Georgia Tech

DATABASE SYSTEM IMPLEMENTATION

GT 4420/6422 // SPRING 2019 // @JOY_ARULRAJ

LECTURE #12: OLTP INDEXES (PART II)

CREATING THE NEXT®

TODAY'S AGENDA

B+Tree Overview

Index Implementation Issues

ART Index

LOGISTICS

Reminder: Problem set due on Feb 21st.

Reminder: Mid-term Exam on Feb 26th.

Reminder: Project Proposals due on Feb 28th.

B+TREE

A **B+Tree** is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in **O(log n)**.

→ Generalization of a binary search tree in that a node can have more than two children.

→ Optimized for systems that read and write large blocks of data.

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

Computing Surveys, Vol 11, No 2, June 1979

B+TREE PROPERTIES

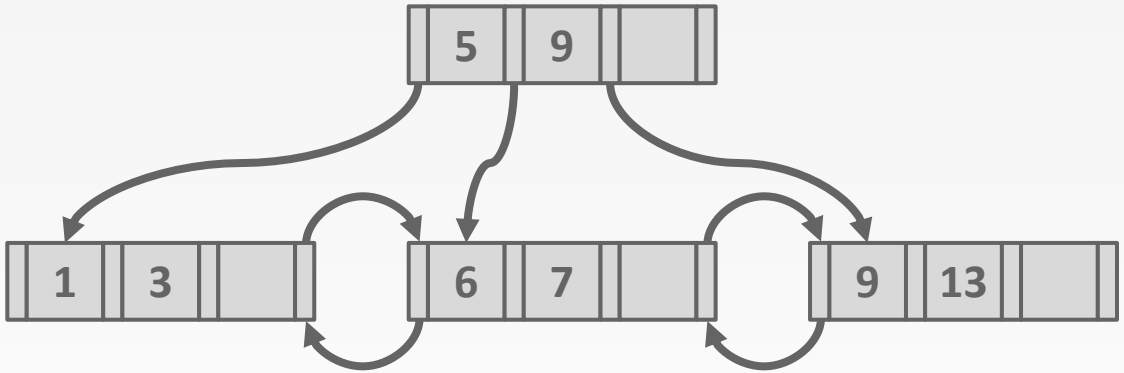
A B+tree is an ***M***-way search tree with the following properties:

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

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

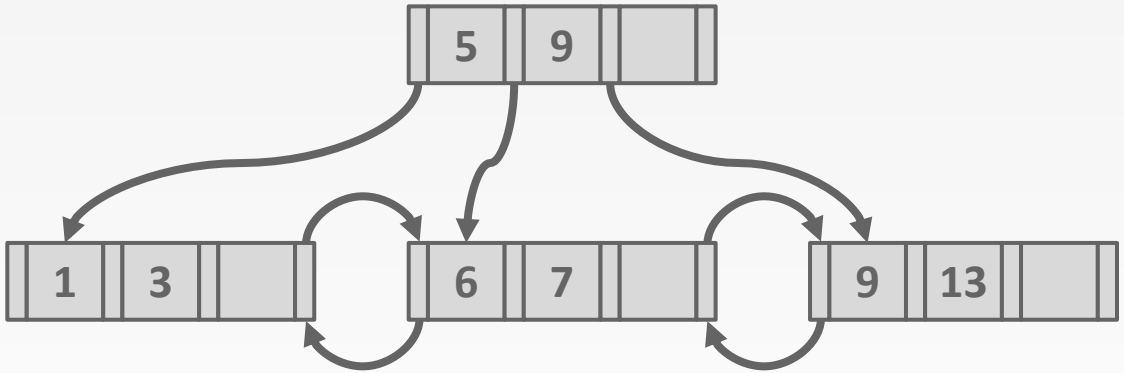
→ Every inner node with **k** keys has **k+1** non-null children

B+TREE EXAMPLE

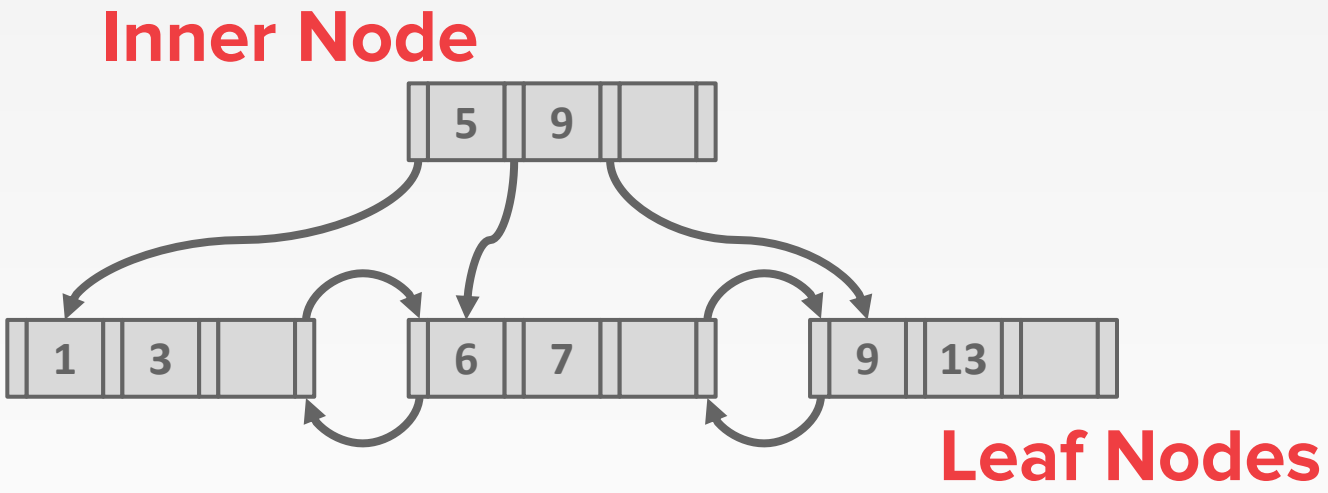


B+TREE EXAMPLE

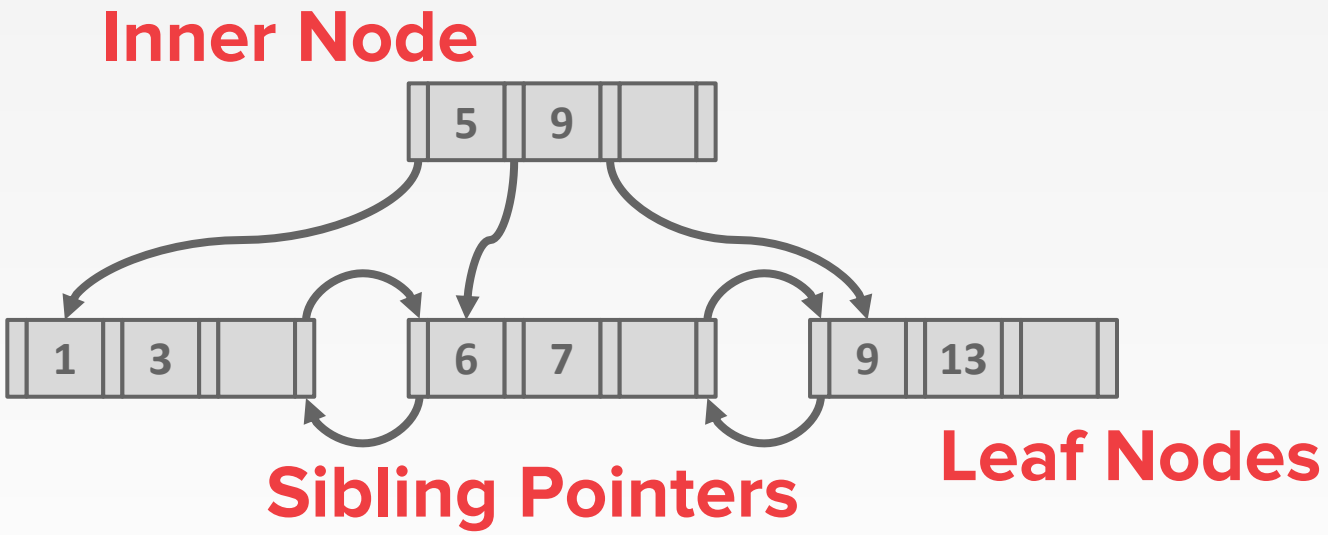
Inner Node



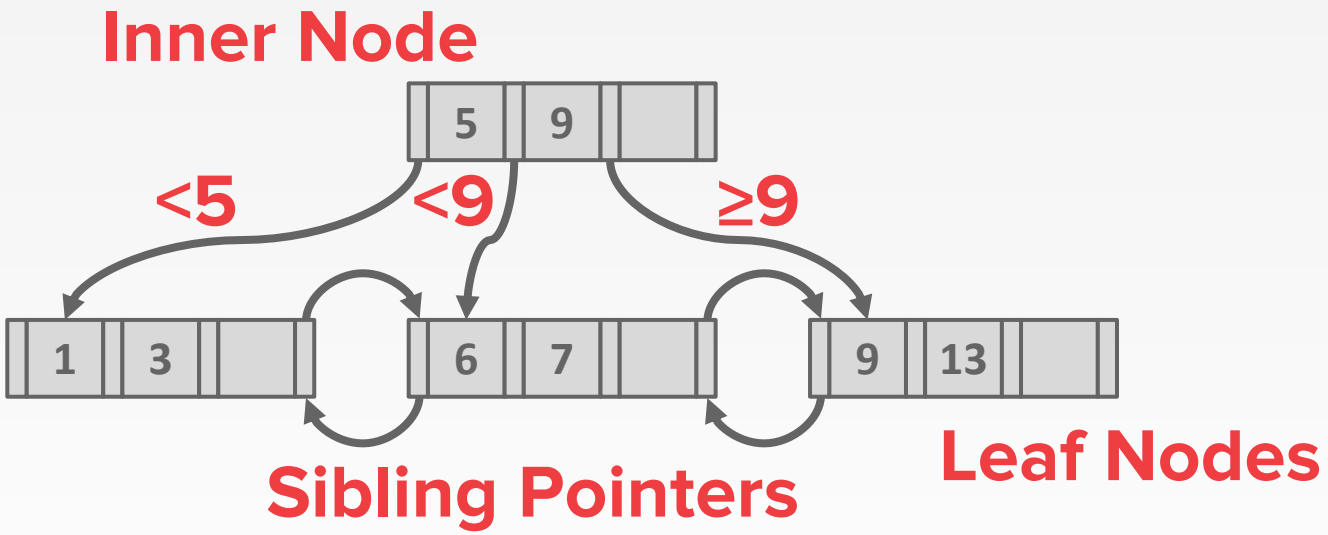
B+TREE EXAMPLE



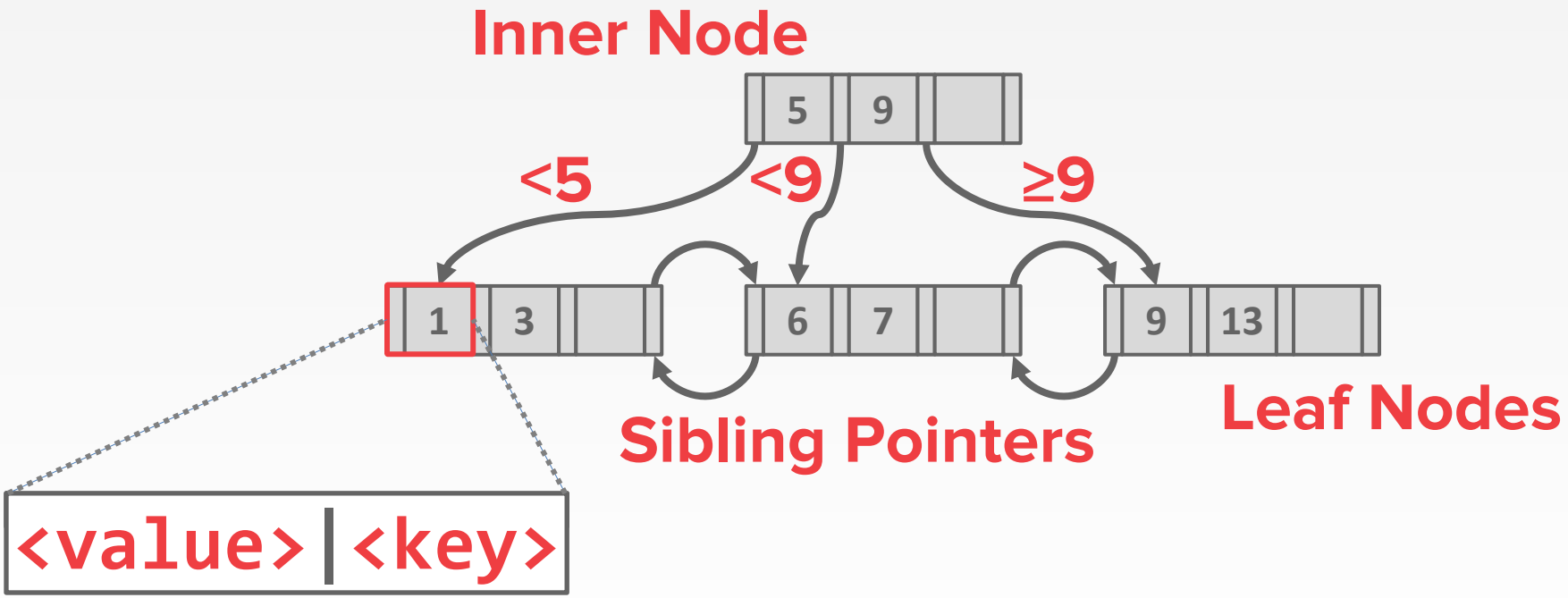
B+TREE EXAMPLE



B+TREE EXAMPLE



B+TREE EXAMPLE



NODES

Every node in the B+Tree contains an array of key/value pairs.

→ The keys will always be the column or columns that you built your index on

→ The values will differ based on whether the node is classified as **inner nodes** or **leaf nodes.**

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

LEAF NODE VALUES

Approach #1: Record Ids

→ A pointer to the location of the tuple that the index entry corresponds to.

Approach #2: Tuple Data

→ The actual contents of the tuple is stored in the leaf node.
→ Secondary indexes have to store the record id as their values.

LEAF NODE VALUES

Approach #1: Record Ids

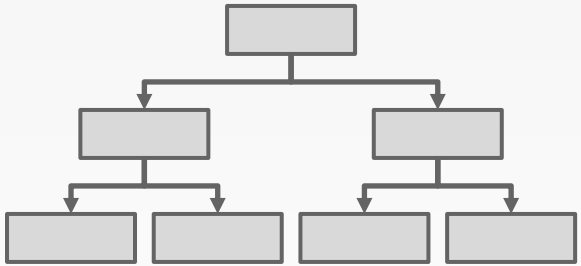
→ A pointer to the location of the tuple that the index entry corresponds to.

Approach #2: Tuple Data

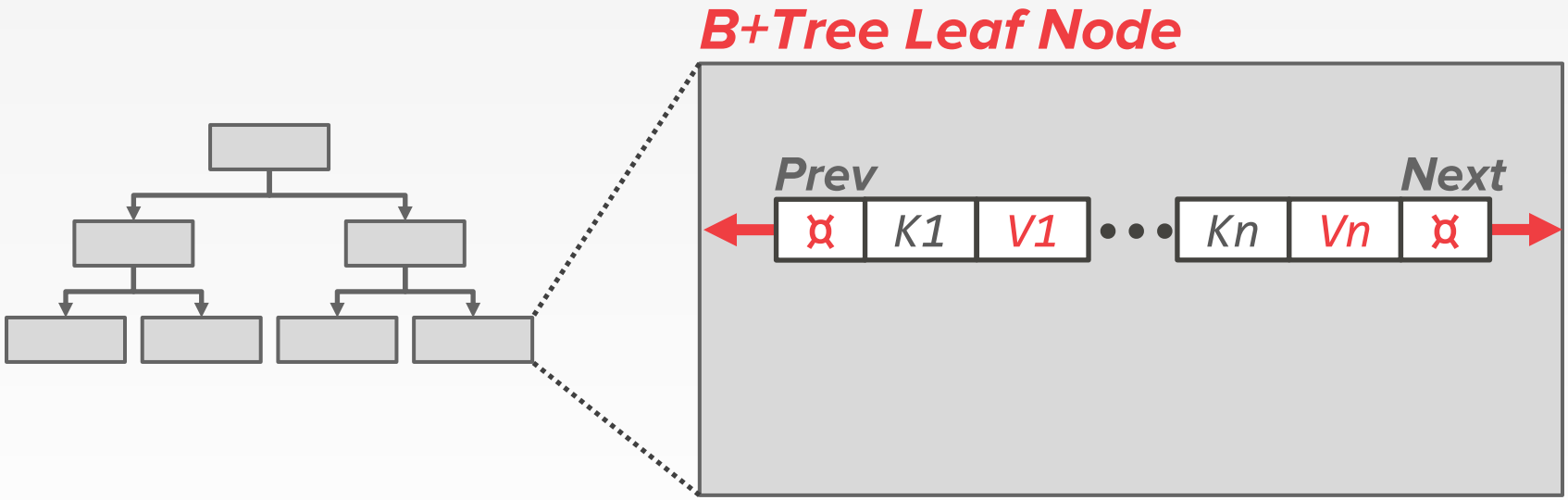
→ The actual contents of the tuple is stored in the leaf node.

→ Secondary indexes have to store the record id as their values.

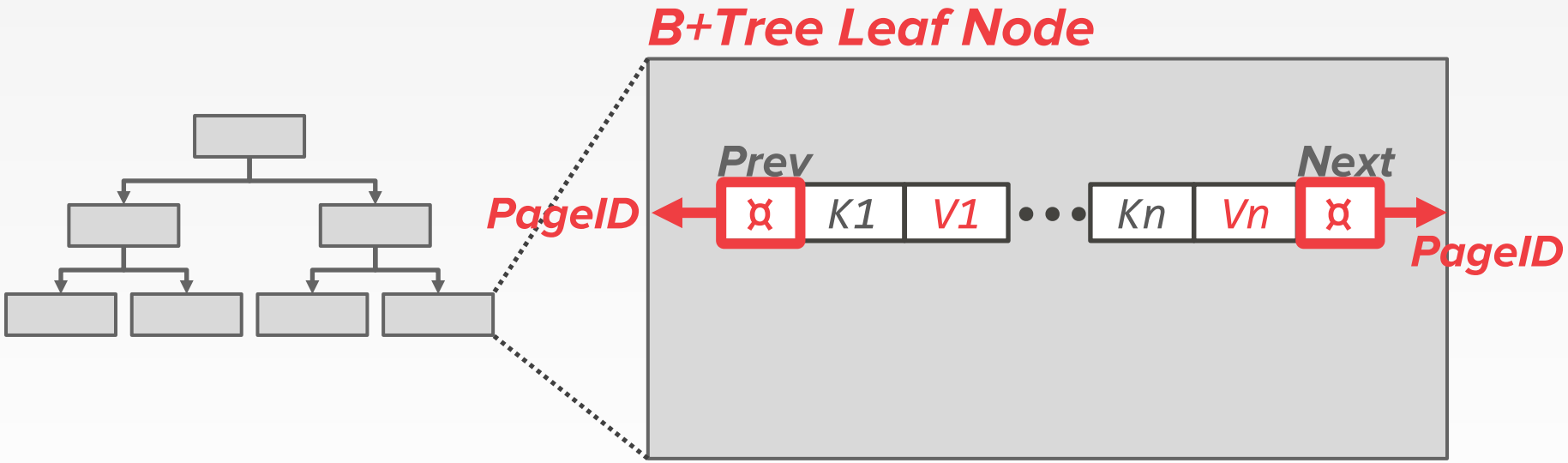
B+TREE LEAF NODES



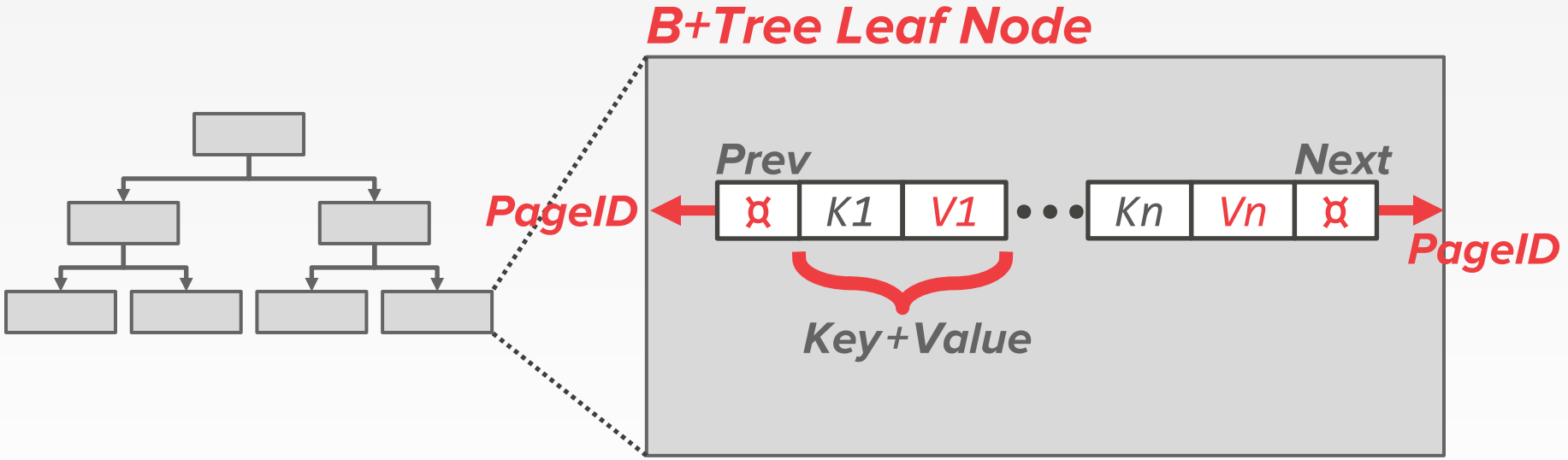
B+TREE LEAF NODES



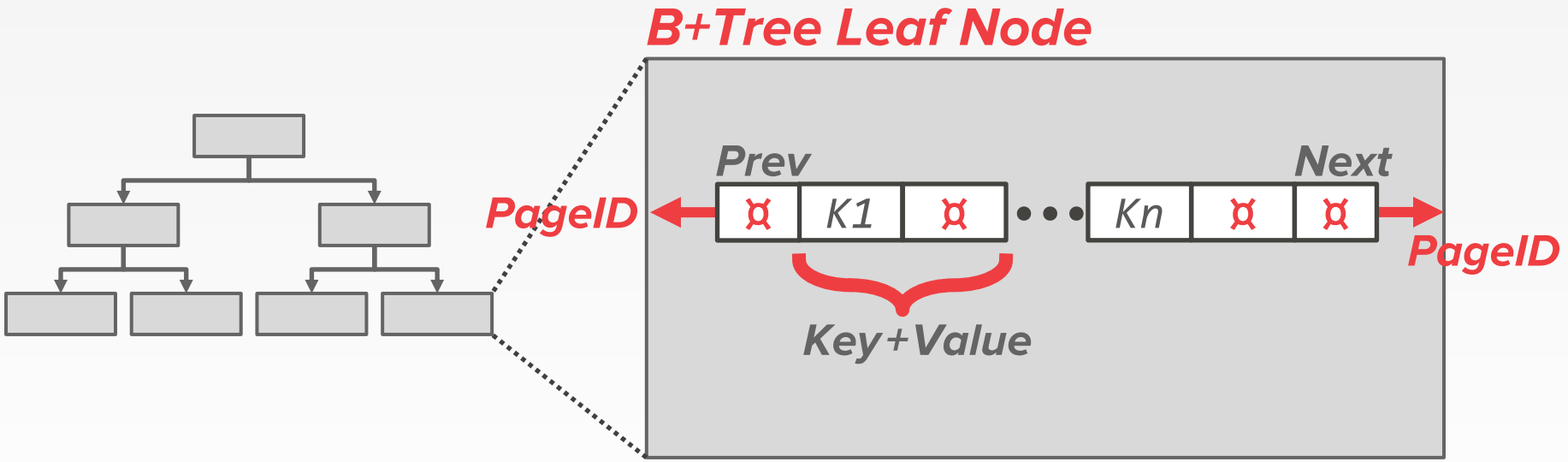
B+TREE LEAF NODES



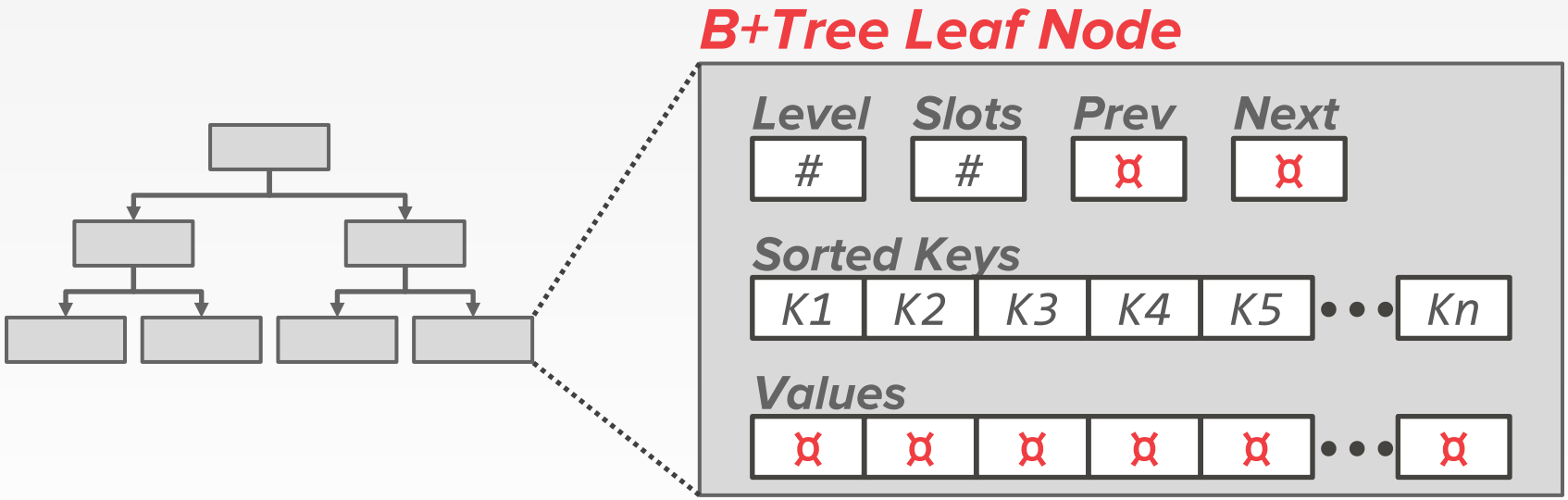
B+TREE LEAF NODES



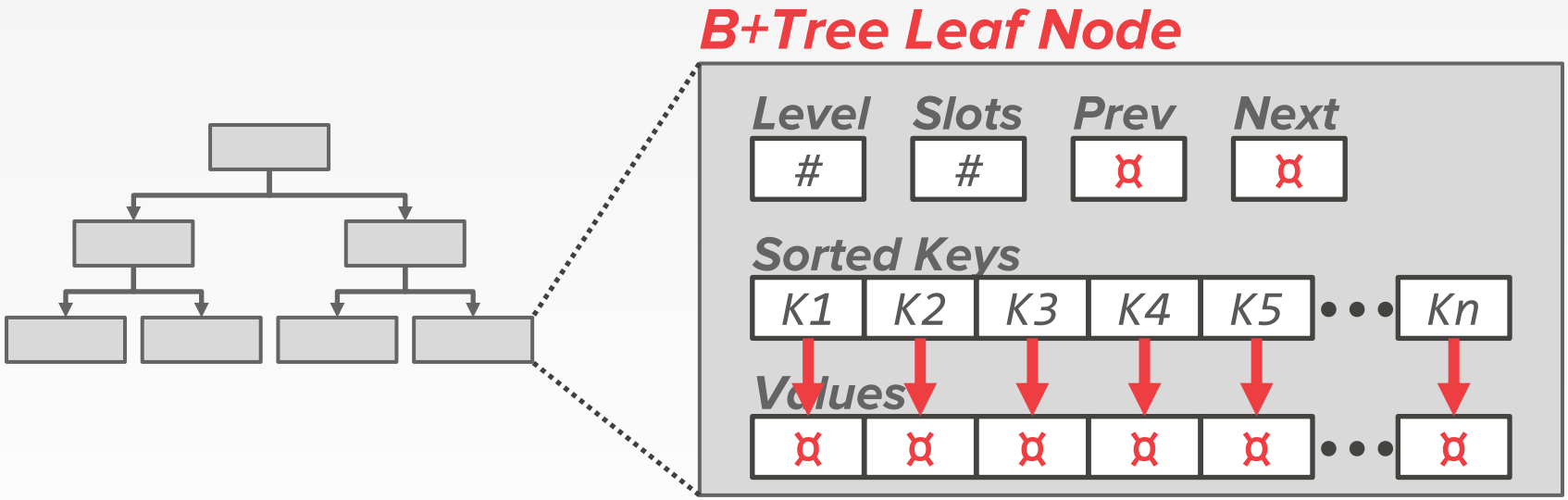
B+TREE LEAF NODES



B+TREE LEAF NODES



B+TREE LEAF NODES



B+TREE INSERT

Find correct leaf **L**.
Put data entry into **L** in sorted order.

