Intro to Databases (COMP_SCI 339)

07 Index Concurrency Control

Northwestern University

WINTER 2024

Andrew Crotty

ADMINISTRIVIA

Project #2 is due Sunday 2/4 @ 11:59pm

Project #3 will be released tonight and is due Sunday 2/18 @ 11:59pm

OBSERVATION

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

But a DBMS should allow multiple threads to concurrently access data structures to take advantage of additional CPU cores and hide disk I/O stalls.

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 transactions.
→ Held for transaction 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

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

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** → **pthread_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** → **pthread_mutex** → **futex**

```
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** → **pthread_mutex** → **futex**

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

OS Latch
Userspace Latch

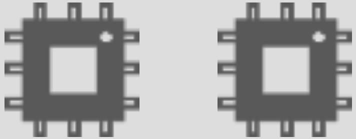
LATCH IMPLEMENTATIONS

Approach #1: Blocking OS Mutex

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

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

OS Latch
Userspace Latch

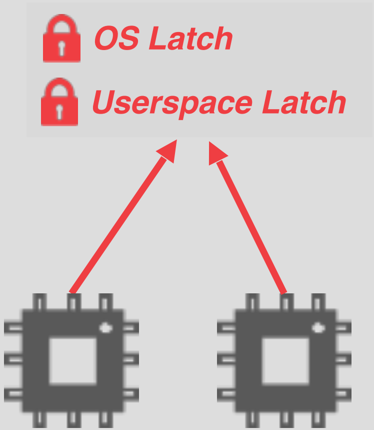


LATCH IMPLEMENTATIONS

Approach #1: Blocking OS Mutex

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

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



LATCH IMPLEMENTATIONS

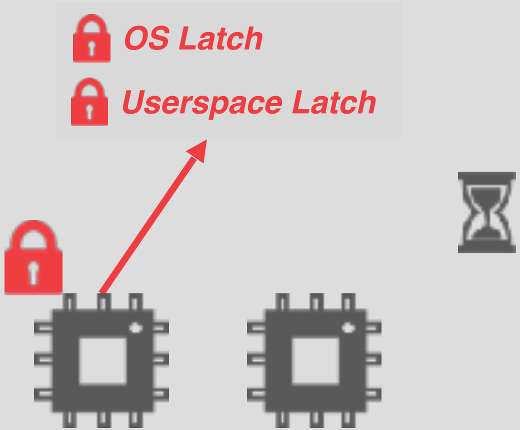
Approach #1: Blocking OS Mutex

→ Simple to use

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

→ Example: **std::mutex** → **pthread_mutex** → **futex**

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



LATCH IMPLEMENTATIONS

Approach #2: 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**

LATCH IMPLEMENTATIONS

Approach #2: 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**

LATCH IMPLEMENTATIONS

Approach #2: 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**

Latch

LATCH IMPLEMENTATIONS

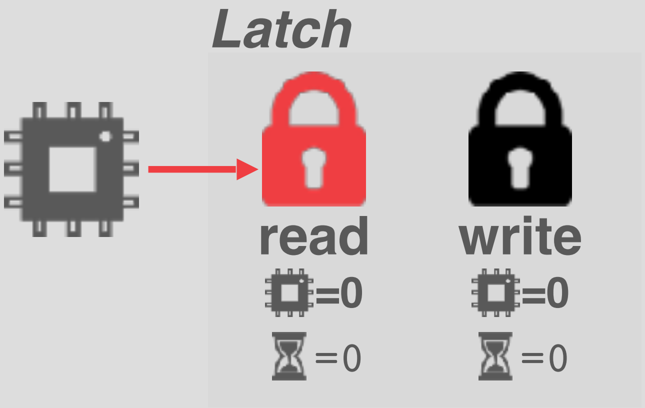
Approach #2: Reader-Writer Latches

→ 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**

LATCH IMPLEMENTATIONS

Approach #2: Reader-Writer Latches

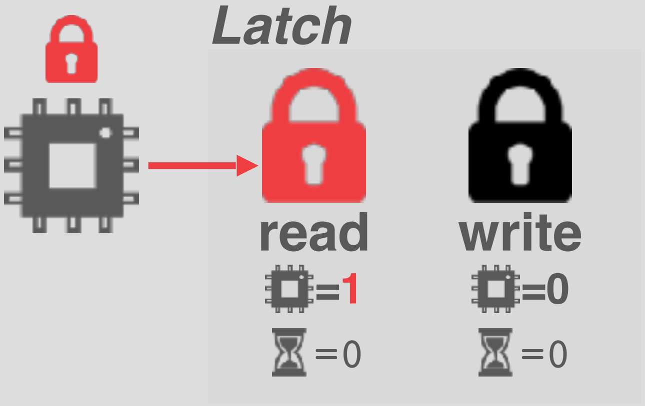
→ Allows for concurrent readers. Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.
→ Example: **std::shared_mutex** → **pthread_rwlock**



LATCH IMPLEMENTATIONS

Approach #2: Reader-Writer Latches

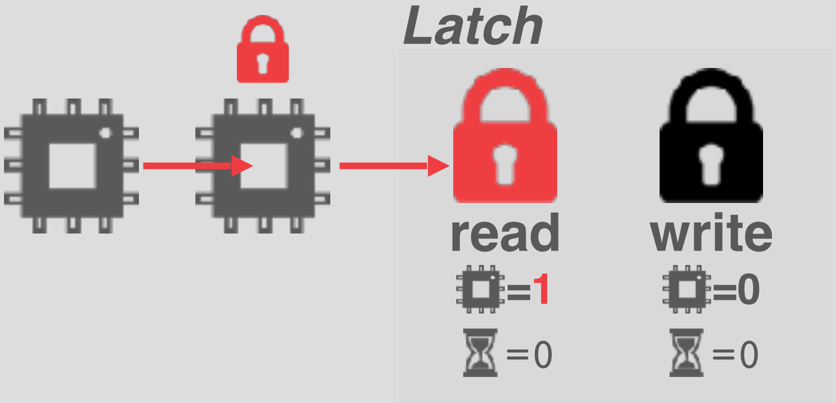
→ Allows for concurrent readers. Must manage read/ write queues to avoid starvation.
→ Can be implemented on top of spinlocks.
→ Example: **std::shared_mutex** → **pthread_rwlock**



LATCH IMPLEMENTATIONS

Approach #2: Reader-Writer Latches

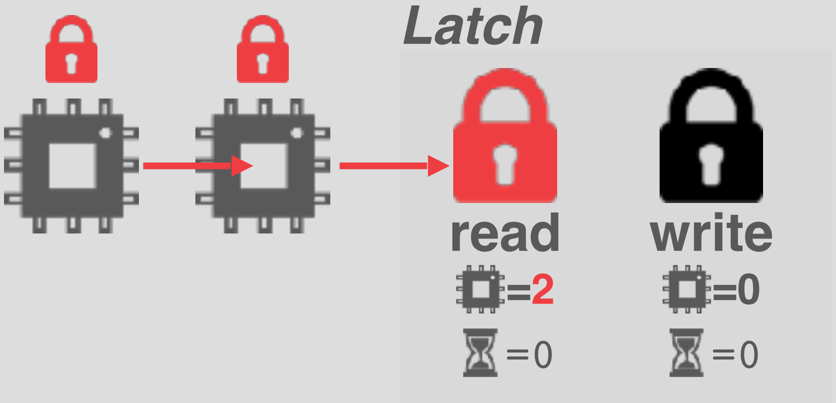
→ Allows for concurrent readers. Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.
→ Example: **std::shared_mutex** → **pthread_rwlock**



LATCH IMPLEMENTATIONS

Approach #2: Reader-Writer Latches

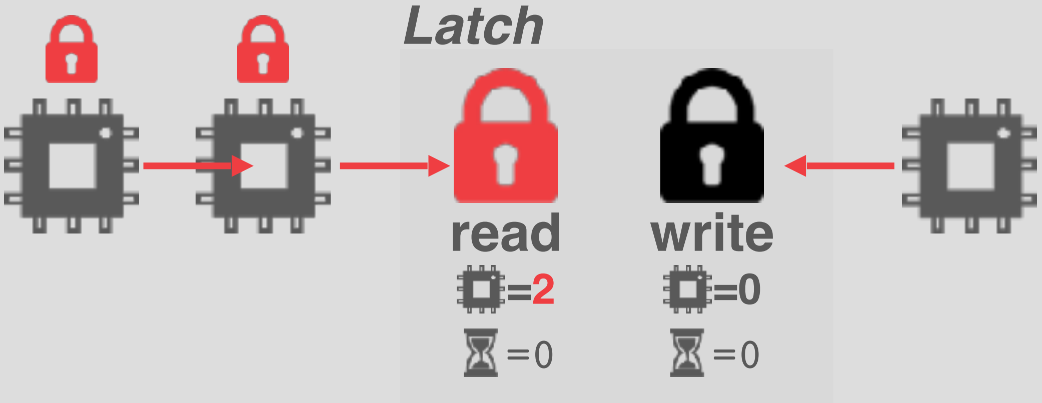
→ Allows for concurrent readers. Must manage read/write queues to avoid starvation.
→ Can be implemented on top of spinlocks.
→ Example: **std::shared_mutex** → **pthread_rwlock**



LATCH IMPLEMENTATIONS

Approach #2: 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**



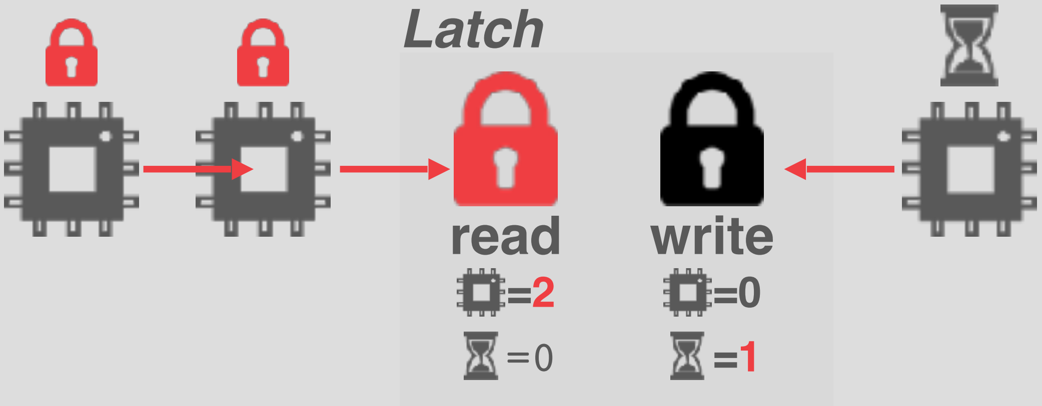
LATCH IMPLEMENTATIONS

Approach #2: 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**



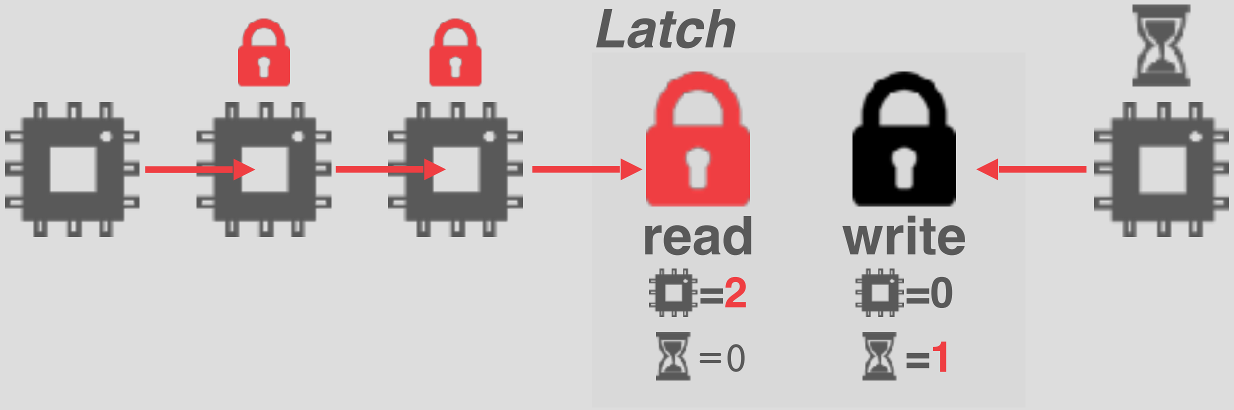
LATCH IMPLEMENTATIONS

Approach #2: 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**



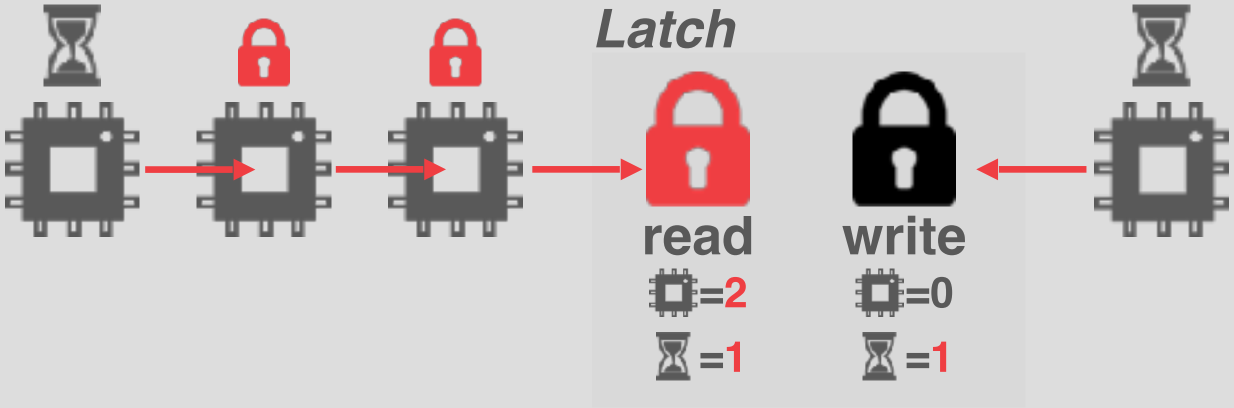
LATCH IMPLEMENTATIONS

Approach #2: 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**



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

Address *New Value*

