Carnegie Mellon University

Index Concurrency Control

Lecture #09

Database Systems
15-445/15-645
Fall 2018

Andy Pavlo
Computer Science
Carnegie Mellon Univ.

ADMINISTRIVIA

Project #1 is due TODAY!

Homework #2 is due Friday Sept 28th @ 11:59pm

Project #2 first checkpoint is due Monday Oct 8th.

OBSERVATION

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

But we need to allow multiple threads to safely access our data structures to take advantage of additional CPU cores.

OBSERVATION

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

But we need to allow multiple threads to safely access our data structures to take advantage of additional CPU cores.

VOLTDB
Doesn't Do This!

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 I see the data that I am 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 I see the data that I am supposed to see?

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

TODAY'S AGENDA

Latch Modes

Index Crabbing/Coupling

Leaf Scans

Delayed Parent Updates

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

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

Source: Goetz Graefe

LOCKS VS. LATCHES

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

Source: Goetz Graefe

LOCKS VS. LATCHES

Lecture 17

	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 are allowed to read the same item at the same time.
→ A thread can acquire the read latch if another thread has it in read mode.

Write Mode

→ Only one thread is allowed to access the item.
→ A thread cannot acquire a write latch if another thread holds the latch in any mode.

Compatibility Matrix

	Read	Write
Read	✔	X
Write	X	X

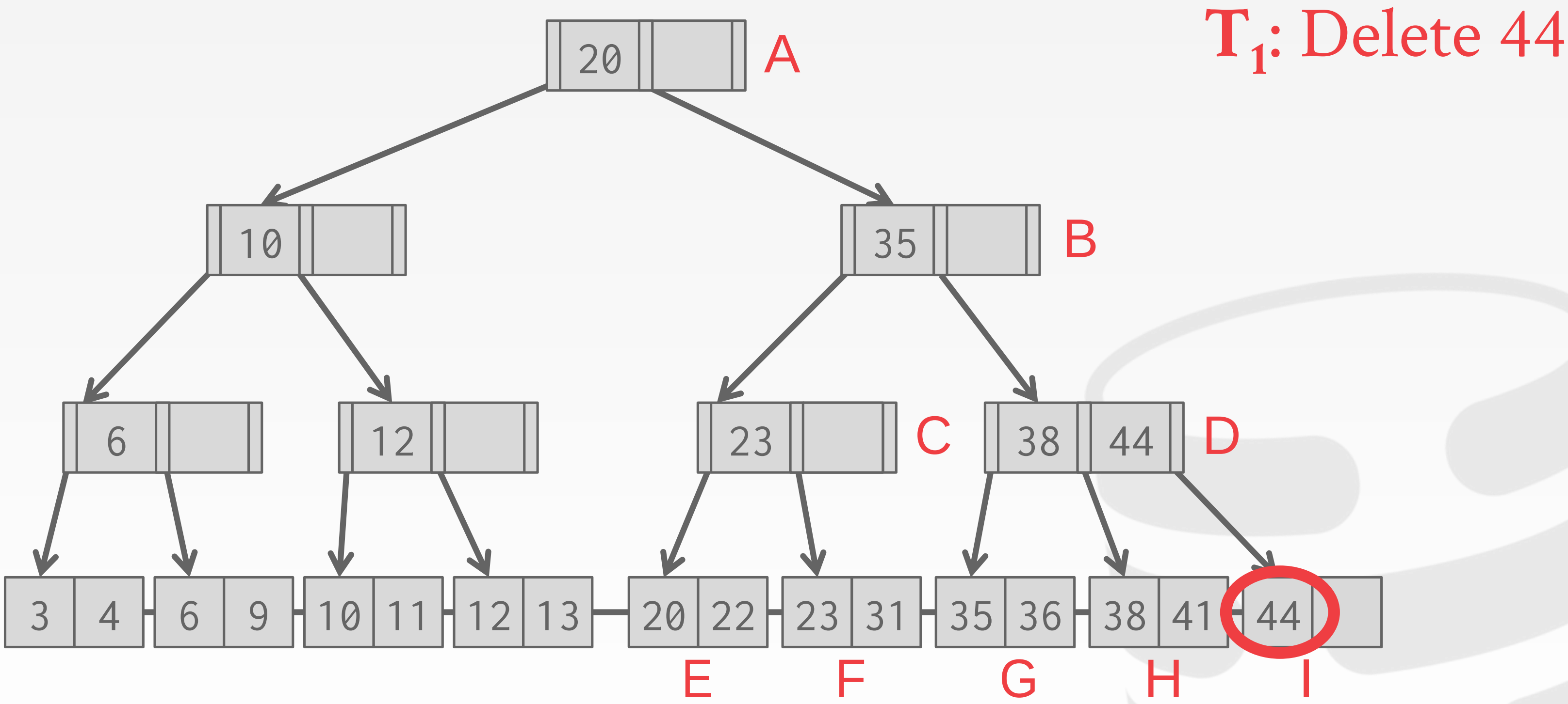
B+TREE CONCURRENCY CONTROL

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

We need to protect from two types of problems:

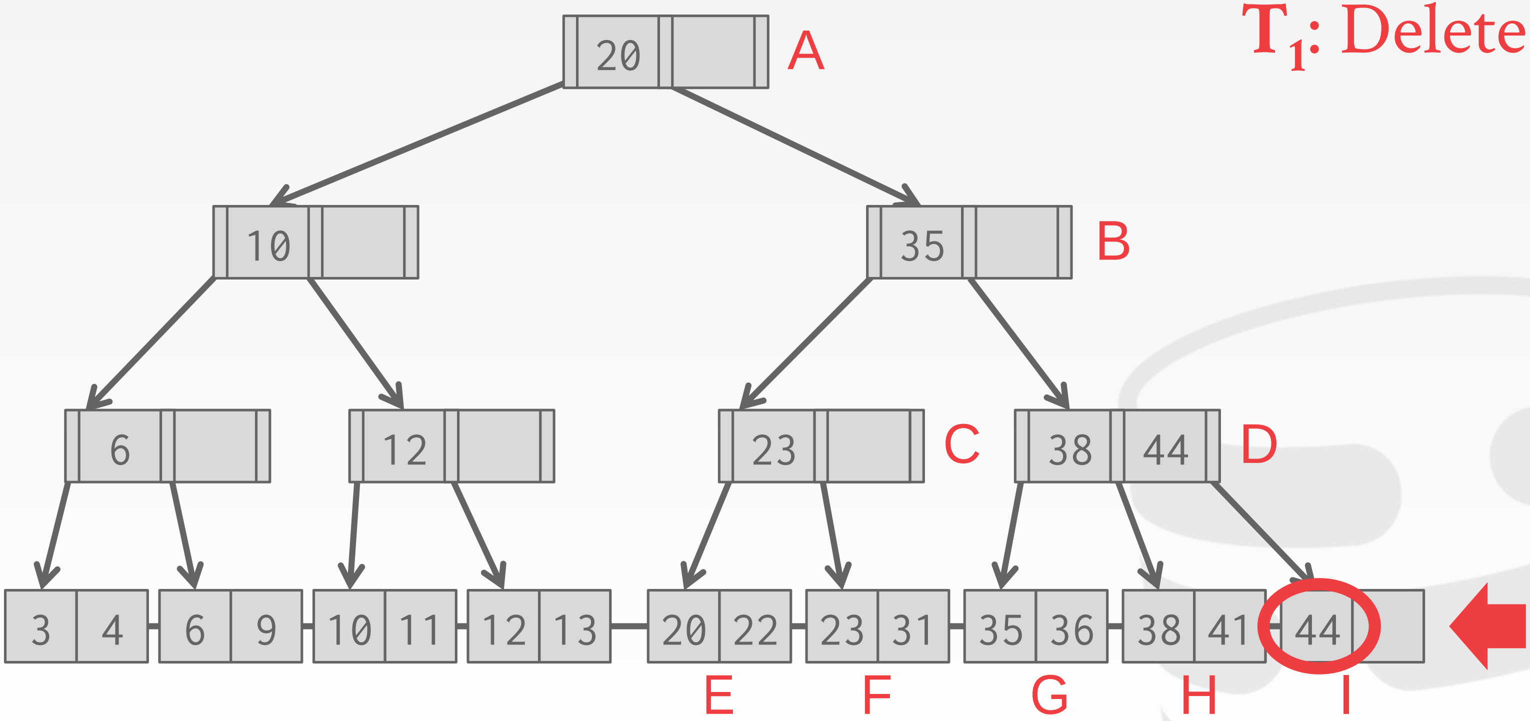
- → Threads trying to modify the contents of a node at the same time.
- → One thread traversing the tree while another thread splits/merges nodes.

B+TREE MULTI-THREADED EXAMPLE

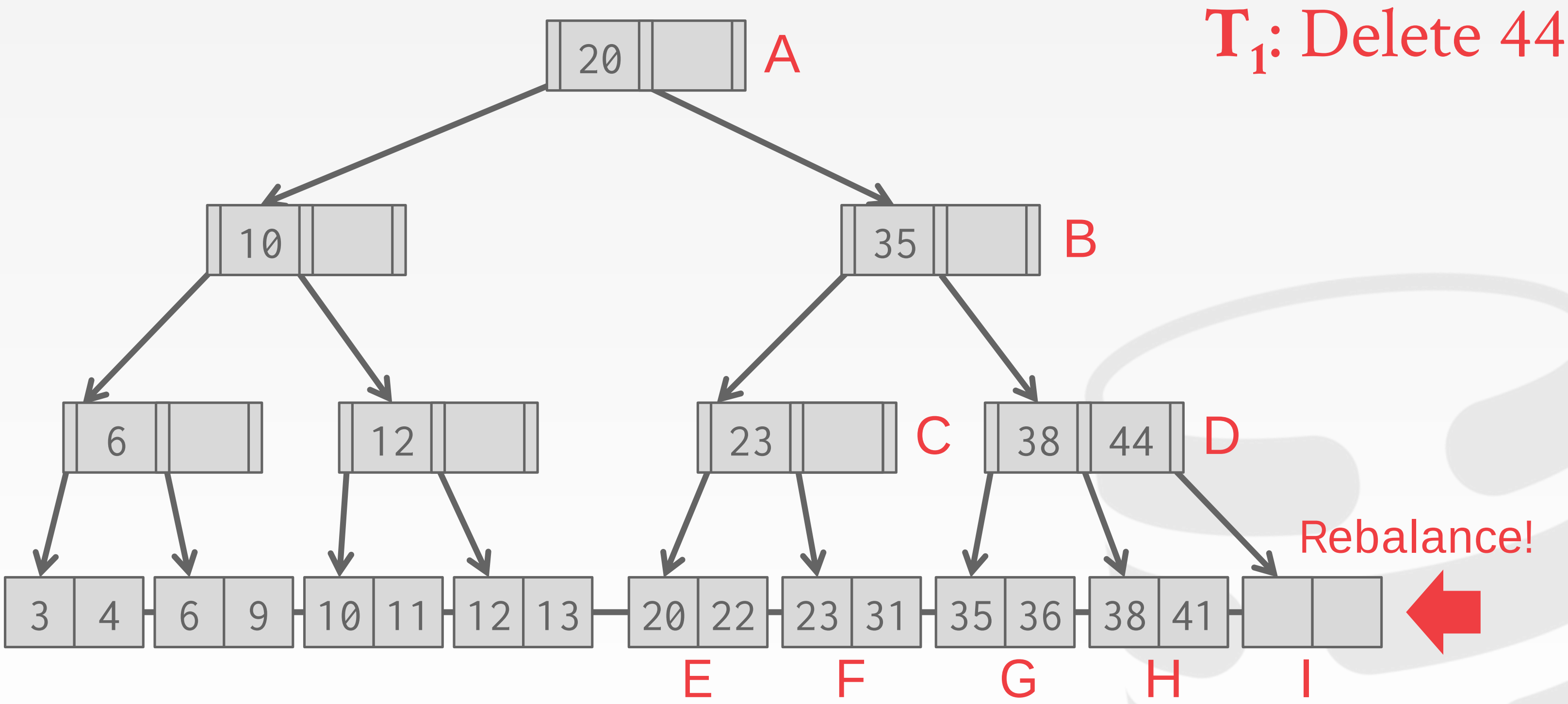


B+TREE MULTI-THREADED EXAMPLE

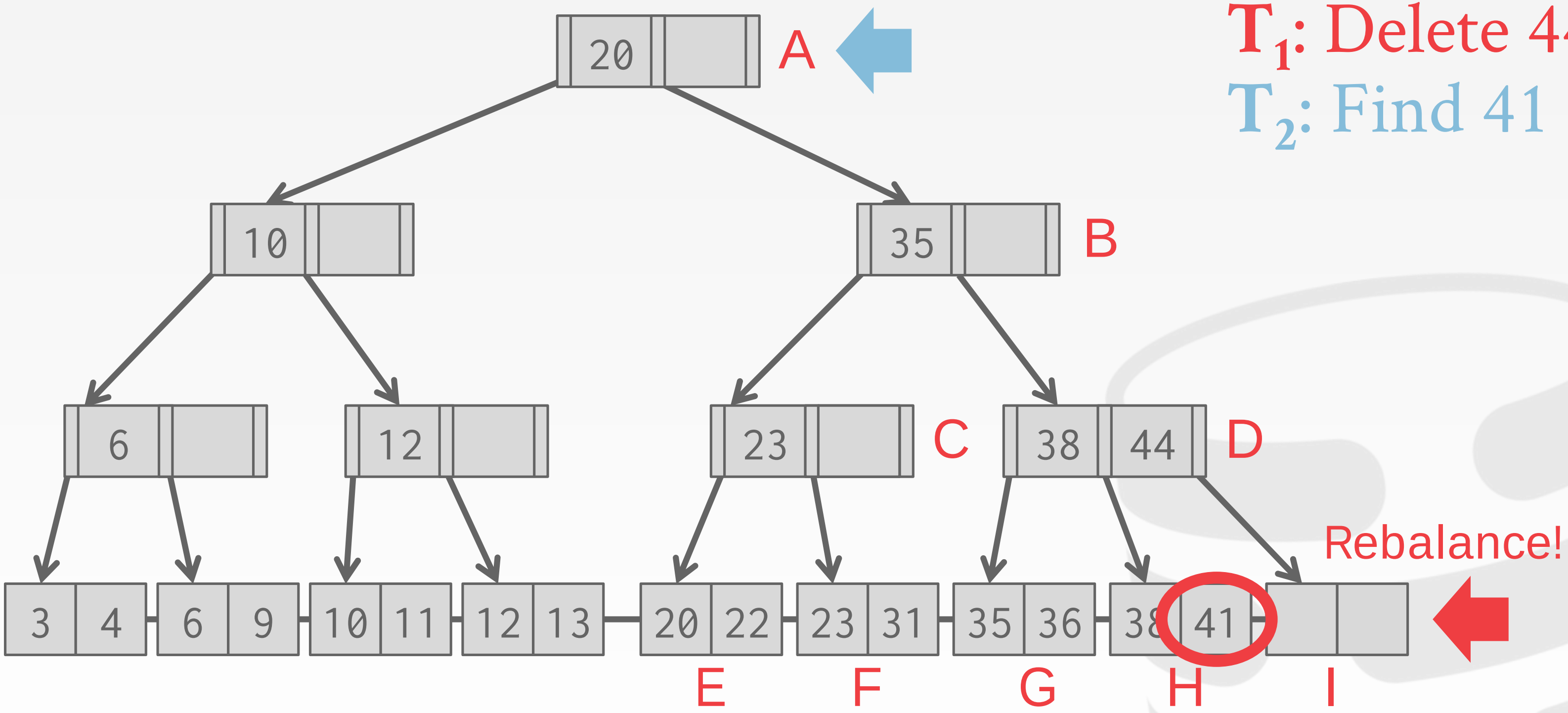
T_1: Delete 44



B+TREE MULTI-THREADED EXAMPLE



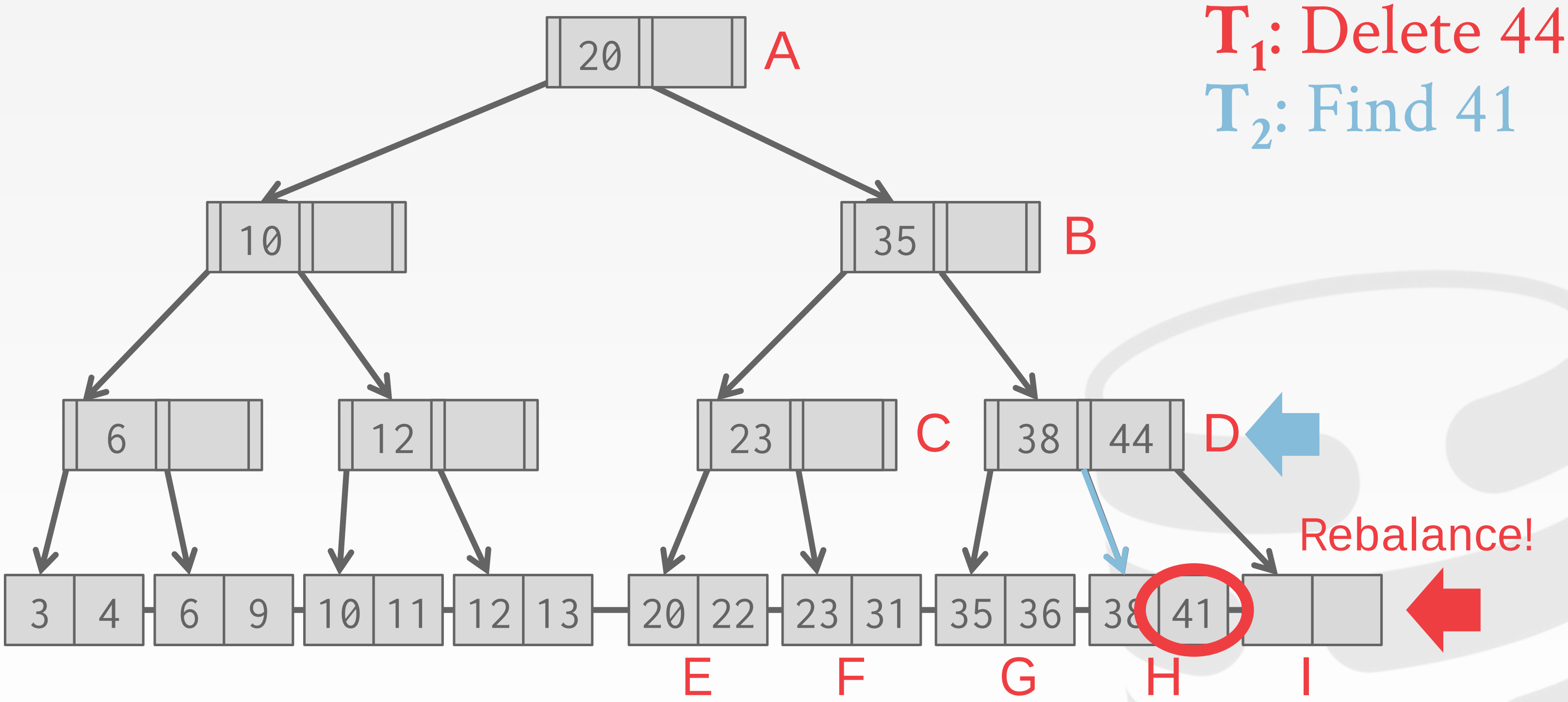
B+TREE MULTI-THREADED EXAMPLE



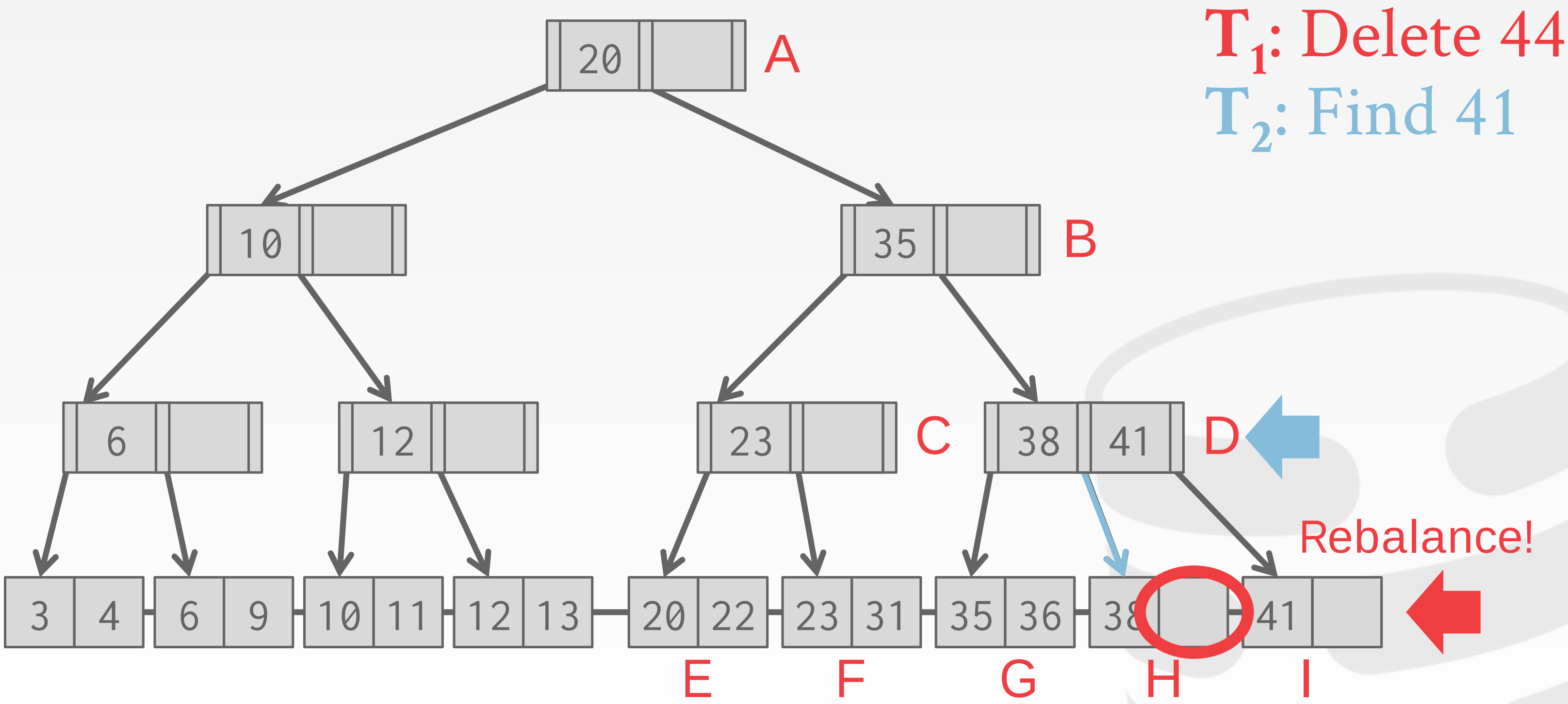
T_1: Delete 44

T_2: Find 41

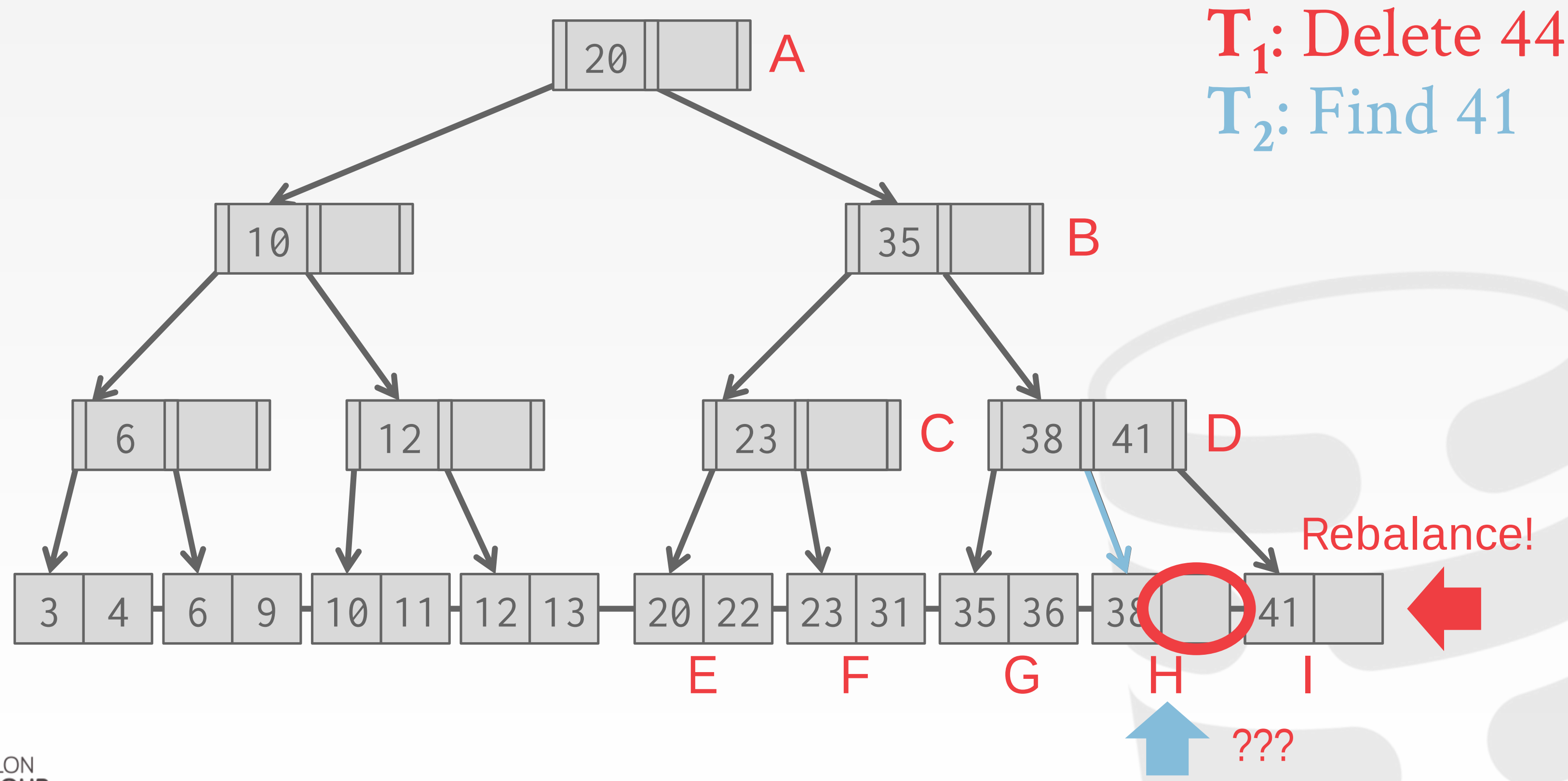
B+TREE MULTI-THREADED EXAMPLE



B+TREE MULTI-THREADED EXAMPLE



B+TREE MULTI-THREADED EXAMPLE



LATCH CRABBING/COUPLING

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

Basic Idea:

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

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

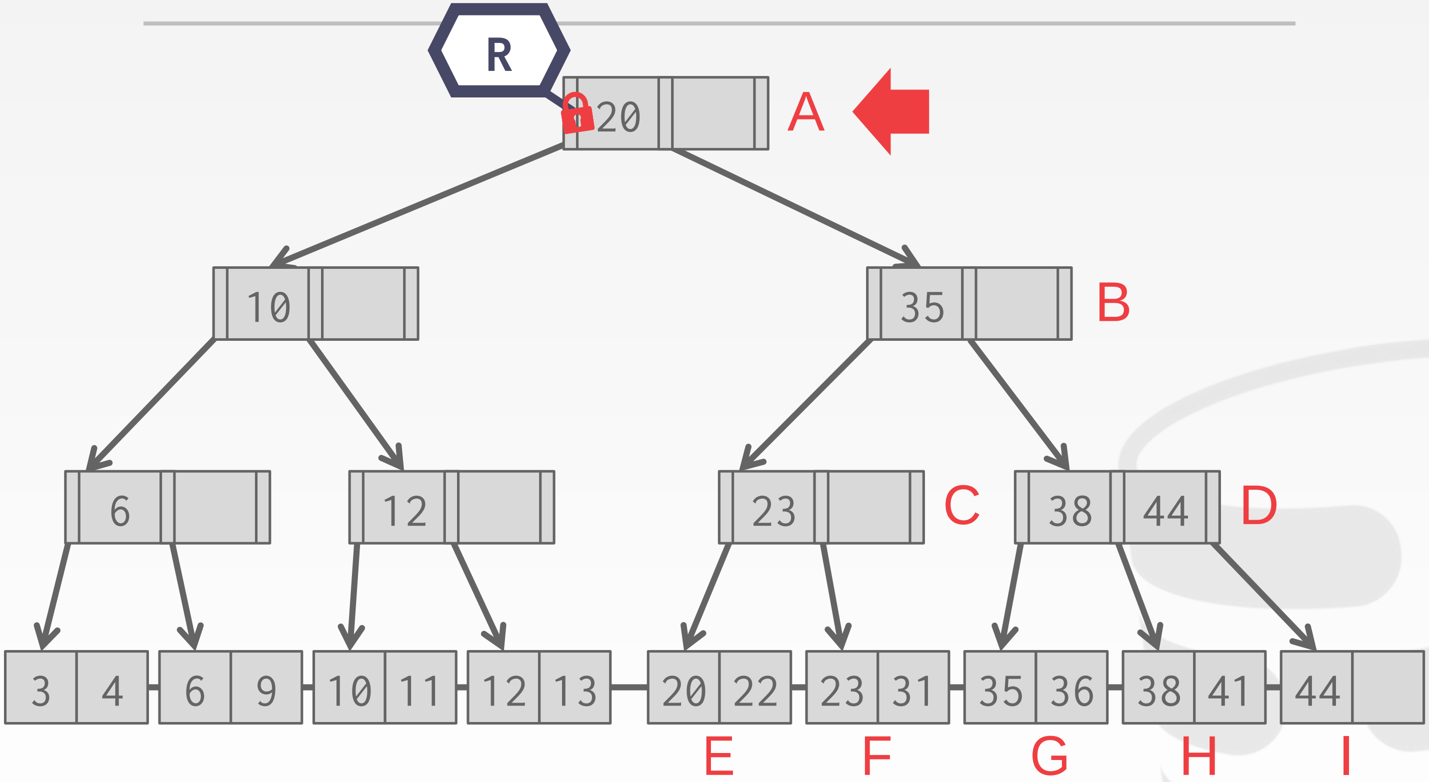
→ Not full (on insertion)
→ More than half-full (on deletion)

LATCH CRABBING/COUPLING

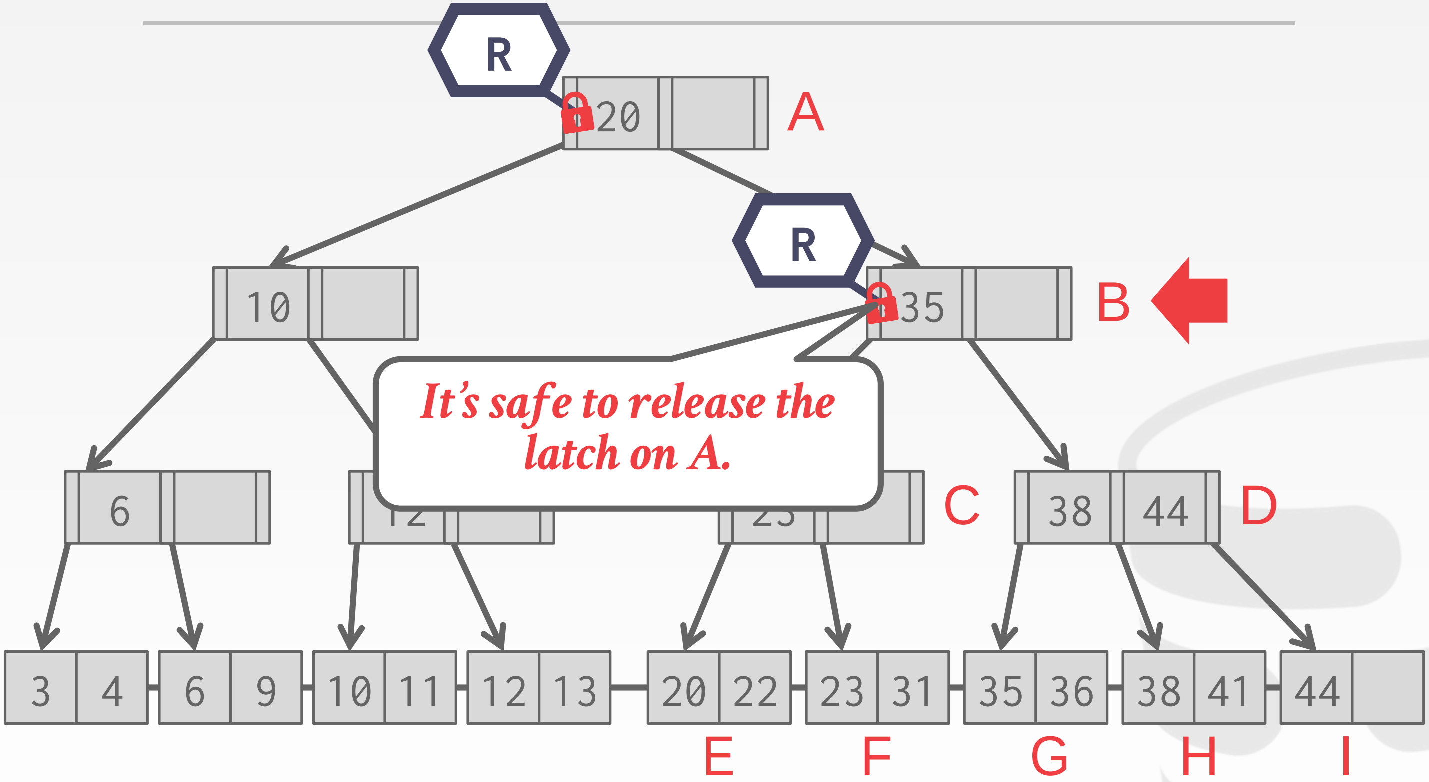
Search: Start at root and go down; repeatedly,
→ Acquire **R** latch on child
→ Then unlatch parent

