数据结构与算法

第11章　索引技术

主讲：赵海燕
北京大学信息科学技术学院
“数据结构与算法”教学组

国家精品课“数据结构与算法”
http://www.jpk.pku.edu.cn/pkujpk/course/sjjg/

张铭，王腾蛟，赵海燕
高等教育出版社，2008. 6, “十一五”国家级规划教材

静态索引的局限

- 索引表一经产生不再改变（除非重组），不随操作加以维护
- 不适于记录插入、删除频繁的数据

如何解决？

动态索引

- 基本概念
- B树
- B^+树
- VSAM
- B树的性能分析
- 动态索引和静态索引性能的比较

动态索引结构

- 在运行过程中，插入和删除记录时索引结构本身也可能发生改变
  - 可以保持较好的检索性能
  - 有效支持所有操作，包括：
    - 插入、删除
    - 范围查询
    - 多种检索码
  - 例如，数据库中常用的B树、 B+树，以及基于B+树的VSAM（Virtual Storage Access Method）
    - VSAM 基于分页技术，与存储设备无关

动态索引结构面临的问题

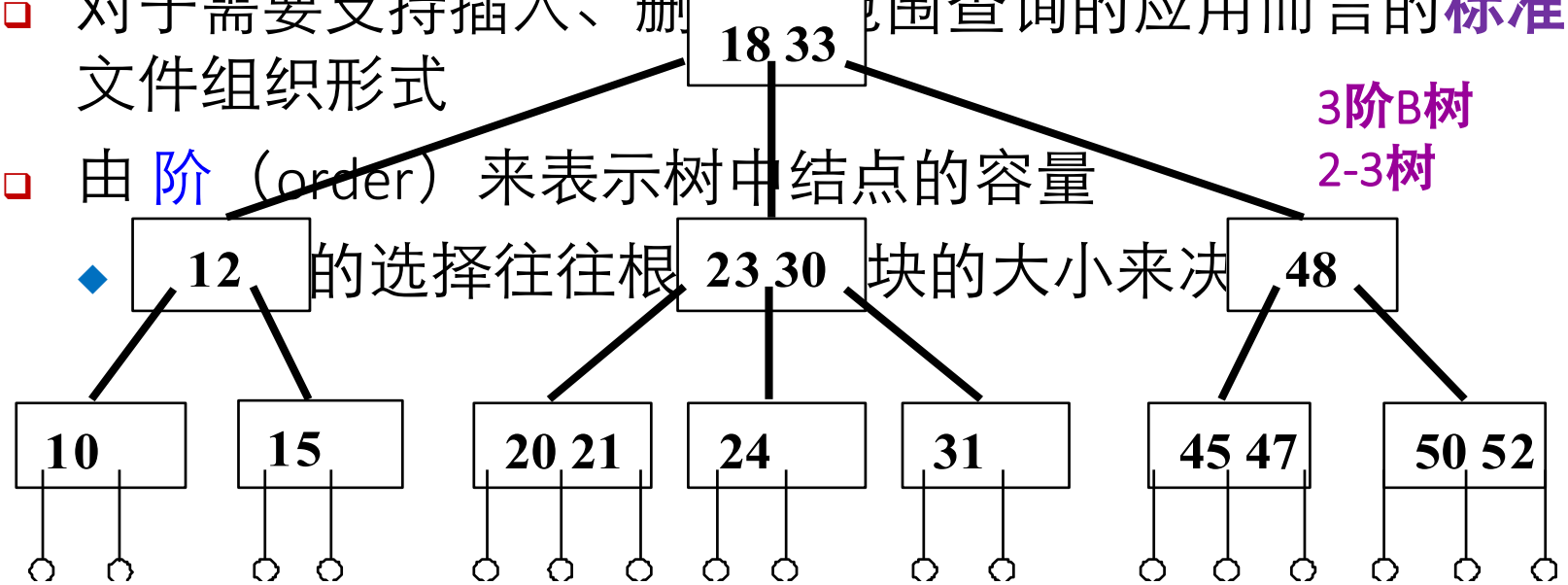
- 索引级数多
  - 相应指针空间占用量大
- 辅助索引维护困难
  - 索引的动态改变导致关键码地址的改变，可能会引起属性项索引（辅助索引）的改变
- 需考虑并行策略
  - 某个用户在查找数据时，可能会有另一用户在插入数据而修改了索引，导致前一个用户找不到数据

树索引

- 采用BST/AVL等树结构的索引可提供：
  - 独立、不依赖于数据的实现
  - 支持插入、删除操作
  - 保持平衡的开销相当小，尽管在设计上比较复杂
- 困难
  - log(N)的检索时间远逊于一个好的散列表
  - 为取得好的性能，树必须保持平衡
  - 若索引量巨大，BST&AVL在磁盘上难以有效存储

B树

- **一种平衡的多分树** (Balanced Tree)， 1970年由 R. Bayer 和 E. Mccreight 提出
  - 大型文件的组织方法
  - 对于需要支持插入、删除和范围查询的应用而言的**标准**文件组织形式
  - 由 阶（order）来表示树中结点的容量
    - 的选择往往根 块的大小来决

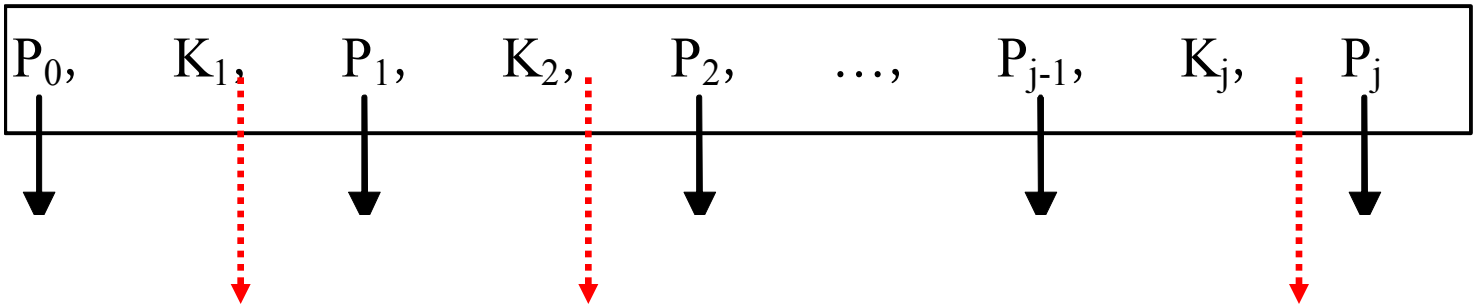


m阶B树的结构定义

- 满足下列条件
  - 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$\lceil m/2 \rceil$个子结点；
  - ③ 根结点**至少**有两个子结点
    - 唯一例外： 根结点为叶结点时，因其没有子结点，此时B树只包含一个结点
  - ④ 所有叶结点均在同一层；
  - ⑤ 有 *k* 个子结点的非根结点恰好包含 *k-1* 个关键码

B树的结点结构

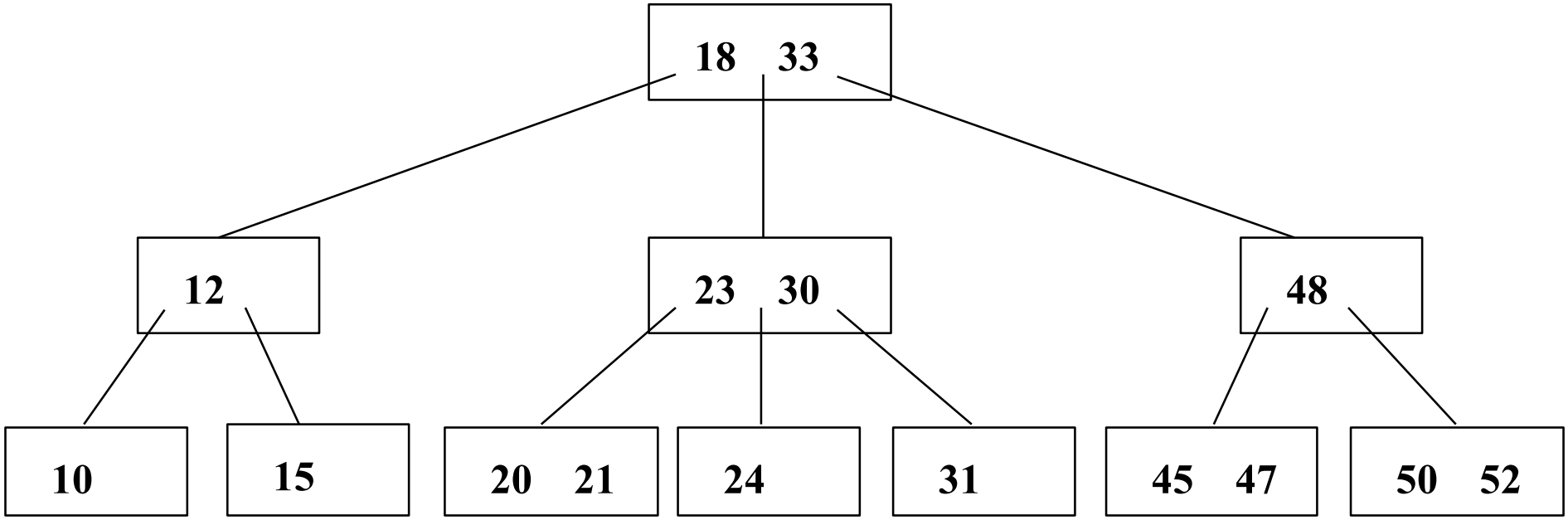
- B树的一个结点包含 *j* 个关键码、*j+1*个指针，一般形式为：



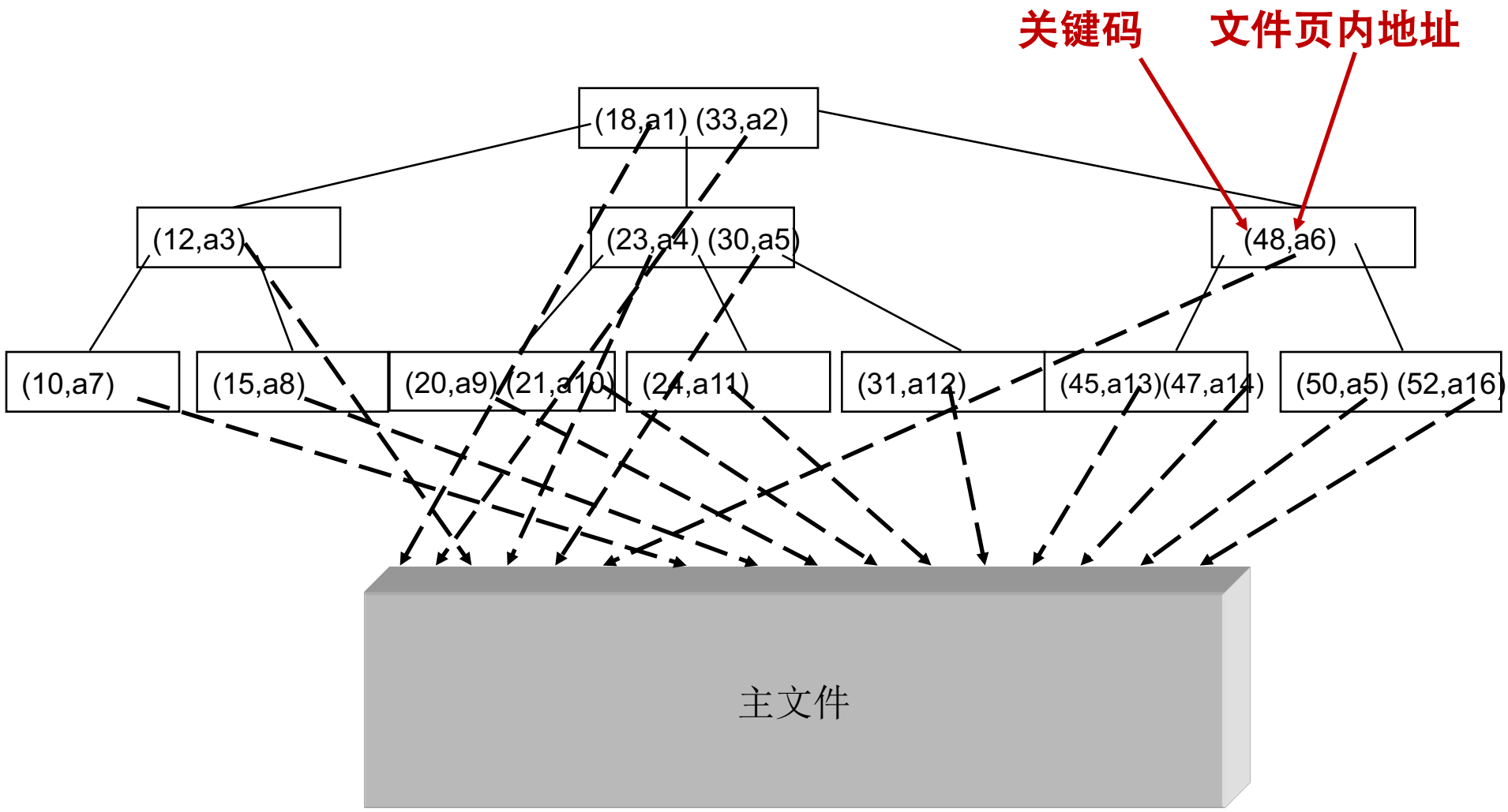
- **其中，**
  - K_i 是关键码值，$K_1<K_2<...<K_j$，
  - P_i 是指向包括K_i到 K_{i+1} 之间的关键码的子树的指针
- 还有**别的指针**吗?

