第8章：索引结构

Index Structures

邹兆年

哈尔滨工业大学
计算机科学与技术学院
海量数据计算研究中心
电子邮件：znzou@hit.edu.cn

2021年春

Outline[1]

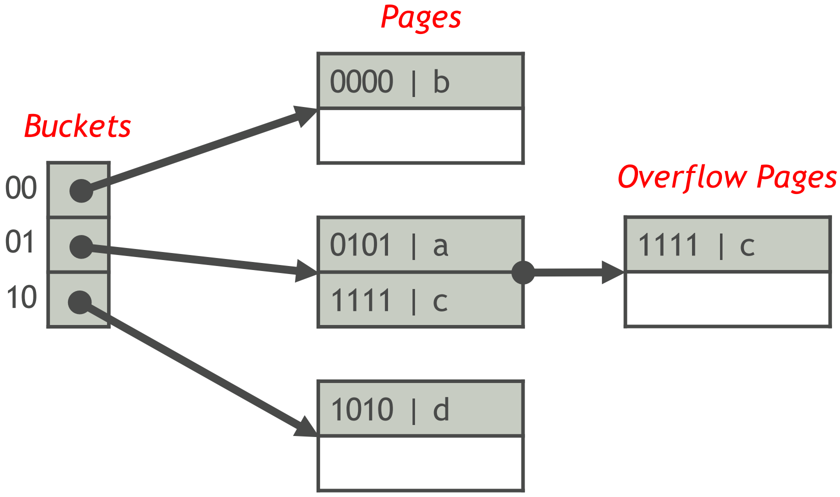
1. Hash-based Index Structures
   - Extensible Hash Tables
   - Linear Hash Tables
2. Tree-based Index Structures
   - B+ Trees
3. Log-Structured Merge-Trees (LSM-Trees)

[1]Updated on April 12, 2021

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



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)

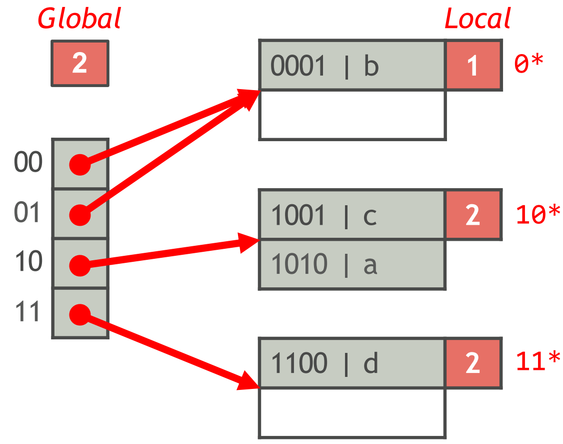
Hash-based Index Structures
Extensible Hash Tables

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 $hash(K)$

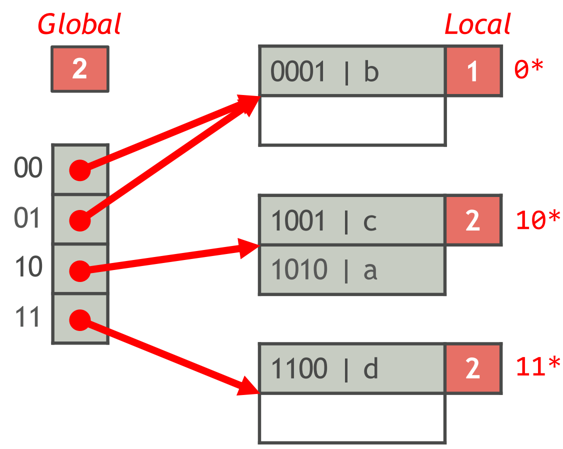
Example:
$hash(a) = 1010, hash(b) = 0001, hash(c) = 1001, hash(d) = 1100$



Extensible Hash Tables (Cont'd)

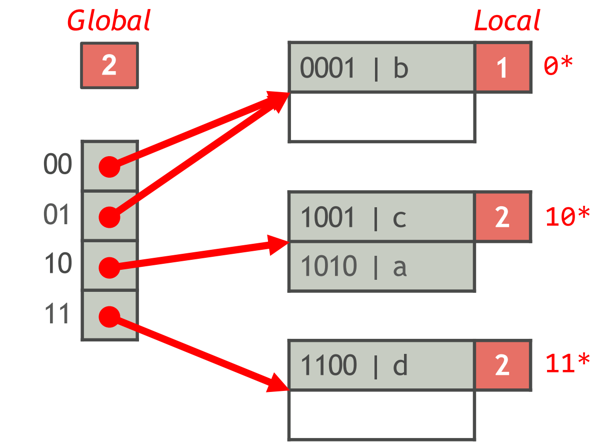
Every bucket keeps a pointer to a page where the index entries in the bucket are stored

- 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



Properties of Extensible Hash Tables

- $\#\text{buckets} = 2^{\text{global\_depth}}$
- The global depth must be greater than or equal to the local depth of any page
- The page that a bucket points to is shared by another bucket if and only if the local depth of the page is less than the global depth

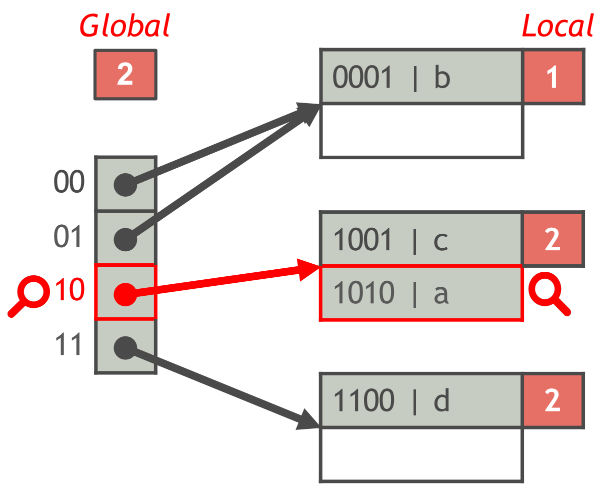


Extensible Hash Table Lookup

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

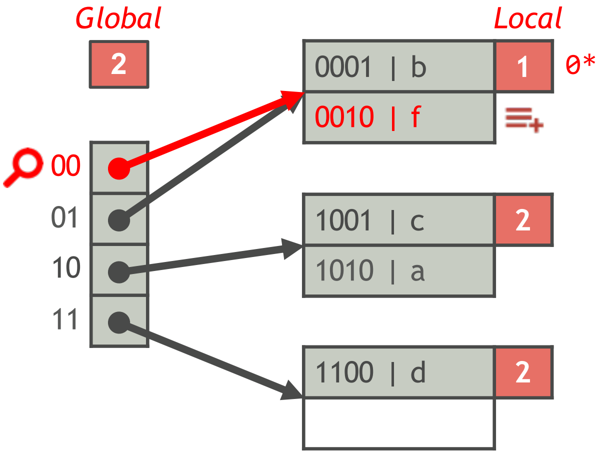


Extensible Hash Table Insert

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

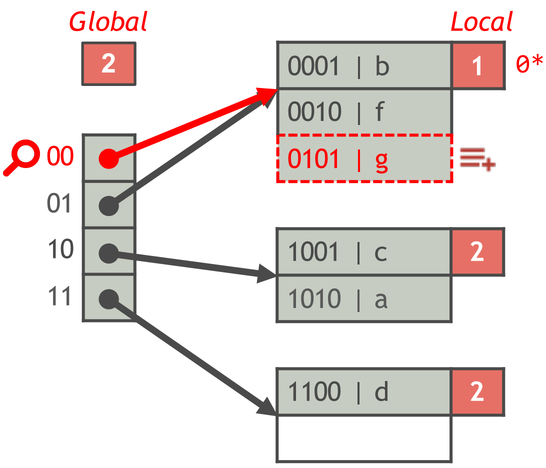


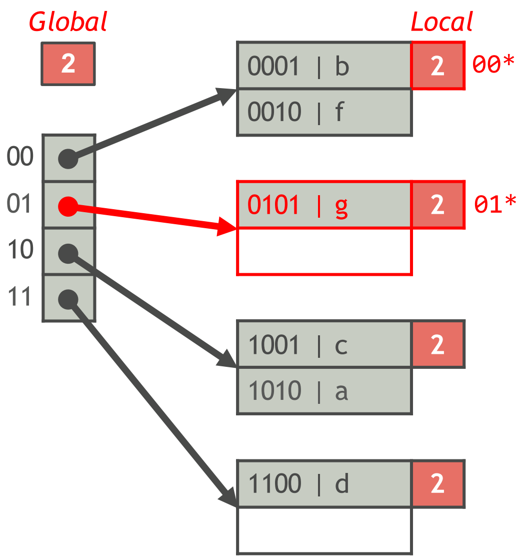
Extensible Hash Table Insert (Cont'd)

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



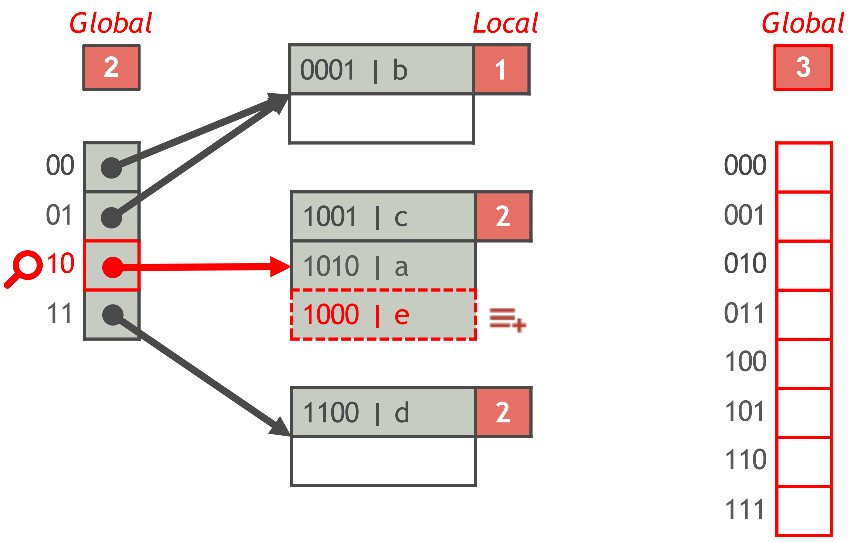


Extensible Hash Table Insert (Cont'd)

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

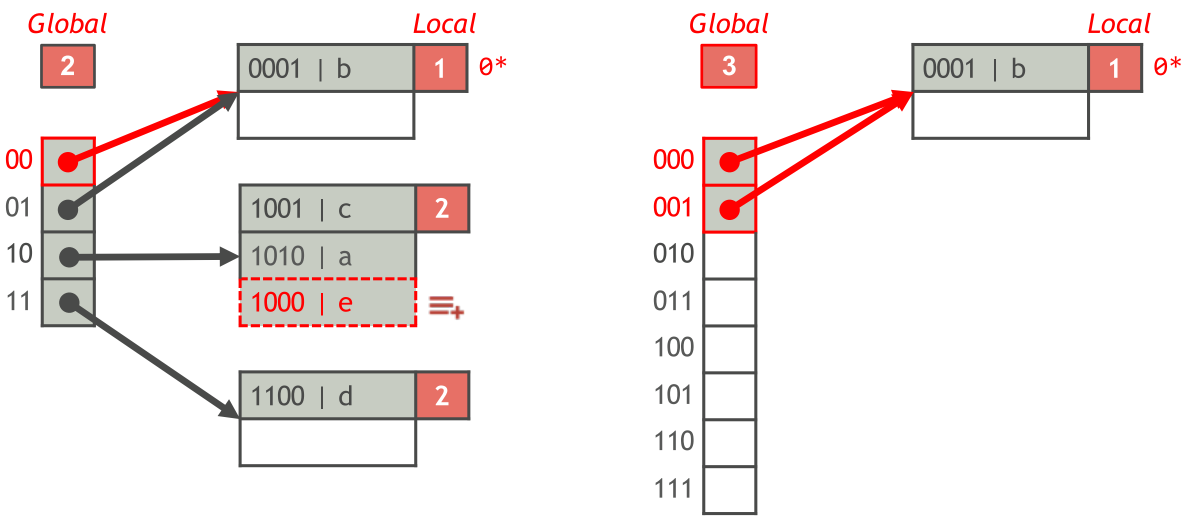
1. Increase the global depth by 1 (double # buckets)
2. Re-organize the buckets; if a page overflows, split it

