

# COMP 421: Files & Databases

## Lecture 7: Indexes, B+ Trees

# Announcements

**Project #1** is due Sept 29<sup>th</sup> @ 11:59pm

**Project #2** will be released Sept 29<sup>th</sup>

**Mid-term Exam** on Oct 15<sup>th</sup>

→ In-class in this room.

→ Get accommodations in now if you have not

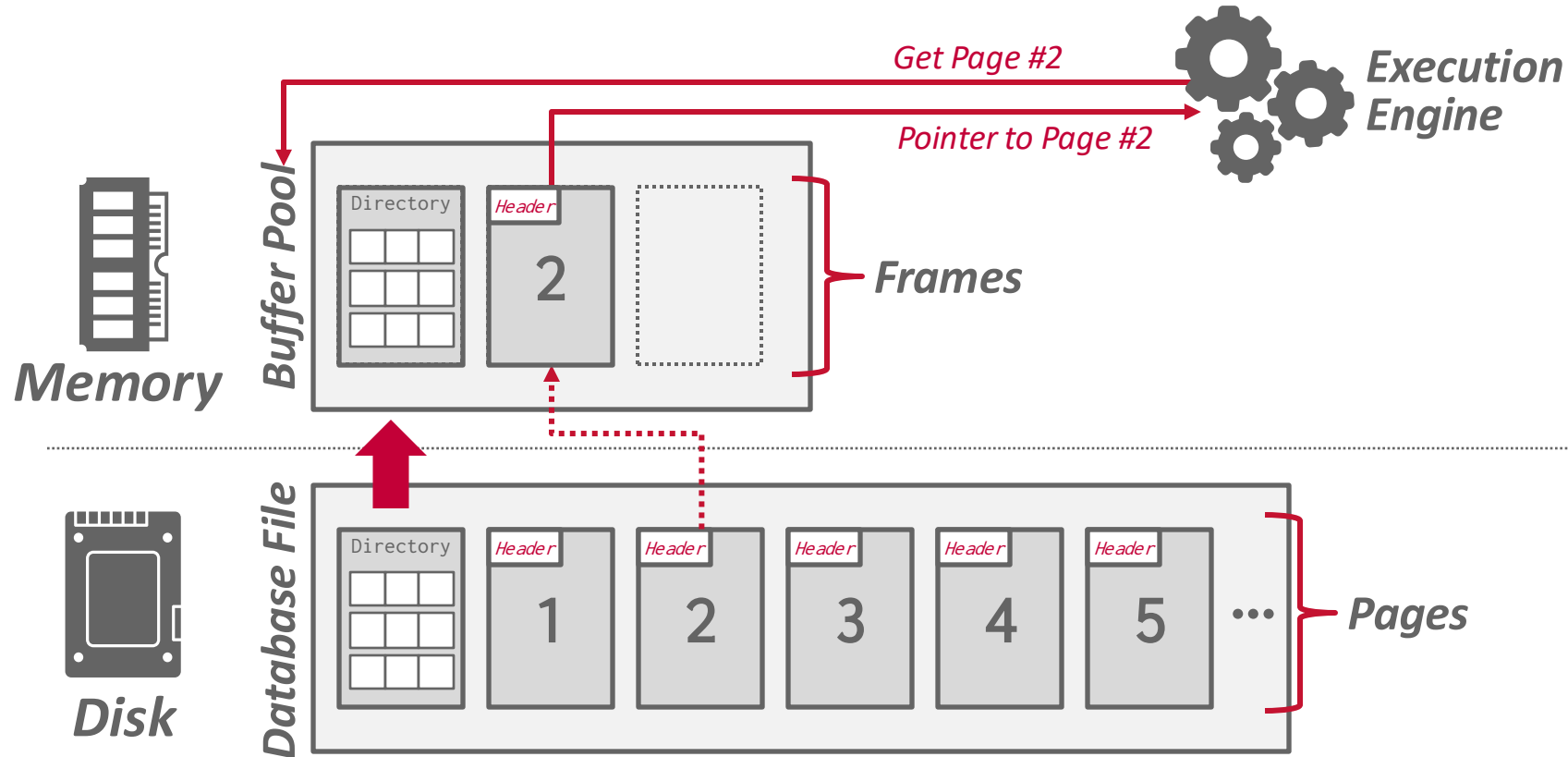
OLAP workloads demand specialized storage solutions

For OLTP, mostly targeted lookups, updates, deletes, inserts

How do we find the data we need?

- Which record IDs to request from storage manager?

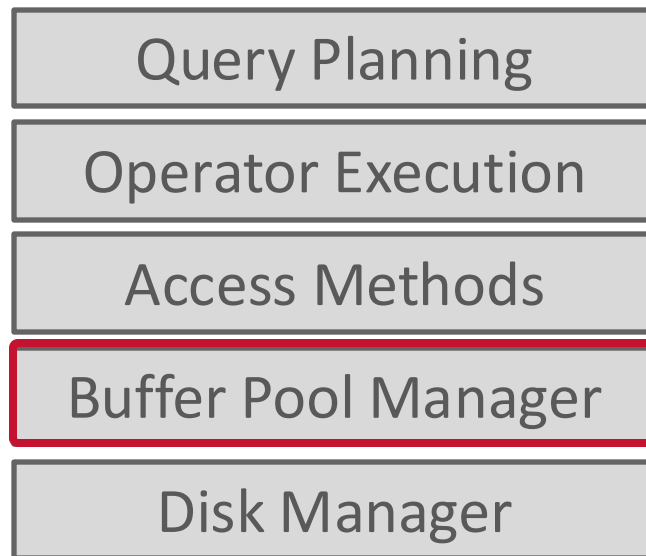
# Disk-oriented DBMS



# Moving Up The Stack...

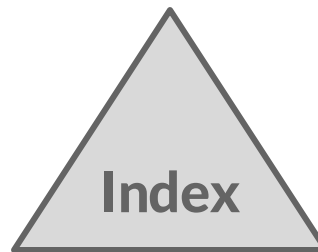
Disk manager and Buffer Pool Manager operate on low-level constructs: page #, record ID

Which pages to get? This comes from higher layers!



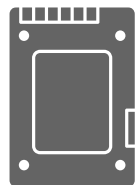
# Searching a Heap File

```
SELECT * FROM useracct
WHERE userName = ?
AND userPass = ?
```



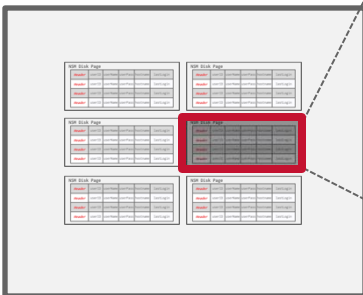
*NSM Disk Page*

<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	userID	userName	userPass	hostname	lastLogin
<i>header</i>	-	-	-	-	-



**Disk**

*Database File*



# Indexes vs. Filters

An **index** data structure over a subset of a table's attributes that are organized and/or sorted to provide the location of specific tuples using those attributes.

→ Example: B+Tree

A **filter** is a data structure that answers set membership queries; it tells you whether a record (likely) exists for a key but not where it is located.

→ Example: Bloom Filter

# Today's Agenda

B+Tree Overview

Design Choices

Optimizations



# B-Tree

## B-Tree

Department, Purdue University, West Lafayette, Indiana 47907

become, de facto, a standard for file organization. File indexes of users, database systems, and general-purpose access methods have all been proposed using B-trees. This paper reviews B-trees and shows why they have been successful. It discusses the major variations of the B-tree, especially the B\*-tree, the relative merits and costs of each implementation. It illustrates a general access method which uses a B-tree.

Phrases: B-tree, B\*-tree, B\*-tree, file organization, index

3.73 3.74 4.33 4 34

might be labeled with the employees' last names. A sequential request requires the searcher to examine the entire file, one folder at a time. On the other hand, a random request implies that the searcher, guided by the labels on the drawers and folders, need only extract one folder.

Associated with a large, randomly accessed file in a computer system is an index which, like the labels on the drawers and folders of the file cabinet, speeds retrieval by directing the searcher to the small part of the file containing the desired item. Figure 1 depicts a file and its index. An index may be physically integrated with the file, like the labels on employee folders, or physically separate, like the labels on the drawers. Usually the index itself is a file. If the index file is large, another index may be built on top of it to speed retrieval further, and so on. The resulting hierarchy is similar to the employee file, where the topmost index consists of labels on drawers, and the next level of index consists of labels on folders.

Natural hierarchies, like the one formed by considering last names as index entries, do not always produce the best performance.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.  
© 1979 ACM 0010-4892/79/0600-0121 \$00 75

postgres / src / backend / access / nbtree / README

1083 lines (959 loc) · 62.8 KB

Code

Blame

Raw



```
1 src/backend/access/nbtree/README
2
3 Btree Indexing
4 =====
5
6 This directory contains a correct implementation of Lehman and Yao's
7 high-concurrency B-tree management algorithm (P. Lehman and S. Yao,
8 Efficient Locking for Concurrent Operations on B-Trees, ACM Transactions
9 on Database Systems, Vol 6, No. 4, December 1981, pp 650-670). We also
10 use a simplified version of the deletion logic described in Lanin and
11 Shasha (V. Lanin and D. Shasha, A Symmetric Concurrent B-Tree Algorithm,
12 Proceedings of 1986 Fall Joint Computer Conference, pp 380-389).
13
14 The basic Lehman & Yao Algorithm
15 -----
```



