Carnegie Mellon University

Database Systems

Index Concurrency Control

15-445/645 FALL 2024 » PROF. ANDY PAVLO

ADMINISTRIVIA

Homework #3 is due Sunday Sept 6th @ 11:59pm

Project #2 is due Sunday Sept 29th @ 11:59pm
→ Recitation next week

Mid-term Exam on Wednesday Oct 9th @ 2:00pm
→ In-class in this room.

UPCOMING DATABASE TALKS

DataFusion Comet (DB Seminar)
→ Monday Sept 30th @ 4:30pm ET
→ Zoom

Oracle Talk (DB Group)
→ Tuesday Oct 1st @ 12:00pm ET
→ GHC 8115

ParadeDB (DB Seminar)
→ Monday Oct 7th @ 4:30pm ET
→ Zoom

OBSERVATION

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

A modern DBMS needs to allow multiple threads to safely access data structures to take advantage of additional CPU cores and hide disk I/O stalls.

They Don't Do This!

VOLTDB KX

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?

TODAY'S AGENDA

Latches Overview

Hash Table Latching

B+Tree Latching

Leaf Node Scans

Project #2 Announcement

LOCKS VS. LATCHES

Locks (Transactions)

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

Latches (Workers)

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

LOCKS VS. LATCHES

Lecture #15

	Locks	*Latches*
Separate…	Transactions	Workers (threads, processes)
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.

Compatibility Matrix

	Read	Write
Read	✓	X
Write	X	X

LATCH IMPLEMENTATION GOALS

Small memory footprint. 闩锁需要内联到数据结构中，因此尽可能占据小的内存空间.

Fast execution path when no contention.

Decentralized management of latches.

Avoid expensive system calls.

Source: Filip Pizlo

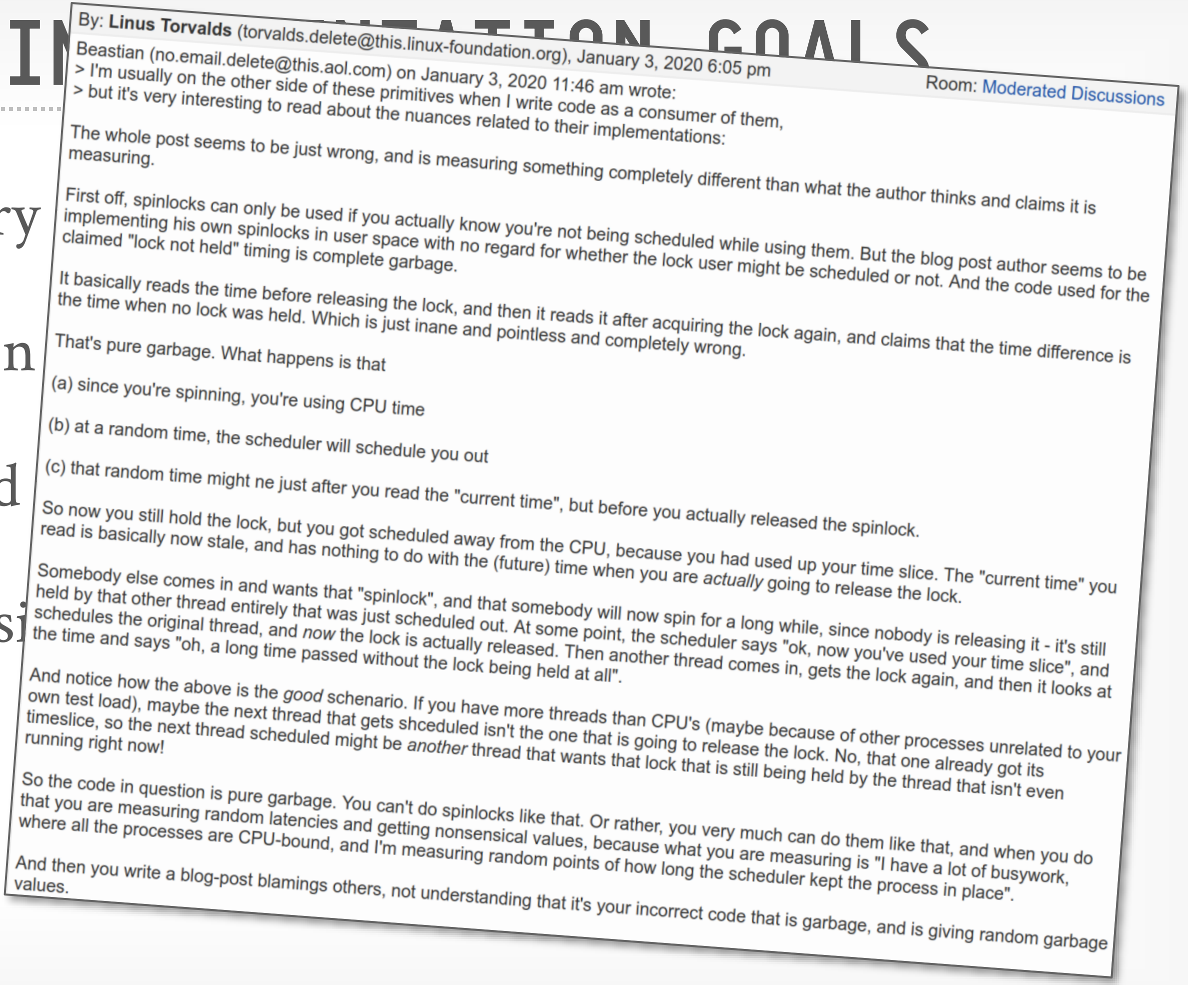
LATCH IMPLEMENTATION GOALS

Small memory

Fast execution

Decentralized

Avoid expensi



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.

Source: Filip Pizlo

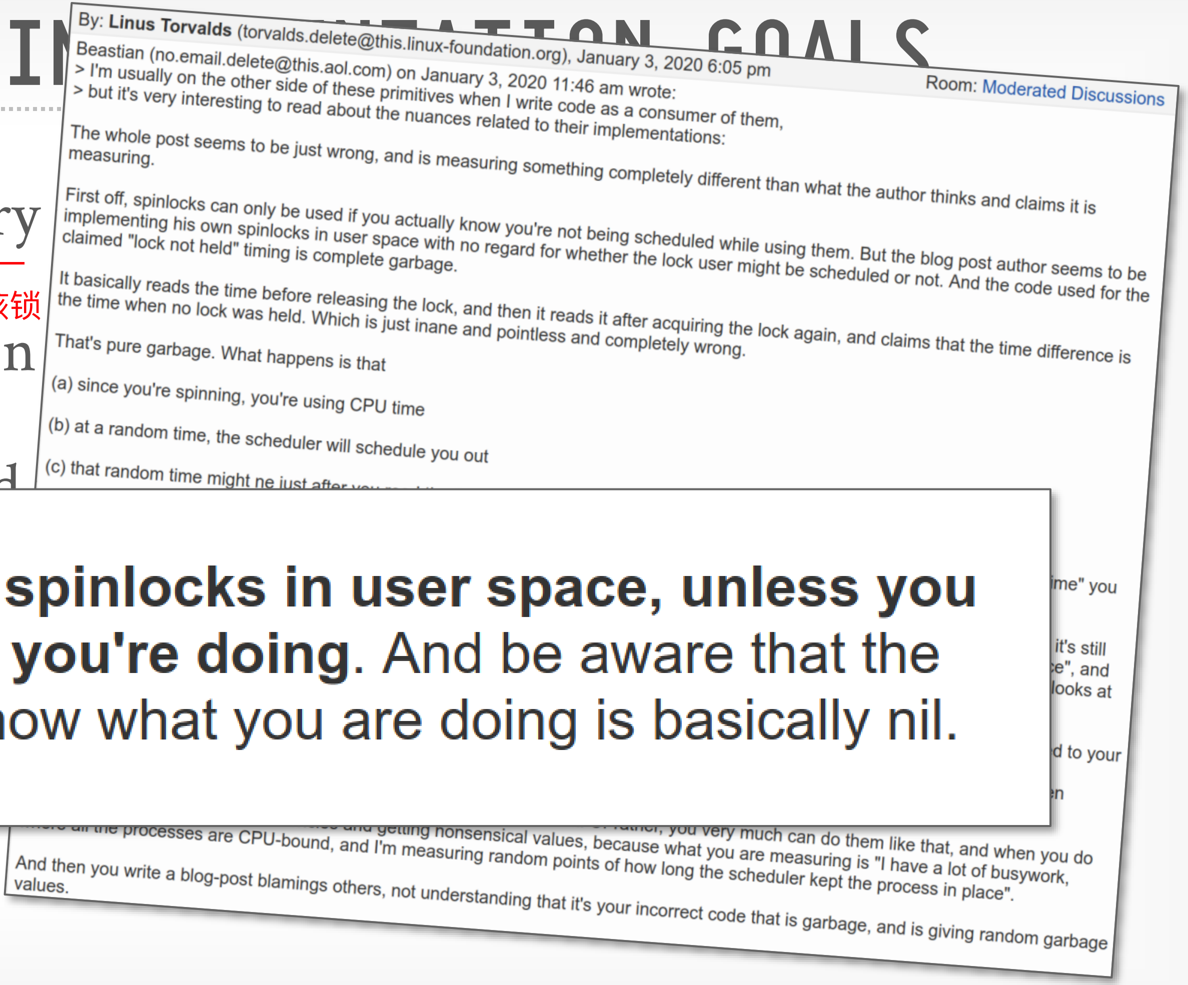
LATCH IMPLEMENTATION GOALS

Small memory

自旋锁是一种锁，当线程尝试获取它时，会在一个循环中简单地等待（“自旋”），反复检查该锁是否可用。

Fast execution

Decentralized



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 ...

... ime" you

... it's still ... e", and ... looks at

... d to your

... n

... processes are CPU-bound, and I'm measuring random points of how long the scheduler kept the process in place". ... getting nonsensical values, because what you are measuring is "I have a lot of busywork, ... you very much can do them like that, and when you do

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.

避免在用户空间使用自旋锁.

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.

Source: Filip Pizlo

LATCH IMPLEMENTATIONS

Test-and-Set Spinlock 硬件提供: 自旋锁

Blocking OS Mutex 操作系统提供的阻塞互斥锁

Reader-Writer Locks

Advanced approaches:
→ Adaptive Spinlock (Apple ParkingLot)
→ Queue-based Spinlock (MCS Locks)
→ Optimistic Lock Coupling (The Germans)

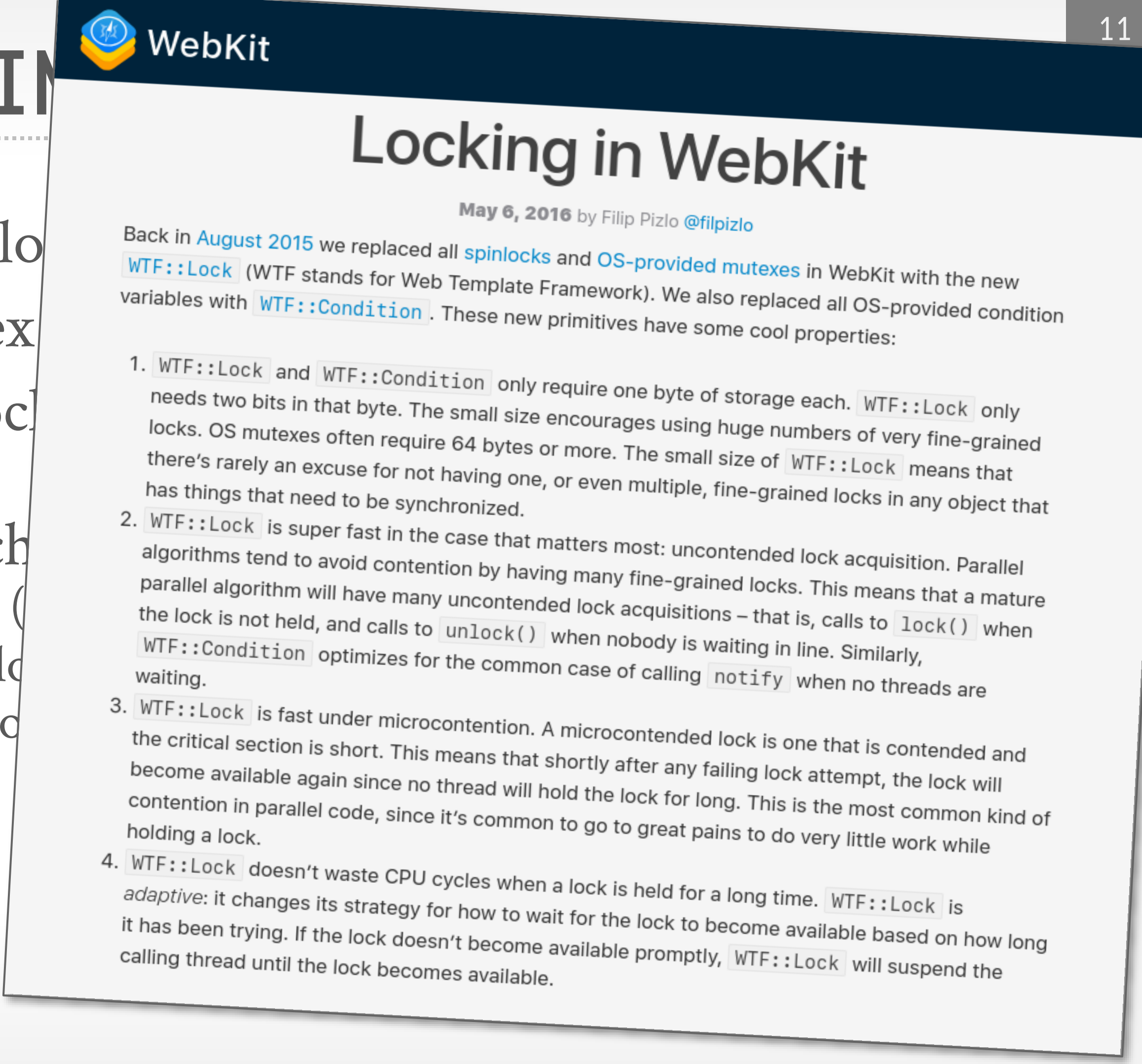
LATCH IN

Test-and-Set Spinlo

Blocking OS Mutex

Reader-Writer Loc

Advanced approach
→ Adaptive Spinlock (
→ Queue-based Spinlo
→ Optimistic Lock Co



Locking in WebKit

May 6, 2016 by Filip Pizlo @filpizlo

Back in August 2015 we replaced all spinlocks and OS-provided mutexes in WebKit with the new `WTF::Lock` (WTF stands for Web Template Framework). We also replaced all OS-provided condition variables with `WTF::Condition`. These new primitives have some cool properties:

1. `WTF::Lock` and `WTF::Condition` only require one byte of storage each. `WTF::Lock` only needs two bits in that byte. The small size encourages using huge numbers of very fine-grained locks. OS mutexes often require 64 bytes or more. The small size of `WTF::Lock` means that there's rarely an excuse for not having one, or even multiple, fine-grained locks in any object that has things that need to be synchronized.
2. `WTF::Lock` is super fast in the case that matters most: uncontended lock acquisition. Parallel algorithms tend to avoid contention by having many fine-grained locks. This means that a mature parallel algorithm will have many uncontended lock acquisitions – that is, calls to `lock()` when the lock is not held, and calls to `unlock()` when nobody is waiting in line. Similarly, `WTF::Condition` optimizes for the common case of calling `notify` when no threads are waiting.
3. `WTF::Lock` is fast under microcontention. A microcontended lock is one that is contended and the critical section is short. This means that shortly after any failing lock attempt, the lock will become available again since no thread will hold the lock for long. This is the most common kind of contention in parallel code, since it's common to go to great pains to do very little work while holding a lock.
4. `WTF::Lock` doesn't waste CPU cycles when a lock is held for a long time. `WTF::Lock` is *adaptive*: it changes its strategy for how to wait for the lock to become available based on how long it has been trying. If the lock doesn't become available promptly, `WTF::Lock` will suspend the calling thread until the lock becomes available.

LATCH IM

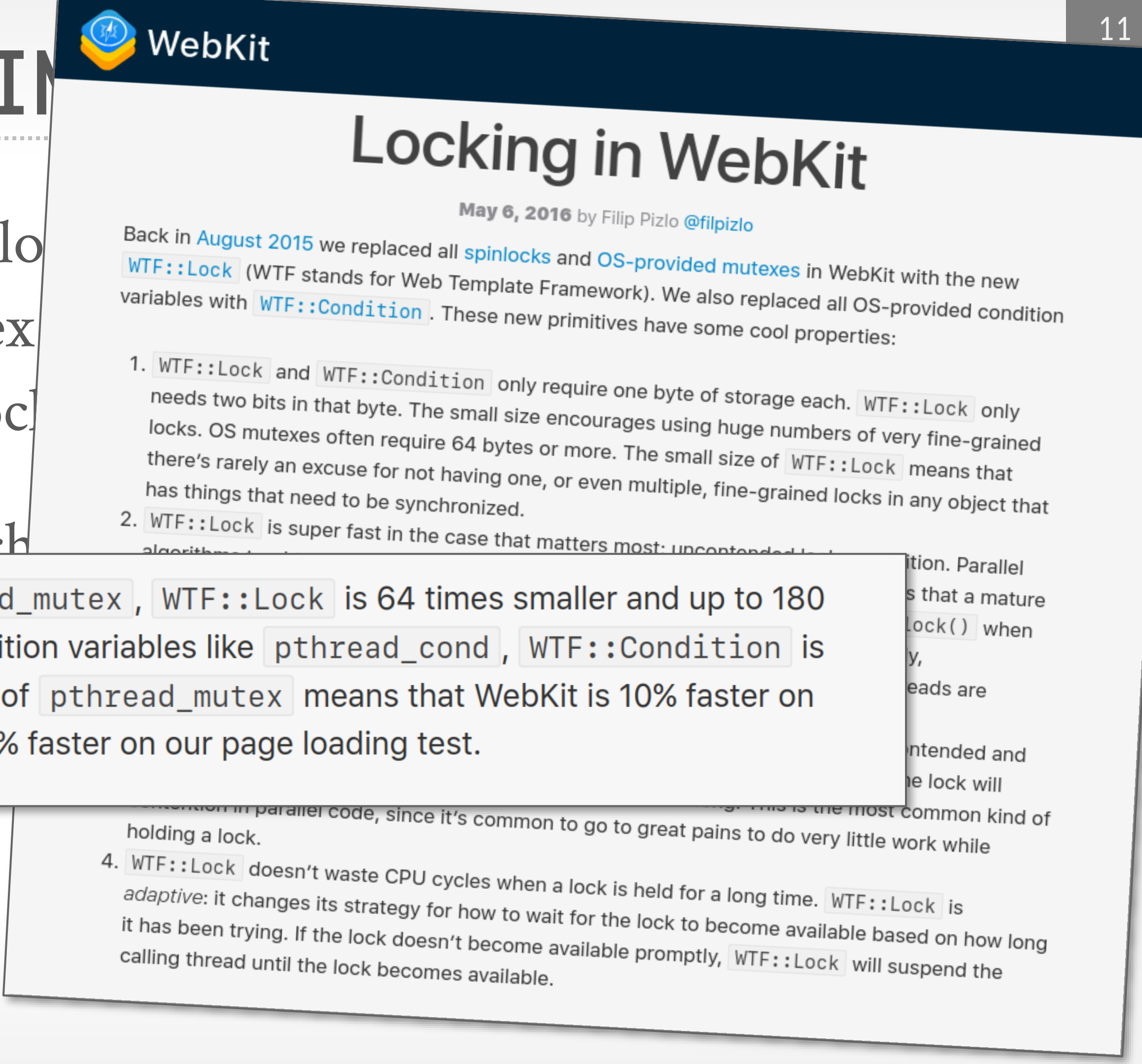
Test-and-Set Spinlo

Blocking OS Mutex

Reader-Writer Loc

避免依赖 OS 提供的 pthread 库实现.

Advanced approach



Locking in WebKit

May 6, 2016 by Filip Pizlo @filpizlo

Back in August 2015 we replaced all spinlocks and OS-provided mutexes in WebKit with the new `WTF::Lock` (WTF stands for Web Template Framework). We also replaced all OS-provided condition variables with `WTF::Condition`. These new primitives have some cool properties:

1. `WTF::Lock` and `WTF::Condition` only require one byte of storage each. `WTF::Lock` only needs two bits in that byte. The small size encourages using huge numbers of very fine-grained locks. OS mutexes often require 64 bytes or more. The small size of `WTF::Lock` means that there's rarely an excuse for not having one, or even multiple, fine-grained locks in any object that has things that need to be synchronized.
2. `WTF::Lock` is super fast in the case that matters most: uncontended lo

   algorithms

   ition. Parallel

   s that a mature

   `lock()` when

   y,

   eads are

   ntended and

   e lock will

   g. This is the most common kind of contention in parallel code, since it's common to go to great pains to do very little work while holding a lock.
4. `WTF::Lock` doesn't waste CPU cycles when a lock is held for a long time. `WTF::Lock` is *adaptive*: it changes its strategy for how to wait for the lock to become available based on how long it has been trying. If the lock doesn't become available promptly, `WTF::Lock` will suspend the calling thread until the lock becomes available.

Compared to OS-provided locks like `pthread_mutex`, `WTF::Lock` is 64 times smaller and up to 180 times faster. Compared to OS-provided condition variables like `pthread_cond`, `WTF::Condition` is 64 times smaller. Using `WTF::Lock` instead of `pthread_mutex` means that WebKit is 10% faster on JetStream, 5% faster on Speedometer, and 5% faster on our page loading test.

LATCH IMPLEMENTATIONS

Approach #1: Test-and-Set Spin Latch (TAS)

→ Very efficient (single instruction to latch/unlatch)

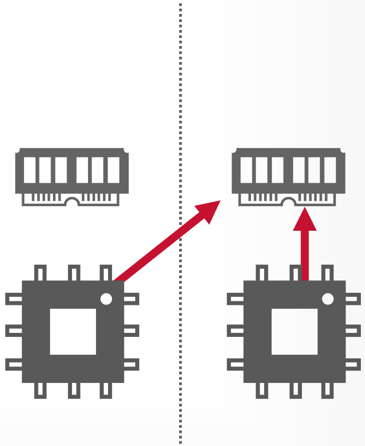
→ Non-scalable, not cache friendly, not OS friendly.

→ Example: **`std::atomic<T>`**

https://en.cppreference.com/w/cpp/atomic/atomic_flag.html

std::atomic<bool>

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



多插槽 CPU 互联

每个 CPU 有自己的局部 DRAM

NUMA: "Non-Uniform Memory Access"

COMPARE-AND-SWAP

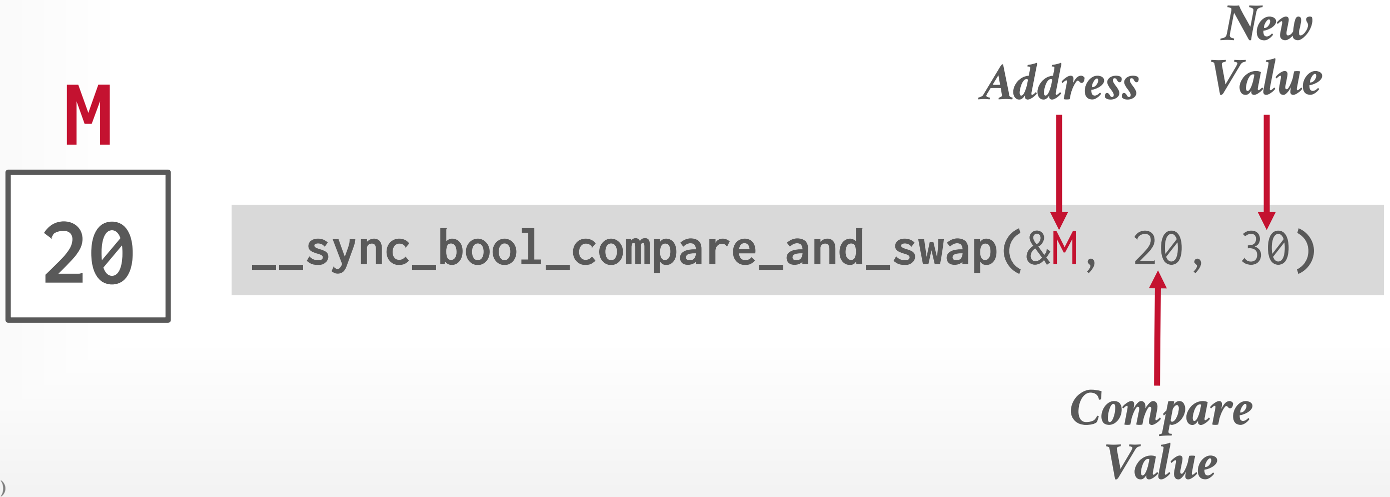
硬件提供了 TAS 和 CAS 操作指令.

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

See C++11 Atomics



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

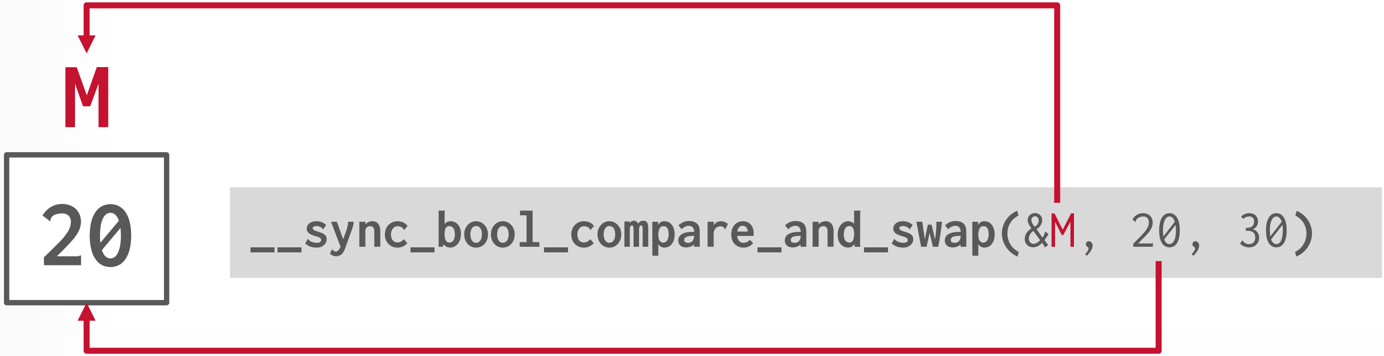
COMPARE-AND-SWAP

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

See C++11 Atomics



M

20

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

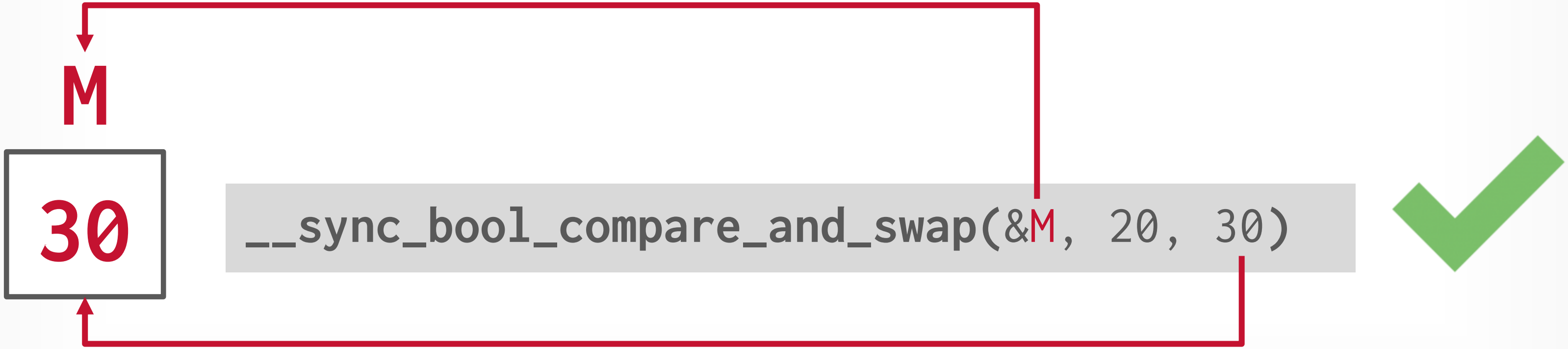
COMPARE-AND-SWAP

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

See C++11 Atomics



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

LATCH IMPLEMENTATIONS

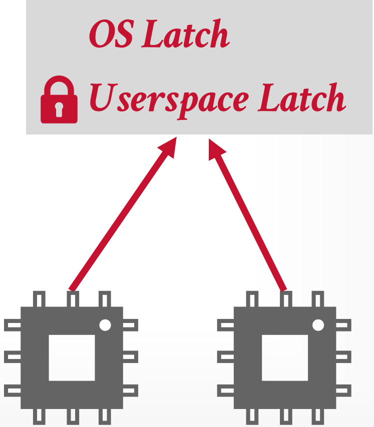
操作系统原语: 操作系统阻塞互斥锁.

Approach #2: Blocking OS Mutex

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

fast user-space locking

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



LATCH IMPLEMENTATIONS

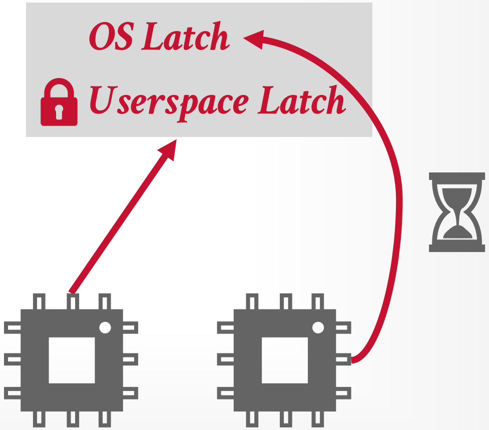
Approach #2: Blocking OS Mutex

→ Simple to use

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

→ Example: `std::mutex` → `pthread_mutex_t` → `futex`

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



LATCH IMPLEMENTATIONS

Approach #3: Reader-Writer Latches

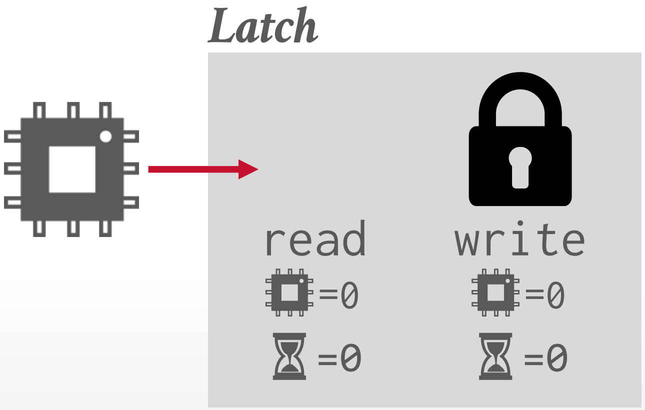
→ Allows for concurrent readers. Must manage read/write queues to avoid starvation.

→ Can be implemented on top of spinlocks.

→ Example: **`std::shared_mutex`** → **`pthread_rwlock_t`**

→ **`pthread_mutex_t`**

→ **`pthread_cond_t`**



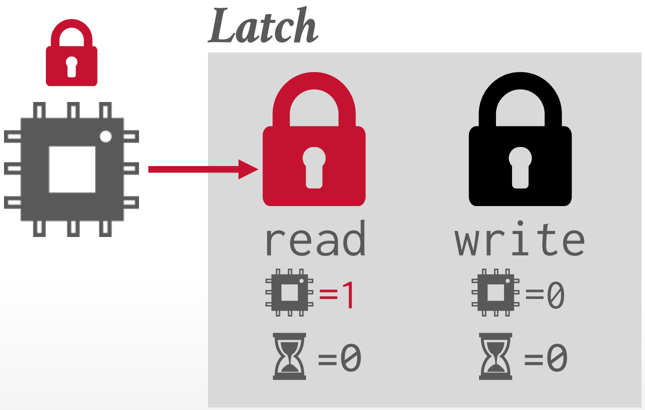
LATCH IMPLEMENTATIONS

Approach #3: Reader-Writer Latches

→ Allows for concurrent readers. Must manage read/write queues to avoid starvation.

→ Can be implemented on top of spinlocks.

→ Example: **`std::shared_mutex`** → `pthread_rwlock_t` → `pthread_mutex_t`, `pthread_cond_t`



LATCH IMPLEMENTATIONS

Approach #3: Reader-Writer Latches

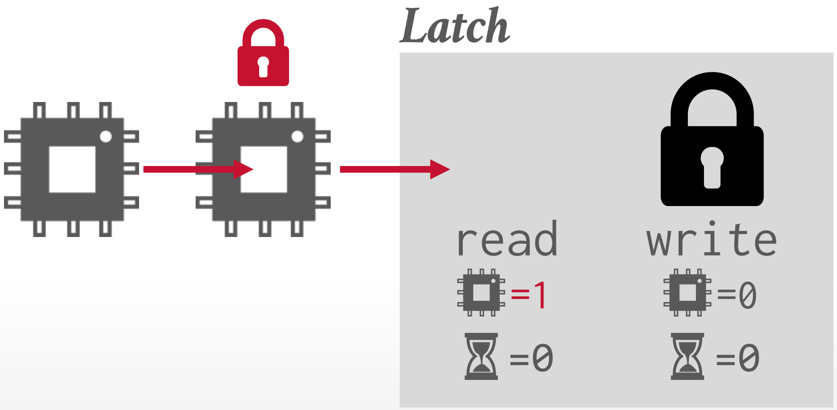
→ Allows for concurrent readers. Must manage read/write queues to avoid starvation.

→ Can be implemented on top of spinlocks.

→ Example: `std::shared_mutex` → `pthread_rwlock_t`

→ `pthread_mutex_t`

→ `pthread_cond_t`



LATCH IMPLEMENTATIONS

Approach #3: Reader-Writer Latches

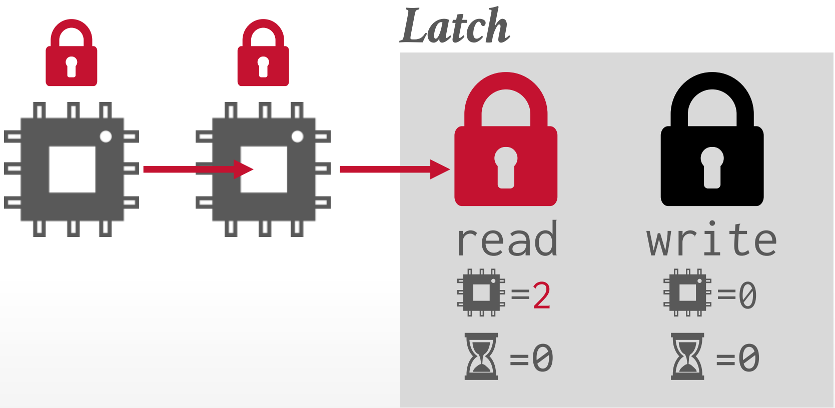
→ Allows for concurrent readers. Must manage read/write queues to avoid starvation.

→ Can be implemented on top of spinlocks.

→ Example: **`std::shared_mutex`** → **`pthread_rwlock_t`**

→ **`pthread_mutex_t`**

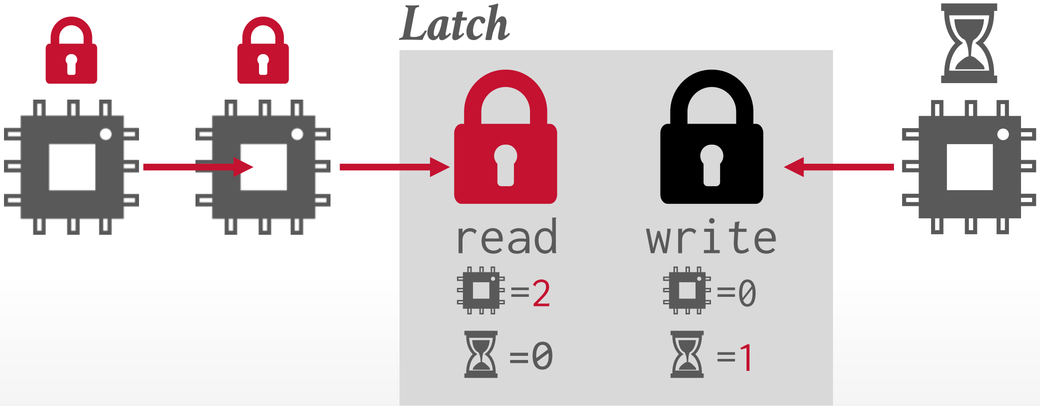
→ **`pthread_cond_t`**



LATCH IMPLEMENTATIONS

Approach #3: Reader-Writer Latches

- → Allows for concurrent readers. Must manage read/write queues to avoid starvation.
- → Can be implemented on top of spinlocks.
- → Example: **`std::shared_mutex`** → **`pthread_rwlock_t`** → **`pthread_mutex_t`**, **`pthread_cond_t`**



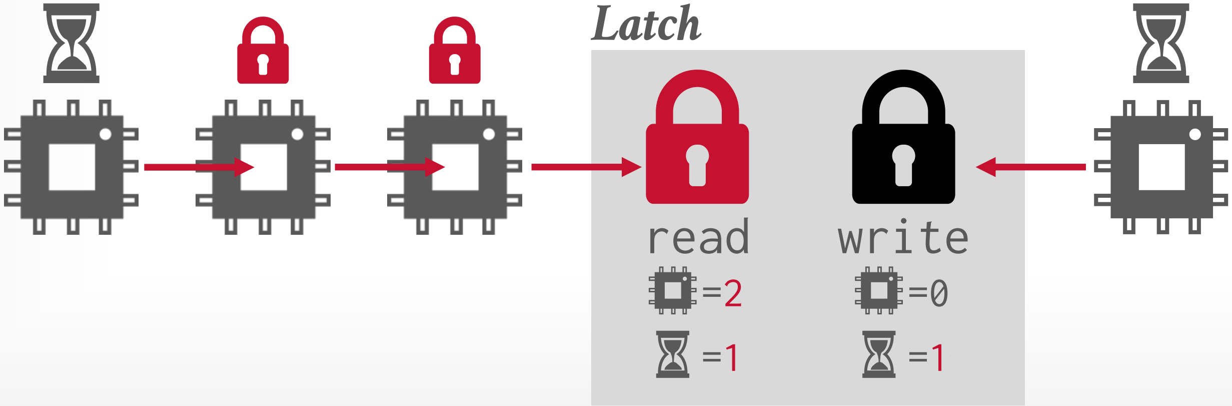
LATCH IMPLEMENTATIONS

Approach #3: Reader-Writer Latches

→ Allows for concurrent readers. Must manage read/write queues to avoid starvation.

