

Lecture #06

Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Index Locking & Latching

@Andy_Pavlo // 15-721 // Spring 2019



TODAY'S AGENDA

Index Locks vs. Latches

Latch Implementations

Index Latching (Logical)

Index Locking (Physical)



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

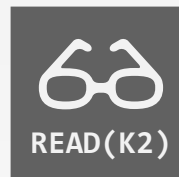
We already know how to use locks to protect objects in the database.

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:



SIMPLE EXAMPLE



Txn #1:



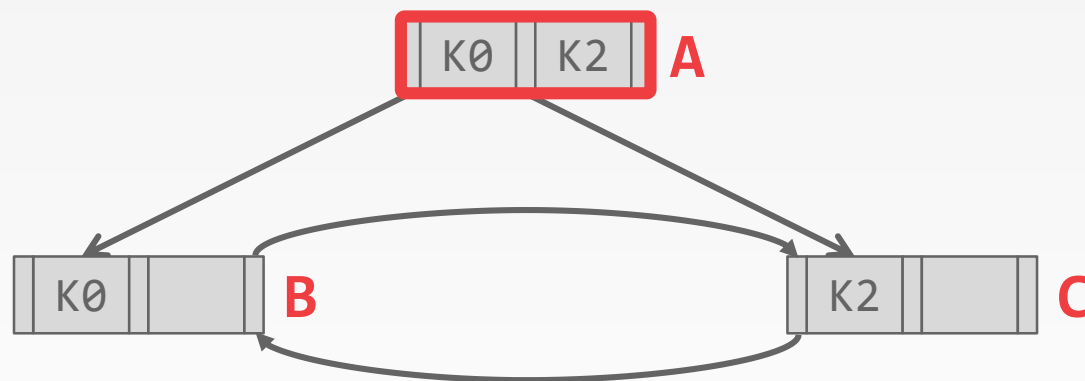
READ(K2)

Txn #2:



INSERT(K1)

SIMPLE EXAMPLE



Txn #1:



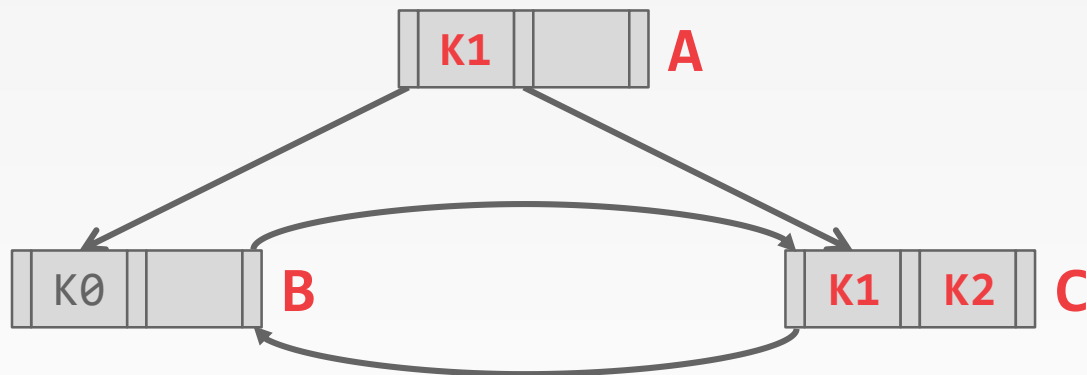
READ(K2)

Txn #2:



INSERT(K1)

SIMPLE EXAMPLE



Txn #1:



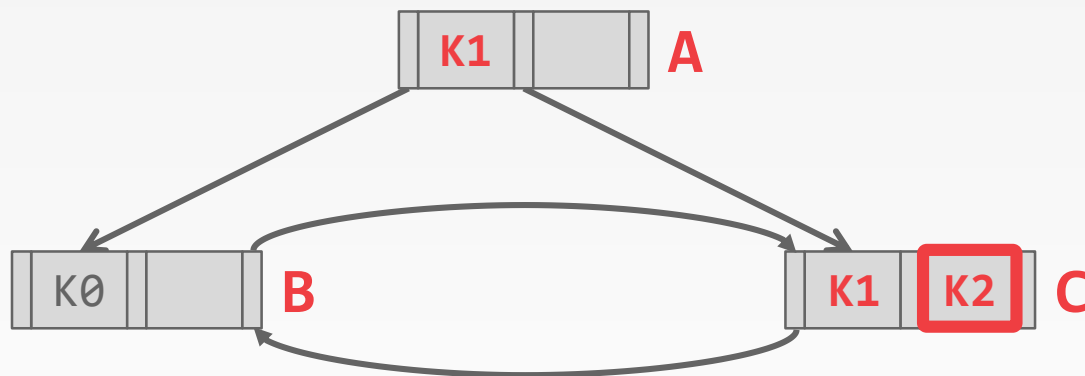
READ(K2)

Txn #2:



INSERT(K1)

SIMPLE EXAMPLE



Txn #1:



READ(K2)

Txn #2:



INSERT(K1)

Txn #1:



READ(K2)

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.



LOCKS VS. LATCHES

	<i>Locks</i>	<i>Latches</i>
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

Source: [Anastasia Ailamaki](#)



