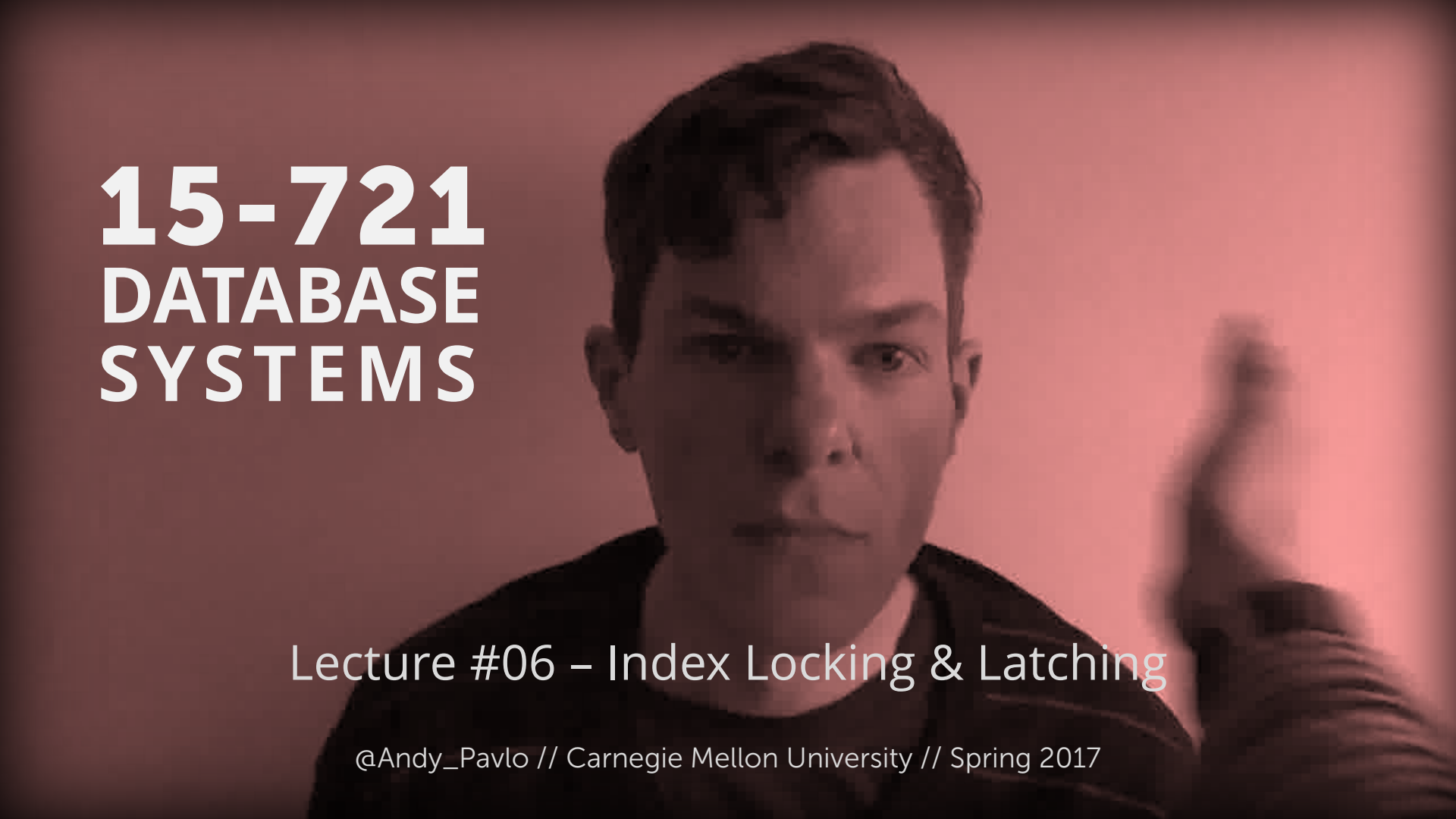


15-721 DATABASE SYSTEMS

Lecture #06 – Index Locking & Latching

@Andy_Pavlo // Carnegie Mellon University // Spring 2017

TODAY'S AGENDA

Index Locks vs. Latches
Latch Implementations
Latch Crabbing
Index Locking

DATABASE INDEX

A data structure that improves the speed of data retrieval operations on a table at the cost of additional writes and storage space.

Indexes are used to quickly locate data without having to search every row in a table every time a table is accessed.

DATA STRUCTURES

Order Preserving Indexes

→ A tree-like structure that maintains keys in some sorted order.
→ Supports all possible predicates with O(log n) searches.

Hashing Indexes

→ An associative array that maps a hash of the key to a particular record.
→ Only supports equality predicates with O(1) searches.

B-TREE VS. B+TREE

The original **B-tree** from 1972 stored keys + values in all nodes in the tree.

→ More memory efficient since each key only appears once in the tree.

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

→ Easier to manage concurrent index access when the values are only in the leaf nodes.

OBSERVATION

We already know how to use locks to protect objects in the database.

But we have to treat indexes differently because the physical structure can change as long as the logical contents are consistent.

SIMPLE EXAMPLE

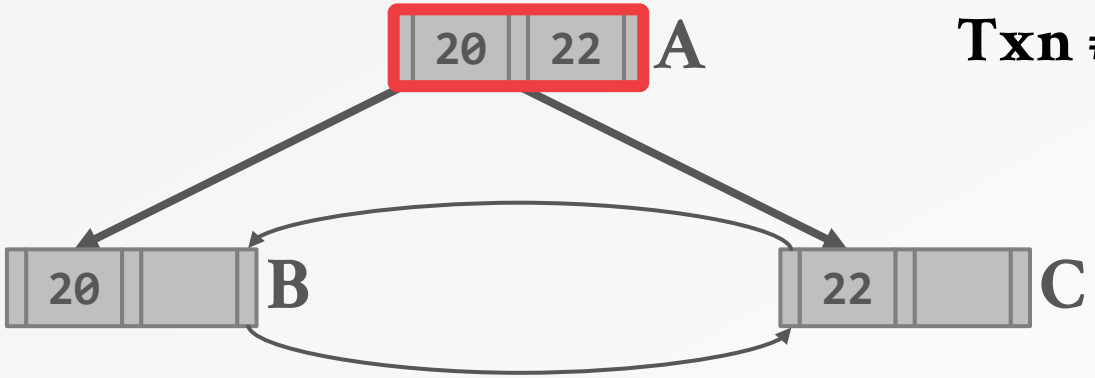
Txn #1: Read '22'

SIMPLE EXAMPLE

Txn #1: Read '22'

Txn #2: Insert '21'

SIMPLE EXAMPLE



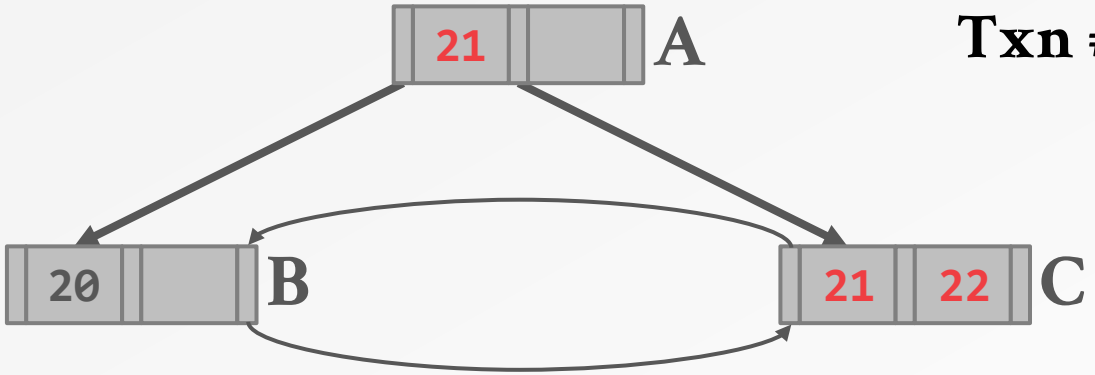
Txn #1: Read '22'

Txn #2: Insert '21'

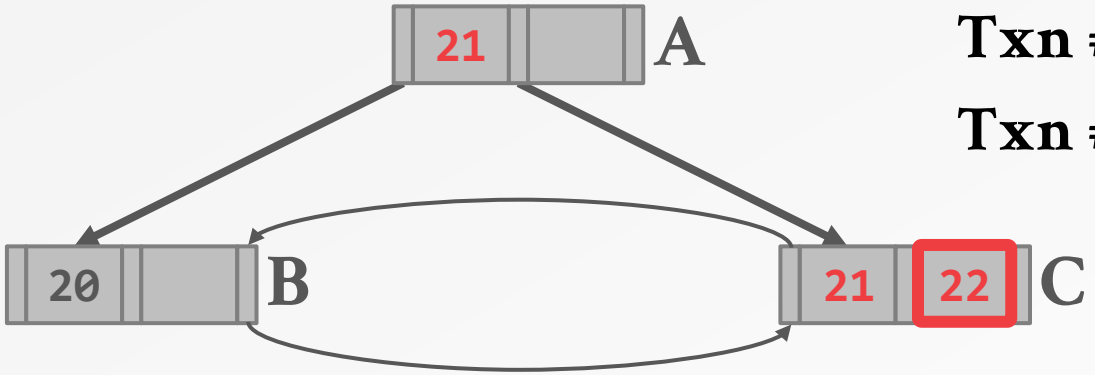
SIMPLE EXAMPLE

Txn #1: Read '22'

Txn #2: Insert '21'



SIMPLE EXAMPLE



Txn #1: Read '22'

Txn #2: Insert '21'

Txn #3: Read '22'

LOCKS VS. LATCHES

Locks

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

Latches

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

A SURVEY OF B-TREE LOCKING TECHNIQUES
TODS 2010

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

LOCK-FREE INDEXES

Possibility #1: No Locks

→ Txns don't acquire locks to access/modify database.
→ Still have to use latches to install updates.

Possibility #2: No Latches

→ Swap pointers using atomic updates to install changes.
→ Still have to use locks to validate txns.

LATCH IMPLEMENTATIONS

Blocking OS Mutex

Test-and-Set Spinlock

Queue-based Spinlock

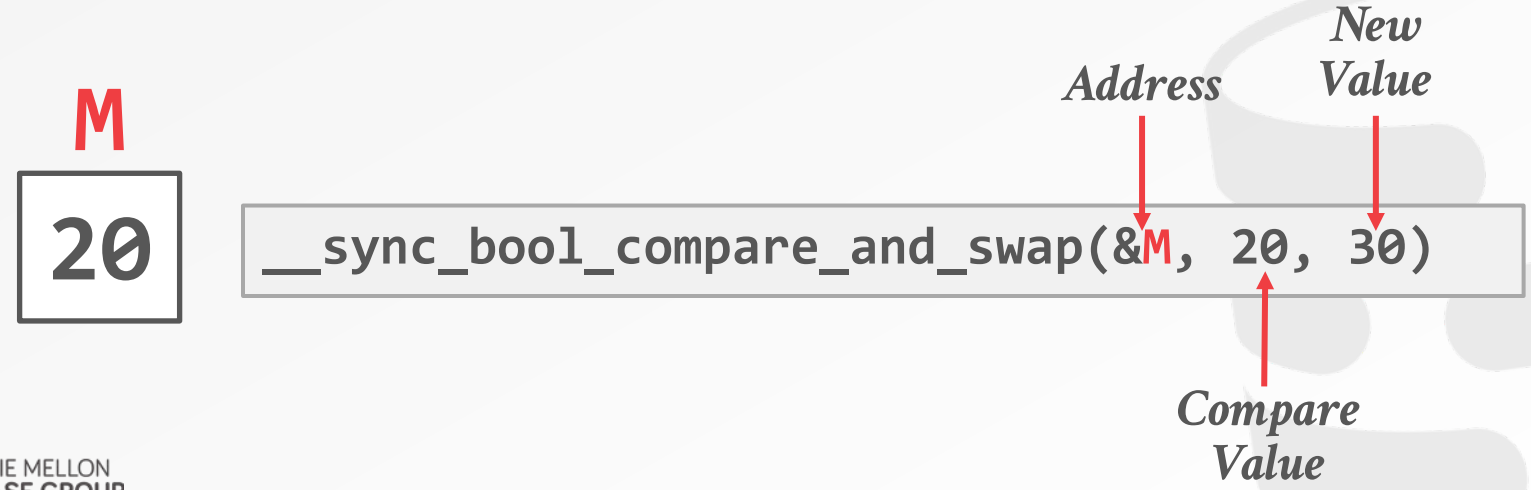
Reader-Writer Locks

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

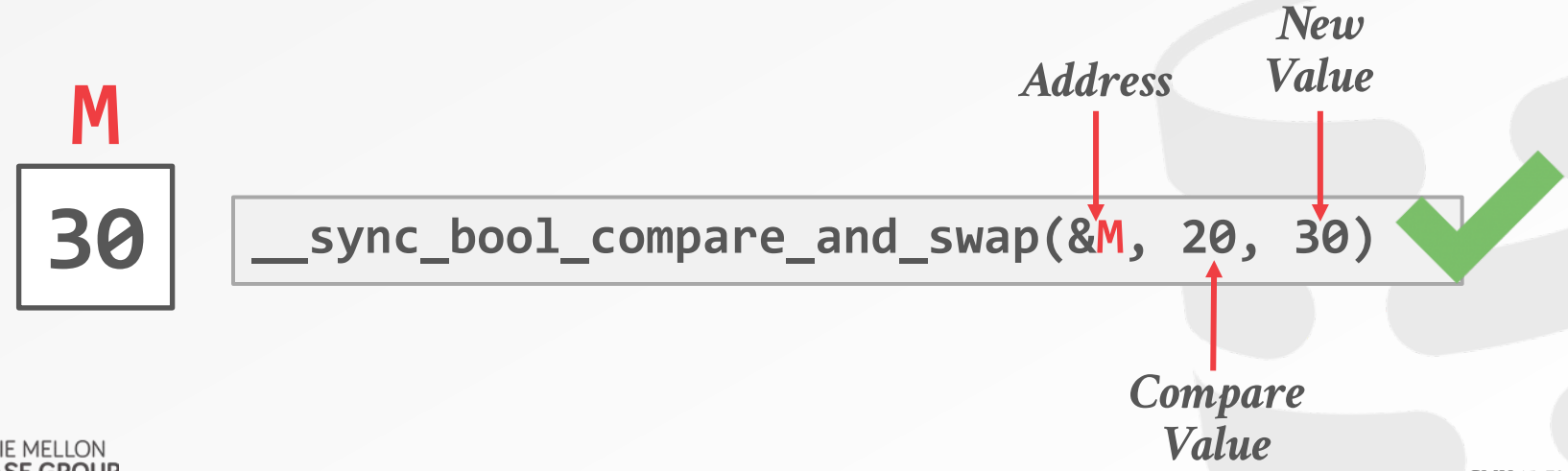


COMPARE-AND-SWAP

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



LATCH IMPLEMENTATIONS

Choice #1: 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

Choice #2: Test-and-Set Spinlock (TAS)

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

std::atomic<bool>

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

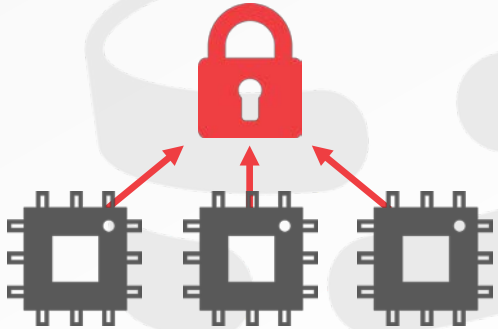
LATCH IMPLEMENTATIONS

Choice #2: Test-and-Set Spinlock (TAS)

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

std::atomic<bool>

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

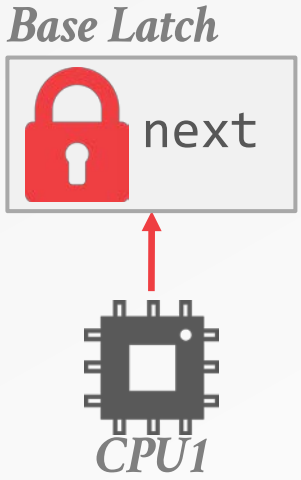


LATCH IMPLEMENTATIONS

Mellor-Crummey and Scott

Choice #3: Queue-based Spinlock (MCS)

→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

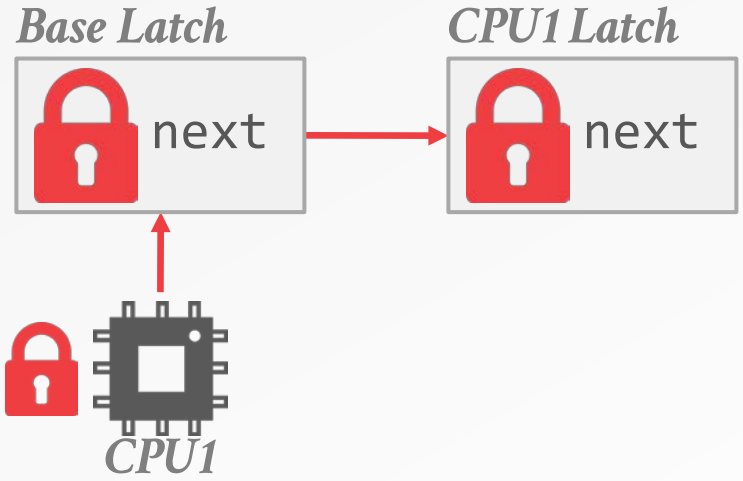


LATCH IMPLEMENTATIONS

Mellor-Crummey and Scott

Choice #3: Queue-based Spinlock (MCS)

→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

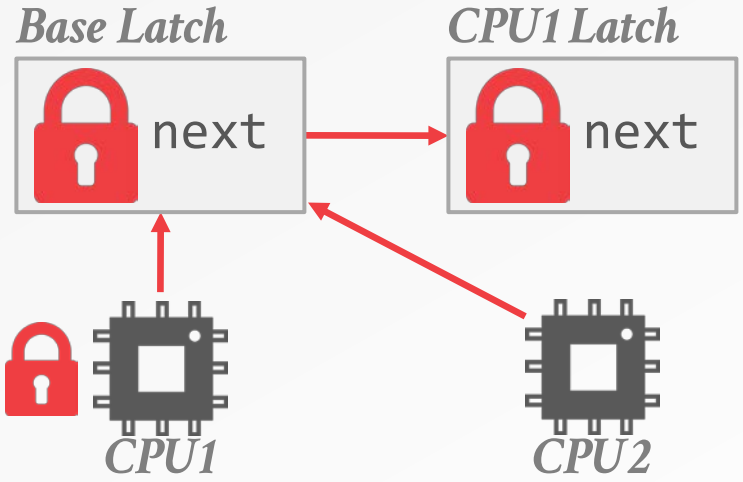


LATCH IMPLEMENTATIONS

Mellor-Crummey and Scott

Choice #3: Queue-based Spinlock (MCS)

→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

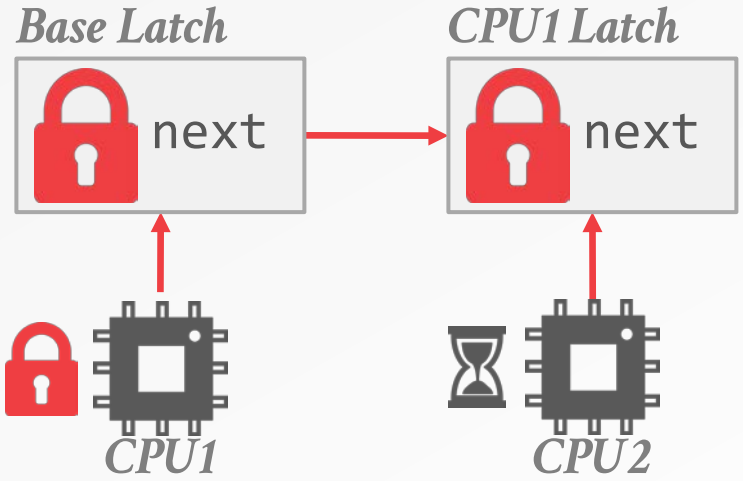


LATCH IMPLEMENTATIONS

Mellor-Crummey and Scott

Choice #3: Queue-based Spinlock (MCS)

→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

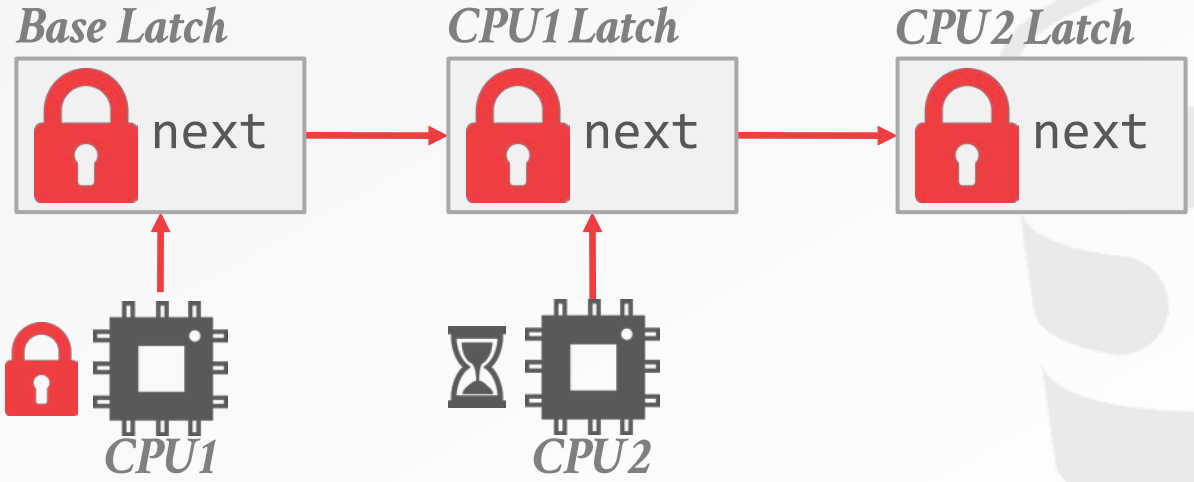


LATCH IMPLEMENTATIONS

Mellor-Crummey and Scott

Choice #3: Queue-based Spinlock (MCS)

→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`

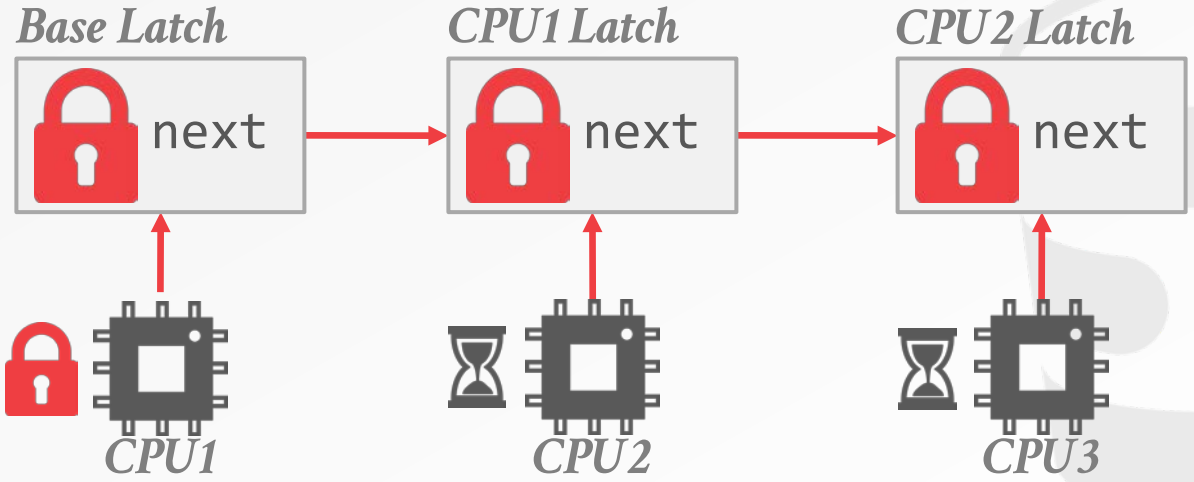


LATCH IMPLEMENTATIONS

Mellor-Crummey and Scott

Choice #3: Queue-based Spinlock (MCS)

