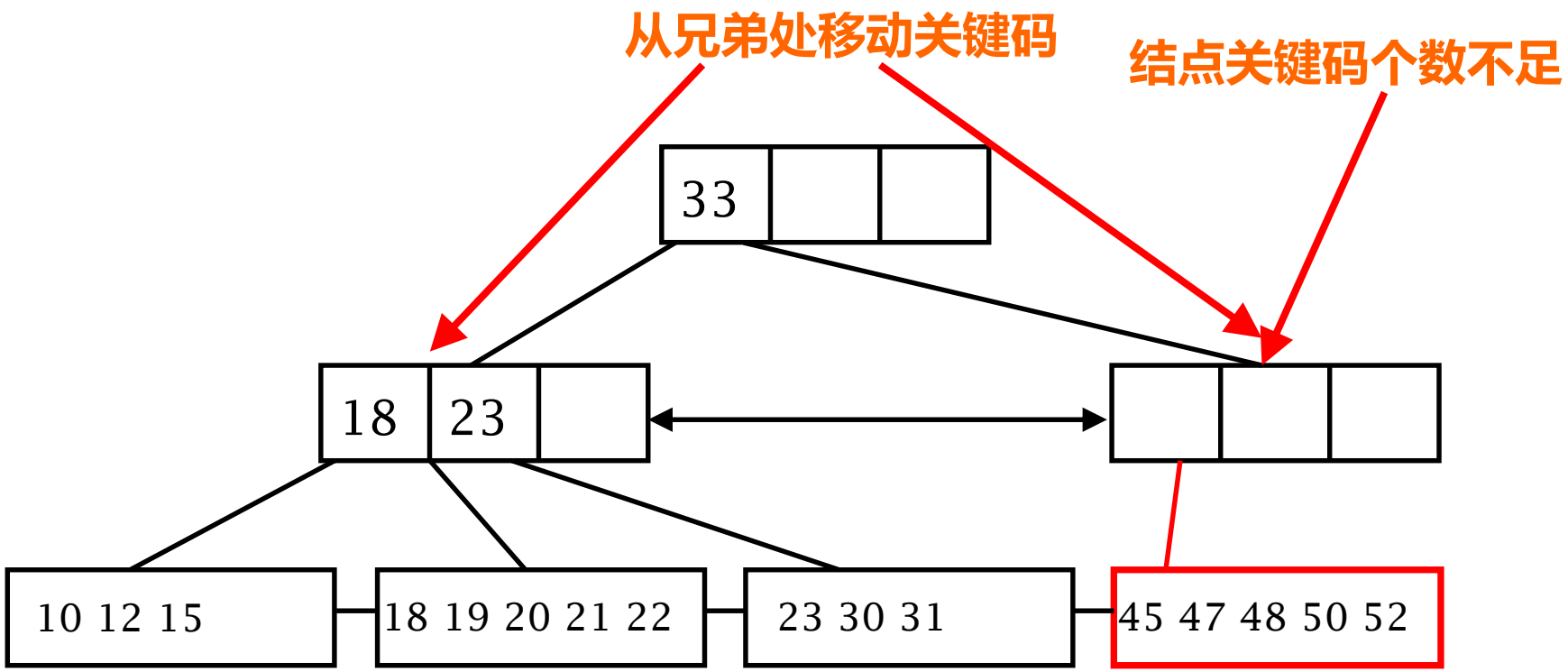


从 B+ 树删除关键码值为 33 的记录



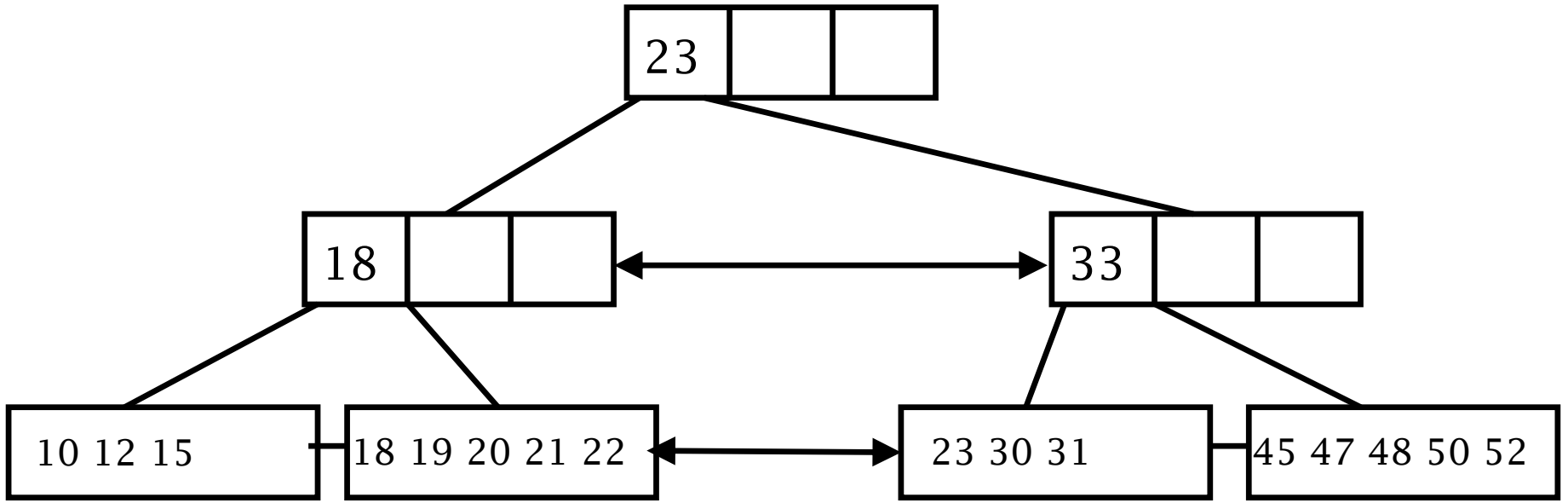
- 叶结点阶 5，内部阶 4

从 B+ 树删除关键码值为 33 的记录



- 叶结点阶 5，内部阶 4

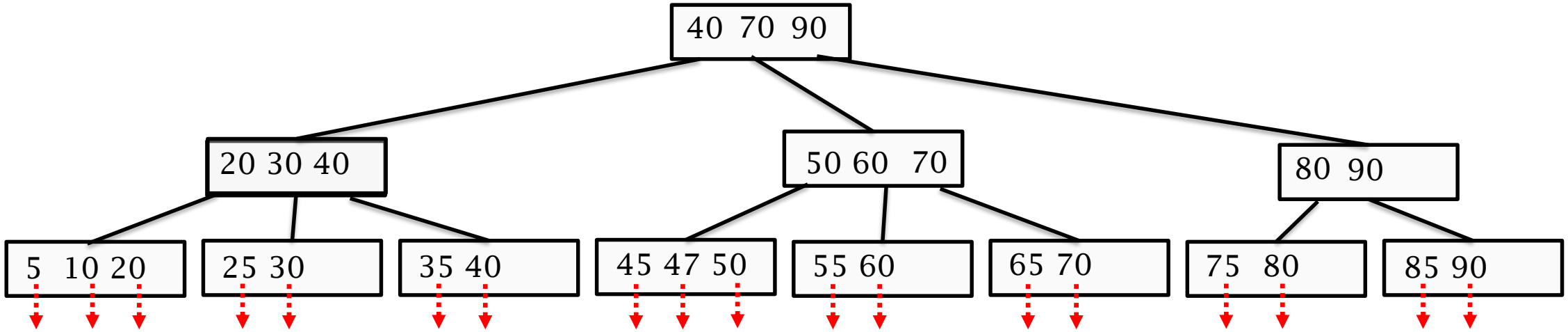
从 B+ 树删除关键码值为 33 的记录



- 叶结点阶 5，内部阶 4

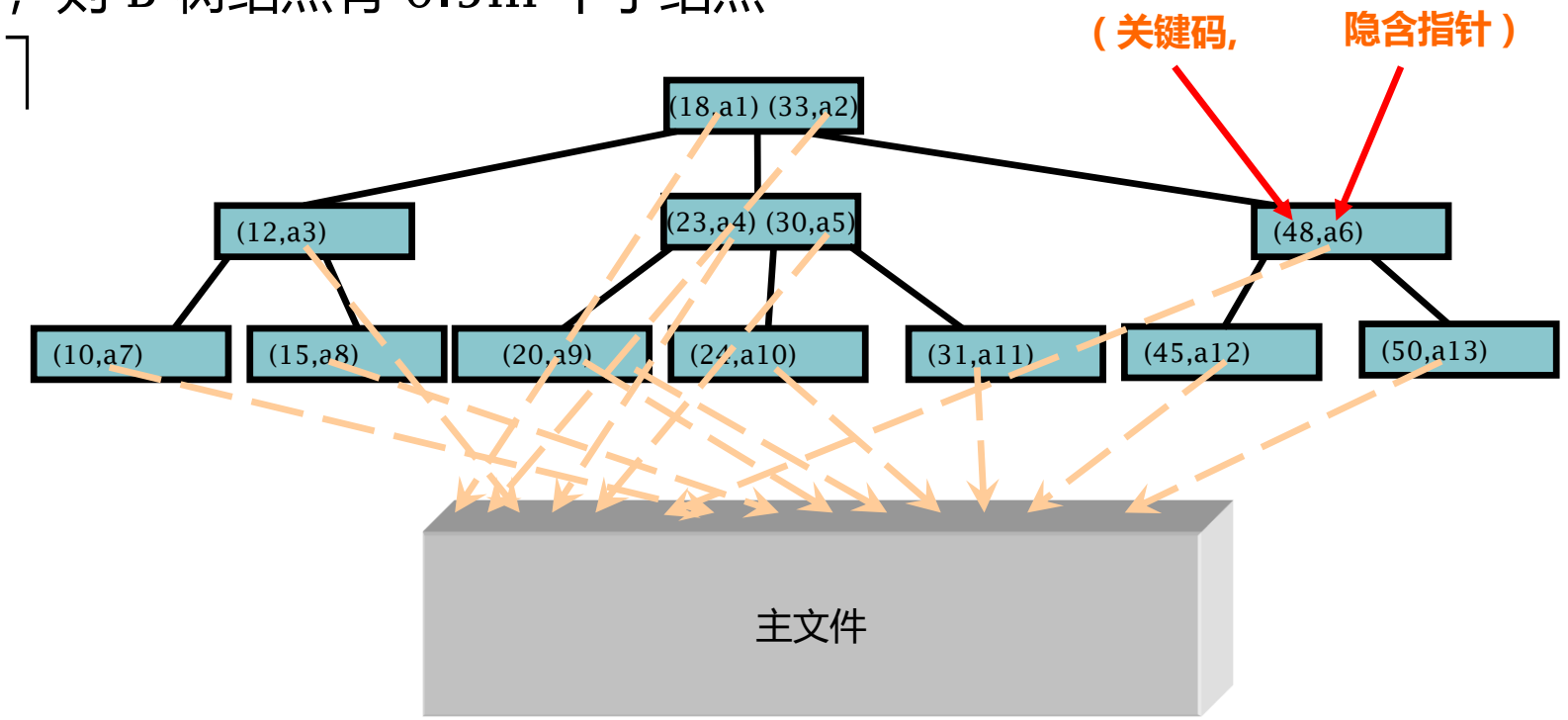
B+ 树的存储效率实际上更高

- 假设一个主文件有 N 个记录，假设一个页块可以存 m 个(关键码，子结点页块地址)二元对
- 假设 B+ 树平均每个结点有 0.75m 个子结点
  - 充盈度为 (1+0.5)/2 = 75%
- 因此 B+ 树的高度为 $\lceil \log_{0.75m} N \rceil$



