Carnegie Mellon University

Index Concurrency Control

Lecture #18

Database Systems
15-445/15-645
Fall 2017

Andy Pavlo
Computer Science Dept.
Carnegie Mellon Univ.

TODAY'S AGENDA

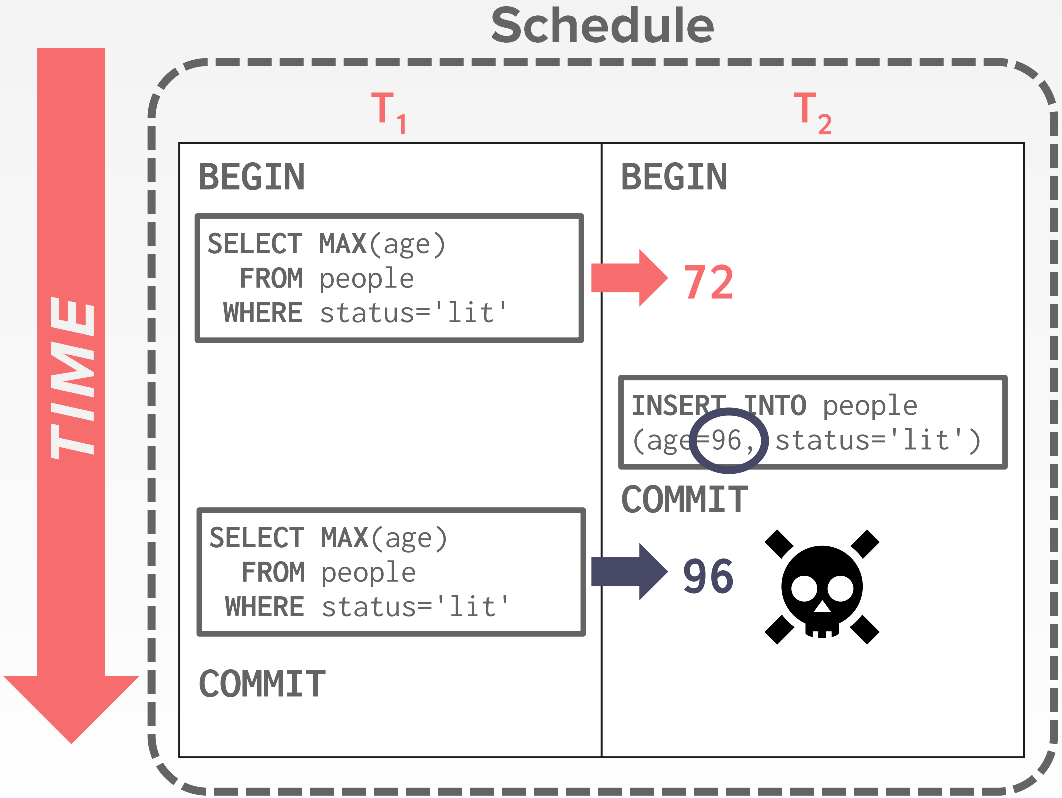
Phantom Problem
Index Locking
Isolation Levels
Index Crabbing

DYNAMIC DATABASES

Recall that so far we have only dealing with transactions that read and update data.

But now if we have insertions, updates, and deletions, we have new problems…

THE PHANTOM PROBLEM



T_1	T_2
BEGIN	BEGIN
`SELECT MAX(age) FROM people WHERE status='lit'`	
	`INSERT INTO people (age=96, status='lit')`
	COMMIT
`SELECT MAX(age) FROM people WHERE status='lit'`	
COMMIT	

```
CREATE TABLE people (
  id SERIAL,
  name VARCHAR,
  age INT,
  status VARCHAR
);
```

WTF?

How did this happen?

→ Because T_1 locked only existing records and not ones under way!

Conflict serializability on reads and writes of individual items guarantees serializability **only** if the set of objects is fixed.

We will solve this problem in the next class.

PREDICATE LOCKING

Lock records that satisfy a logical predicate:

- → Example: `status='lit'`

In general, predicate locking has a lot of locking overhead.

Index locking is a special case of predicate locking that is potentially more efficient.

INDEX LOCKING

If there is a dense index on the status field then the txn can lock index page containing the data with `status='lit'`.

If there are no records with `status='lit'`, the txn must lock the index page where such a data entry would be, if it existed.

LOCKING WITHOUT AN INDEX

If there is no suitable index, then the txn must obtain:

- A lock on every page in the table to prevent a record's `status='lit'` from being changed to `lit`.
- The lock for the table itself to prevent records with `status='lit'` from being added or deleted.

REPEATING SCANS

An alternative is to just re-execute every scan again when the txn commits and check whether it gets the same result.

- → Have to retain the scan set for every range query in a txn.
- → Andy doesn't know of any commercial system that does this (only just Silo?).

WEAKER LEVELS OF ISOLATION

Serializability is useful because it allows programmers to ignore concurrency issues.

But enforcing it may allow too little concurrency and limit performance.

We may want to use a weaker level of consistency to improve scalability.

ISOLATION LEVELS

Controls the extent that a txn is exposed to the actions of other concurrent txns.

Provides for greater concurrency at the cost of exposing txns to uncommitted changes:

- → Dirty Reads
- → Unrepeatable Reads
- → Phantom Reads

ISOLATION LEVELS

Isolation (High→Low)

SERIALIZABLE: No phantoms, all reads repeatable, no dirty reads.

REPEATABLE READS: Phantoms may happen.

READ COMMITTED: Phantoms and unrepeatable reads may happen.

READ UNCOMMITTED: All of them may happen.

ISOLATION LEVELS

	Dirty Read	Unrepeatable Read	Phantom
SERIALIZABLE	No	No	No
REPEATABLE READ	No	No	Maybe
READ COMMITTED	No	Maybe	Maybe
READ UNCOMMITTED	Maybe	Maybe	Maybe

ISOLATION LEVELS

SERIALIZABLE: Obtain all locks first; plus index locks, plus strict 2PL.

REPEATABLE READS: Same as above, but no index locks.

READ COMMITTED: Same as above, but **S** locks are released immediately.

READ UNCOMMITTED: Same as above, but allows dirty reads (no **S** locks).

SQL-92 ISOLATION LEVELS

Not all DBMS support all isolation levels in all execution scenarios (e.g., replication).

The default depends on implementation…

```
SET TRANSACTION ISOLATION LEVEL
  <isolation-level>;
```

ISOLATION LEVELS (2013)

	Default	Maximum
Actian Ingres 10.0/10S	SERIALIZABLE	SERIALIZABLE
Aerospike	READ COMMITTED	READ COMMITTED
Greenplum 4.1	READ COMMITTED	SERIALIZABLE
MySQL 5.6	REPEATABLE READS	SERIALIZABLE
MemSQL 1b	READ COMMITTED	READ COMMITTED
MS SQL Server 2012	READ COMMITTED	SERIALIZABLE
Oracle 11g	READ COMMITTED	SNAPSHOT ISOLATION
Postgres 9.2.2	READ COMMITTED	SERIALIZABLE
SAP HANA	READ COMMITTED	SERIALIZABLE
ScaleDB 1.02	READ COMMITTED	READ COMMITTED
VoltDB	SERIALIZABLE	SERIALIZABLE

Source: Peter Bailis

SQL-92 ACCESS MODES

You can also provide hints to the DBMS about whether a txn will modify the database.

```
SET TRANSACTION <access-mode>;
```

Only two possible modes:

- → `READ WRITE` (Default)
- → `READ ONLY`

Not all DBMSs will optimize execution if you set a txn to in **`READ ONLY`** mode.

LOCKING IN B+TREES

What about locking indexes?

They are not quite like other database elements so we can treat them differently:

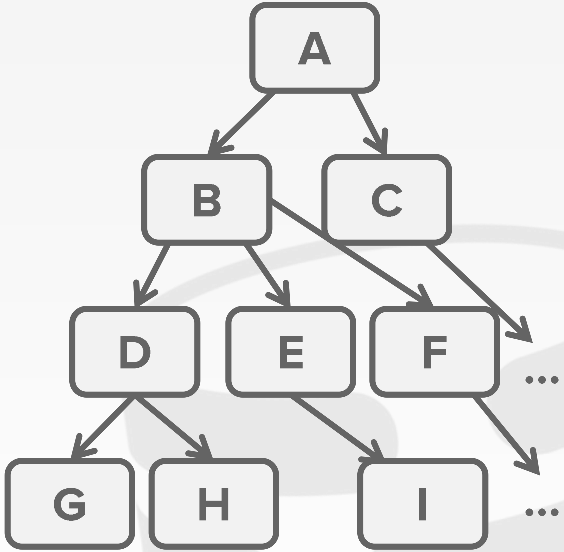
→ It's okay to have non-serializable concurrent access to an index as long as the accuracy of the index is maintained.

EXAMPLE

T_1 wants to insert entry in node **H**

T_2 wants to insert entry in node **I**

Why not use 2PL?



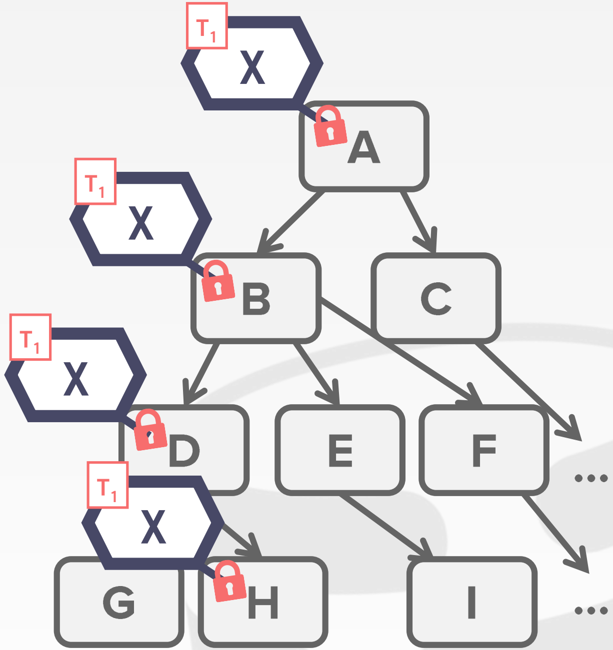
EXAMPLE

T_1 wants to insert entry in node **H**

T_2 wants to insert entry in node **I**

Why not use 2PL?

Because txns have to hold on to their locks for too long!



LOCK CRABBING

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

Basic Idea:

→ Get lock for parent.
→ Get lock for child
→ Release lock 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)

LOCK CRABBING

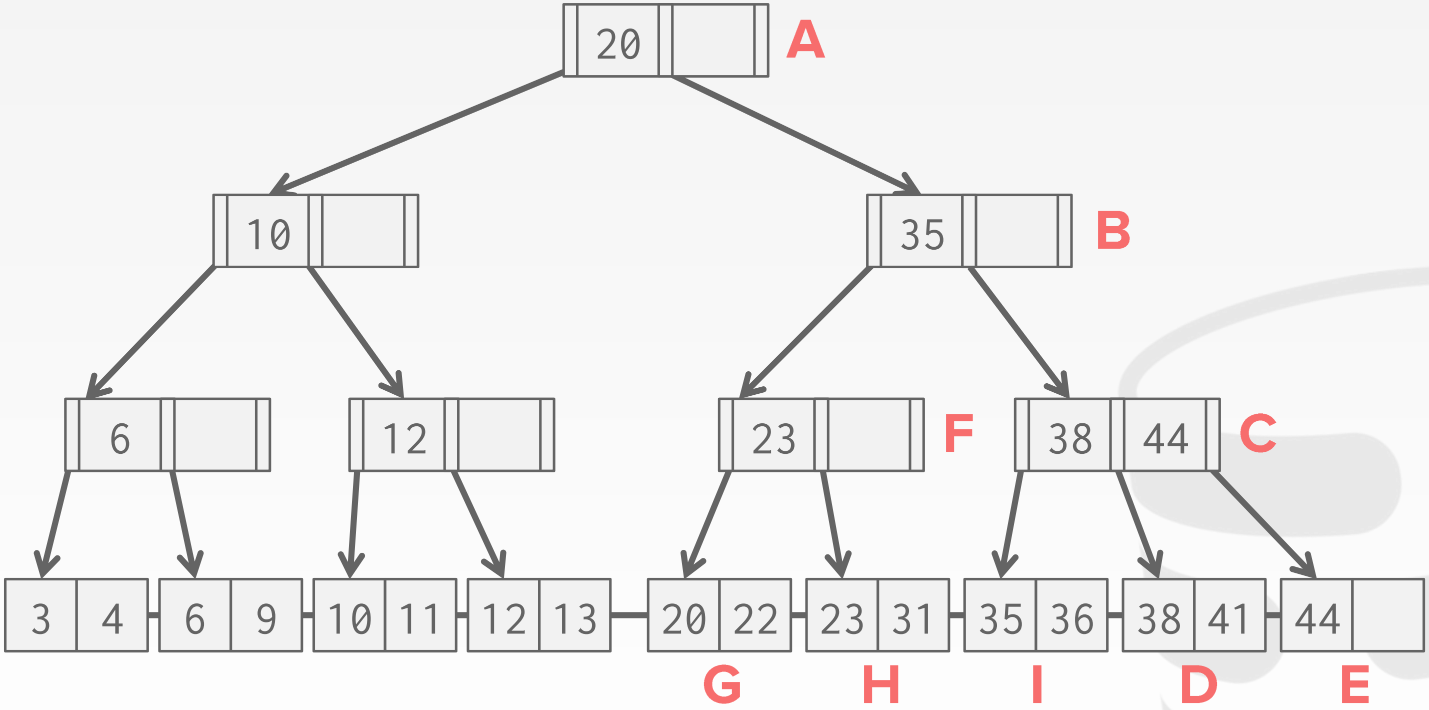
Search: Start at root and go down; repeatedly,

- → Acquire S lock on child
- → Then unlock parent

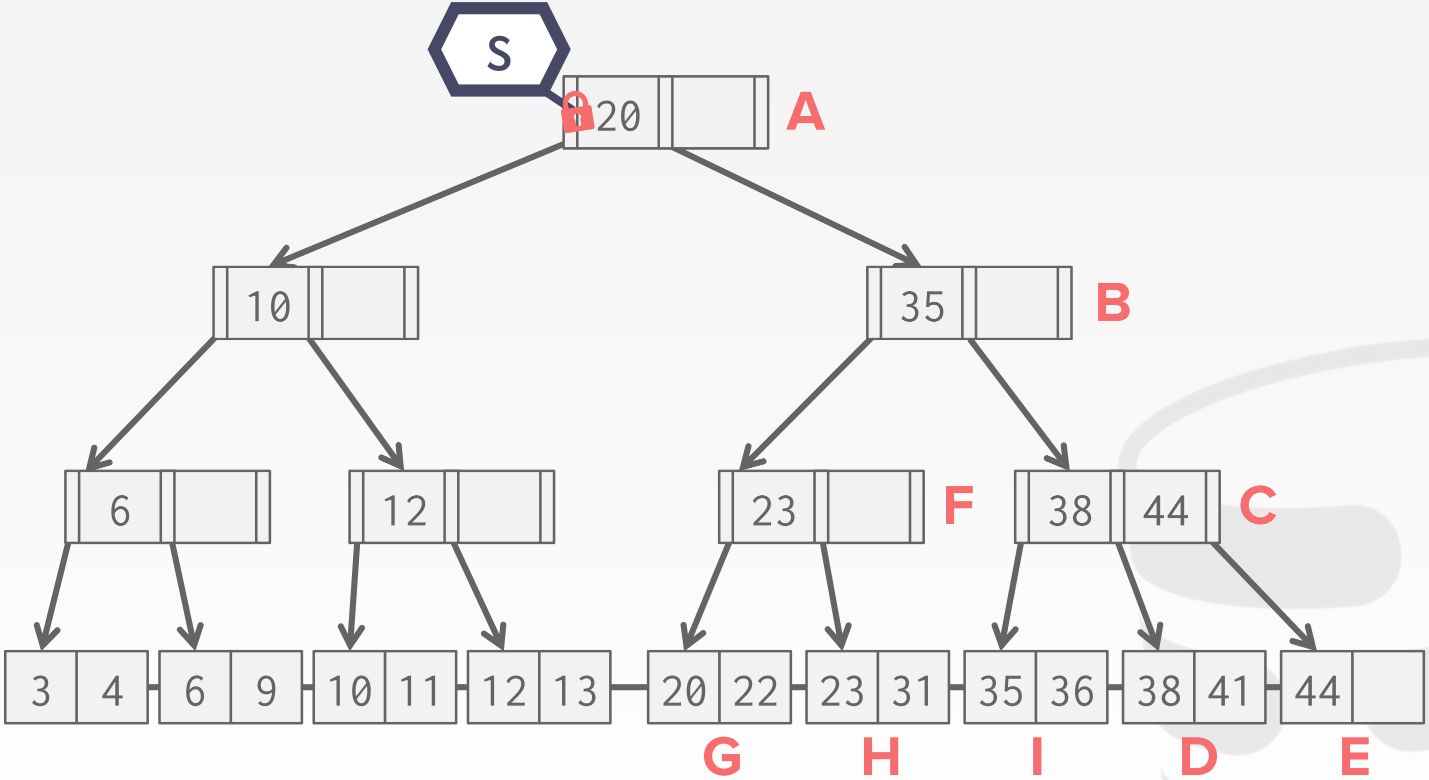
Insert/Delete: Start at root and go down, obtaining X locks as needed. Once child is locked, check if it is safe:

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

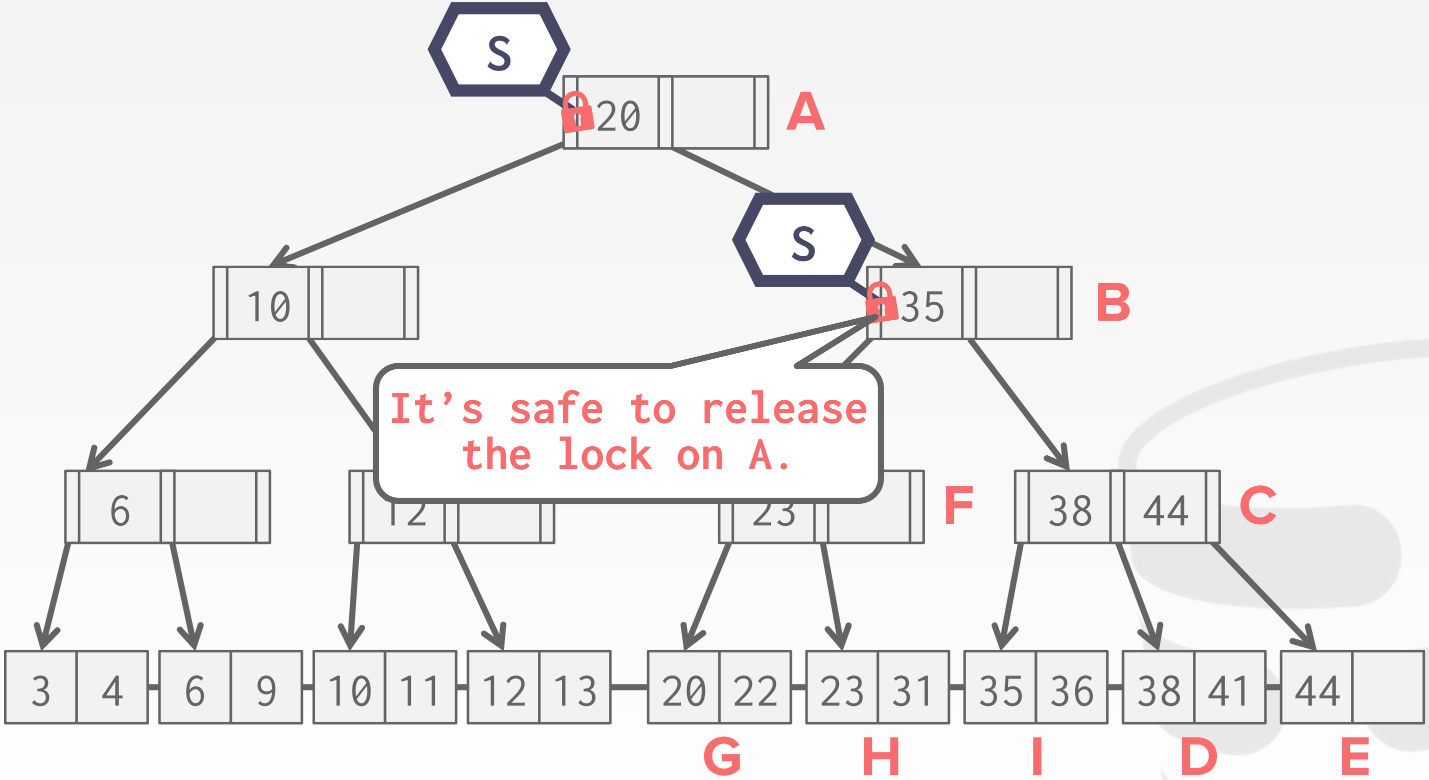
EXAMPLE #1 – SEARCH 38



EXAMPLE #1 – SEARCH 38

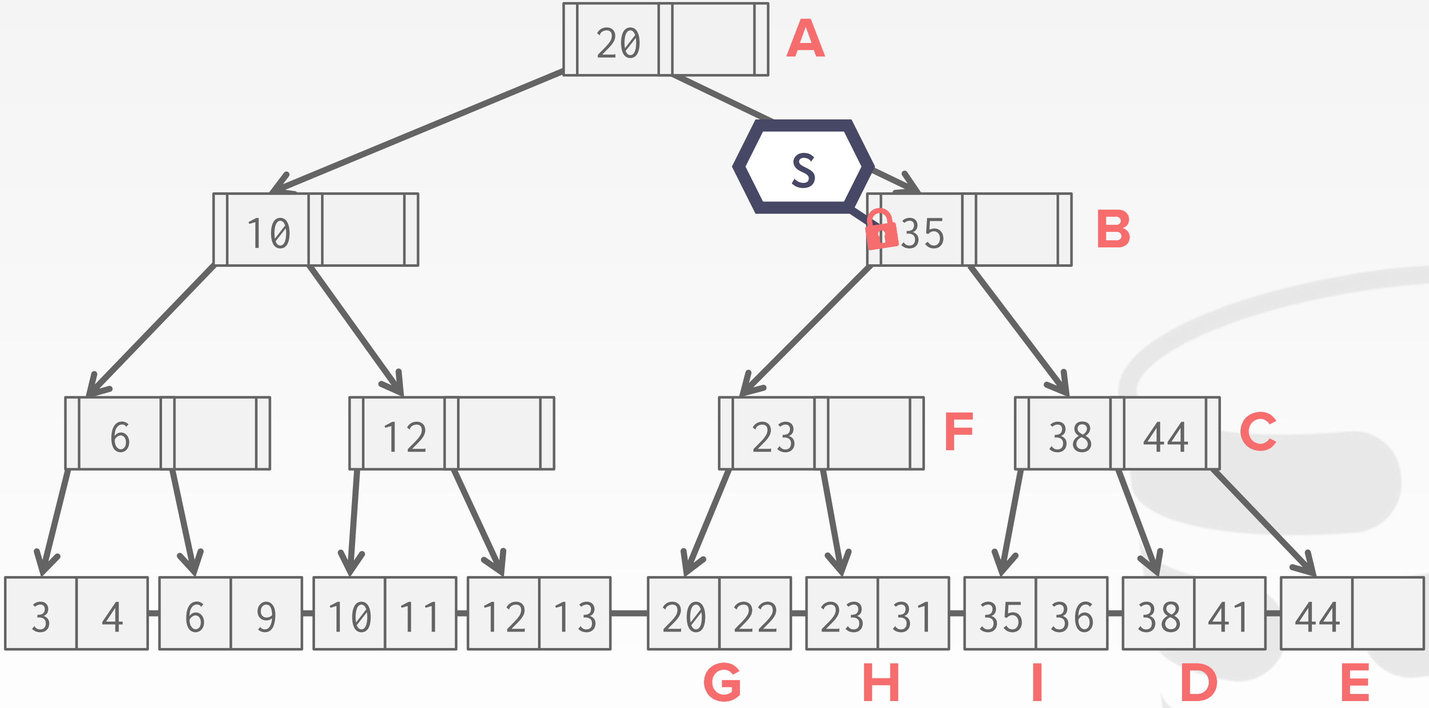


EXAMPLE #1 – SEARCH 38

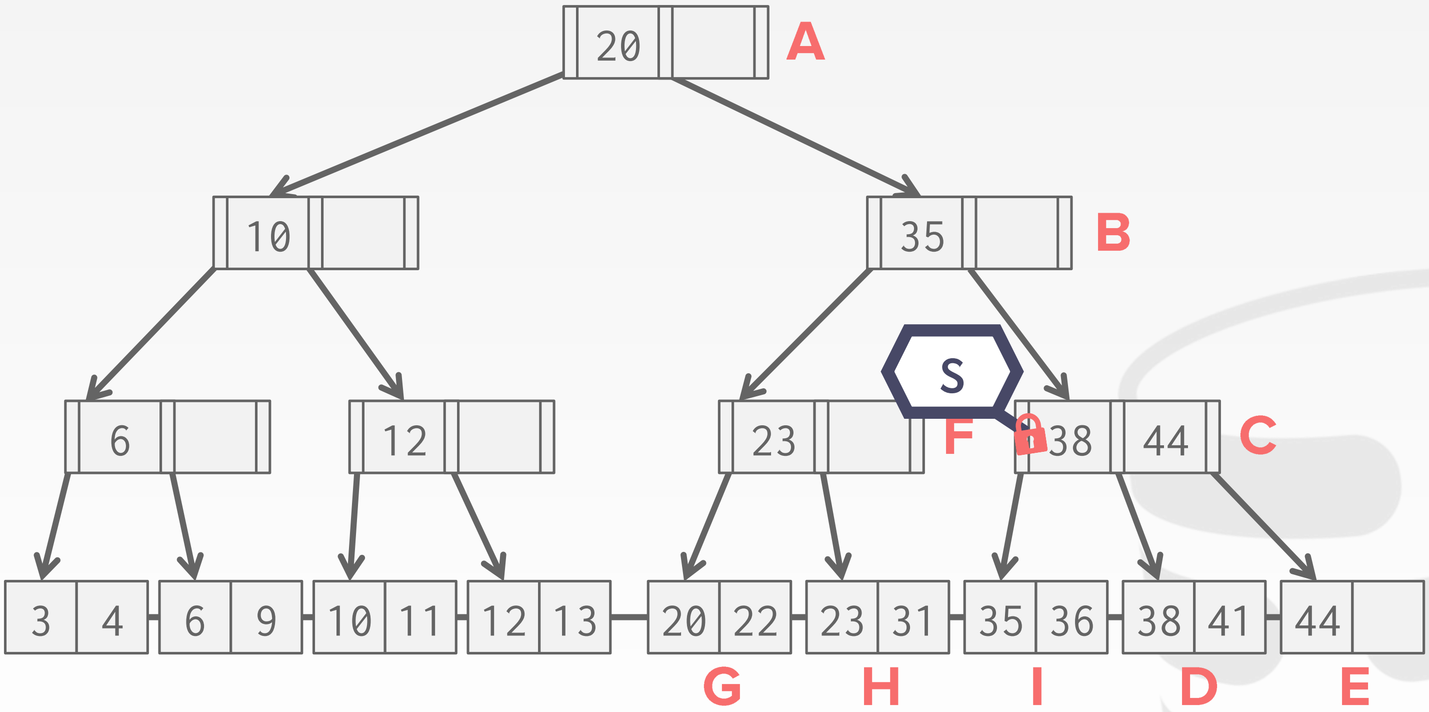


It's safe to release the lock on A.