→ Can be implemented on top of spinlocks.

→ Example: `std::shared_mutex` → `pthread_rwlock_t` → `pthread_mutex_t`, `pthread_cond_t`



HASH TABLE LATCHING

哈希表的优点: 不会发生死锁, 因为所有线程都朝相同的方向执行.

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

Latch 的粒度: 页面锁和槽锁.

Approach #1: Page/Block Latches

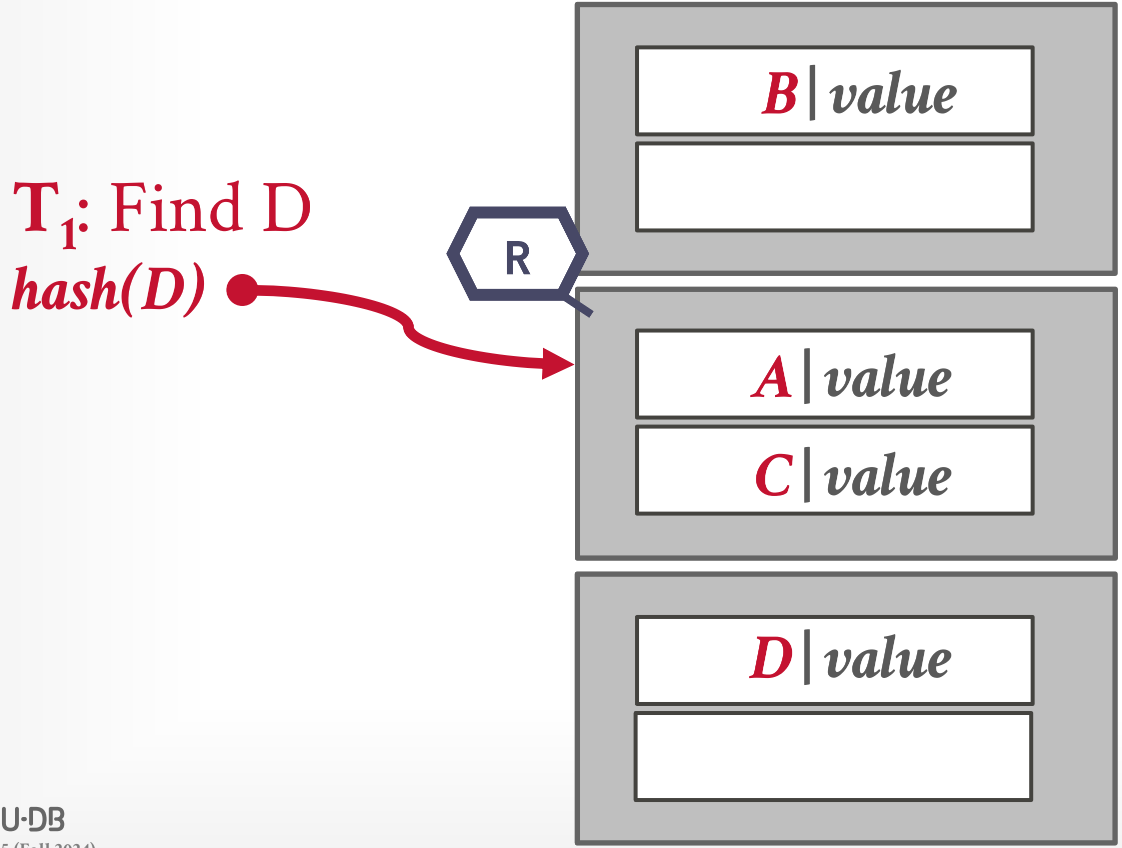
- → Each page/block has its own reader-writer latch that protects its entire contents.
- → Threads acquire either a read or write latch before they access a page/block.

Approach #2: Slot Latches

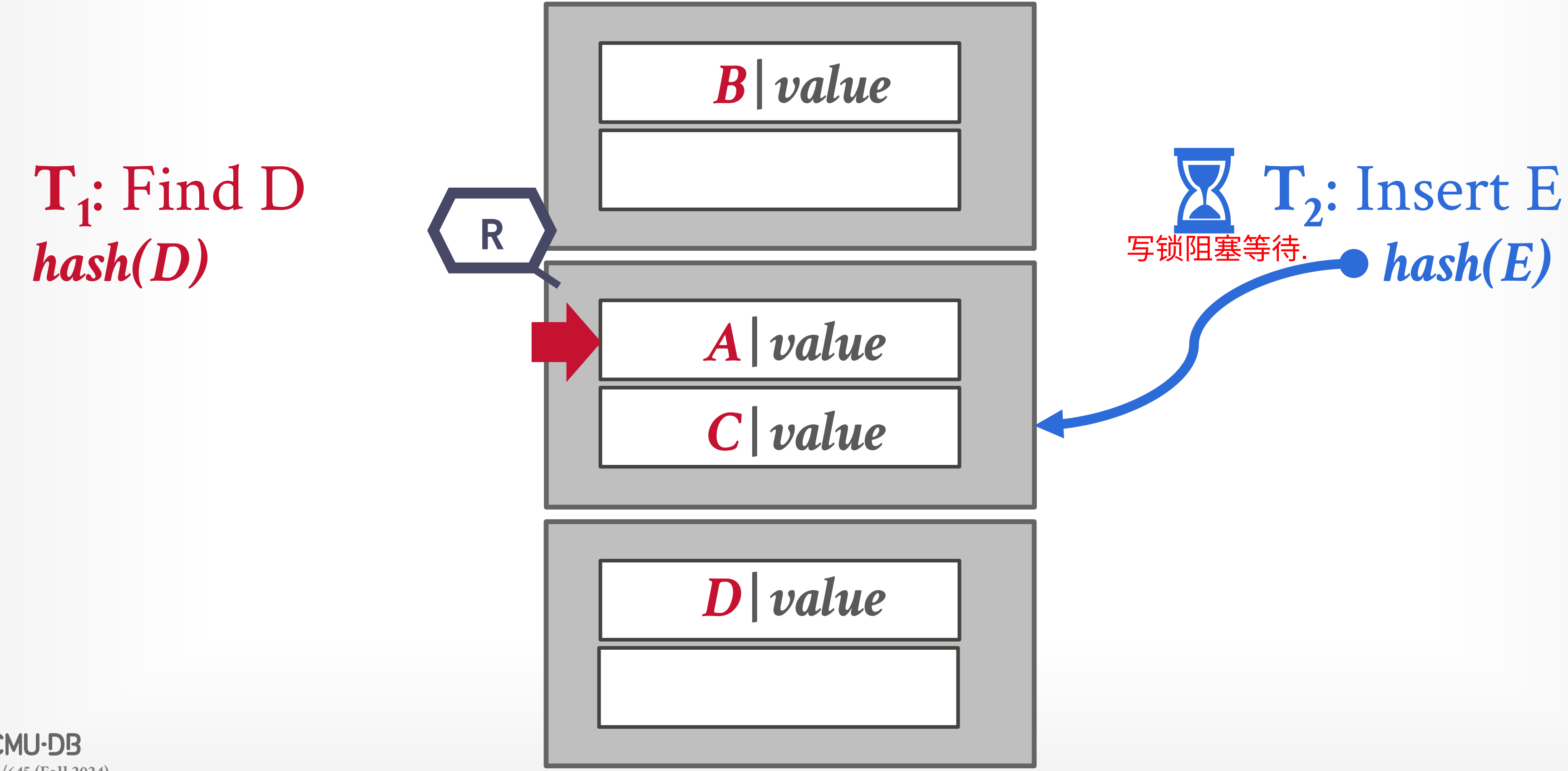
锁的粒度更小.

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

HASH TABLE: PAGE/BLOCK LATCHES

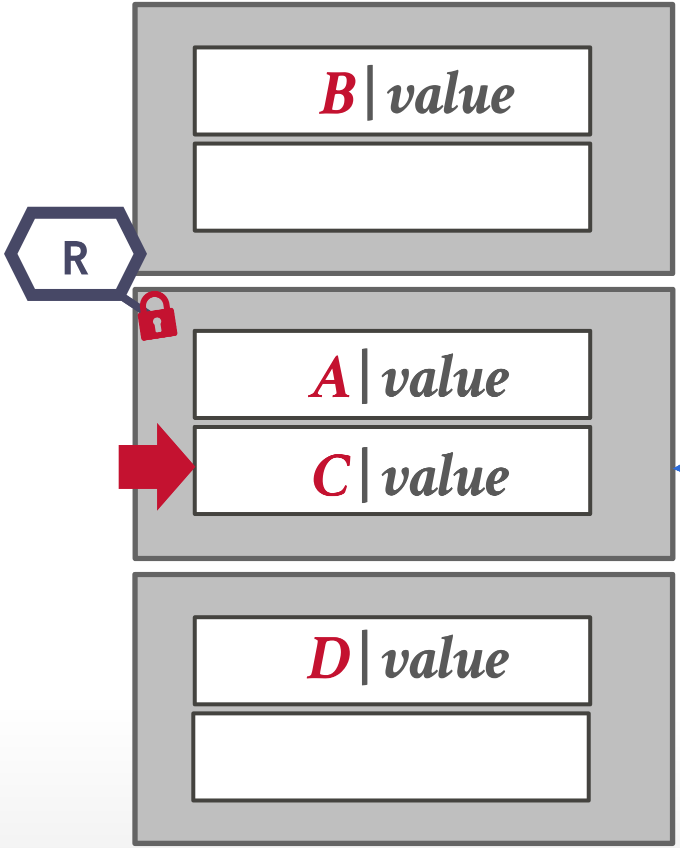


HASH TABLE: PAGE/BLOCK LATCHES



HASH TABLE: PAGE/BLOCK LATCHES

T_1: Find D
hash(D)



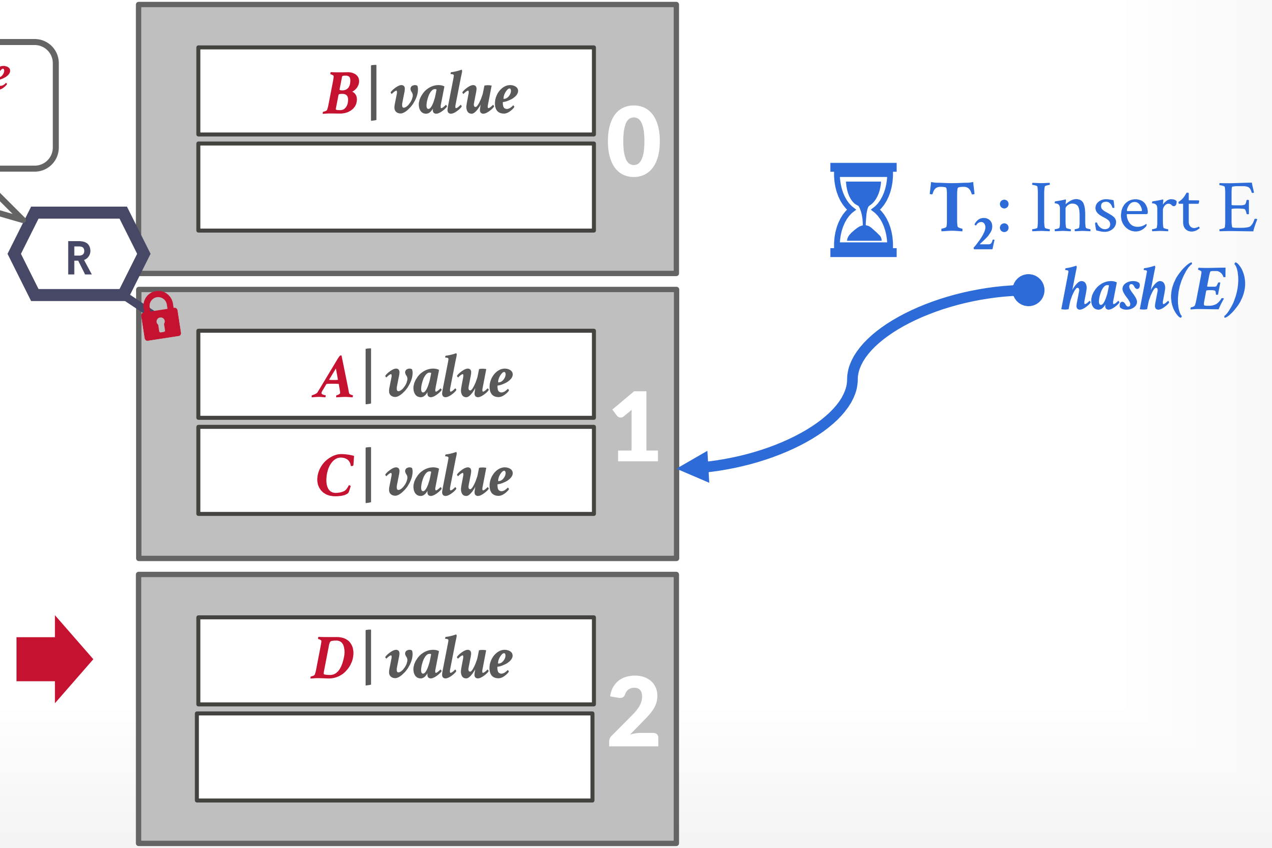
T_2: Insert E
hash(E)

HASH TABLE: PAGE/BLOCK LATCHES

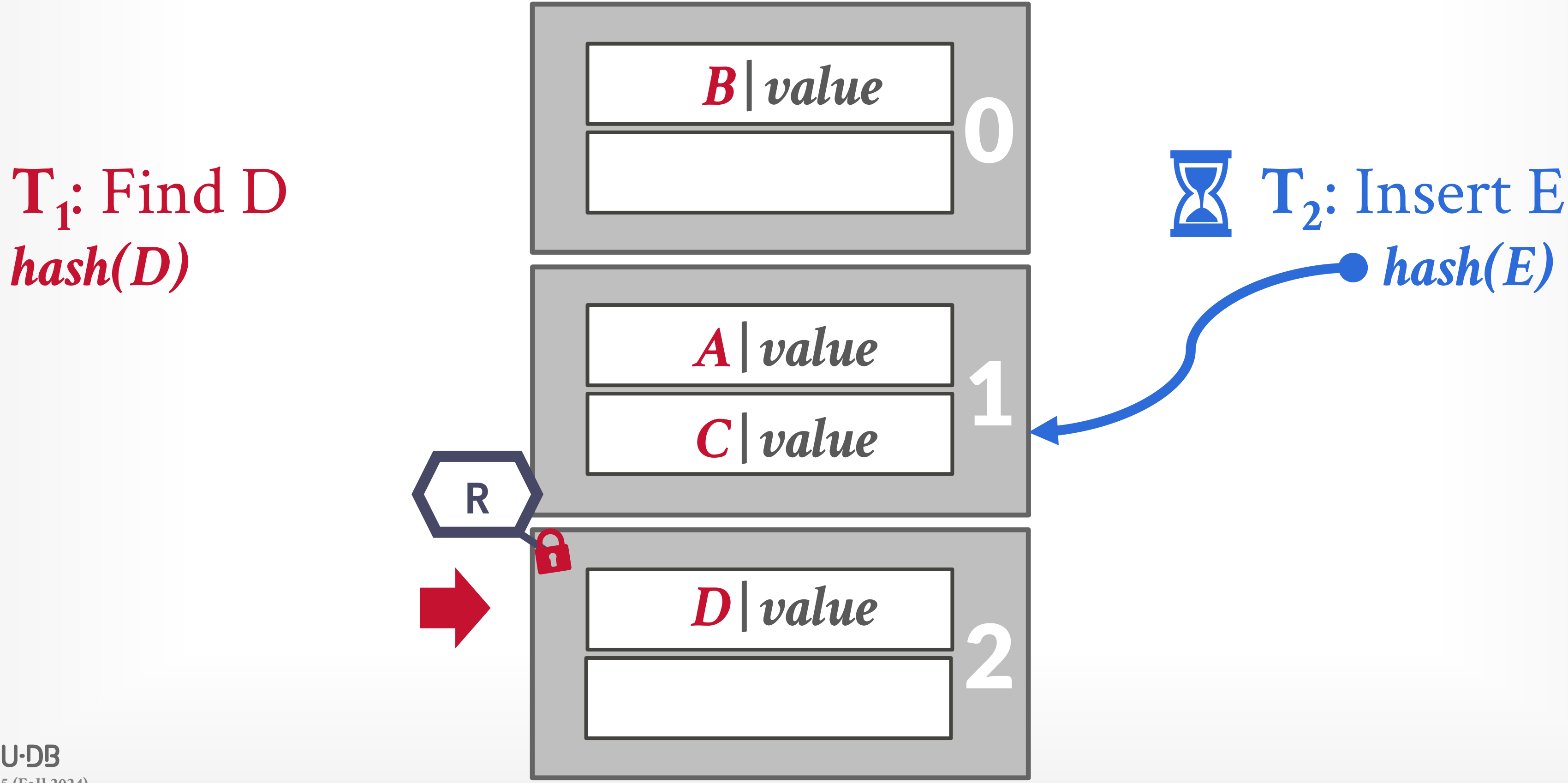
It's safe to release the latch on Page #1.

T_1: Find D

hash(D)

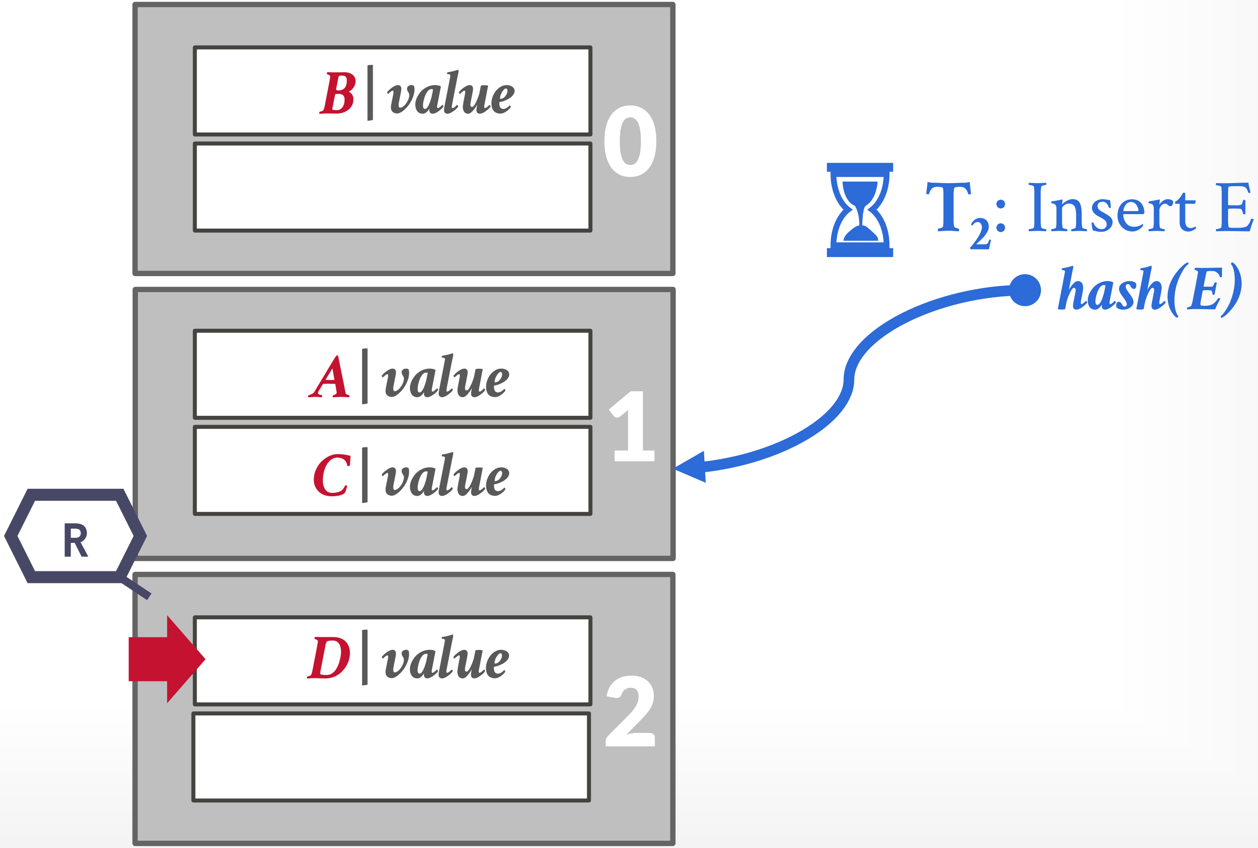


HASH TABLE: PAGE/BLOCK LATCHES



HASH TABLE: PAGE/BLOCK LATCHES

T$_1$: Find D
hash(D)

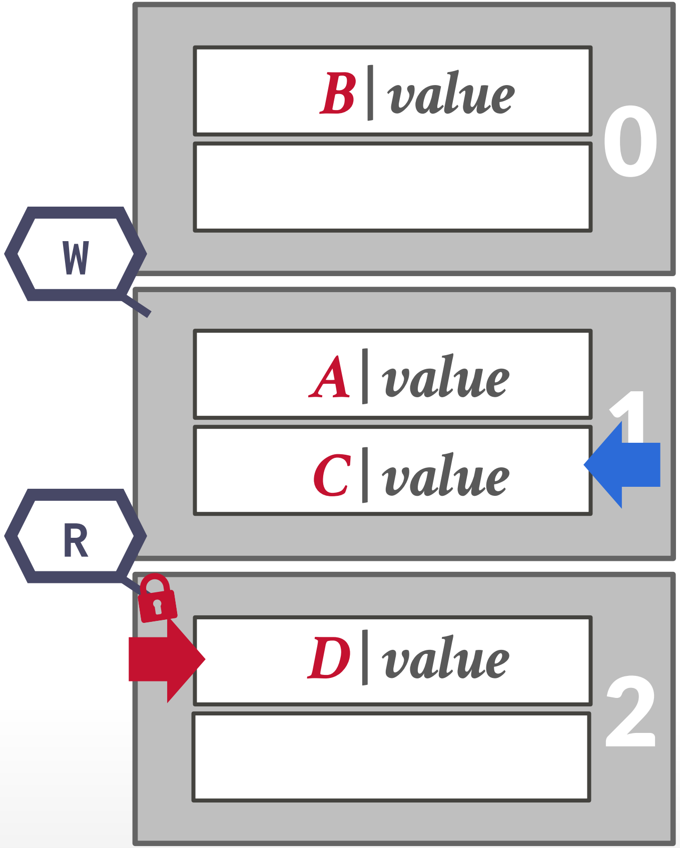


T$_2$: Insert E

HASH TABLE: PAGE/BLOCK LATCHES

T_1: Find D
hash(D)

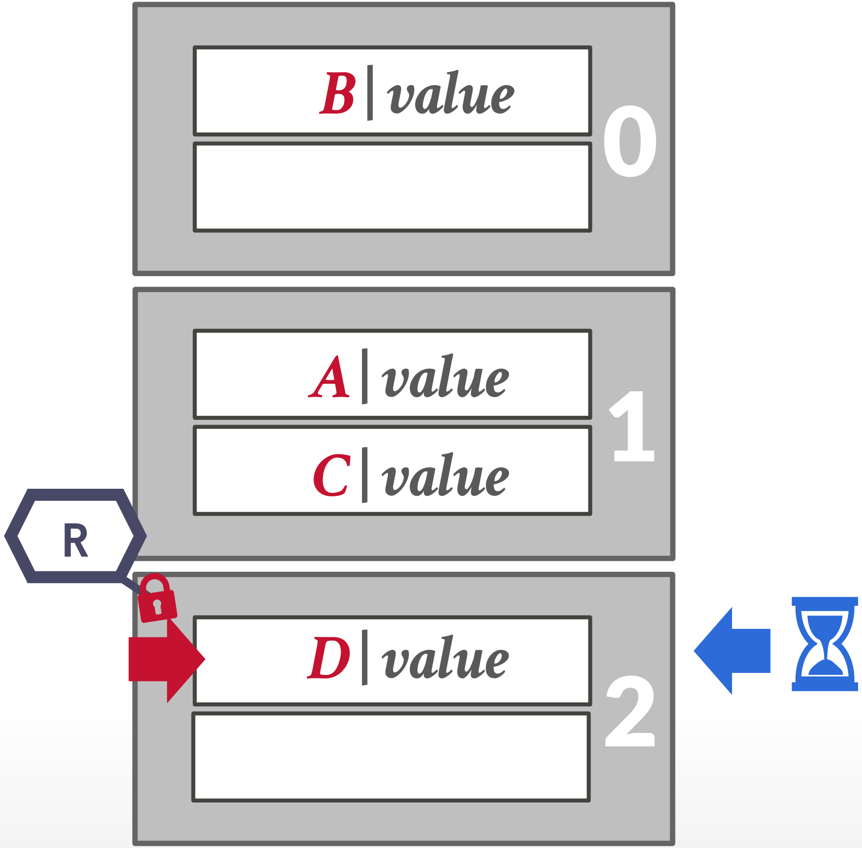
T_2: Insert E
hash(E)



HASH TABLE: PAGE/BLOCK LATCHES

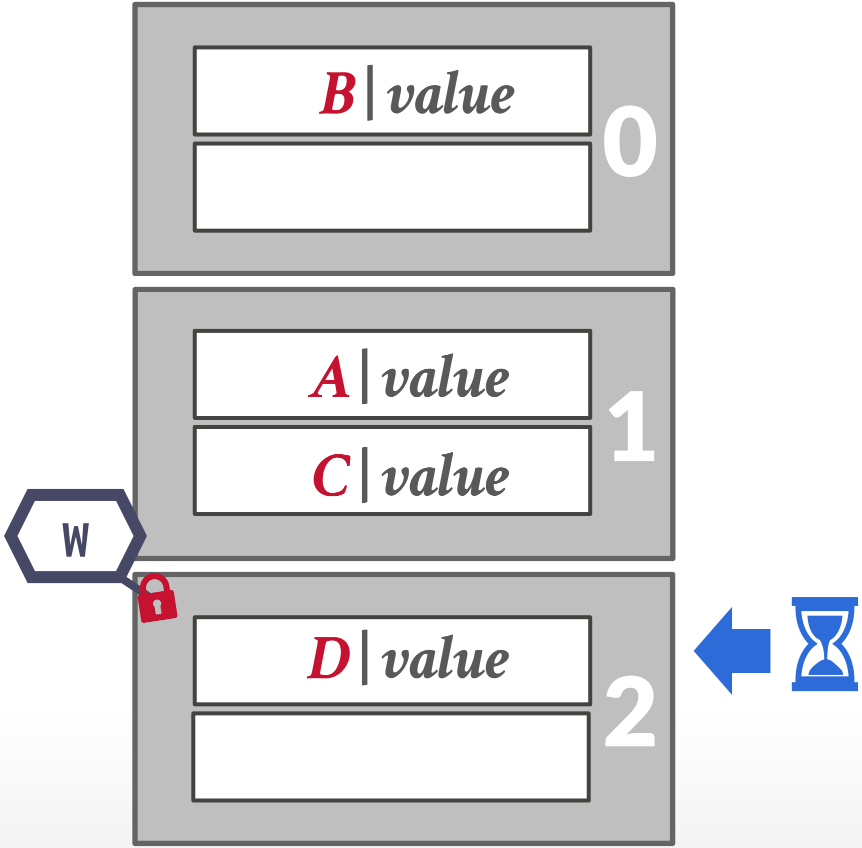
T_1: Find D
hash(D)

T_2: Insert E
hash(E)



HASH TABLE: PAGE/BLOCK LATCHES

T$_1$: Find D
hash(D)

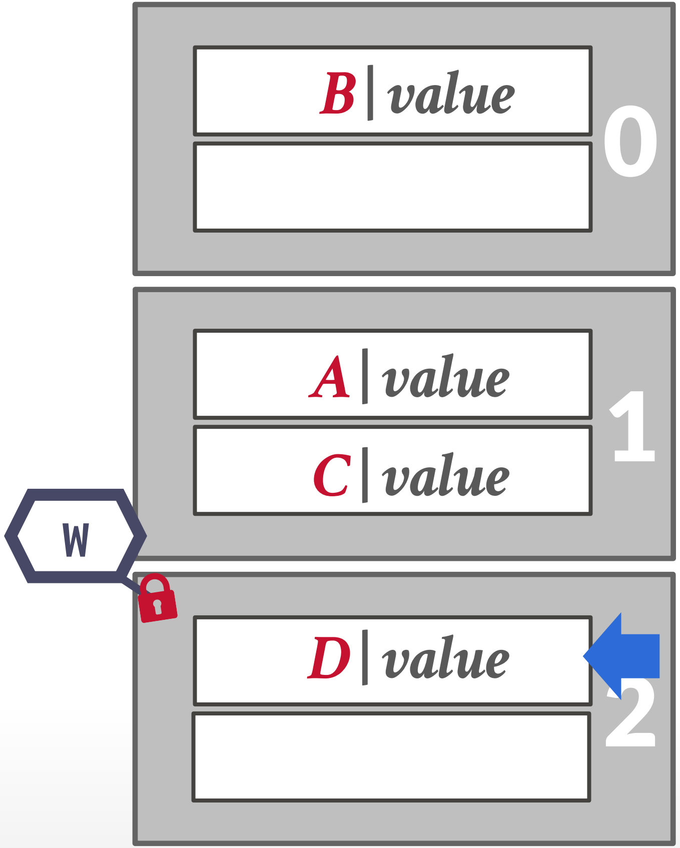


T$_2$: Insert E
hash(E)

HASH TABLE: PAGE/BLOCK LATCHES

T$_1$: Find D

hash(D)



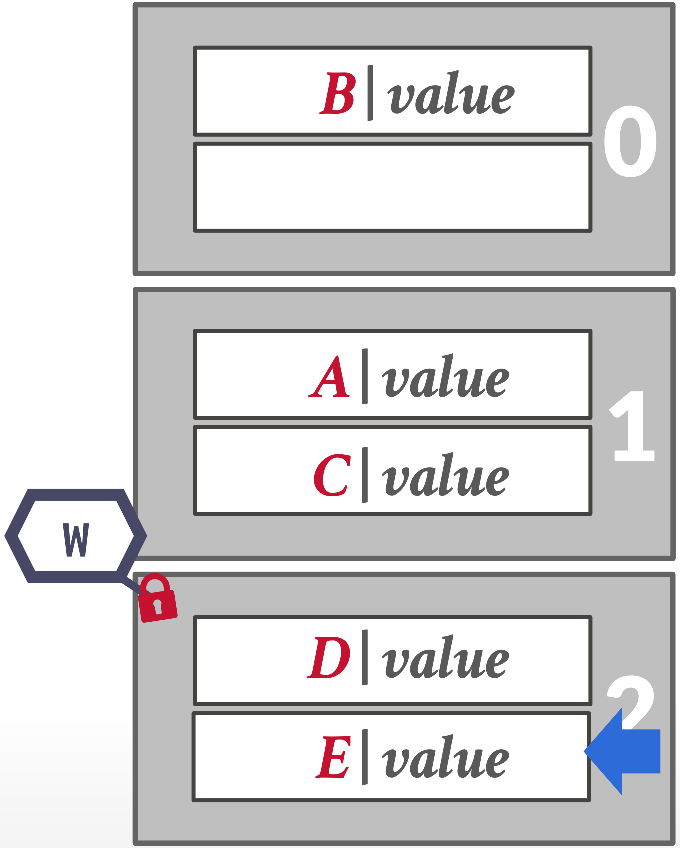
T$_2$: Insert E

hash(E)

HASH TABLE: PAGE/BLOCK LATCHES

T_1: Find D

hash(D)

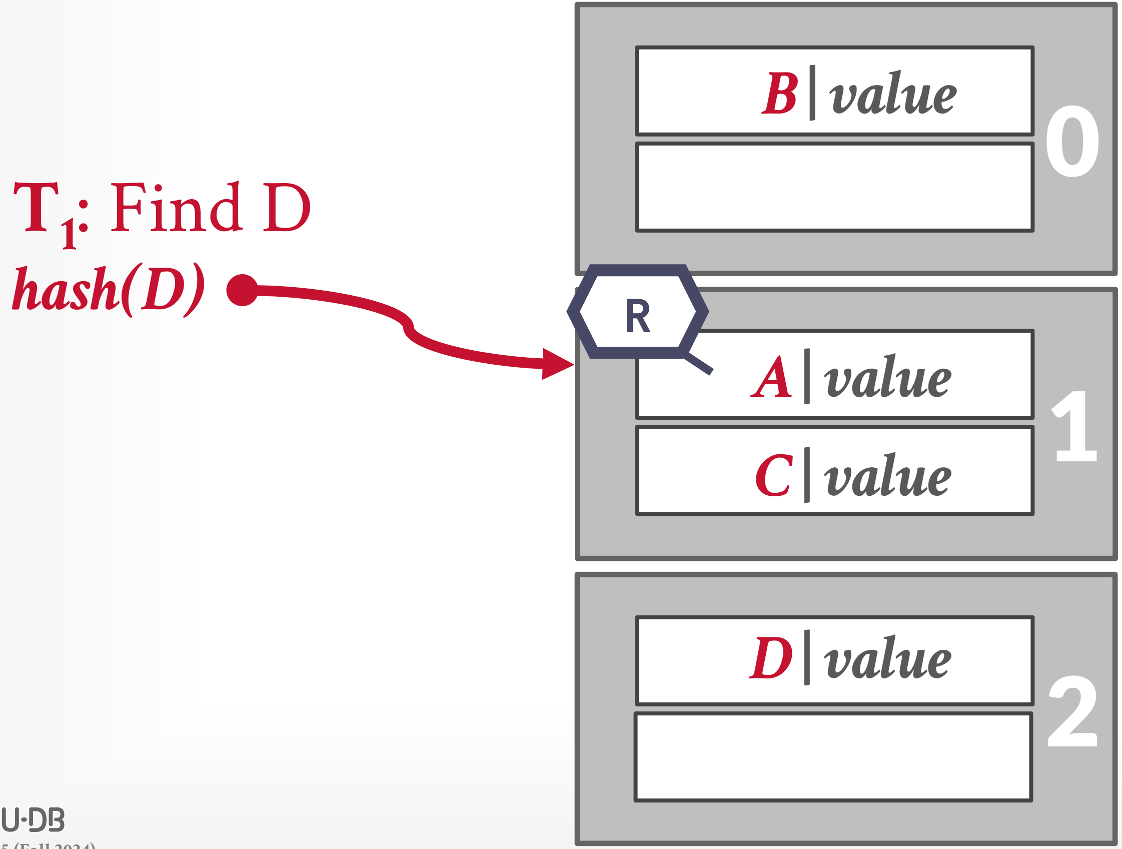


T_2: Insert E

hash(E)

HASH TABLE: SLOT LATCHES

T_1: Find D

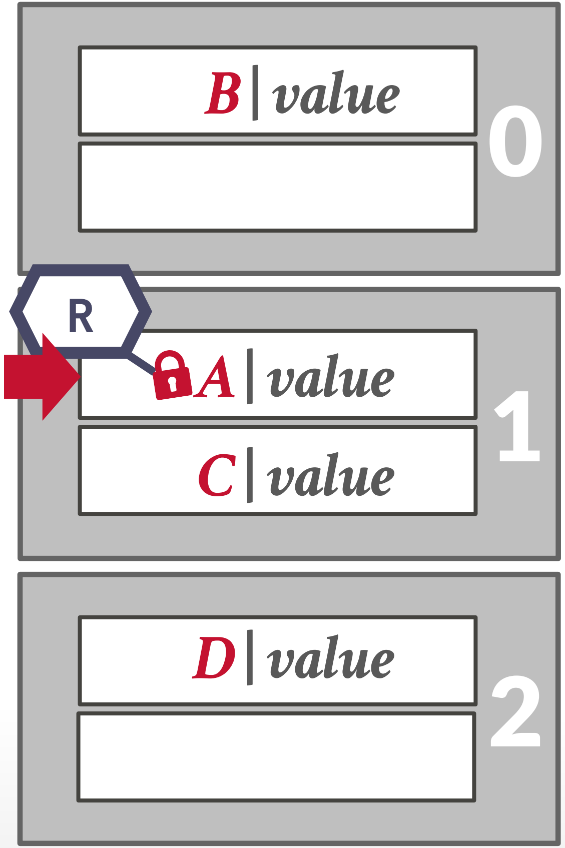


T_2: Insert E

hash(E)