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



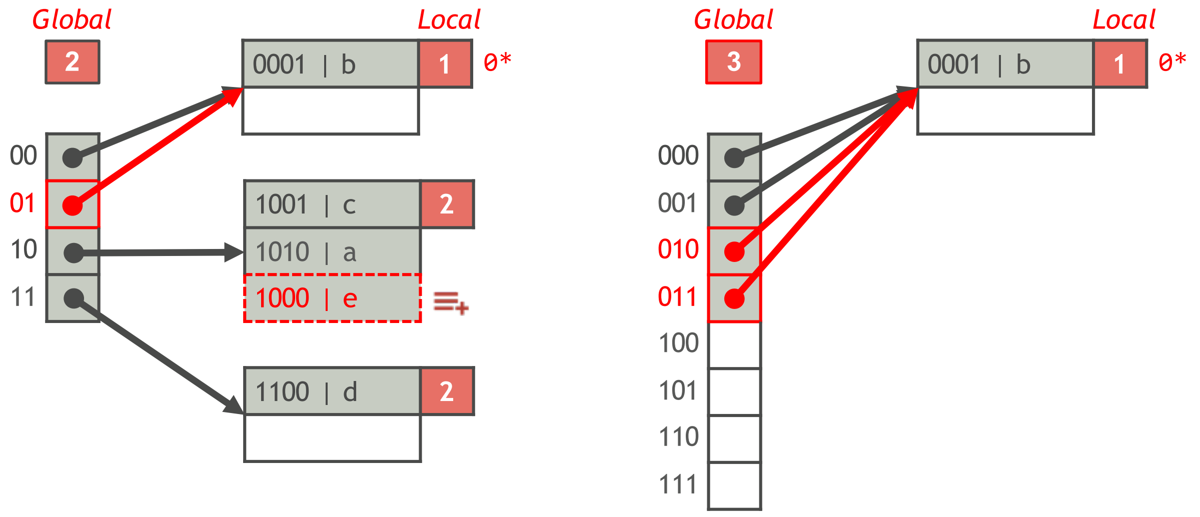
Extensible Hash Table Insert: Example

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



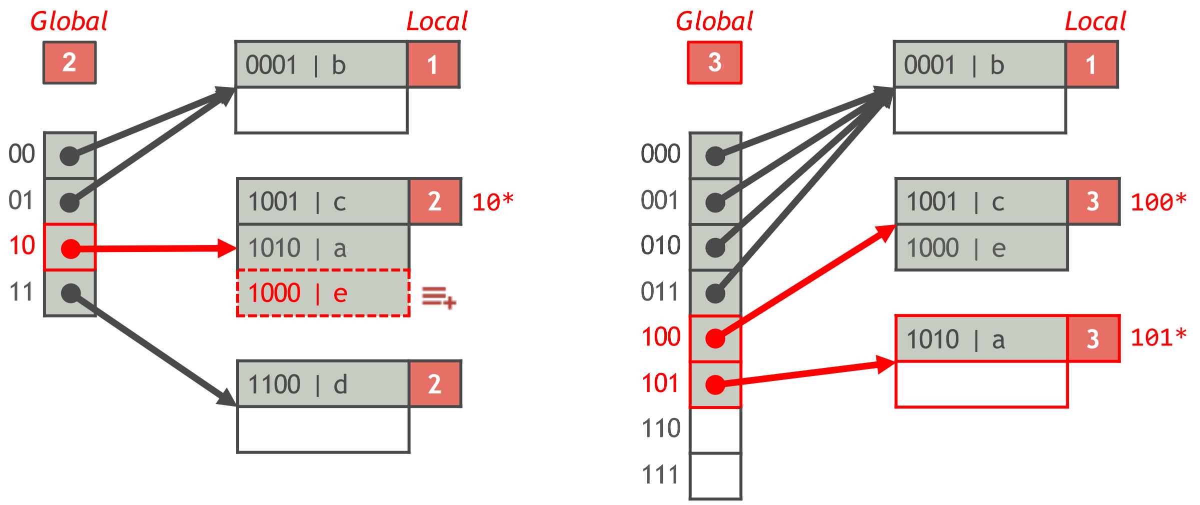
Extensible Hash Table Insert: Example

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



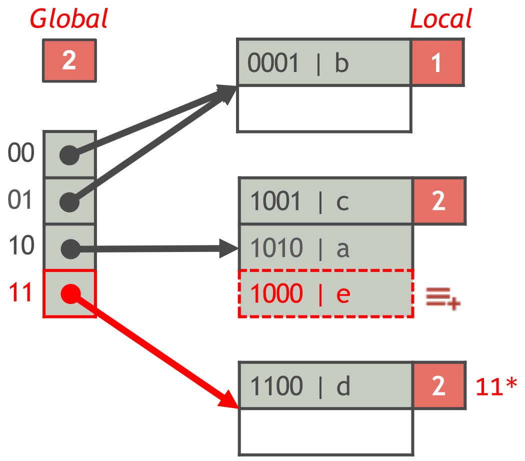
Extensible Hash Table Insert: Example

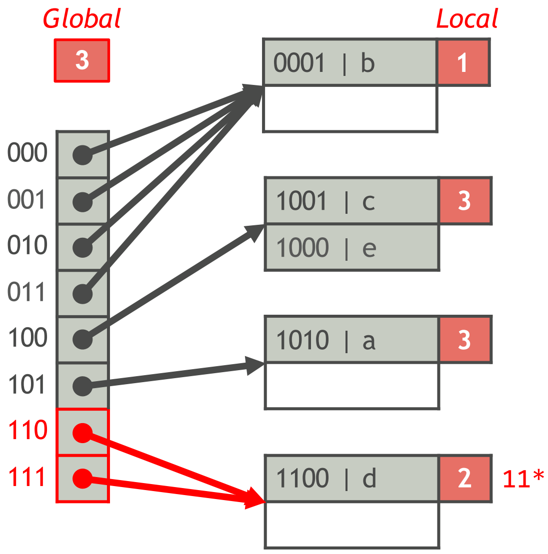
Example: $K = e$, $hash(e) = 1000$



Extensible Hash Table Insert: Example

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



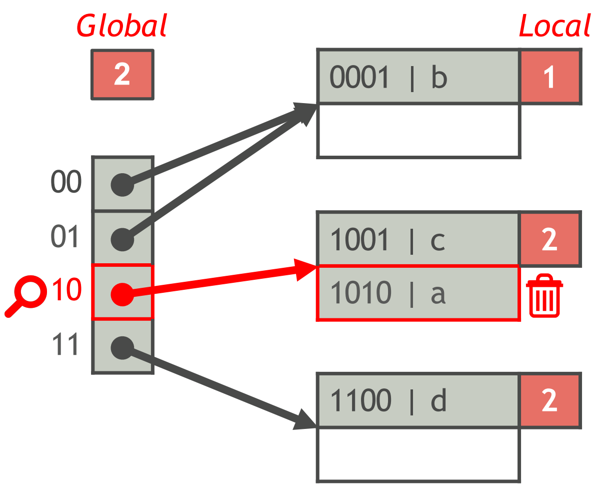


Extensible Hash Table Delete

Delete the index entry with key K

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

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



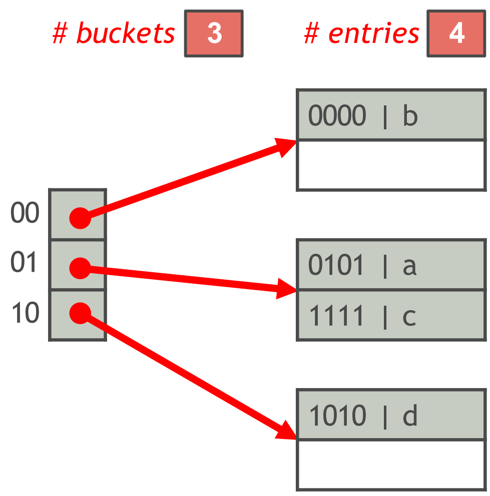
Hash-based Index Structures
Linear Hash Tables

Linear Hash Tables (线性哈希表)

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



Hashing Scheme

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

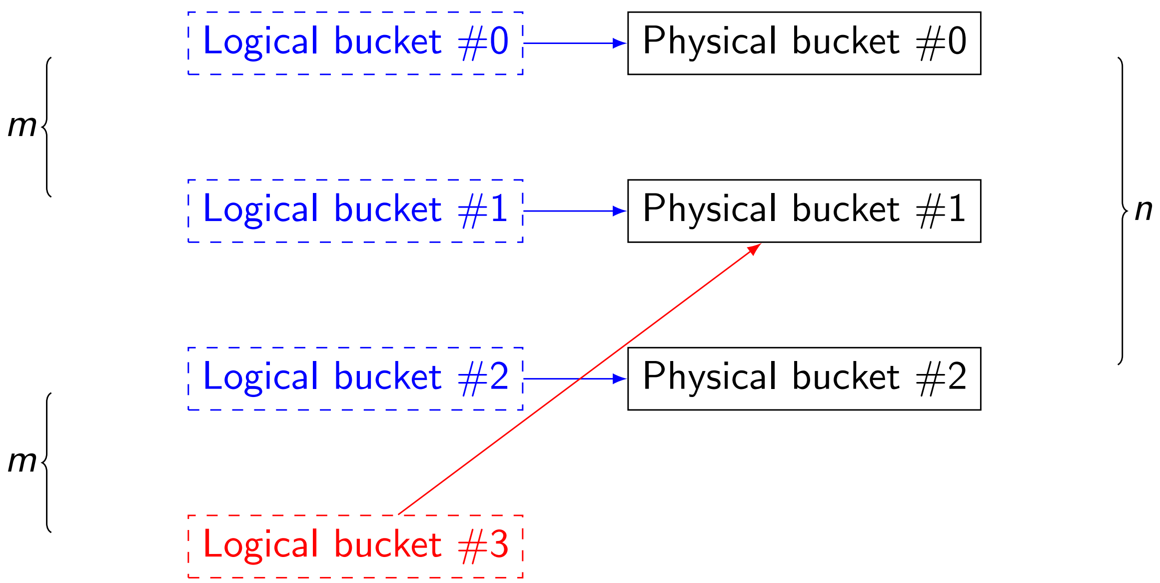
Example: Let $n = 3$. We have $m = 2$ because $2 = m \le n < 2m = 4$

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

The buckets are NOT load-balanced

Hashing Scheme (Cont'd)

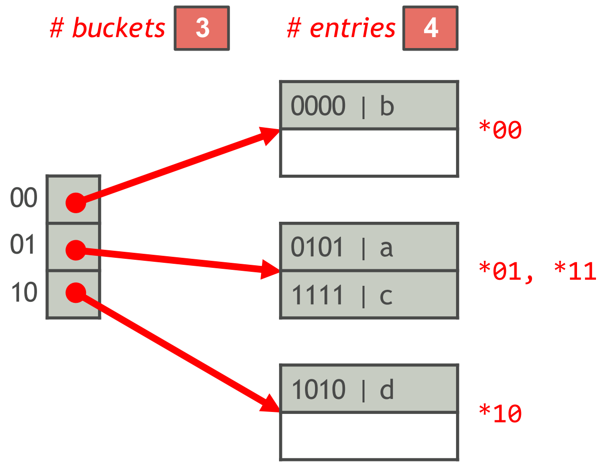
- The logical bucket number $b(K)$ for key K is $hash(K) \mod 2m$
- The physical bucket number for key K is $b(K)$ if $b(K) < n$
- The physical bucket number for key K is $b(K) \mod m$ if $b(K) \ge n$



Hashing Scheme (Cont'd)