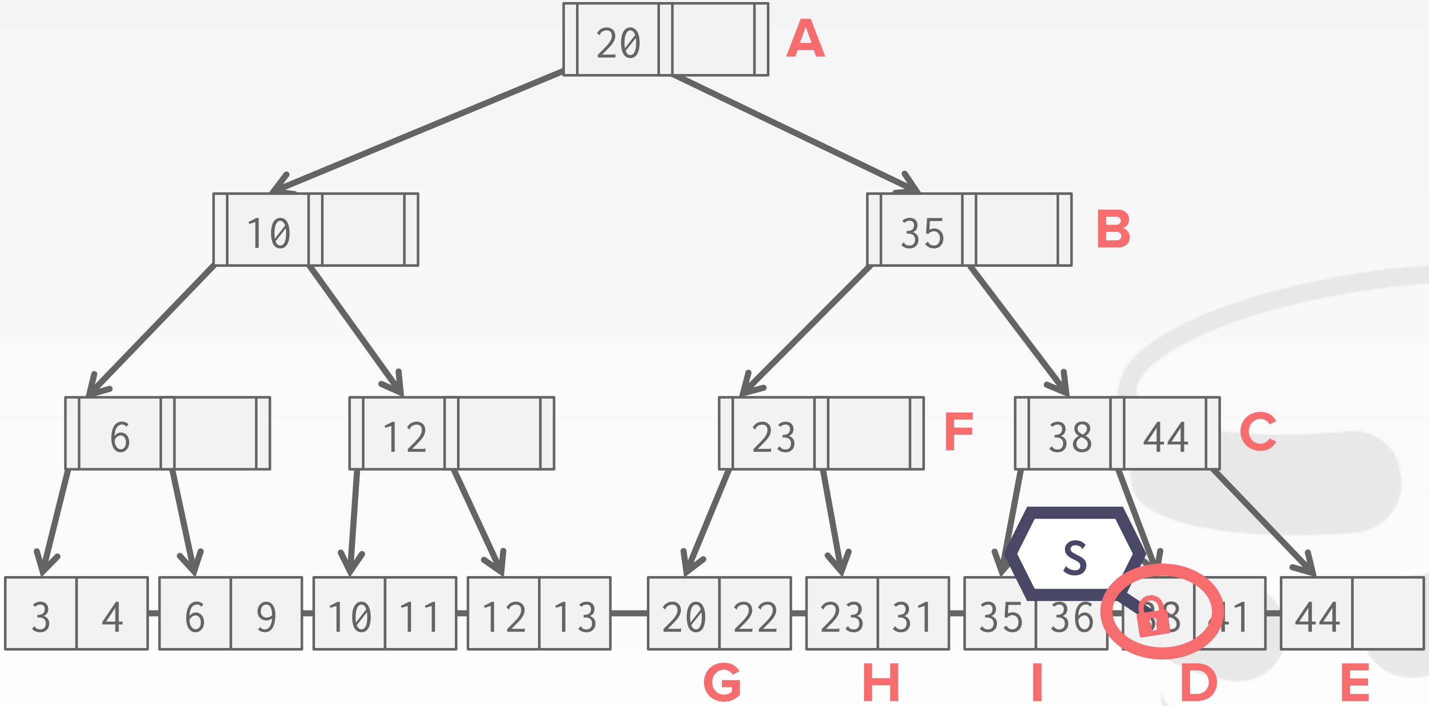
EXAMPLE #1 – SEARCH 38



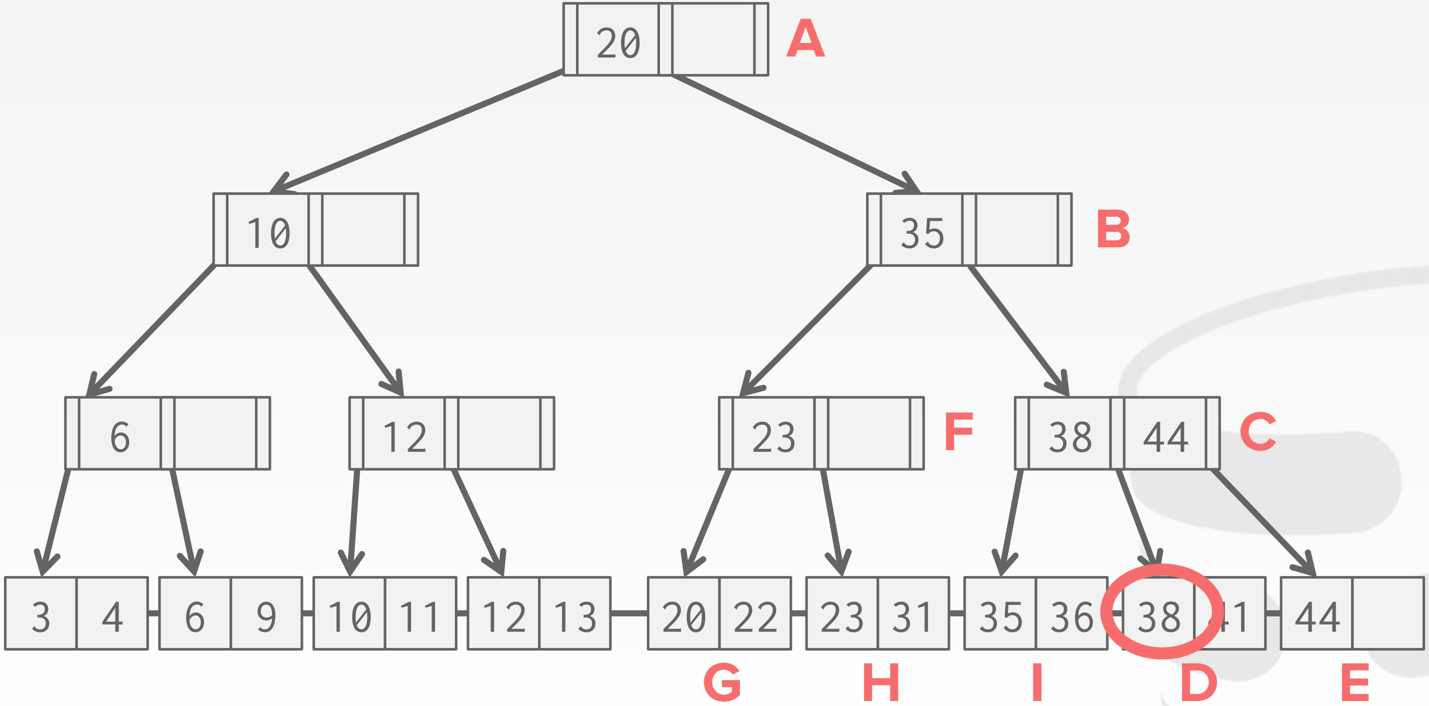
EXAMPLE #1 – SEARCH 38



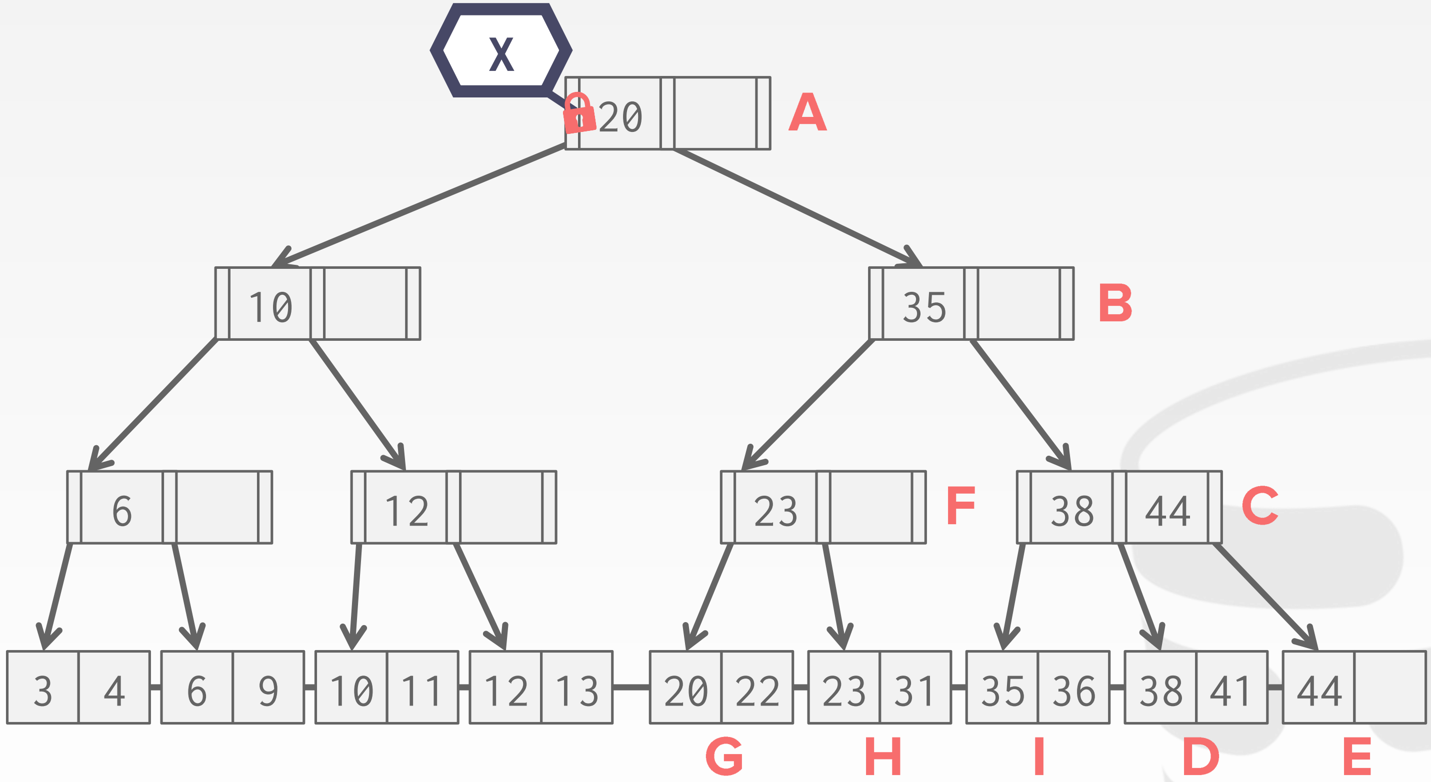
EXAMPLE #1 – SEARCH 38



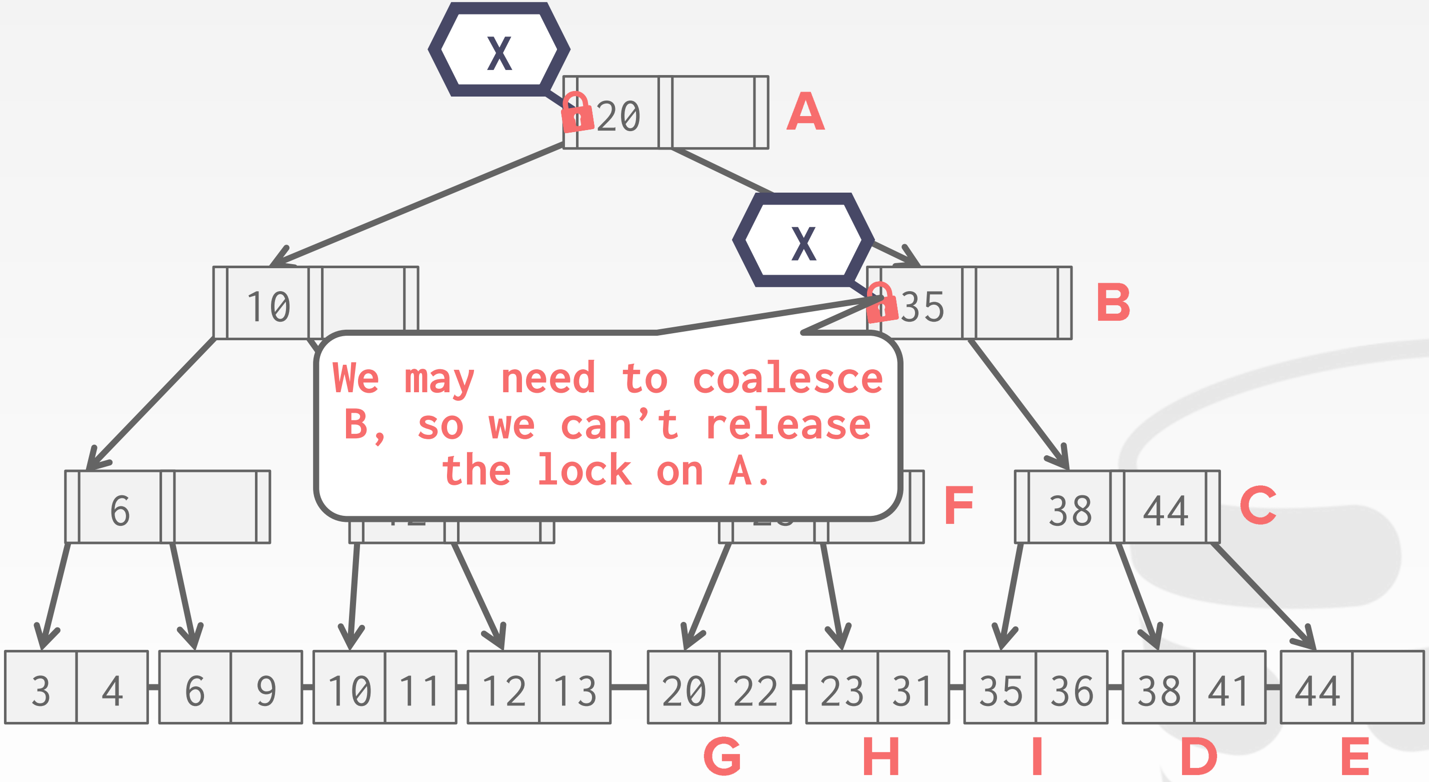
EXAMPLE #1 – SEARCH 38



EXAMPLE #2 – DELETE 38

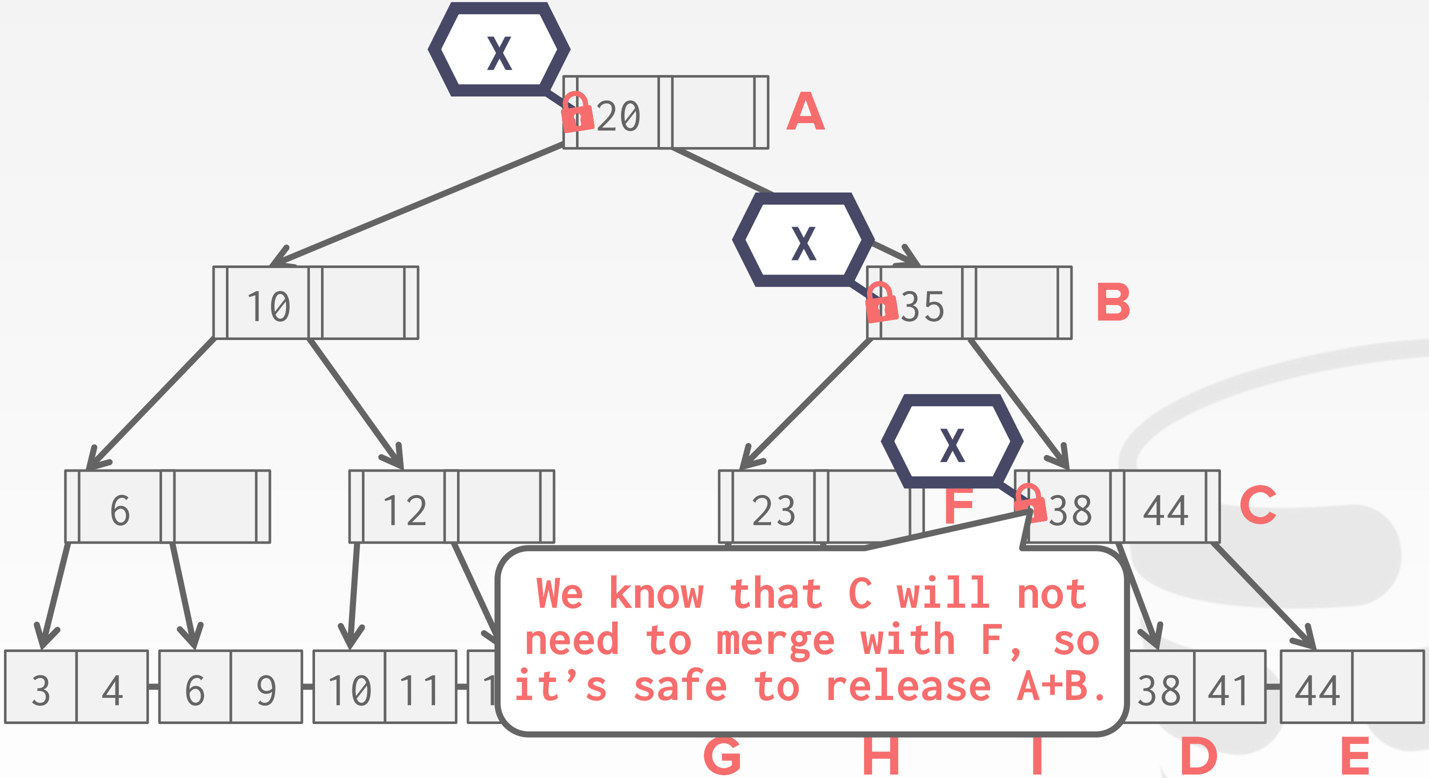


EXAMPLE #2 – DELETE 38



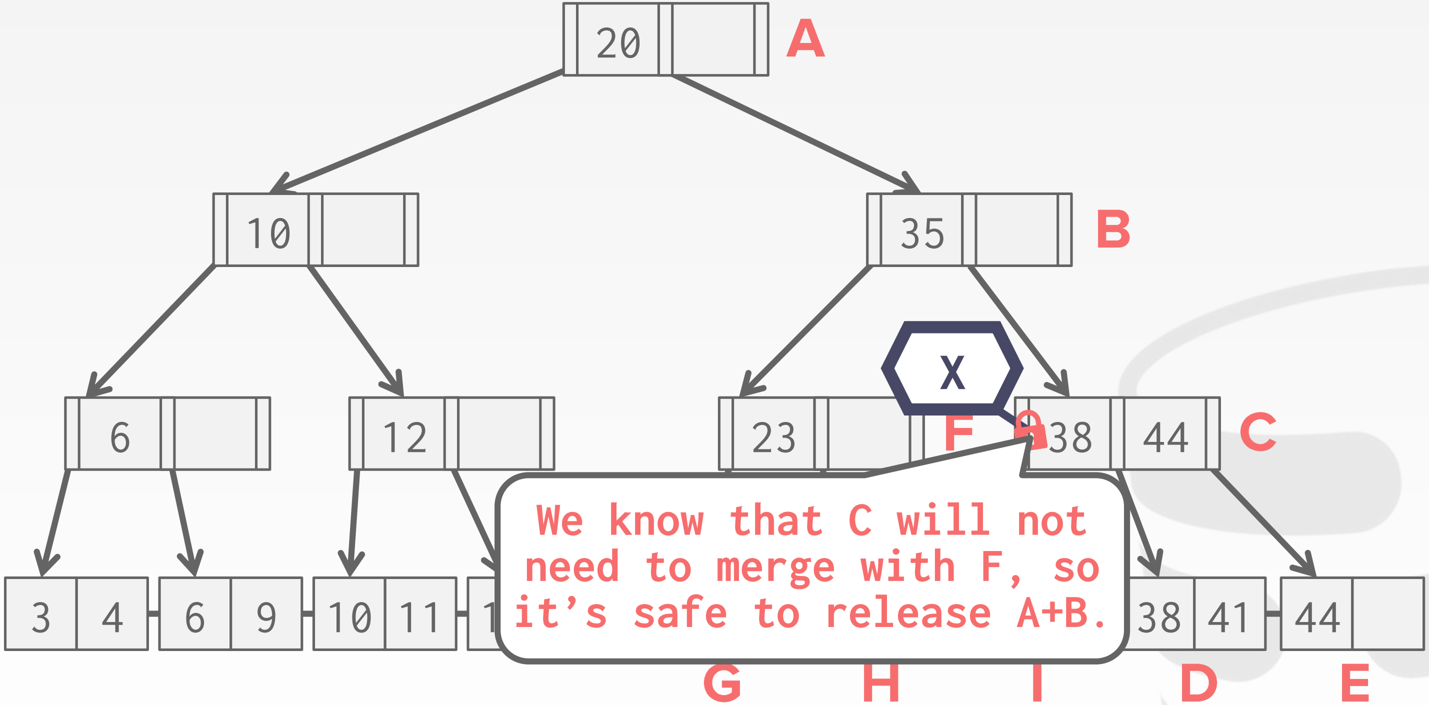
We may need to coalesce B, so we can't release the lock on A.

