Chapter 8: Index Structures

第8章：索引结构

Zhaonian Zou

Massive Data Computing Research Center
School of Computer Science and Technology
Harbin Institute of Technology, China
Email: znzou@hit.edu.cn

Spring 2020

Outline[1]

大纲1

1基于哈希的索引结构
可扩展哈希表
线性哈希表

2基于树的索引结构
B +树

[1] Updated on March 28, 2020

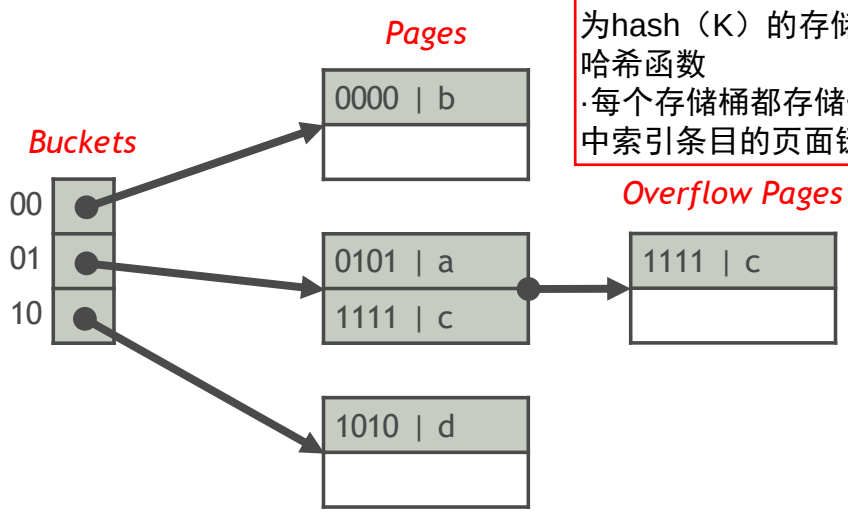
基于哈希的索引结构

Hash-based Index Structures

Secondary-Storage Hash Tables (外存哈希表)

- A secondary-storage hash table consists of a number of buckets
- An index entry with key K is put in the bucket numbered $hash(K)$, where *hash* is a hash function
- Each bucket stores a pointer to a linked list of pages holding the index entries in the bucket

·外存哈希表由多个存储桶组成
·将具有键K的索引条目放入编号为hash（K）的存储桶中，其中hash是哈希函数
·每个存储桶都存储一个指向指向存储桶中索引条目的页面链接列表的指针



Categories of Secondary-Storage Hash Tables

Static Hash Tables (静态哈希表)

- The number of buckets does not change

Dynamic Hash Tables (动态哈希表)

- The number of buckets is allowed to vary so that there is about one block per bucket
- Extensible hash tables (可扩展哈希表)
- Linear hash tables (线性哈希表)

二级存储哈希表的类别

静态哈希表
·桶数不变
动态哈希表
·桶的数量允许变化，以便每个桶大约有一个块
可扩展哈希表
线性哈希表

基于哈希的索引结构
可扩展哈希表

Hash-based Index Structures
Extensible Hash Tables

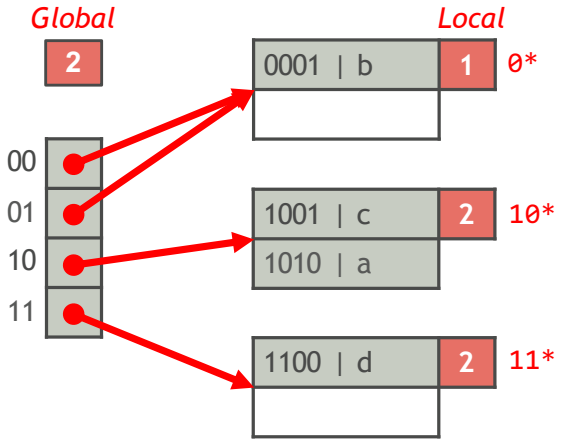
Extensible Hash Tables (可扩展哈希表)

An extensible hash table is comprised of 2^i buckets

- i is called the global depth
- An index entry with key K belongs to the bucket numbered by the first i bits of $hash(K)$

可扩展的哈希表由2i个存储桶组成
i被称为全局深度
索引键为K的索引条目属于以hash（K）的前i位编号的存储桶

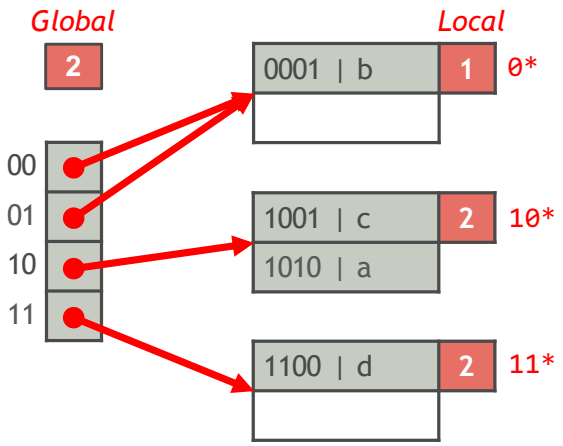
Example:
$hash(a) = 1010, hash(b) = 0001, hash(c) = 1001, hash(d) = 1100$



Extensible Hash Tables (Cont'd)

Every bucket keeps a pointer to a page where the index entries in the bucket are stored

- Several buckets can share a page if all the index entires in those buckets can fit in the page
- Every page records # bits of $hash(K)$ (local depth) used to determine the membership of index entires in this page



每个存储桶都有一个指向存储该存储桶中索引条目的页面的指针
·如果这些存储桶中的所有索引都可以装入页面，则多个存储桶可以共享一个页面
·每页记录＃位哈希值（K）（局部深度），这些值用于确定此页面中索引整体的成员资格

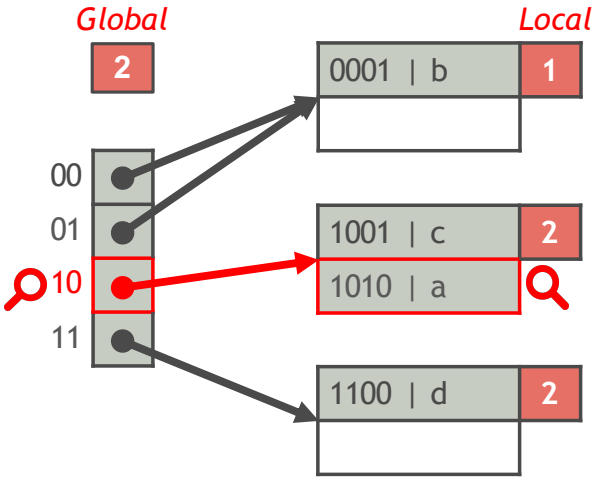
Extensible Hash Table Lookup

用键K查找索引条目
1确定条目所属的存储桶
2在存储桶指向的页面中找到条目
例如：K = a，hash（a）= 1010

Find the index entry with key K

1. Determine the bucket where the entry belongs to
2. Find the entry in the page that the bucket points to

Example: $K = a$, $hash(a) = 1010$



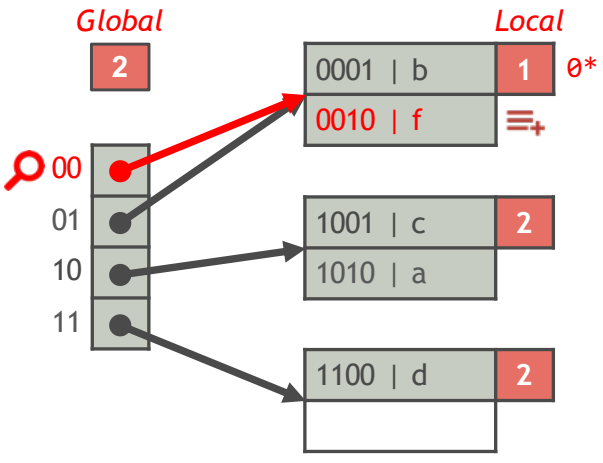
Extensible Hash Table Insert

可扩展哈希表插入
用键K插入索引条目
1找到要插入条目的页面P
2如果P有足够的空间，请完成！
否则，将P分为P和新页面P0
示例：K = f，hash（f）= 0010

Insert an index entry with key K

1. Find the page P where the entry is to be inserted
2. If P has enough space, done!
   Otherwise, split P into P and a new page P'

Example: $K = f$, $hash(f) = 0010$

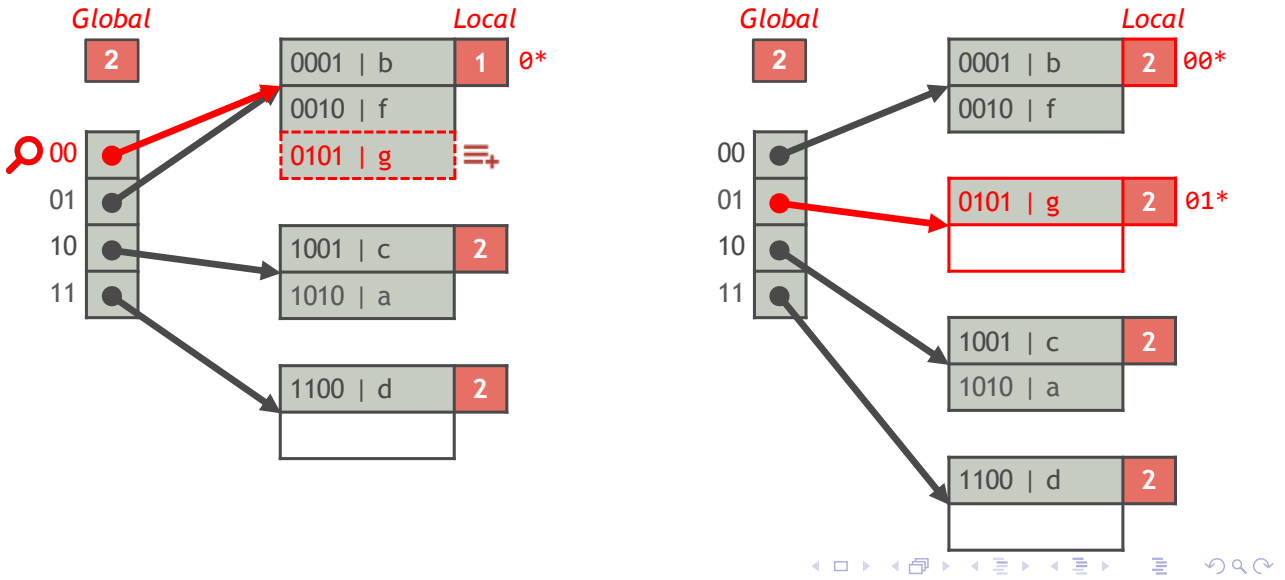


可扩展哈希表插入（续）
如果P溢出，并且P的局部深度小于整体深度，
1将P的局部深度增加1
2将P中的某些索引条目重新分配给新的存储区页面P0（P和P0具有相同的局部深度）

If P overflows and the local depth of P is less than the global depth,

1. Increase P's local depth by 1
2. Re-assign some index entries in P to a new bucket page P' (P and P' have the same local depth)

Example: $K = g$, $hash(g) = 0101$



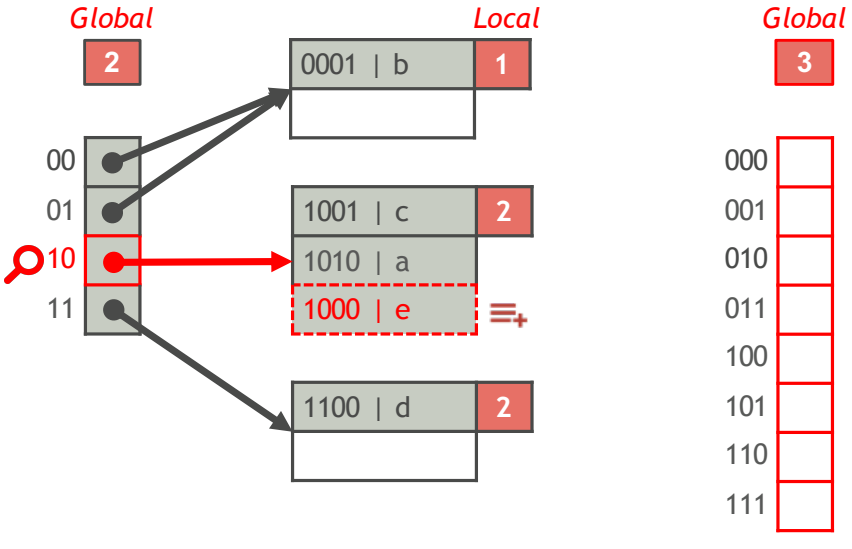
可扩展哈希表插入（续）
如果P溢出，并且P的局部深度等于整体深度，
1将全局深度增加1（双倍桶数）
2重新组织水桶； 如果页面溢出，则将其拆分