- The buckets are numbered from 0 to $n-1$
- Let $m = 2^{\lfloor \log_2 n \rfloor}$, so $m \le n < 2m$
- 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:
$hash(a) = 0101, hash(b) = 0000, hash(c) = 1111, hash(d) = 1010$

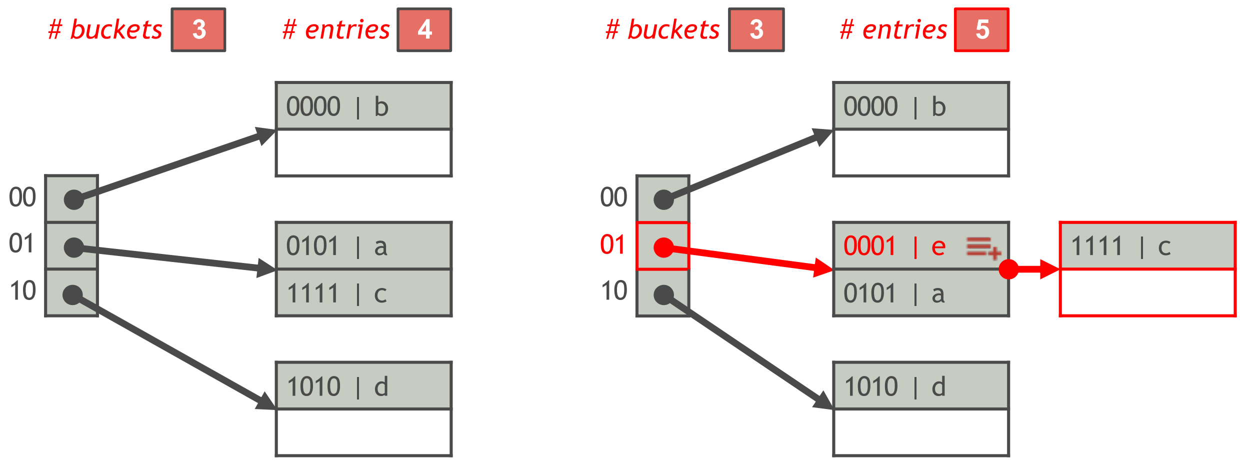


Linear Hash Table Insert

Insert an index entry with key K

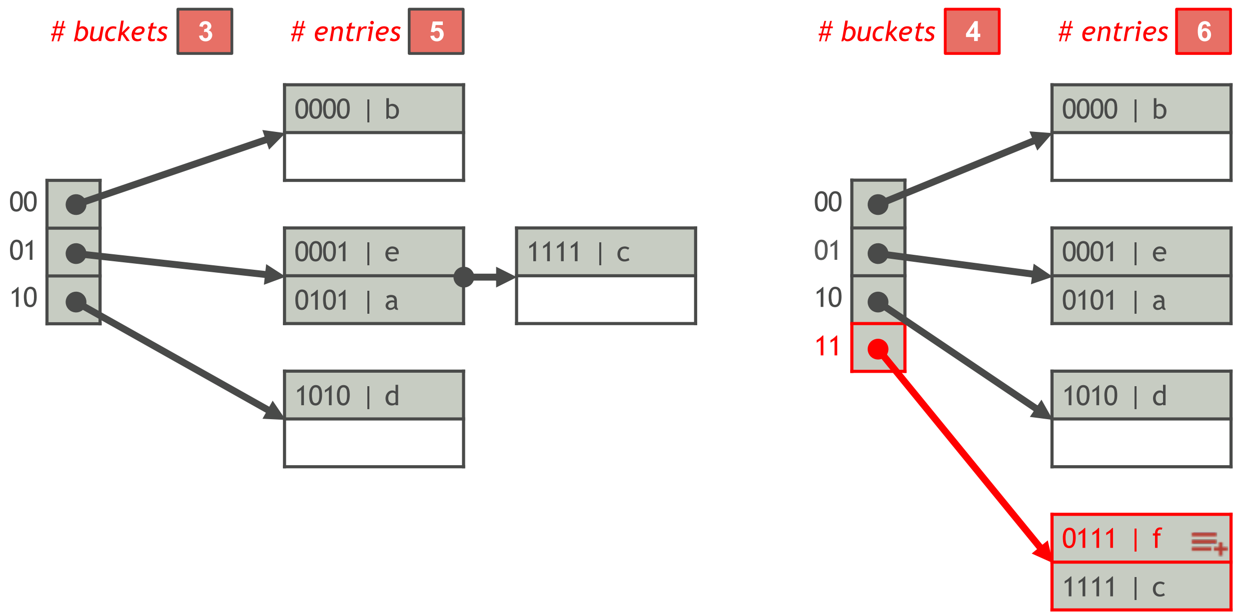
1. Insert the entry into the bucket B where it belongs to
2. Increase # entries by 1
3. If # entries $\le \theta bn$, done!
   Otherwise, increase # buckets by 1 and redistribute the entries in B

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



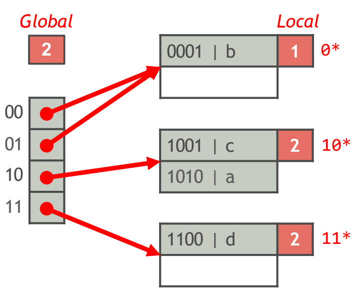
Linear Hash Table Insert (Cont'd)

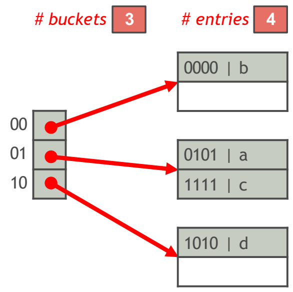
Example: $hash(f) = 0111$, $\theta = 0.85$



Extensible Hash Tables VS Linear Hash Tables

	Extensible hash tables	Linear hash tables
# Buckets	2^{global_depth}	n
Bucket pages	A bucket points to a single page	A bucket points to a linked list of pages
Hashing scheme	The first *global_depth* fits of $hash(K)$	$hash(K)$ mod $2m$ or $hash(K)$ mod m
Page split condition	A page overflows	$\#entries > \theta bn$
Hash table expansion	# buckets is doubled (*global_depth* is increased by 1)	# buckets is increased by 1





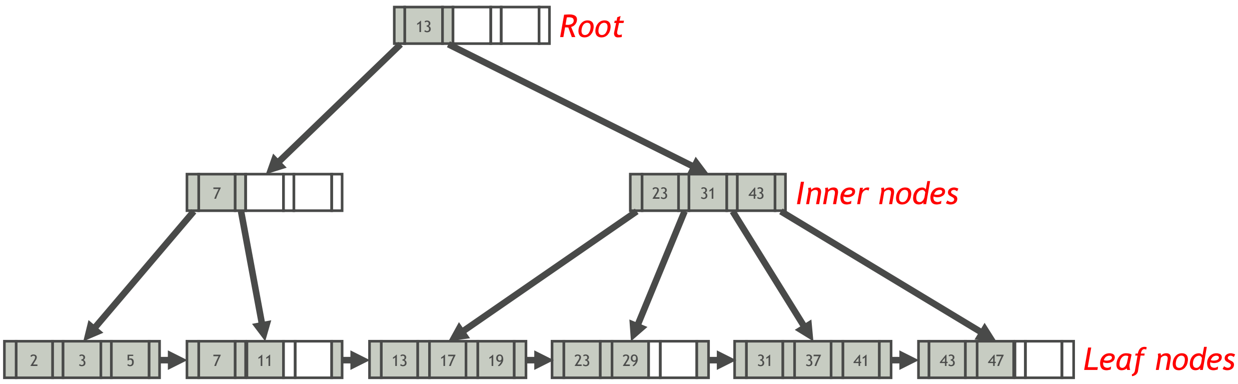
Tree-based Index Structures

Tree-based Index Structures
B+ Trees

B+ Trees

A B+ tree is an M-way search tree with the following properties:

- 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
  $M/2 - 1 \le \#keys \le M - 1$[2]
- Every inner node with k keys has $k+1$ non-null children
- Every node fits a page

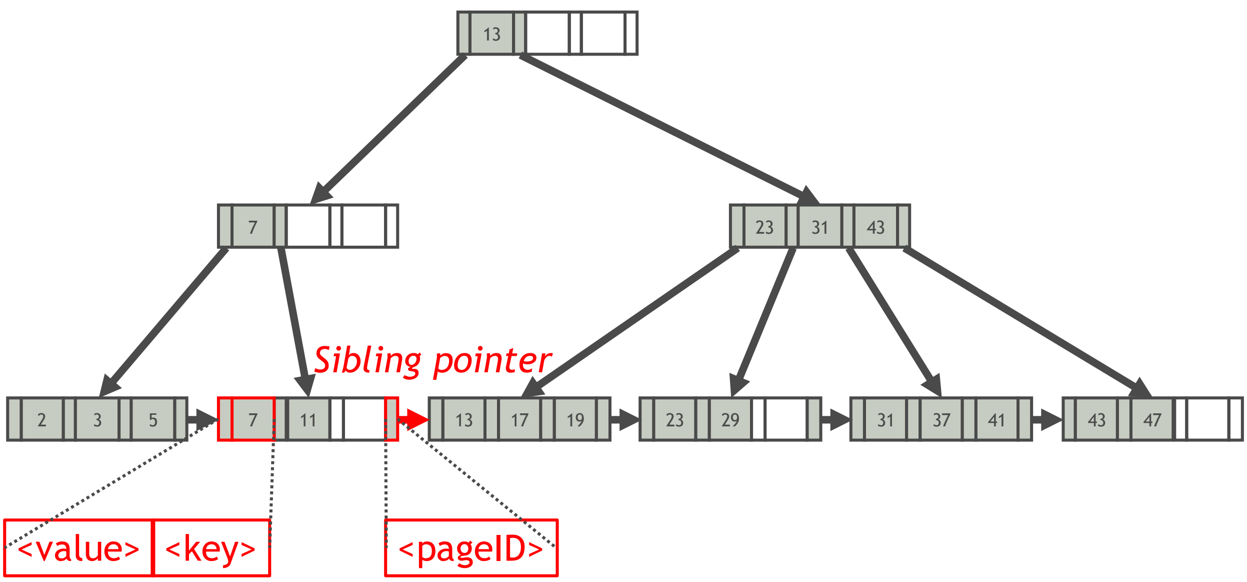


[2] Raghu Ramakrishnan, Johannes Gehrke. Database Management Systems, 3rd Edition. 2003.

