Lecture 16: Index Concurrency Control

Administrivia

- Mid-term grades
- Assignment 3 and Sheet 3 due on Nov 1

Today's Agenda

Recap

Index Data Structures

- List of Data Structures: Hash Tables, B+Trees, Radix Trees
- Most DBMSs automatically create an index to enforce **integrity constraints**.
- B+Trees are the way to go for indexing data.

Observation

- We assumed that all 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 and hide disk I/O stalls.

Concurrency Control

- A **concurrency control protocol** is the method that the DBMS uses to ensure "correct" results for concurrent operations on a shared object.
- A protocol's correctness criteria can vary:
    - **Logical Correctness:** Am I reading the data that I am supposed to read?
    - **Physical Correctness:** Is the internal representation of the object sound?

Latches Overview

Locks vs. Latches

- **Locks**
  - Protects the database's **logical contents** from other txns.
  - Held for the duration of the transaction.
  - Need to be able to rollback changes.
- **Latches**
  - Protects the critical sections of the DBMS's internal **physical data structures** from other threads.
  - Held for the duration of the operation.
  - 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 (*a.k.a.*, Shared, Exclusive)
Deadlock	Detection & Resolution	Avoidance
. . . by. . .	Waits-for, Timeout, Aborts	Coding Discipline
Kept in. . .	Lock Manager	Protected Data Structure

Reference

Latch Modes

- **Read Mode**
  - Multiple threads can read the same object at the same time.
  - A thread can acquire the read latch if another thread has it in read mode.
- **Write Mode**
  - Only one thread can access the object.
  - A thread cannot acquire a write latch if another thread holds the latch in any mode.

	Read	Write
Read	✓	X
Write	X	X

Latch Implementations

- Blocking OS Mutex
- Test-and-Set Spin Latch
- Reader-Writer Latch

Latch Implementations

- **Approach 1: Blocking OS Mutex**
  - Simple to use
  - Non-scalable (about 25 ns per lock/unlock invocation)
  - Example: std::mutex

```
std::mutex m;

m.lock();
// Do something special...
m.unlock();
```

Latch Implementations

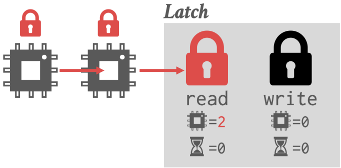
- **Approach 2: Test-and-Set Spin Latch (TAS)**
  - Very efficient (single instruction to latch/unlatch)
  - Non-scalable, not cache friendly
  - Example: std::atomic<T>
  - Unlike OS mutex, spin latches do **not** suspend thread execution
  - Atomic operations are faster if contention between threads is sufficiently **low**

```
std::atomic_flag latch; // atomic of boolean type (lock-free)

while (latch.test_and_set(...)) {
        // Retry? Yield? Abort?
}
```

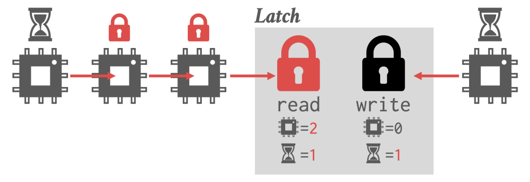
Latch Implementations

- **Approach 3: Reader-Writer Latch**
  - Allows for concurrent readers
  - Must manage read/write queues to avoid starvation
  - Can be implemented on top of spinlocks



Latch Implementations

- **Approach 3: Reader-Writer Latch**
  - Allows for concurrent readers
  - Must manage read/write queues to avoid starvation
  - Can be implemented on top of spinlocks



Hash Table Latching

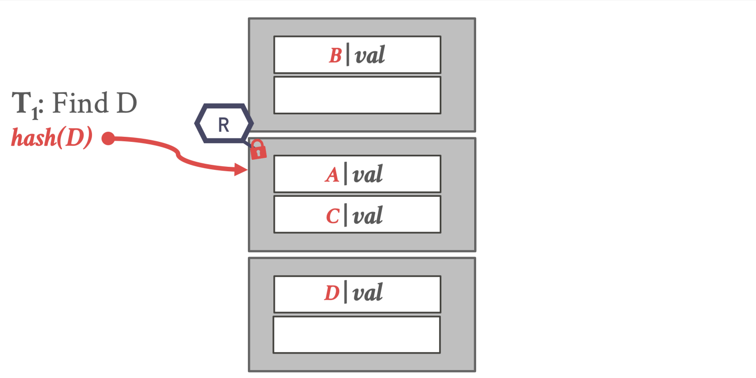
Hash Table Latching

- Easy to support concurrent access due to the limited ways in which threads access the data structure.
  - All threads move in the same direction and only access a **single page/slot at a time**.
  - Deadlocks are **not** possible.
- To resize the table, take a **global latch** on the entire table (*i.e.*, in the header page).

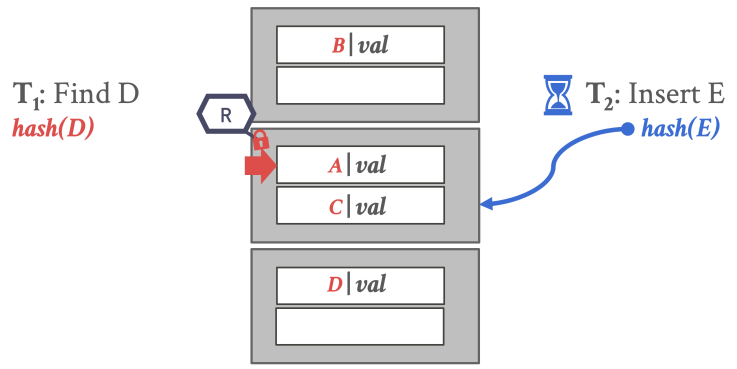
Hash Table Latching

- **Approach 1: Page Latches**
  - Each page has its own reader-write latch that protects its entire contents.
  - Threads acquire either a read or write latch before they access a page.
- **Approach 2: Slot Latches**
  - Each slot has its own latch.
  - Can use a single mode latch to reduce meta-data and computational overhead.