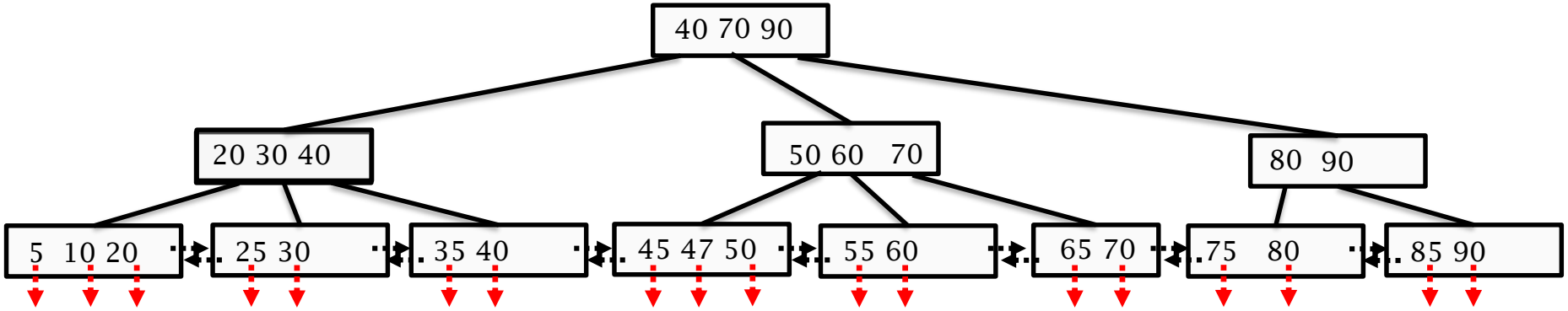
11.4.4 B 树 B+ 树性能比较

- 可以容纳 m 个(关键码，子结点页块指针)，假设关键码所占字节数与指针相同
  - 可以容纳 B 树的(关键码，隐含指针，子结点页块指针)最多为 2m/3 (B 树为 0.67m 阶)。
- 假设 B 树充盈度也是 75%，则 B 树结点有 0.5m 个子结点
- B 树的高度为 $\lceil \log_{0.5m} N \rceil$

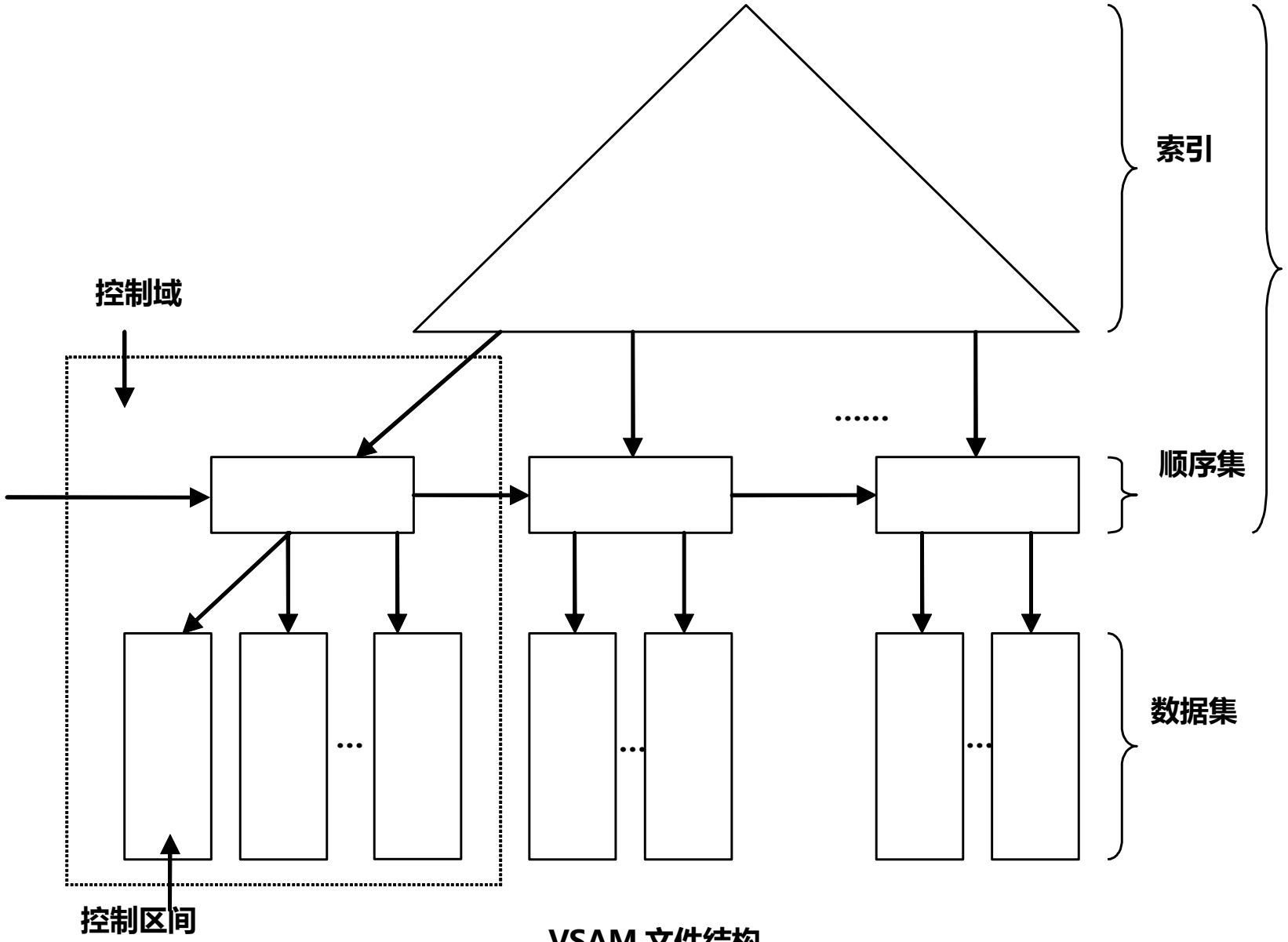


B+ 树应用得更为广泛

- B+ 树的存储效率更高、检索层次更少（树较矮）
- 因此，B+ 树应用得更为广泛
  - 数据库系统主码（primary key）索引
  - 基于B+树的磁盘文件虚拟存储存取管理 VSAM (Virtual Storage Access Method)，取代了基于多分树的 ISAM



VSAM 的组成



VSAM 文件结构

思考

- 1. 是否存在 2 阶 B^+ 树?
- 2. 为什么相比于 B^+ 树，B 树存储效率低?
- 3. 查阅数据库的相关文献，看看 B+ 树的作用。

主要内容

- 基本概念
- 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 位索引技术
- 11.6 红黑树

11.5 位索引技术

- B 树适合于查找并取回少量记录的情况
- 对于数据仓库的复杂交互式查询，B树有三个缺点：
  1. B 树对唯一值极少的（低基数）数据字段几乎毫无价值
  2. 在数据仓库中构造和维护索引的代价高
  3. 对于带有分组及聚合条件的复杂查询无能为力

对于数据库表的位图索引

date	store	state	class	sales
3/1	32	NY	A	6
3/1	36	AL	A	9
3/1	38	NY	B	5
3/1	41	AK	A	11
3/1	43	NY	A	9
3/1	46	AK	B	3

State = NY

1
0
1
0
1
0

Class=A

1
1
0
1
1
0

state=AK	state=AL	…	state=NY
0	0		1
0	1		0
0	0		1
1	0		0
0	0		1
1	0		0

长为 n 的位向量的集合
（ n 为文件的记录数）

特征文件

Signature file （也译为“签名文件”）

- 记录30：foo, bar, baz
- 记录40：baz, bar
- 记录50：foo

记录	bar	baz	foo
30	1	1	1
40	1	1	0
50	0	0	1

位图索引特点

1. 按“列”为单位存储数据
2. 列数据比行数据更易进行压缩，可节省 50% 的磁盘空间
3. 索引空间比 B 树小

思考

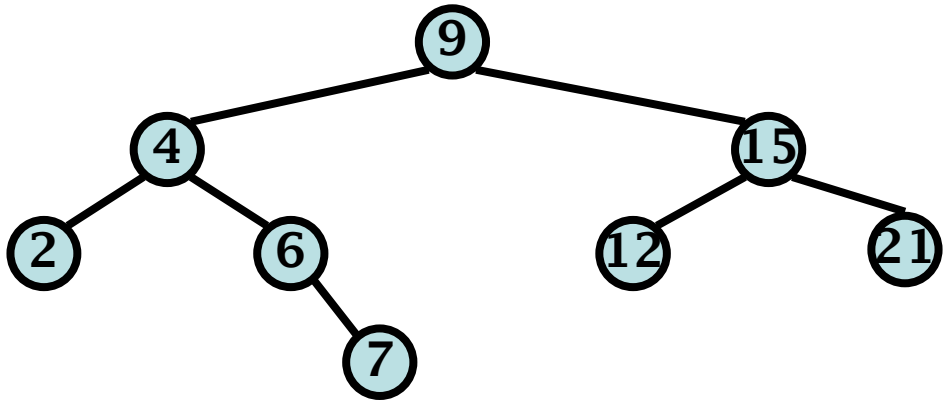
- 调研列数据库中的位图索引

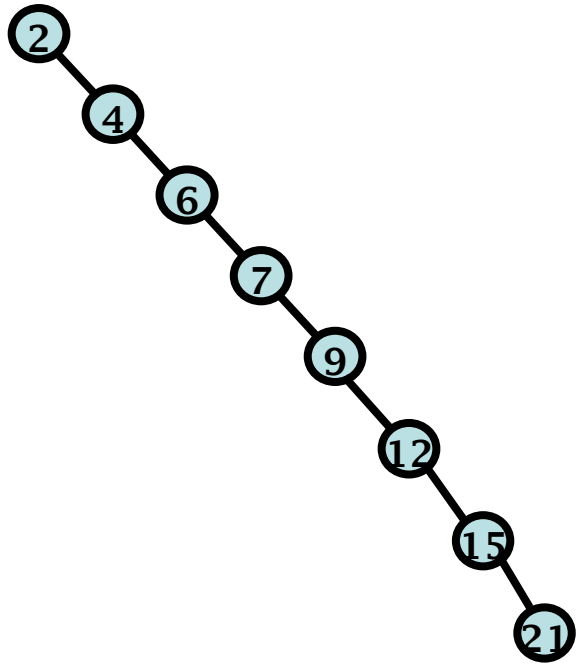
主要内容

- 基本概念
- 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 位索引技术
- 11.6 红黑树
  - 拓展：红黑树与2-3-4树，AA树

BST的平衡问题

- 理想状况：插入、删除、查找时间代价为O($\log n$)
- 输入9,4,2,6,7,15,12,21
- 输入2,4, 6,7, 9, 12,15, 21