B+ Tree Leaf Nodes

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

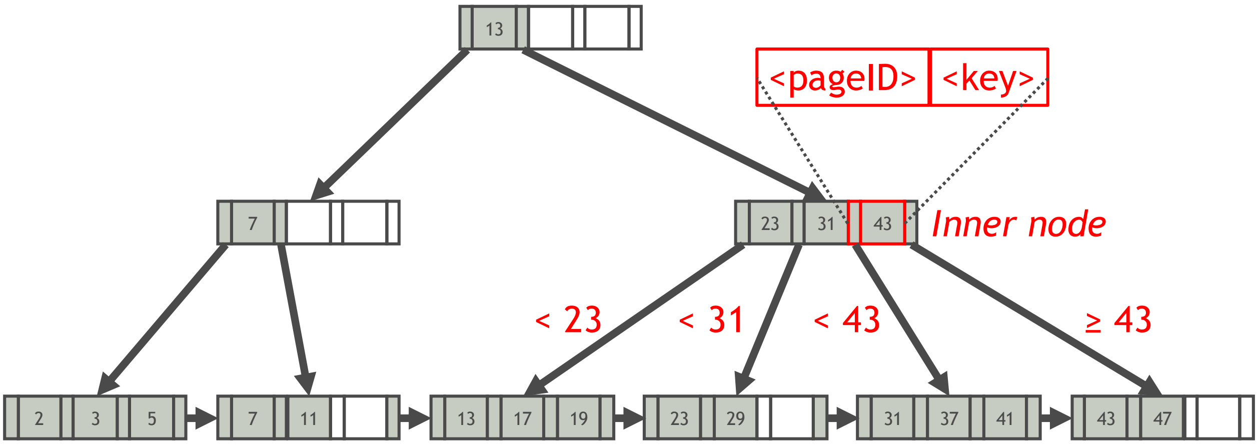
- The index entry array is (usually) kept in sorted key order



B+ Tree Inner Nodes

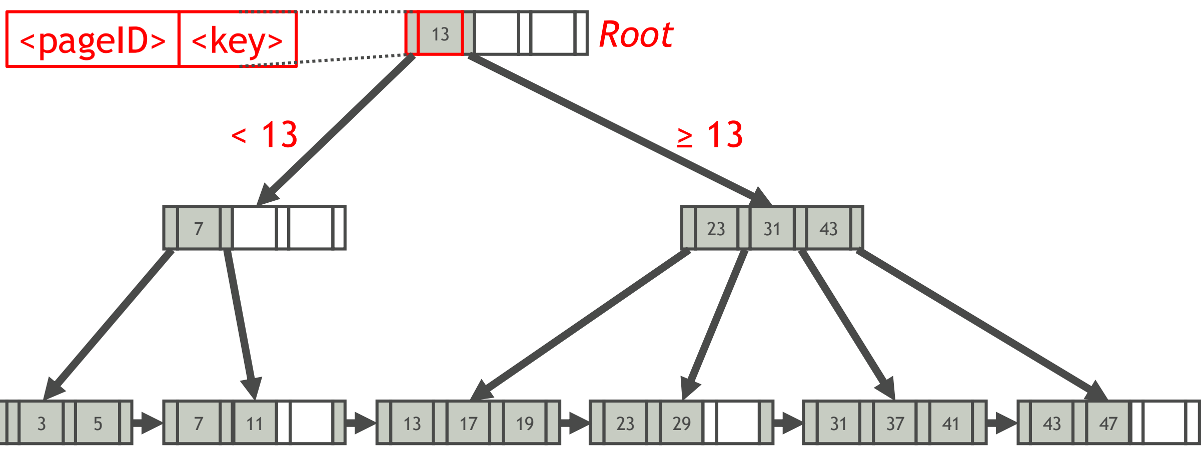
Every inner node is comprised of an array of keys and an array of pointers to its children

- 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

The root contains at least one key

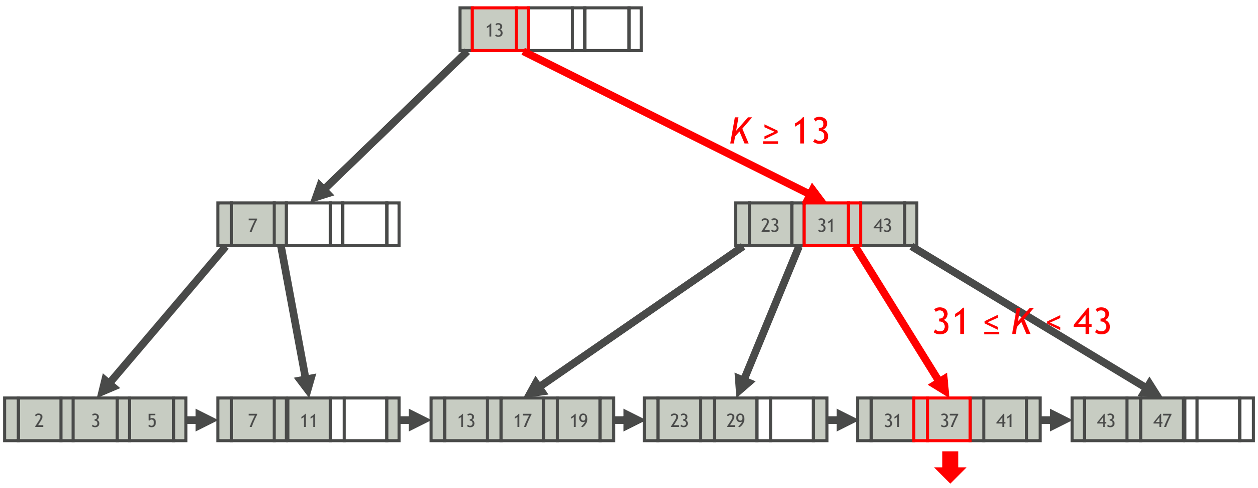


B+ Tree Lookup

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: $K = 37$

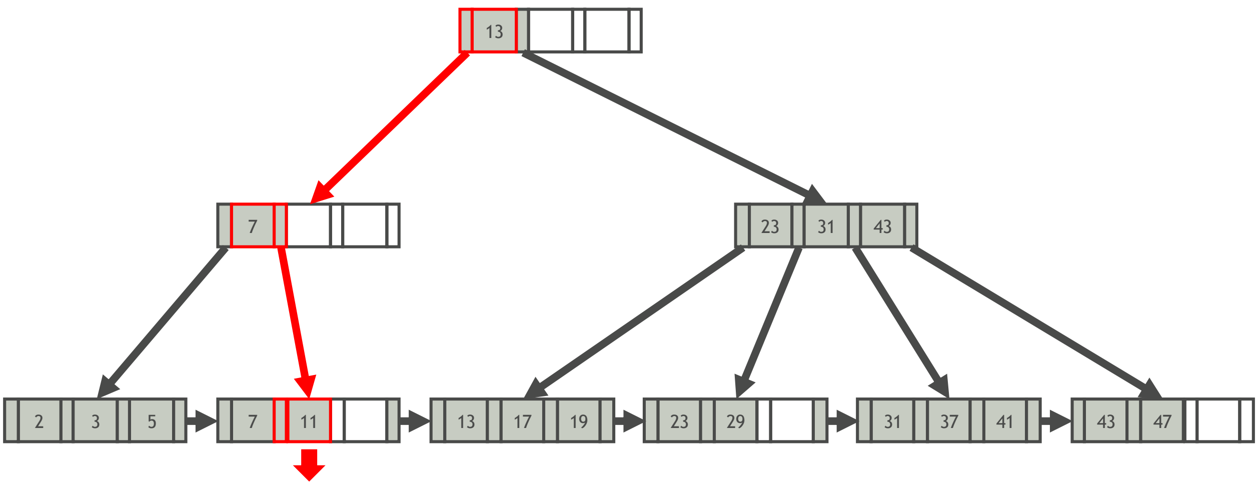


B+ Tree Range Query

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

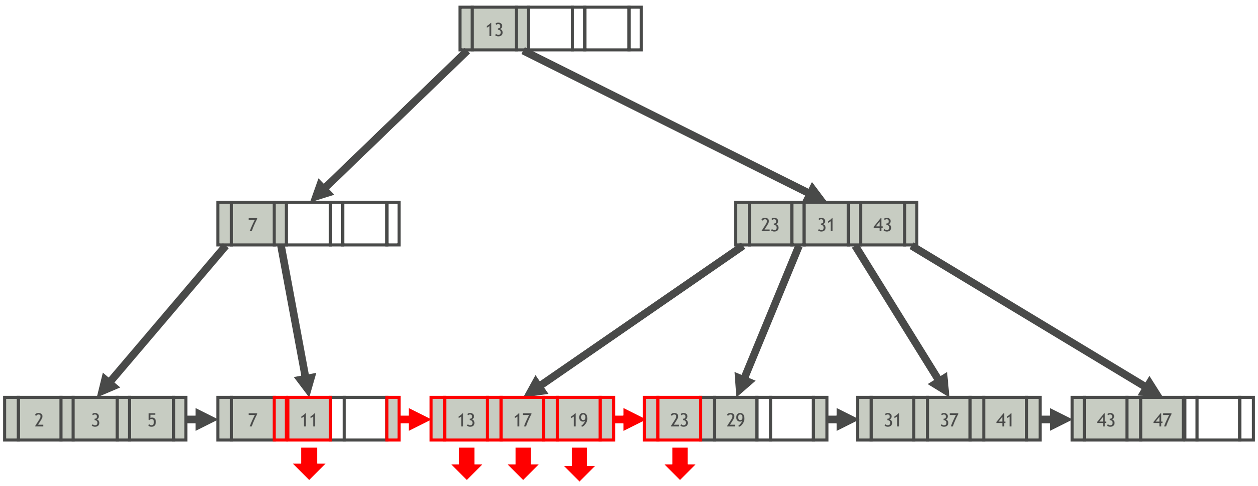
1. Find the first index entry E with the smallest key $\geq L$
2. Scan the contiguous index entries with keys $\leq U$ to the right of E

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



B+ Tree Range Query (Cont'd)

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



B+ Tree Insert

Insert an index entry with key K

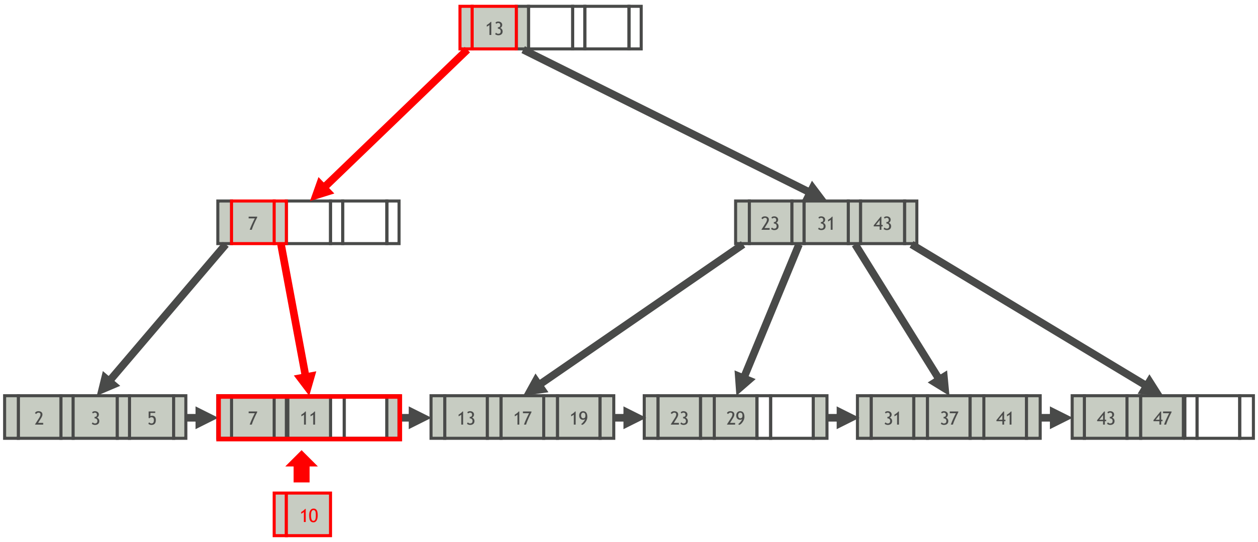
1. Find the correct leaf node L where the entry is to be inserted
2. Put the entry into L in sorted key order
3. If L has enough space, done!
   Otherwise, split the keys in L into L and a new node L_2
   1. Redistribute the entries evenly, copy up the middle key
   2. Insert an index entry pointing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

To split an inner node,

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

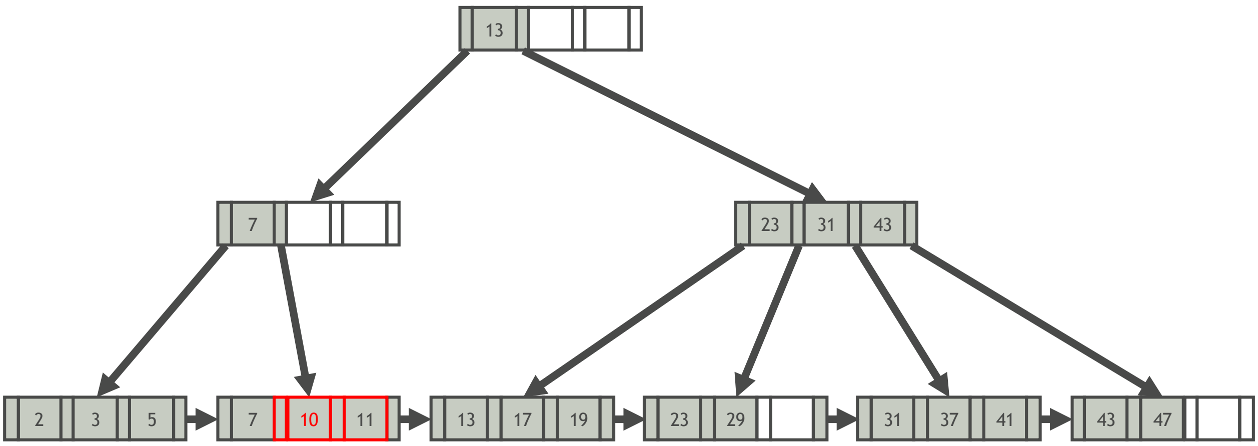
B+ Tree Insert: Example 1 (w/o Node Split)