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

Else, must split **L** 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 VISUALIZATION

https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html

Source: David Gales (Univ. of San Francisco)

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

B+TREES IN PRACTICE

Typical Fill-Factor: 67%.

→ Average Fanout = 2*100*0.67 = 134

Typical Capacities:

→ Height 4: 1334 = 312,900,721 entries

→ Height 3: 1333 = 2,406,104 entries

Pages per level:

→ Level 1 = 1 page = 8 KB

→ Level 2 = 134 pages = 1 MB

→ Level 3 = 17,956 pages = 140 MB

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 doesn't include a pkey, the DBMS will automatically make a hidden row id pkey.

Other DBMSs cannot use them at all.

SELECTION CONDITIONS

The DBMS can use a B+Tree index if the query provides any of the attributes of the search key.

Example: Index on **<a,b,c>**

→ Supported: **(a=5 AND b=3)**

→ Supported: **(b=3)**.

Not all DBMSs support this.

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

B+TREE DESIGN CHOICES

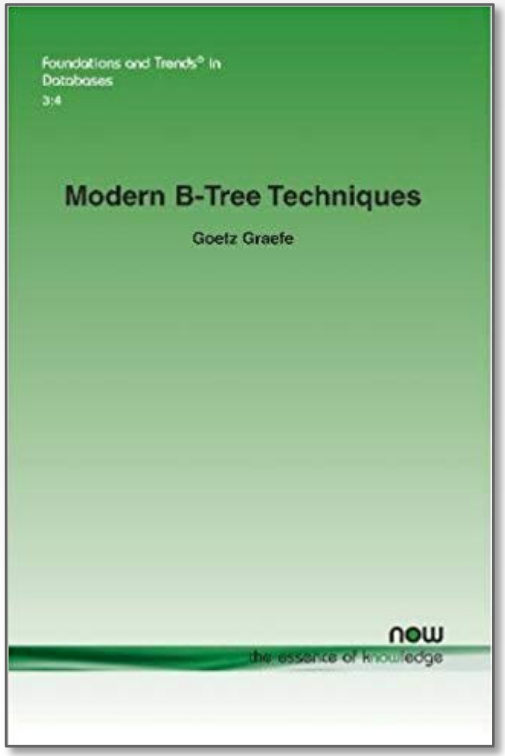
Node Size

Merge Threshold

Intra-Node Search

Variable Length Keys

Non-Unique Indexes



NODE SIZE

The slower the disk, 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 don't always merge nodes when it is half full.

Delaying a merge operation may reduce the amount of reorganization.

May be better to just let underflows to exist and then periodically rebuild entire tree.

INTRA-NODE SEARCH

Approach #1: Linear

→ Scan node keys from beginning to end.

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.

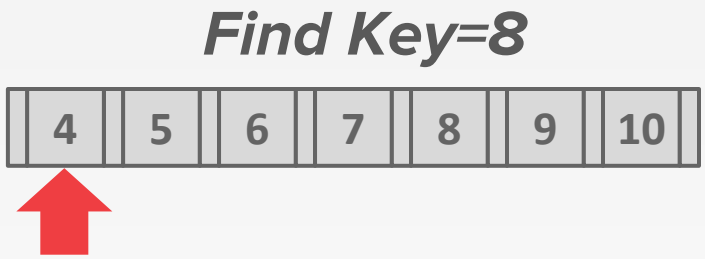
Approach #2: Binary

→ 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.

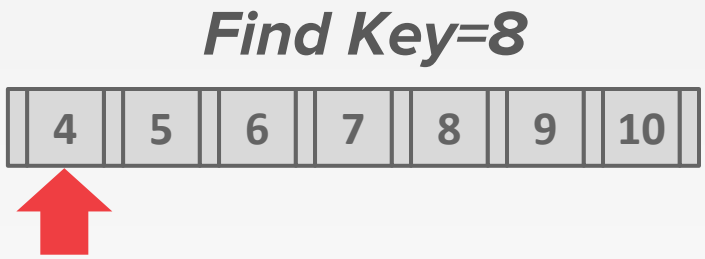
Approach #2: Binary

→ 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.

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.

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

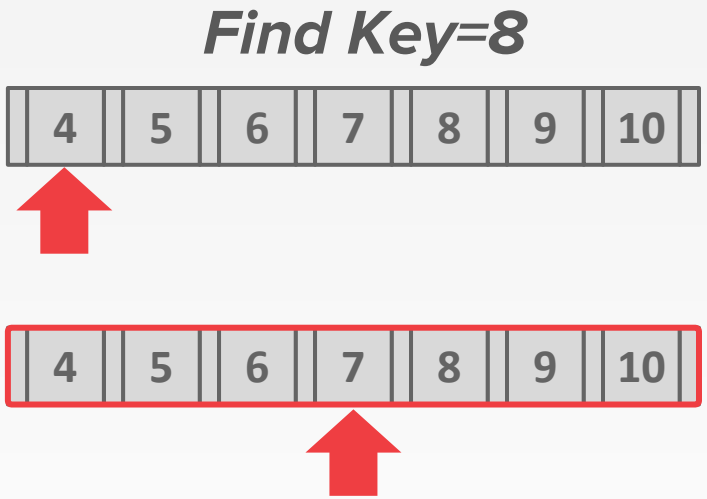
→ Scan node keys from beginning to end.

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

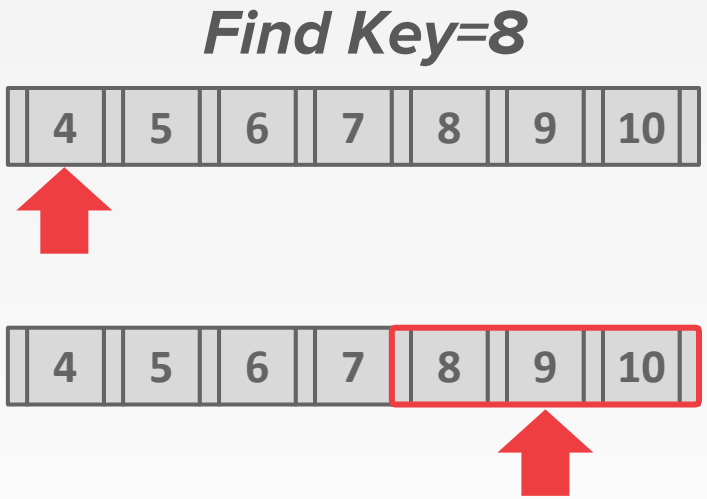
→ Scan node keys from beginning to end.

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

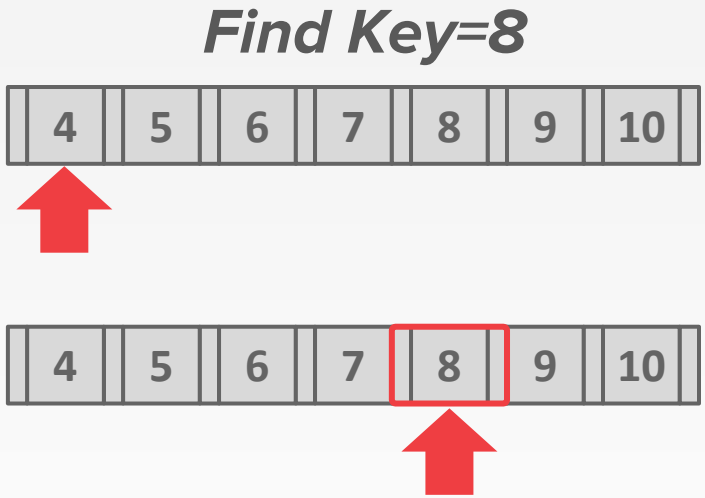
→ Scan node keys from beginning to end.

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

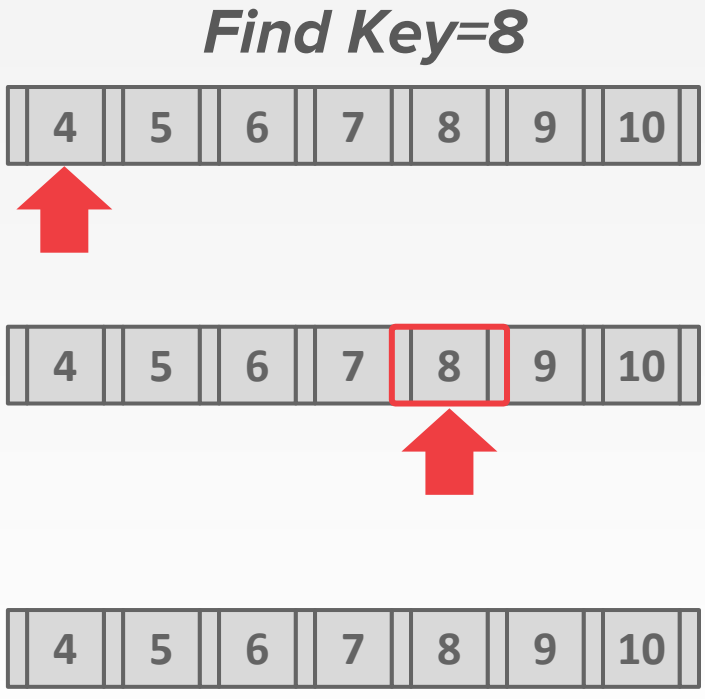
→ Scan node keys from beginning to end.

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

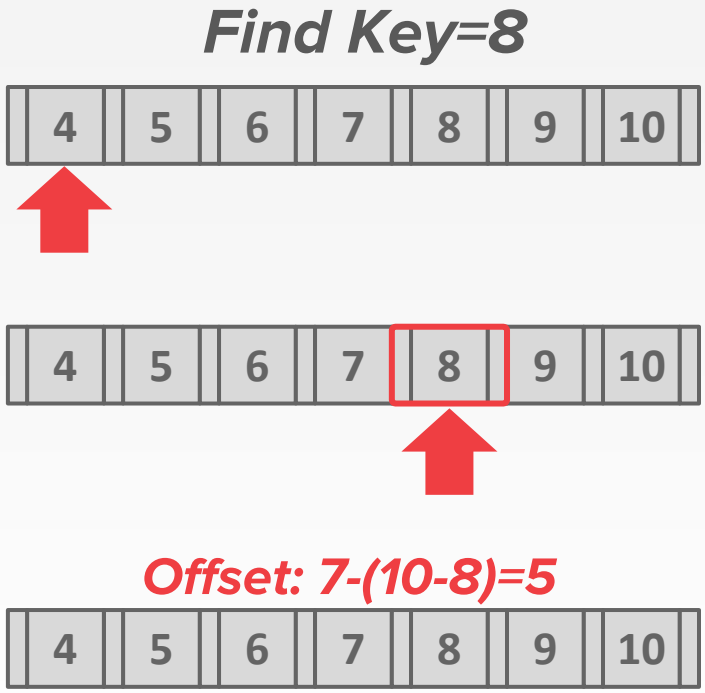
→ Scan node keys from beginning to end.

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

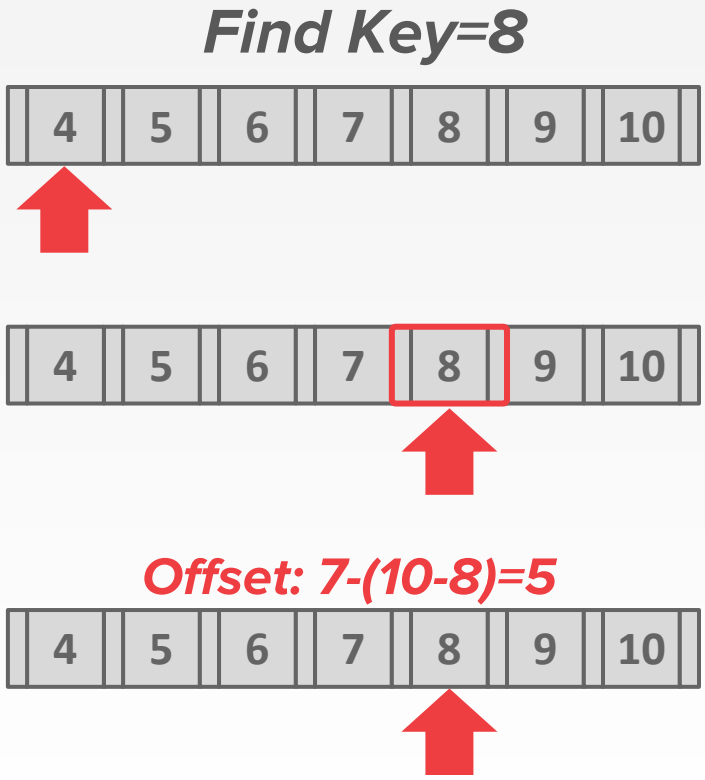
→ Scan node keys from beginning to end.

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.



INDEX IMPLEMENTATION ISSUES

Bulk Insert

Pointer Swizzling

Prefix Compression

Memory Pools

Garbage Collection

Non-Unique Keys

Variable-length Keys

Prefix Compression

BULK INSERT

The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.

BULK INSERT

The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.

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

BULK INSERT

The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.

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

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

BULK INSERT

The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.

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

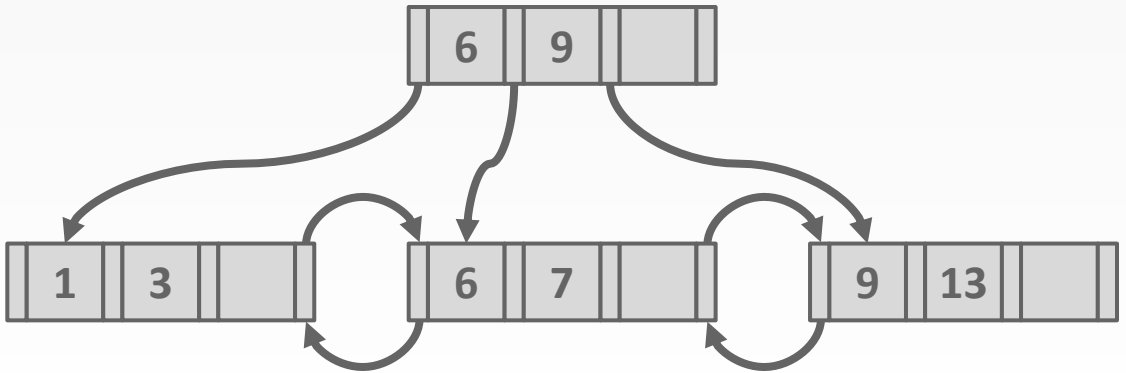
Sorted Keys: 1, 3, 6, 7, 9, 13

BULK INSERT

The fastest/best way to build a B+Tree 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



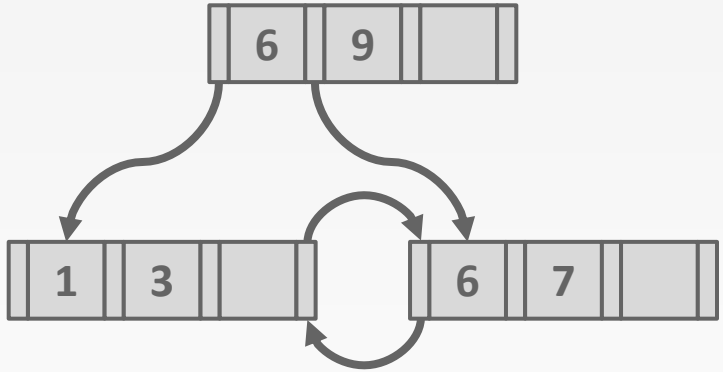
POINTER SWIZZLING

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

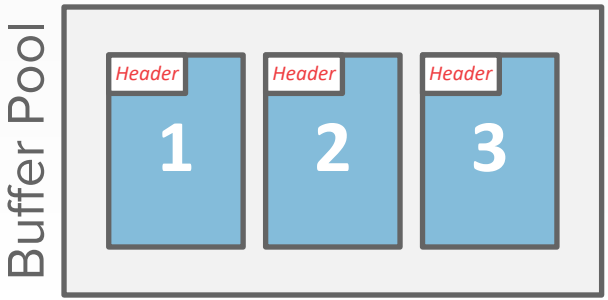
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.

POINTER SWIZZLING

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

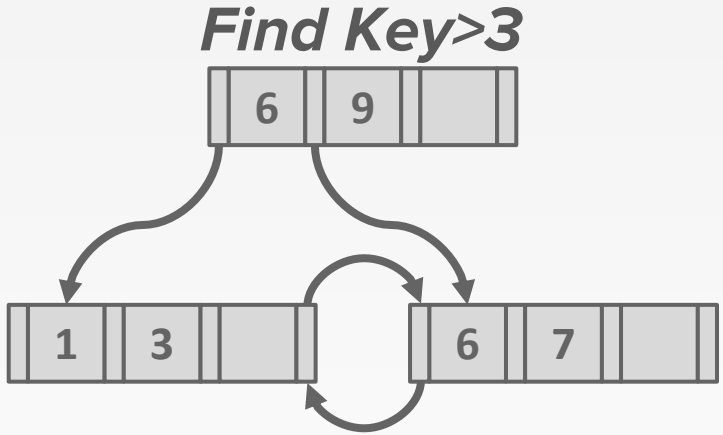


If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.

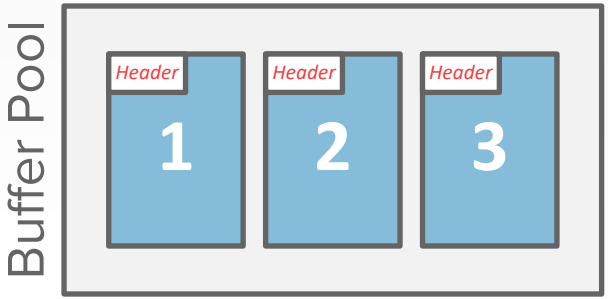


POINTER SWIZZLING

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

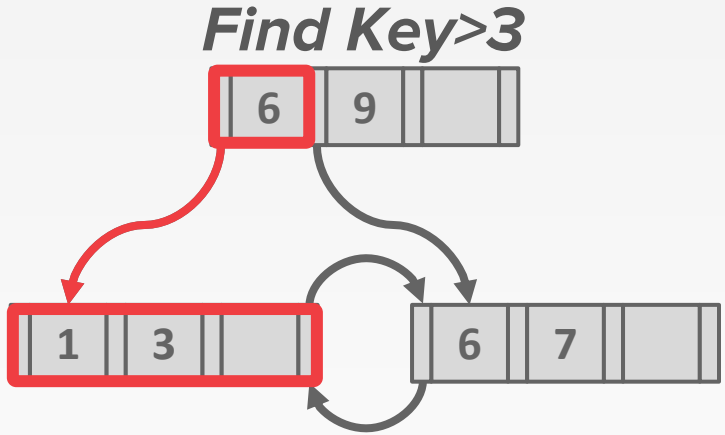


If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.

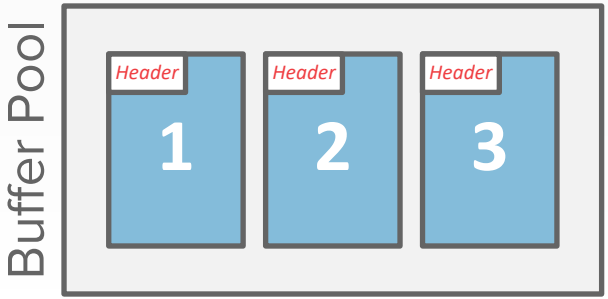


POINTER SWIZZLING

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

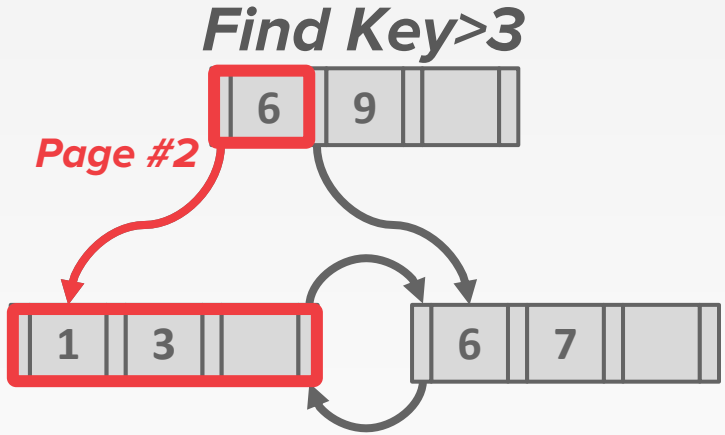


If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.

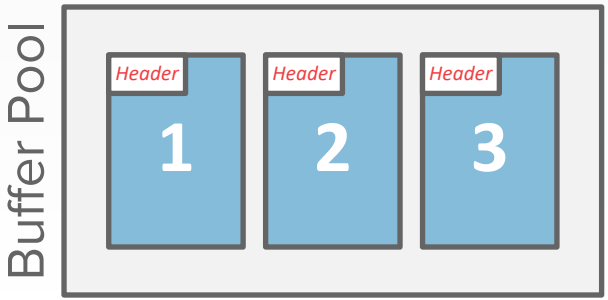


POINTER SWIZZLING

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



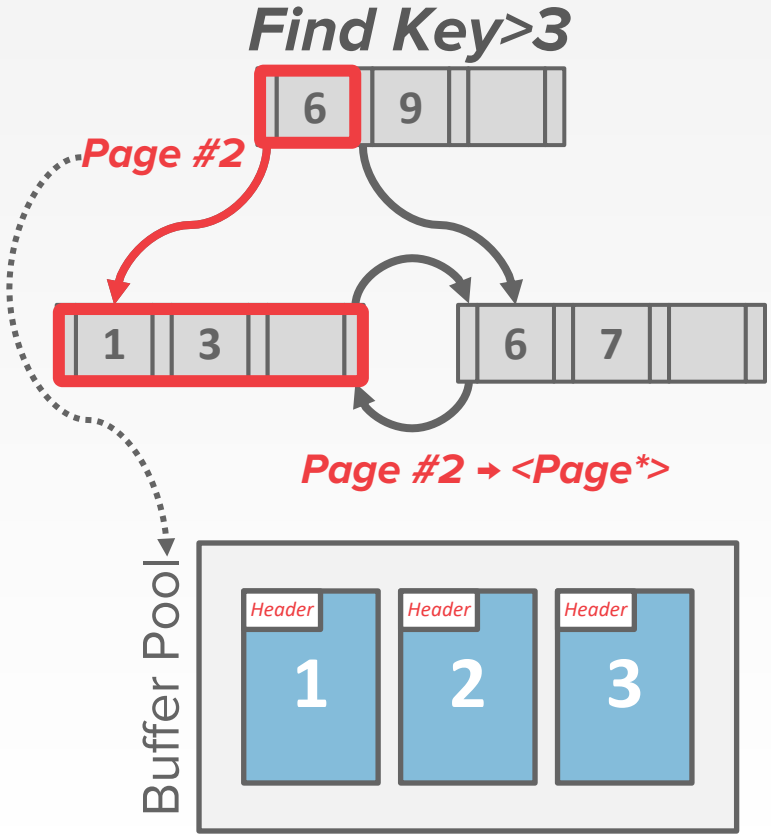
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.