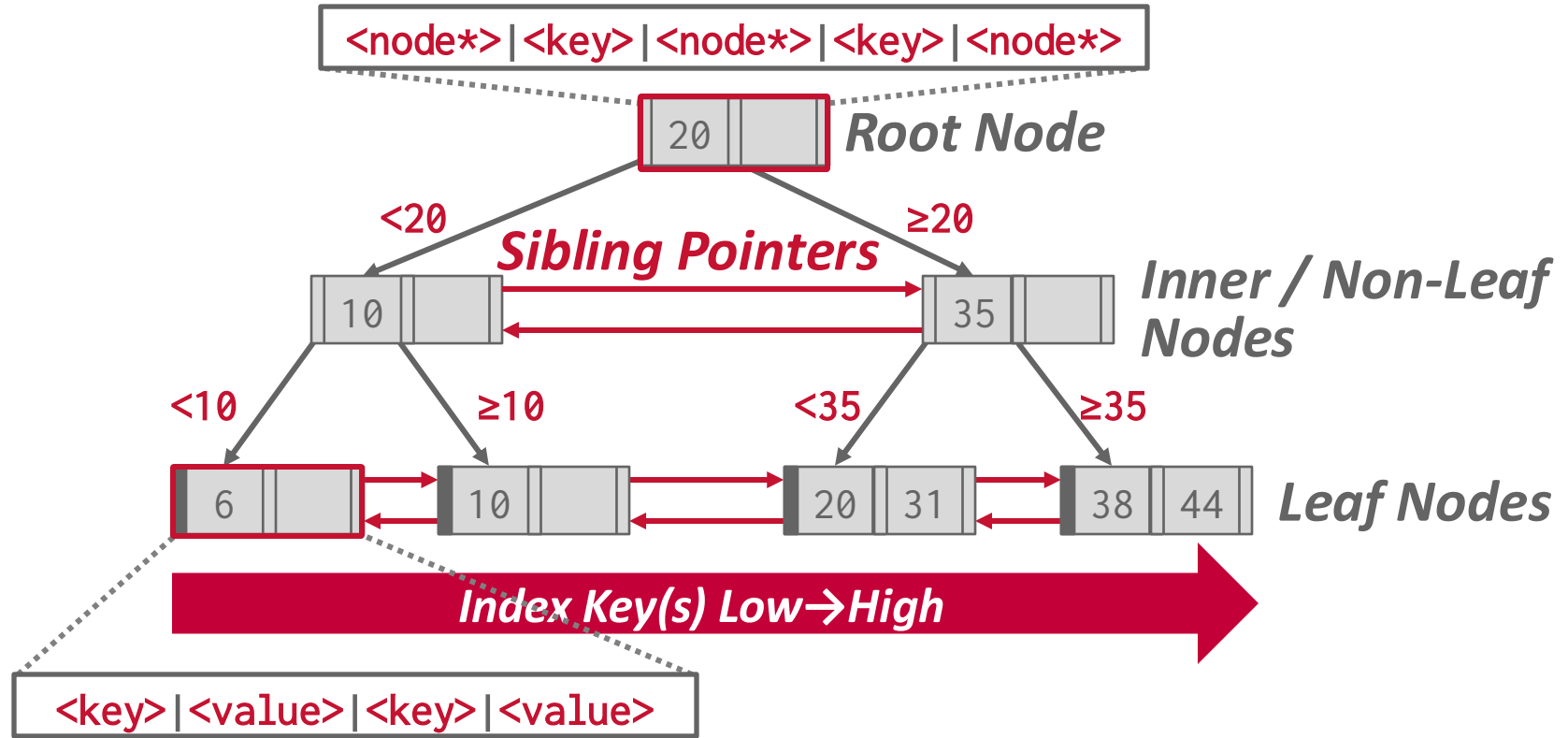
# B+Tree

A **B+Tree** is a self-balancing, ordered  **$m$** -way tree for searches, sequential access, insertions, and deletions in  **$O(\log_m n)$**  where  **$m$**  is the tree fanout.

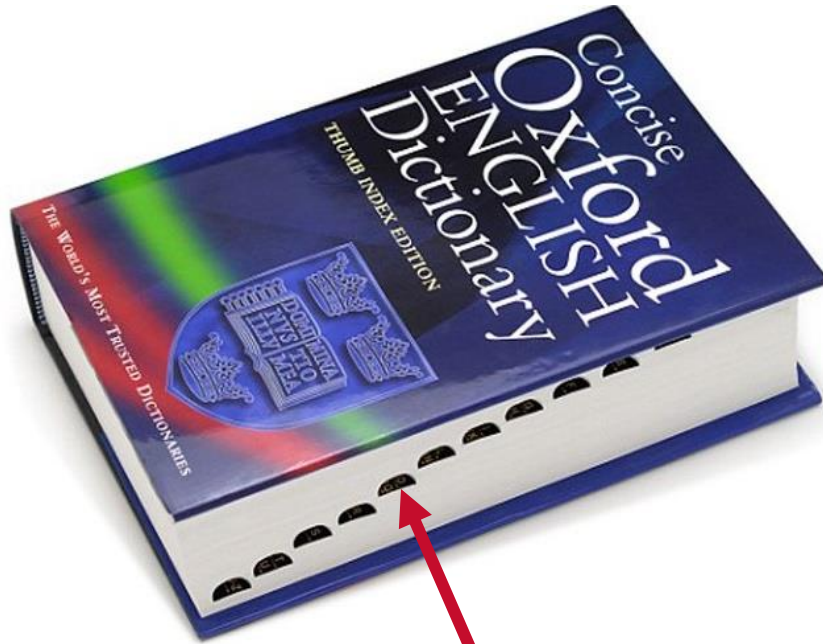
- It is perfectly balanced (i.e., every leaf node is at the same depth in the tree)
- Every node other than the root is at least half-full  
 **$m/2 - 1 \leq \#keys \leq m - 1$**
- Every inner node with  **$k$**  keys has  **$k + 1$**  non-null children.
- Optimized for reading/writing large data blocks.

Some real-world implementations relax these properties, but we will ignore that for now...

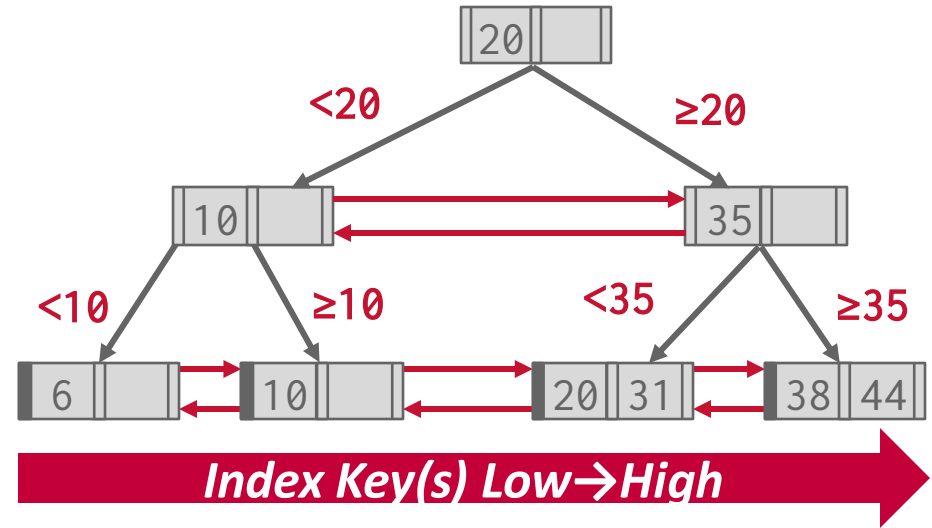
# B+Tree Example



# B+Tree Example



"M words begin here"



# Nodes

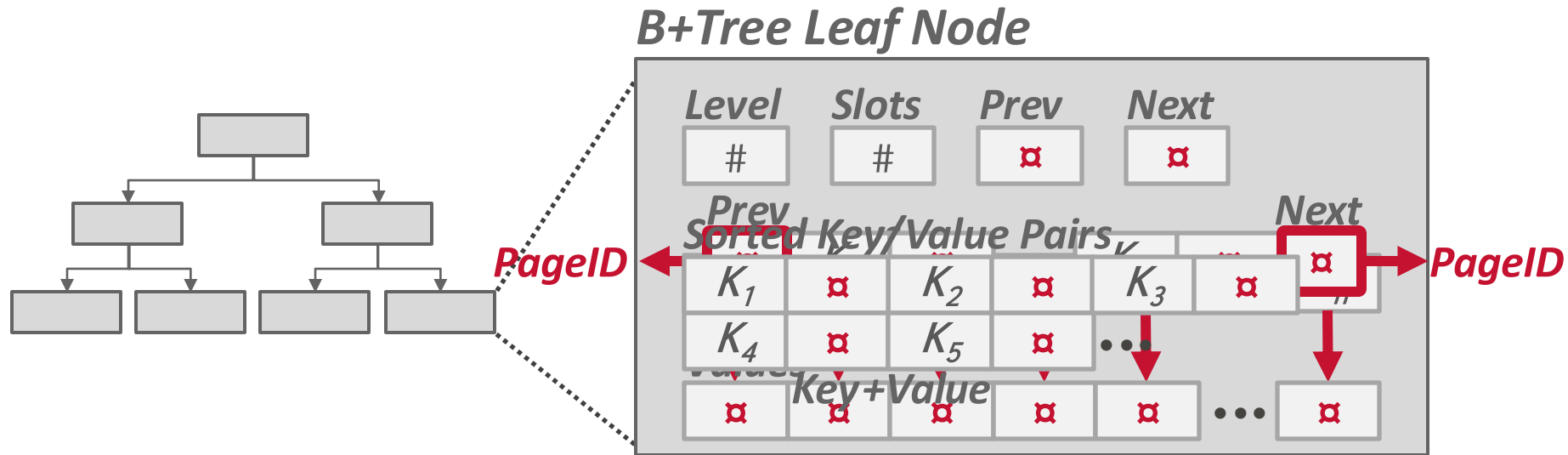
Every B+Tree node is comprised of an array of key/value pairs.

- The keys are derived from the index's target attribute(s).
- The associated data will differ based on whether the node is classified as an inner node or a leaf node.

The arrays are (usually) kept in sorted key order.

Store all **NULL** keys at either first or last leaf nodes.

# B+tree Leaf Nodes



# Leaf Node Values

## Approach #1: Record IDs

- A pointer to the location of the tuple to which the index entry corresponds.
- Most common implementation.



## Approach #2: Tuple Data

- Index-Organized Storage
- Primary Key Index: Leaf nodes store the contents of the tuple.
- Secondary Indexes: Leaf nodes store tuples' primary key as their values.



# B-Tree Vs. B+Tree

The original **B-Tree** from 1971 stored keys and values in all nodes in the tree.

→ More space-efficient, since each key only appears once in the tree.

A **B+Tree** only stores values in leaf nodes. Inner nodes only guide the search process.



# B+Tree – INSERT

Find correct leaf node  $L$ .

Insert data entry into  $L$  in sorted order.

If  $L$  has enough space, done!