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

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

M

30

Address *New Value*

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

Compare Value

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

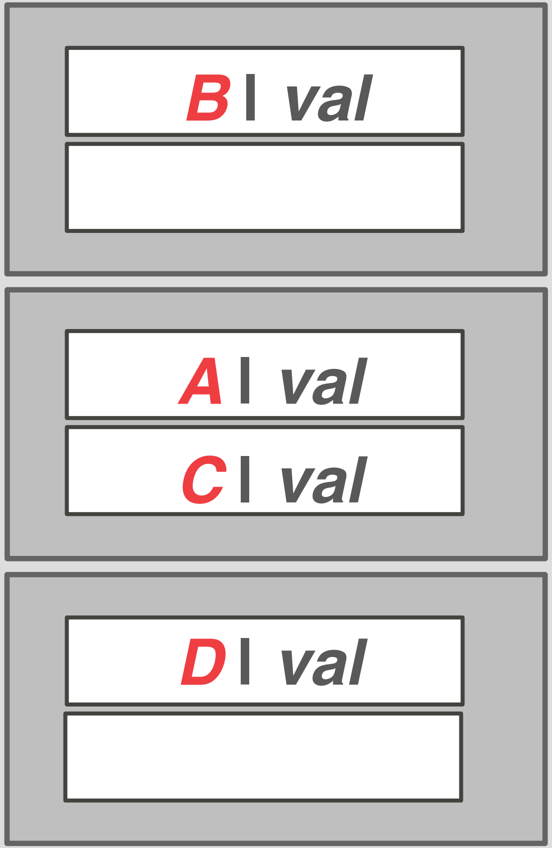
Approach #1: Page Latches

→ 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

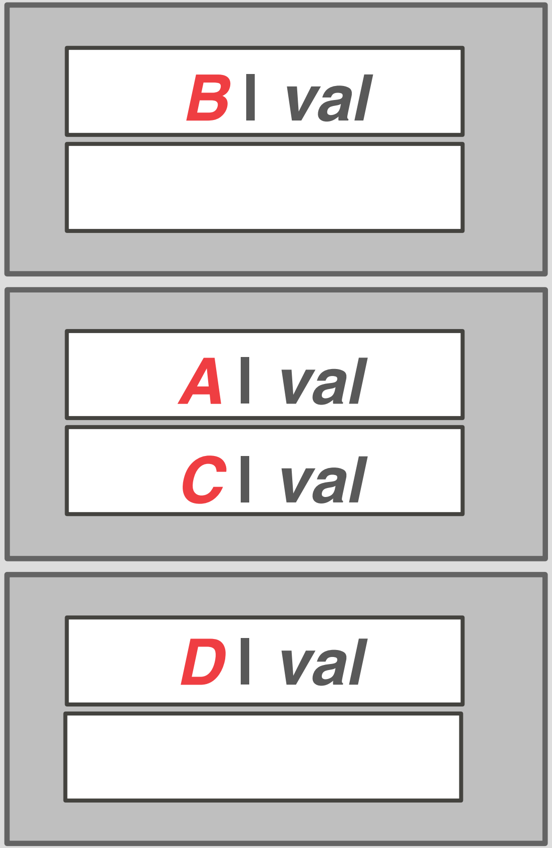
→ Each slot has its own latch.
→ Can use a single-mode latch to reduce meta-data and computational overhead.

HASH TABLE – PAGE LATCHES

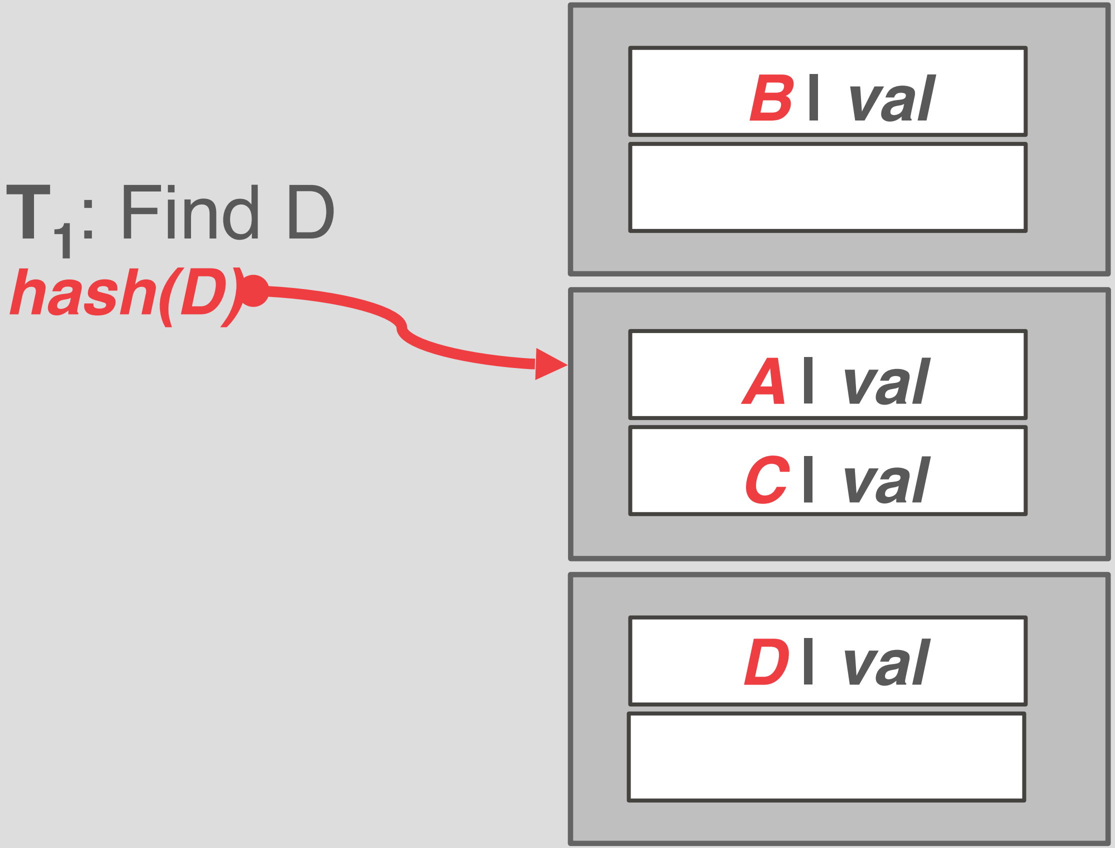


HASH TABLE – PAGE LATCHES

T_1: Find D
hash(D)



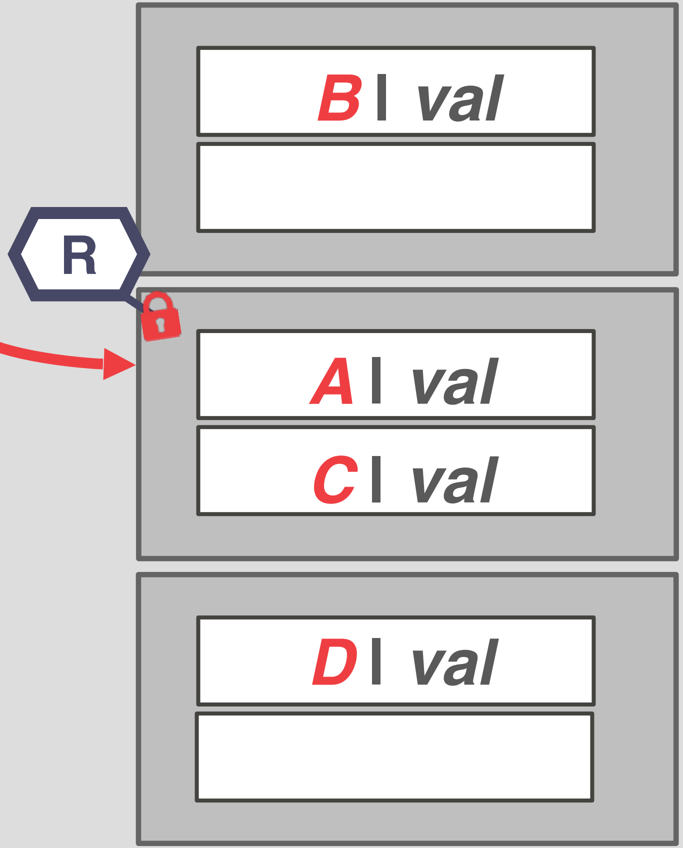
HASH TABLE – PAGE LATCHES



HASH TABLE – PAGE LATCHES

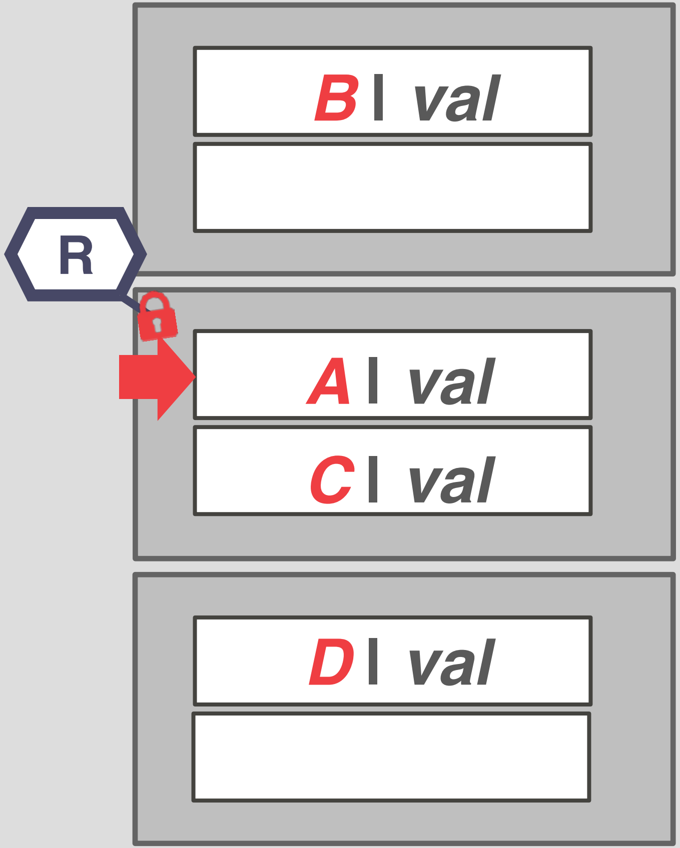
T_1: Find D

hash(D)



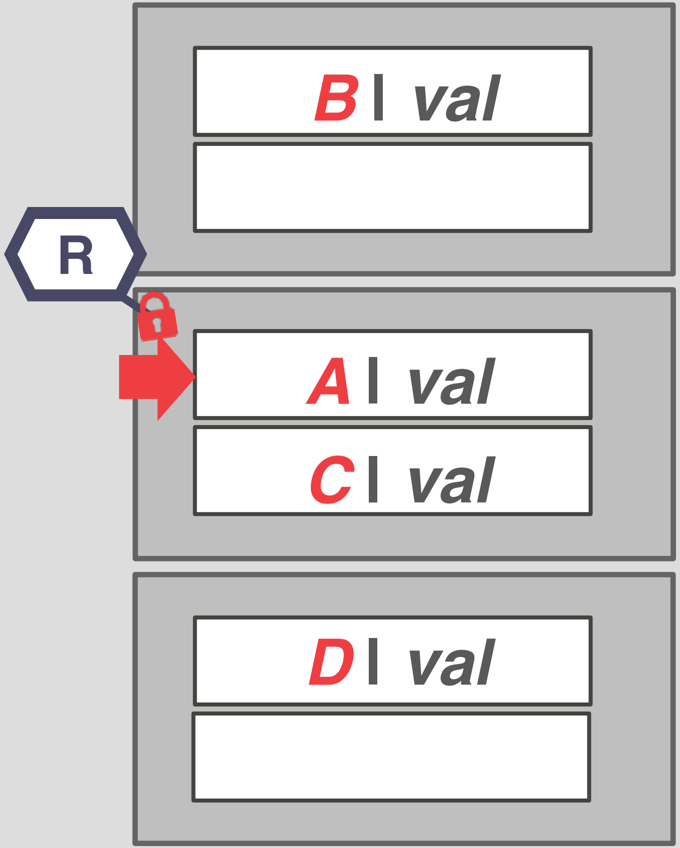
HASH TABLE – PAGE LATCHES

T_1: Find D
hash(D)



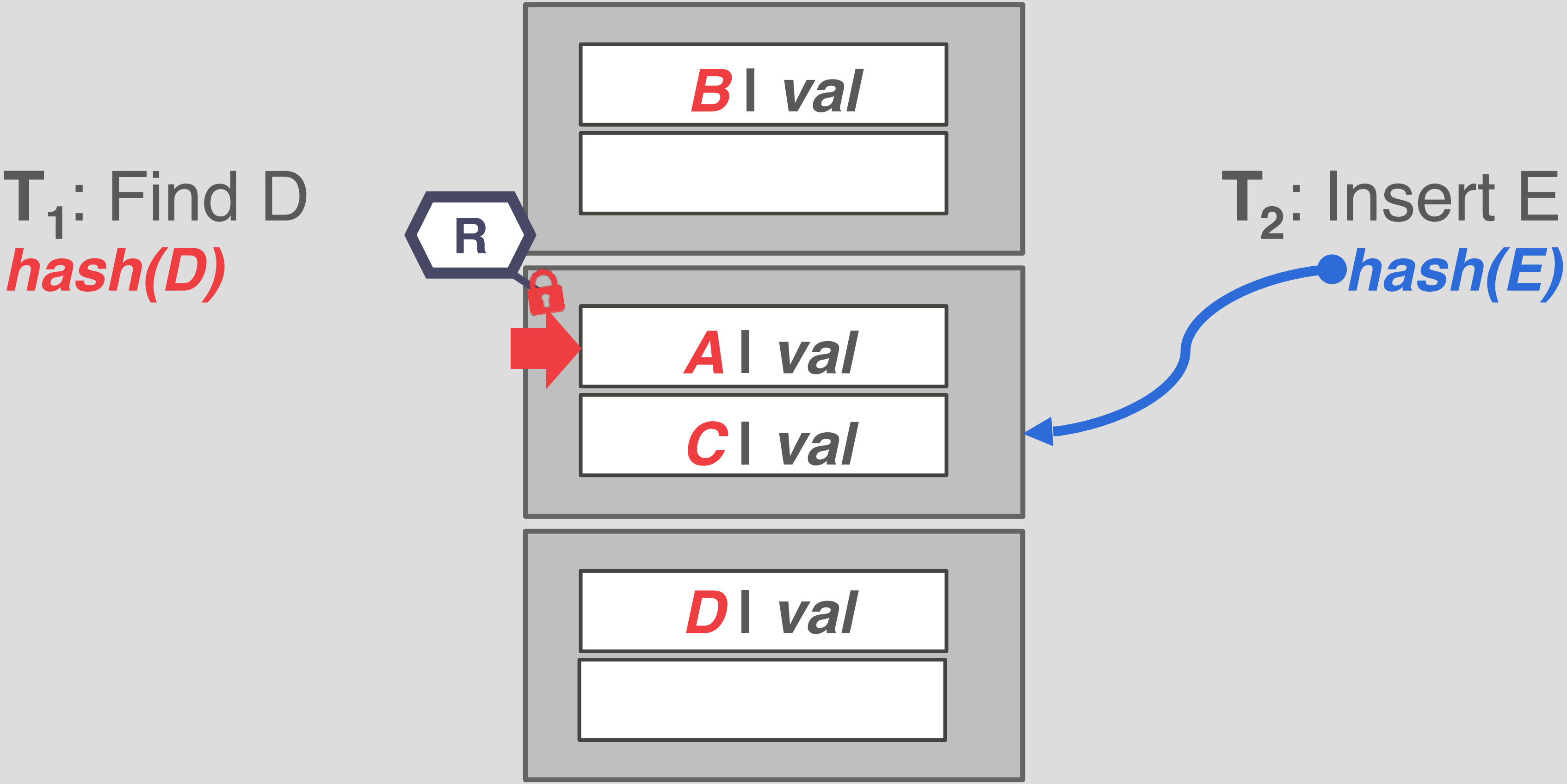
HASH TABLE – PAGE LATCHES

T_1: Find D
hash(D)

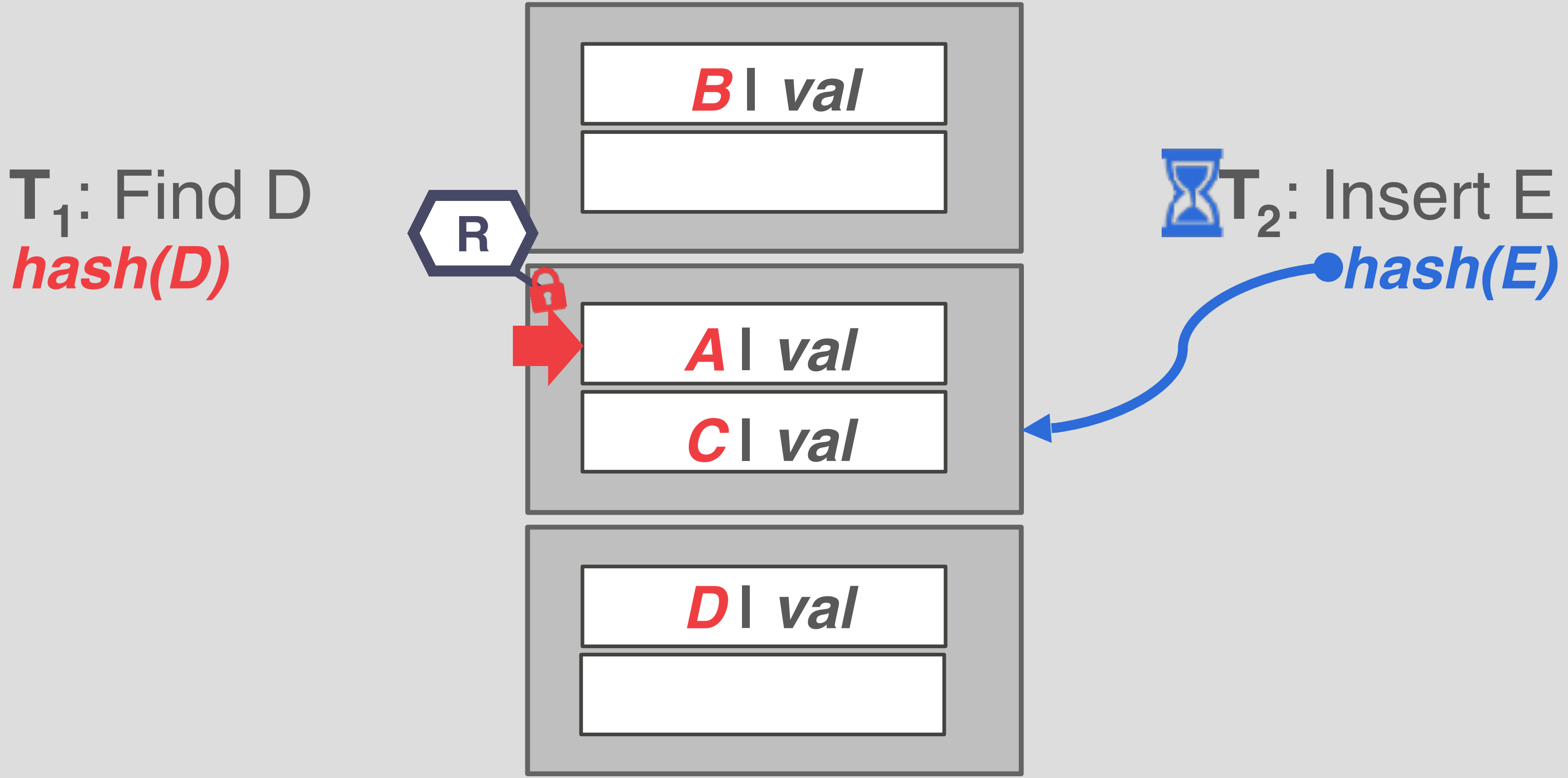


T_2: Insert E
hash(E)

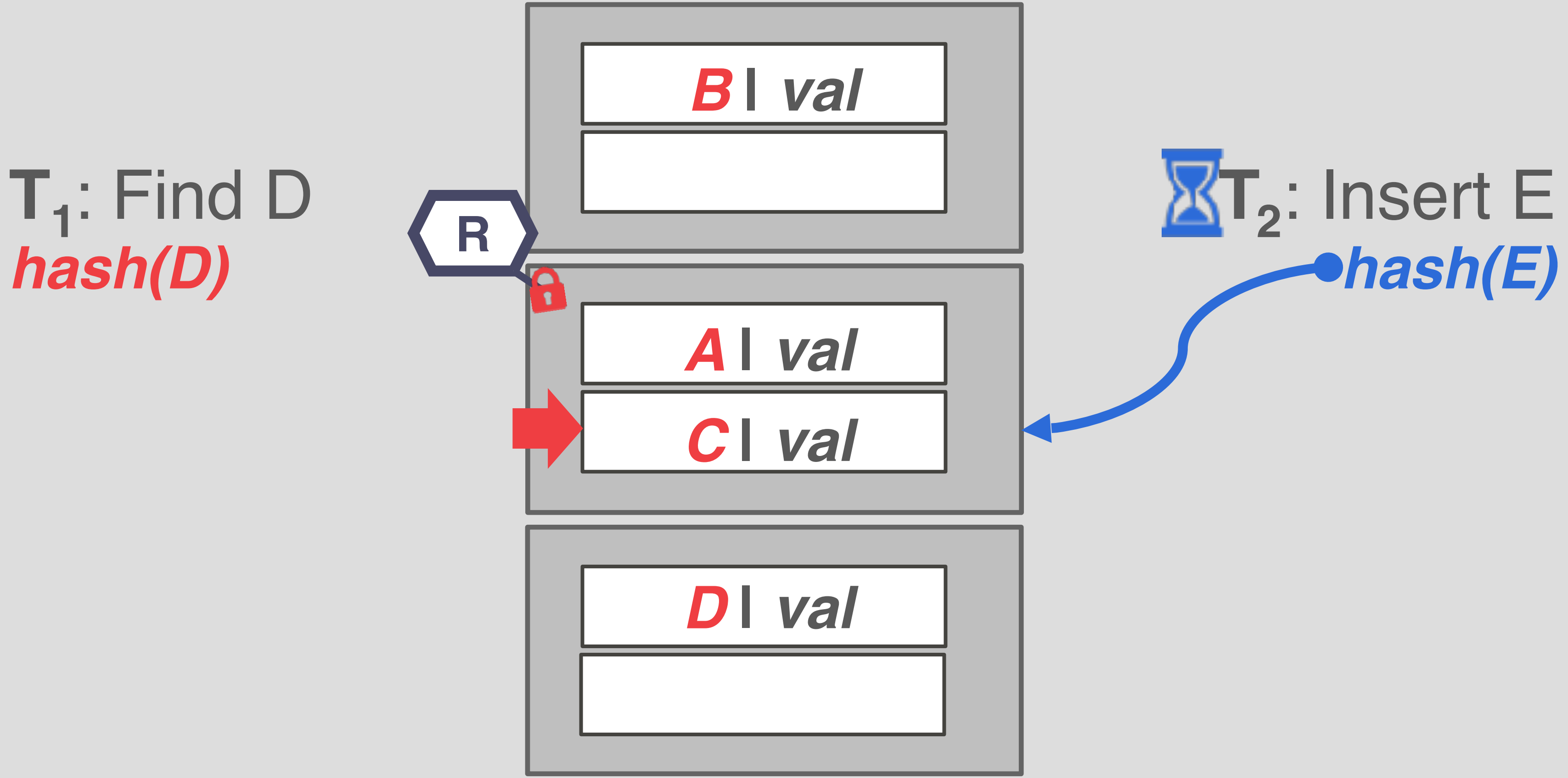
HASH TABLE – PAGE LATCHES



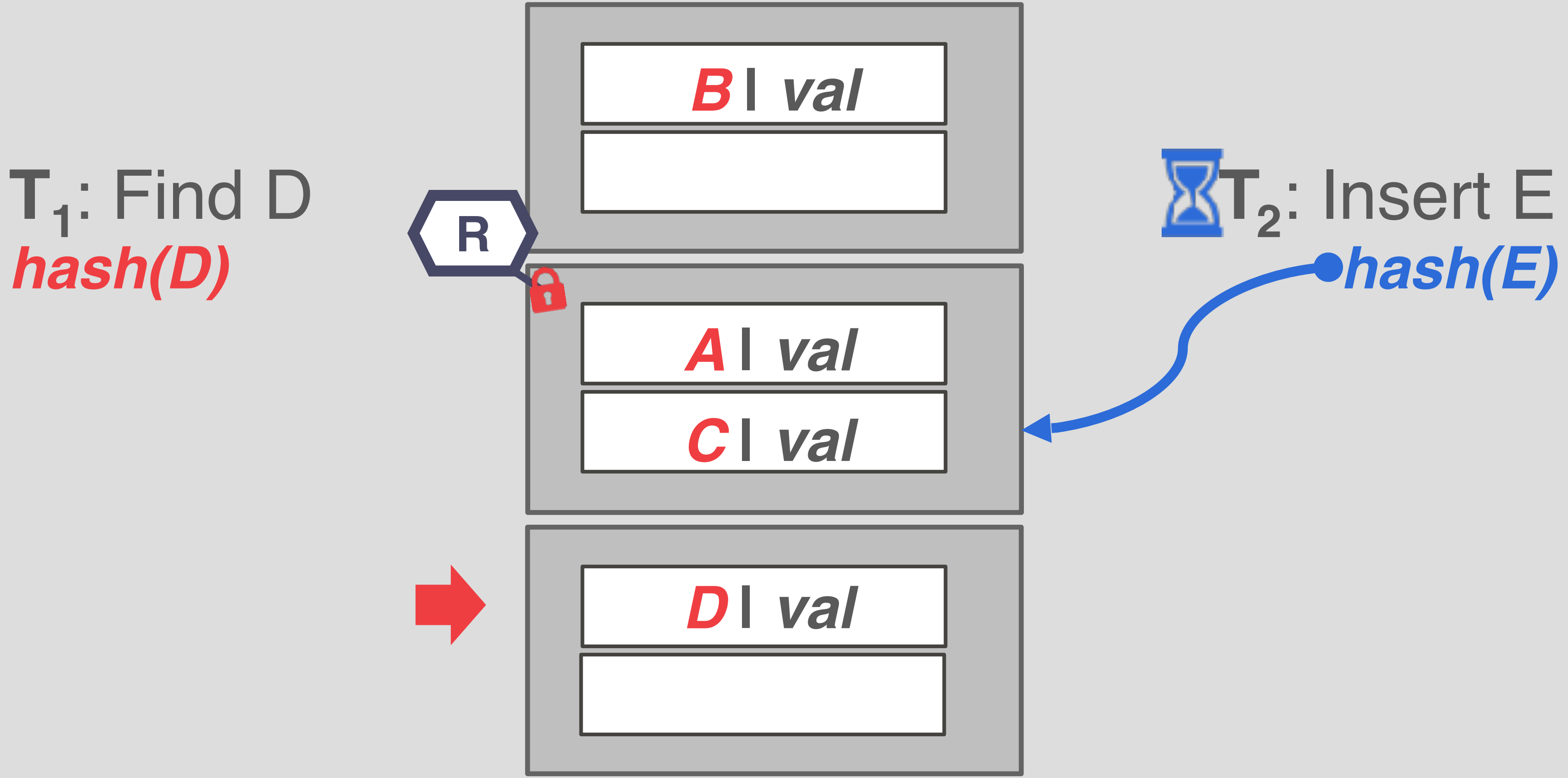
HASH TABLE – PAGE LATCHES



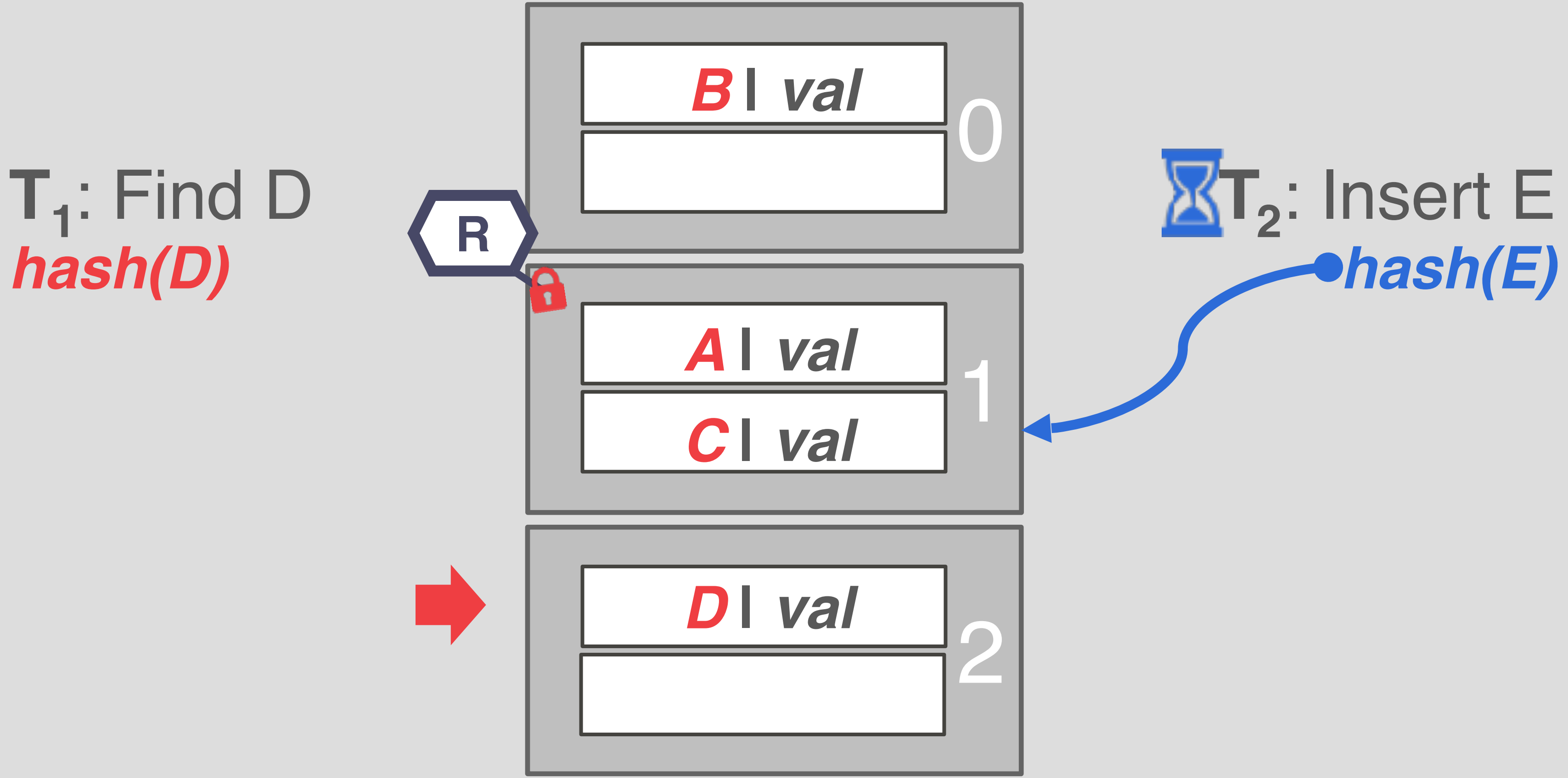
HASH TABLE – PAGE LATCHES



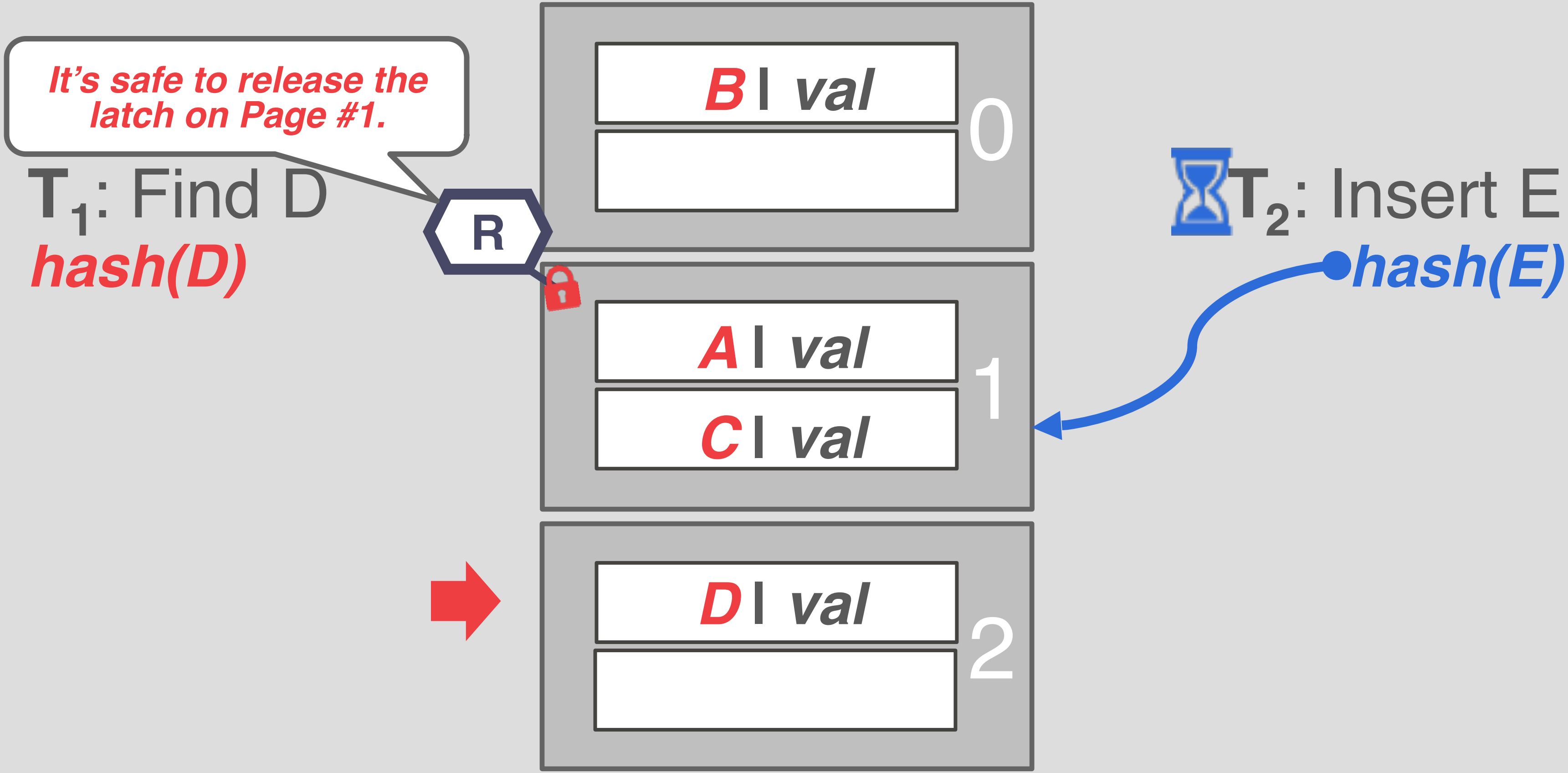
HASH TABLE – PAGE LATCHES



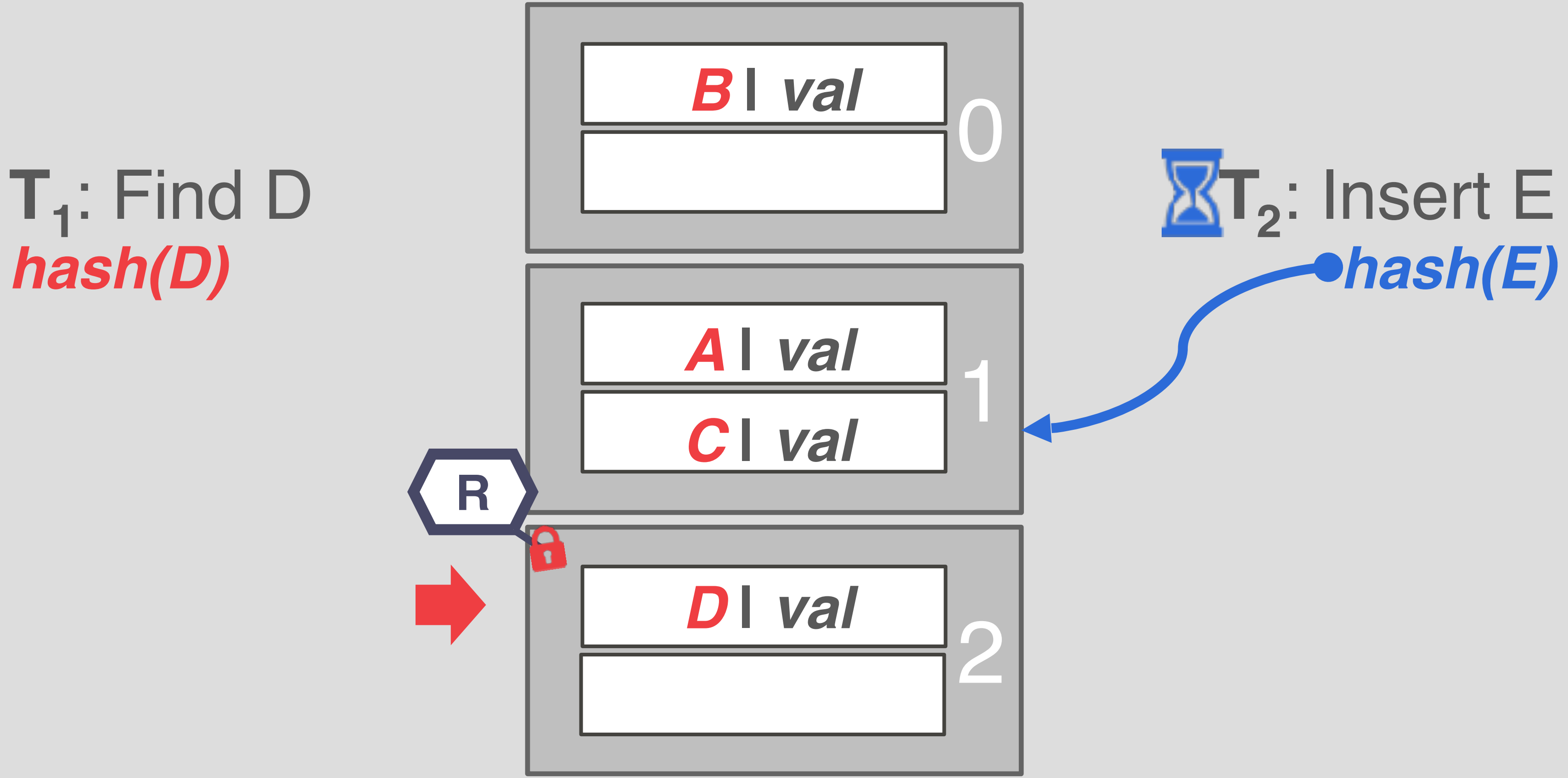
HASH TABLE – PAGE LATCHES



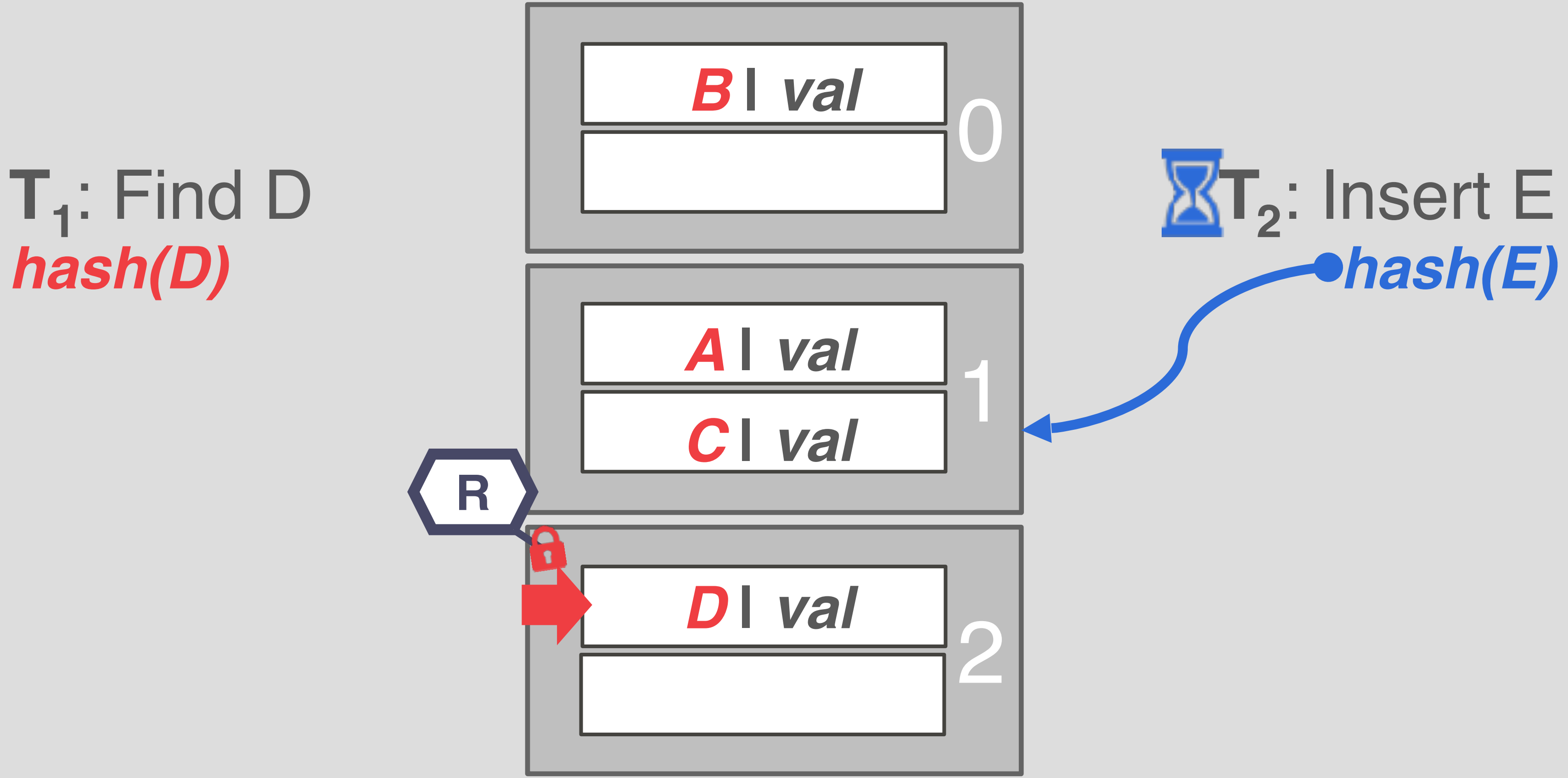
HASH TABLE – PAGE LATCHES



HASH TABLE – PAGE LATCHES

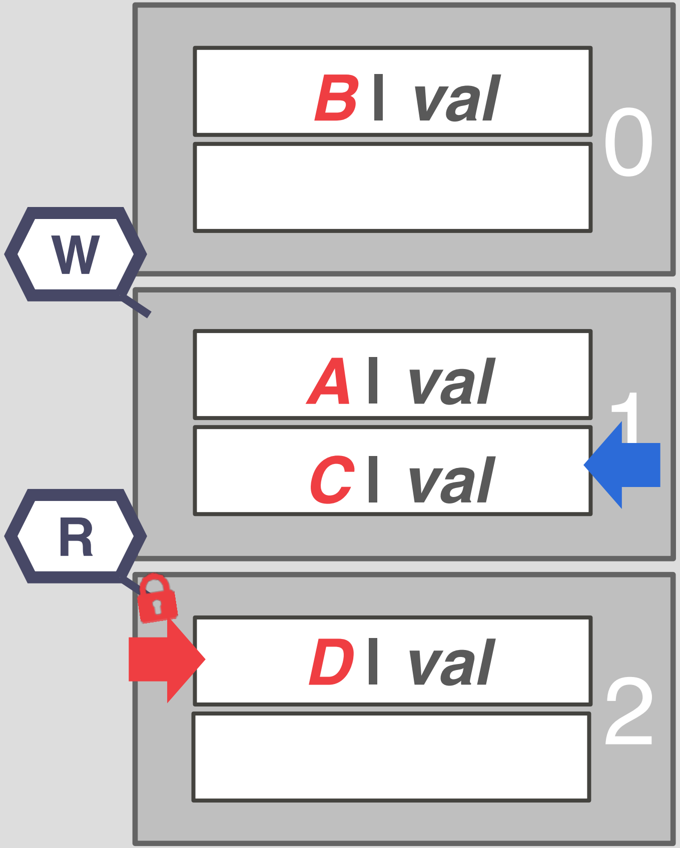


HASH TABLE – PAGE LATCHES



HASH TABLE – PAGE LATCHES

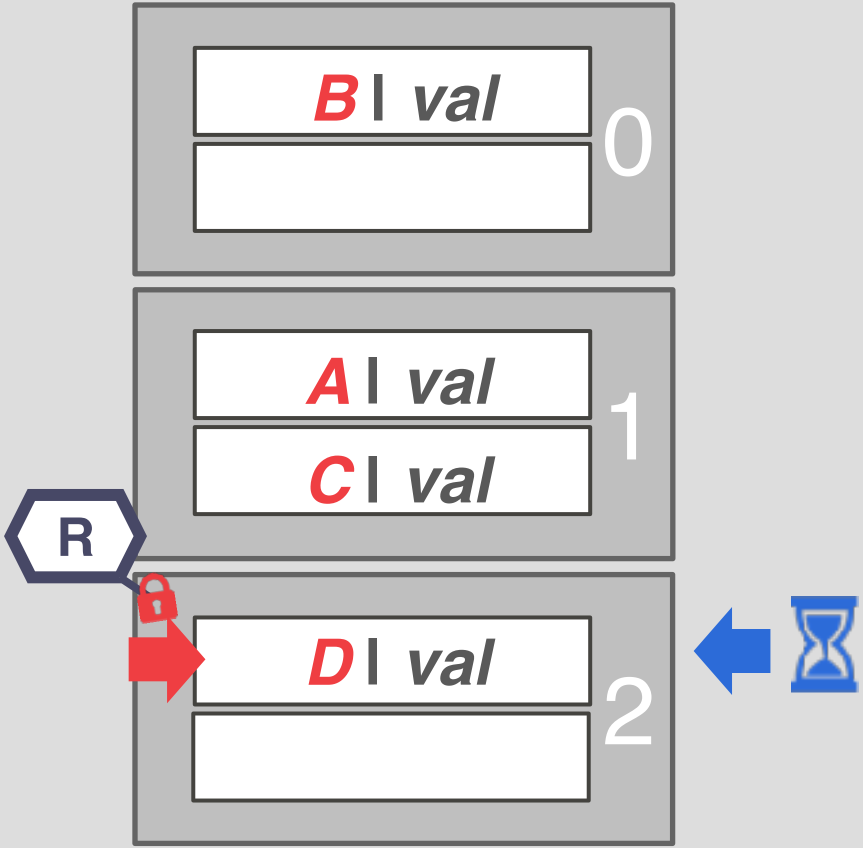
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

HASH TABLE – PAGE LATCHES

T_1: Find D
hash(D)

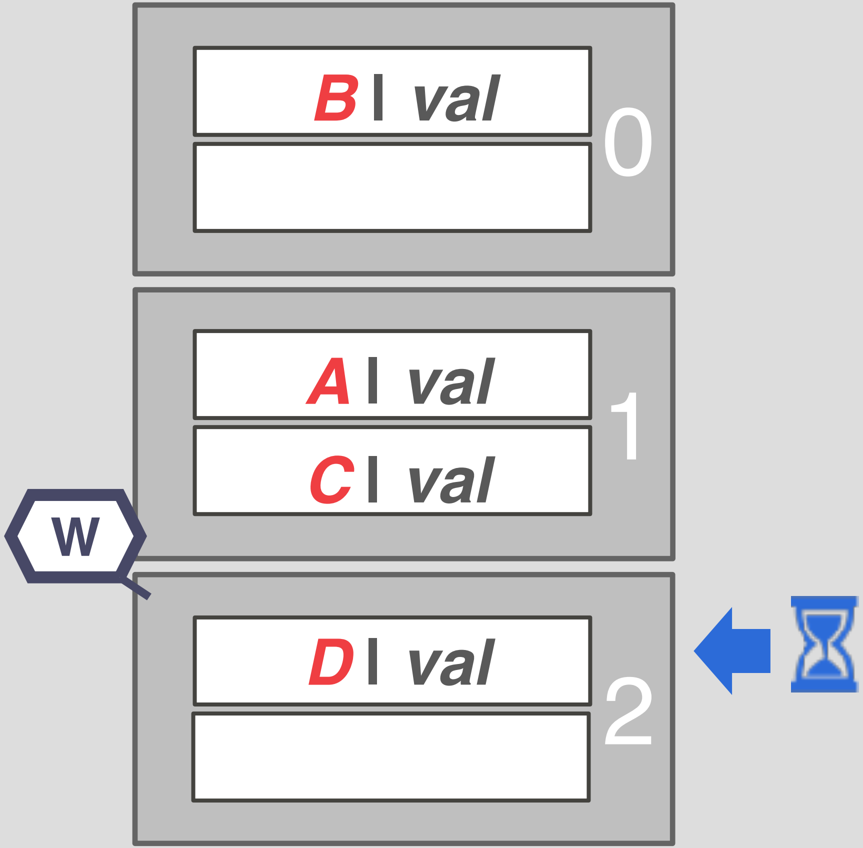


T_2: Insert E
hash(E)

HASH TABLE – PAGE LATCHES

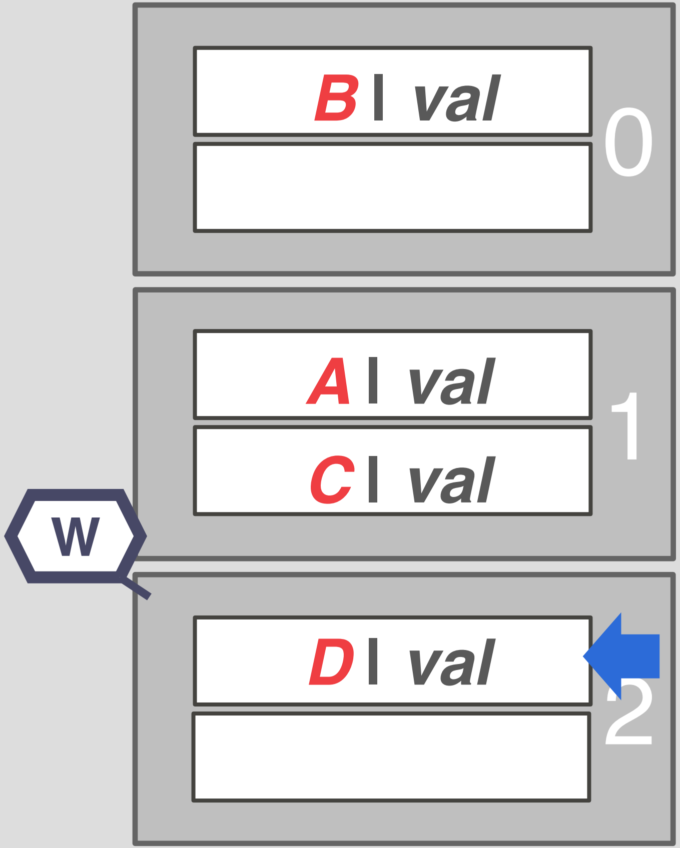
T_1: Find D
hash(D)

T_2: Insert E
hash(E)



HASH TABLE – PAGE LATCHES

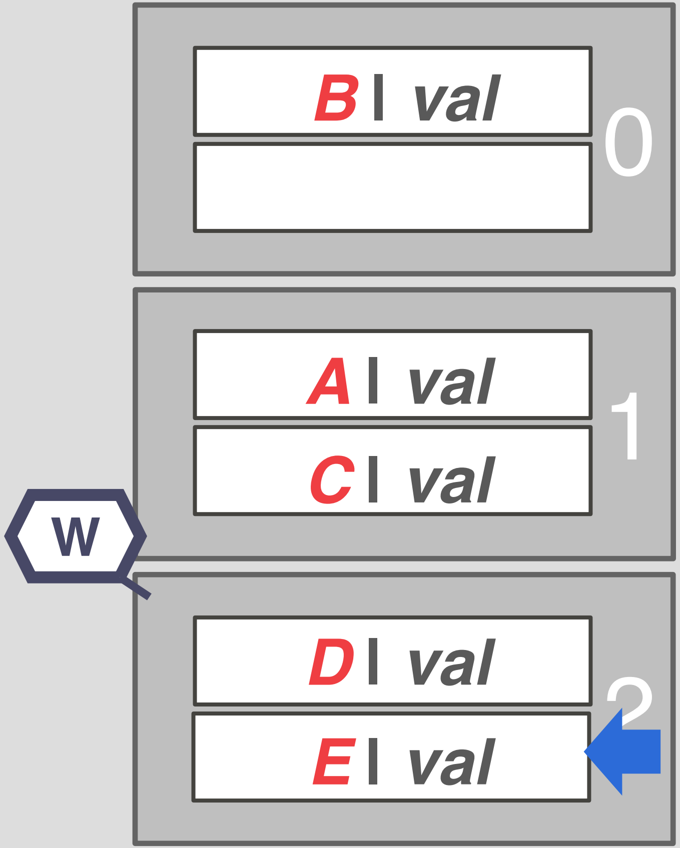
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

HASH TABLE – PAGE LATCHES

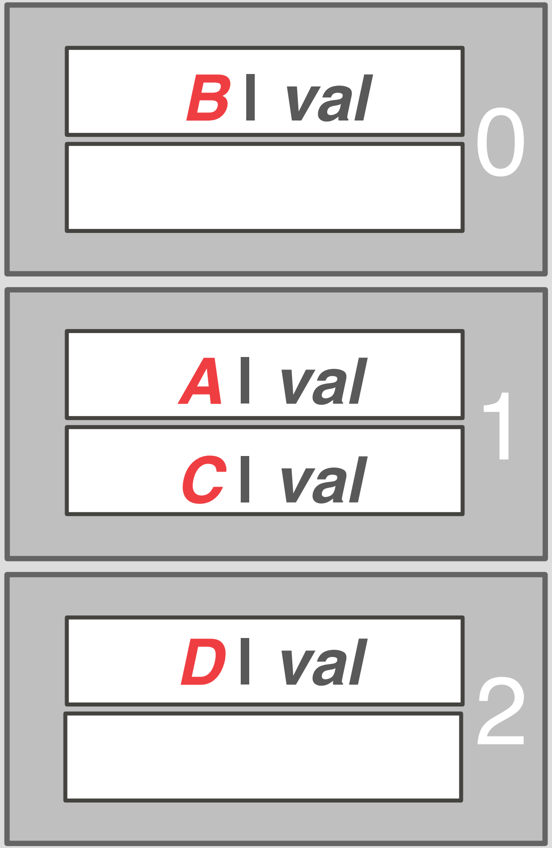
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

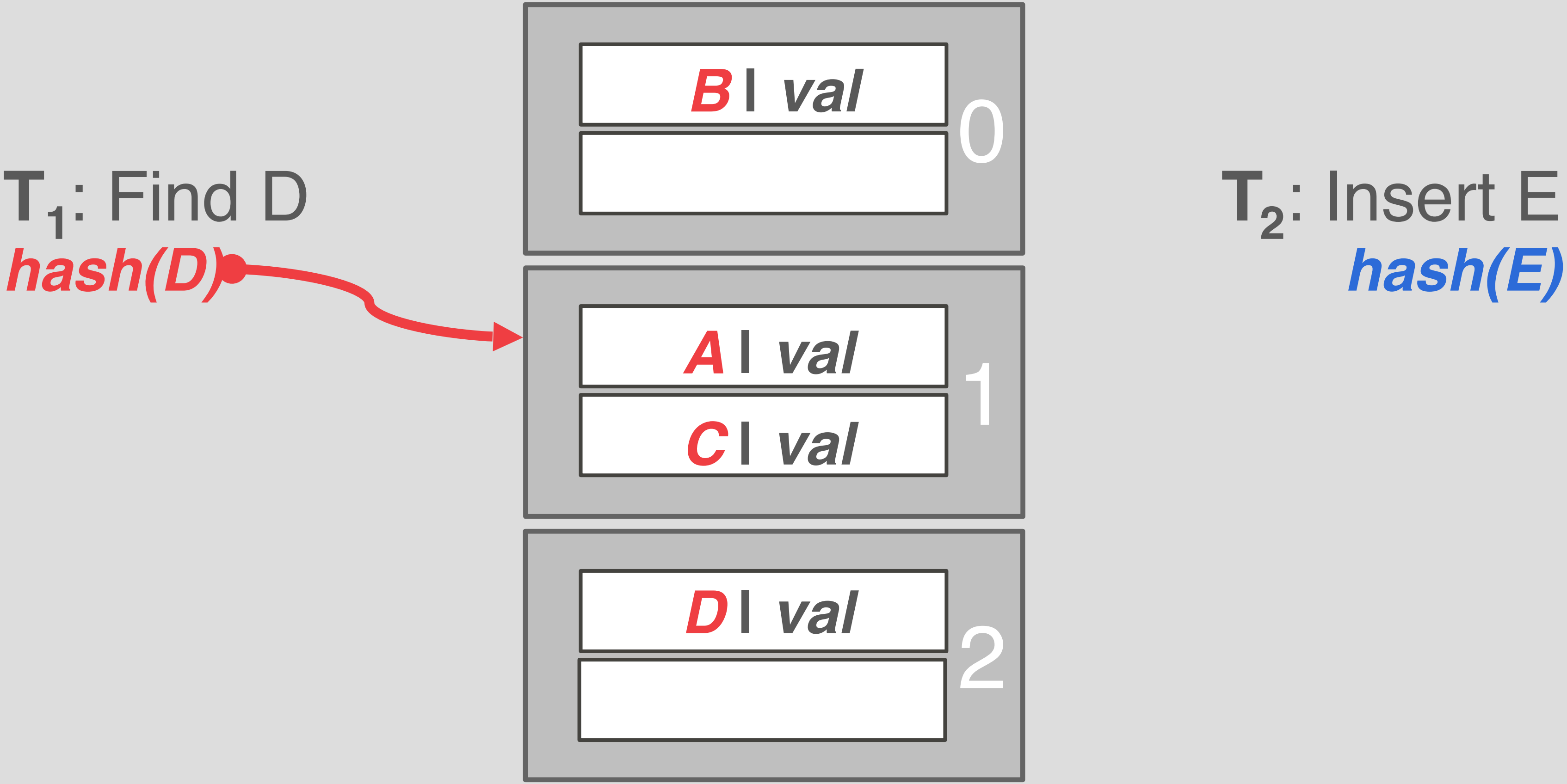
HASH TABLE – SLOT LATCHES

T_1: Find D
hash(D)



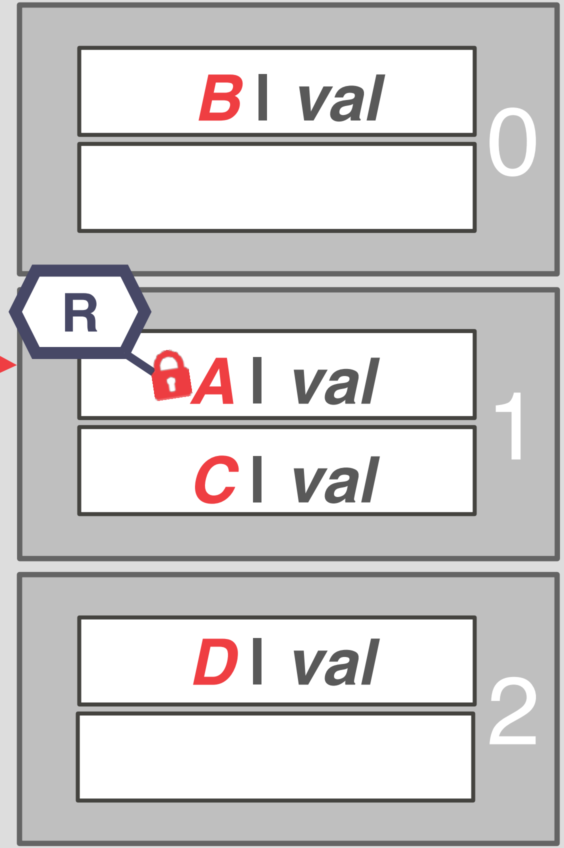
T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES



HASH TABLE – SLOT LATCHES

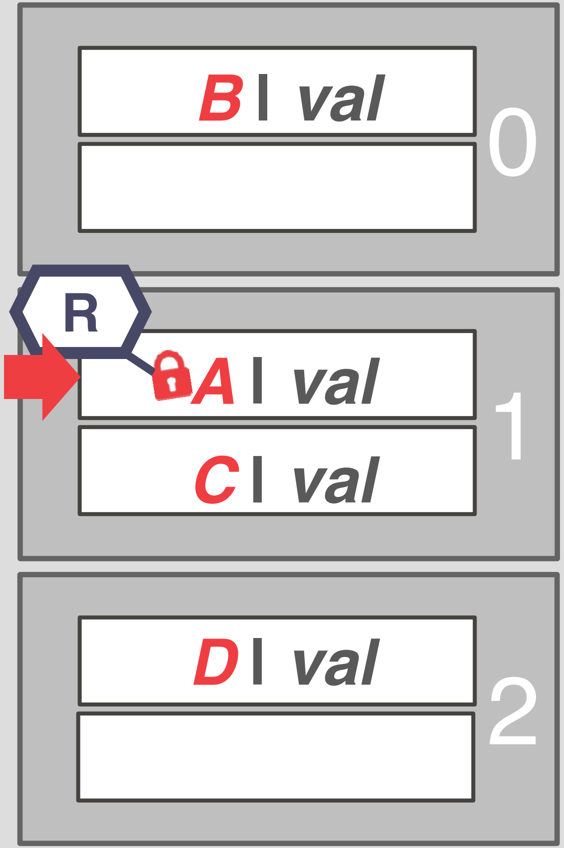
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

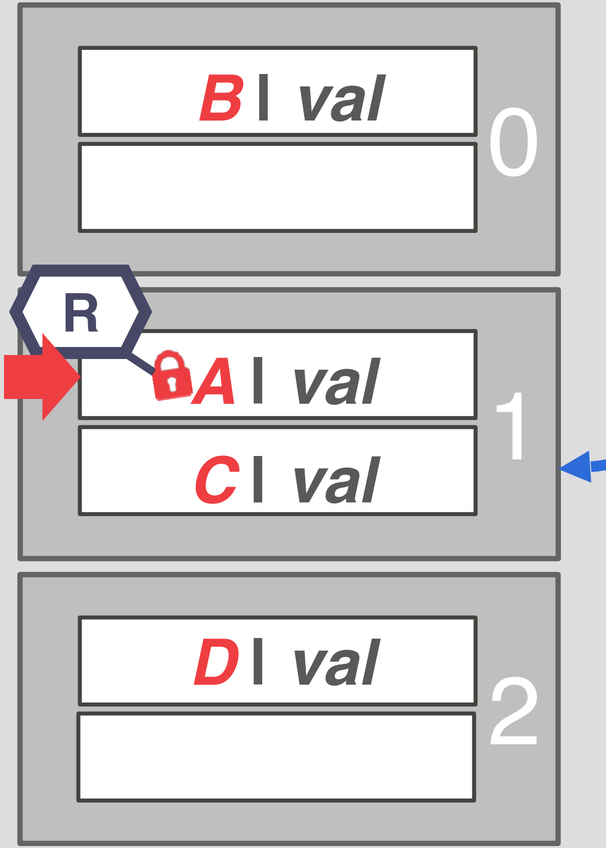
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

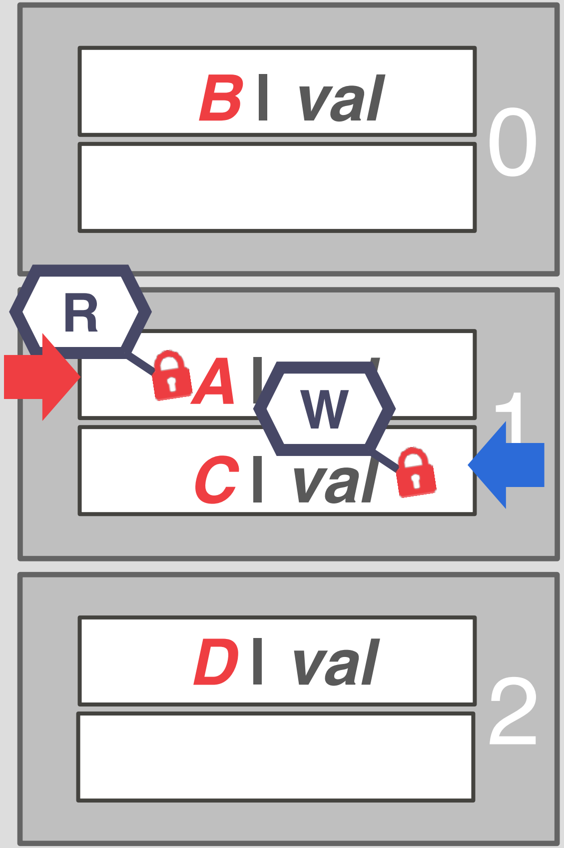
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

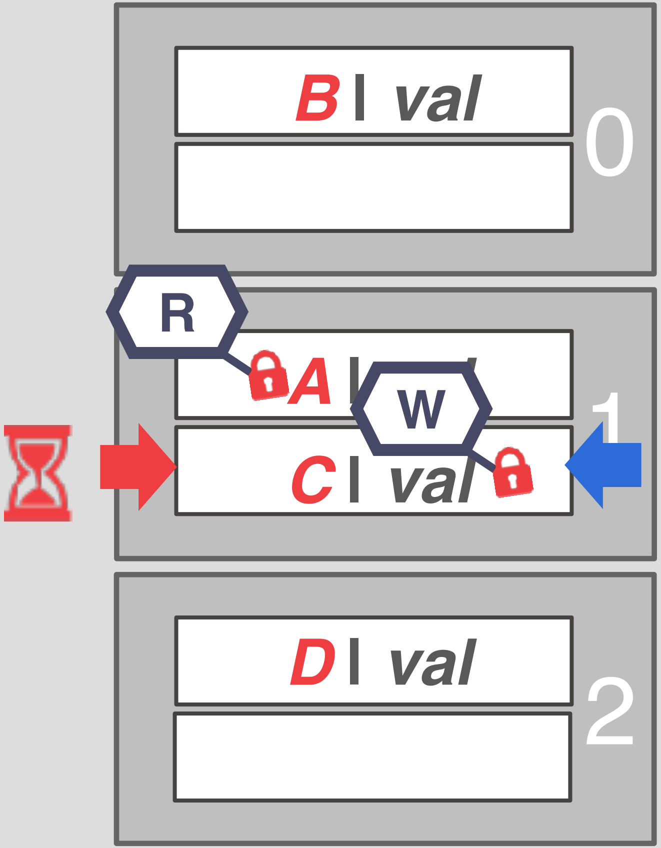
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

T_1: Find D
hash(D)



T_2: Insert E
hash(E)

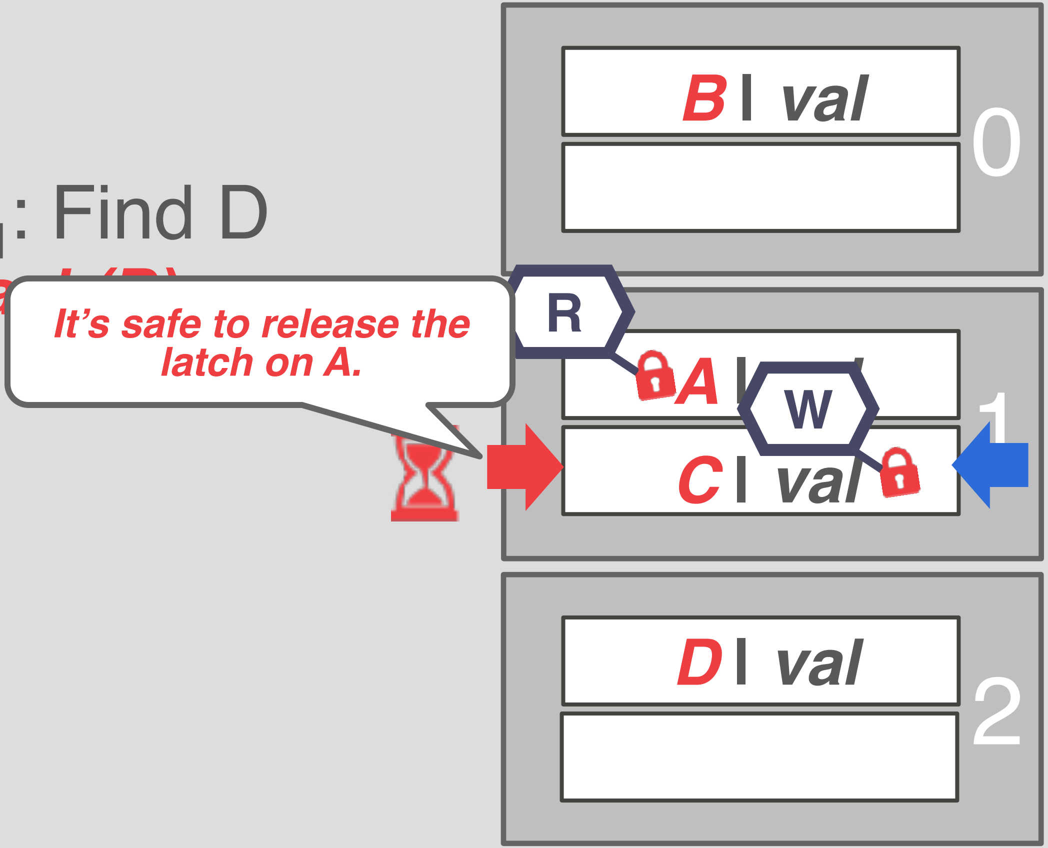
HASH TABLE – SLOT LATCHES

T_1: Find D

ha

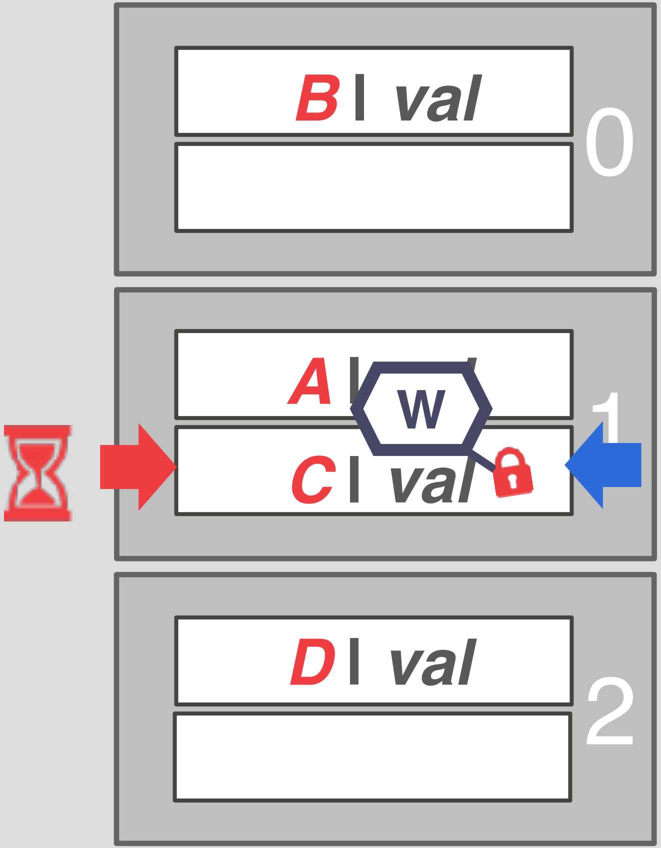
T_2: Insert E

hash(E)



HASH TABLE – SLOT LATCHES

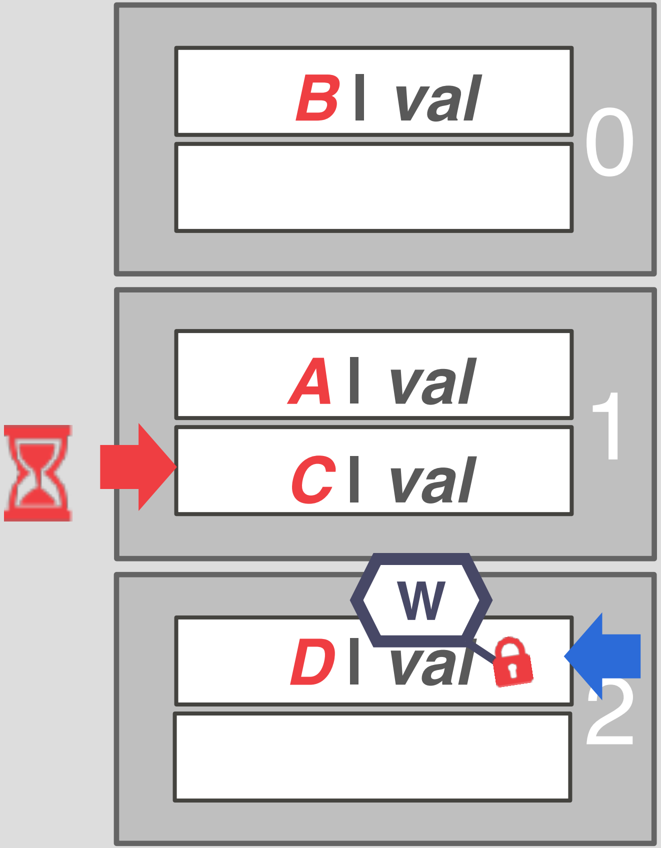
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

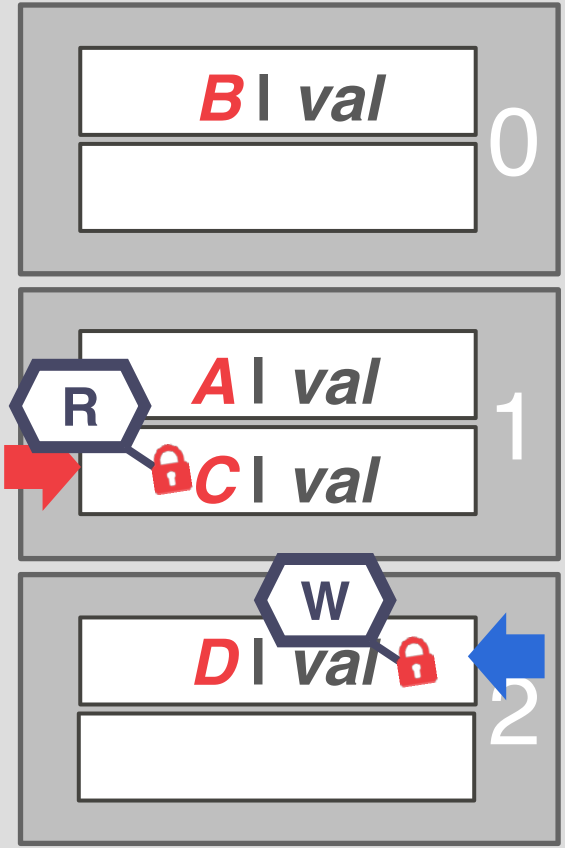
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

T_1: Find D
hash(D)

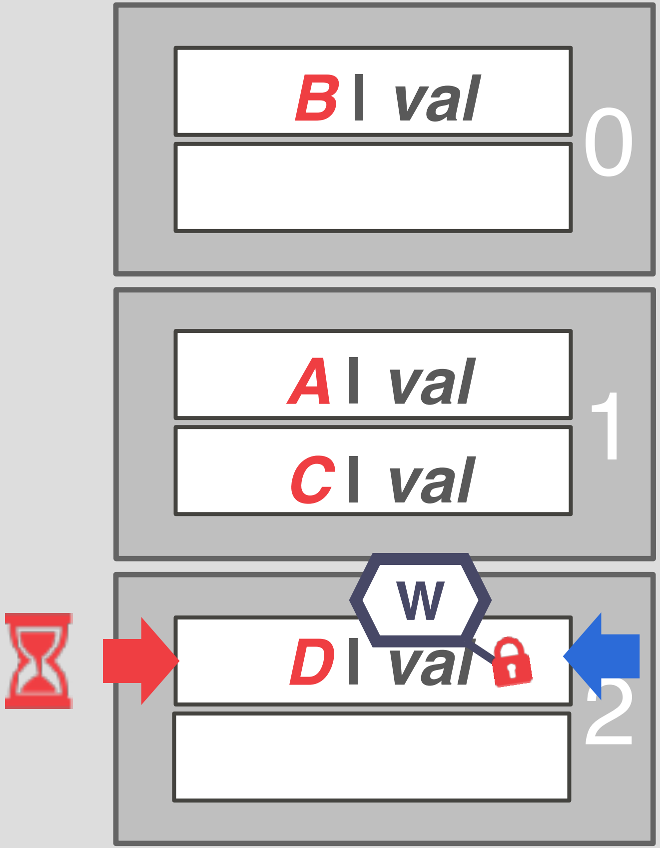


T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

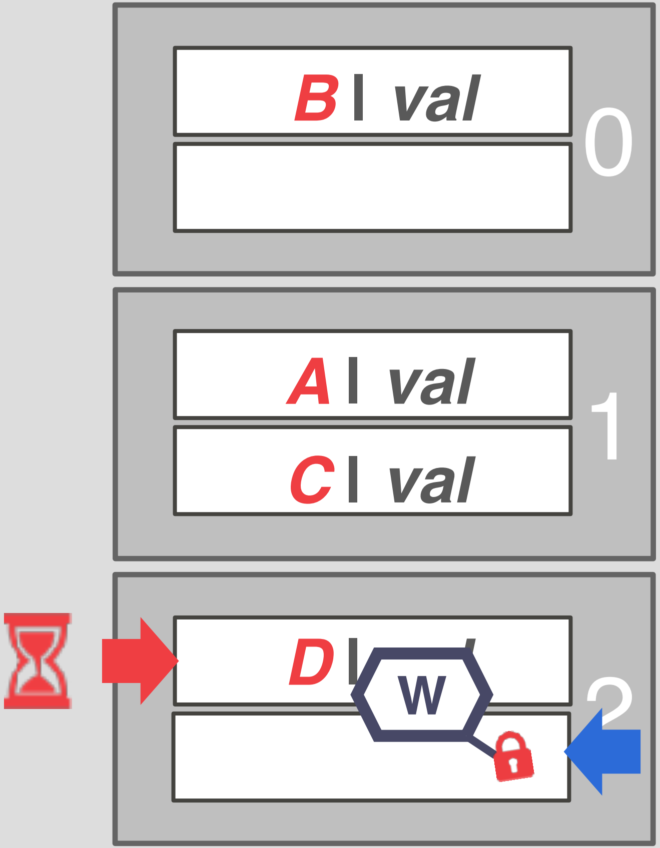
T_1: Find D
hash(D)

T_2: Insert E
hash(E)



HASH TABLE – SLOT LATCHES

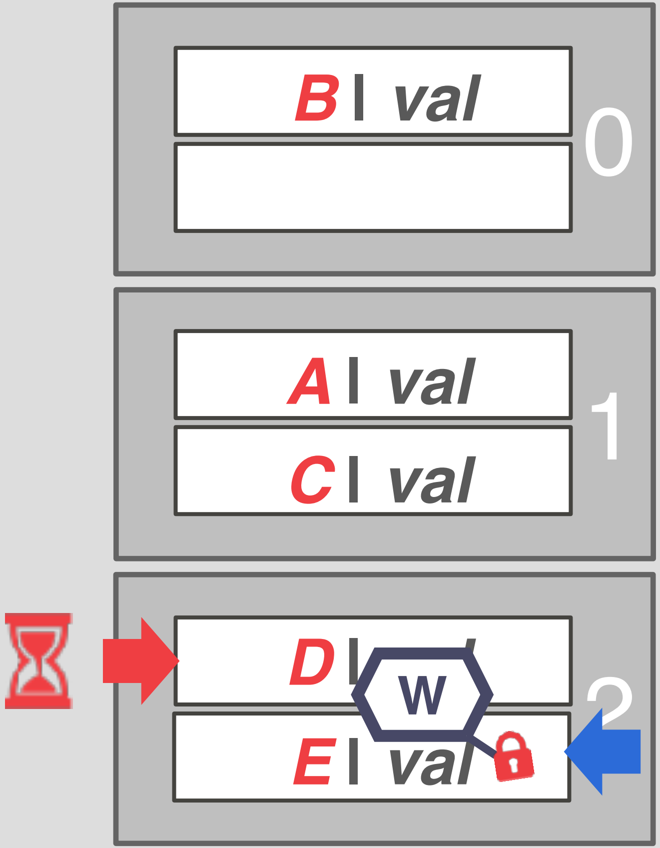
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

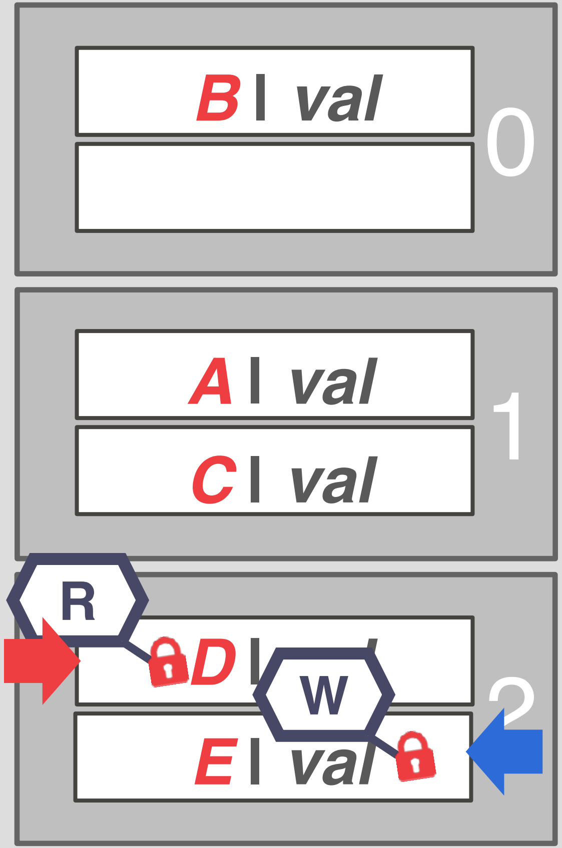
T_1: Find D
hash(D)



T_2: Insert E
hash(E)

HASH TABLE – SLOT LATCHES

T_1: Find D
hash(D)



T_2: Insert E
hash(E)

B+TREE CONCURRENCY CONTROL

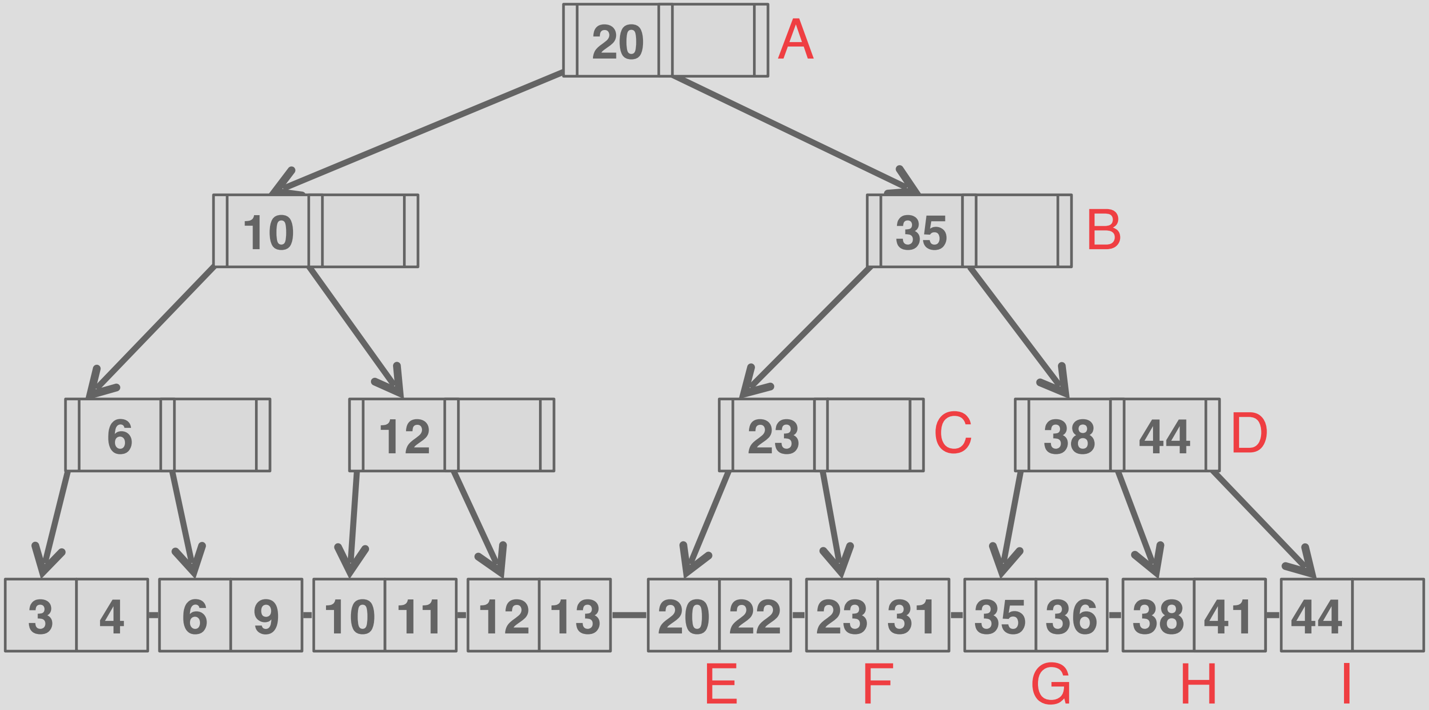
We want to allow multiple threads to read and update a B+Tree at the same time.