EXAMPLE #2 – DELETE 38



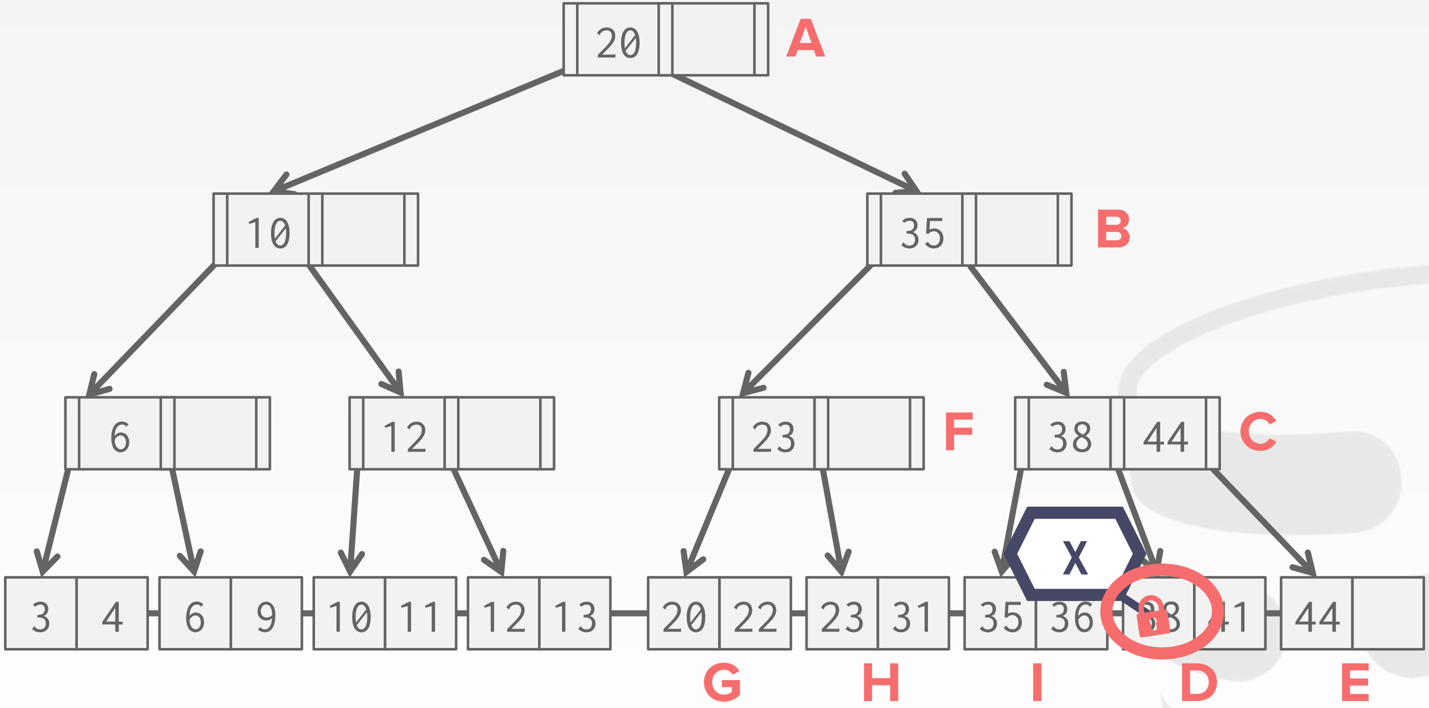
We know that C will not need to merge with F, so it's safe to release A+B.

EXAMPLE #2 – DELETE 38

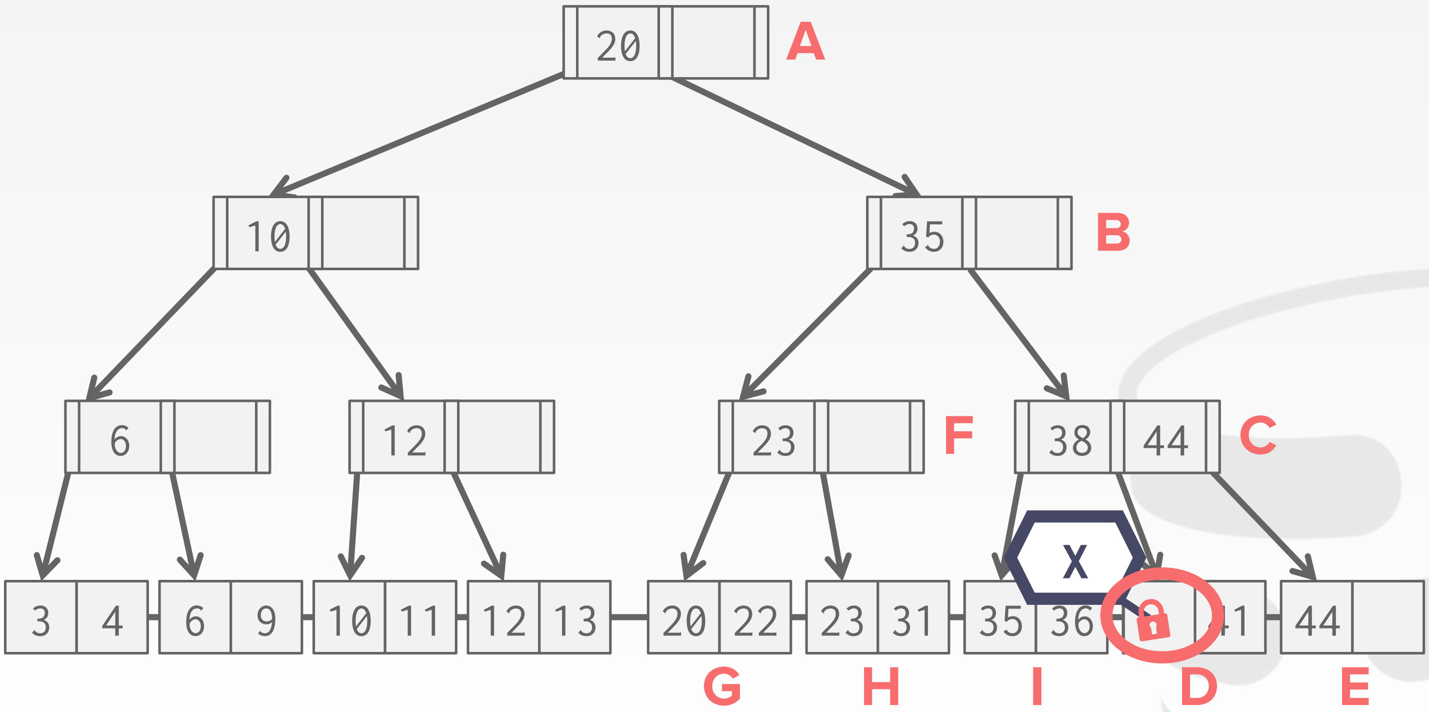


We know that C will not need to merge with F, so it's safe to release A+B.

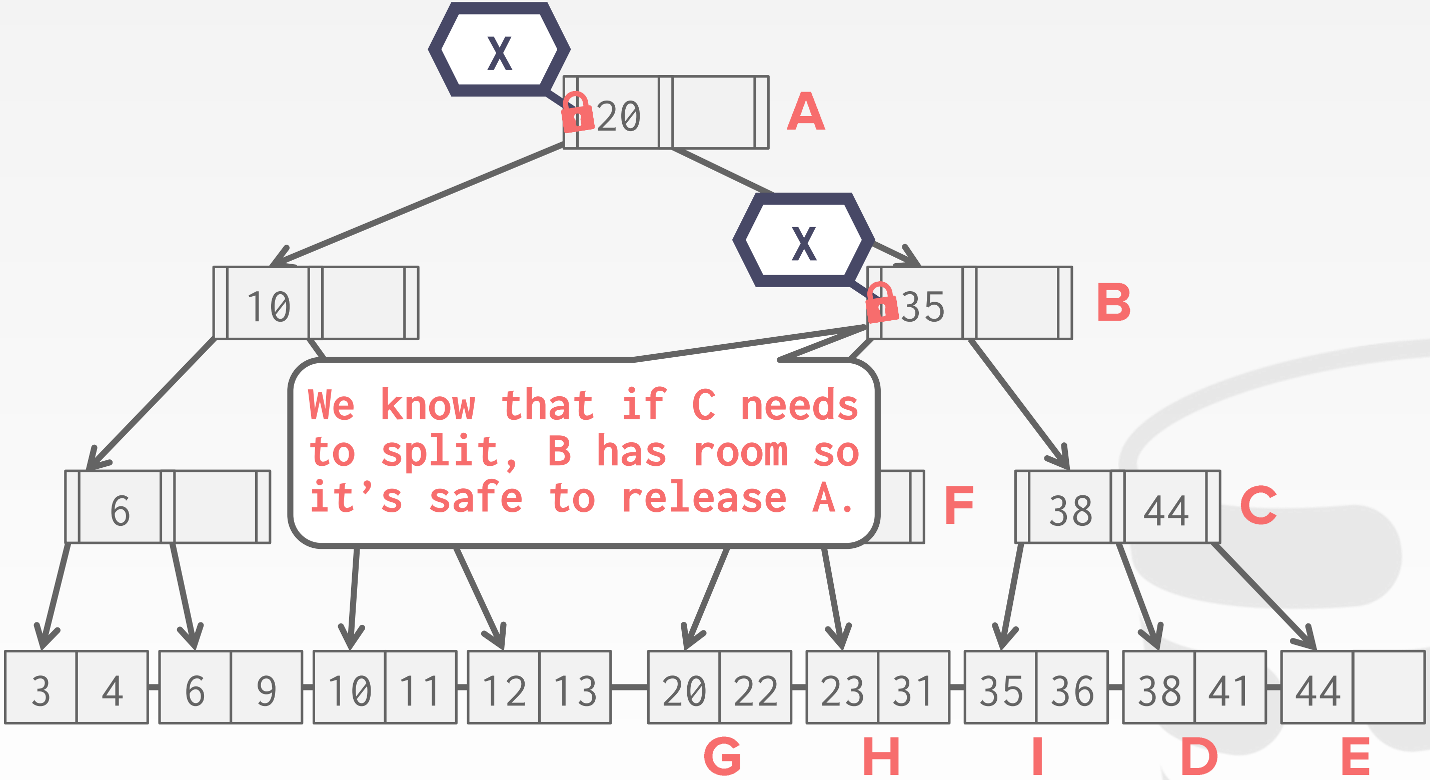
EXAMPLE #2 – DELETE 38



EXAMPLE #2 – DELETE 38

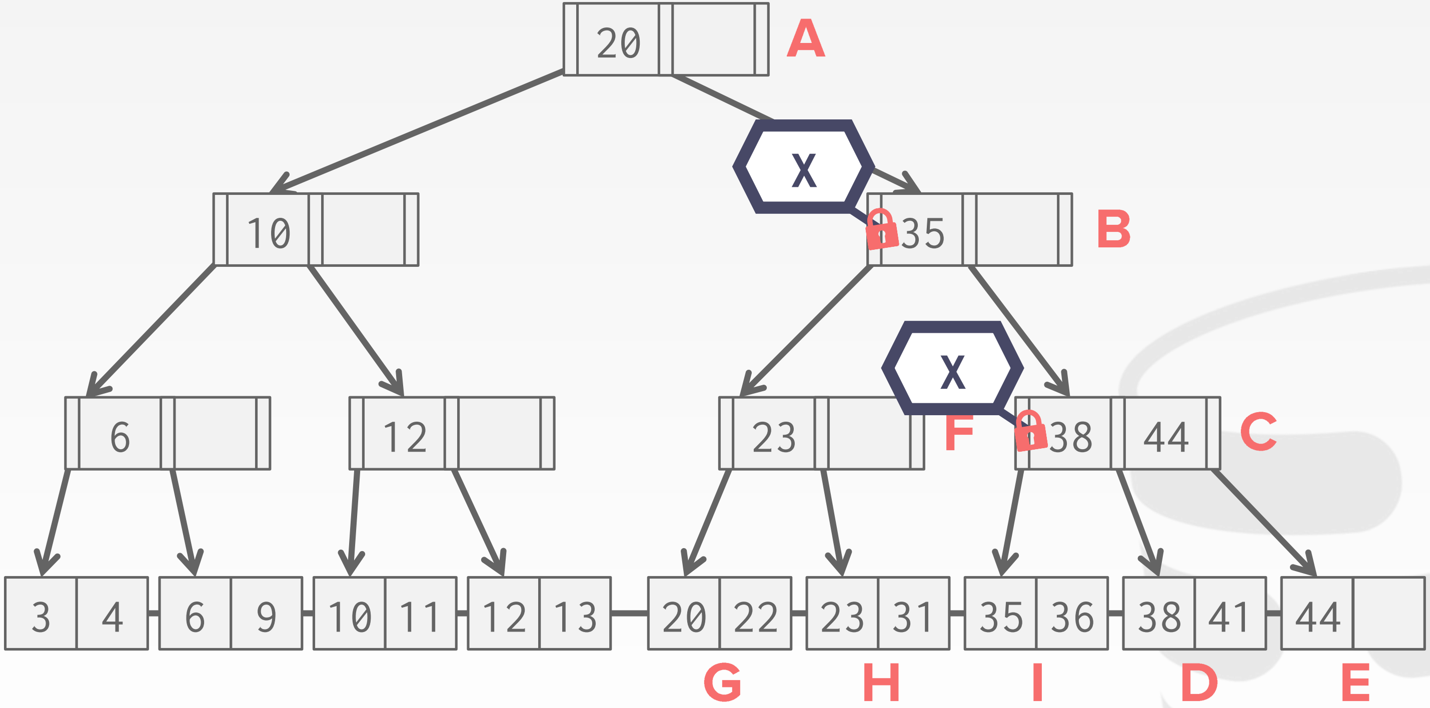


EXAMPLE #3 – INSERT 45

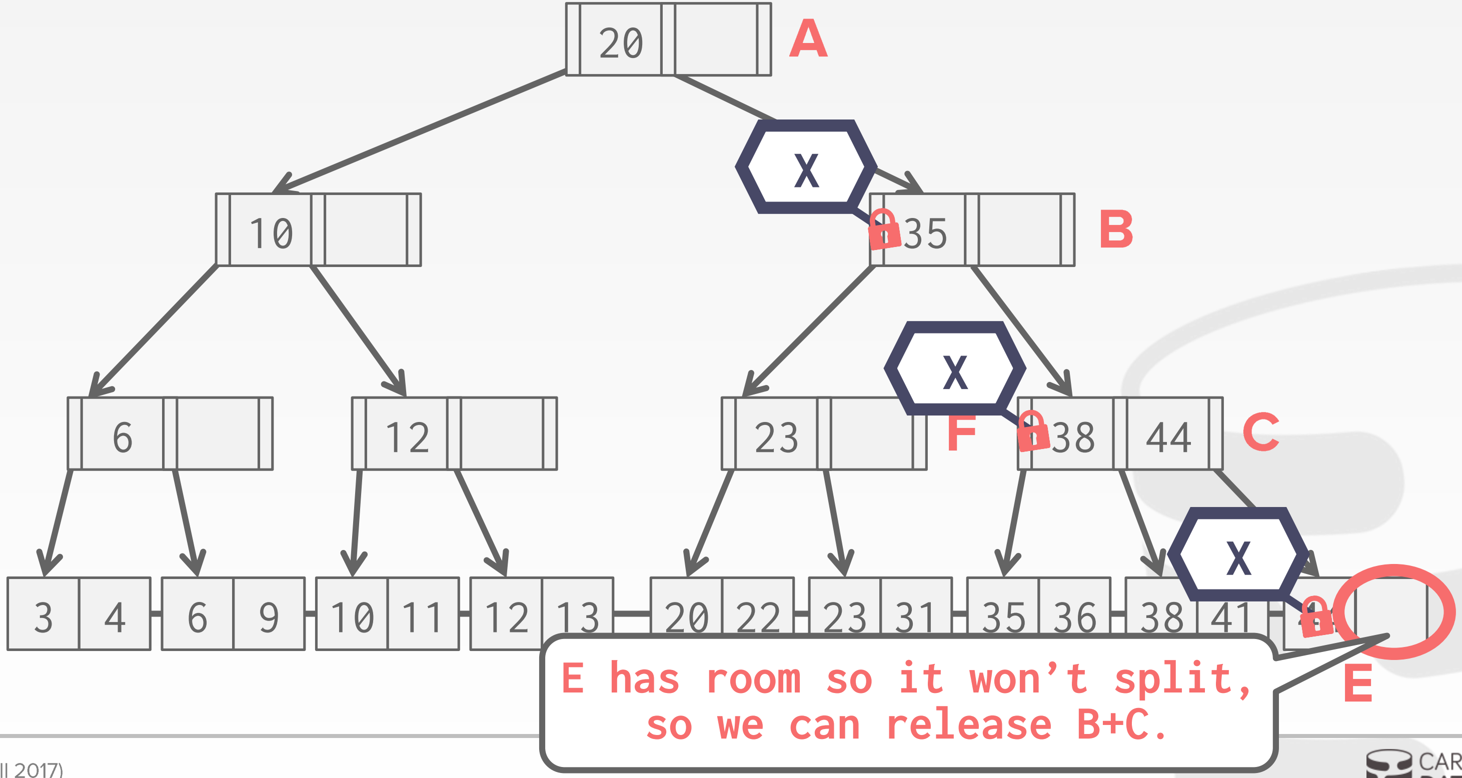


We know that if C needs to split, B has room so it's safe to release A.