HASH TABLE: SLOT LATCHES

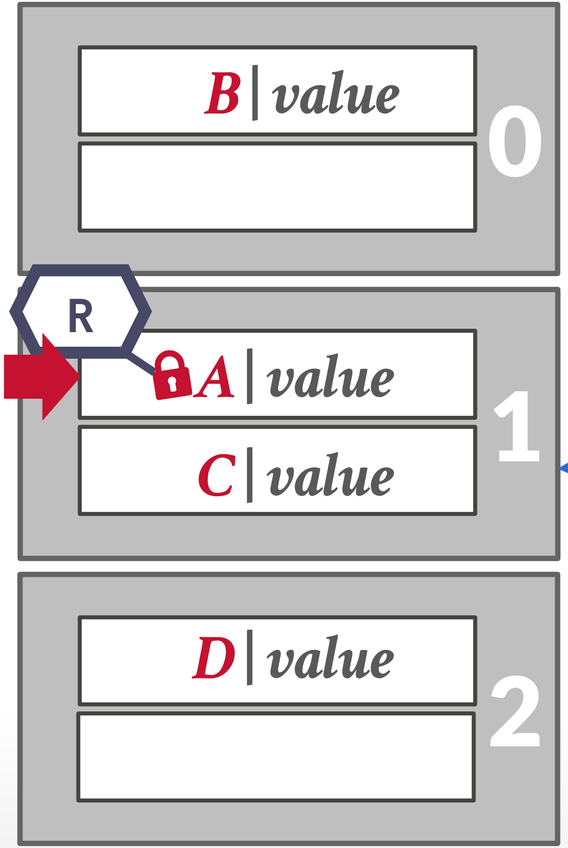
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

HASH TABLE: SLOT LATCHES

T_1: Find D
hash(D)

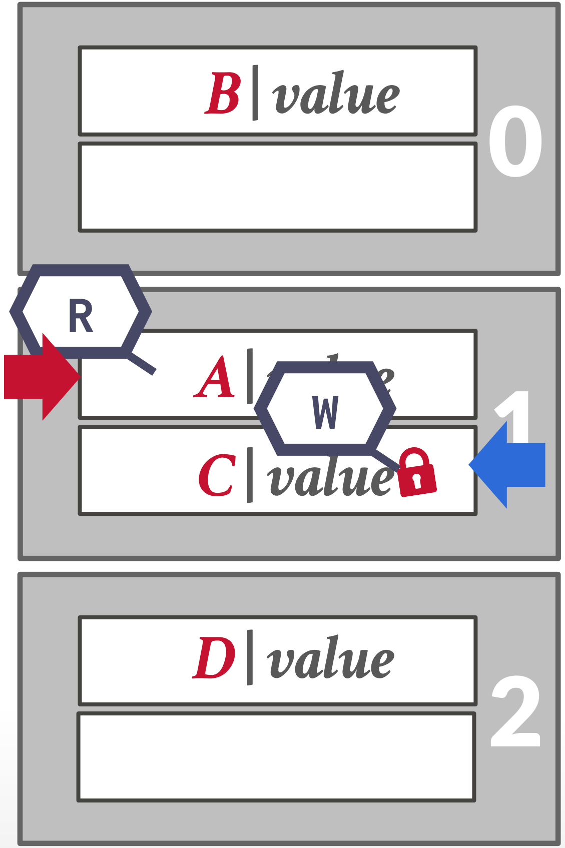


T_2: Insert E
hash(E)

HASH TABLE: SLOT LATCHES

T$_1$: Find D

hash(D)

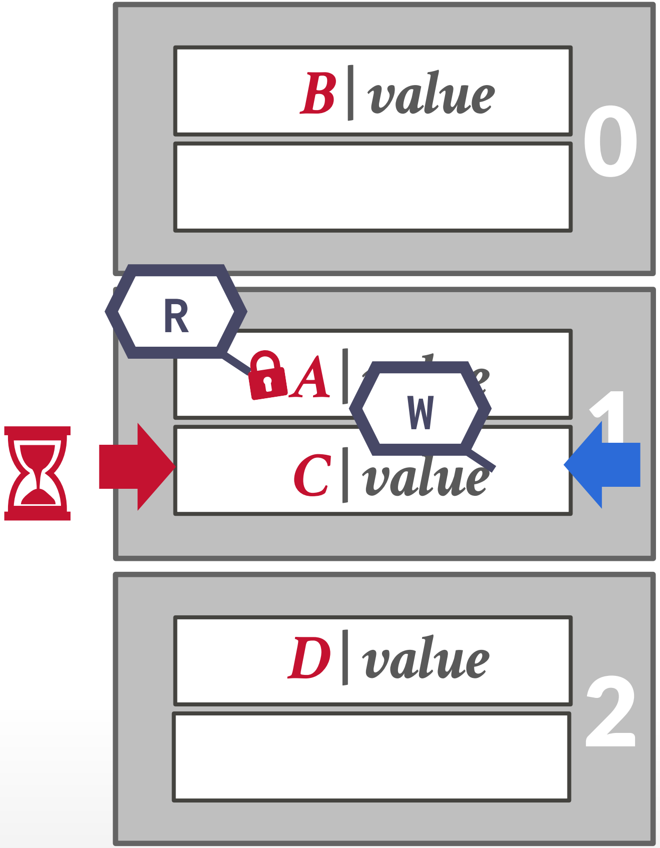


T$_2$: Insert E

hash(E)

HASH TABLE: SLOT LATCHES

T$_1$: Find D
hash(D)



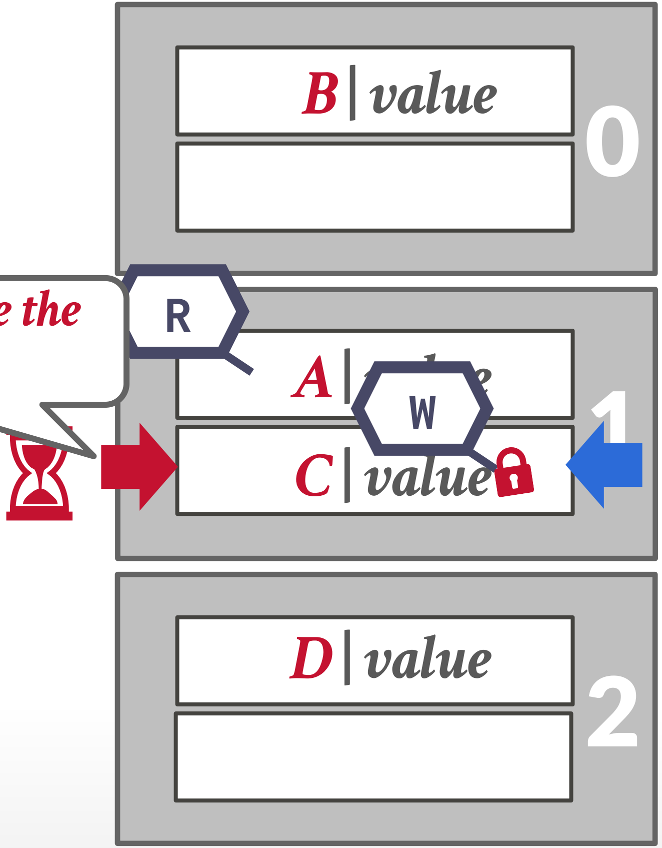
T$_2$: Insert E
hash(E)

HASH TABLE: SLOT LATCHES

T_1: Find D

has

It's safe to release the latch on A



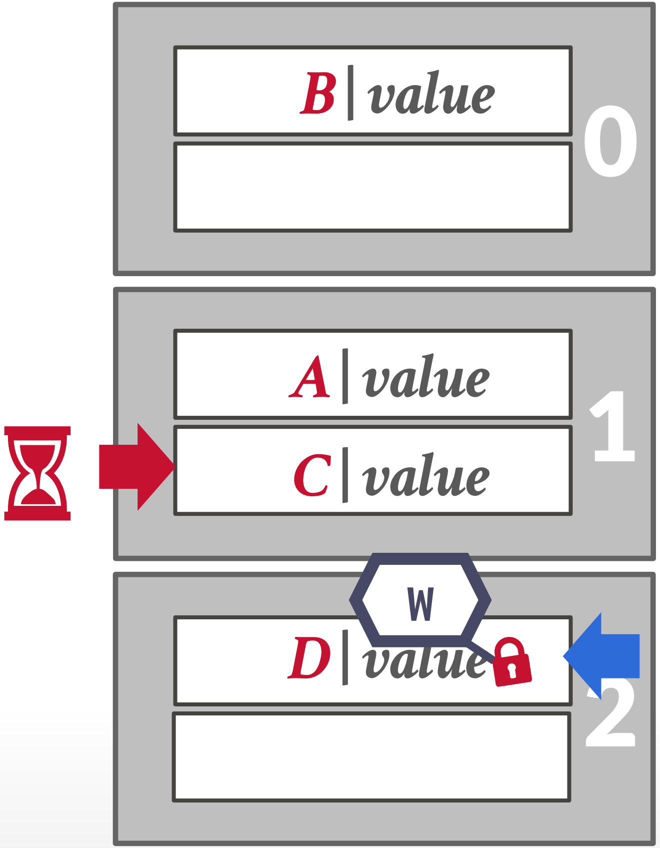
T_2: Insert E

hash(E)

HASH TABLE: SLOT LATCHES

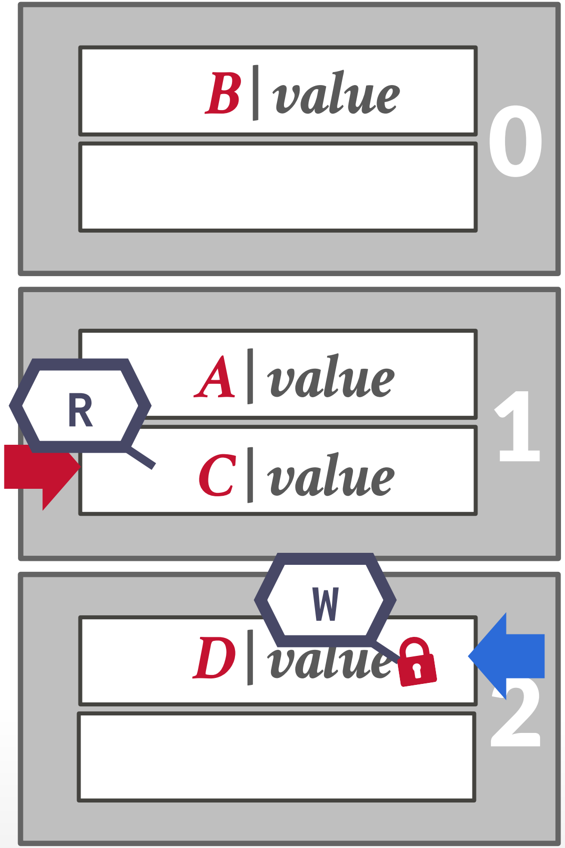
T_1: Find D
hash(D)

T_2: Insert E
hash(E)



HASH TABLE: SLOT LATCHES

T$_1$: Find D
hash(D)

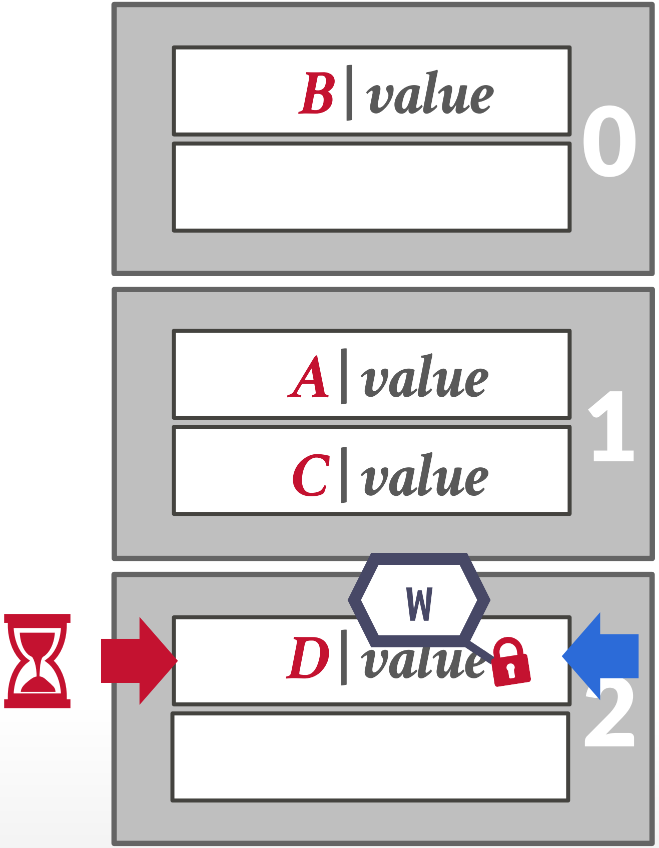


T$_2$: Insert E
hash(E)

HASH TABLE: SLOT LATCHES

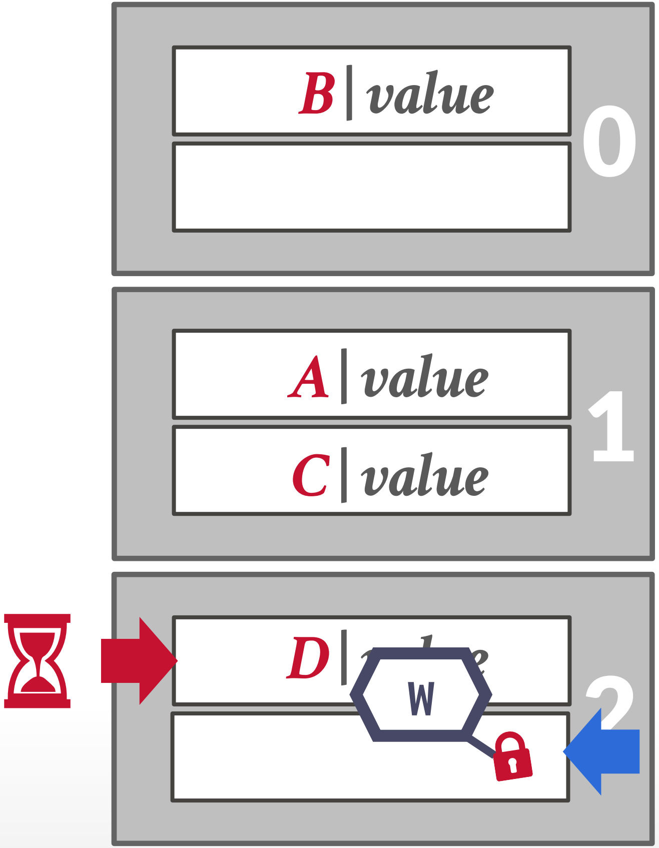
T$_1$: Find D
hash(D)

T$_2$: Insert E
hash(E)



HASH TABLE: SLOT LATCHES

T$_1$: Find D
hash(D)



T$_2$: Insert E
hash(E)

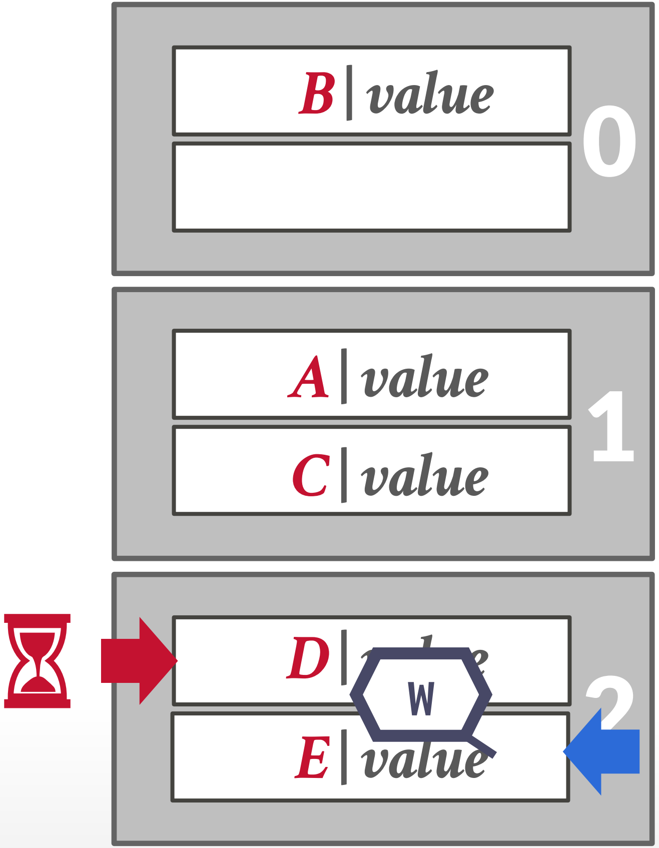
HASH TABLE: SLOT LATCHES

T$_1$: Find D

hash(D)

T$_2$: Insert E

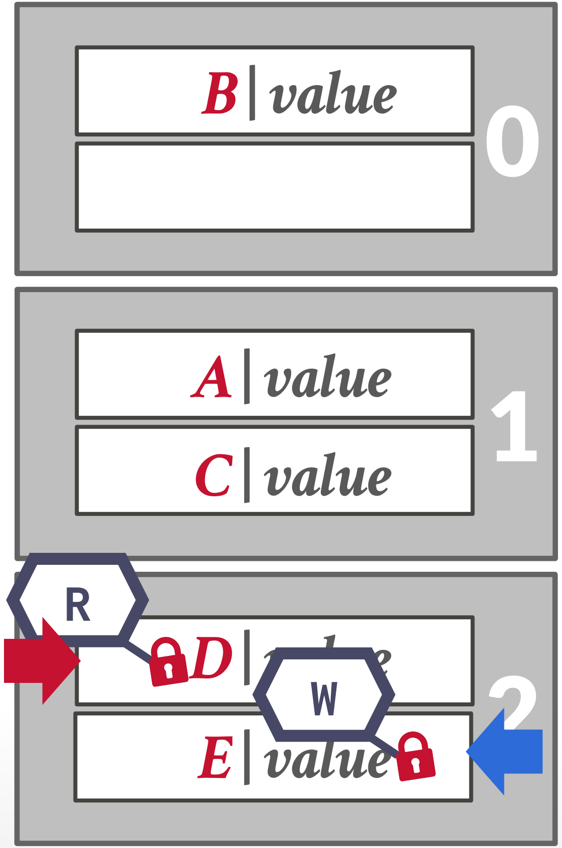
hash(E)



HASH TABLE: SLOT LATCHES

T$_1$: Find D

hash(D)



T$_2$: Insert E

hash(E)

B+TREE CONCURRENCY CONTROL

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

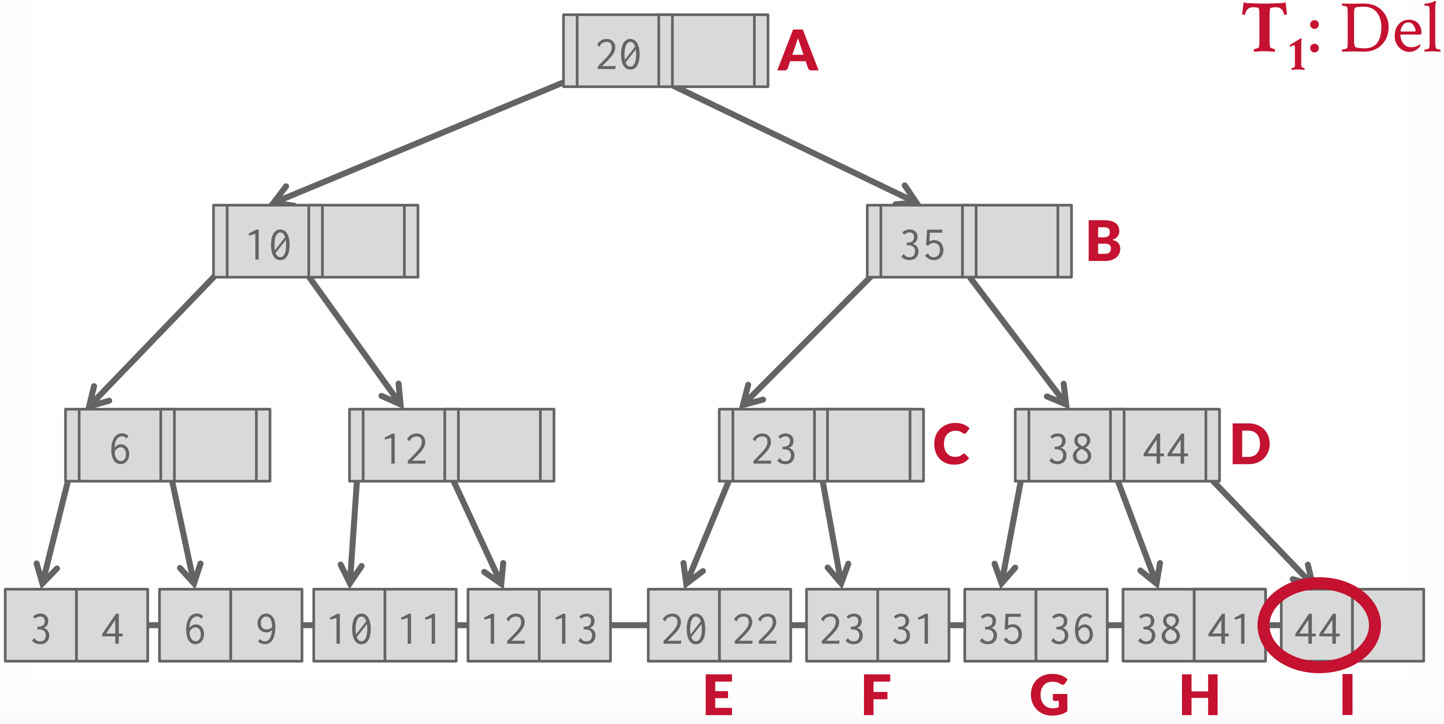
We need to protect against two types of problems:

→ Threads trying to modify the contents of a node at the same time.

→ One thread traversing the tree while another thread splits/merges nodes.

