

Carnegie Mellon University

# DATABASE SYSTEMS

## B+Trees for Everyone!

LECTURE #08 » 15-445/645 FALL 2025 » PROF. ANDY PAVLO



# ADMINISTRIVIA

---



**Project #1** is due Sunday Sept 28<sup>th</sup> @ 11:59pm

→ See Recitation Video ([@83](#))

→ Office Hours: Saturday Sept 27<sup>th</sup> @ 3:00-5:00pm in GHC 5207

**Homework #3** is due Sunday Oct 5<sup>th</sup> @ 11:59pm

**Mid-Term Exam** is on Wednesday Oct 8<sup>th</sup>

→ Lectures #01–11 (inclusive)

→ Study guide will be released early next week.

# UPCOMING DATABASE TALKS

---



## Apache Iceberg (DB Seminar)

→ Monday Sep 22<sup>nd</sup> @ 4:30pm ET

→ Zoom



## Apache Hudi (DB Seminar)

→ Monday Sep 29<sup>th</sup> @ 4:30pm ET

→ Zoom



## MotherDuck (DB Seminar)

→ Monday Oct 6<sup>th</sup> @ 4:30pm ET

→ Zoom



# ERRATA

---



Course Staff Feedback

Correction on OLAP name origin

Cours  
Corre



@SplitBrainDeezNutz 1 hour ago

i heard tje dj cache's radio show last week and he was ripping it  
that boy needs to be featuered more he gets what wants and  
that is money for him and not for andy cuz dj cache needs  
money



Reply



@SplitBrainDeezNutz 1 hour ago

i heard tje dj cache's radio  
that



@Sweaty-the-Glove 5 days ago

Yo DJ Cache has sickest beats he needs to get paid.  
how can I buy his merch?



5



Reply

and he was ripping it  
what wants and  
cache needs



@SplitBrainDeezNutz 1 hour ago

i heard tje dj cache's radio  
that



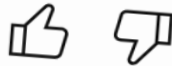
@Sweaty-the-Glove 5 days ago

Yo DJ Cache has sickest beats he needs to get paid.  
how can I buy his merch?



@bwknylfcfjjwij2 3 weeks ago

its still watchable at 1.25x speed. guess he wasn't hyped  
enough yet.



Reply

and he was ripping it  
what wants and  
cache needs



@SplitBrainDeezNutz 1 hour ago

i heard tje dj cache's radio  
that



@Sweaty-the-Glove 5 days ago

Yo DJ Cache has sickest beats he needs to get paid.  
how can I buy his merch?



@bwknylfcfjjwij2 3 weeks ago

its still watchable at 1.25x speed. guess he  
enough yet

asn't hyped



@williamguru 10 days ago

You can either be a nerd or you can be a gangster, but you can't  
be both. 😂



Reply

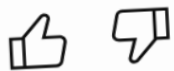




**@Sweaty-the-Glove**  
Yo DJ Cache has s  
how can I buy his r



**@williamguru** 10  
You can either b  
be both. 😂



Rep



**@Milhouse77BS** 3 days ago

**13:00** AFAIK, E. F. Codd was the one who got in trouble with getting paid for a paper on OLAP. From wikipedia: "However, the term did not appear until 1993 when it was coined by Edgar F. Codd, who has been described as "the father of the relational database". Codd's paper[1] resulted from a short consulting assignment which Codd undertook for former Arbor Software (later Hyperion Solutions, and in 2007 acquired by Oracle), as a sort of marketing coup.

The company had released its own OLAP product, Essbase, a year earlier. As a result, Codd's "twelve laws of online analytical processing" were explicit in their reference to Essbase. There was some ensuing controversy and when Computerworld learned that Codd was paid by Arbor, it retracted the article."

Show less



Reply



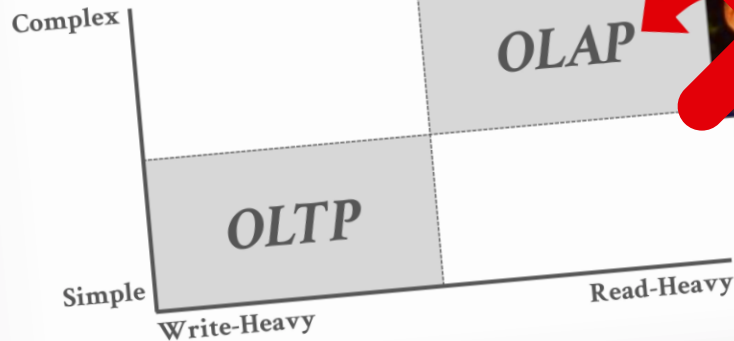
@Sweaty-the-Glove  
Yo DJ Cache has s  
how can I buy his r



@Milhouse77BS 3 days ago

13:00 AFAIK, E. F. Codd was the one who got in trouble with getting paid for a paper on OLAP. From wikipedia: "However, the term did not appear until 1993 when it was coined by Edgar F. Codd, who has been described as "the father of the relational database". Codd's paper[1] resulted from a short consulting assignment which Codd undertook for former Arbor Software (Hyperion Solutions, and in 2007 acquired by Oracle), as a marketing coup.

## DATABASE WORKLOADS



Jim Gray

Source: [Mike Stonebraker](#)

Company had released its own OLAP product, Essbase, a tier. As a result, Codd's "twelve laws of online analytical ng" were explicit in their reference to Essbase. There e ensuing controversy and when Computerworld at Codd was paid by Arbor, it retracted the article."

Reply

# 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.

# EXTENDIBLE HASHING

---



Chained-hashing approach that splits buckets incrementally instead of letting the linked list grow forever.

Multiple slot locations can point to the same bucket chain.

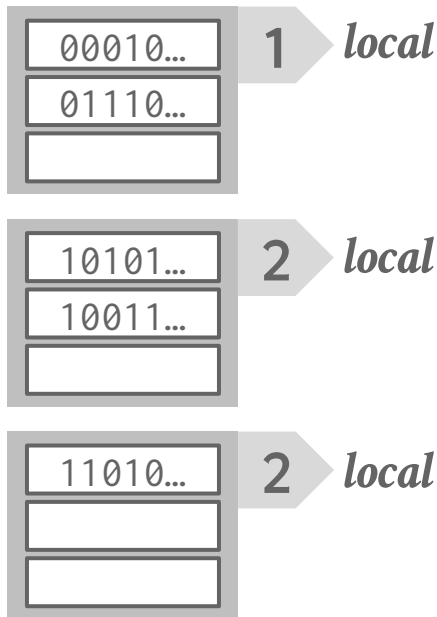
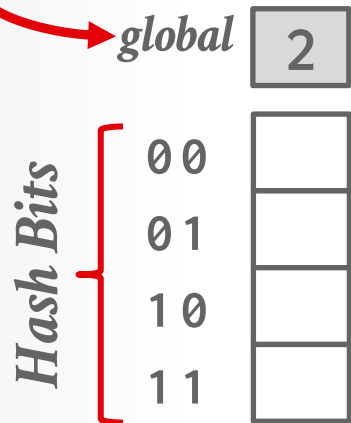
Reshuffle bucket entries on split and increase the number of bits to examine.

→ Data movement is localized to just the split chain.

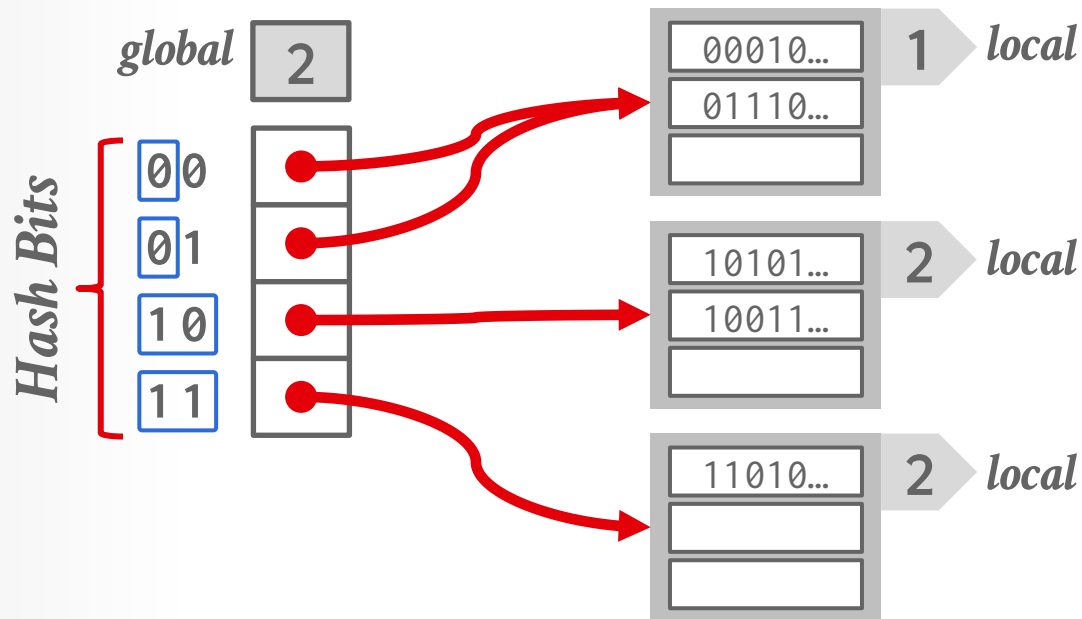


# EXTENDIBLE HASHING

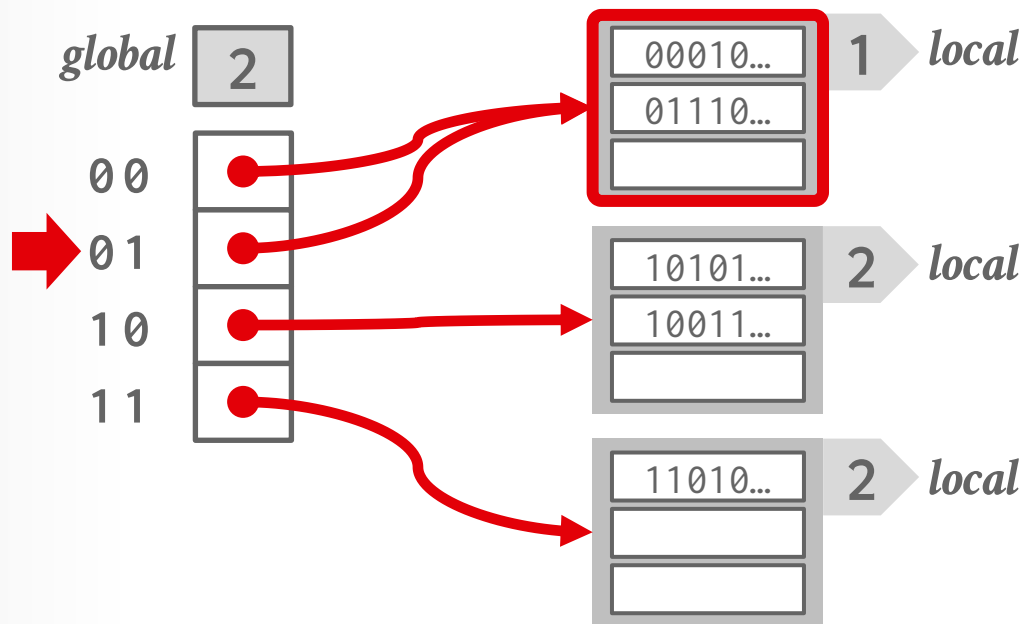
*Max number of bits to examine in hashes*



# EXTENDIBLE HASHING

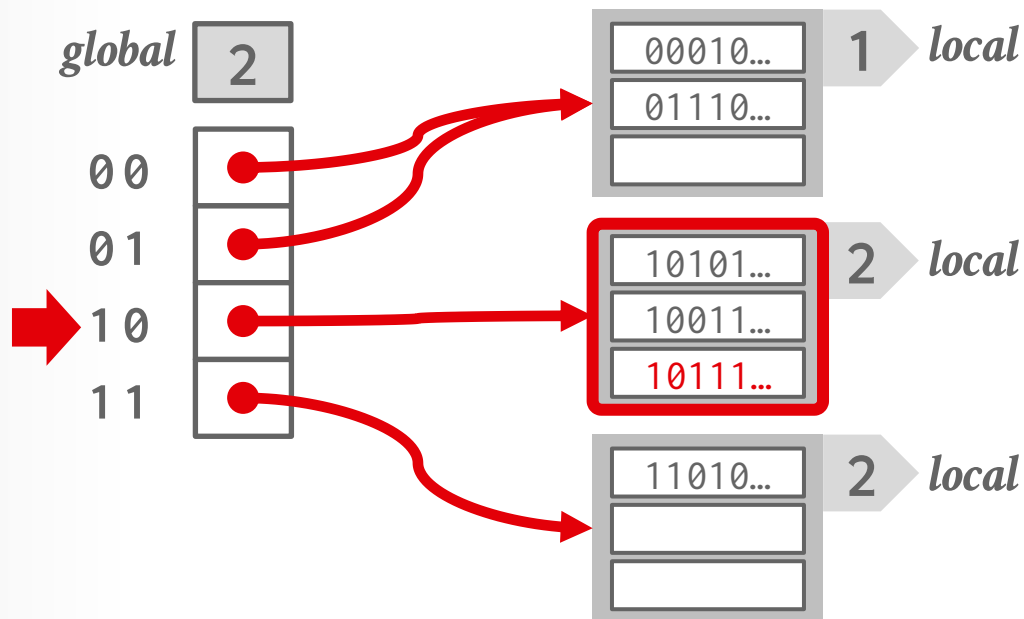


# EXTENDIBLE HASHING



Get A  
*hash*(A) = 01110...

# EXTENDIBLE HASHING



Get A

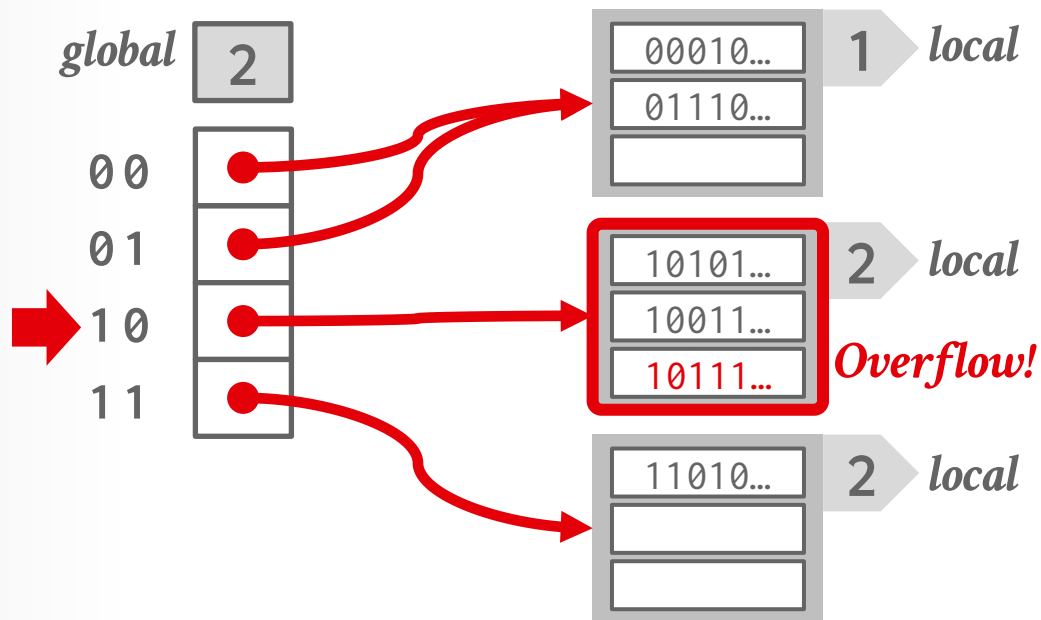
$hash(A) = 01110...$

Put B

$hash(B) = 10111...$



# EXTENDIBLE HASHING



Get A

$hash(A) = 01110...$

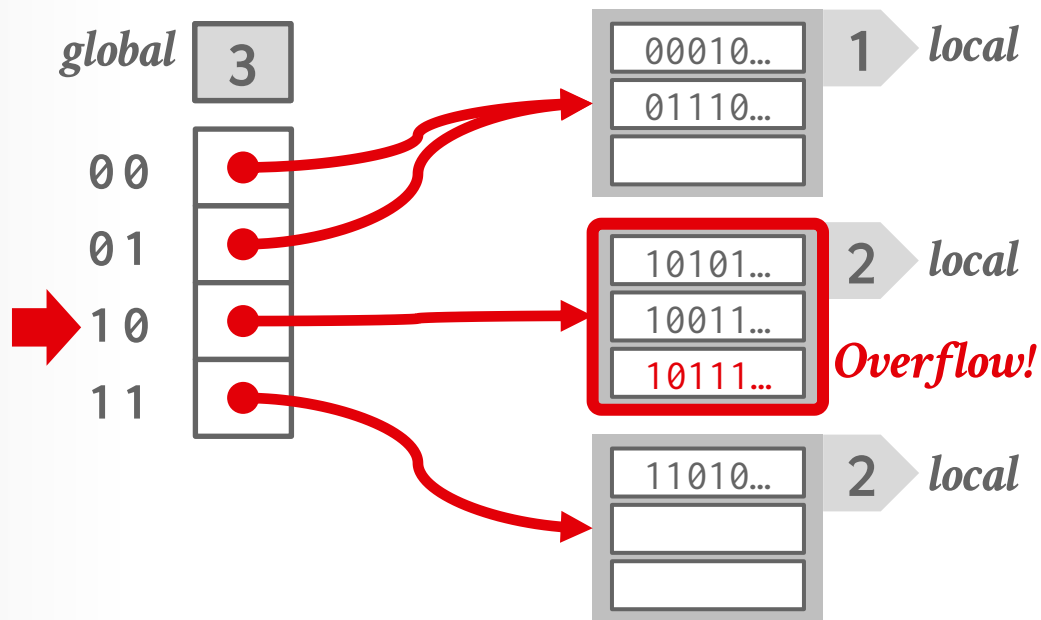
Put B

$hash(B) = 10111...$

Put C

$hash(C) = 10100...$

# EXTENDIBLE HASHING



Get A

$hash(A) = 01110...$

Put B

$hash(B) = 10111...$

Put C

$hash(C) = 10100...$

# EXTENDIBLE HASHING

global **3**

|     |  |
|-----|--|
| 000 |  |
| 010 |  |
| 100 |  |
| 110 |  |
| 001 |  |
| 011 |  |
| 101 |  |
| 111 |  |

|          |   |
|----------|---|
| 00010... | 1 |
| 01110... |   |
|          |   |

|          |   |
|----------|---|
|          | 3 |
| 10011... |   |
|          |   |

|          |   |
|----------|---|
| 10101... | 3 |
|          |   |
| 10111... |   |

|          |   |
|----------|---|
| 11010... | 2 |
|          |   |
|          |   |

Get A

*hash(A) = 01110...*

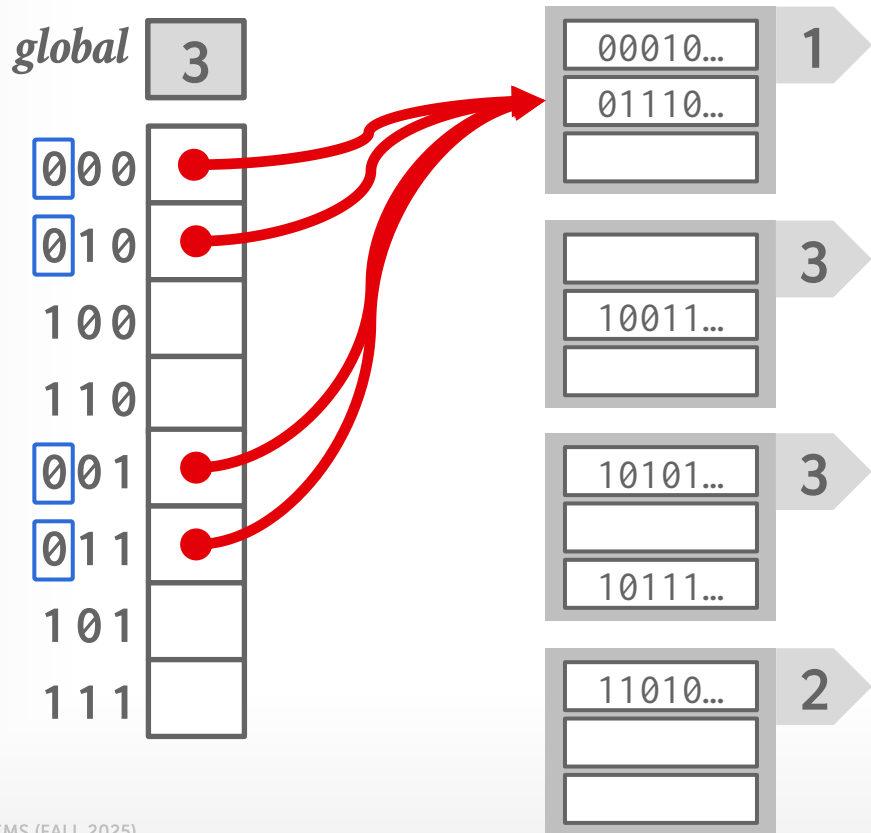
Put B

*hash(B) = 10111...*

Put C

*hash(C) = 10100...*

# EXTENDIBLE HASHING



Get A

$hash(A) = 01110...$

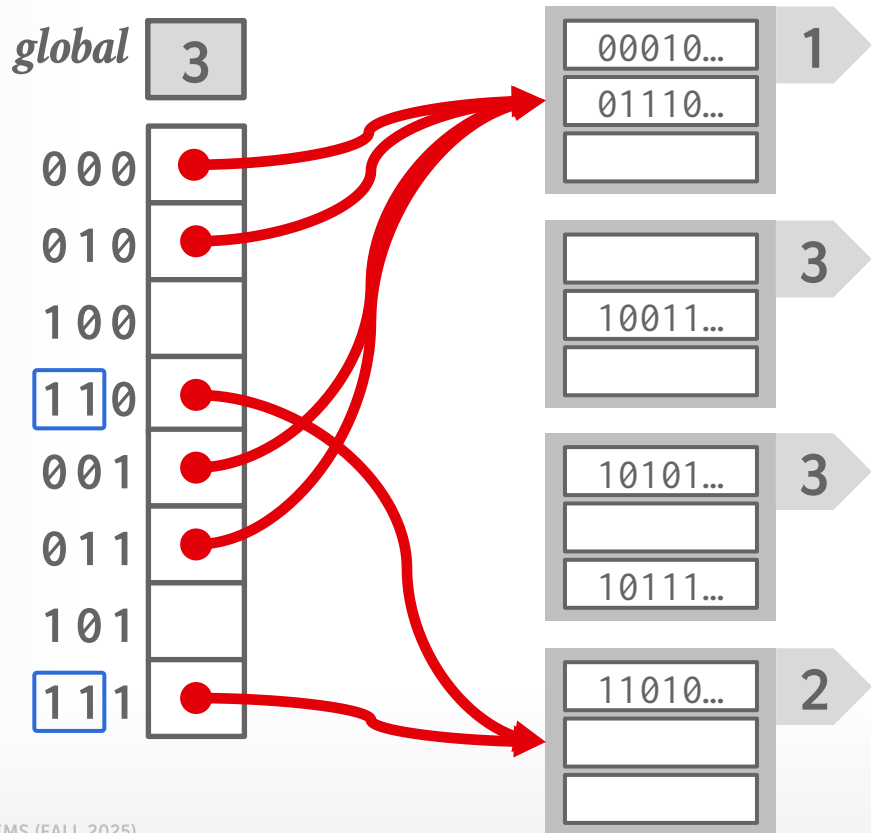
Put B

$hash(B) = 10111...$

Put C

$hash(C) = 10100...$

# EXTENDIBLE HASHING



Get A

*hash(A) = 01110...*

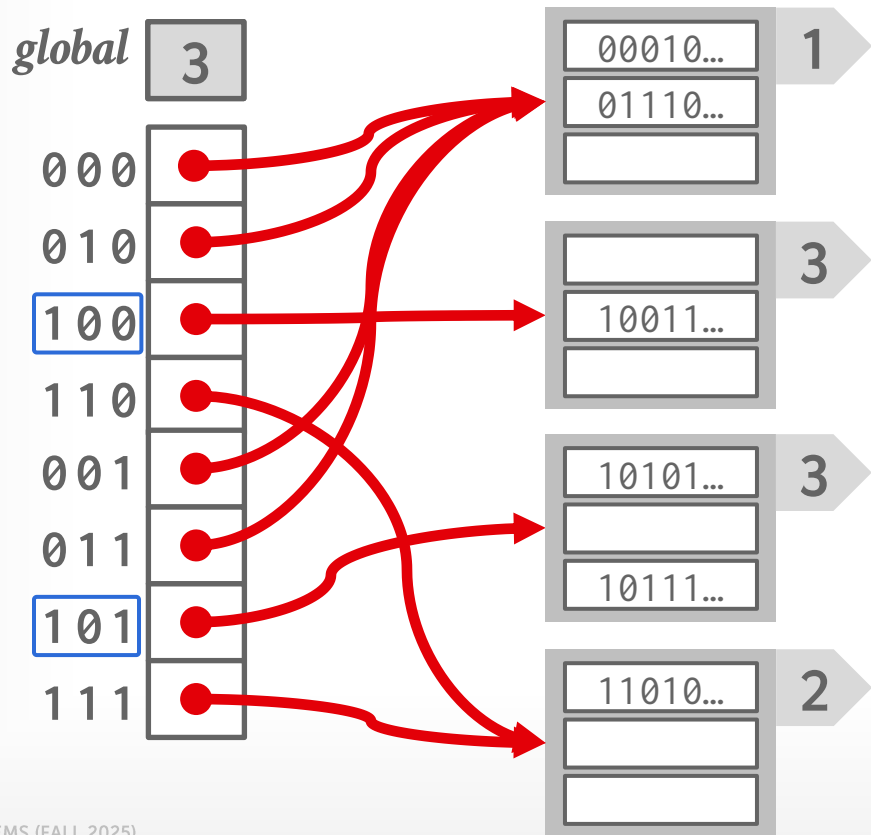
Put B

*hash(B) = 10111...*

Put C

*hash(C) = 10100...*

# EXTENDIBLE HASHING



Get A

*hash(A) = 01110...*

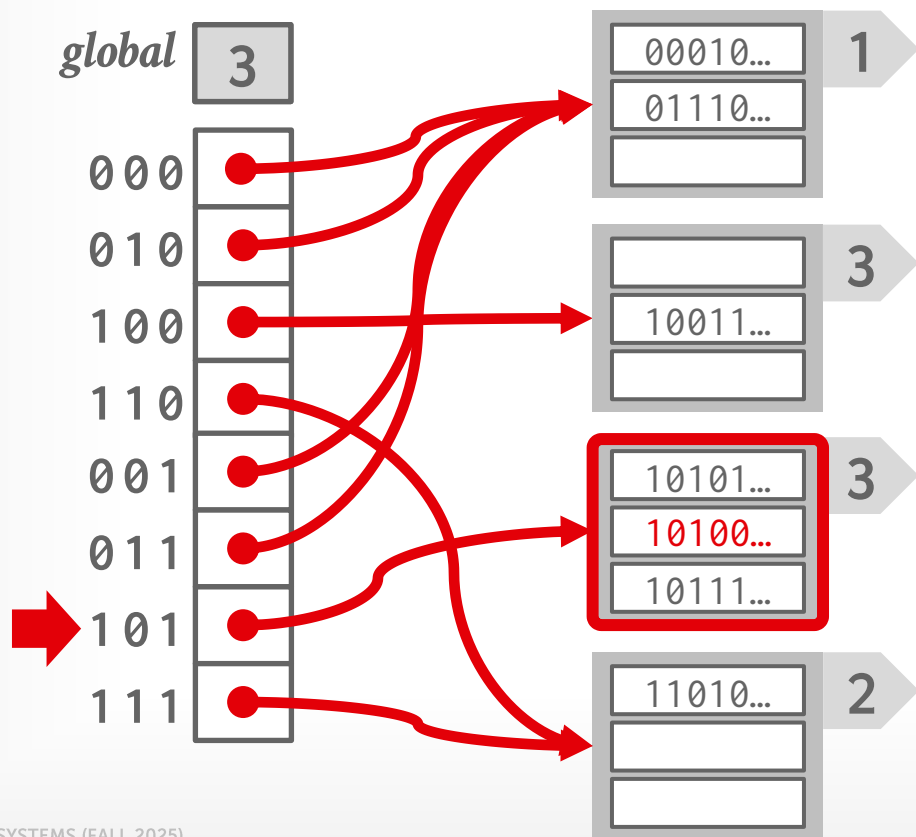
Put B

*hash(B) = 10111...*

Put C

*hash(C) = 10100...*

# EXTENDIBLE HASHING



Get A

$hash(A) = 01110...$

Put B

$hash(B) = 10111...$

Put C

$hash(C) = 10100...$

# LINEAR HASHING

---

The hash table maintains a pointer that tracks the next bucket to split.

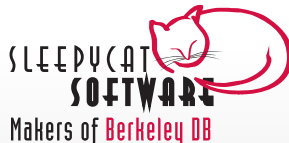
→ When any bucket overflows, split the bucket at the pointer location.

Use multiple hashes to find the right bucket for a given key.