EXAMPLE #3 – INSERT 45

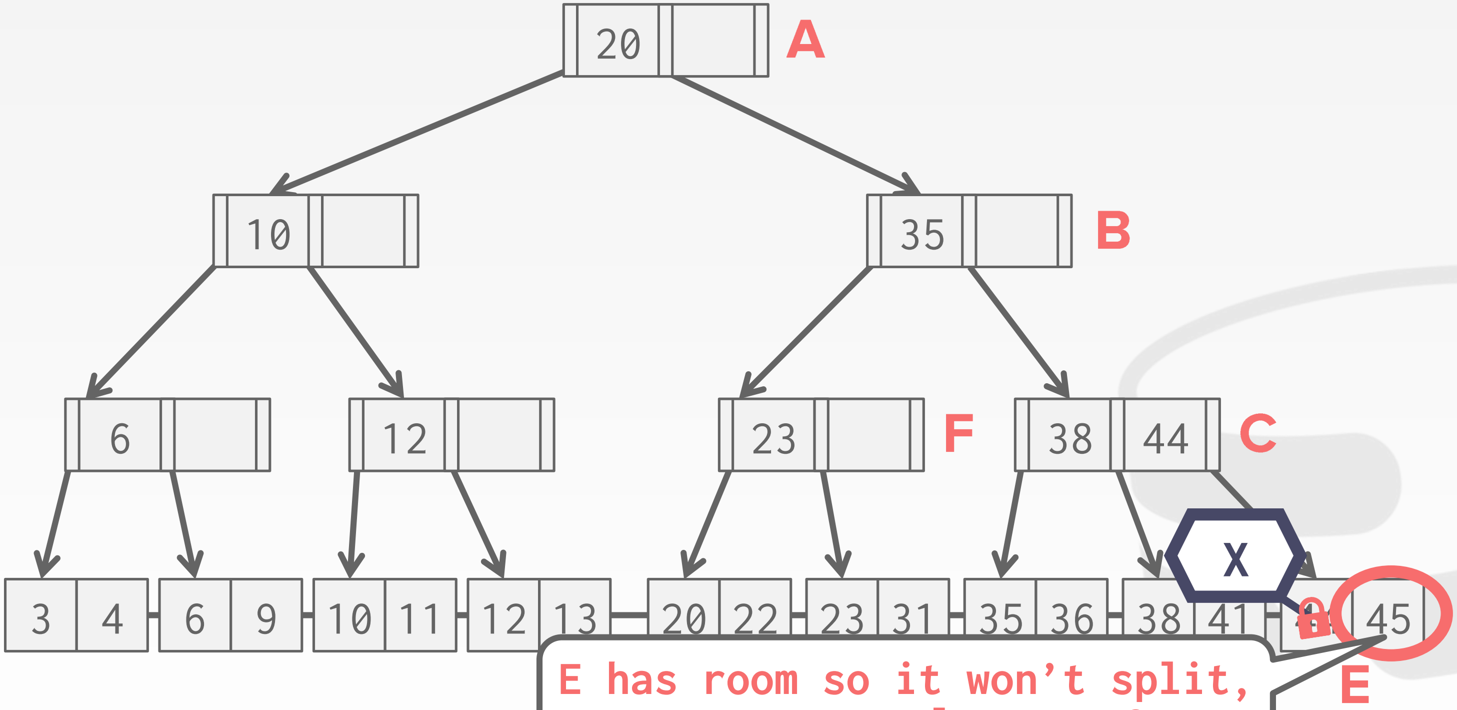


EXAMPLE #3 – INSERT 45



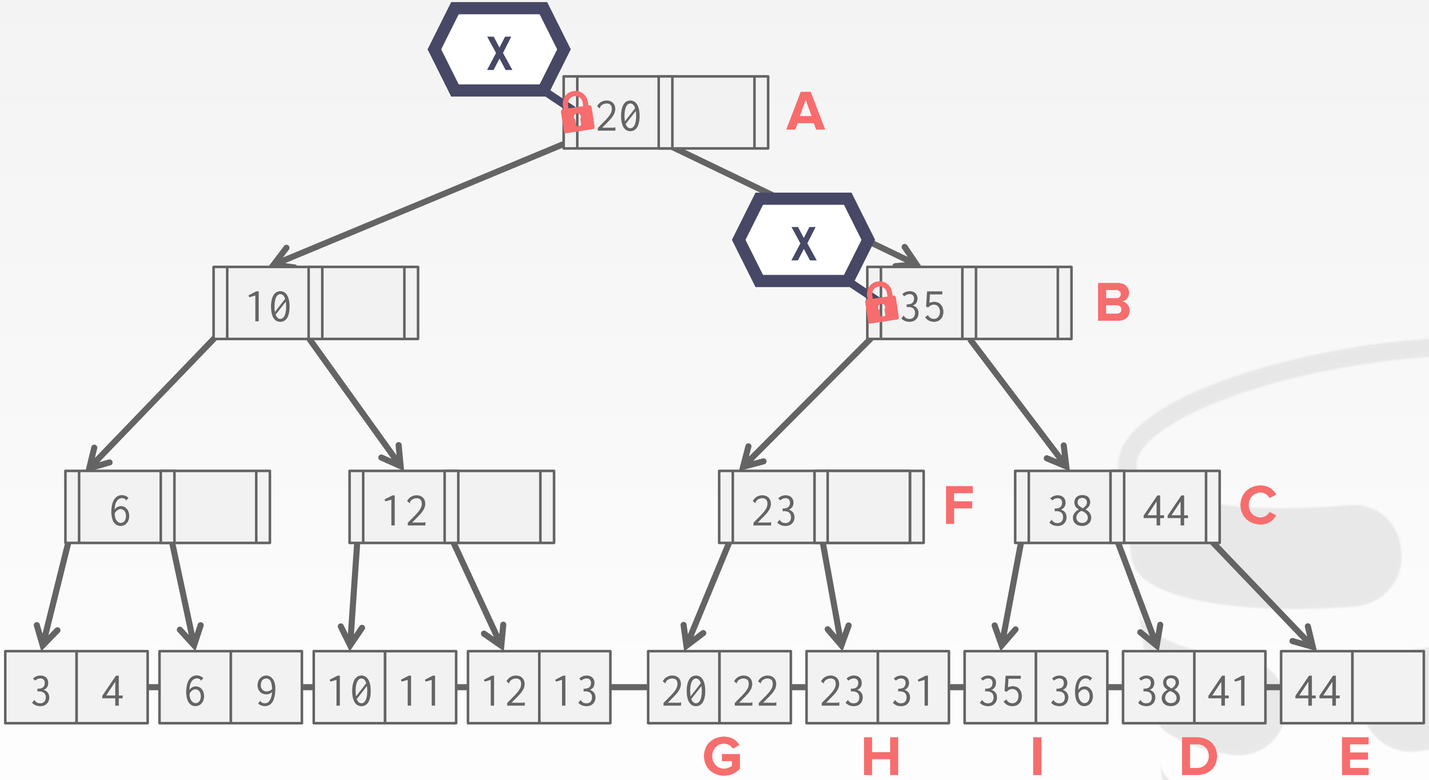
E has room so it won't split,
so we can release B+C.

EXAMPLE #3 – INSERT 45

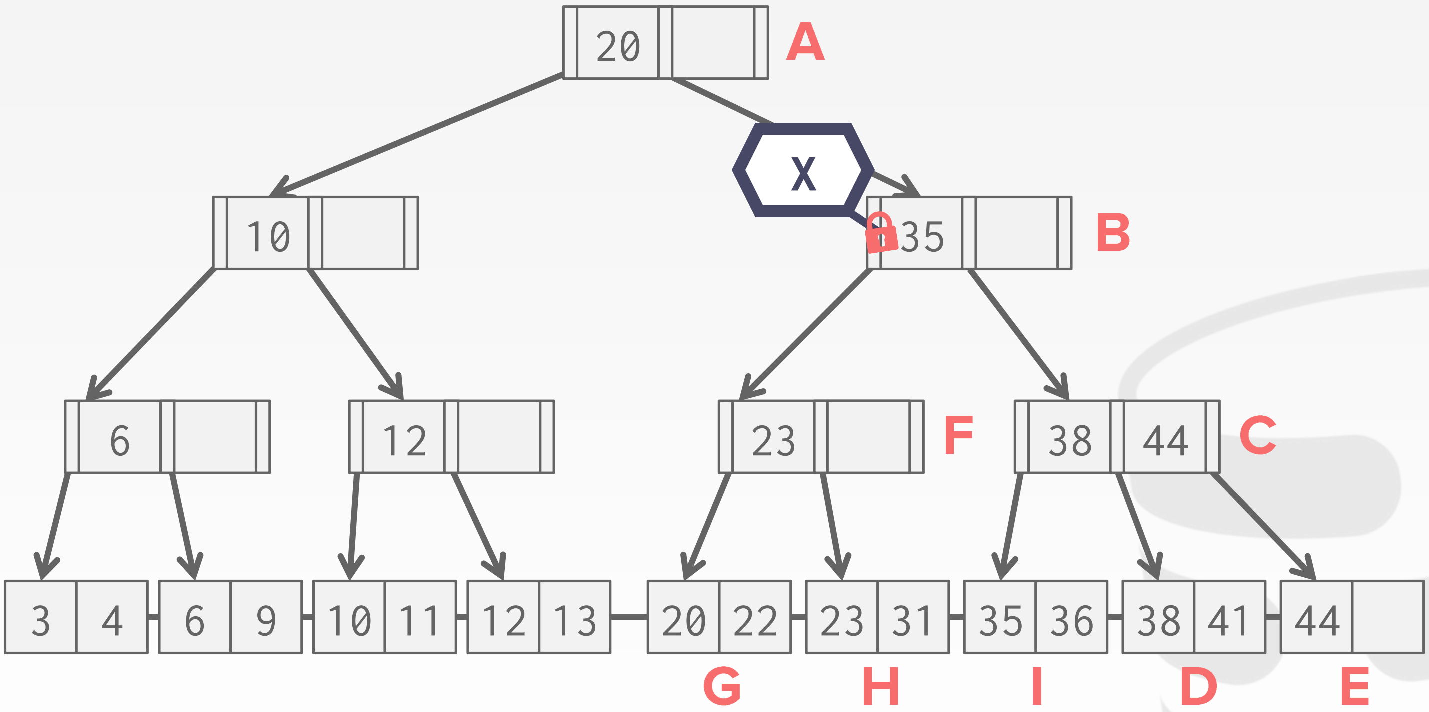


E has room so it won't split,
so we can release B+C.

