香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

SCHOOL OF
DATA SCIENCE
數據科學學院

Ack: Prof. Jignesh Patel @ CMU
Prof. Andy Pavlo @CMU

# CSC3170
# 8: B+Tree Indexes

Chenhao Ma
School of Data Science
The Chinese University of Hong Kong, Shenzhen

# Last Lecture

- Hash tables are important data structures that are used all throughout a DBMS.
  - Space Complexity: $\boldsymbol{O(n)}$
  - Average Time Complexity: $\boldsymbol{O(1)}$
- Static vs. Dynamic Hashing schemes

- DBMSs use mostly hash tables for their internal data structures.

# This Lecture

- B+Tree Overview
- Design Choices
- Optimizations

# B+Tree

# B-Tree Family

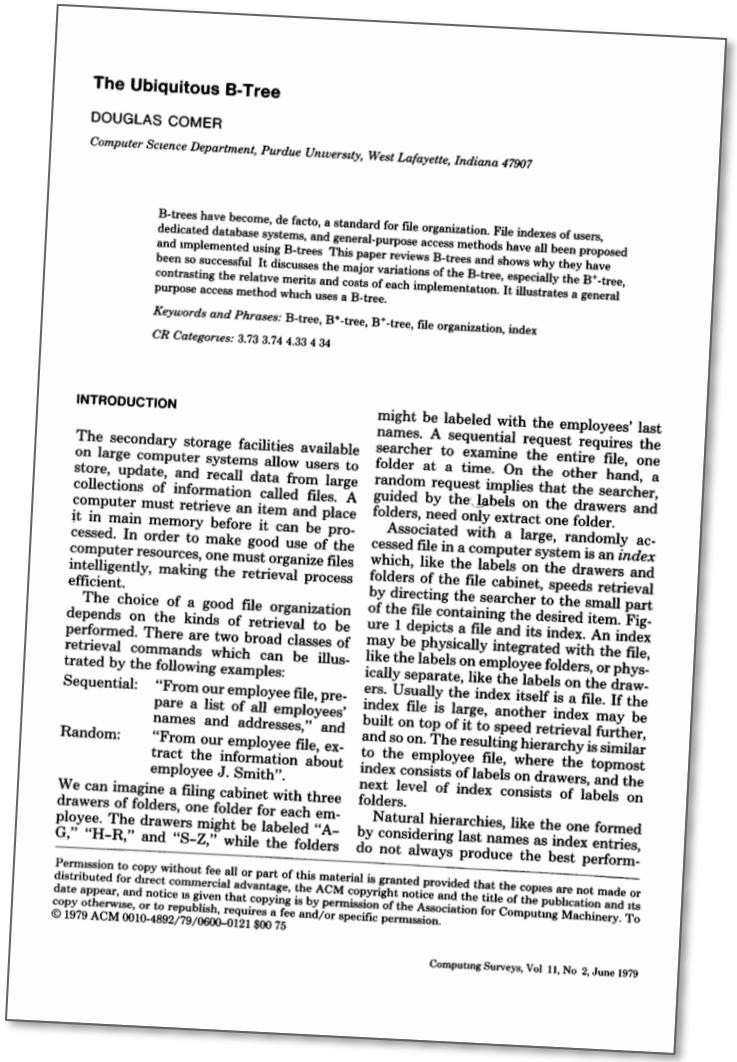
- There is a specific data structure called a **B-Tree**.

- People also use the term to generally refer to a class of balanced tree data structures:
  - **B-Tree** (1971)
  - **B+Tree** (1973)
  - **B*Tree** (1977?)
  - **B$^{link}$-Tree** (1981)
  - **Bε-Tree** (2003)
  - **Bw-Tree** (2013)

# B-Tree Family

- There is a specific data structure called a **B-Tree**.

- People also use the term to generally refer to a class of balanced tree data structures:
  - **B-Tree** (1971)
  - **B+Tree** (1973)
  - **B*Tree** (1977?)
  - **B$^{link}$-Tree** (1981)
  - **Bε-Tree** (2003)
  - **Bw-Tree** (2013)



The Ubiquitous B-Tree

DOUGLAS COMER

*Computer Science Department, Purdue University, West Lafayette, Indiana 47907*

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

*Keywords and Phrases:* B-tree, B*-tree, B+-tree, file organization, index

*CR Categories:* 3.73 3.74 4.33 4 34

INTRODUCTION

The secondary storage facilities available on large computer systems allow users to store, update, and recall data from large collections of information called files. A computer must retrieve an item and place it in main memory before it can be processed. In order to make good use of the computer resources, one must organize files intelligently, making the retrieval process efficient.

The choice of a good file organization depends on the kinds of retrieval to be performed. There are two broad classes of retrieval commands which can be illustrated by the following examples:

Sequential: "From our employee file, prepare a list of all employees' names and addresses," and

Random: "From our employee file, extract the information about employee J. Smith".

We can imagine a filing cabinet with three drawers of folders, one folder for each employee. The drawers might be labeled "A–G," "H–R," and "S–Z," while the folders 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 perform-

# B-Tree Family

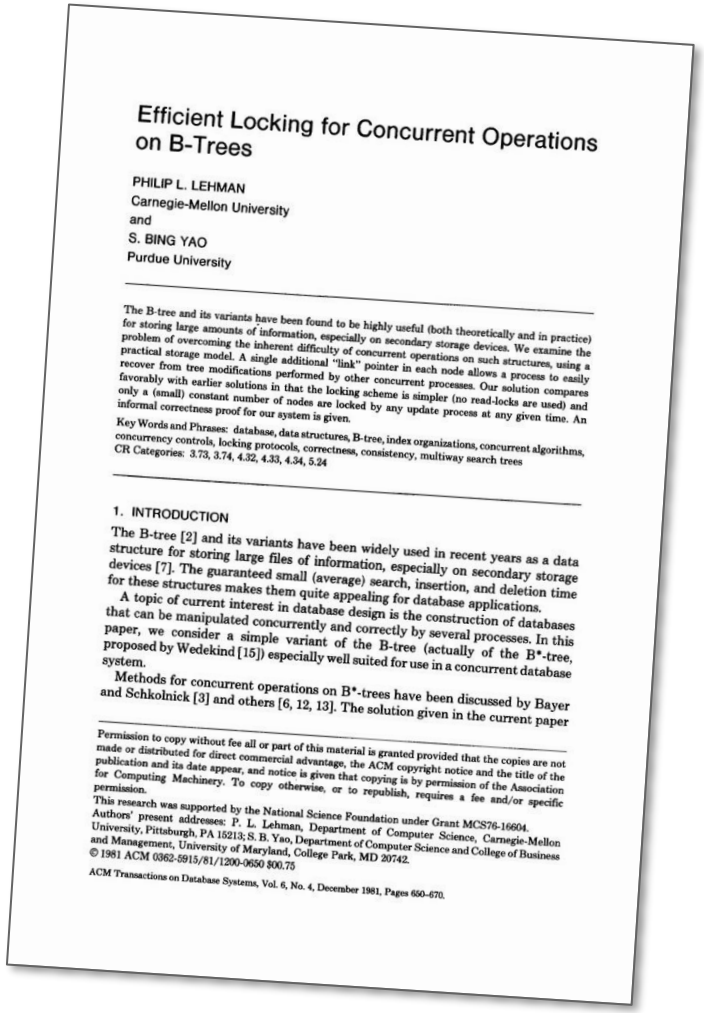
- There is a specific data structure called a **B-Tree**.

- People also use the term to generally refer to a class of balanced tree data structures:
  - **B-Tree** (1971)
  - **B+Tree** (1973)
  - **B*Tree** (1977?)
  - **B$^{link}$-Tree** (1981)
  - **Bε-Tree** (2003)
  - **Bw-Tree** (2013)

# B-Tree Family

- There is a specific data structure called a **B-Tree**.

- People also use the term to generally refer to a class of balanced tree data structures:
  - **B-Tree** (1971)
  - **B+Tree** (1973)
  - **B*Tree** (1977?)
  - **B$^{link}$-Tree** (1981)
  - **Bε-Tree** (2003)
  - **Bw-Tree** (2013)



Efficient Locking for Concurrent Operations on B-Trees

PHILIP L. LEHMAN
Carnegie-Mellon University
and
S. BING YAO
Purdue University

The B-tree and its variants have been found to be highly useful (both theoretically and in practice) for storing large amounts of information, especially on secondary storage devices. We examine the problem of overcoming the inherent difficulty of concurrent operations on such structures, using a practical storage model. A single additional "link" pointer in each node allows a process to easily recover from tree modifications performed by other concurrent processes. Our solution compares favorably with earlier solutions in that the locking scheme is simpler (no read-locks are used) and only a (small) constant number of nodes are locked by any update process at any given time. An informal correctness proof for our system is given.

Key Words and Phrases: database, data structures, B-tree, index organizations, concurrent algorithms, concurrency controls, locking protocols, correctness, consistency, multiway search trees
CR Categories: 3.73, 3.74, 4.32, 4.33, 4.34, 5.24

1. INTRODUCTION

The B-tree [2] and its variants have been widely used in recent years as a data structure for storing large files of information, especially on secondary storage devices [7]. The guaranteed small (average) search, insertion, and deletion time for these structures makes them quite appealing for database applications.

A topic of current interest in database design is the construction of databases that can be manipulated concurrently and correctly by several processes. In this paper, we consider a simple variant of the B-tree (actually of the B*-tree, proposed by Wedekind [15]) especially well suited for use in a concurrent database system.

Methods for concurrent operations on B*-trees have been discussed by Bayer and Schkolnick [3] and others [6, 12, 13]. The solution given in the current paper

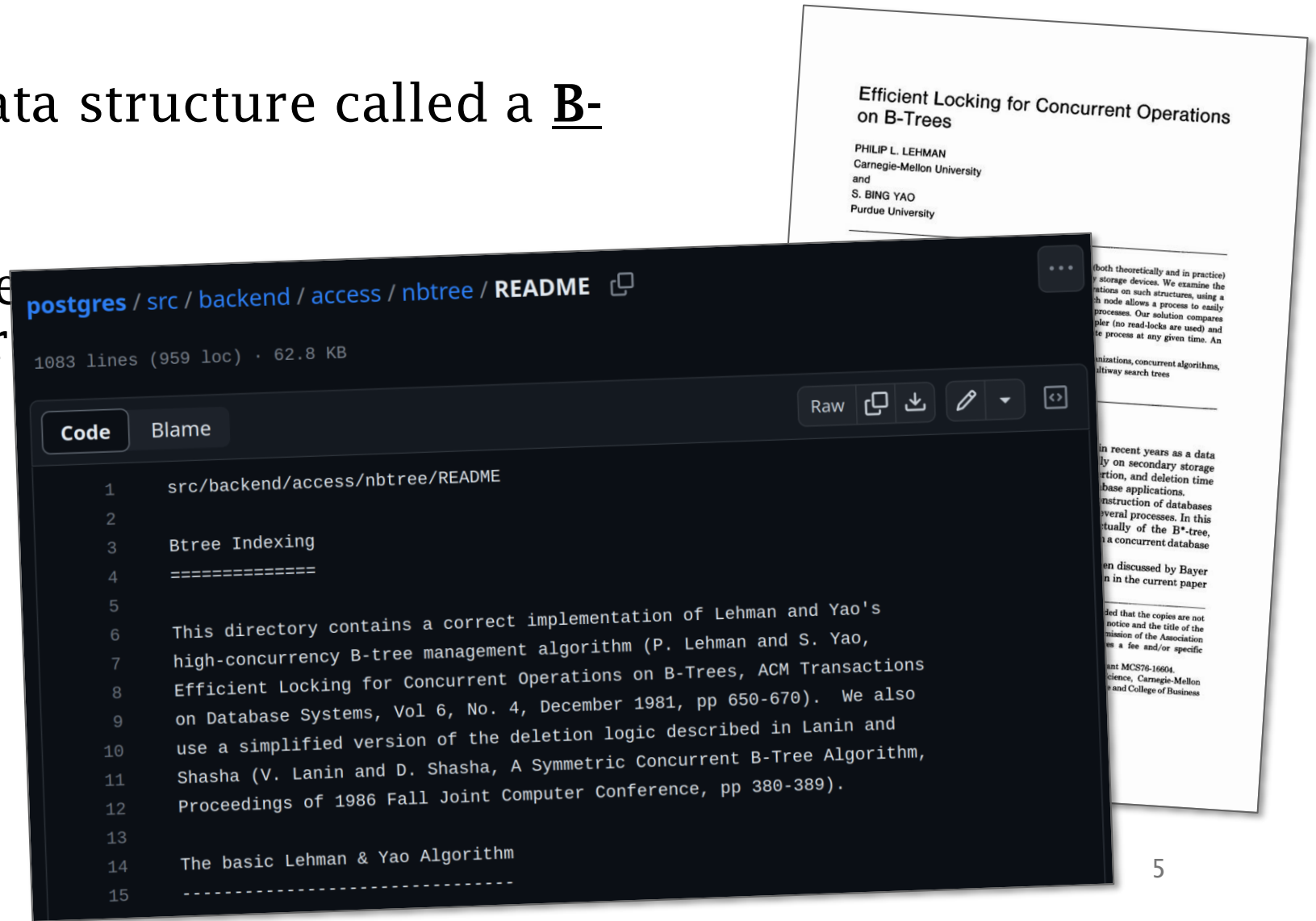

This research was supported by the National Science Foundation under Grant MCS76-16604.
Authors' present addresses: P. L. Lehman, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213; S. B. Yao, Department of Computer Science and College of Business and Management, University of Maryland, College Park, MD 20742.

ACM Transactions on Database Systems, Vol. 6, No. 4, December 1981, Pages 650–670.

# B-Tree Family

- There is a specific data structure called a **B-Tree**.

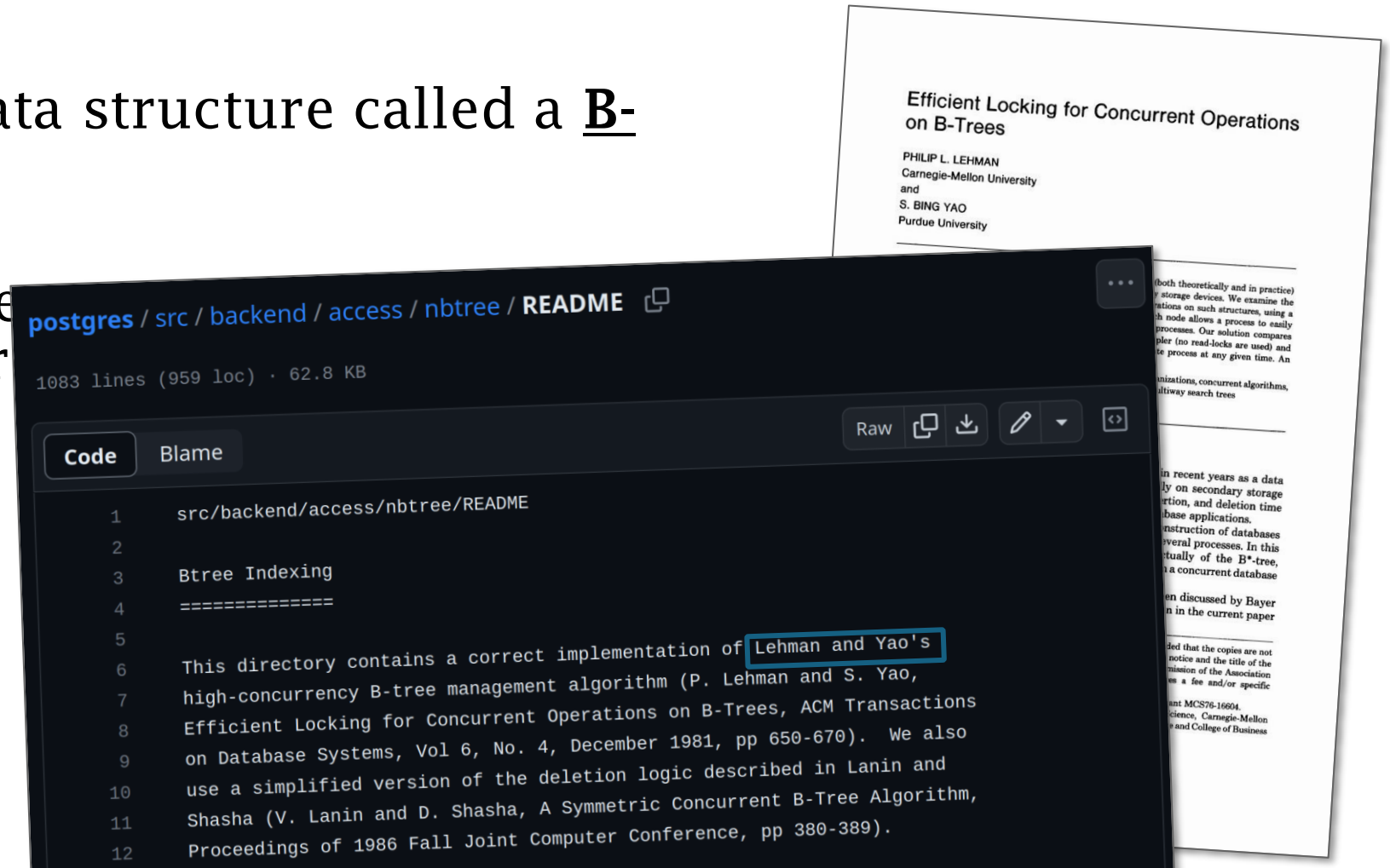
- People also use the te
  a class of balanced tr
  - **B-Tree** (1971)
  - **B+Tree** (1973)
  - **B*Tree** (1977?)
  - **B$^{link}$-Tree** (1981)
  - **Bε-Tree** (2003)
  - **Bw-Tree** (2013)



# B-Tree Family

- There is a specific data structure called a **B-Tree**.

- People also use the te
  a class of balanced tr
  - **B-Tree** (1971)
  - **B+Tree** (1973)
  - **B*Tree** (1977?)
  - **B$^{link}$-Tree** (1981)
  - **Bε-Tree** (2003)
  - **Bw-Tree** (2013)



# B-Tree Family

- There is a specific data structure called a **B-Tree**.

- People also use the term to generally refer to a class of balanced tree data structures:
  - **B-Tree** (1971)
  - **B+Tree** (1973)
  - **B*Tree** (1977?)
  - **B$^{link}$-Tree** (1981)
  - **Bε-Tree** (2003)
  - **Bw-Tree** (2013)

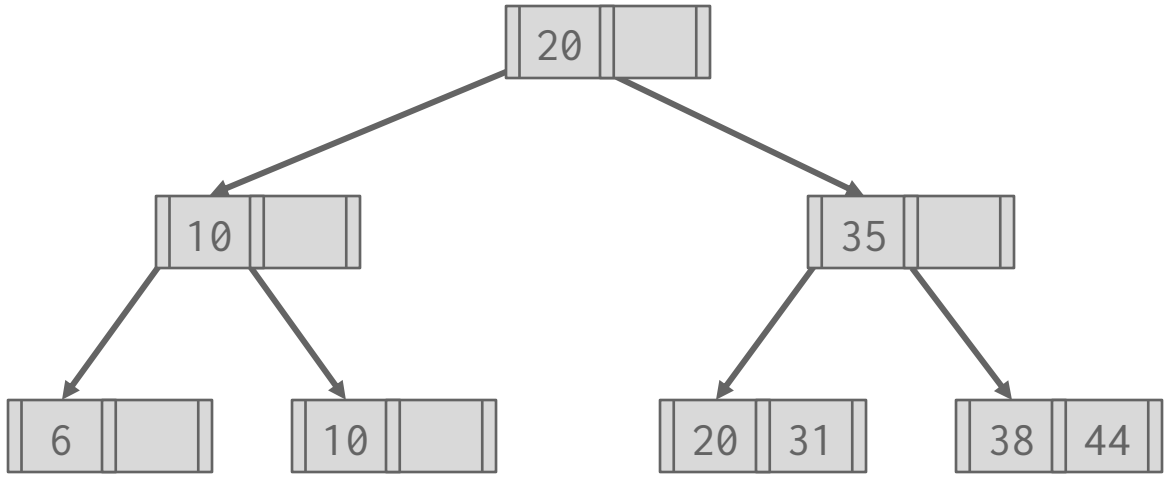
# B+Tree

- A **B+Tree** is a self-balancing, ordered tree data structure that allows searches, sequential access, insertions, and deletions in $O(\log_m n)$.
  - Generalization of a binary search tree, since a node can have more than two children.
  - Optimized for systems that read and write large blocks of data.
  - **m** is the fanout of the tree.

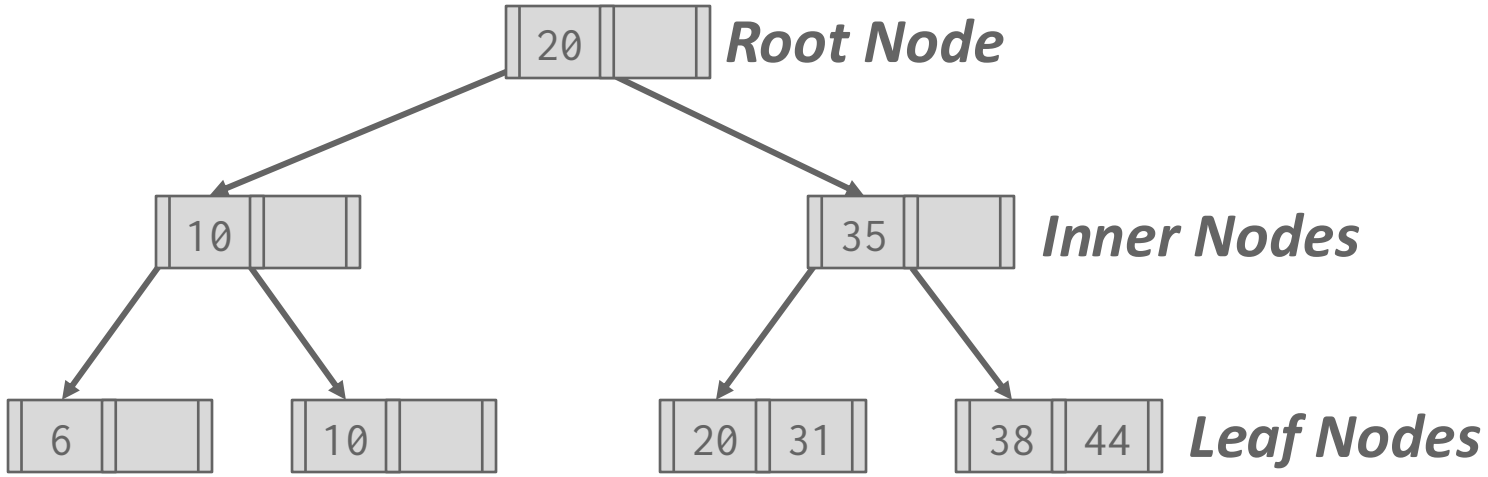
# B+Tree Properties

- 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 in the tree)
  - Every leaf node is at least half-full
    $\lceil (M-1)/2 \rceil \leq$ #keys $\leq M-1$
  - Every non-leaf node other than the root is at least half-full
    $\lceil M/2 \rceil - 1 \leq$ #keys $\leq M-1$
  - Every inner node with k keys has k+1 non-null children

