第8章：索引结构

Index Structures

李东博

哈尔滨工业大学
计算学部
物联网与泛在智能研究中心
电子邮件：ldb@hit.edu.cn

2023年春

教学内容[1]

1. Indexes
2. Hash-based Index Structures
   - Extensible Hash Tables
   - Linear Hash Tables
3. Tree-based Index Structures
   - B+ Trees
4. Log-Structured Merge-Trees (LSM-Trees)
5. Advanced Topics in Indexing

[1] 更新于2023年2月16日

Indexes

索引(Index)

索引能够帮助DBMS快速找到关系中满足搜索条件的元组

- 索引对于提高查询处理效率至关重要 演示

Example (索引)

索引

Sname	元组地址
Abby	$addr_3$
Ed	$addr_2$
Elsa	$addr_1$
Nick	$addr_4$

地址
$addr_1$
$addr_2$
$addr_3$
$addr_4$

Student关系

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
PH-001	Nick	M	20	Physics

查询: `SELECT Sdept FROM Student WHERE Sname = 'Elsa';`

- 如果没有索引，则只能通过扫描Student关系来完成查询
- 如果有上述索引，则可以通过该索引来快速找到元组

索引的分类

按照索引的实现方式，可将索引分为两类

- 有序索引(ordered index): 通过按索引键有序排列索引项来实现索引
- 哈希索引(hash index): 通过按索引键哈希值分桶来实现索引

有序索引(Ordered Index)

索引键(index key): 索引根据一组属性(索引键)来定位元组

- 索引记录了元组的索引键值与元组地址的对应关系

索引项(index entry): 索引中的(键值，地址)对

- 有序索引中的索引项按索引键值排序

Example (有序索引)

有序索引

Sname	元组地址
Abby	$addr_3$
Ed	$addr_2$
Elsa	$addr_1$
Nick	$addr_4$

Student关系

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

哈希索引(Hash Index)

哈希索引由若干桶(bucket)构成

- h: 哈希函数
- 键为K的索引项属于编号为$h(K)$的桶
- 哈希索引只支持索引键上的等值查找

Example (哈希索引)

哈希索引

桶0

h(Sdept)	地址
h('Math') = 0	$addr_3$

桶1

h(Sdept)	地址
h('CS') = 1	$addr_1$
h('CS') = 1	$addr_2$

桶2

h(Sdept)	地址
h('Physics') = 2	$addr_4$

Student关系

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

有序索引的分类(一)

根据数据文件中的元组是否按索引键排序，可将有序索引可分为两类

- 聚簇索引(clustered index)
- 非聚簇索引(nonclustered index)

聚簇索引(Clustered Index)

如果数据文件中的元组是按索引键排序的，则索引是聚簇索引 演示

- 聚簇索引的索引键通常是关系的主键
- 一个关系上通常只有一个聚簇索引(为什么?)

Example (聚簇索引)

聚簇索引

Sno	元组地址
CS-001	$addr_1$
CS-002	$addr_2$
MA-001	$addr_3$
PH-001	$addr_4$

地址
$addr_1$
$addr_2$
$addr_3$
$addr_4$

Student关系

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
PH-001	Nick	M	20	Physics

非聚簇索引(Nonclustered Index)

如果数据文件中的元组不是按索引键排序的，则索引是非聚簇索引

- 一个关系上可以有多个非聚簇索引

Example (非聚簇索引)

非聚簇索引

Sname	元组地址
Abby	$addr_3$
Ed	$addr_2$
Elsa	$addr_1$
Nick	$addr_4$

Student关系

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

索引组织表(Index-Organized Table)

索引组织表 = 聚簇索引文件 + 数据文件

- 在聚簇索引的索引项中存储元组本身，而不是元组地址
- 无需根据元组地址从磁盘读元组，减少1次I/O

Example (索引组织表)

索引组织表

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
PH-001	Nick	M	20	Physics

有序索引的分类(二)

根据关系中每个元组在索引中是否都有一个对应索引项，可将有序索引可分为两类

- 稠密索引(dense index)
- 稀疏索引(sparse index)

稠密索引(Dense Index)

如果关系中每个元组在索引中都有一个对应索引项，则索引是稠密索引

- 非聚簇索引一定是稠密索引

Example (稠密索引)

非聚簇索引

Sname	元组地址
Abby	$addr_3$
Ed	$addr_2$
Elsa	$addr_1$
Nick	$addr_4$

地址
$addr_1$
$addr_2$
$addr_3$
$addr_4$

Student关系

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
PH-001	Nick	M	20	Physics

稀疏索引(Sparse Index)

如果关系中只有部分元组在索引中有对应索引项，则索引是稀疏索引

- 聚簇索引通常是稀疏索引
- 可以只对数据文件每页中的第一个元组建立索引项
- 可以只对数据文件每个不同的索引键值的第一个元组建立索引项

Example (稀疏索引)

聚簇索引

Sno	元组地址
CS-001	$addr_1$
MA-001	$addr_3$

Student关系

	Sno	Sname	Ssex	Sage	Sdept
$Page_1$					
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$Page_2$	Sno	Sname	Ssex	Sage	Sdept
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

有序索引的分类(三)

根据索引键是否为关系的主键，可将有序索引可分为两类

- 主索引(primary index)
- 二级索引(secondary index)

主索引(Primary Index)

主索引的索引键是关系的主键

- 一个关系只有一个主索引

Example (主索引)

主索引

Sno	元组地址
CS-001	$addr_1$
CS-002	$addr_2$
MA-001	$addr_3$
PH-001	$addr_4$

Student关系

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

二级索引(Secondary Index)

二级索引的索引键不是关系的主键

- 二级索引通常是非聚簇索引
- 一个关系可以有多个二级索引

Example (二级索引)

二级索引

Sname	元组地址
Abby	$addr_3$
Ed	$addr_2$
Elsa	$addr_1$
Nick	$addr_4$

Student关系

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

创建主索引

- 在CREATE TABLE或ALTER TABLE语句中使用PRIMARY KEY声明主键时，自动建立主索引
- 只能在CREATE TABLE或ALTER TABLE语句中声明主索引

Example (创建主索引)

```
CREATE TABLE Student (
  Sno CHAR(6),
  Sname VARCHAR(10),
  Ssex CHAR,
  Sage INT,
  Sdept VARCHAR(20),
  PRIMIARY KEY (Sno));
```

创建二级索引

语句: CREATE INDEX 索引名 ON 关系名(索引键)

- 用ASC或DESC声明索引属性的排序方式

Example (创建二级索引)

```
CREATE INDEX idx_sname_sage ON Student (Sname, Sage DESC);
```

MySQL中的索引[2]

- 主索引是索引组织表
- 二级索引的索引项中存储的不是元组地址，而是元组的主键值

Example (MySQL中的主索引和二级索引)

MySQL二级索引

Sname	Sno
Abby	MA-001
Ed	CS-002
Elsa	CS-001
Nick	PH-001

MySQL主索引/索引组织表

Sno	Sname	Ssex	Sage	Sdept
CS-001	Elsa	F	19	CS
CS-002	Ed	M	19	CS
MA-001	Abby	F	18	Math
PH-001	Nick	M	20	Physics

思考题

分析MySQL索引设计的优缺点

[2]MySQL InnoDB存储引擎

唯一索引(Unique Index)

唯一索引(unique index)的索引键值不能重复

- 主索引一定是唯一索引
- 二级索引不一定是唯一索引

Example (唯一索引vs 非唯一索引)

唯一索引

Sno	元组地址
CS-001	$addr_2$
CS-002	$addr_3$
MA-001	$addr_4$
PH-001	$addr_1$

Student关系

地址	Sno	Sname	Ssex	Sage	Sdept
$addr_1$	CS-001	Elsa	F	19	CS
$addr_2$	CS-002	Ed	M	19	CS
$addr_3$	MA-001	Abby	F	18	Math
$addr_4$	PH-001	Nick	M	20	Physics

创建唯一索引

创建唯一索引有两种方法

- 在CREATE TABLE或ALTER TABLE语句中使用UNIQUE声明唯一约束时，自动创建唯一索引
- 使用语句: CREATE UNIQUE INDEX 索引名 ON 关系名(索引键)

Example (创建唯一索引)

```
CREATE TABLE Student (
  Sno CHAR(6) PRIMARY KEY,
  Sname VARCHAR(10),
  Ssex CHAR,
  Sage INT,
  Sdept VARCHAR(20),
  UNIQUE (Sname));

CREATE UNIQUE INDEX ukey_sname ON Student(Sname);
```

外键索引(Foreign Key Index)

外键索引的索引键是关系的外键

- 当被参照关系的元组被删除时，外键索引可以加快参照完整性检查
- 当被参照关系的元组的主键值被修改时，外键索引可以加快参照完整性检查
- `ON DELETE|UPDATE [NO ACTION|RESTRICT|CASCADE|SET NULL|DEFAULT]`

Example (外键索引)

外键索引

Sno	元组地址
CS-001	$addr_4$
CS-001	$addr_5$
CS-002	$addr_6$
MA-001	$addr_7$
PH-001	$addr_1$
PH-001	$addr_2$
PH-001	$addr_3$

SC

地址	Sno	Cno	Grade
$addr_1$	PH-001	1002	92
$addr_2$	PH-001	2003	85
$addr_3$	PH-001	3006	88
$addr_4$	CS-001	1002	95
$addr_5$	CS-001	3006	90
$addr_6$	CS-002	3006	80
$addr_7$	MA-001	1002	

创建外键索引

- 在CREATE TABLE或ALTER TABLE语句中使用FOREIGN KEY声明外键时，会为外键创建索引

Example (创建外键索引)

```
CREATE TABLE SC (
  Sno CHAR(6),
  Cno CHAR(4),
  Grade INT,
  PRIMARY KEY (Sno, Cno),
  FOREIGN KEY (Sno) REFERENCES Student);
```

删除索引

删除二级索引

- PostgreSQL语句: `DROP INDEX 索引名;`
- MySQL语句: `DROP INDEX 索引名 ON 关系名;`
- 删除二级索引后不需要重新组织关系中的元组

删除主索引

- PostgreSQL中不能直接删除主索引，只能删除主键约束
- MySQL语句: `DROP INDEX 'PRIMARY' ON 关系名;`
- 删除主索引后需要重新组织关系中元组

索引结构(Index Structures)

有序索引的数据结构

- 平衡树
- 跳表(skiplist)：多用于内存数据库系统
- 字典树(trie)：多用于内存数据库系统
- 日志结构合并树(log-structured merge-tree, LSM-tree)：多用于NoSQL数据库系统的存储引擎

哈希索引的数据结构

- 哈希表

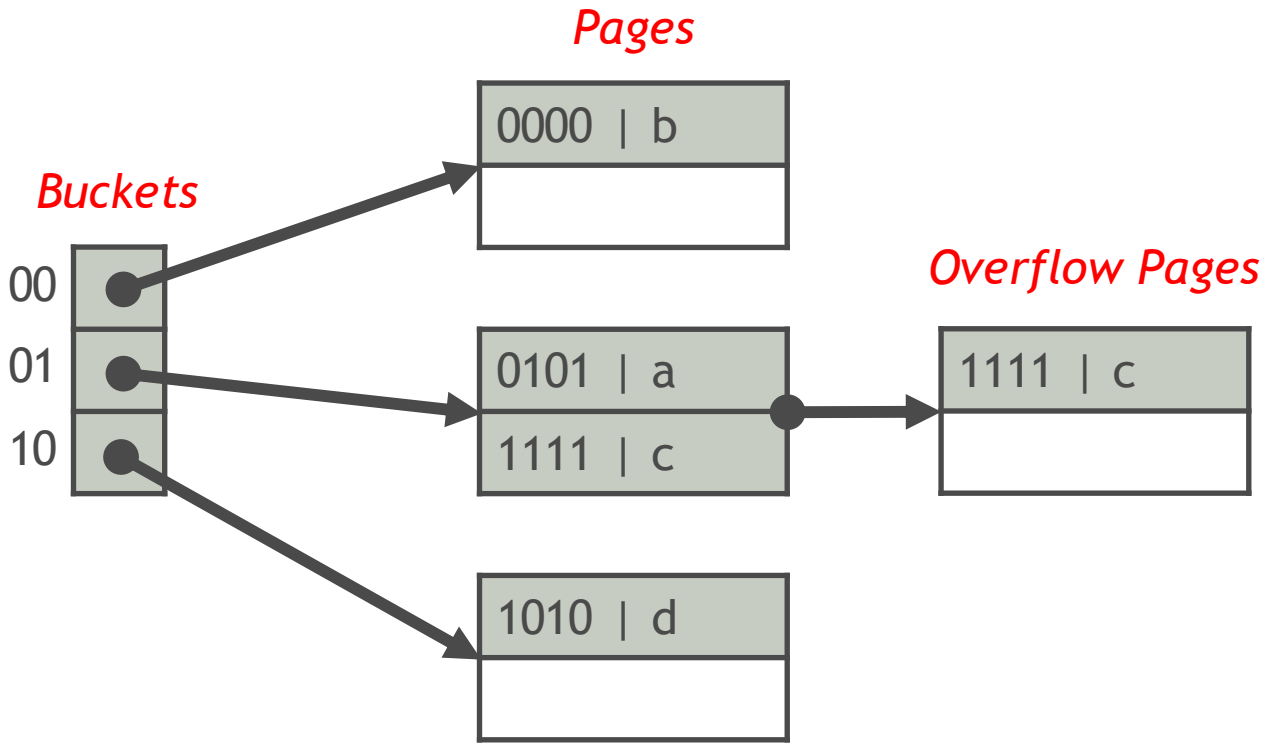
不同的索引结构具有不同的功能和性能

Hash-based Index Structures

外存哈希表(Secondary-Storage Hash Tables)

一个外存哈希表包含多个桶(bucket)

- 设$hash$是一个哈希函数，键为K的索引项(index entry)属于编号为$hash(K)$的桶
- 每个桶中存放一个指针，指向存储该桶中索引项的页的链表



外存哈希表的分类

静态哈希表(Static Hash Tables)

- 桶的数量固定不变

动态哈希表(Dynamic Hash Tables)

- 桶的数量动态变化，使每个桶中的索引项存储在大约1个页中
- 可扩展哈希表(extensible hash tables)
- 线性哈希表(linear hash tables)

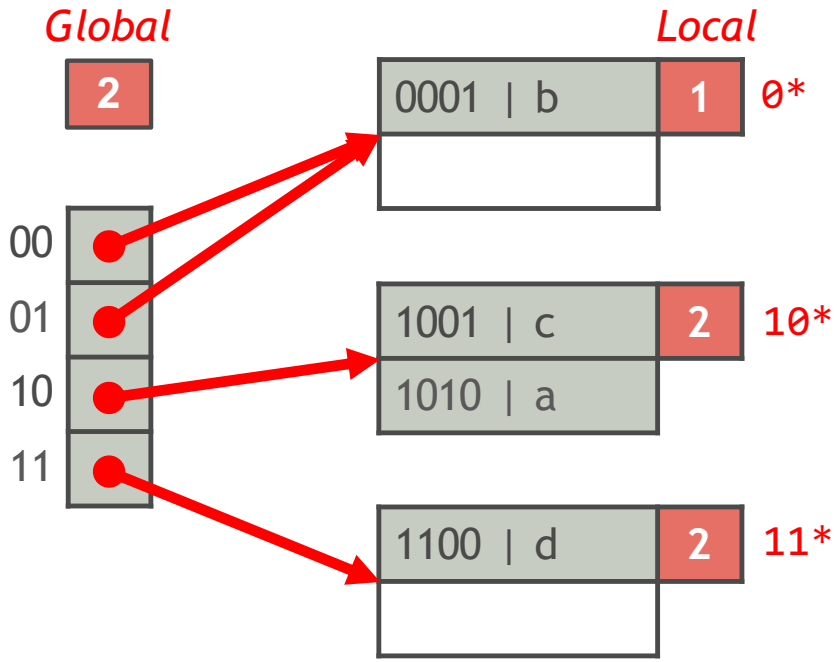
Hash-based Index Structures
Extensible Hash Tables

可扩展哈希表(Extensible Hash Tables)

一个可扩展哈希表包含2^i个桶

- i: 全局深度(global depth)
- 键值为K的索引项属于编号等于$hash(K)$的前i位的桶

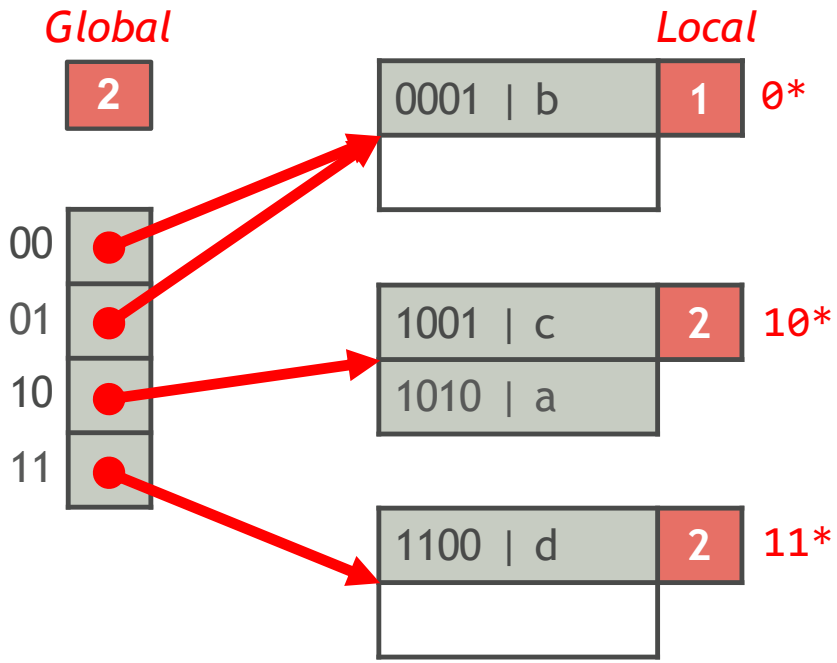
例: $hash(a) = 1010, hash(b) = 0001, hash(c) = 1001, hash(d) = 1100$



可扩展哈希表(续)

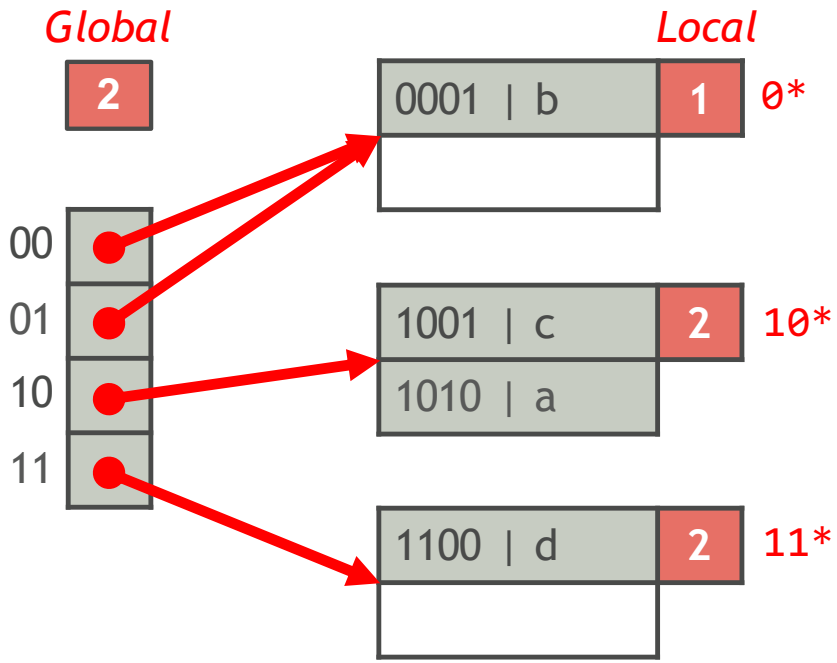
每个桶中存放一个指针，指向存储该桶中索引项的页

- 每个桶均没有溢出页(overflow page)
- 如果容纳得下，多个相邻桶中的全部索引项可以存入同一个页
- 每个页记录一个局部深度(local depth) j，该页中的全部索引项的$hash(K)$的前j位相同，用于标识这些索引项都存于这个页



可扩展哈希表的性质

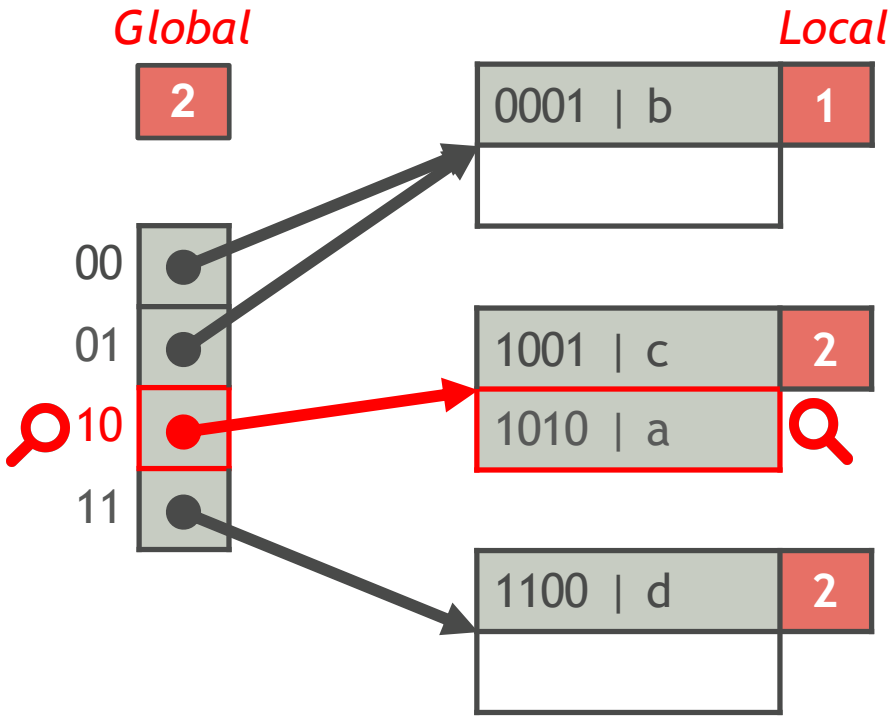
- 桶数$= 2^{\text{global_depth}}$
- 全局深度$\geq$每个页的局部深度
- 一个页被多个桶共享当且仅当这个页的局部深度小于全局深度
- 设一个页的局部深度为j，则页中全部索引项的$hash(K)$的前j位相同