Ext t'd)

If P overflows and the local depth of P is equal to the global depth,

1. Increase the global depth by 1 (double # buckets)
2. Re-organize the buckets; if a page overflows, split it

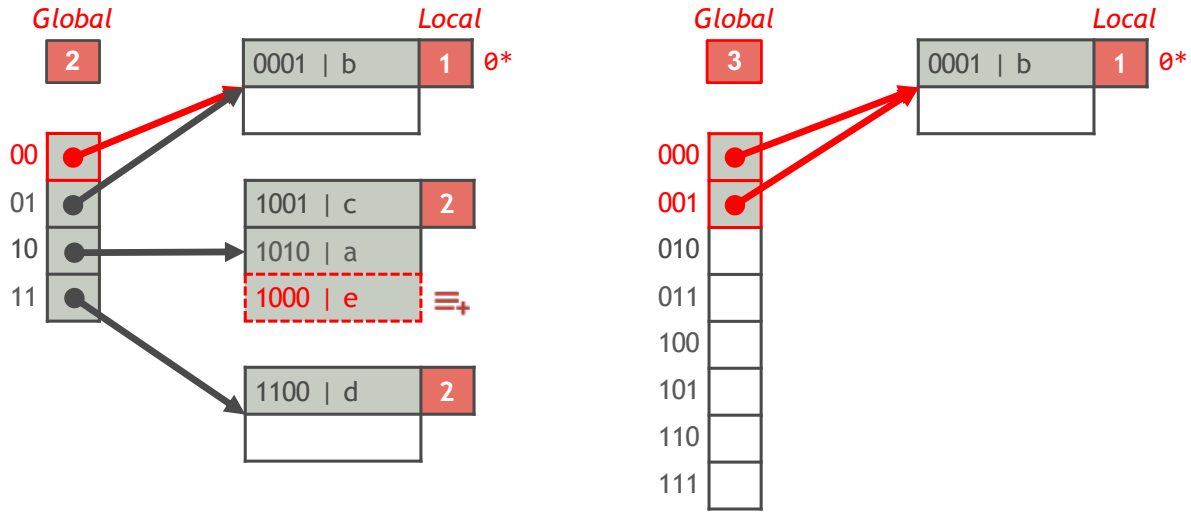
Example: $K = e$, $hash(e) = 1000$



可扩展哈希表插入：示例

Extensible Hash Table Insert: Example

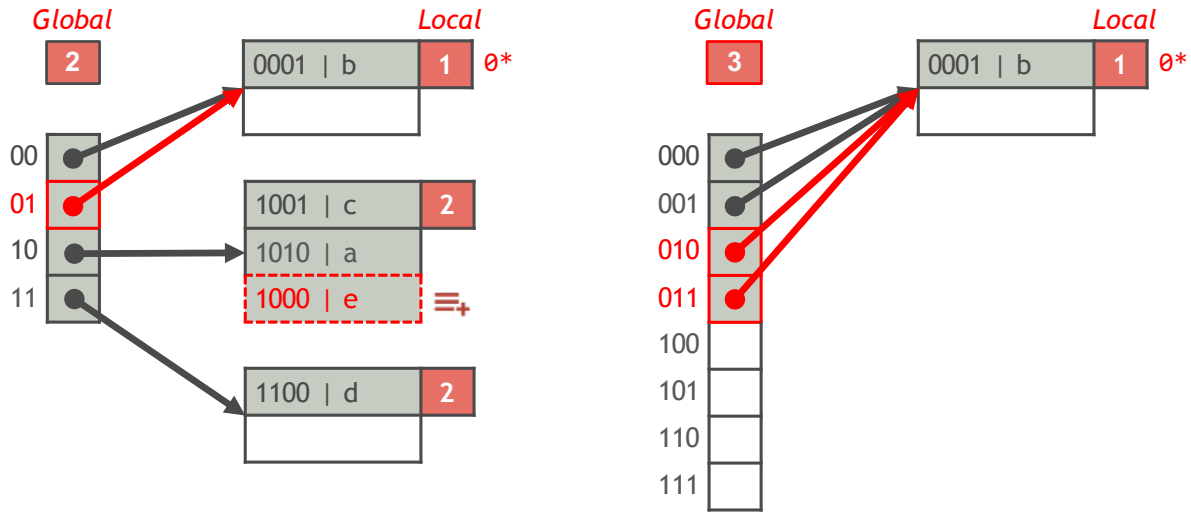
Example: $K = e$, $hash(e) = 1000$



可扩展哈希表插入：示例

Extensible Hash Table Insert: Example

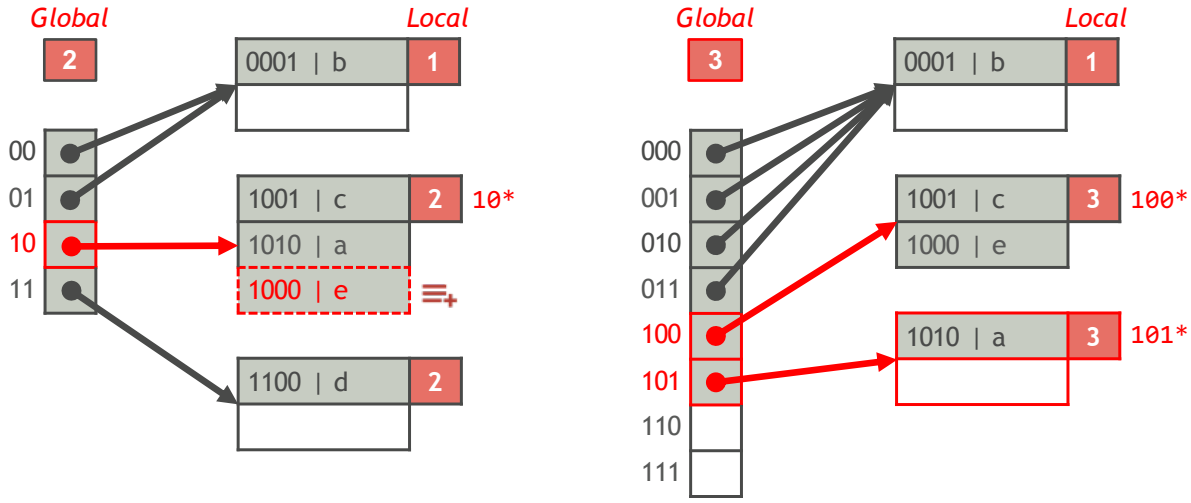
Example: $K = e$, $hash(e) = 1000$



可扩展哈希表插入：示例

Extensible Hash Table Insert: Example

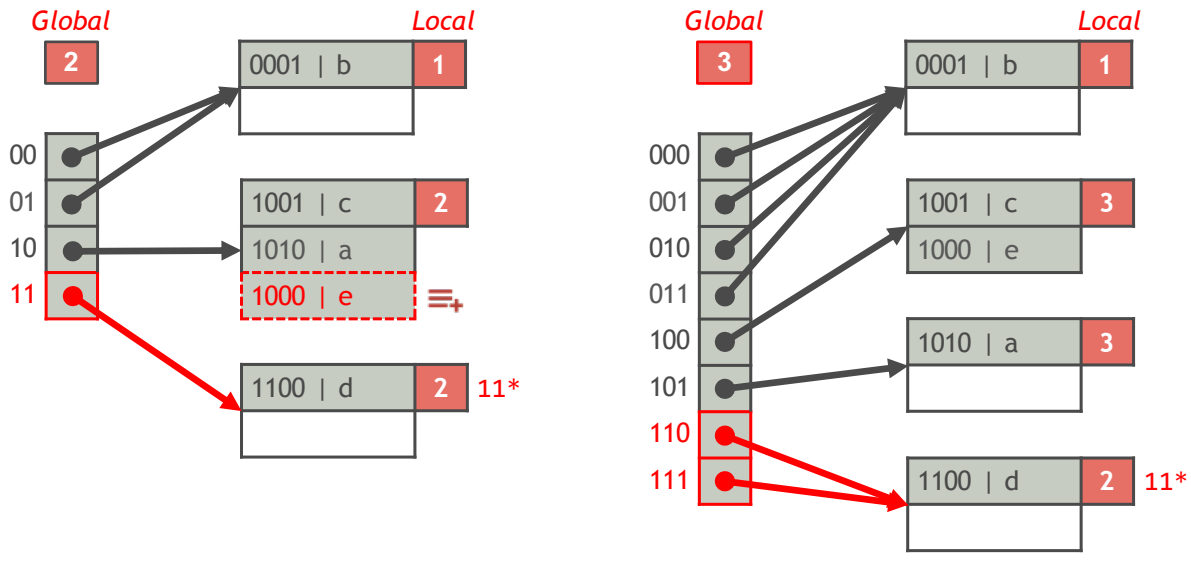
Example: $K = e$, $hash(e) = 1000$



可扩展哈希表插入：示例

Extensible Hash Table Insert: Example

Example: $K = e$, $hash(e) = 1000$



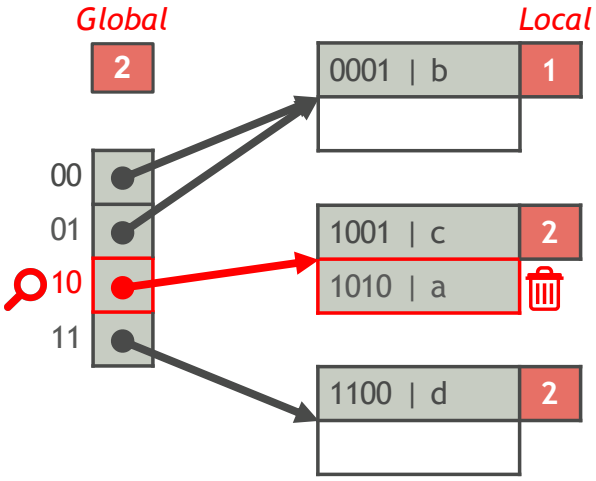
Extensible Hash Table Delete

可扩展哈希表删除
用键K删除索引条目
1查找条目所属的页面
2从页面删除条目

Delete the index entry with key K

1. Find the page where the entry belongs to
2. Delete the entry from the page

Example: $K = a$, $hash(a) = 1010$



基于哈希的索引结构
线性哈希表

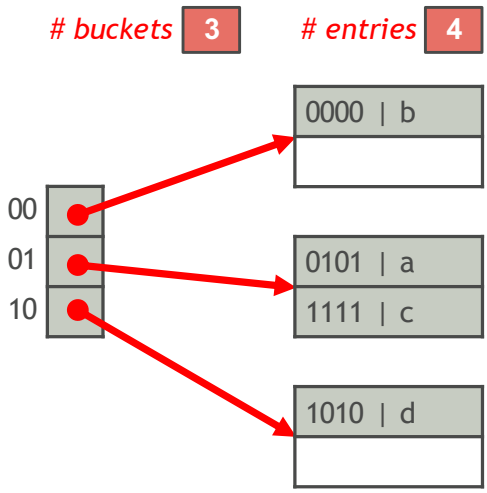
Hash-based Index Structures
Linear Hash Tables

线性哈希表
线性哈希表由n个存储桶组成
每个存储桶都有一个指向存储该存储桶中索引条目的页面链接列表的指针
假设每个页面最多可容纳b个索引条目。 线性哈希表最多存储✓bn个条目，其中0 <✓<1是阈值

A linear hash table is comprised of n buckets

- Every bucket keeps a pointer to a linked list of pages holding the index entries in the bucket
- Suppose each page can hold at most b index entries. The linear hash table stores at most θbn entries, where $0 < \theta < 1$ is a threshold