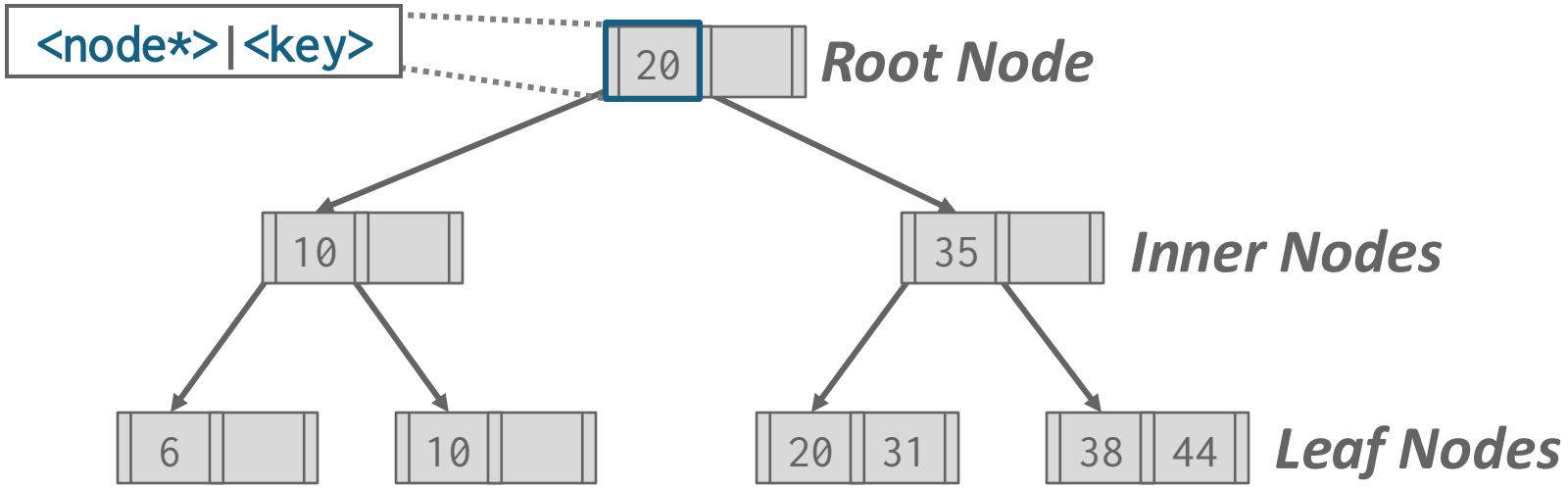
# B+Tree Example



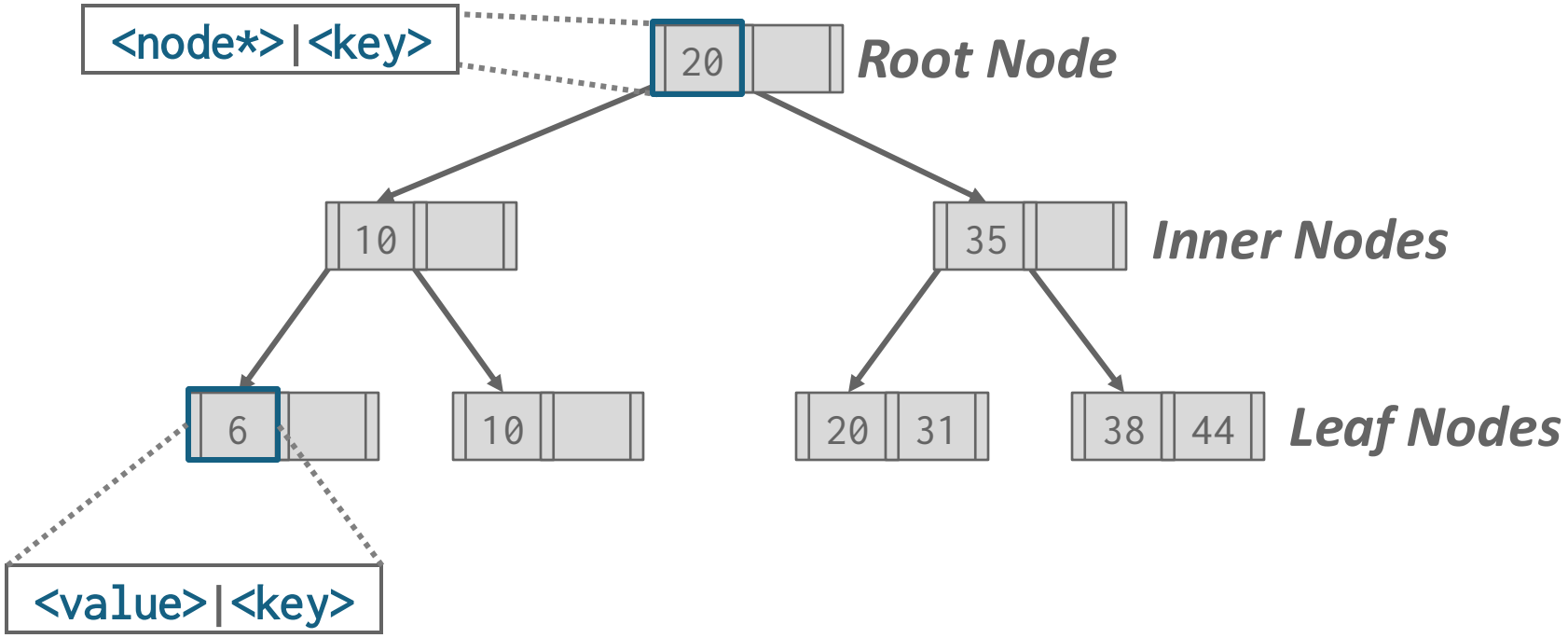
# B+Tree Example



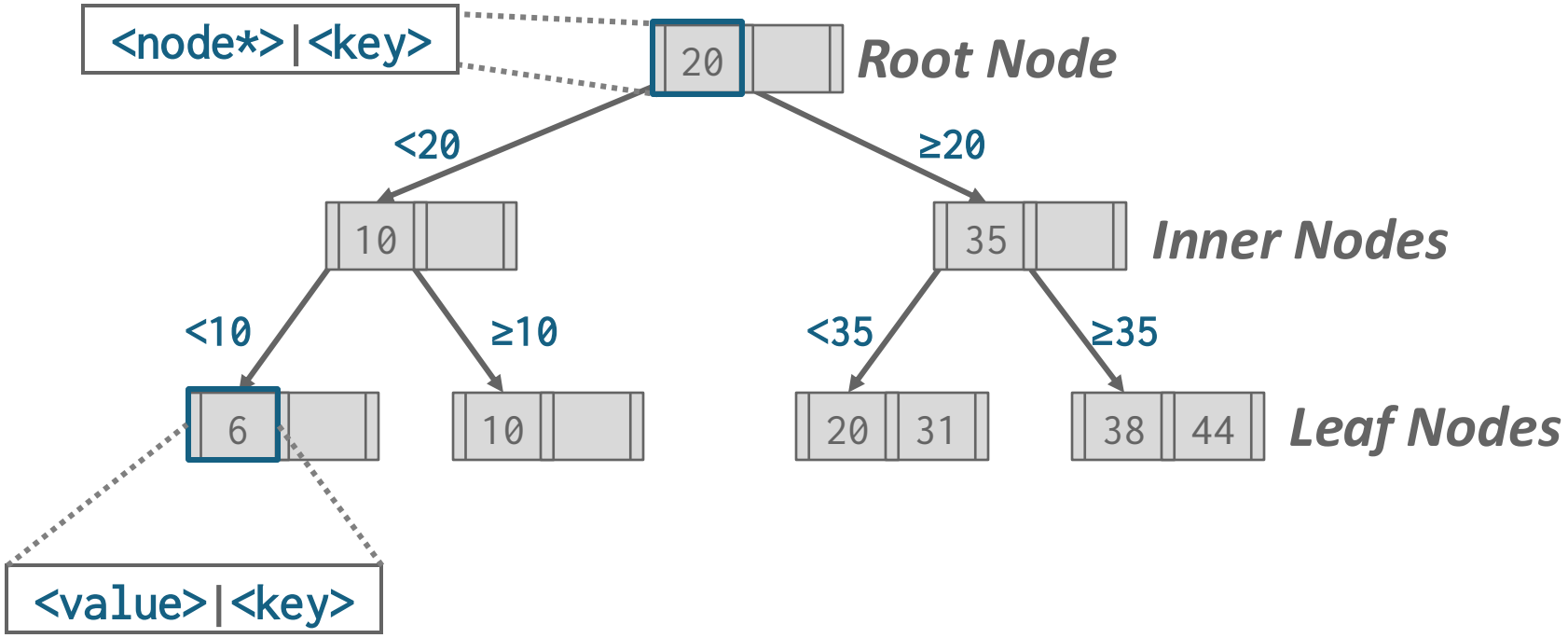
# B+Tree Example



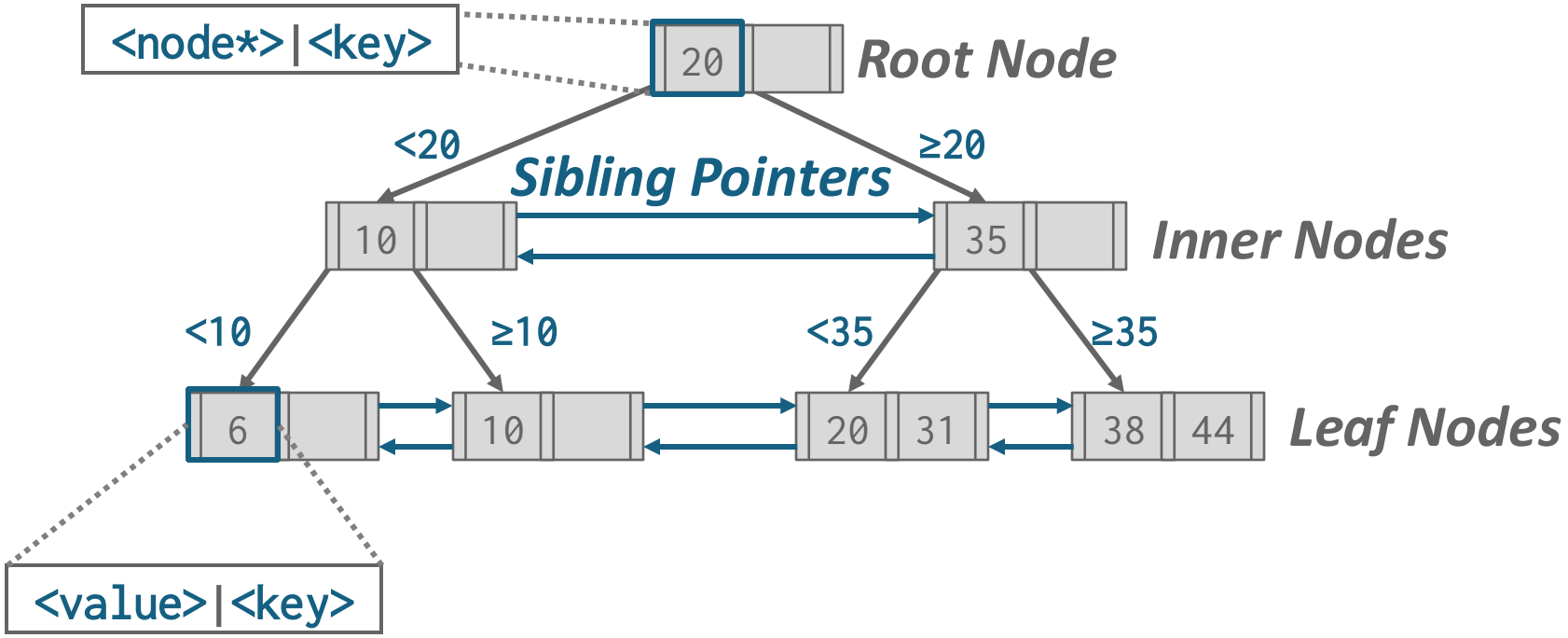
# B+Tree Example



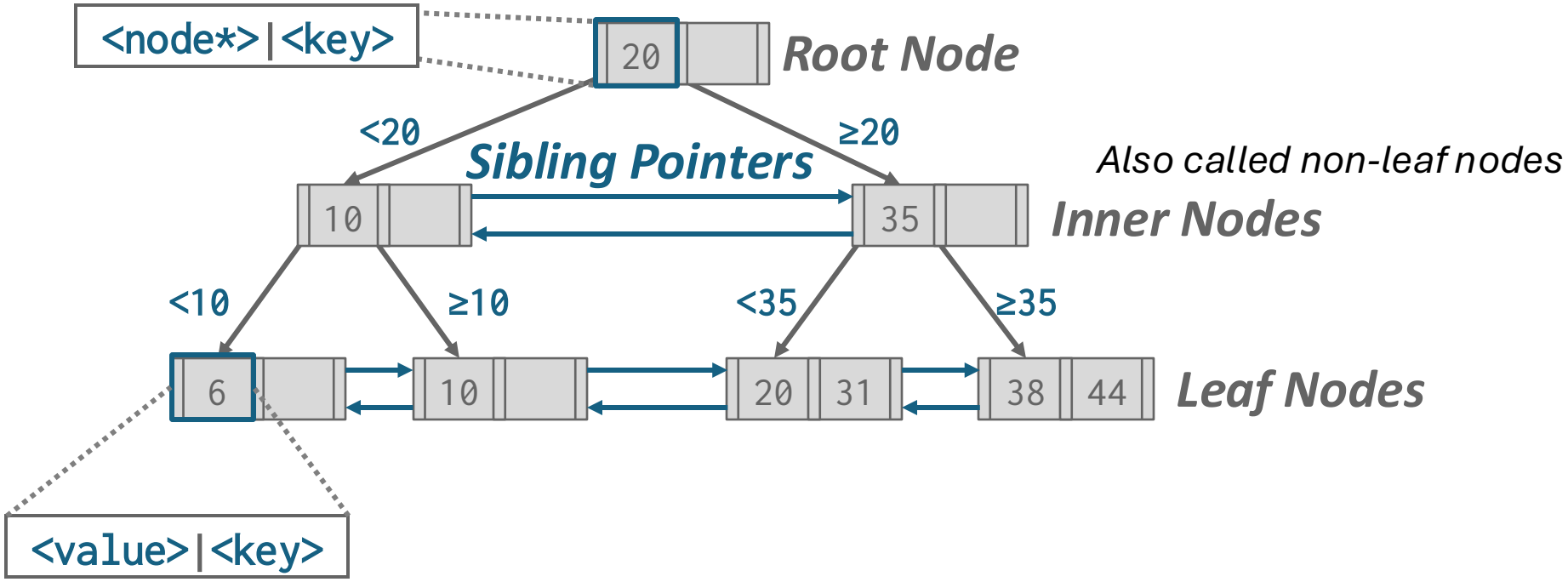
# B+Tree Example



# B+Tree Example



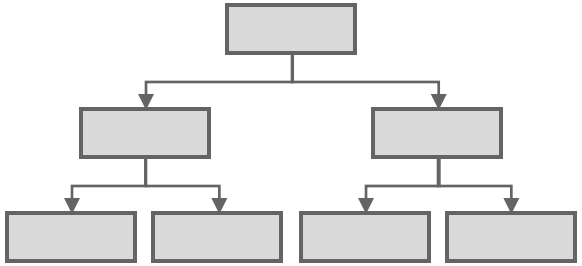
# B+Tree Example



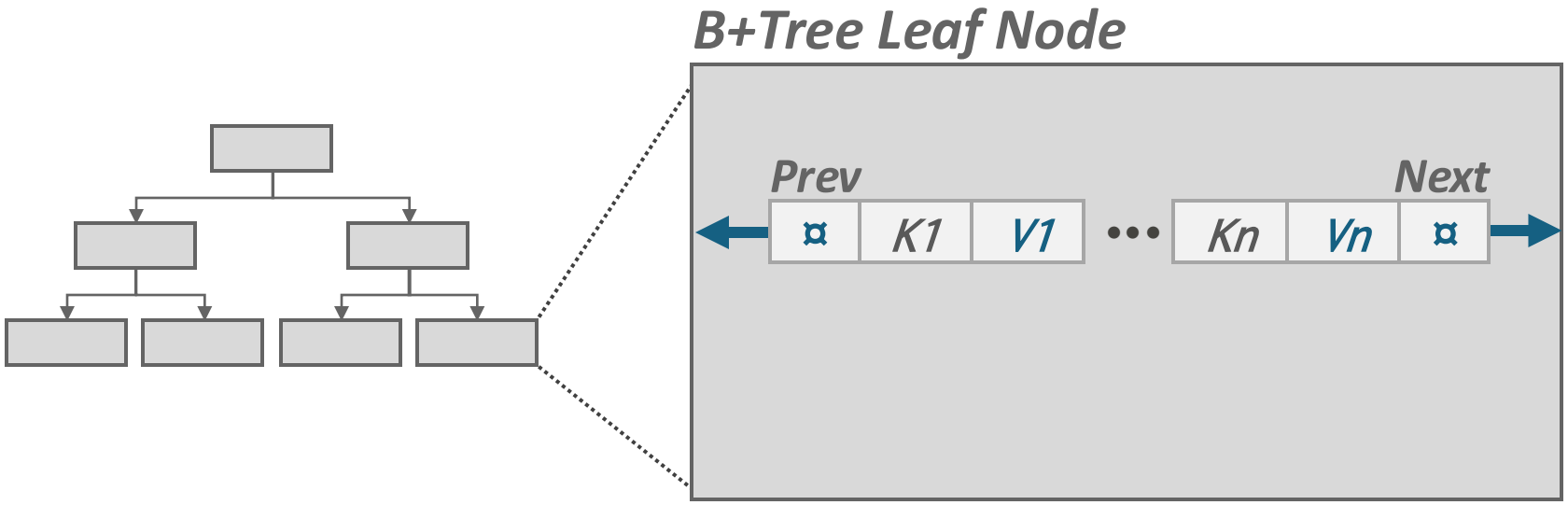
# Nodes

- Every B+Tree node is comprised of an array of key/value pairs.
  - The keys are derived from the attribute(s) that the index is based on.
  - The values 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.

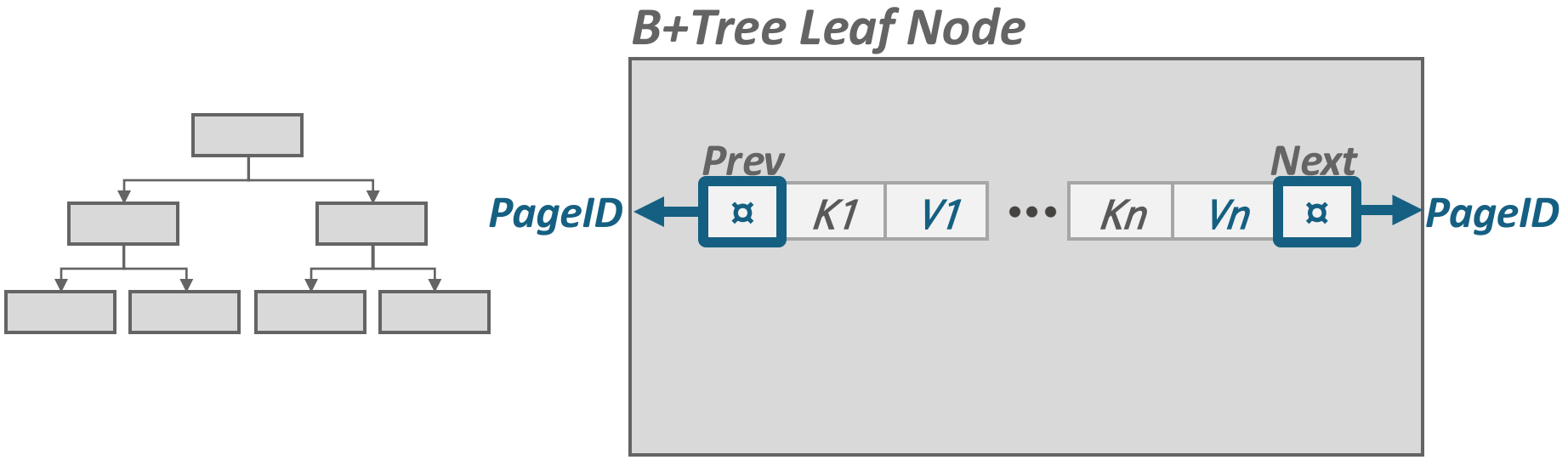
# B+Tree Leaf Nodes



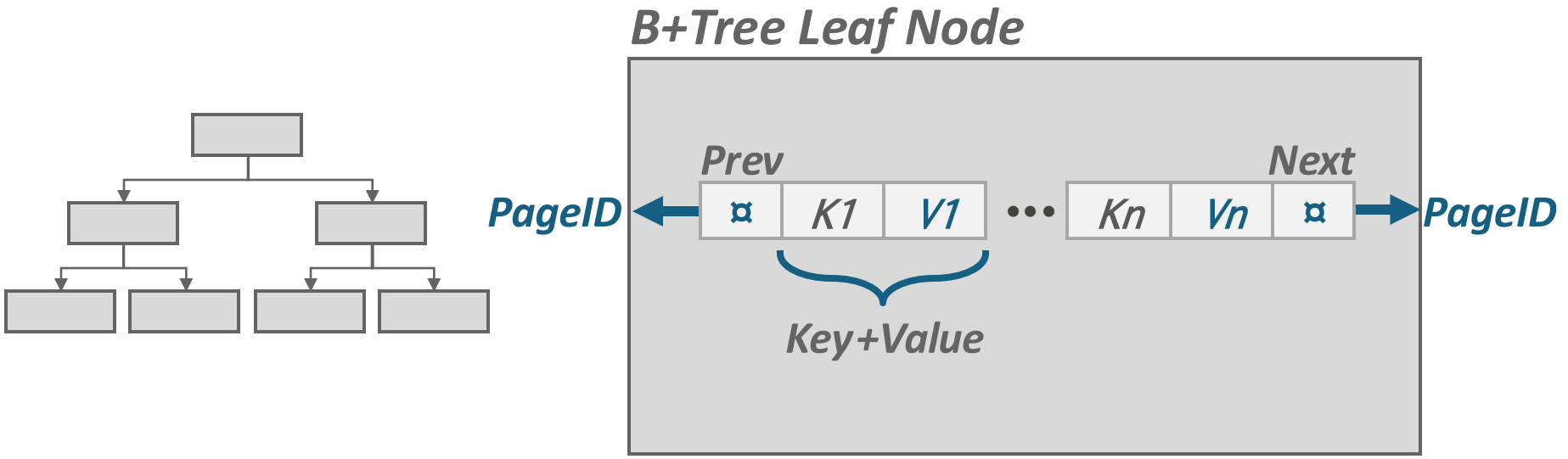
# B+Tree Leaf Nodes



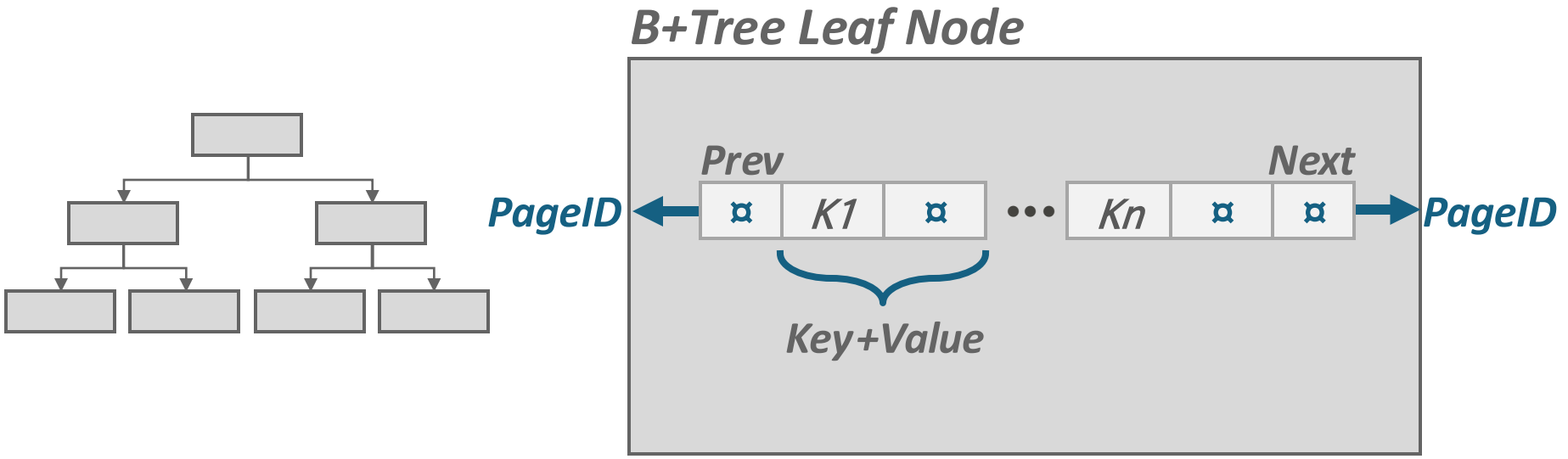
# B+Tree Leaf Nodes



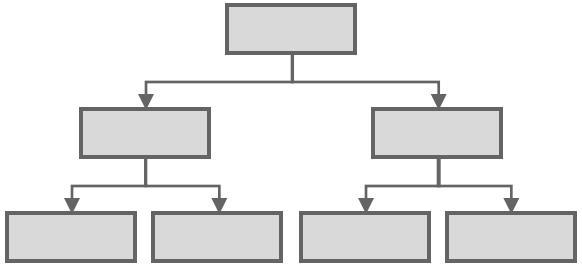
# B+Tree Leaf Nodes

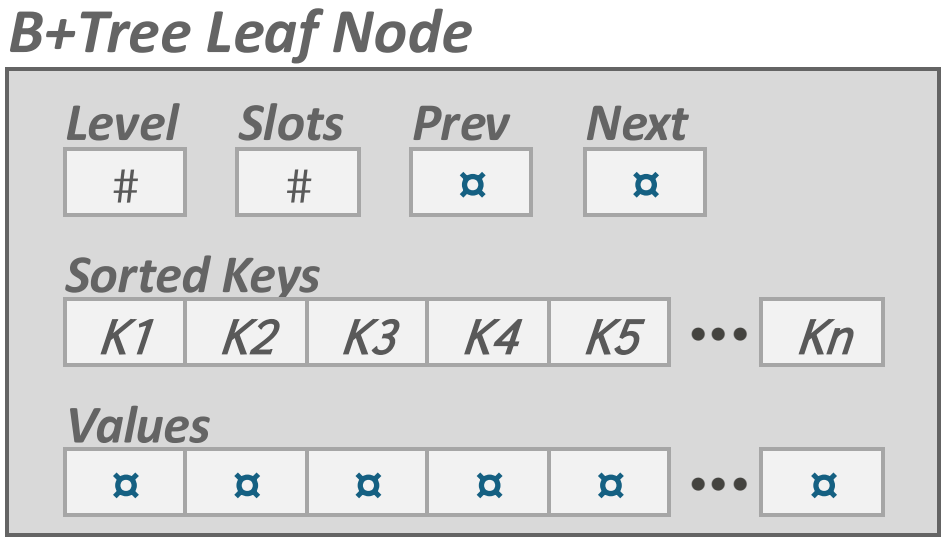


# B+Tree Leaf Nodes

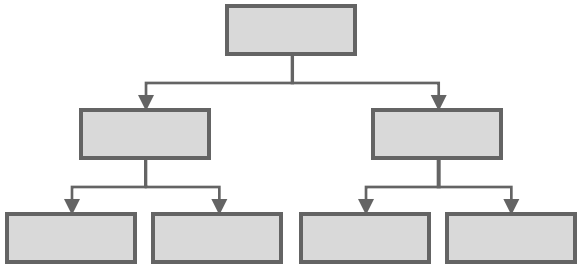


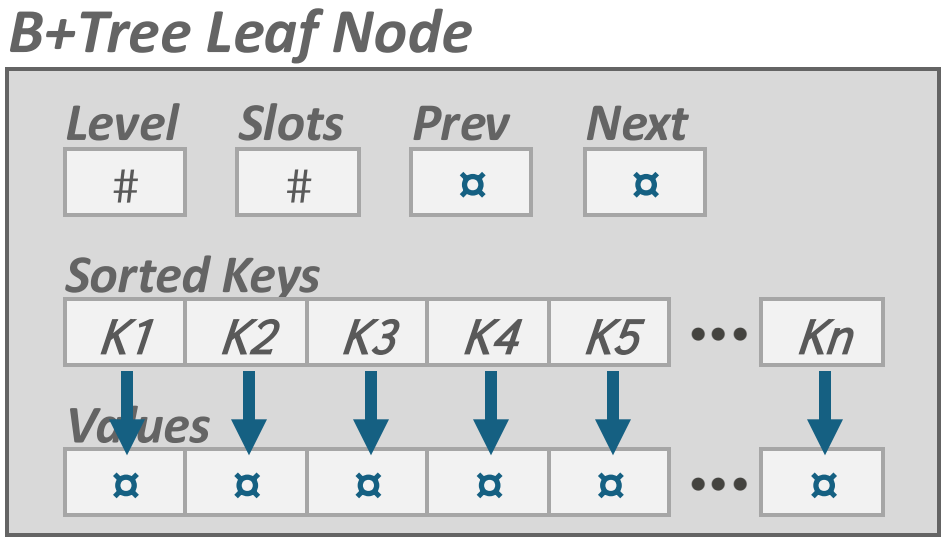
# B+Tree Leaf Nodes





# 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.
- **Approach #2: Tuple Data**
  - AKA Index-Organized Storage
  - The leaf nodes store the actual contents of the tuple.
  - Secondary indexes must store the Record ID as their values.

# Leaf Node Values

- **Approach #1: Record IDs**
  - A pointer to the location of the tuple to which the index entry corresponds.
- **Approach #2: Tuple Data**
  - AKA Index-Organized Storage
  - The leaf nodes store the actual contents of the tuple.
  - Secondary indexes must store the Record ID as their values.

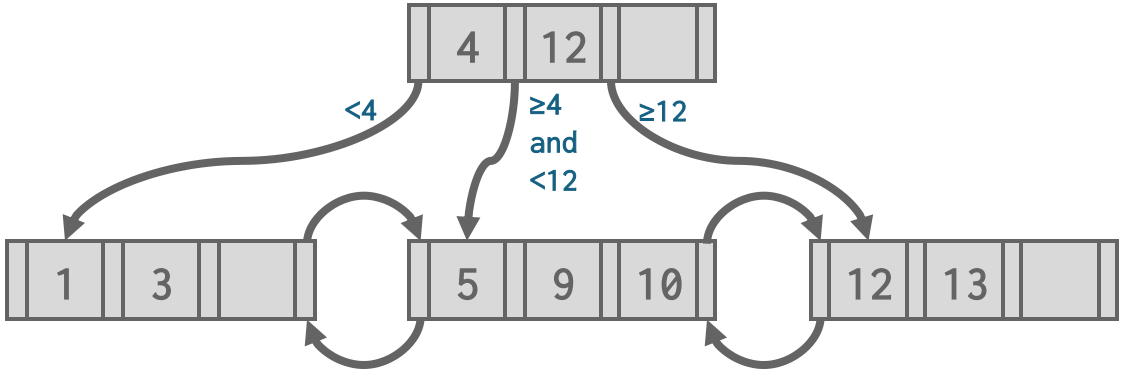
# B-Tree vs. B+Tree

- The original **B-Tree** from 1972 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.

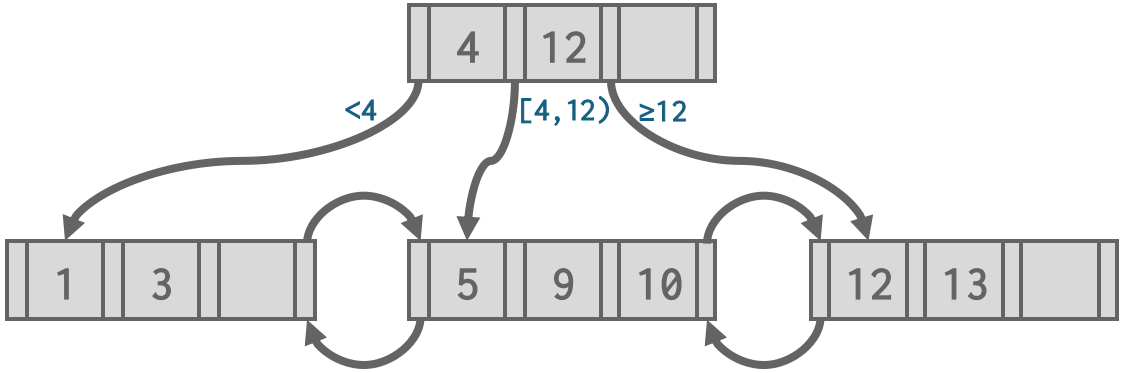
# B+Tree: Insert

- 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 L2
  - Redistribute entries evenly, copy up middle key.
  - Insert index entry pointing to L2 into parent of L.
- To split inner node, redistribute entries evenly, but push up middle key.