We need to protect against two types of problems:

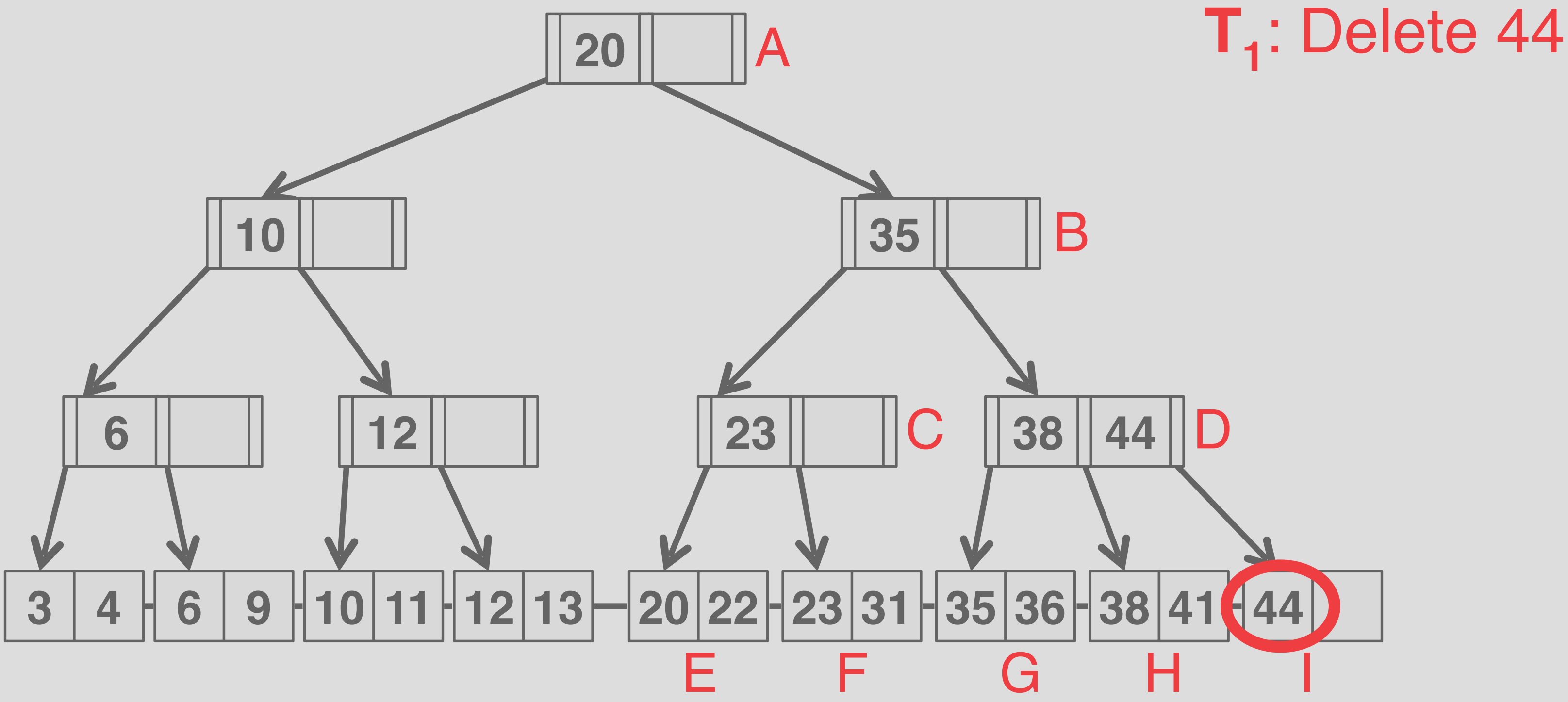
→ Threads trying to modify the contents of a node at the same time.

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

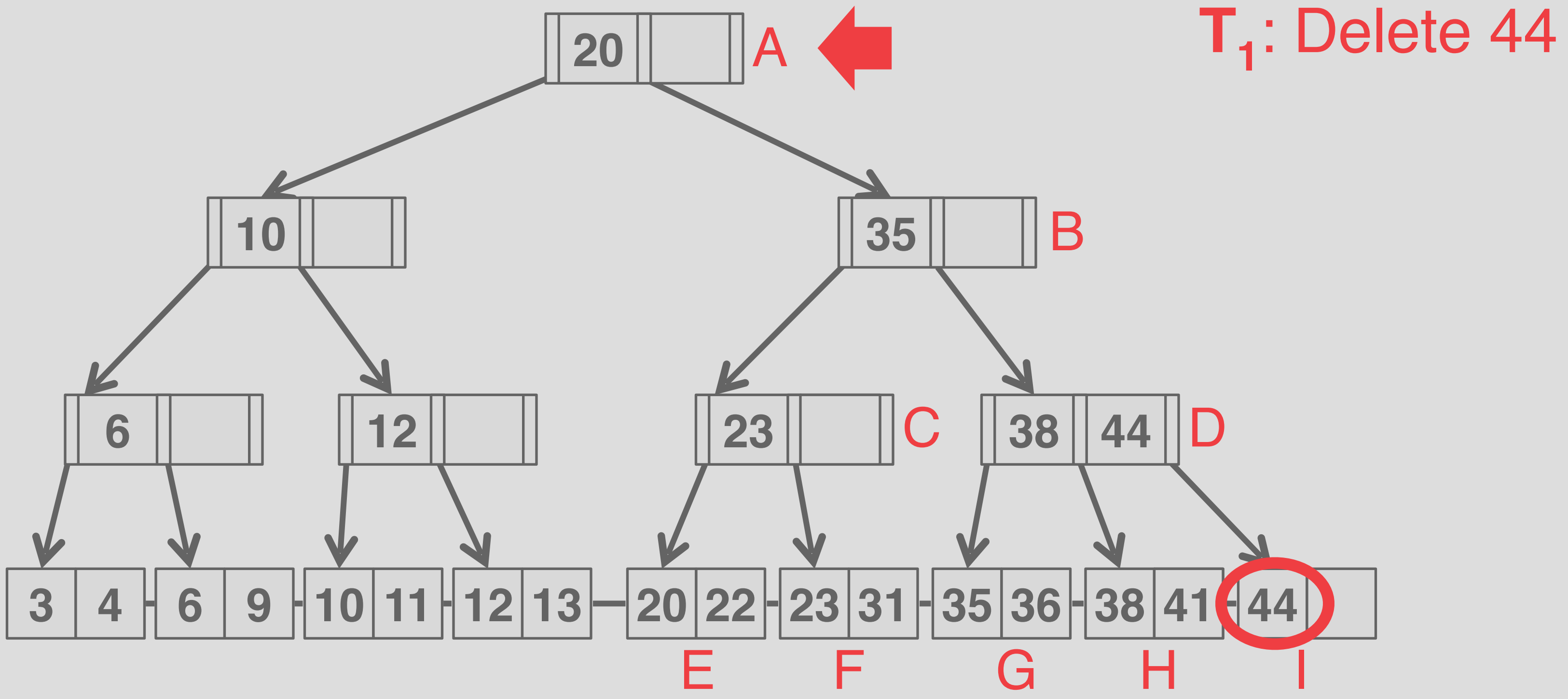
B+TREE MULTI-THREADED EXAMPLE



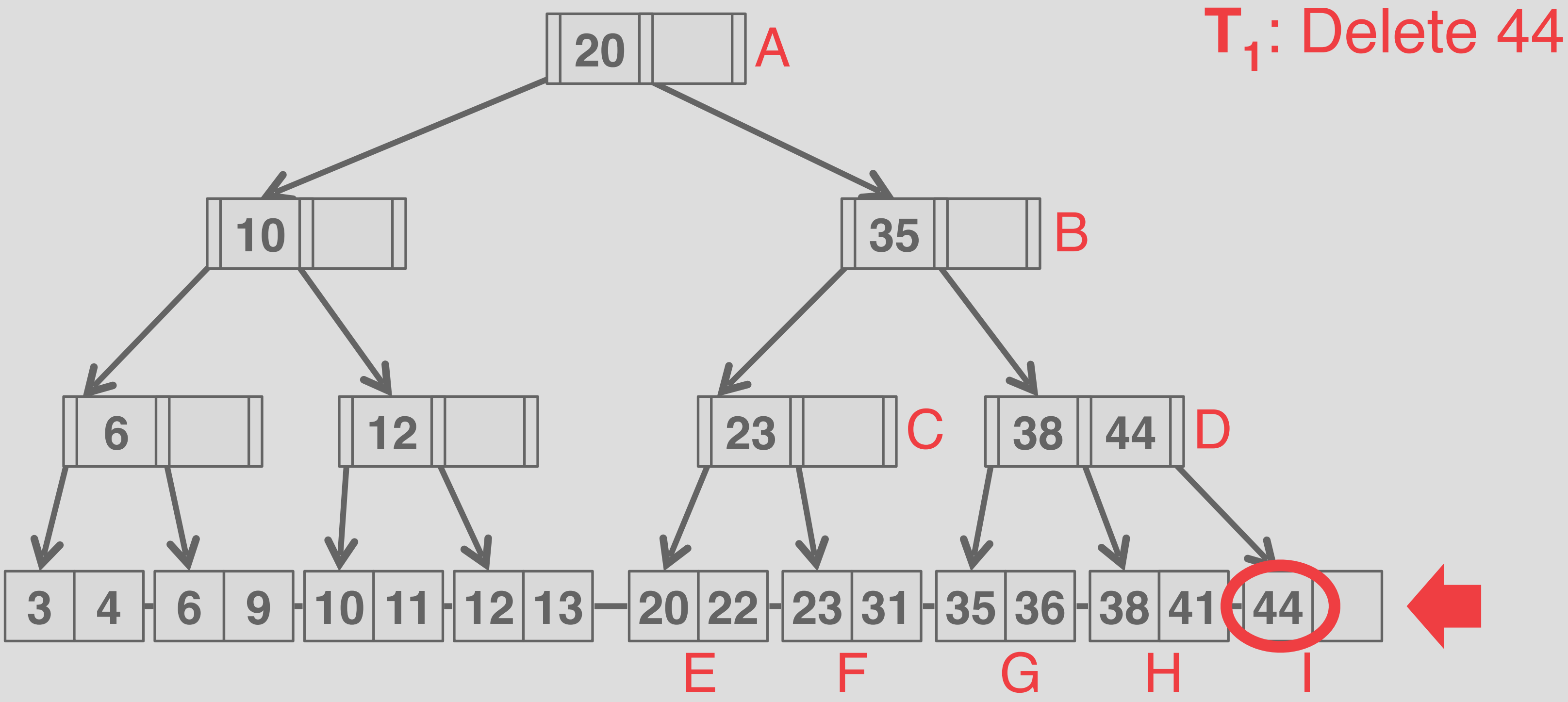
B+TREE MULTI-THREADED EXAMPLE



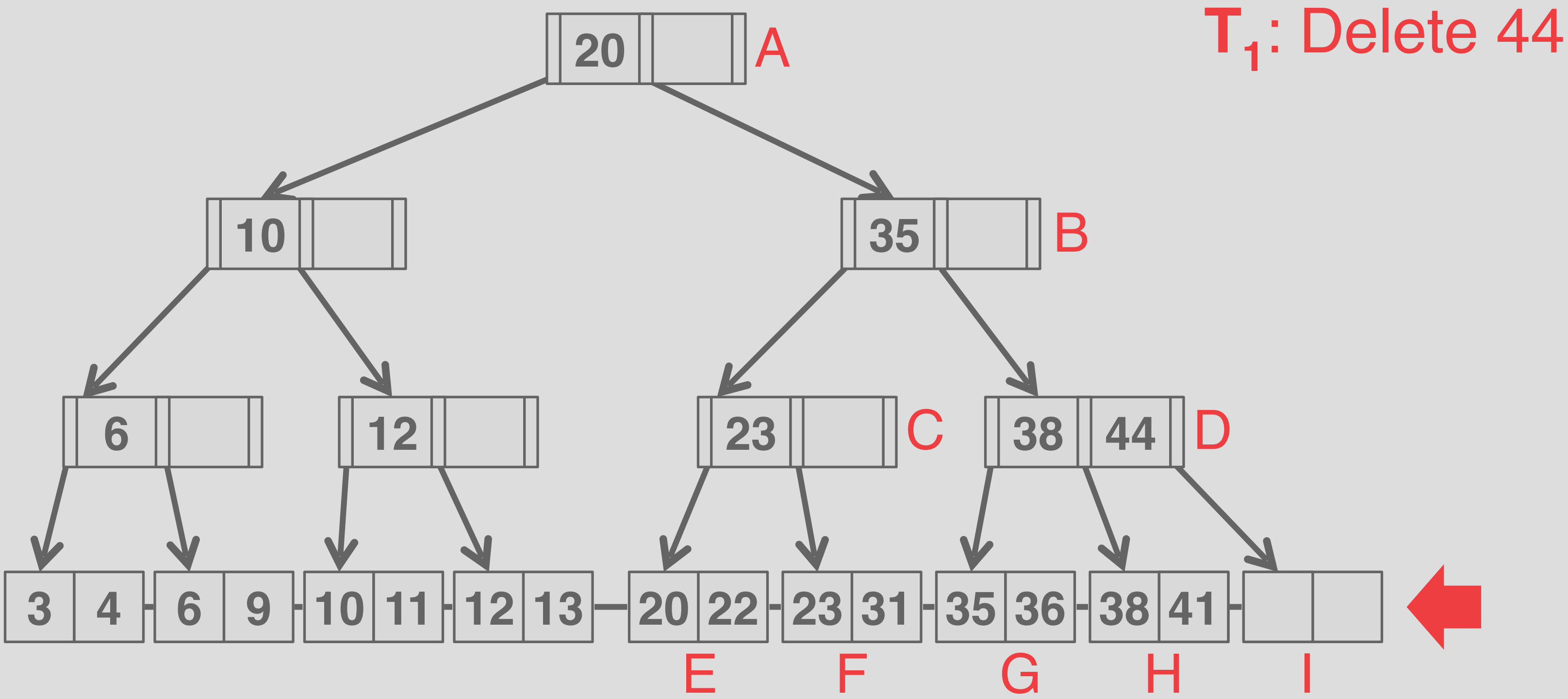
B+TREE MULTI-THREADED EXAMPLE



B+TREE MULTI-THREADED EXAMPLE

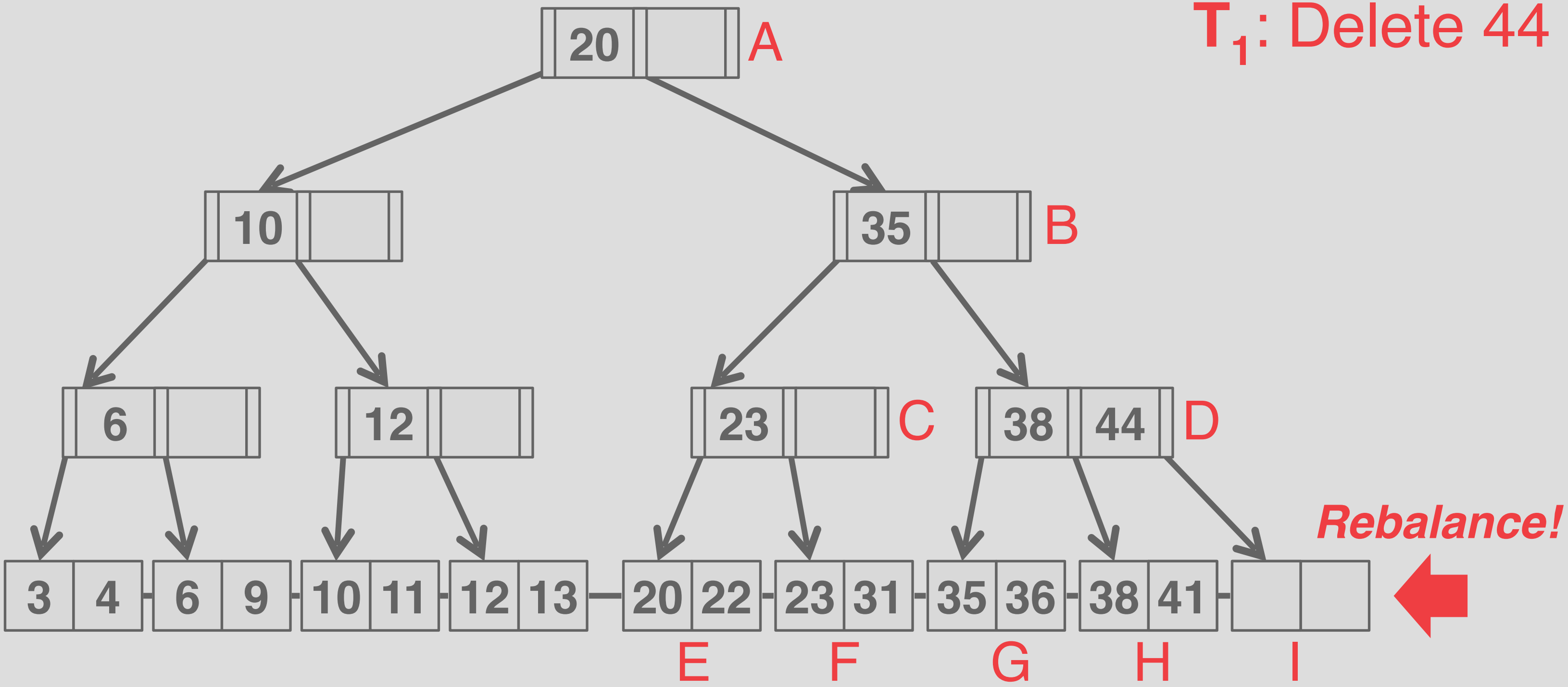


B+TREE MULTI-THREADED EXAMPLE



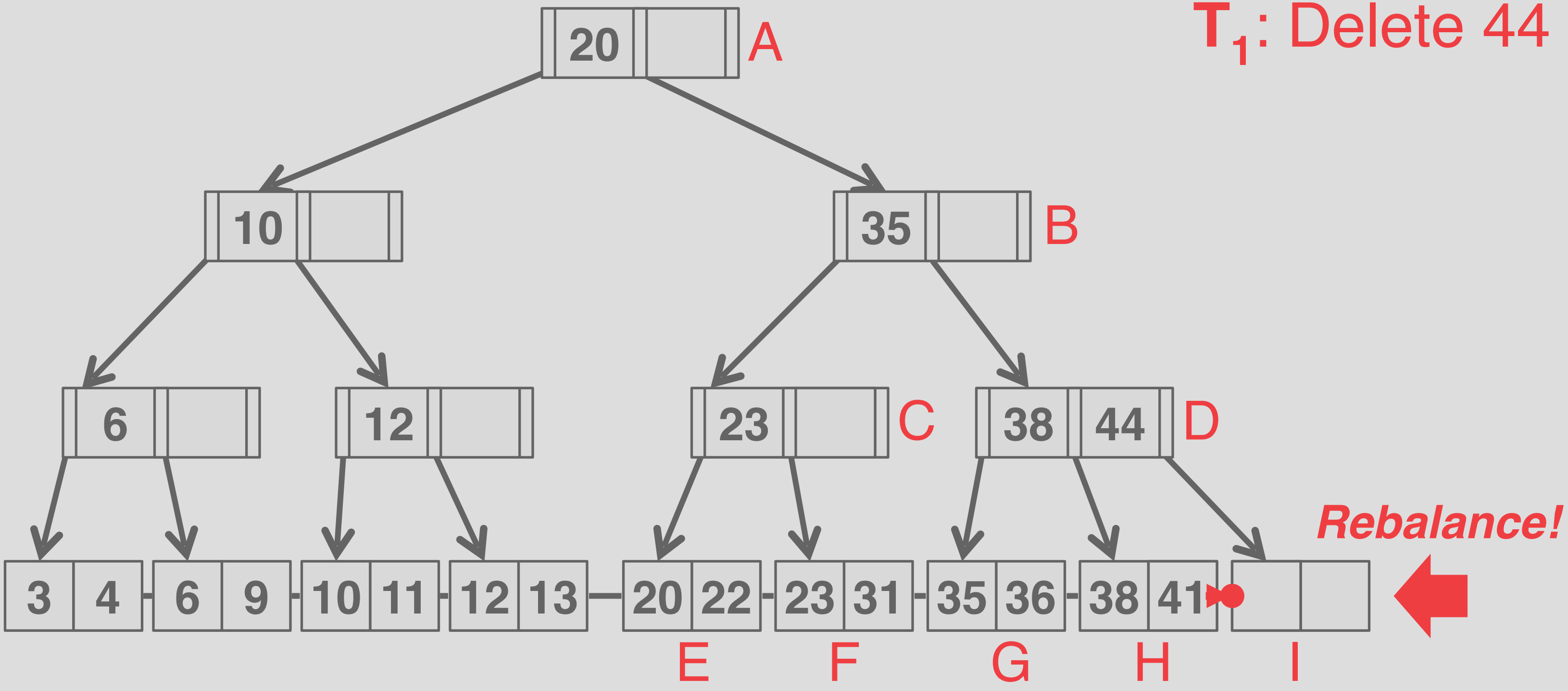
B+TREE MULTI-THREADED EXAMPLE

T$_1$: Delete 44

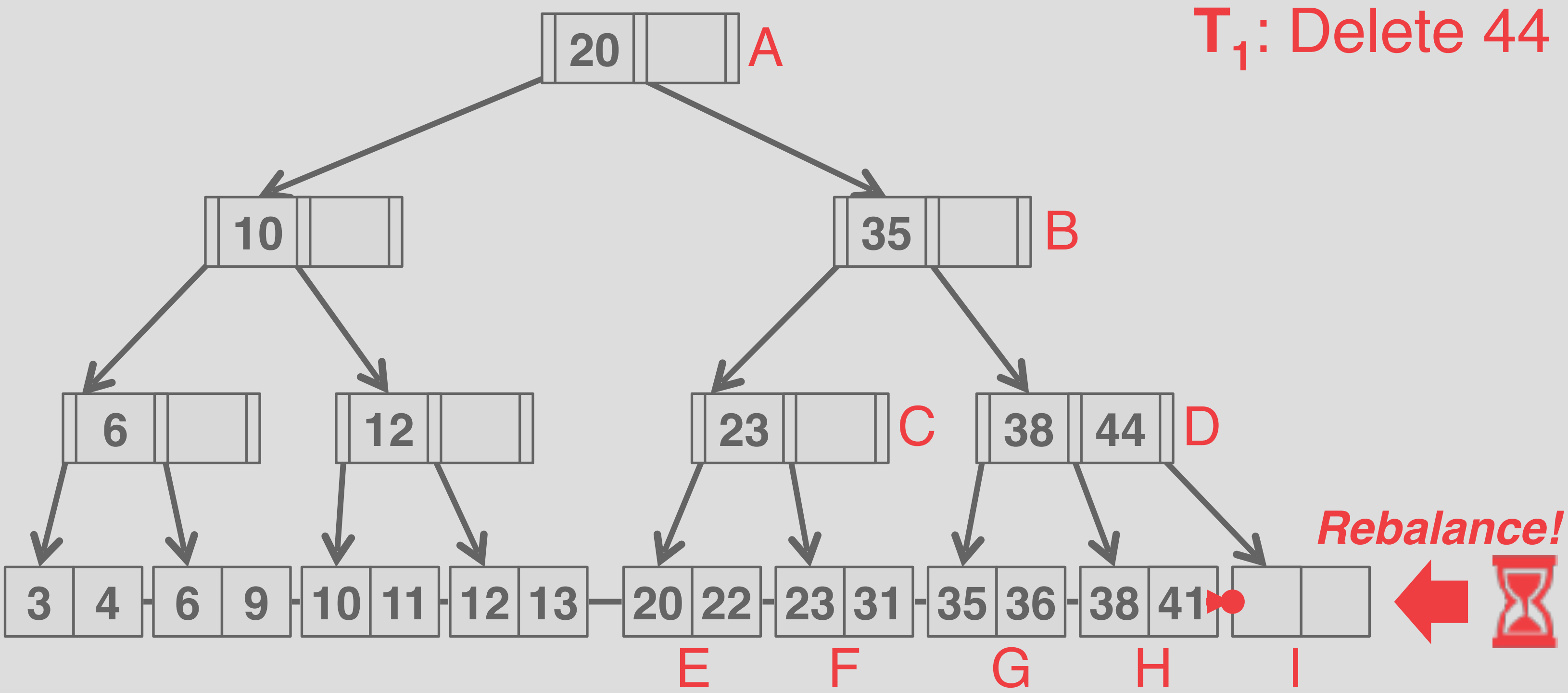


B+TREE MULTI-THREADED EXAMPLE

T_1: Delete 44



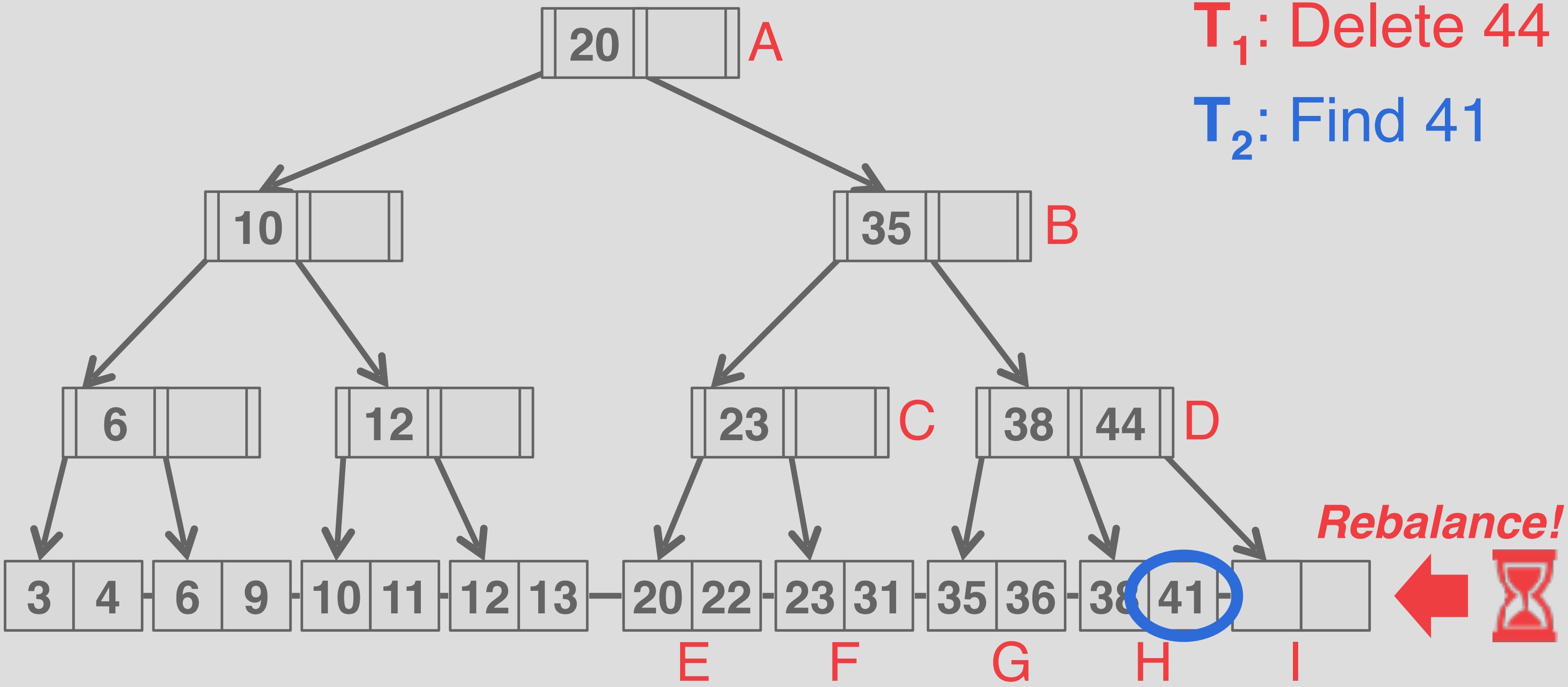
B+TREE MULTI-THREADED EXAMPLE



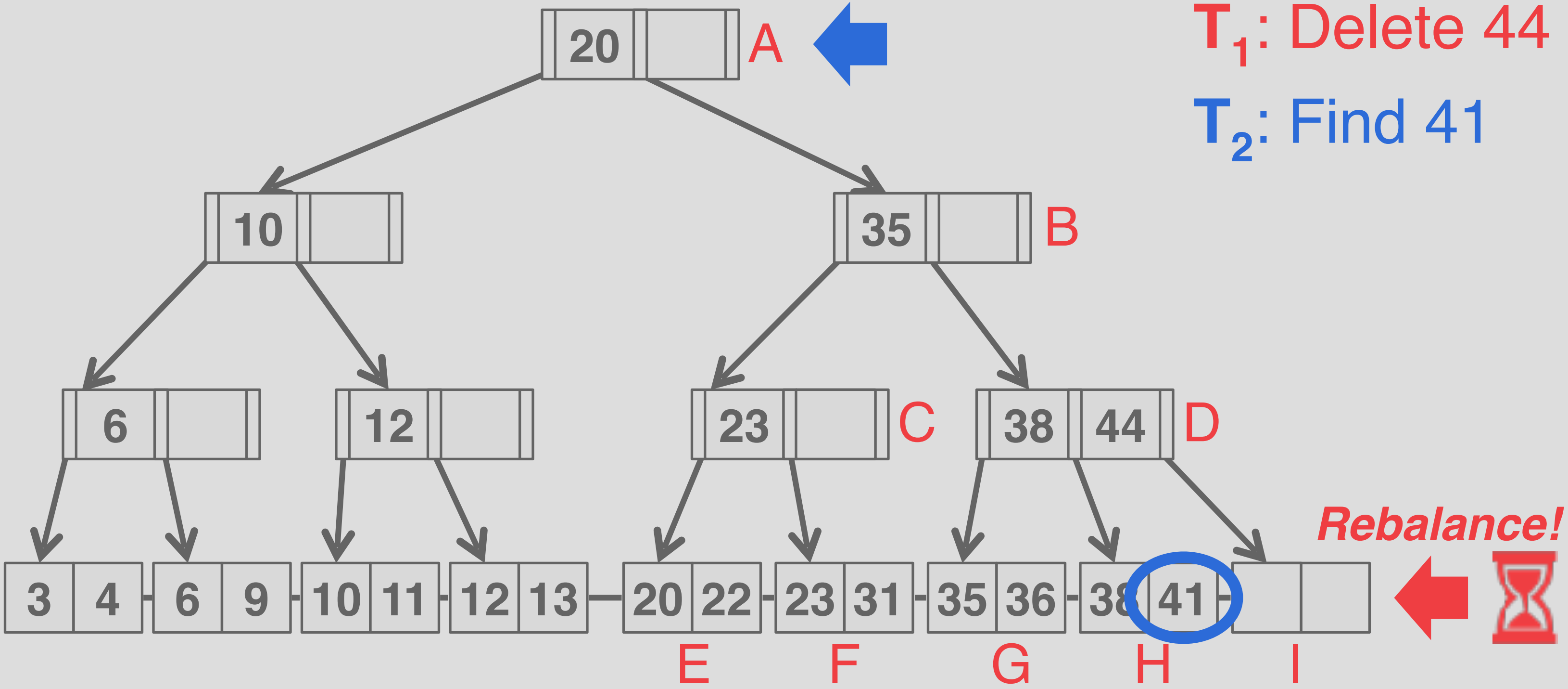
B+TREE MULTI-THREADED EXAMPLE

T_1: Delete 44

T_2: Find 41



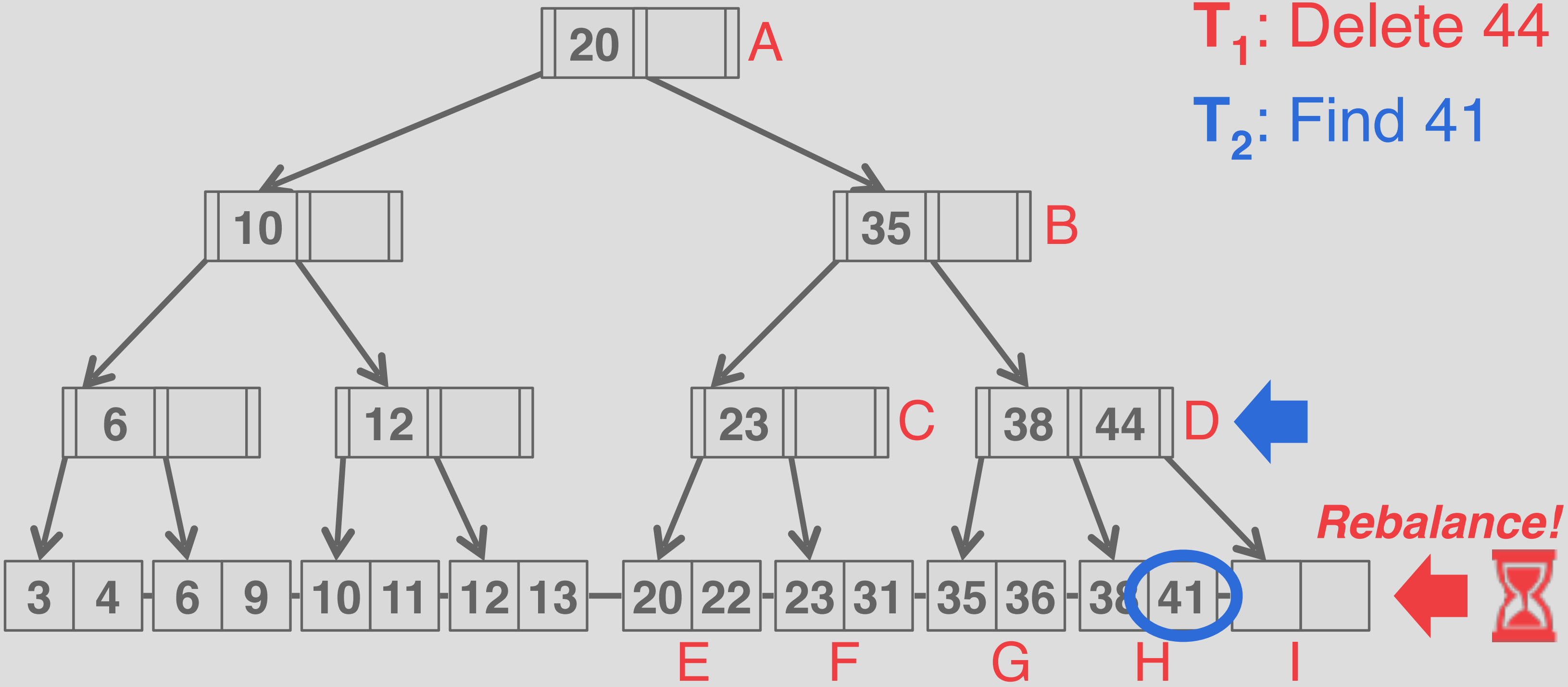
B+TREE MULTI-THREADED EXAMPLE



T_1: Delete 44

T_2: Find 41

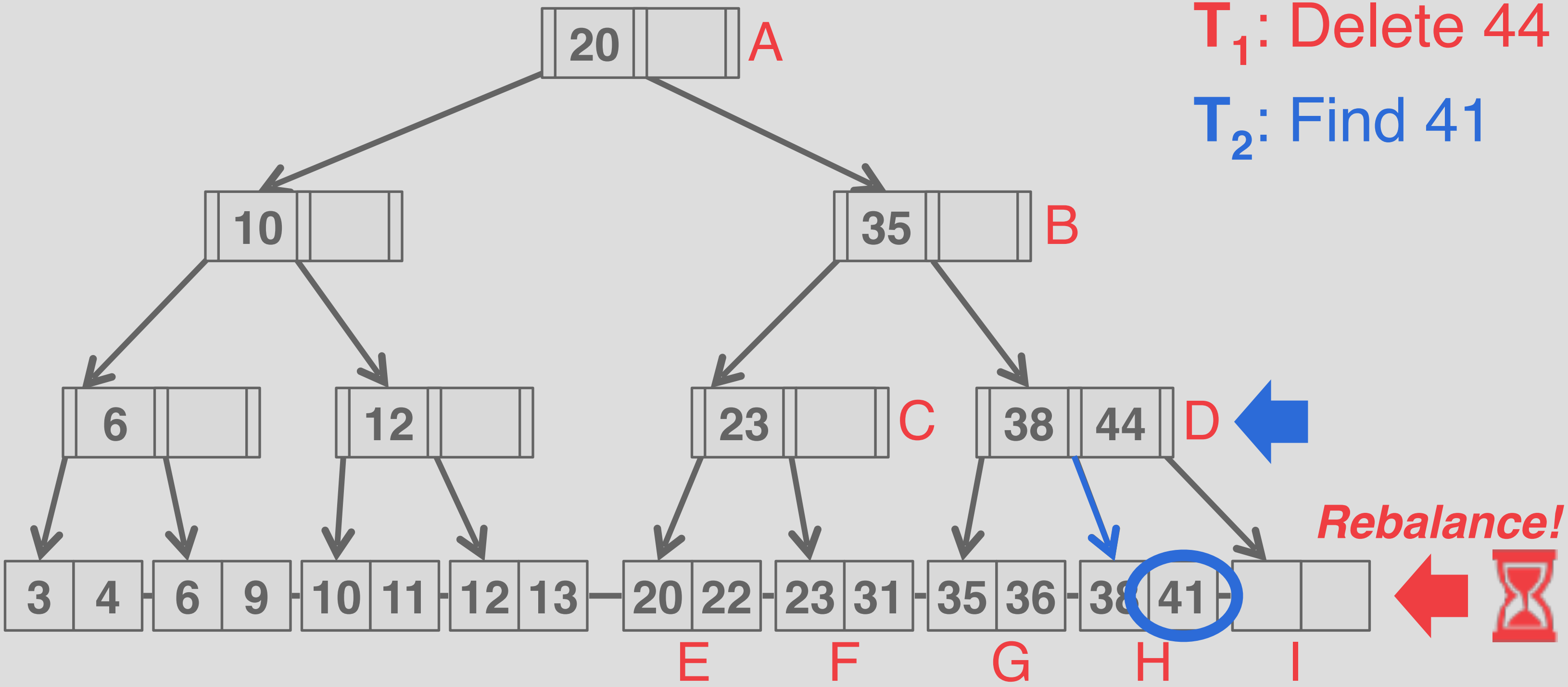
B+TREE MULTI-THREADED EXAMPLE



T_1: Delete 44

T_2: Find 41

B+TREE MULTI-THREADED EXAMPLE



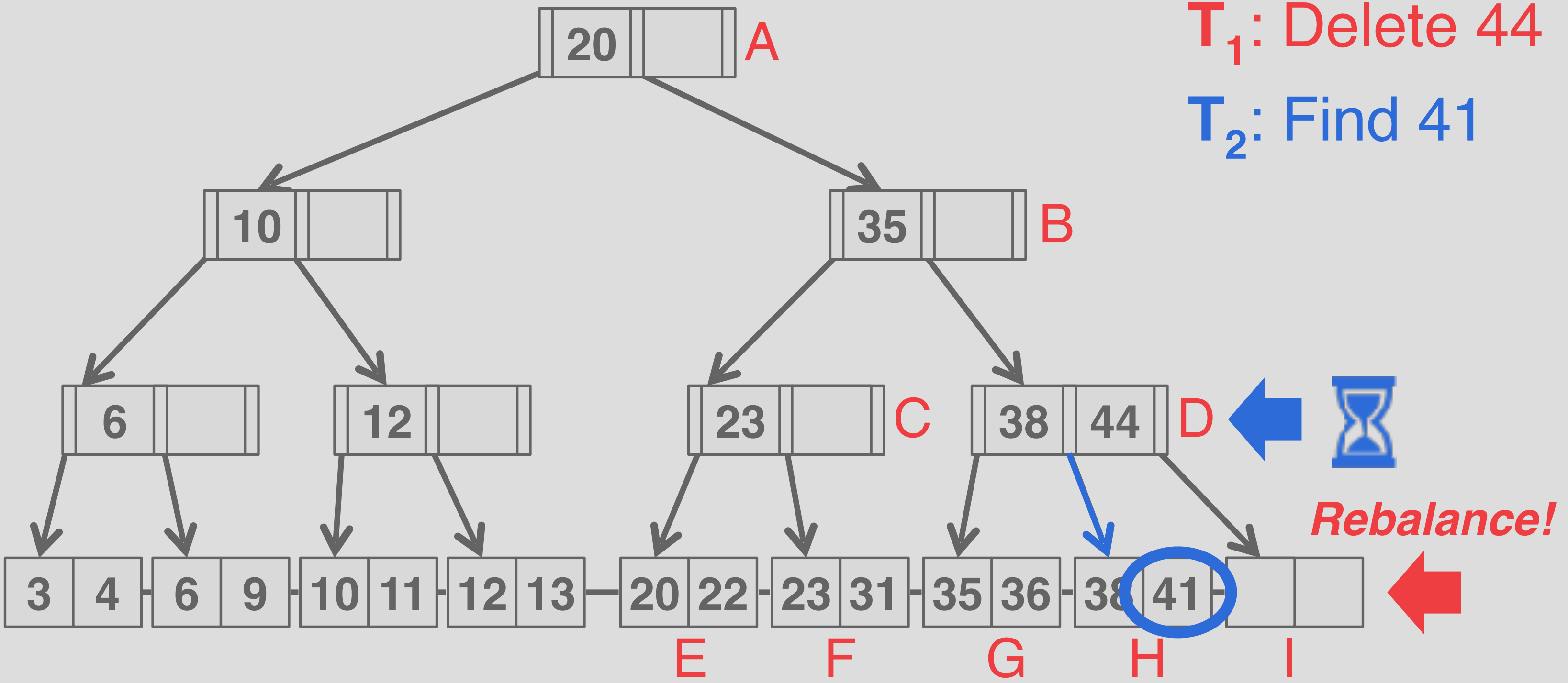
T_1: Delete 44

T_2: Find 41

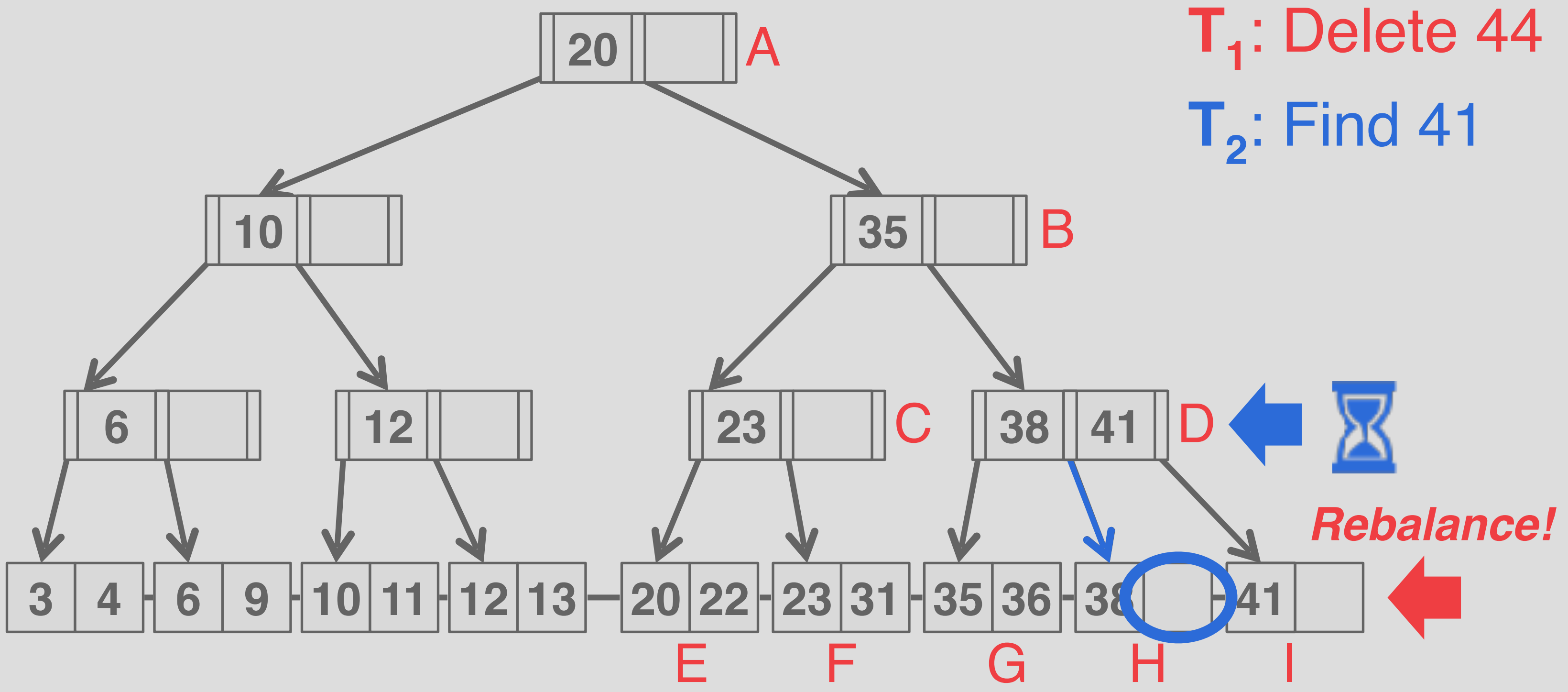
B+TREE MULTI-THREADED EXAMPLE

T$_1$: Delete 44

T$_2$: Find 41



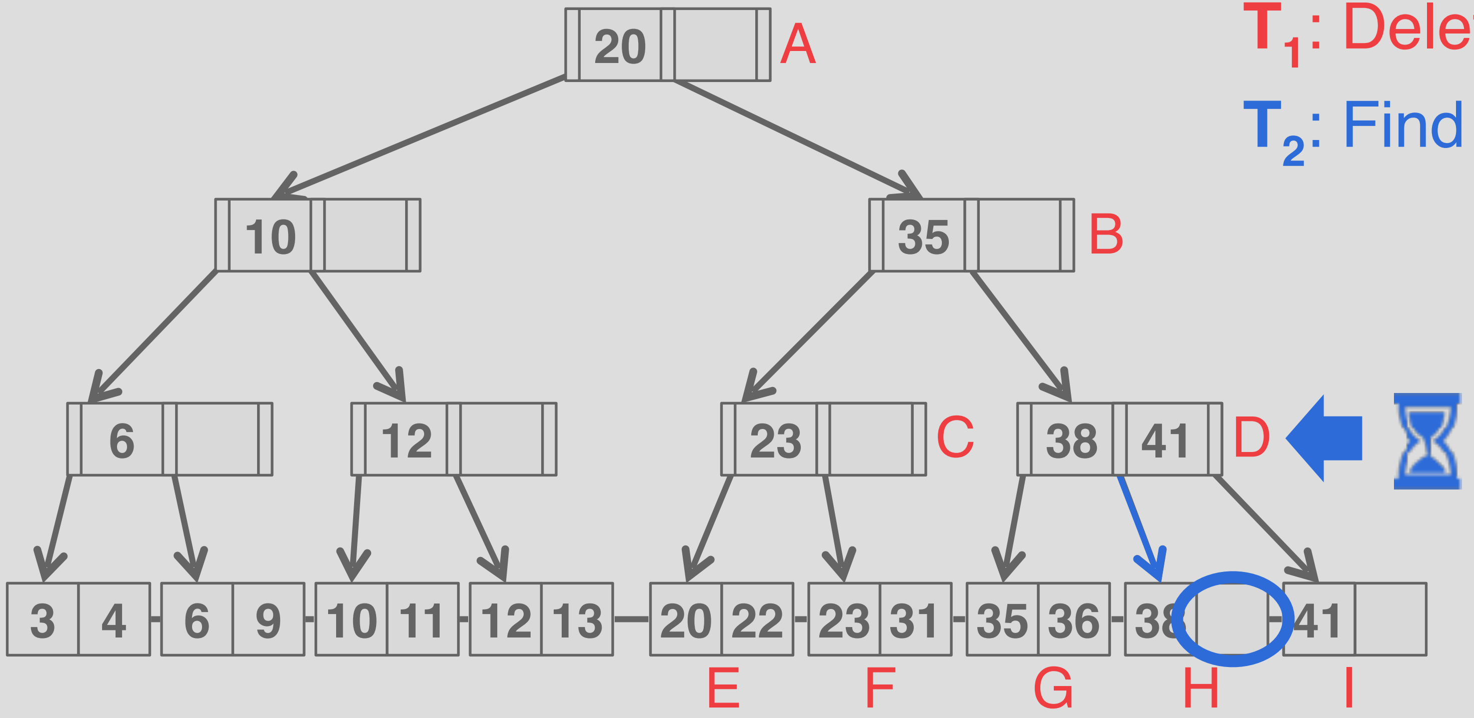
B+TREE MULTI-THREADED EXAMPLE



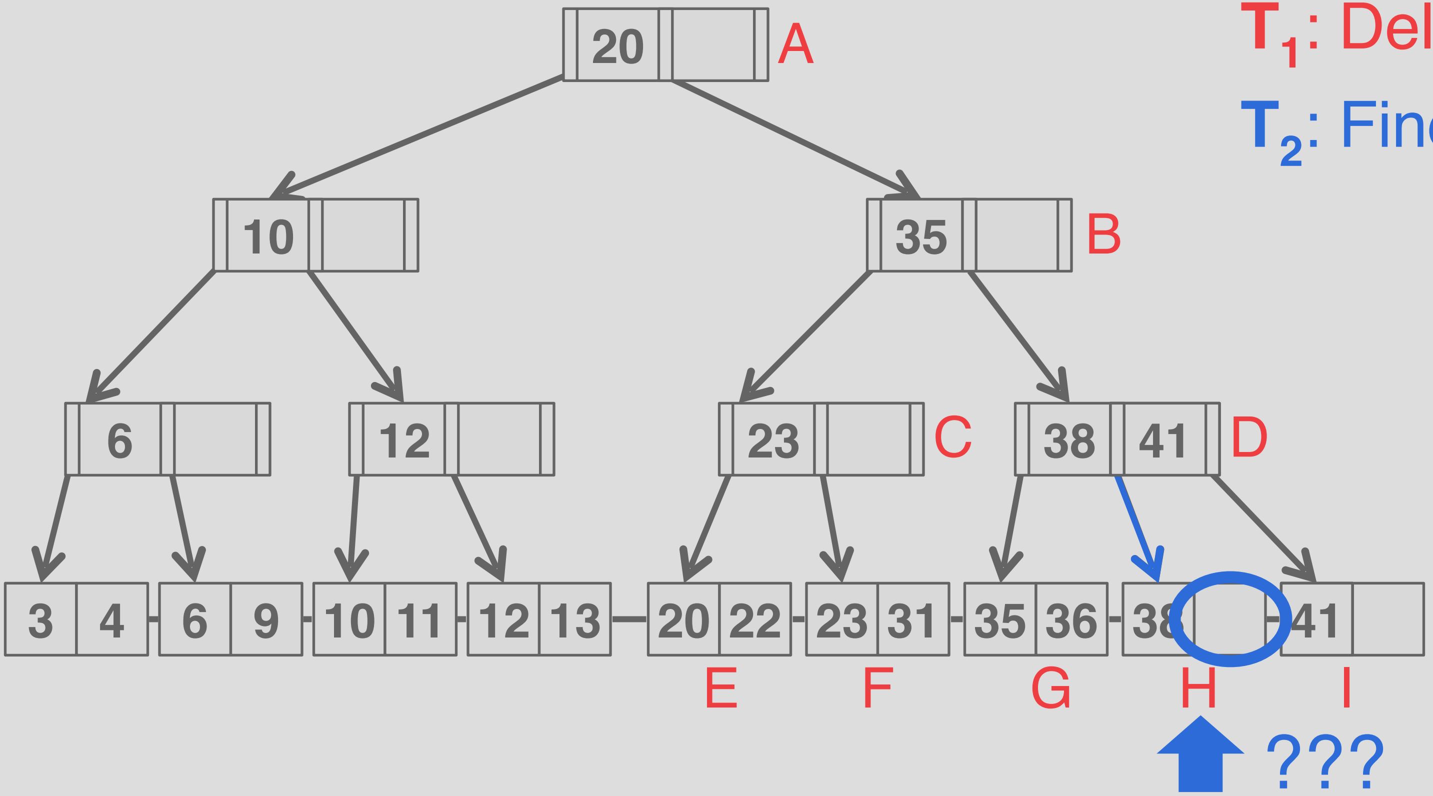
B+TREE MULTI-THREADED EXAMPLE