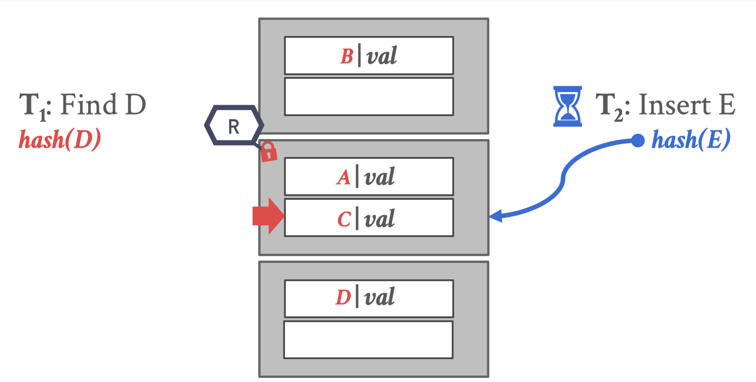
Hash Table - Page Laches



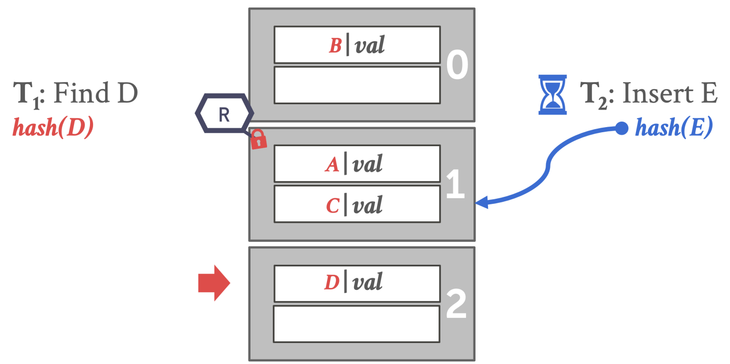
Hash Table - Page Laches



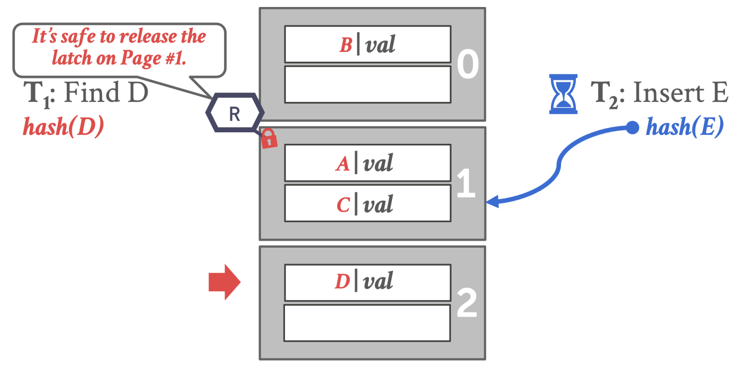
Hash Table - Page Laches



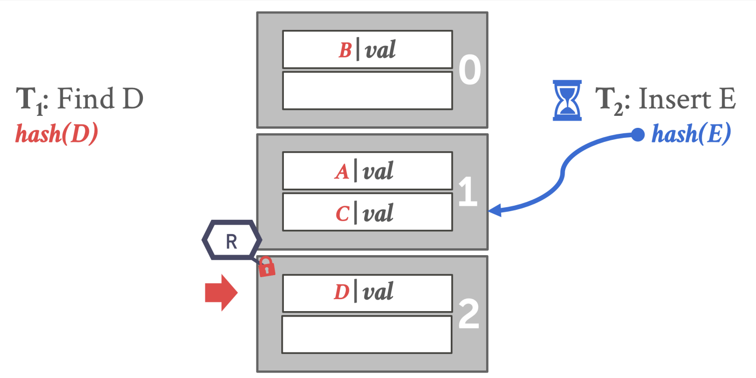
Hash Table - Page Laches



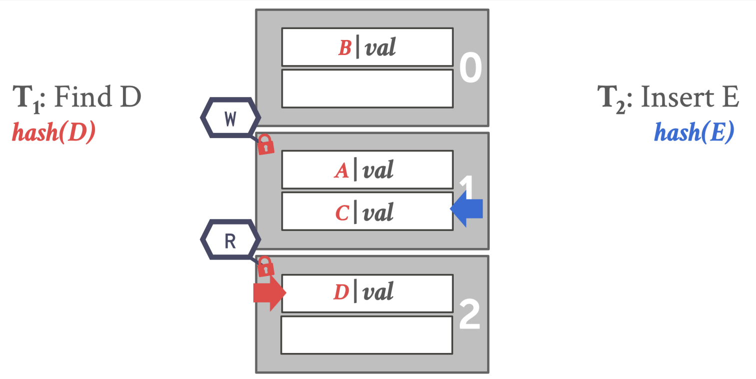
Hash Table - Page Laches



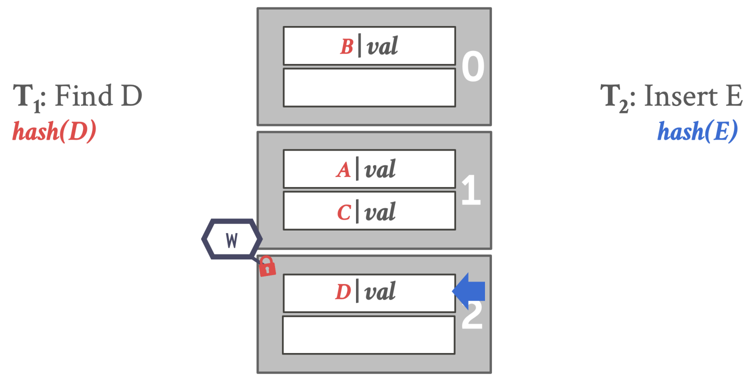
Hash Table - Page Laches



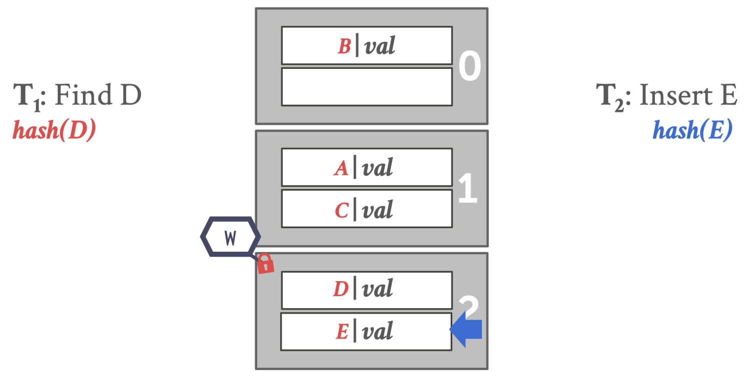
Hash Table - Page Laches



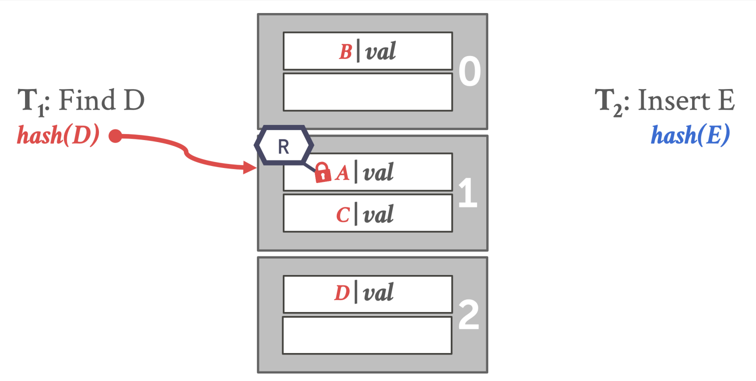
Hash Table - Page Laches



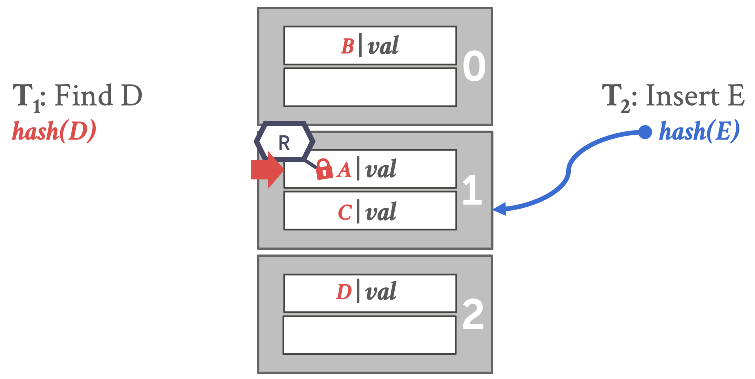
Hash Table - Page Laches



Hash Table - Slot Laches



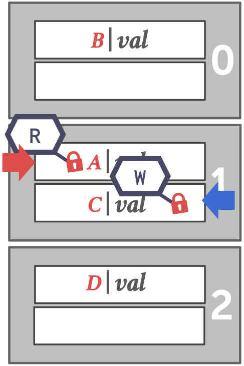
Hash Table - Slot Laches



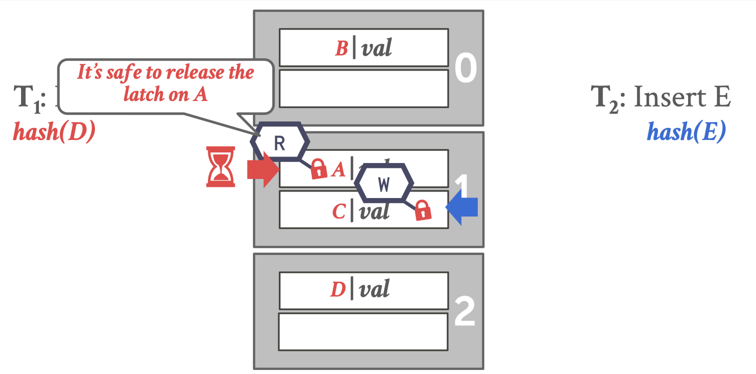
Hash Table - Slot Laches

T$_1$: Find D
hash(D)

T$_2$: Insert E
hash(E)



Hash Table - Slot Laches

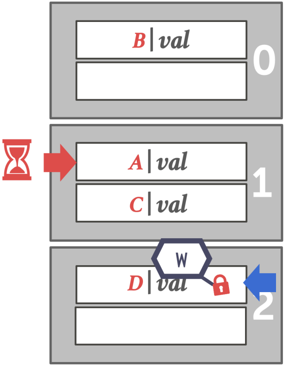


It's safe to release the latch on A

Hash Table - Slot Laches

T$_1$: Find D

hash(D)

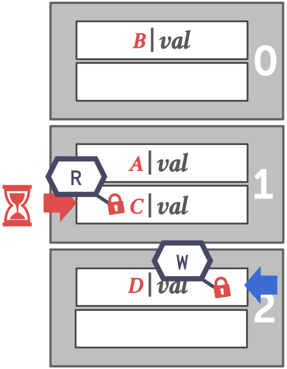


T$_2$: Insert E

hash(E)

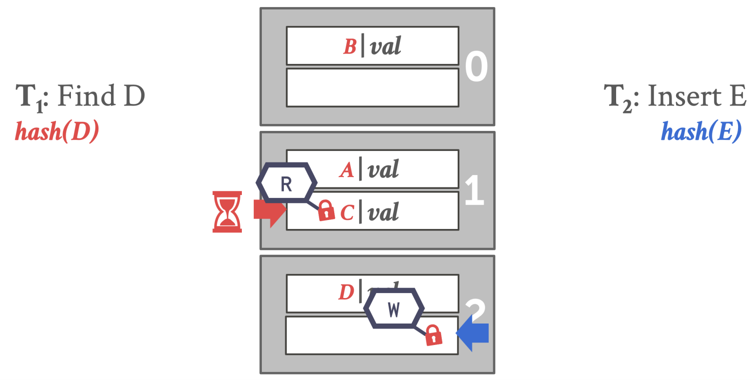
Hash Table - Slot Laches

T$_1$: Find D
hash(D)

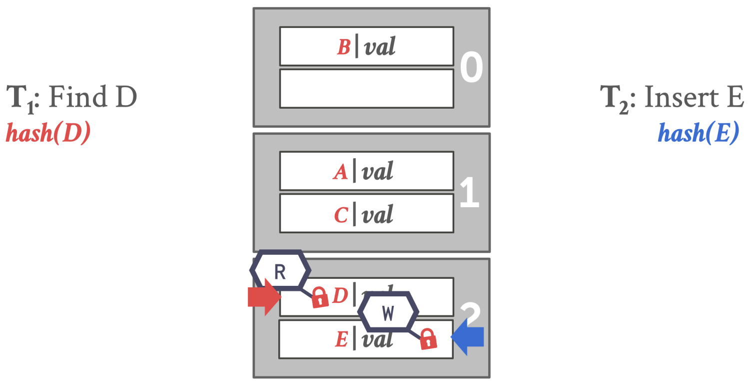


T$_2$: Insert E
hash(E)

Hash Table - Slot Laches



Hash Table - Slot Laches



T$_1$: Find D
hash(D)

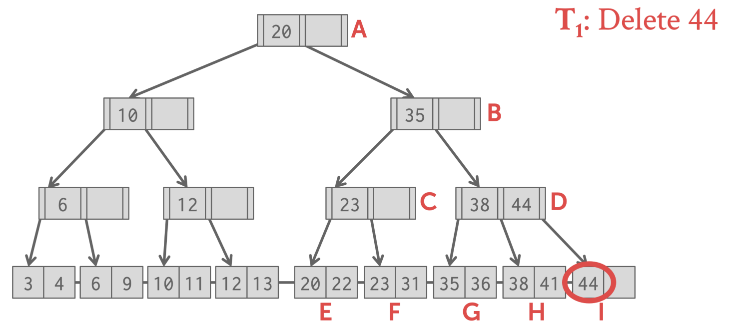
T$_2$: Insert E
hash(E)

B+Tree Concurrency Control

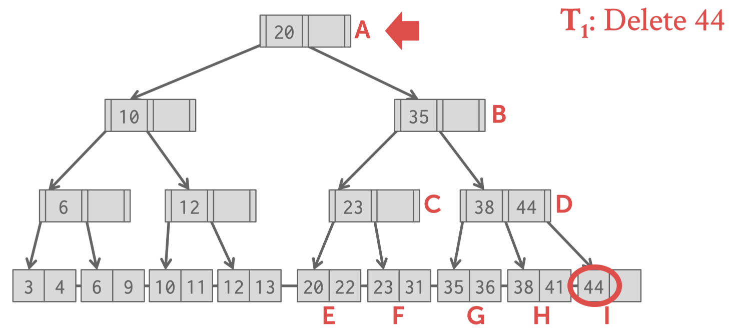
B+Tree Concurrency Control

- We want to allow multiple threads to read and update a B+Tree at the same time.
- We need to handle 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 Concurrency Control: Example