→ More efficient than mutex, better cache locality
→ Non-trivial memory management
→ Example: `std::atomic<Latch*>`



LATCH IMPLEMENTATIONS

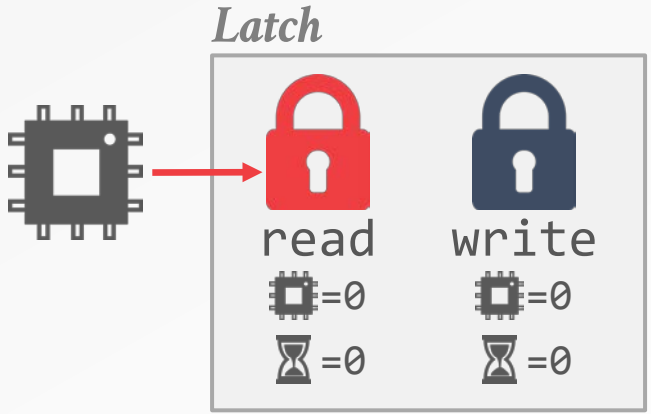
Choice #4: Reader-Writer Locks

→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks

LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

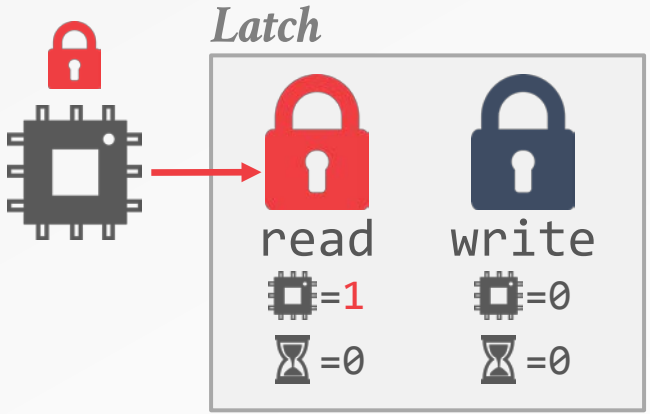
→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

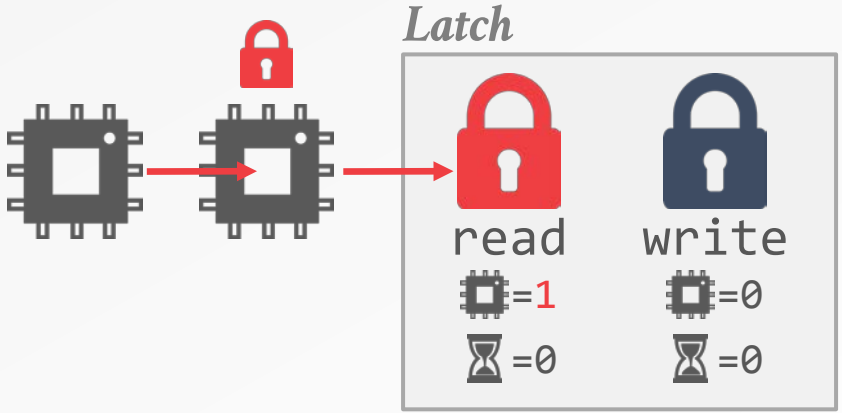
→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

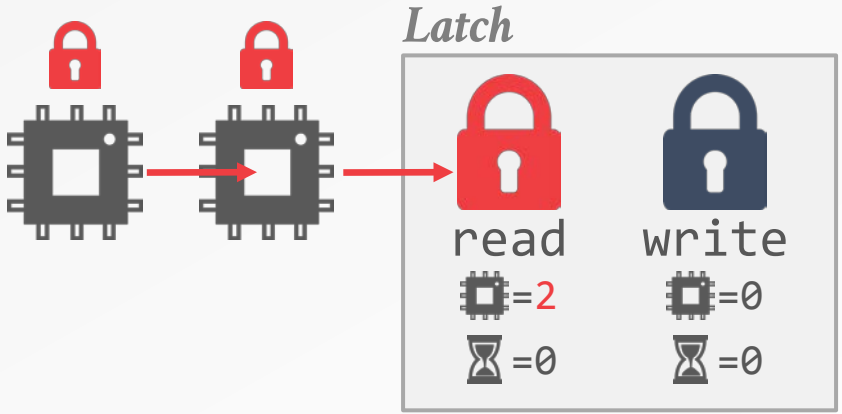
→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

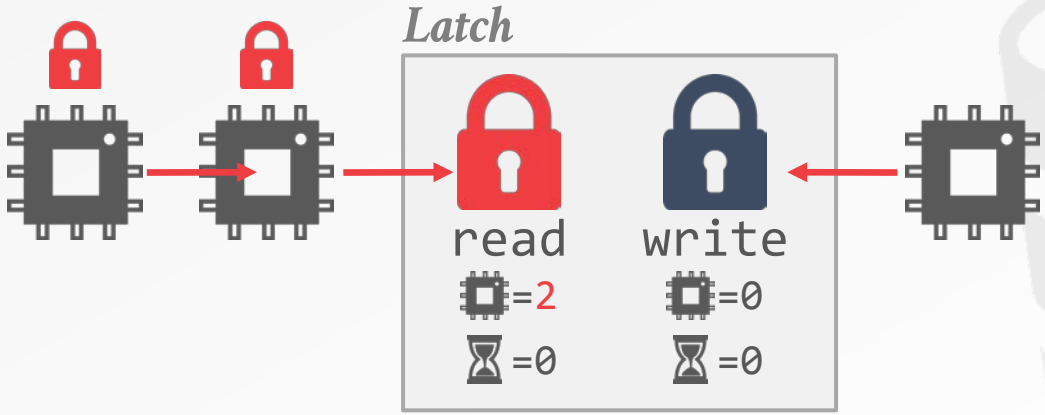
→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

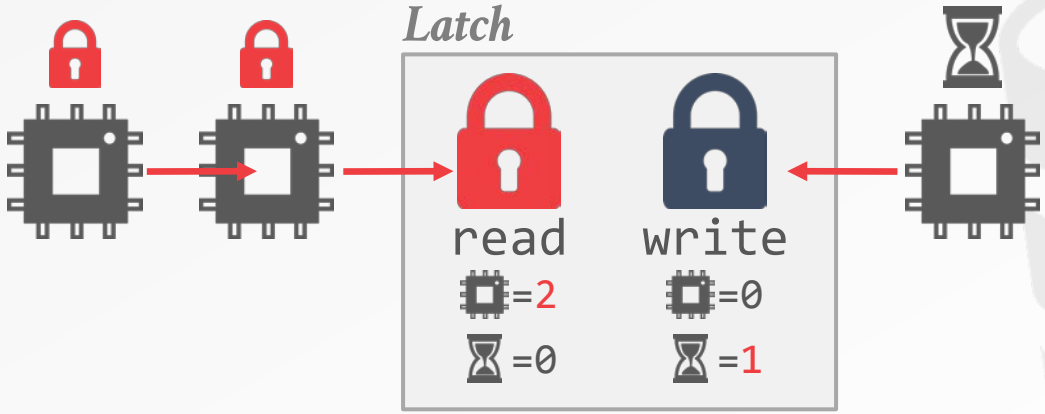
→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

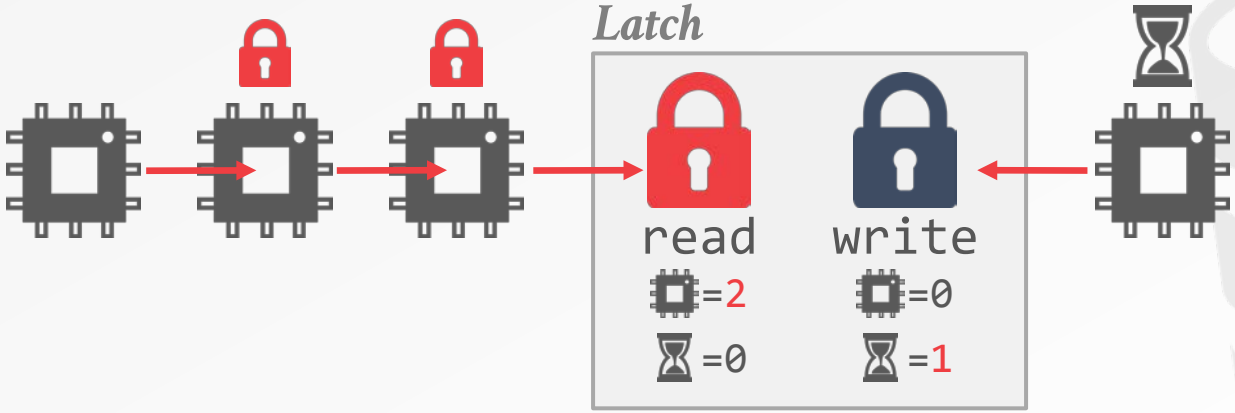
→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

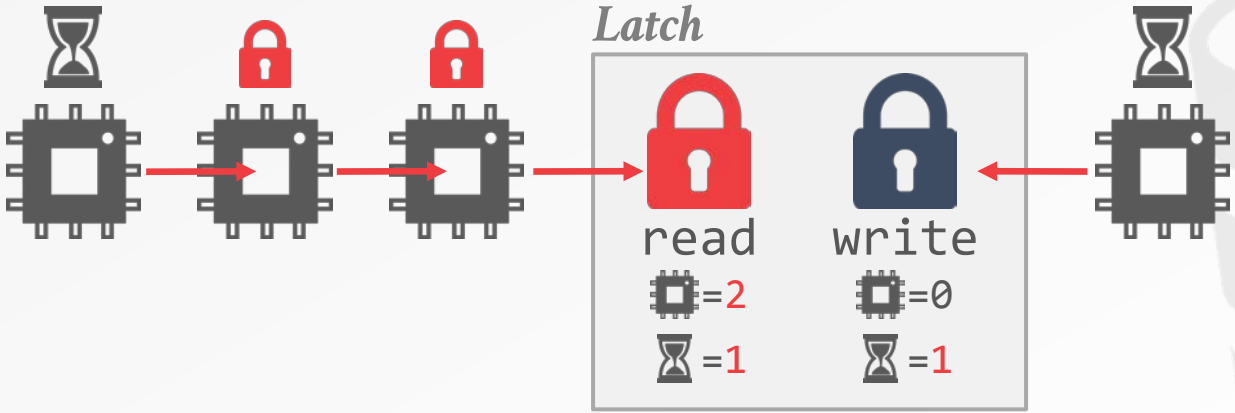
→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

→ Allows for concurrent readers
→ Have to manage read/write queues to avoid starvation
→ Can be implemented on top of spinlocks



LATCH CRABBING

Acquire and release latches on B+Tree nodes when traversing the data structure.

A thread can release latch on a parent node if its child node considered **safe**.

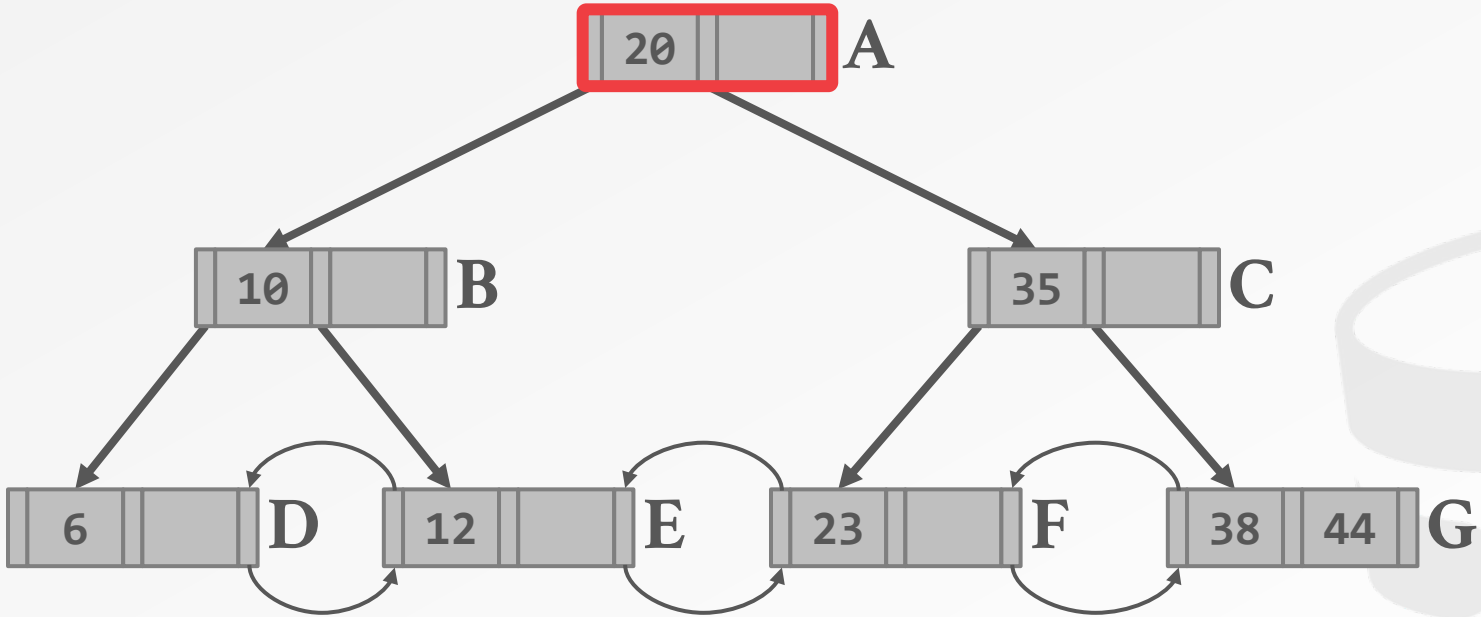
→ Any node that won't split or merge when updated.
→ Not full (on insertion)
→ More than half-full (on deletion)

LATCH CRABBING

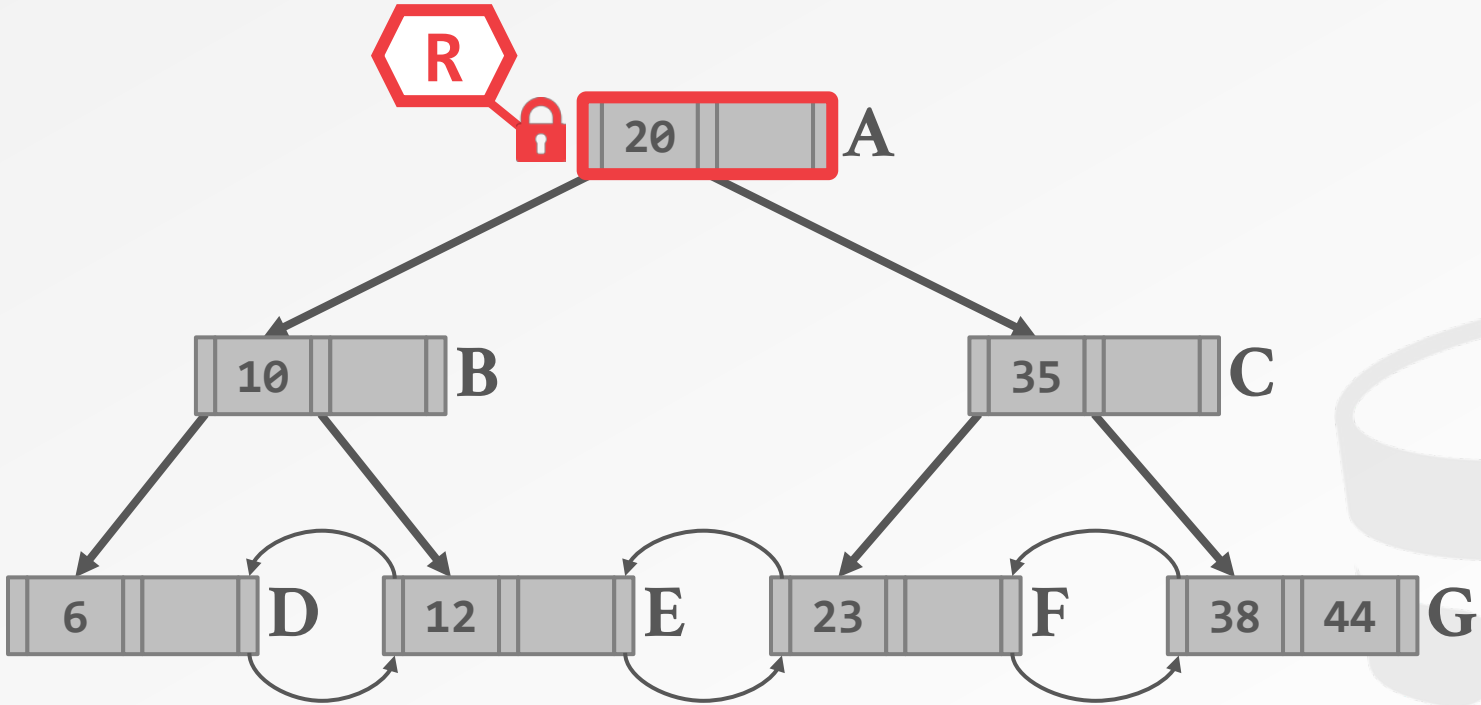
Search: Start at root and go down; repeatedly,
→ Acquire read (R) latch on child
→ Then unlock parent if the child is safe.

Insert/Delete: Start at root and go down, obtaining write (W) latches as needed.
Once child is locked, check if it is safe:
→ If child is safe, release all locks on ancestors.

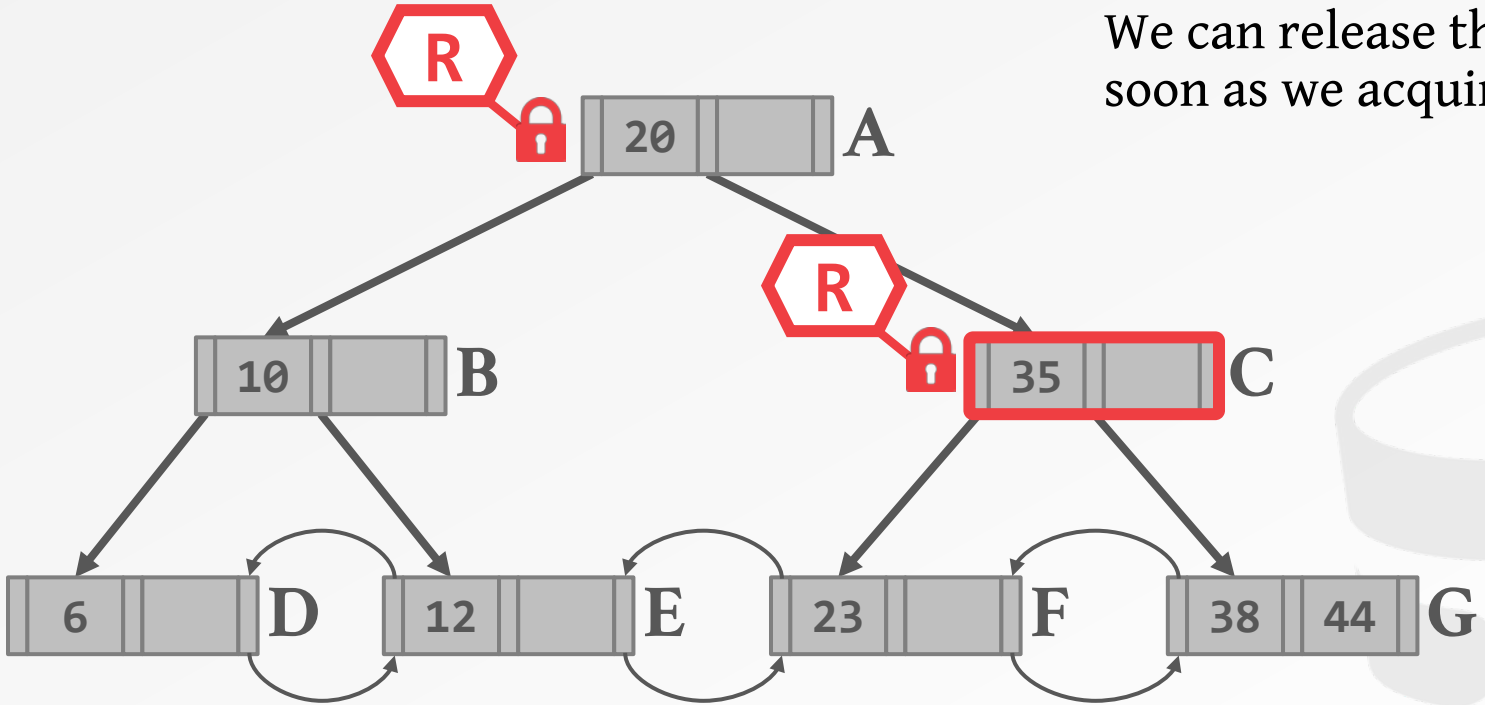
EXAMPLE #1: SEARCH 23



EXAMPLE #1: SEARCH 23



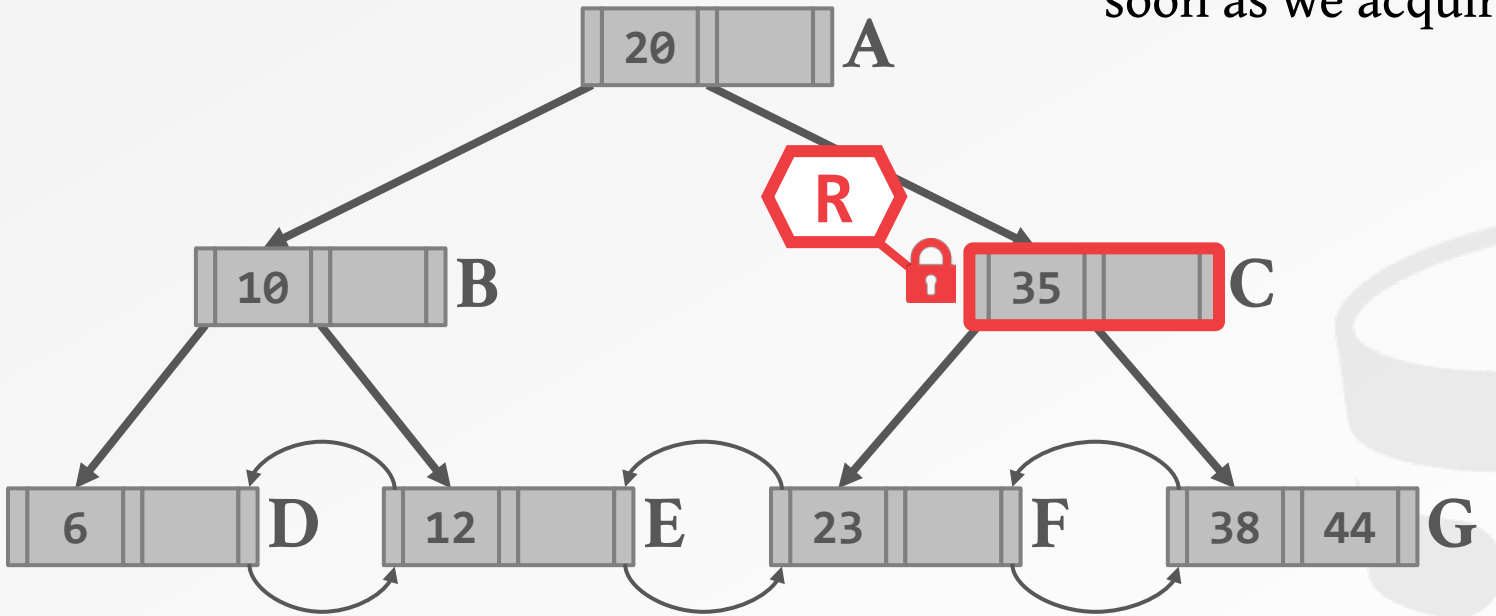
EXAMPLE #1: SEARCH 23



We can release the latch on **A** as soon as we acquire the latch for **C**.

