Carnegie Mellon University

08 Index Concurrency Control

Intro to Database Systems
15-445/15-645
Fall 2021

Andrew Crotty
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Project #1 was due last night @ 11:59pm

Homework #2 is due Sunday, Oct 3rd @ 11:59pm

Project #2 will be released today and is due on Sunday, Oct 17th @ 11:59pm

QUESTIONS FROM LAST CLASS

(1) Non-prefix lookups in multi-attribute B+Trees

(2) Efficiently merging B+Trees

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)`

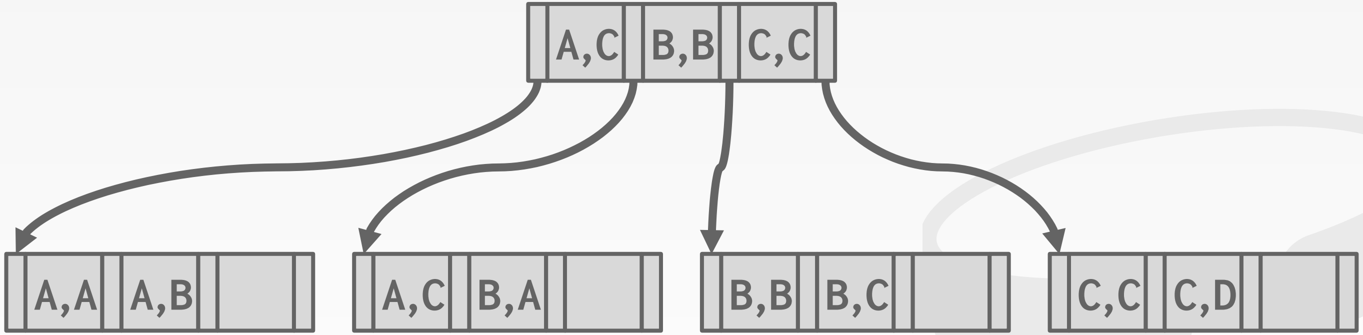
Not all DBMSs support this.

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

SELECTION CONDITIONS

Find Key=(A,B)

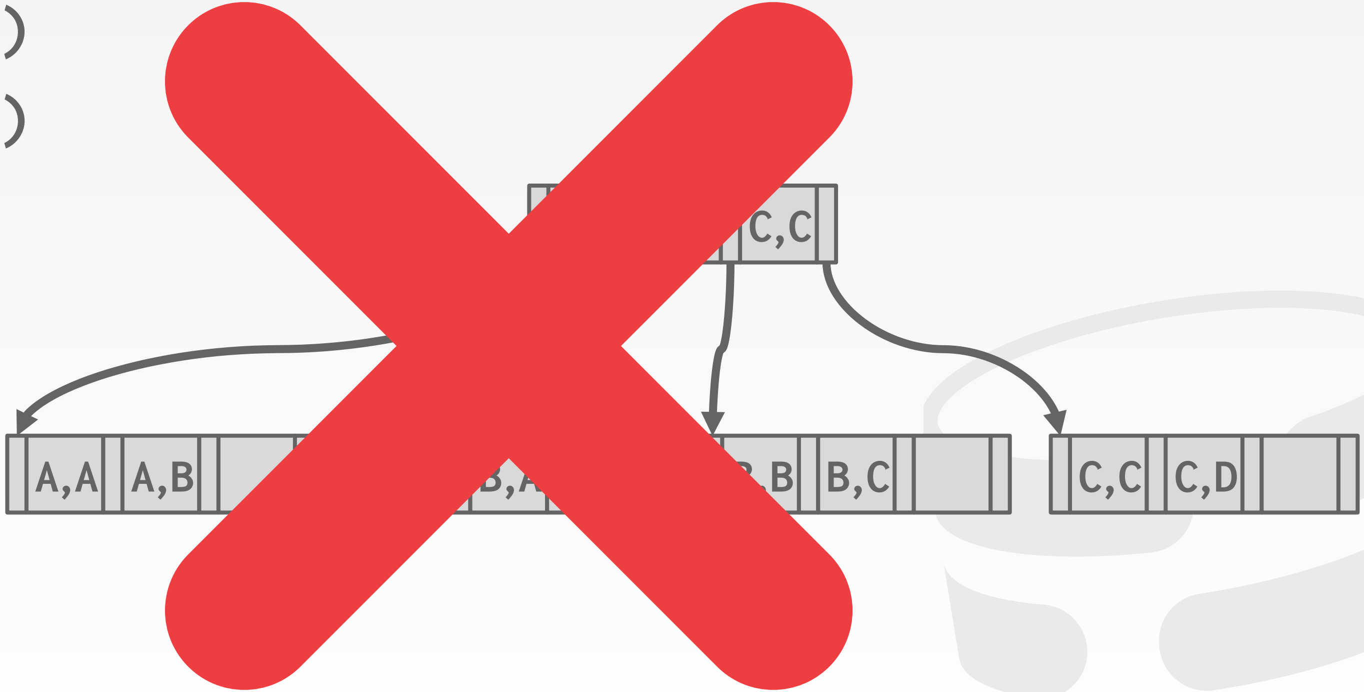
Find Key=(A,*)



SELECTION CONDITIONS

Find Key=(A,B)

Find Key=(A,*)



SELECTION CONDITIONS

Example: Index on `<col1,col2,col3>`

→ Column Values: `{A,B,C,D}`

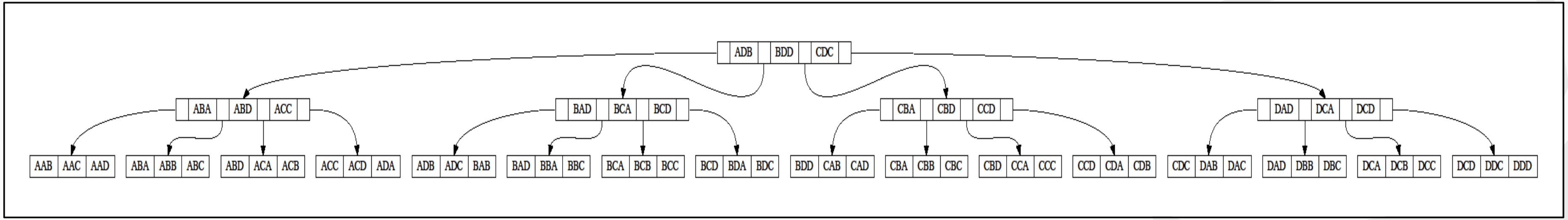
→ Supported: `col2 = B`

SELECTION CONDITIONS

Example: Index on <col1,col2,col3>

→ Column Values: {A,B,C,D}

→ Supported: col2 = B

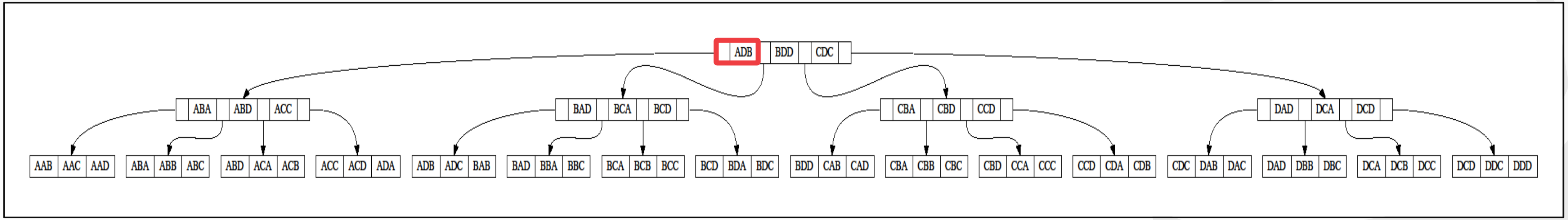


SELECTION CONDITIONS

Example: Index on `<col1,col2,col3>`

→ Column Values: `{A,B,C,D}`

→ Supported: `col2 = B`

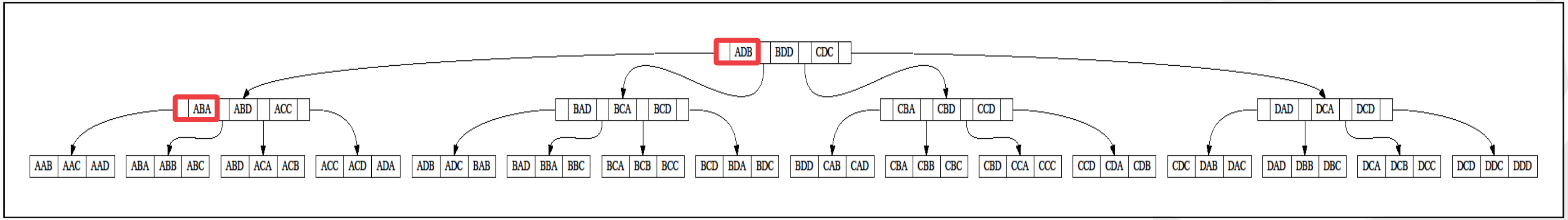


SELECTION CONDITIONS

Example: Index on <col1,col2,col3>

→ Column Values: {A,B,C,D}

→ Supported: col2 = B

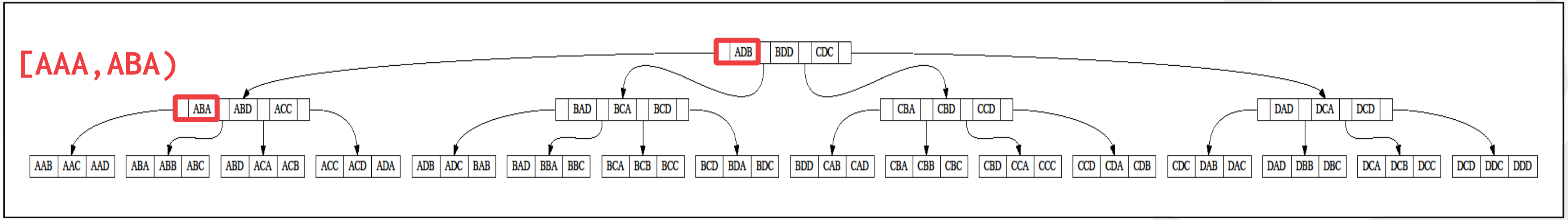


SELECTION CONDITIONS

Example: Index on <col1,col2,col3>

→ Column Values: {A,B,C,D}

→ Supported: col2 = B

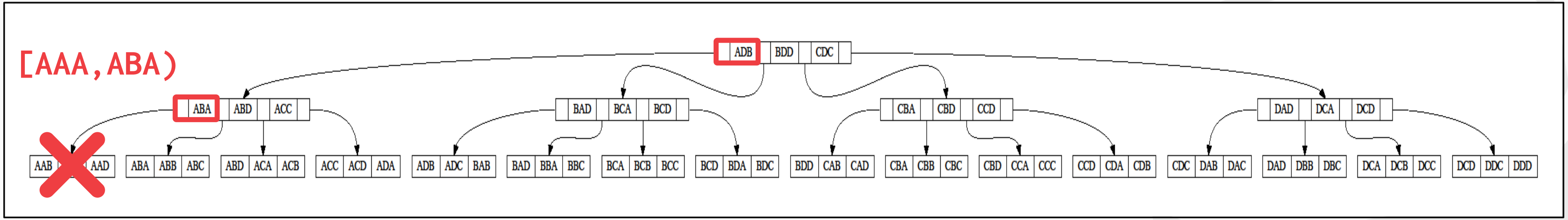


SELECTION CONDITIONS

Example: Index on <col1,col2,col3>

→ Column Values: {A,B,C,D}

→ Supported: col2 = B

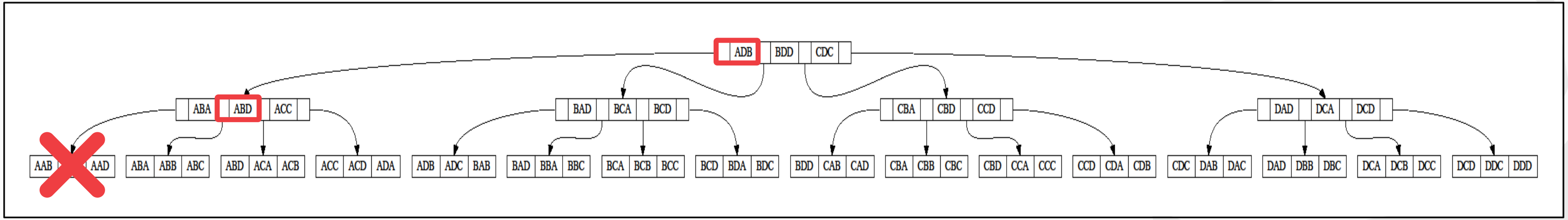


SELECTION CONDITIONS

Example: Index on **<col1,col2,col3>**

→ Column Values: **{A,B,C,D}**

→ Supported: **col2 = B**

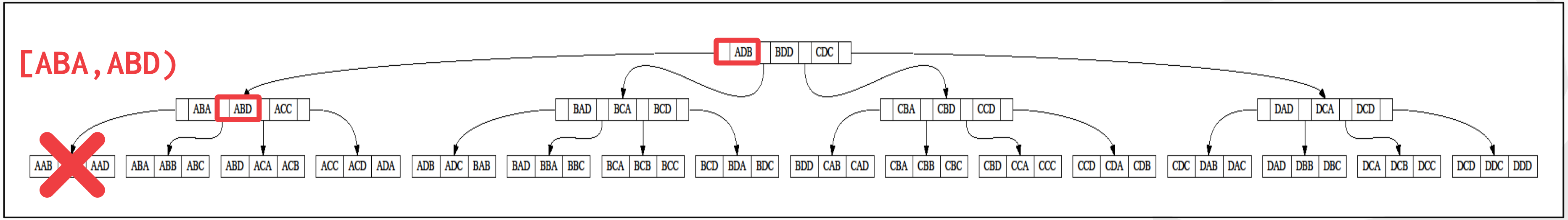


SELECTION CONDITIONS

Example: Index on <col1,col2,col3>

→ Column Values: {A,B,C,D}

→ Supported: col2 = B

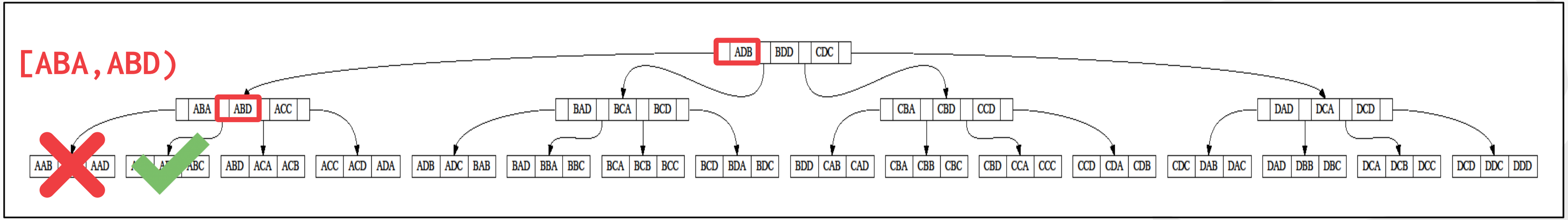


SELECTION CONDITIONS

Example: Index on `<col1,col2,col3>`

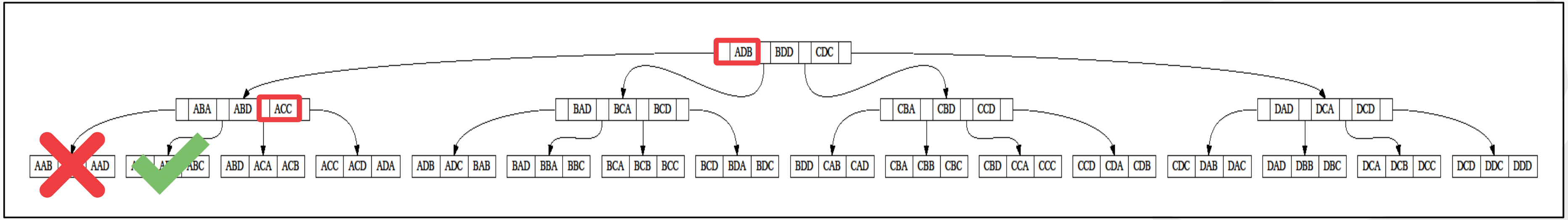
→ Column Values: `{A,B,C,D}`

→ Supported: `col2 = B`



SELECTION CONDITIONS

Example: Index on <col1,col2,col3>
→ Column Values: {A,B,C,D}
→ Supported: col2 = B

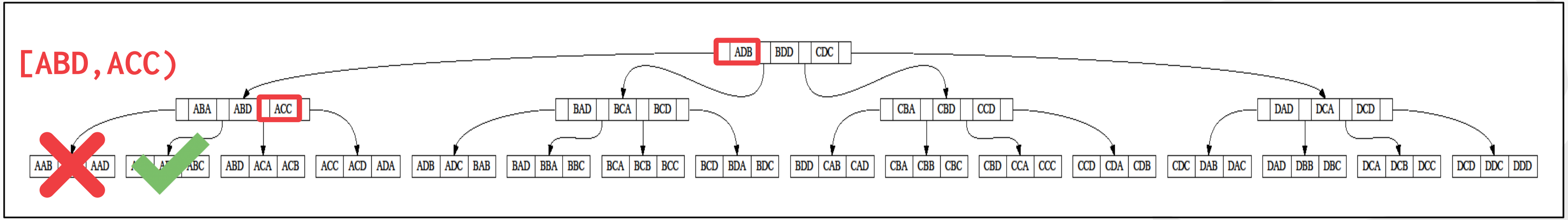


SELECTION CONDITIONS

Example: Index on **<col1,col2,col3>**

→ Column Values: **{A,B,C,D}**

→ Supported: **col2 = B**

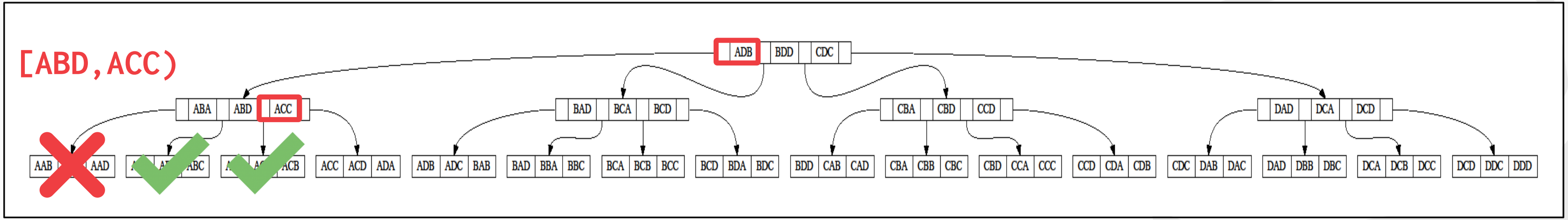


SELECTION CONDITIONS

Example: Index on `<col1,col2,col3>`

→ Column Values: `{A,B,C,D}`

→ Supported: `col2 = B`

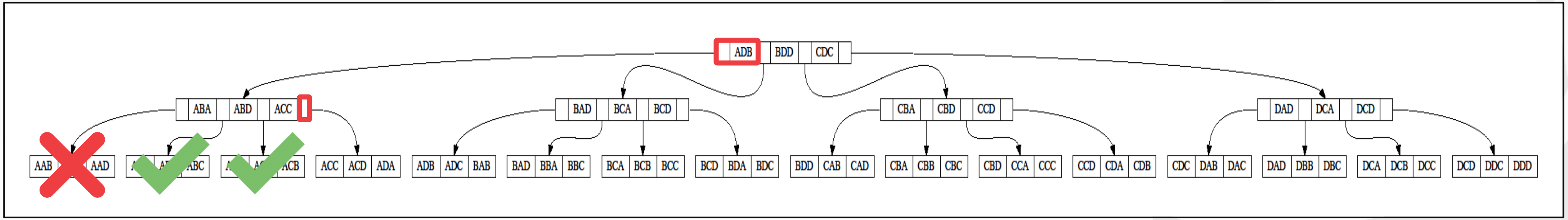


SELECTION CONDITIONS

Example: Index on <col1,col2,col3>

→ Column Values: `{A,B,C,D}`

→ Supported: `col2 = B`

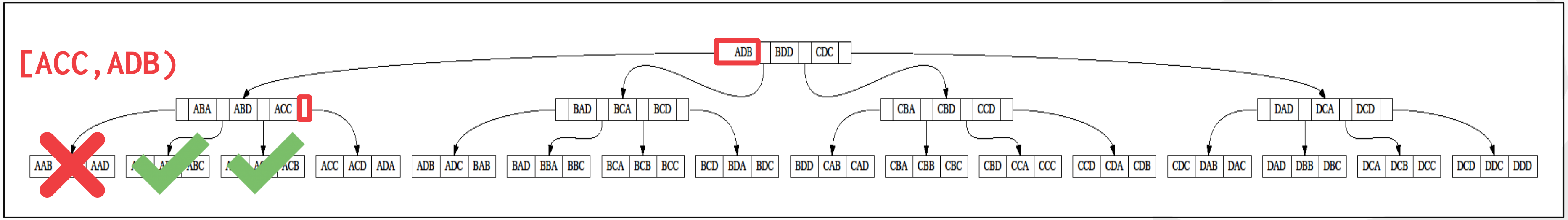


SELECTION CONDITIONS

Example: Index on <col1,col2,col3>

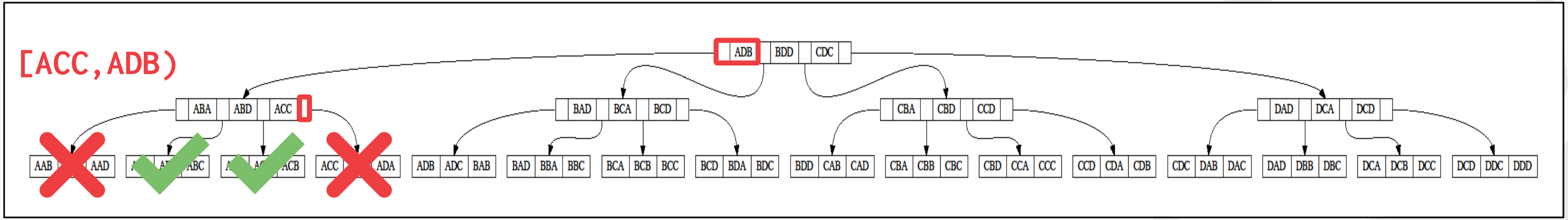
→ Column Values: {A,B,C,D}

→ Supported: col2 = B



SELECTION CONDITIONS

Example: Index on <col1,col2,col3>
→ Column Values: {A,B,C,D}
→ Supported: col2 = B

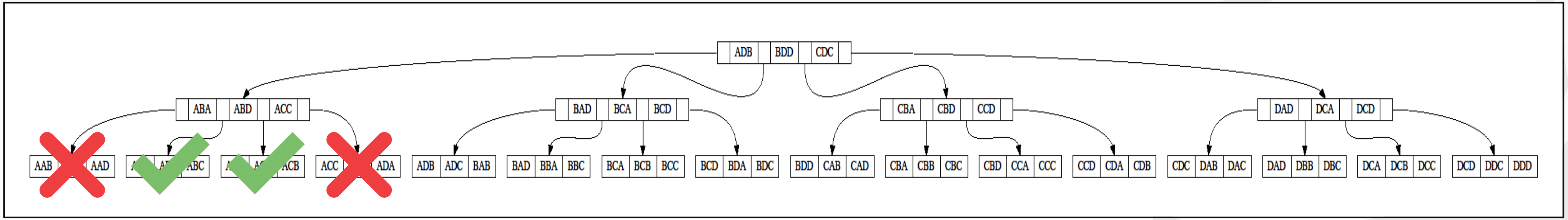


SELECTION CONDITIONS

Example: Index on <col1,col2,col3>

→ Column Values: {A,B,C,D}

→ Supported: col2 = B

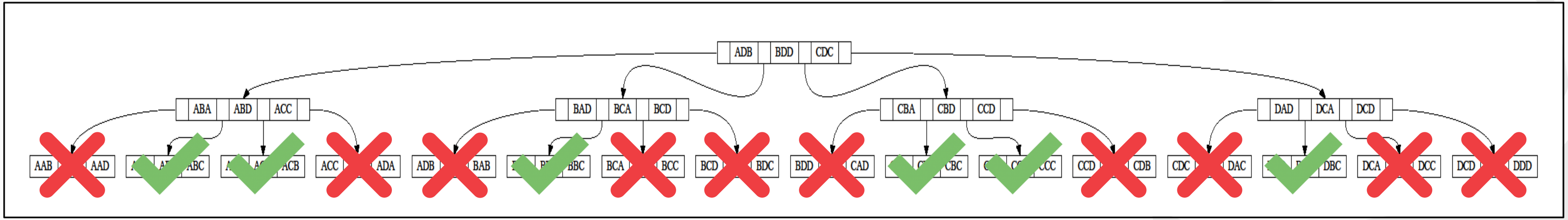


SELECTION CONDITIONS

Example: Index on `<col1,col2,col3>`

→ Column Values: `{A,B,C,D}`

→ Supported: `col2 = B`

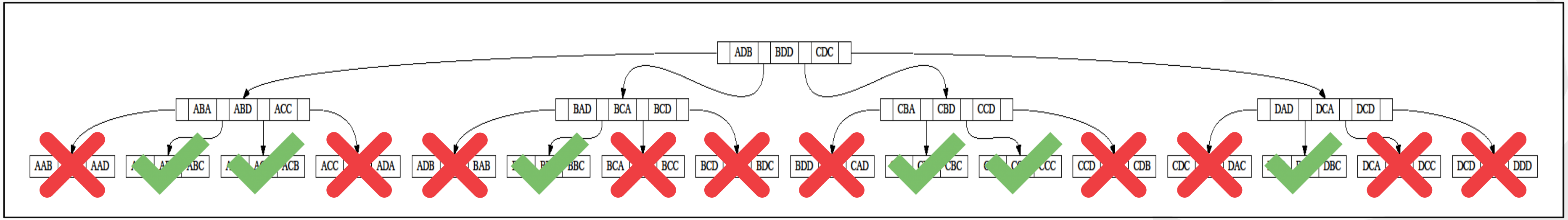


SELECTION CONDITIONS

Example: Index on <col1,col2,col3>

→ Column Values: {A,B,C,D}

→ Supported: col2 = B



"Skip Scan"

MERGING B+TREES

Approach #1: Off-line

→ Block all operations until done merging.

Approach #2: Eager

→ Access both during merge; move batches eagerly.

Approach #3: Background

→ Copy + merge in background; apply missed updates.

Approach #4: Lazy

→ Designate one as <u>main</u> and other as <u>secondary</u>.

→ If leaf in main not yet updated, merge corresponding key range from secondary.

Online B-Tree Merging

Xiaowei Sun Rui Wang Betty Salzberg •
College of Computer and Information Science, Northeastern University
{xwsun, bradrui, salzberg}@ccs.neu.edu

Chendong Zou
IBM
zouc@us.ibm.com

ABSTRACT

Many scenarios involve merging of two B-tree indexes, both covering the same key range. Increasing demand for continuous availability and high performance requires that such merging be done online, with minimal interference to normal user transactions. In this paper we present an online B-tree merging method, in which the merging of leaf pages in two B-trees are piggybacked lazily with normal user transactions, thus making the merging I/O efficient and allowing user transactions to access only one index instead of both. The concurrency control mechanism is designed to interfere as little as possible with ongoing user transactions. Merging is made forward recoverable by following a conventional logging protocol, with a few extensions. Should a system failure occur, both indexes being merged can be recovered to a consistent state and no merging work is lost. Experiments and analysis show the I/O savings and the performance, and compare variations on the basic algorithm.

1. INTRODUCTION

Many application and system maintenance scenarios require merging of two B-tree indexes covering *the same key range*. Two such scenarios occur during data migration in a parallel database system where data partitioning is not by key range, but by other methods such as hashing. First, for load balancing, data partitions may move from a "hot" node to a "cool" node. Second, when a node is to be deleted, the data on that node has to be distributed to other nodes. Accordingly, at the destination node, any *primary B-tree index* (with data in the leaf pages) and all *secondary B-tree indexes* (with references to data in the leaf pages) should be updated. For this purpose, temporary B-trees may need to be constructed on the moved data and merged later with the already existing B-tree indexes.

A third scenario occurs during batch data insertion in a centralized database system. For high efficiency, temporary B-tree indexes on newly inserted data may need to be constructed and merged into existing indexes. Yet another scenario occurs during the maintenance of a partitioned B-tree index [5]. In a partitioned B-tree, the whole tree is partitioned according to an artificial leading column.

•This work was partly supported by NSF grant IIS 0073063 and by a grant from Microsoft Corp.

SIGMOD 2005 June 14-16, 2005 Baltimore, Maryland, USA.

Each partition is a B-tree. During system maintenance, different partitions may need to be merged.

1.1 Straightforward Approaches

First let us look at some straightforward approaches to B-tree merging and describe some of their drawbacks.

Off-line Approach: In off-line B-tree merging, both indexes are X-locked until the merging is finished. All incoming user transactions will wait until the locks are released. Thus user transactions are forbidden access to the indexes while the indexes are being merged. This is the simplest approach but may incur an unacceptable amount of waiting time for user transactions.

Eager Approach: Another approach requires all user transactions to access *both* indexes while merging is going on and has separate non-user transactions move batches of entries *eagerly*. The system source code for operations such as exact-match searching, range searching, insertion, deletion and updating on B-trees must be extensively changed to adjust to the requirement of accessing both indexes. This complicates user transaction logic (or B-tree access module logic if the B-tree access module accesses both indexes on behalf of user transactions).

Background Approach: A third approach makes a copy for each index, merges them in the background, then catches up changes by applying log records from the indexes in use, finally switches the merged index online. This approach involves higher complexity, and suffers from extra space requirements and I/O cost for copying (thus low efficiency), especially when the indexes are very large. In addition, as in the eager approach, user transactions must access both indexes while merging is taking place.

For continuous availability, the merging of two B-tree indexes should be done *online*, allowing concurrent user transactions, including searches and modifications. For stable system performance, the merging process should be *efficient*, without slowing down the concurrent user transactions too much. In addition, for *practicability*, the merging algorithm should be relatively easy to integrate into realistic DBMS implementations. None of the straightforward approaches just discussed satisfies all these requirements. We have sought to improve on the straightforward approaches.

1.2 Lazy Approach

This paper presents an online B-tree merging approach where merging is piggybacked *lazily* on user transactions. We designate one B-tree index as the **main-index** and the other (usually the smaller one) as the **second-index**. User transactions operate through the

331

OBSERVATION

We (mostly) assumed all the data structures that we have discussed so far are single-threaded.

But we need to allow multiple threads to safely access our data structures to take advantage of additional CPU cores and hide disk I/O stalls.

OBSERVATION

We (mostly) assumed all the data structures that we have discussed so far are single-threaded.

But we need to allow multiple threads to safely access our data structures to take advantage of additional CPU cores and hide disk I/O stalls.

They Don't Do This!

VOLTDB

redis

H-Store

CONCURRENCY CONTROL

A **concurrency control** protocol is the method that the DBMS uses to ensure "correct" results for concurrent operations on a shared object.

A protocol's correctness criteria can vary:

→ **Logical Correctness:** Can a thread see the data that it is supposed to see?

→ **Physical Correctness:** Is the internal representation of the object sound?

CONCURRENCY CONTROL

A **concurrency control** protocol is the method that the DBMS uses to ensure “correct” results for concurrent operations on a shared object.

A protocol's correctness criteria can vary:

→ **Logical Correctness:** Can a thread see the data that it is supposed to see?

→ **Physical Correctness:** Is the internal representation of the object sound?

TODAY'S AGENDA

Latches Overview

Hash Table Latching

B+Tree Latching

Leaf Node Scans

LOCKS VS. LATCHES

Locks

→ Protect the database's logical contents from other txns.
→ Held for txn duration.
→ Need to be able to rollback changes.

Latches

→ Protect the critical sections of the DBMS's internal data structure from other threads.
→ Held for operation duration.
→ Do not need to be able to rollback changes.

LOCKS VS. LATCHES

	Locks	*Latches*
Separate…	User Transactions	Threads
Protect…	Database Contents	In-Memory Data Structures
During…	Entire Transactions	Critical Sections
Modes…	Shared, Exclusive, Update, Intention	Read, Write
Deadlock	Detection & Resolution	Avoidance
…by…	Waits-for, Timeout, Aborts	Coding Discipline
Kept in…	Lock Manager	Protected Data Structure

Source: Goetz Graefe

LOCKS VS. LATCHES

	Locks	*Latches*
Separate…	User Transactions	Threads
Protect…	Database Contents	In-Memory Data Structures
During…	Entire Transactions	Critical Sections
Modes…	Shared, Exclusive, Update, Intention	Read, Write
Deadlock	Detection & Resolution	Avoidance
…by…	Waits-for, Timeout, Aborts	Coding Discipline
Kept in…	Lock Manager	Protected Data Structure

Source: Goetz Graefe

LOCKS VS. LATCHES

Lecture 16

	Locks	*Latches*
Separate…	User Transactions	Threads
Protect…	Database Contents	In-Memory Data Structures
During…	Entire Transactions	Critical Sections
Modes…	Shared, Exclusive, Update, Intention	Read, Write
Deadlock	Detection & Resolution	Avoidance
…by…	Waits-for, Timeout, Aborts	Coding Discipline
Kept in…	Lock Manager	Protected Data Structure

Source: Goetz Graefe

LATCH MODES

Read Mode

→ Multiple threads can read the same object at the same time.
→ A thread can acquire the read latch if another thread has it in read mode.

Write Mode

→ Only one thread can access the object.
→ A thread cannot acquire a write latch if another thread has it in any mode.

LATCH MODES

Read Mode

→ Multiple threads can read the same object at the same time.
→ A thread can acquire the read latch if another thread has it in read mode.

Write Mode

→ Only one thread can access the object.
→ A thread cannot acquire a write latch if another thread has it in any mode.

Compatibility Matrix

	Read	Write
Read	✔	X
Write	X	X

LATCH IMPLEMENTATIONS

Blocking OS Mutex

Test-and-Set Spin Latch

Reader-Writer Latches

LATCH IMPLEMENTATIONS

Approach #1: Blocking OS Mutex

→ Simple to use

→ Non-scalable (about 25ns per lock/unlock invocation)

→ Example: `std::mutex`

LATCH IMPLEMENTATIONS

Approach #1: Blocking OS Mutex

→ Simple to use
→ Non-scalable (about 25ns per lock/unlock invocation)
→ Example: **std::mutex**

```
std::mutex m;
 ⋮
m.lock();
// Do something special...
m.unlock();
```

LATCH IMPLEMENTATIONS

Approach #1: Blocking OS Mutex

→ Simple to use
→ Non-scalable (about 25ns per lock/unlock invocation)
→ Example: **std::mutex**

```
std::mutex m;→pthread_mutex_t
 ⋮
m.lock();
// Do something special...
m.unlock();
```

LATCH IMPLEMENTATIONS

Approach #1: Blocking OS Mutex

→ Simple to use
→ Non-scalable (about 25ns per lock/unlock invocation)
→ Example: **std::mutex**

```
std::mutex m; → pthread_mutex_t
 ⋮                      ↓
                      futex
m.lock();
// Do something special...
m.unlock();
```

LATCH IMPLEMENTATIONS

Approach #1: Blocking OS Mutex

→ Simple to use
→ Non-scalable (about 25ns per lock/unlock invocation)
→ Example: **std::mutex**

OS Latch
Userspace Latch

```
std::mutex m; → pthread_mutex_t
 ⋮                    ↓
                    futex
m.lock();
// Do something special...
m.unlock();
```

LATCH IMPLEMENTATIONS

Approach #1: Blocking OS Mutex

→ Simple to use

→ Non-scalable (about 25ns per lock/unlock invocation)

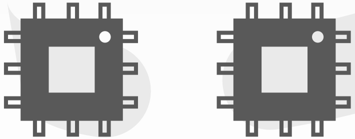
→ Example: **std::mutex**

```
std::mutex m;
  ⋮
m.lock();
// Do something special...
m.unlock();
```

std::mutex m; → pthread_mutex_t → futex

OS Latch

Userspace Latch



LATCH IMPLEMENTATIONS

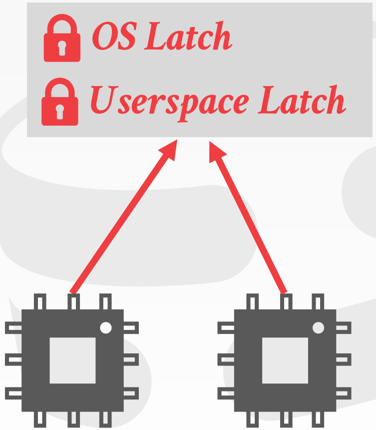
Approach #1: Blocking OS Mutex

→ Simple to use

→ Non-scalable (about 25ns per lock/unlock invocation)

→ Example: **std::mutex**

```
std::mutex m; → pthread_mutex_t
 ⋮                    ↓
                    futex
m.lock();
// Do something special...
m.unlock();
```

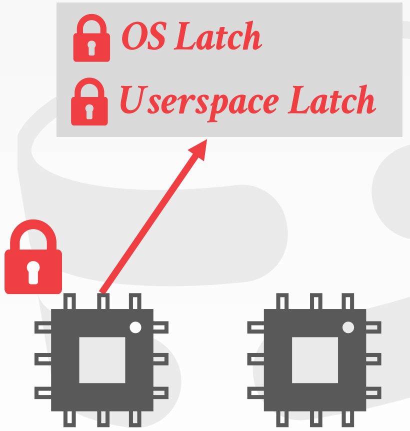


LATCH IMPLEMENTATIONS

Approach #1: Blocking OS Mutex

→ Simple to use
→ Non-scalable (about 25ns per lock/unlock invocation)
→ Example: **std::mutex**

```
std::mutex m; → pthread_mutex_t
  ⋮                    ↓
                     futex
m.lock();
// Do something special...
m.unlock();
```

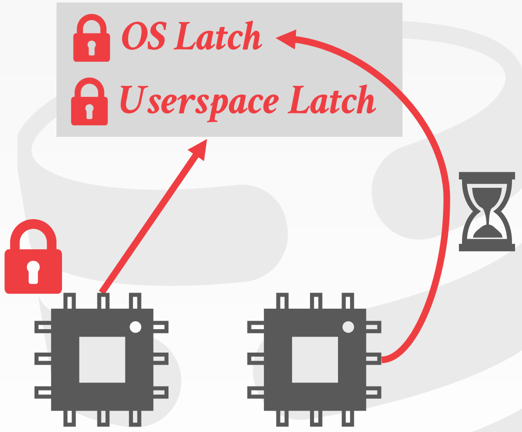


LATCH IMPLEMENTATIONS

Approach #1: Blocking OS Mutex

→ Simple to use
→ Non-scalable (about 25ns per lock/unlock invocation)
→ Example: **std::mutex**

```
std::mutex m; → pthread_mutex_t
                       ↓
  :                  futex
m.lock();
// Do something special...
m.unlock();
```



LATCH IMPLEMENTATIONS

Approach #2: Test-and-Set Spin Latch (TAS)

→ Very efficient (single instruction to latch/unlatch)

→ Non-scalable, not cache-friendly, not OS-friendly

→ Example: **`std::atomic<T>`**

LATCH IMPLEMENTATIONS

Approach #2: Test-and-Set Spin Latch (TAS)

→ Very efficient (single instruction to latch/unlatch)
→ Non-scalable, not cache-friendly, not OS-friendly
→ Example: **`std::atomic<T>`**

```
std::atomic_flag latch;
 ⋮
while (latch.test_and_set(…)) {
    // Retry? Yield? Abort?
}
```

LATCH IMPLEMENTATIONS

Approach #2: Test-and-Set Spin Latch (TAS)

→ Very efficient (single instruction to latch/unlatch)
→ Non-scalable, not cache-friendly, not OS-friendly
→ Example: **std::atomic<T>**

std::atomic<bool>

```
std::atomic_flag latch;
 ⋮
while (latch.test_and_set(…)) {
   // Retry? Yield? Abort?
}
```

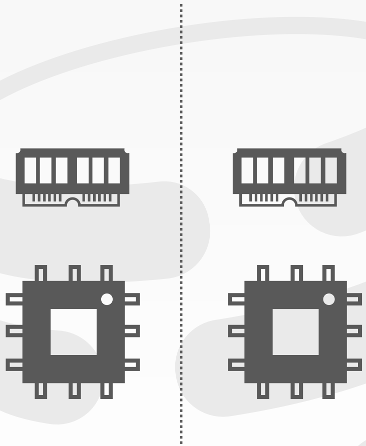
LATCH IMPLEMENTATIONS

Approach #2: Test-and-Set Spin Latch (TAS)

→ Very efficient (single instruction to latch/unlatch)
→ Non-scalable, not cache-friendly, not OS-friendly
→ Example: **std::atomic<T>**

std::atomic<bool>

```
std::atomic_flag latch;
 ⋮
while (latch.test_and_set(…)) {
    // Retry? Yield? Abort?
}
```



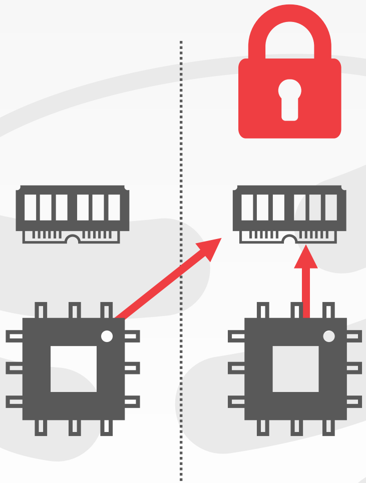
LATCH IMPLEMENTATIONS

Approach #2: Test-and-Set Spin Latch (TAS)

→ Very efficient (single instruction to latch/unlatch)
→ Non-scalable, not cache-friendly, not OS-friendly
→ Example: **std::atomic<T>**

std::atomic<bool>

```
std::atomic_flag latch;
 ⋮
while (latch.test_and_set(…)) {
    // Retry? Yield? Abort?
}
```



LATC

Approach

→ Very effic

→ Non-scal

→ Example:

std::atomic<bool>

```
std::ato
  :
while (l
    // Re
}
```

By: **Linus Torvalds** (torvalds.delete@this.linux-foundation.org), January 3, 2020 6:05 pm

Room: Moderated Discussions

Beastian (no.email.delete@this.aol.com) on January 3, 2020 11:46 am wrote:
> I'm usually on the other side of these primitives when I write code as a consumer of them,
> but it's very interesting to read about the nuances related to their implementations:

The whole post seems to be just wrong, and is measuring something completely different than what the author thinks and claims it is measuring.

First off, spinlocks can only be used if you actually know you're not being scheduled while using them. But the blog post author seems to be implementing his own spinlocks in user space with no regard for whether the lock user might be scheduled or not. And the code used for the claimed "lock not held" timing is complete garbage.

It basically reads the time before releasing the lock, and then it reads it after acquiring the lock again, and claims that the time difference is the time when no lock was held. Which is just inane and pointless and completely wrong.

That's pure garbage. What happens is that

(a) since you're spinning, you're using CPU time

(b) at a random time, the scheduler will schedule you out

(c) that random time might ne just after you read the "current time", but before you actually released the spinlock.

So now you still hold the lock, but you got scheduled away from the CPU, because you had used up your time slice. The "current time" you read is basically now stale, and has nothing to do with the (future) time when you are *actually* going to release the lock.

Somebody else comes in and wants that "spinlock", and that somebody will now spin for a long while, since nobody is releasing it - it's still held by that other thread entirely that was just scheduled out. At some point, the scheduler says "ok, now you've used your time slice", and schedules the original thread, and *now* the lock is actually released. Then another thread comes in, gets the lock again, and then it looks at the time and says "oh, a long time passed without the lock being held at all".

And notice how the above is the *good* scenario. If you have more threads than CPU's (maybe because of other processes unrelated to your own test load), maybe the next thread that gets shceduled isn't the one that is going to release the lock. No, that one already got its timeslice, so the next thread scheduled might be *another* thread that wants that lock that is still being held by the thread that isn't even running right now!

So the code in question is pure garbage. You can't do spinlocks like that. Or rather, you very much can do them like that, and when you do that you are measuring random latencies and getting nonsensical values, because what you are measuring is "I have a lot of busywork, where all the processes are CPU-bound, and I'm measuring random points of how long the scheduler kept the process in place".

And then you write a blog-post blamings others, not understanding that it's your incorrect code that is garbage, and is giving random garbage values.

LATC

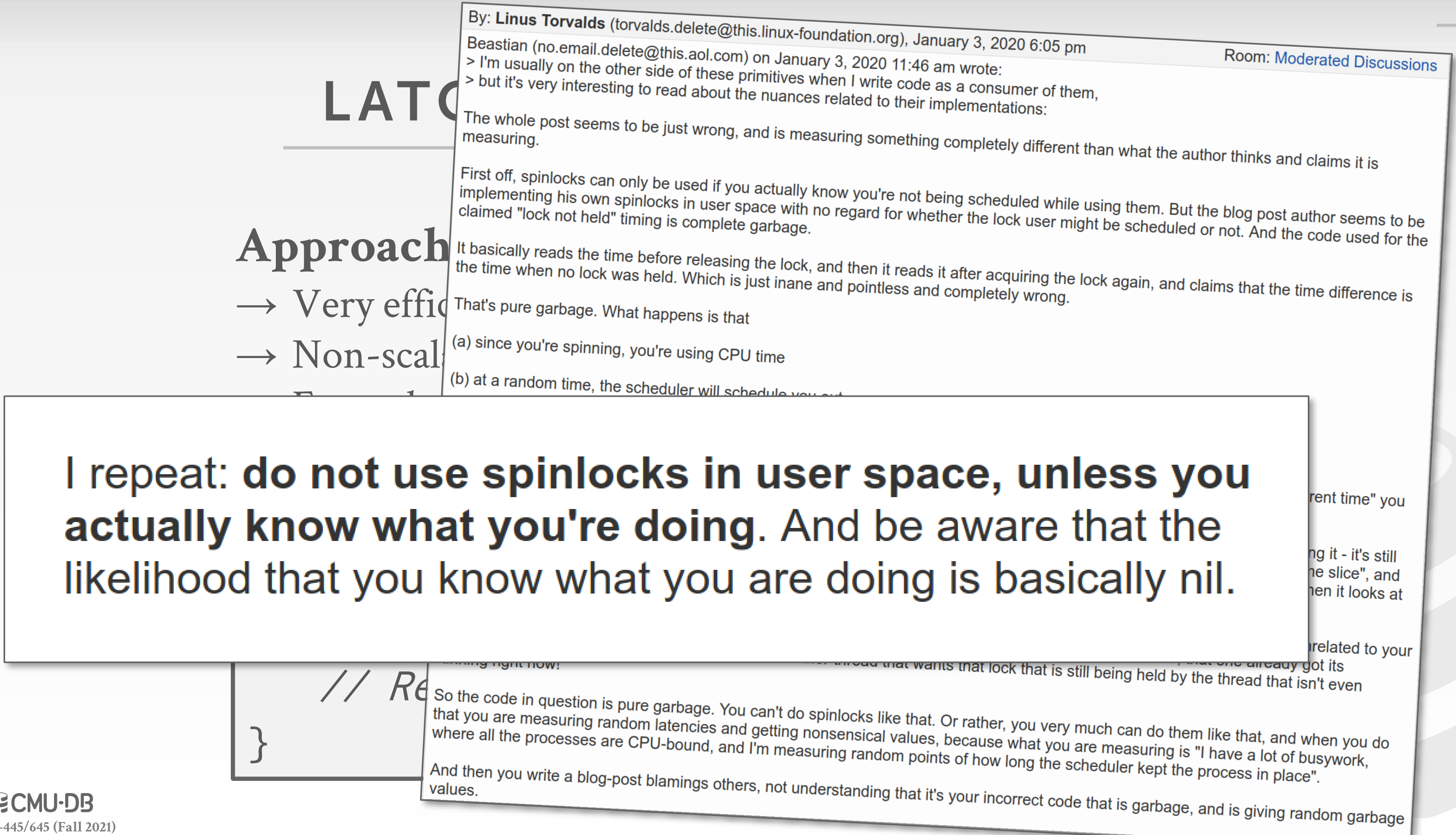
Approach

→ Very effic

→ Non-scal

// Re

}



By: **Linus Torvalds** (torvalds.delete@this.linux-foundation.org), January 3, 2020 6:05 pm

Room: Moderated Discussions

Beastian (no.email.delete@this.aol.com) on January 3, 2020 11:46 am wrote:

> I'm usually on the other side of these primitives when I write code as a consumer of them,
> but it's very interesting to read about the nuances related to their implementations:

The whole post seems to be just wrong, and is measuring something completely different than what the author thinks and claims it is measuring.

First off, spinlocks can only be used if you actually know you're not being scheduled while using them. But the blog post author seems to be implementing his own spinlocks in user space with no regard for whether the lock user might be scheduled or not. And the code used for the claimed "lock not held" timing is complete garbage.