查找索引项

1. 确定索引项所属的桶
2. 在桶指向的页中查找索引项

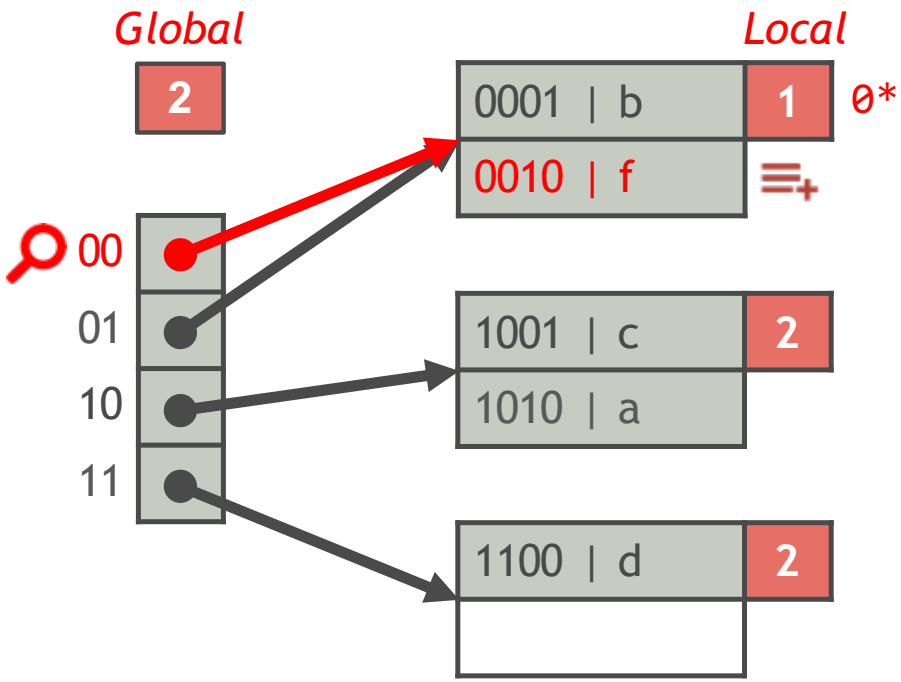
例: $K = a$, $hash(a) = 1010$



插入索引项

1. 找到索引项被插入的页P
2. 如果P中有足够的空闲空间，则将索引项插入P中；否则，分裂P

例: $K = f$, $hash(f) = 0010$

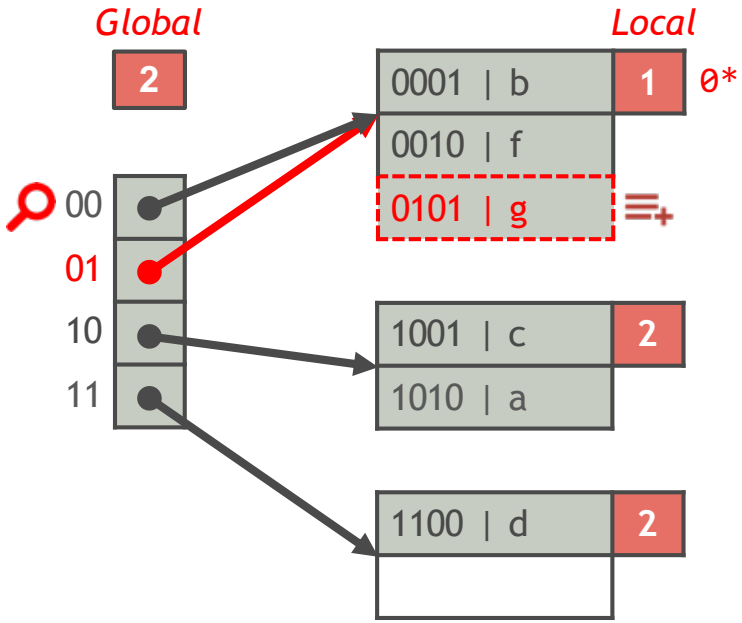


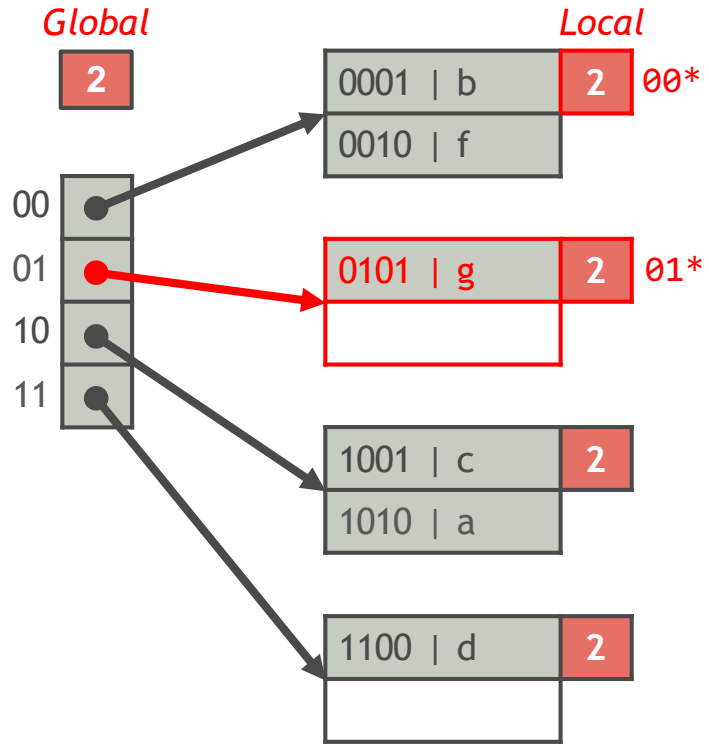
插入索引项(续)

如果P溢出且P的局部深度小于全局深度

1. 将P的局部深度j加1
2. 创建一个新页P'，令P和P'的局部深度相同
3. 根据键的哈希值的前j位，将P中索引项在P和P'中重新分配
4. 更新指向P的桶中的指针

例: $K = g$, $hash(g) = 0101$



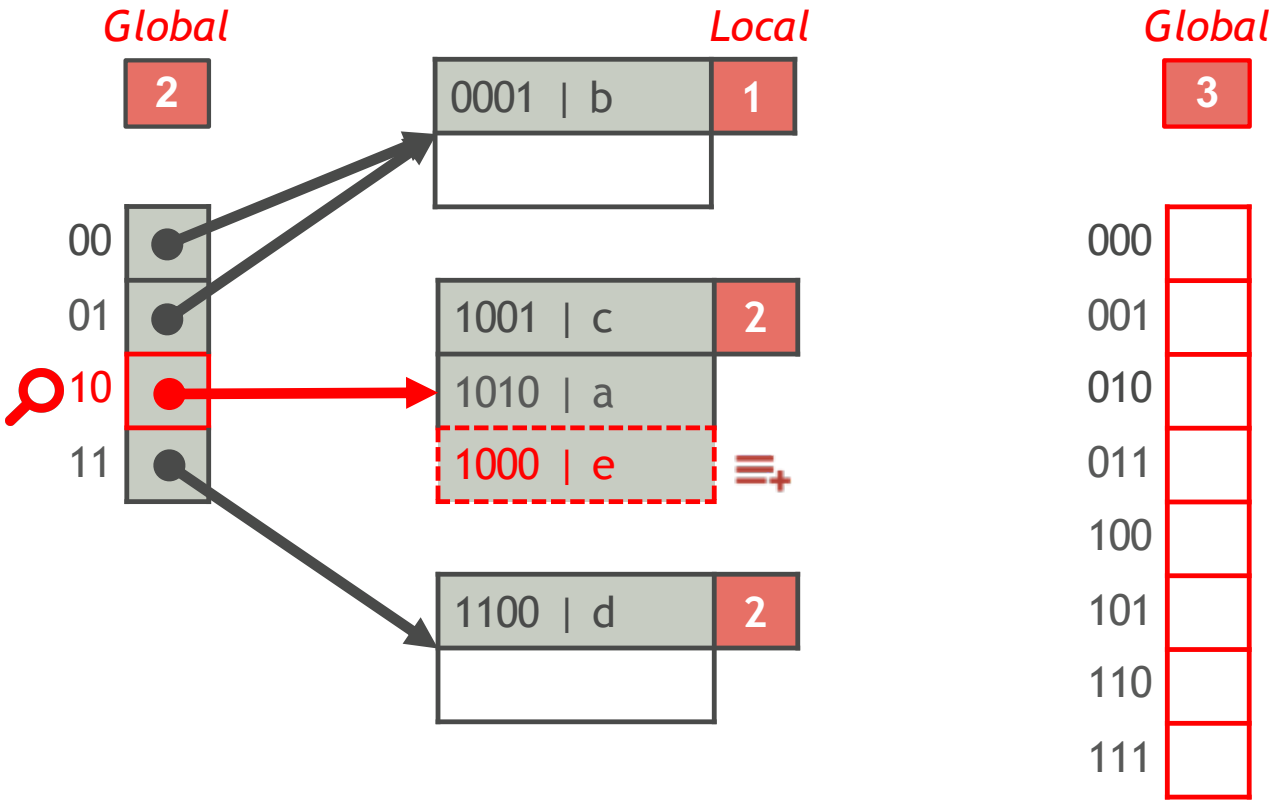


插入索引项(续)

如果P溢出且P的局部深度等于全局深度

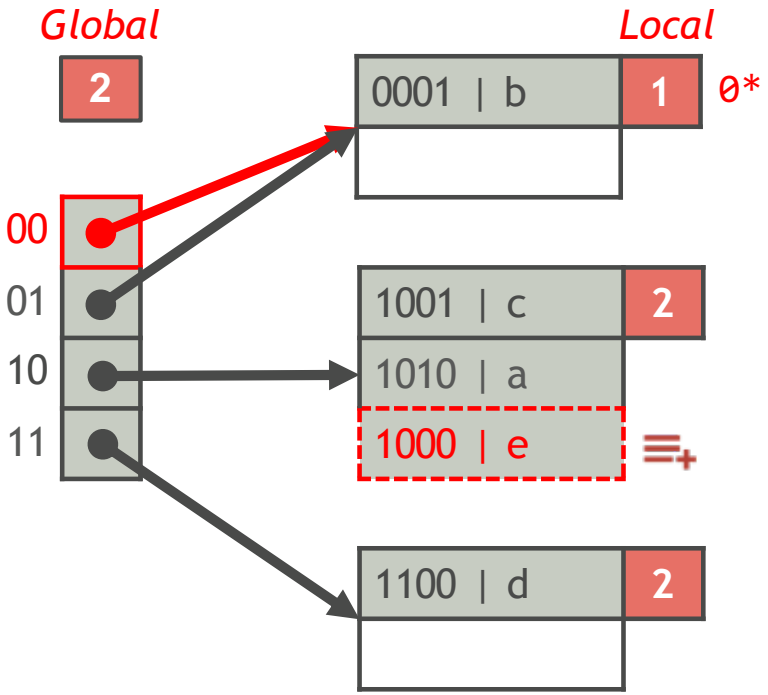
1. 将全局深度加1，即桶的数量翻倍
2. 更新每个桶中的页指针
3. 对于P，使用前面介绍的方法分裂P

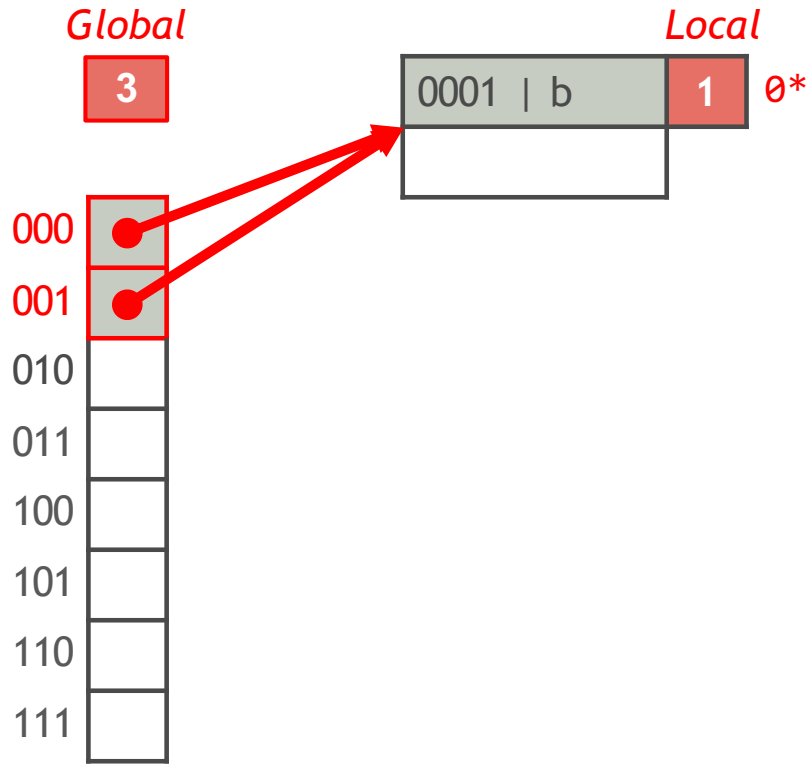
例: $K = e$, $hash(e) = 1000$



步骤1/5

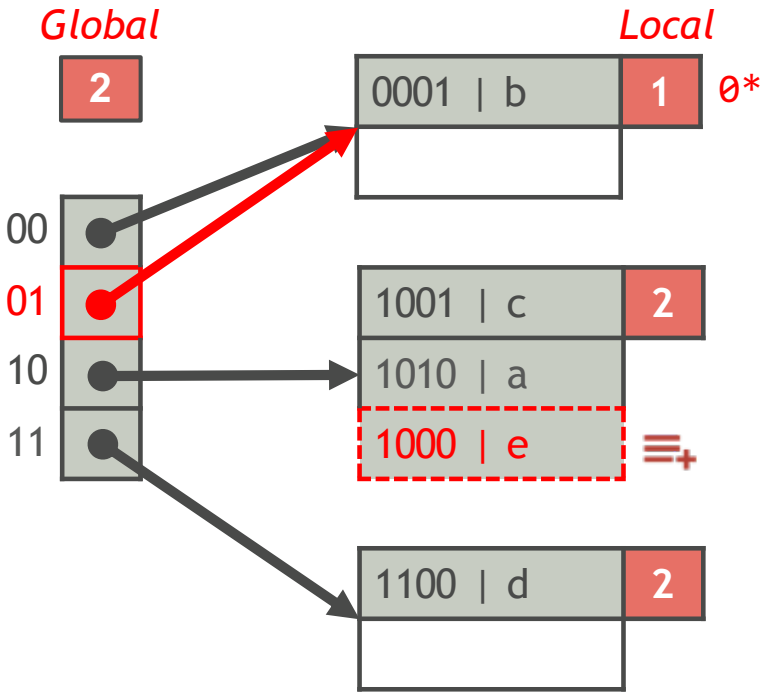
插入索引项(续)

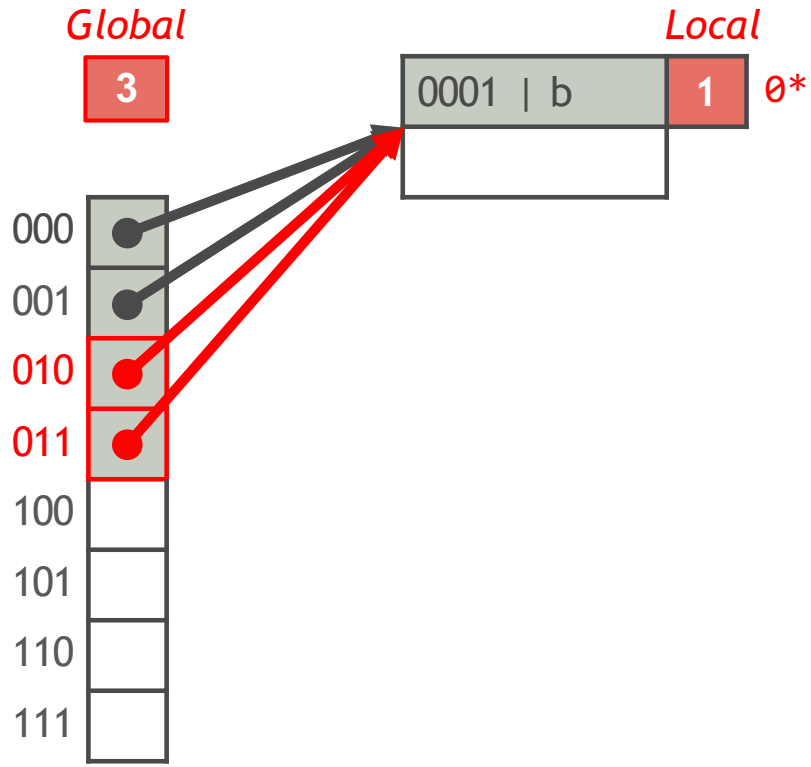

Global
Local
2
0001 | b
1
0*
00
01
10
11
1001 | c
2
1010 | a
1000 | e
1100 | d
2


Global
Local
3
0001 | b
1
0*
000
001
010
011
100
101
110
111

步骤2/5

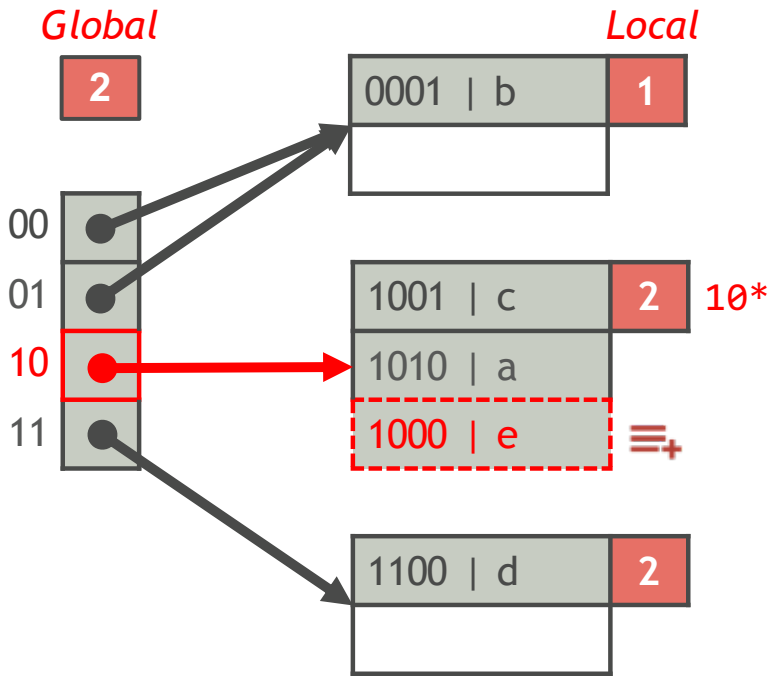
插入索引项(续)

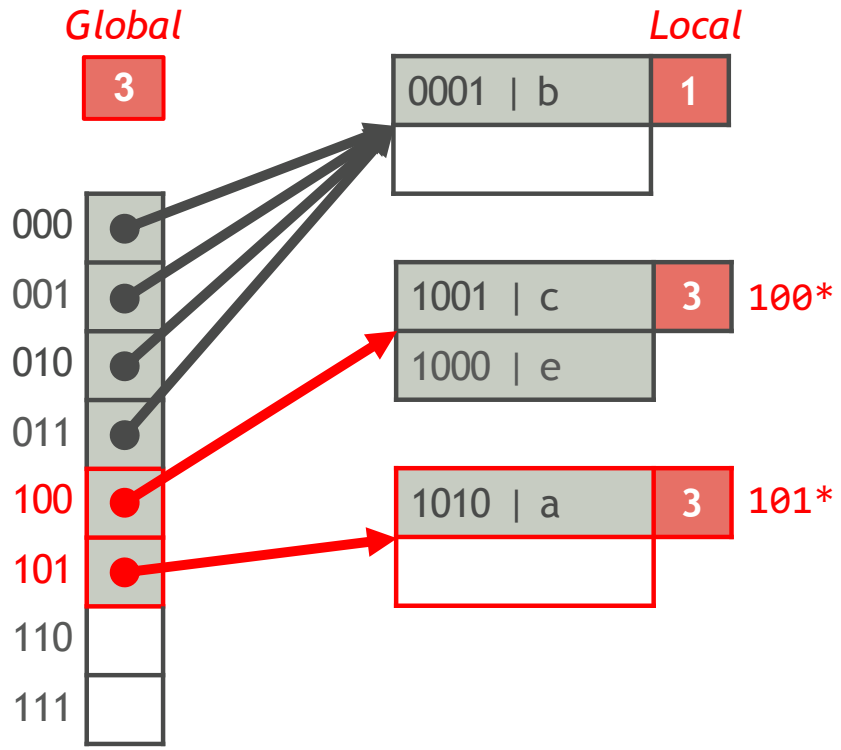

Global
Local
2
0001 | b
1
0*
00
01
10
11
1001 | c
2
1010 | a
1000 | e
1100 | d
2


Global
Local
3
0001 | b
1
0*
000
001
010
011
100
101
110
111

步骤3/5

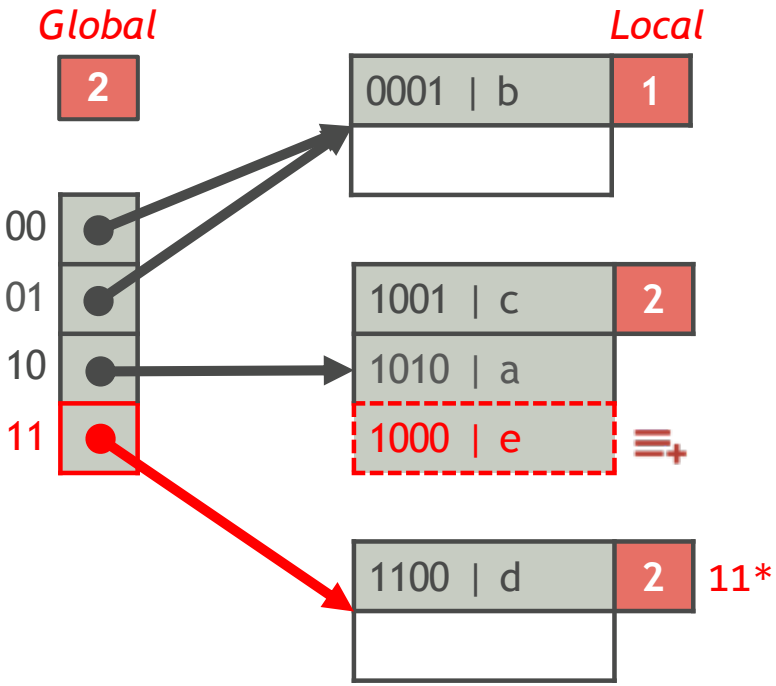
插入索引项(续)

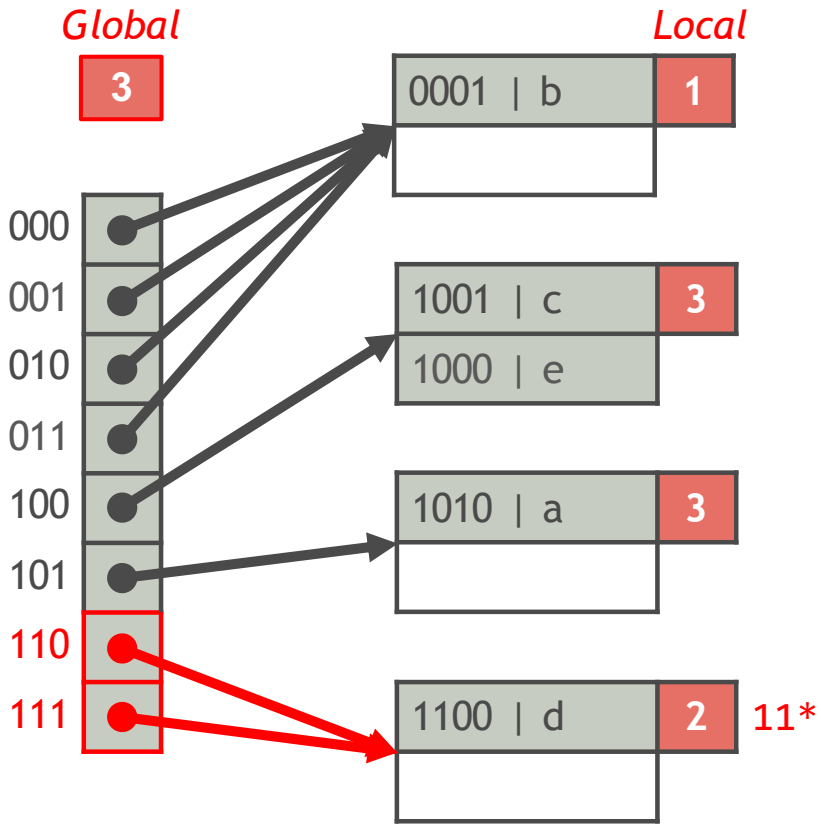

Global
Local
2
0001 | b
1
00
01
10
11
1001 | c
2
10*
1010 | a
1000 | e
1100 | d
2


Global
Local
3
0001 | b
1
000
001
010
011
100
101
110
111
1001 | c
3
100*
1000 | e
1010 | a
3
101*

步骤4/5

插入索引项(续)


Global
2
Local
0001 | b
1
00
01
10
11
1001 | c
2
1010 | a
1000 | e
1100 | d
2
11*

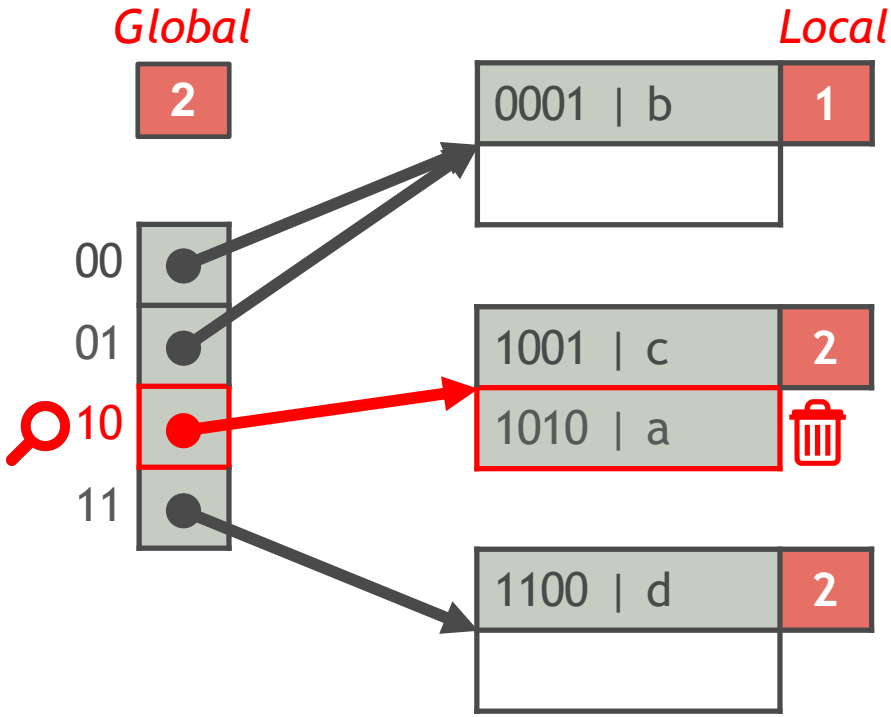

Global
3
Local
0001 | b
1
000
001
010
011
100
101
110
111
1001 | c
3
1000 | e
1010 | a
3
1100 | d
2
11*

步骤5/5

删除索引项

1. 找到索引项所在的页
2. 从页中删除索引项

例: $K = a$, $hash(a) = 1010$



思考题

删除索引项后，是否需要合并页?