B树

- B树中结点的隐含指针



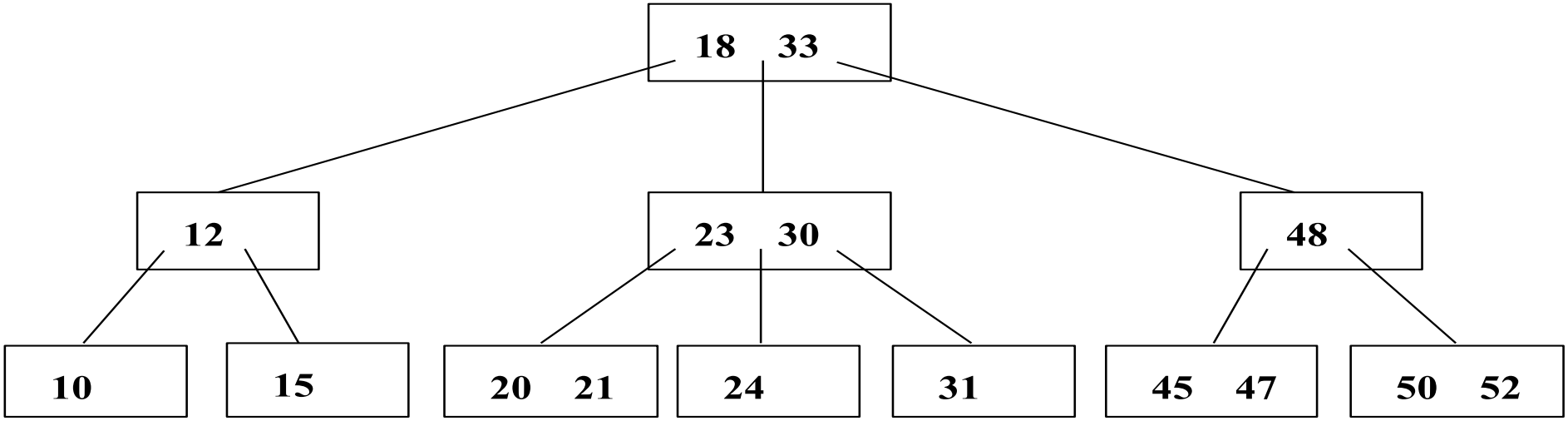
B树


关键码
文件页内地址
(18,a1) (33,a2)
(12,a3)
(23,a4) (30,a5)
(48,a6)
(10,a7)
(15,a8)
(20,a9) (21,a10)
(24,a11)
(31,a12)
(45,a13)(47,a14)
(50,a5) (52,a16)
主文件

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
- 保证树中至少有一定比例的结点是满的
  - 改进空间的利用率
  - 减少检索和更新操作的磁盘读取次数
- 把（值接近的）相关记录放在同一磁盘块中，利用访问局部性原理

2-3树 = 3 阶B树



B树上的检索

- **交替的两步过程**
  1. **读入根结点**，在其所包含的关键码K_1，…，K_j中查找给定的关键码值
     - **找到，检索成功**
  2. 否则，**确定要查的关键码值所在区间**（某个K_i和K_{i+1}之间），按该区间的指针P_i所指向的下一层结点继续查找
- **上述过程后，若P_i指向外部空结点，表示检索失败**
- **读、写盘的次数依赖于?**

B树的检索长度

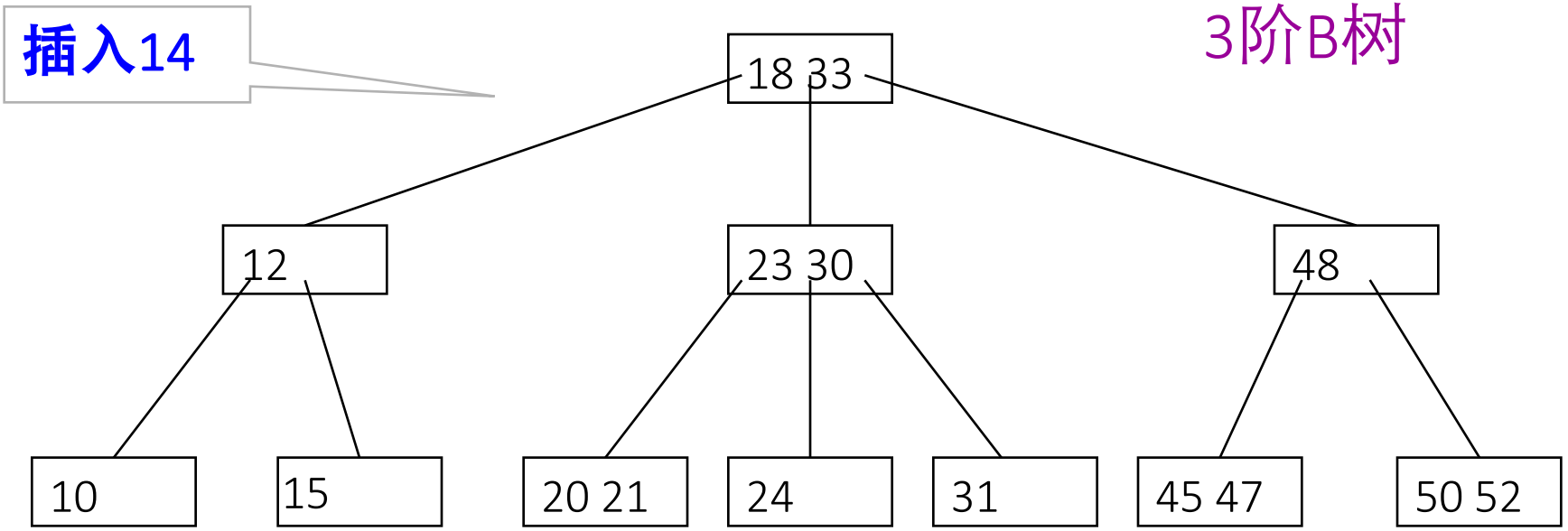
- **设B树的高度为 h**
  - 独根树高为1
- 自顶向下检索到叶结点的过程中可能需要的读盘次数?
  - h ?

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**传递**到**根结点**(则树升高一层)
- **伴随的操作：分裂**