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

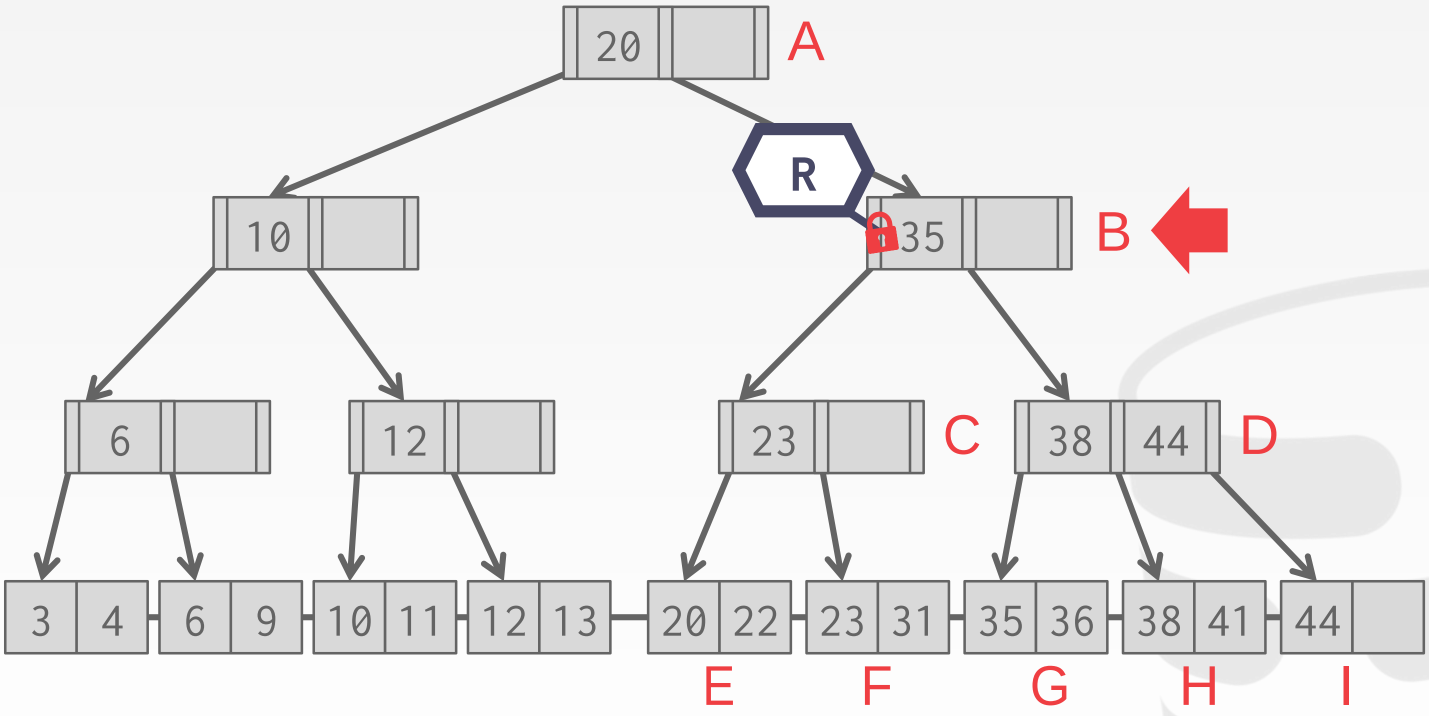
EXAMPLE #1 – SEARCH 38



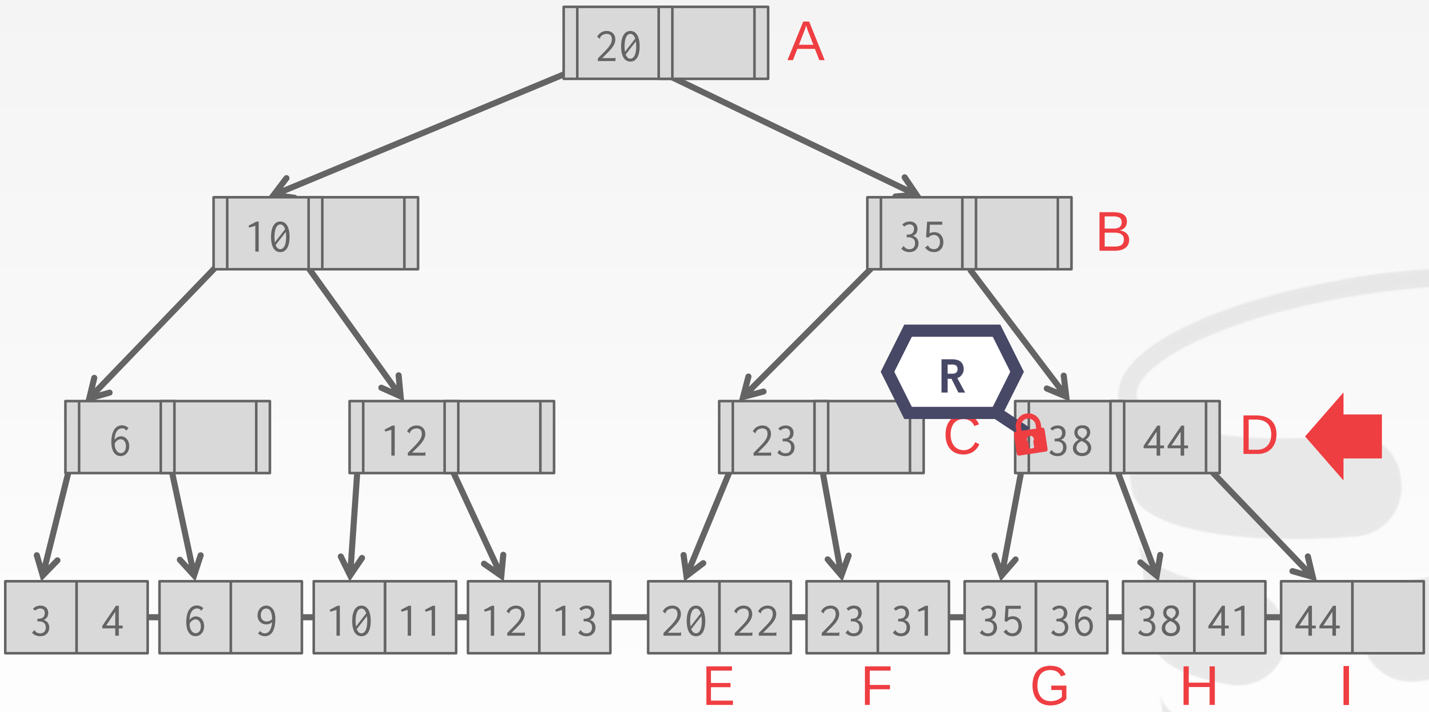
EXAMPLE #1 – SEARCH 38



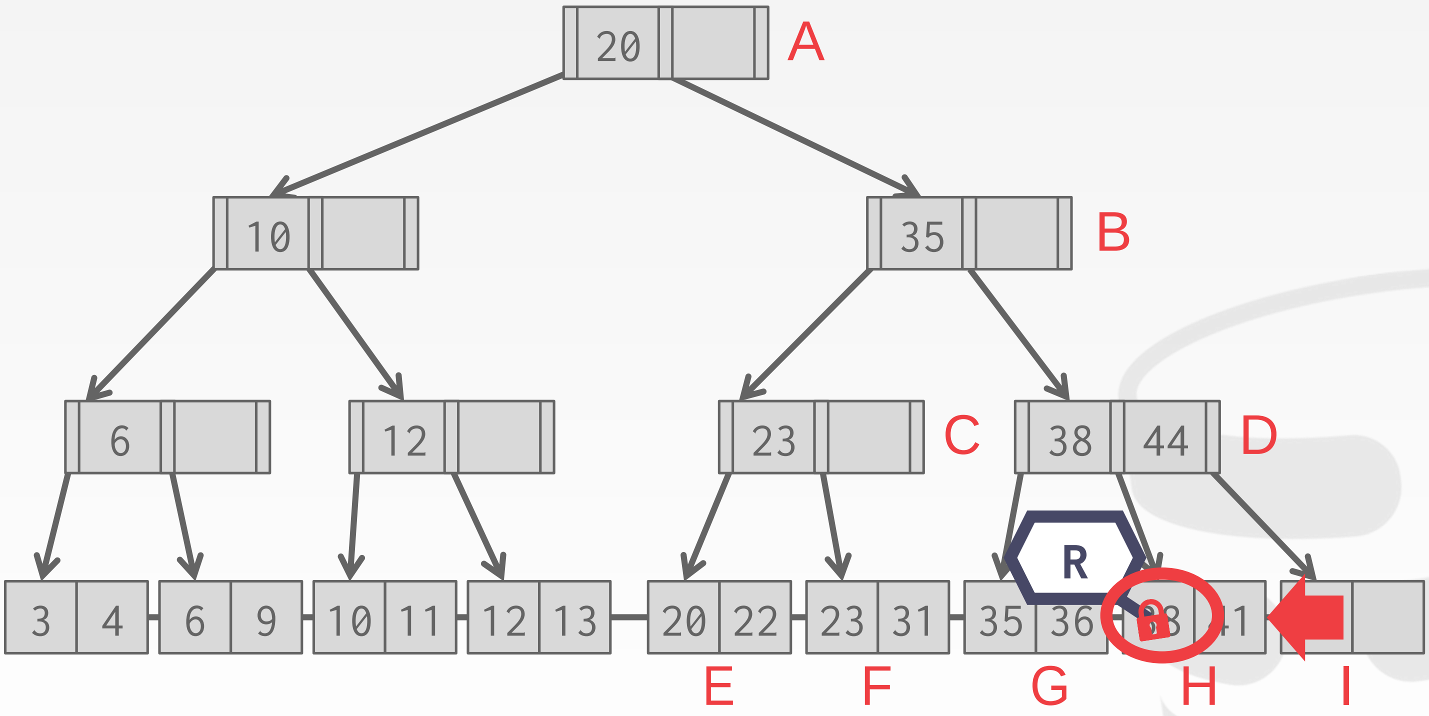
EXAMPLE #1 – SEARCH 38



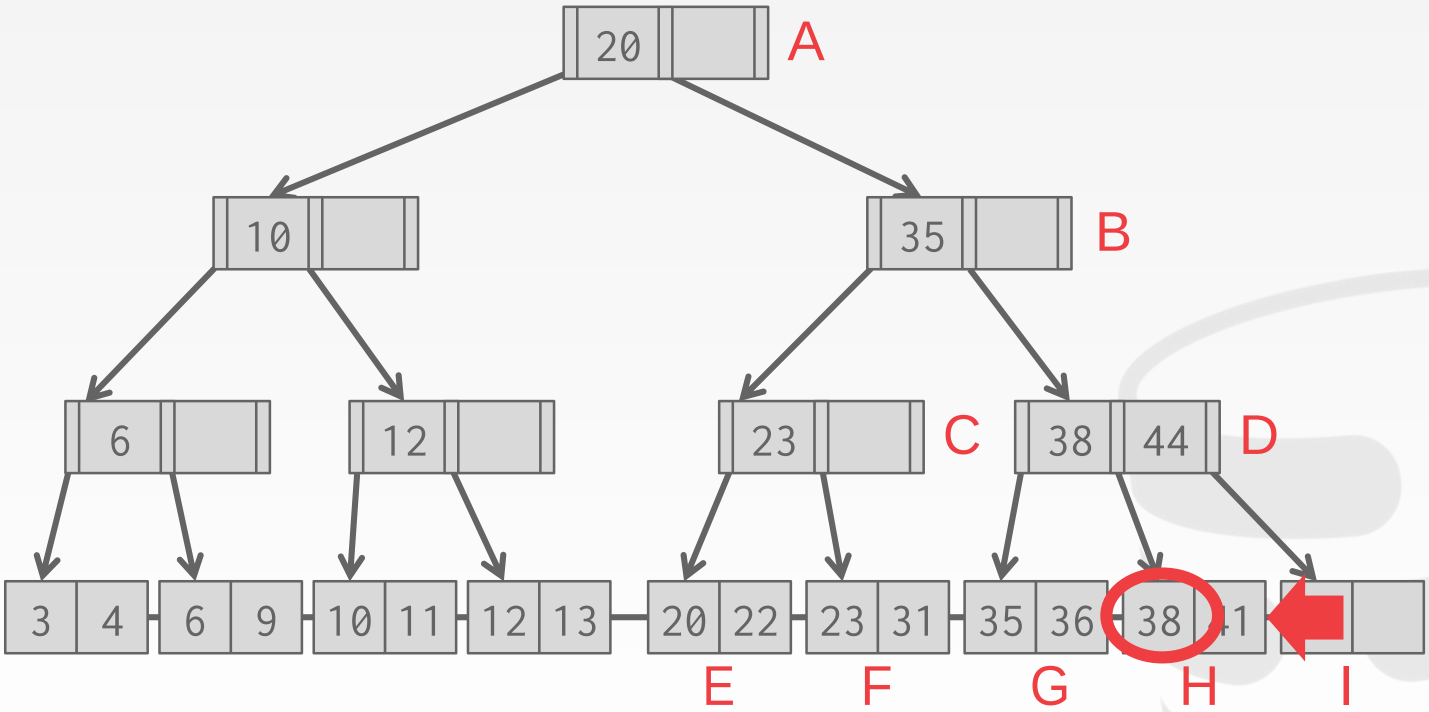
EXAMPLE #1 – SEARCH 38



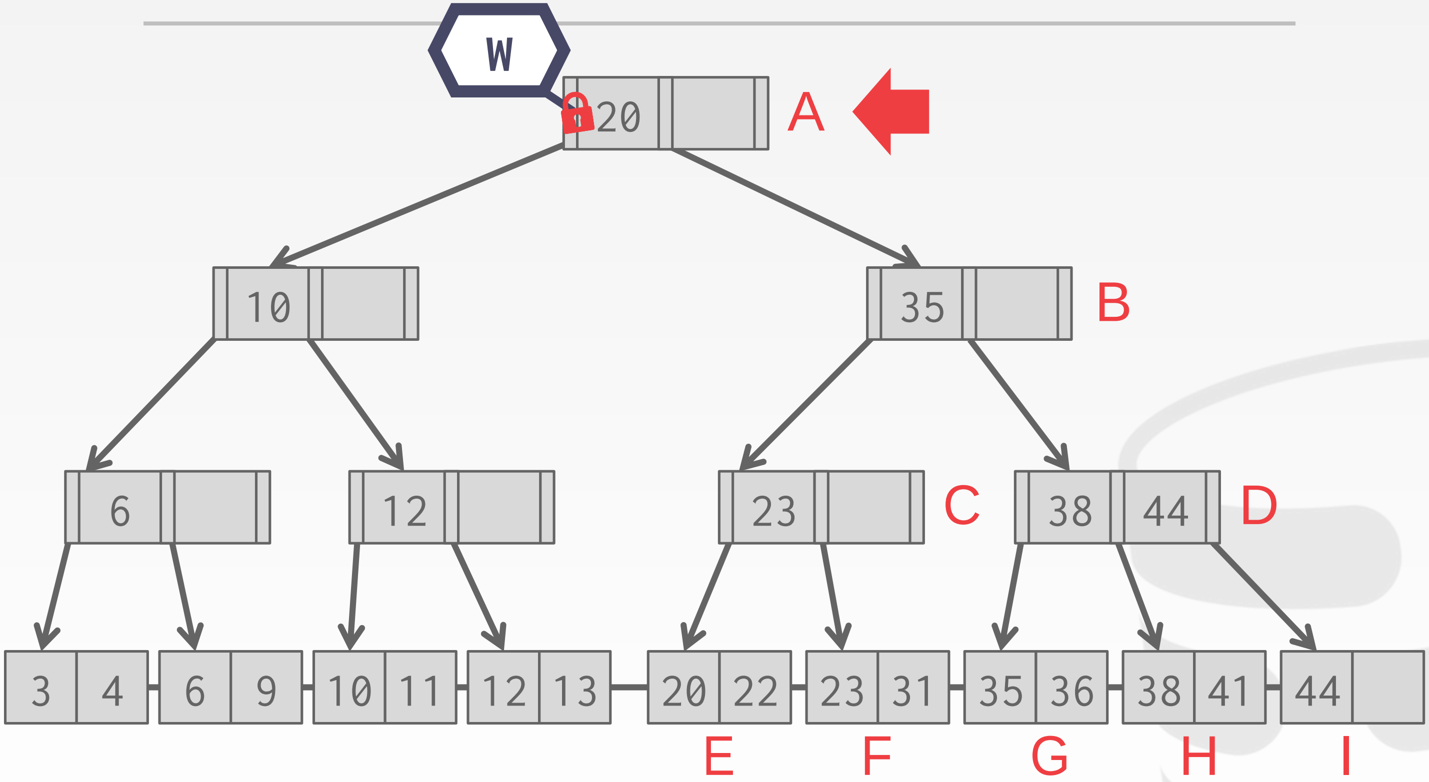
EXAMPLE #1 – SEARCH 38



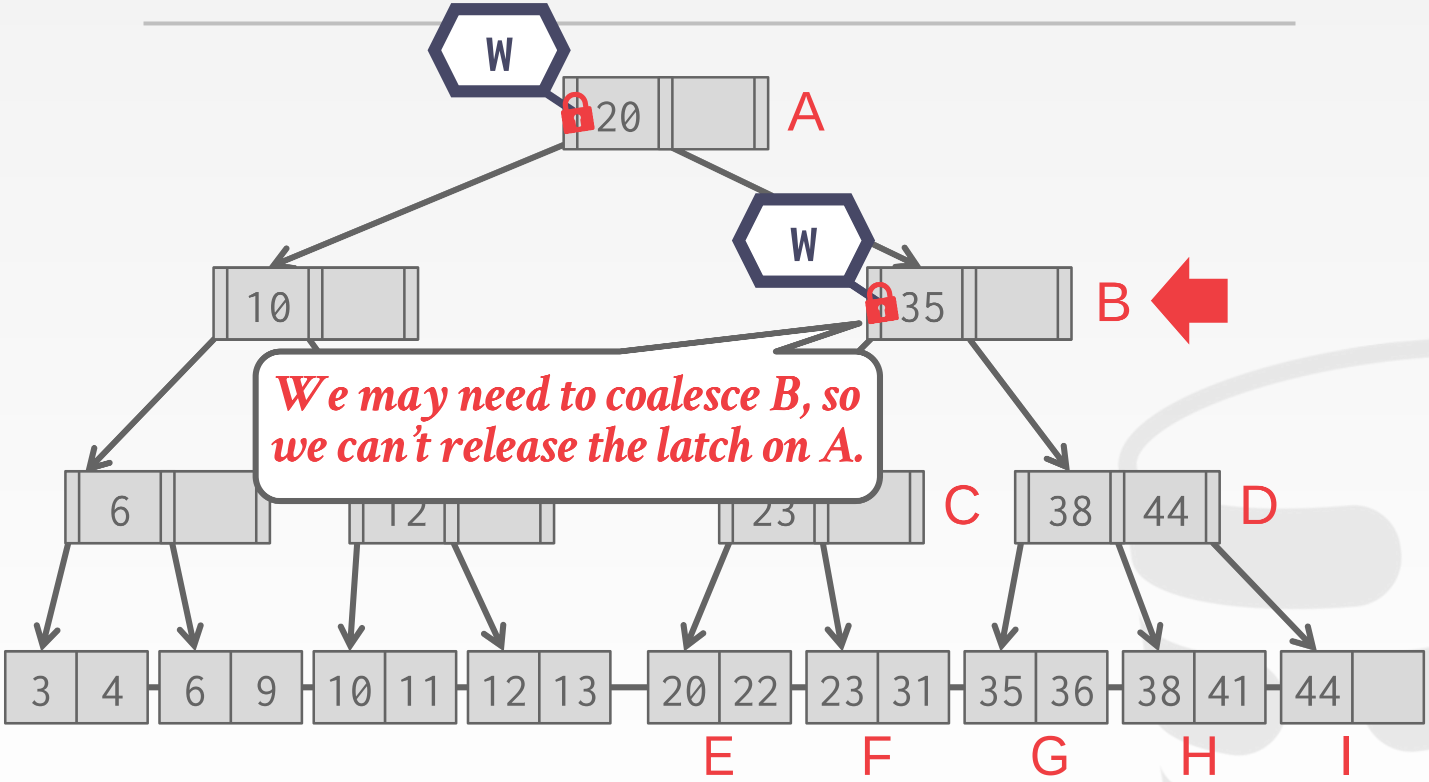
EXAMPLE #1 – SEARCH 38



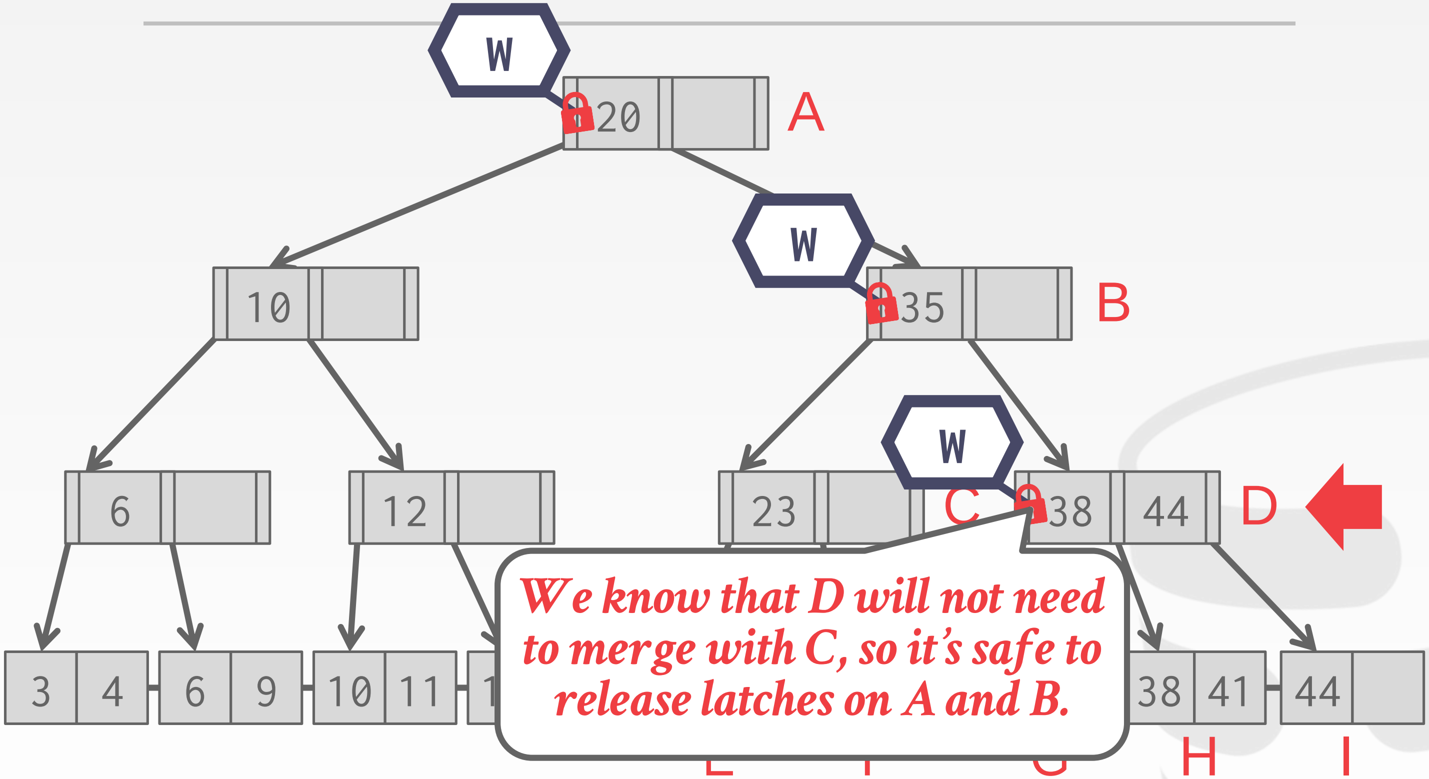
EXAMPLE #2 – DELETE 38



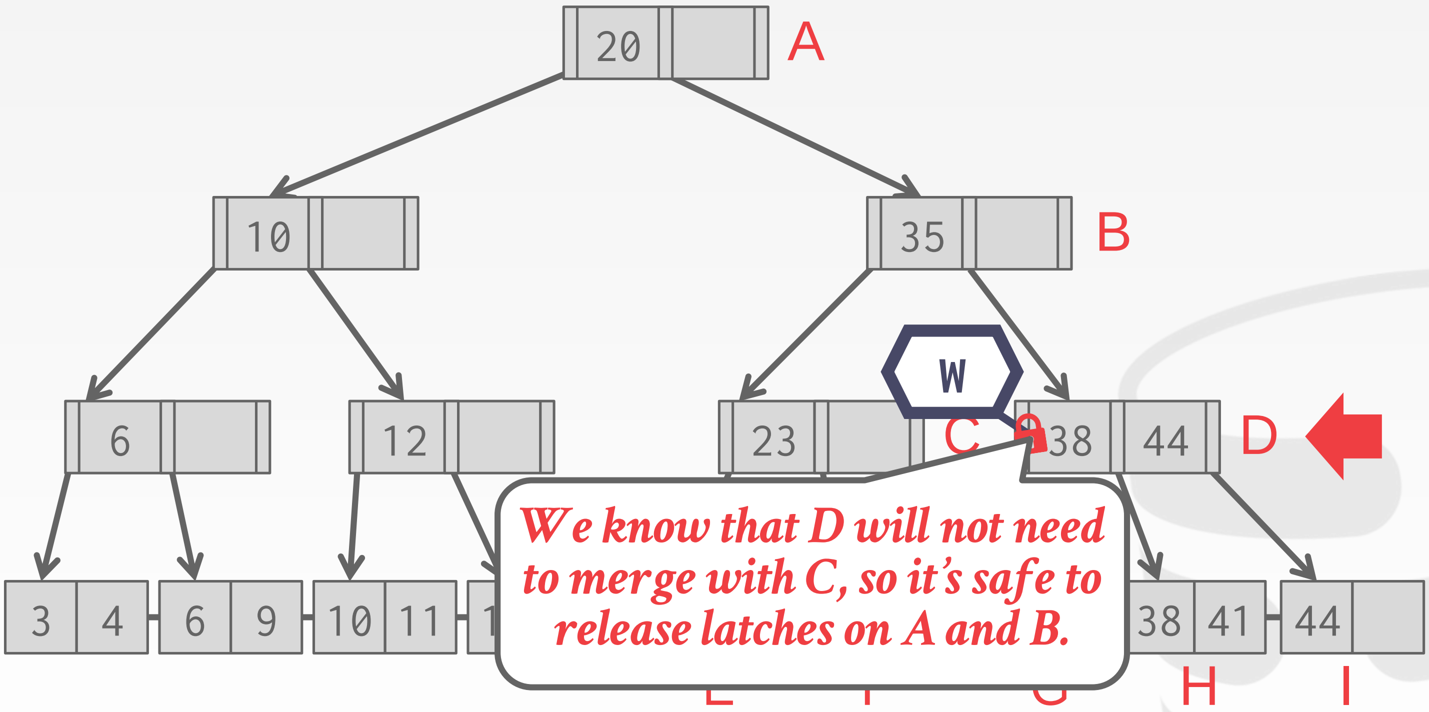
EXAMPLE #2 – DELETE 38



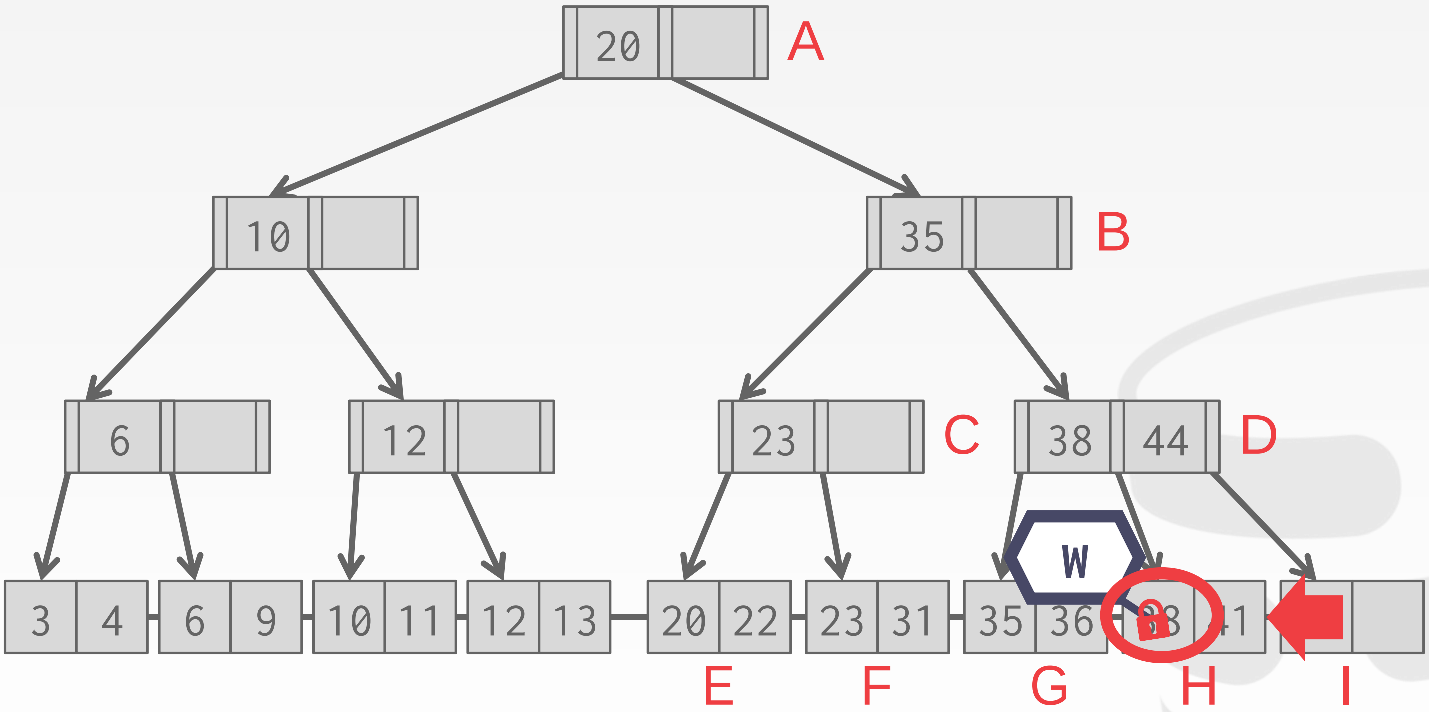
EXAMPLE #2 – DELETE 38



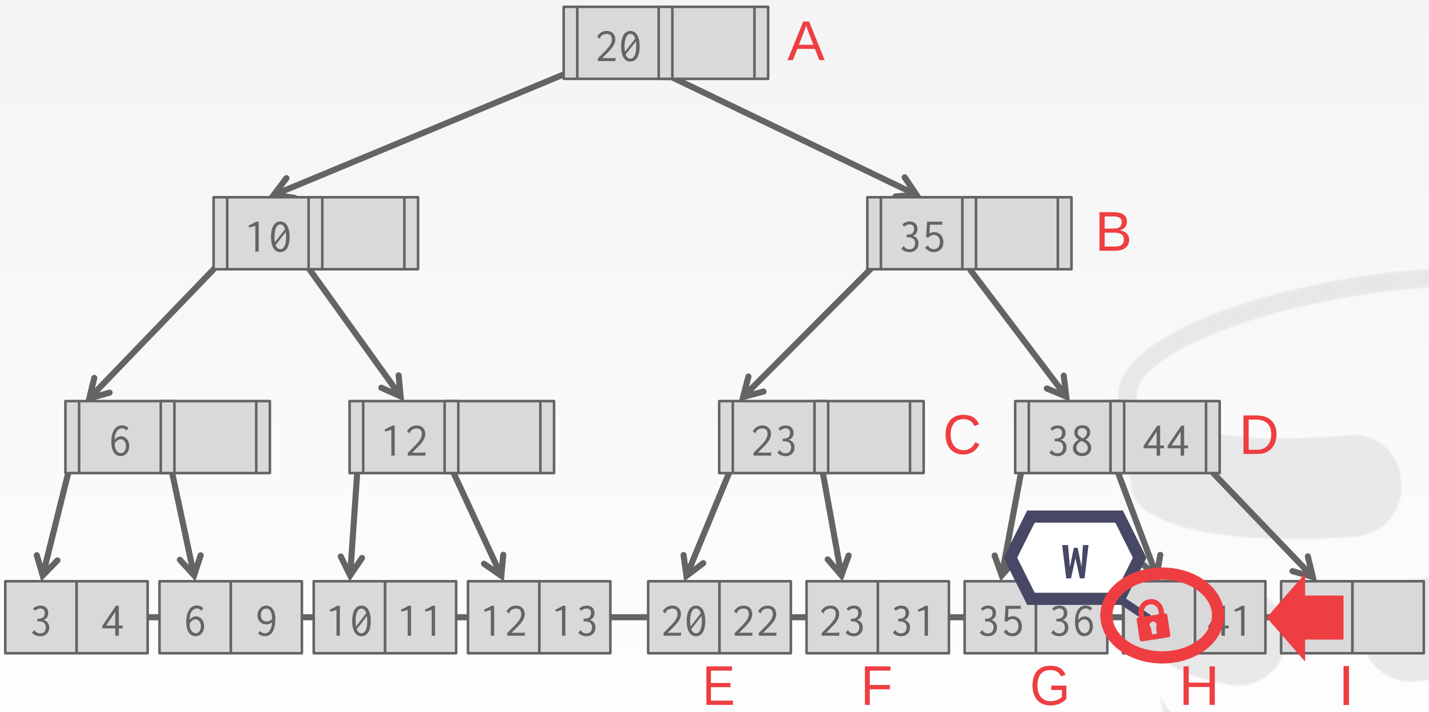
EXAMPLE #2 – DELETE 38



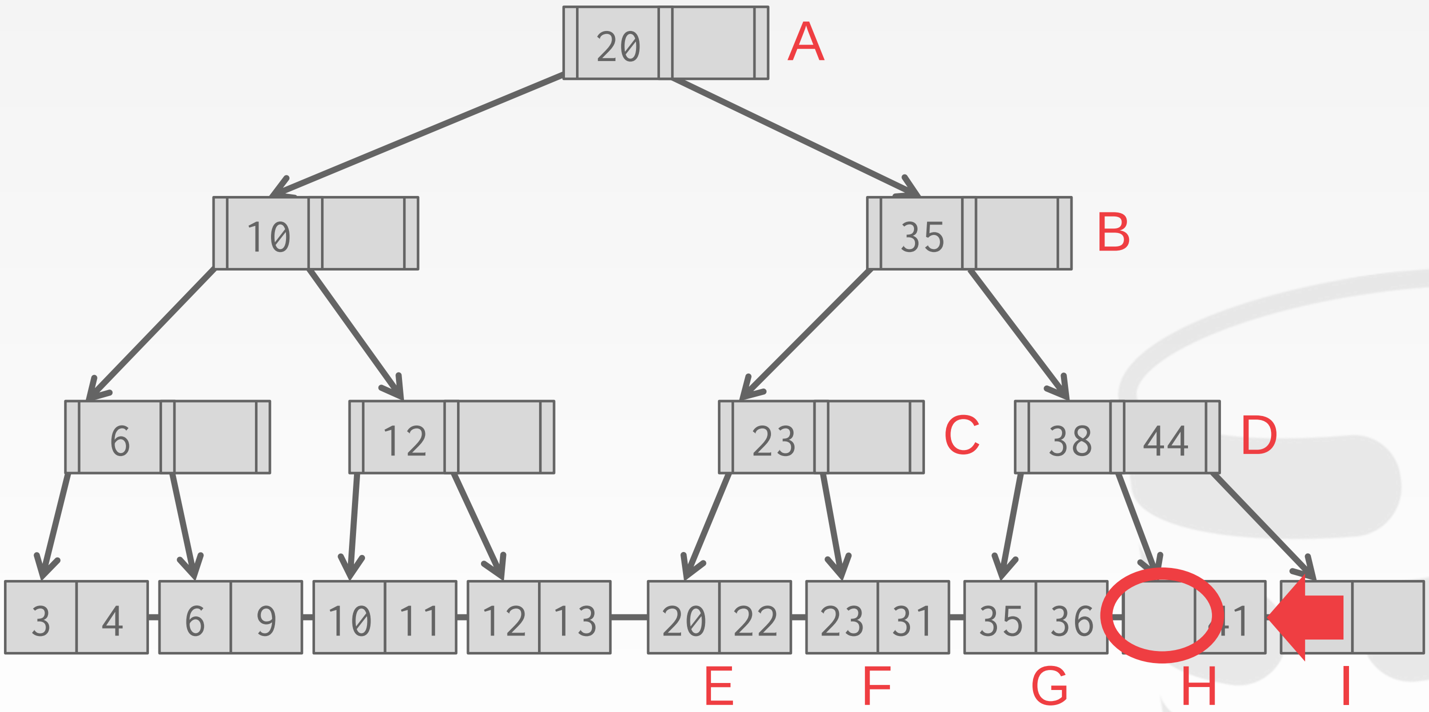
EXAMPLE #2 – DELETE 38



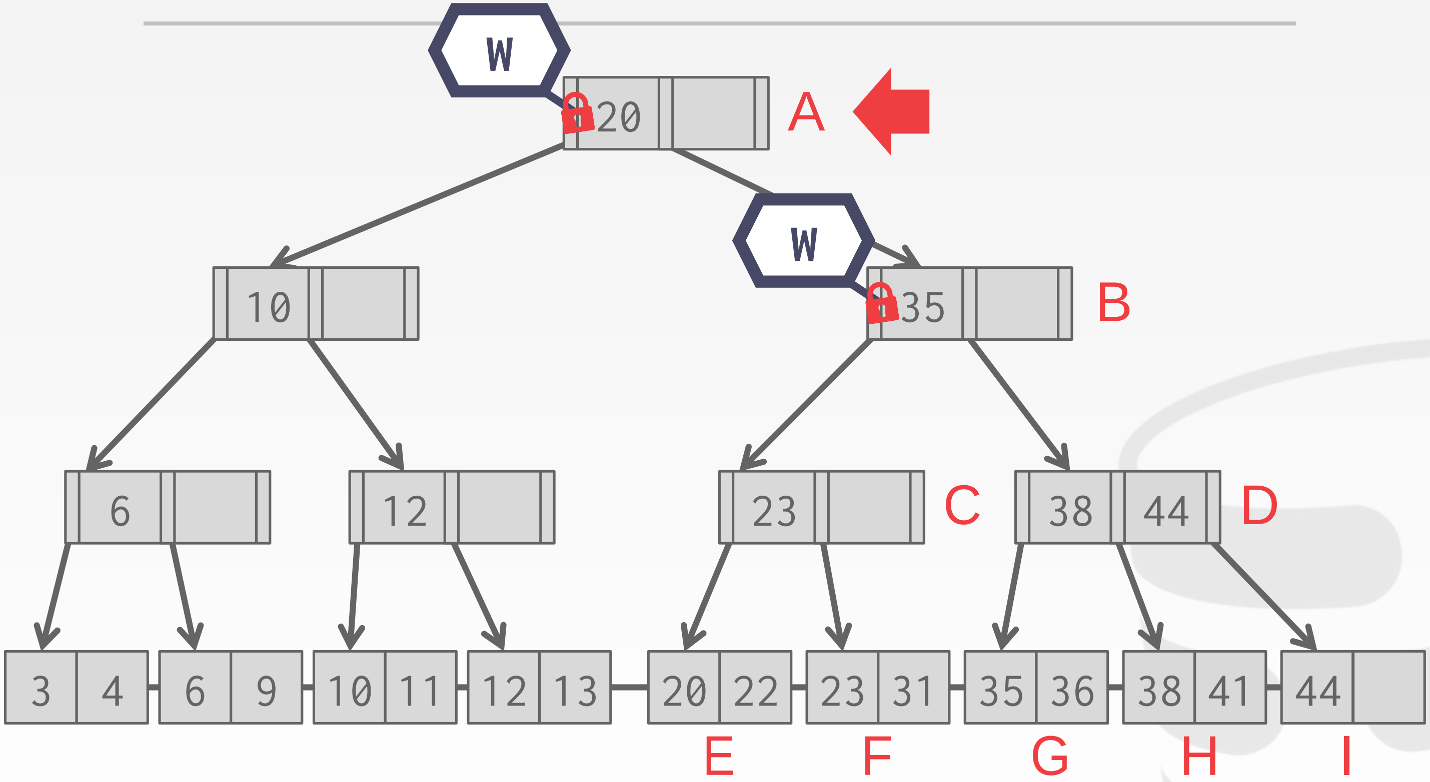
EXAMPLE #2 – DELETE 38



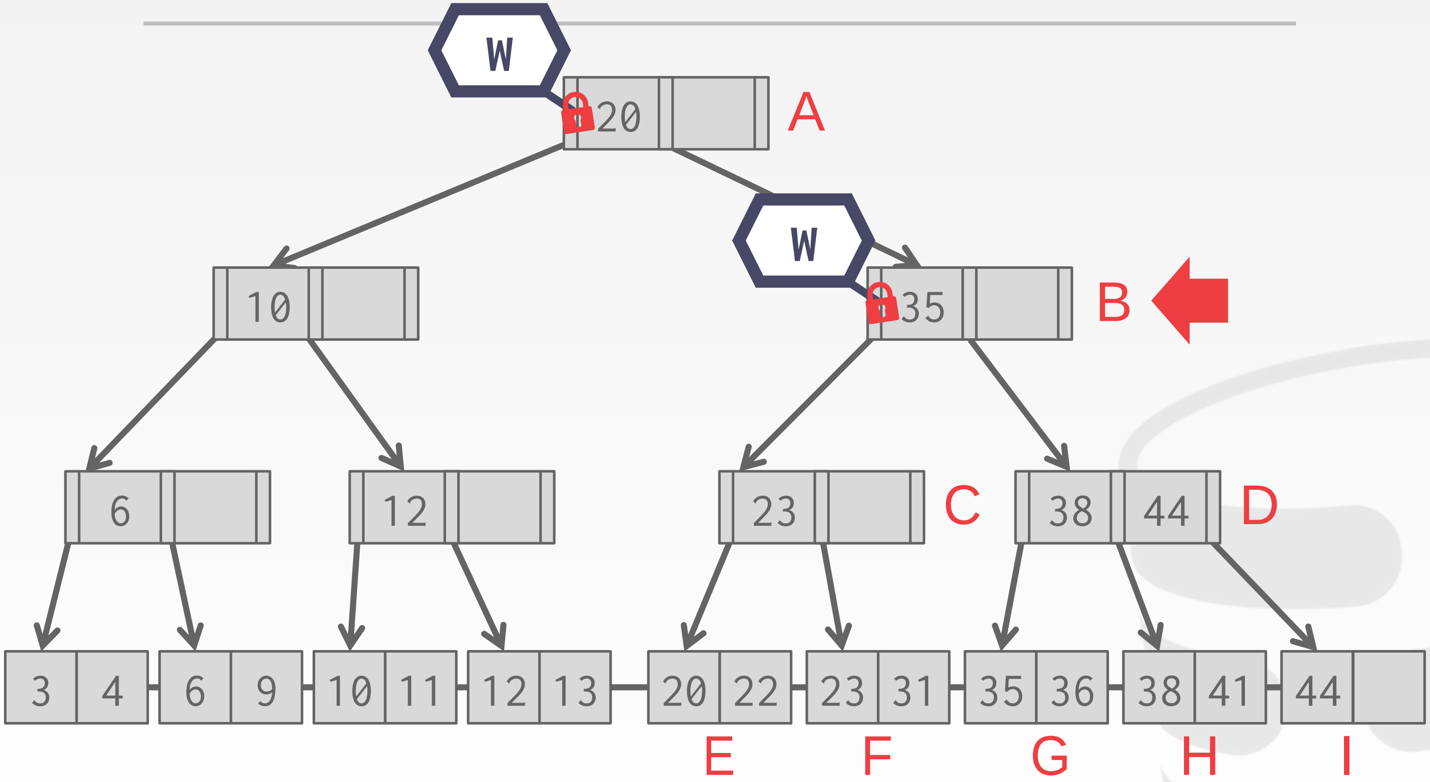
EXAMPLE #2 – DELETE 38



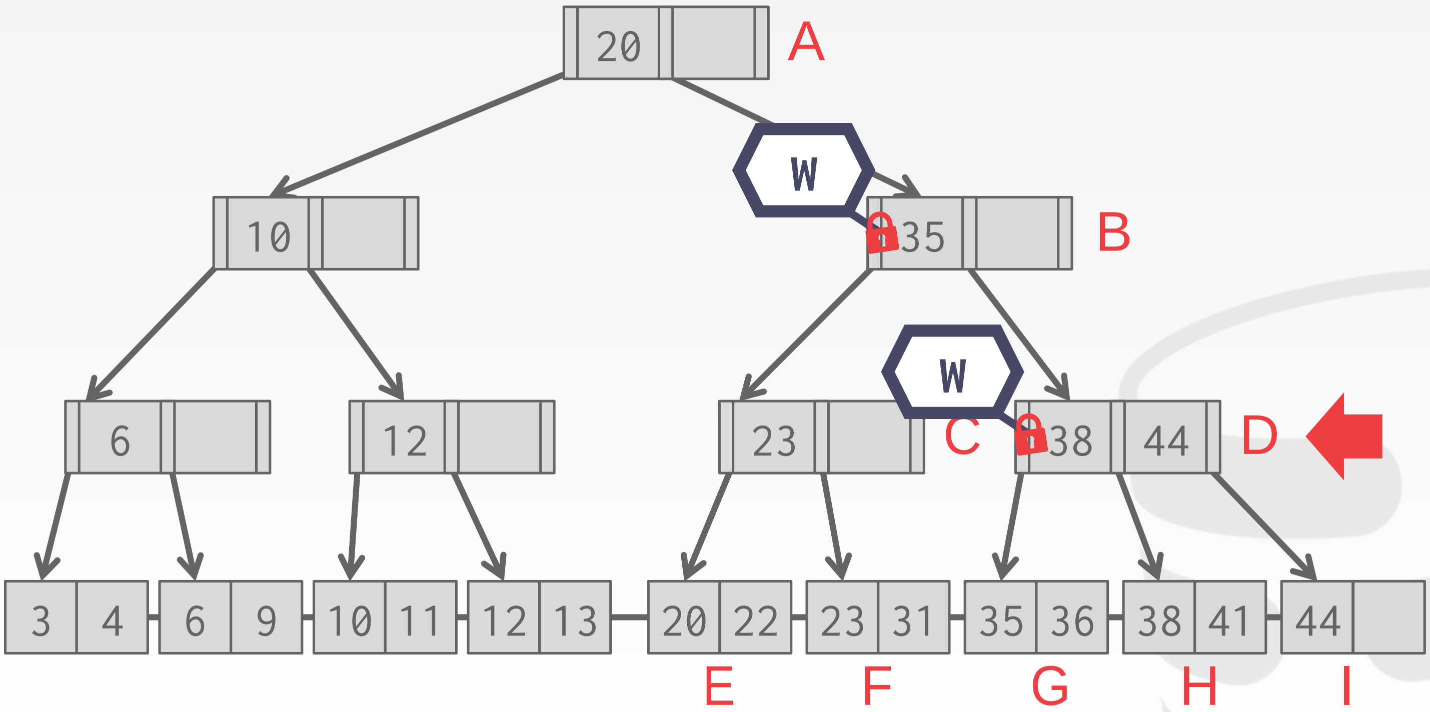
EXAMPLE #3 – INSERT 45



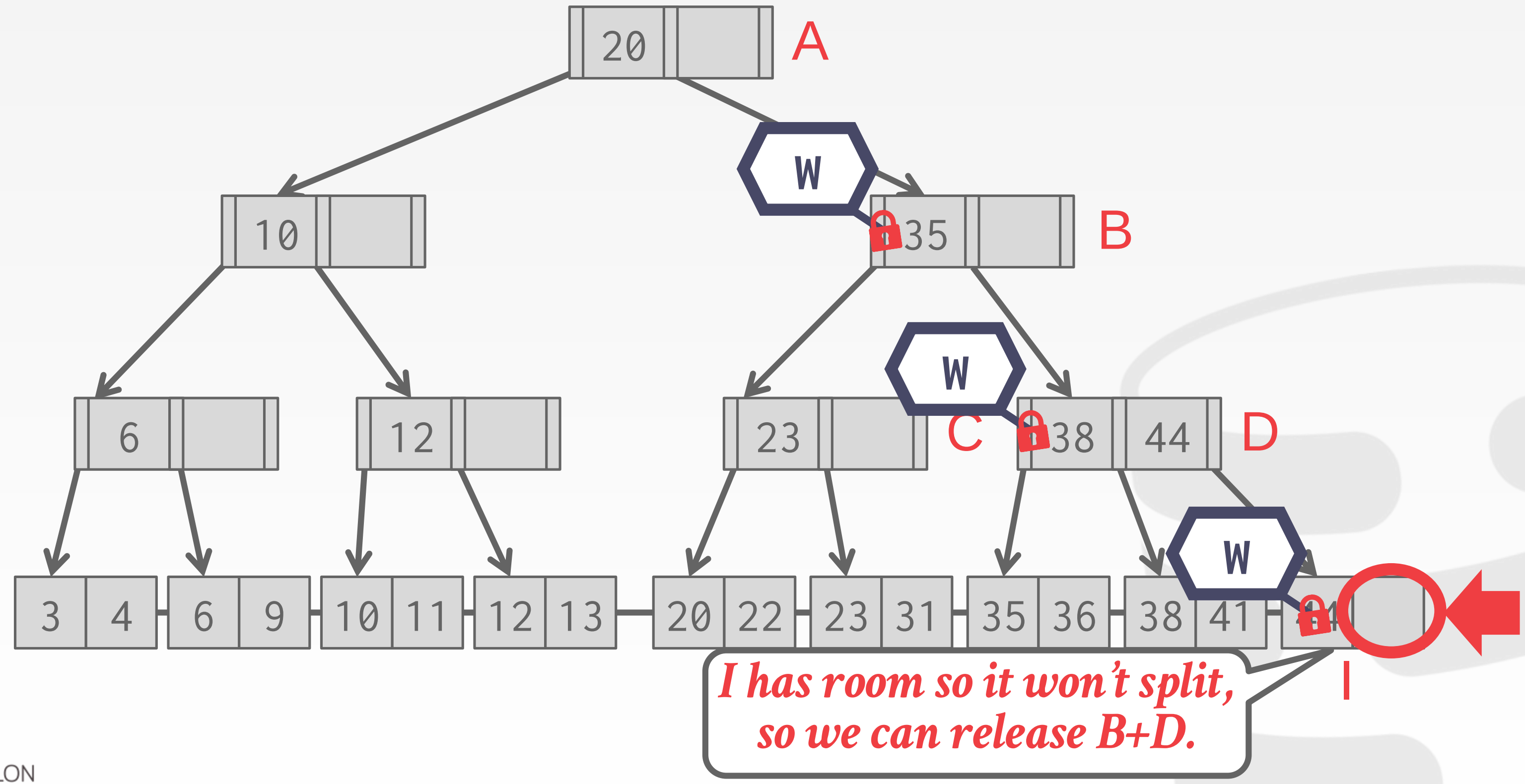
EXAMPLE #3 – INSERT 45