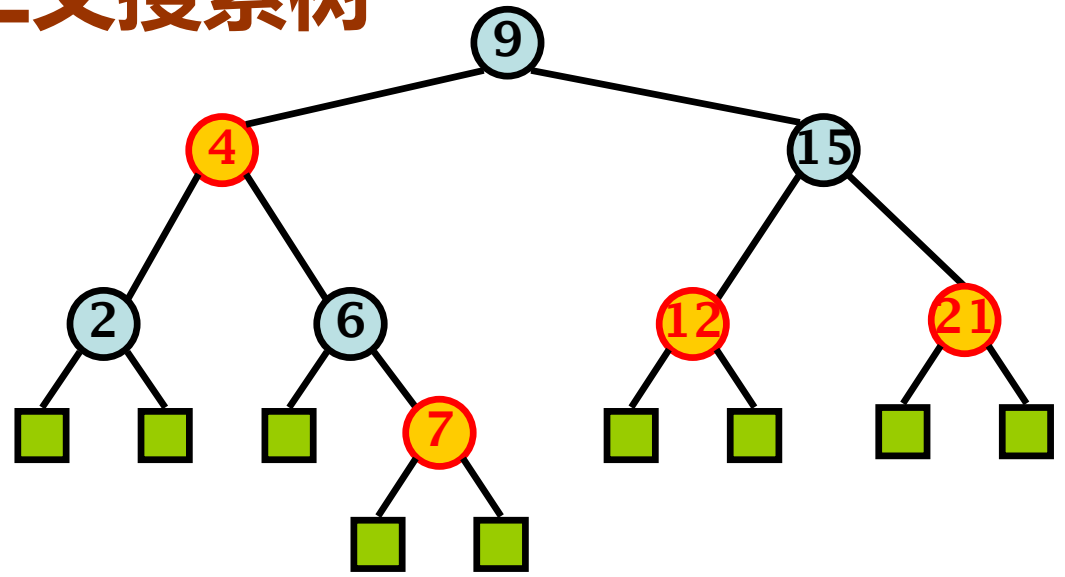


11.6 红黑树

- 11.6.1 红黑树定义:

  red-black tree, 简称RB-tree
- 11.6.2 红黑树相关性质
- 11.6.3 结点插入算法
- 11.6.4 结点删除算法

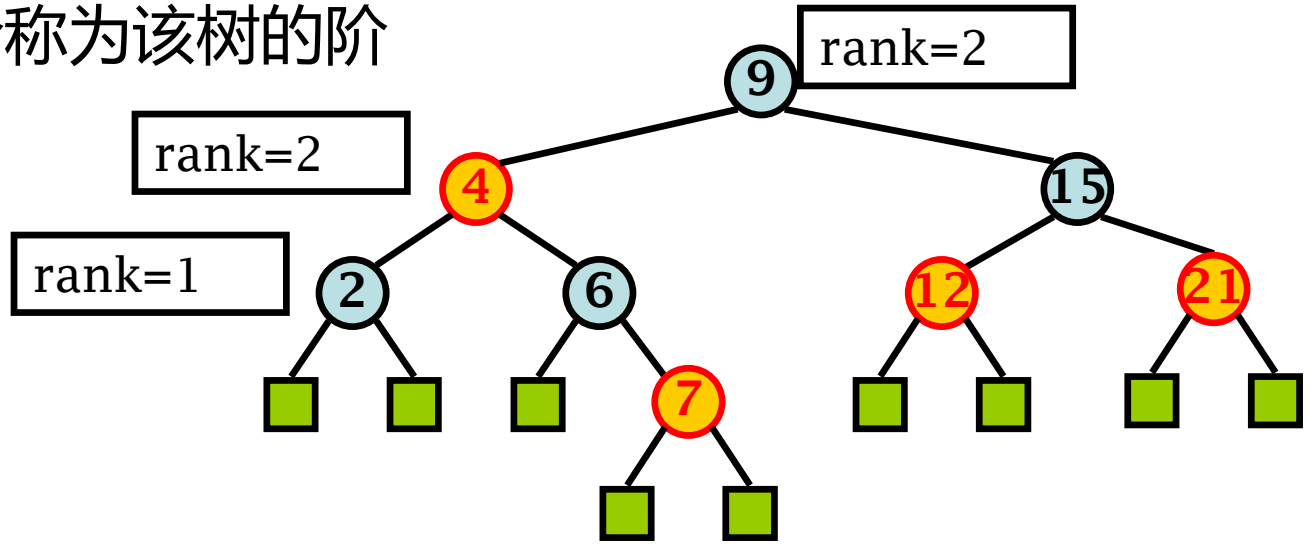
红黑树：平衡的 扩充 二叉搜索树



- 颜色特征：结点是“**红色**”或“**黑色**”；
- 根特征 ：根结点永远是“黑色”的；
- 外部特征：扩充外部叶结点都是空的“黑色”结点；
- 内部特征：“红色”结点的两个子结点都是“黑色”的，不允许两个连续的红色结点；
- 深度特征：任何结点到其子孙外部结点的每条简单路径都包含相同数目的“黑色”结点

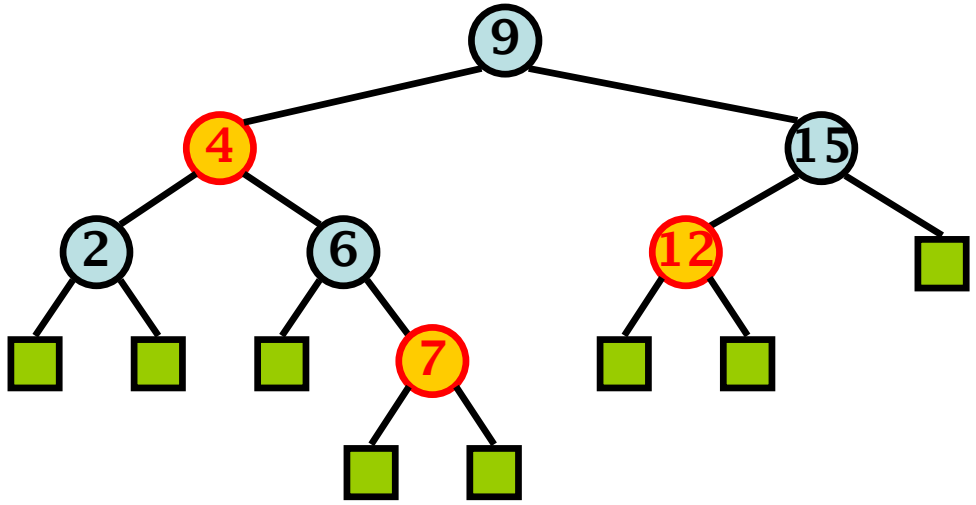
红黑树的阶

- 结点X的阶（rank，也称“黑色高度”，简称“**黑高**”）
  - 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
- 外部结点的阶是零
- 根的阶称为该树的阶

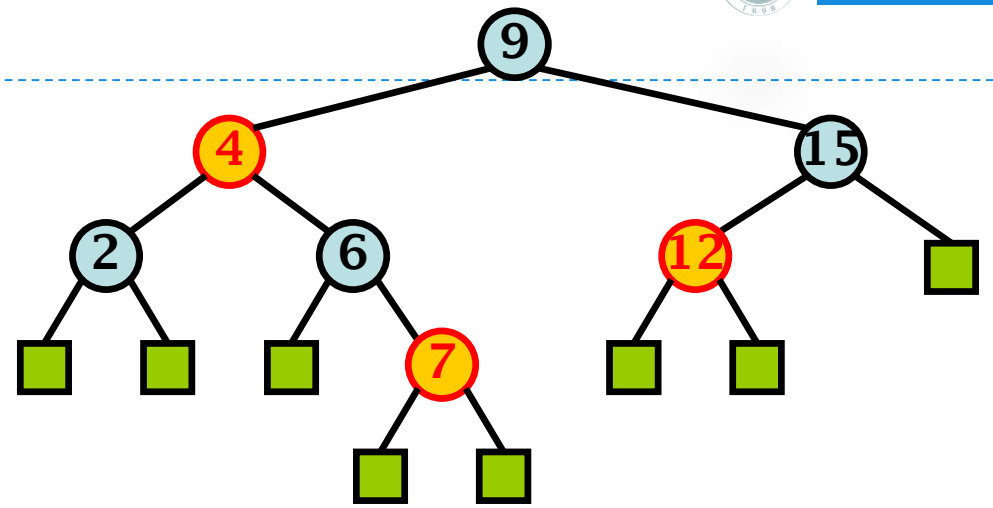


11.6.2 红黑树的性质

- (1) 红黑树是满二叉树　空叶结点也看作结点
- (2) 阶为 k 的红黑树路径长度 **最短是 k，最长是 2k**
  从根到叶的简单路径长度



红黑树的性质



- (2)' 阶为 k 的红黑树树高 **最小是 k+1，最高是 2k+1**
- (3) 阶为 k 的红黑树的内部结点

  最少是一棵完全满二叉树，内部结点数最少是 2^k-1
- (4) n 个内部结点的红黑树树高

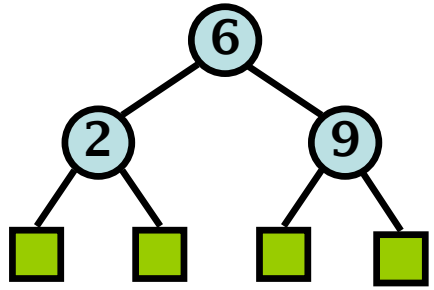
  最大是 $2\log_2(n+1)+1$
- **证明：**

  设红黑树的阶为 k，设红黑树的树高是 h。

  由性质（2）' 得 h <= 2k+1，则 k >= (h-1) / 2

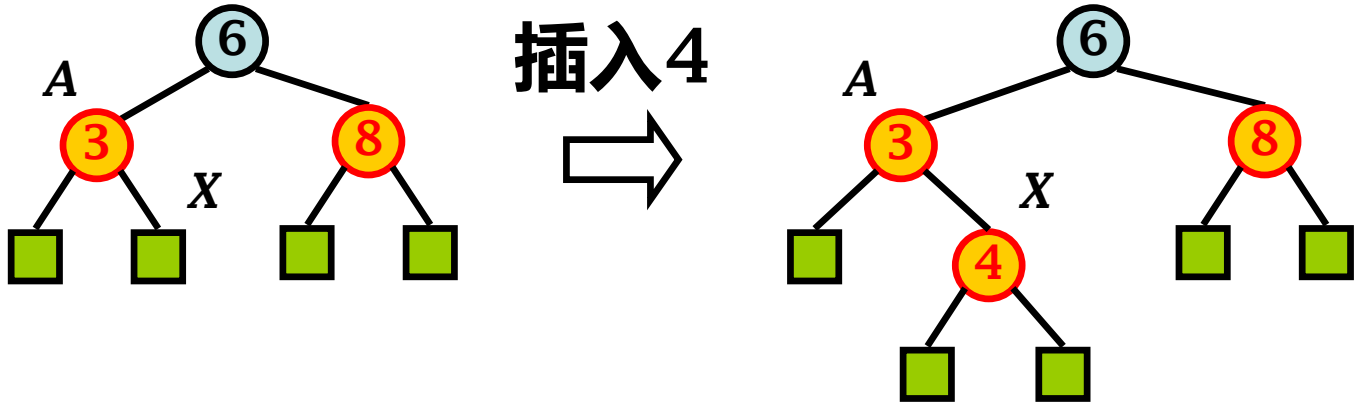
  由性质（3）得 $n >= 2^k - 1$, 即 $n >= 2^{(h-1)/2} - 1$