# B+Tree: Insert

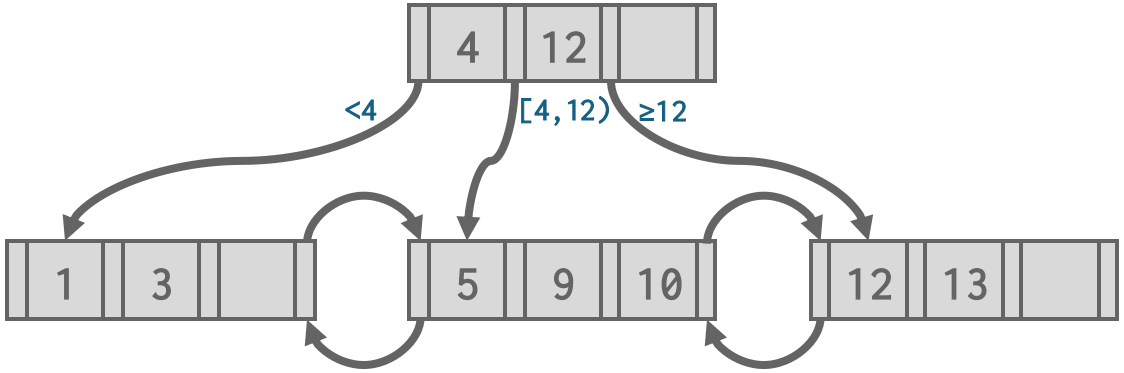


# B+Tree: Insert



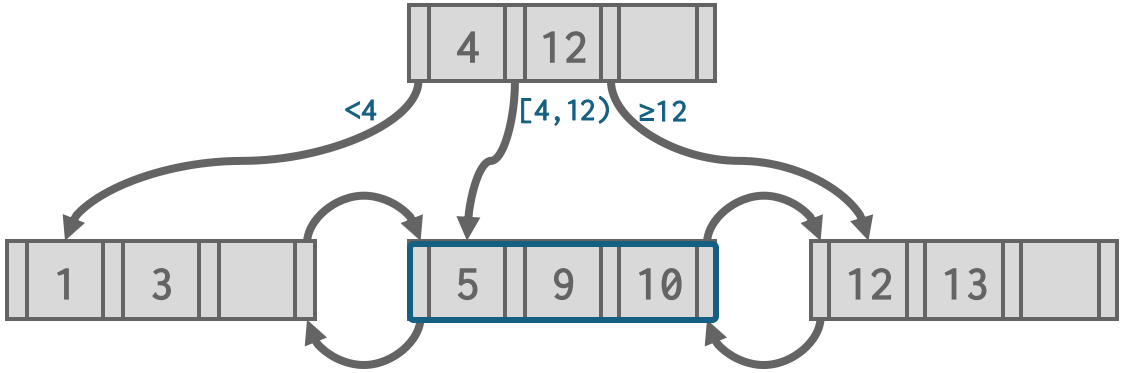
# B+Tree: Insert

Insert 6



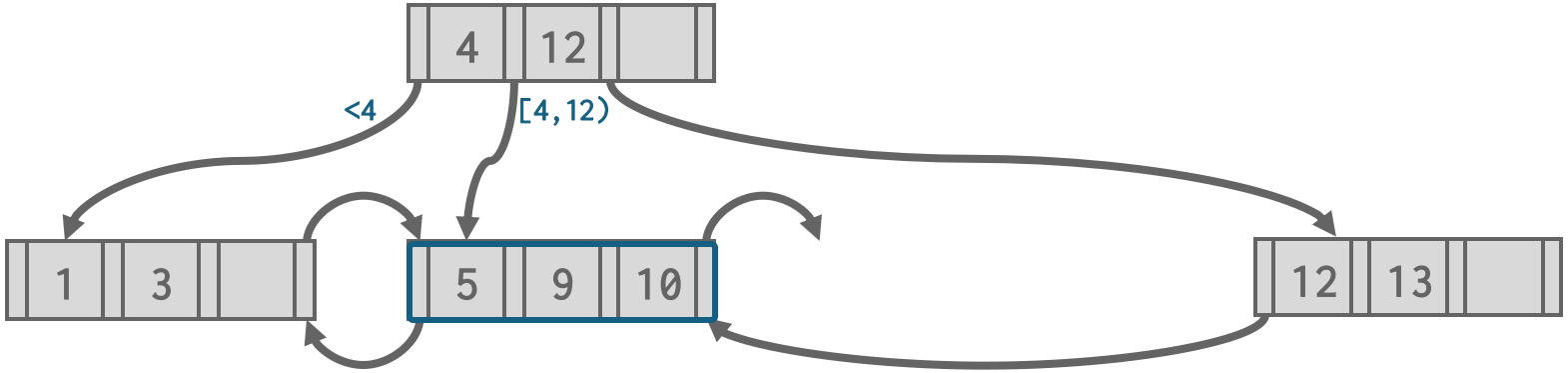
# B+Tree: Insert

Insert 6



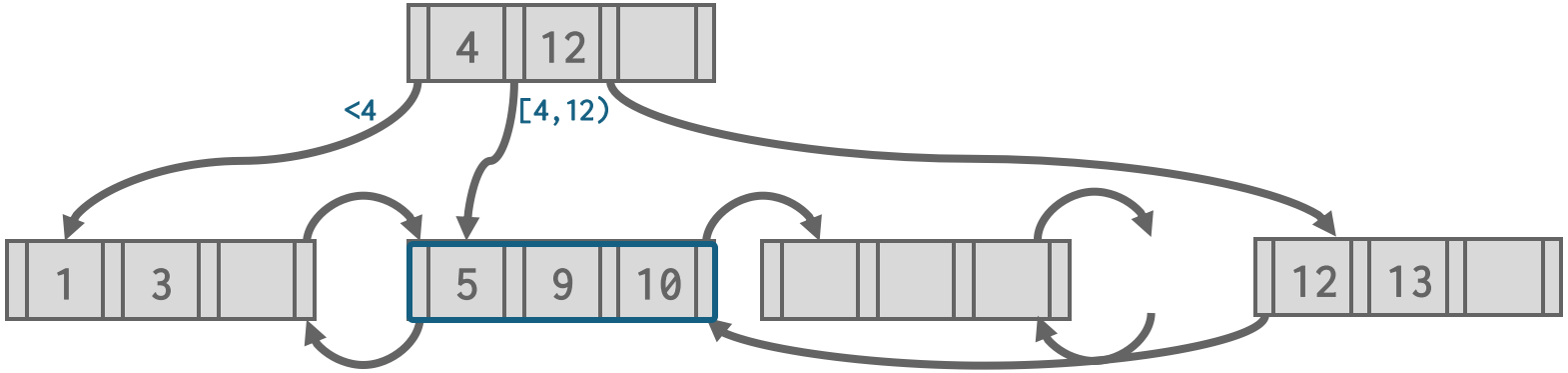
# B+Tree: Insert

Insert 6



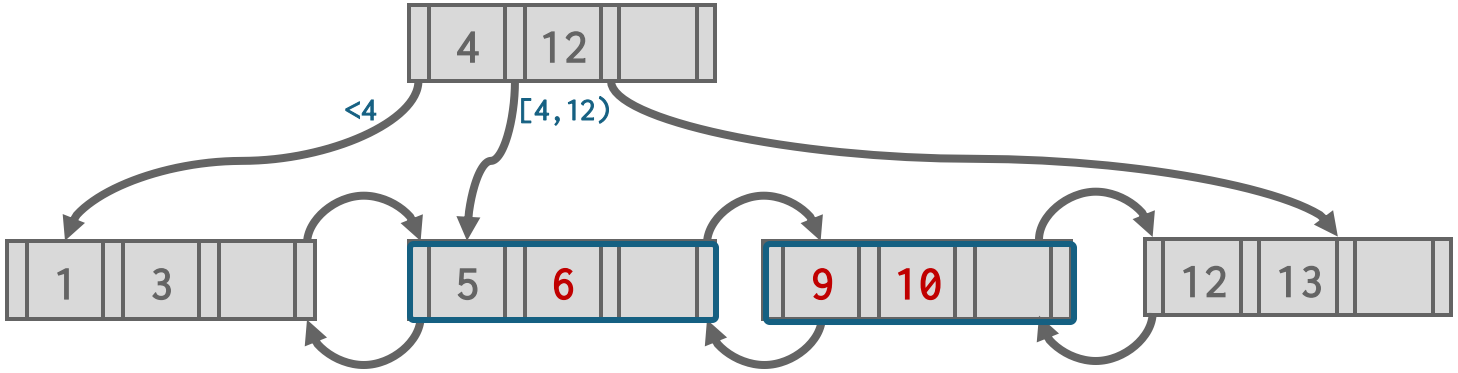
# B+Tree: Insert

Insert 6



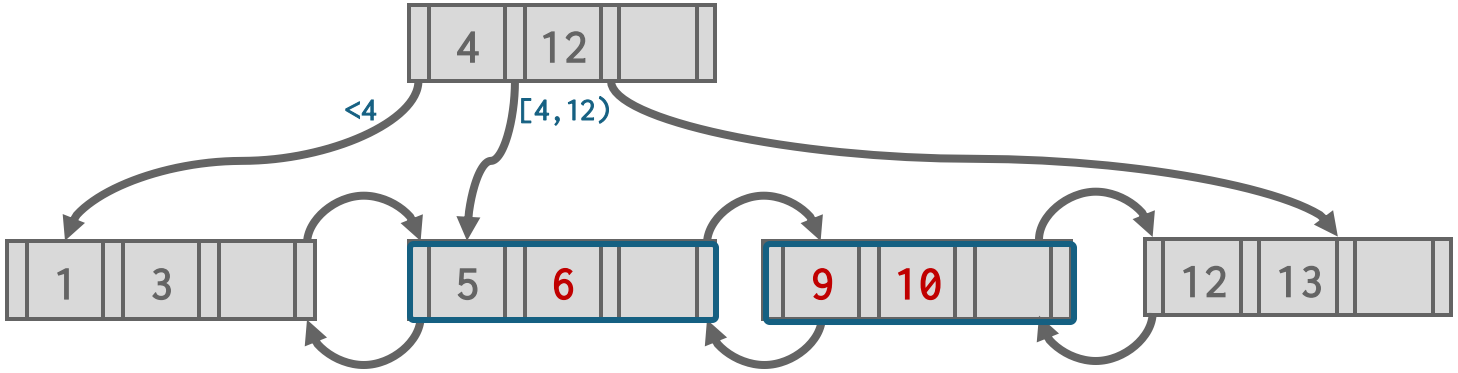
# B+Tree: Insert

Insert 6



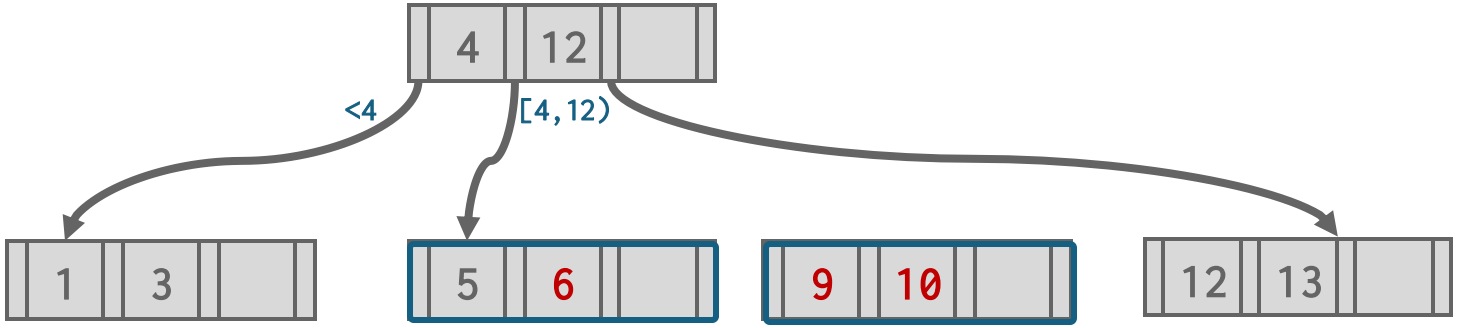
# B+Tree: Insert

Insert 6



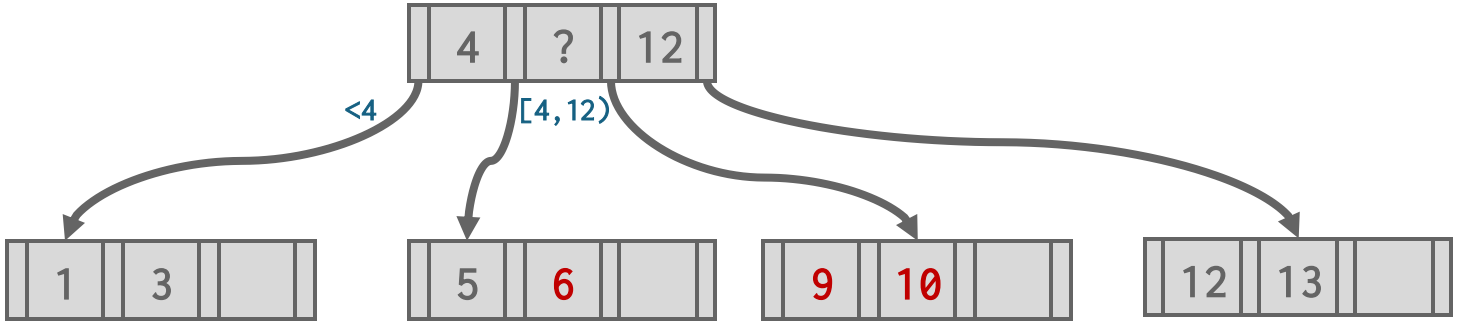
# B+Tree: Insert

Insert 6



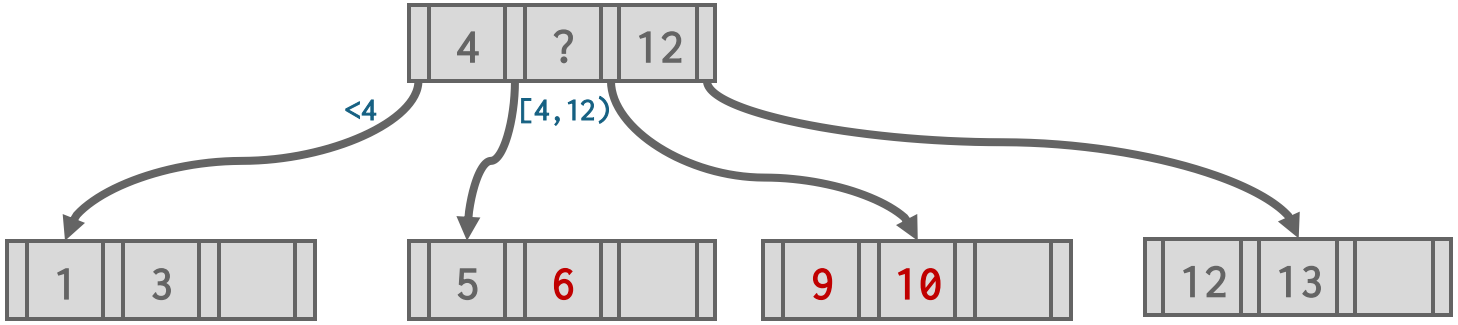
# B+Tree: Insert

Insert 6



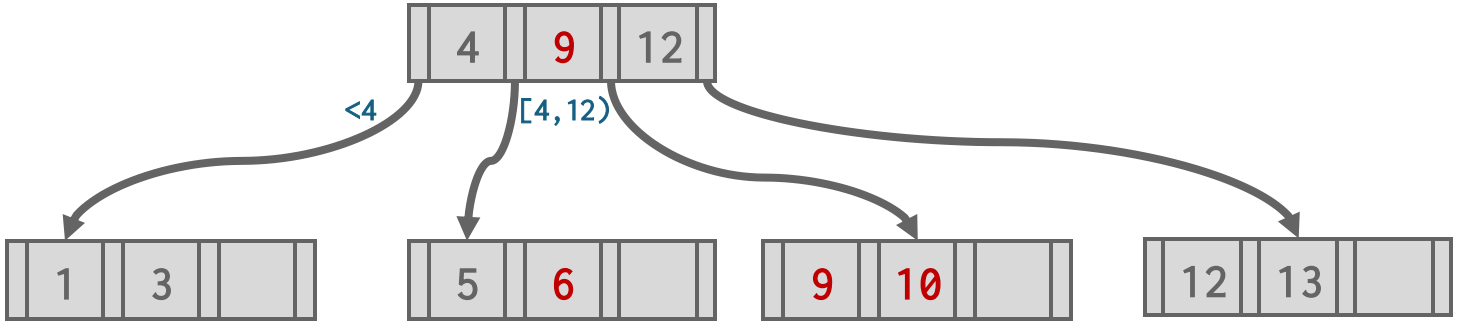
# B+Tree: Insert

Insert 6



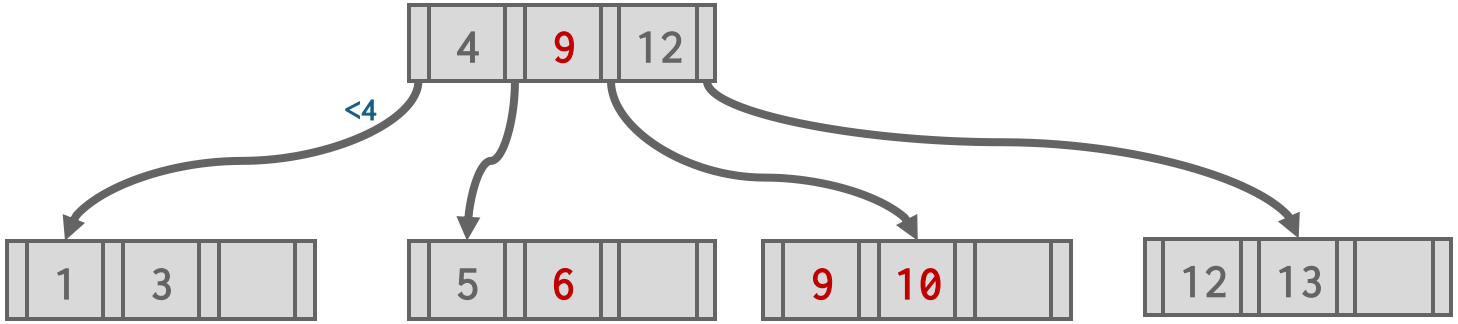
# B+Tree: Insert

Insert 6



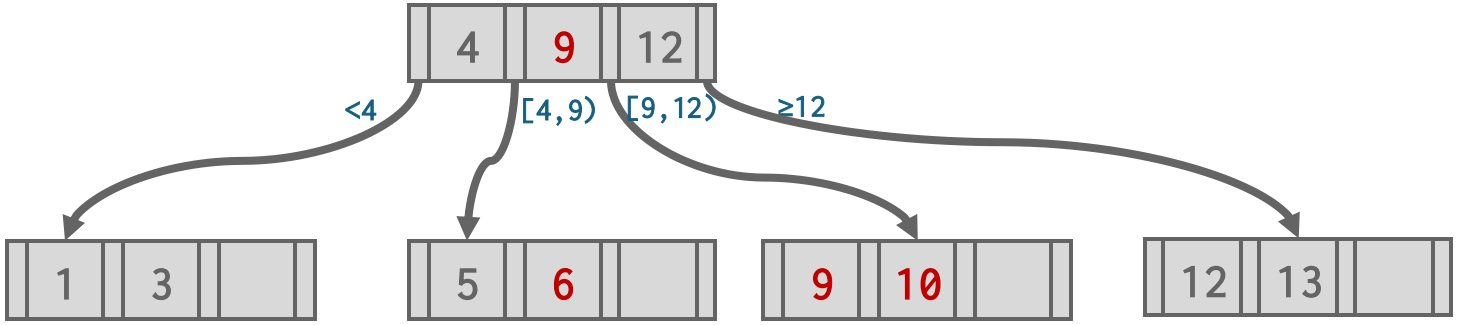
# B+Tree: Insert

Insert 6



# B+Tree: Insert

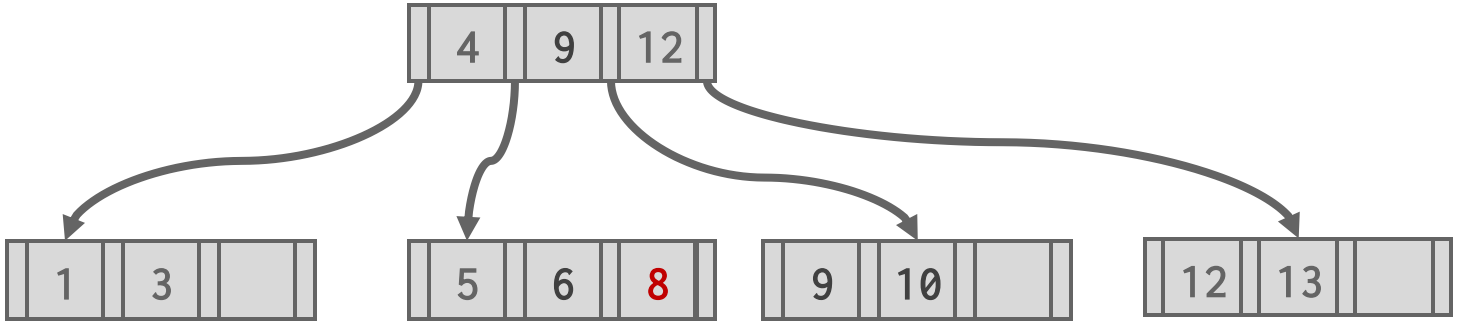
Insert 6



# B+Tree: Insert

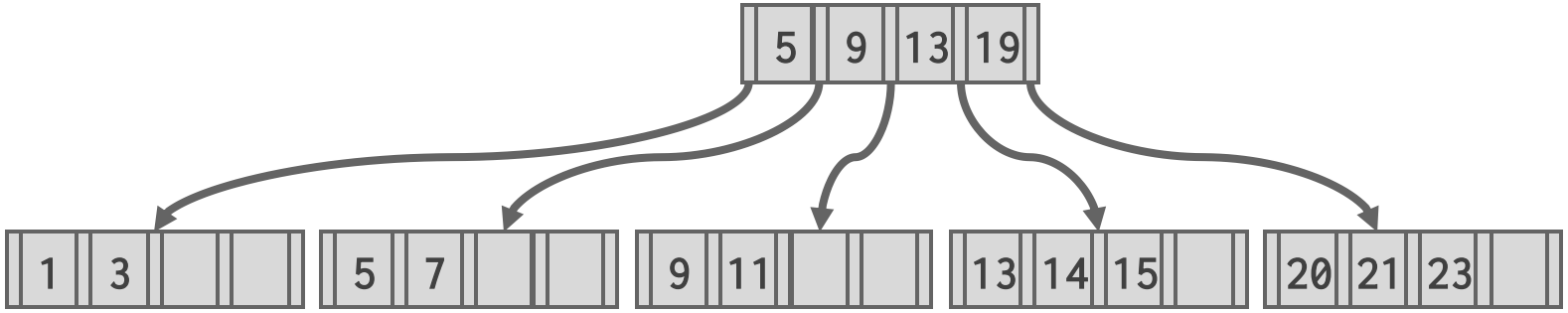
Insert 6

Insert 8



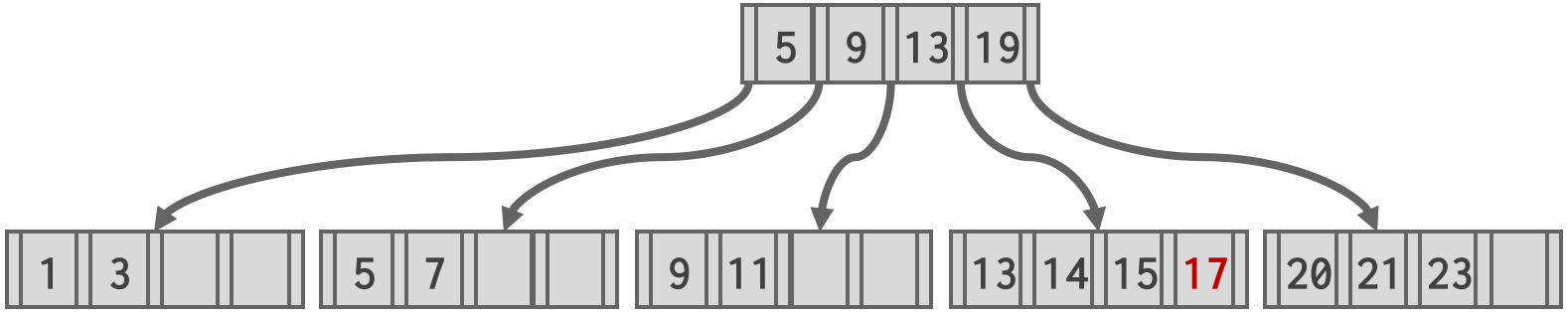
# Insert the Key 17

Note: new example/tree.

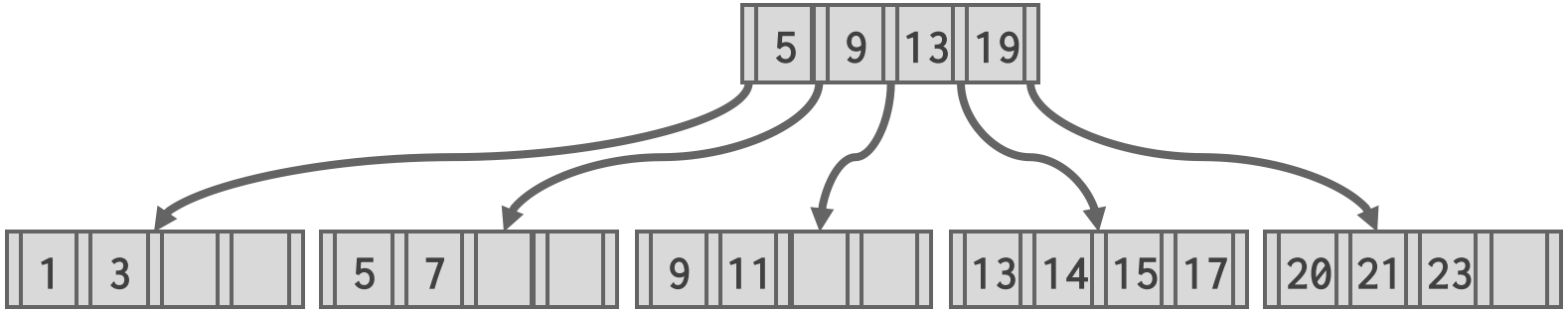


# Insert the Key 17

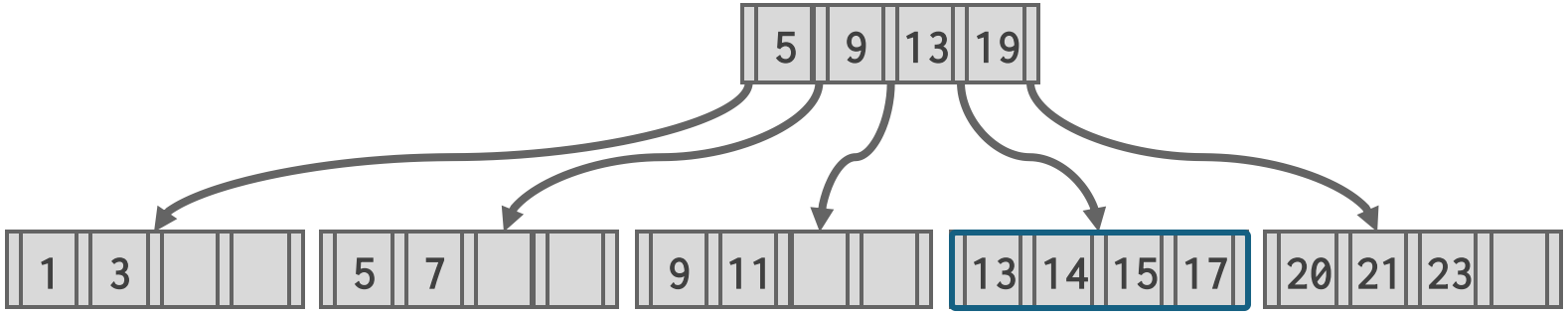
Note: new example/tree.



# Next, Insert the Key 16

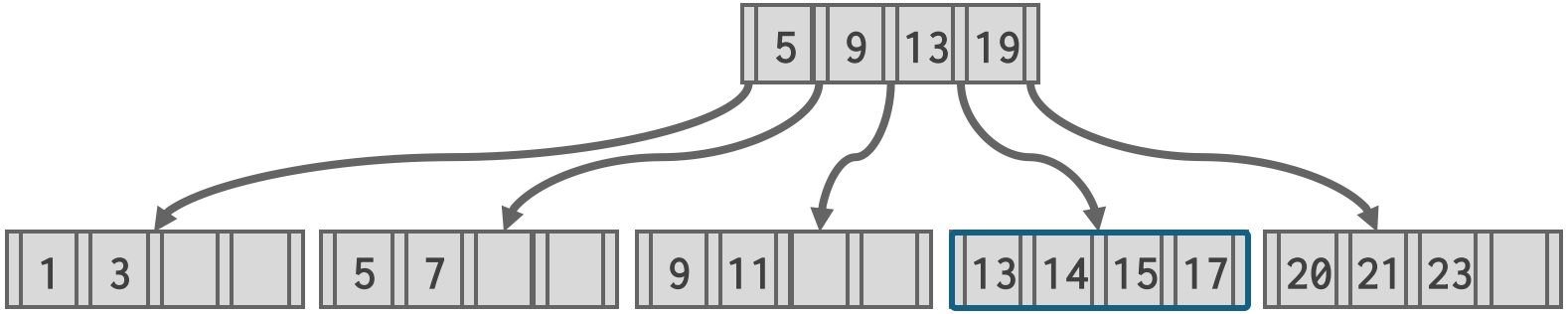


# Next, Insert the Key 16



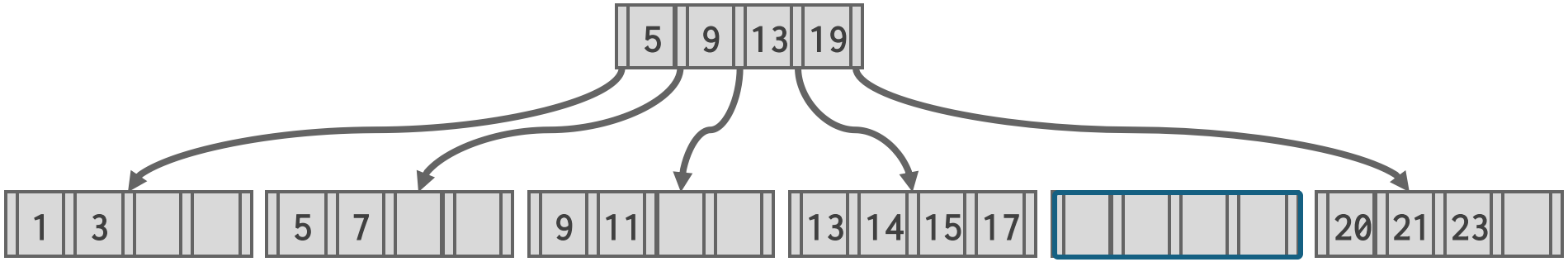

No space in the node where the new key “belongs”.

# Next, Insert the Key 16

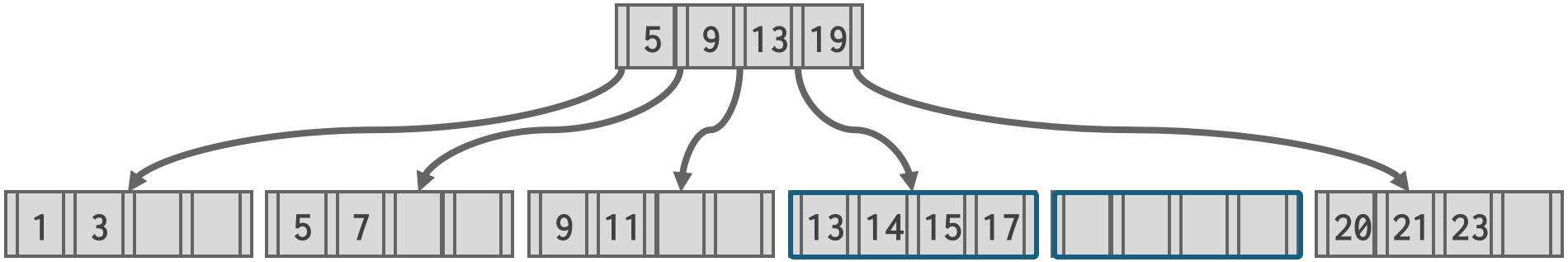



Split the node!
Copy the middle key.
Push the key up.

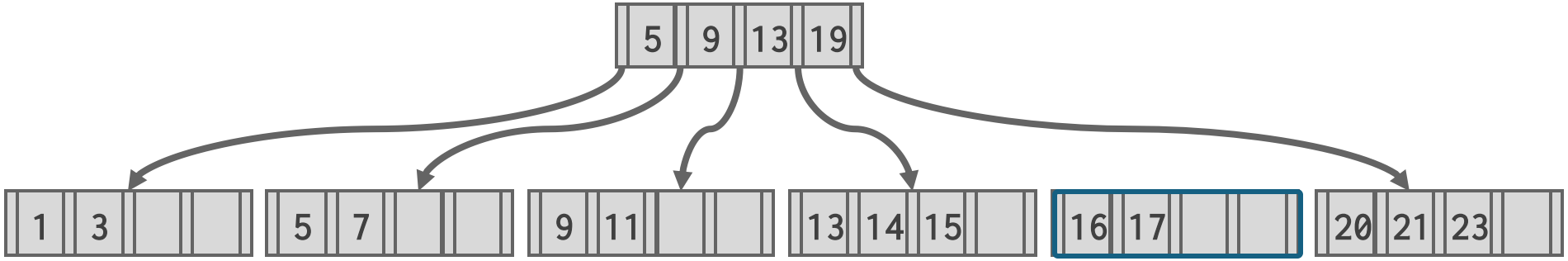
# Next, Insert the Key 16



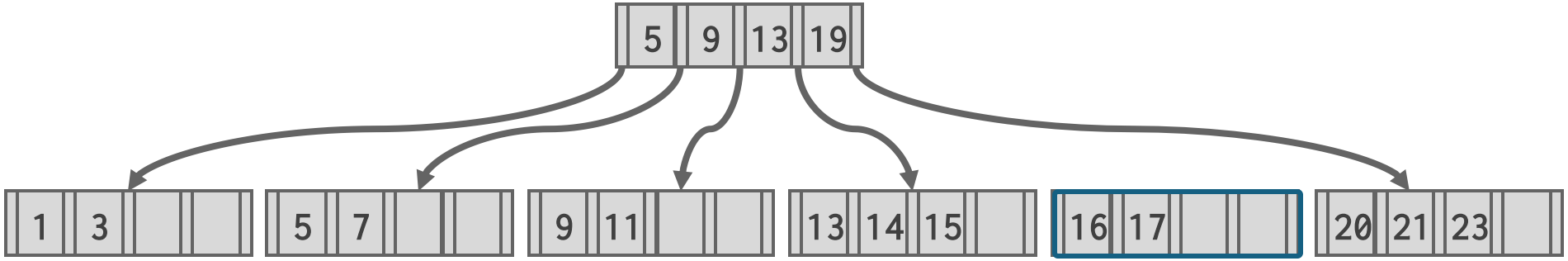
# Next, Insert the Key 16



# Next, Insert the Key 16

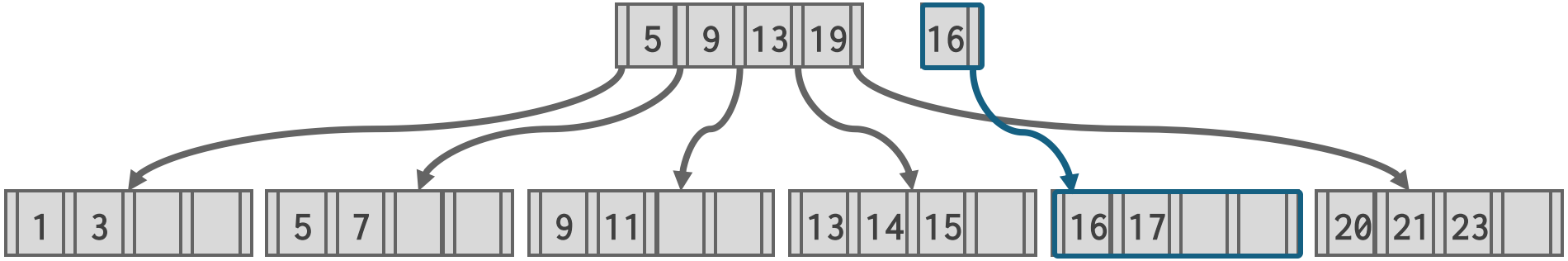


# Next, Insert the Key 16



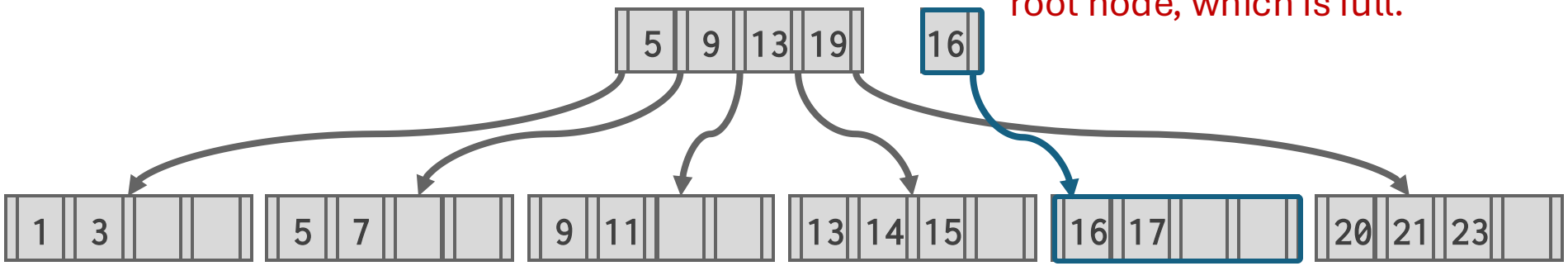

But, this is an "orphan" node. No parent node points to it.

# Next, Insert the Key 16




But, this is an "orphan" node. No parent node points to it.

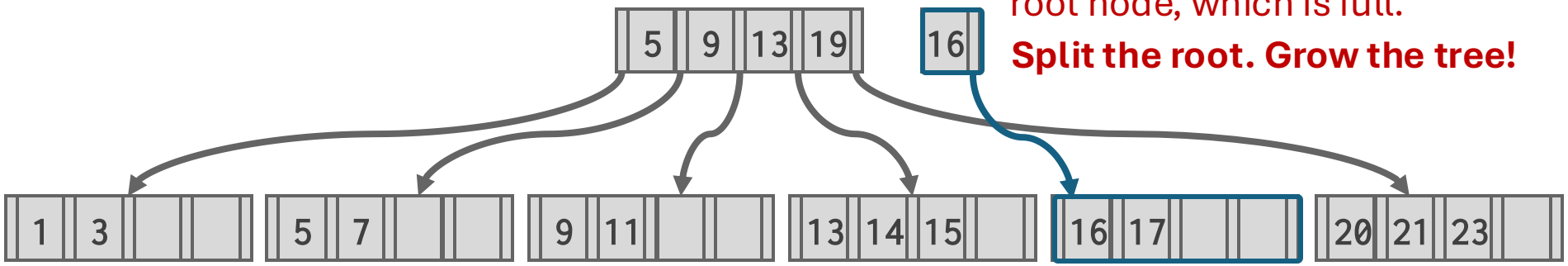
# Next, Insert the Key 16




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

But, this is an "orphan" node. No parent node points to it.

# Next, Insert the Key 16

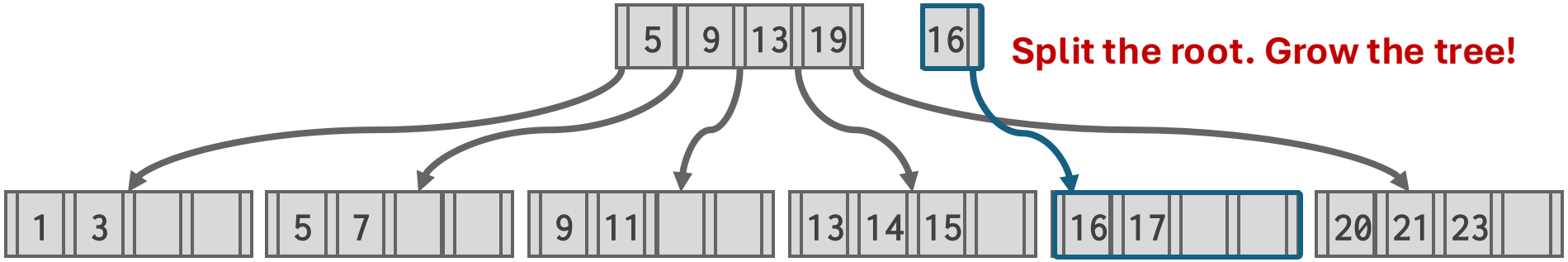



Want to create a key, pointer pair like this. But can't insert it in the root node, which is full.

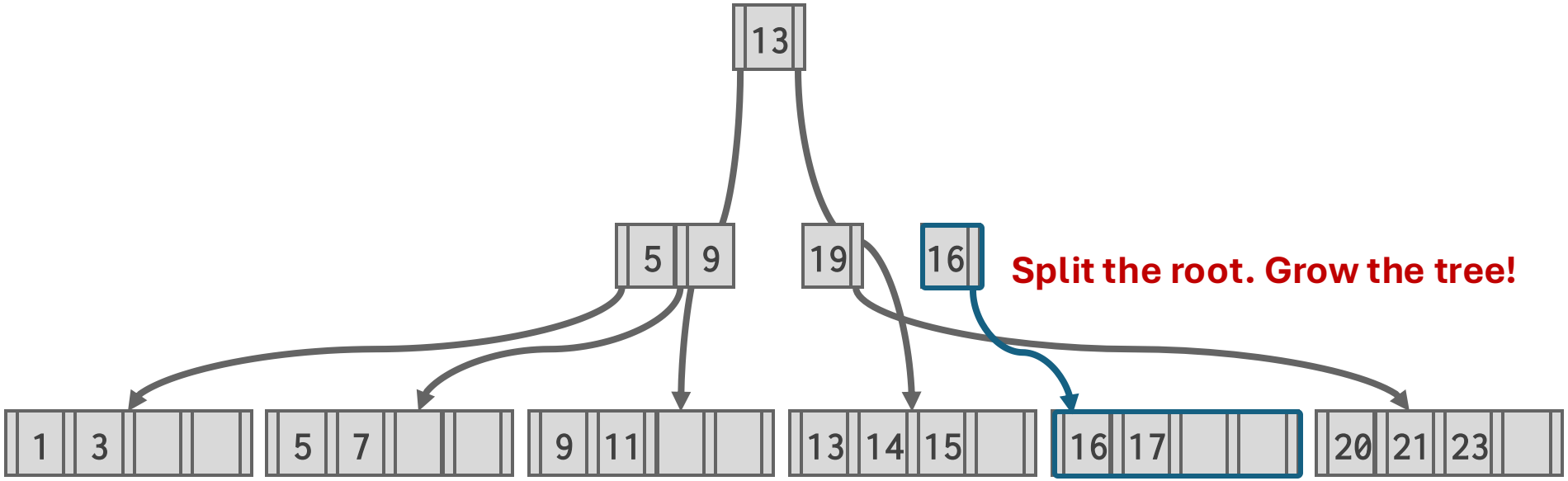
**Split the root. Grow the tree!**

But, this is an "orphan" node. No parent node points to it.