Can use different overflow criterion:

→ Space Utilization

→ Average Length of Overflow Chains





# LINEAR HASHING

*Split  
Pointer*



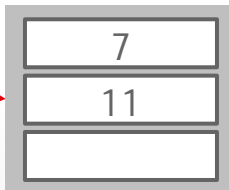
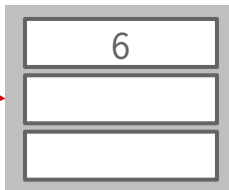
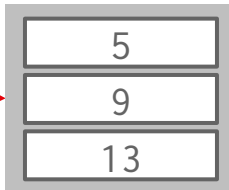
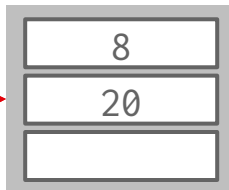
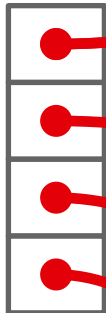
*Bucket  
Pointers*

0

1

2

3

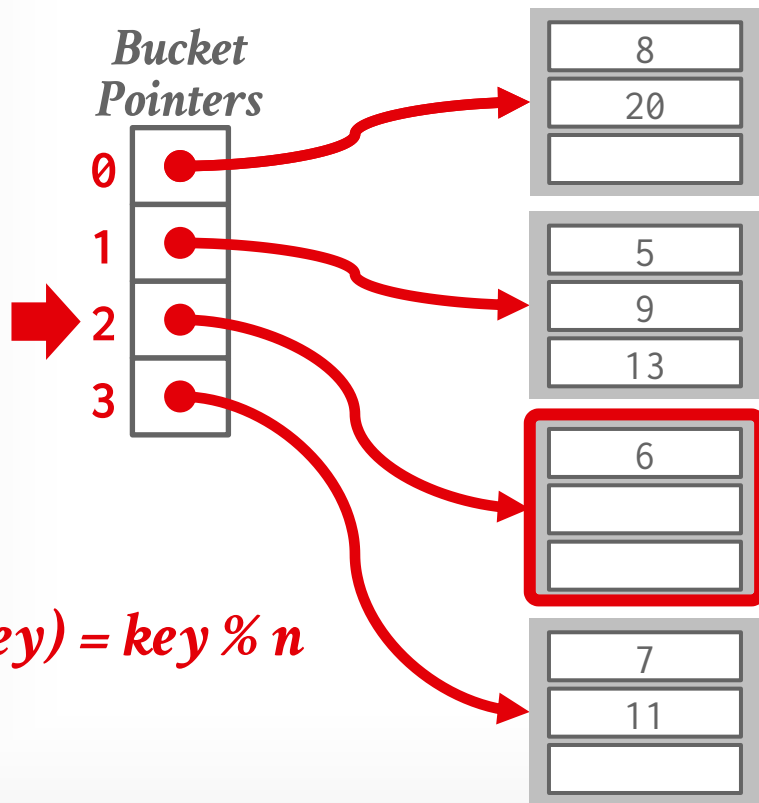


$$\text{hash}_1(\text{key}) = \text{key} \% n$$

# LINEAR HASHING



Split  
Pointer



$$hash_1(key) = key \% n$$

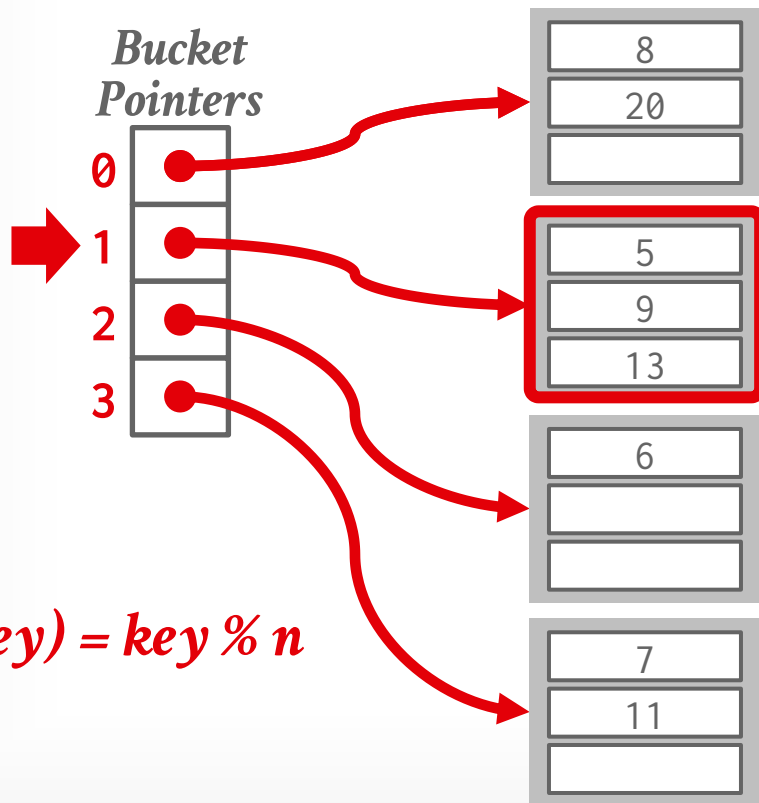
Get 6

$$hash_1(6) = 6 \% 4 = 2$$

# LINEAR HASHING



Split  
Pointer



$$hash_1(key) = key \% n$$

Get 6

$$hash_1(6) = 6 \% 4 = 2$$

Put 17

$$hash_1(17) = 17 \% 4 = 1$$



# LINEAR HASHING

Split  
Pointer

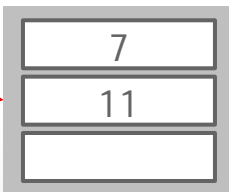
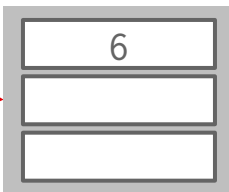
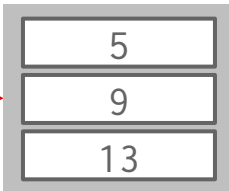
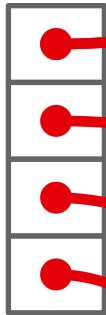
Bucket  
Pointers

0

1

2

3



*Overflow!*

Get 6

$$hash_1(6) = 6 \% 4 = 2$$

Put 17

$$hash_1(17) = 17 \% 4 = 1$$

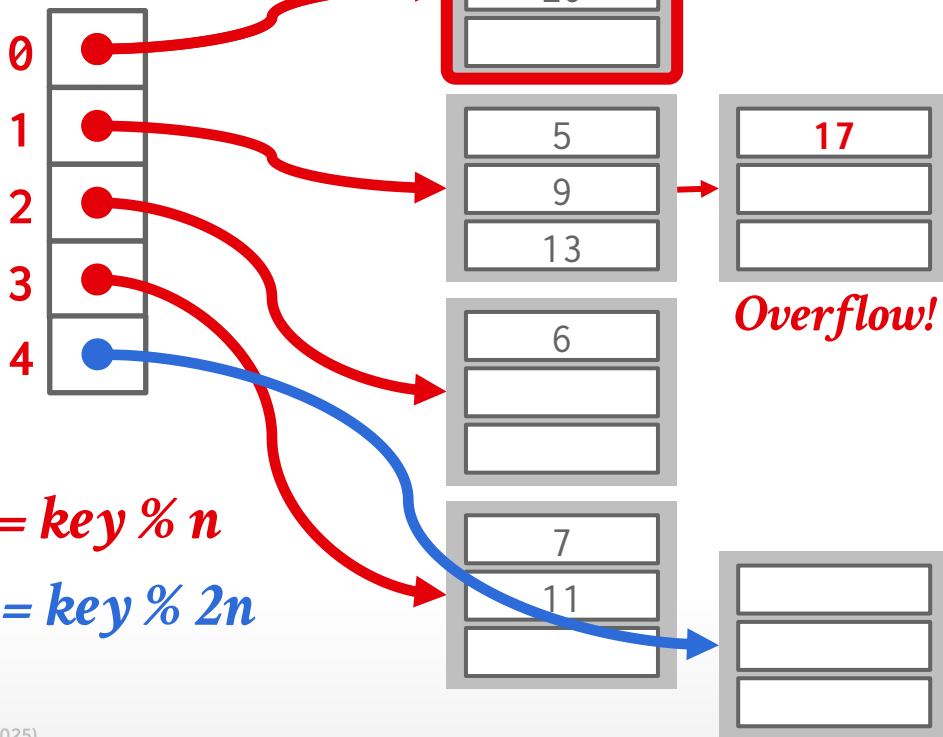
$$hash_1(key) = key \% n$$

# LINEAR HASHING



Split  
Pointer  
➔

Bucket  
Pointers



Get 6

$$\text{hash}_1(6) = 6 \% 4 = 2$$

Put 17

$$\text{hash}_1(17) = 17 \% 4 = 1$$

$$\text{hash}_1(\text{key}) = \text{key} \% n$$

$$\text{hash}_2(\text{key}) = \text{key} \% 2n$$

# LINEAR HASHING



Split  
Pointer

Bucket  
Pointers

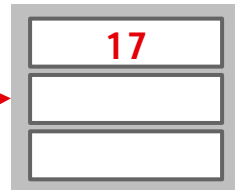
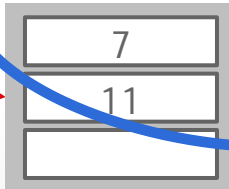
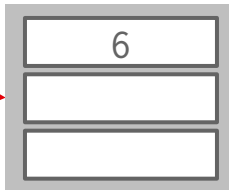
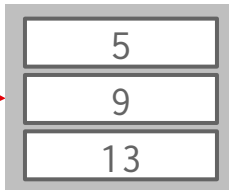
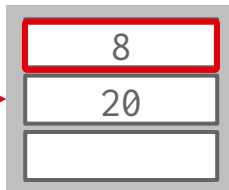
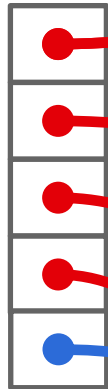
0

1

2

3

4



Get 6

$$hash_1(6) = 6 \% 4 = 2$$

Put 17

$$hash_1(17) = 17 \% 4 = 1$$

$$hash_2(8) = 8 \% 8 = 0$$

$$hash_1(key) = key \% n$$

$$hash_2(key) = key \% 2n$$

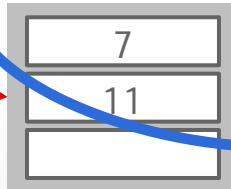
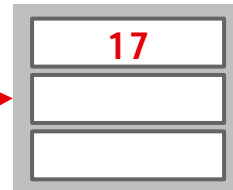
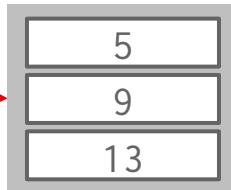
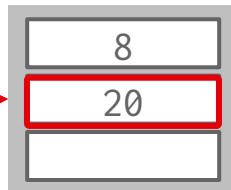
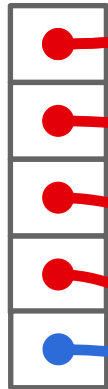
# LINEAR HASHING



Split  
Pointer  
➔

Bucket  
Pointers

0  
1  
2  
3  
4



Get 6

$$hash_1(6) = 6 \% 4 = 2$$

Put 17

$$hash_1(17) = 17 \% 4 = 1$$

$$hash_2(8) = 8 \% 8 = 0$$

$$hash_2(20) = 20 \% 8 = 4$$

$$hash_1(key) = key \% n$$

$$hash_2(key) = key \% 2n$$

# LINEAR HASHING



Split  
Pointer



Bucket  
Pointers

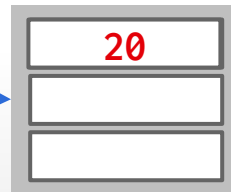
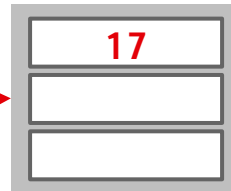
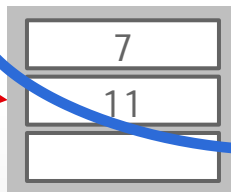
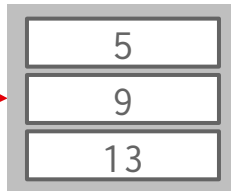
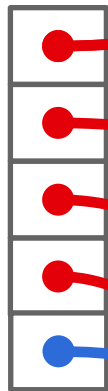
0

1

2

3

4



Get 6

$$\text{hash}_1(6) = 6 \% 4 = 2$$

Put 17

$$\text{hash}_1(17) = 17 \% 4 = 1$$

$$\text{hash}_2(8) = 8 \% 8 = 0$$

$$\text{hash}_2(20) = 20 \% 8 = 4$$

$$\text{hash}_1(\text{key}) = \text{key} \% n$$

$$\text{hash}_2(\text{key}) = \text{key} \% 2n$$



# LINEAR HASHING



Split  
Pointer



Bucket  
Pointers

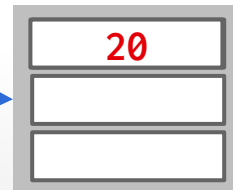
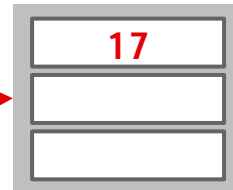
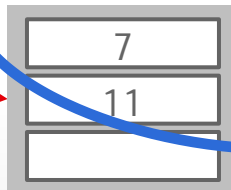
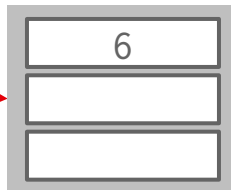
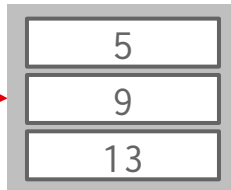
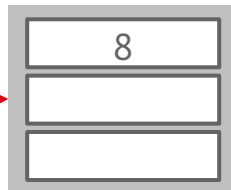
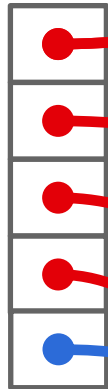
0

1

2

3

4



Get 6

$$\text{hash}_1(6) = 6 \% 4 = 2$$

Put 17

$$\text{hash}_1(17) = 17 \% 4 = 1$$

$$\text{hash}_2(8) = 8 \% 8 = 0$$

$$\text{hash}_2(20) = 20 \% 8 = 4$$

$$\text{hash}_1(\text{key}) = \text{key} \% n$$

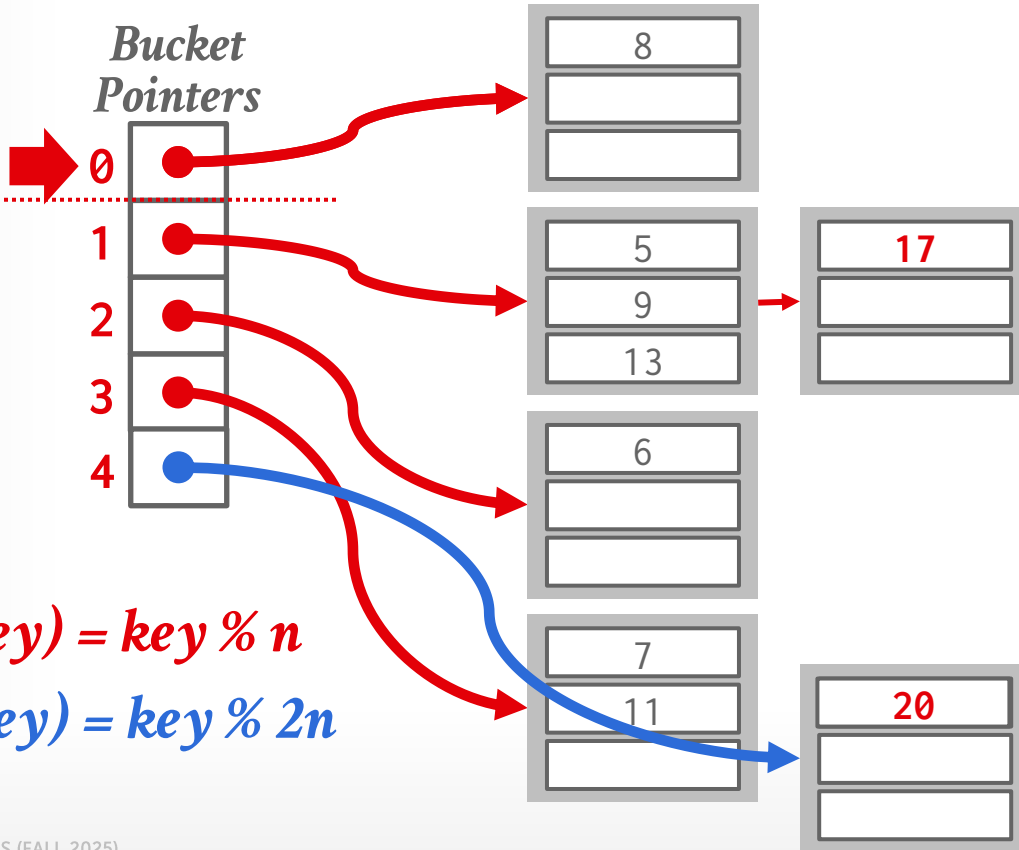
$$\text{hash}_2(\text{key}) = \text{key} \% 2n$$

# LINEAR HASHING



Split  
Pointer

Bucket  
Pointers



$$hash_1(key) = key \% n$$

$$hash_2(key) = key \% 2n$$

Get 6

$$hash_1(6) = 6 \% 4 = 2$$

Put 17

$$hash_1(17) = 17 \% 4 = 1$$

$$hash_2(8) = 8 \% 8 = 0$$

$$hash_2(20) = 20 \% 8 = 4$$

Get 20

$$hash_1(20) = 20 \% 4 = 0$$

# LINEAR HASHING



Split  
Pointer

Bucket  
Pointers

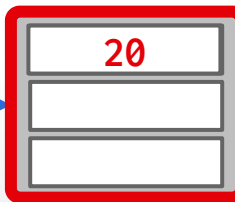
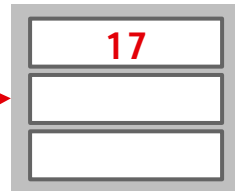
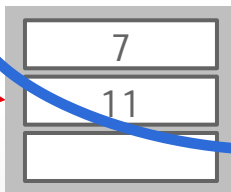
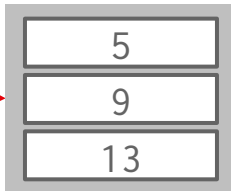
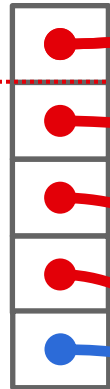
0

1

2

3

4



$$hash_1(key) = key \% n$$

$$hash_2(key) = key \% 2n$$

Get 6

$$hash_1(6) = 6 \% 4 = 2$$

Put 17

$$hash_1(17) = 17 \% 4 = 1$$

$$hash_2(8) = 8 \% 8 = 0$$

$$hash_2(20) = 20 \% 8 = 4$$

Get 20

$$hash_1(20) = 20 \% 4 = 0$$

$$hash_2(20) = 20 \% 8 = 4$$

# LINEAR HASHING



Split  
Pointer

Bucket  
Pointers

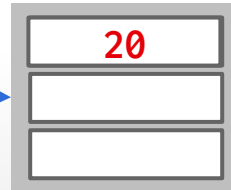
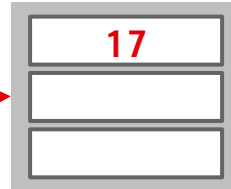
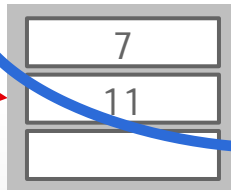
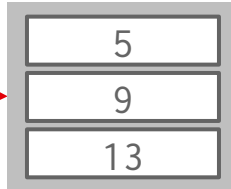
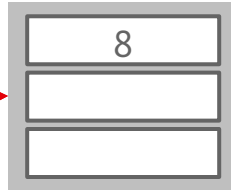
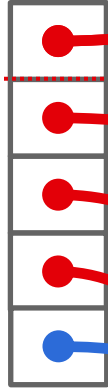
0

1

2

3

4



$$\text{hash}_1(\text{key}) = \text{key} \% n$$

$$\text{hash}_2(\text{key}) = \text{key} \% 2n$$

Get 6

$$\text{hash}_1(6) = 6 \% 4 = 2$$

Put 17

$$\text{hash}_1(17) = 17 \% 4 = 1$$

$$\text{hash}_2(8) = 8 \% 8 = 0$$

$$\text{hash}_2(20) = 20 \% 8 = 4$$

Get 20

$$\text{hash}_1(20) = 20 \% 4 = 0$$

$$\text{hash}_2(20) = 20 \% 8 = 4$$

Get 9

$$\text{hash}_1(9) = 9 \% 4 = 1$$

# LINEAR HASHING



Split  
Pointer

Bucket  
Pointers

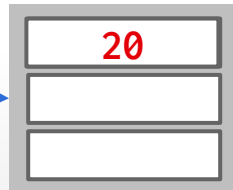
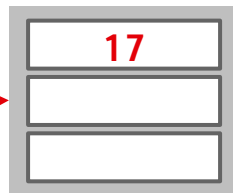
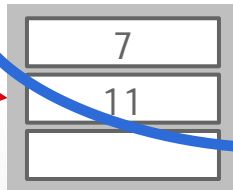
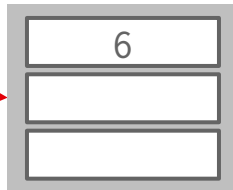
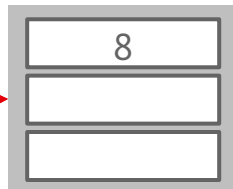
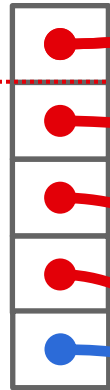
0

1

2

3

4



$$hash_1(key) = key \% n$$

$$hash_2(key) = key \% 2n$$

Get 6

$$hash_1(6) = 6 \% 4 = 2$$

Put 17

$$hash_1(17) = 17 \% 4 = 1$$

$$hash_2(8) = 8 \% 8 = 0$$

$$hash_2(20) = 20 \% 8 = 4$$

Get 20

$$hash_1(20) = 20 \% 4 = 0$$

$$hash_2(20) = 20 \% 8 = 4$$

Get 9

$$hash_1(9) = 9 \% 4 = 1$$

# LINEAR HASHING: RESIZING

---

Splitting buckets based on the split pointer will eventually get to all overflowed buckets.

→ When the pointer reaches the last slot, remove the first hash function and move pointer back to beginning.

If the "highest" bucket below the split pointer is empty, the hash table could remove it and move the splinter pointer in reverse direction.

# LINEAR HASHING: DELETES

Split  
Pointer

Bucket  
Pointers

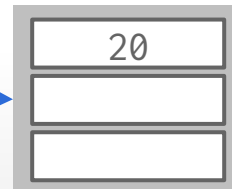
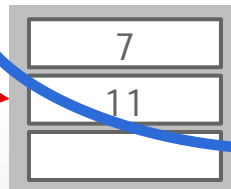
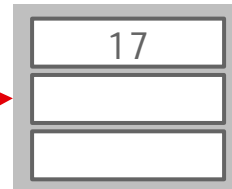
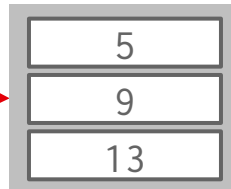
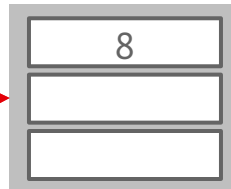
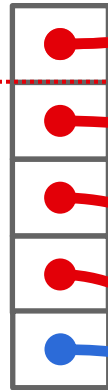
0

1

2

3

4



Delete 20

$$\text{hash}_1(20) = 20 \% 4 = 0$$

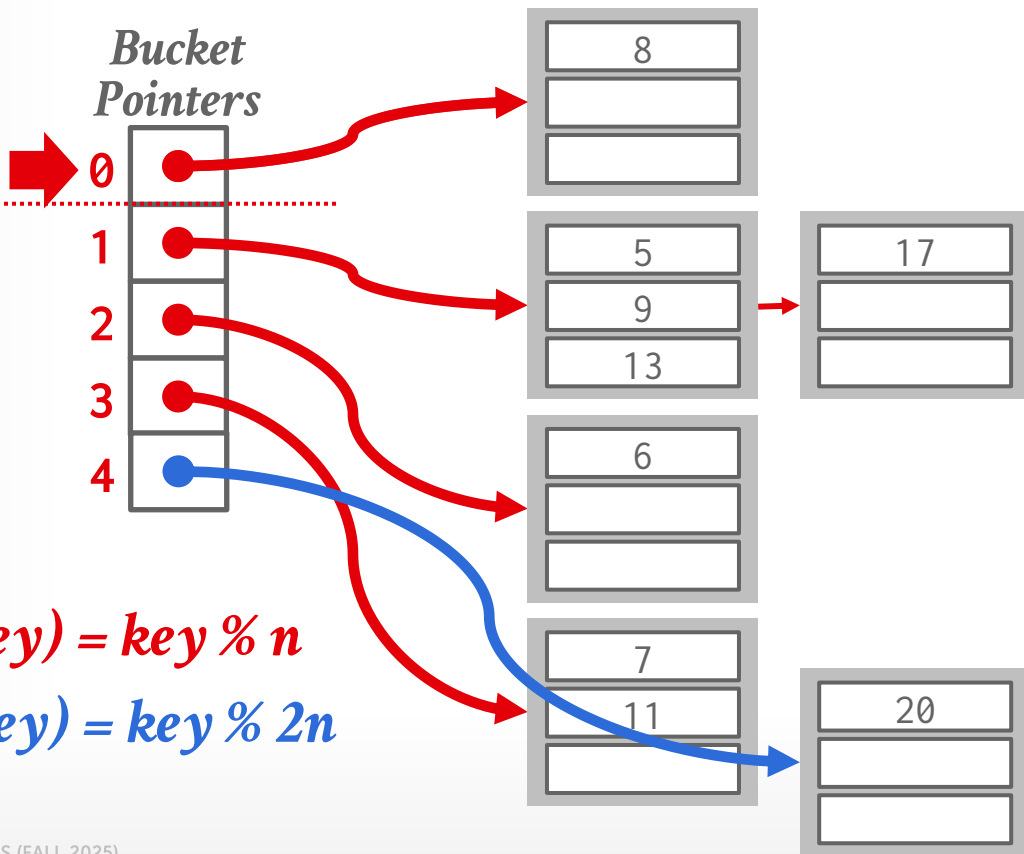
$$\text{hash}_1(\text{key}) = \text{key} \% n$$

$$\text{hash}_2(\text{key}) = \text{key} \% 2n$$

# LINEAR HASHING: DELETES

Split  
Pointer

Bucket  
Pointers



Delete 20

$$hash_1(20) = 20 \% 4 = 0$$

$$hash_1(key) = key \% n$$

$$hash_2(key) = key \% 2n$$



# LINEAR HASHING: DELETES

Split  
Pointer

Bucket  
Pointers

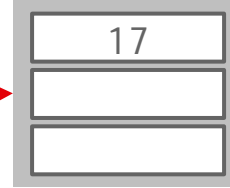
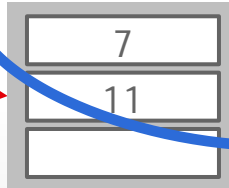
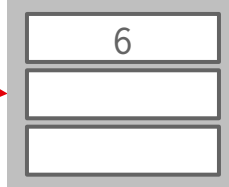
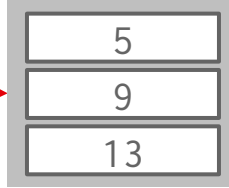
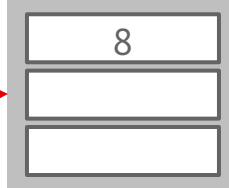
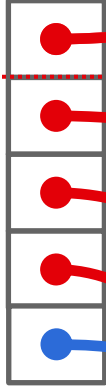
0

1

2

3

4



Delete 20

$$\text{hash}_1(20) = 20 \% 4 = 0$$

$$\text{hash}_2(20) = 20 \% 8 = 4$$

$$\text{hash}_1(\text{key}) = \text{key} \% n$$

$$\text{hash}_2(\text{key}) = \text{key} \% 2n$$

# LINEAR HASHING: DELETES

Split  
Pointer

Bucket  
Pointers

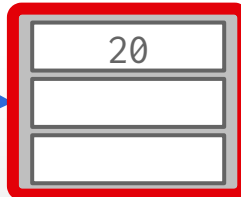
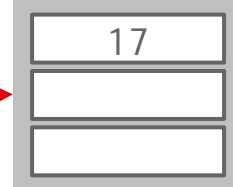
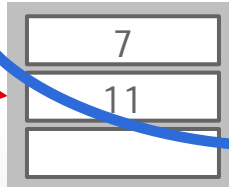
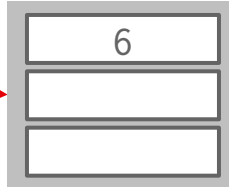
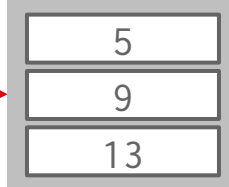
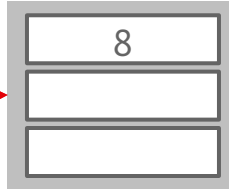
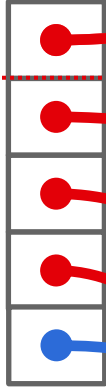
0

1

2

3

4



Delete 20

$$\text{hash}_1(20) = 20 \% 4 = 0$$

$$\text{hash}_2(20) = 20 \% 8 = 4$$

$$\text{hash}_1(\text{key}) = \text{key} \% n$$

$$\text{hash}_2(\text{key}) = \text{key} \% 2n$$

# LINEAR HASHING: DELETES

Split  
Pointer

Bucket  
Pointers

0

1

2

3

4

8

5

9

13

6

7

11

17

Delete 20

$$\text{hash}_1(20) = 20 \% 4 = 0$$

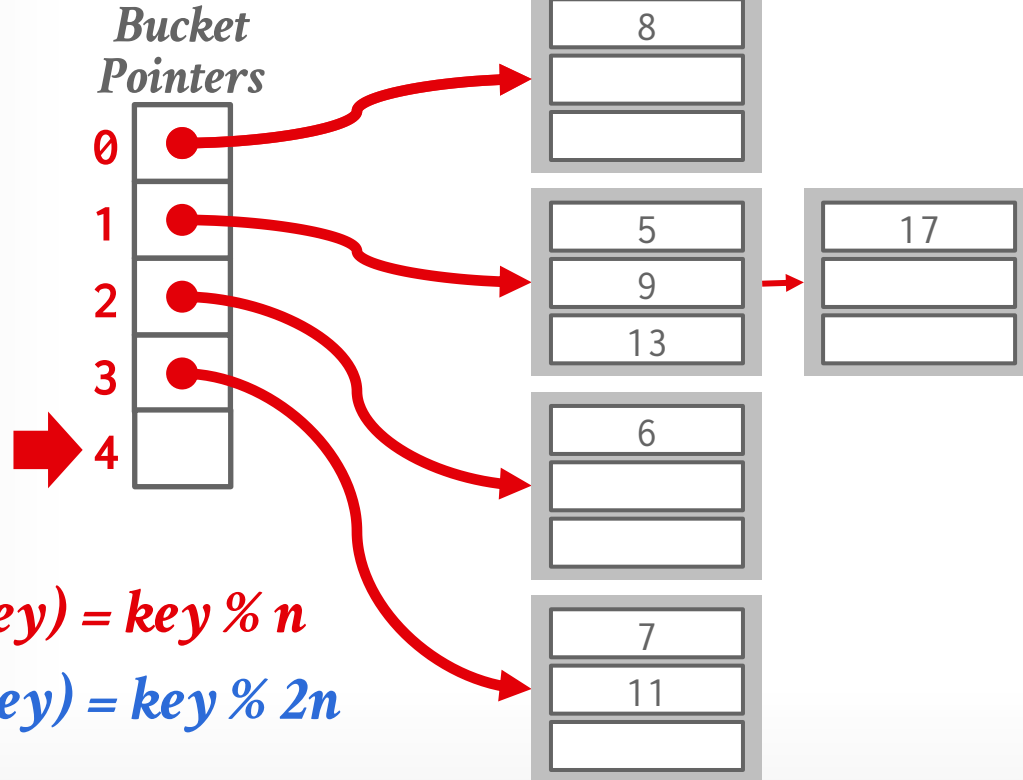
$$\text{hash}_2(20) = 20 \% 8 = 4$$

$$\text{hash}_1(\text{key}) = \text{key} \% n$$

$$\text{hash}_2(\text{key}) = \text{key} \% 2n$$

# LINEAR HASHING: DELETES

Split  
Pointer  
➔



Delete 20

$$\text{hash}_1(20) = 20 \% 4 = 0$$

$$\text{hash}_2(20) = 20 \% 8 = 4$$

$$\text{hash}_1(\text{key}) = \text{key} \% n$$

$$\text{hash}_2(\text{key}) = \text{key} \% 2n$$

# LINEAR HASHING: DELETES

Split  
Pointer



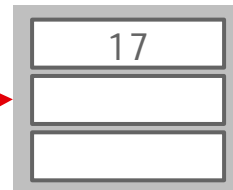
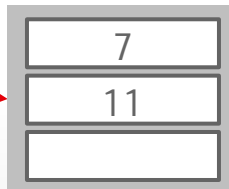
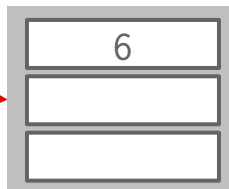
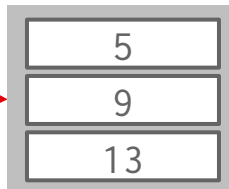
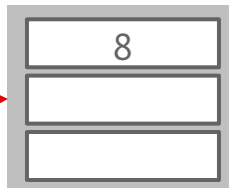
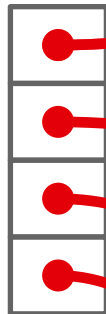
Bucket  
Pointers

0

1

2

3



Delete 20

$$\text{hash}_1(20) = 20 \% 4 = 0$$

$$\text{hash}_2(20) = 20 \% 8 = 4$$

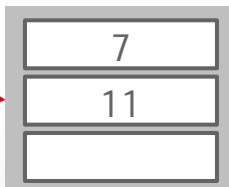
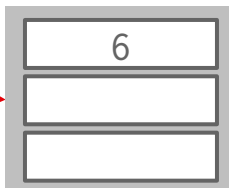
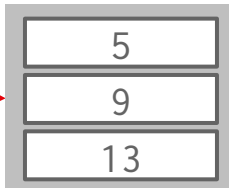
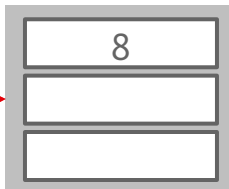
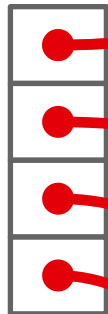
$$\text{hash}_1(\text{key}) = \text{key} \% n$$

# LINEAR HASHING: DELETES

Split  
Pointer  
➔

Bucket  
Pointers

0  
1  
2  
3



Overflow!

Delete 20

$$\text{hash}_1(20) = 20 \% 4 = 0$$

$$\text{hash}_2(20) = 20 \% 8 = 4$$

Put 21

$$\text{hash}_1(21) = 21 \% 4 = 1$$

$$\text{hash}_1(\text{key}) = \text{key} \% n$$

# 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** (1970)
- **B+Tree** (1973)
- **B\*Tree** (1977?)
- **B<sup>link</sup>-Tree** (1981)
- **B&-Tree** (2003)
- **Bw-Tree** (2013)



# B-TREE FA

There is a specific data structure

People also use the term to gener  
balanced tree data structures:

→ **B-Tree** (1970)

→ **B+Tree** (1973)

→ **B\*Tree** (1977?)