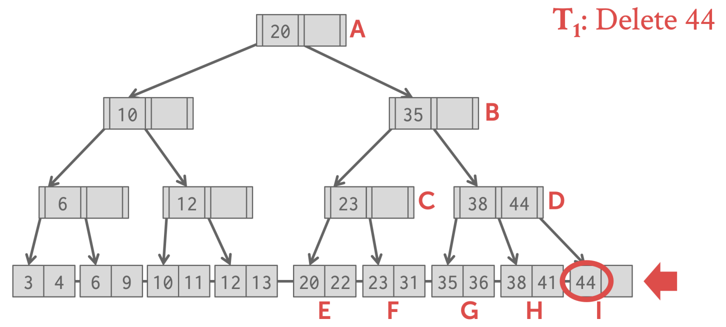


B+Tree Concurrency Control: Example

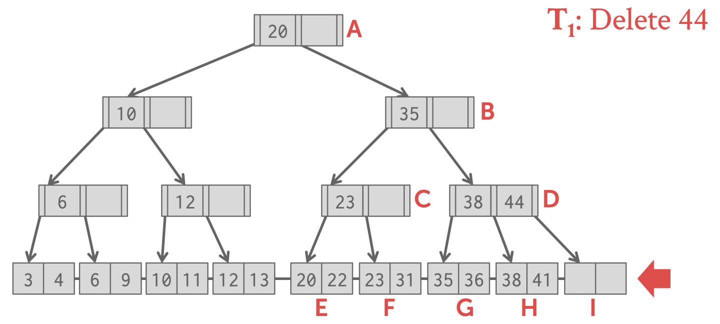


T$_1$: Delete 44

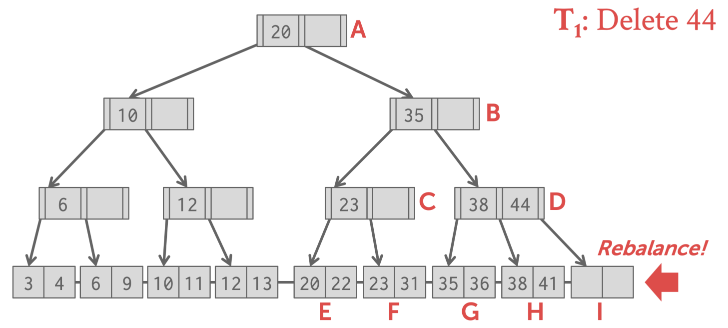
B+Tree Concurrency Control: Example



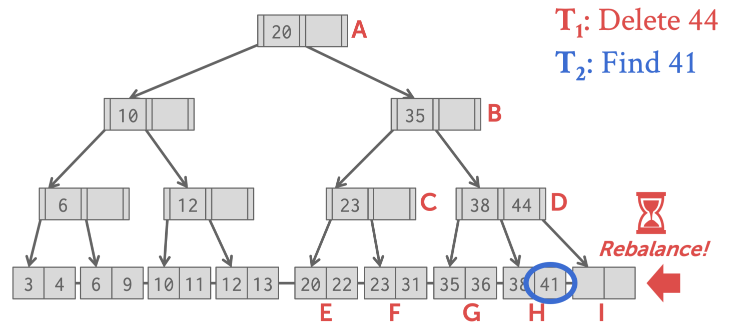
B+Tree Concurrency Control: Example



B+Tree Concurrency Control: Example



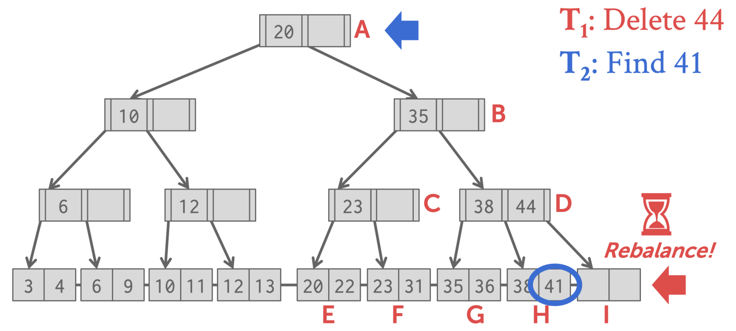
B+Tree Concurrency Control: Example



T_1: Delete 44

T_2: Find 41

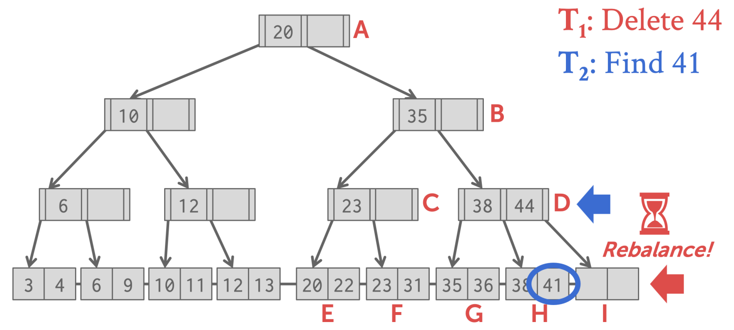
B+Tree Concurrency Control: Example



T$_1$: Delete 44

T$_2$: Find 41

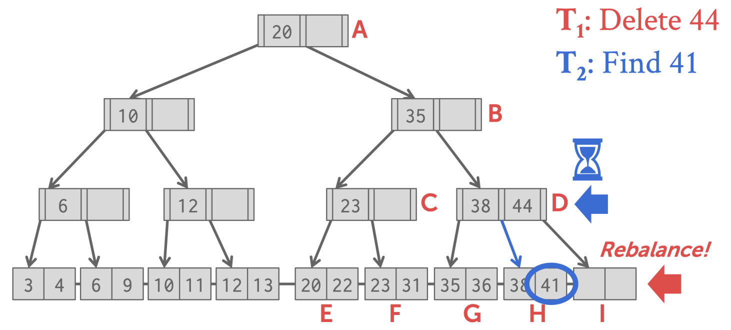
B+Tree Concurrency Control: Example



T$_1$: Delete 44

T$_2$: Find 41

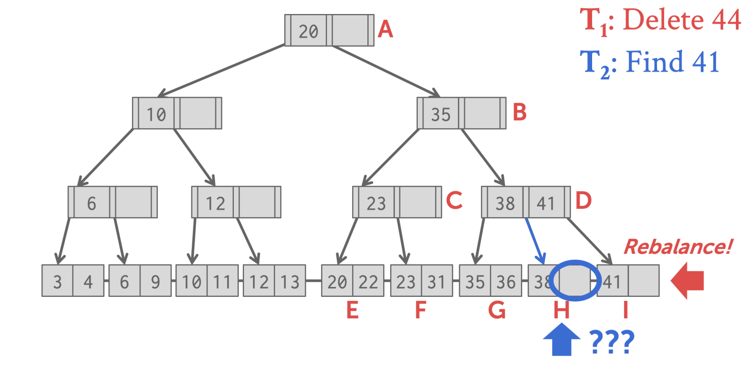
B+Tree Concurrency Control: Example



T$_1$: Delete 44

T$_2$: Find 41

B+Tree Concurrency Control: Example



T_1: Delete 44

T_2: Find 41

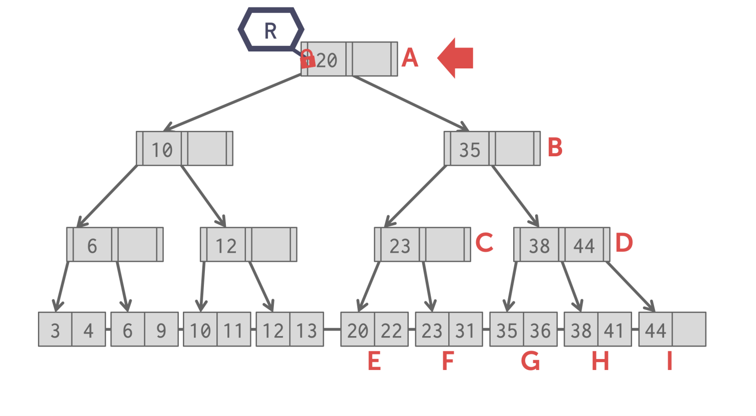
Latch Crabbing/Coupling

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

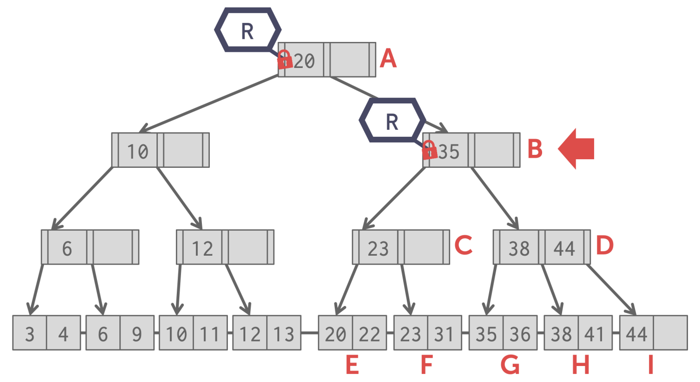
Latch Crabbing/Coupling

- **Find:** 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.

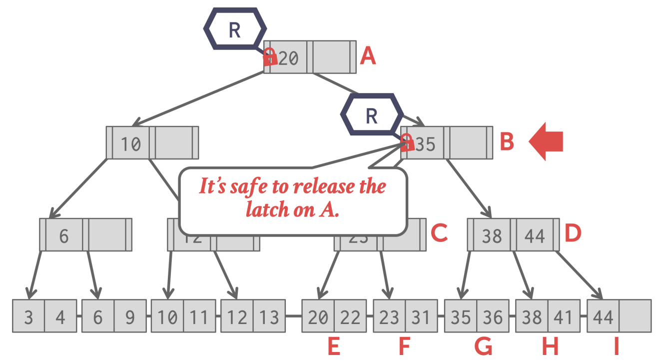
Example 1 - Find 38



Example 1 - Find 38

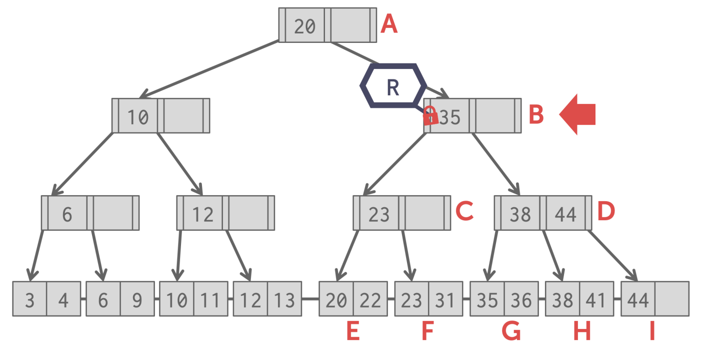


Example 1 - Find 38

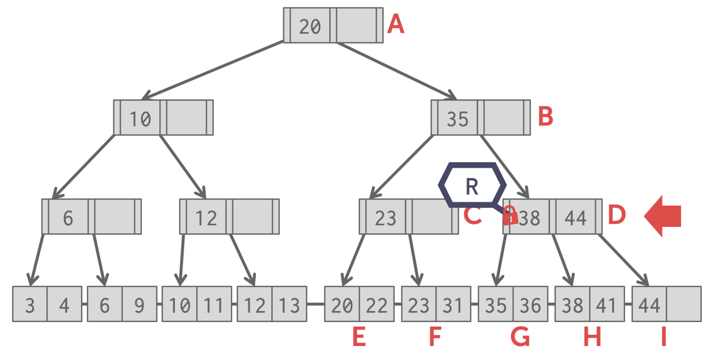


It's safe to release the latch on A.