Example: $b = 2$, $\theta = 0.85$

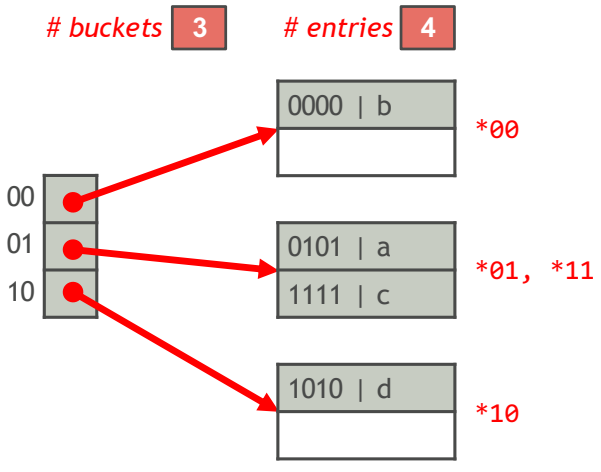


散列方案
桶的编号从0到n 1
令m = 2blog2 nc，因此mðn <2m
如果hash（K）mod 2m <n，则键为K的索引条目属于存储区hash（K）mod 2m； 否则，它属于bucket hash（K）mod m

Hashing Scheme

- The buckets are numbered from 0 to $n - 1$
- Let $m = 2^{\lfloor \log_2 n \rfloor}$, so $m \leq n < 2m$
- If $hash(K) \mod 2m < n$, index entry with key K belongs to bucket $hash(K) \mod 2m$; Otherwise, it belongs to bucket $hash(K) \mod m$

Example:
$hash(a) = 0101, hash(b) = 0000, hash(c) = 1111, hash(d) = 1010$



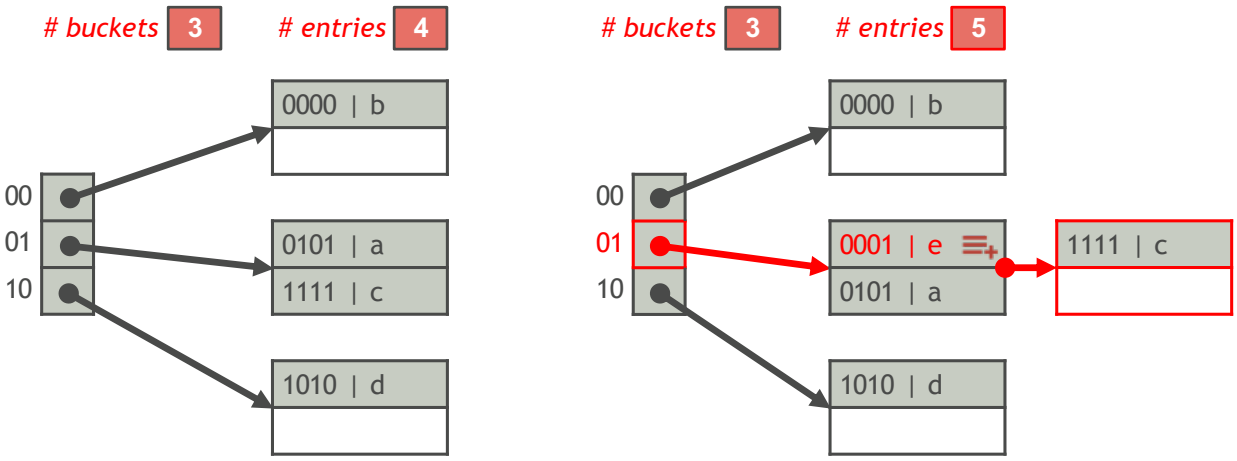
线性哈希表插入
用键K插入索引条目
1将条目插入到它所属的存储桶B中
2将＃条目增加1
3如果＃个条目ð✓bn，那就完成了！
否则，将＃个存储桶加1，然后重新分配B中的条目

Linear Hash Table Insert

Insert an index entry with key K

1. Insert the entry into the bucket B where it belongs to
2. Increase # entries by 1
3. If # entries $\leq \theta bn$, done!
   Otherwise, increase # buckets by 1 and redistribute the entries in B

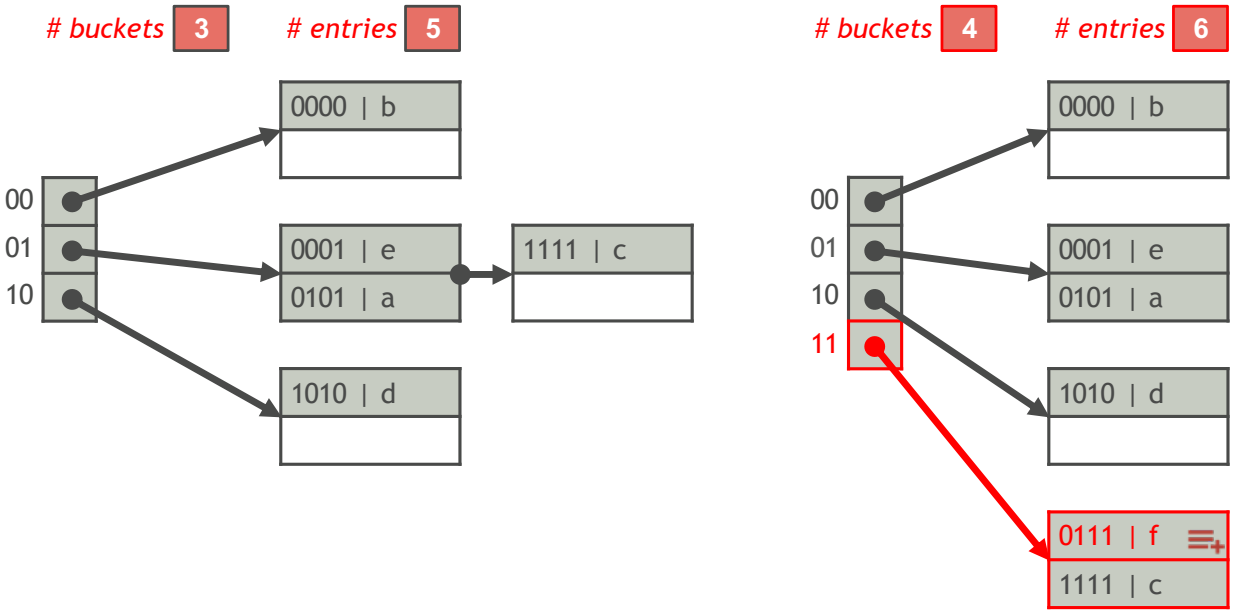
Example: $hash(e) = 0001$, $\theta = 0.85$



线性哈希表插入（续）

Linear Hash Table Insert (Cont'd)

Example: $hash(f) = 0111$, $\theta = 0.85$



基于树的索引结构

Tree-based Index Structures

基于树的索引结构
B +树

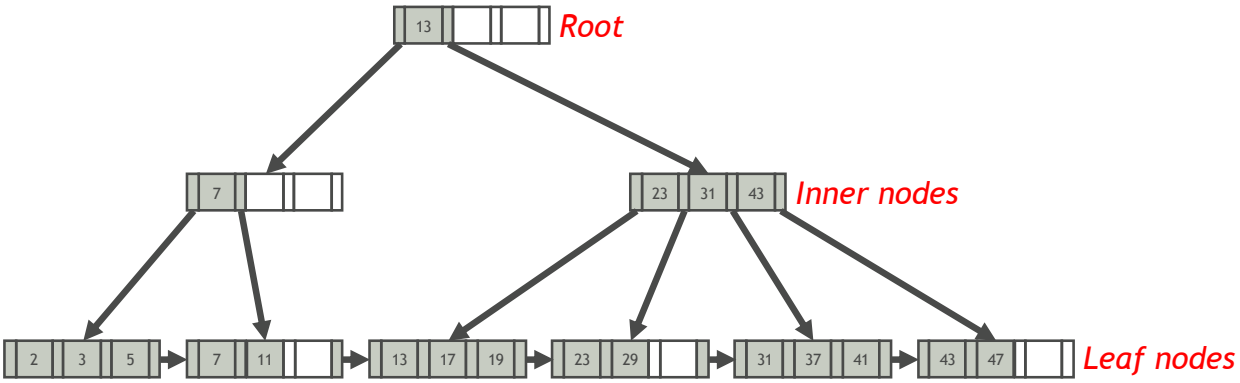
Tree-based Index Structures
B+ Trees

B +树
B +树是具有以下属性的M向搜索树：
它是完美平衡的（即每个叶节点处于相同深度）
除根以外的每个节点至少为半满的M / 2 1ð#keysðM 1
每个具有k个键的内部节点都有k + 1个非空子级
每个节点适合一个页面

B+ Trees

A B+ tree is an *M*

- It is perfectly balanced (i.e., every leaf node is at the same depth)
- Every node other than the root is at least half-full
  $M/2 - 1 \leq \#keys \leq M - 1$
- Every inner node with k keys has $k + 1$ non-null children
- Every node fits a page

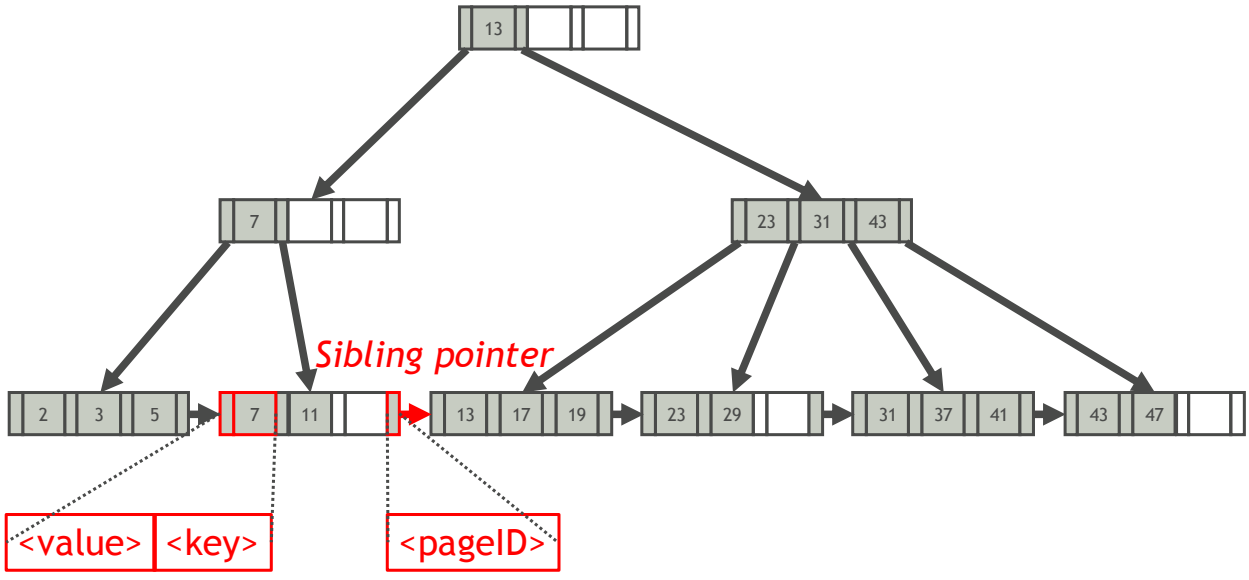


B +树叶节点
每个叶节点都由一个索引条目（键/值对）数组和一个指向其右兄弟的指针组成
索引条目数组（通常）按已排序的键顺序保留

B+

Every leaf node is comprised of an array of index entries (key/value pairs) and a pointer to its right sibling