POINTER SWIZZLING

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

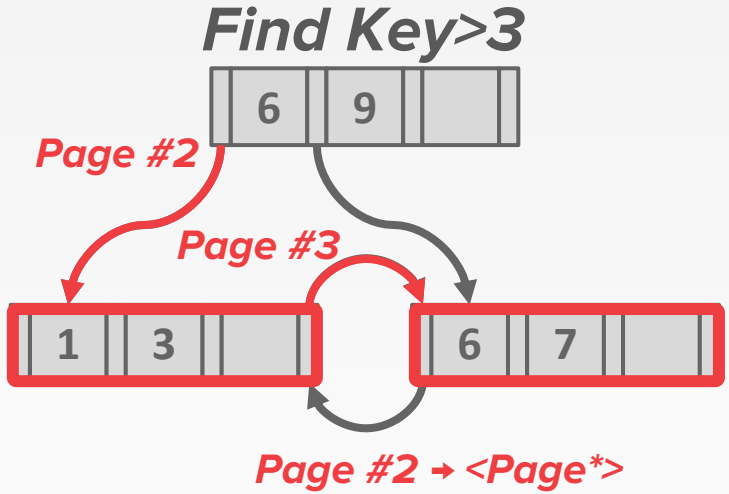
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.

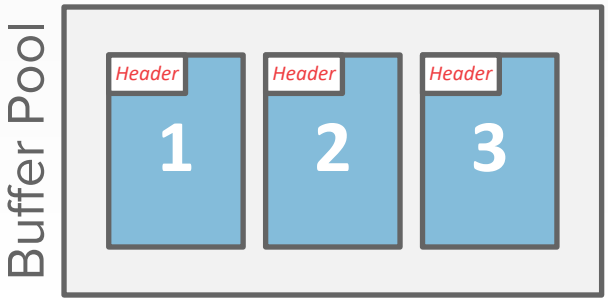


POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS has to 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, thereby removing the need to get address from the page table.

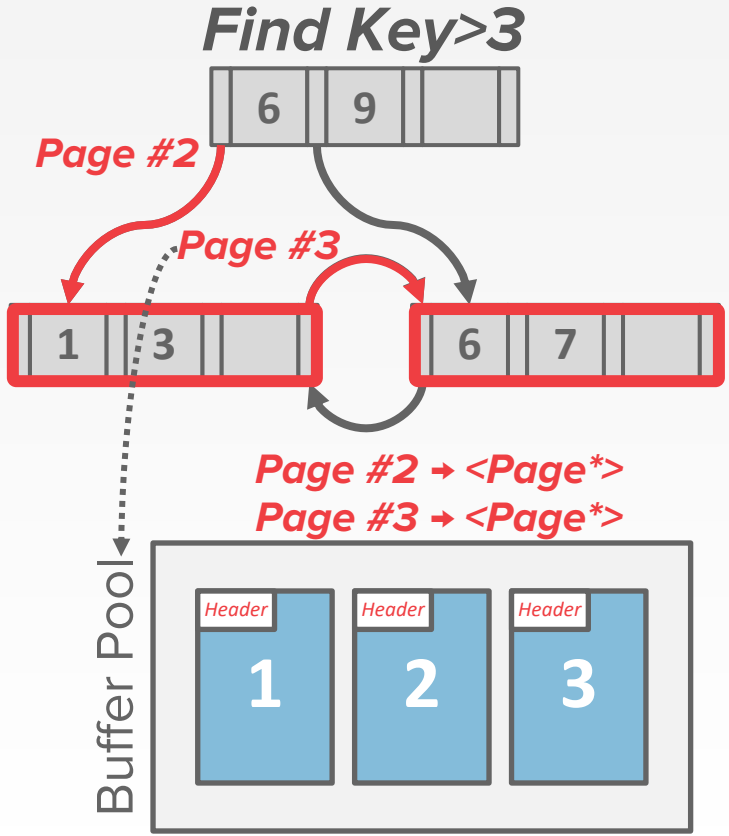




POINTER SWIZZLING

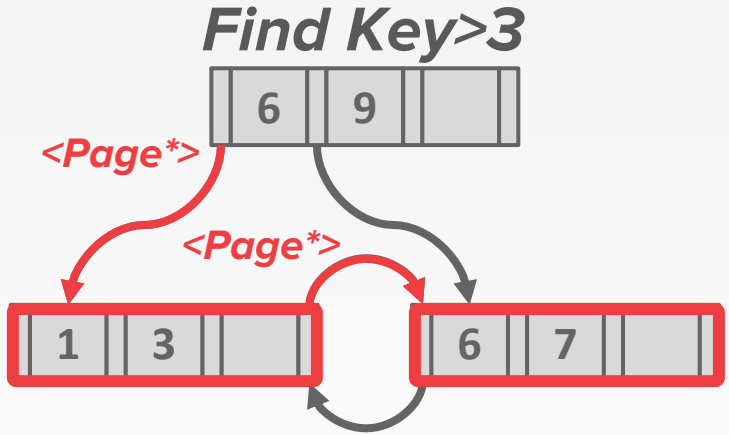
Nodes use page ids to reference other nodes in the index. The DBMS has to get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids, thereby removing the need to get address from the page table.

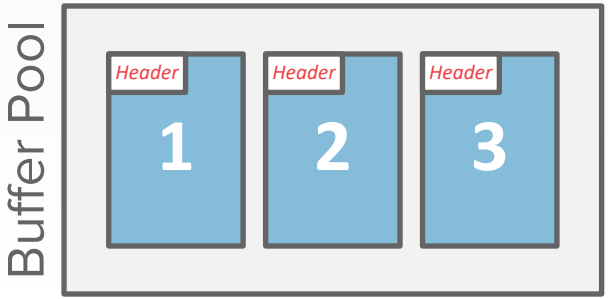


POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS has to 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, thereby removing the need to get address from the page table.



MEMORY POOLS

We don't want to be calling `malloc` and `free` anytime we need to add or delete a node.

This could lead to a system call.

→ If you call `malloc` to request 10 bytes of memory, the allocator may invoke the `sbrk` (or `mmap`) system call to request 4K bytes from OS.

→ Then, when you call `malloc` next time to request another 10 bytes, it may not have to issue a system call; instead, it may return a pointer within allocated memory.

MEMORY POOLS

If all the nodes are the same size, then the index can maintain a pool of available nodes.

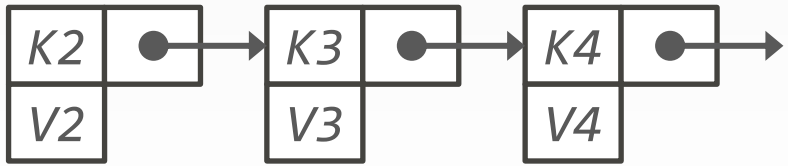
→ **Insert**: Grab a free node, otherwise create a new one.
→ **Delete:** Add the node back to the free pool.

Need some policy to decide when to retract the pool size (garbage collection & de-fragmentation).

GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

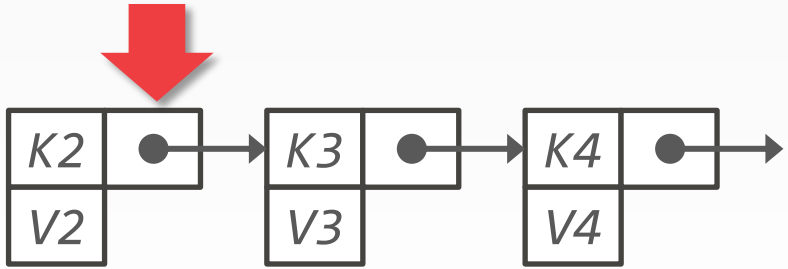
→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…



GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

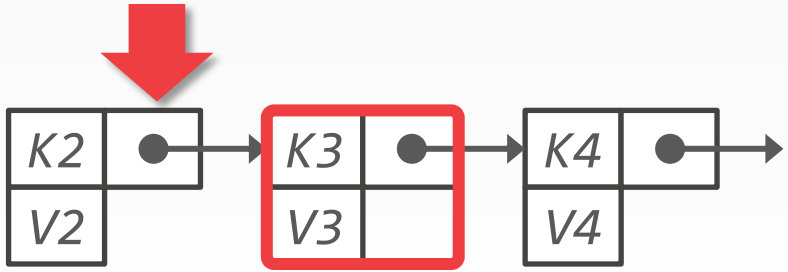
→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…



GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

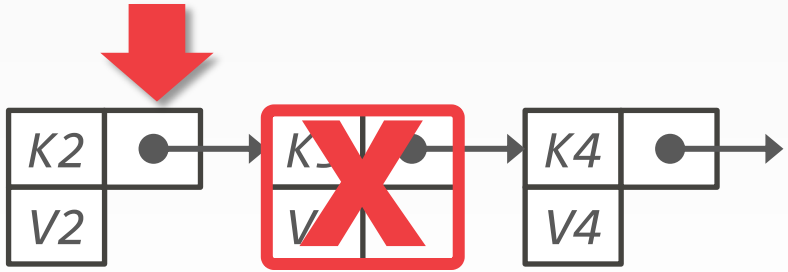
→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…



GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…



GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…

GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…

REFERENCE COUNTING

Maintain a counter for each node to keep track of the number of threads that are accessing it.

→ Increment the counter before accessing.
→ Decrement it when finished.
→ A node is only safe to delete when the count is zero.

This has bad performance for multi-core CPUs

→ Incrementing/decrementing counters causes a lot of cache coherence traffic.

OBSERVATION

We don't actually care about the actual value of the reference counter. We only need to know when it reaches zero.

We don't have to perform garbage collection immediately when the counter reaches zero.

Source: Stephen Tu

EPOCH GARBAGE COLLECTION

Maintain a global epoch counter that is periodically updated (e.g., every 10 ms).

→ Keep track of what threads enter the index during an epoch and when they leave.

Mark the current epoch of a node when it is marked for deletion.

→ The node can be reclaimed once all threads have left that epoch (and all preceding epochs).

Also known as ***Read-Copy-Update*** (RCU) in Linux.

NON-UNIQUE INDEXES

Approach #1: Duplicate Keys

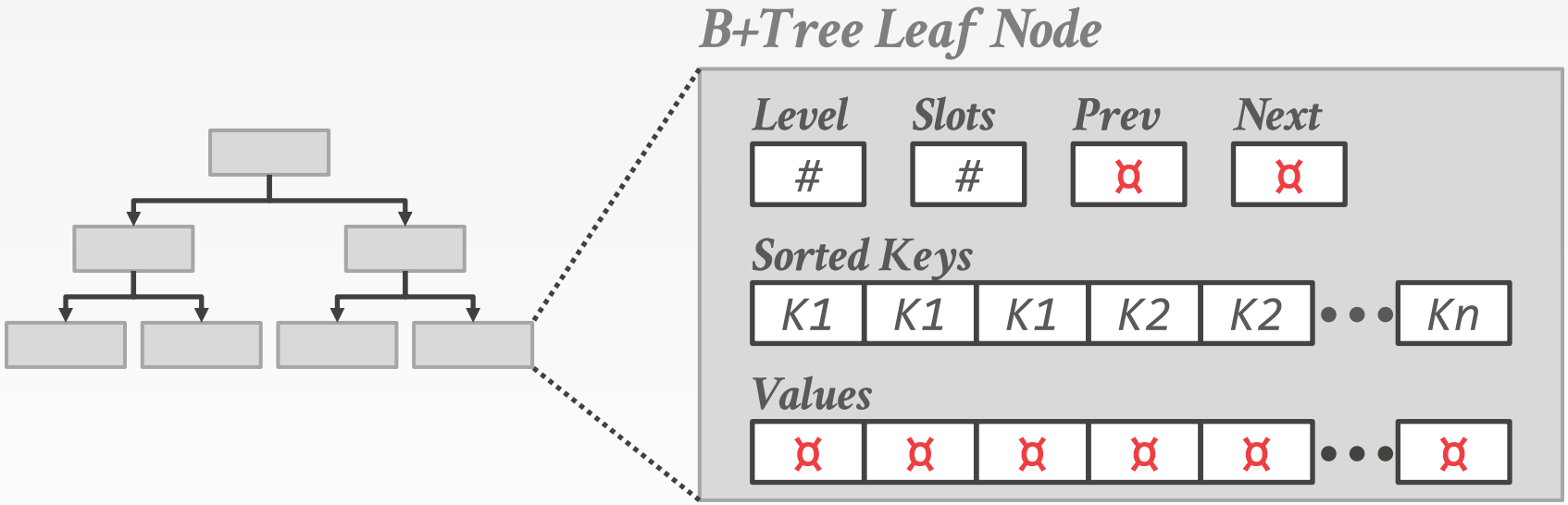
→ Use the same node layout but store duplicate keys multiple times.

Approach #2: Value Lists

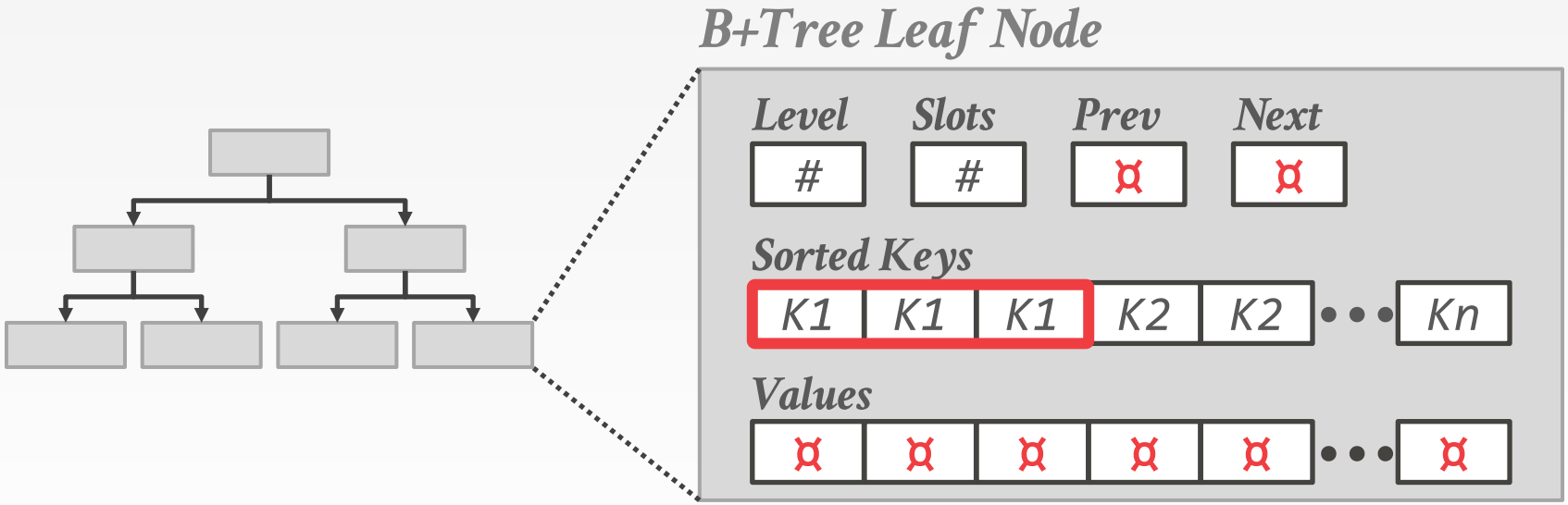
→ Store each key only once and maintain a linked list of unique values.

MODERN B-TREE TECHNIQUES
NOW PUBLISHERS 2010

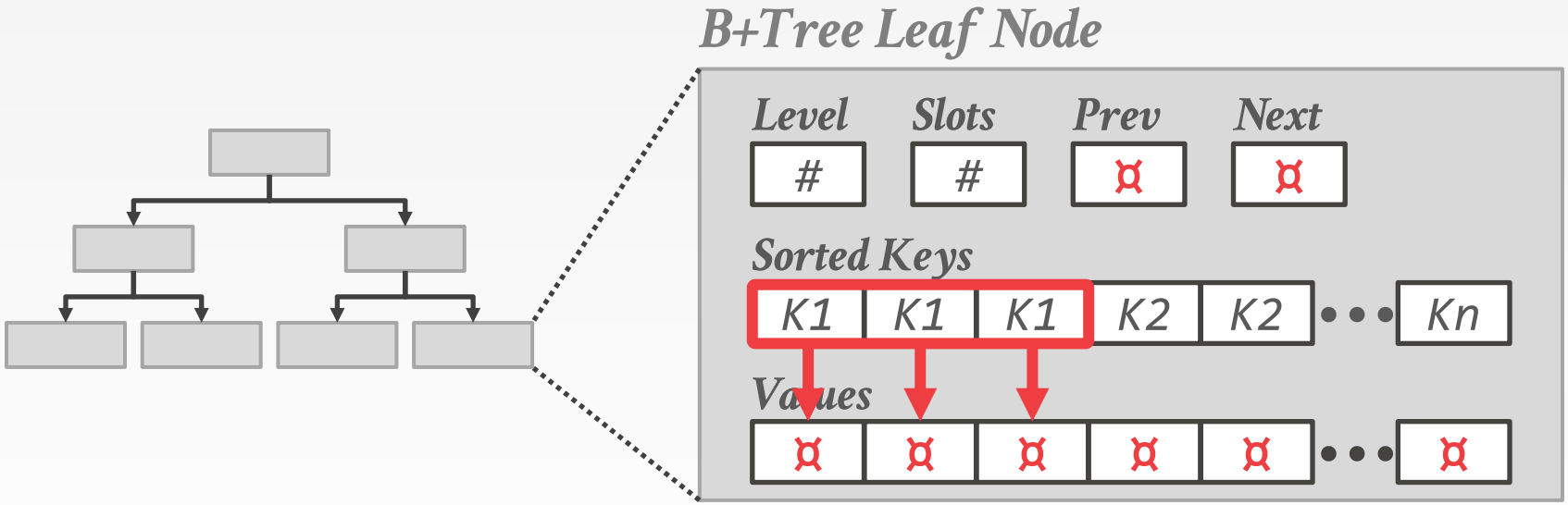
NON-UNIQUE: DUPLICATE KEYS



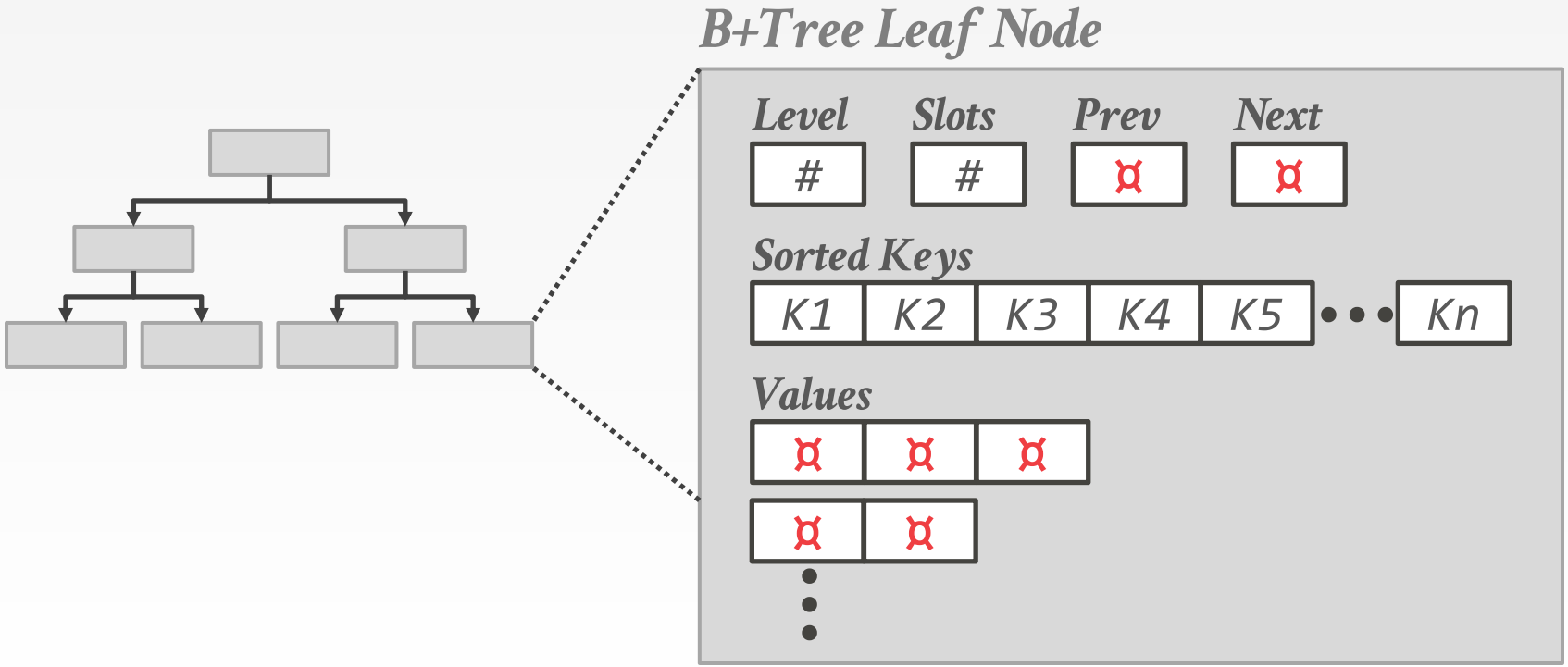
NON-UNIQUE: DUPLICATE KEYS



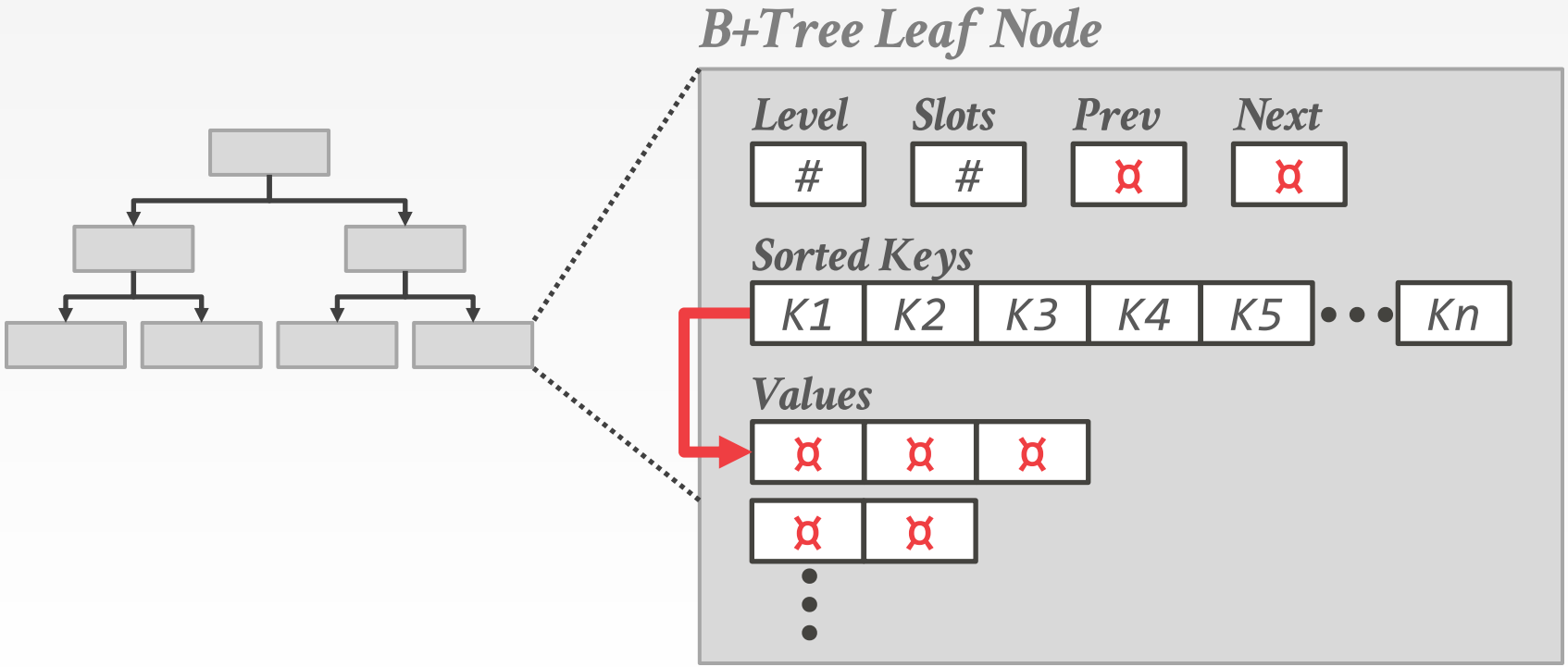
NON-UNIQUE: DUPLICATE KEYS



NON-UNIQUE: VALUE LISTS



NON-UNIQUE: VALUE LISTS



VARIABLE LENGTH KEYS

Approach #1: Pointers

→ Store the keys as pointers to the tuple's attribute.