Example: $K = 10$



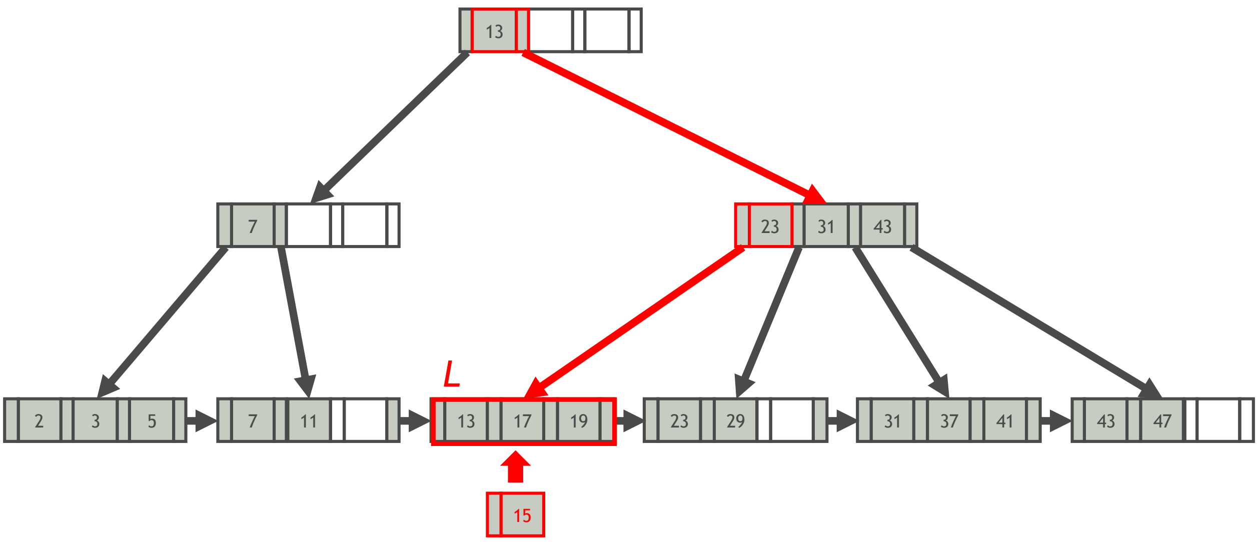
B+ Tree Insert: Example 1 (w/o Node Split)

Example: $K = 10$



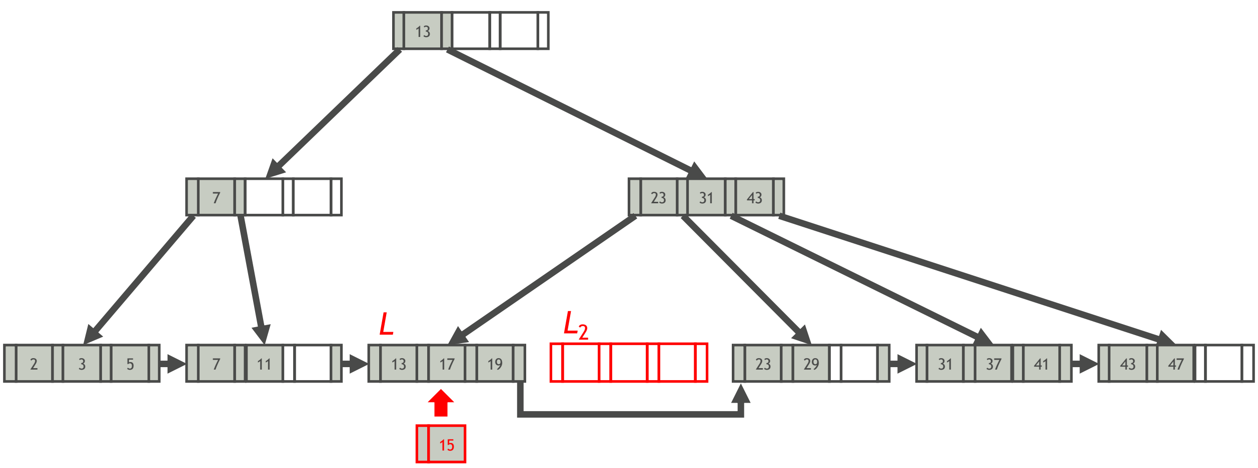
B+ Tree Insert: Example 2 (w/ Node Split)

Example: $K = 15$



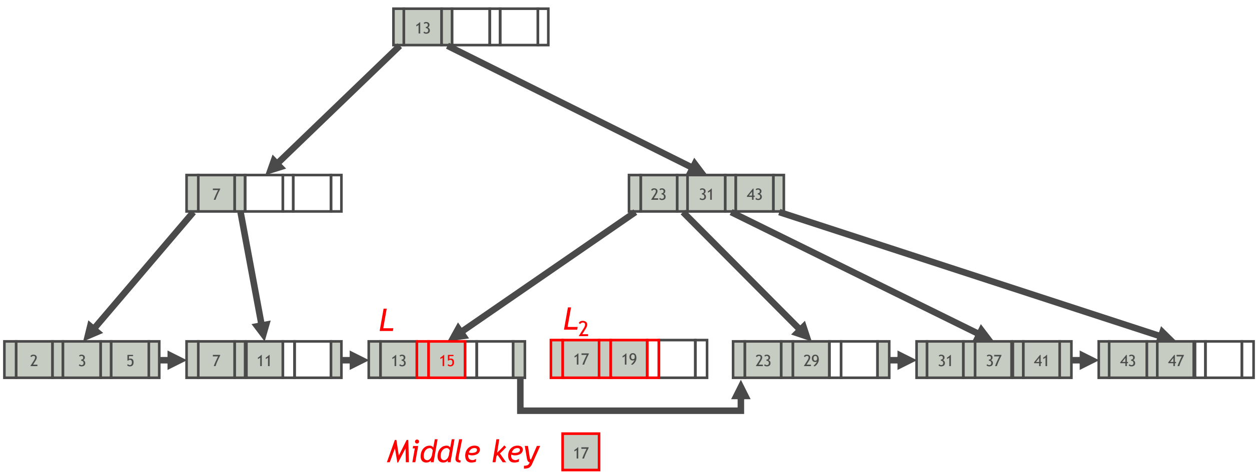
B+ Tree Insert: Example 2 (w/ Node Split)

Example: $K = 15$



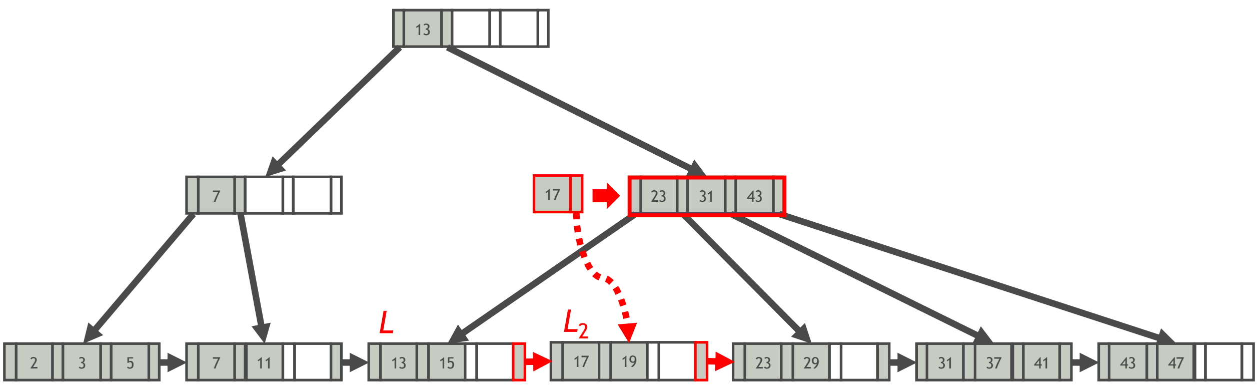
B+ Tree Insert: Example 2 (w/ Node Split)

Example: $K = 15$



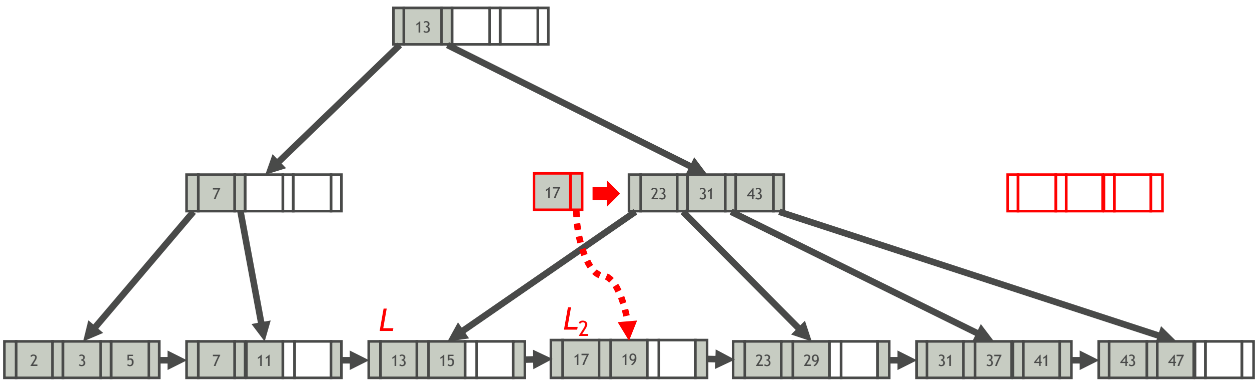
B+ Tree Insert: Example 2 (w/ Node Split)

Example: $K = 15$



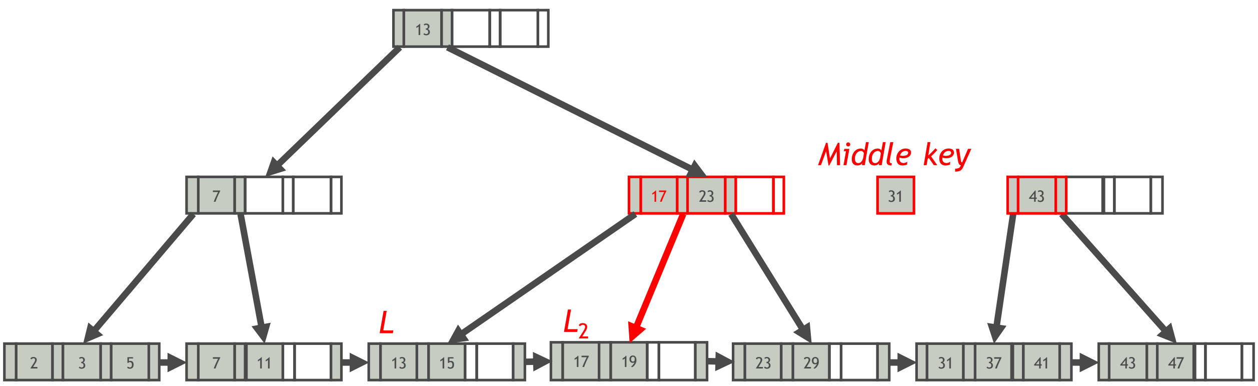
B+ Tree Insert: Example 2 (w/ Node Split)