  可得出 $h <= 2\log_2(n+1)+1$



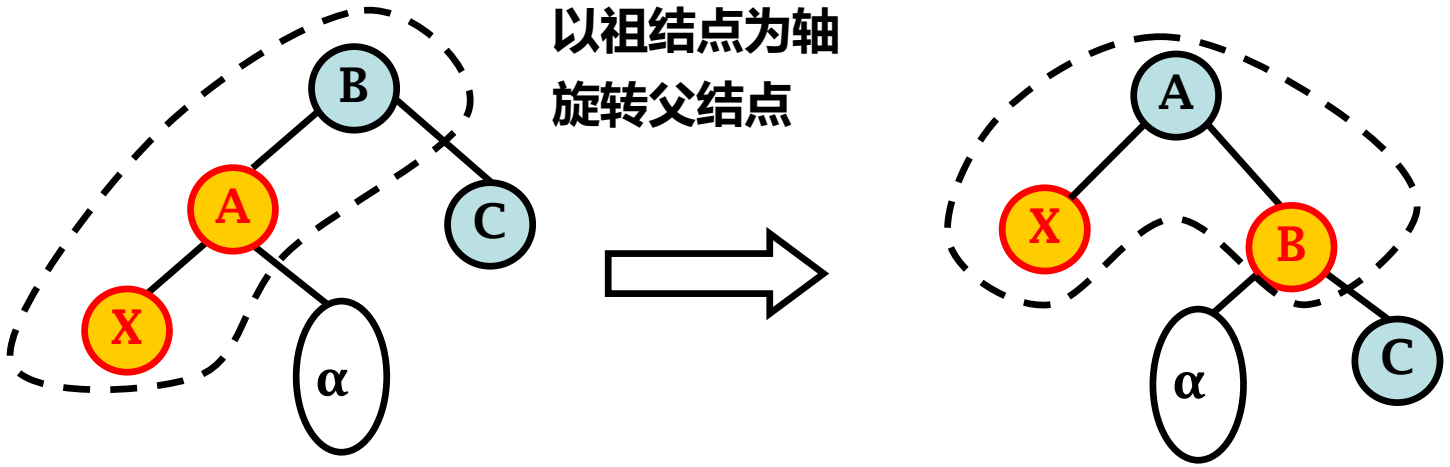
11.6.3 插入算法

- 先调用 BST 的插入算法
  - 新记录标记为什么颜色？ **红色**
  - 若父结点是黑色，则算法结束
- 否则，双红调整



插入算法调整 1：重构

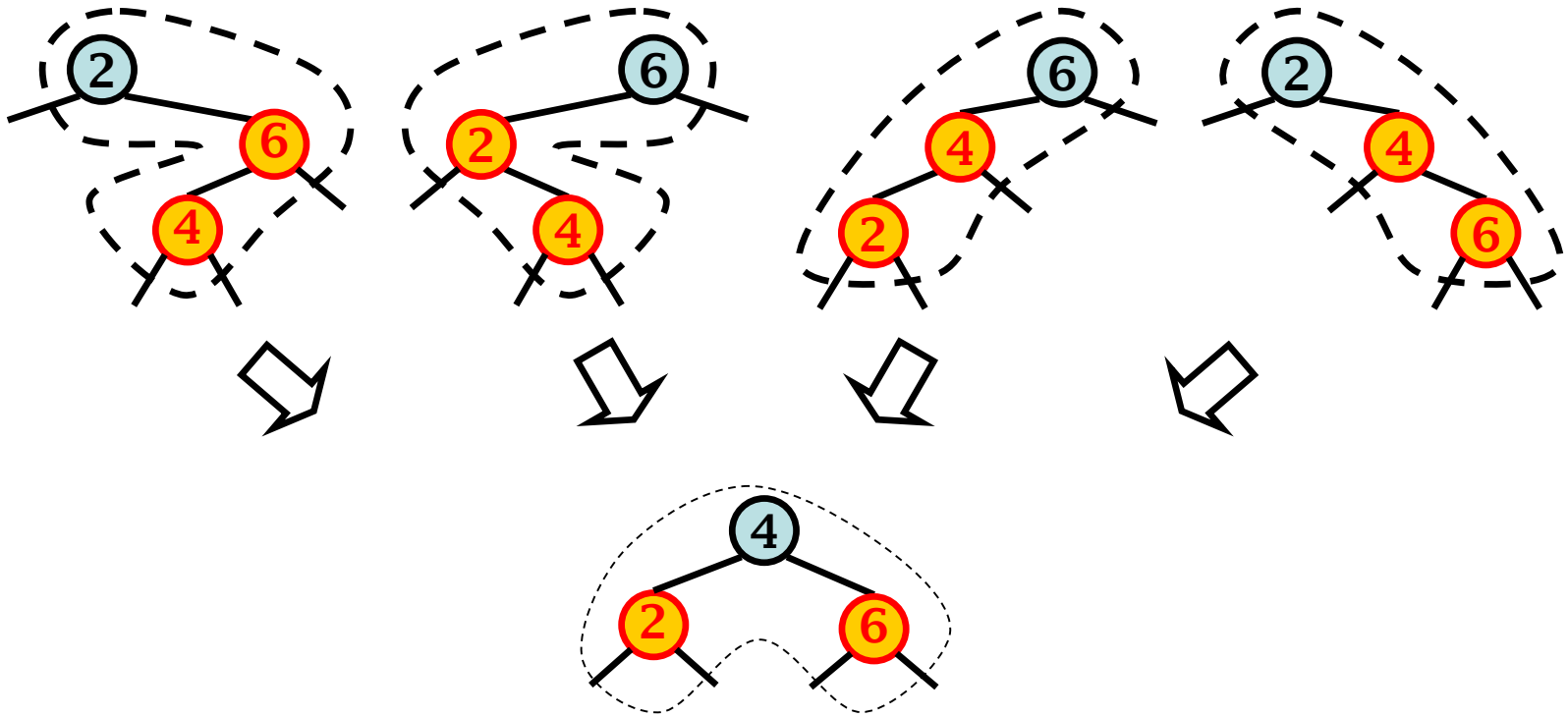
- 情况1：新增结点 X 的叔父结点是黑色



- 每个结点的阶都保持原值，调整完成

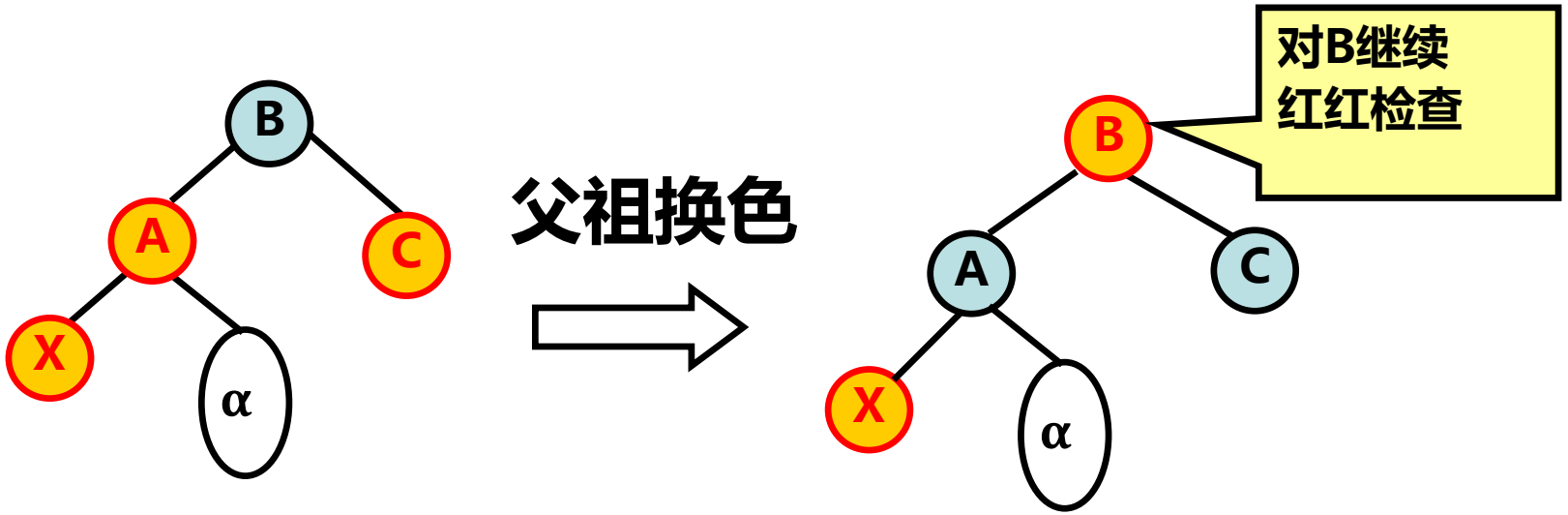
4 种形式的结构调整

- 原则：保持 BST 的中序性质



插入算法调整 2：换色

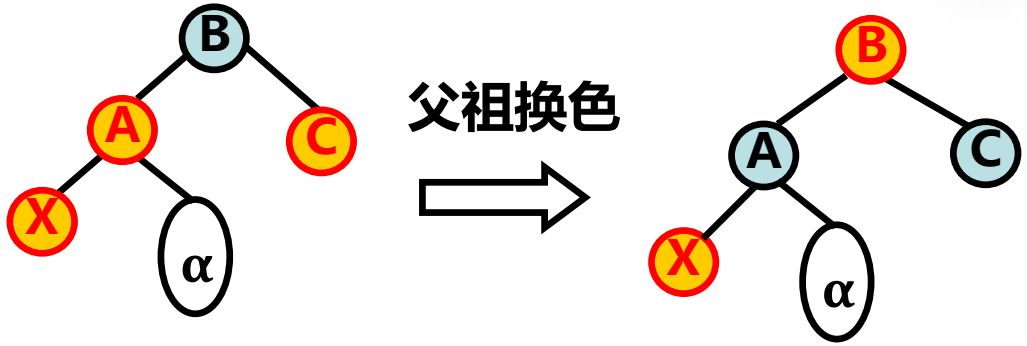
- 情况 2：新增结点 X 的叔父结点也是红色

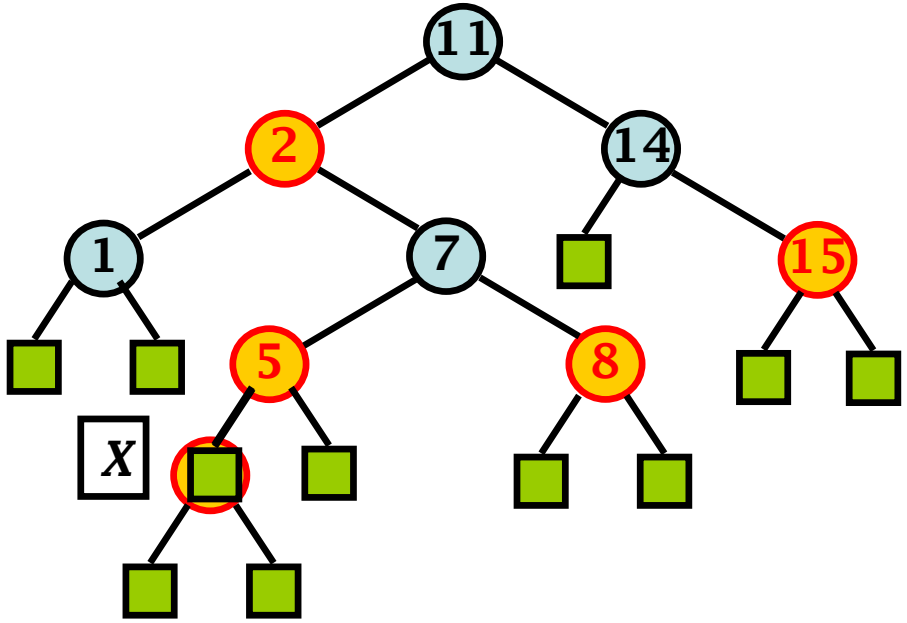


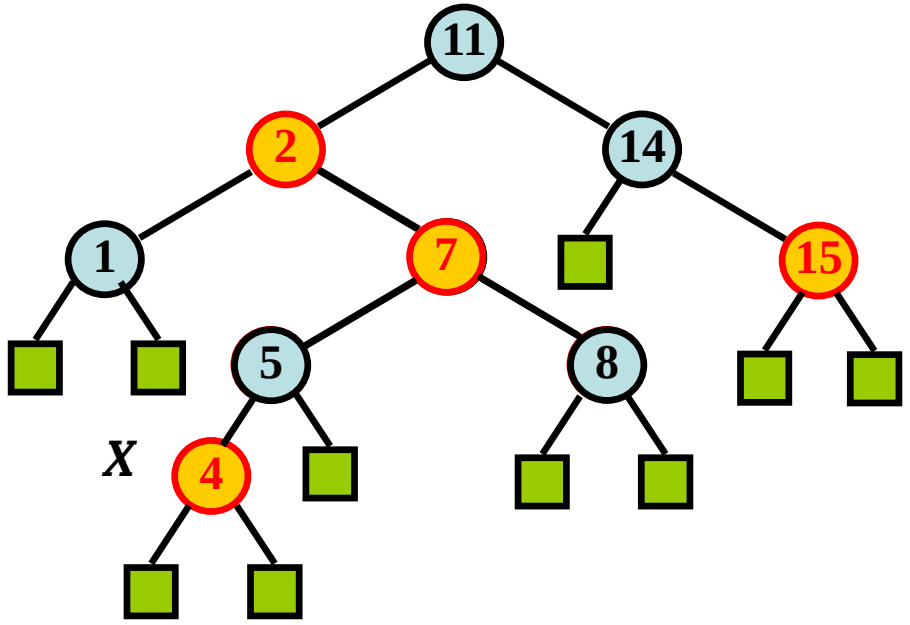
- 需要继续检查平衡
  - 如果**根变成了红**，那么直接**把根标为黑色**