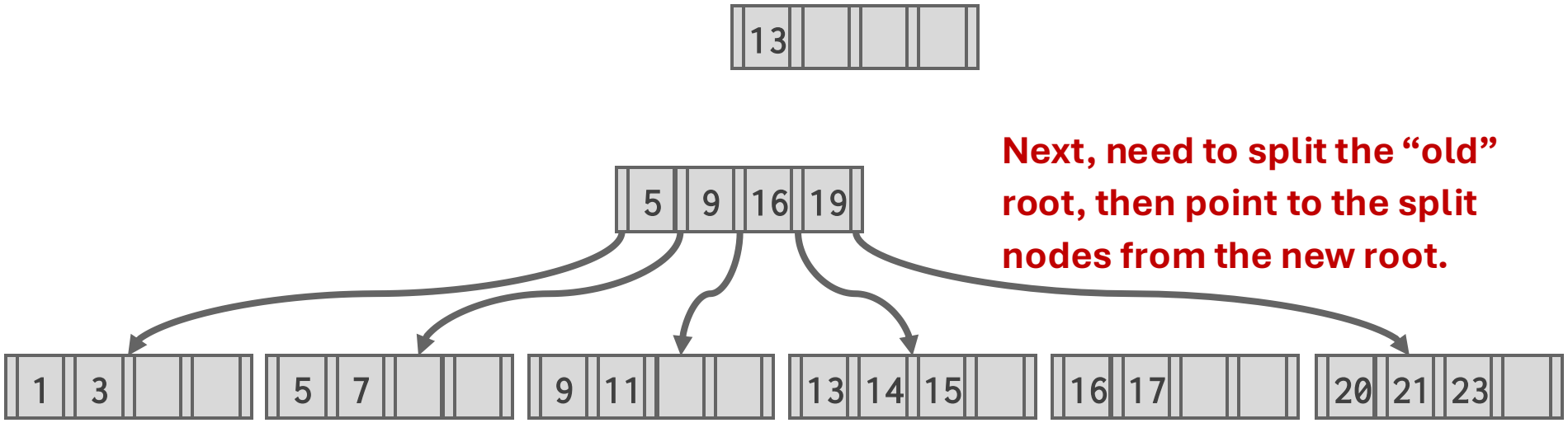
# Next, Insert the Key 16



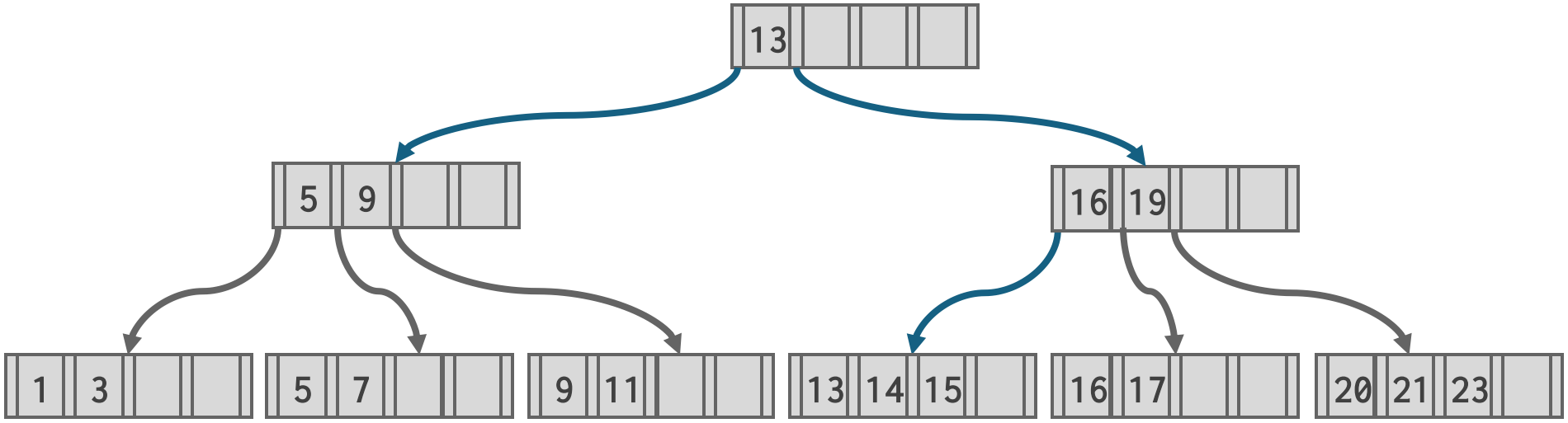
# Next, Insert the Key 16



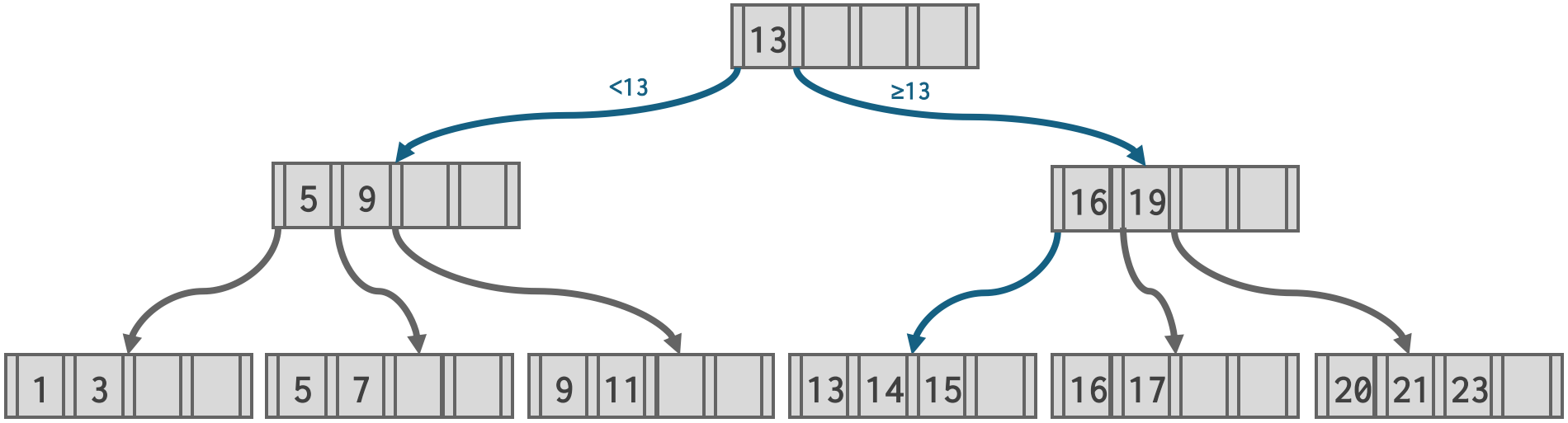
# Next, Insert the Key 16



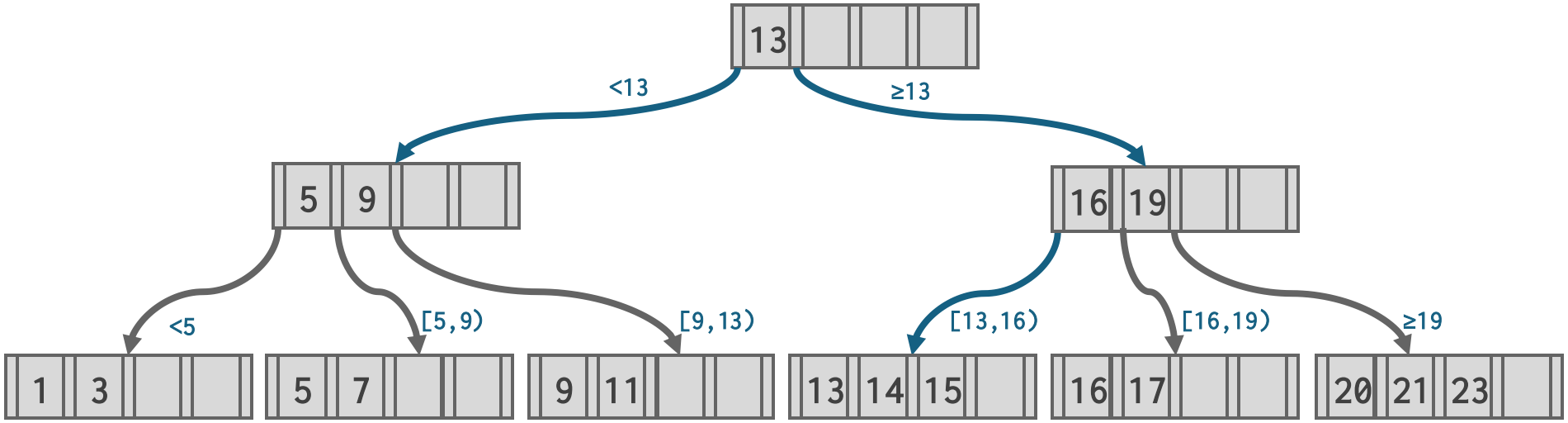
# Next, Insert the Key 16



# Next, Insert the Key 16



# Next, Insert the Key 16

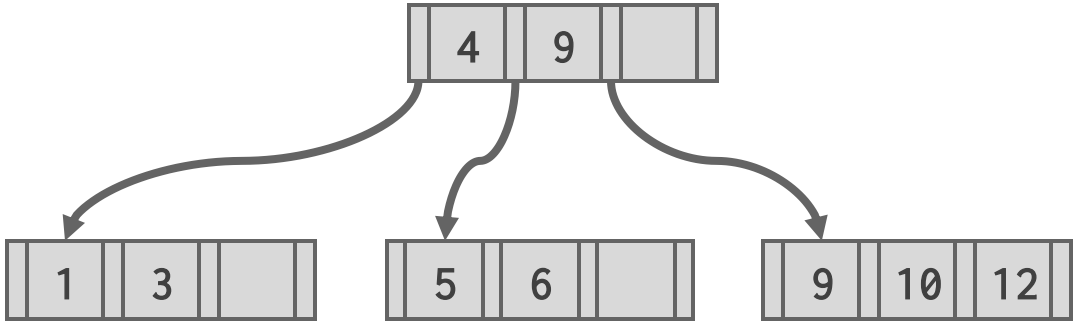


# B+Tree: Delete

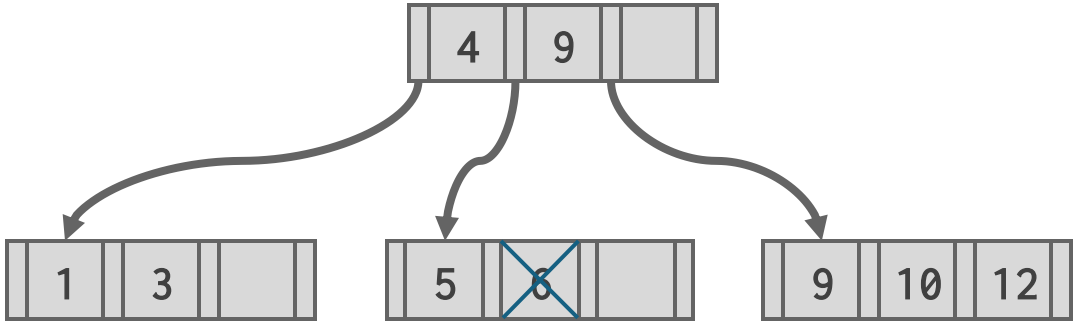
- Start at root, find leaf `L` where entry belongs.
  Remove the entry.
  If `L` is at least half-full, done!
  If `L` has only `M/2-1` entries,
  - Try to re-distribute, borrowing from sibling (adjacent node with same parent as `L`).
  - If re-distribution fails, merge `L` and sibling.

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

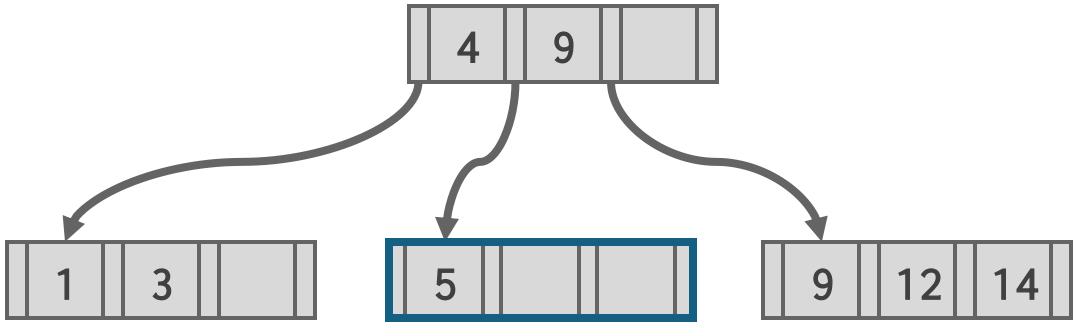
# Delete the Key 6



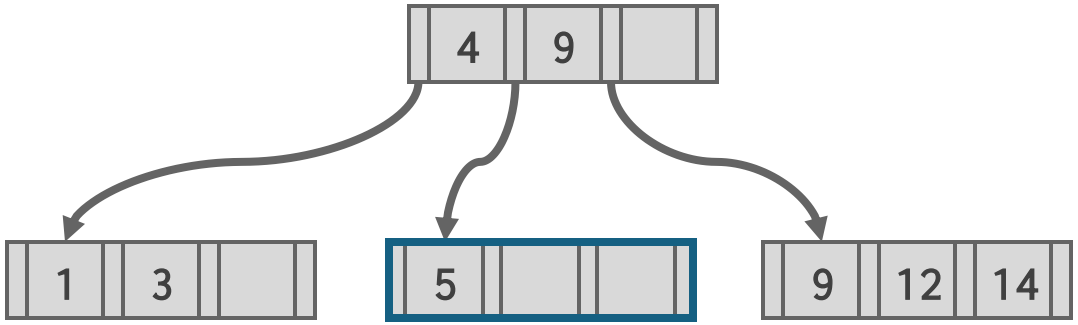
# Delete the Key 6



# Delete the Key 6

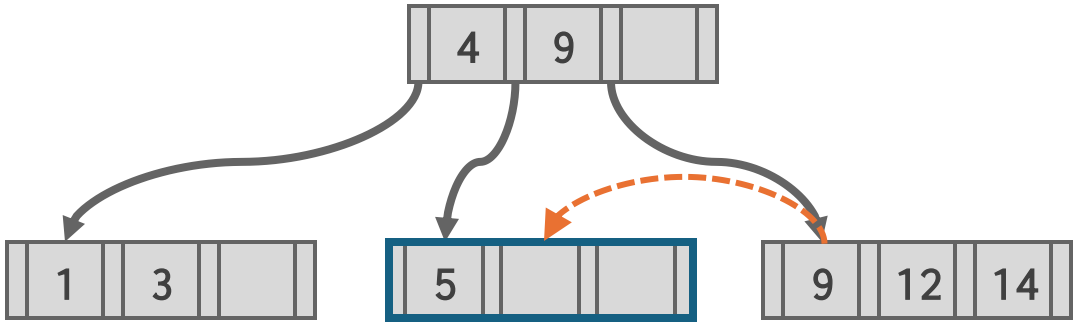


# Delete the Key 6



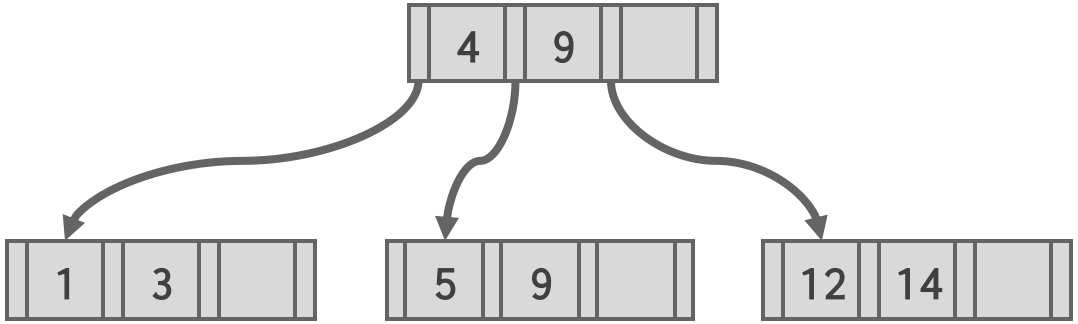

Borrow from a “rich” neighbor.

# Delete the Key 6




Borrow from a “rich” neighbor.

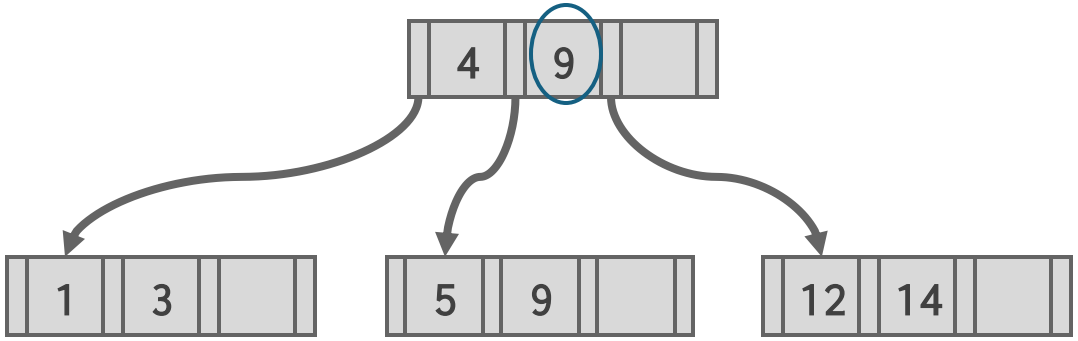
# Delete the Key 6




Borrow from a “rich” neighbor.

Could borrow from either neighbor.

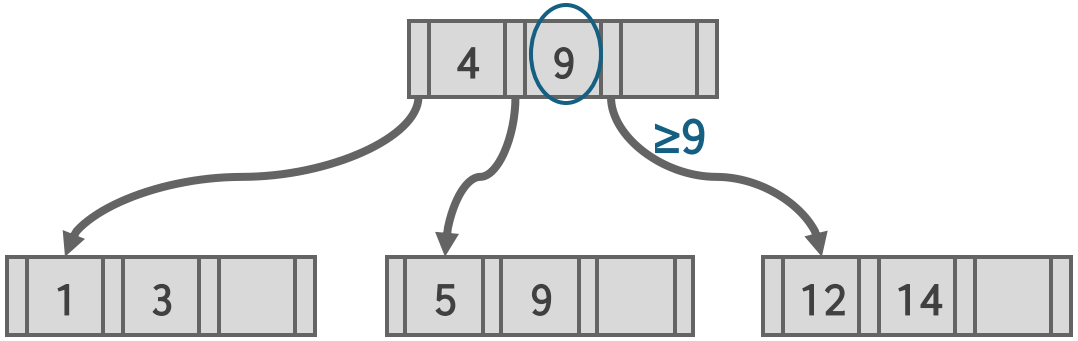
# Delete the Key 6




Borrow from a “rich” neighbor.

Could borrow from either neighbor.

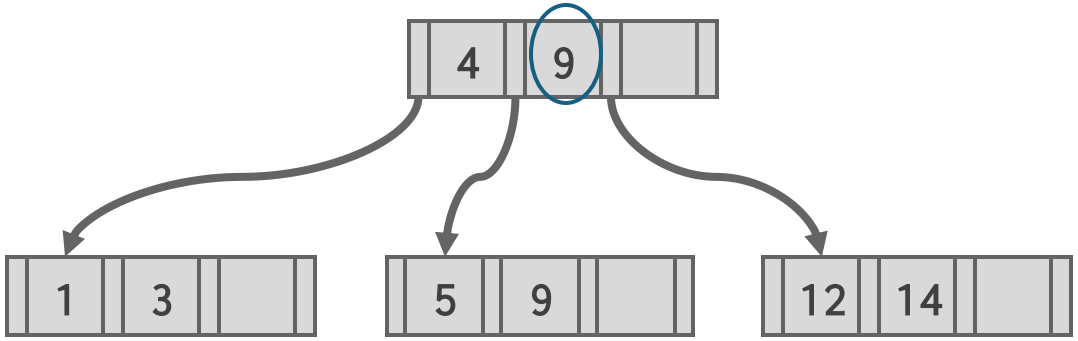
# Delete the Key 6




Borrow from a “rich” neighbor.

Could borrow from either neighbor.

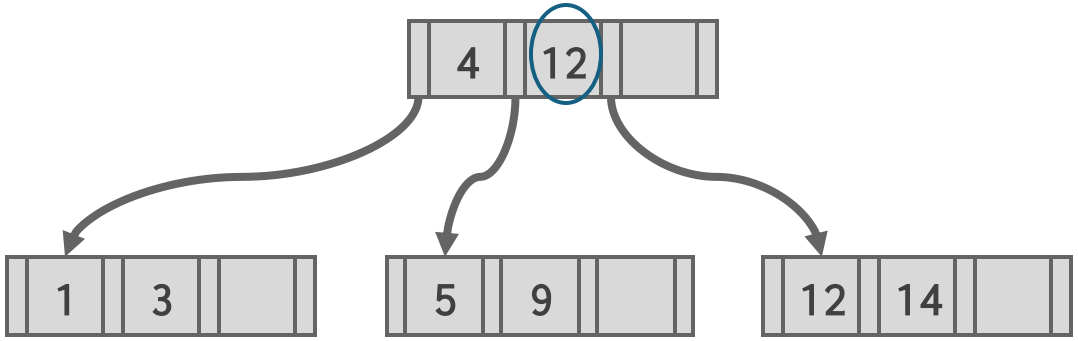
# Delete the Key 6




Borrow from a “rich” neighbor.

Could borrow from either neighbor.

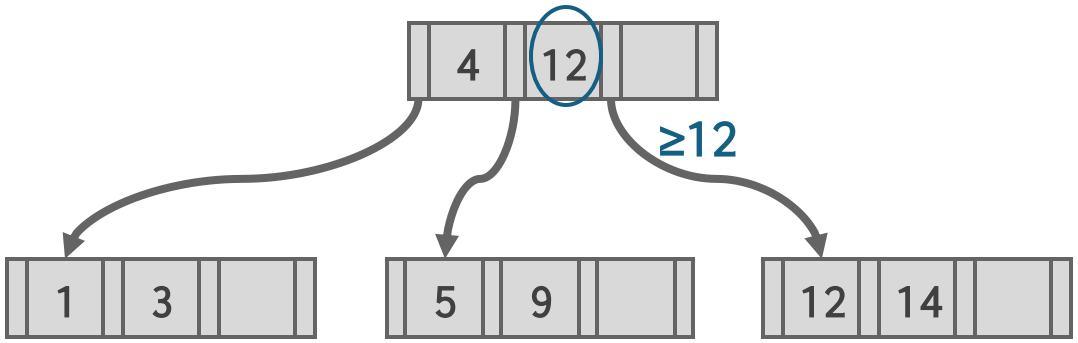
# Delete the Key 6




Borrow from a “rich” neighbor.

Could borrow from either neighbor.

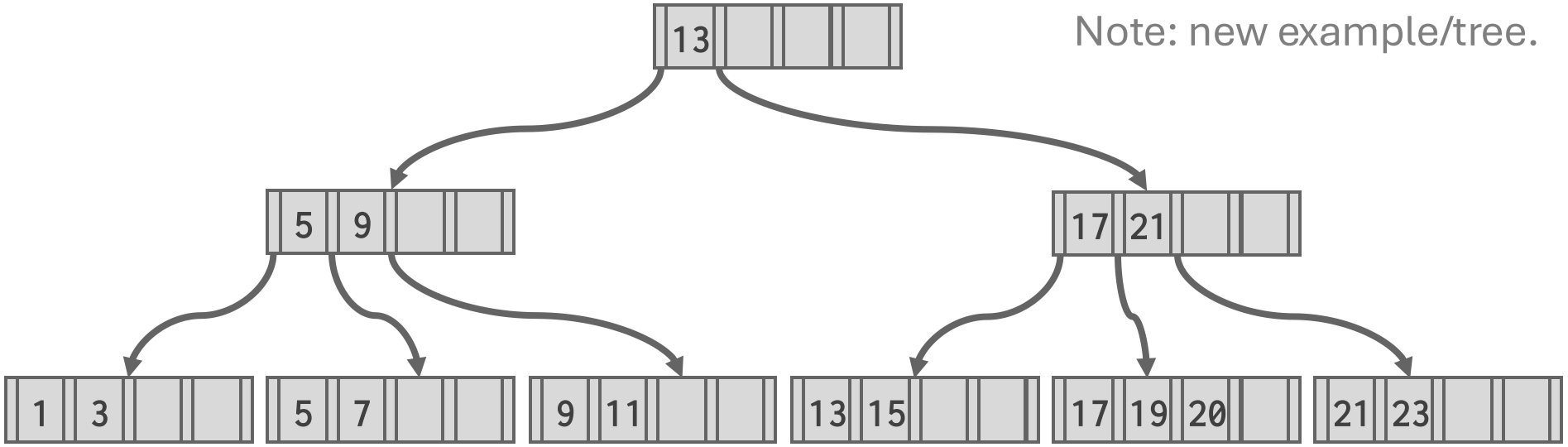
# Delete the Key 6




Borrow from a “rich” neighbor.

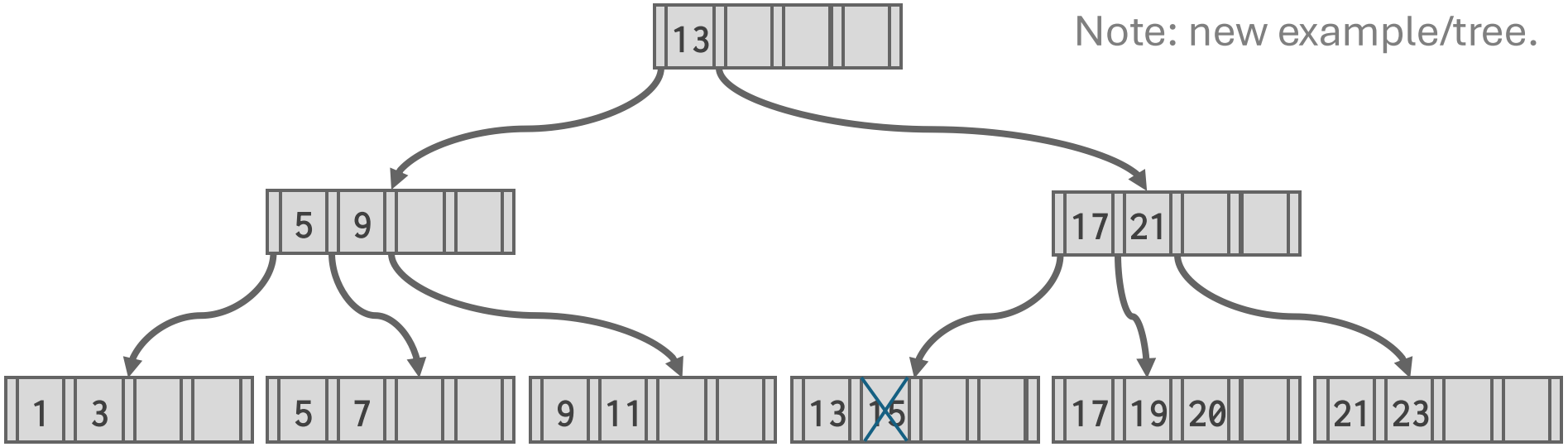
Could borrow from either neighbor.

# Delete the Key 15



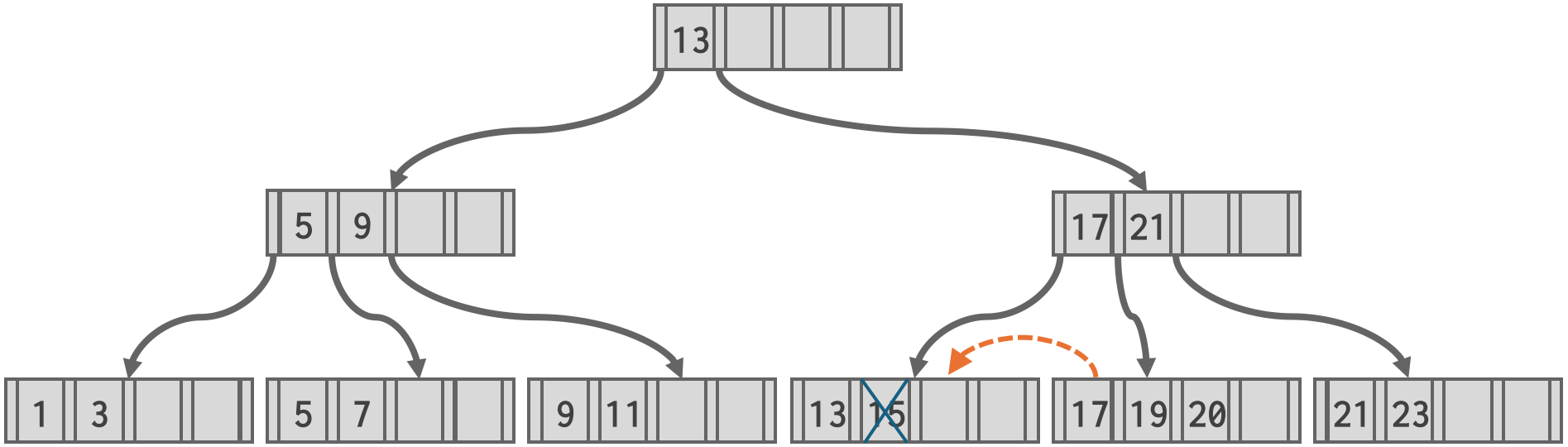

Note: new example/tree.

# Delete the Key 15

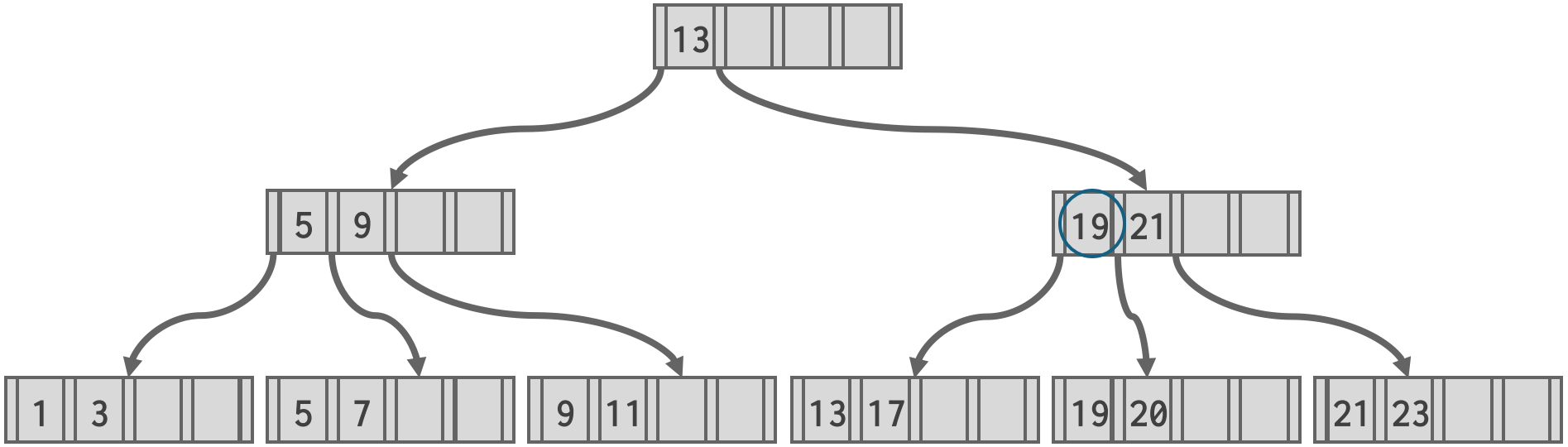



Note: new example/tree.