EXAMPLE #4 – INSERT 25

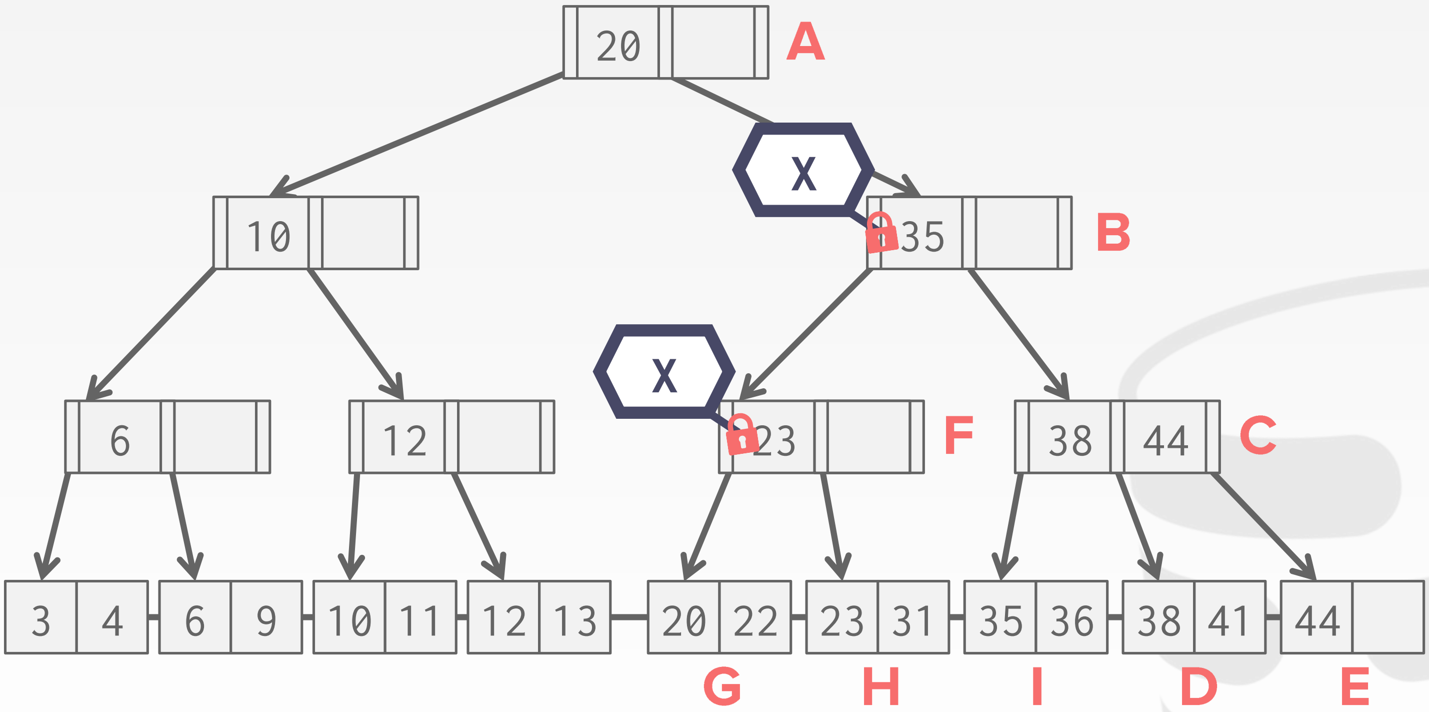


EXAMPLE #4 – INSERT 25

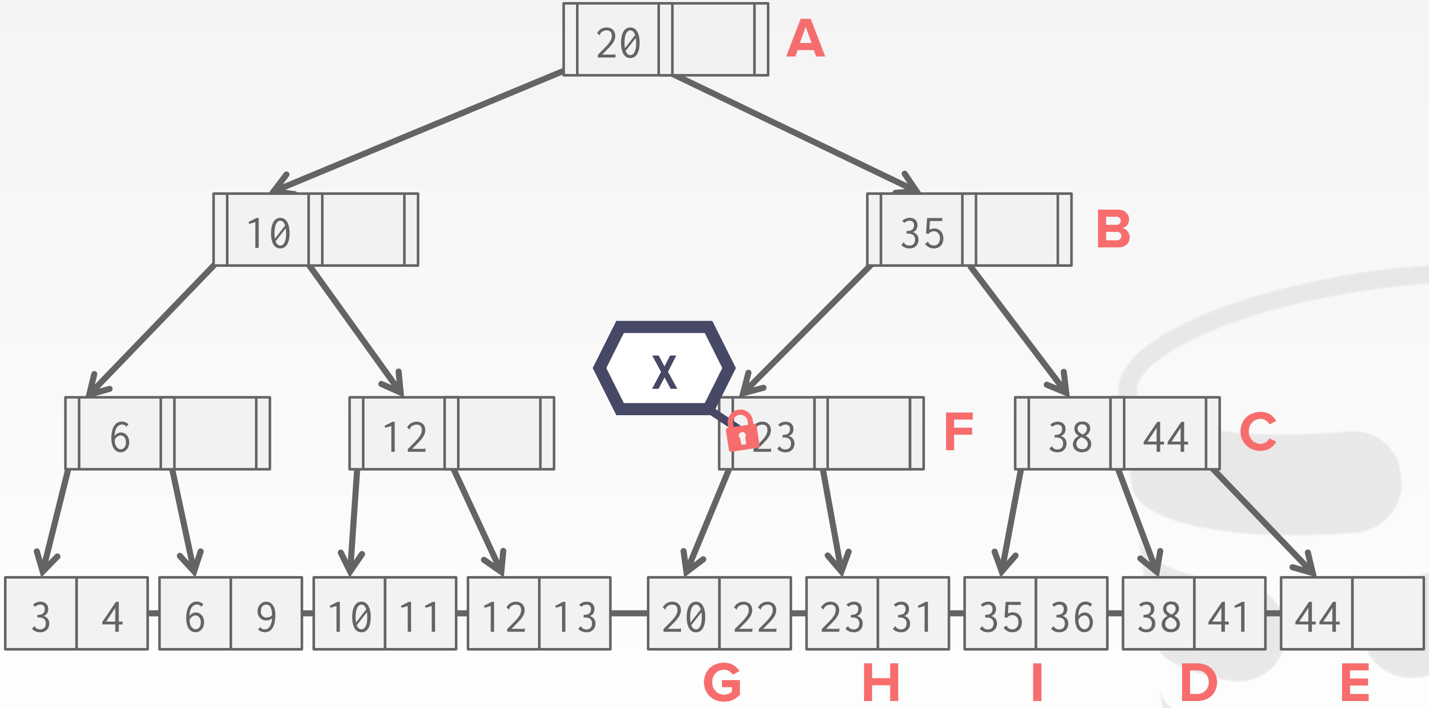


CARNEGIE MELLON
DATABASE GROUP

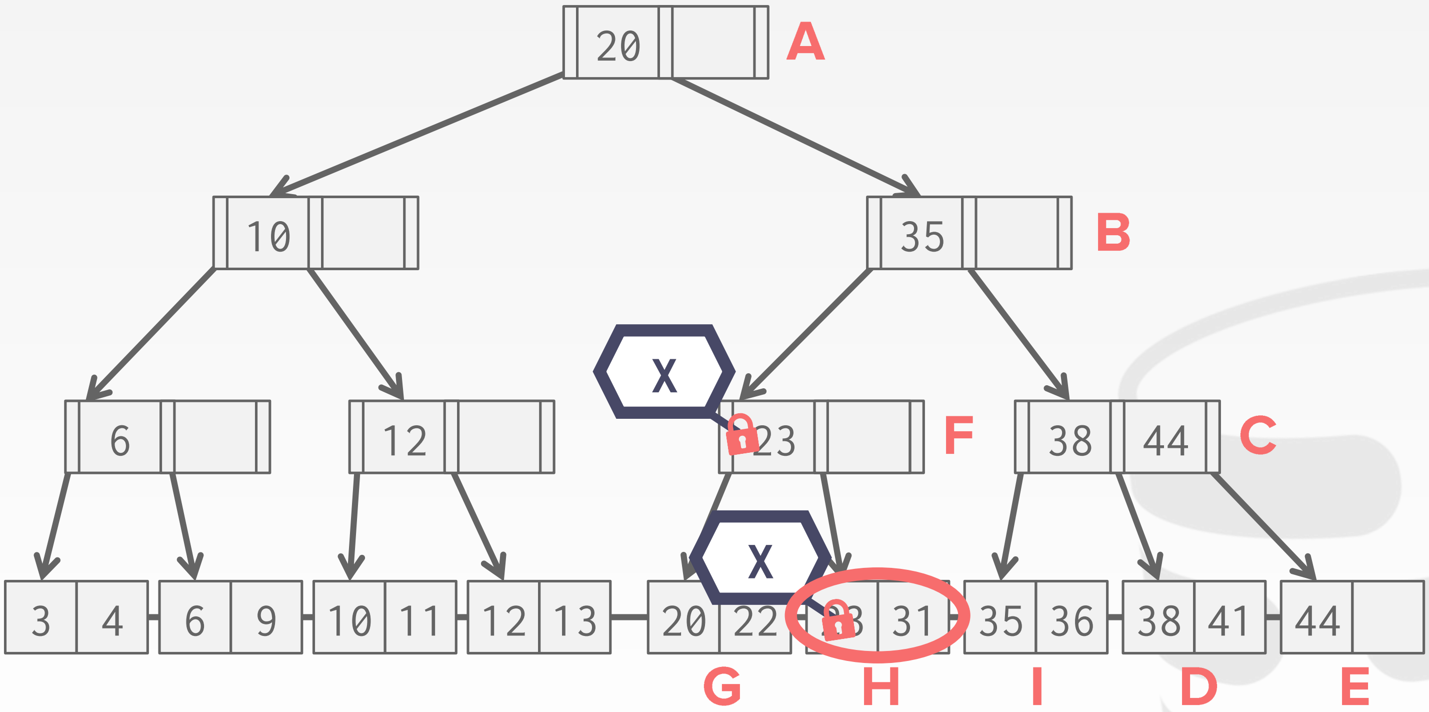
EXAMPLE #4 – INSERT 25



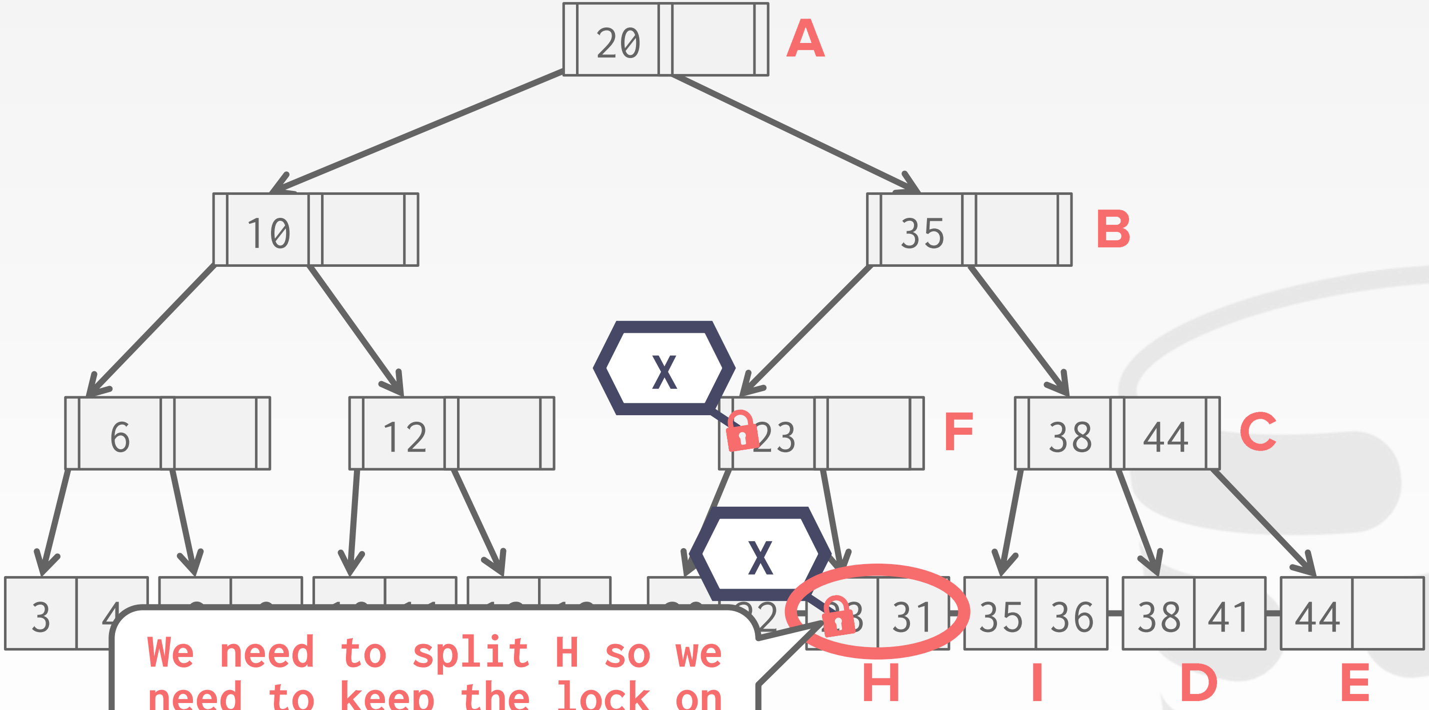
EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25

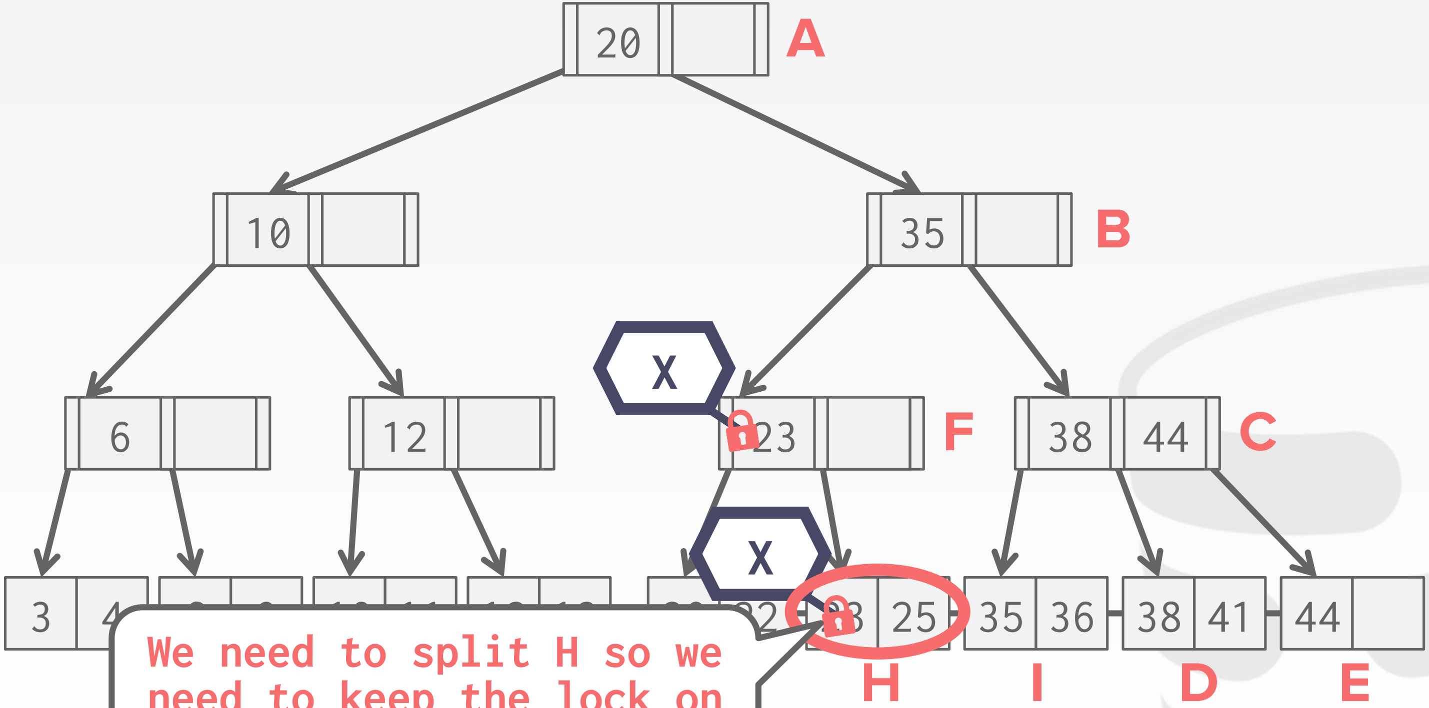


EXAMPLE #4 – INSERT 25



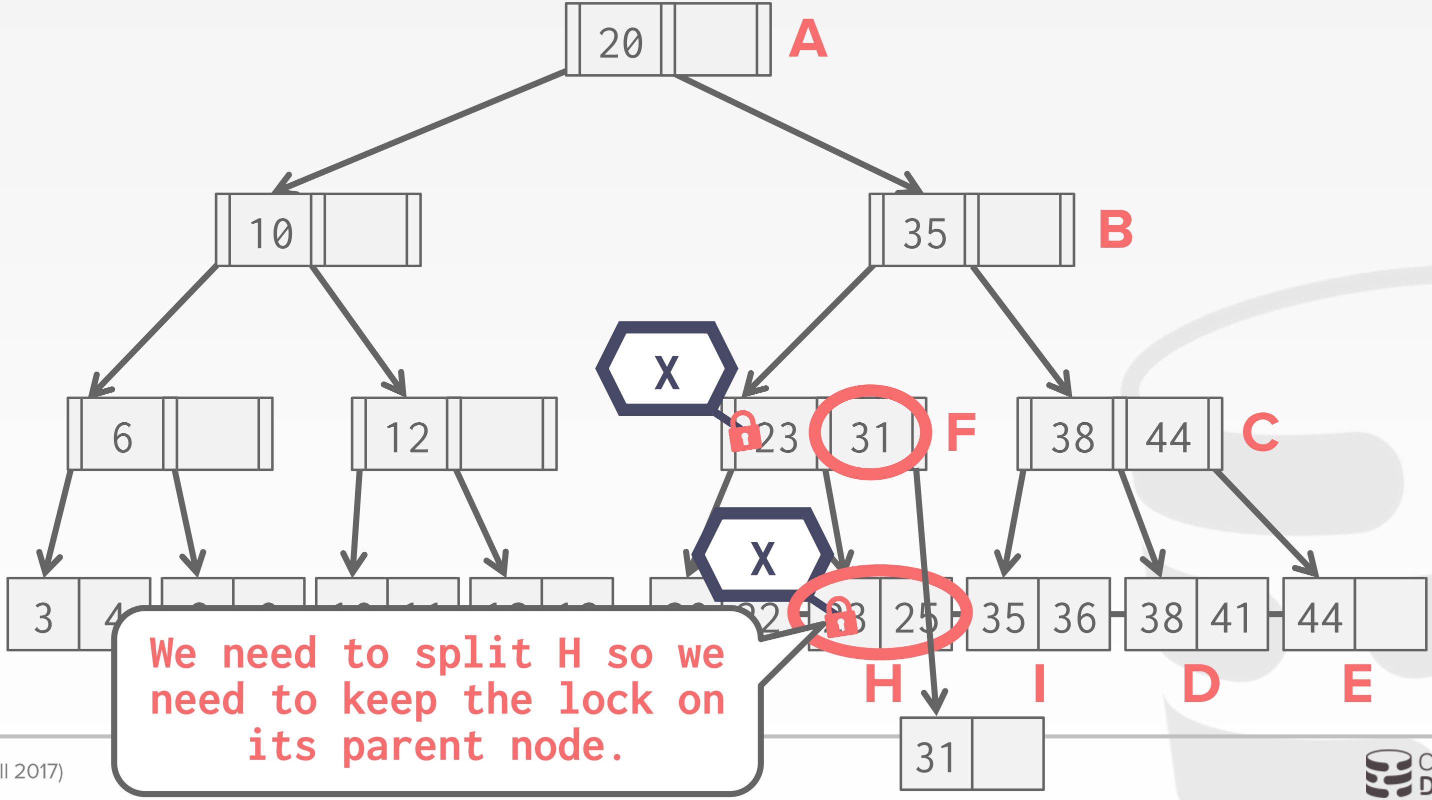
We need to split H so we need to keep the lock on its parent node.