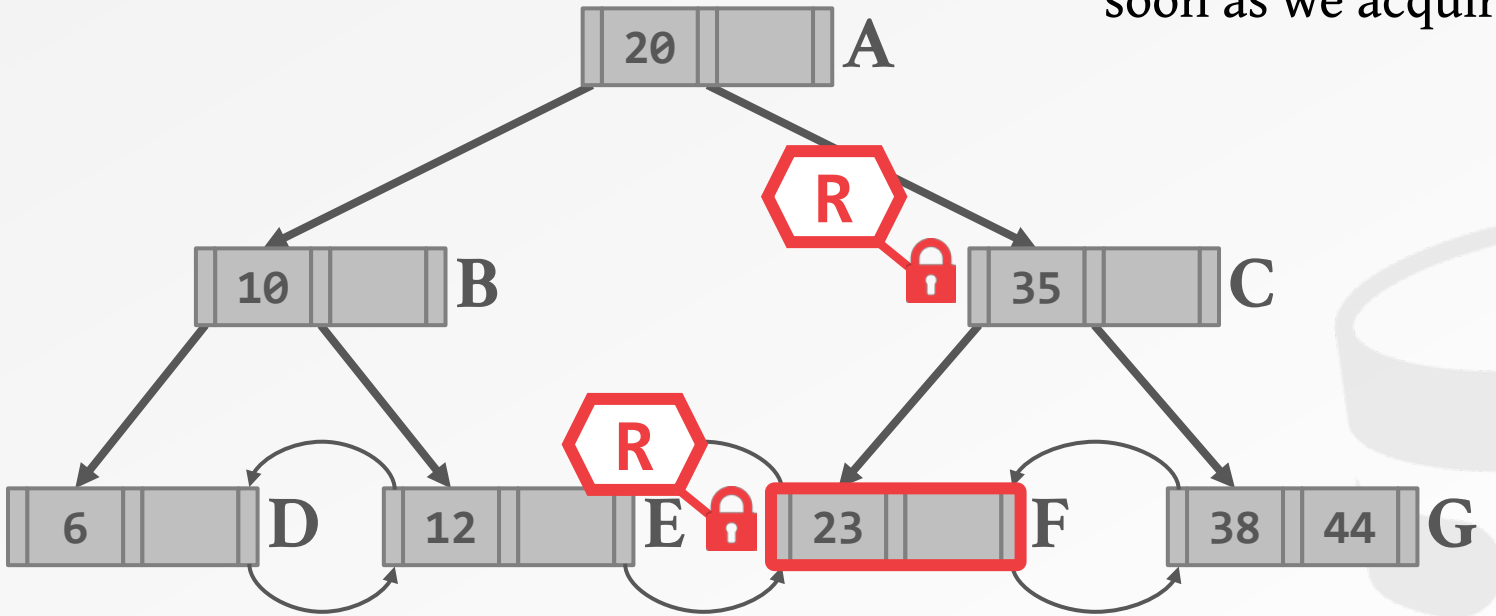
EXAMPLE #1: SEARCH 23

We can release the latch on **A** as soon as we acquire the latch for **C**.



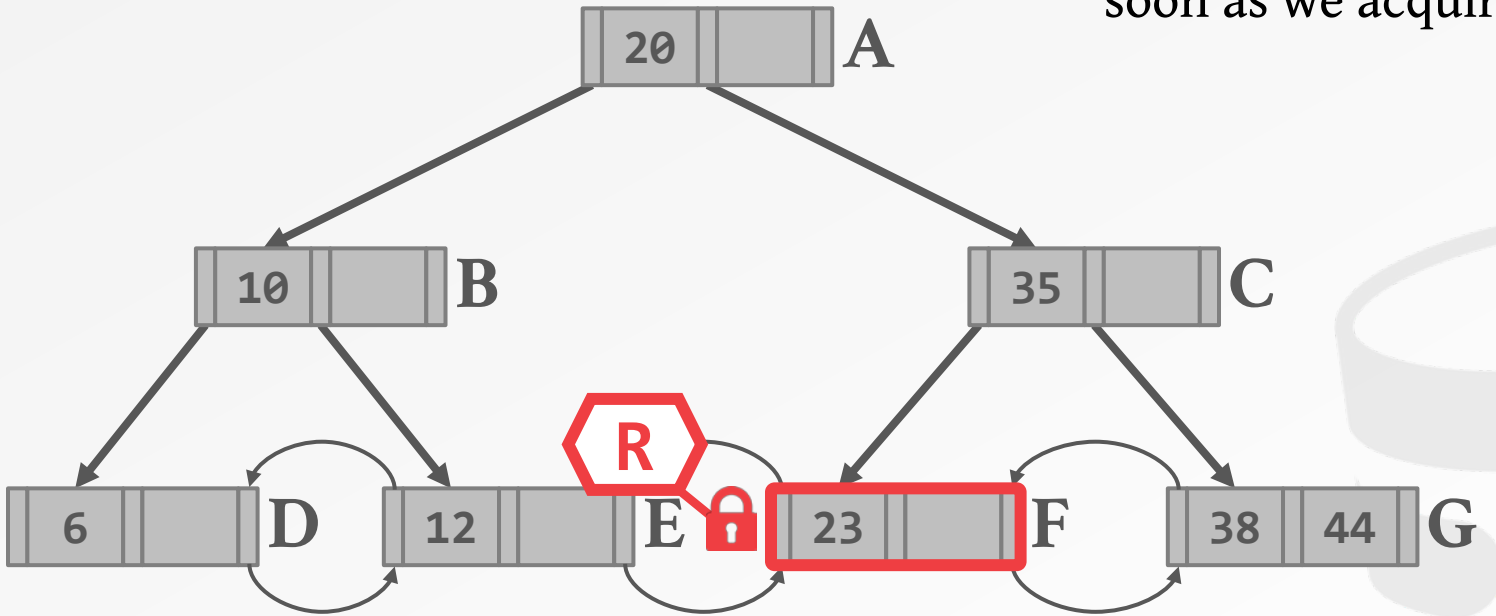
EXAMPLE #1: SEARCH 23

We can release the latch on **A** as soon as we acquire the latch for **C**.

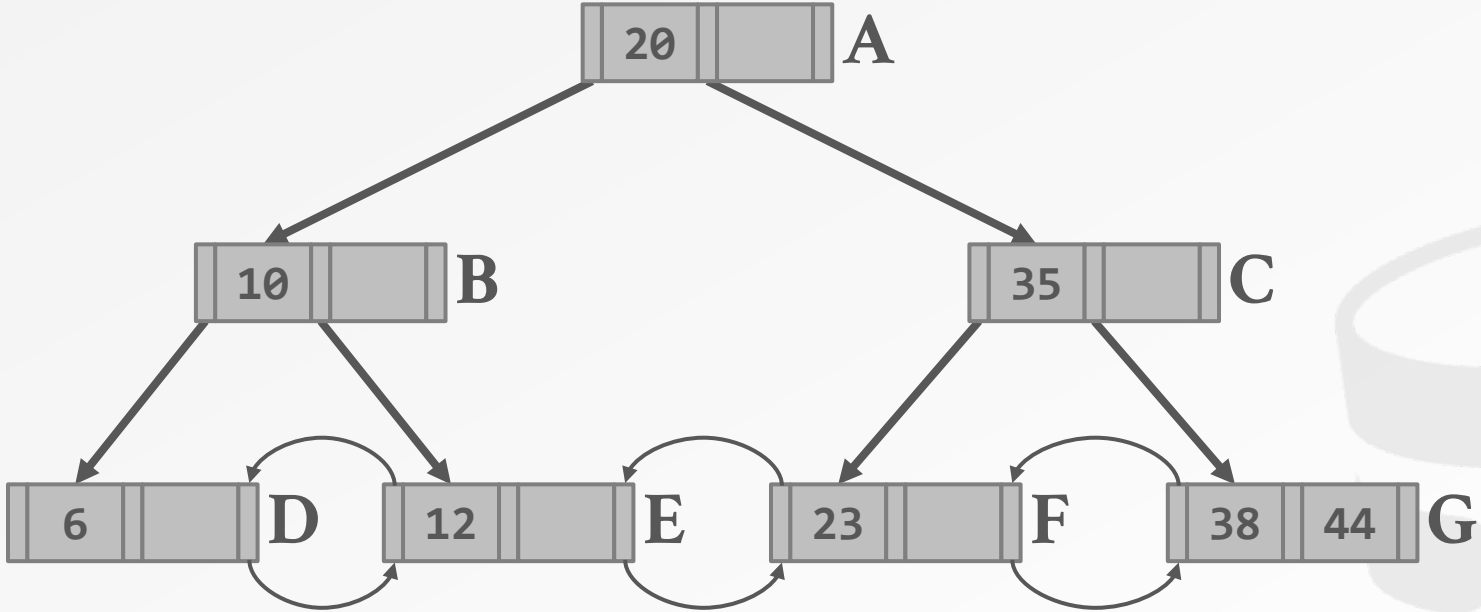


EXAMPLE #1: SEARCH 23

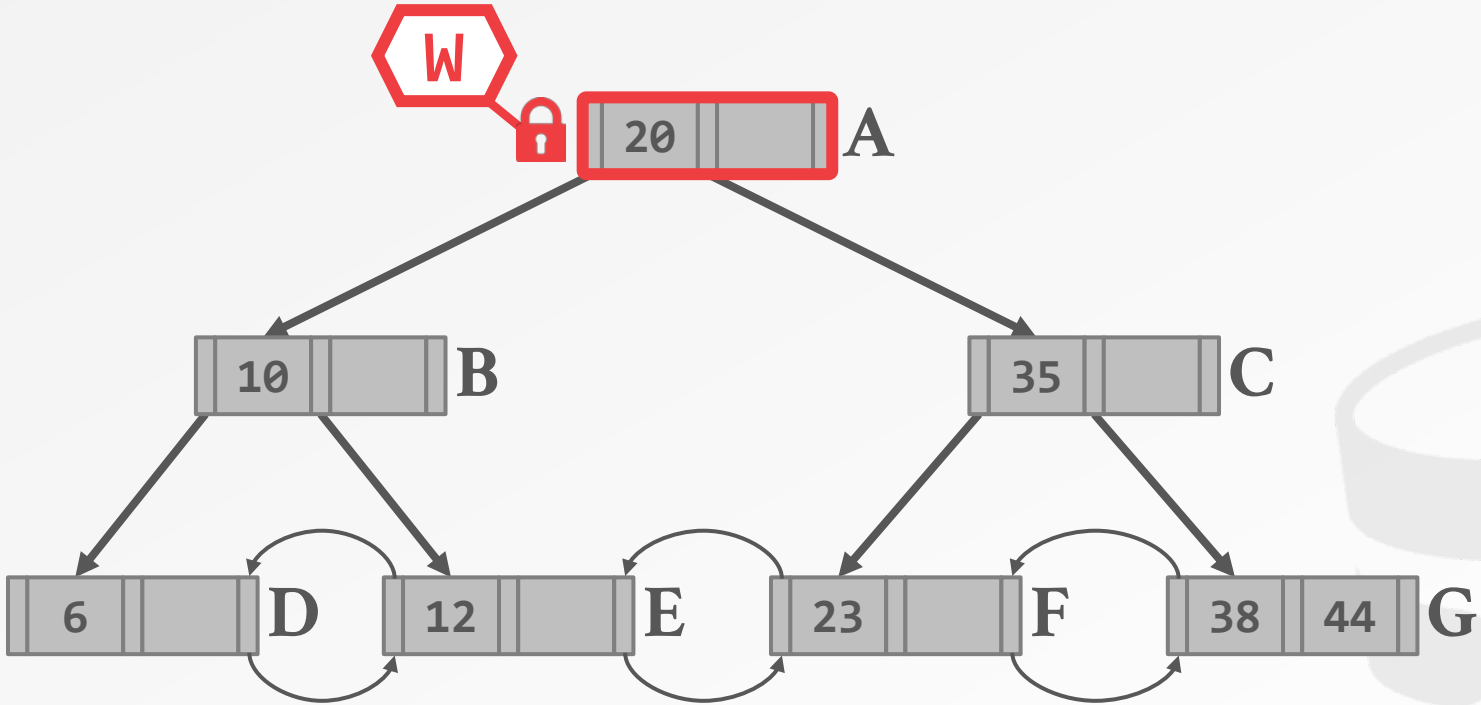
We can release the latch on **A** as soon as we acquire the latch for **C**.



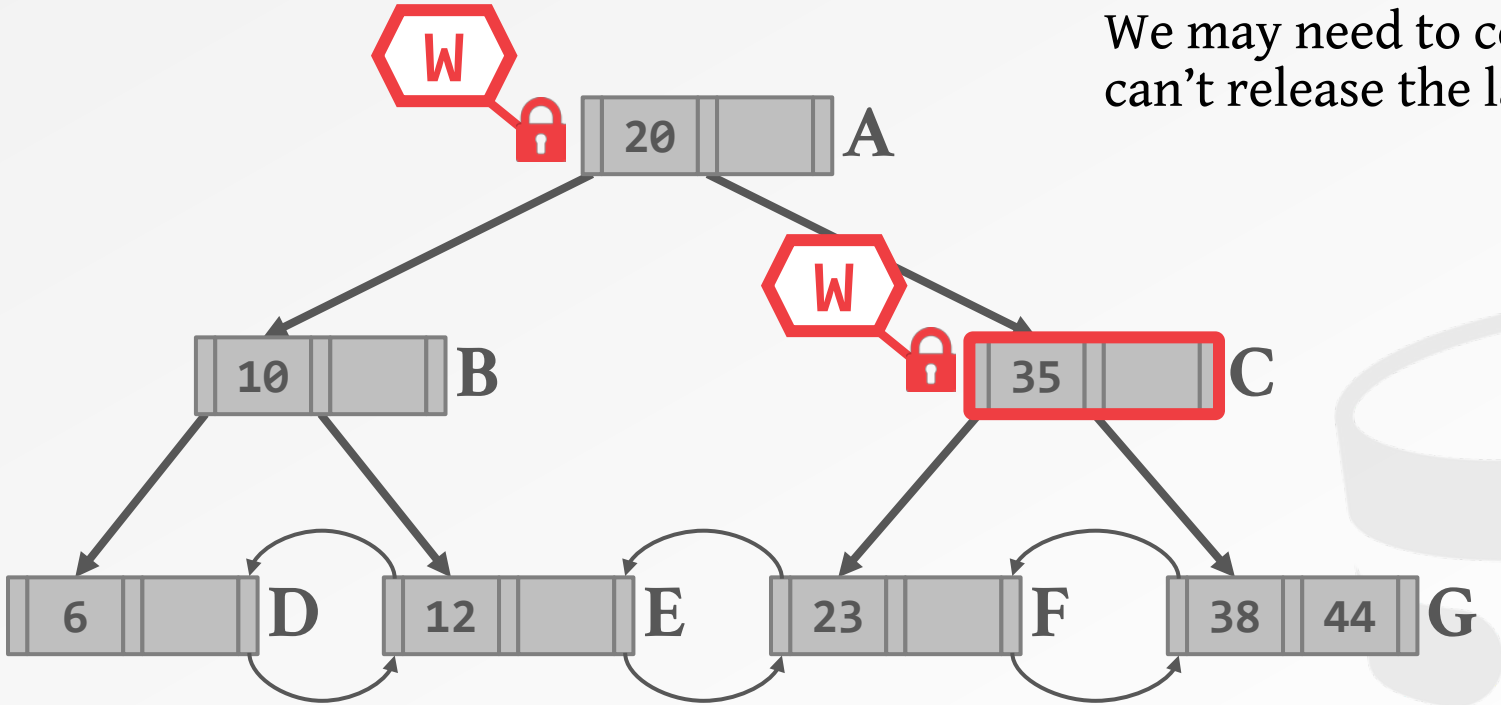
EXAMPLE #2: DELETE 44



EXAMPLE #2: DELETE 44

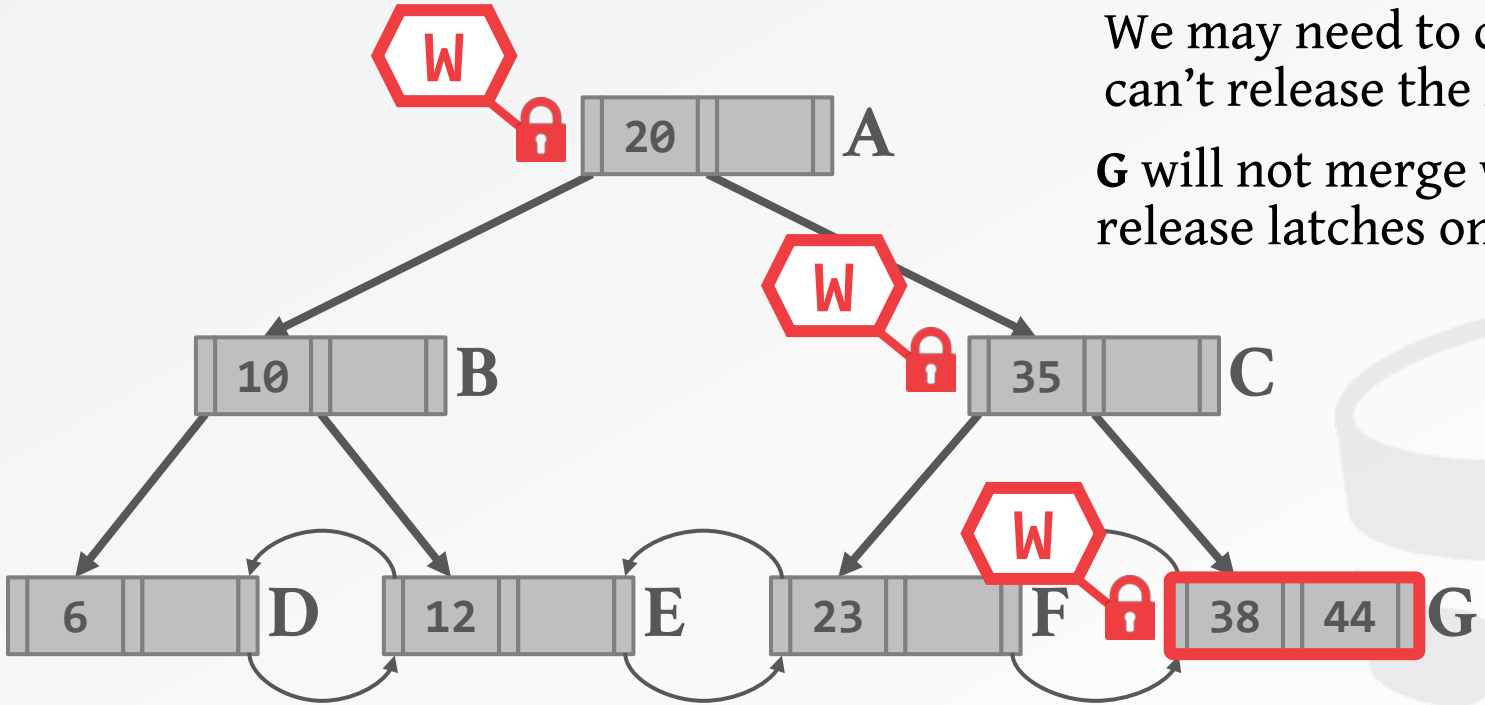


EXAMPLE #2: DELETE 44



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

EXAMPLE #2: DELETE 44



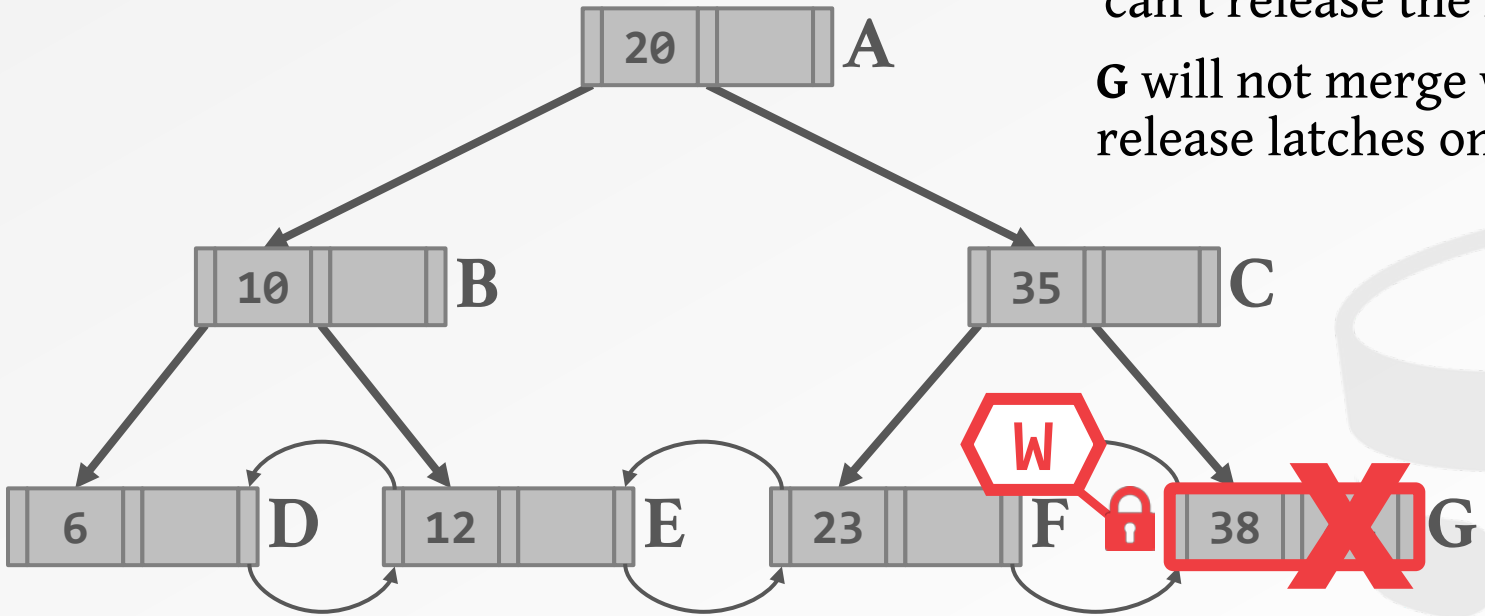
We may need to coalesce **C**, so we can't release the latch on **A**.

G will not merge with **F**, so we can release latches on **A** and **C**.

