Carnegie Mellon University

Intro to Database Systems (15-445/645)

Lecture #08

B+Tree Index

SPRING 2024 ❯❯ Prof. Jignesh Patel

LAST CLASS

Hash tables are important data structures that are used all throughout a DBMS.

- → Space Complexity: **O(n)**
- → Average Time Complexity: **O(1)**

Static vs. Dynamic Hashing schemes

DBMSs use mostly hash tables for their internal data structures.

TODAY'S AGENDA

B+Tree Overview

Design Choices

Optimizations

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:

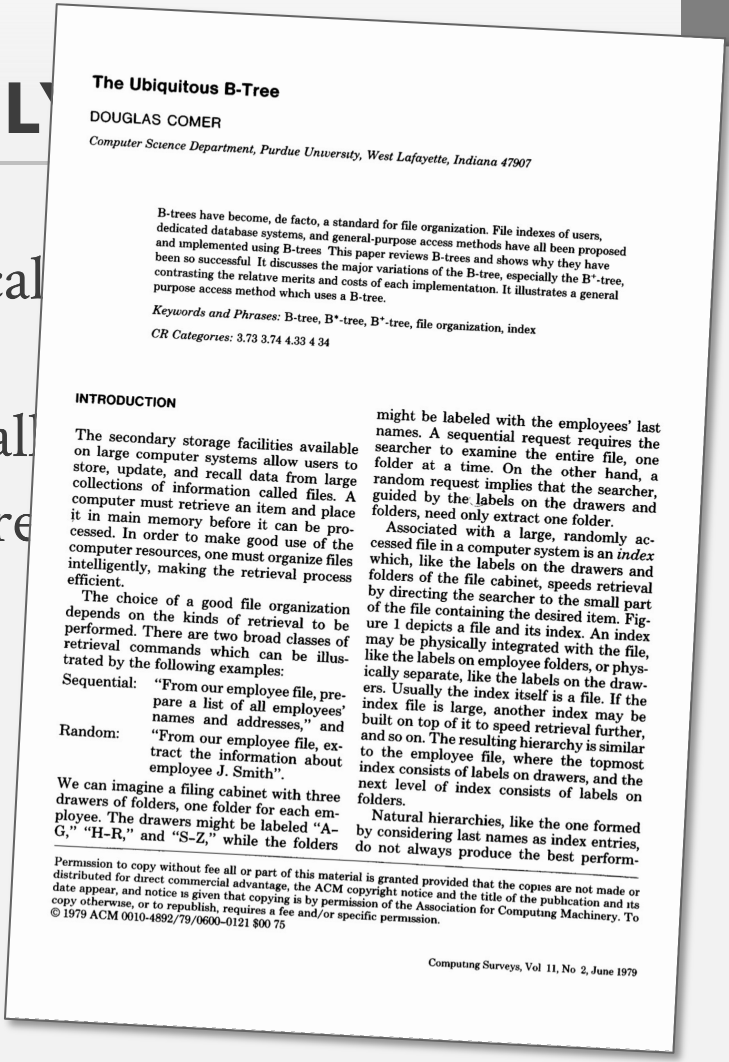
→ **B-Tree** (1971)
→ **B+Tree** (1973)
→ **B*Tree** (1977?)
→ **B^{link}-Tree** (1981)
→ **Bε-Tree** (2003)
→ **Bw-Tree** (2013)

B-TREE FAMILY

There is a specific data structure cal

People also use the term to generall
class of balanced tree data structure

- → **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-

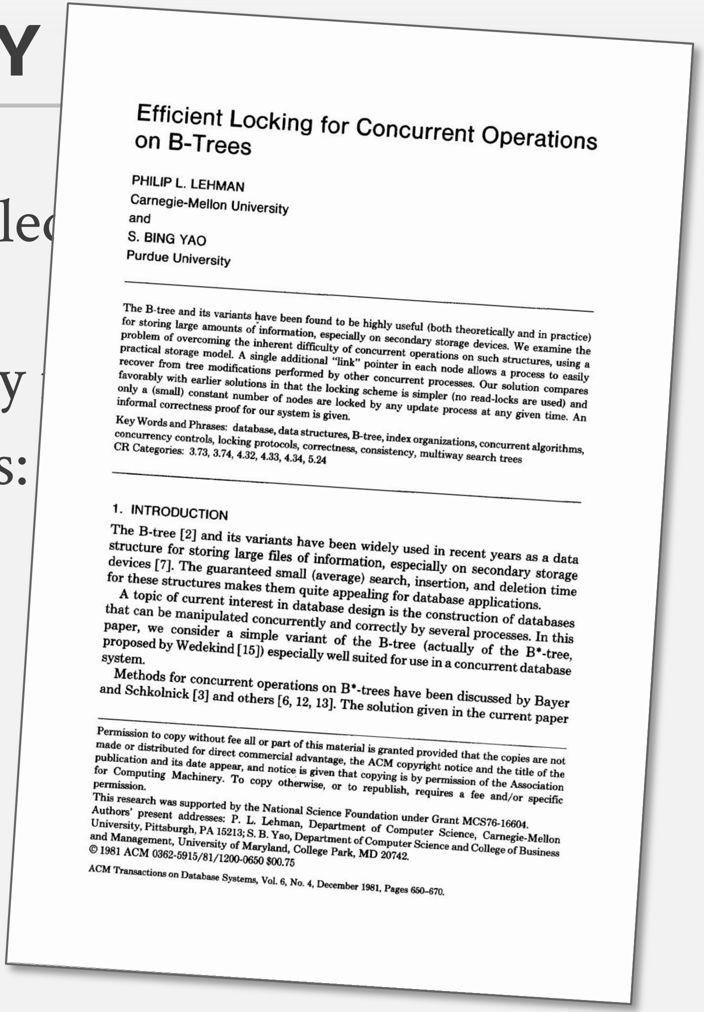
Computing Surveys, Vol 11, No 2, June 1979

B-TREE FAMILY

There is a specific data structure called

People also use the term to generally 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

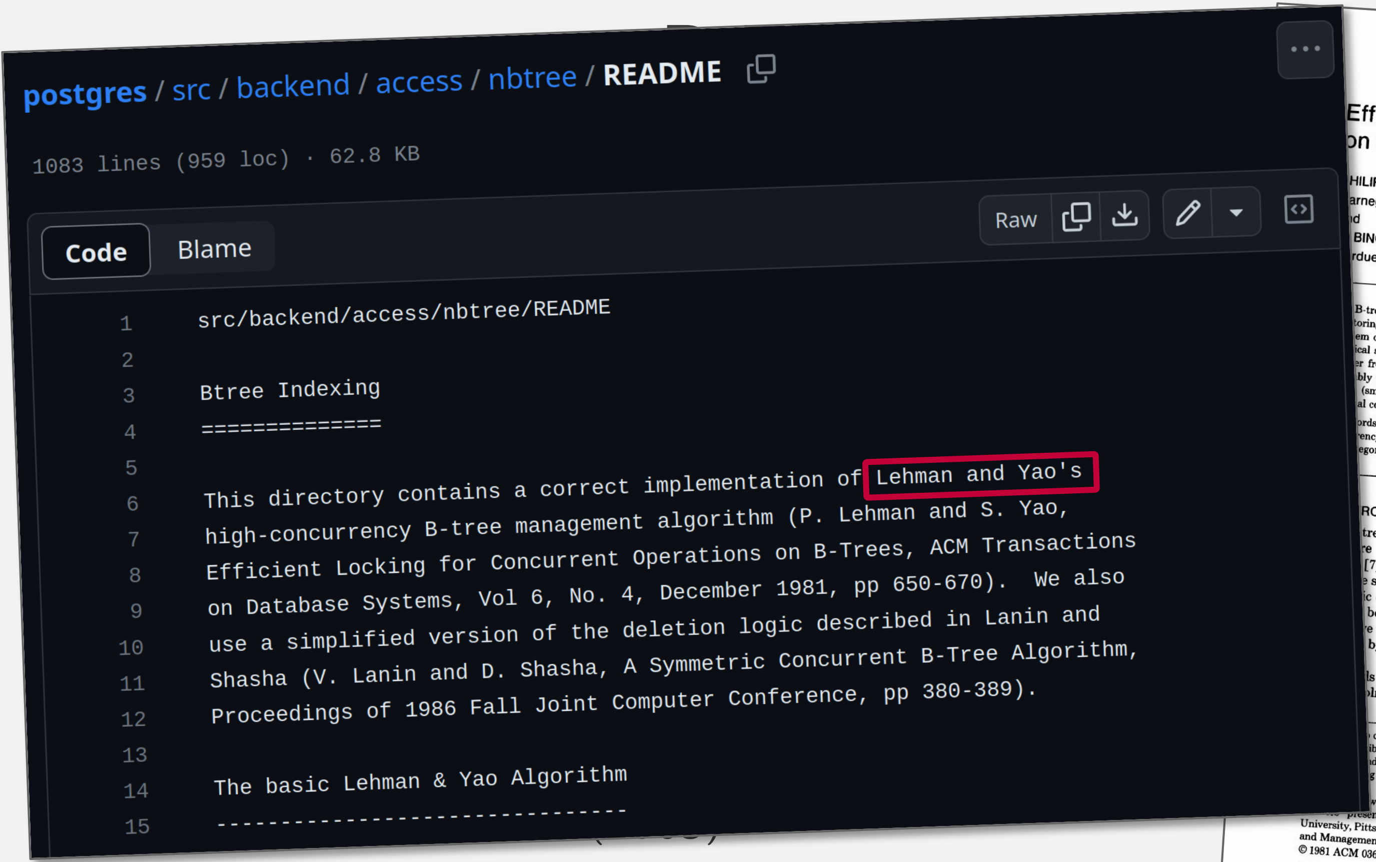
The B-tree [2] and its variants have been widely used in recent years as a data structure for storing large files of information, especially on secondary storage devices [7]. The guaranteed small (average) search, insertion, and deletion time for these structures makes them quite appealing for database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


postgres / src / backend / access / nbtree / README

1083 lines (959 loc) · 62.8 KB

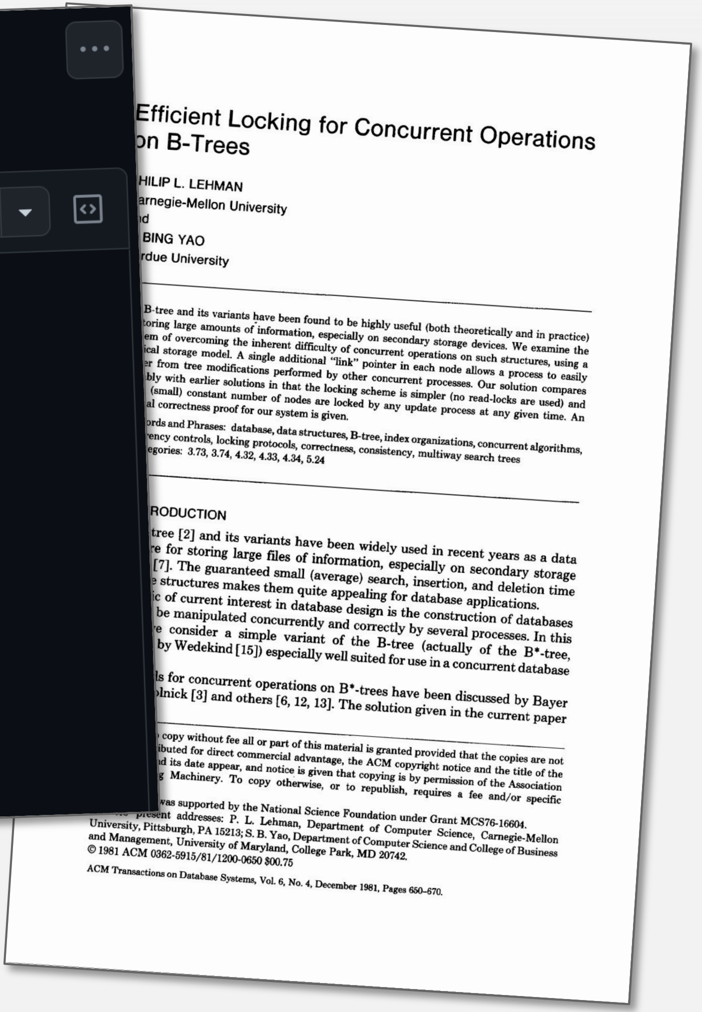
Code Blame Raw

```
src/backend/access/nbtree/README

Btree Indexing
==============

This directory contains a correct implementation of Lehman and Yao's
high-concurrency B-tree management algorithm (P. Lehman and S. Yao,
Efficient Locking for Concurrent Operations on B-Trees, ACM Transactions
on Database Systems, Vol 6, No. 4, December 1981, pp 650-670).  We also
use a simplified version of the deletion logic described in Lanin and
Shasha (V. Lanin and D. Shasha, A Symmetric Concurrent B-Tree Algorithm,
Proceedings of 1986 Fall Joint Computer Conference, pp 380-389).

The basic Lehman & Yao Algorithm
--------------------------------
```


Efficient Locking for Concurrent Operations
on B-Trees

HILIP L. LEHMAN
arnegie-Mellon University
nd
BING YAO
rdue University

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

ords and Phrases: database, data structures, B-tree, index organizations, concurrent algorithms, ency controls, locking protocols, correctness, consistency, multiway search trees
egories: 3.73, 3.74, 4.32, 4.33, 4.34, 5.24

RODUCTION

tree [2] and its variants have been widely used in recent years as a data re for storing large files of information, especially on secondary storage [7]. The guaranteed small (average) search, insertion, and deletion time e structures makes them quite appealing for database applications. ic of current interest in database design is the construction of databases be manipulated concurrently and correctly by several processes. In this e consider a simple variant of the B-tree (actually of the B*-tree, by Wedekind [15]) especially well suited for use in a concurrent database

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

copy without fee all or part of this material is granted provided that the copies are not ibuted for direct commercial advantage, the ACM copyright notice and the title of the d its date appear, and notice is given that copying is by permission of the Association g Machinery. To copy otherwise, or to republish, requires a fee and/or specific

was supported by the National Science Foundation under Grant MCS76-16604.
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.
© 1981 ACM 0362-5915/81/1200-0650 $00.75
ACM Transactions on Database Systems, Vol. 6, No. 4, December 1981, Pages 650–670.

→ **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 $\mathbf{O(\log_f n)}$.

→ 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.

→ **f** is the fanout of the tree.

B+TREE PROPERTIES

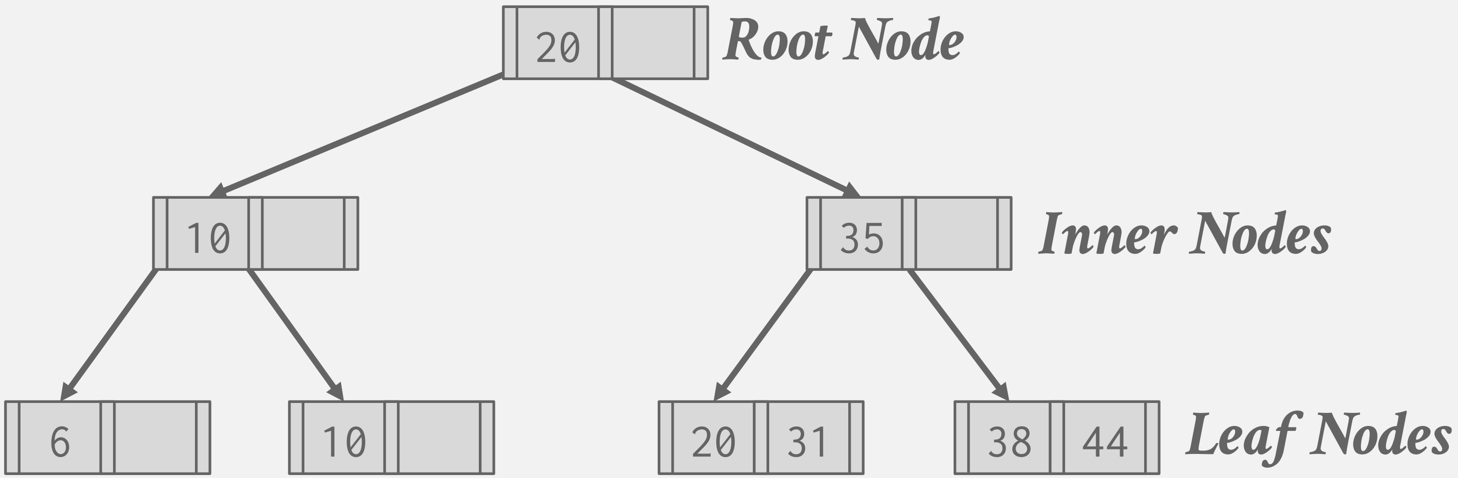
A B+Tree is an ***M***-way search tree with the following properties:

→ It is perfectly balanced (i.e., every leaf node is at the same depth in the tree)

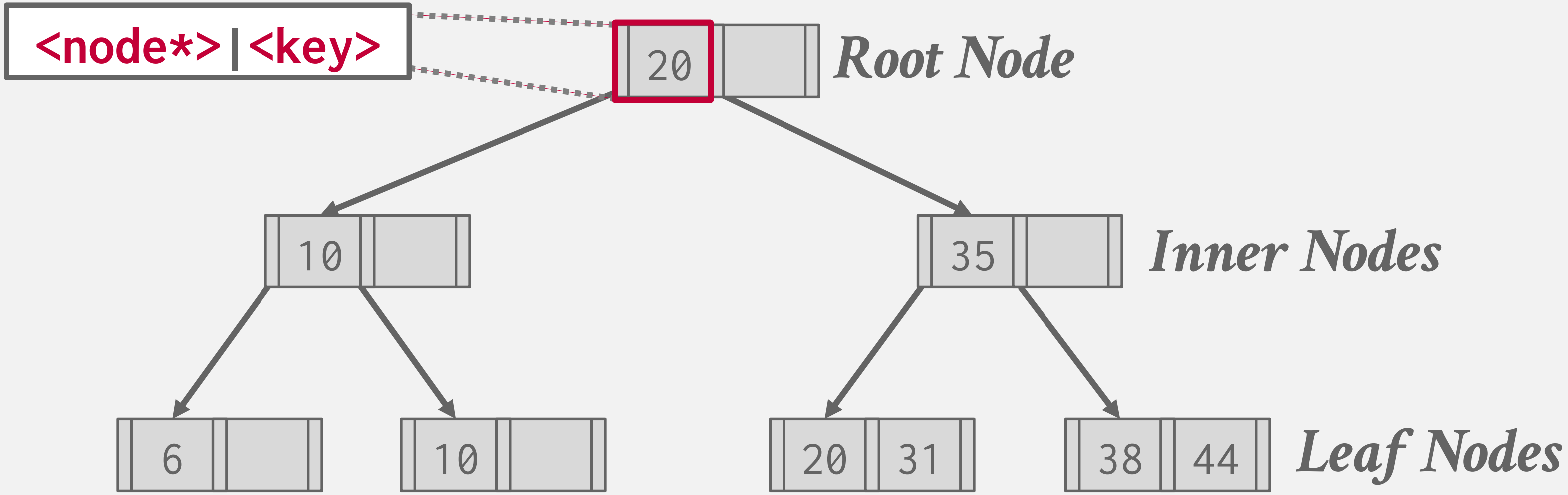
→ Every node other than the root is at least half-full
M/2-1 ≤ #keys ≤ M-1