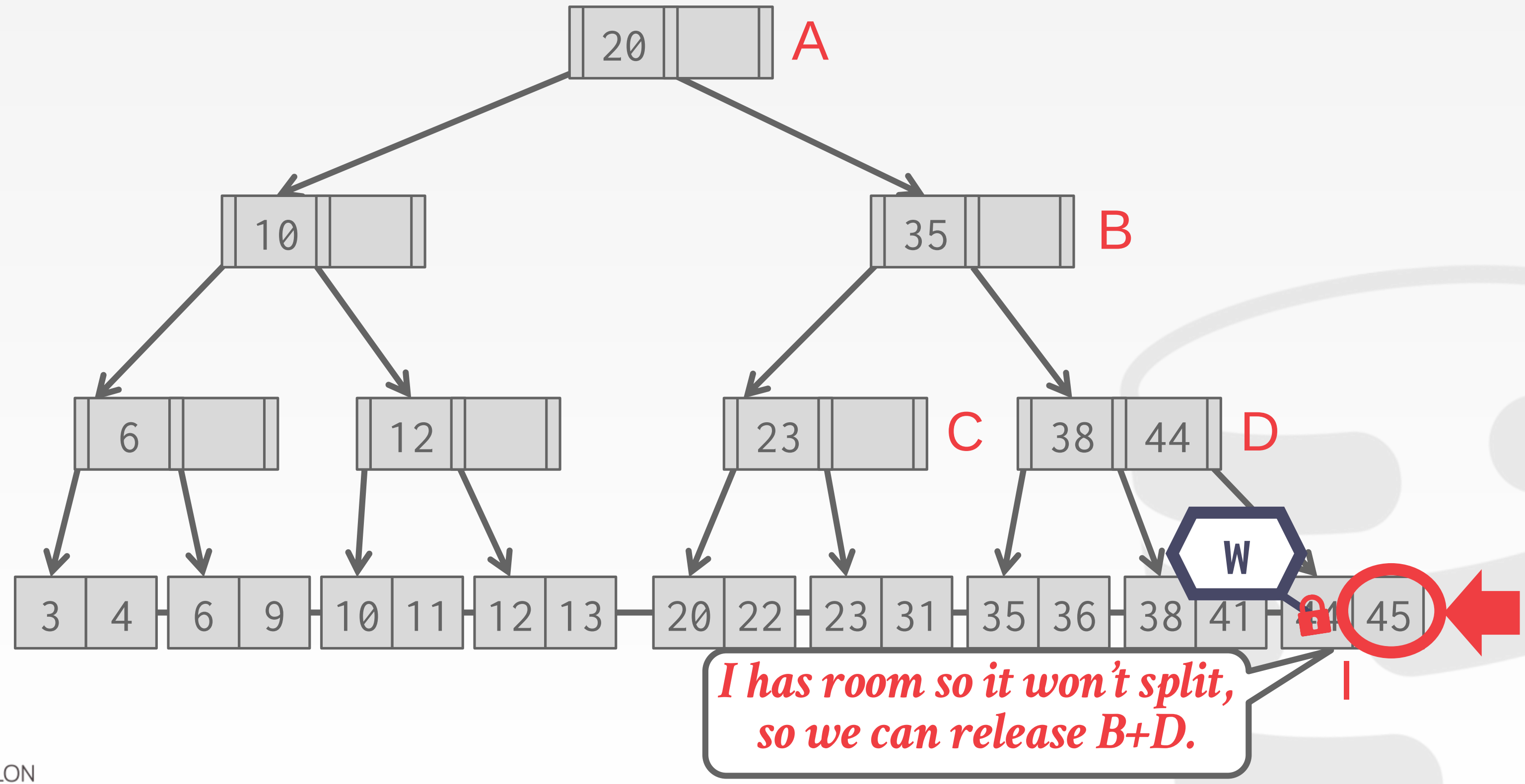
EXAMPLE #3 – INSERT 45



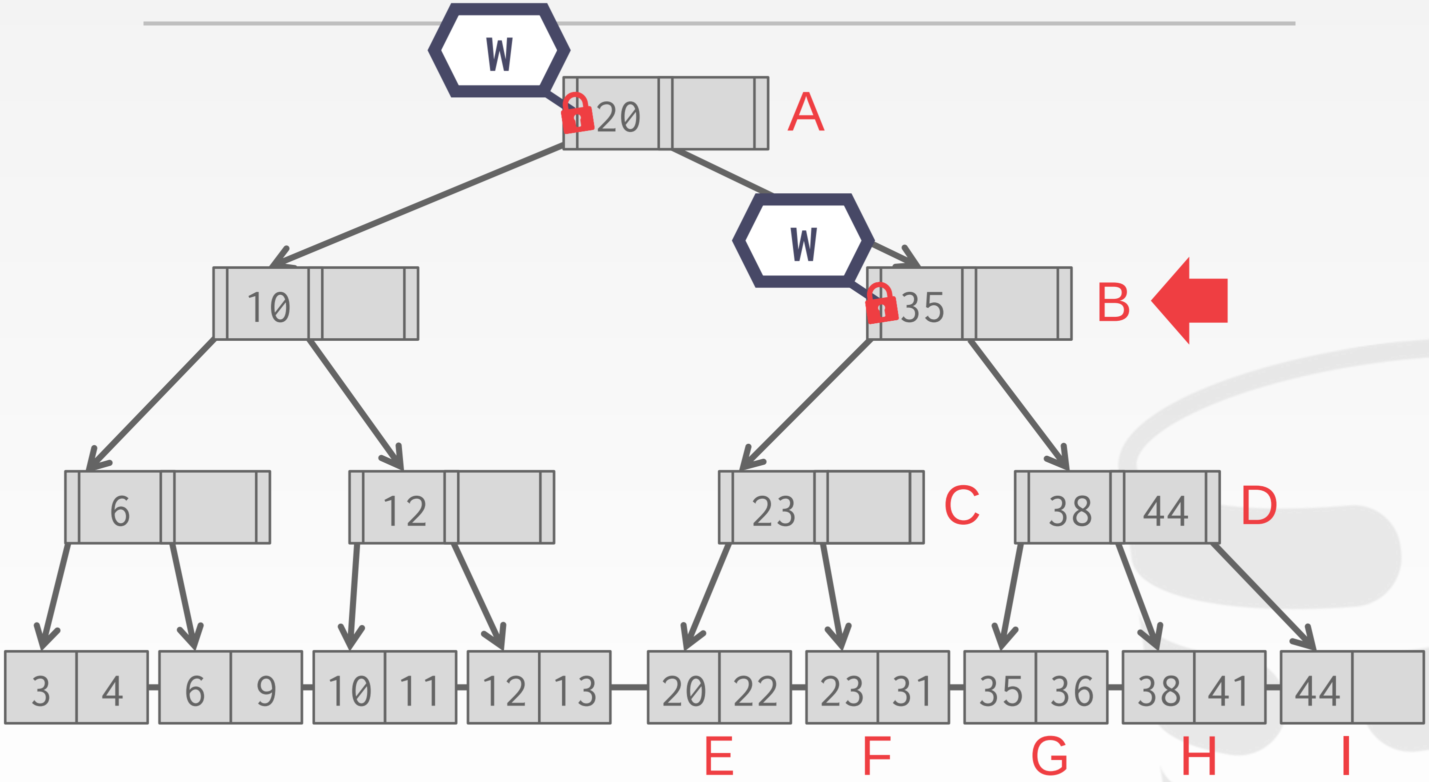
EXAMPLE #3 – INSERT 45



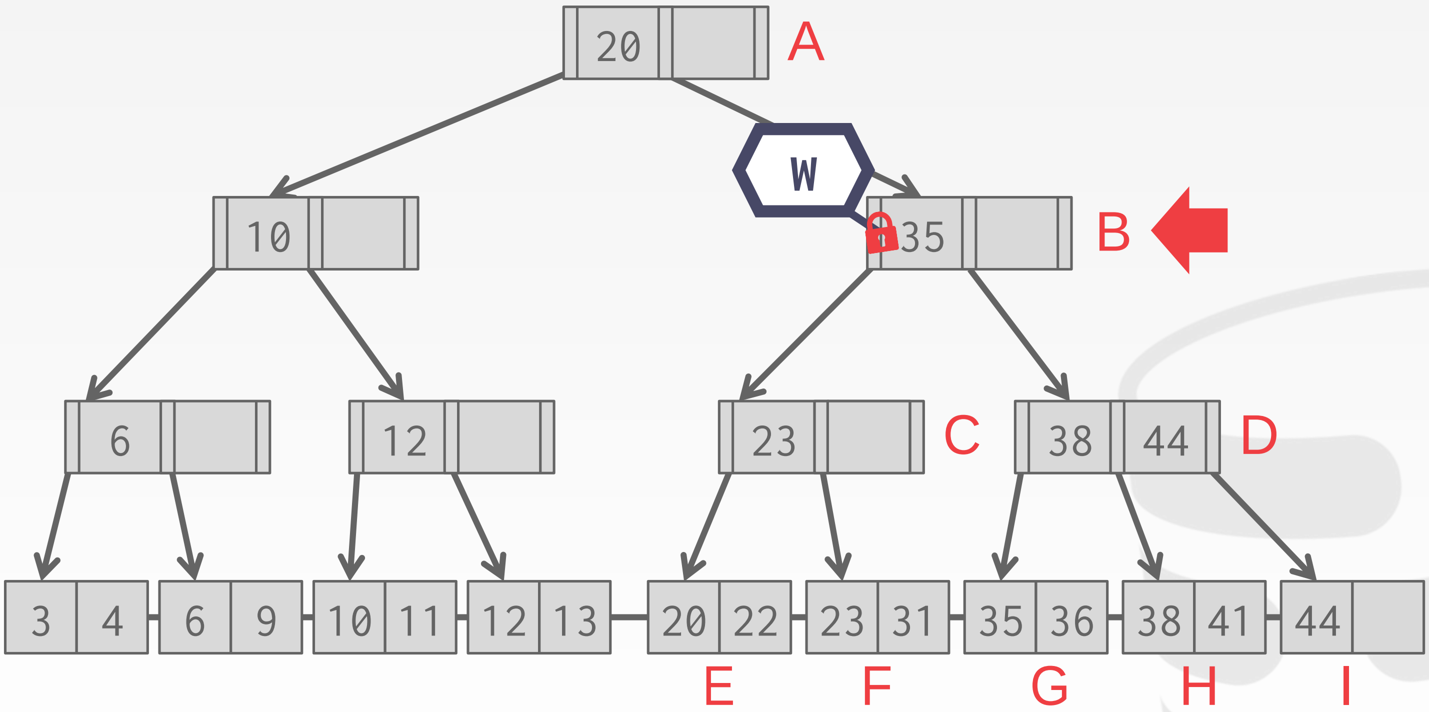
EXAMPLE #3 – INSERT 45



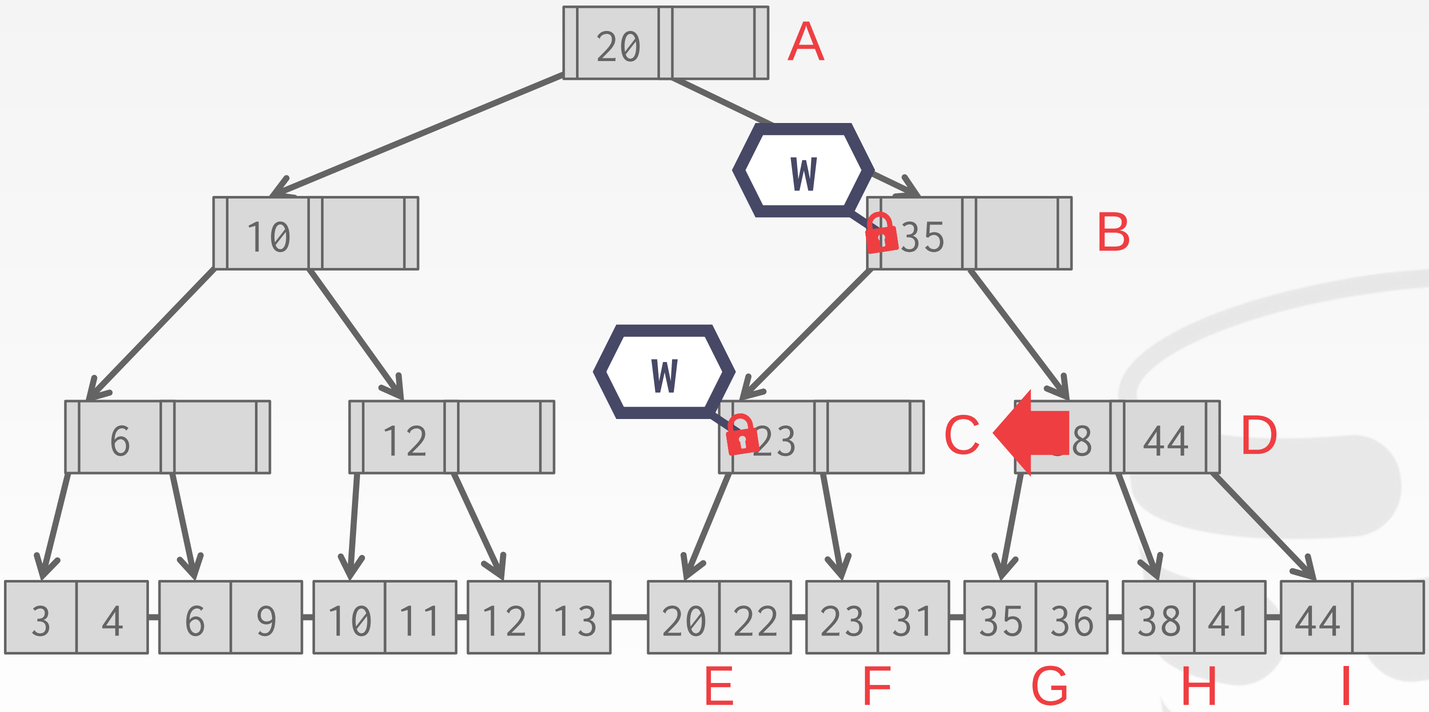
EXAMPLE #4 – INSERT 25



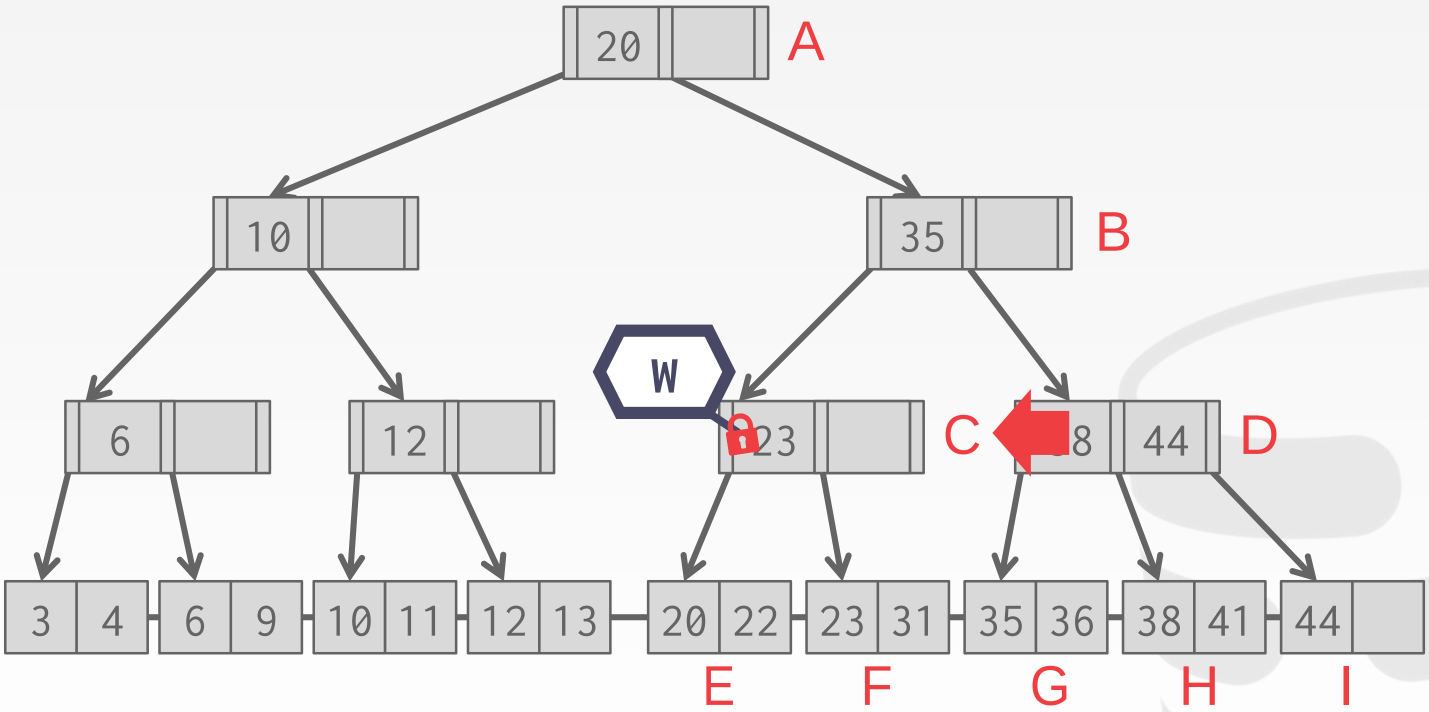
EXAMPLE #4 – INSERT 25



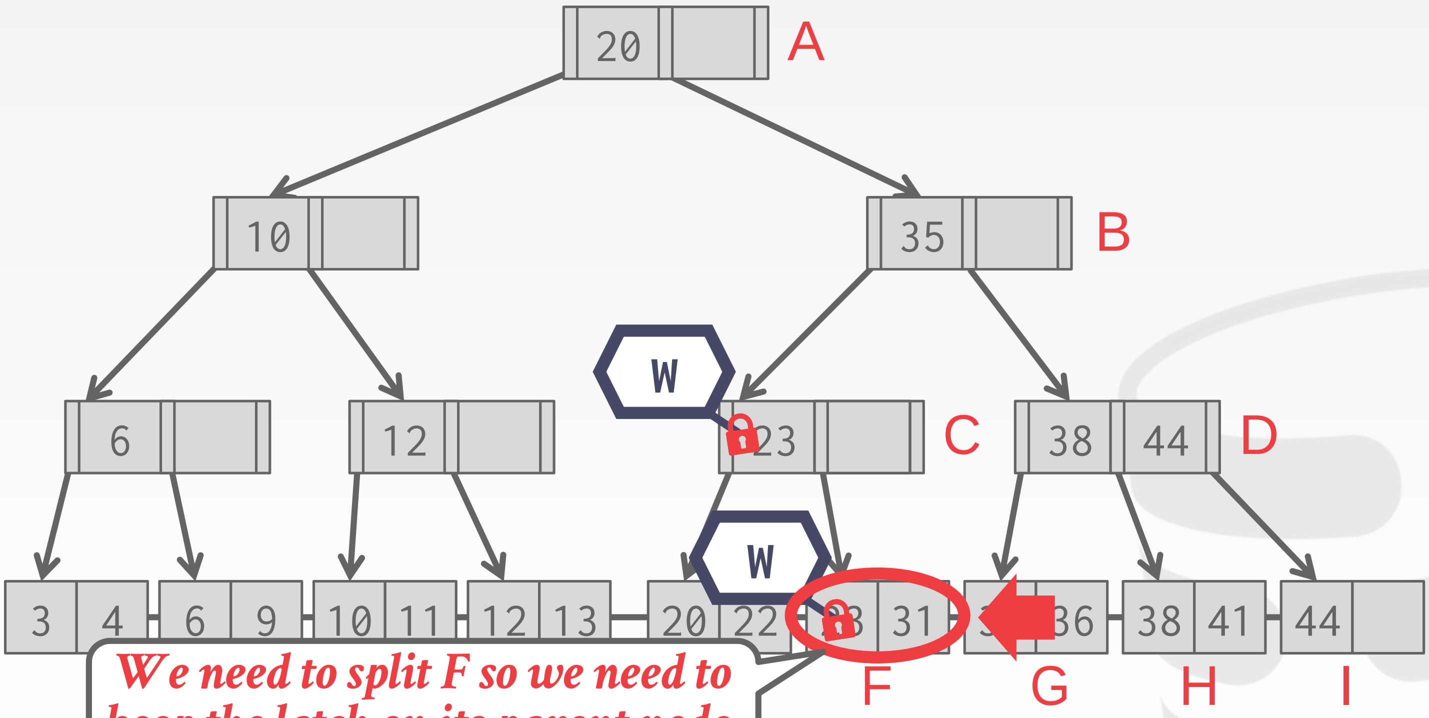
EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25

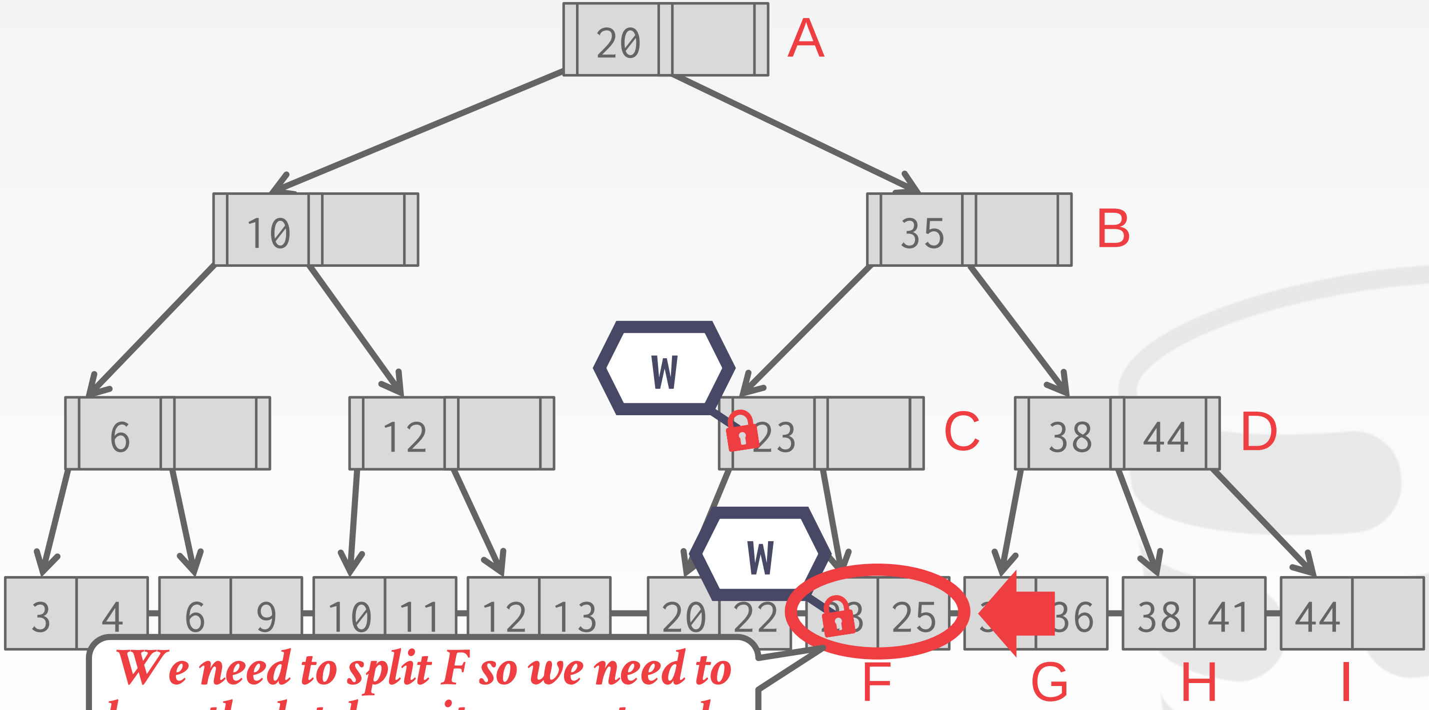


EXAMPLE #4 – INSERT 25



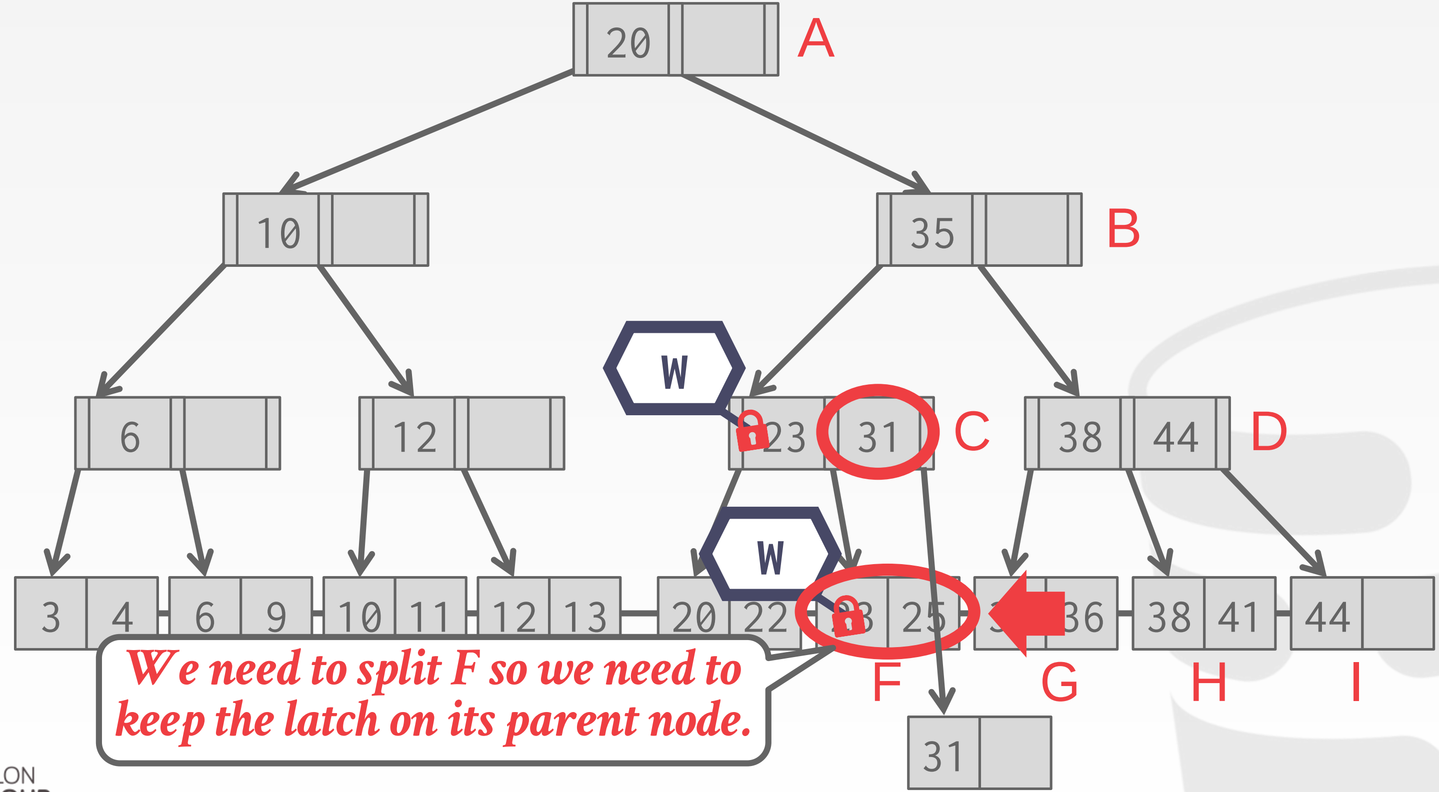
We need to split F so we need to keep the latch on its parent node.

EXAMPLE #4 – INSERT 25



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

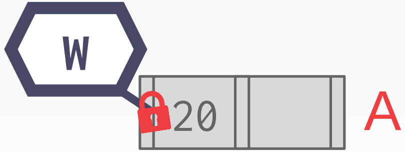
EXAMPLE #4 – INSERT 25



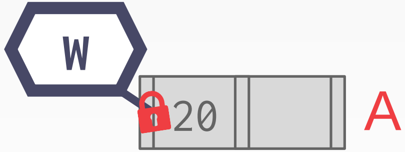
OBSERVATION

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

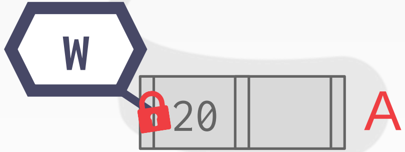
Delete 38



Insert 45



Insert 25



OBSERVATION

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