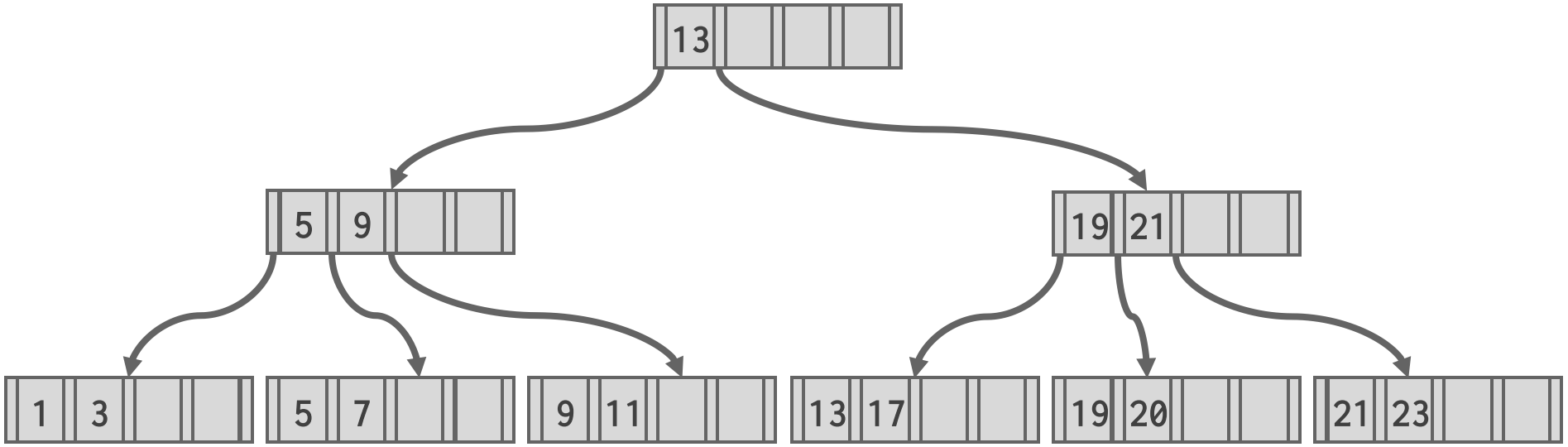
# Delete the Key 15



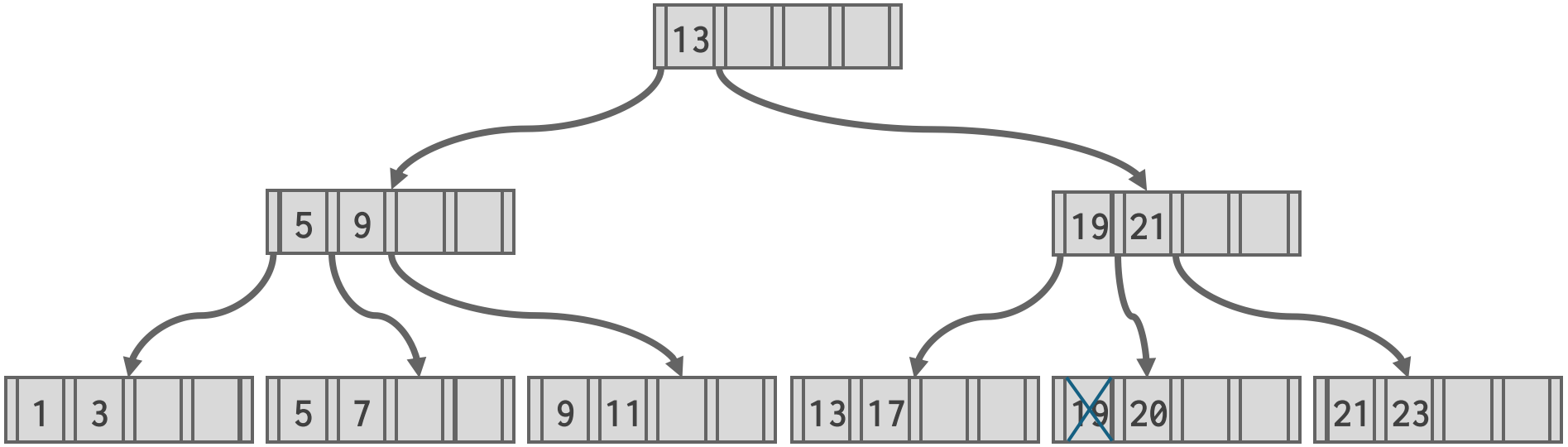
# Delete the Key 15



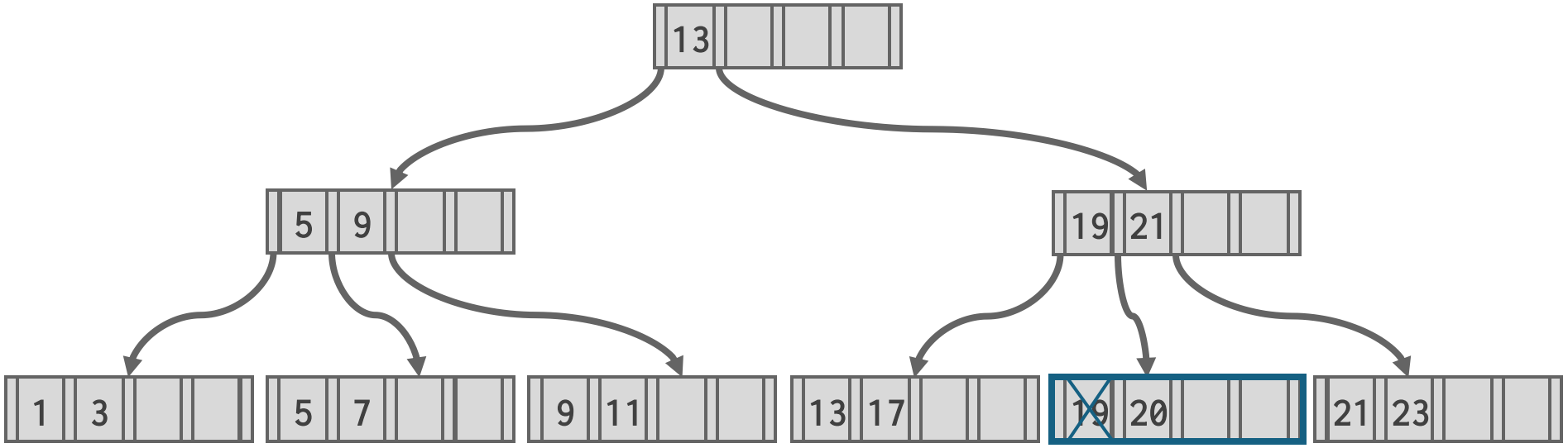
# Delete the Key 19



# Delete the Key 19



# Delete the Key 19

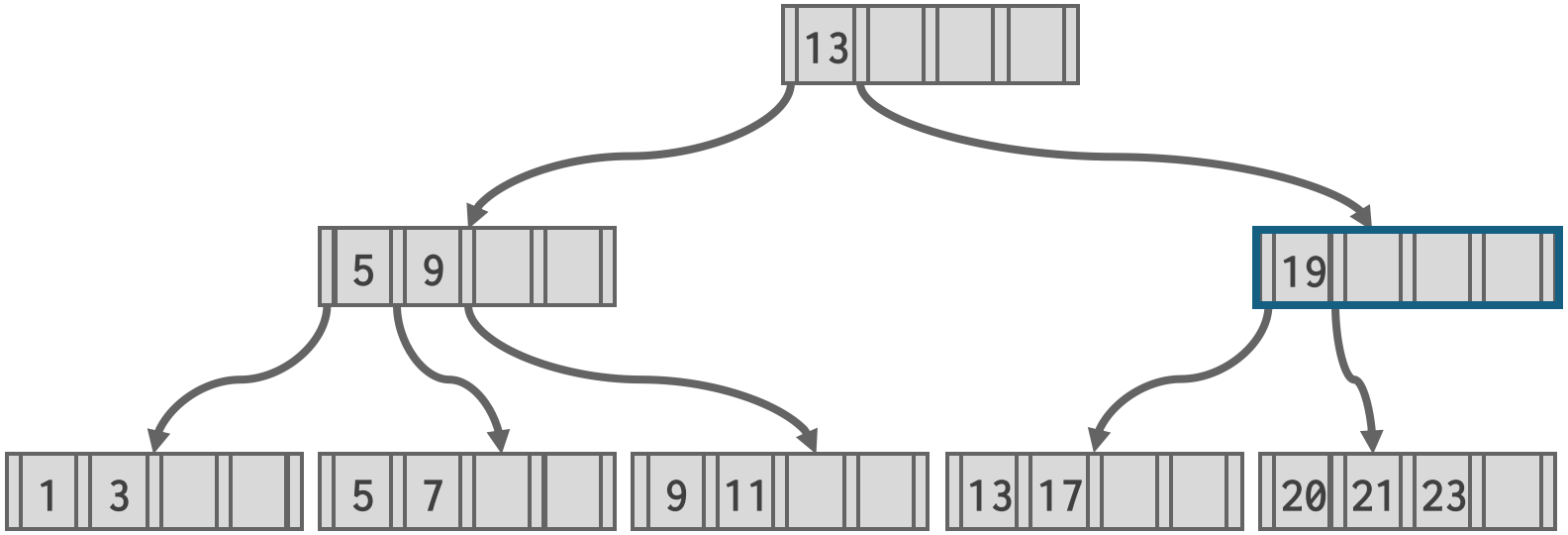



Under-filled.

No “rich” neighbors to borrow.

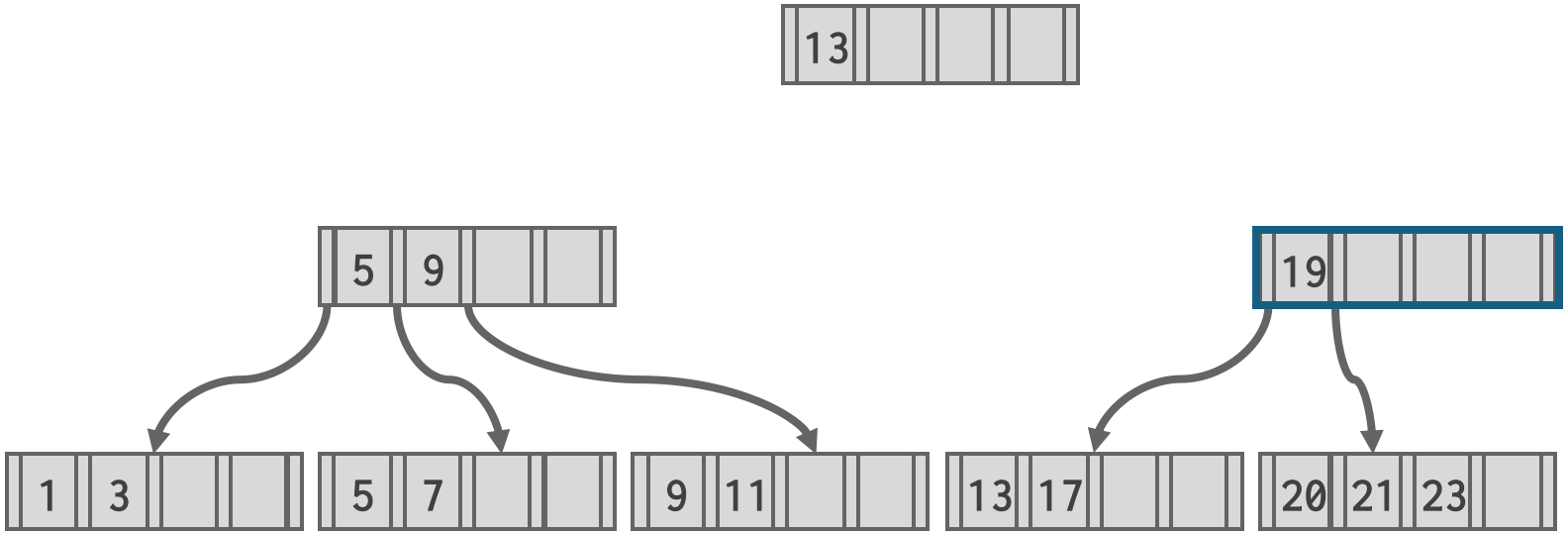
Merge with a sibling

# Delete the Key 19



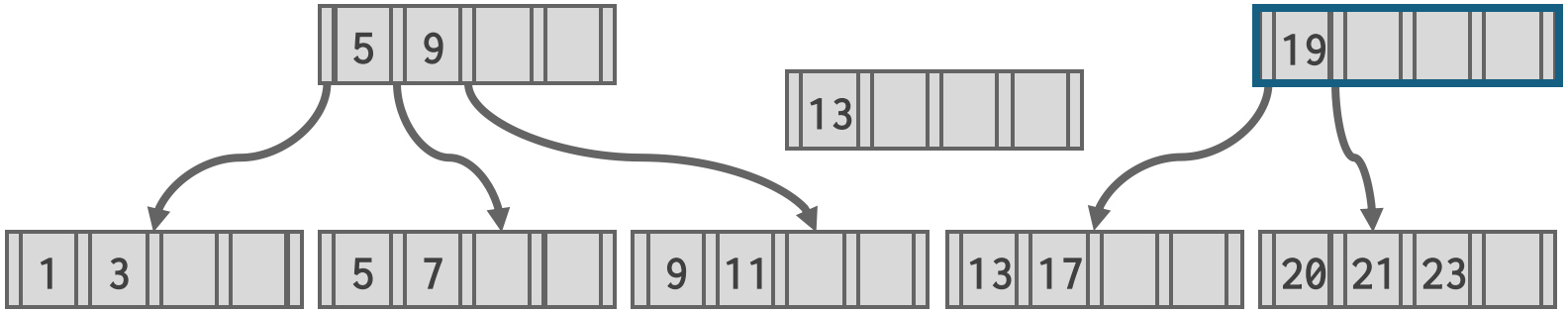

This node is under-filled!
Pull-down.

# Delete the Key 19




This node is under-filled!
Pull-down.

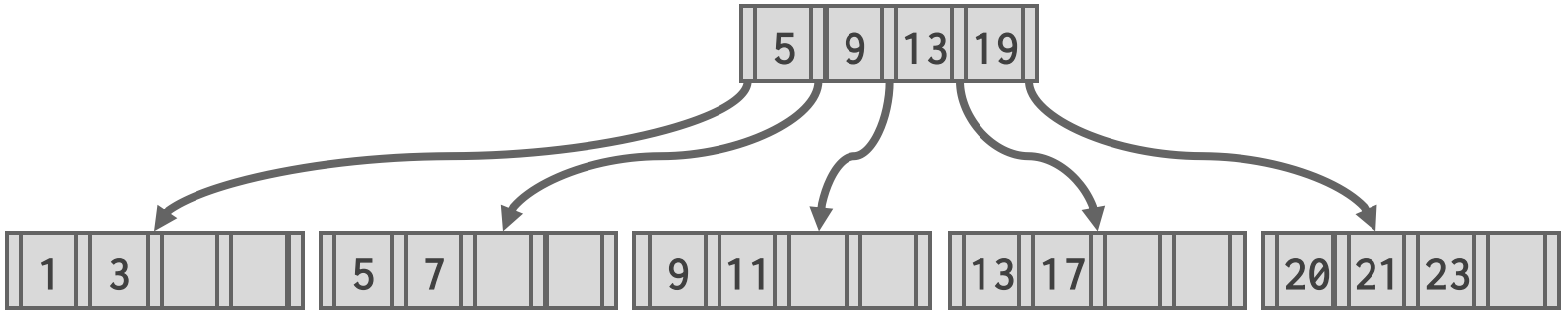
# Delete the Key 19




This node is under-filled!
Pull-down.

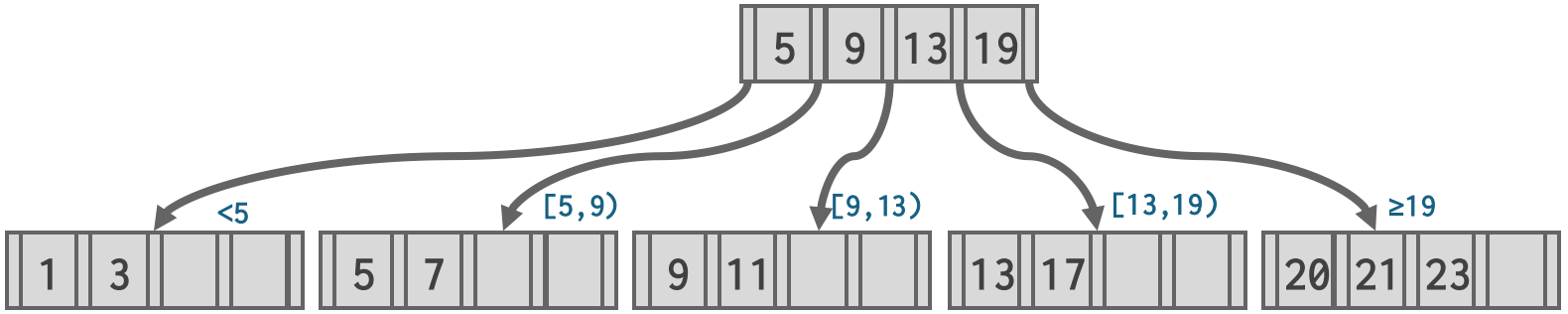
# Delete the Key 19

The tree has shrunk in height.



# Delete the Key 19

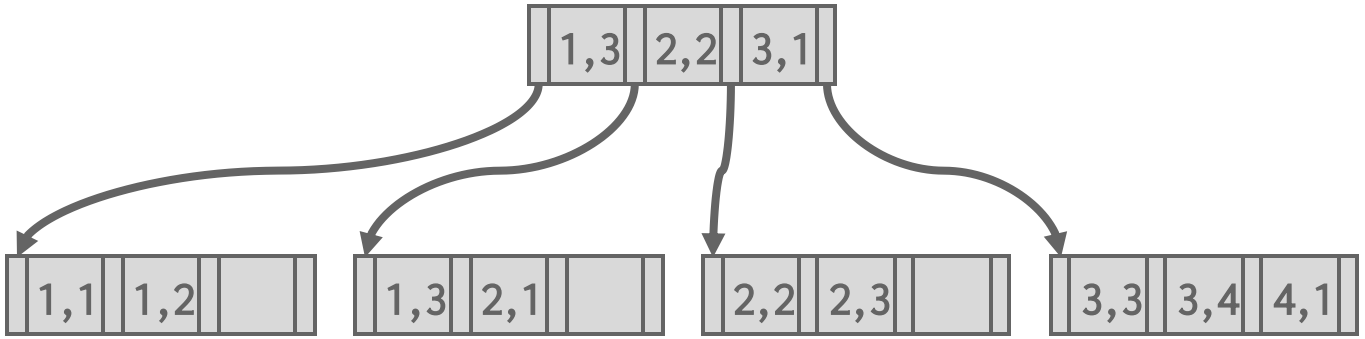
The tree has shrunk in height.



# Composite Index

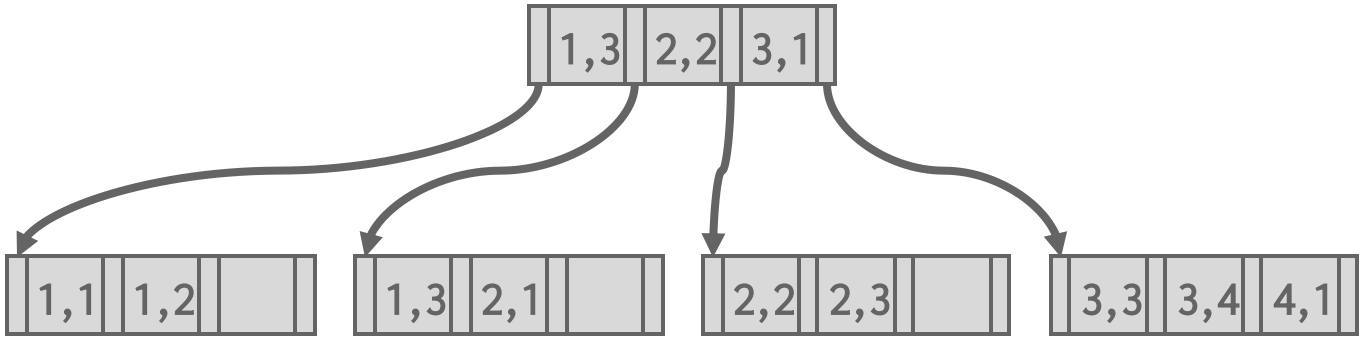
- Composite Index: The key is composed of multiple attributes.
- `CREATE INDEX LFM_name ON artist`
  `(last_name, first_name, middle_names NULLS FIRST);`
- Can use a B+Tree index if the query provides a "prefix" of composite key. Example: Index on `<a,b,c>`
  - Supported: `(a=1 AND b=2 AND c=3)`
  - Supported: `(a=1 AND b=2)`
  - NOT (generally) supported: `(b=2)`, `(c=3)`
- For a hash index, we must have all attributes in search key.

# Selection Conditions



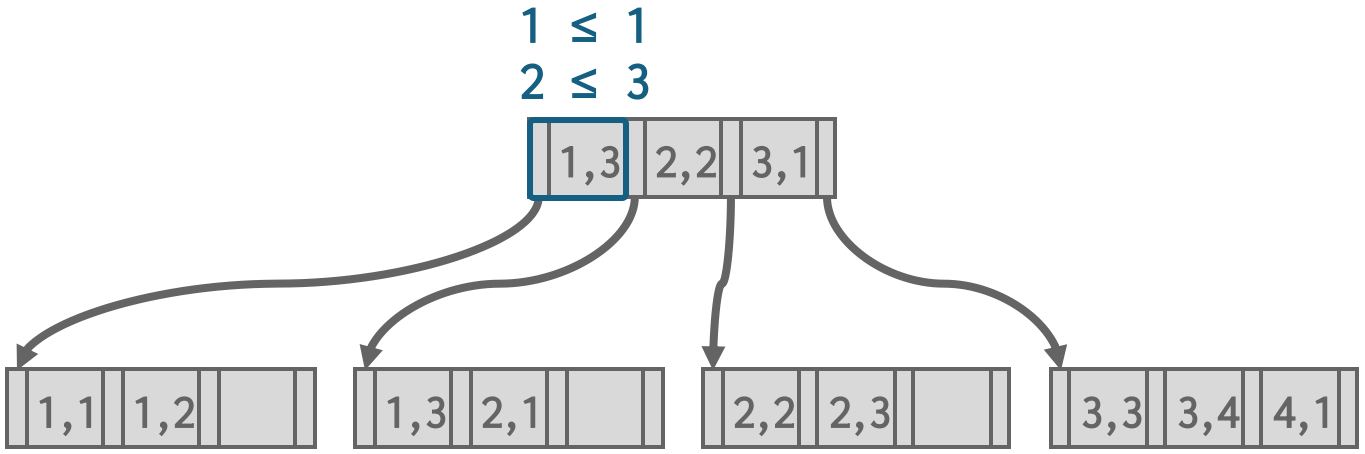
# Selection Conditions

Find Key=(1,2)



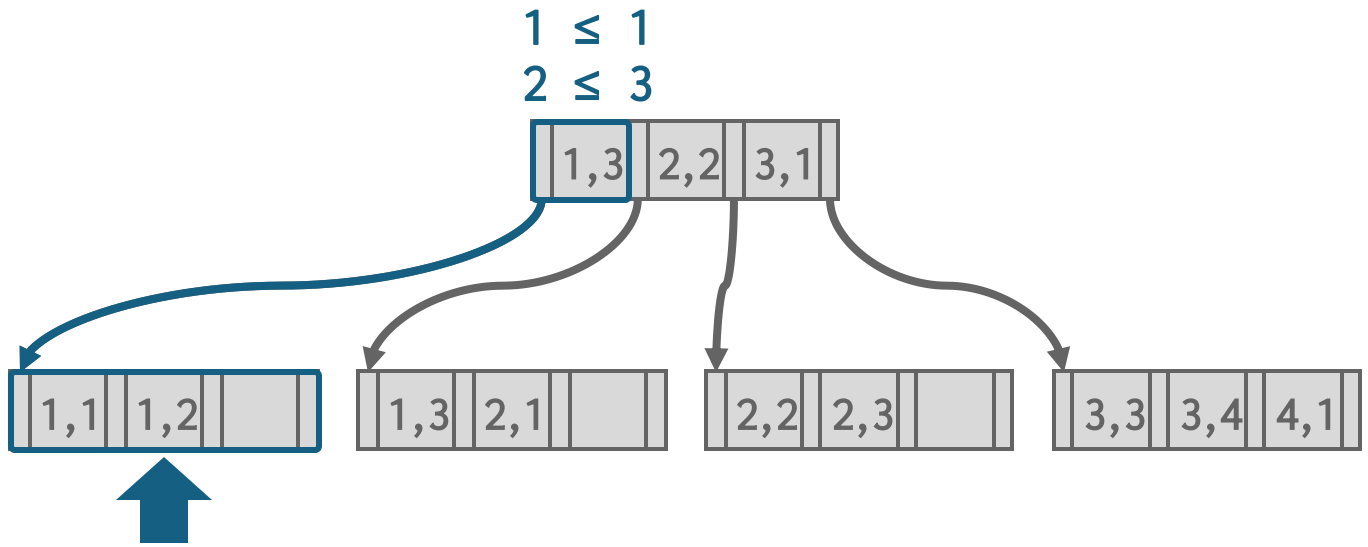
# Selection Conditions

Find Key=(1,2)



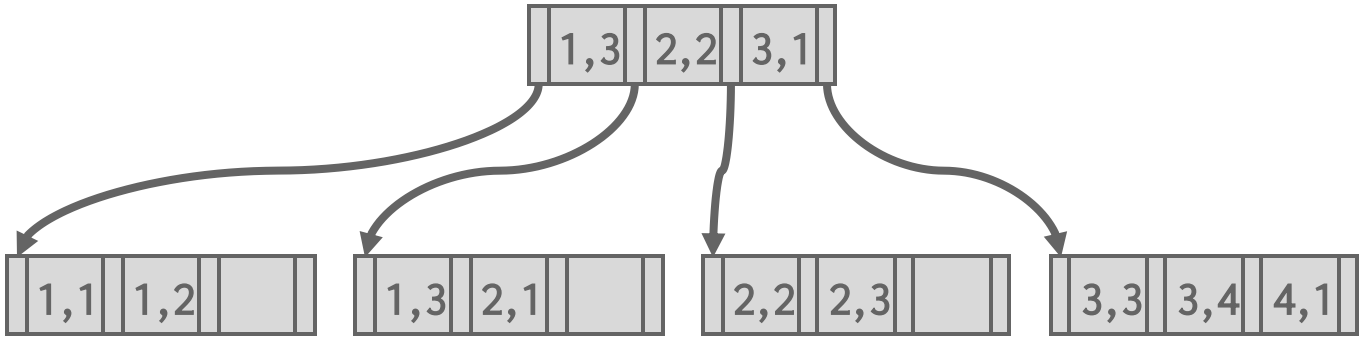
# Selection Conditions

Find Key=(1,2)



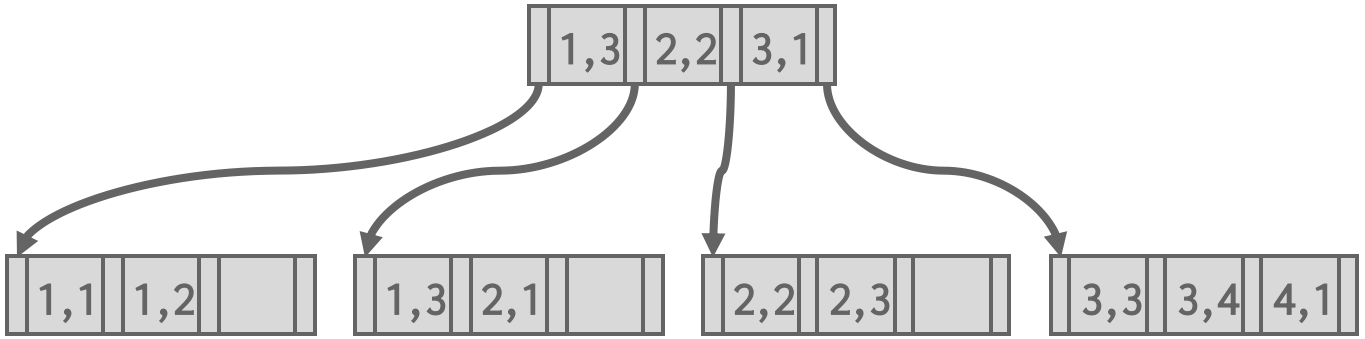
# Selection Conditions

Find Key=(1,2)



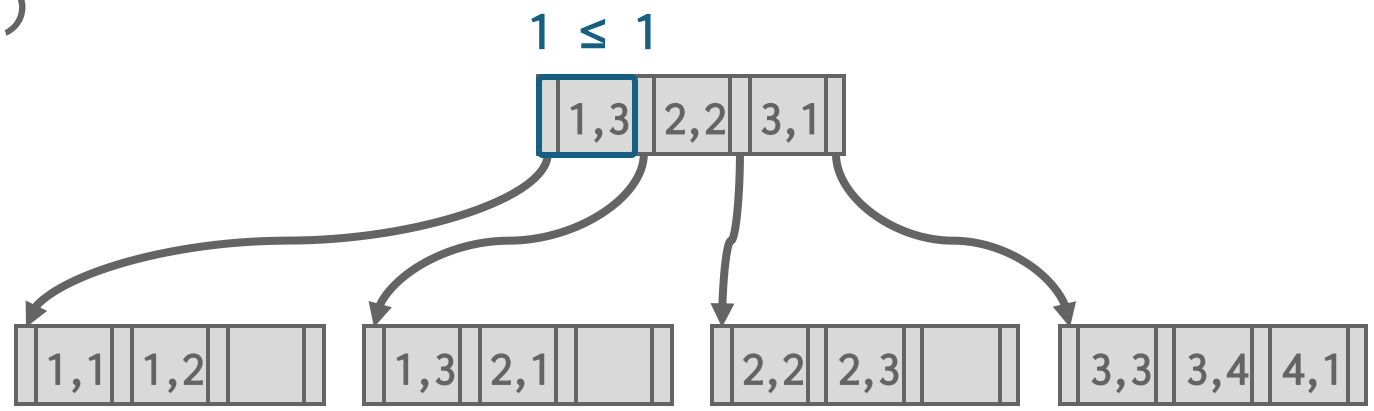
# Selection Conditions

```
Find Key=(1,2)
Find Key=(1,*)
```



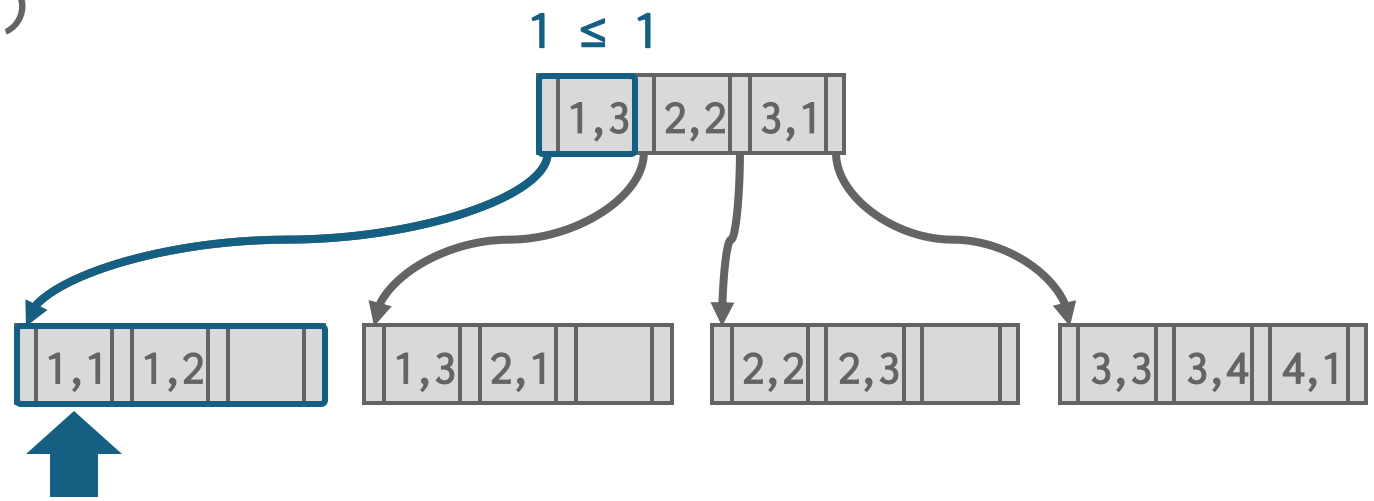
# Selection Conditions

```
Find Key=(1,2)
Find Key=(1,*)
```



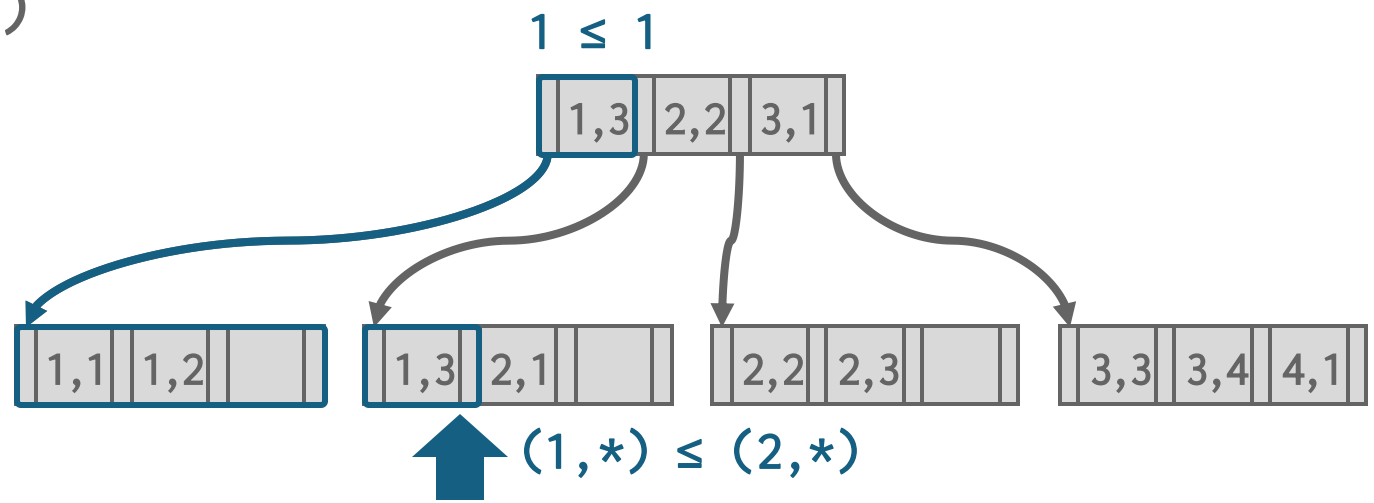
# Selection Conditions

```
Find Key=(1,2)
Find Key=(1,*)
```