Approach #2: Variable Length Nodes

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

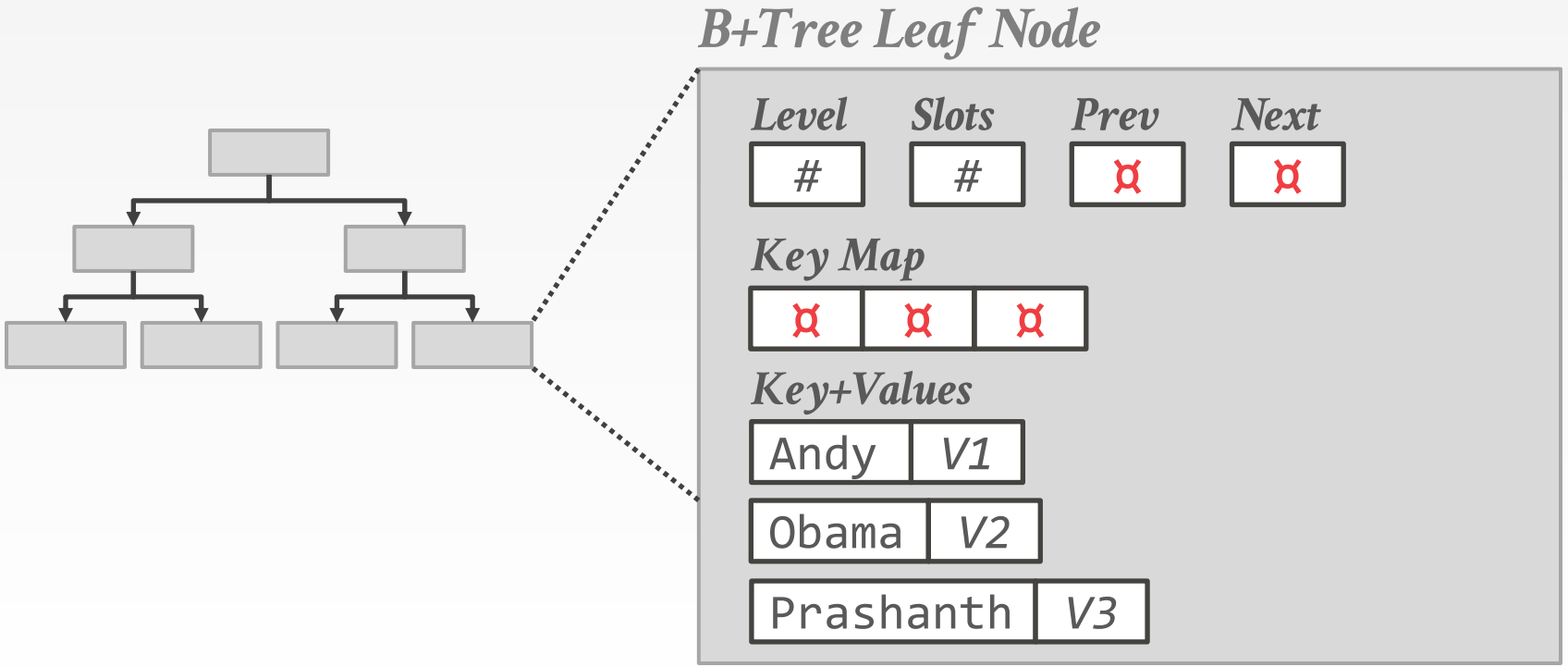
Approach #3: Padding

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

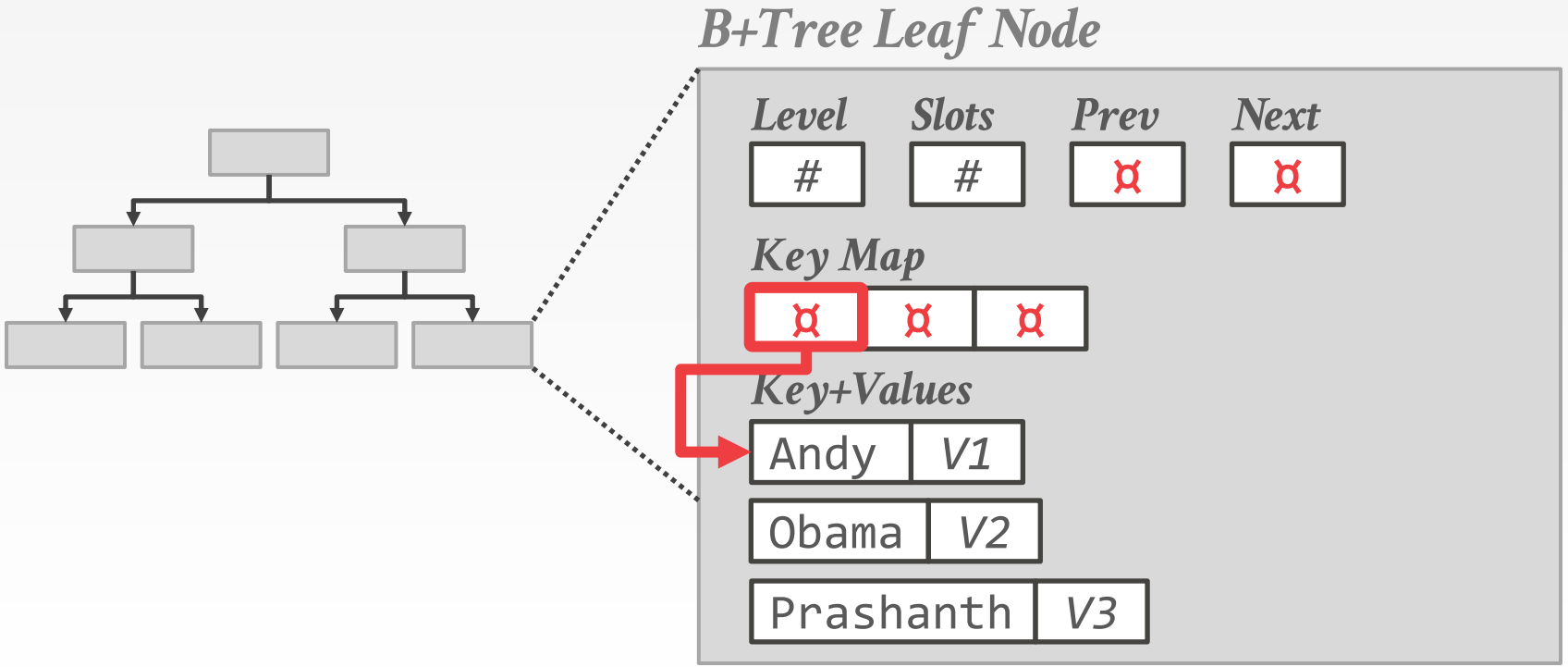
Approach #4: Key Map / Indirection

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

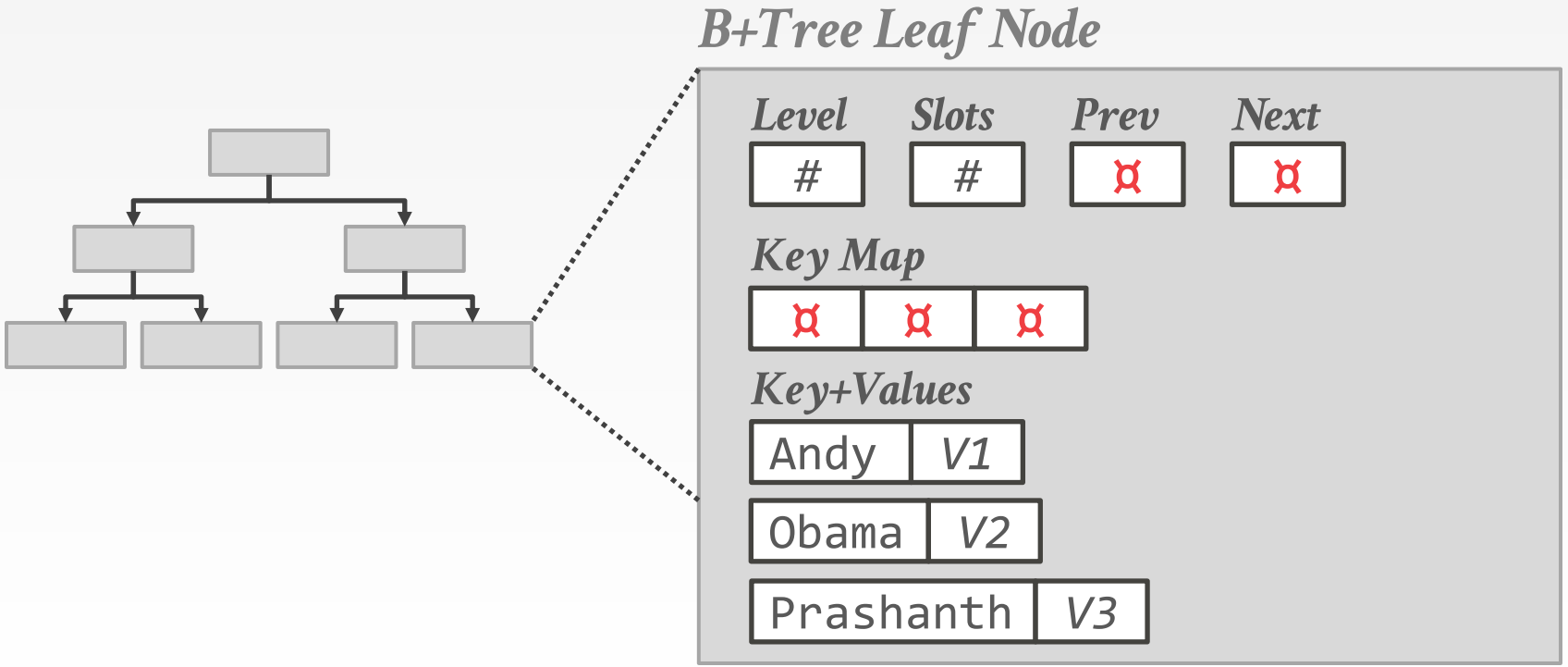
KEY MAP / INDIRECTION



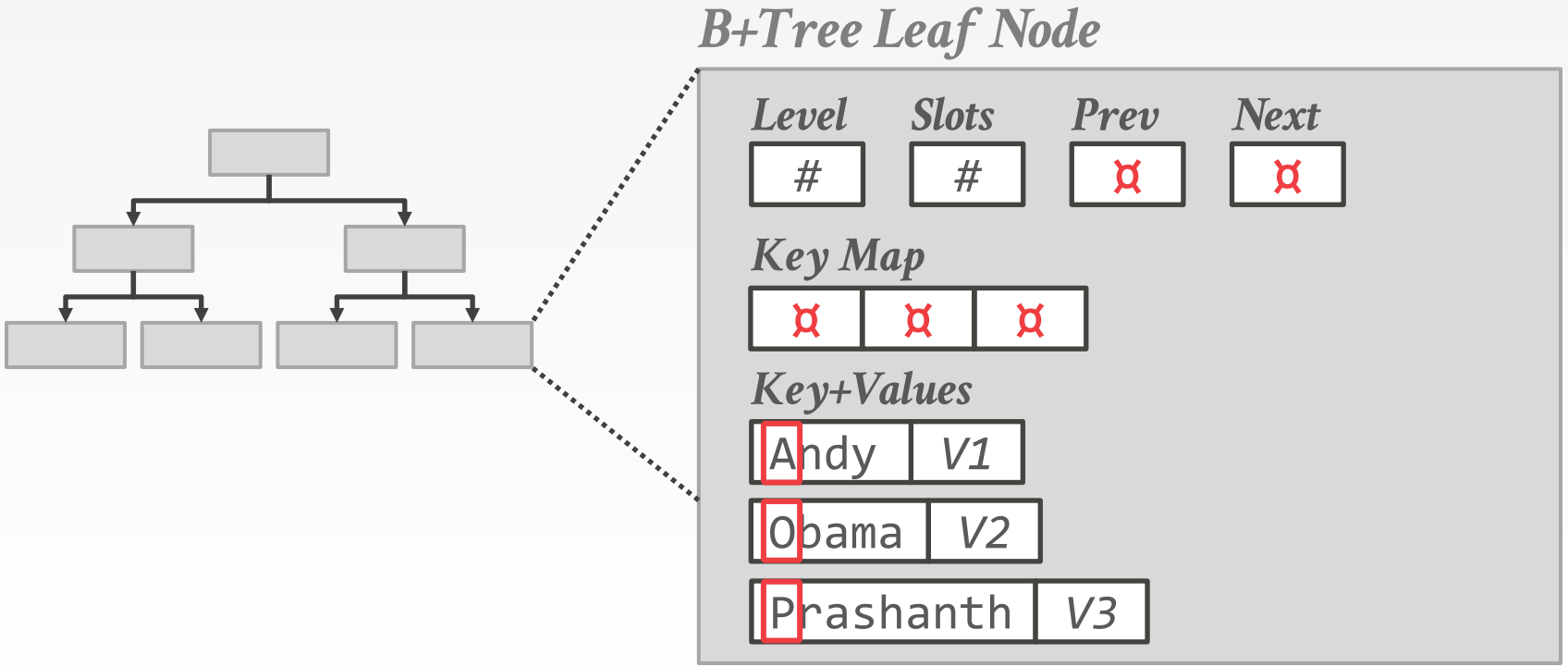
KEY MAP / INDIRECTION



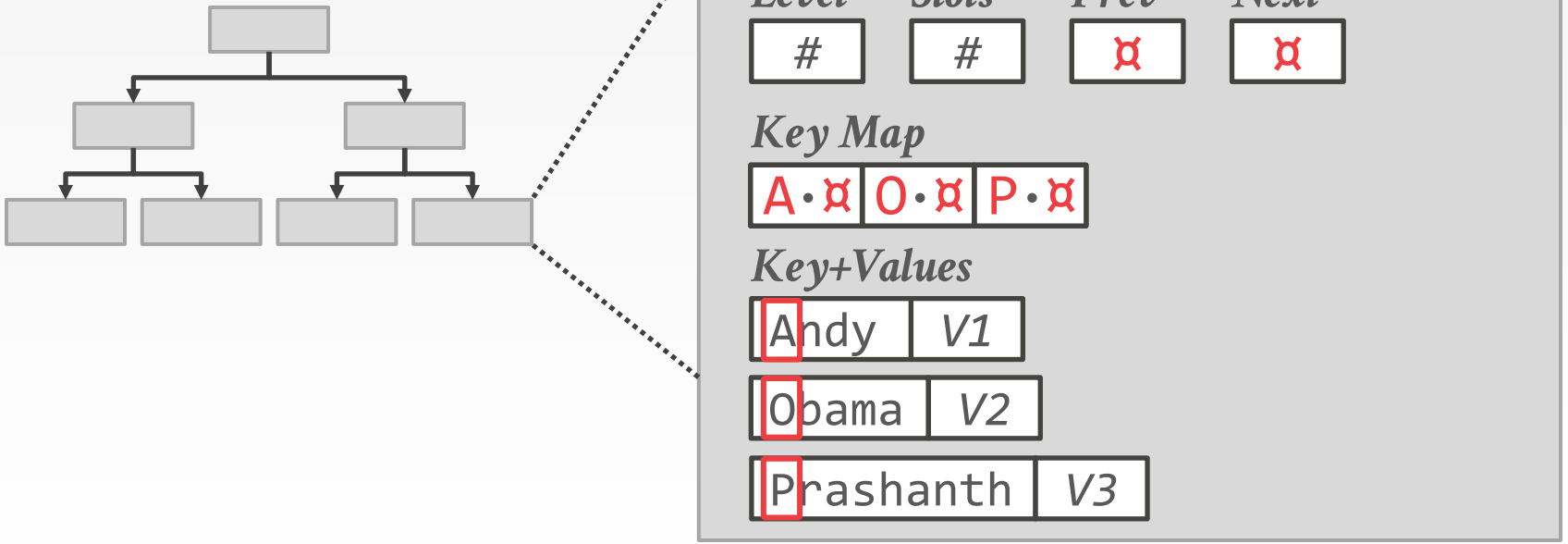
KEY MAP / INDIRECTION



KEY MAP / INDIRECTION



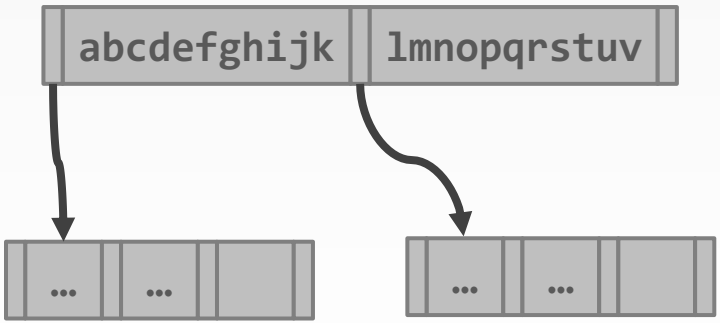
KEY MAP / INDIRECTION



PREFIX COMPRESSION

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

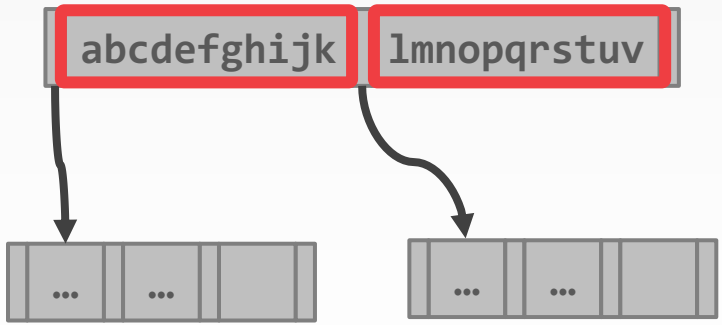
Since keys are sorted in lexicographical order, there will be a lot of duplicated prefixes.



PREFIX COMPRESSION

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

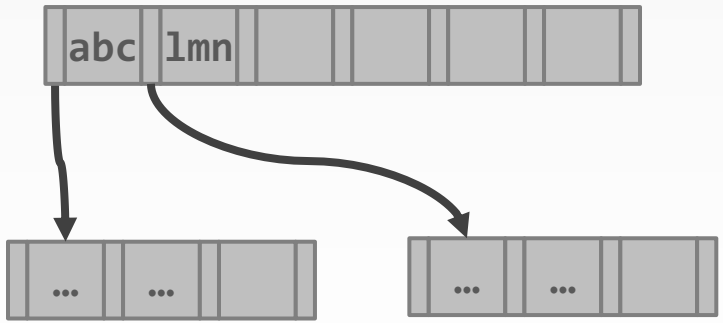
Since keys are sorted in lexicographical order, there will be a lot of duplicated prefixes.



PREFIX COMPRESSION

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

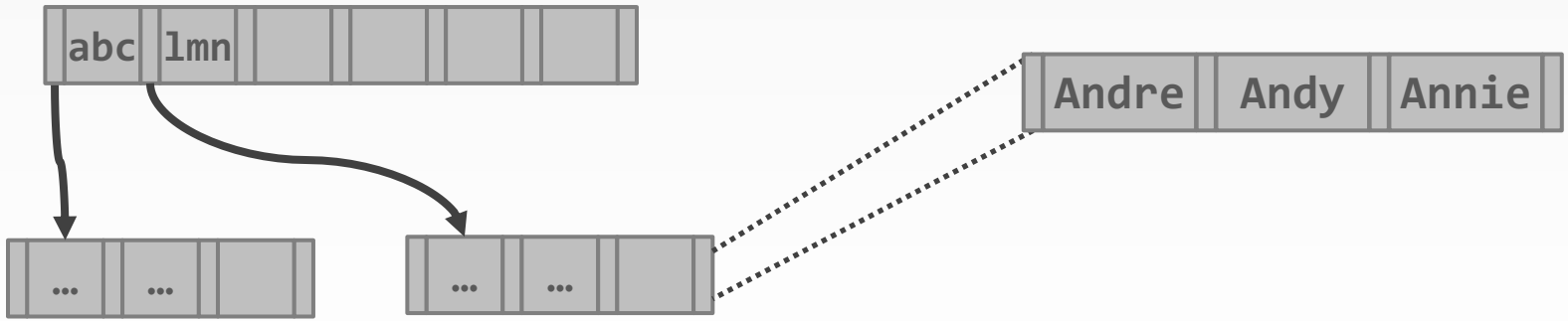
Since keys are sorted in lexicographical order, there will be a lot of duplicated prefixes.



PREFIX COMPRESSION

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

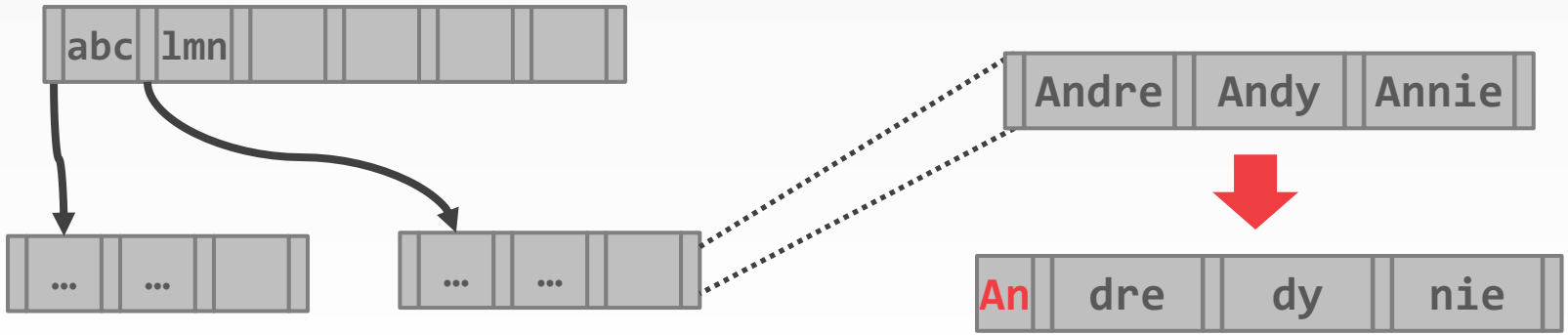
Since keys are sorted in lexicographical order, there will be a lot of duplicated prefixes.



PREFIX COMPRESSION

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

Since keys are sorted in lexicographical order, there will be a lot of duplicated prefixes.



ADAPATIVE RADIX TREE (ART)

Uses digital representation of keys to examine prefixes 1-by-1 instead of comparing entire key.

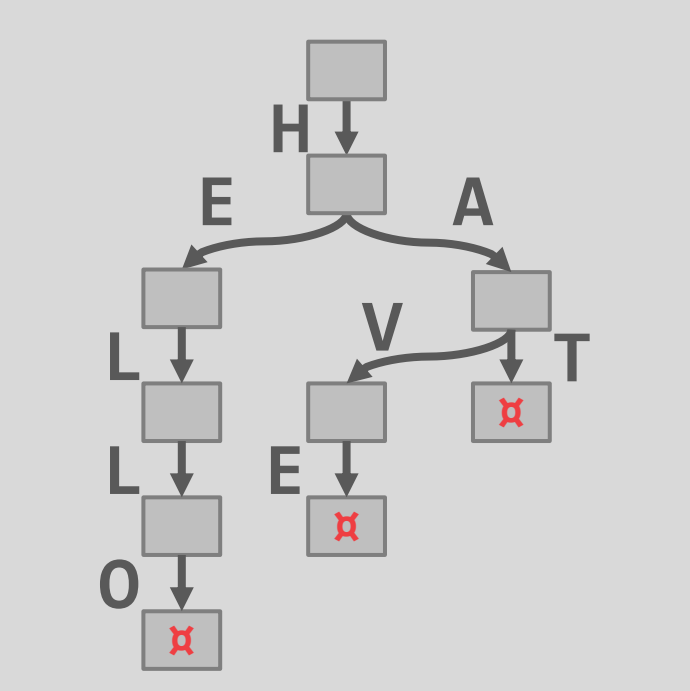
Radix trees properties:

→ The height of the tree depends on the length of keys.
(unlike B+tree where height depends on the number of keys)
→ Does not require rebalancing
→ The path to a leaf node represents the key of the leaf
→ Keys are stored implicitly and can be reconstructed from paths.
→ Structure does not depend on order of key insertion

THE ADAPTIVE RADIX TREE: ARTFUL INDEXING FOR MAIN-MEMORY DATABASES
ICDE 2013

TRIE VS. RADIX TREE

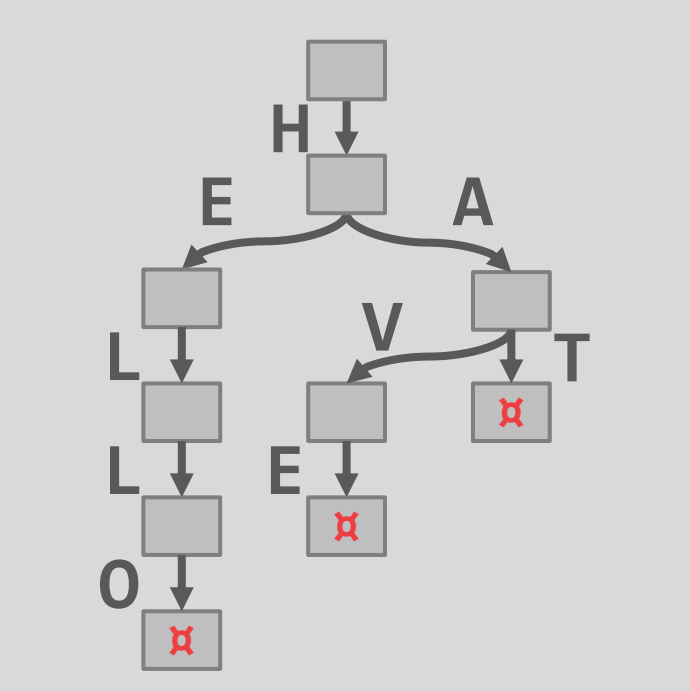
Trie (Re`trie'val - 1959)



Keys: HELLO, HAT, HAVE

TRIE VS. RADIX TREE

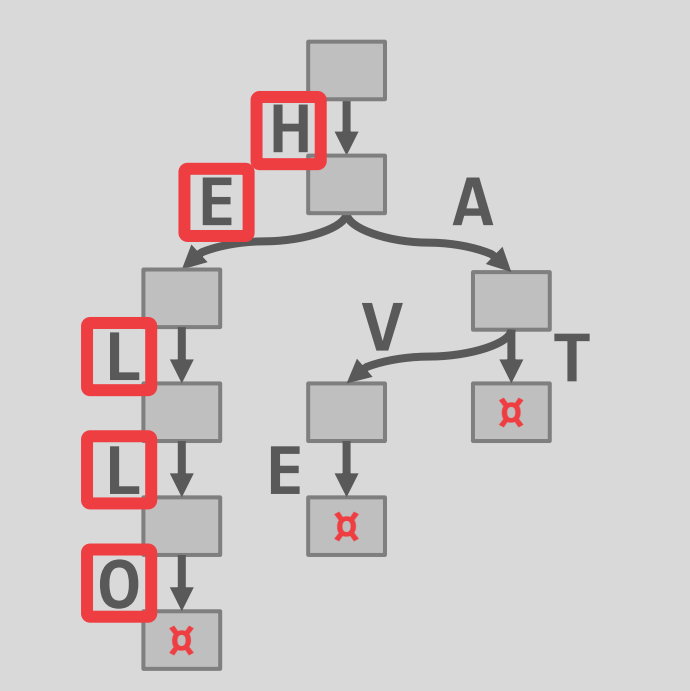
Trie (Re`trie'val - 1959)