EXAMPLE #4 – INSERT 25



We need to split H so we need to keep the lock on its parent node.

EXAMPLE #4 – INSERT 25

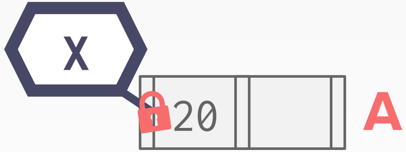


We need to split H so we need to keep the lock on its parent node.

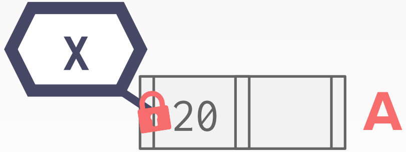
OBSERVATION

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

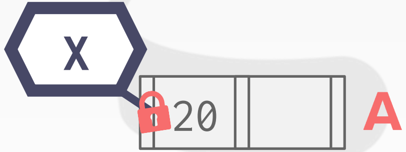
Delete 38



Insert 45



Insert 25



OBSERVATION

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

Locking the root every time becomes a bottleneck with higher concurrency.

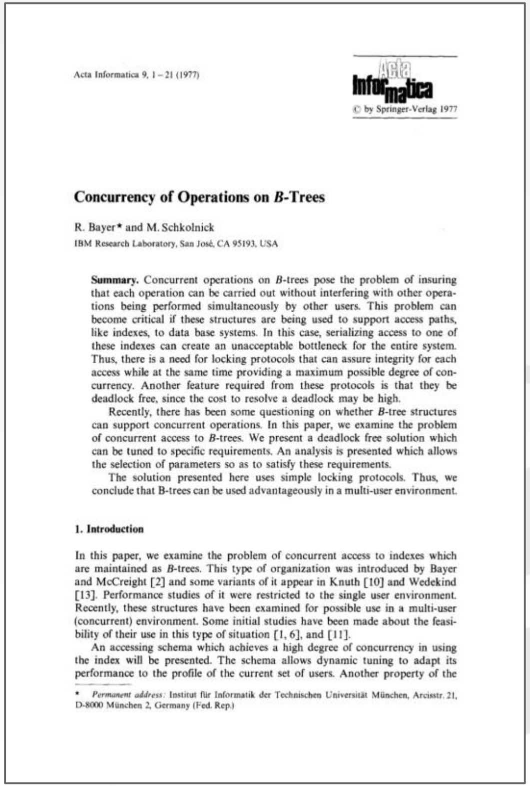
Can we do better?

BETTER TREE LOCKING ALGORITHM

Assume that the leaf is safe, and use S-locks & crabbing to reach it, and verify.

If leaf is not safe, then do previous algorithm.

Rudolf Bayer, Mario Schkolnick: Concurrency of Operations on B-Trees. Acta Inf. 9: 1-21 (1977)



Acta Informatica 9, 1 – 21 (1977)

Acta Informatica

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

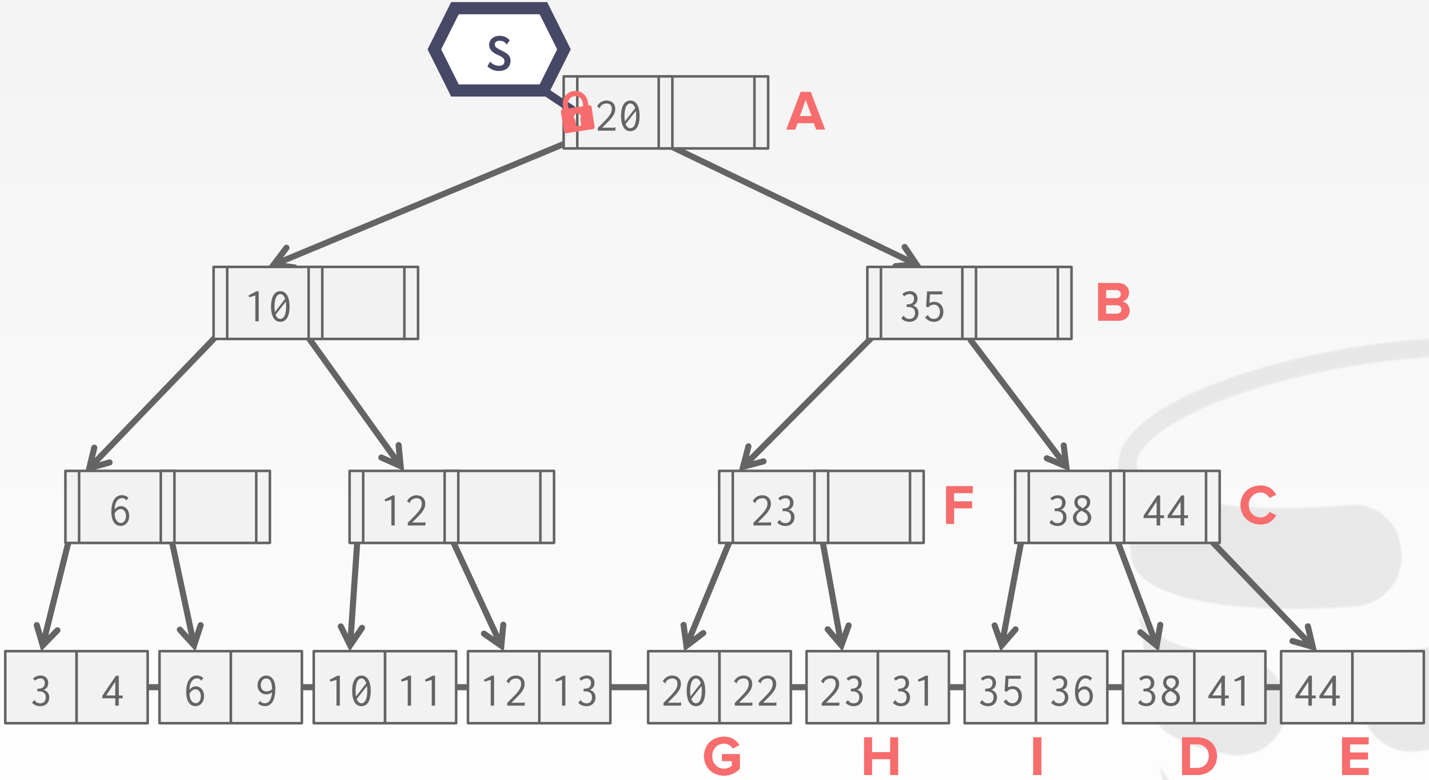
In this paper, we examine the problem of concurrent access to indexes which are maintained as *B*-trees. This type of organization was introduced by Bayer and McCreight [2] and some variants of it appear in Knuth [10] and Wedekind [13]. Performance studies of it were restricted to the single user environment. 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