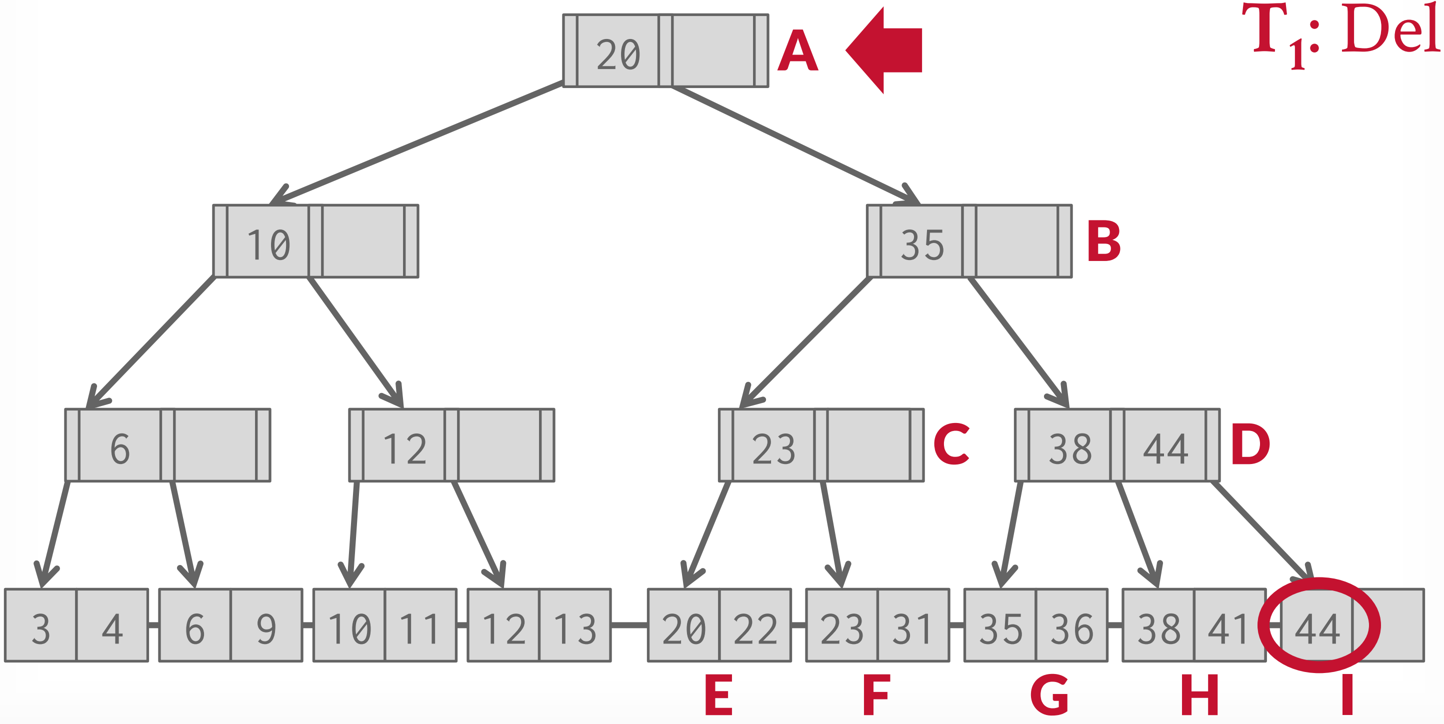
B+TREE MULTI-THREADED EXAMPLE

T_1: Delete 44



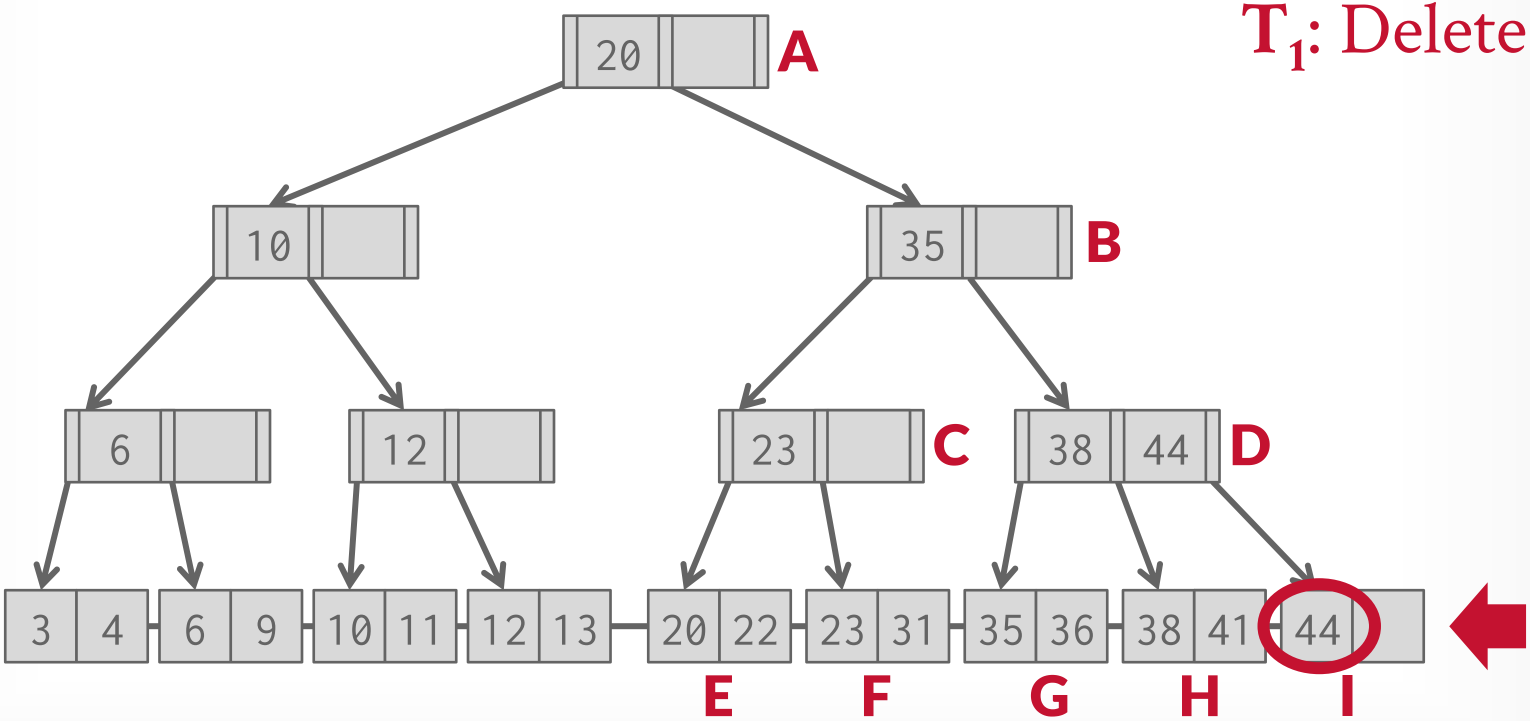
B+TREE MULTI-THREADED EXAMPLE

T_1: Delete 44



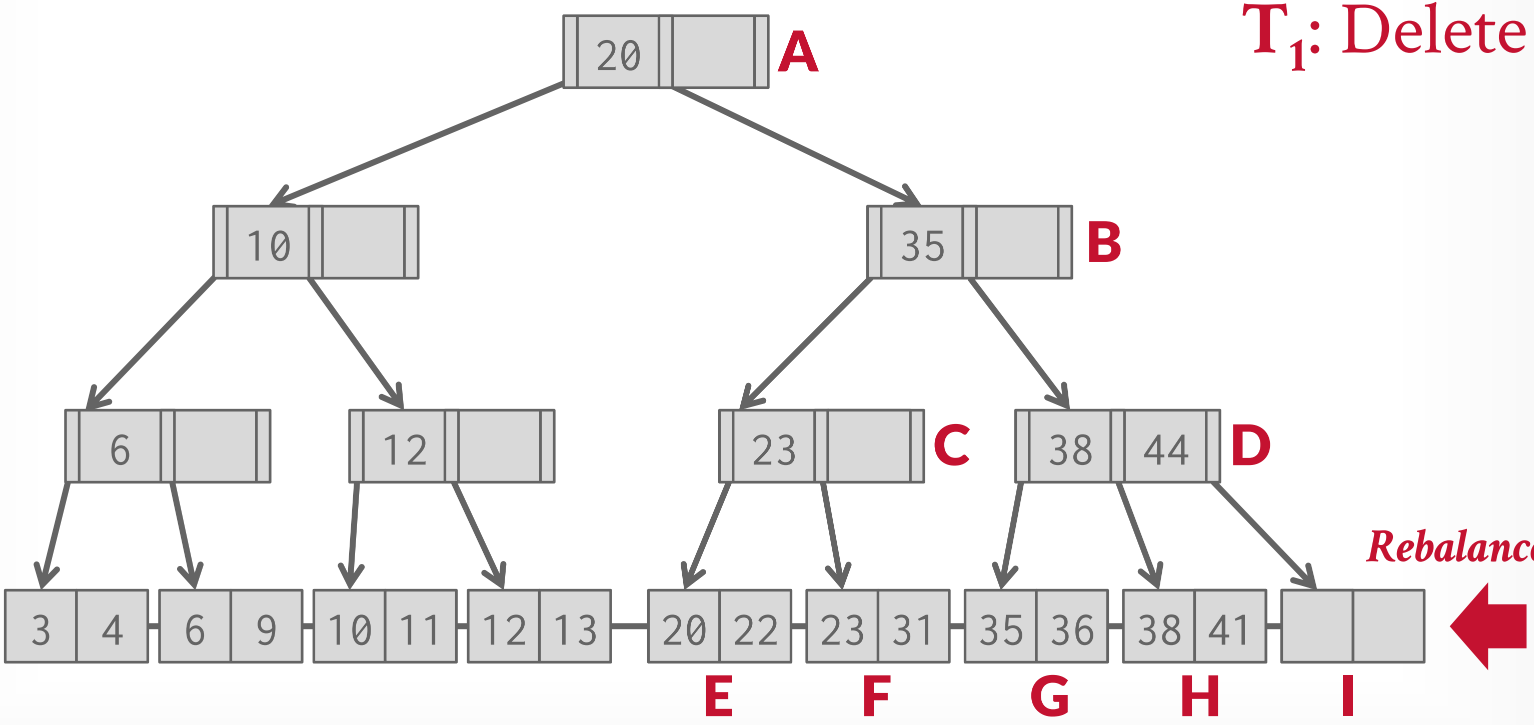
B+TREE MULTI-THREADED EXAMPLE

T_1: Delete 44



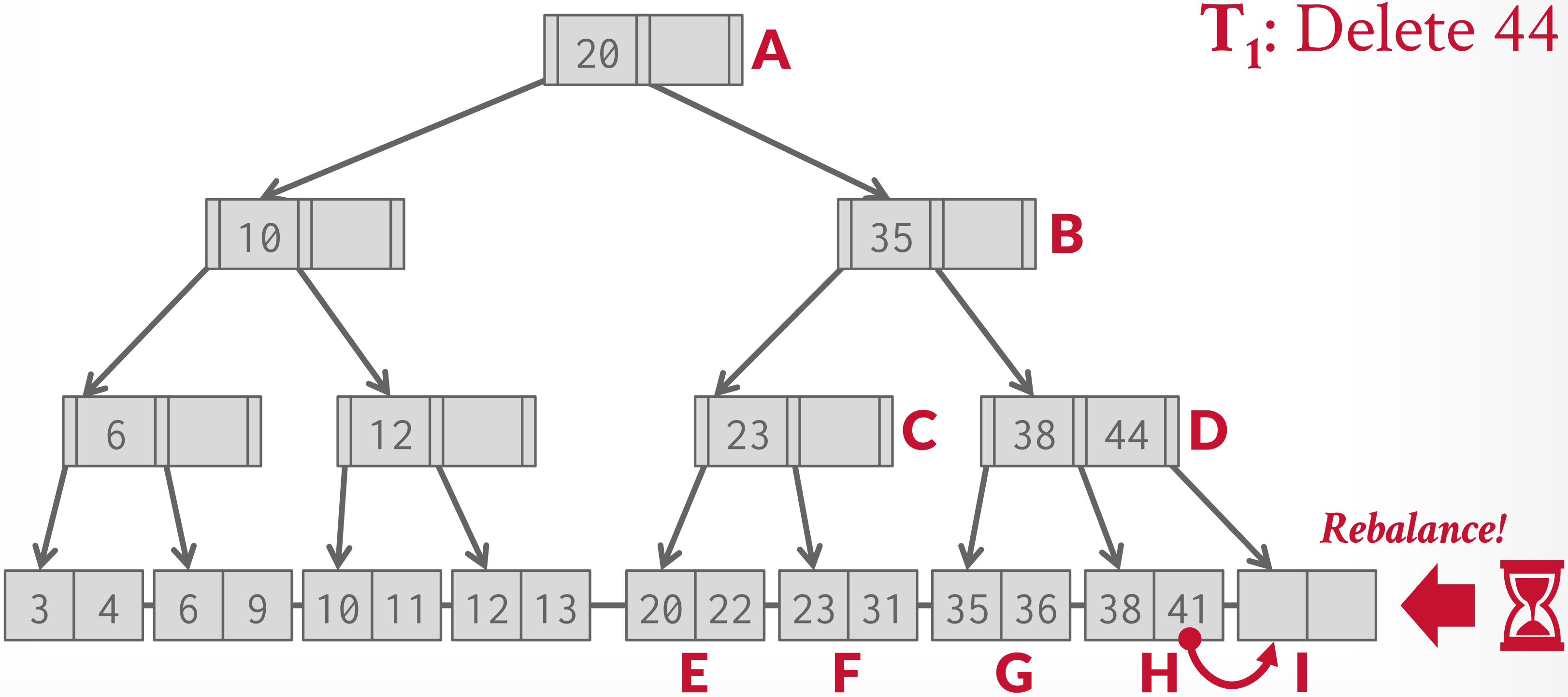
B+TREE MULTI-THREADED EXAMPLE

T_1: Delete 44



B+TREE MULTI-THREADED EXAMPLE

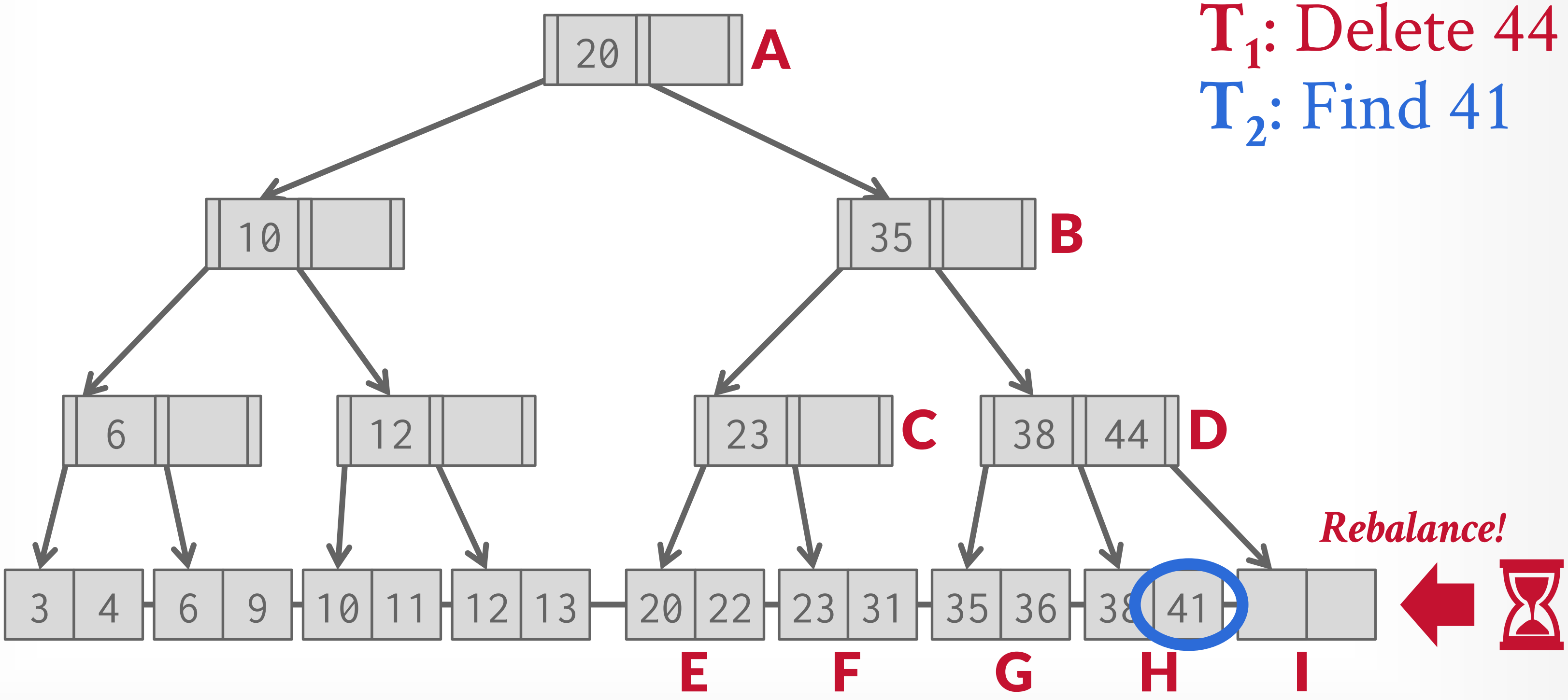
T$_1$: Delete 44



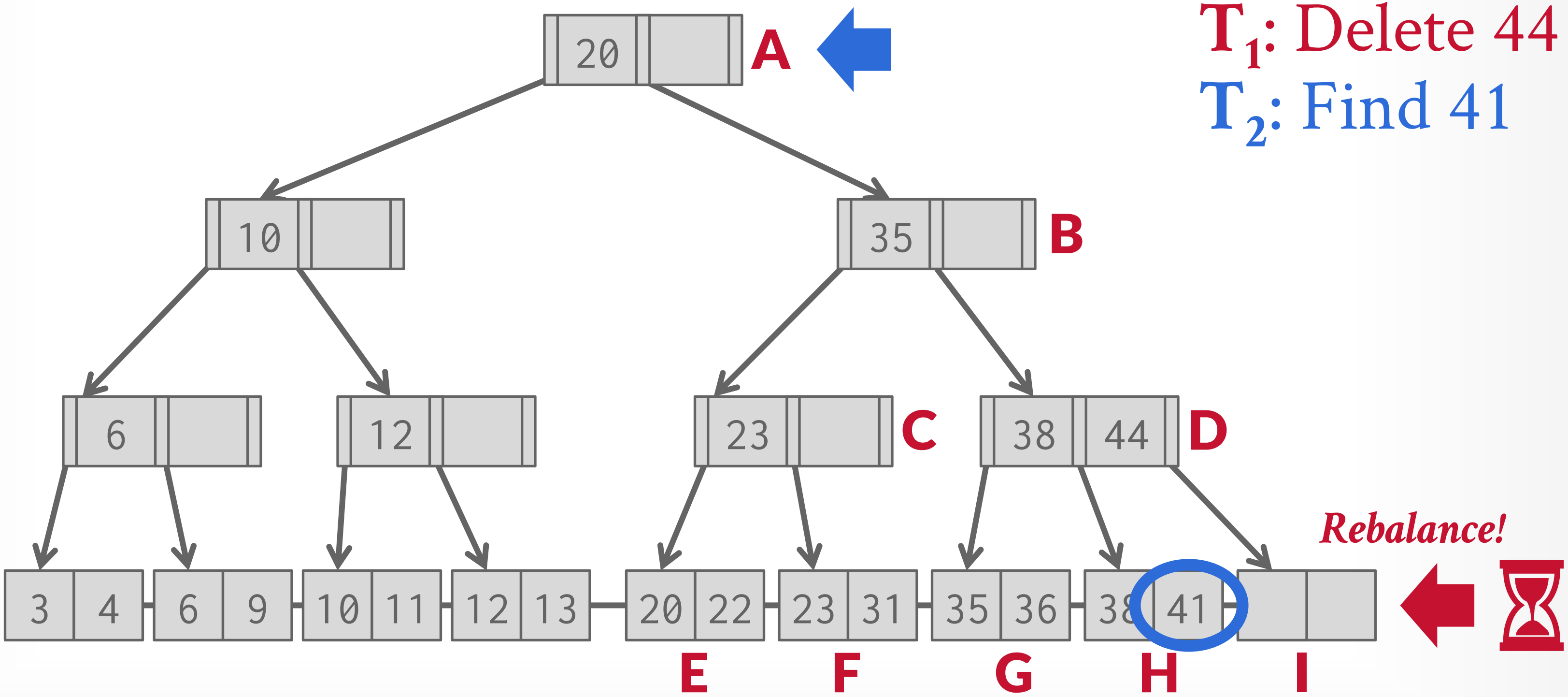
B+TREE MULTI-THREADED EXAMPLE

T_1: Delete 44

T_2: Find 41



B+TREE MULTI-THREADED EXAMPLE



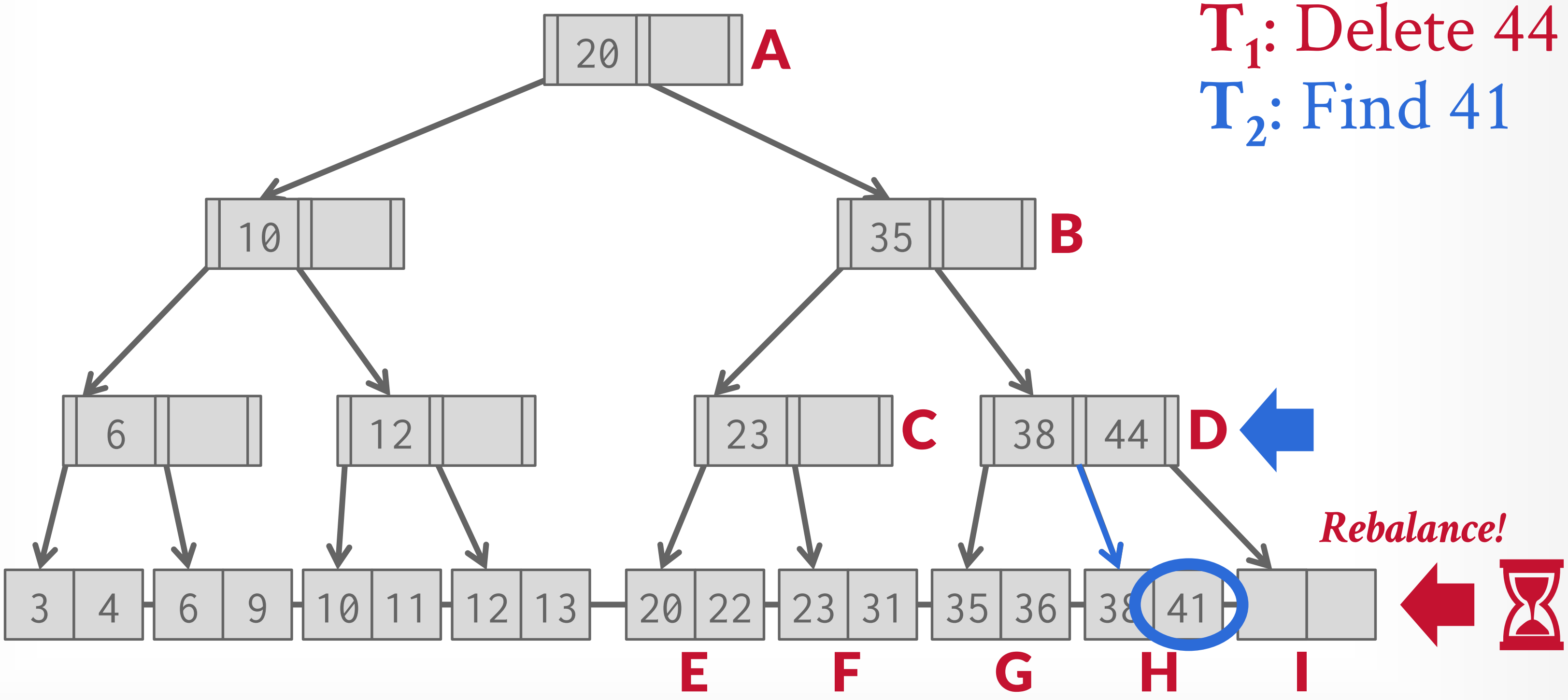
T_1: Delete 44

T_2: Find 41

B+TREE MULTI-THREADED EXAMPLE

T_1: Delete 44

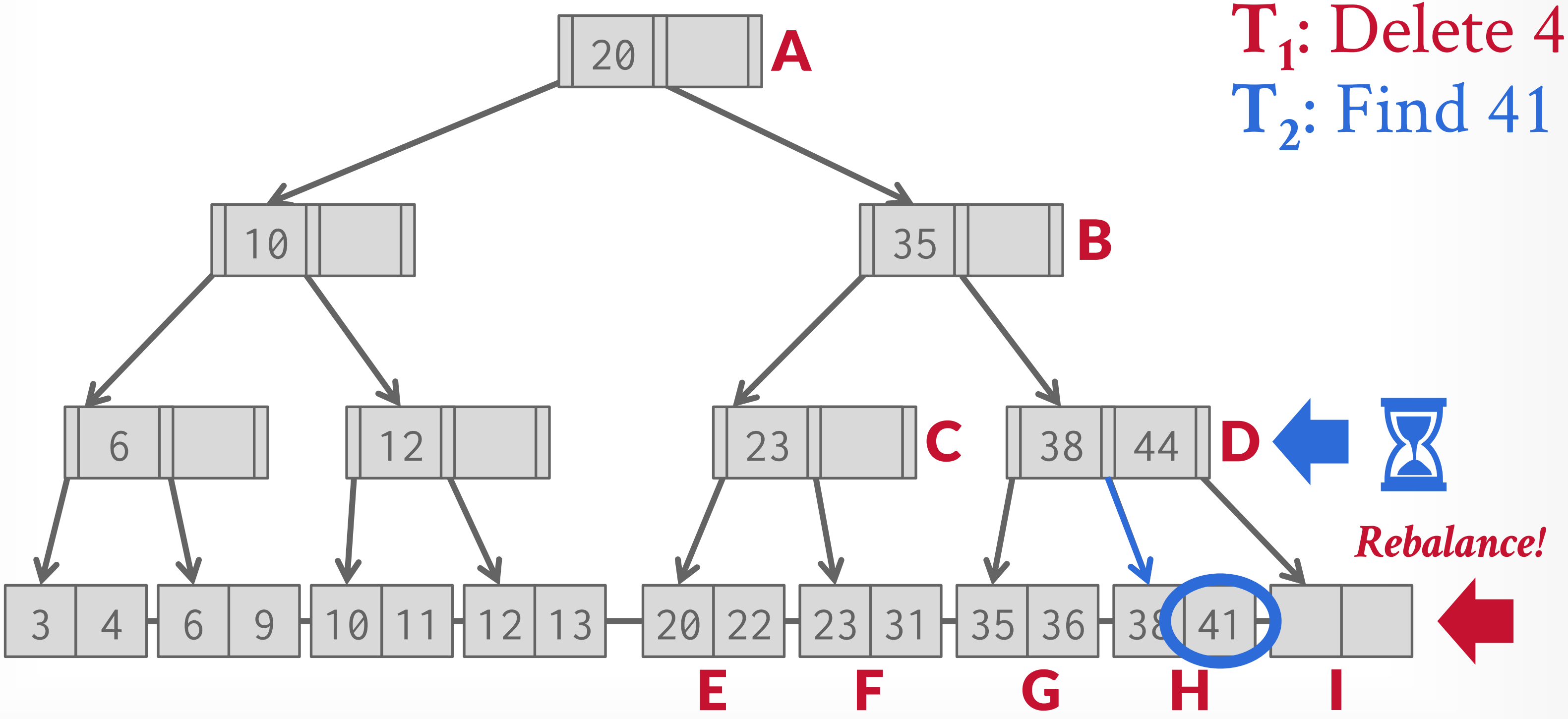
T_2: Find 41



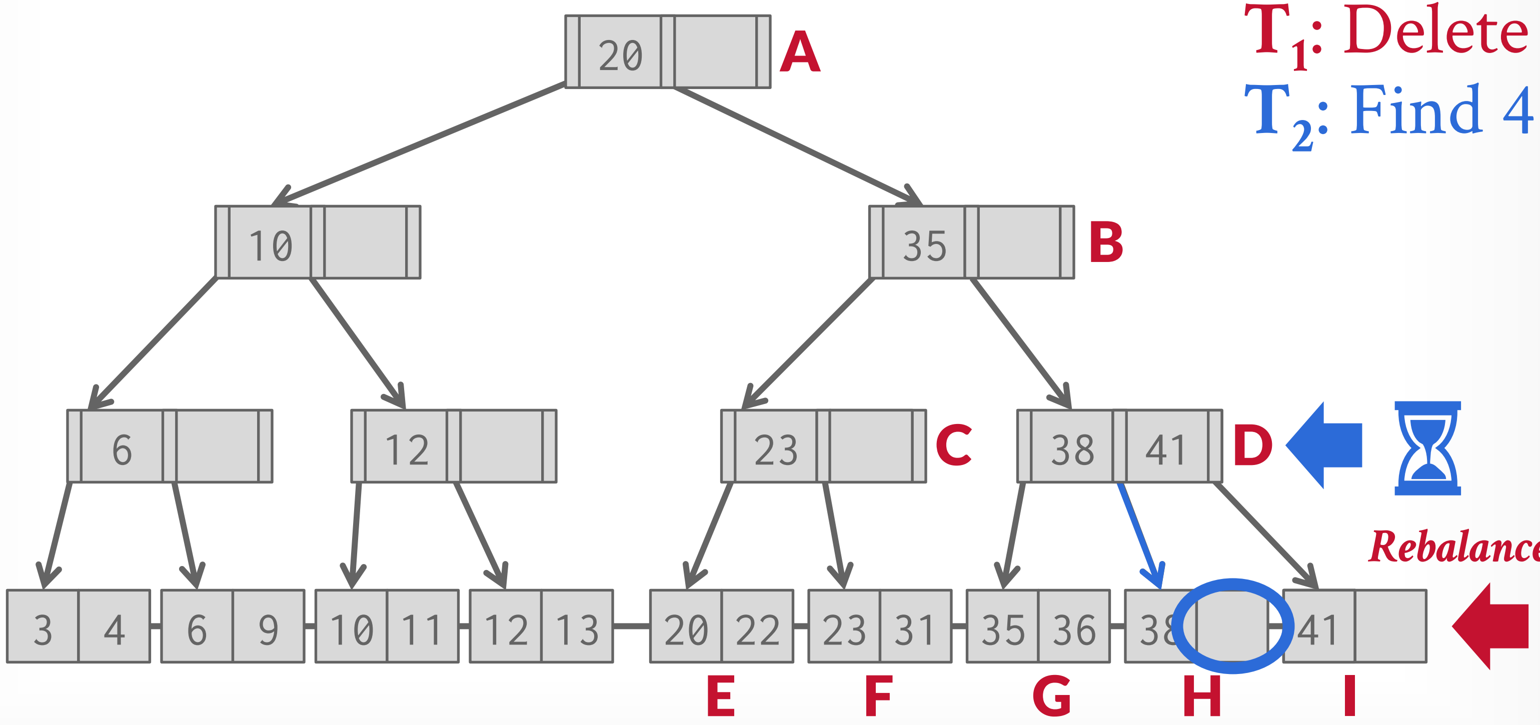
B+TREE MULTI-THREADED EXAMPLE

T_1: Delete 44

T_2: Find 41



B+TREE MULTI-THREADED EXAMPLE



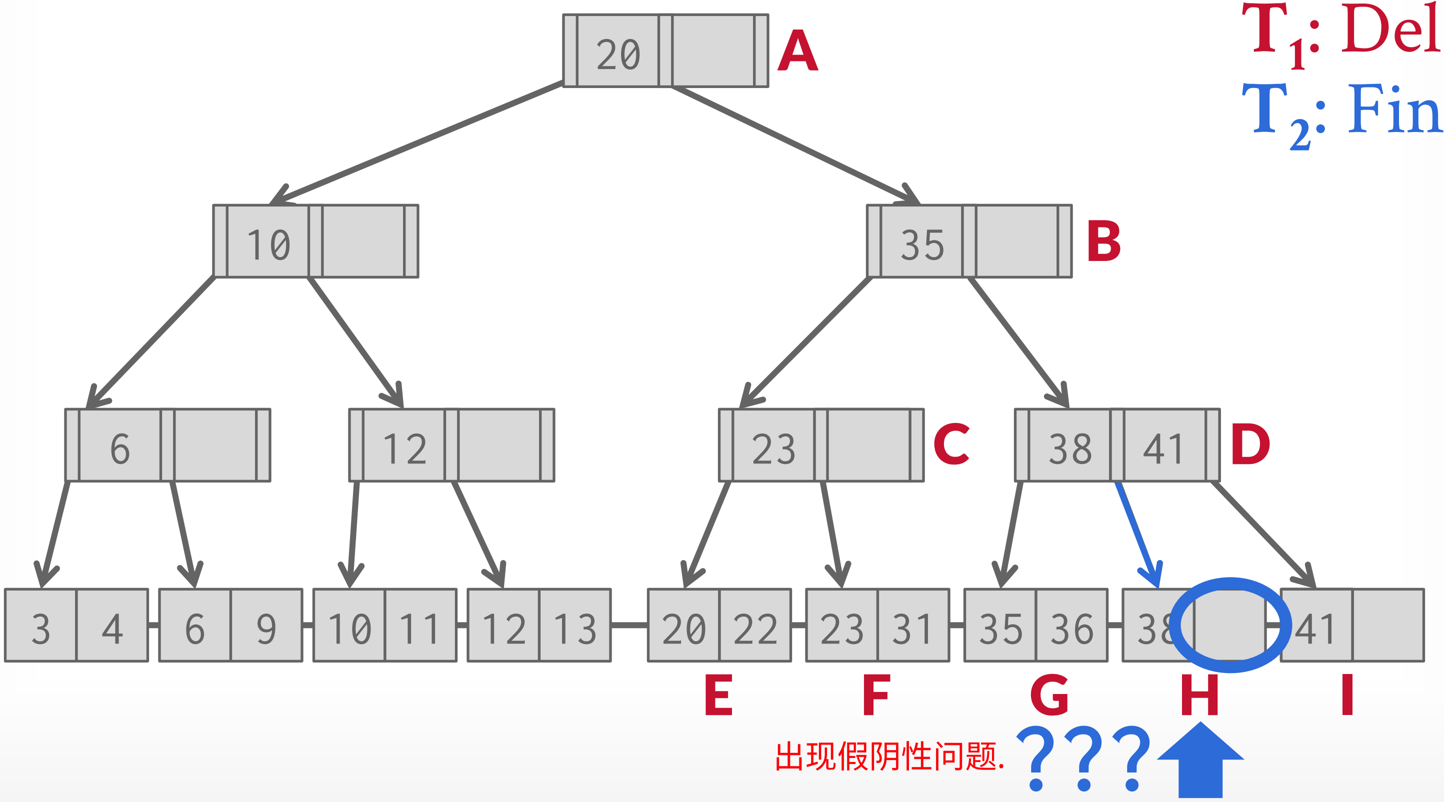
T_1: Delete 44

T_2: Find 41

B+TREE MULTI-THREADED EXAMPLE

T₁: Delete 44

T₂: Find 41



出现假阴性问题.

LATCH CRABBING/COUPLING

Protocol to allow multiple threads to access/modify B+Tree at the same time. 线程向下移动时如何获取闩锁.

- → Get latch for parent 1. 先获取父节点上的闩锁并保持;
- → Get latch for child 2. 获取导航的孩子节点的闩锁;
- → Release latch for parent if "safe" 3. 如果孩子节点是安全的, 则释放父节点上的闩锁.

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

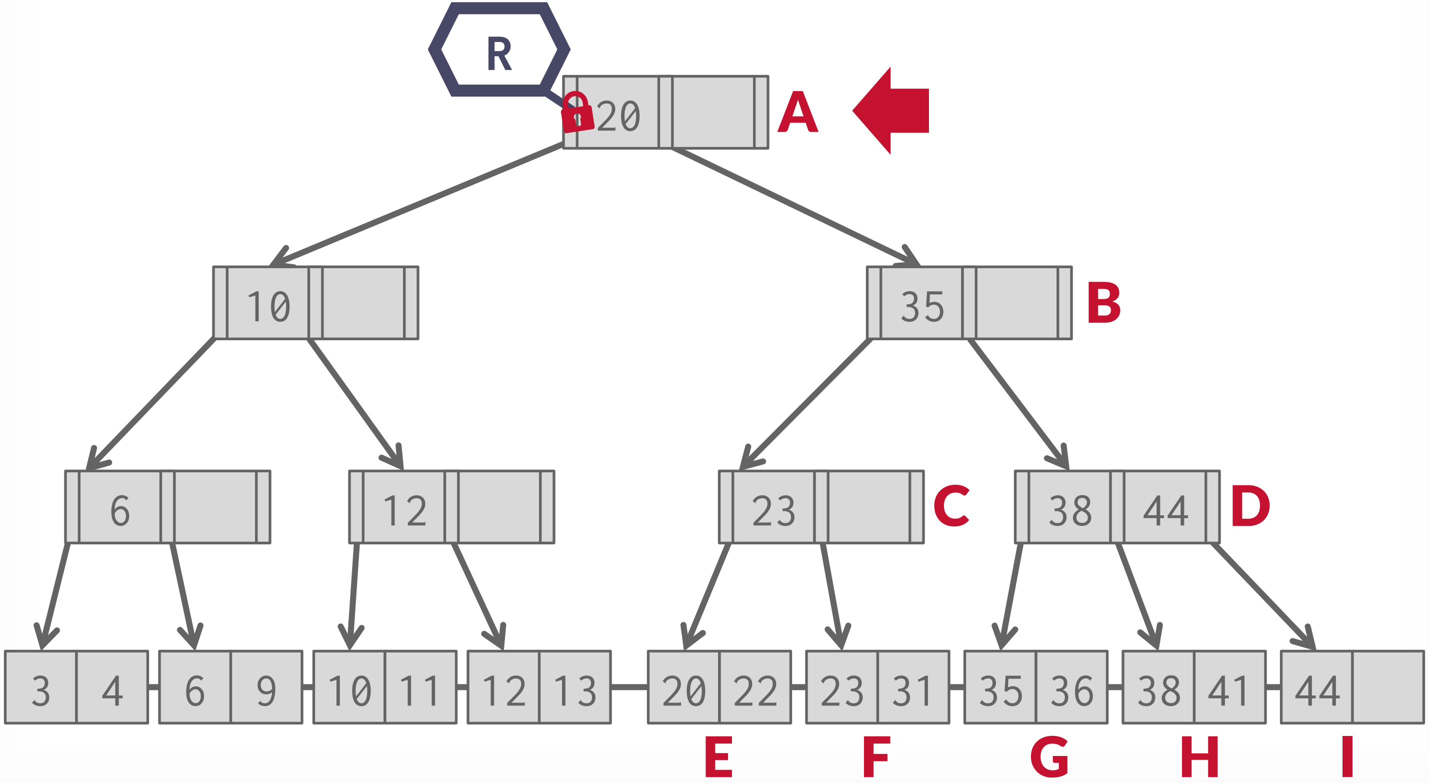
- → Not full (on insertion)
- → More than half-full (on deletion)

LATCH CRABBING/COUPLING

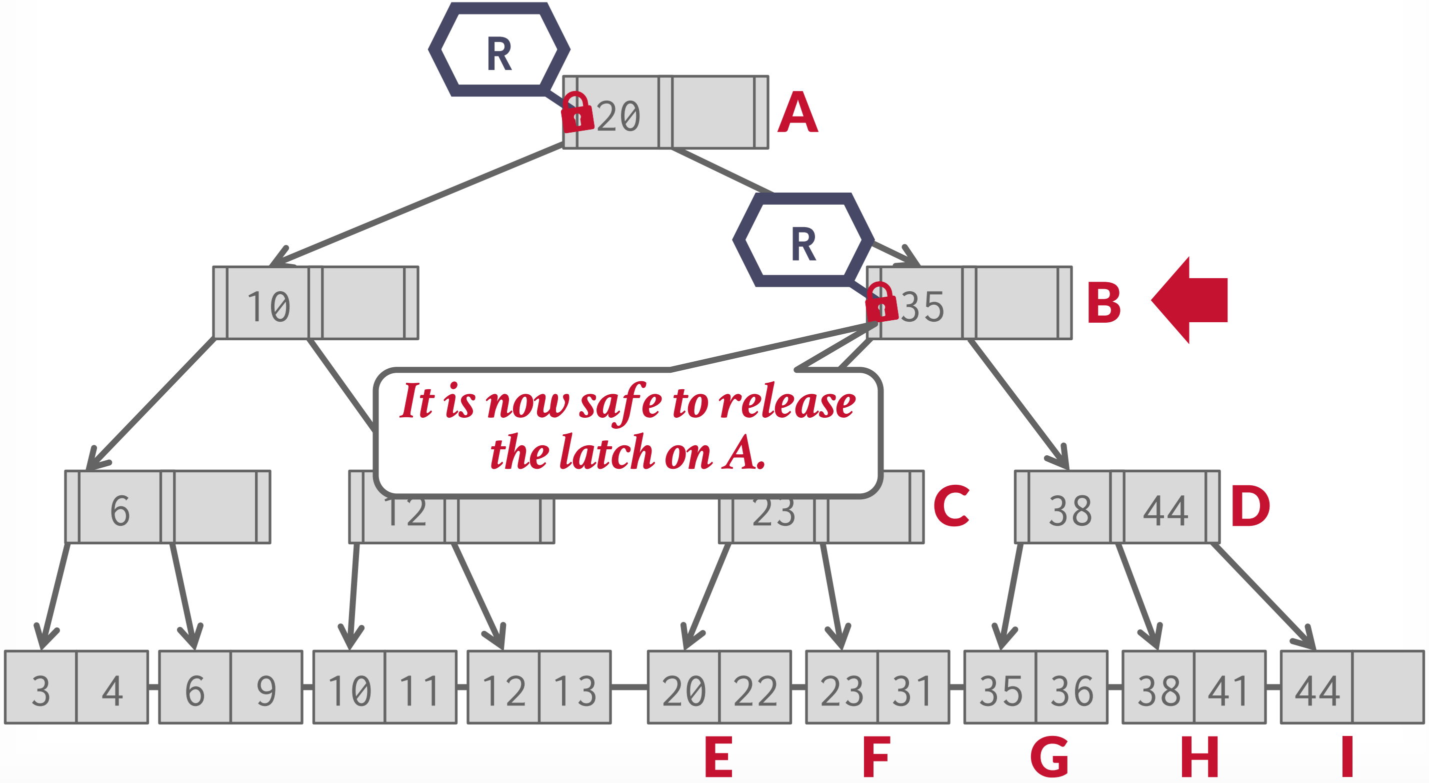
Find: Start at root and traverse down the tree:
→ Acquire **R** latch on child,
→ Then unlatch parent.
→ Repeat until we reach the leaf node.

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

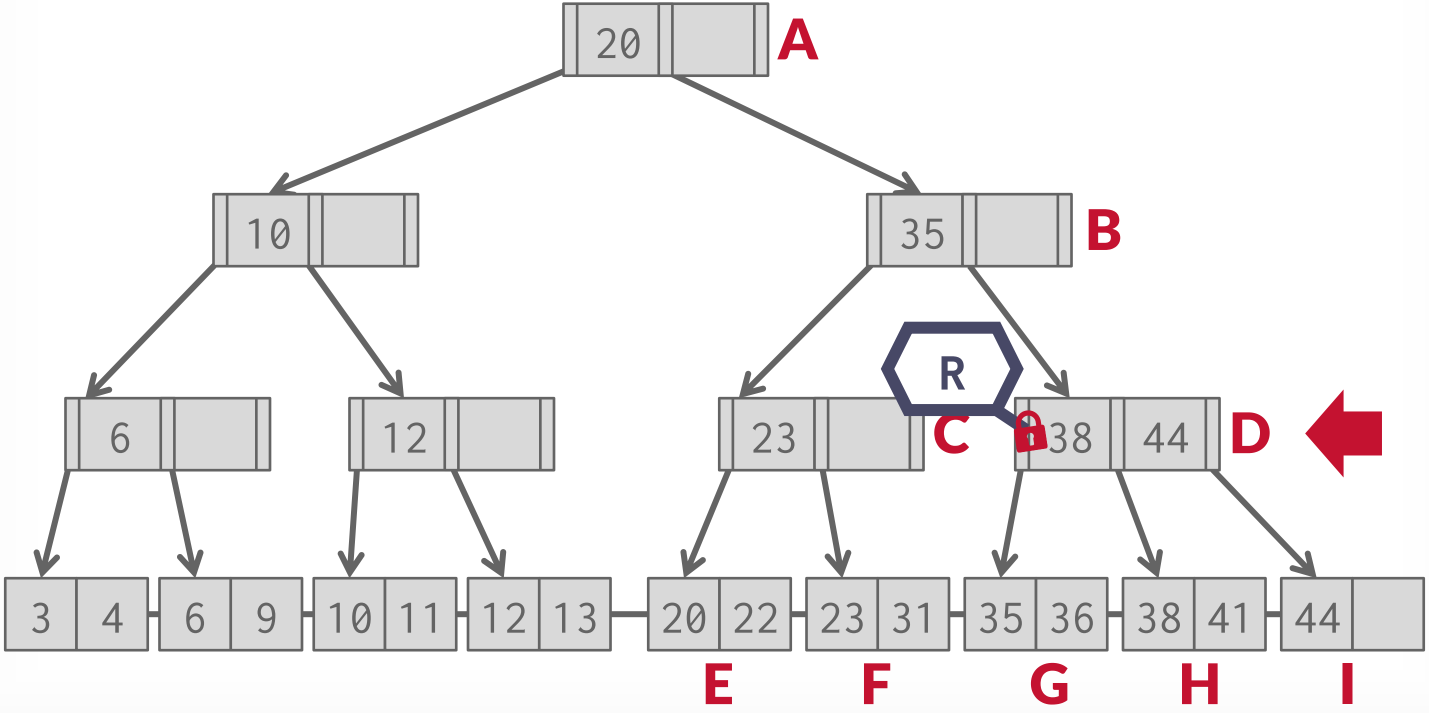


EXAMPLE #1 – FIND 38

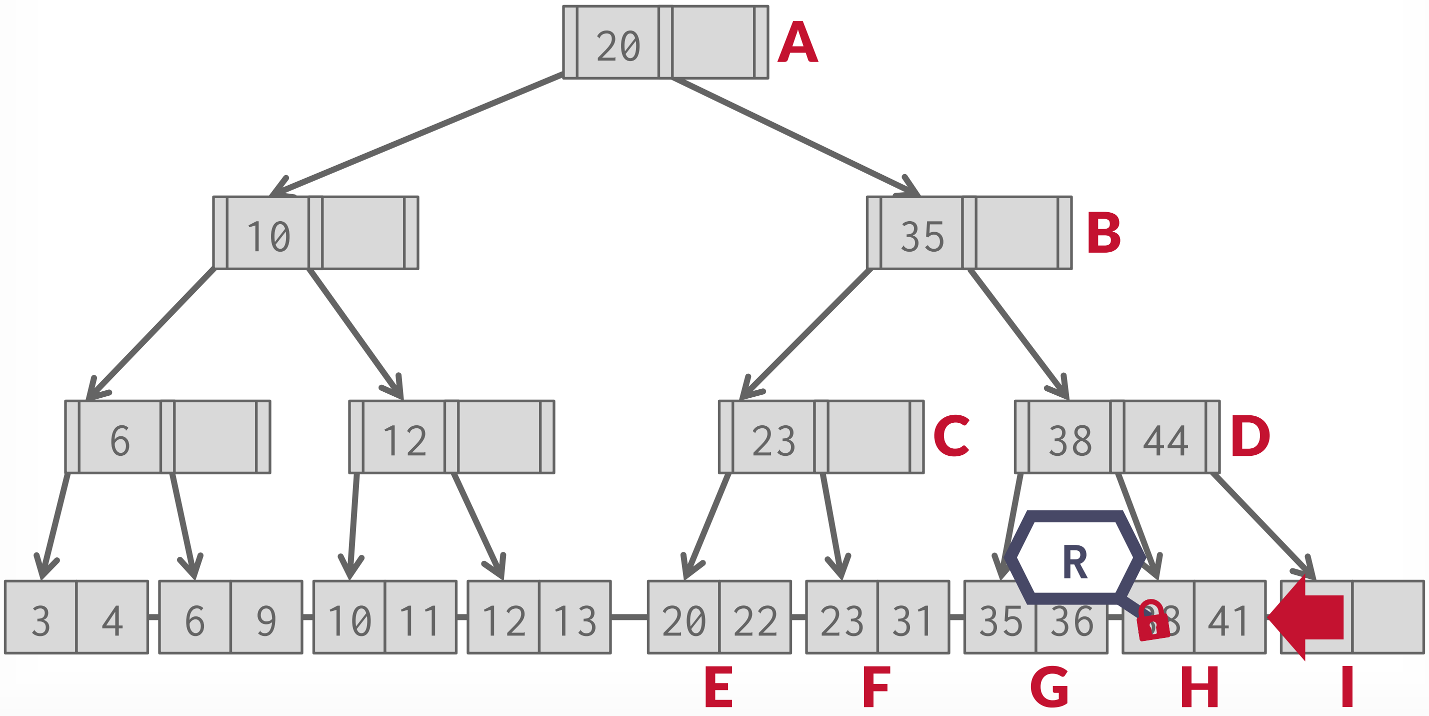


It is now safe to release the latch on A.