插入 4

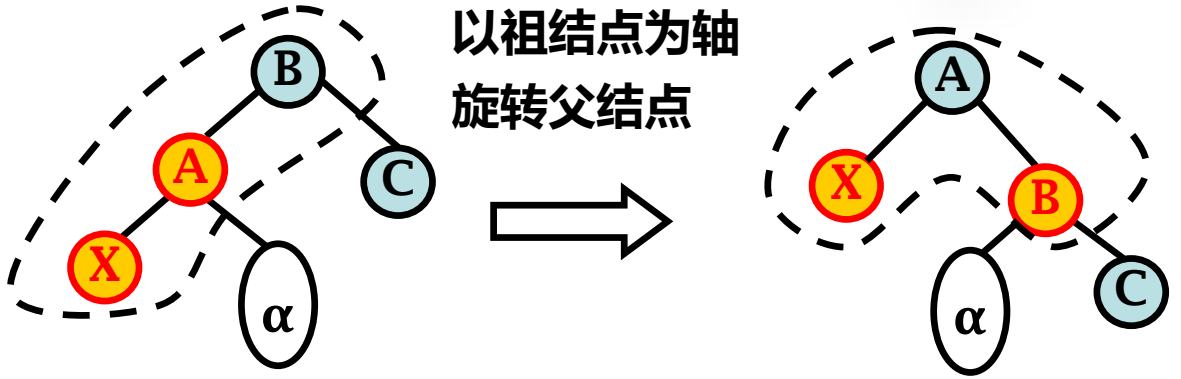
- 情况 2 红红冲突
  - 父和叔父也是红
- 父祖换色



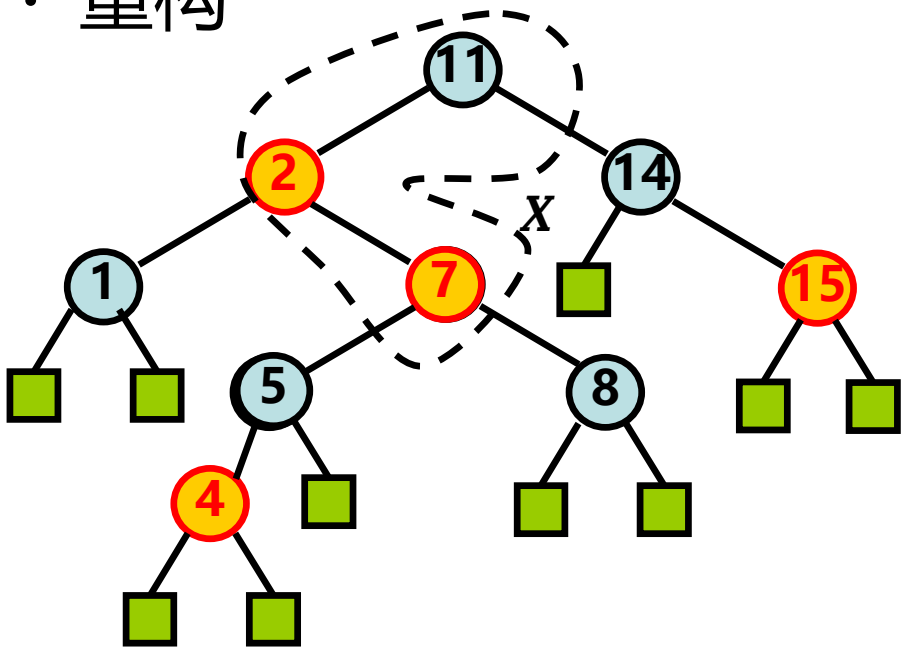


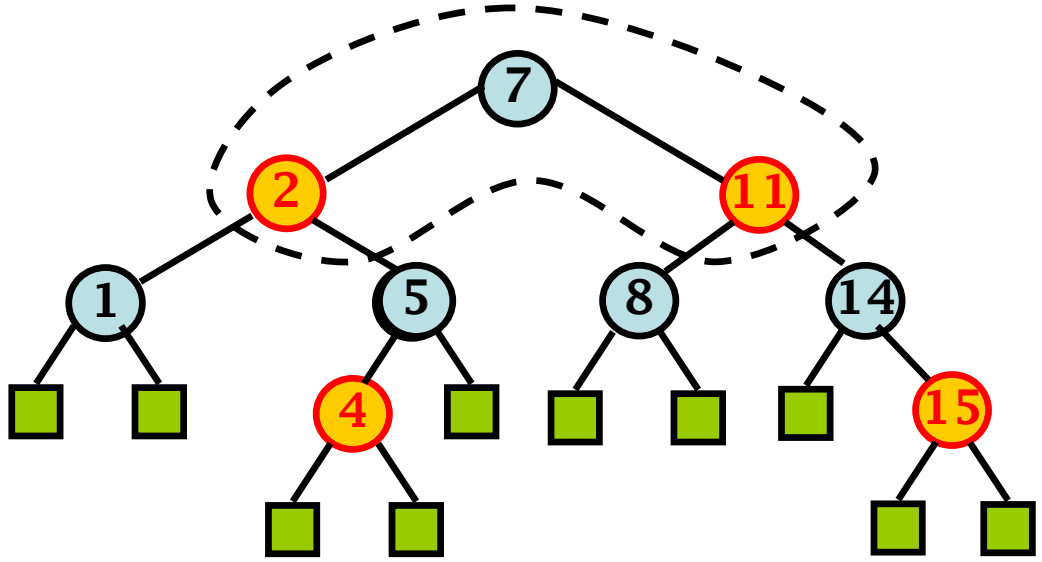


插入 4



- 情况 1 红红冲突
  - 叔父是黑
- 重构





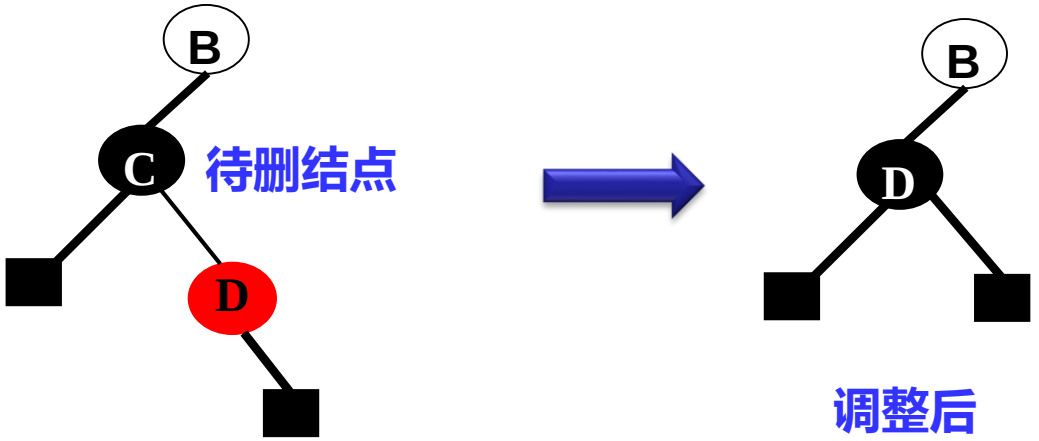
11.6.4 删除算法

- 先调用 BST 的删除算法
  - 待删除的结点有一个以上的外部空指针，则直接删除
  - 否则在右子树中找到其后继结点进行值交换（着色不变）删除
- v 是被删除的内结点，w 是被删外结点，X 是 w 的兄弟

（1）待删结点有1个外部叶结点

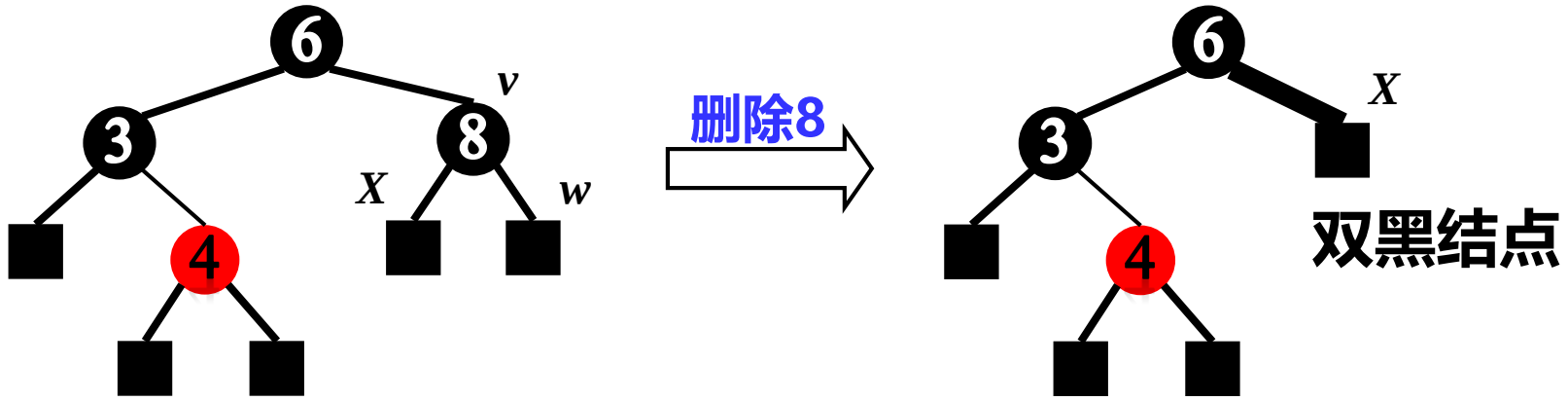
  - 则可知其非空子结点肯定为红色，待删结点肯定为黑色；

为什么?