→ **B<sup>link</sup>-Tree** (1981)

→ **B $\epsilon$ -Tree** (2003)

→ **Bw-Tree** (2013)

107

D1-82-0989

ORGANIZATION AND MAINTENANCE OF LARGE  
ORDERED INDICES

by

R. Bayer

and

E. McCreight

Mathematical and Information Sciences Report No. 20

Mathematical and Information Sciences Laboratory

BOEING SCIENTIFIC RESEARCH LABORATORIES

July 1970

# B-TREE FA

There is a specific data structure

People also use the term to gener  
balanced tree data structures:

→ **B-Tree** (1970)

→ **B+Tree** (1973)

→ **B\*Tree** (1977?)

→ **B<sup>link</sup>-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-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.  
© 1979 ACM 0010-4892/79/0600-0121 \$00.75

# B-TREE FAMILY

There is a specific data structure called a

People also use the term to generally refer to balanced tree data structures:

- B-Tree (1970)
- B+Tree (1973)
- B\*Tree (1977?)
- **B<sup>link</sup>-Tree (1981)**
- B $\epsilon$ -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, system 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

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

This research was supported by the National Science Foundation under Grant MCS76-16694. 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.

© 1981 ACM 0362-5915/81/1200-0650 \$00.75  
ACM Transactions on Database Systems, Vol. 6, No. 4, December 1981, Pages 650-670.

# B-TREE FAMILY



13

postgres / src / backend / access / nbtree / README

1083 lines (959 loc) · 62.8 KB

Code

Blame

Raw



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

## Efficient Locking for Concurrent Operations on B-Trees

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

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 new locking scheme. 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 requires 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.

Keywords and Phrases: database, data structures, B-tree, index organizations, concurrent algorithms, concurrency controls, locking protocols, correctness, consistency, multiway search trees

Categories: 3.73, 3.74, 4.32, 4.33, 4.34, 5.24

### INTRODUCTION

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 of these structures makes them quite appealing for database applications. One of the topics 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, as defined by Wedekind [15]) especially well suited for use in a concurrent database.

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

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

This work 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.

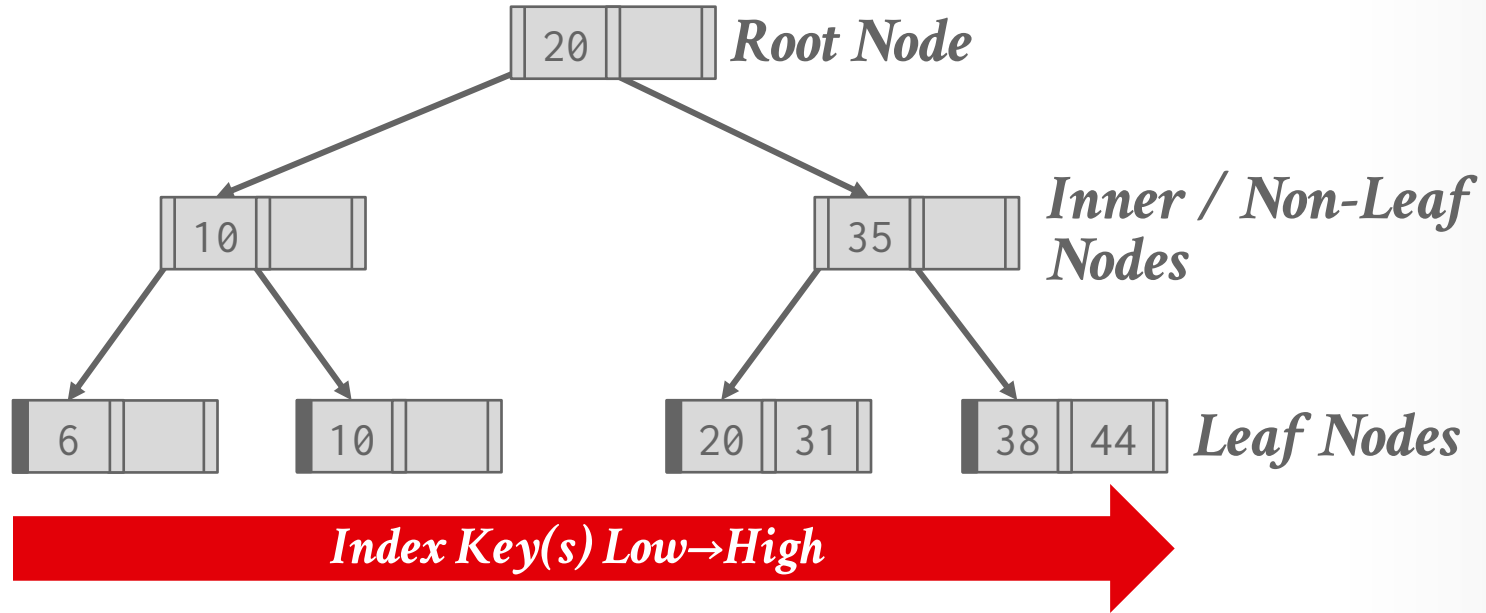
# B+TREE

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

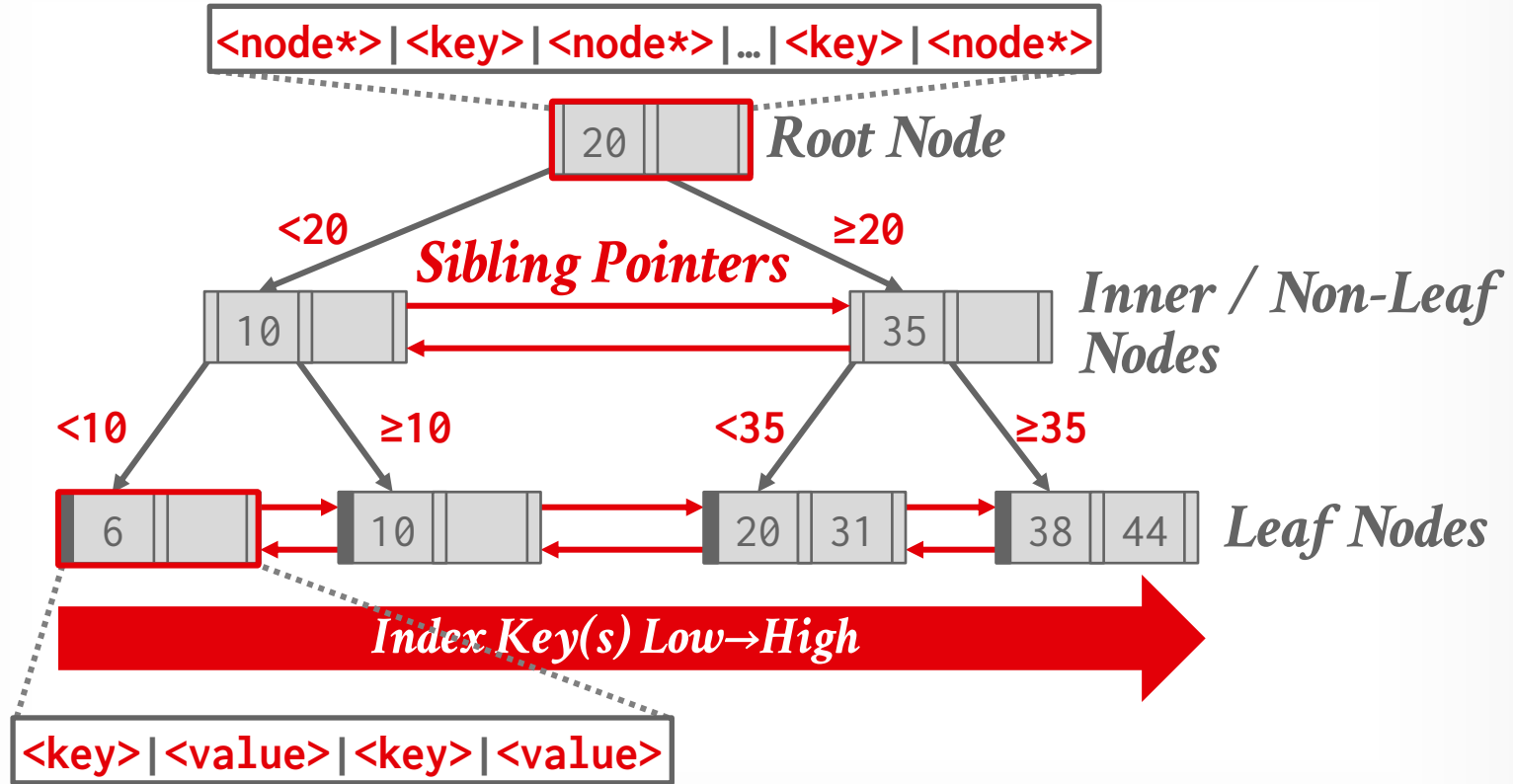
- It is perfectly balanced (i.e., every leaf node is at the same depth in the tree)
- Every node other than the root is at least half-full  
 **$m/2 - 1 \leq \#keys \leq m - 1$**
- Every inner node with  **$k$**  keys has  **$k + 1$**  non-null children.
- Optimized for reading/writing large data blocks.

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

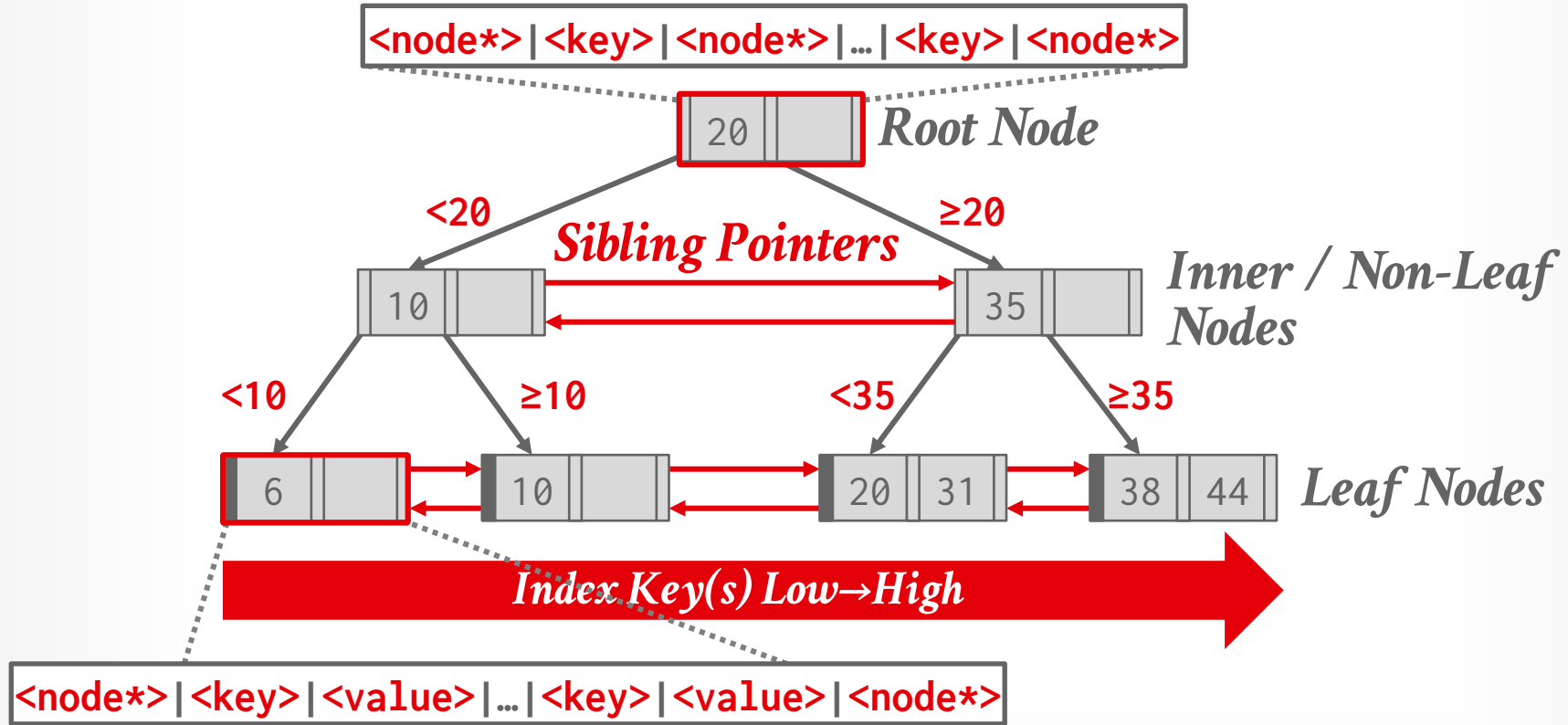
# B+TREE EXAMPLE



# B+TREE EXAMPLE

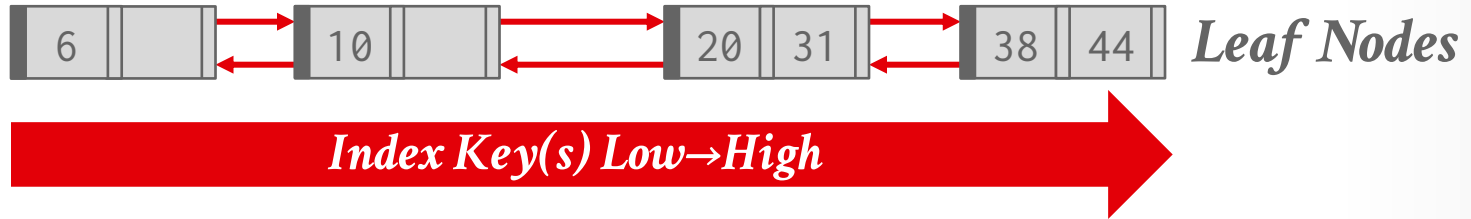


# B+TREE EXAMPLE





# B+TREE EXAMPLE



# NODES

---

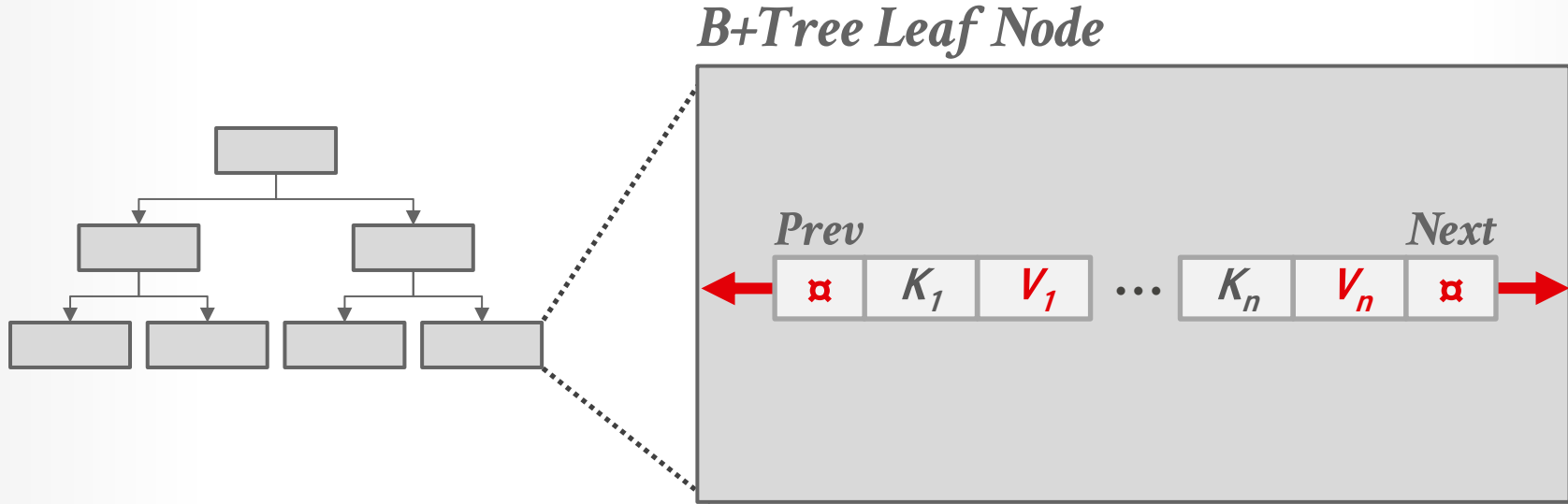
Every B+Tree node is comprised of an array of key/value pairs.

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

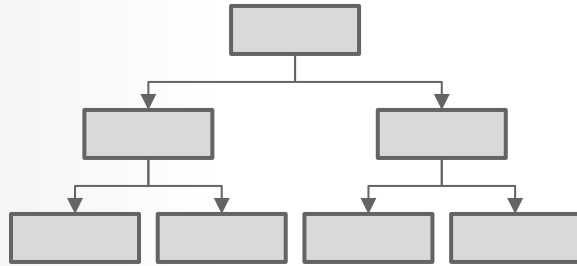
The arrays are (usually) kept in sorted key order.

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

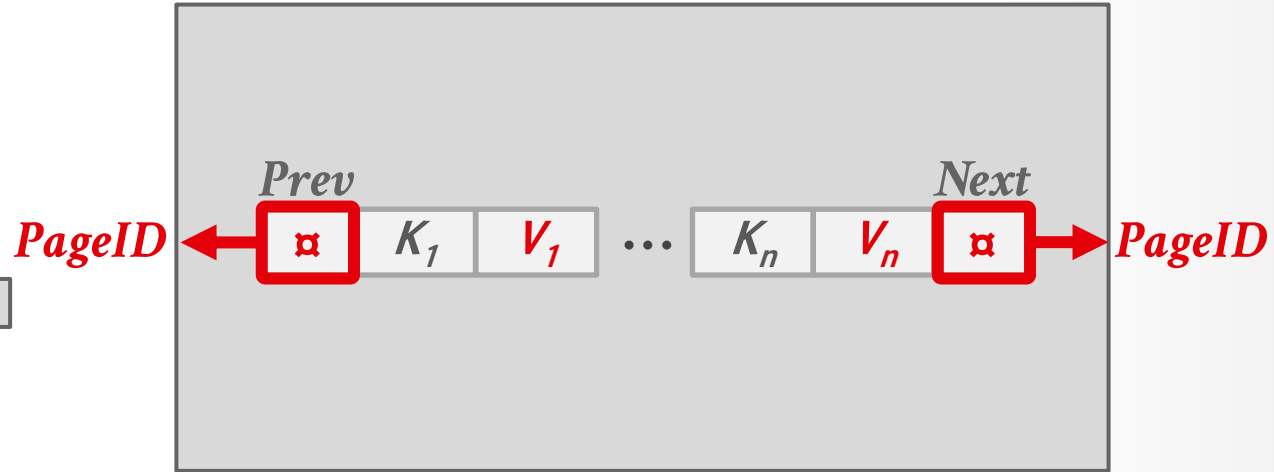
# B+TREE LEAF NODES



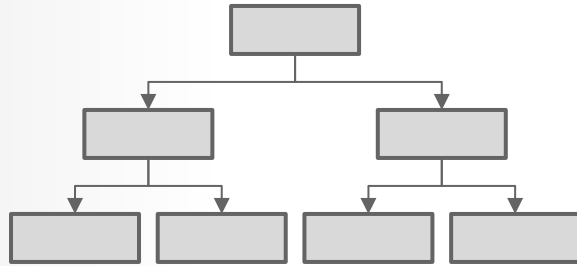
# B+TREE LEAF NODES



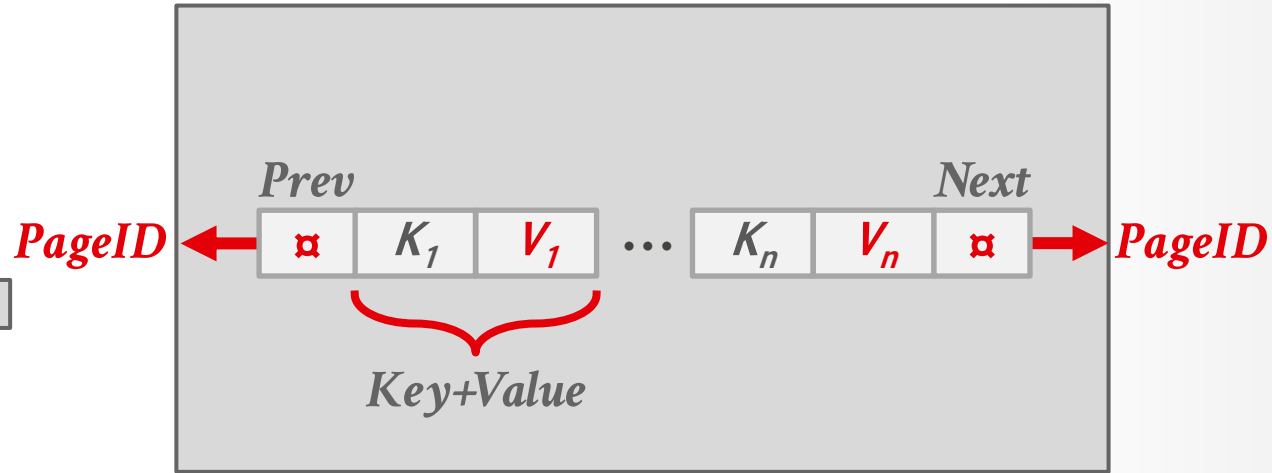
*B+Tree Leaf Node*



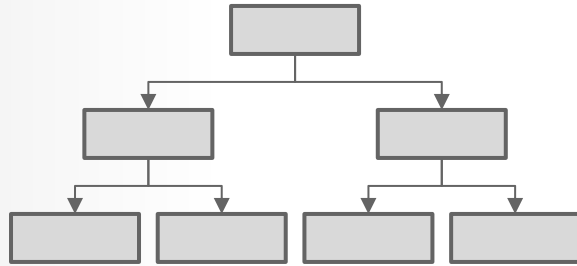
# B+TREE LEAF NODES



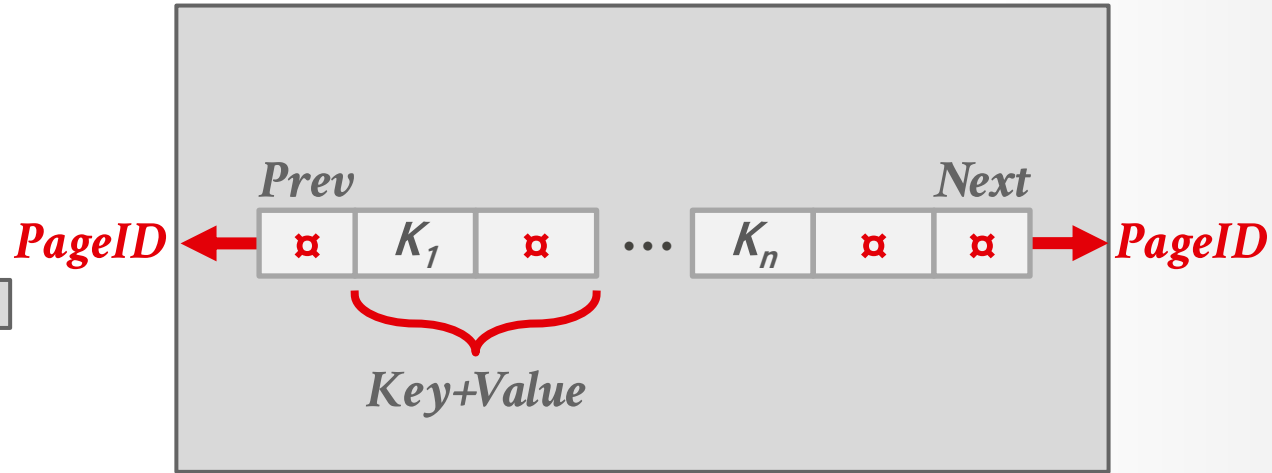
*B+Tree Leaf Node*



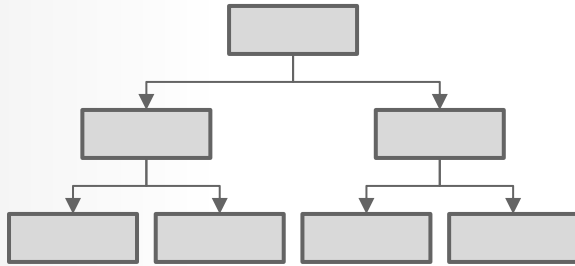
# B+TREE LEAF NODES



*B+Tree Leaf Node*



# B+TREE LEAF NODES



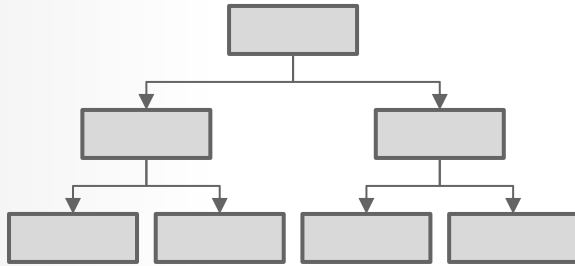
## *B+Tree Leaf Node*

| <i>Level</i> | <i>Slots</i> | <i>Prev</i> | <i>Next</i> | <i>High Key</i> |
|--------------|--------------|-------------|-------------|-----------------|
| #            | #            | ☐           | ☐           | $K_x$           |

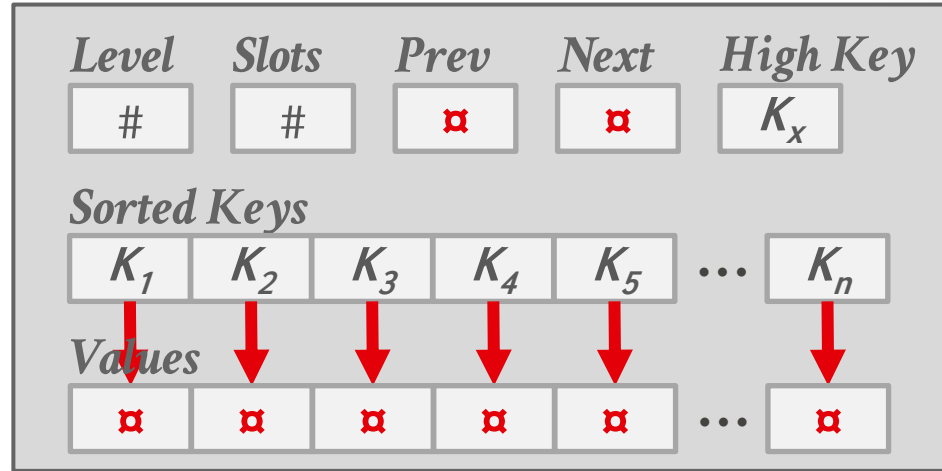
### *Sorted Key/Value Pairs*

|       |   |       |   |       |   |
|-------|---|-------|---|-------|---|
| $K_1$ | ☐ | $K_2$ | ☐ | $K_3$ | ☐ |
| $K_4$ | ☐ | $K_5$ | ☐ | ...   |   |

# B+TREE LEAF NODES



## B+Tree Leaf Node





# LEAF NODE VALUES

## Approach #1: Record IDs

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



## Approach #2: Tuple Data

- Index-Organized Storage (Lecture #05)
- Primary Key Index: Leaf nodes store the contents of the tuple.
- Secondary Indexes: Leaf nodes store tuples' primary key as their values.



# B-TREE VS. B+TREE

---

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

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

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

# B+TREE: INSERT

---



Find correct leaf node **L**.

Insert data entry into **L** in sorted order.

If **L** has enough space, done!

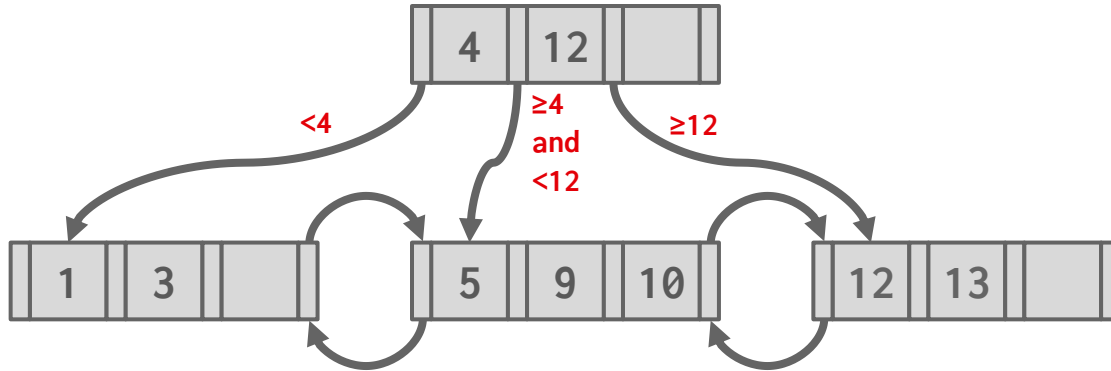
Otherwise, split **L** keys into **L** and a new node **L<sub>2</sub>**

→ Redistribute entries evenly, copy up middle key.

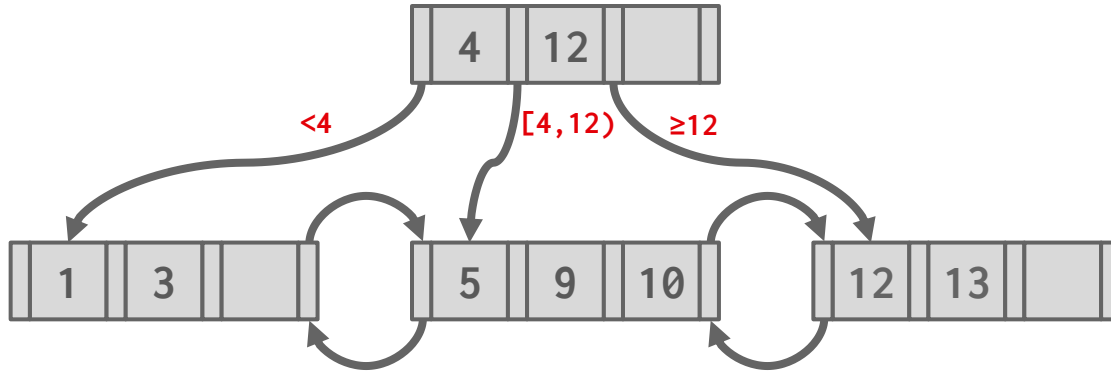
→ Insert index entry pointing to **L<sub>2</sub>** into parent of **L**.