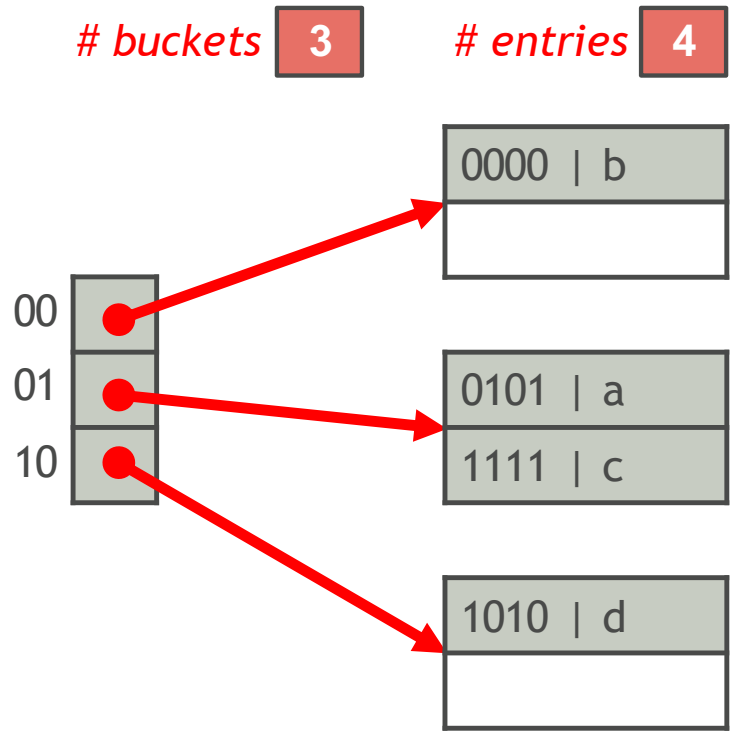
Hash-based Index Structures
Linear Hash Tables

线性哈希表(Linear Hash Tables)

线性哈希表包含n个桶

- 每个桶中保存一个指针，指向存储该桶中索引项的页的链表
- 假设每个页最多存储b个索引项，每个桶只有1个页，则线性哈希表中最多存储θbn个索引项，其中$0 < \theta < 1$是一个阈值
- 记录线性哈希表中桶的数量(# buckets)和索引项的数量(# entries)

例: $b = 2, \theta = 0.85$



哈希方案(Hashing Scheme)

- 设桶号为$0, 1, \ldots, n-1$
- 令$m = 2^{\lfloor \log_2 n \rfloor}$，因此$m \leq n < 2m$
- 对于键值为K的索引项，如果$hash(K) \mod 2m < n$，则该索引项属于编号为$hash(K) \mod 2m$的桶；否则，该索引项属于编号为$hash(K) \mod m$的桶

Example (线性哈希表的哈希方案)

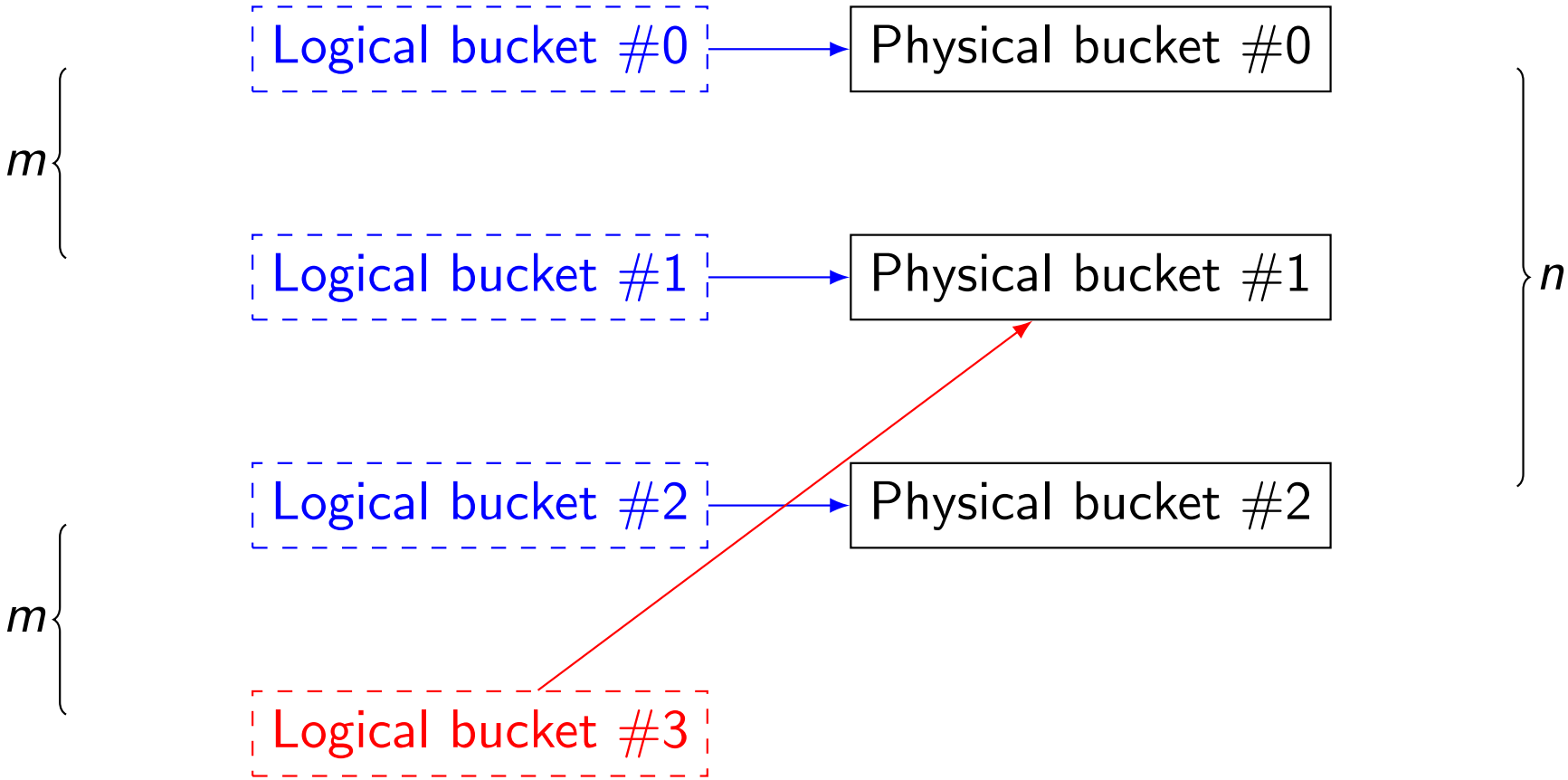
设$n = 3$，有$m = 2$

Bucket #0	$hash(K) = 0, 4, 8, \ldots$
Bucket #1	$hash(K) = 1, 3, 5, 7, 9, \ldots$
Bucket #2	$hash(K) = 2, 6, 10, \ldots$

桶的负载不平衡

哈希方案的解释

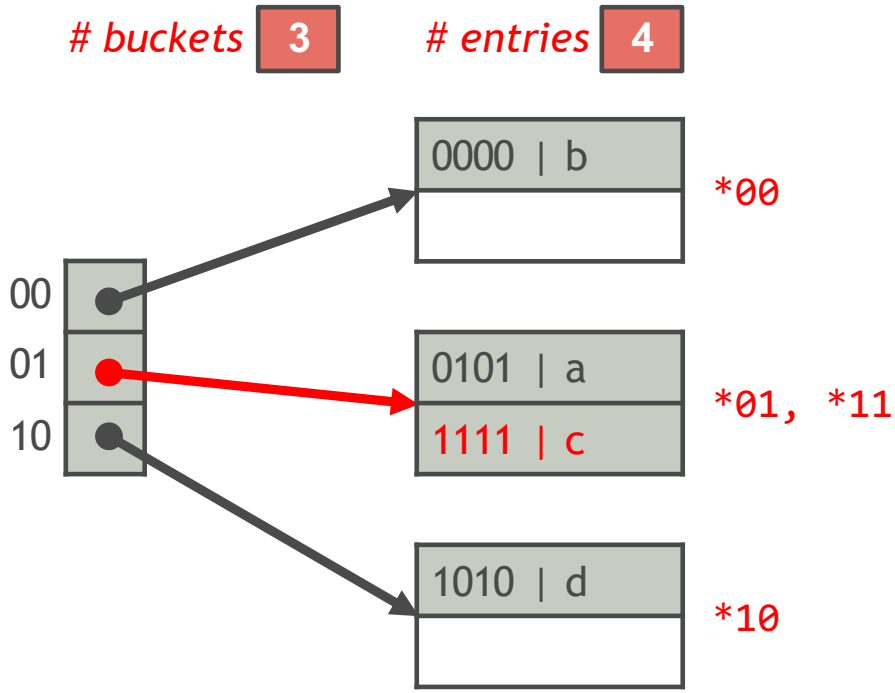
- 逻辑上有$2m$个桶，物理上有n个桶，$n < 2m$
- 键值为K的索引项的逻辑桶号$b(K) = hash(K) \mod 2m$
- 如果$b(K) < n$，则该索引项属于$b(K)$号物理桶
- 如果$b(K) \geq n$，$b(K)$号物理桶不存在，则该索引项被放入$b(K) \mod m$号物理桶



查找索引项

1. 确定索引项所属的桶
2. 在桶指向的页链表中查找索引项

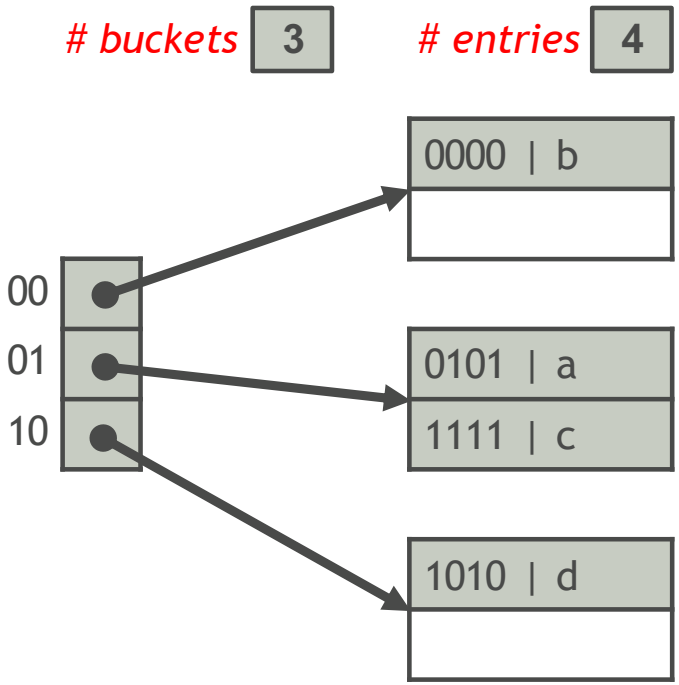
例: $K = c, hash(c) = 1111$。根据哈希方案，键为c的索引项在01号桶中

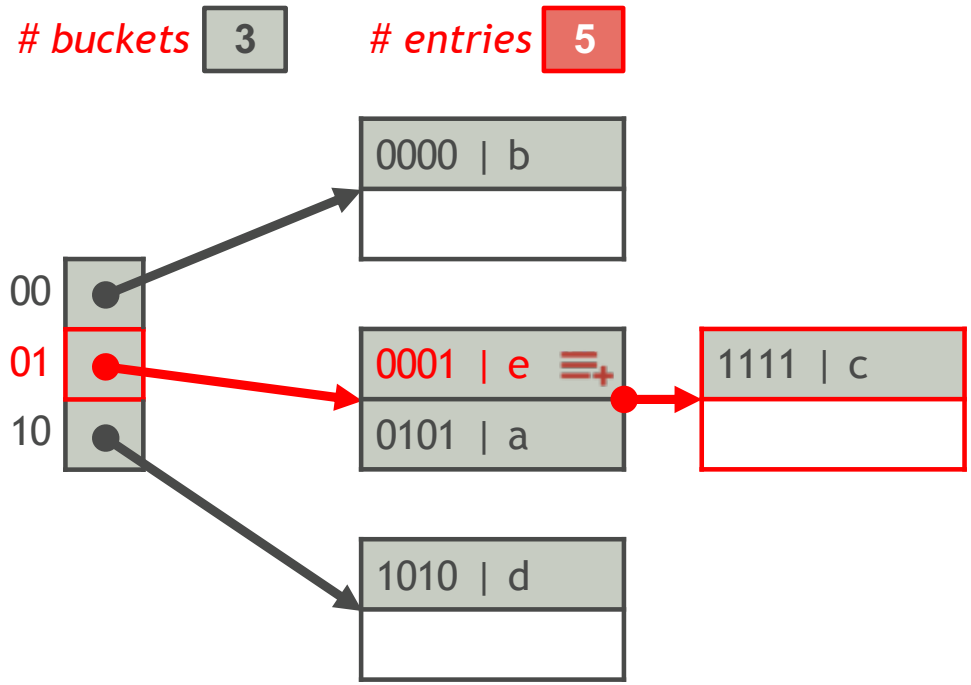


插入索引项

1. 将索引项插入它所属的桶B
2. 将索引项的数量(# entries)加1
3. 如果# entries $\leq \theta bn$，则插入完成；否则，将桶的数量(# buckets)加1，按照哈希方案重新分配哈希表中的索引项

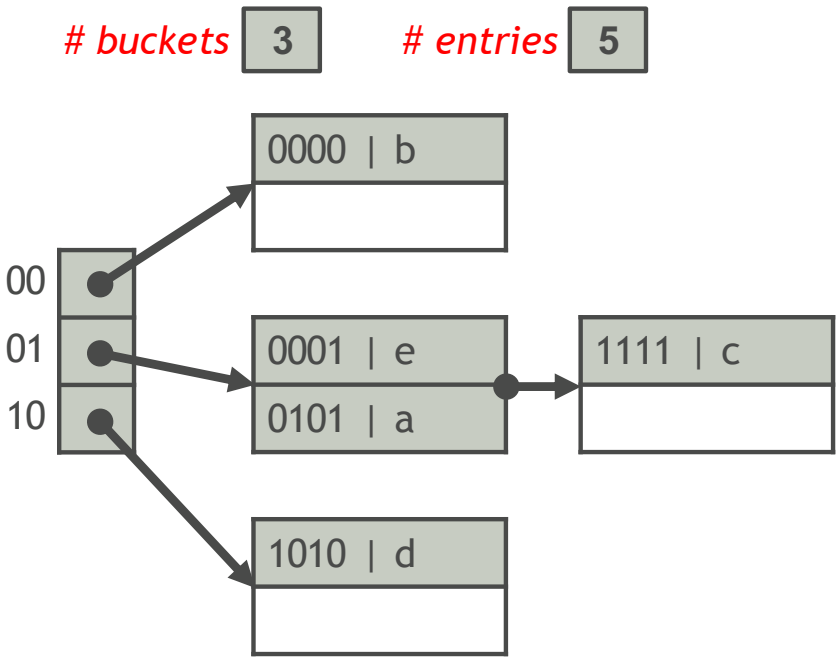
例1: $hash(e) = 0001, \theta = 0.85$

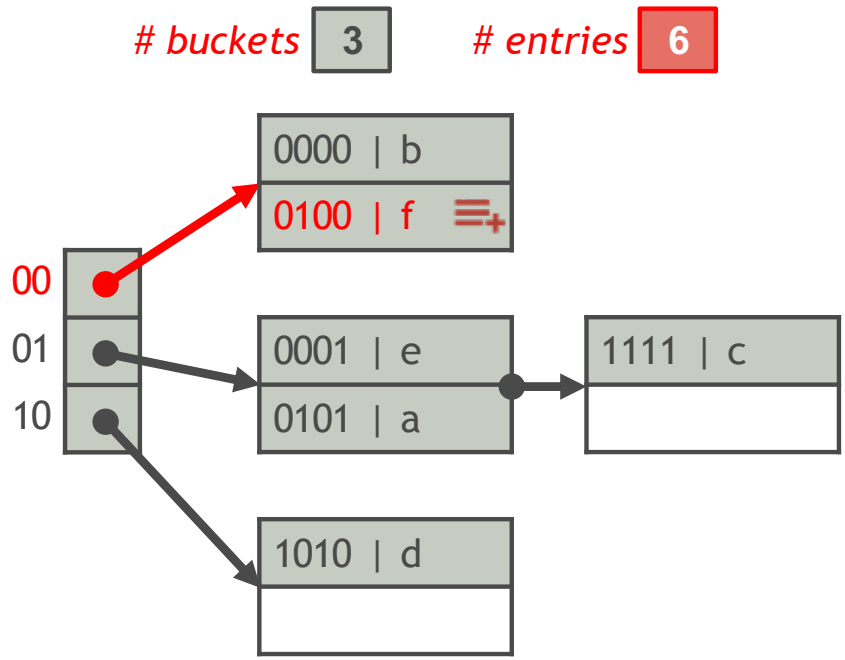




插入索引项(续)

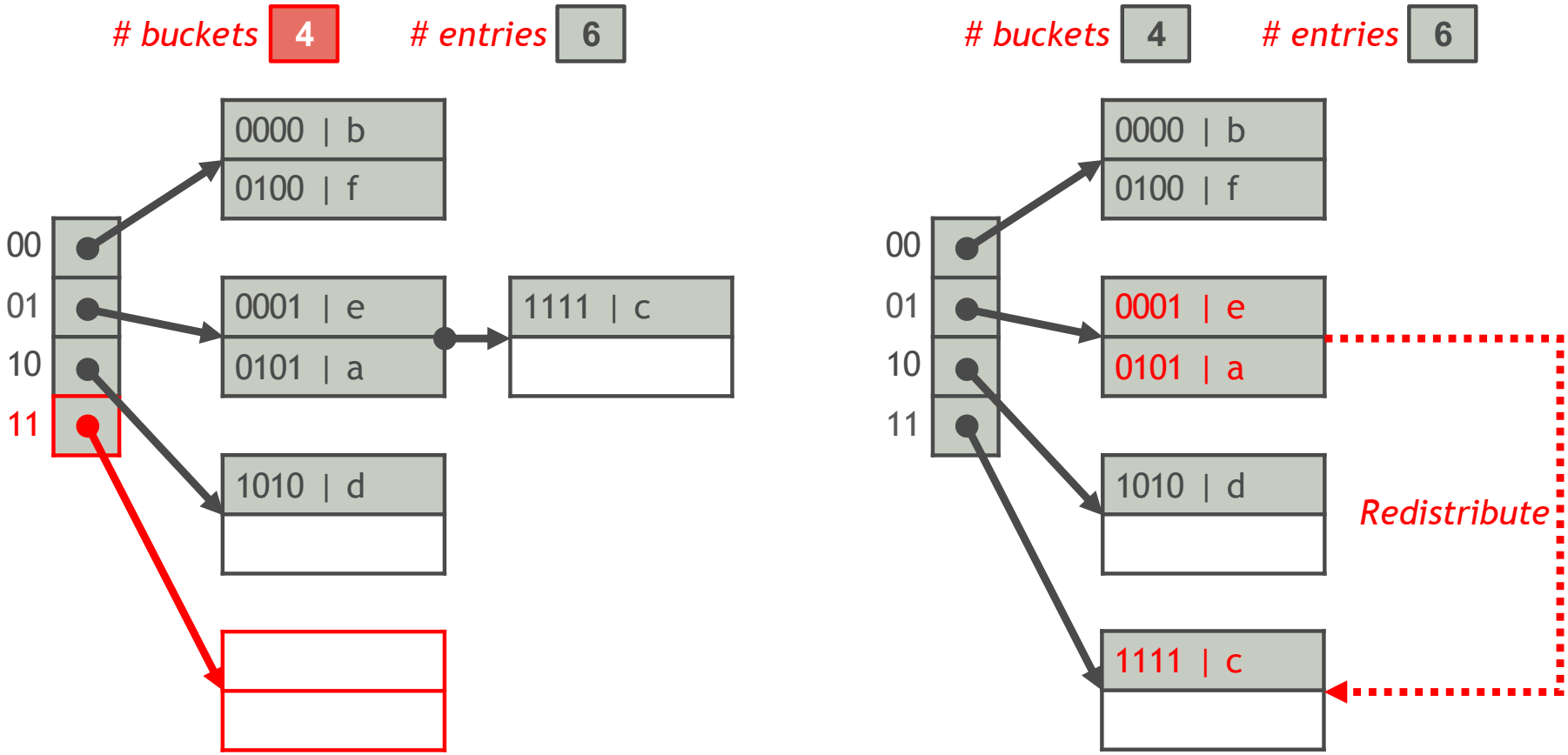
例2: $hash(f) = 0100, \theta = 0.85$





步骤1/2

插入索引项(续)