→ Every inner node with **k** keys has **k+1** non-null children

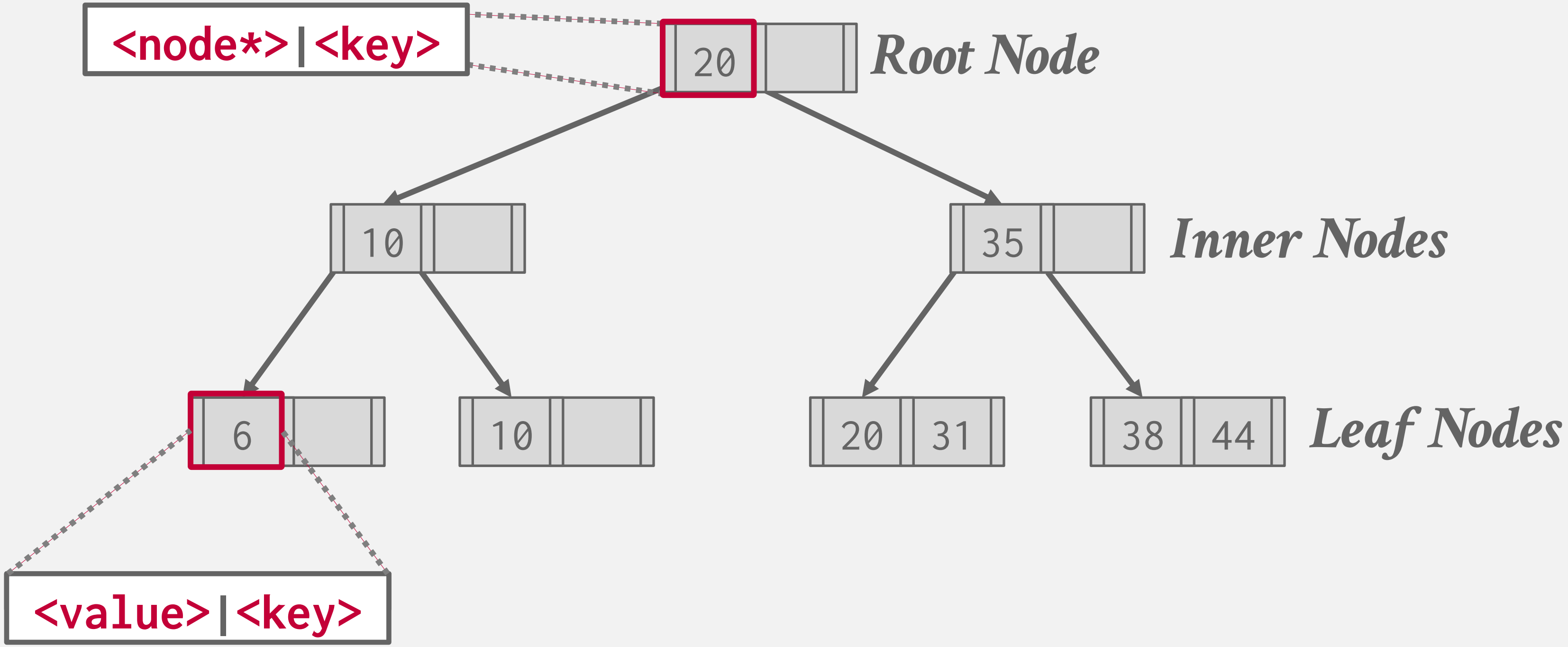
B+TREE EXAMPLE



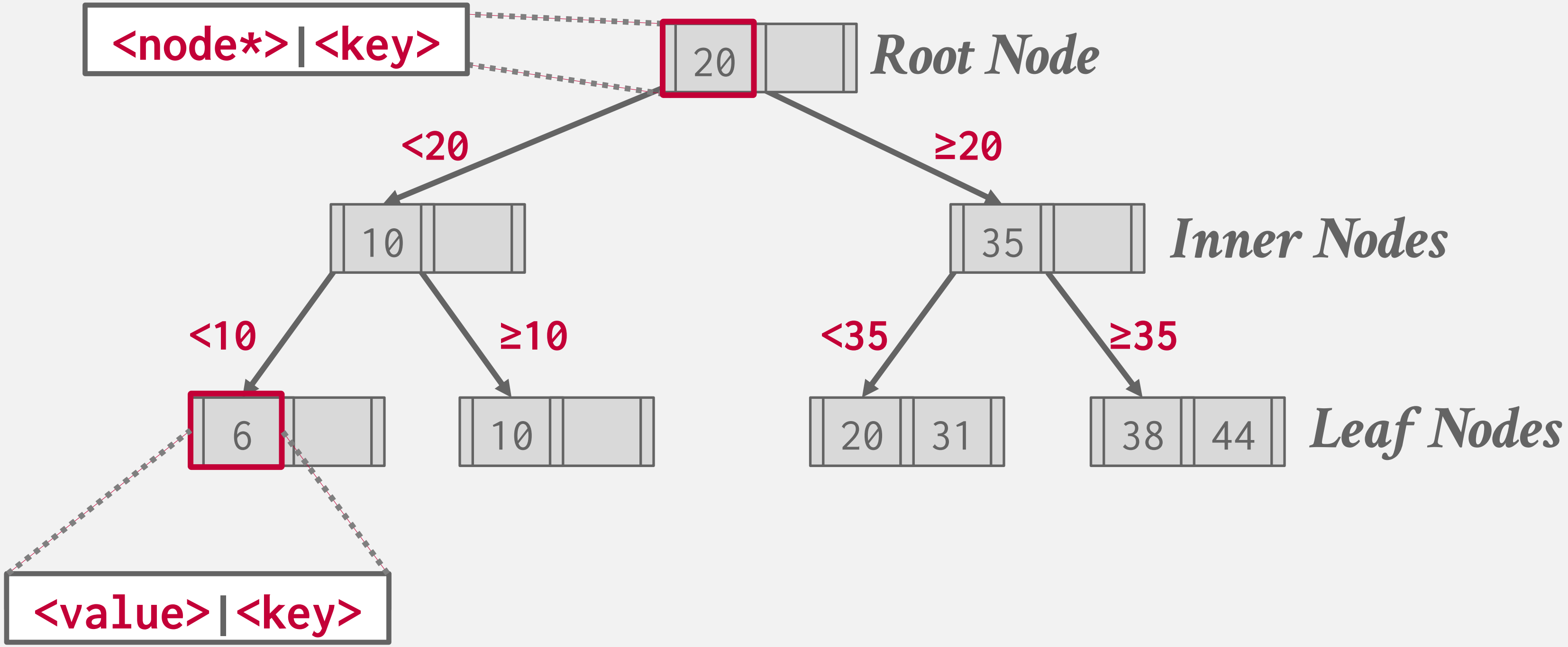
B+TREE EXAMPLE



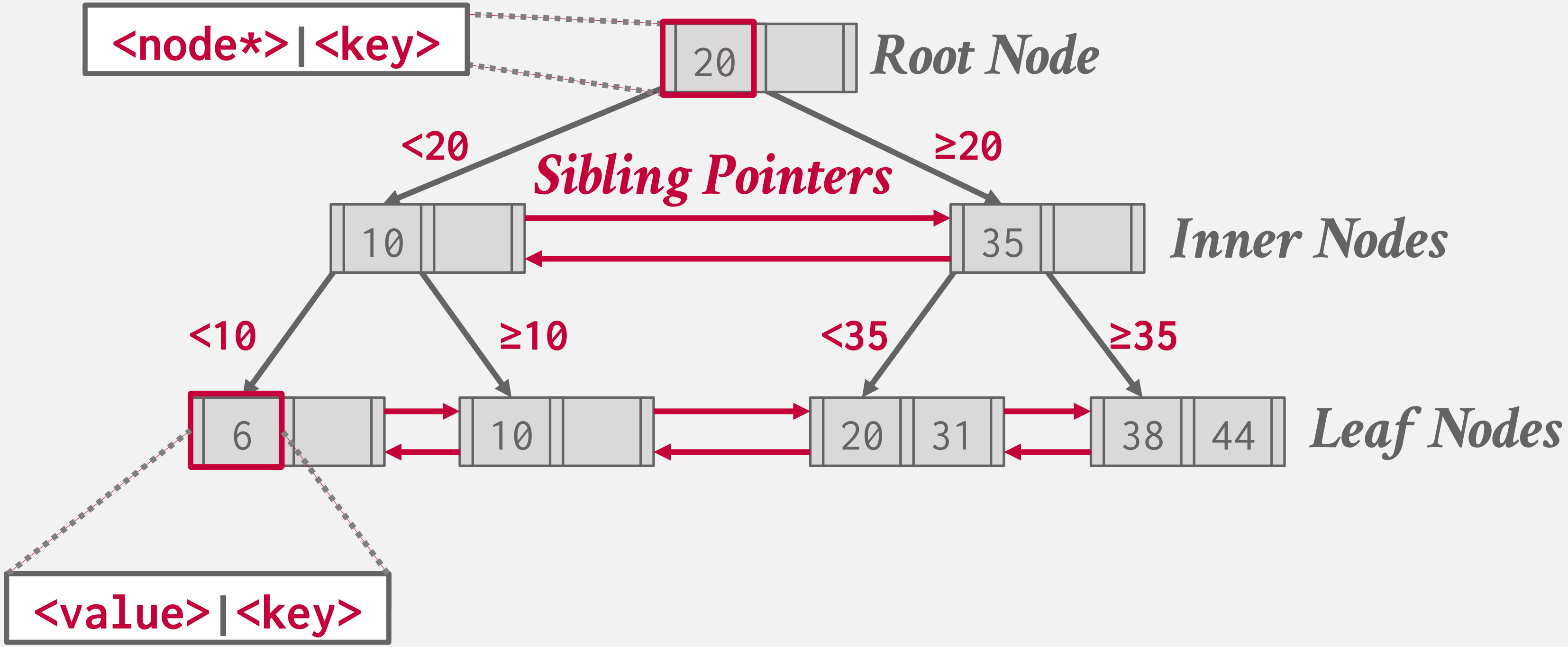
B+TREE EXAMPLE



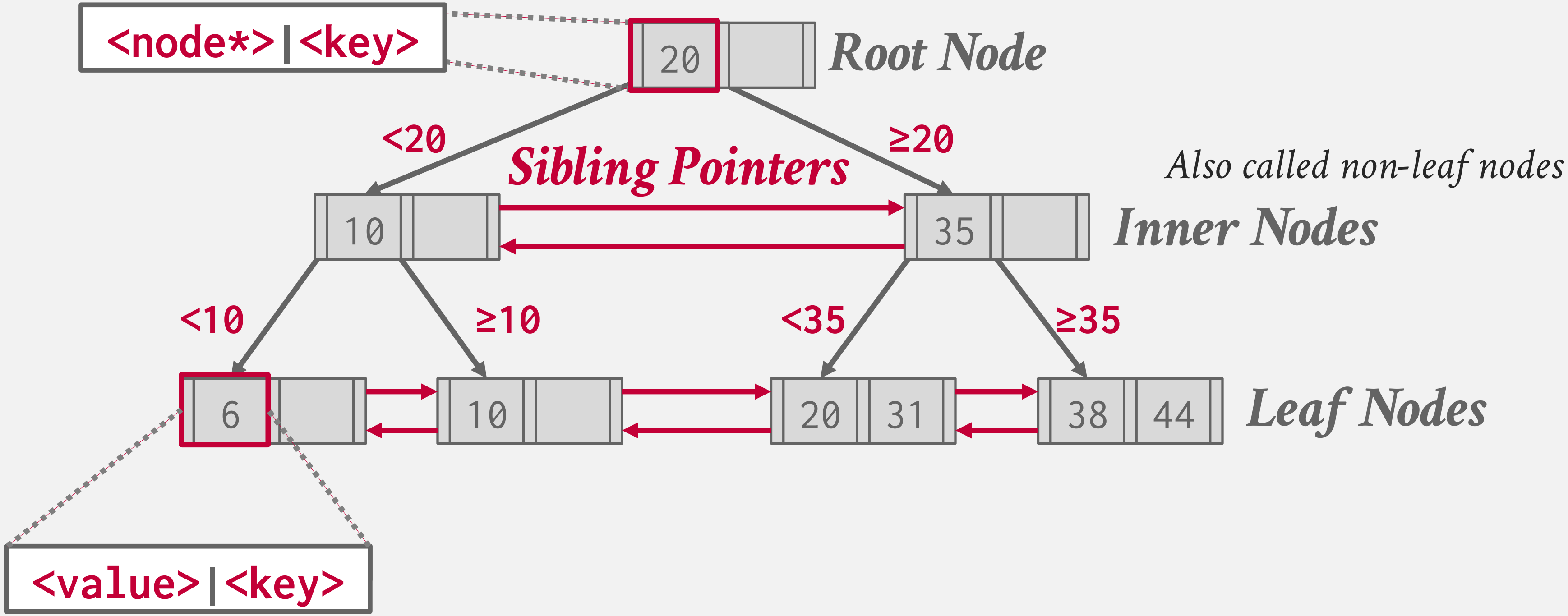
B+TREE EXAMPLE



B+TREE EXAMPLE



B+TREE EXAMPLE



NODES

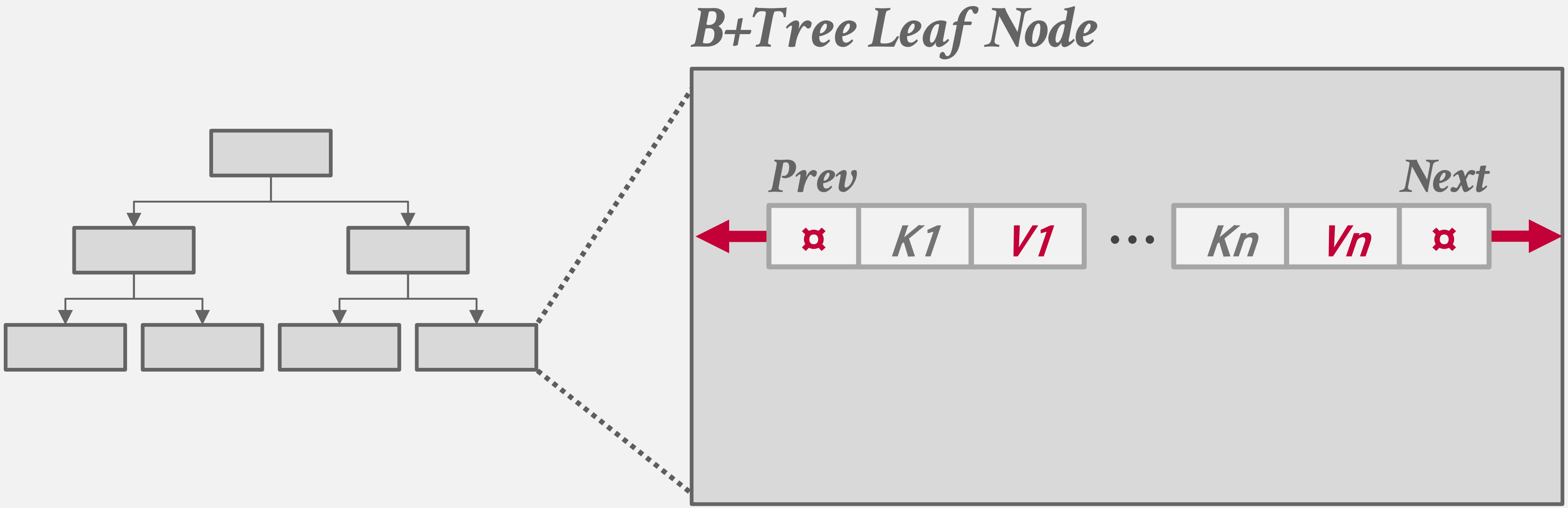
Every B+Tree node is comprised of an array of key/value pairs.

- → 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.**

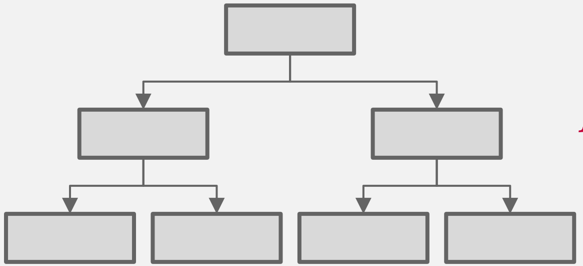
The arrays are (usually) kept in sorted key order.

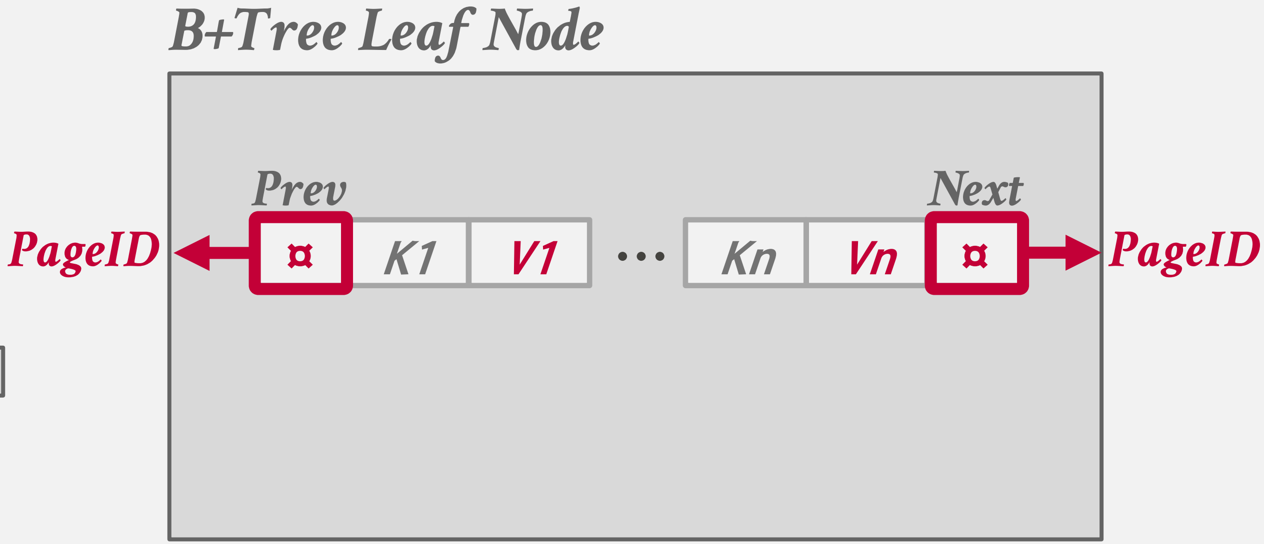
Store all **NULL** keys at either first or last leaf nodes.

B+TREE LEAF NODES

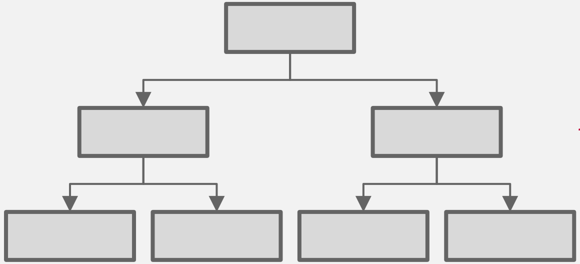


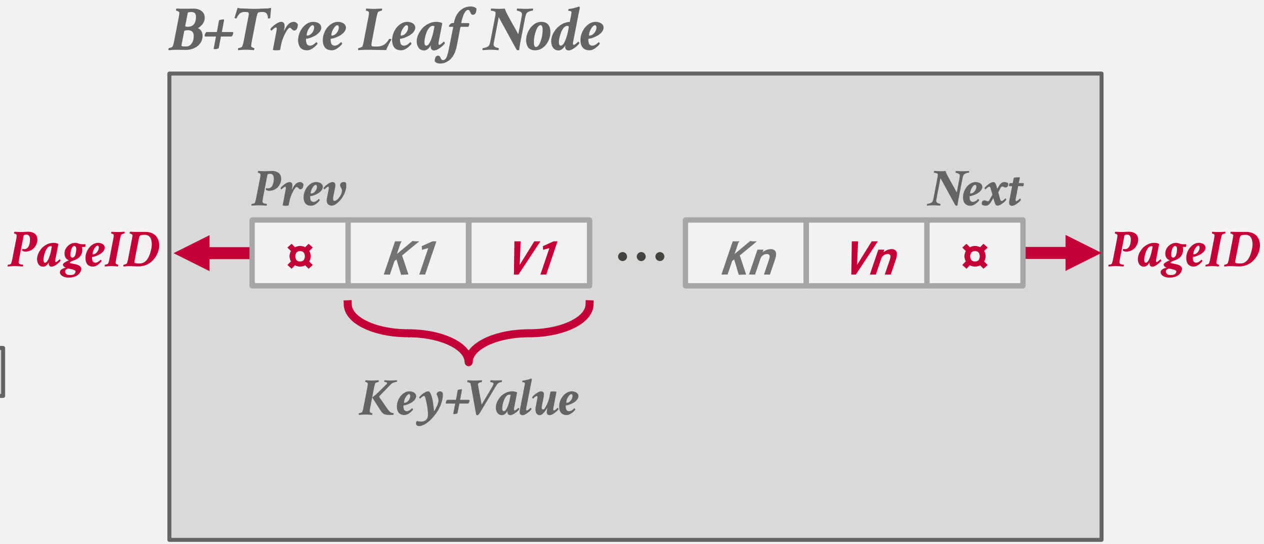
B+TREE LEAF NODES



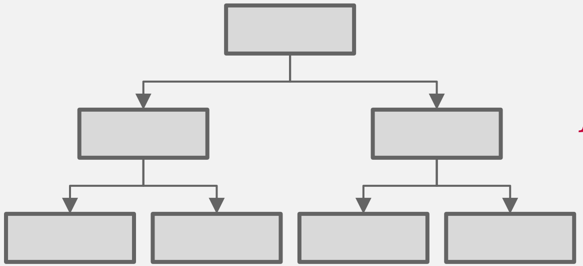


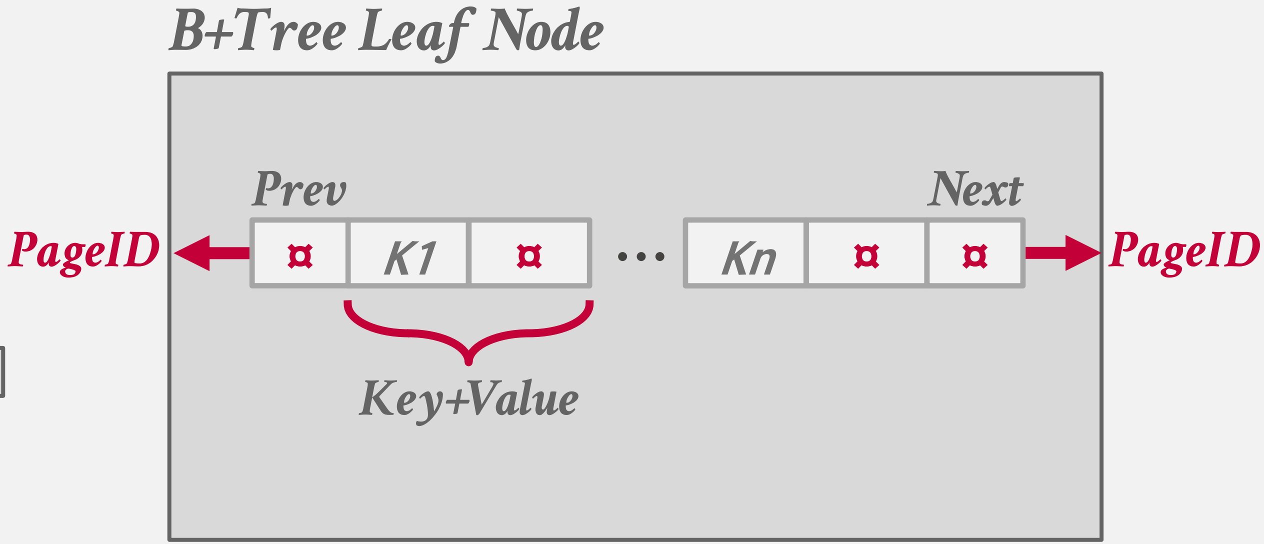
B+TREE LEAF NODES



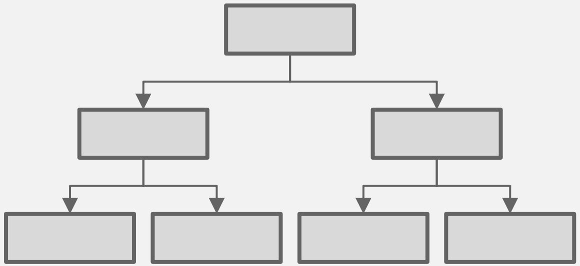


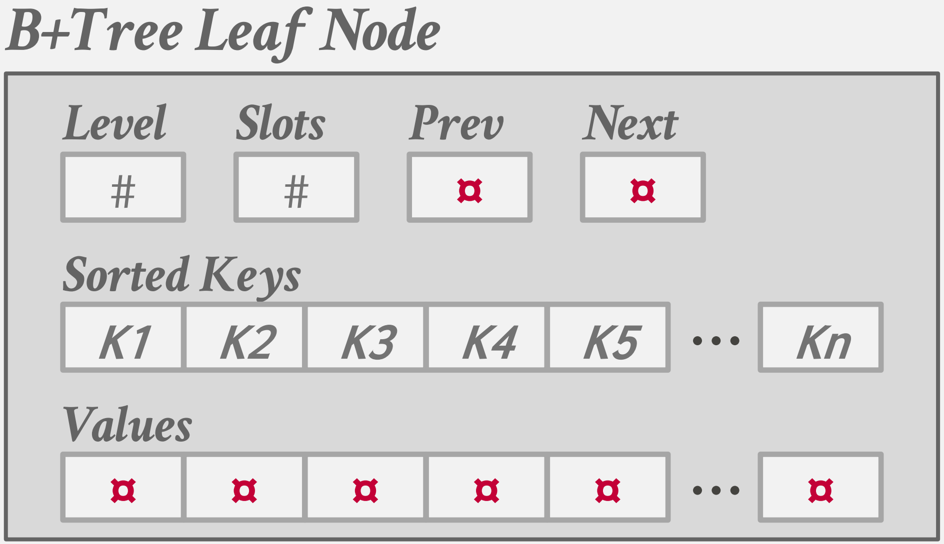
B+TREE LEAF NODES



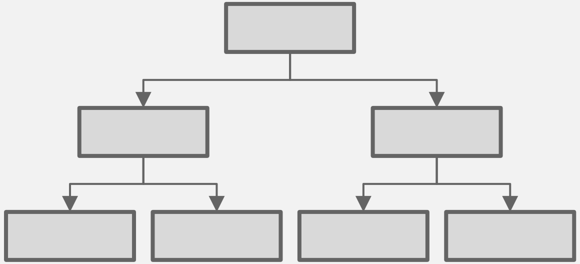


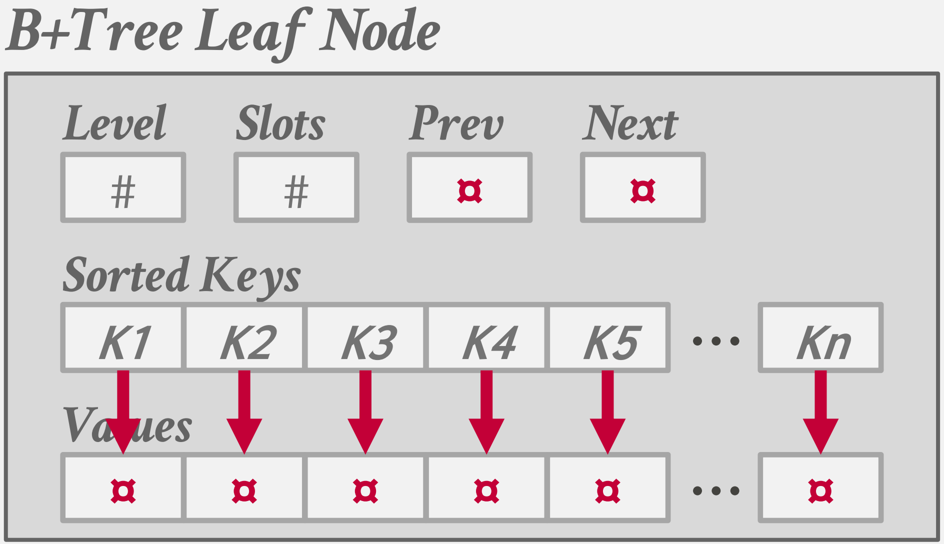
B+TREE LEAF NODES





B+TREE LEAF NODES





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**

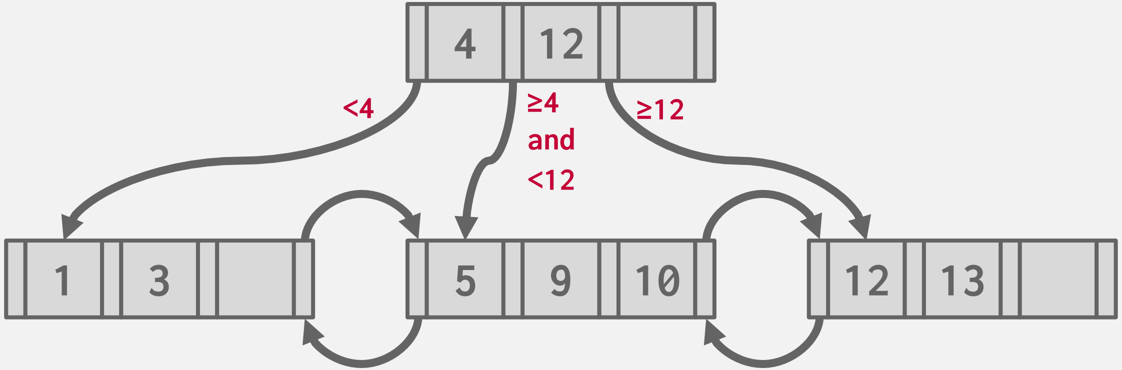
→ Redistribute entries evenly, copy up middle key.

→ Insert index entry pointing to **L2** into parent of **L**.

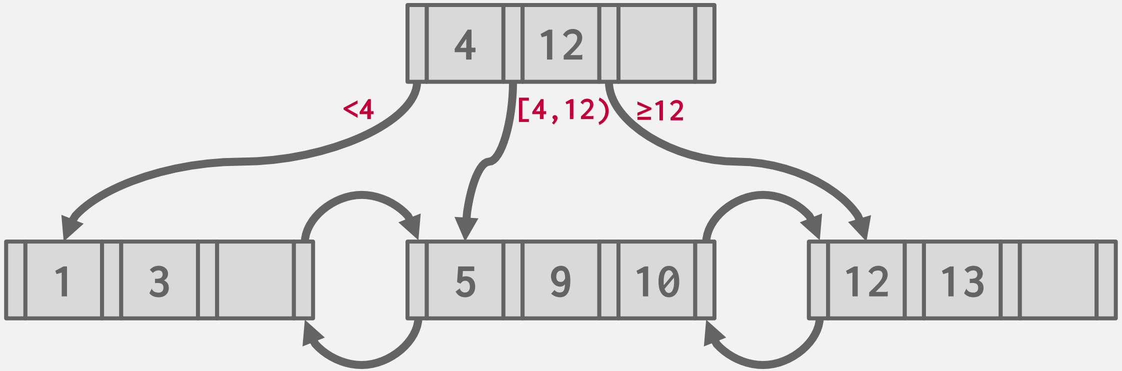
To split inner node, redistribute entries evenly, but push up middle key.