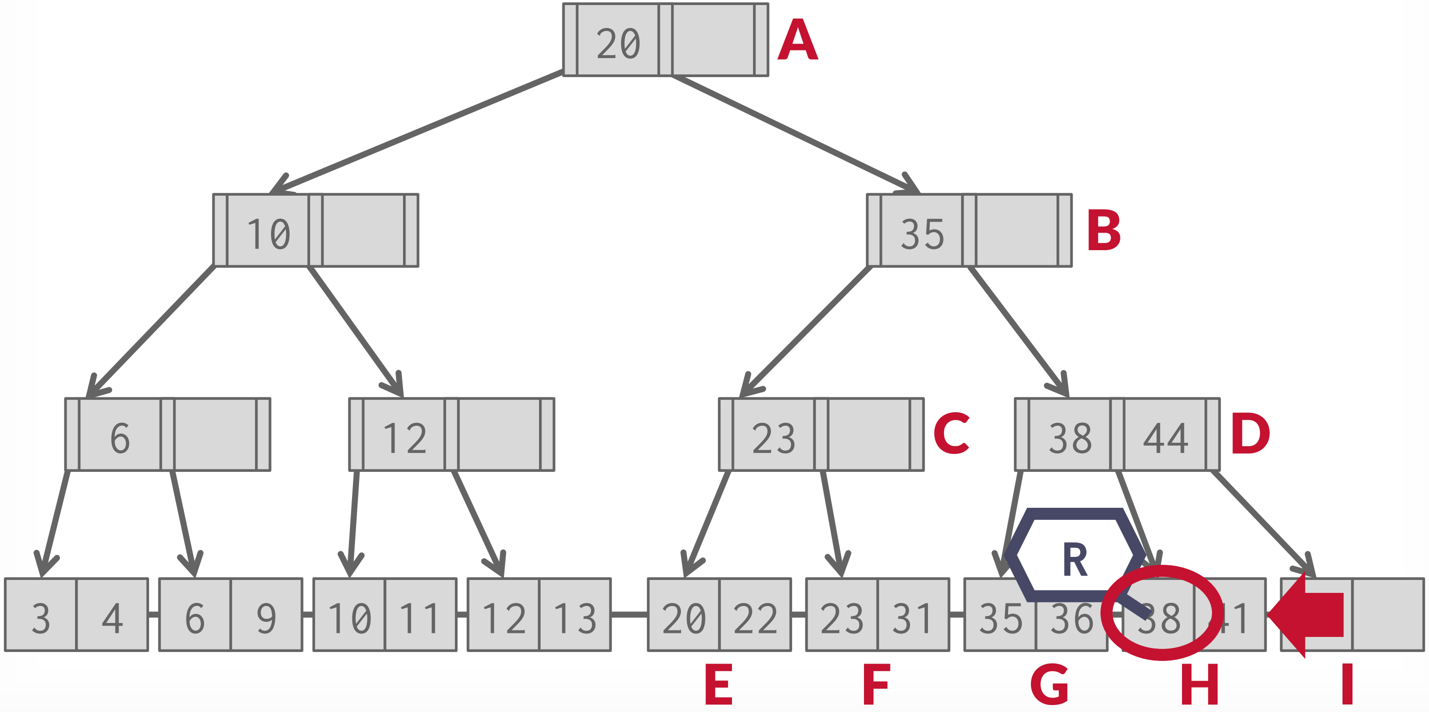
EXAMPLE #1 – FIND 38



EXAMPLE #1 – FIND 38

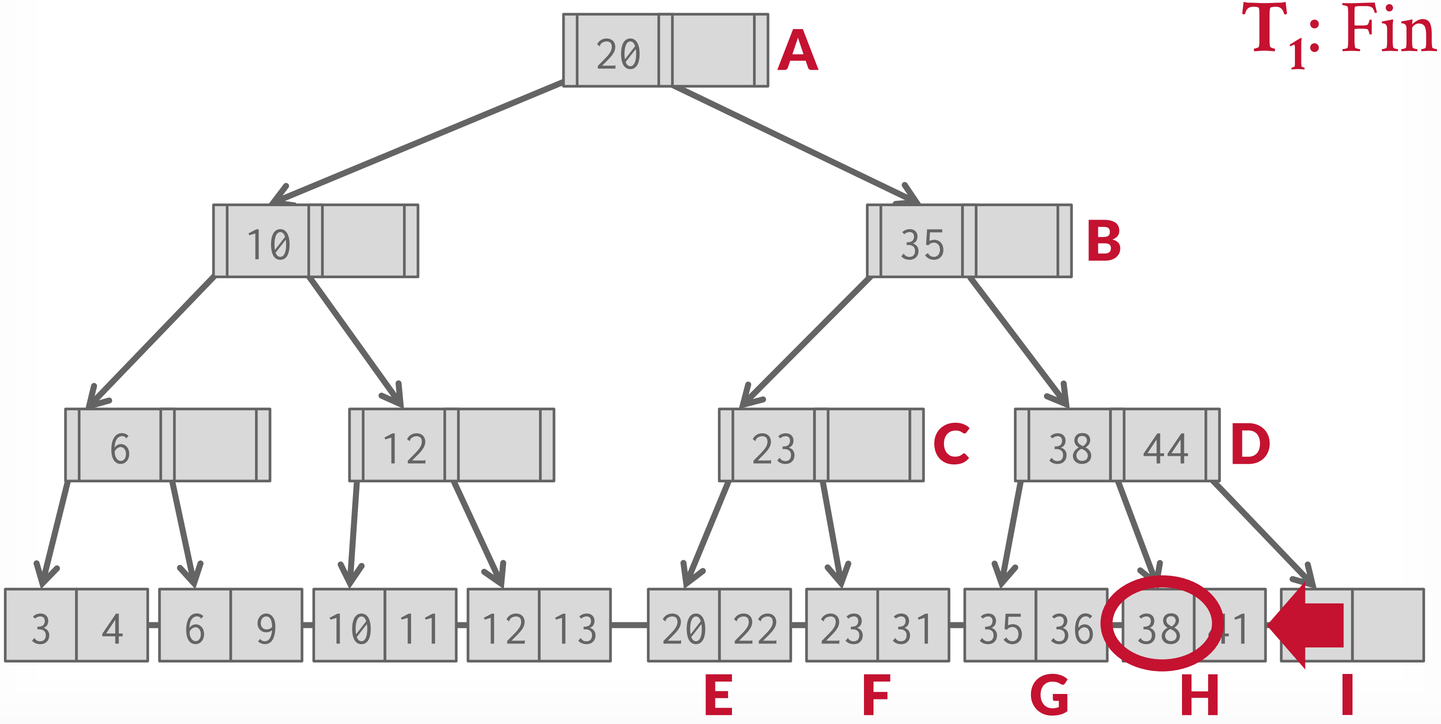


EXAMPLE #1 – FIND 38

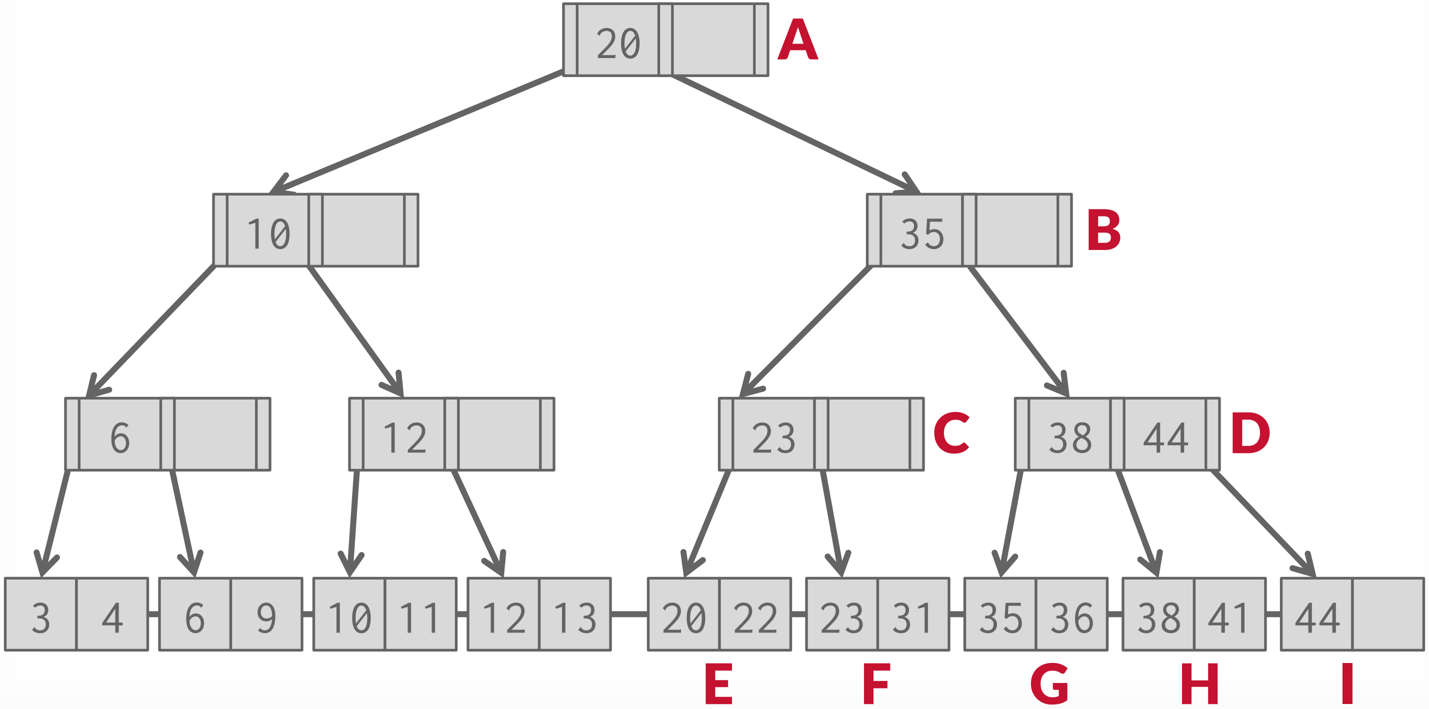


EXAMPLE #1 – FIND 38

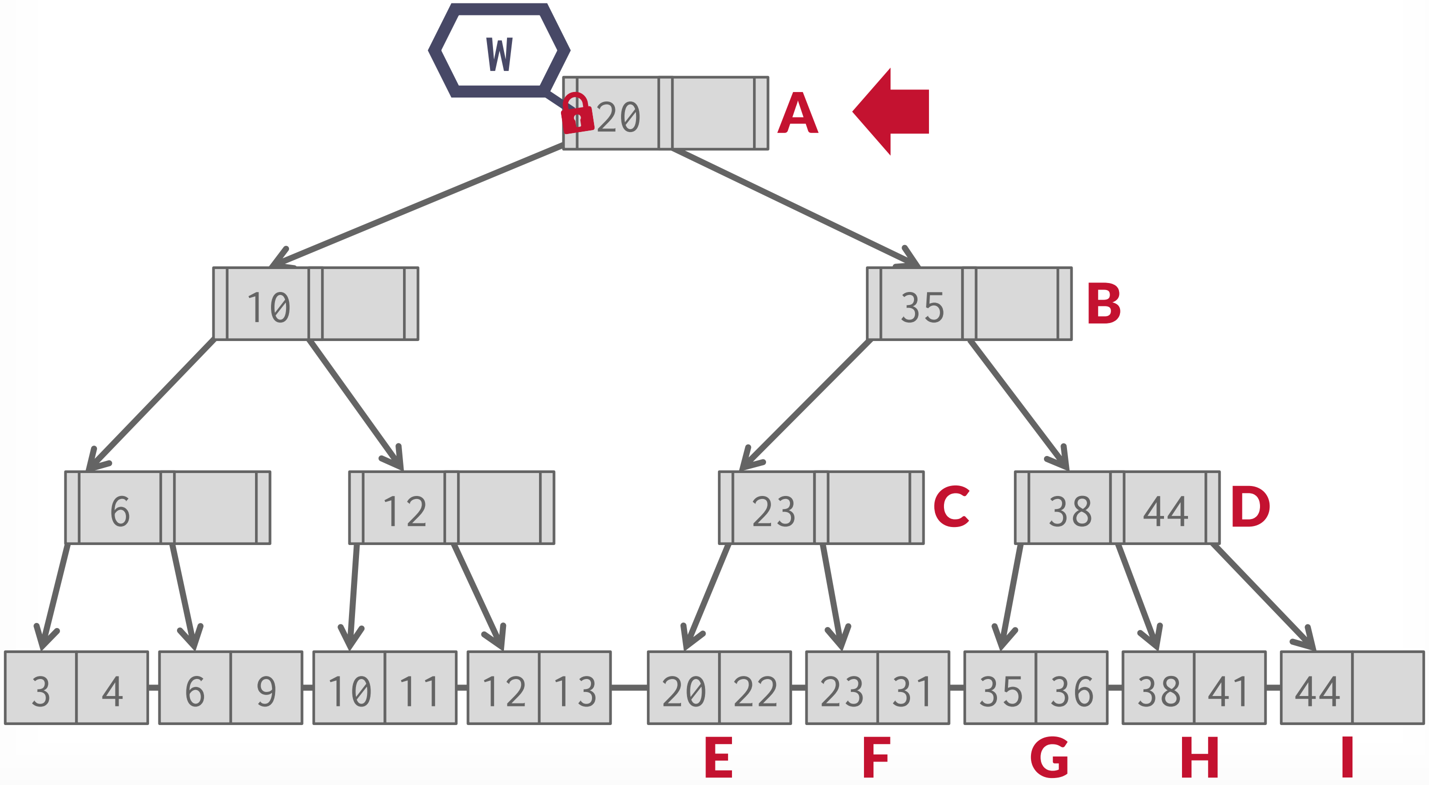
T_1: Find 38



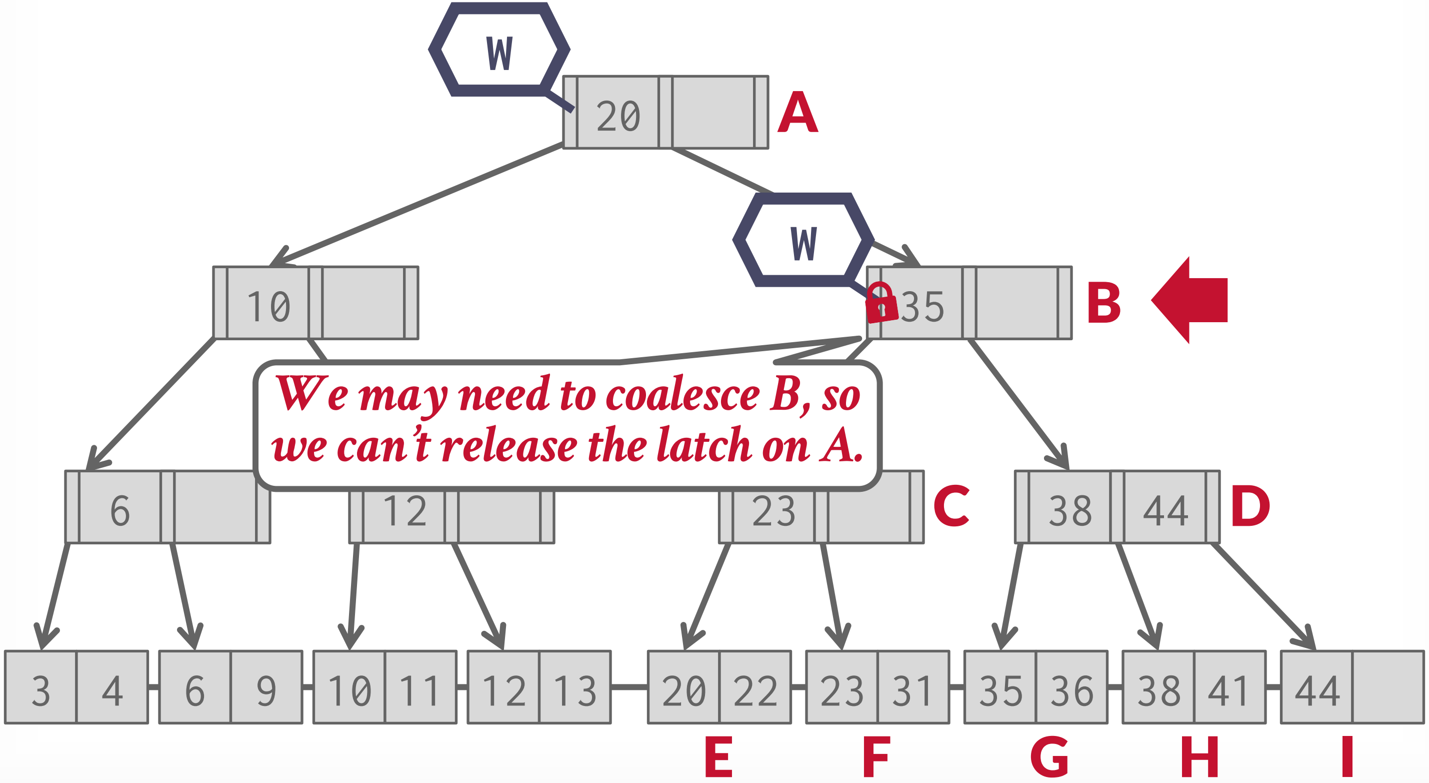
EXAMPLE #2 – DELETE 38



EXAMPLE #2 – DELETE 38

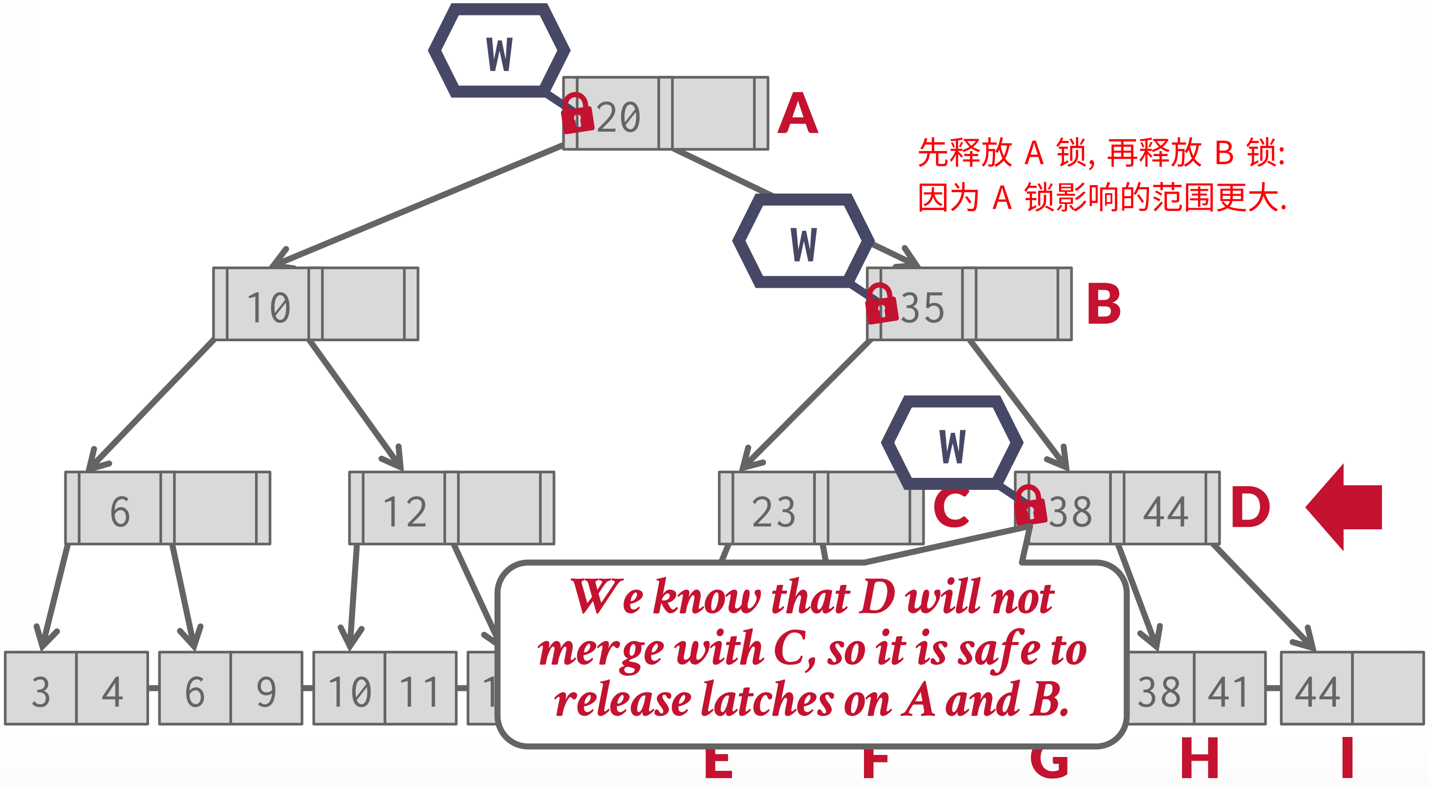


EXAMPLE #2 – DELETE 38



We may need to coalesce B, so we can't release the latch on A.

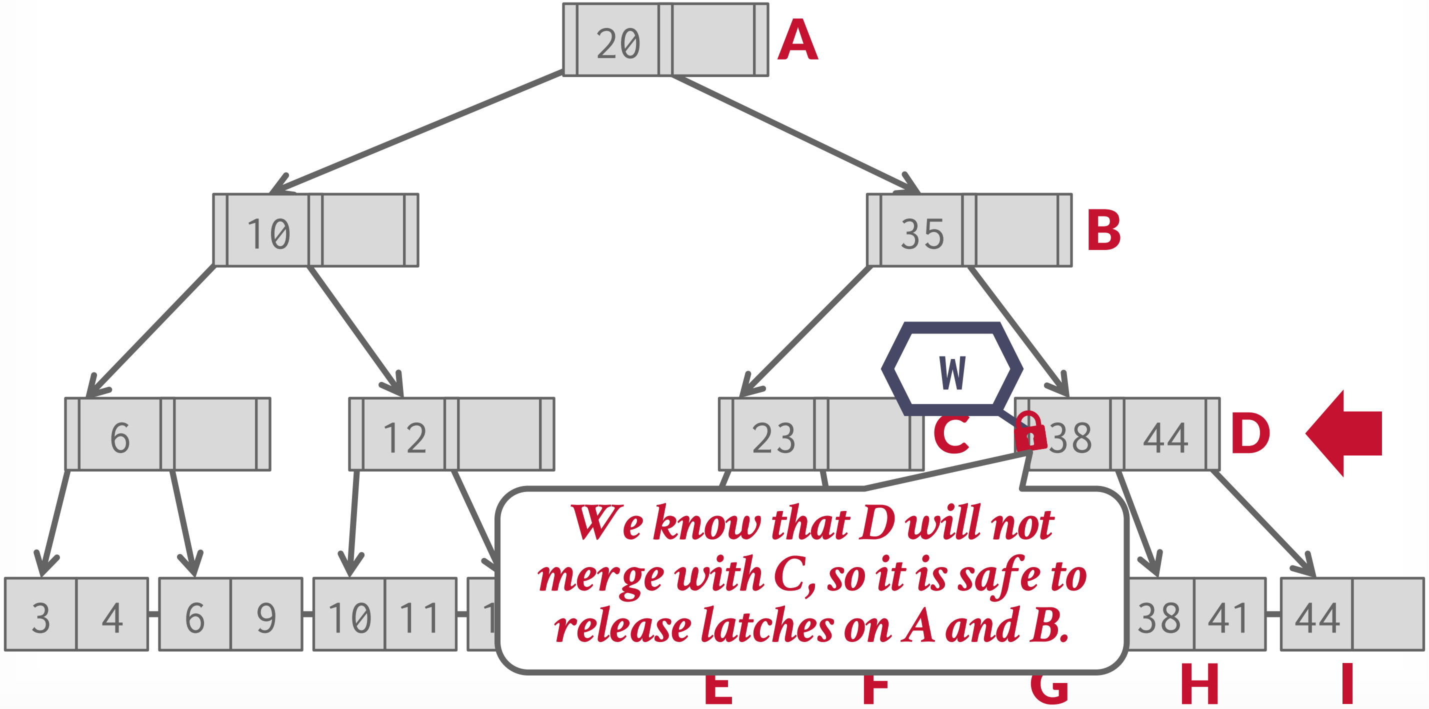
EXAMPLE #2 – DELETE 38



先释放 A 锁, 再释放 B 锁:
因为 A 锁影响的范围更大.

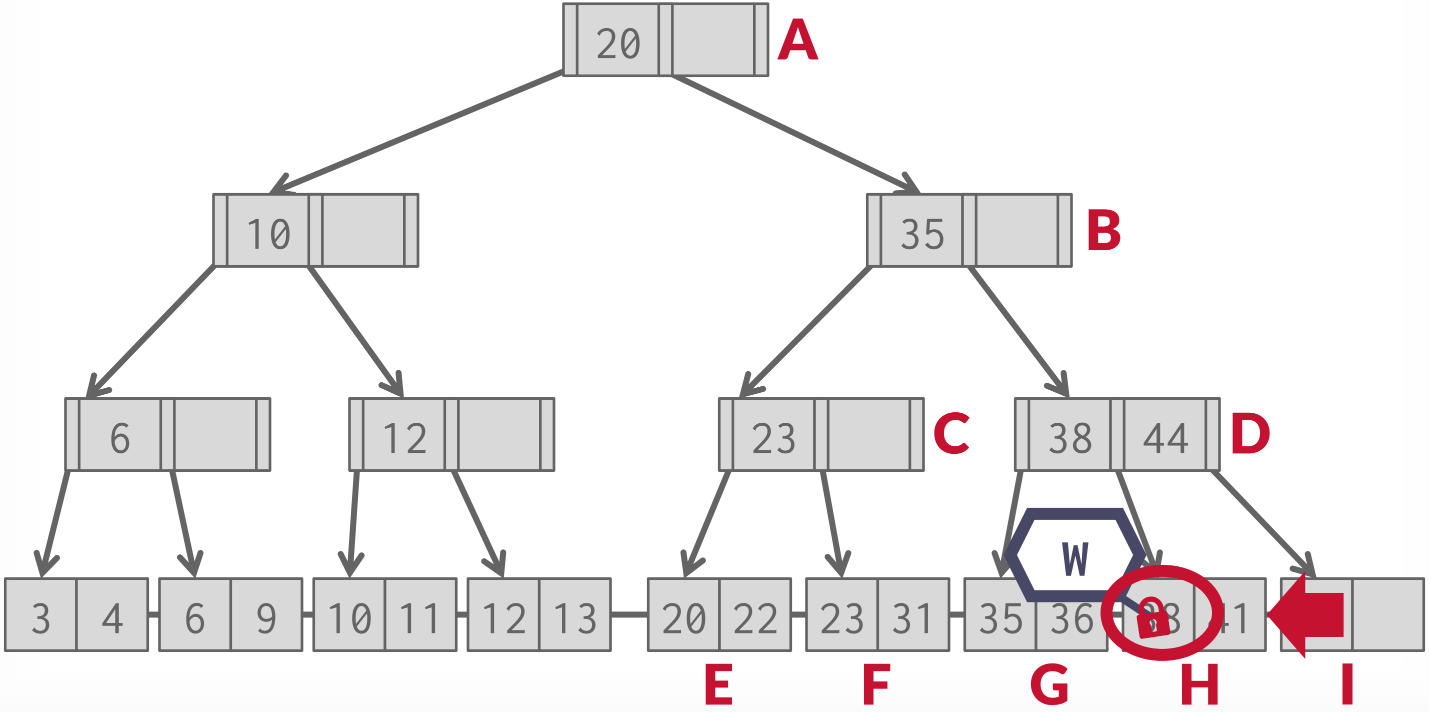
We know that D will not merge with C, so it is safe to release latches on A and B.

EXAMPLE #2 – DELETE 38

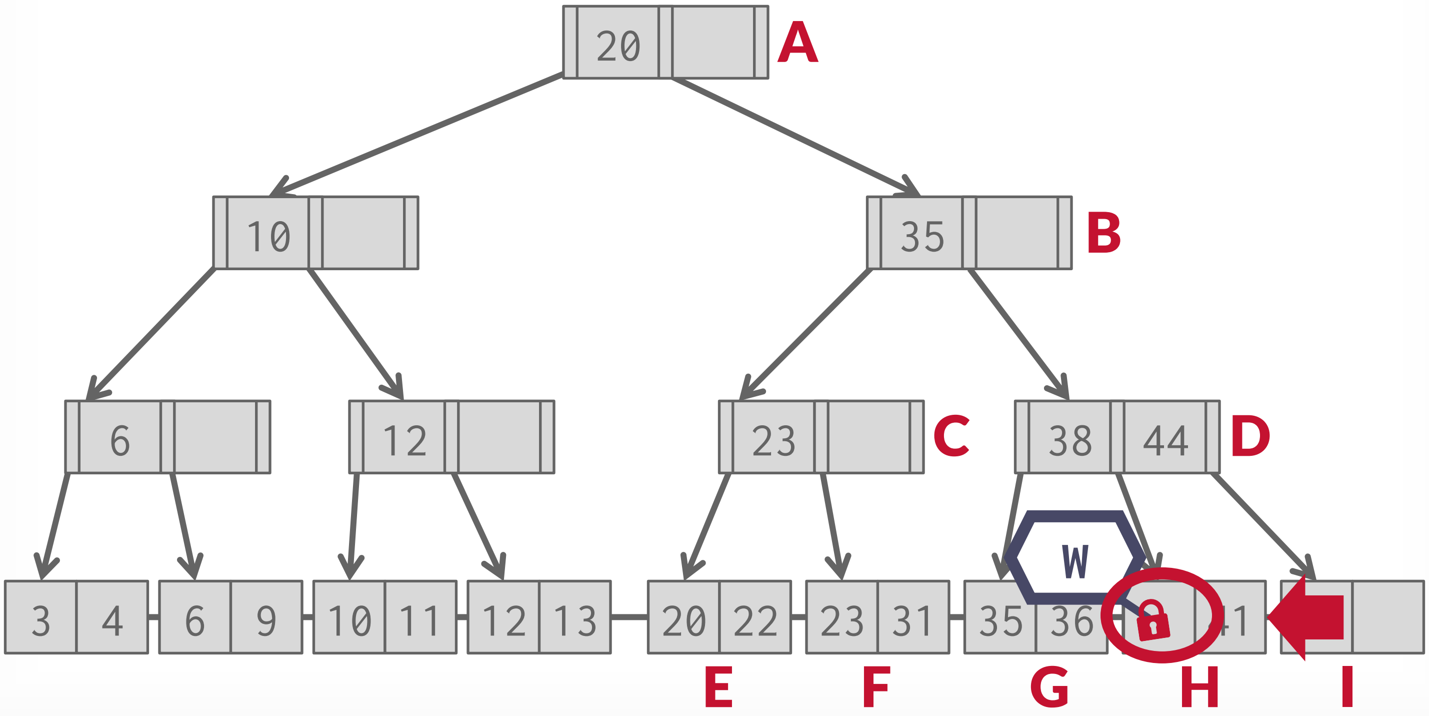


We know that D will not merge with C, so it is safe to release latches on A and B.