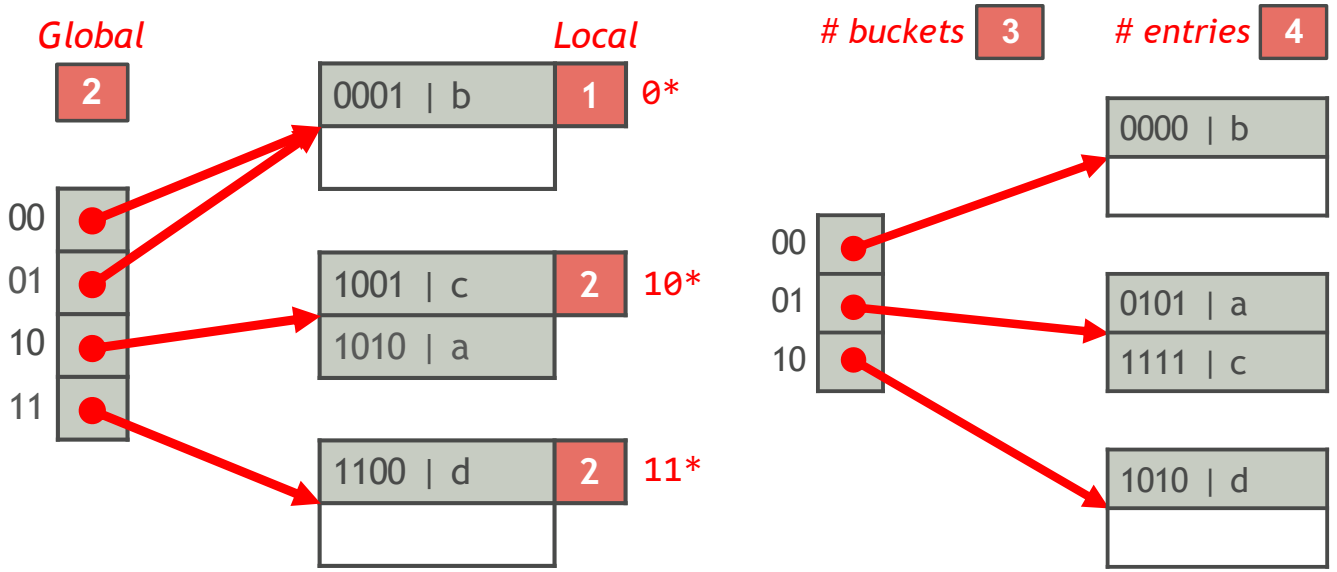


步骤2/2

因为新创建的桶的编号为11，所以只需将原来01号桶中应属于11号桶的索引项重新分配到11号桶中；其他桶中的索引项无需重分配

可扩展哈希表VS 线性哈希表

	可扩展哈希表	线性哈希表
桶的数量	2^{global_depth}	n
是否有溢出页	无	有
哈希方案	$hash(K)$的前$global_depth$位	$hash(K) \mod 2m$ 或$hash(K) \mod m$
页分裂条件	页发生溢出	$\#entries > \theta bn$
增加桶的方法	桶数翻倍($global_depth$加1)	桶数加1



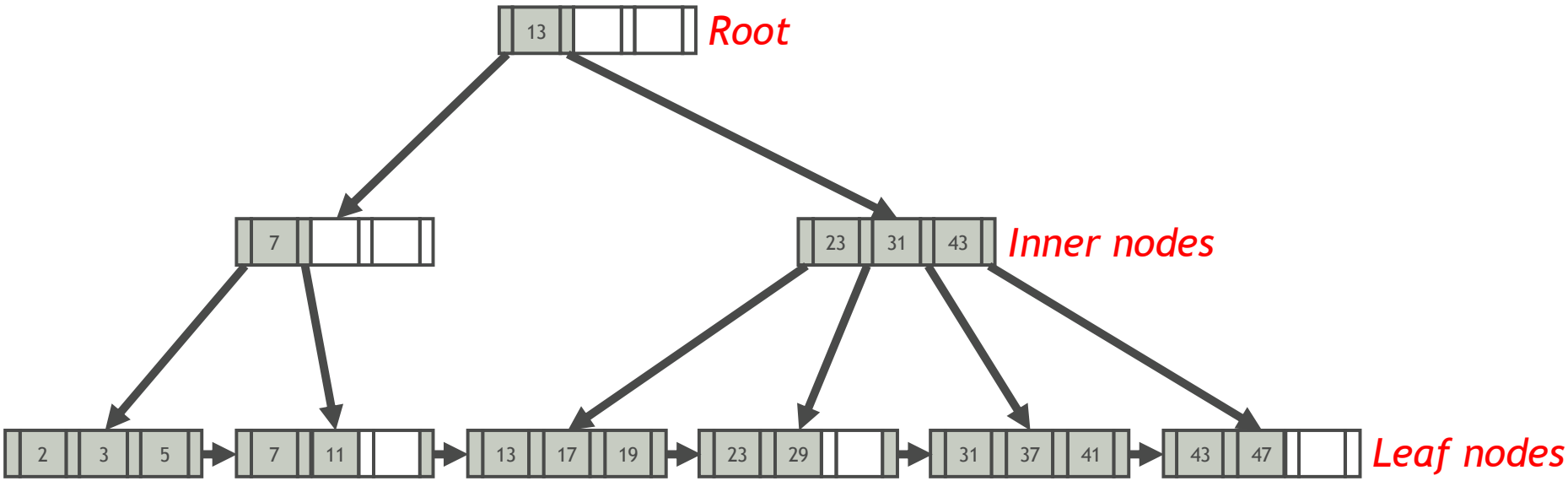
Tree-based Index Structures

Tree-based Index Structures
B+ Trees

B+树(B+ Trees)

B+树是一棵M路平衡搜索树，它具有以下性质：

- B+树是一棵完美的平衡树，所有叶节点都在同一层上
- 除根节点外，每个节点至少“半满”，即$M/2-1 \leq \#keys \leq M-1$[3]
- 每个节点恰好放入1个页

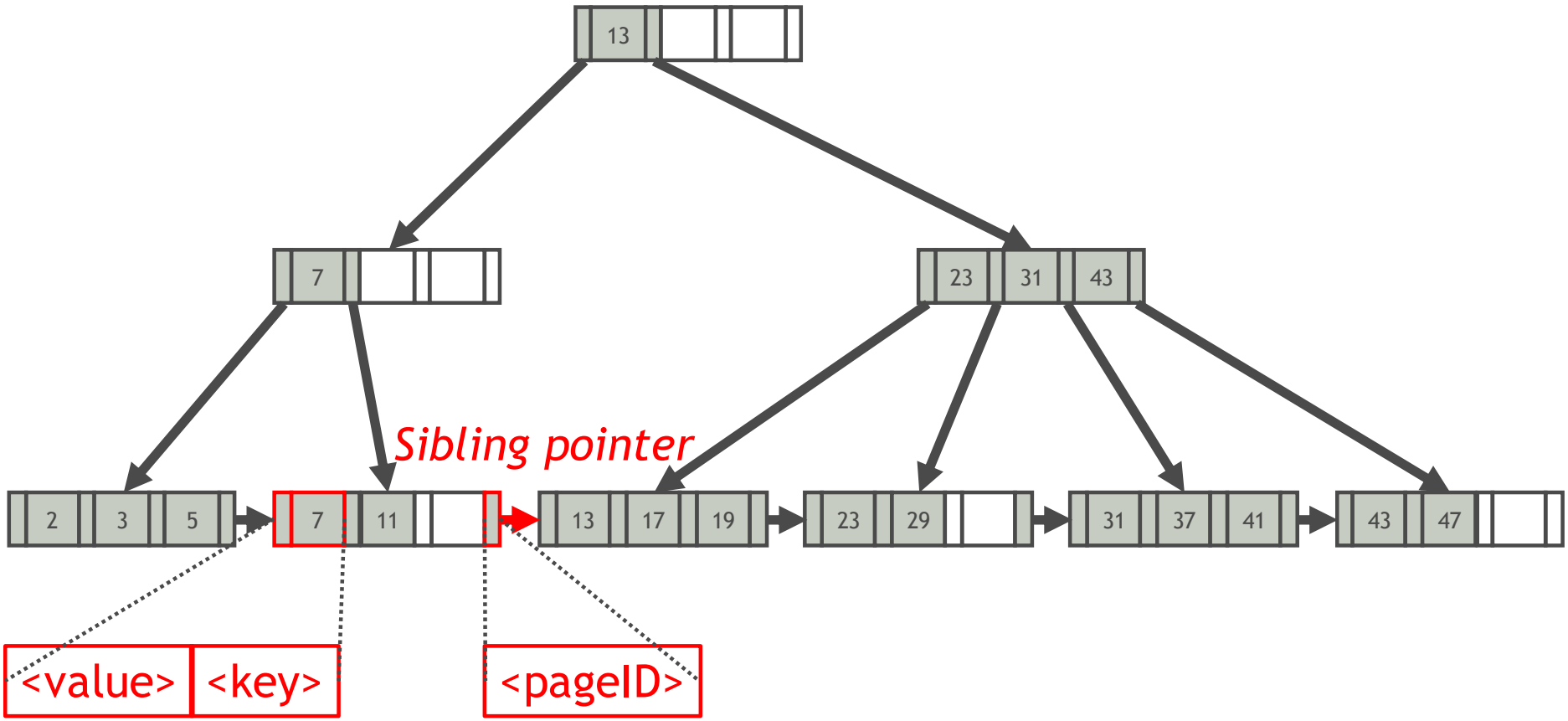


[3]Raghu Ramakrishnan, Johannes Gehrke. Database Management Systems, 3rd Edition. 2003

B+树的叶节点(Leaf Nodes)

每个叶节点包含一个索引项数组和一个指向右侧兄弟叶节点的指针(右兄弟节点的页号)

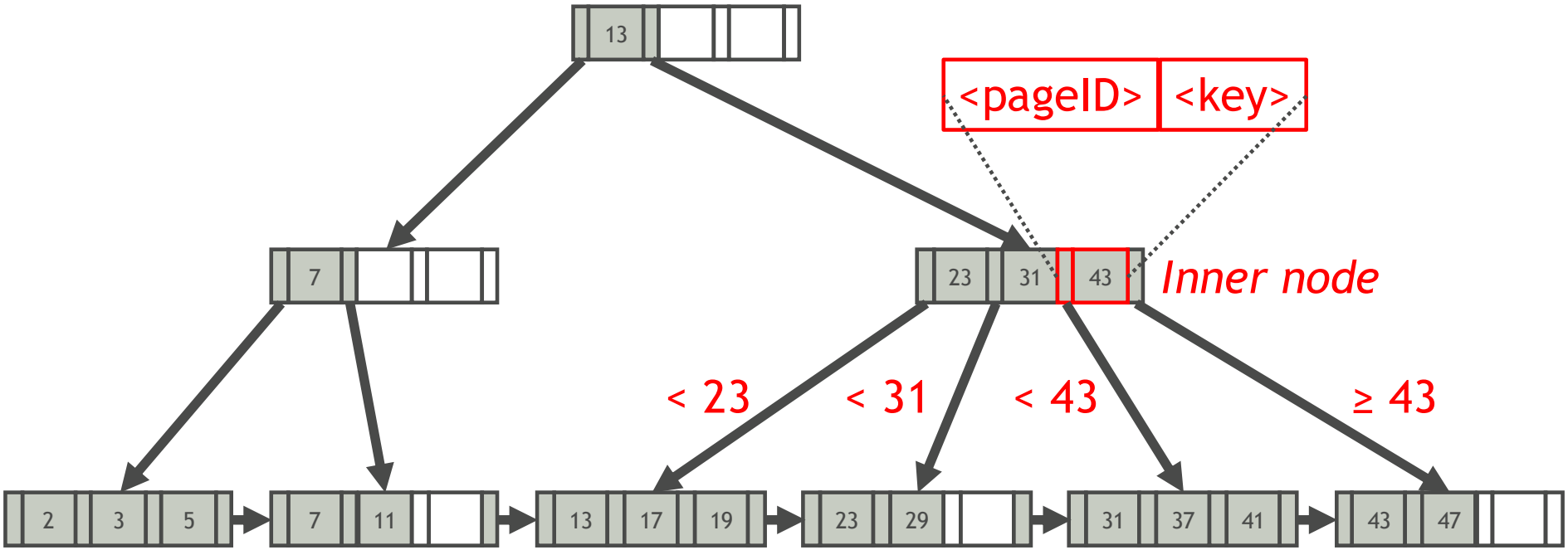
- 索引项数组通常按索引键排序



B+树内节点(Inner Nodes)

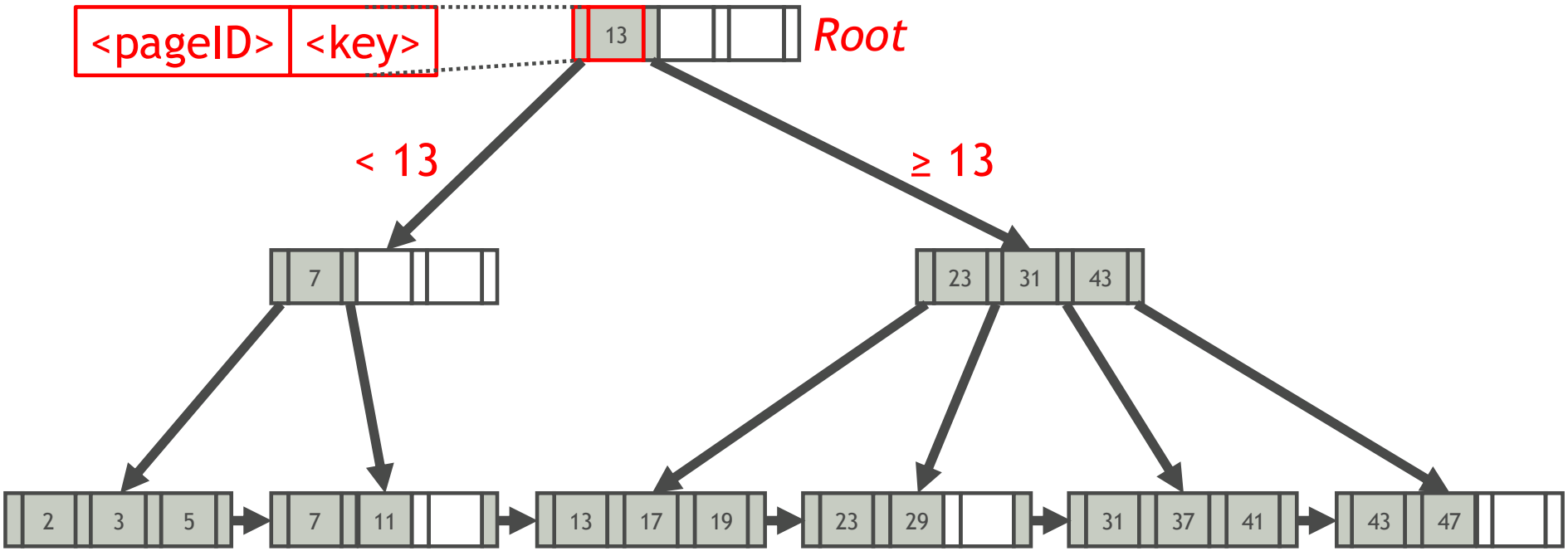
每个内节点包含一个键数组*Key*和一个指向儿子节点的指针的数组*Ptr*

- *Key*中有k的非空键值当且仅当*Ptr*中有$k+1$个非空指针
- *Key*中的键值排序
- *Ptr*[0]指向的子树中的键值$<$ *Key*[0]
- *Ptr*$[k+1]$指向的子树中的键值$\geq$ *Key*$[k]$
- *Key*$[i-1] \leq$ *Ptr*$[i]$指向的子树中的键值$<$ *Key*$[i]$



B+树的根节点(Root Node)

根节点和内节点的内部结构相同，但不要求“半满”(根节点中包含至少1个键即可)

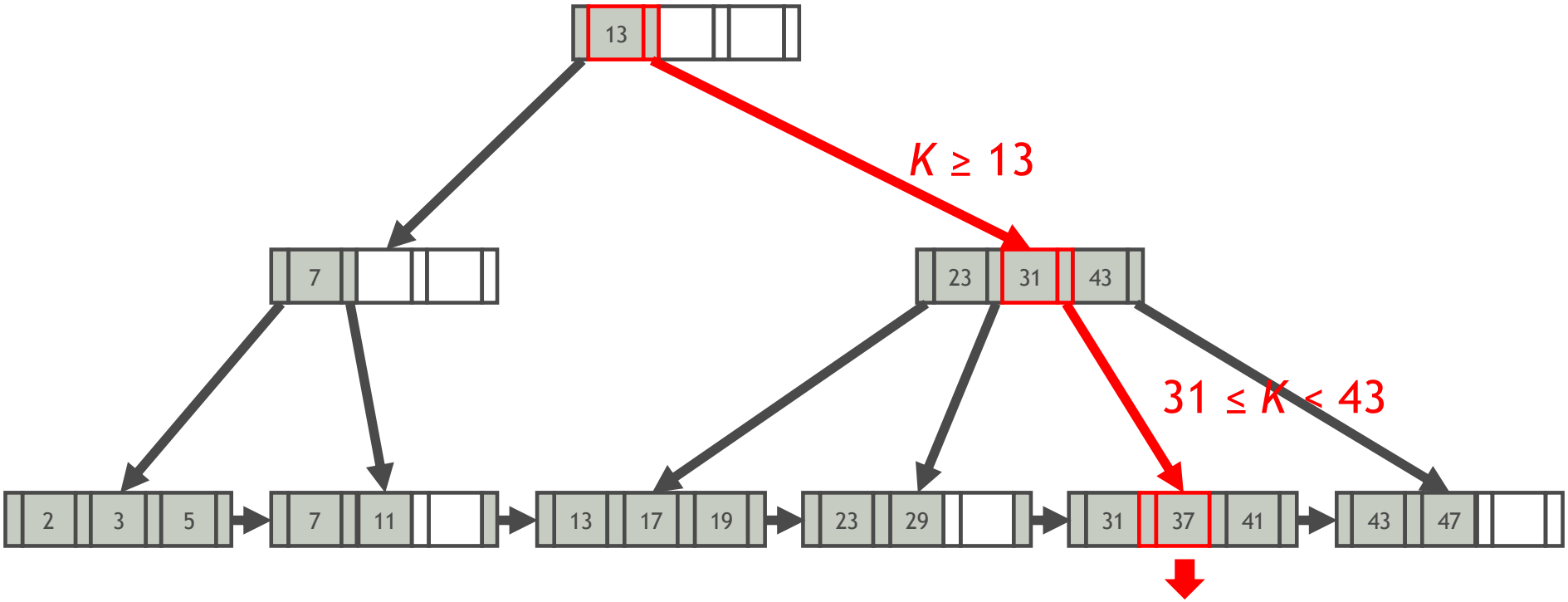


查找索引项

查找键为K

1. 在内节点的引导下，找到K属于哪个叶节点
2. 在该叶节点中查找键值为K的索引项

例: $K = 37$

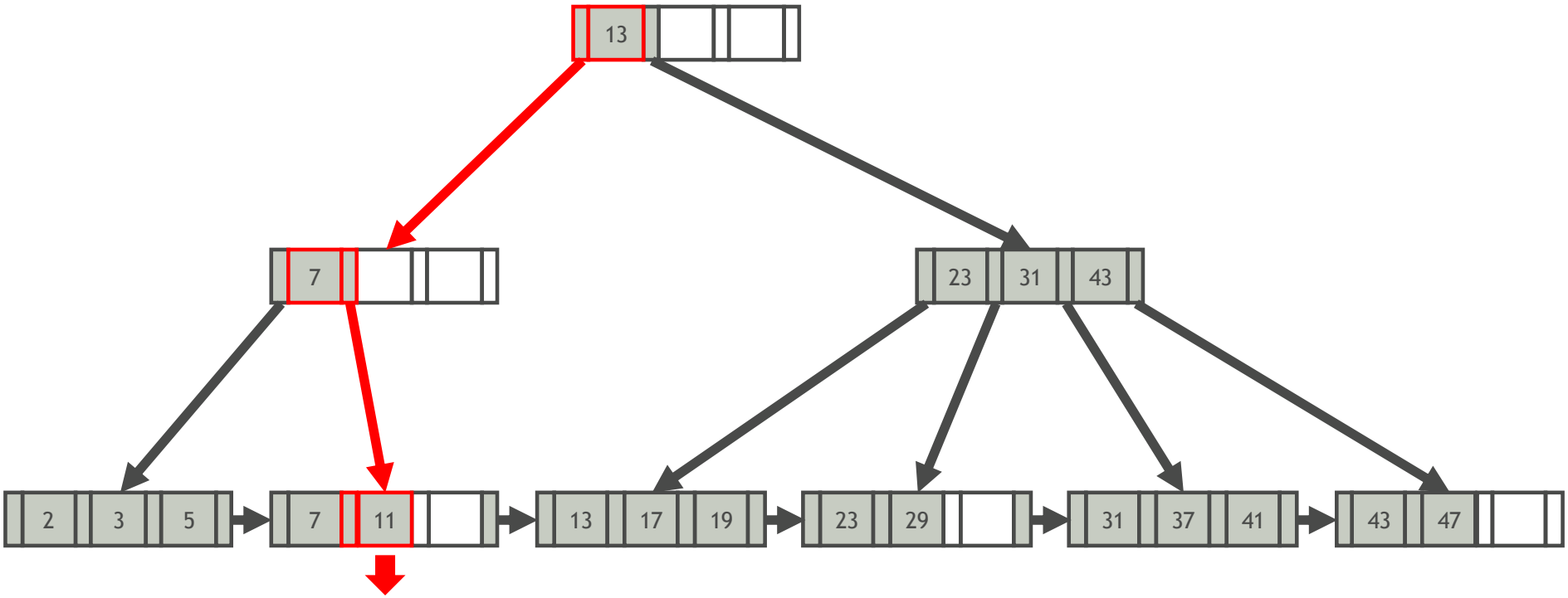


区间查询

查找键在区间$[L, U]$内的全部索引项

1. 找到具有大于等于L的最小键的索引项E
2. 扫描E右侧的索引项，如果键$\leq U$，则输出；否则，终止

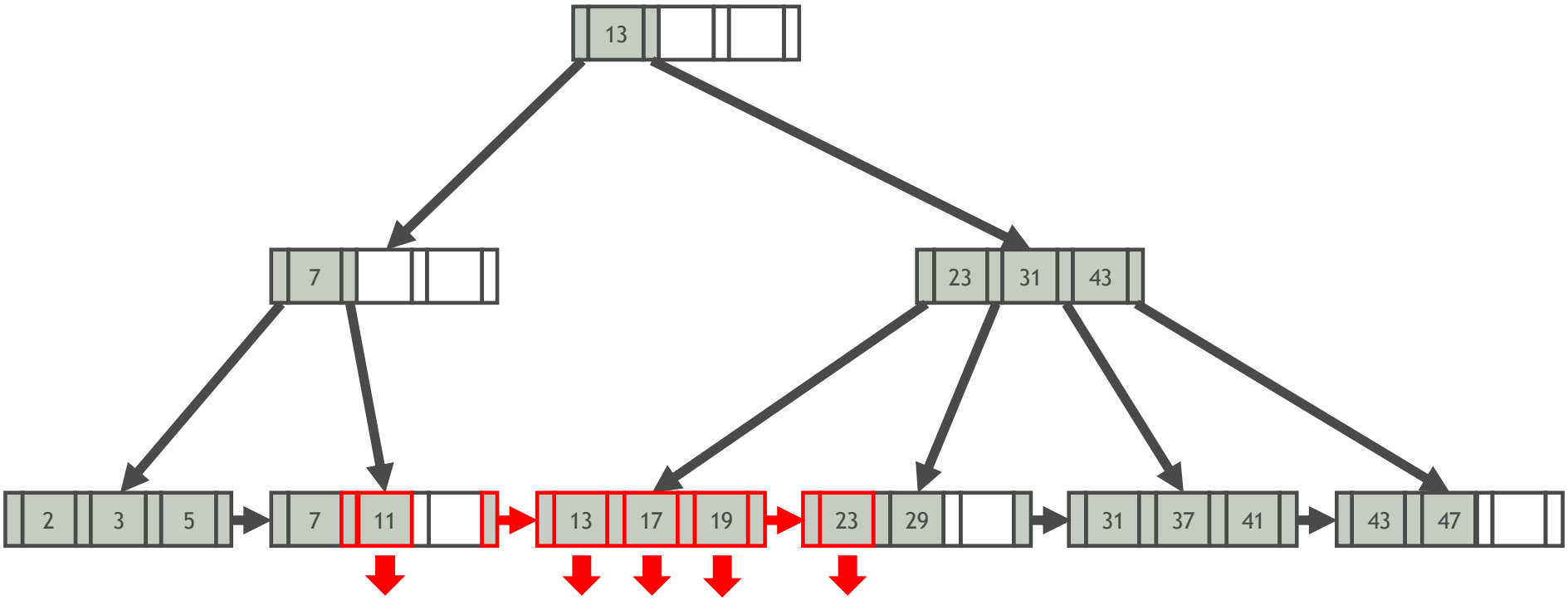
例: $K \in [10, 25]$



步骤1/2

区间查询(续)