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

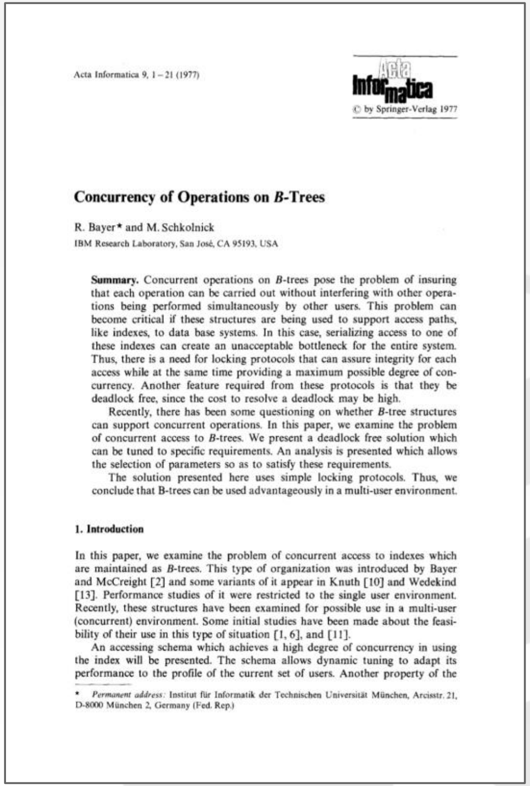
Can we do better?

BETTER LATCHING ALGORITHM

Assume that the leaf node is safe.

Use read latches and crabbing to reach it, and verify that it is safe.

If leaf is not safe, then do previous algorithm using write latches.



Acta Informatica 9, 1 – 21 (1977)

Acta Informatica

© by Springer-Verlag 1977

Concurrency of Operations on *B*-Trees

R. Bayer* and M. Schkolnick

IBM Research Laboratory, San José, CA 95193, USA

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

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

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

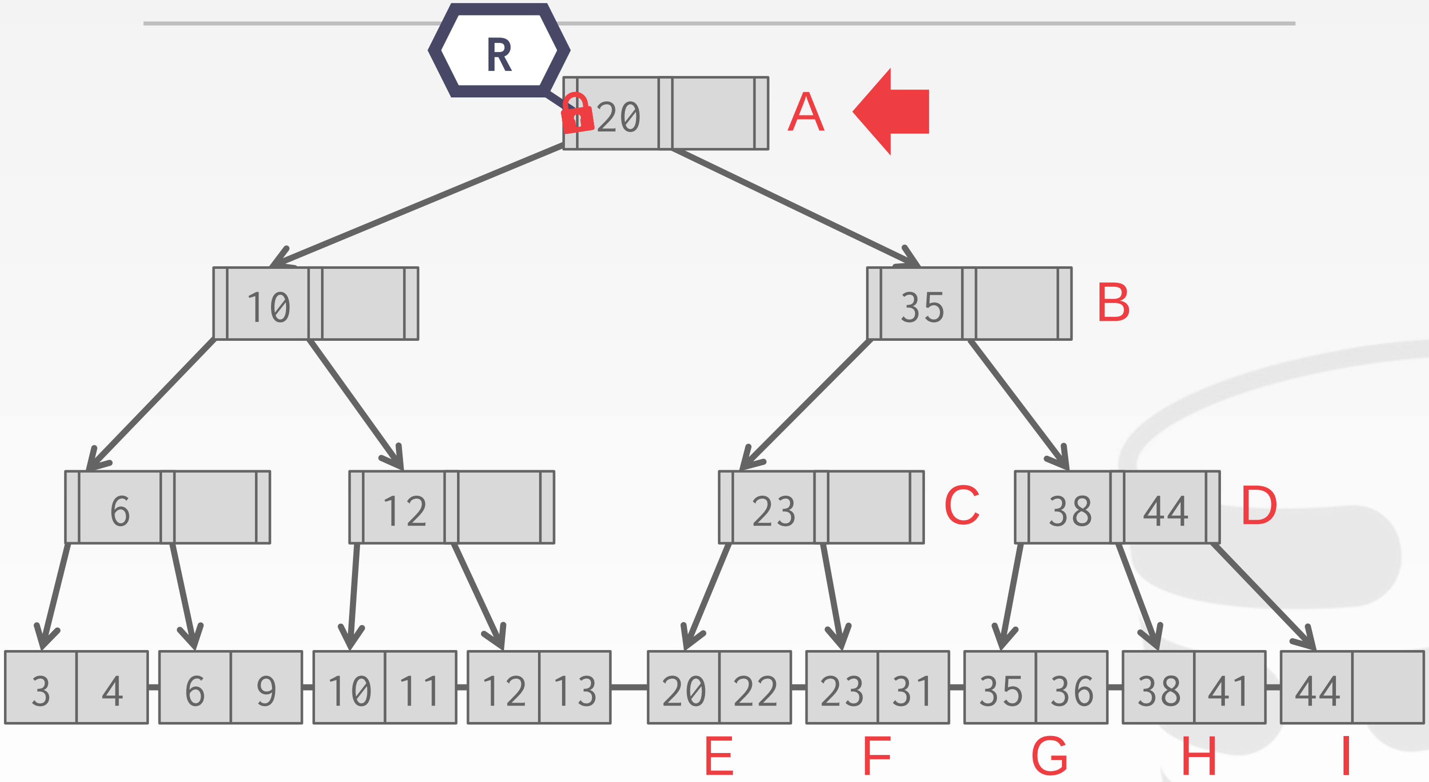
1. Introduction

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

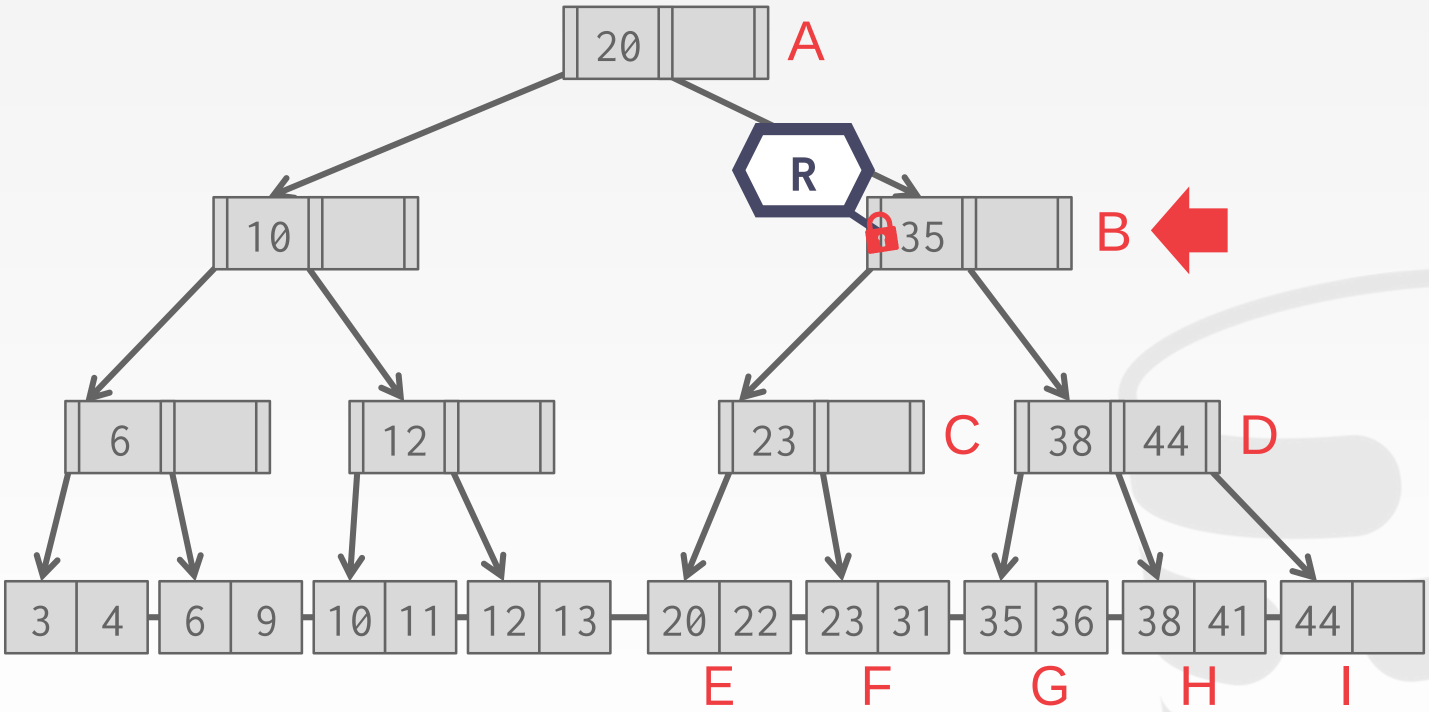
An accessing schema which achieves a high degree of concurrency in using the index will be presented. The schema allows dynamic tuning to adapt its performance to the profile of the current set of users. Another property of the