- The index entry array is (usually) kept in sorted key order

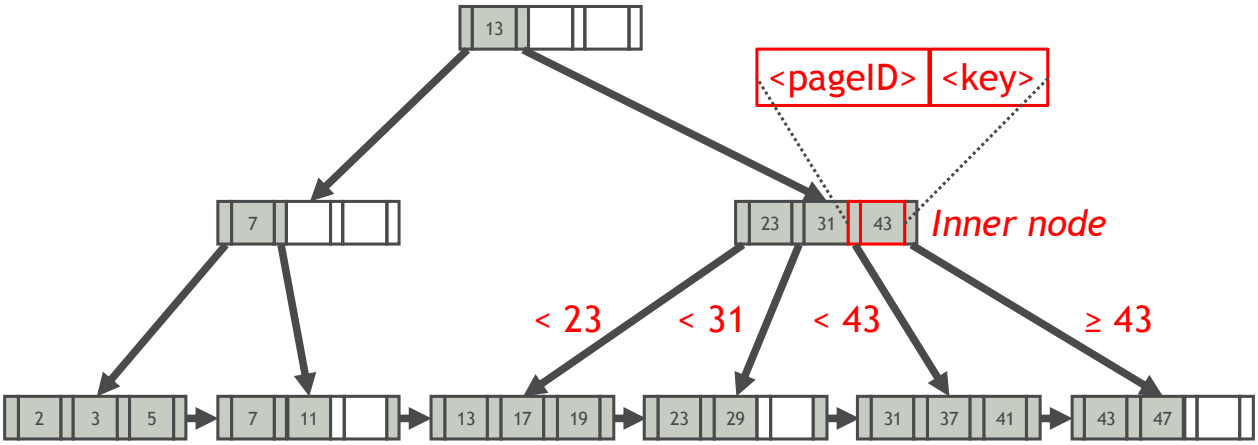


B +树内部节点
每个内部节点都由一个键数组和一个指向其子节点的指针数组组成
键是从索引所基于的属性派生的
数组通常按排序的键顺序保存

B+ Tree Inner Nodes

Every inner node is comprised of an array of keys and an array of pointers to its children

- The keys are derived from the attribute(s) that the index is based on
- The arrays are (usually) kept in sorted key order

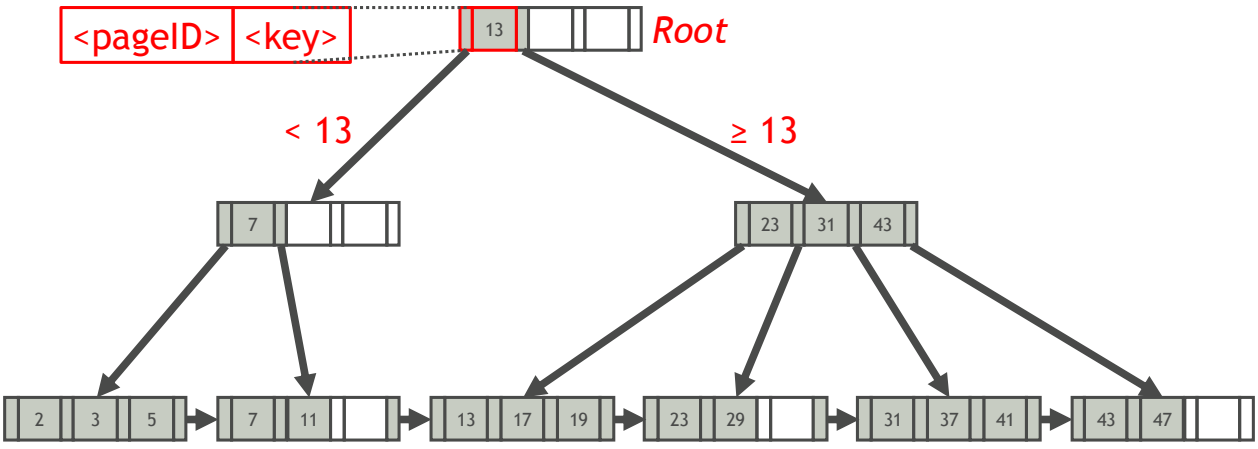


B+ Tree Root Node

B +树根节点

根包含至少一个密钥

The root contains at least one key



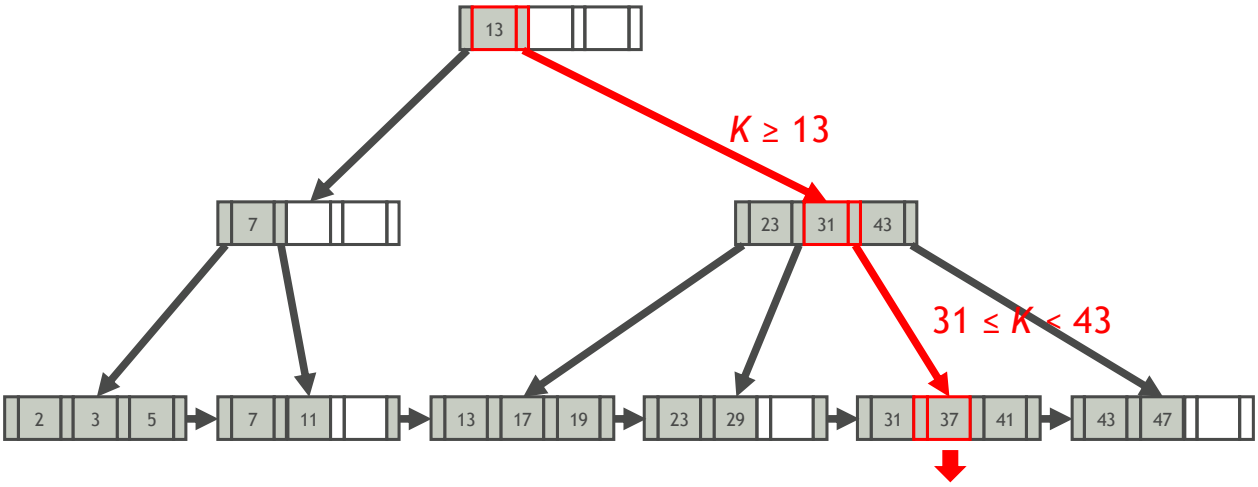
B +树查找
用键K查找索引条目
1通过遵循内部节点中的键的方向来找到K所属的叶节点
2在叶节点中用键K查找条目

B+ Tree Lookup

Find the index entry with key K

1. Find the leaf node where K belongs to by following the direction of the keys in the inner nodes
2. Find the entry with key K in the leaf node

Example: $K = 37$



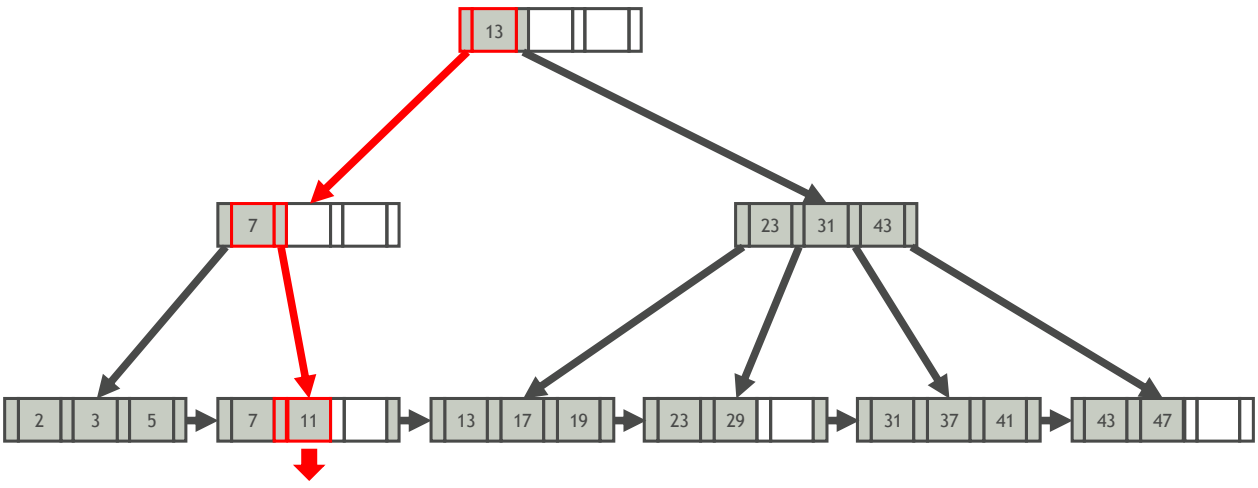
B +树范围查询
用键K 2 [L，U]查找索引条目
1找到具有最小键L的第一个索引条目E
2使用E右侧的δU键扫描连续的索引条目

B+ Tree Range Query

Find the index entries with keys $K \in [L, U]$

1. Find the first index entry E with the smallest key $\geq L$
2. Scan the contiguous index entries with keys $\leq U$ to the right of E

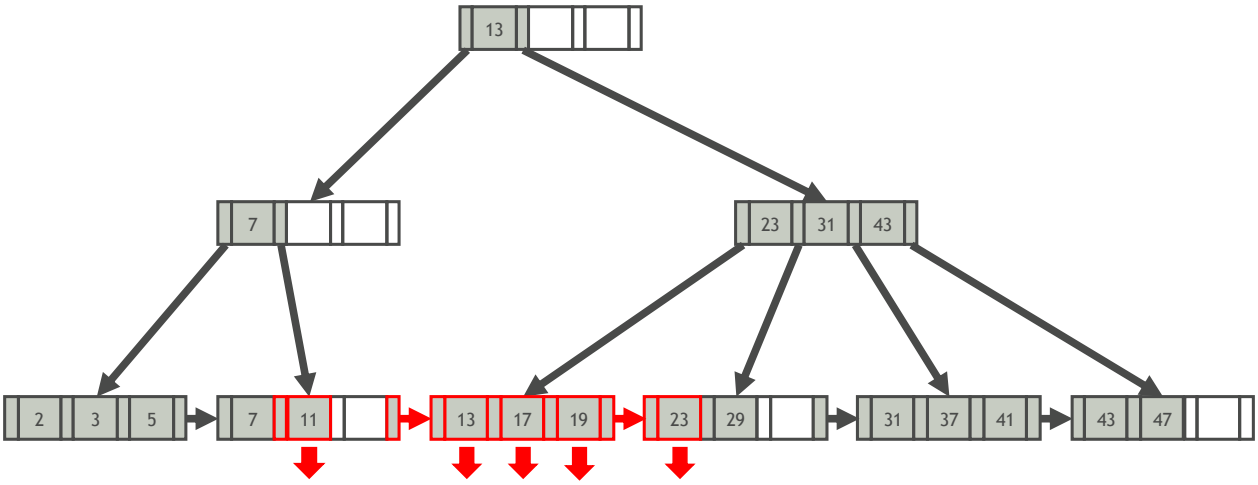
Example: $K \in [10, 25]$



B+ Tree Range Query (Cont'd)

B +树范围查询（续）

Example: $K \in [10, 25]$



B+ Tree Insert

Insert an index entry with key K

1. Find the correct leaf node L where the entry is to be inserted
2. Put the entry into L in sorted key order
3. If L has enough space, done!
   Otherwise, split the keys in L into L and a new node L_2
   1. Redistribute the entries evenly, copy up the middle key
   2. Insert an index entry pointing to L_2 into the parent of L

To split an inner node,

1. Redistribute the entries evenly
2. Push up the middle key

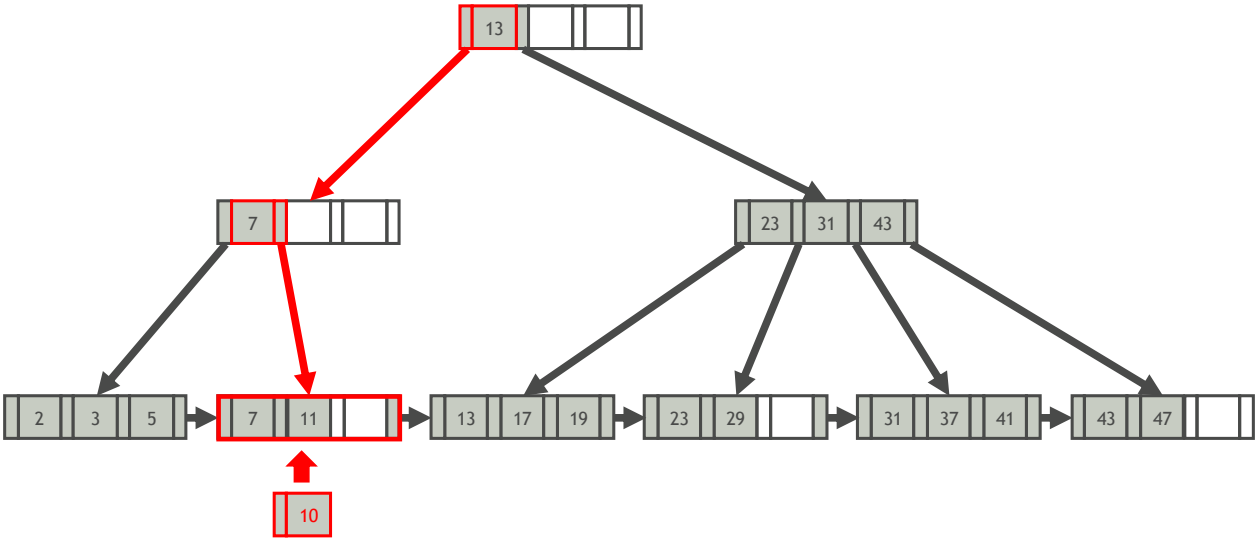
B +树插入

用键K插入索引条目
1找到要在其中插入条目的正确叶节点L
2按键顺序将条目放入L
3如果L有足够的空间，请完成！
否则，将L中的键分为L和新节点L2
1均匀分配条目，复制中间键
2将指向L2的索引条目插入L的父级
要分割内部节点，
1均匀地重新分配条目
2按下中键