Example: $K = 15$



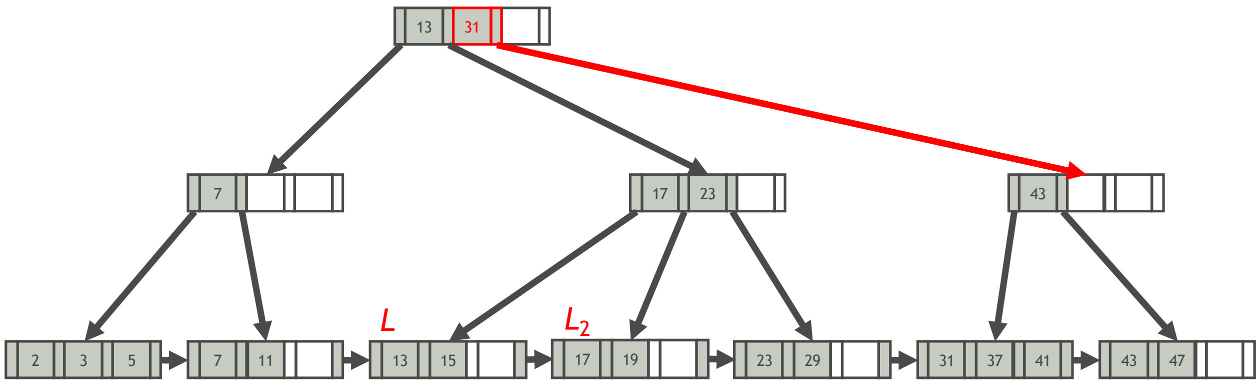
B+ Tree Insert: Example 2 (w/ Node Split)

Example: $K = 15$



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

Example: $K = 15$



B+ Tree Delete

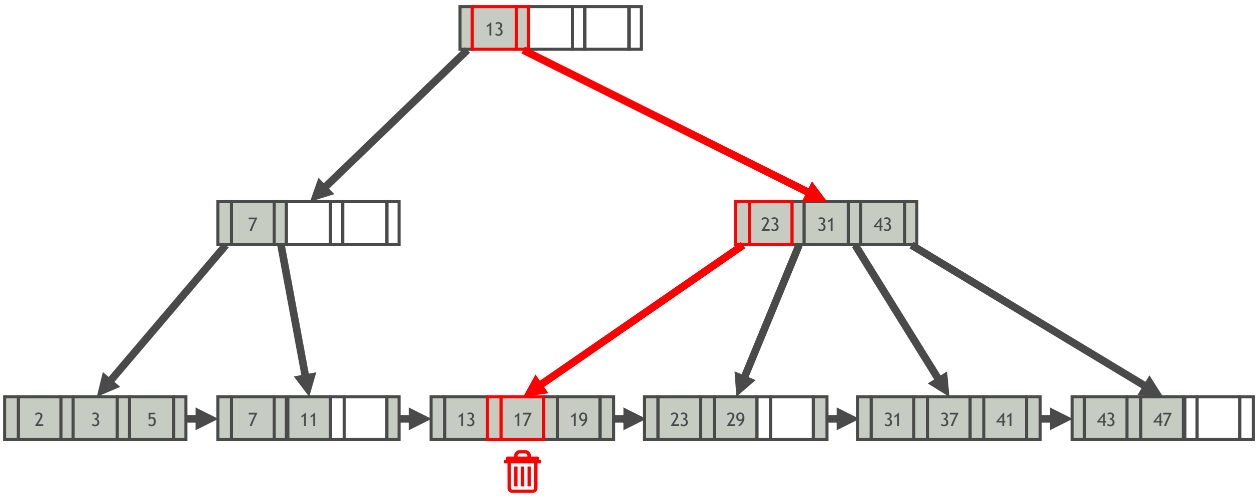
Delete an index entry with key K

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

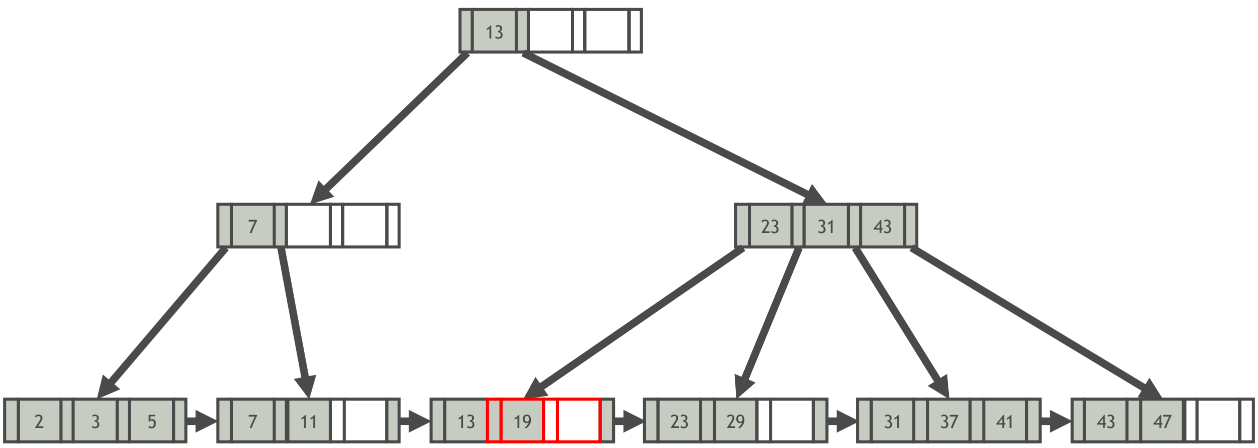
B+ Tree Delete: Example 1 (w/o Node Underflow)

Example: $K = 17$



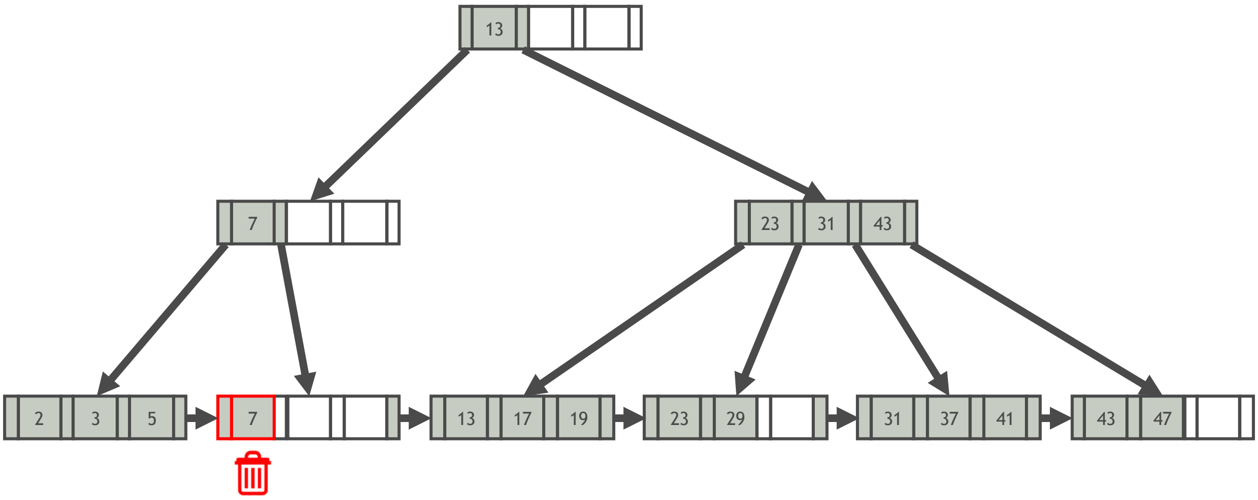
B+ Tree Delete: Example 1 (w/o Node Underflow)

Example: $K = 17$



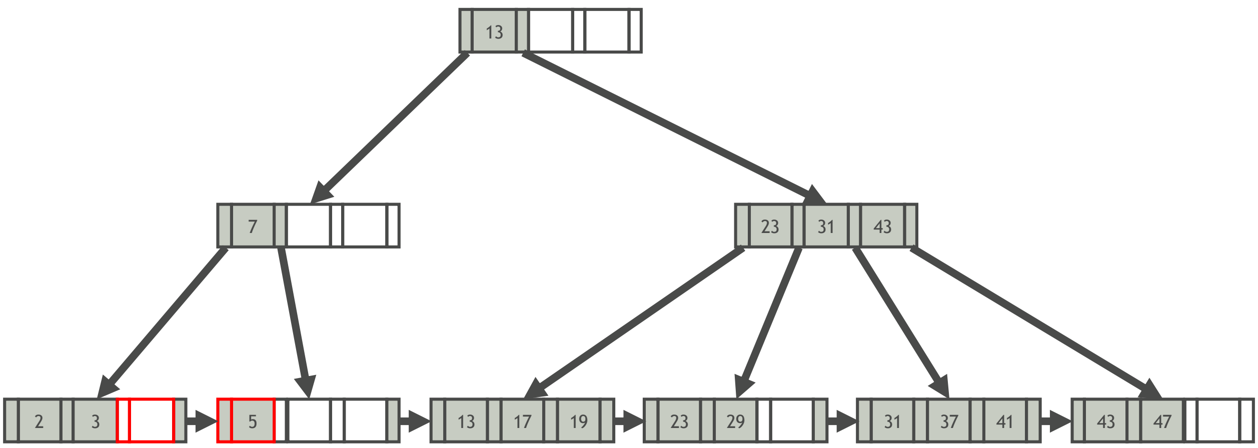
B+ Tree Delete: Example 2 (Key Redistribution)

Example: $K = 7$



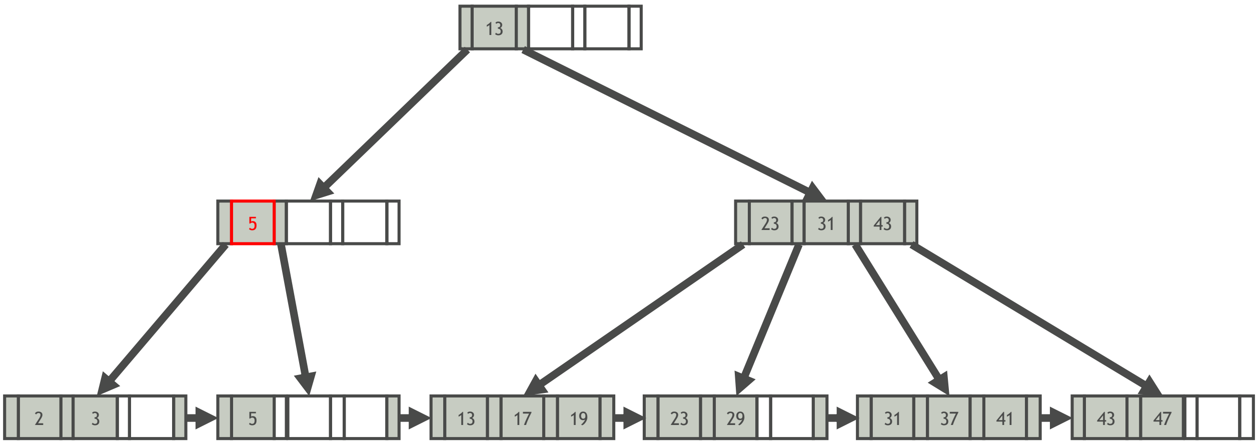
B+ Tree Delete: Example 2 (Key Redistribution)