例: $K \in [10, 25]$



步骤2/2

插入索引项

插入键为K的索引项

1. 找到K应在的叶节点L
2. 将索引项插入L
3. 如果L不溢出，则插入完成；否则，分裂(split)L

叶节点分裂

分裂叶节点L

1. 创建一个新的叶节点L_2
2. 将L中的索引项平分，前一半留在L中，后一半移入L_2中
3. 将L_2中最小的键存入“中间键(middle key)”变量
4. 在叶节点链表中，将L_2插到L的右边
5. 在L的父节点N中插入middle key及指向L_2的指针
6. 如果N不溢出，则完成对L的分裂；否则，继续分裂N

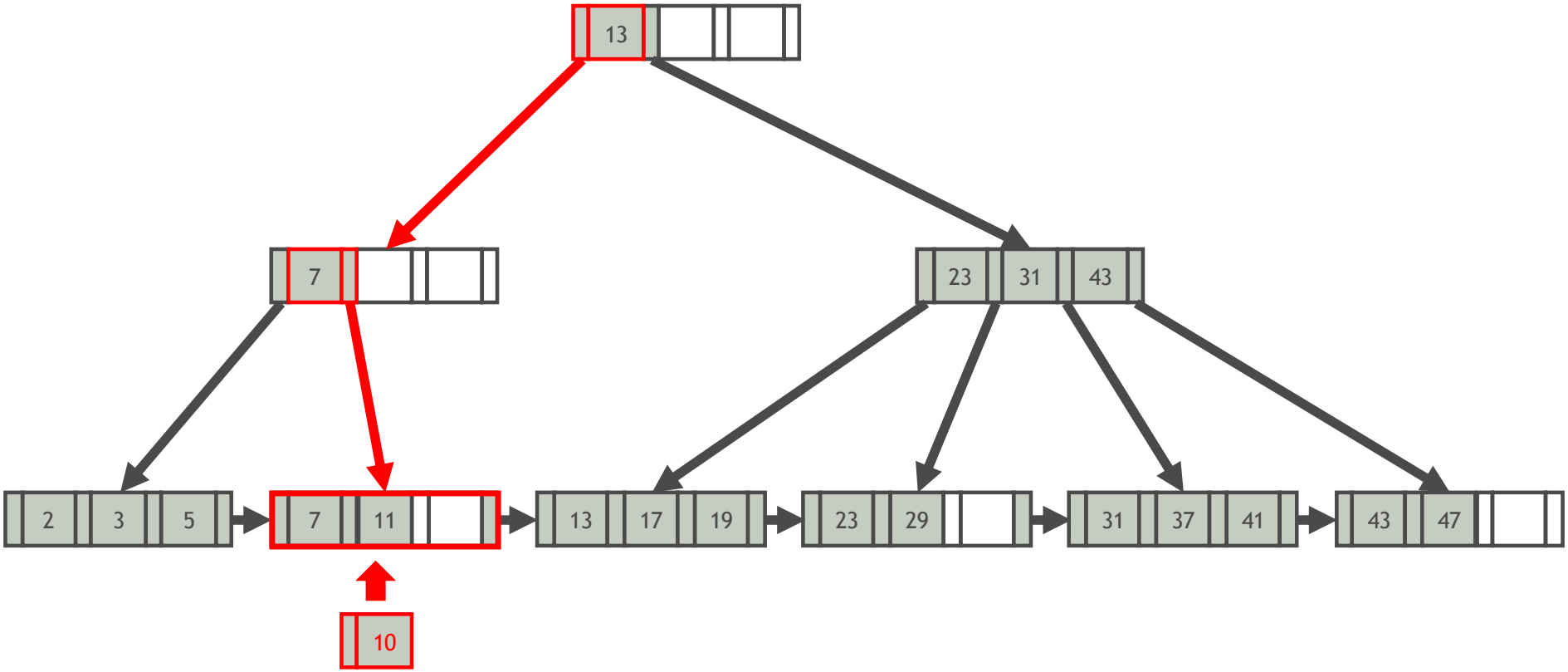
内节点分裂

分裂内节点N

1. 创建一个新的内节点N_2
2. 将N中的指针平分，前一半留在N中，后一半移入N_2中
3. 将N中多余的键存入“中间键(middle key)”变量
4. 如果N是根节点，则创建一个新的根节点N'，并在N'中插入一个指向N的指针
5. 在N的父节点N'中插入middle key及指向N_2的指针
6. 如果N'不溢出，则完成对N的分裂；否则，继续分裂N'

插入索引项(续)

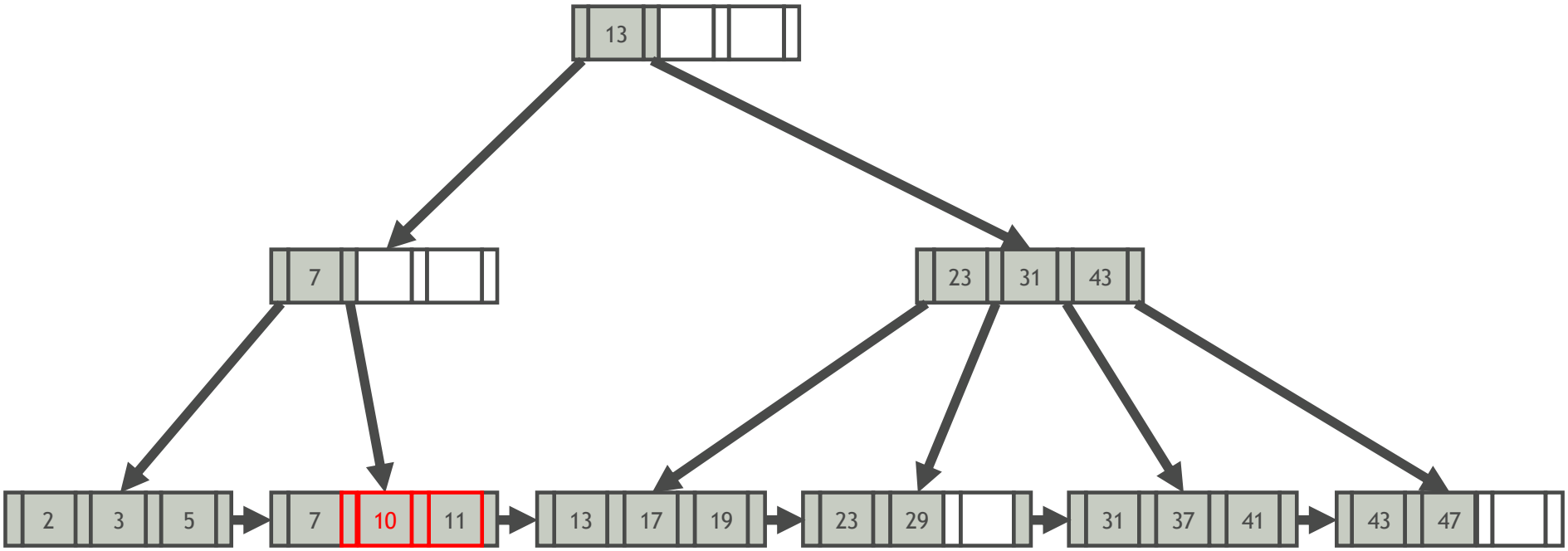
例1: $K = 10$ (无节点分裂)



步骤1/2

插入索引项(续)

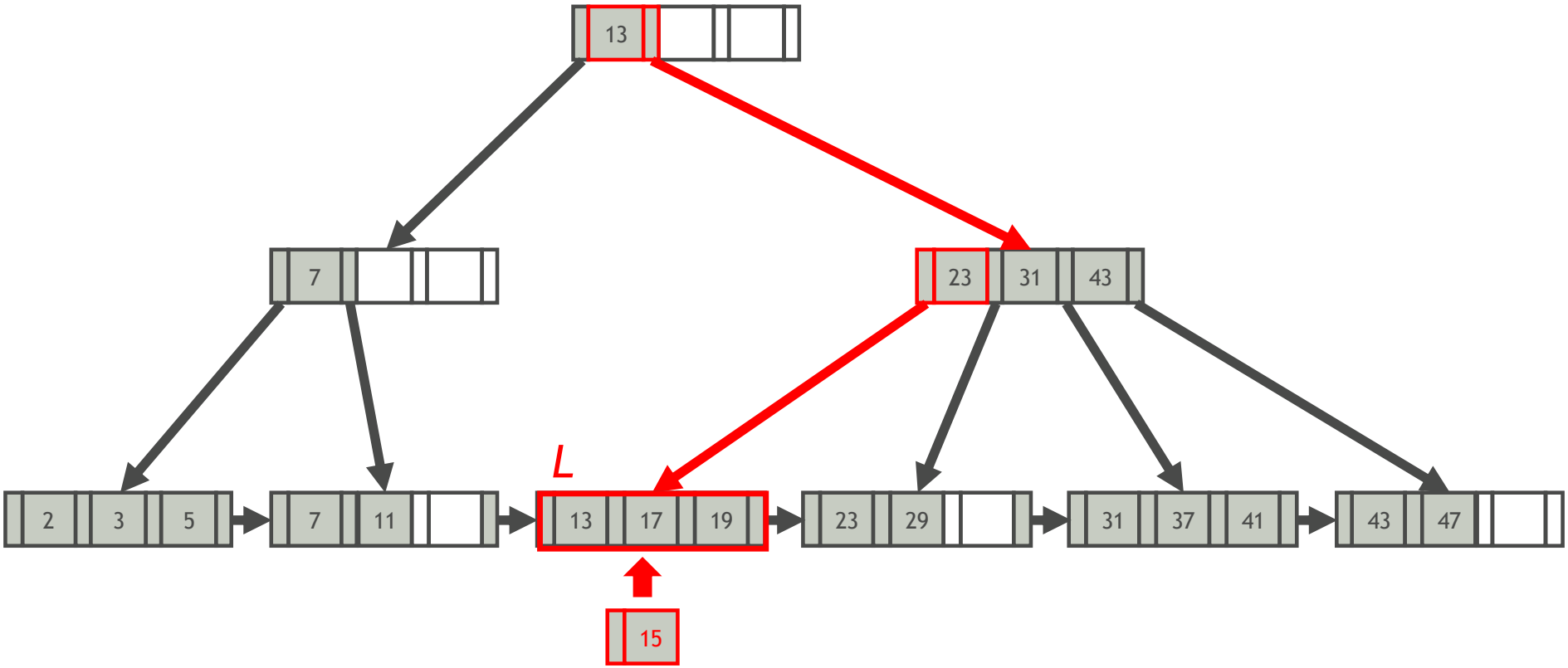
例1: $K = 10$ (无节点分裂)



步骤2/2

插入索引项(续)

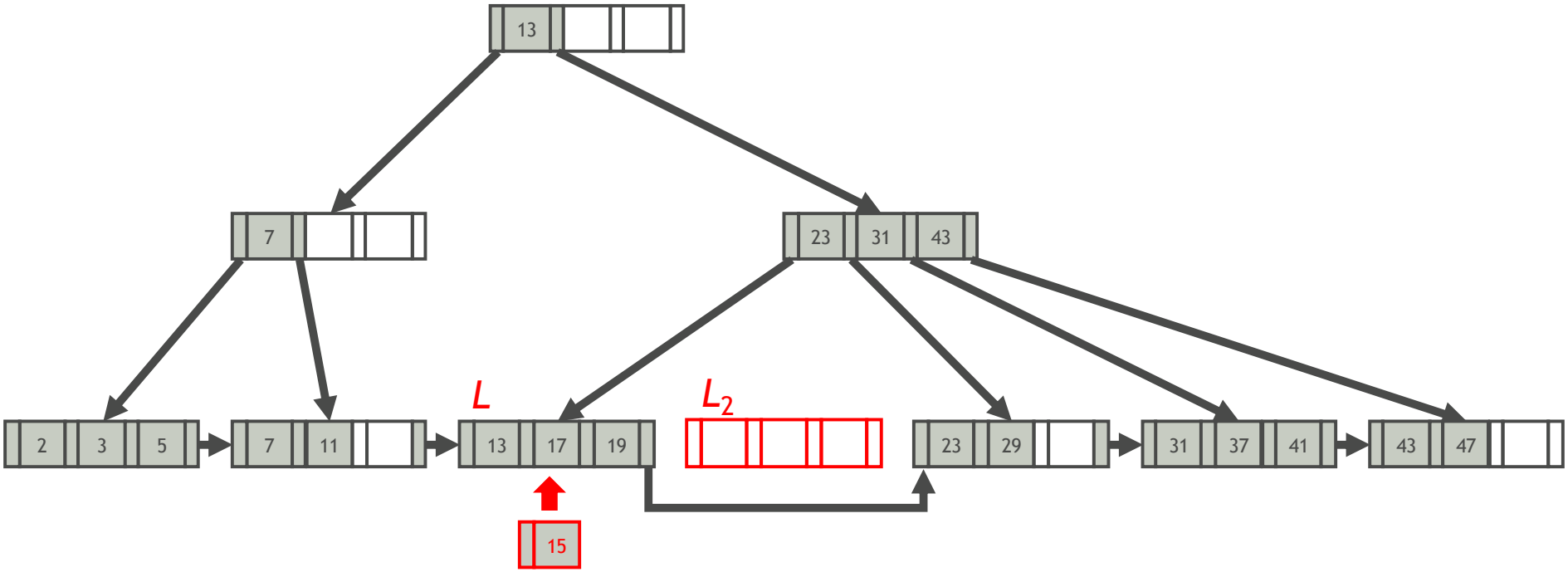
例2: $K = 15$ (有节点分裂)



步骤1/7

插入索引项(续)

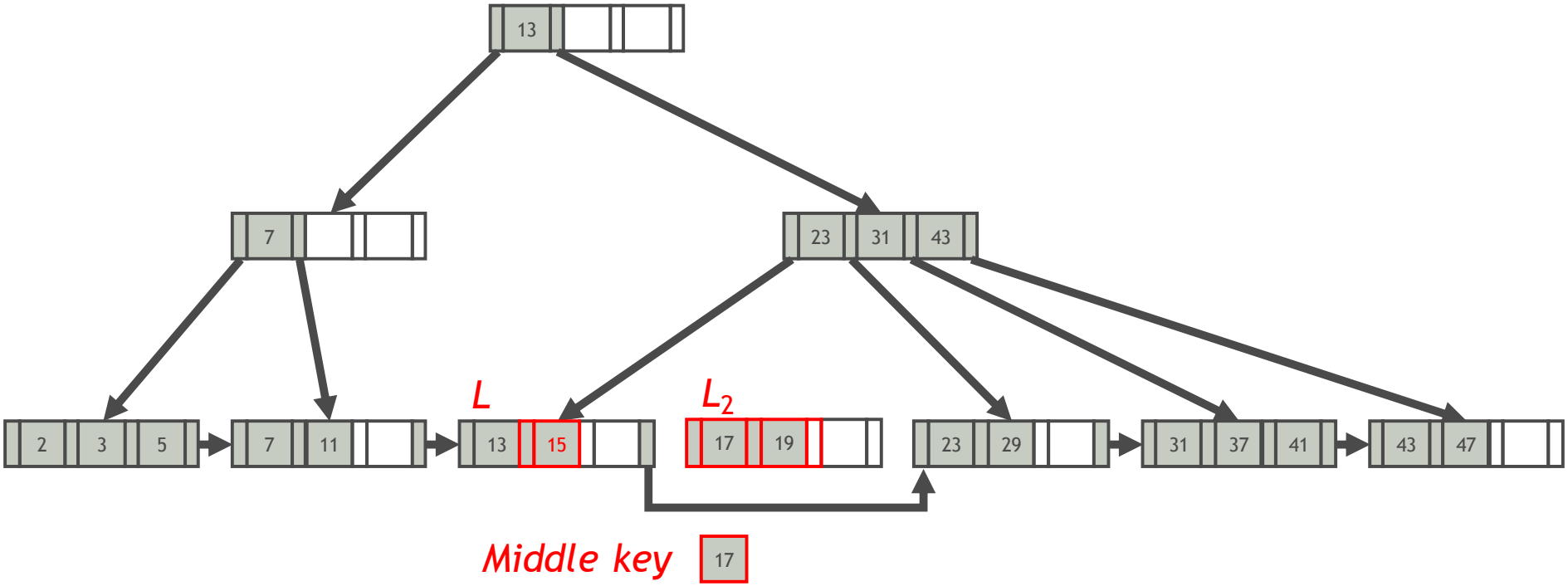
例2: $K = 15$ (有节点分裂)



步骤2/7

插入索引项(续)

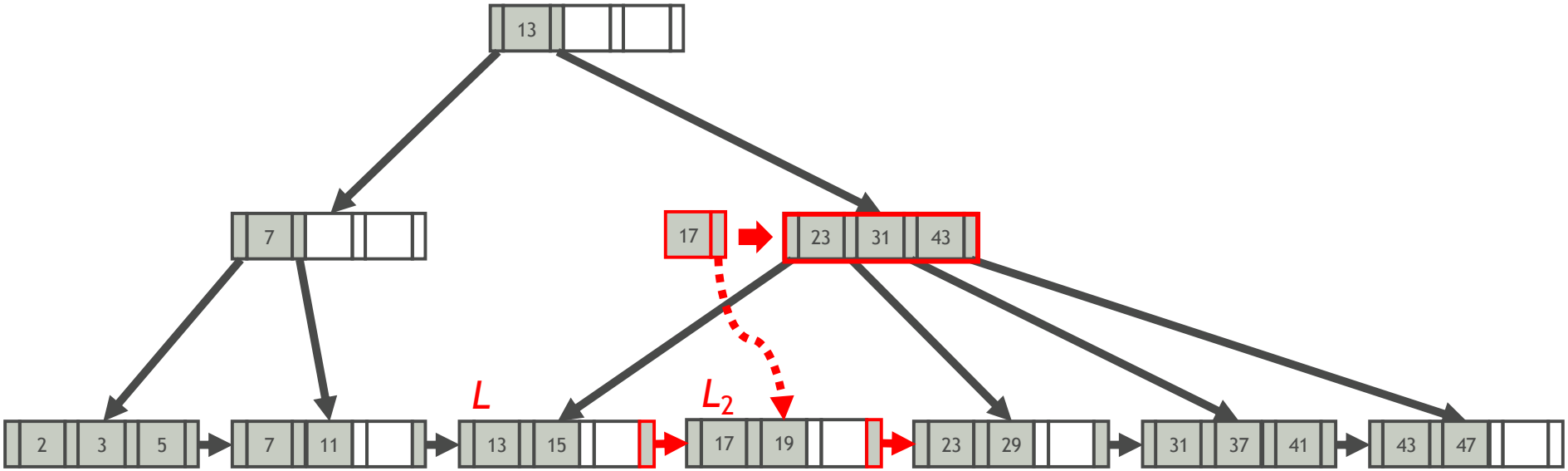
例2: $K = 15$ (有节点分裂)



步骤3/7

插入索引项(续)

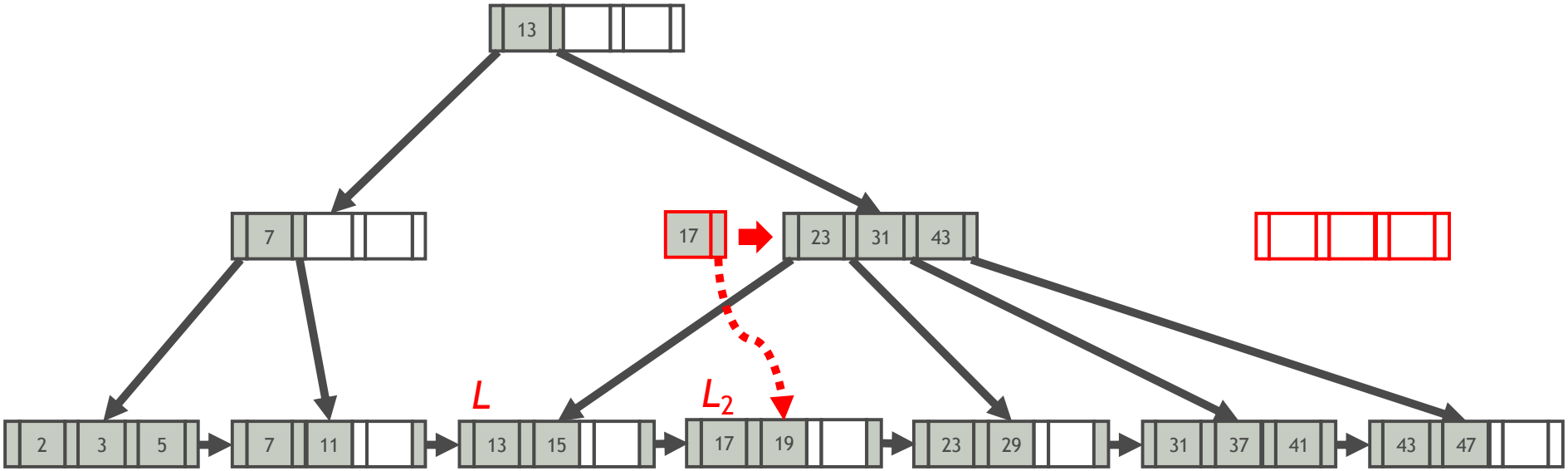
例2: $K = 15$ (有节点分裂)



步骤4/7

插入索引项(续)