Keys: HELLO, HAT, HAVE

TRIE VS. RADIX TREE

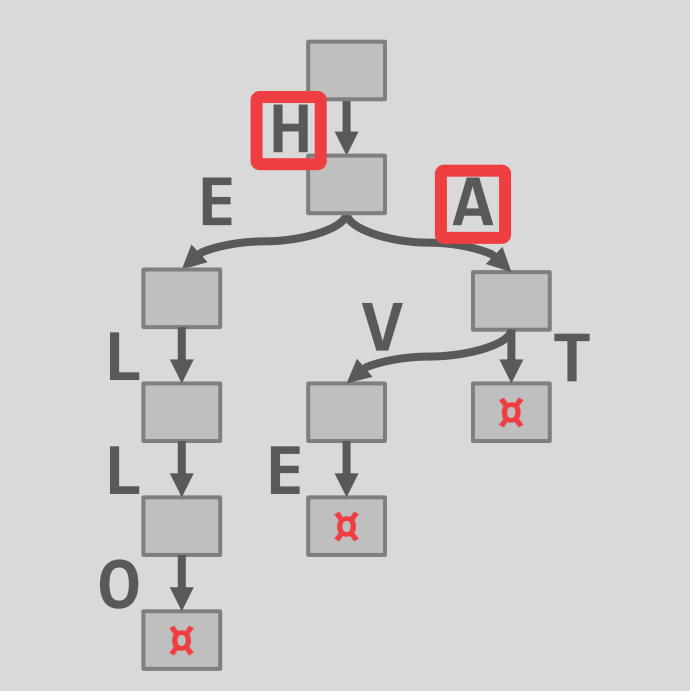
Trie (Re`trie'val - 1959)



Keys: HELLO, HAT, HAVE

TRIE VS. RADIX TREE

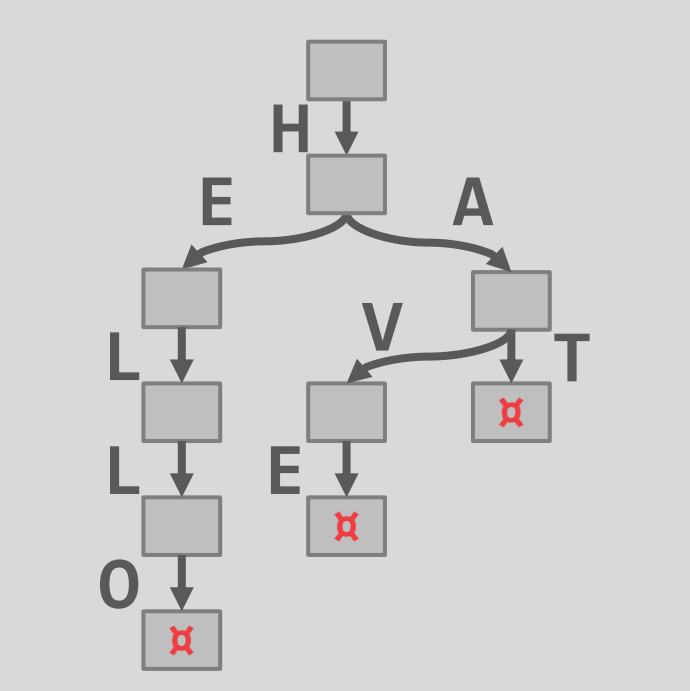
Trie (Re`trie'val - 1959)



Keys: HELLO, HAT, HAVE

TRIE VS. RADIX TREE

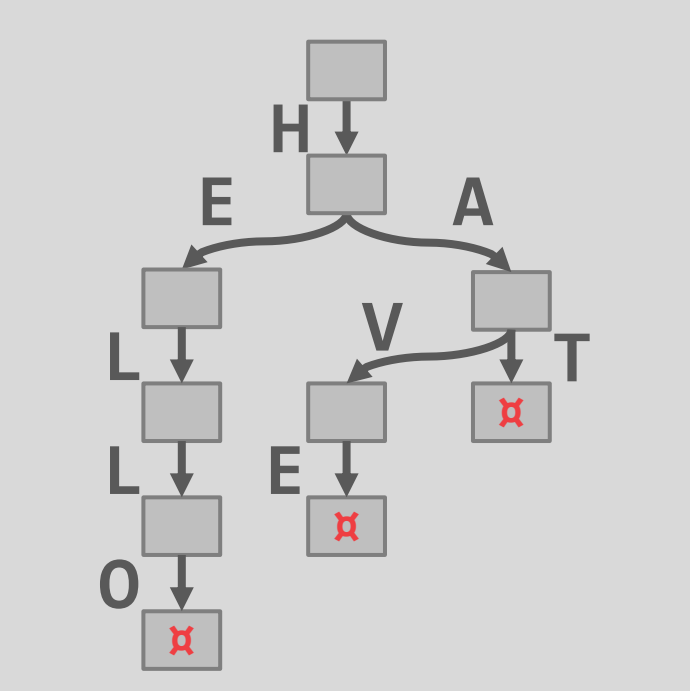
Trie (Re`trie'val - 1959)



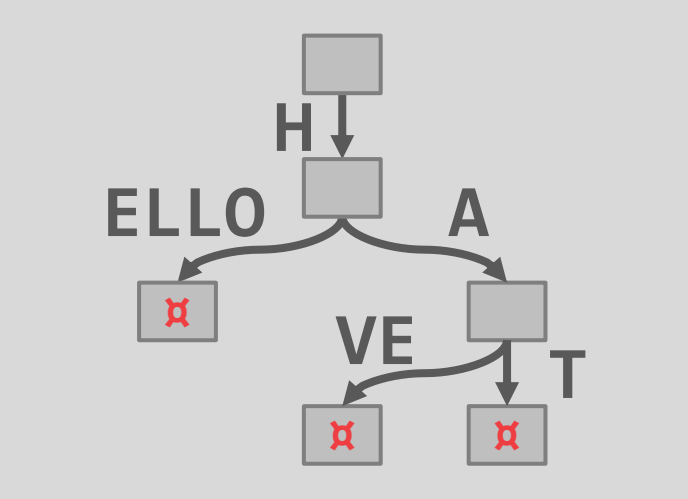
Keys: HELLO, HAT, HAVE

TRIE VS. RADIX TREE

Trie (Re`trie'val - 1959)



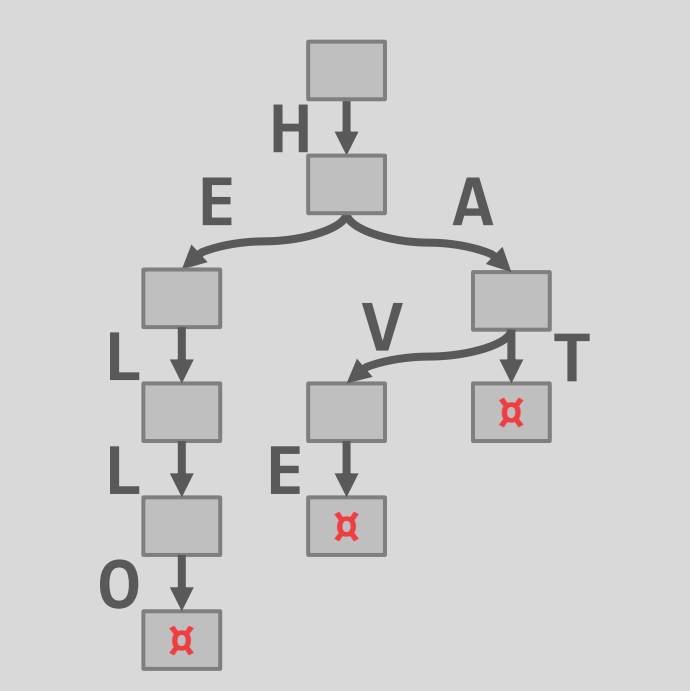
Radix Tree



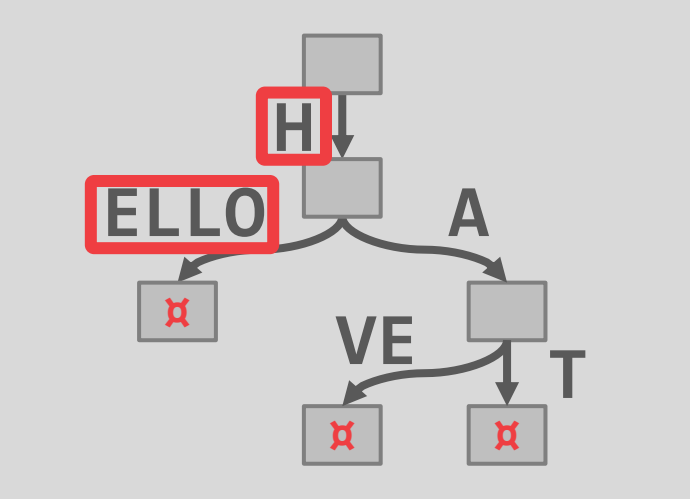
Keys: HELLO, HAT, HAVE

TRIE VS. RADIX TREE

Trie (Re`trie'val - 1959)



Radix Tree



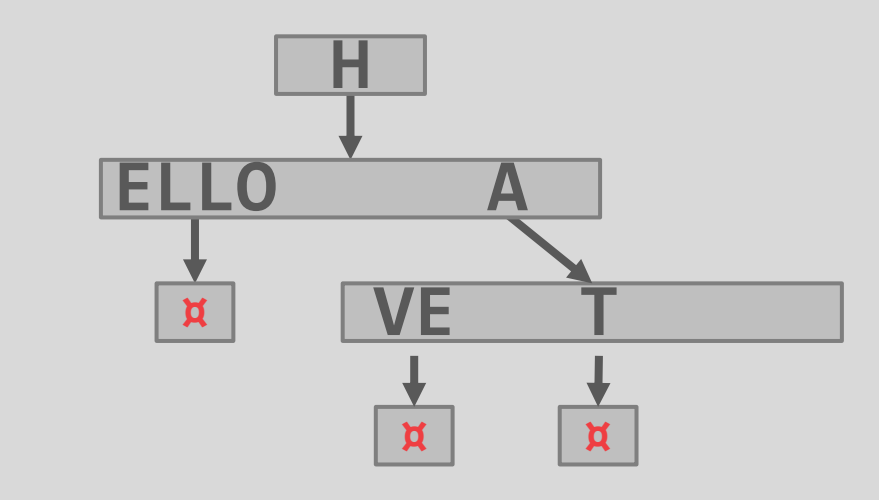
Keys: HELLO, HAT, HAVE

ART: ADAPTIVELY SIZED NODES

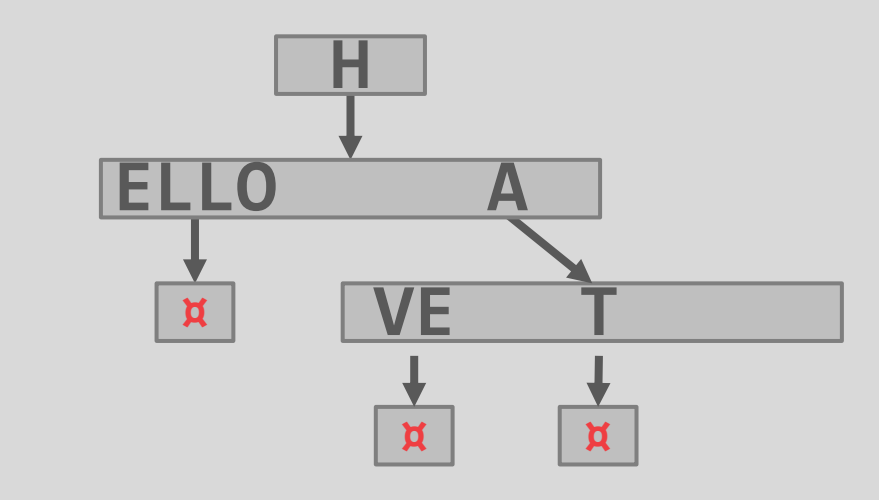
The index supports four different internal node types with different capacities.

Pack in multiple digits into a single node to improve cache locality.

ART: MODIFICATIONS

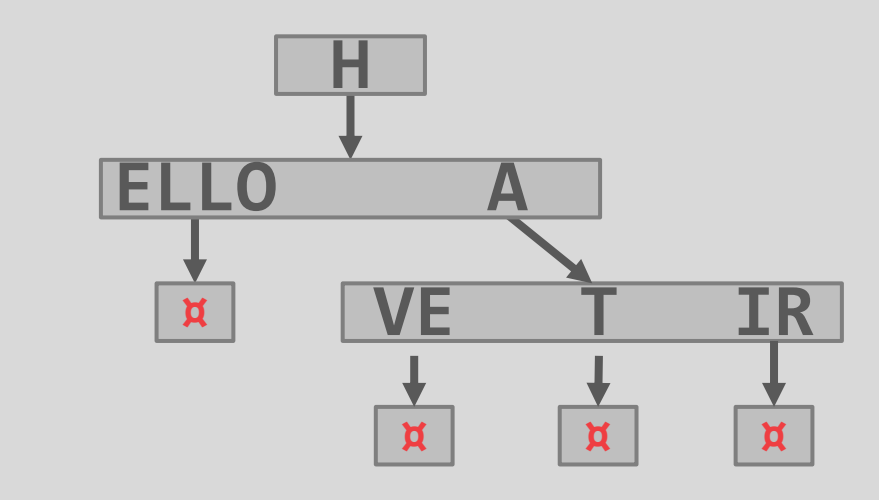


ART: MODIFICATIONS



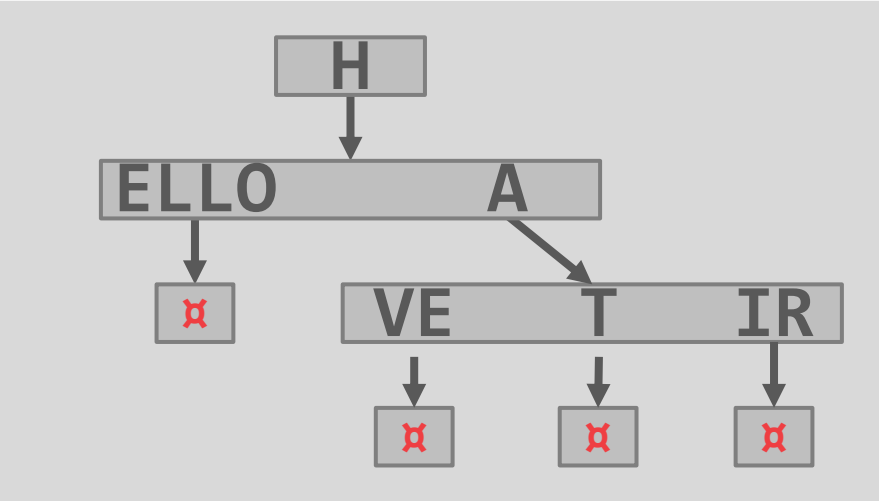
Operation: Insert **HAIR**

ART: MODIFICATIONS



Operation: Insert **HAIR**

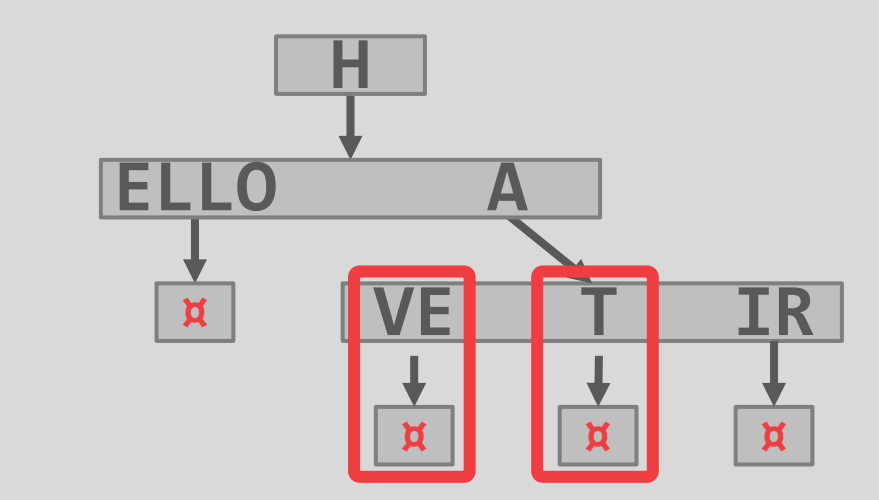
ART: MODIFICATIONS



Operation: Insert **HAIR**

Operation: Delete **HAT**, **HAVE**

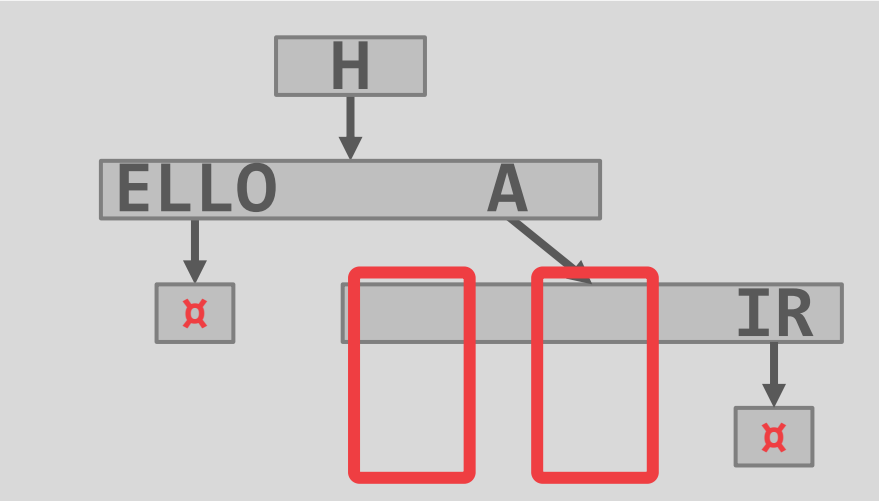
ART: MODIFICATIONS



Operation: Insert **HAIR**

Operation: Delete **HAT**, **HAVE**

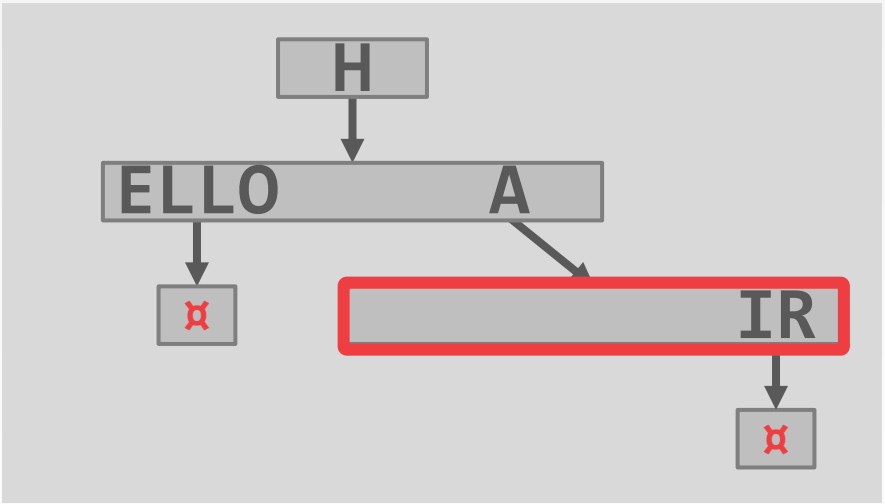
ART: MODIFICATIONS



Operation: Insert **HAIR**

Operation: Delete **HAT**, **HAVE**

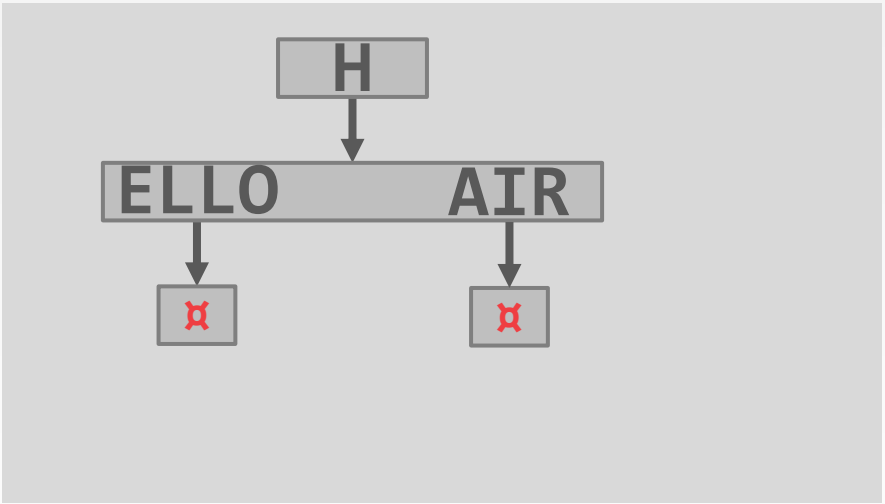
ART: MODIFICATIONS



Operation: Insert **HAIR**

Operation: Delete **HAT**, **HAVE**

ART: MODIFICATIONS



Operation: Insert **HAIR**

Operation: Delete **HAT**, **HAVE**

ART: BINARY COMPARABLE KEYS

Not all attribute types can be decomposed into binary comparable digits for a radix tree.

→ **Unsigned Integers:** Byte order must be flipped to big endian representation for little endian machines (x86).
→ **Signed Integers:** Flip two's-complement so that negative numbers are smaller than positive.
→ **Floats**: Classify into group (neg vs. pos, normalized vs. denormalized), then store as unsigned integer.
→ **Compound**: Transform each attribute separately.