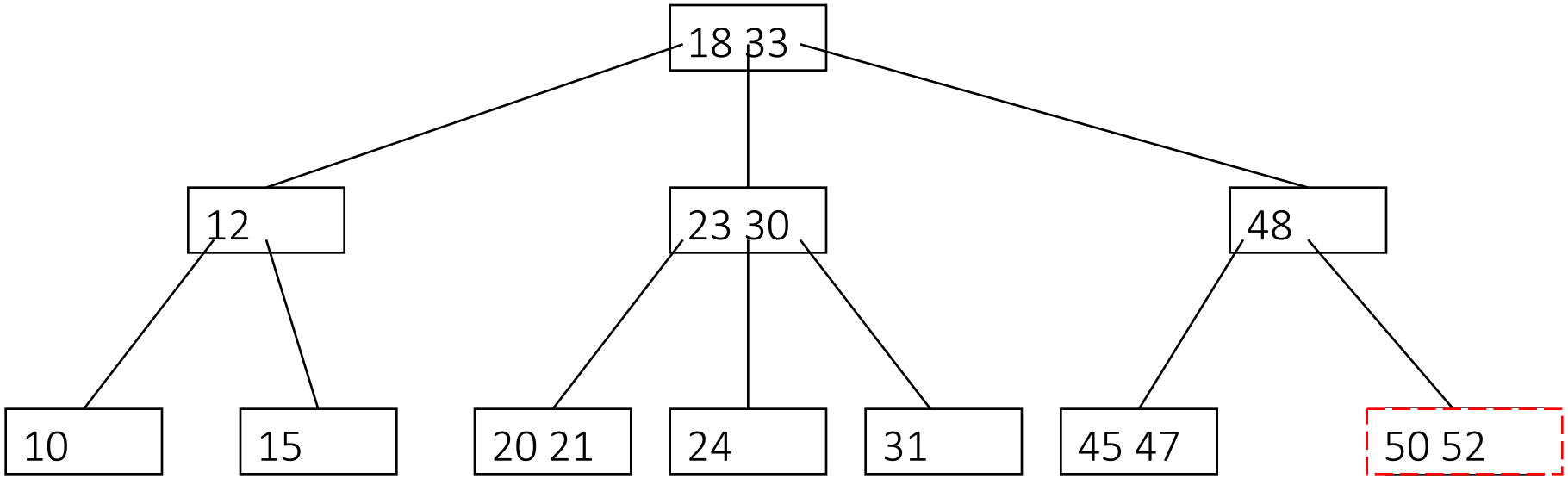
B树插入：简单插入示例

- **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



B树插入：分裂示例

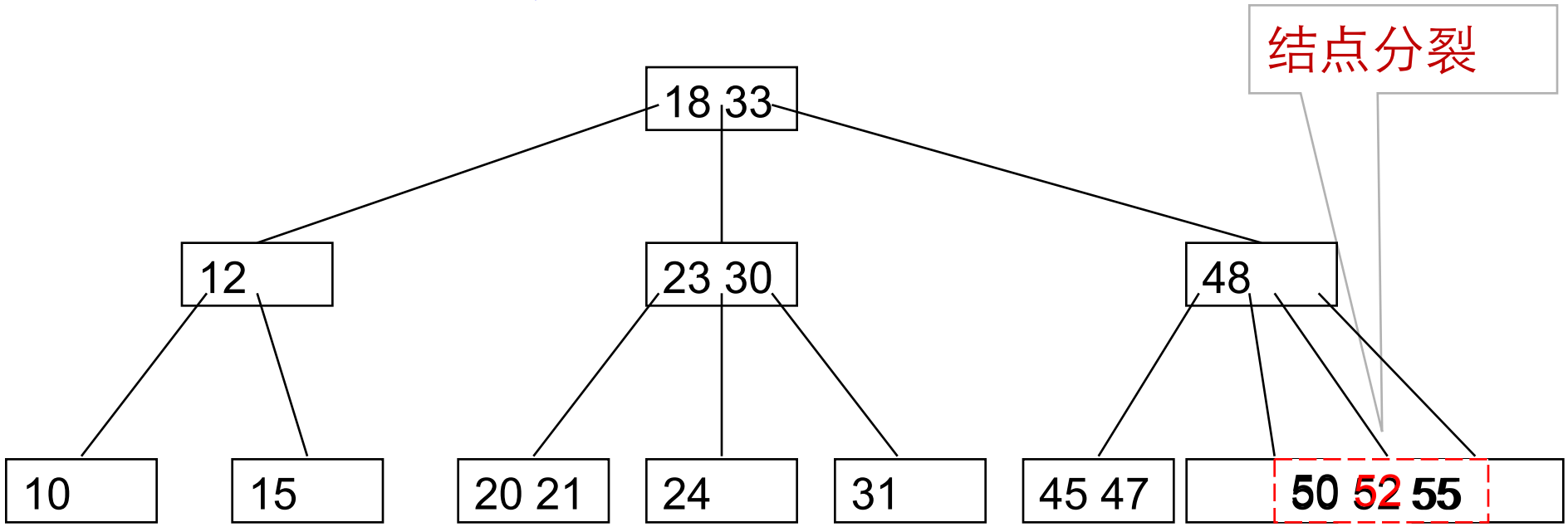
m=3，插入55



55

B树插入：传递示例

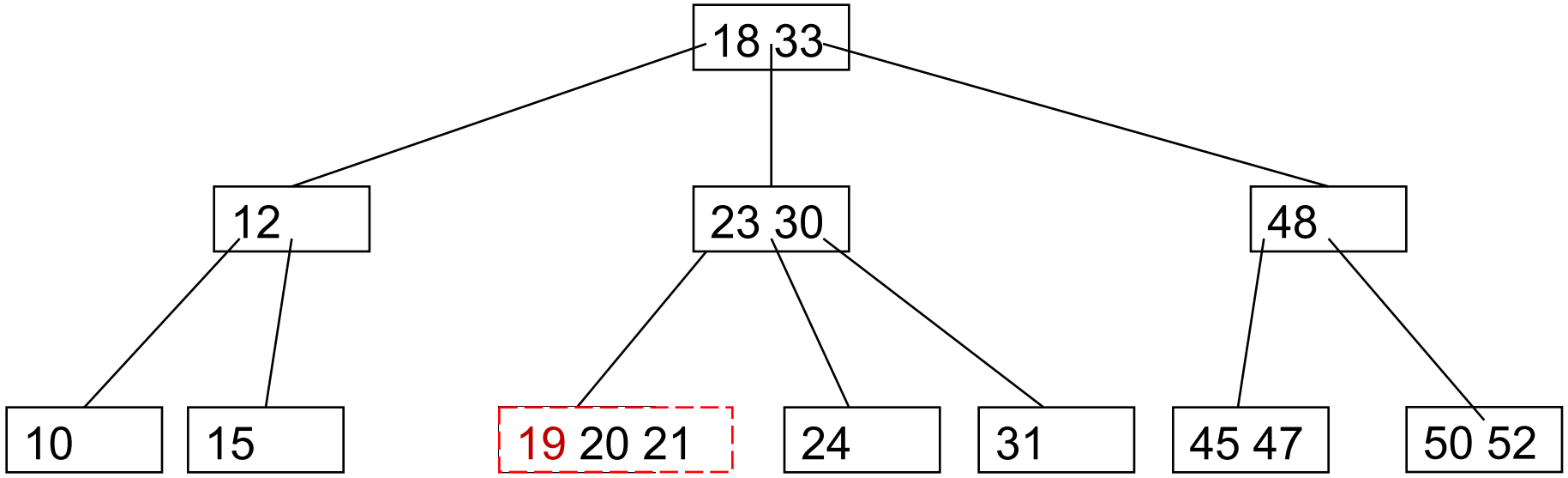
m=3，叶结点分裂，把52提升到父结点



读写次数?

B树插入：传递至根结点示例

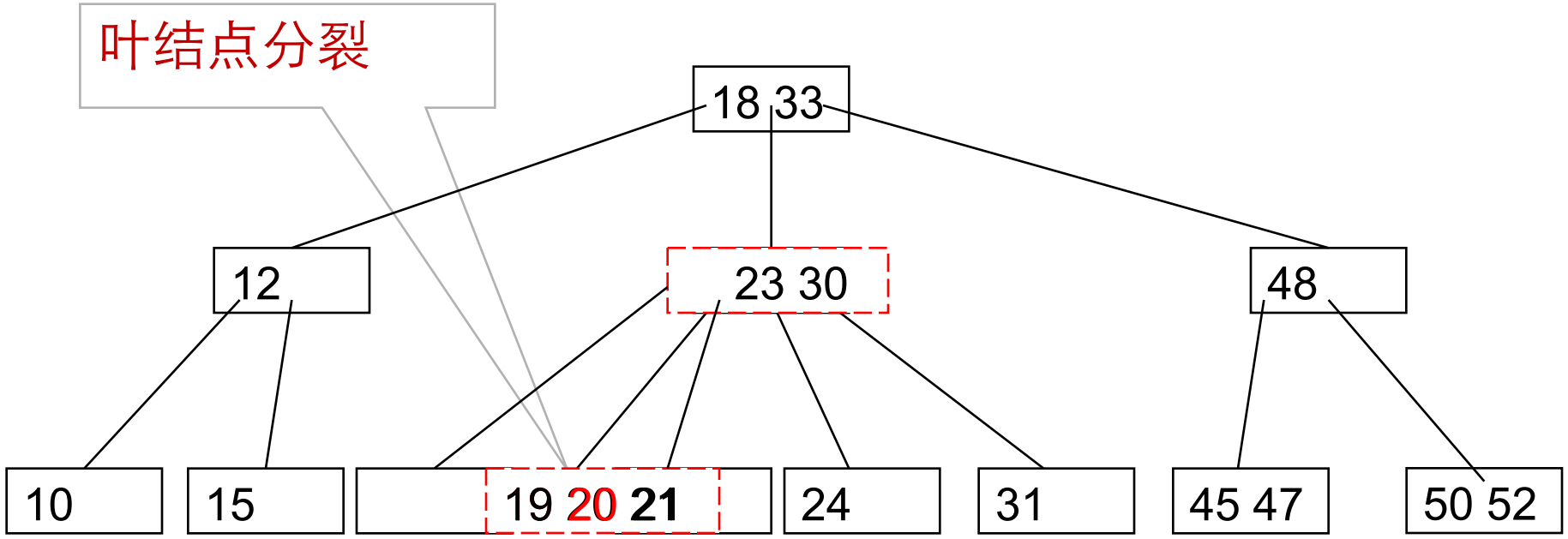
m=3，插入19



19

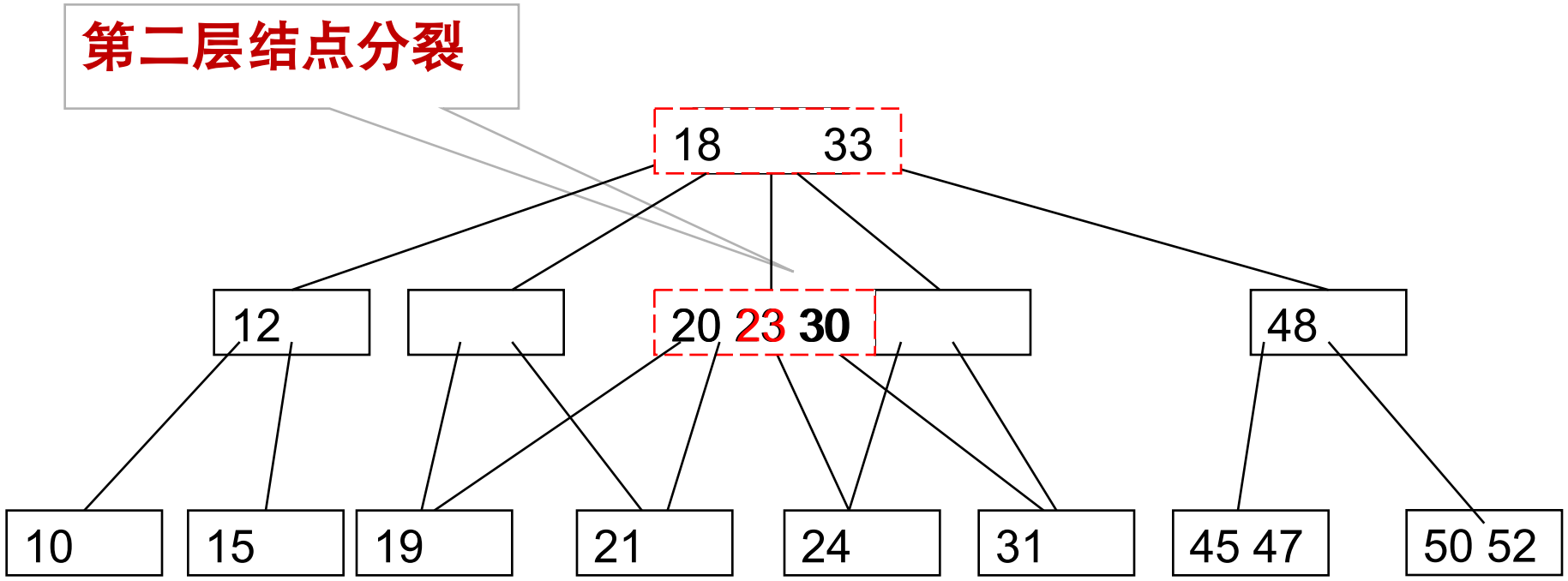
B树插入：传递至根结点示例

m=3



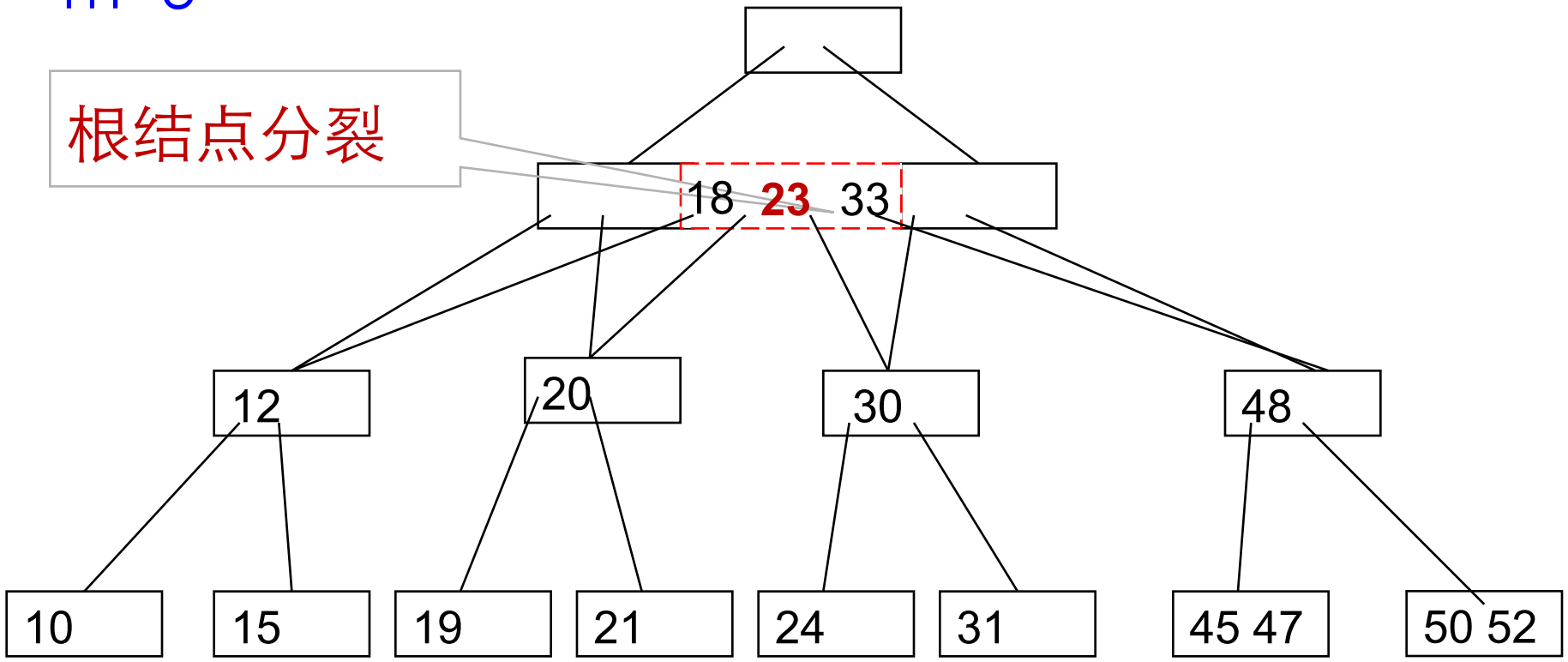
B树插入：传递至根结点示例

m=3



B树插入：传递至根结点示例

m=3



读写次数?