Otherwise, split  $L$  keys into  $L$  and a new node  $L_2$

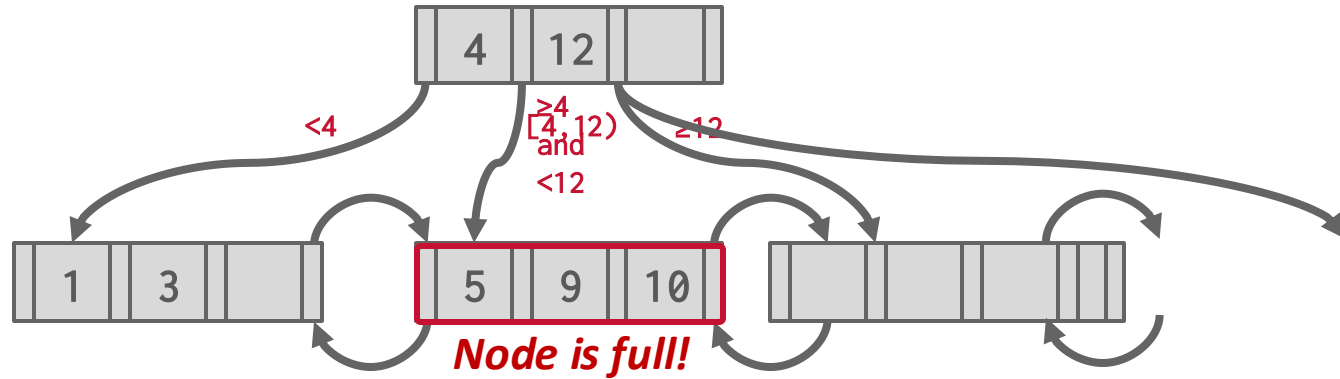
→ Redistribute entries evenly, copy up middle key.

→ Insert index entry pointing to  $L_2$  into parent of  $L$ .

To split inner node, redistribute entries evenly,  
but push up middle key.

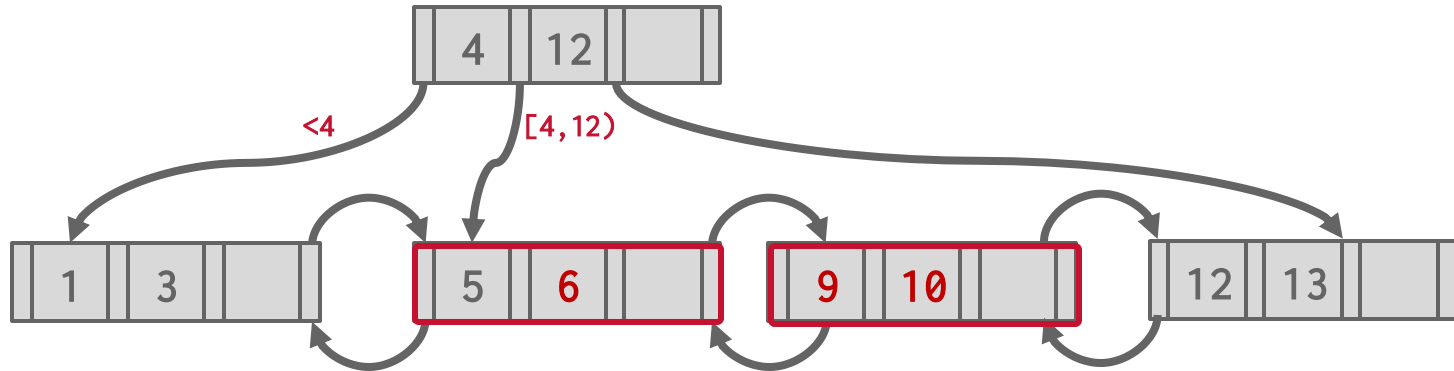
# B+Tree – INSERT Example (1)

Insert 6



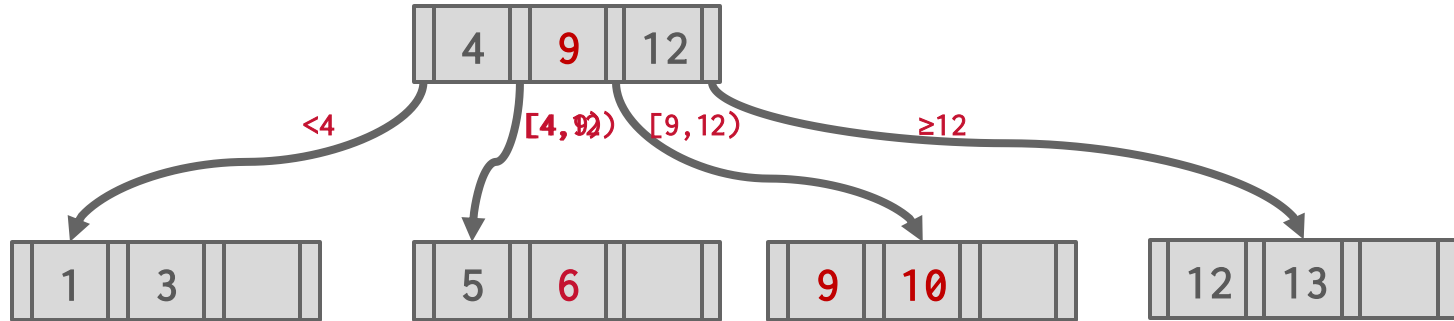
# B+TREE – INSERT EXAMPLE (1)