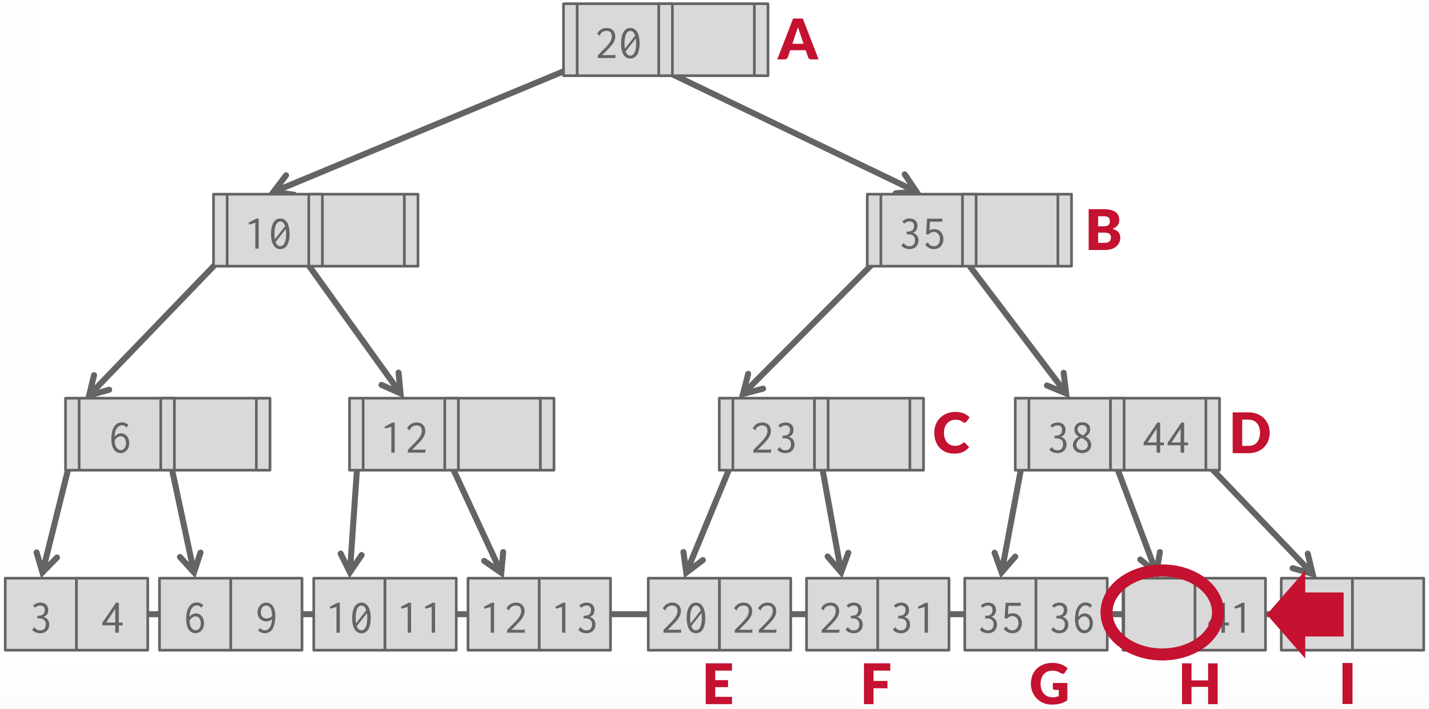
EXAMPLE #2 – DELETE 38



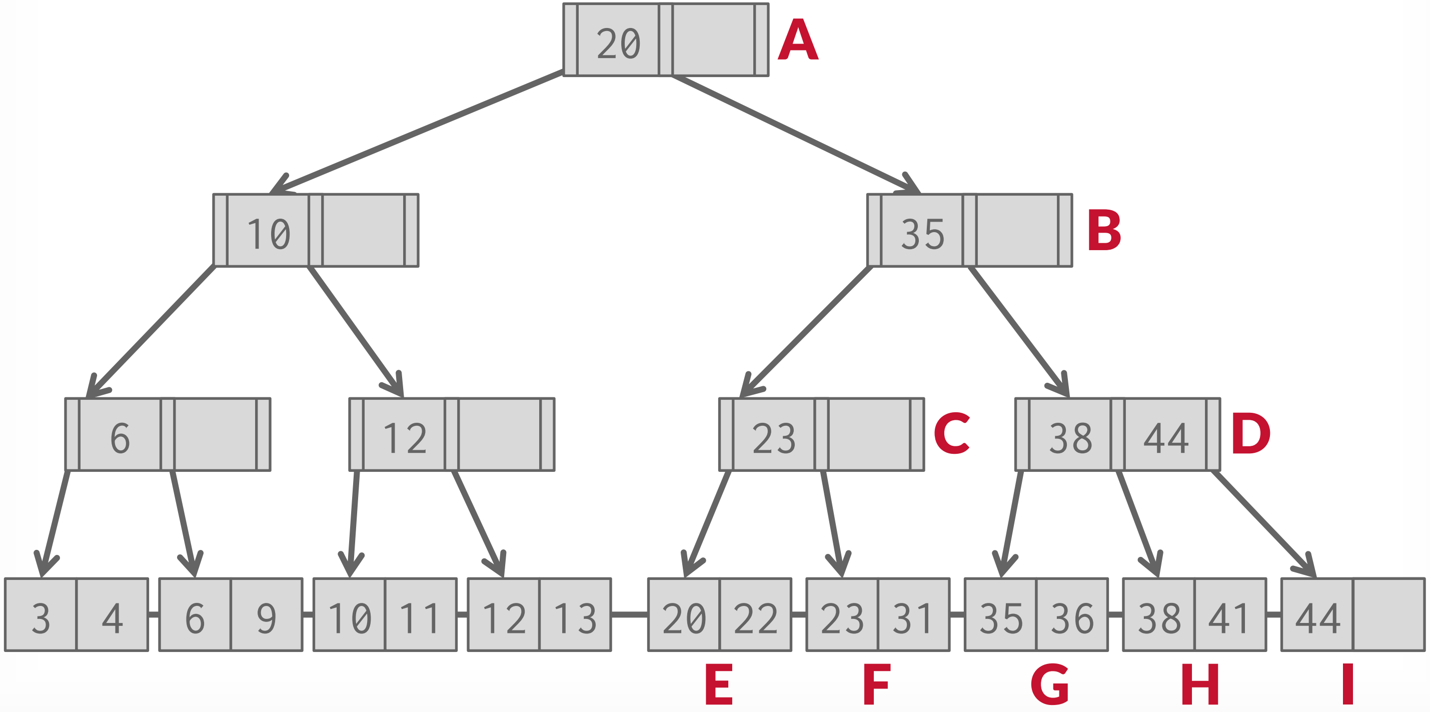
EXAMPLE #2 – DELETE 38



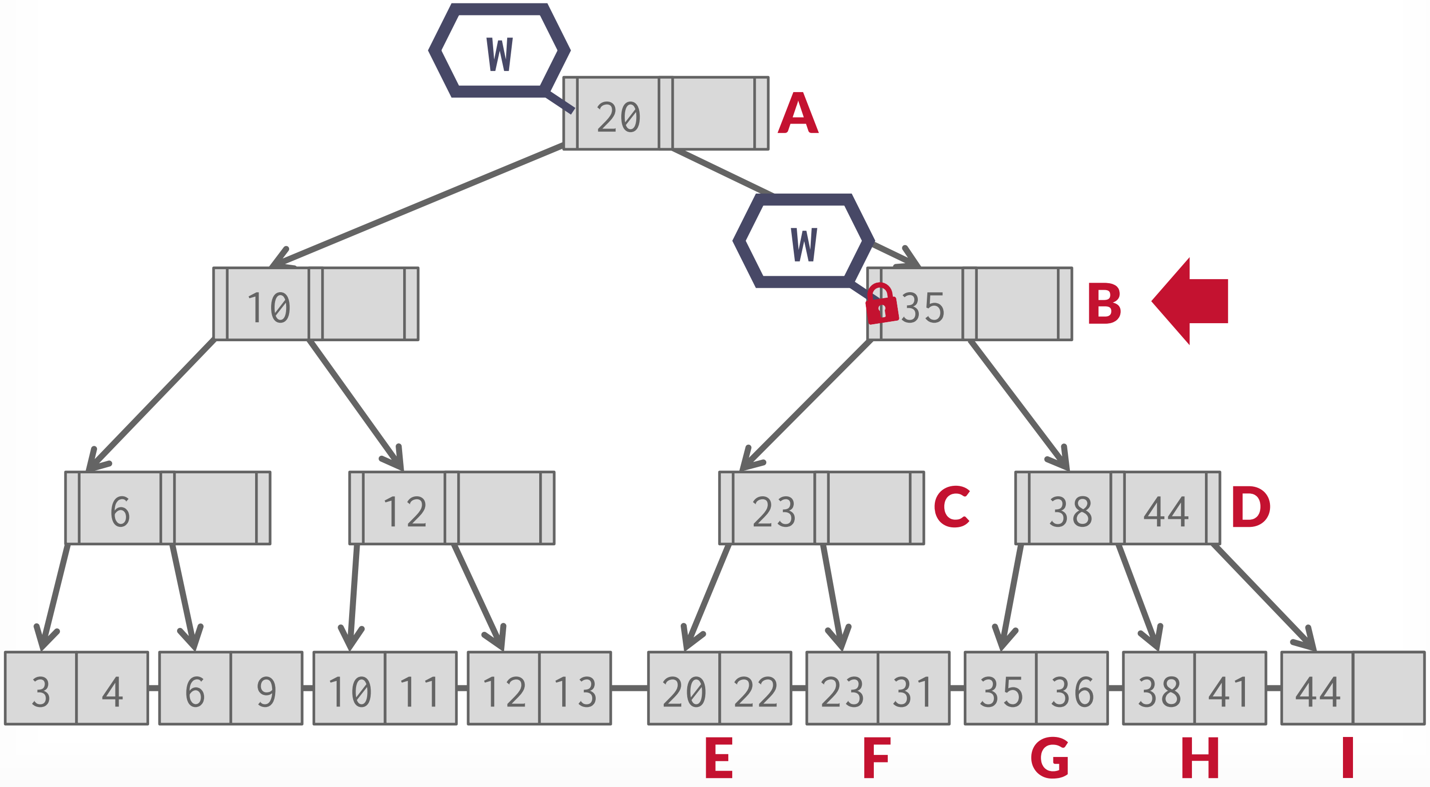
EXAMPLE #2 – DELETE 38



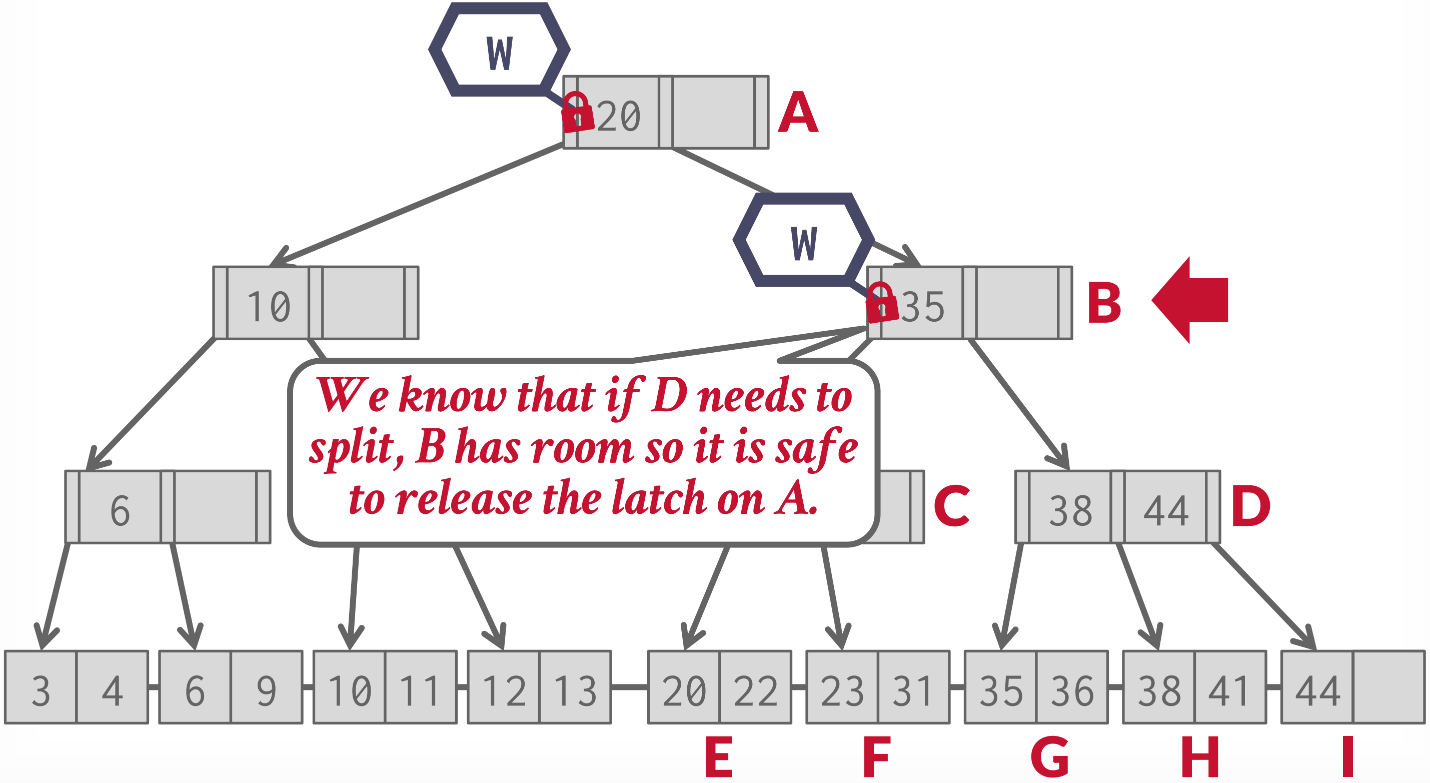
EXAMPLE #3 – INSERT 45



EXAMPLE #3 – INSERT 45

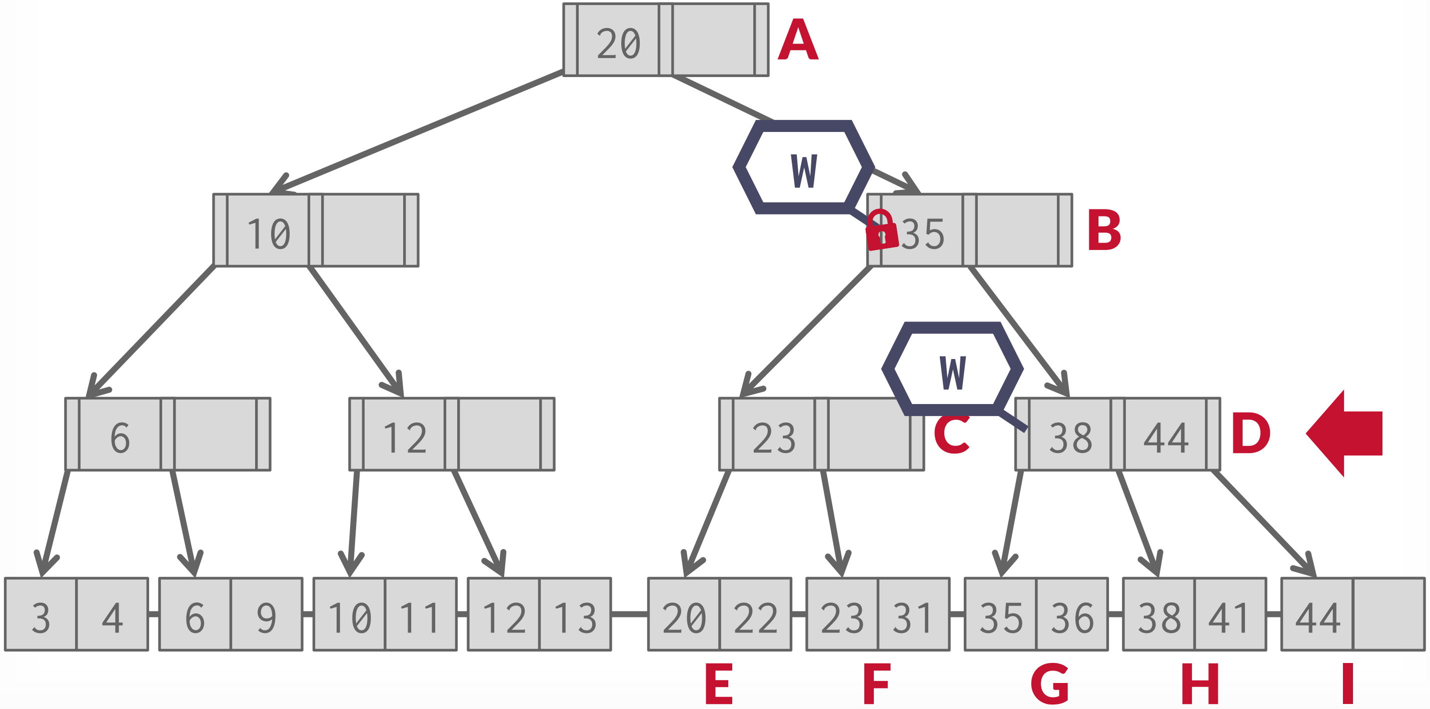


EXAMPLE #3 – INSERT 45

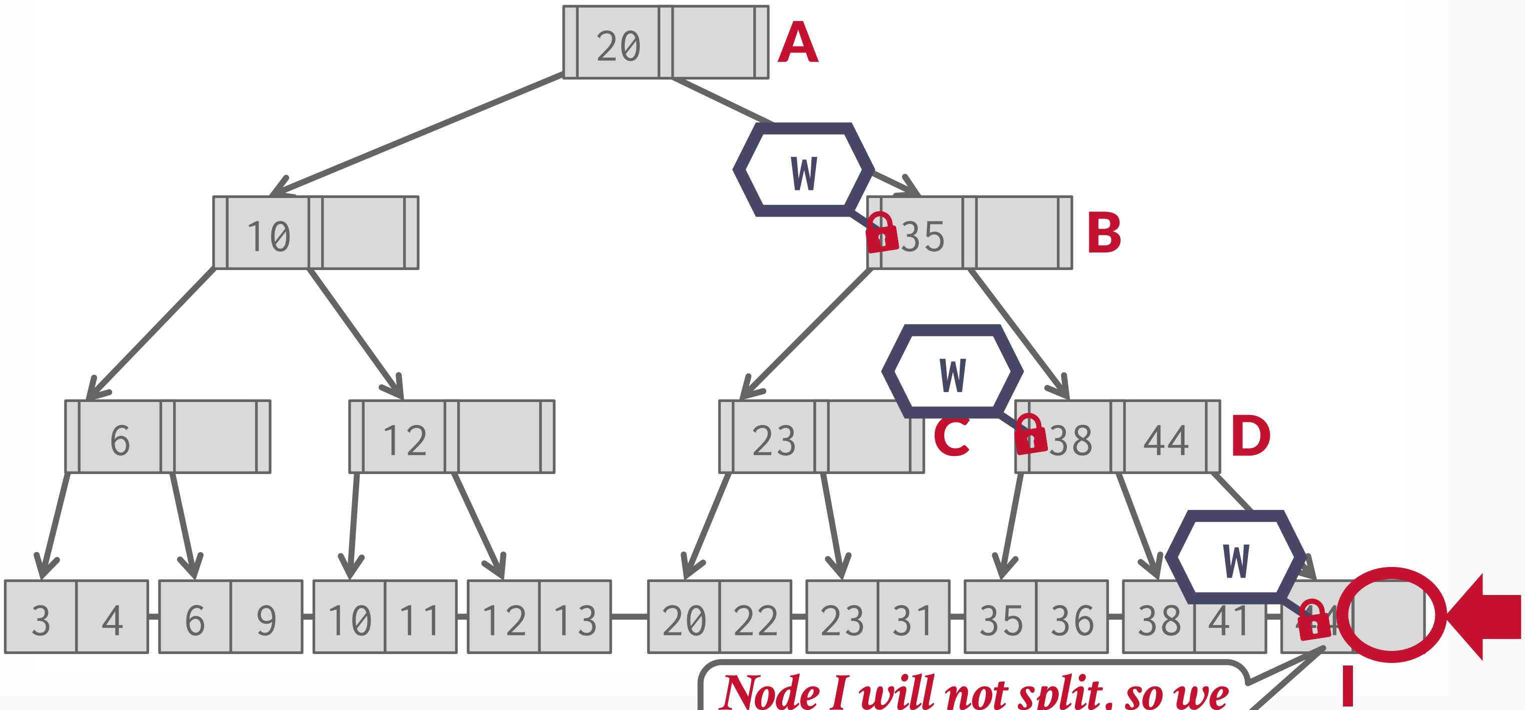


We know that if D needs to split, B has room so it is safe to release the latch on A.

EXAMPLE #3 – INSERT 45

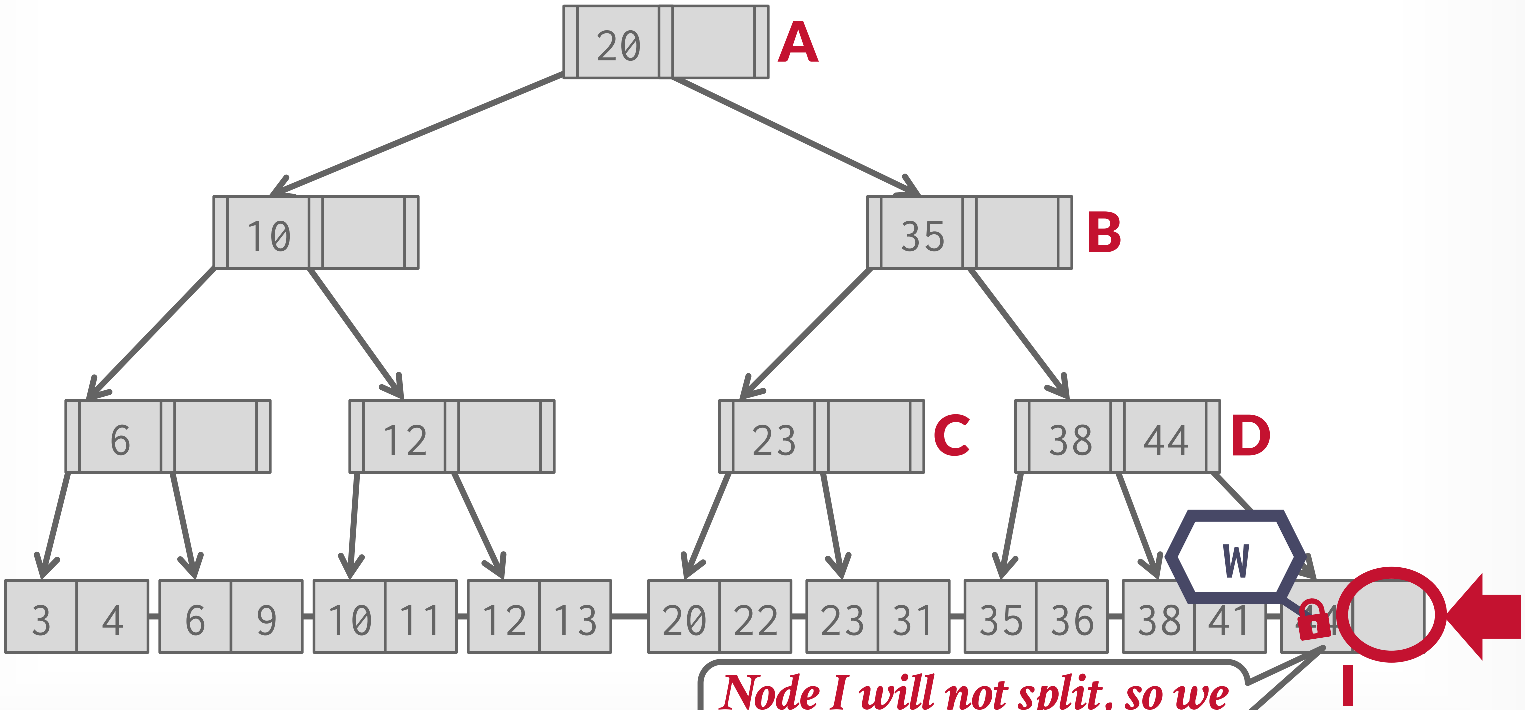


EXAMPLE #3 – INSERT 45



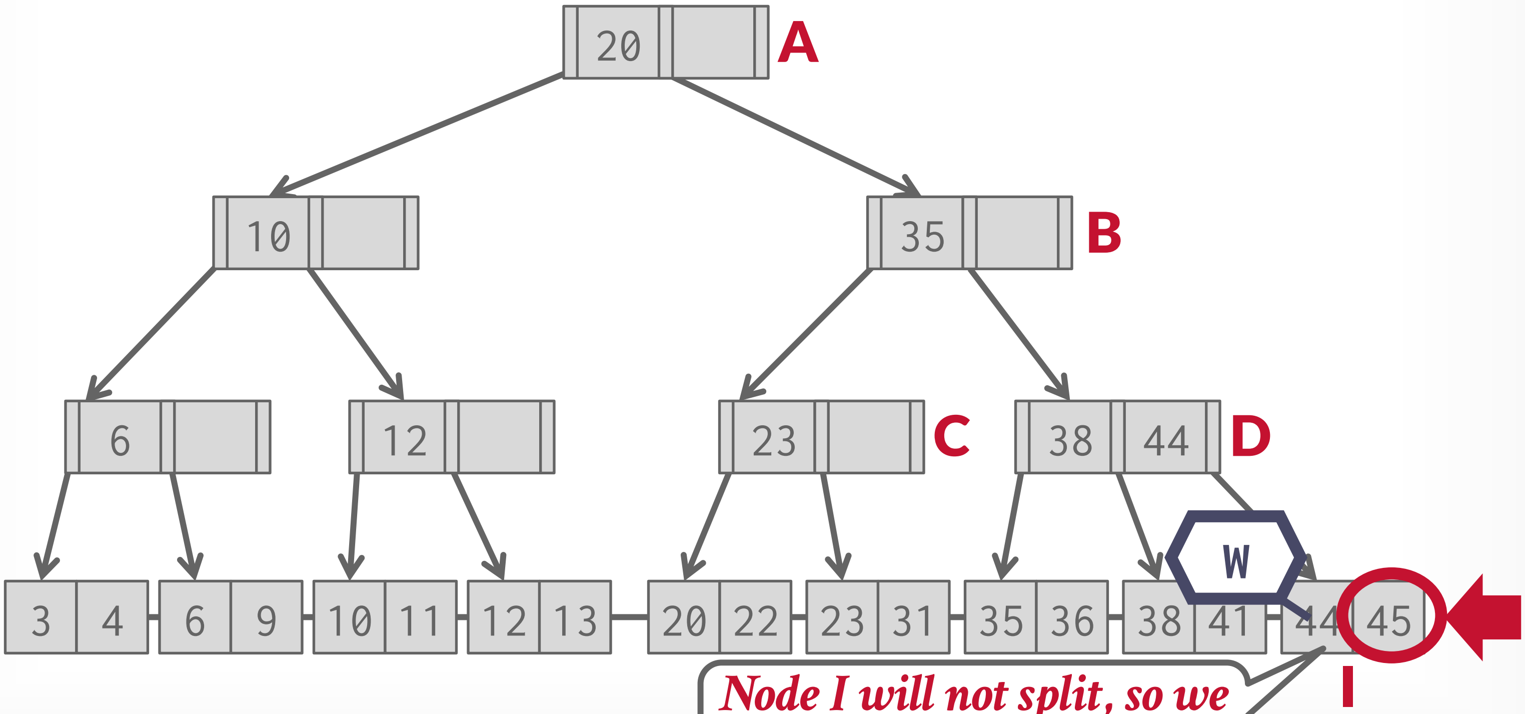
Node I will not split, so we can release B+D.

EXAMPLE #3 – INSERT 45



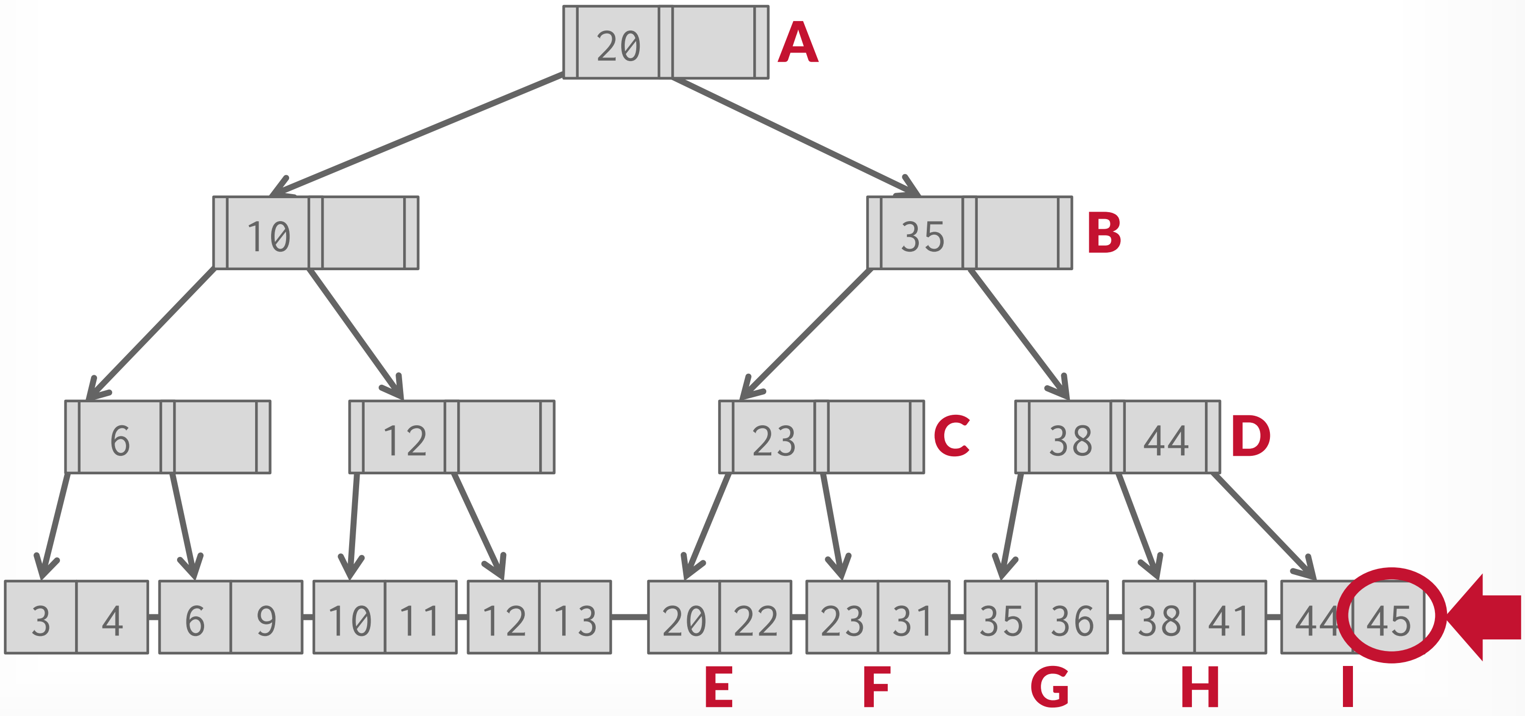
Node I will not split, so we can release B+D.

EXAMPLE #3 – INSERT 45

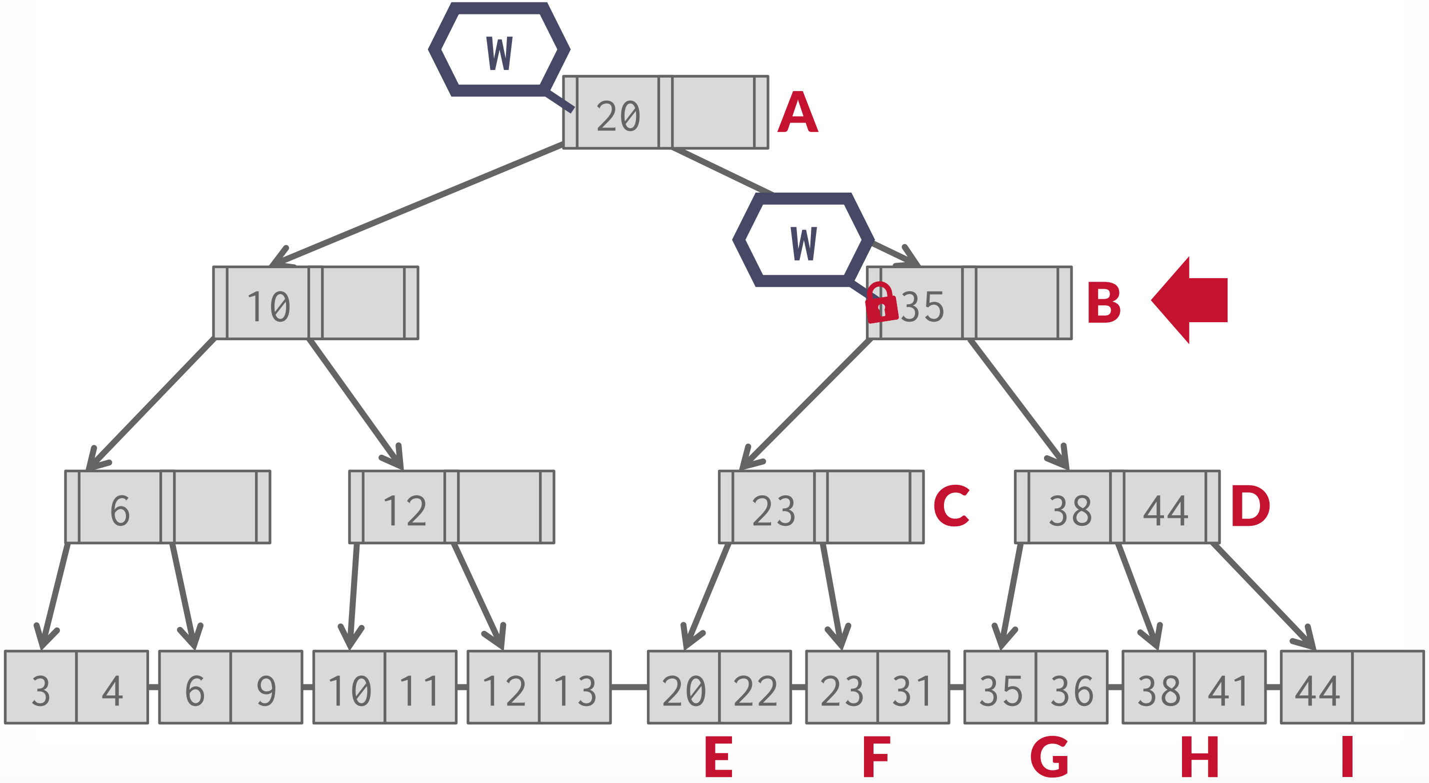


Node I will not split, so we can release B+D.

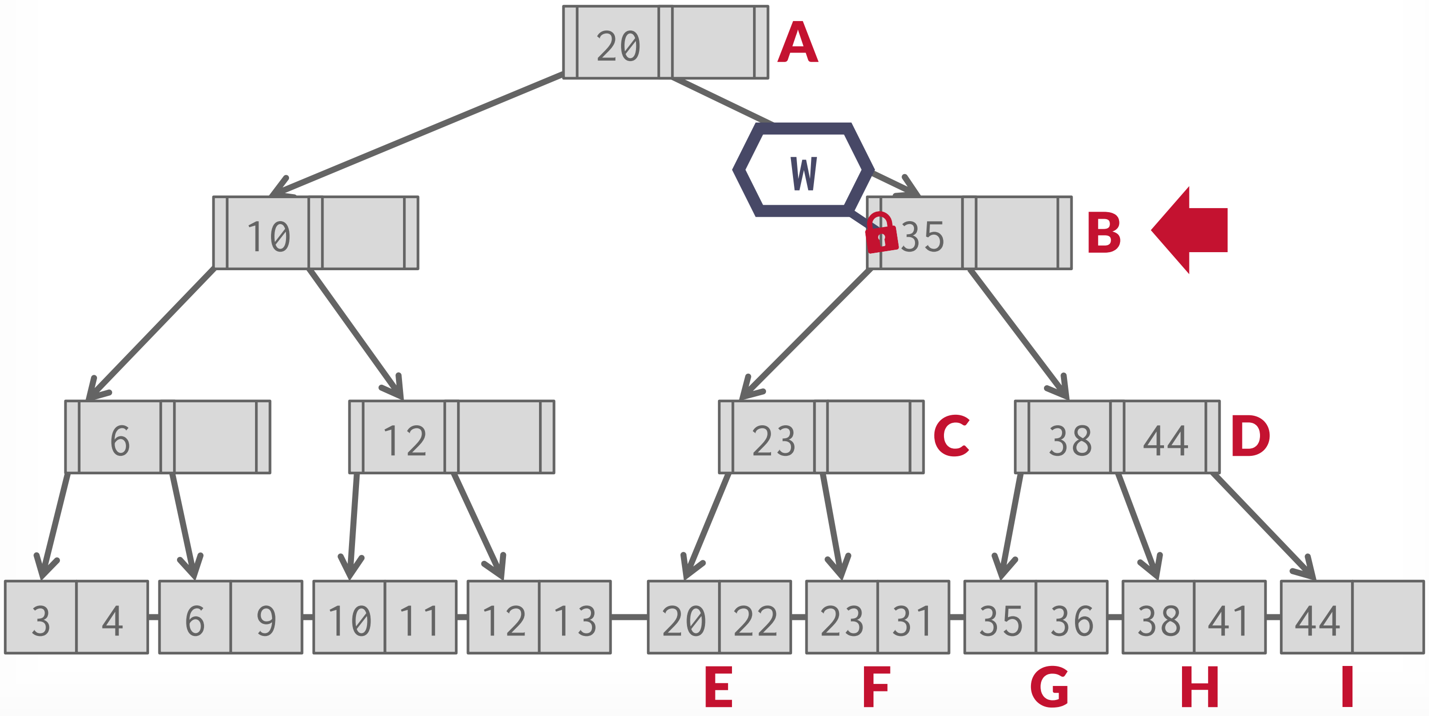
EXAMPLE #3 – INSERT 45



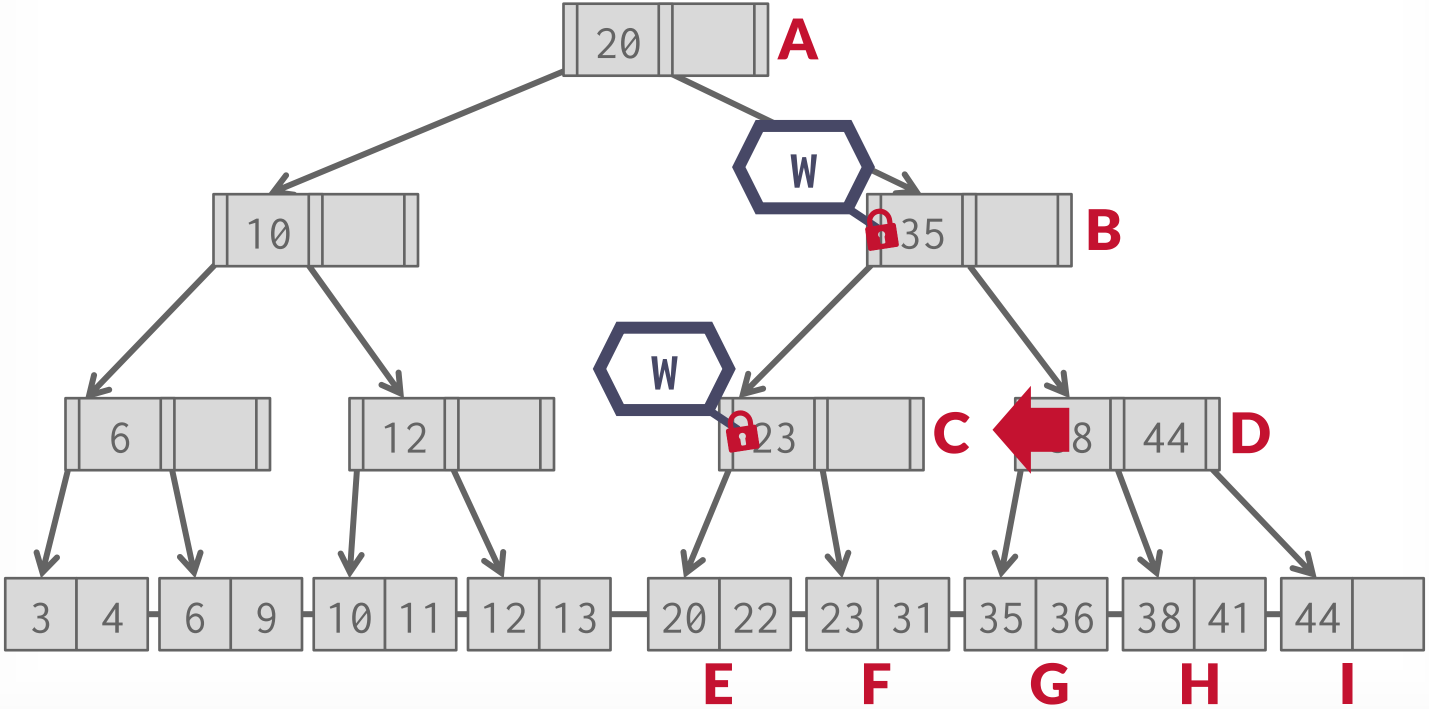
EXAMPLE #4 – INSERT 25



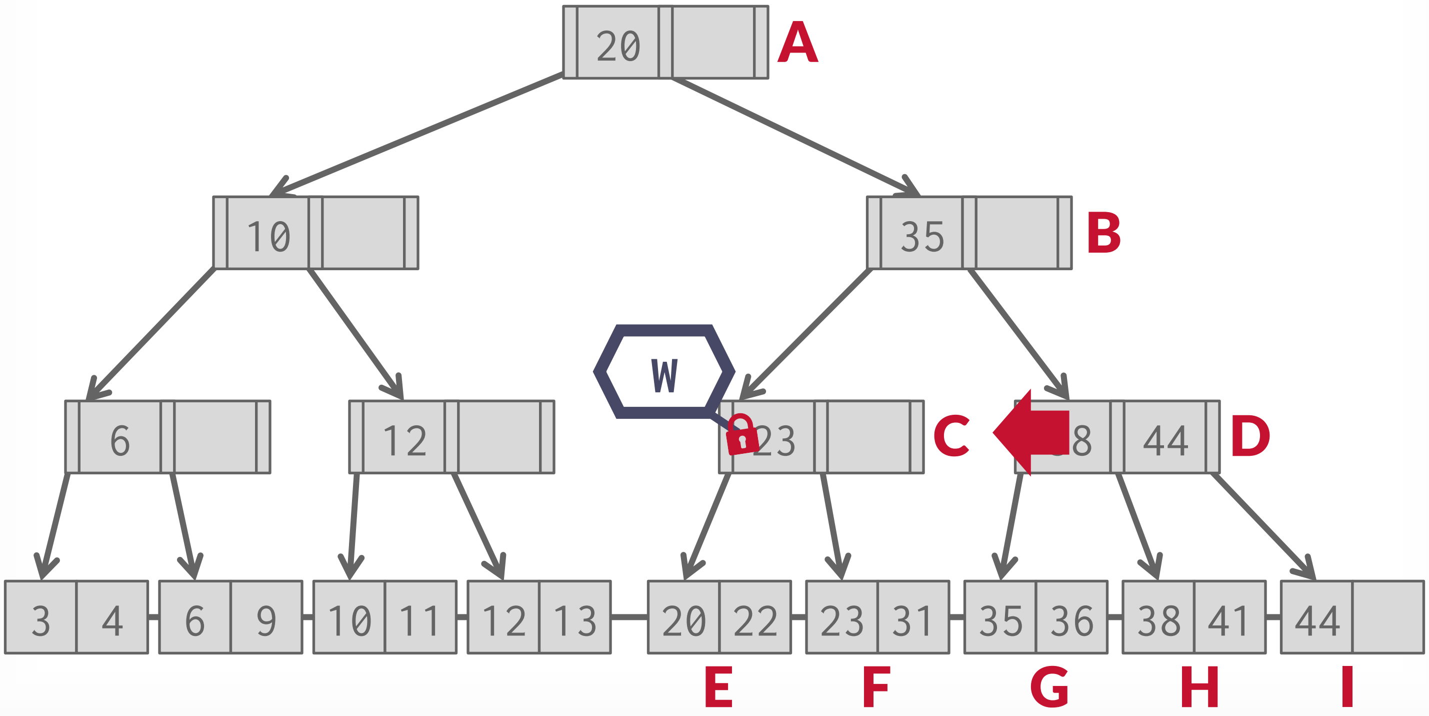
EXAMPLE #4 – INSERT 25



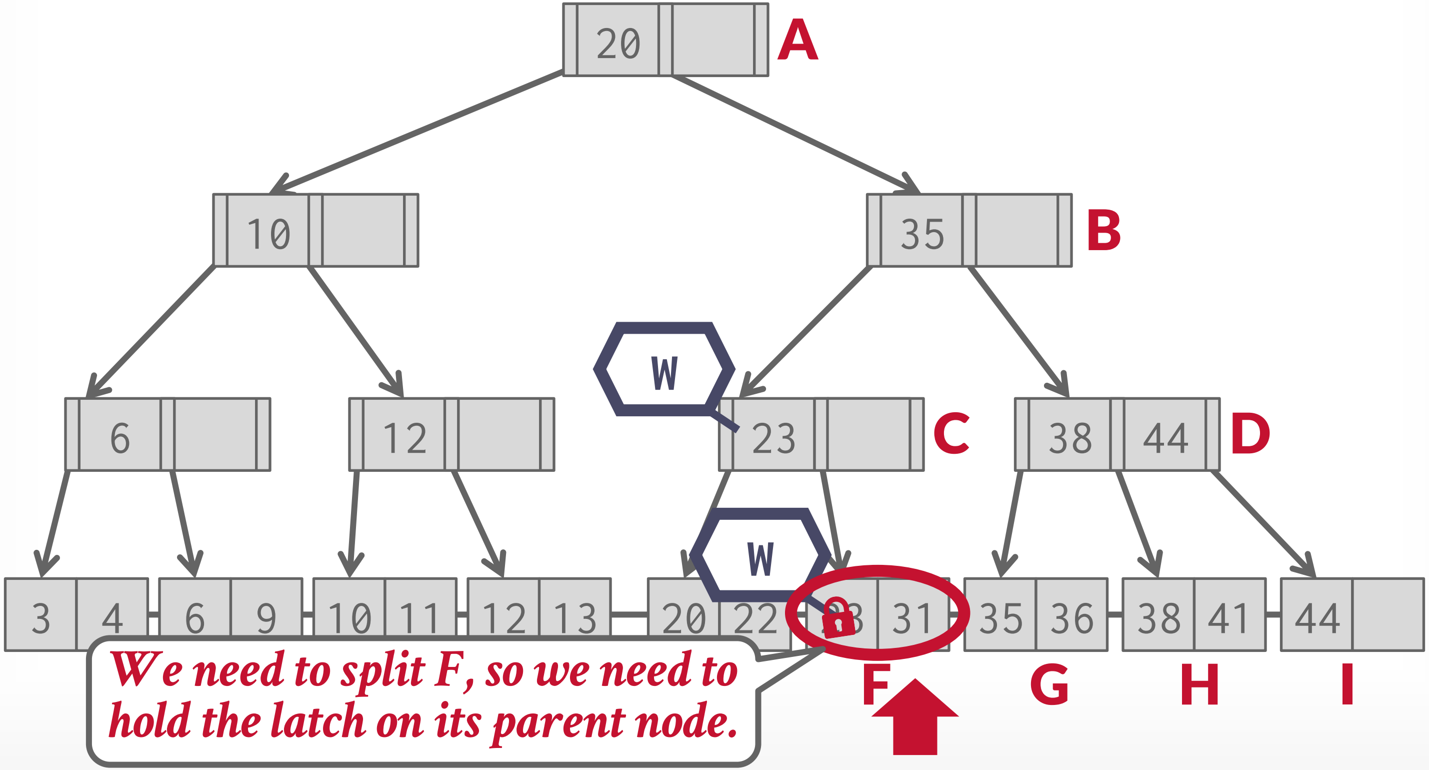
EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25