It basically reads the time before releasing the lock, and then it reads it after acquiring the lock again, and claims that the time difference is the time when no lock was held. Which is just inane and pointless and completely wrong.

That's pure garbage. What happens is that

(a) since you're spinning, you're using CPU time

(b) at a random time, the scheduler will schedule you out

...rent time" you

...ng it - it's still ...ne slice", and ...hen it looks at

...related to your ... that one already got its ... thread that wants that lock that is still being held by the thread that isn't even running right now!

So the code in question is pure garbage. You can't do spinlocks like that. Or rather, you very much can do them like that, and when you do that you are measuring random latencies and getting nonsensical values, because what you are measuring is "I have a lot of busywork, where all the processes are CPU-bound, and I'm measuring random points of how long the scheduler kept the process in place".

And then you write a blog-post blamings others, not understanding that it's your incorrect code that is garbage, and is giving random garbage values.

I repeat: **do not use spinlocks in user space, unless you actually know what you're doing**. And be aware that the likelihood that you know what you are doing is basically nil.

LATCH IMPLEMENTATIONS

Choice #3: Reader-Writer Latches

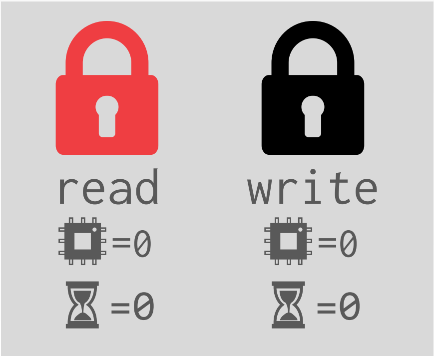
→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches

LATCH IMPLEMENTATIONS

Choice #3: Reader-Writer Latches

→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches

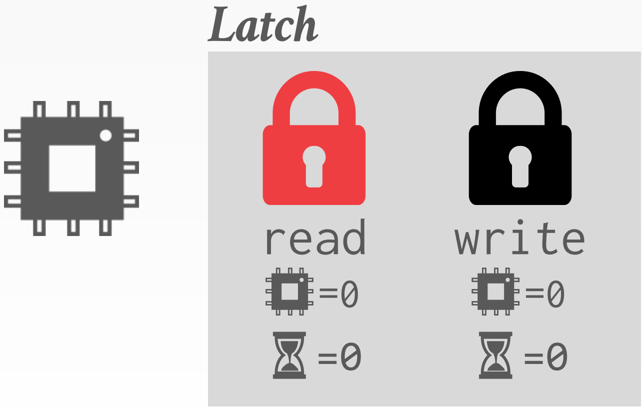
Latch



LATCH IMPLEMENTATIONS

Choice #3: Reader-Writer Latches

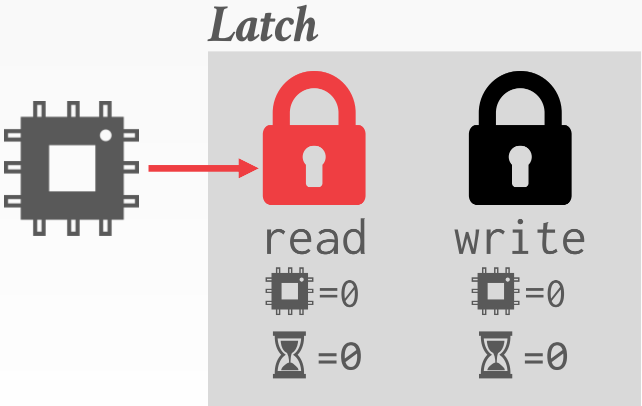
→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches



LATCH IMPLEMENTATIONS

Choice #3: Reader-Writer Latches

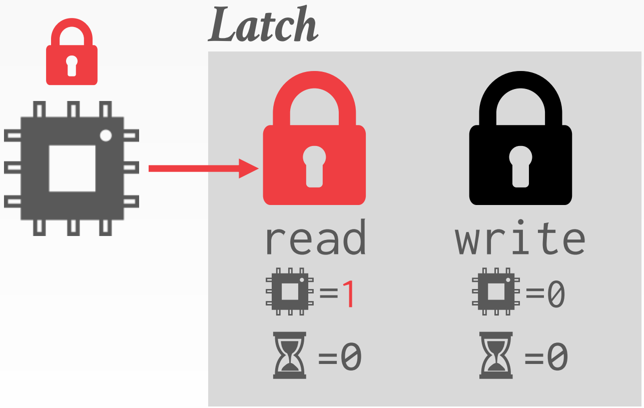
→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches



LATCH IMPLEMENTATIONS

Choice #3: Reader-Writer Latches

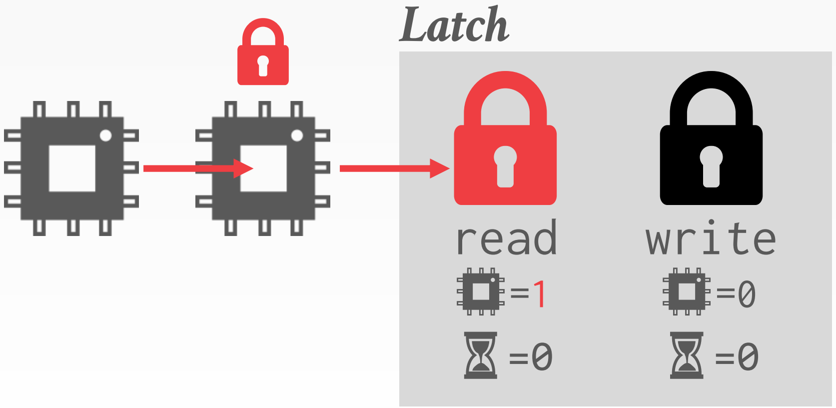
→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches



LATCH IMPLEMENTATIONS

Choice #3: Reader-Writer Latches

→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches



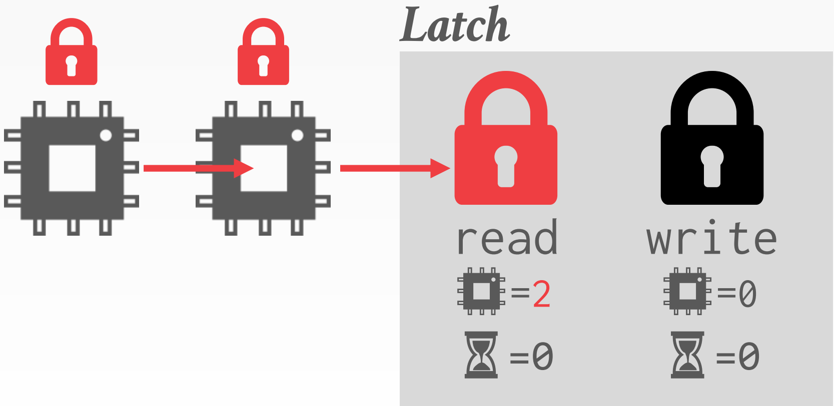
LATCH IMPLEMENTATIONS

Choice #3: Reader-Writer Latches

→ Allows for concurrent readers

→ Must manage read/write queues to avoid starvation

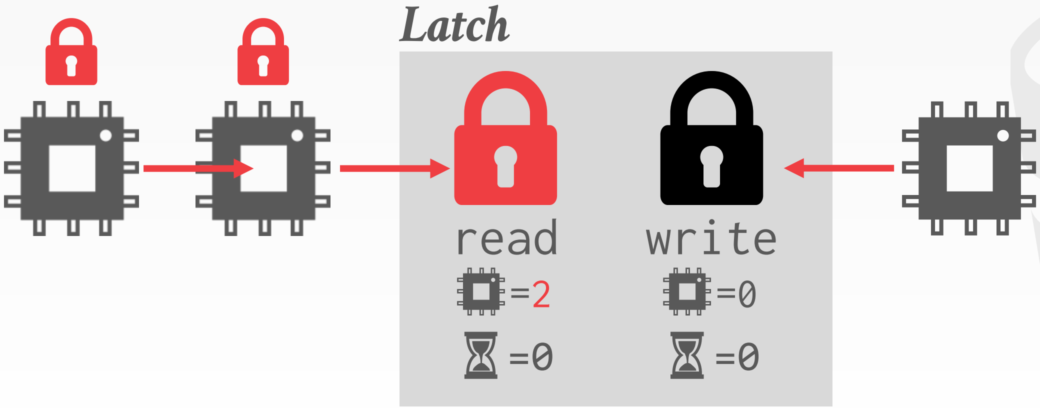
→ Can be implemented on top of spin latches



LATCH IMPLEMENTATIONS

Choice #3: Reader-Writer Latches

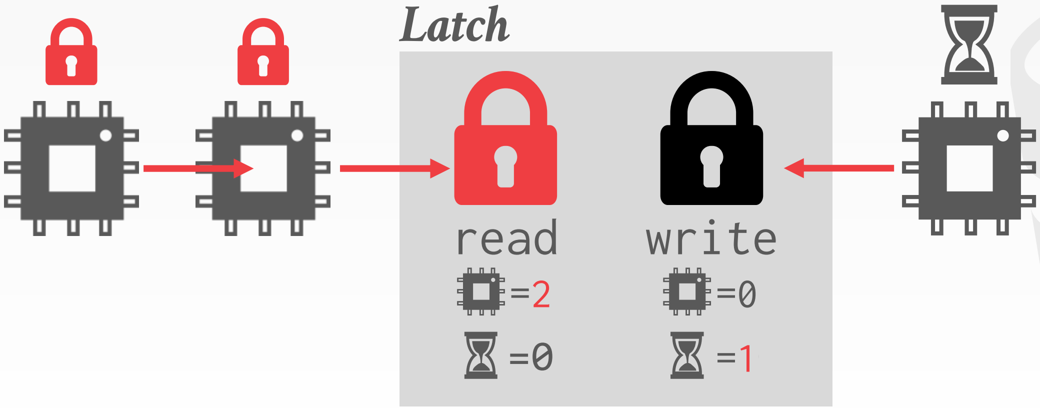
→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches



LATCH IMPLEMENTATIONS

Choice #3: Reader-Writer Latches

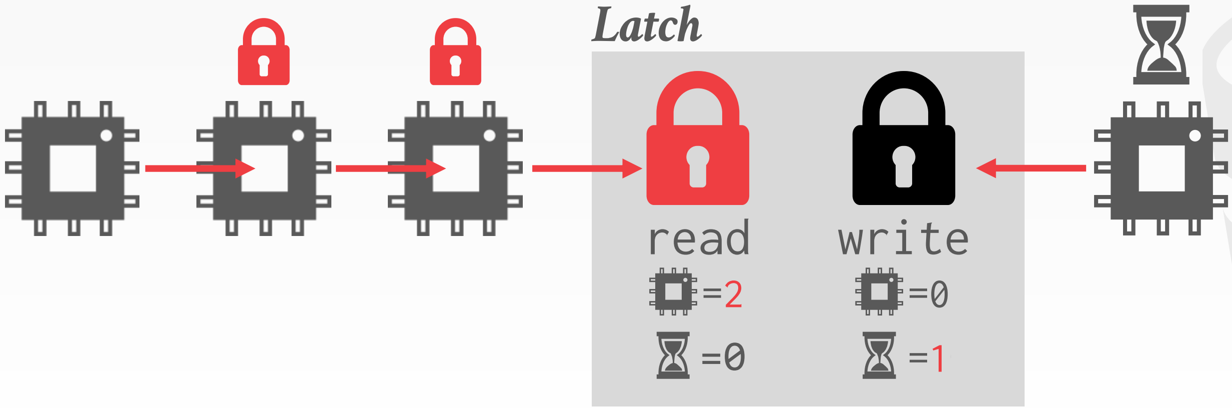
→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches



LATCH IMPLEMENTATIONS

Choice #3: Reader-Writer Latches

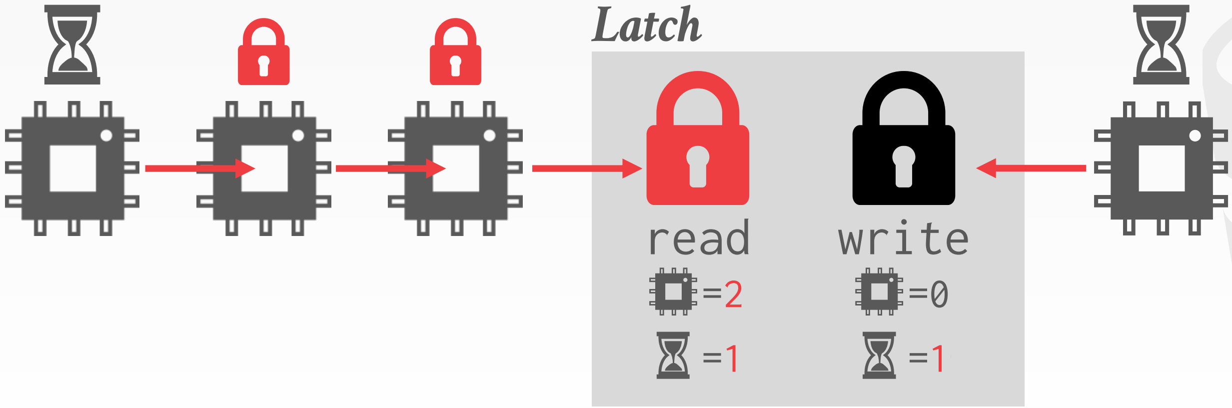
→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches



LATCH IMPLEMENTATIONS

Choice #3: Reader-Writer Latches

→ Allows for concurrent readers
→ Must manage read/write queues to avoid starvation
→ Can be implemented on top of spin latches



HASH TABLE LATCHING

Easy to support concurrent access due to the limited ways threads access the data structure.

→ All threads move in the same direction and only access a single page/slot at a time.
→ Deadlocks are not possible.

To resize the table, take a global write latch on the entire table (e.g., in the header page).

HASH TABLE LATCHING

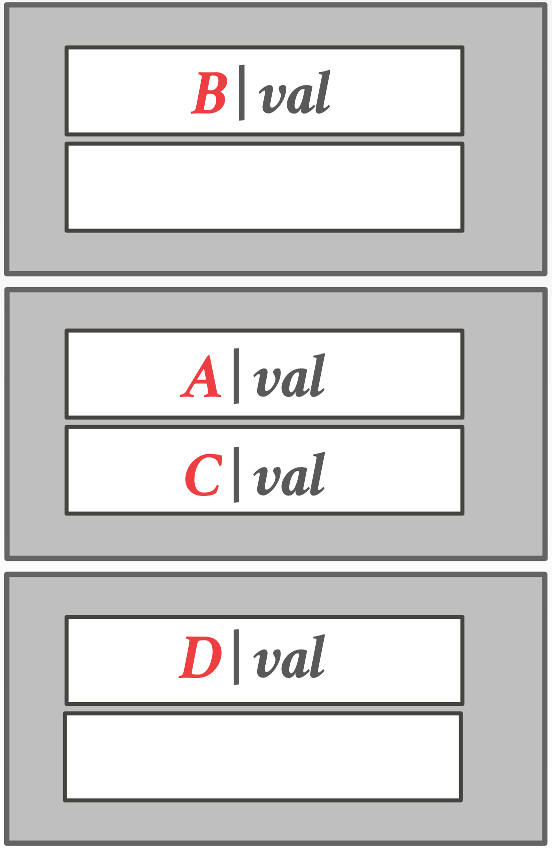
Approach #1: Page Latches

→ Each page has its own reader-writer latch that protects its entire contents.
→ Threads acquire either a read or write latch before they access a page.

Approach #2: Slot Latches

→ Each slot has its own latch.
→ Can use a single-mode latch to reduce meta-data and computational overhead.

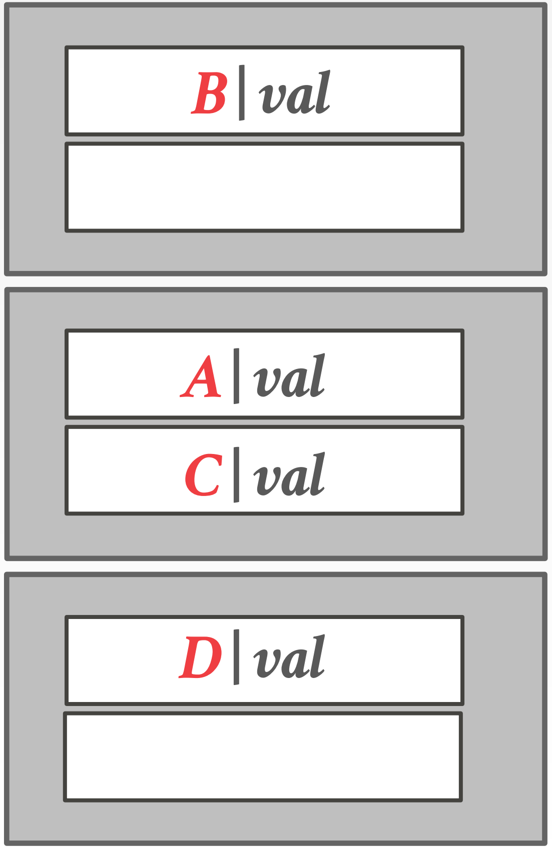
HASH TABLE – PAGE LATCHES



HASH TABLE – PAGE LATCHES

T_1: Find D

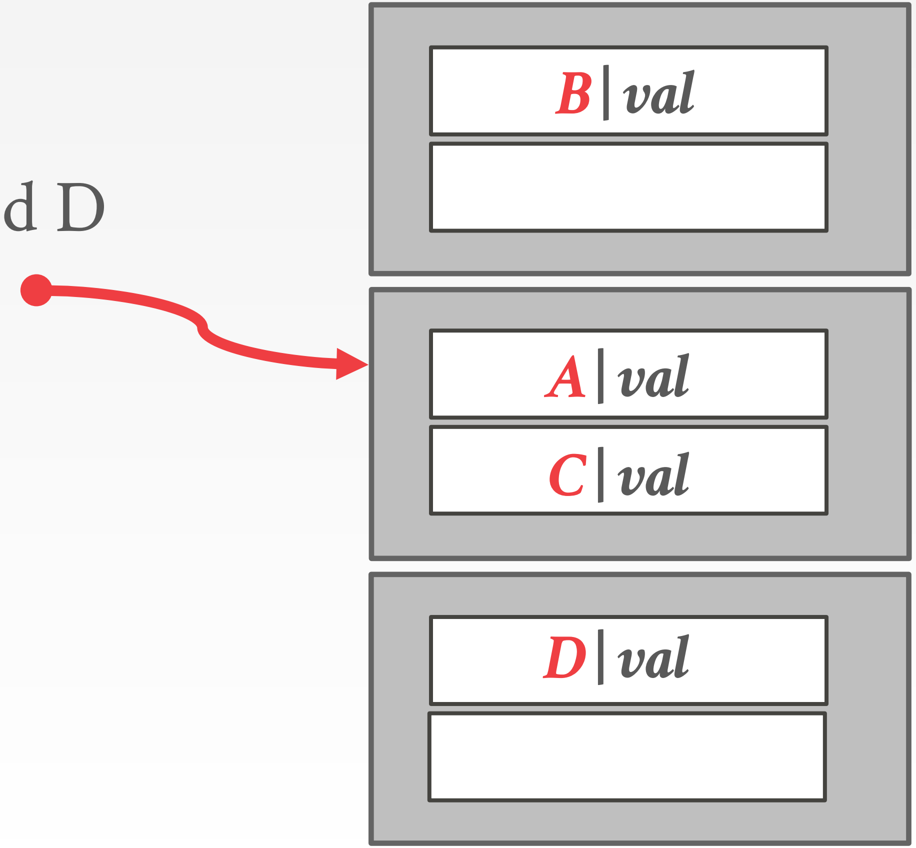
hash(D)



HASH TABLE – PAGE LATCHES

T_1: Find D

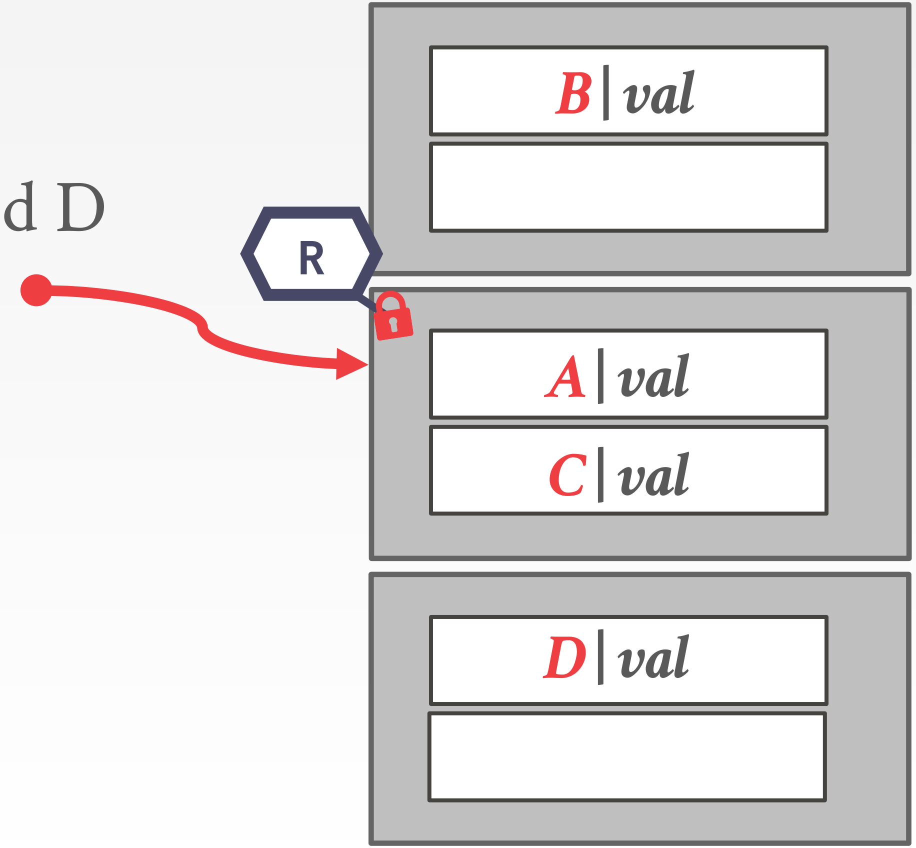
hash(D)



HASH TABLE – PAGE LATCHES

T_1: Find D

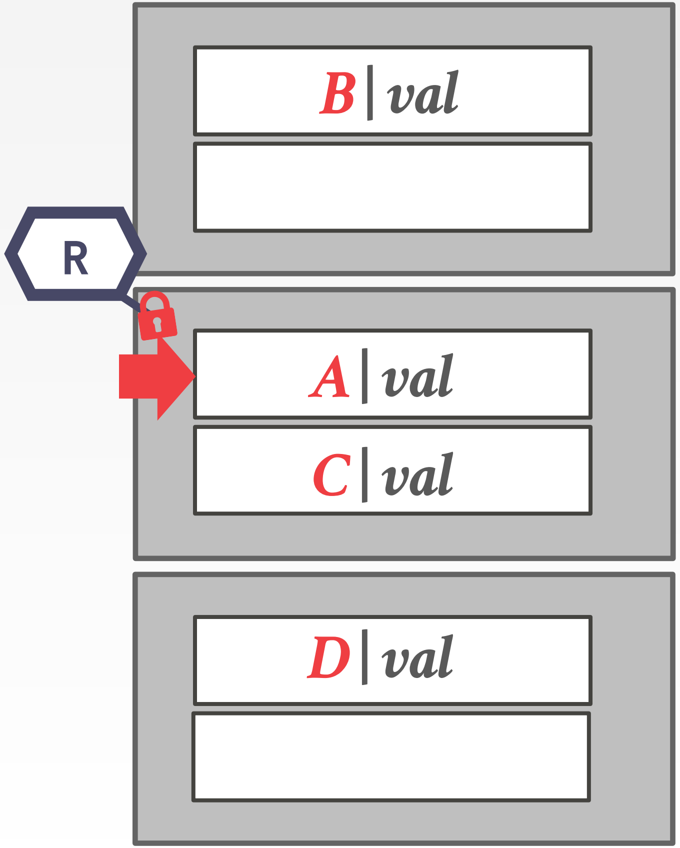
hash(D)



HASH TABLE – PAGE LATCHES

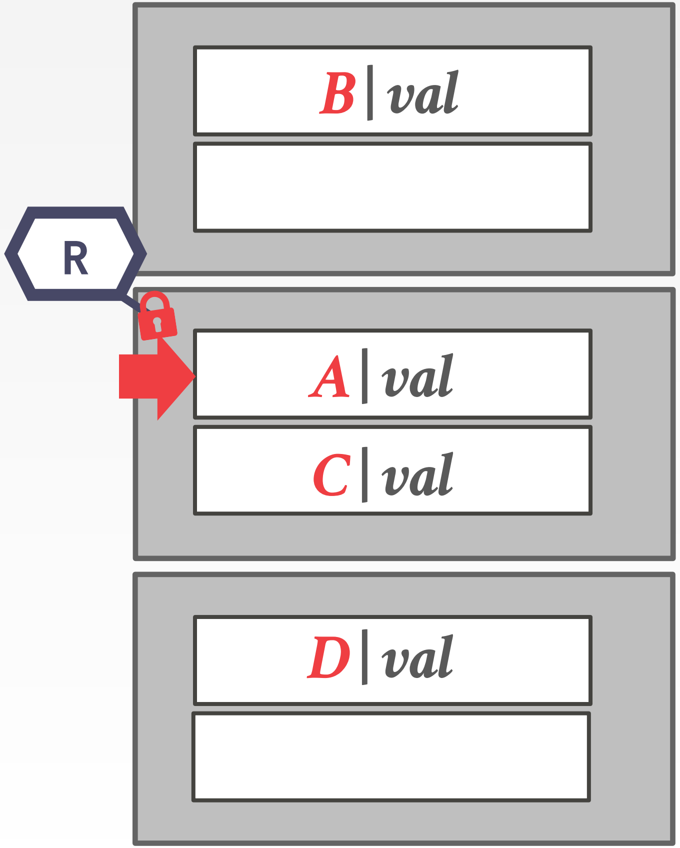
T_1: Find D

hash(D)



HASH TABLE – PAGE LATCHES

T_1: Find D
hash(D)

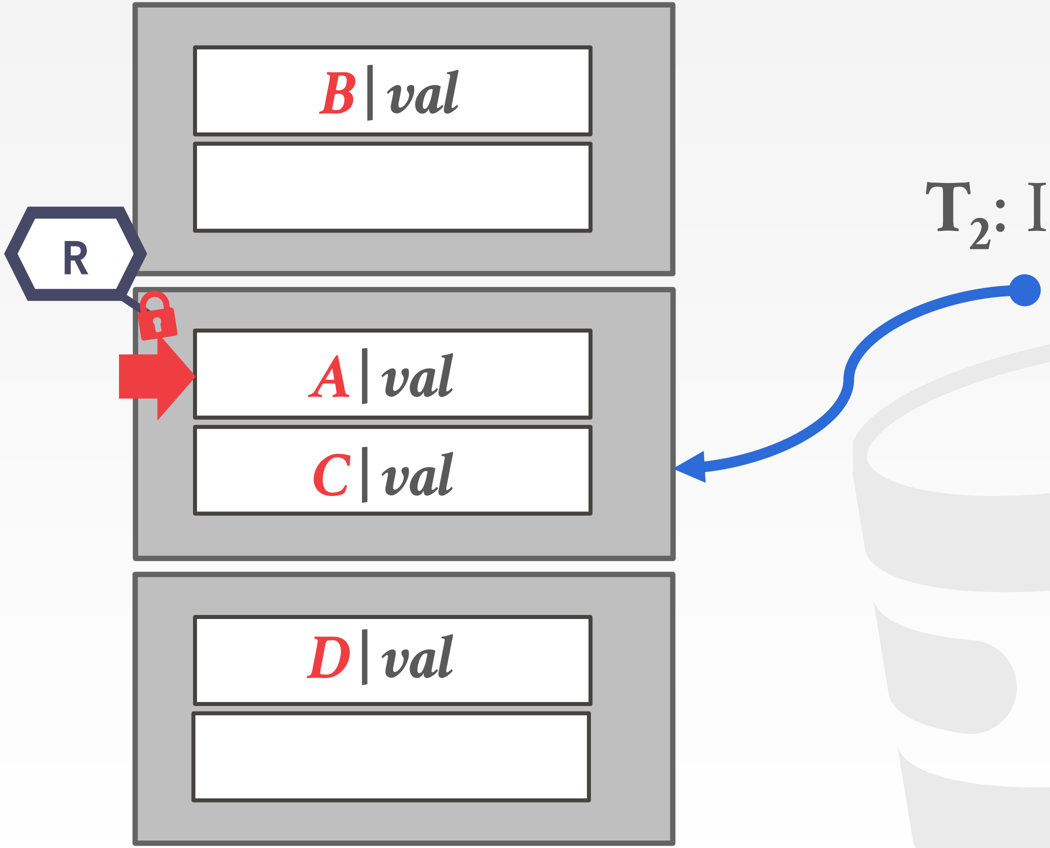


T_2: Insert E
hash(E)

HASH TABLE – PAGE LATCHES

T_1: Find D
hash(D)

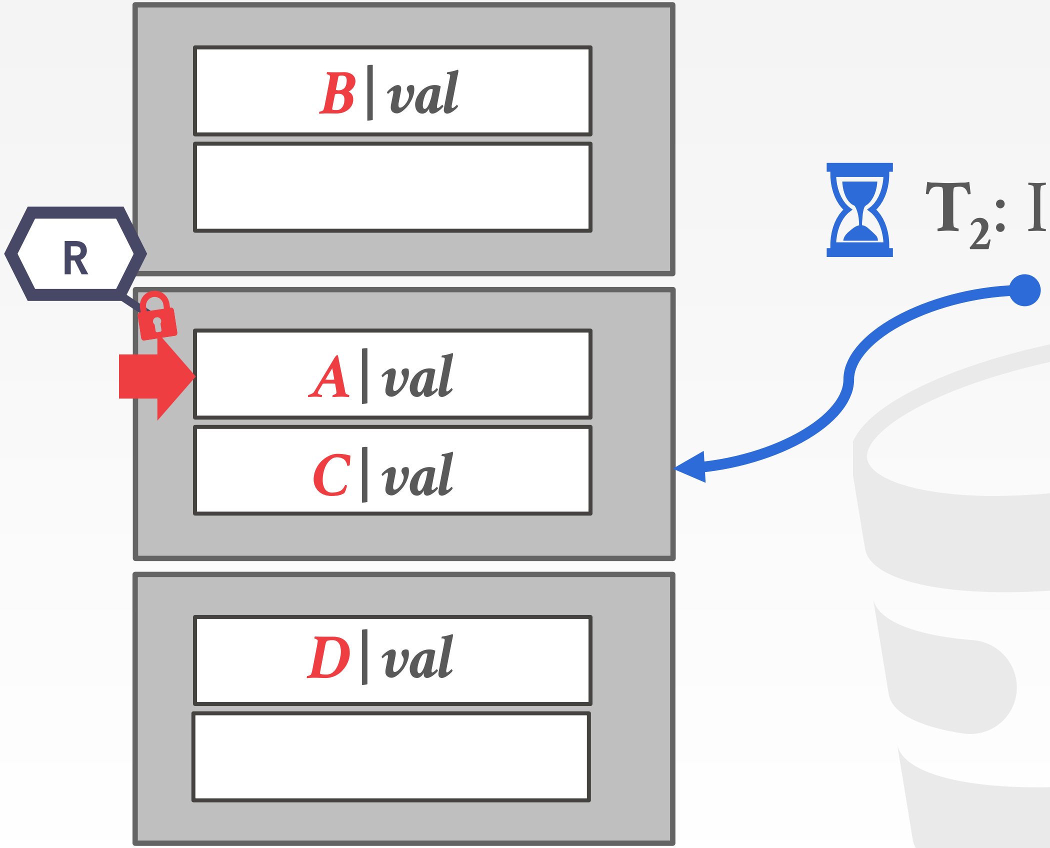
T_2: Insert E
hash(E)



HASH TABLE – PAGE LATCHES

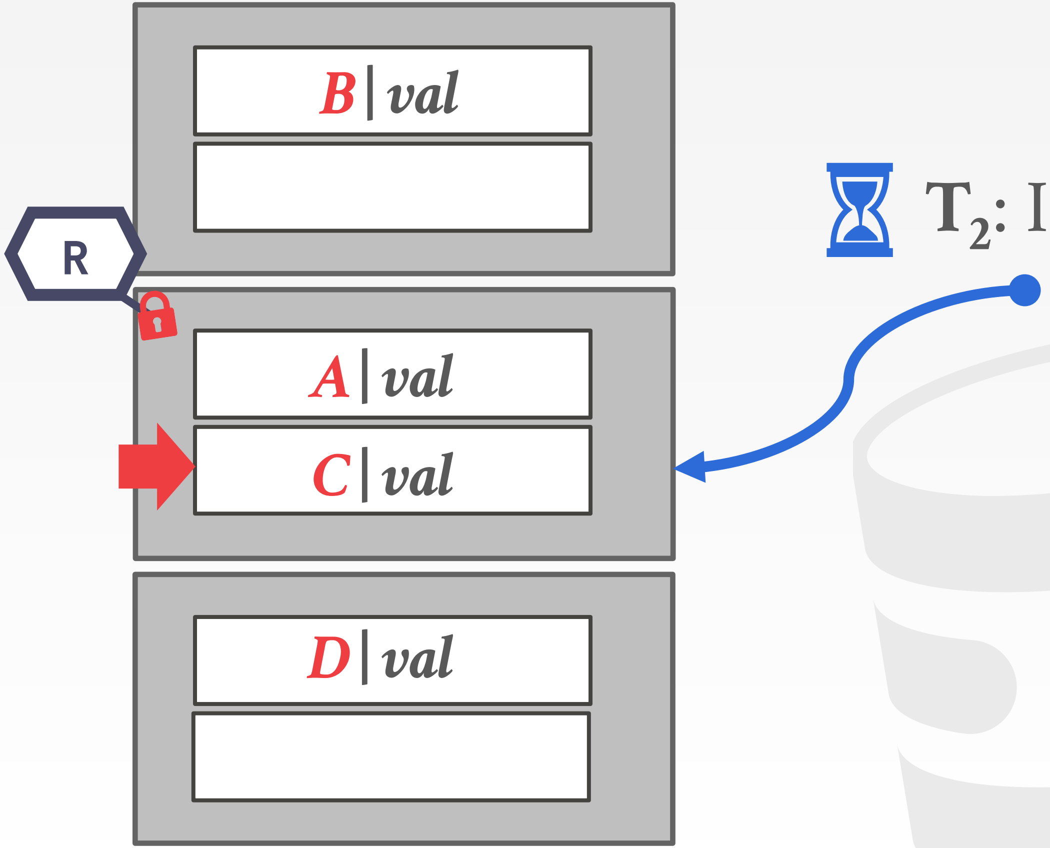
T_1: Find D
hash(D)

T_2: Insert E
hash(E)



HASH TABLE – PAGE LATCHES

T_1: Find D
hash(D)

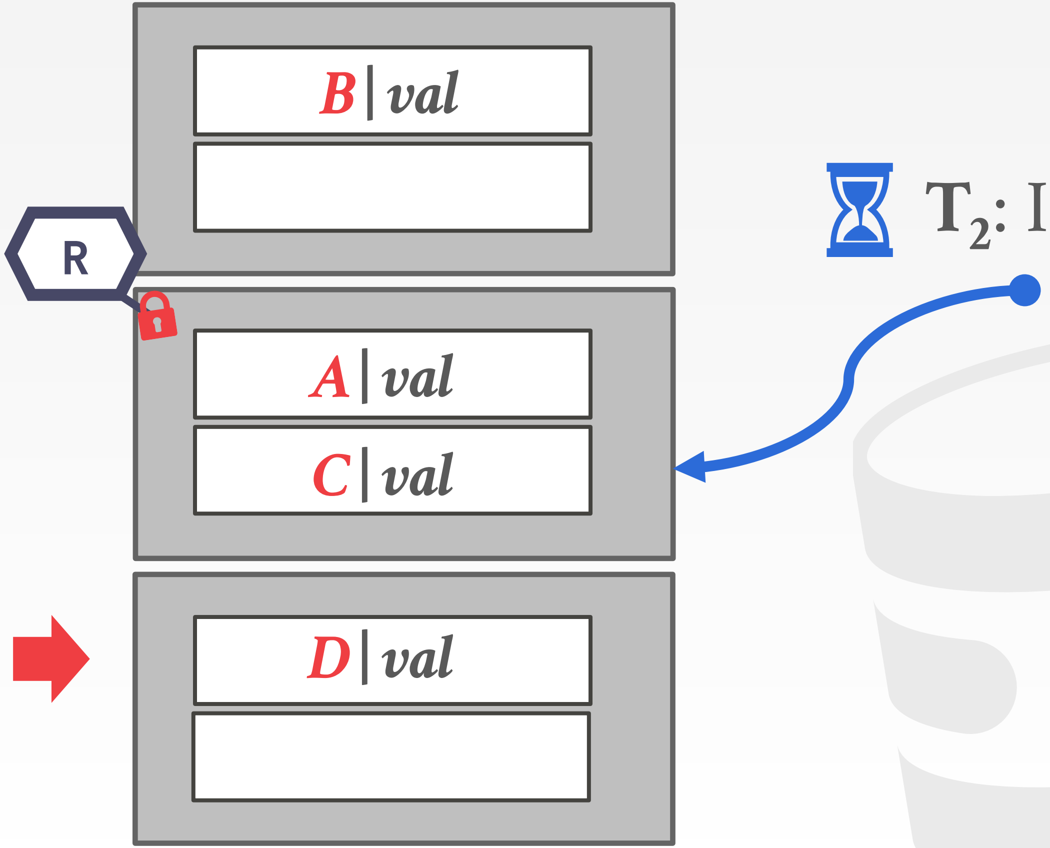


T_2: Insert E
hash(E)

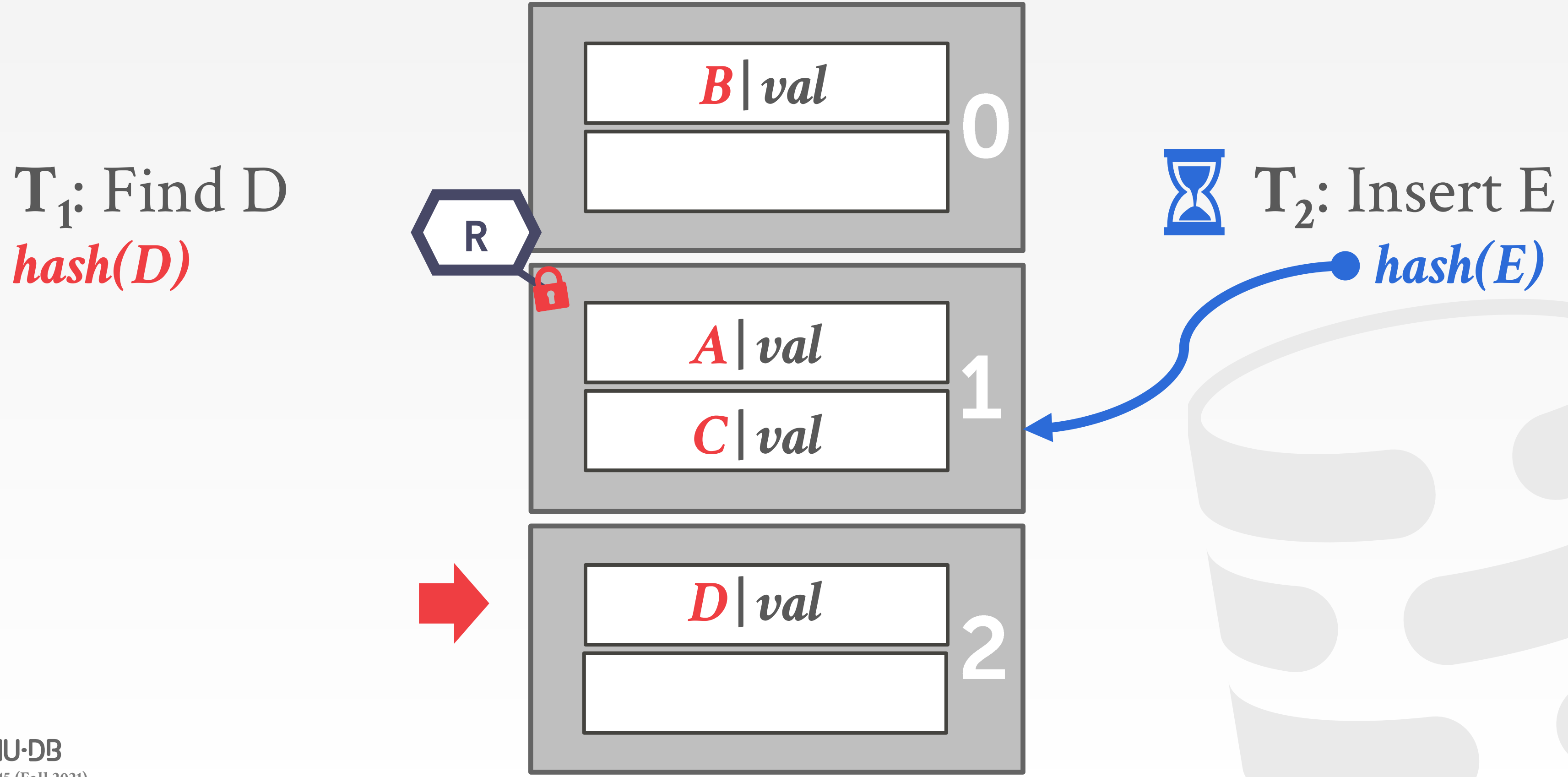
HASH TABLE – PAGE LATCHES

T$_1$: Find D
hash(D)

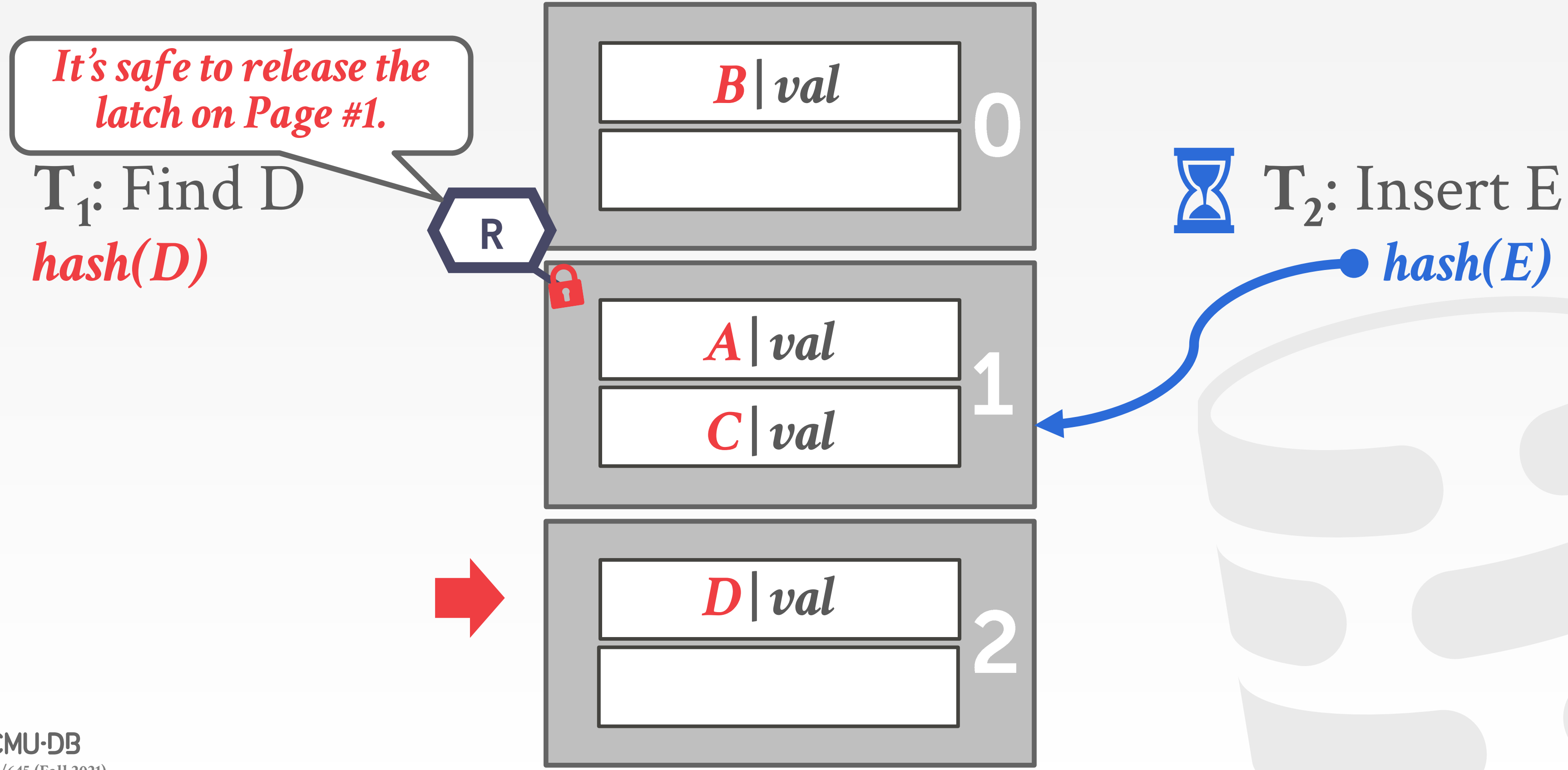
T$_2$: Insert E
hash(E)



HASH TABLE – PAGE LATCHES

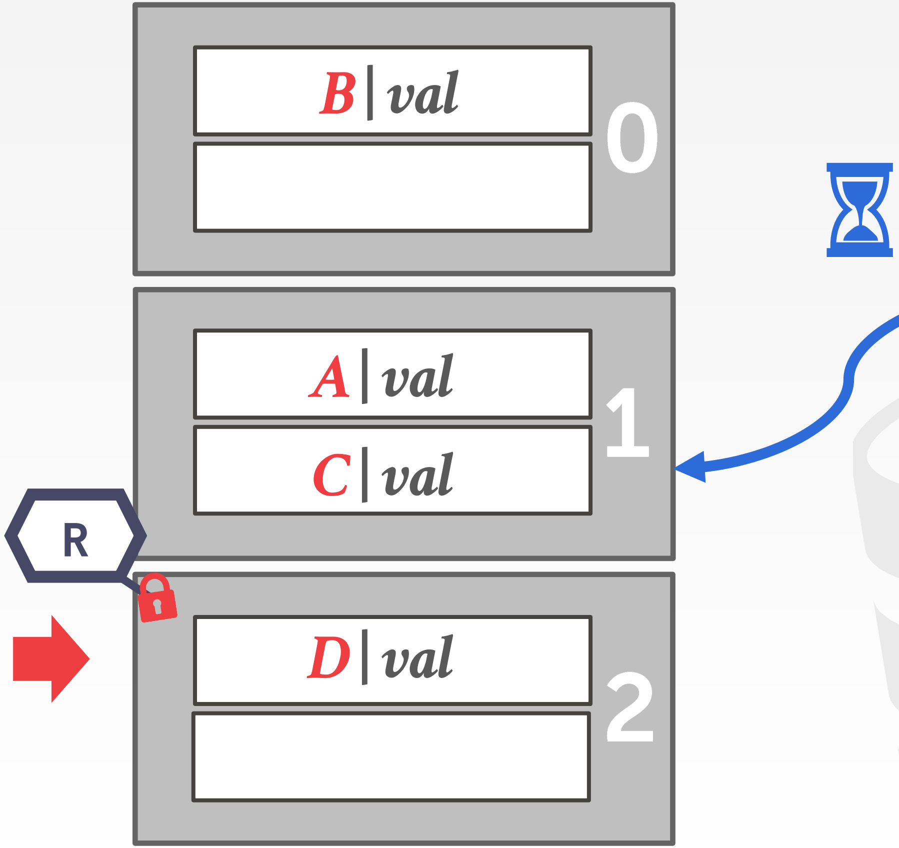


HASH TABLE – PAGE LATCHES



HASH TABLE – PAGE LATCHES

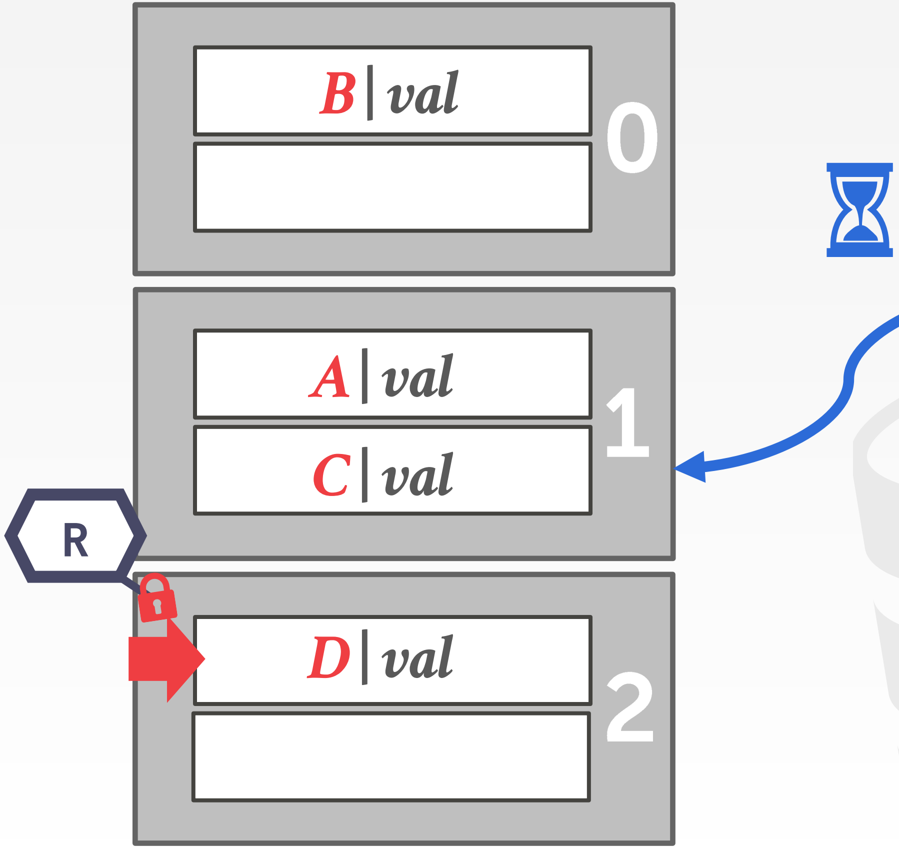
T$_1$: Find D
hash(D)



