Concurrency Control for Indexes

Use Latches (NOT Locks): What and Why

- Many data structures inside the DBMS where you want mutual exclusion
  - Pin count on buffer page
  - Wait queue in lock table
  - B+-tree pages (we'll learn this)
- Instead of using a heavy-weight lock, we use latches
  - Not a 2-phase lock: we will unlatch almost immediately (short duration)
  - Usually in-memory next to the data structure in question (Fast!)
  - S and X latches for reads and writes, respectively
  - Have to latch/unlatch *very carefully* – only DBMS engineers get access to these APIs

B^+-Tree Concurrency Control

- Two possible problems
  - Two Xacts/threads modifying the contents of a node at the same time
  - One Xact/thread traversing the tree while another splits/merges nodes

Simple idea: B+-tree Latching

- Latch paths
  - Base case: Get a latch on the root
  - Induction:
    - Assume you have a latch on a node N
    - Get a latch on the appropriate child C of N

- Problems?

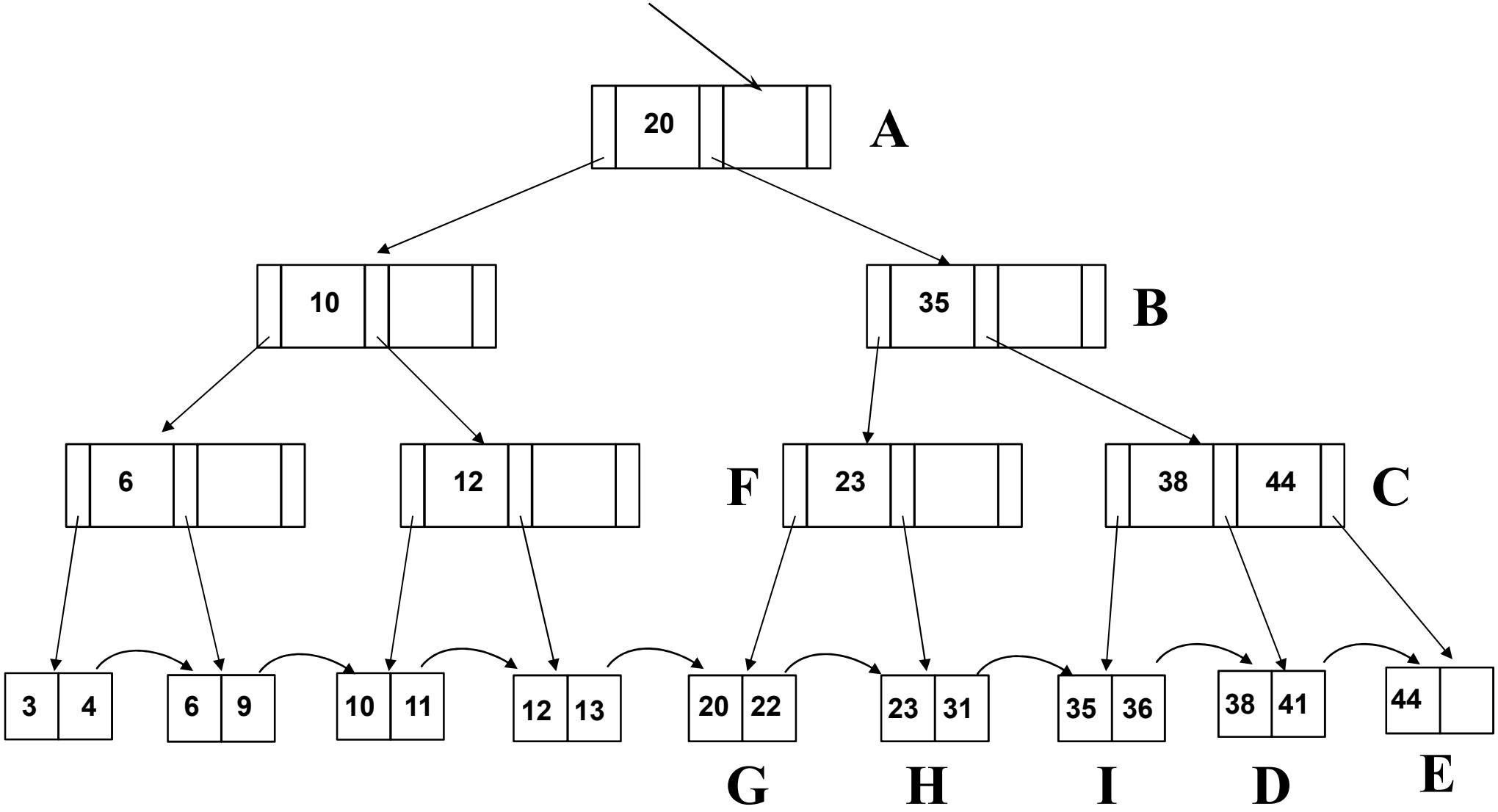
Latch Coupling ("crabbing")