访外次数

- 上述插入过程有 10 次对B树的访外操作
  - **读盘 3 次**（a、c、g）
  - **写盘 7 次**（g、g’、c、c’、a、a’、t）
- **不考虑**对主数据文件的**访外操作，也不考虑申请新磁盘块的开销**

B树操作的访外次数

- 假设内存工作区足够大，使得向下检索时读入的结点在插入后向上分裂时不必从磁盘再次读入
  - **读盘次数与检索相同**
  - **最少写盘次数：1 次**
    - 不分裂，写出这个关键码插入后的结点
  - **有结点分裂的情况下，插入操作的最少写盘次数?**

1次插入操作最多读写次数

- 总共h 层，每层都需要分裂（包括根）
- 分裂一个非根结点要向磁盘写出 2 个结点，分裂根结点（最后一次）要写出 3 个结点

= 找插入结点向下读盘次数 +

+ 分裂**非根结点**时写盘次数 +

+ 分裂**根结点**时写盘次数

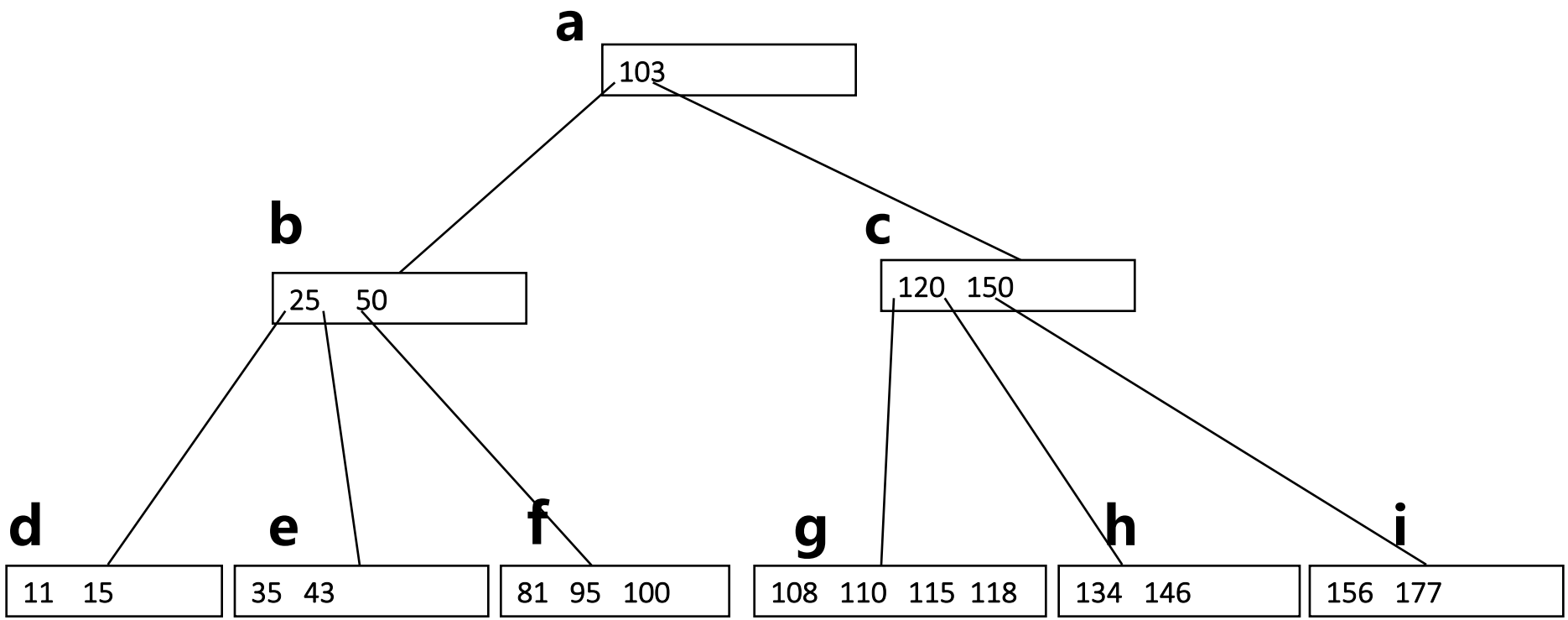
$= h + 2(h-1) + 3$

$= 3h + 1$

B树的删除

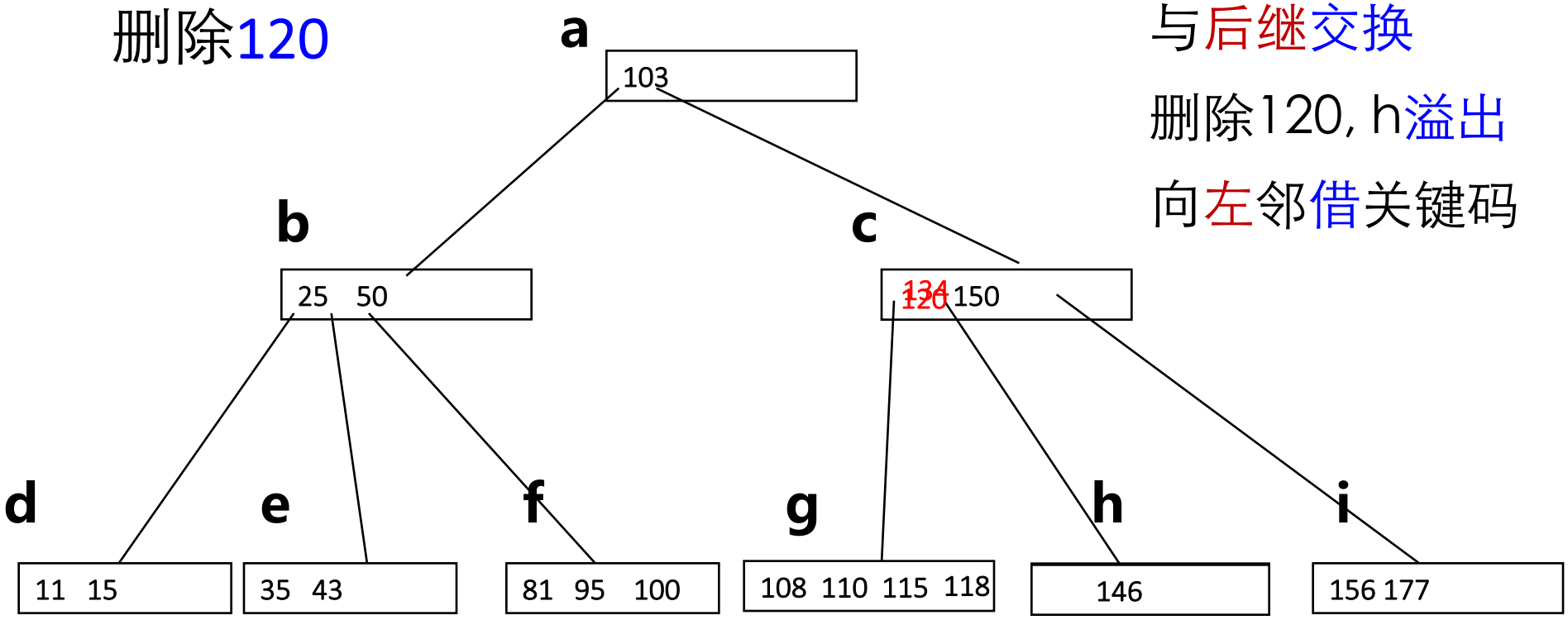
- **交换 – 删除 – 借关键码 – 合并**
  - 保证所删除关键码处在**最底层**，否则与其后继**交换**
  - 若删除后，结点中关键码数目不够最低限（下溢），则先看**左、右兄弟**有无多余的关键码可**借**，若有，则该结点与选中兄弟的所有关键码，连同父结点中的**分界码**，看成一个结点，采用插入算法的方法，**分成三部分**
  - 若其左、右兄弟中关键码数目都处在临界限，则把该结点与兄弟之一，以及父结点中相应的分界码一起**合并**成一个结点（父结点中关键码数目减少一个）
  - **合并过程**可能一直传递到根（树高降低一层）

B树删除示例： 5阶


a
103
b
25 50
c
120 150
d
11 15
e
35 43
f
81 95 100
g
108 110 115 118
h
134 146
i
156 177