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



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

Address

New Value

Compare Value

COMPARE-AND-SWAP

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

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

→ Otherwise operation fails

M

30

Address *New Value*

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

Compare Value



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?  
}
```

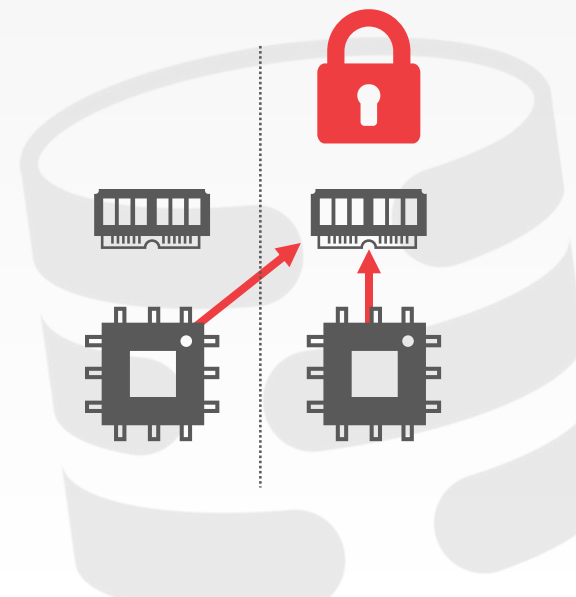
LATCH IMPLEMENTATIONS

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*>`

Base Latch



LATCH IMPLEMENTATIONS

Mellor-Crummey and Scott

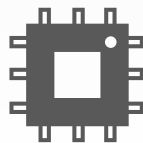
Choice #3: Queue-based Spinlock (MCS)

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

Base Latch



CPU1 Latch



CPU1

LATCH IMPLEMENTATIONS

Mellor-Crummey and Scott

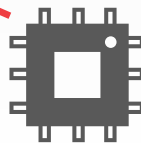
Choice #3: Queue-based Spinlock (MCS)

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

Base Latch



CPU1 Latch



CPU1

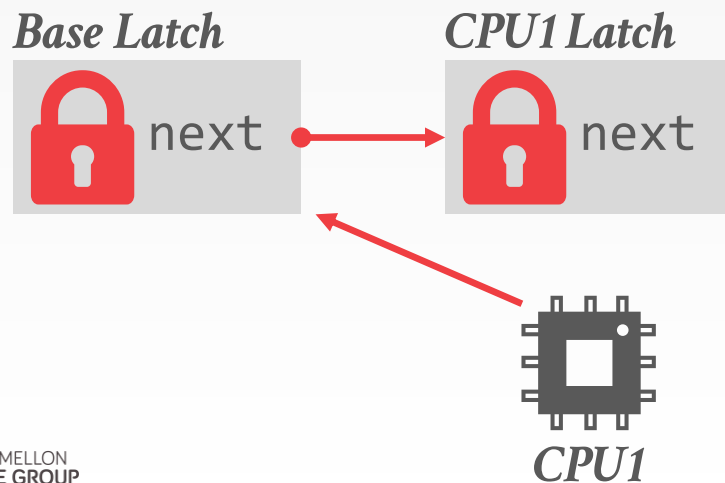


LATCH IMPLEMENTATIONS

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

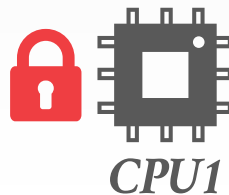
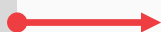
Choice #3: Queue-based Spinlock (MCS)

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

Base Latch



CPU1 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*>`

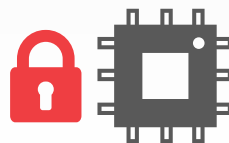
Base Latch



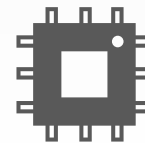
CPU1 Latch



CPU2 Latch



CPU1



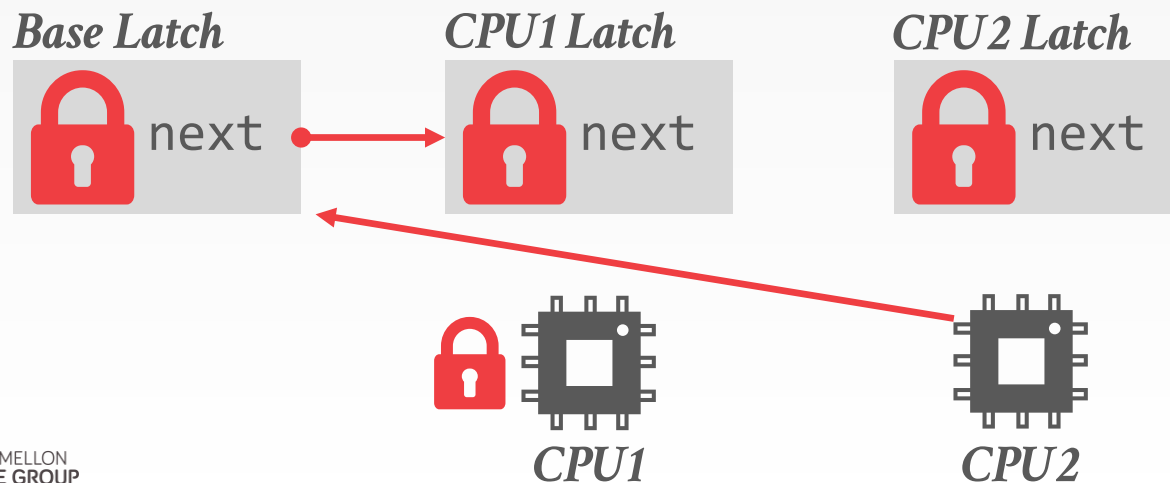
CPU2

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*>`

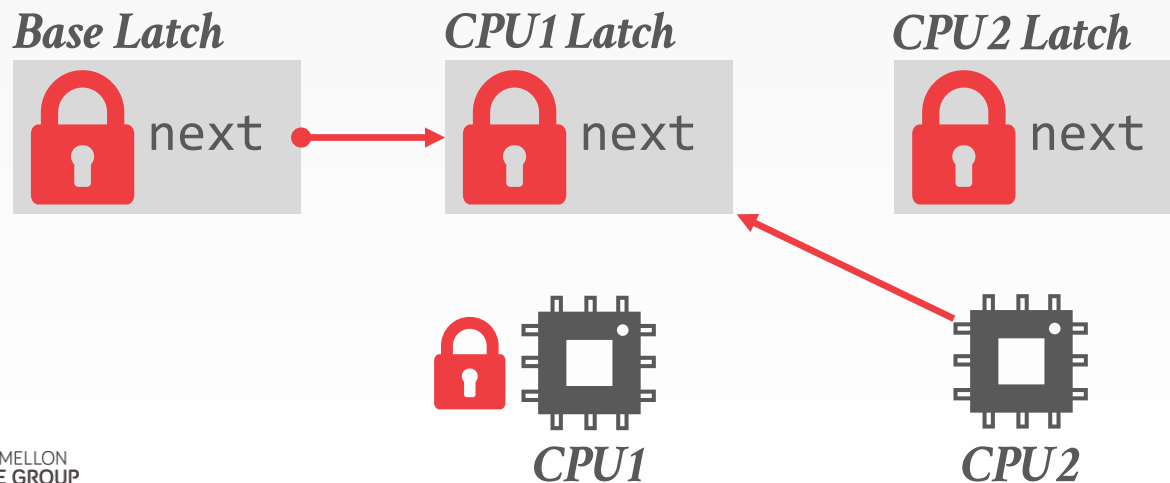


LATCH IMPLEMENTATIONS

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*>`

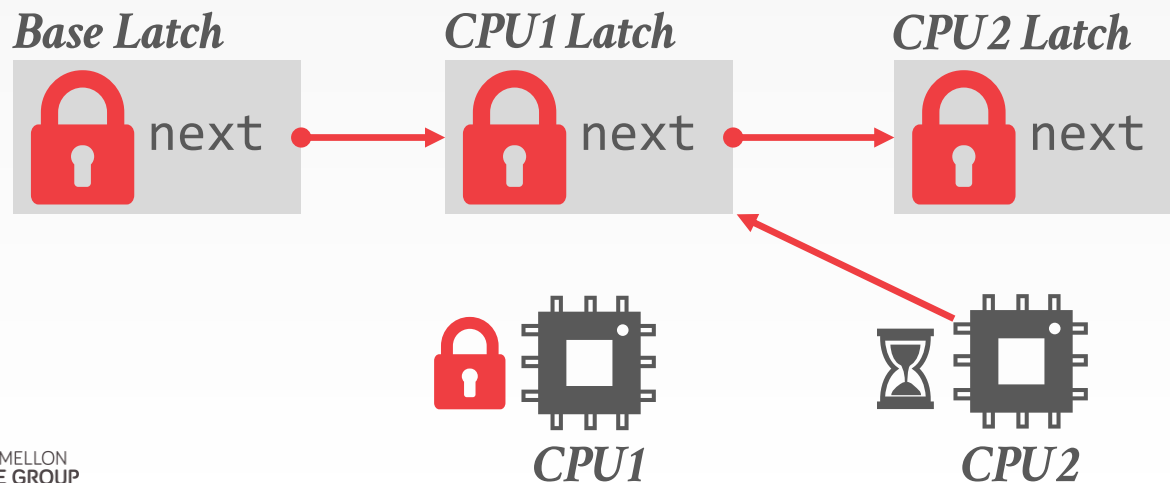


LATCH IMPLEMENTATIONS

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*>`

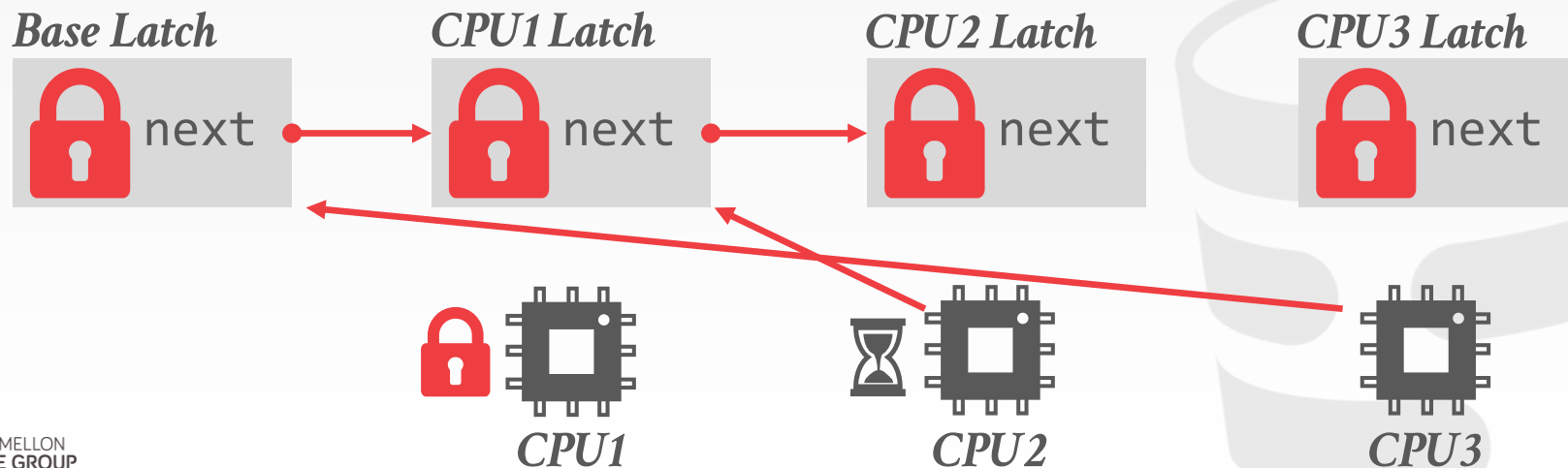


LATCH IMPLEMENTATIONS

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*>`

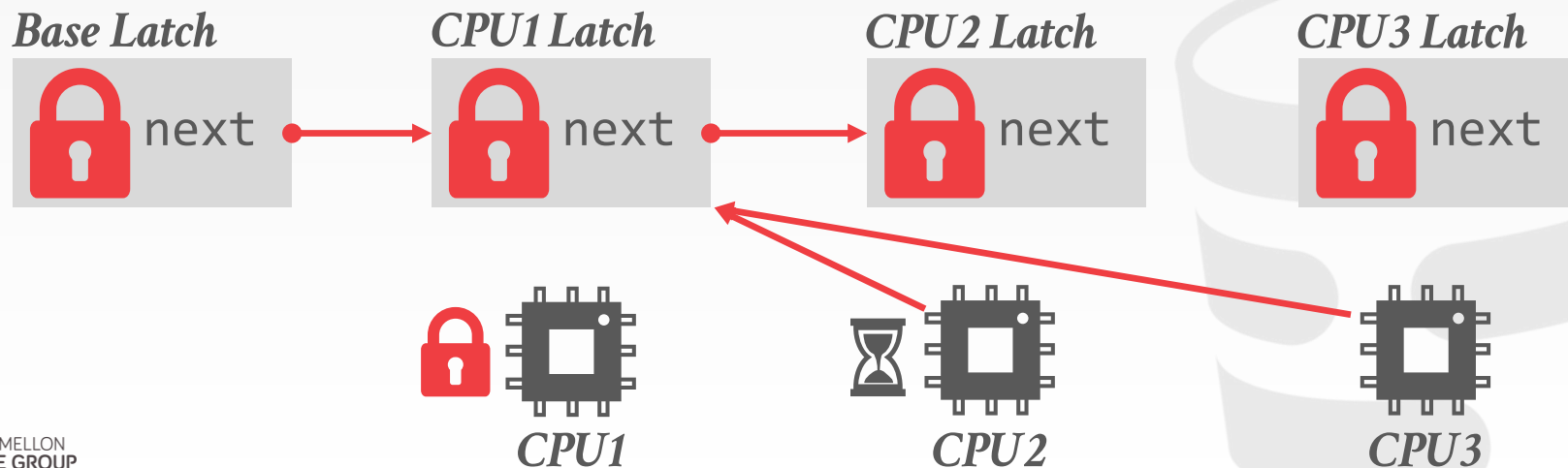


LATCH IMPLEMENTATIONS

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*>`

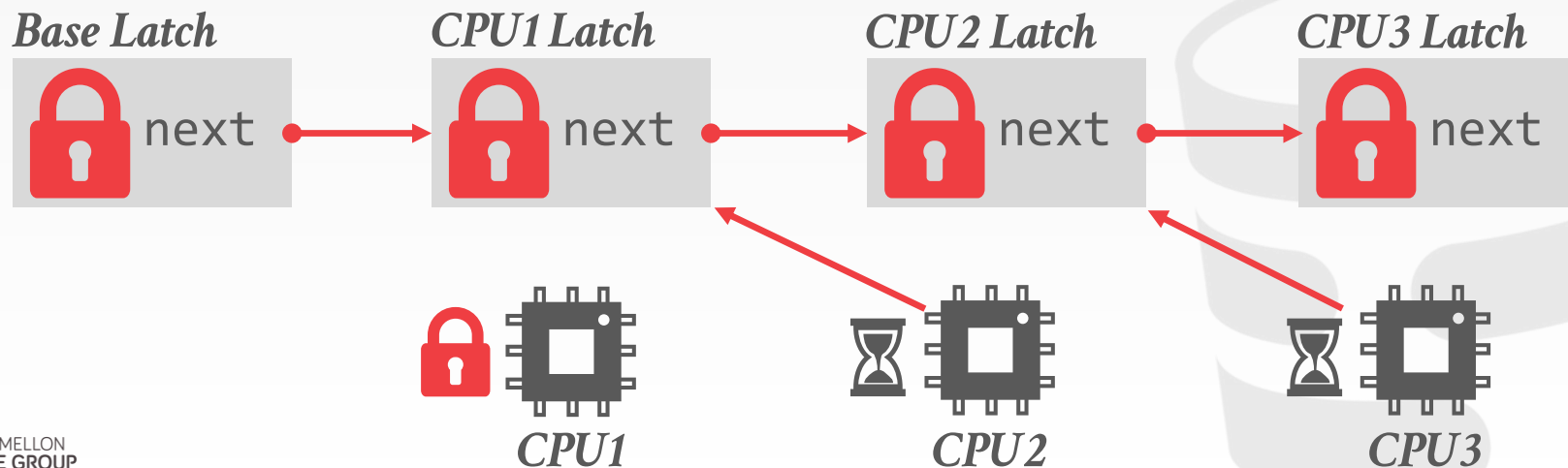


LATCH IMPLEMENTATIONS

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*>`

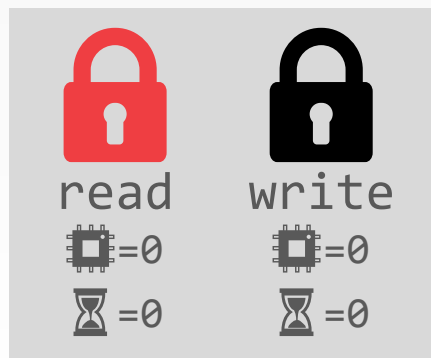


LATCH IMPLEMENTATIONS

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

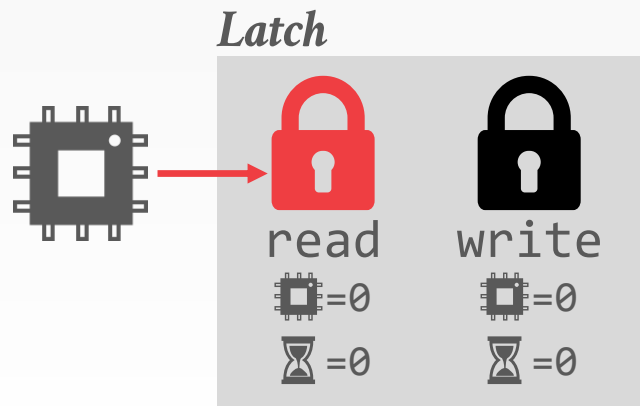
Latch



LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

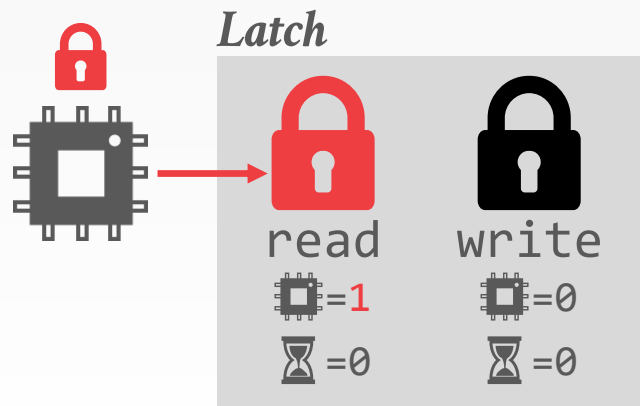
- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

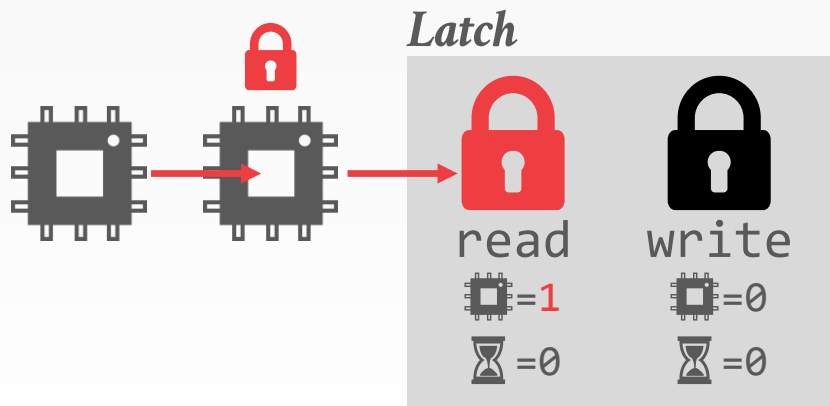
- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

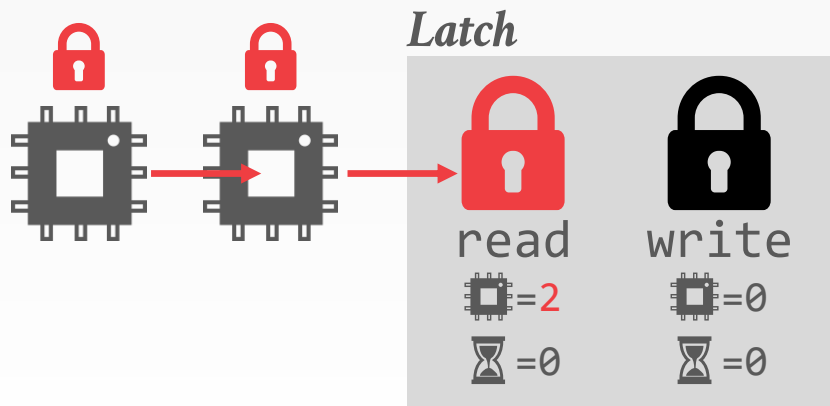
- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

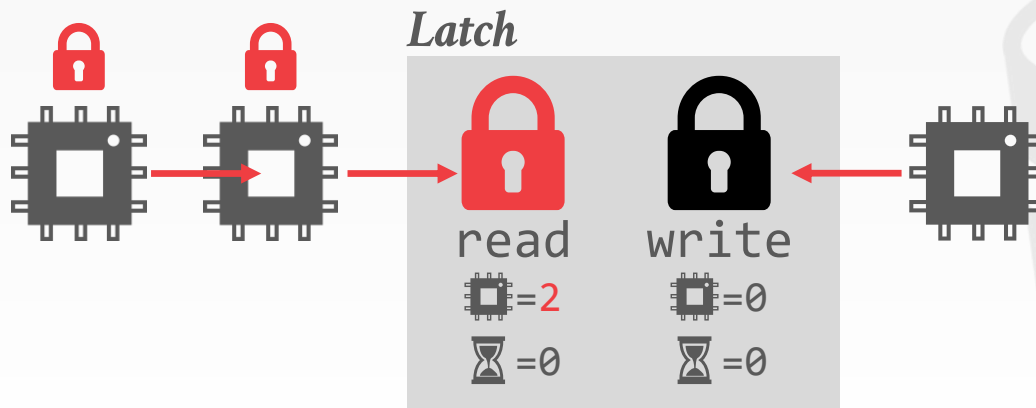
- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

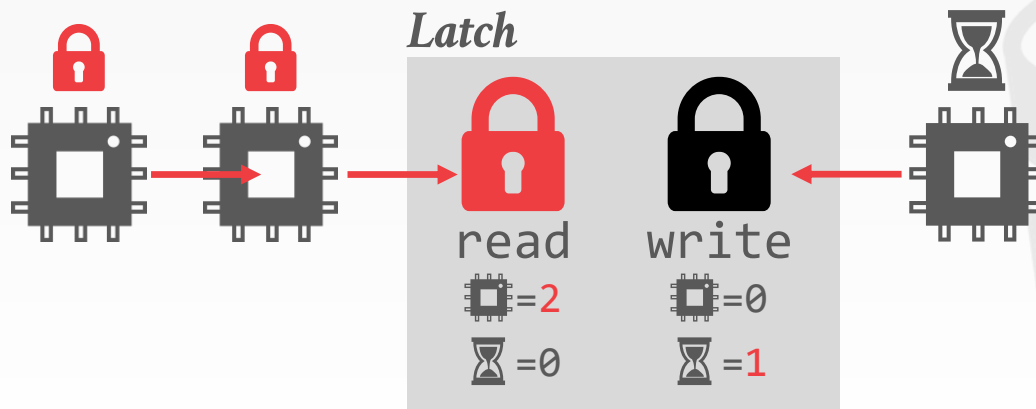
- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