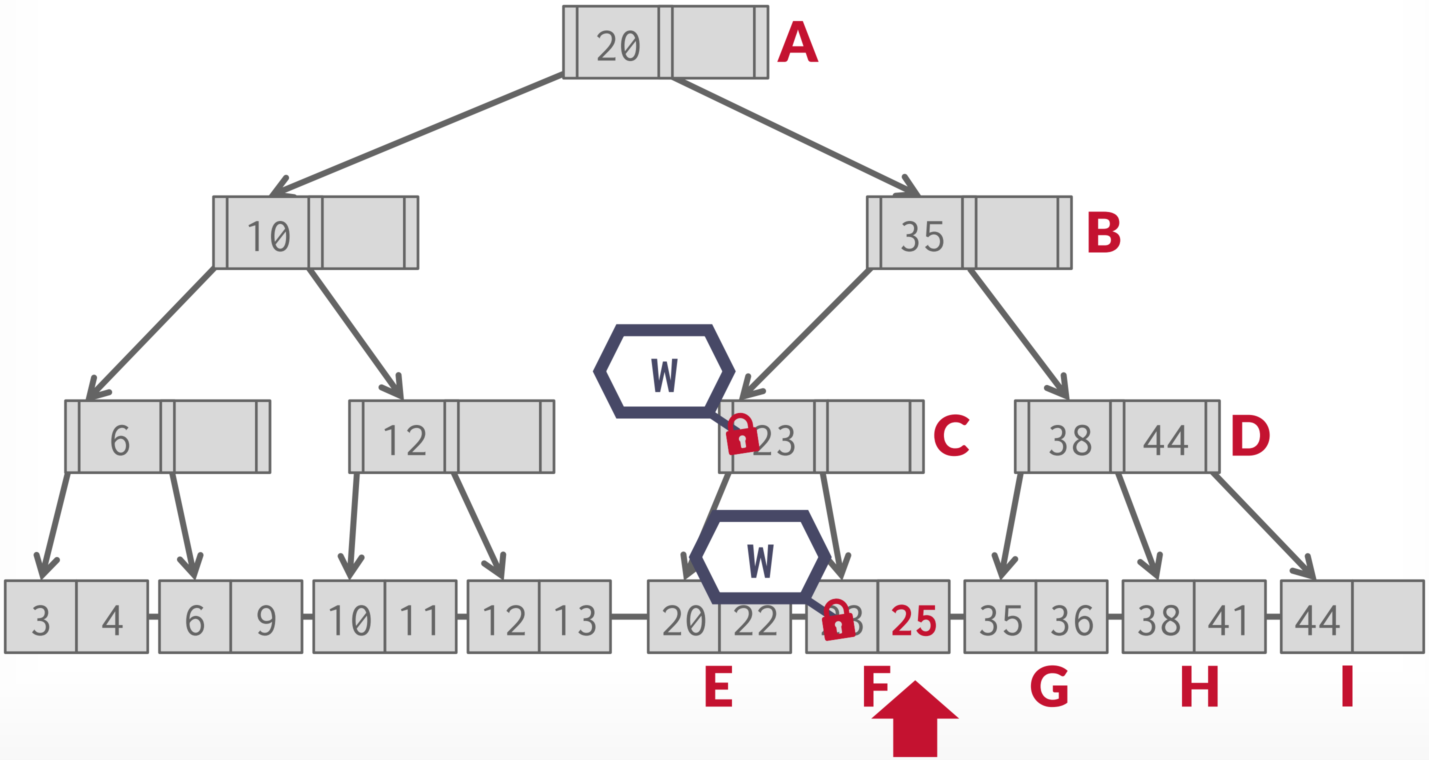


EXAMPLE #4 – INSERT 25

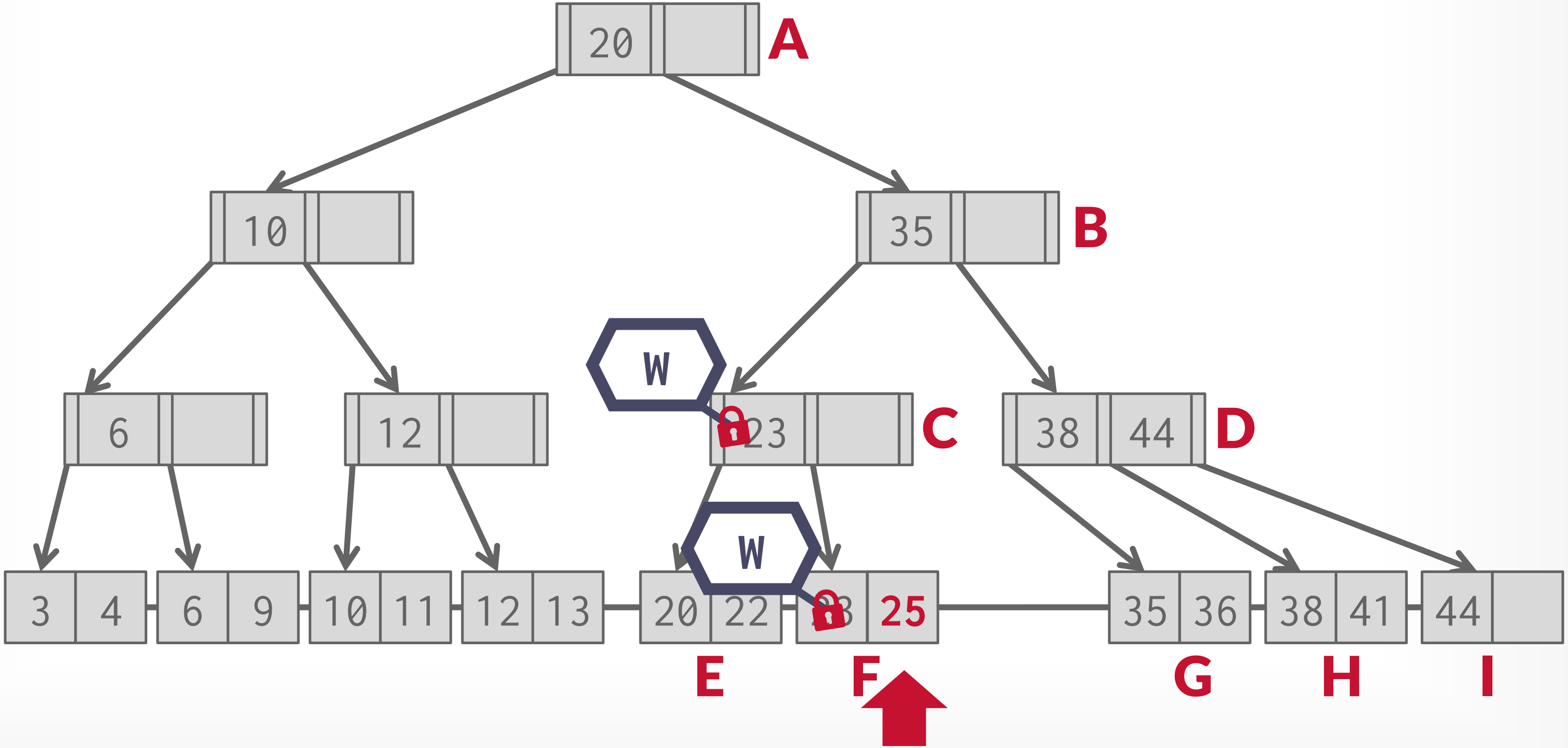


We need to split F, so we need to hold the latch on its parent node.

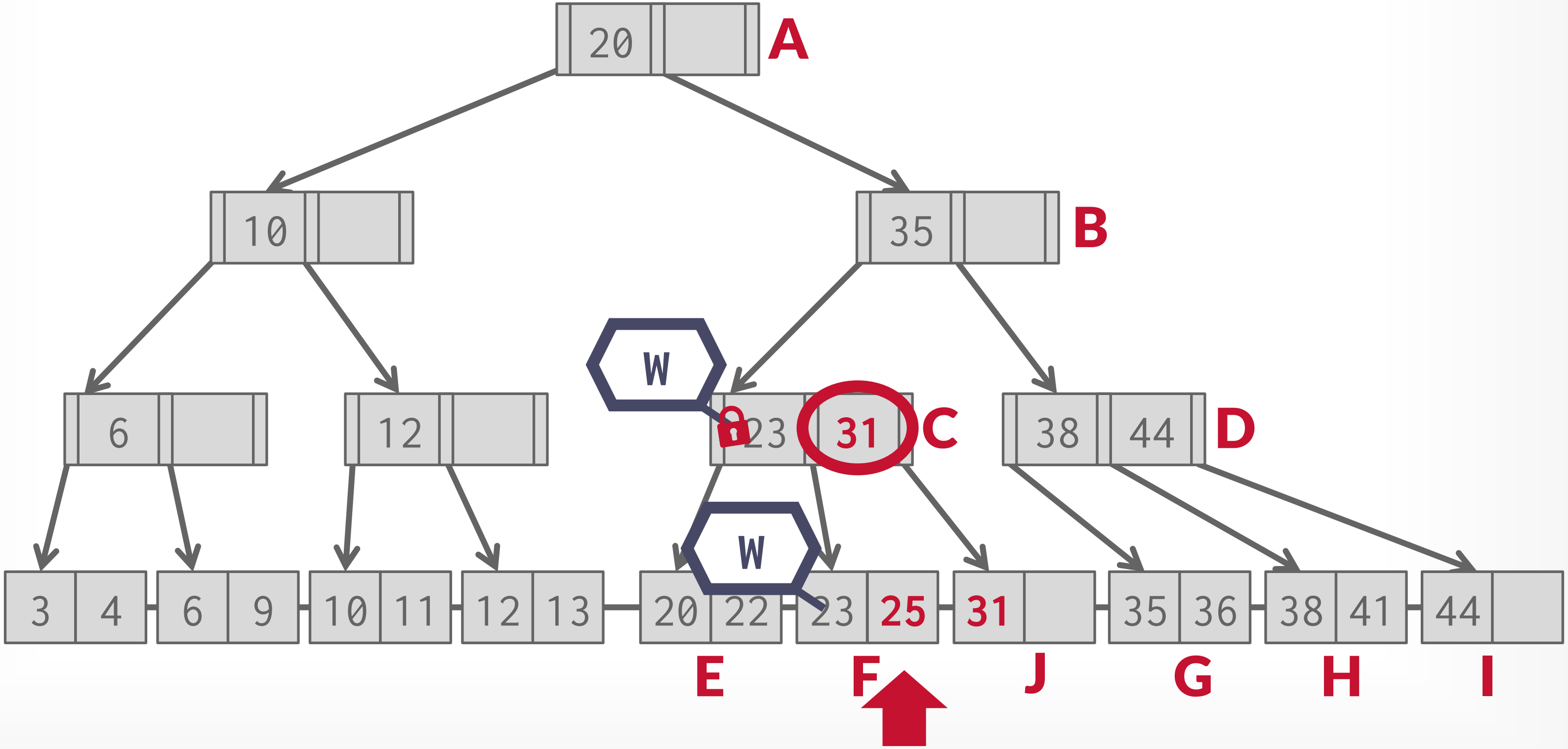
EXAMPLE #4 – INSERT 25



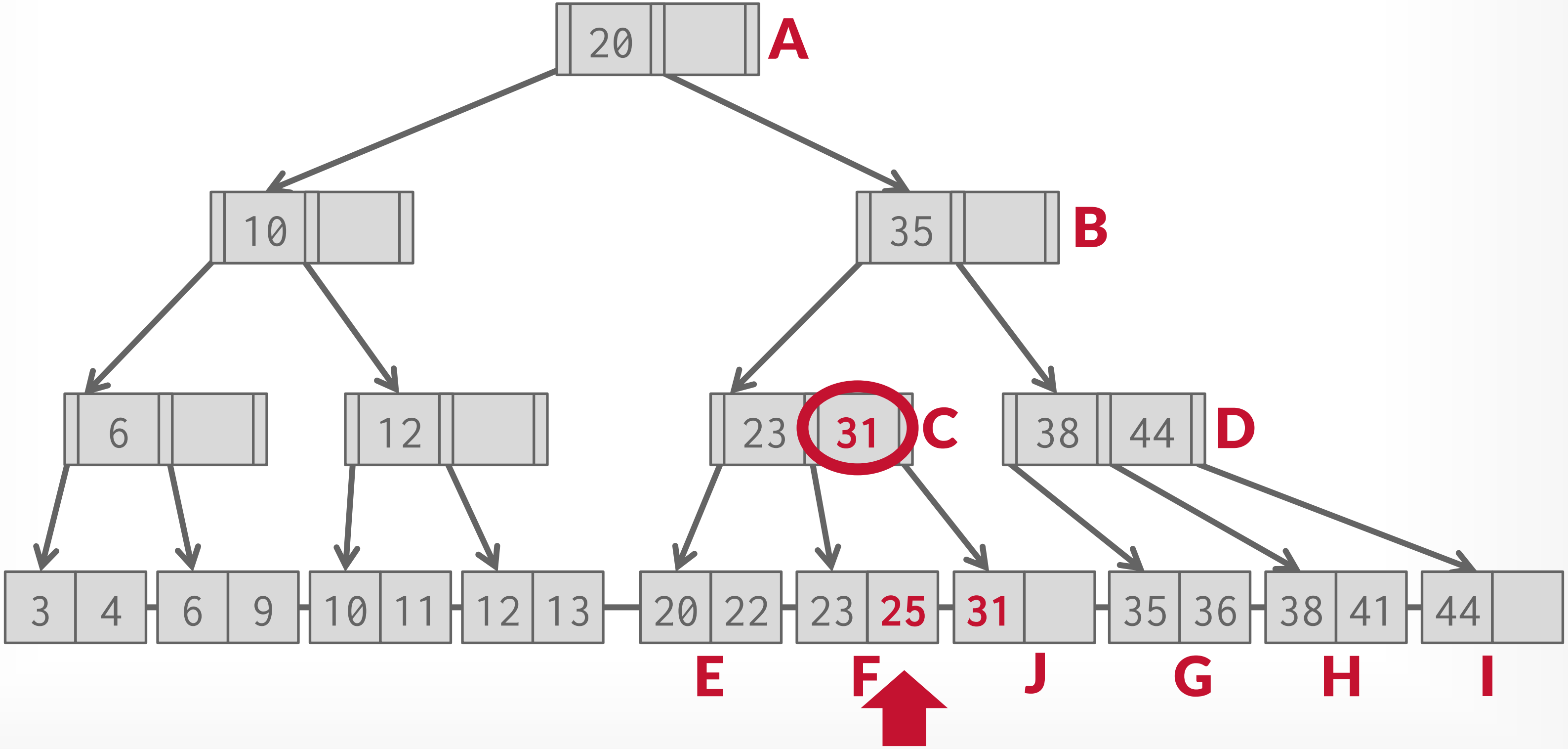
EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25

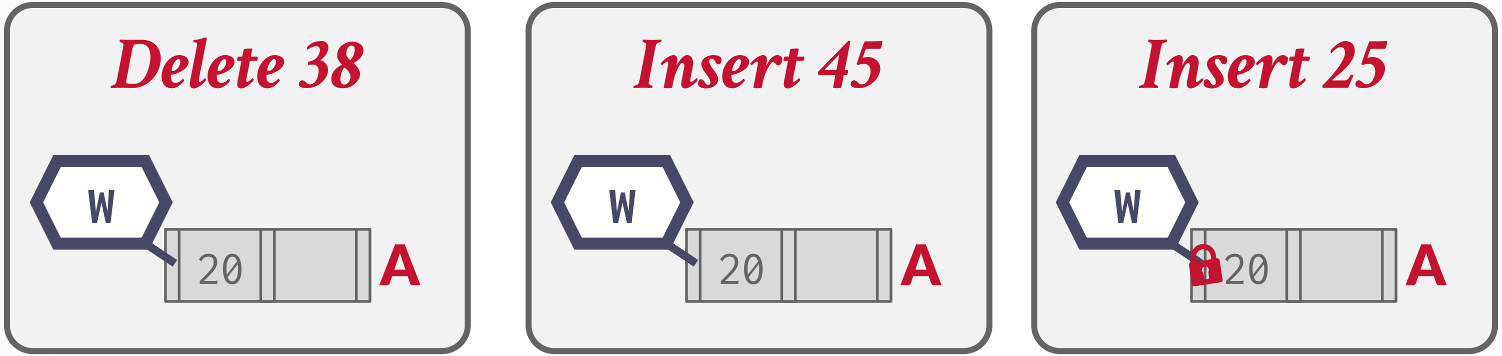


EXAMPLE #4 – INSERT 25



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

根节点 root node 是安全节点: 发生拆分/合并操作的概率极低, 可乐观地不加锁进行遍历, 提高并发度.

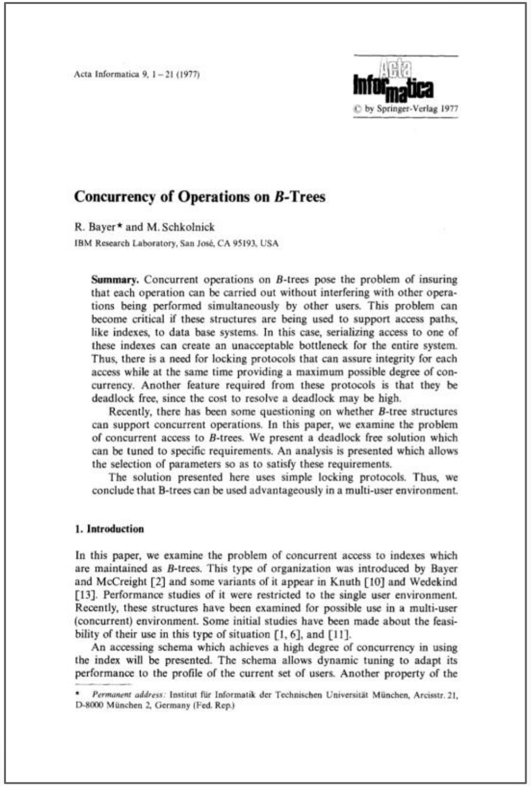
Most modifications to a B+Tree will not require a split or merge.

假设在拆分/合并操作很少发生的情况下遍历 B+Tree.

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

一直向下加读锁遍历到叶节点, 然后以写锁获取该节点进行操作.

If a worker guesses wrong, repeat traversal with pessimistic algorithm.



Acta Informatica 9, 1 – 21 (1977)

Acta Informatica

© by Springer-Verlag 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.

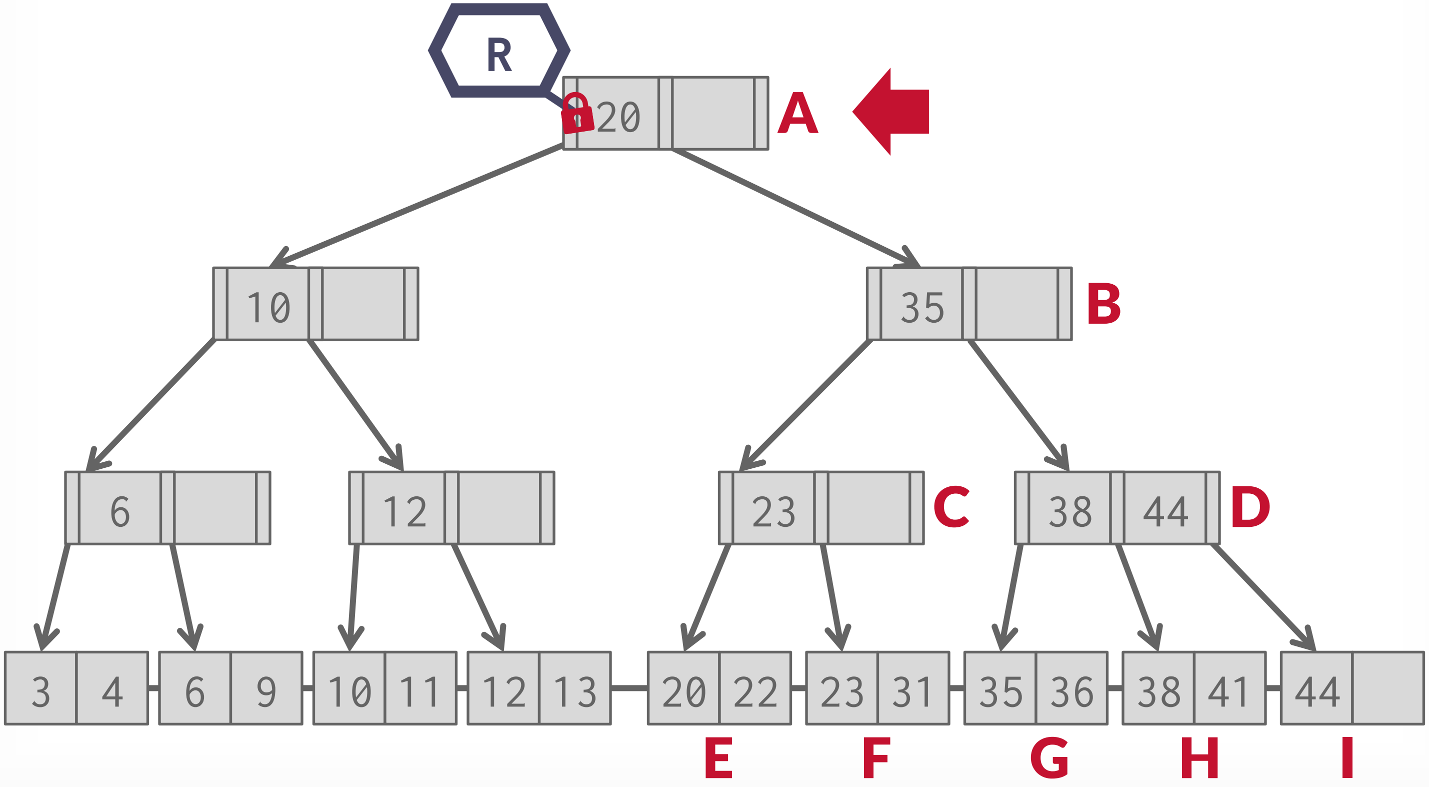
Insert/Delete:

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

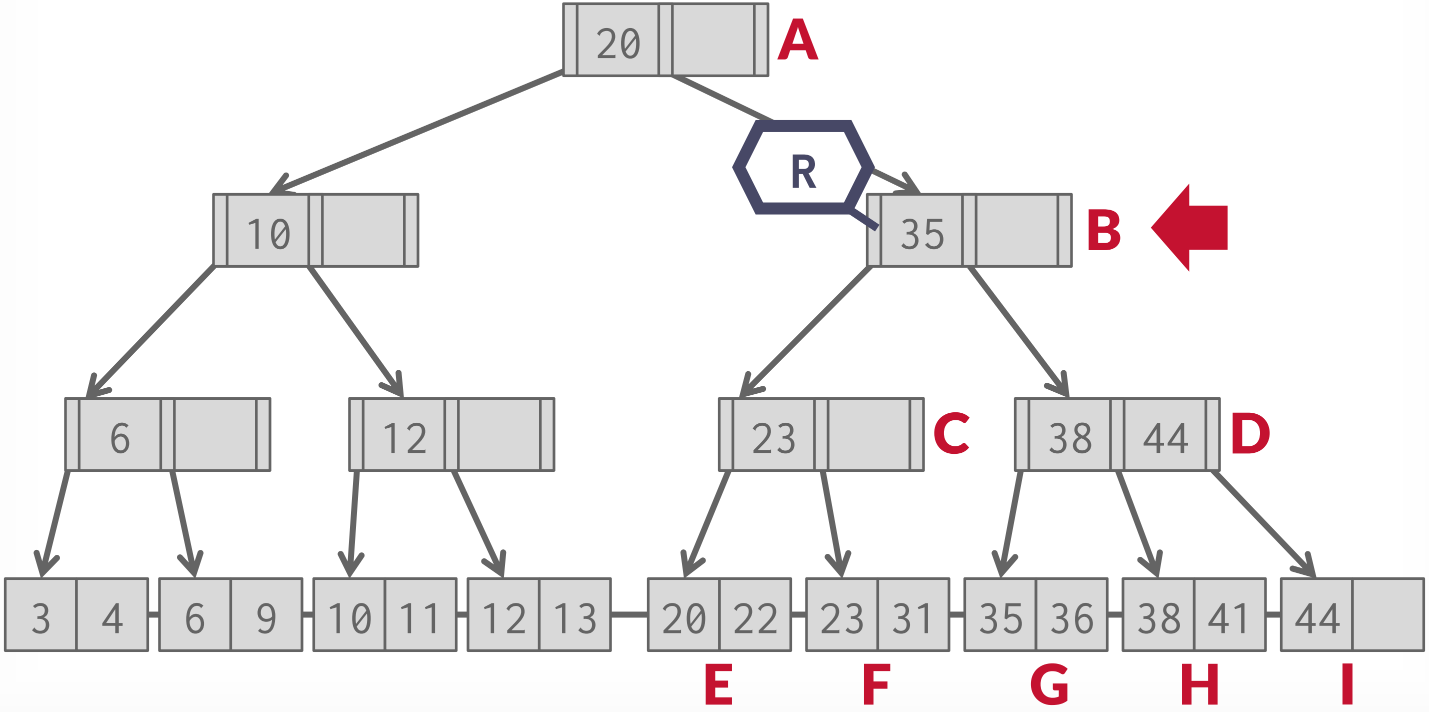
→ If leaf is not safe, release all latches, and restart thread using previous insert/delete protocol with write latches.

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

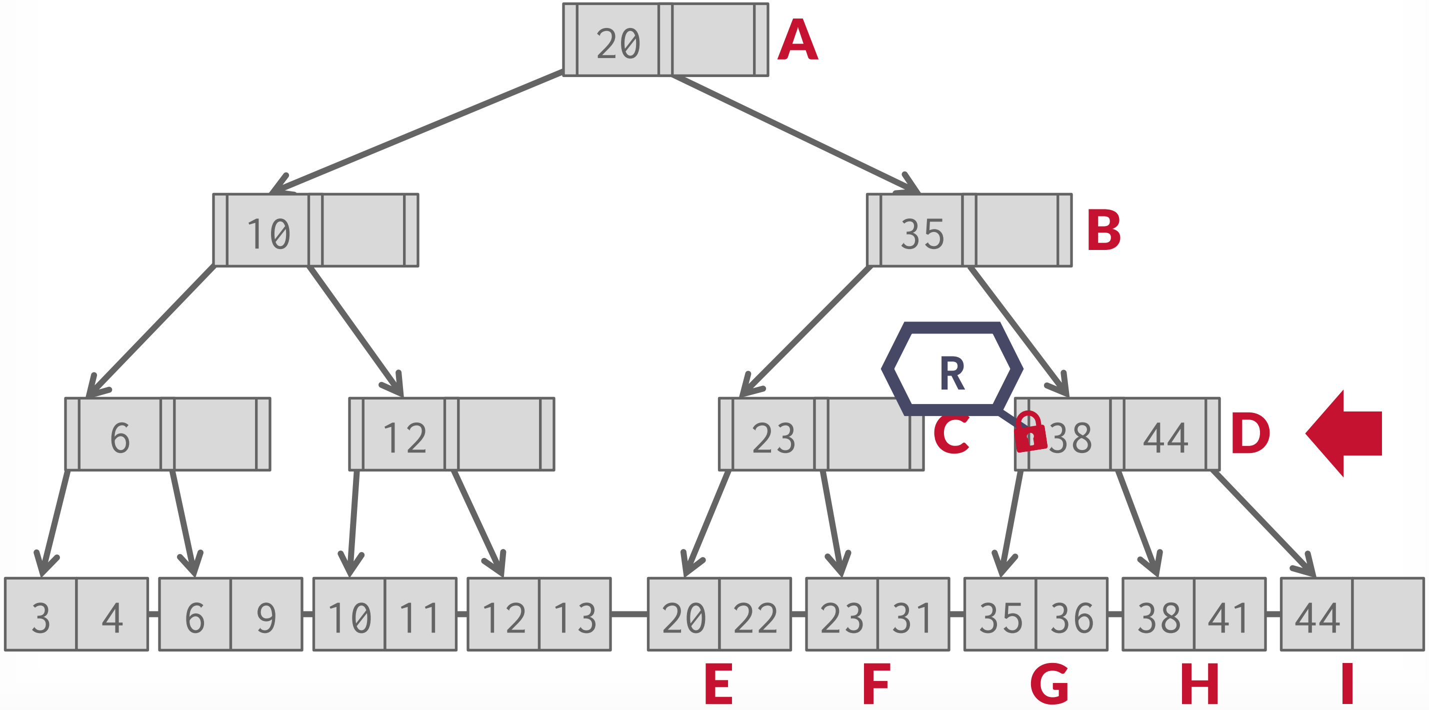
EXAMPLE #2 – DELETE 38



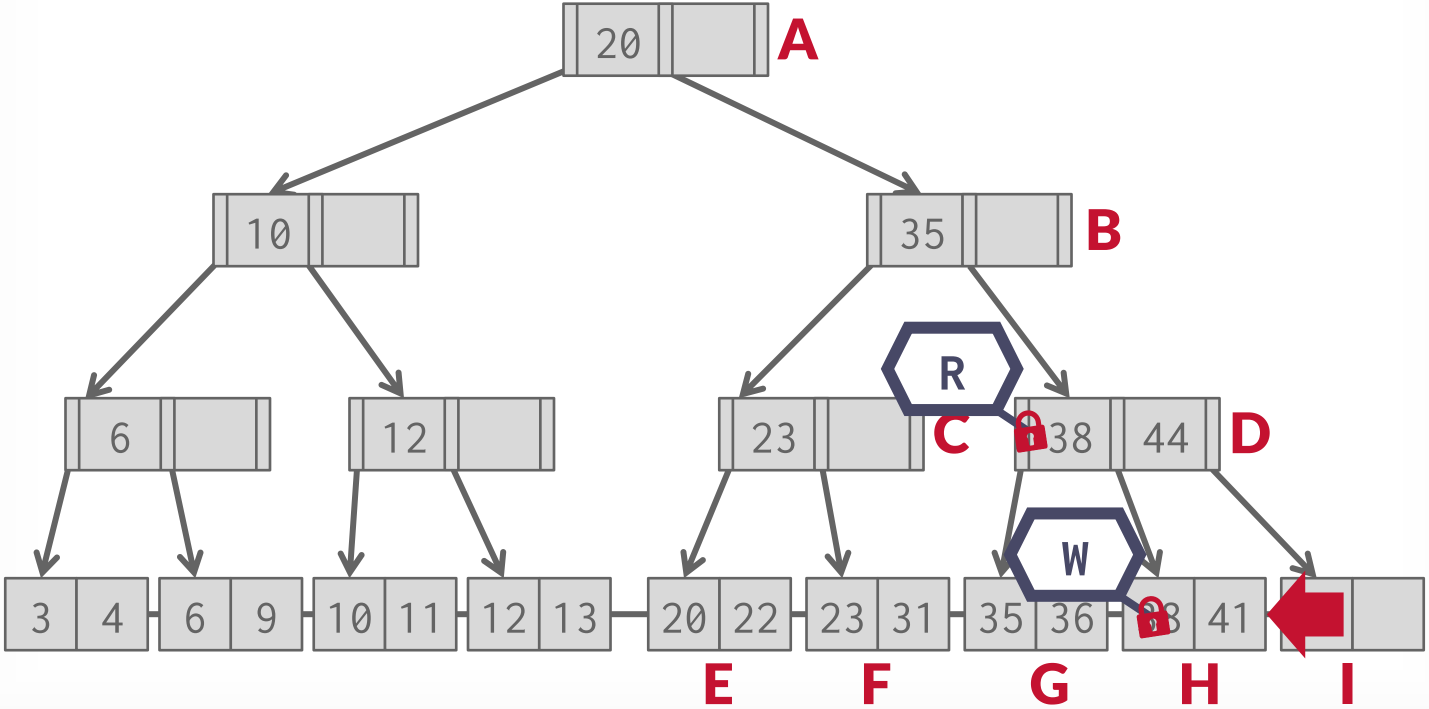
EXAMPLE #2 – DELETE 38



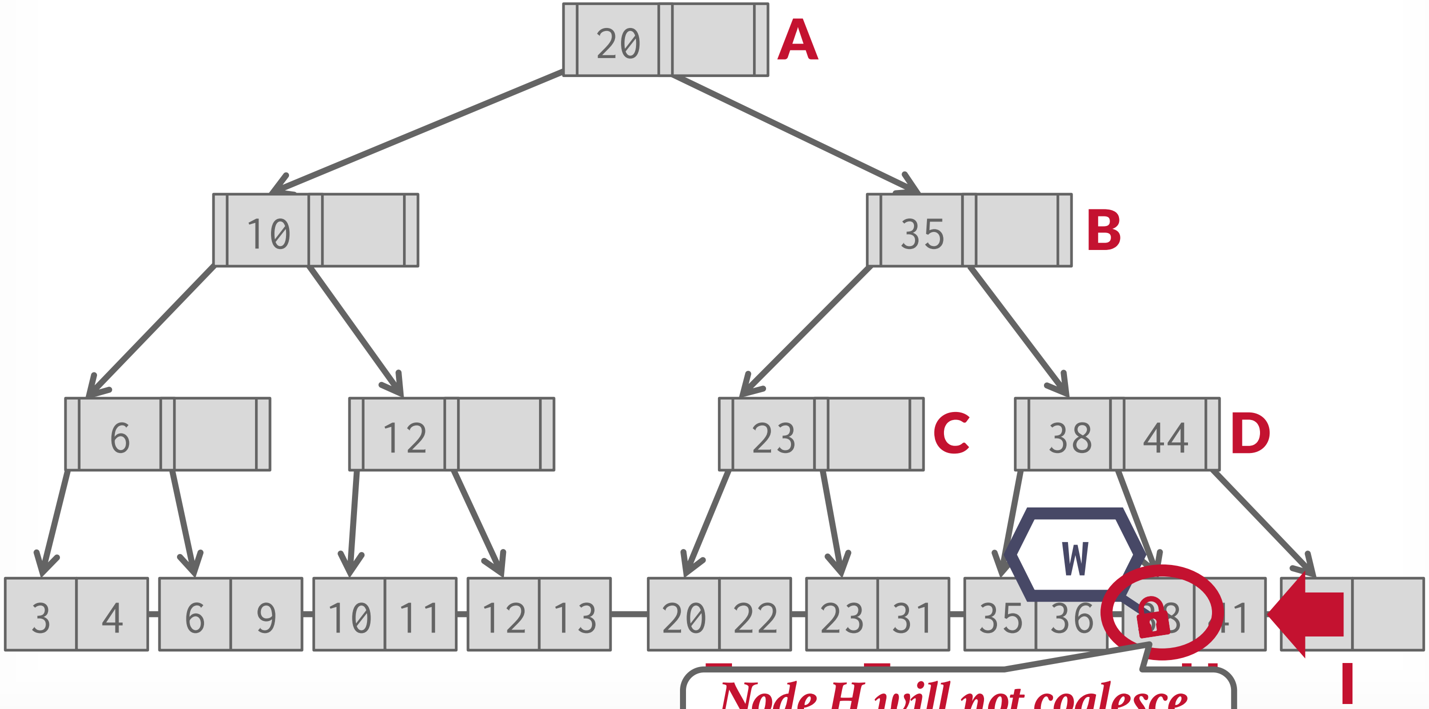
EXAMPLE #2 – DELETE 38



EXAMPLE #2 – DELETE 38

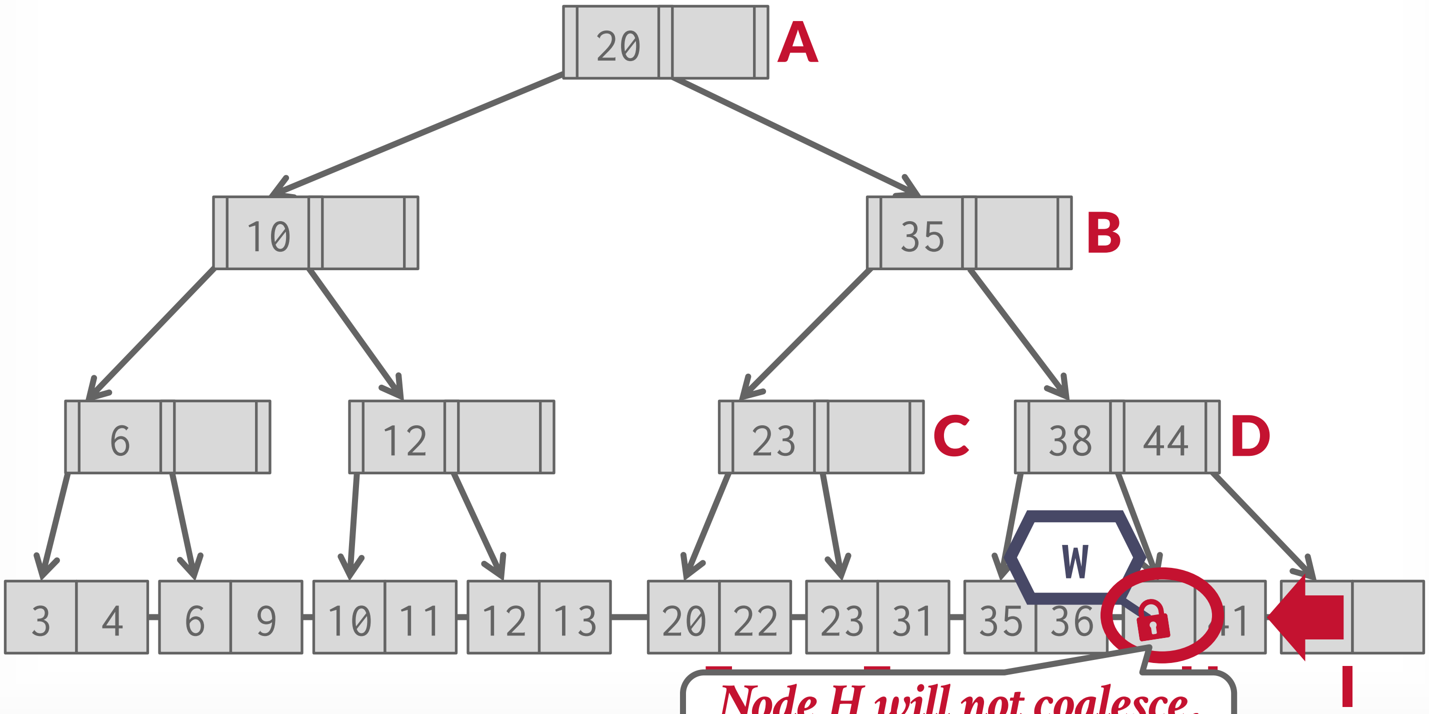


EXAMPLE #2 – DELETE 38



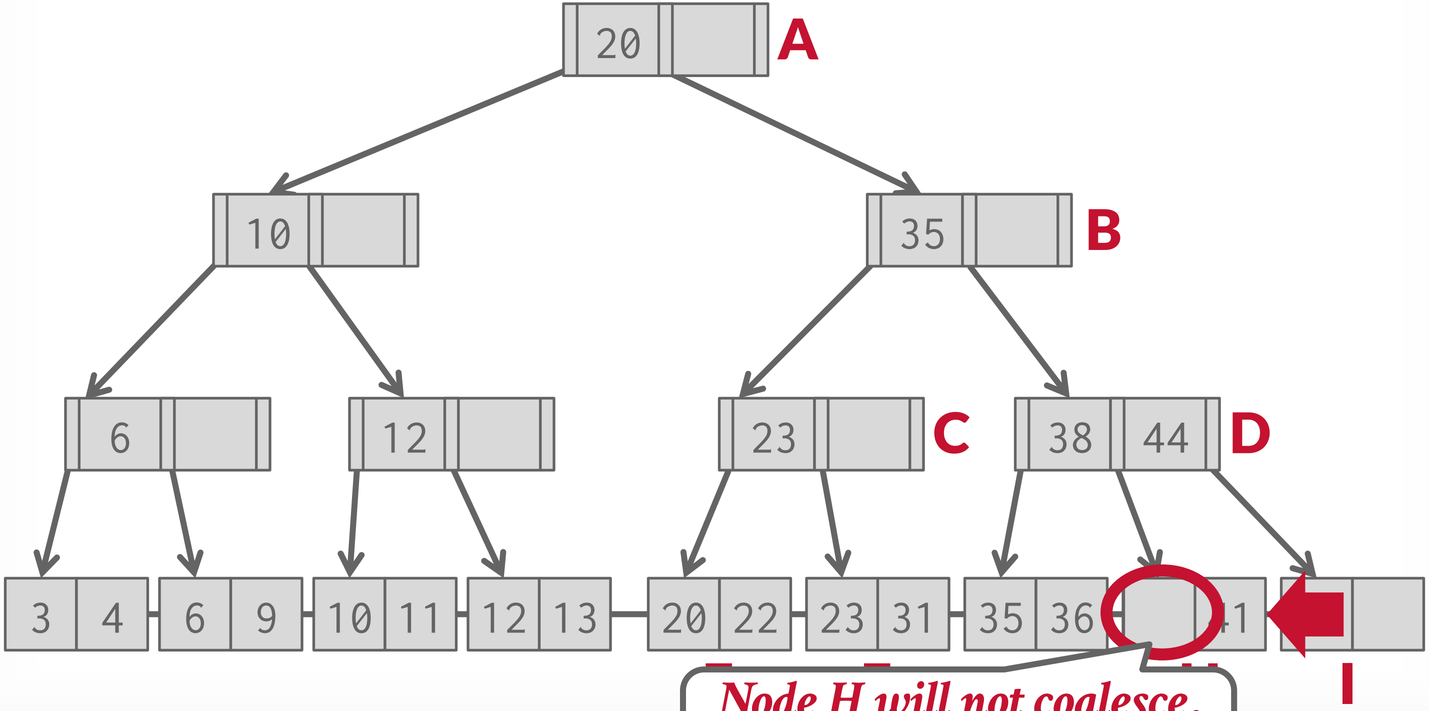
Node H will not coalesce, so we're safe!