- Basic idea
  - Base case: Get a latch on the root
  - Induction:
    - Assume you have a latch on a node N
    - Get a latch on the appropriate child C of N
    - Release latch on N (for parent) if "safe"
- A node is safe when it will not be split or merge when updated
  - Not full (on insertion)
  - More than half-full (on deletion)
- Benefits?

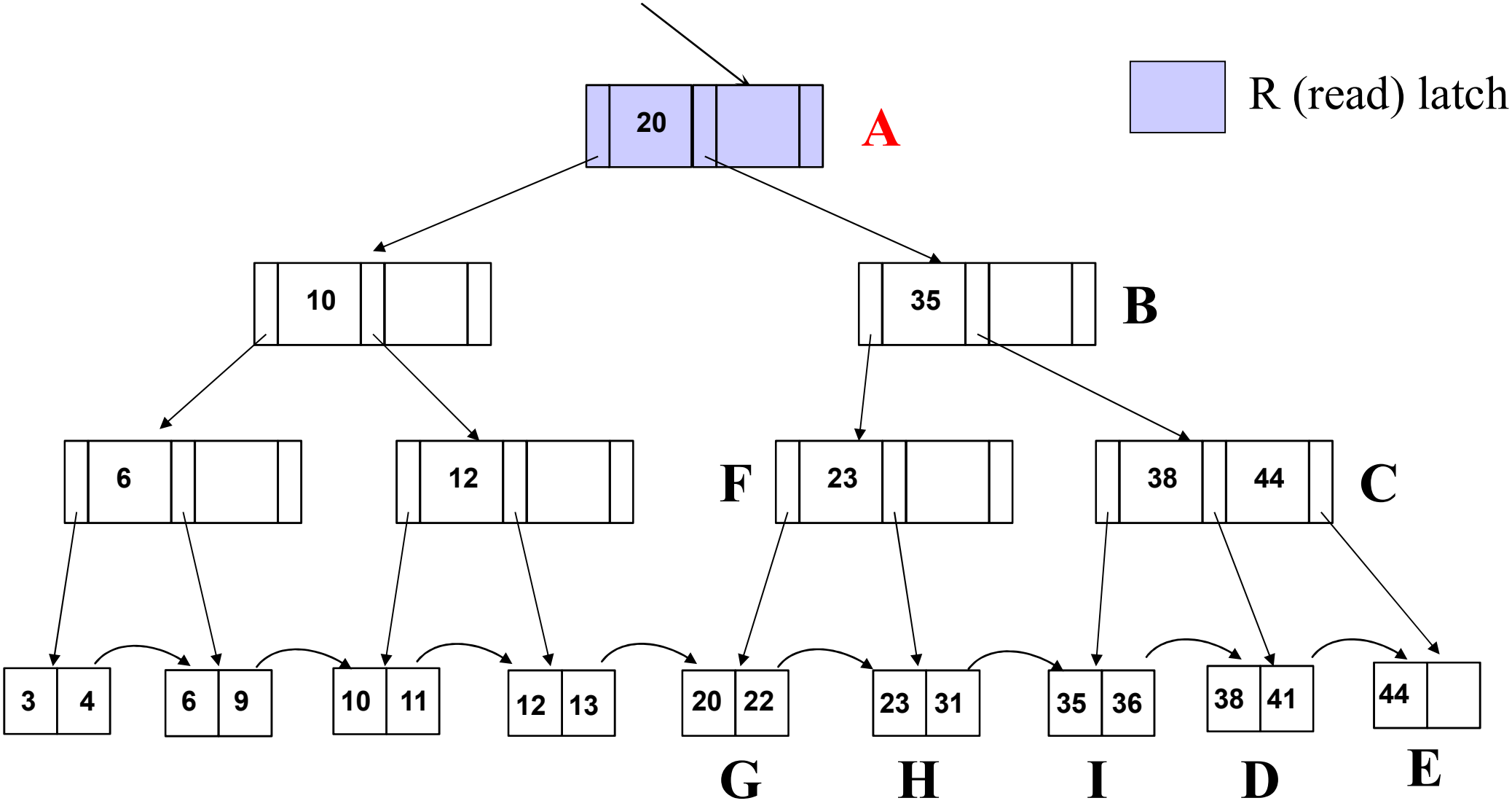
Latch Coupling (“crabbing”)