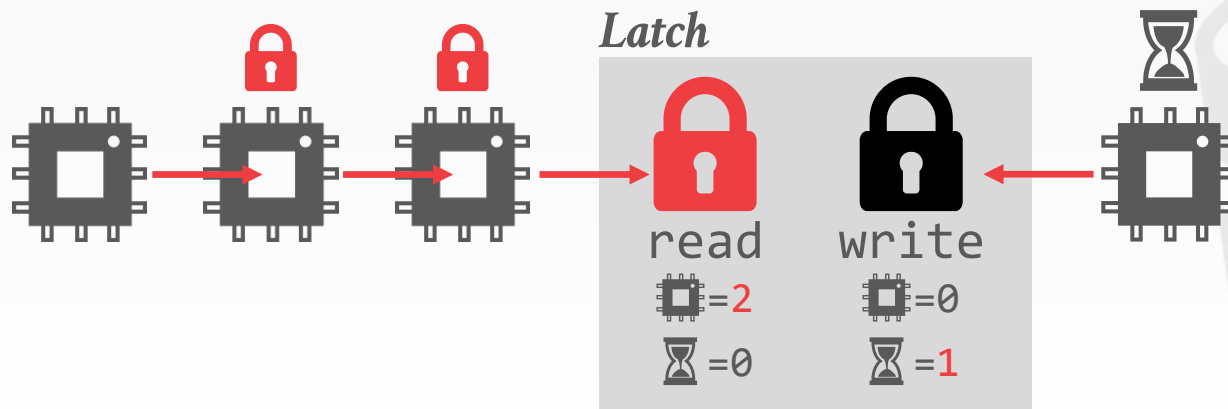
- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



LATCH IMPLEMENTATIONS

Choice #4: Reader-Writer Locks

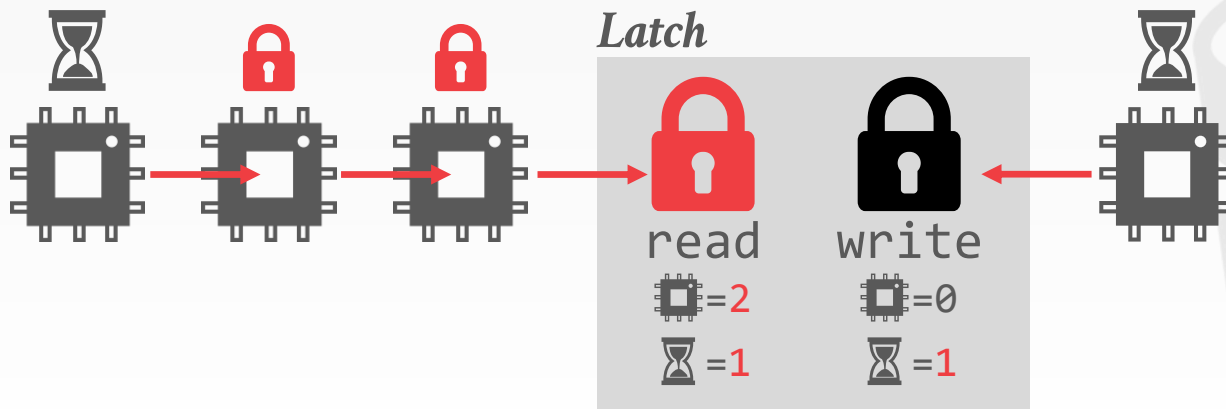
- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



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 /COUPLING

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 the parent node.

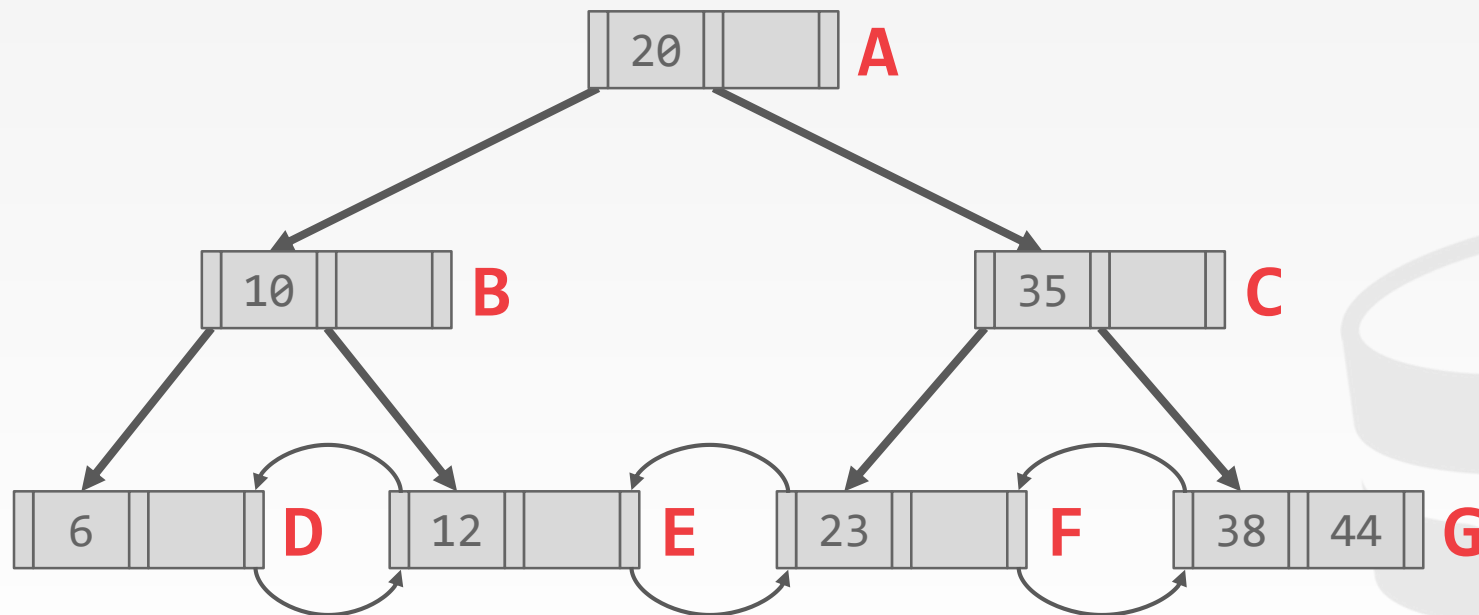
Insert/Delete: Start at root and go down, obtaining write (**W**) latches as needed.

Once child is locked, check if it is safe:

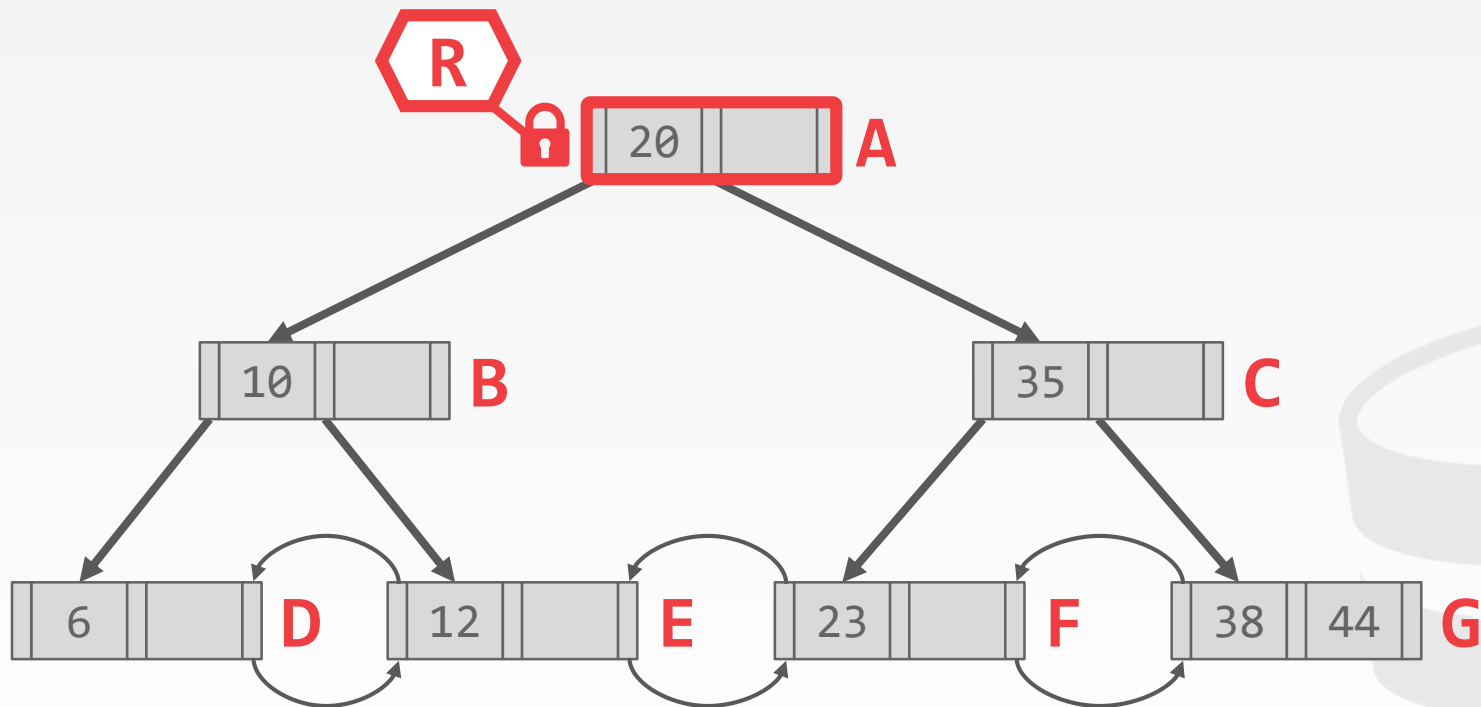
- If child is safe, release all locks on ancestors.



EXAMPLE #1: SEARCH 23

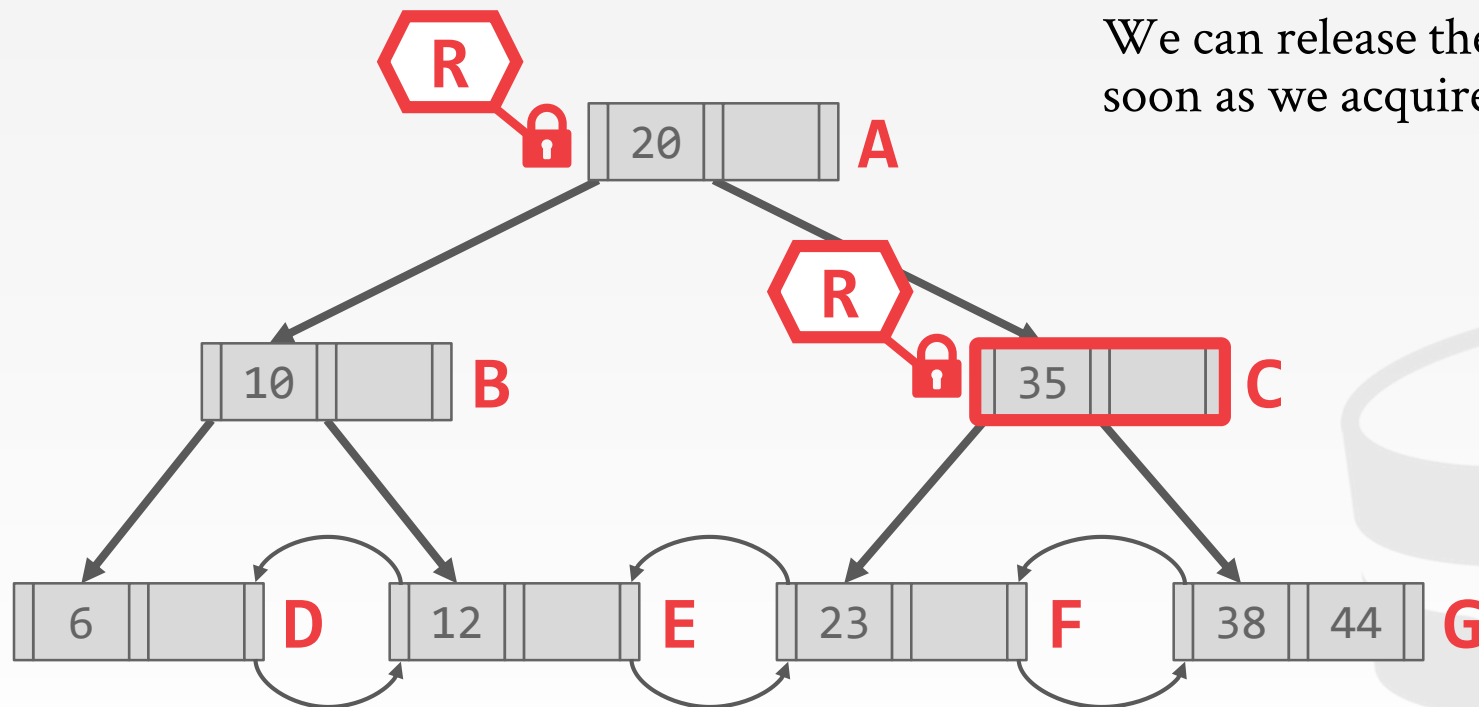


EXAMPLE #1: SEARCH 23



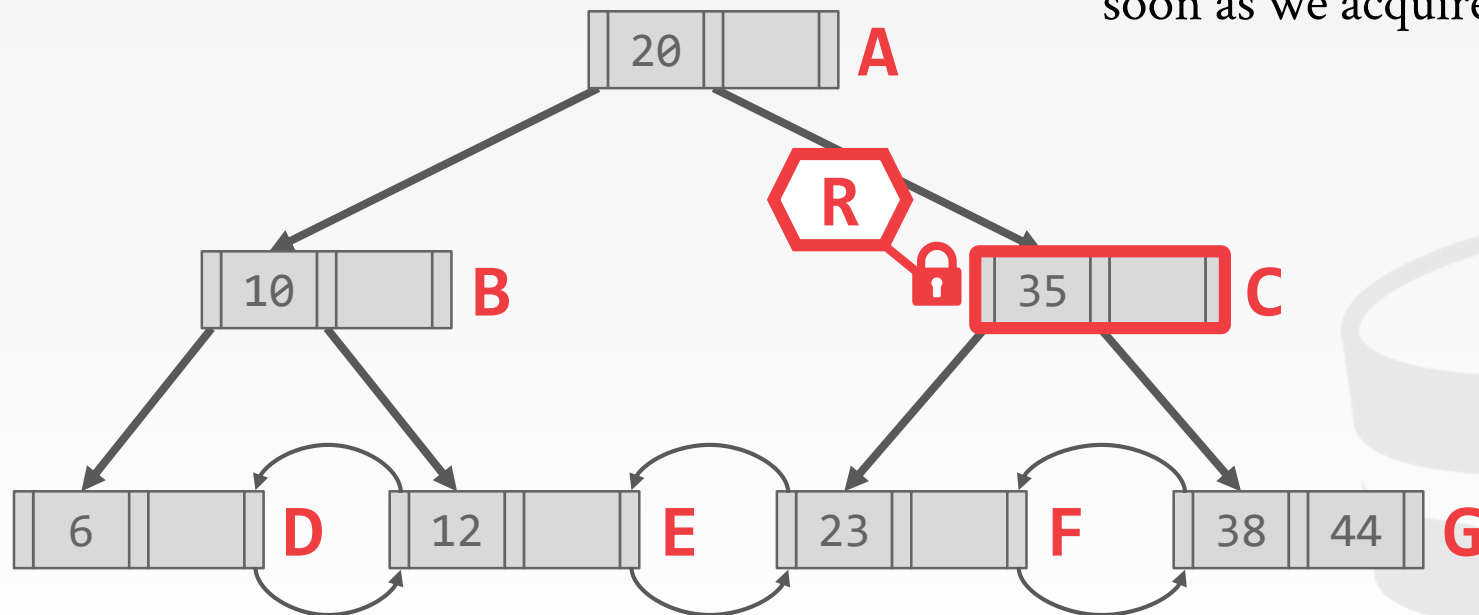
EXAMPLE #1: SEARCH 23

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



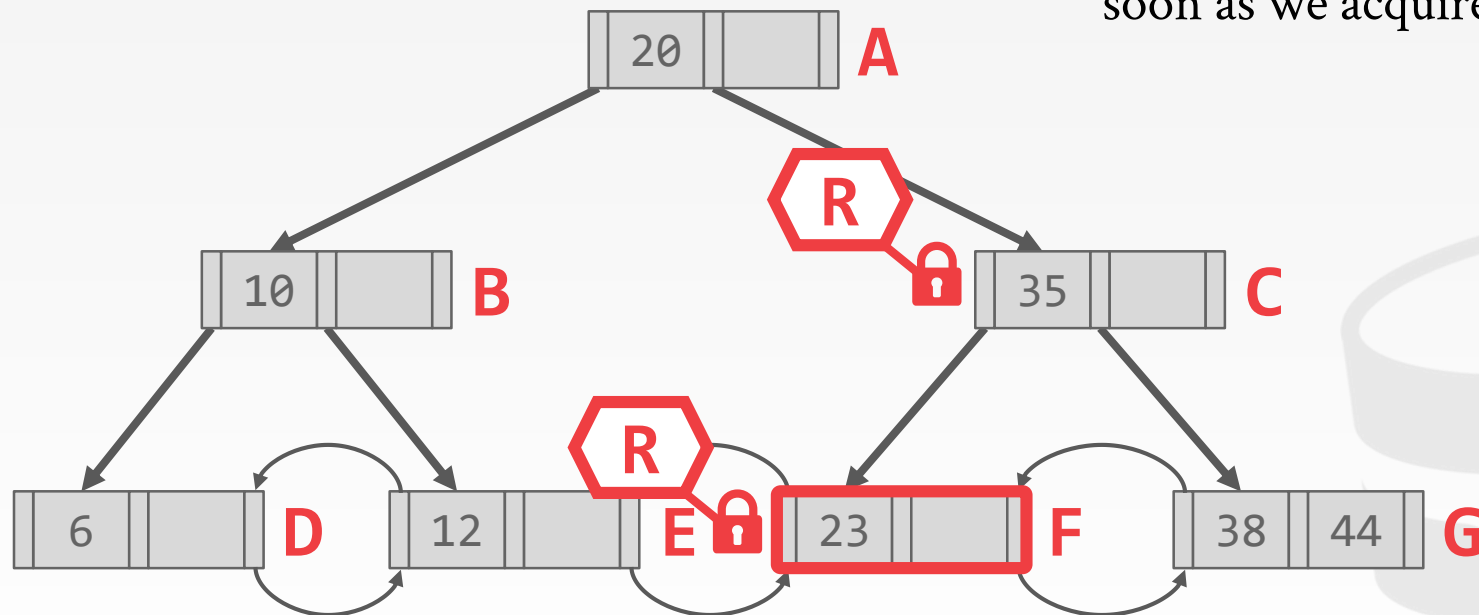
EXAMPLE #1: SEARCH 23

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



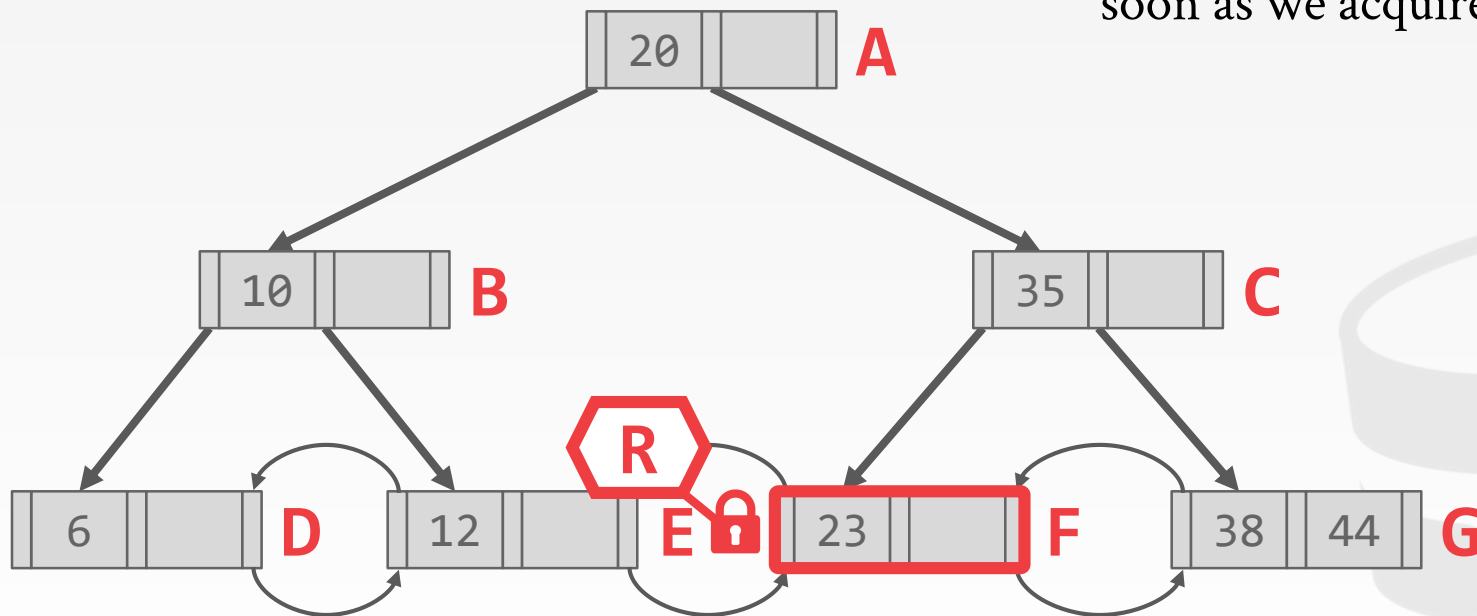
EXAMPLE #1: SEARCH 23

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

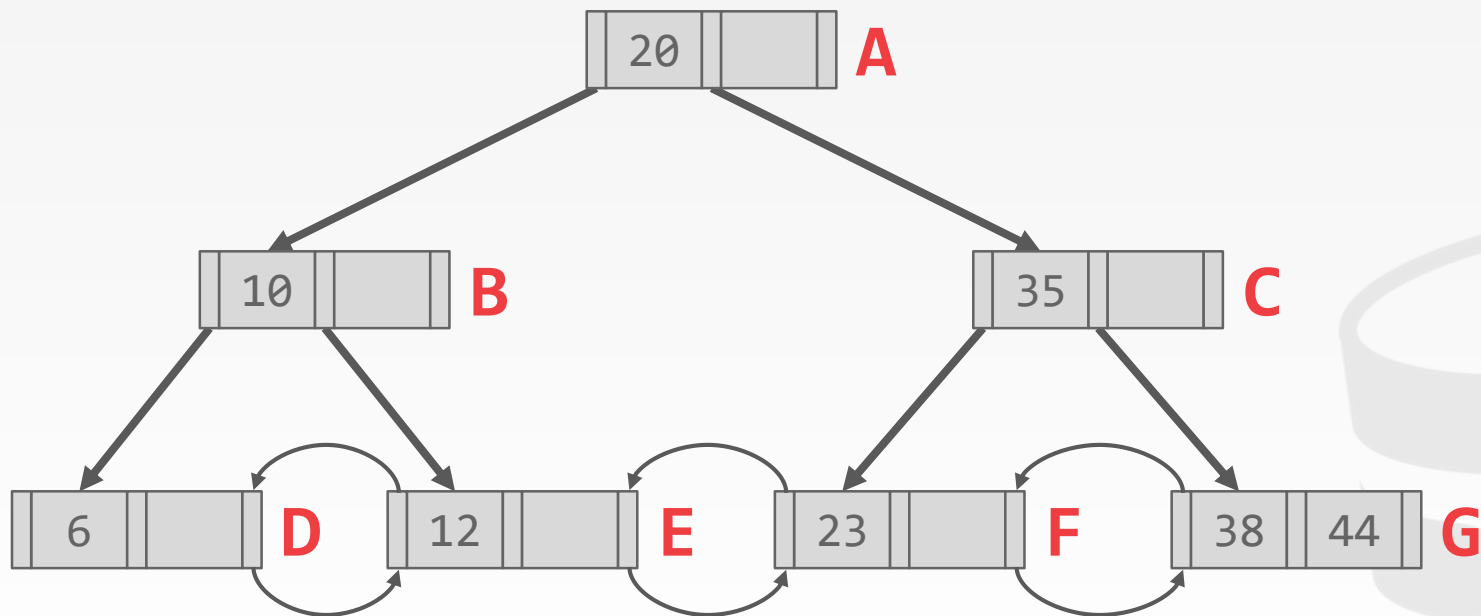


EXAMPLE #1: SEARCH 23

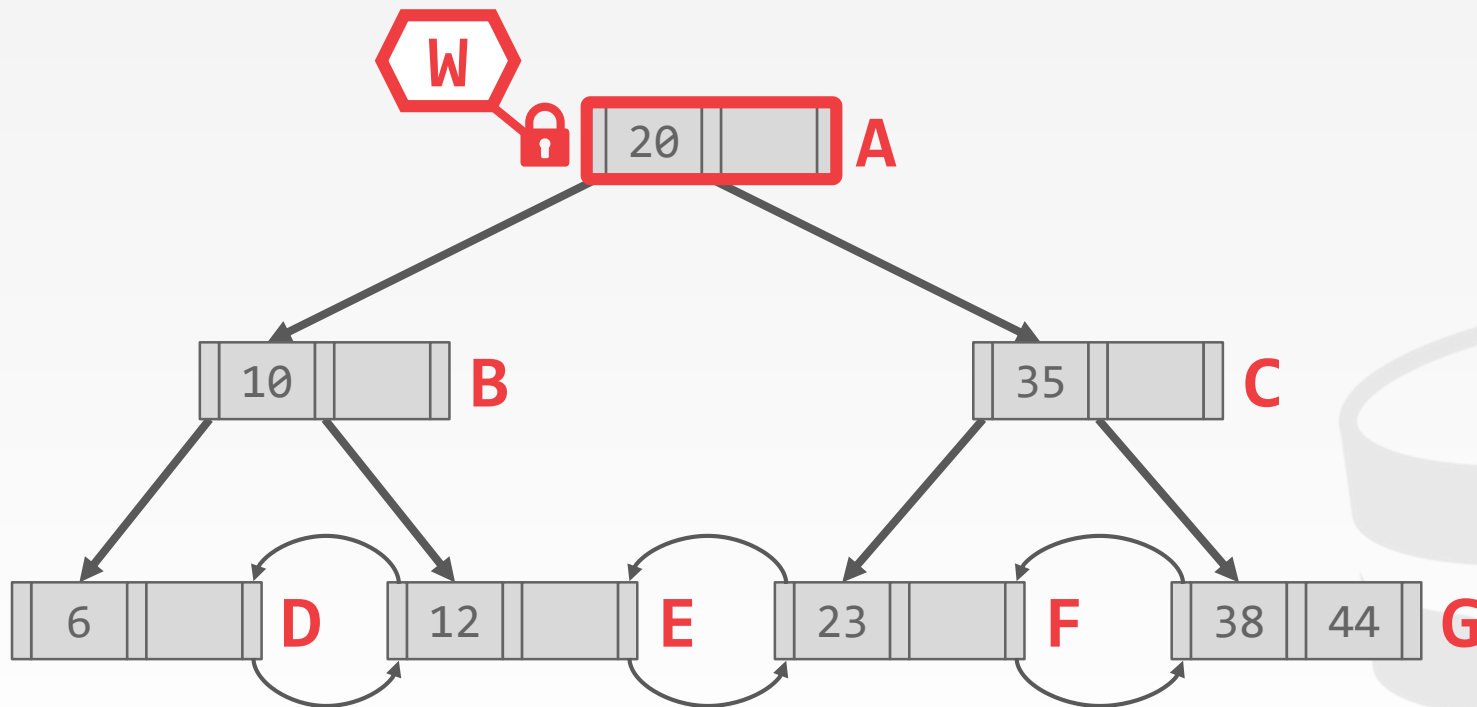
We can release the latch on **A** as soon as we acquire the latch for **C**.



EXAMPLE #2: DELETE 44

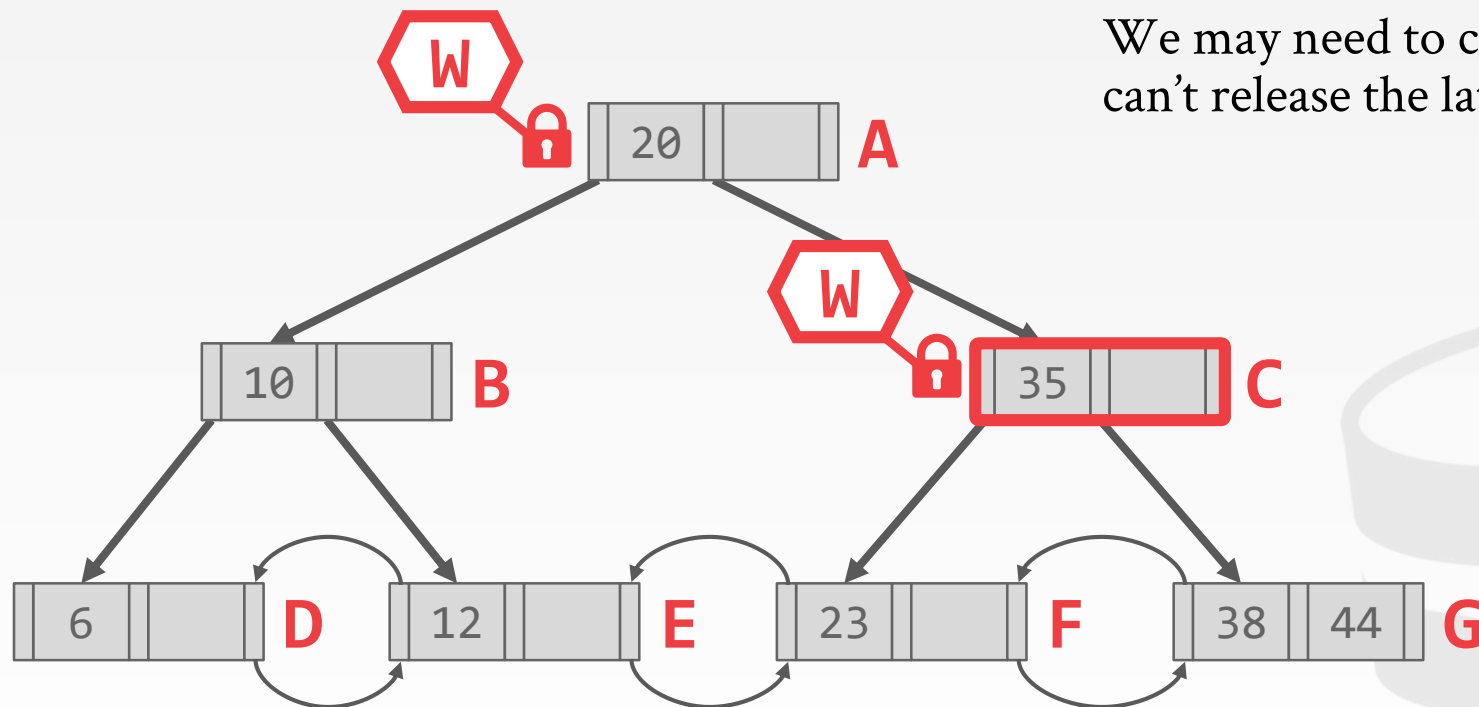


EXAMPLE #2: DELETE 44

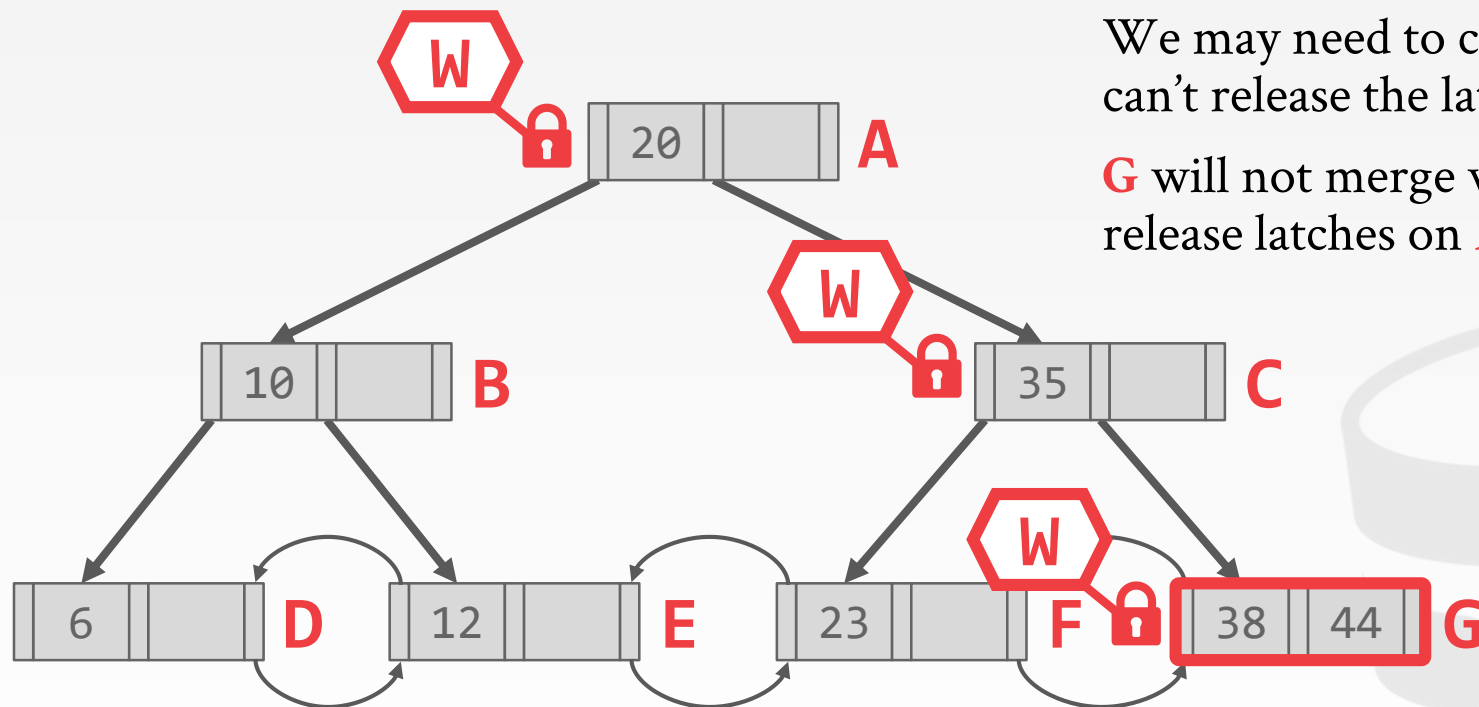


EXAMPLE #2: DELETE 44

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



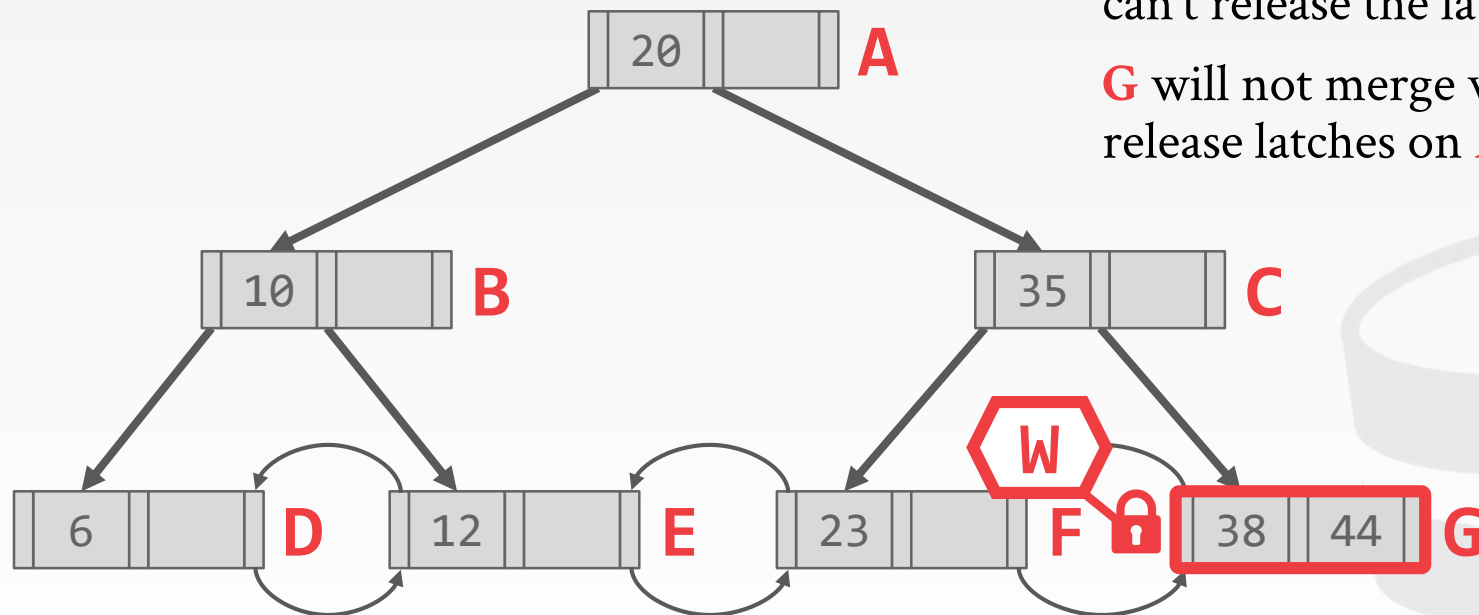
EXAMPLE #2: DELETE 44



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**.

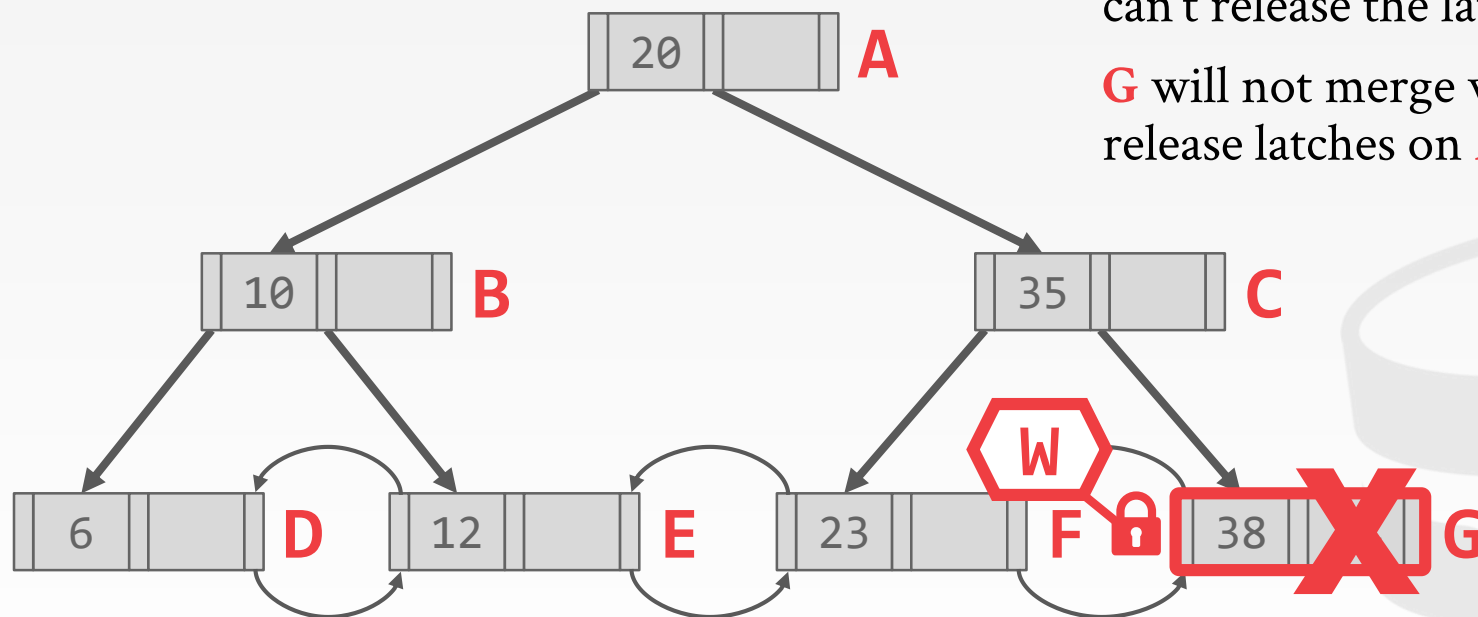
EXAMPLE #2: DELETE 44



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**.

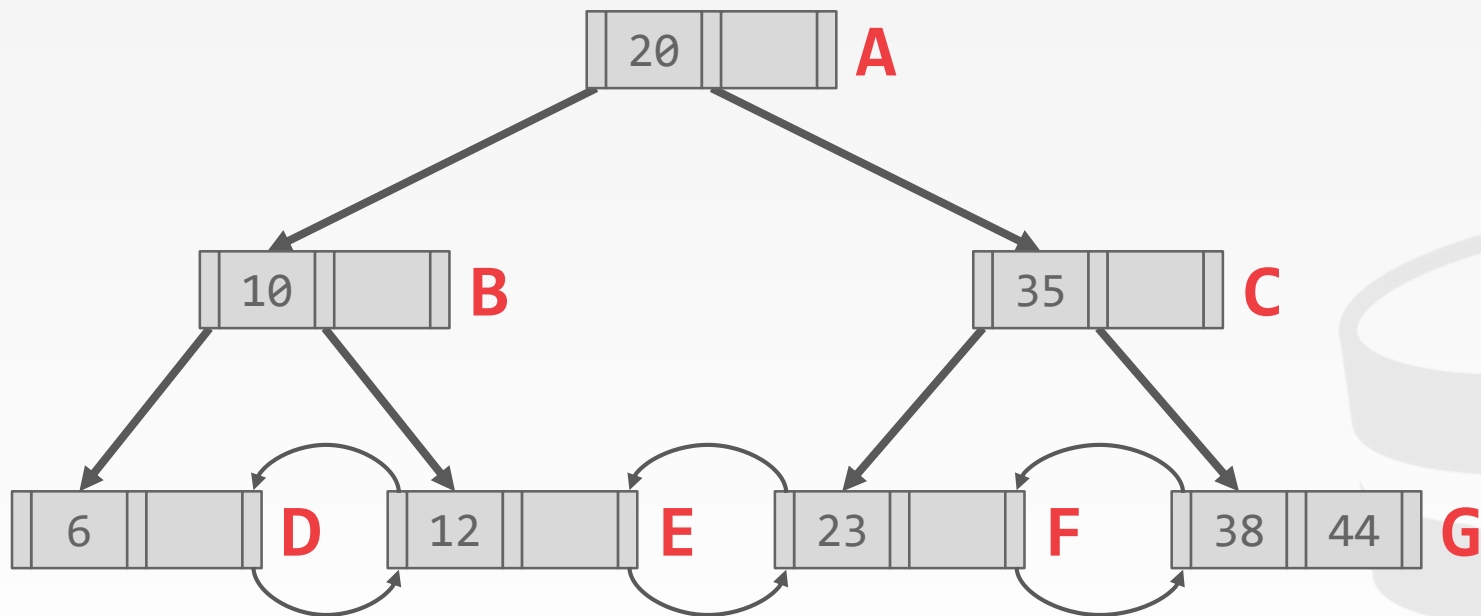
EXAMPLE #2: DELETE 44



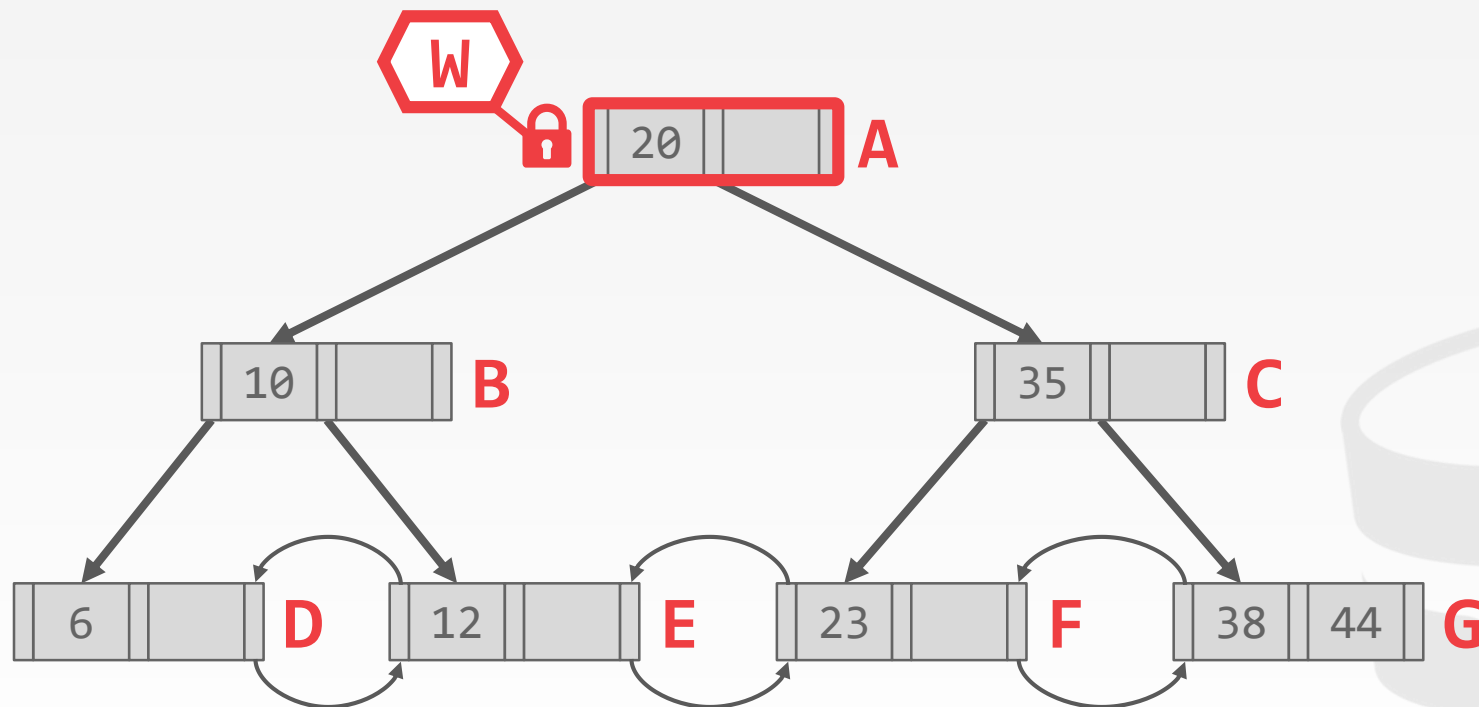
We may need to coalesce **C**, so we can't release the latch on **A**.

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

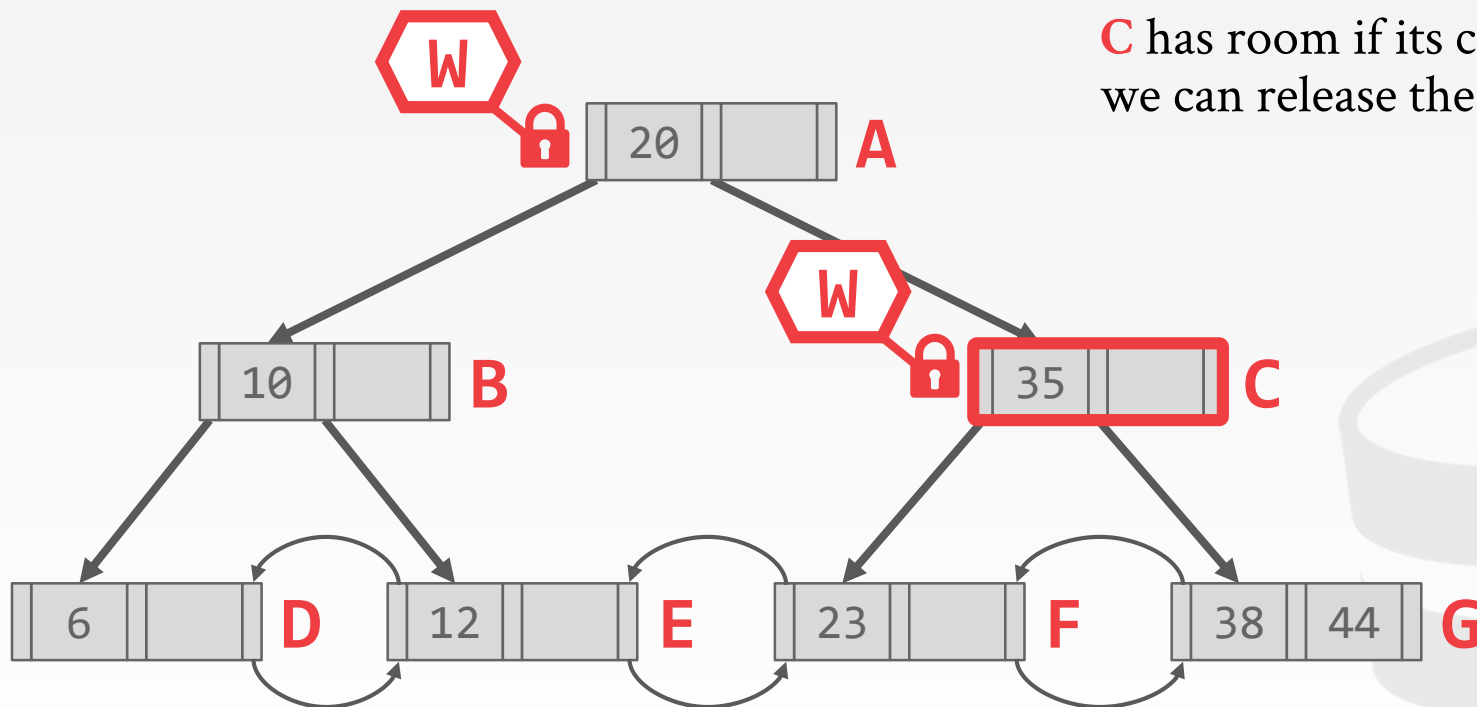
EXAMPLE #3: INSERT 40



EXAMPLE #3: INSERT 40



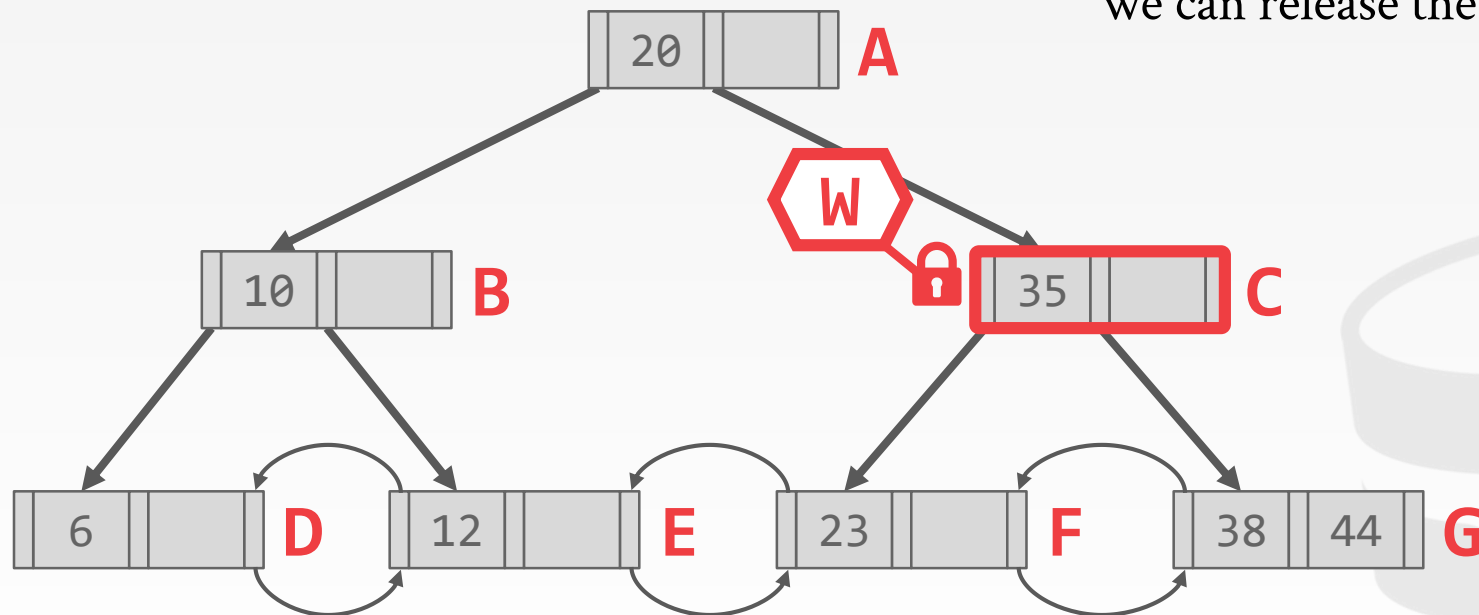
EXAMPLE #3: INSERT 40



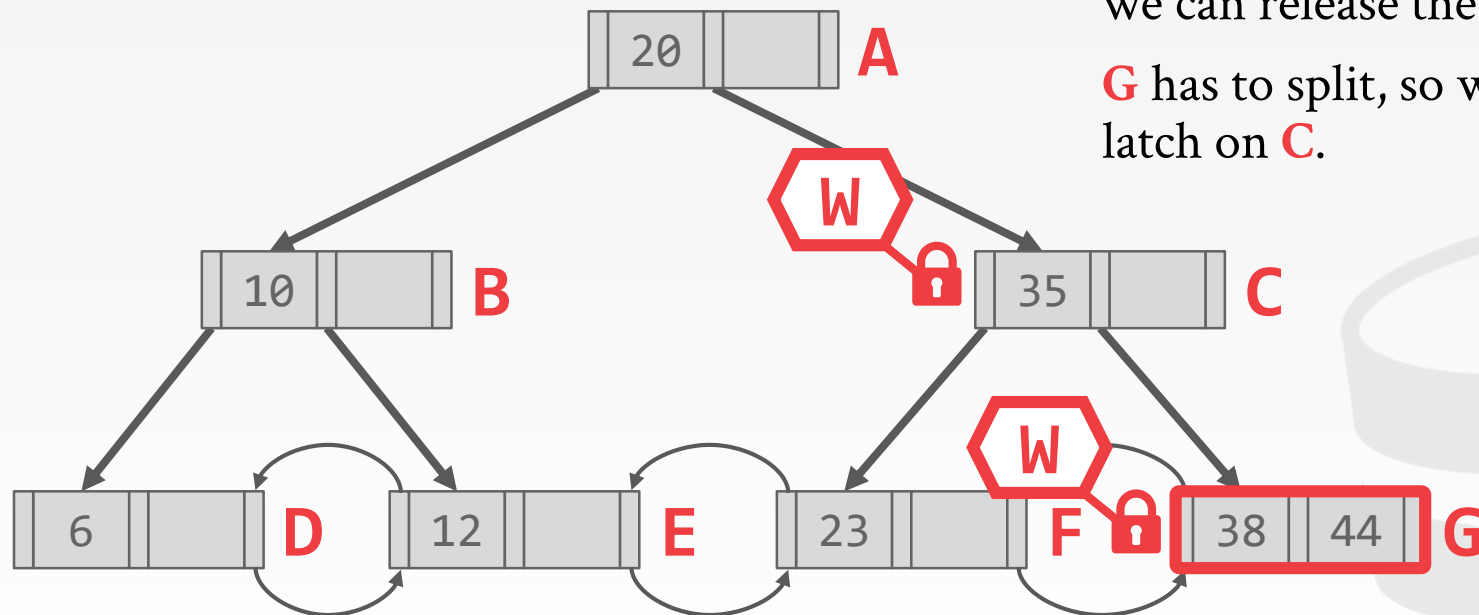
C has room if its child has to split, so we can release the latch on **A**.

EXAMPLE #3: INSERT 40

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



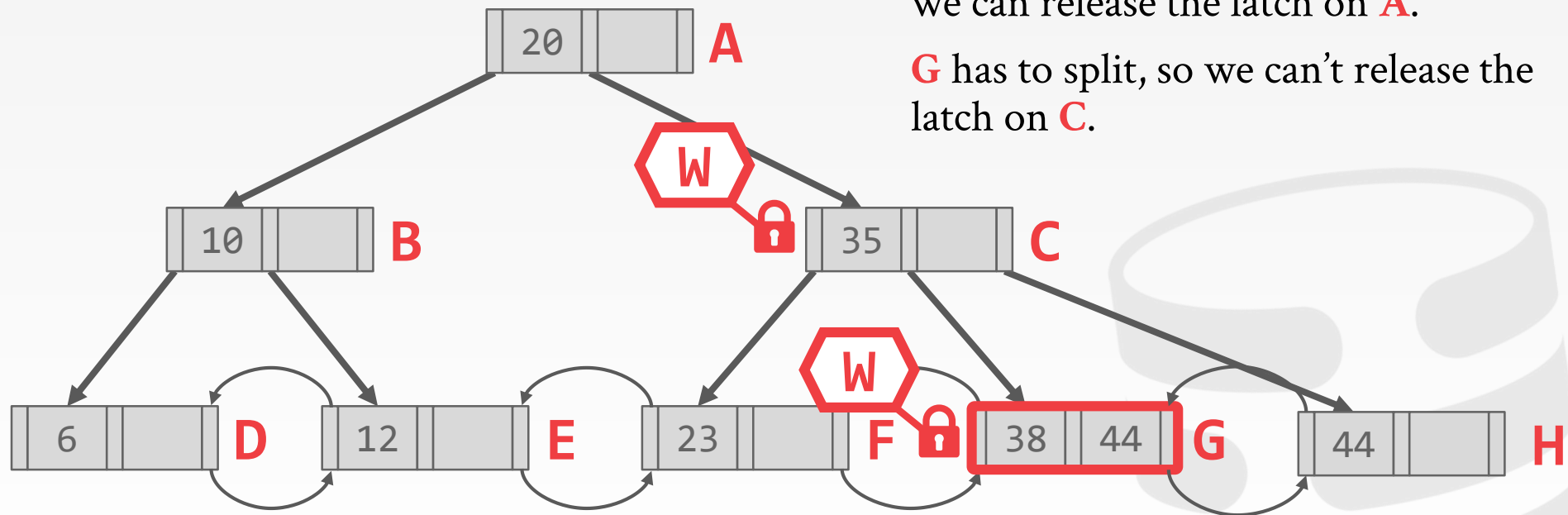
EXAMPLE #3: INSERT 40



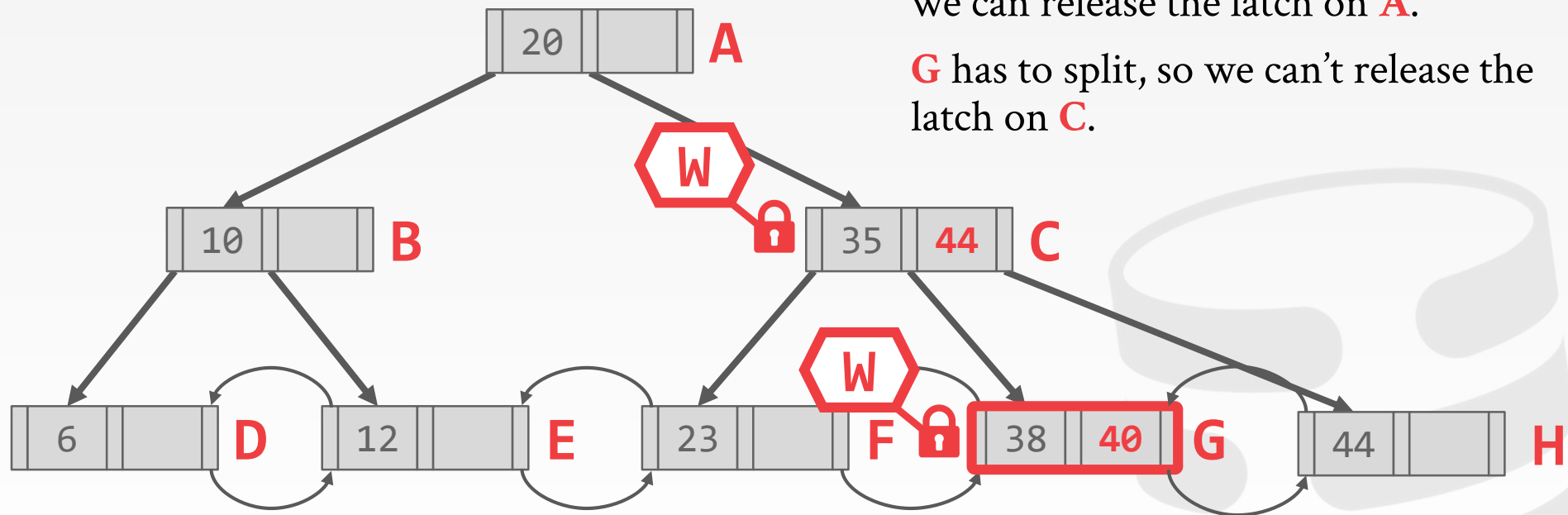
C has room if its child has to split, so we can release the latch on **A**.

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

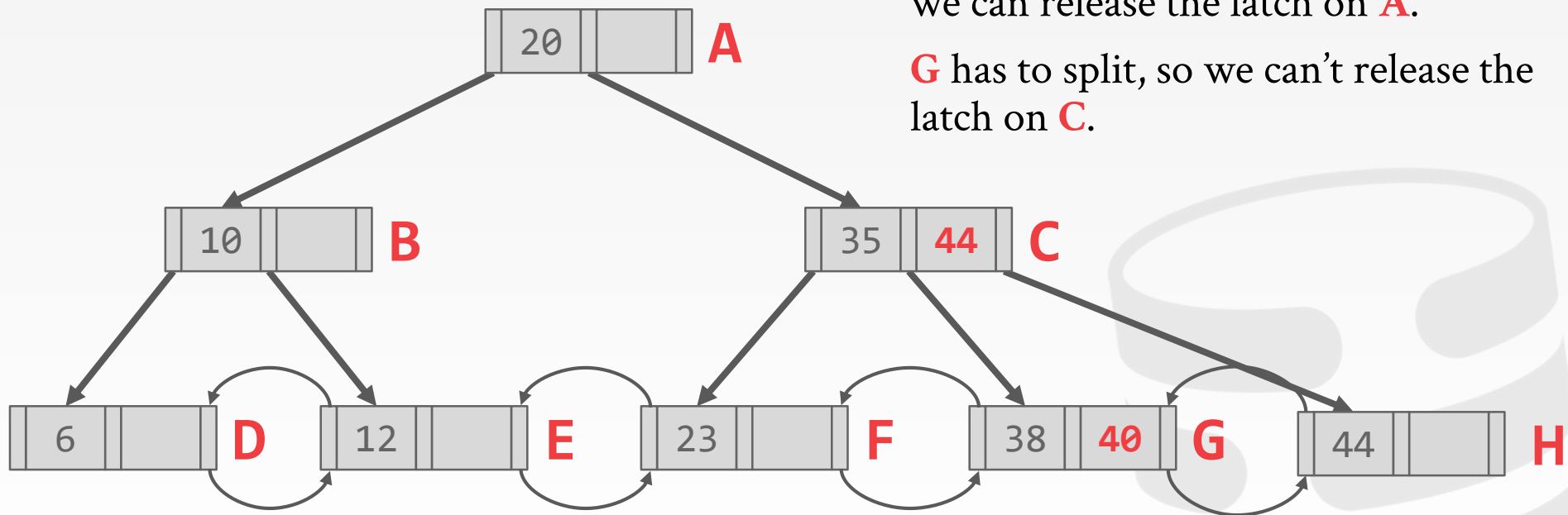
EXAMPLE #3: INSERT 40



EXAMPLE #3: INSERT 40



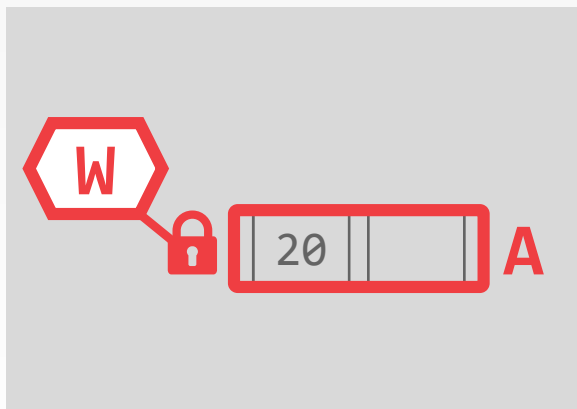
EXAMPLE #3: INSERT 40



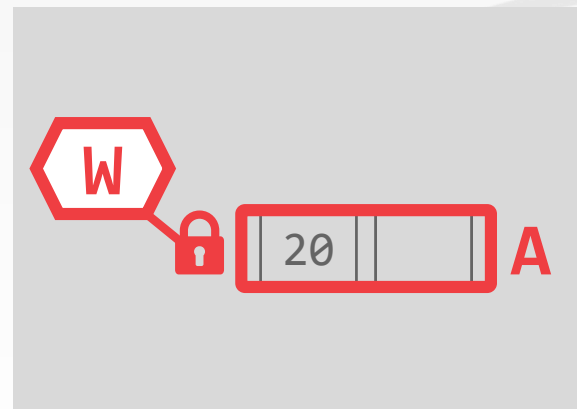
OBSERVATION

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

Delete 44



Insert 40



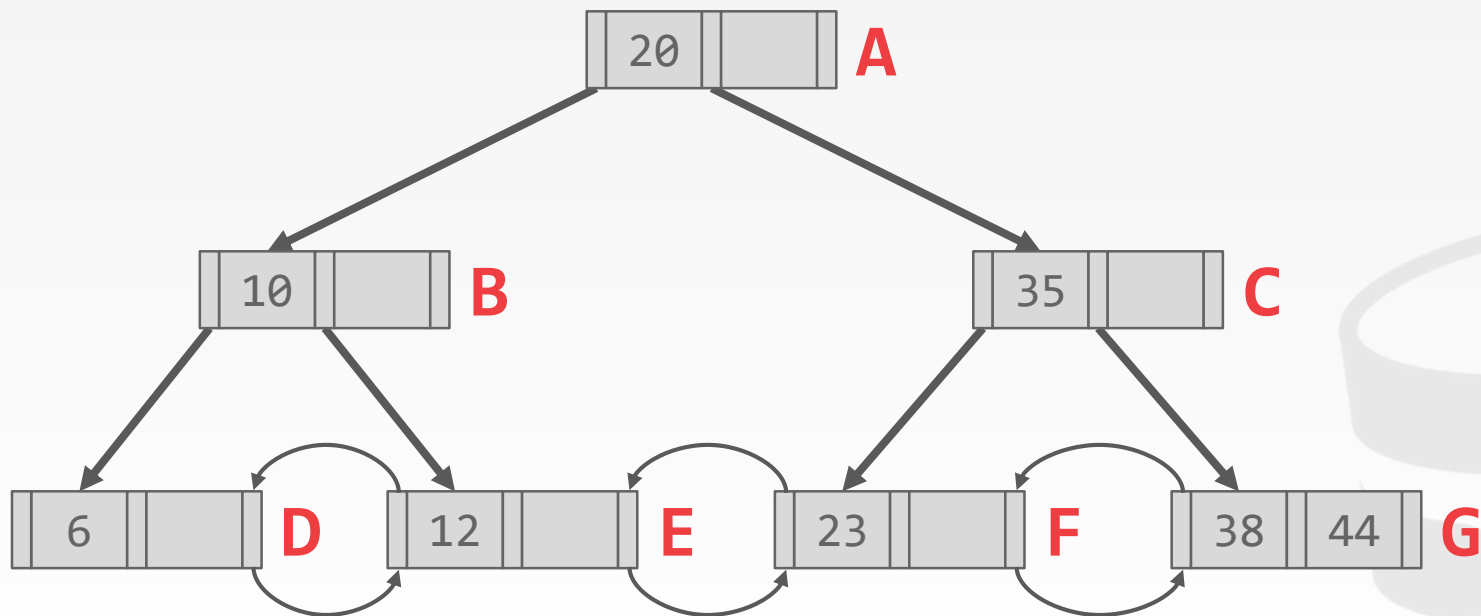
BETTER LATCH CRABBING

Optimistically assume that the leaf is safe.