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

# B+TREE: INSERT EXAMPLE (1)

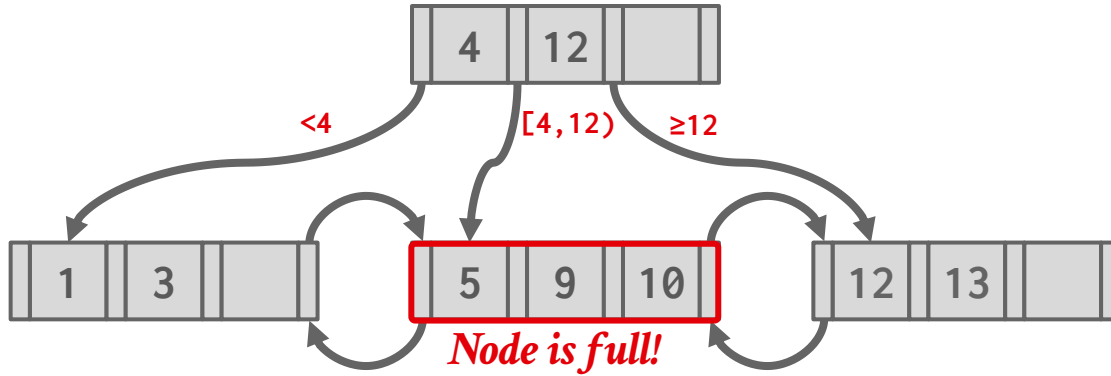


# B+TREE: INSERT EXAMPLE (1)



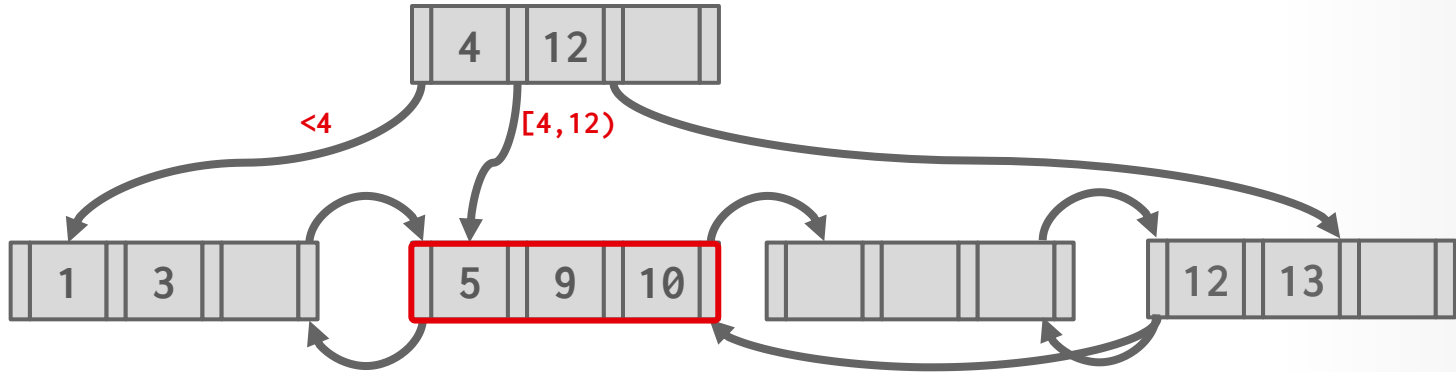
# B+TREE: INSERT EXAMPLE (1)

Insert 6



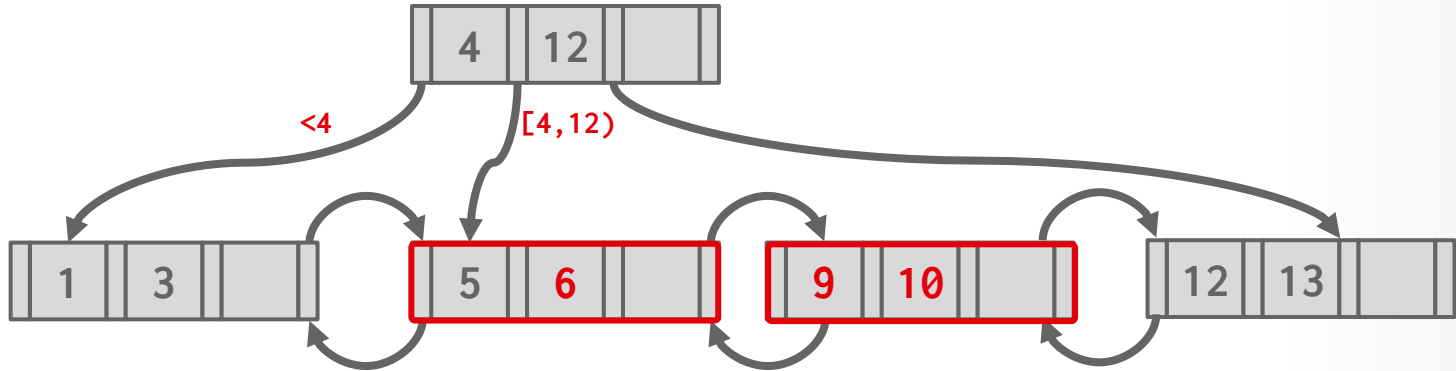
# B+TREE: INSERT EXAMPLE (1)

Insert 6



# B+TREE: INSERT EXAMPLE (1)

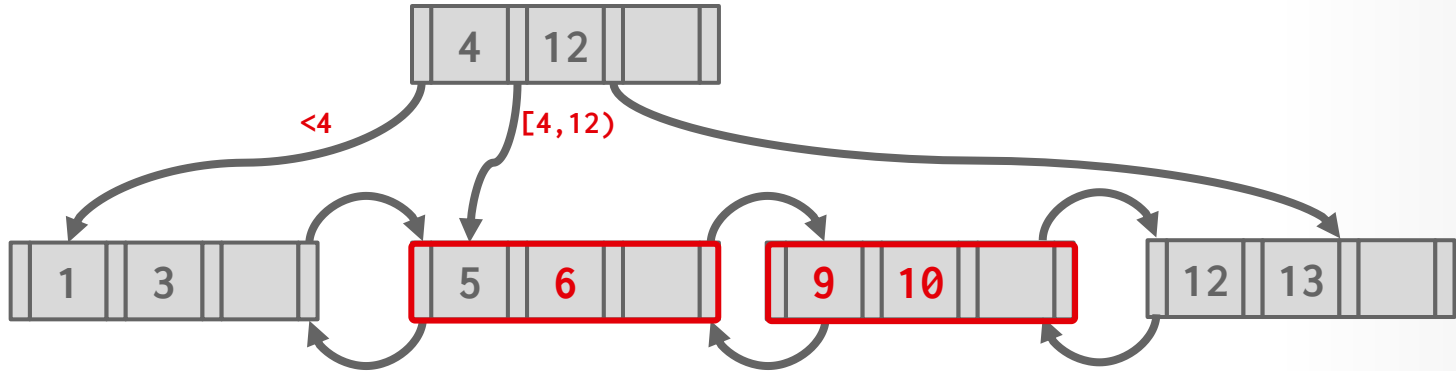
Insert 6





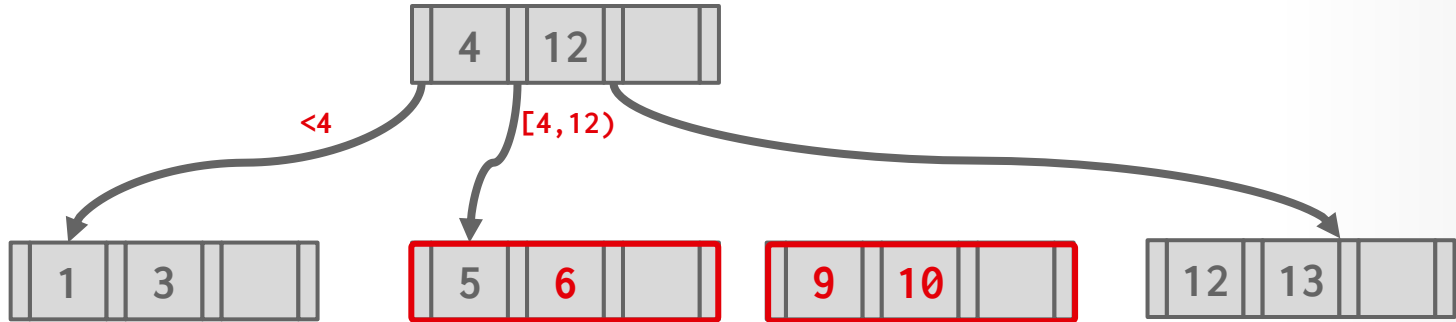
# B+TREE: INSERT EXAMPLE (1)

Insert 6



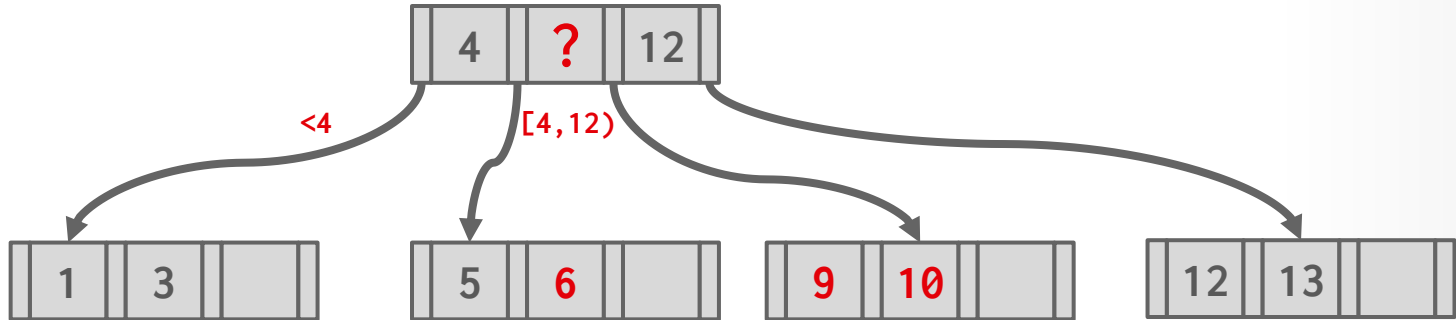
# B+TREE: INSERT EXAMPLE (1)

Insert 6



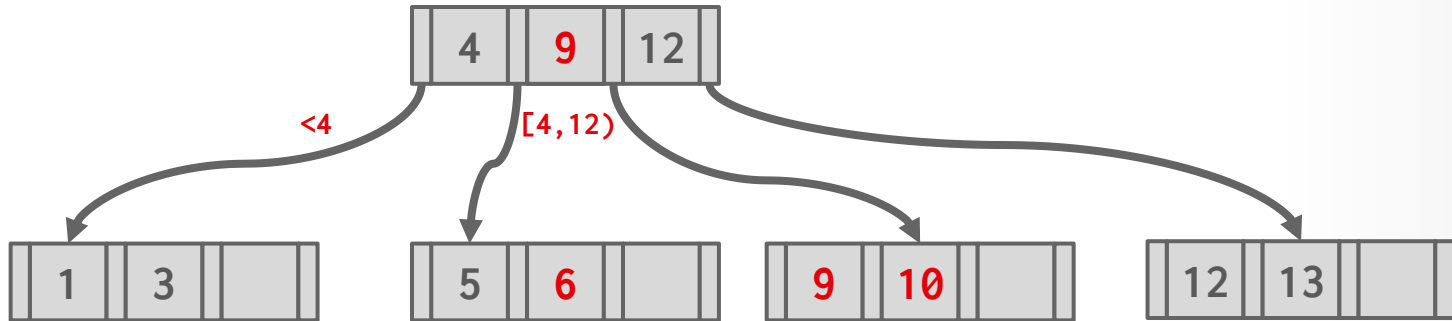
# B+TREE: INSERT EXAMPLE (1)

Insert 6



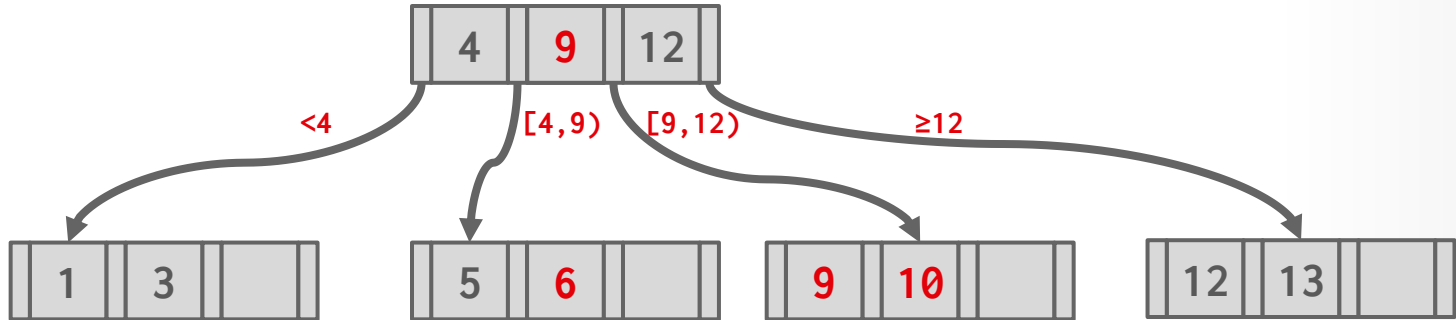
# B+TREE: INSERT EXAMPLE (1)

Insert 6



# B+TREE: INSERT EXAMPLE (1)

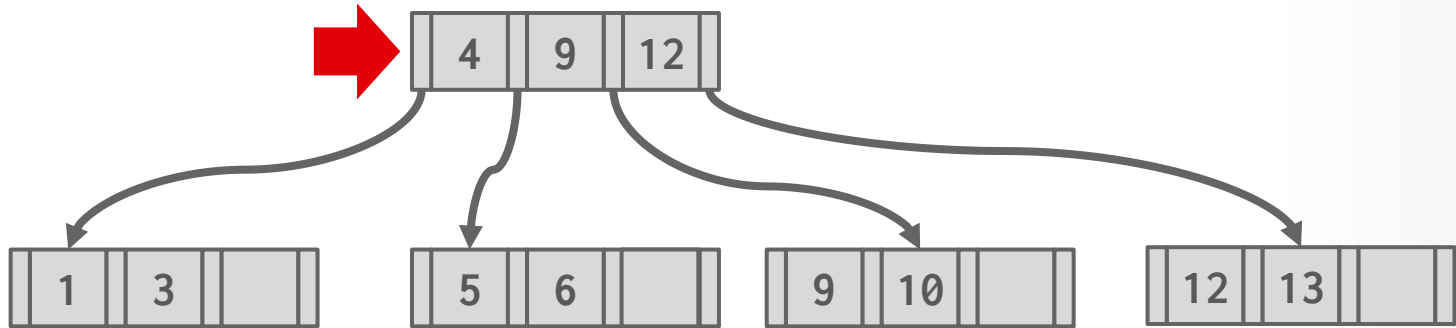
Insert 6



# B+TREE: INSERT EXAMPLE (1)

Insert 6

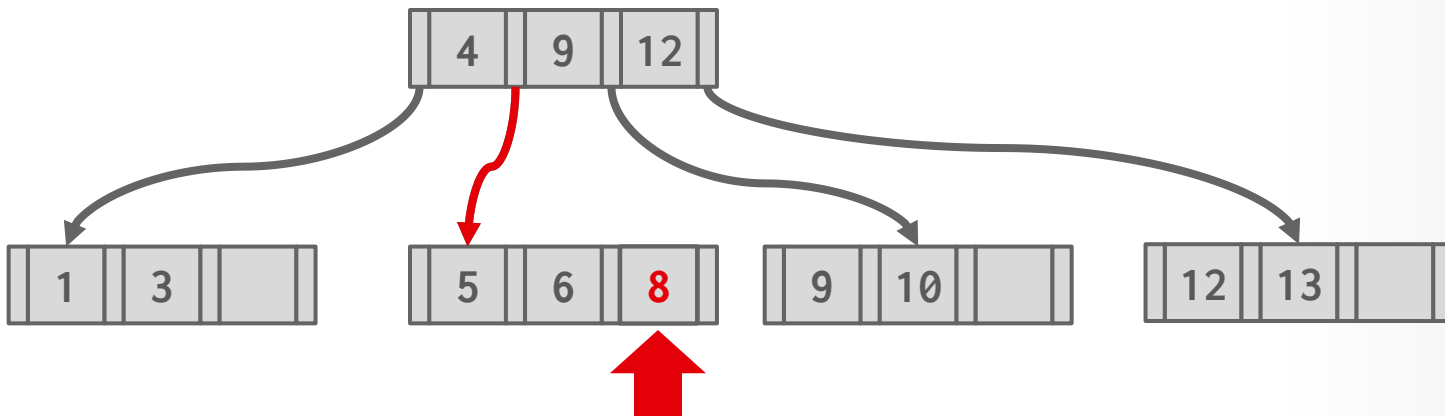
Insert 8



# B+TREE: INSERT EXAMPLE (1)

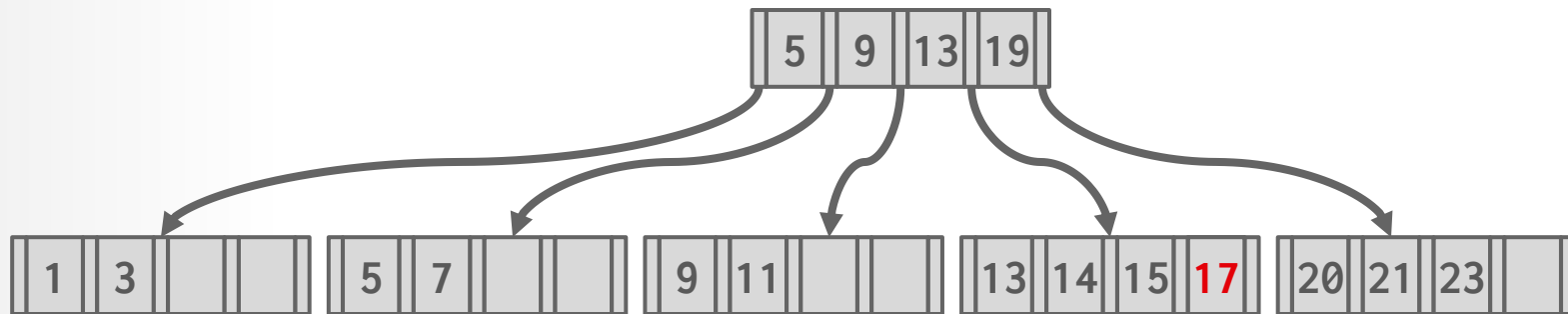
Insert 6

Insert 8



# B+TREE: INSERT EXAMPLE (2)

Insert 17



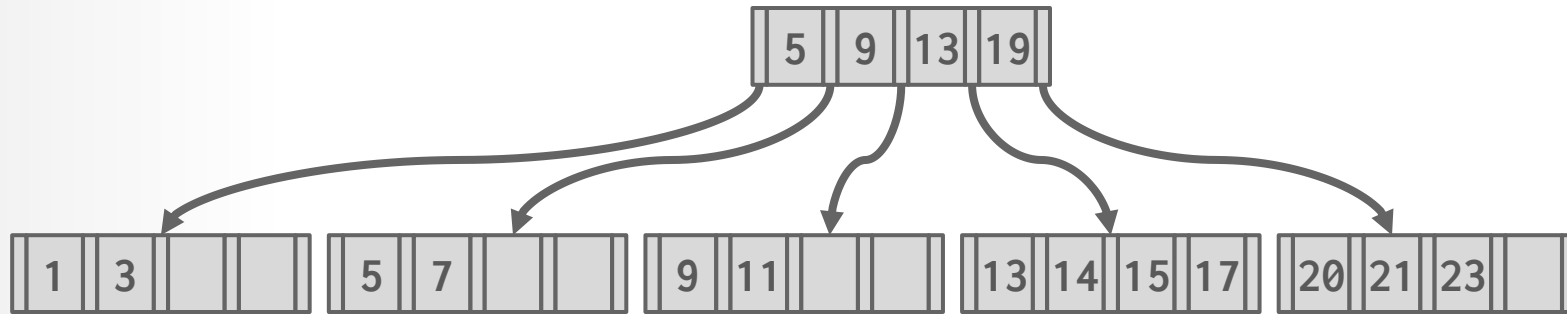
*Note: New Example/Tree.*



# B+TREE: INSERT EXAMPLE (3)

Insert 17

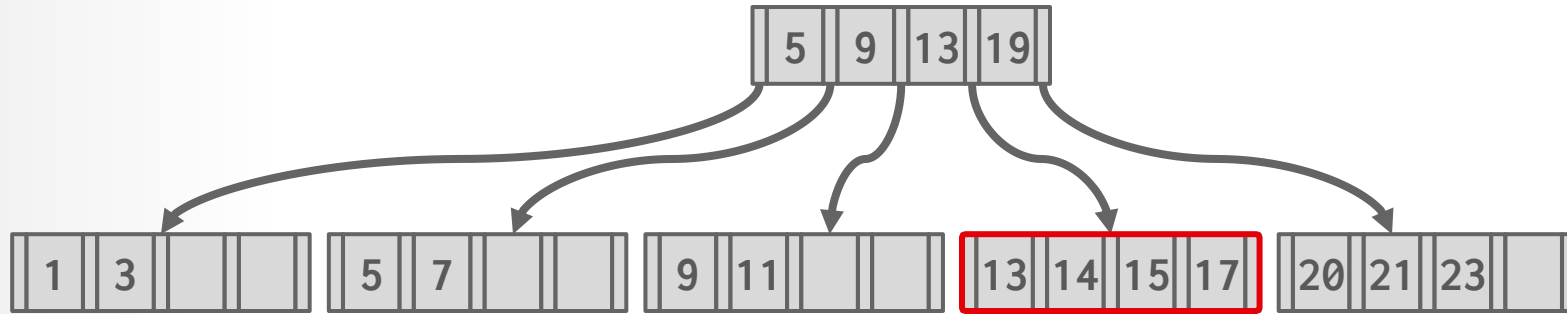
Insert 16



# B+TREE: INSERT EXAMPLE (3)

Insert 17

Insert 16

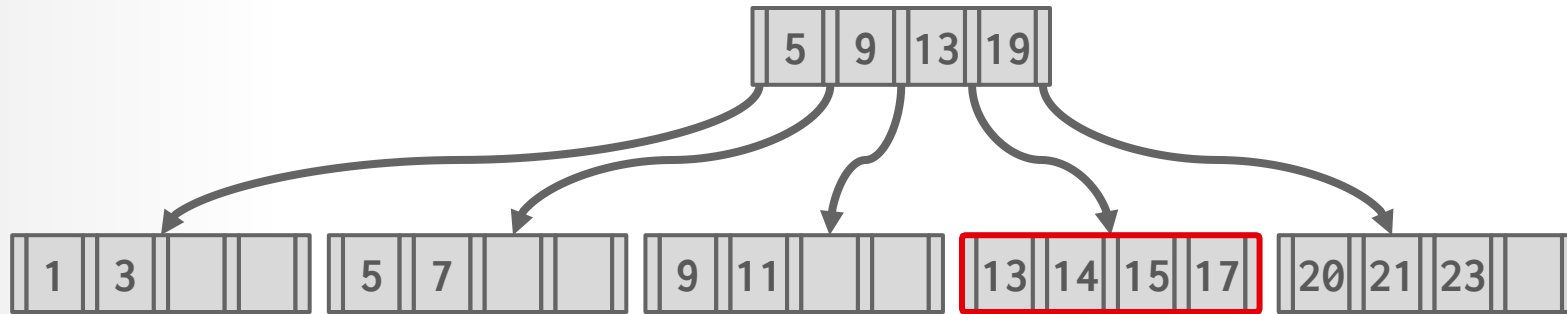


*No space in the node where  
the new key "belongs".*

# B+TREE: INSERT EXAMPLE (3)

Insert 17

Insert 16



*Split the node!*

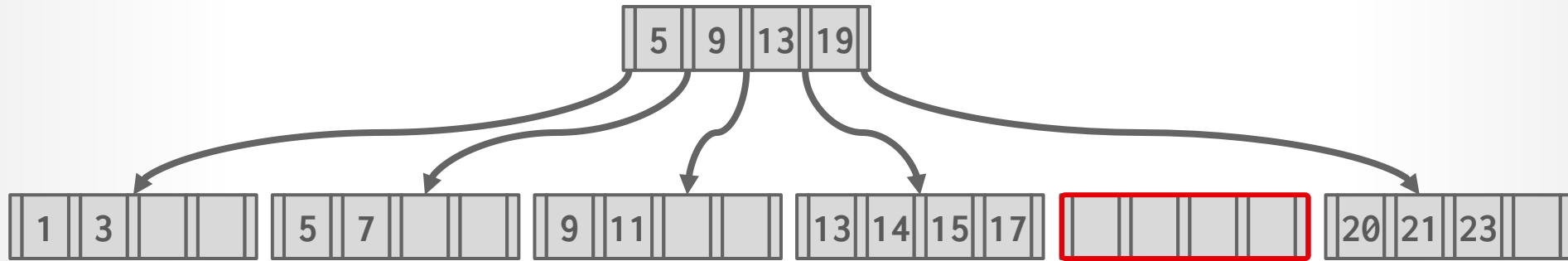
*Copy the middle key.*

*Push the key up.*

# B+TREE: INSERT EXAMPLE (3)

Insert 17

Insert 16



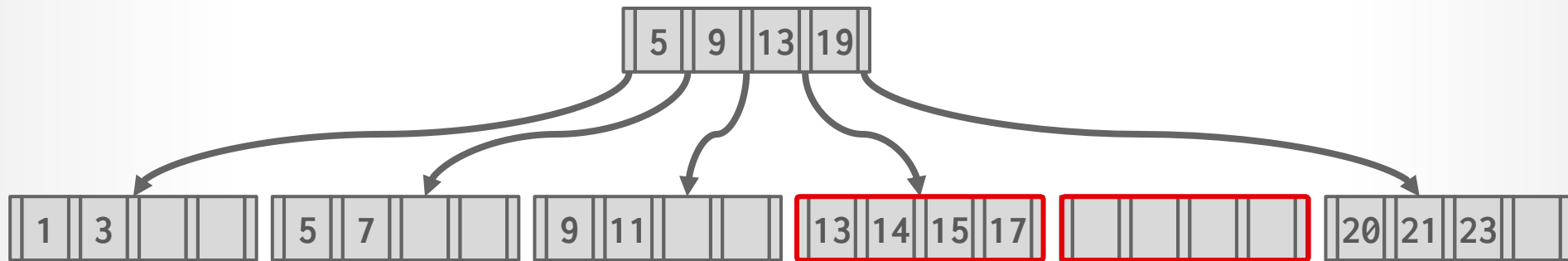
*New Node!*

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

# B+TREE: INSERT EXAMPLE (3)

Insert 17

Insert 16



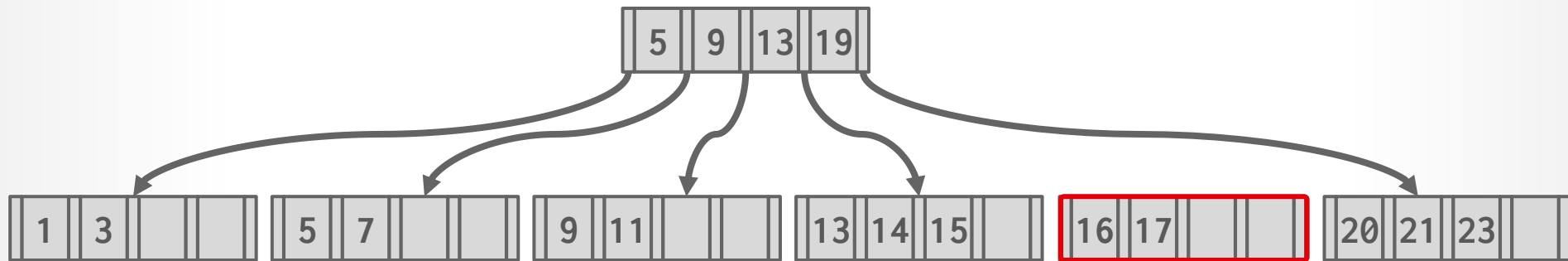
*New Node!*

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

# B+TREE: INSERT EXAMPLE (3)

Insert 17

Insert 16



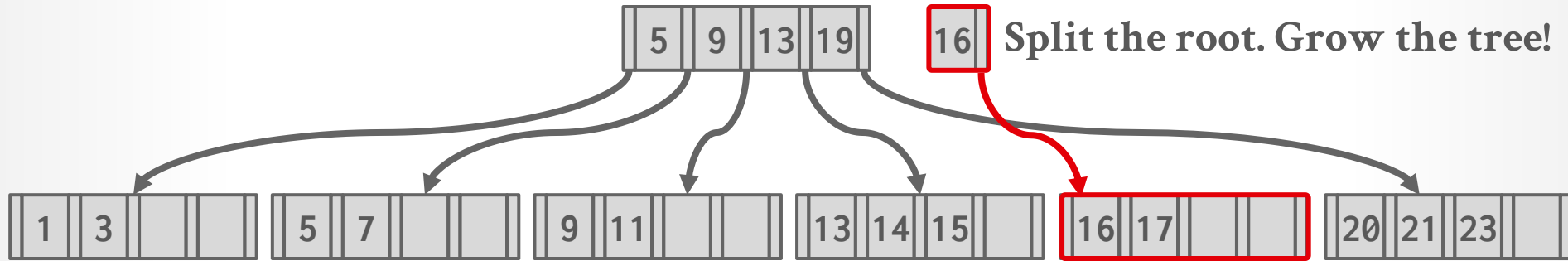
*But this is an “orphan” node!  
No parent node points to it.*

# B+TREE: INSERT EXAMPLE (3)

Insert 17

Insert 16

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

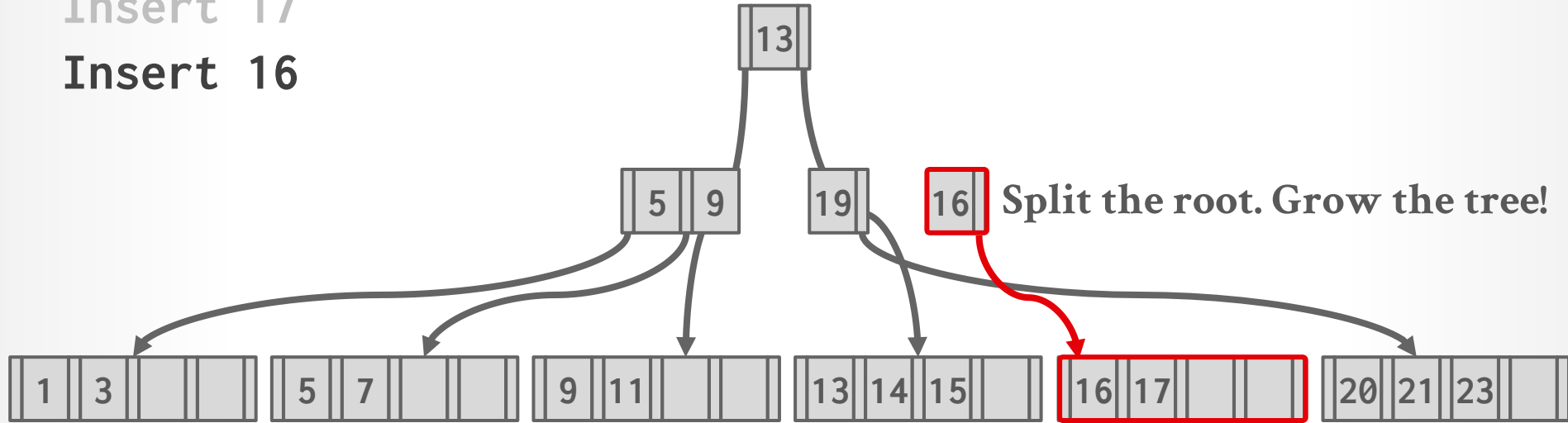


*But this is an "orphan" node!  
No parent node points to it.*

# B+TREE: INSERT EXAMPLE (3)

Insert 17

Insert 16

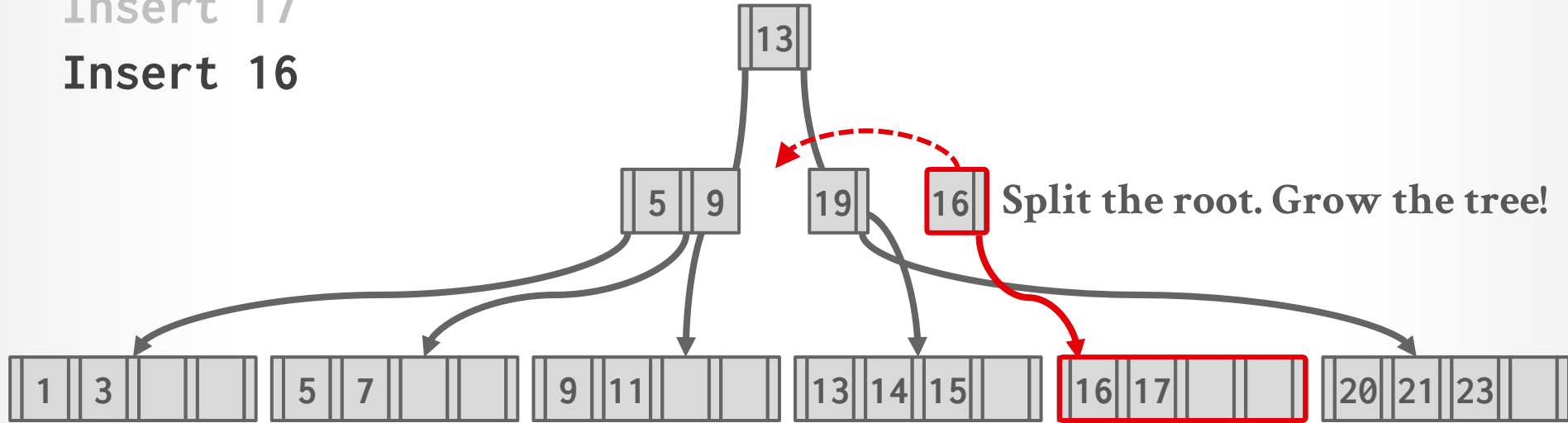




# B+TREE: INSERT EXAMPLE (3)

Insert 17

Insert 16



# B+TREE: INSERT EXAMPLE (3)

Insert 17

Insert 16



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



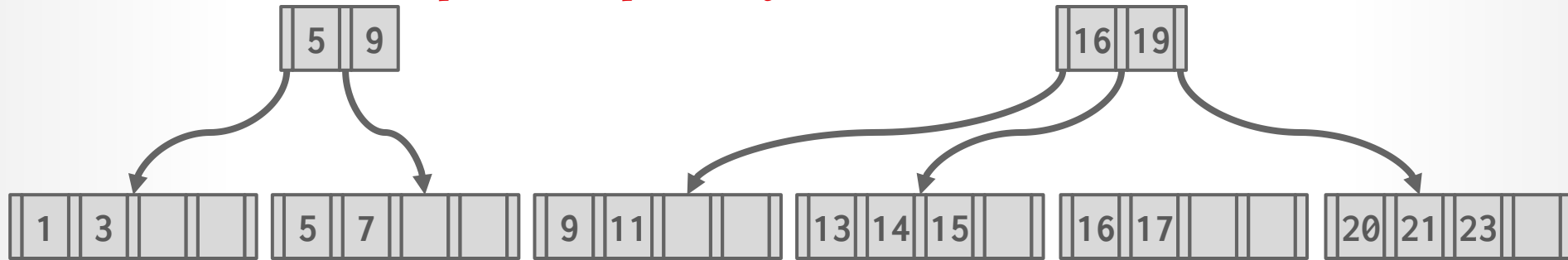
# B+TREE: INSERT EXAMPLE (3)

Insert 17

Insert 16



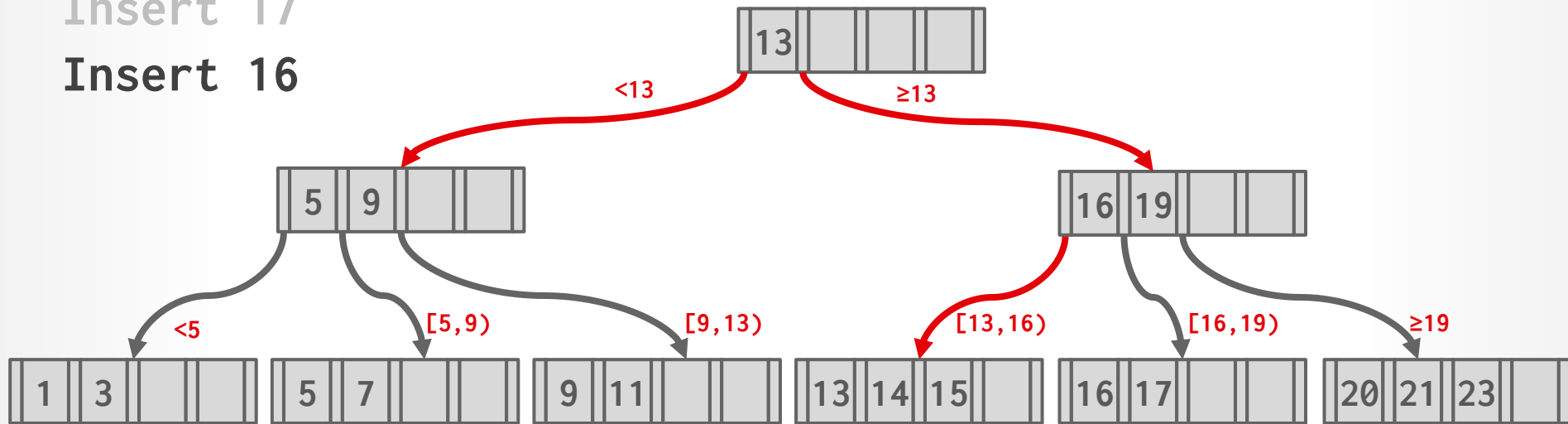
*Next, need to split the "old" root, then point to the split nodes from the new root.*



# B+TREE: INSERT EXAMPLE (3)

Insert 17

Insert 16



# B+TREE: DELETE

---

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

Remove the entry.