Source: Chris Re

B+TREE – INSERT

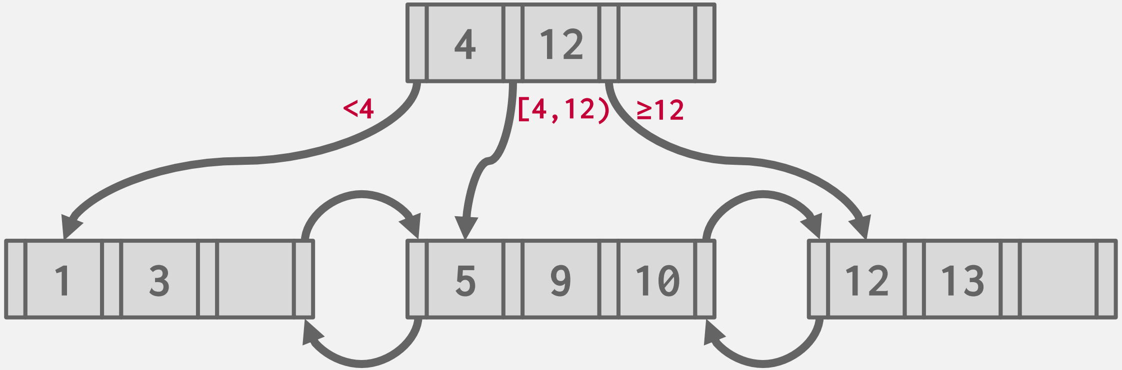


B+TREE – INSERT



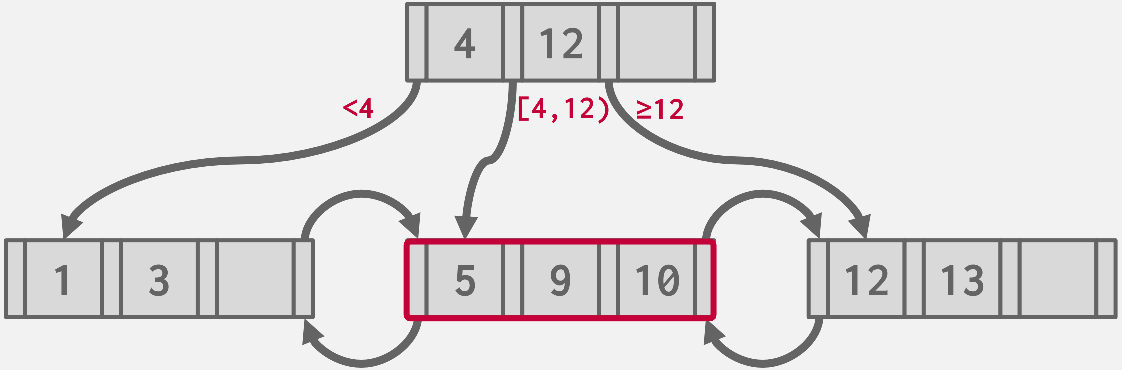
B+TREE – INSERT

Insert 6



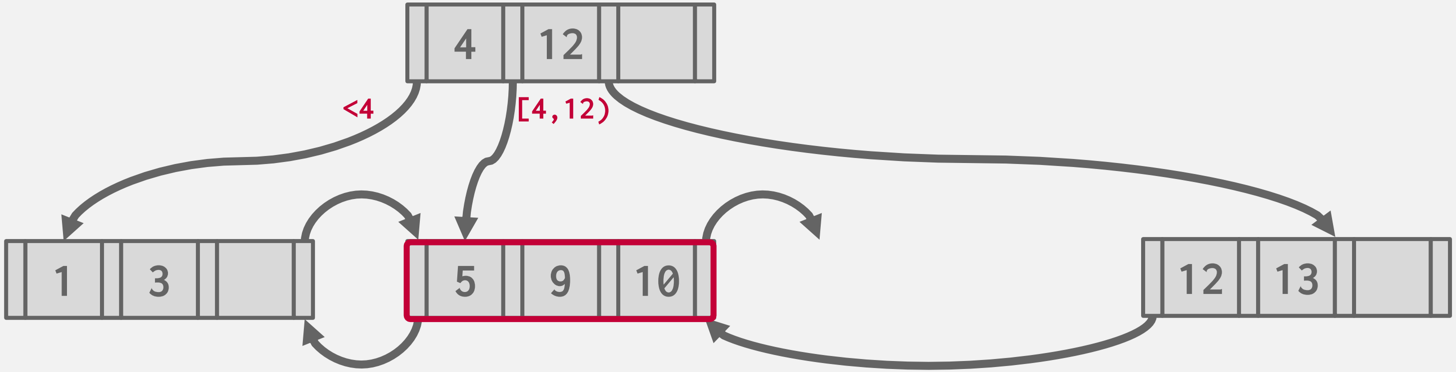
B+TREE – INSERT

Insert 6



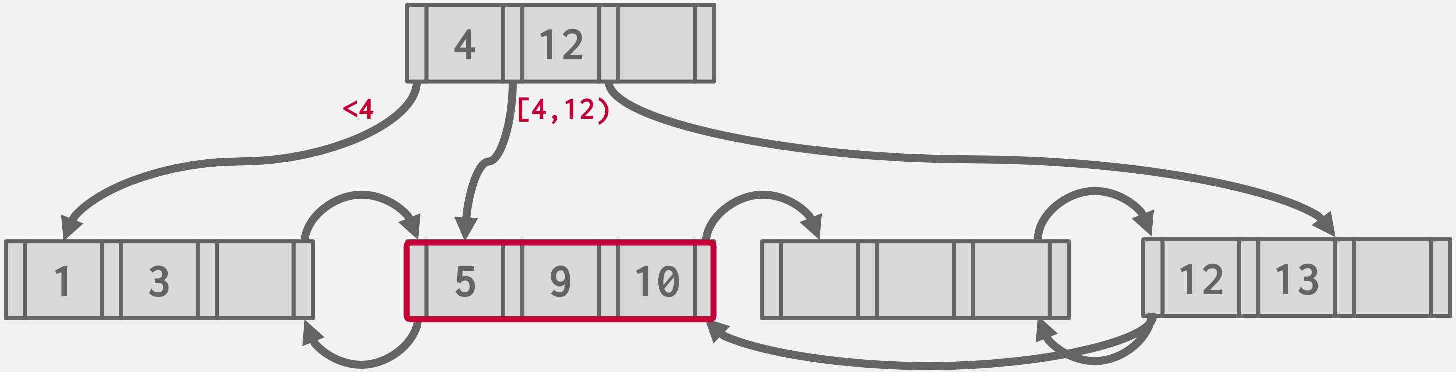
B+TREE – INSERT

Insert 6



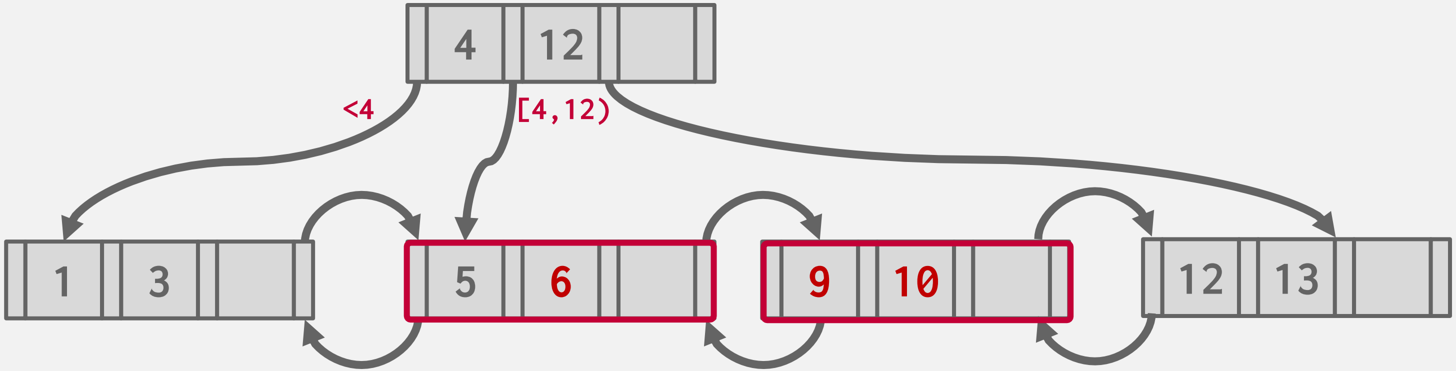
B+TREE – INSERT

Insert 6



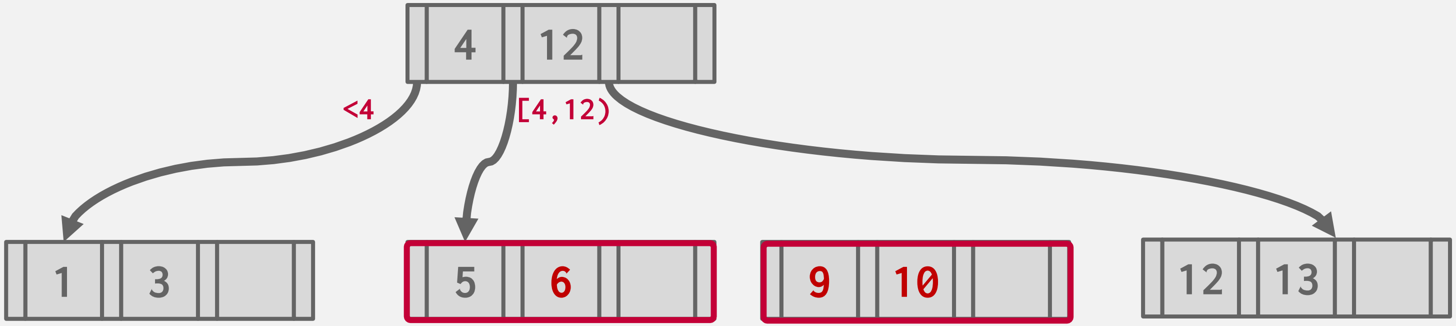
B+TREE – INSERT

Insert 6



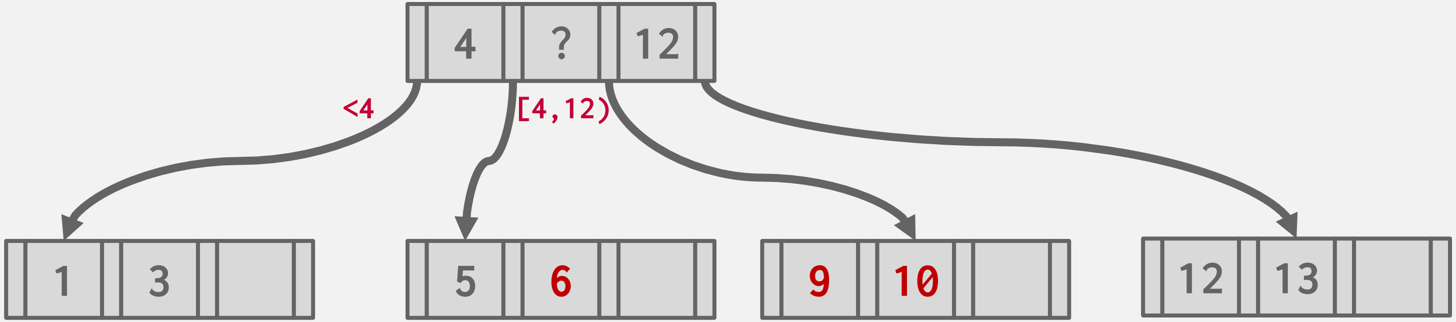
B+TREE – INSERT

Insert 6



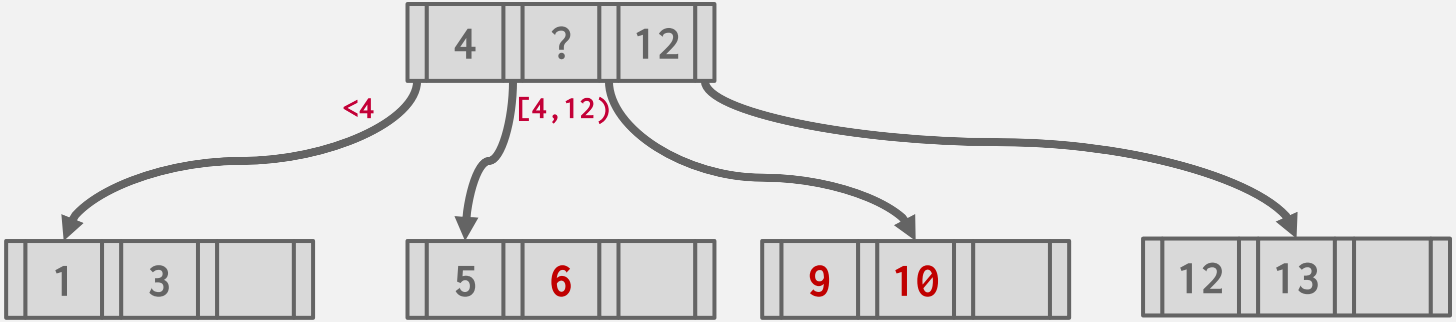
B+TREE – INSERT

Insert 6



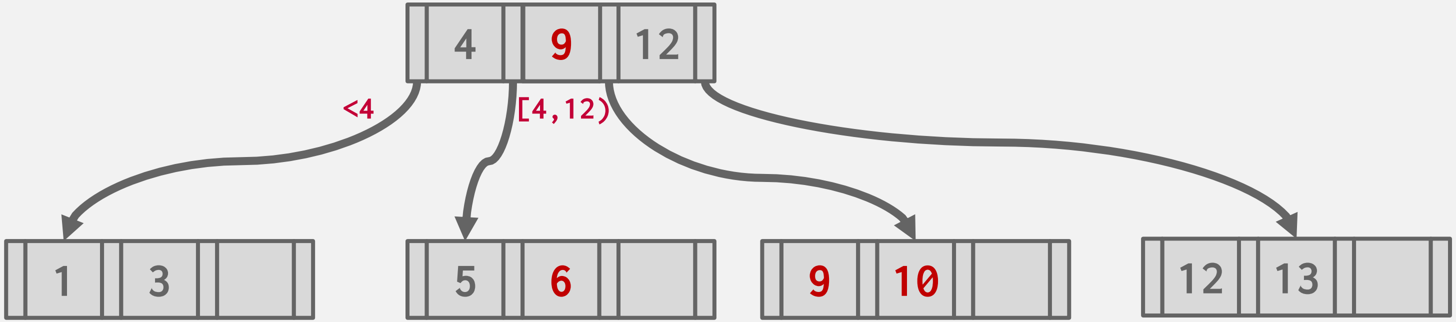
B+TREE – INSERT

Insert 6



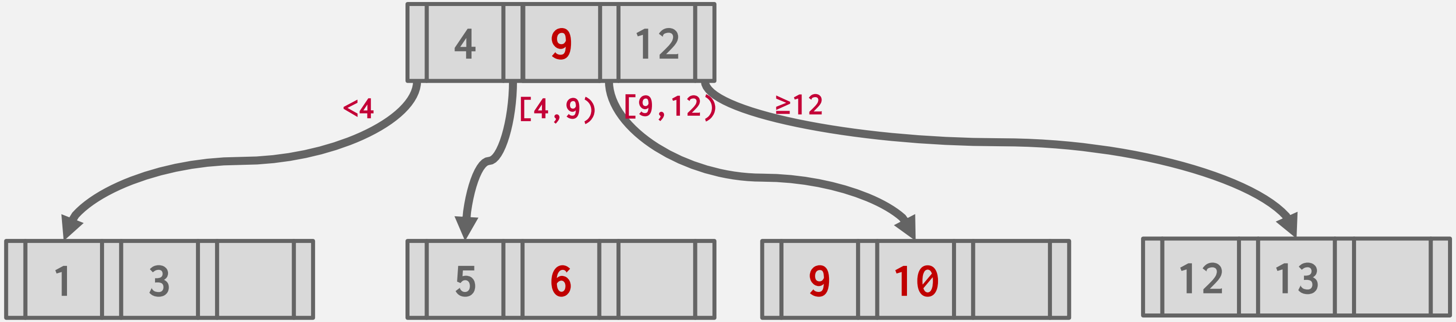
B+TREE – INSERT

Insert 6



B+TREE – INSERT

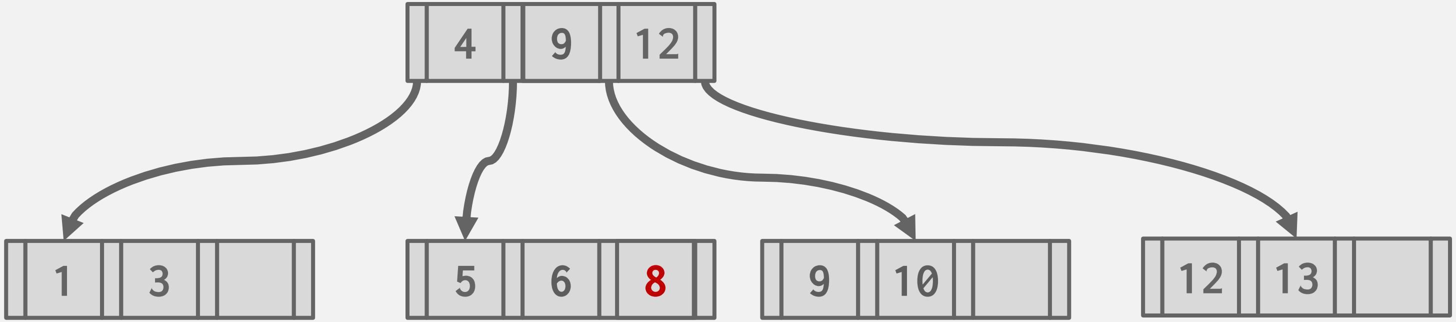
Insert 6



B+TREE – INSERT

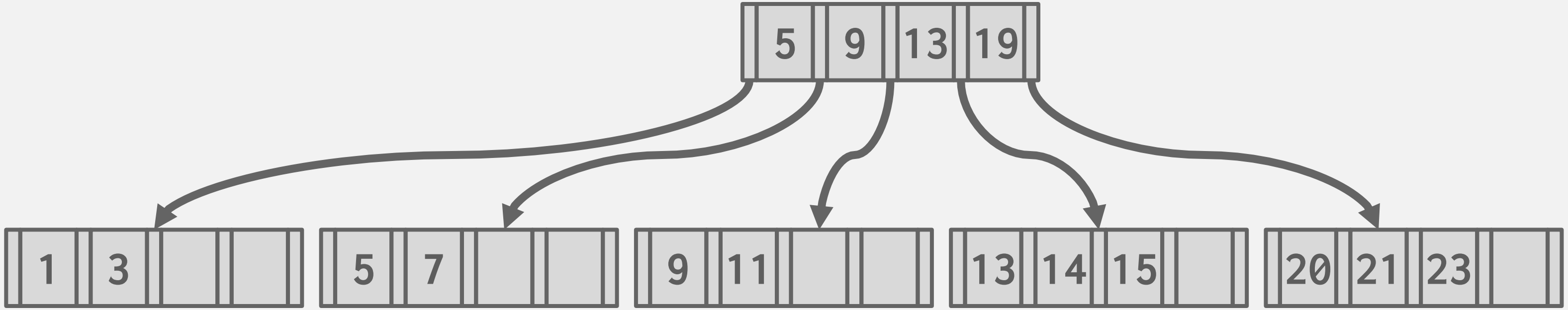
Insert 6

Insert 8



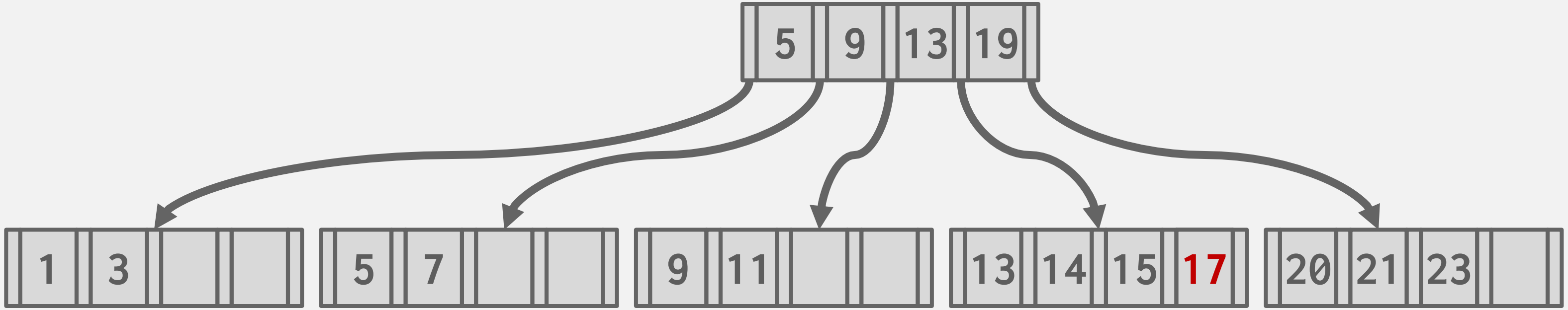
INSERT THE KEY 17

Note: new example/tree.

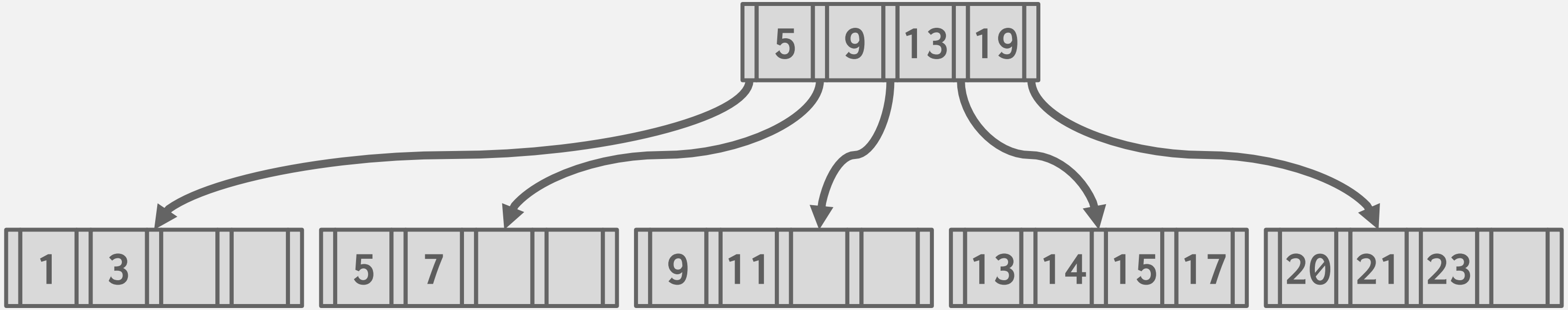


INSERT THE KEY 17

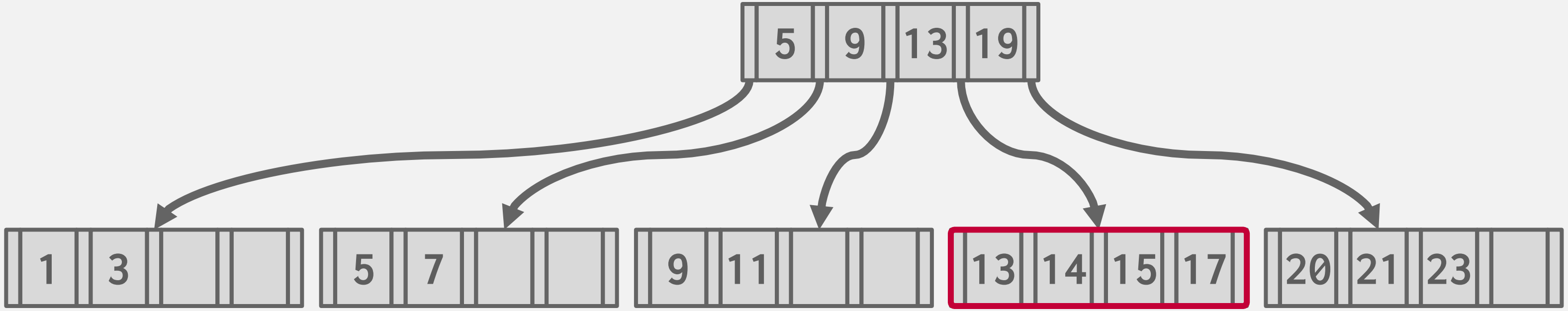
Note: new example/tree.



NEXT, INSERT THE KEY 16

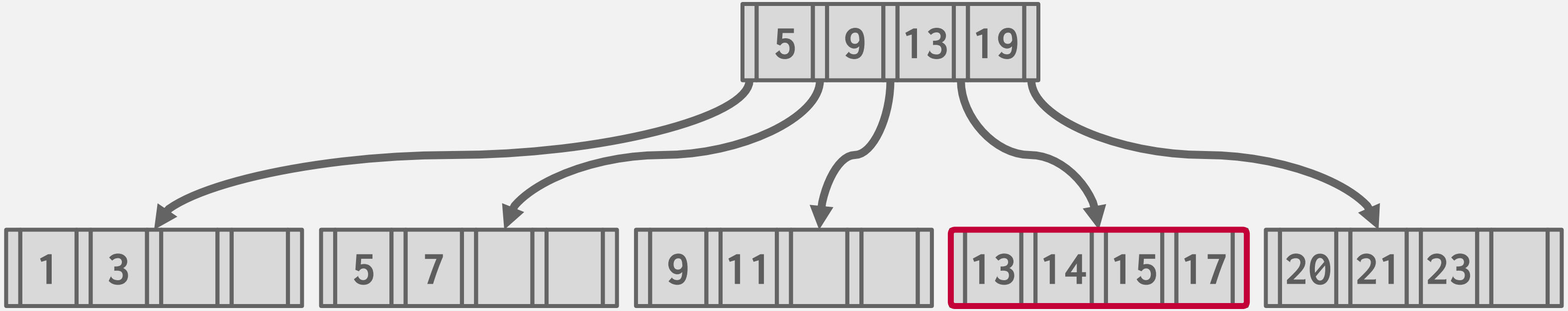


NEXT, INSERT THE KEY 16



No space in the node where the new key "belongs".

NEXT, INSERT THE KEY 16

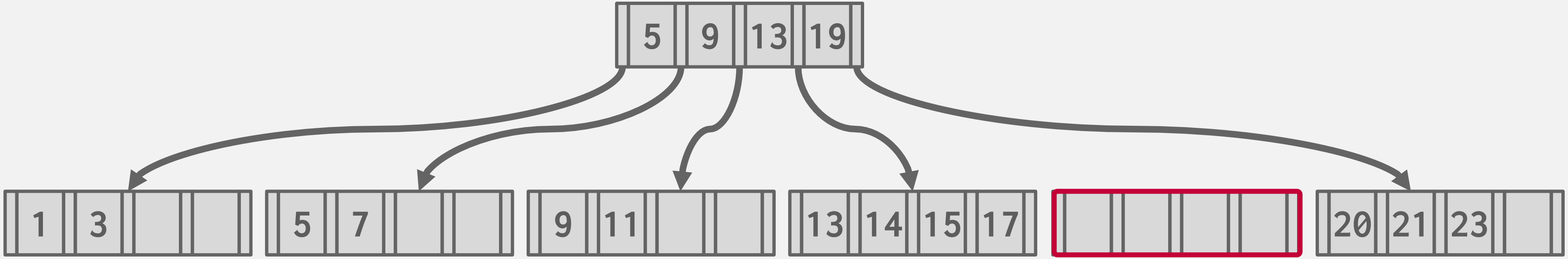


Split the node!

Copy the middle key.

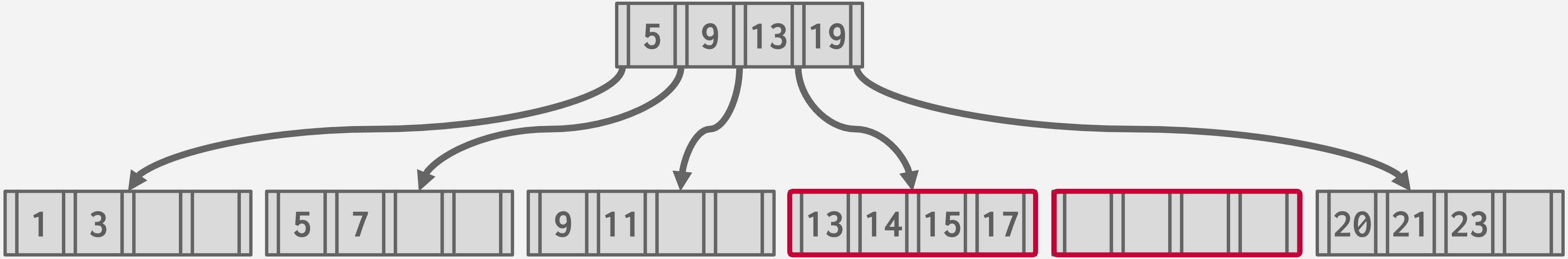
Push the key up.

NEXT, INSERT THE KEY 16



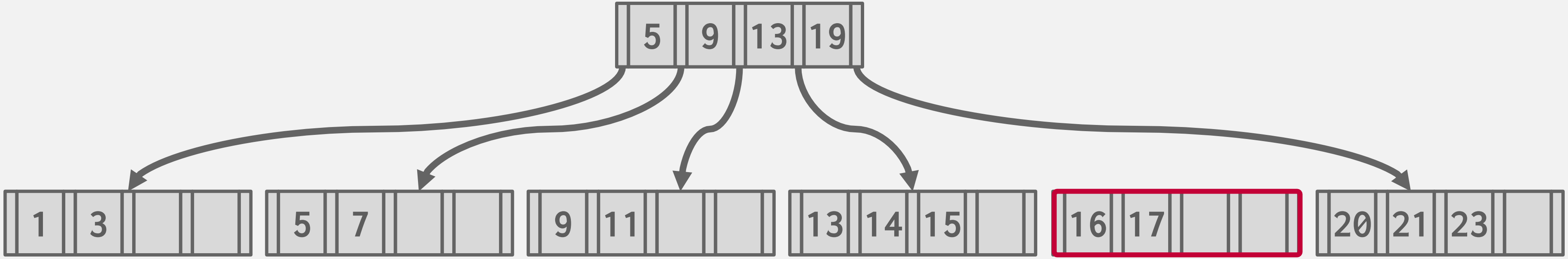
New node.
Shuffle keys from the node that triggered the split.

NEXT, INSERT THE KEY 16



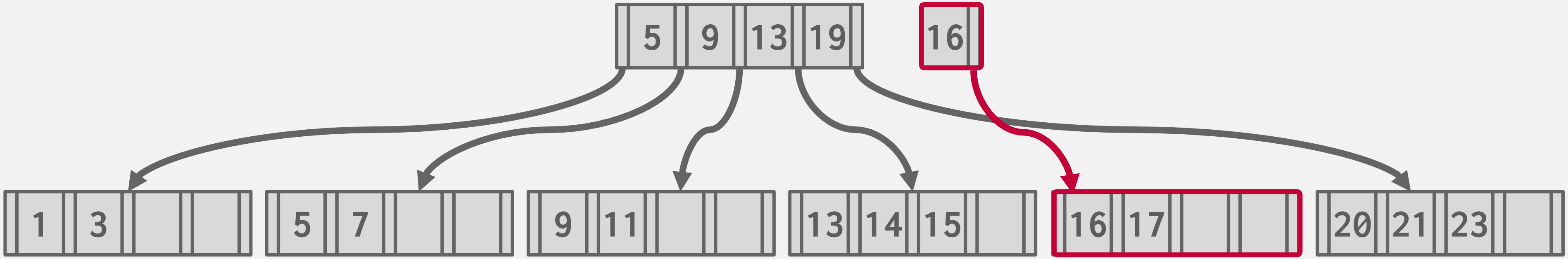
New node.
Shuffle keys from the node that triggered the split.