Example: $K = 7$



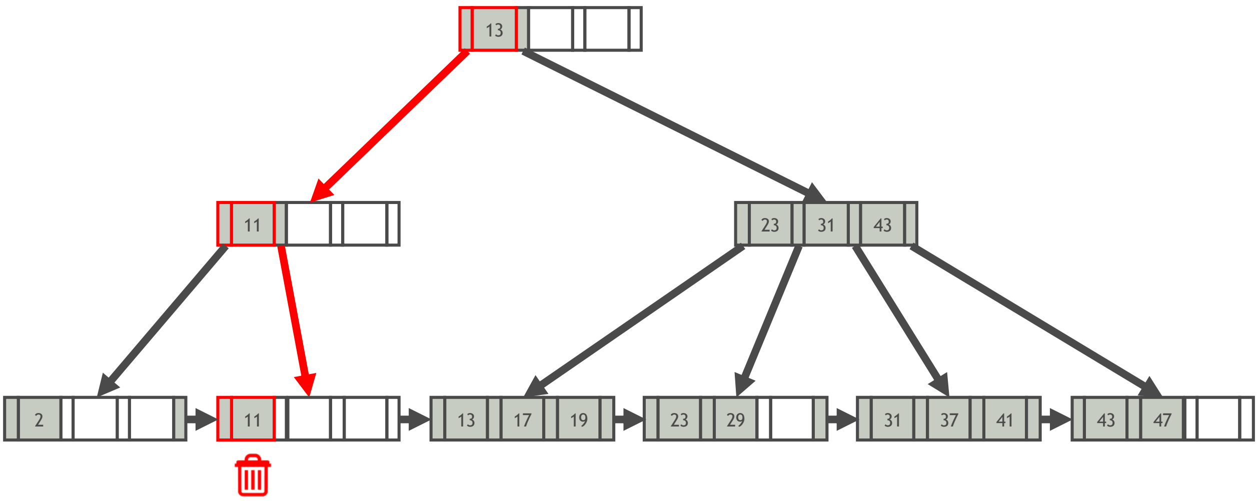
B+ Tree Delete: Example 2 (Key Redistribution)

Example: $K = 7$



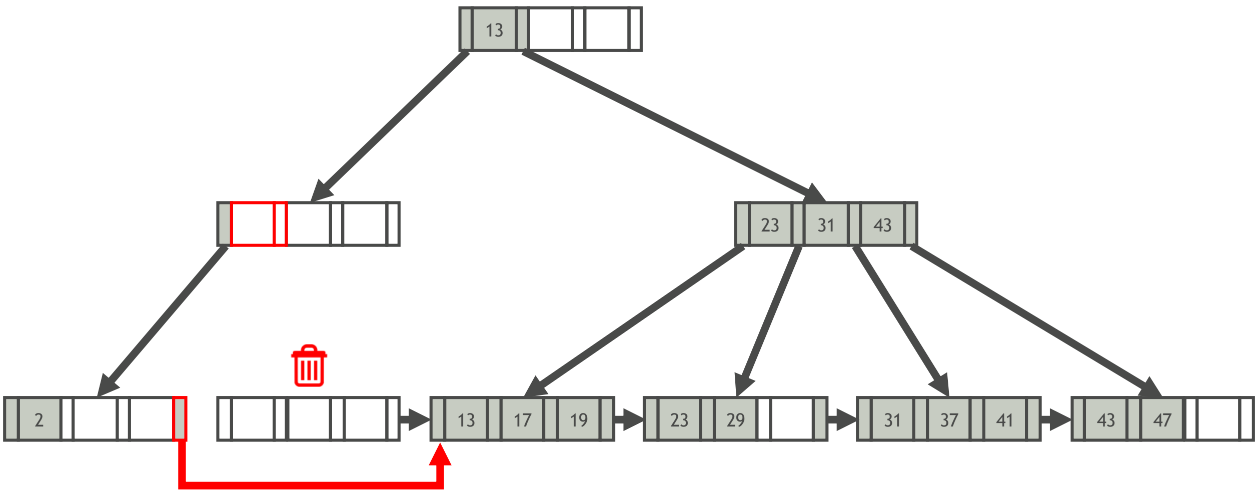
B+ Tree Delete: Example 3 (w/ Node Merge)

Example: $K = 11$



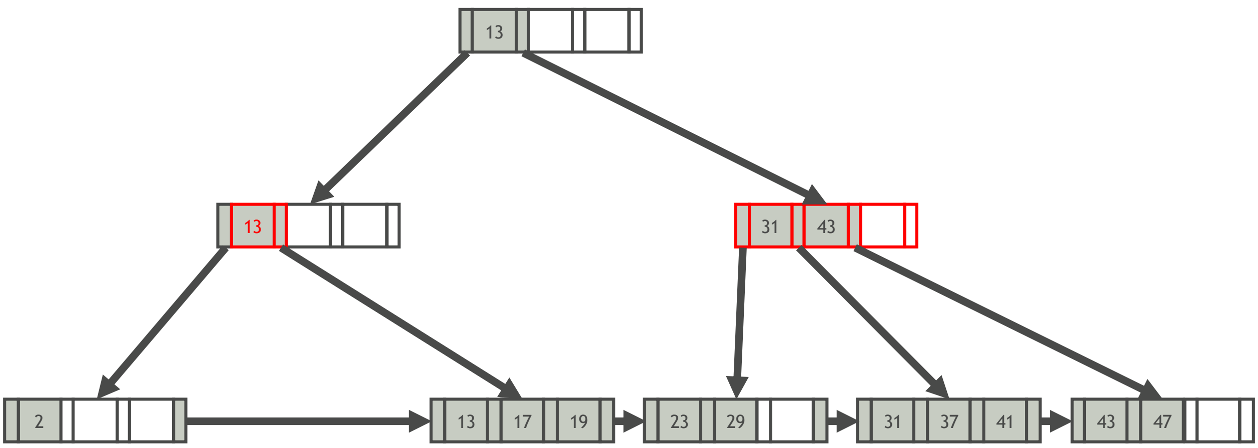
B+ Tree Delete: Example 3 (w/ Node Merge)

Example: $K = 11$



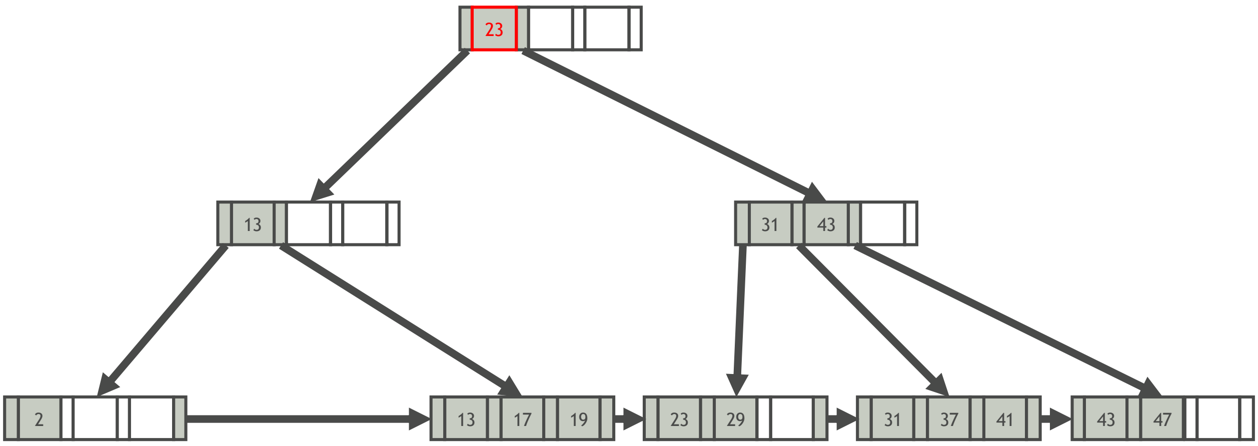
B+ Tree Delete: Example 3 (w/ Node Merge)

Example: $K = 11$



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

Example: $K = 11$



B+ Tree Demo

https://cmudb.io/btree

Key Compression

- The number of disk I/Os to retrieve a data entry in a B+ tree = the height of the tree $\approx \log_{fan\_out}(\#\text{ of data entries})$
- 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

Prefix Compression (前缀压缩)

- 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

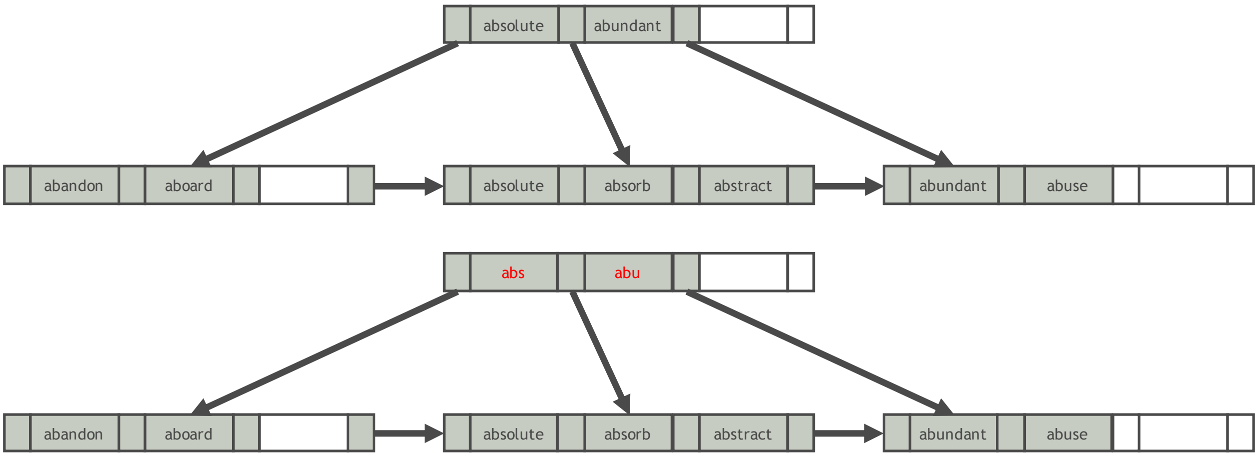
$\downarrow$ Prefix compression

Prefix: Micro

phone	soft	wave

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



Bulk Loading (批量加载)

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

Top-Down Approach

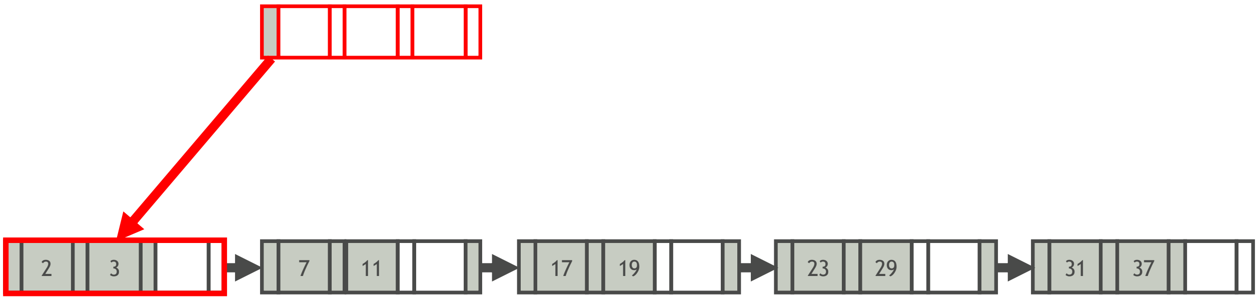
- Insert the index entries one at a time
- Expensive, because each entry requires to start from the root and go down to the appropriate leaf node

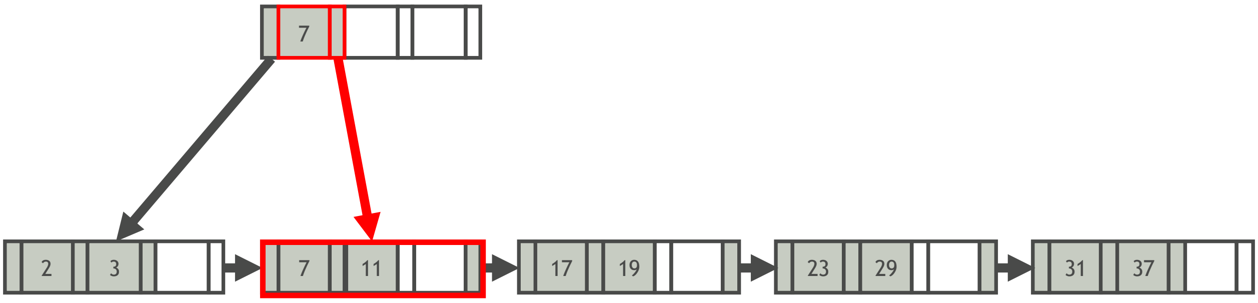
Bottom-Up Approach

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

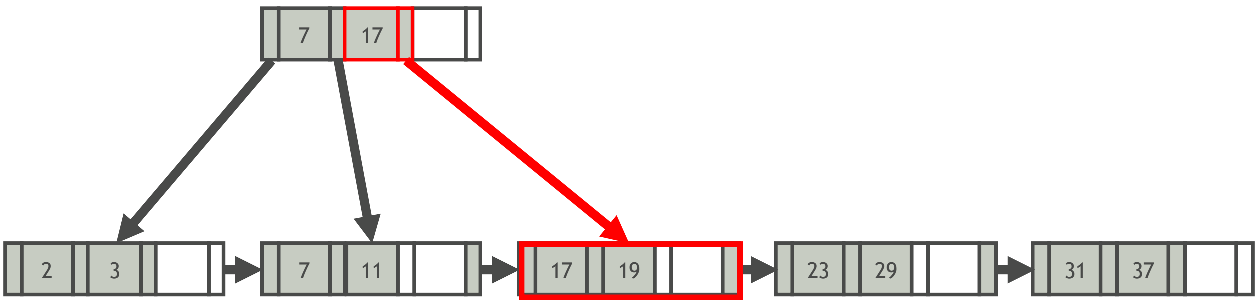
Sorted keys: 2, 3, 7, 11, 17, 19, 23, 29, 31, 37