Insert 6



# B+TREE – INSERT EXAMPLE (1)

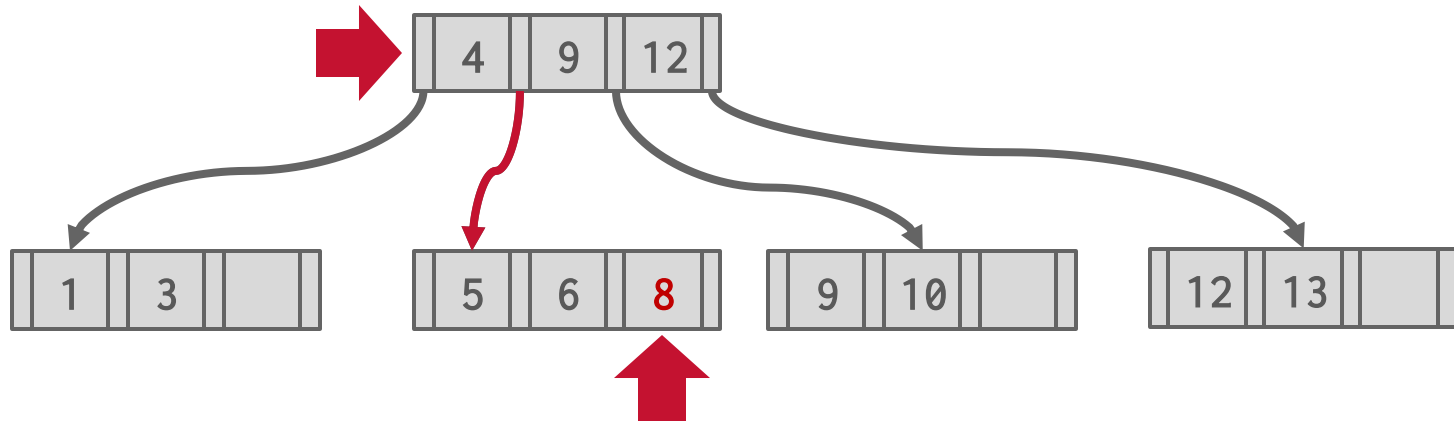
Insert 6



# B+TREE – INSERT EXAMPLE (1)

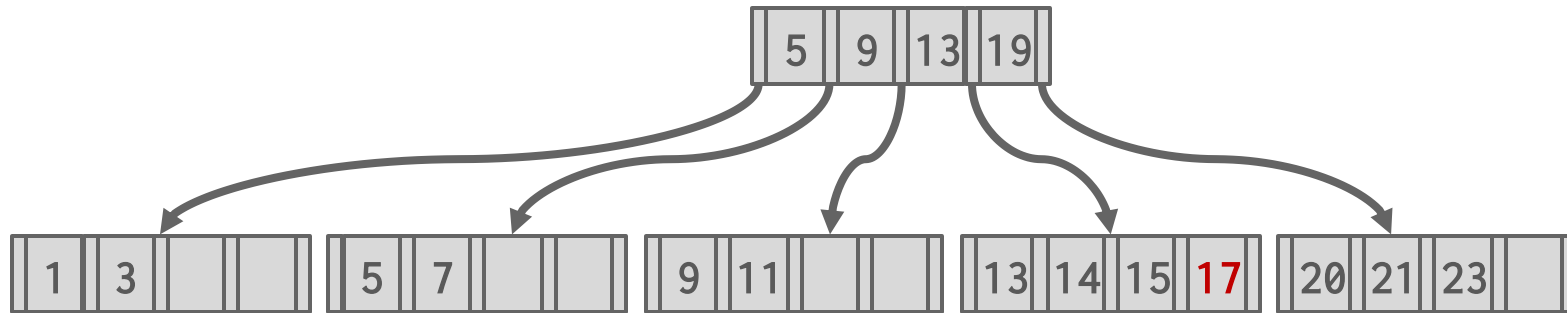
Insert 6

Insert 8



# B+TREE – INSERT EXAMPLE (2)

Insert 17

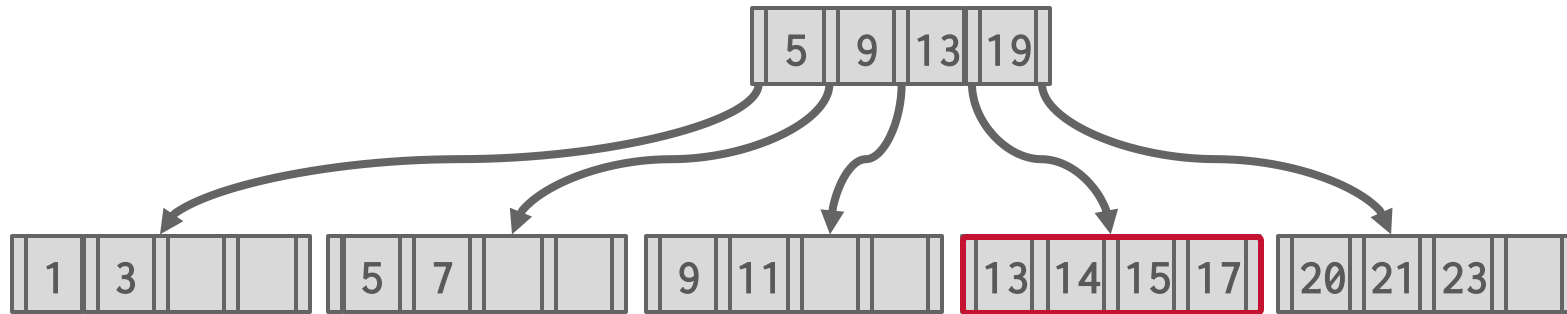


*Note: New Example/Tree.*

# B+TREE – INSERT EXAMPLE (3)

Insert 17

Insert 16

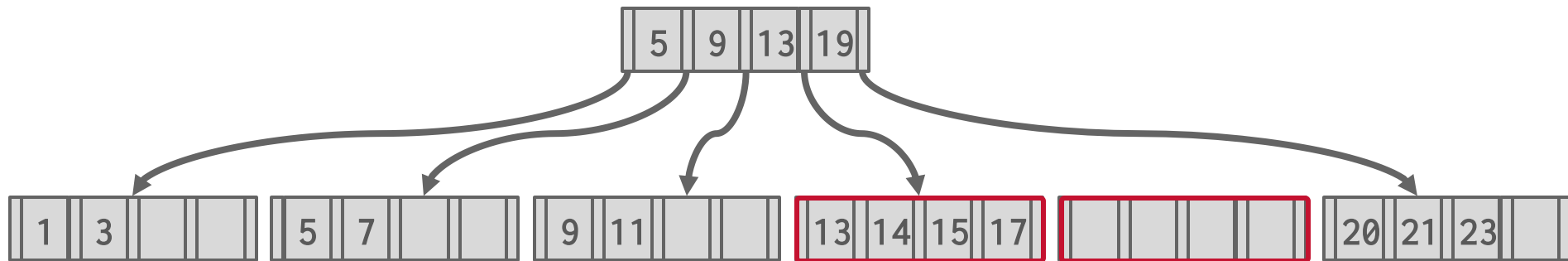


*Split the node  
Copy the middle key  
'14' up.  
Push the key up.*

# B+TREE – INSERT EXAMPLE (3)

Insert 17

Insert 16



***New Node!***  
***Shuffle keys from the  
node that triggered the  
split.***

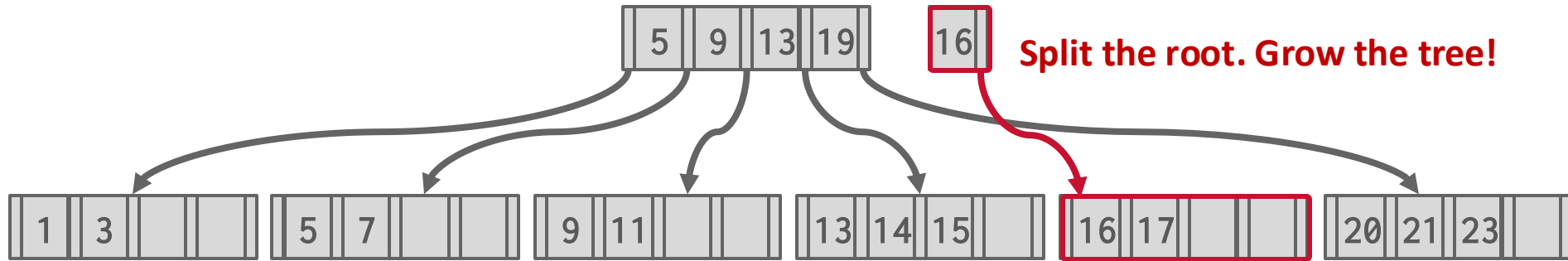


# B+TREE – INSERT EXAMPLE (3)

Insert 17

Insert 16

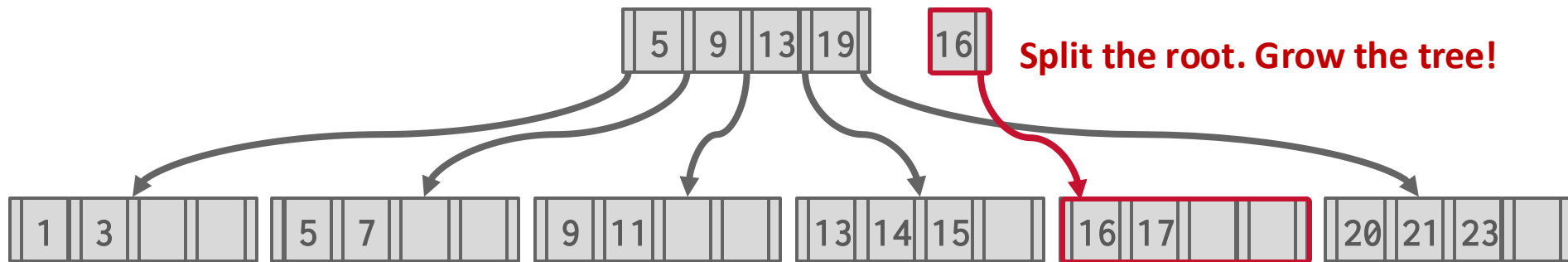
*Want to create a key, pointer pair like this. But cannot insert it in the root node, which is full.*



# B+TREE – INSERT EXAMPLE (3)

Insert 17

Insert 16



# B+TREE – INSERT EXAMPLE (3)

Insert 17

Insert 16



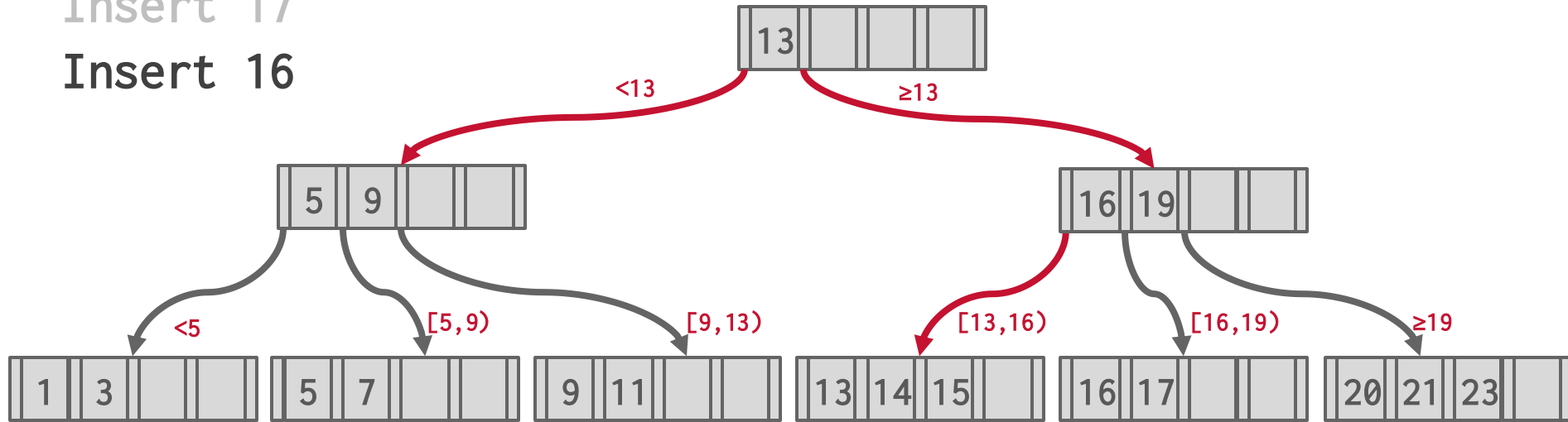
*Next, need to split the “old” root, then point to the split nodes from the new root.*



# B+TREE – INSERT EXAMPLE (3)

Insert 17

Insert 16



# B+Tree – DELETE

Start at root, find leaf **L** where entry belongs.

Remove the entry.

If **L** is at least half-full, done!

If **L** has only  $m/2-1$  entries,

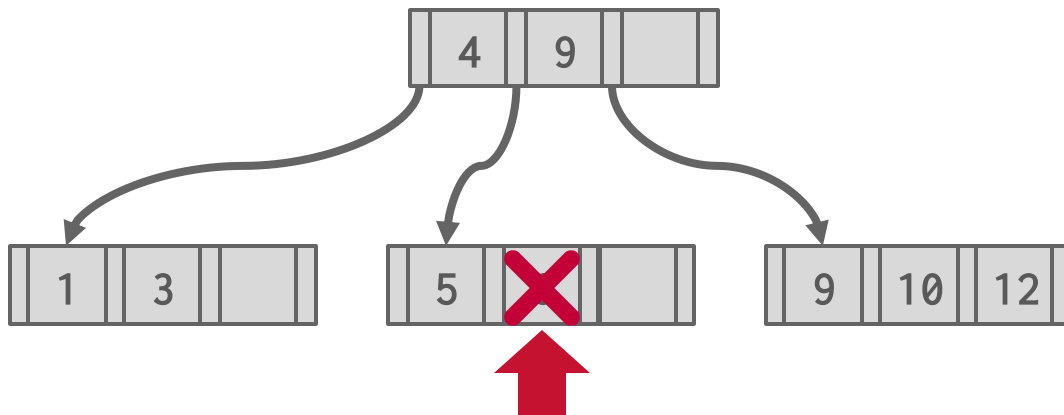
→ Try to re-distribute, borrowing from sibling (adjacent node with same parent as **L**).

→ If re-distribution fails, merge **L** and sibling.

If merge occurred, must delete entry (pointing to **L** or sibling) from parent of **L**.