NEXT, INSERT THE KEY 16



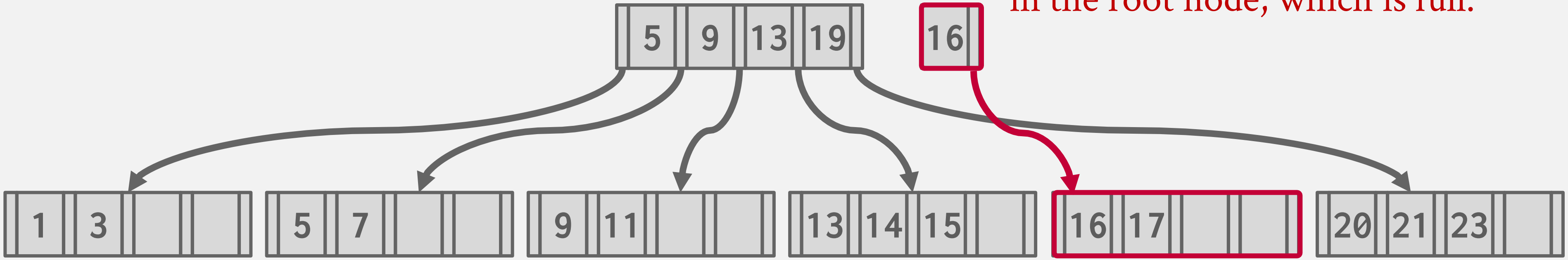
But, this is an "orphan" node. No parent node points to it.

NEXT, INSERT THE KEY 16



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

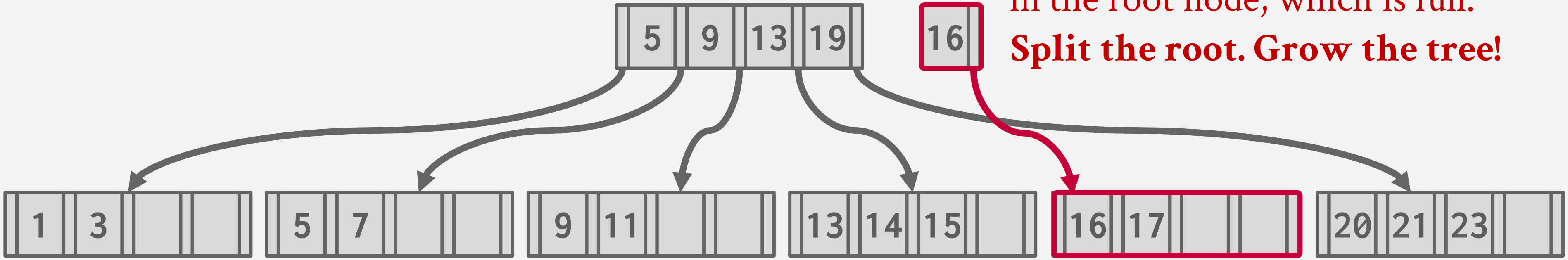
NEXT, INSERT THE KEY 16



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.

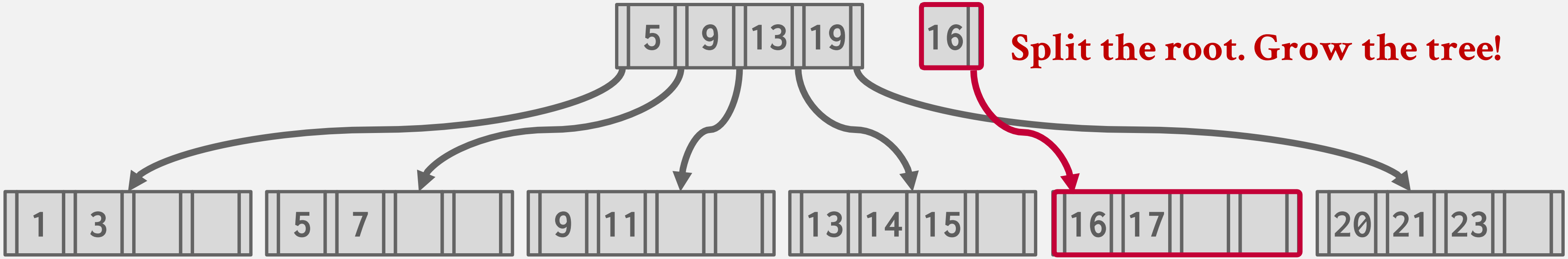
NEXT, INSERT THE KEY 16



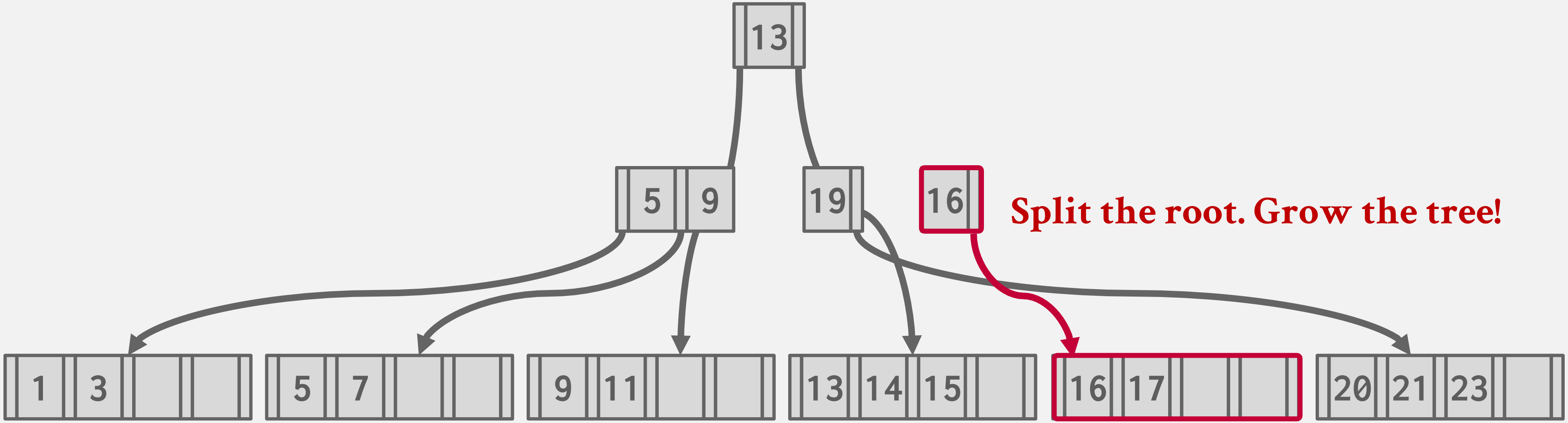
Want to create a key, pointer pair like this. But can't insert it in the root node, which is full. **Split the root. Grow the tree!**

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

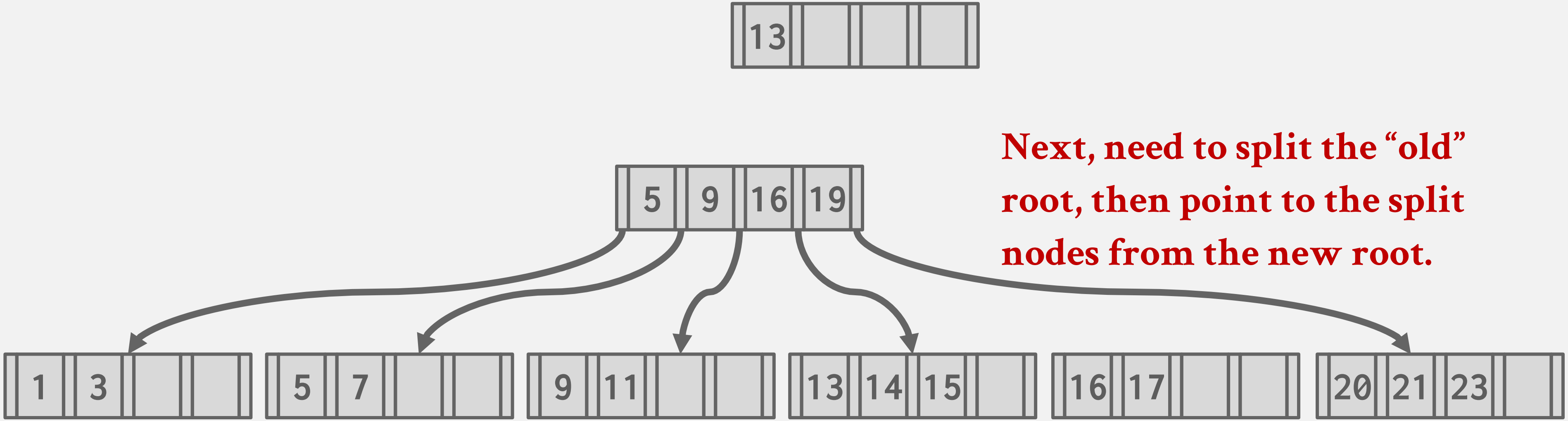
NEXT, INSERT THE KEY 16



NEXT, INSERT THE KEY 16

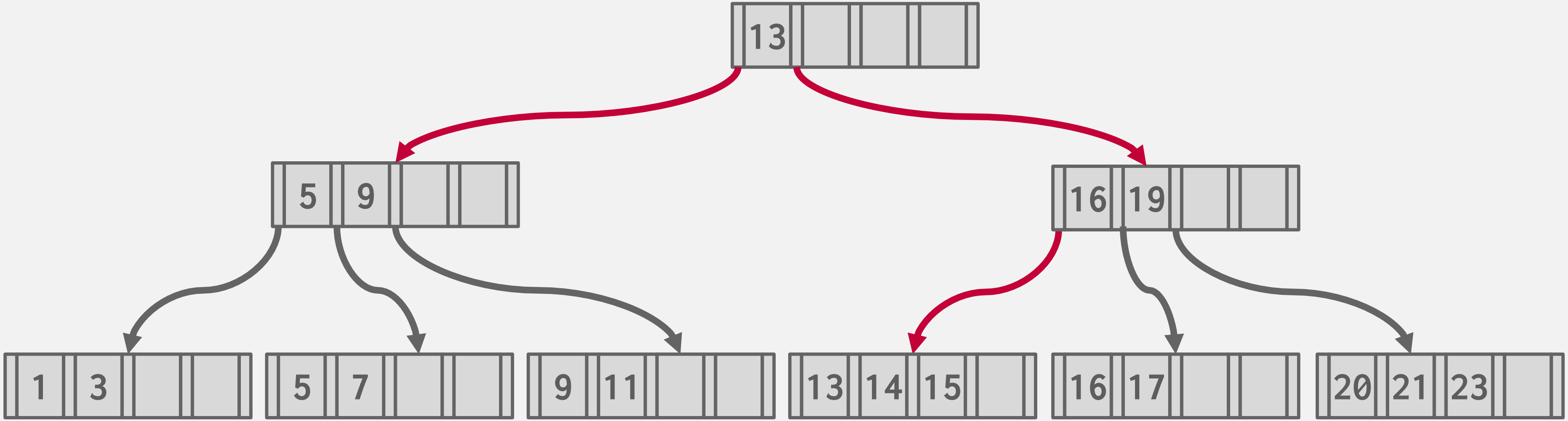


NEXT, INSERT THE KEY 16

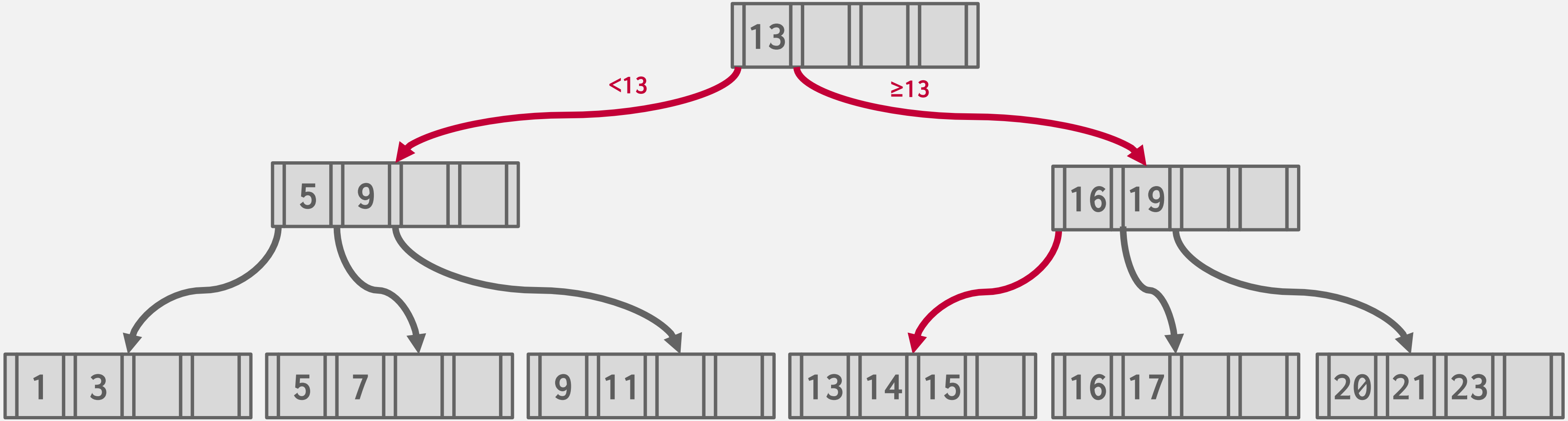


Next, need to split the "old" root, then point to the split nodes from the new root.

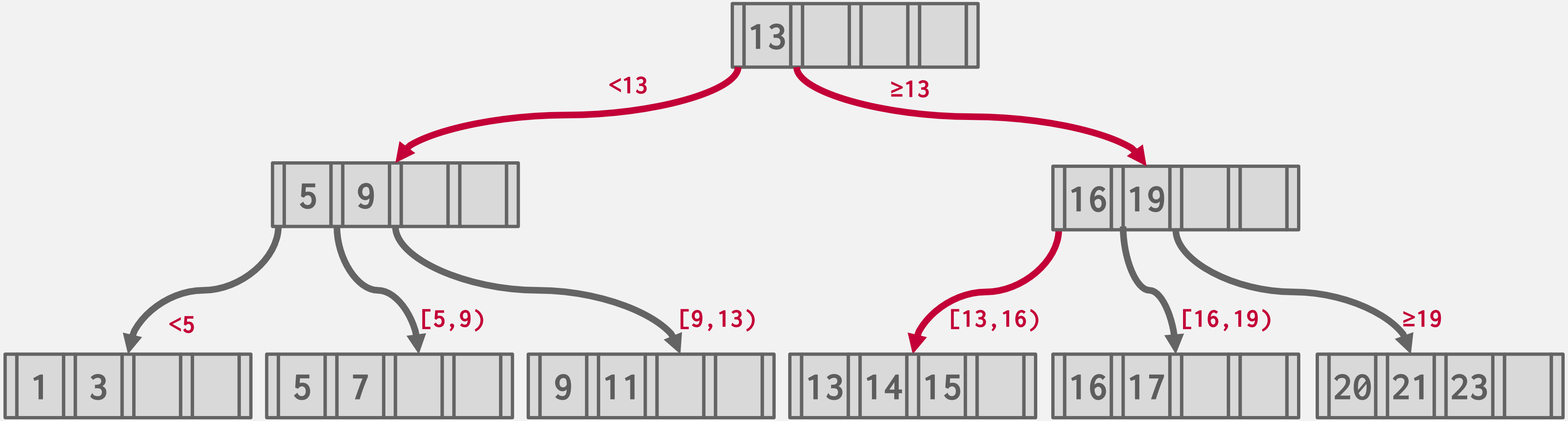
NEXT, INSERT THE KEY 16



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,

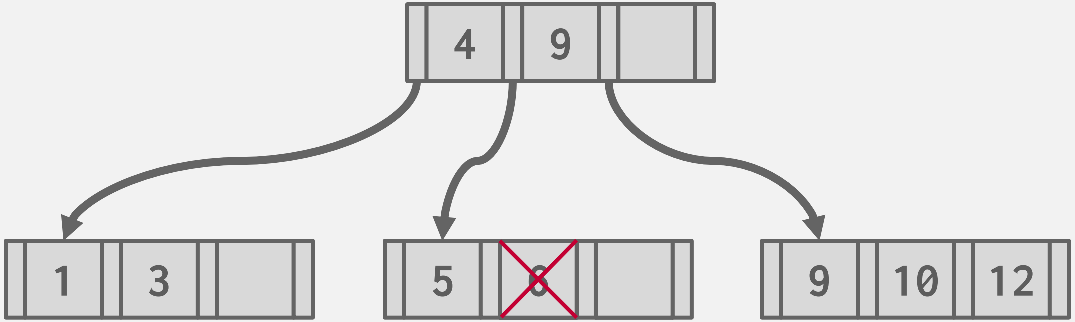
→ Try to re-distribute, borrowing from sibling (adjacent node with same parent as **L**).

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

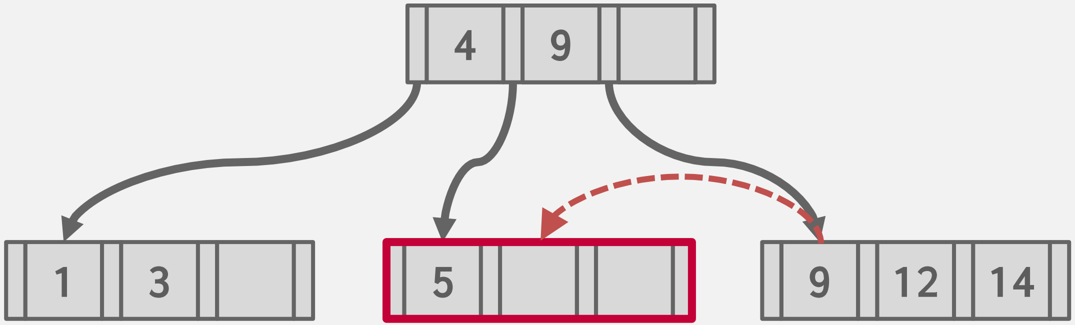
If merge occurred, must delete entry (pointing to **L** or sibling) from parent of **L**.

Source: Chris Re

DELETE THE KEY 6

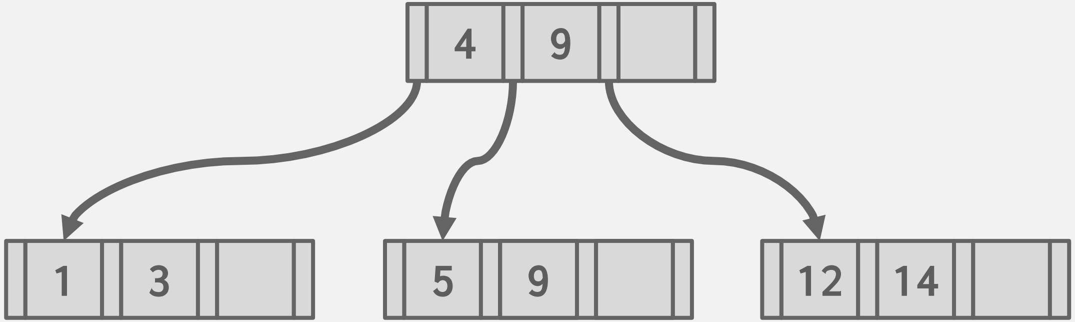


DELETE THE KEY 6



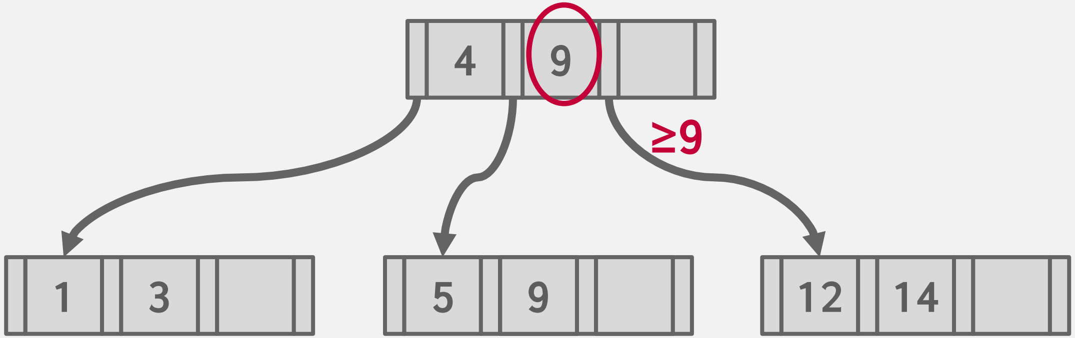
Borrow from a "rich" neighbor.

DELETE THE KEY 6



Borrow from a "rich" neighbor.
Could borrow from either neighbor.

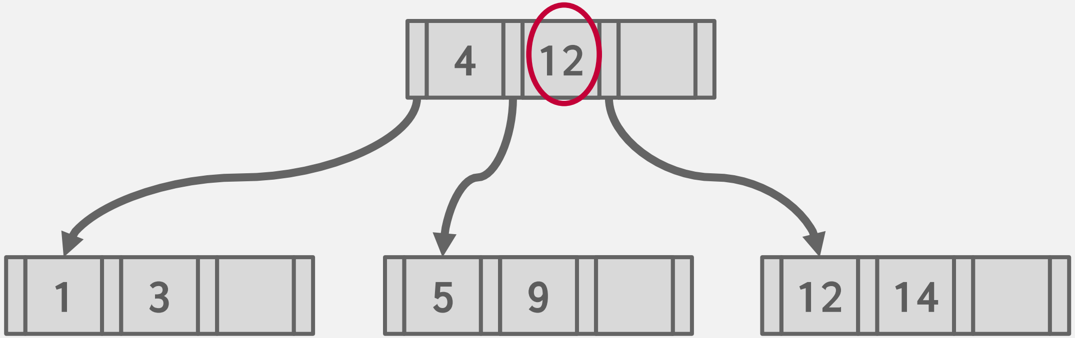
DELETE THE KEY 6



Borrow from a “rich” neighbor.

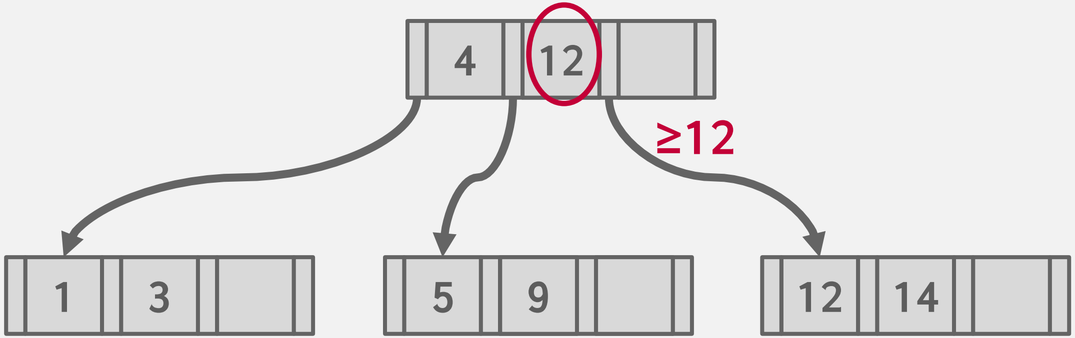
Could borrow from either neighbor.

DELETE THE KEY 6



Borrow from a "rich" neighbor.
Could borrow from either neighbor.

DELETE THE KEY 6

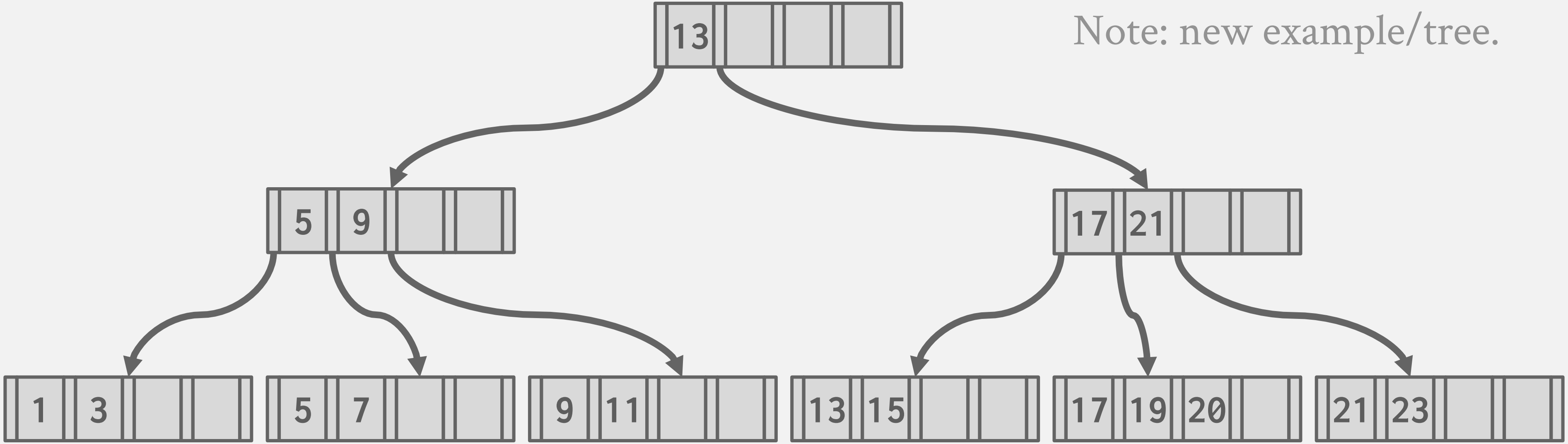


Borrow from a "rich" neighbor.

Could borrow from either neighbor.

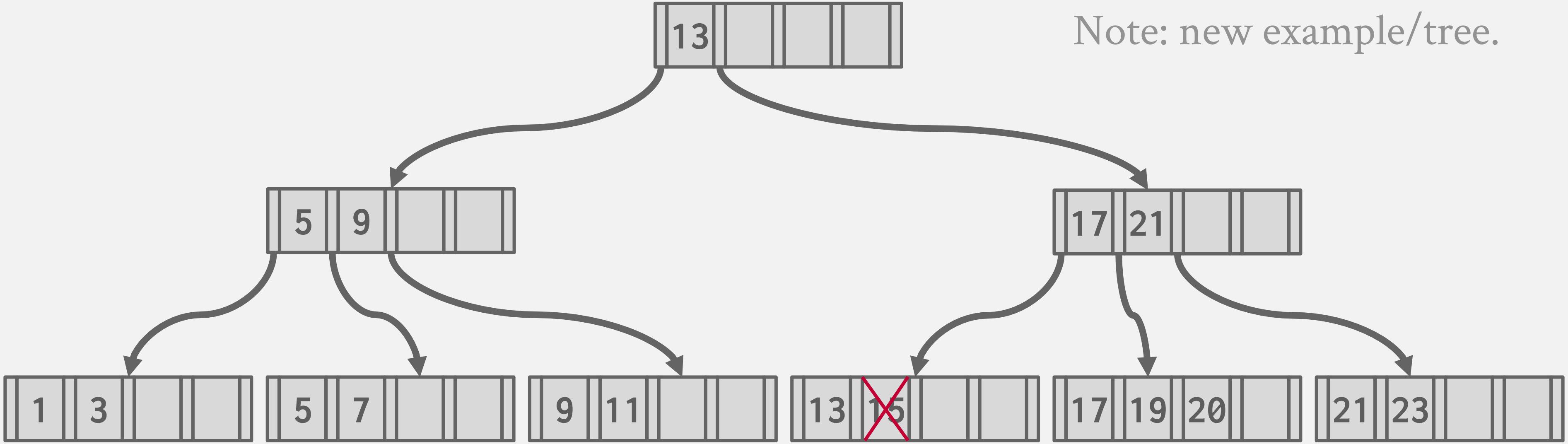
DELETE THE KEY 15

Note: new example/tree.