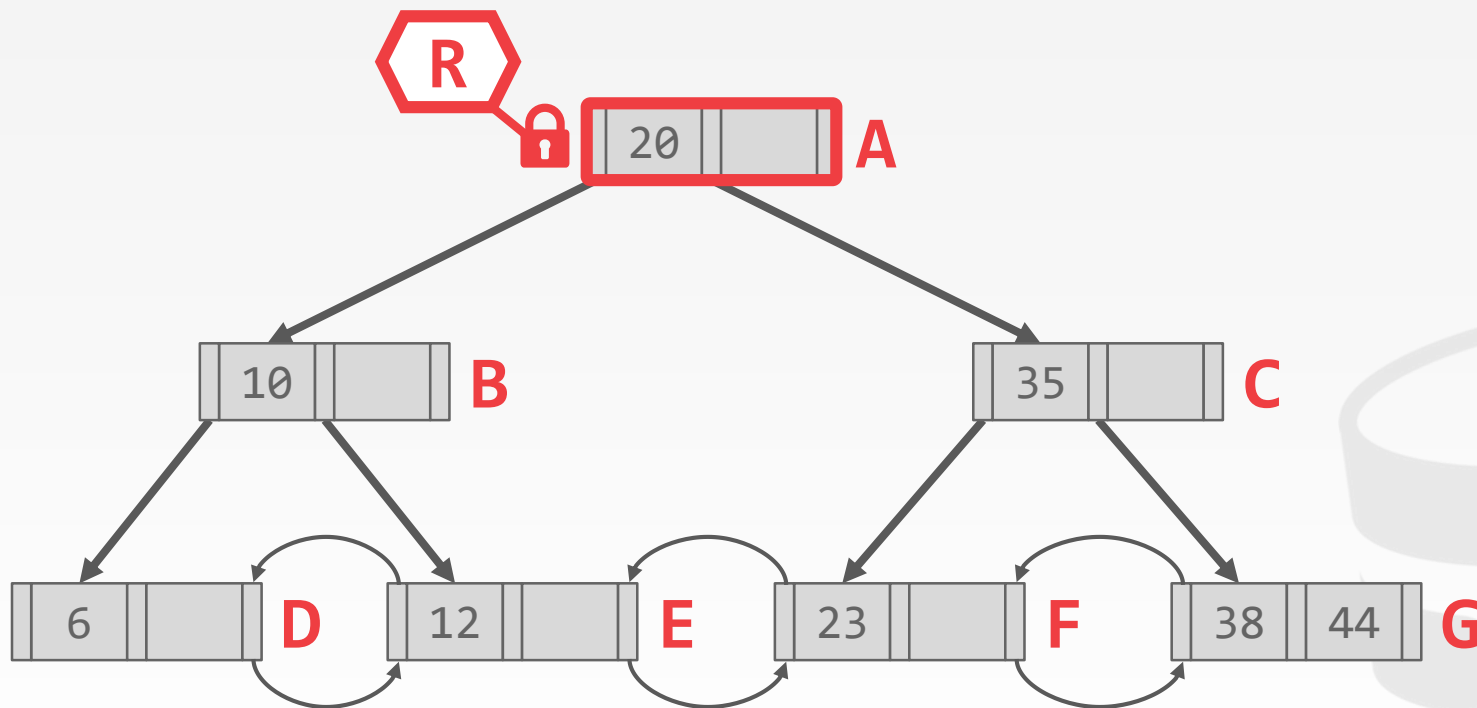
- Take **R** latches as you traverse the tree to reach it and verify.
- If leaf is not safe, then do previous algorithm.



EXAMPLE #4: DELETE 44

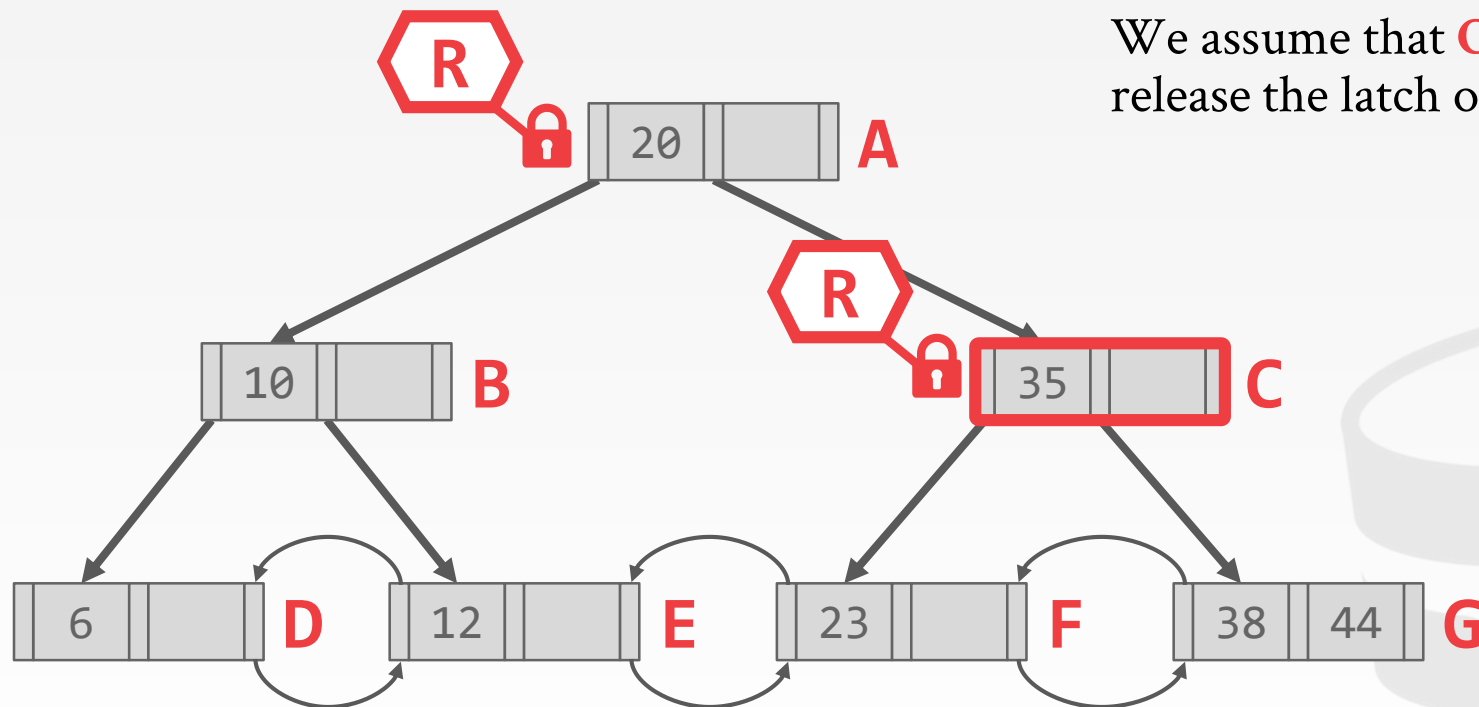


EXAMPLE #4: DELETE 44



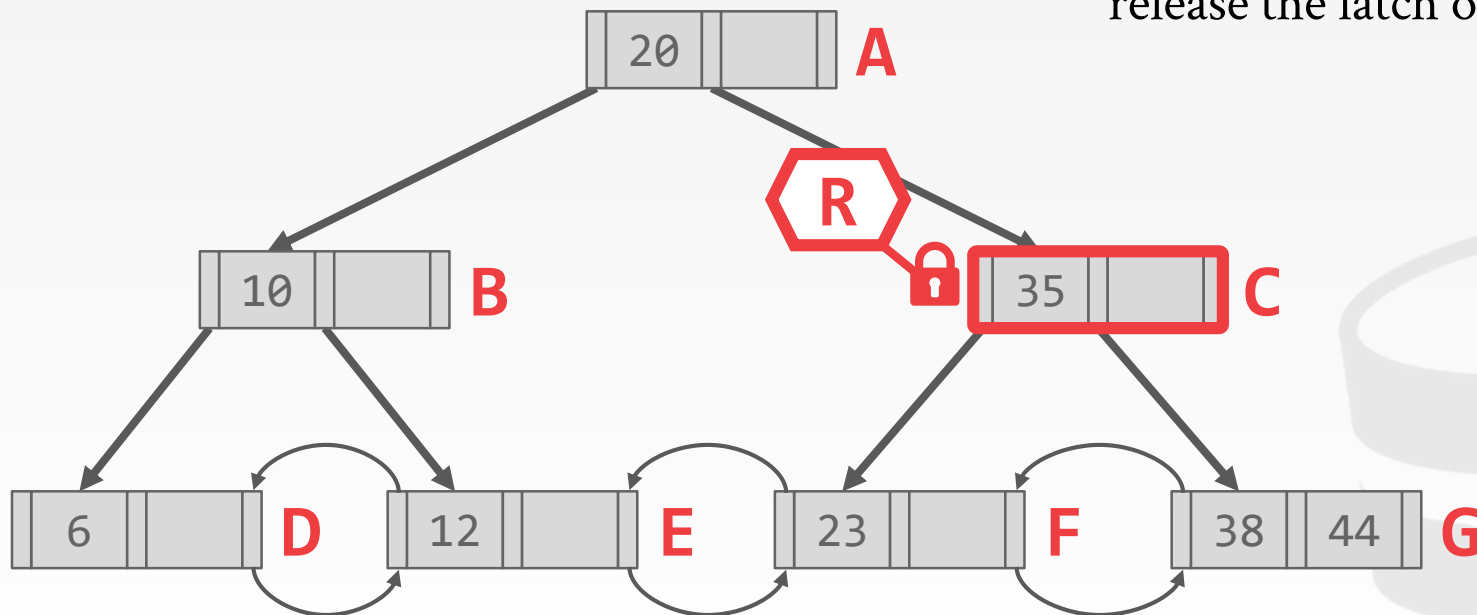
EXAMPLE #4: DELETE 44

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



EXAMPLE #4: DELETE 44

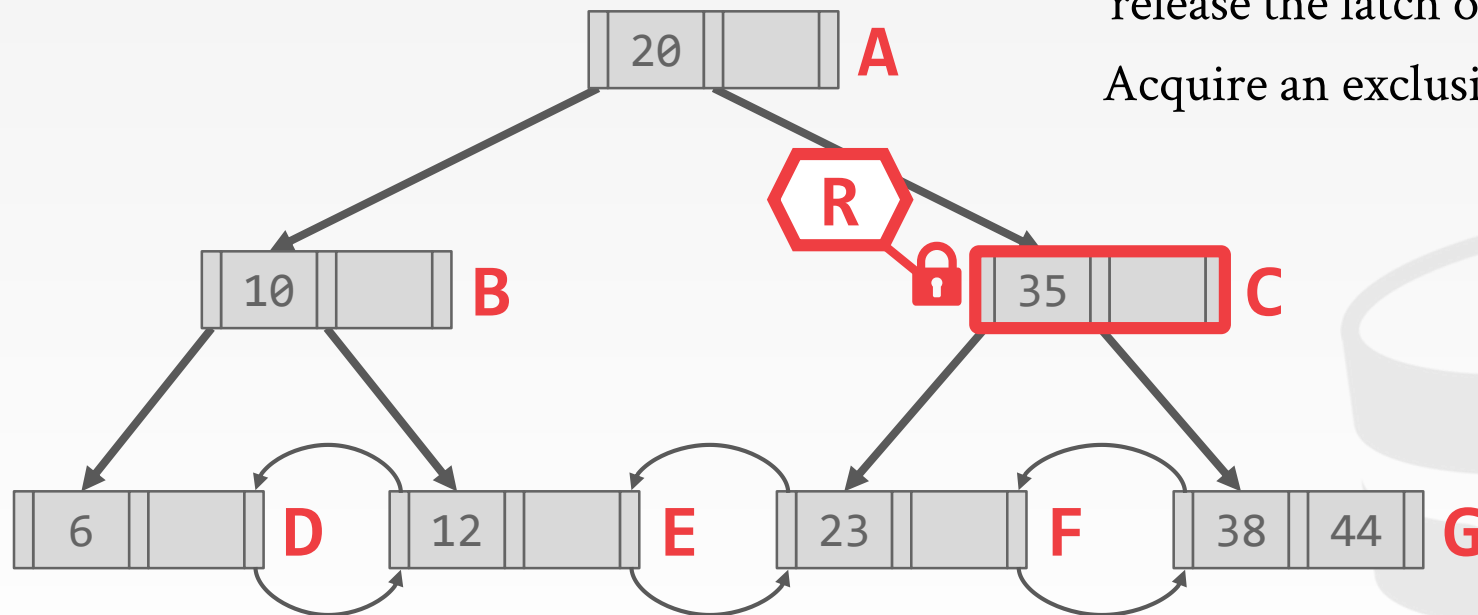
We assume that **C** is safe, so we can release the latch on **A**.



EXAMPLE #4: DELETE 44

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

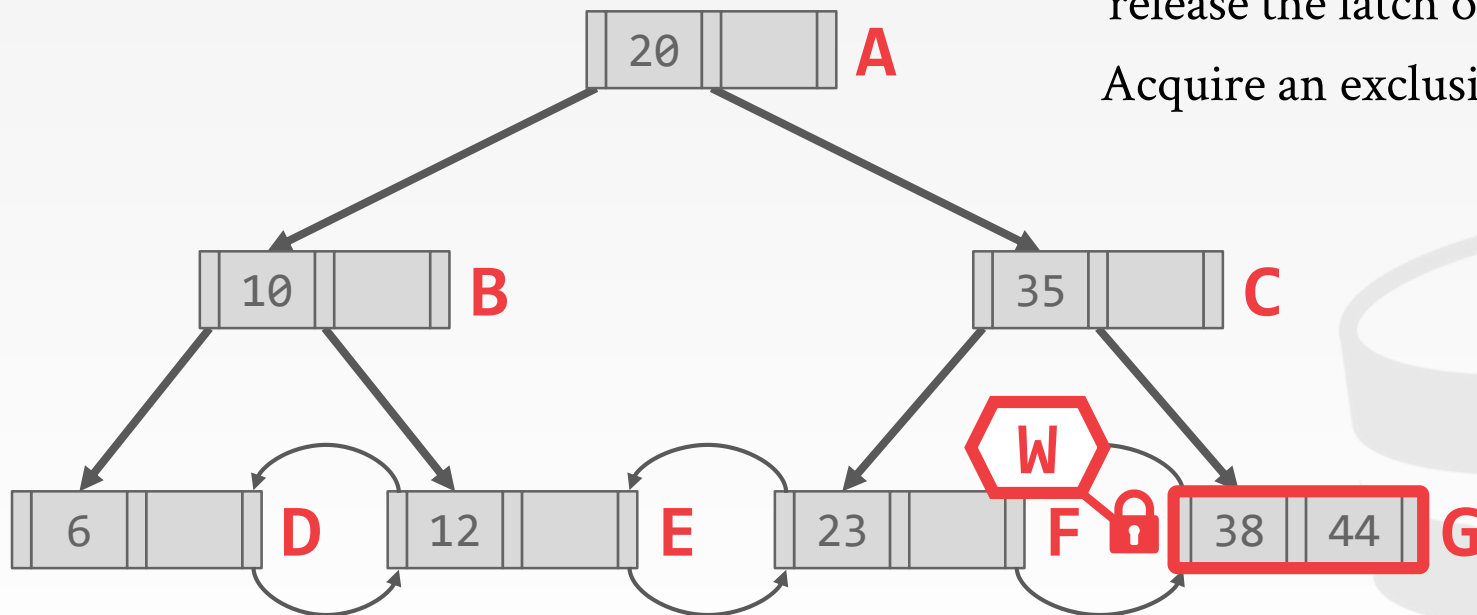
Acquire an exclusive latch on **G**.



EXAMPLE #4: DELETE 44

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

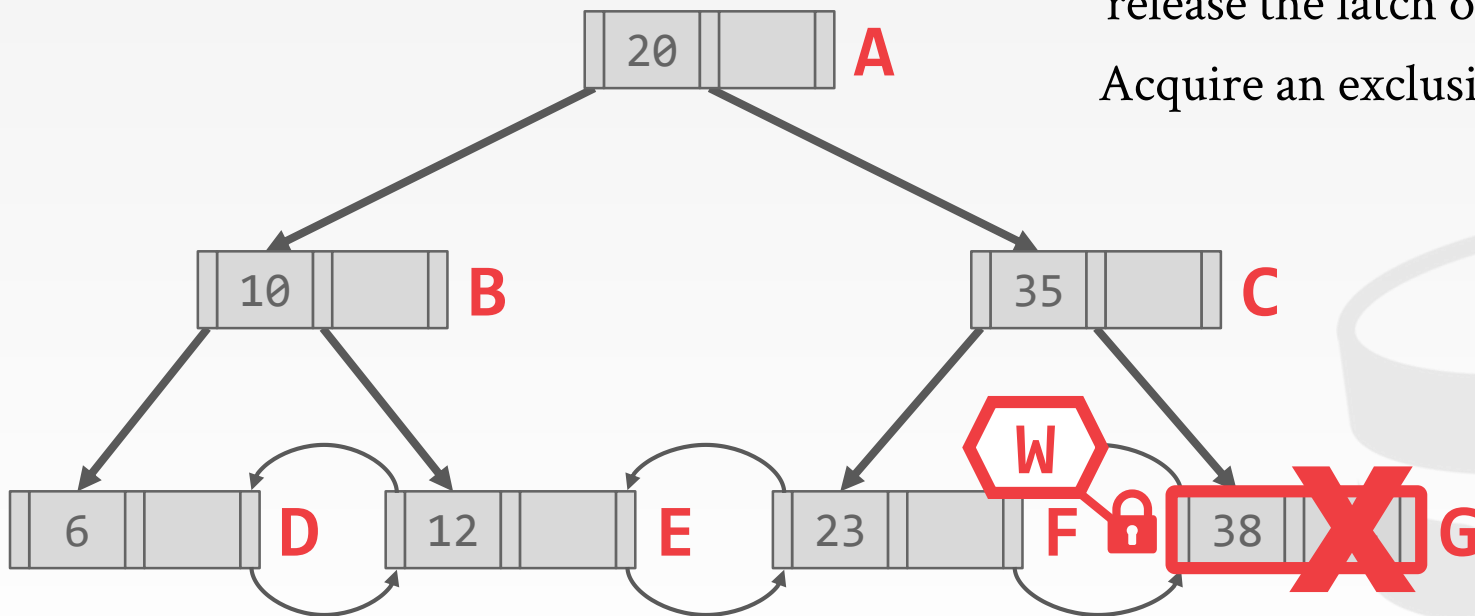
Acquire an exclusive latch on **G**.



EXAMPLE #4: DELETE 44

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

Acquire an exclusive latch on **G**.

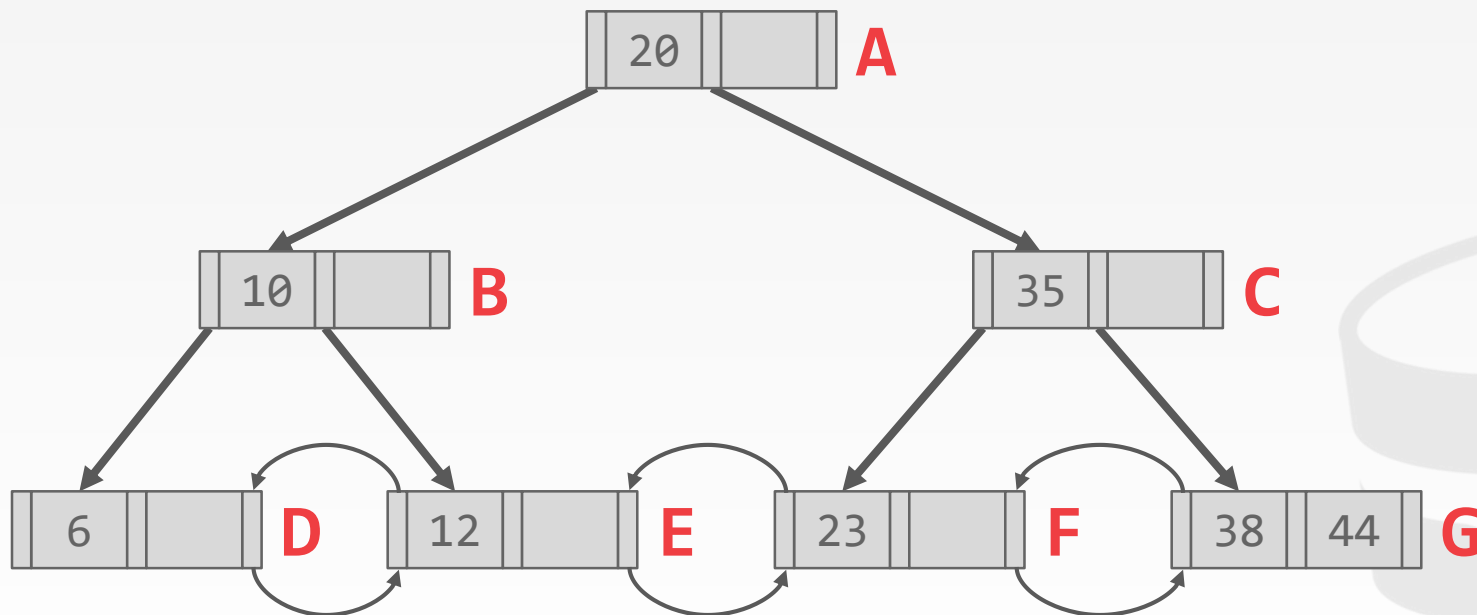


OBSERVATION

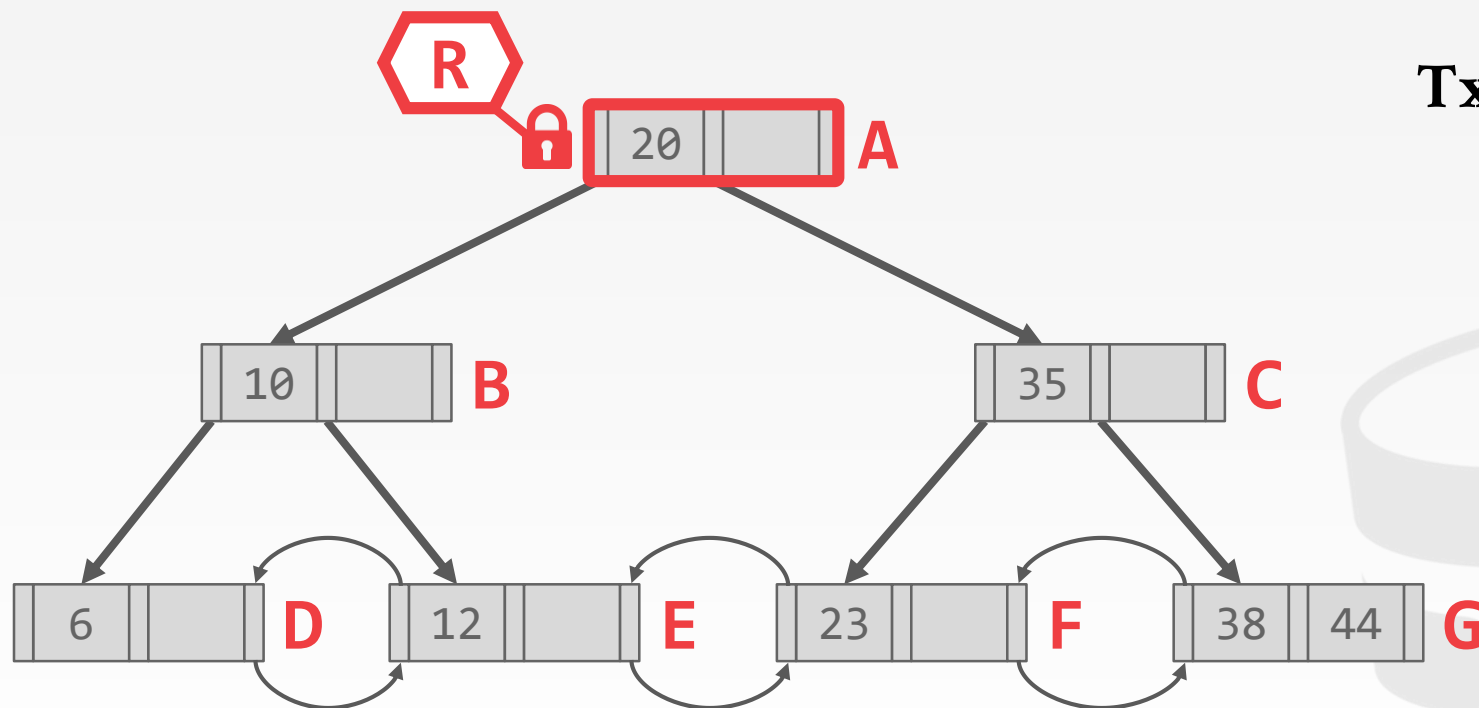
Crabbing ensures that txns do not corrupt the internal data structure during modifications.

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:

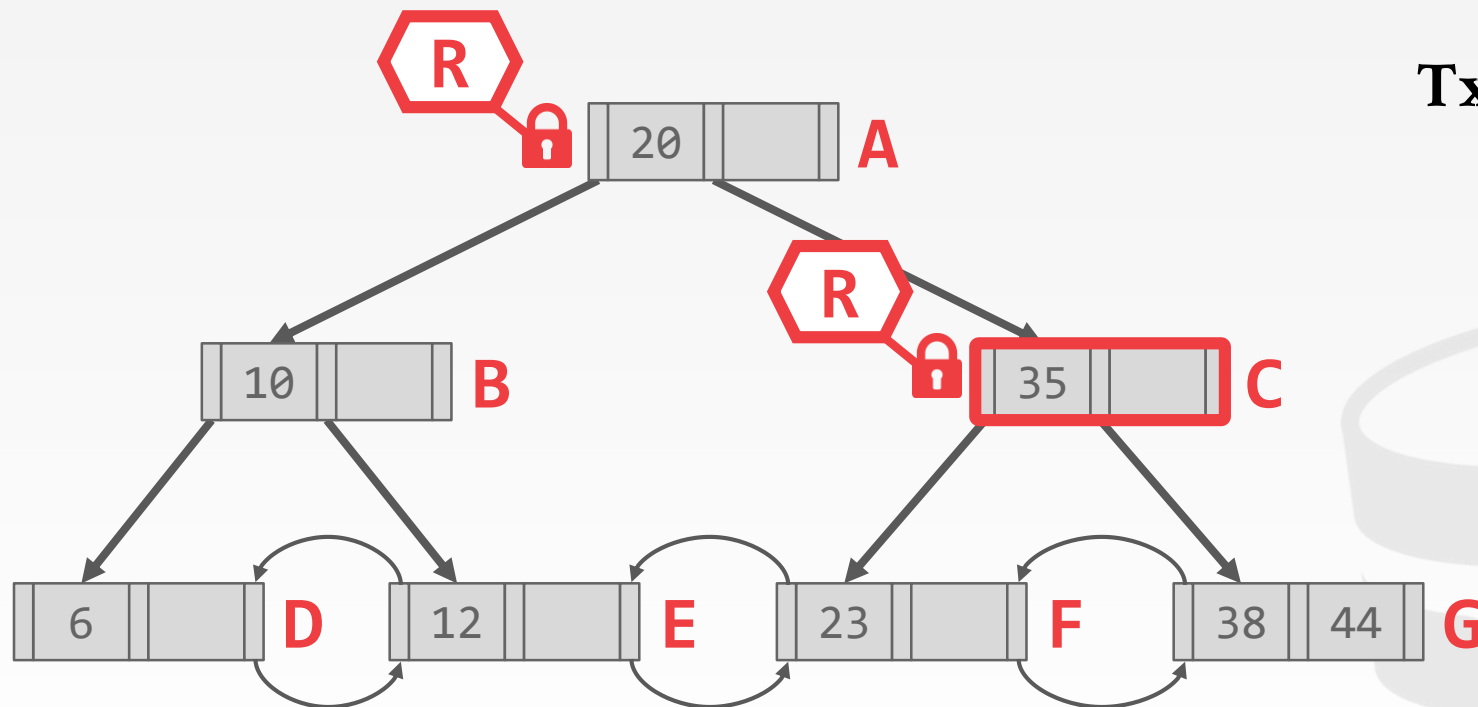


PROBLEM SCENARIO #1

Txn #1:



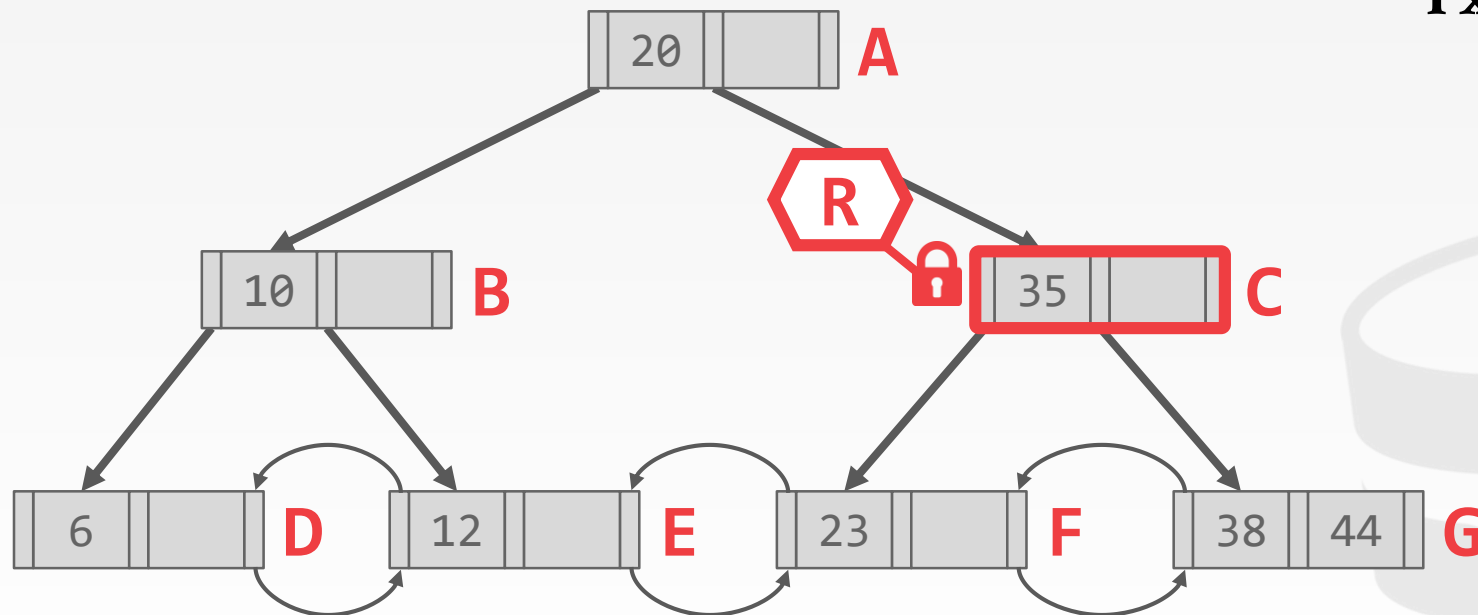
READ(25)



PROBLEM SCENARIO #1

Txn #1:


READ(25)

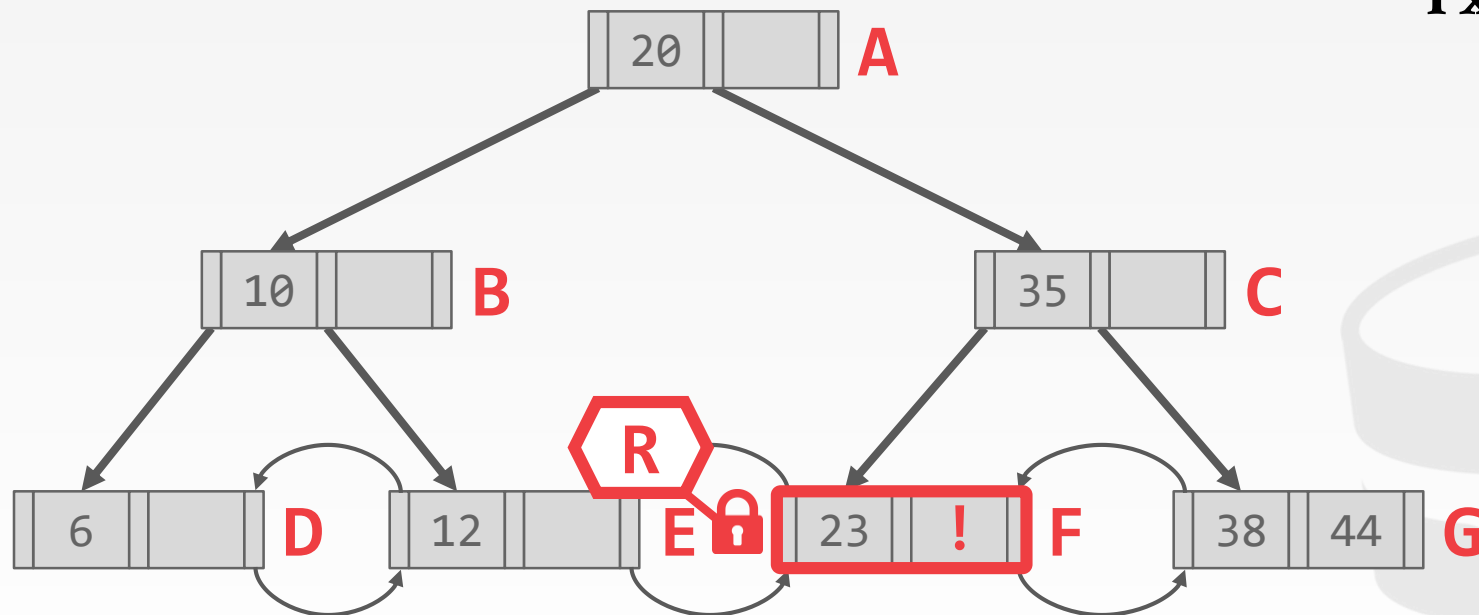


PROBLEM SCENARIO #1

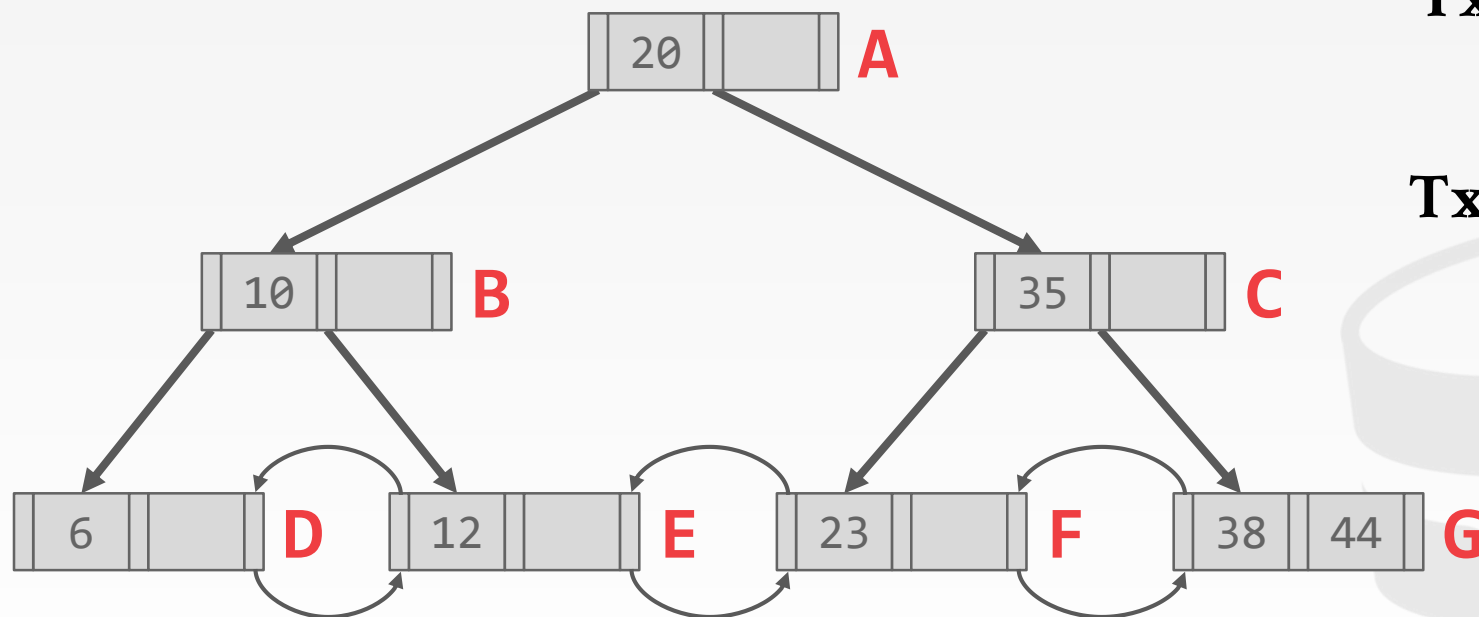
Txn #1:



READ(25)



PROBLEM SCENARIO #1



Txn #1:



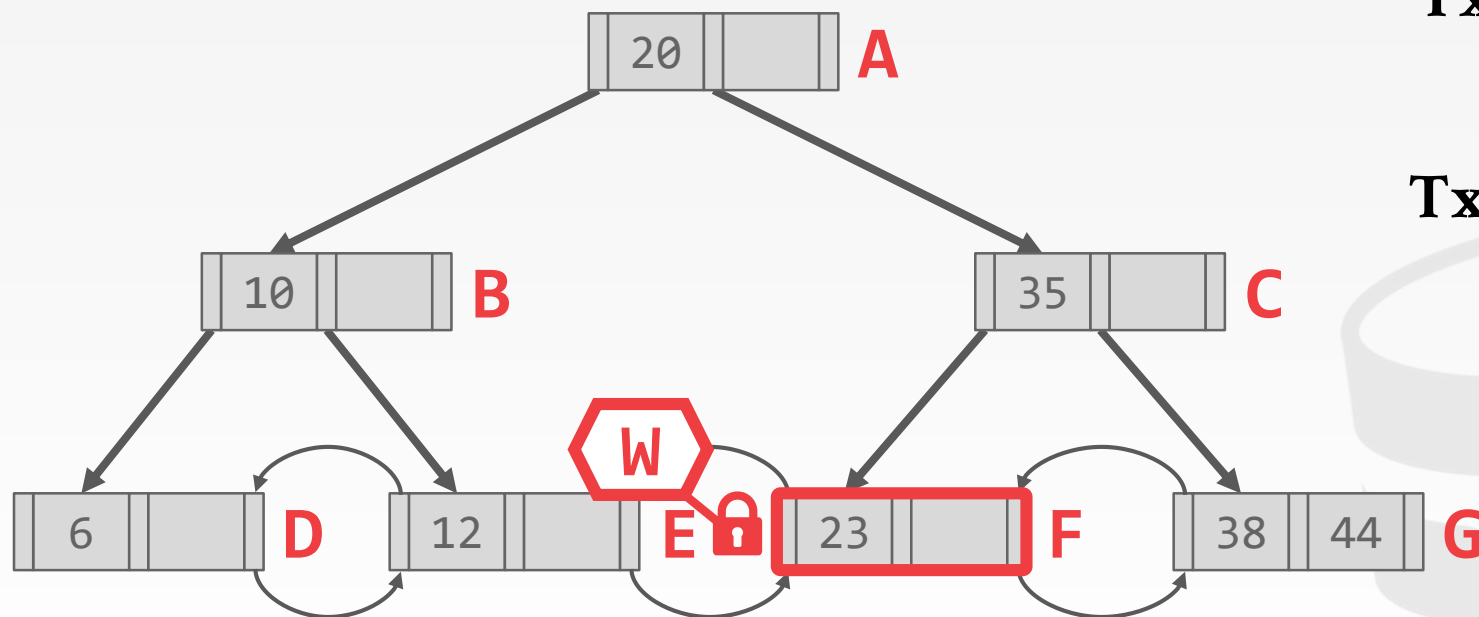
READ(25)

Txn #2:



INSERT(25)

PROBLEM SCENARIO #1



Txn #1:



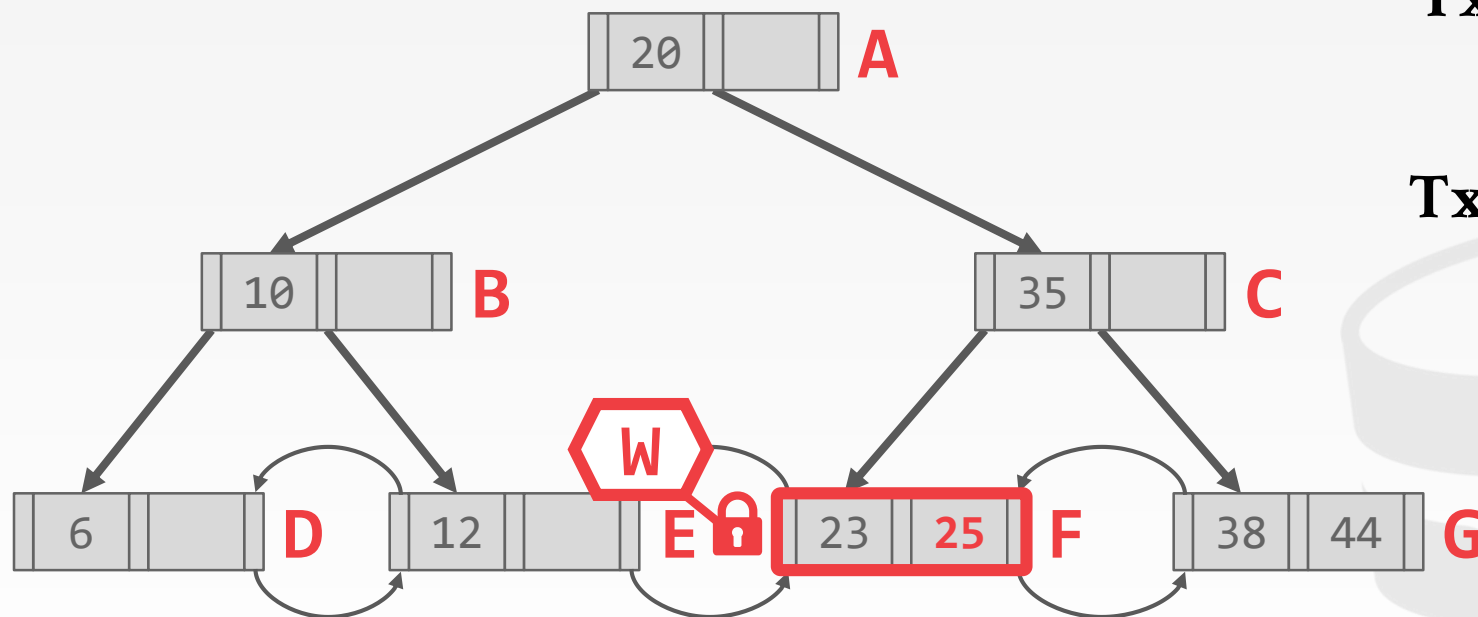
READ(25)

Txn #2:



INSERT(25)

PROBLEM SCENARIO #1



Txn #1:



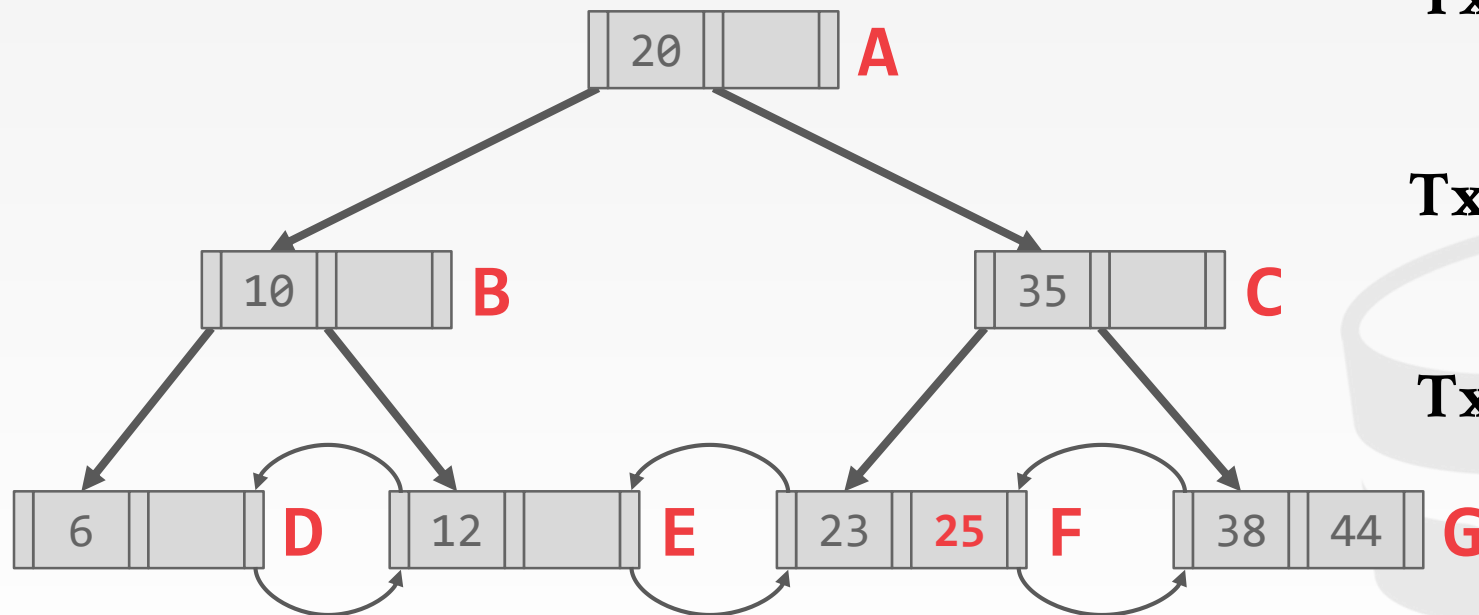
READ(25)

Txn #2:



INSERT(25)

PROBLEM SCENARIO #1



Txn #1:



READ(25)

Txn #2:



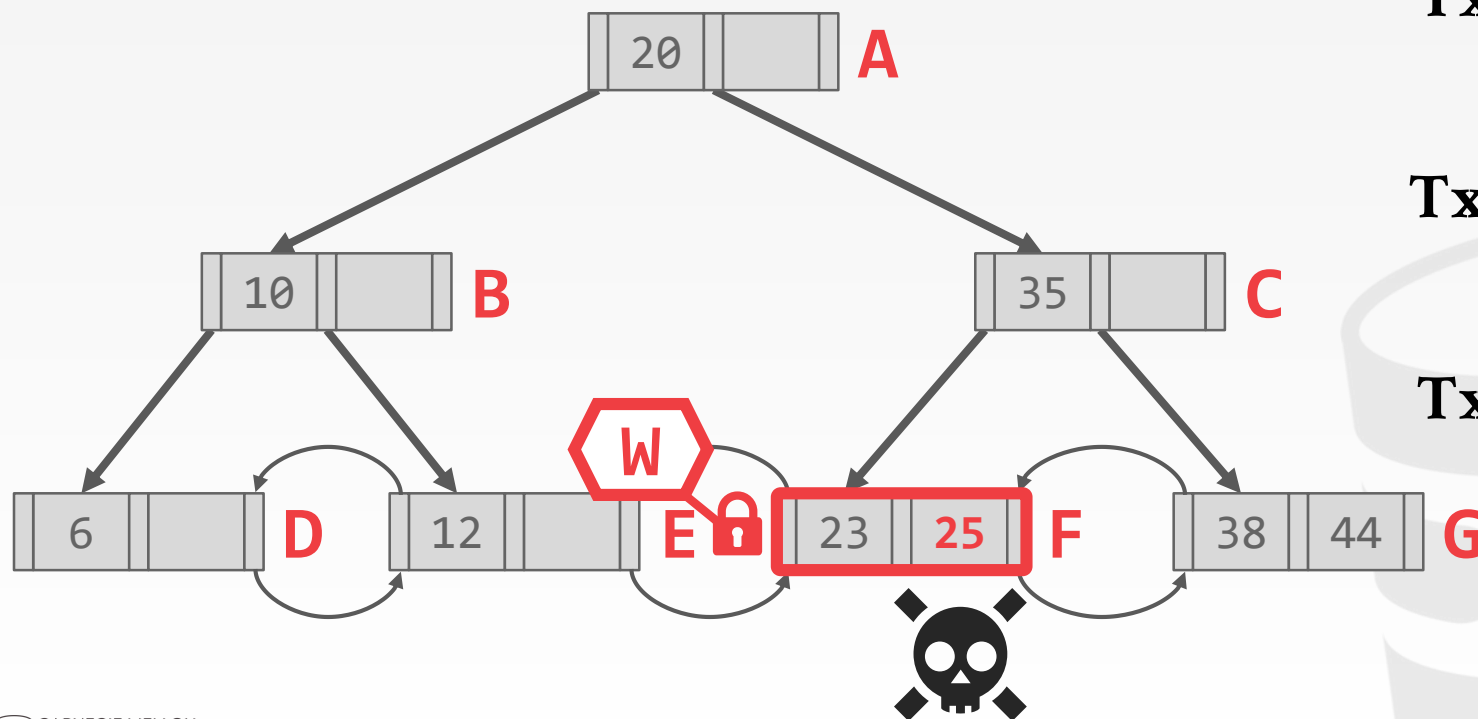
INSERT(25)

Txn #1:



INSERT(25)

PROBLEM SCENARIO #1



Txn #1:



READ(25)

Txn #2:



INSERT(25)

Txn #1:



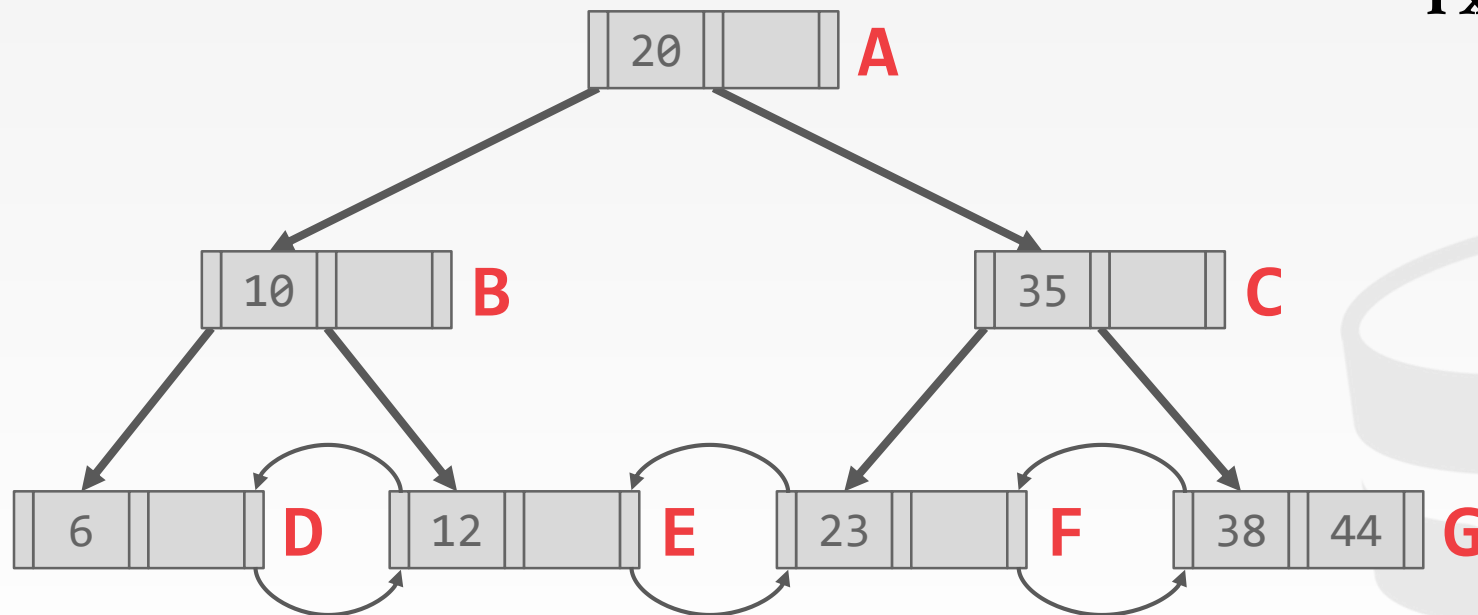
INSERT(25)

PROBLEM SCENARIO #2

Txn #1:



[12, 23]

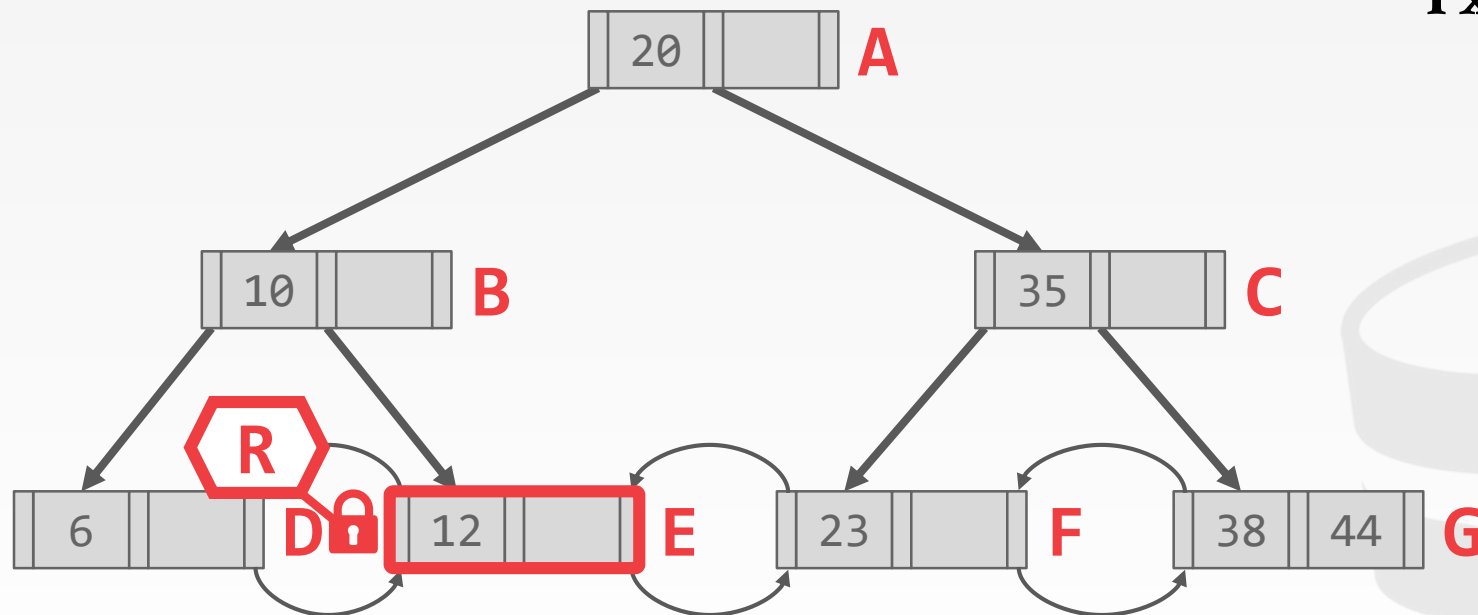


PROBLEM SCENARIO #2

Txn #1:



[12, 23]

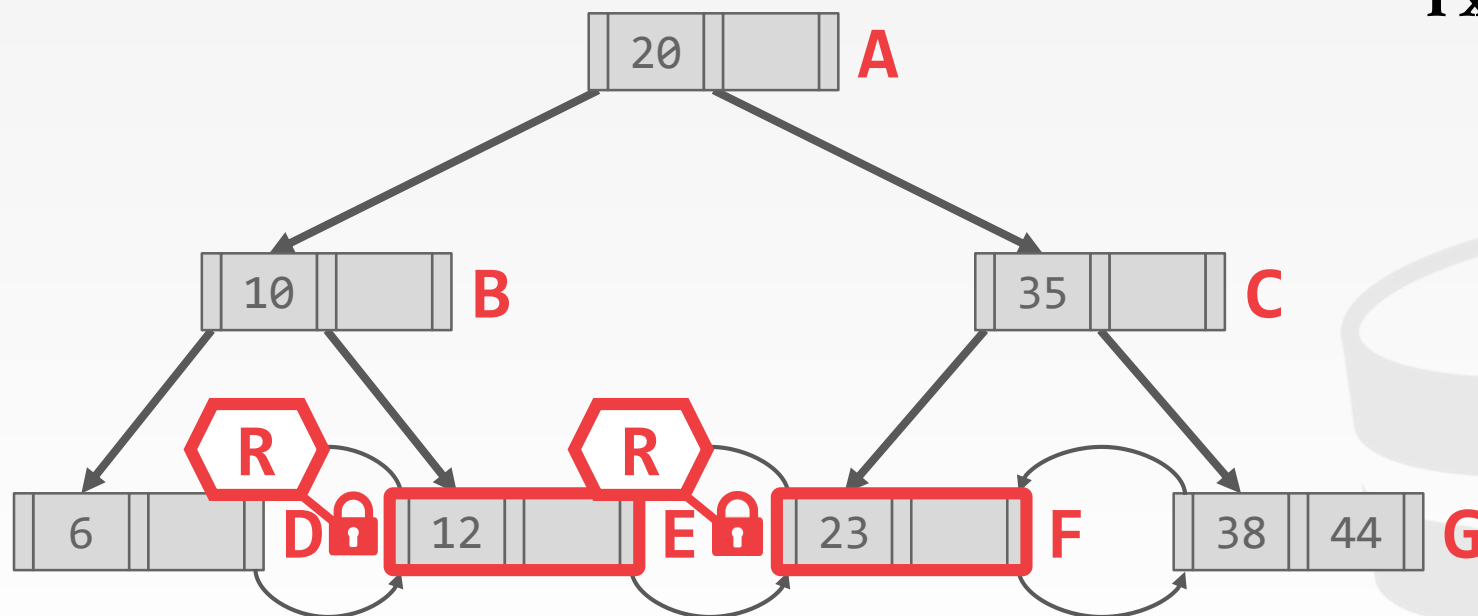


PROBLEM SCENARIO #2

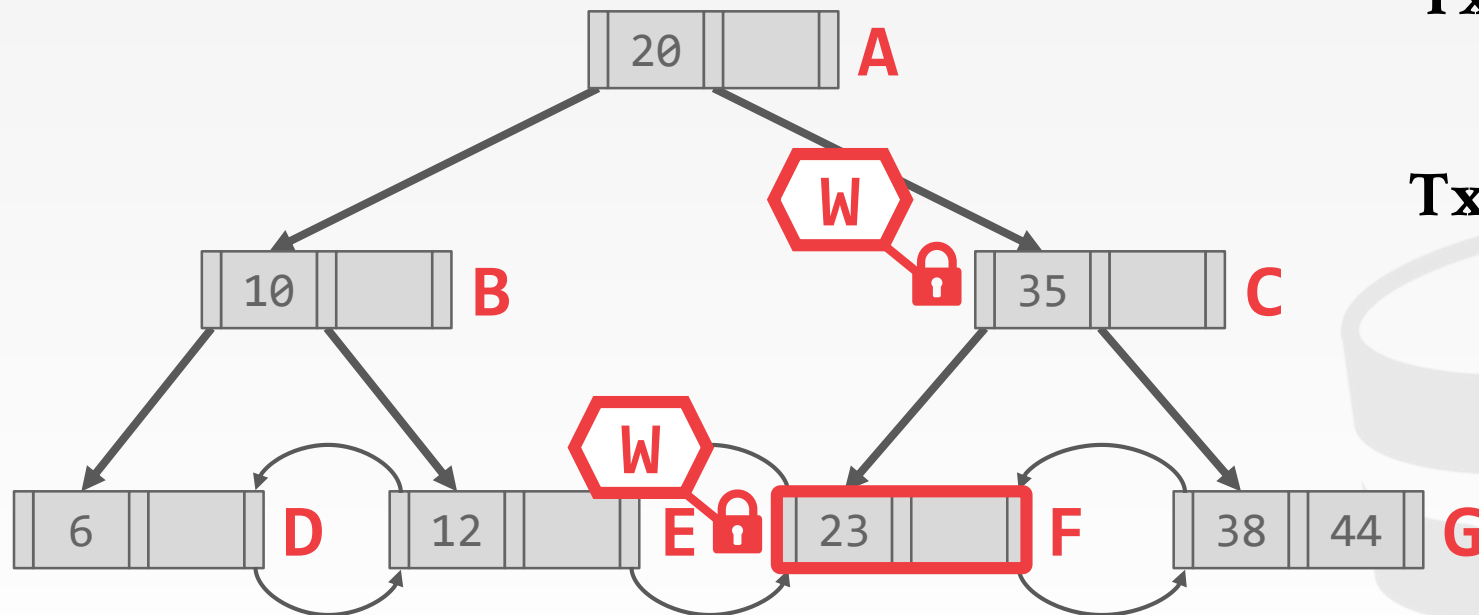
Txn #1:



[12, 23]



PROBLEM SCENARIO #2



Txn #1:



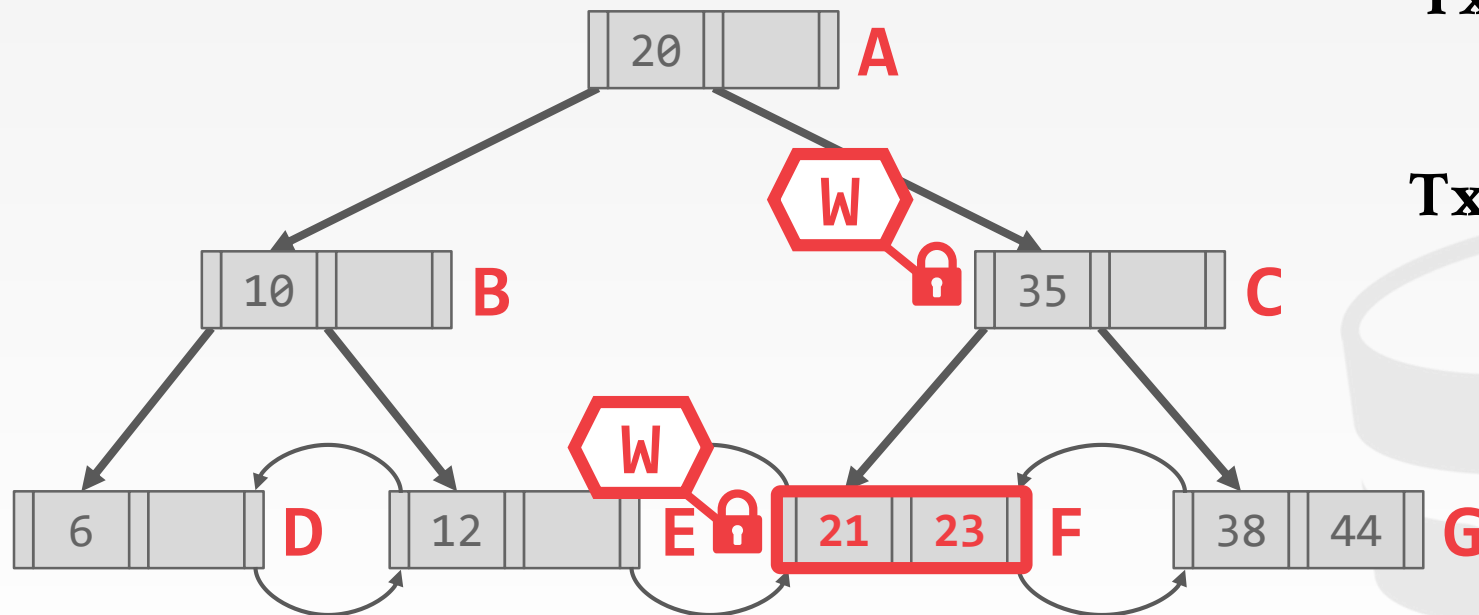
[12, 23]

Txn #2:



INSERT(21)

PROBLEM SCENARIO #2



Txn #1:



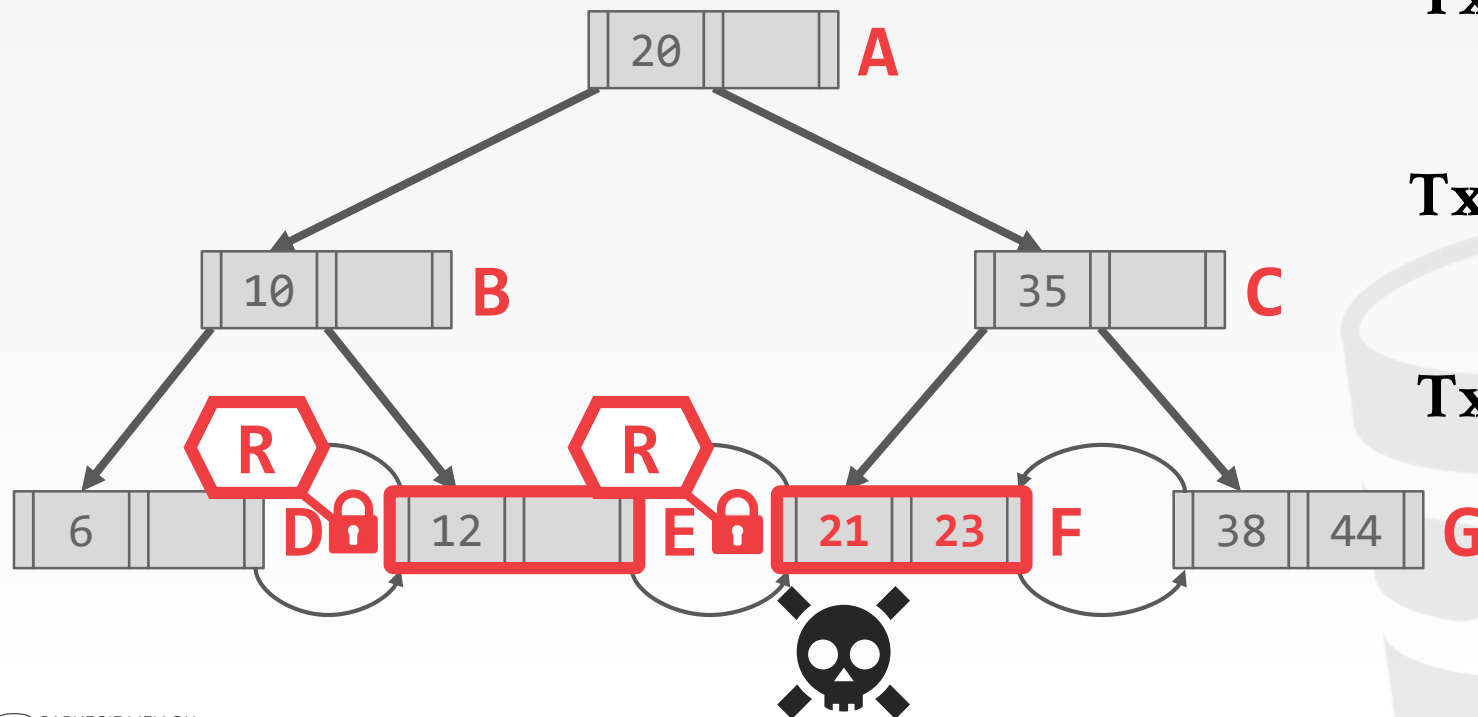
[12, 23]

Txn #2:



INSERT(21)

PROBLEM SCENARIO #2



Txn #1:



[12, 23]

Txn #2:



INSERT(21)

Txn #1:



[12, 23]

INDEX LOCKS

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

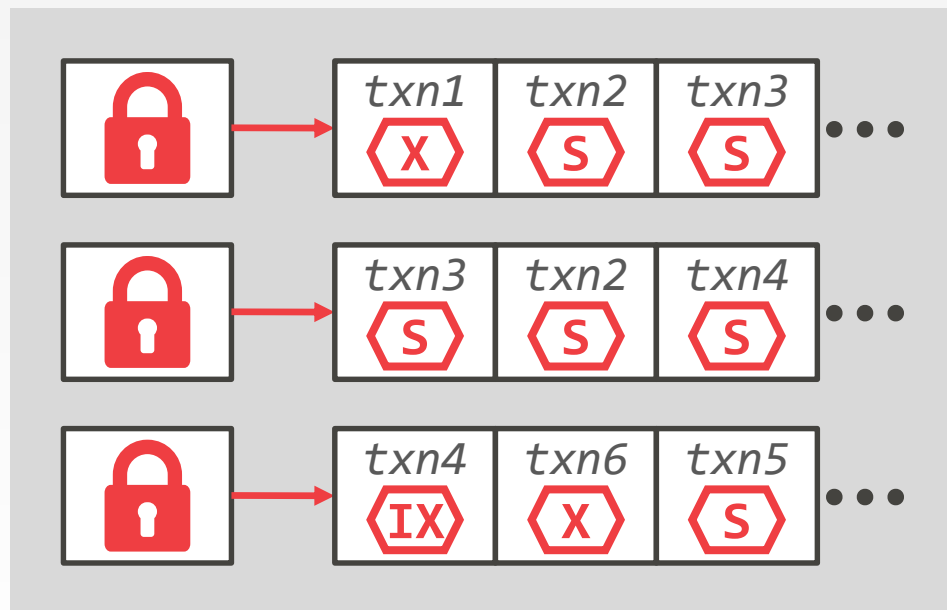
Difference with index latches:

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

Can be used with any order-preserving index.

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.



PREDICATE LOCKS