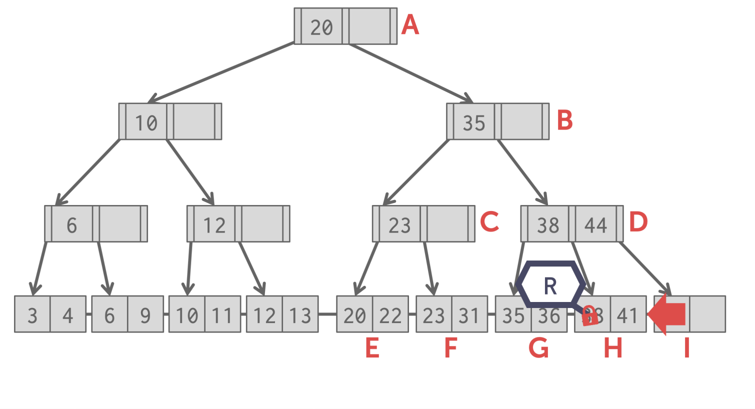
Example 1 - Find 38



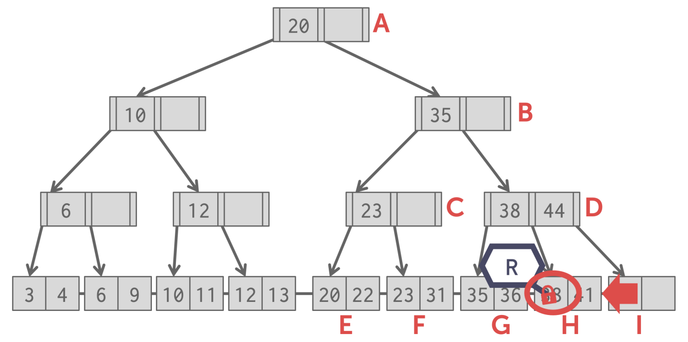
Example 1 - Find 38



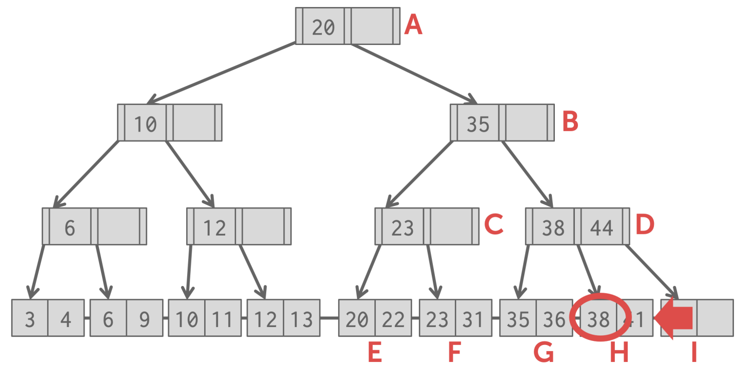
Example 1 - Find 38



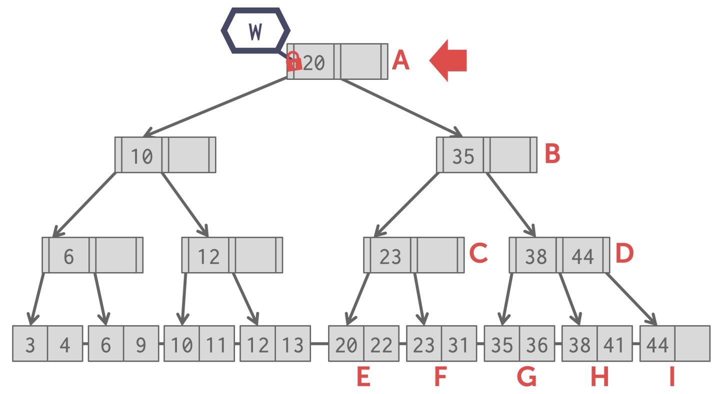
Example 1 - Find 38



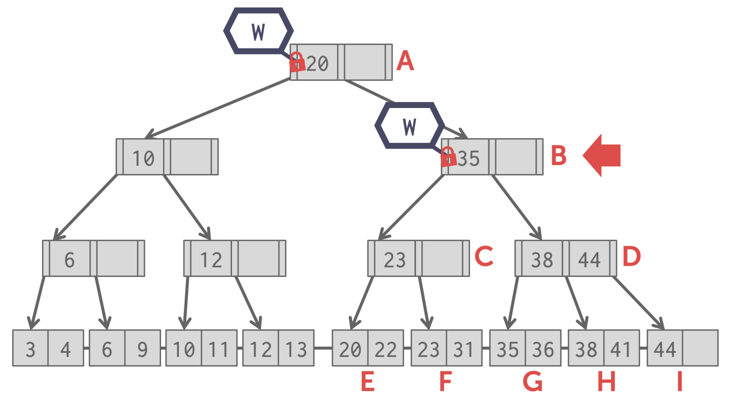
Example 1 - Find 38



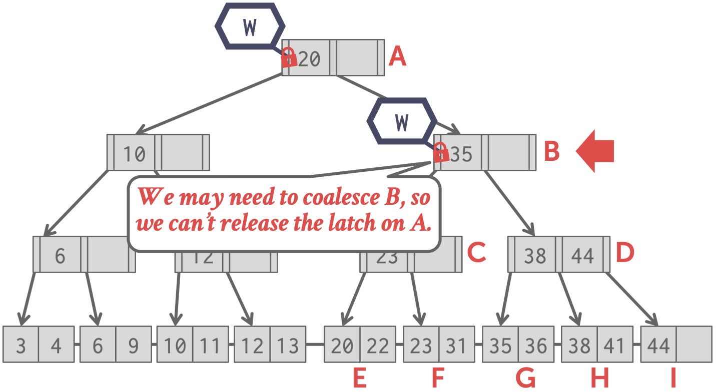
Example 2 - Delete 38



Example 2 - Delete 38

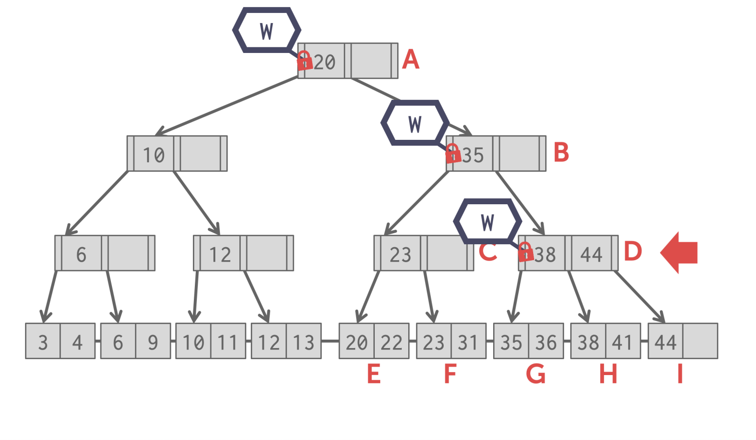


Example 2 - Delete 38

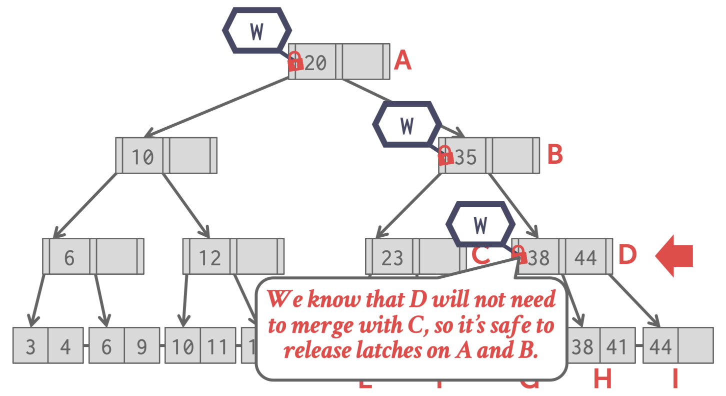


We may need to coalesce B, so we can't release the latch on A.

Example 2 - Delete 38

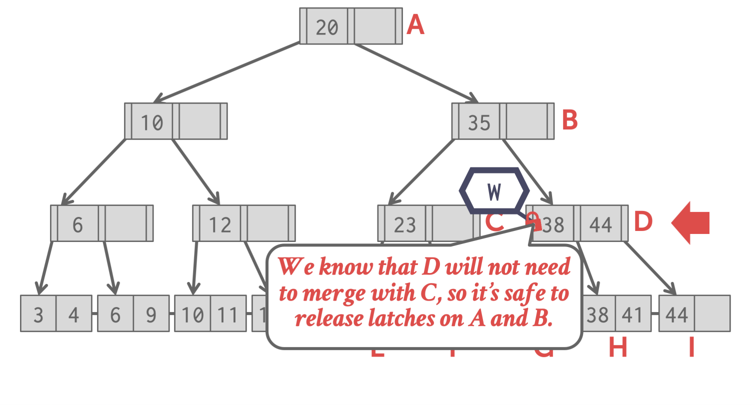


Example 2 - Delete 38



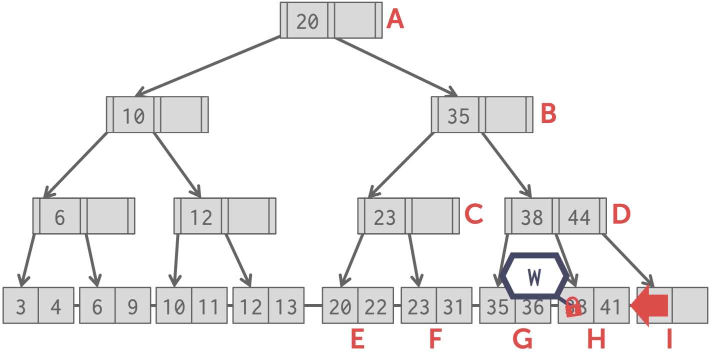
We know that D will not need to merge with C, so it's safe to release latches on A and B.

Example 2 - Delete 38

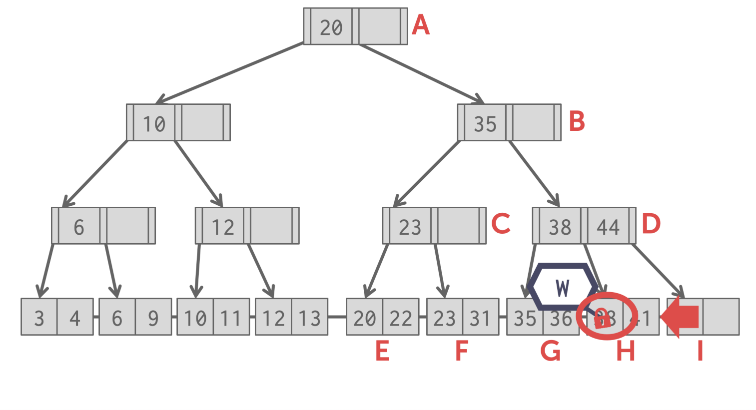


We know that D will not need to merge with C, so it's safe to release latches on A and B.