(2) 待删除结点有2个外部叶结点

- 如果待删结点是红色，则直接删除（例如4号结点）；
- 否则，需要标记为**双黑**结点，根据其兄弟结点进行重构调整



因此，仅需处理双黑结点的调整问题！

11.6.4 删除算法

根据双黑 X 的兄弟 C 进行调整

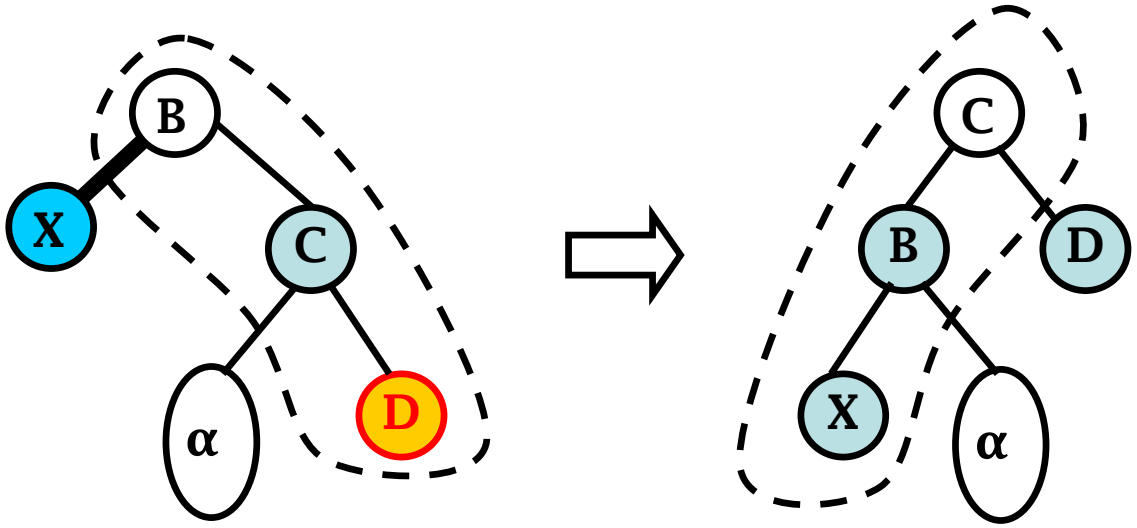
假设X是左子结点（若X为右孩子，则对称）

- **情况 1**： C 是黑色，且子结点有红色
  - 重构，完成操作
- **情况 2**：C 是黑色，且有两个黑子结点
  - 换色
  - 父结点 B 原为黑色，可能需要从 B 继续向上调整
- **情况 3**： C 是红色
  - 转换状态
  - C 转为父结点，调整为情况 1 或 2 继续处理

11.6.4 删除算法

情况 1(a) C 黑色，其子D红
重构：侄子红结点八字

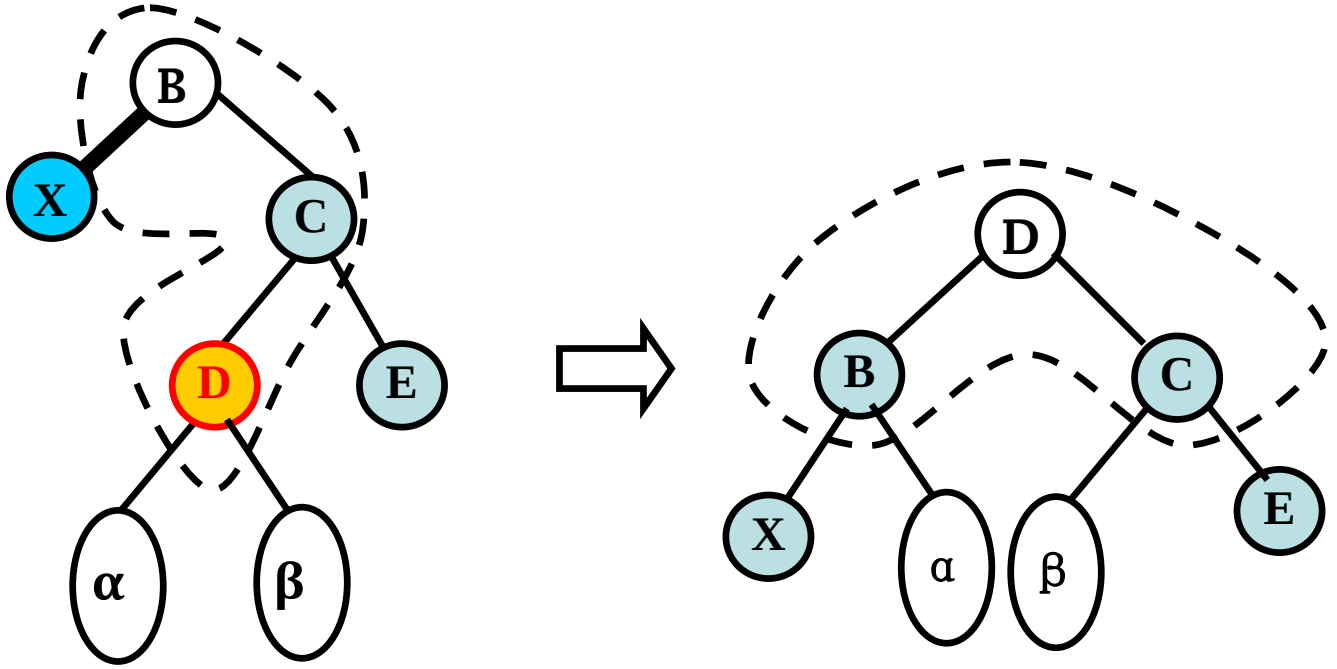
- 将兄弟结点 C 提上去
- C 继承原父结点的颜色
- 然后把B着为黑色，D 着为黑色，其他颜色不变即可



11.6.4 删除算法

情况 1(b) C 黑色，其子D红
重构：侄子红结点同边顺

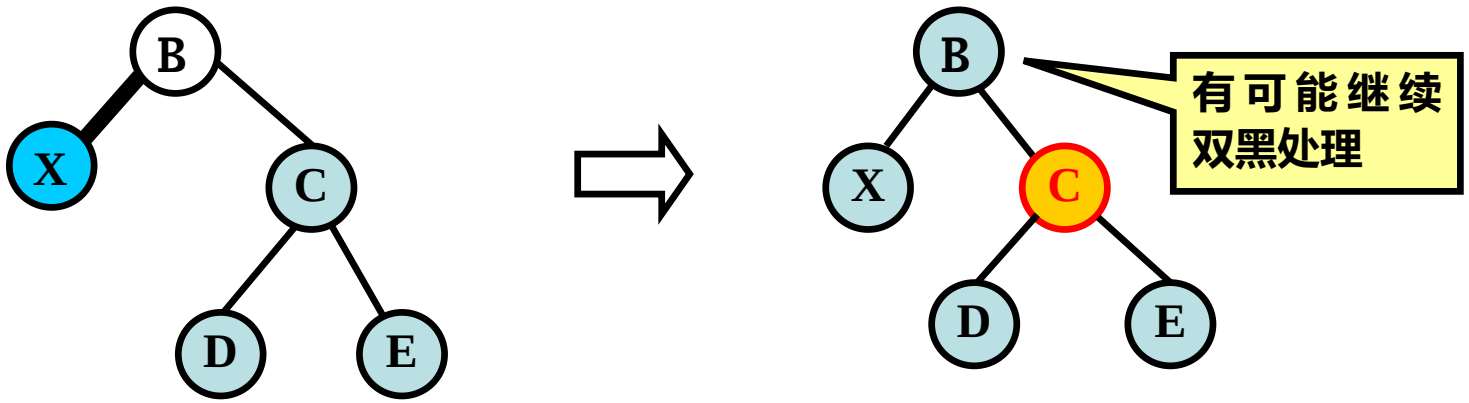
- 将 D 结点旋转为 C 结点的父结点，D 继承原子根 B 的颜色，B 着为黑色



11.6.4 删除算法

情况 2：兄弟 C黑，且有两个黑子结点

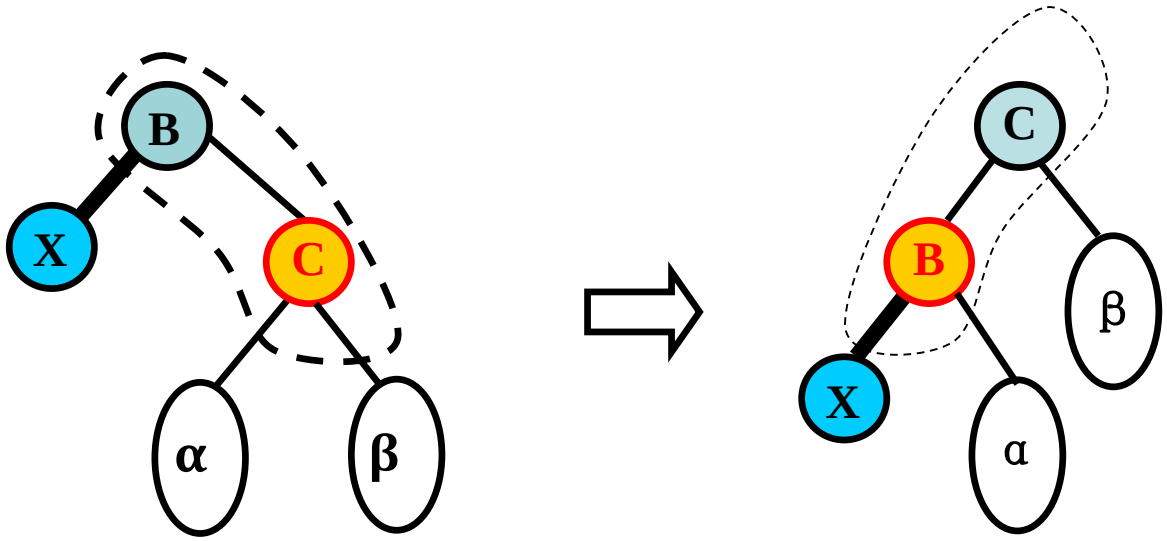
- 把 C 着红色， B 着黑色
  - 如果 B 原为红色，或者 B为根，则算法结束
  - 否则，对 B 继续作“**双黑**”调整