ART: BINARY COMPARABLE KEYS

Int Key: 168496141

ART: BINARY COMPARABLE KEYS

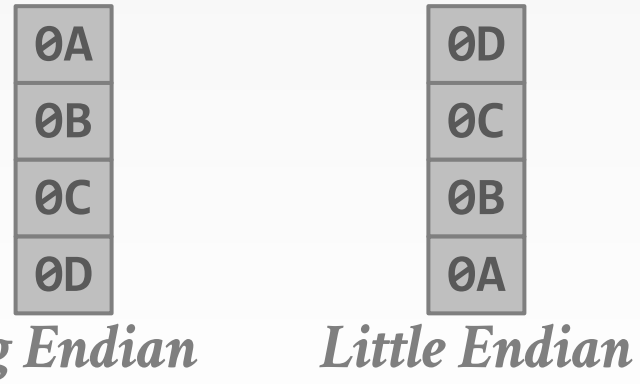
Int Key: 168496141

Hex Key: 0A 0B 0C 0D

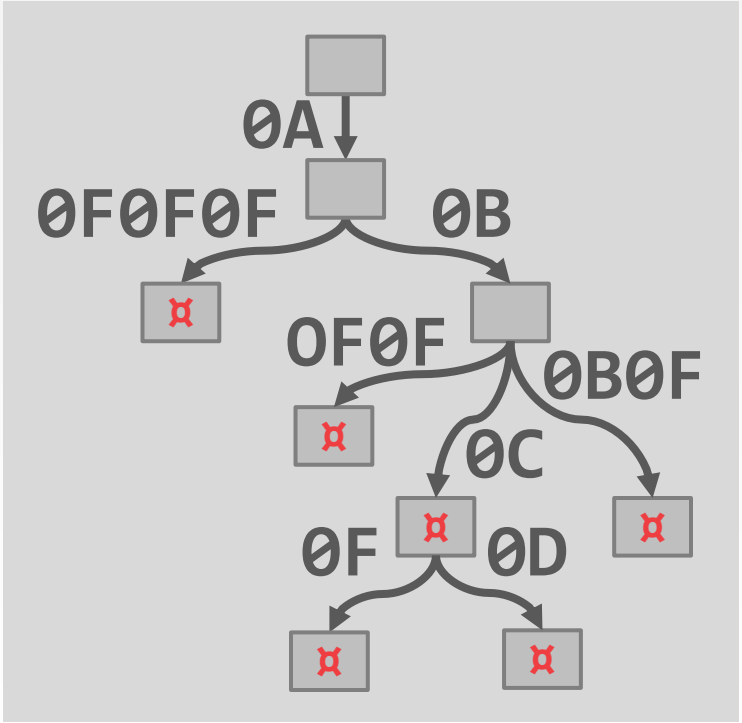
ART: BINARY COMPARABLE KEYS

Int Key: 168496141

Hex Key: 0A 0B 0C 0D

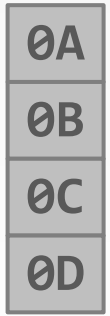


ART: BINARY COMPARABLE KEYS



Int Key: 168496141

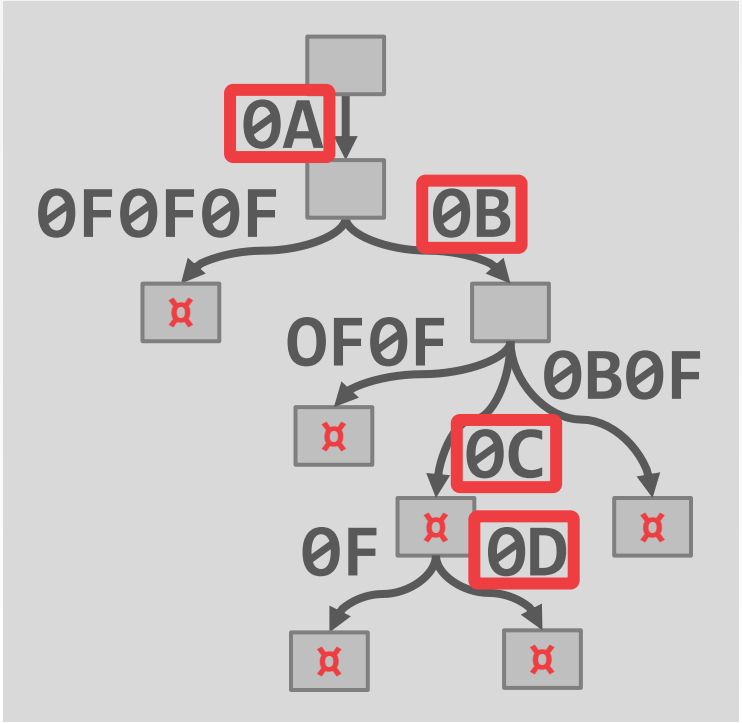
Hex Key: 0A 0B 0C 0D



Big Endian

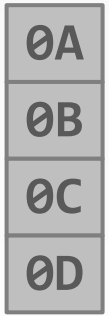
Little Endian

ART: BINARY COMPARABLE KEYS

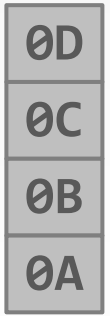


Int Key: 168496141

Hex Key: 0A 0B 0C 0D

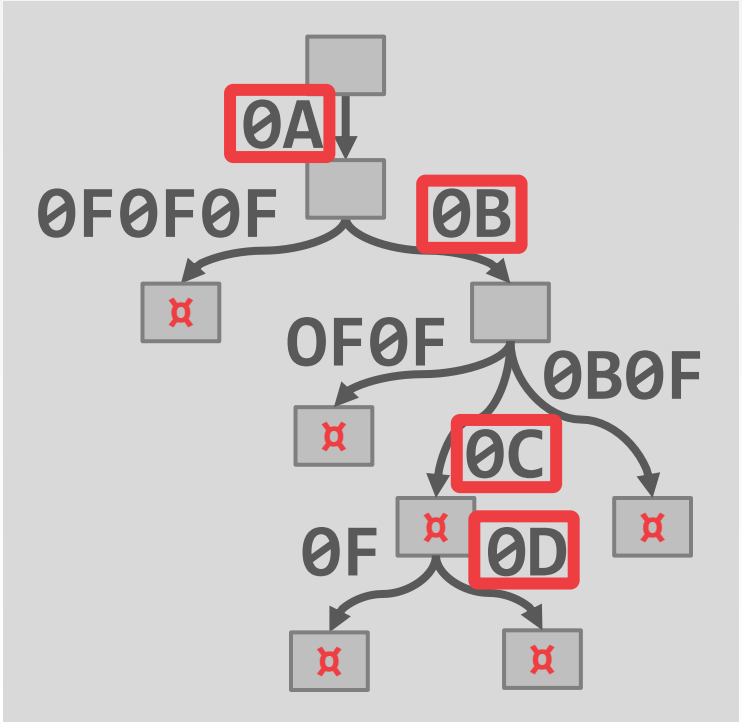


Big Endian



Little Endian

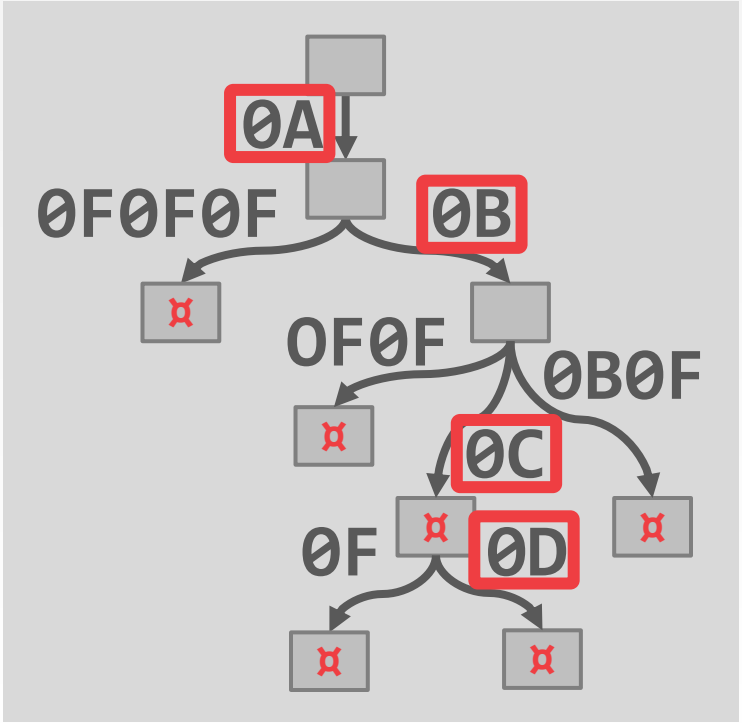
ART: BINARY COMPARABLE KEYS



Int Key: 168496141

Hex Key: 0A 0B 0C 0D

ART: BINARY COMPARABLE KEYS



Int Key: 168496141

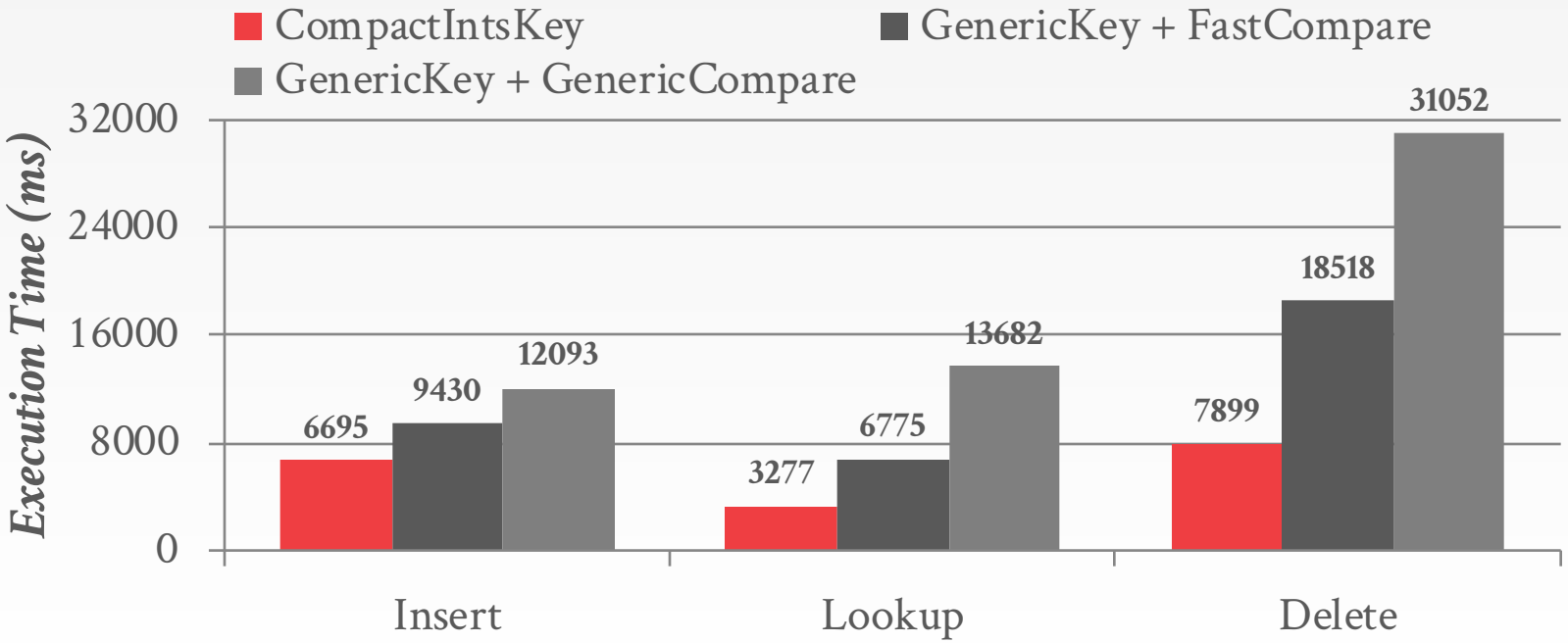
Hex Key: 0A 0B 0C 0D

Lookup: 658205

Hex: 0A 0B 1D

BINARY COMPARABLE KEYS

Peloton w/ Bw-Tree Index
Data Set: 10m keys (three 64-bit ints)



CONCURRENT ART INDEX

HyPer's ART is **not** latch-free.

→ The authors argue that it would be a significant amount of work to make it latch-free.

Approach #1: Optimistic Lock Coupling

Approach #2: Read-Optimized Write Exclusion

THE ART OF PRACTICAL SYNCHRONIZATION
DaMoN 2016

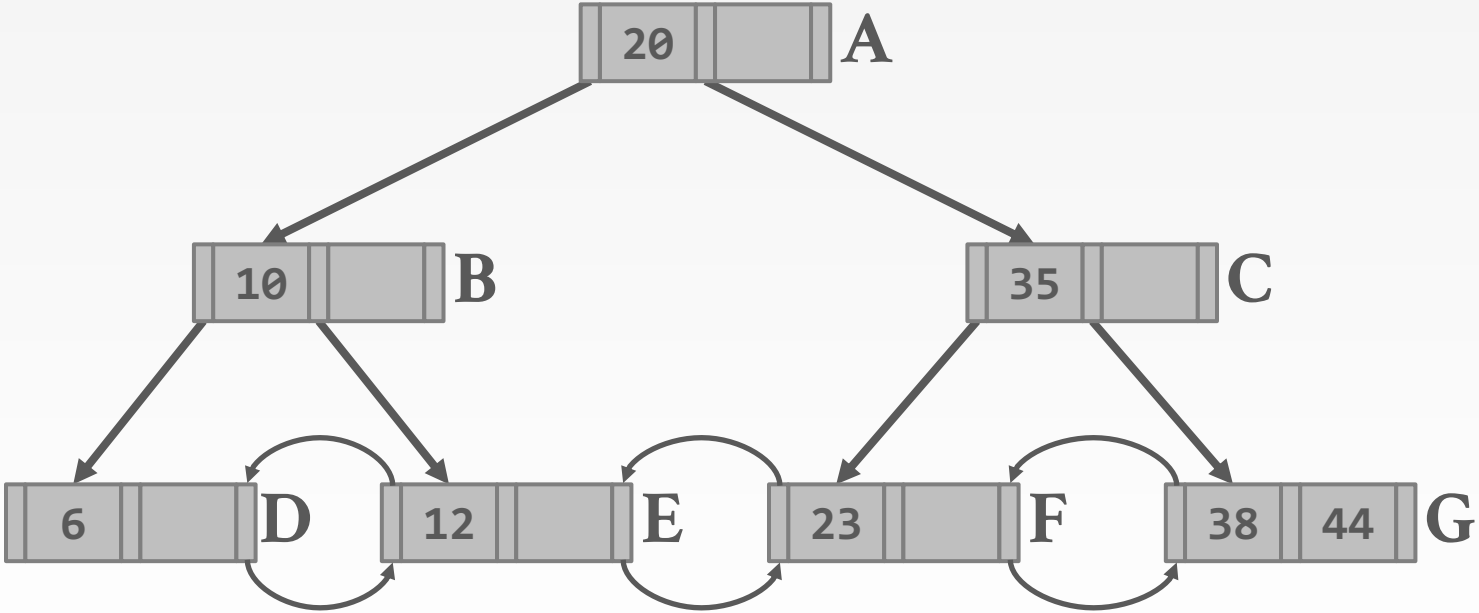
OPTIMISTIC LOCK COUPLING

Optimistic crabbing scheme where writers are not blocked on readers.

→ Writers increment counter when they acquire latch.
→ Readers can proceed if a node's latch is available.
→ It then checks whether the latch's counter has changed from when it checked the latch.