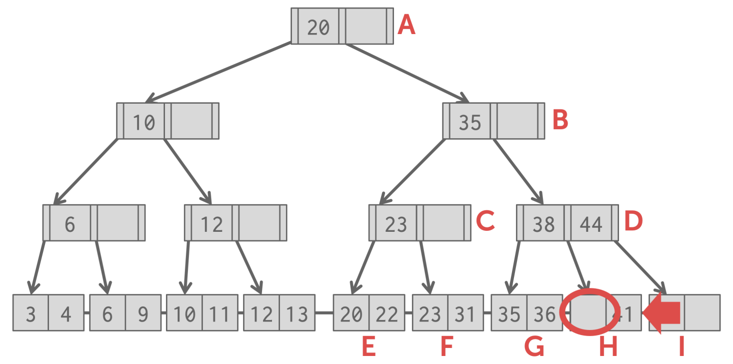
Example 2 - Delete 38



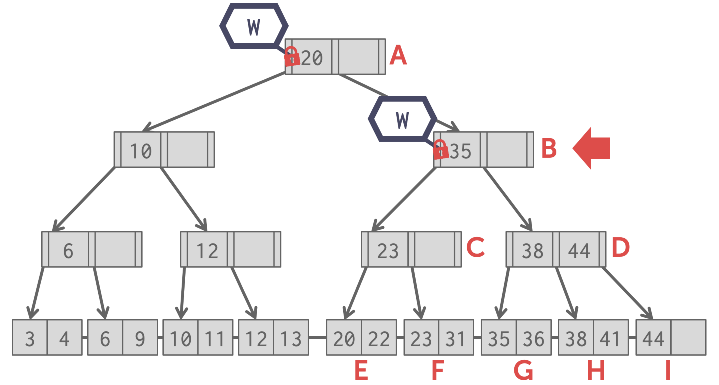
Example 2 - Delete 38



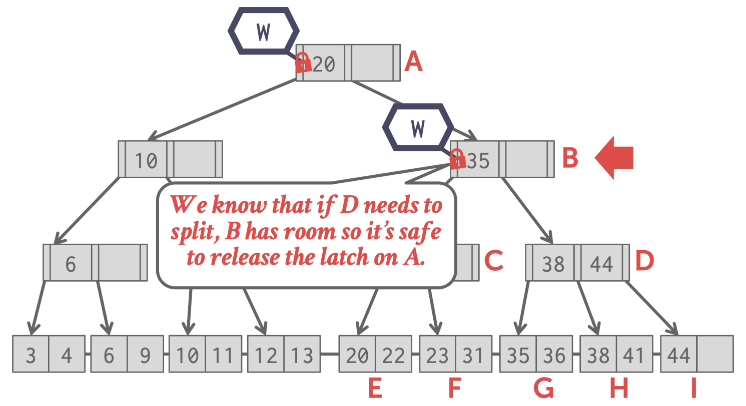
Example 2 - Delete 38



Example 3 - Insert 45

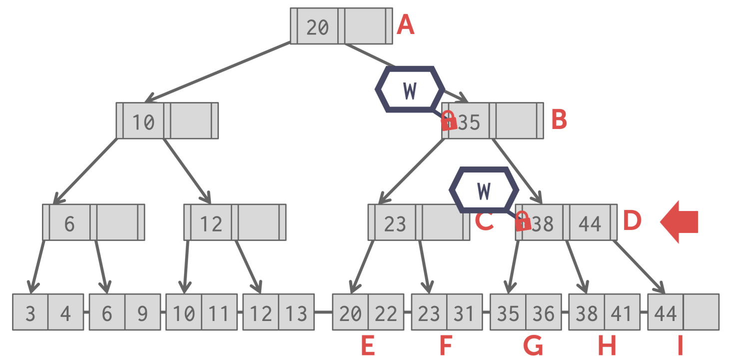


Example 3 - Insert 45

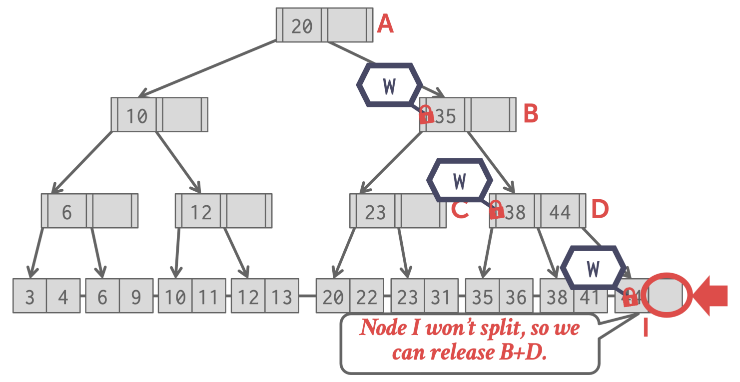


We know that if D needs to split, B has room so it's safe to release the latch on A.

Example 3 - Insert 45

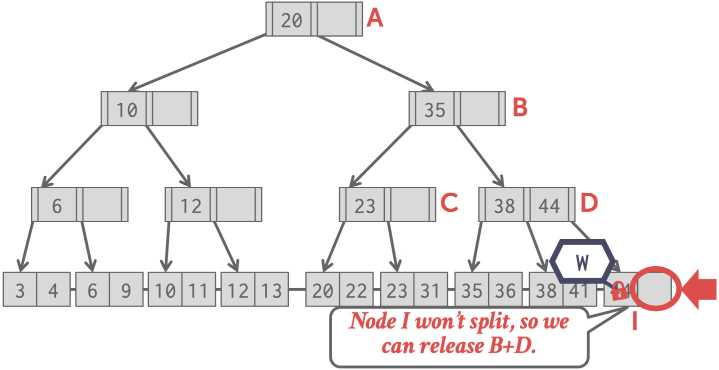


Example 3 - Insert 45



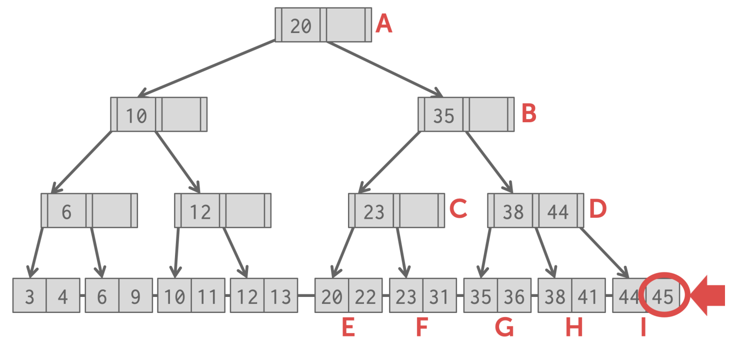
Node I won't split, so we can release B+D.

Example 3 - Insert 45

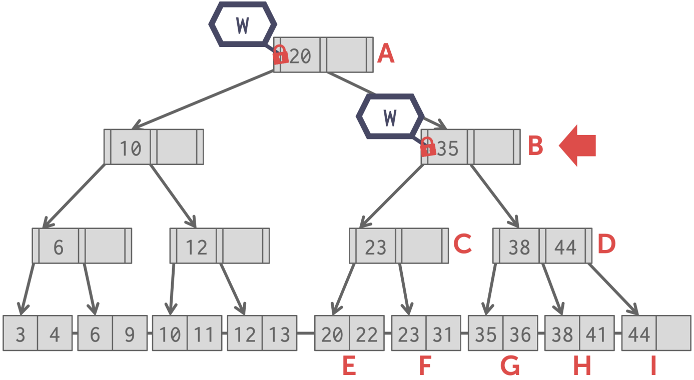


Node I won't split, so we can release B+D.

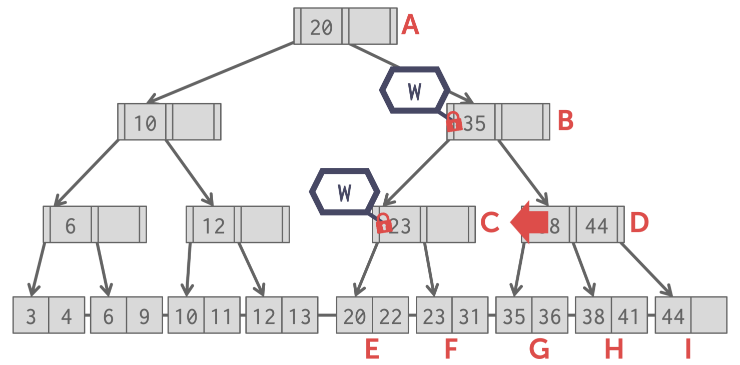
Example 3 - Insert 45



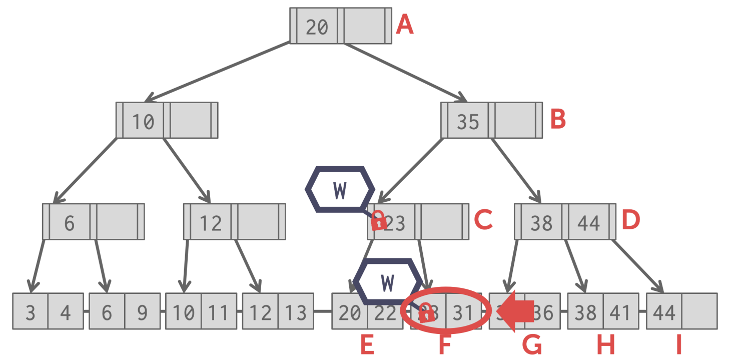
Example 4 - Insert 25



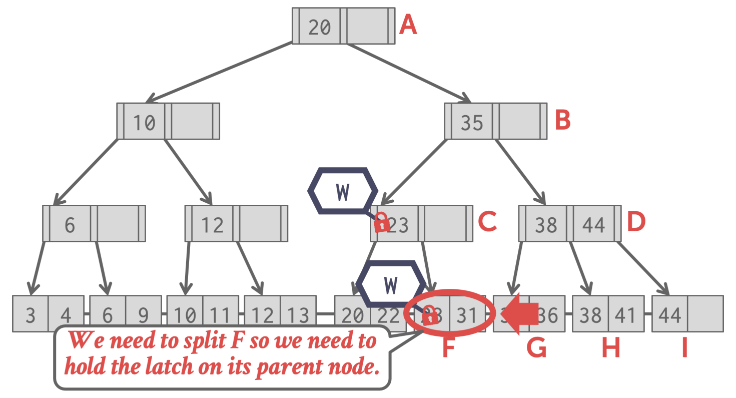
Example 4 - Insert 25



Example 4 - Insert 25

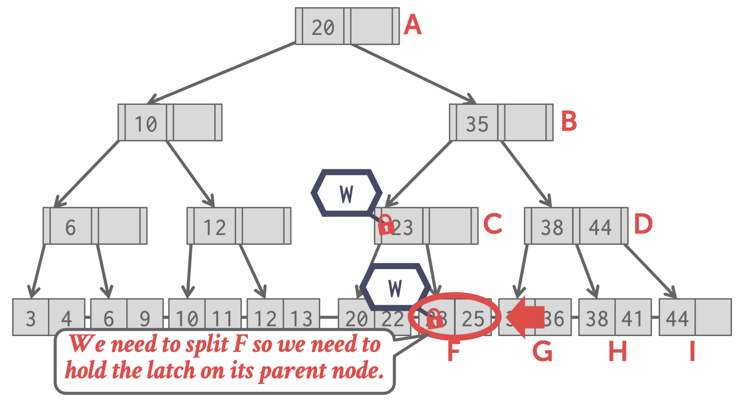


Example 4 - Insert 25



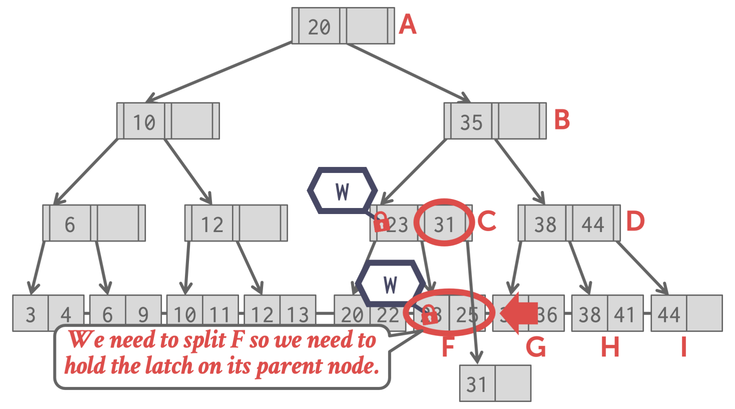
We need to split F so we need to hold the latch on its parent node.