- Read
  - 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

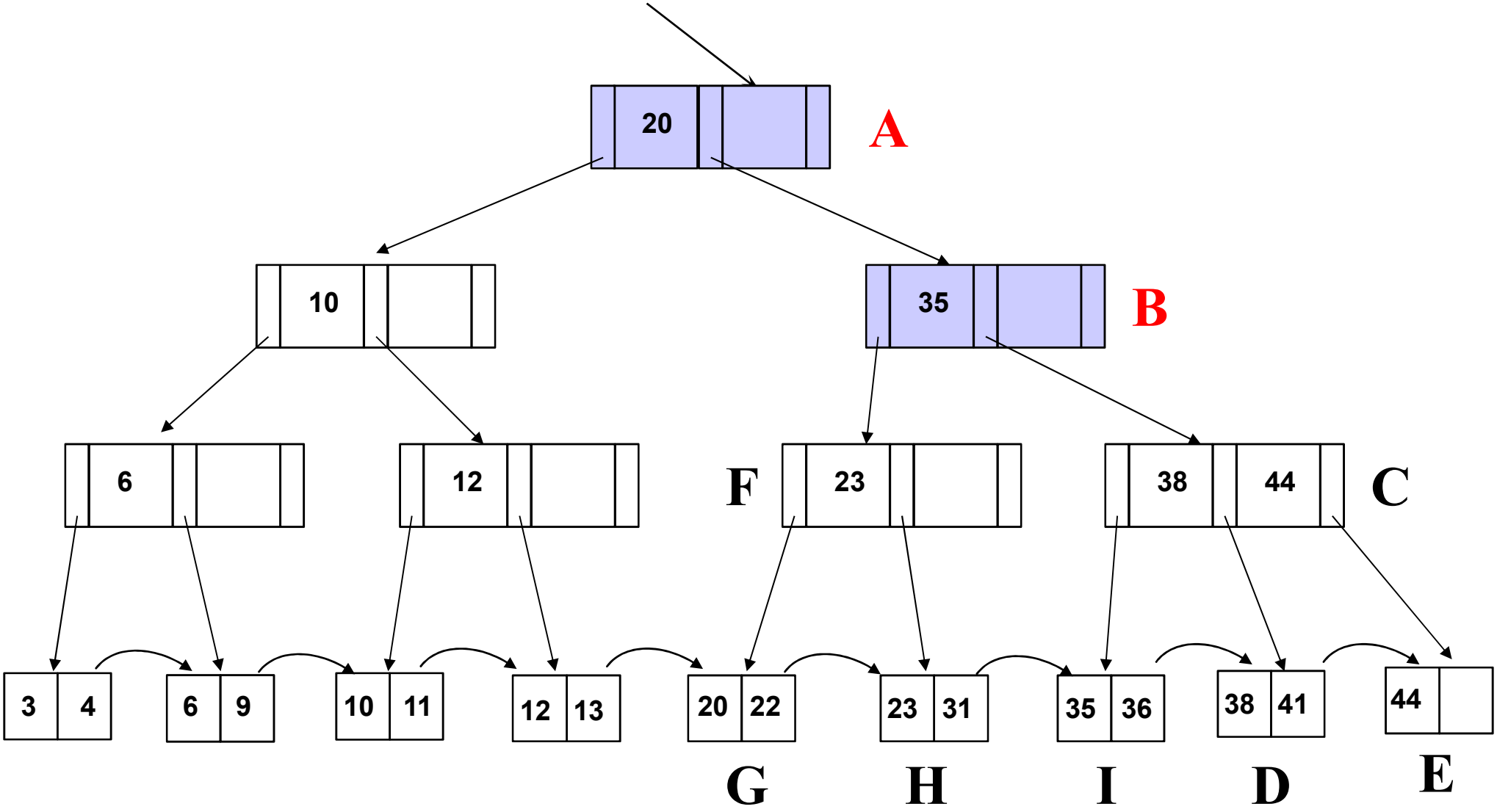
B+-tree (Tree-based) Latching – Crabbing Protocol