T_1: Delete 44

T_2: Find 41



B+TREE MULTI-THREADED EXAMPLE



T_1: Delete 44

T_2: Find 41

LATCH CRABBING/COUPLING

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

→ 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

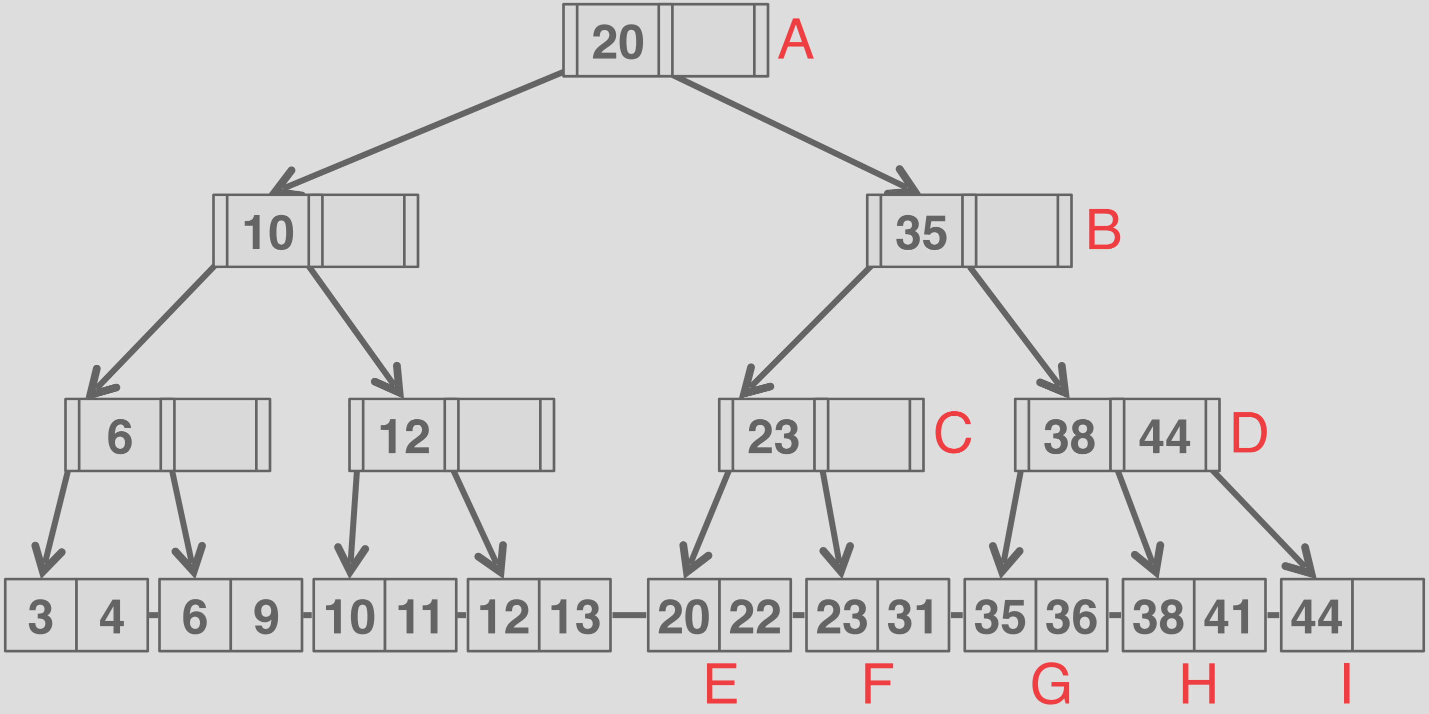
Find: Start at root and traverse down the tree:

→ Acquire **R** latch on child,
→ Then unlatch parent.
→ Repeat until we reach the leaf node.

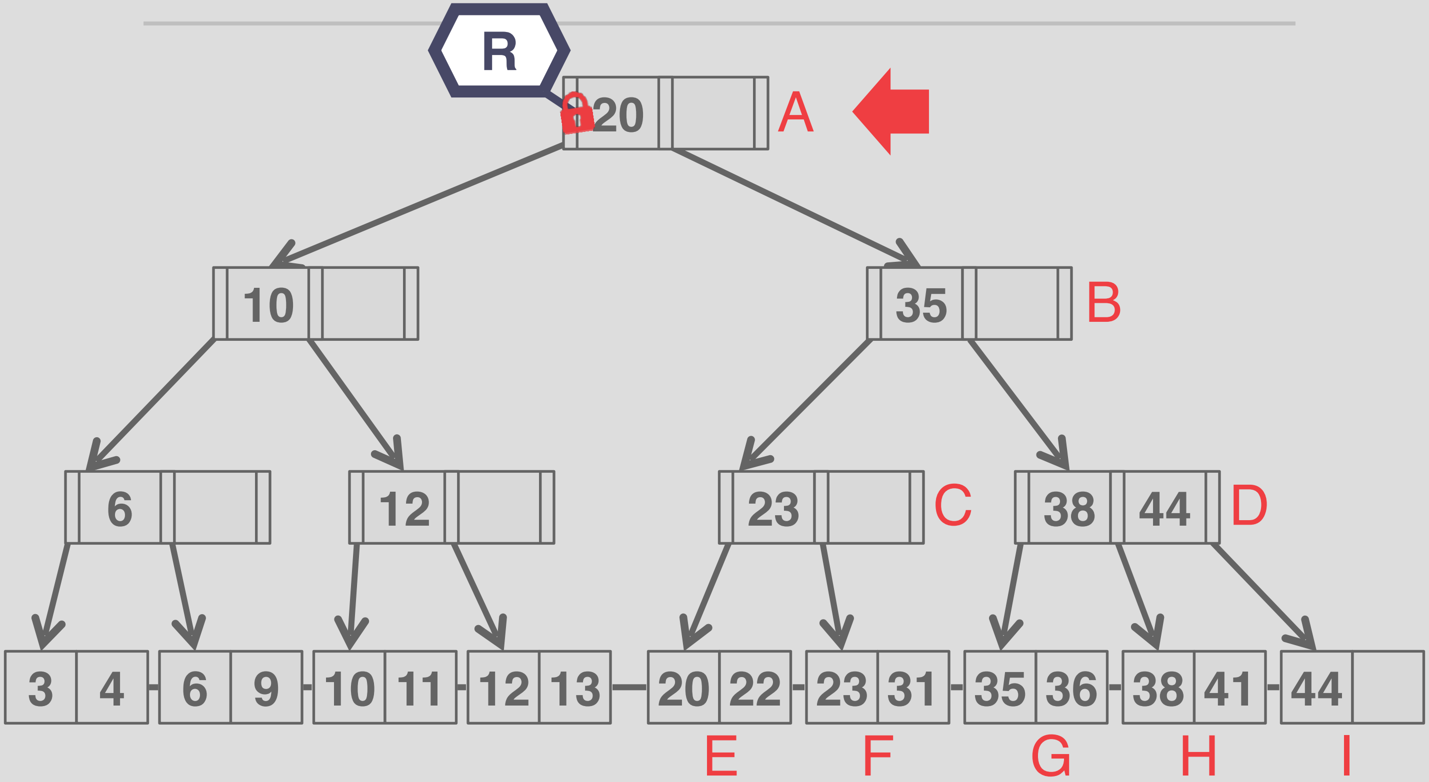
Insert/Delete: Start at root and go down, obtaining **W** latches as needed. Once child is latched, check if it is safe:

→ If child is safe, release all latches on ancestors

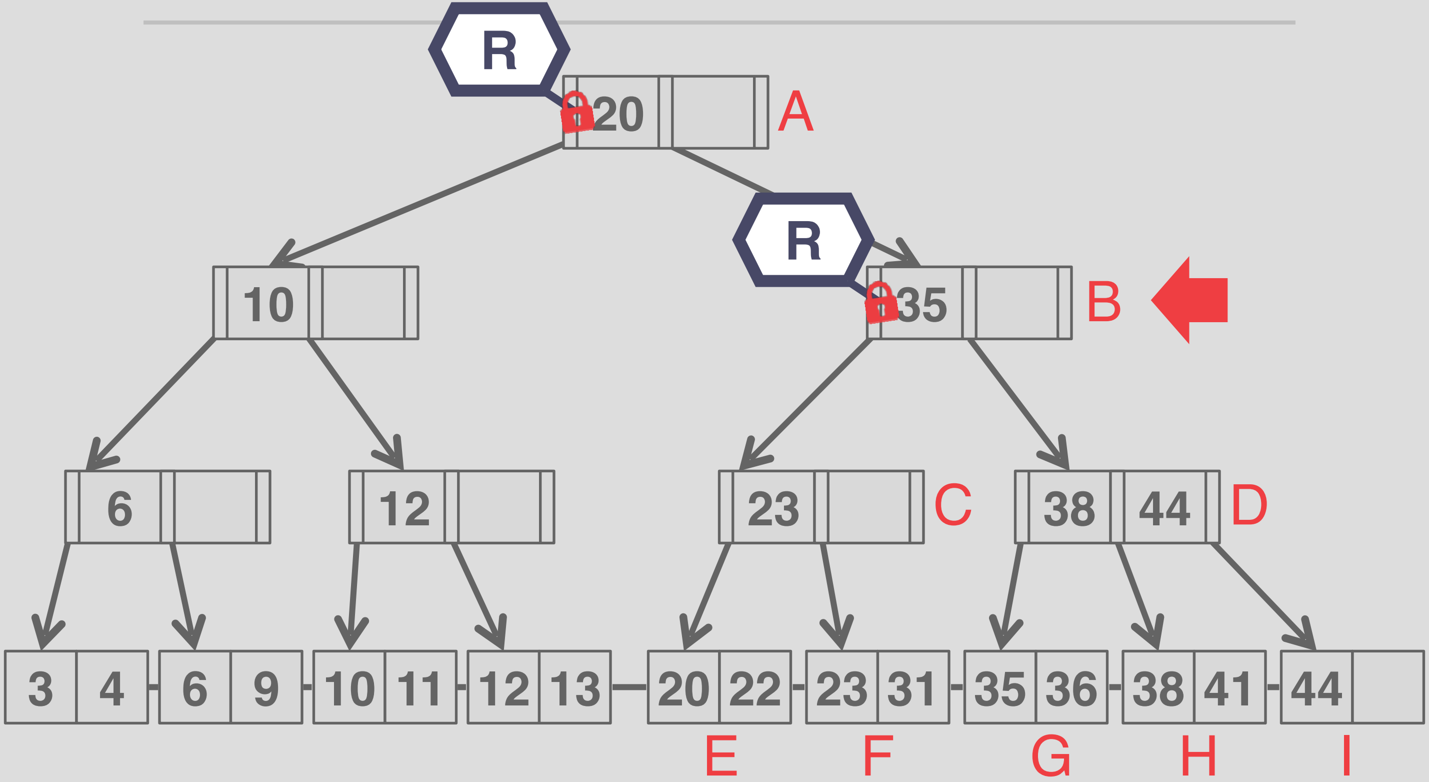
EXAMPLE #1 – FIND 38



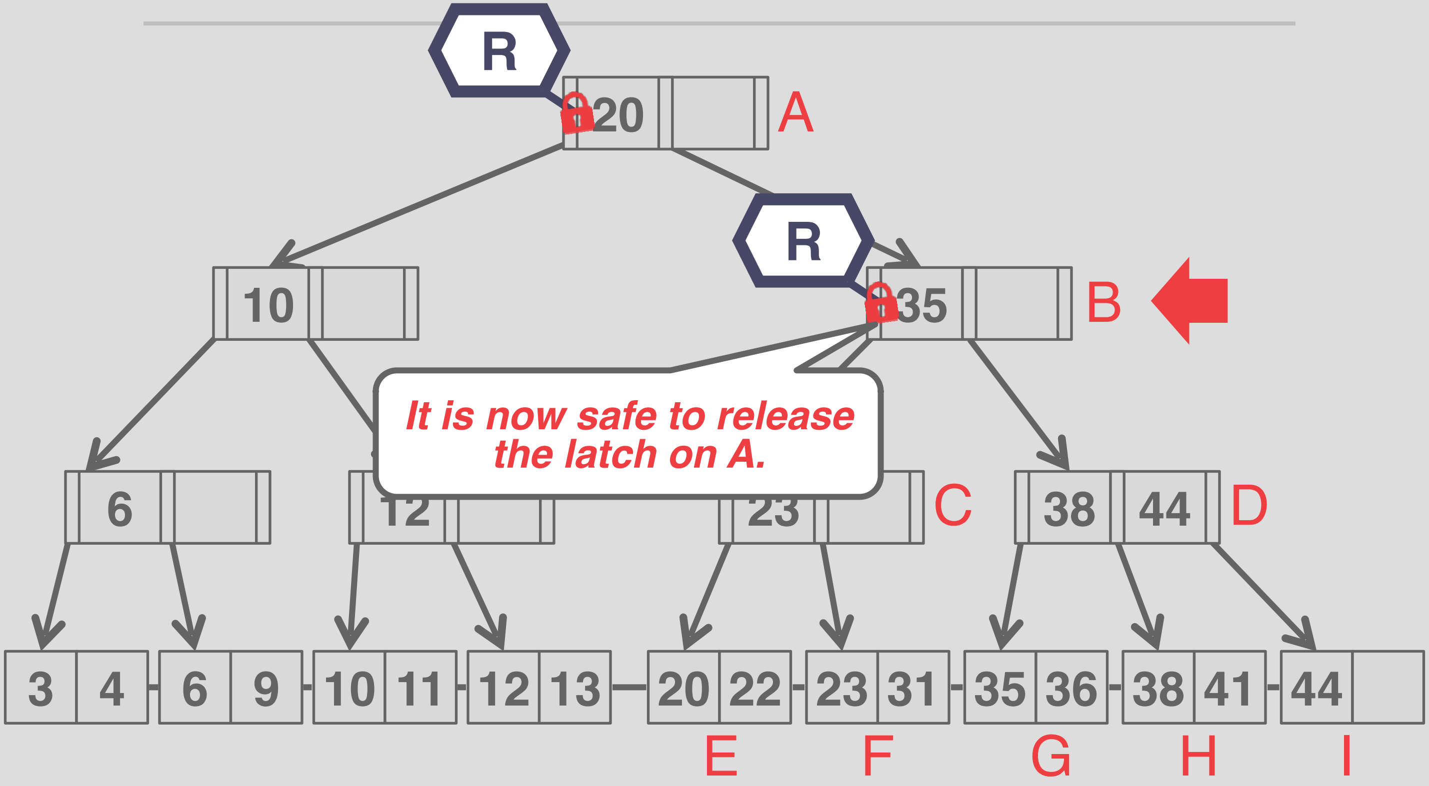
EXAMPLE #1 – FIND 38



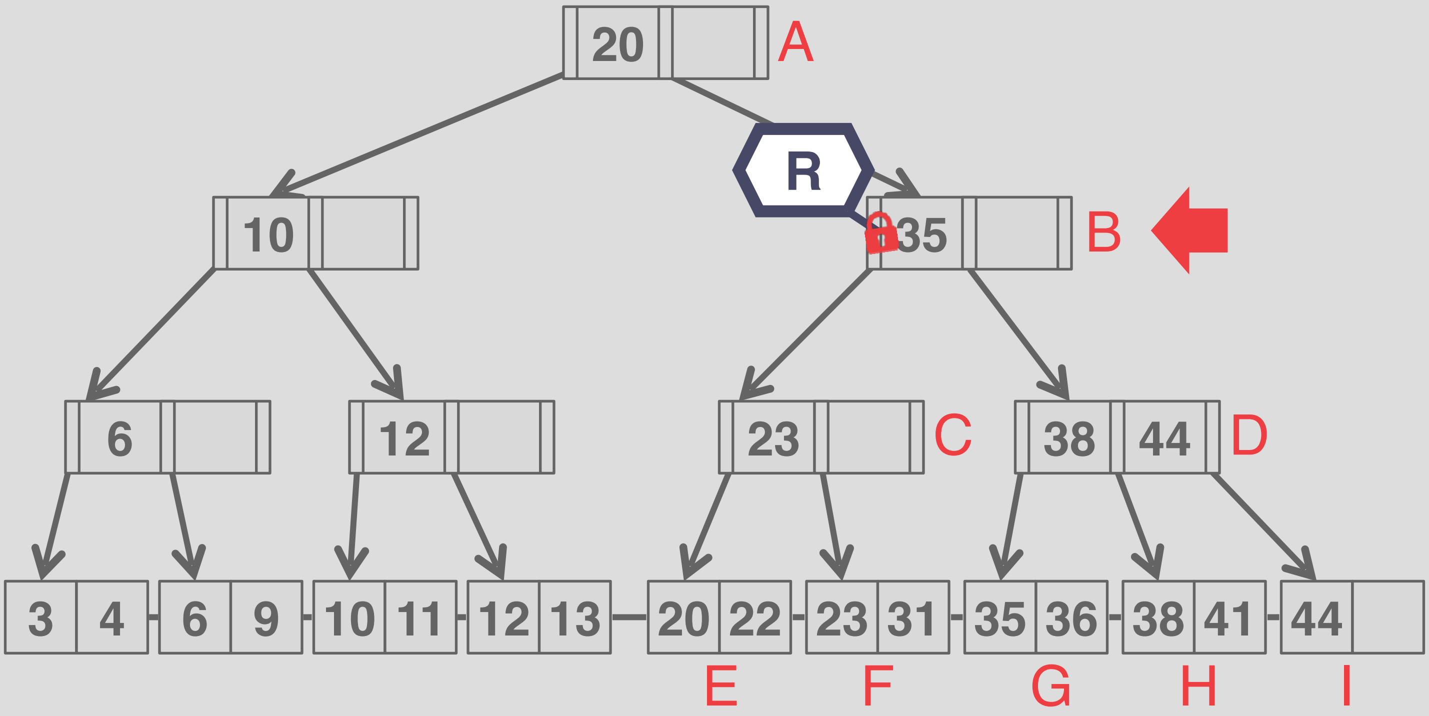
EXAMPLE #1 – FIND 38



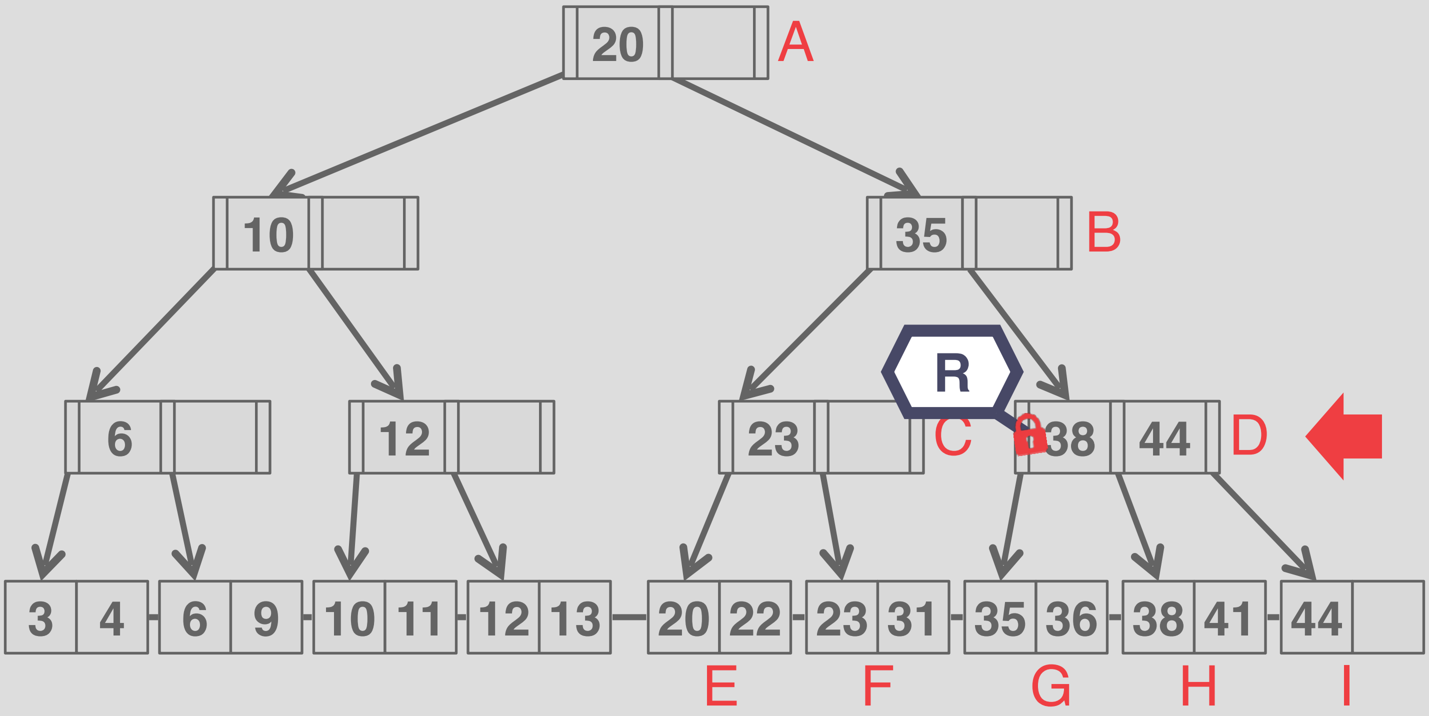
EXAMPLE #1 – FIND 38



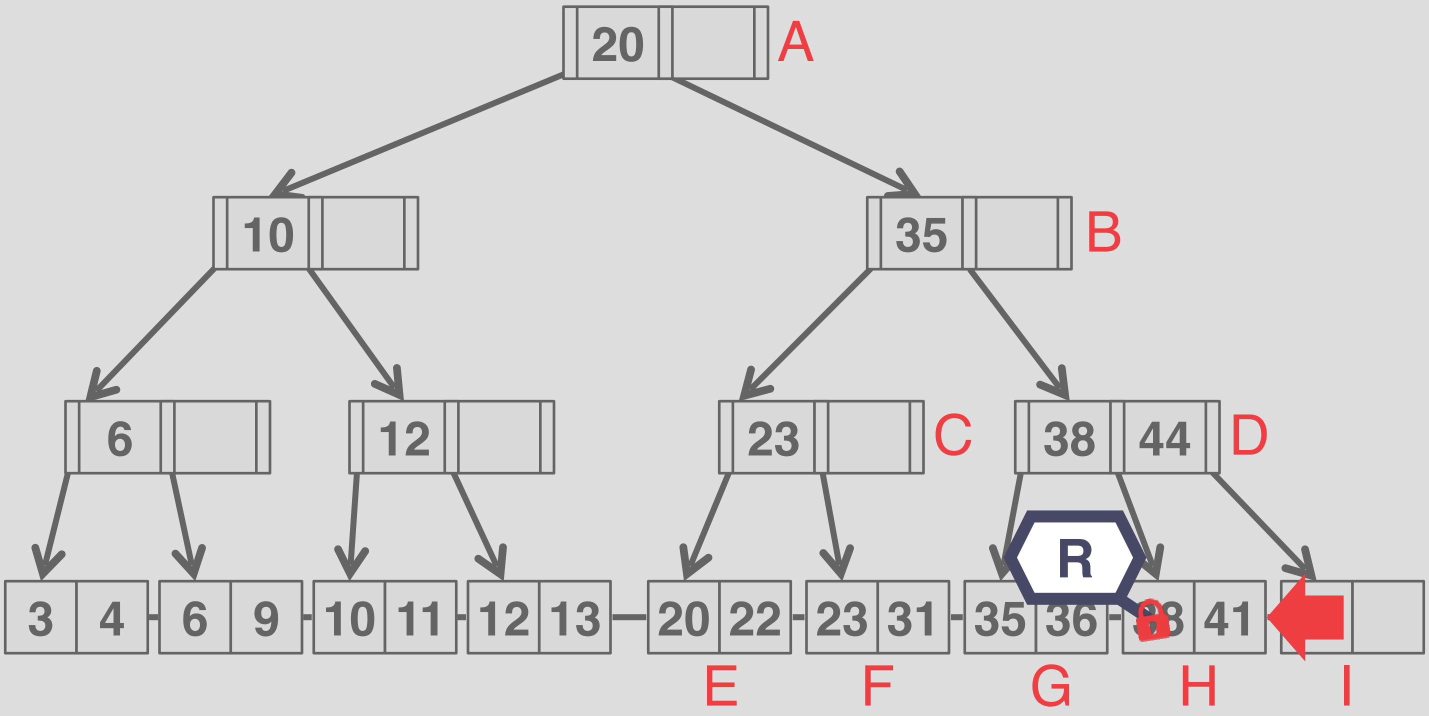
EXAMPLE #1 – FIND 38



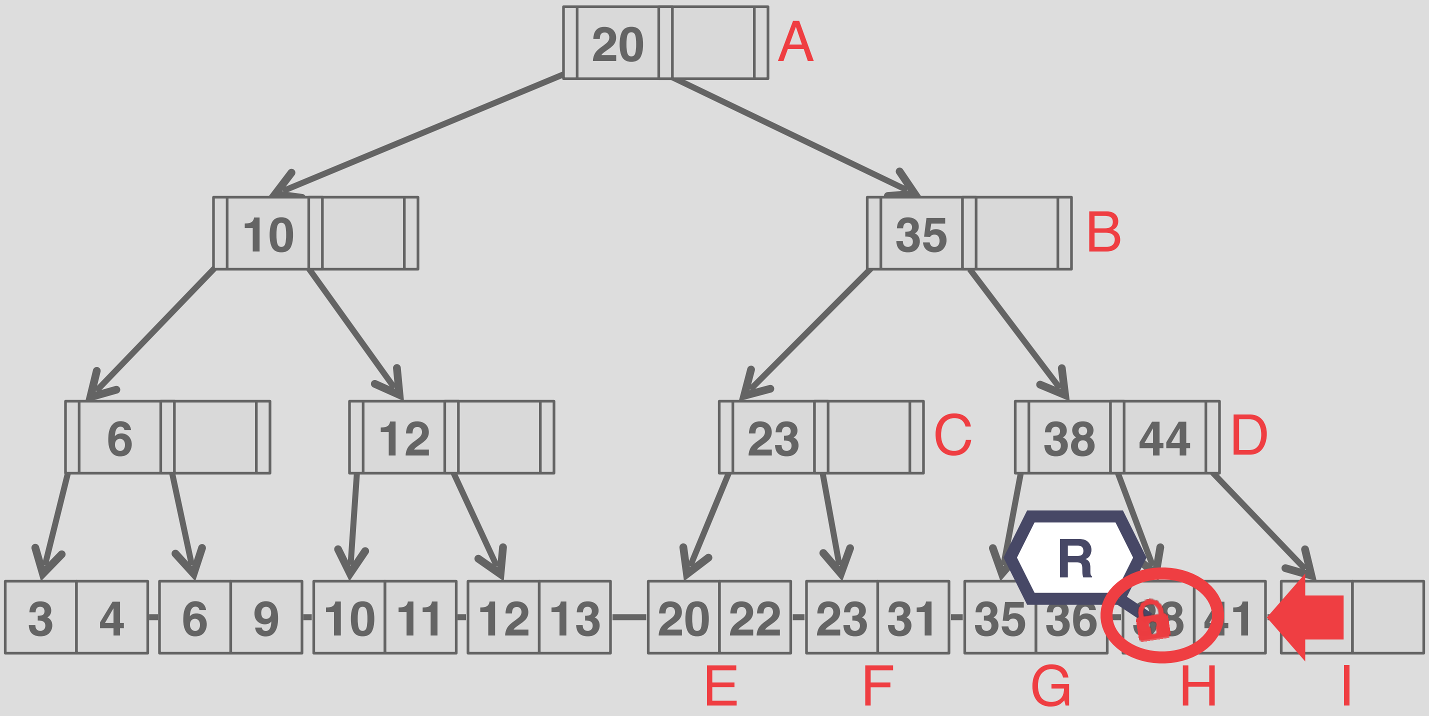
EXAMPLE #1 – FIND 38



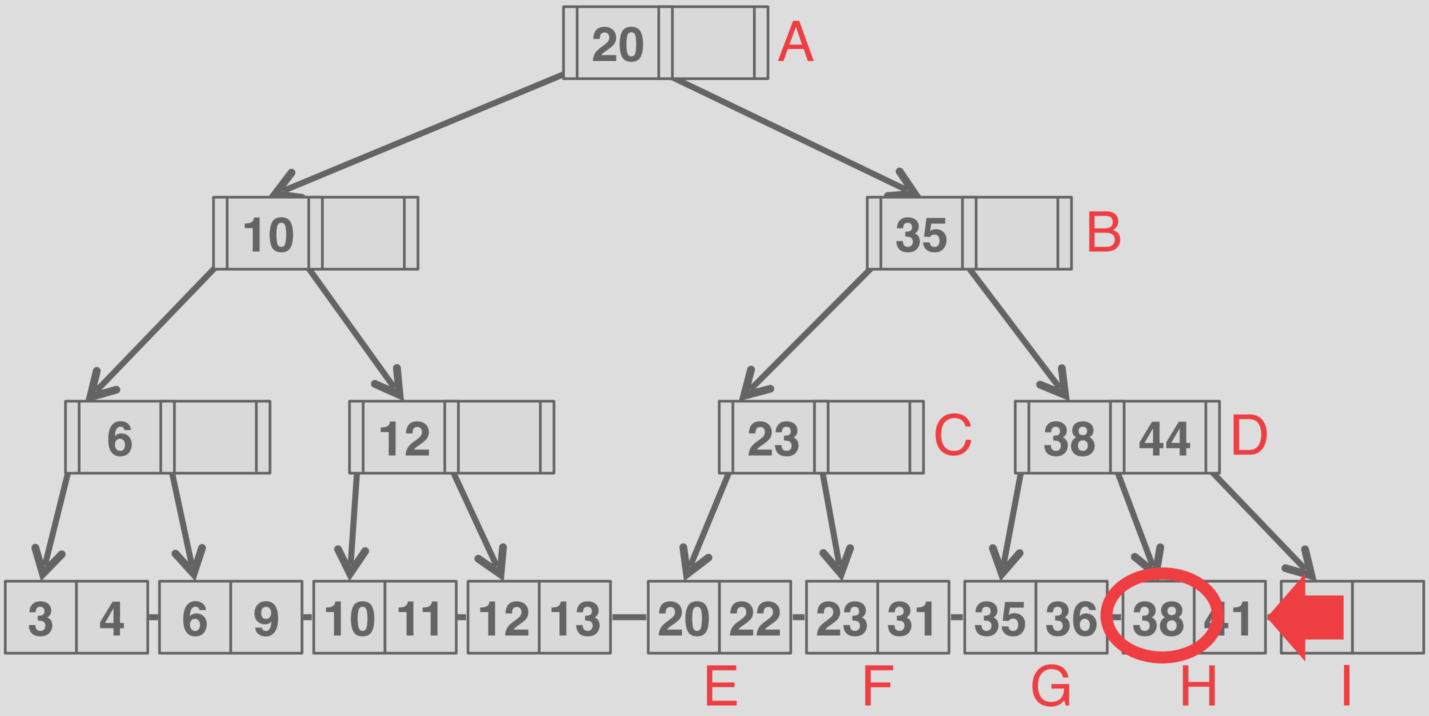
EXAMPLE #1 – FIND 38



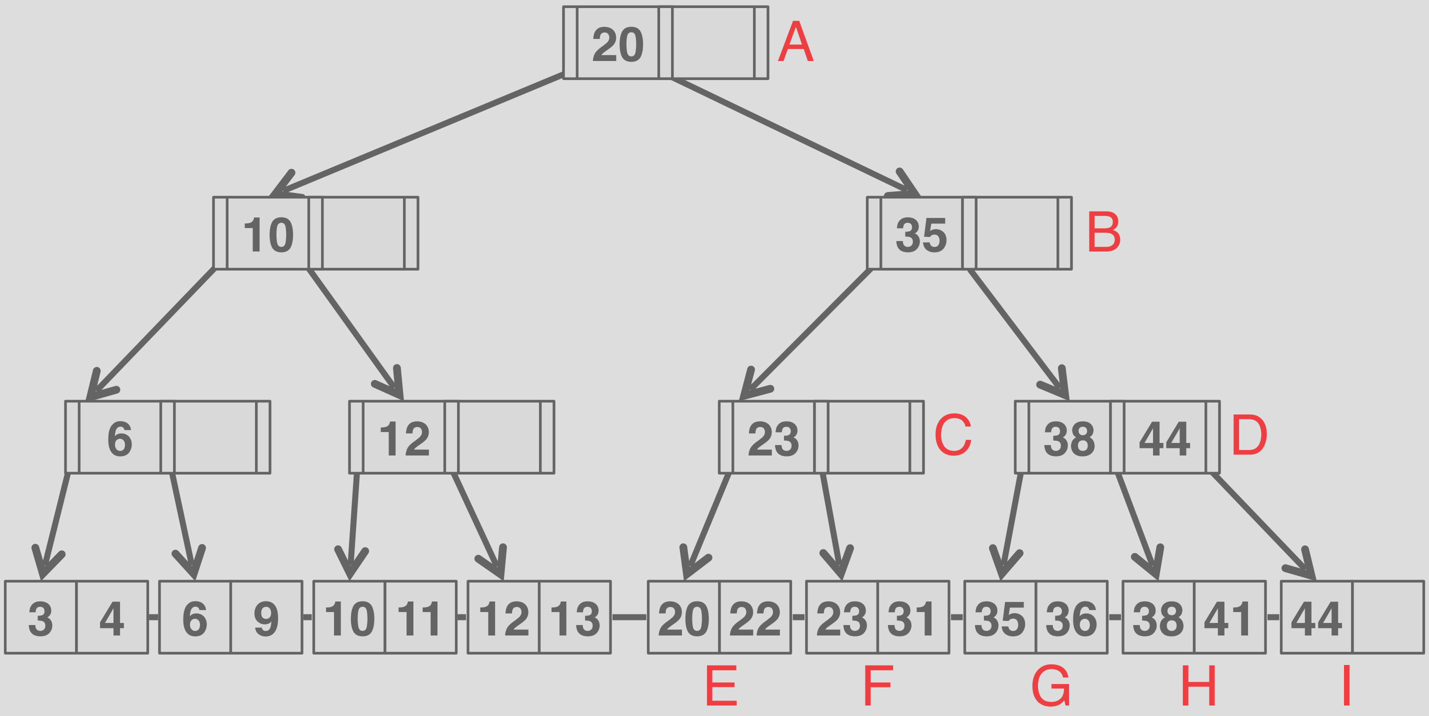
EXAMPLE #1 – FIND 38



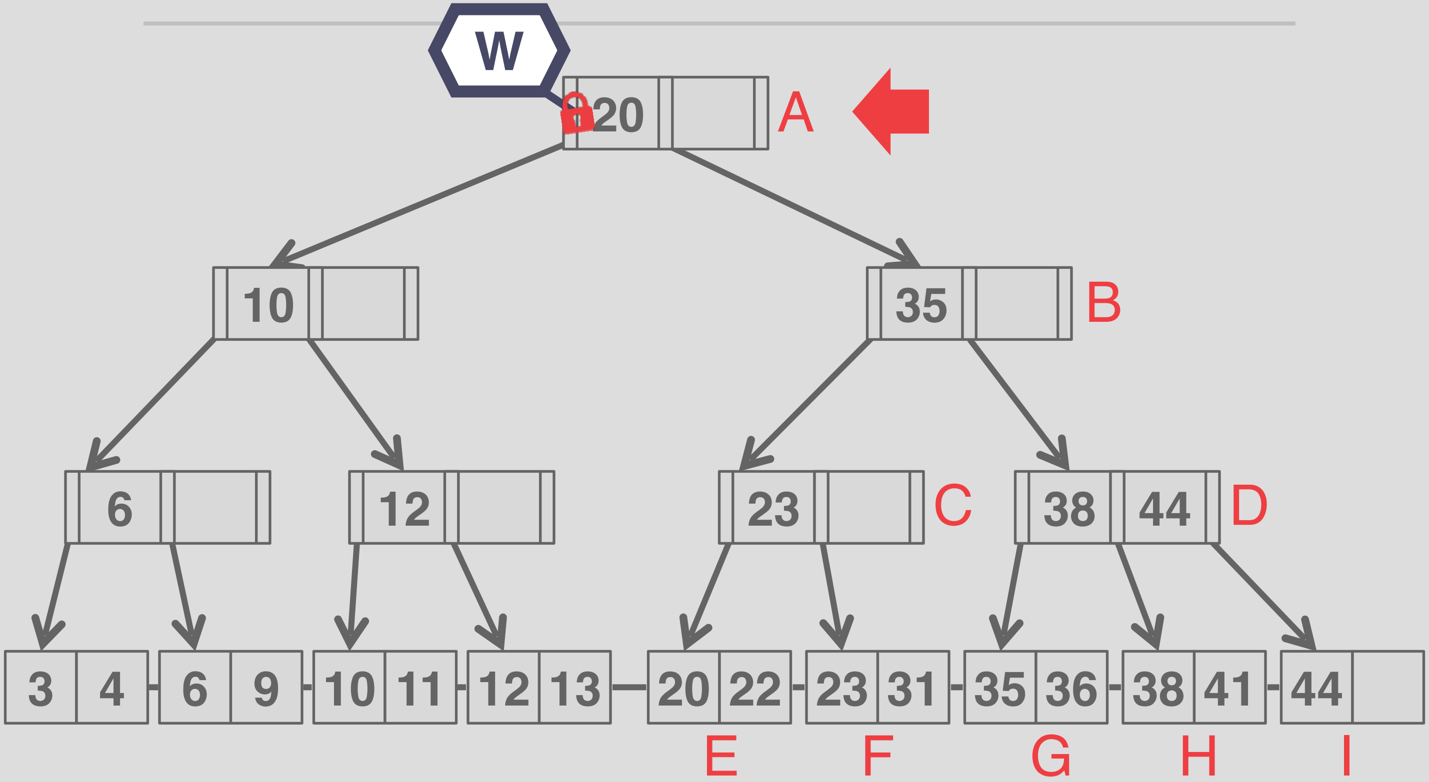
EXAMPLE #1 – FIND 38



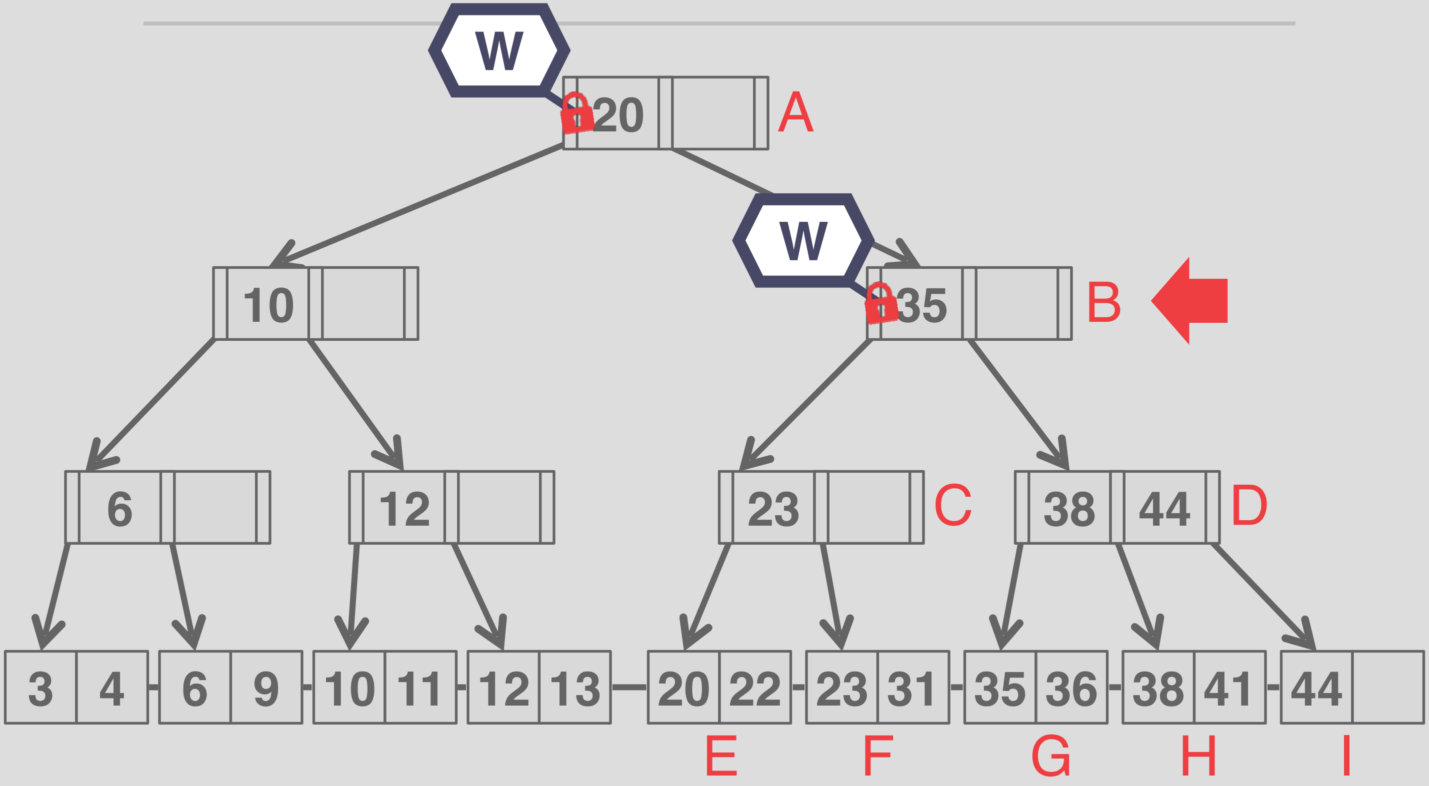
EXAMPLE #2 – DELETE 38



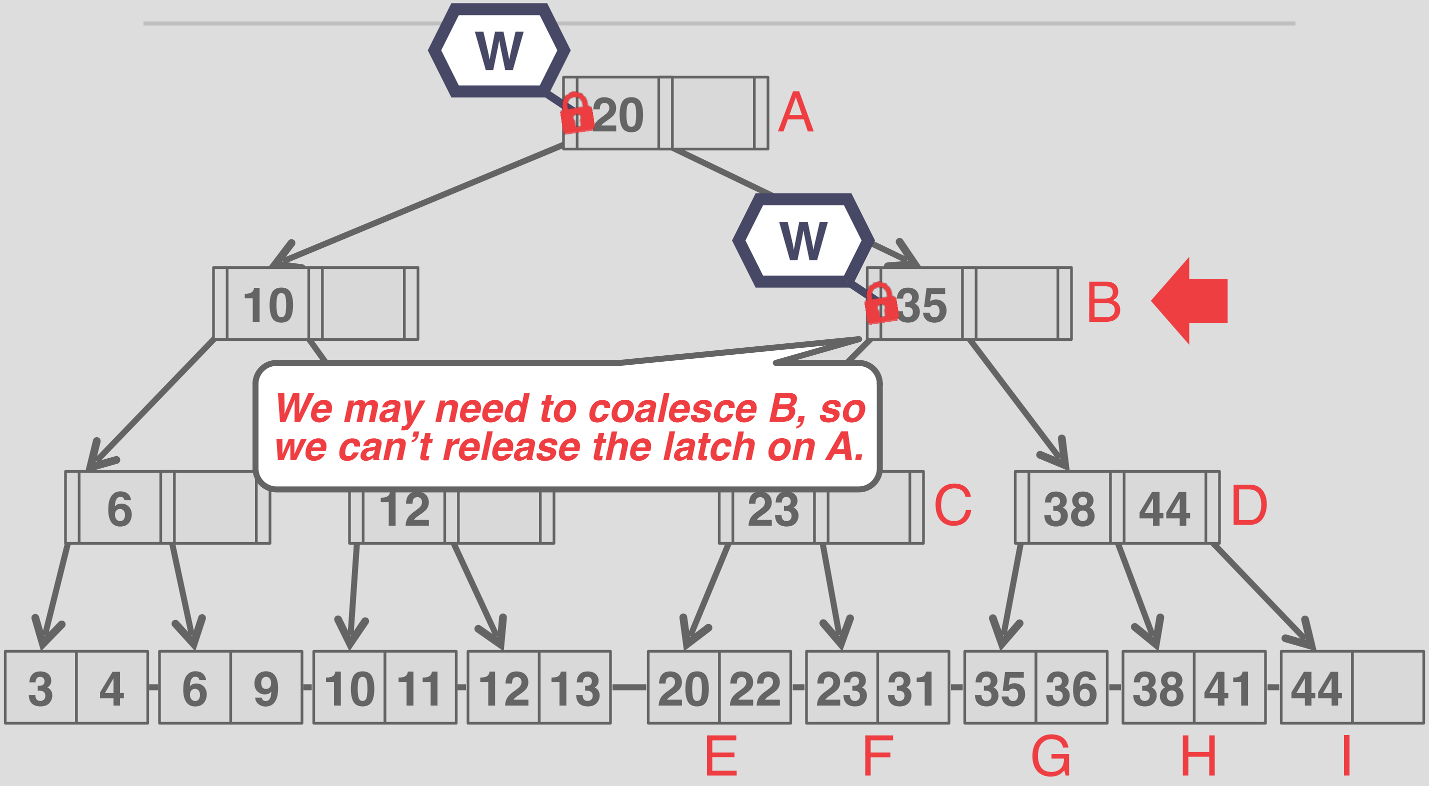
EXAMPLE #2 – DELETE 38



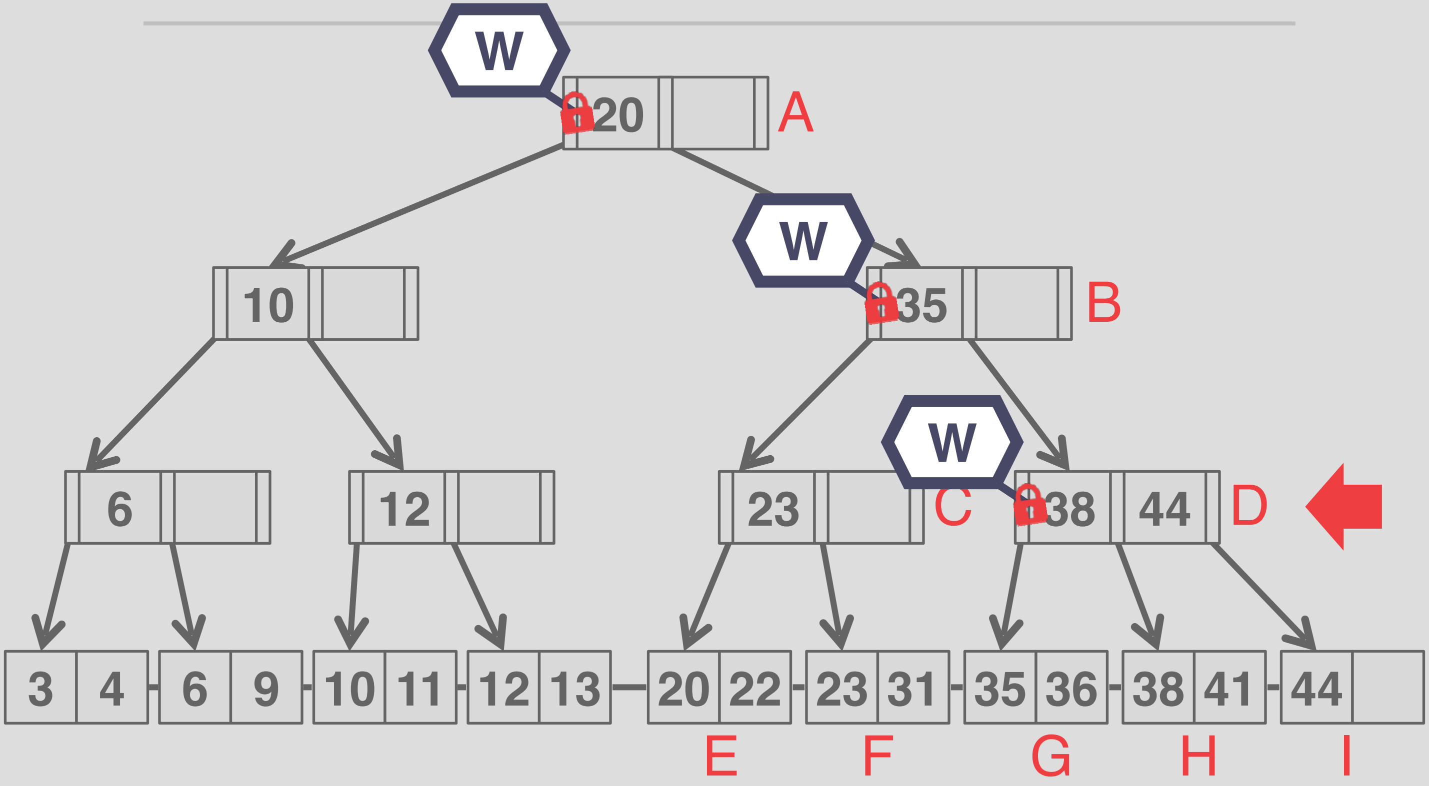
EXAMPLE #2 – DELETE 38



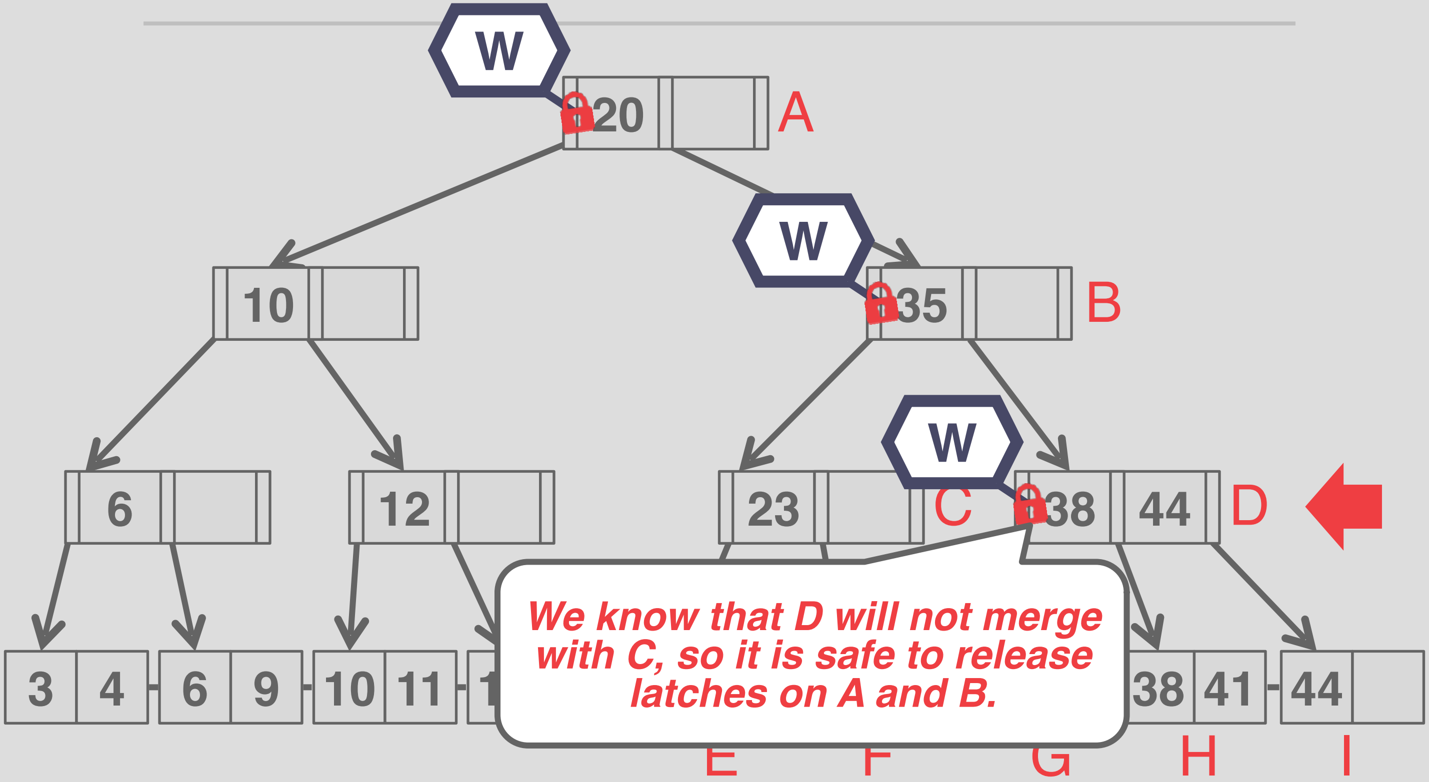