T$_2$: Insert E
hash(E)

HASH TABLE – PAGE LATCHES

T_1: Find D
hash(D)

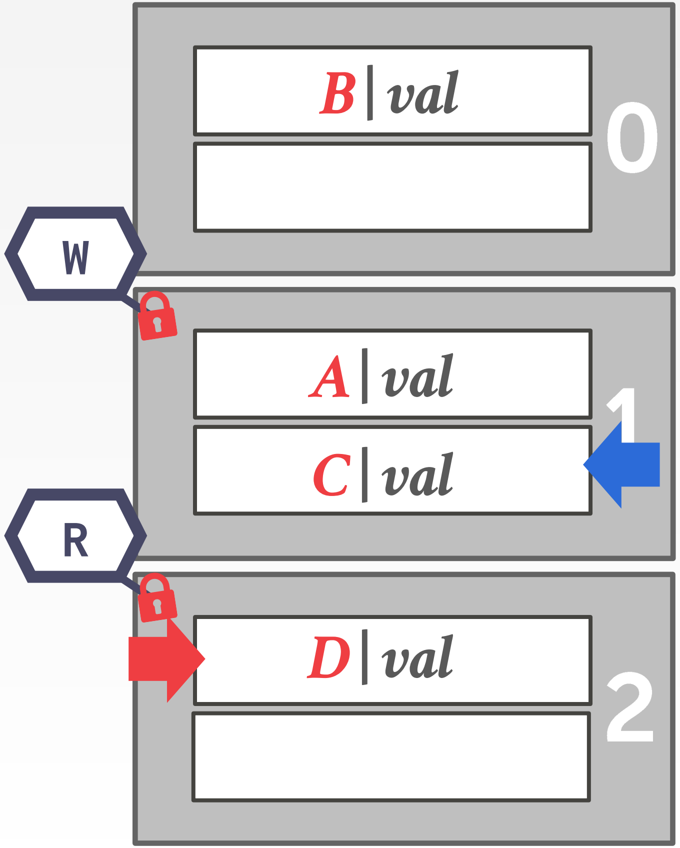


T_2: Insert E
hash(E)

HASH TABLE – PAGE LATCHES

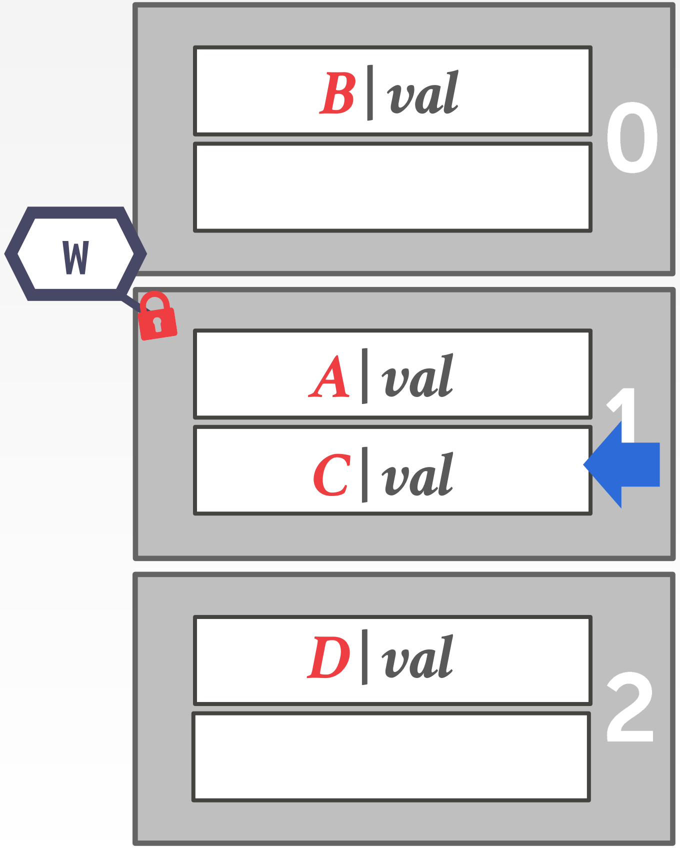
T_1: Find D
hash(D)

T_2: Insert E
hash(E)



HASH TABLE – PAGE LATCHES

T_1: Find D
hash(D)

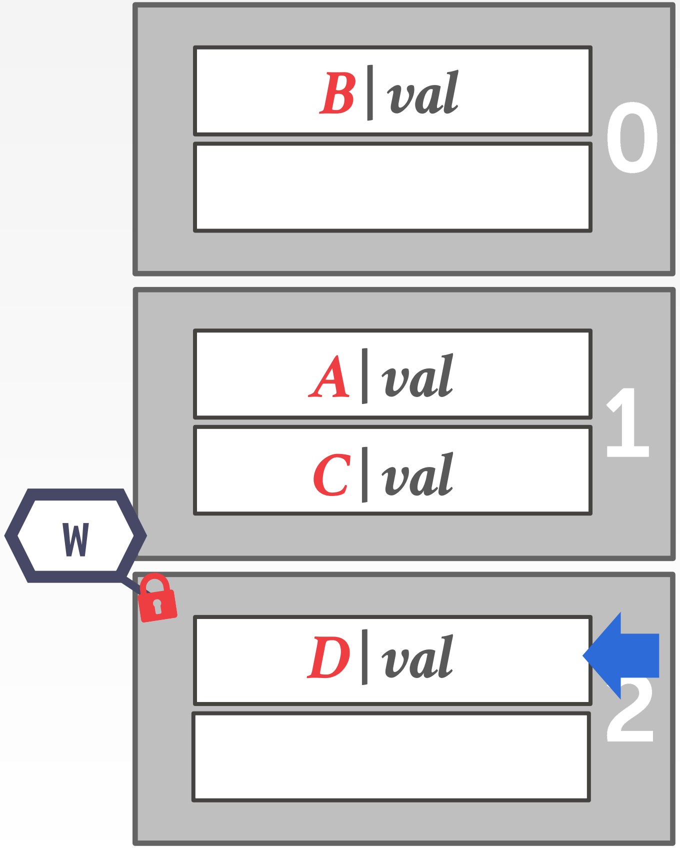


T_2: Insert E
hash(E)

HASH TABLE – PAGE LATCHES

T_1: Find D

hash(D)

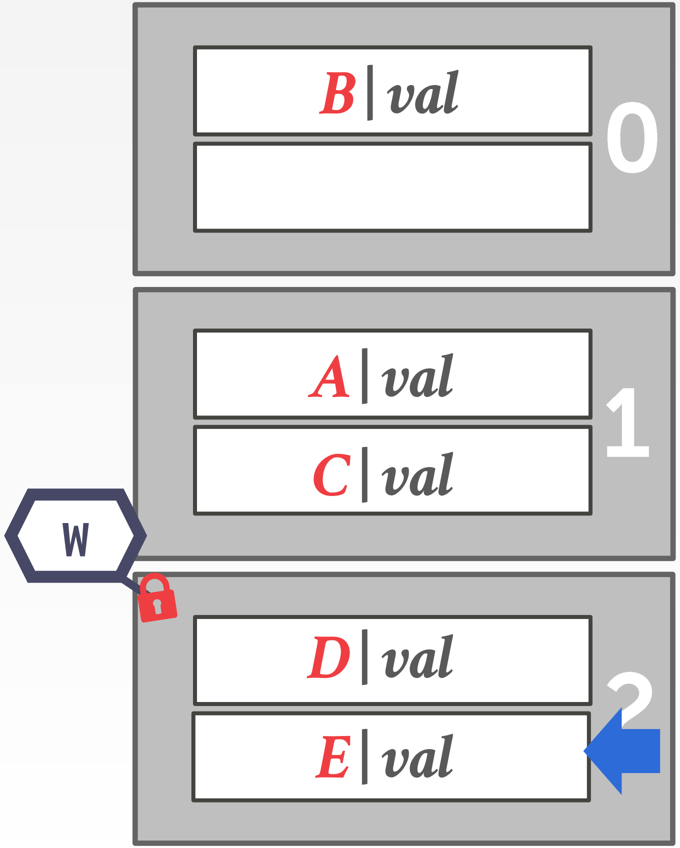


T_2: Insert E

hash(E)

HASH TABLE – PAGE LATCHES

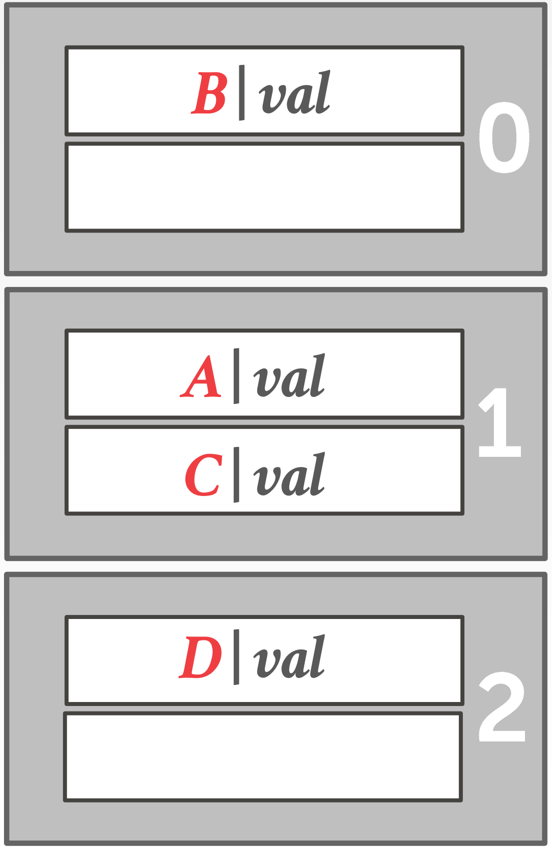
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

T_1: Find D
hash(D)



T_2: Insert E
hash(E)

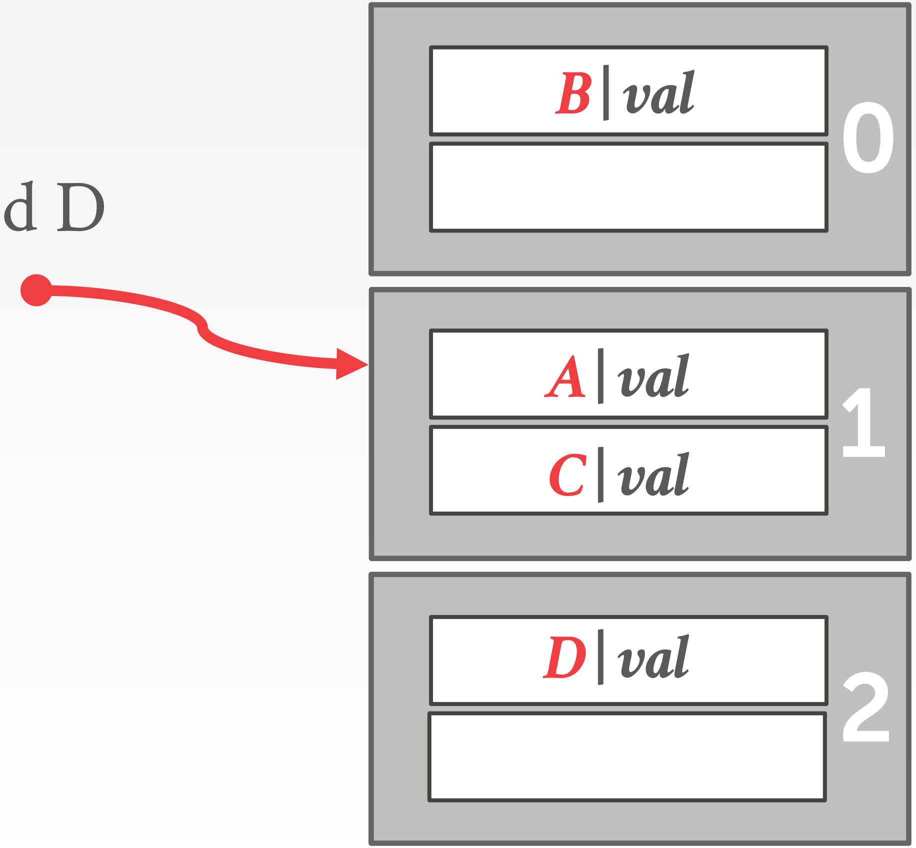
HASH TABLE – SLOT LATCHES

T_1: Find D

hash(D)

T_2: Insert E

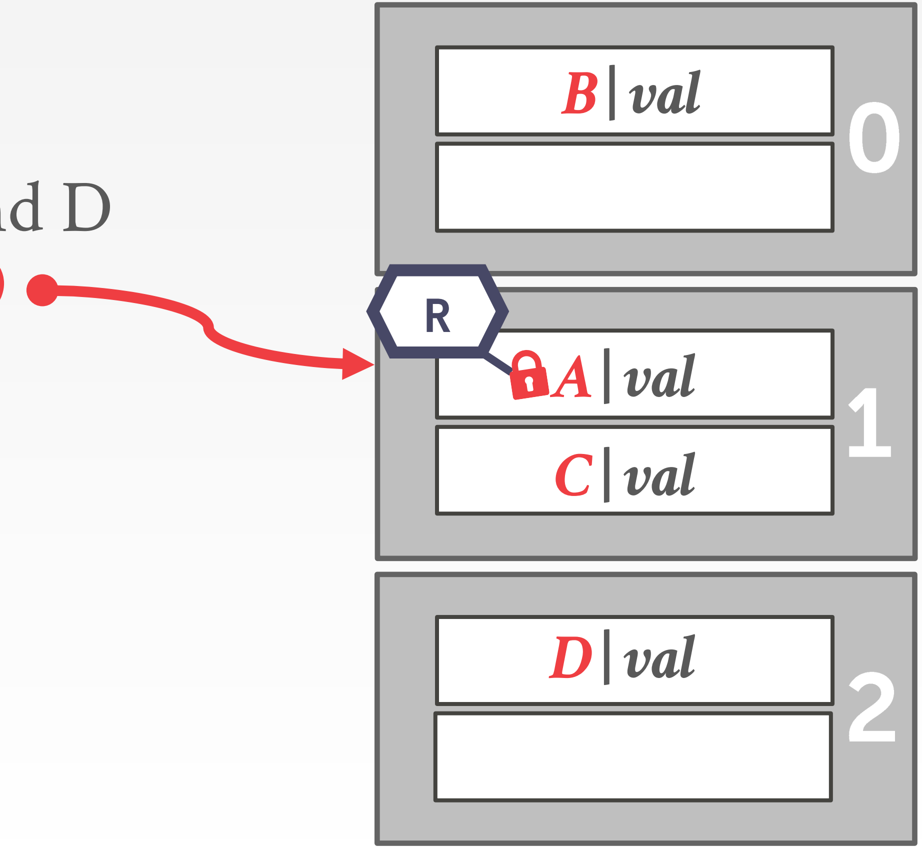
hash(E)



HASH TABLE – SLOT LATCHES

T_1: Find D

hash(D)

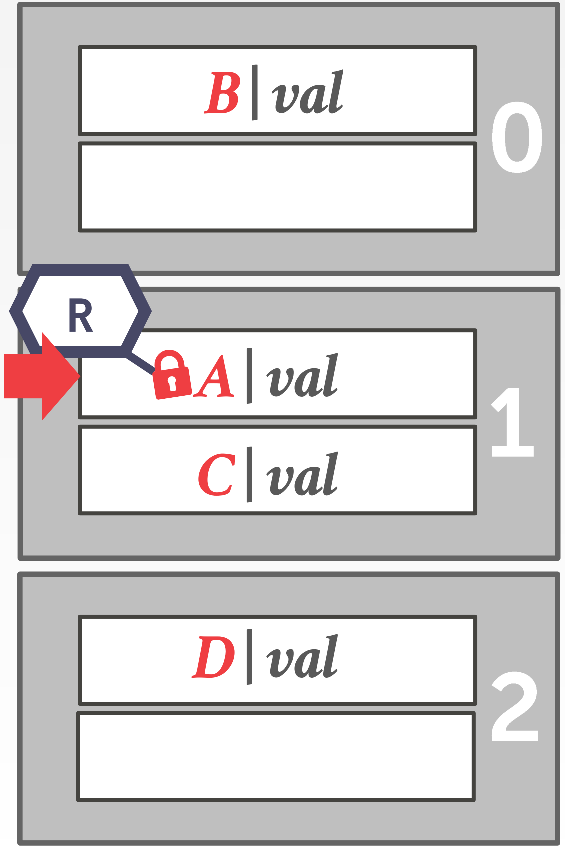


T_2: Insert E

hash(E)

HASH TABLE – SLOT LATCHES

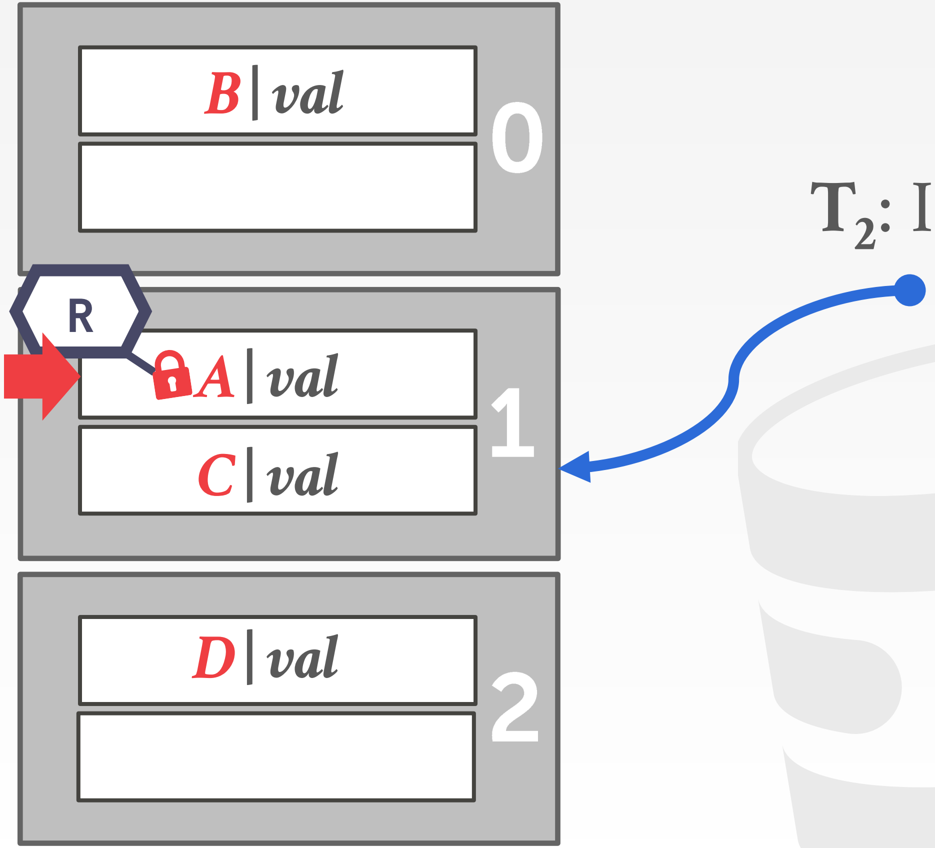
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

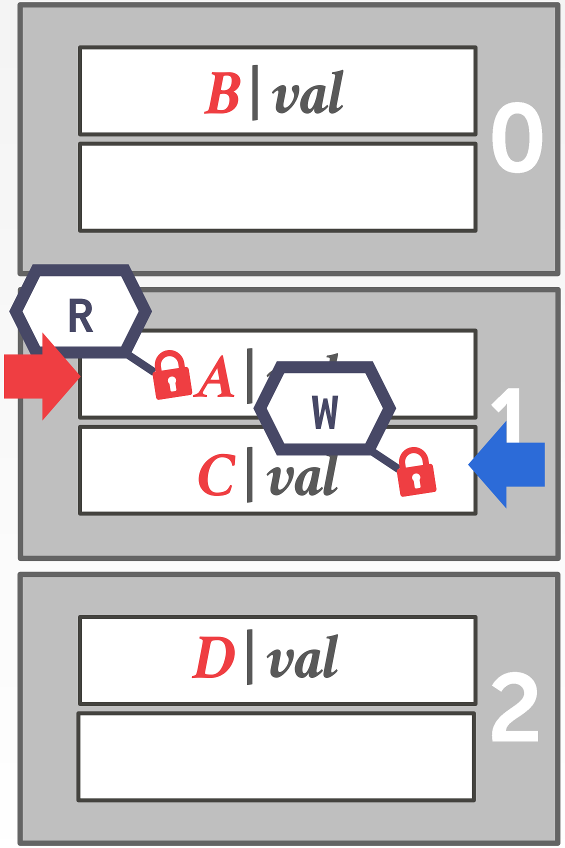
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

T_1: Find D
hash(D)

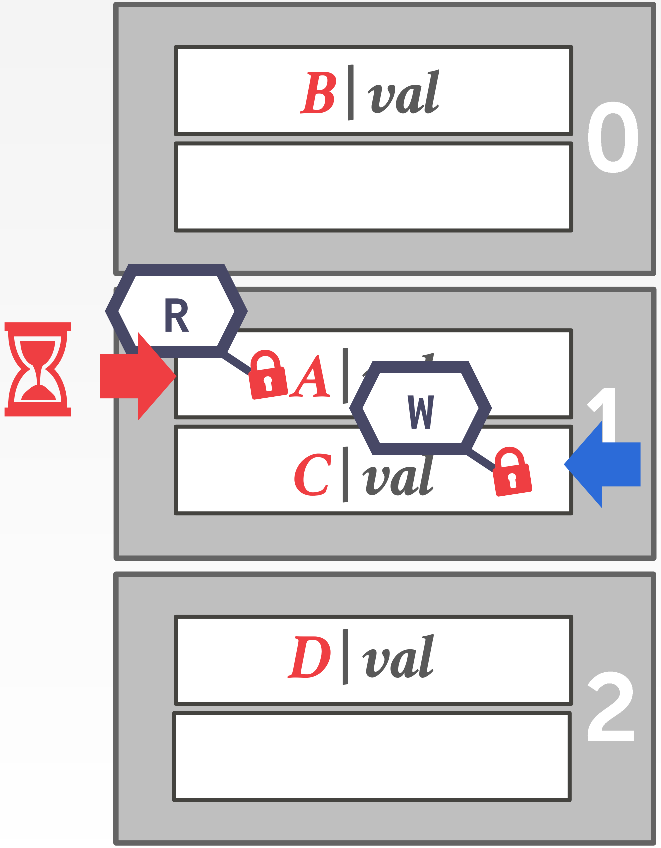


T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

T_1: Find D

hash(D)



T_2: Insert E

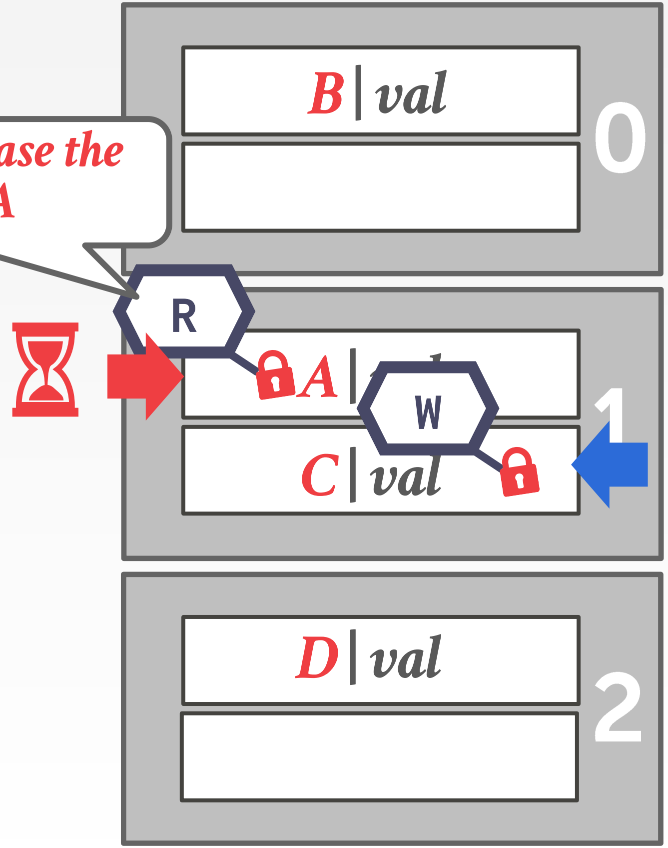
hash(E)

HASH TABLE – SLOT LATCHES

T_1:

hash(D)

It's safe to release the latch on A



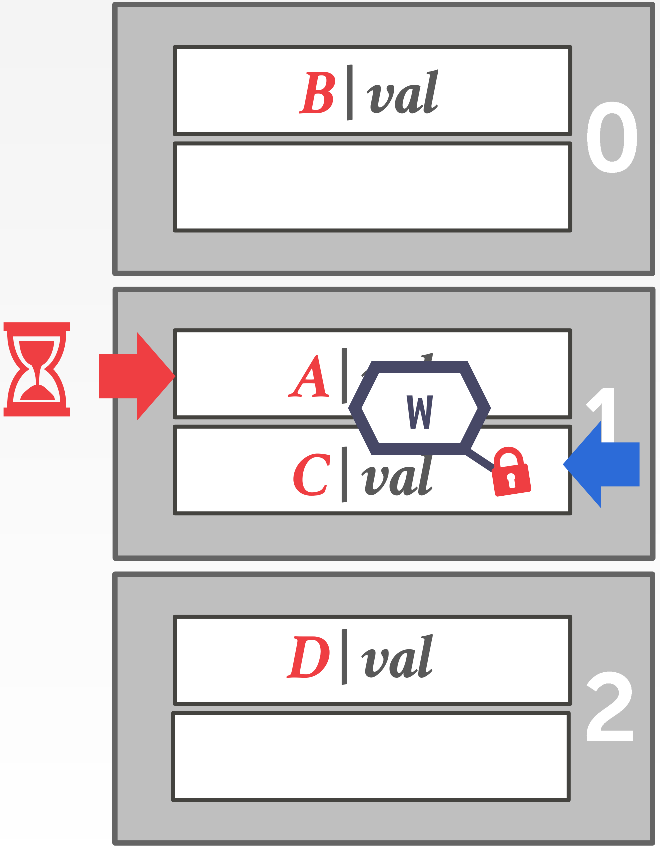
T_2: Insert E

hash(E)

HASH TABLE – SLOT LATCHES

T_1: Find D

hash(D)

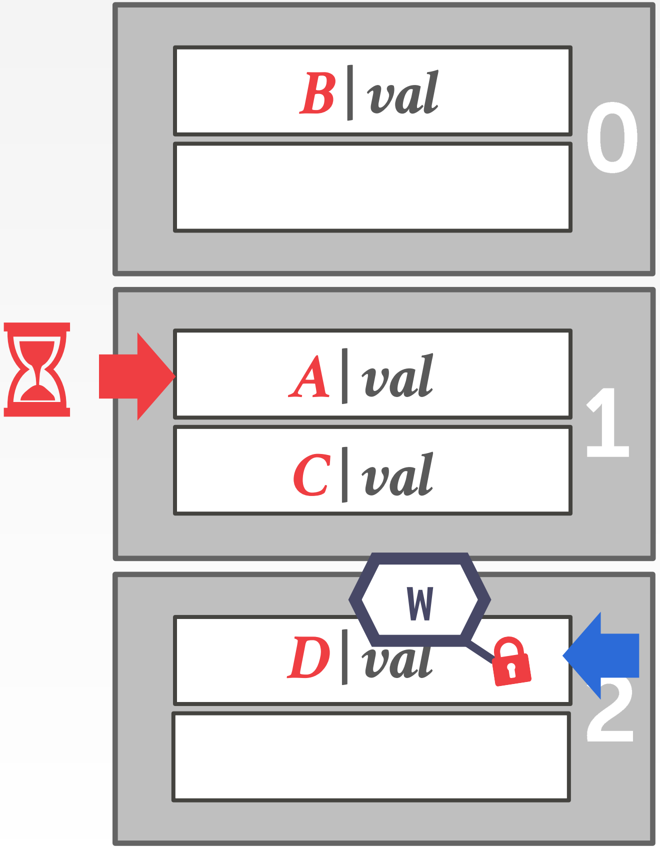


T_2: Insert E

hash(E)

HASH TABLE – SLOT LATCHES

T_1: Find D
hash(D)

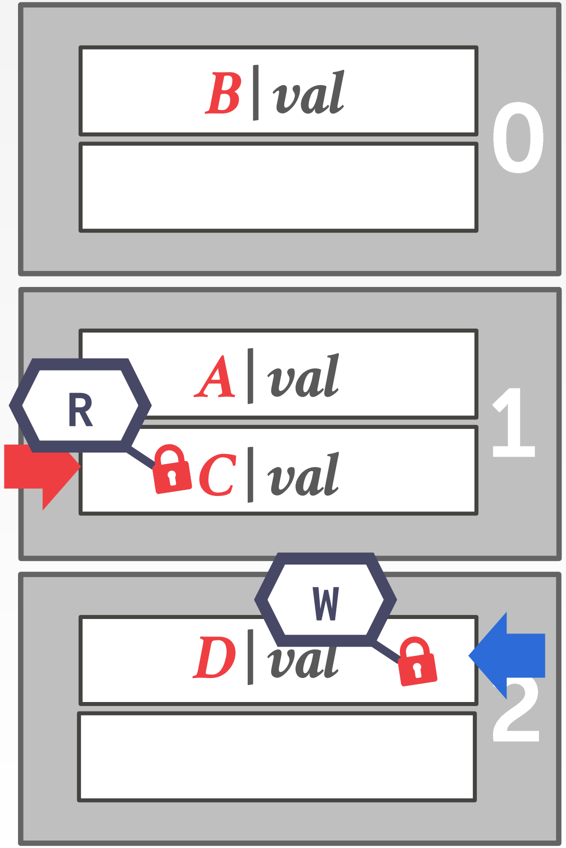


T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

T_1: Find D

hash(D)

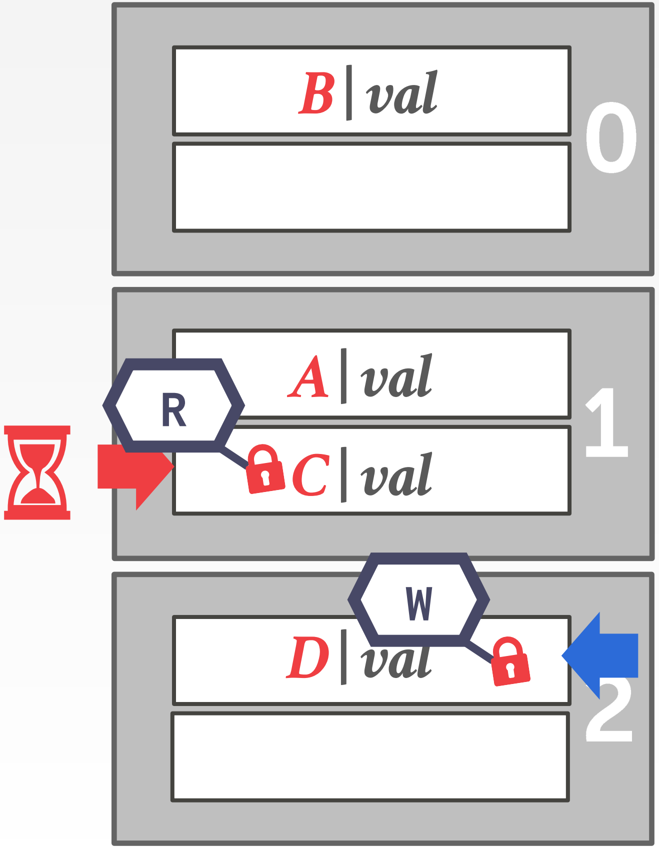


T_2: Insert E

hash(E)

HASH TABLE – SLOT LATCHES

T_1: Find D
hash(D)

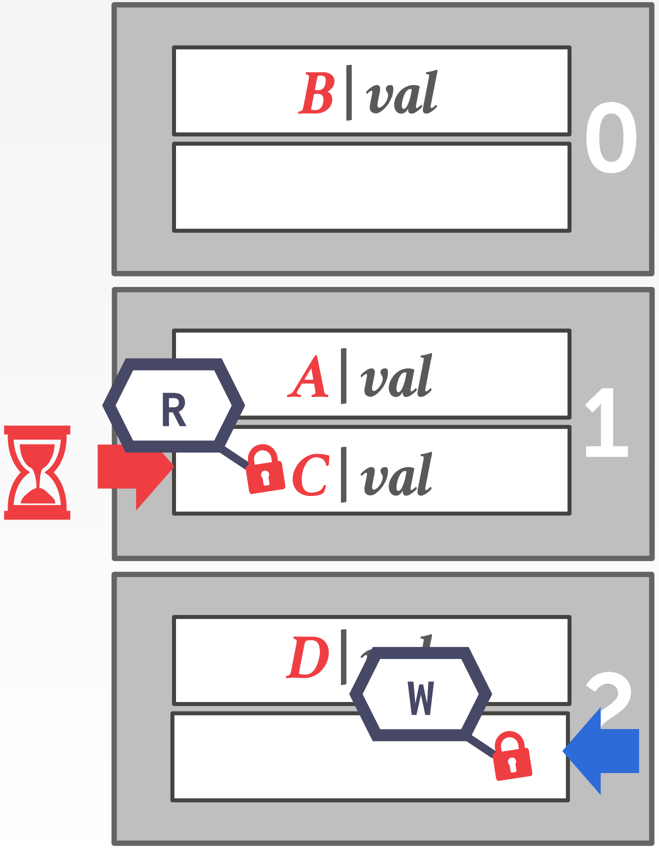


T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

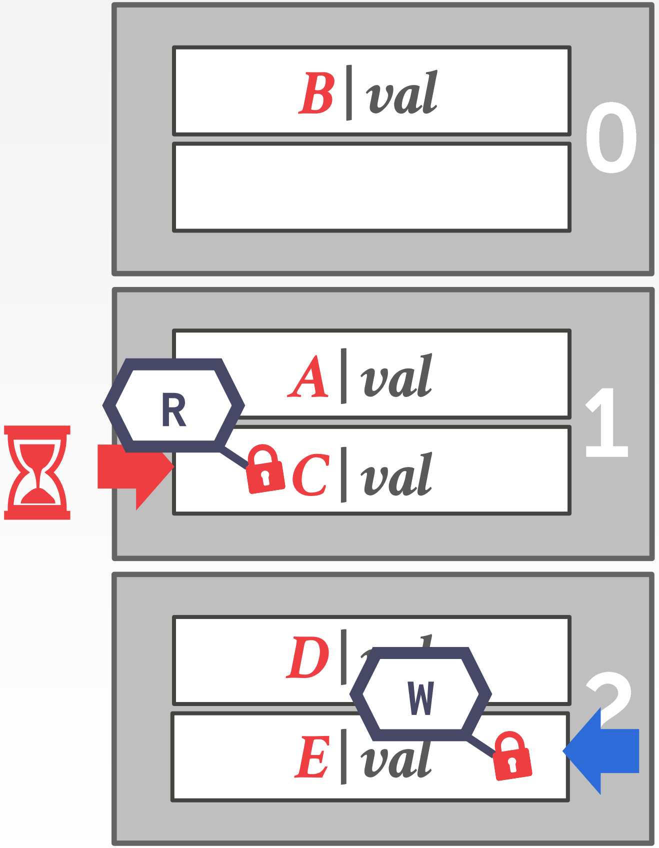
T_1: Find D
hash(D)

T_2: Insert E
hash(E)



HASH TABLE – SLOT LATCHES

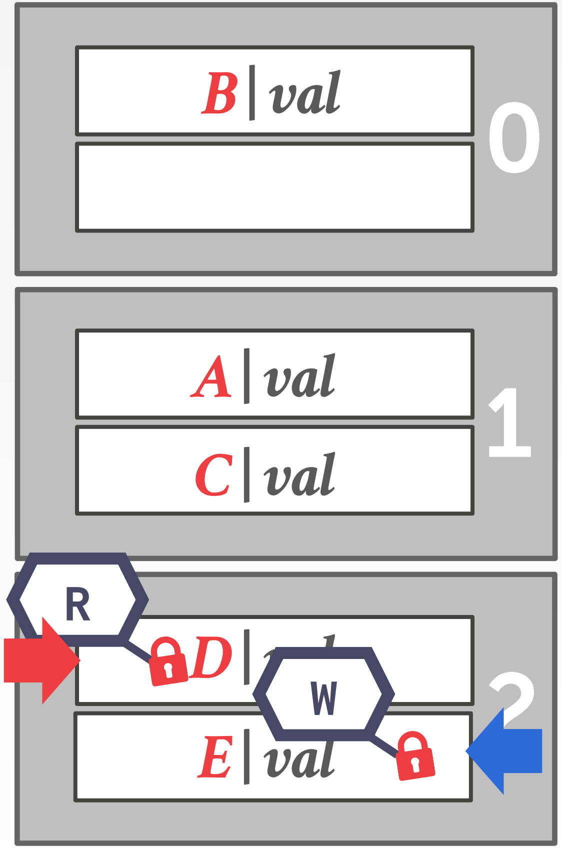
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

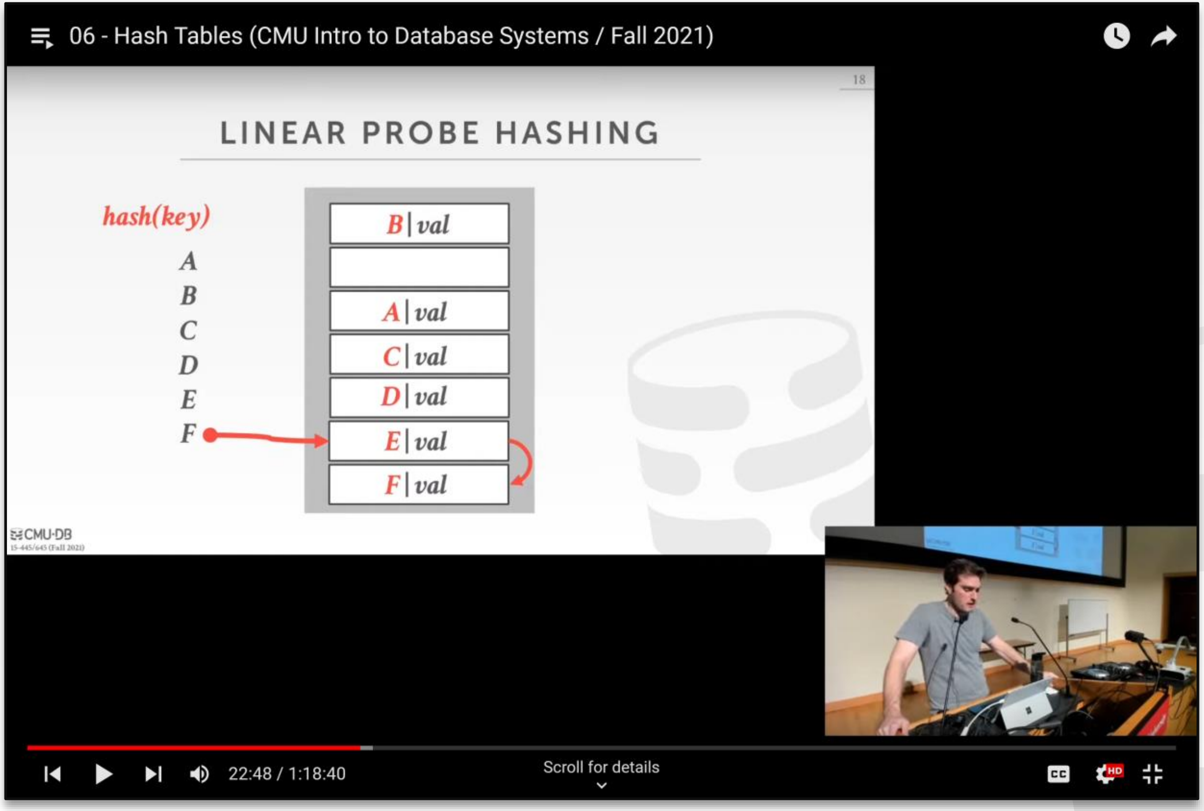
HASH TABLE – SLOT LATCHES

T_1: Find D
hash(D)

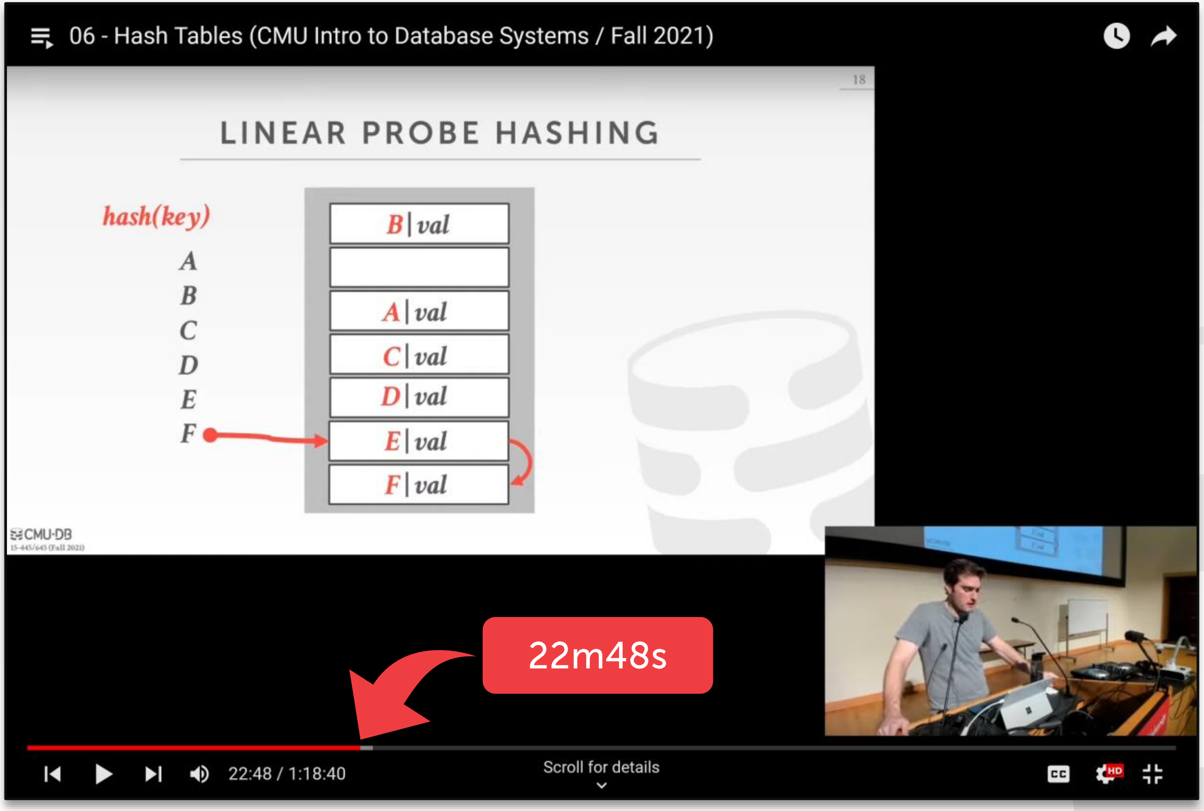


T_2: Insert E
hash(E)

HASH TABLE – NO LATCHES?



HASH TABLE – NO LATCHES?



COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location `M` to a given value `V`

→ If values are equal, installs new given value `V'` in `M`

→ Otherwise, operation fails

M

20

```
__sync_bool_compare_and_swap(&M, 20, 30)
```

COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V'** in **M**

→ Otherwise, operation fails

M

20

```
__sync_bool_compare_and_swap(&M, 20, 30)
```

Address

New Value

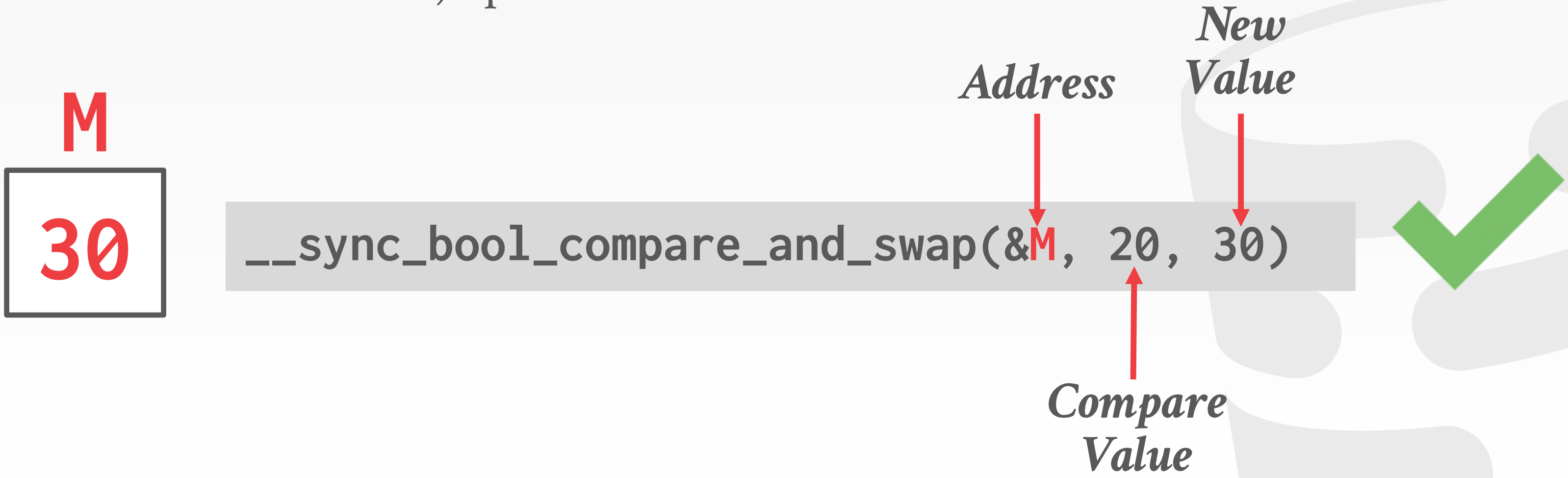
Compare Value

COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V'** in **M**

→ Otherwise, operation fails



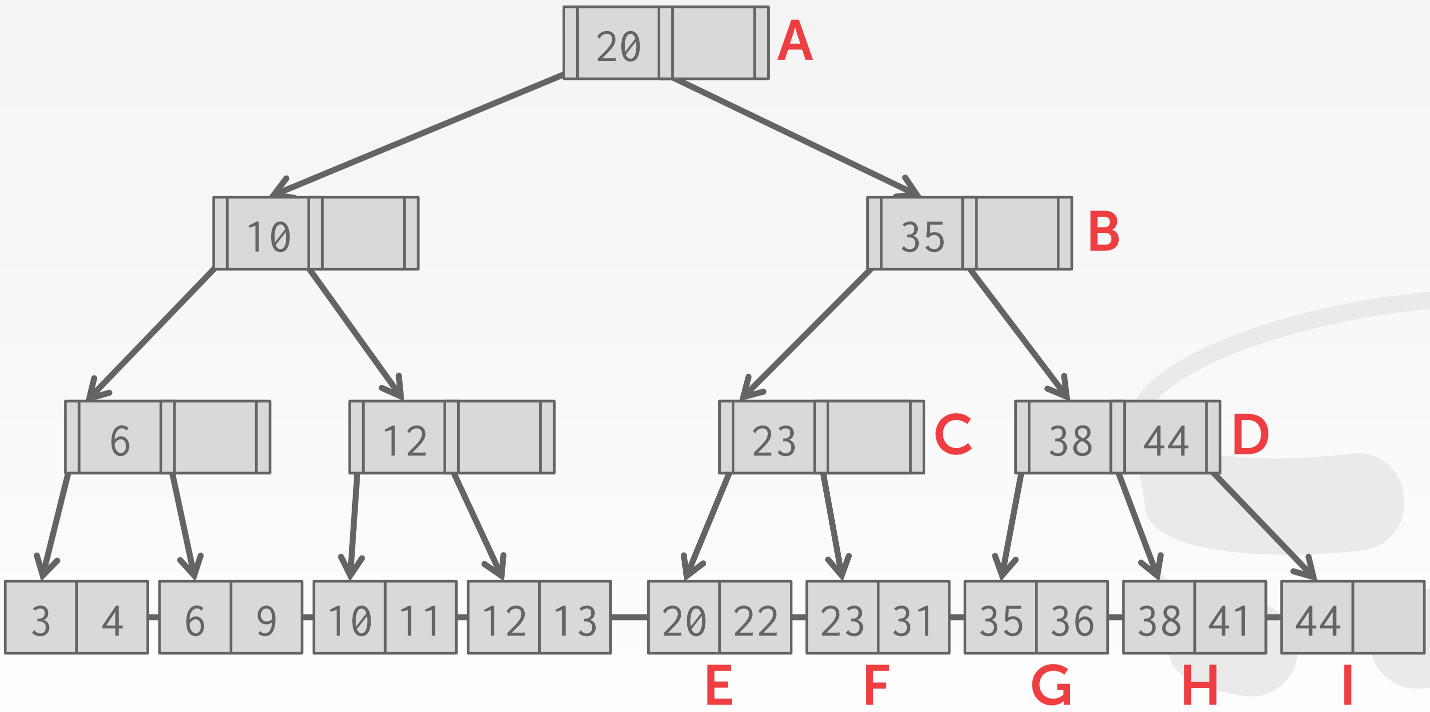
B+TREE CONCURRENCY CONTROL

We want to allow multiple threads to read and update a B+Tree at the same time.