B树删除示例： 5阶

删除120



与后继交换

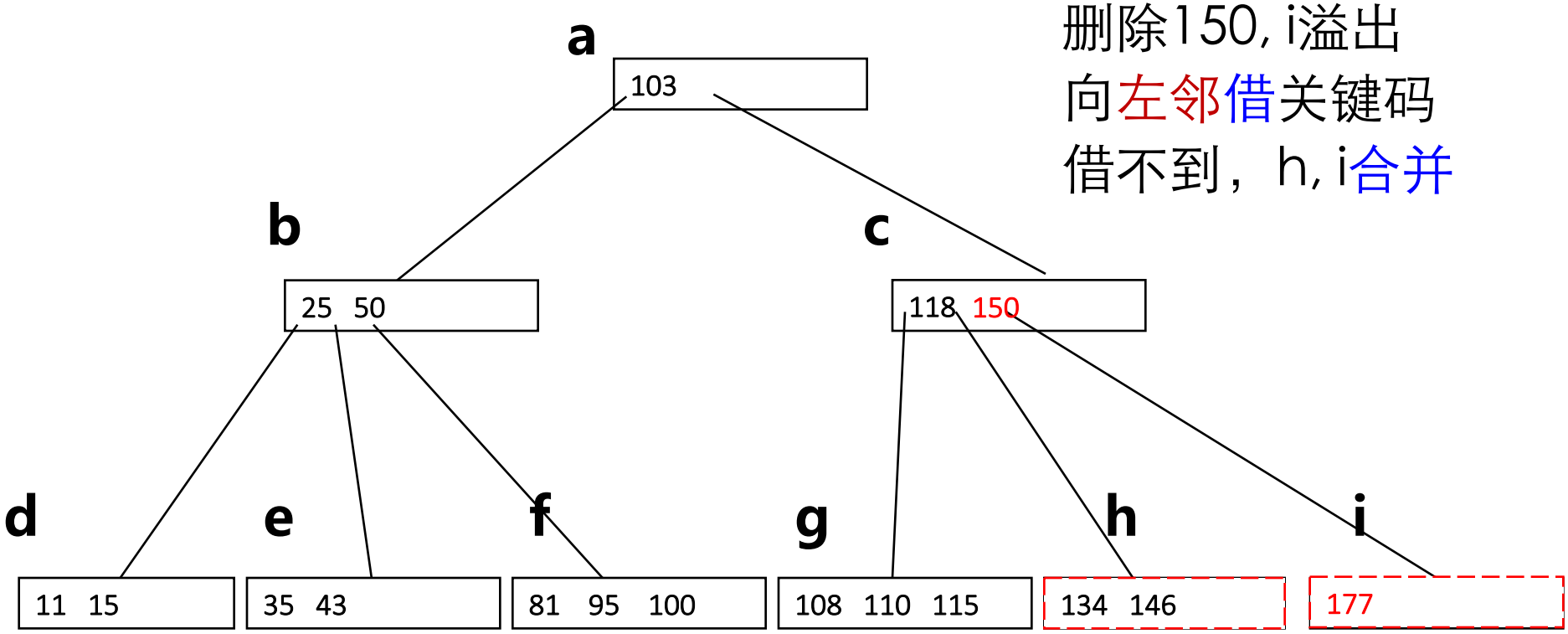
删除120, h溢出

向左邻借关键码

交换-删除-借关键码

B树删除示例： 5阶

继续删除150



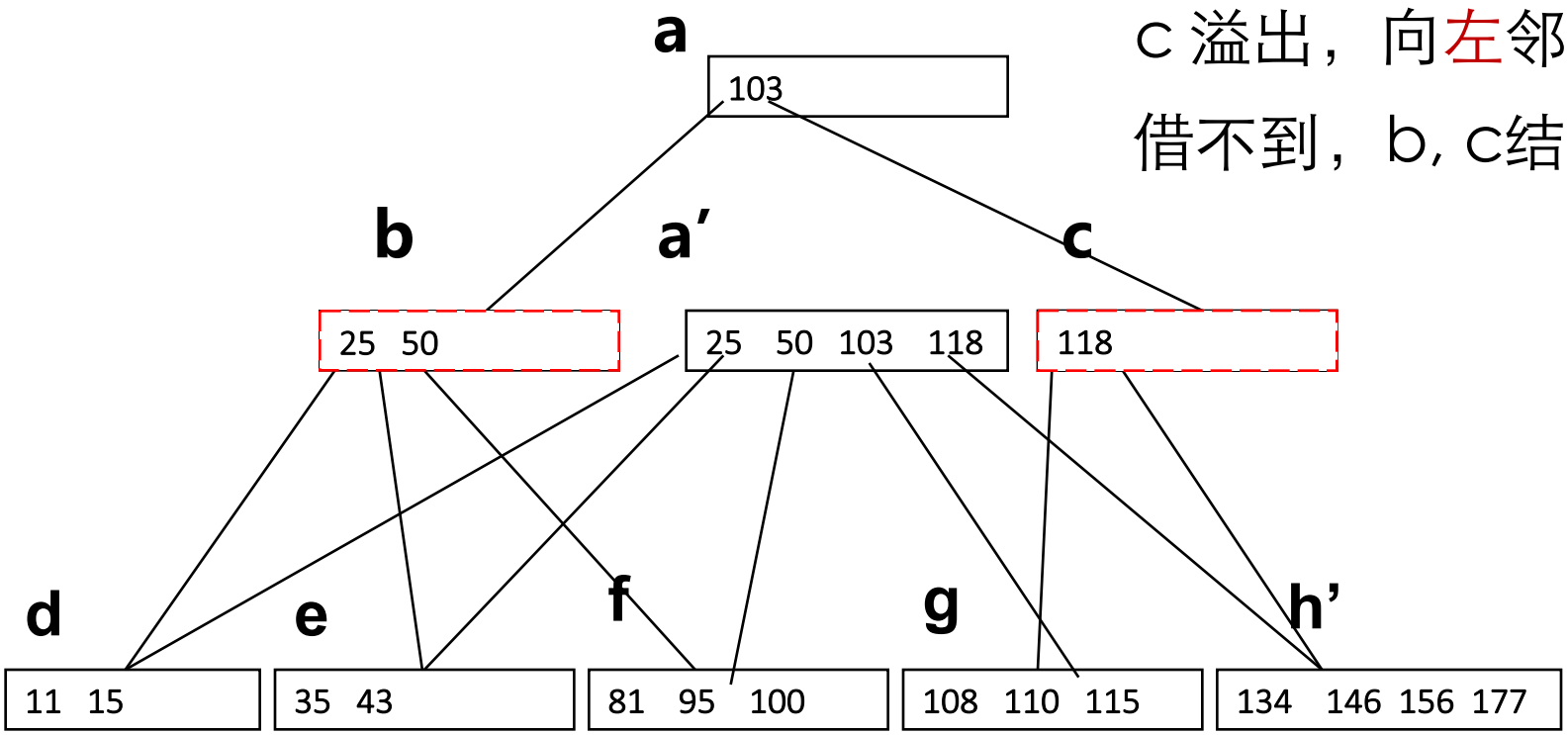
与后继交换
删除150, i溢出
向左邻借关键码
借不到， h, i合并

B树删除示例：5阶

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



读写次数?

B+树

- B树的一种**变形**
- **叶结点**上**存储信息**的树
  - 所有的关键码均出现在叶结点层
  - 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**，起到**限界**的作用

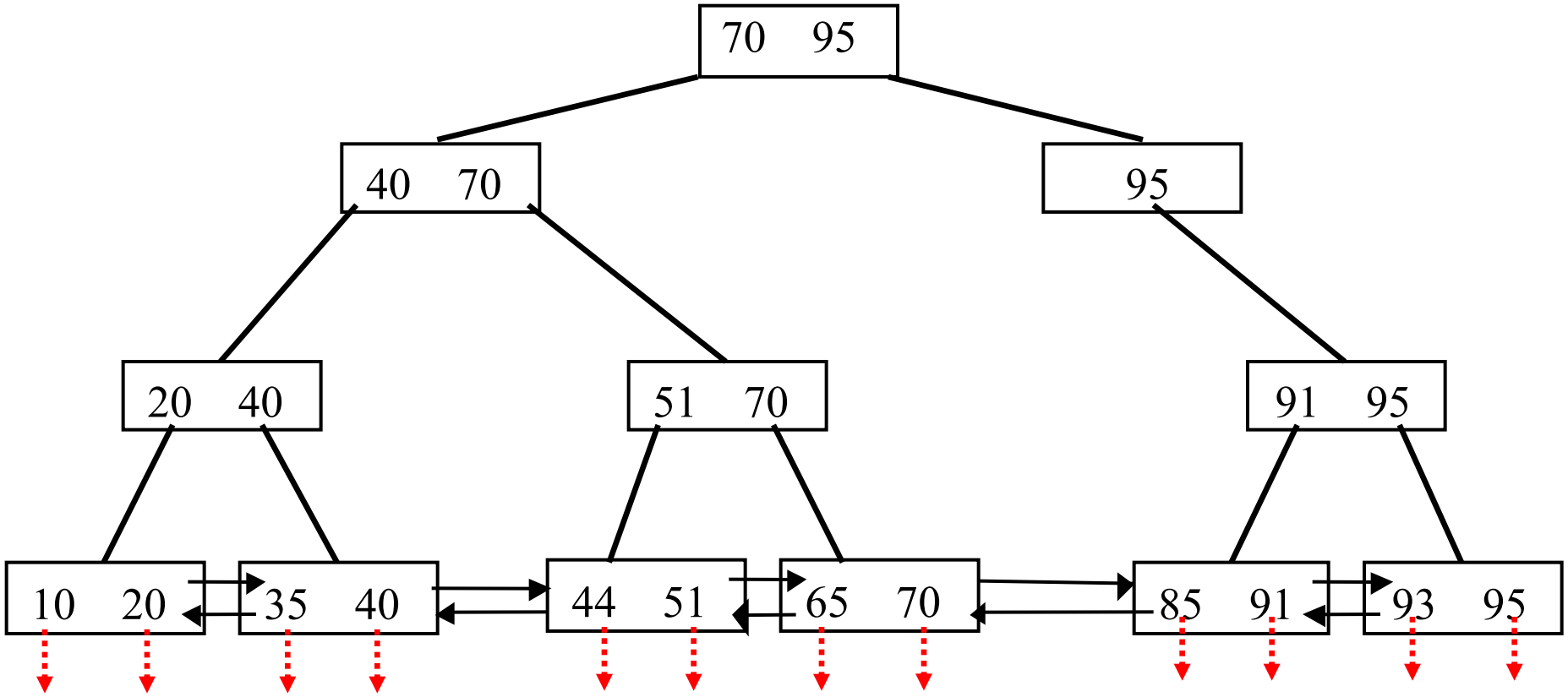
B+树的定义

m阶B+树满足：

- 每个结点**至多**有m个子结点；
- 每个结点（除根外）**至少**有 $\lceil m/2 \rceil$ 个子结点；
- 根结点**至少**有两个子结点；
- 有k个子结点的结点必有k个关键码（作为子结点的**上界** or **下界**）