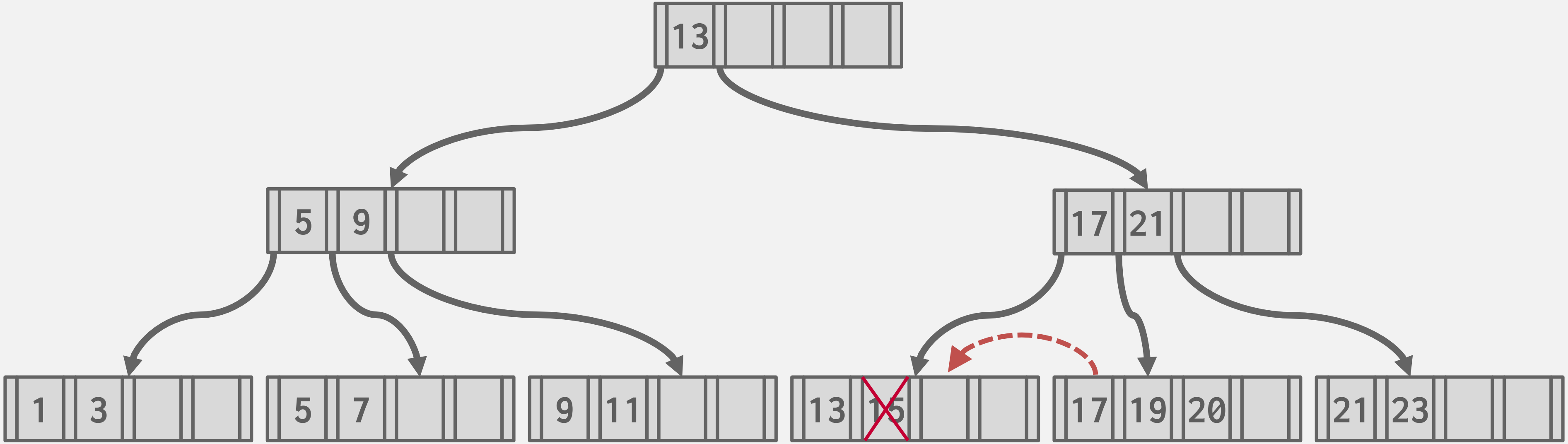


DELETE THE KEY 15

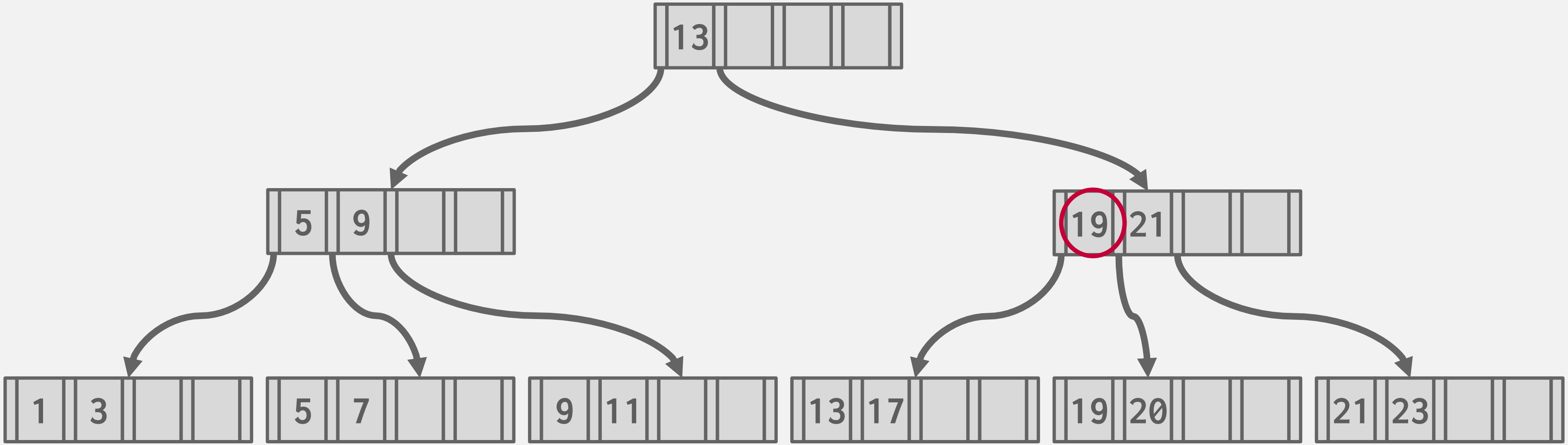
Note: new example/tree.



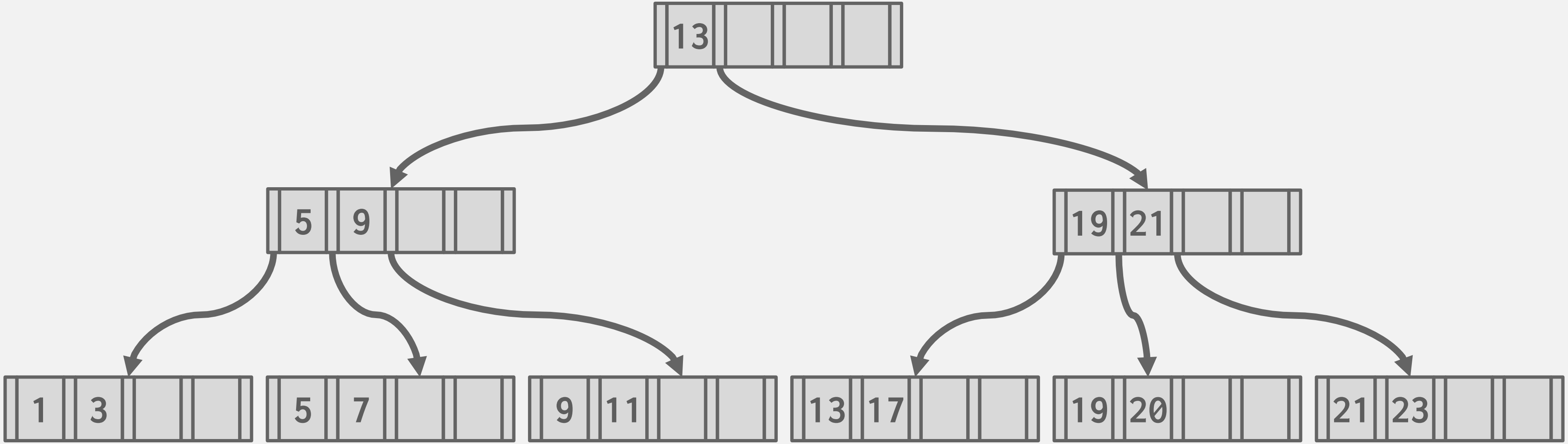
DELETE THE KEY 15



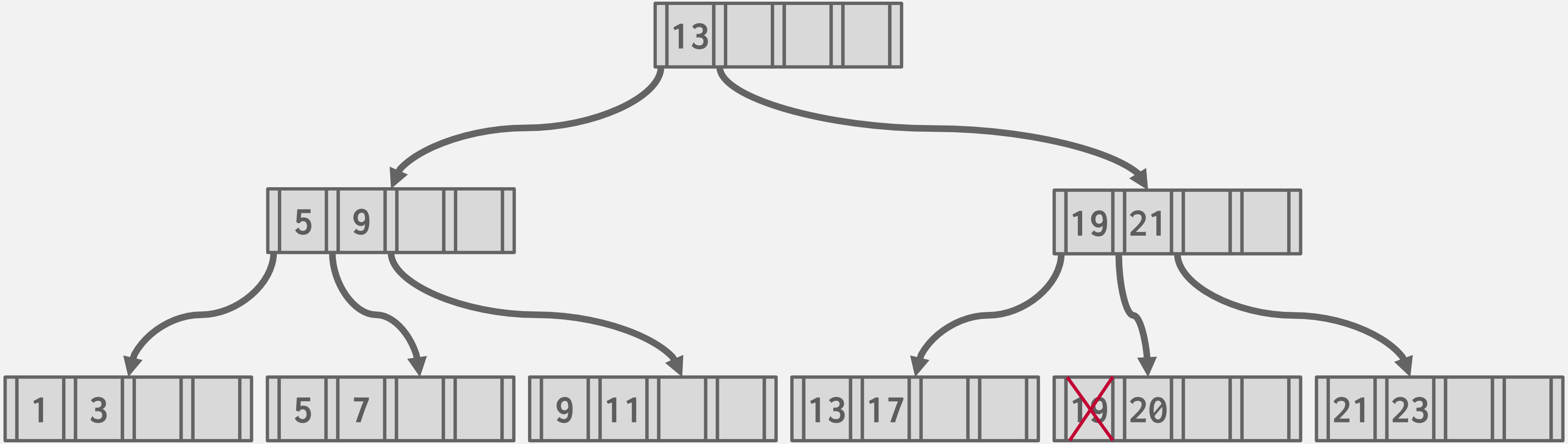
DELETE THE KEY 15



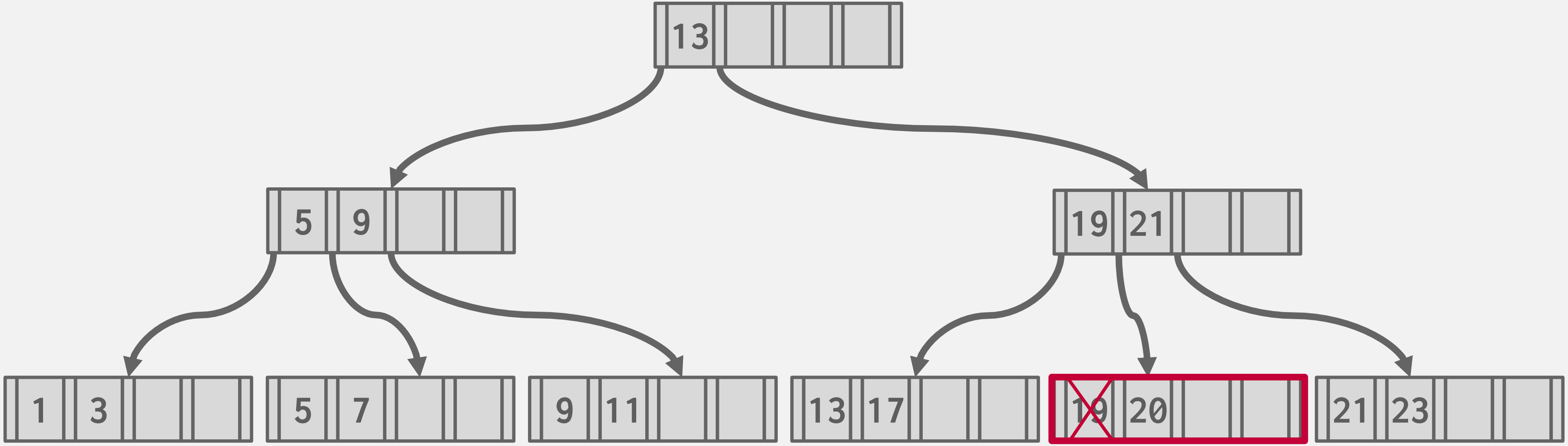
NEXT, DELETE THE KEY 19



NEXT, DELETE THE KEY 19



NEXT, DELETE THE KEY 19

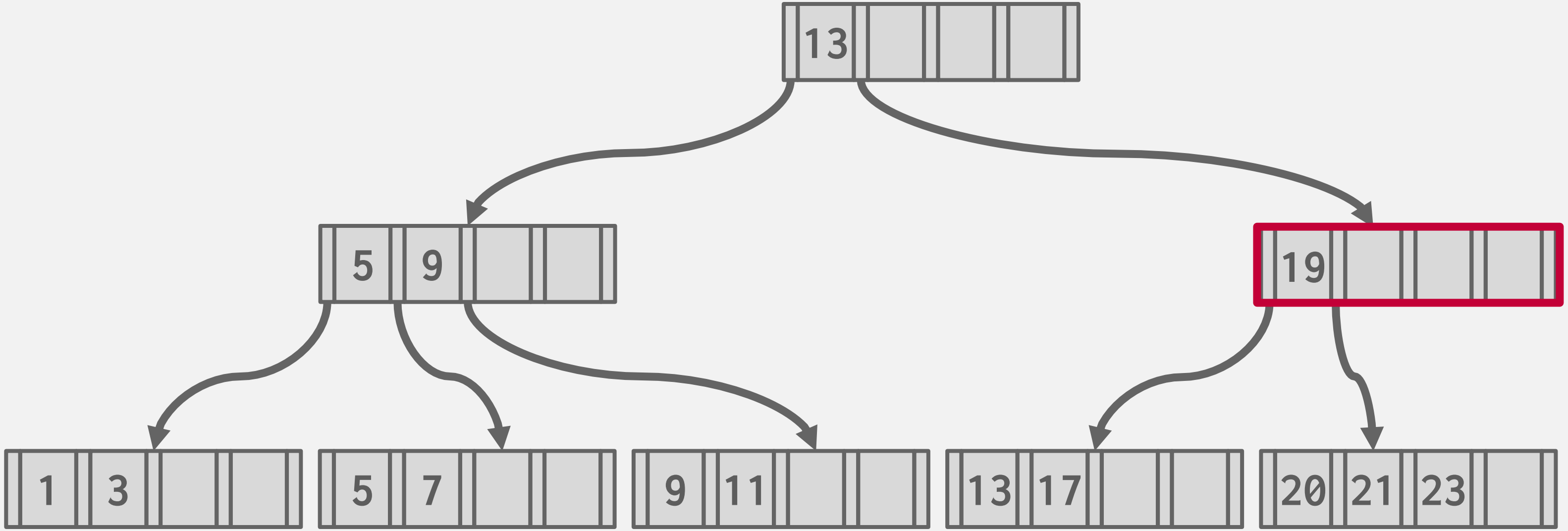


Under-filled.

No "rich" neighbors to borrow.

Merge with a sibling

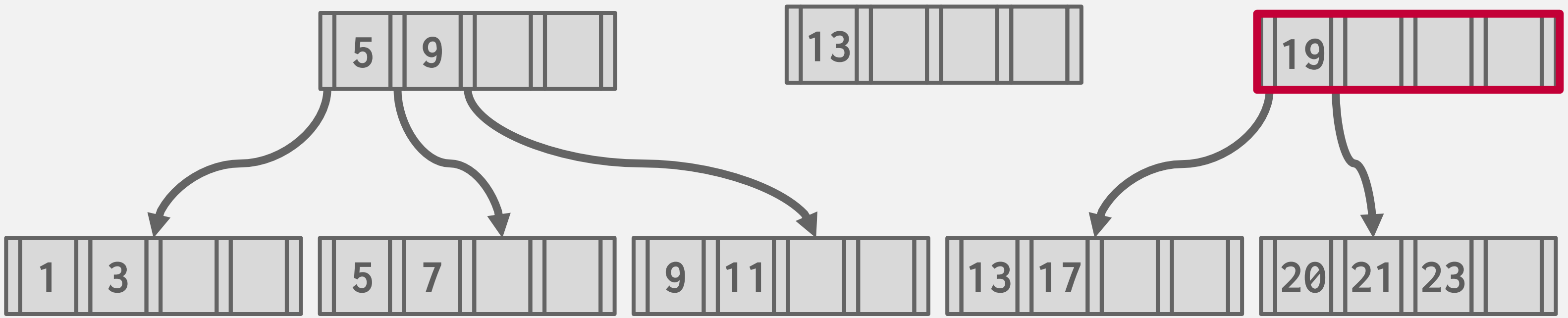
NEXT, DELETE THE KEY 19



This node is under-filled!

Pull-down.

NEXT, DELETE THE KEY 19

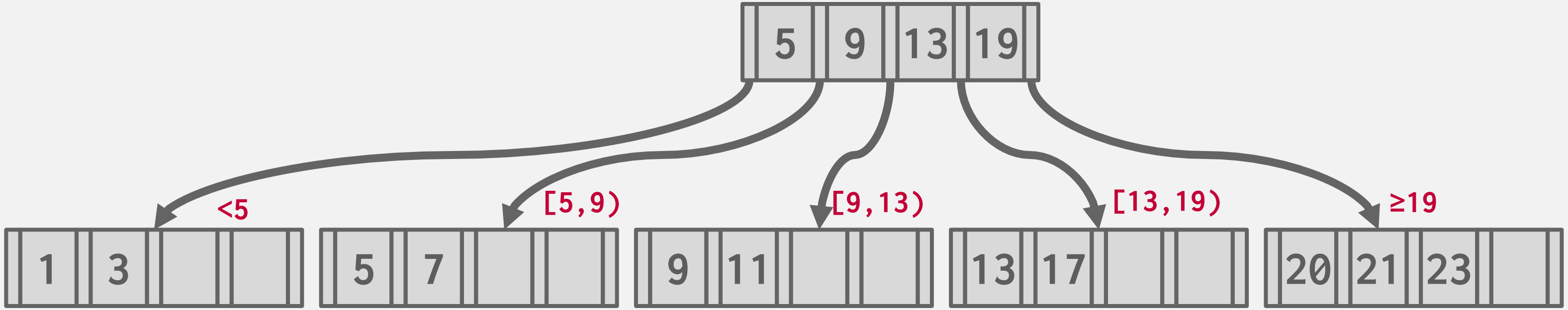


This node is under-filled!

Pull-down.

NEXT, 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)`

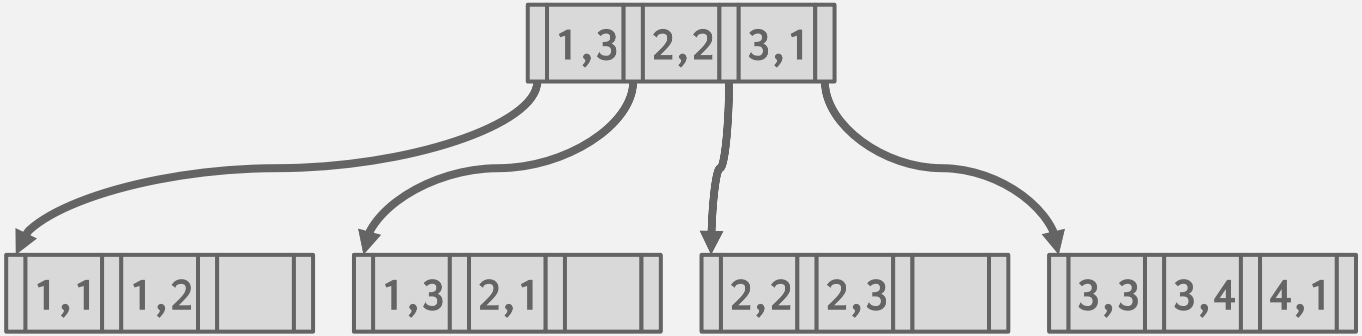
→ Supported: `(a=1 AND b=2)`

→ NOT (generally) supported: `(b=2)`, `(c=3)`

For a hash index, we must have all attributes in search key.

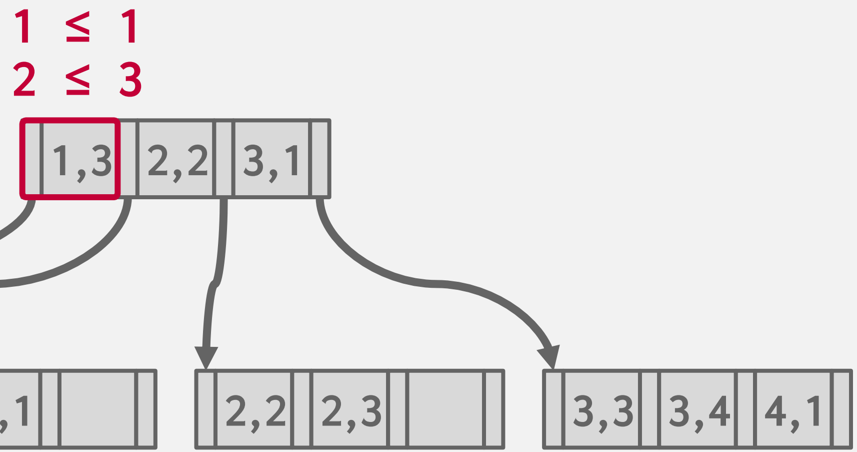
SELECTION CONDITIONS

Find Key=(1,2)



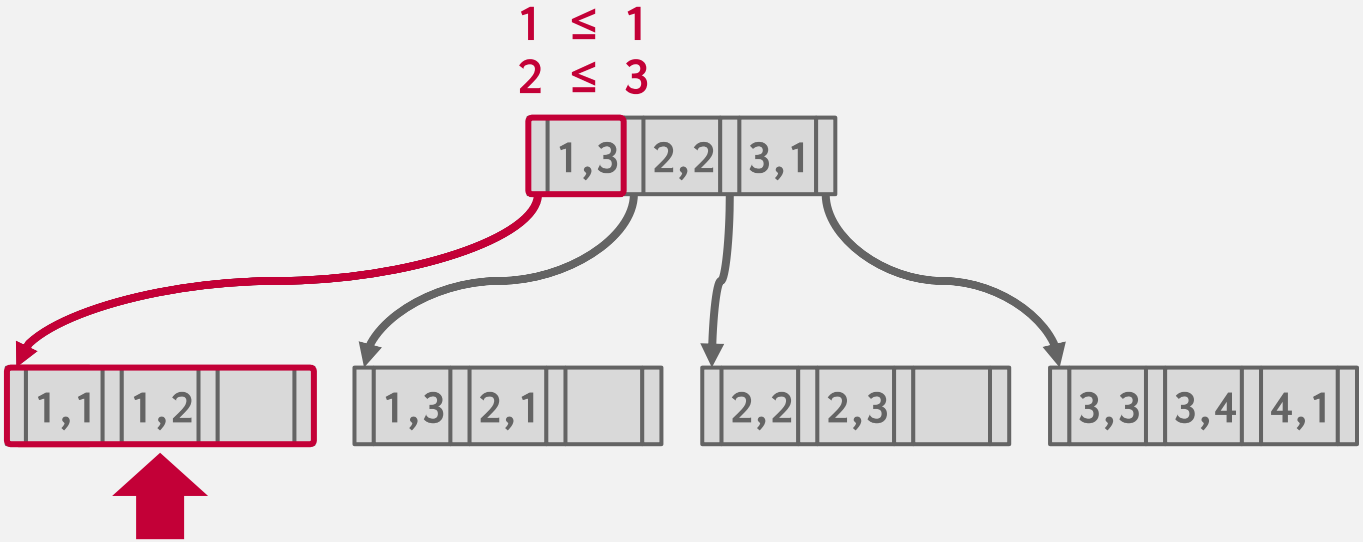
SELECTION CONDITIONS

Find Key=(1,2)



SELECTION CONDITIONS

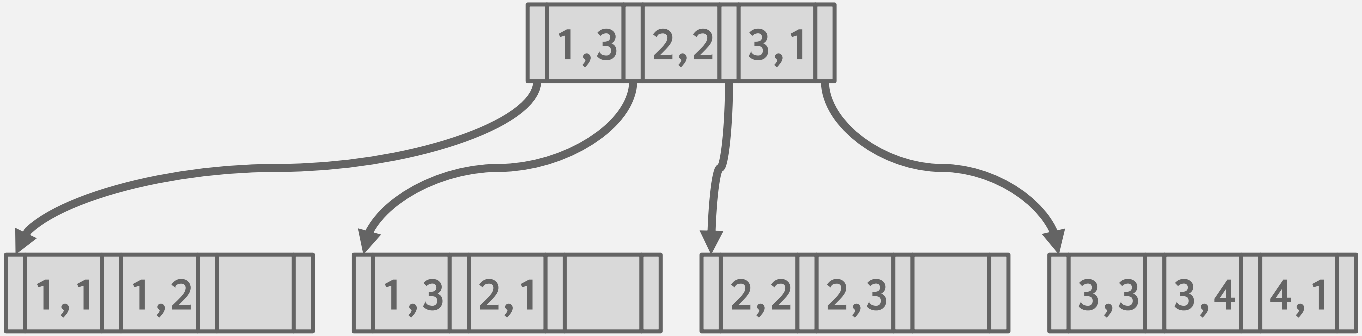
Find Key=(1,2)



SELECTION CONDITIONS

Find Key=(1,2)

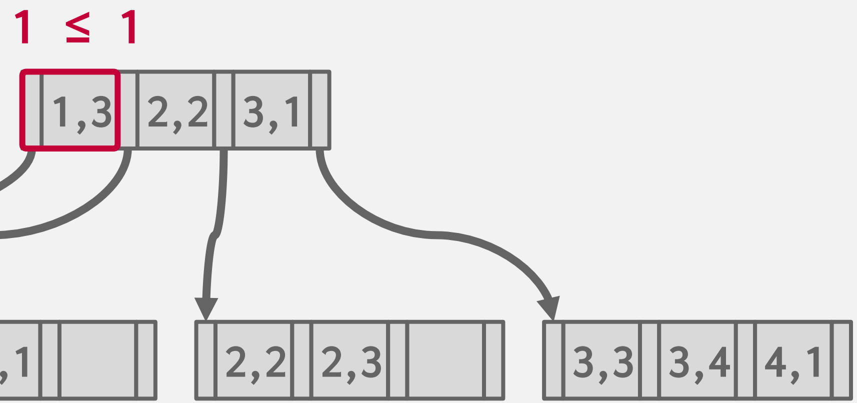
Find Key=(1,*)



SELECTION CONDITIONS

Find Key=(1,2)

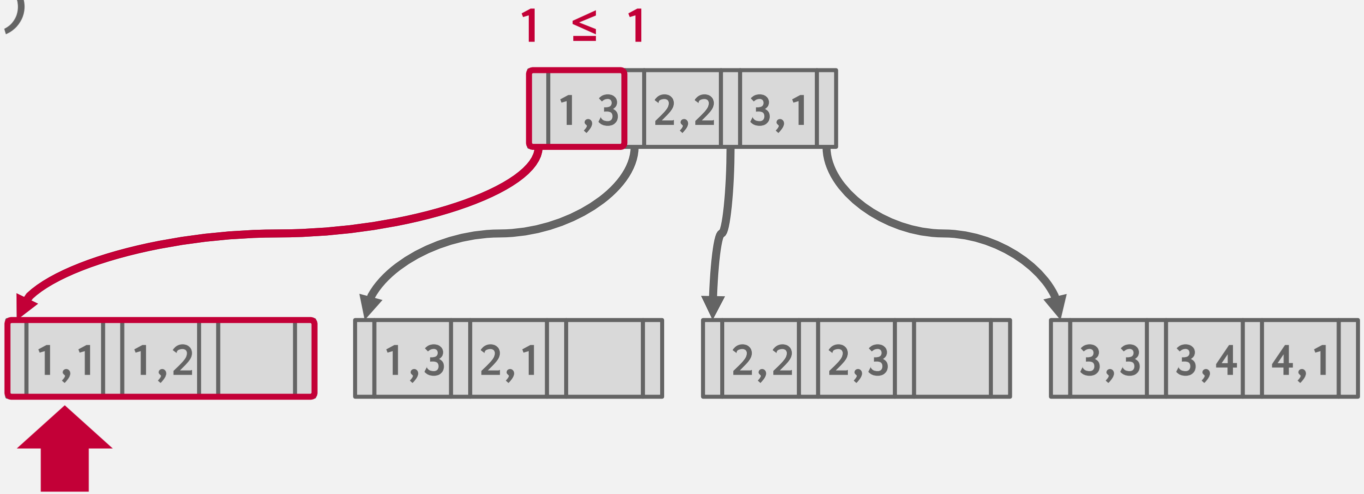
Find Key=(1,*)