We need to protect against two types of problems:

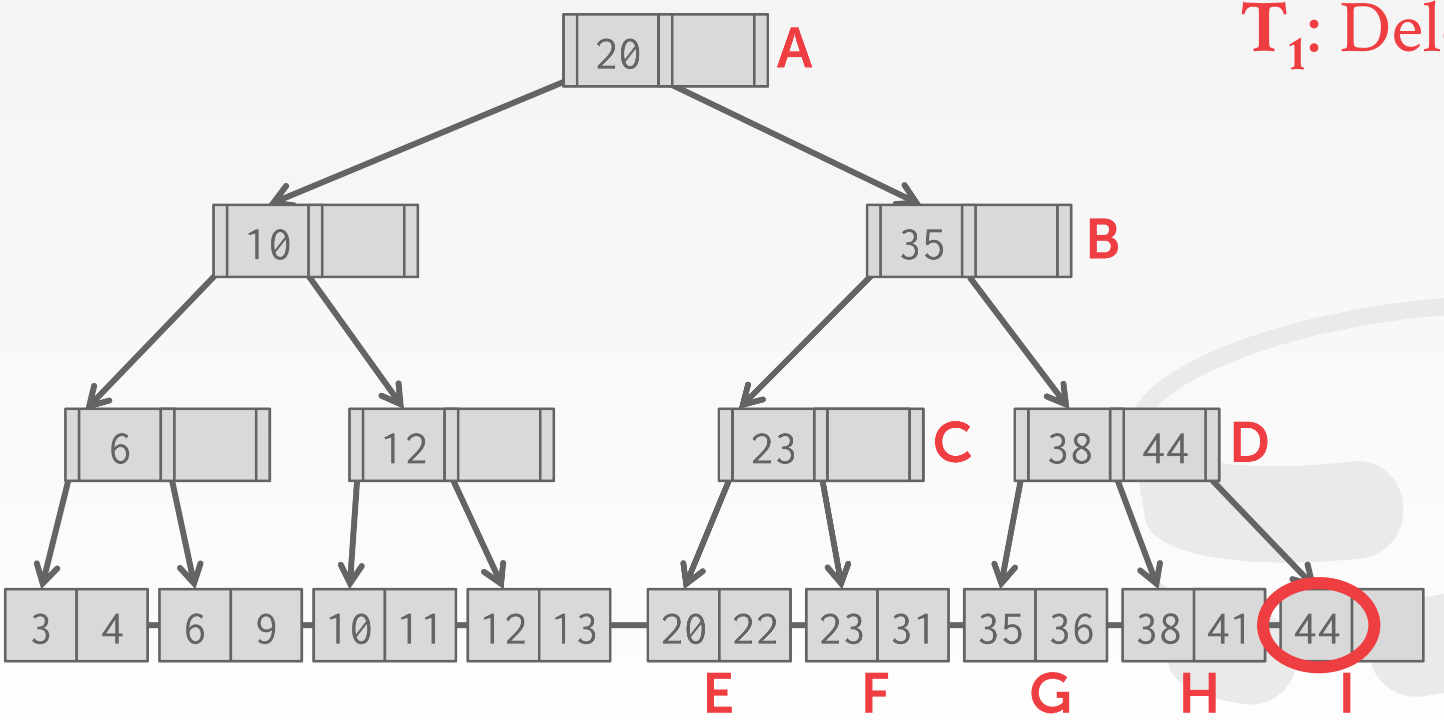
→ Threads trying to modify the contents of a node at the same time.
→ One thread traversing the tree while another thread splits/merges nodes.

B+TREE MULTI-THREADED EXAMPLE

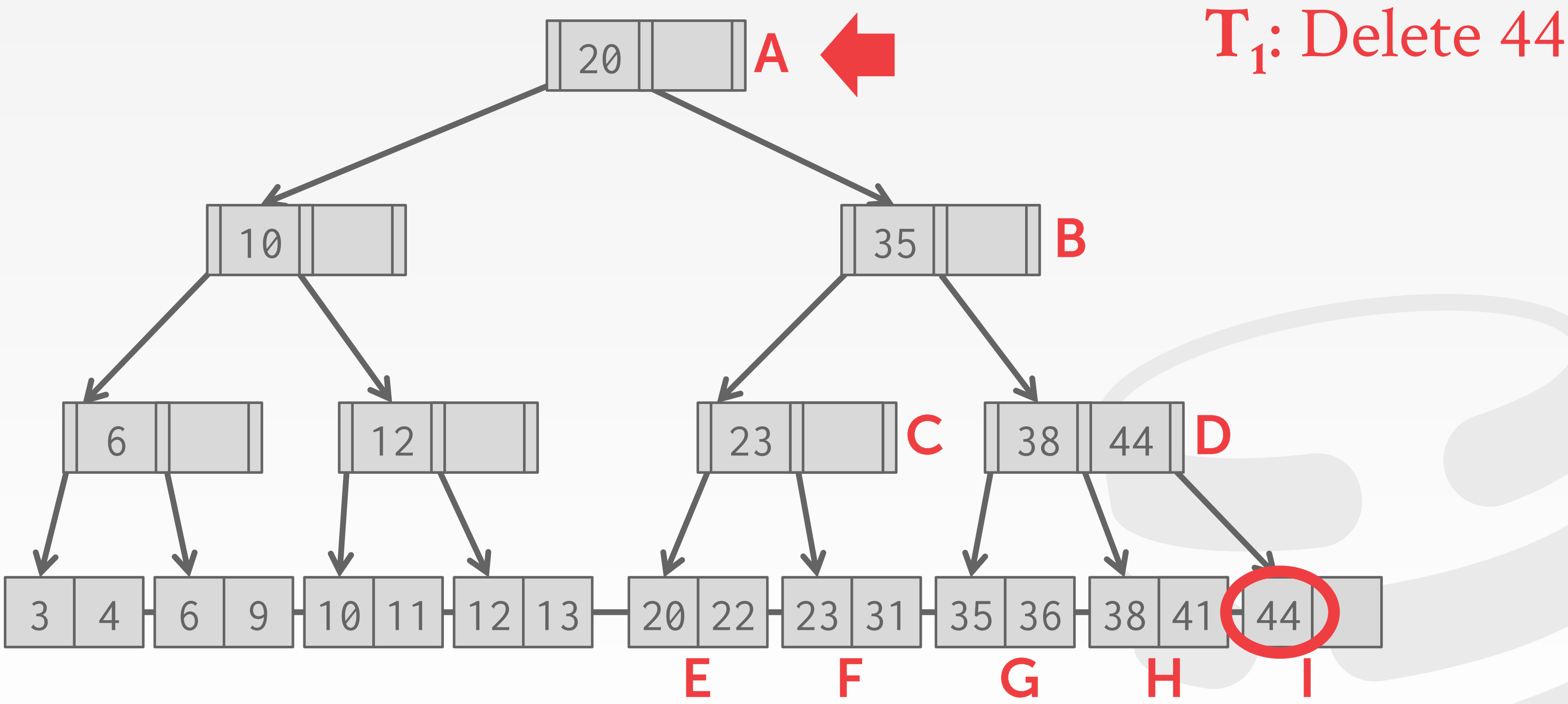


B+TREE MULTI-THREADED EXAMPLE

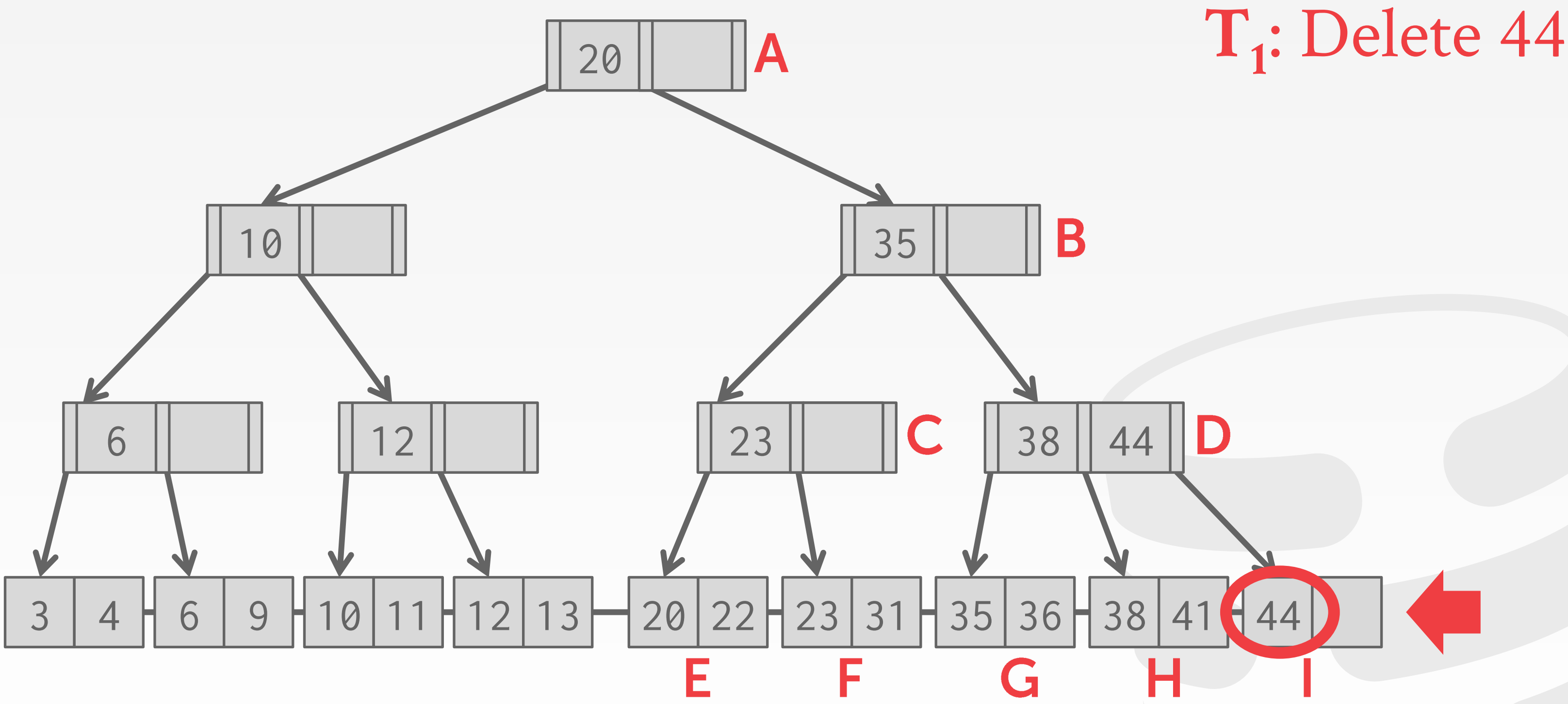
T_1: Delete 44



B+TREE MULTI-THREADED EXAMPLE

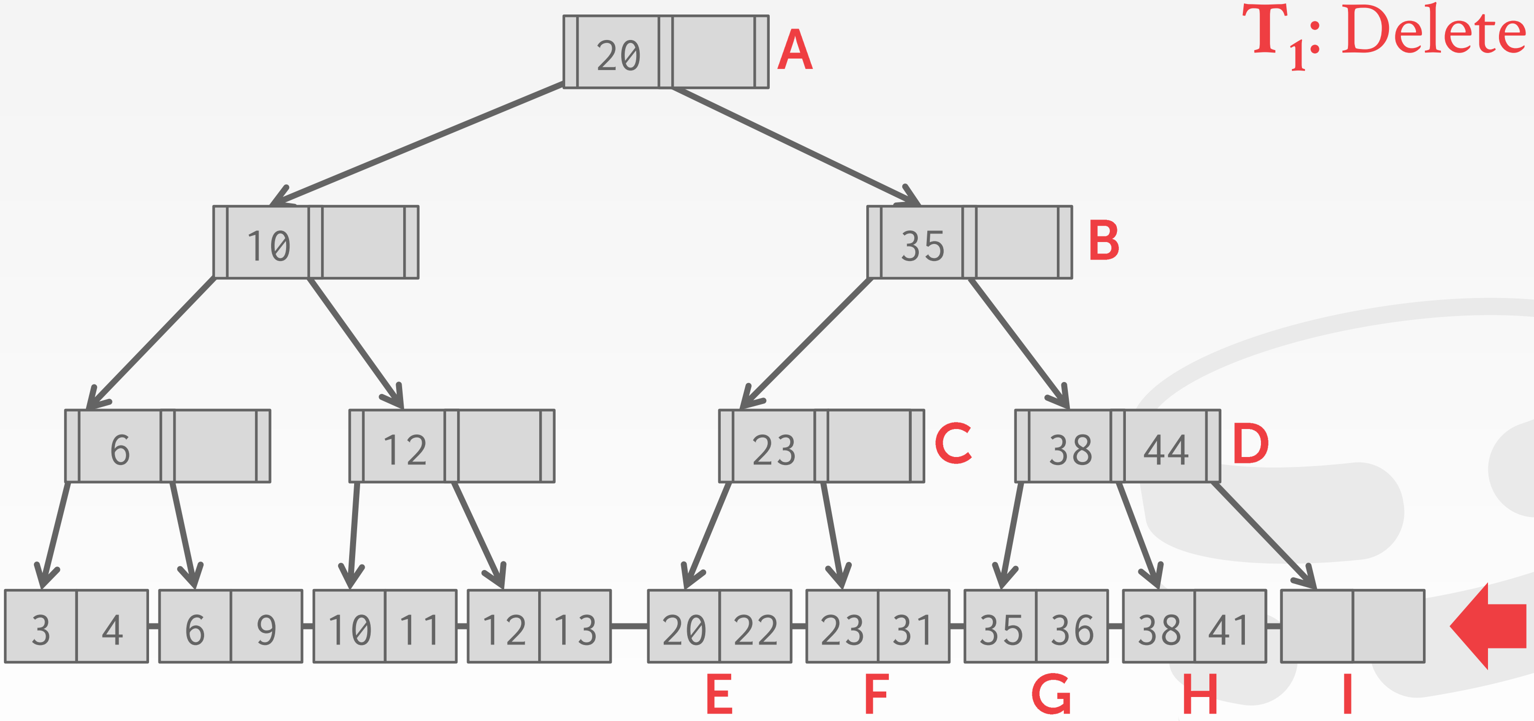


B+TREE MULTI-THREADED EXAMPLE

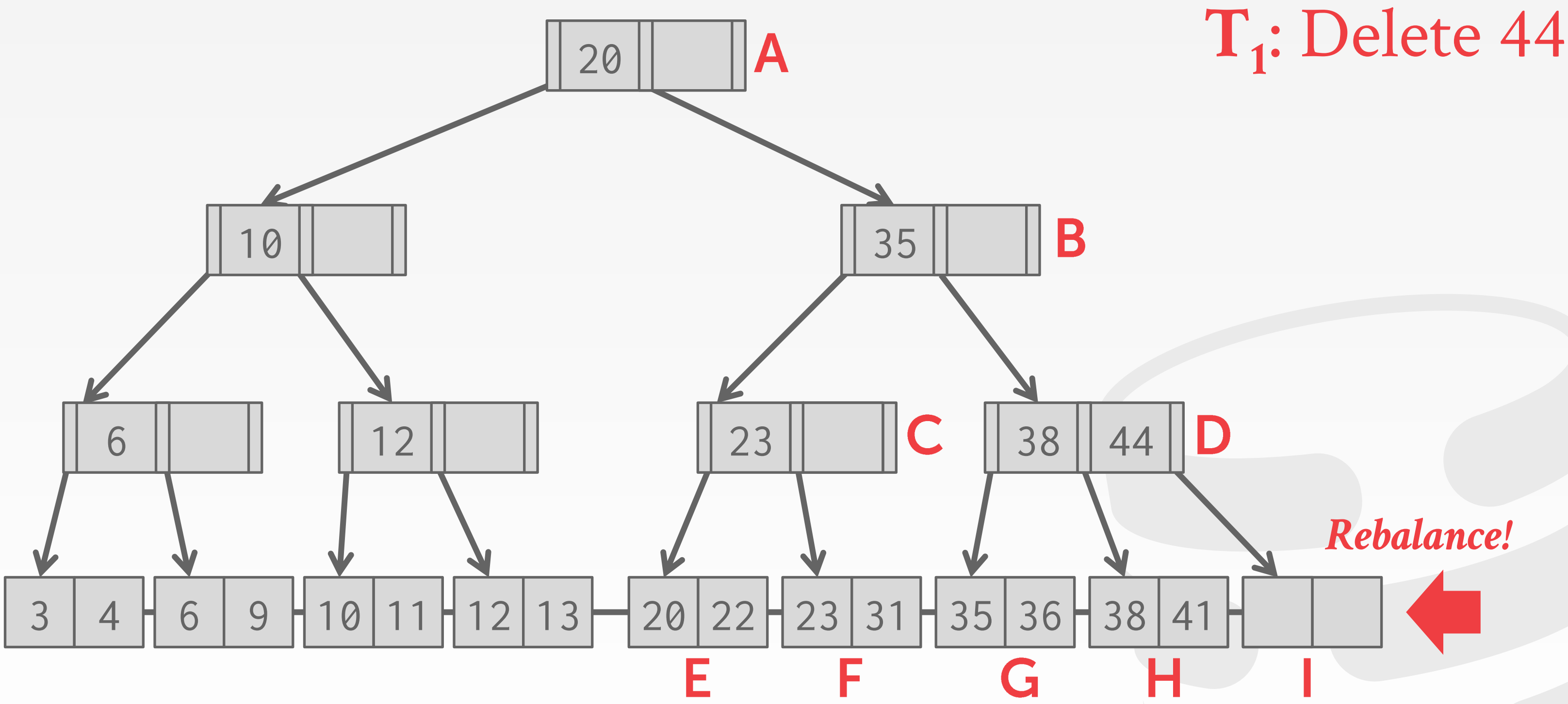


B+TREE MULTI-THREADED EXAMPLE

T_1: Delete 44

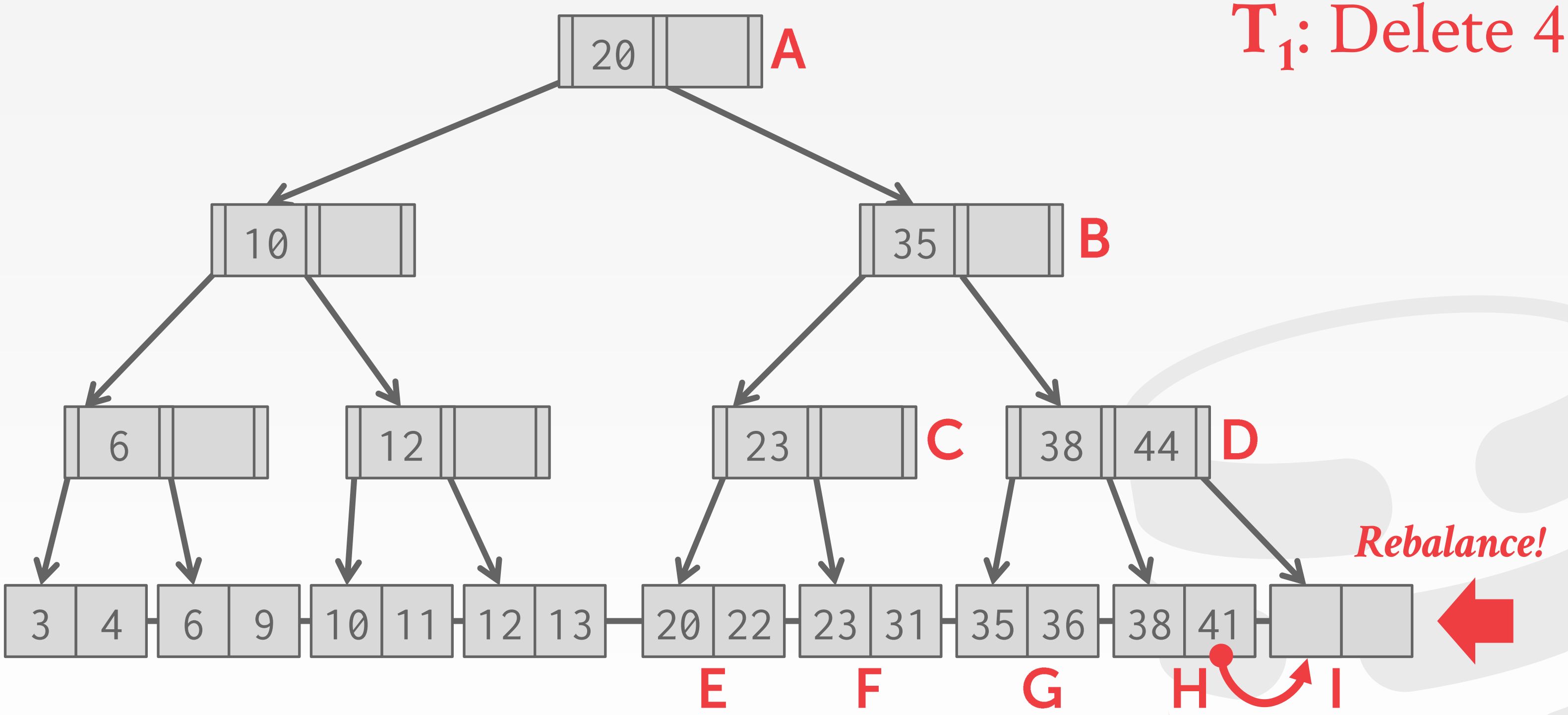


B+TREE MULTI-THREADED EXAMPLE

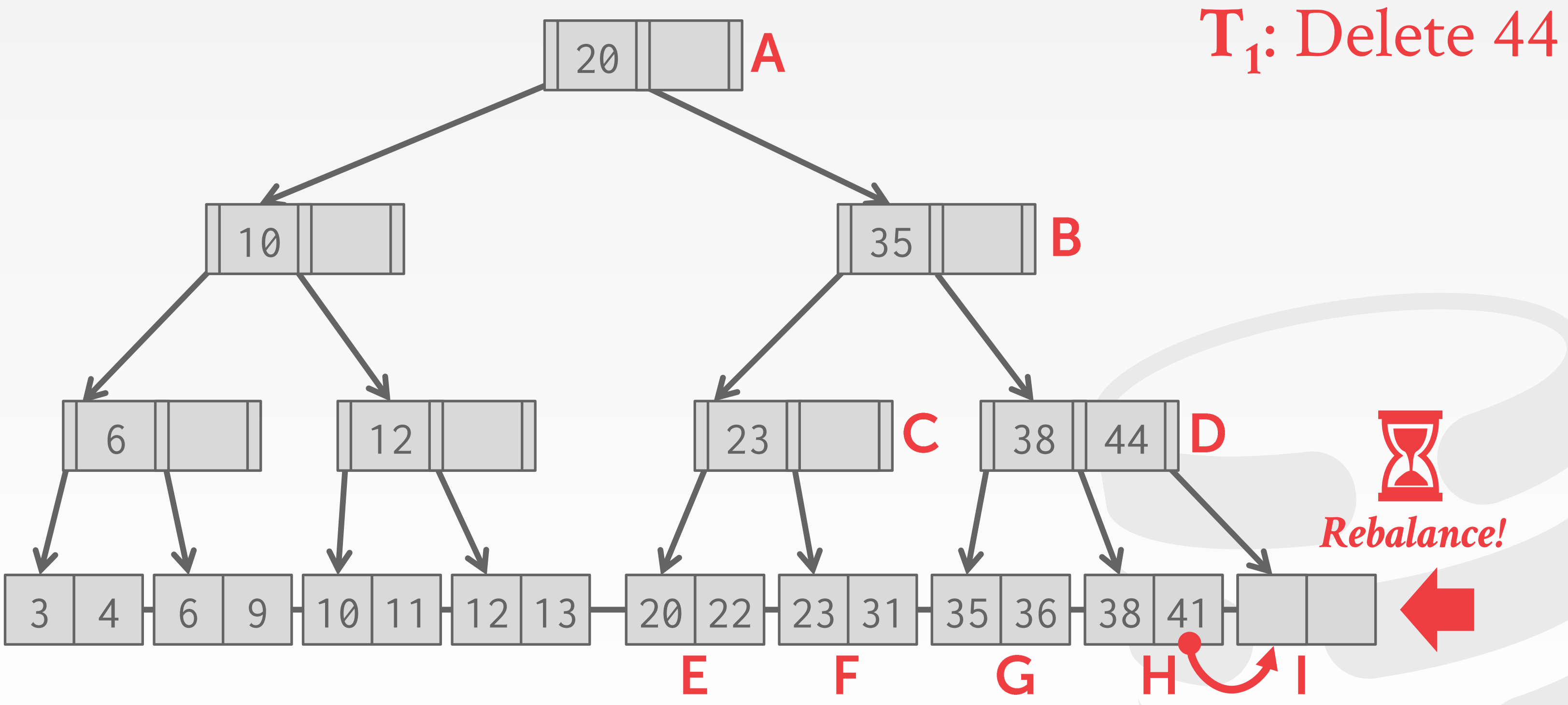


B+TREE MULTI-THREADED EXAMPLE

T$_1$: Delete 44



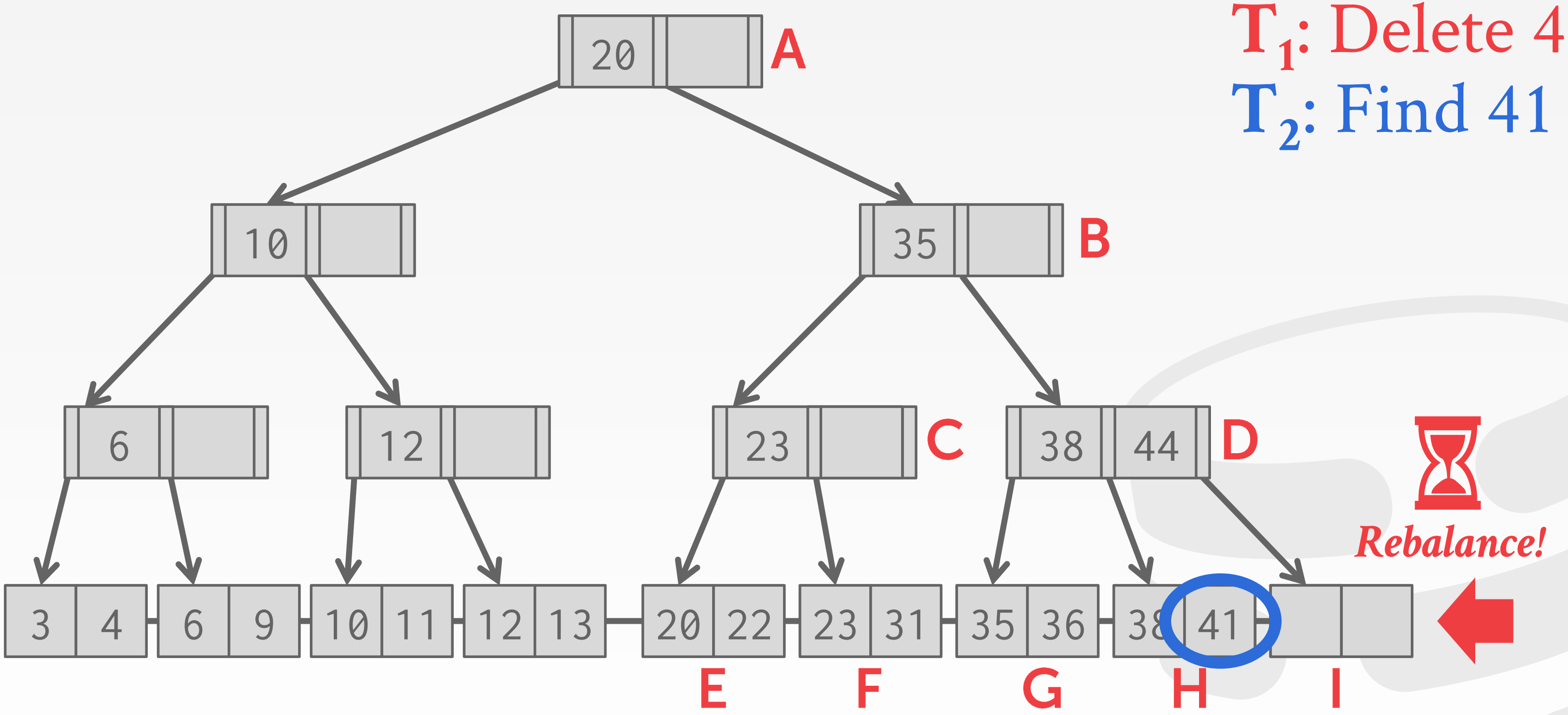
B+TREE MULTI-THREADED EXAMPLE



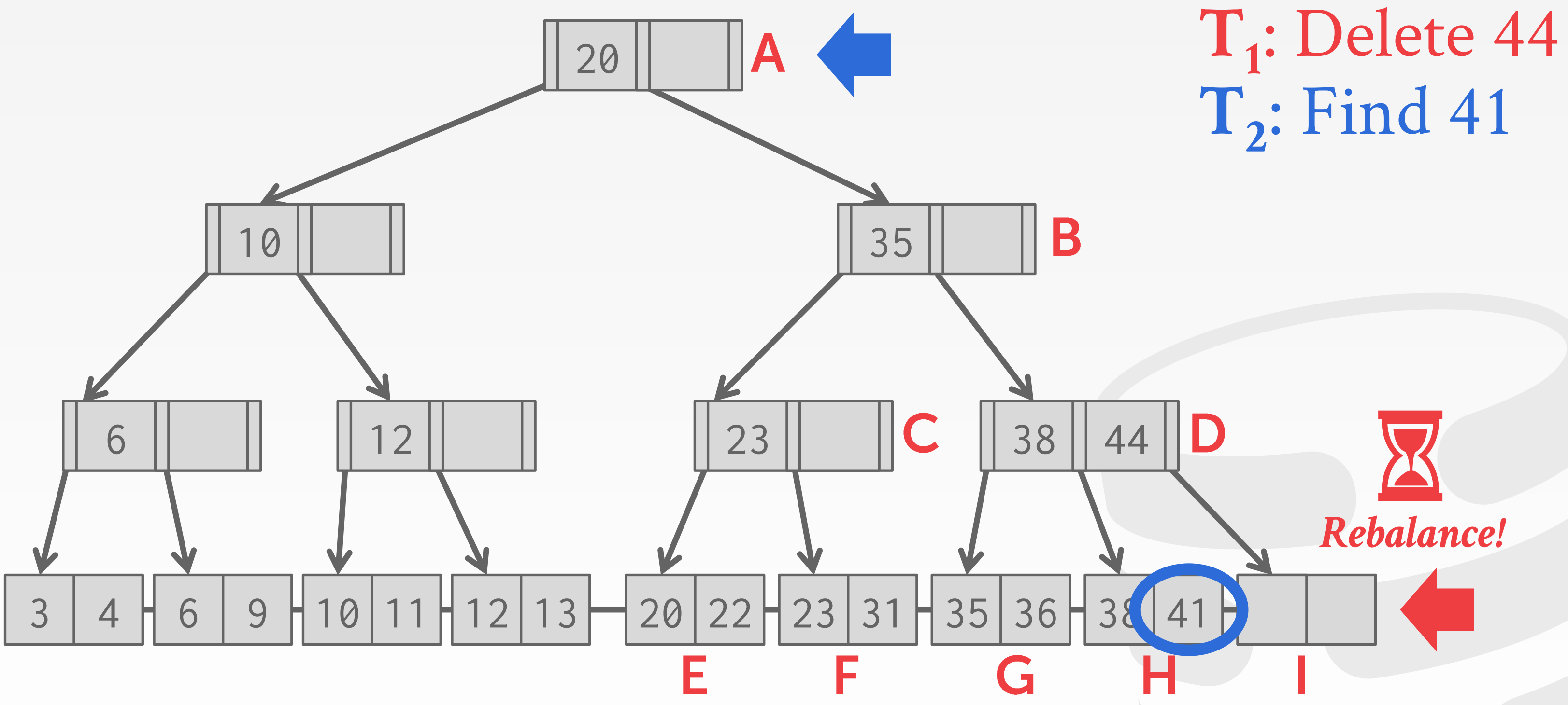
B+TREE MULTI-THREADED EXAMPLE

T_1: Delete 44

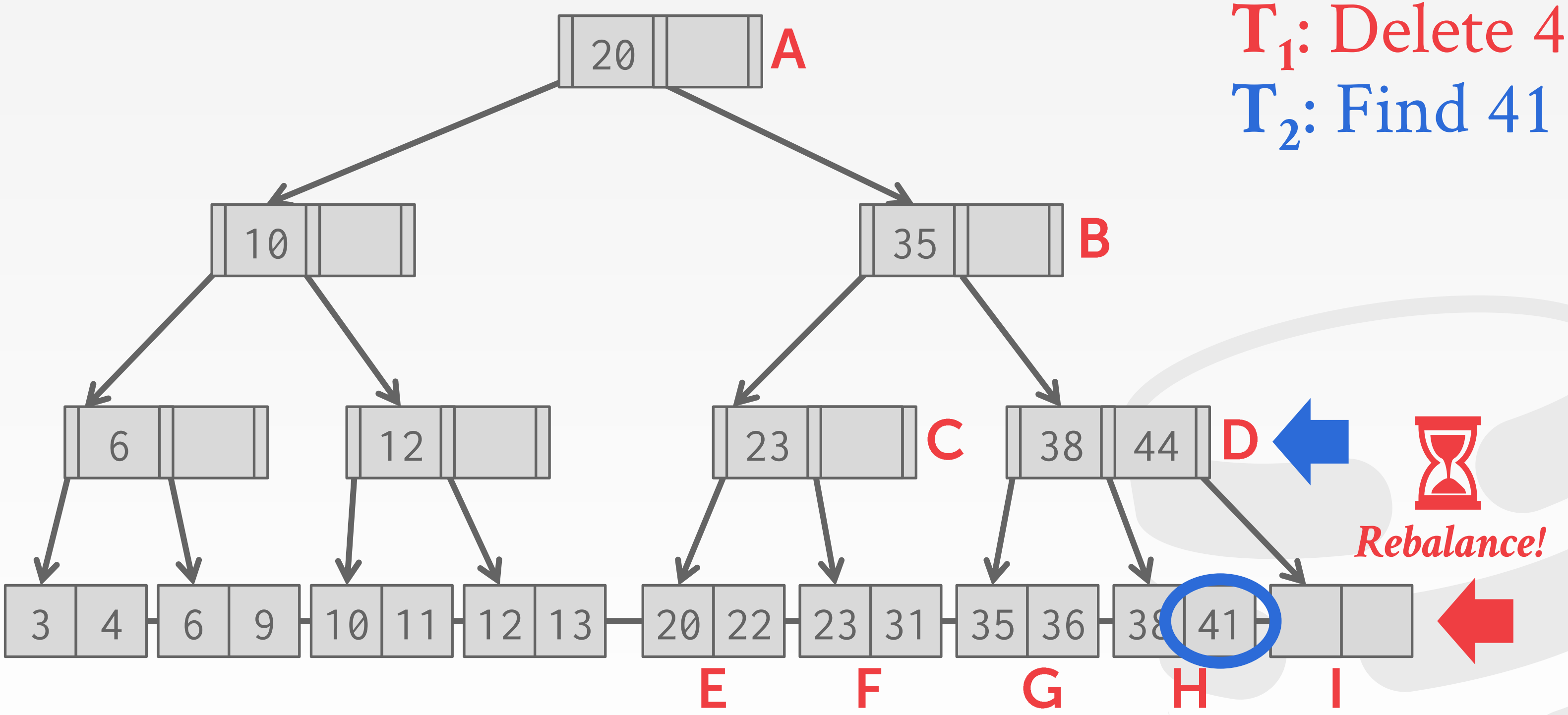
T_2: Find 41



B+TREE MULTI-THREADED EXAMPLE



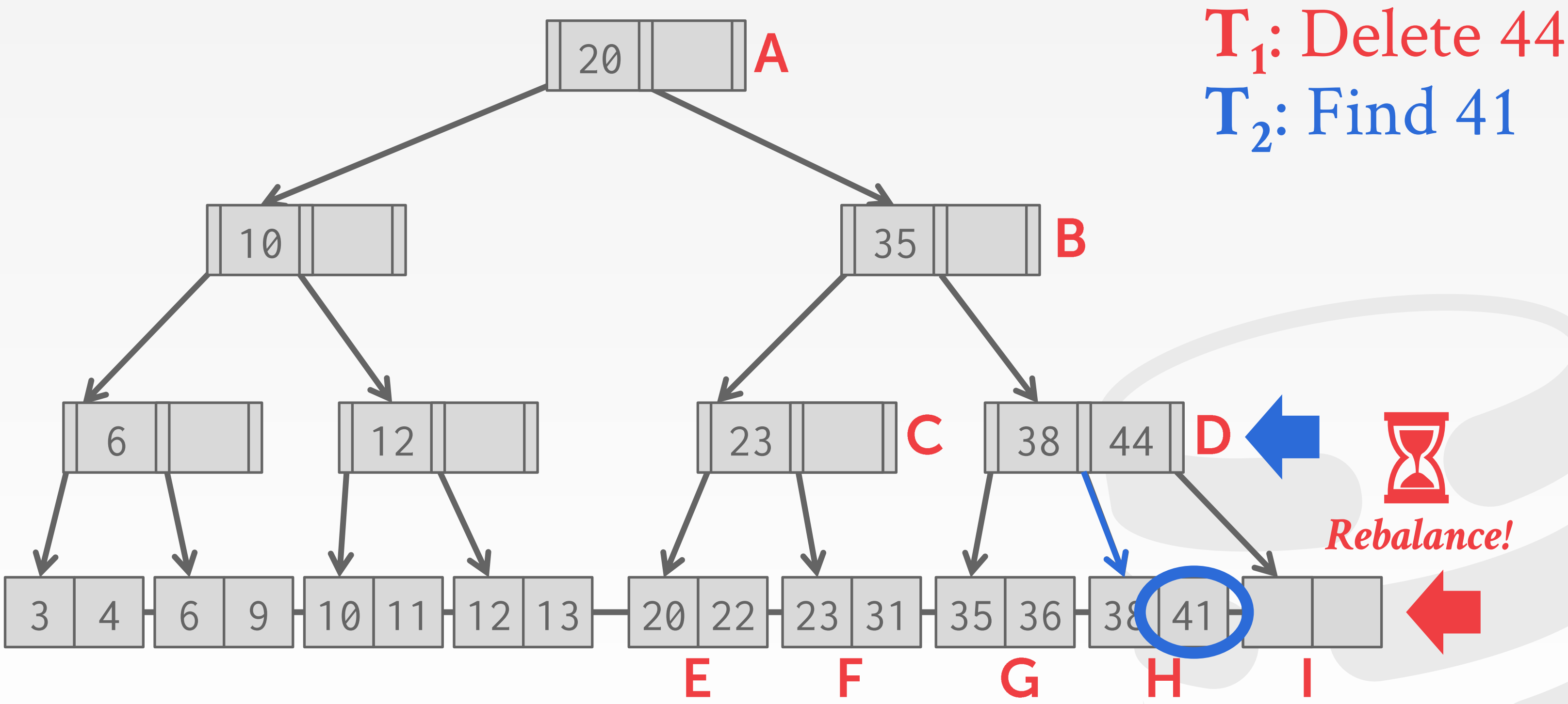
B+TREE MULTI-THREADED EXAMPLE



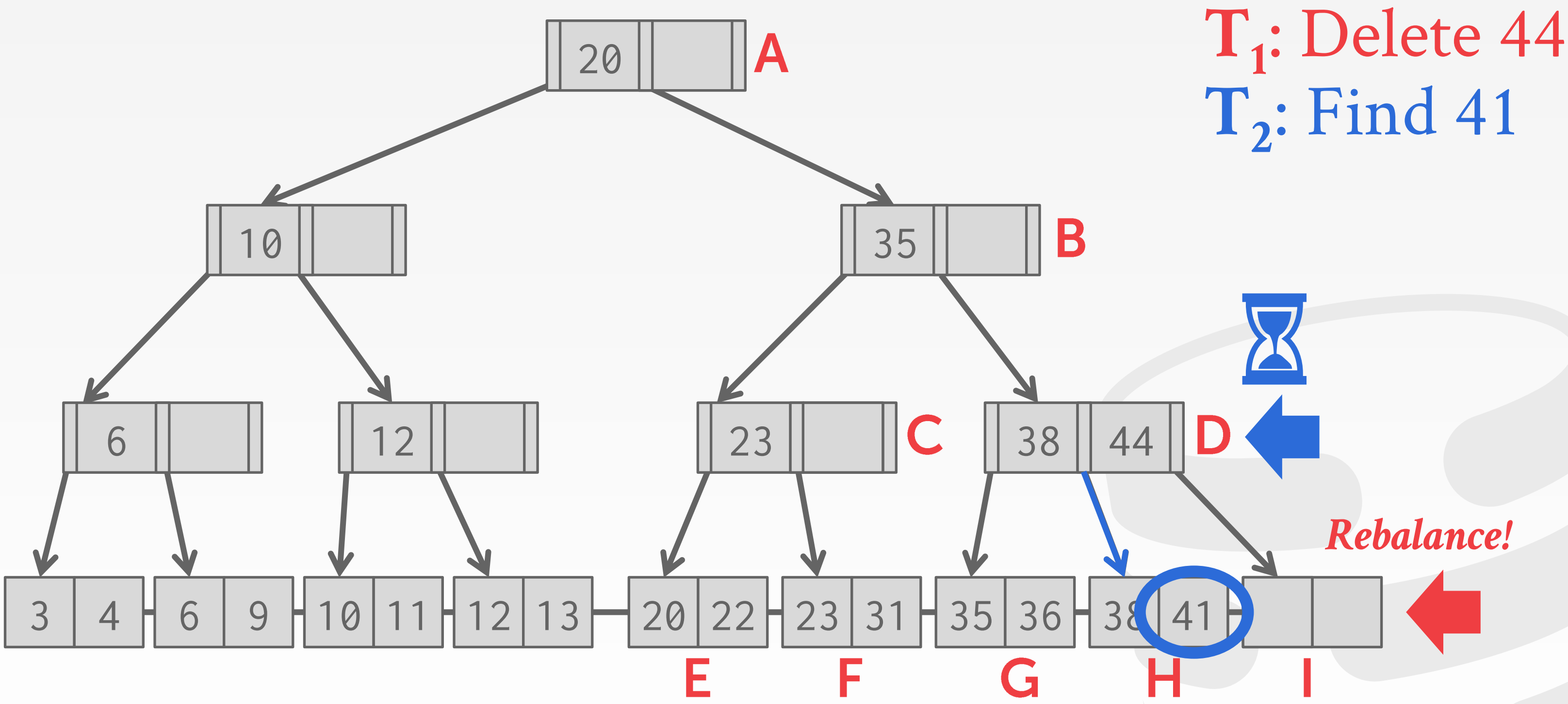
T_1: Delete 44

T_2: Find 41

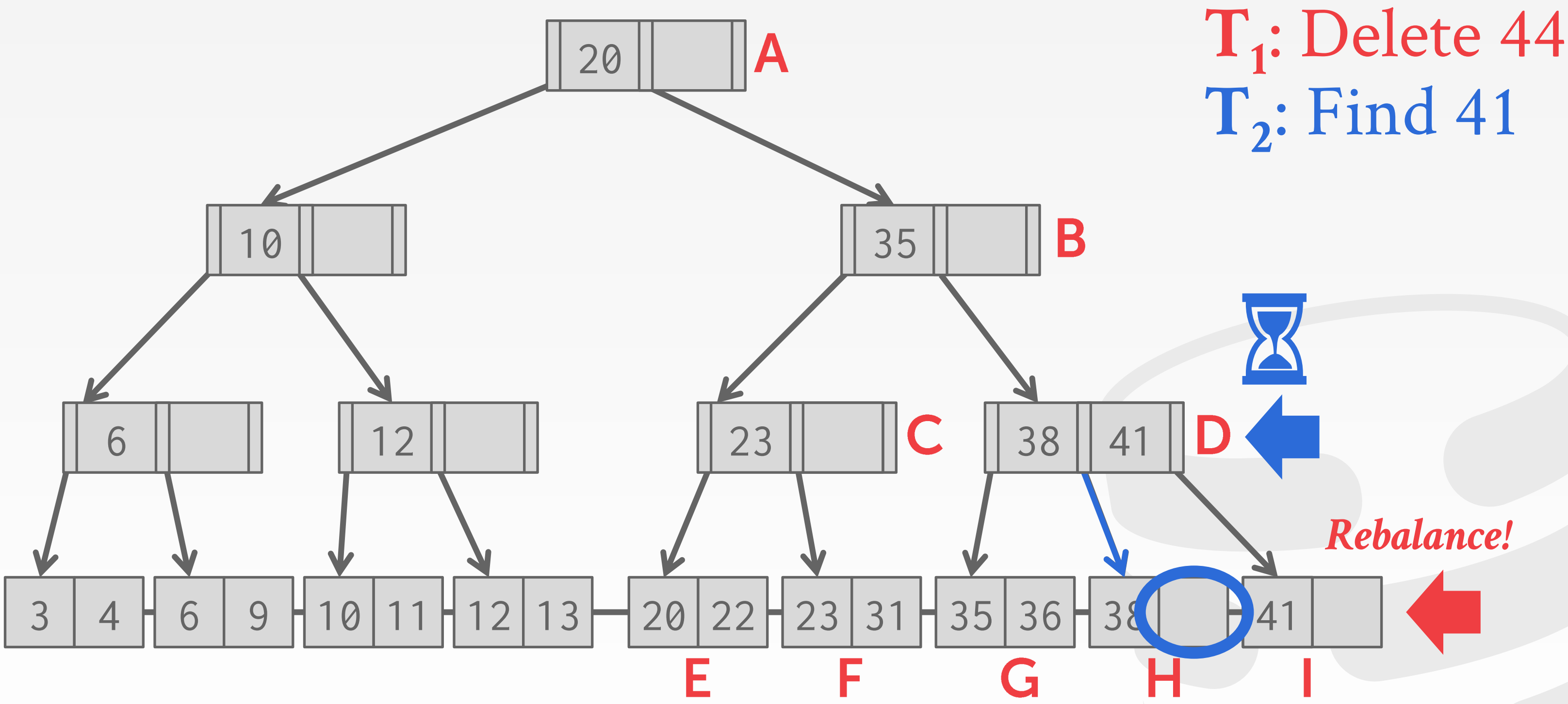
B+TREE MULTI-THREADED EXAMPLE



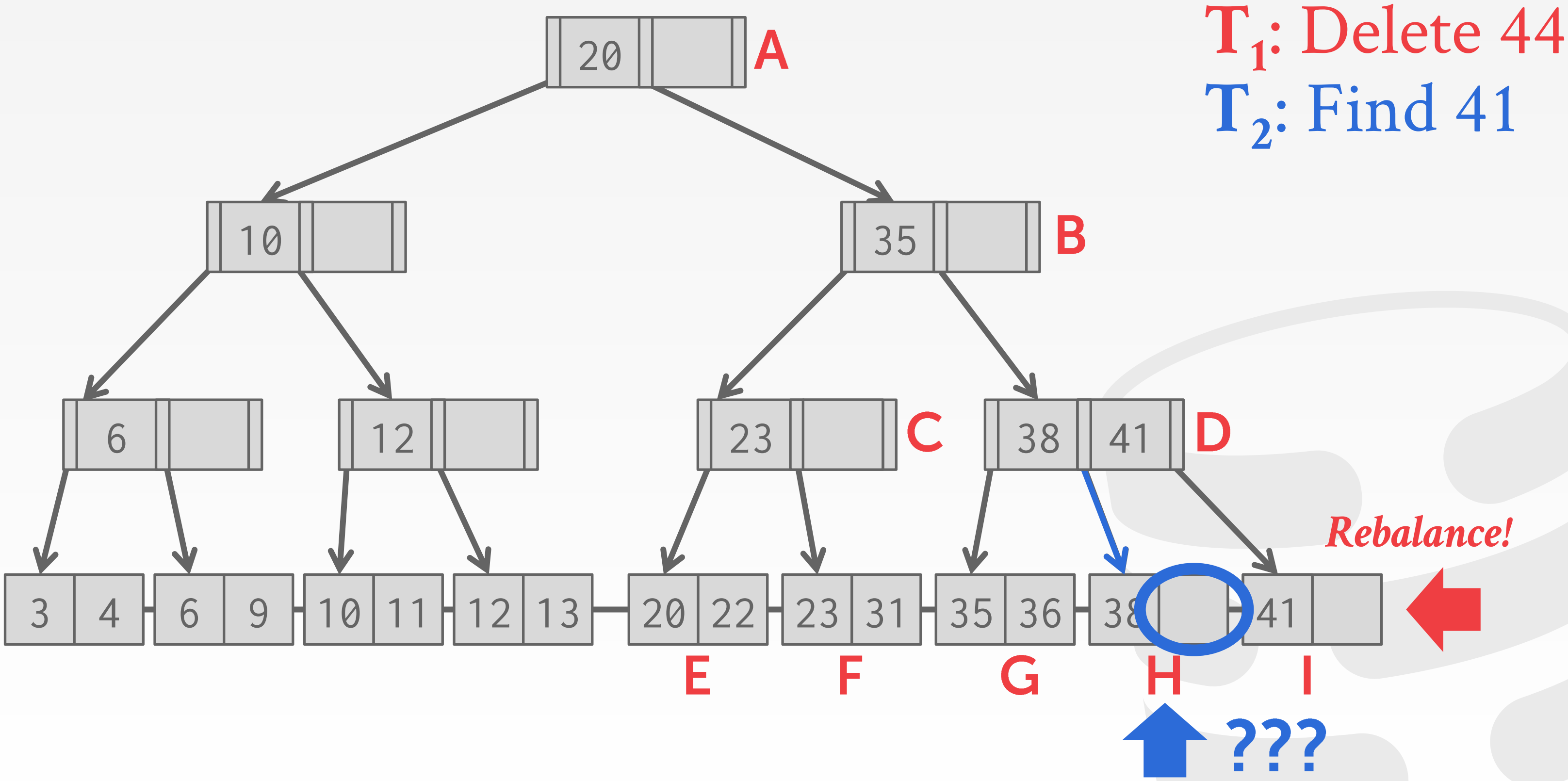
B+TREE MULTI-THREADED EXAMPLE



B+TREE MULTI-THREADED EXAMPLE



B+TREE MULTI-THREADED EXAMPLE



LATCH CRABBING/COUPLING

Protocol to allow multiple threads to access/modify B+Tree at the same time.