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

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



Records in Table "account"



name='Biggie'



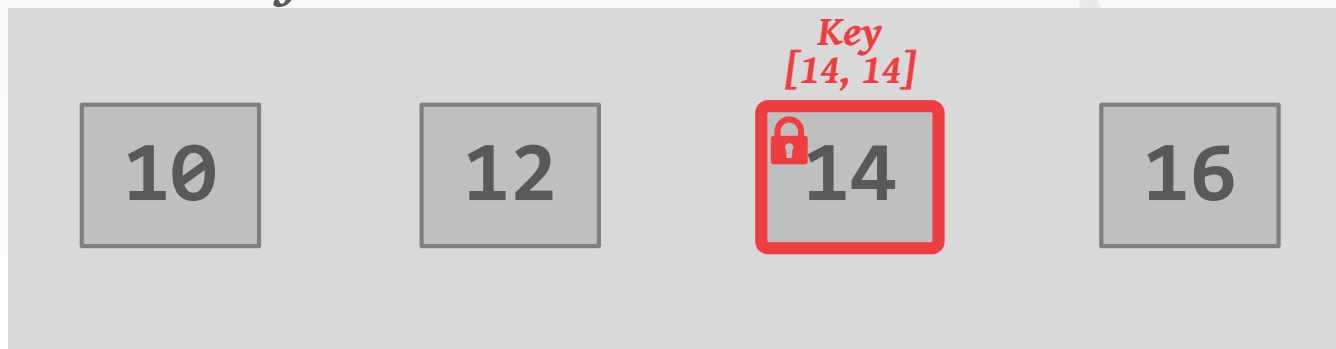
name='Biggie' ^
balance=100

KEY-VALUE LOCKS

Locks that cover a single key value.

Need “virtual keys” for non-existent values.

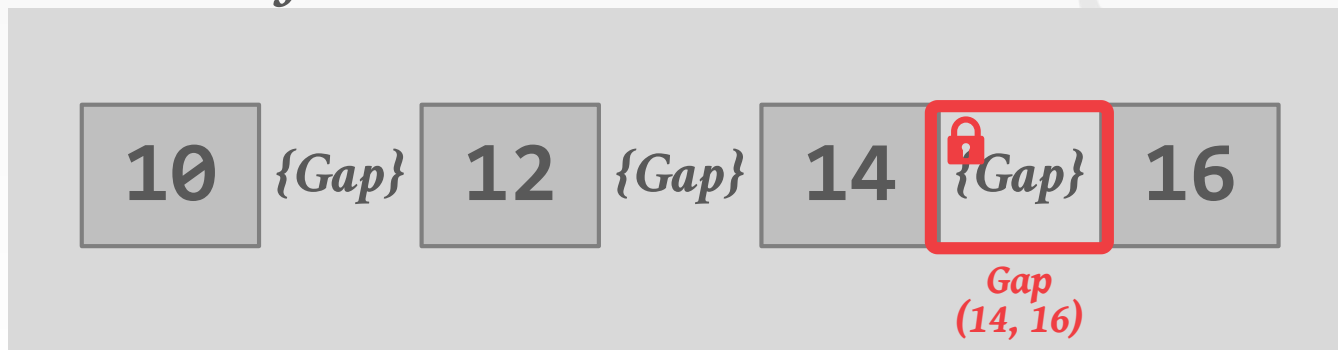
B+Tree Leaf Node



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



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.

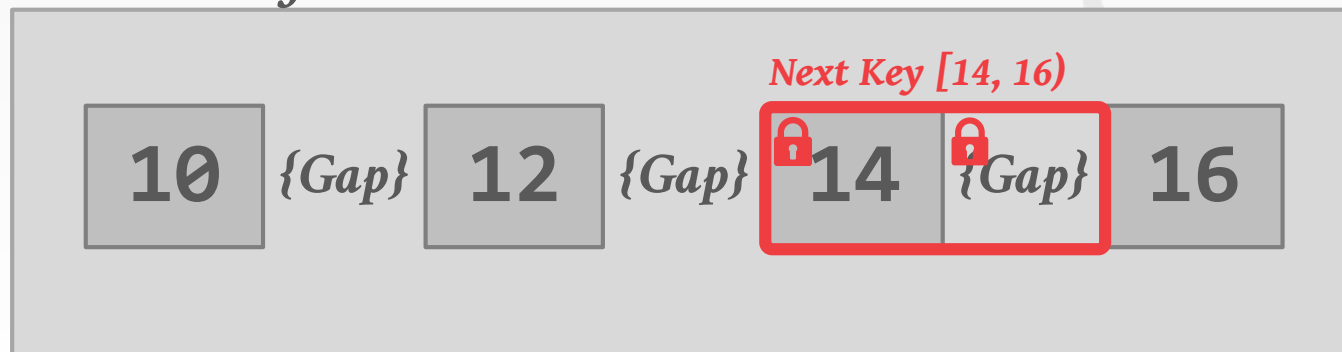


KEY-RANGE LOCKS

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)

B+Tree Leaf Node

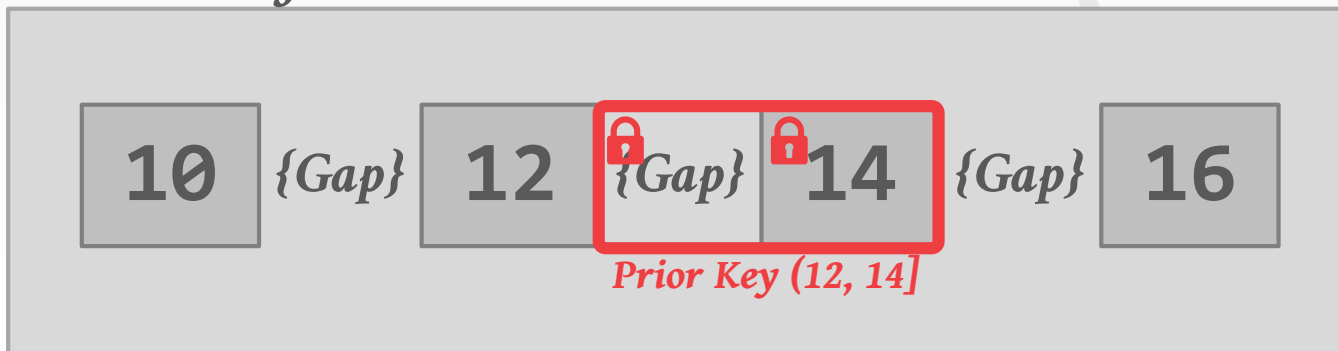


KEY-RANGE LOCKS

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)

B+Tree Leaf Node

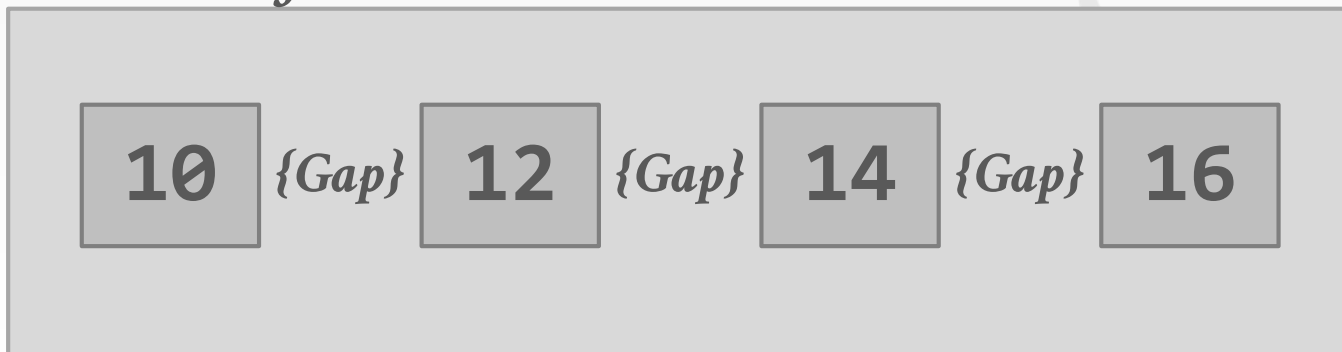


HIERARCHICAL LOCKING

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

→ Reduces the number of visits to lock manager.

B+Tree Leaf Node



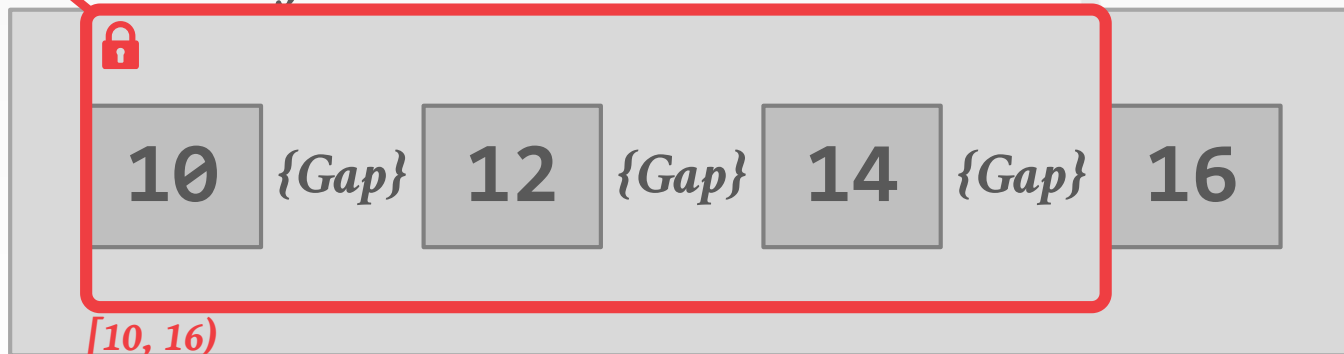
HIERARCHICAL LOCKING

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

→ Reduces the number of visits to lock manager.



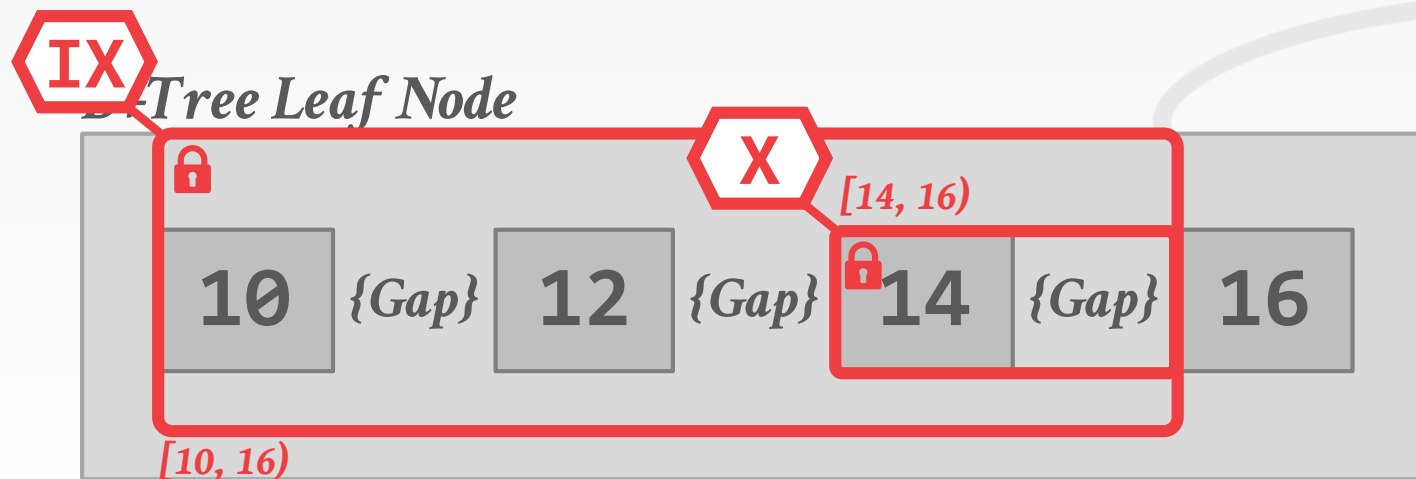
B+Tree Leaf Node



HIERARCHICAL LOCKING

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

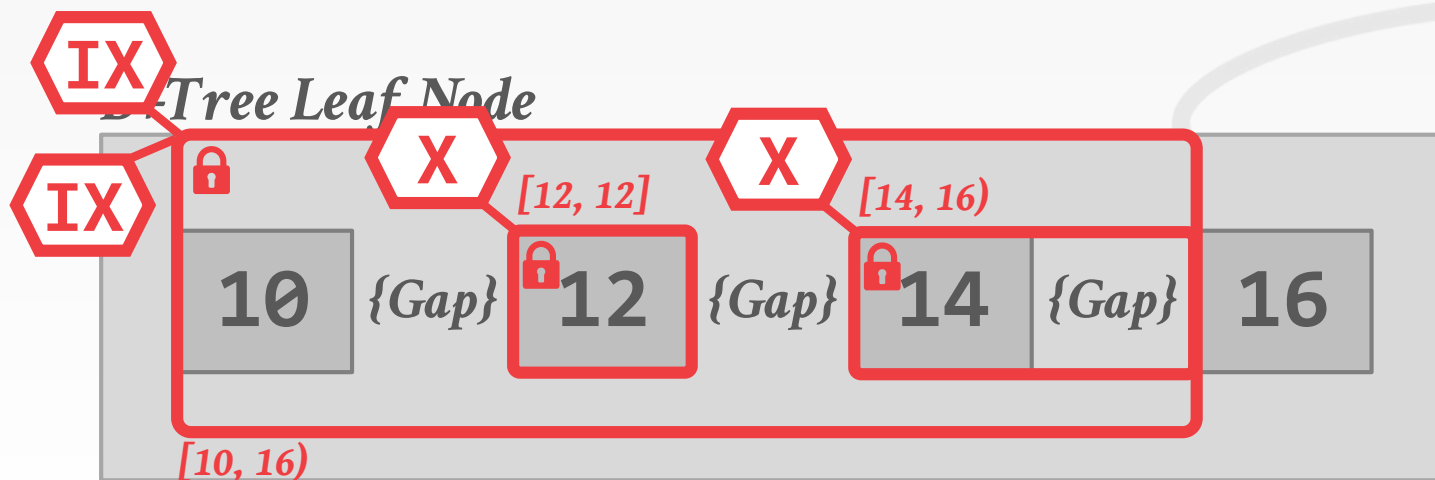
→ Reduces the number of visits to lock manager.



HIERARCHICAL LOCKING

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

→ Reduces the number of visits to lock manager.



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.



ANDY'S
**TIPS FOR
PROFILING**

MOTIVATION

Consider a program with functions **foo** and **bar**.

How can we speed it up with only a debugger ?

- Randomly pause it during execution
- Collect the function call stack



RANDOM PAUSE METHOD

Consider this scenario

- Collected 10 call stack samples
- Say 6 out of the 10 samples were in **foo**

What percentage of time was spent in **foo**?

- Roughly 60% of the time was spent in **foo**
- Accuracy increases with # of samples



AMDAHL'S LAW

Say we optimized **foo** to run two times faster

What's the expected overall speedup ?

→ 60% of time spent in **foo** drops in half

→ 40% of time spent in **bar** unaffected

By Amdahl's law, overall speedup = $\frac{1}{\frac{p}{s} + (1-p)}$

→ **p** = percentage of time spent in optimized task

→ **s** = speed up for the optimized task

→ Overall speedup = $\frac{1}{\frac{0.6}{2} + 0.4} = 1.4$ times faster

PROFILING TOOLS FOR REAL

Choice #1: Valgrind

→ Heavyweight binary instrumentation framework with different tools to measure different events.

Choice #2: Perf

→ Lightweight tool that uses hardware counters to capture events during execution.



CHOICE #1: VALGRIND

Instrumentation framework for building dynamic analysis tools.

- **memcheck**: a memory error detector
- **callgrind**: a call-graph generating profiler
- **massif**: memory usage tracking.



KCACHGRIND

Using callgrind to profile the index test and Peloton in general:

```
$ valgrind --tool=callgrind --trace-children=yes  
./relwithdebinfo/concurrent_read_benchmark
```

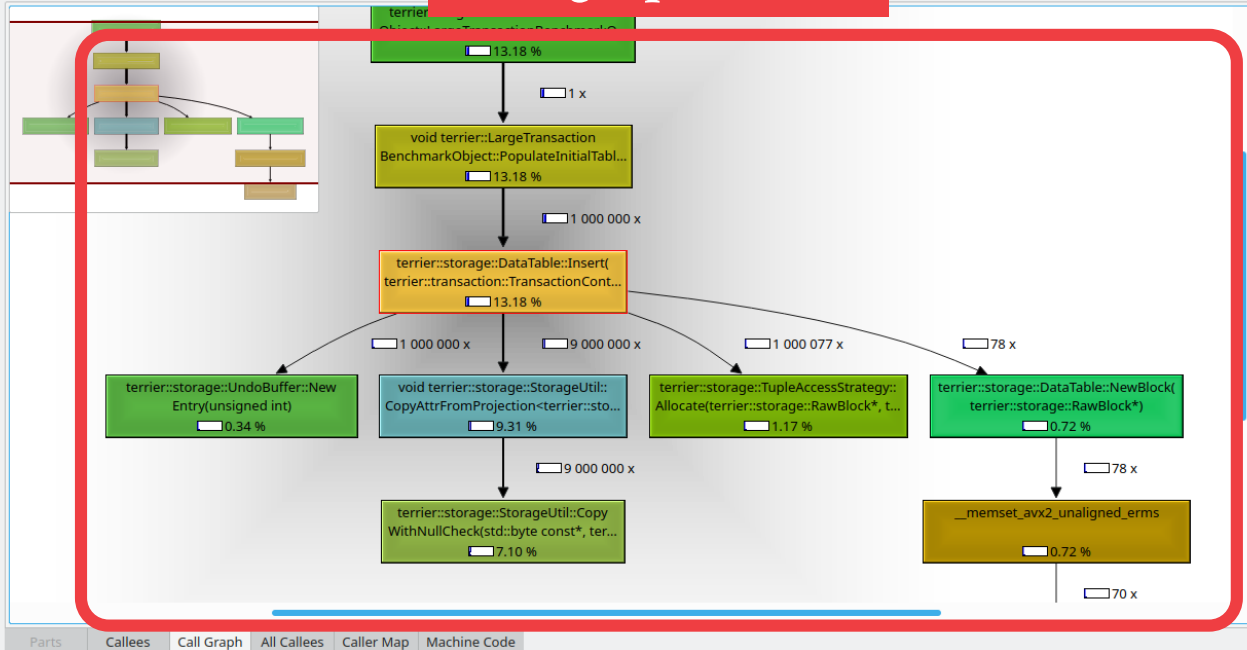
Profile data visualization tool:

```
$ kcachegrind callgrind.out.12345
```

Cumulative Time Distribution

Incl.	Self	Called	Function	Local
63.32	0.00	(0)	0x0000000000001090	ld-2.2
63.30	0.00	1	_start	concl
63.30	0.00	1	(below main)	libc-2
63.28	0.00	1	main	concl
63.28	0.00	1	benchmark::RunSpecifiedB...	concl
63.28	0.00	1	benchmark::RunSpecifiedB...	concl
63.28	0.00	1	terrier::ConcurrentReadBe...	concl
63.27	0.00	1	terrier::LargeTransactionBe...	concl
63.27	10.36	1	void terrier::LargeTransacti...	concl
36.68	0.00	21	start_thread	libpth
36.68	0.00	33	0x000000000000bd570	libstd
35.99	35.99	72 000 000	unsigned char std::uniform...	concl
32.73	0.00	32	std::thread::State_impl<st...	concl
32.73	0.00	16	std::Function_handler<voi...	concl
32.73	0.16	16	std::Function_handler<voi...	concl
26.49	2.80	1 000 000	terrier::LargeTransactionBe...	concl
24.39	0.00	12	clone'2	libc-2
24.39	0.00	12	start_thread'2	libpth
13.18	1.38	1 000 000	terrier::storage::DataTabl...	concl
13.13	0.37	1 000 000	std::Function_handler<voi...	concl
10.22	0.01	1 000 000	terrier::storage::DataTabl...	concl
10.22	1.01	1 000 000	bool terrier::storage::DataT...	concl
9.31	2.21	9 000 000	void terrier::storage::Stora...	concl
8.44	3.79	9 000 000	void terrier::storage::Stora...	concl
7.10	7.10	9 000 000	terrier::storage::StorageUtil...	concl
6.33	0.14	8 010 333	operator delete(void*)	libstd
6.19	6.19	8 010 990	free	libc-2
5.32	0.84	8 010 335	operator new(unsigned long)	libstd
4.76	0.28	1 000 000	terrier::RandomWorkloadTr...	concl

Callgraph View



CHOICE #2: PERF

Tool for using the performance counters subsystem in Linux.

- **-e** = sample the event cycles at the user level only
- **-c** = collect a sample every 2000 occurrences of event