EXAMPLE #2 – DELETE 38



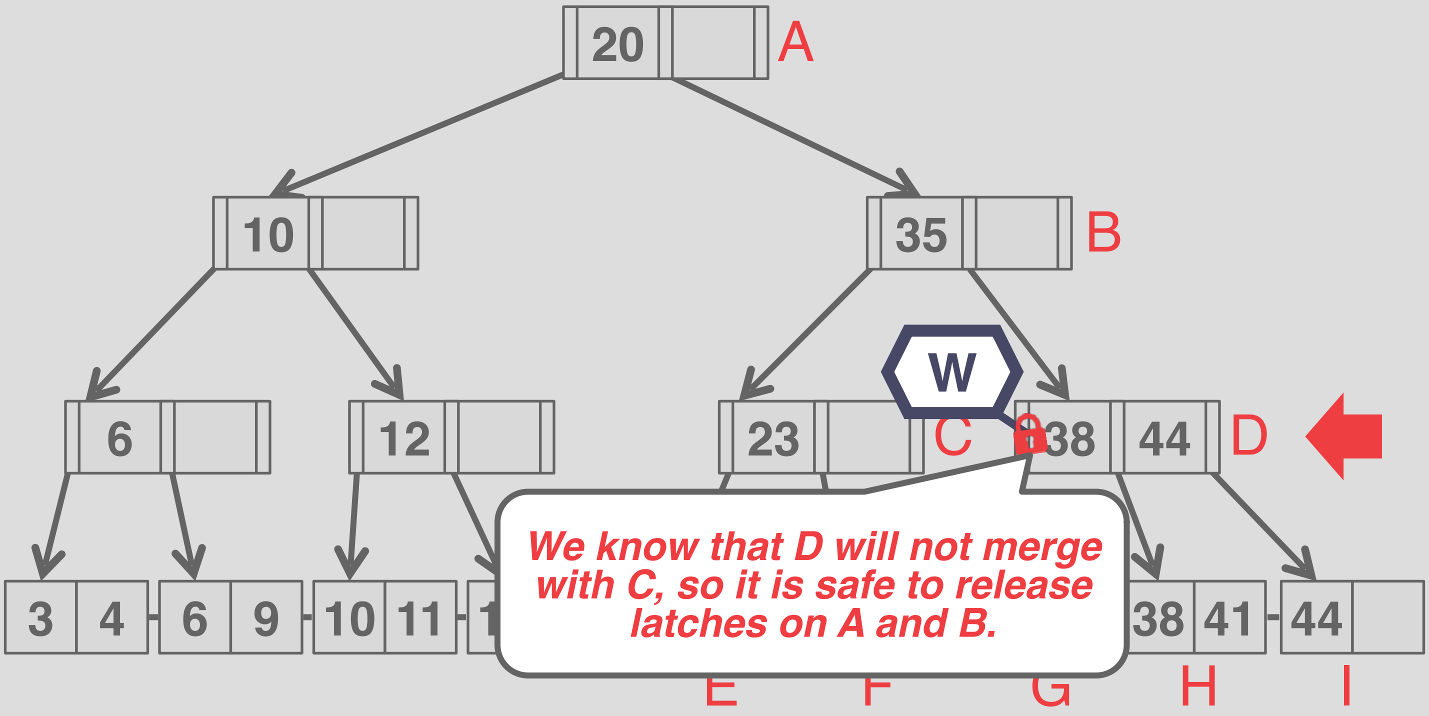
EXAMPLE #2 – DELETE 38



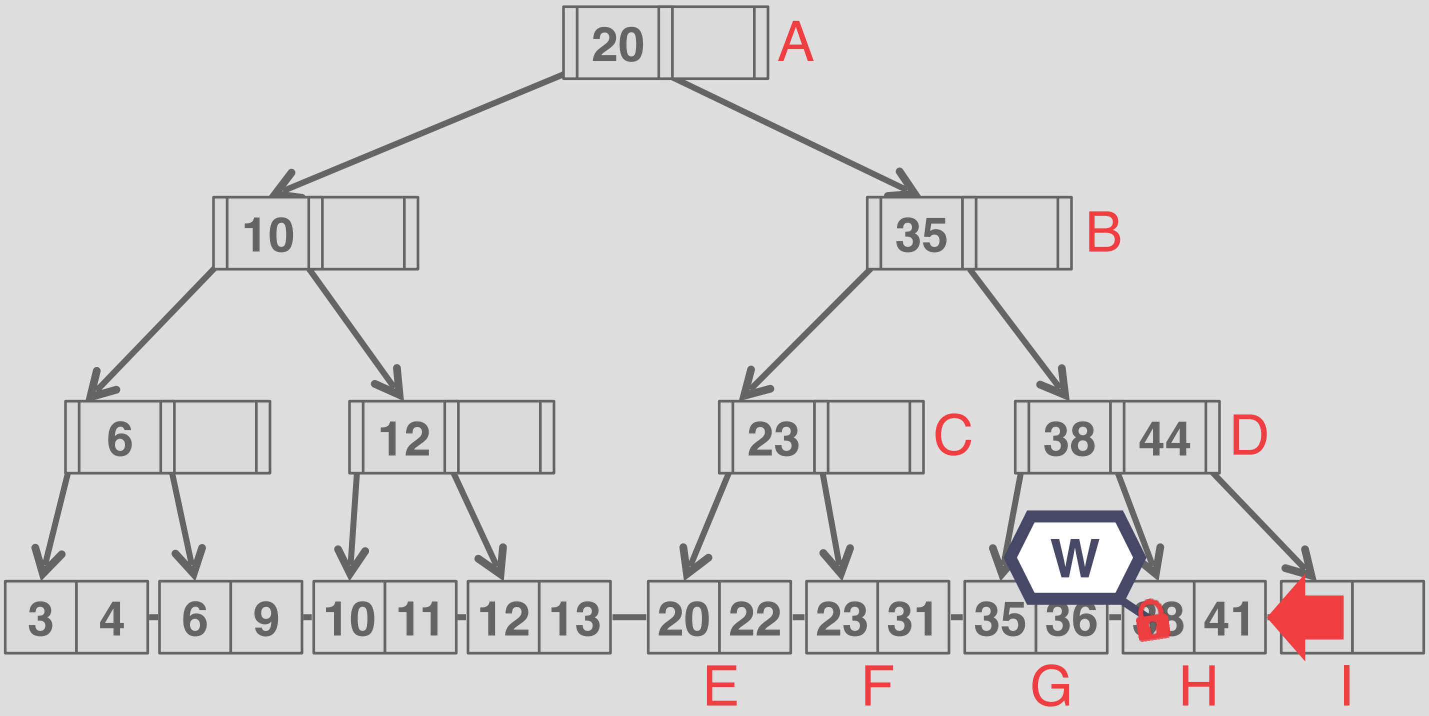
EXAMPLE #2 – DELETE 38



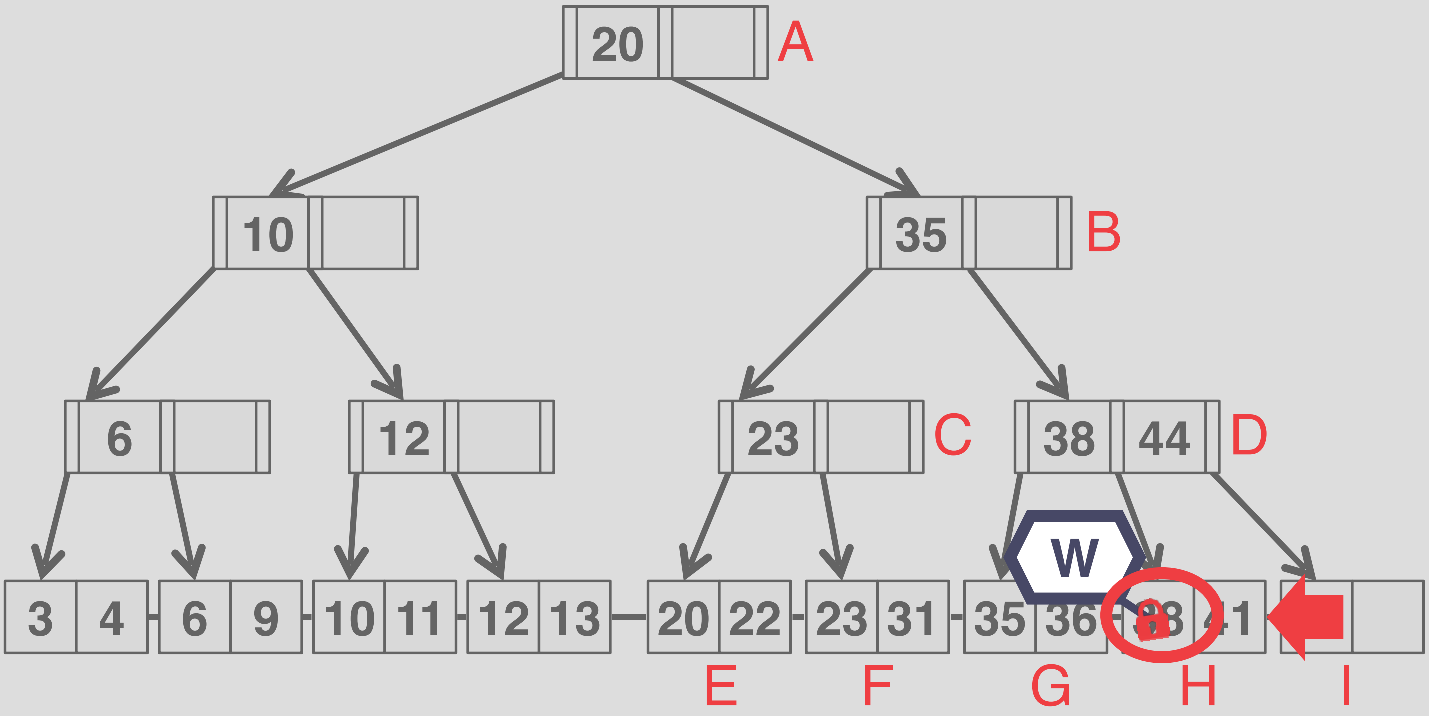
EXAMPLE #2 – DELETE 38



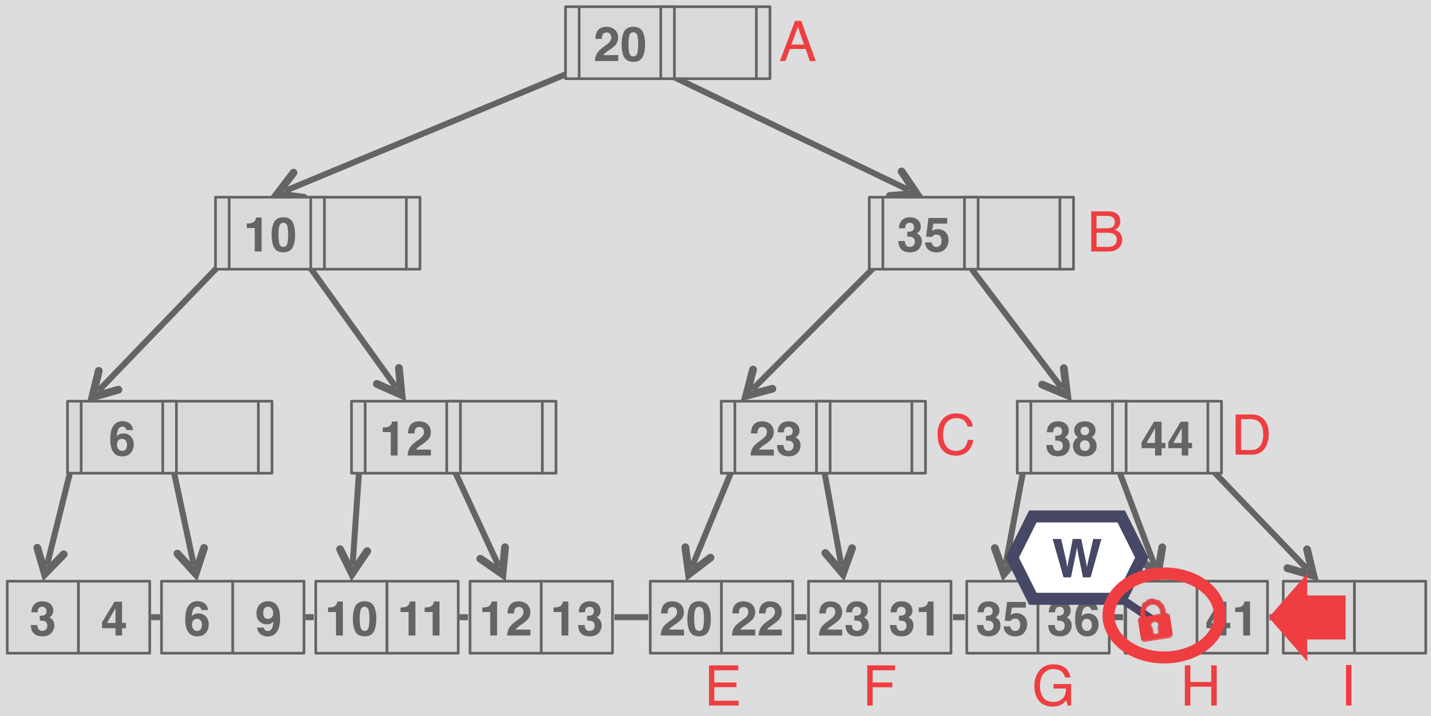
EXAMPLE #2 – DELETE 38



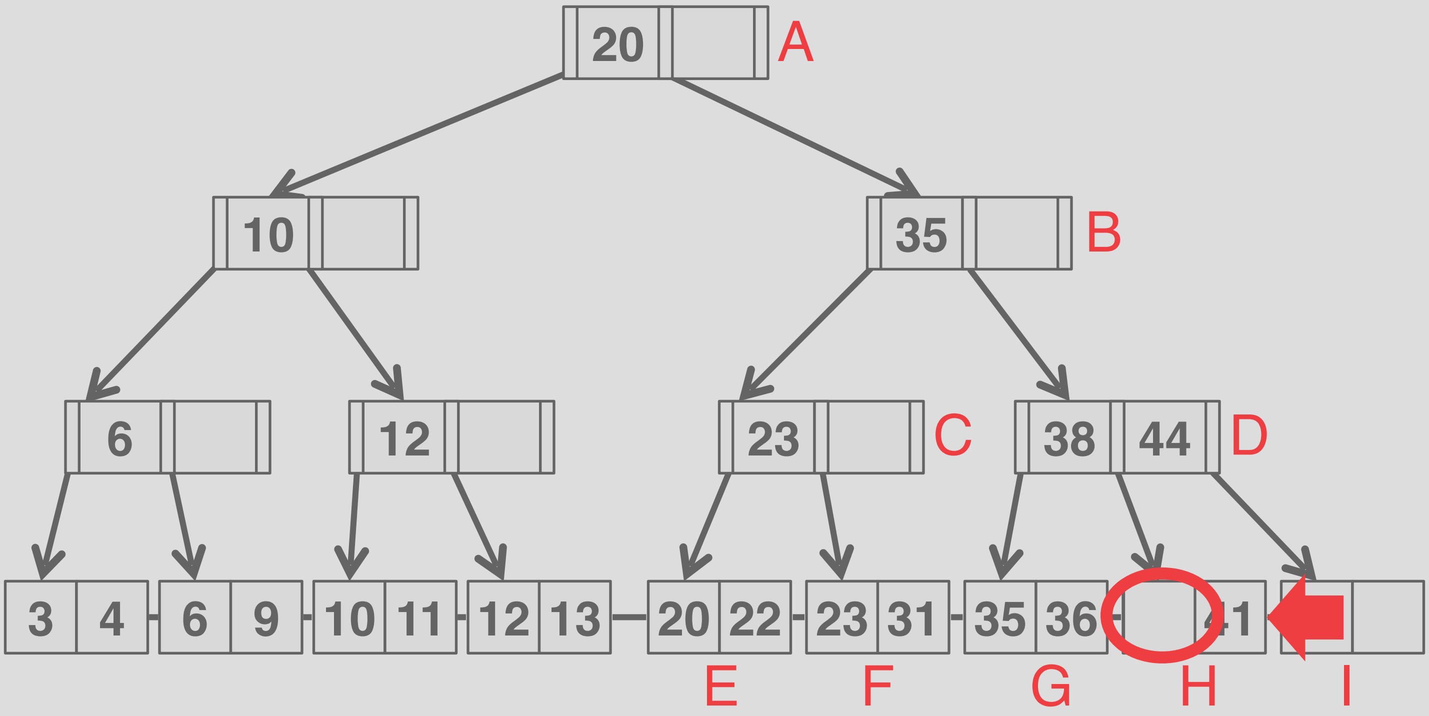
EXAMPLE #2 – DELETE 38



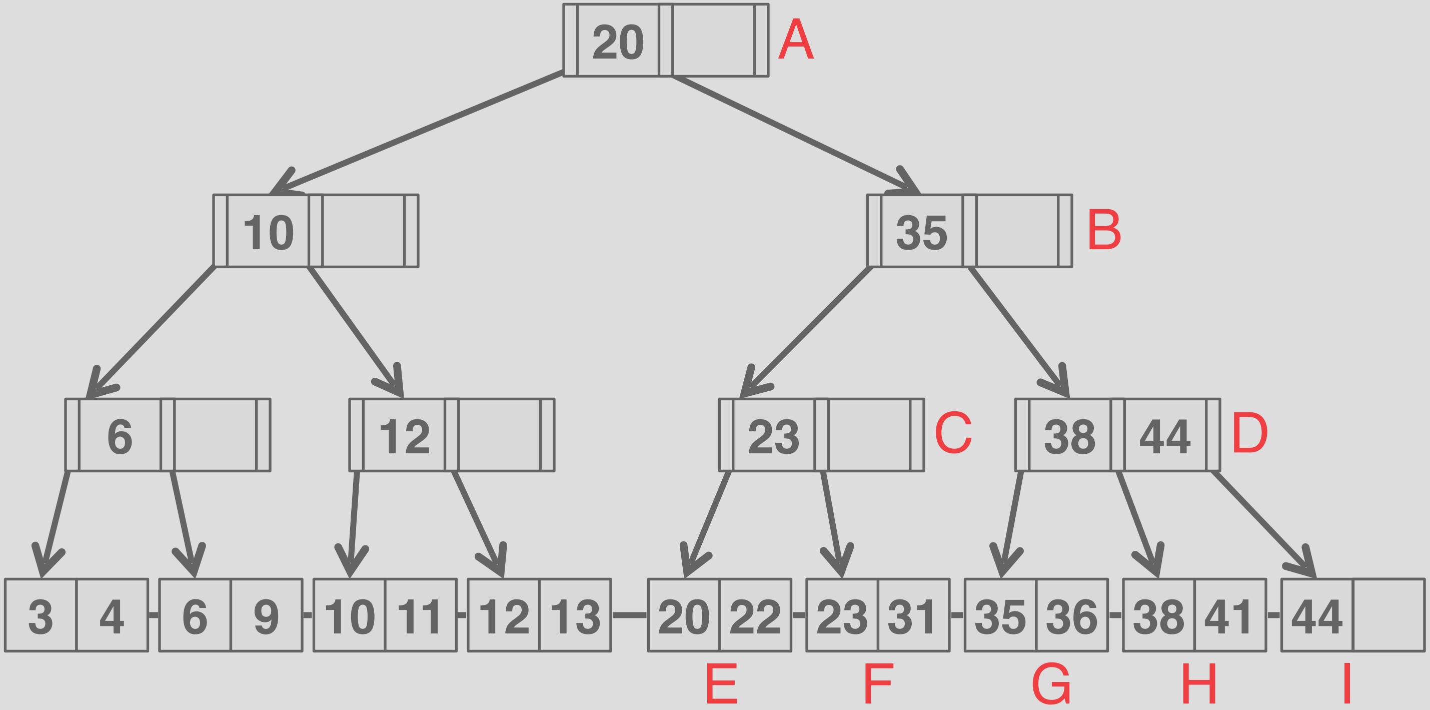
EXAMPLE #2 – DELETE 38



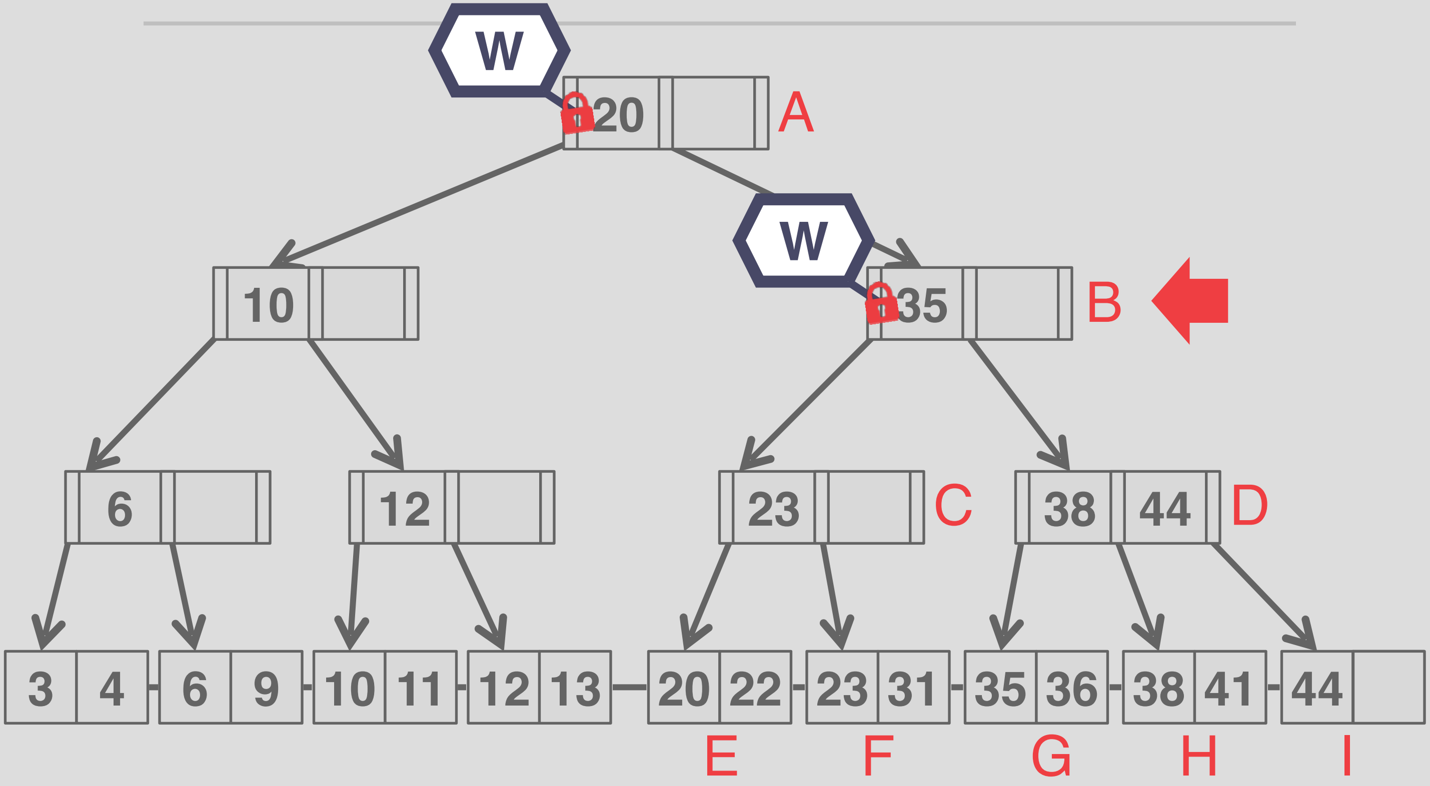
EXAMPLE #2 – DELETE 38



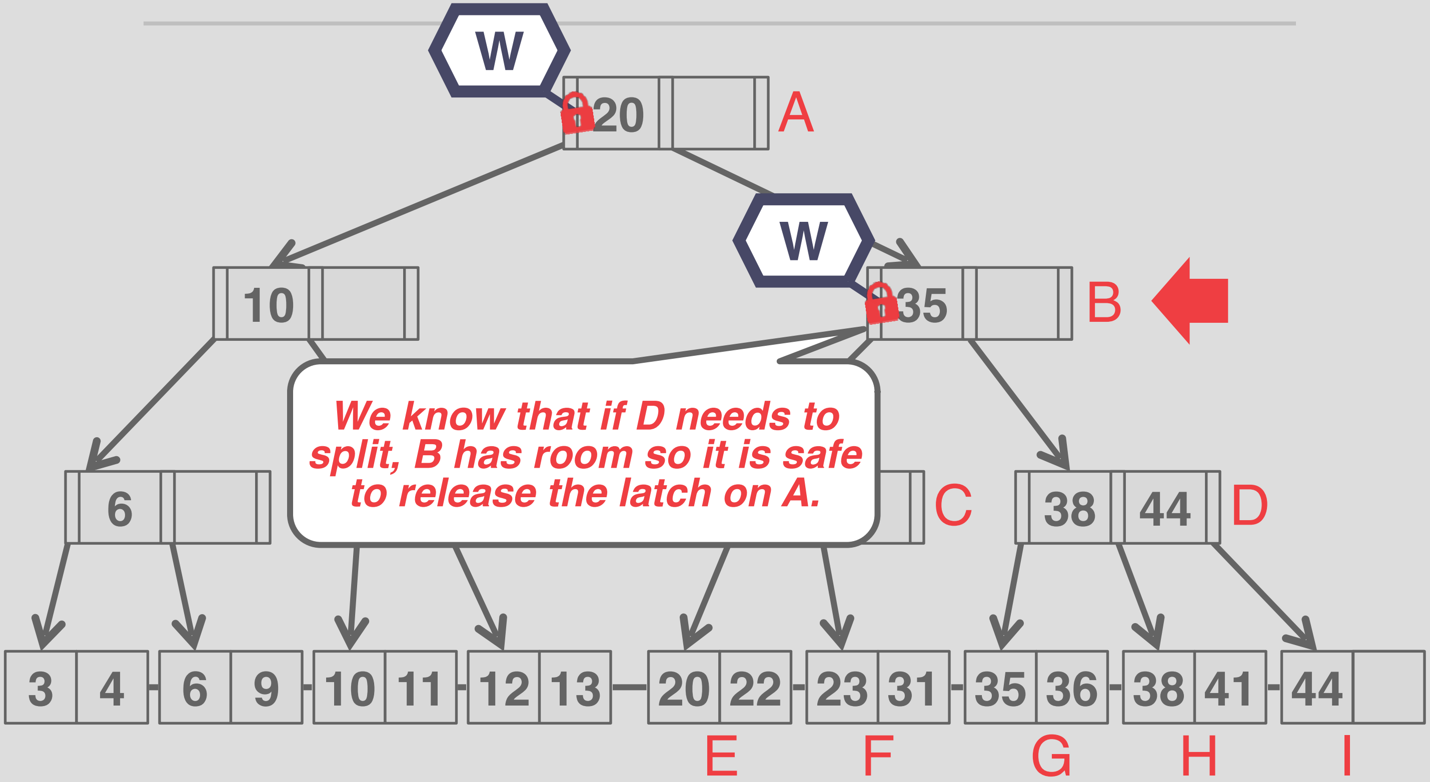
EXAMPLE #3 – INSERT 45



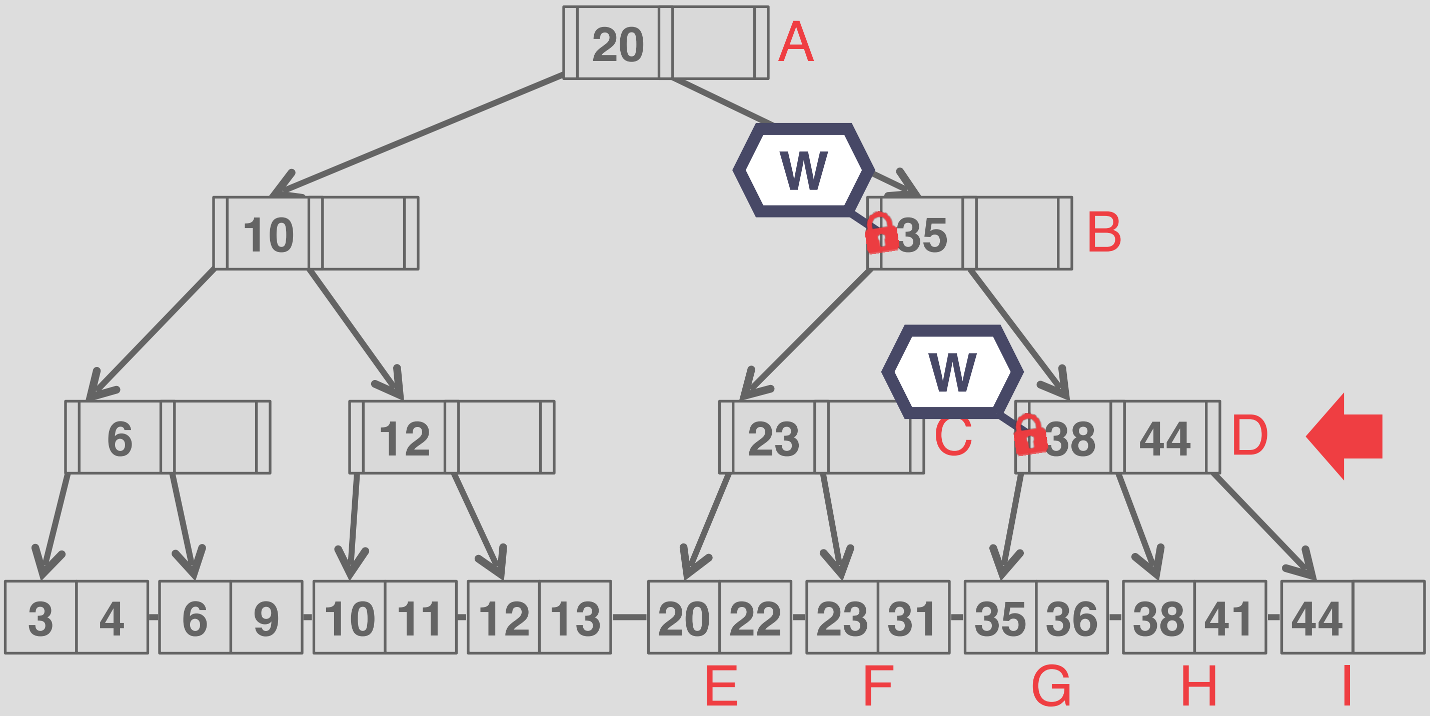
EXAMPLE #3 – INSERT 45



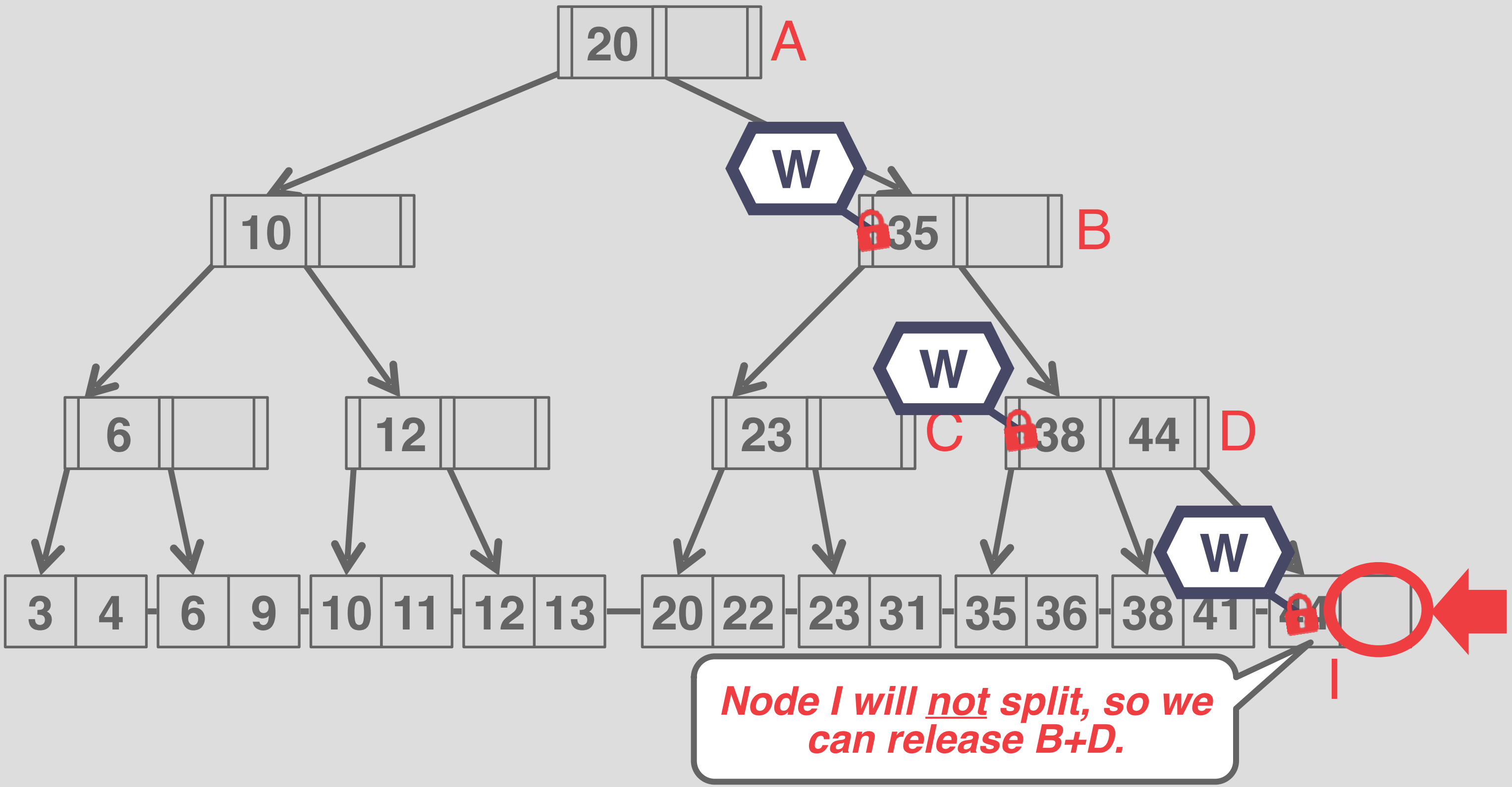
EXAMPLE #3 – INSERT 45



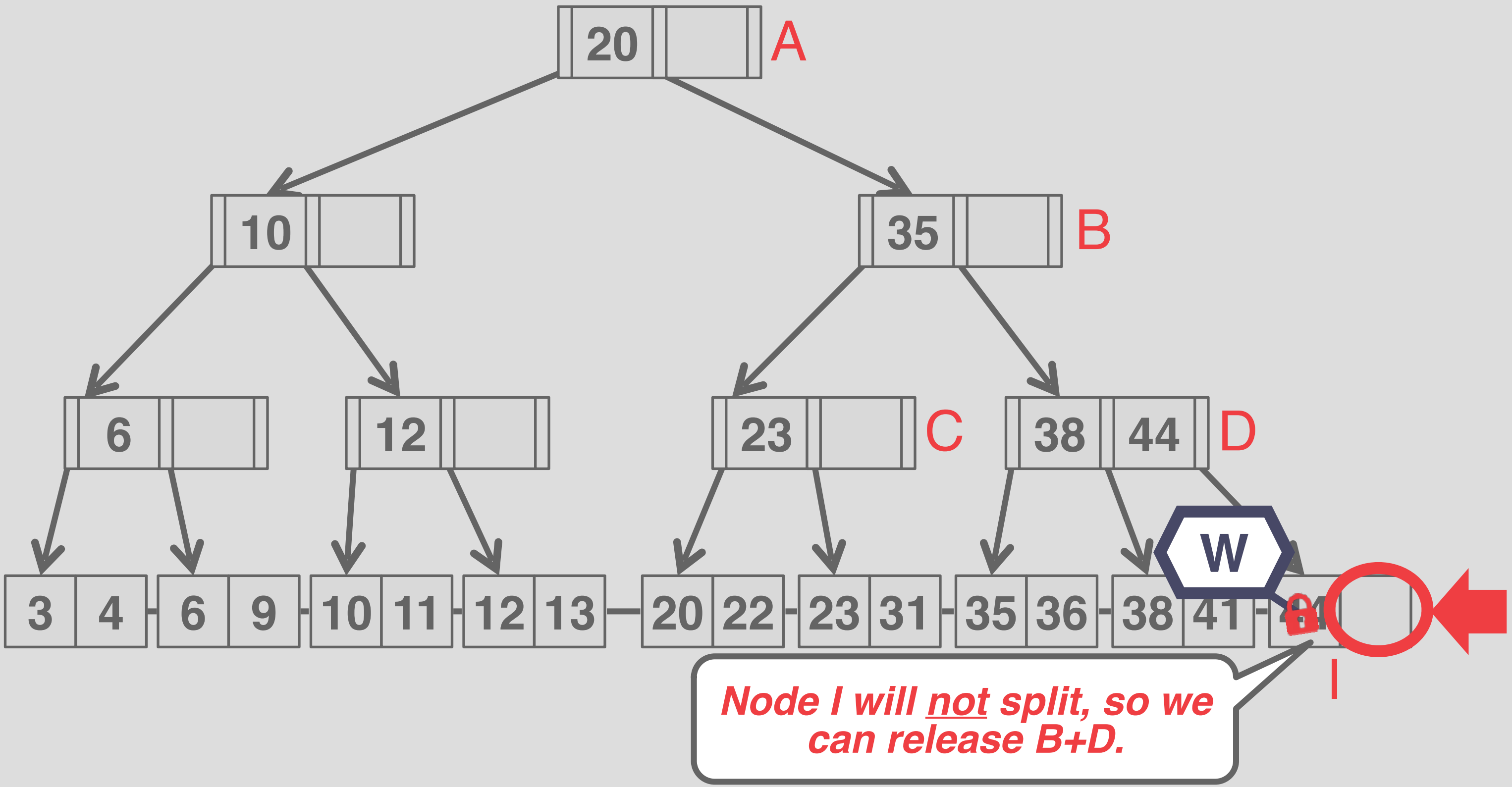
EXAMPLE #3 – INSERT 45



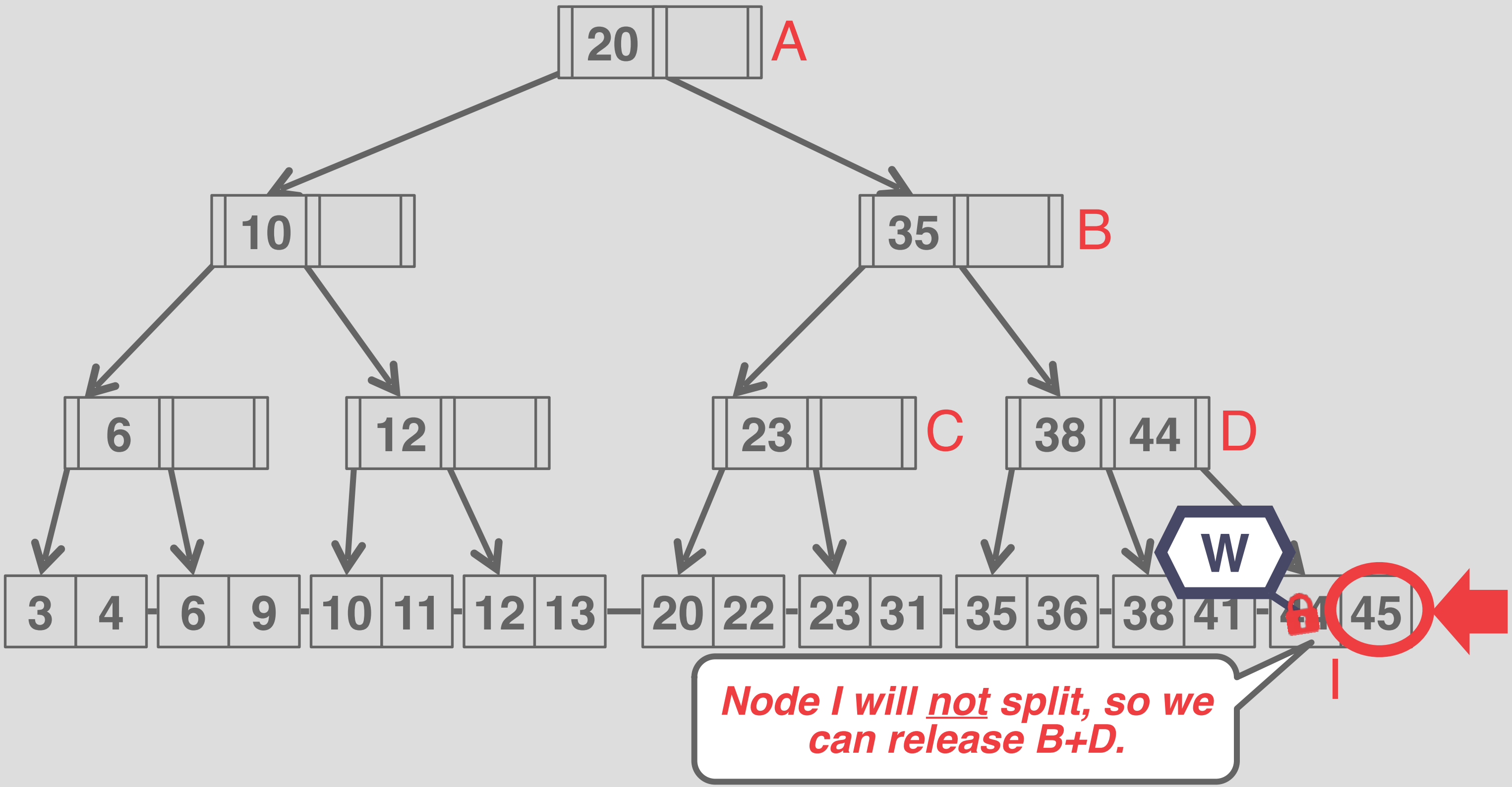
EXAMPLE #3 – INSERT 45



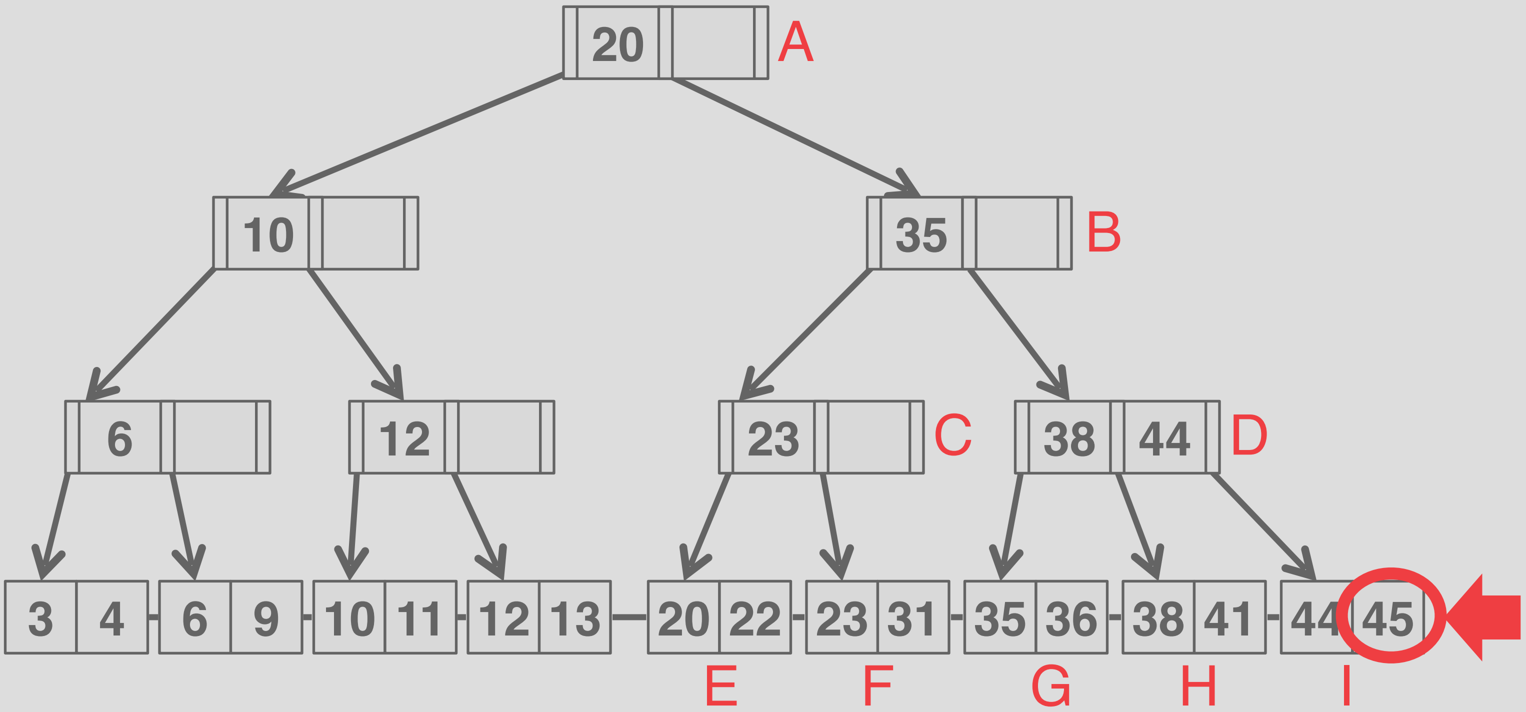
EXAMPLE #3 – INSERT 45



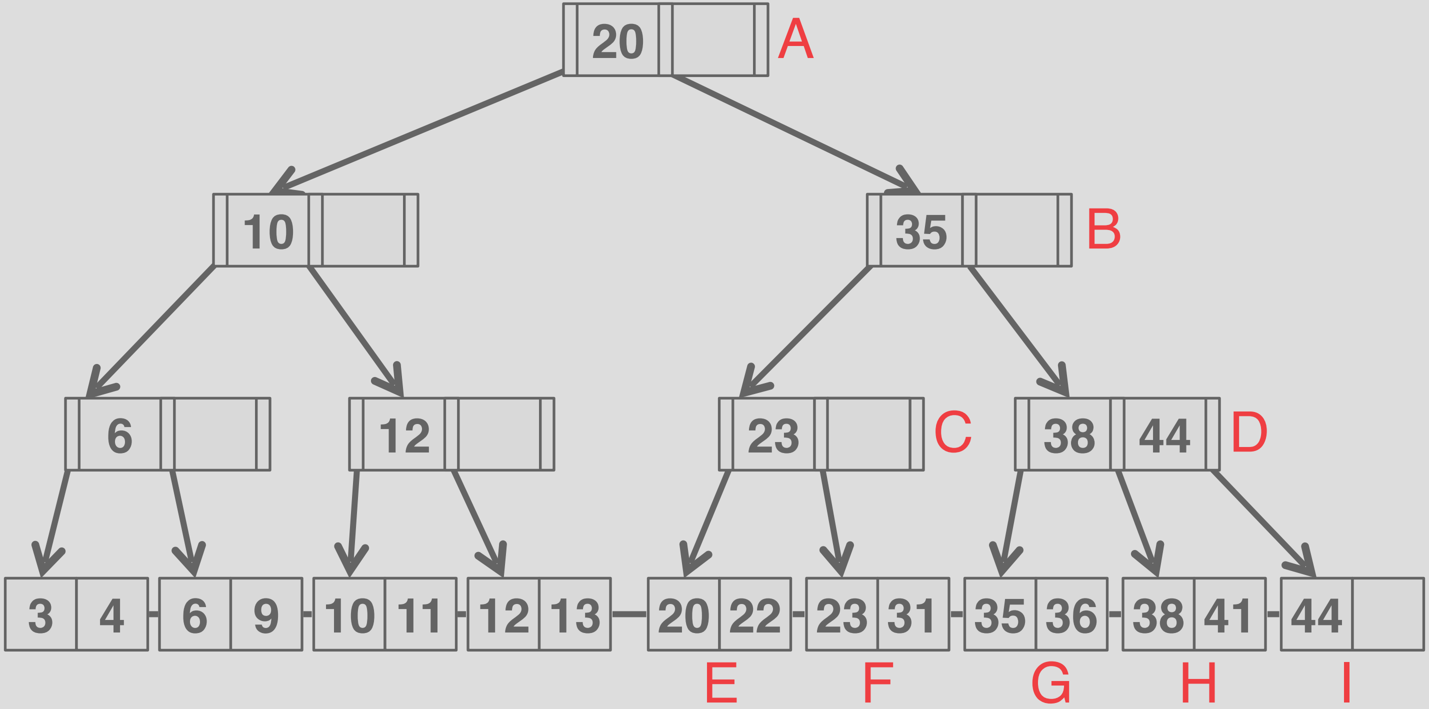
EXAMPLE #3 – INSERT 45



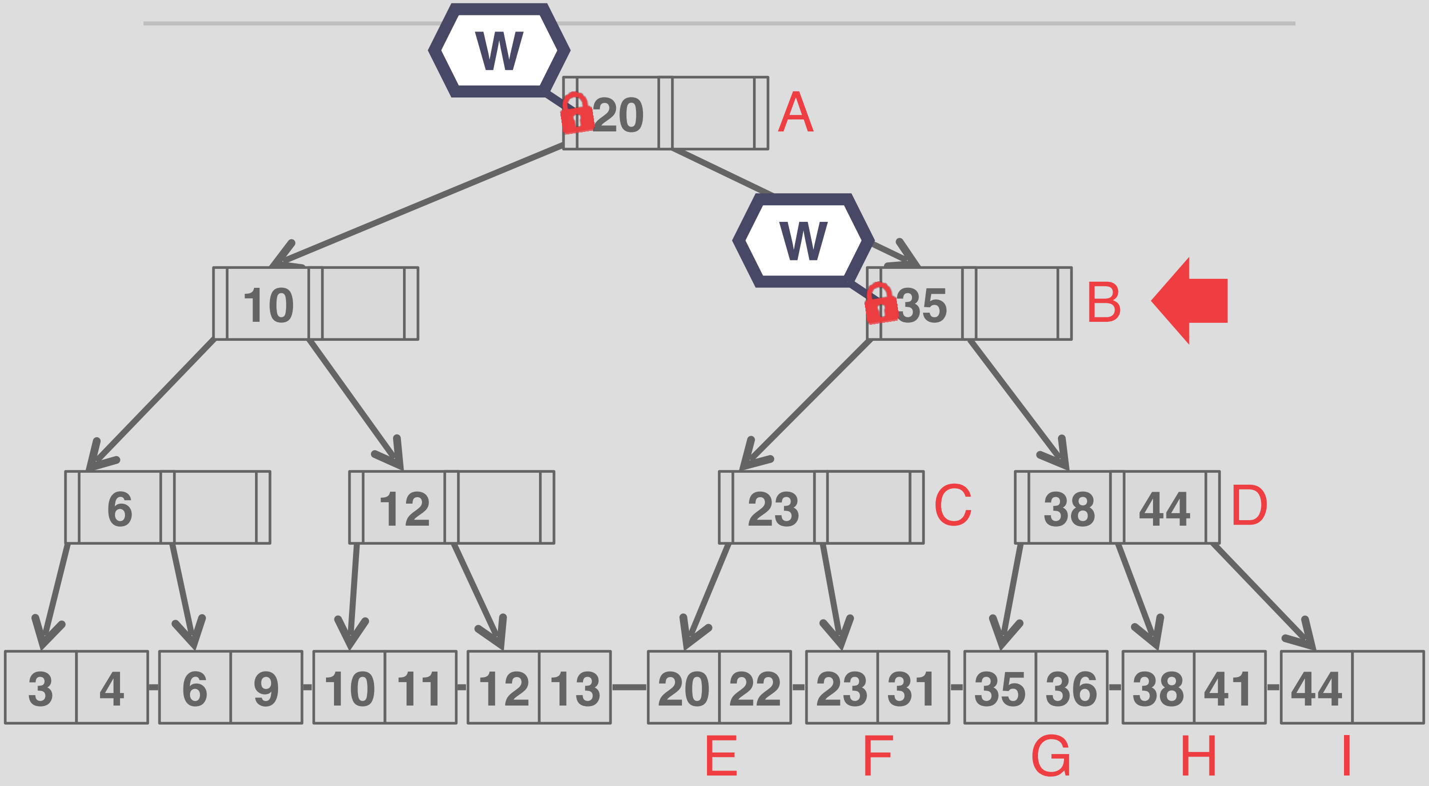
EXAMPLE #3 – INSERT 45



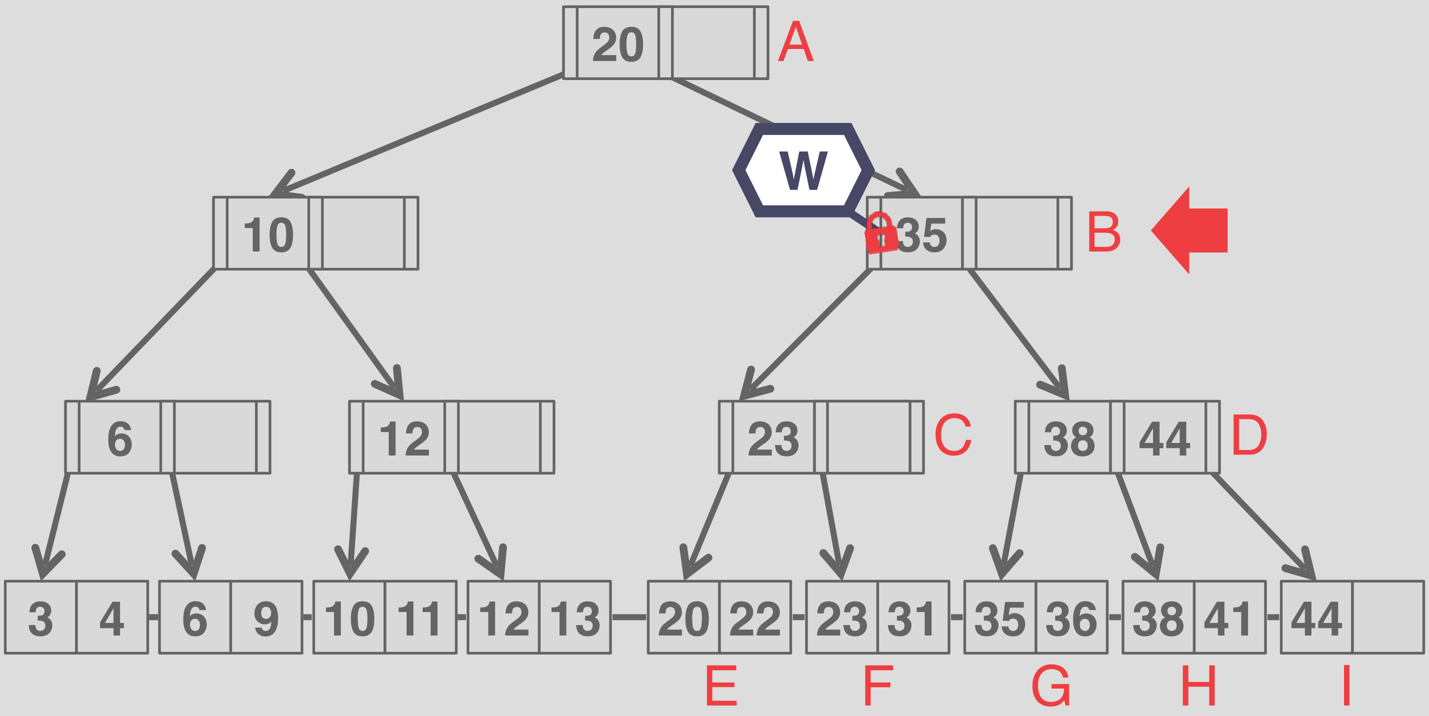
EXAMPLE #4 – INSERT 25



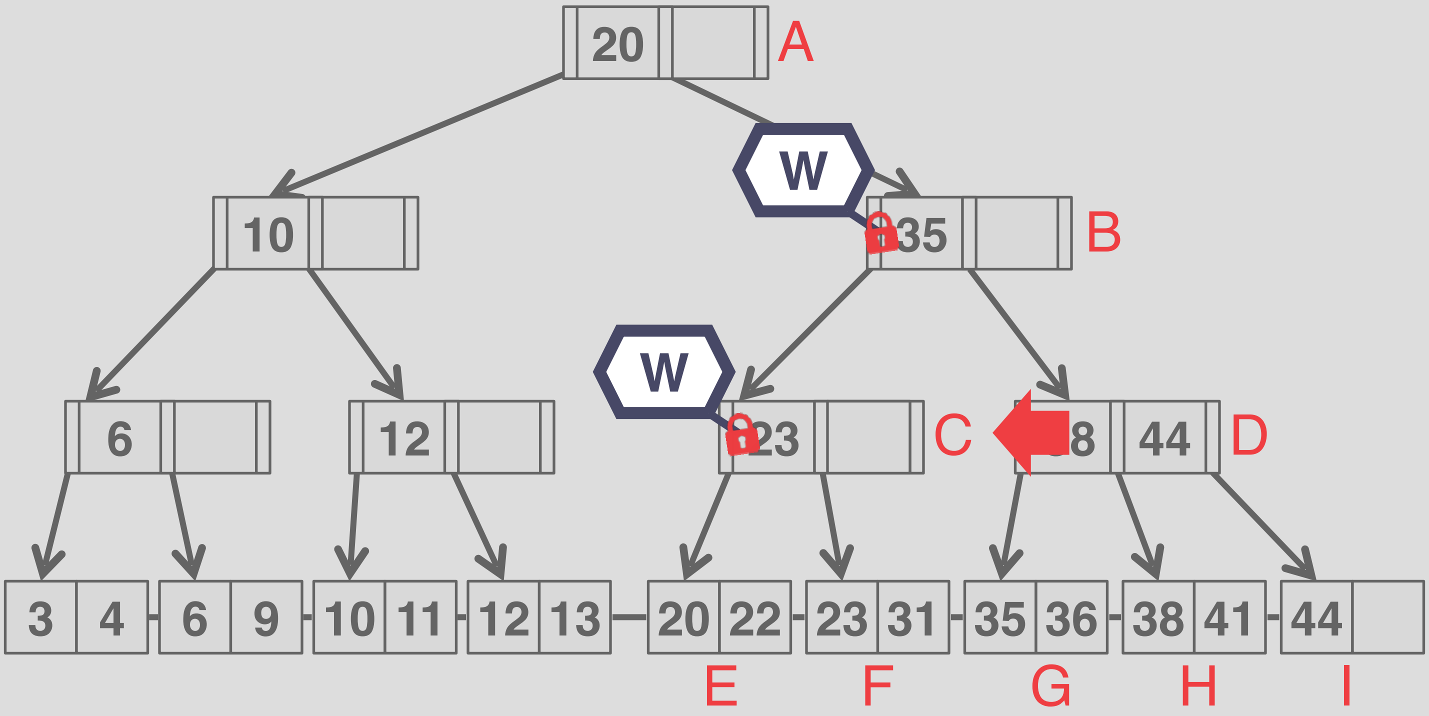
EXAMPLE #4 – INSERT 25



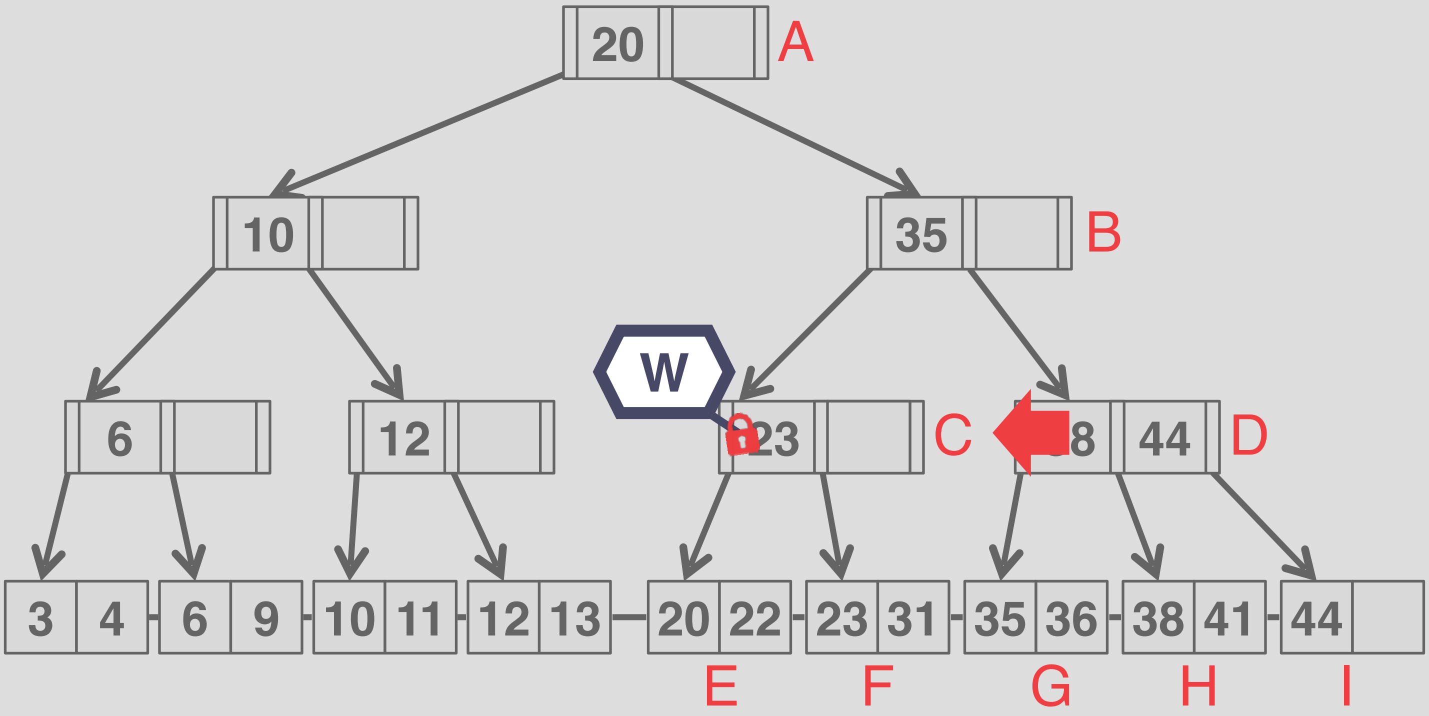
EXAMPLE #4 – INSERT 25



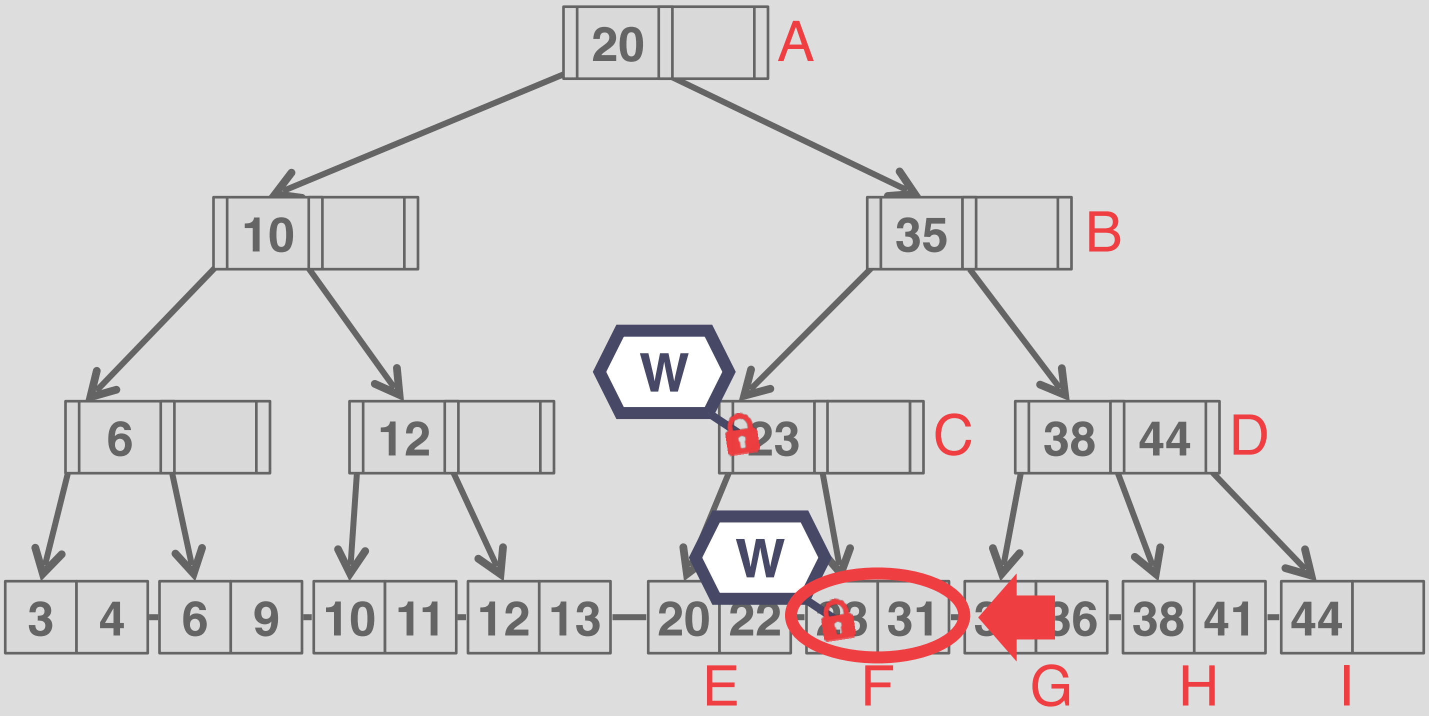
EXAMPLE #4 – INSERT 25



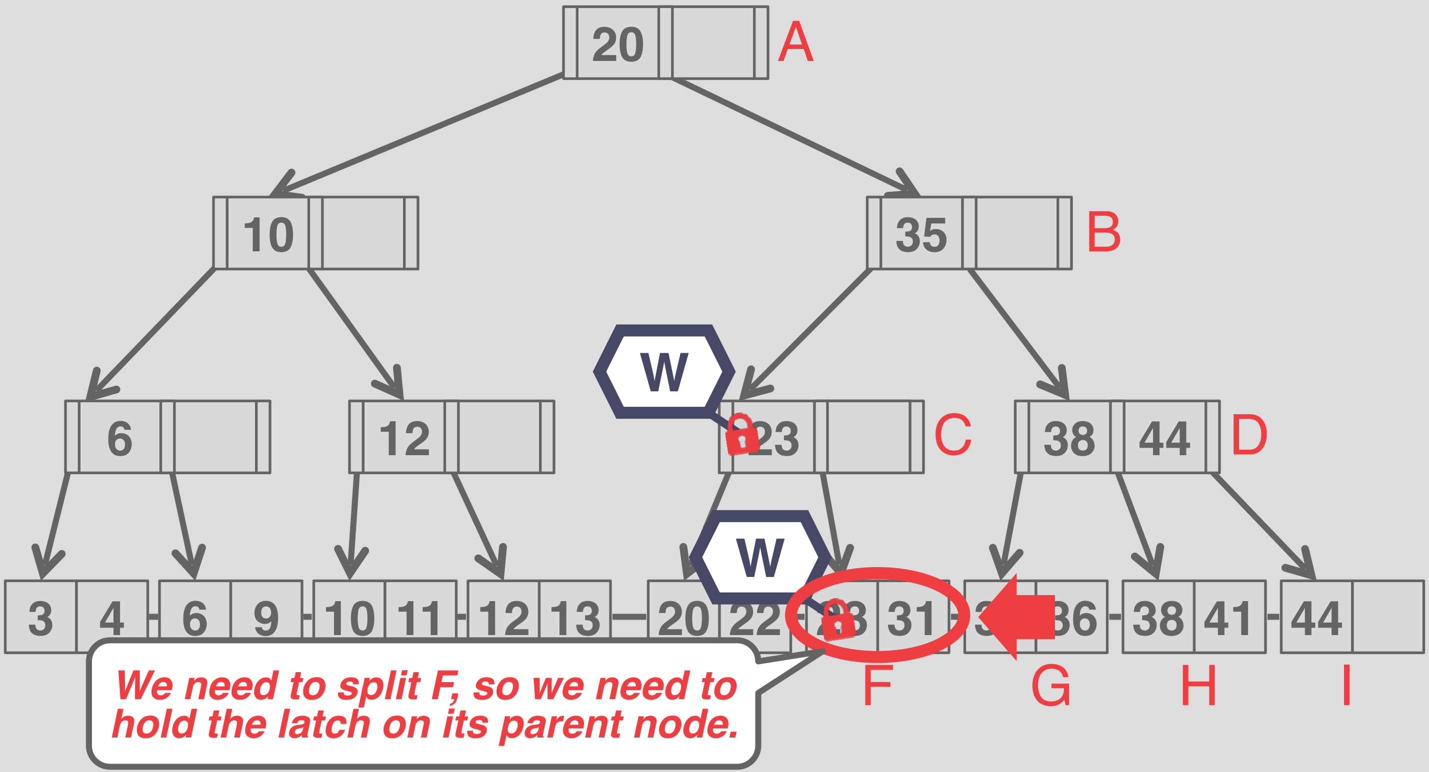
EXAMPLE #4 – INSERT 25



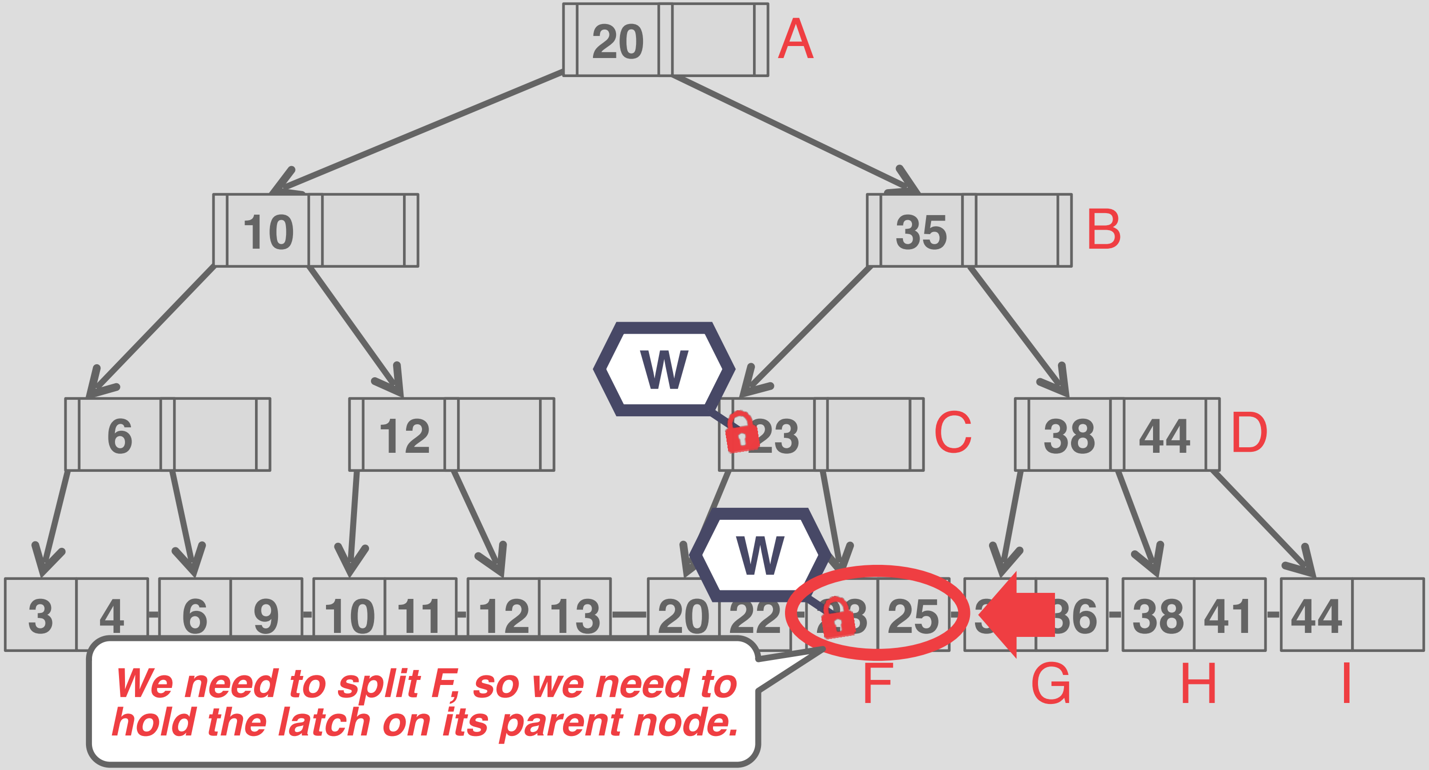
EXAMPLE #4 – INSERT 25



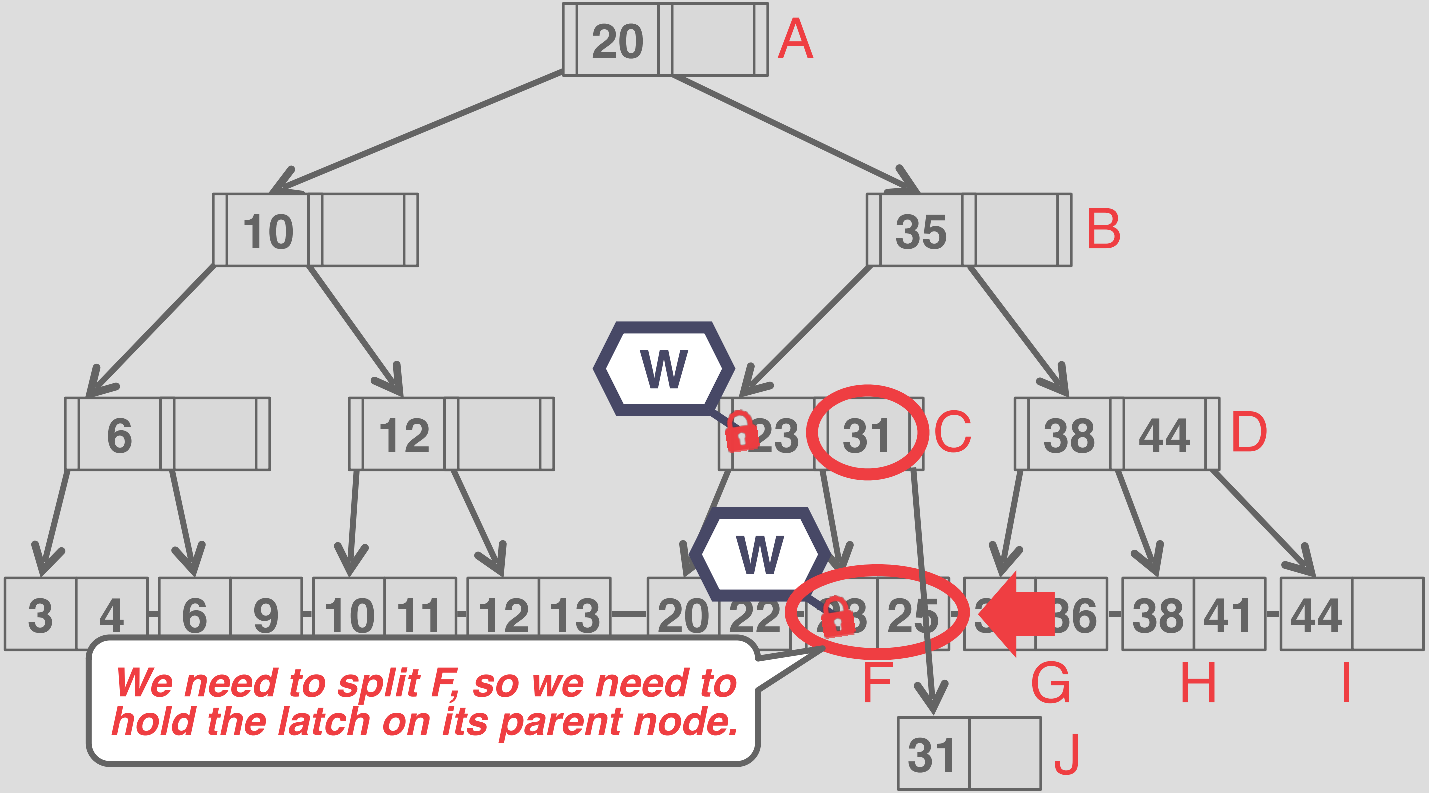
EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25



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

OBSERVATION

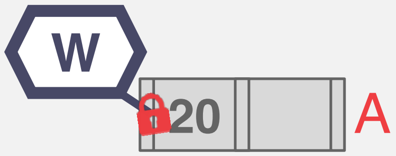
OBSERVATION

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

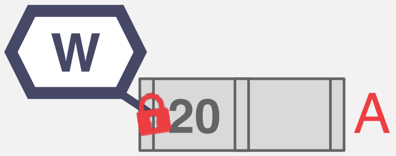
OBSERVATION

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

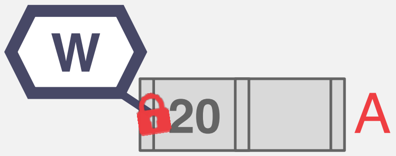
Delete 38



Insert 45

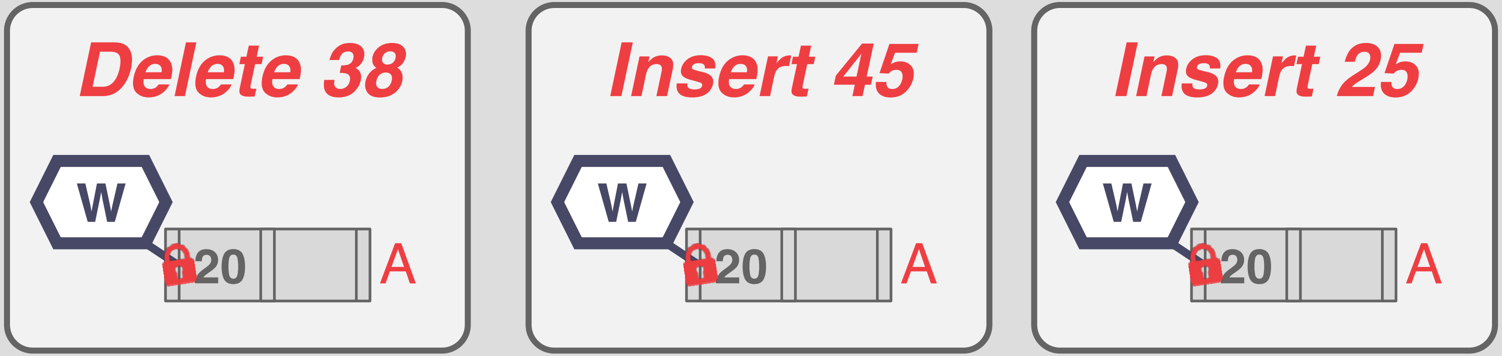


Insert 25



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

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.


Concurrency of Operations on B-Trees

BETTER LATCHING ALGORITHM

Search: Same as before.

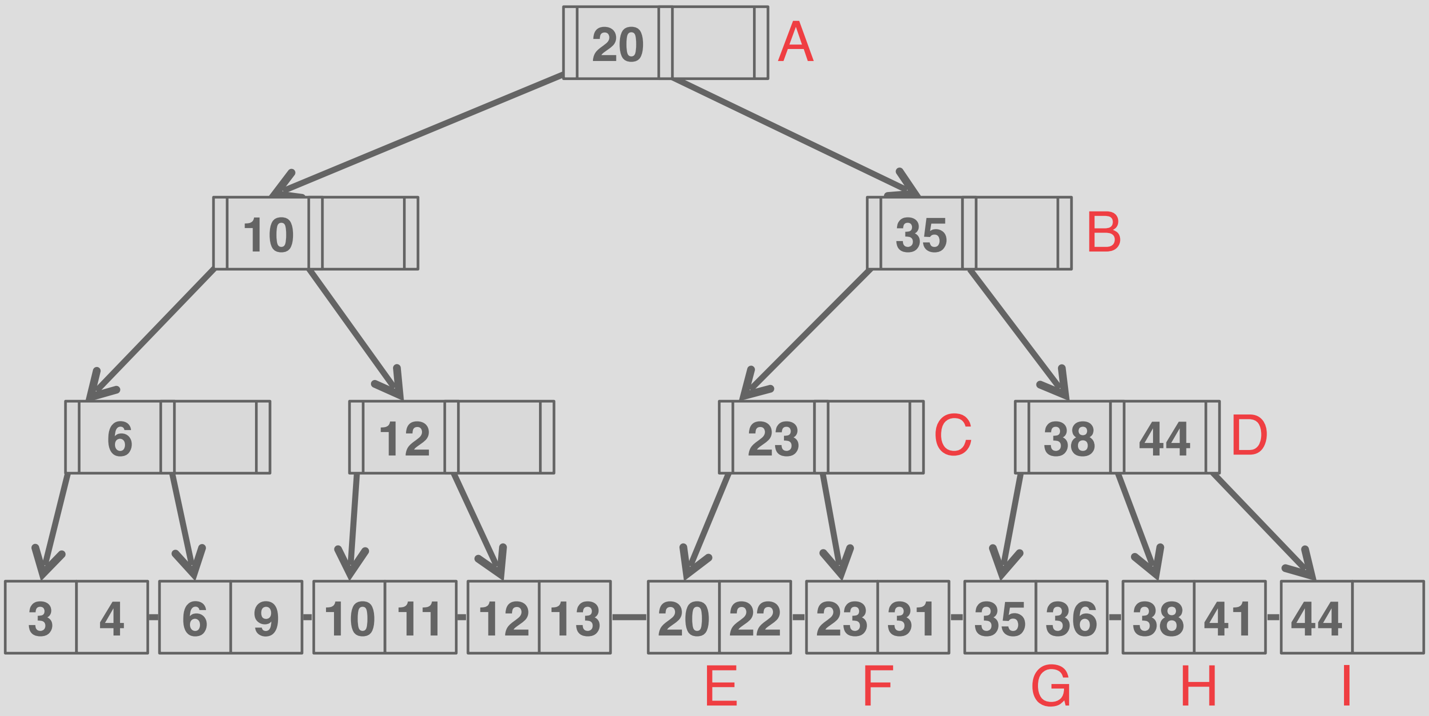
Insert/Delete:

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

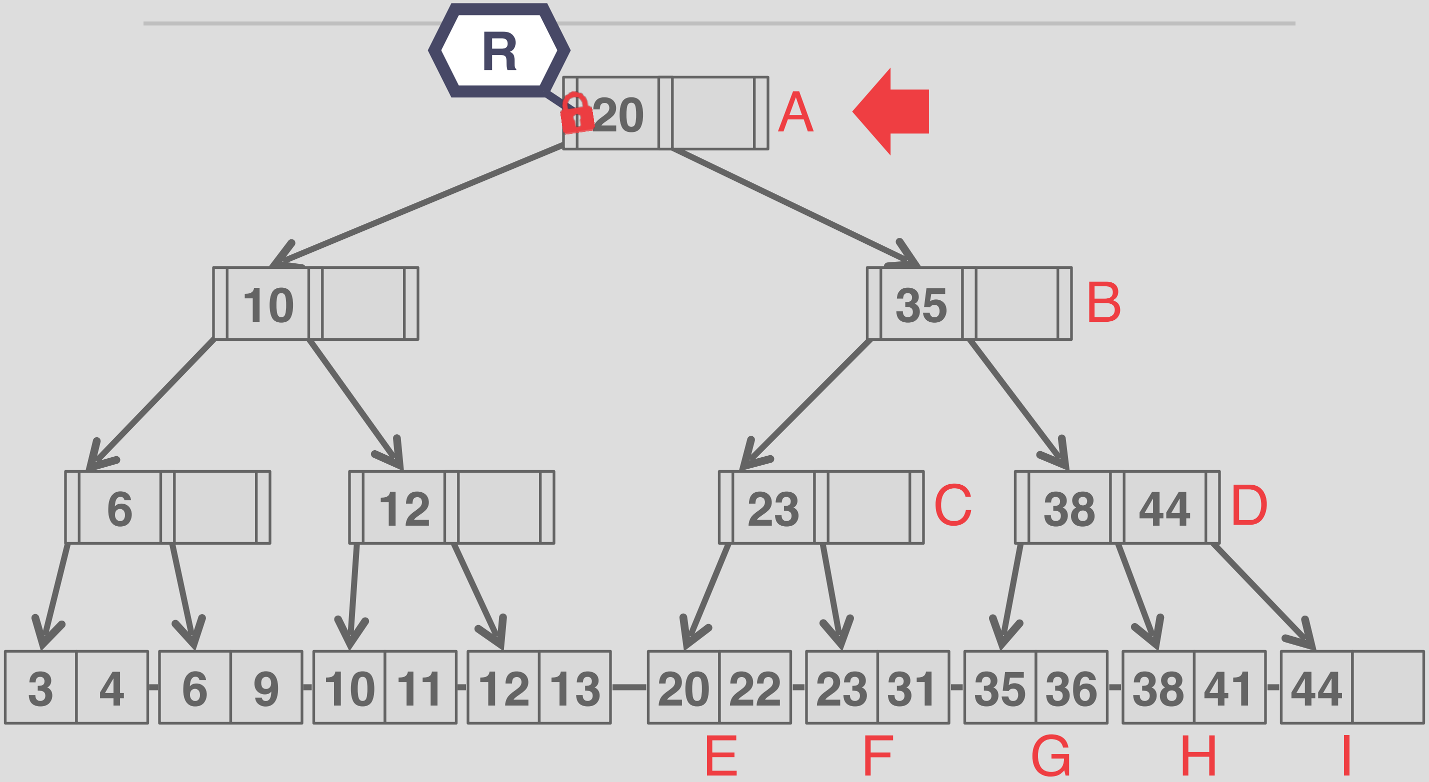
→ If leaf is not safe, release all latches, and restart thread using previous insert/delete protocol with write latches.

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

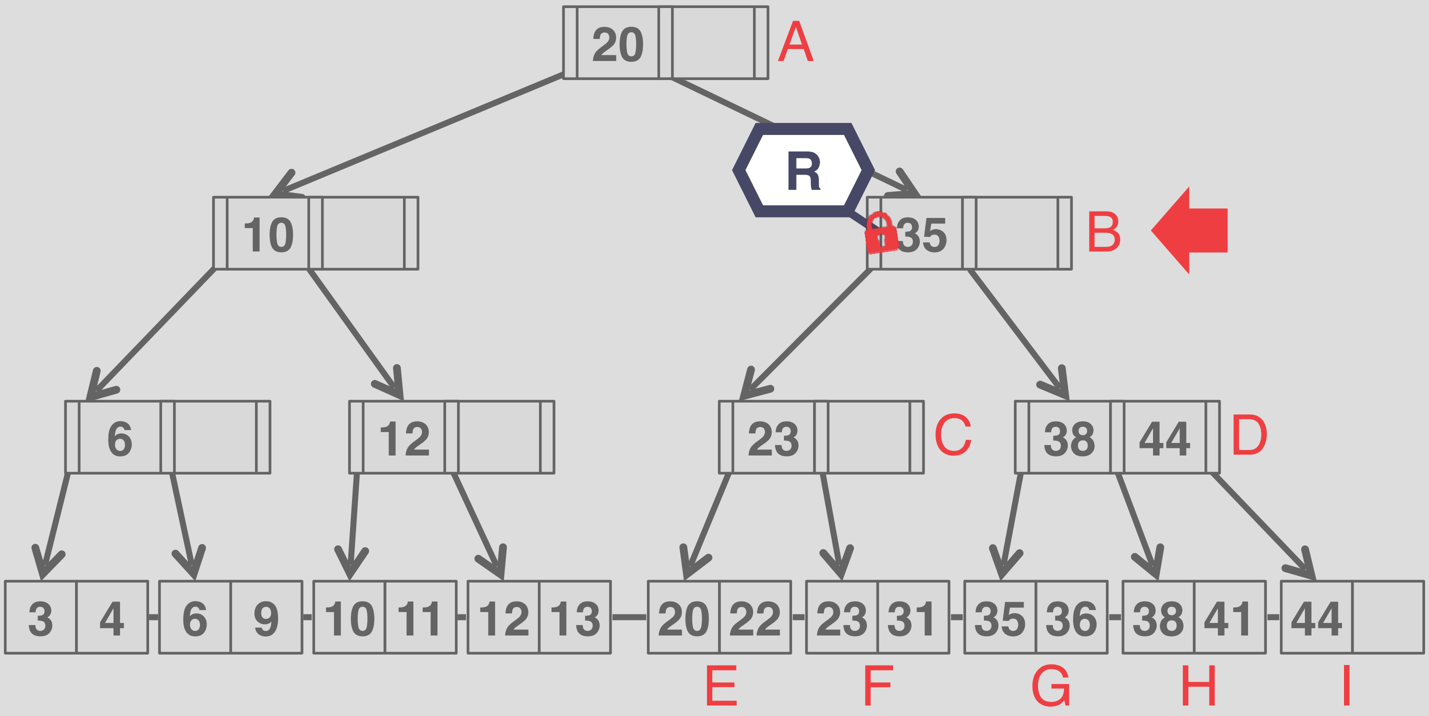
EXAMPLE #2 – DELETE 38



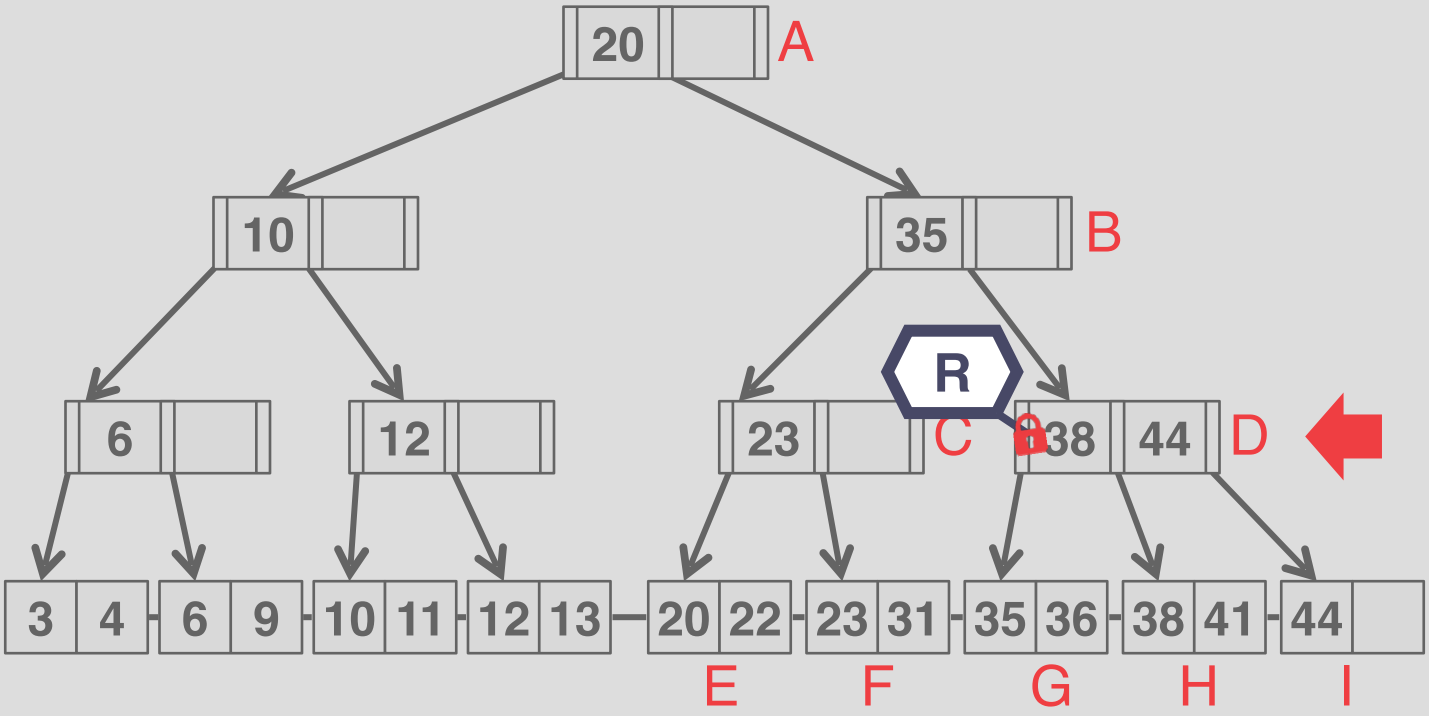
EXAMPLE #2 – DELETE 38



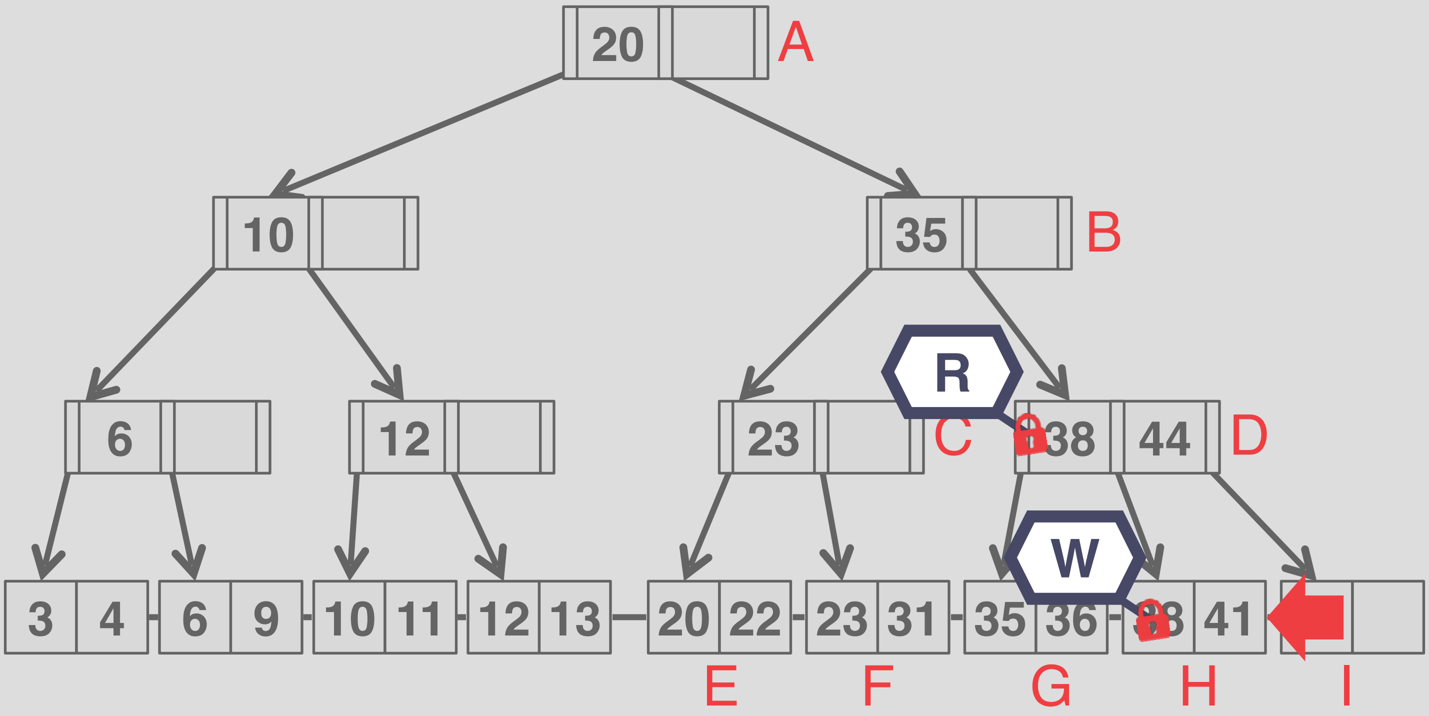
EXAMPLE #2 – DELETE 38



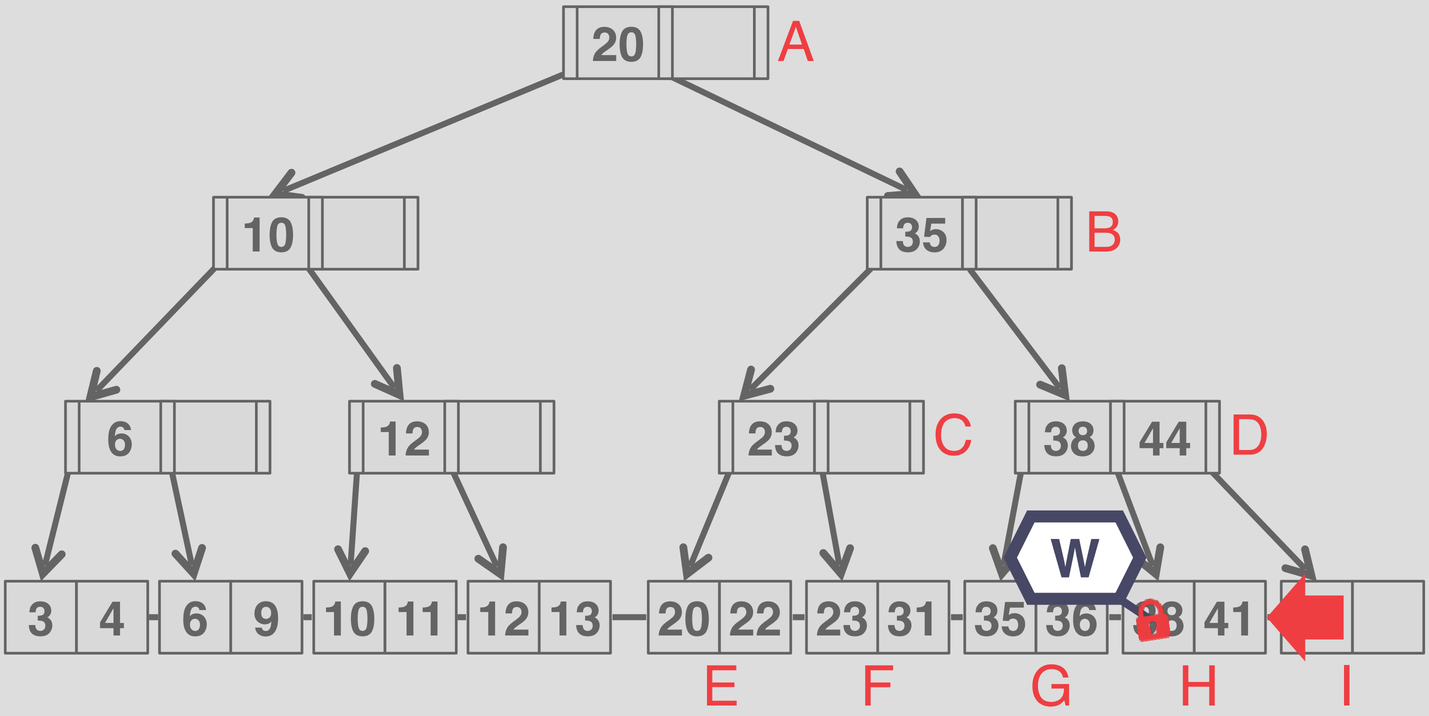
EXAMPLE #2 – DELETE 38



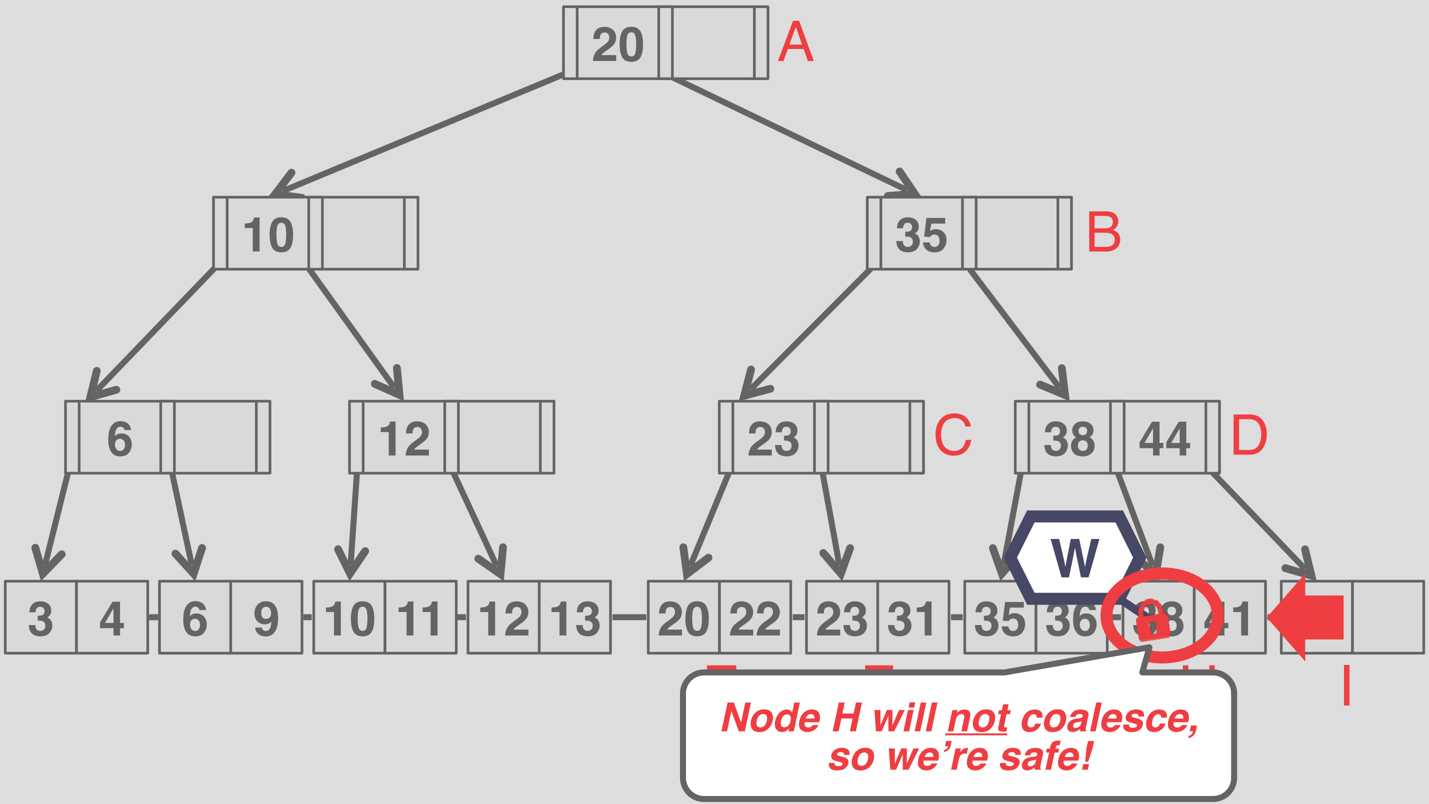
EXAMPLE #2 – DELETE 38



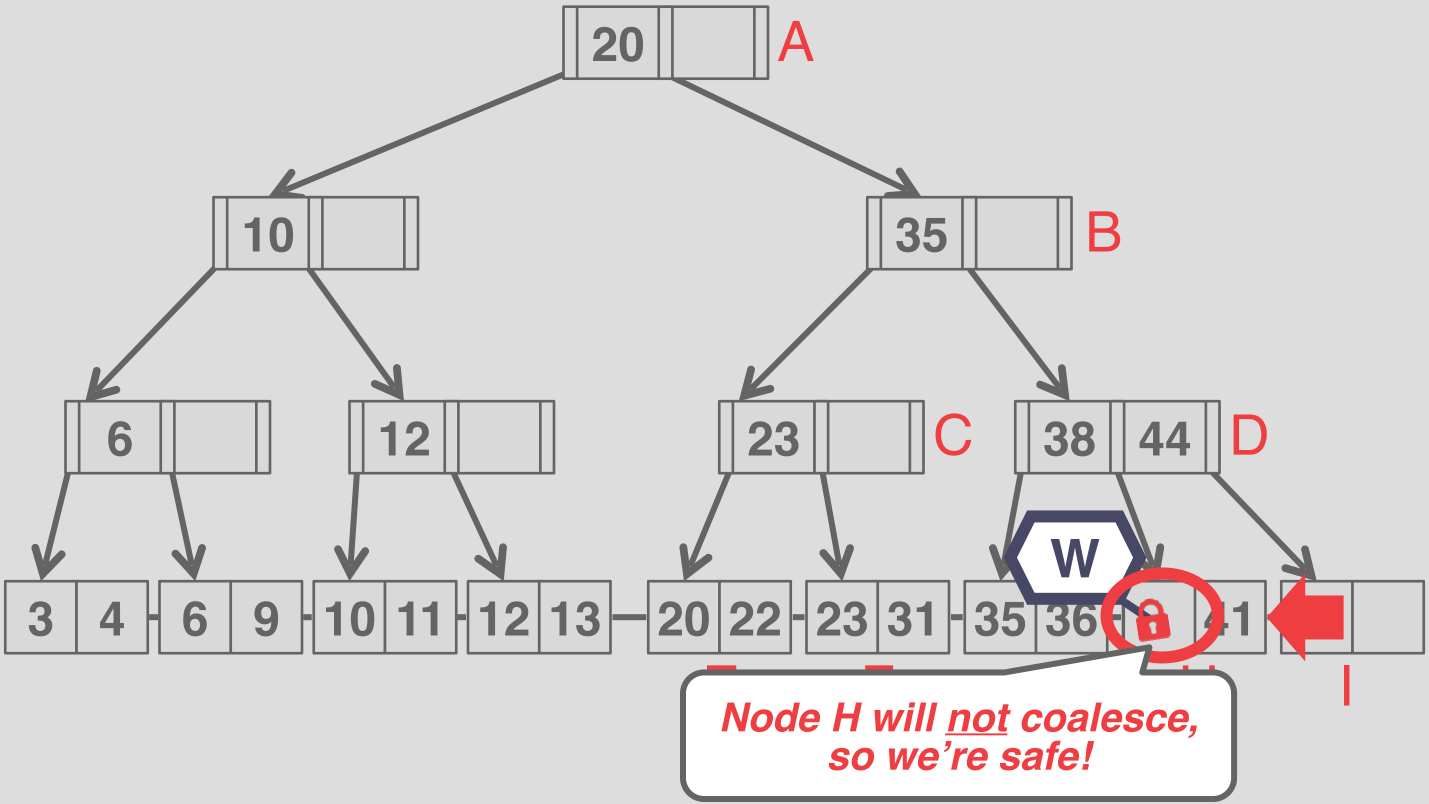
EXAMPLE #2 – DELETE 38



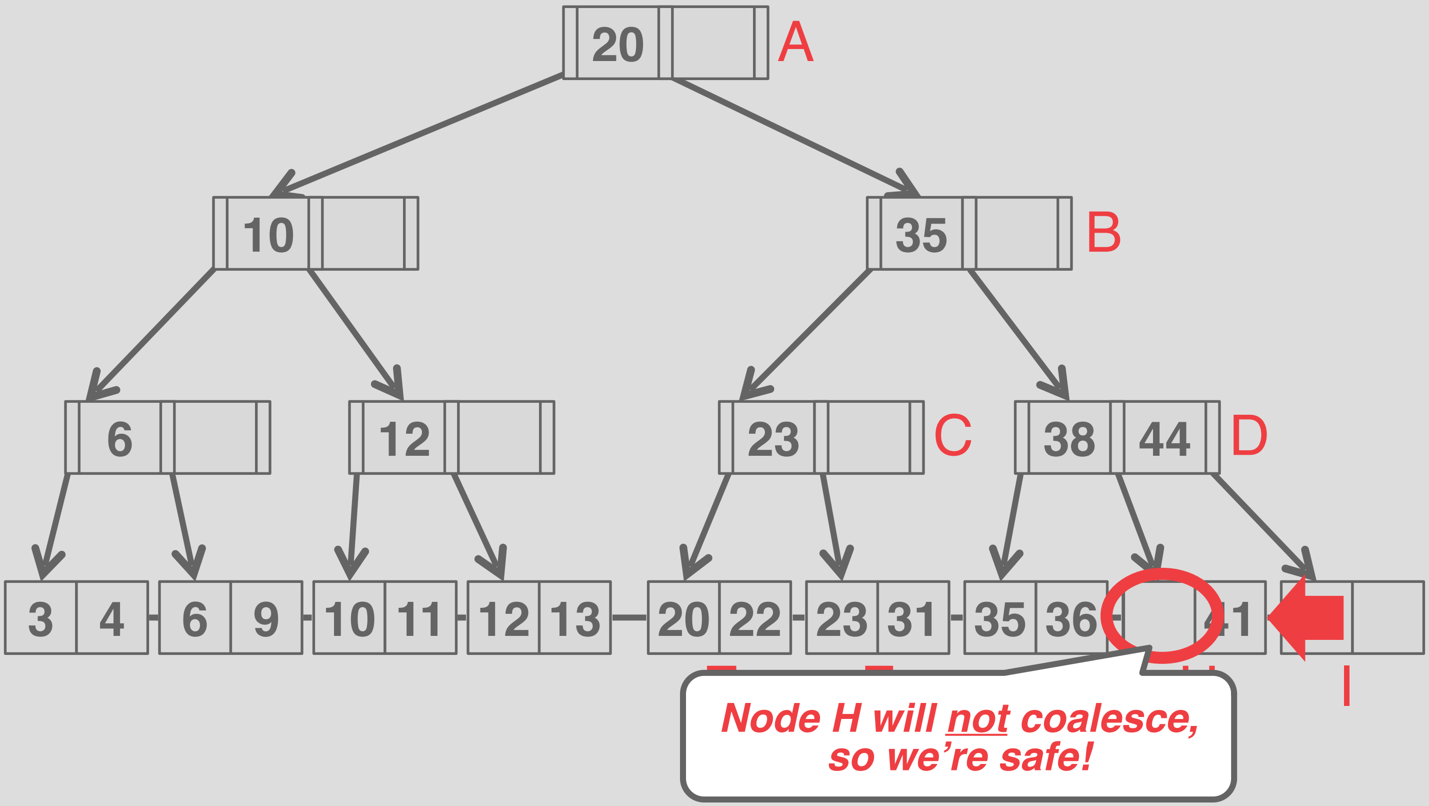