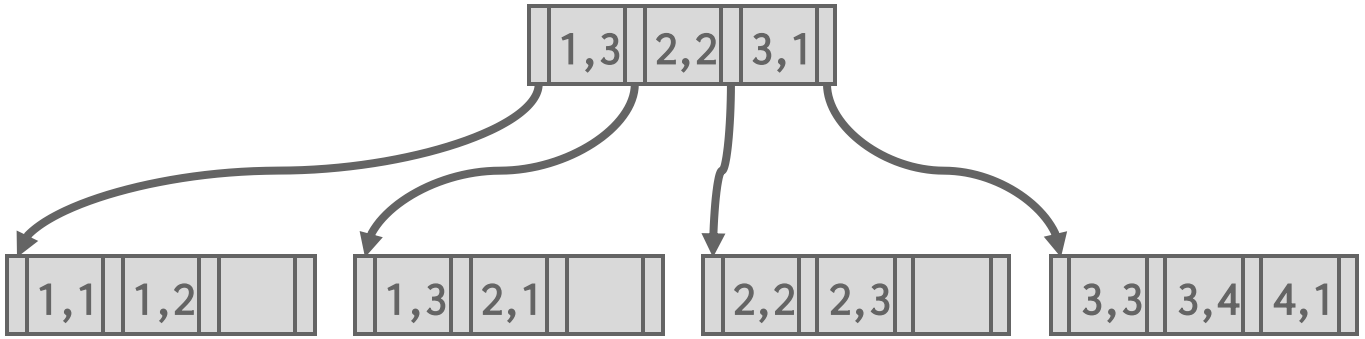
# Selection Conditions

```
Find Key=(1,2)
Find Key=(1,*)
```



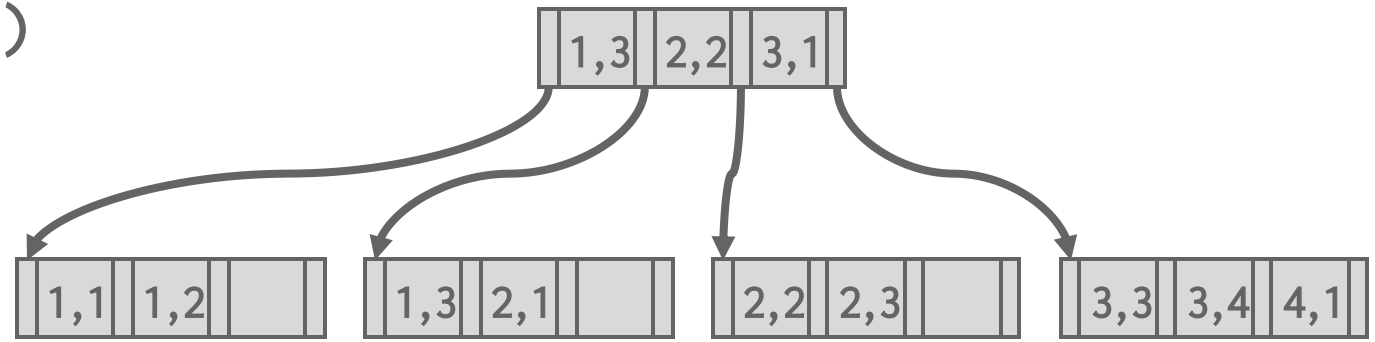
# Selection Conditions

```
Find Key=(1,2)
Find Key=(1,*)
```



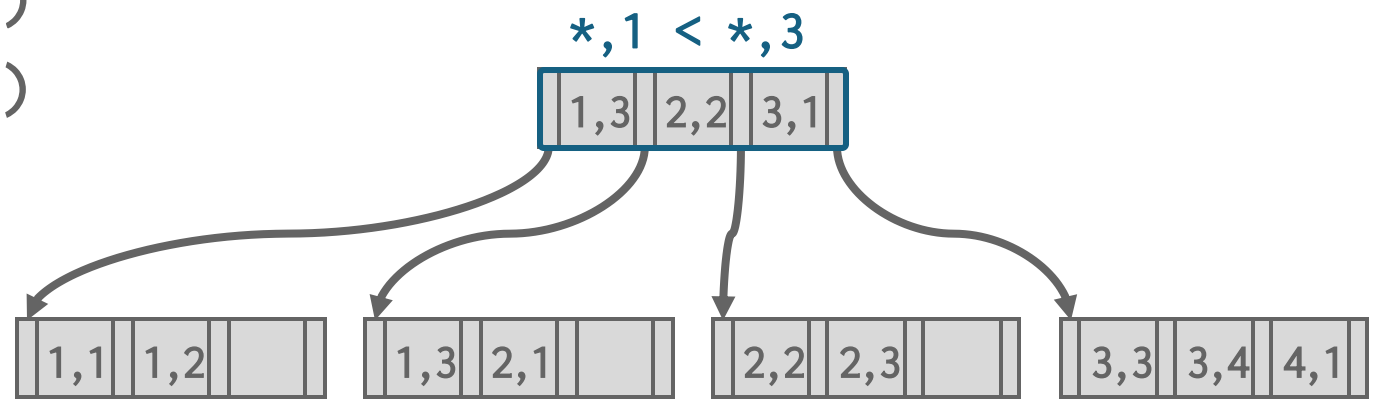
# Selection Conditions

```
Find Key=(1,2)
Find Key=(1,*)
Find Key=(*,1)
```



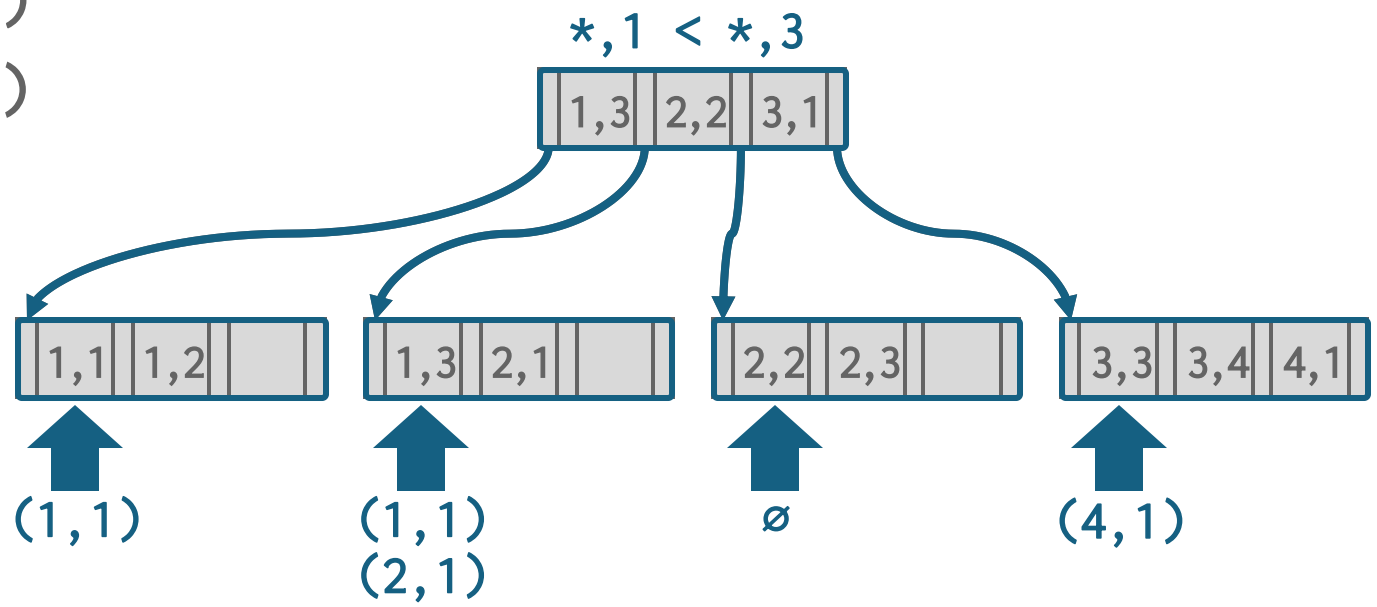
# Selection Conditions

```
Find Key=(1,2)
Find Key=(1,*)
Find Key=(*,1)
```



# Selection Conditions

```
Find Key=(1,2)
Find Key=(1,*)
Find Key=(*,1)
```

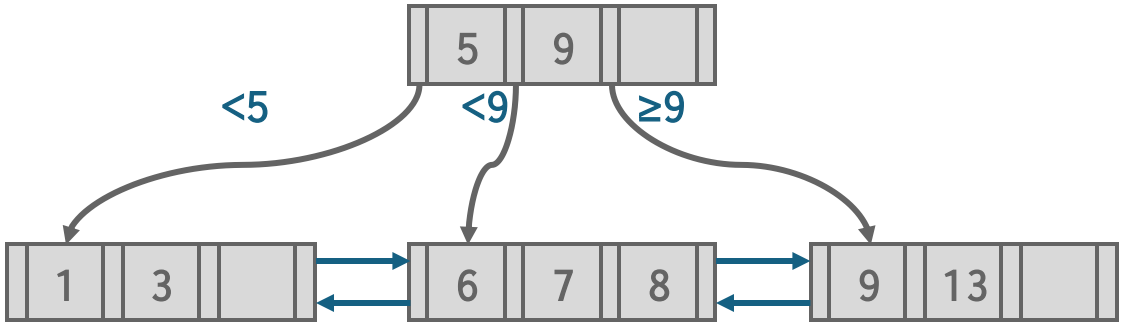


# B+Tree: Duplicate Keys

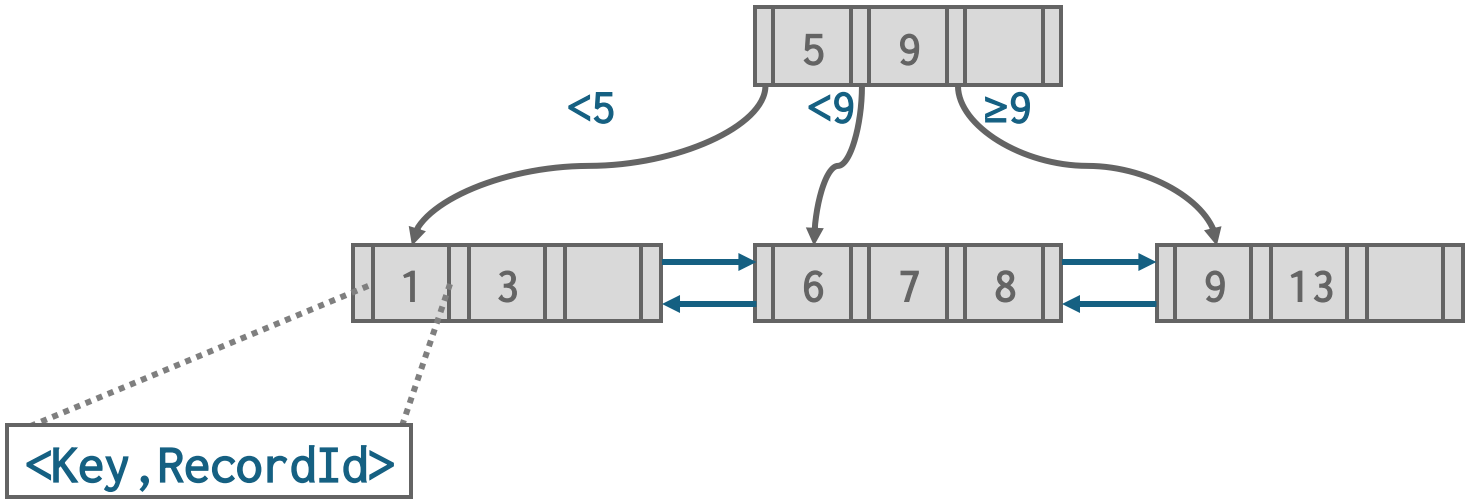
- **Approach #1: Append Record ID**
  - Add the tuple's unique Record ID as part of the key to ensure that all keys are unique.
  - The DBMS can still use partial keys to find tuples.

- **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

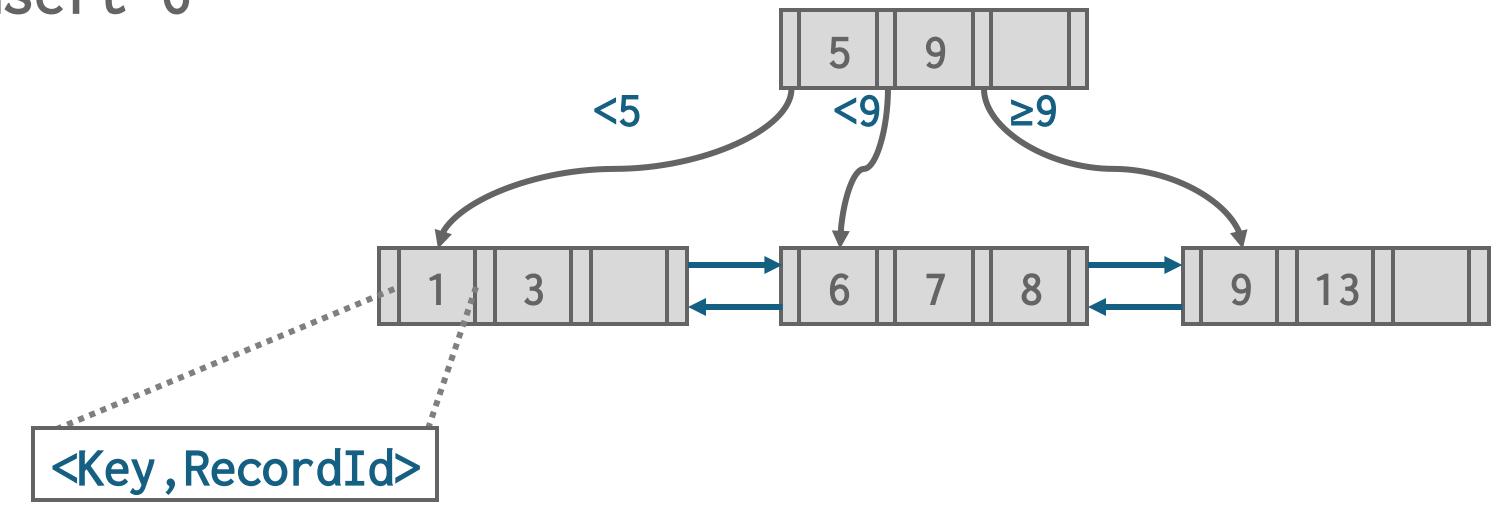


# B+Tree: Append Record ID

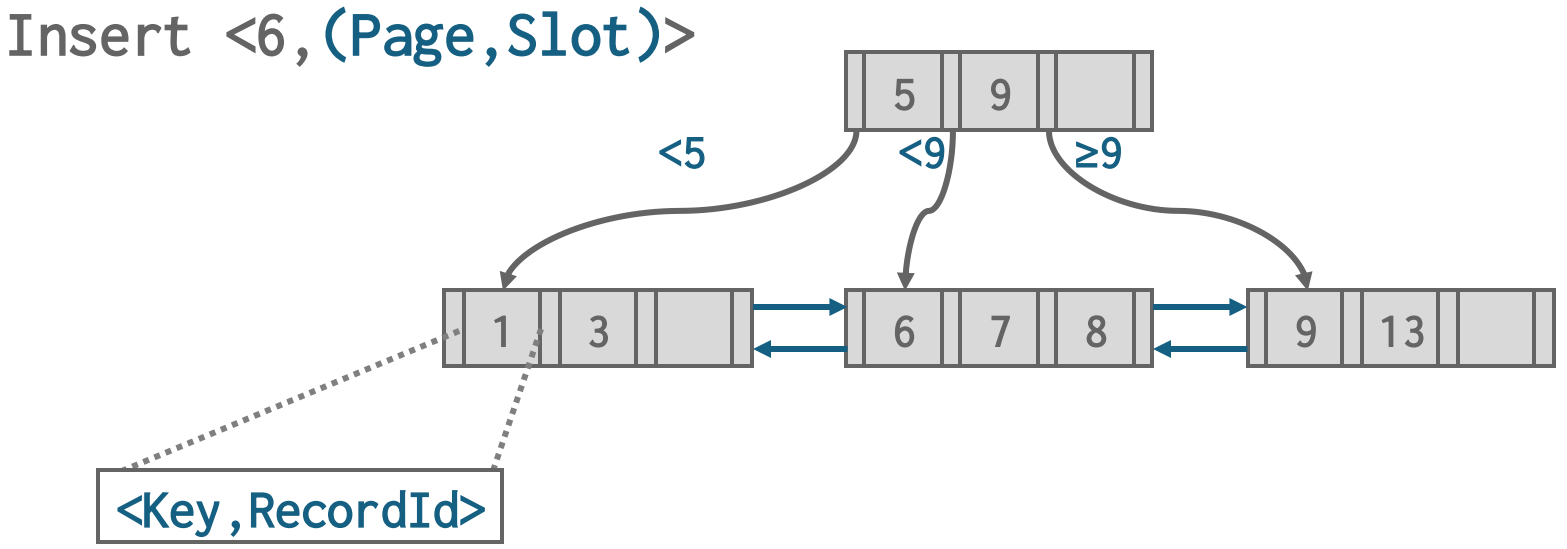


# B+Tree: Append Record ID

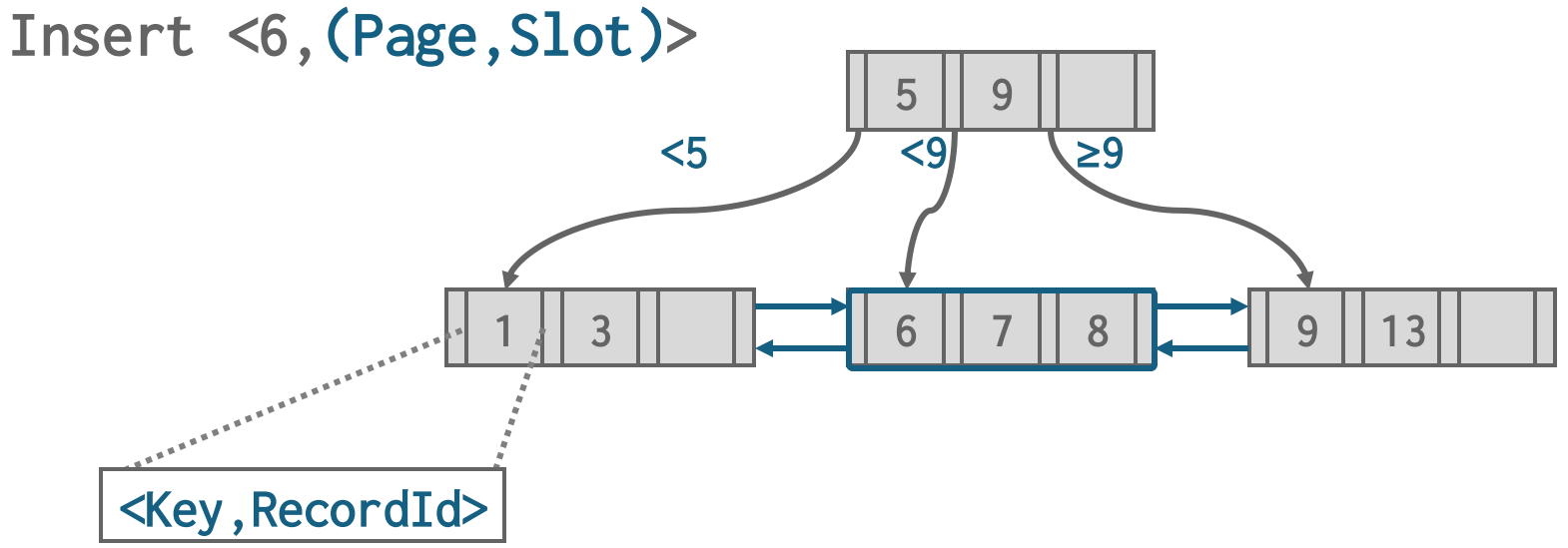
Insert 6



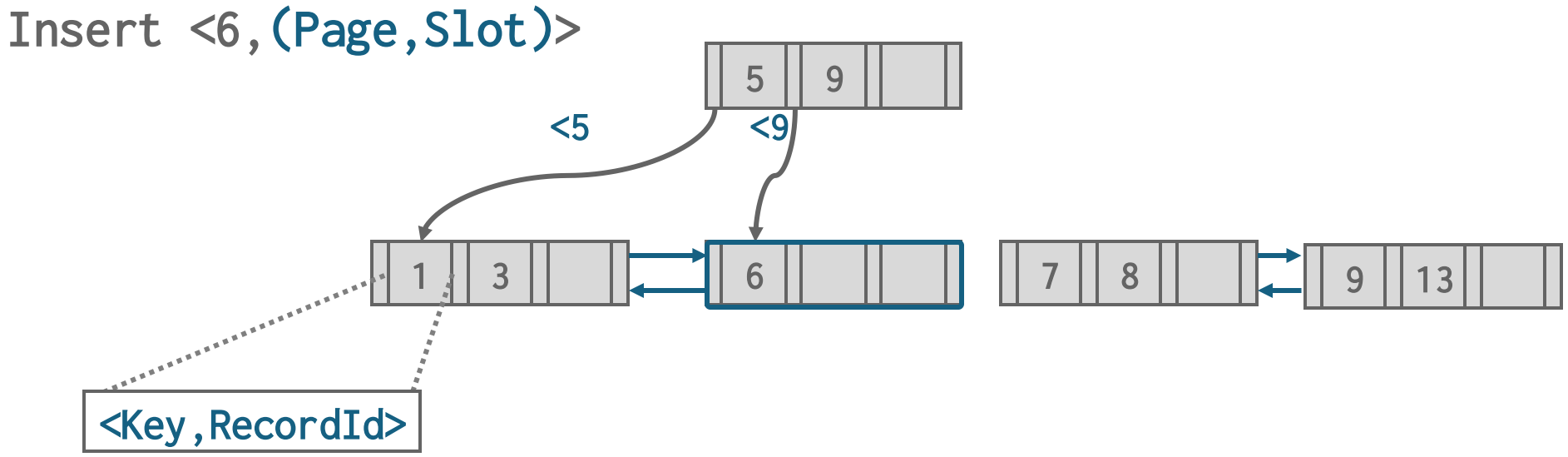
# B+Tree: Append Record ID



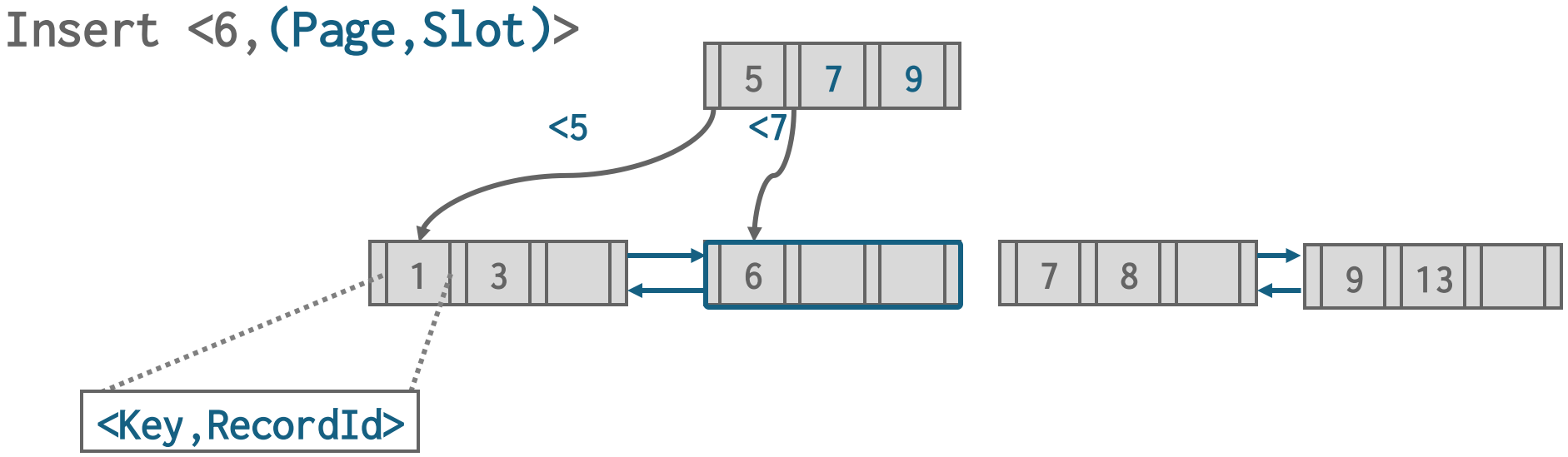
# B+Tree: Append Record ID



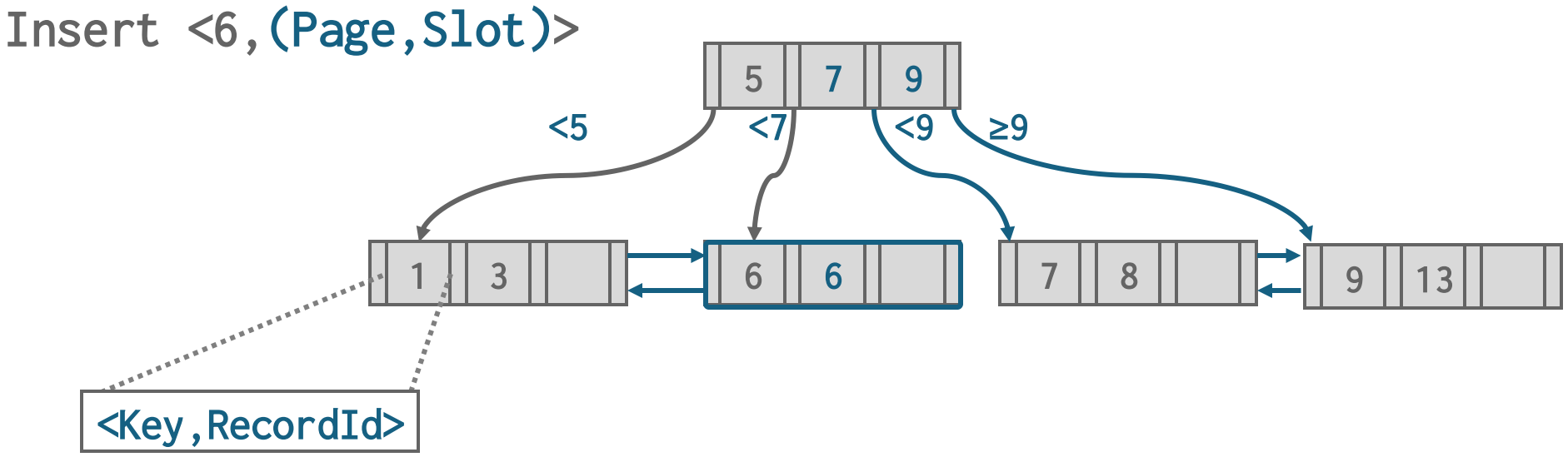
# B+Tree: Append Record ID



# B+Tree: Append Record ID

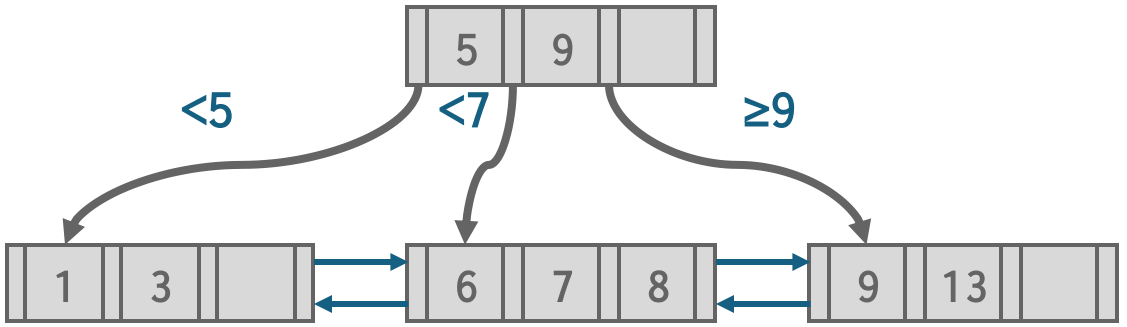


# B+Tree: Append Record ID



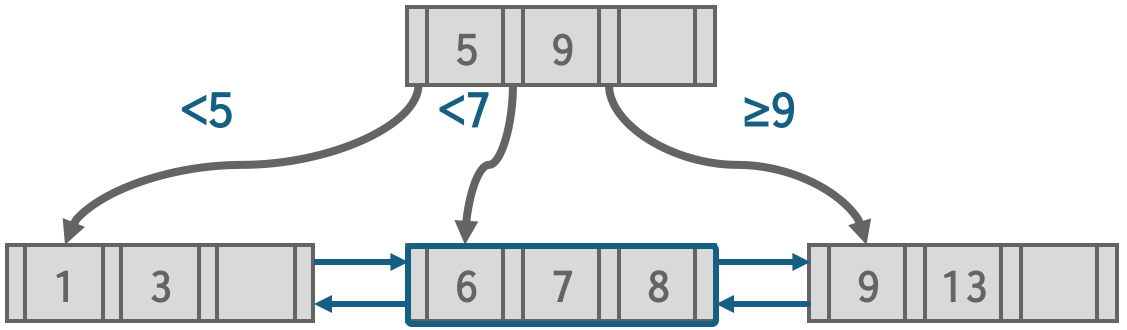
# B+Tree: Overflow Leaf Nodes

Insert 6



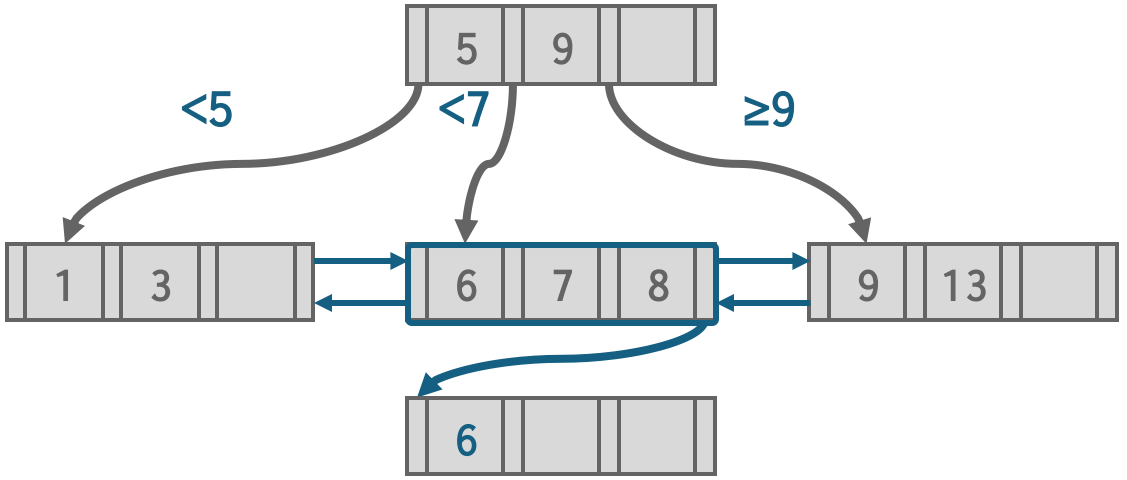
# B+Tree: Overflow Leaf Nodes

Insert 6



# B+Tree: Overflow Leaf Nodes

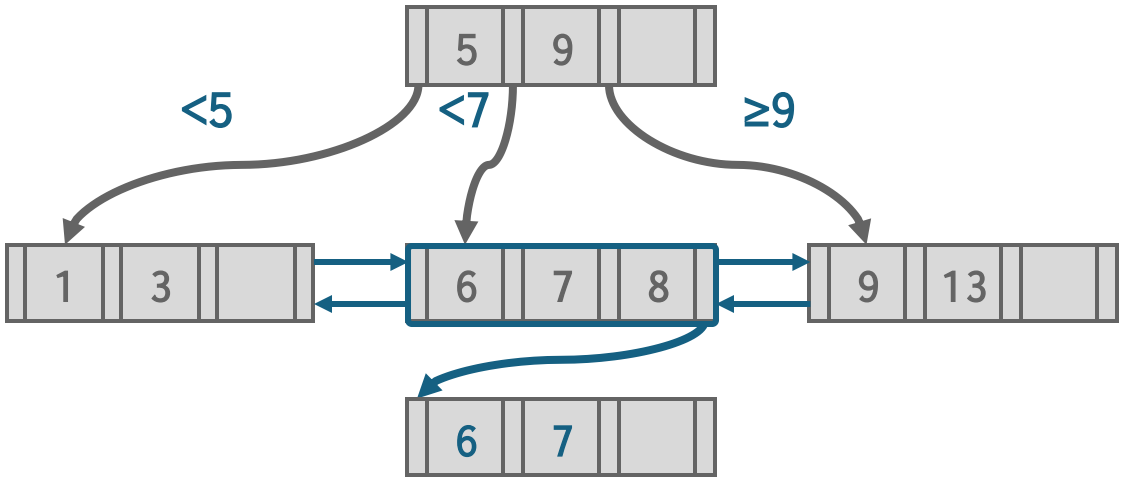
Insert 6



# B+Tree: Overflow Leaf Nodes

Insert 6

Insert 7

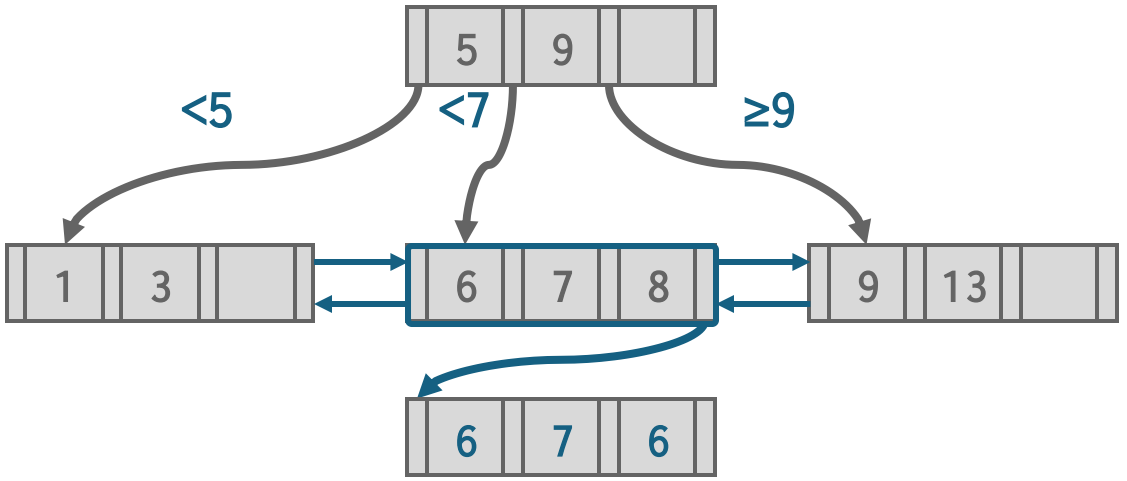


# B+Tree: Overflow Leaf Nodes

Insert 6

Insert 7

Insert 6

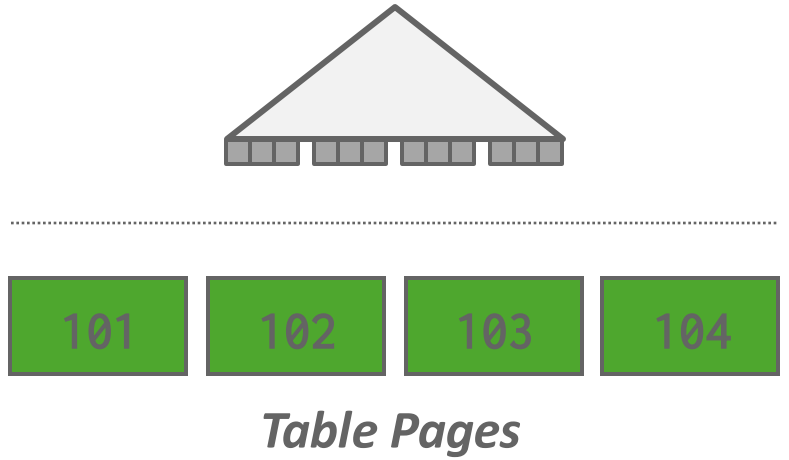


# Clustered Indexes

- 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.