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

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

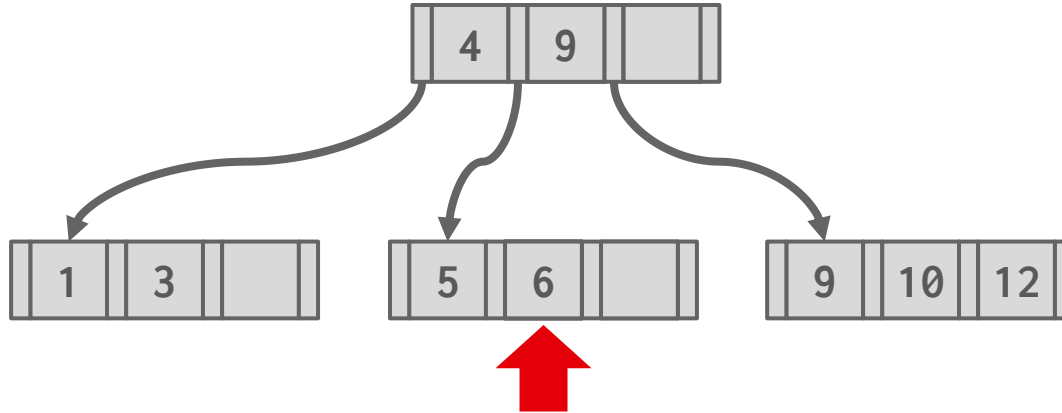
→ Try to re-distribute, borrowing from sibling (adjacent node with same parent as **L**).

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

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

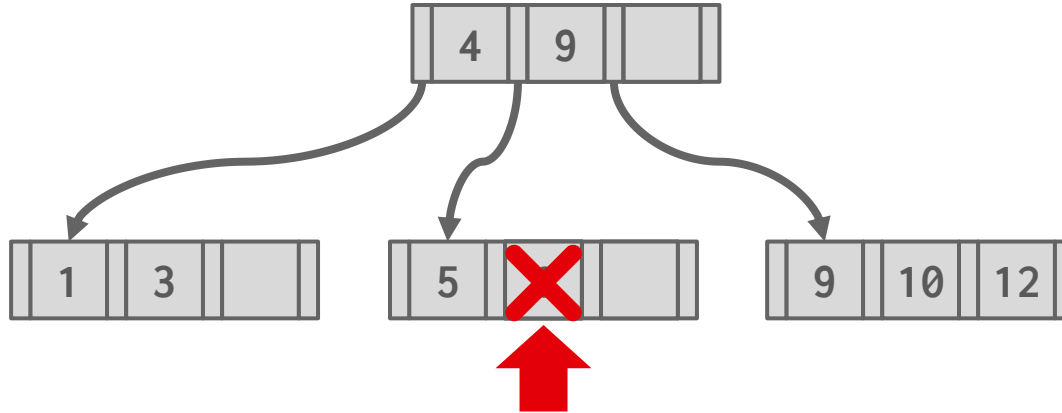
# B+TREE: DELETE EXAMPLE (1)

Delete 6



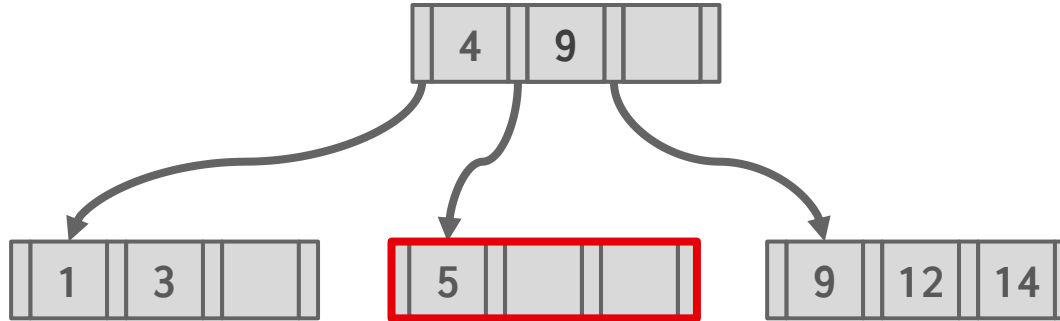
# B+TREE: DELETE EXAMPLE (1)