Example 4 - Insert 25



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

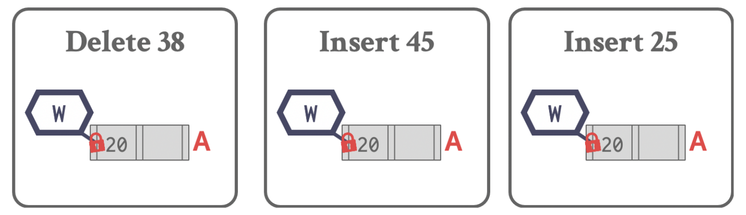
Example 4 - Insert 25



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

Observation

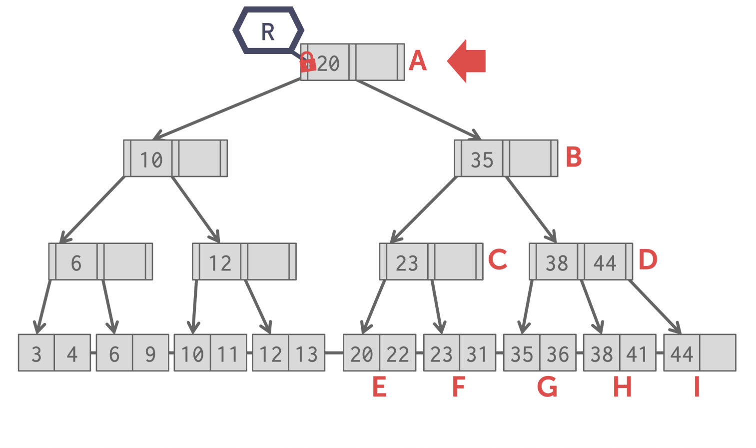
- What was the first step that all the update examples did on the B+Tree?
- Taking a write latch on the root every time becomes a bottleneck with higher concurrency.
- Can we do better?



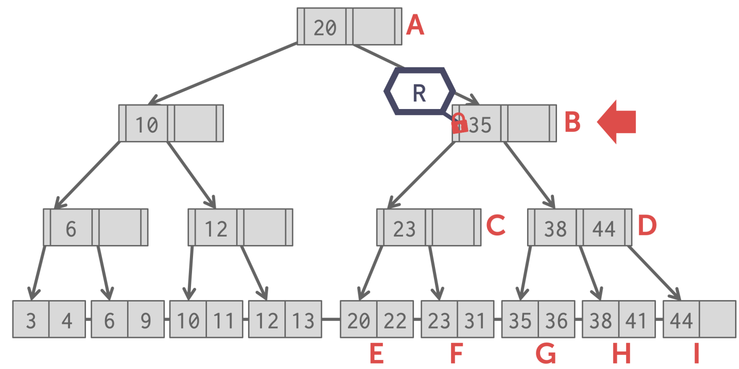
Better Latching Algorithm

- Assume that the leaf node is safe.
- Use read latches and crabbing to reach it, and then verify that it is safe.
- If leaf is not safe, then do previous algorithm using write latches.
- Reference

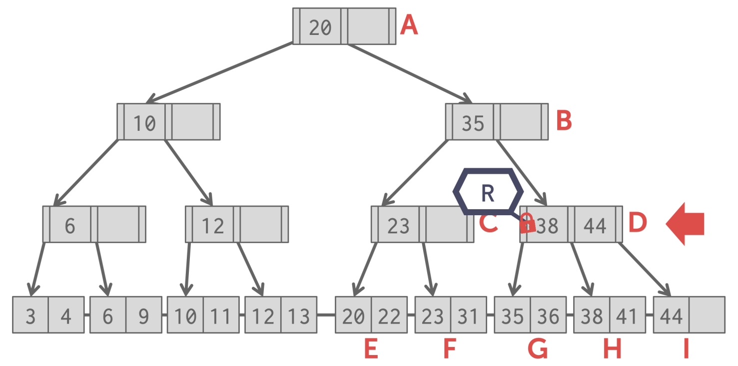
Example 2 - Delete 38



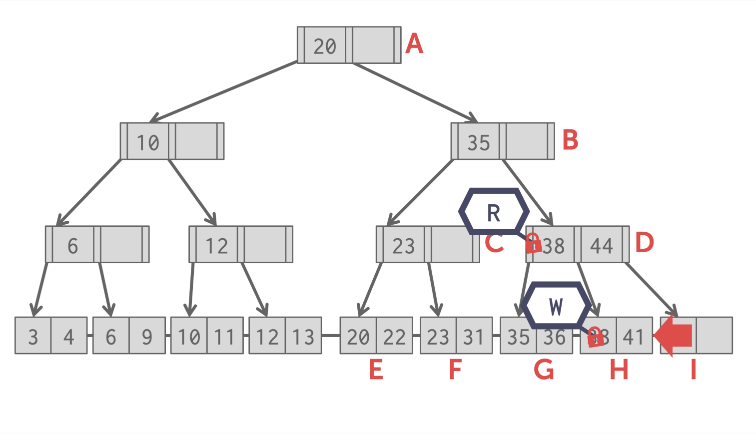
Example 2 - Delete 38



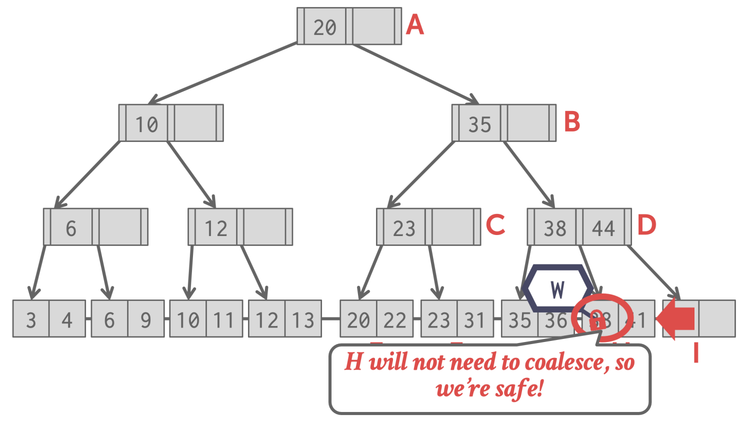
Example 2 - Delete 38



Example 2 - Delete 38

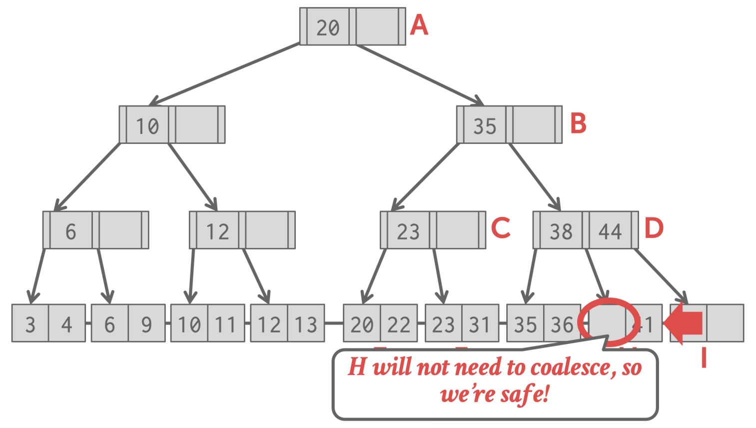


Example 2 - Delete 38



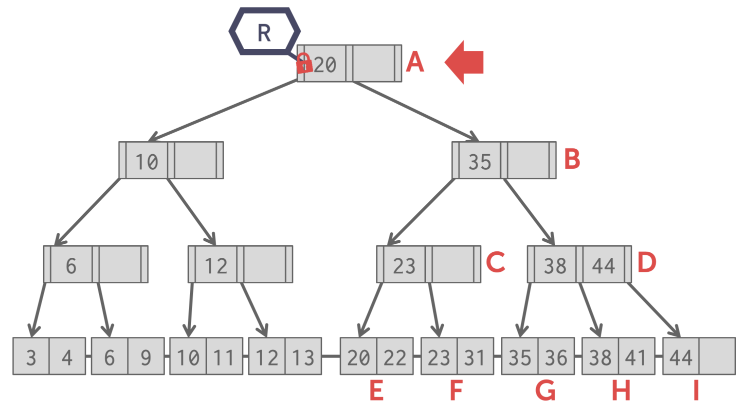
H will not need to coalesce, so we're safe!

Example 4 - Insert 25

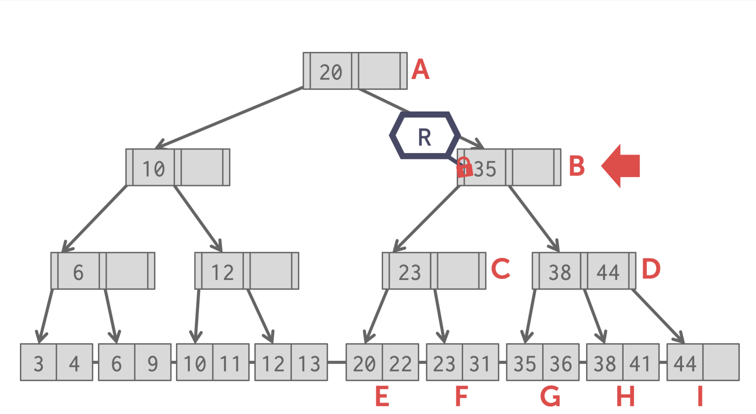


H will not need to coalesce, so we're safe!

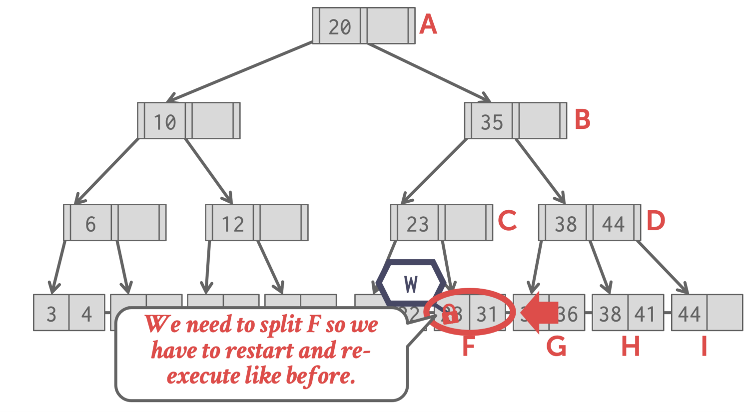
Example 4 - Insert 25



Example 4 - Insert 25



Example 4 - Insert 25



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

Better Latching Algorithm

- **Find:** 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 **W** 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.