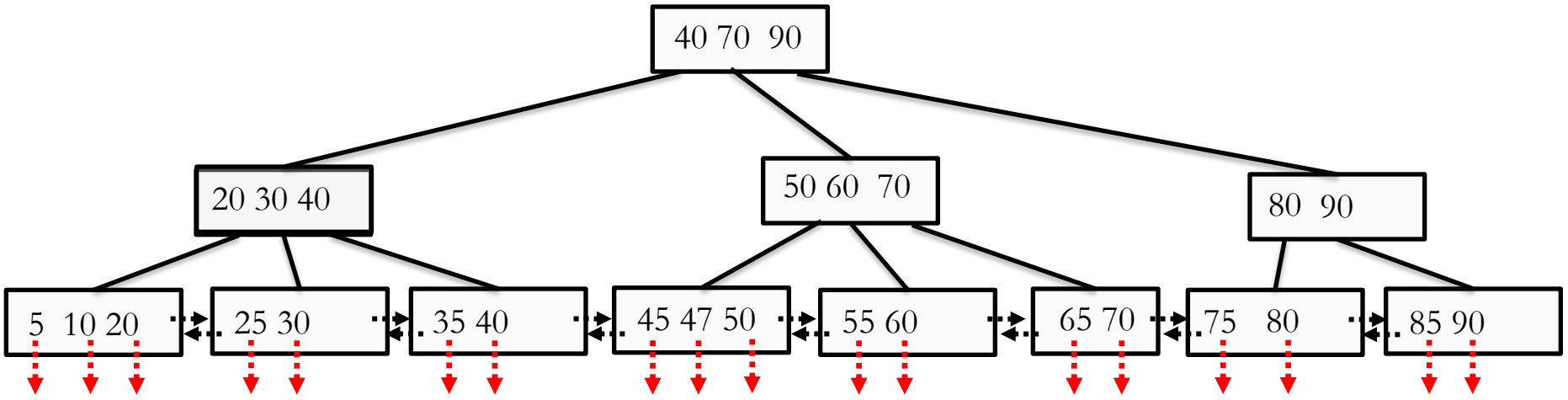
其实，考虑到从根开始建立B+树的特殊情况，根可以为空或者独根

2阶B+树示例



3 阶B+树示例

- 一般而言， B+树的阶 ≥ 3



B+树上的检索

- **检索到叶结点层**
  - 在上层找到待查关键码时**并不停止**
  - 需继续沿指针向下查到叶结点层的这个关键码为止

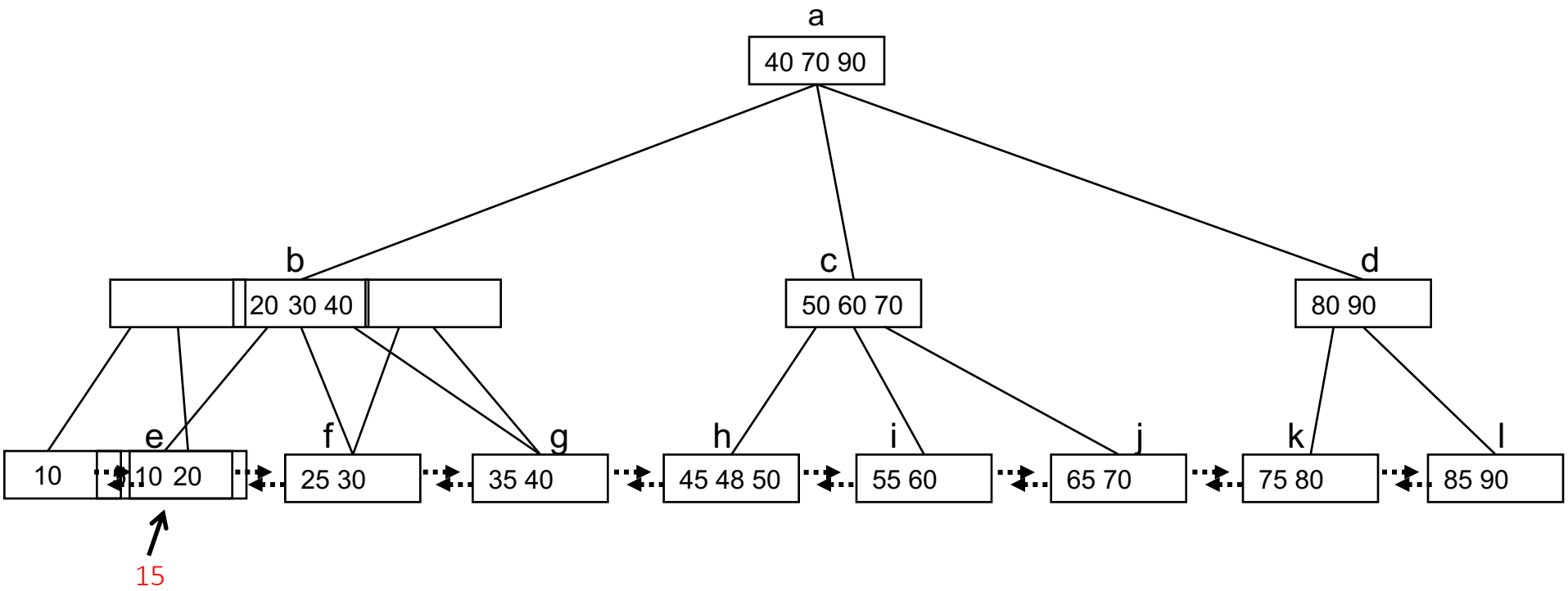
- **B^+树的叶结点可链接起来，形成一个双链表**
  - 适合顺序检索（范围检索）
  - 实际应用更为广泛

需要的话，每层结点也可顺序链接

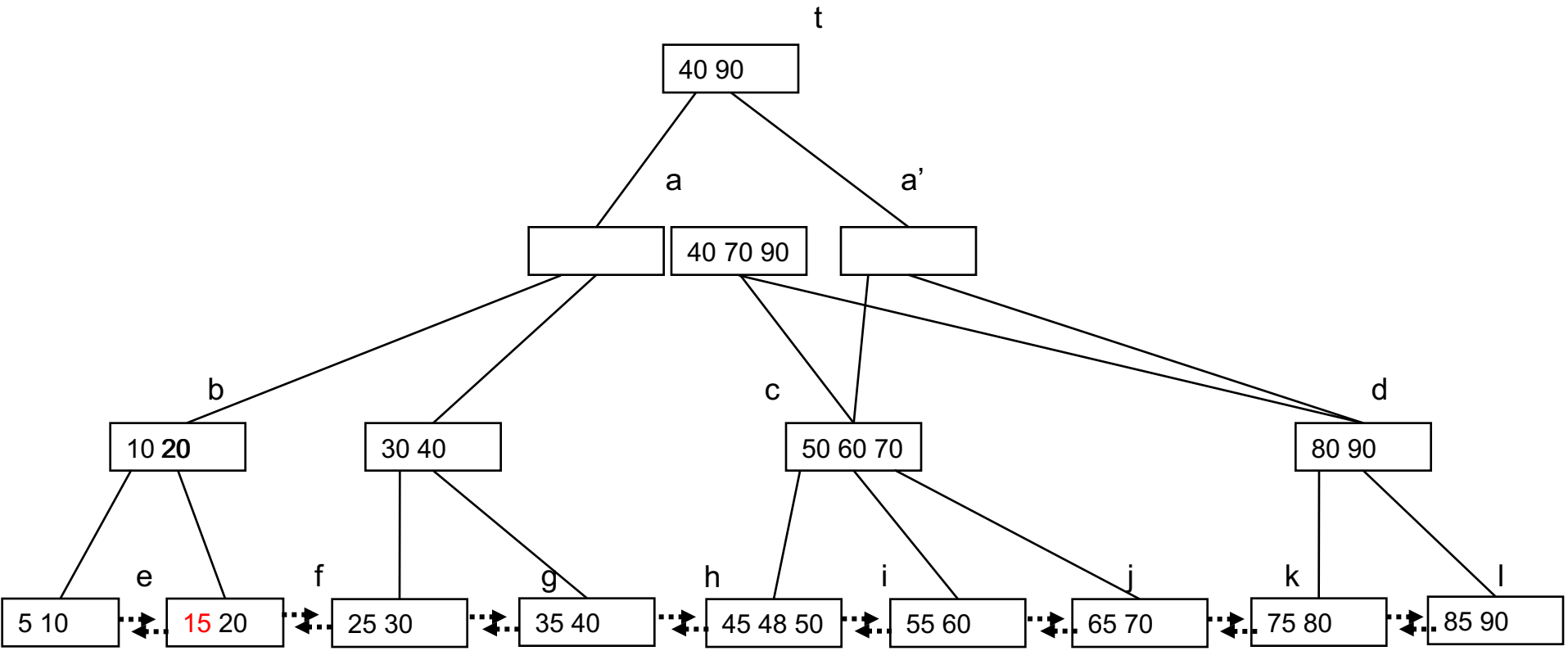
B+树的插入

- **伴生着分裂**
  - 过程与B树类似
  - 须保证上一层结点中包含下层分裂后的两个结点的最大（或最小）关键码

3阶B+树插入示例


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
45 48 50
55 60
65 70
75 80
85 90
15

3阶B+树插入示例：树高增大

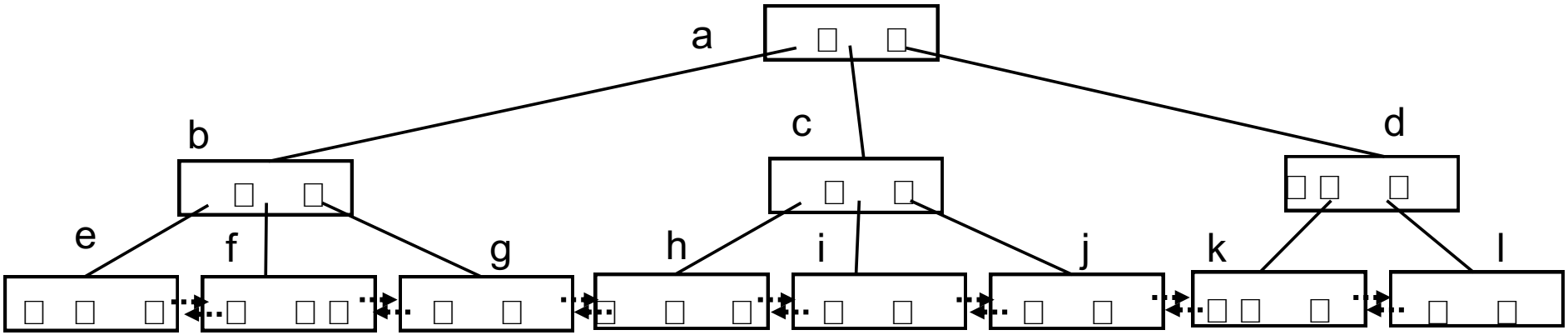


B+树的删除

- 当关键码下溢时，与左、右兄弟进行调整、合并的处理和B树类似
- 关键码在叶结点层删除后，其在上层的**复本**可保留，做为一个“**分界关键码**”存在
  - 也可替换为新的最大关键码（或最小关键码）