EXAMPLE #2 – DELETE 38



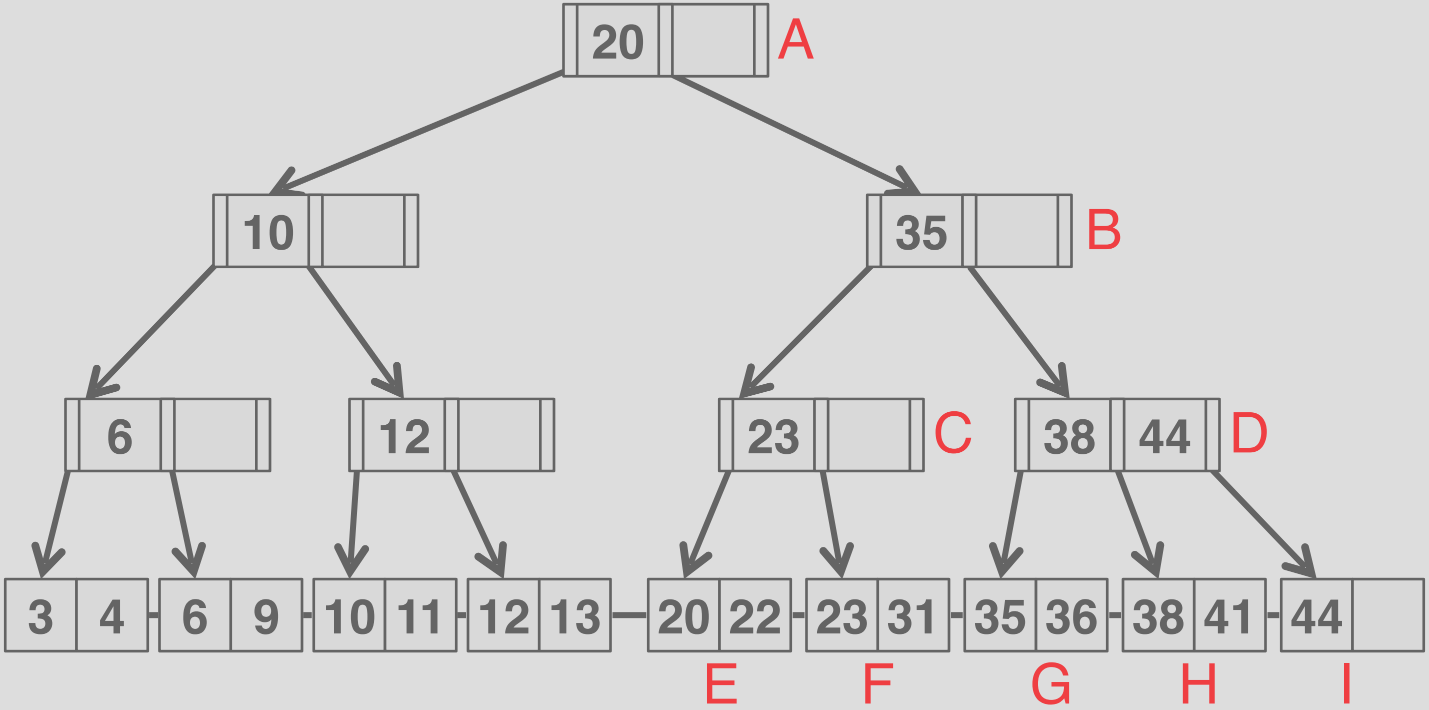
EXAMPLE #2 – DELETE 38



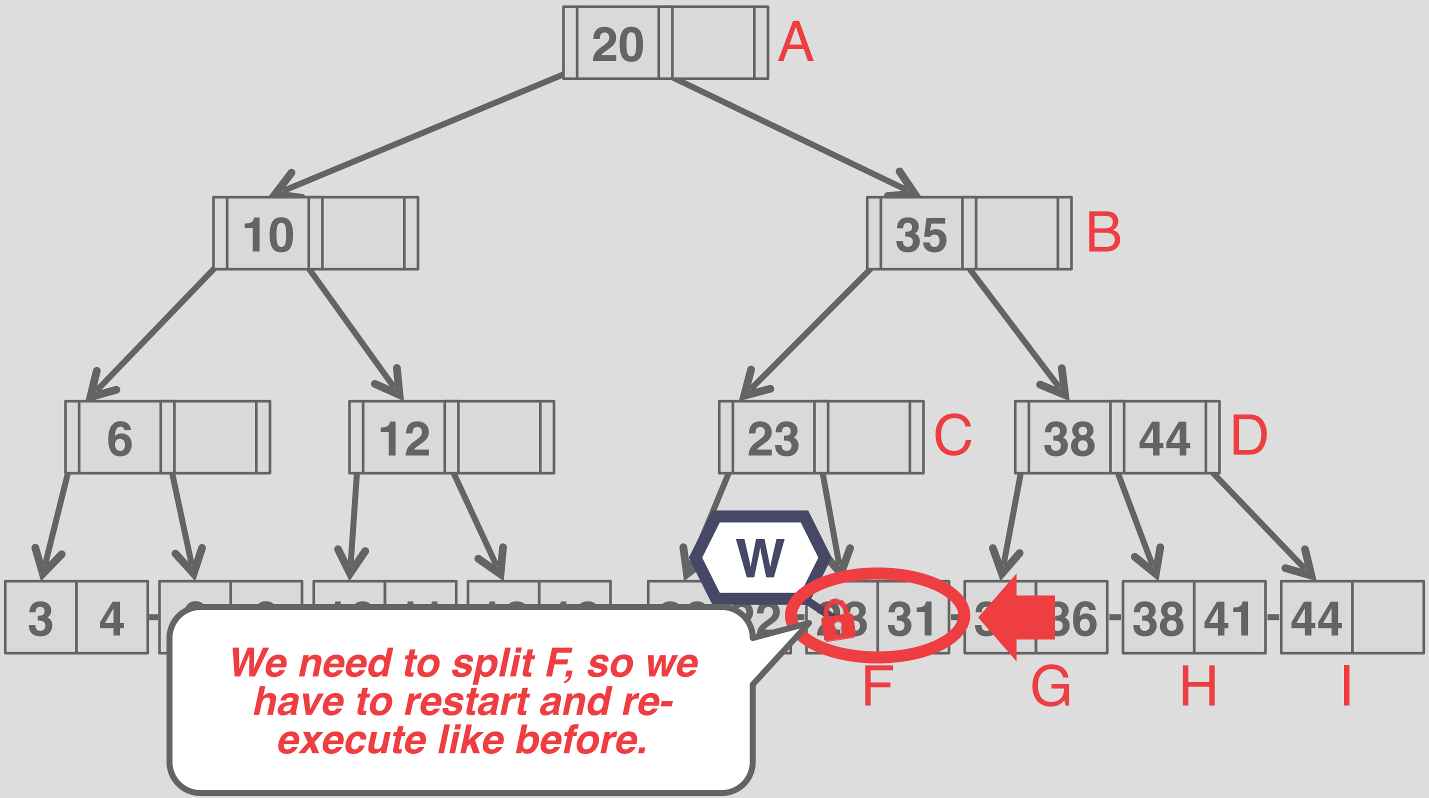
EXAMPLE #2 – DELETE 38



EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25



OBSERVATION

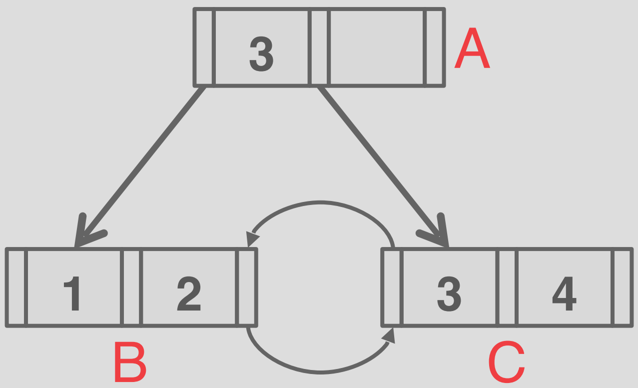
The threads in all the examples so far have acquired latches in a "top-down" manner.

→ 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.

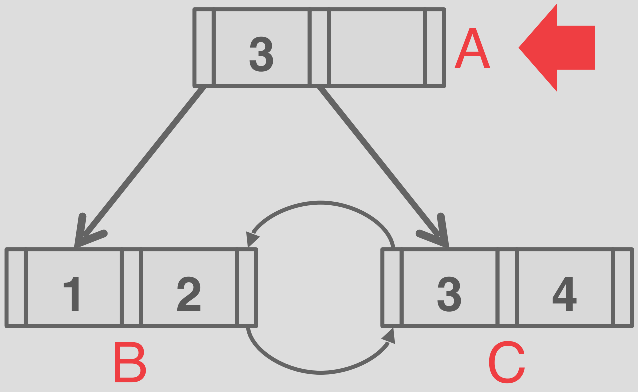
But what if threads want to move from one leaf node to another leaf node?

LEAF NODE SCAN EXAMPLE #1



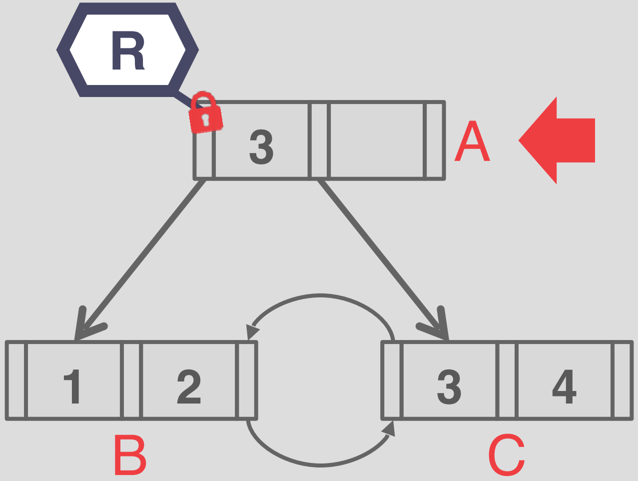
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



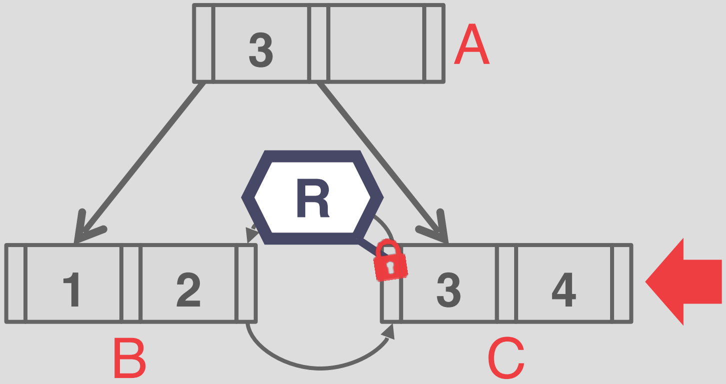
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



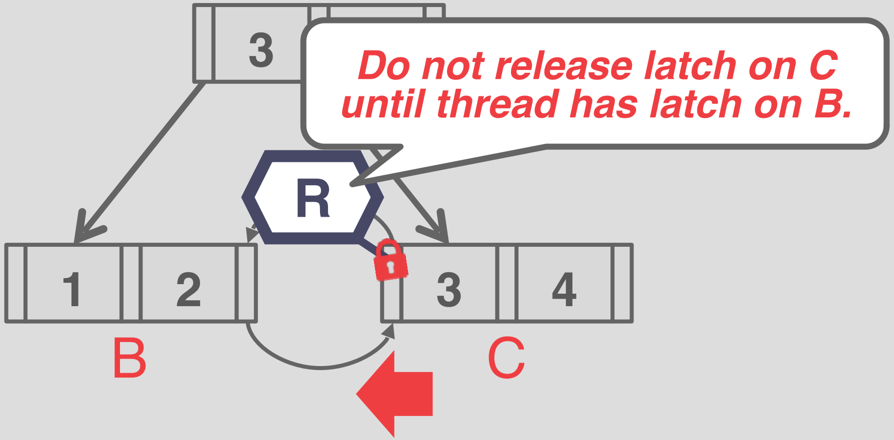
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



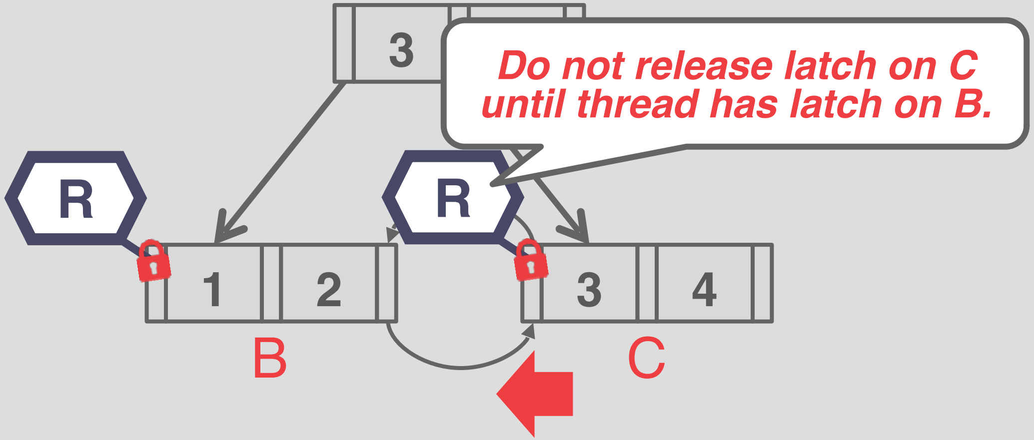
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



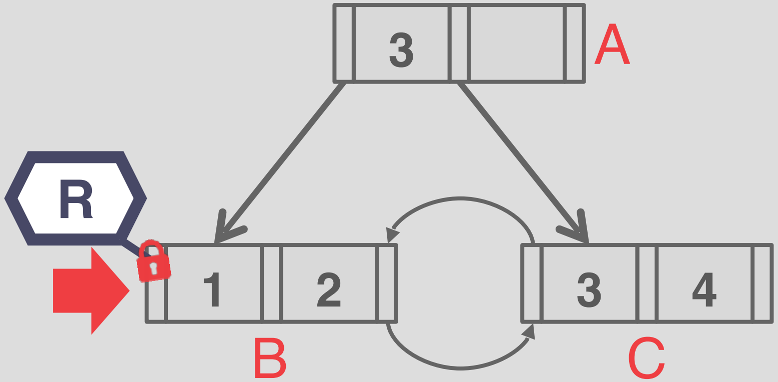
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



LEAF NODE SCAN EXAMPLE #1

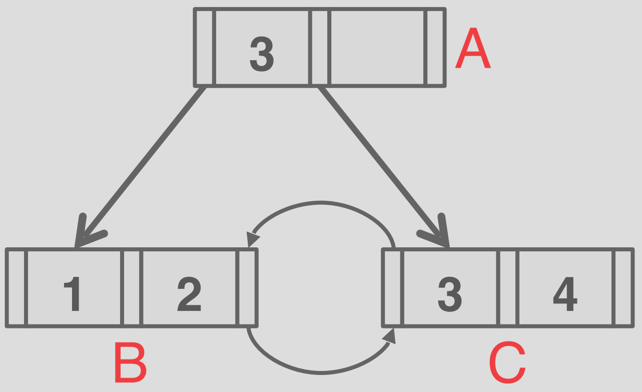
T_1: Find Keys < 4



LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

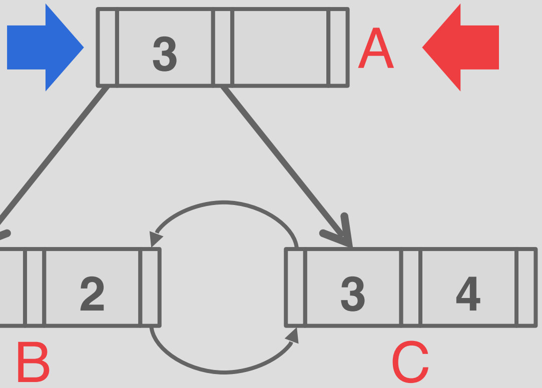
T_2: Find Keys > 1



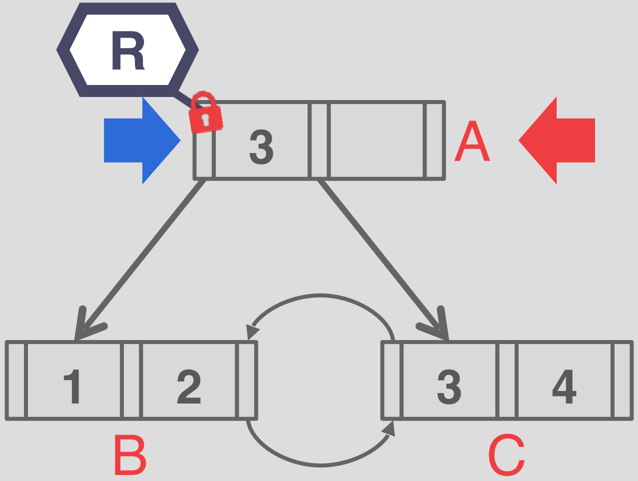
LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

T_2: Find Keys > 1



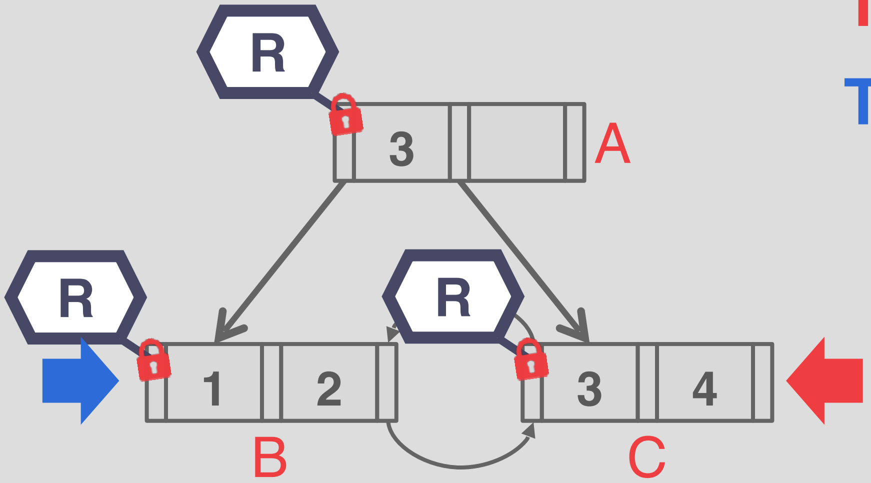
LEAF NODE SCAN EXAMPLE #2



T_1: Find Keys < 4