B+ Tree Insert: Example 1 (w/o Node Split)

B +树插入：示例1（不带节点拆分）

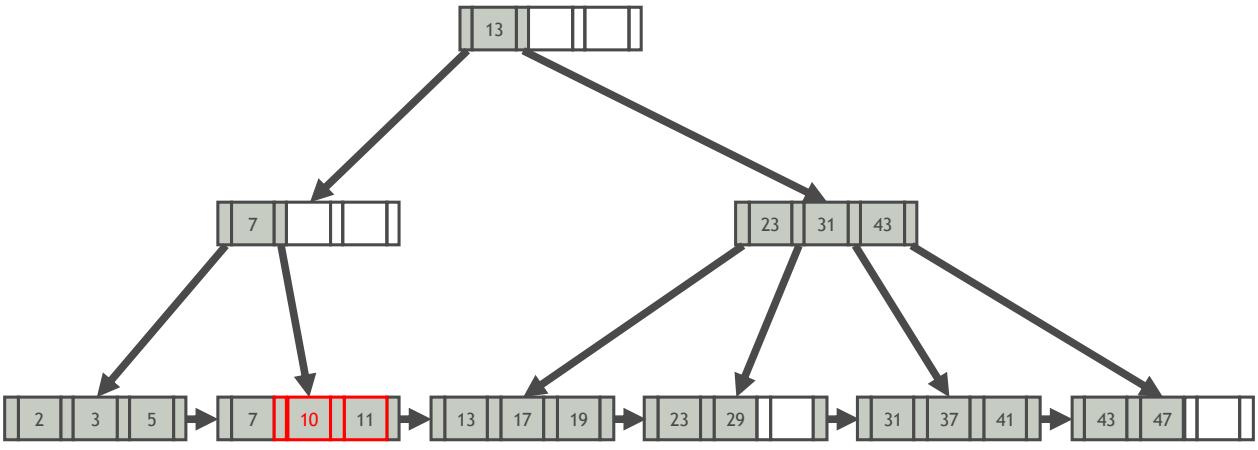
Example: $K = 10$



B +树插入：示例1（不带节点拆分）

B+ Tree Insert: Example 1 (w/o Node Split)

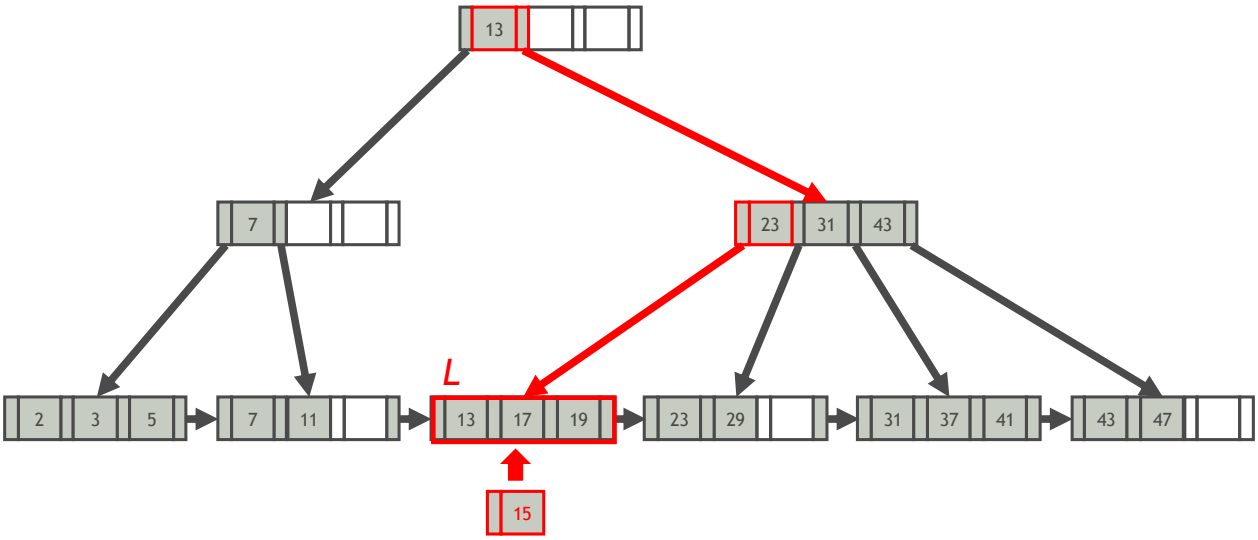
Example: $K = 10$



B +树插入：示例2（带节点拆分）

B+ Tree Insert: Example 2 (w/ Node Split)

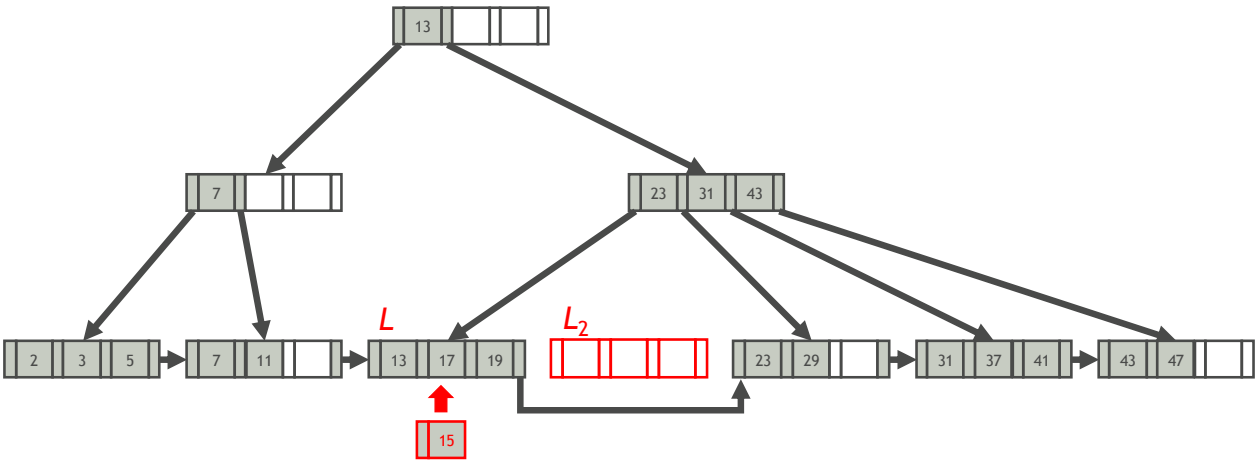
Example: $K = 15$



B +树插入：示例2（带节点拆分）

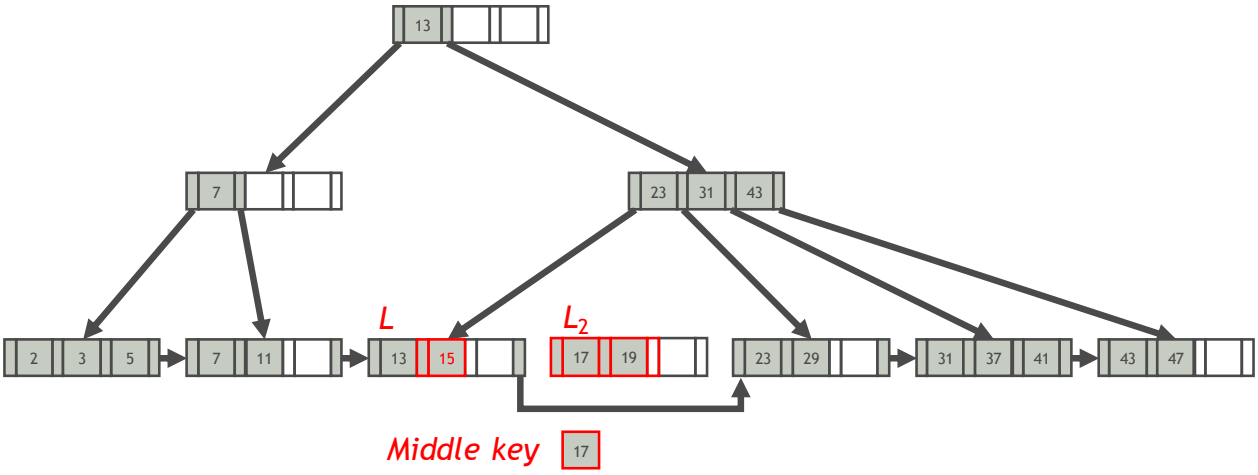
B+ Tree Insert: Example 2 (w/ Node Split)

Example: $K = 15$



B+ Tree Insert: Example 2 (w/ Node Split)

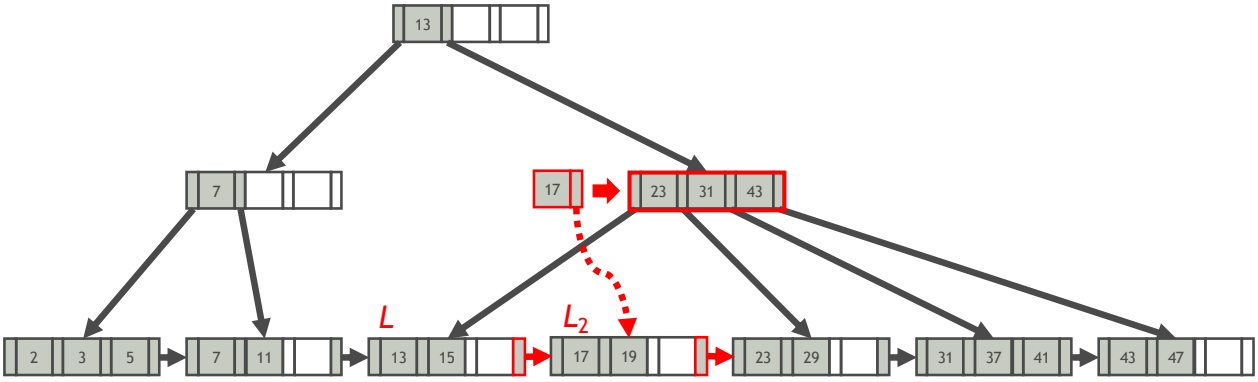
Example: $K = 15$



B +树插入：示例2（带节点拆分）

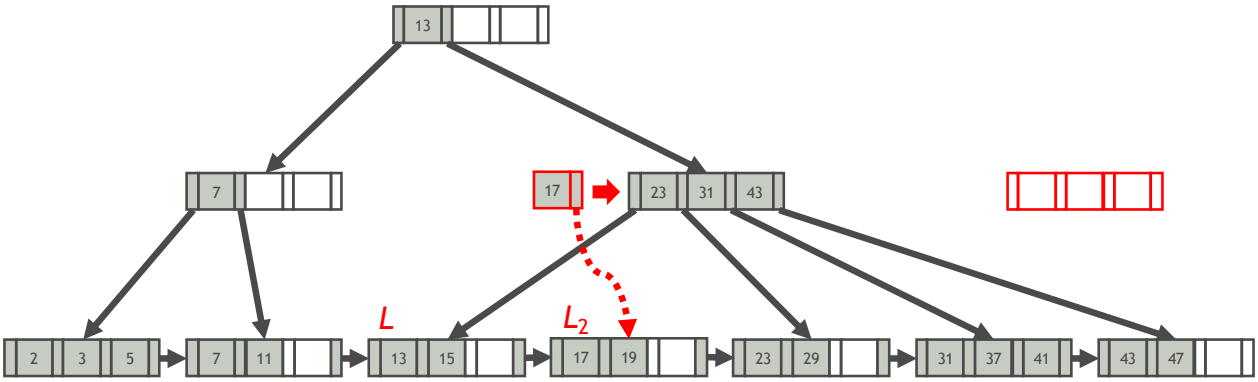
B+ Tree Insert: Example 2 (w/ Node Split)