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

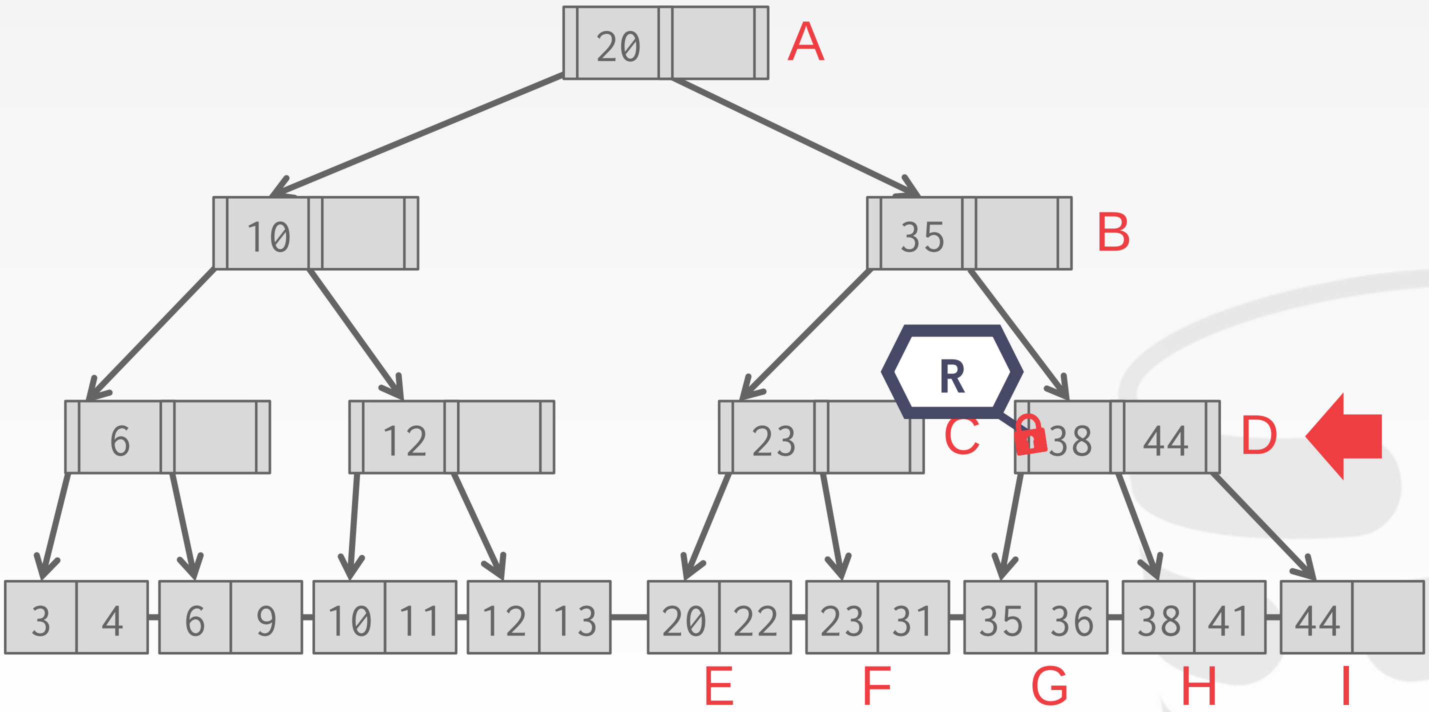
EXAMPLE #2 – DELETE 38



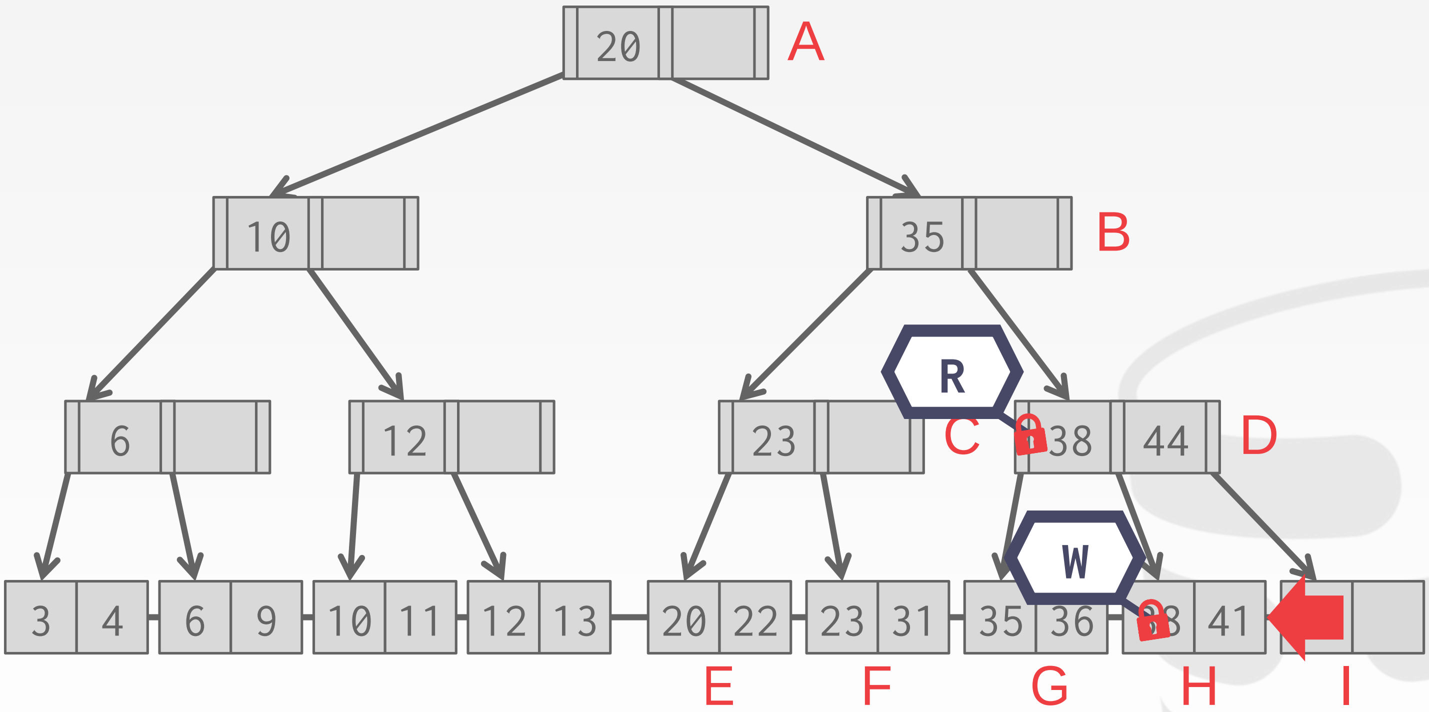
EXAMPLE #2 – DELETE 38



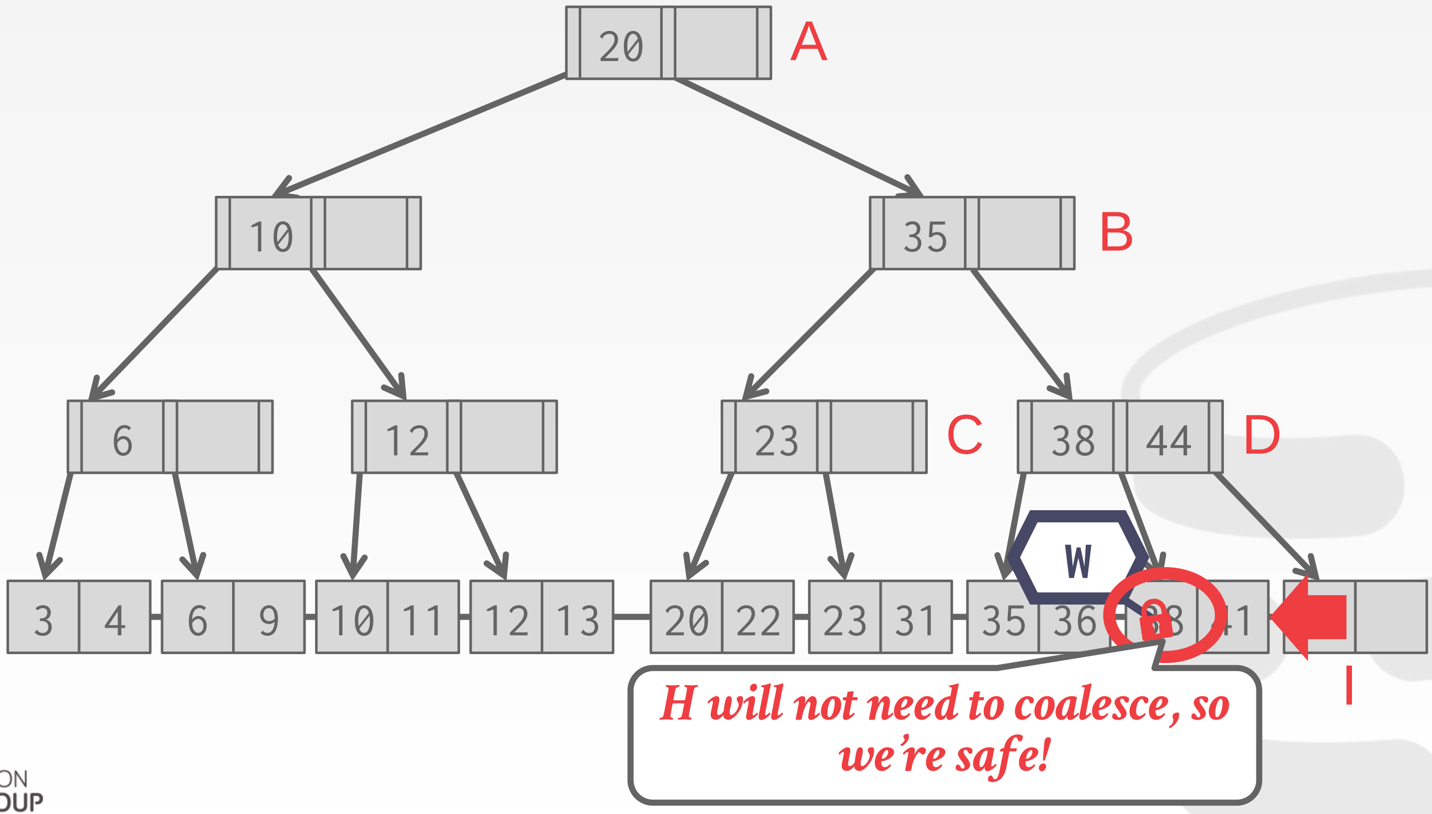
EXAMPLE #2 – DELETE 38



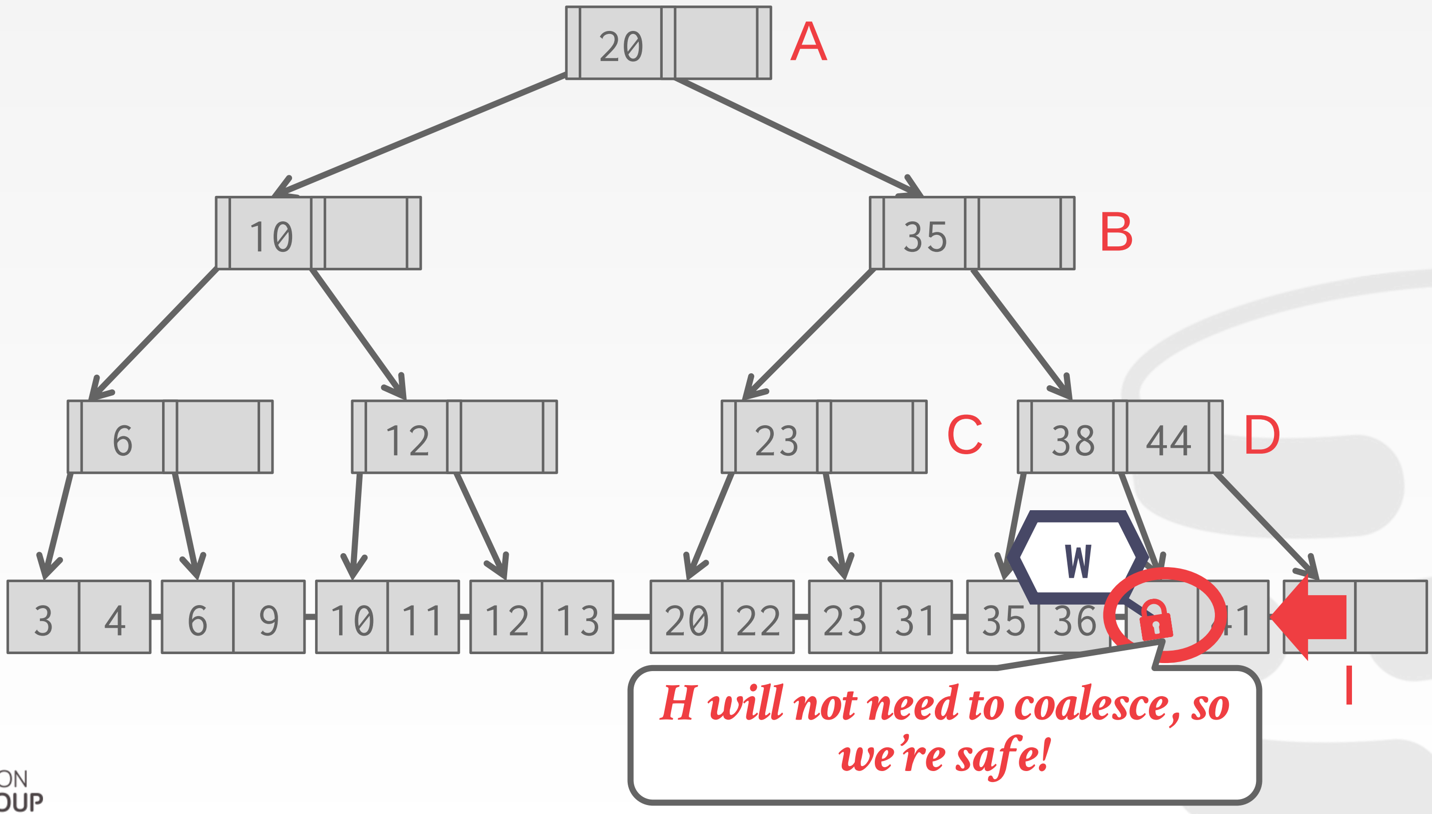
EXAMPLE #2 – DELETE 38



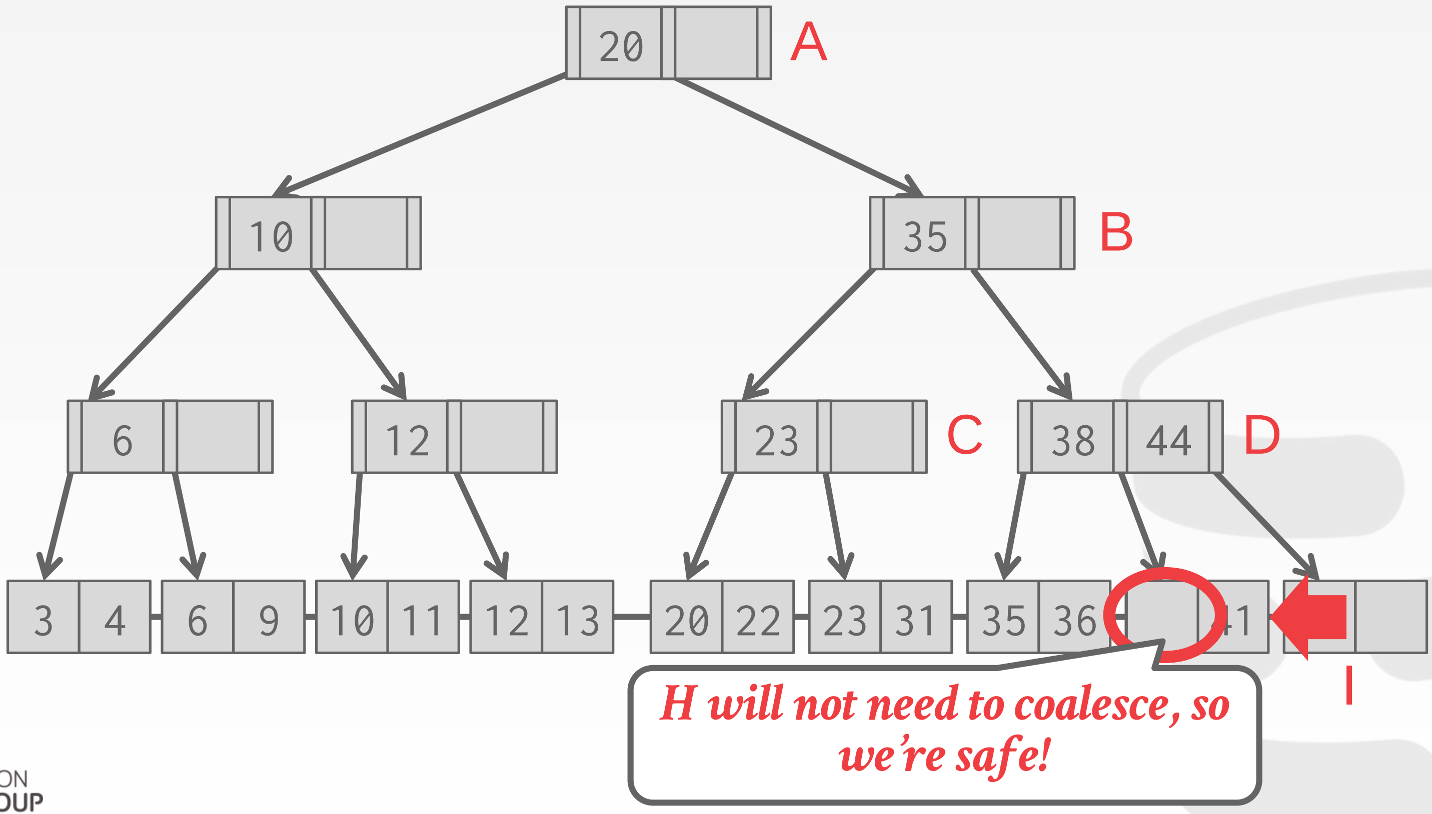
EXAMPLE #2 – DELETE 38



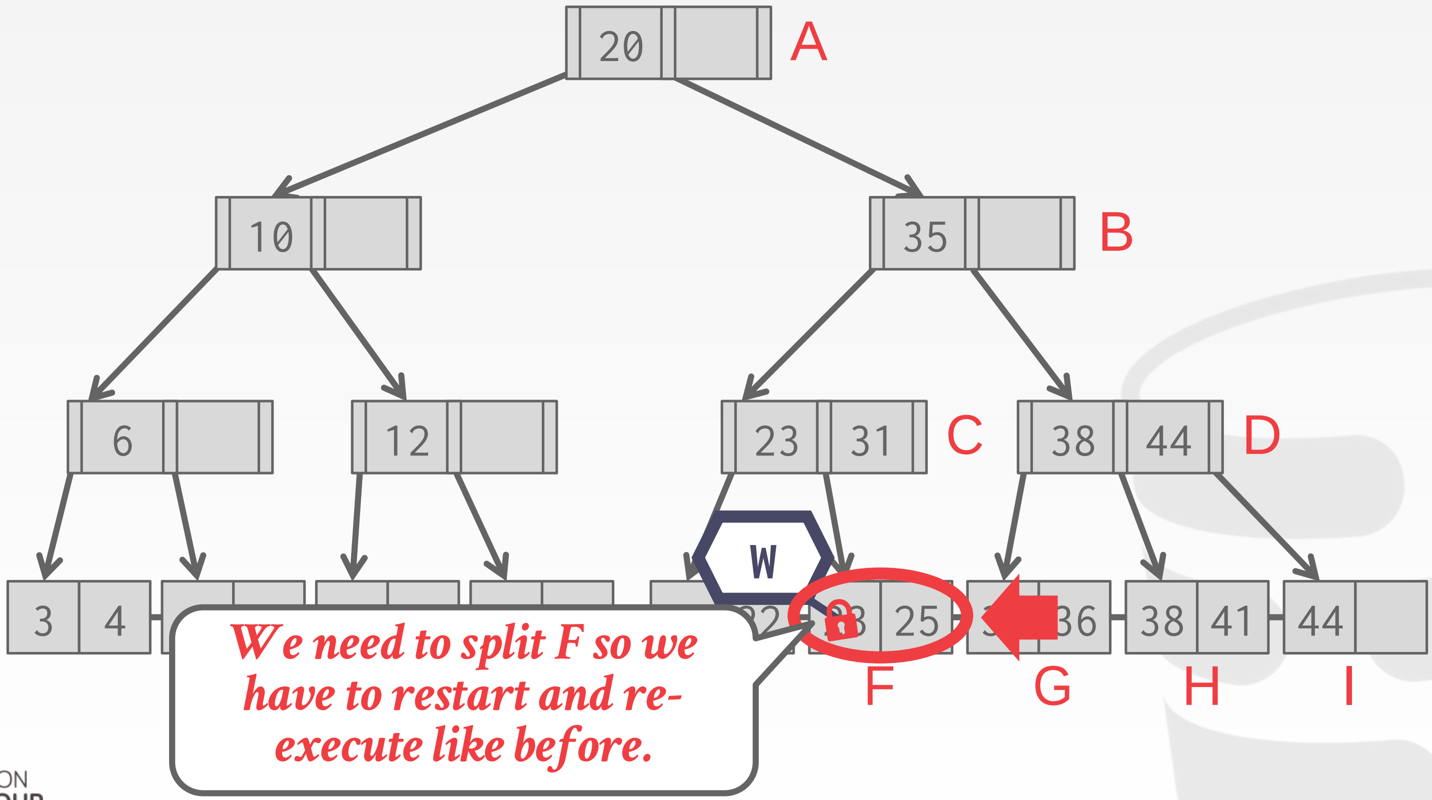
EXAMPLE #2 – DELETE 38



EXAMPLE #2 – DELETE 38



EXAMPLE #4 – INSERT 25



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.

OBSERVATION

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

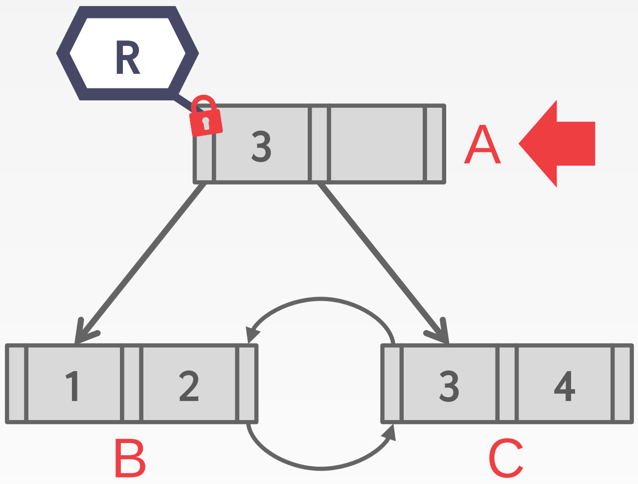
→ A thread can only acquire a latch from a node that is below its current node.

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

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

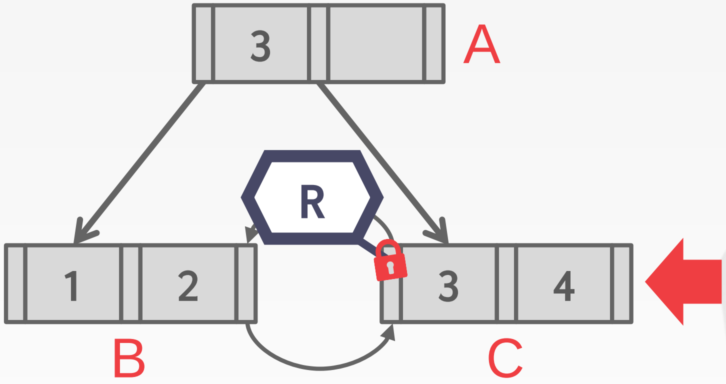
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



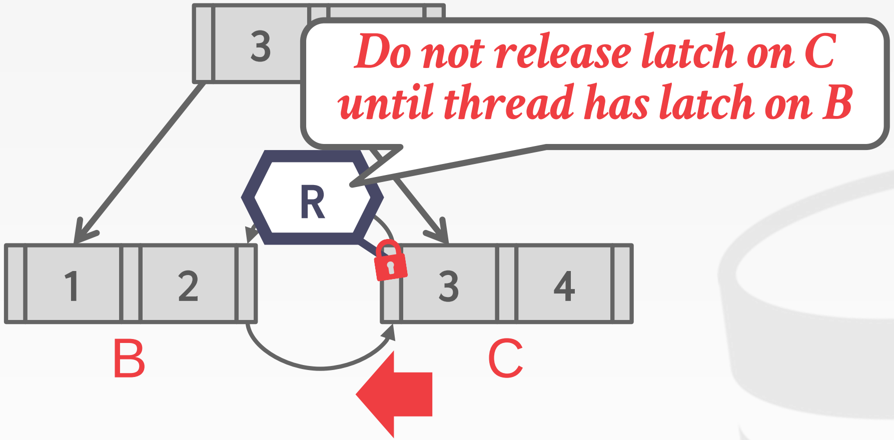
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



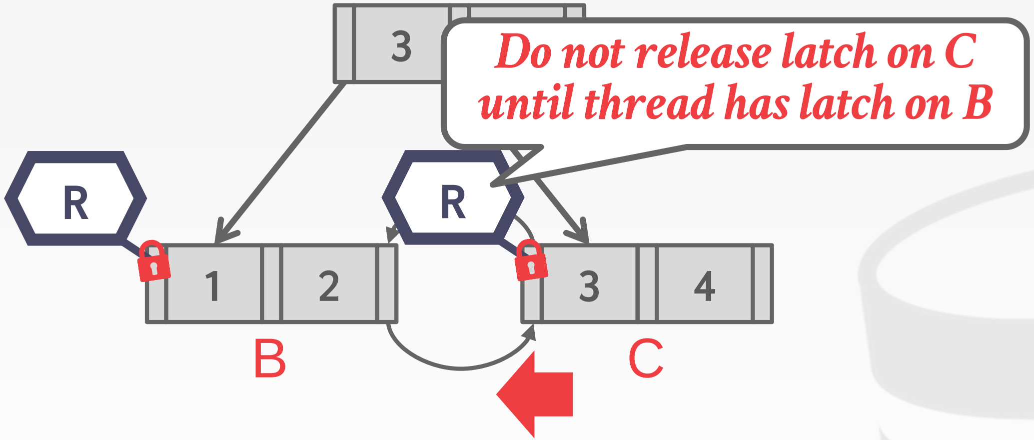
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



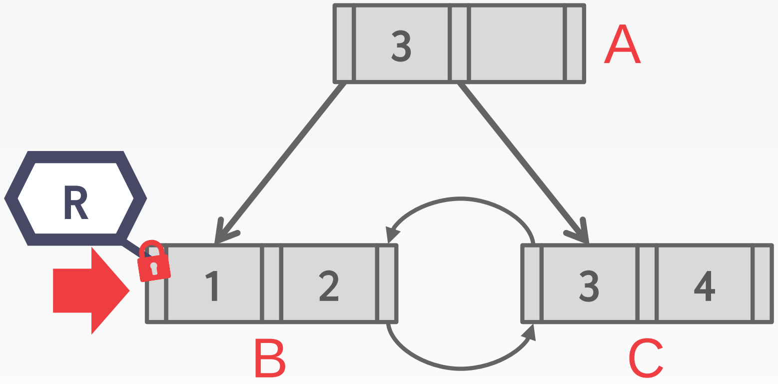
LEAF NODE SCAN EXAMPLE #1

T_1: Find Keys < 4



LEAF NODE SCAN EXAMPLE #1

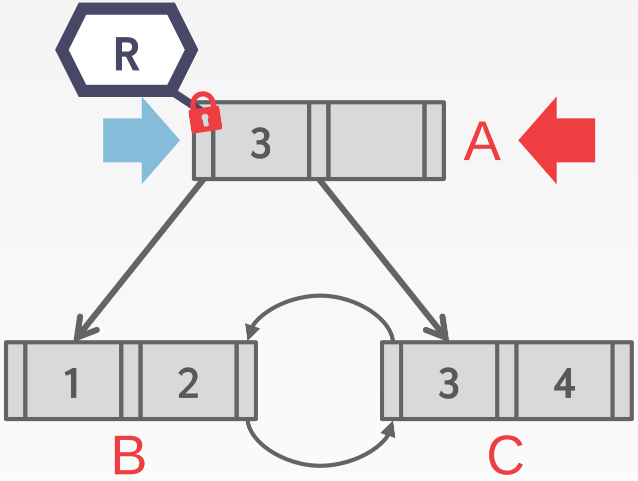
T_1: Find Keys < 4



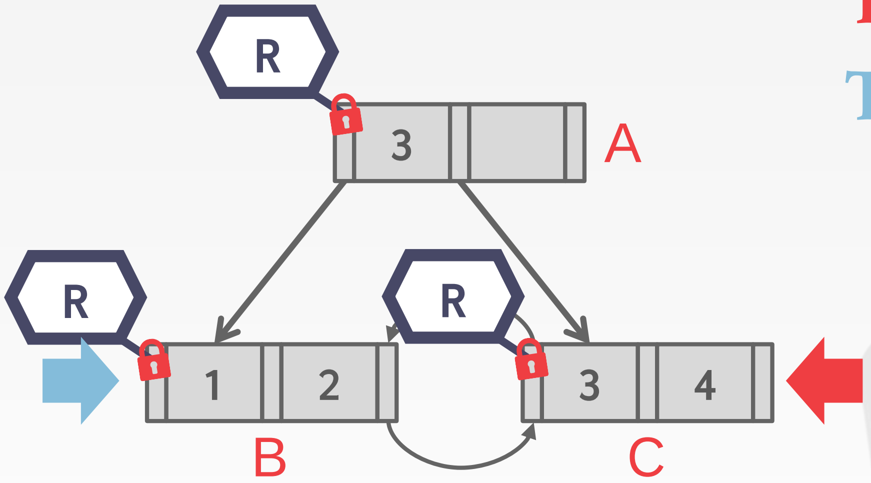
LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #2