SELECTION CONDITIONS

Find Key=(1,2)

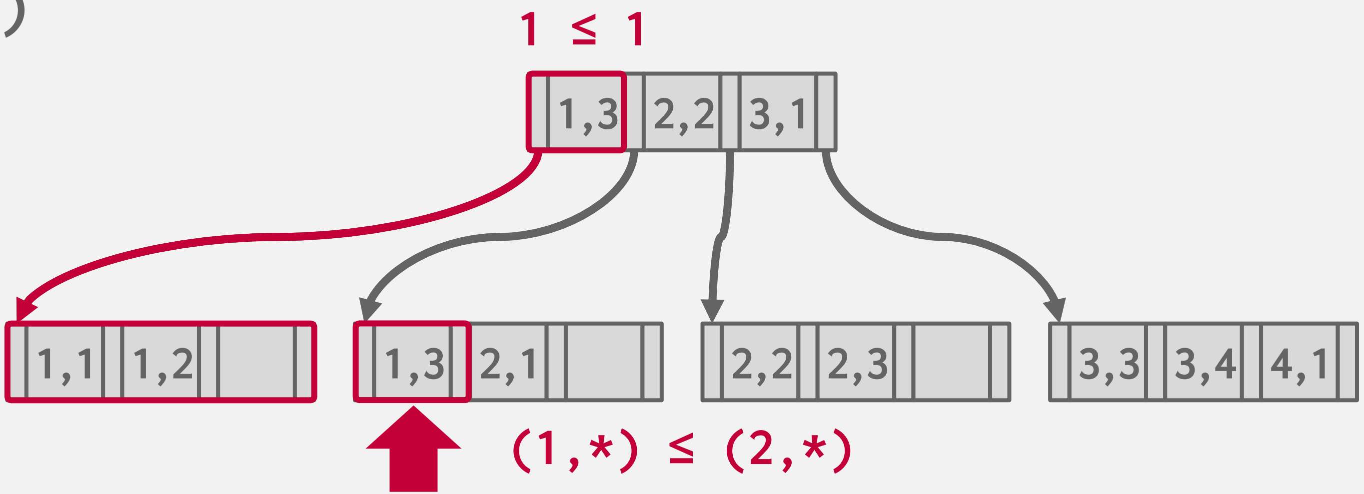
Find Key=(1,*)



SELECTION CONDITIONS

Find Key=(1,2)

Find Key=(1,*)

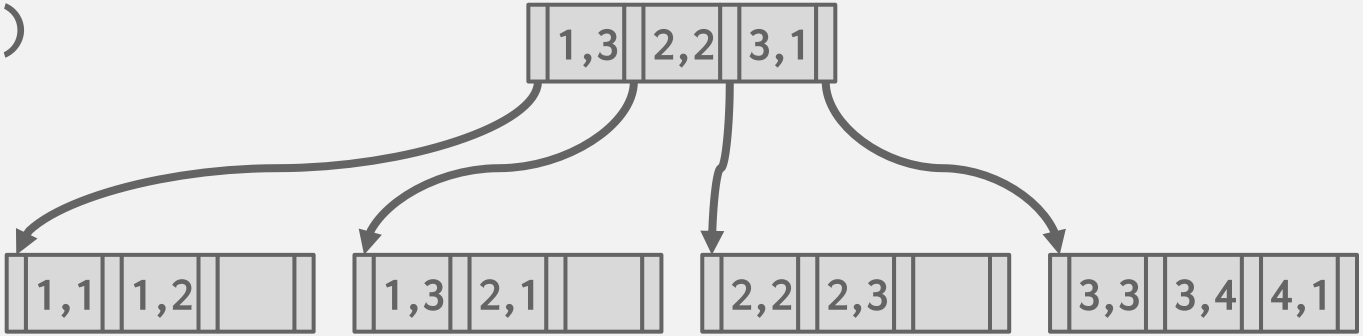


SELECTION CONDITIONS

Find Key=(1,2)

Find Key=(1,*)

Find Key=(*,1)

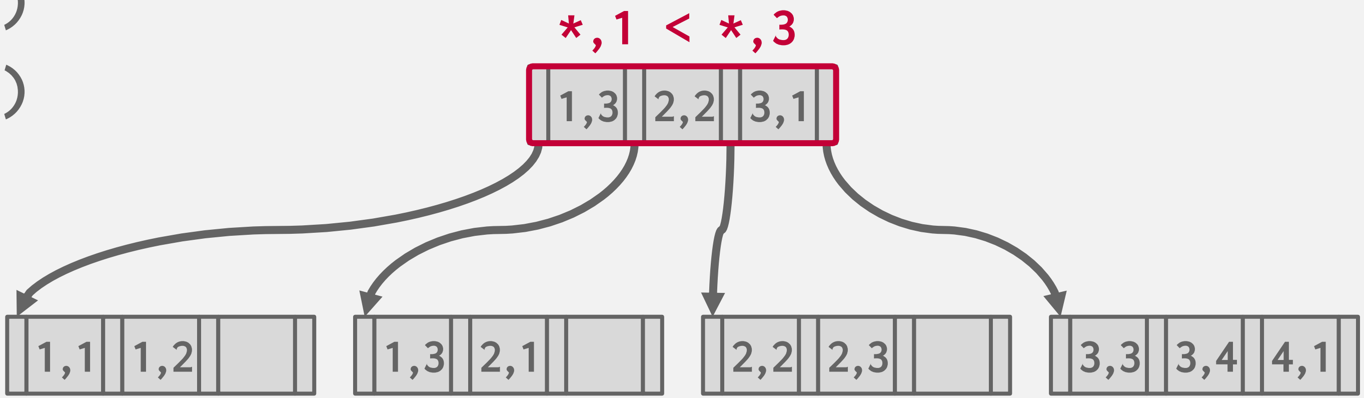


SELECTION CONDITIONS

Find Key=(1,2)

Find Key=(1,*)

Find Key=(*,1)

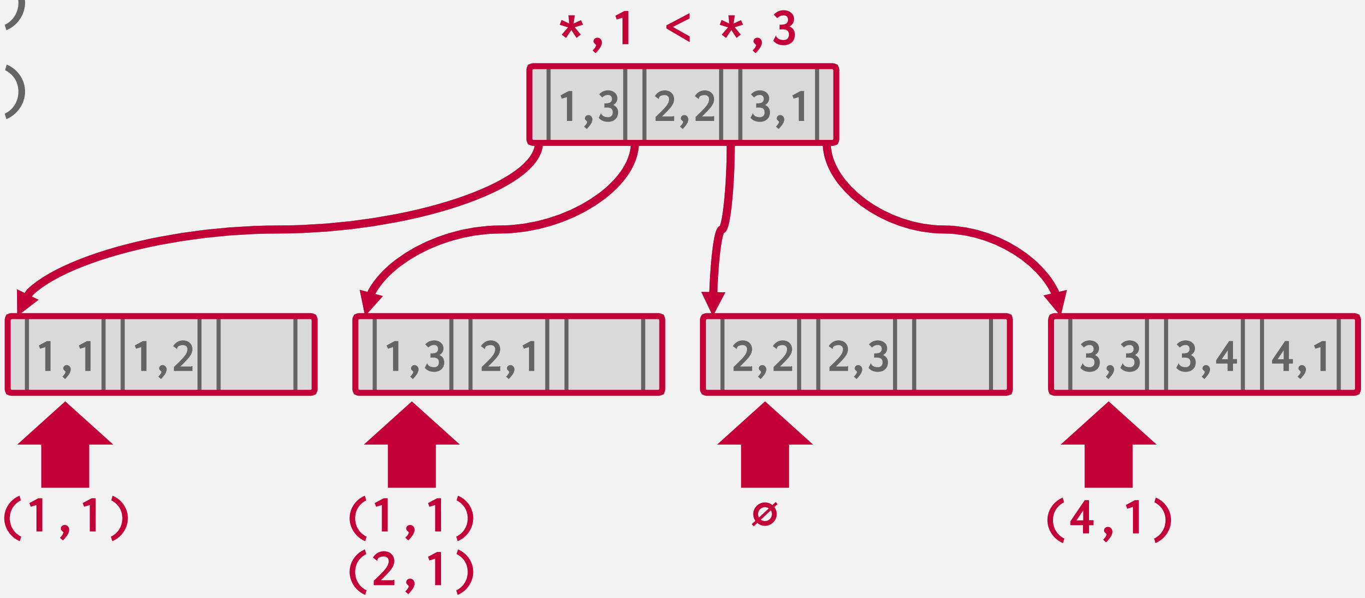


SELECTION CONDITIONS

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.

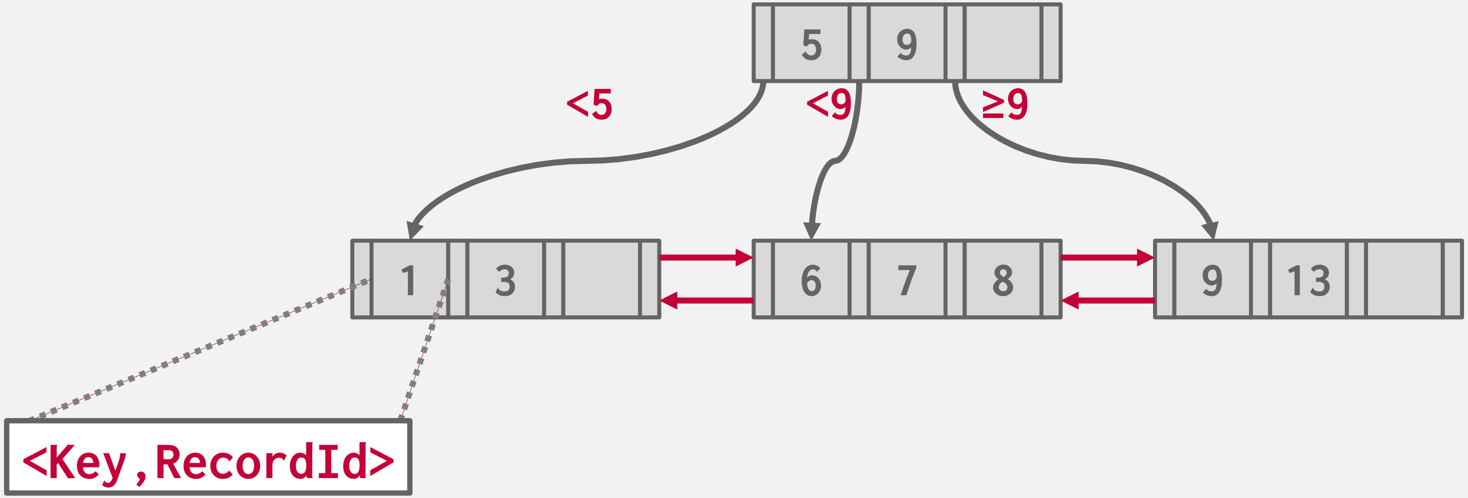
→ The DBMS can still use partial keys to find tuples.

Approach #2: Overflow Leaf Nodes

→ Allow leaf nodes to spill into overflow nodes that contain the duplicate keys.

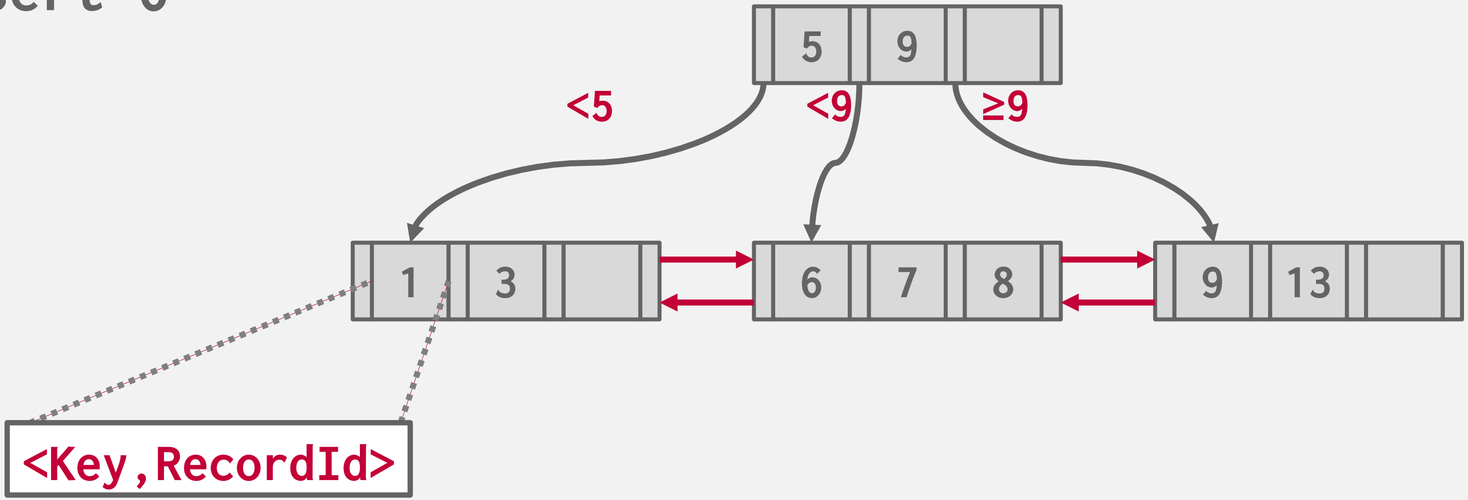
→ This is more complex to maintain and modify.

B+TREE – APPEND RECORD ID



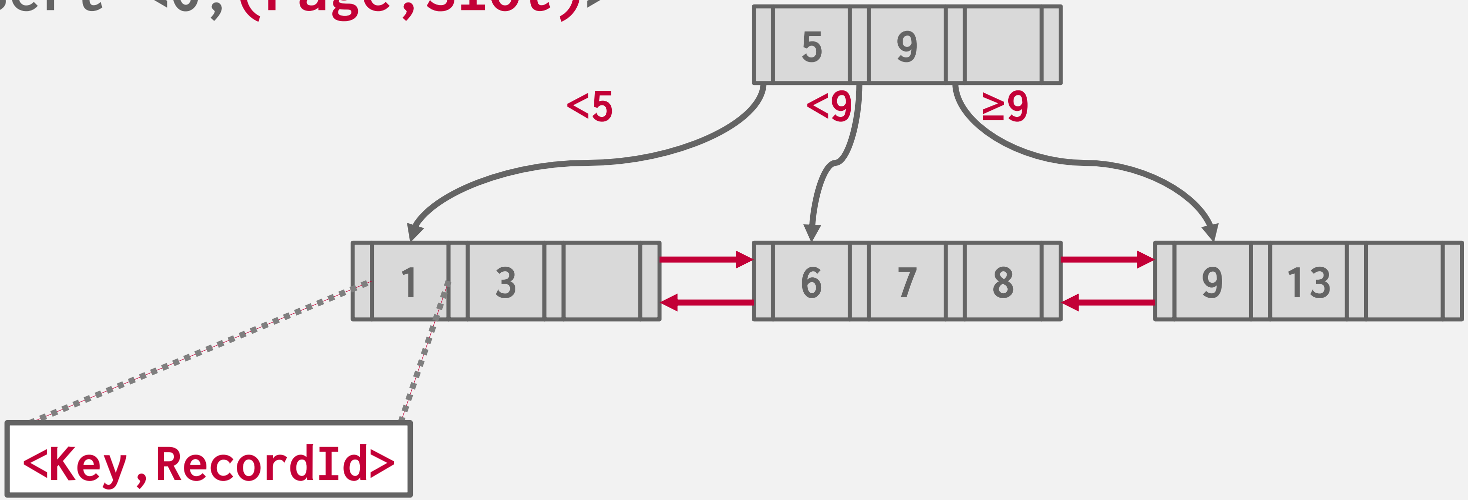
B+TREE – APPEND RECORD ID

Insert 6



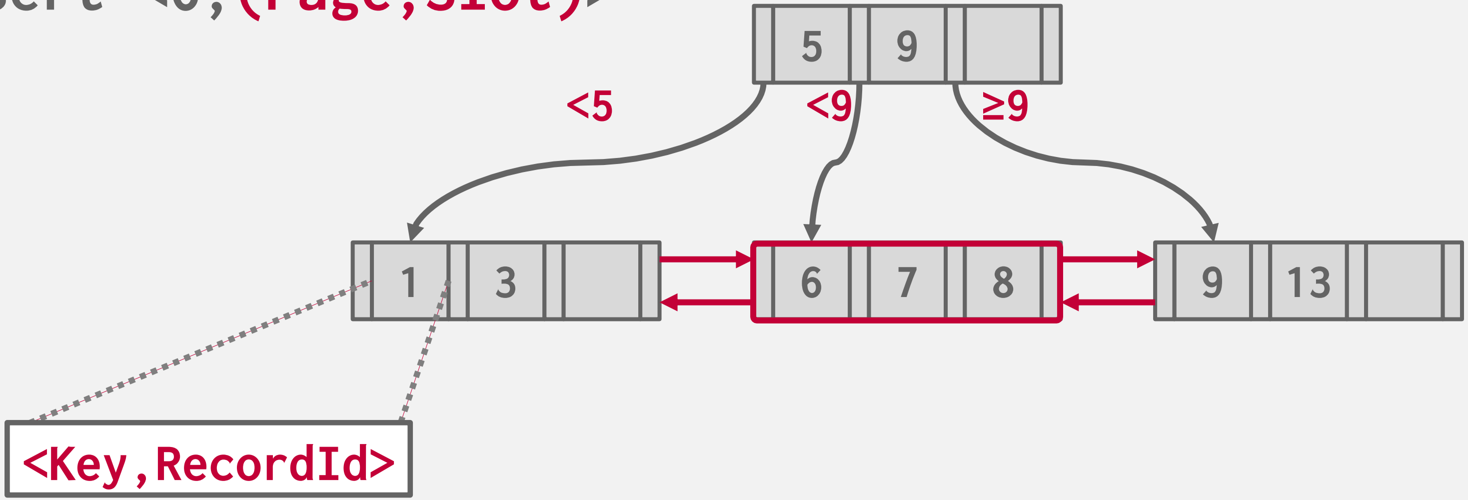
B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



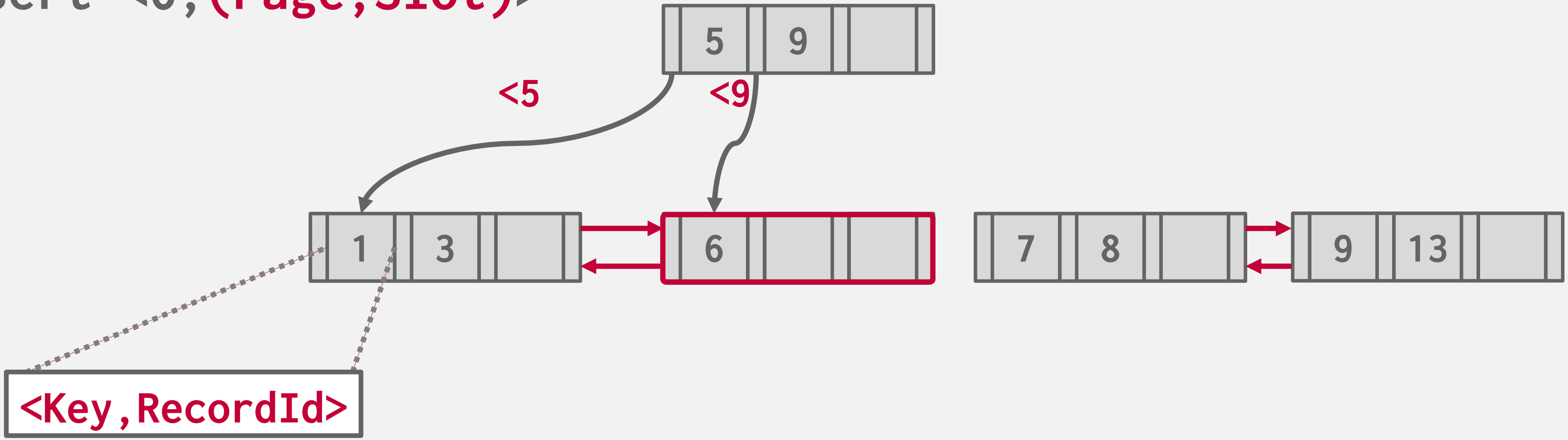
B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



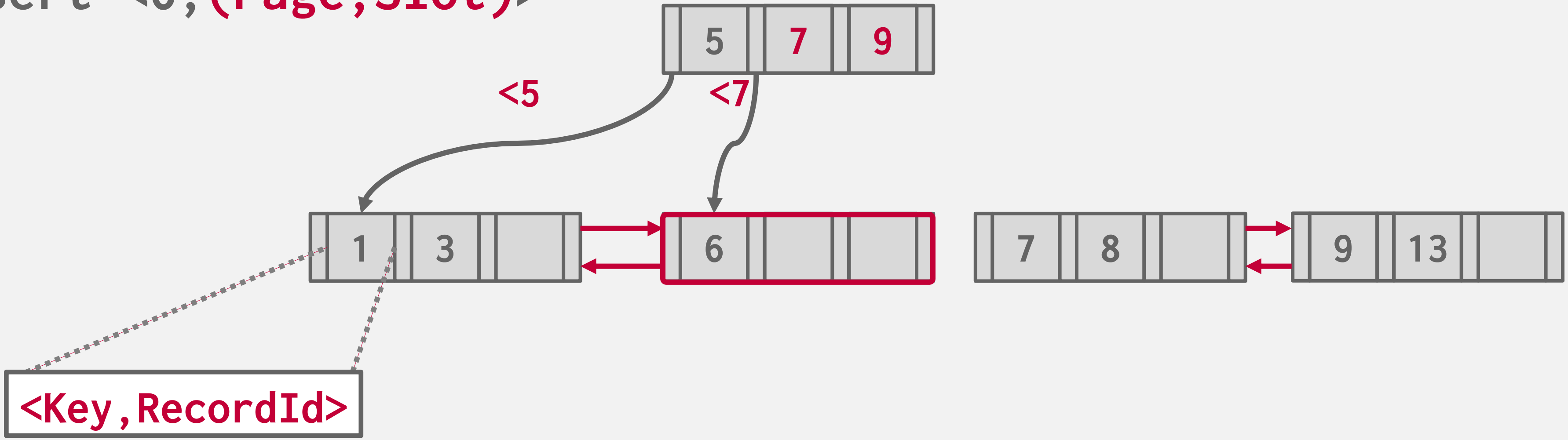
B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



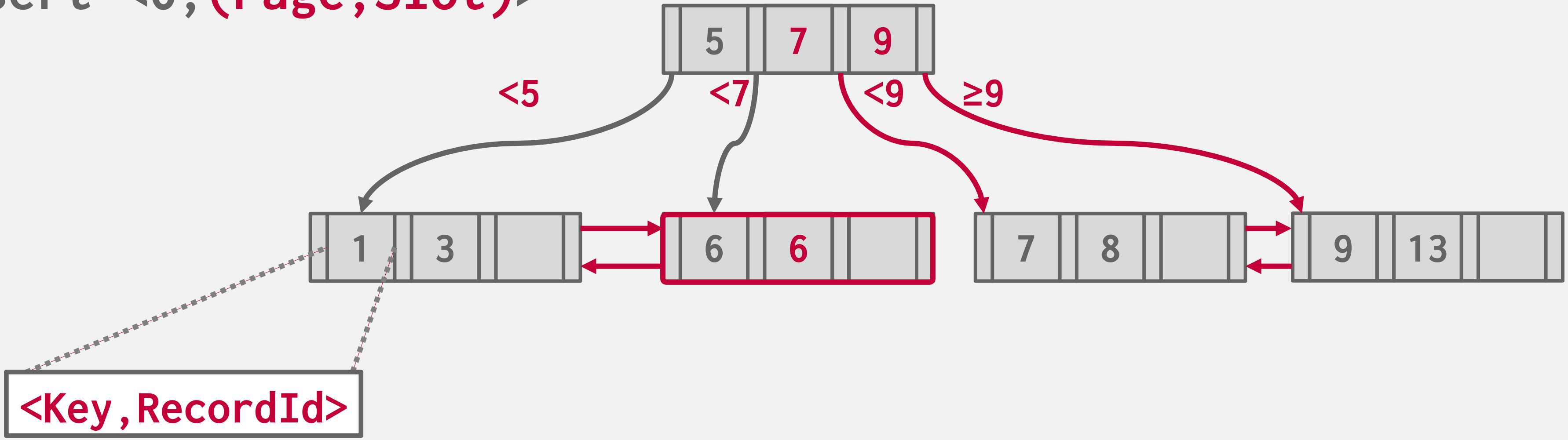
B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