Delete 6



# B+TREE: DELETE EXAMPLE (1)

Delete 6



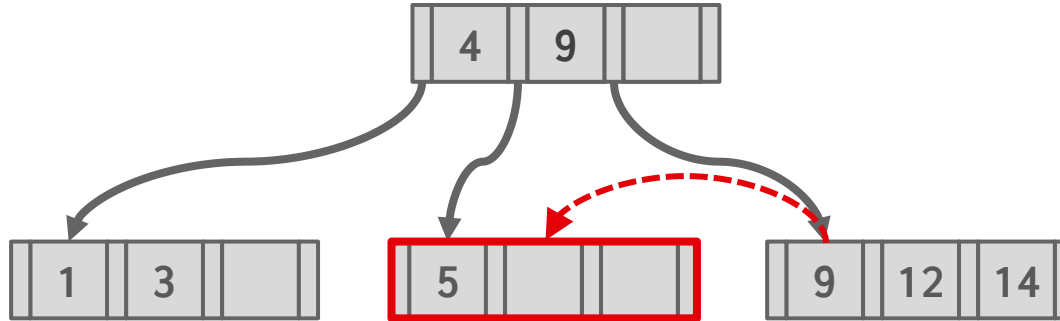
*Borrow from a "rich" sibling node.*

*Could borrow from either sibling.*



# B+TREE: DELETE EXAMPLE (1)

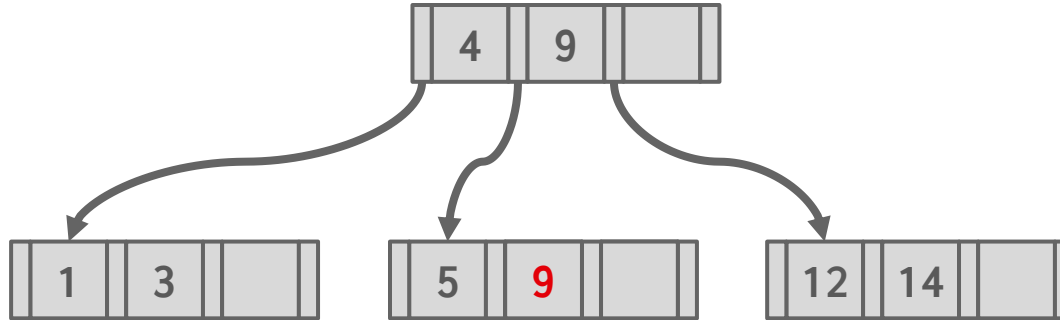
Delete 6



*Borrow from a "rich" sibling node.  
Could borrow from either sibling.*

# B+TREE: DELETE EXAMPLE (1)

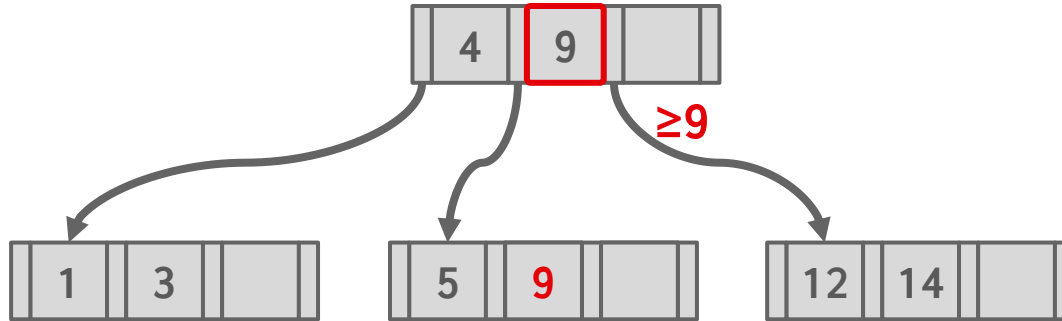
Delete 6



*Need to update parent node!*

# B+TREE: DELETE EXAMPLE (1)

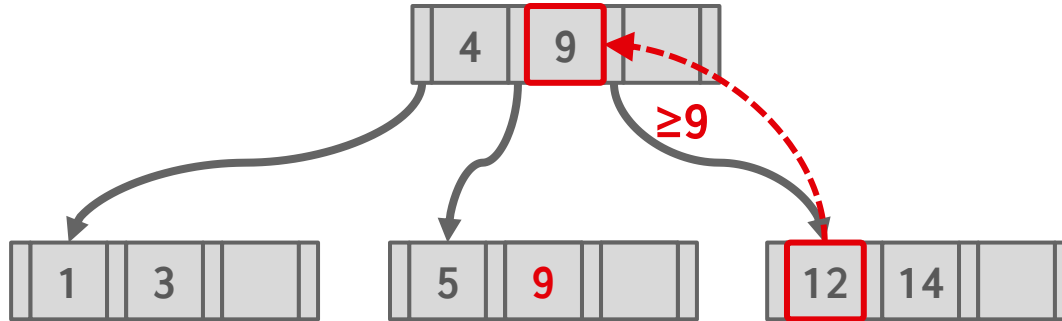
Delete 6



*Need to update parent node!*

# B+TREE: DELETE EXAMPLE (1)

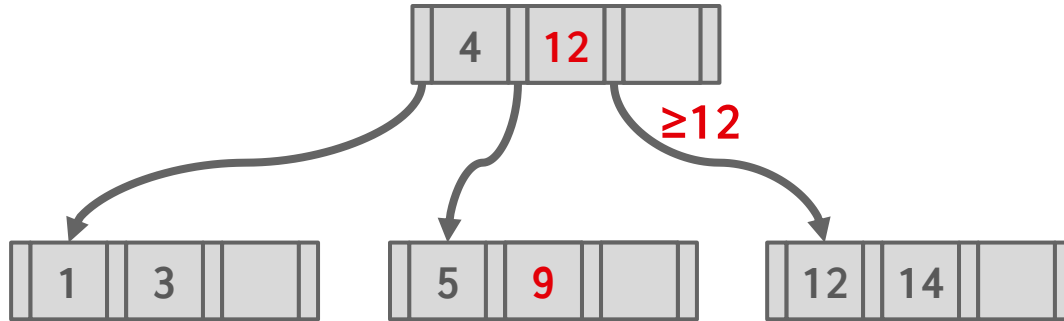
Delete 6



*Need to update parent node!*

# B+TREE: DELETE EXAMPLE (1)

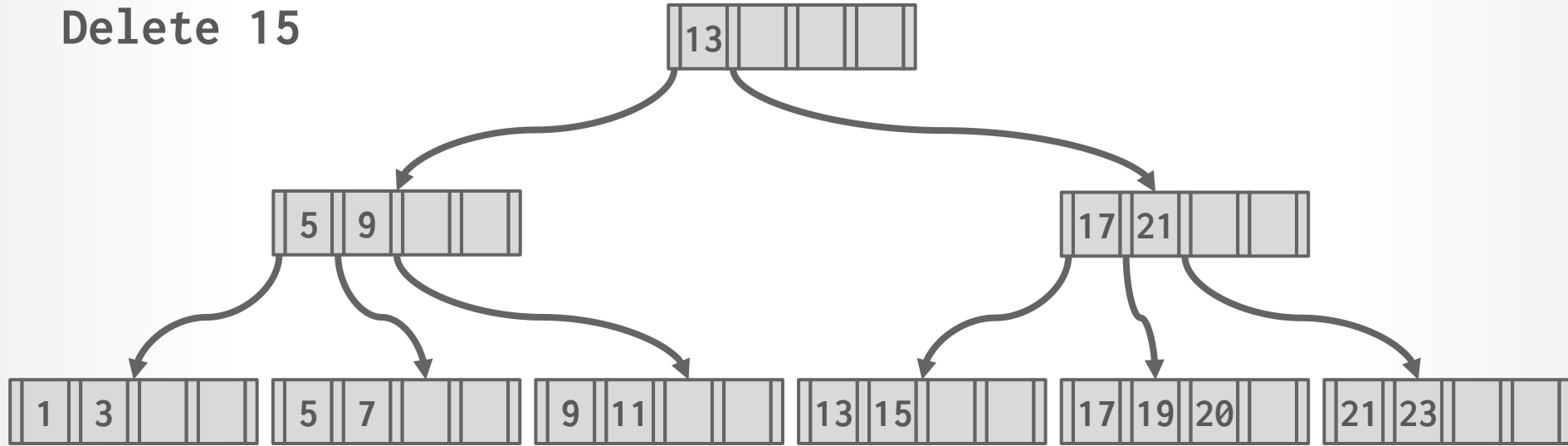
Delete 6



*Need to update parent node!*

# B+TREE: DELETE EXAMPLE (2)

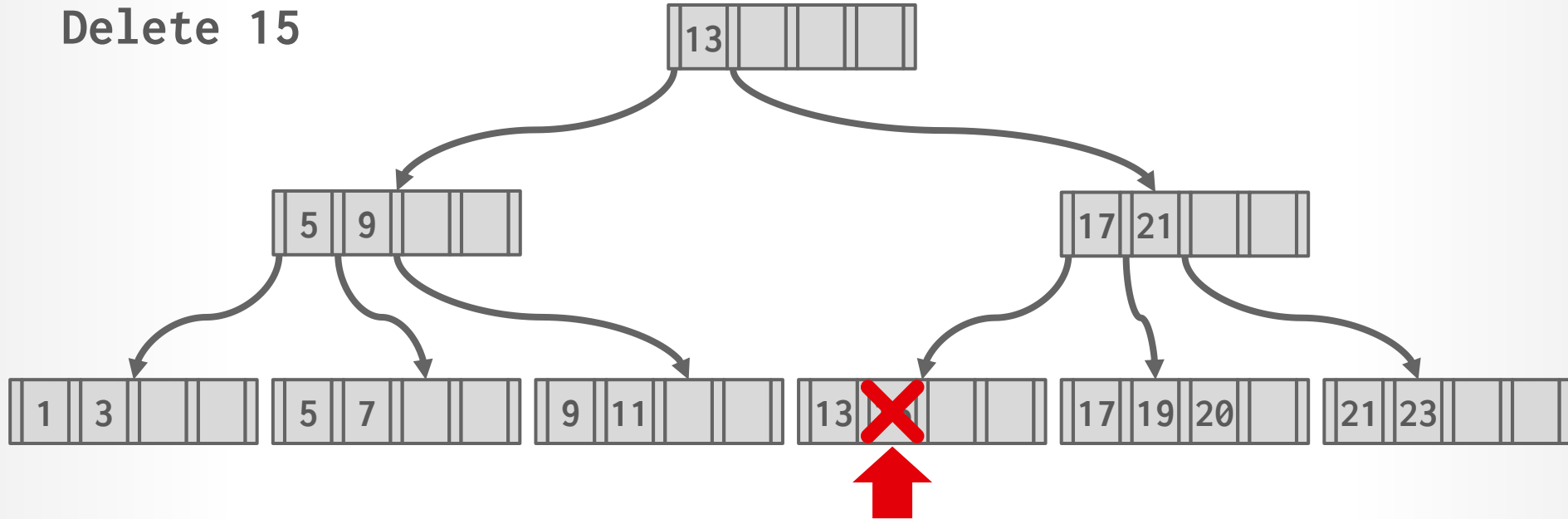
Delete 15



*Note: New Example/Tree.*

# B+TREE: DELETE EXAMPLE (2)

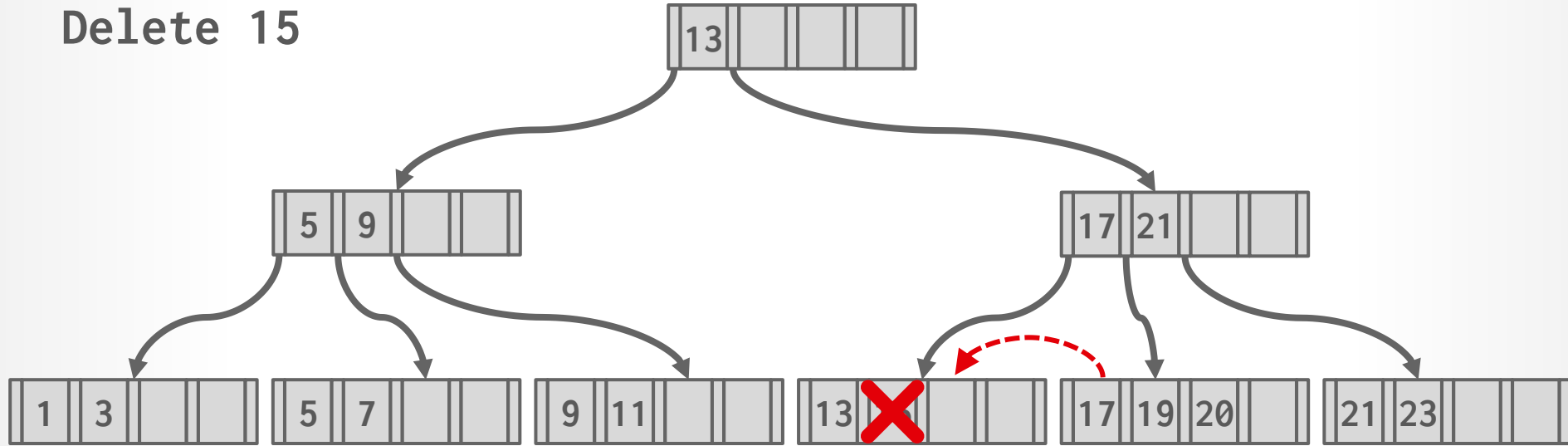
Delete 15



*Note: New Example/Tree.*

# B+TREE: DELETE EXAMPLE (2)

Delete 15



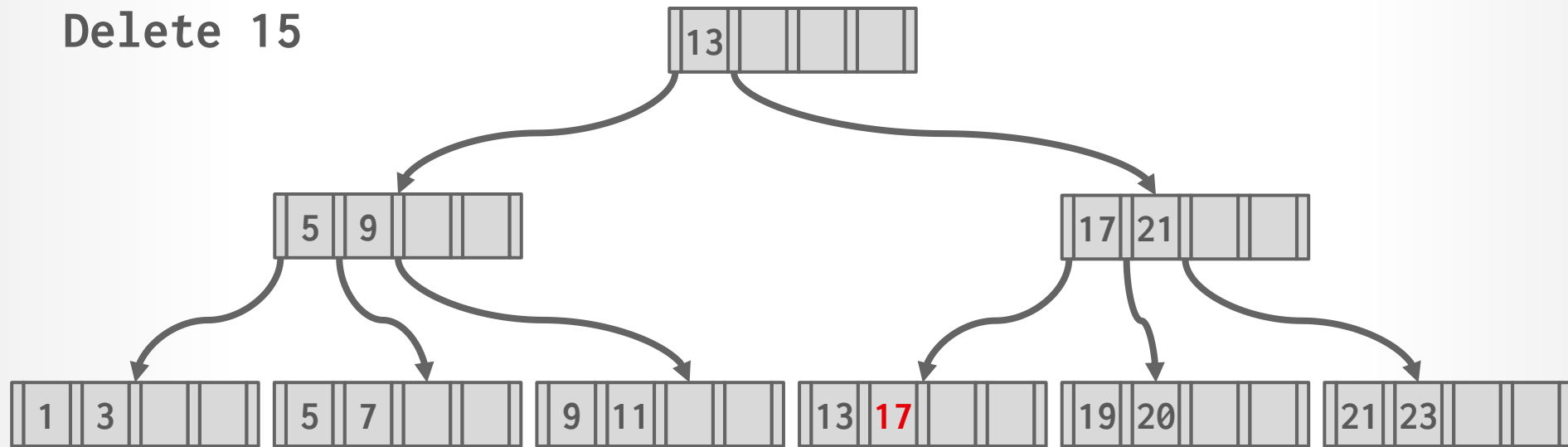
*Borrow from a "rich" sibling node.*

*Note: New Example/Tree.*



# B+TREE: DELETE EXAMPLE (2)

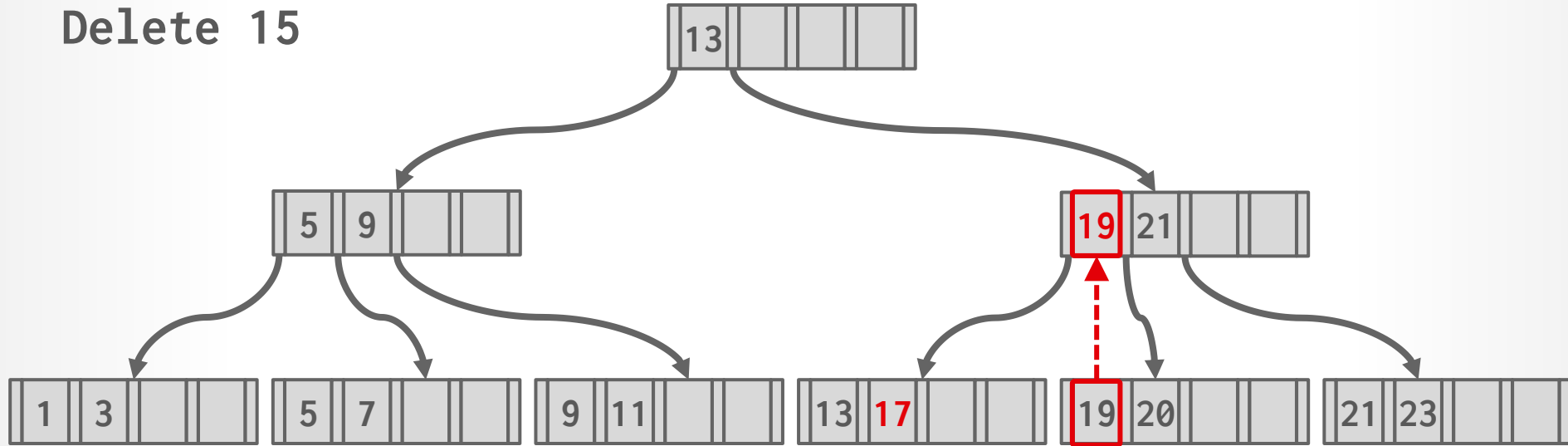
Delete 15



*Need to update parent node!*

# B+TREE: DELETE EXAMPLE (2)

Delete 15

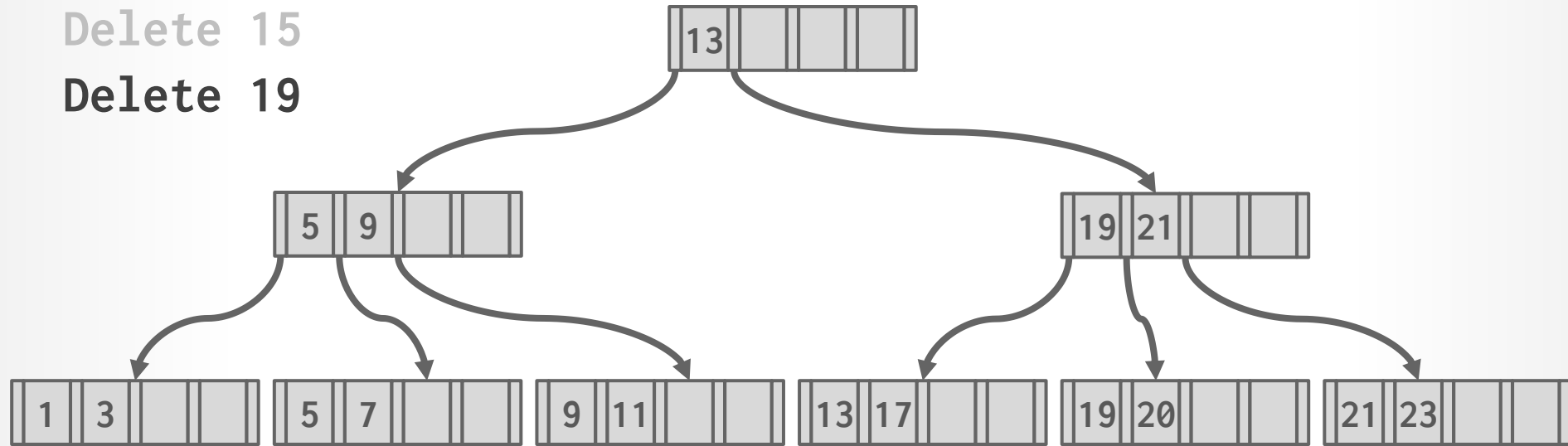


*Need to update parent node!*

# B+TREE: DELETE EXAMPLE (3)

Delete 15

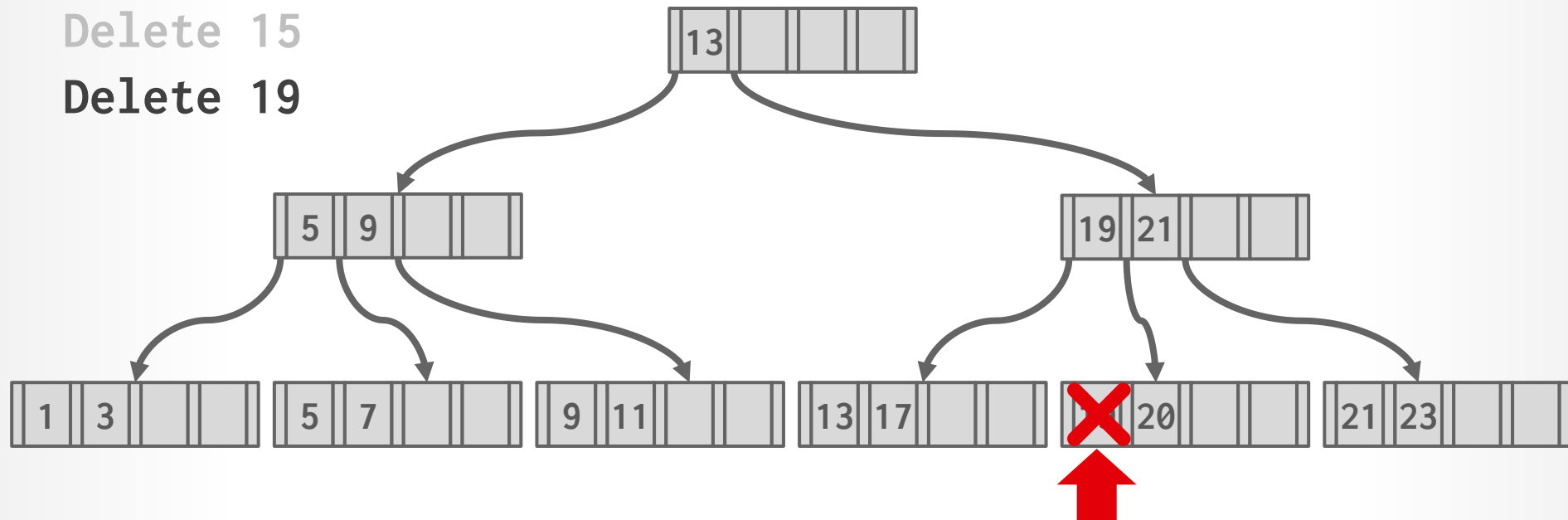
Delete 19



# B+TREE: DELETE EXAMPLE (3)

Delete 15

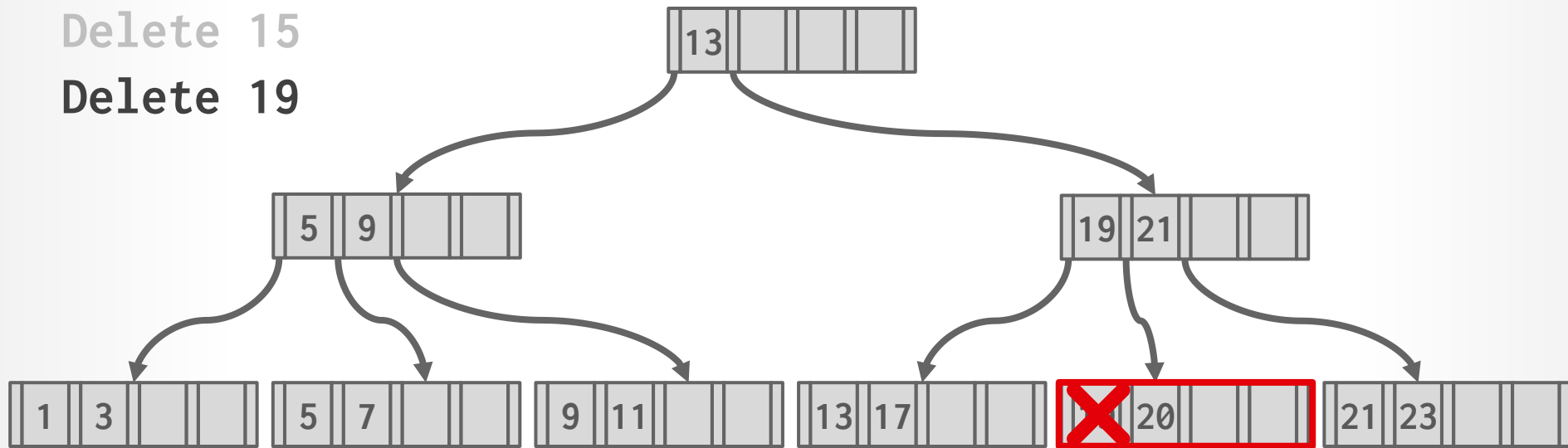
Delete 19



# B+TREE: DELETE EXAMPLE (3)

Delete 15

Delete 19



*Under-filled!*

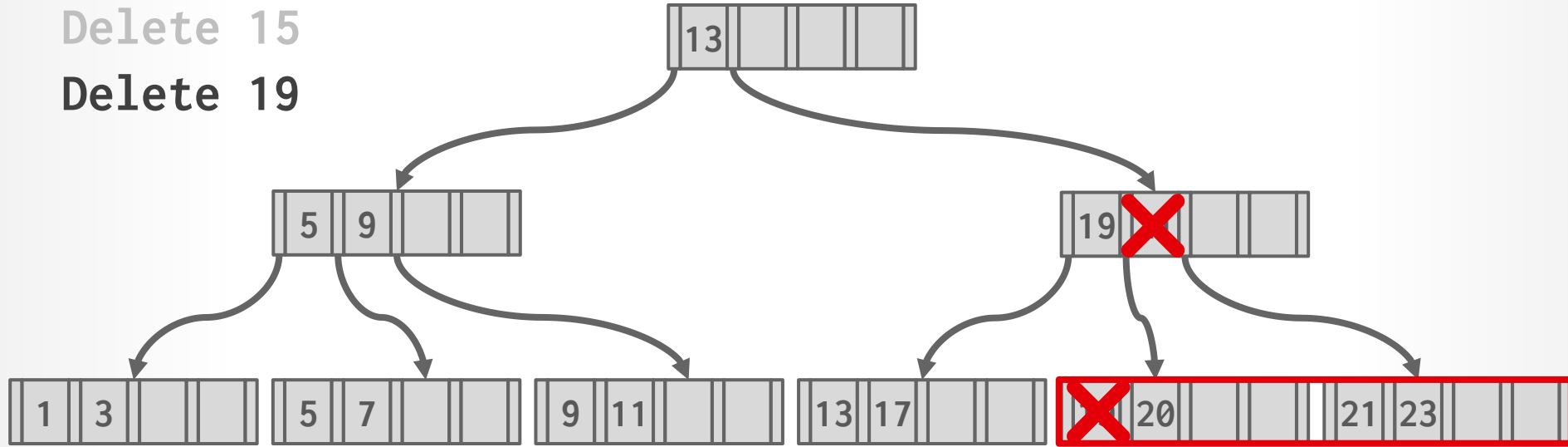
*No "rich" sibling nodes to borrow.*

*Merge with a sibling*

# B+TREE: DELETE EXAMPLE (3)

Delete 15

Delete 19



*Under-filled!*

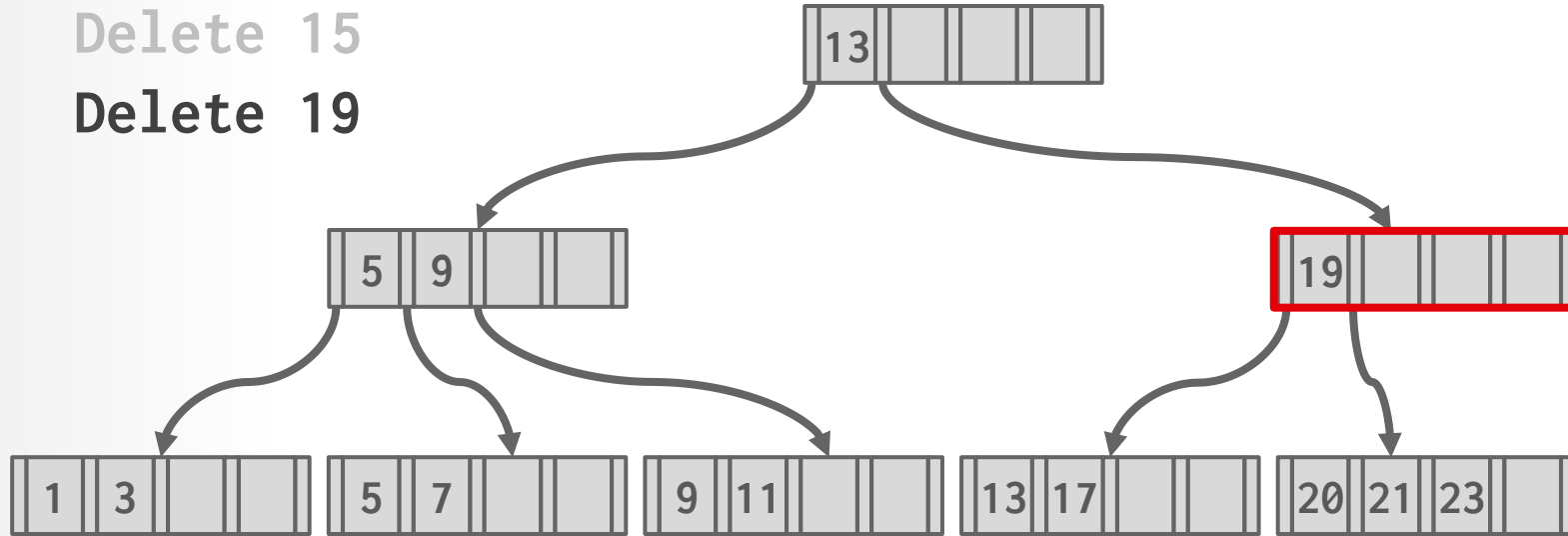
*No "rich" sibling nodes to borrow.*

*Merge with a sibling*

# B+TREE: DELETE EXAMPLE (3)

Delete 15

Delete 19

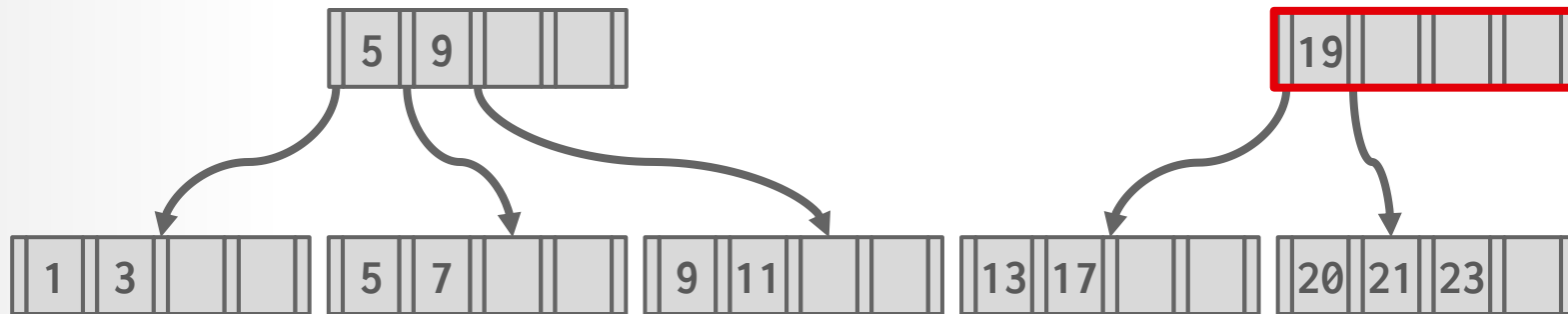


*This node is  
under-filled!  
Pull-down.*

# B+TREE: DELETE EXAMPLE (3)

Delete 15

Delete 19



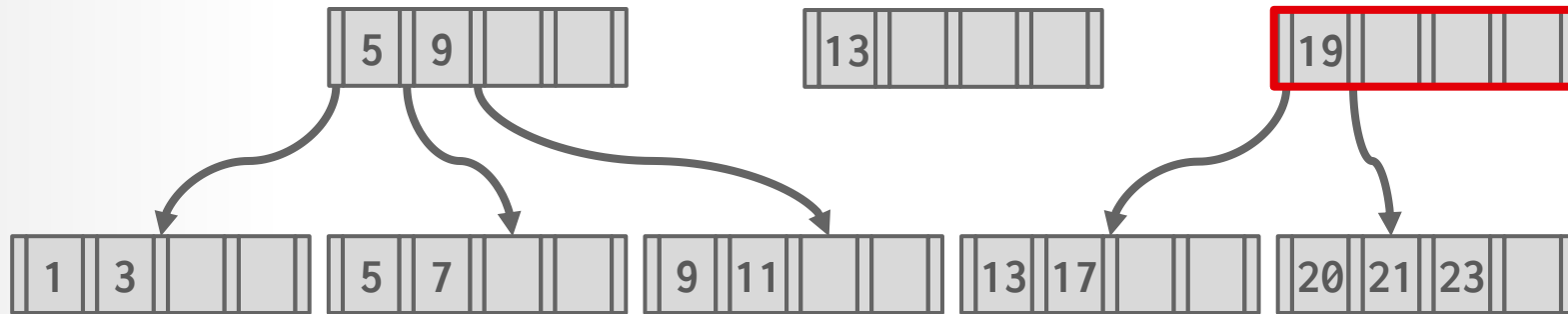
*This node is under-filled!  
Pull-down.*



# B+TREE: DELETE EXAMPLE (3)

Delete 15

Delete 19

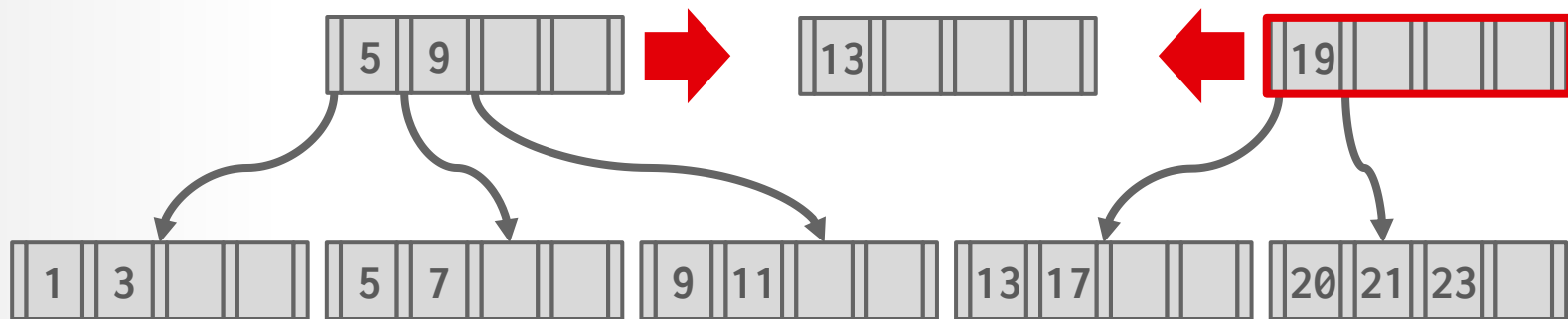


*This node is  
under-filled!  
Pull-down.*

# B+TREE: DELETE EXAMPLE (3)

Delete 15

Delete 19



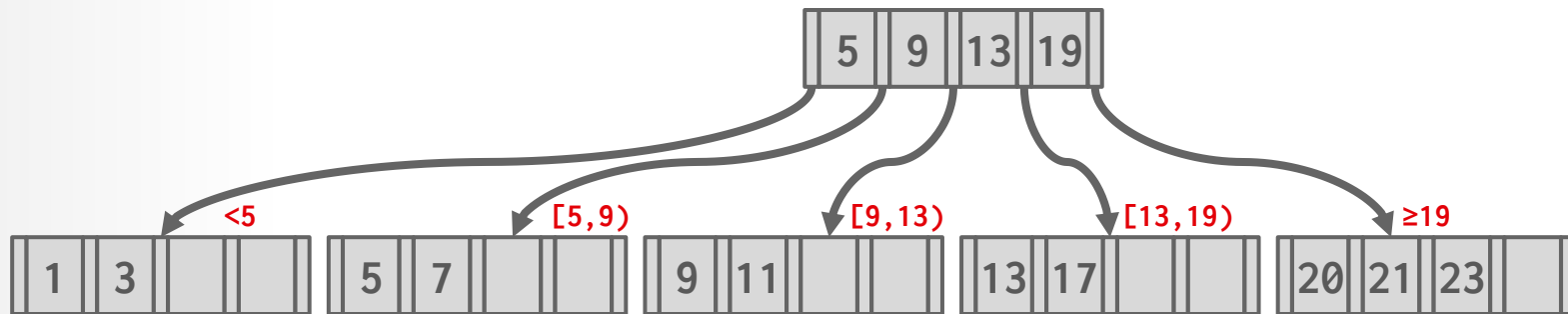
*This node is  
under-filled!  
Pull-down.*

# B+TREE: DELETE EXAMPLE (3)

Delete 15

Delete 19

*The tree has shrunk in height.*



# COMPOSITE INDEX

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

→ Example: Index on **<a,b,c>**

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

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

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

→ Supported: **(a=1 AND b=2)**

→ Rarely Supported: **(b=2), (c=3)**

# COMPOSITE INDEX

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

→ Example: Index on **<a, b, c>**

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

*Sort Order*

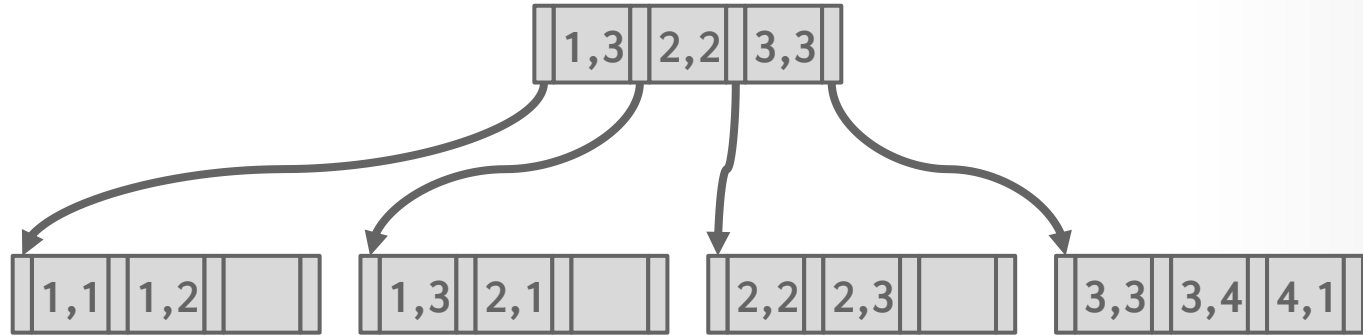
*Null Handling*

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

- Supported: **(a=1 AND b=2 AND c=3)**
- Supported: **(a=1 AND b=2)**
- Rarely Supported: **(b=2), (c=3)**

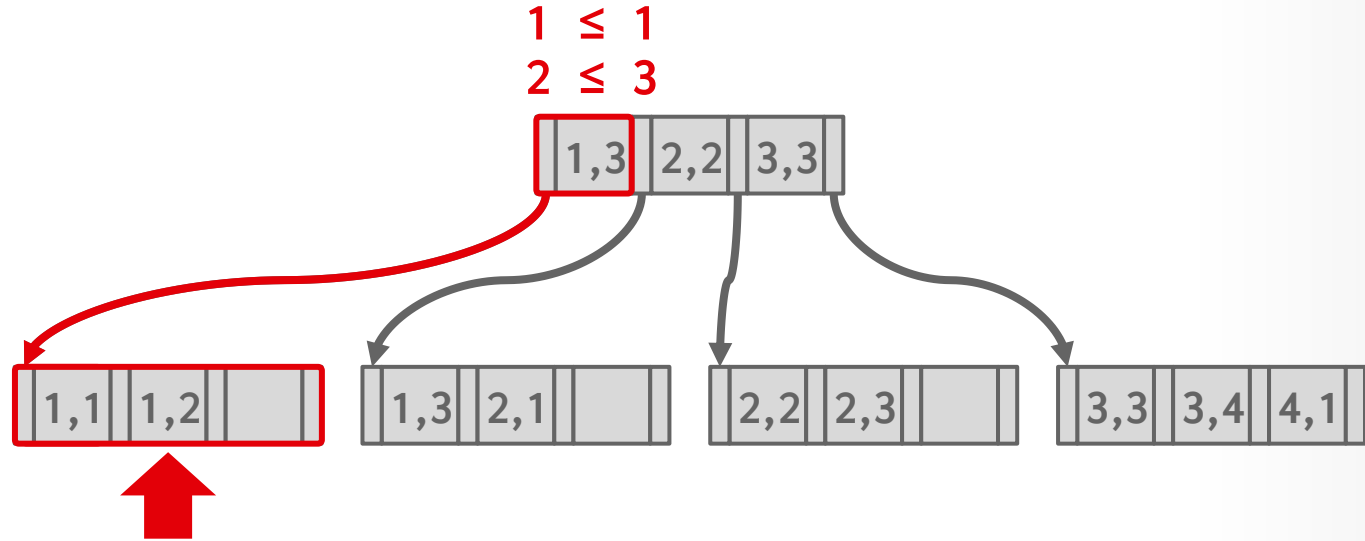
# SELECTION CONDITIONS

Find Key=(1,2)



# SELECTION CONDITIONS

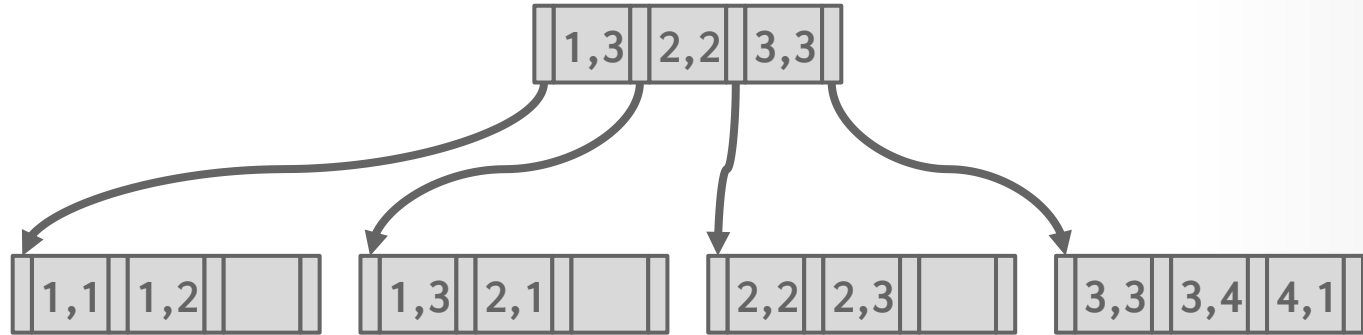
Find Key=(1,2)



# SELECTION CONDITIONS

Find Key=(1,2)

Find Key=(1,\*)

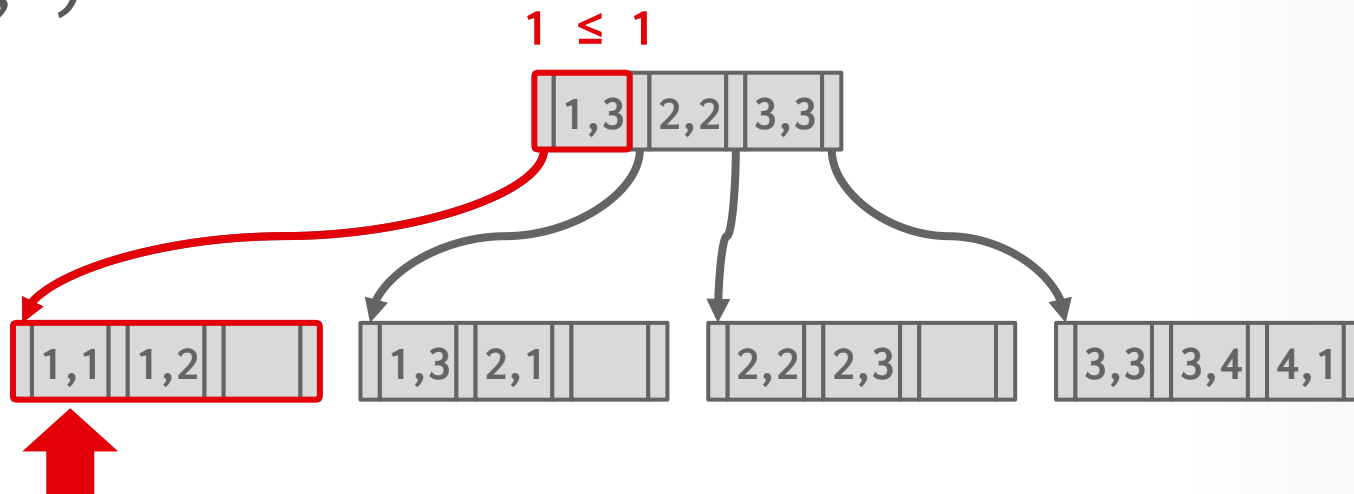




# SELECTION CONDITIONS

Find Key=(1,2)

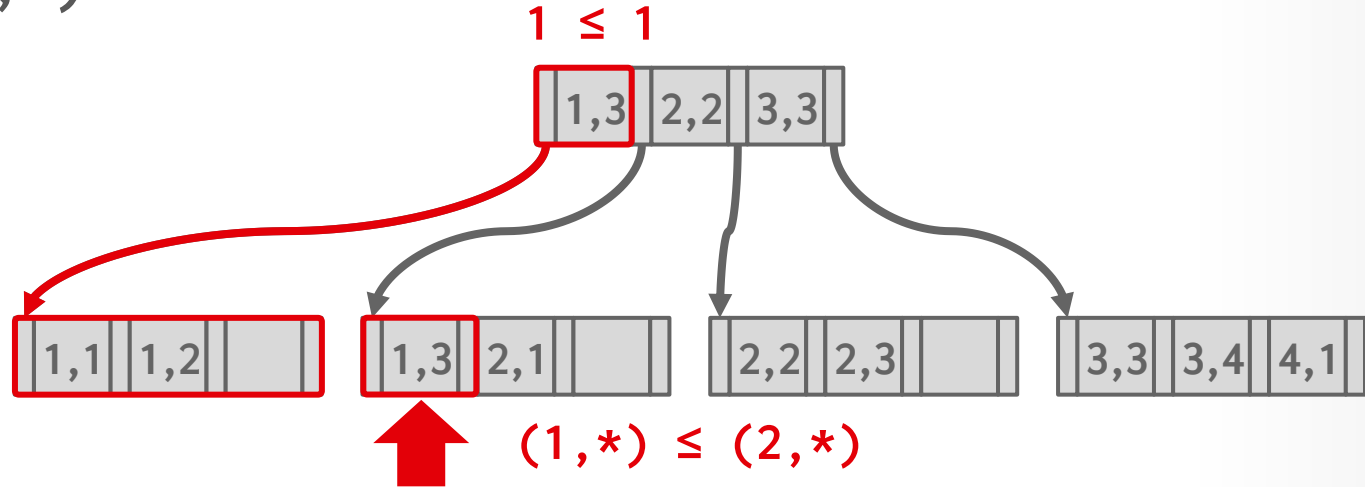
Find Key=(1,\*)



# SELECTION CONDITIONS

Find Key=(1,2)

Find Key=(1,\*)

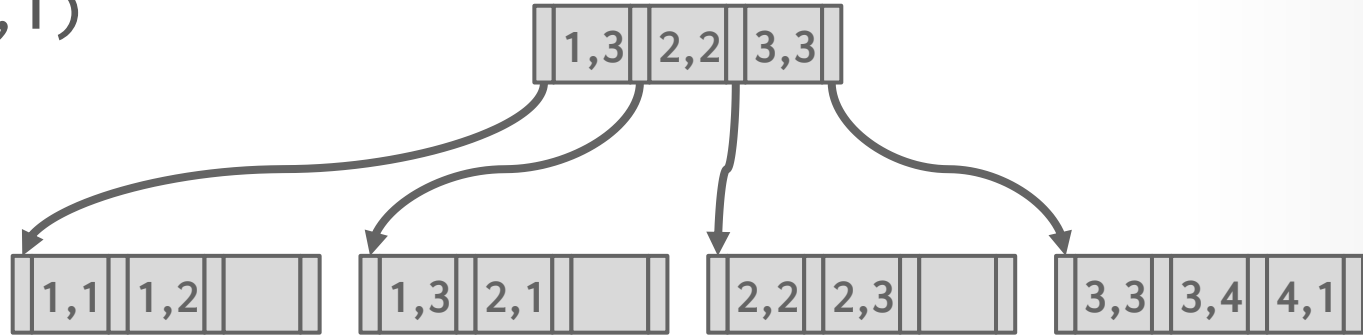


# SELECTION CONDITIONS

Find Key=(1,2)

Find Key=(1,\*)

Find Key=(\*,1)

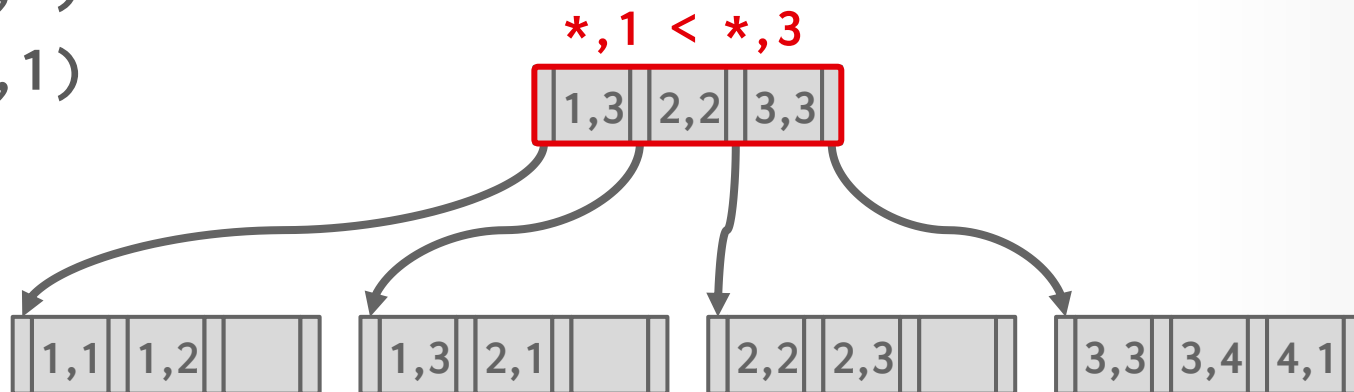


# SELECTION CONDITIONS

Find Key=(1,2)

Find Key=(1,\*)

Find Key=(\*,1)

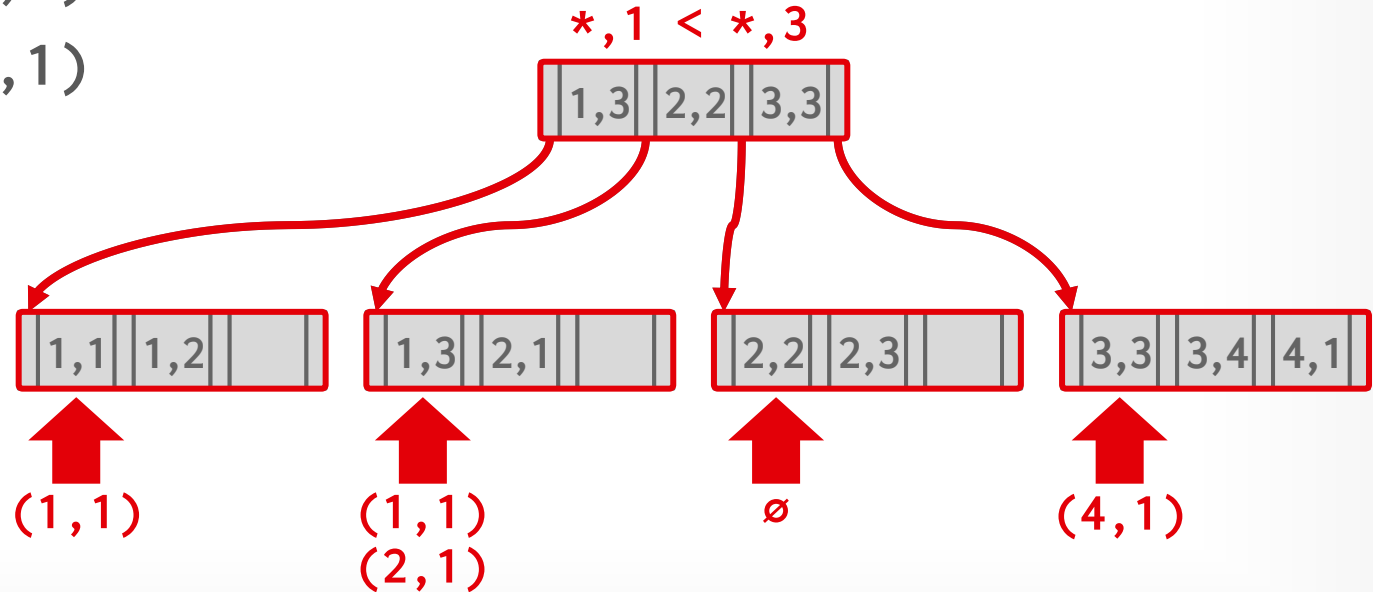


# SELECTION CONDITIONS

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.

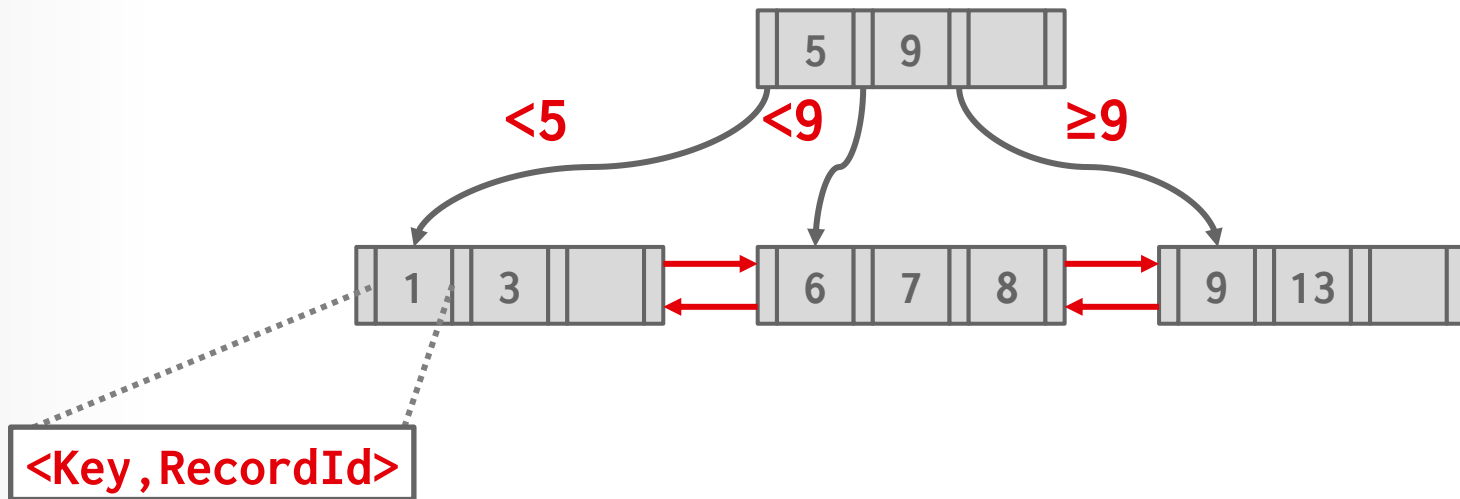


***Don't  
Do This!***

## Approach #2: Overflow Leaf Nodes

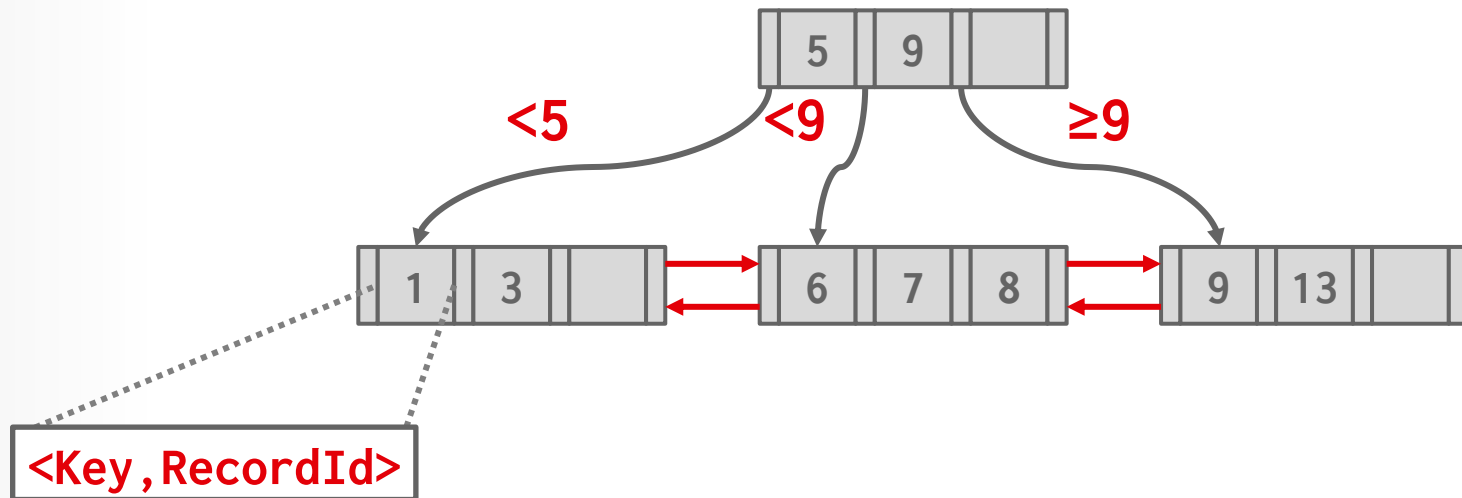
- Allow leaf nodes to spill into overflow nodes that contain the duplicate keys.
- This is more complex to maintain and modify.