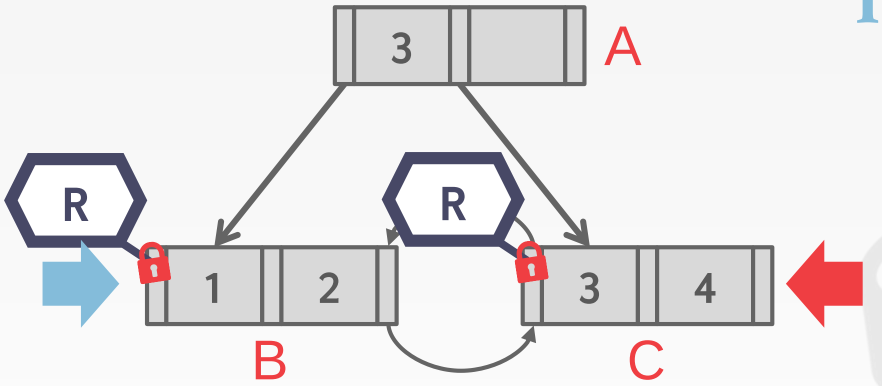
T_1: Find Keys < 4

T_2: Find Keys > 1

LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

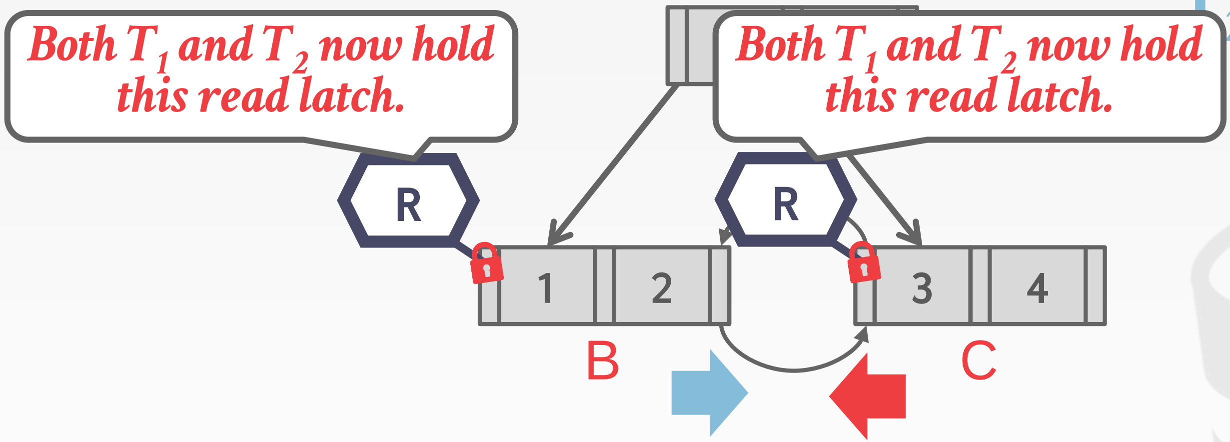
T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

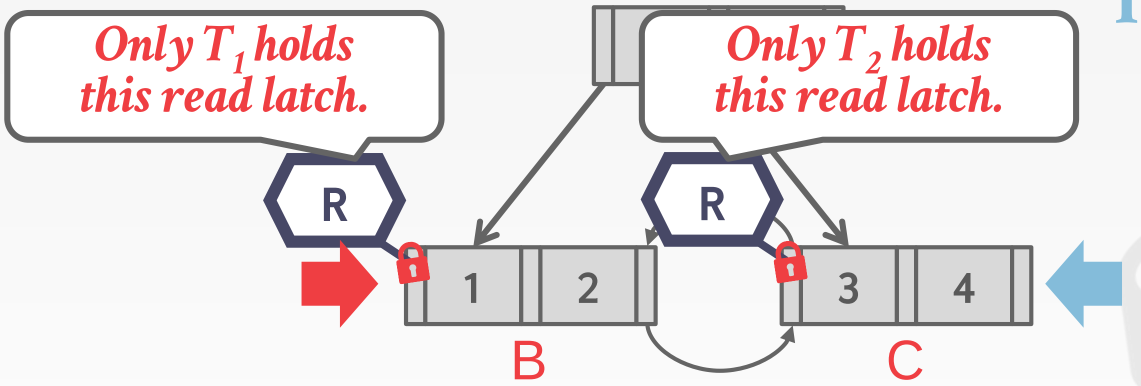
T_2: Find Keys > 1



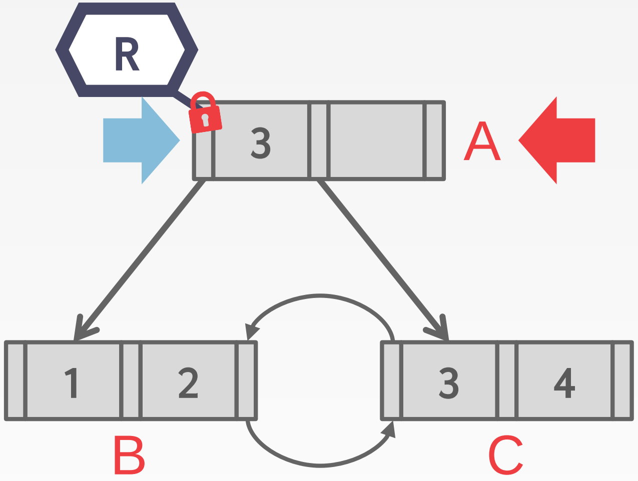
LEAF NODE SCAN EXAMPLE #2

T_1: Find Keys < 4

T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #3



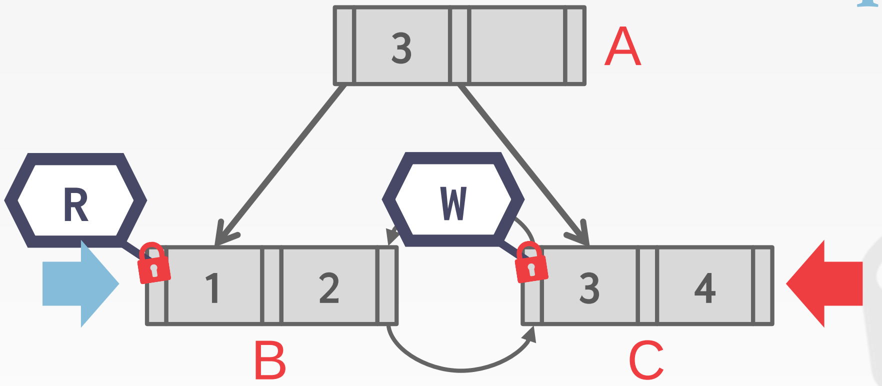
T_1: Delete 4

T_2: Find Keys > 1

LEAF NODE SCAN EXAMPLE #3

T_1: Delete 4

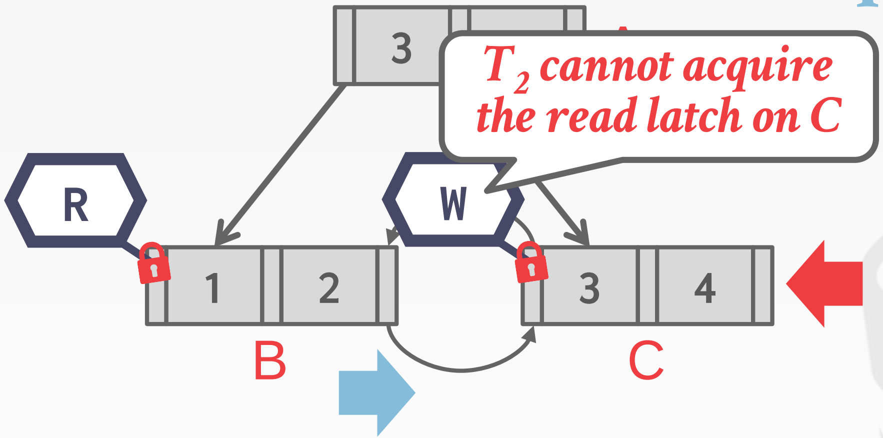
T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #3

T_1: Delete 4

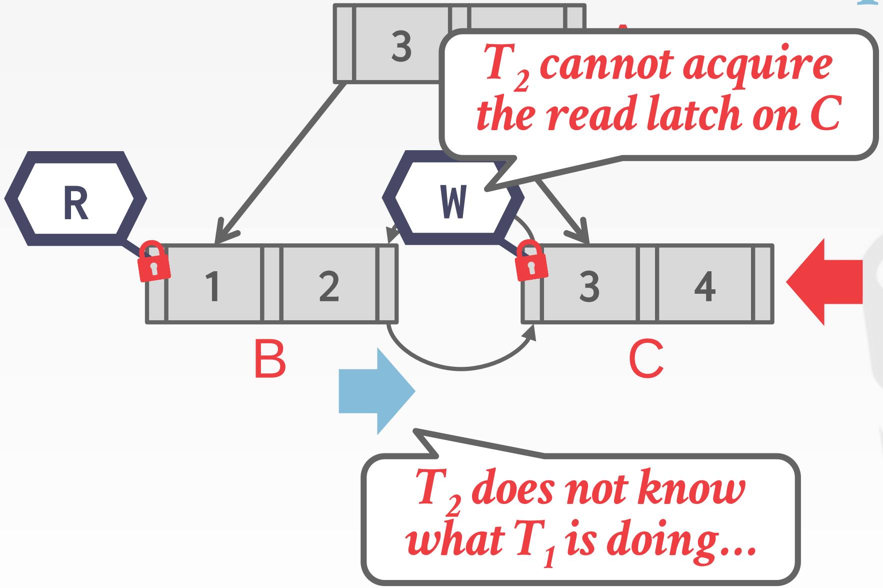
T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #3

T_1: Delete 4

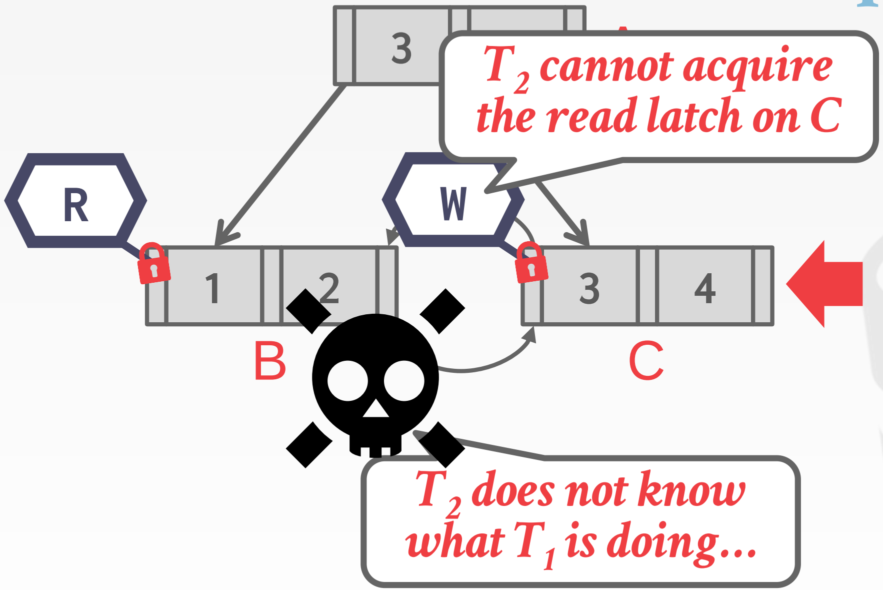
T_2: Find Keys > 1



LEAF NODE SCAN EXAMPLE #3

T_1: Delete 4

T_2: Find Keys > 1



LEAF NODE 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.
B+tree code must cope with failed latch acquisitions.

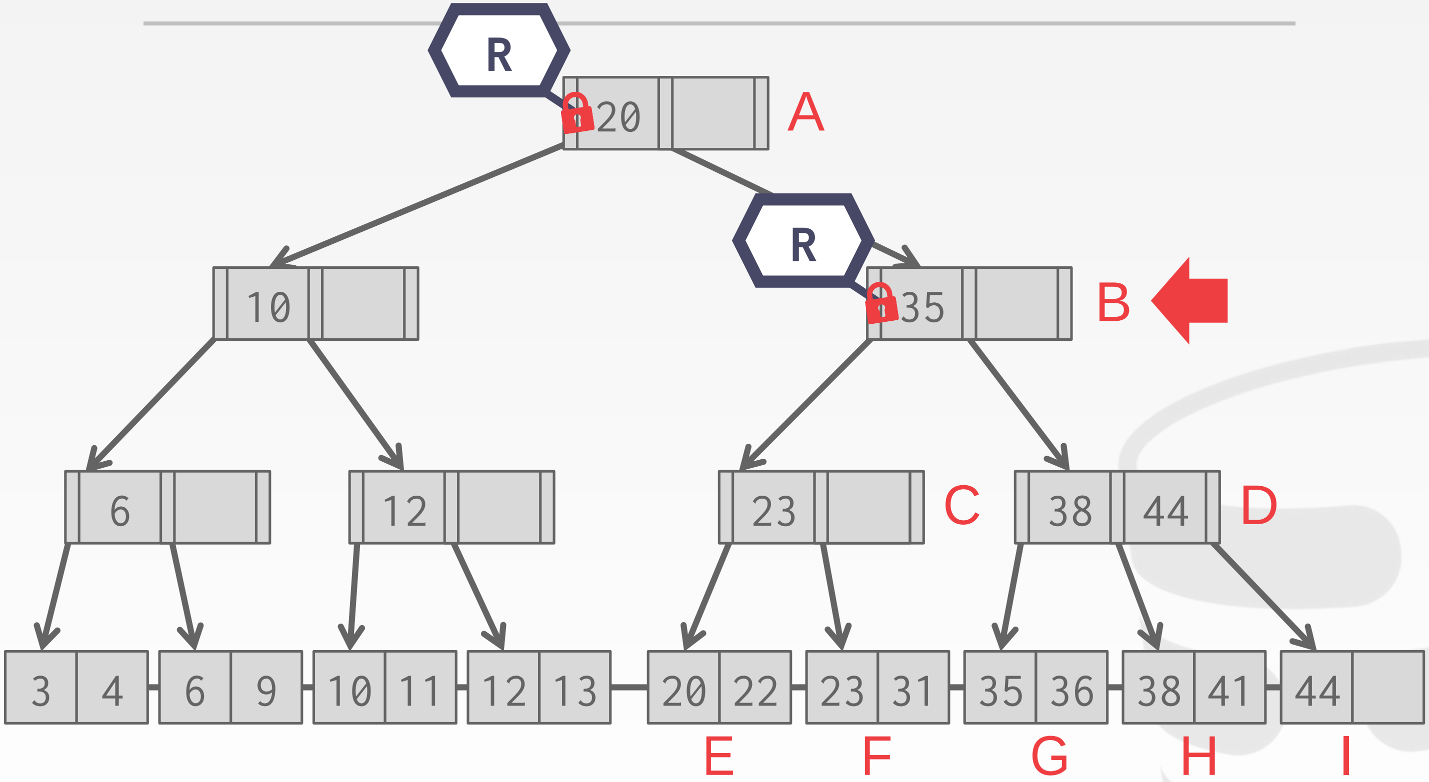
DELAYED PARENT UPDATES

Every time a leaf node overflows, we have to update at least three nodes.

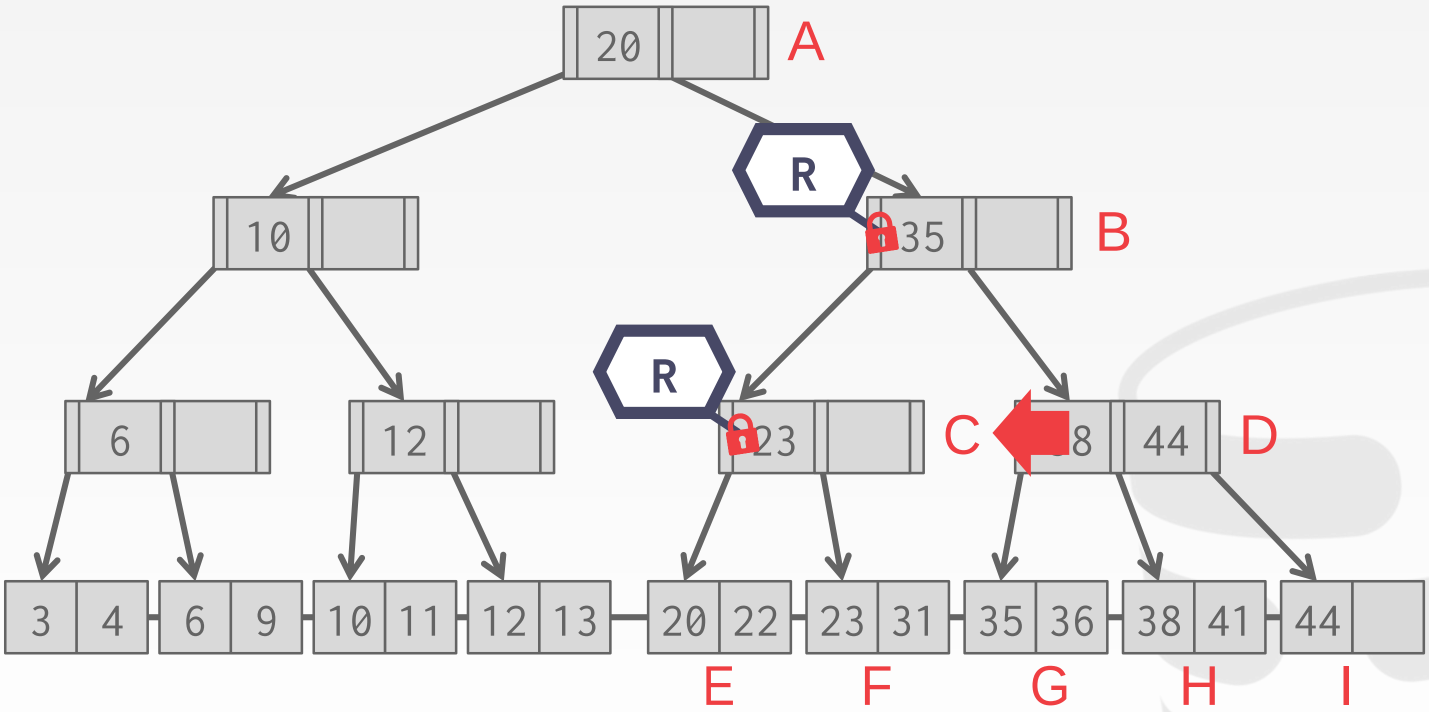
→ The leaf node being split.
→ The new leaf node being created.
→ The parent node.

B^{link}-Tree Optimization: When a leaf node overflows, delay updating its parent node.

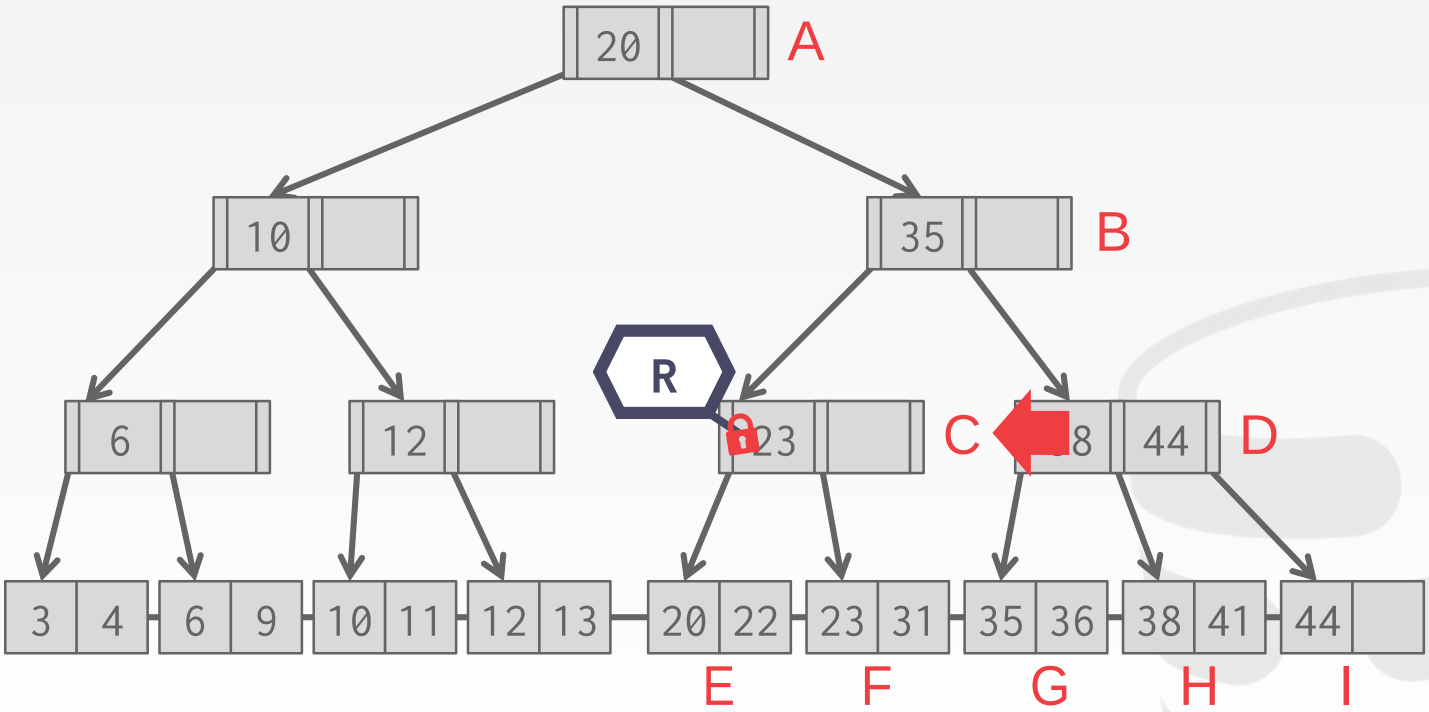
EXAMPLE #4 – INSERT 25



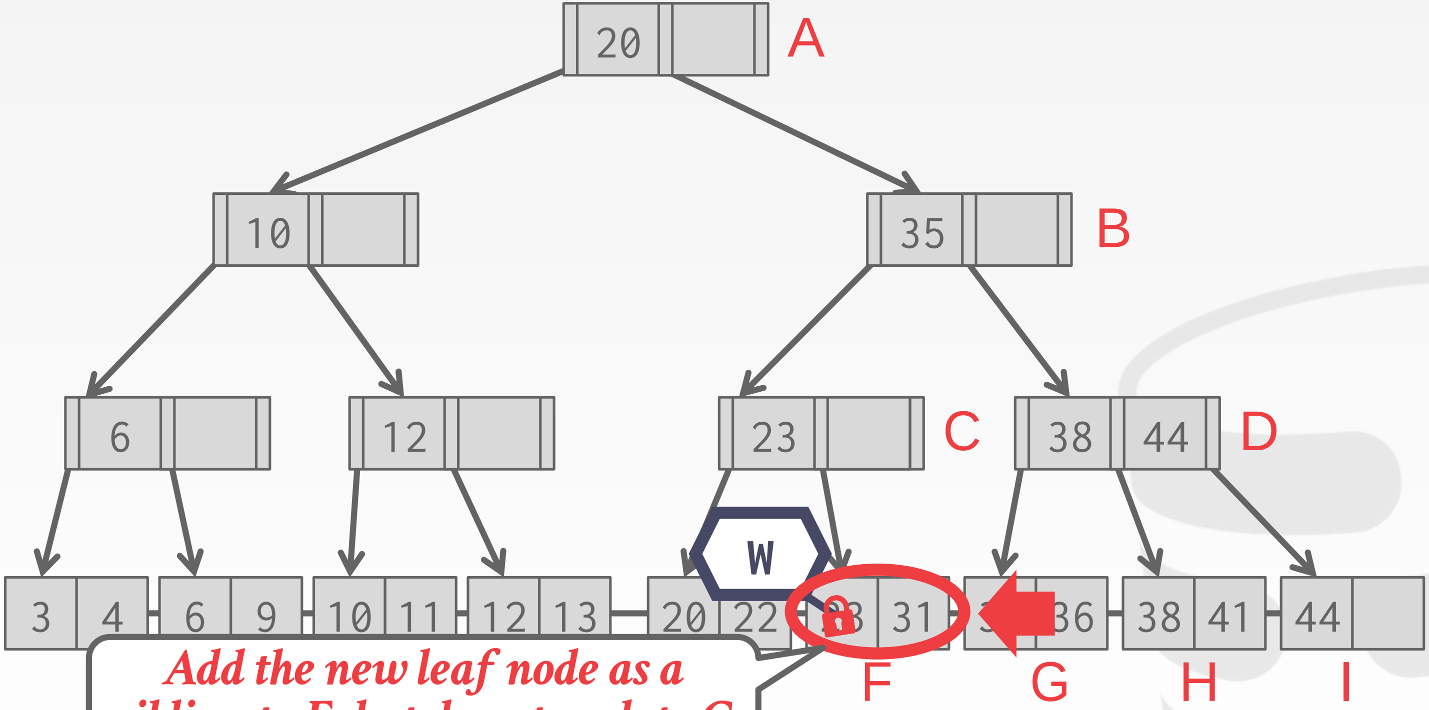
EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25