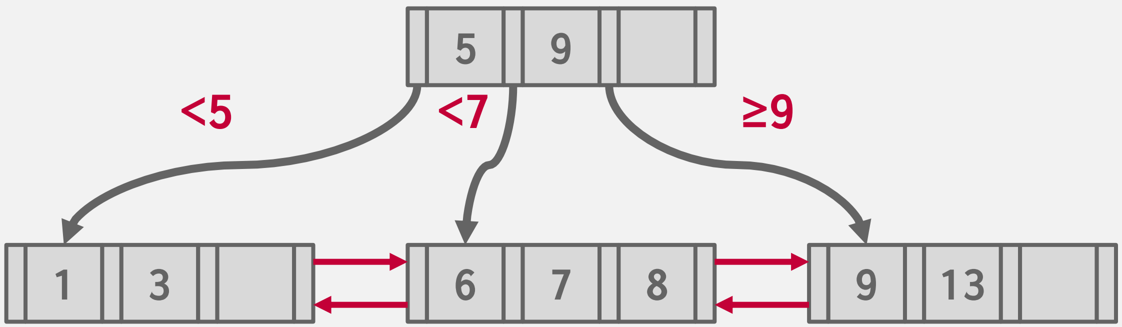
B+TREE – APPEND RECORD ID

Insert <6,(Page,Slot)>



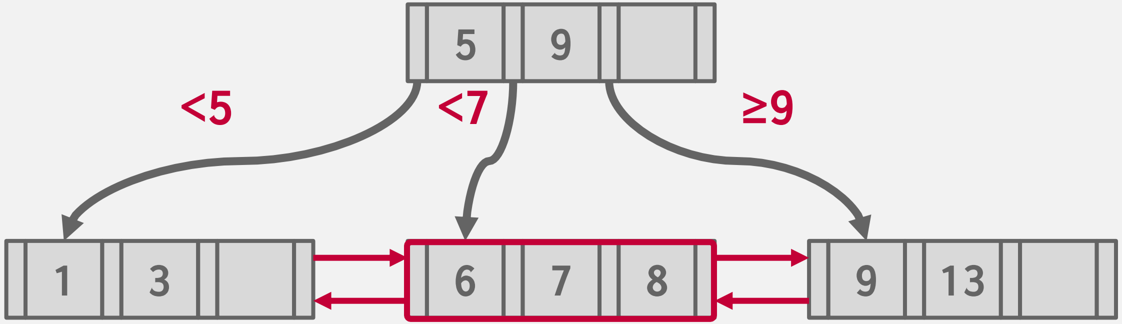
B+TREE – OVERFLOW LEAF NODES

Insert 6



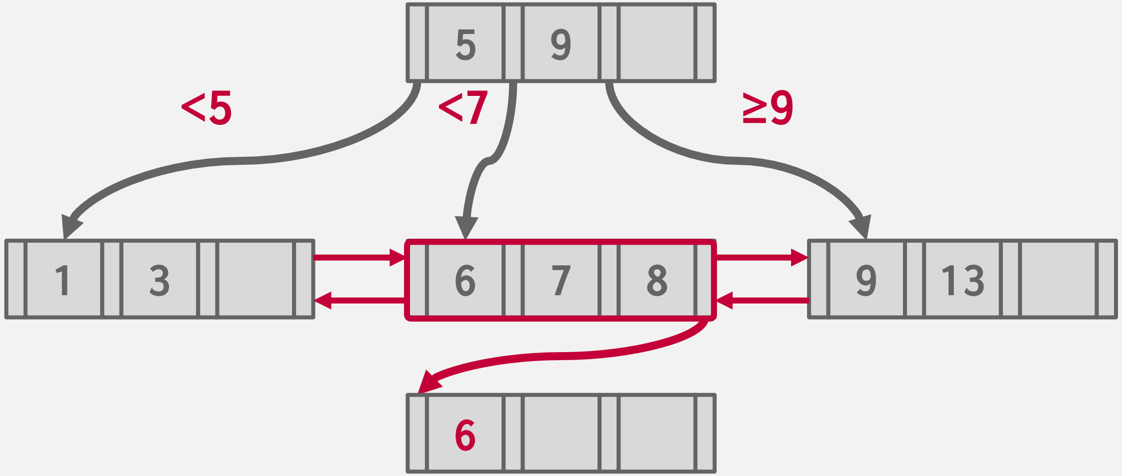
B+TREE – OVERFLOW LEAF NODES

Insert 6



B+TREE – OVERFLOW LEAF NODES

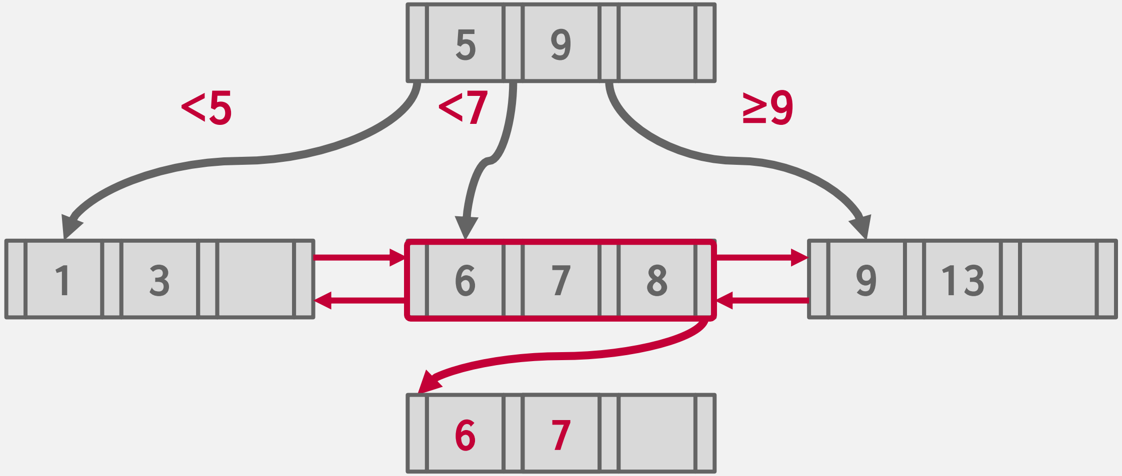
Insert 6



B+TREE – OVERFLOW LEAF NODES

Insert 6

Insert 7

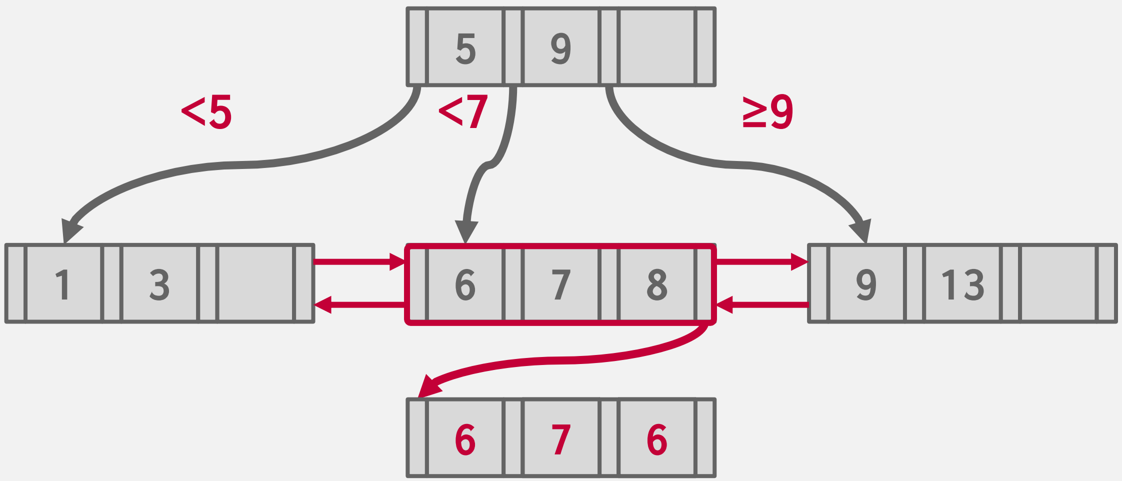


B+TREE – OVERFLOW LEAF NODES

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.

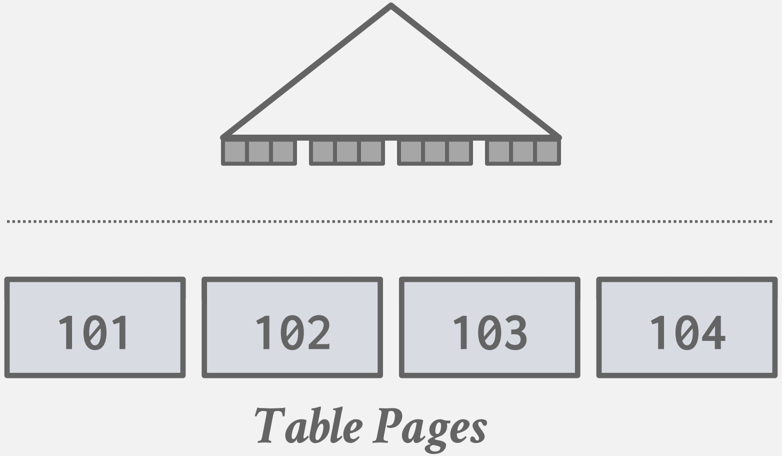
- → If a table does not contain a primary key, the DBMS will automatically make a hidden primary key.

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.

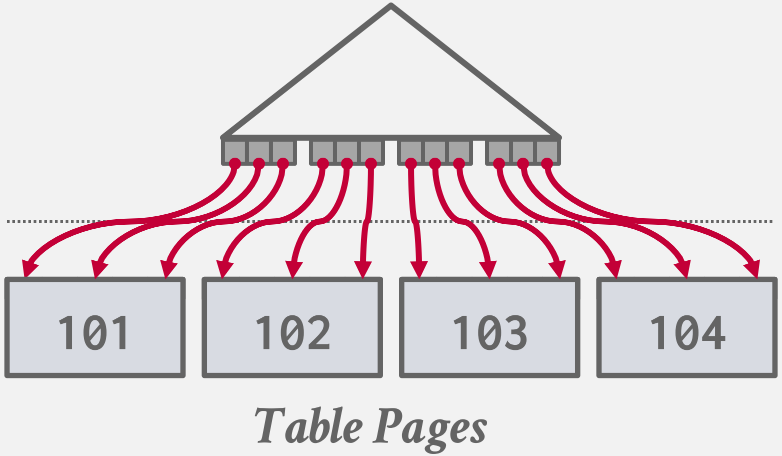
This will always be better than sorting data for each query.



CLUSTERED B+TREE

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

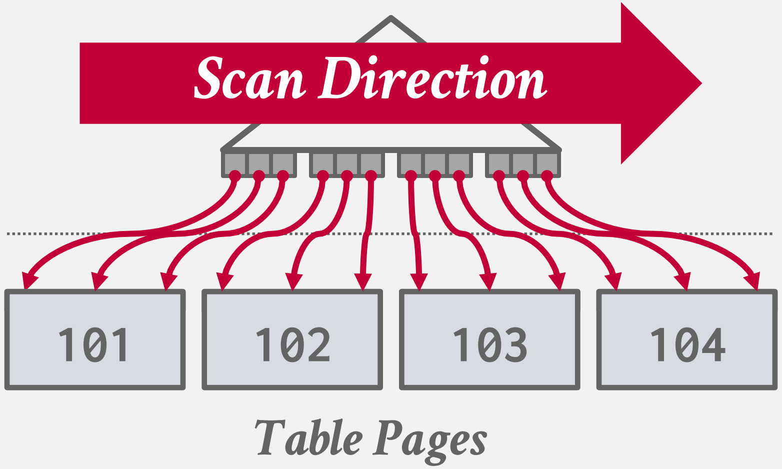
This will always be better than sorting data for each query.



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.

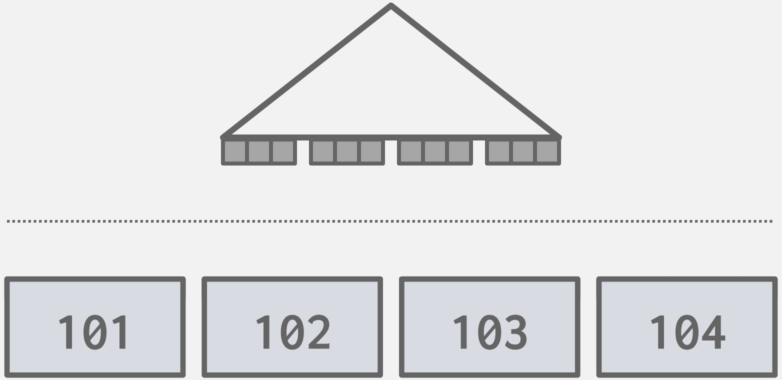


INDEX SCAN PAGE SORTING

Retrieving tuples in the order they appear in a non-clustered index is inefficient due to redundant reads.

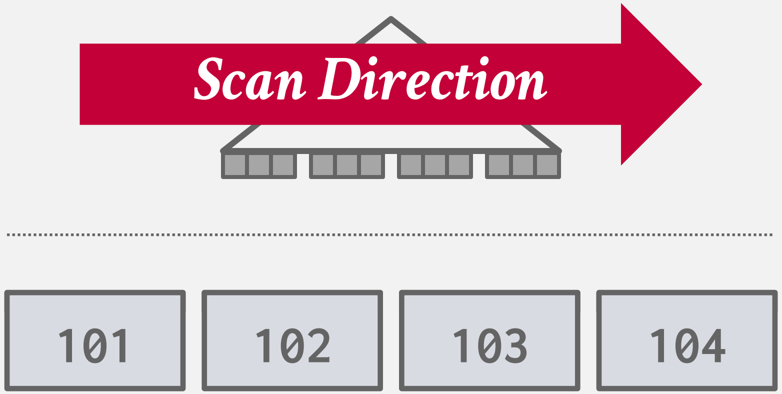
A better approach is to find all the tuples that the query needs and then sort them based on their page ID.

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.

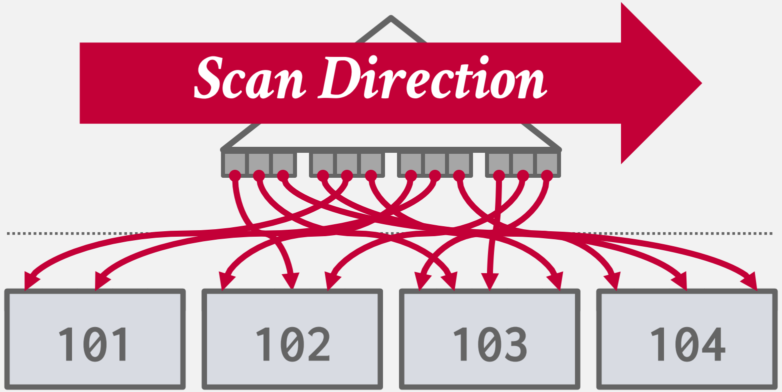
The DBMS retrieves each page once.

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.

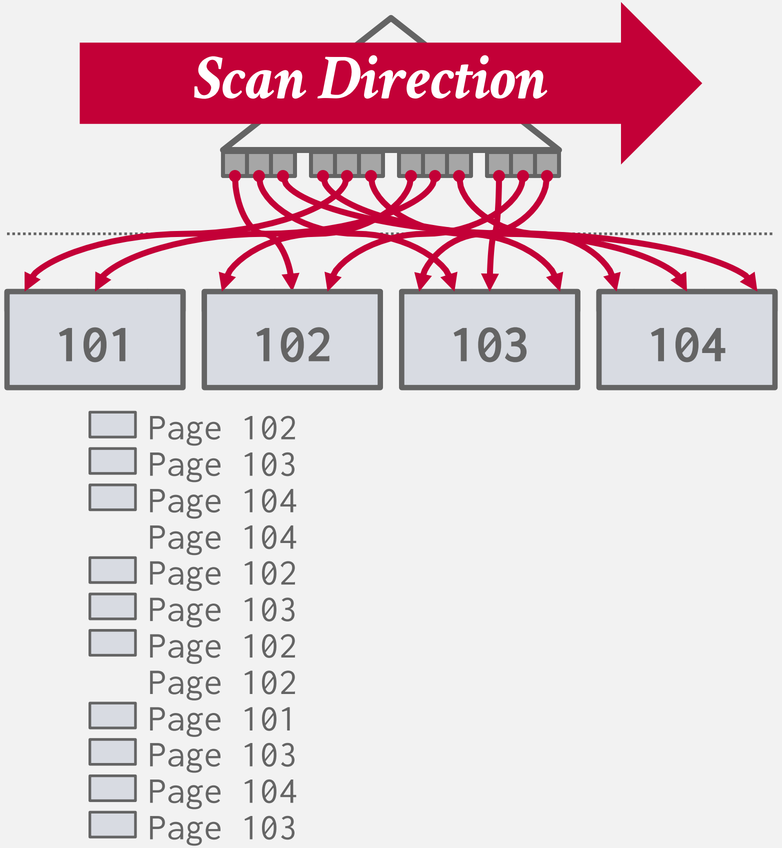


INDEX SCAN PAGE SORTING

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.

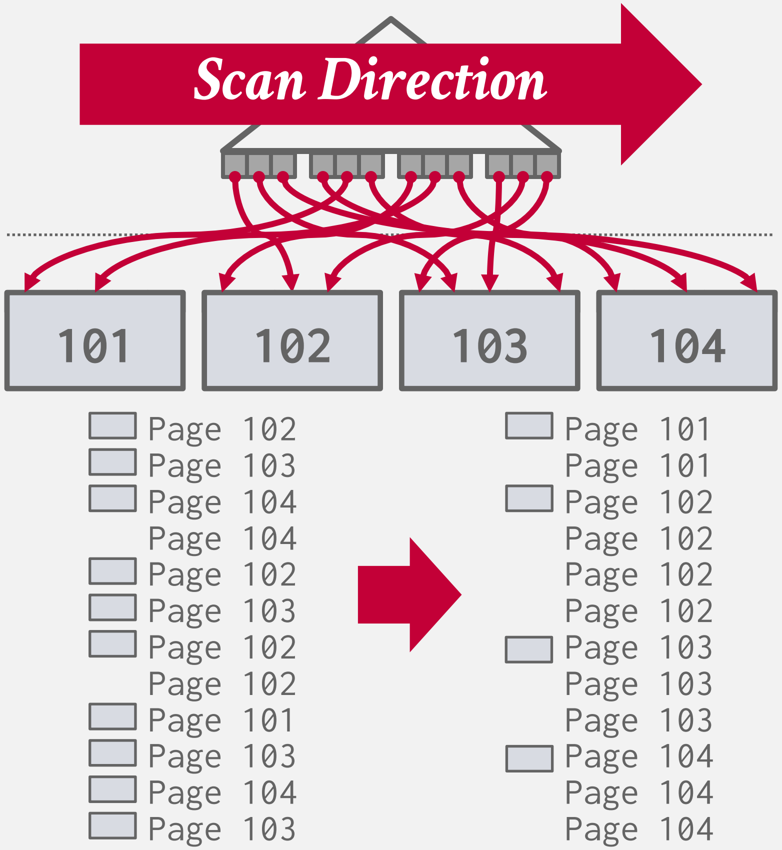


INDEX SCAN PAGE SORTING

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.



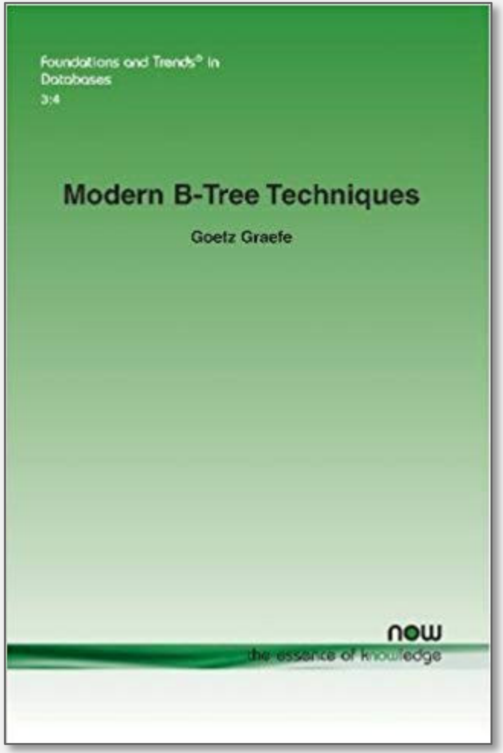
B+TREE DESIGN CHOICES

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.

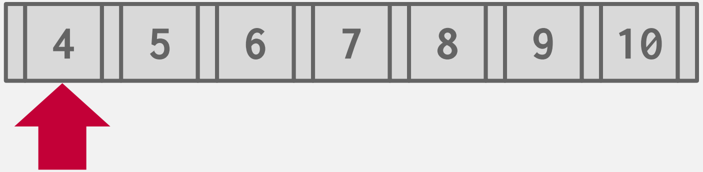
INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

→ Use SIMD to vectorize comparisons.

Find Key=8



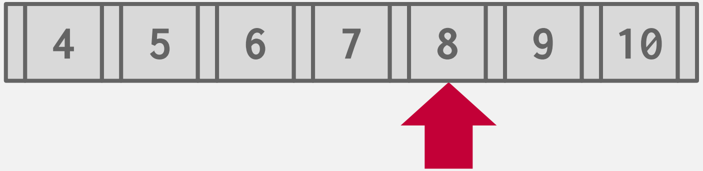
INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

→ Use SIMD to vectorize comparisons.

Find Key=8



INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

→ Use SIMD to vectorize comparisons.

Find Key=8

INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

→ Use SIMD to vectorize comparisons.

Find Key=8


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

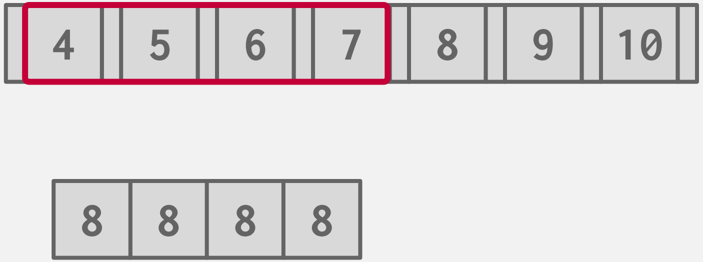
INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

→ Use SIMD to vectorize comparisons.

Find Key=8



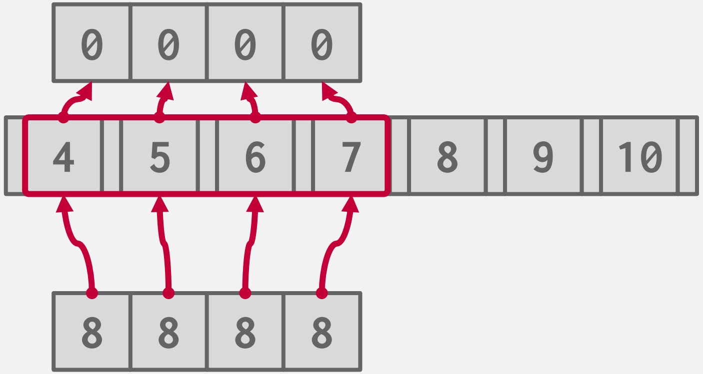

```
_mm_cmpeq_epi32_mask(a, b)
```

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.

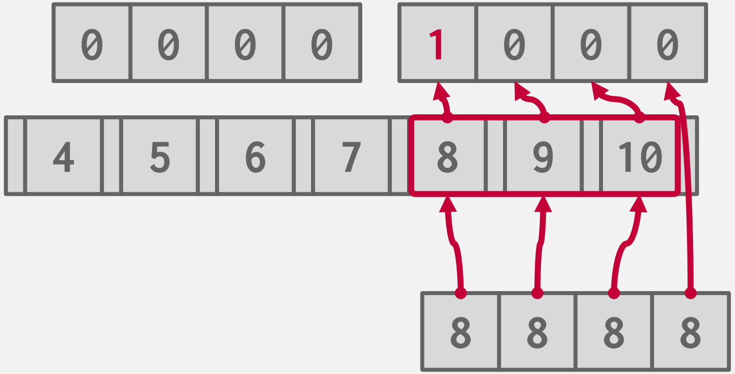
→ Use SIMD to vectorize comparisons.

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

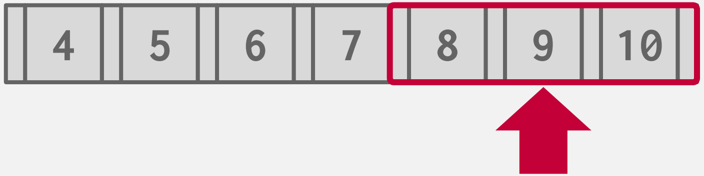
Approach #1: Linear

→ 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

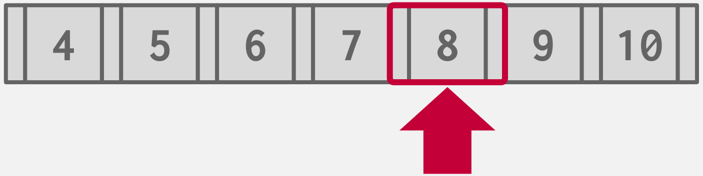
Approach #1: Linear

→ 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.

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.

*Offset: (8-4)*7/(10-4)=4*

INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

→ Use SIMD to vectorize comparisons.

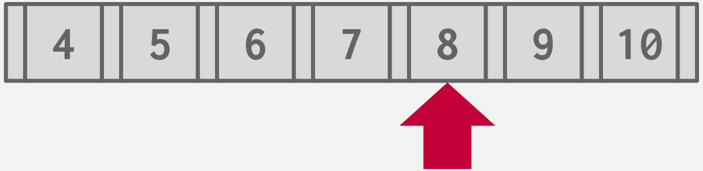
Approach #2: Binary

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

Approach #3: Interpolation

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

*Offset: (8-4)*7/(10-4)=4*



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.

Binary vs. Interpolation search: Tradeoffs change based on hardware trends.

Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?

Peter Van Sandt, Yannis Chronis, Jignesh M. Patel
Department of Computer Sciences, University of Wisconsin-Madison
{van-sandt,chronis,jignesh}@cs.wisc.edu

ABSTRACT

In this paper, we focus on the problem of searching sorted, in-memory datasets. This is a key data operation, and Binary Search is the de facto algorithm that is used in practice. We consider an alternative, namely Interpolation Search, which can take advantage of hardware trends by using complex calculations to save memory accesses. Historically, Interpolation Search was found to underperform compared to other search algorithms in this setting, despite its superior asymptotic complexity. Also, Interpolation Search is known to perform poorly on non-uniform data. To address these issues, we introduce SIP (Slope reuse Interpolation), an optimized implementation of Interpolation Search, and TIP (Three point Interpolation), a new search algorithm that uses linear fractions to interpolate on non-uniform distributions. We evaluate these two algorithms against a similarly optimized Binary Search method using a variety of real and synthetic datasets. We show that SIP is up to 4 times faster on uniformly distributed data and TIP is 2-3 times faster on non-uniformly distributed data in some cases. We also design a meta-algorithm to switch between these different methods to automate picking the higher performing search algorithm, which depends on factors like data distribution.

CCS CONCEPTS

• **Information systems → Point lookups**; *Main memory engines*.

KEYWORDS

In-memory search; Interpolation Search; Binary Search

ACM ISBN 978-1-4503-5643-5/19/06...$15.00
https://doi.org/10.1145/3299869.3300075

ACM Reference Format:
Peter Van Sandt, Yannis Chronis, Jignesh M. Patel. 2019. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3300075

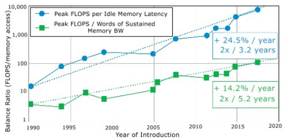


Figure 1: Speed comparison of representative processor and main memory technologies [27]. The performance of processors is measured in FLOPS. The performance of main memory is measured as peak FLOPS to sustained memory bandwidth (GFLOP/sec) / (Words/sec) and peak FLOPS per idle memory latency (GFLOP/sec) ∗ sec. In the conventional von Neumann architectural path, main memory speed is poised to become (relatively) slower compared to the speed of computing inside processors.

1 INTRODUCTION

Searching in-memory, sorted datasets is a fundamental data operation [23]. Today, Binary Search is the de facto search method that is used in practice, as it is an efficient and asymptotically optimal in the worst case algorithm. Binary Search is a primitive in many popular data systems and frameworks (e.g. LevelDB [25] and Pandas [30]).

Designing algorithms around hardware trends can yield significant performance gains. A key technological trend is the diverging CPU and memory speeds, which is illustrated in Figure 1. This trend favors algorithms that can use more computation to reduce memory accesses [4, 6, 16, 21, 27, 38]. The focus of this paper is on exploring the impact of this trend

OPTIMIZATIONS

Prefix Compression

Deduplication

Suffix Truncation

Pointer Swizzling

Bulk Insert

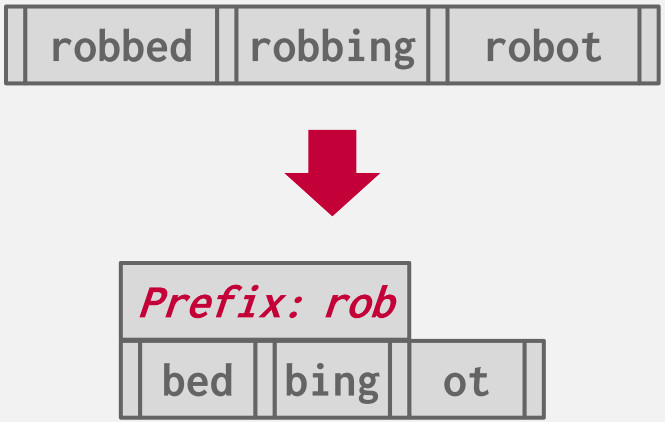
Buffered Updates

Many more…

PREFIX COMPRESSION

Sorted keys in the same leaf node are likely to have the same prefix.