# B+TREE: APPEND RECORD ID



# B+TREE: APPEND RECORD ID

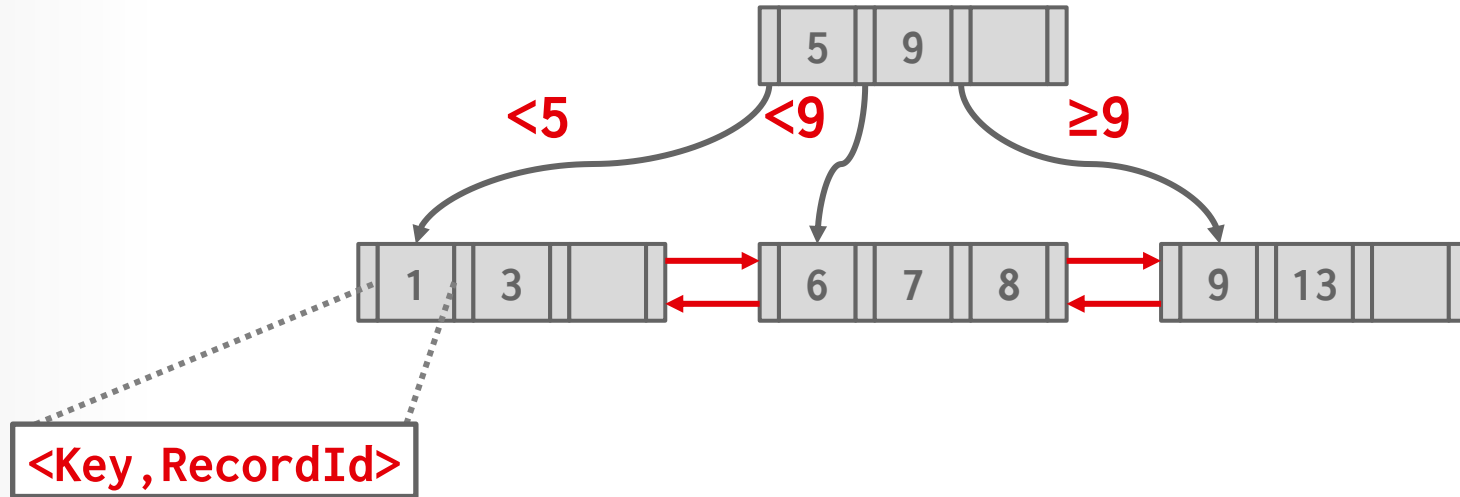
Insert 6





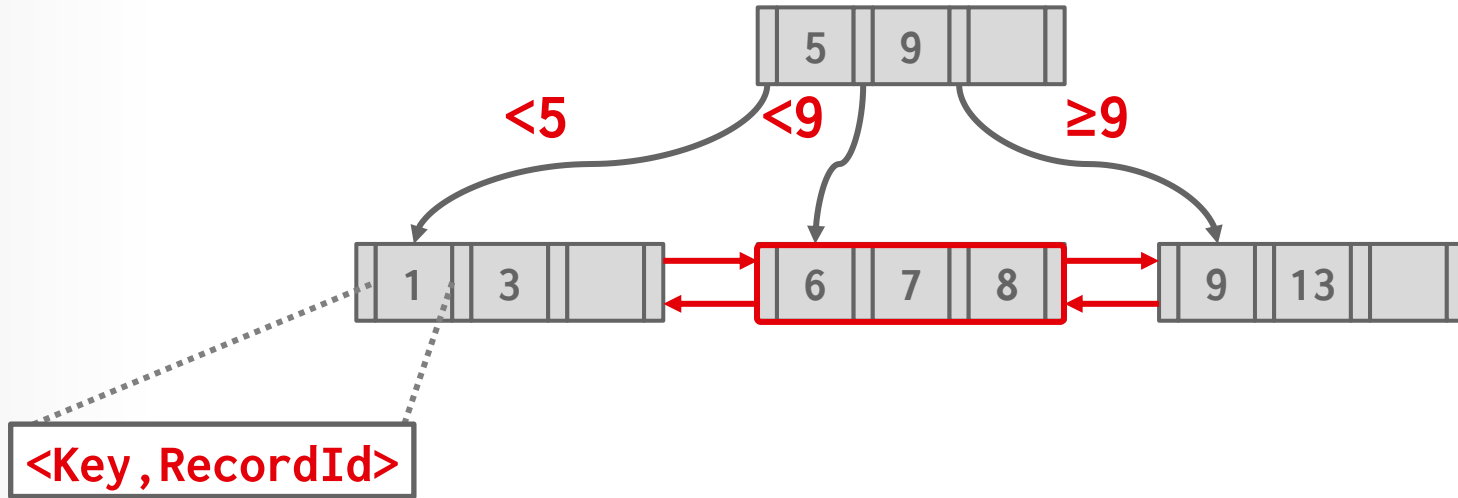
# B+TREE: APPEND RECORD ID

Insert  $\langle 6, (\text{Page}, \text{Slot}) \rangle$



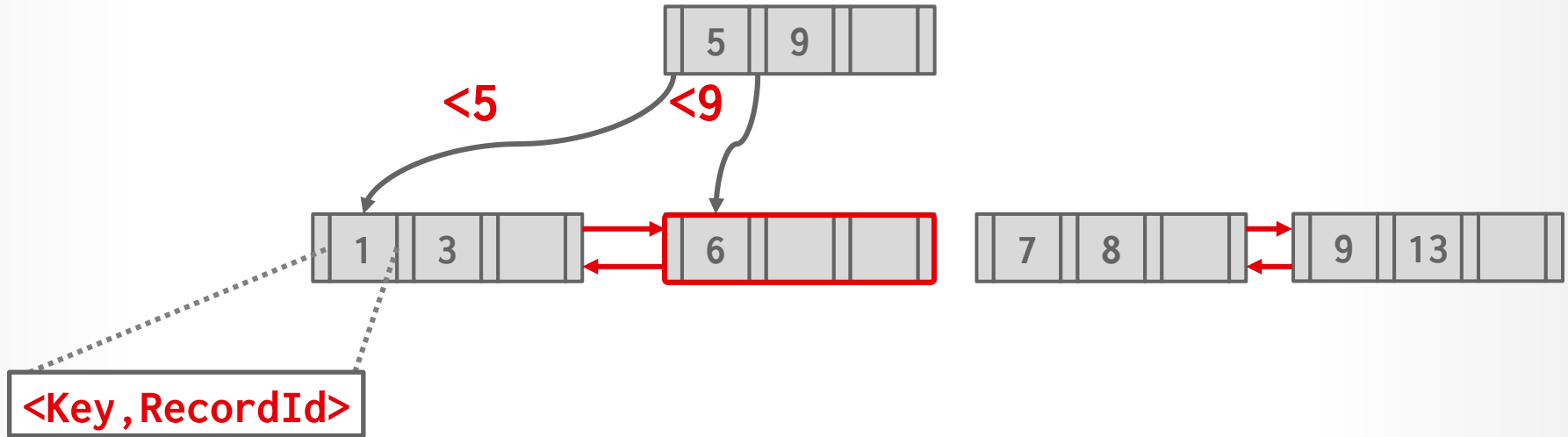
# B+TREE: APPEND RECORD ID

Insert  $\langle 6, (\text{Page}, \text{Slot}) \rangle$



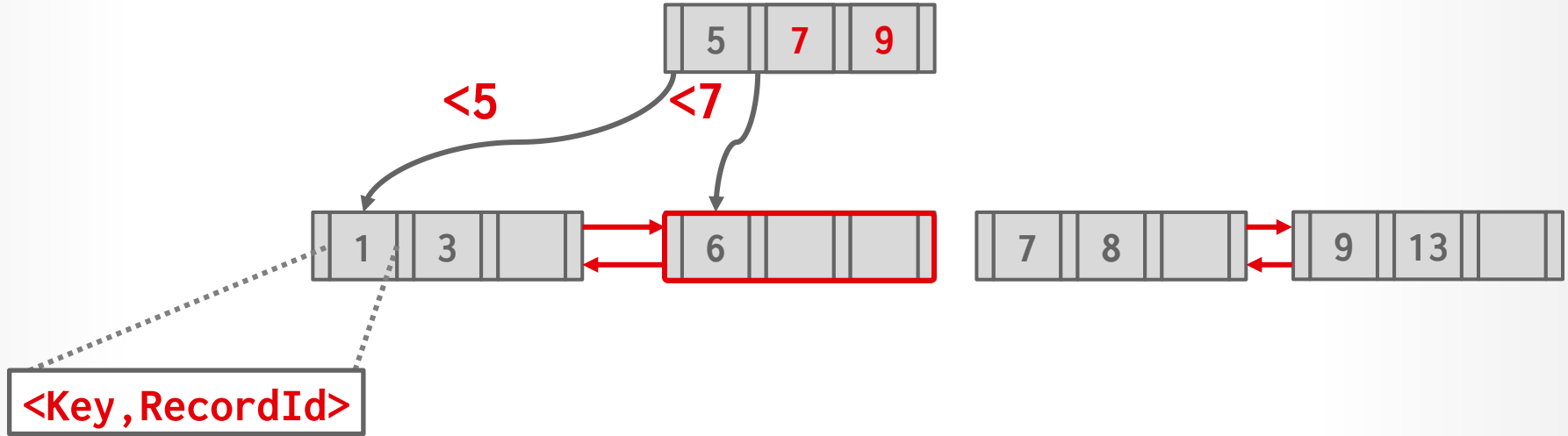
# B+TREE: APPEND RECORD ID

Insert  $\langle 6, (\text{Page}, \text{Slot}) \rangle$



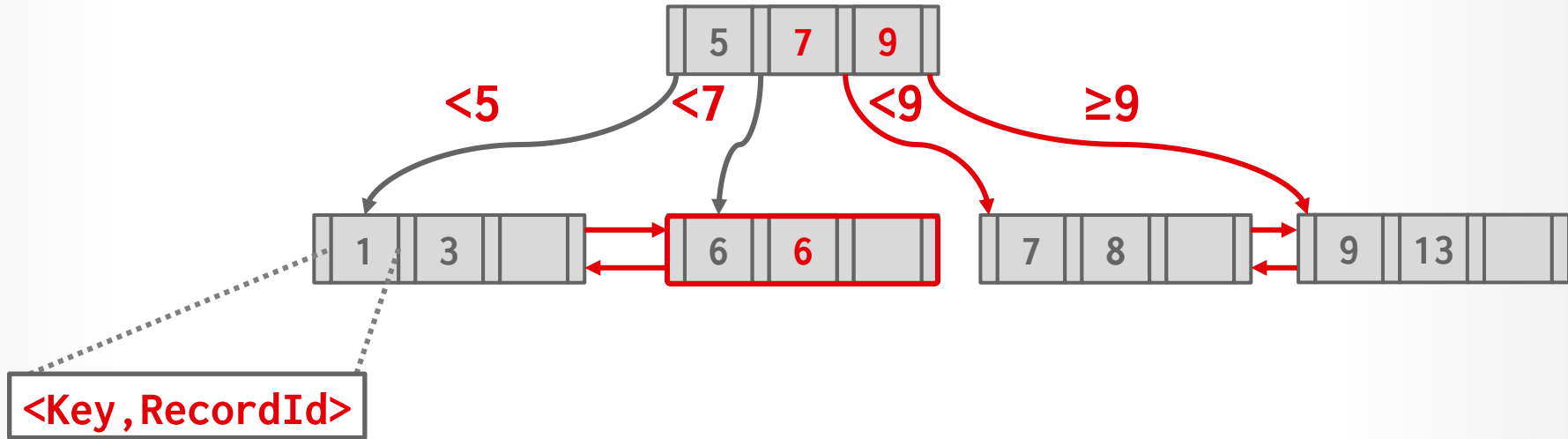
# B+TREE: APPEND RECORD ID

Insert  $\langle 6, (\text{Page}, \text{Slot}) \rangle$



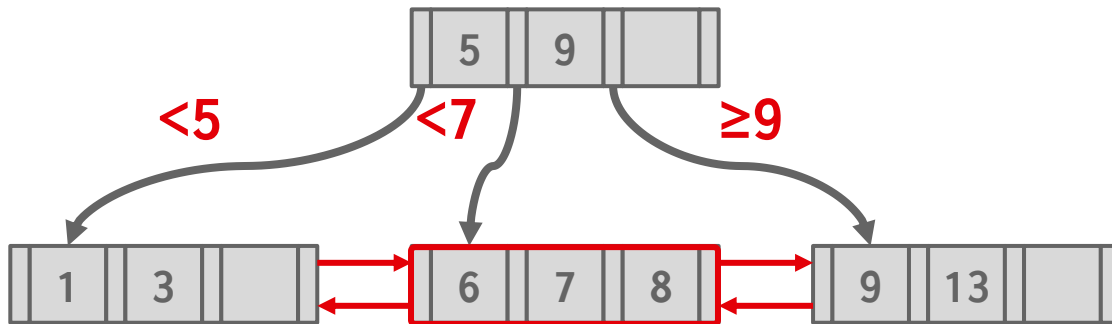
# B+TREE: APPEND RECORD ID

Insert  $\langle 6, (\text{Page}, \text{Slot}) \rangle$



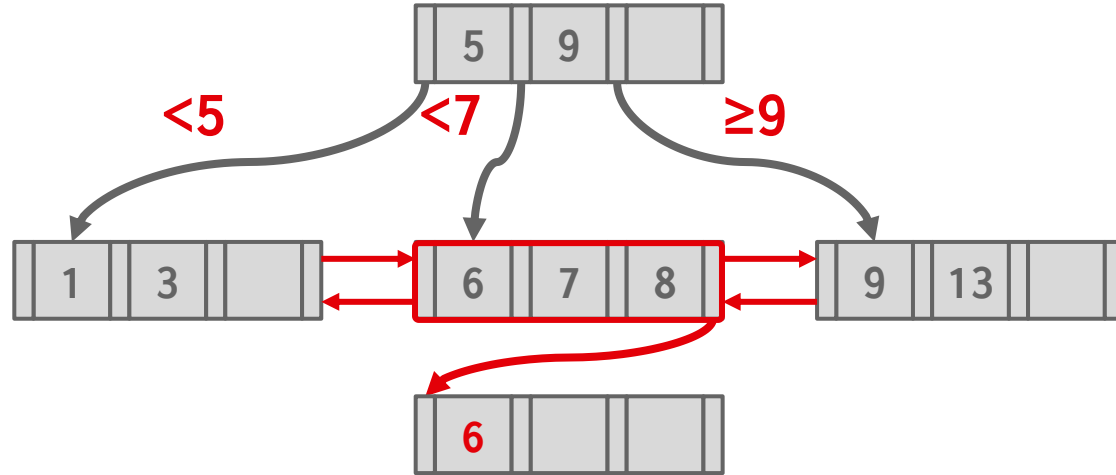
# B+TREE: OVERFLOW LEAF NODES

Insert 6



# B+TREE: OVERFLOW LEAF NODES

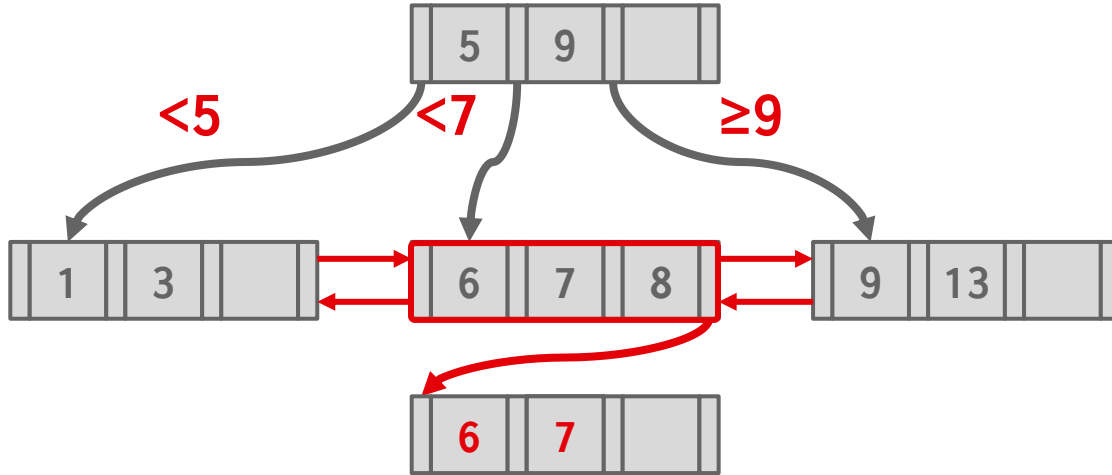
Insert 6



# B+TREE: OVERFLOW LEAF NODES

Insert 6

Insert 7



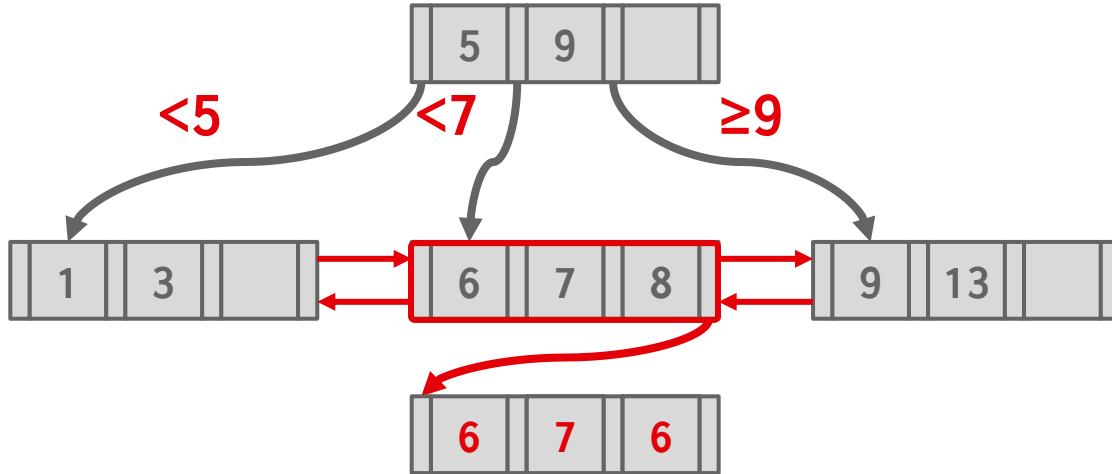


# B+TREE: OVERFLOW LEAF NODES

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.

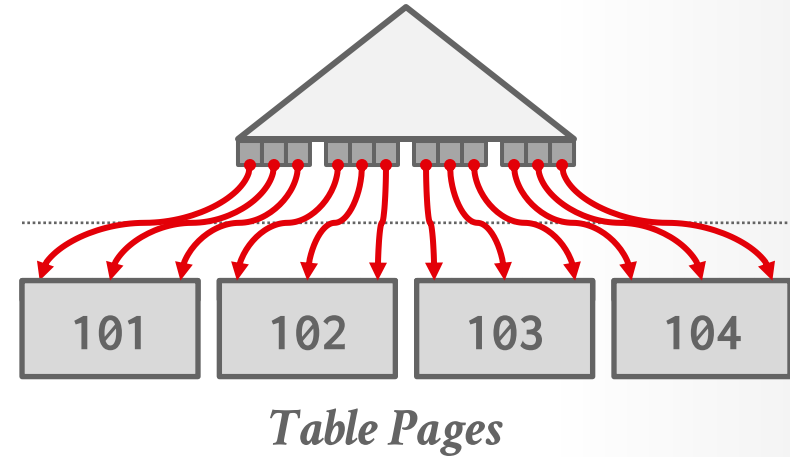
→ If a table does not contain a primary key, the DBMS will automatically make a hidden primary key.

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.

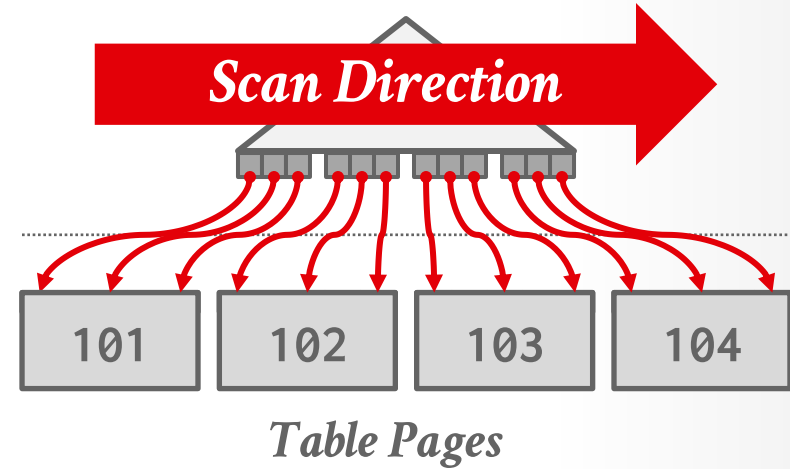
This will always be better than  
sorting data for each query.



# 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.



# 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.

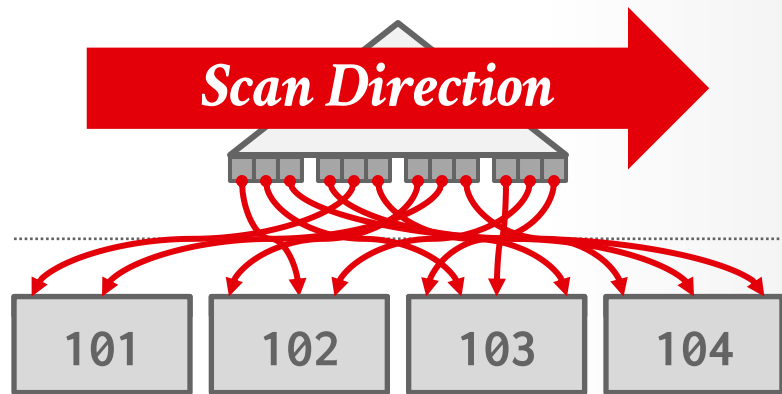


# INDEX SCAN PAGE SORTING

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.

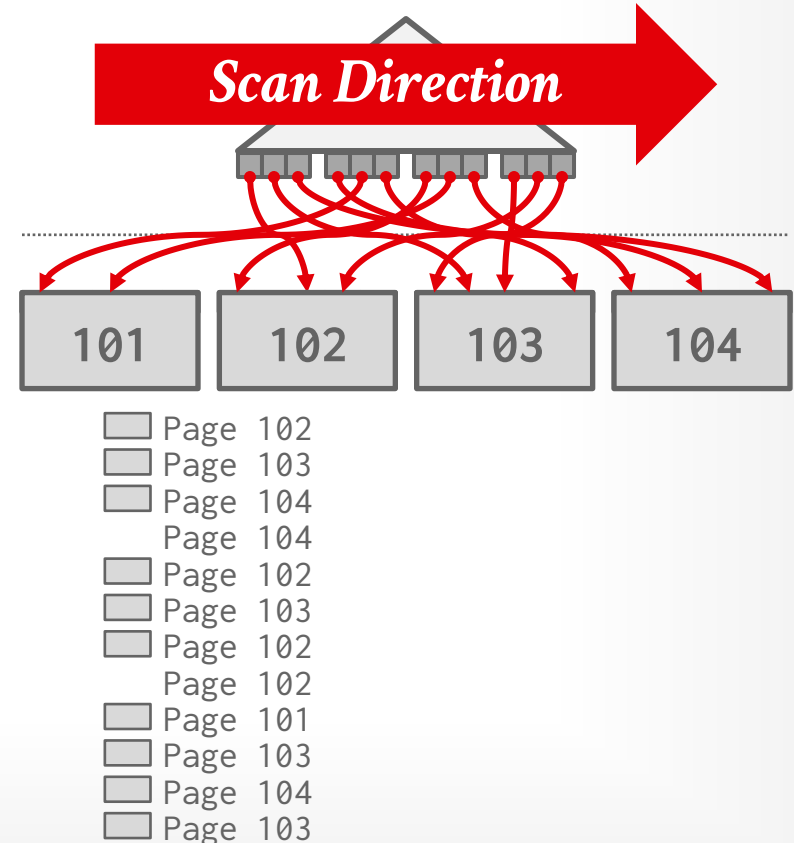


# INDEX SCAN PAGE SORTING

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.

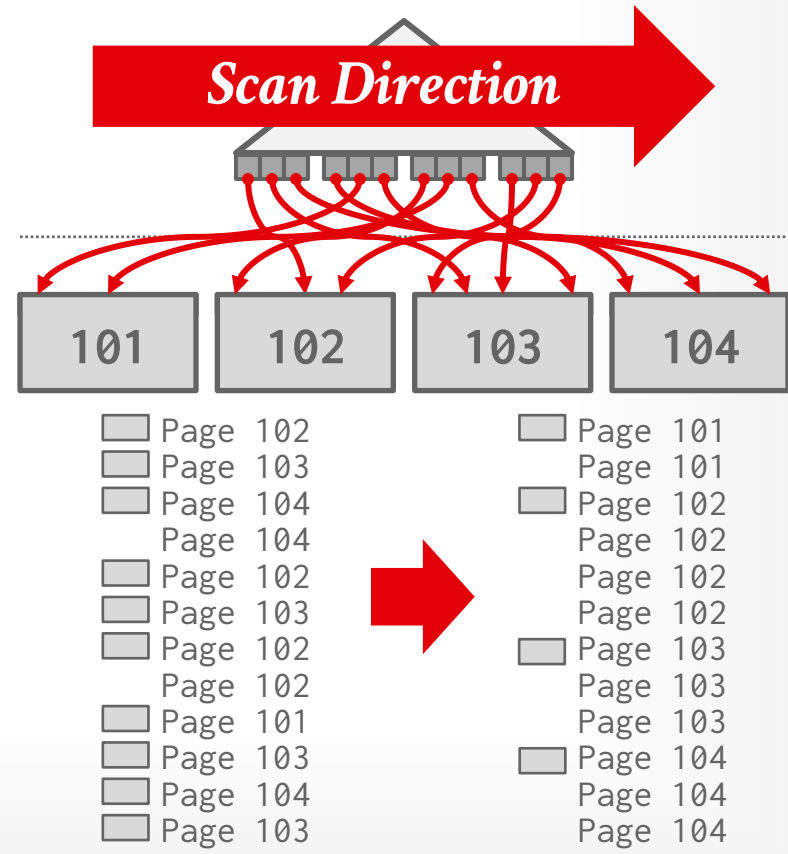
The DBMS retrieves each page once.



# 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

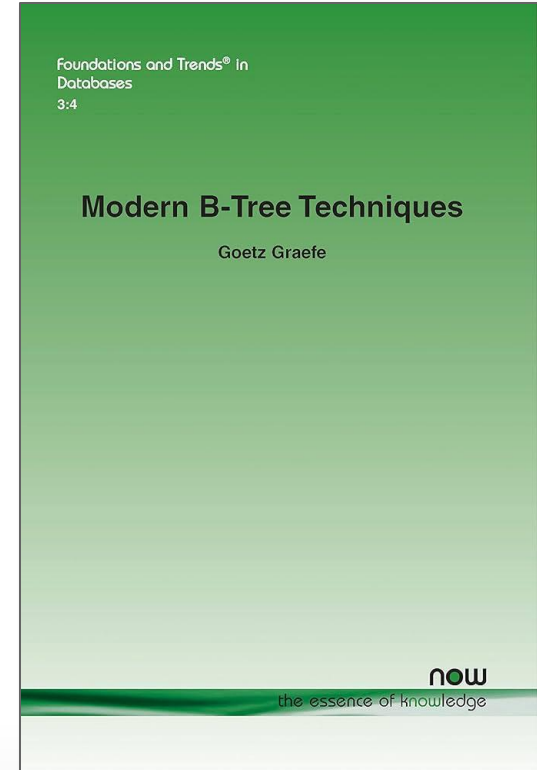
---

Node Size

Merge Threshold

Variable-Length Keys

Intra-Node Search



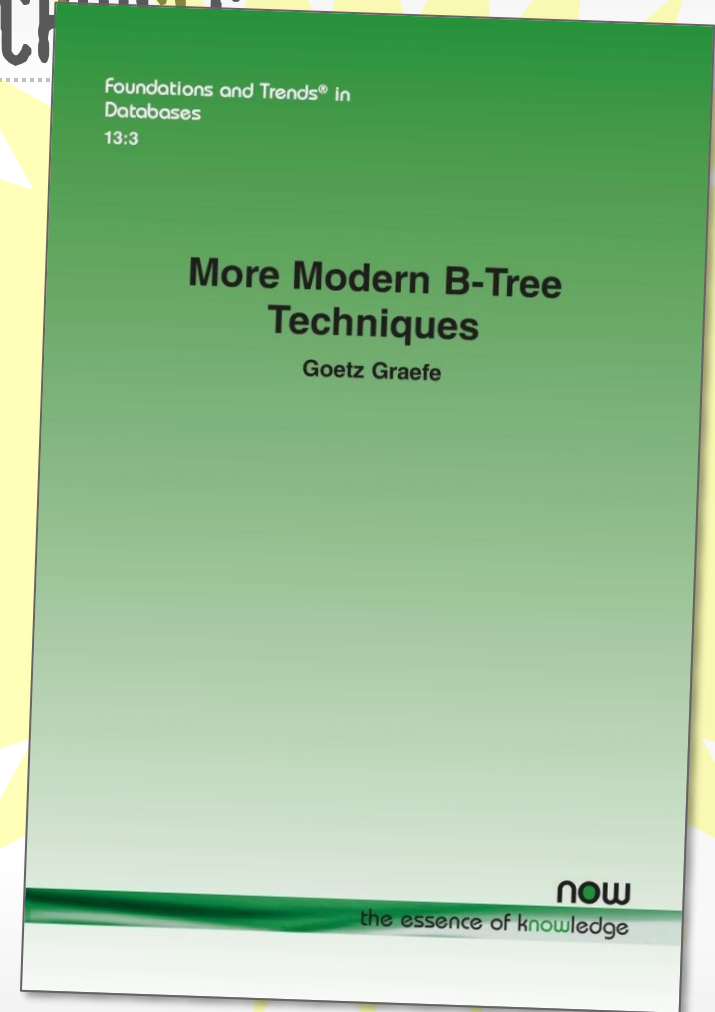
# B+TREE DESIGN CHOICES

Node Size

Merge Threshold

Variable-Length Keys

Intra-Node Search



# NODE SIZE

---



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

→ HDD: ~1MB

→ SSD: ~10KB

→ In-Memory: ~512B

Optimal sizes can vary depending on the workload

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

# MERGE THRESHOLD

---

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

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

Delaying a merge operation may reduce the amount of reorganization.