Basic Idea:

→ Get latch for parent
→ Get latch for child
→ Release latch for parent if "safe"

A **safe node** is one that will not split or merge when updated.

→ Not full (on insertion)
→ More than half-full (on deletion)

LATCH CRABBING/COUPLING

Find: Start at root and go down; repeatedly,

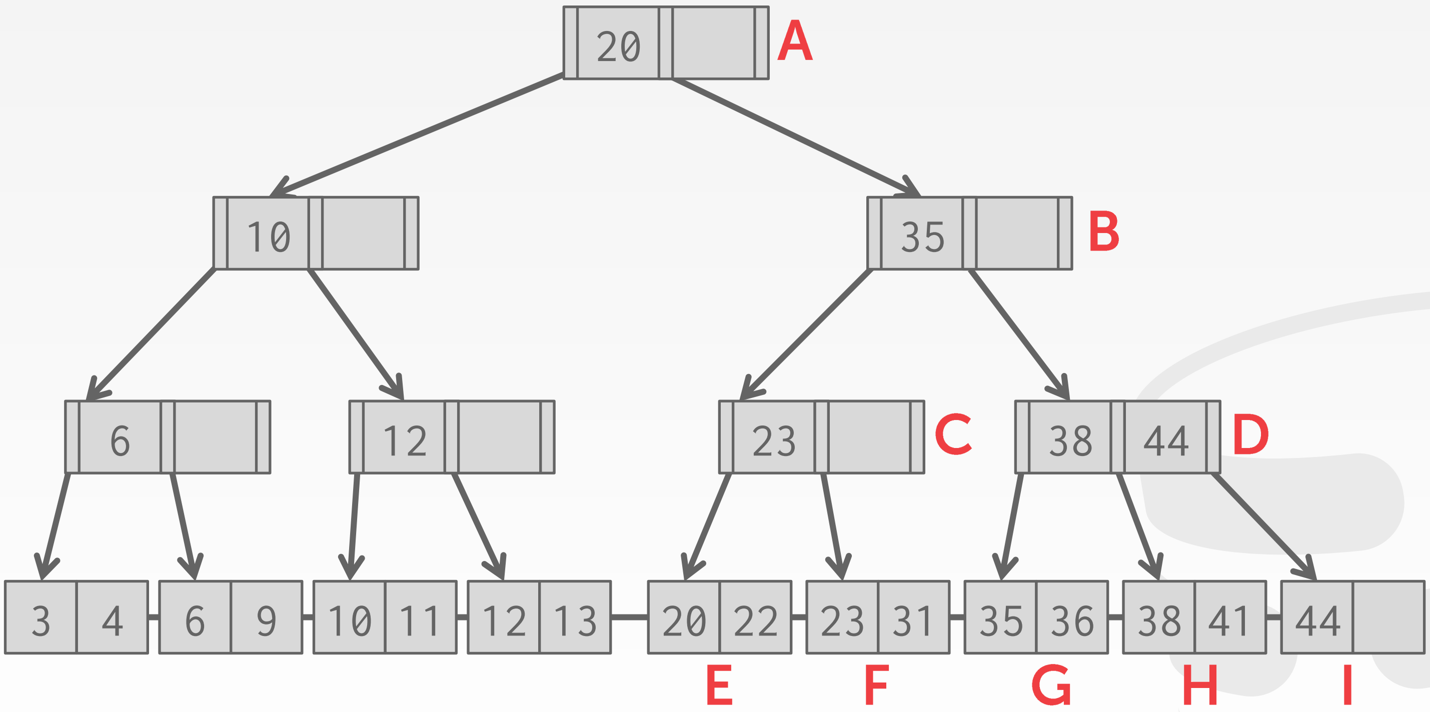
→ Acquire **R** latch on child

→ Then unlatch parent

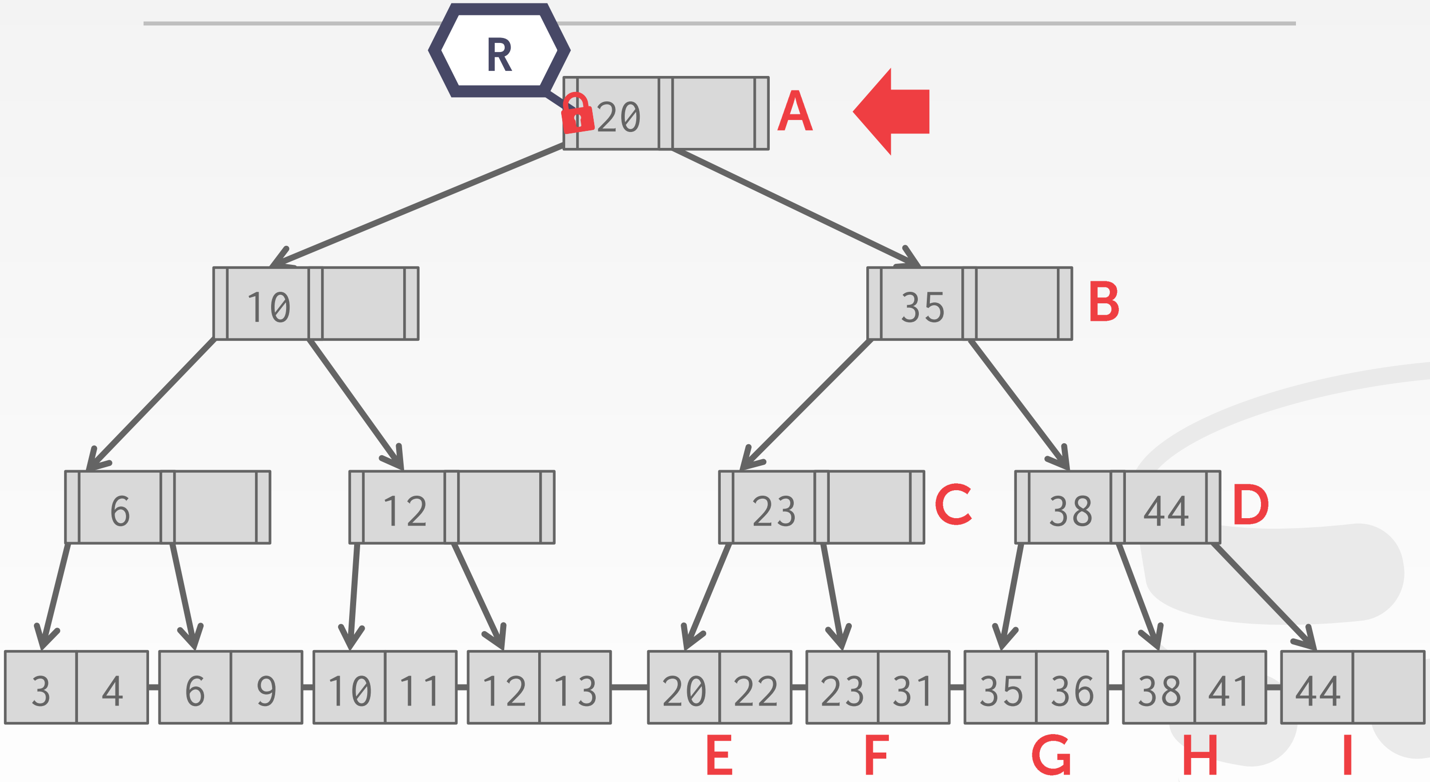
Insert/Delete: Start at root and go down, obtaining **W** latches as needed. Once child is latched, check if it is safe:

→ If child is safe, release all latches on ancestors

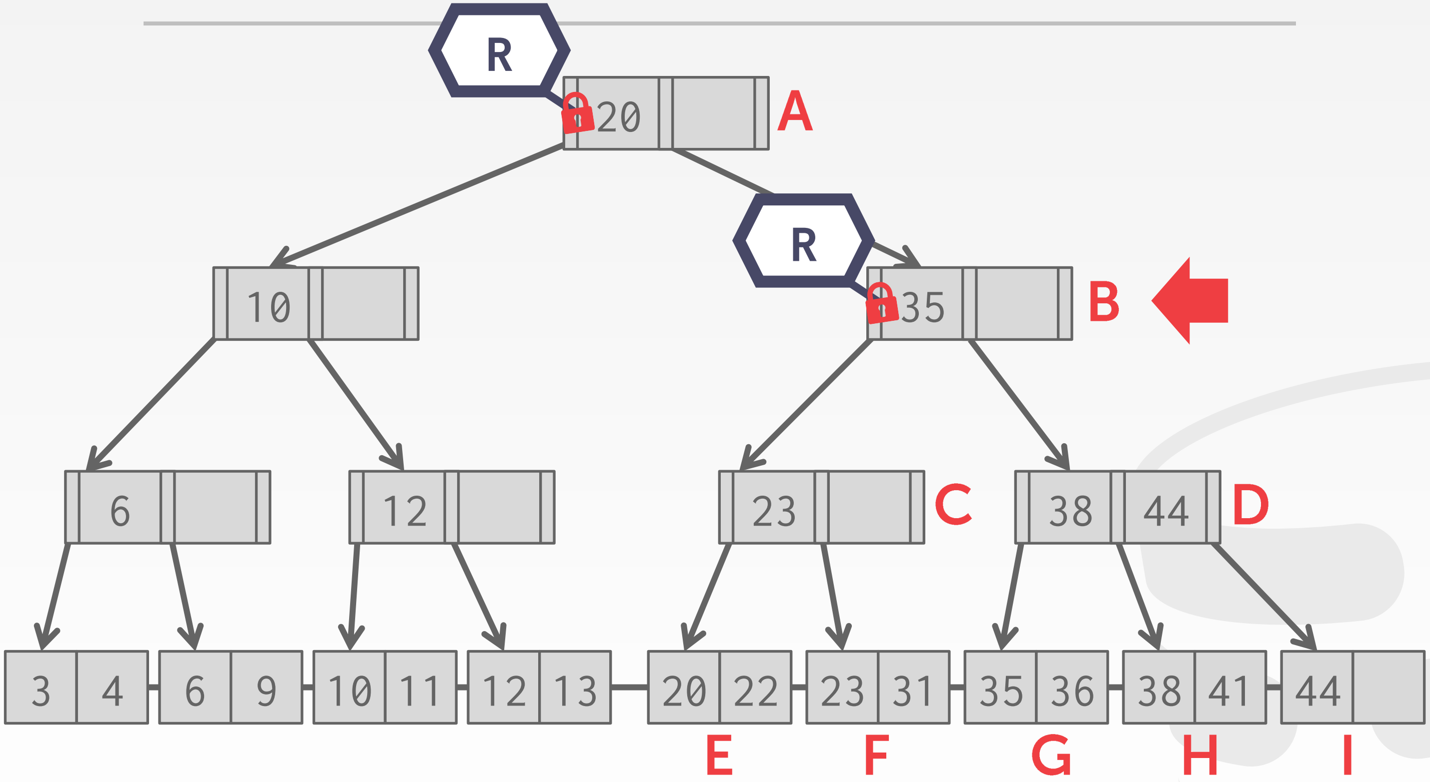
EXAMPLE #1 – FIND 38



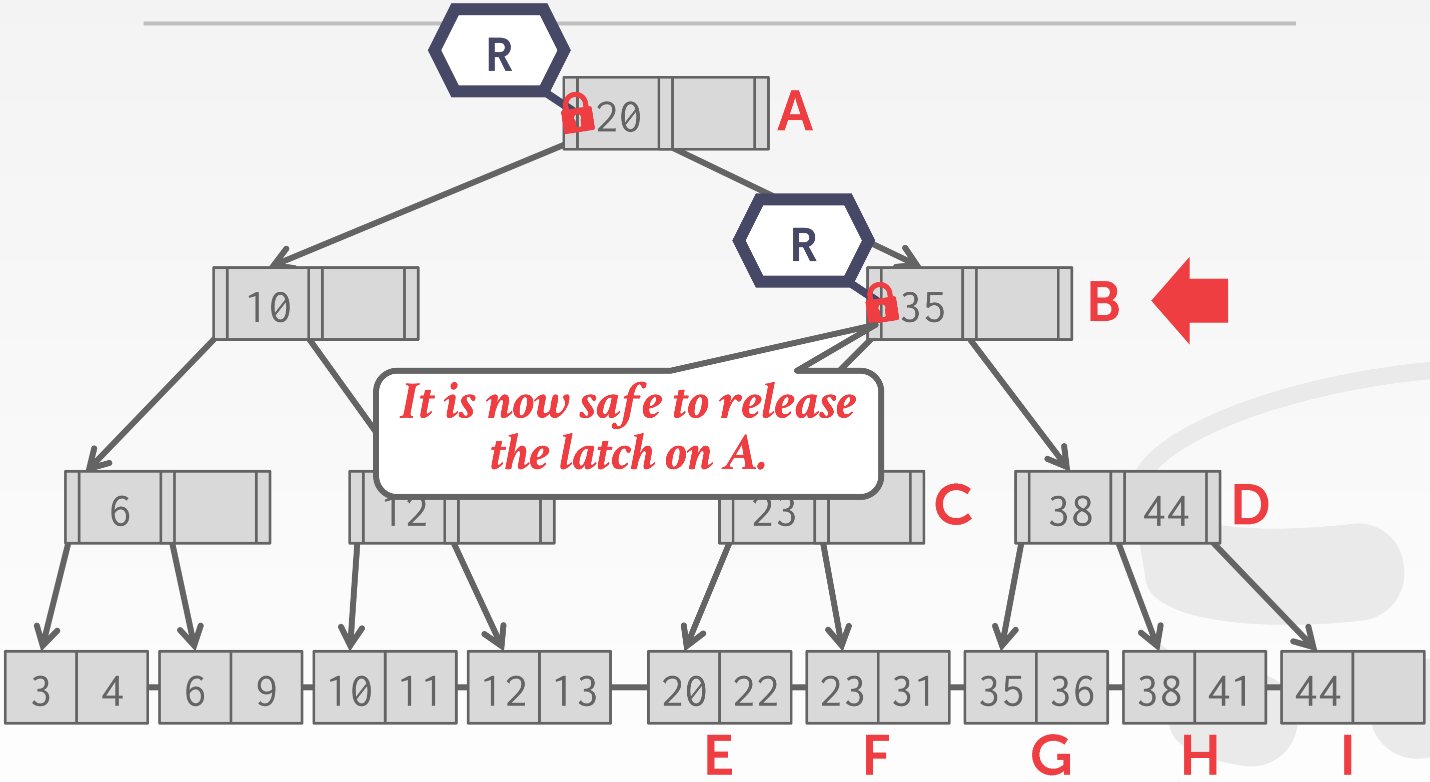
EXAMPLE #1 – FIND 38



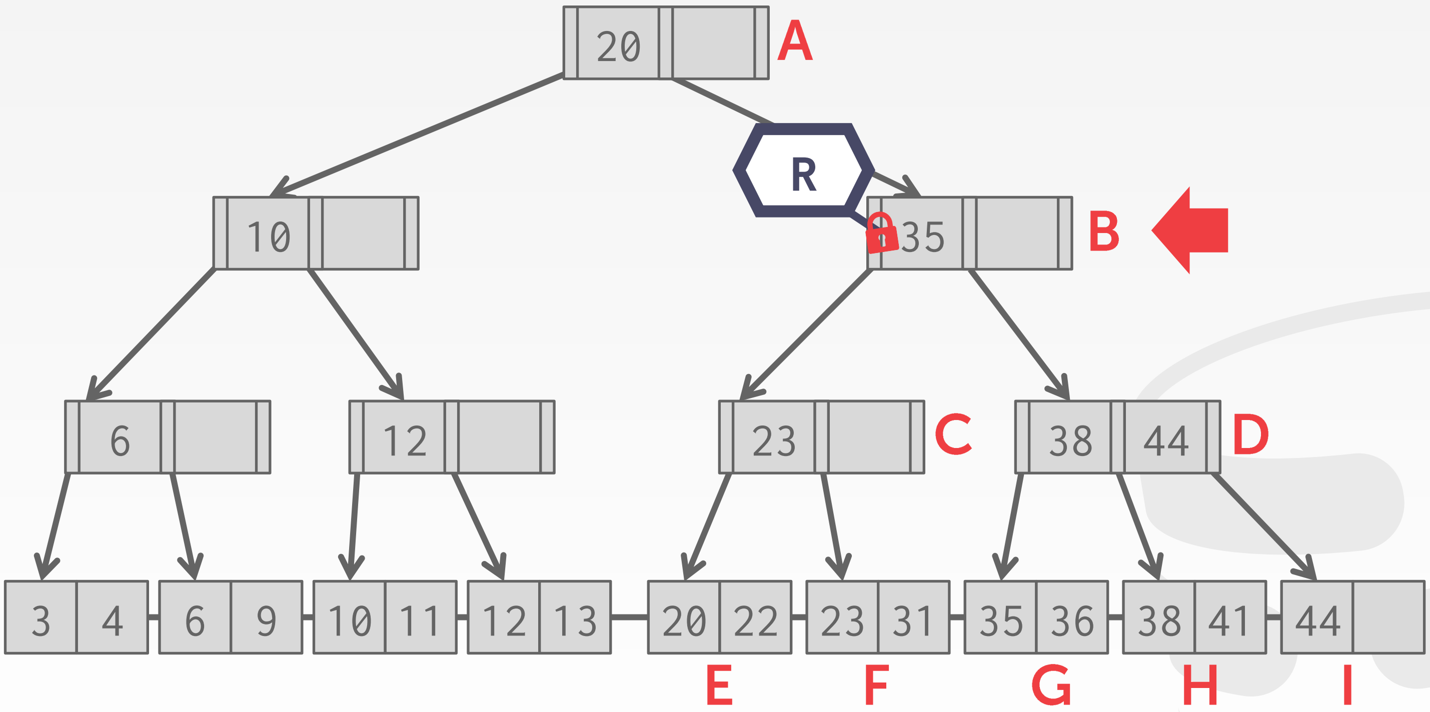
EXAMPLE #1 – FIND 38



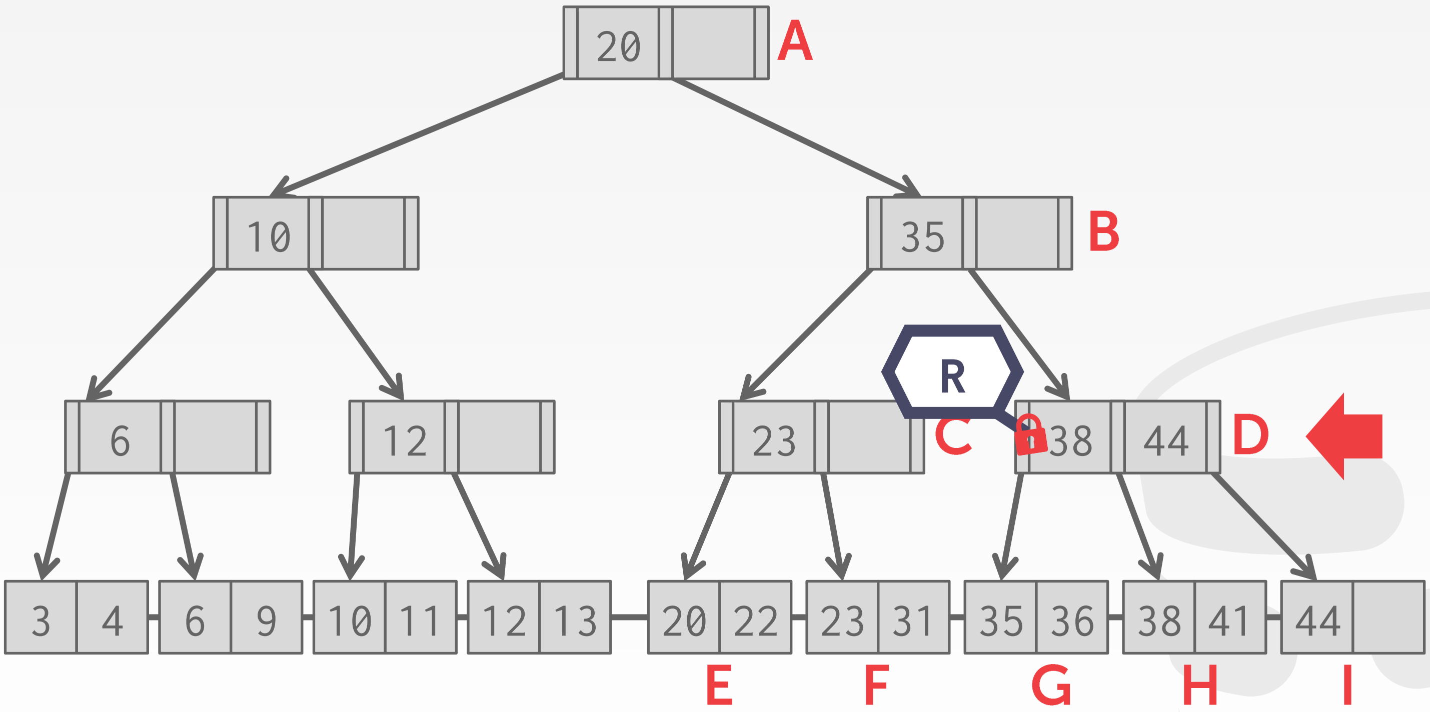
EXAMPLE #1 – FIND 38



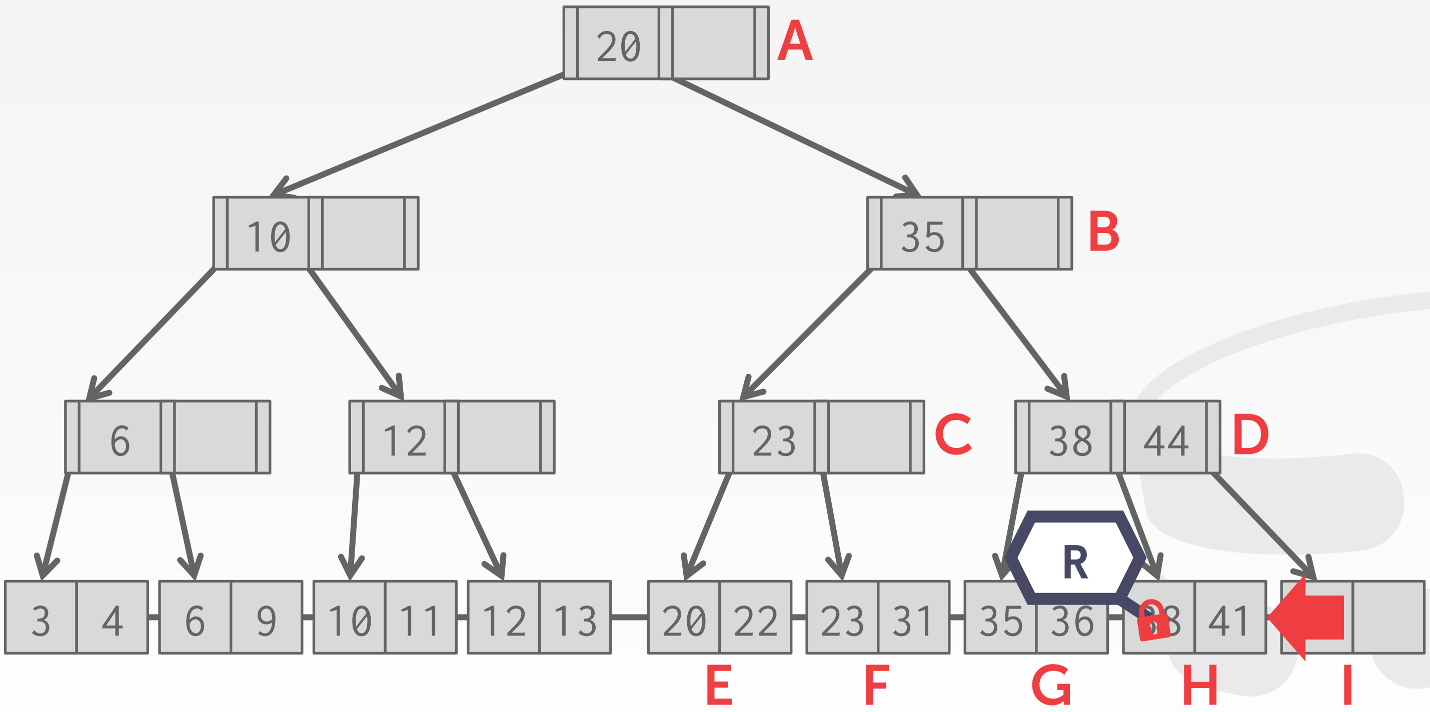
EXAMPLE #1 – FIND 38



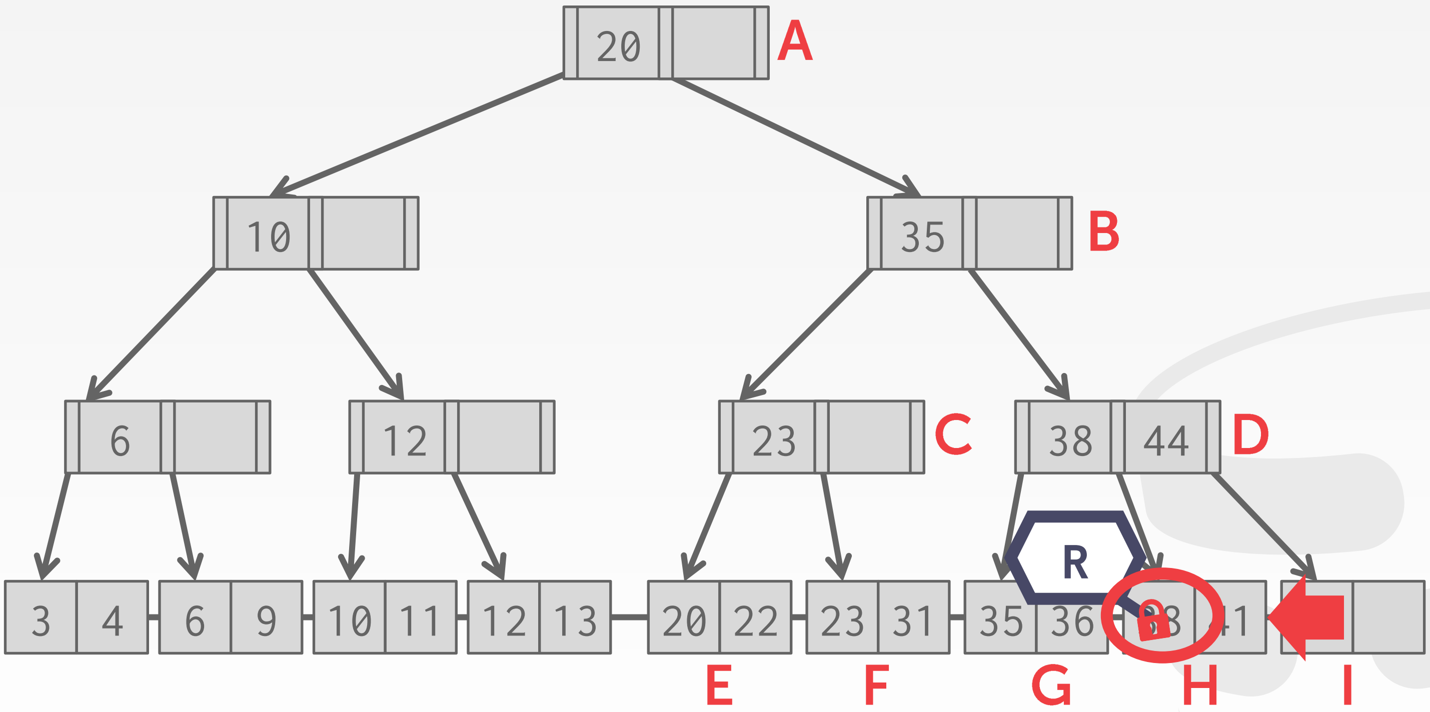
EXAMPLE #1 – FIND 38



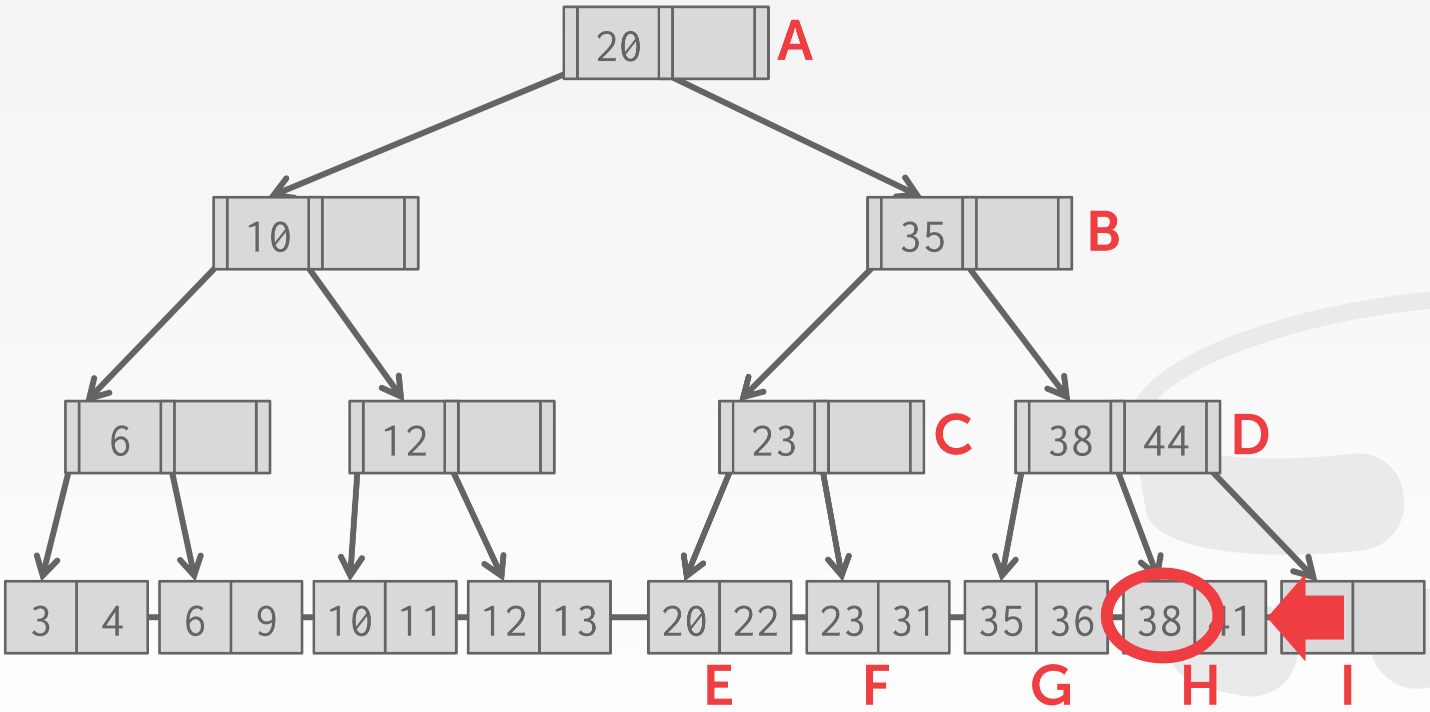
EXAMPLE #1 – FIND 38



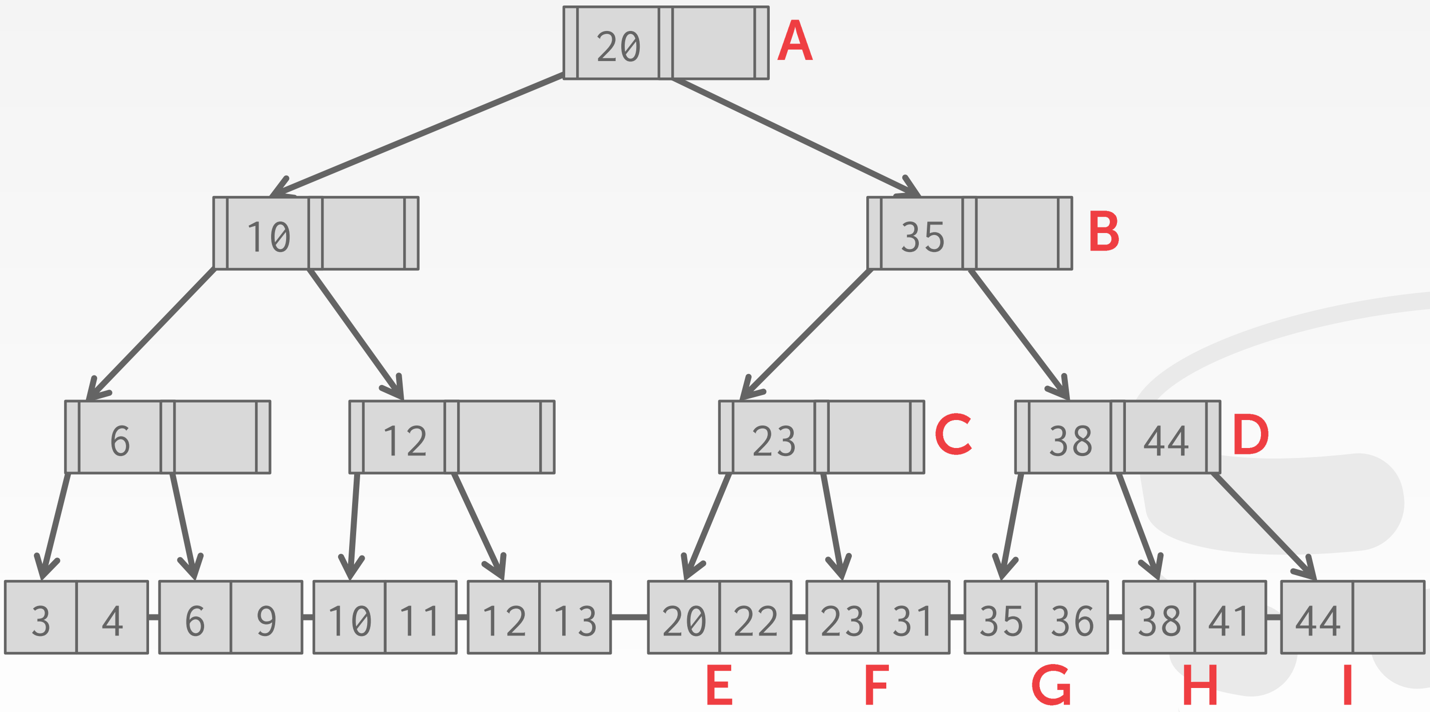
EXAMPLE #1 – FIND 38



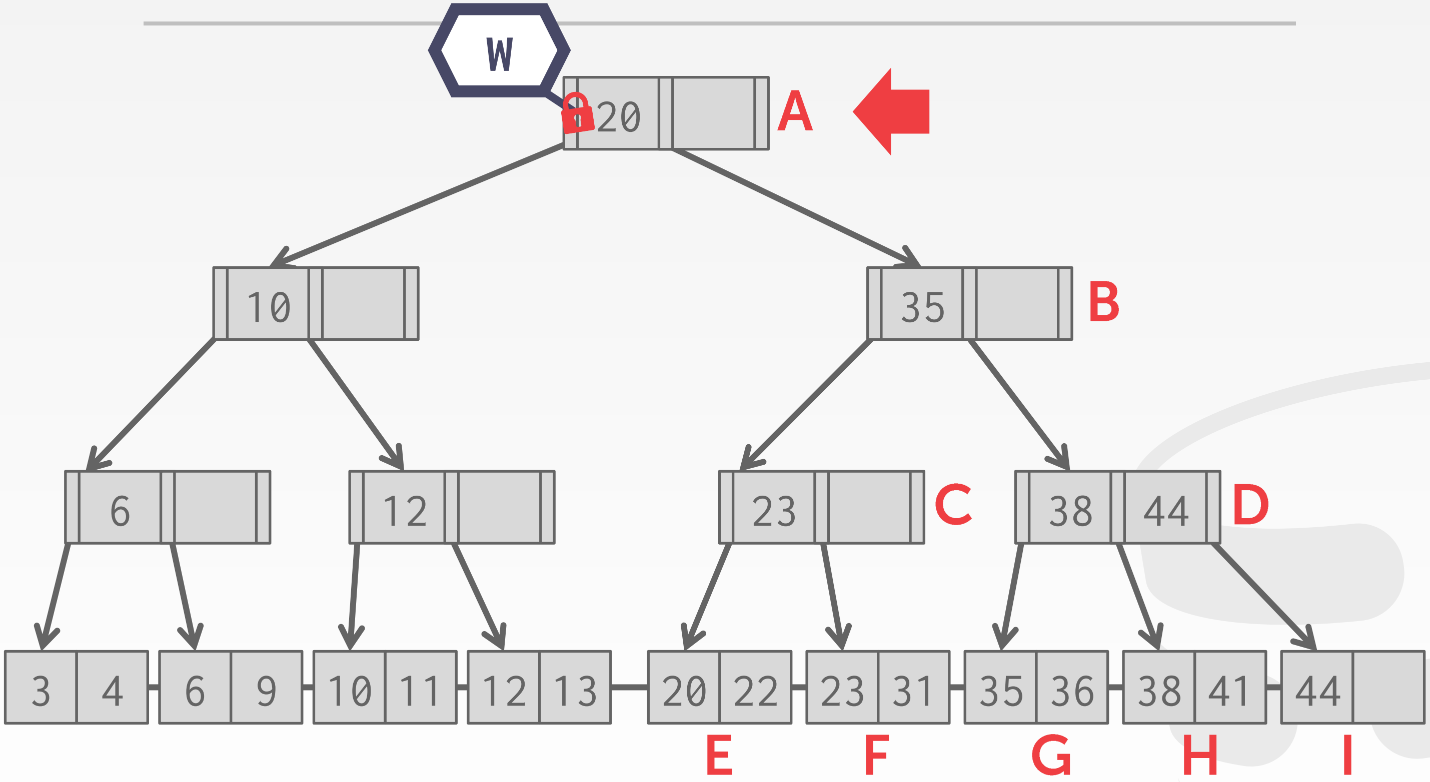
EXAMPLE #1 – FIND 38



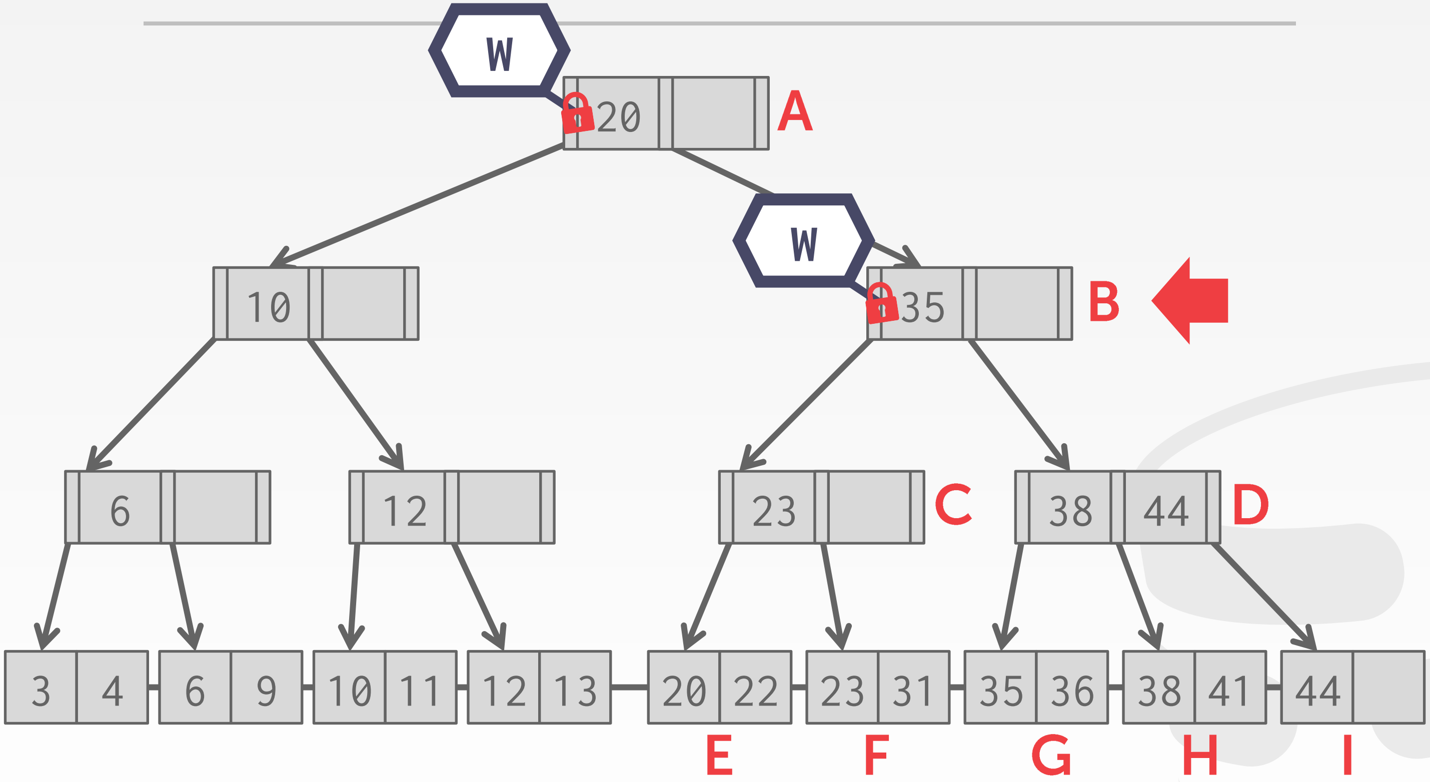
EXAMPLE #2 – DELETE 38



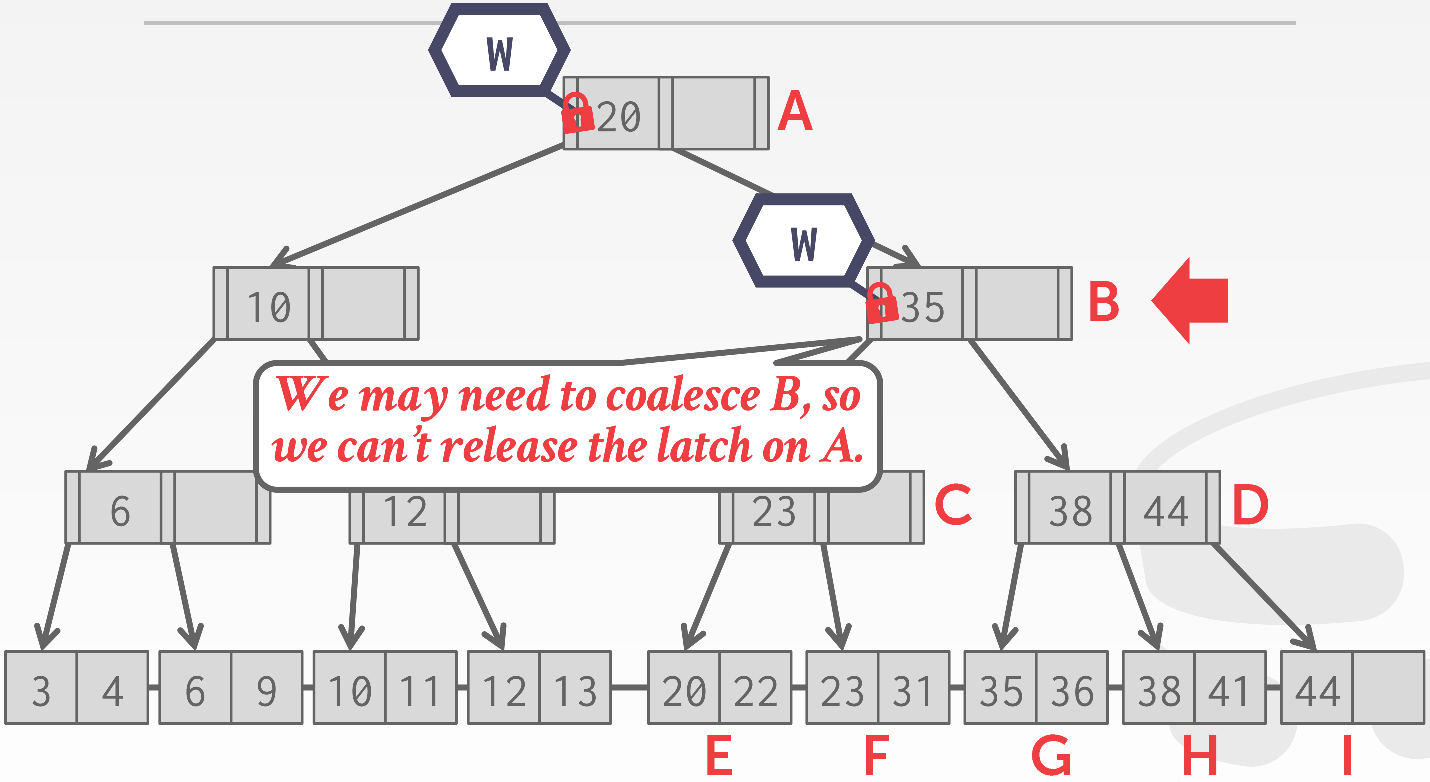
EXAMPLE #2 – DELETE 38



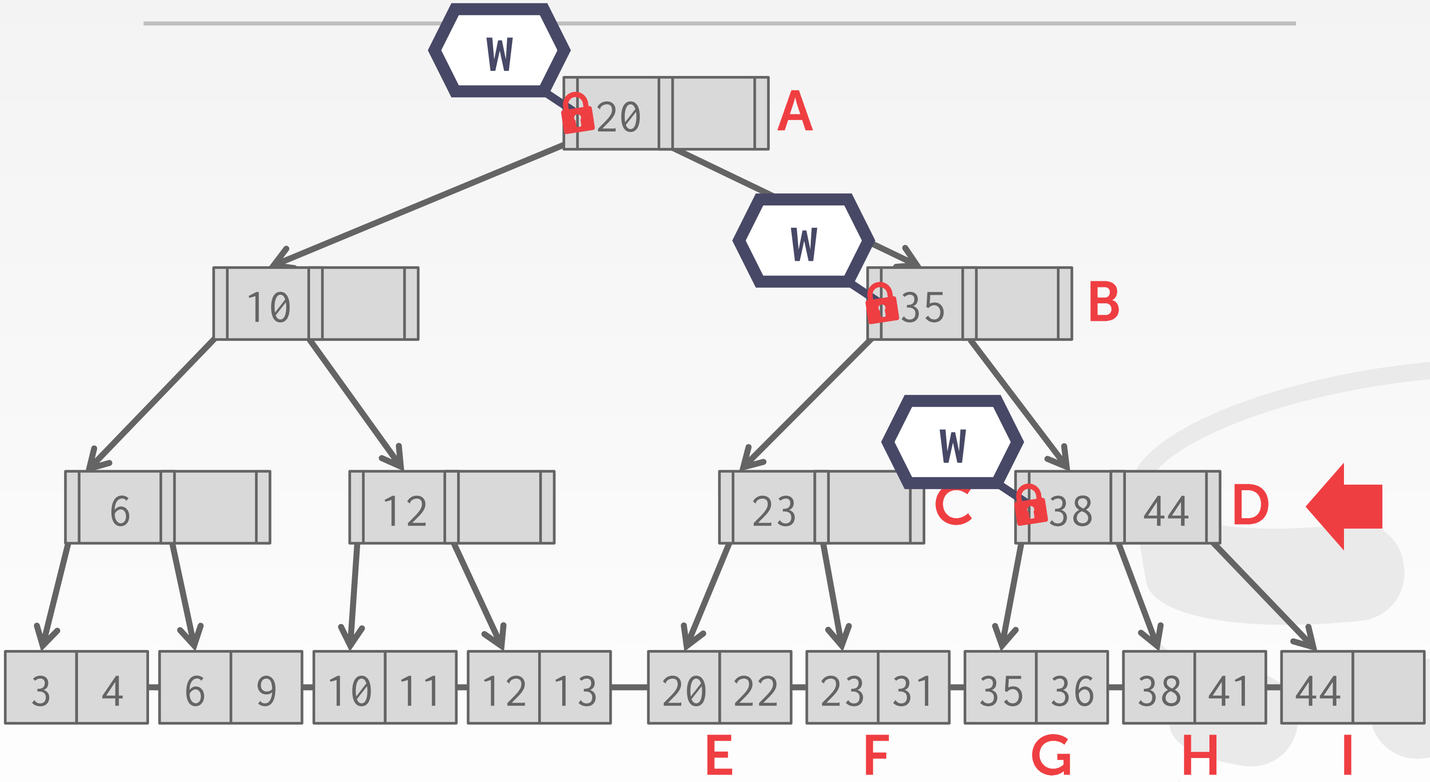
EXAMPLE #2 – DELETE 38



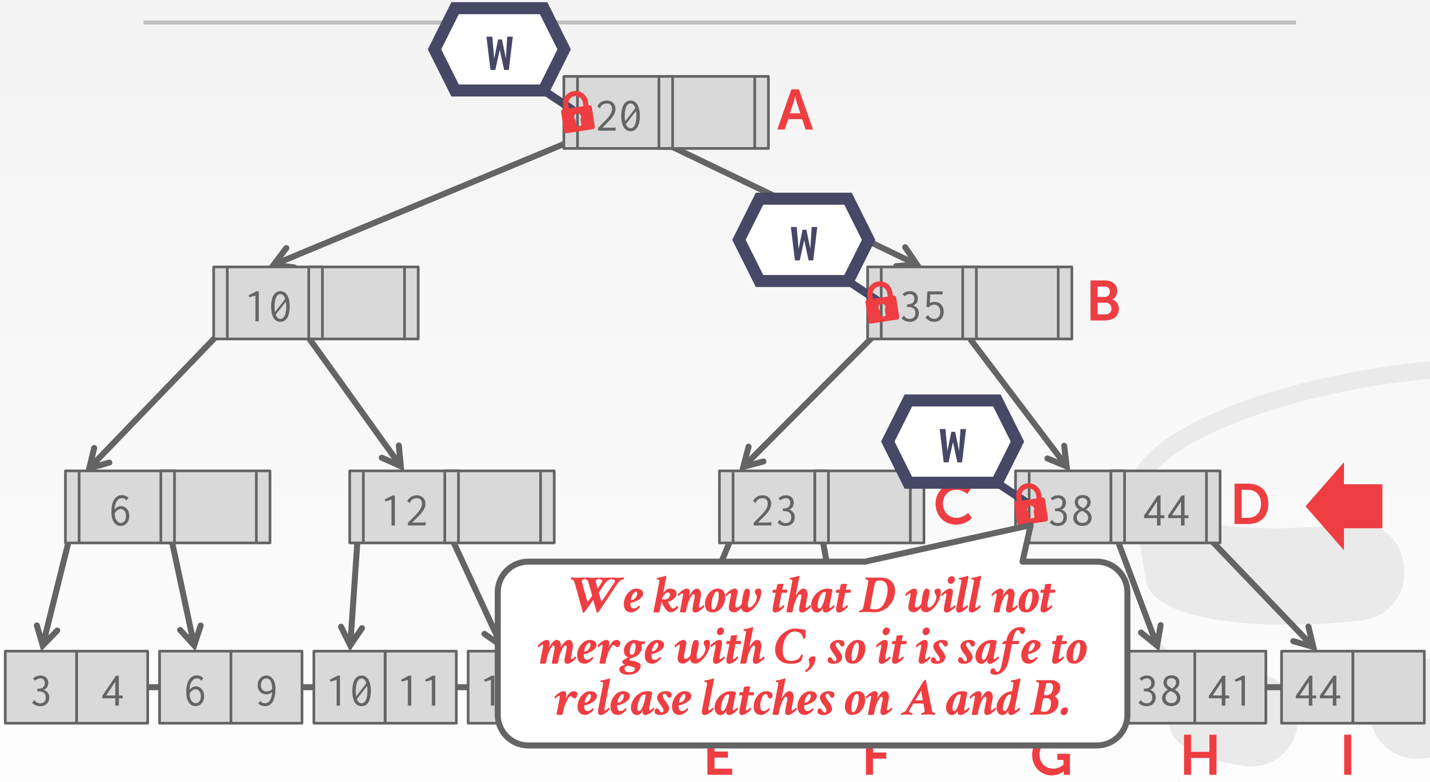
EXAMPLE #2 – DELETE 38



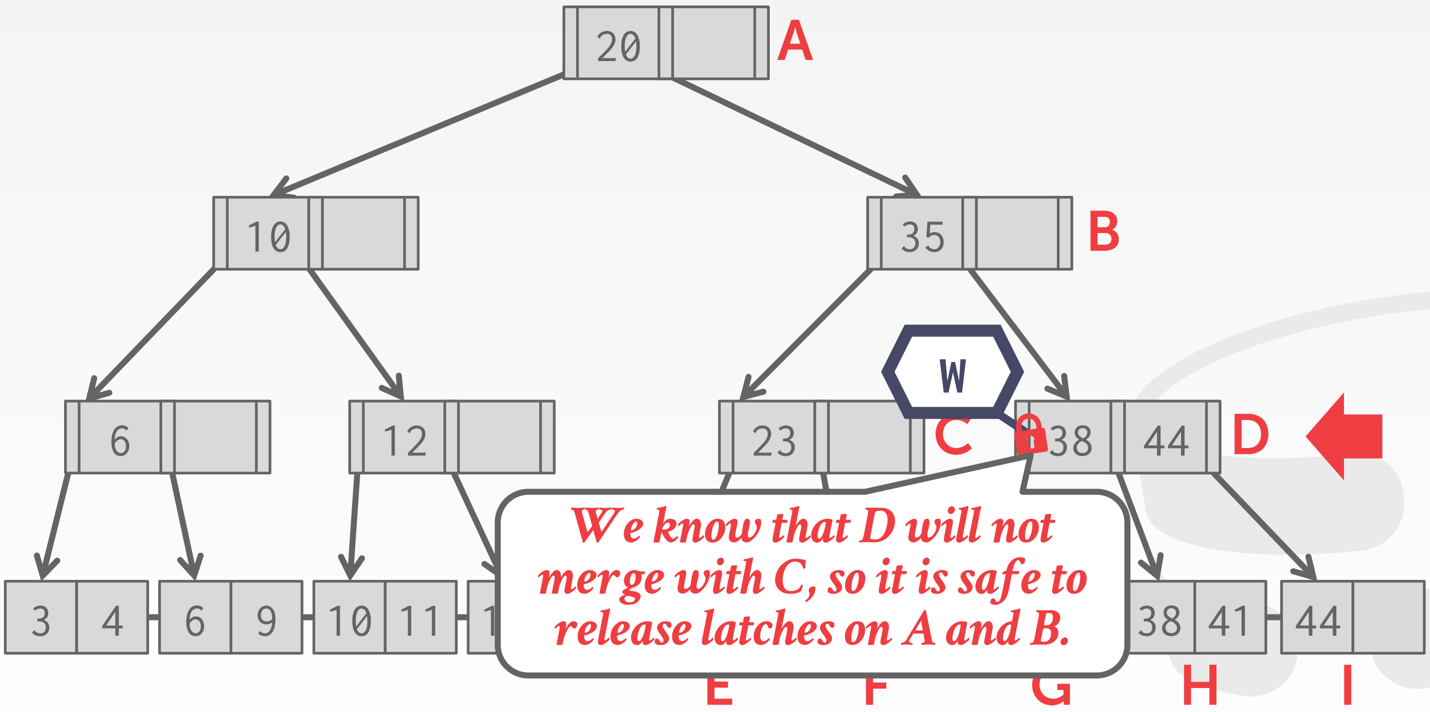
EXAMPLE #2 – DELETE 38



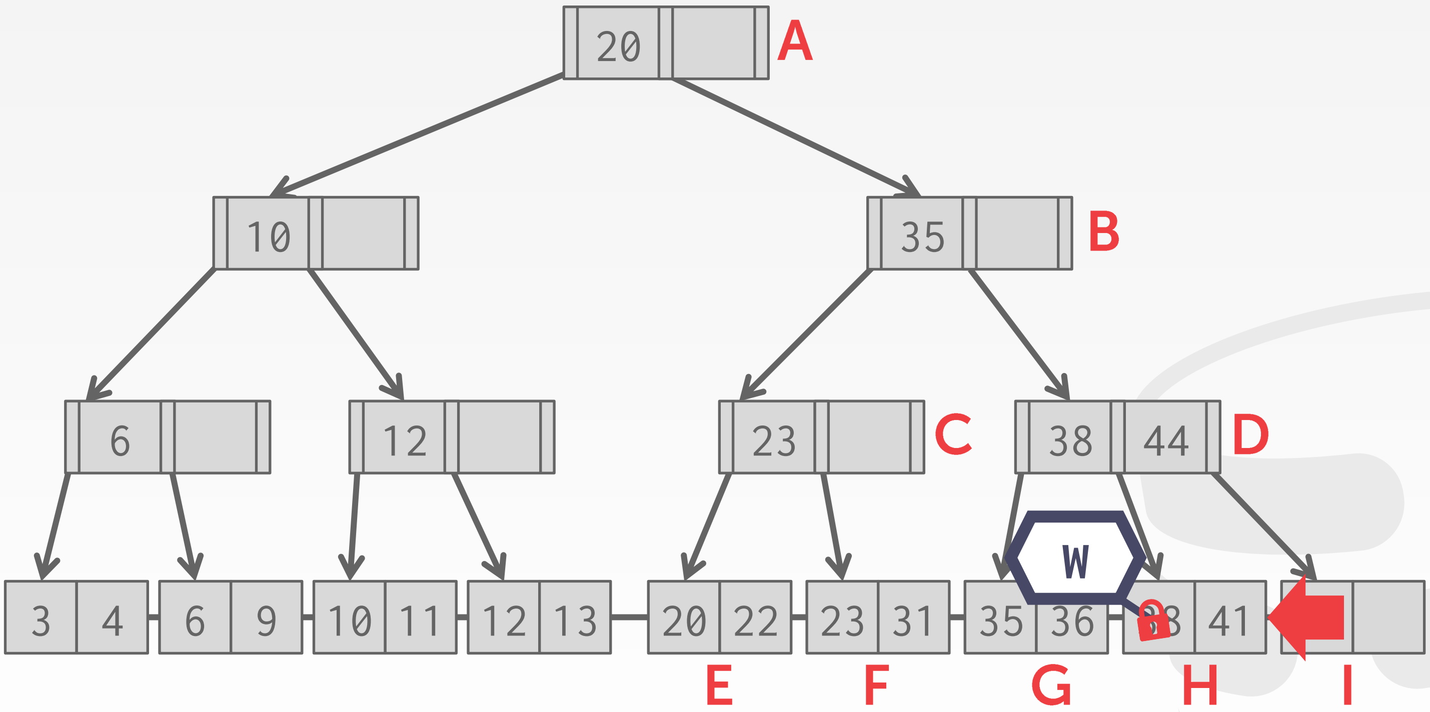
EXAMPLE #2 – DELETE 38



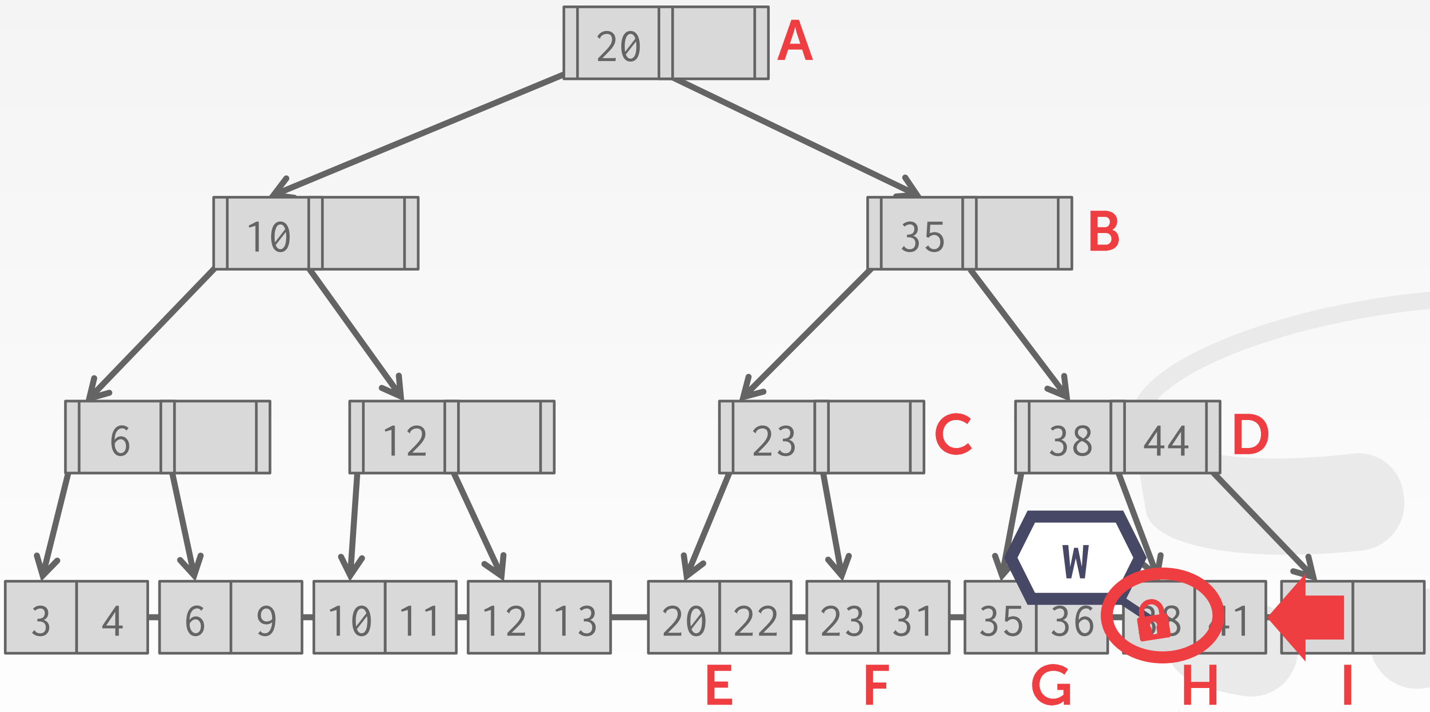
EXAMPLE #2 – DELETE 38



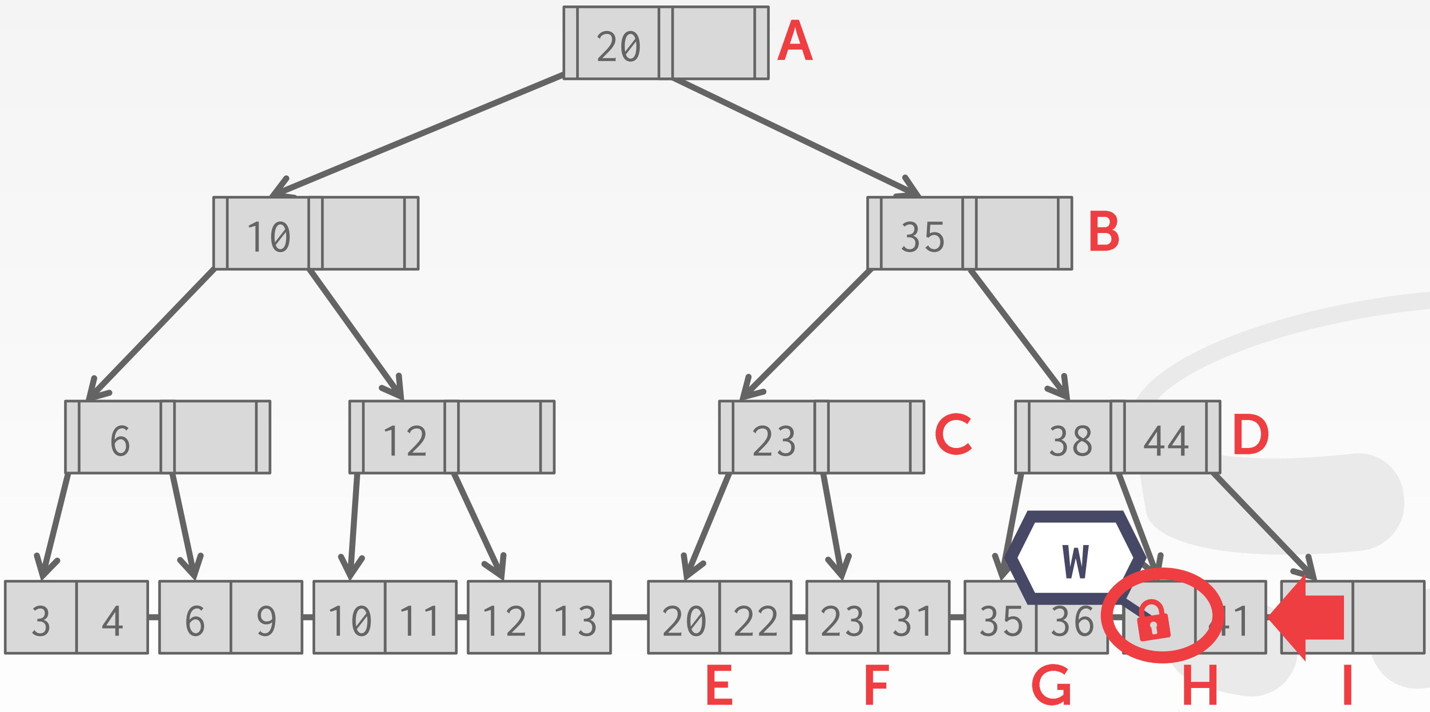
EXAMPLE #2 – DELETE 38



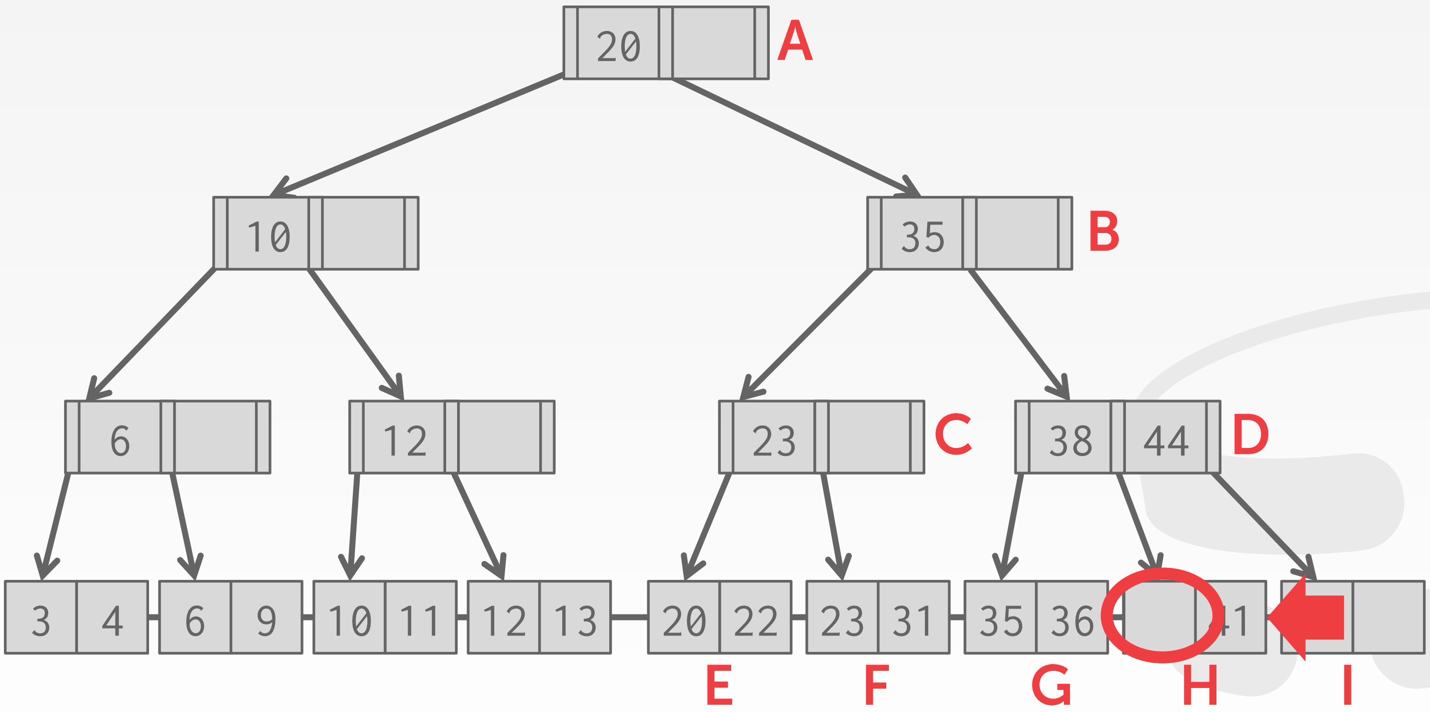
EXAMPLE #2 – DELETE 38



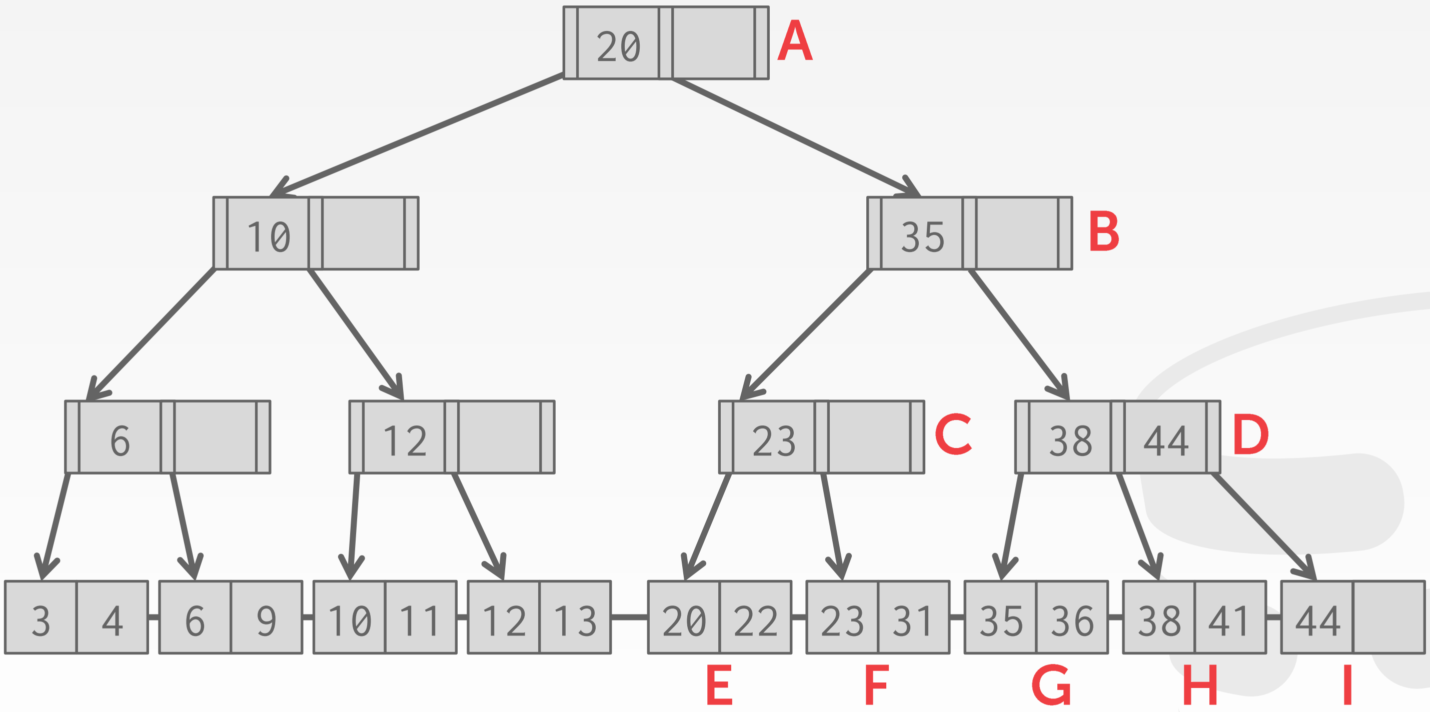
EXAMPLE #2 – DELETE 38



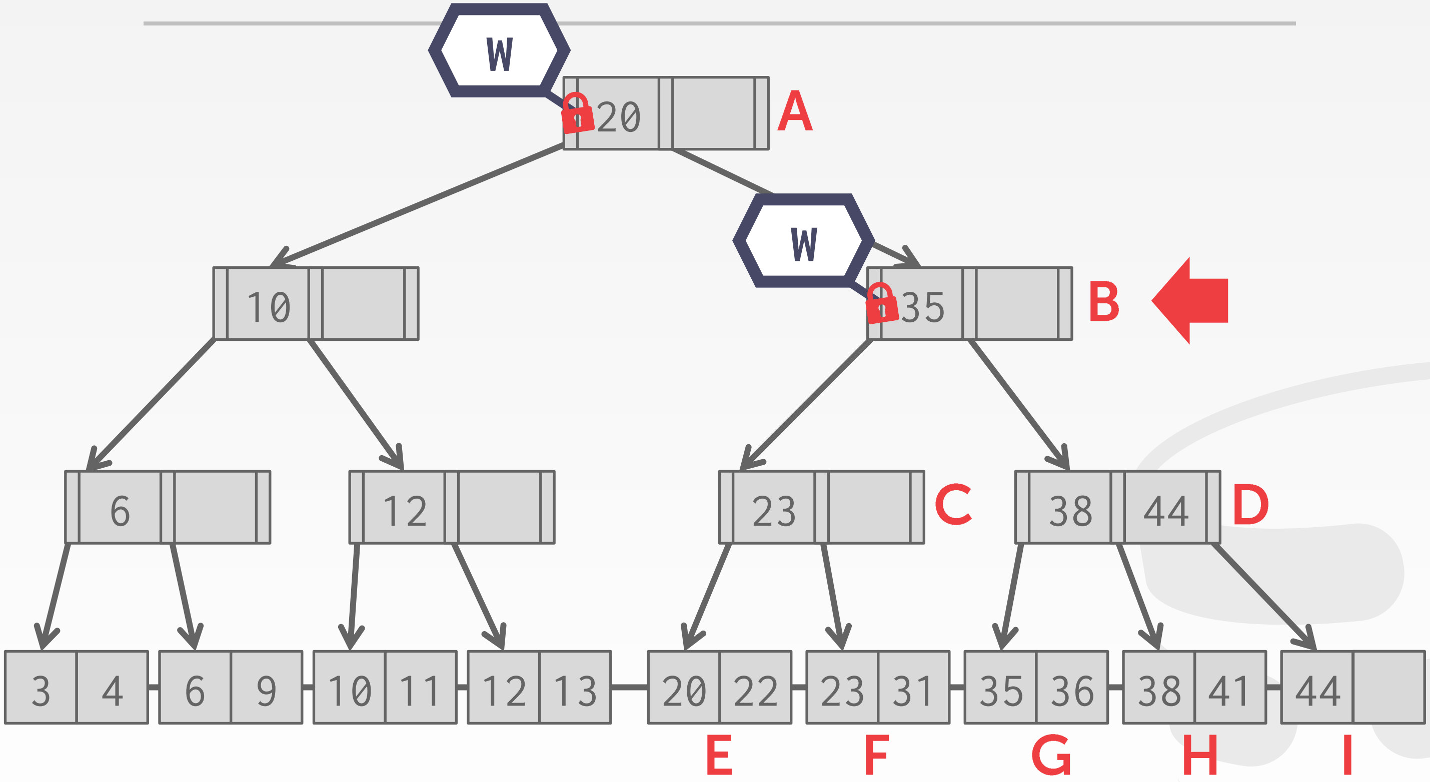
EXAMPLE #2 – DELETE 38



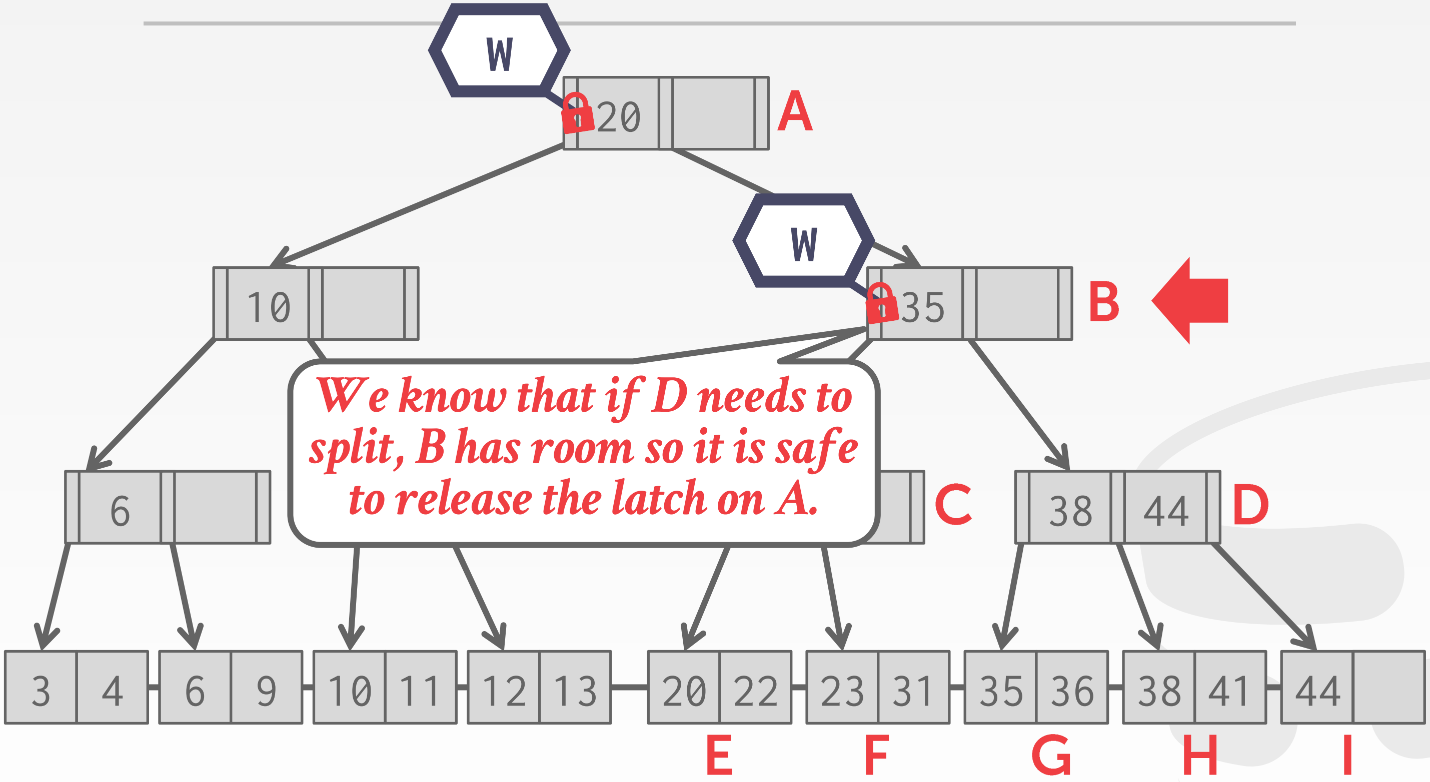
EXAMPLE #3 – INSERT 45



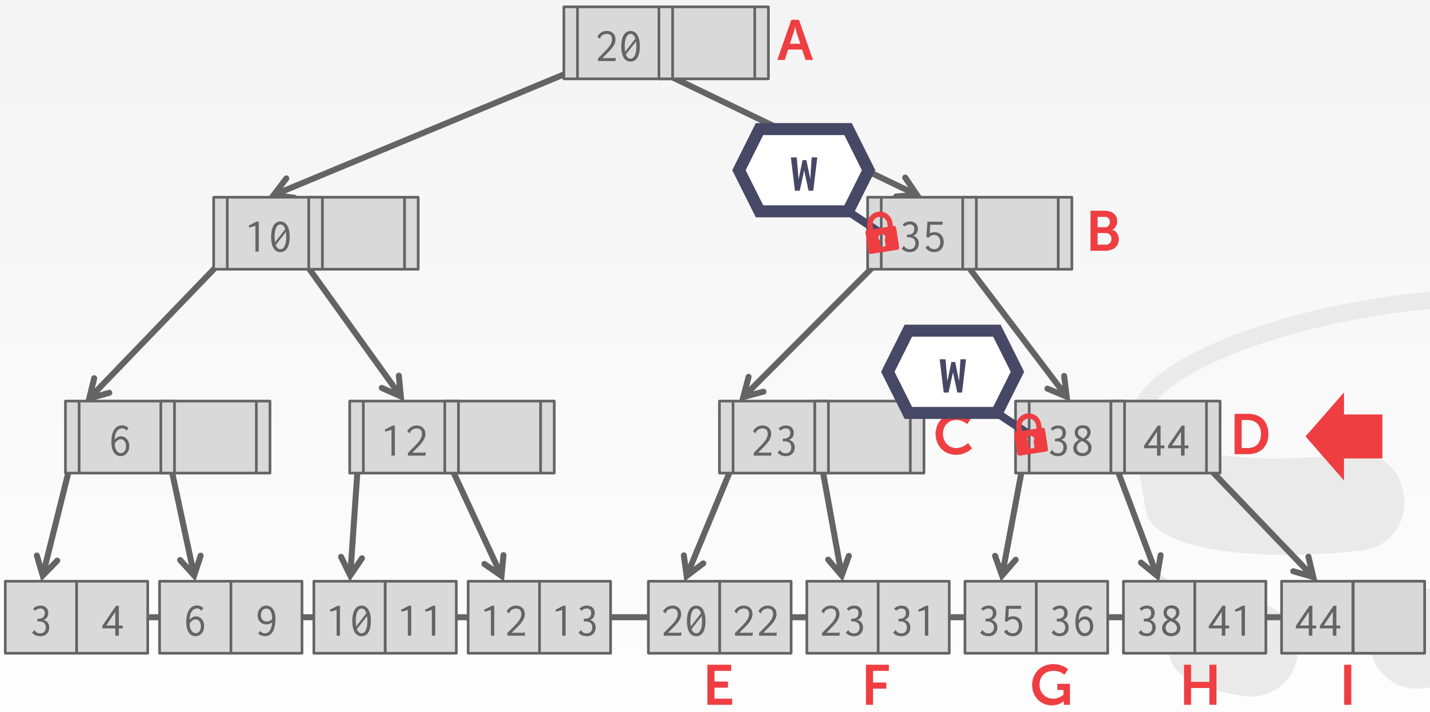
EXAMPLE #3 – INSERT 45



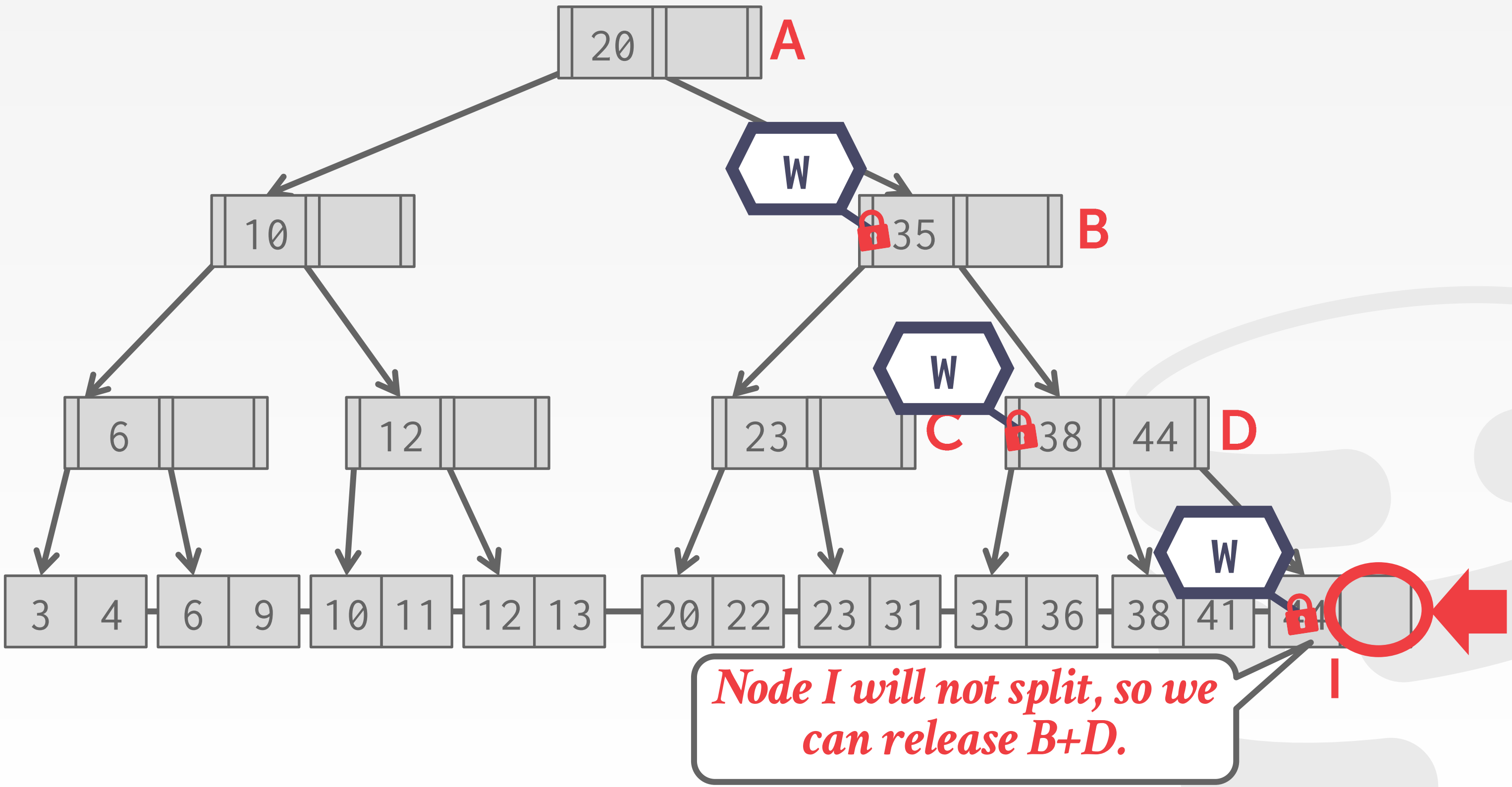
EXAMPLE #3 – INSERT 45



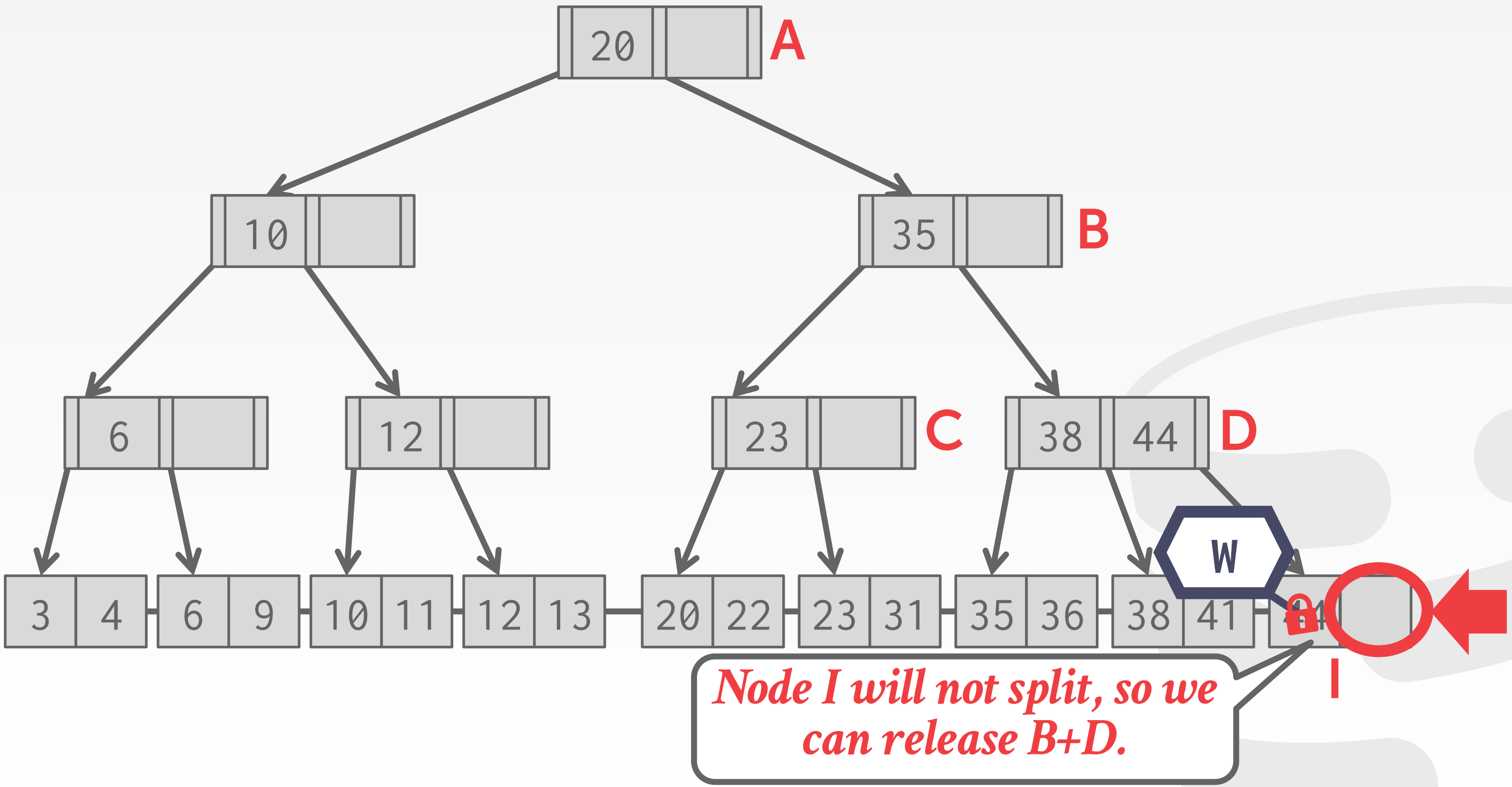
EXAMPLE #3 – INSERT 45



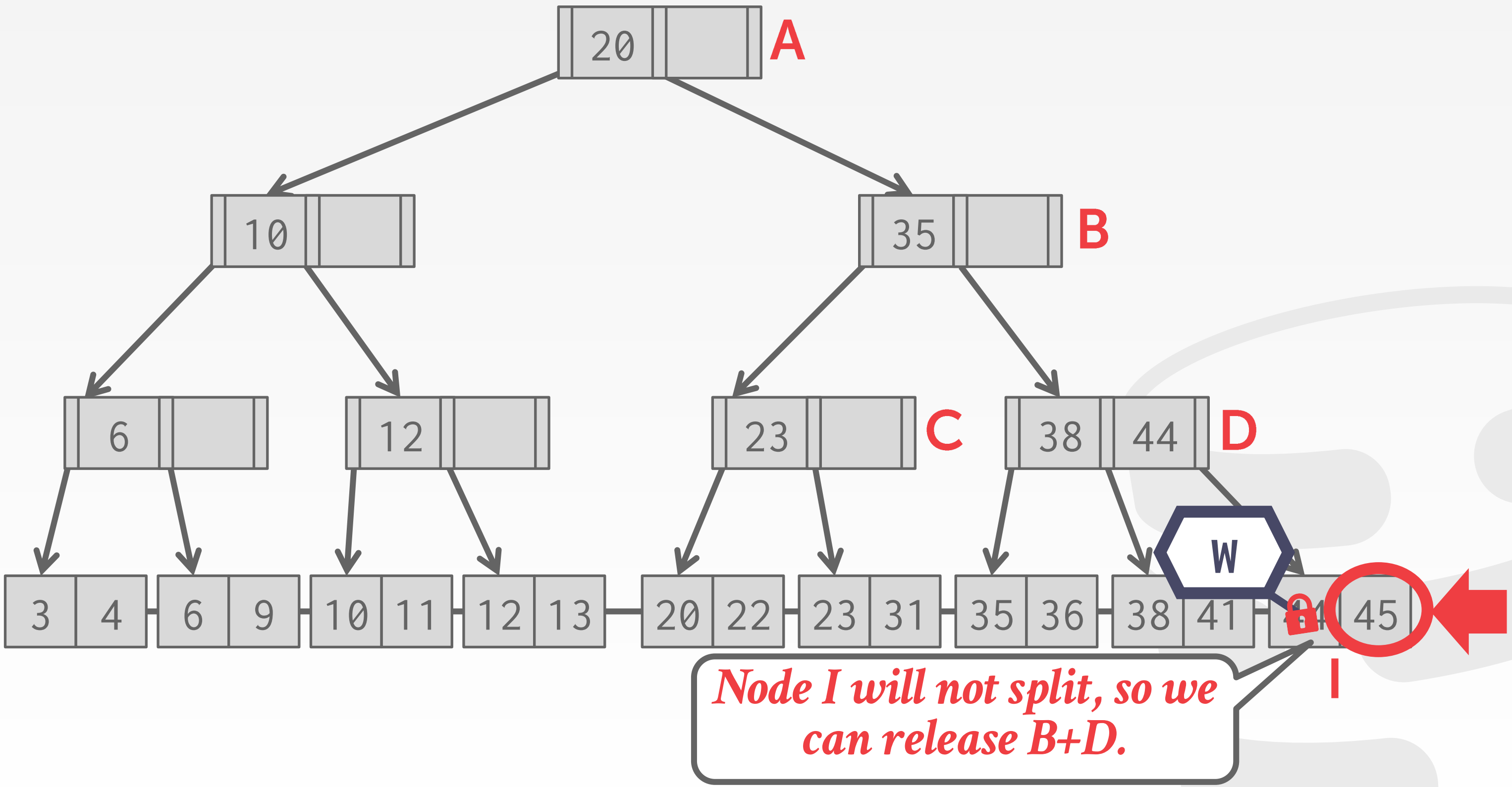
EXAMPLE #3 – INSERT 45



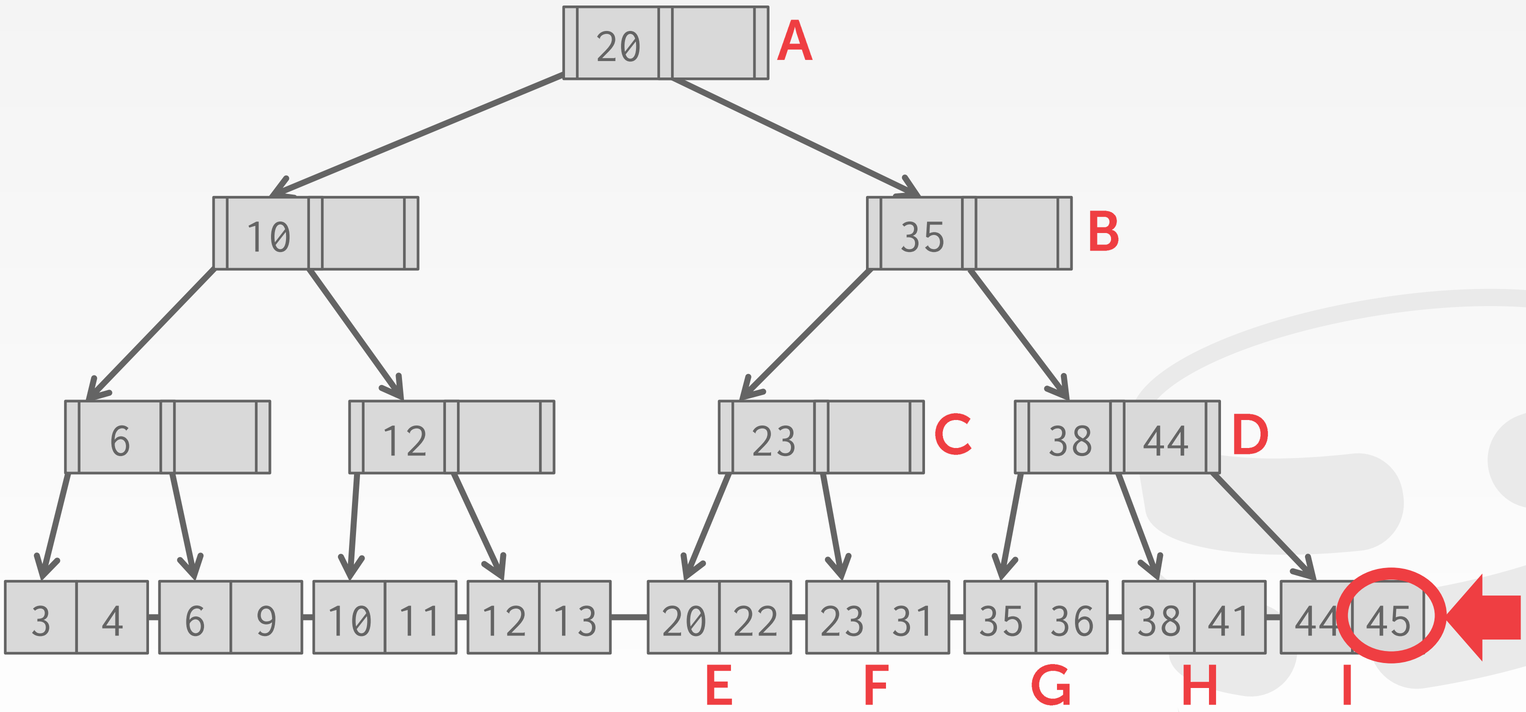
EXAMPLE #3 – INSERT 45



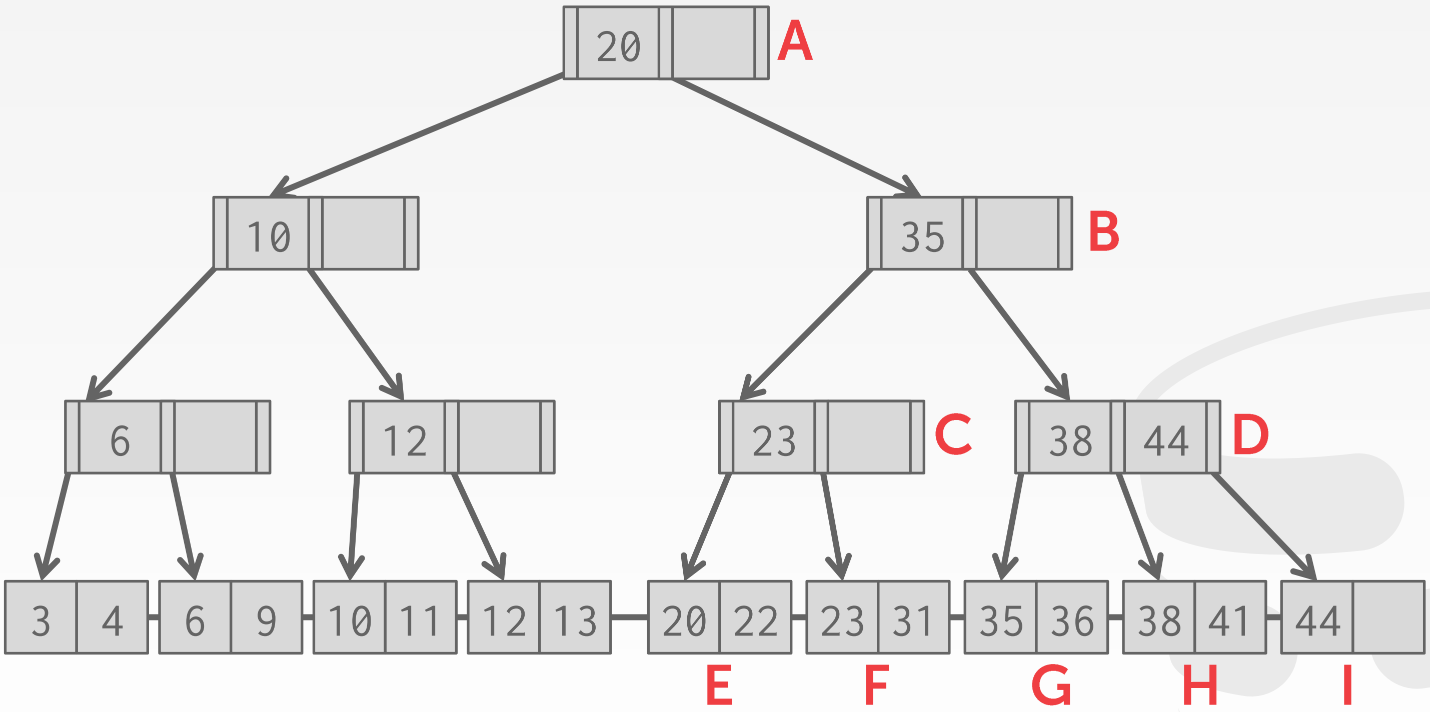
EXAMPLE #3 – INSERT 45



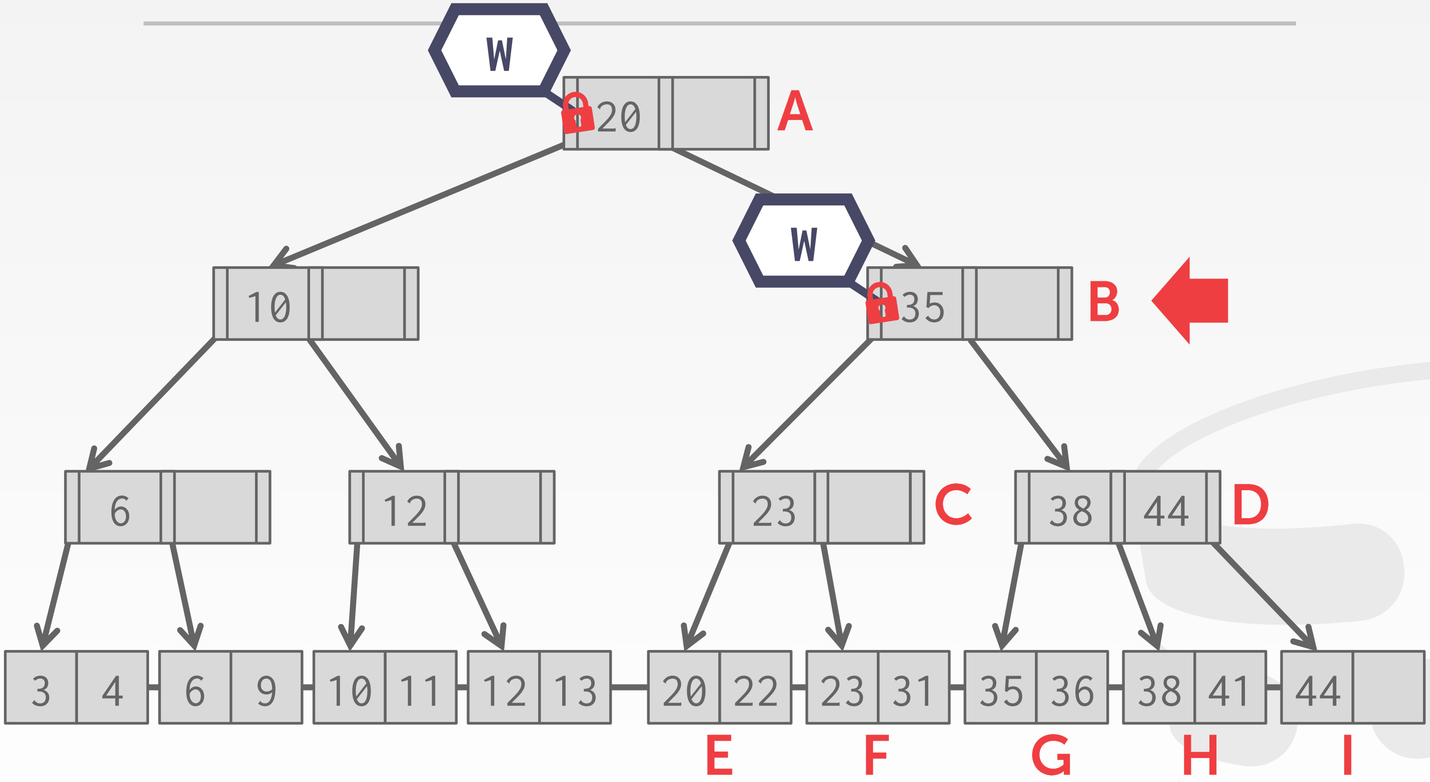
EXAMPLE #3 – INSERT 45



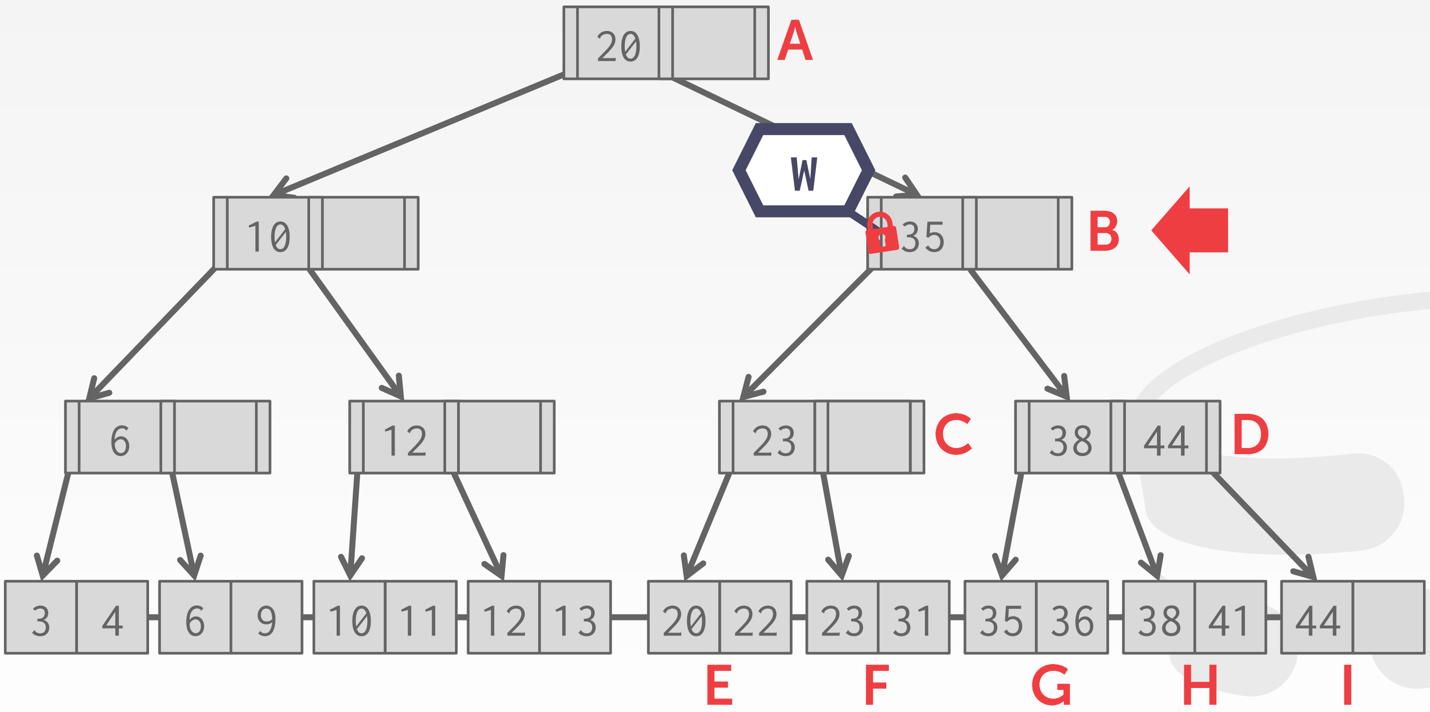
EXAMPLE #4 – INSERT 25



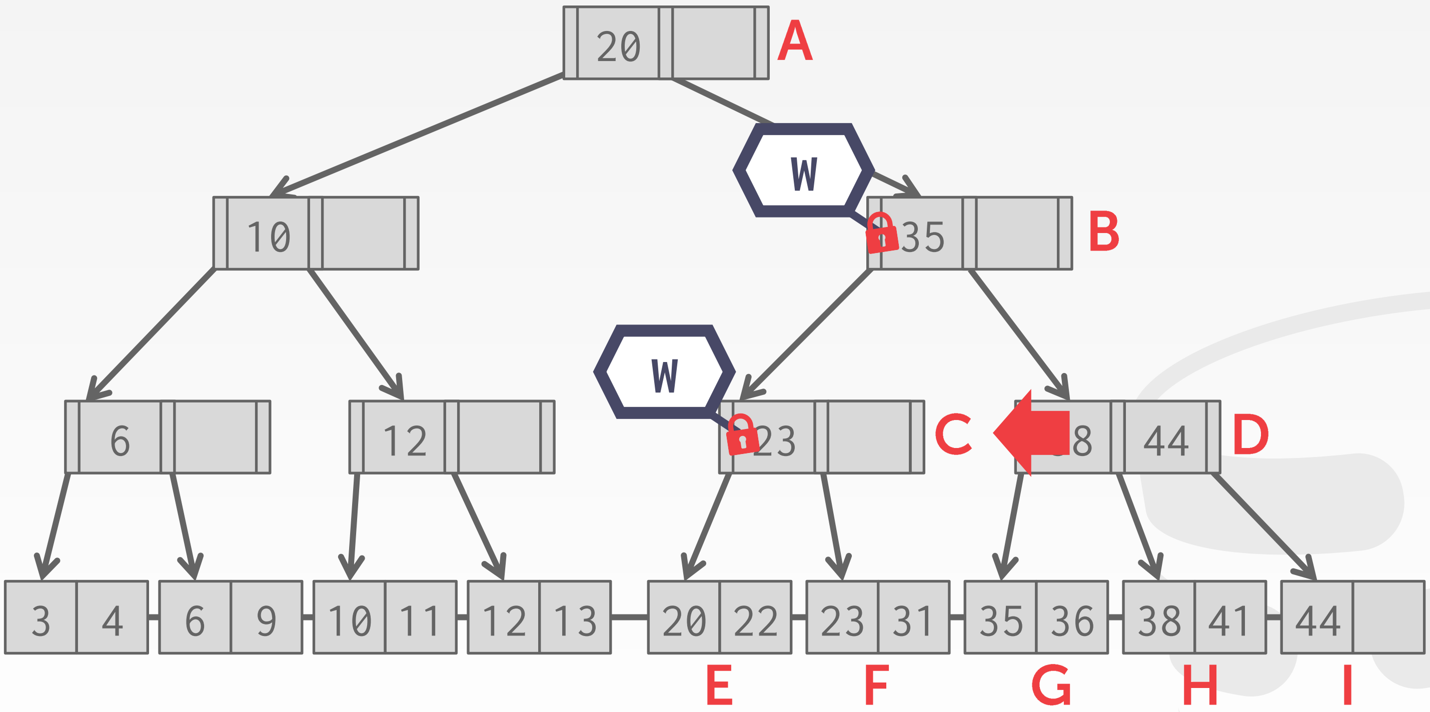
EXAMPLE #4 – INSERT 25



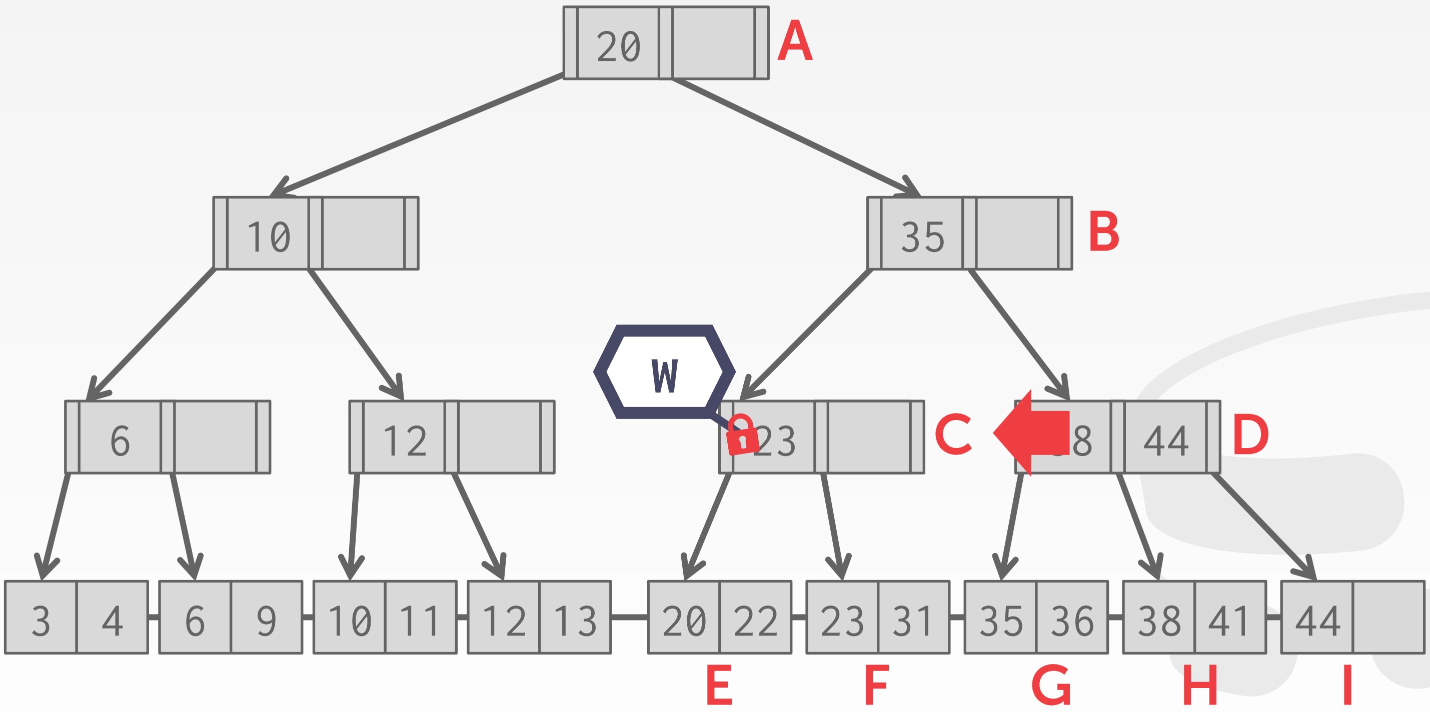
EXAMPLE #4 – INSERT 25



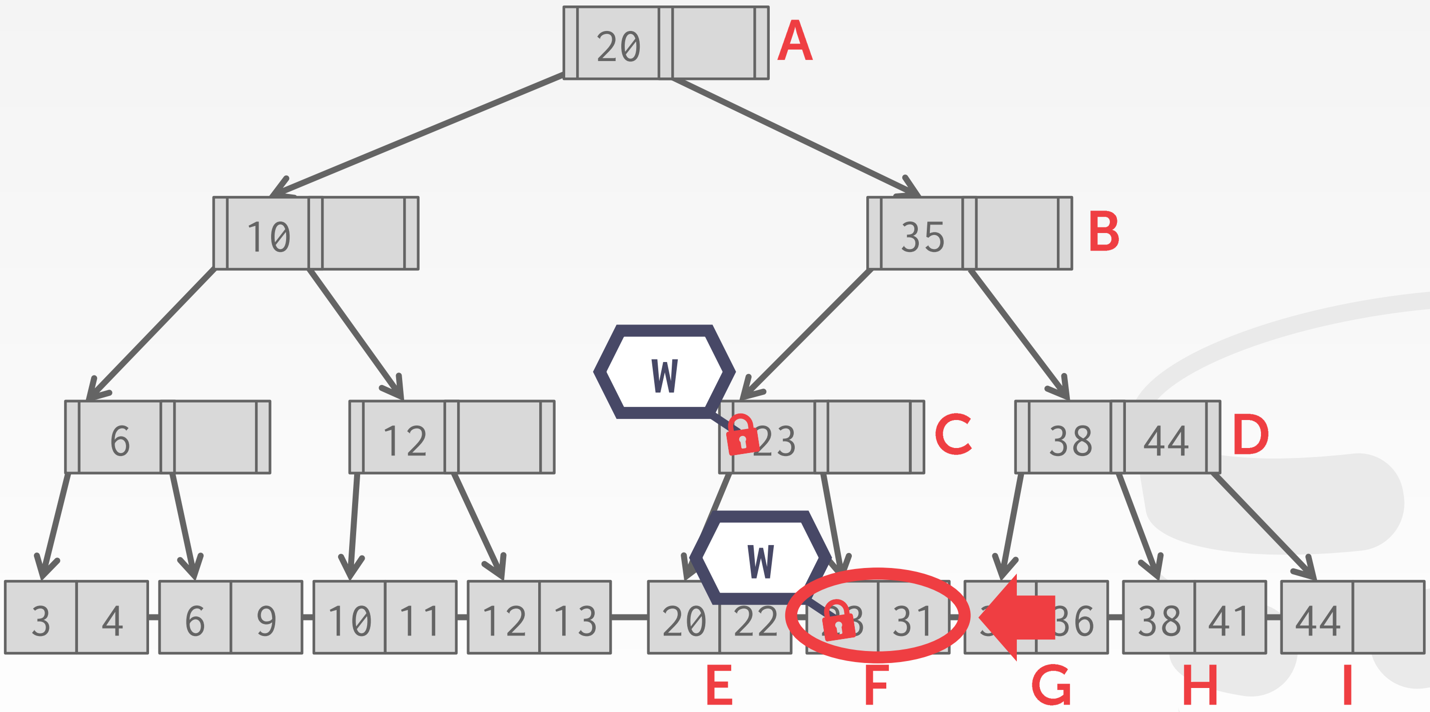
EXAMPLE #4 – INSERT 25



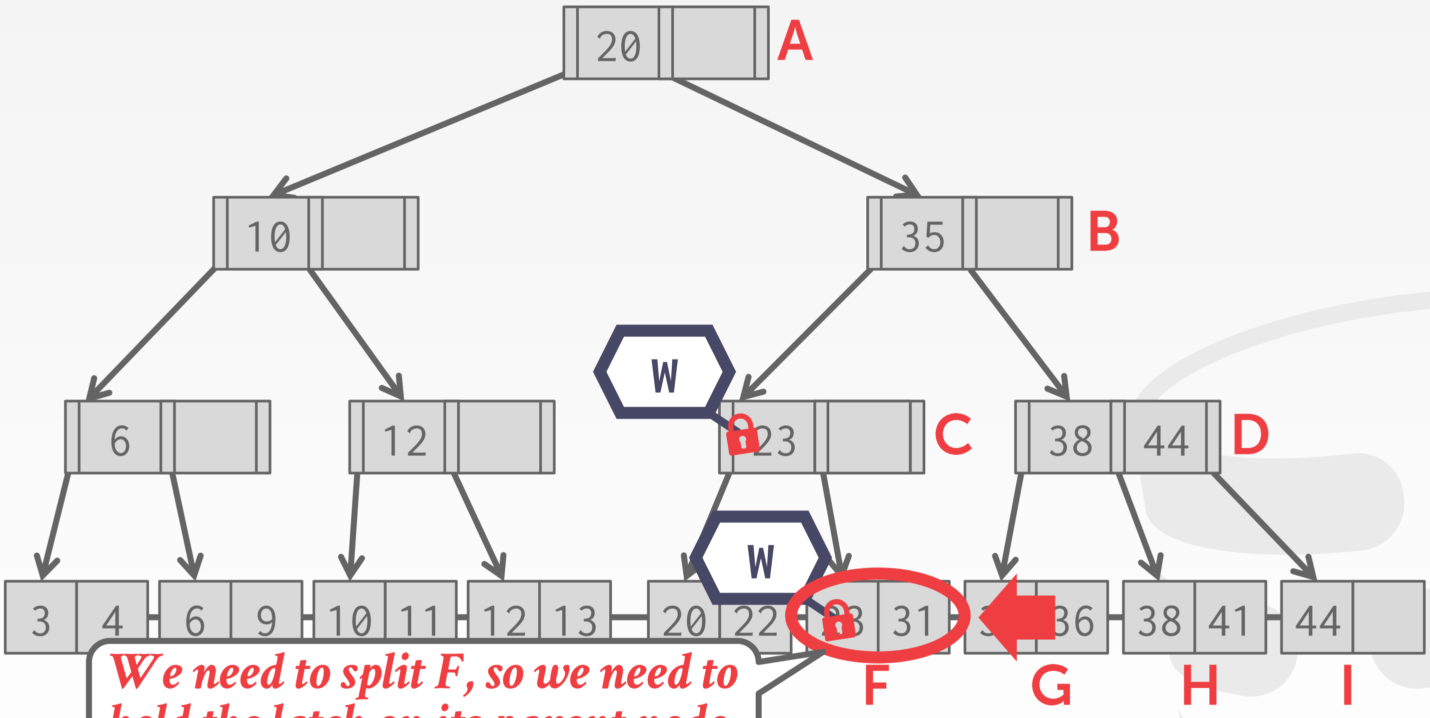
EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25

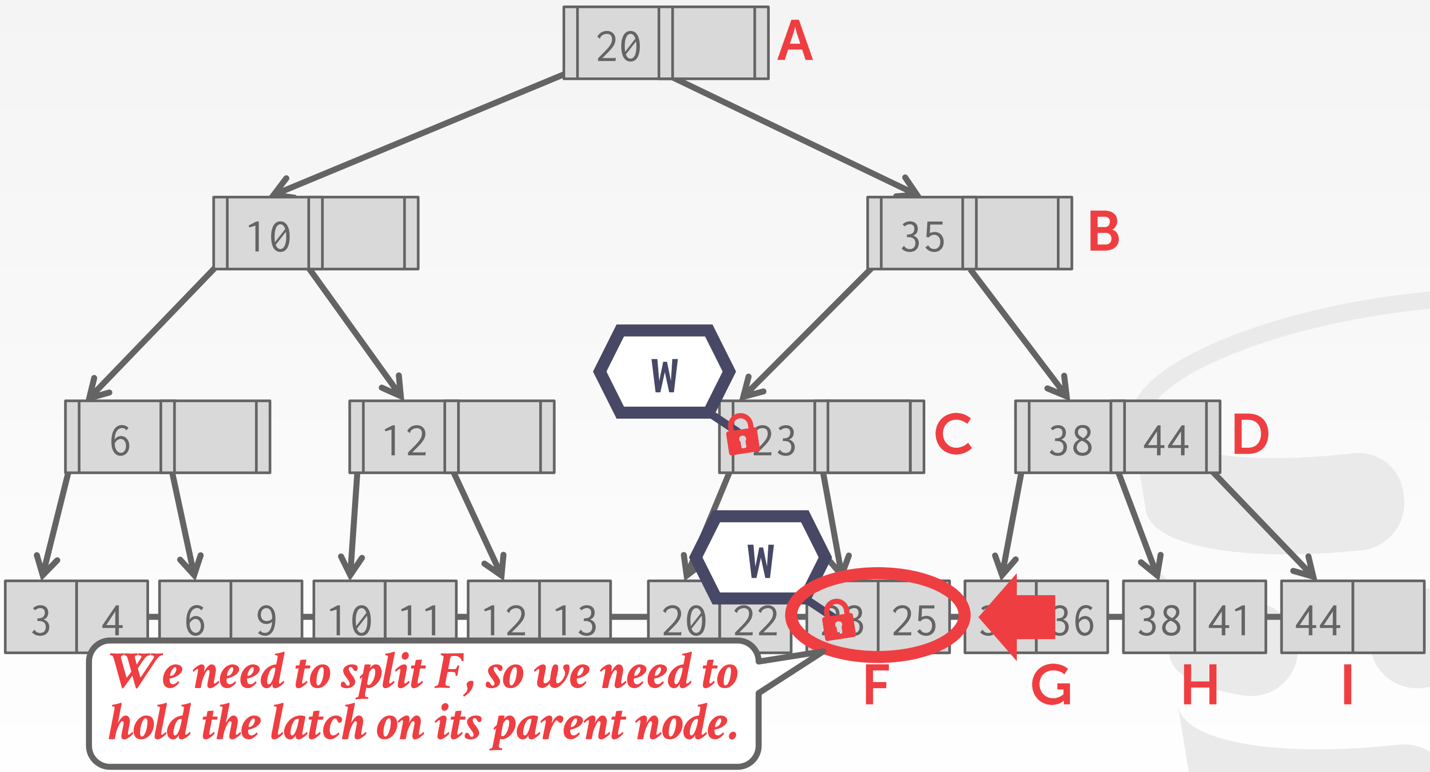


EXAMPLE #4 – INSERT 25

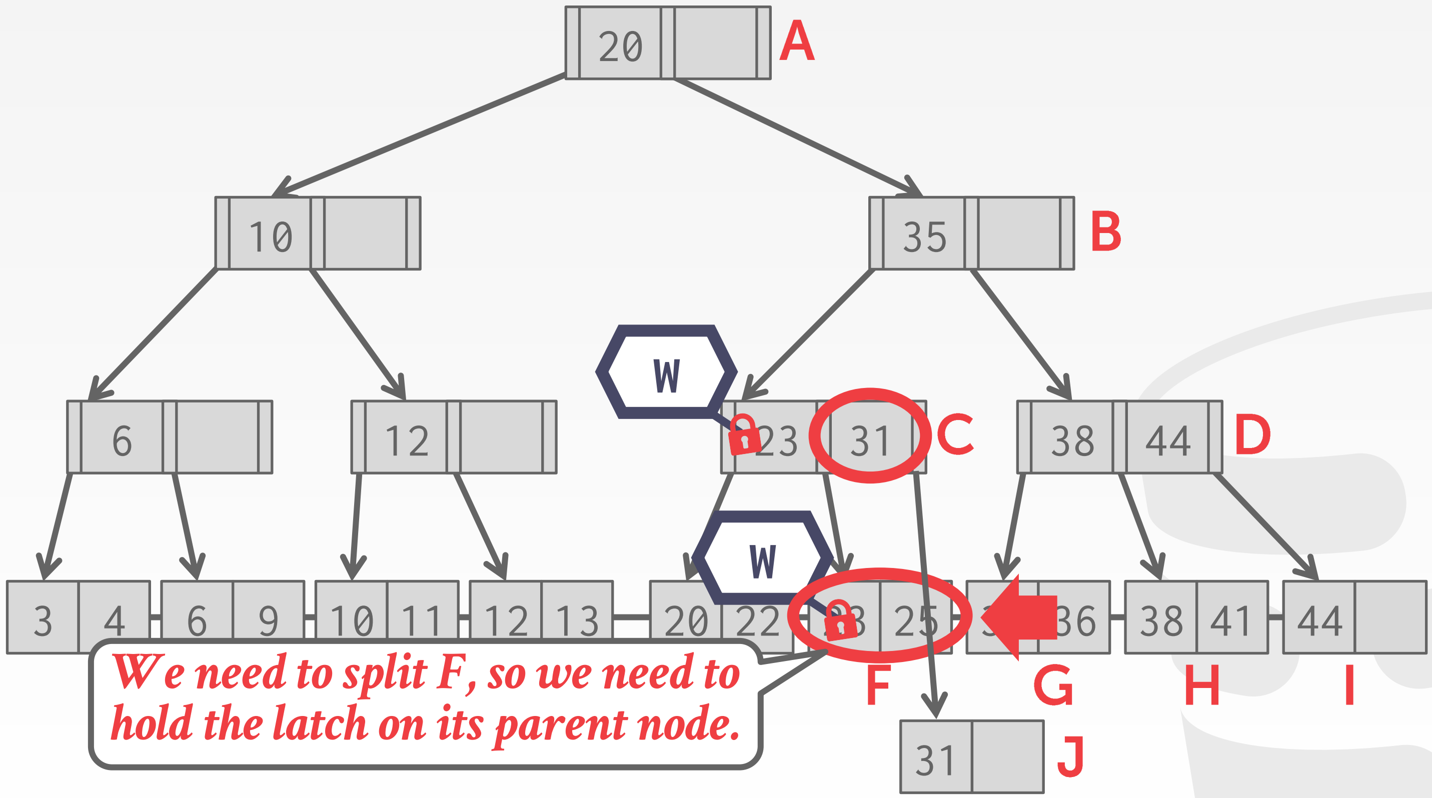


We need to split F, so we need to hold the latch on its parent node.

EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25

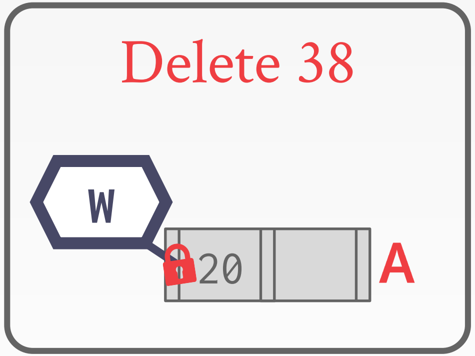


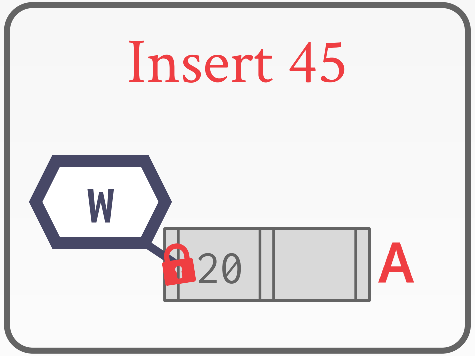
OBSERVATION

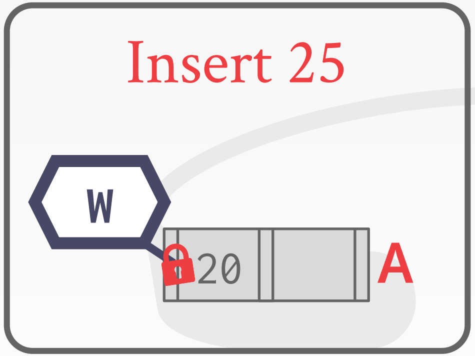
What was the first step that all the update examples did on the B+Tree?

OBSERVATION

What was the first step that all the update examples did on the B+Tree?

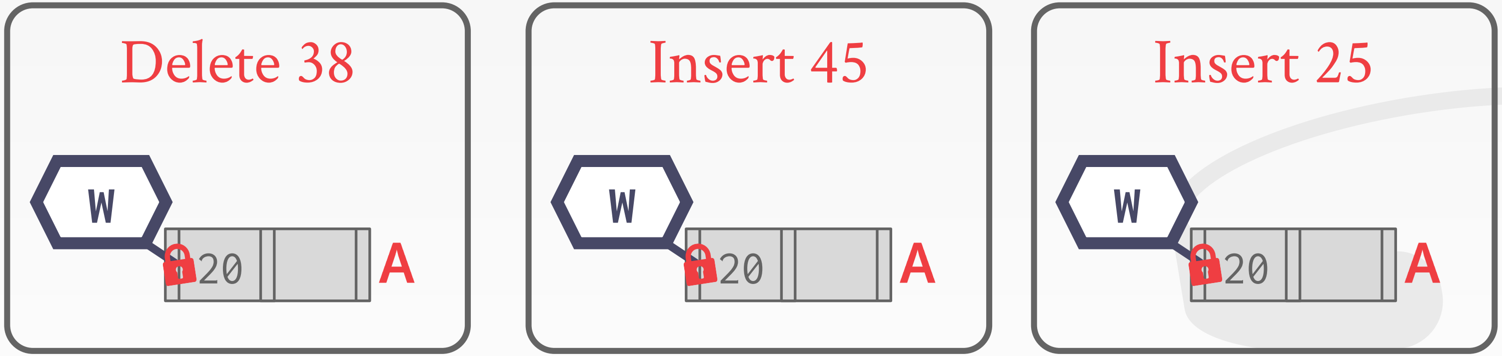






OBSERVATION

What was the first step that all the update examples did on the B+Tree?



Taking a write latch on the root every time becomes a bottleneck with higher concurrency.

BETTER LATCHING ALGORITHM