11.6.4 删除算法

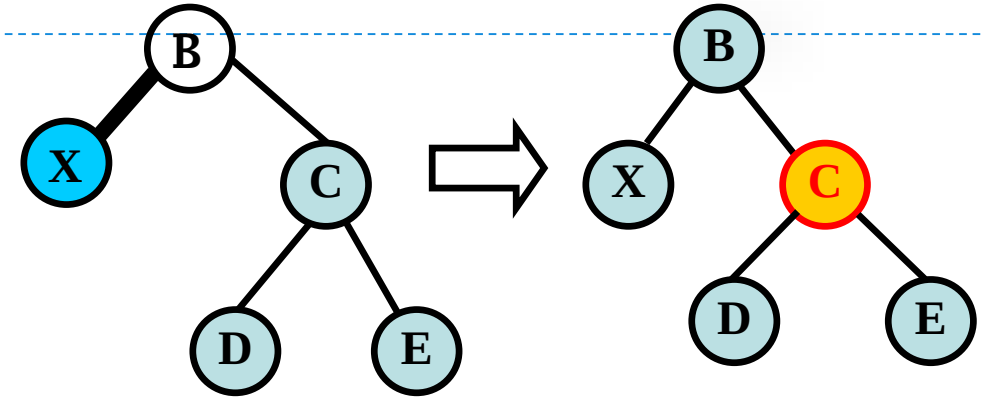
情况 3：兄弟 C 是红色

- 旋转
- X 结点仍是“双黑”结点，转化为前面2种情况
  - 观察 α 子树根结点的颜色

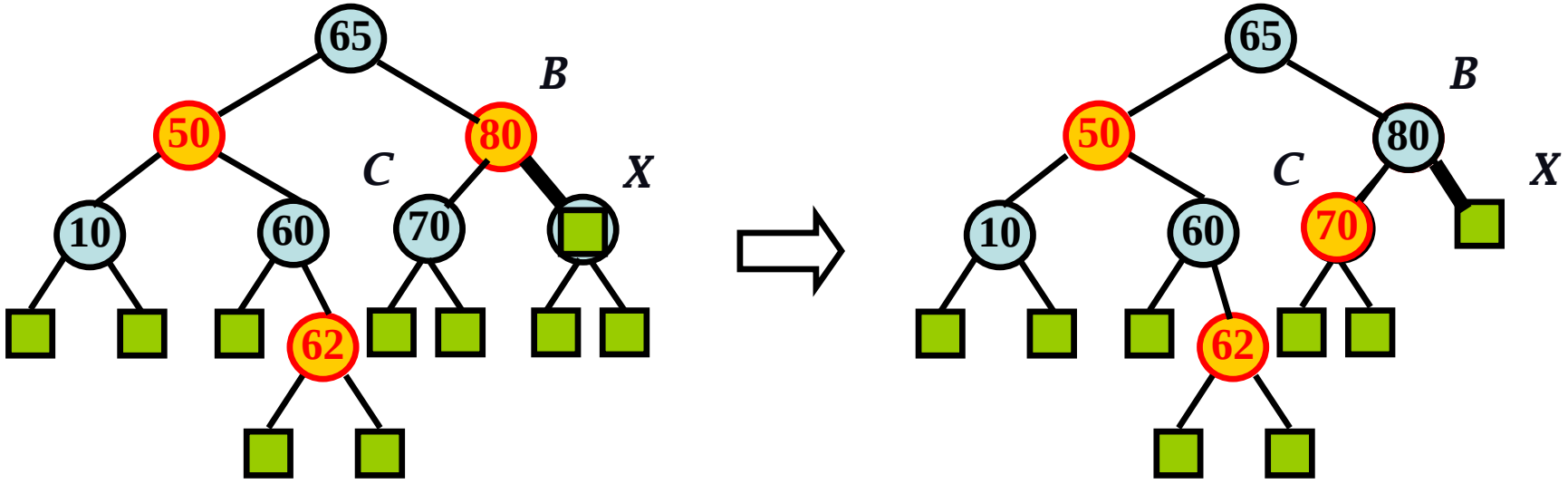


11.6.4 删除算法

删除 90



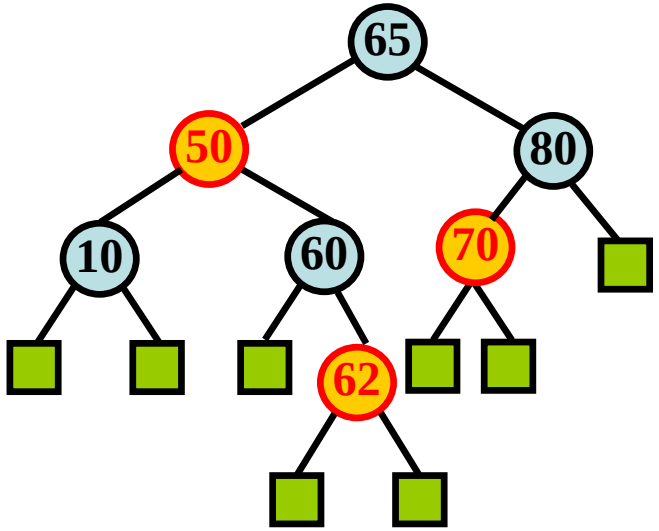
- 当前结点变为 80 的右黑叶结点
- C 是黑色, 且有两个黑色子结点
  - 情况 2



11.6.4 删除算法

删除 70

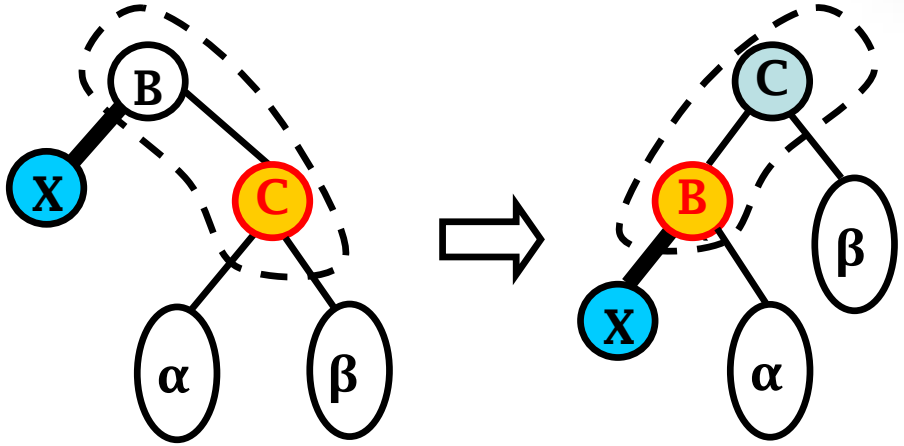
- 红结点，不要调整

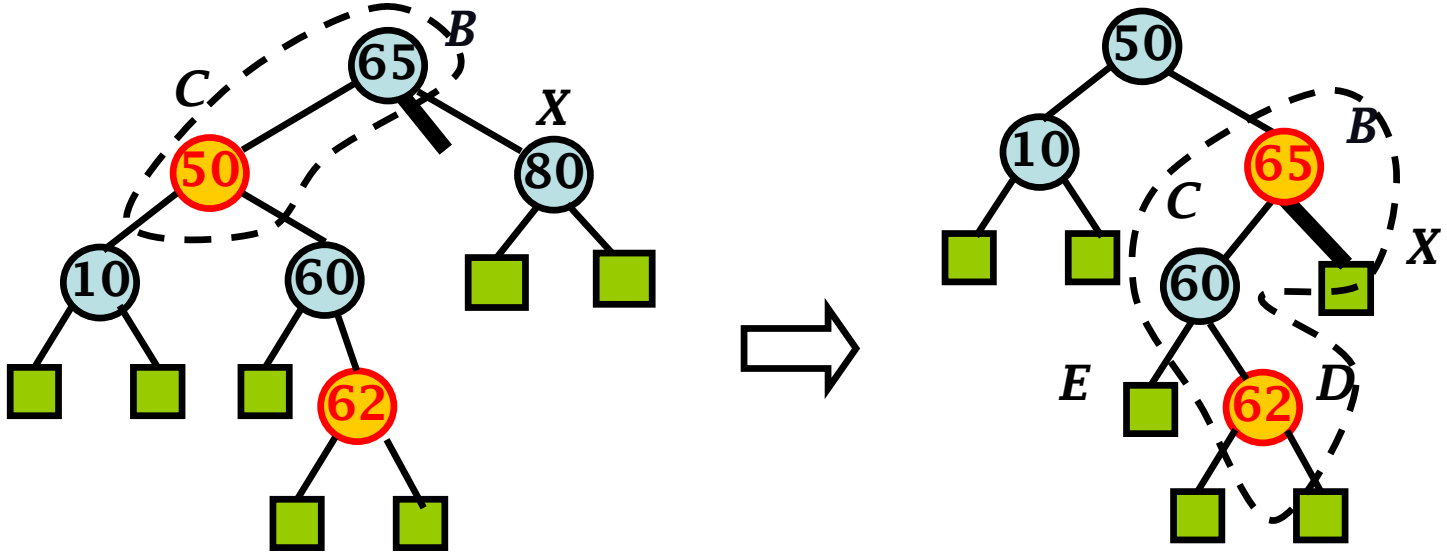


11.6.4 删除算法

删除 80

- 当前结点 X 变为 65 的右黑叶结点
- C 是红色 —> **情况 3**

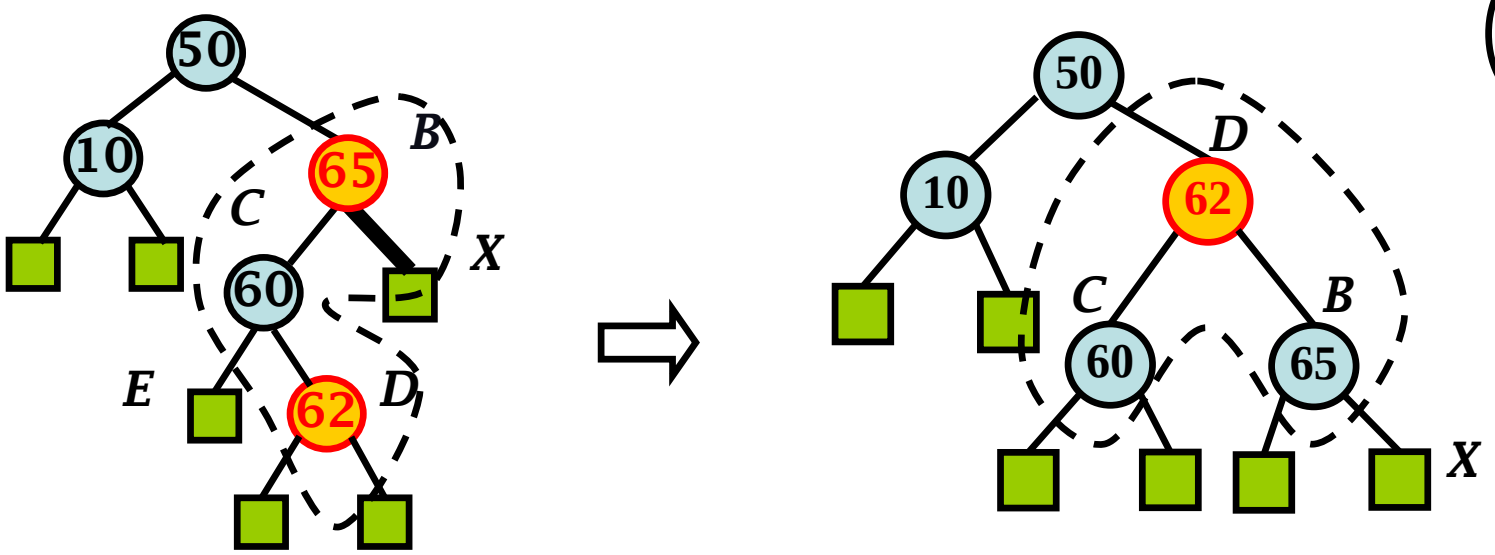


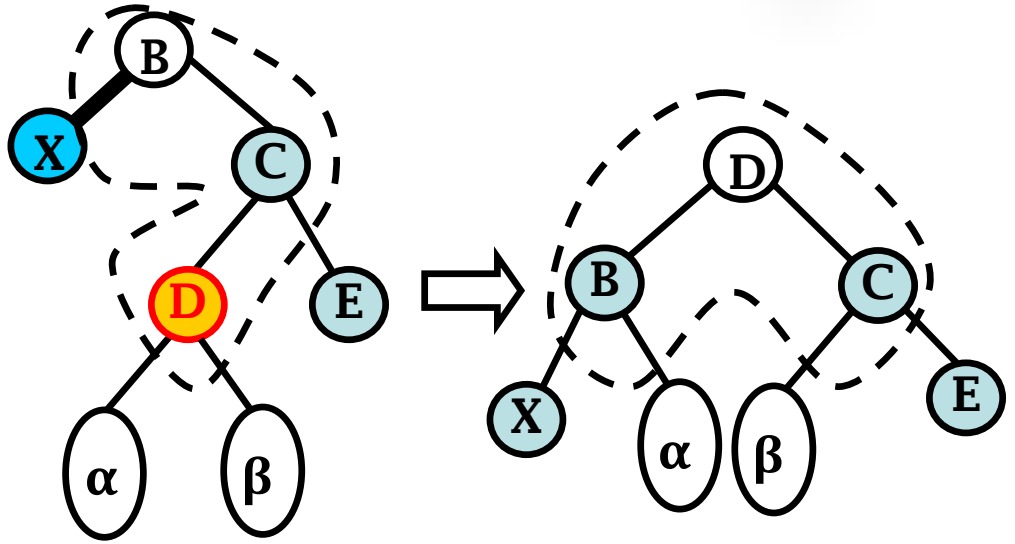


11.6.4 删除算法

删除 80（继续调整）

- C 是黑色，且左子黑、右子红
  - 情况 1(b)