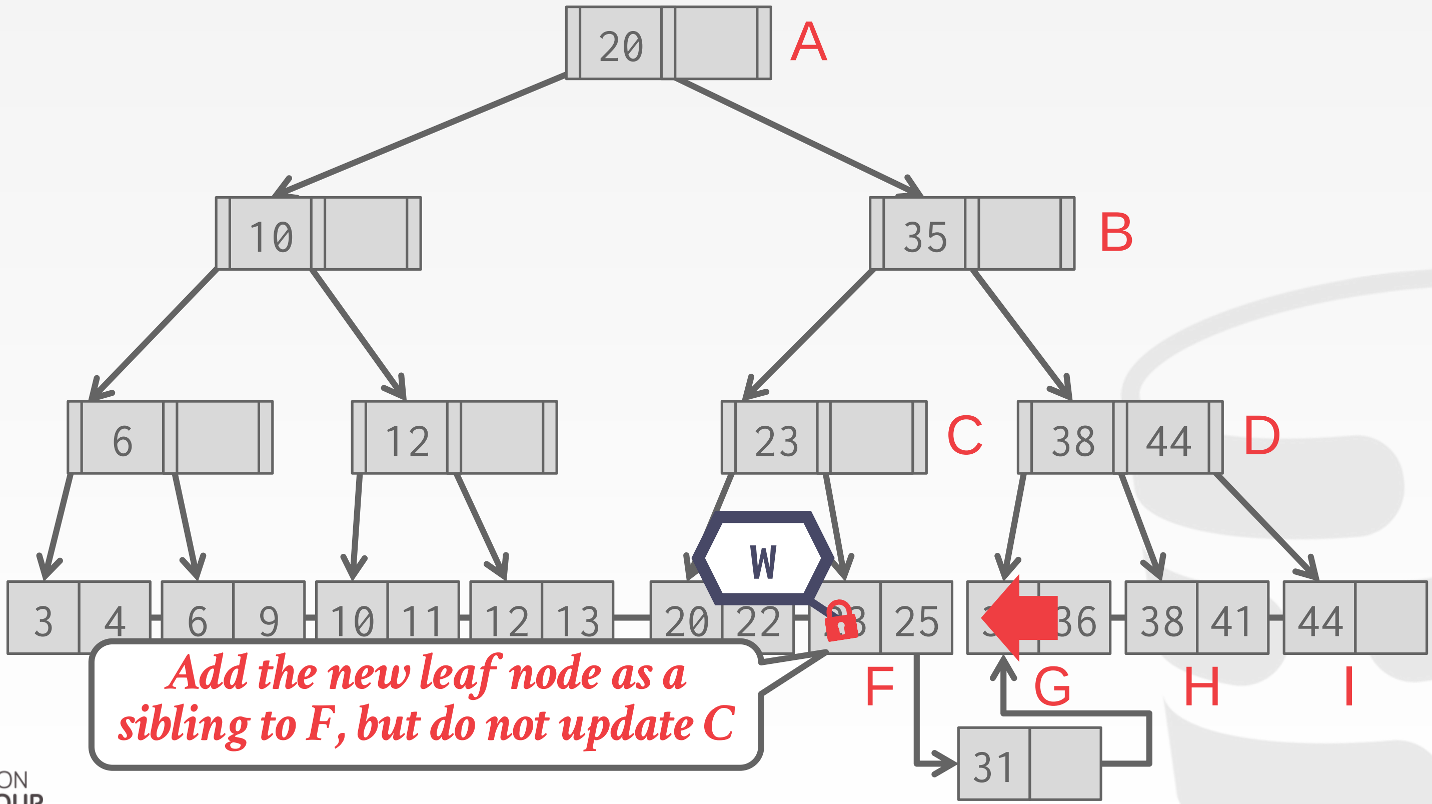


EXAMPLE #4 – INSERT 25

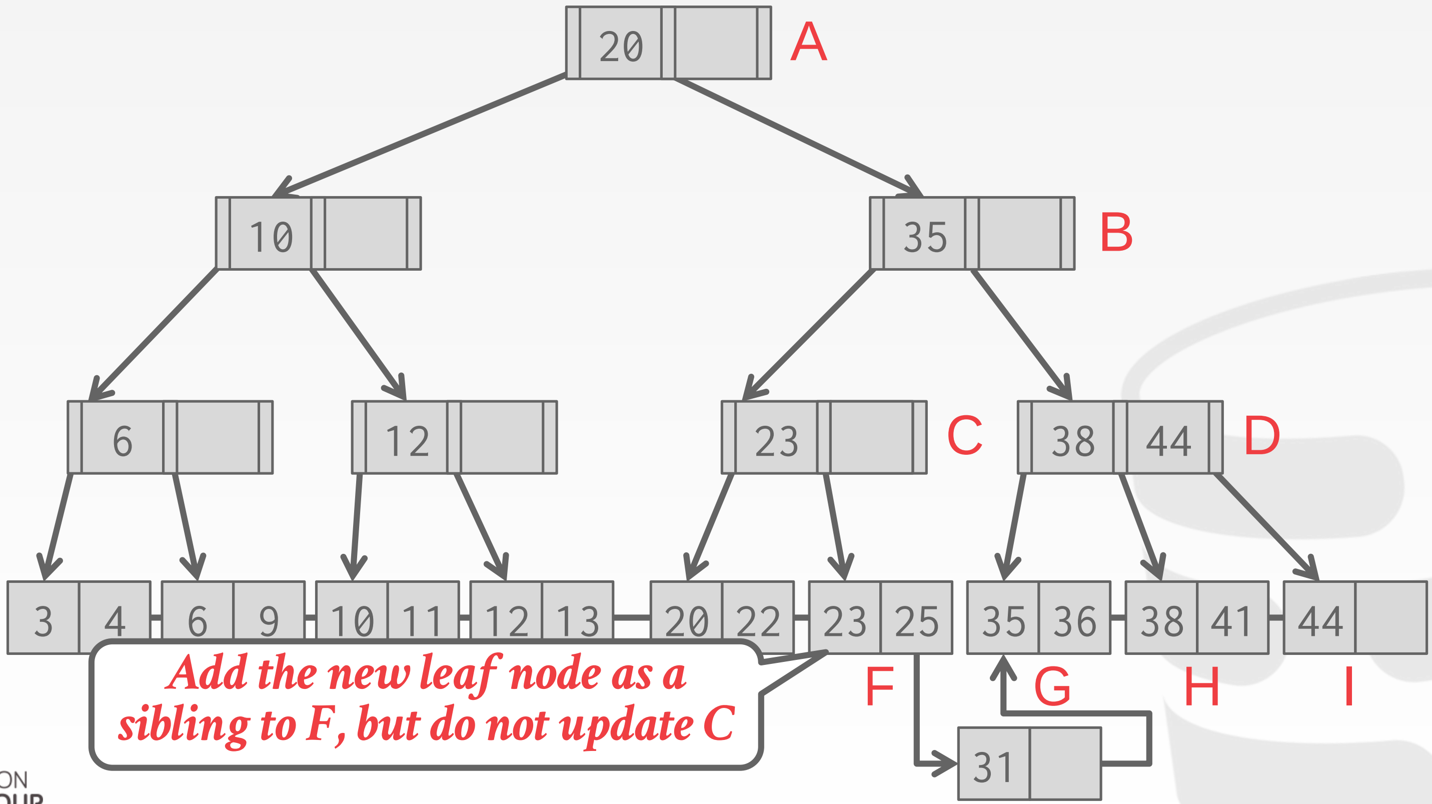


Add the new leaf node as a sibling to F, but do not update C

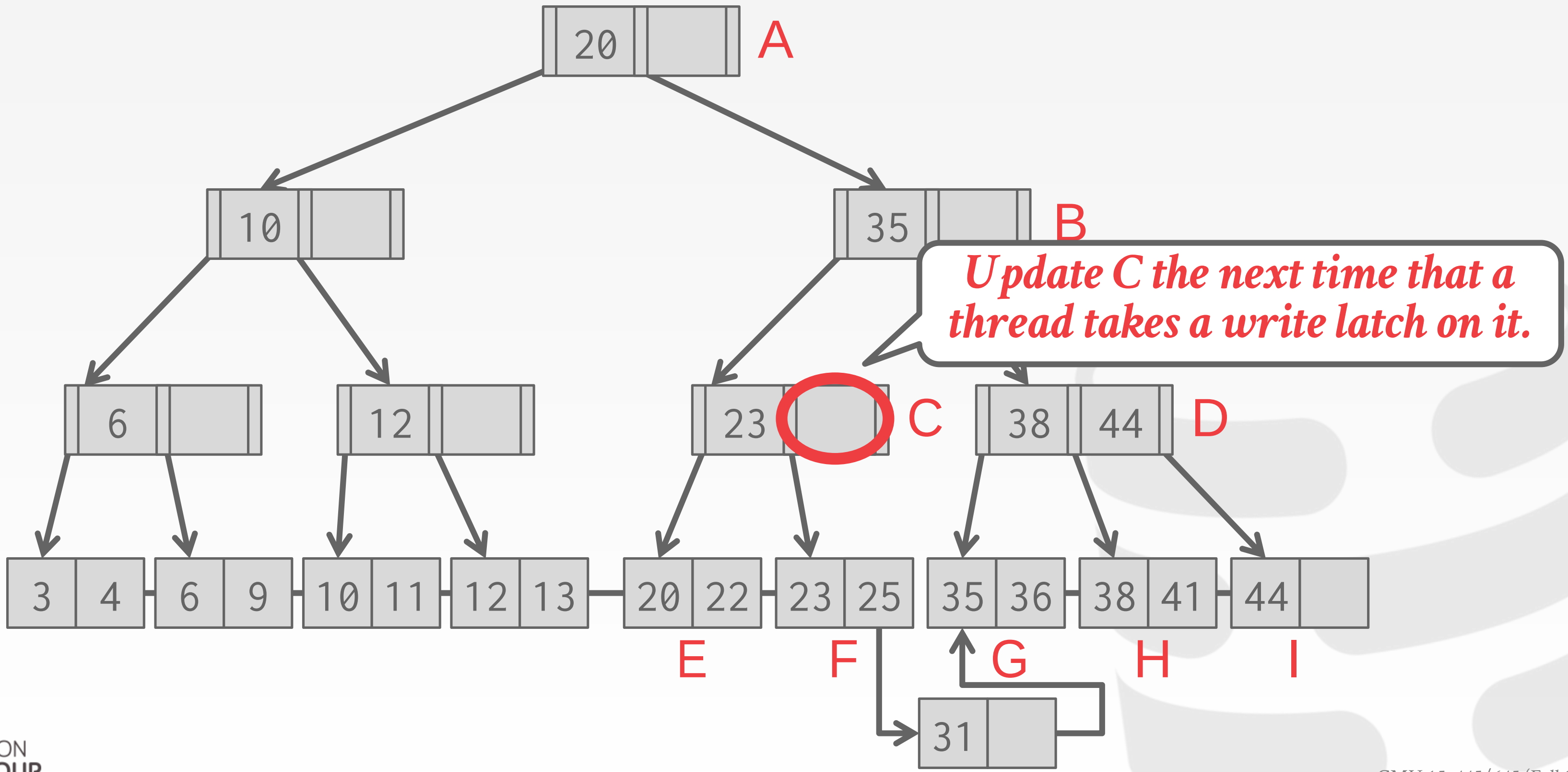
EXAMPLE #4 – INSERT 25



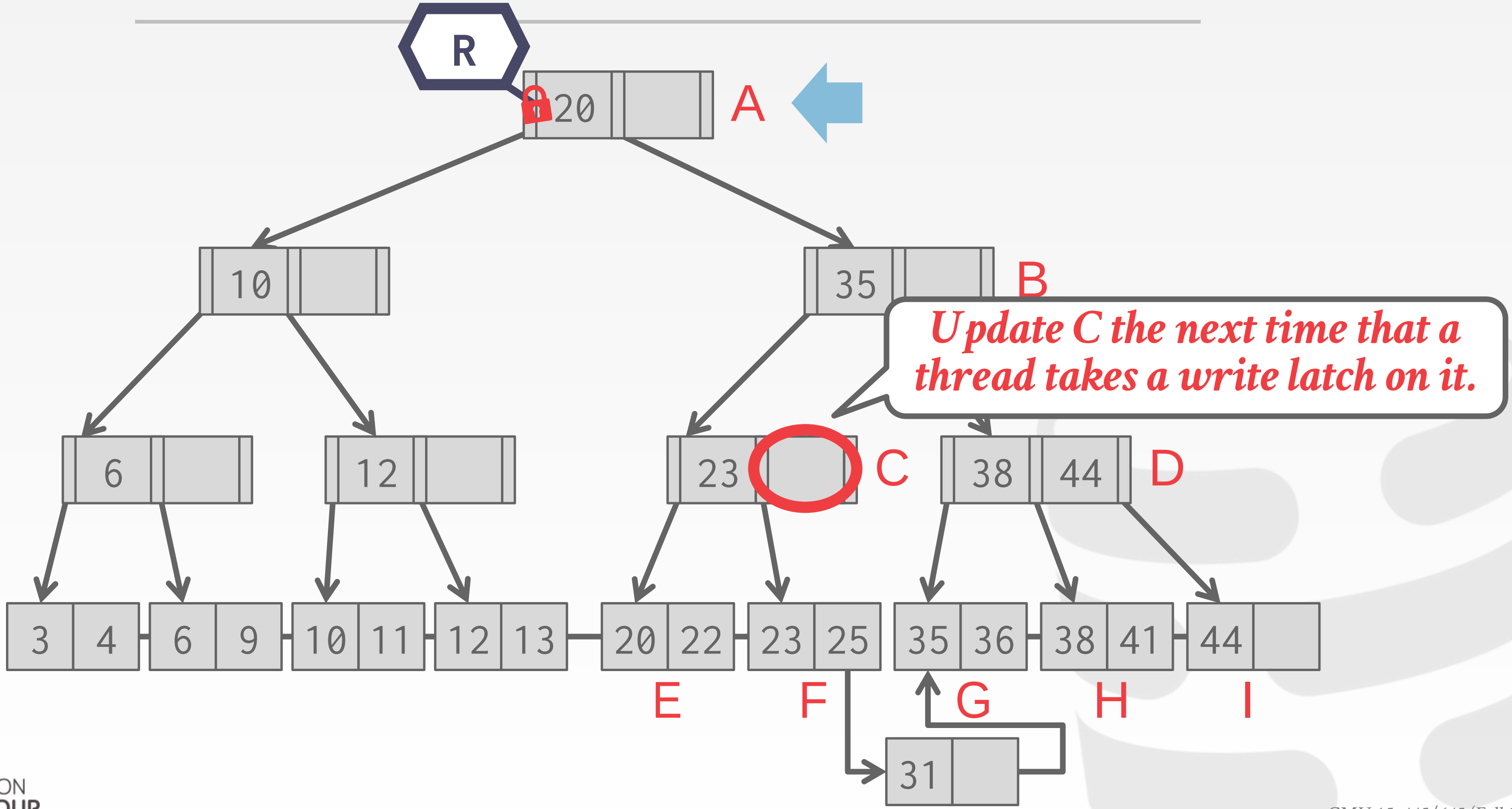
EXAMPLE #4 – INSERT 25



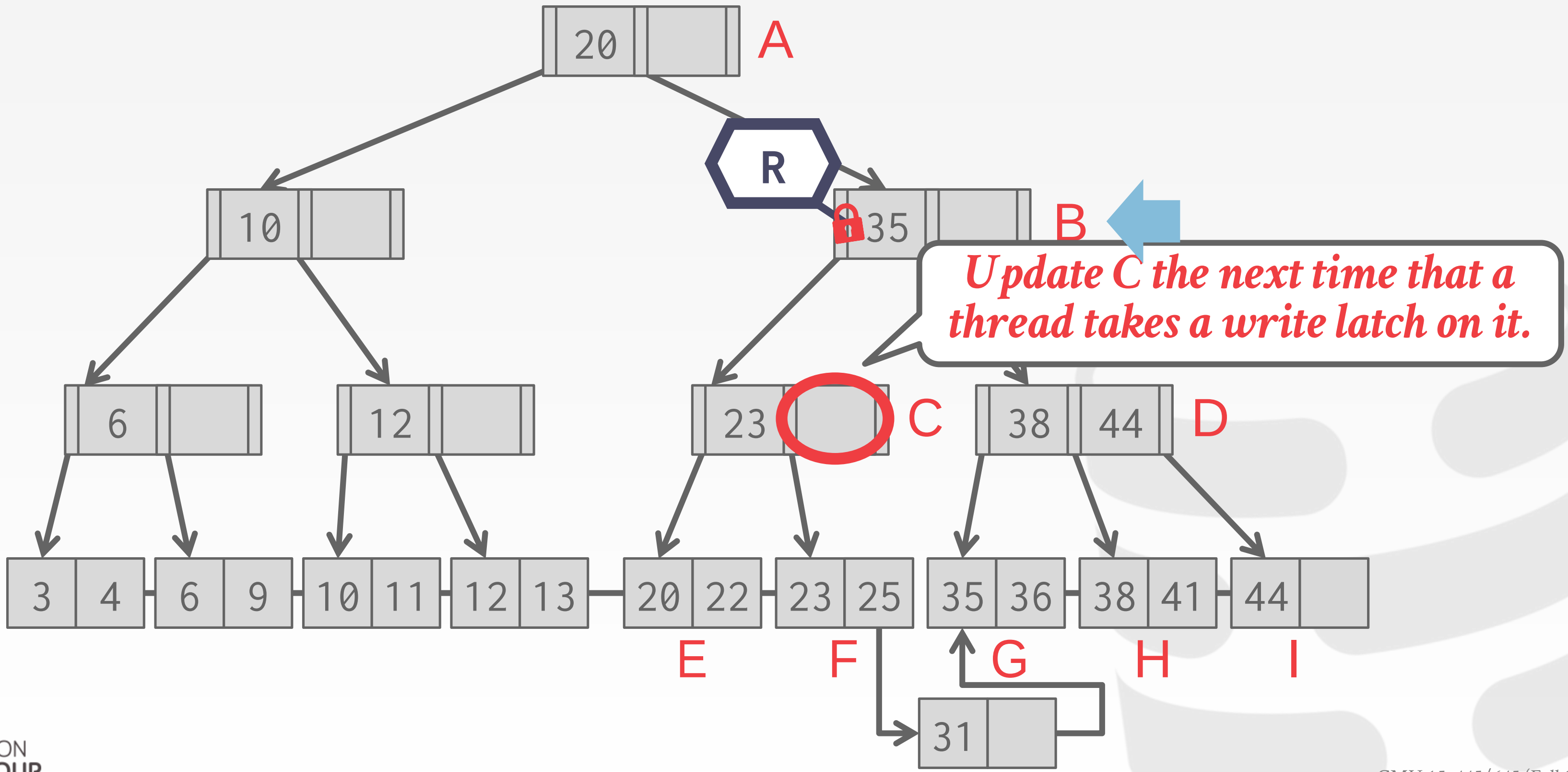
EXAMPLE #4 – INSERT 25



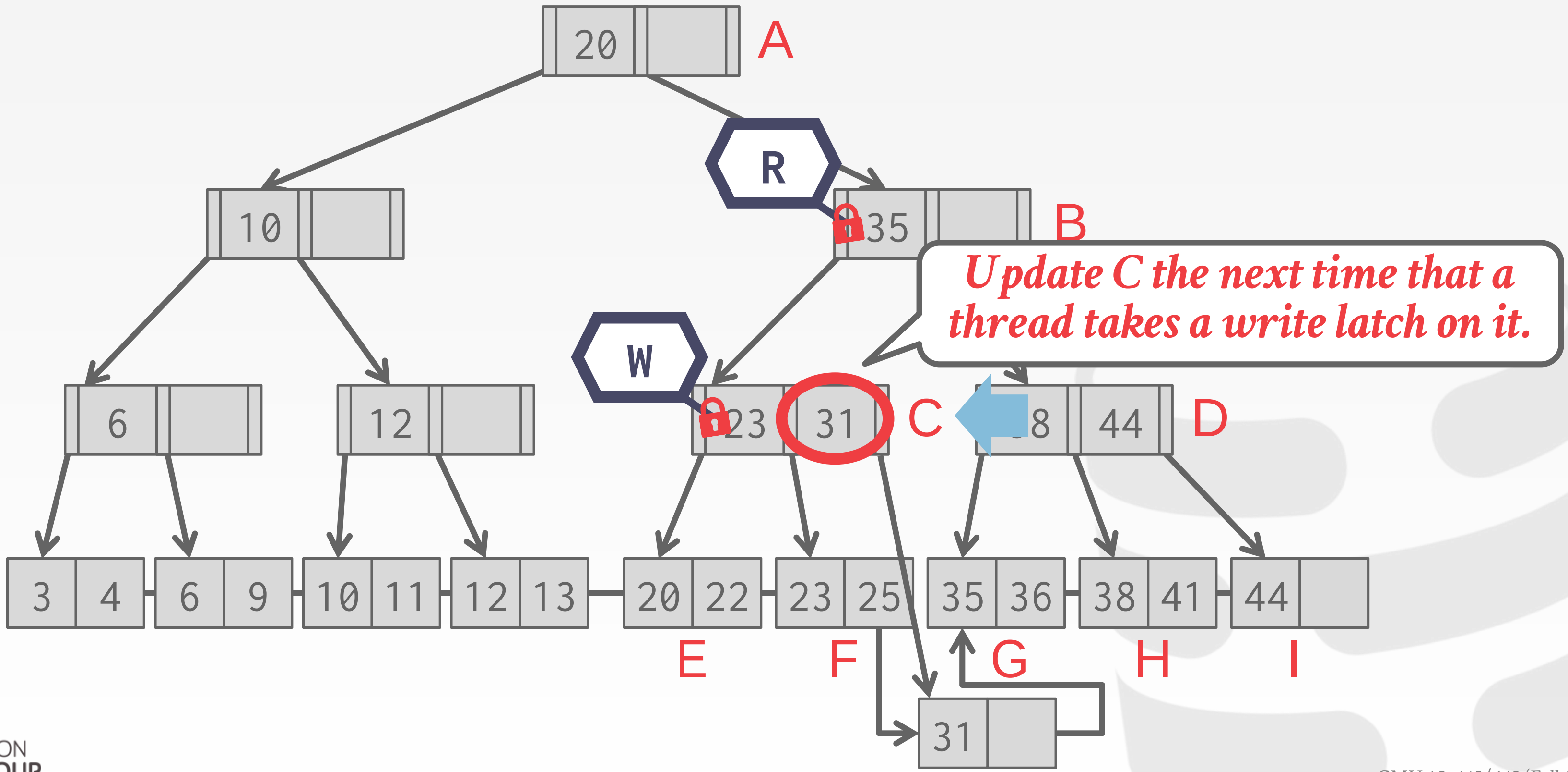
EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25



CONCLUSION

Making a data structure thread-safe seems easy to understand but it is notoriously difficult in practice.

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

PROJECT #2

You will build a thread-safe B+tree.
→ Page Layout
→ Data Structure
→ STL Iterator
→ Latch Crabbing

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

https://15445.courses.cs.cmu.edu/fall2018/project2/

CHECKPOINT #1

Due Date: October 8th @ 11:59pm
Total Project Grade: 40%

Page Layouts

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

Data Structure (Find + Insert)

→ Support point queries (single key).
→ Support inserts with node splitting.
→ Does not need to be thread-safe.

CHECKPOINT #2

Due Date: October 19th @ 11:59pm
Total Project Grade: 60%

Data Structure (Deletion)

→ Support removal of keys with sibling stealing + merging.

Index Iterator

→ Create a STL iterator for range scans.

Concurrent Index

→ Implement latch crabbing/coupling.

DEVELOPMENT HINTS

Follow the textbook semantics and algorithms.
→ See Chapter 15.10

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

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

THINGS TO NOTE

Do **not** change any file other than the ten that you have to hand it.

We will provide an updated source tarball. You will need to copy over your files from Project #1.

Post your questions on Piazza or come to TA office hours.

PLAGIARISM WARNING

Your project implementation must be your own work.

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

→ Do **not** publish your implementation on Github.

Plagiarism will **not** be tolerated.
See CMU's Policy on Academic Integrity for additional information.

NEXT CLASS

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