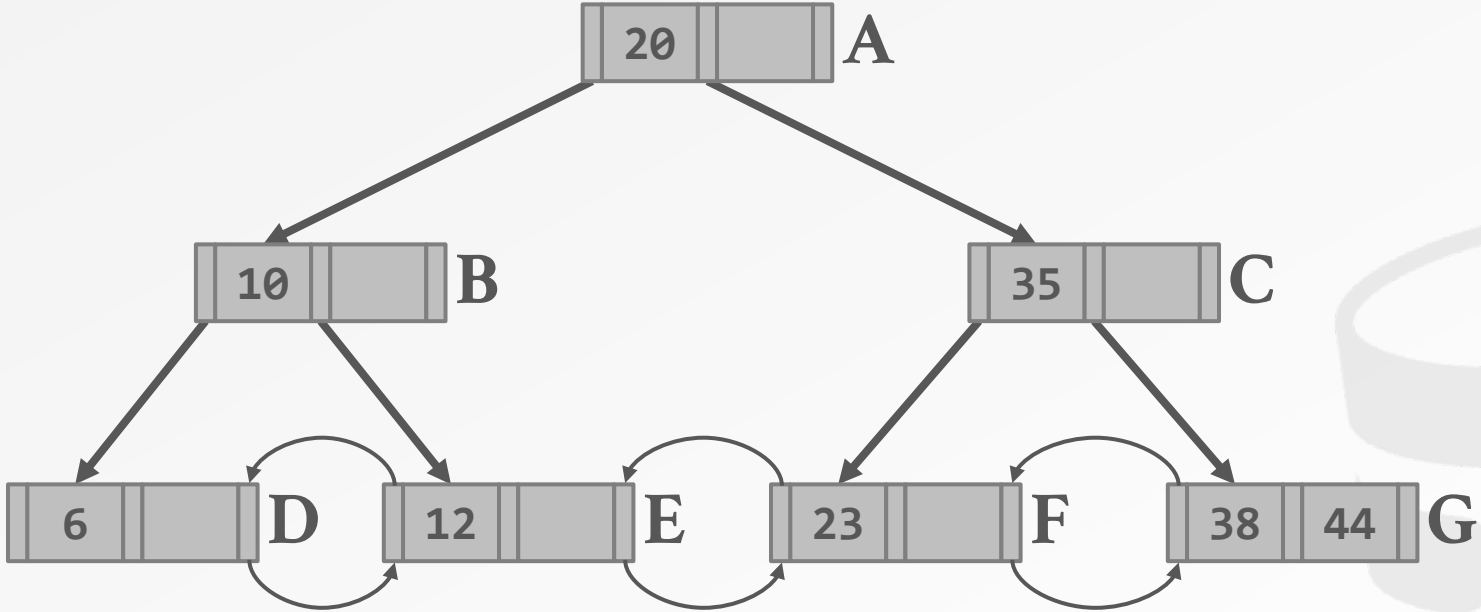
EXAMPLE #2: DELETE 44

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

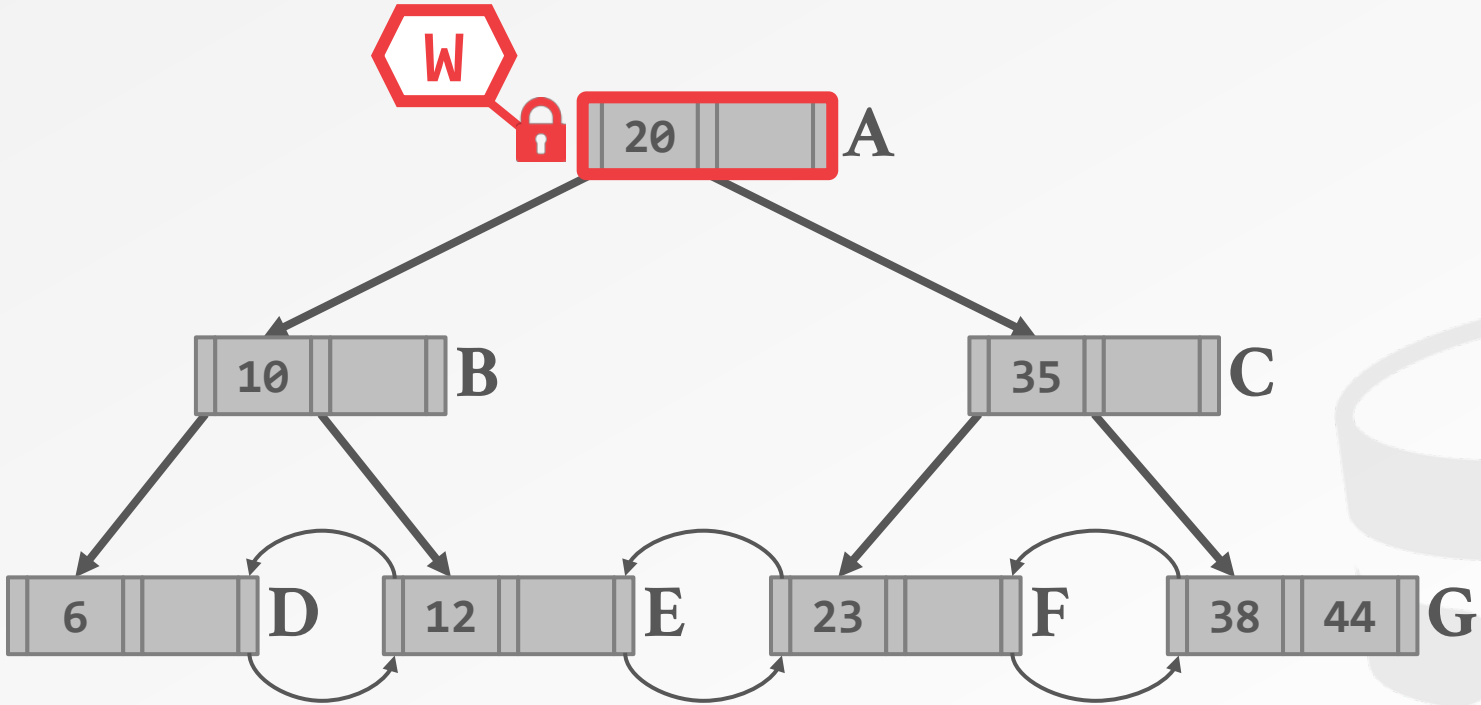
G will not merge with **F**, so we can release latches on **A** and **C**.



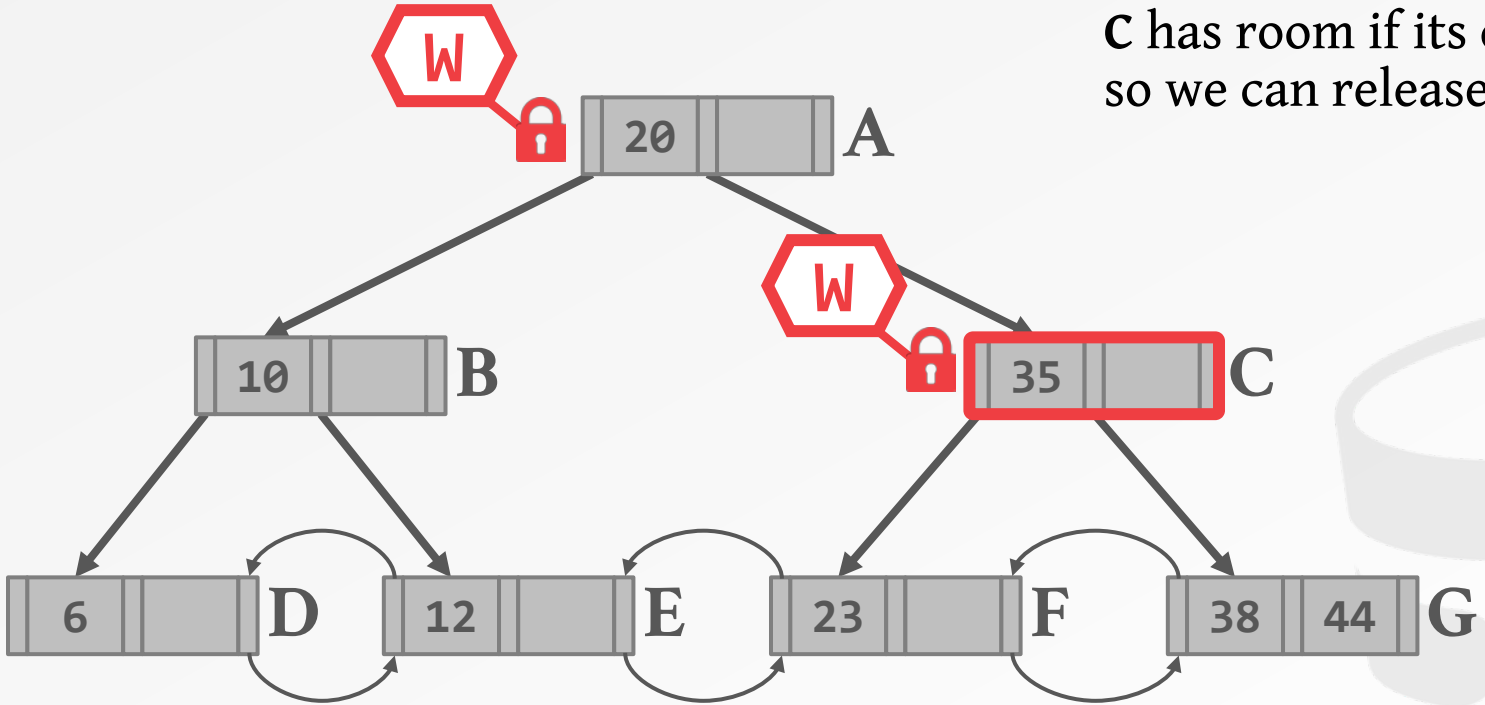
EXAMPLE #3: INSERT 40



EXAMPLE #3: INSERT 40



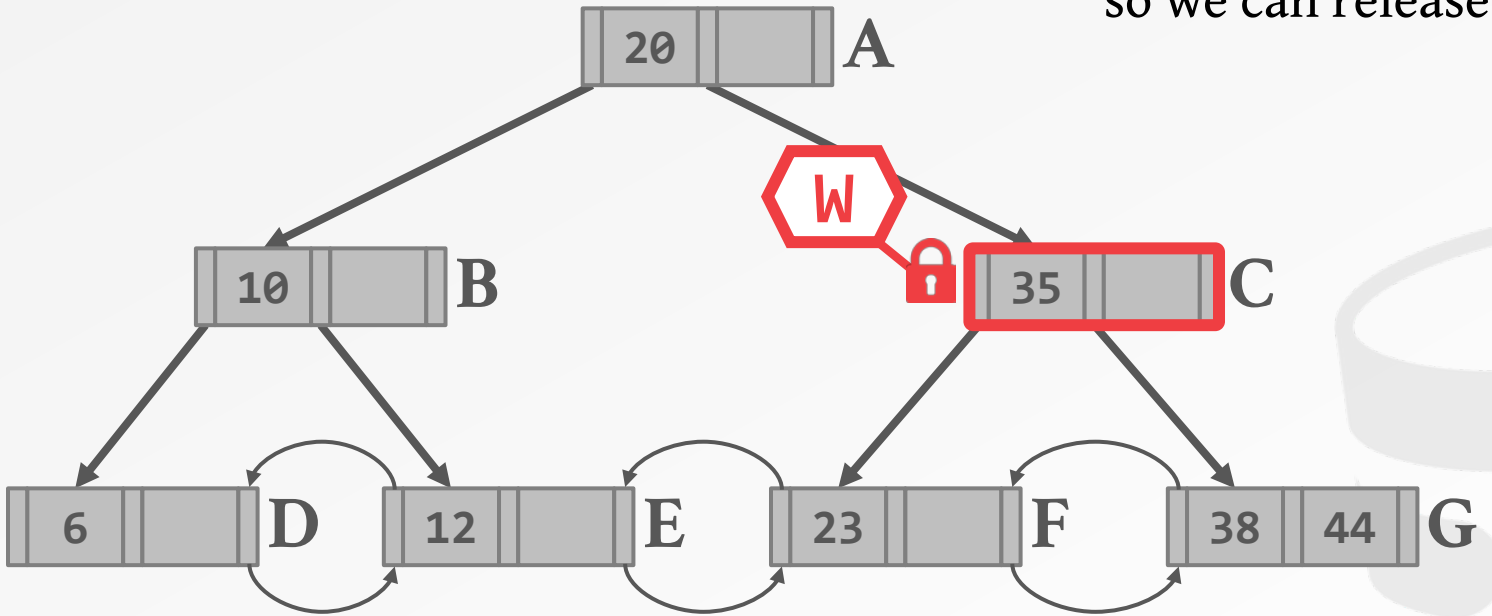
EXAMPLE #3: INSERT 40



C has room if its child has to split, so we can release the latch on **A**.

EXAMPLE #3: INSERT 40

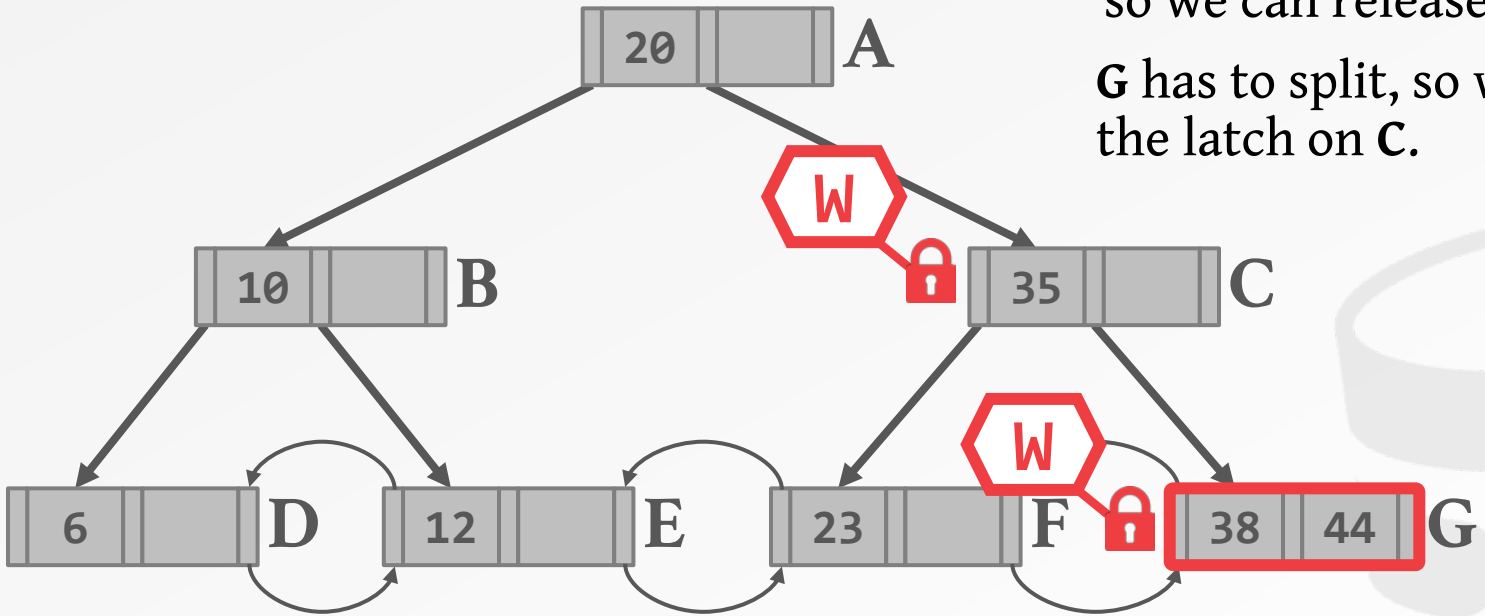
C has room if its child has to split, so we can release the latch on **A**.



EXAMPLE #3: INSERT 40

C has room if its child has to split, so we can release the latch on **A**.

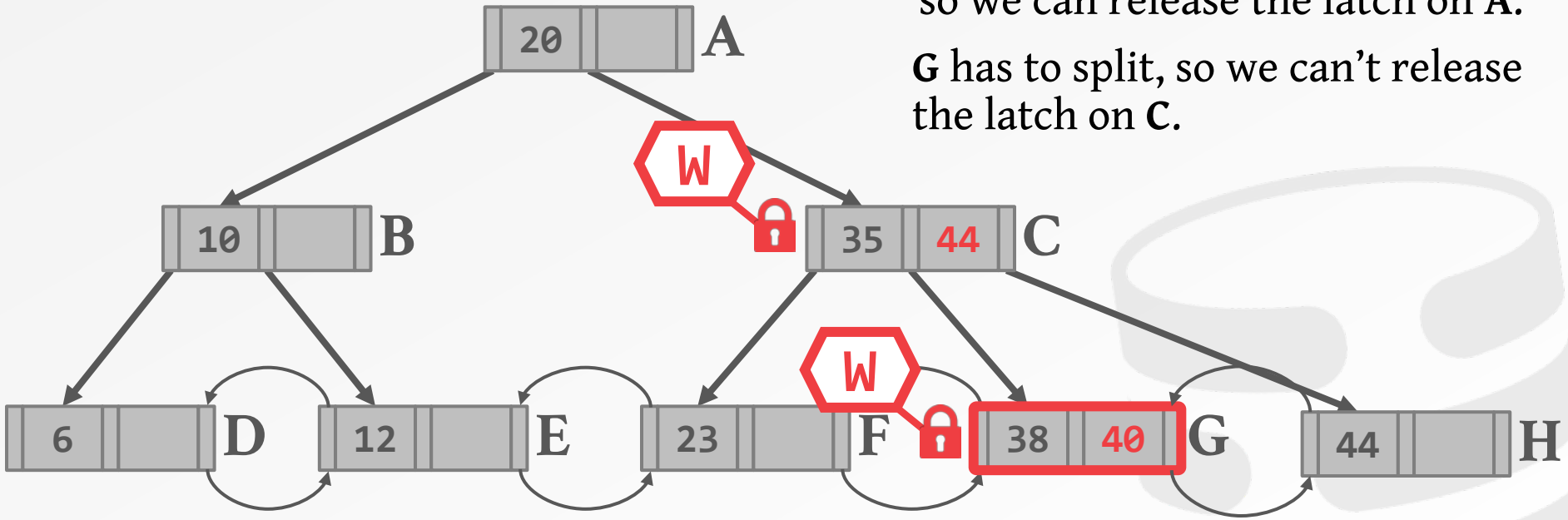
G has to split, so we can't release the latch on **C**.



EXAMPLE #3: INSERT 40

C has room if its child has to split, so we can release the latch on **A**.

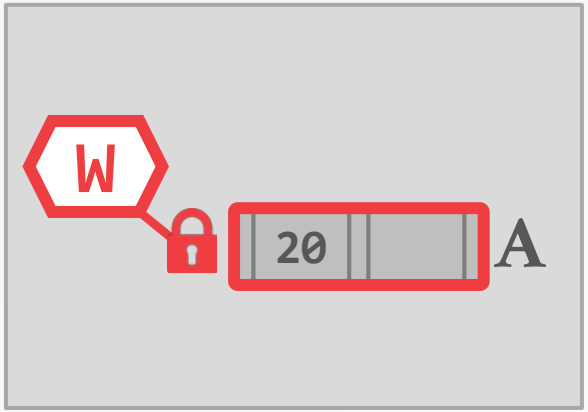
G has to split, so we can't release the latch on **C**.



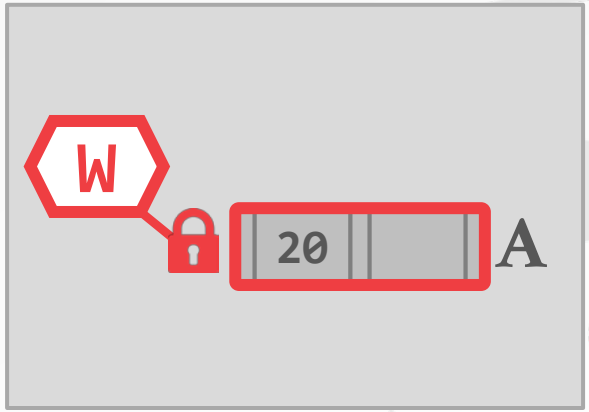
OBSERVATION

What was the first step that the DBMS took in the two examples that updated the index?

Delete 44



Insert 40



BETTER LATCH CRABBING

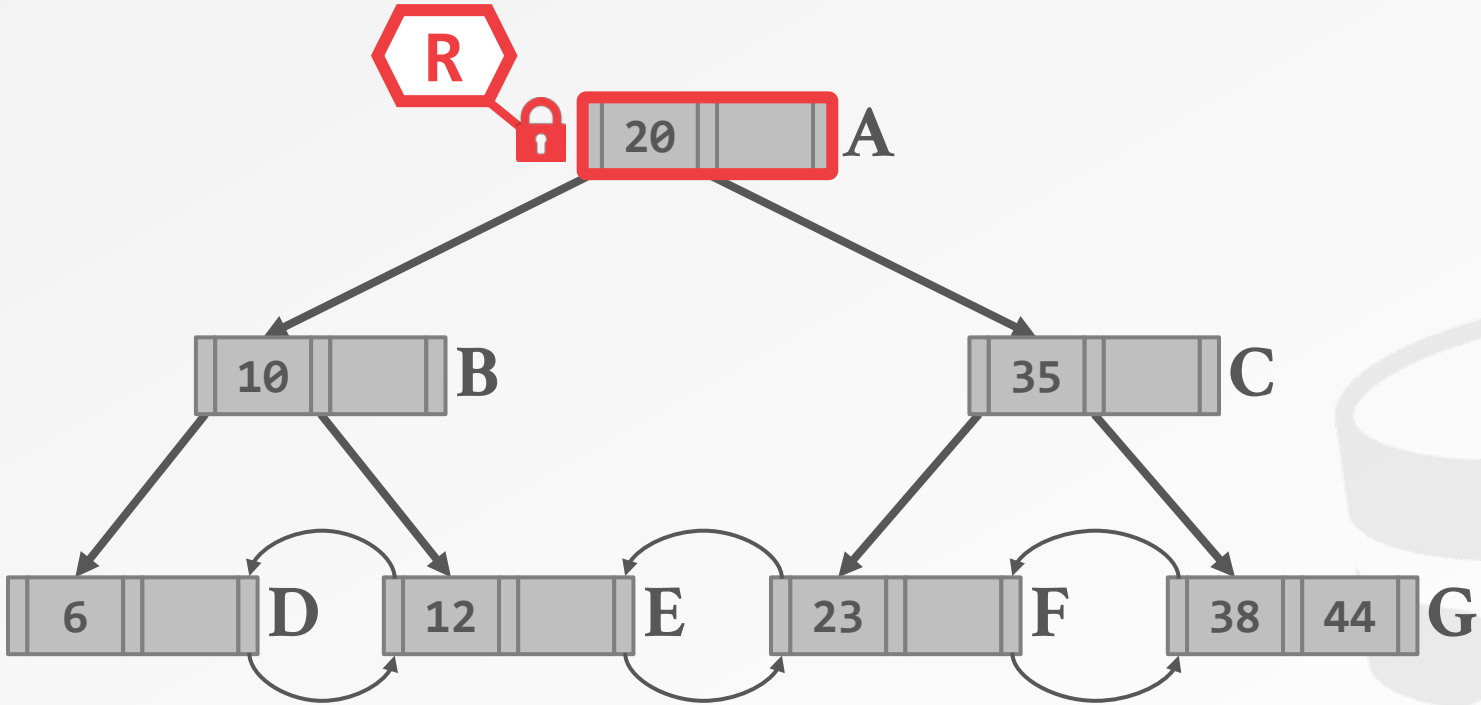
Optimistically assume that the leaf is safe.

→ Take **R** latches as you traverse the tree to reach it and verify.