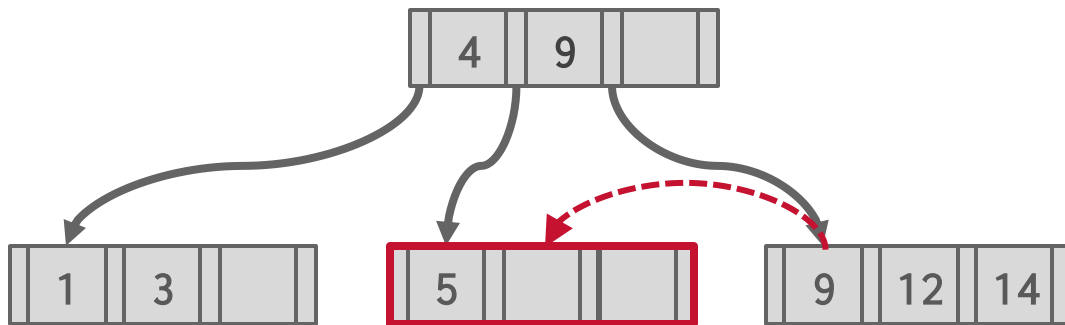
# B+Tree – DELETE Example (1)

Delete 6



# B+Tree – DELETE Example (1)

Delete 6

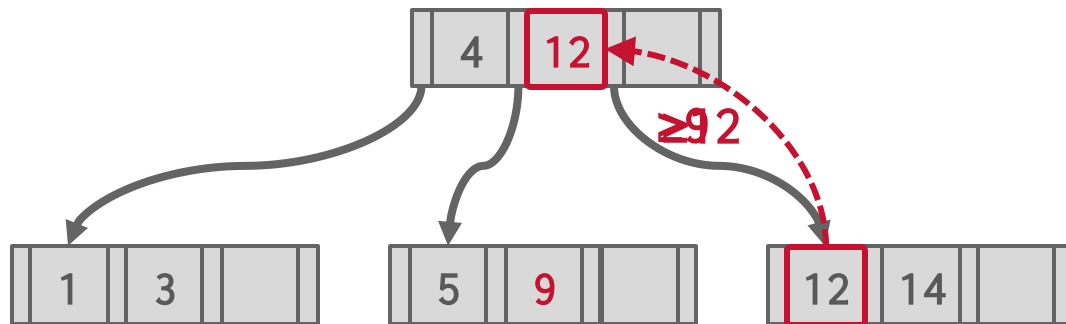


*Borrow from a “rich” sibling node.*

*Could borrow from either sibling.*

# B+Tree – DELETE Example (1)

Delete 6

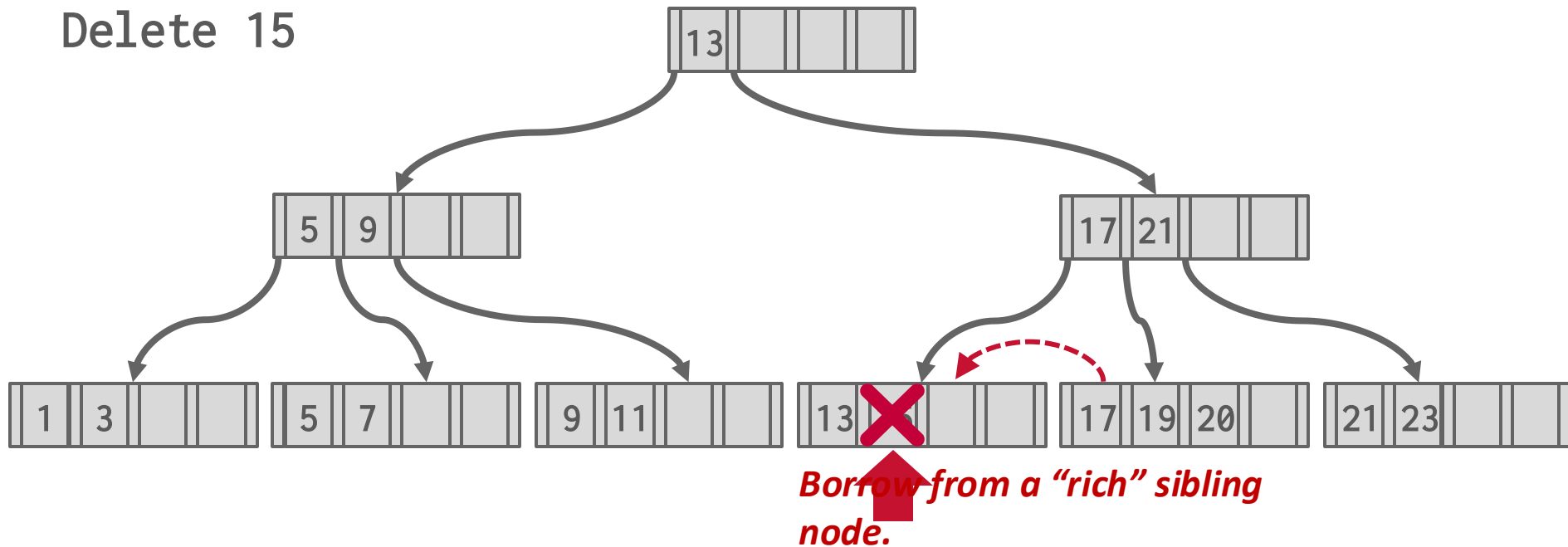


*Need to update parent node!*



# B+Tree – DELETE Example (2)

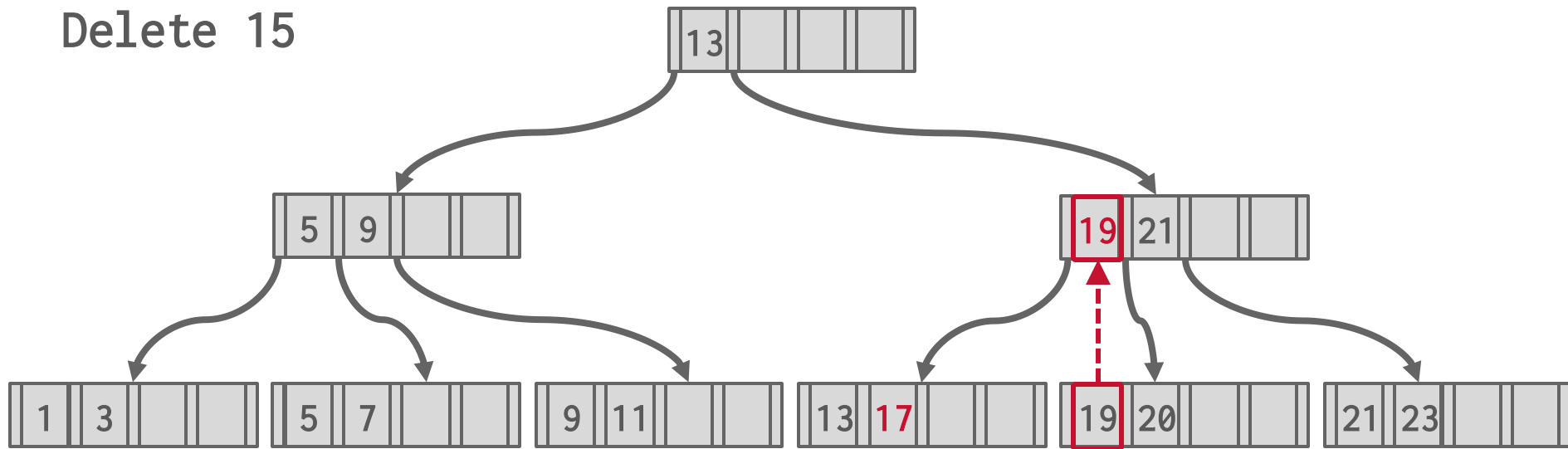
Delete 15



Note: New Example/Tree.