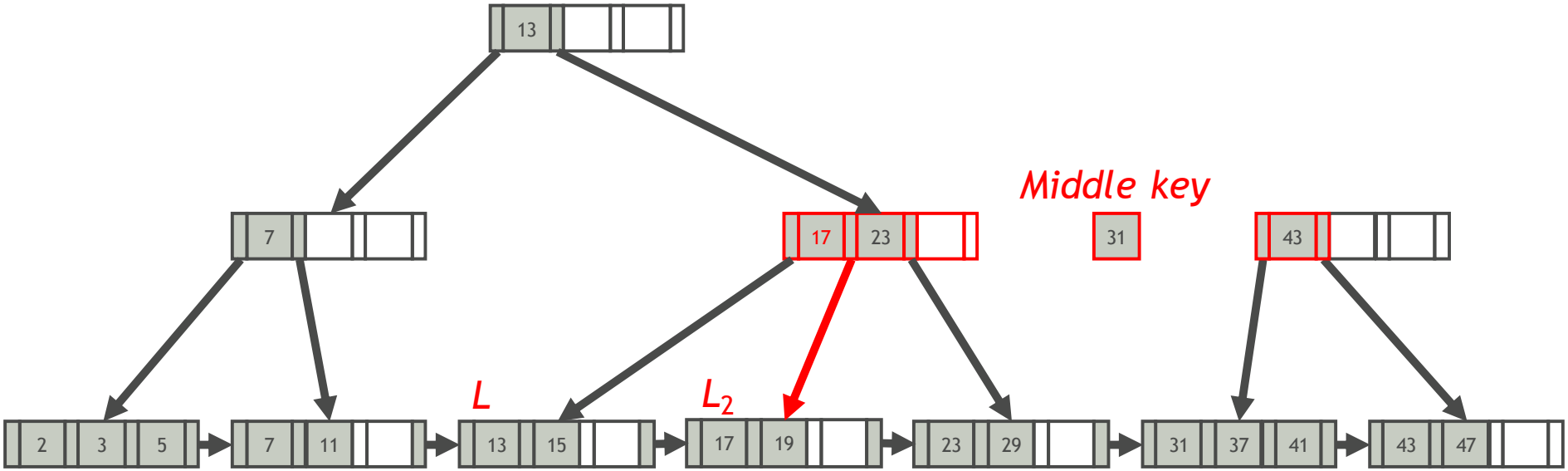
例2: $K = 15$ (有节点分裂)



步骤5/7

插入索引项(续)

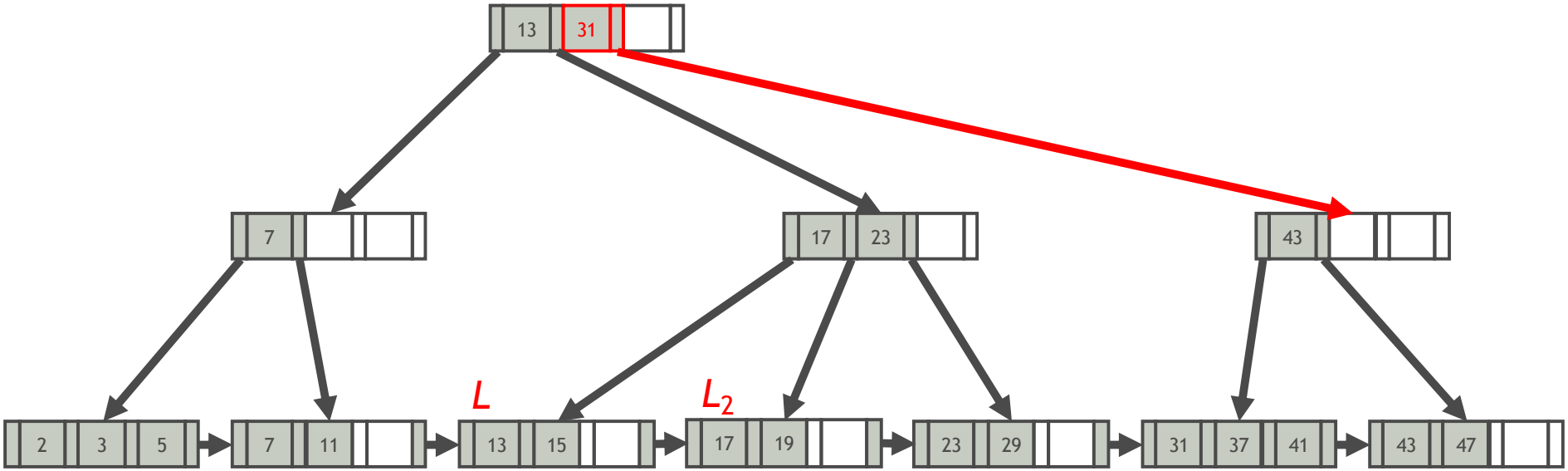
例2: $K = 15$ (有节点分裂)



步骤6/7

插入索引项(续)

例2: $K = 15$ (有节点分裂)



步骤7/7

删除索引项

删除键为K的索引项

1. 找到K所在的叶节点L
2. 从L中删除键为K索引项
3. 如果L至少半满，则完成删除；否则，处理L，使L至少半满

删除索引项(续)

使叶节点L至少半满的处理方法

1. 尝试从L相邻的兄弟节点借一个索引项，使两者均至少半满
2. 如果借不到，则将L与其兄弟节点合并(merge)

节点合并

1. 如果L与左侧兄弟节点L_1合并，则从L的父节点中删除指向L的指针及相应的键；
   如果L与右侧兄弟节点L_2合并，则从L的父节点中删除指向L_2的指针及相应的键
2. 如果L的父节点N至少半满，则完成合并；否则，处理N，使N至少半满
   - 如果N是根节点，且N中只有一个指针，则删除N
   - 如果N是内节点，则处理N，使N至少半满

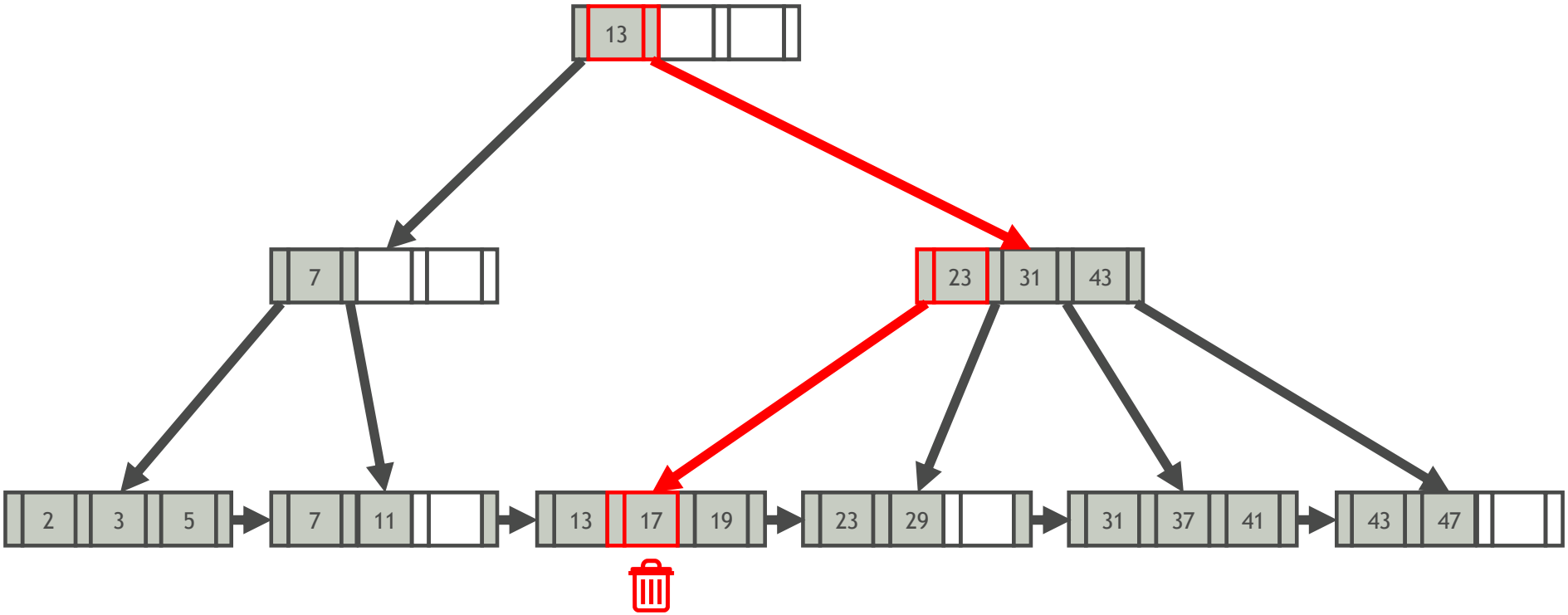
删除索引项(续)

使内节点N至少半满的处理方法

1. 尝试从N相邻的兄弟节点借一个指针及键，使两者均至少半满
2. 如果借不到，则将N与其兄弟节点合并(merge)

删除索引项(续)

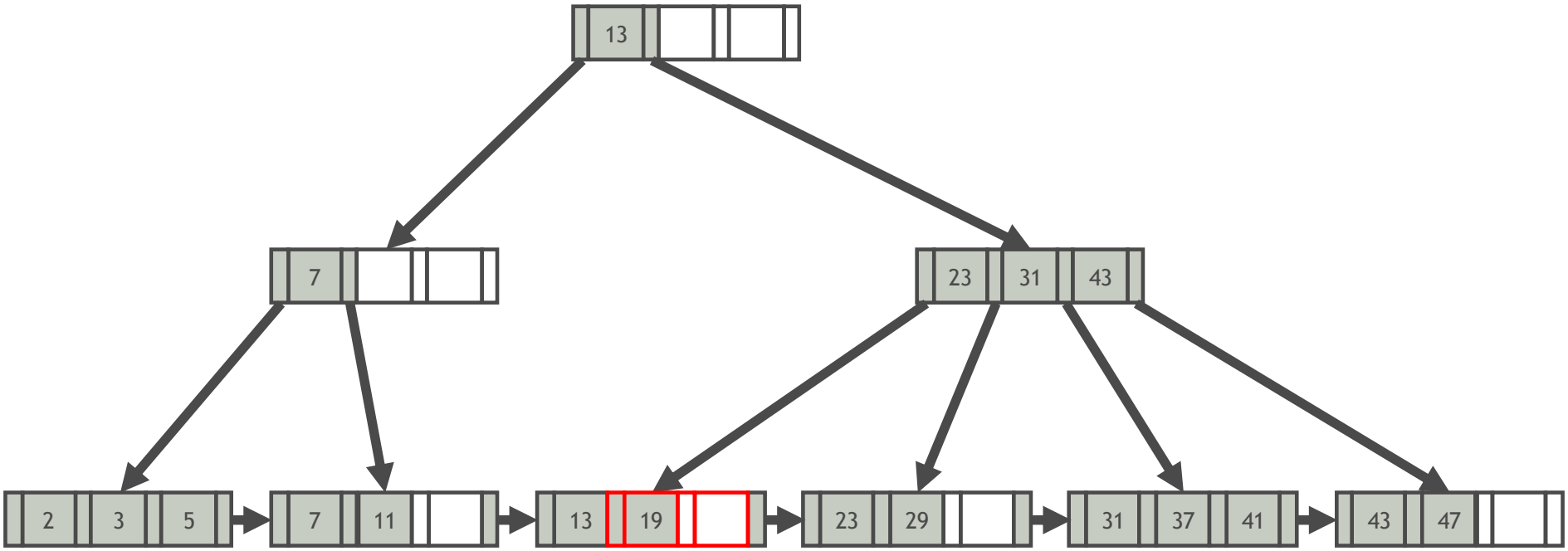
例1: $K = 17$ (没有键重分布及节点合并)



步骤1/2

删除索引项(续)

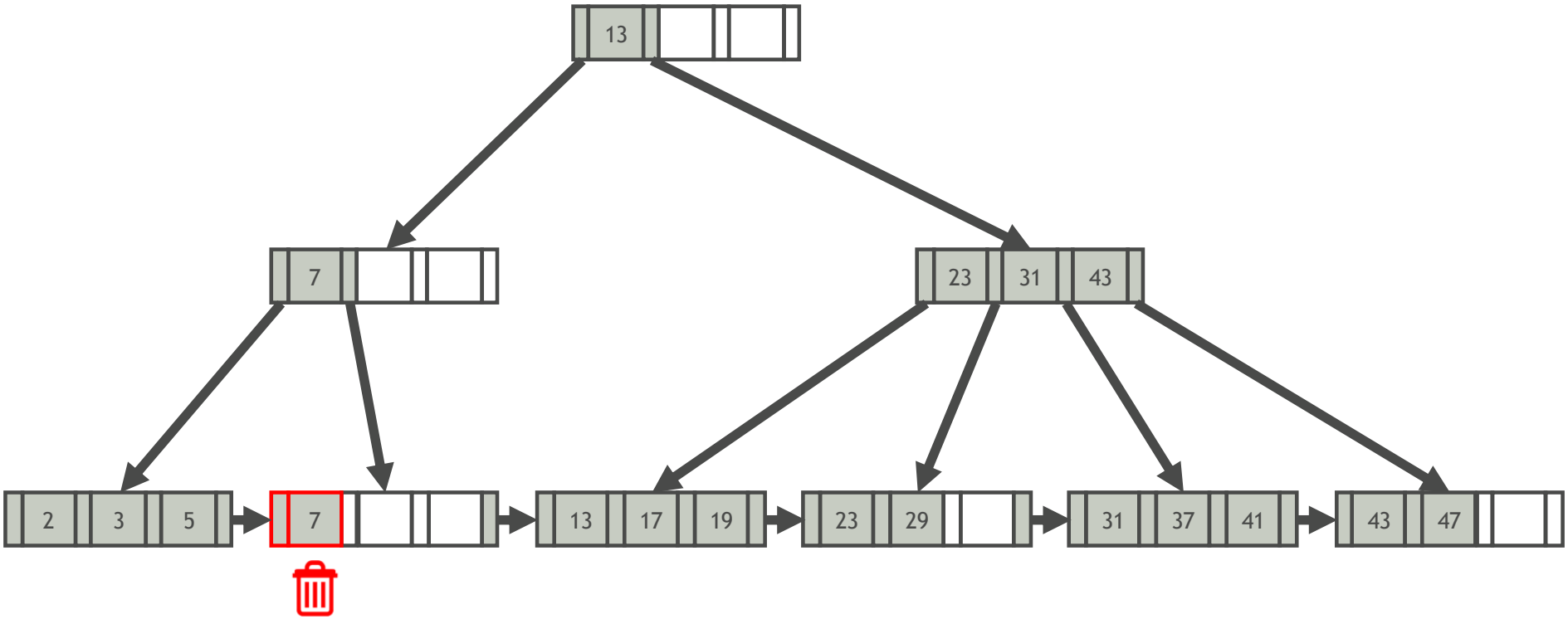
例1: $K = 17$ (没有键重分布及节点合并)



步骤2/2

删除索引项(续)

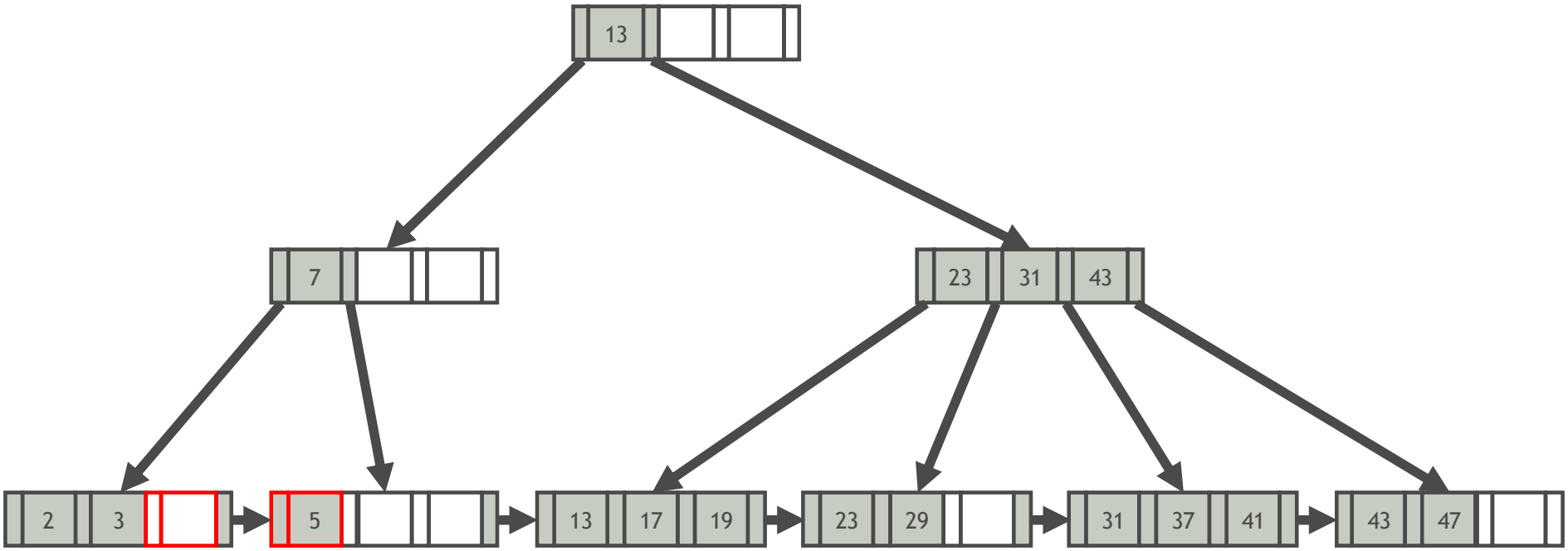
例2: $K = 7$ (需要重分布键)



步骤1/3

删除索引项(续)

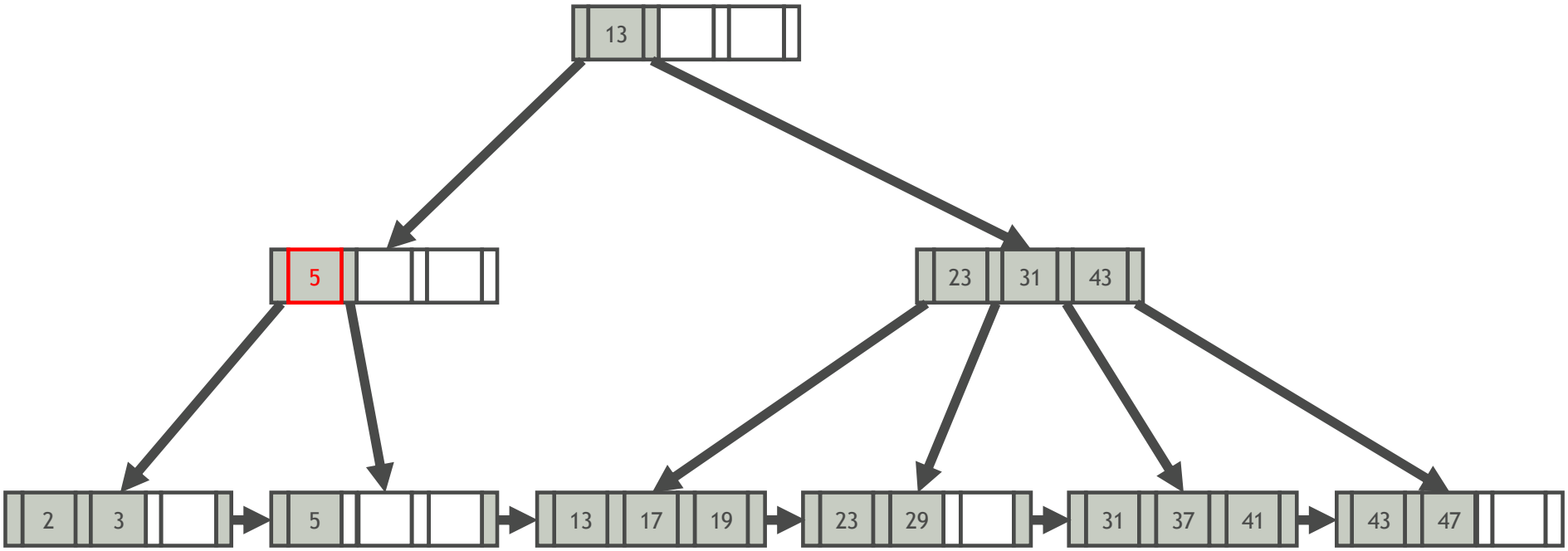
例2: $K = 7$ (需要重分布键)



步骤2/3

删除索引项(续)

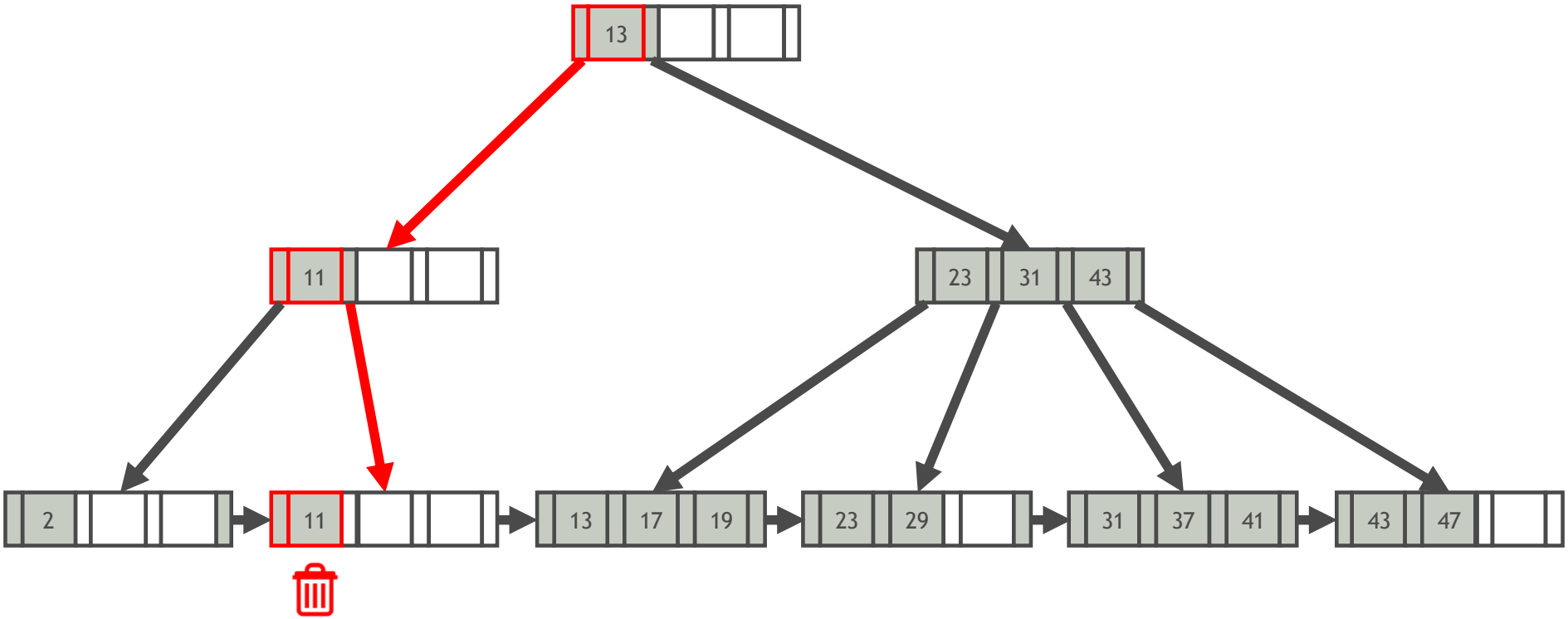
例2: $K = 7$ (需要重分布键)



步骤3/3

删除索引项(续)