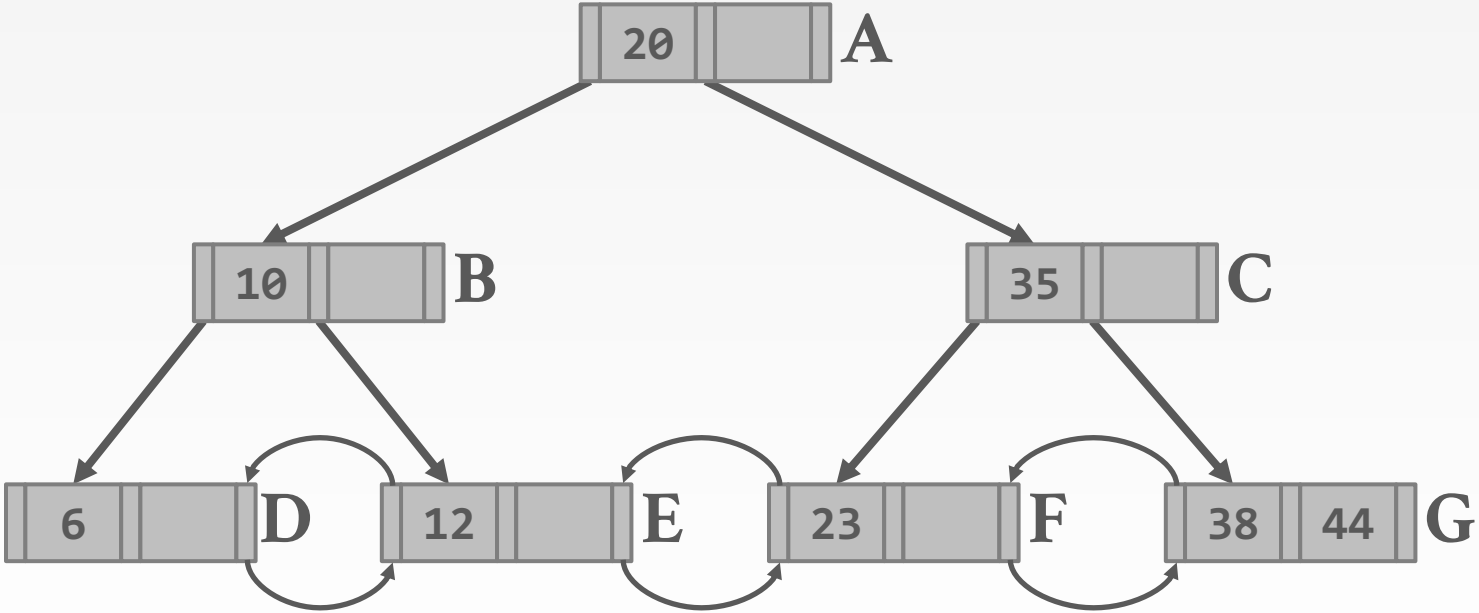
OPTIMISTIC LOCK COUPLING

SEARCH 44



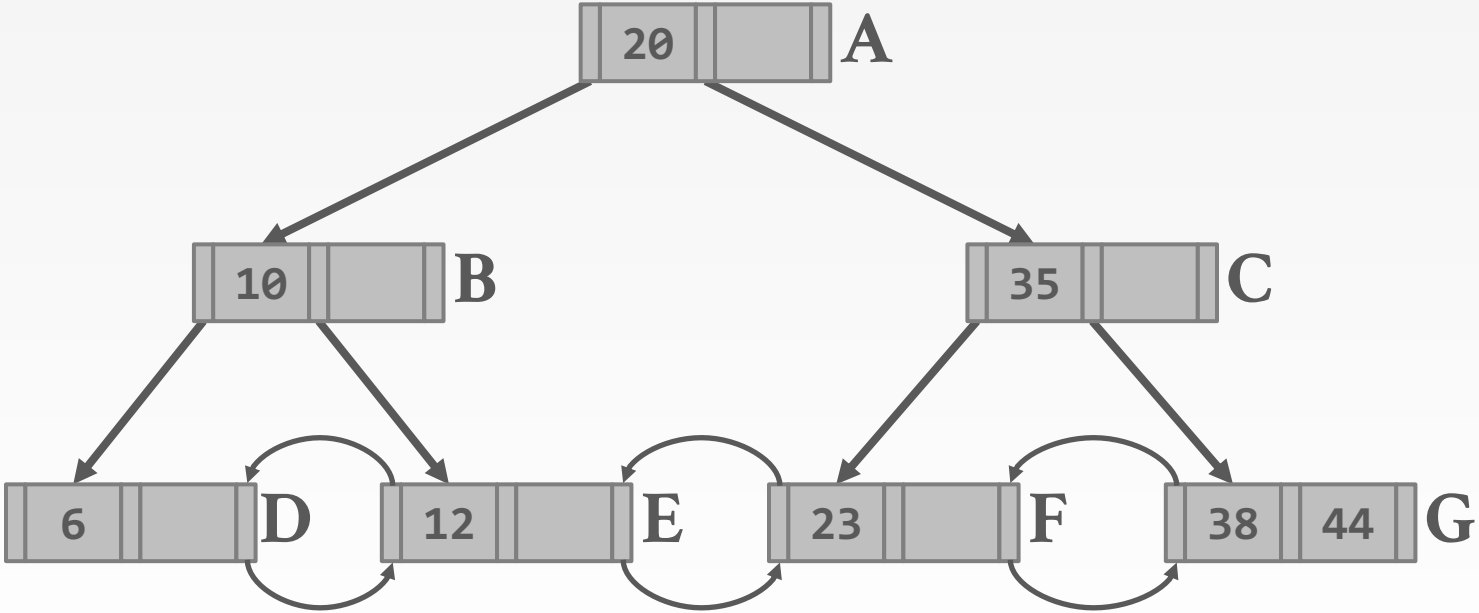
OPTIMISTIC LOCK COUPLING

SEARCH 44



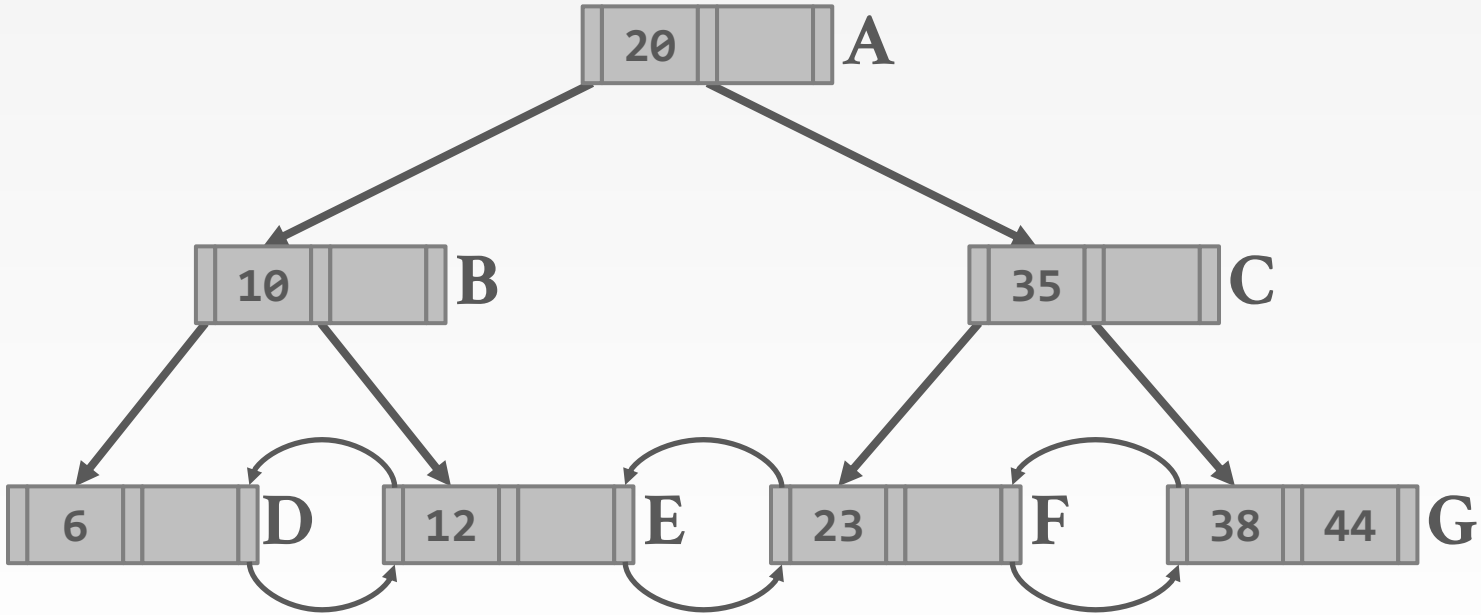
OPTIMISTIC LOCK COUPLING

SEARCH 44



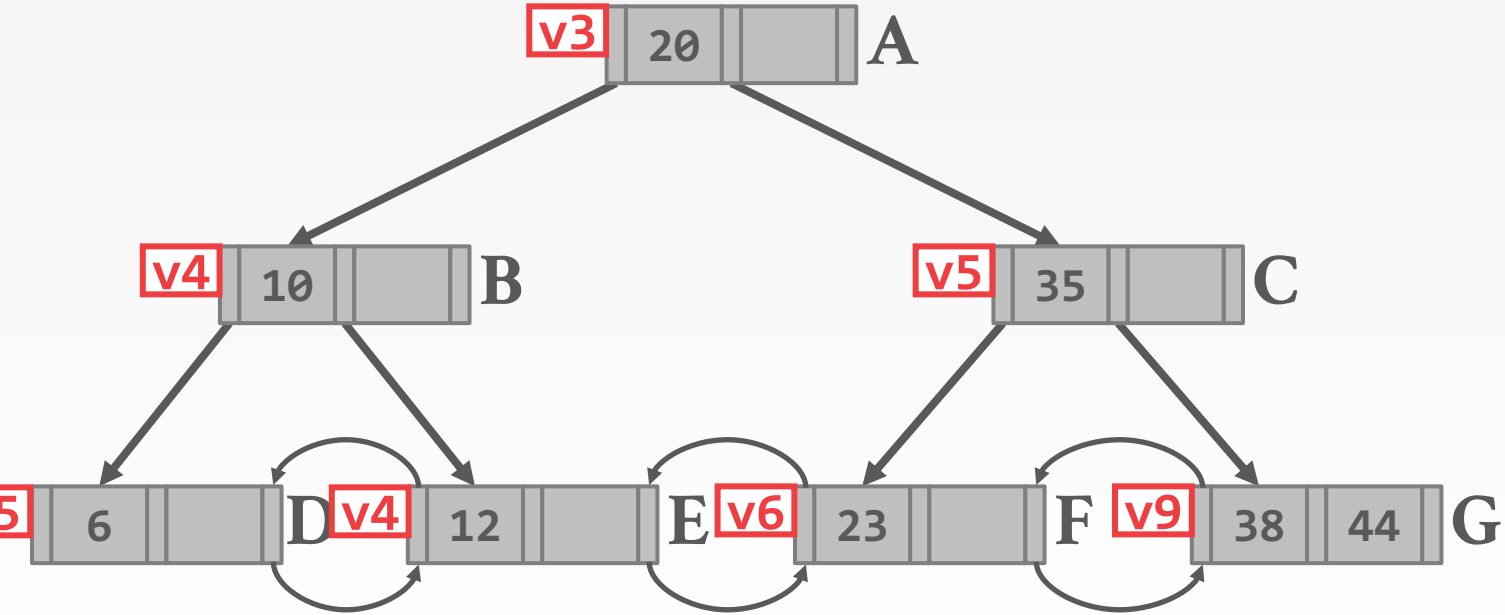
OPTIMISTIC LOCK COUPLING

SEARCH 44



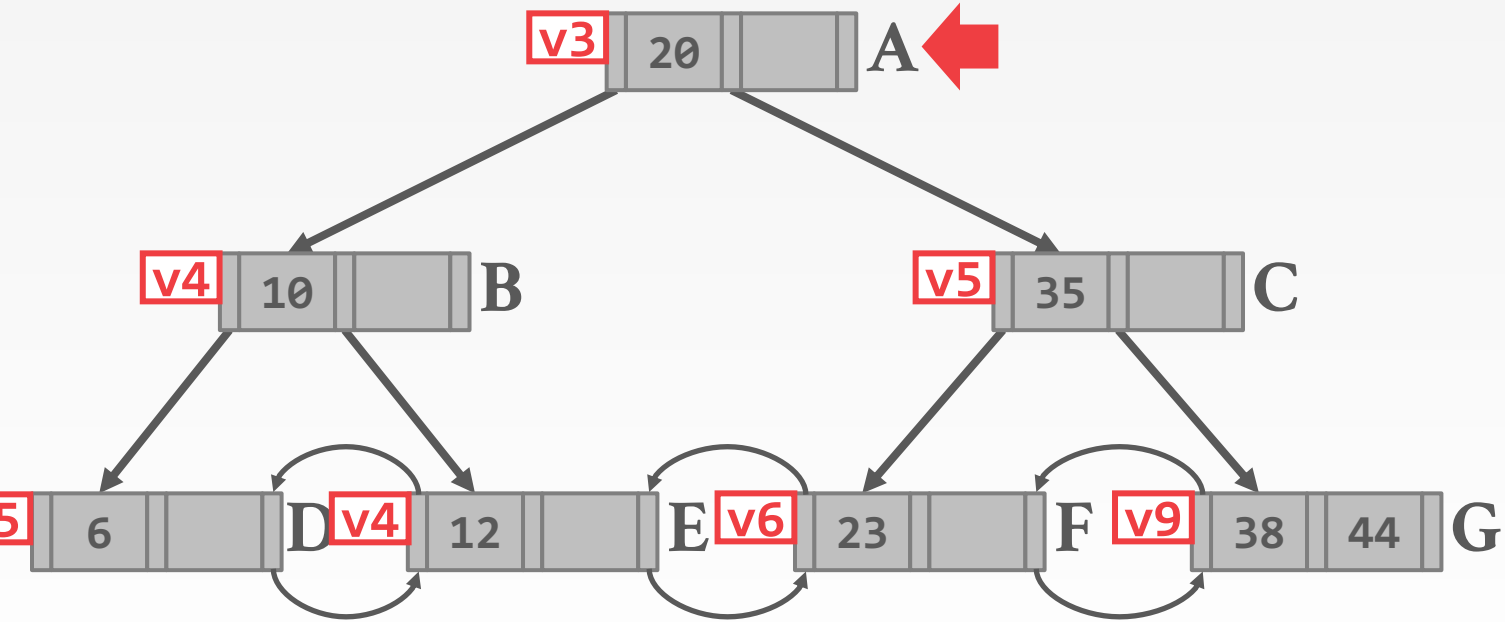
OPTIMISTIC LOCK COUPLING

SEARCH 44



OPTIMISTIC LOCK COUPLING

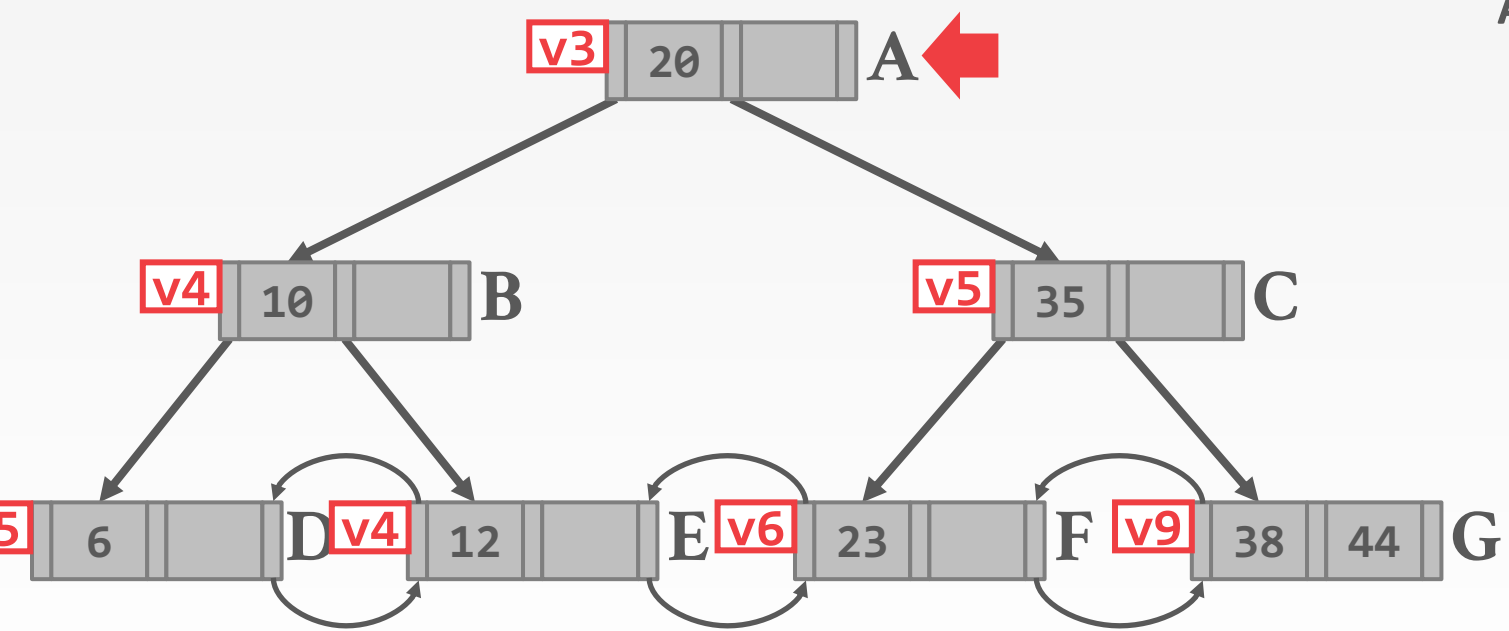
SEARCH 44



OPTIMISTIC LOCK COUPLING

SEARCH 44

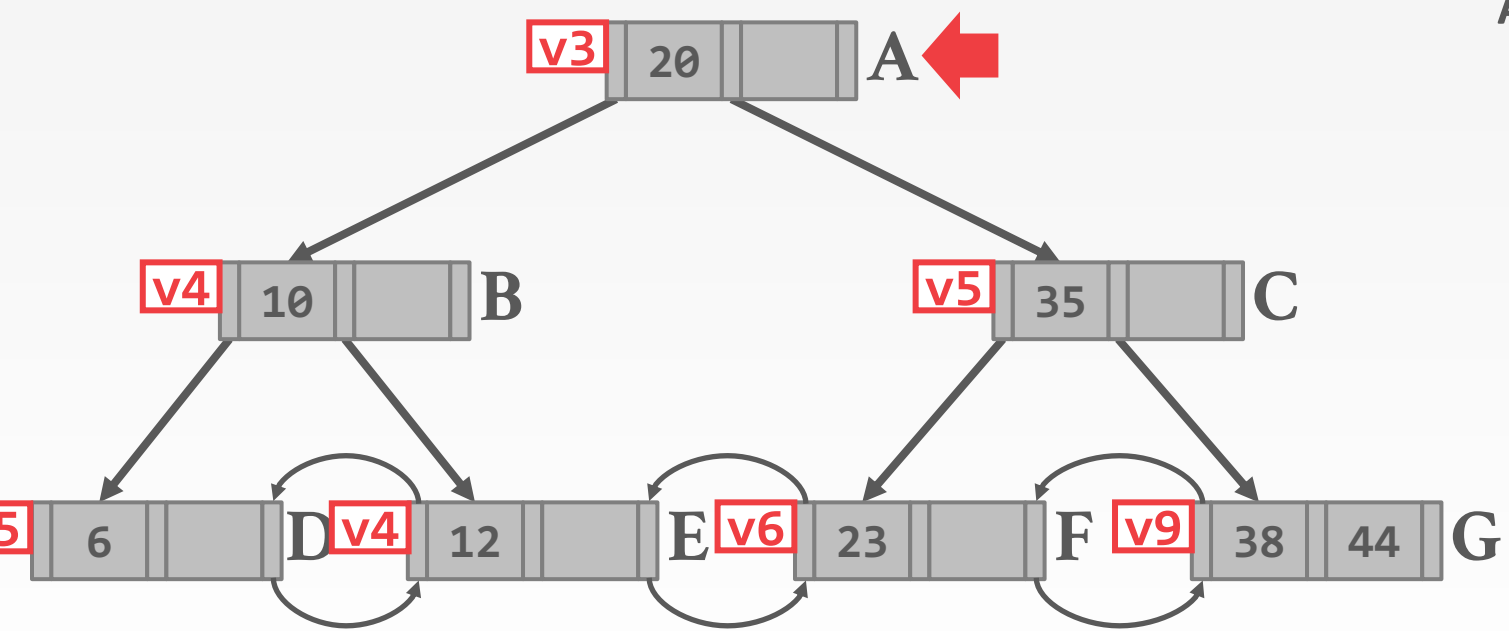
A: Read v3
A: Search Node



OPTIMISTIC LOCK COUPLING

SEARCH 44

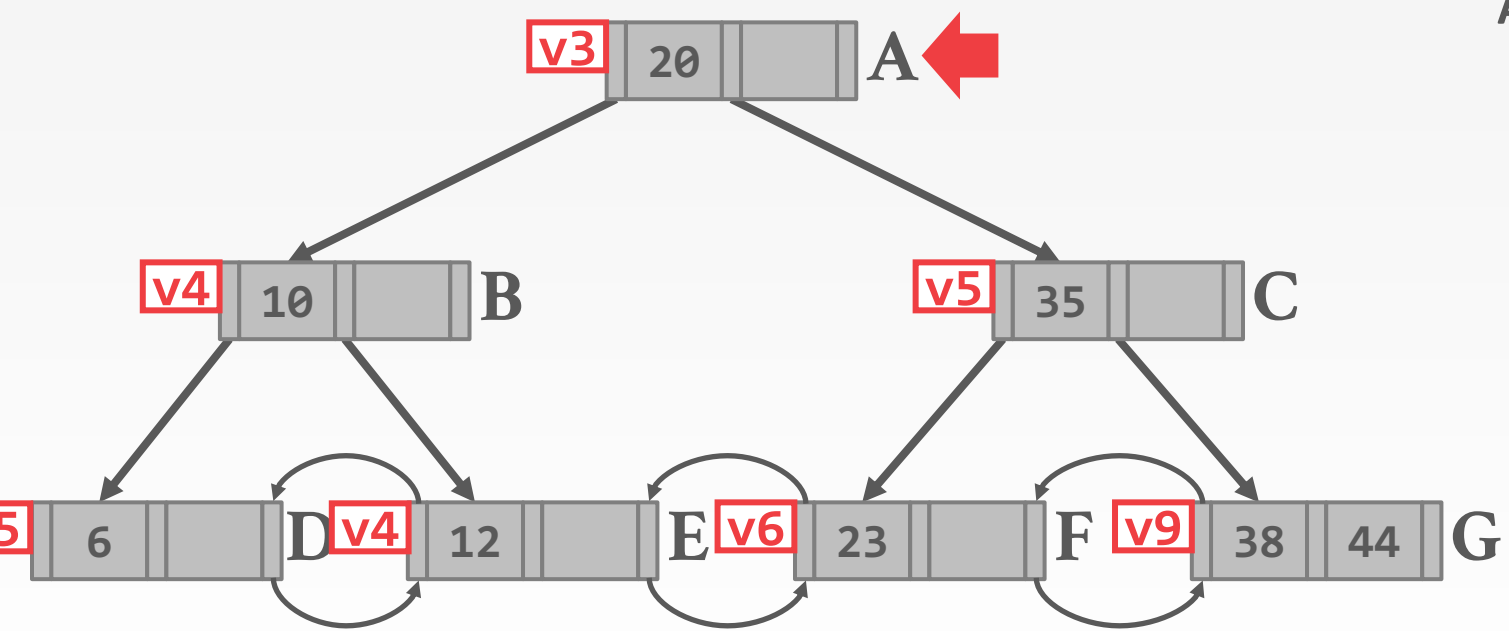
A: Read v3
A: Search Node



OPTIMISTIC LOCK COUPLING

SEARCH 44

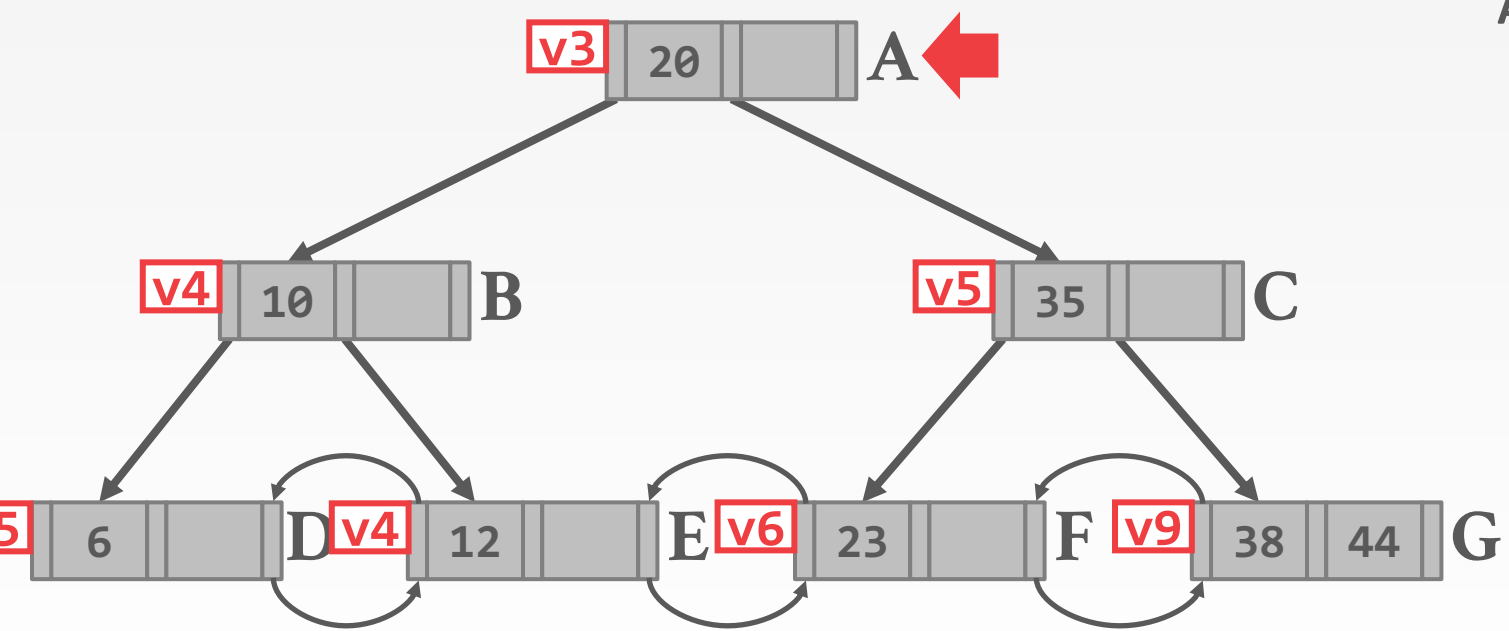
A: Read v3
A: Search Node



OPTIMISTIC LOCK COUPLING

SEARCH 44

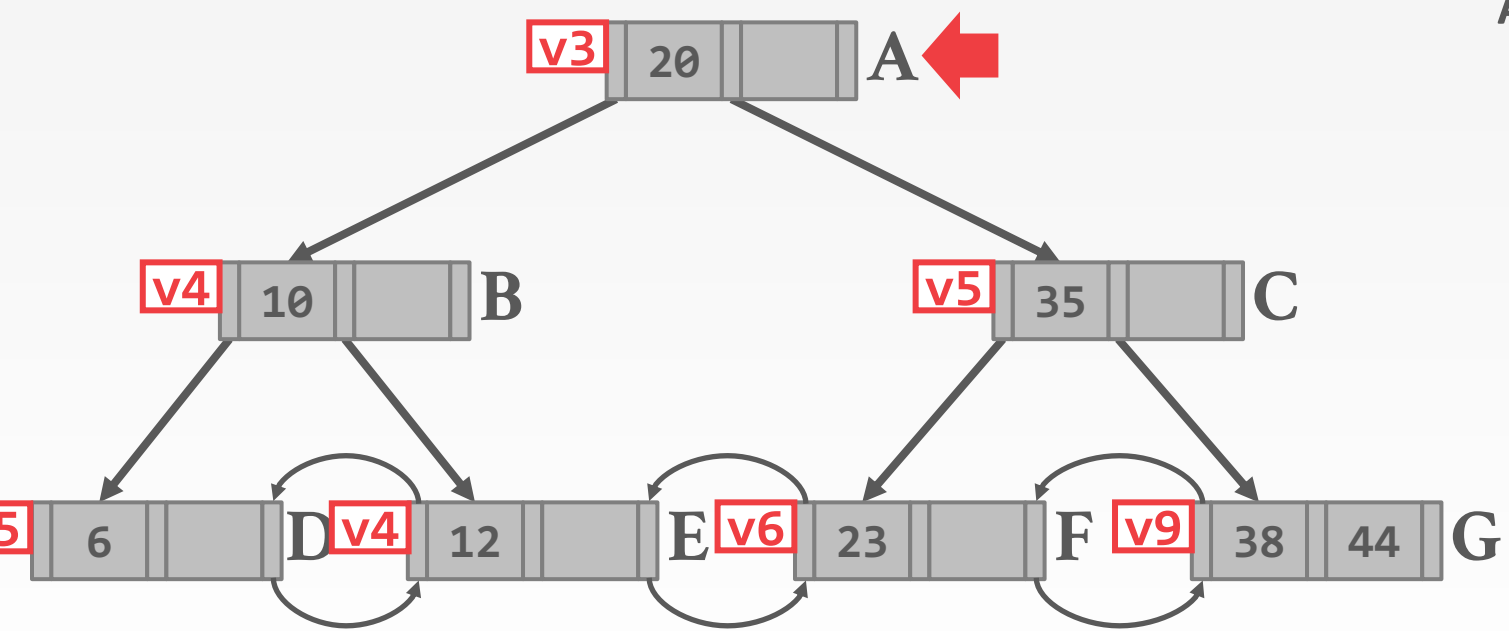
A: Read v3
A: Search Node



OPTIMISTIC LOCK COUPLING

SEARCH 44

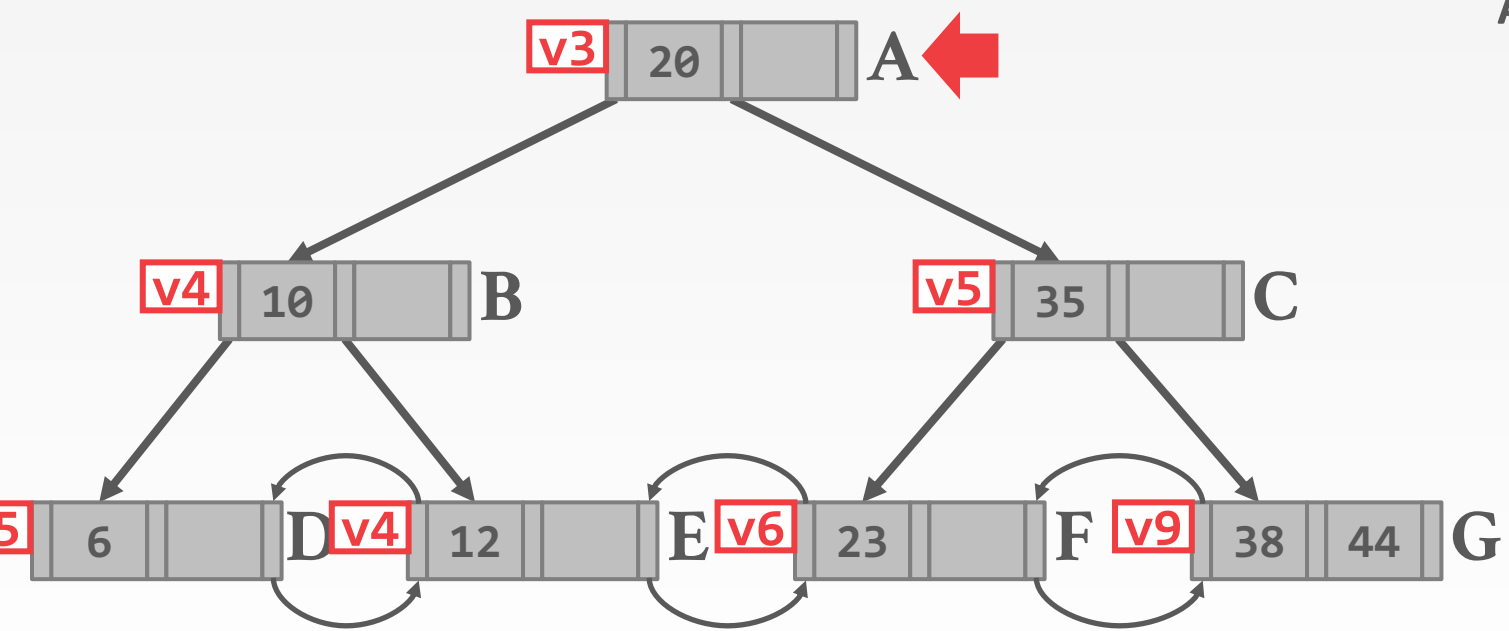
A: Read v3
A: Search Node



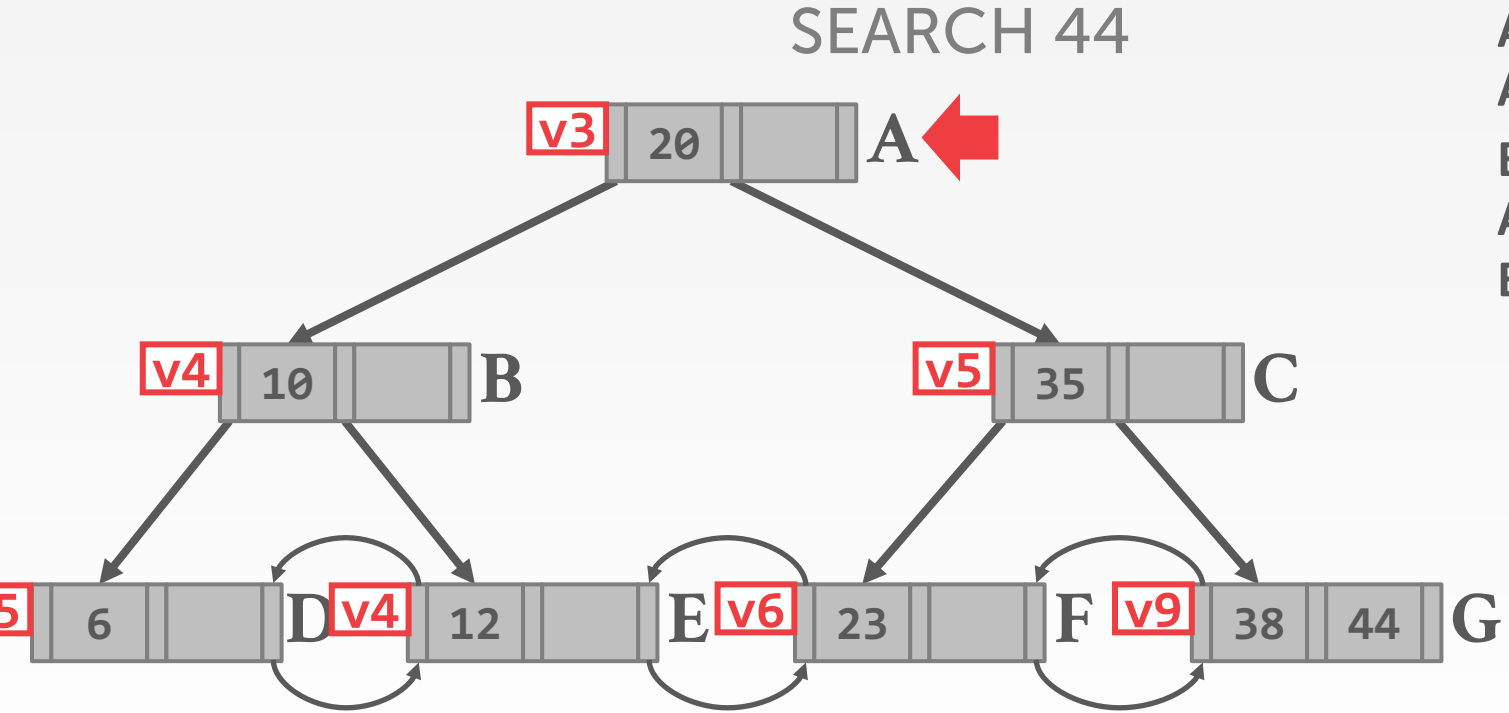
OPTIMISTIC LOCK COUPLING

SEARCH 44

A: Read v3
A: Search Node

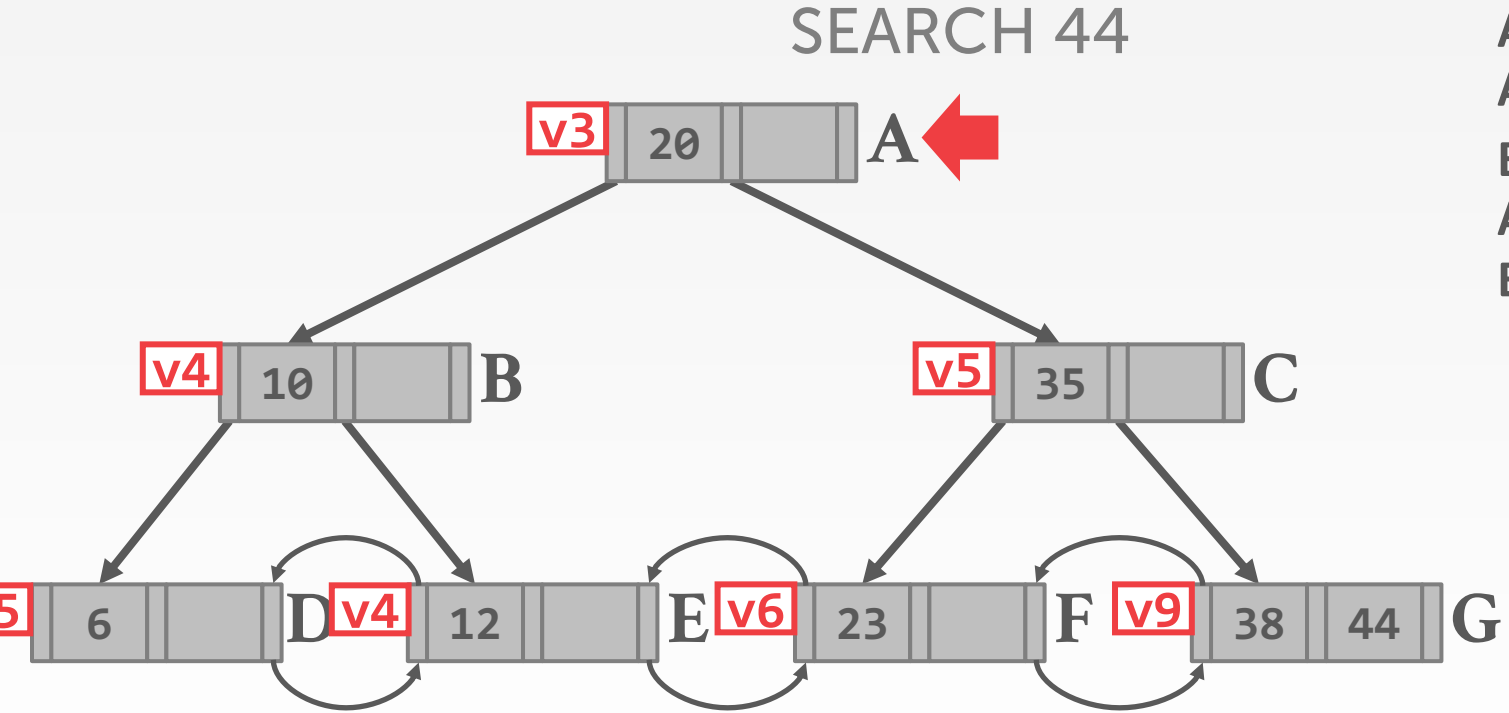


OPTIMISTIC LOCK COUPLING



A: Read v3
A: Search Node
B: Read v5
A: Recheck v3
B: Search Node

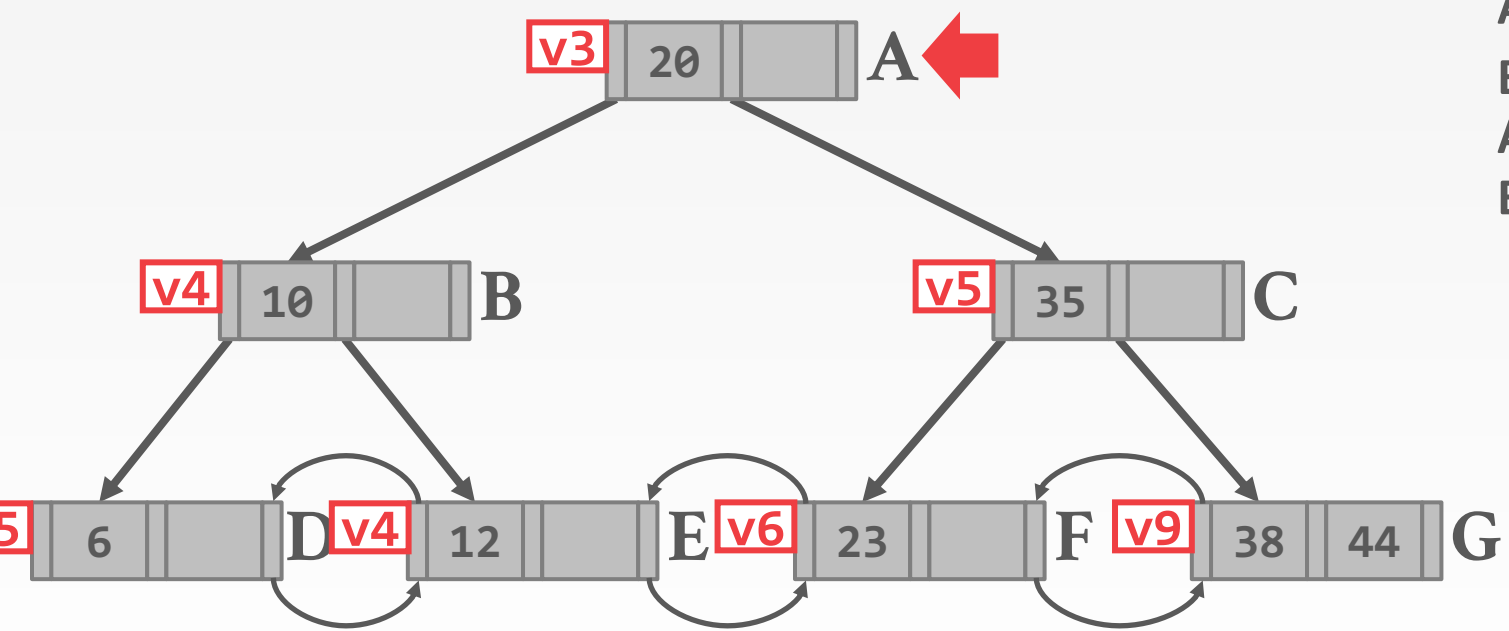
OPTIMISTIC LOCK COUPLING



A: Read v3
A: Search Node
B: Read v5
A: Recheck v3
B: Search Node

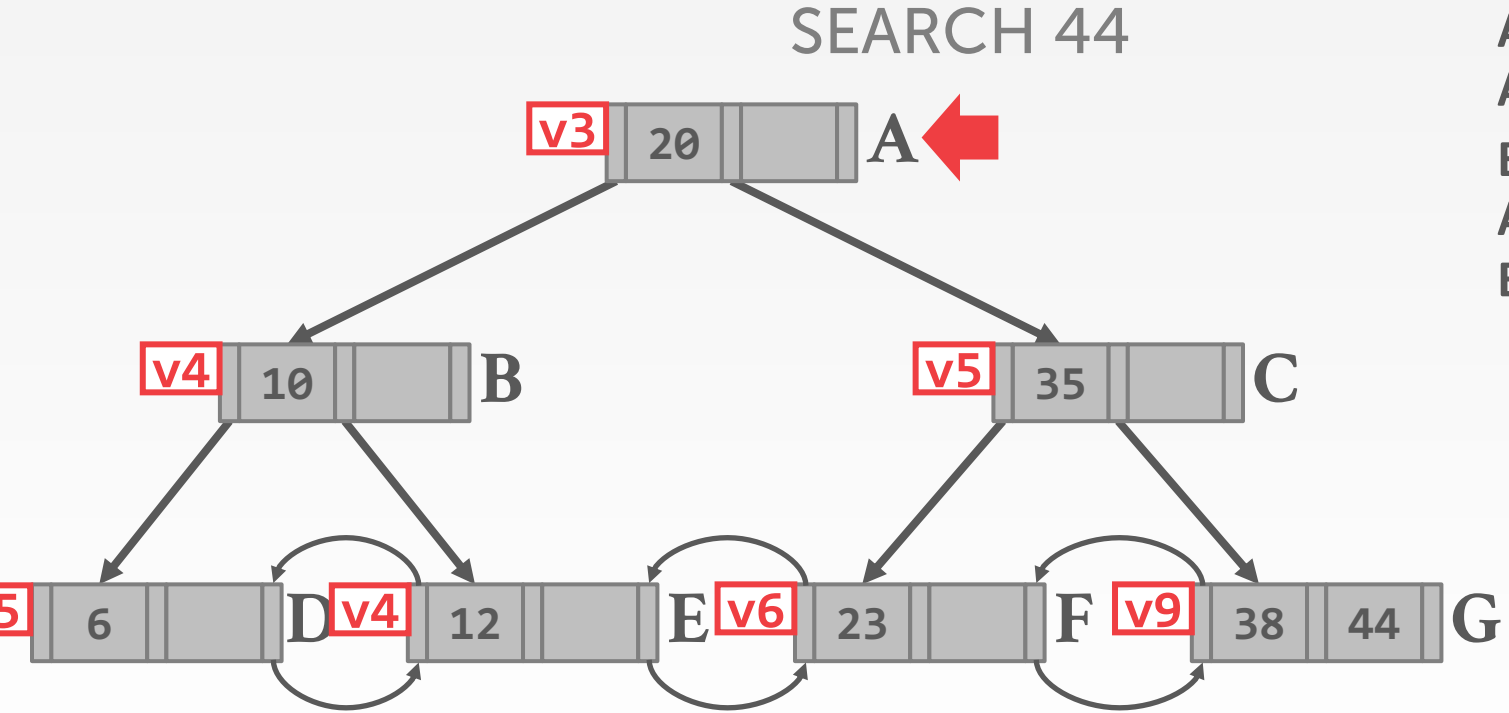
OPTIMISTIC LOCK COUPLING

SEARCH 44



A: Read v3
A: Search Node
B: Read v5
A: Recheck v3
B: Search Node

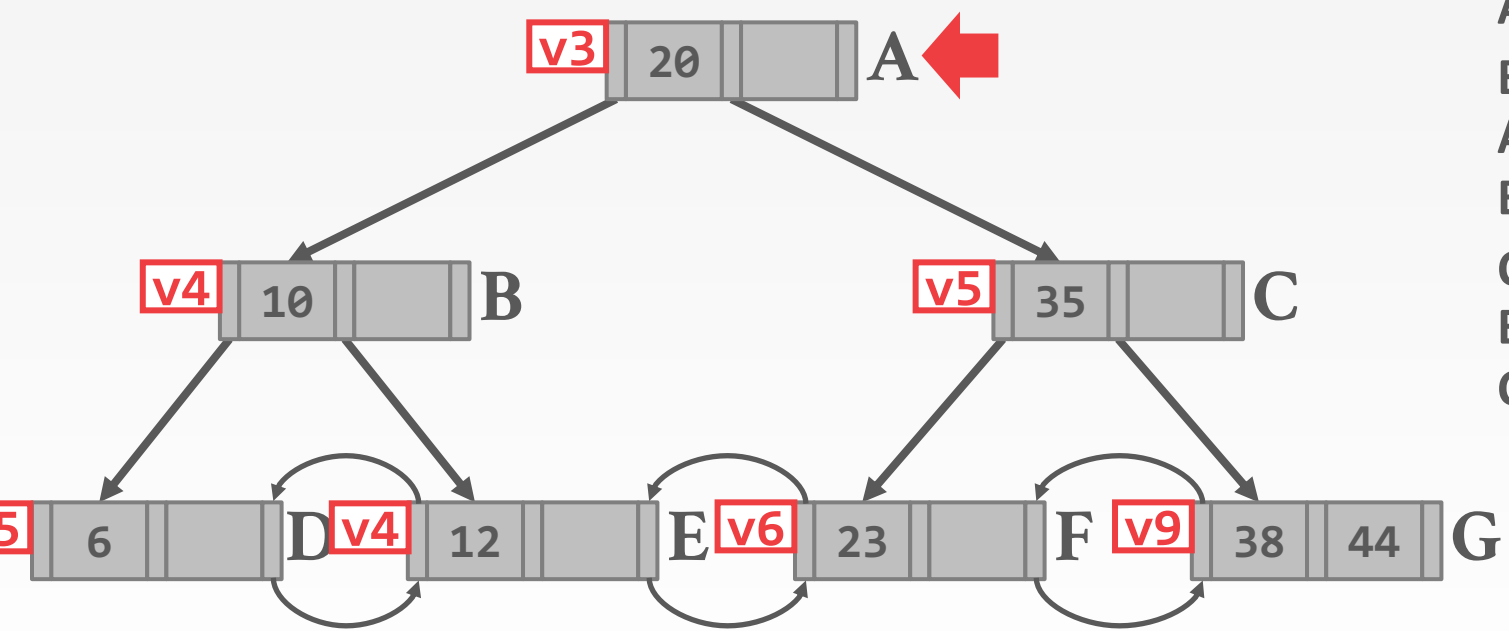