# B+Tree – DELETE Example (2)

Delete 15

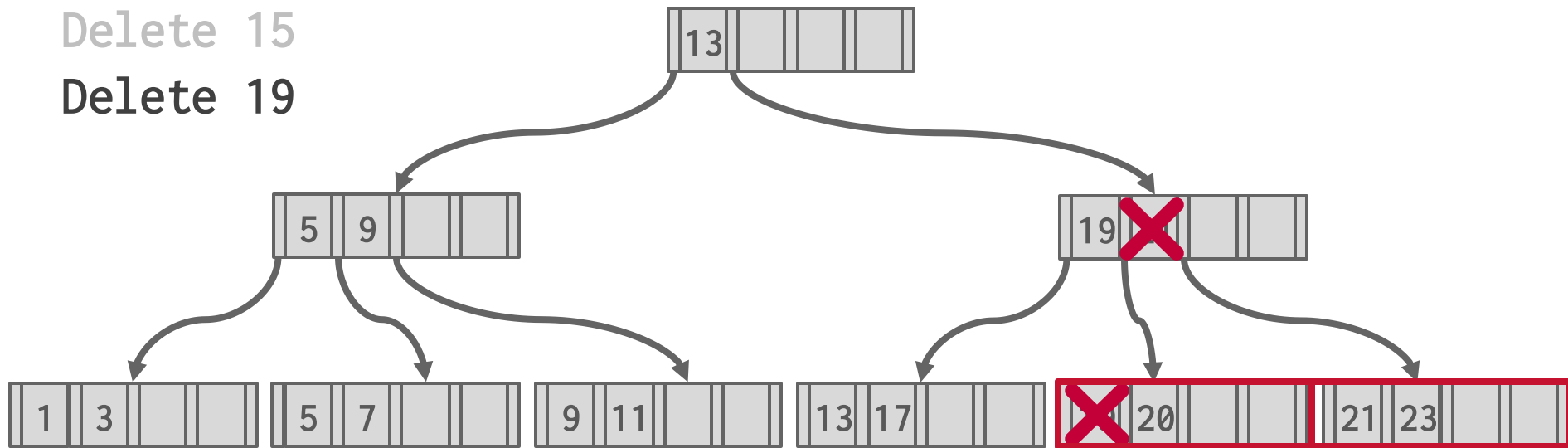


*Need to update parent node!*

# B+Tree – DELETE Example (3)

Delete 15

Delete 19

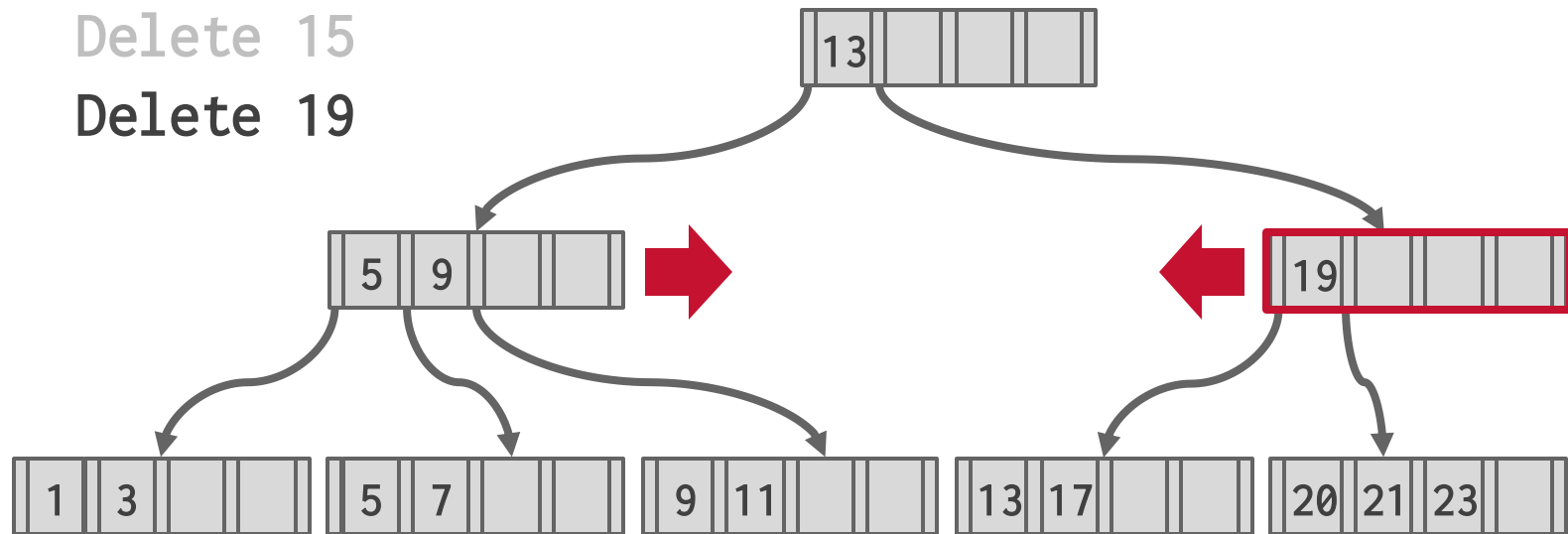


**Under-filled!**  
**No "rich" sibling nodes to borrow.**  
**Merge with a sibling**

# B+Tree – DELETE Example (3)

Delete 15

Delete 19



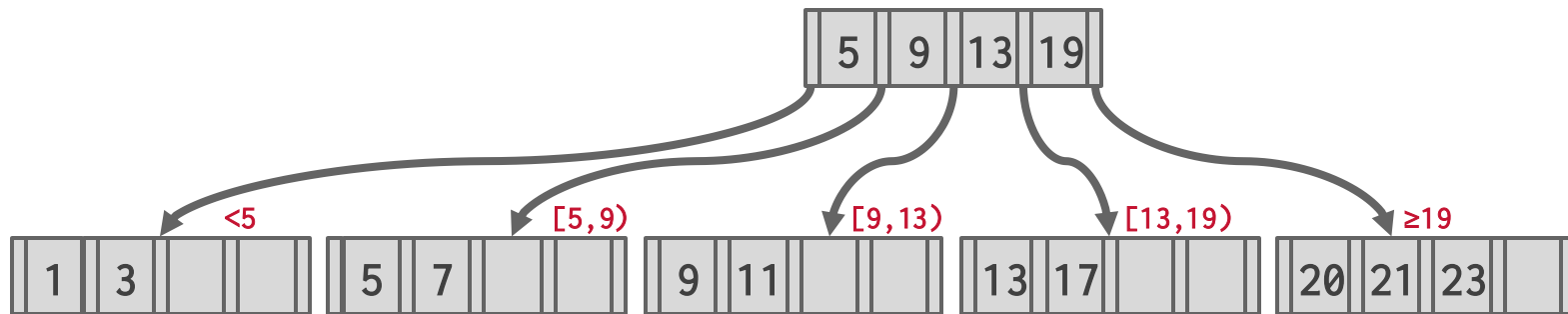
*This node is  
under-filled!  
Pull-down.*

# B+Tree – DELETE Example (3)

Delete 15

Delete 19

*The tree has shrunk in height.*

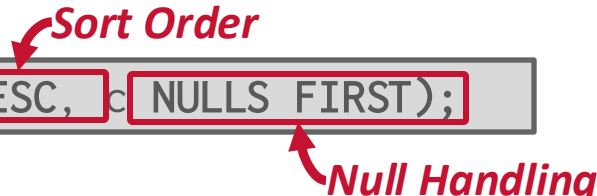


# Composite Index

A **composite index** is when the key is comprised of two or more attributes.

→ Example: Index on **<a,b,c>**

```
CREATE INDEX my_idx ON xxx (a, b DESC, c NULLS FIRST);
```



DBMS can use B+Tree index if the query provides a “prefix” of composite key.

→ Supported: **(a=1 AND b=2 AND c=3)**

→ Supported: **(a=1 AND b=2)**

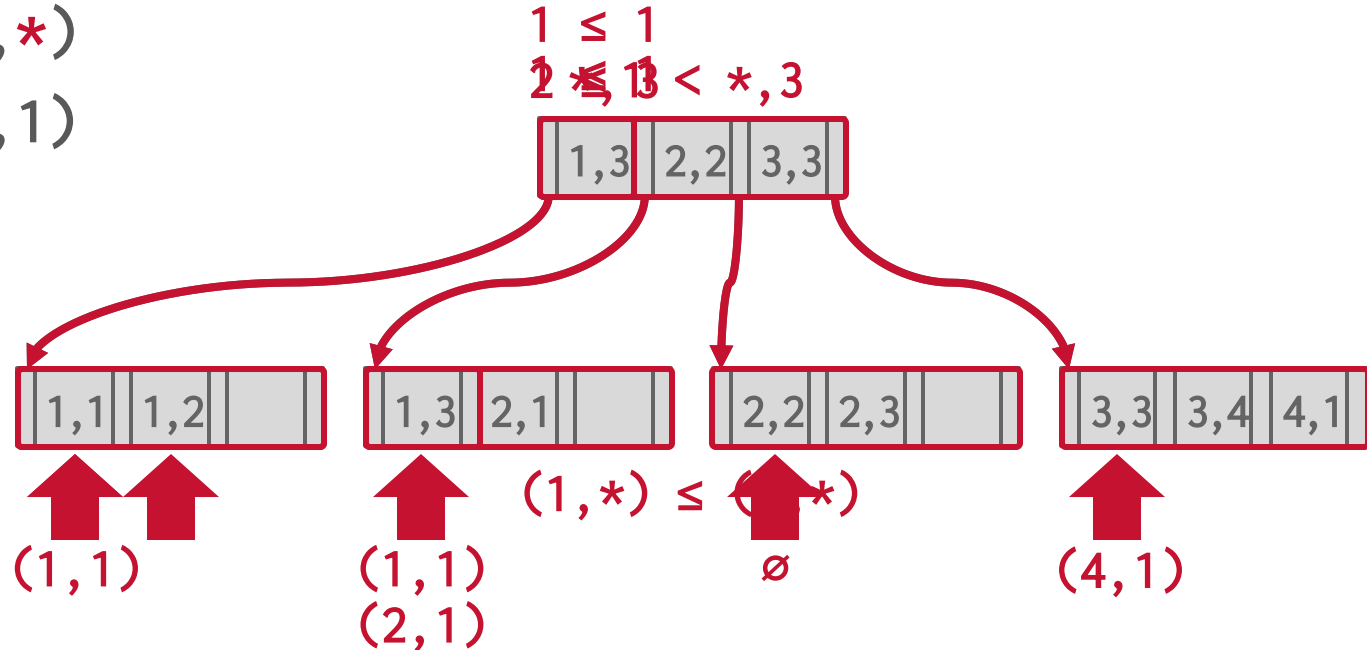
→ Rarely Supported: **(b=2), (c=3)**

# Selection Conditions

Find Key=(1,2)

Find Key=(1,\*)

Find Key=(\*,1)



# B+Tree – Duplicate Keys

## **Approach #1: Append Record ID**

- Add the tuple's unique Record ID as part of the key to ensure that all keys are unique.
- The DBMS can still use partial keys to find tuples.

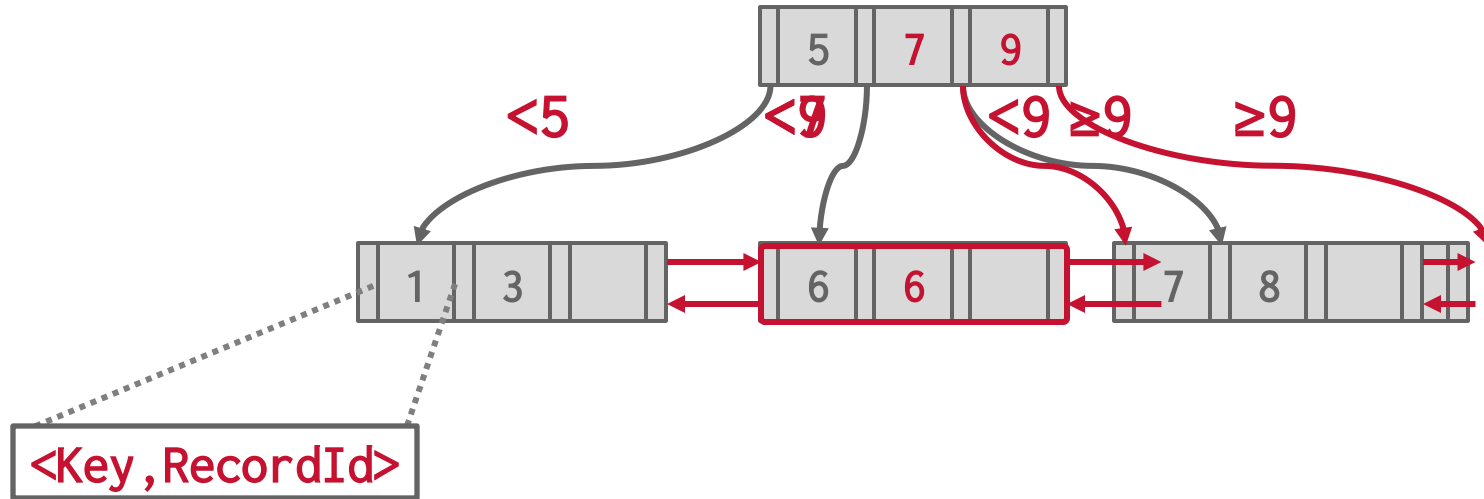
## **Approach #2: Overflow Leaf Nodes**

- Allow leaf nodes to spill into overflow nodes that contain the duplicate keys.
- This is more complex to maintain and modify.