Example: $K = 15$



B+ Tree Insert: Example 2 (w/ Node Split)

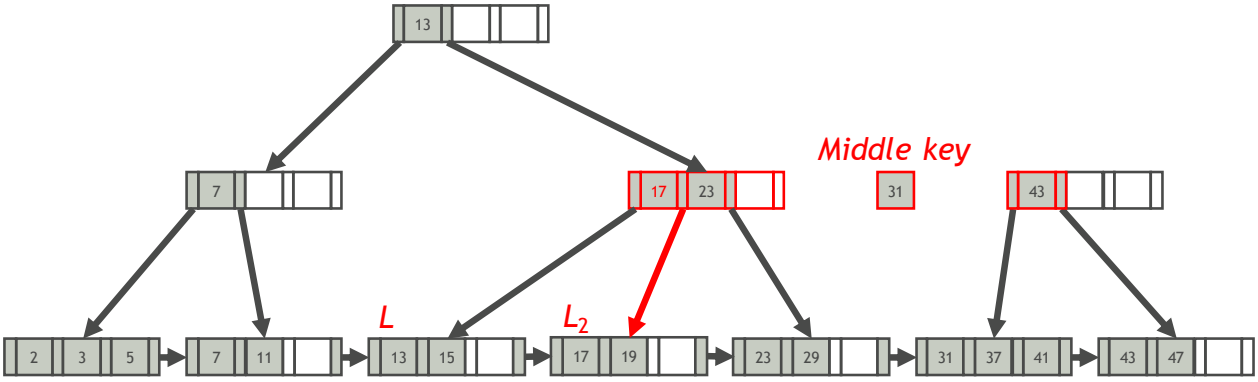
Example: $K = 15$



B +树插入：示例2（带节点拆分）

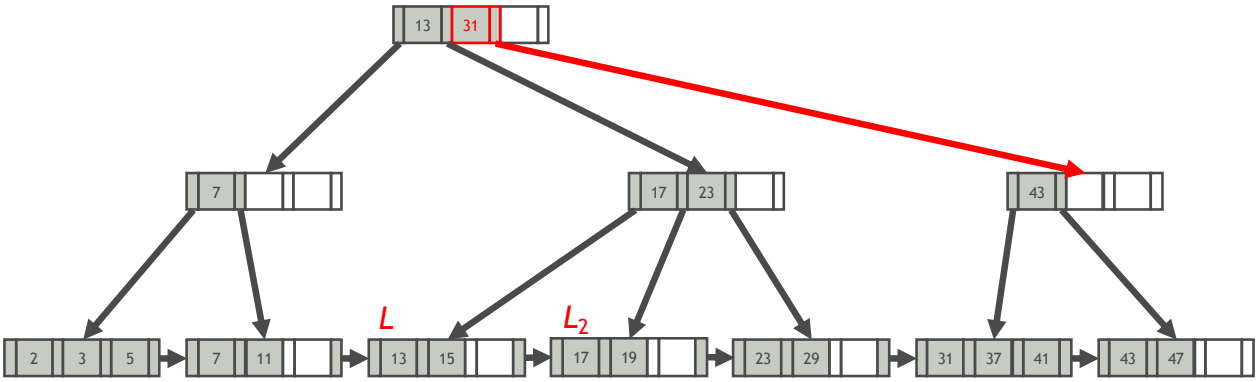
B+ Tree Insert: Example 2 (w/ Node Split)

Example: $K = 15$



B+ Tree Insert: Example 2 (w/ Node Split)

Example: $K = 15$



B +树删除

用键K删除索引条目
1找到条目所属的叶节点L
2从L删除条目
3如果L至少满一半，请完成！
除此以外，
1尝试从兄弟姐妹那里借钱进行再分配
2如果重新分配失败，则合并L及其同级
如果发生合并，则必须从L的父级删除指向L或同级的条目

B+ Tree Delete

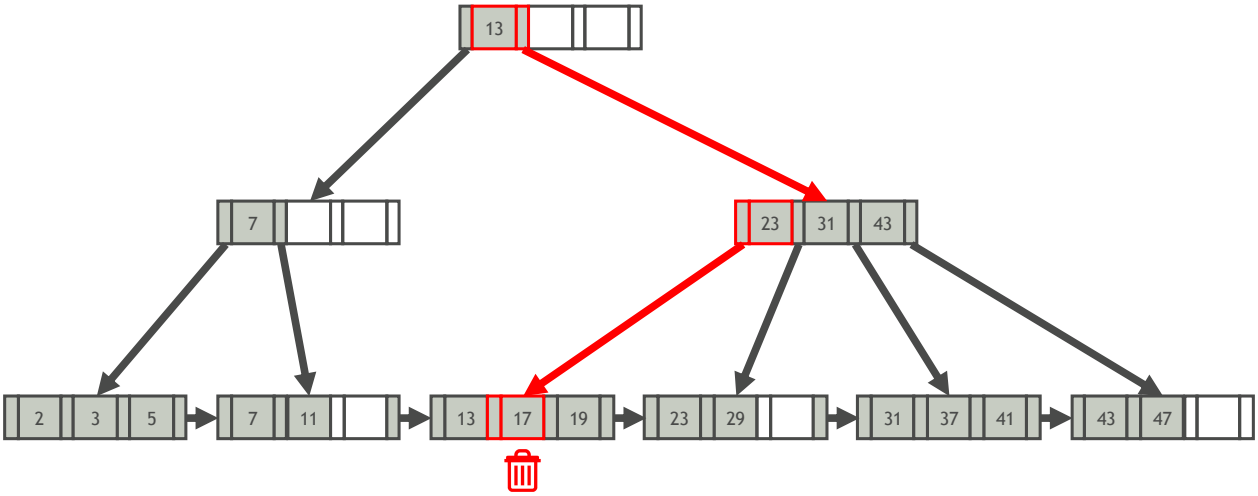
Delete an index entry with key K

1. Find the leaf node L where the entry belongs to
2. Remove the entry from L
3. If L is at least half-full, done!
   Otherwise,
   1. Try to redistribute, borrowing from sibling
   2. If redistribution fails, merge L and its sibling

If merge occurred, must delete entry pointing to L or the sibling from the parent of L

B+ Tree Delete: Example 1 (w/o Node Underflow)

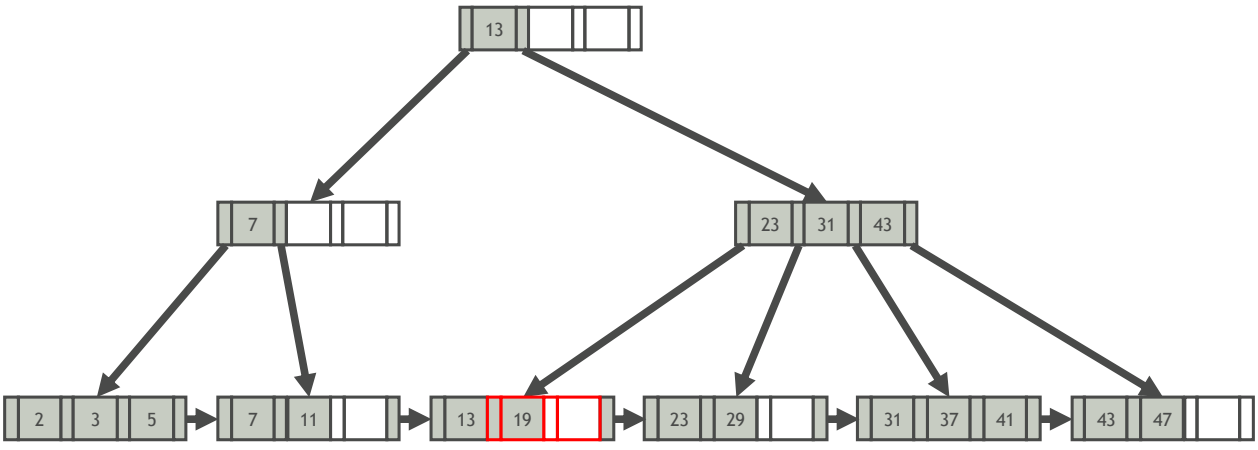
Example: $K = 17$



B +树删除：示例1（无节点下流）

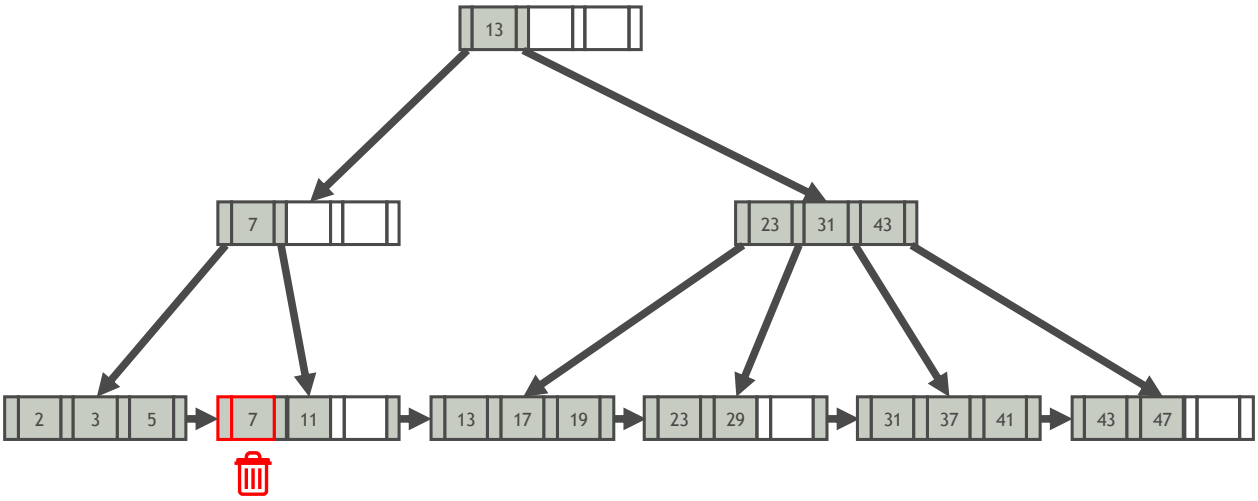
B+ Tree Delete: Example 1 (w/o Node Underflow)

Example: $K = 17$



B+ Tree Delete: Example 2 (Key Redistribution)

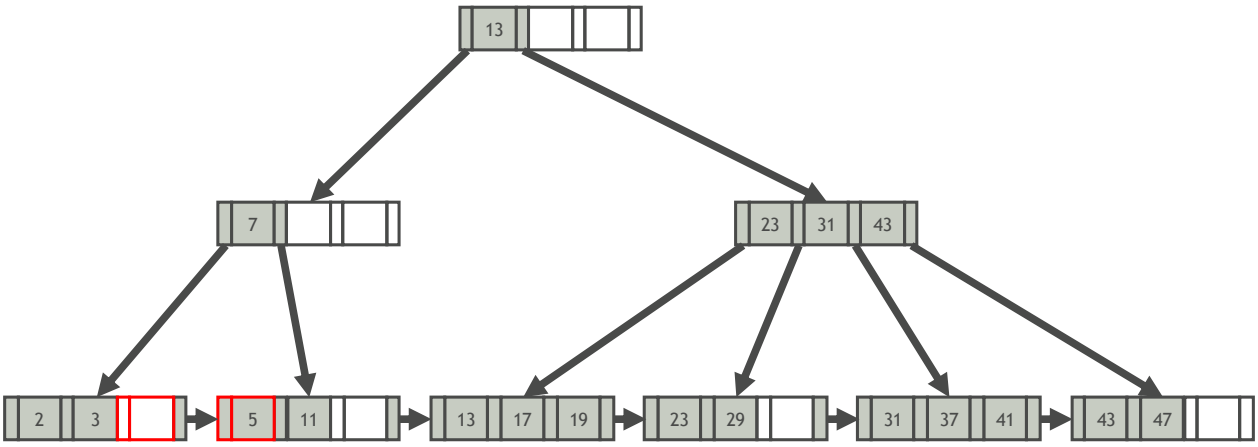
Example: $K = 7$



B +树删除：示例2（密钥重新分发）

B+ Tree Delete: Example 2 (Key Redistribution)