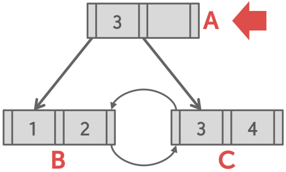
Leaf Node Scans

Observation

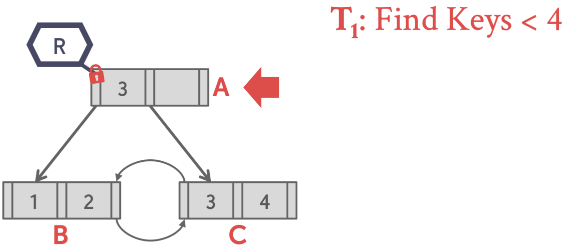
- The threads in all the examples so far have acquired latches in a **top-down** manner.
    - A thread can only acquire a latch from a node that is below its current node.
    - If the desired latch is unavailable, the thread must wait until it becomes available.
- But what if we want to move from one leaf node to another leaf node?
- Leaf nodes can include **hint keys** to approximate the next key at your sibling.

Leaf Node Scan - Example 1

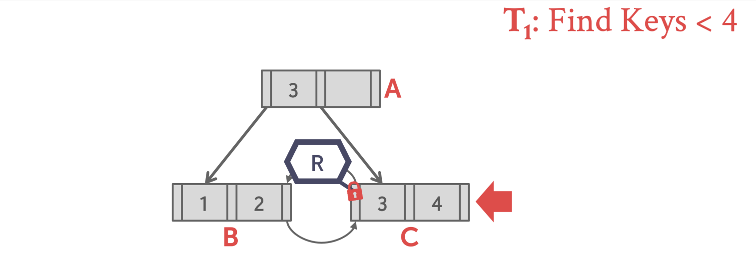
T_1: Find Keys < 4



Leaf Node Scan - Example 1

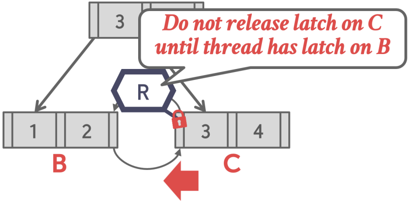

T$_1$: Find Keys < 4

Leaf Node Scan - Example 1



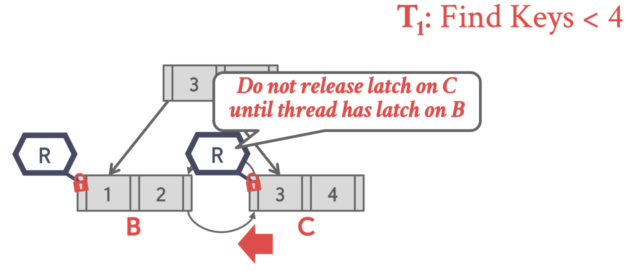
Leaf Node Scan - Example 1

T_1: Find Keys < 4



Do not release latch on C until thread has latch on B

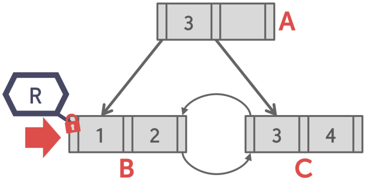
Leaf Node Scan - Example 1



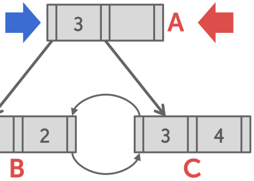
Do not release latch on C until thread has latch on B

Leaf Node Scan - Example 1

T_1: Find Keys < 4



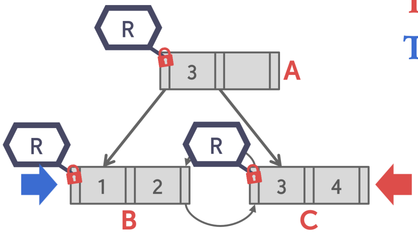
Leaf Node Scan - Example 2



T_1: Find Keys < 4

T_2: Find Keys > 1

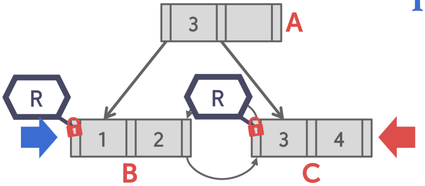
Leaf Node Scan - Example 2



T$_1$: Find Keys < 4

T$_2$: Find Keys > 1

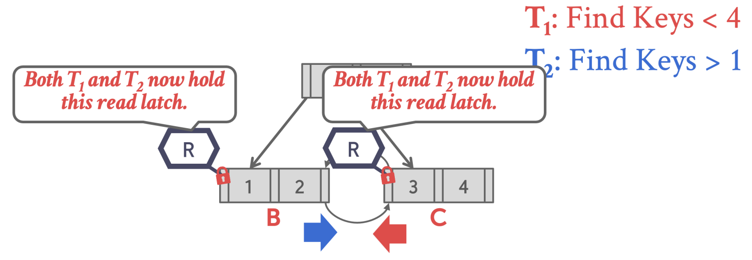
Leaf Node Scan - Example 2



T_1: Find Keys < 4

T_2: Find Keys > 1

Leaf Node Scan - Example 2



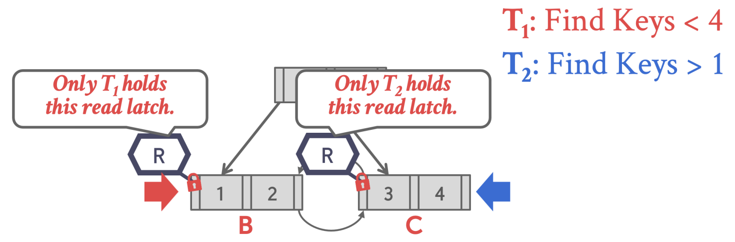
T_1: Find Keys < 4

T_2: Find Keys > 1

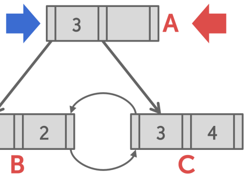
Both T_1 and T_2 now hold this read latch.

Both T_1 and T_2 now hold this read latch.

Leaf Node Scan - Example 2



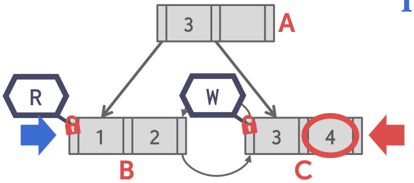
Leaf Node Scan - Example 3



T_1: Delete 4

T_2: Find Keys > 1

Leaf Node Scan - Example 3



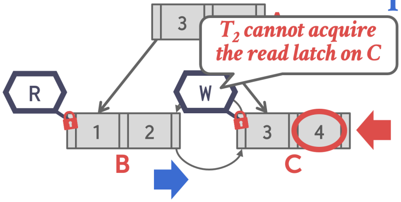
T_1: Delete 4

T_2: Find Keys > 1

Leaf Node Scan - Example 3

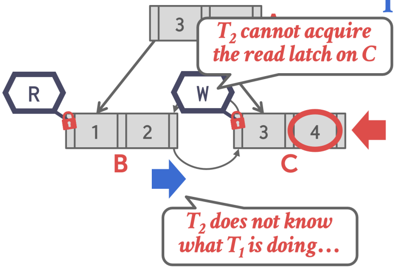
T_1: Delete 4

T_2: Find Keys > 1



T_2 cannot acquire the read latch on C

Leaf Node Scan - Example 3



T~1~: Delete 4

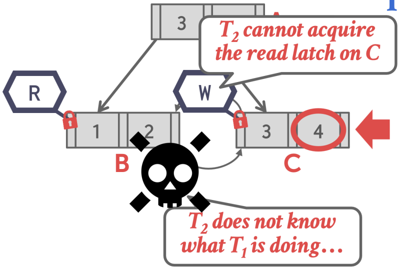
T_1: Delete 4

T_2: Find Keys > 1

T_2 cannot acquire the read latch on C

T_2 does not know what T_1 is doing…

Leaf Node Scan - Example 3



T_1: Delete 4

T_2: Find Keys > 1

T_2 cannot acquire the read latch on C

T_2 does not know what T_1 is doing…

Leaf Node Scans

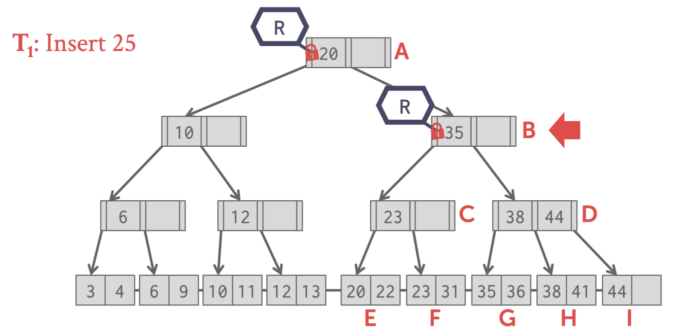
- Latches do **not** support deadlock detection or avoidance.
- The only way we can deal with this problem is through **coding discipline**.
- The leaf node sibling latch acquisition protocol must support a fail-fast **no-wait** mode.
- B+Tree implementation must cope with failed latch acquisitions.

B^{link}-Tree

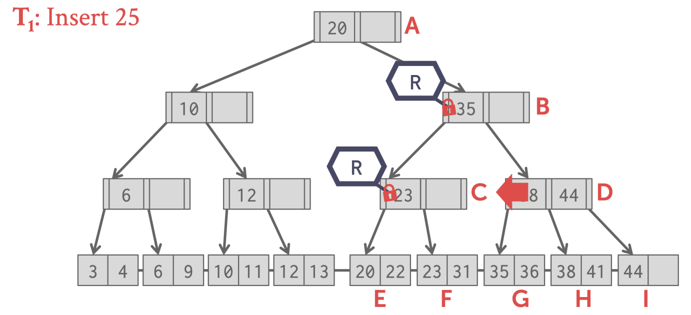
B^{link}-Tree

- Every time a leaf node overflows, we must update at least **three** nodes.
    - The leaf node being split.
    - The new leaf node being created.
    - The parent node.
- **Optimization:** When a leaf node overflows, delay updating its parent node.
- Reference