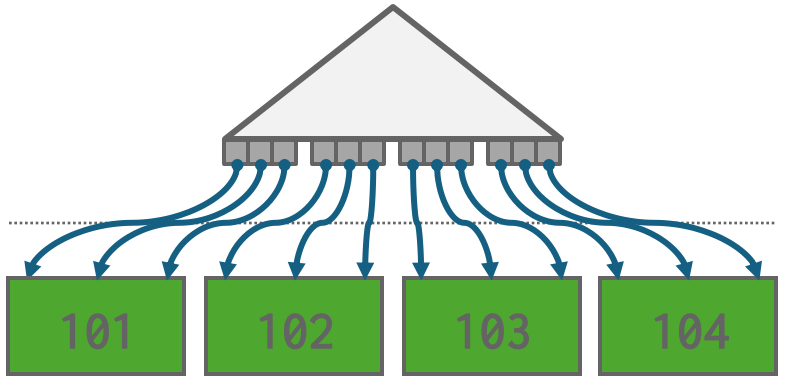
# Clustered B+Tree

- 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.



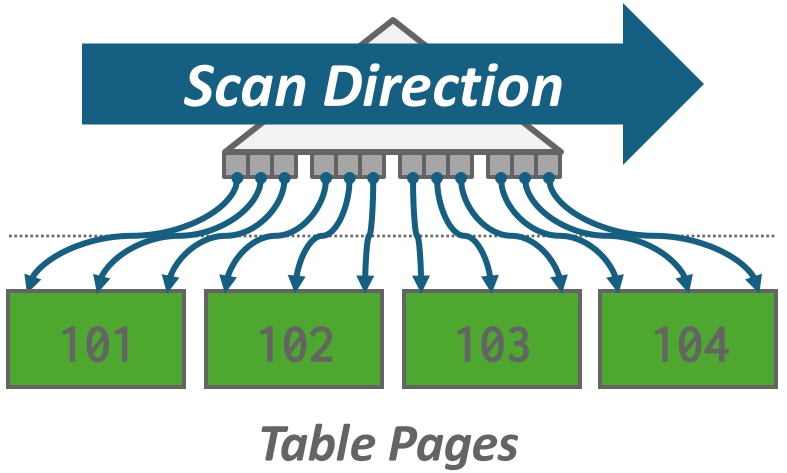
# Clustered B+Tree

- 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.



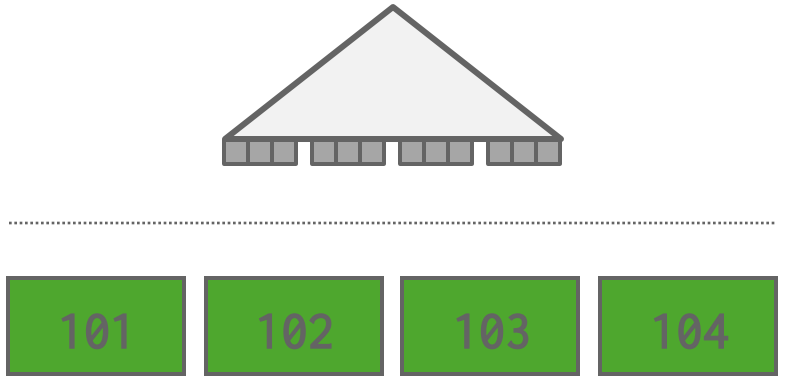
# Clustered B+Tree

- 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.



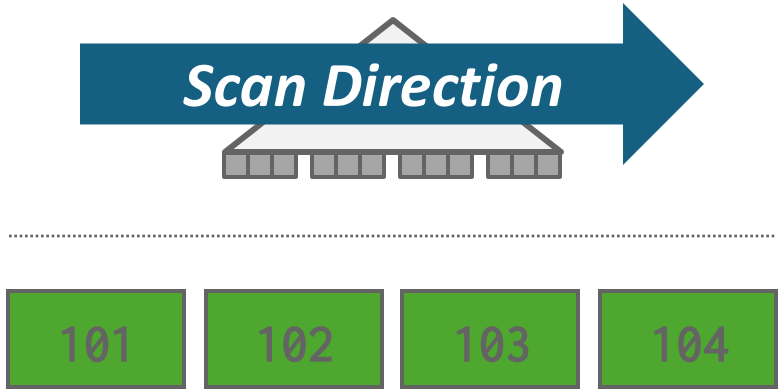
# Index Scan Page Sorting

- Retrieving tuples in the order they appear in a non-clustered index is 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.



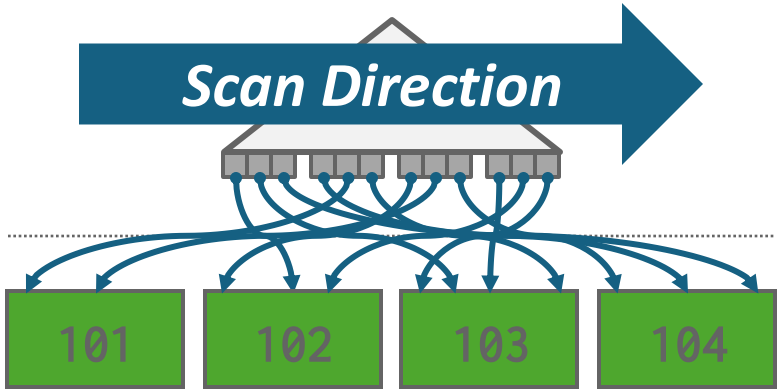
# Index Scan Page Sorting

- Retrieving tuples in the order they appear in a non-clustered index is 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.



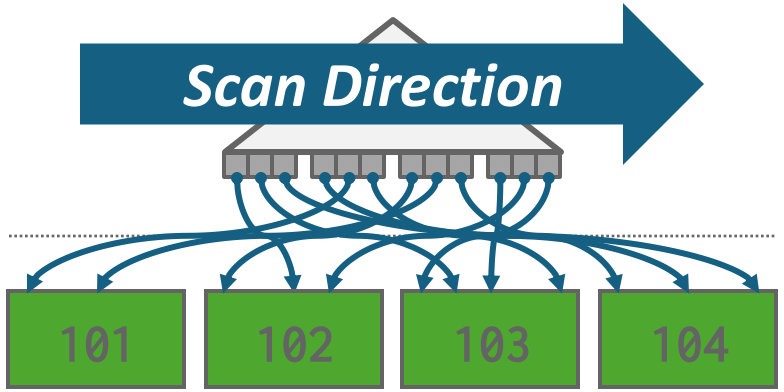
# Index Scan Page Sorting

- Retrieving tuples in the order they appear in a non-clustered index is 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.



# Index Scan Page Sorting

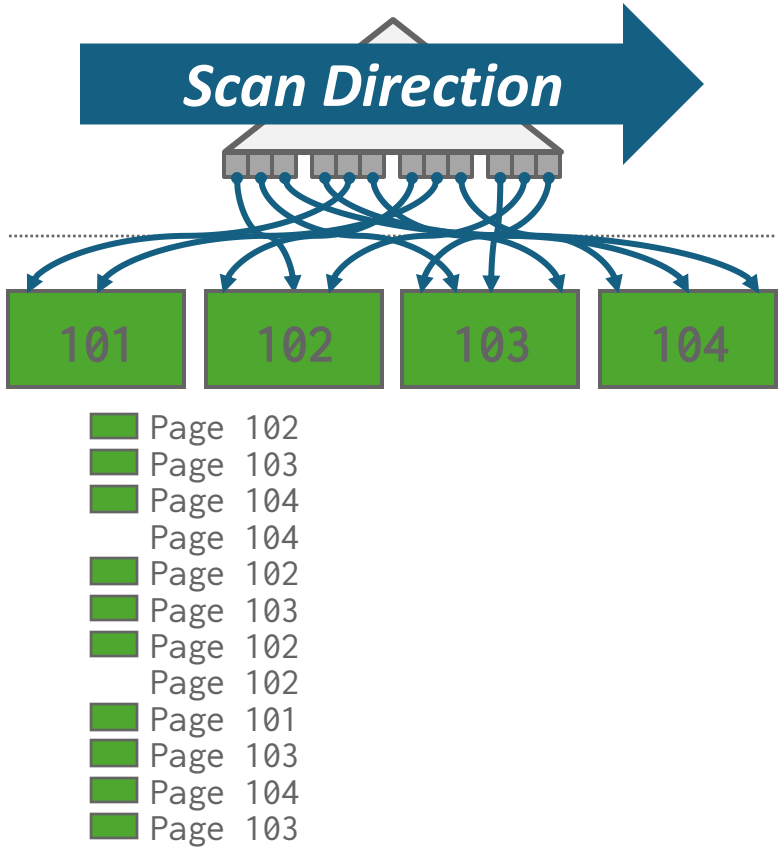
- Retrieving tuples in the order they appear in a non-clustered index is inefficient due to redundant reads.
- A better approach is to find all the tuples that the query needs and then sort them based on their page ID.
- The DBMS retrieves each page once.




Page 102
Page 103
Page 104
Page 104
Page 102
Page 103
Page 102
Page 102
Page 101
Page 103
Page 104
Page 103

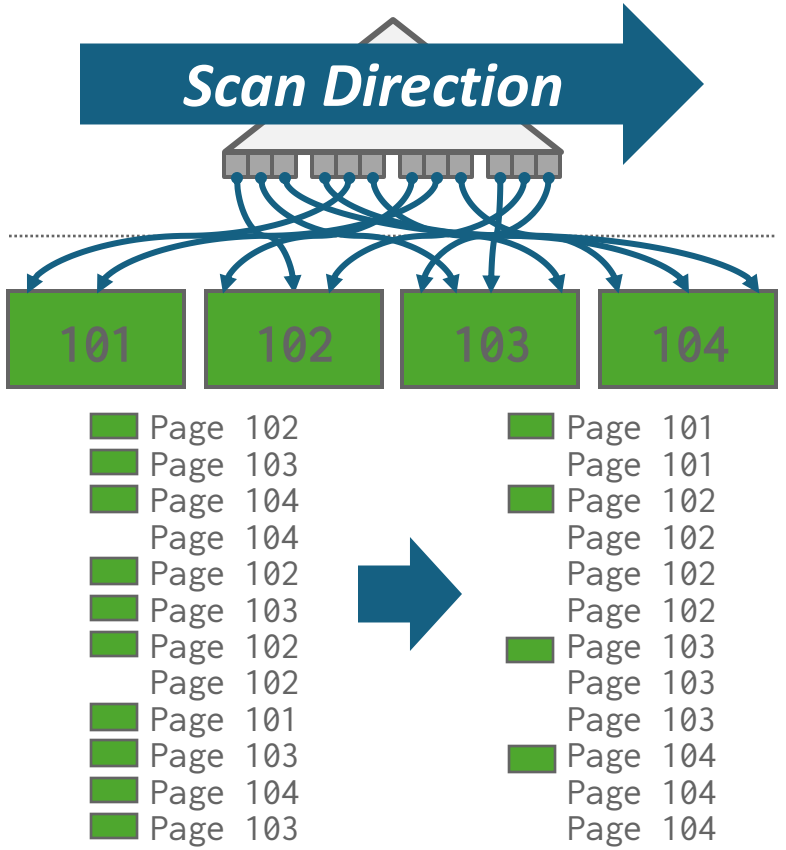
# Index Scan Page Sorting

- Retrieving tuples in the order they appear in a non-clustered index is 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.



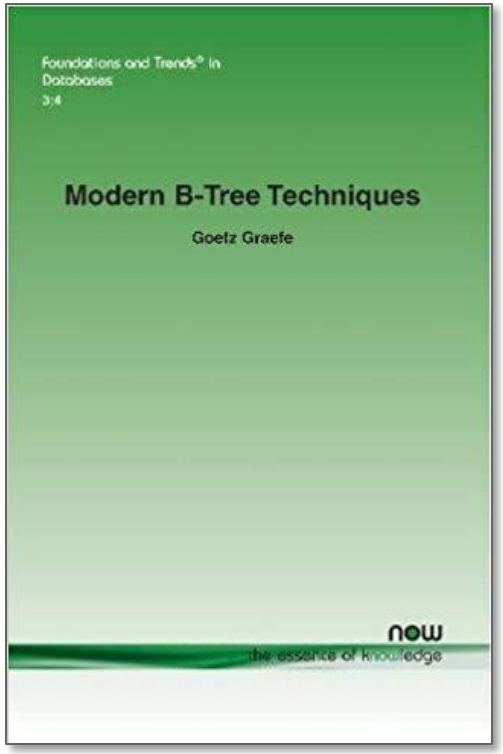
# Index Scan Page Sorting

- Retrieving tuples in the order they appear in a non-clustered index is 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.



# 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 just 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.

# Intra-Node Search

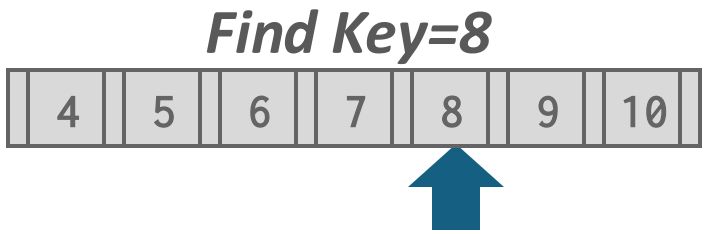
- **Approach #1: Linear**
  - Scan node keys from beginning to end.
  - Use SIMD to vectorize comparisons.

# Intra-Node Search

- **Approach #1: Linear**
  - Scan node keys from beginning to end.
  - Use SIMD to vectorize comparisons.

# Intra-Node Search

- **Approach #1: Linear**
  - Scan node keys from beginning to end.
  - Use SIMD to vectorize comparisons.



# Intra-Node Search

- **Approach #1: Linear**
  - Scan node keys from beginning to end.
  - Use SIMD to vectorize comparisons.

# Intra-Node Search

- **Approach #1: Linear**
  - Scan node keys from beginning to end.
  - Use SIMD to vectorize comparisons.

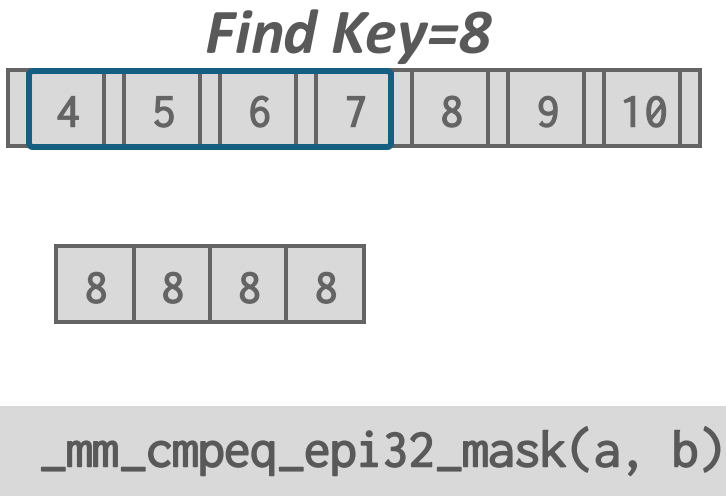
# Intra-Node Search

- **Approach #1: Linear**
  - Scan node keys from beginning to end.
  - Use SIMD to vectorize comparisons.


```
_mm_cmpeq_epi32_mask(a, b)
```

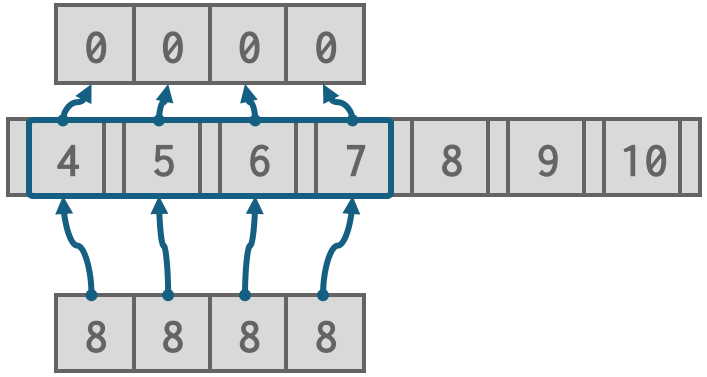
# Intra-Node Search

- **Approach #1: Linear**
  - Scan node keys from beginning to end.
  - Use SIMD to vectorize comparisons.



# Intra-Node Search

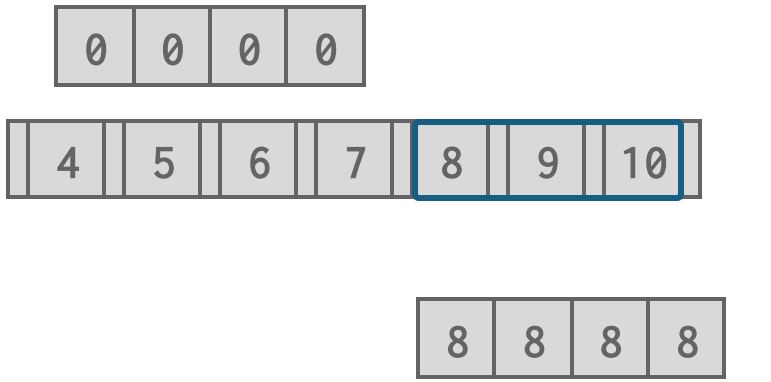
- **Approach #1: Linear**
  - Scan node keys from beginning to end.
  - Use SIMD to vectorize comparisons.




```
_mm_cmpeq_epi32_mask(a, b)
```

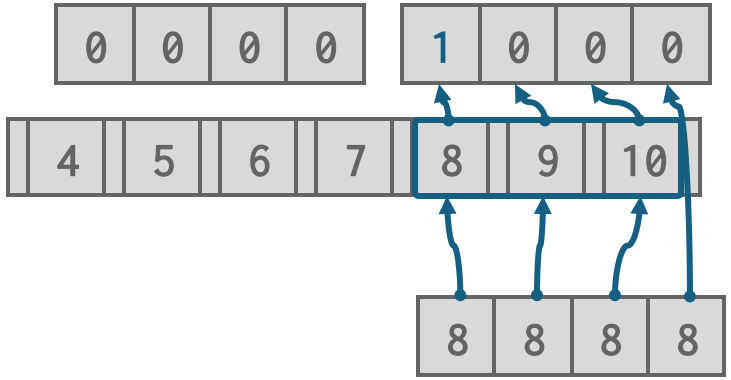
# Intra-Node Search

- **Approach #1: Linear**
  - Scan node keys from beginning to end.
  - Use SIMD to vectorize comparisons.



# Intra-Node Search

- **Approach #1: Linear**
  - Scan node keys from beginning to end.
  - Use SIMD to vectorize comparisons.



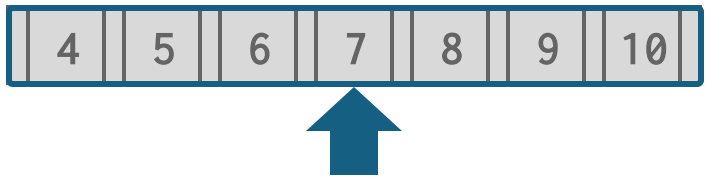

```
_mm_cmpeq_epi32_mask(a, b)
```

# 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.

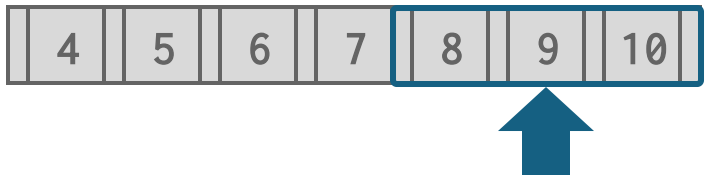
# Intra-Node Search

- **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.



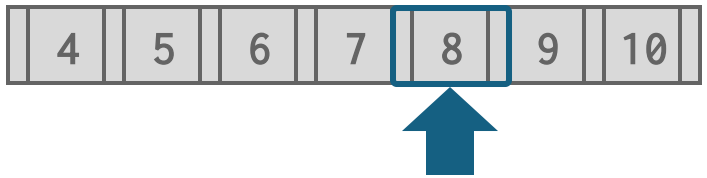
# Intra-Node Search

- **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.



# 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.

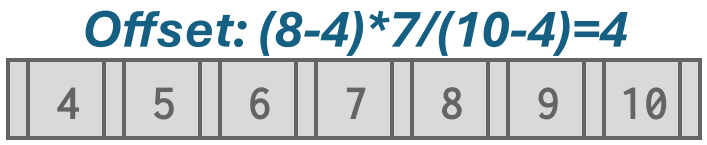


# Intra-Node Search

- **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.

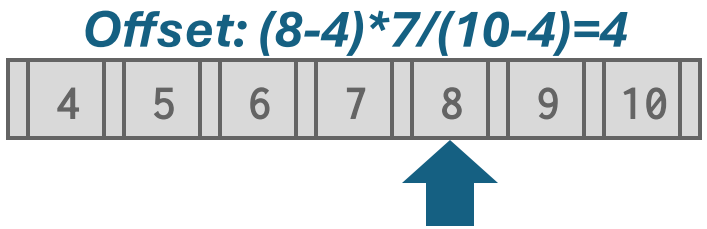
# Intra-Node Search

- **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.



# 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...

# Optimizations

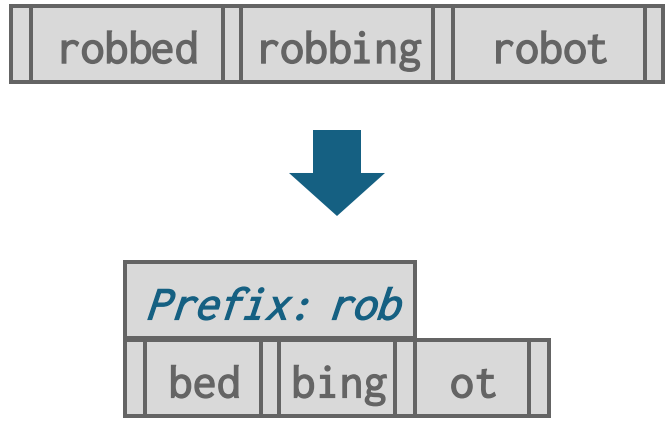
- Prefix Compression
- Deduplication
- Suffix Truncation
- Pointer Swizzling
- Bulk Insert
- Buffered Updates
- Many more…

# 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.

# 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

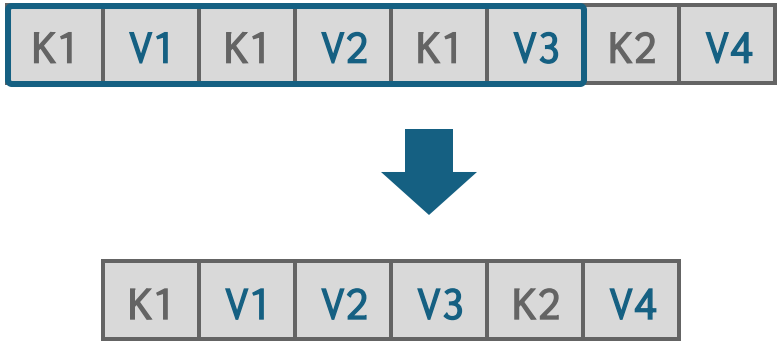
- Non-unique indexes can end up storing multiple copies of the same key in leaf nodes.
- The leaf node can store the key once and then maintain a “posting list” of tuples with that key (similar to what we discussed for hash tables).

# 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 (similar to what we discussed for hash tables).

# Deduplication

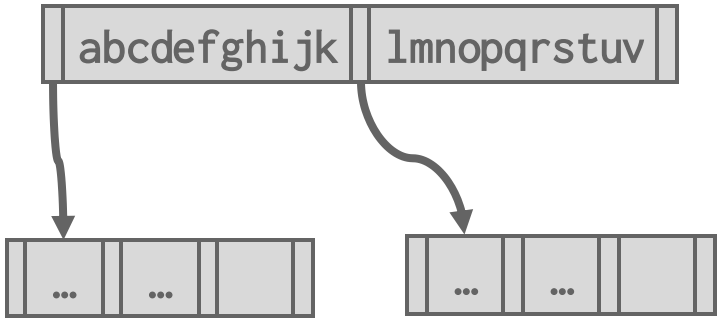
- Non-unique indexes can end up storing multiple copies of the same key in leaf nodes.
- The leaf node can store the key once and then maintain a “posting list” of tuples with that key (similar to what we discussed for hash tables).



# Suffix Truncation

- The keys in the inner nodes are only used to “direct traffic”.
  - We don't need the entire key.

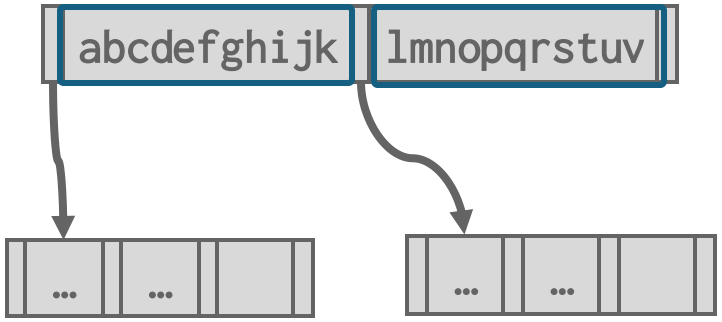
- Store a minimum prefix that is needed to correctly route probes into the index.



# Suffix Truncation

- The keys in the inner nodes are only used to “direct traffic”.
  - We don't need the entire key.

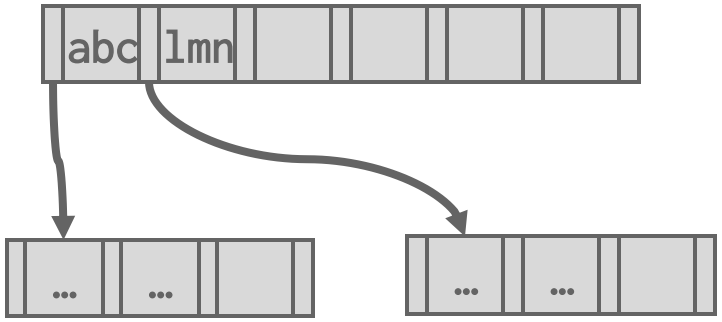
- Store a minimum prefix that is needed to correctly route probes into the index.



# 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.



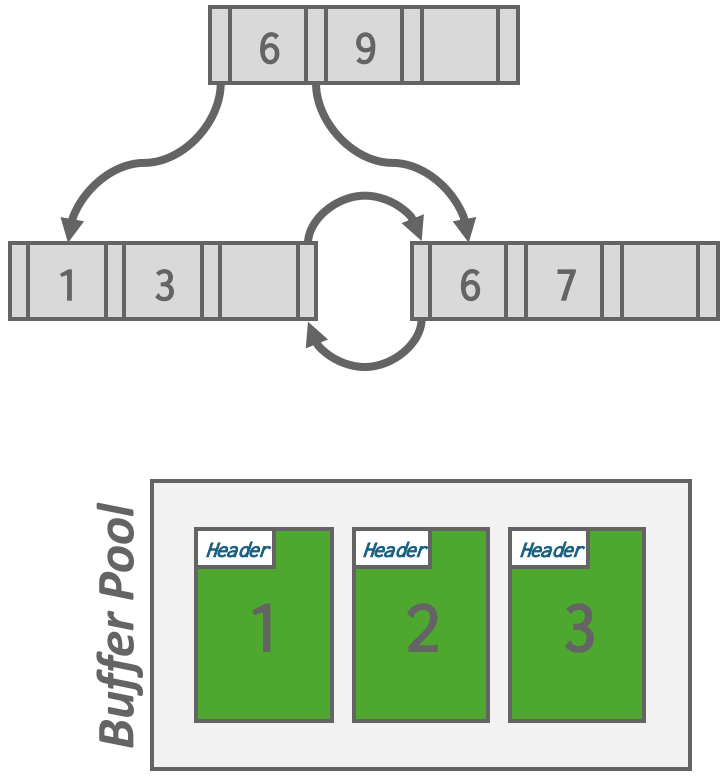
# Pointer Swizzling

- 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.

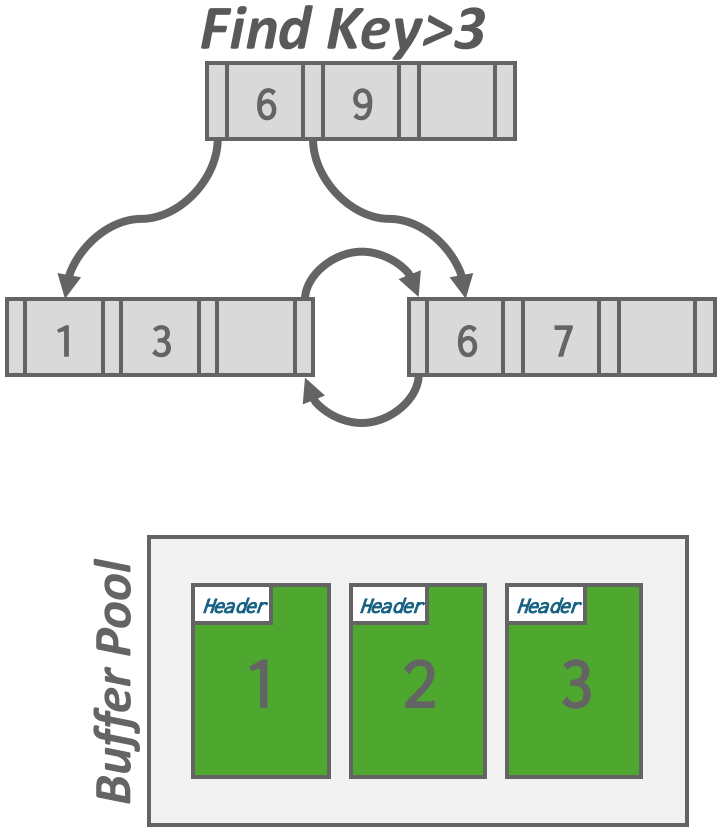
# Pointer Swizzling

- 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.



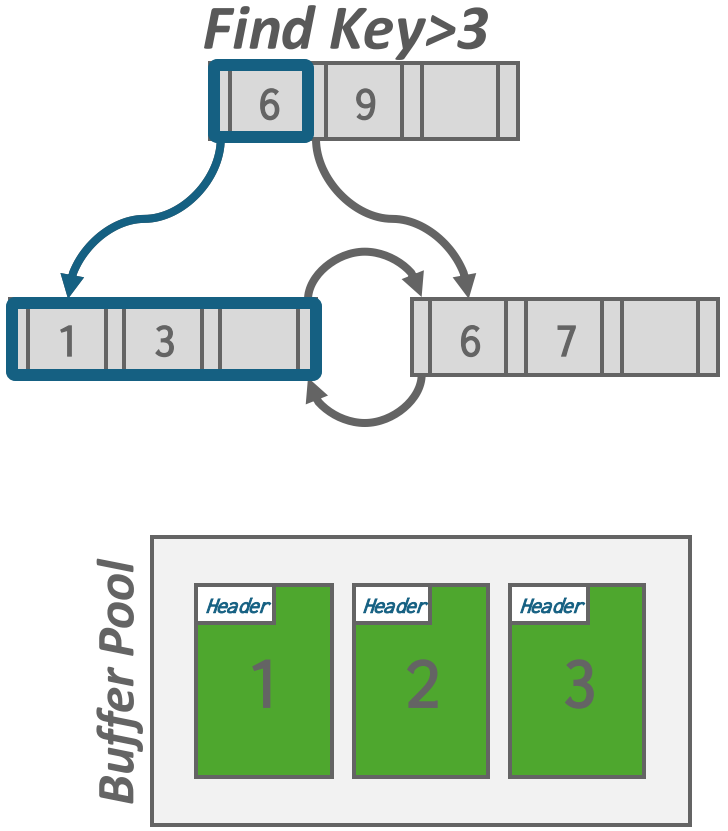
# Pointer Swizzling

- 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.