# B+Tree – Append Record ID

Insert ~~6~~6, (Page, Slot)>

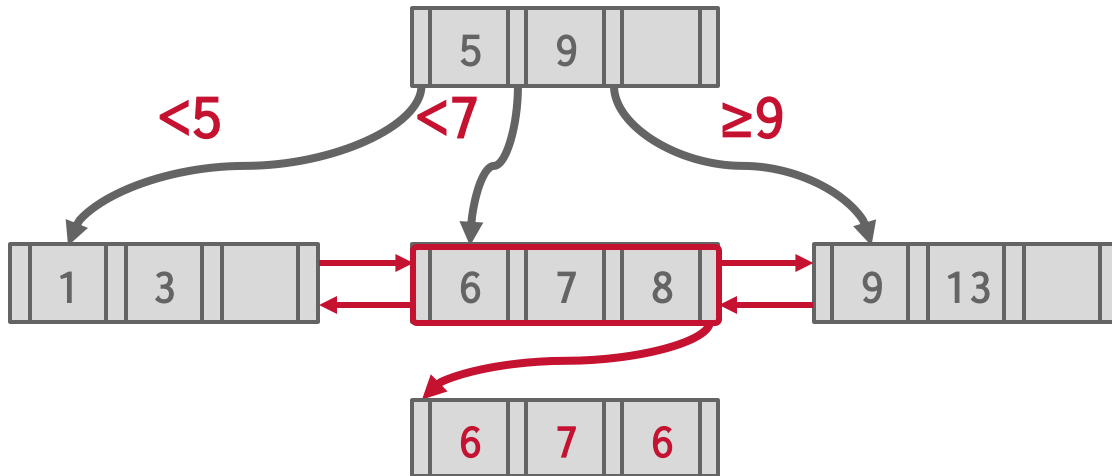


# B+Tree – Overflow Leaf Nodes

Insert 6

Insert 7

Insert 6



# Clustered Indexes

The table is stored in the sort order specified by the primary key.

→ Can be either heap- or index-organized storage.

Some DBMSs always use a clustered index.

→ If a table does not contain a primary key, the DBMS will automatically make a hidden primary key.

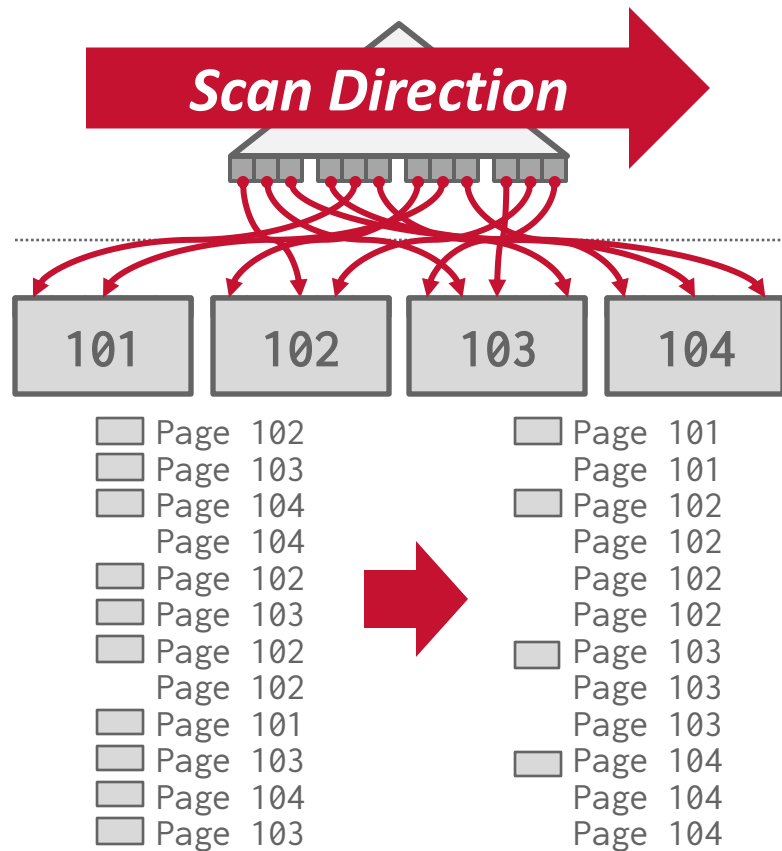
Other DBMSs cannot use them at all.

# Index Scan Page Sorting

Retrieving tuples in the order they appear in a is generally inefficient due to redundant reads.

A better approach is to find all the tuples that the query needs and then sort them based on their page ID.

The DBMS retrieves each page once.



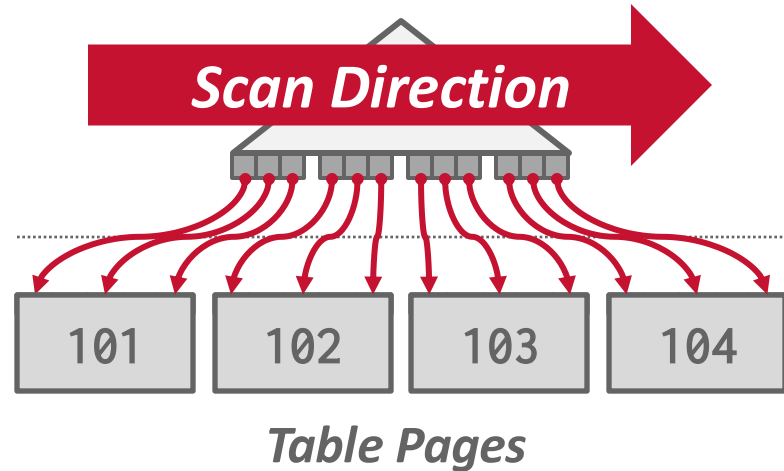
# Clustered Indexes

An extreme solution is "index organized storage"

- Store pages in sorted order also

Traverse to the left-most leaf page and then retrieve tuples from all leaf pages.

This will always be better than sorting data for each query.



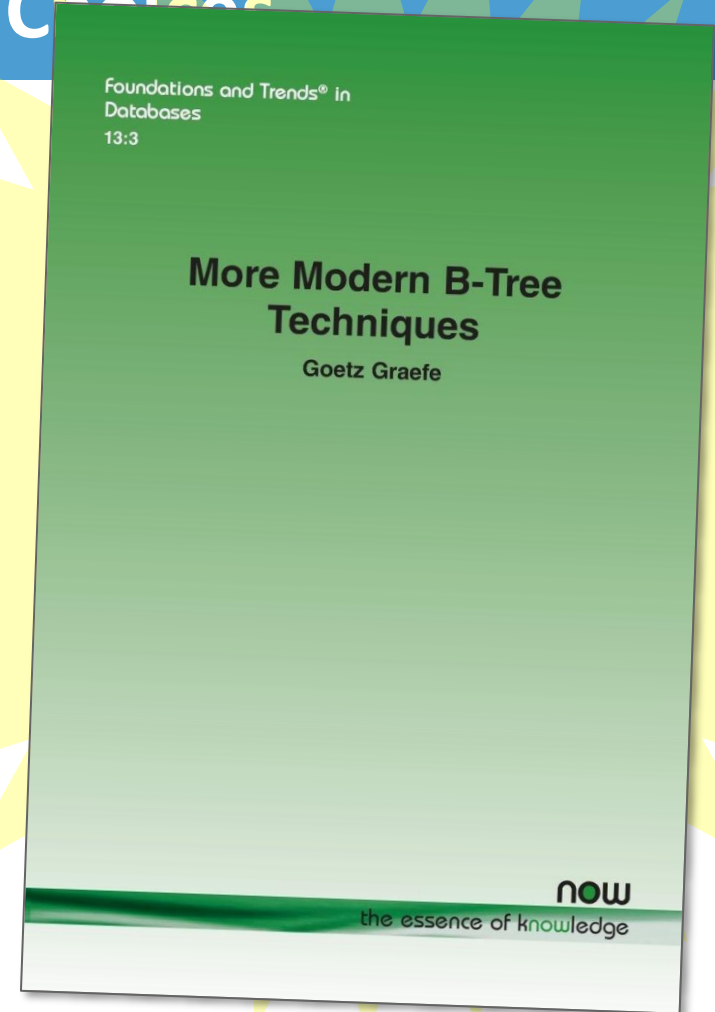
# B+Tree Design Choices

Node Size

Merge Threshold

Variable-Length Keys

Intra-Node Search



# Node Size

The slower the storage device, the larger the optimal node size for a B+Tree.

→ HDD: ~1MB

→ SSD: ~10KB

→ In-Memory: ~512B

Optimal sizes can vary depending on the workload

→ Leaf Node Scans vs. Root-to-Leaf Traversals

# Merge Threshold

Some DBMSs do not always merge nodes when they are half full.

→ Average occupancy rate for B+Tree nodes is 69%.

Delaying a merge operation may reduce the amount of reorganization.

It may also be better to let smaller nodes exist and then periodically rebuild entire tree.

This is why PostgreSQL calls their B+Tree a "non-balanced" B+Tree ([nbtree](#)).



# Variable-length Keys

## Approach #1: Pointers

- Store the keys as pointers to the tuple's attribute.
- Also called T-Trees (in-memory DBMSs)

## Approach #2: Variable-Length Nodes

- The size of each node in the index can vary.
- Requires careful memory management.

## Approach #3: Padding

- Always pad the key to be max length of the key type.

## Approach #4: Key Map / Indirection

- Embed an array of pointers that map to the key + value list within the node.