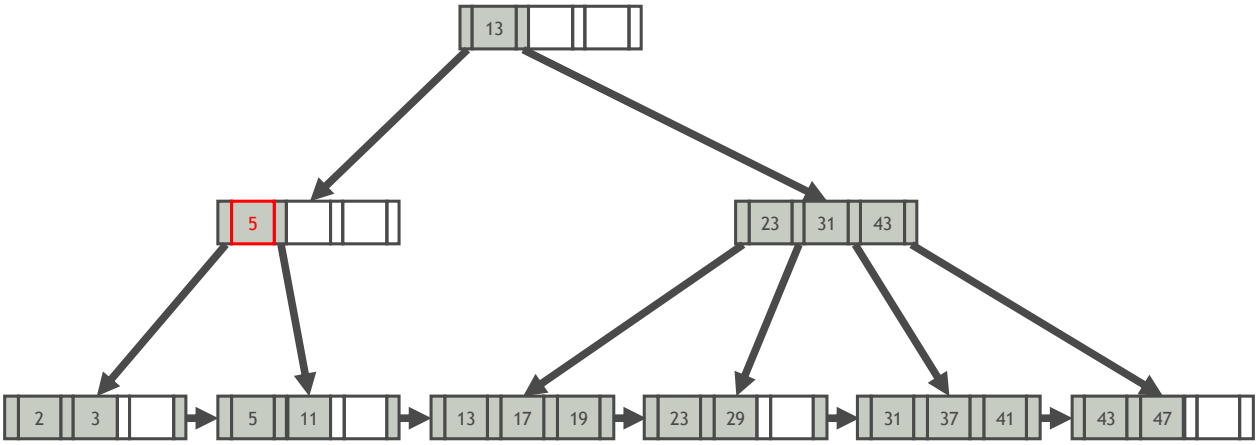
Example: $K = 7$



B +树删除：示例2（密钥重新分发）

B+ Tree Delete: Example 2 (Key Redistribution)

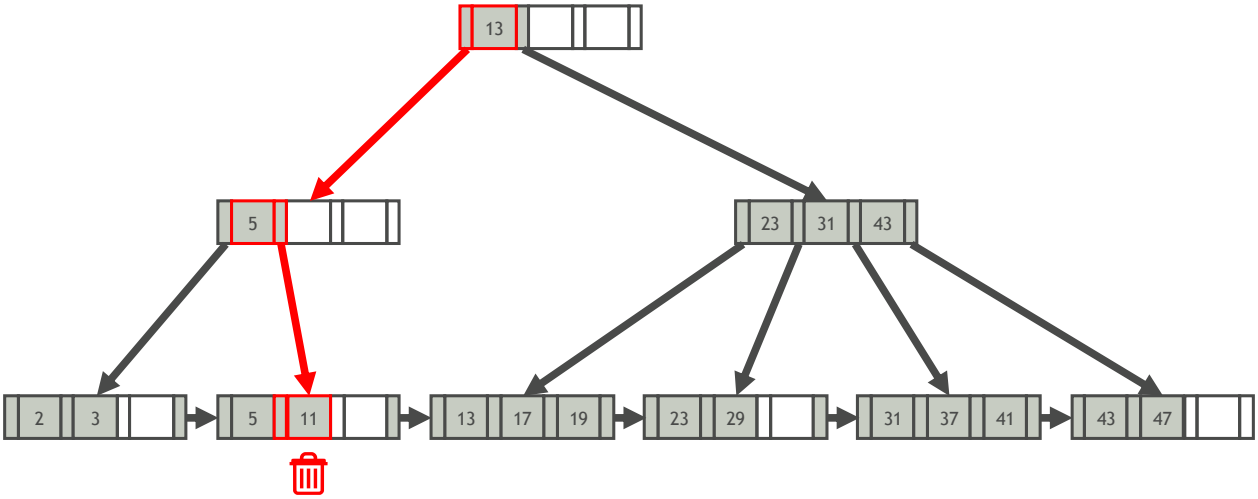
Example: $K = 7$



B +树删除：示例3（密钥重新分发）

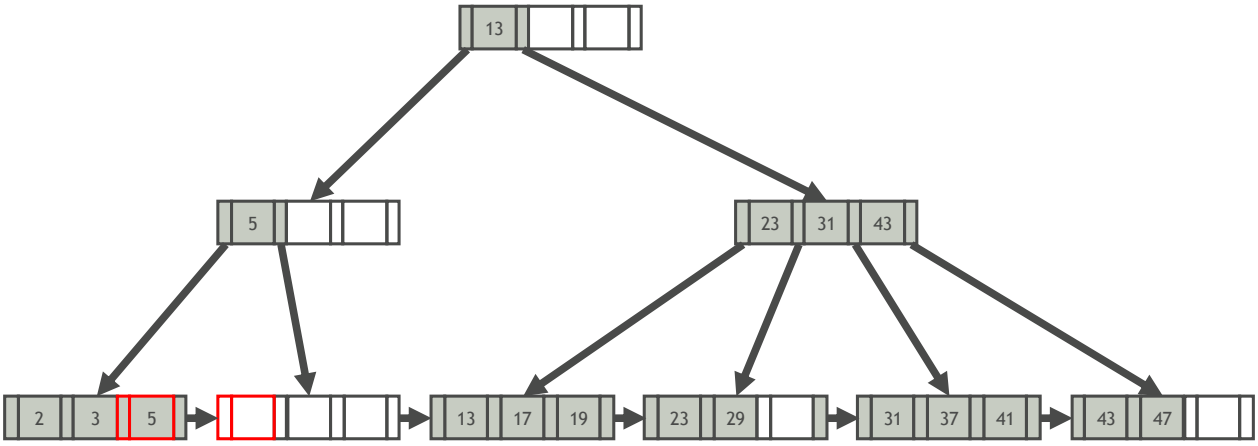
B+ Tree Delete: Example 3 (w/ Node Merge)

Example: $K = 11$



B+ Tree Delete: Example 3 (w/ Node Merge)

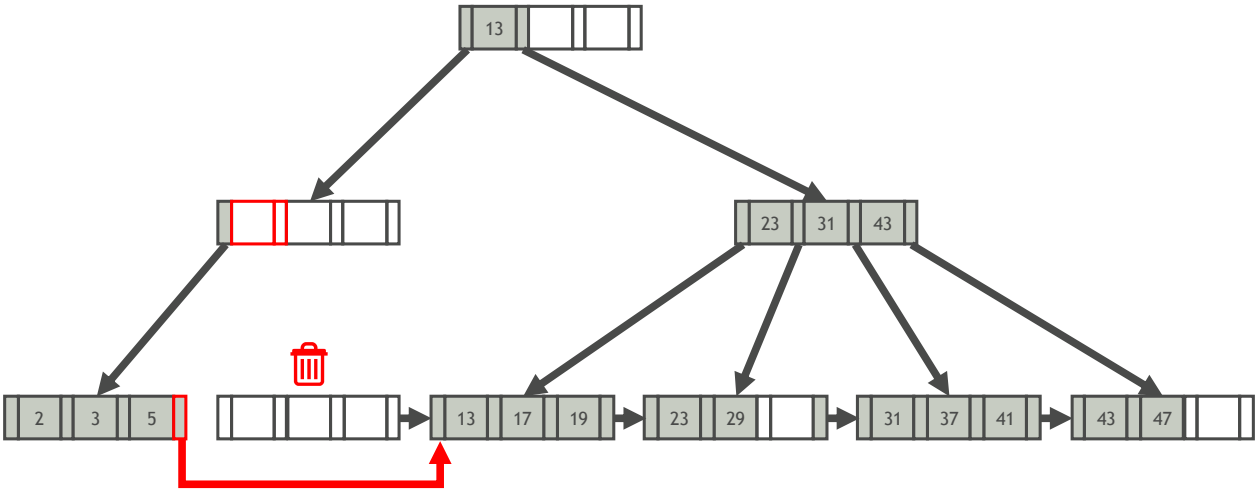
Example: $K = 11$



B +树删除：示例3（密钥重新分发）

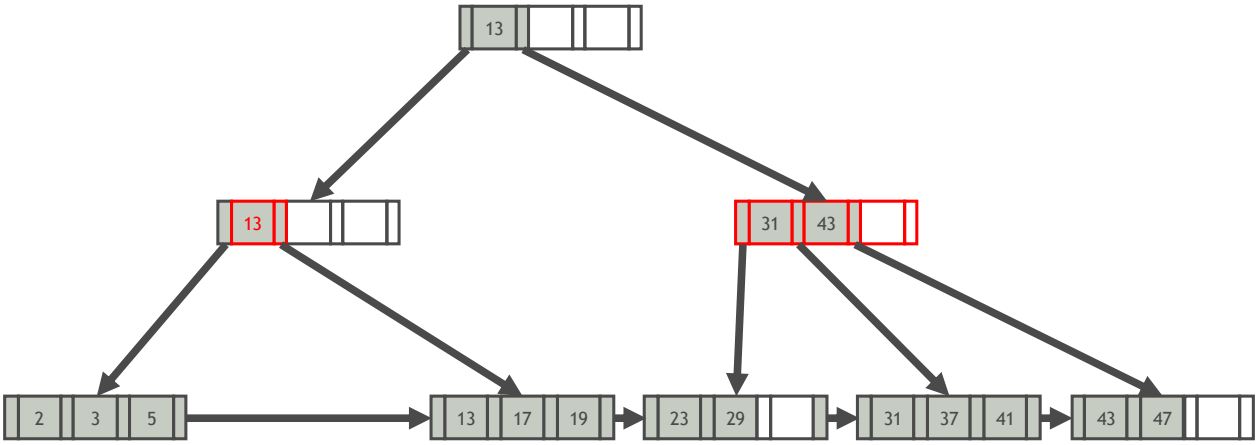
B+ Tree Delete: Example 3 (w/ Node Merge)

Example: $K = 11$



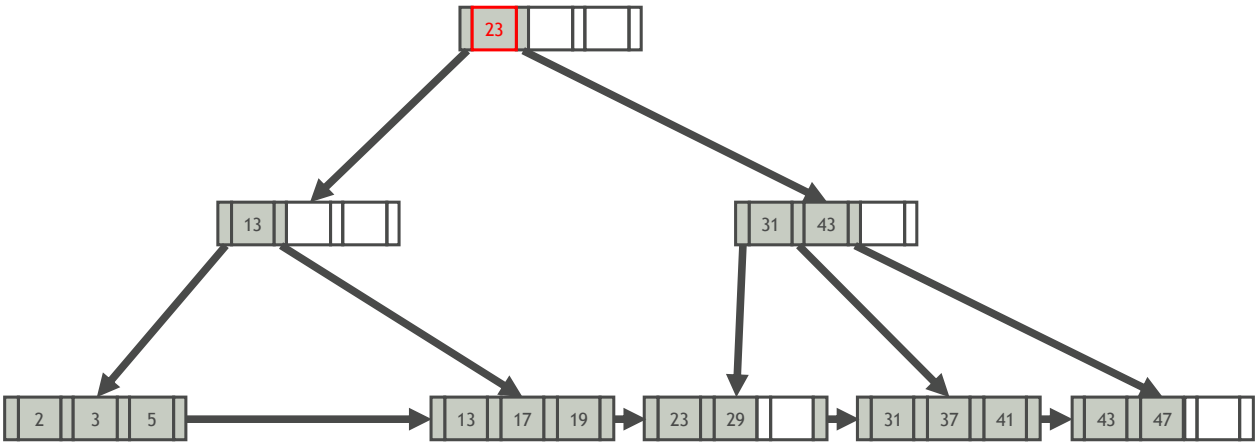
B+ Tree Delete: Example 3 (w/ Node Merge)

Example: $K = 11$



B+ Tree Delete: Example 3 (w/ Node Merge)

Example: $K = 11$



Key Compression

- The number of disk I/Os to retrieve a data entry in a B+ tree = the height of the tree $\approx \log_{fan\_out}$(# of data entries)
- The fan-out (扇出) of the tree is the number of index entries fit on a page, which is determined by the size of index entries
- The size of an index entry depends primarily on the size of the search key value
- Search key values are very long $\Longrightarrow$ the fan-out is low $\Longrightarrow$ the tree is high $\Longrightarrow$ the query time is long