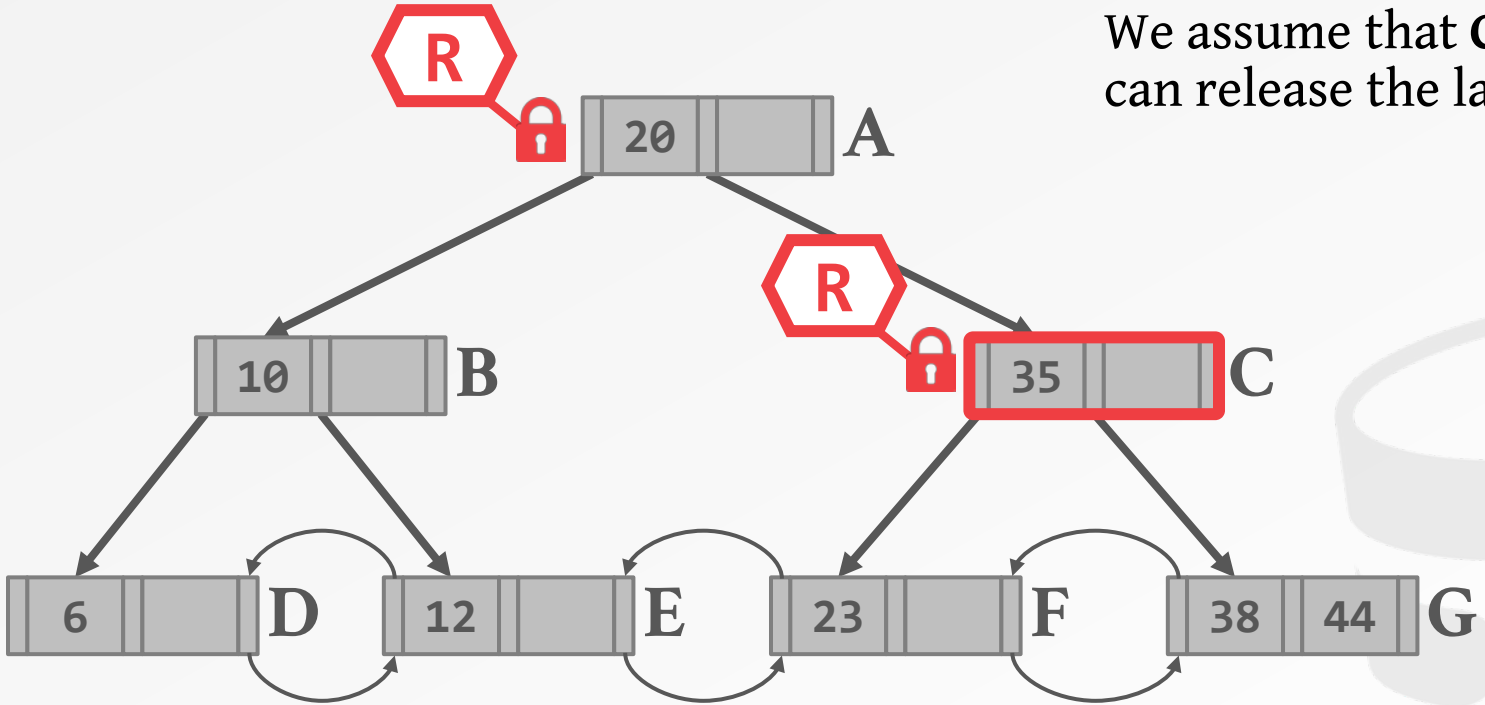
→ If leaf is not safe, then do previous algorithm.

CONCURRENCY OF OPERATIONS ON B-TREES
Acta Informatica 9: 1-21 1977

EXAMPLE #4: DELETE 44



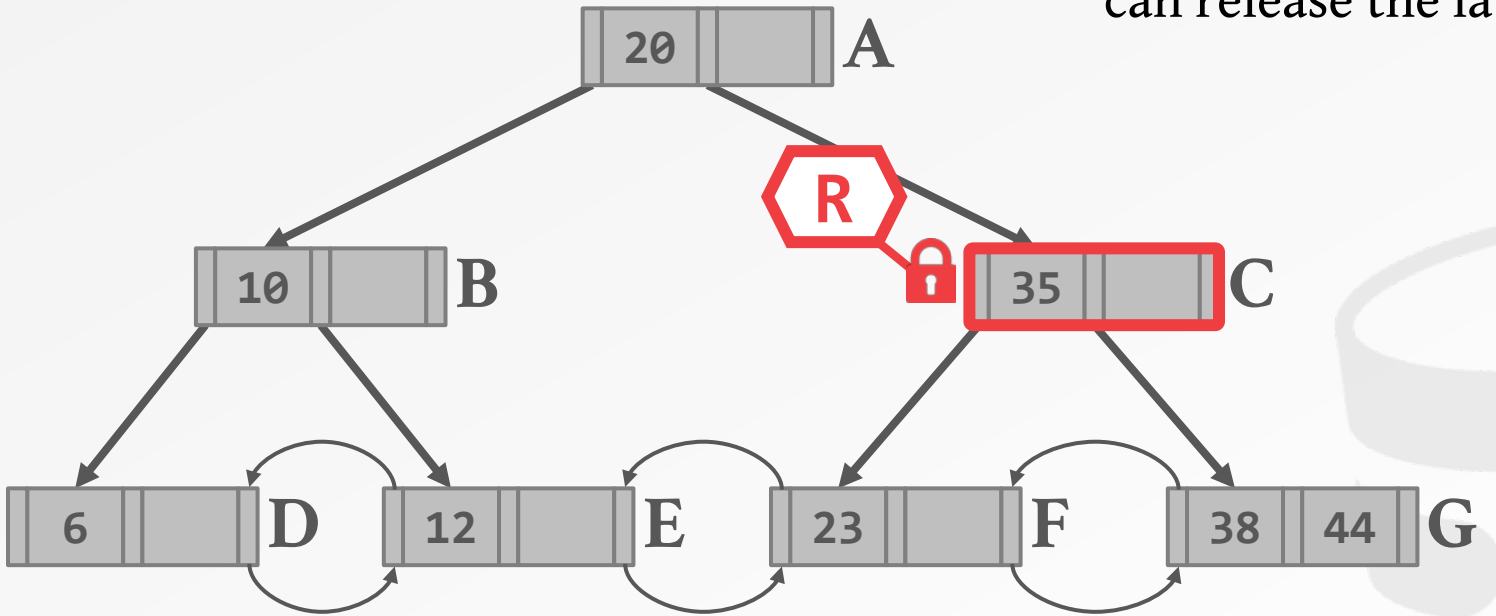
EXAMPLE #4: DELETE 44



We assume that **C** is safe, so we can release the latch on **A**.

EXAMPLE #4: DELETE 44

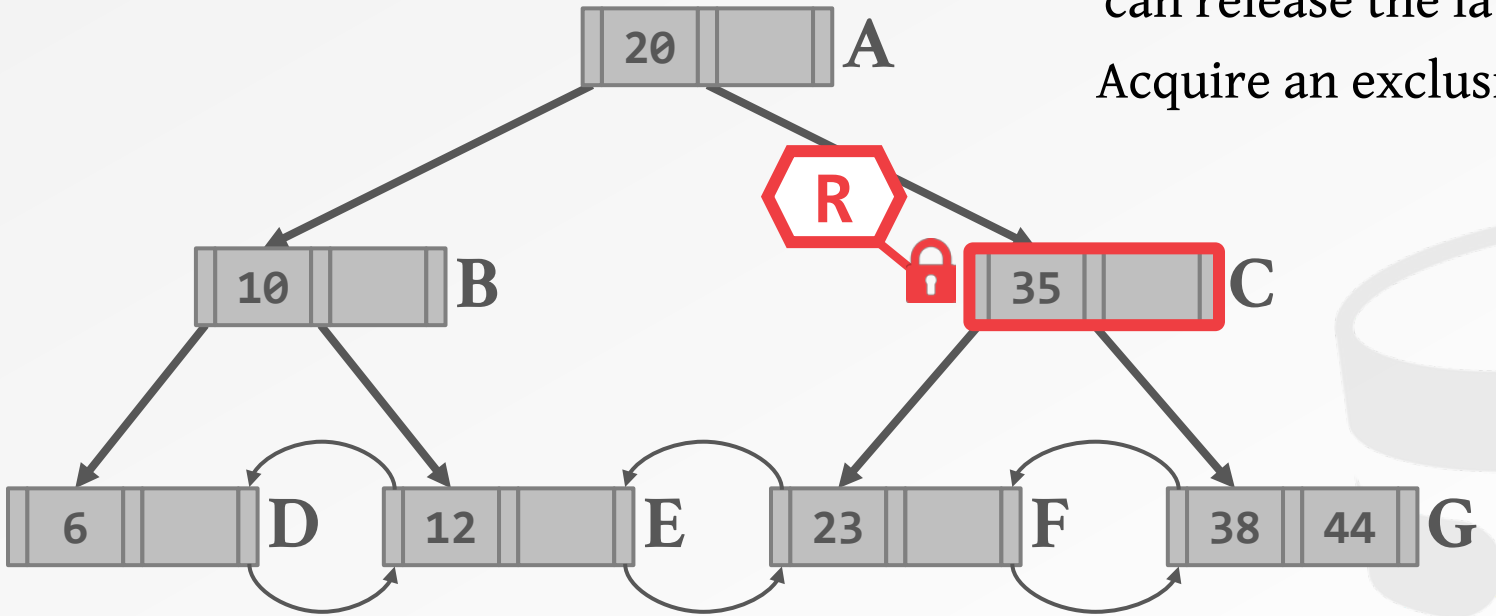
We assume that **C** is safe, so we can release the latch on **A**.



EXAMPLE #4: DELETE 44

We assume that **C** is safe, so we can release the latch on **A**.

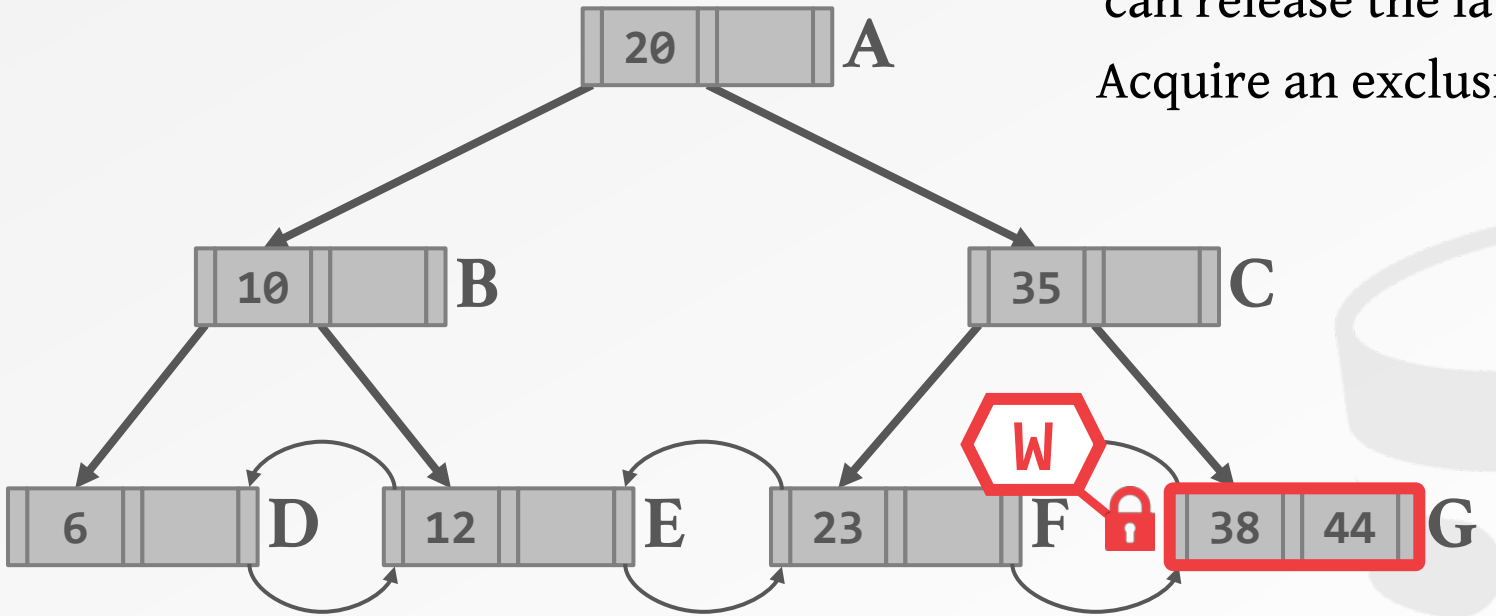
Acquire an exclusive latch on **G**.



EXAMPLE #4: DELETE 44

We assume that **C** is safe, so we can release the latch on **A**.

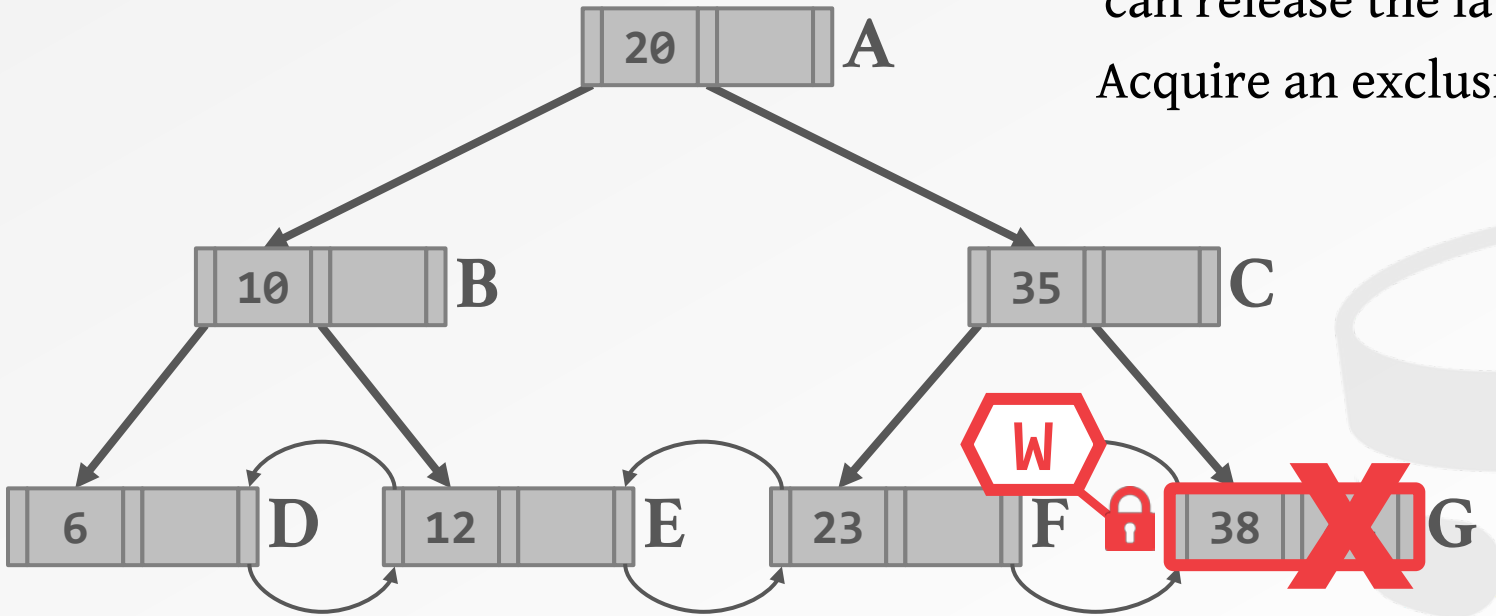
Acquire an exclusive latch on **G**.



EXAMPLE #4: DELETE 44

We assume that **C** is safe, so we can release the latch on **A**.

Acquire an exclusive latch on **G**.

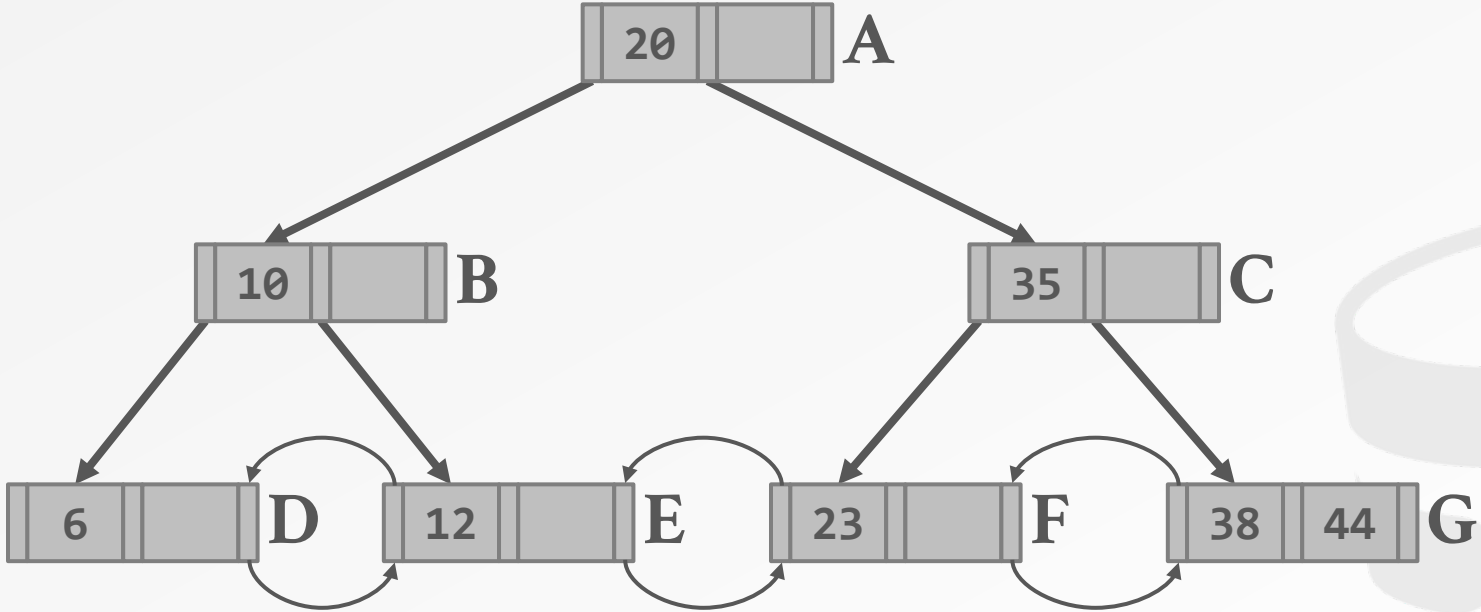


OBSERVATION

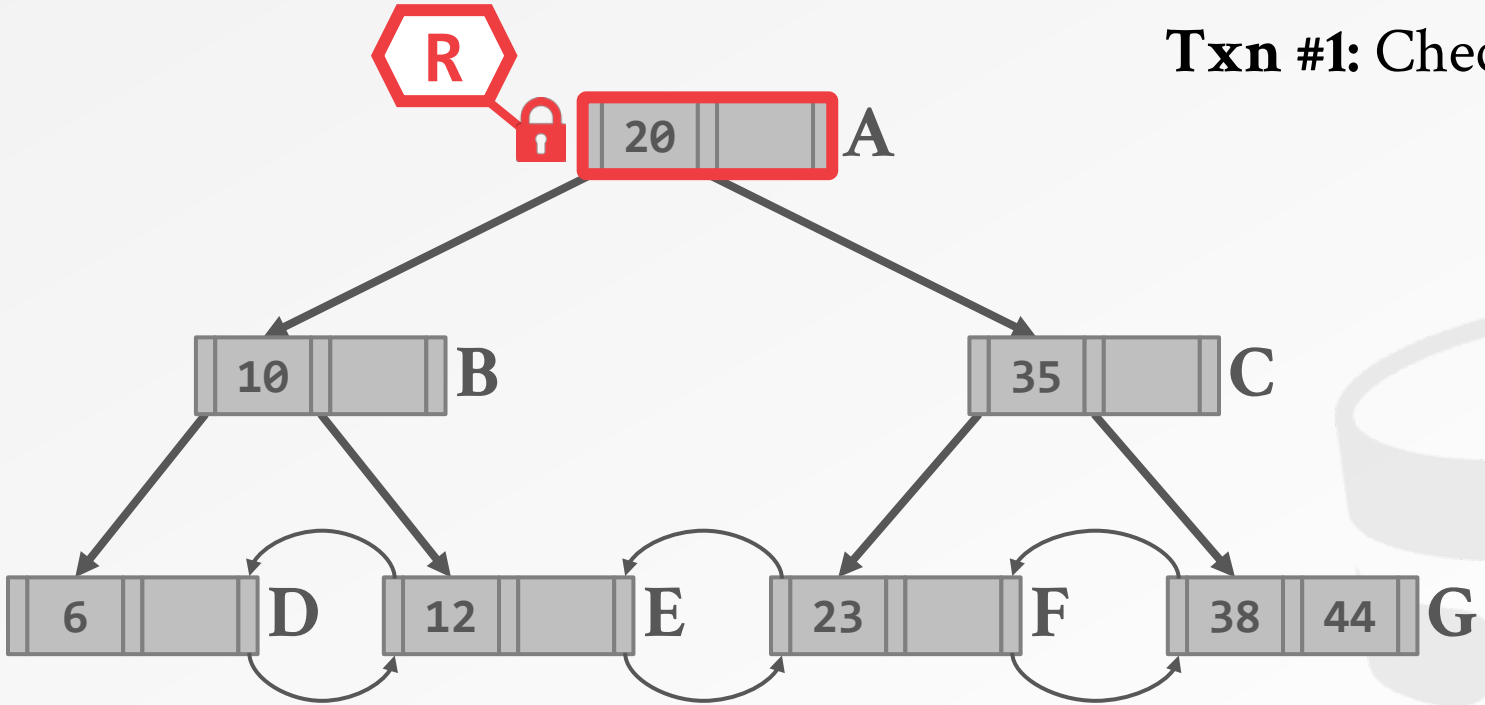
Crabbing ensures that txns do not corrupt the internal data structure during modifications.