T_2: Find Keys > 1

LEAF NODE SCAN EXAMPLE #2



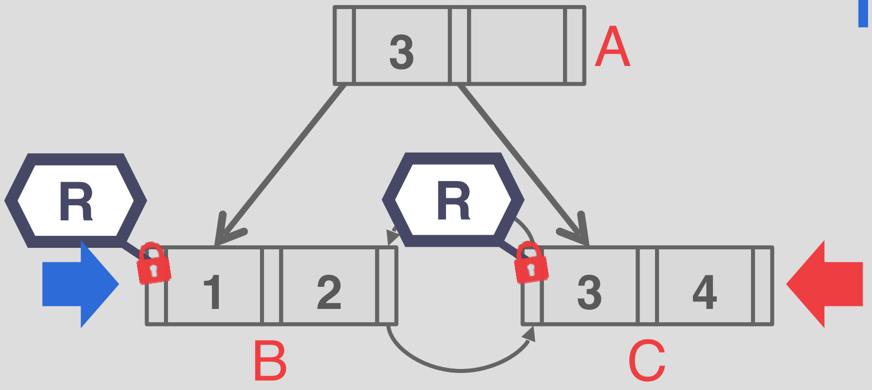
T_1: Find Keys < 4

T_2: Find Keys > 1

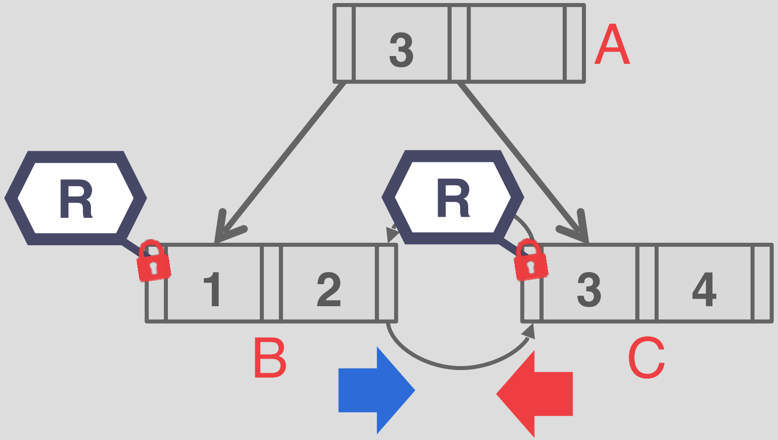
LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #2



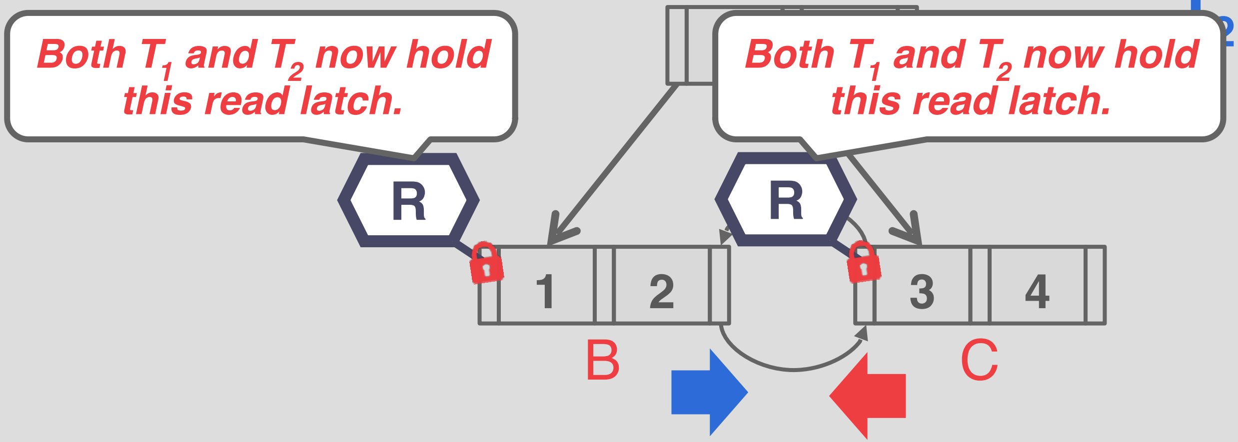
T_1: Find Keys < 4

T_2: Find Keys > 1

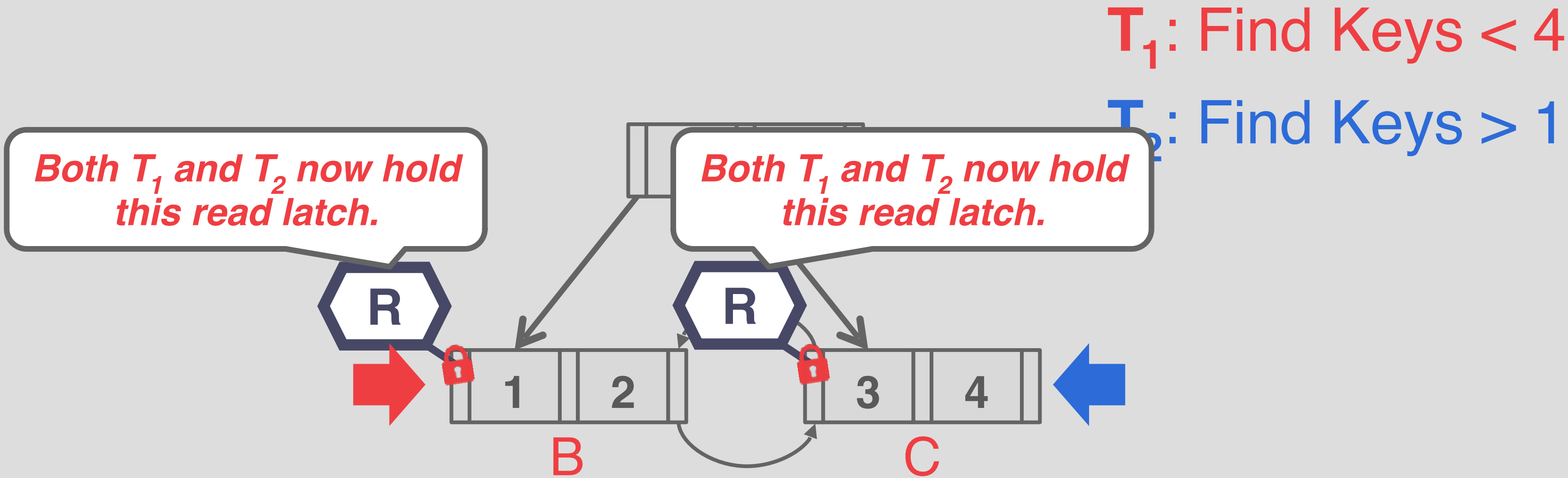
LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

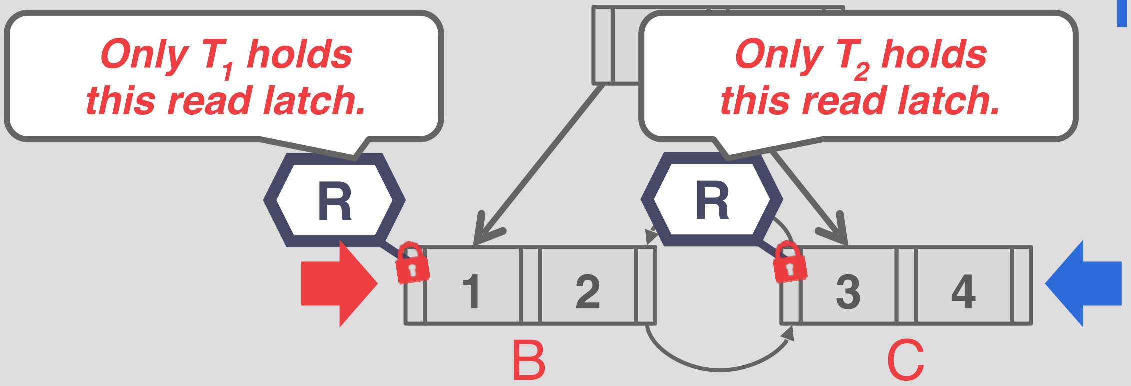
T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #2



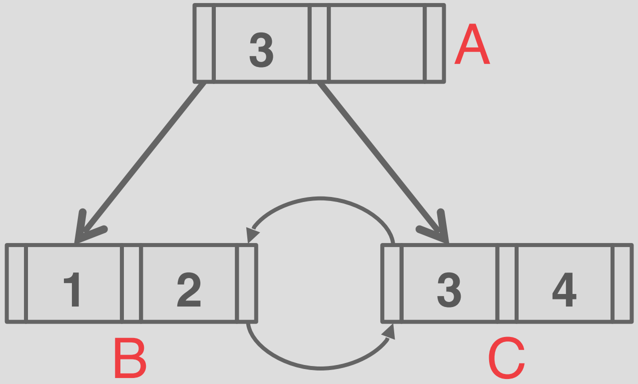
LEAF NODE SCAN EXAMPLE #2



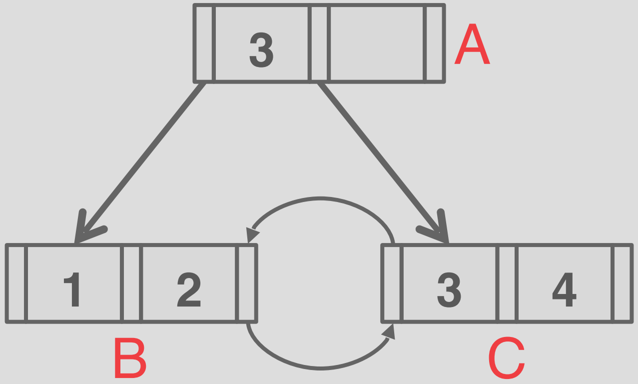
T_1: Find Keys < 4

T_2: Find Keys > 1

LEAF NODE SCAN EXAMPLE #3



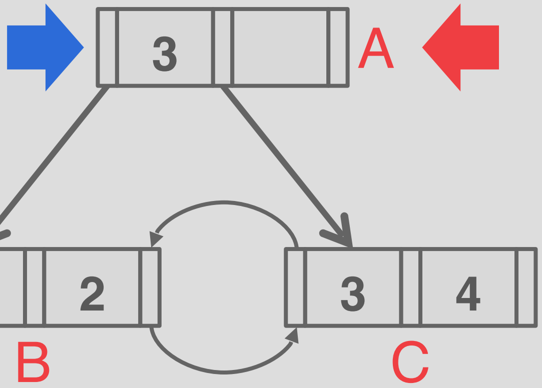
LEAF NODE SCAN EXAMPLE #3



T_1: Delete 4

T_2: Find Keys > 1

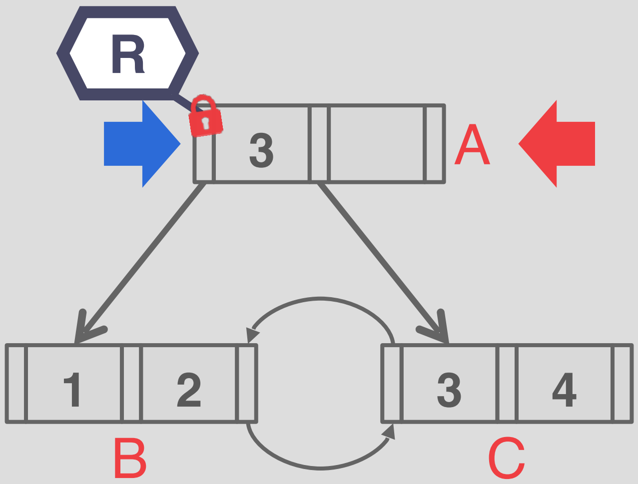
LEAF NODE SCAN EXAMPLE #3



T_1: Delete 4

T_2: Find Keys > 1

LEAF NODE SCAN EXAMPLE #3



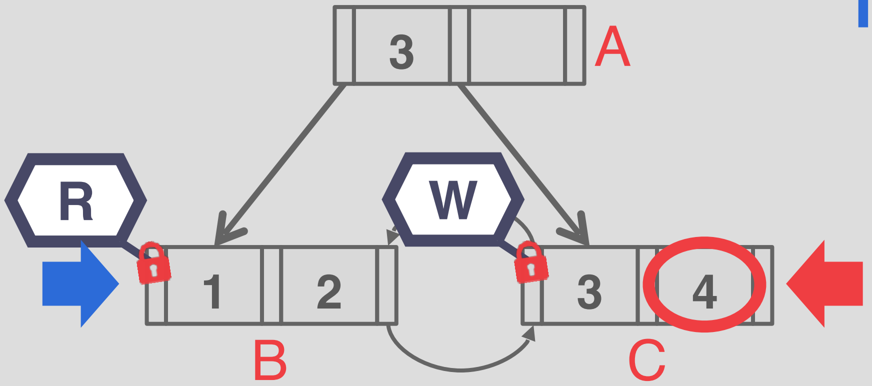
T_1: Delete 4

T_2: Find Keys > 1

LEAF NODE SCAN EXAMPLE #3

T_1: Delete 4

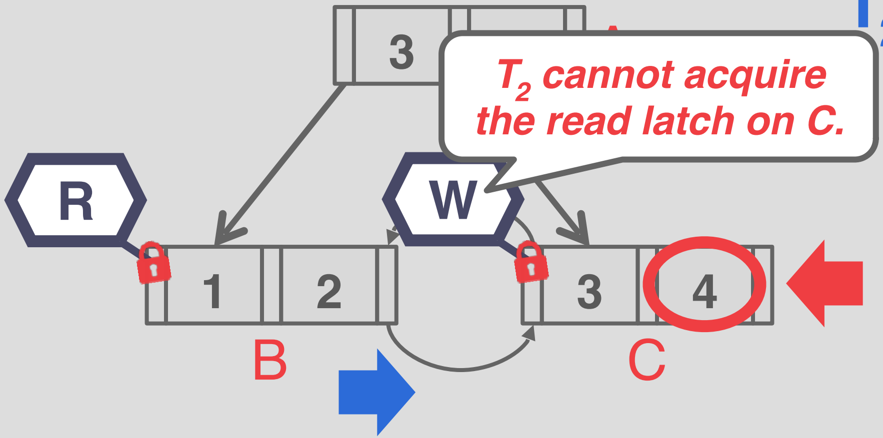
T_2: Find Keys > 1



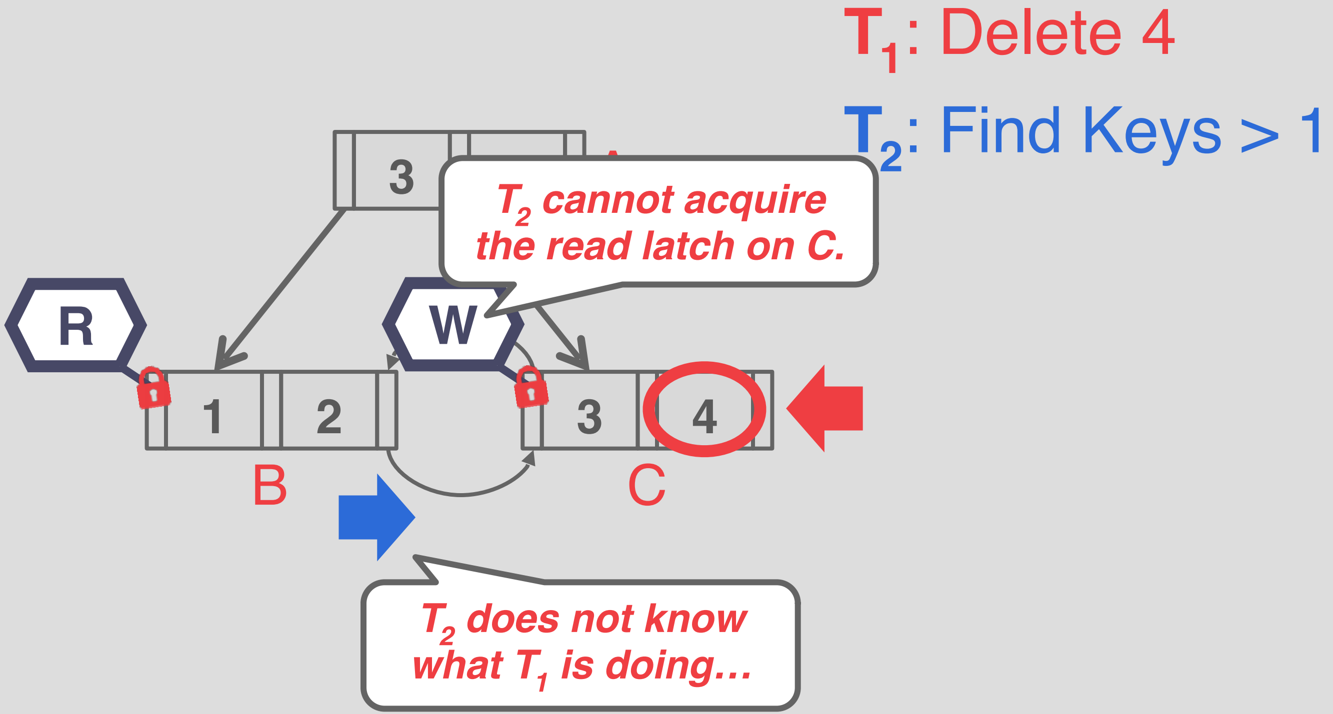
LEAF NODE SCAN EXAMPLE #3

T_1: Delete 4

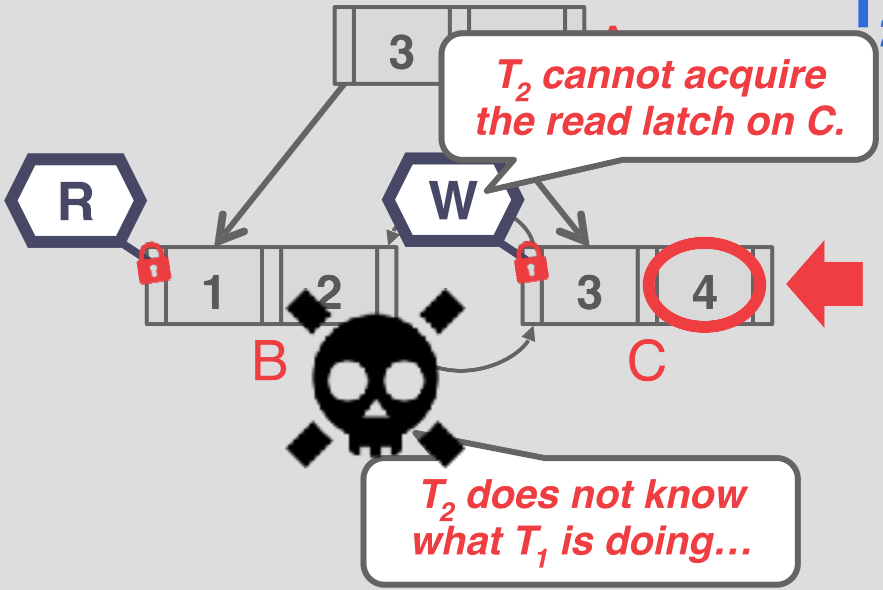
T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #3



LEAF NODE SCAN EXAMPLE #3



T_1: Delete 4

T_2: Find Keys > 1

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 will finally discuss how to execute queries…