按键压缩

·在B +树中检索数据条目的磁盘I / O数量=树的高度↕logfan out（数据条目数）
·树的扇出（G˙）是页面上的索引条目数，它由索引条目的大小确定
·索引条目的大小主要取决于搜索键值的大小
·搜索键值很长=）扇出度很低=）树很高=）查询时间很长

Prefix Compression (前缀压缩)

·同一叶节点中的已排序键可能具有相同的前缀
·不必每次都存储整个密钥，而是提取公共前缀并仅存储每个密钥的唯一后缀

- Sorted keys in the same leaf node are likely to have the same prefix
- Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key

Microphone	Microsoft	Microwave

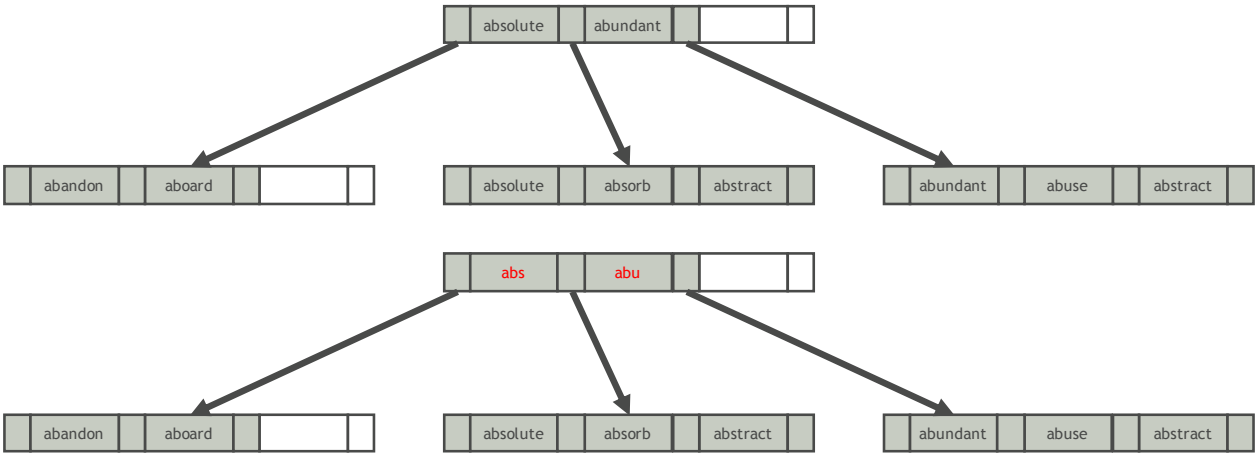
↓ Prefix compression

Prefix: Micro | phone | soft | wave |

·内部节点中的键仅用于引导交易
·我们不需要将密钥全部存储在内部节点中
·存储正确路由探针所需的最小前缀

Suffix Truncation (后缀截断)

- The keys in the inner nodes are only used to direct traffic
- We need not store the keys in their entirety in inner nodes
- Store a minimum prefix that is needed to correctly route probes



在现有索引条目集上创建B +树
自上而下的方法
·一次插入一个索引项
·昂贵，因为每个条目都需要从根开始并向下到适当的叶节点

自下而上的方法
1根据搜索键对索引条目进行排序
2分配一个空的内部节点作为根，并在其中插入指向已排序条目的第一页的指针
3叶子页面的条目总是插入到叶子级别上方最右边的内部节点中。 该页面填满后，将其拆分

Creating a B+ tree on an existing set of index entries

Top-Down Approach

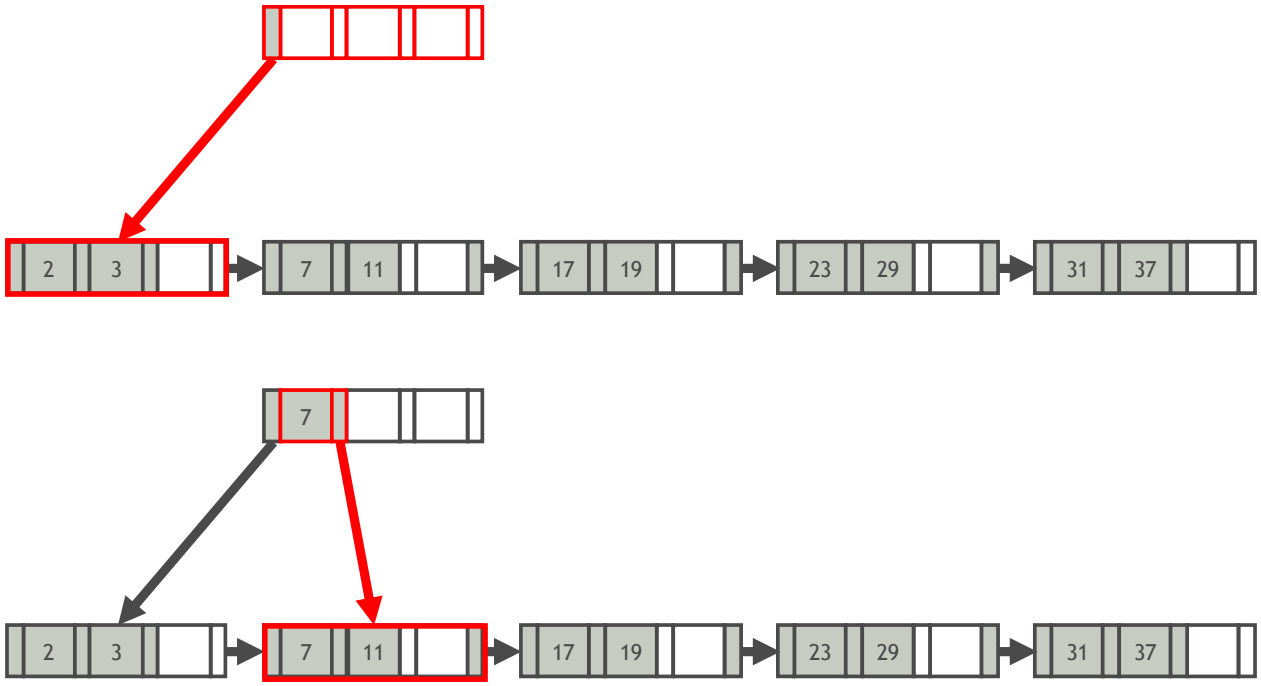
- Insert the index entries one at a time
- Expensive, because each entry requires to start from the root and go down to the appropriate leaf node

Bottom-Up Approach

1. Sort the index entries according to the search key
2. Allocate an empty inner node as the root and insert a pointer to the first page of sorted entries into it
3. Entries for the leaf pages are always inserted into the right-most inner node just above the leaf level. When that page fills up, it is split

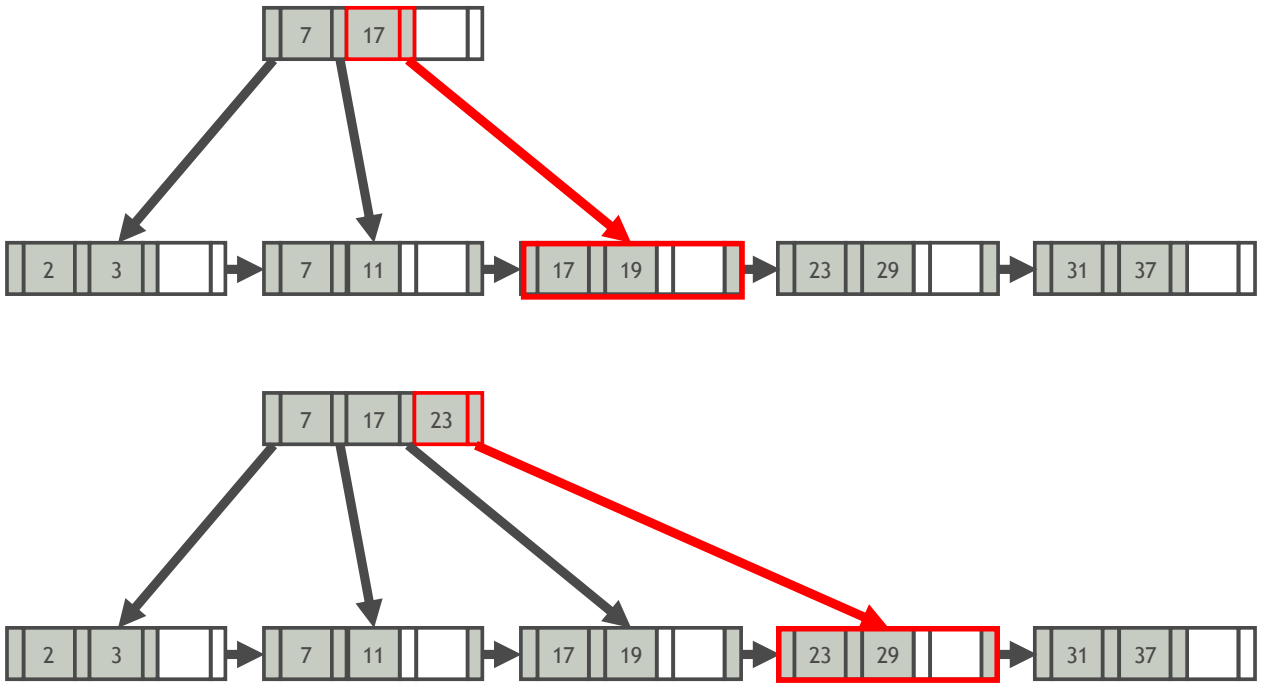
Bulk Loading: Example

Sorted keys: 2, 3, 7, 11, 17, 19, 23, 29, 31, 37



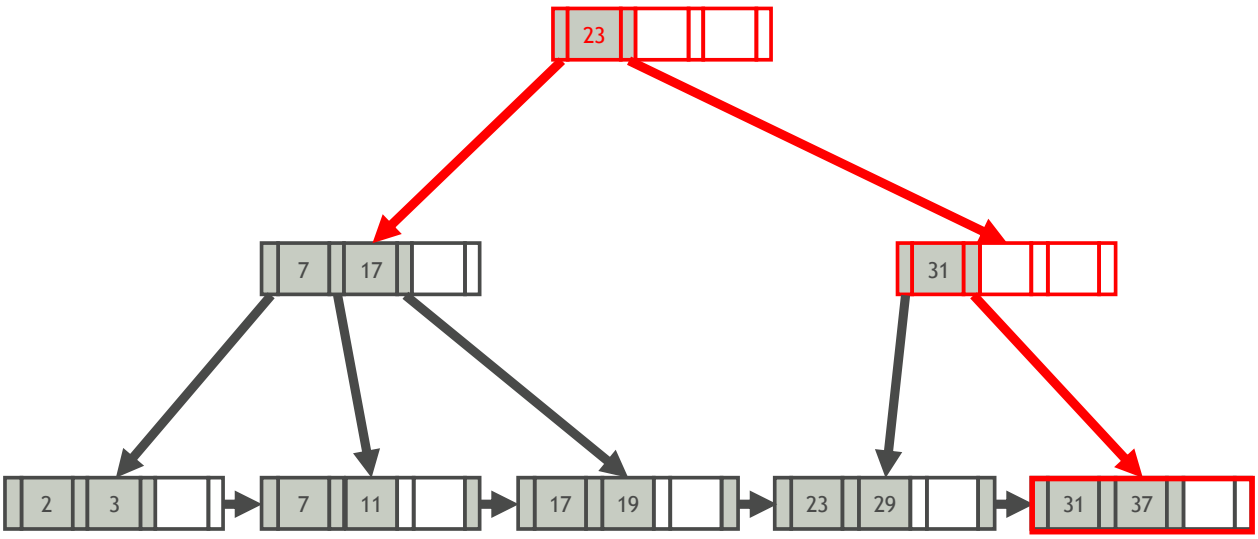
Bulk Loading: Example

Sorted keys: 2, 3, 7, 11, 17, 19, 23, 29, 31, 37



Bulk Loading: Example

Sorted keys: 2, 3, 7, 11, 17, 19, 23, 29, 31, 37



Summary

摘要

1基于哈希的索引结构
可扩展哈希表
线性哈希表

2基于树的索引结构
B +树

1. Hash-based Index Structures
   - Extensible Hash Tables
   - Linear Hash Tables
2. Tree-based Index Structures
   - B+ Trees

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。