# Intra-Node Search

## Approach #1: Linear

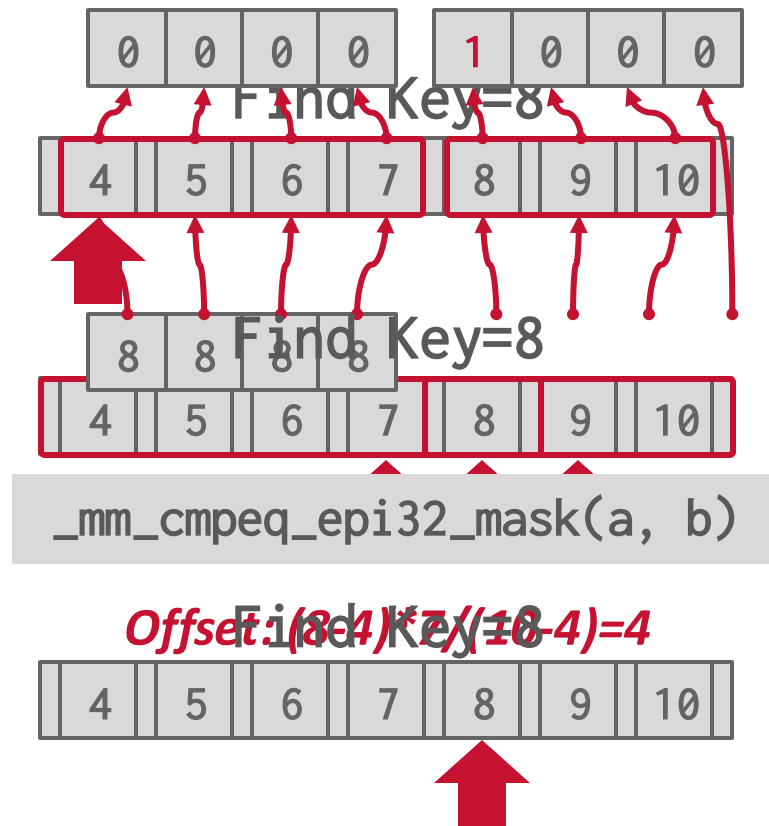
- Scan node keys from beginning to end.
- Use SIMD to vectorize comparisons.

## Approach #2: Binary

- Jump to middle key, pivot left/right depending on comparison.

## Approach #3: Interpolation

- Approximate location of desired key based on known distribution of keys.



# Optimizations

Prefix Compression

Deduplication

Suffix Truncation

Pointer Swizzling

Bulk Insert

Buffered Updates

Many more...

# Prefix Compression

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.

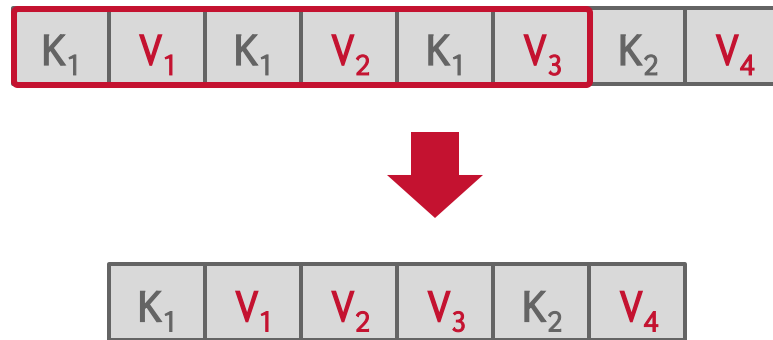
→ Many variations.



# Deduplication

Non-unique indexes can end up storing multiple copies of the same key in leaf nodes.

The leaf node can store the key once and then maintain a "posting list" of tuples with that key.

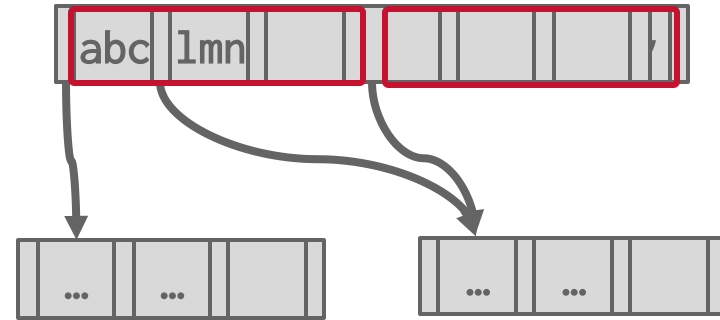


# Suffix Truncation

The keys in the inner nodes are only used to "direct traffic".

→ We don't need the entire key.

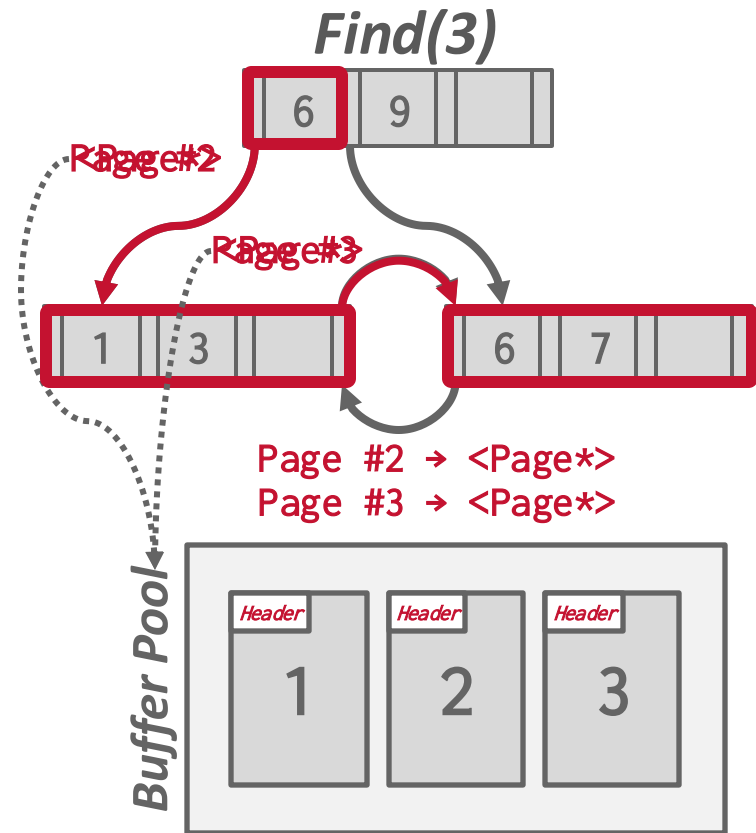
Store a minimum prefix that is needed to correctly route probes into the index.



# Pointer Swizzling

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.

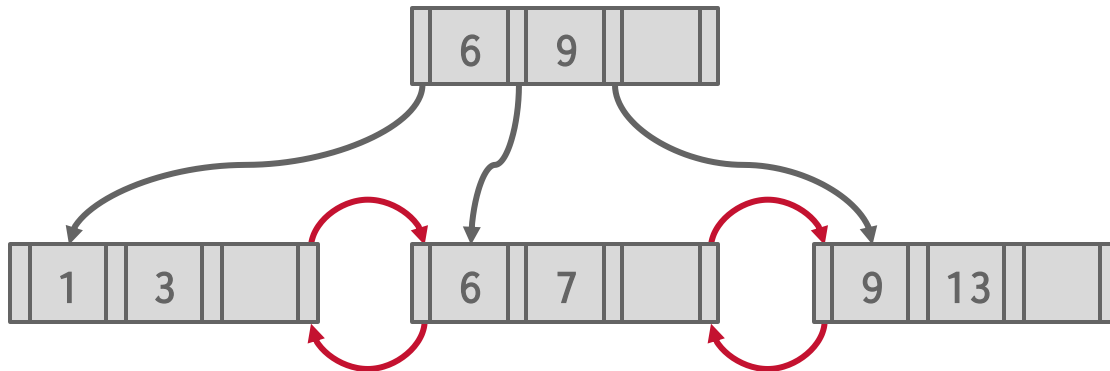


# Bulk Insert

The fastest way to build a new B+Tree for an existing table is to first sort the keys and then build the index from the bottom up.

Keys: 3, 7, 9, 13, 6, 1

**Sorted Keys: 1, 3, 6, 7, 9, 13**





# Observation

Modifying a B+tree is expensive when the DBMS has to split/merge nodes.

- Worst case is when DBMS reorganizes the entire tree.
- The worker that causes a split/merge is responsible for doing the work.

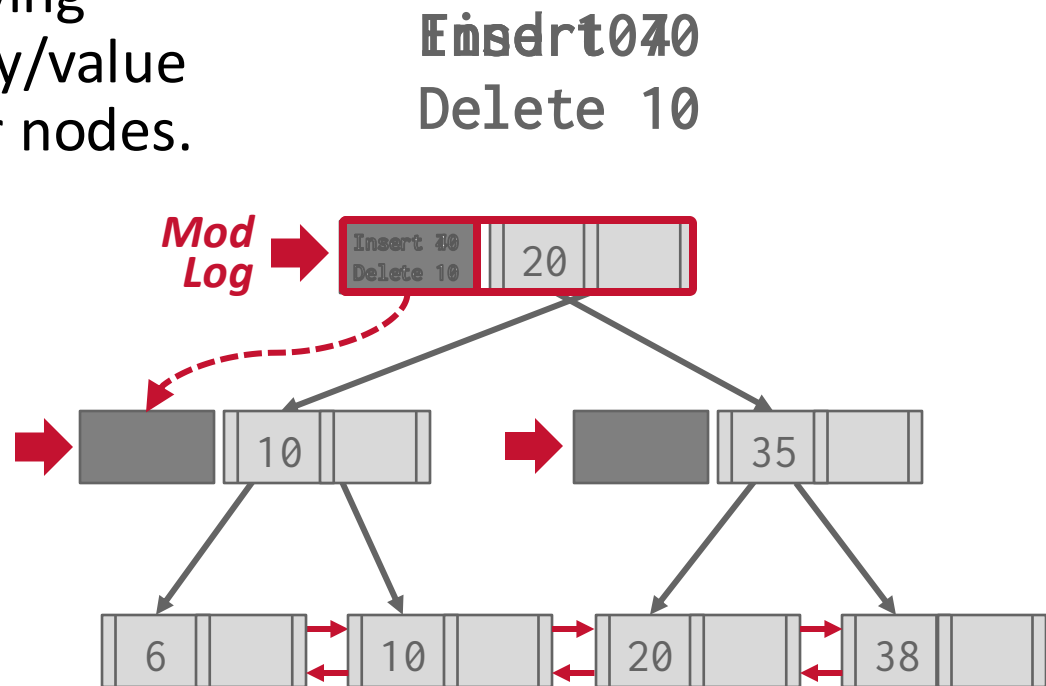
What if there was a way to delay updates and then apply multiple changes together in a batch?

# Write-Optimized B+Tree (B $\epsilon$ -Tree)

Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes.

→ aka **Fractal Trees** / **B $\epsilon$ -trees**.

Updates cascade down to lower nodes incrementally when buffers get full.



Tokute

DB

Rela

moDB

STS



# Conclusion

The venerable B+Tree is (almost) always a good choice for your DBMS.

# Next Week

Zhongrui takes over for 2 classes

More algorithms and data structures used inside the DBMS

- Hash Tables

- Sorting (external sort)