Most modifications to a B+Tree will not require a split or merge.

Instead of assuming that there will be a split/merge, optimistically traverse the tree using read latches.

If you guess wrong, repeat traversal with the pessimistic algorithm.

Acta Informatica 9, 1 – 21 (1977)

Concurrency of Operations on *B*-Trees

R. Bayer* and M. Schkolnick
IBM Research Laboratory, San José, CA 95193, USA

Summary. Concurrent operations on *B*-trees pose the problem of insuring that each operation can be carried out without interfering with other operations being performed simultaneously by other users. This problem can become critical if these structures are being used to support access paths, like indexes, to data base systems. In this case, serializing access to one of these indexes can create an unacceptable bottleneck for the entire system. Thus, there is a need for locking protocols that can assure integrity for each access while at the same time providing a maximum possible degree of concurrency. Another feature required from these protocols is that they be deadlock free, since the cost to resolve a deadlock may be high.

Recently, there has been some questioning on whether *B*-tree structures can support concurrent operations. In this paper, we examine the problem of concurrent access to *B*-trees. We present a deadlock free solution which can be tuned to specific requirements. An analysis is presented which allows the selection of parameters so as to satisfy these requirements.

The solution presented here uses simple locking protocols. Thus, we conclude that B-trees can be used advantageously in a multi-user environment.

1. Introduction

In this paper, we examine the problem of concurrent access to indexes which are maintained as *B*-trees. This type of organization was introduced by Bayer and McCreight [2] and some variants of it appear in Knuth [10] and Wedekind [13]. Performance studies of it were restricted to the single user environment. Recently, these structures have been examined for possible use in a multi-user (concurrent) environment. Some initial studies have been made about the feasibility of their use in this type of situation [1, 6], and [11].

An accessing schema which achieves a high degree of concurrency in using the index will be presented. The schema allows dynamic tuning to adapt its performance to the profile of the current set of users. Another property of the

* *Permanent address:* Institut für Informatik der Technischen Universität München, Arcisstr. 21, D-8000 München 2, Germany (Fed. Rep.)

BETTER LATCHING ALGORITHM

Search: Same as before.

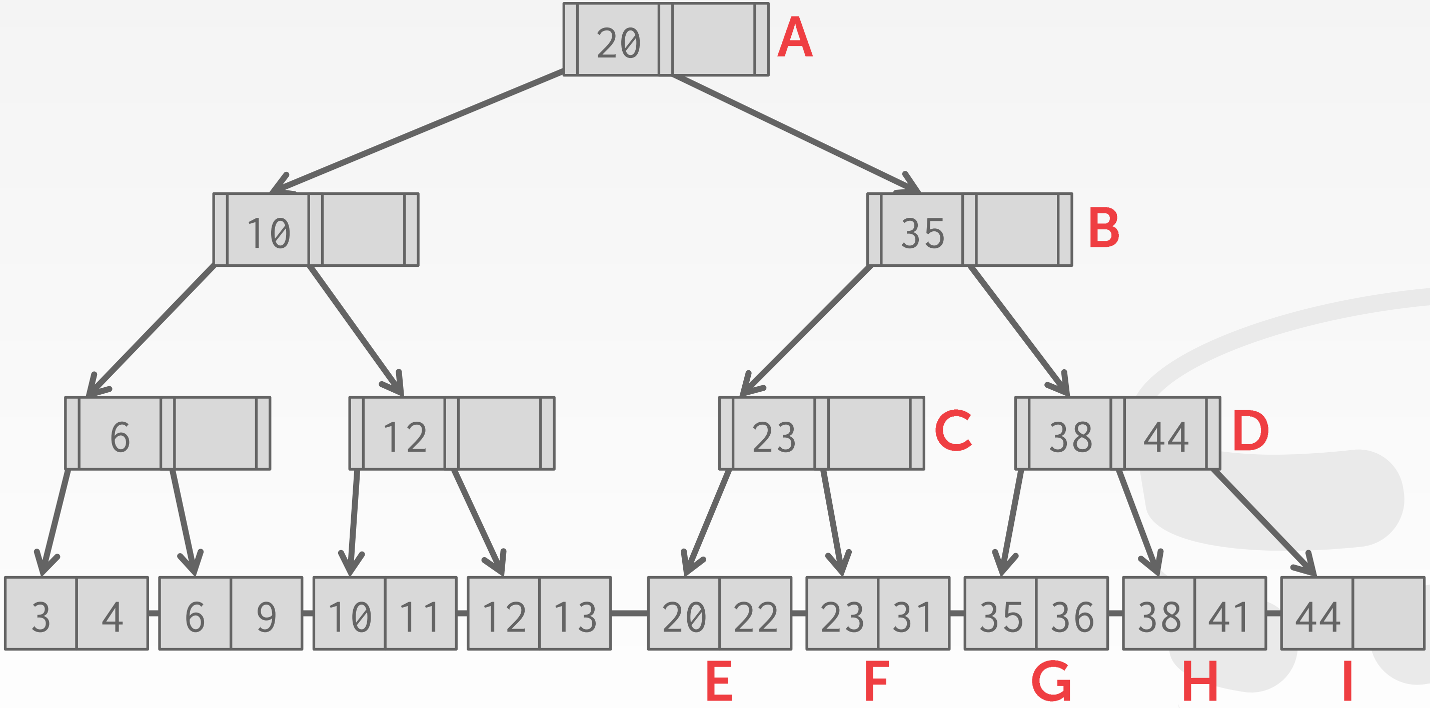
Insert/Delete:

→ Set latches as if for search, get to leaf, and set **W** latch on leaf.

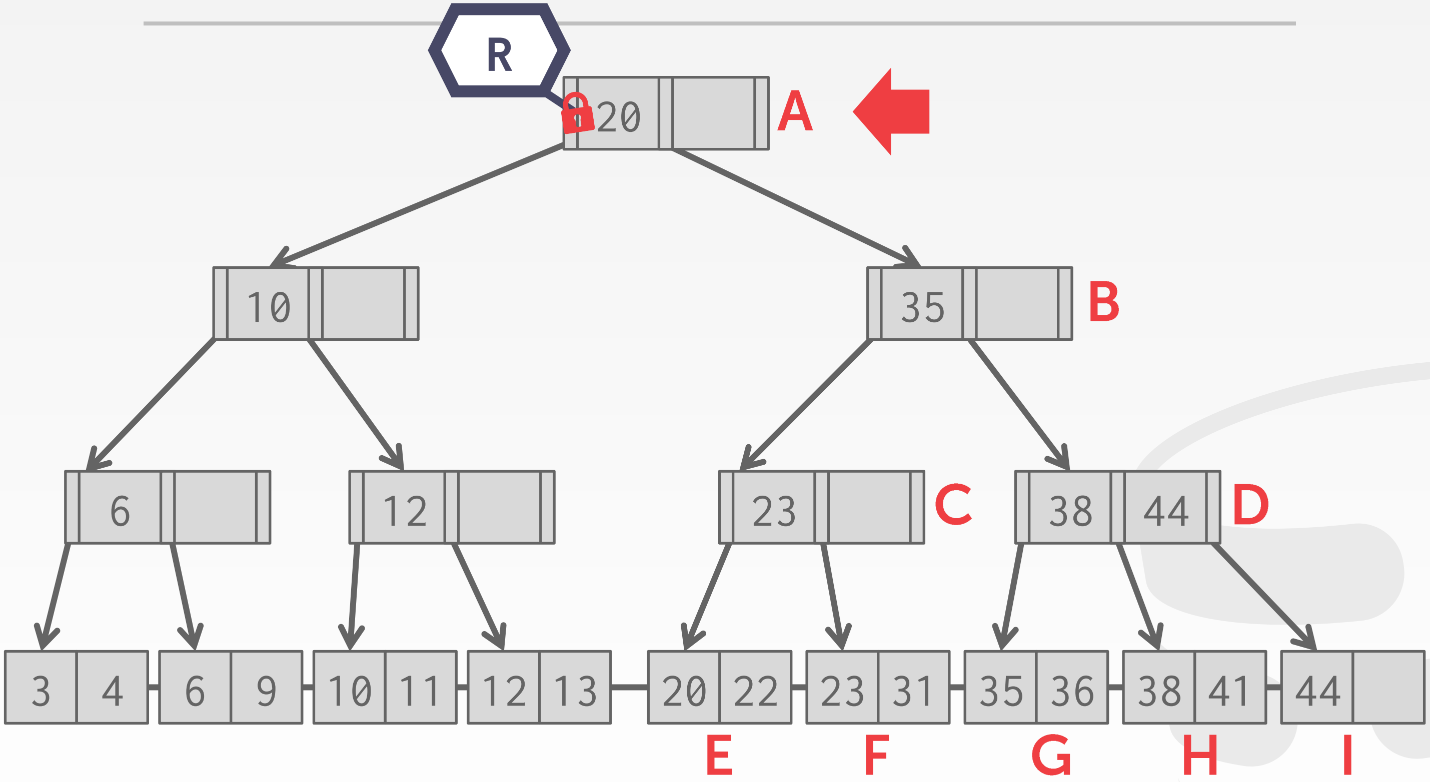
→ If leaf is not safe, release all latches, and restart thread using previous insert/delete protocol with write latches.

This approach optimistically assumes that only leaf node will be modified; if not, **R** latches set on the first pass to leaf are wasteful.