But because txns release latches on each node as soon as they are finished their operations, we cannot guarantee that phantoms do not occur…

PROBLEM SCENARIO #1

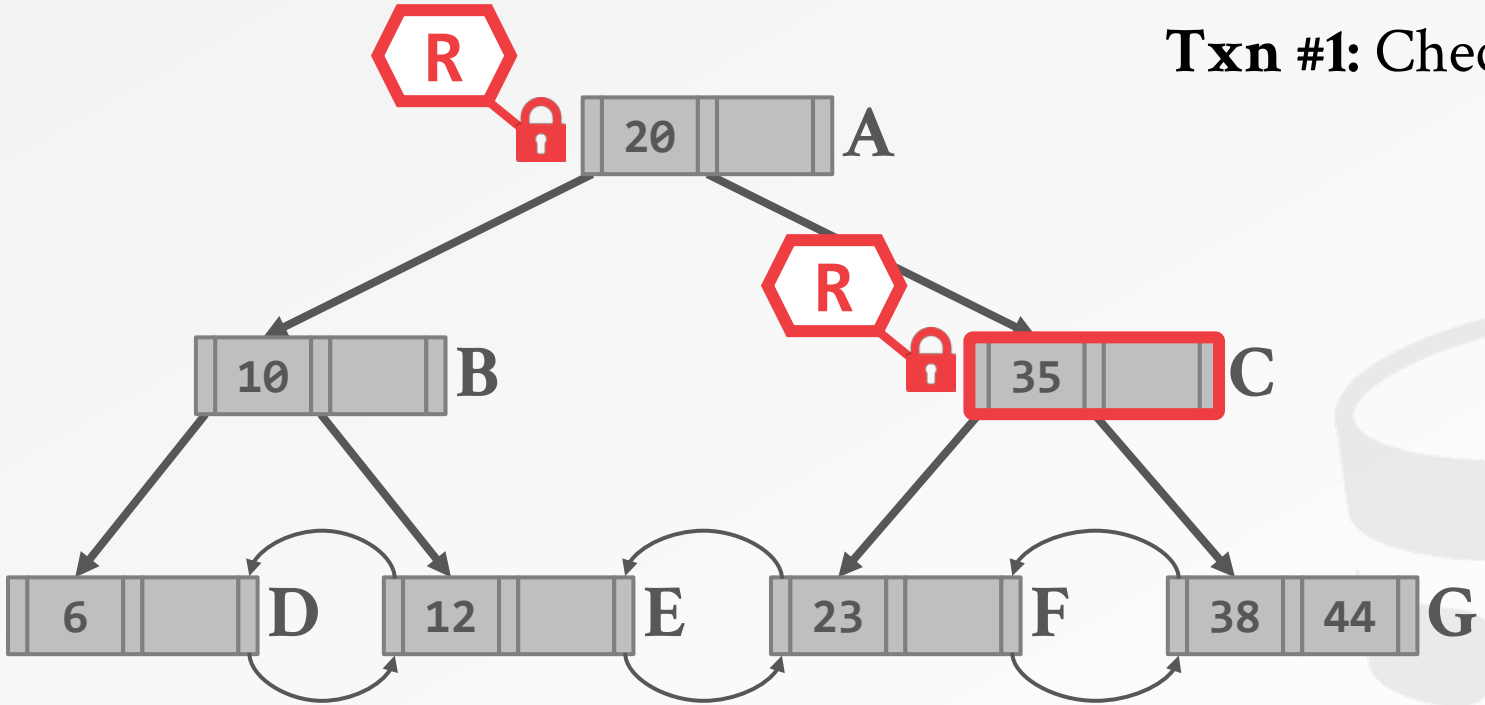


PROBLEM SCENARIO #1



Txn #1: Check if 25 exists

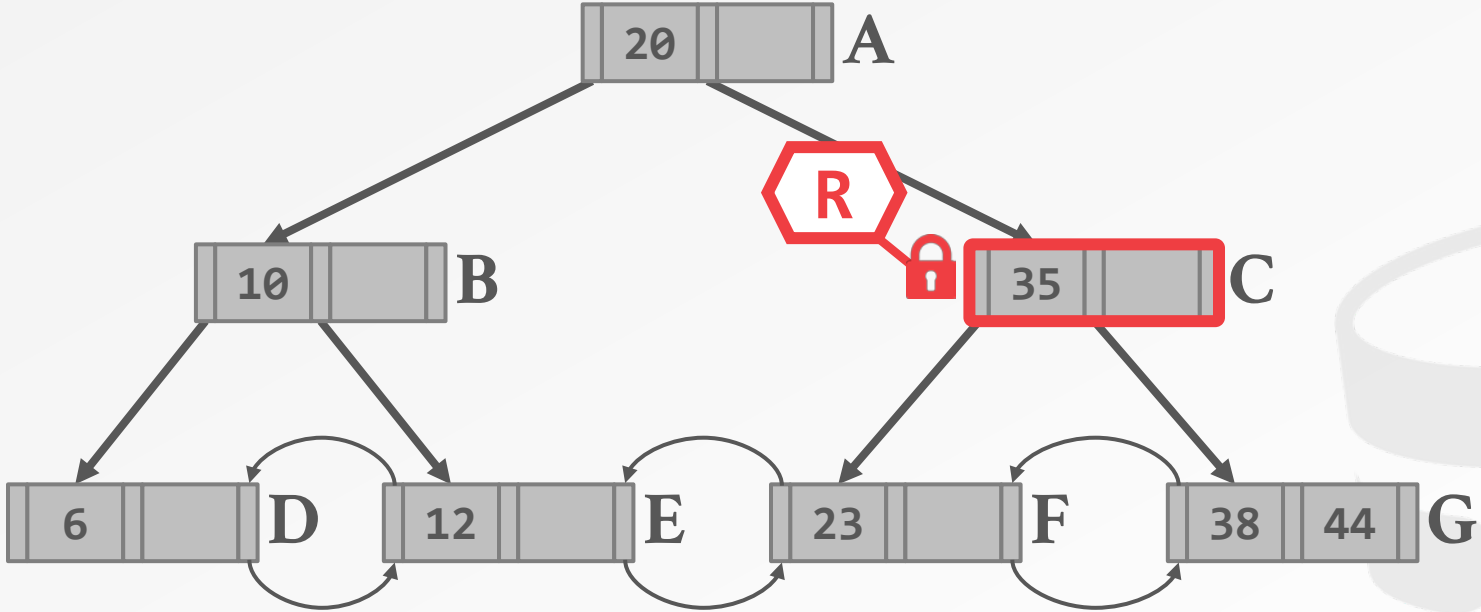
PROBLEM SCENARIO #1



Txn #1: Check if 25 exists

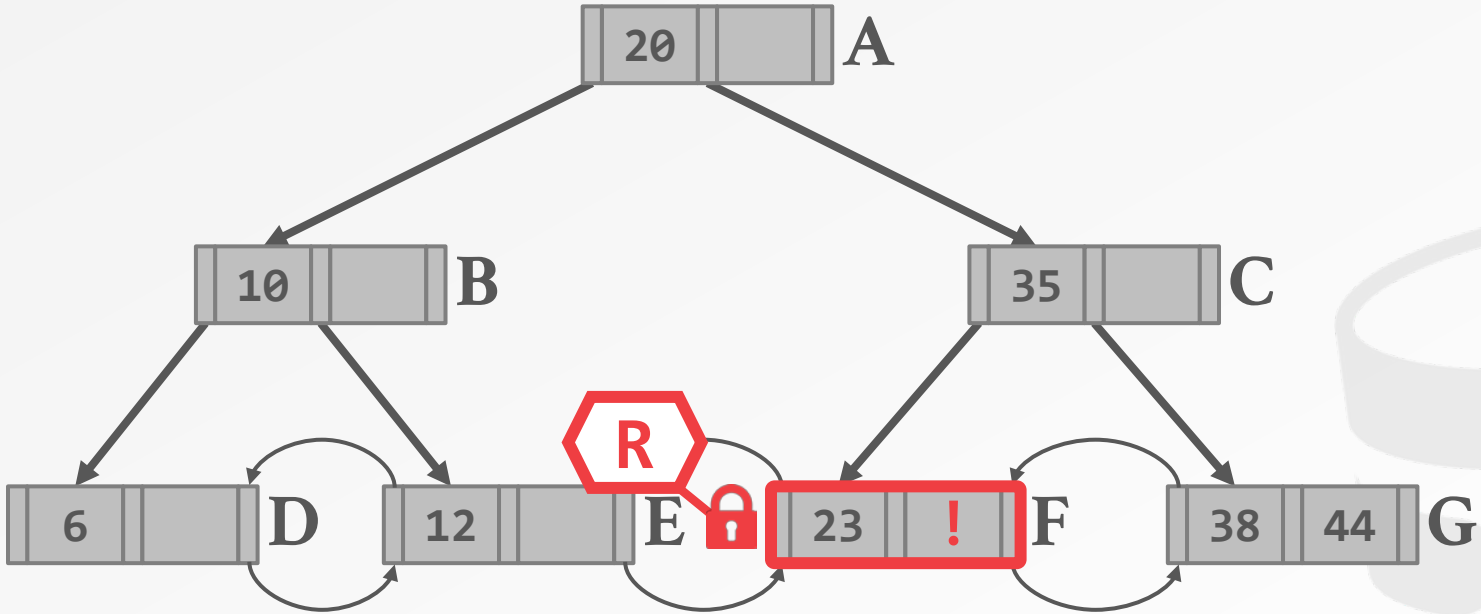
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists



PROBLEM SCENARIO #1

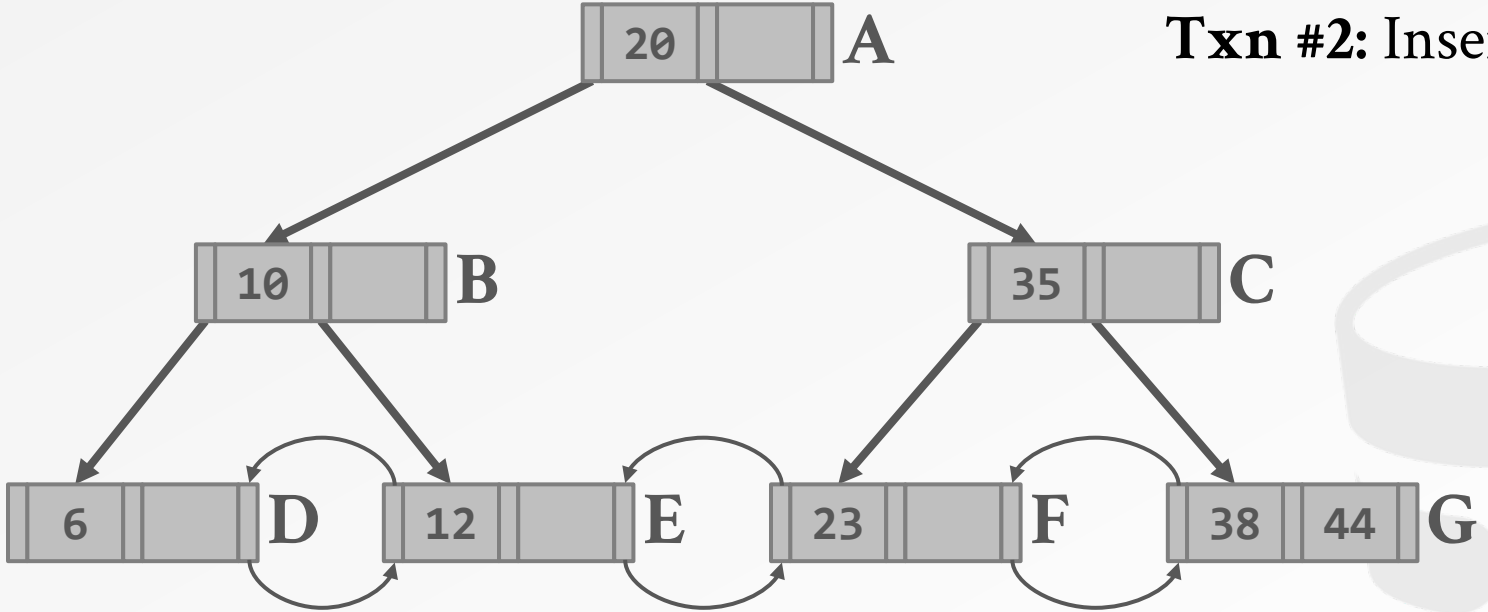
Txn #1: Check if 25 exists



PROBLEM SCENARIO #1

Txn #1: Check if 25 exists

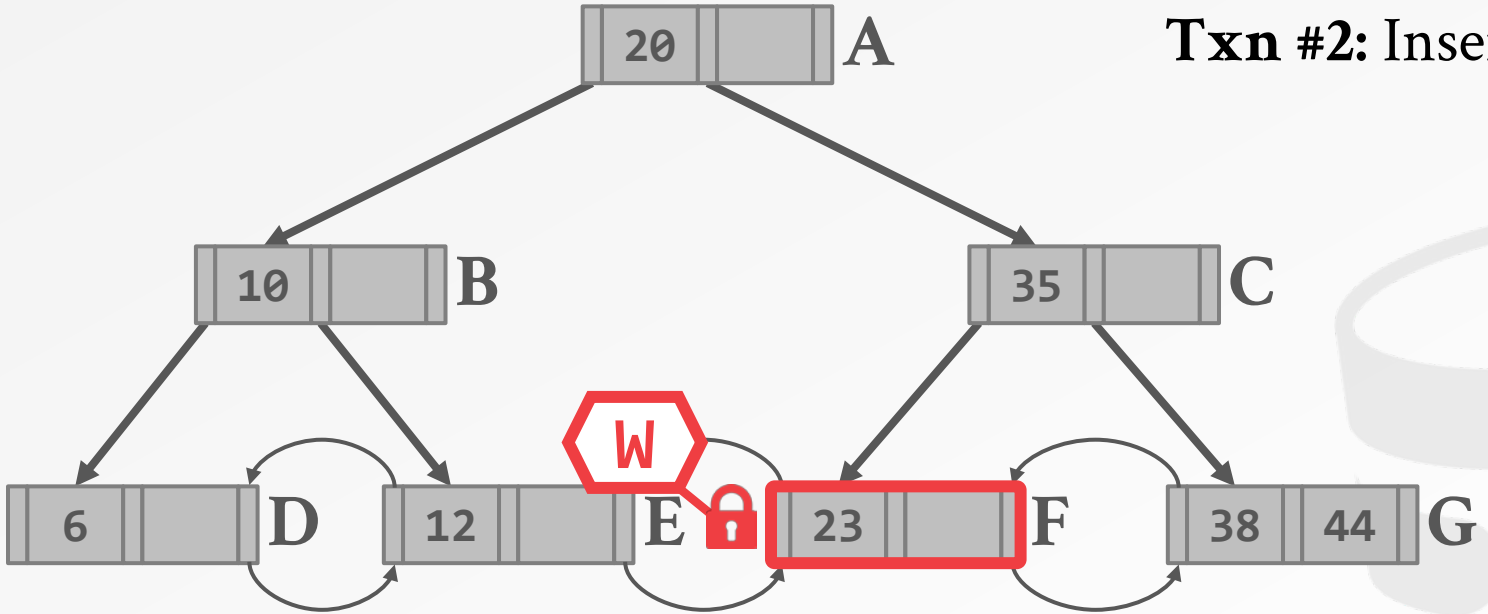
Txn #2: Insert 25



PROBLEM SCENARIO #1

Txn #1: Check if 25 exists

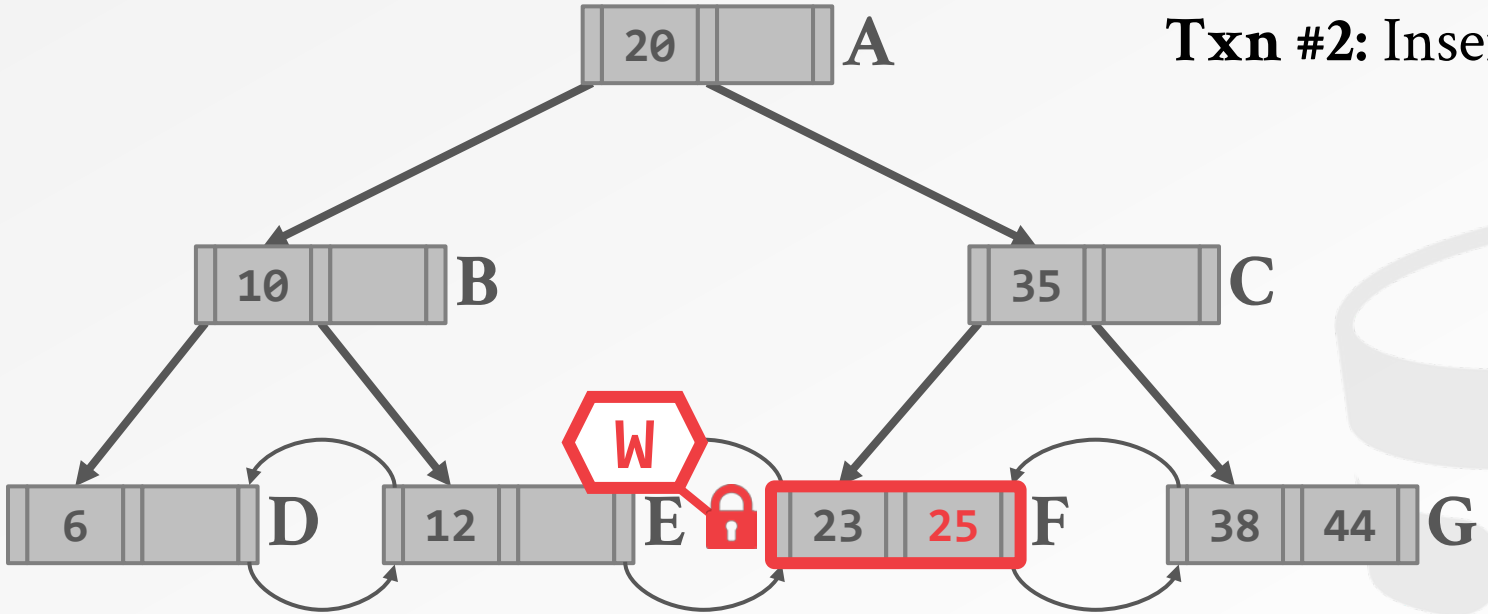
Txn #2: Insert 25



PROBLEM SCENARIO #1

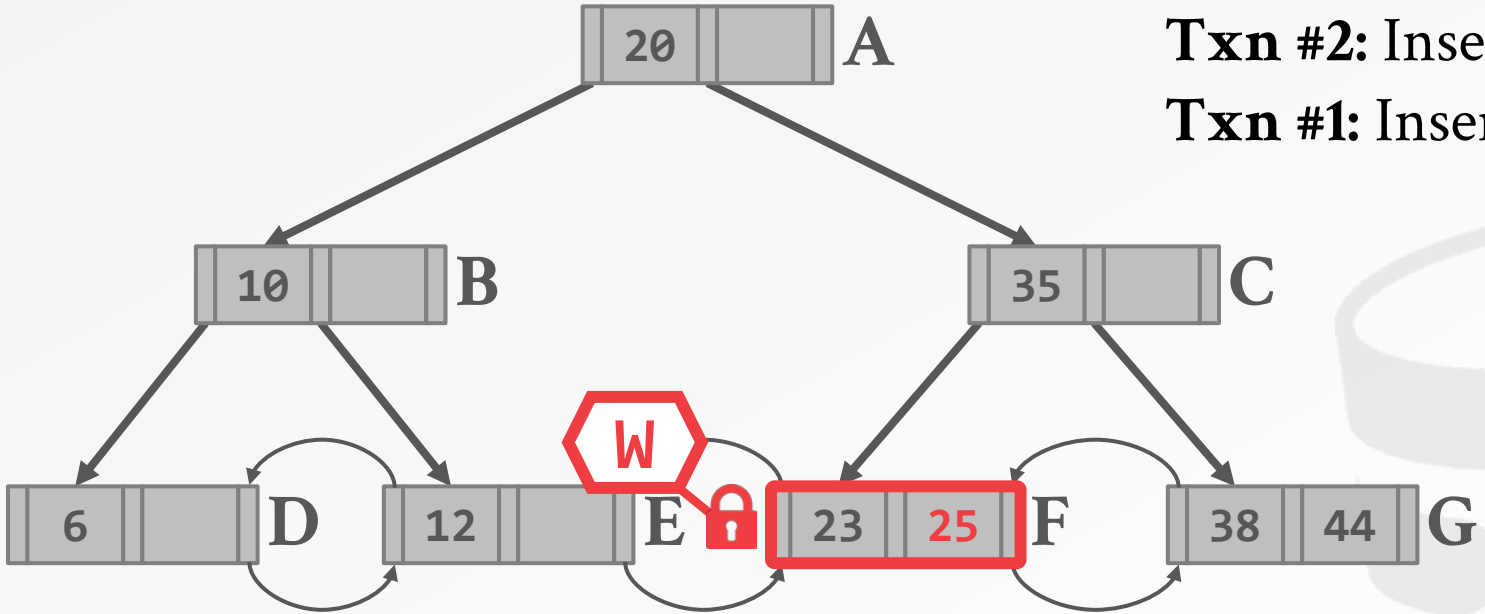
Txn #1: Check if 25 exists

Txn #2: Insert 25



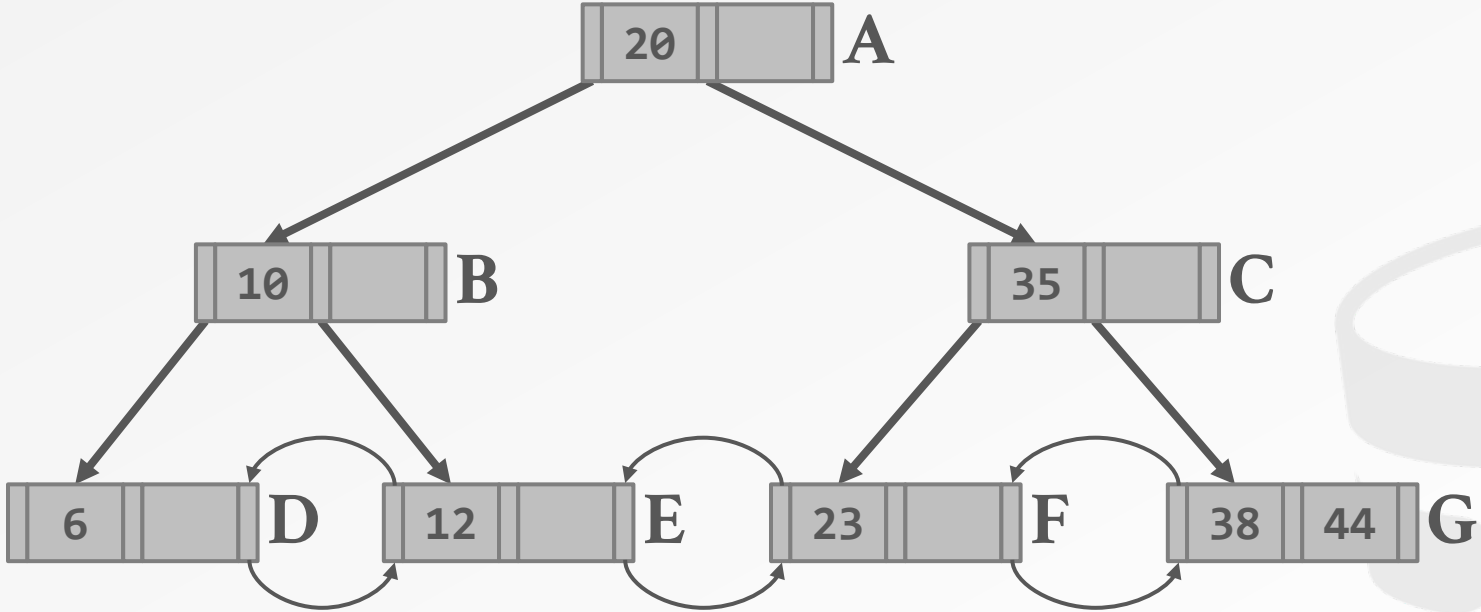
PROBLEM SCENARIO #1

Txn #1: Check if 25 exists
Txn #2: Insert 25
Txn #1: Insert 25



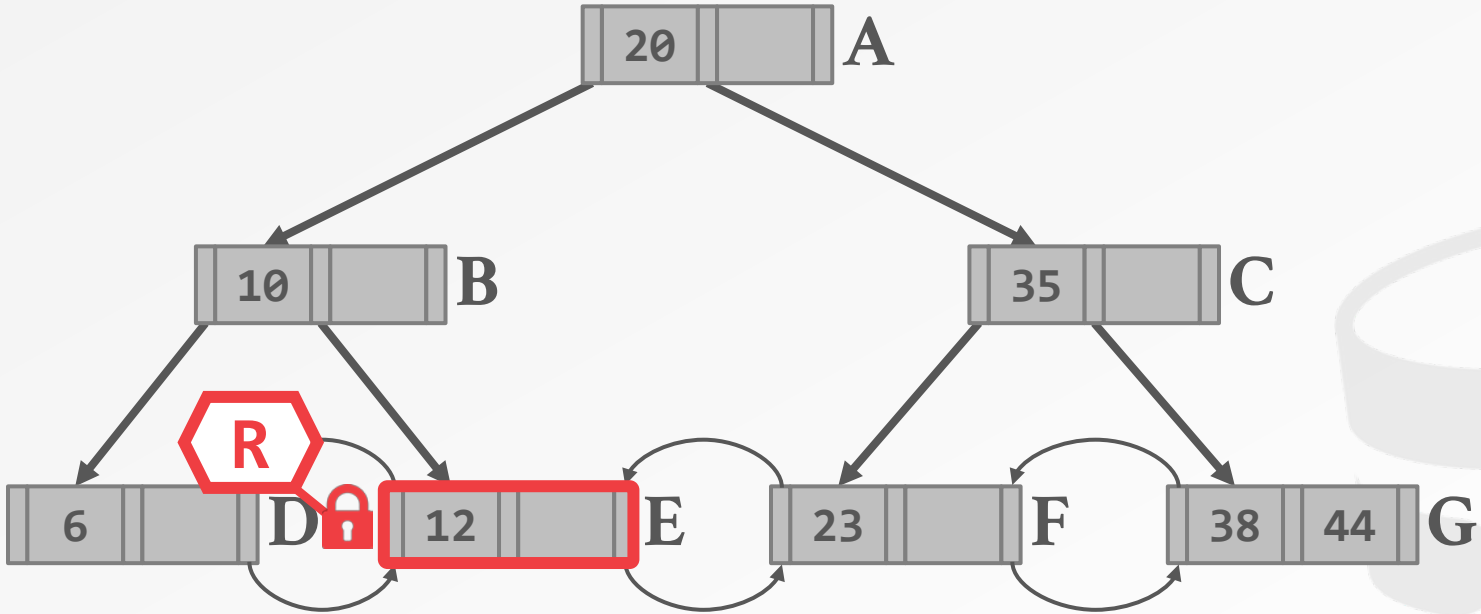
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]