OPTIMISTIC LOCK COUPLING



A: Read v3
A: Search Node
B: Read v5
A: Recheck v3
B: Search Node

OPTIMISTIC LOCK COUPLING

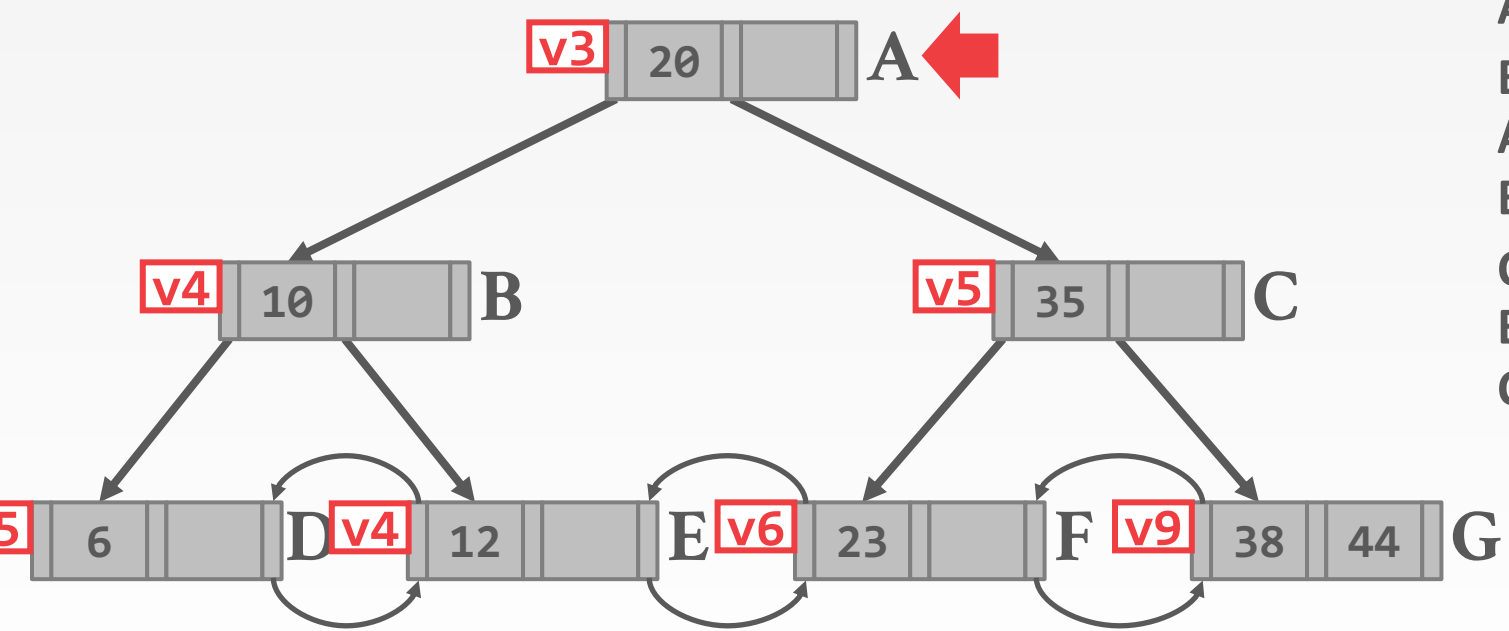
SEARCH 44



A: Read v3
A: Search Node
B: Read v5
A: Recheck v3
B: Search Node
C: Read v9
B: Recheck v5
C: Search Node

OPTIMISTIC LOCK COUPLING

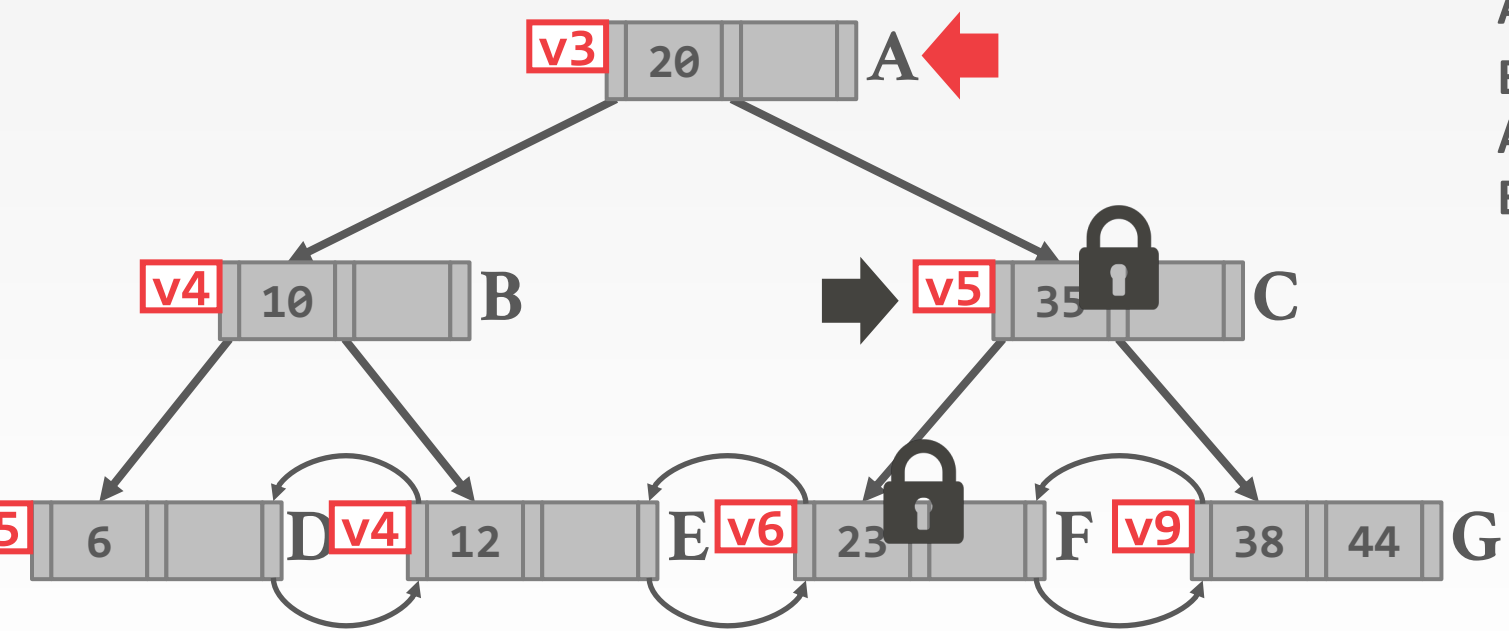
SEARCH 44



A: Read v3
A: Search Node
B: Read v5
A: Recheck v3
B: Search Node
C: Read v9
B: Recheck v5
C: Search Node

OPTIMISTIC LOCK COUPLING

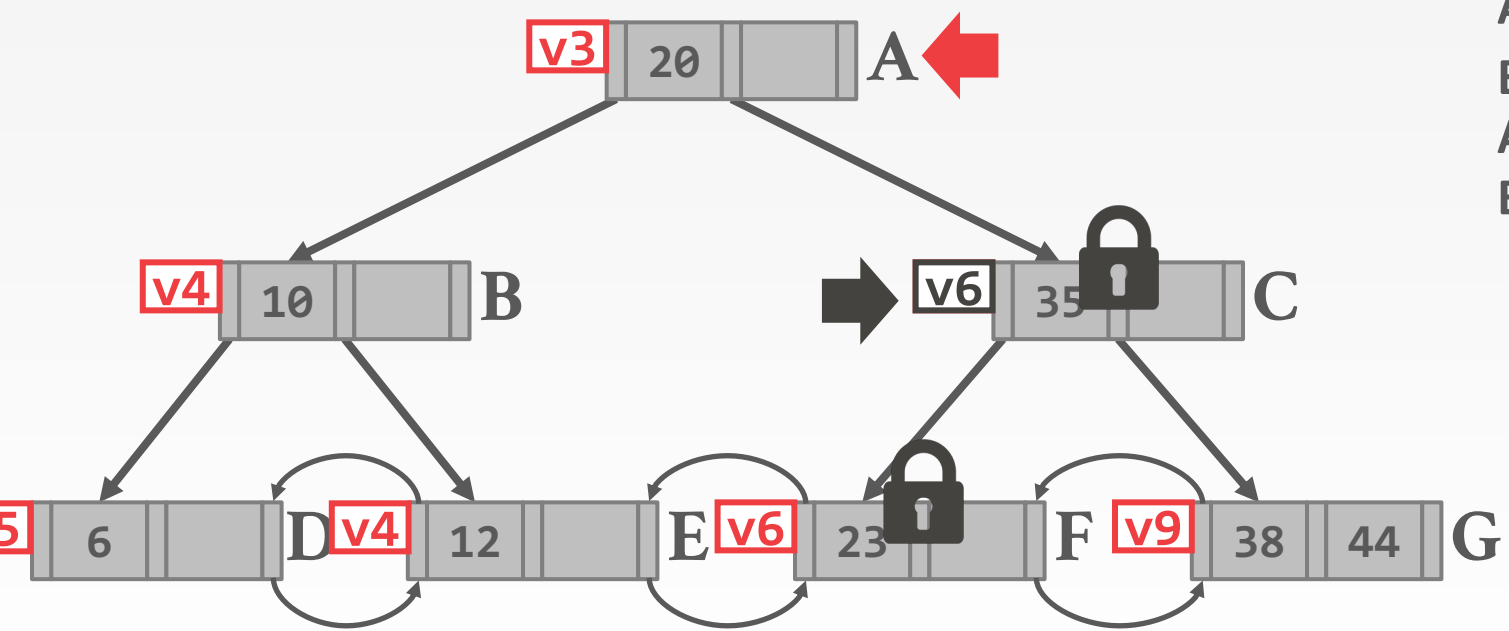
SEARCH 44



A: Read v3
A: Search Node
B: Read v5
A: Recheck v3
B: Search Node

OPTIMISTIC LOCK COUPLING

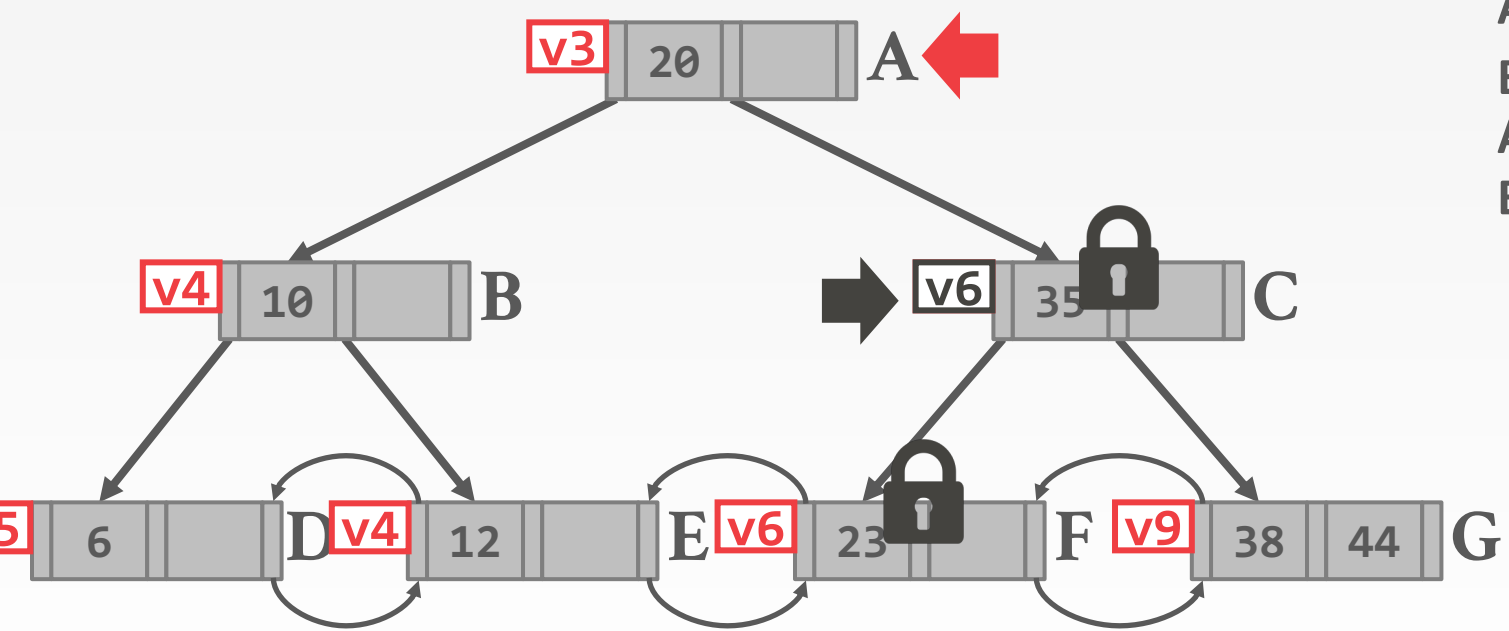
SEARCH 44



A: Read v3
A: Search Node
B: Read v5
A: Recheck v3
B: Search Node

OPTIMISTIC LOCK COUPLING

SEARCH 44



A: Read v3
A: Search Node
B: Read v5
A: Recheck v3
B: Search Node

X

READ-OPTIMIZED WRITE EXCLUSION

Each node includes an exclusive lock that blocks only other writers and not readers.

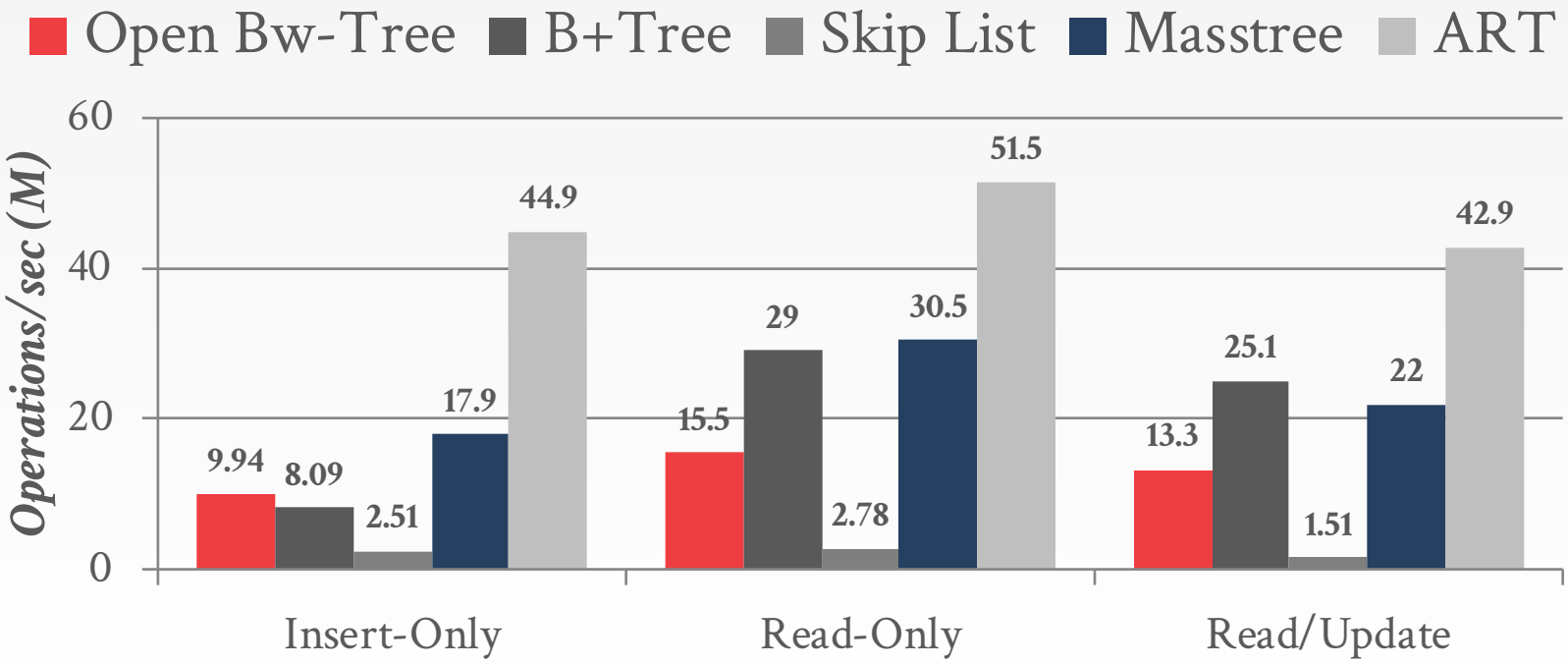
→ Readers proceed without checking versions or locks.

→ Every writer must ensure that reads are always consistent.

Requires fundamental changes to how threads make modifications to the data structure.

IN-MEMORY INDEXES

Processor: 1 socket, 10 cores w/ 2×HT
Workload: 50m Random Integer Keys (64-bit)



Source: Ziqi Wang

PARTING THOUGHTS

Andy was wrong about the Bw-Tree and latch-free indexes.

NEXT CLASS

Query Compilation

Reminder: Mid-term Exam on Feb 26th.

Reminder: Project Proposals due on Feb 26th.