```
$ perf record -e cycles:u -c 2000  
./relwithdebinfo/concurrent_read_benchmark
```

Uses counters for tracking events

- On counter overflow, the kernel records a sample
- Sample contains info about program execution

PERF VISUALIZATION

We can also use **perf** to visualize the generated profile for our application.

```
$ perf report
```

There are also third-party visualization tools:

→ Hotspot

Samples: 9M of event 'cycles:u', Event count (approx.): 18388130000

Overhead	Command	Shared Object	Symbol
17.89%	concurrent_read	concurrent_read_benchmark	[.] _ZN7terrier7storage11StorageUtil17CopyWithNullCheckINS0_12Projecte
9.41%	concurrent_read	concurrent_read_benchmark	[.] _ZN7terrier11transaction18TransactionManager6CommitEPNS0_18Transac
9.36%	concurrent_read	concurrent_read_benchmark	[.] _ZN7terrier11transaction18TransactionManager16BeginTransactionEPNS
8.10%	concurrent_read	concurrent_read_benchmark	[.] _ZNSt24uniform_int_distributionIhEclIst26linear_congruential_engin
5.27%	concurrent_read	concurrent_read_benchmark	[.] _ZN7terrier7storage11StorageUtil22CopyAttrIntoProjectionINS0_12Pro
3.53%	concurrent_read	libc-2.27.so	[.] _int_malloc
3.28%	concurrent_read	libc-2.27.so	[.] __sched_yield
3.08%	concurrent_read	libc-2.27.so	[.] cfree@GLIBC_2.2.5
3.06%	concurrent_read	concurrent_read_benchmark	[.] _ZNSt17_Function_handlerIFvvEZN7terrier31LargeTransactionBenchmark
2.87%	concurrent_read	concurrent_read_benchmark	[.] _ZNK7terrier7storage9DataTable24AtomicallyReadVersionPtrENS0_9Tupl
2.72%	concurrent_read	concurrent_read_benchmark	[.] _ZNKSt10_HashTableIN7terrier6common15StrongTypeAliasINS0_11transac
2.45%	concurrent_read	concurrent_read_benchmark	[.] _ZN7terrier7storage16GarbageCollector18ProcessUnlinkQueueEv
1.86%	concurrent_read	concurrent_read_benchmark	[.] _ZN7terrier7storage11StorageUtil17CopyWithNullCheckEPKSt4byteRKNS0
1.74%	concurrent_read	libtbb.so.2	[.] 0x00000000000018ac4
1.58%	concurrent_read	libc-2.27.so	[.] malloc
1.20%	concurrent_read	concurrent_read_benchmark	[.] _ZNSt10_HashTableIN7terrier6common15StrongTypeAliasINS0_11transact
0.99%	concurrent_read	libc-2.27.so	[.] __memset_avx2_unaligned_erms
0.98%	concurrent_read	concurrent_read_benchmark	[.] _ZNSt17_Function_handlerIFvjEZN7terrier31LargeTransactionBenchmark
0.90%	concurrent_read	libtbb.so.2	[.] 0x000000000000185cb
0.89%	concurrent_read	concurrent_read_benchmark	[.] _ZN7terrier31LargeTransactionBenchmarkObject22SimulateOneTransacti
0.83%	concurrent_read	concurrent_read_benchmark	[.] _ZSt18generate_canonicalIdLm53Est26linear_congruential_engineImLm1
0.72%	concurrent_read	concurrent_read_benchmark	[.] _ZN7terrier11transaction18TransactionManager9LogCommitEPNS0_18Tran
	concurrent_read	concurrent_read_benchmark	[.] _ZN7terrier31LargeTransactionBenchmarkObject20PopulateInitialTable
	libtbb.so.2		[.] 0x00000000000018ac6
	[kernel]		[k] 0xffffffff000005e000
	concurrent_read_benchmark		[.] _ZNK7terrier7storage9DataTable16SelectIntoBufferINS0_12ProjectedRo
	[kernel]		[k] 0xffffffff00000e2000

Cumulative Event Distribution

Cannot load tips.txt file, please install perf!

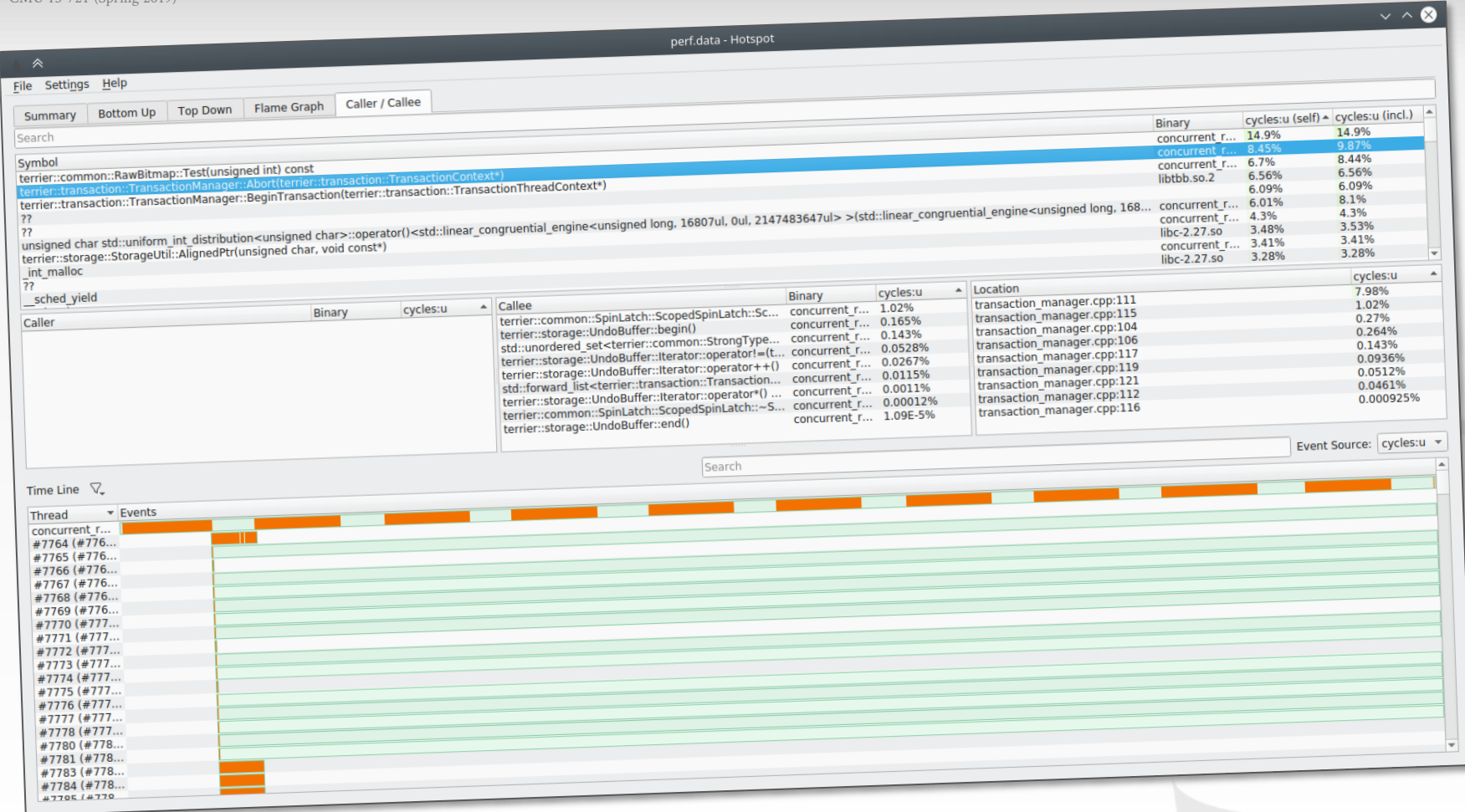
PERF VISUALIZATION

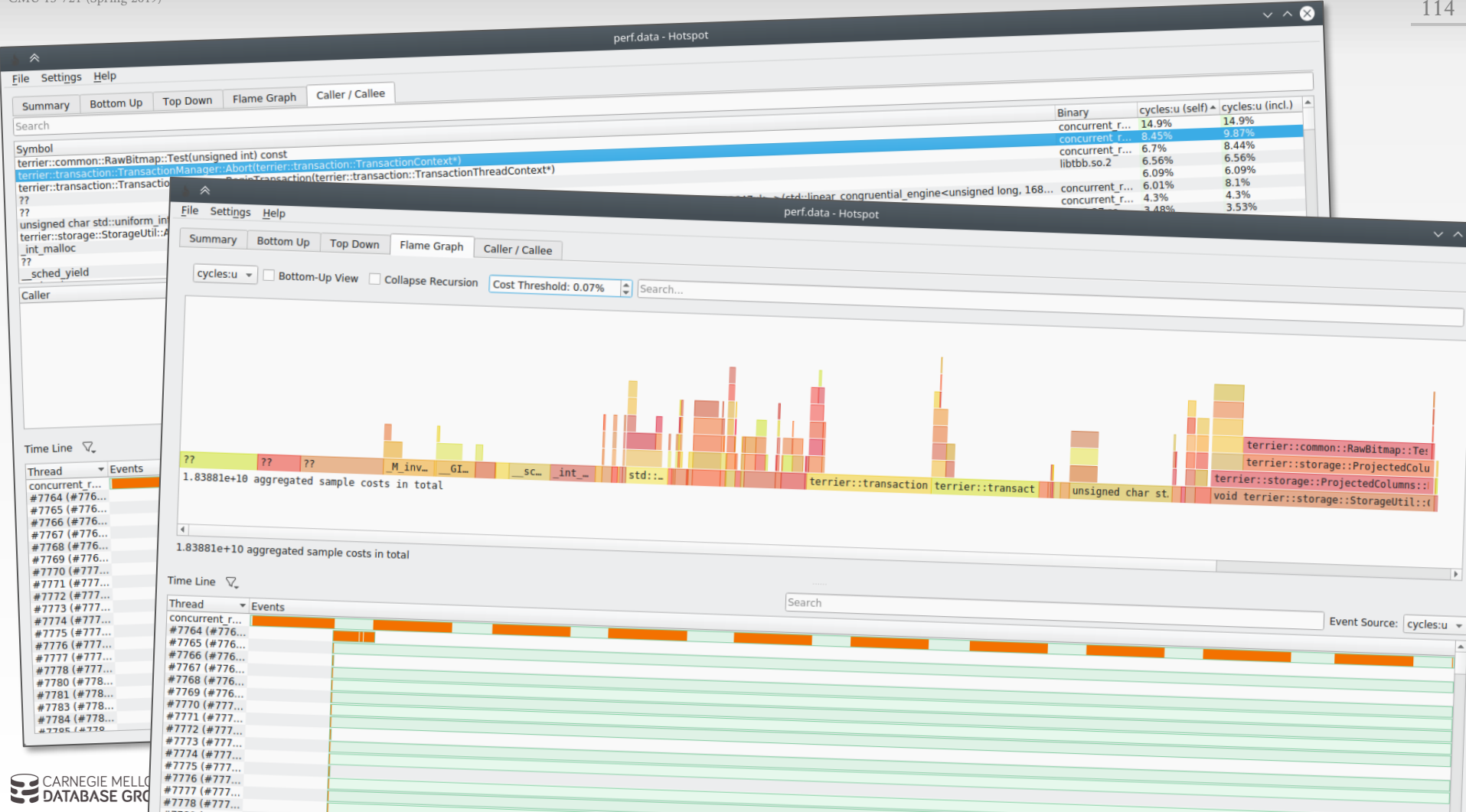
We can also use **perf** to visualize the generated profile for our application.

```
$ perf report
```

There are also third-party visualization tools:

→ Hotspot





PERF EVENTS

Supports several other events like:

- L1-dcache-load-misses
- branch-misses

To see a list of events:

```
$ perf list
```

Another usage example:

```
$ perf record -e cycles,LLC-load-misses -c 2000  
./relwithdebinfo/concurrent_read_benchmark
```

REFERENCES

Valgrind

- [The Valgrind Quick Start Guide](#)
- [Callgrind](#)
- [Kcachegrind](#)
- [Tips for the Profiling/Optimization process](#)

Perf

- [Perf Tutorial](#)
- [Perf Examples](#)
- [Perf Analysis Tools](#)



NEXT CLASS

Index Key Representation

Memory Allocation & Garbage Collection

T-Trees (1980s / TimesTen)

Bw-Tree (Hekaton)

Concurrent Skip Lists (MemSQL)