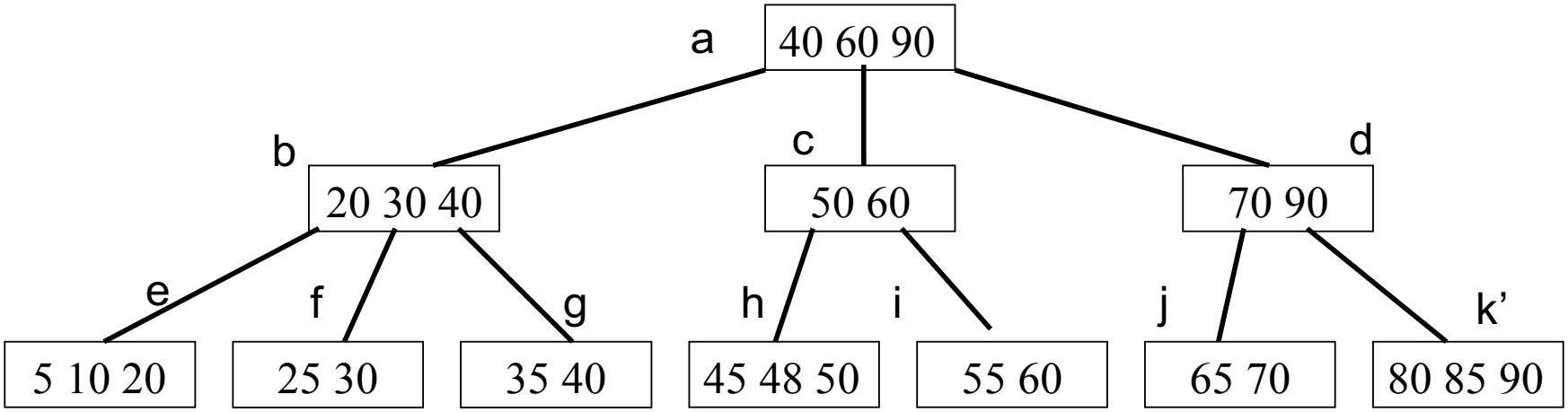
3阶B+树删除示例

删除75



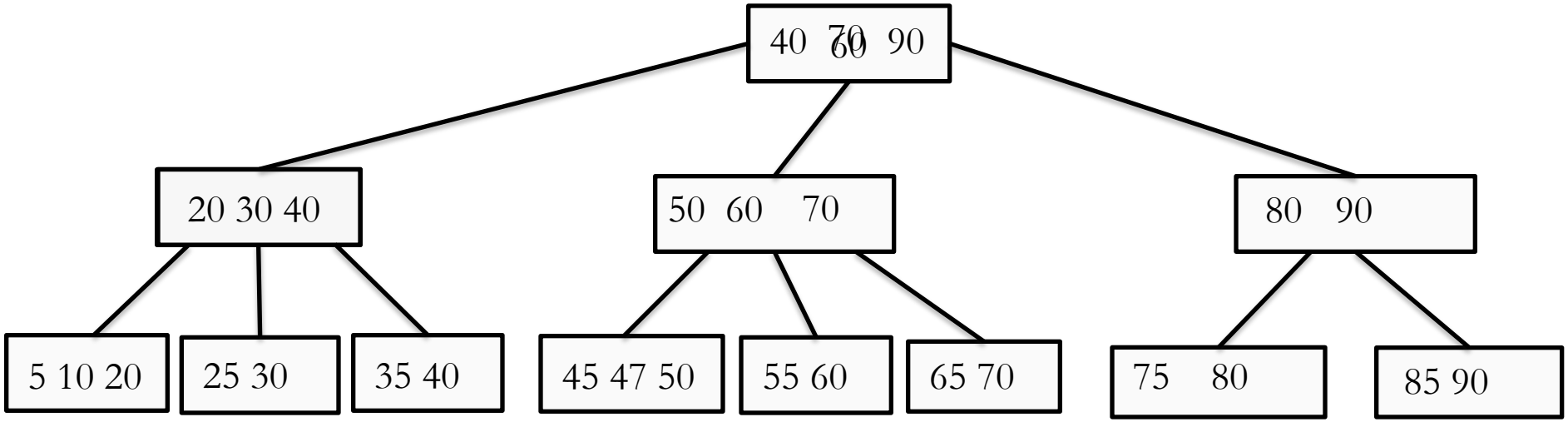
3阶B+树删除示例

- 沿a、d、k查找，找到叶结点
- 在k中删去75，发生下溢出，



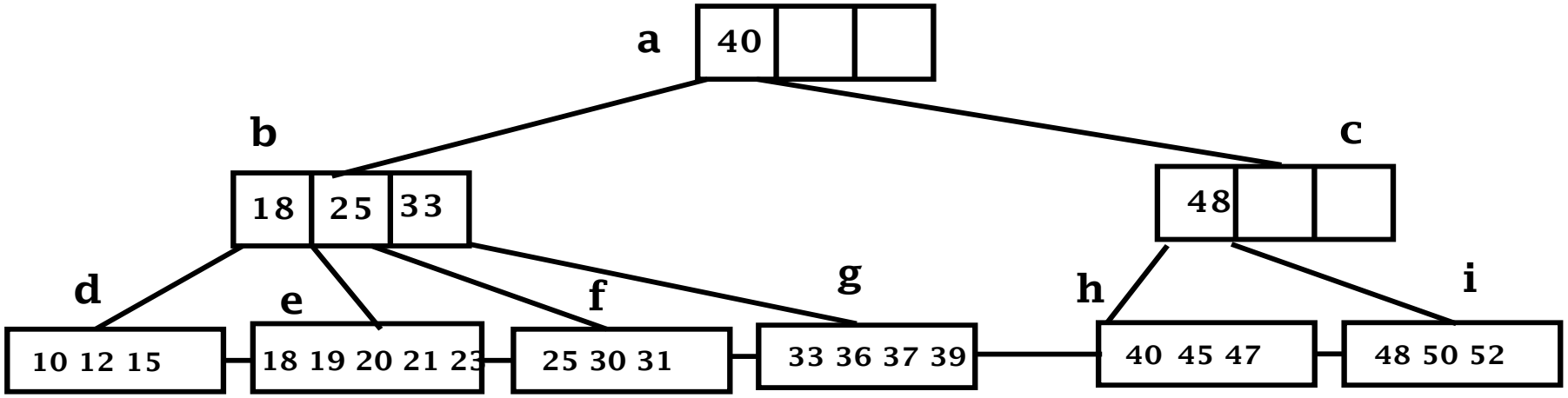
3阶B+树删除示例

1. 剩余关键码 80 与右邻合并为新结点（80,85, 90）
2. 父结点中原分界码 80 删除
3. 父结点下溢出
4. 借左邻的关键码，关键码平分
5. 根结点中的分界码70修改为60