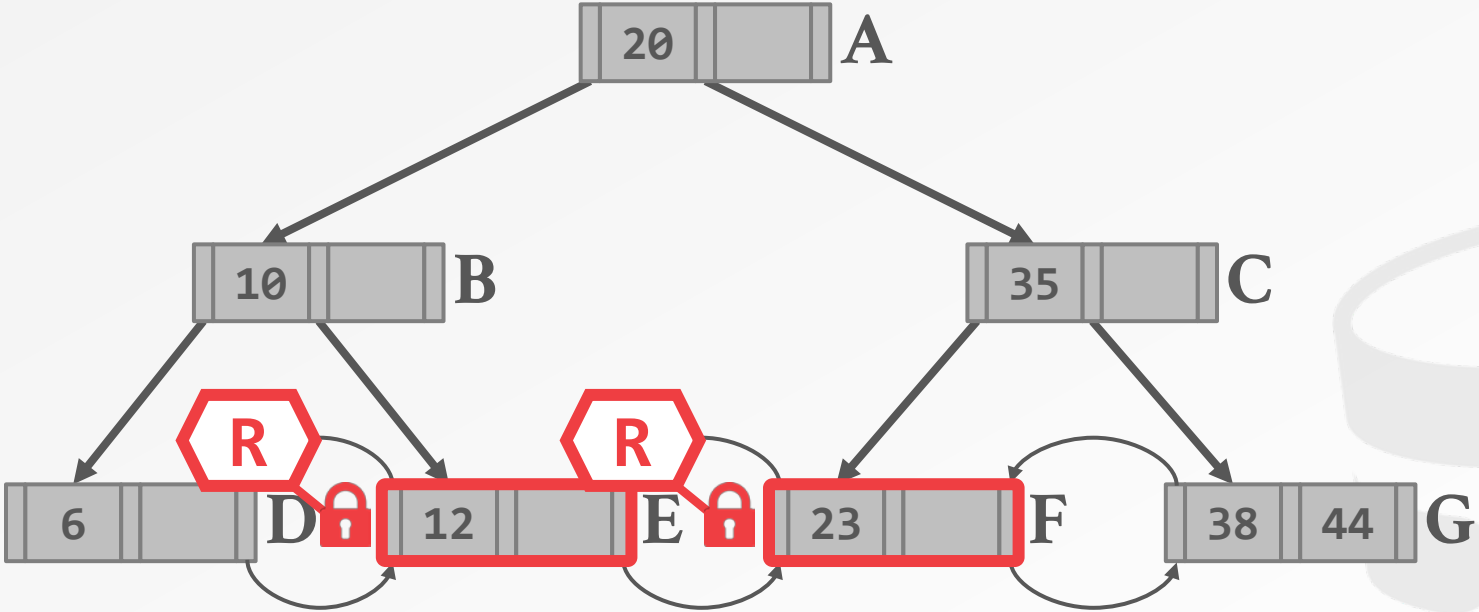
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]



PROBLEM SCENARIO #2

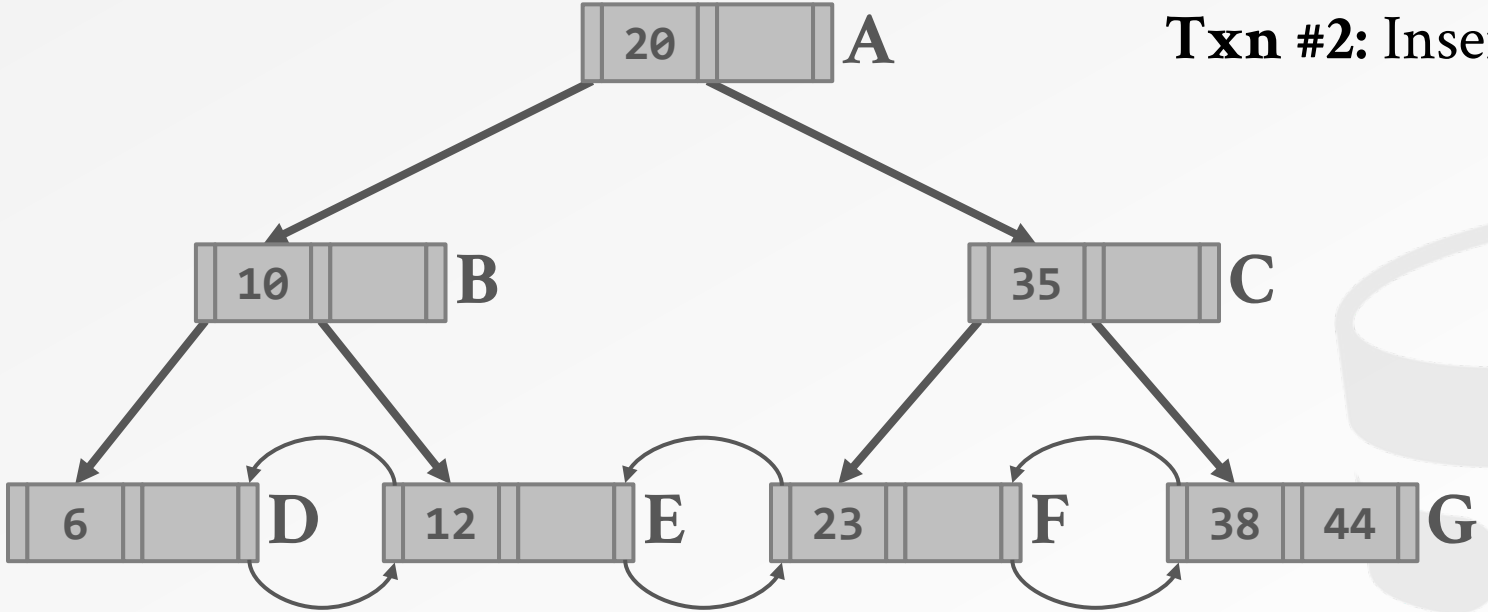
Txn #1: Scan [12, 23]



PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]

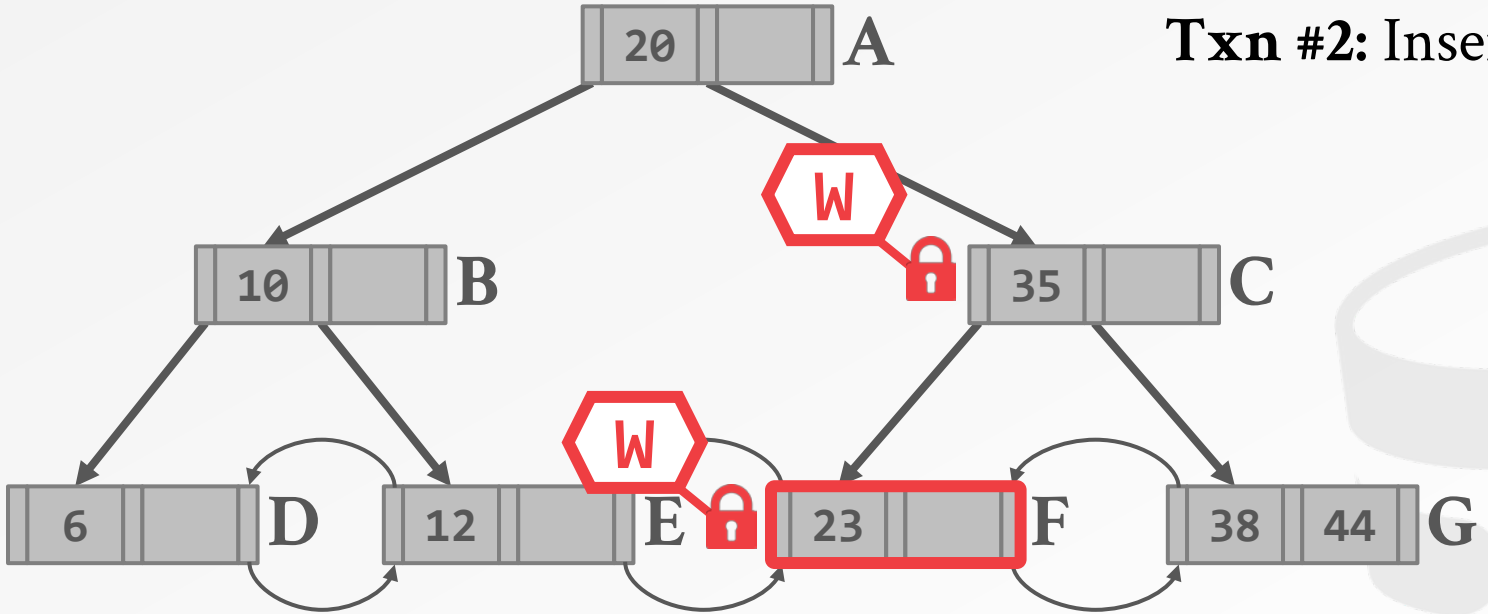
Txn #2: Insert 21



PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]

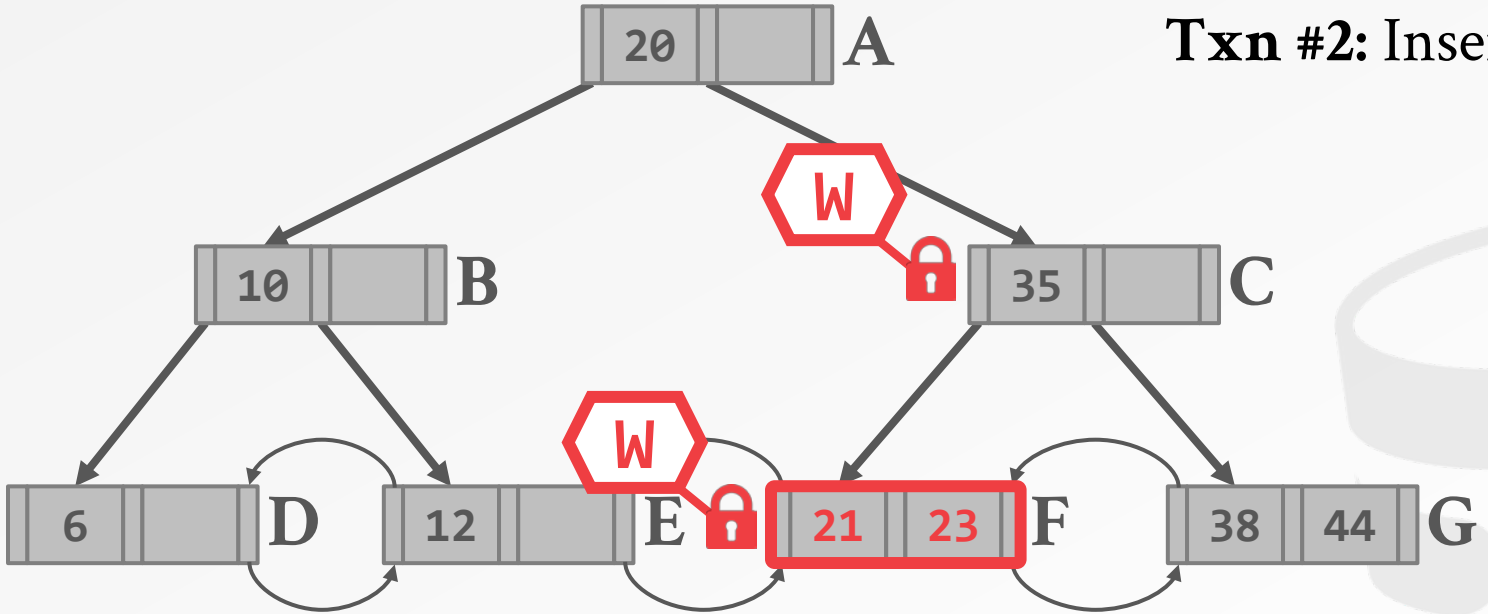
Txn #2: Insert 21



PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]

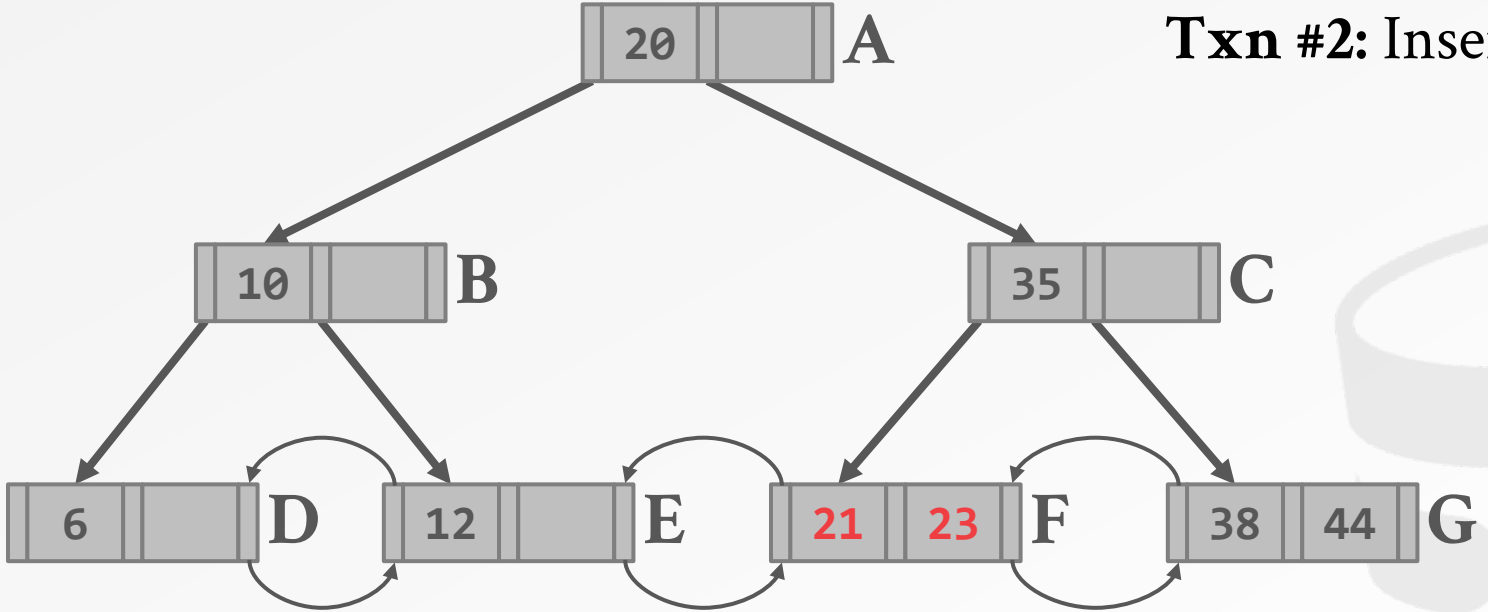
Txn #2: Insert 21



PROBLEM SCENARIO #2

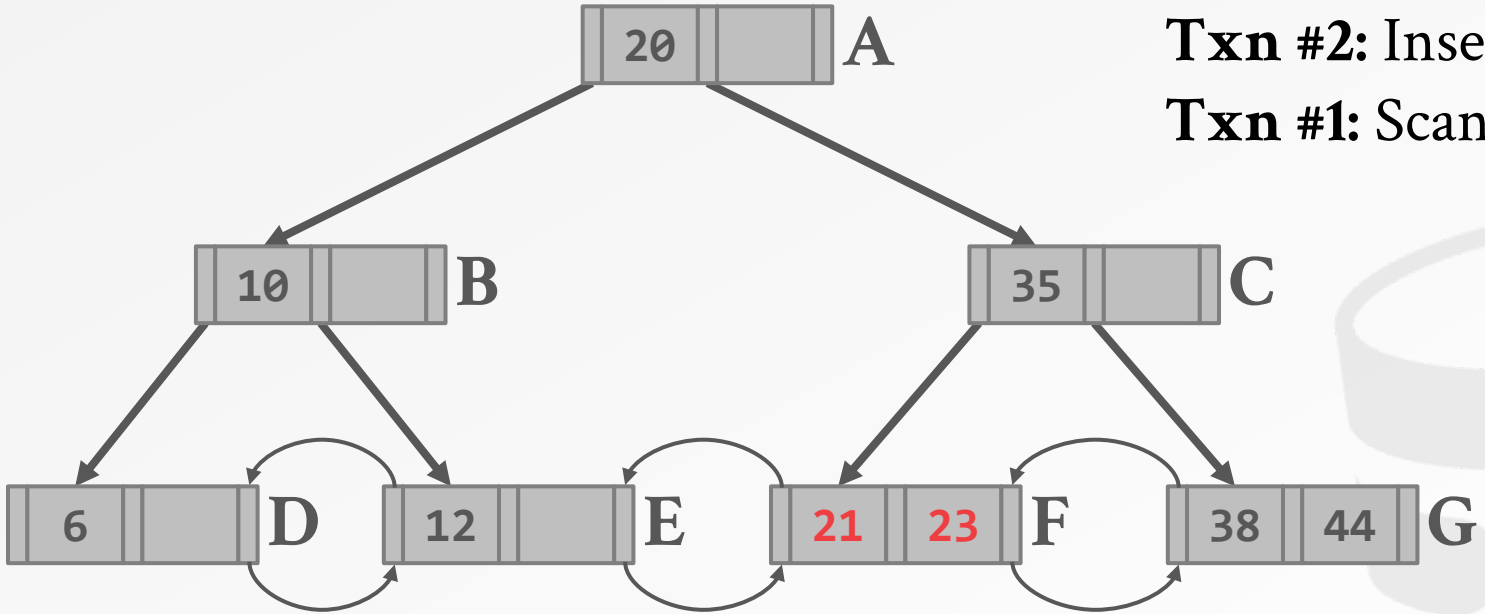
Txn #1: Scan [12, 23]

Txn #2: Insert 21



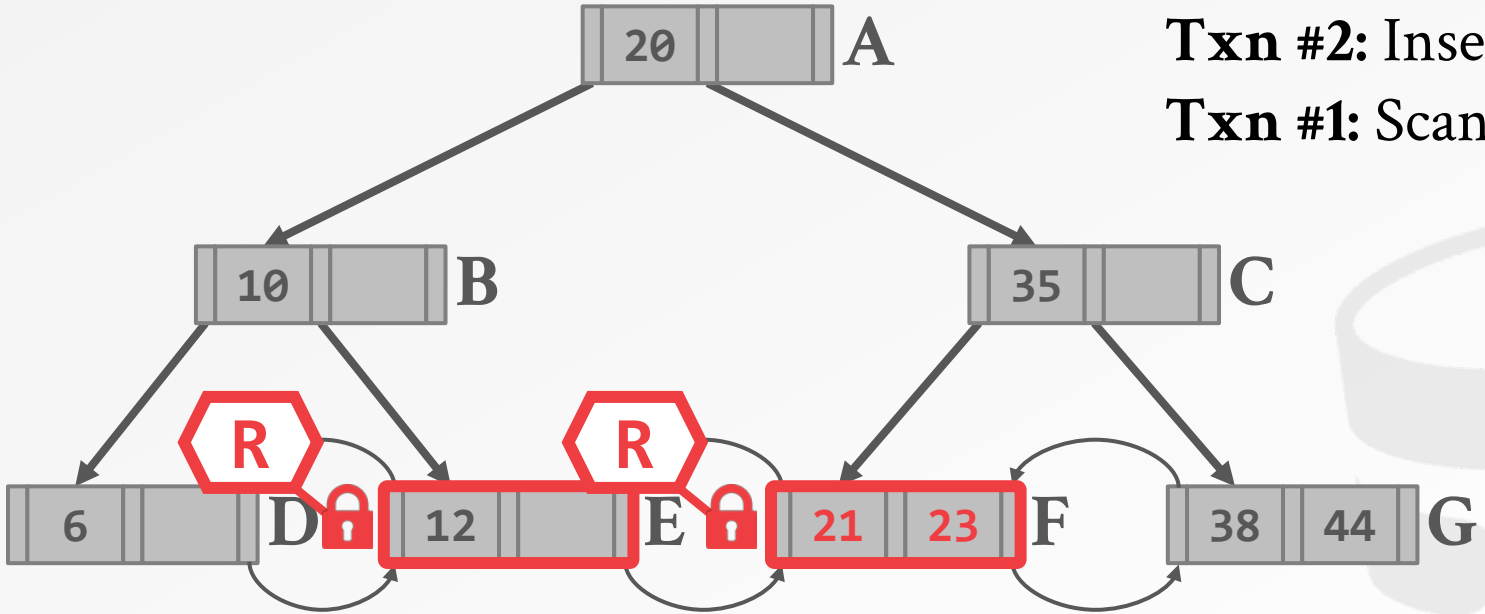
PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]
Txn #2: Insert 21
Txn #1: Scan [12, 23]



PROBLEM SCENARIO #2

Txn #1: Scan [12, 23]
Txn #2: Insert 21
Txn #1: Scan [12, 23]



INDEX LOCKS

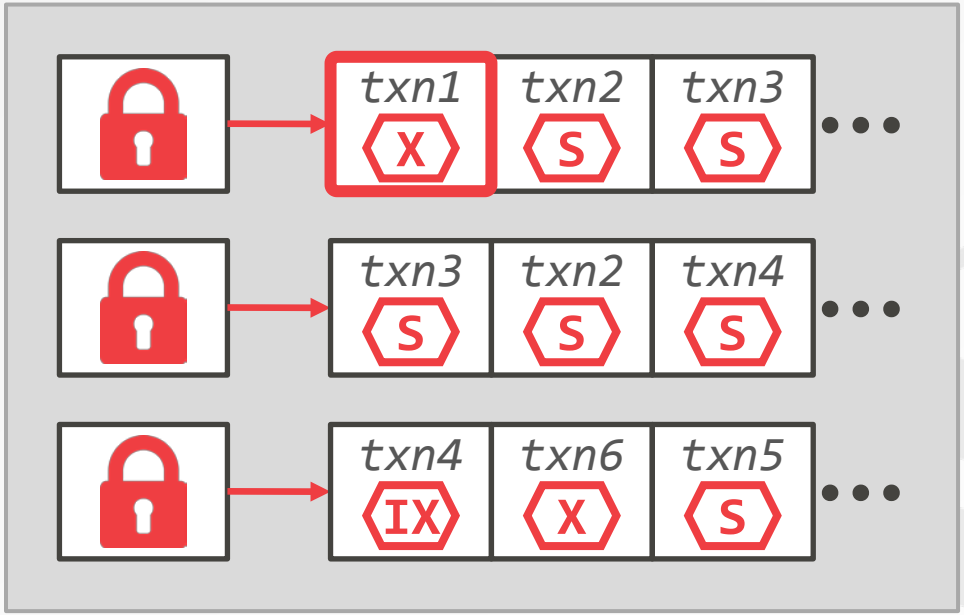
Need a way to protect the index's logical contents from other txns to avoid phantoms.

Difference with index latches:

→ Locks are held for the entire duration of a txn.
→ Only acquired at the leaf nodes.
→ Not physically stored in index data structure.