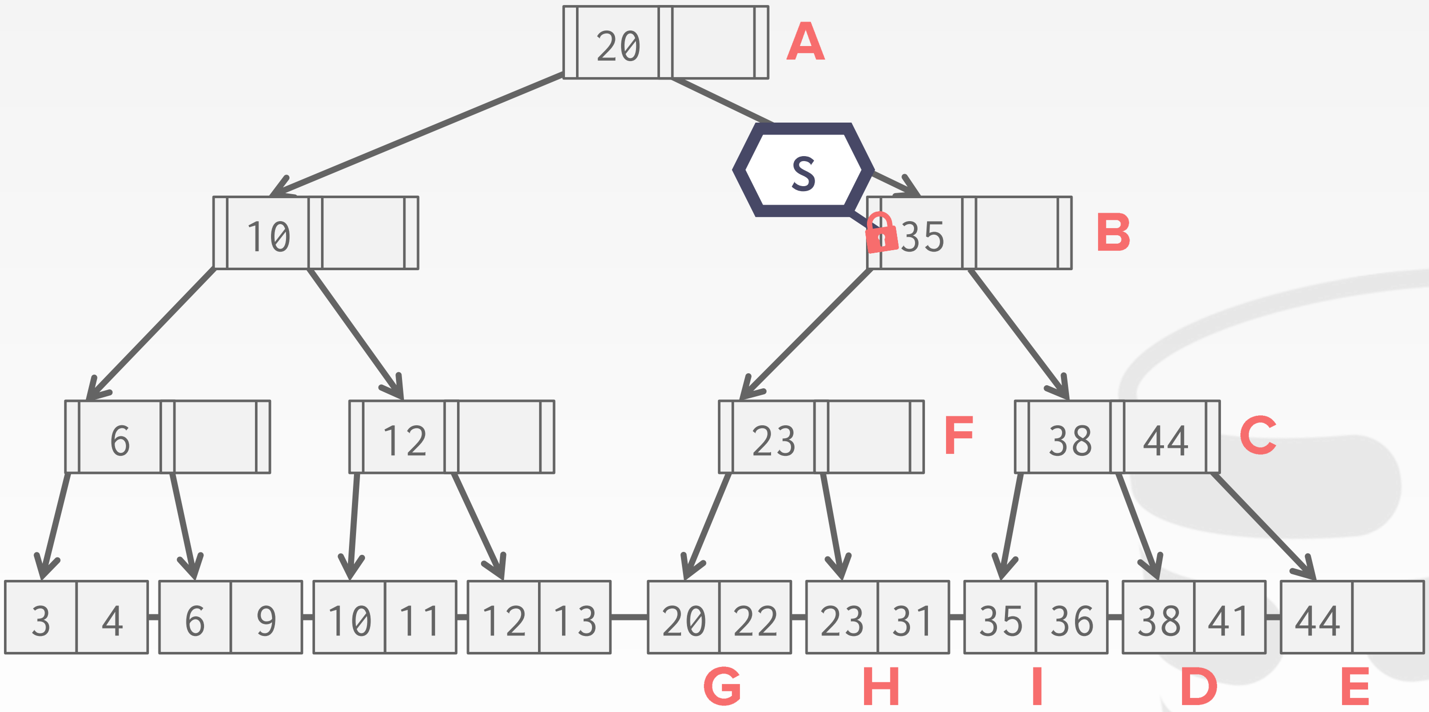
---

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

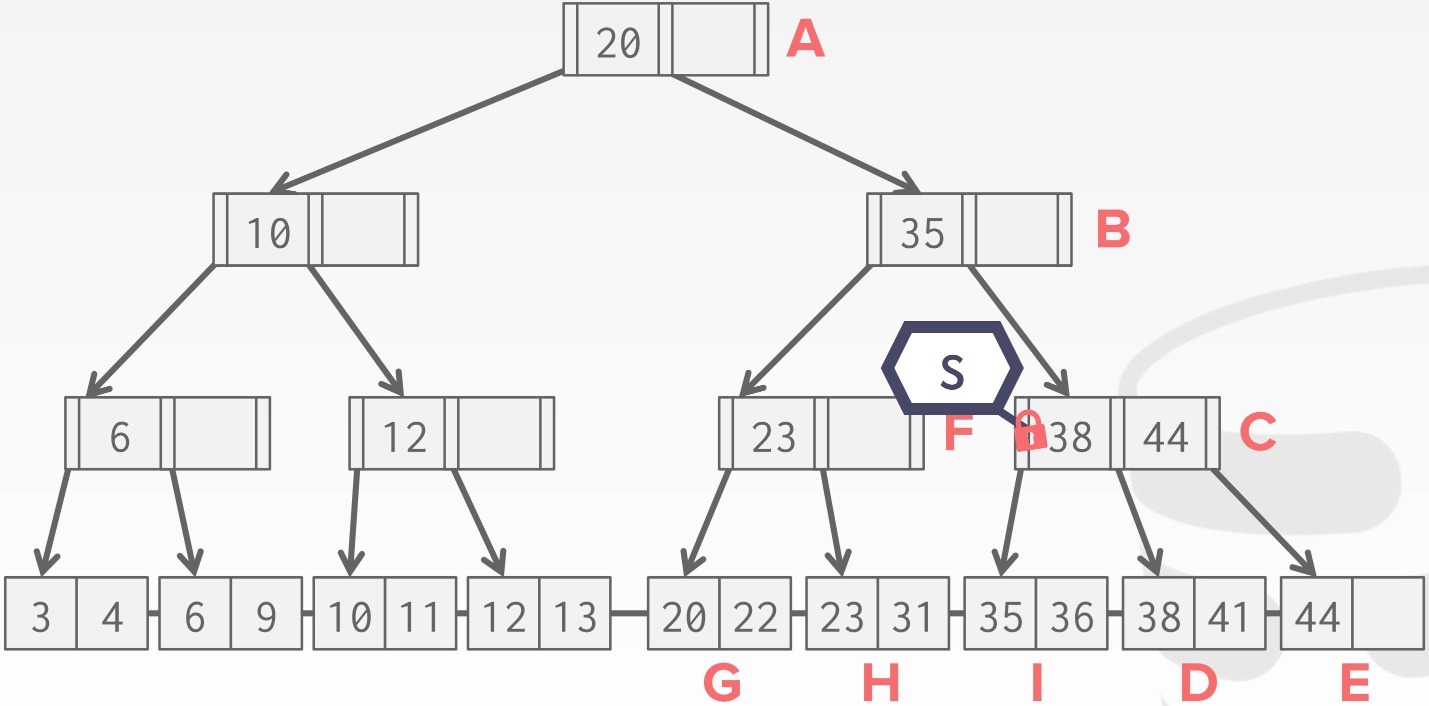
EXAMPLE #2 – DELETE 38



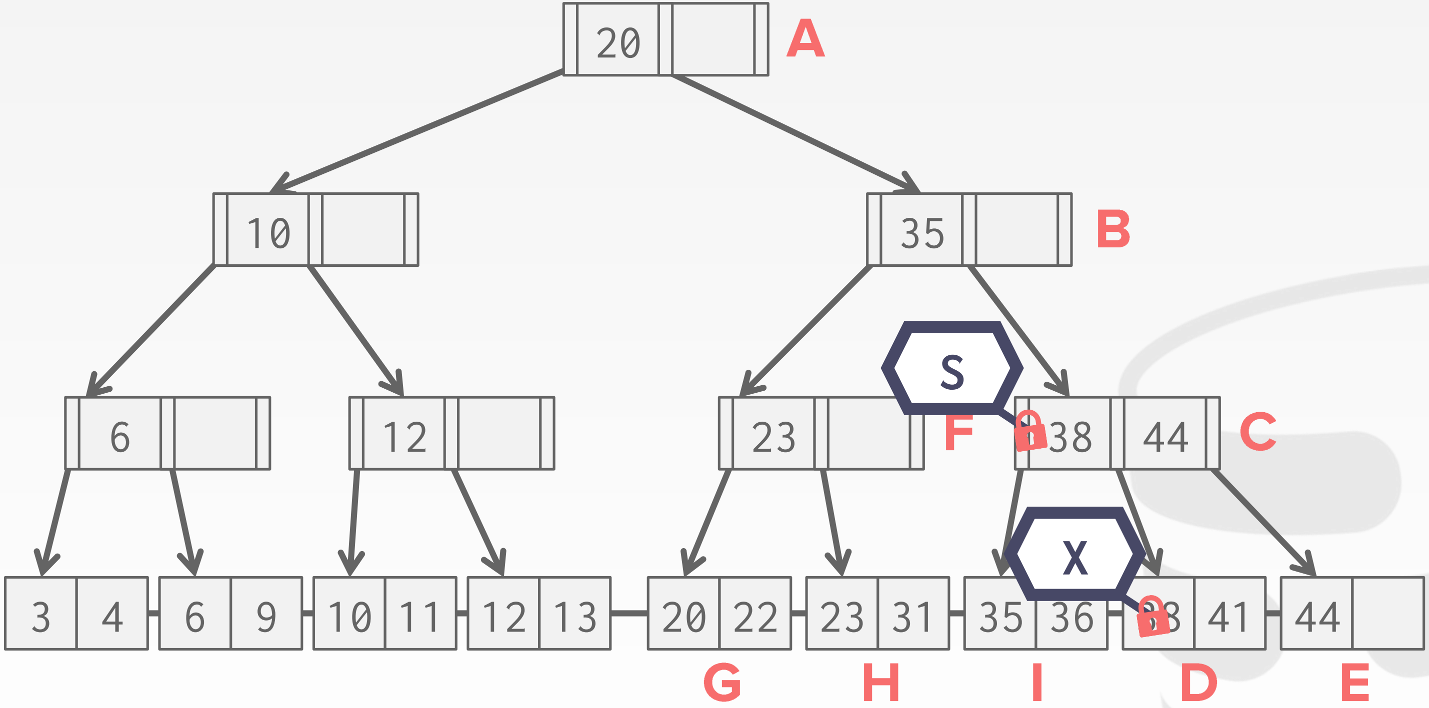
EXAMPLE #2 – DELETE 38



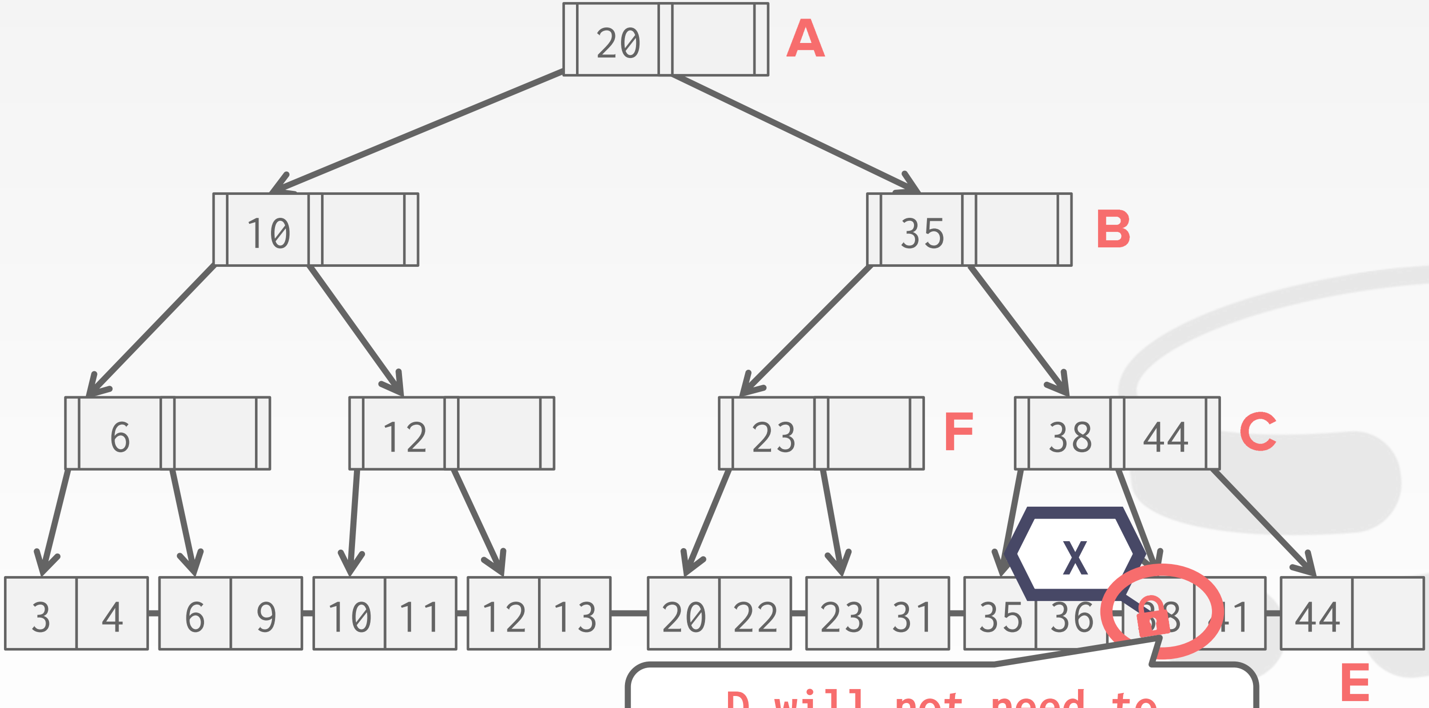
EXAMPLE #2 – DELETE 38



EXAMPLE #2 – DELETE 38

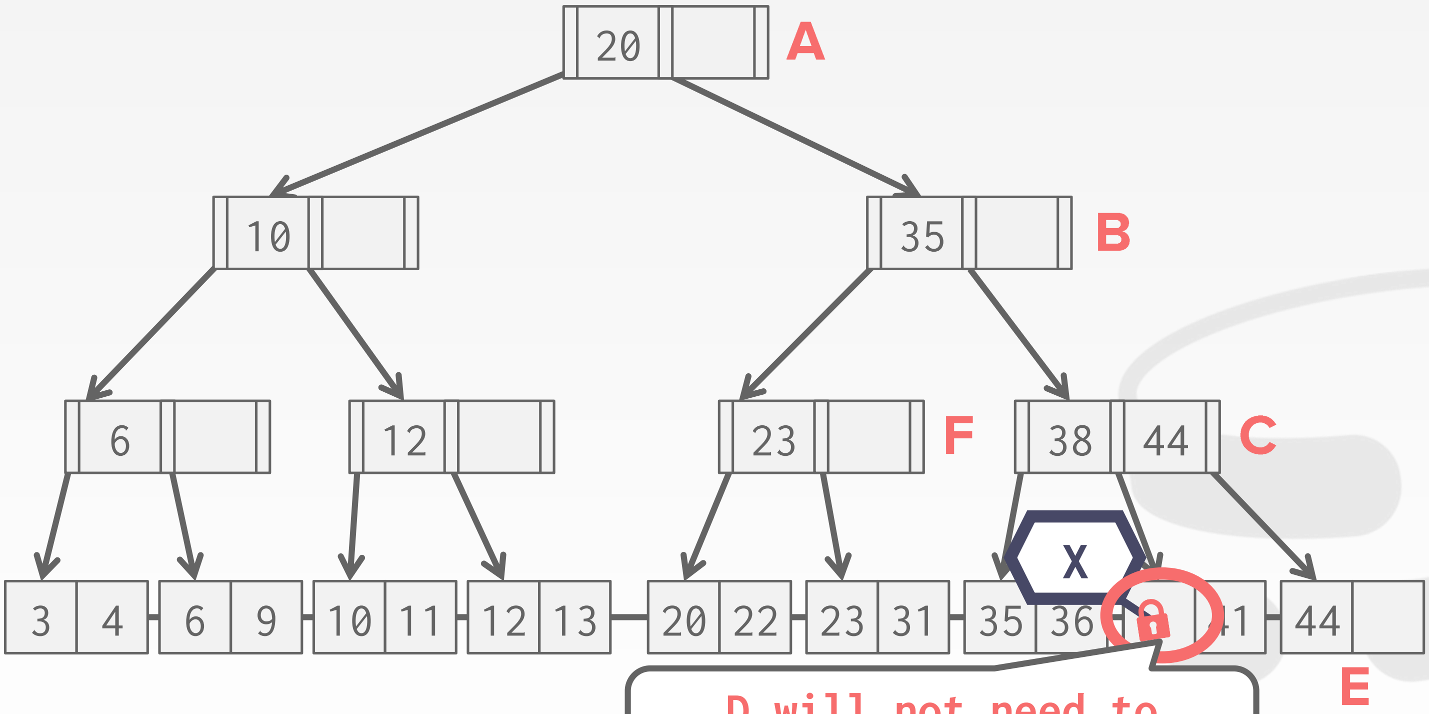


EXAMPLE #2 – DELETE 38



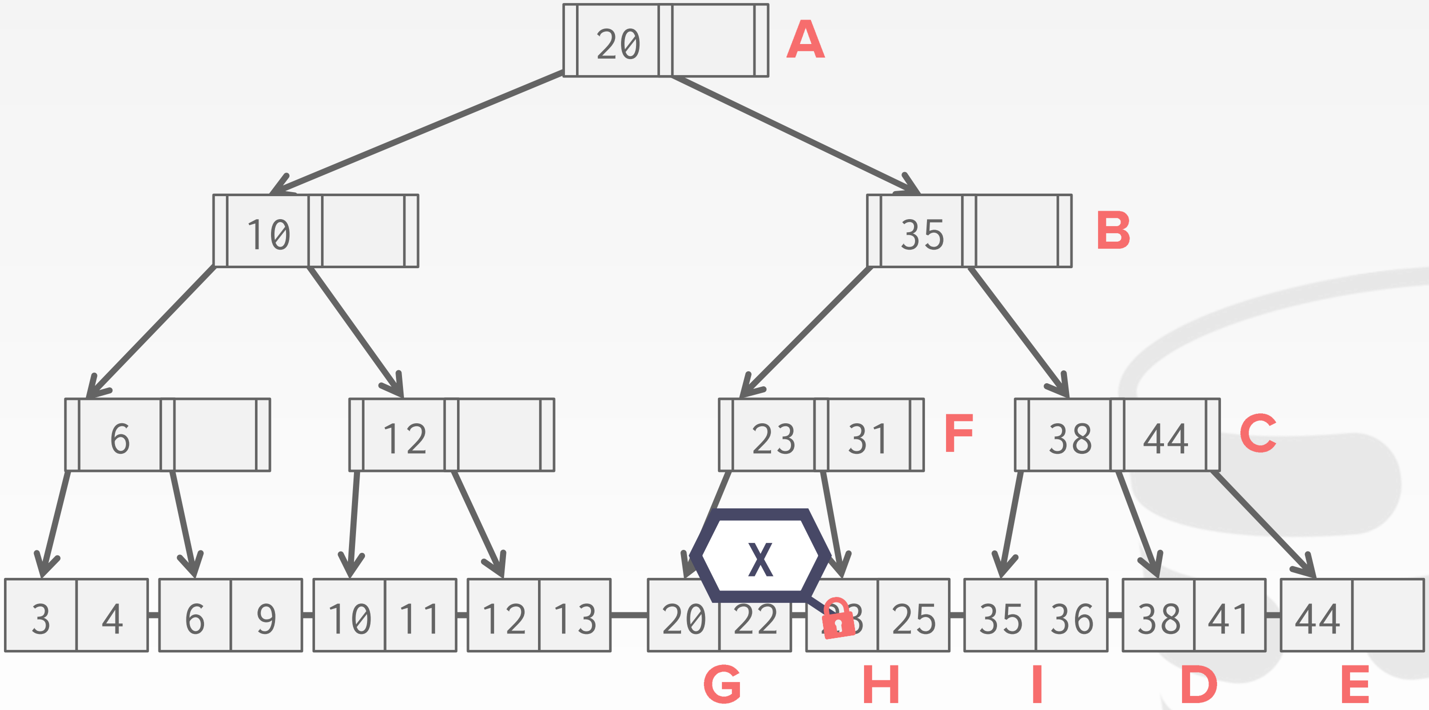
D will not need to coalesce, so we're safe!

EXAMPLE #2 – DELETE 38

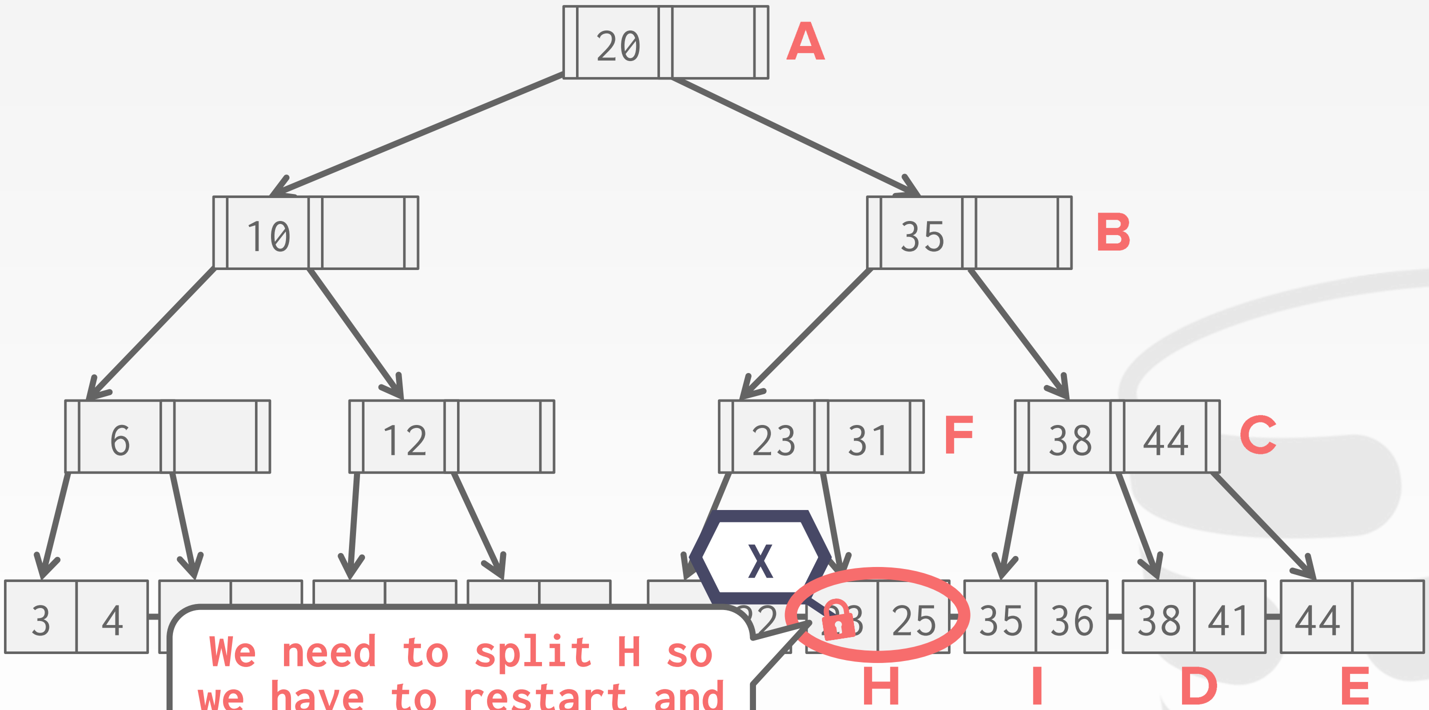


D will not need to coalesce, so we're safe!

EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25



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

BETTER TREE LOCKING ALGORITHM

Search: Same as before.

Insert/Delete:

→ Set locks as if for search, get to leaf, and set X lock on leaf.

→ If leaf is not safe, release all locks, and restart txn using previous Insert/Delete protocol.

Gambles that only leaf node will be modified; if not, S locks set on the first pass to leaf are wasteful.

ADDITIONAL POINTS

Which order to release locks in multiple-granularity locking?

From the bottom up

Which order to release latches in B+Tree latching?

As early as possible to maximize concurrency.

CONCLUSION

Indexes make concurrency control hard because it's a essentially a second copy of the data.

Most applications do not execute with `SERIALIZABLE` isolation.

NEXT CLASS

More Concurrency Control!!!
Timestamp Ordering!!!