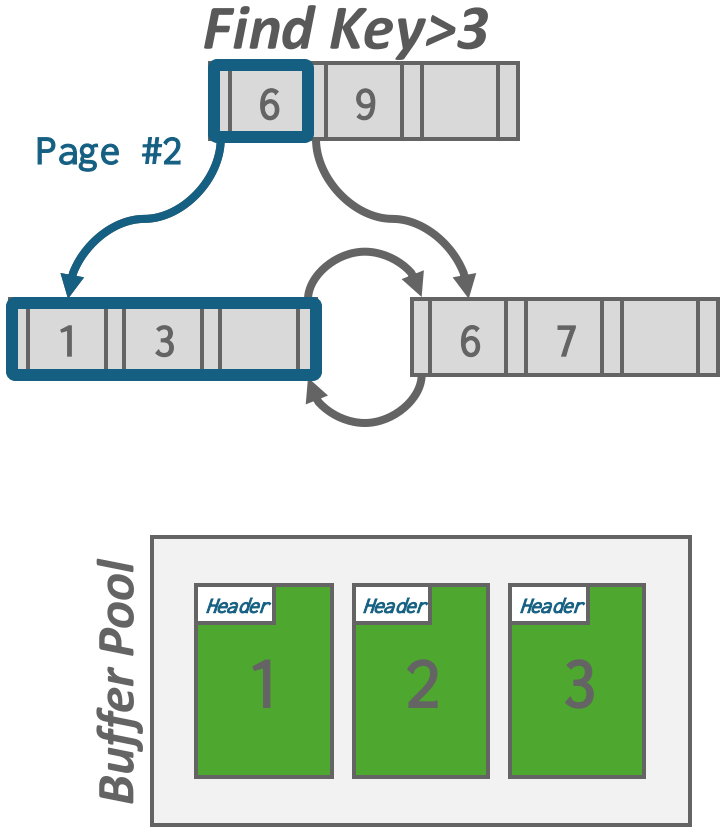
# Pointer Swizzling

- 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.



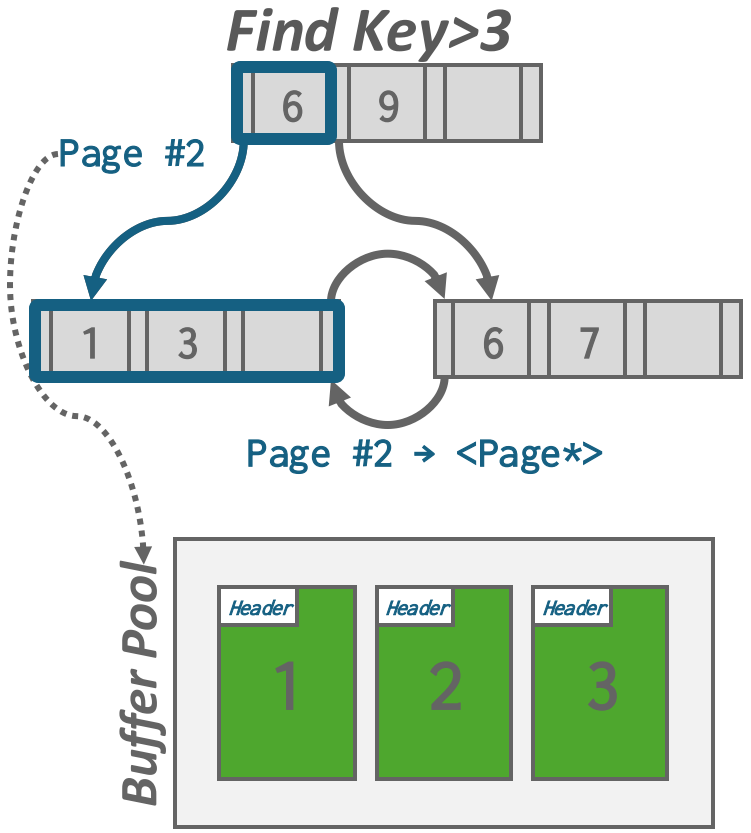
# Pointer Swizzling

- 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.



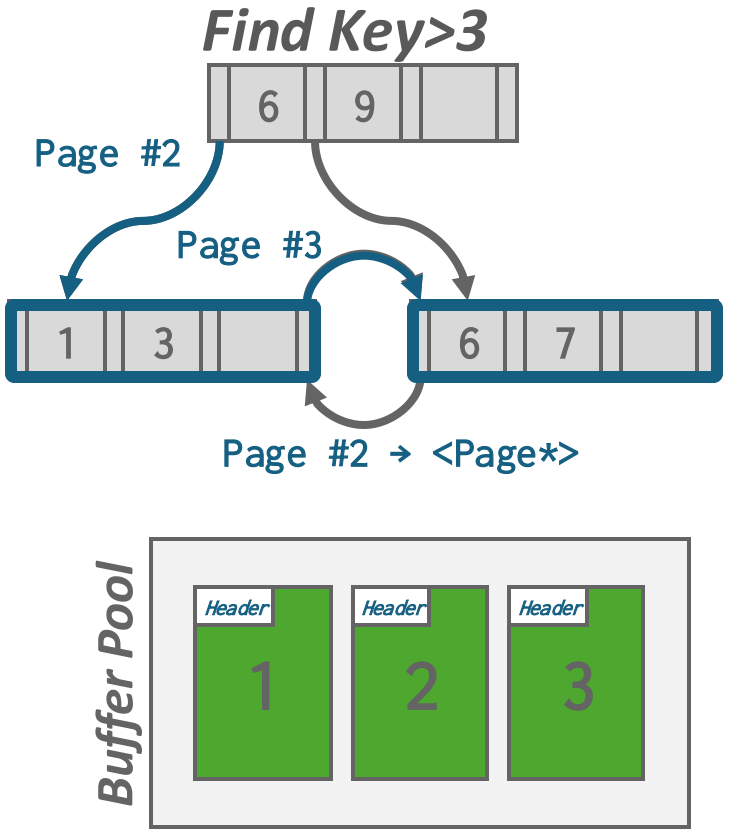
# Pointer Swizzling

- 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.



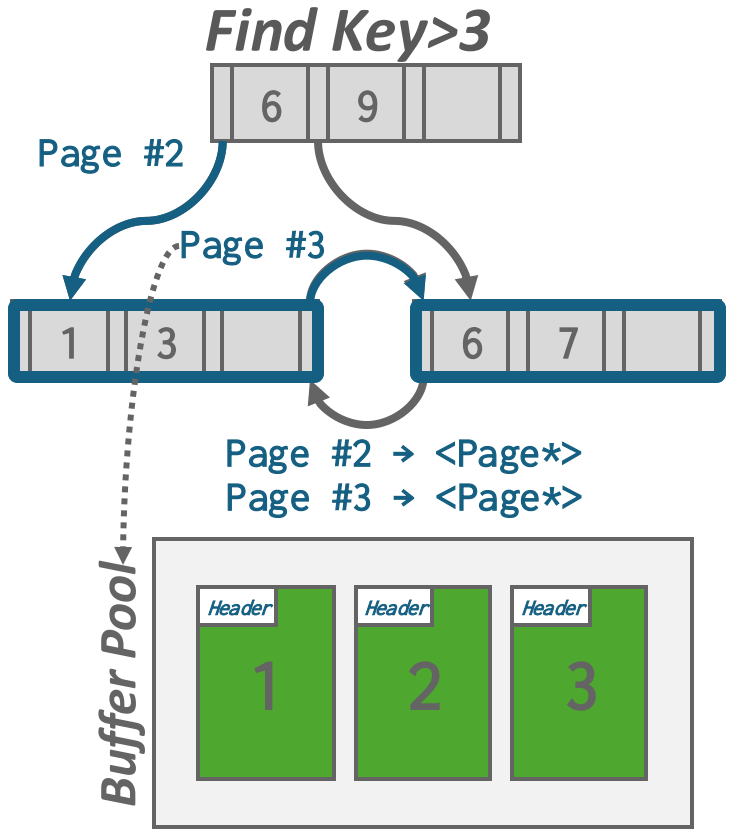
# Pointer Swizzling

- 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.



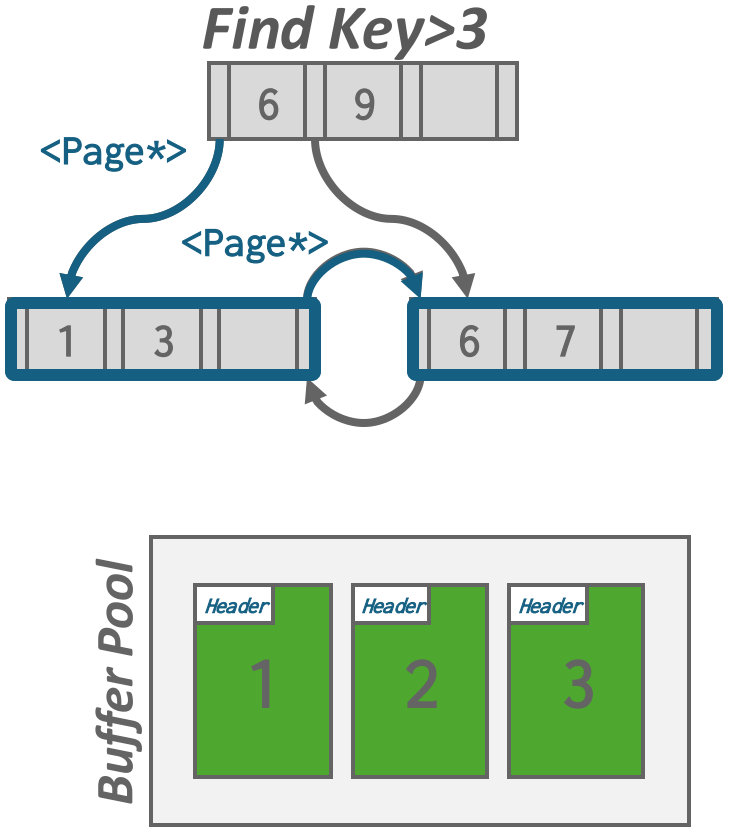
# Pointer Swizzling

- 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.



# 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.

# 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**

# 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**

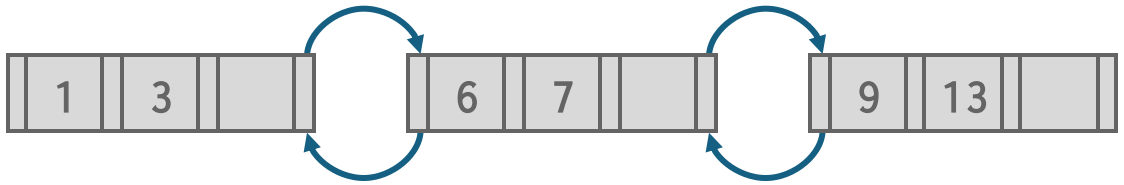
**Sorted Keys: 1, 3, 6, 7, 9, 13**

# 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**

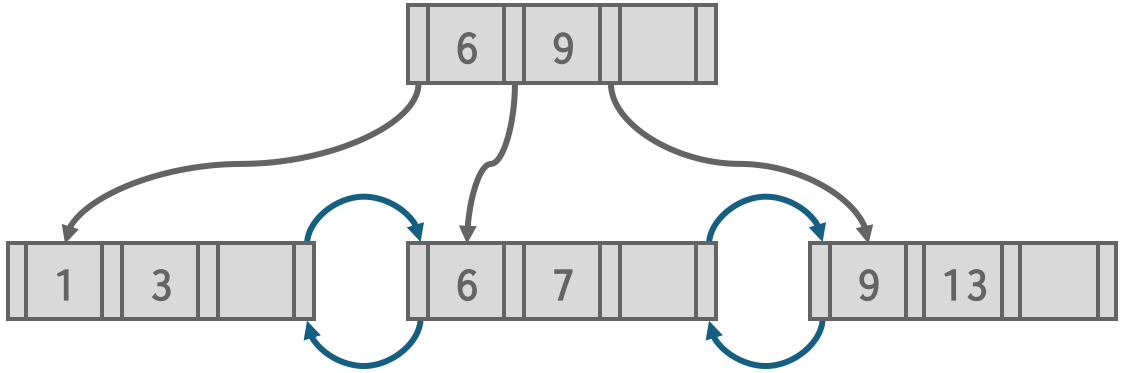


# Bulk Insert

- 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**



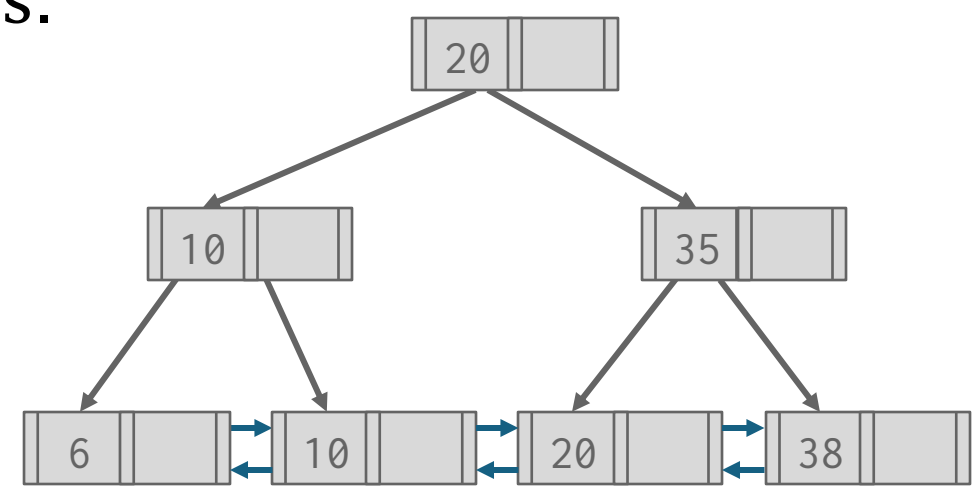
# Observation

- 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

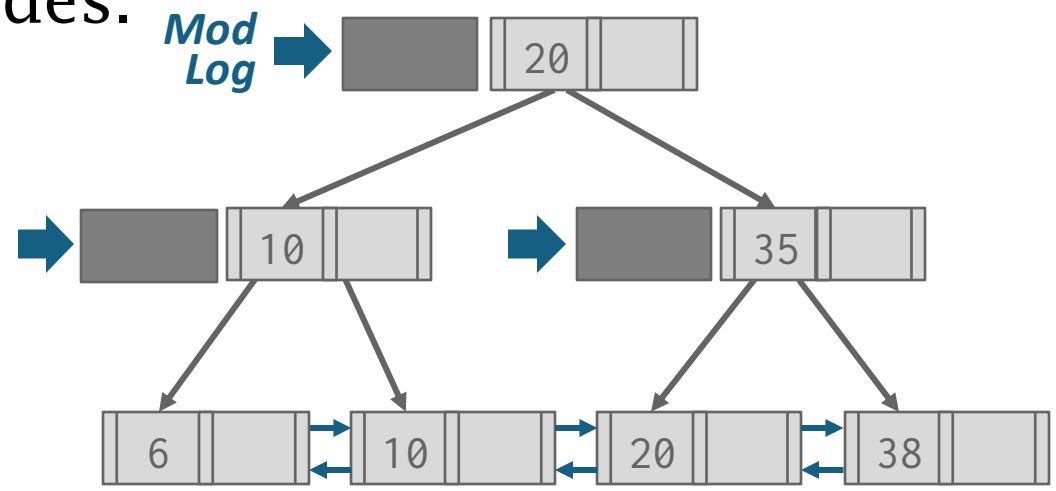
- Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes.
  - Also known as **Bε-trees**.
- Updates cascade down to lower nodes incrementally when buffers get full.




Tokutek SPLINTERDB RelationalAI

# Write-Optimized B+Tree

- Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes.
  - Also known as **Bε-trees**.
- Updates cascade down to lower nodes incrementally when buffers get full.

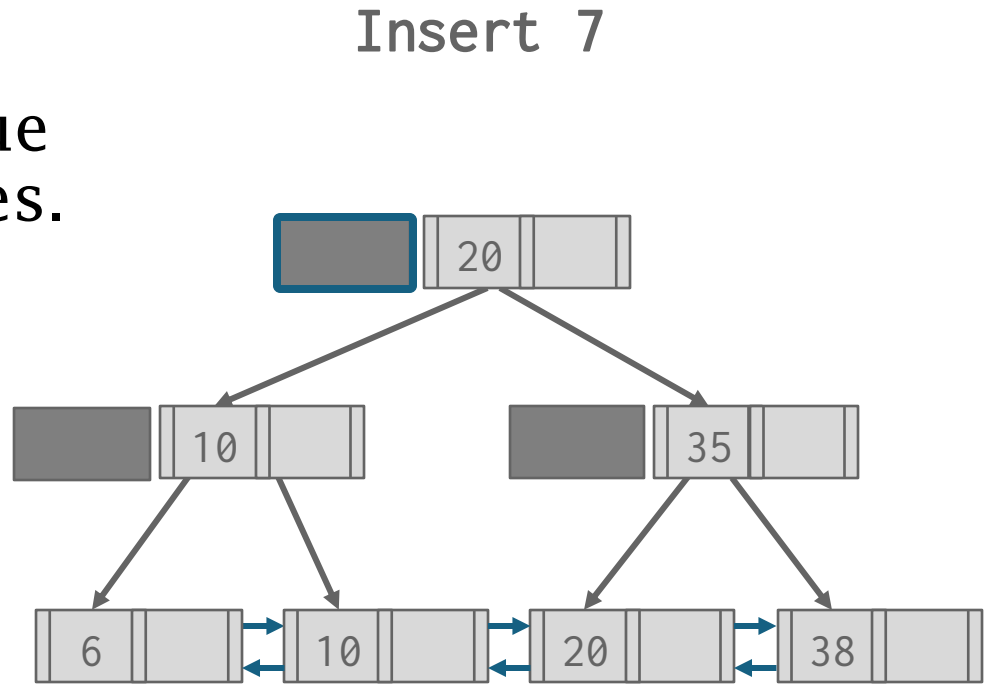



Tokutek SPLINTERDB

RelationalAI

# Write-Optimized B+Tree

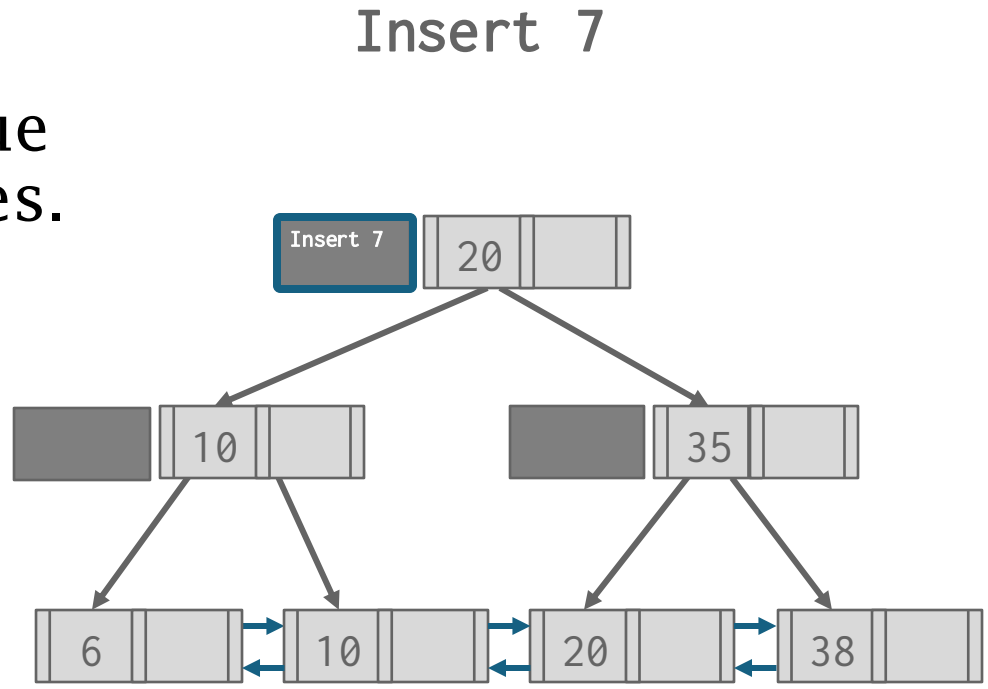
- Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes.
  - Also known as **Bε-trees**.
- Updates cascade down to lower nodes incrementally when buffers get full.




Tokutek SPLINTERDB RelationalAI

# Write-Optimized B+Tree

- Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes.
  - Also known as **Bε-trees**.
- Updates cascade down to lower nodes incrementally when buffers get full.

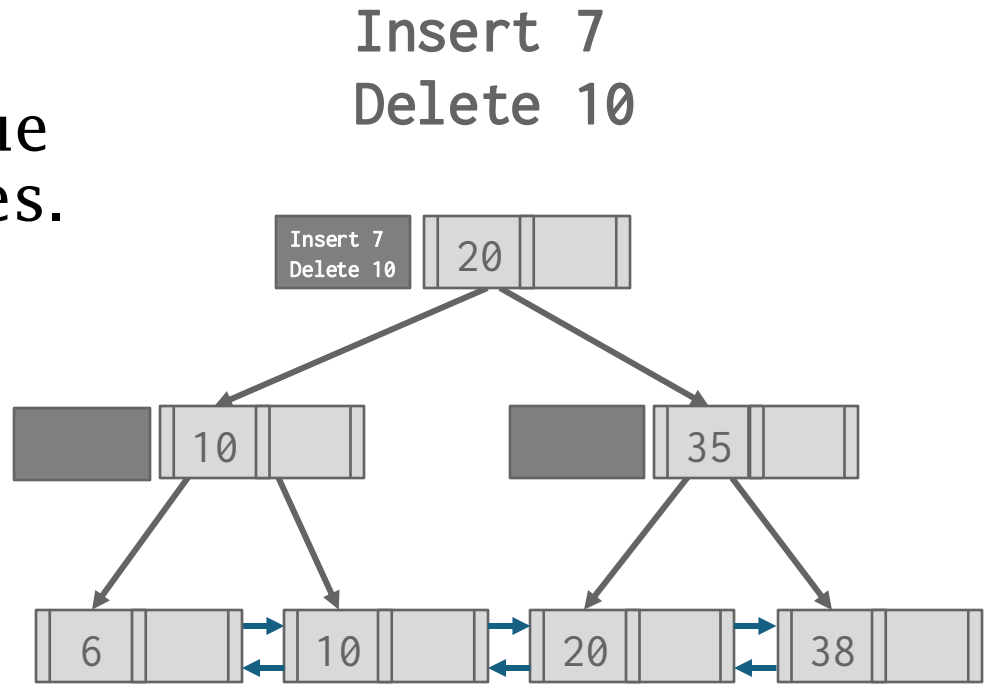



Tokutek SPLINTERDB

RelationalAI

# Write-Optimized B+Tree

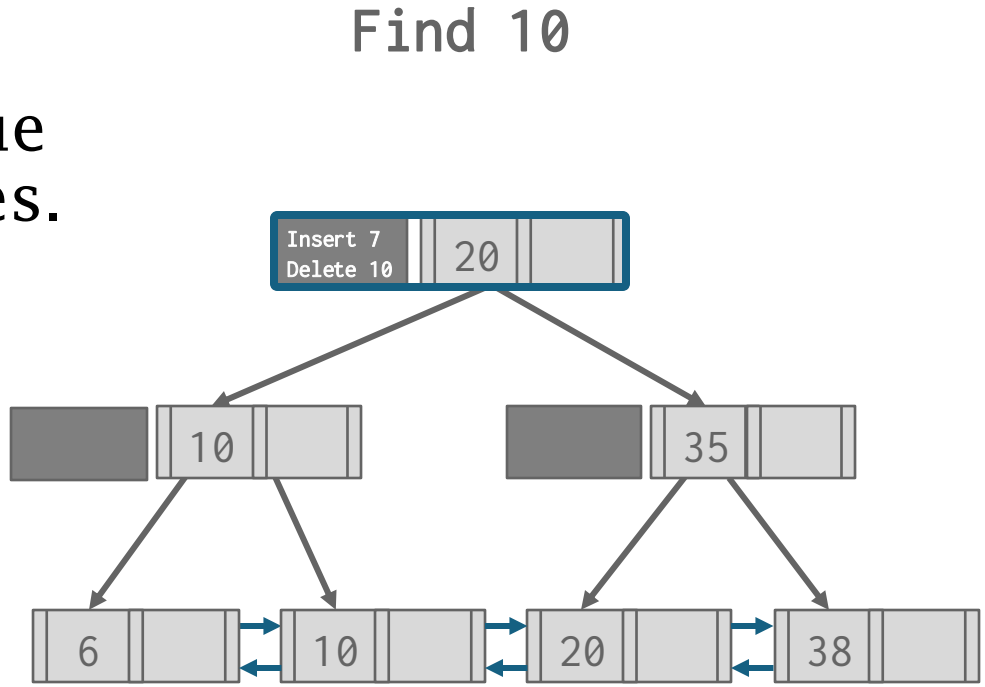
- Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes.
  - Also known as **Bε-trees**.
- Updates cascade down to lower nodes incrementally when buffers get full.




Tokutek SPLINTERDB RelationalAI

# Write-Optimized B+Tree

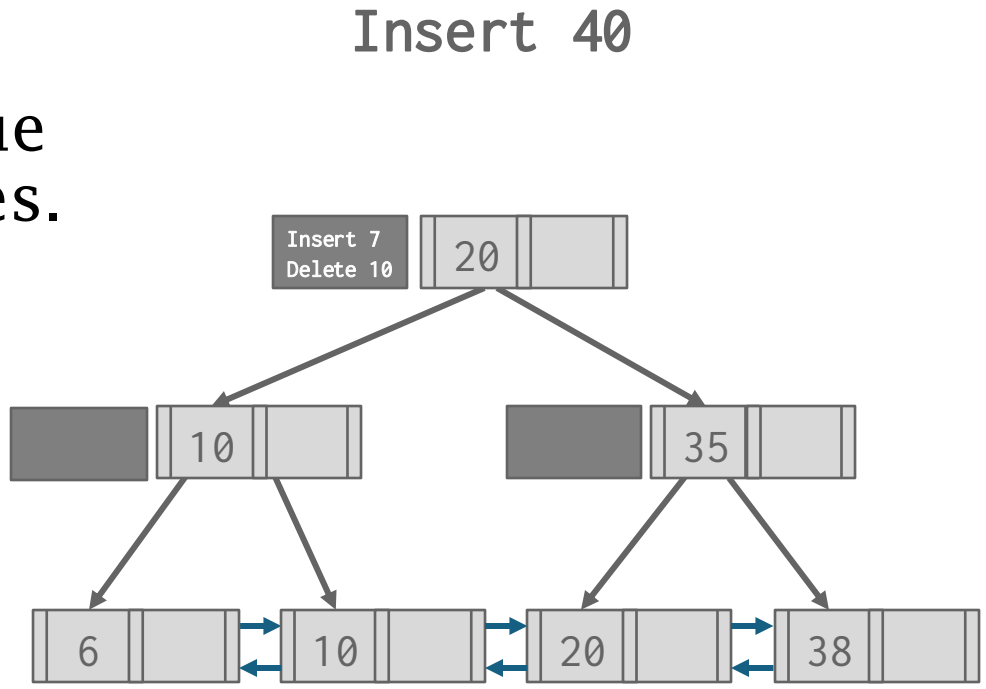
- Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes.
  - Also known as **Bε-trees**.
- Updates cascade down to lower nodes incrementally when buffers get full.




Tokutek SPLINTERDB RelationalAI

# Write-Optimized B+Tree

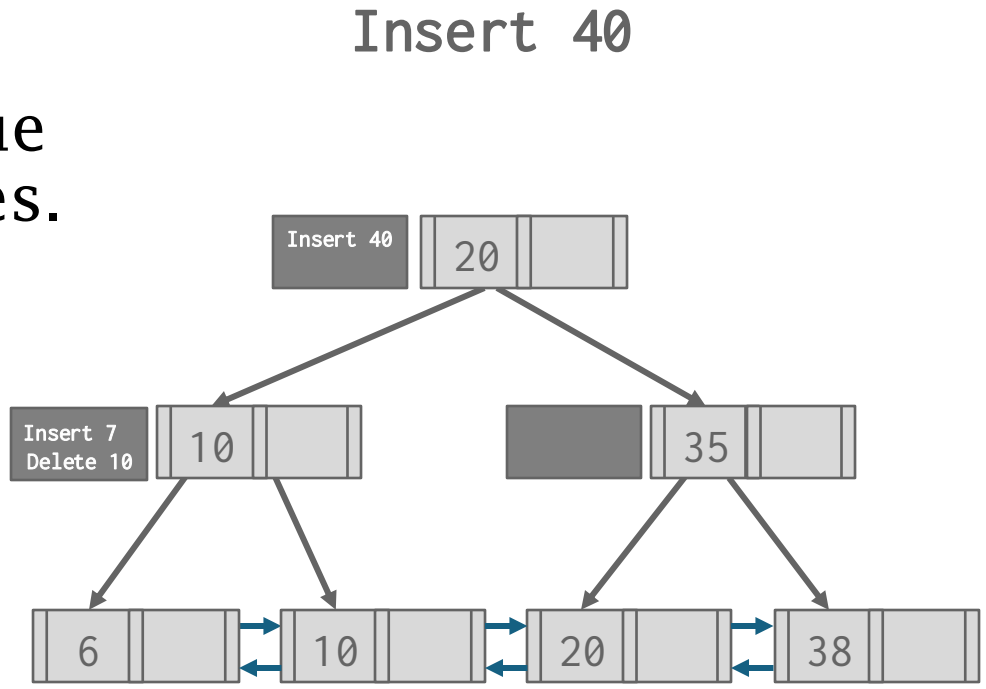
- Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes.
  - Also known as **Bε-trees**.
- Updates cascade down to lower nodes incrementally when buffers get full.




Tokutek SPLINTERDB RelationalAI

# Write-Optimized B+Tree

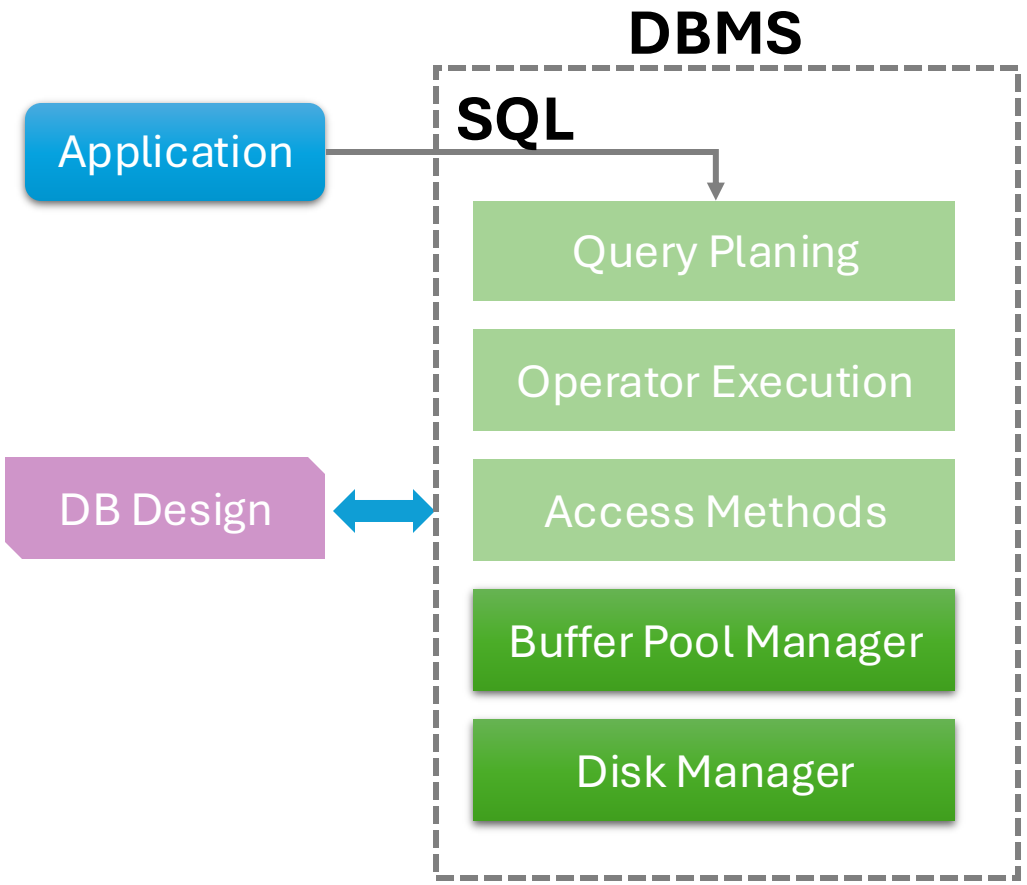
- Instead of immediately applying updates, store changes to key/value entries in log buffers at inner nodes.
  - Also known as **Bε-trees**.
- Updates cascade down to lower nodes incrementally when buffers get full.




Tokutek SPLINTERDB RelationalAI

# Conclusion

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



# Conclusion

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