EXAMPLE #2 – DELETE 38



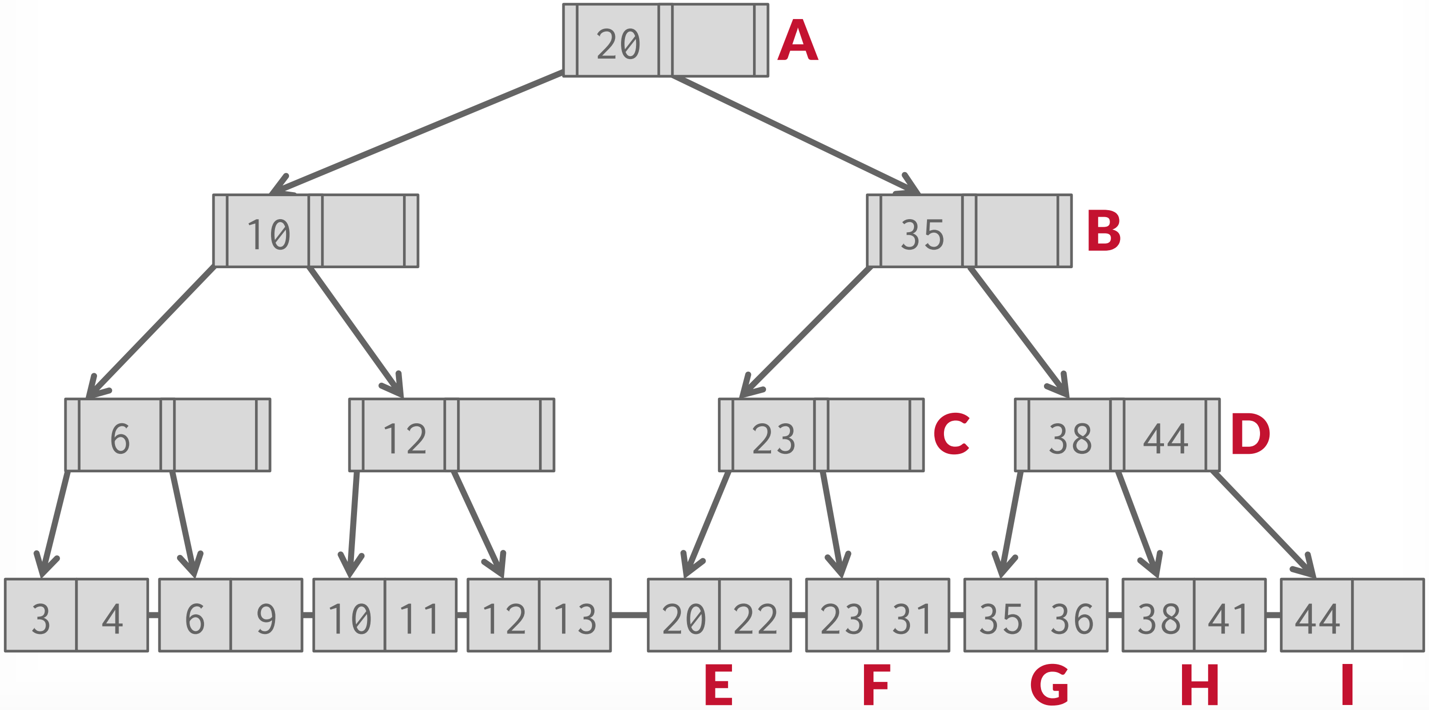
Node H will not coalesce, so we're safe!

EXAMPLE #2 – DELETE 38

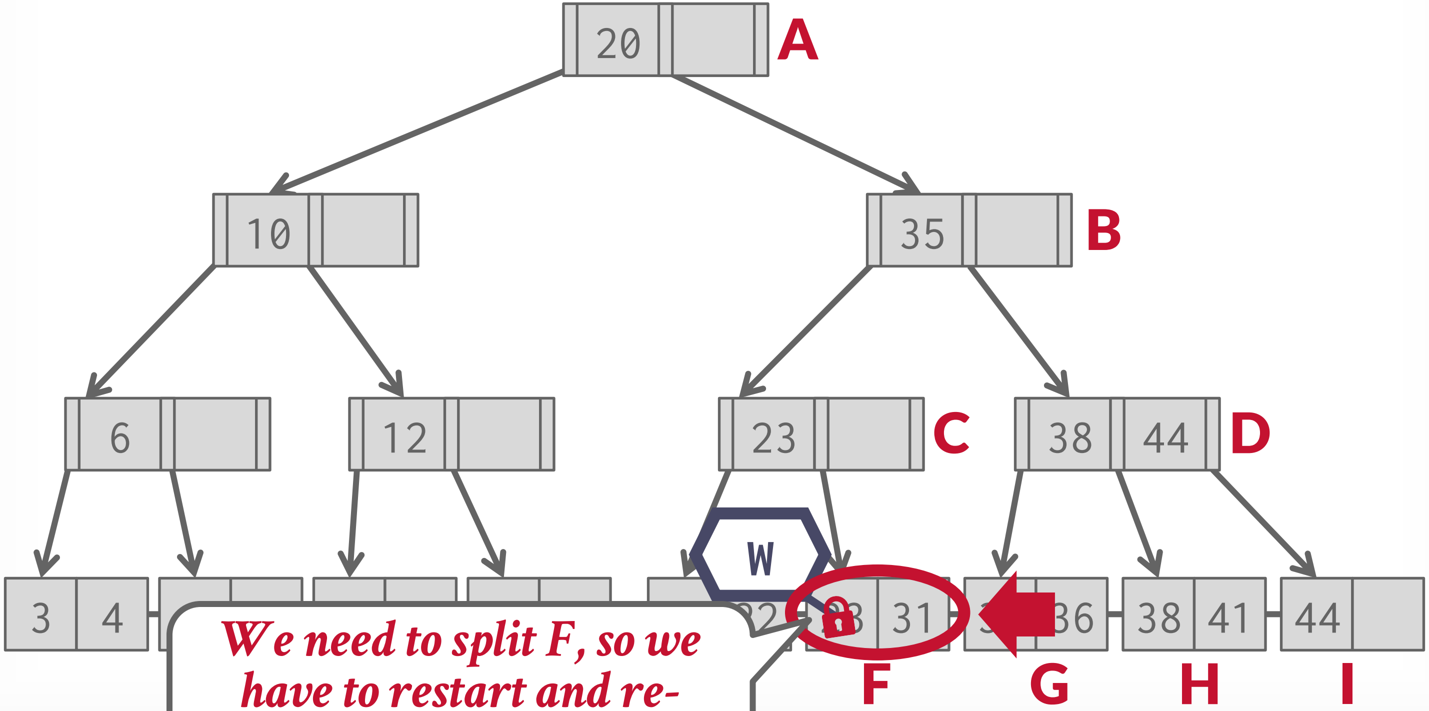


Node H will not coalesce, so we're safe!

EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25



We need to split F, so we have to restart and re-execute like before.

重新加写锁执行.

OBSERVATION

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

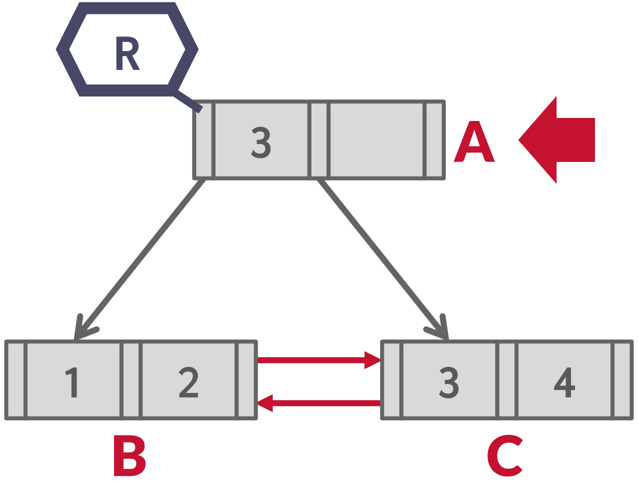
→ A thread can only acquire a latch from a node that is below its current node.

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

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

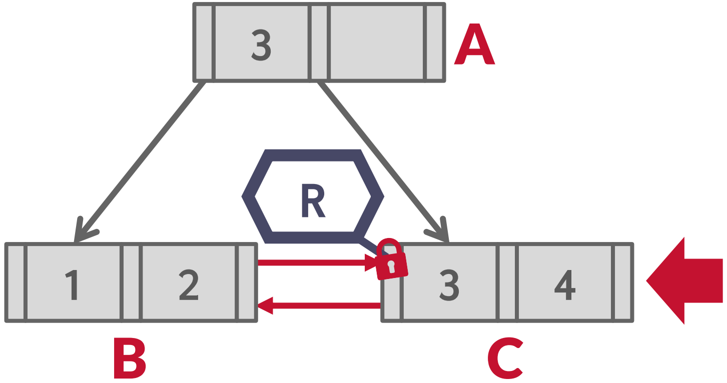
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



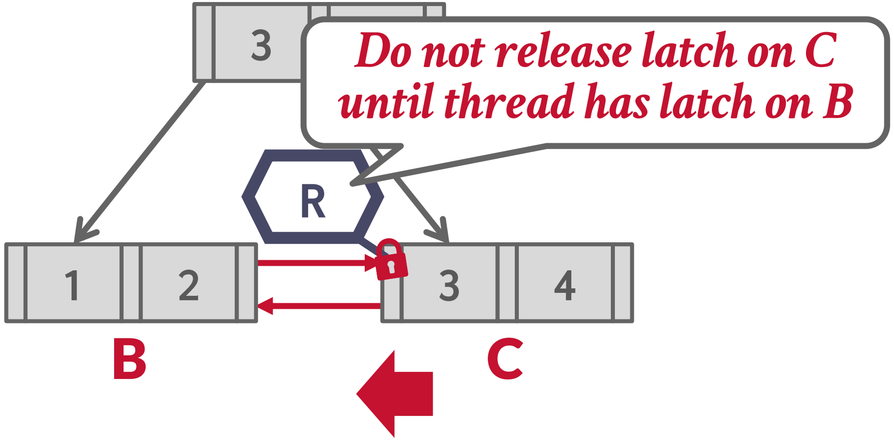
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



LEAF NODE SCAN EXAMPLE #1

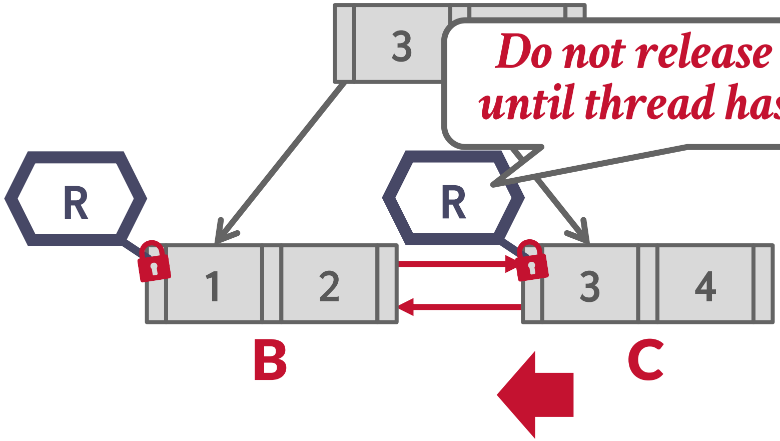
T$_1$: Find Keys < 4



Do not release latch on C until thread has latch on B

LEAF NODE SCAN EXAMPLE #1

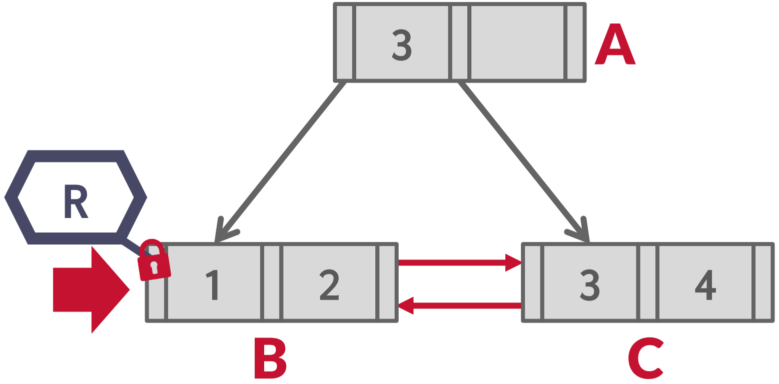
T_1: Find Keys < 4



Do not release latch on C until thread has latch on B

LEAF NODE SCAN EXAMPLE #1

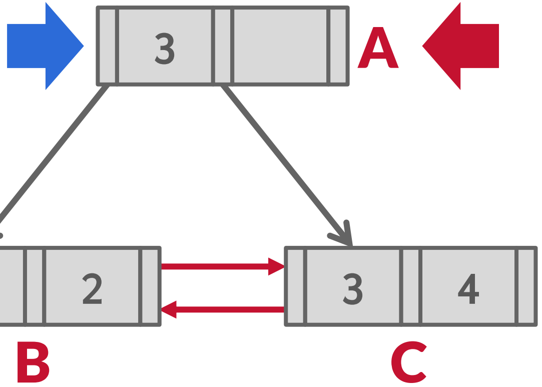
T_1: Find Keys < 4



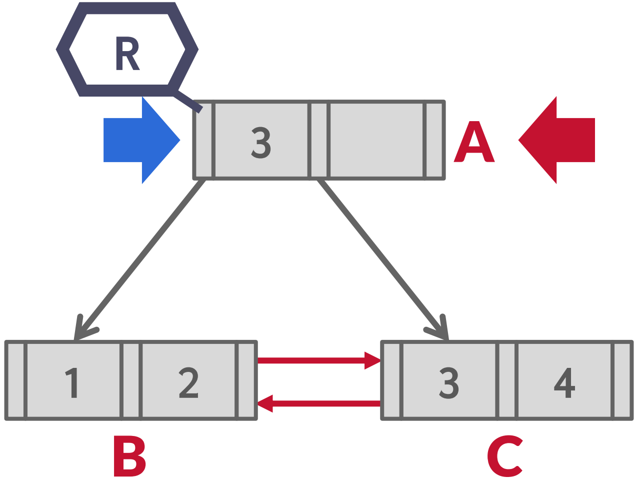
LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

T_2: Find Keys > 1



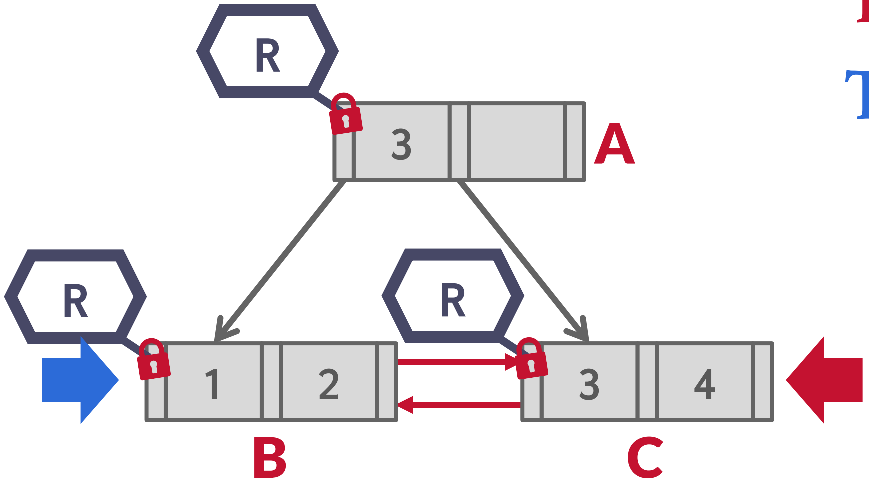
LEAF NODE SCAN EXAMPLE #2



T_1: Find Keys < 4

T_2: Find Keys > 1

LEAF NODE SCAN EXAMPLE #2



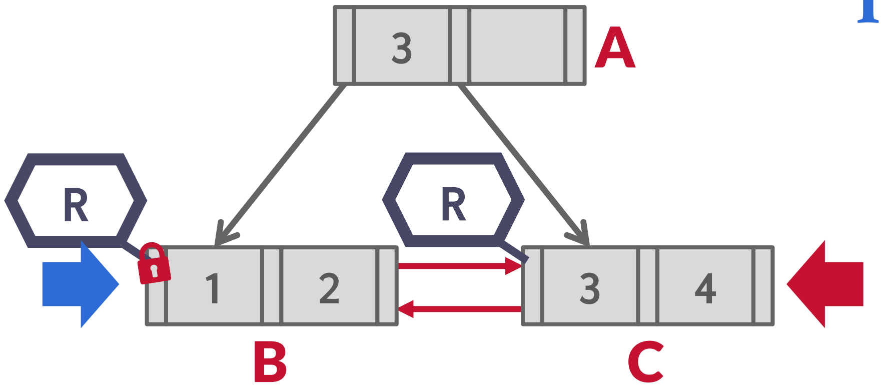
T_1: Find Keys < 4

T_2: Find Keys > 1

LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

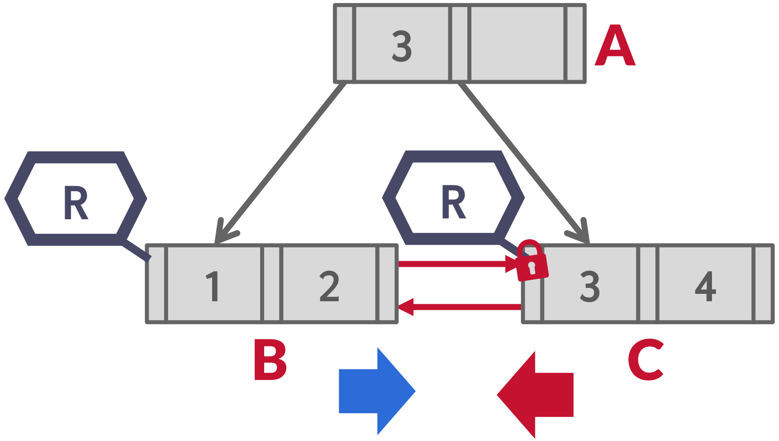
T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

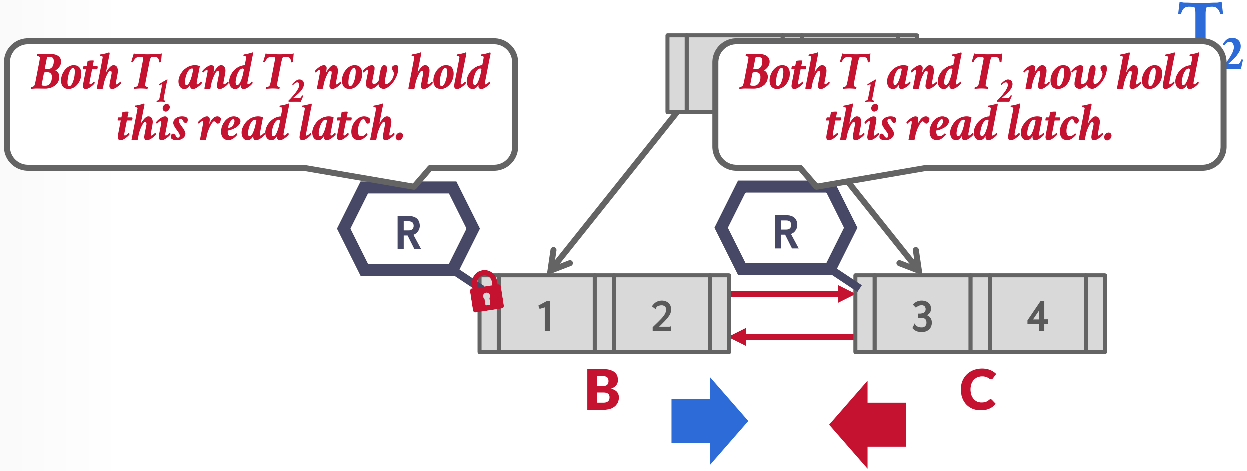
T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

T_2: Find Keys > 1



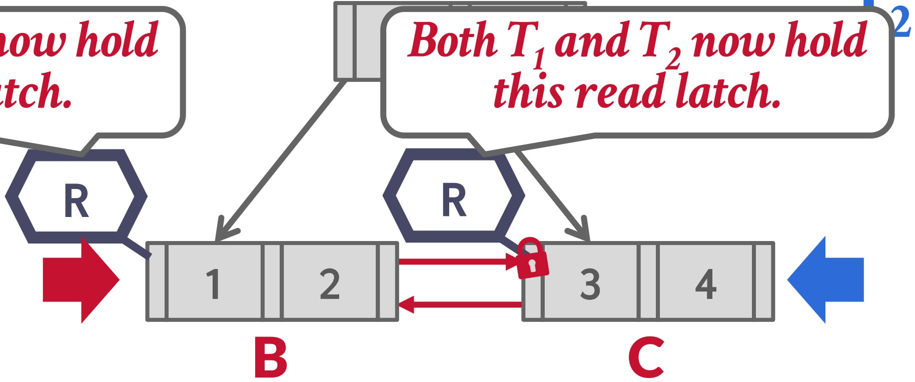
Both T_1 and T_2 now hold this read latch.

Both T_1 and T_2 now hold this read latch.

LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

T_2: Find Keys > 1



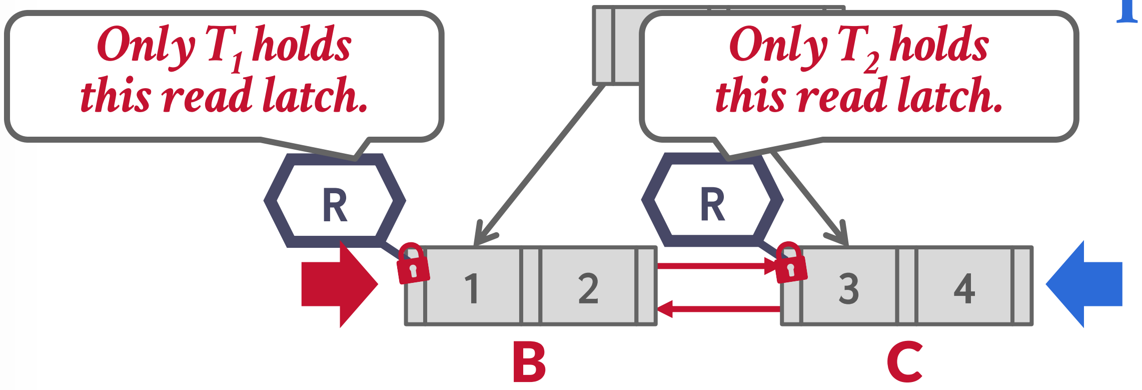
Both T_1 and T_2 now hold this read latch.

Both T_1 and T_2 now hold this read latch.

LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

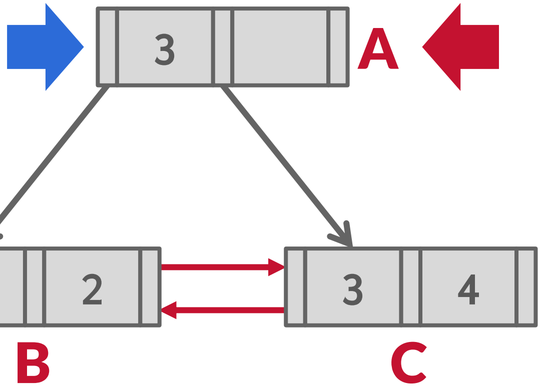
T_2: Find Keys > 1


Only T_1 holds this read latch.
Only T_2 holds this read latch.
R
R
1
2
3
4
B
C

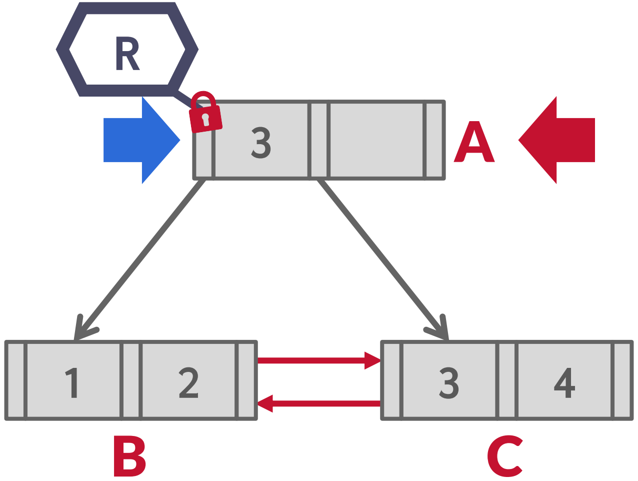
LEAF NODE SCAN EXAMPLE #3

T_1: Delete 4

T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #3



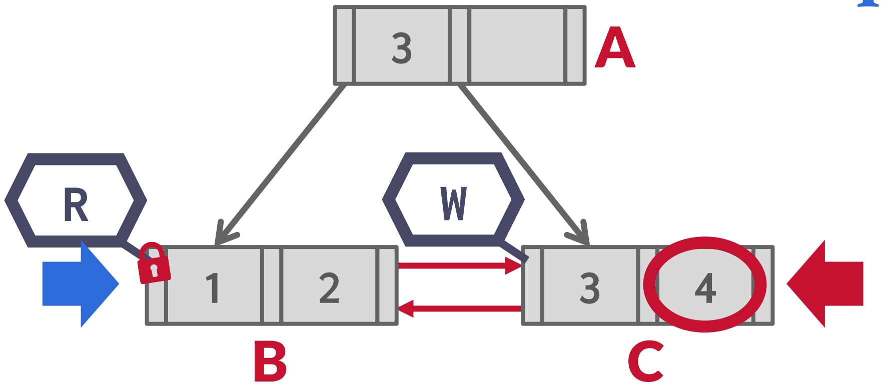
T$_1$: Delete 4

T$_2$: Find Keys > 1

LEAF NODE SCAN EXAMPLE #3

T_1: Delete 4

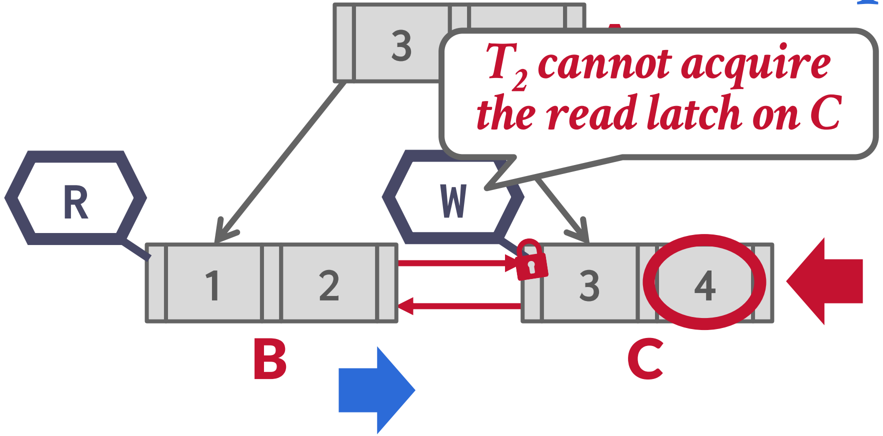
T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #3

T_1: Delete 4

T_2: Find Keys > 1

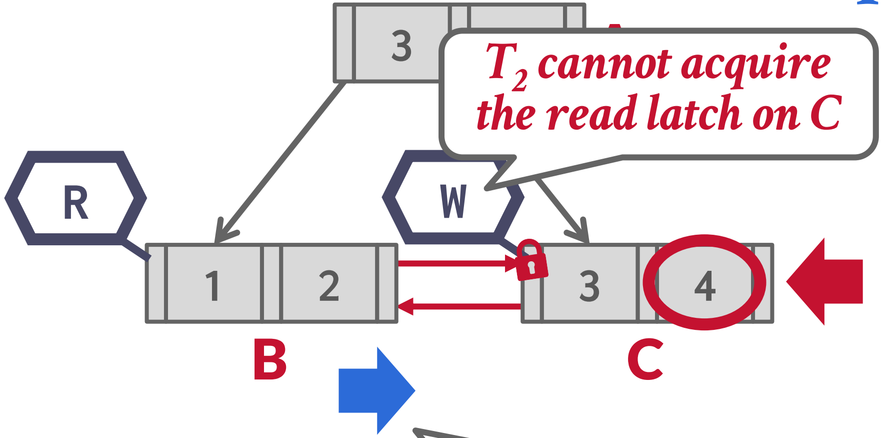


T_2 cannot acquire the read latch on C

LEAF NODE SCAN EXAMPLE #3

T_1: Delete 4

T_2: Find Keys > 1



T_2 cannot acquire the read latch on C

T_2 does not know what T_1 is doing…

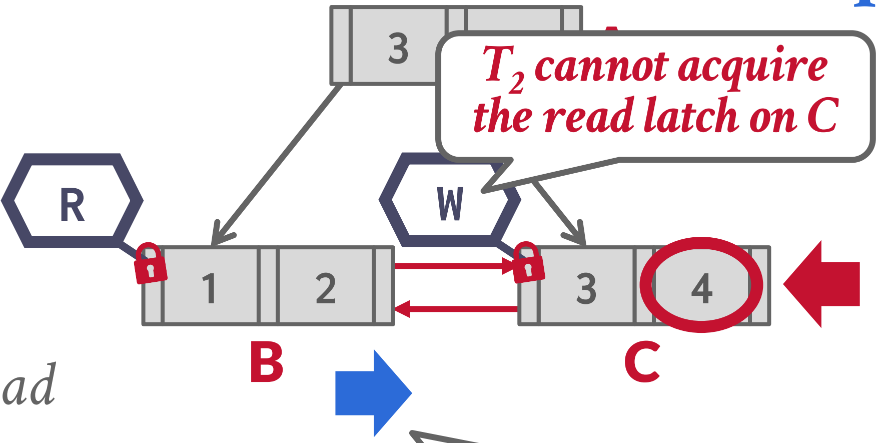
LEAF NODE SCAN EXAMPLE #3

T_1: Delete 4

T_2: Find Keys > 1

T_2 Choices?

- ⏳ *Wait*
- ☠ *Kill Ourself*
- *Kill Other Thread*



T_2 cannot acquire the read latch on C

T_2 does not know what T_1 is doing…

LEAF NODE SCAN EXAMPLE #3

T_1: Delete 4

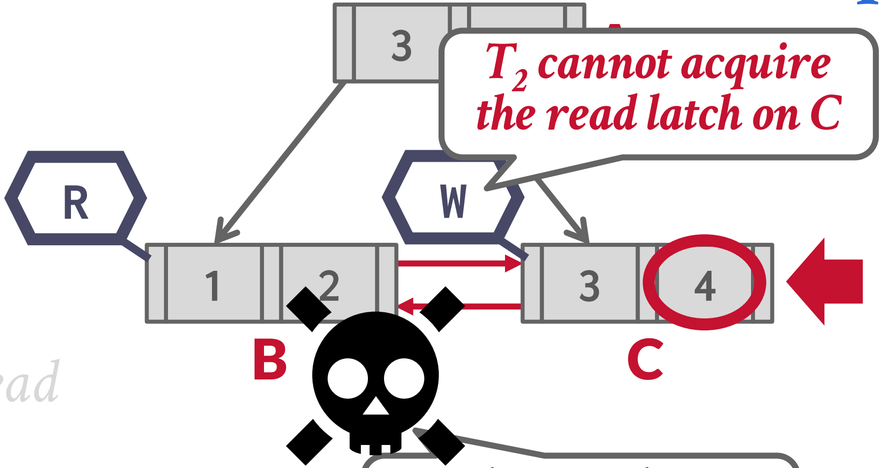
T_2: Find Keys > 1

T_2 Choices?

Wait

☠ *Kill Ourself*

Kill Other Thread



T_2 cannot acquire the read latch on C

T_2 does not know what T_1 is doing…

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.

→ Usually transparent to end-user / application.

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

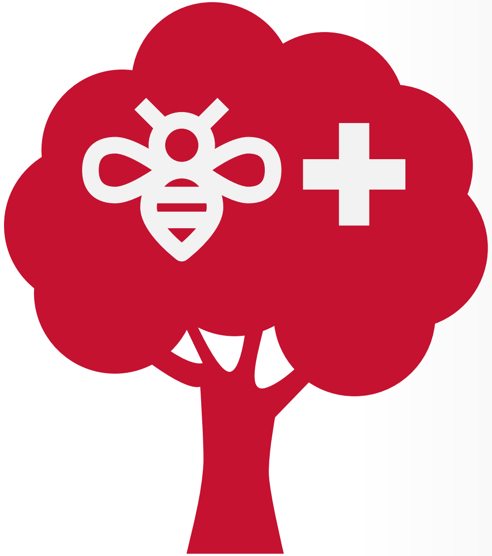
We are finally going to discuss how to execute some queries…

PROJECT #2

You will build a thread-safe B+tree backed by your buffer pool manager.

→ Page Layout
→ Insert/Delete/Find Operations
→ Iterator
→ Latch Crabbing

We define the API for you. You need to provide the method implementations.



WARNING:
This is more difficult than Project #1.
Start immediately!

https://15445.courses.cs.cmu.edu/fall2024/project2

TASKS

Task #1: Page Layouts

→ How each node will store its key/values in a page.
→ You only need to support unique keys.

Task #2: Operations

→ Support point queries (single key).
→ Support inserts with node splitting.
→ Support removal of keys with sibling stealing + merging.
→ Does not need to be thread-safe.

TASKS

Task #3: Index Iterator

→ Create a STL iterator for range scans on leaf nodes.
→ You only need to support ascending scans.

Task #4: Concurrent Index

→ Introduce latch crabbing/coupling protocol to support safe concurrent operations.
→ Make sure you have splits / merges working correctly before proceeding with this task.

DEVELOPMENT HINTS

Follow the textbook semantics and algorithms.

Set the page size to be small (e.g., 512B) when you first start so that you can see more splits/merges.

Make sure that you protect the internal B+Tree **root_page_id** member.

EXTRA CREDIT

Gradescope Leaderboard runs your code with a specialized in-memory version of BusTub.

The top 20 fastest implementations in the class will receive extra credit for this assignment.

→ **#1:** 50% bonus points
→ **#2–10:** 25% bonus points
→ **#11–20:** 10% bonus points

You must pass all the test cases to qualify!

PLAGIARISM WARNING

The homework and projects must be your own original work. They are **not** group assignments.

You may **not** copy source code from other people or the web.

Plagiarism is **not** tolerated. You will get lit up.

→ Please ask me if you are unsure.

See CMU's Policy on Academic Integrity for additional information.