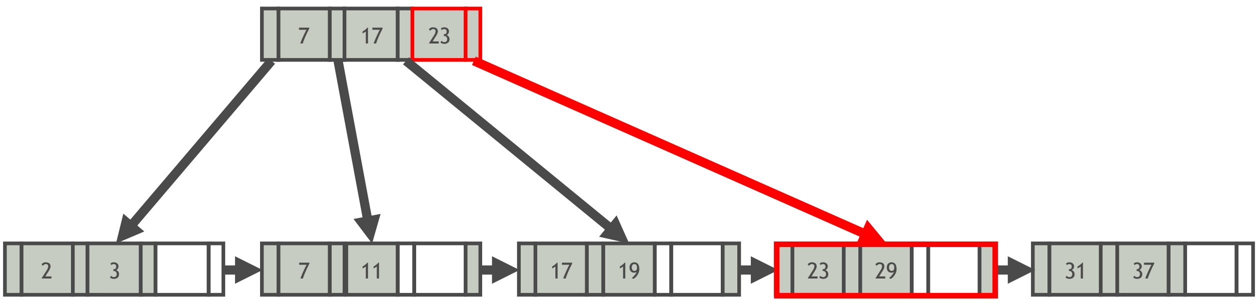




Bulk Loading: Example

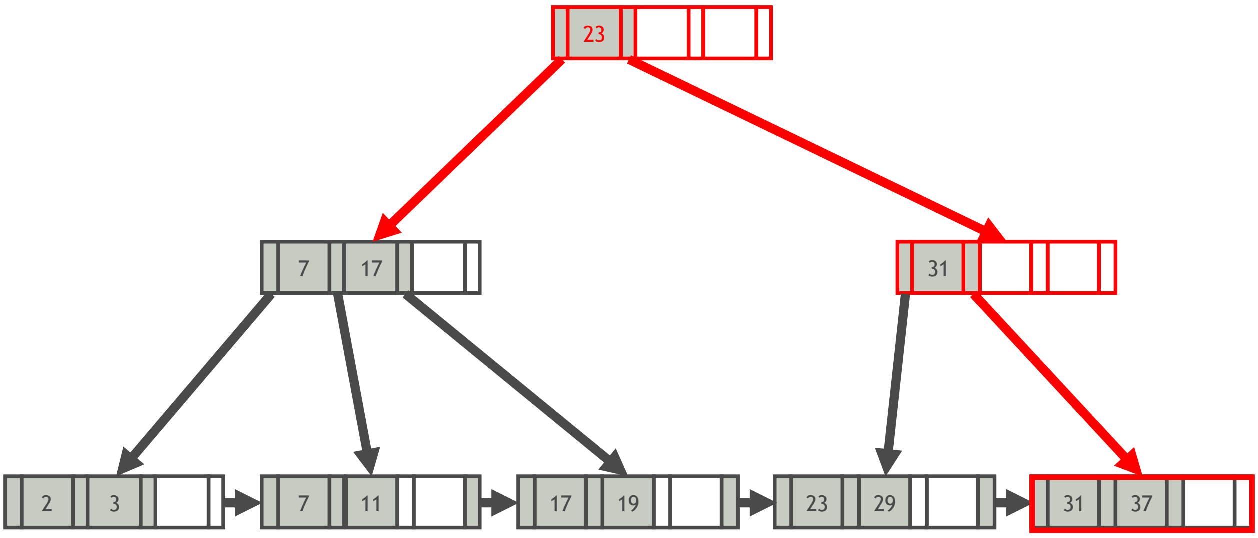
Sorted keys: 2, 3, 7, 11, 17, 19, 23, 29, 31, 37





Bulk Loading: Example

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



Log-Structured Merge-Trees (LSM-Trees)

Log-Structured Merge-Trees (LSM-Trees)

The log-structured merge-tree (LSM-tree) has been widely adopted in the storage layers of modern NoSQL systems

- LevelDB, RocksDB, HBase, Cassandra, TiDB and so on

An LSM-tree applies out-of-place updates

- First, it buffers all writes in memory
- Subsequently, it flushes the writes to disk and merges them using sequential I/Os

Basic LSM-Tree Structure

- **Memtable**: a mutable B+ tree or hash table in main memory
- **Immutable file**: an immutable sorted file on disk

Before compaction (merge)

Memtable | (3, 333), (7, 777) | *in memory*

Immutable file | (2, 222), (3, 123), (5, 555), (8, 888) | *on disk*

After compaction

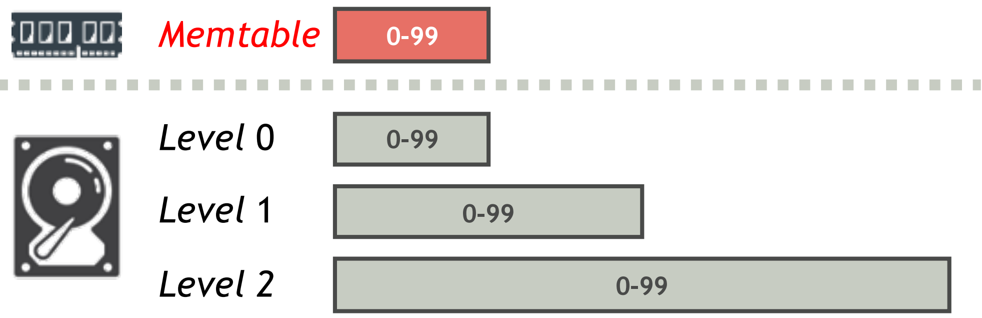
Memtable | Empty |

Immutable file | (2, 222), (3, 333), (5, 555), (7, 777), (8, 888) |

Disadvantage: When the immutable file is very large, the lookup and the merge on the file are costly

Leveled LSM-Tree Structure

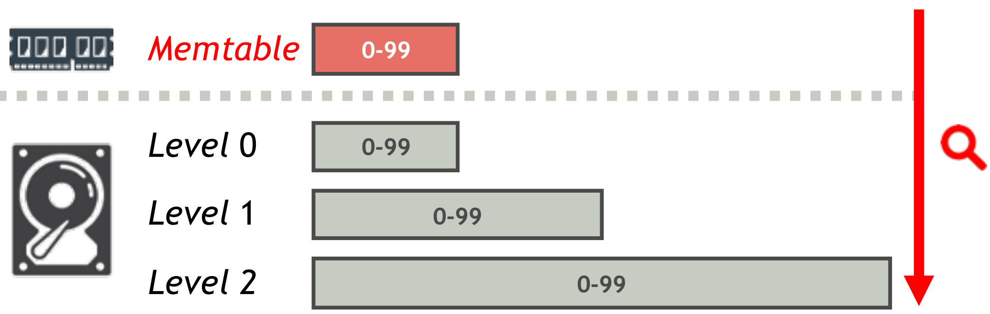
- **Memtable**: a mutable B+ tree or hash table in main memory
- **Level 0**: an immutable copy of the memtable on disk
- **Level** i ($i \geq 1$): immutable sorted file on disk
  - The key-value pairs in level $i+1$ are older than those in level i
  - Level $i+1$ is T times larger than level i



LSM-Tree Lookup

Find the latest value for the key K

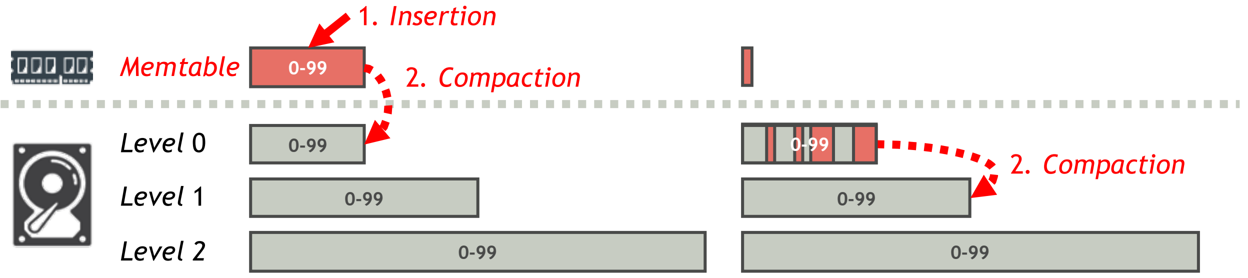
1. Find the key-value pair with key K in the memtable. If K is contained in the memtable, return the value for K (if a tombstone of K is found, return "not found"); Otherwise, find the latest value for the key K in the levels.
2. While $i \leq n$, find the key K in level i. If K is contained in level i, return the value for K (if a tombstone of K is found, return "not found")
3. If K is not contained in level n, return "not found"



LSM-Tree Insertion

Insert a key-value pair (K, V)

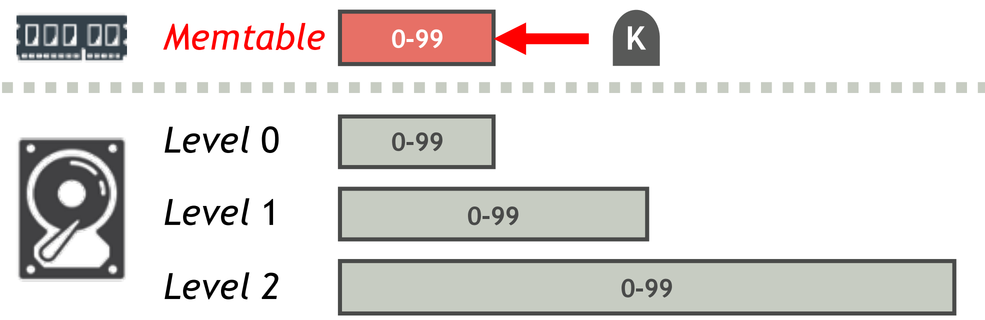
1. Insert (K, V) into the memtable (in-place update)
2. If the memtable does not overflow (exceed its maximum allowable size), done! Otherwise, compact the key-value pairs in the memtable to level 0
3. If level i overflows, compact the key-value pairs in level i to level $i + 1$



LSM-Tree Deletion

Delete the key-value pair with key K

1. Insert a *tombstone* for key K into the memtable
2. In compaction, any elder key-value pair with key K are deleted when it is merged with the tombstone of K



B+ Trees VS LSM-Trees

	B+ trees	LSM-trees
Update method	In-place update	Out-of-place update
Space amplification	Low (only one copy for a key)	High (many copies for a key)
Write performance	Low (random I/Os)	High (sequential I/Os)
Space utilization	Fragmentation (1/4 of a page is not used)	High (key-value pairs are compacted into sorted runs)
Concurrency control and failure recovery	Complicated	Simple (sorted runs are immutable, and compaction is out-of-place)

Summary

1. Hash-based Index Structures
    - Extensible Hash Tables
    - Linear Hash Tables
2. Tree-based Index Structures
    - B+ Trees
3. Log-Structured Merge-Trees (LSM-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。