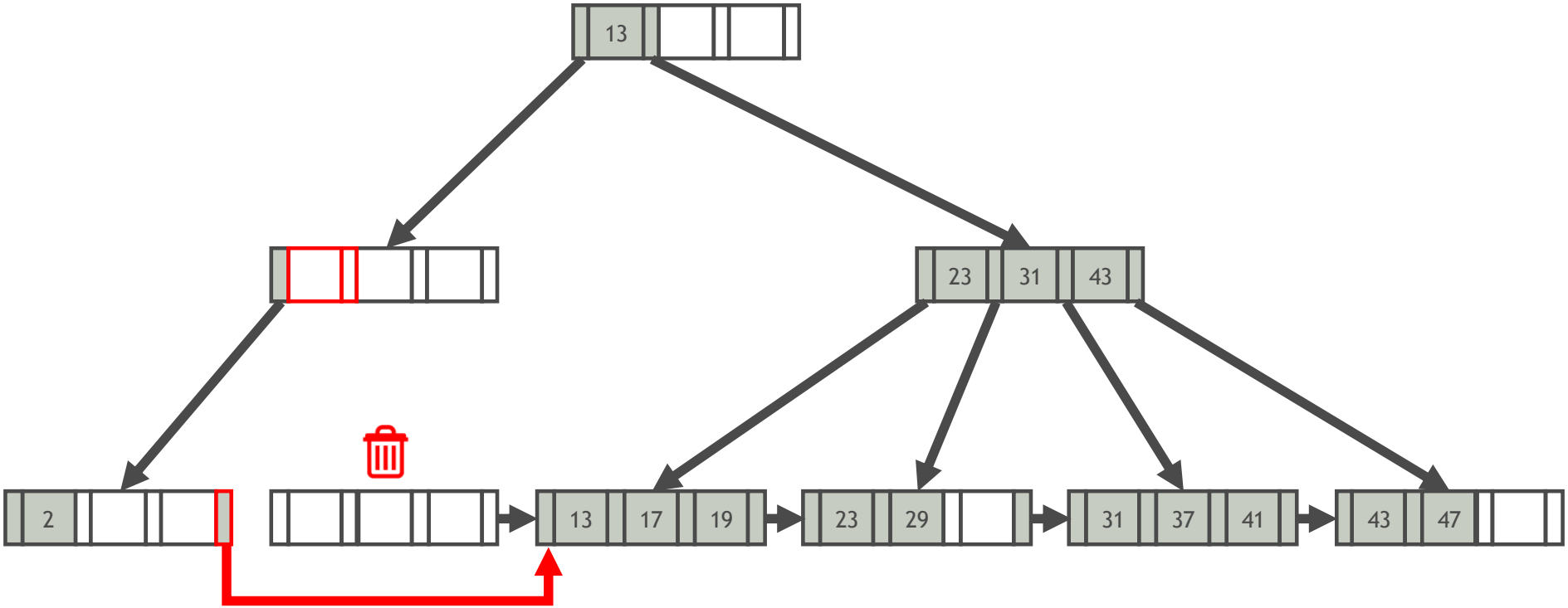
例3: $K = 11$ (需要合并节点)



步骤1/4

删除索引项(续)

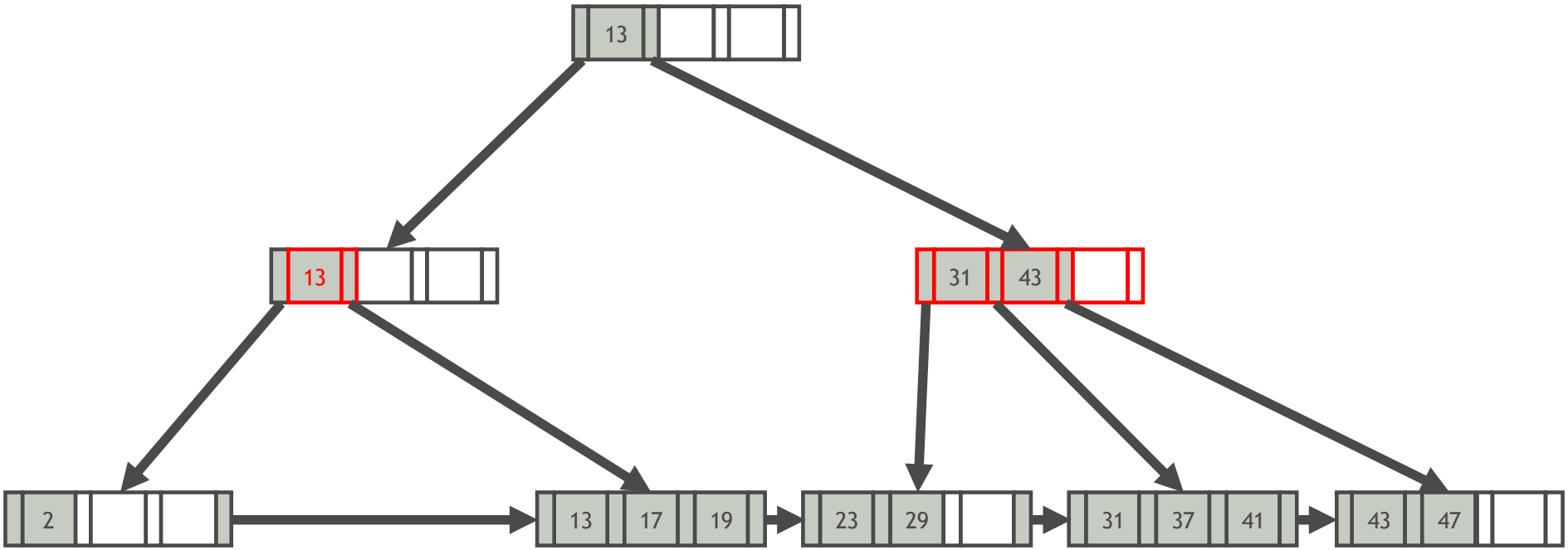
例3: $K = 11$ (需要合并节点)



步骤2/4

删除索引项(续)

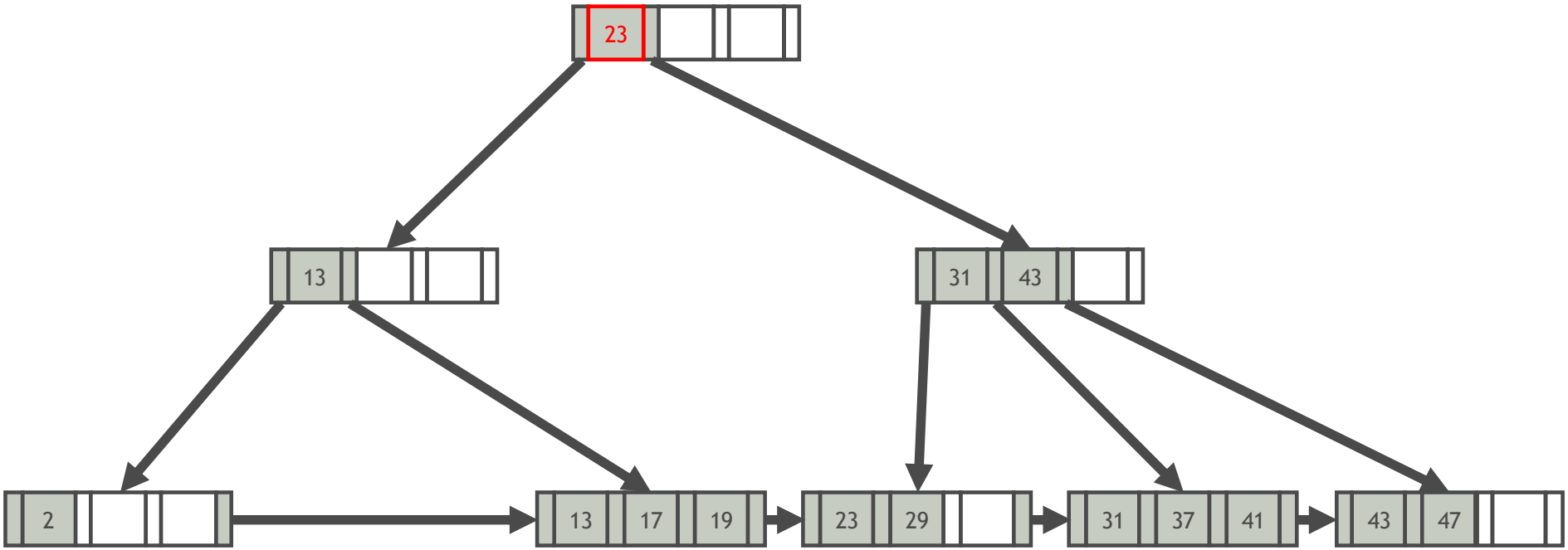
例3: $K = 11$ (需要合并节点)



步骤3/4

删除索引项(续)

例3: $K = 11$ (需要合并节点)



步骤4/4

B+树演示

https://cmudb.io/btree

键压缩(Key Compression)

对键进行压缩，尽可能减少键的长度

- 从B+树中查找一个索引项所需的磁盘I/O数= B+树的高度≈ $\log_{fan\_out}$(# of index entries)
- 索引键越长$\Longrightarrow$ 扇出数越小$\Longrightarrow$ B+树越高$\Longrightarrow$ 查询时间越长

前缀压缩(Prefix Compression)

- 同一叶节点中的键很可能具有相同的前缀(prefix)
- 提取键的公共前缀，只存储每个键的后缀(suffix)

Example (前缀压缩)

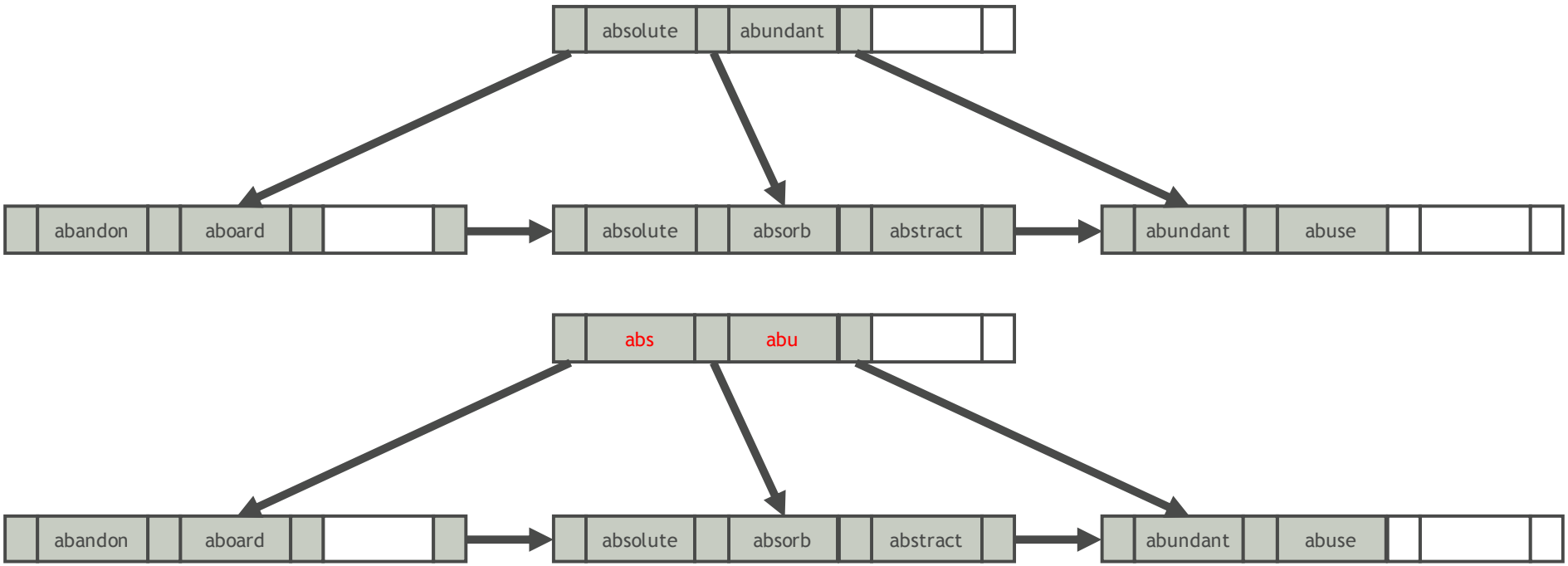
Microphone	Microsoft	Microwave

↓ 前缀压缩

前缀: Micro | phone | soft | wave |

后缀截断(Suffix Truncation)

- 内节点中的键仅用于导航
- 不需要在内节点中存储整个键
- 在保证正确导航的前提下，只需存储每个键的最短前缀即可



批量加载(Bulk Loading)

在一组页中的索引项上建立B+树

自顶向下的方法

- 从一棵空的B+树开始，每次插入一个索引项
- 缺点: 插入每个索引项都需要从根节点向下走到叶节点

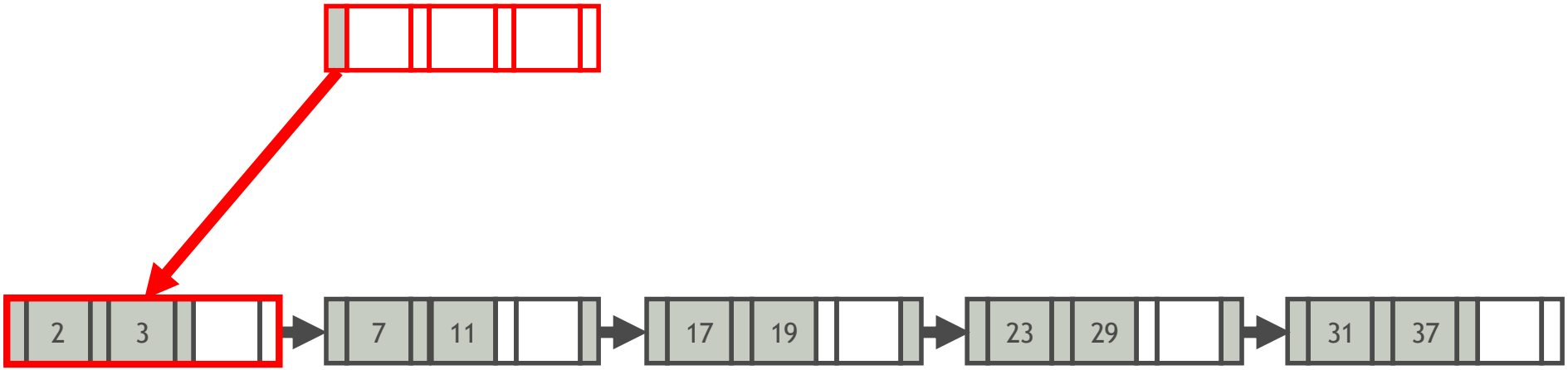
批量加载(续)

自底向上的方法

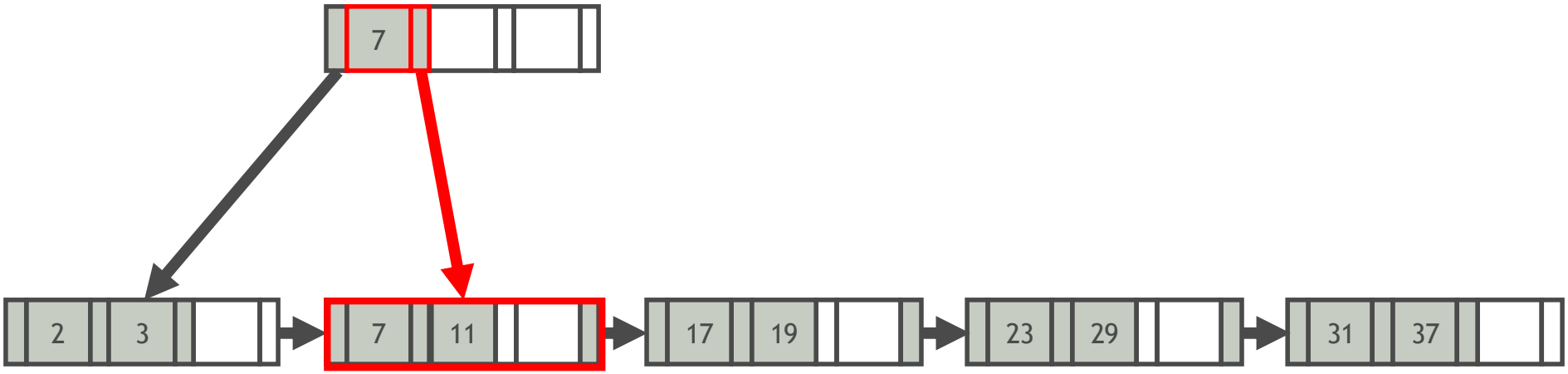
1. 使用外存归并排序算法对所有页中的索引项排序
2. 将每个页作为一个叶节点，建立叶节点链表
3. 创建一个空的内节点作为根，并插入一个指针，指向第一个叶节点
4. 对叶节点链表中的下一个叶节点L，向叶节点上层最右边的内节点插入L中最小的键及指向L的指针；如果内节点溢出，则分裂
5. 重复第4步，直至所有叶节点都插入B+树为止

批量加载(续)

例: 在排好序的键2, 3, 7, 11, 17, 19, 23, 29, 31, 37上建立B+树

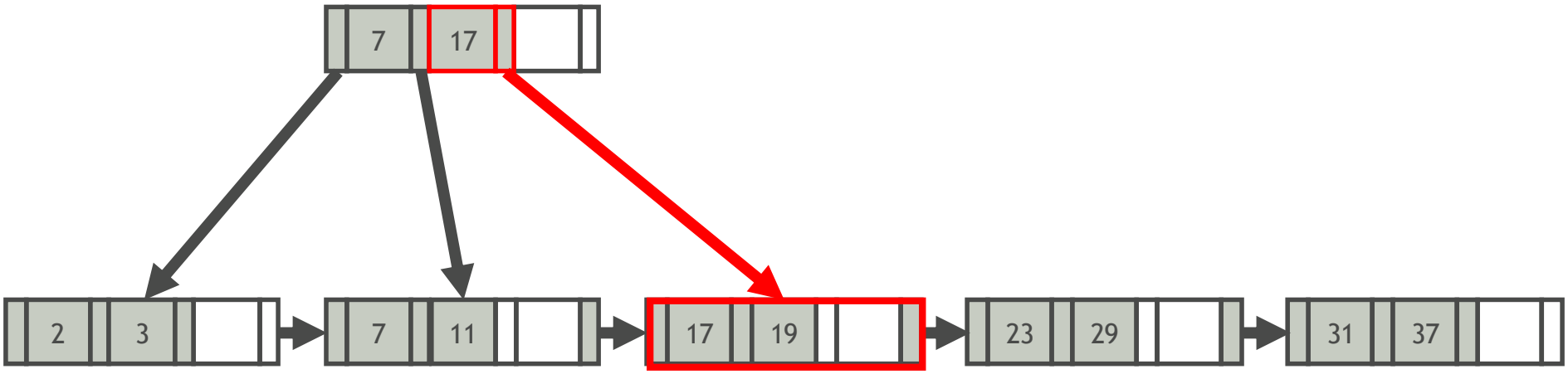


步骤1/5

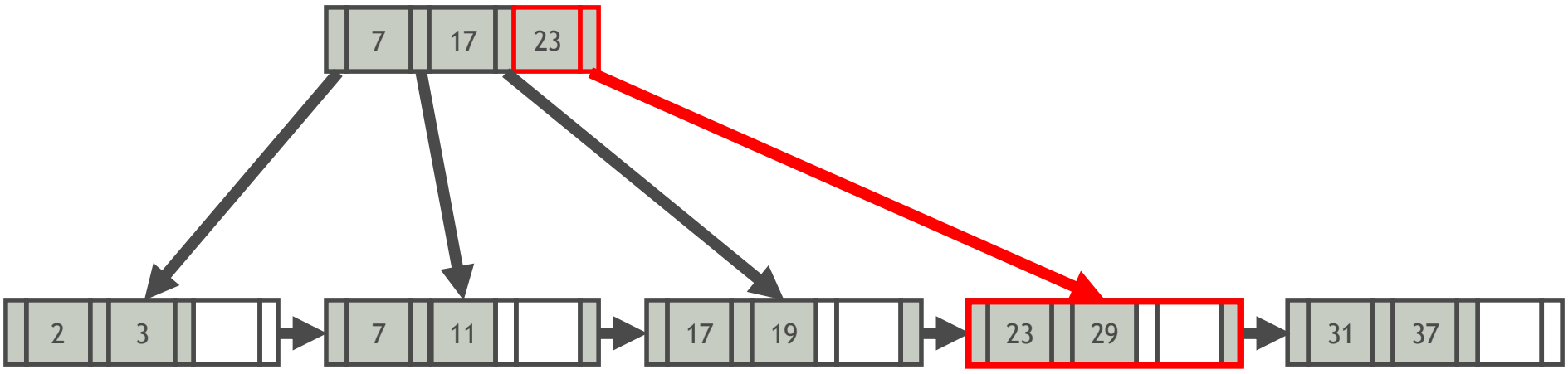


步骤2/5

批量加载(续)

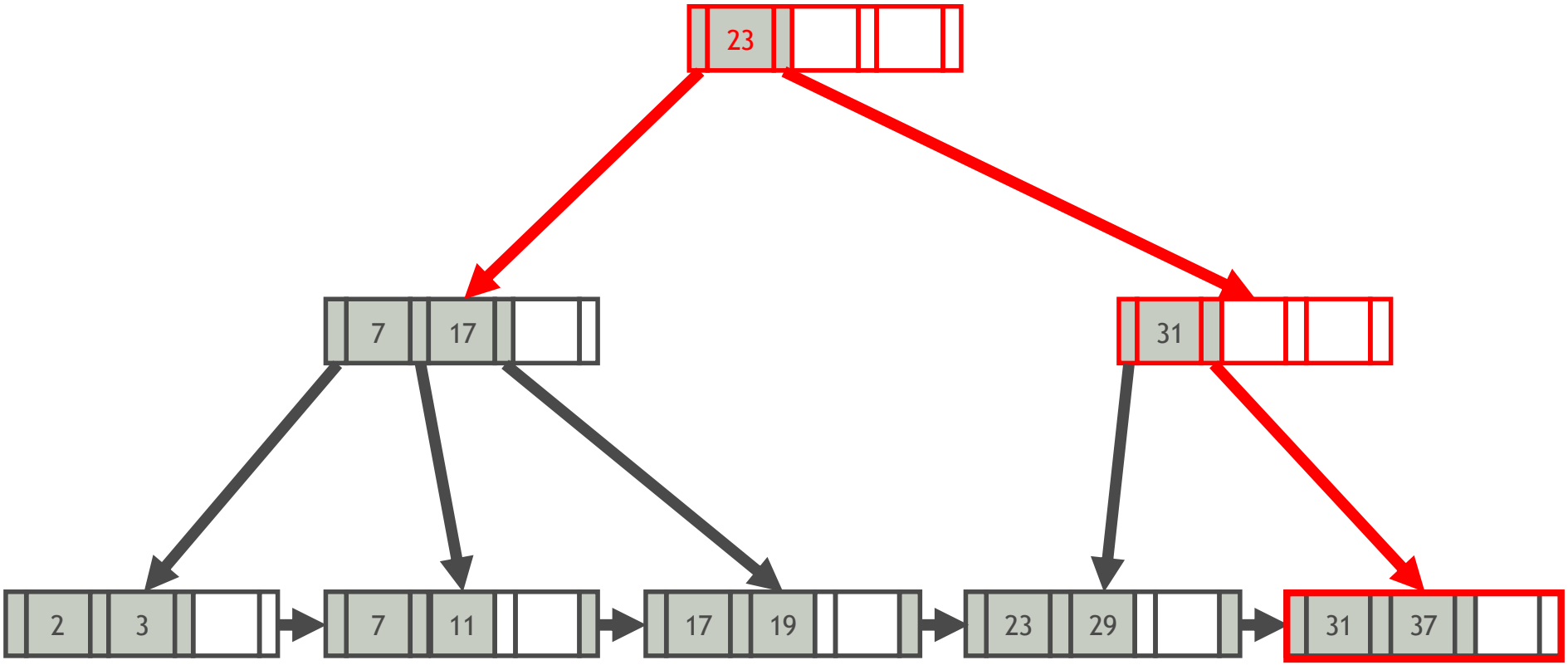


步骤3/5



步骤4/5

批量加载(续)