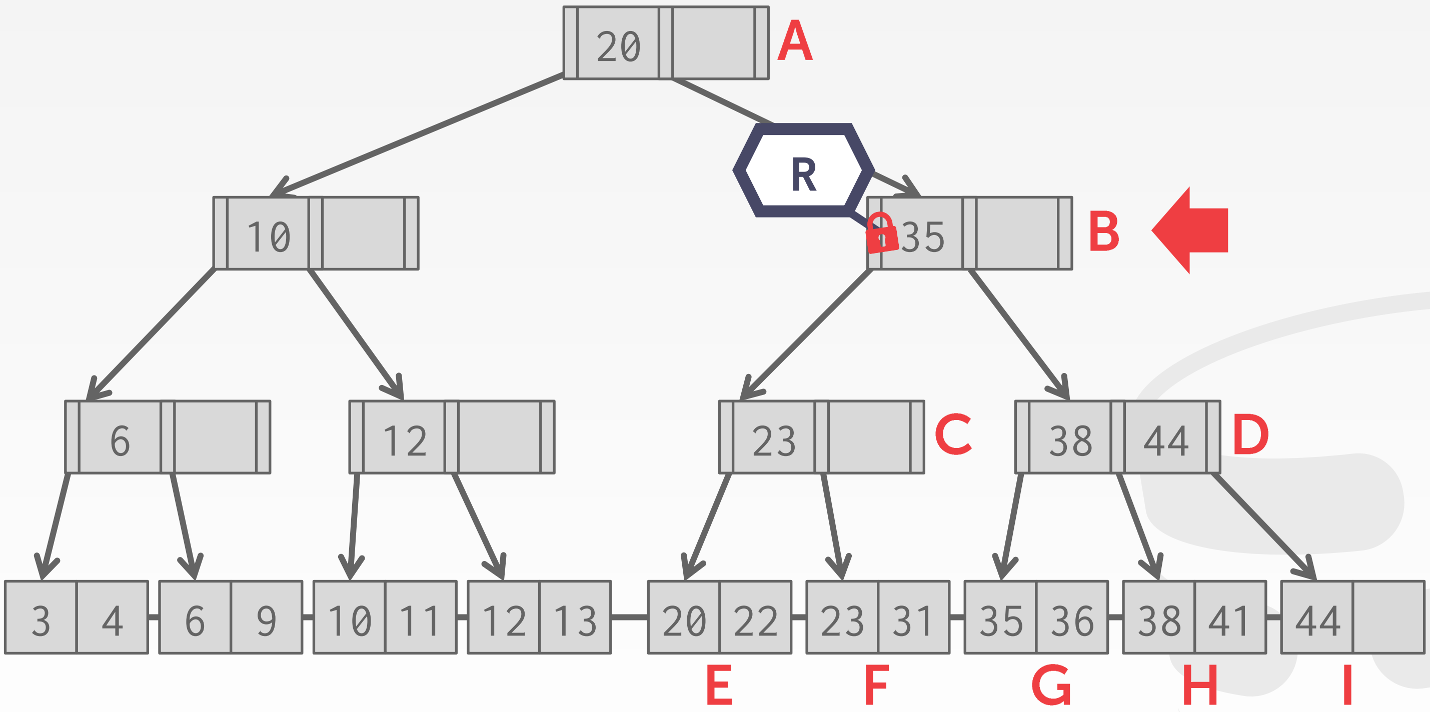
EXAMPLE #2 – DELETE 38



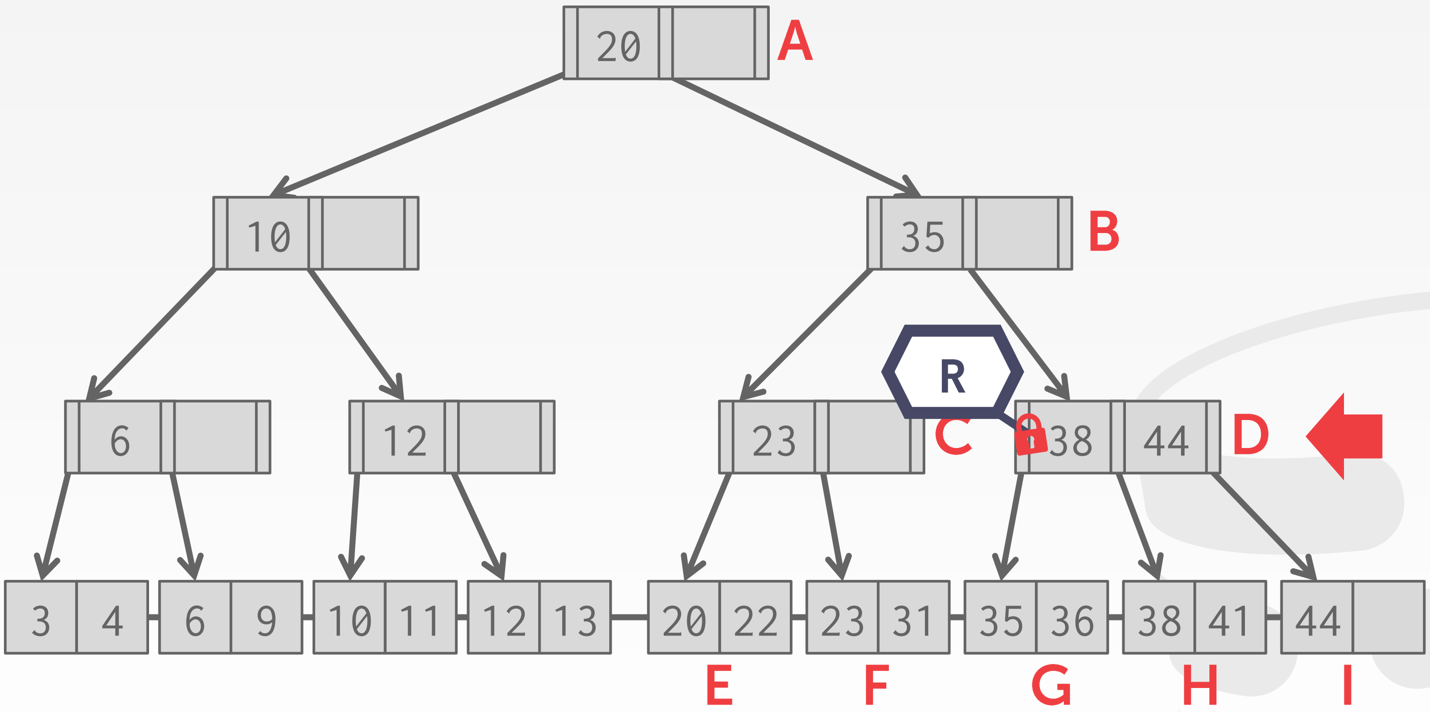
EXAMPLE #2 – DELETE 38



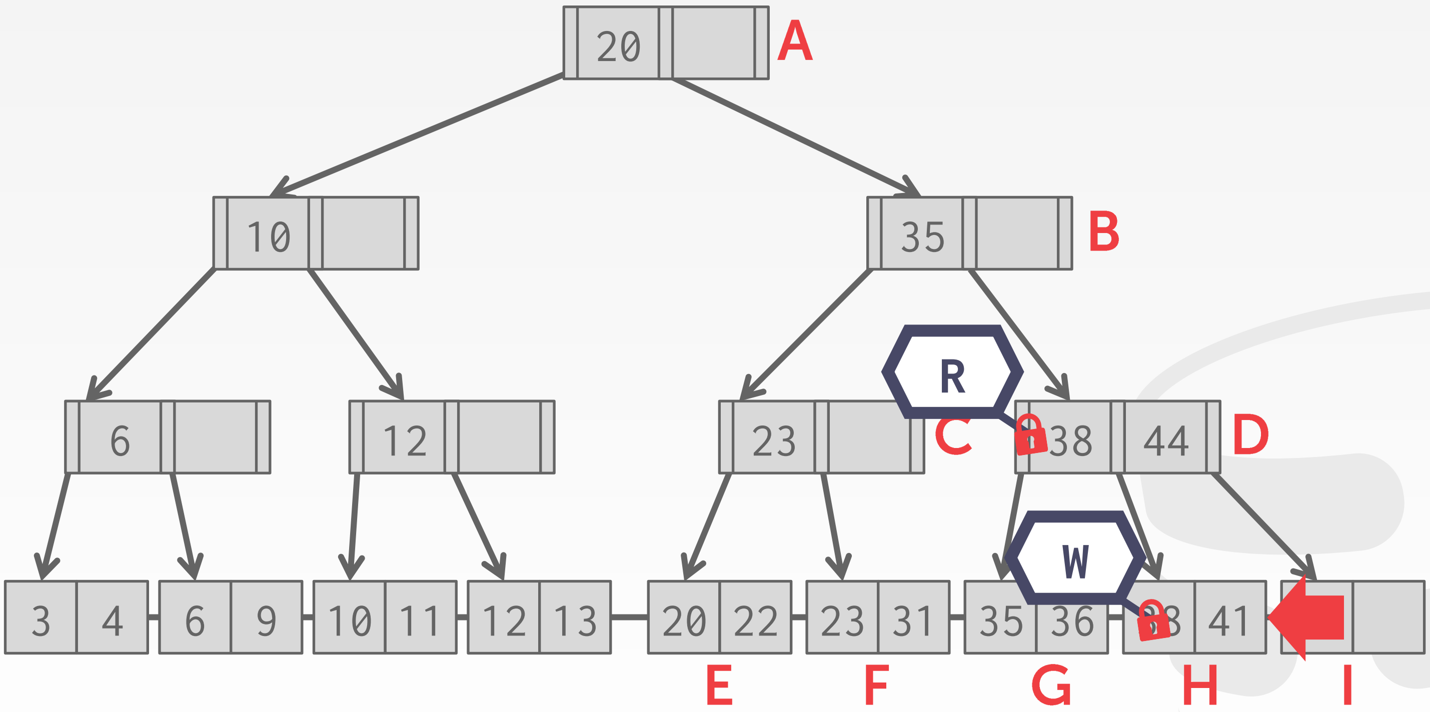
EXAMPLE #2 – DELETE 38



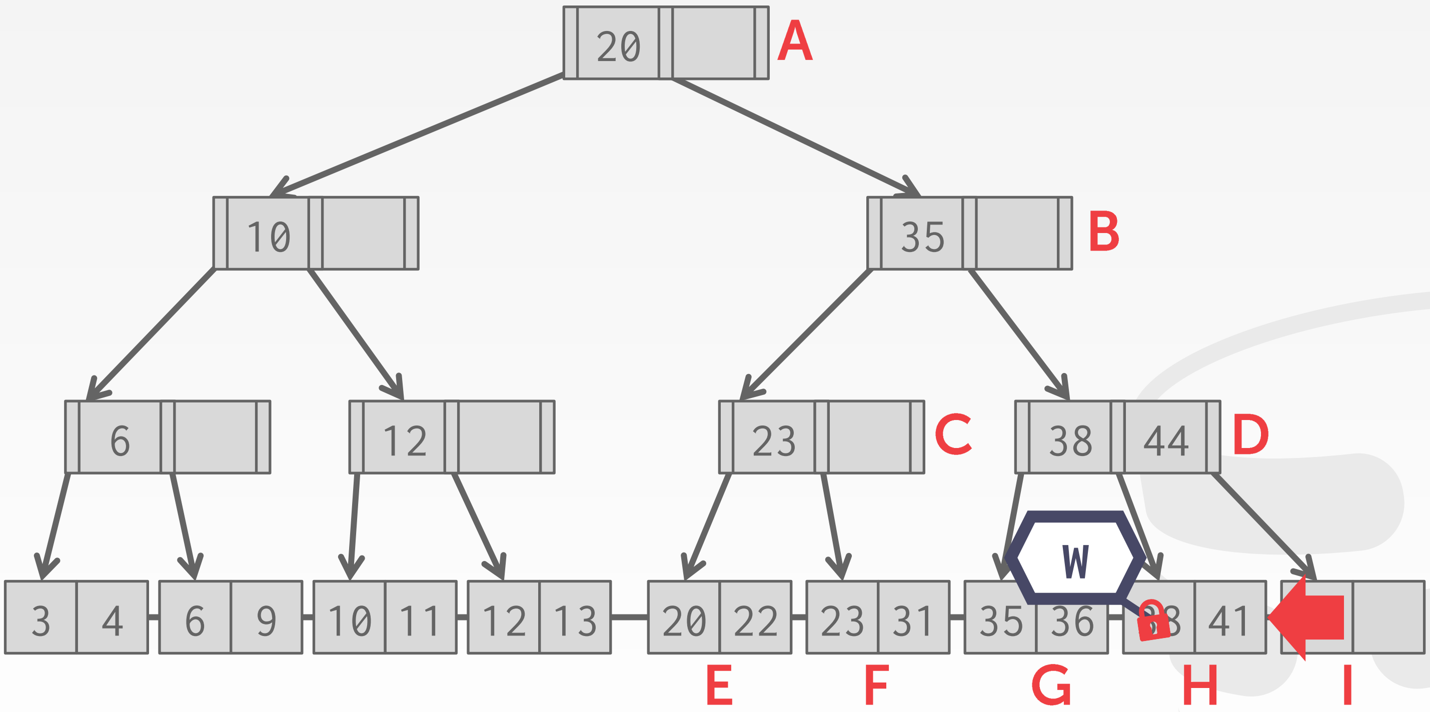
EXAMPLE #2 – DELETE 38



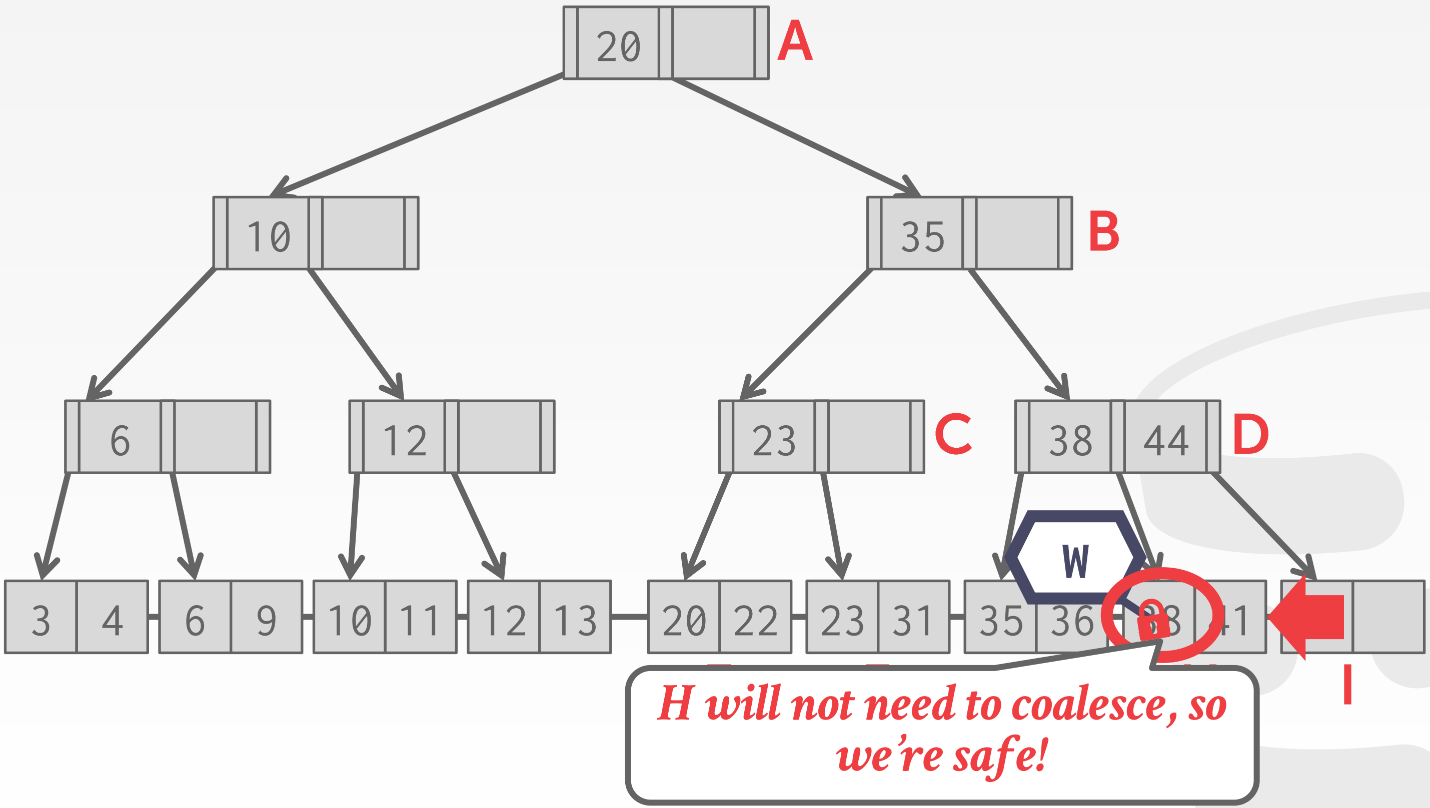
EXAMPLE #2 – DELETE 38



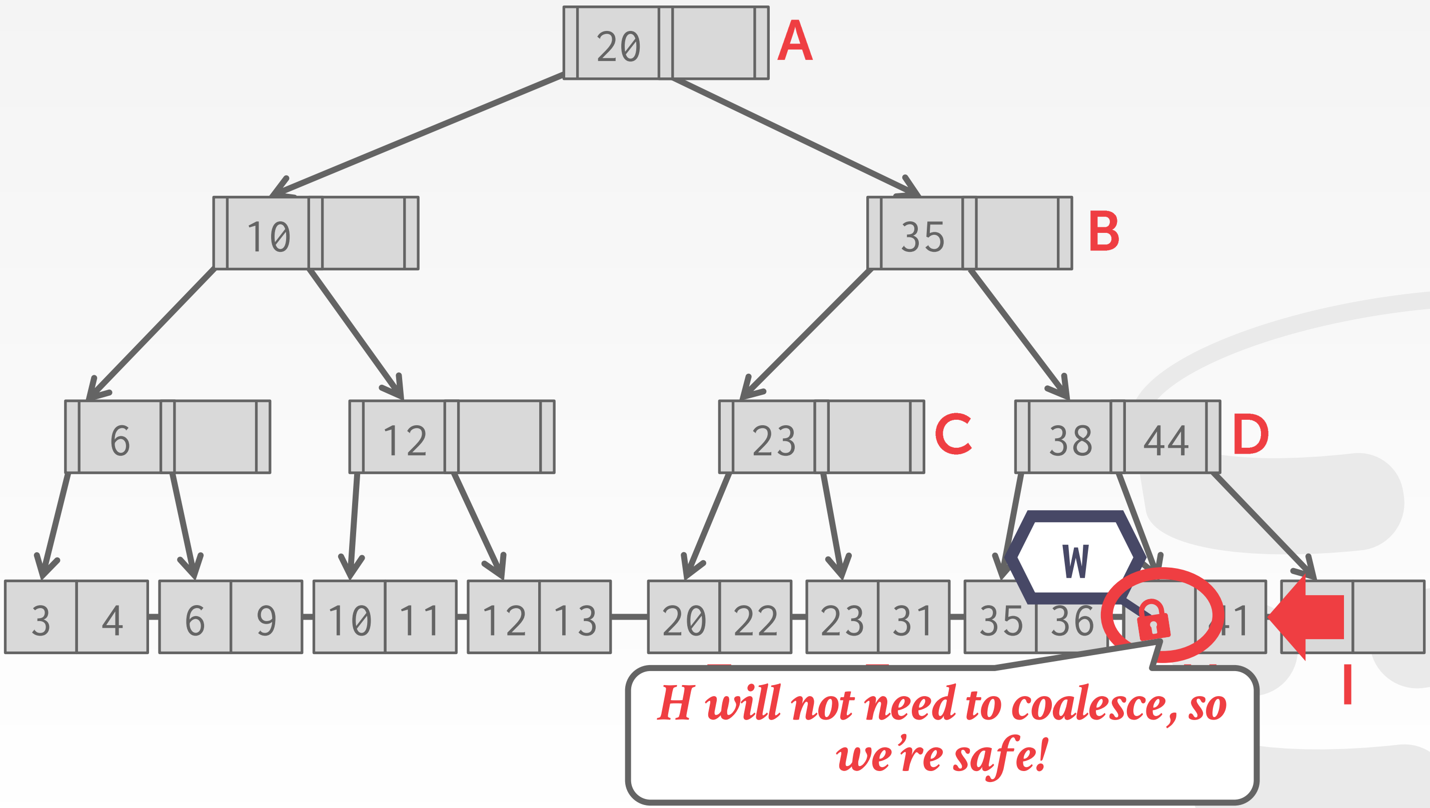
EXAMPLE #2 – DELETE 38



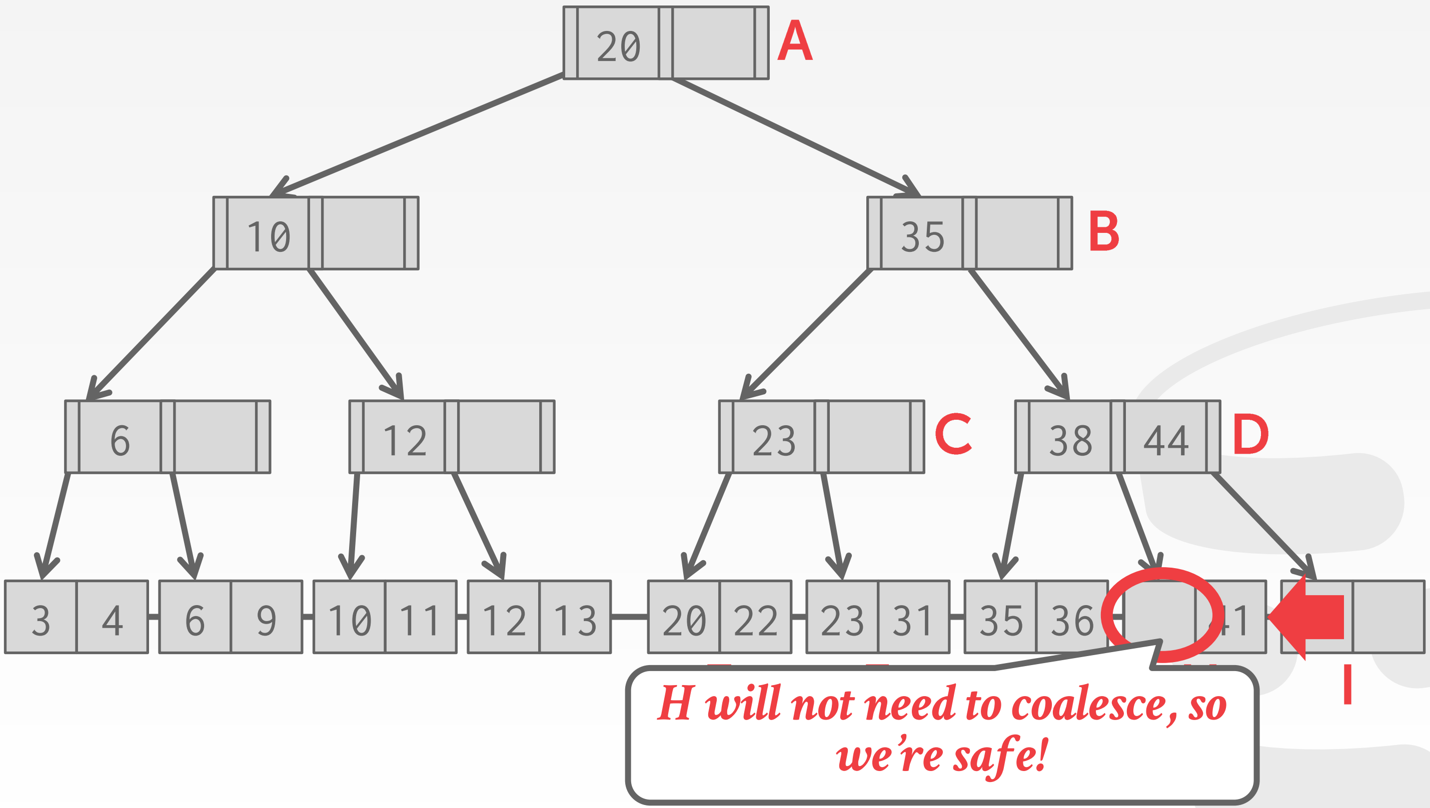
EXAMPLE #2 – DELETE 38



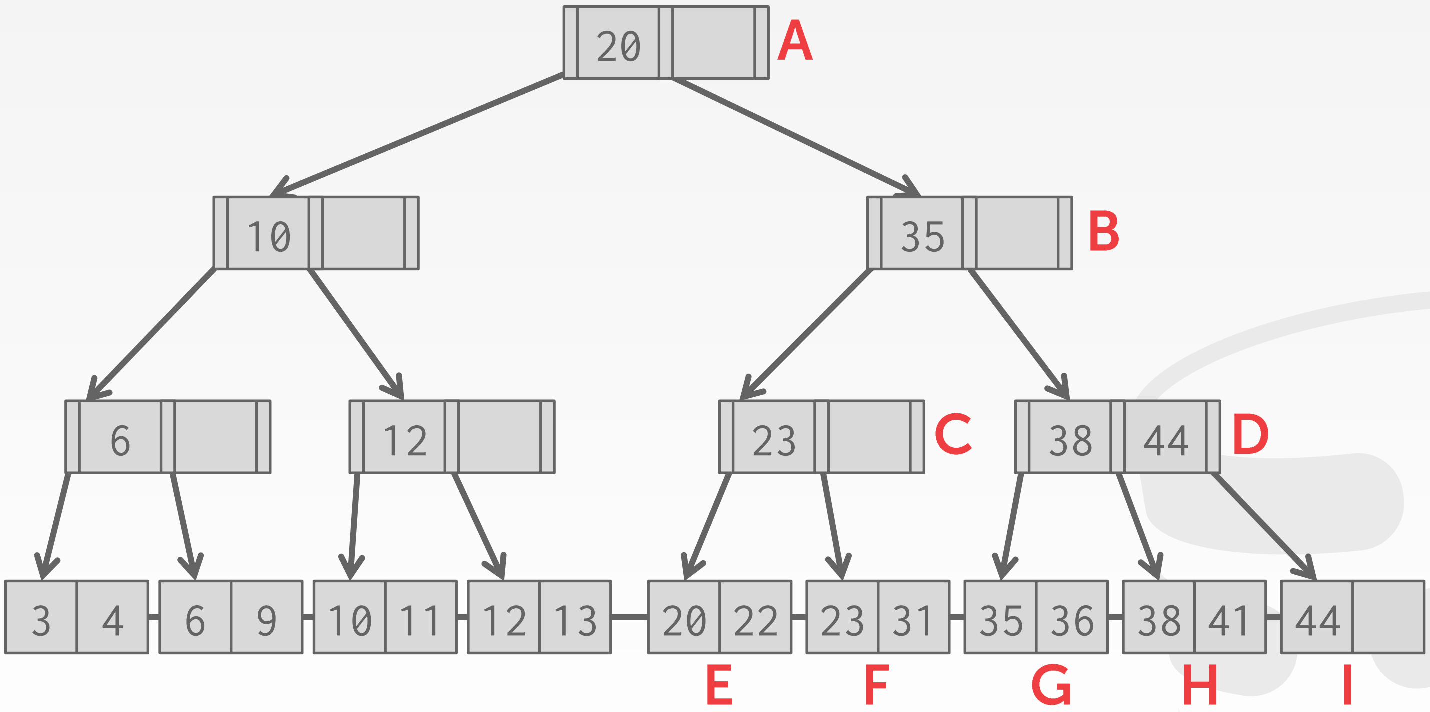
EXAMPLE #2 – DELETE 38



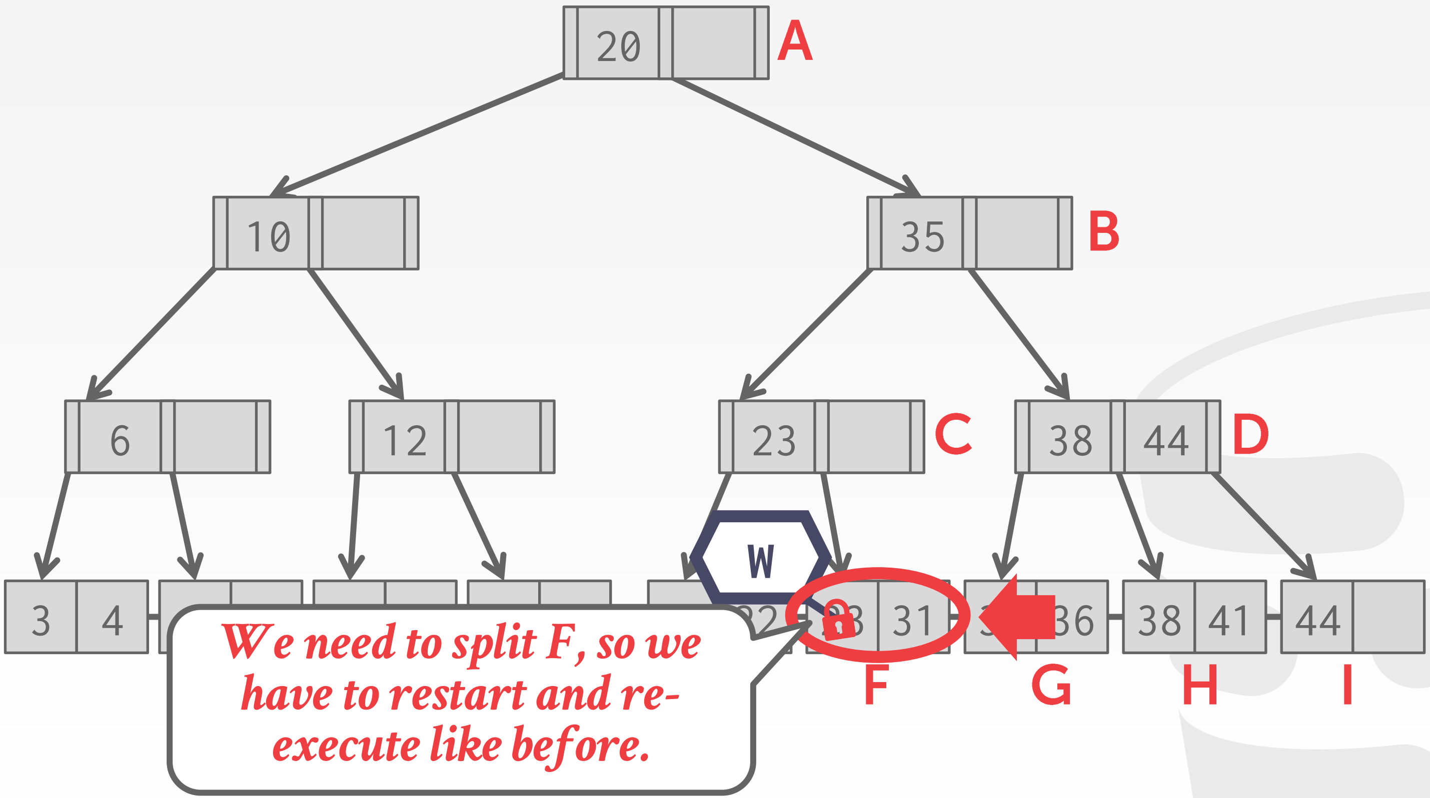
EXAMPLE #2 – DELETE 38



EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25



OBSERVATION

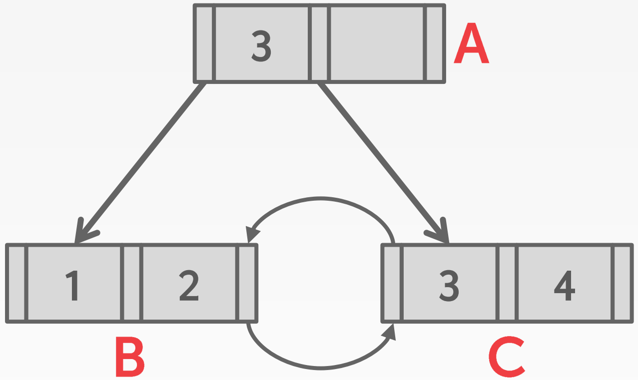
The threads in all the examples so far have acquired latches in a "top-down" manner.

→ A thread can only acquire a latch from a node that is below its current node.

→ If the desired latch is unavailable, the thread must wait until it becomes available.

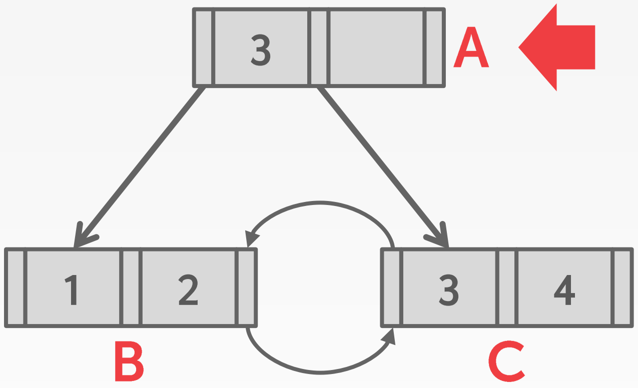
But what if we want to move from one leaf node to another leaf node?

LEAF NODE SCAN EXAMPLE #1



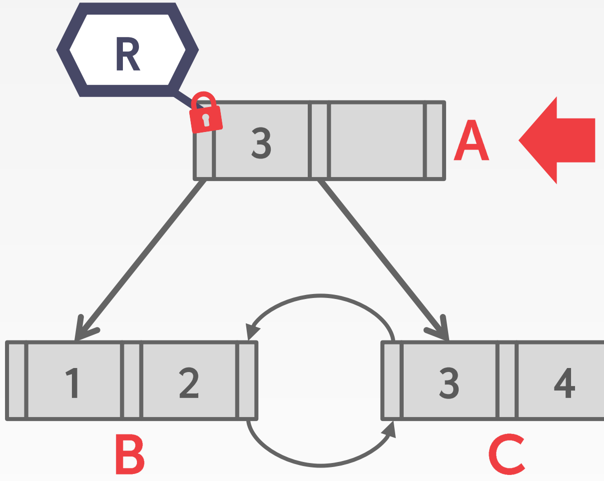
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



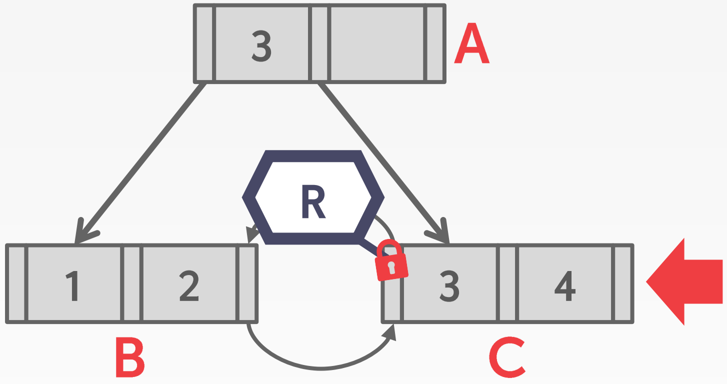
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



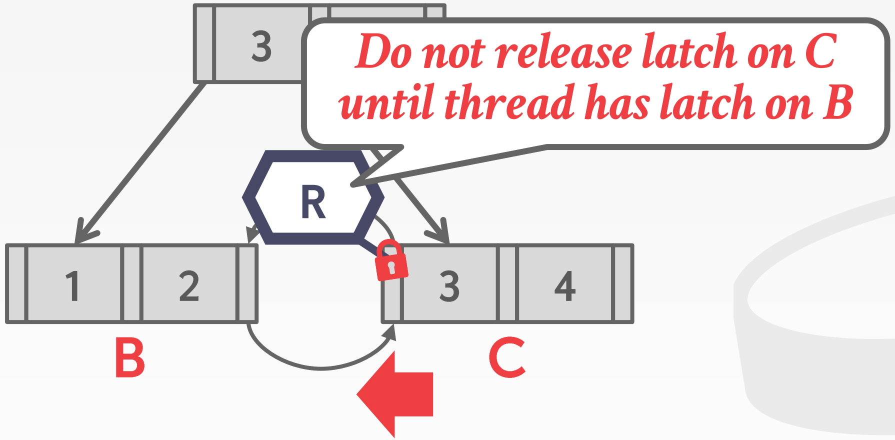
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



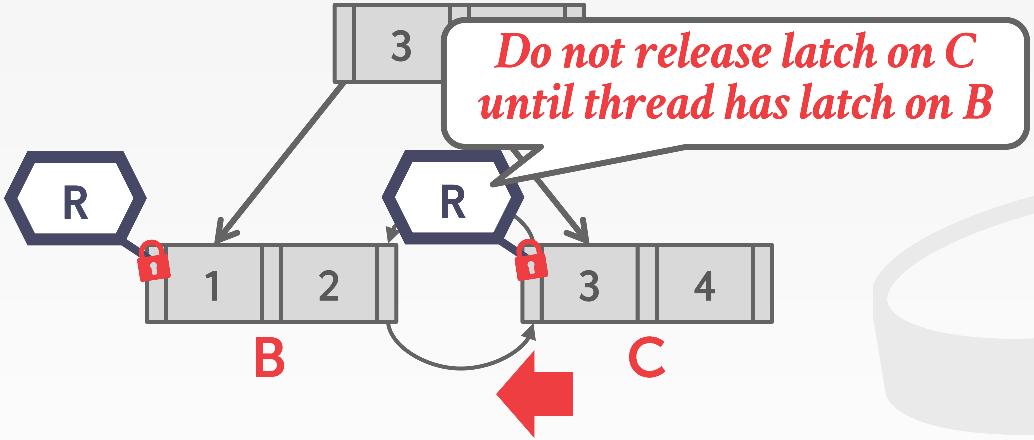
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



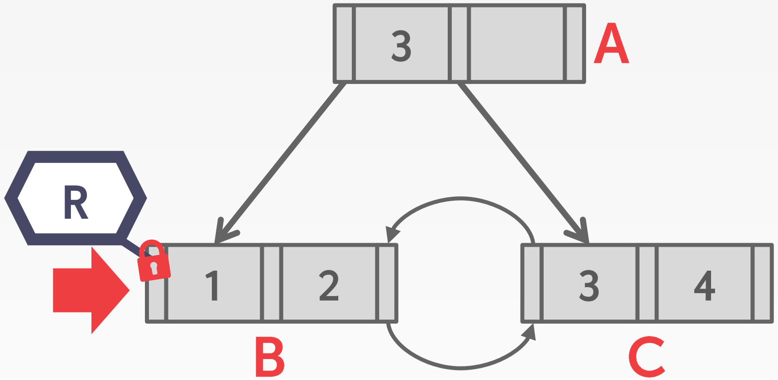
LEAF NODE SCAN EXAMPLE #1

T$_1$: Find Keys < 4



LEAF NODE SCAN EXAMPLE #1

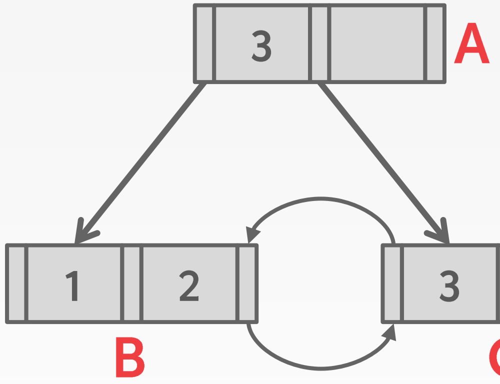
T_1: Find Keys < 4



LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

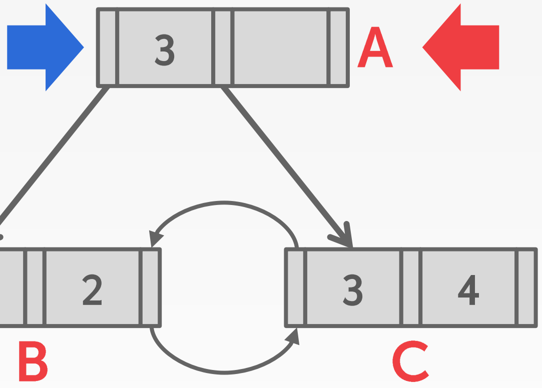
T_2: Find Keys > 1



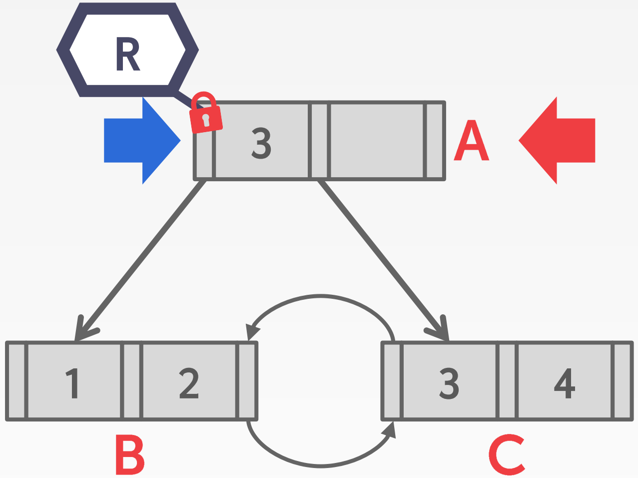
LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

T_2: Find Keys > 1



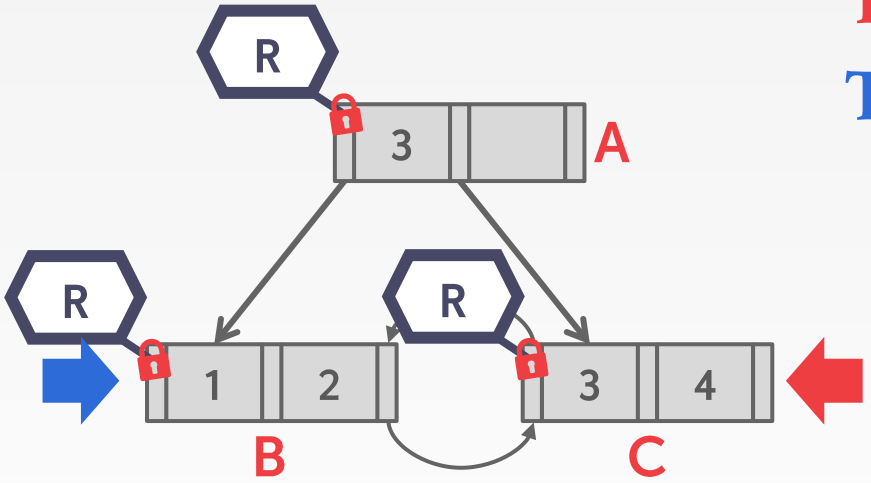
LEAF NODE SCAN EXAMPLE #2



T_1: Find Keys < 4

T_2: Find Keys > 1

LEAF NODE SCAN EXAMPLE #2



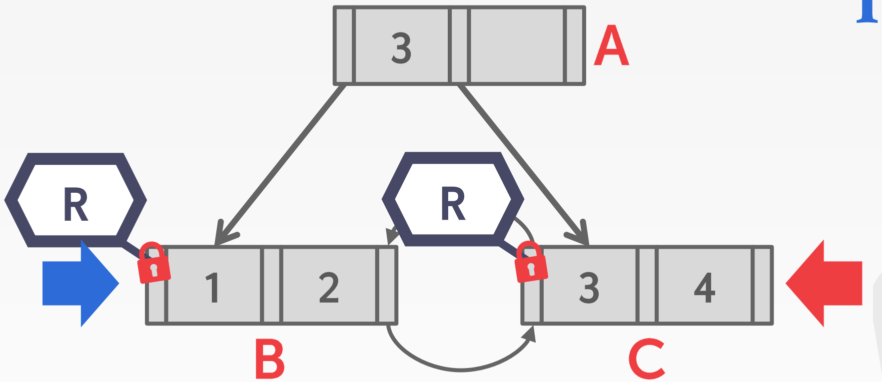
T_1: Find Keys < 4

T_2: Find Keys > 1

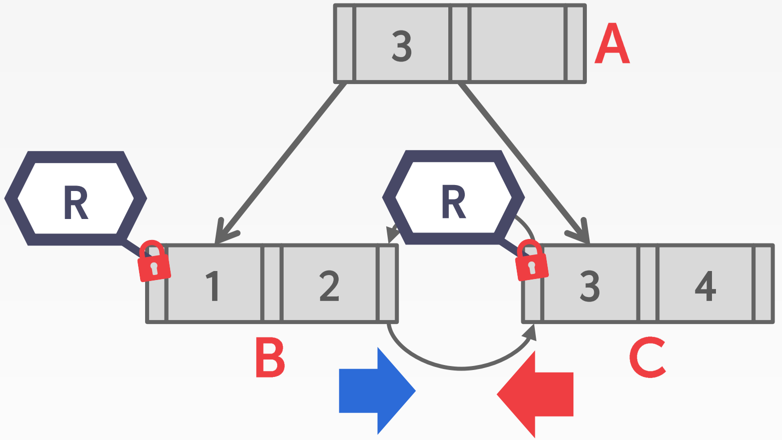
LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #2



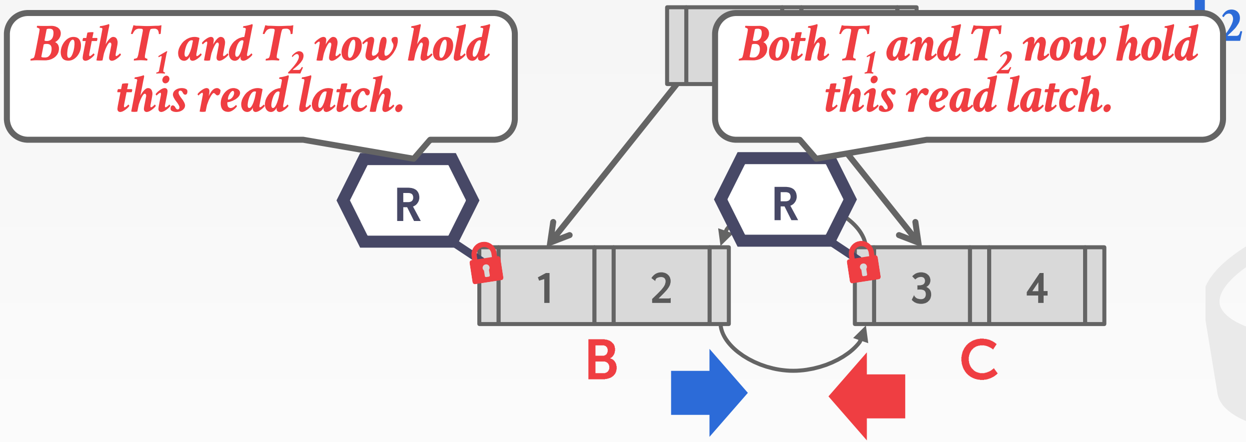
T_1: Find Keys < 4

T_2: Find Keys > 1

LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

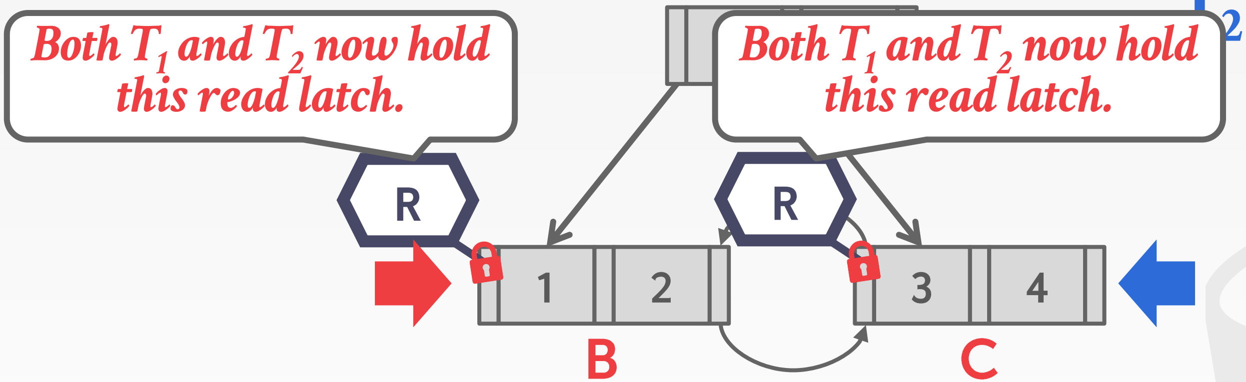
T_2: Find Keys > 1



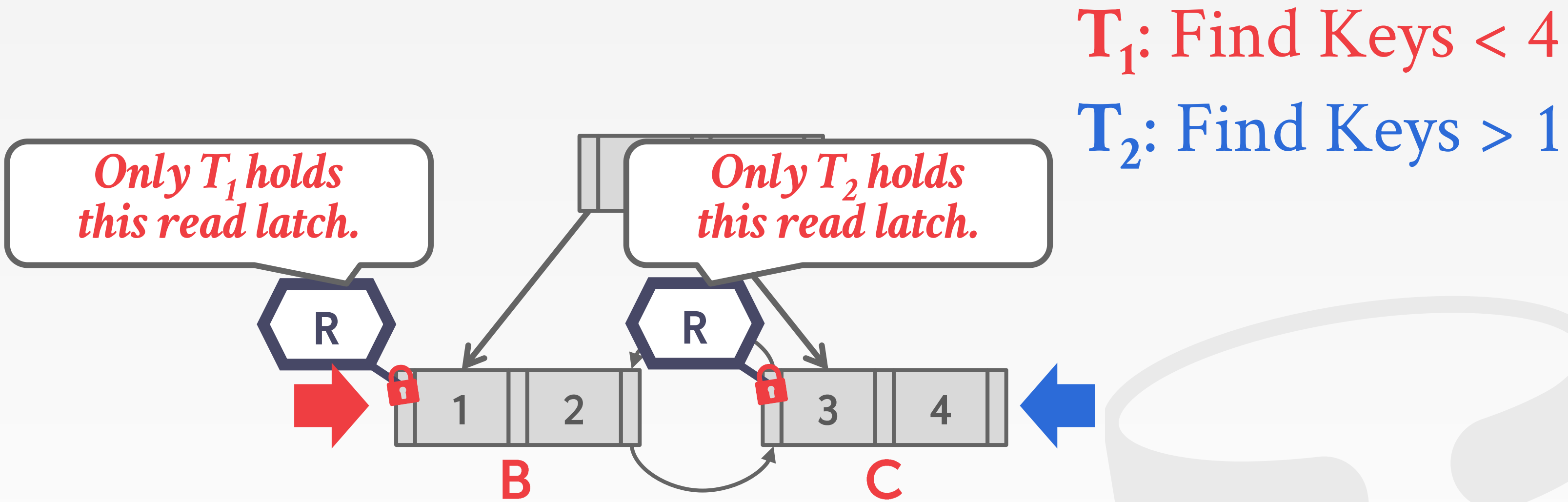
LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #2



LEAF NODE SCANS

Latches do not support deadlock detection or avoidance. The only way we can deal with this problem is through coding discipline.

The leaf node sibling latch acquisition protocol must support a "no-wait" mode.

The DBMS's data structures must cope with failed latch acquisitions.

CONCLUSION

Making a data structure thread-safe is notoriously difficult in practice.

We focused on B+Trees, but the same high-level techniques are applicable to other data structures.

NEXT CLASS

We are finally going to discuss how to execute some queries…