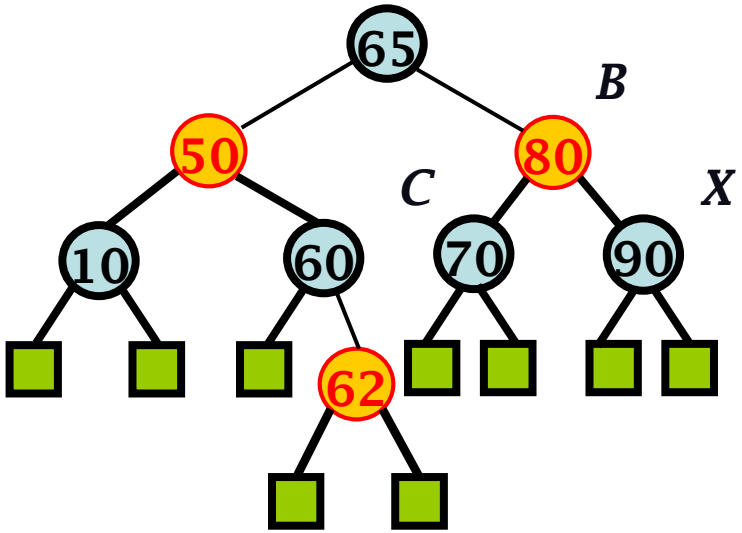
11.6.4 删除算法

红黑树操作的时间代价

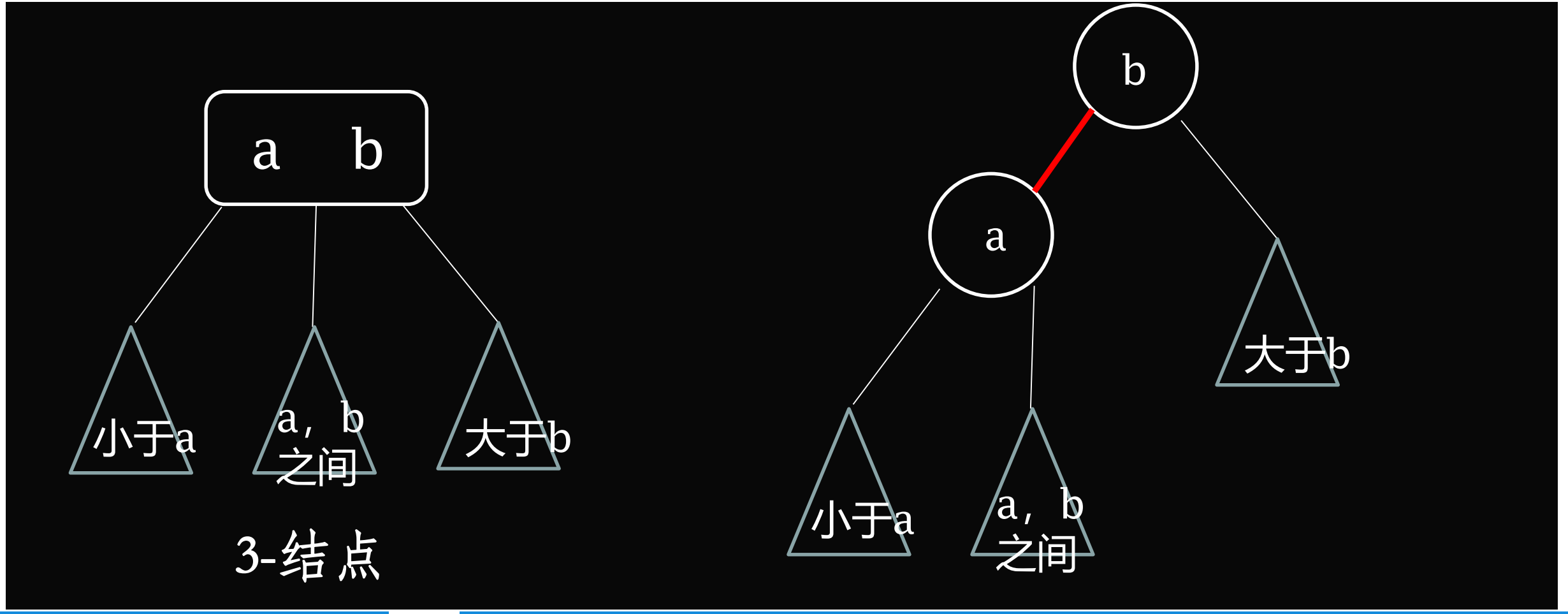
- 检索平均和最差时间都是$O(\log_2 n)$
  - 自根向下查找，代价就是树高
- 插入、删除平均和最差时间都是$O(\log_2 n)$
  - 先检索到待操作的位置
  - 自底向根的方向调整，局部的调整操作
- 红黑树的应用比较广泛
  - Java集合中的TreeSet和TreeMap
  - C++ STL中的set和map
  - 以及Linux虚拟内存的管理

思考

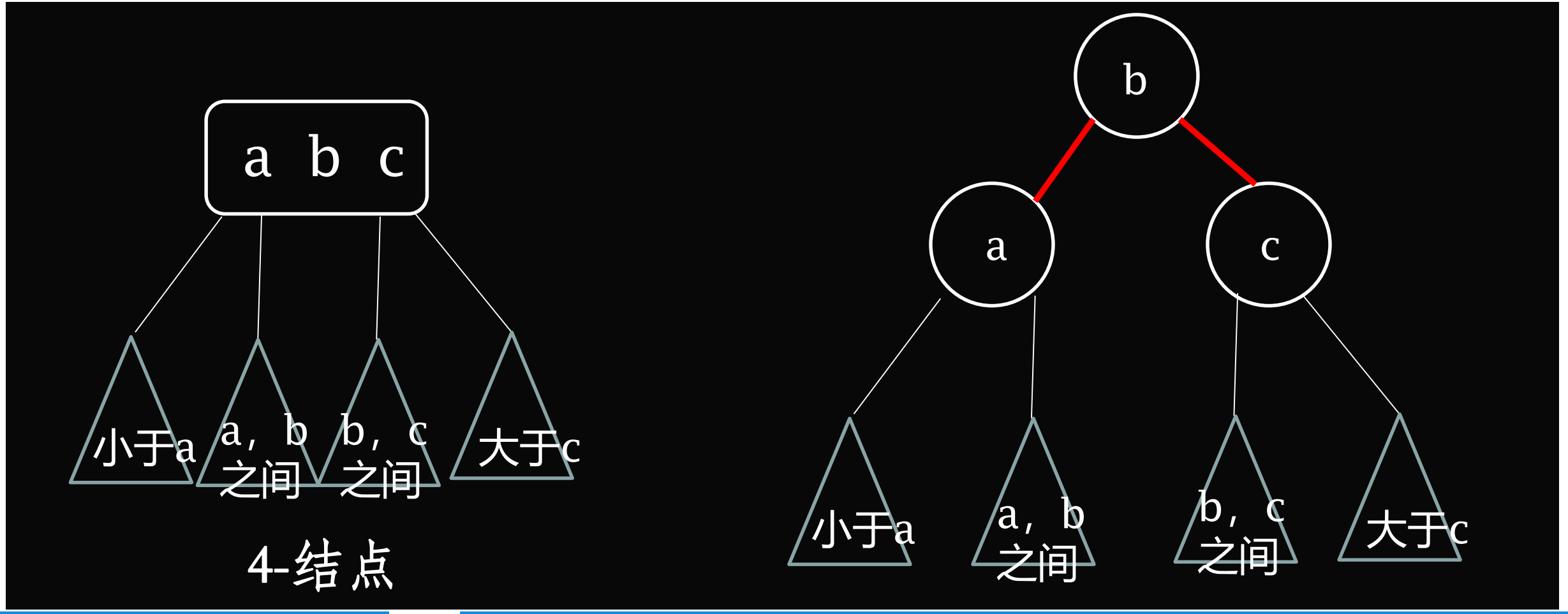
- 红黑树构造
  - （数据，左指针，右指针，颜色，父指针）
  - （数据，左指针，右指针，颜色）



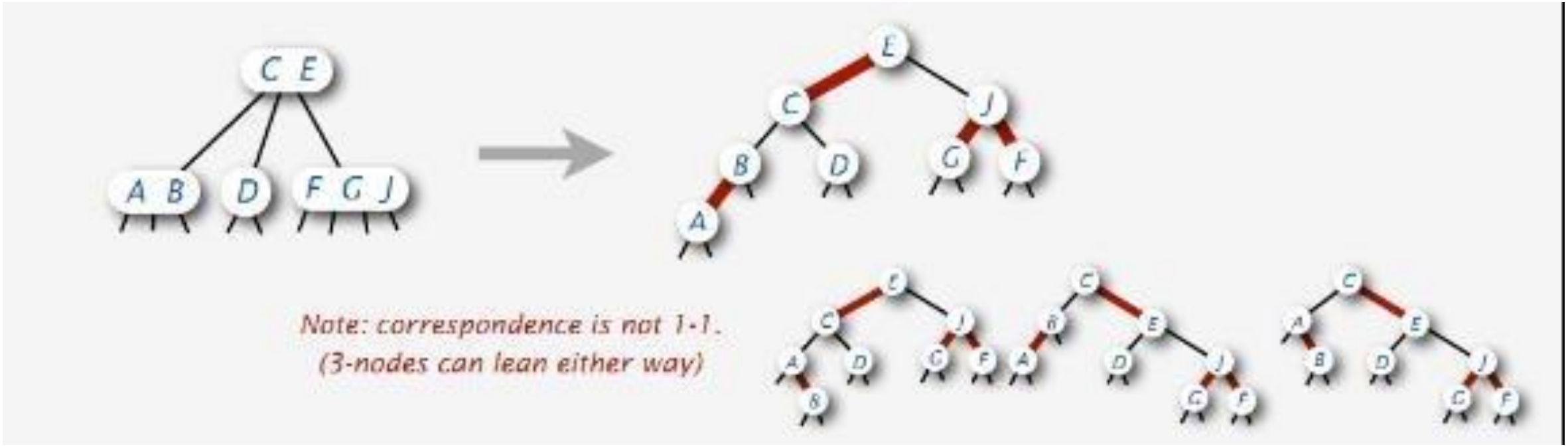
四阶B树（2-3-4树）与红黑树



四阶B树（2-3-4树）与红黑树

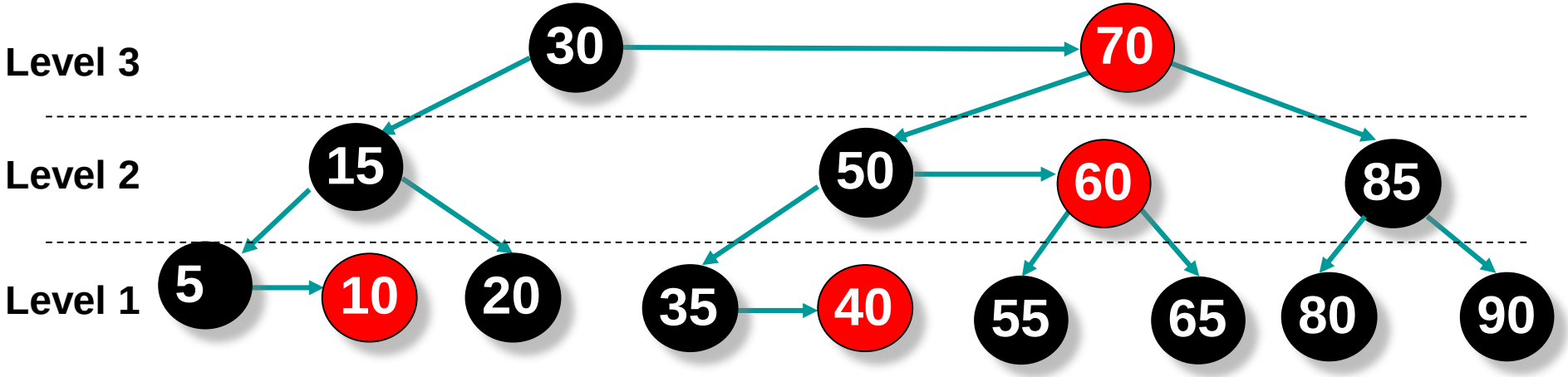


从2-3-4 树到红黑树


C E
A B
D
F G J
Note: correspondence is not 1-1.
(3-nodes can lean either way)

AA树

- AA-Tree是红黑树的变种
  - **没有左红子结点**
  - 红色的孩子与父结点处于同一级别
  - **存储结点的级别**，而不是结点的颜色



数据结构与算法

谢谢倾听

国家精品课“数据结构与算法”

http://jpk.pku.edu.cn/course/sjjg/

https://www.icourse163.org/course/PKU-1002534001

张铭，王腾蛟，赵海燕

高等教育出版社，2008. 6。“十二五”国家级规划教材