步骤5/5

Log-Structured Merge-Trees (LSM-Trees)

B+树的原地更新(In-Place Updates)

B+树使用原地更新

- 直接使用新数据覆盖旧数据
- 数据更新过程产生大量随机磁盘I/O

日志结构合并树(Log-Structured Merge-Trees, LSM-Trees)

LSM树被广泛用于NoSQL数据库系统的存储层

- LevelDB、RocksDB、HBase、Cassandra、TiDB等

LSM树执行异地更新(out-of-place updates)

- 写操作首先缓存在内存中
- 内存缓冲区中的写操作后续会刷写到磁盘文件，并与现有文件合并
- 数据更新过程只使用顺序磁盘I/O

LSM树的基本结构

LSM树由两部分构成

- Memtable: 内存B+树或内存哈希表
- 不可变文件(immutable file): 磁盘上不可更新的文件

Example (LSM树)

Memtable | (3, 333), (7, 777) | 内存

Immutable file | (2, 222), (3, 123), (5, 555), (8, 888) | 磁盘

LSM树的查找操作

查找键为K的索引项

1. 首先在memtable中查找键为K的索引项
2. 如果找到，则返回索引项；否则，在不可变文件中查找索引项

Example (LSM树的查找操作)

$K = 3$

Memtable | (3, 333), (7, 777) | 内存

Immutable file | (2, 222), (3, 123), (5, 555), (8, 888) | 磁盘

Example (LSM树的查找操作)

$K = 5$

Memtable | (3, 333), (7, 777) | 内存

Immutable file | (2, 222), (3, 123), (5, 555), (8, 888) | 磁盘

缺点： 当不可变文件非常大时，在文件上查找的效率非常低

LSM树的更新操作

更新键为K的索引项

1. 首先将更新操作缓存在memtable中(原地更新)
2. 当memtable写满后，将memtable的内容与immutable文件的内容合并(compact)，合并后的内容写入新文件，并用新文件替换旧文件

Example (LSM树的更新操作)

合并前

Memtable | (3, 333), (Delete 8) | 内存

Immutable file | (2, 222), (3, 123), (5, 555), (8, 888) | 磁盘

合并后

Memtable | $\emptyset$ | 内存

Immutable file | (2, 222), (3, 333), (5, 555) | 磁盘

缺点：当不可变文件非常大时，合并过程非常慢

分层LSM树

分层LSM树包含多个层

- Memtable: 内存B+树或内存哈希表
- Level 0: memtable在磁盘上的不可变副本(键-值对按键排序)
- Level i ($i \geq 1$): 磁盘上的不可变有序文件(键-值对按键排序)
  - 第$i+1$层的键-值对比第i层的键-值对旧
  - 第$i+1$层的文件比第i层的文件大T倍

Memtable 0-99

Level 0 0-99

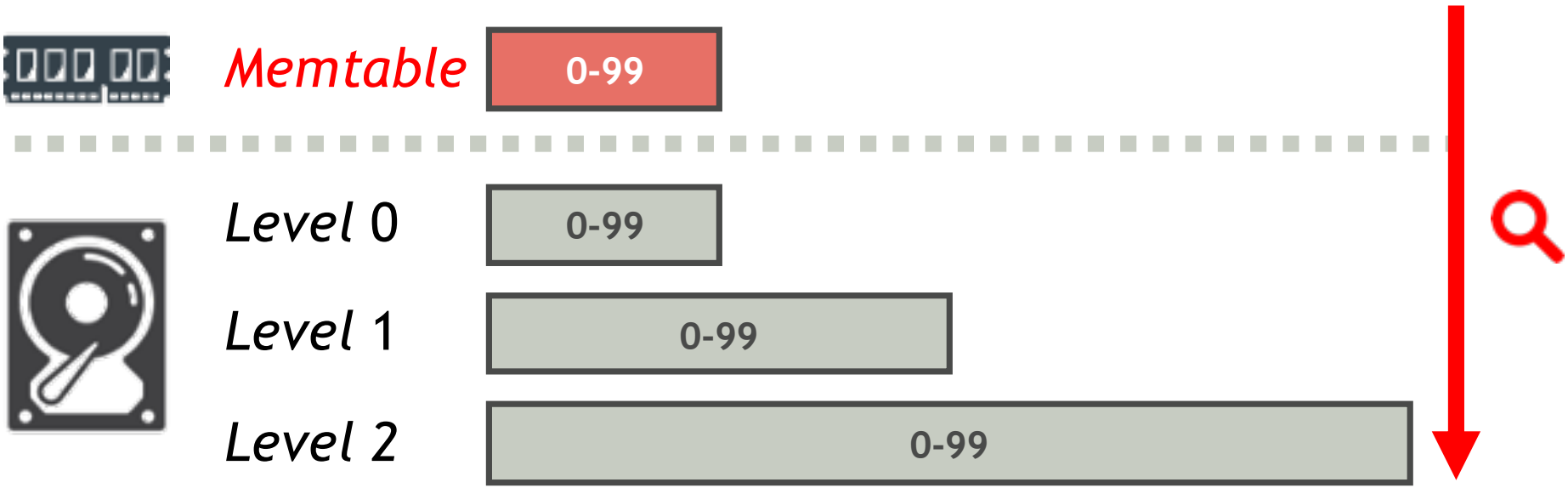
Level 1 0-99

Level 2 0-99

分层LSM树的查找操作

在分层LSM树上，从上向下查找键为K的键-值对

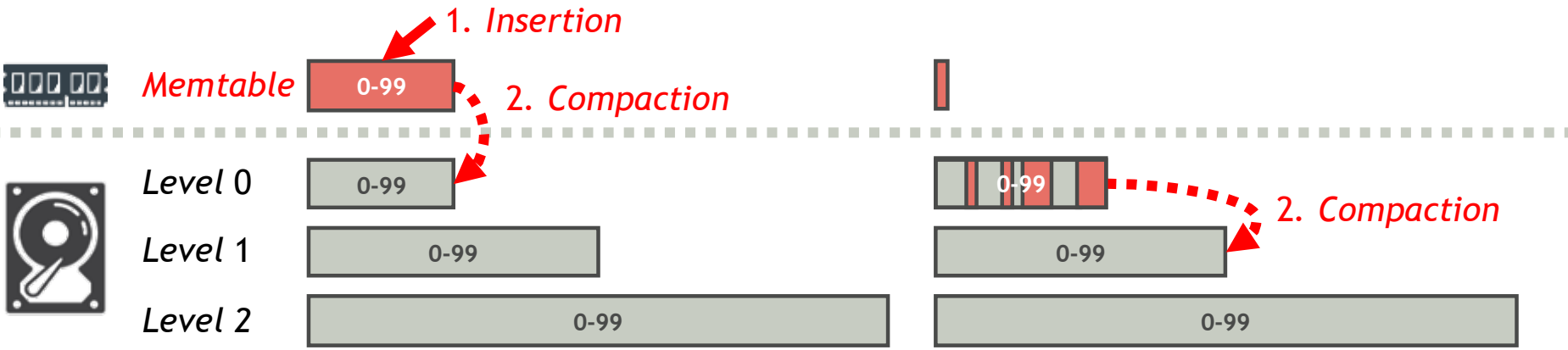
1. 如果找到了键为K的键-值对，则返回该键-值对
2. 如果找到了一个墓碑(tombstone)，则返回“不存在”
3. 如果在所有层上均未找到，则返回“不存在”



分层LSM树的插入操作

插入键-值对(K, V)

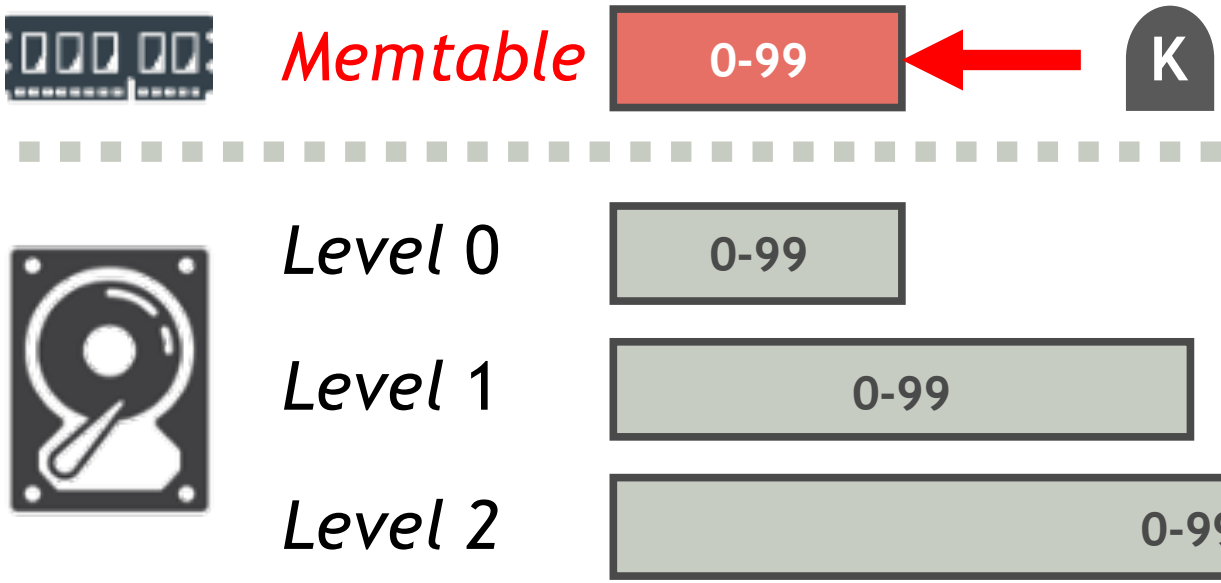
1. 将(K, V)插入memtable (内存中，原地更新)
2. 如果memtable未溢出，则完成插入；否则，将memtable中的键-值对写入第0层，成为不可变的有序文件(磁盘中，顺序I/O)
3. 如果第i层溢出，则将第i层的键-值对合并到第$i+1$层(磁盘中，异地更新，顺序I/O)



分层LSM树的删除操作

删除键为K的键-值对

1. 在memtable中为K插入一个墓碑(tombstone)
2. 在合并时，删除键为K且比墓碑旧的键-值对



B+树VS LSM-树

	B+树	LSM-树
更新方法	原地更新	异地更新
空间放大(space amplification)	低(一个键只有1个副本)	高(一个键有多个副本)
写性能	低(随机I/O)	高(顺序I/O)
空间利用率	空间碎片化(平均一个页有1/4空闲)	高(键-值对在不可变文件中有序存储)
并发控制与故障恢复	复杂	简单(文件不可更改且合并操作只进行异地更新)

Advanced Topics in Indexing

位图索引(Bitmap Index)

当属性的取值较少时，可以使用位图索引加快多属性上的选择查询

- 对关系R的属性A的每个取值v建一个长度为$|R|$的位图I_v
- I_v的第i位为1当且仅当R中第i个元组的A属性值为v

Example (位图索引)

Student关系

编号	Sno	Sname	Ssex	Sage	Sdept
0	CS-001	Elsa	F	19	CS
1	CS-002	Ed	M	19	CS
2	MA-001	Abby	F	18	Math
3	PH-001	Nick	M	20	Physics

Ssex上的位图索引

'F': 1010

'M': 0101

Sdept上的位图索引

'CS': 1100

'Math': 0010

'Physics': 0001

```
SELECT * FROM Student WHERE Ssex = 'F' AND Sdept = 'CS';
```

'F': 1010 bitwise AND 'CS': 1100 = 1000，因此元组0是查询结果

空间索引(Spatial Index)

空间索引用于索引空间数据(spatial data)

- kd树(kd-tree)
- R树(R-tree)

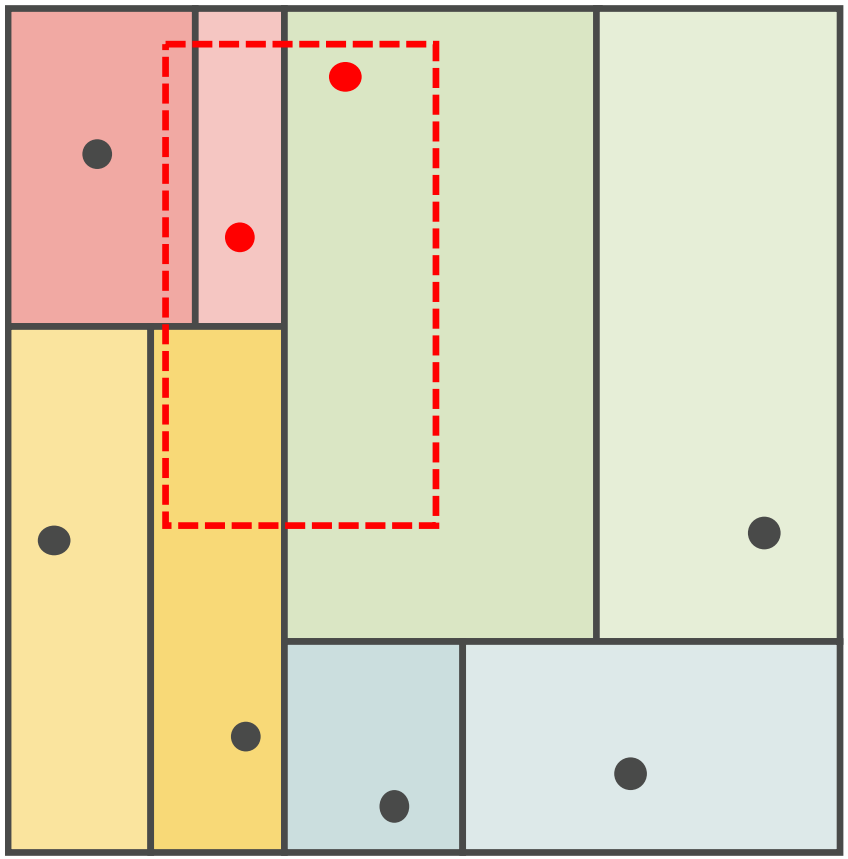
空间数据查询

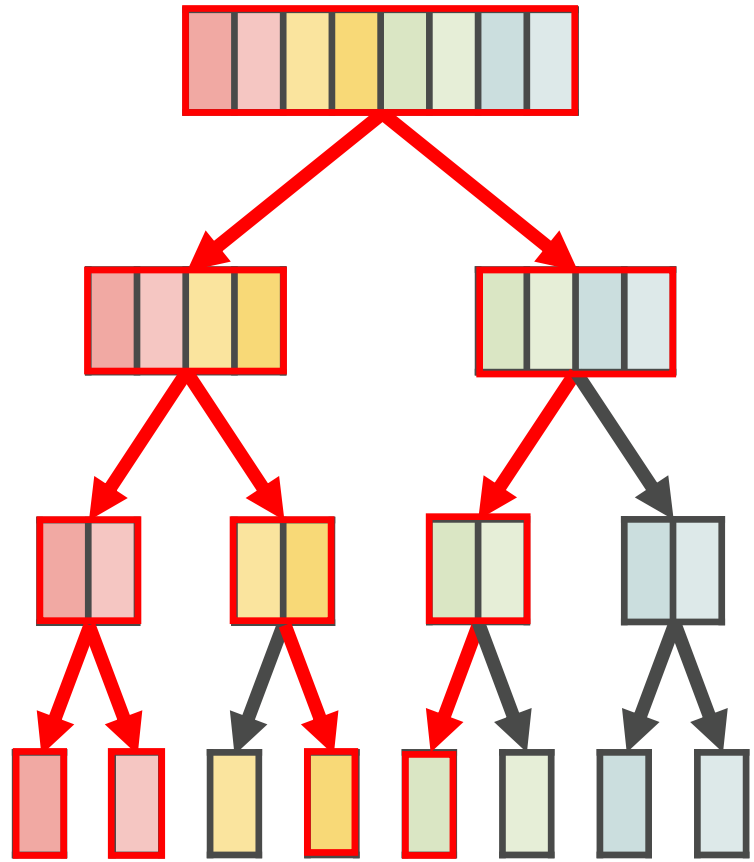
- 范围查询(range query): 查询位于给定范围内的数据对象
- 近邻查询(nearest neighbor query): 查询与一个给定对象距离最近的一个或多个其他对象

kd树(kd-Tree)

kd树是一种多维索引结构

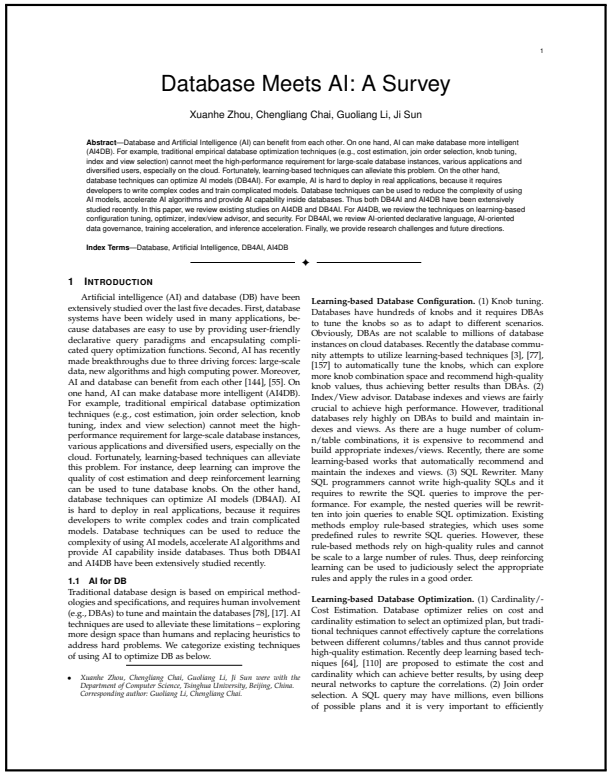
- kd树的每一层都把数据空间分成两部分
- 如果kd树的一个节点代表的子空间与查询区域不相交，则该节点中的所有数据对象都不可能是查询结果





基于人工智能的索引结构

使用人工智能技术，优化索引结构设计或设计基于模型的索引结构



Database Meets AI: A Survey

Xuanhe Zhou, Chengliang Chai, Guoliang Li, Ji Sun

Abstract—Database and Artificial Intelligence (AI) can benefit from each other. On one hand, AI can make database more intelligent (AI4DB). For example, traditional empirical database optimization techniques (e.g., cost estimation, join order selection, knob tuning, index and view selection) cannot meet the high-performance requirement for large-scale database instances, various applications and diversified users, especially on the cloud. Fortunately, learning-based techniques can alleviate this problem. On the other hand, database techniques can optimize AI models (DB4AI). For example, AI is hard to deploy in real applications, because it requires developers to write complex codes and train complicated models. Database techniques can be used to reduce the complexity of using AI models, accelerate AI algorithms and provide AI capability inside databases. Thus both DB4AI and AI4DB have been extensively studied recently. In this paper, we review existing studies on AI4DB and DB4AI. For AI4DB, we review the techniques on learning-based configuration tuning, optimizer, index/view advisor, and security. For DB4AI, we review AI-oriented declarative language, AI-oriented data governance, training acceleration, and inference acceleration. Finally, we provide research challenges and future directions.

Index Terms—Database, Artificial Intelligence, DB4AI, AI4DB

1 Introduction

Artificial intelligence (AI) and database (DB) have been extensively studied over the last five decades. First, database systems have been widely used in many applications, because databases are easy to use by providing user-friendly declarative query paradigms and encapsulating complicated query optimization functions. Second, AI has recently made breakthroughs due to three driving forces: large-scale data, new algorithms and high computing power. Moreover, AI and database can benefit from each other [144], [55]. On one hand, AI can make database more intelligent (AI4DB). For example, traditional empirical database optimization techniques (e.g., cost estimation, join order selection, knob tuning, index and view selection) cannot meet the high-performance requirement for large-scale database instances, various applications and diversified users, especially on the cloud. Fortunately, learning-based techniques can alleviate this problem. For instance, deep learning can improve the quality of cost estimation and deep reinforcement learning can be used to tune database knobs. On the other hand, database techniques can optimize AI models (DB4AI). AI is hard to deploy in real applications, because it requires developers to write complex codes and train complicated models. Database techniques can be used to reduce the complexity of using AI models, accelerate AI algorithms and provide AI capability inside databases. Thus both DB4AI and AI4DB have been extensively studied recently.

1.1 AI for DB

Traditional database design is based on empirical methodologies and specifications, and requires human involvement (e.g., DBAs) to tune and maintain the databases [78], [17]. AI techniques are used to alleviate these limitations – exploring more design space than humans and replacing heuristics to address hard problems. We categorize existing techniques of using AI to optimize DB as below.

- *Xuanhe Zhou, Chengliang Chai, Guoliang Li, Ji Sun were with the Department of Computer Science, Tsinghua University, Beijing, China. Corresponding author: Guoliang Li, Chengliang Chai.*

Learning-based Database Configuration. (1) Knob tuning. Databases have hundreds of knobs and it requires DBAs to tune the knobs so as to adapt to different scenarios. Obviously, DBAs are not scalable to millions of database instances on cloud databases. Recently the database community attempts to utilize learning-based techniques [3], [77], [157] to automatically tune the knobs, which can explore more knob combination space and recommend high-quality knob values, thus achieving better results than DBAs. (2) Index/View advisor. Database indexes and views are fairly crucial to achieve high performance. However, traditional databases rely highly on DBAs to build and maintain indexes and views. As there are a huge number of column/table combinations, it is expensive to recommend and build appropriate indexes/views. Recently, there are some learning-based works that automatically recommend and maintain the indexes and views. (3) SQL Rewriter. Many SQL programmers cannot write high-quality SQLs and it requires to rewrite the SQL queries to improve the performance. For example, the nested queries will be rewritten into join queries to enable SQL optimization. Existing methods employ rule-based strategies, which uses some predefined rules to rewrite SQL queries. However, these rule-based methods rely on high-quality rules and cannot be scale to a large number of rules. Thus, deep reinforcing learning can be used to judiciously select the appropriate rules and apply the rules in a good order.

Learning-based Database Optimization. (1) Cardinality/Cost Estimation. Database optimizer relies on cost and cardinality estimation to select an optimized plan, but traditional techniques cannot effectively capture the correlations between different columns/tables and thus cannot provide high-quality estimation. Recently deep learning based techniques [64], [110] are proposed to estimate the cost and cardinality which can achieve better results, by using deep neural networks to capture the correlations. (2) Join order selection. A SQL query may have millions, even billions of possible plans and it is very important to efficiently

X. Zhou, C. Chai, G. Li, J. Sun. **Database Meets Artificial Intelligence: A Survey**. *IEEE Transactions on Knowledge and Data Engineering*, 34(3):1096–1116, 2022.

Section 2.3.1 “Learned Data Structure”

总结

1. Indexes
2. Hash-based Index Structures
   - Extensible Hash Tables
   - Linear Hash Tables
3. Tree-based Index Structures
   - B+ Trees
4. Log-Structured Merge-Trees (LSM-Trees)
5. Advanced Topics in Indexing

Q&A

1. 当B+树进行删除操作时，若一个节点不足半满，是优先向左兄弟借，还是优先向右兄弟借呢?
   答: 都可以，取决于B+树的具体实现方法。

致谢

- 感谢詹儒彦(1190202307)、金彦铮(1190200418)、杨宇辰(1190300611)、蔡思娣(1190201925)同学指出课件中的错误