It may also be better to let underfilled 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

|   |   |   |   |   |   |    |
|---|---|---|---|---|---|----|
| 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|----|



# INTRA-NODE SEARCH

## Approach #1: Linear

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

Find Key=8

|   |   |   |   |   |   |    |
|---|---|---|---|---|---|----|
| 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|----|

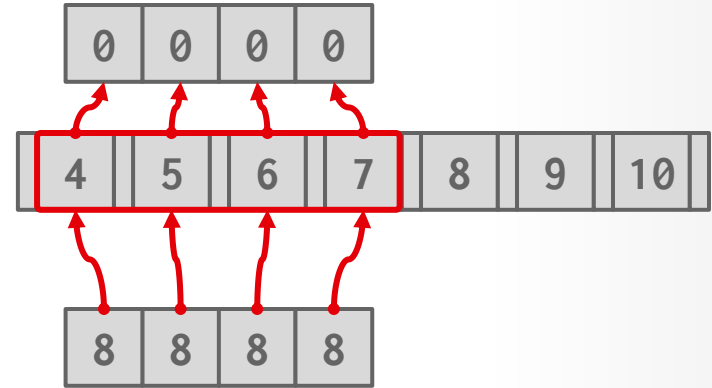
```
_mm_cmpeq_epi32_mask(a, b)
```



# INTRA-NODE SEARCH

## Approach #1: Linear

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

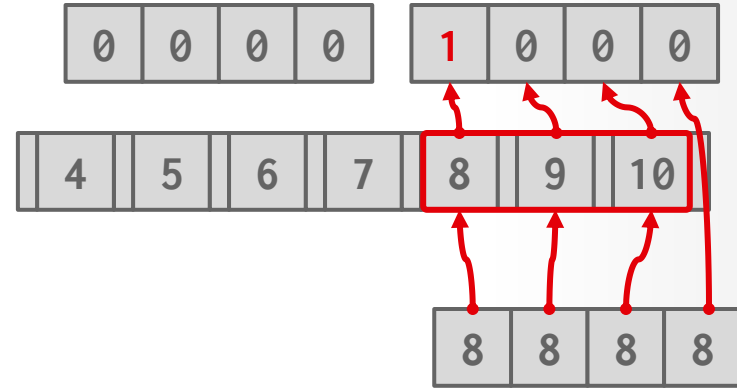


```
_mm_cmpeq_epi32_mask(a, b)
```

# INTRA-NODE SEARCH

## Approach #1: Linear

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



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

# INTRA-NODE SEARCH

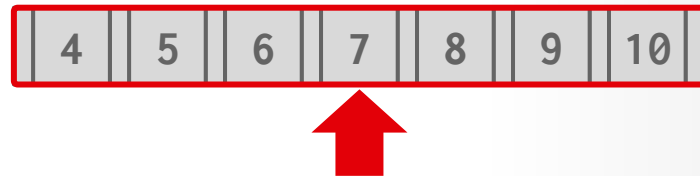
## Approach #1: Linear

- 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.

Find Key=8



# 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.



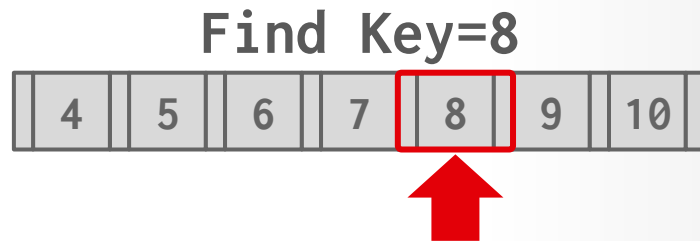
# INTRA-NODE SEARCH

## 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.

A

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

$$\text{Offset: } (8-4) \times 7 / (10-4) = 4$$

|   |   |   |   |   |   |    |
|---|---|---|---|---|---|----|
| 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|----|



# OPTIMIZATIONS

---

Pointer Swizzling

Buffered Updates (*B~~E~~-trees*)

Partial Indexes

Include Columns

Prefix Compression

Deduplication

Suffix Truncation

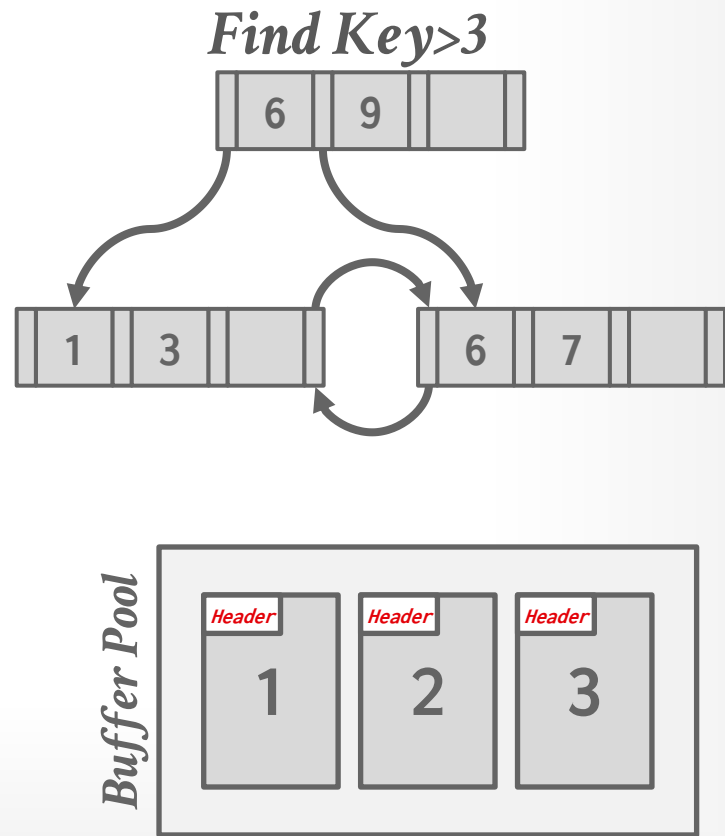
Bulk Insert

Many more...

# 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.

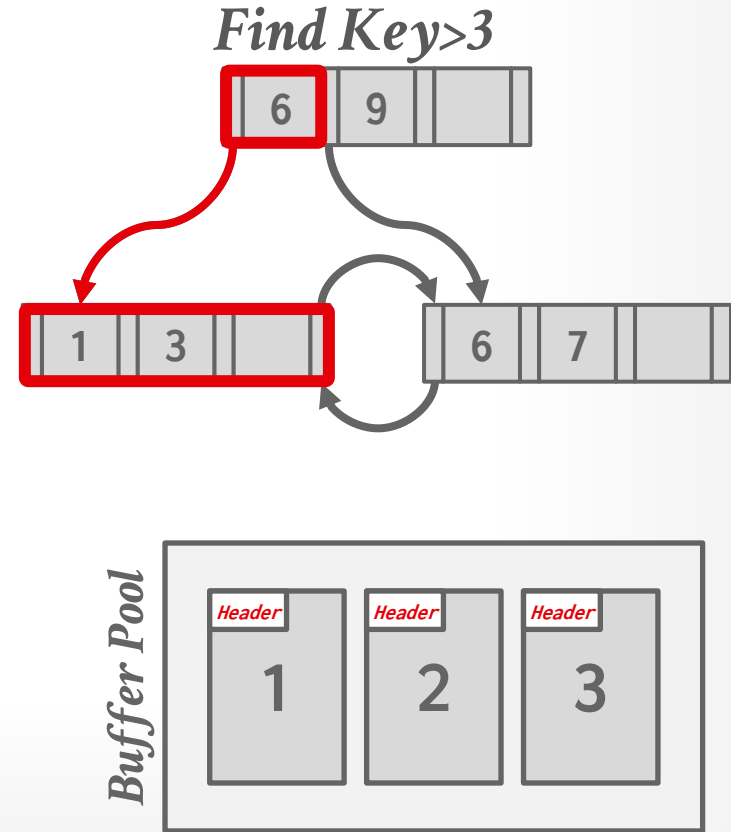




# POINTER SWIZZLING

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

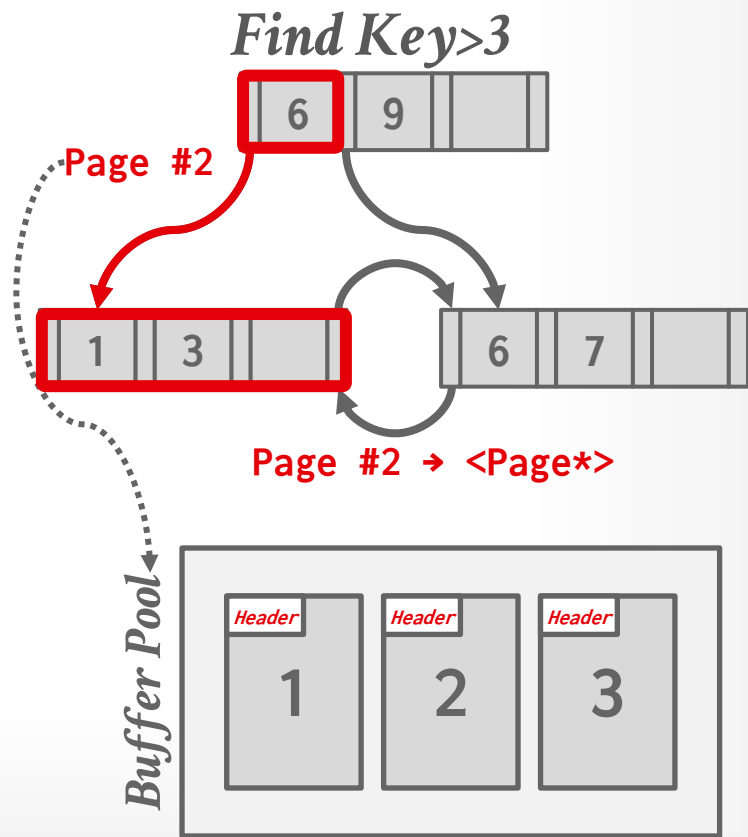
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



# 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.

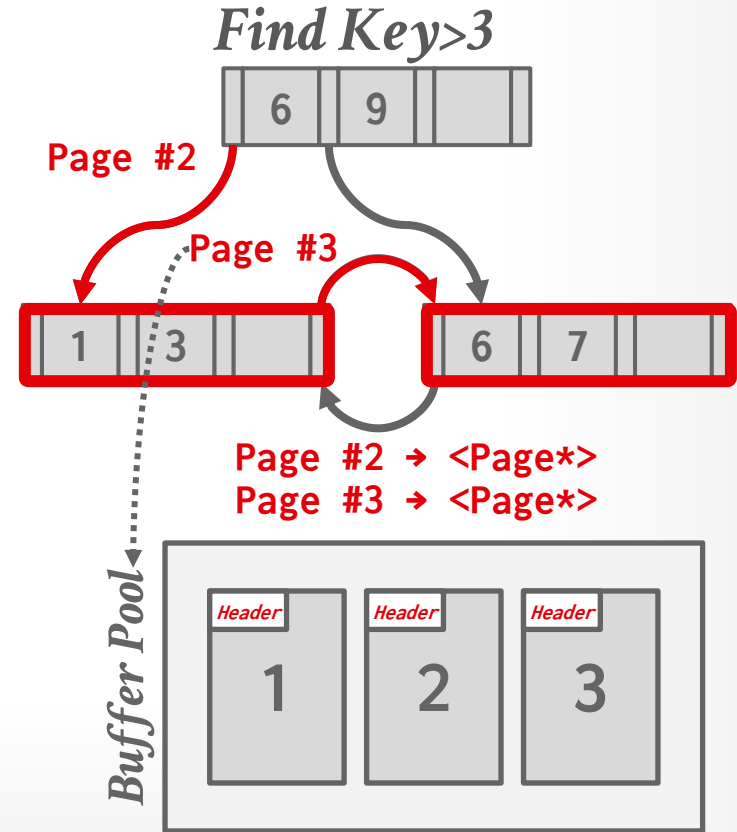
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



# 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.

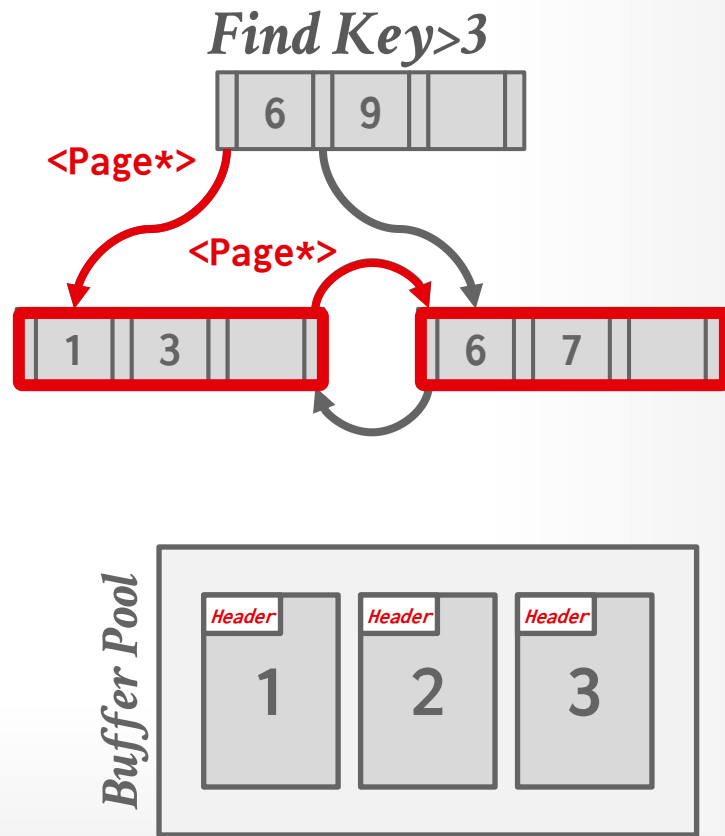
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



# 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.



# 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.

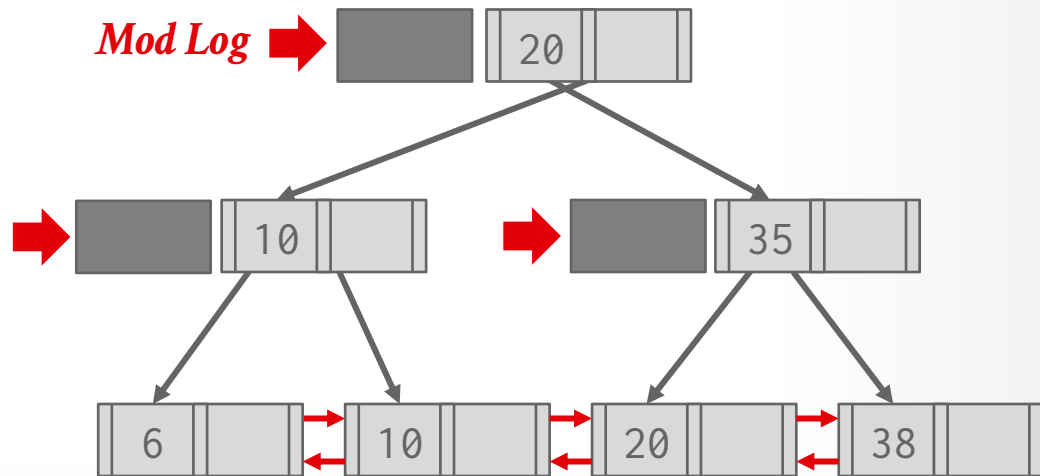
What if there was a way to delay updates and then apply multiple changes together in a batch?

# WRITE-OPTIMIZED B+TREE

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

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

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



**Tokutek**  **SPLINTERDB**

 **RelationalAI**  **ChromoDB**

**STSDb**

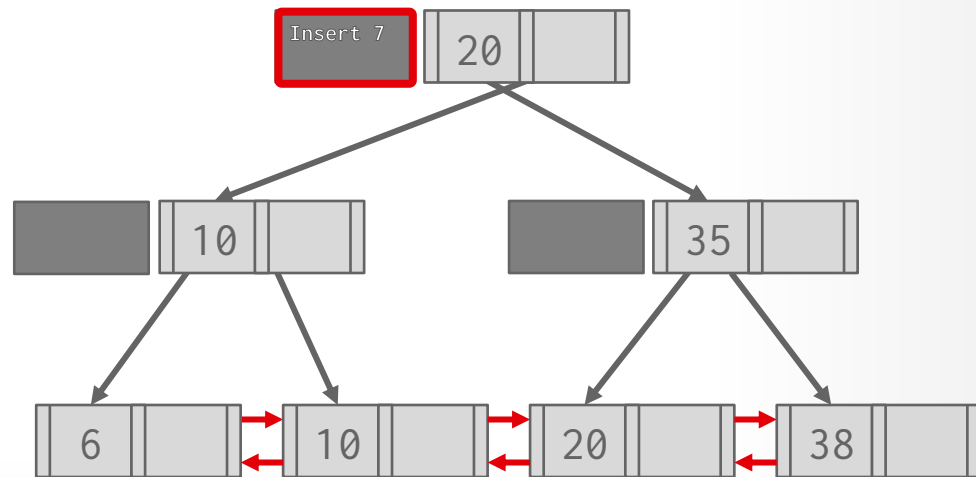
# WRITE-OPTIMIZED B+TREE

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

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

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

Insert 7



**Tokutek**  SPLINTERDB

 **RelationalAI**  **ChromoDB**

**STSDb**

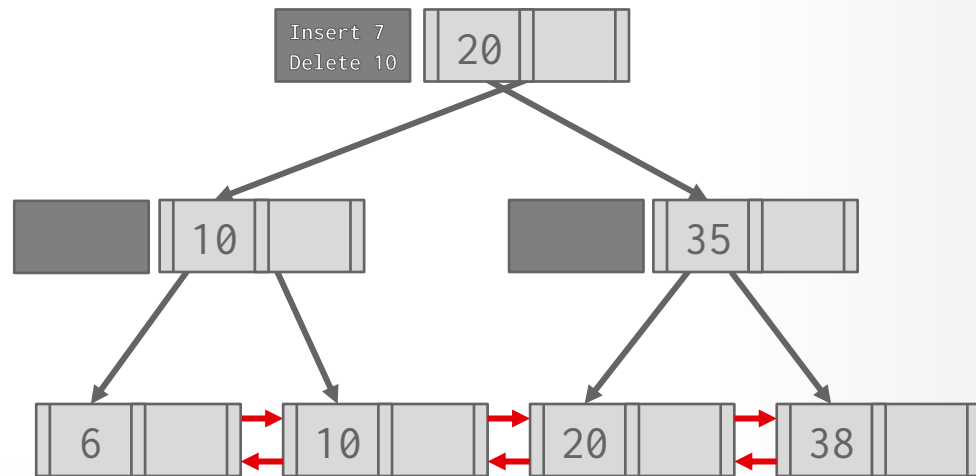
# WRITE-OPTIMIZED B+TREE

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

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

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

Insert 7  
Delete 10



**Tokutek**  **SPLINTERDB**

 **RelationalAI**  **ChromoDB**

**STSDb**



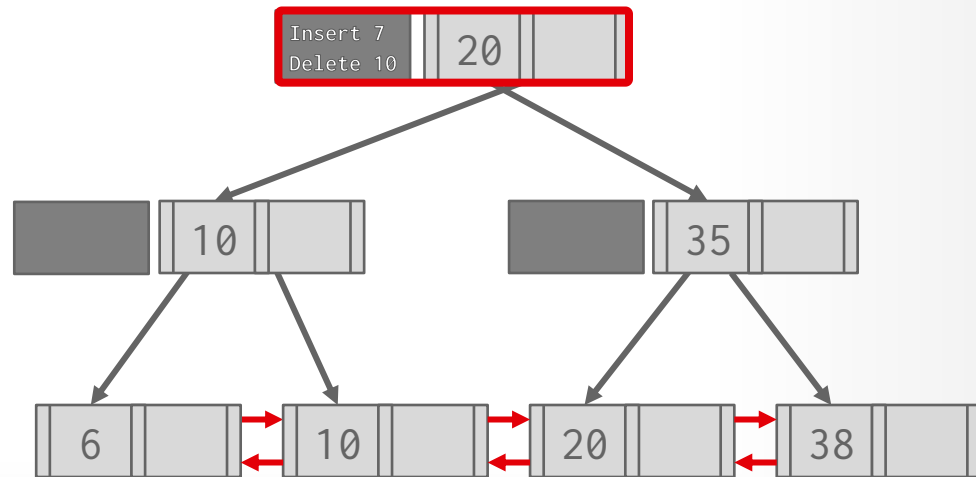
# WRITE-OPTIMIZED B+TREE

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

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

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

Find 10



**Tokutek**  SPLINTERDB

 **RelationalAI**  **ChromoDB**

**STSDb**

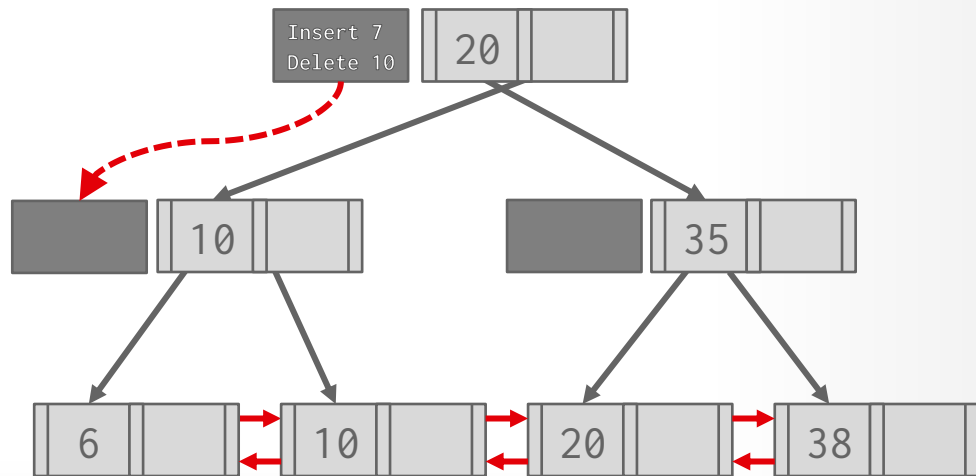
# WRITE-OPTIMIZED B+TREE

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

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

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

Insert 40



**Tokutek**  **SPLINTERDB**

 **RelationalAI**  **ChromoDB**

**STSDb**

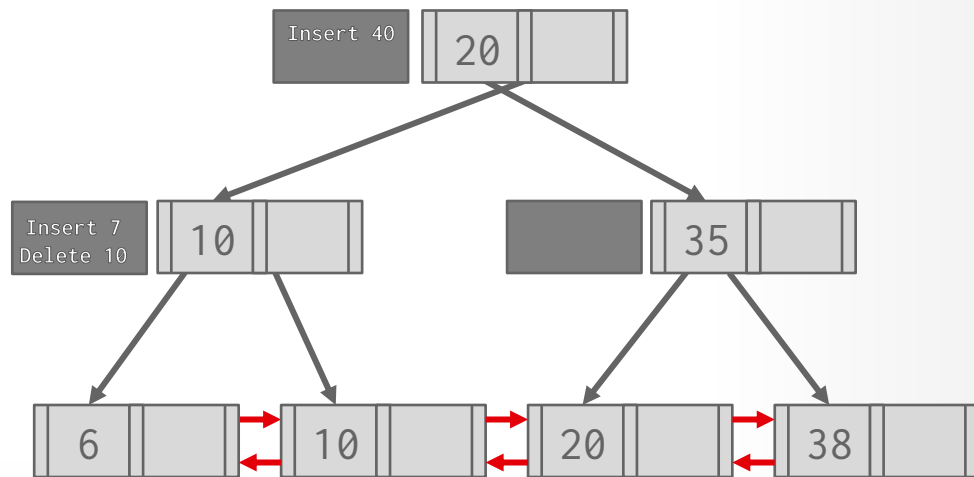
# WRITE-OPTIMIZED B+TREE

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

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

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

Insert 40



**Tokutek**  SPLINTERDB

 **RelationalAI**  **ChromoDB**

**STSDb**

# PARTIAL INDEXES

---

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

```
CREATE INDEX idx_foo  
      ON foo (a, b)  
      WHERE c = 'WuTang';
```

One common use case is to partition indexes by date ranges.

→ Create a separate index per month, year.

# PARTIAL INDEXES

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

```
CREATE INDEX idx_foo  
ON foo (a, b)  
WHERE c = 'WuTang';
```

One common use case is to partition indexes by date ranges.

→ Create a separate index per month, year.

# PARTIAL INDEXES

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

One common use case is to partition indexes by date ranges.  
→ Create a separate index per month, year.

```
CREATE INDEX idx_foo  
ON foo (a, b)  
WHERE c = 'WuTang';
```

```
SELECT b FROM foo  
WHERE a = 123  
AND c = 'WuTang';
```

# PARTIAL INDEXES

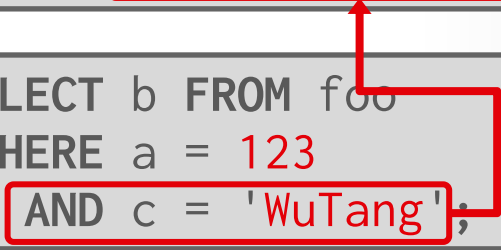
Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

One common use case is to partition indexes by date ranges.

→ Create a separate index per month, year.

```
CREATE INDEX idx_foo  
ON foo (a, b)  
WHERE c = 'WuTang';
```

```
SELECT b FROM foo  
WHERE a = 123  
AND c = 'WuTang';
```



# PARTIAL INDEXES

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

One common use case is to partition indexes by date ranges.

→ Create a separate index per month, year.

```
CREATE INDEX idx_foo  
ON foo(a, b)  
WHERE c = 'WuTang';
```

```
SELECT b FROM foo  
WHERE a = 123
```



# INDEX INCLUDE COLUMNS

---

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```
CREATE INDEX idx_foo  
      ON foo (a, b)  
      INCLUDE (c);
```

# INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```
CREATE INDEX idx_foo  
      ON foo (a, b)  
      INCLUDE (c);
```

```
SELECT b FROM foo  
WHERE a = 123  
      AND c = 'WuTang';
```

# INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```
CREATE INDEX idx_foo  
ON foo(a, b)  
INCLUDE (c);
```

```
SELECT b FROM foo  
WHERE a = 123  
AND c = 'WuTang';
```



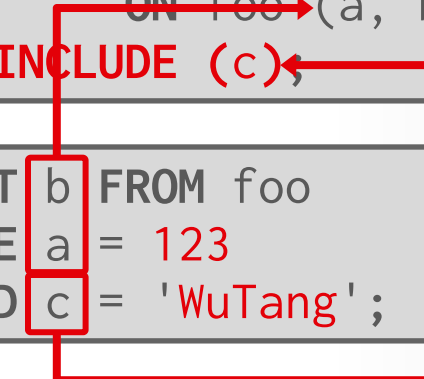
# INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

```
CREATE INDEX idx_foo  
ON foo(a, b)  
INCLUDE (c);
```

```
SELECT b FROM foo  
WHERE a = 123  
AND c = 'WuTang';
```



# INDEX INCLUDE COLUMNS

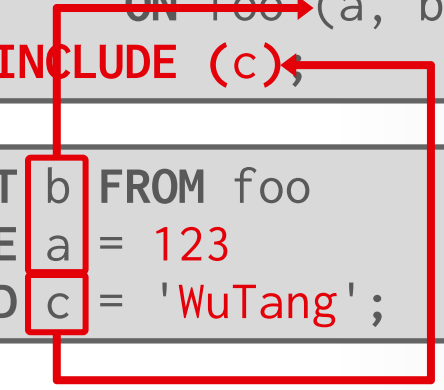
Embed additional columns in indexes to support index-only queries.

These extra columns are only stored in the leaf nodes and are not part of the search key.

If all attributes a query needs are available in an index, then the DBMS does not need to retrieve the tuple.  
→ AKA covering index or index-only scans.

```
CREATE INDEX idx_foo  
ON foo(a, b)  
INCLUDE (c);
```

```
SELECT b FROM foo  
WHERE a = 123  
AND c = 'WuTang';
```



# 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.

|  |        |         |       |  |
|--|--------|---------|-------|--|
|  | robbed | robbing | robot |  |
|--|--------|---------|-------|--|



|                    |     |      |    |
|--------------------|-----|------|----|
| <i>Prefix: rob</i> |     |      |    |
|                    | bed | bing | ot |

# DEDUPLICATION

---

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

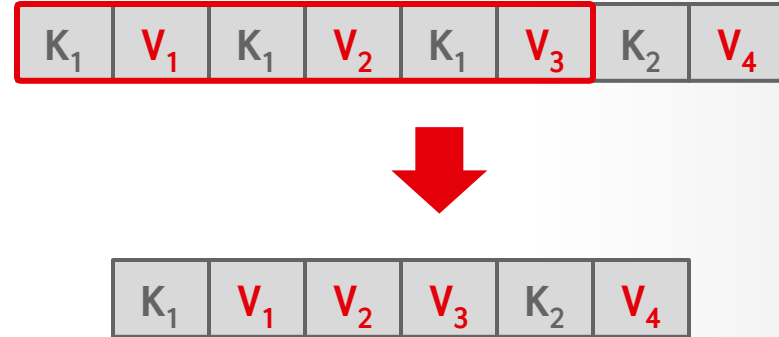
|       |       |       |       |       |       |       |       |
|-------|-------|-------|-------|-------|-------|-------|-------|
| $K_1$ | $V_1$ | $K_1$ | $V_2$ | $K_1$ | $V_3$ | $K_2$ | $V_4$ |
|-------|-------|-------|-------|-------|-------|-------|-------|

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).



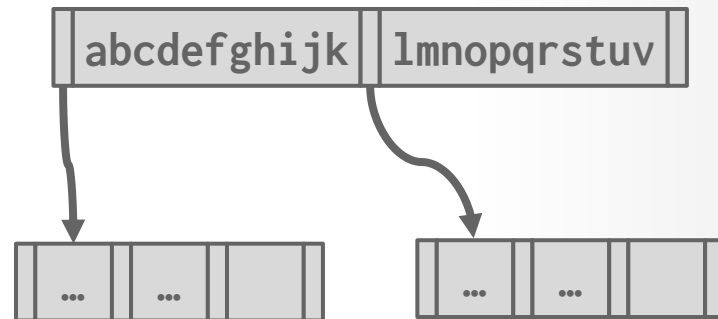


# SUFFIX TRUNCATION

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.

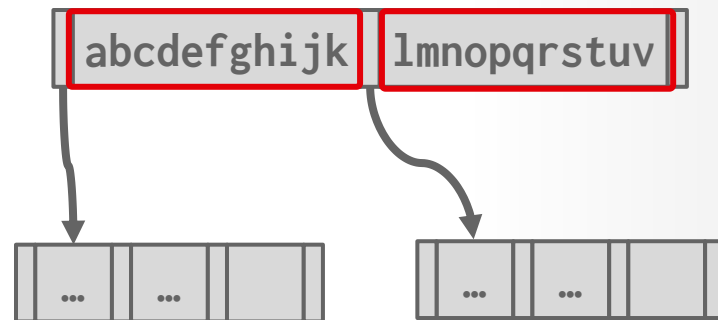


# SUFFIX TRUNCATION

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.

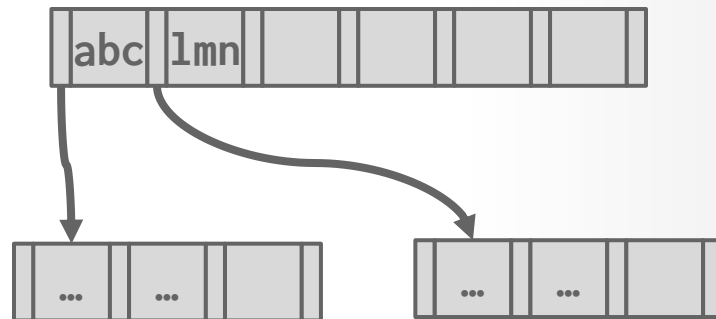


# SUFFIX TRUNCATION

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.

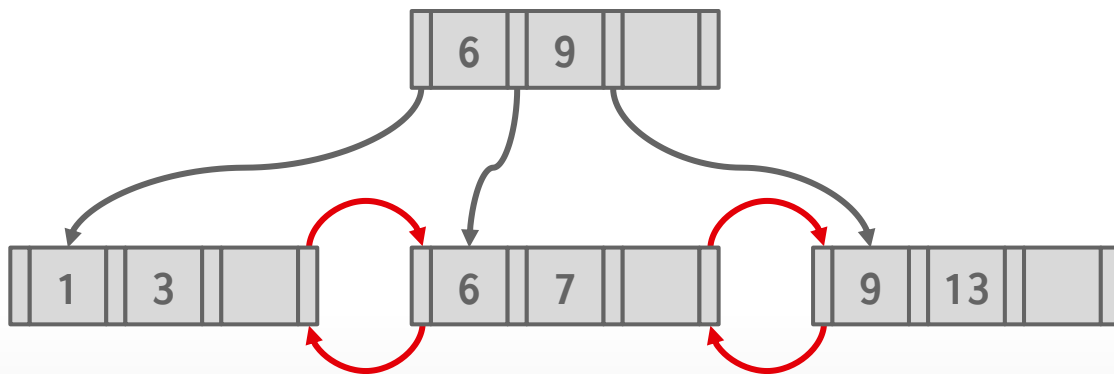


# BULK INSERT

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**



# CONCLUSION

---

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

# NEXT CLASS

---

Bloom Filters

Tries / Radix Trees / Patricia Trees

Skip Lists

Inverted Indexes

Vector Indexes