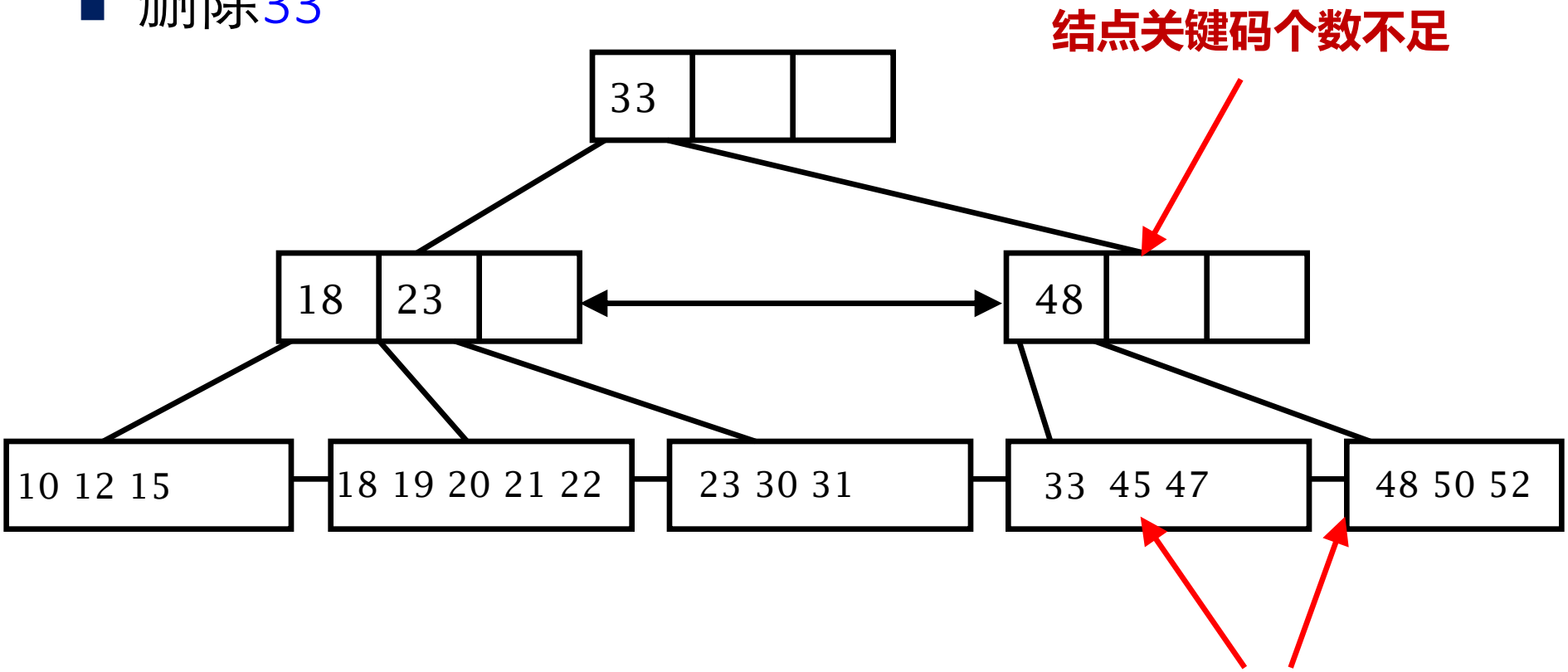
另一种B+树

- **叶结点中关键码数目与非叶结点的不同**
  - 内部非叶结点构成B树
  - **叶**的阶与**B+树**一致
  - e.g.，叶结点阶5，内部阶4



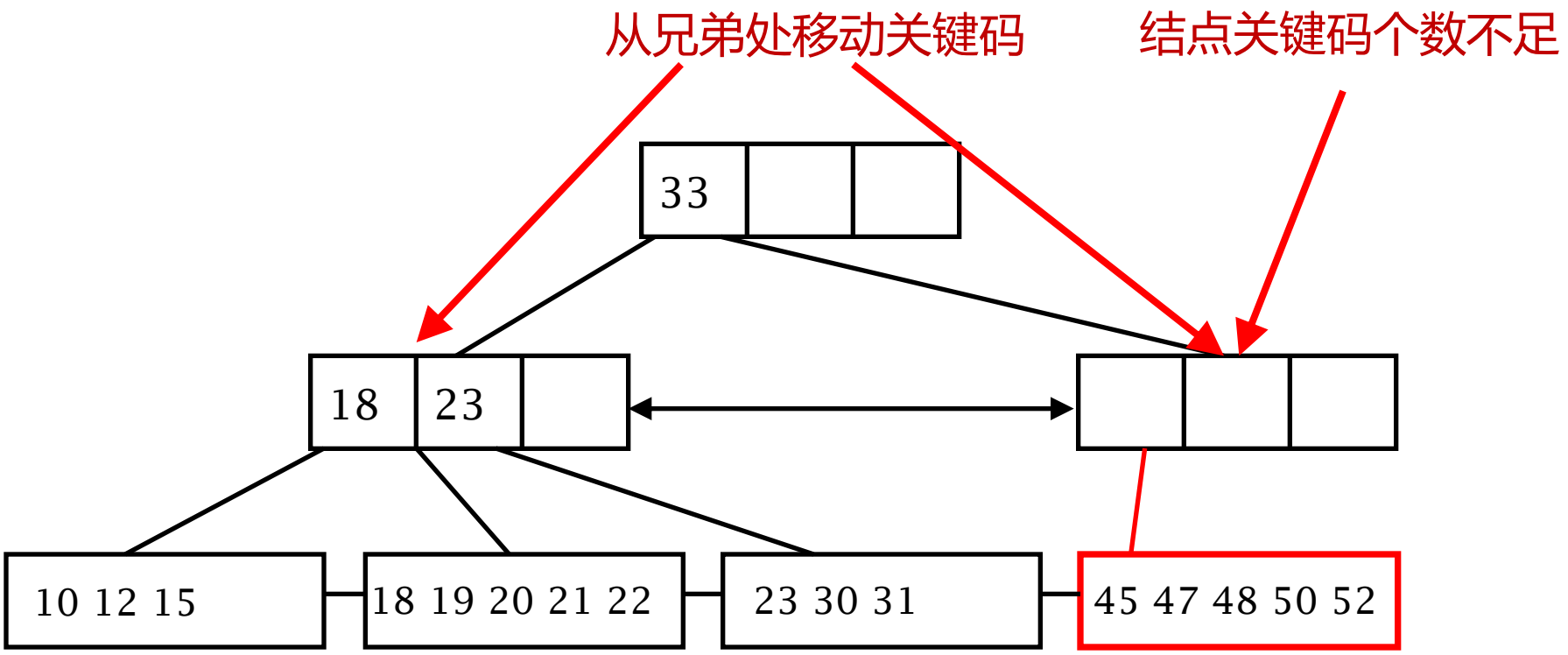
另一种B+树： 删除示例

- 删除33

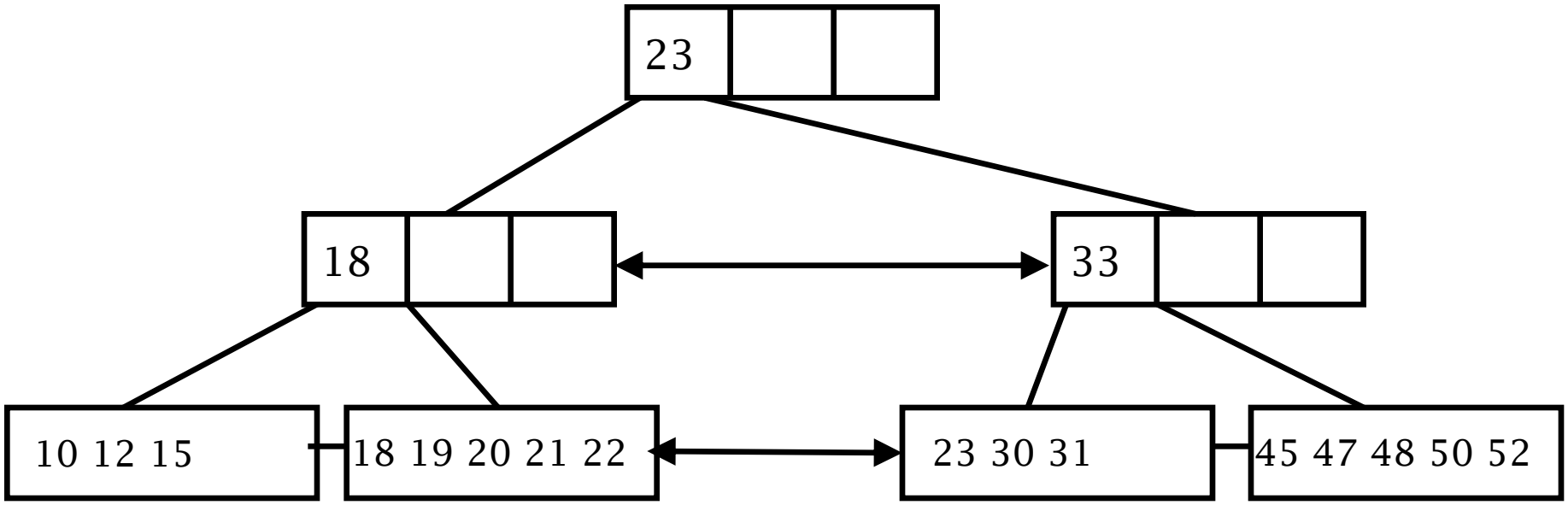


叶结点阶 5，内部阶 4

另一种B+树： 删除示例

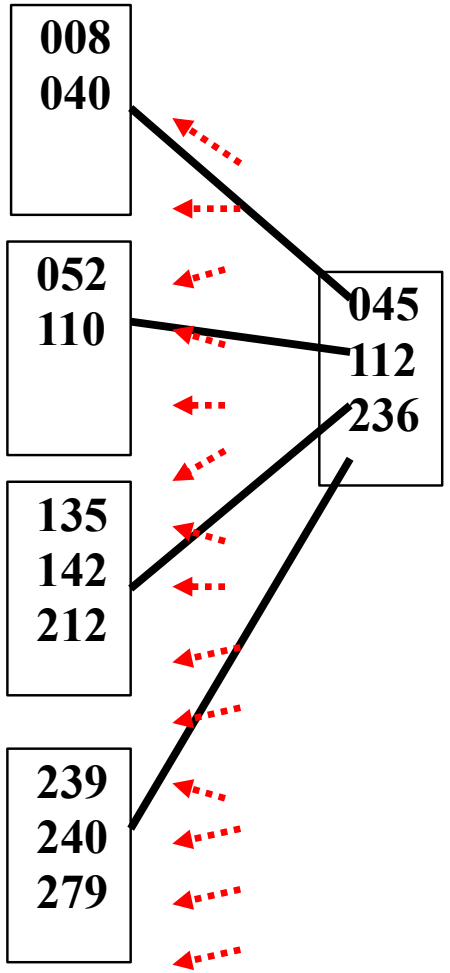

从兄弟处移动关键码
结点关键码个数不足
33
18
23
10 12 15
18 19 20 21 22
23 30 31
45 47 48 50 52

另一种B+树： 删除示例



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 1层至少2 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

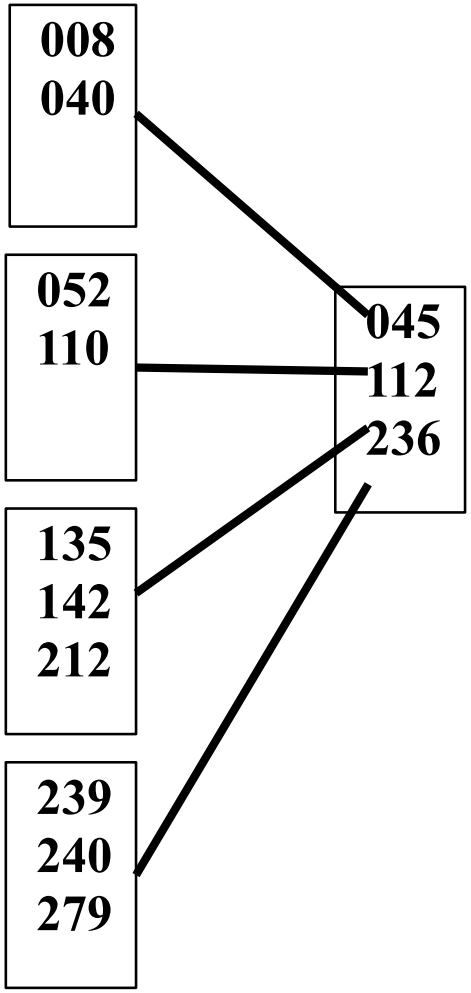


$$N+1\geq 2\cdot\lceil m/2\rceil^{k-1},\qquad k\leq 1+\log_{\lceil m/2\rceil}(\frac{N+1}{2})$$

B树的性能分析

- **示例**
  - N = 1,999,998，m = 199 **时**
    - k = 4
    - **一次检索最多** 4 **层** $k \le 1 + \log_{\lceil m/2 \rceil}(\frac{N+1}{2})$

结点分裂次数



- 设关键码数为N（指针$N+1$），内部结点数为 p

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

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

- **最差情况**下每插入一个结点都经过分裂（除第1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

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

B树存储效率分析

- **整个B 树结构关键码不重复**
  - 每个关键码实质上是二元组（关键码，页块地址）
    - 其中隐含记录地址指针： 隐含指针

B+树的存储效率

- **多级索引形式**
  - 最下层为所有关键码的全集
    - 故，此层可组织成顺序链表
  - 非叶层结点中的关键码不需要隐含指针

B+ tree PK B tree： 存储效率

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

$$\lceil \log_{0.75m} N \rceil$$

B^+ tree PK B tree： 存储效率

- 可容纳 *m*个（关键码，子结点页块指针）结点，假设关键码所占字节数与指针相同
  - 最多可容纳B树的（关键码，隐含指针，子结点页块指针）的*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+ tree PK B tree

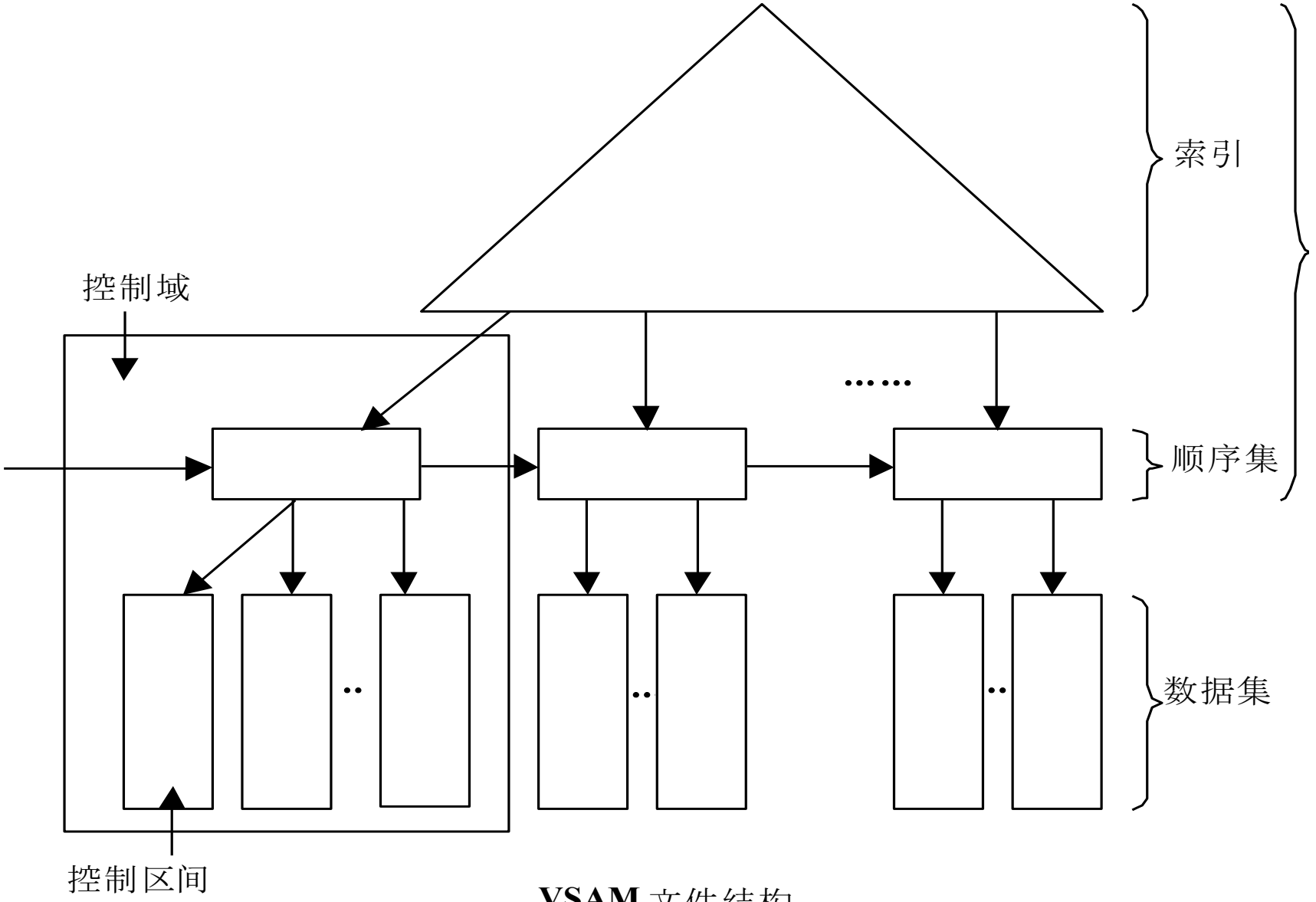
- **B+树的存储效率更高**
  - 检索层次更少（树较矮）
- **操作更方便**
  - 插入删除操作更方便
  - 树型结构随机存取
  - 叶层的双链顺序访问
- **∴ B+树应用得更为广泛**

补充：VSAM

- VSAM (Virtual Storage Access Method)： **虚拟存储存取方法**
  - B^+树的应用
  - 一种索引顺序文件的组织方式
  - 与存储设备无关，存储单位是“逻辑”的

VSAM的组成

- 索引集
- 顺序集（顺序集索引）
  - 和索引集共同形成了B^+树结构的文件索引
- 数据集
  - 存放文件记录
  - 记录可以是变长的



VSAM 文件结构

思考

- 是否 存在符合定义的 2阶B树？ 若有的话是否具有实用价值？ 为什么？
- B树删除是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分裂的方法?
- B树的定义中关于度的定义为⌈m/2⌉到m之间，可否调整为其他范围?
- 为什么相比于B+树，B树存储效率低?

B家族树不适用的场合

- **B树适于场景：查询并返回少量记录**
- **对于数据仓库的复杂交互式查询，B 树则存在明显缺点：**
  - B树对取值极少的（低基数）数据字段几乎毫无价值
  - 在数据仓库中构造和维护索引的代价高
  - 对于带有分组及聚合条件的复杂查询无能为力

位索引技术

Bit-Map（位图）索引

- **假设文件的记录数为 *n*， 则其数据域 *x* （某个字段） 的一个位图索引是一个长度为n的位向量的集合**
  - 每一个位向量对应于域 *x* 可能出现的值
  - 如果第*i*个记录的数据域 *x* 值为v，那么对应于值v的位向量在位置*i*上的取值为1；否则该向量的位置*i*上的取值为0

数据库表的位图索引

date	store	state	c lass	sales	State＝NY	Class=A
3/1	32	NY	A	6	1	1
3/1	36	AL	A	9	0	1
3/1	38	NY	B	5	1	0
3/1	41	AK	A	11	0	1
3/1	43	NY	A	9	1	1
3/1	46	AK	B	3	0	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	

特征文件

- Signature file （也译为“签名文件”）
- **倒排表**

(30,foo), (30,bar), (30,baz), (40,baz), (40,bar), (50,foo)

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

1 表示记录中出现了相应的字段值

位图索引特点

- 按“**列**”为单位存储数据
- 列数据比行数据更易进行压缩，可节省50%的磁盘空间
- 索引空间比B树**小**

ToDo

- 请调研列数据库中的位图索引