B^{link}-Tree Example

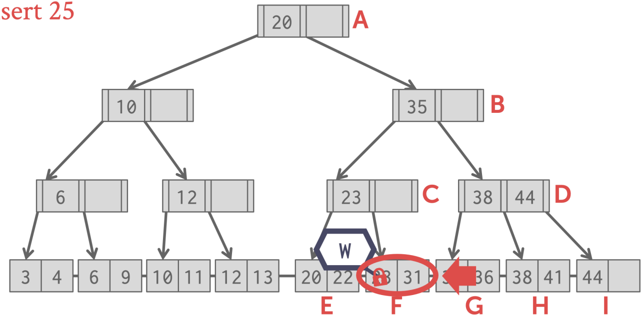


B^{link}-Tree Example

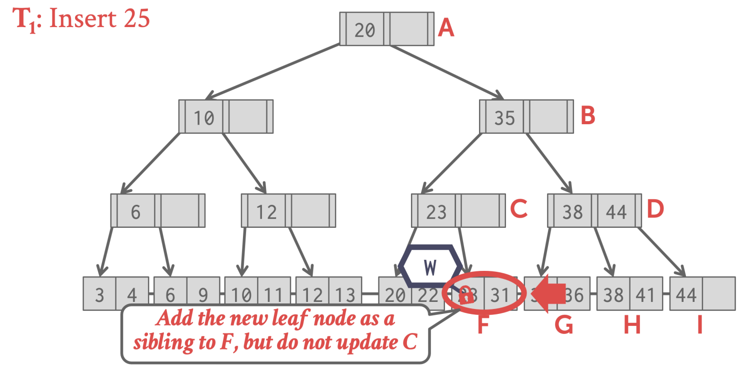


B^{link}-Tree Example

T_1: Insert 25

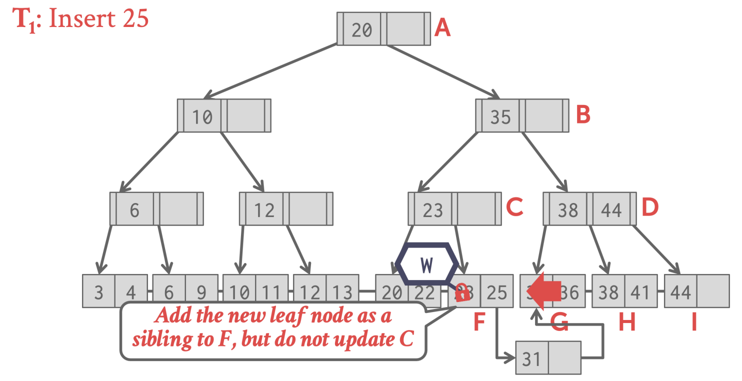


B^{link}-Tree Example

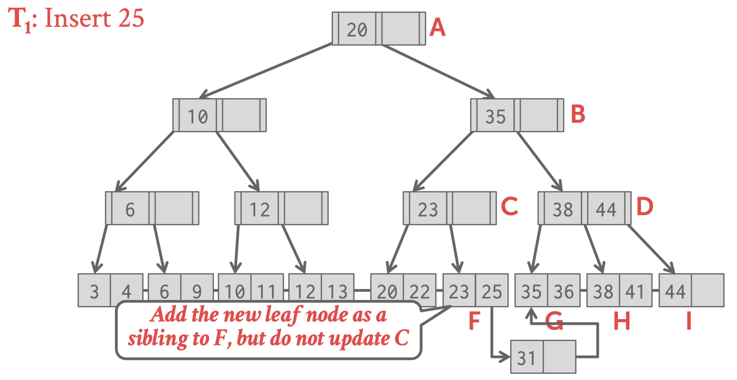

T$_1$: Insert 25

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

B^{link}-Tree Example

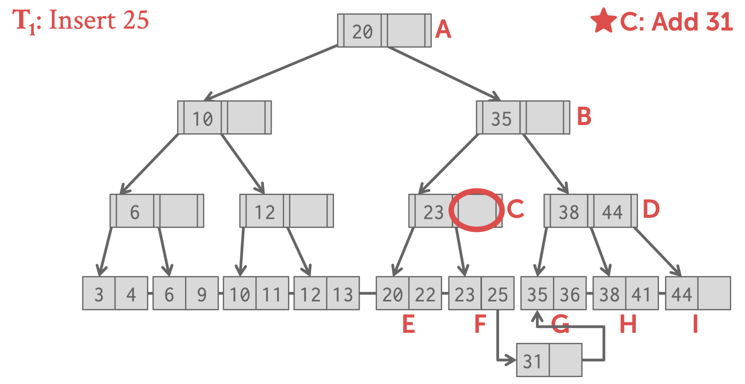


B^{link}-Tree Example

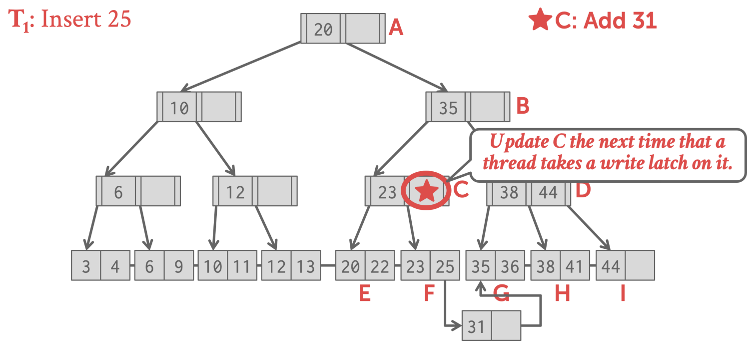

T$_1$: Insert 25

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

B^{link}-Tree Example



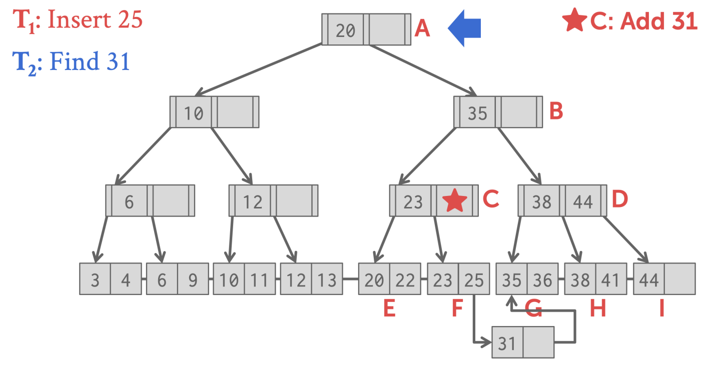
B^{link}-Tree Example


T$_1$: Insert 25

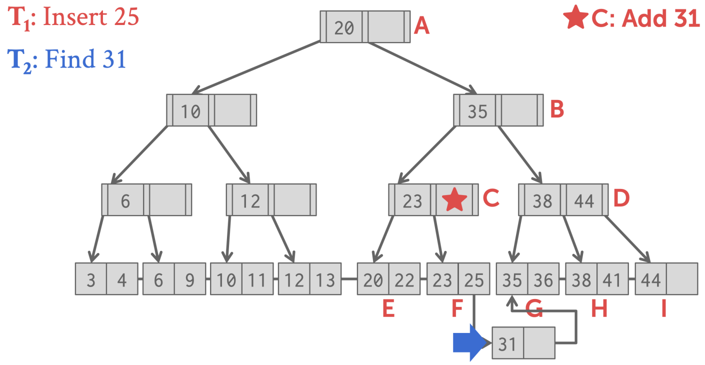
★C: Add 31

Update C the next time that a thread takes a write latch on it.

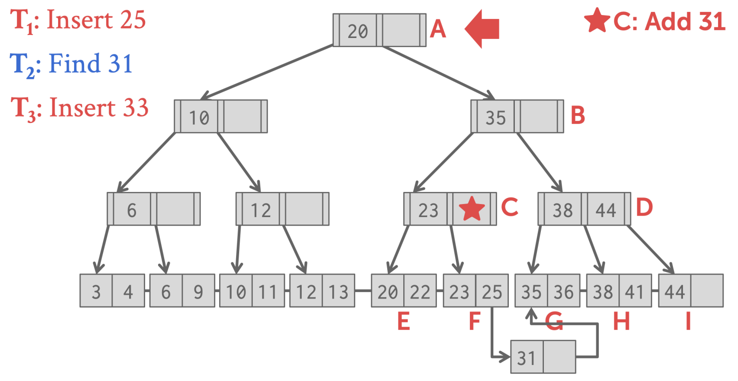
B^{link}-Tree Example



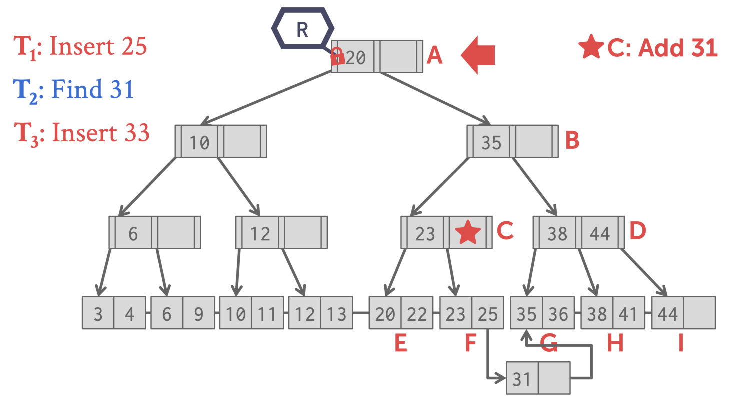
B^{link}-Tree Example



B^{link}-Tree Example



B^{link}-Tree Example


T_1: Insert 25

T_2: Find 31

T_3: Insert 33

★C: Add 31

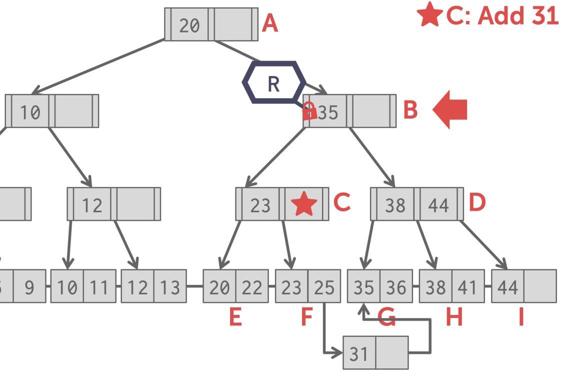
B^{link}-Tree Example

T_1: Insert 25

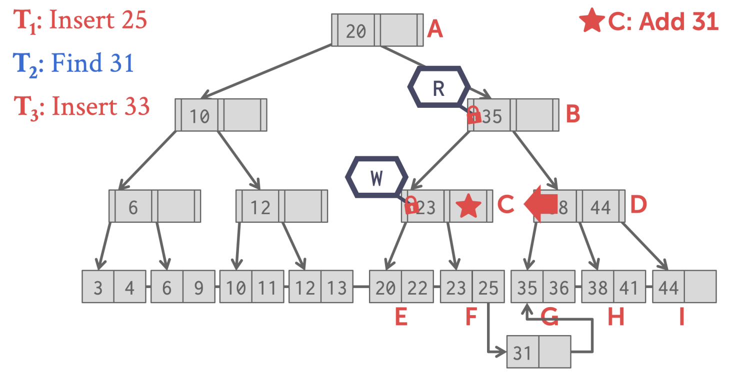
T_2: Find 31

T_3: Insert 33

★C: Add 31



B^{link}-Tree Example


T$_1$: Insert 25

T$_2$: Find 31

T$_3$: Insert 33

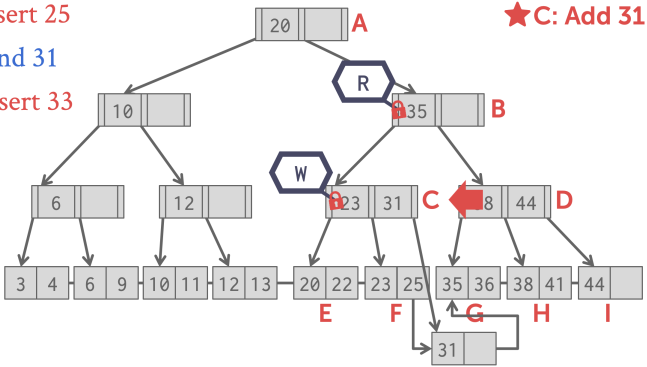
★C: Add 31

B^{link}-Tree Example

T$_1$: Insert 25

T$_2$: Find 31

T$_3$: Insert 33



★C: Add 31

Conclusion

Conclusion

- Making a data structure thread-safe is notoriously difficult in practice.
- We focused on B+Trees but the same high-level techniques are applicable to other data structures.
- Next Class
    - We will learn about modern access methods.