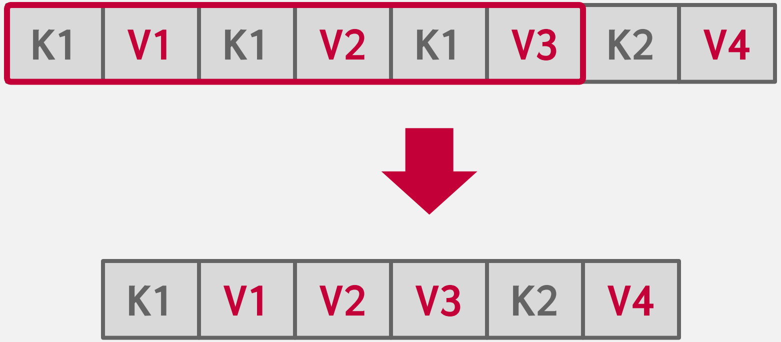
Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.
→ Many variations.



DEDUPLICATION

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

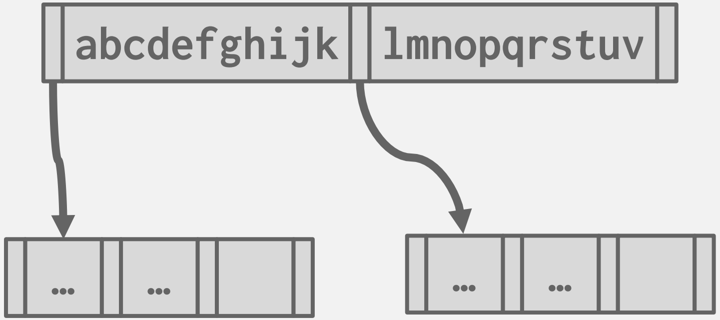
The leaf node can store the key once and then maintain a "posting list" of tuples with that key (similar to what we discussed for hash tables).



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.

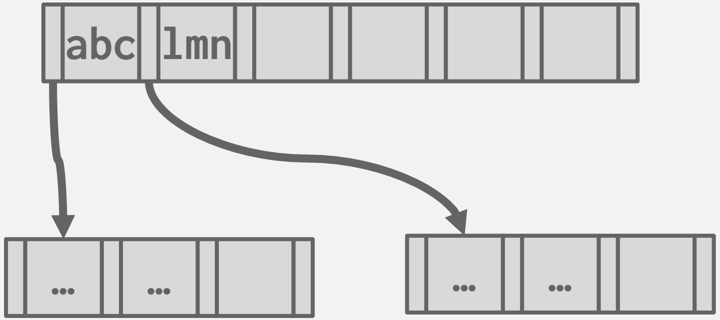


SUFFIX TRUNCATION

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

→ We don't need the entire key.

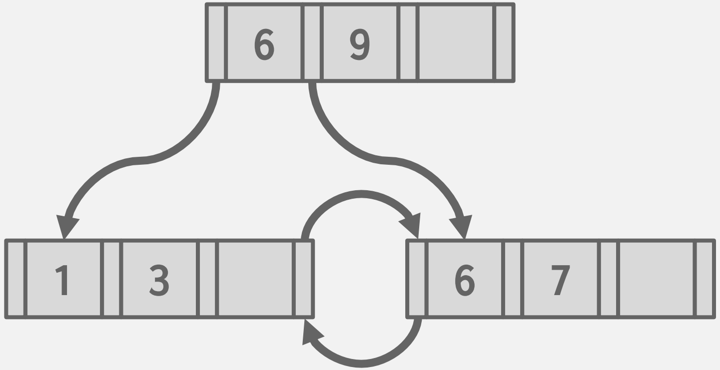
Store a minimum prefix that is needed to correctly route probes into the index.

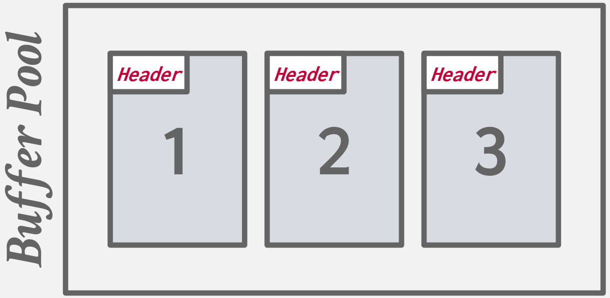


POINTER SWIZZLING

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.

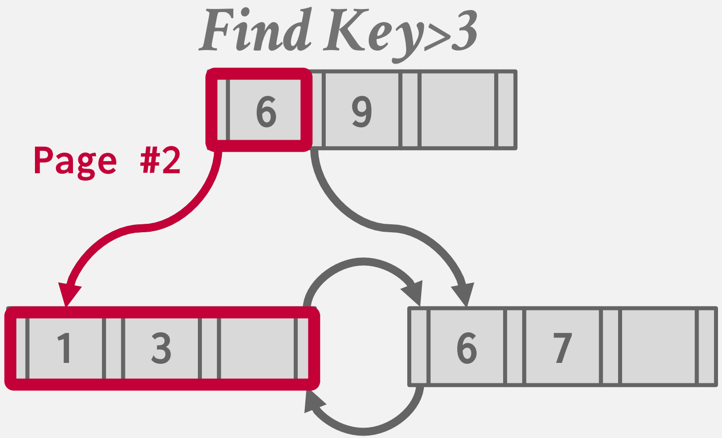


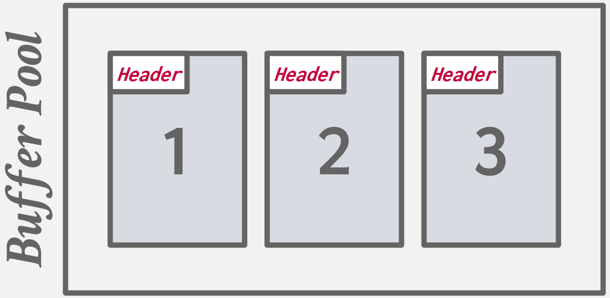


POINTER SWIZZLING

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.

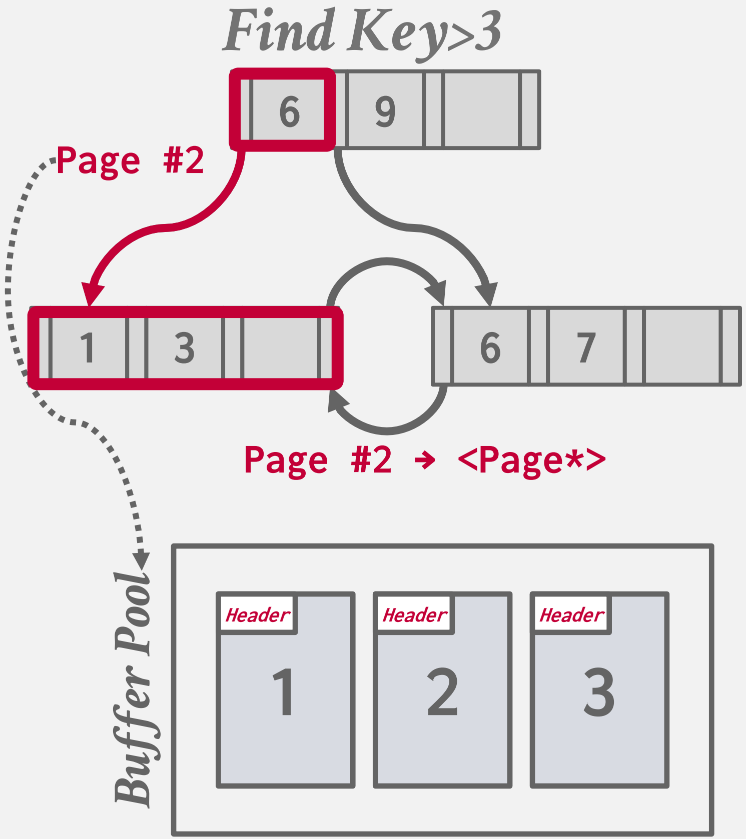




POINTER SWIZZLING

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

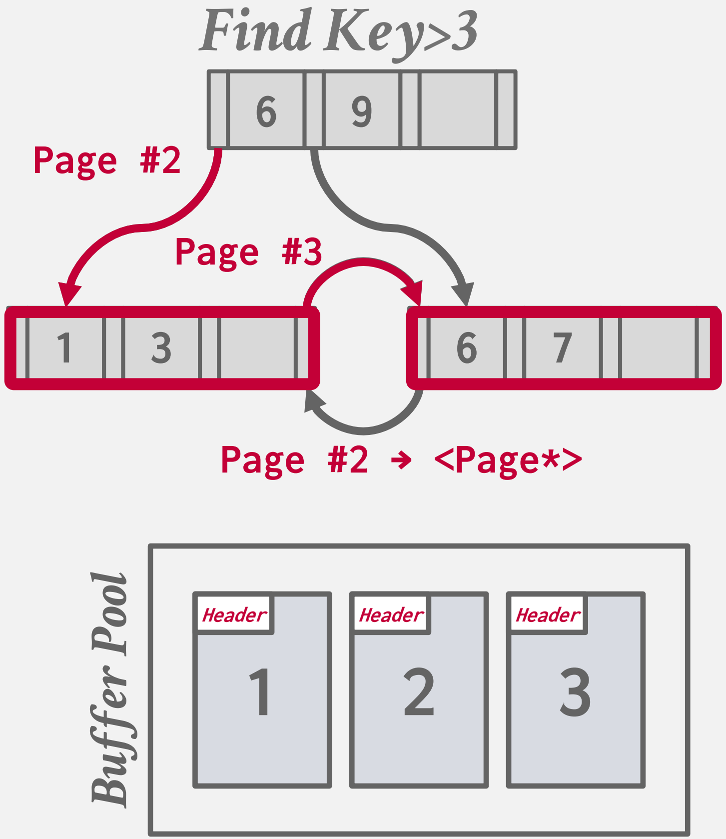
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



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.

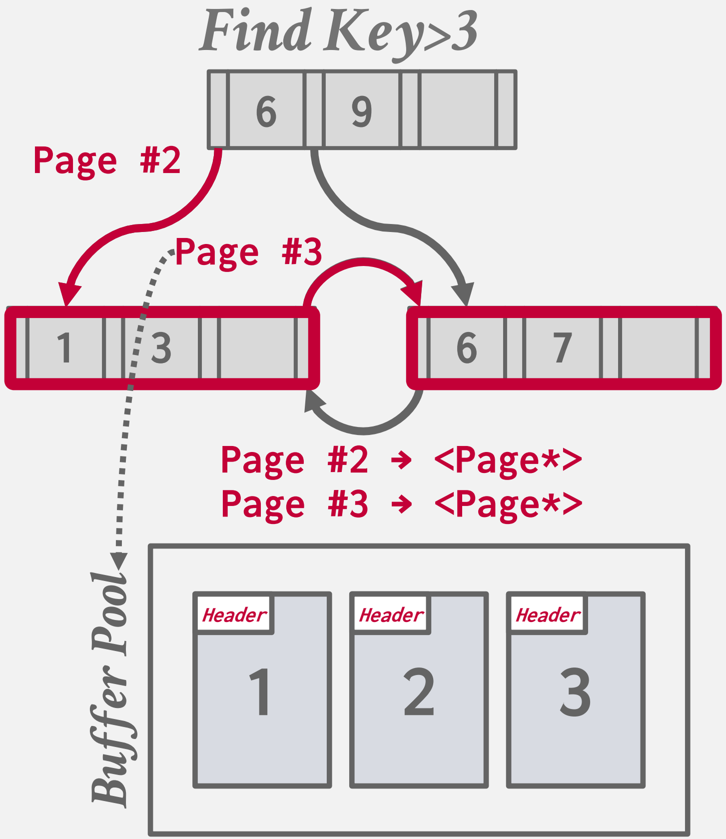
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



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.

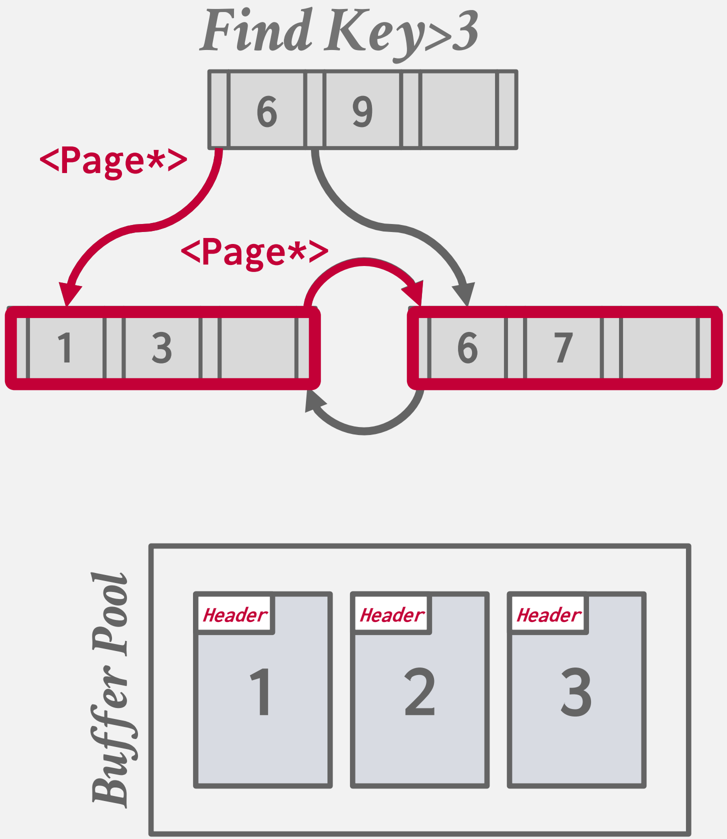
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



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.

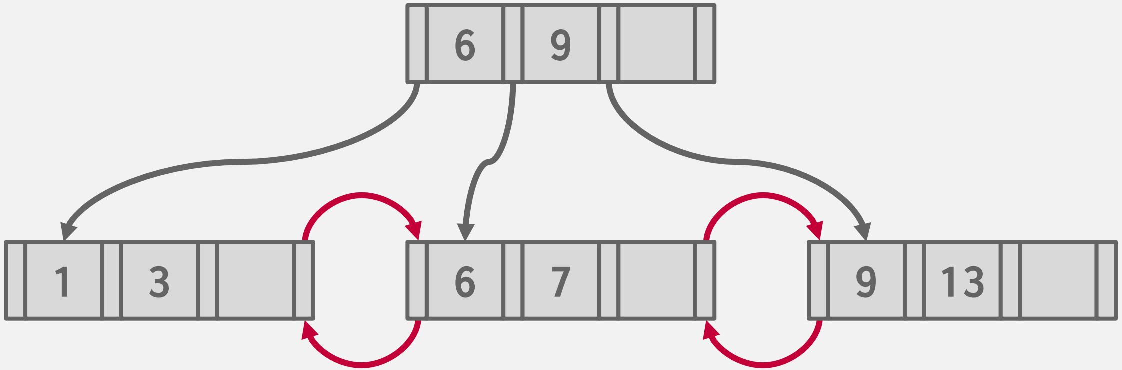


BULK INSERT

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.

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.

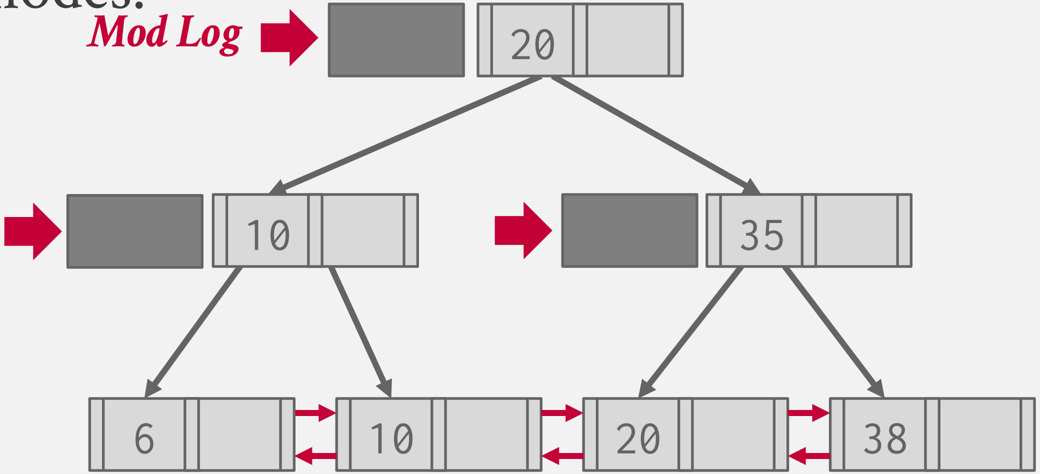
What if there was a way to delay updates and then apply multiple changes together in a batch?

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

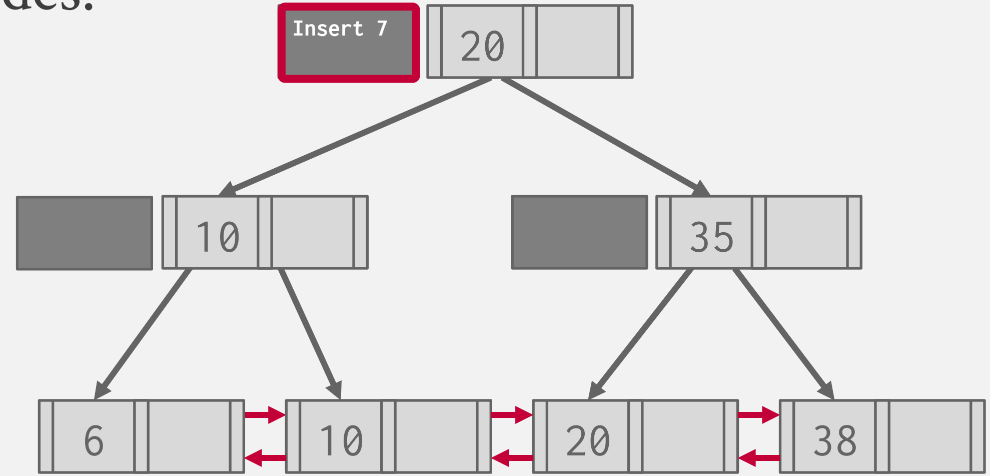
WRITE-OPTIMIZED B+TREE

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.

Insert 7



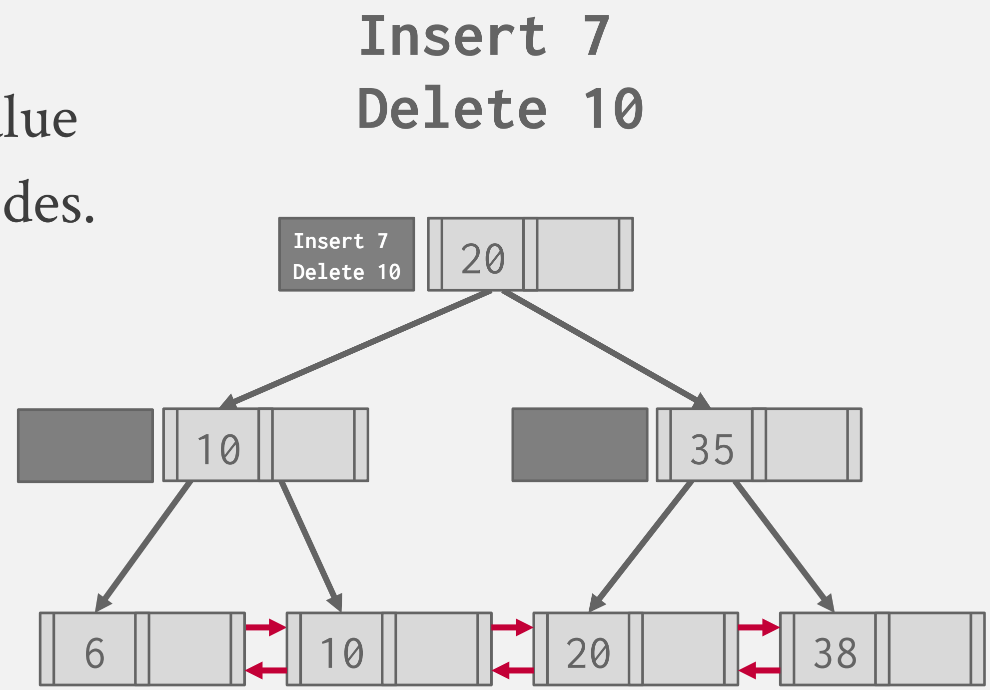
Tokutek SPLINTERDB RelationalAI

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.



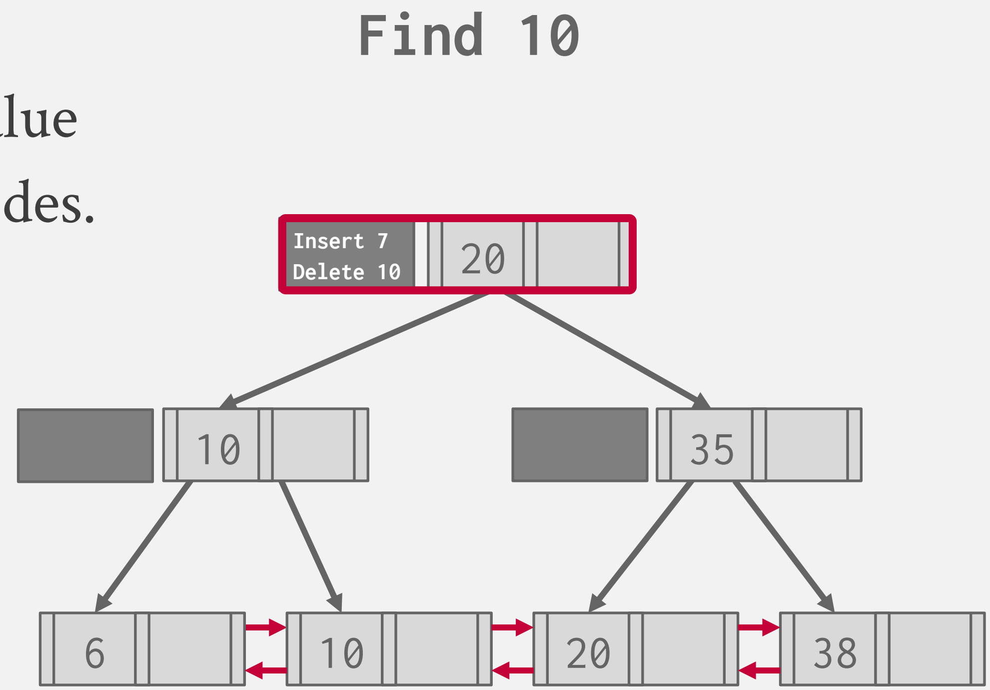
Tokutek SPLINTERDB RelationalAI

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.

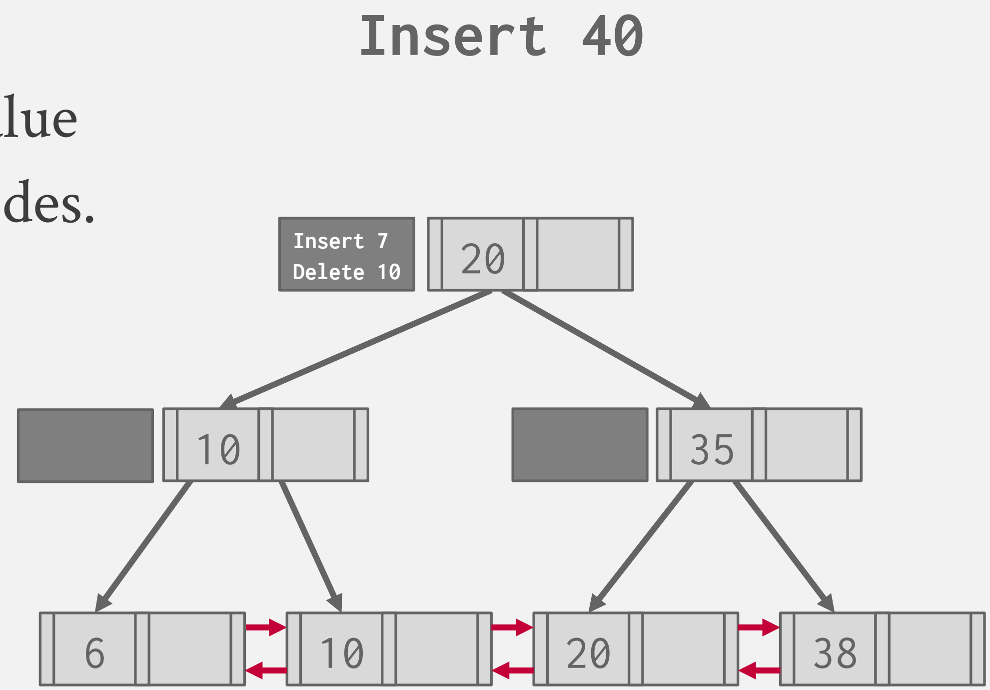


Tokutek SPLINTERDB RelationalAI

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.



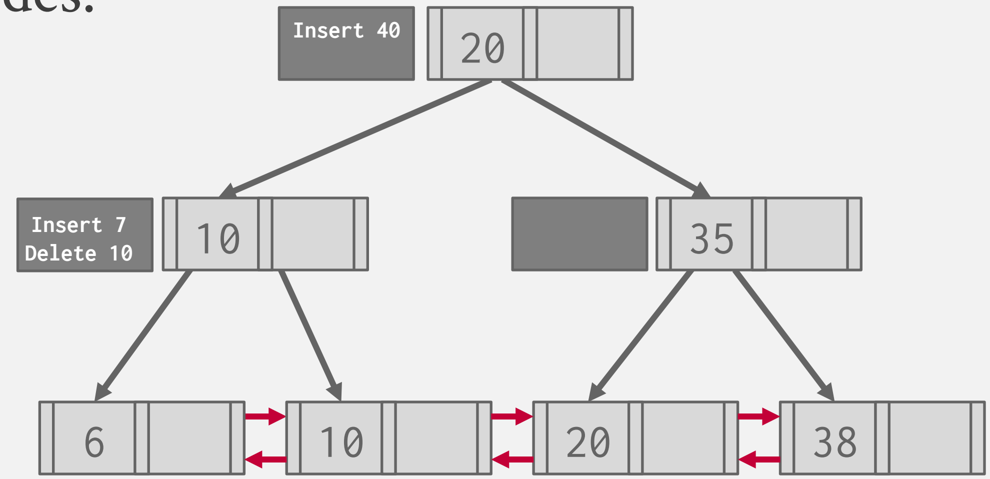
WRITE-OPTIMIZED B+TREE

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.

Insert 40



Tokutek SPLINTERDB RelationalAI

CONCLUSION

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

NEXT CLASS

Index Concurrency Control