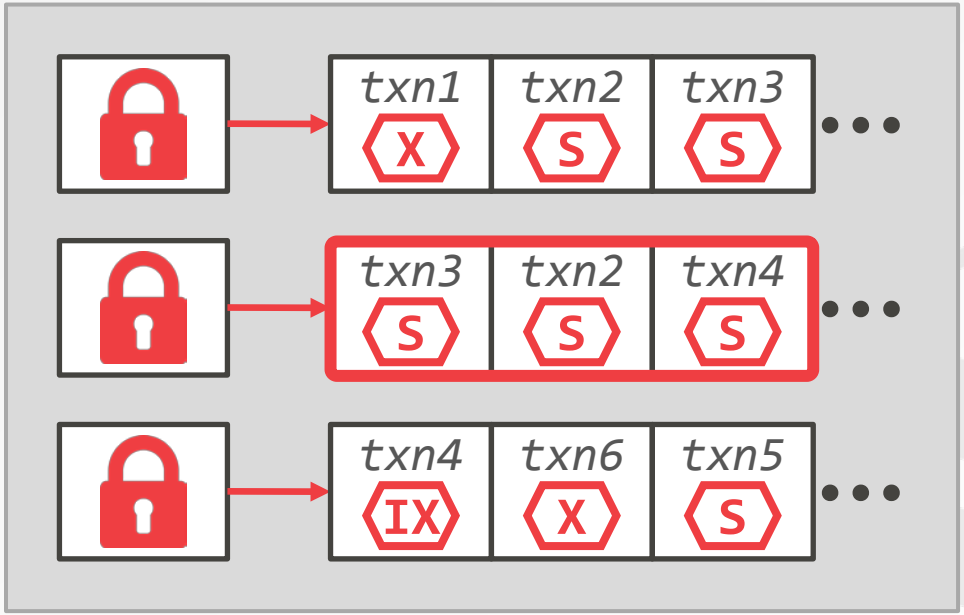
INDEX LOCKS

Lock Table



INDEX LOCKS

Lock Table



INDEX LOCKING SCHEMES

Predicate Locks

Key-Value Locks

Gap Locks

Key-Range Locks

Hierarchical Locking

PREDICATE LOCKS

Proposed locking scheme from System R.

→ Shared lock on the predicate in a **WHERE** clause of a **SELECT** query.

→ Exclusive lock on the predicate in a **WHERE** clause of any **UPDATE**, **INSERT**, or **DELETE** query.

Never implemented in any system.

THE NOTIONS OF CONSISTENCY AND PREDICATE LOCKS IN A DATABASE SYSTEM
CACM 1976

PREDICATE LOCKS

```
SELECT SUM(balance)
  FROM account
 WHERE name = 'Biggie'
```

```
INSERT INTO account
 (name, balance)
 VALUES ('Biggie', 100);
```

Records in Table 'account'

name='Biggie'

PREDICATE LOCKS

```
SELECT SUM(balance)
  FROM account
 WHERE name = 'Biggie'
```

```
INSERT INTO account
 (name, balance)
 VALUES ('Biggie', 100);
```

Records in Table 'account'

name='Biggie'

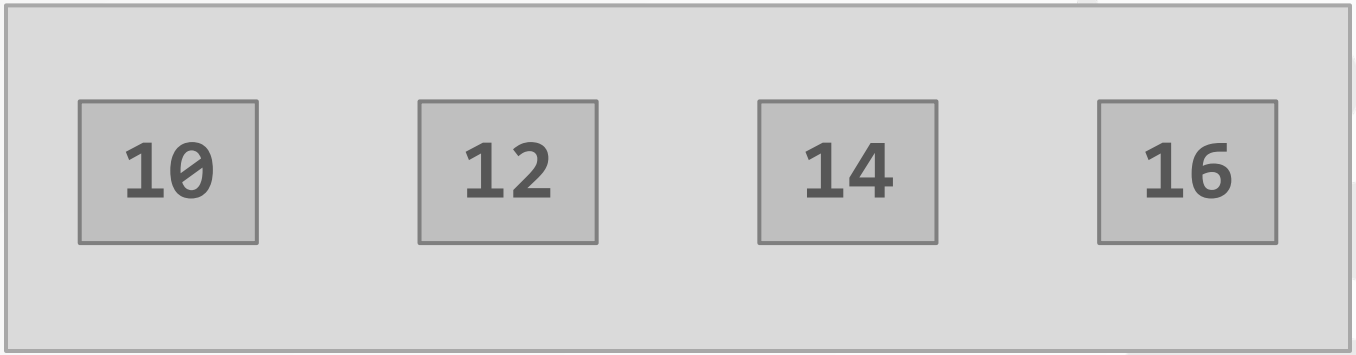
name='Biggie' ∧
balance=100

KEY-VALUE LOCKS

Locks that cover a single key value.

Need “virtual keys” for non-existent values.

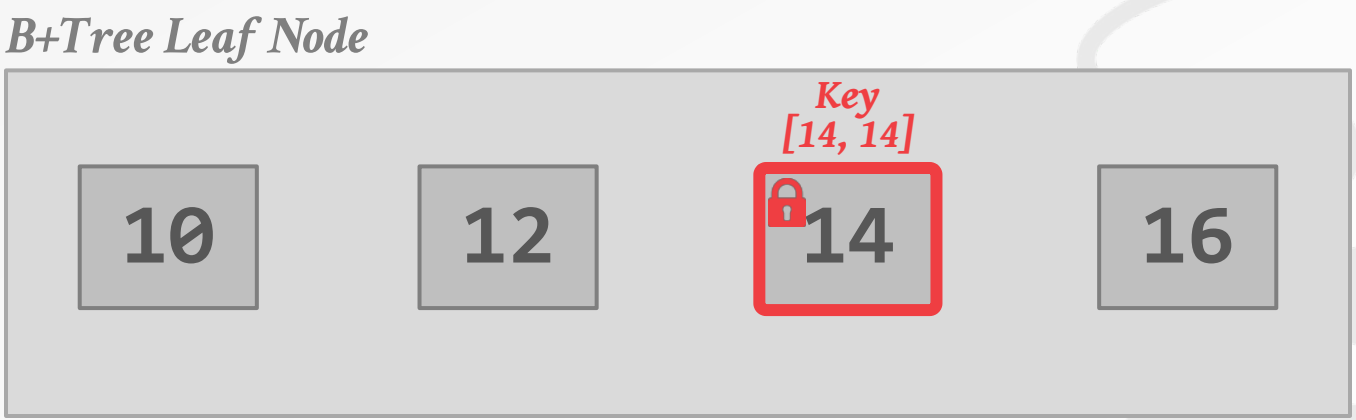
B+Tree Leaf Node



KEY-VALUE LOCKS

Locks that cover a single key value.

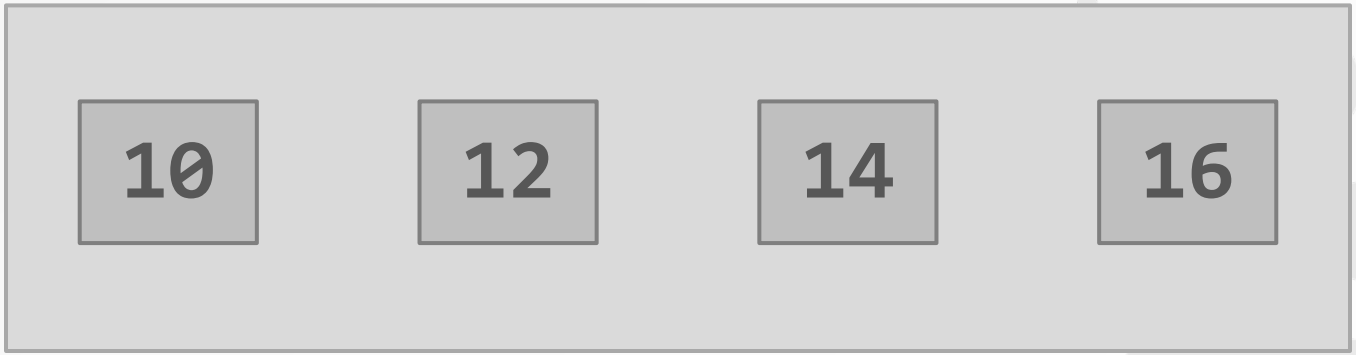
Need "virtual keys" for non-existent values.



GAP LOCKS

Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.

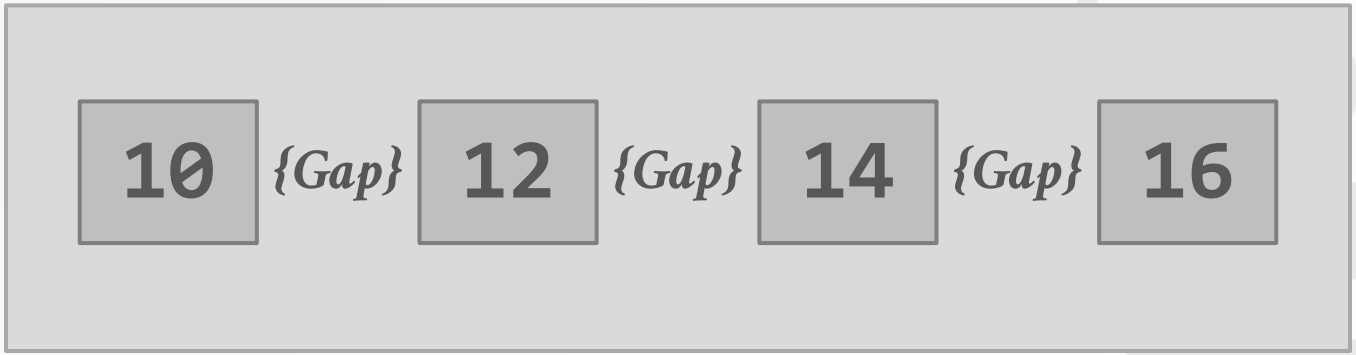
B+Tree Leaf Node



GAP LOCKS

Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.

B+Tree Leaf Node



GAP LOCKS

Each txn acquires a key-value lock on the single key that it wants to access. Then get a gap lock on the next key gap.

KEY-RANGE LOCKS

A txn takes locks on ranges in the key space.

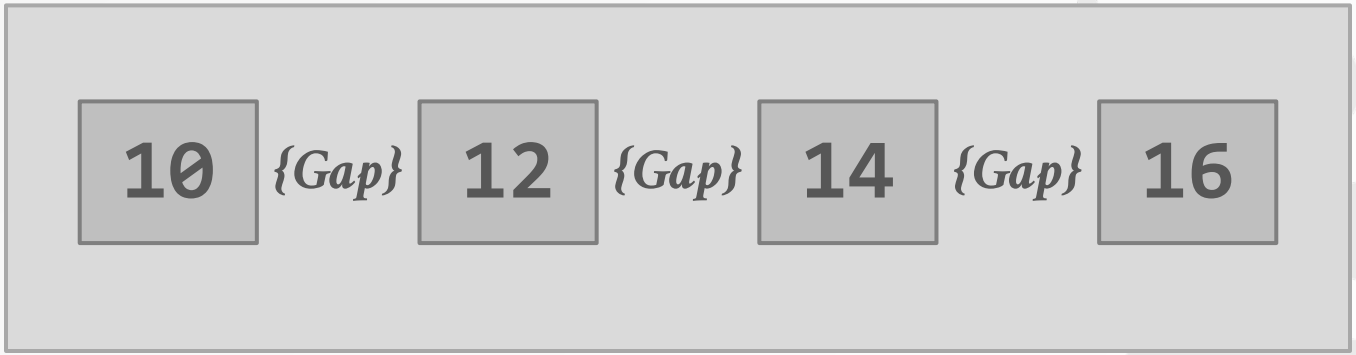
→ Each range is from one key that appears in the relation, to the next that appears.

→ Define lock modes so conflict table will capture commutativity of the operations available.

KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.

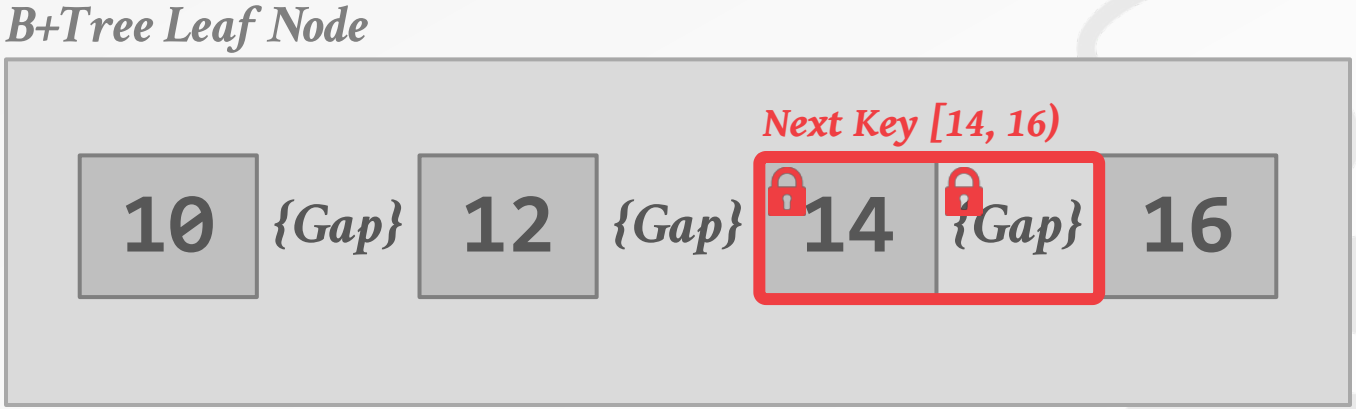
→ Need "virtual keys" for artificial values (infinity)



KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.

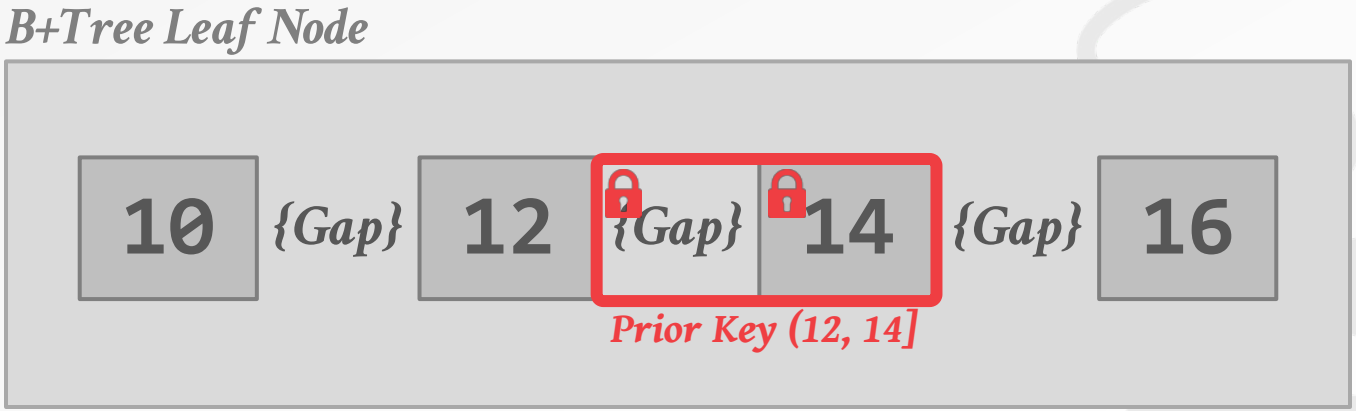
→ Need "virtual keys" for artificial values (infinity)



KEY-RANGE LOCKS

Locks that cover a key value and the gap to the next key value in a single index.

→ Need "virtual keys" for artificial values (infinity)

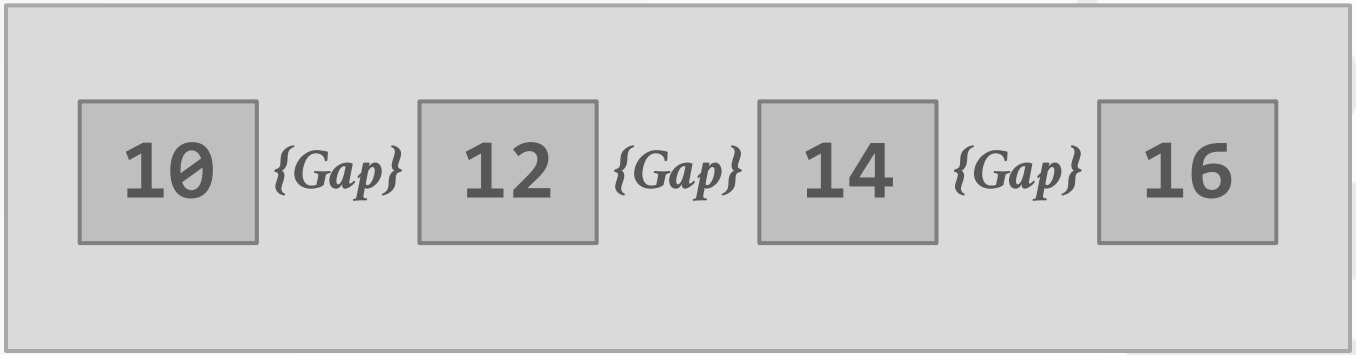


HIERARCHICAL LOCKING

Allow for a txn to hold wider key-range locks with different locking modes.

→ Reduces the number of visits to lock manager.

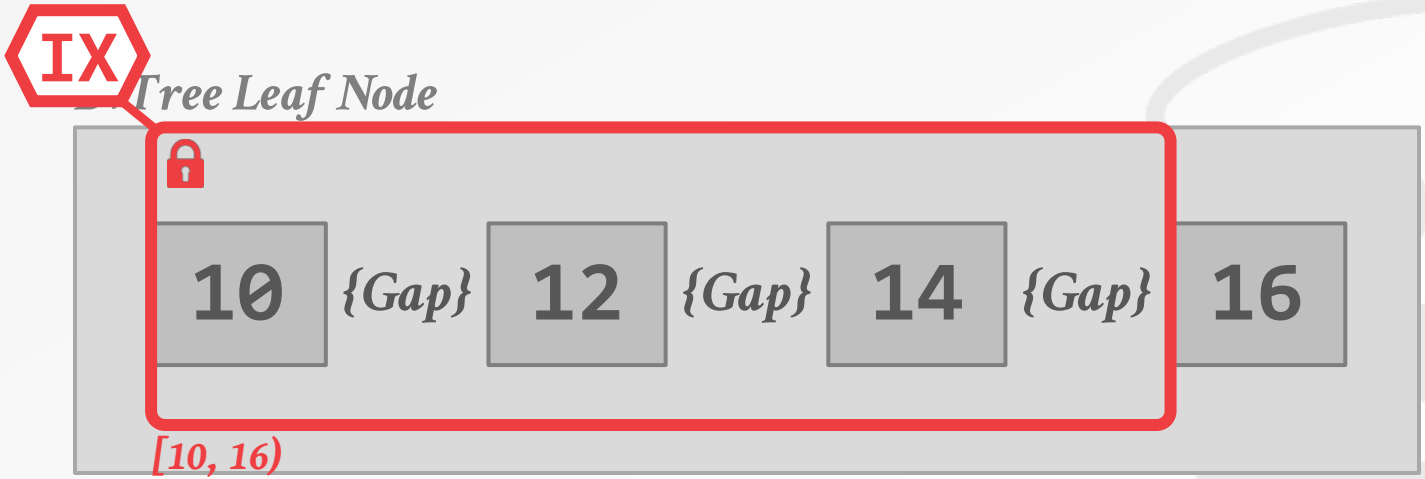
B+Tree Leaf Node



HIERARCHICAL LOCKING

Allow for a txn to hold wider key-range locks with different locking modes.

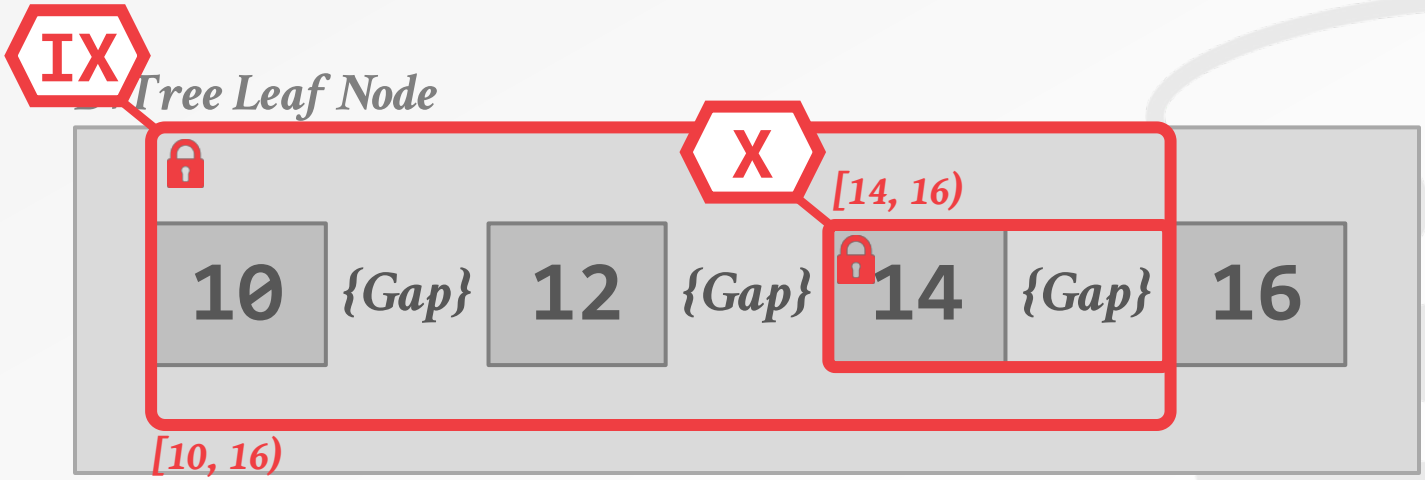
→ Reduces the number of visits to lock manager.



HIERARCHICAL LOCKING

Allow for a txn to hold wider key-range locks with different locking modes.

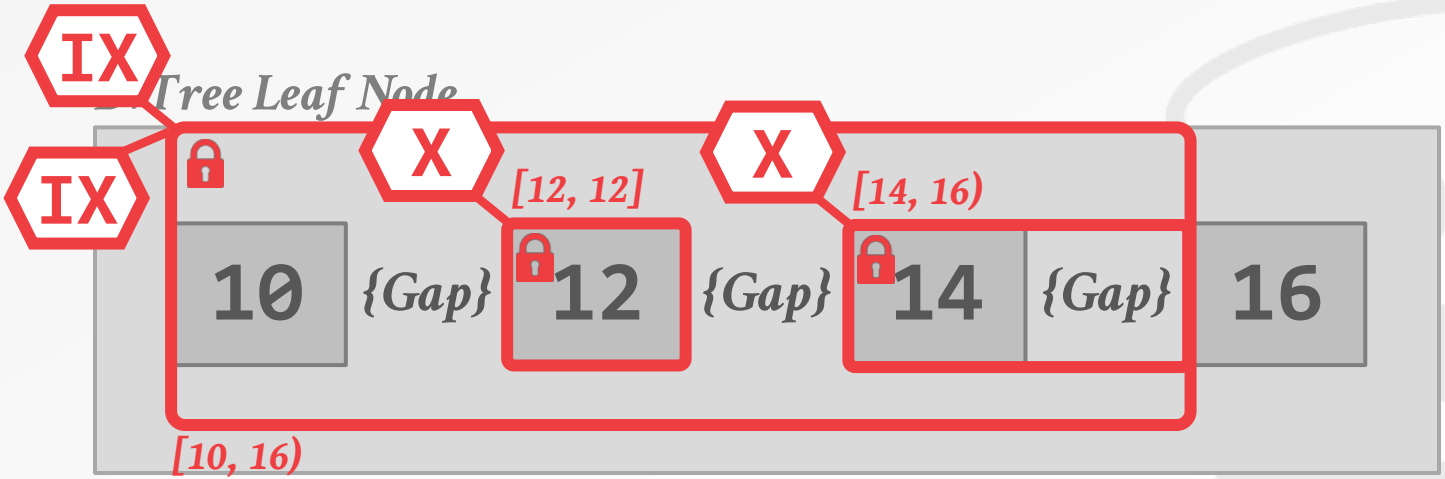
→ Reduces the number of visits to lock manager.



HIERARCHICAL LOCKING

Allow for a txn to hold wider key-range locks with different locking modes.

→ Reduces the number of visits to lock manager.



PARTING THOUGHTS

Hierarchical locking essentially provides predicate locking without complications.

→ Index locking occurs only in the leaf nodes.

→ Latching is to ensure consistent data structure.

Peloton currently does not support serializable isolation with range scans.

NEXT CLASS

Index Key Representation
Memory Allocation & Garbage Collection
T-Trees (1980s / TimesTen)
Concurrent Skip Lists (MemSQL)