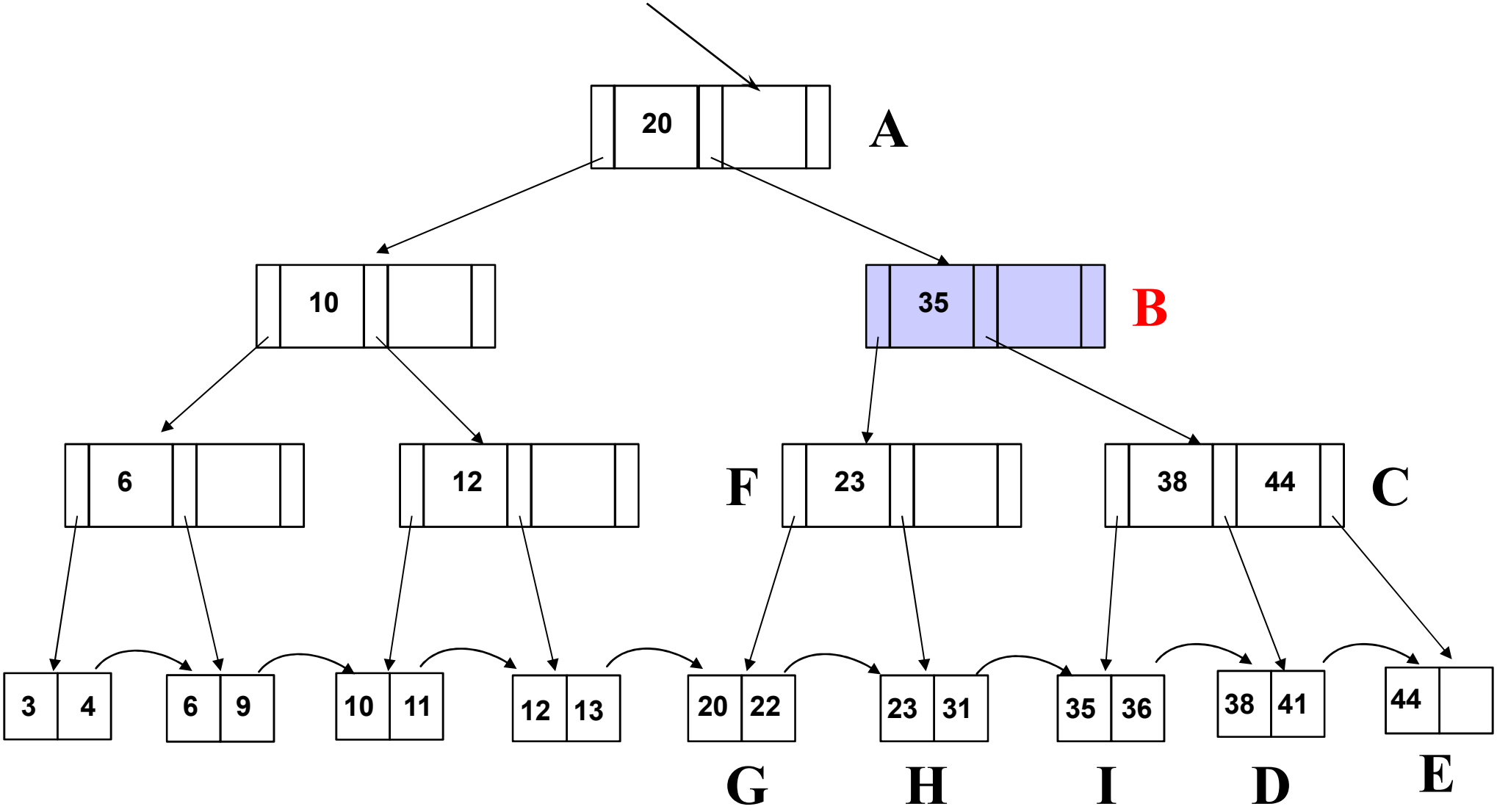
B+-tree Latching (Read 38)



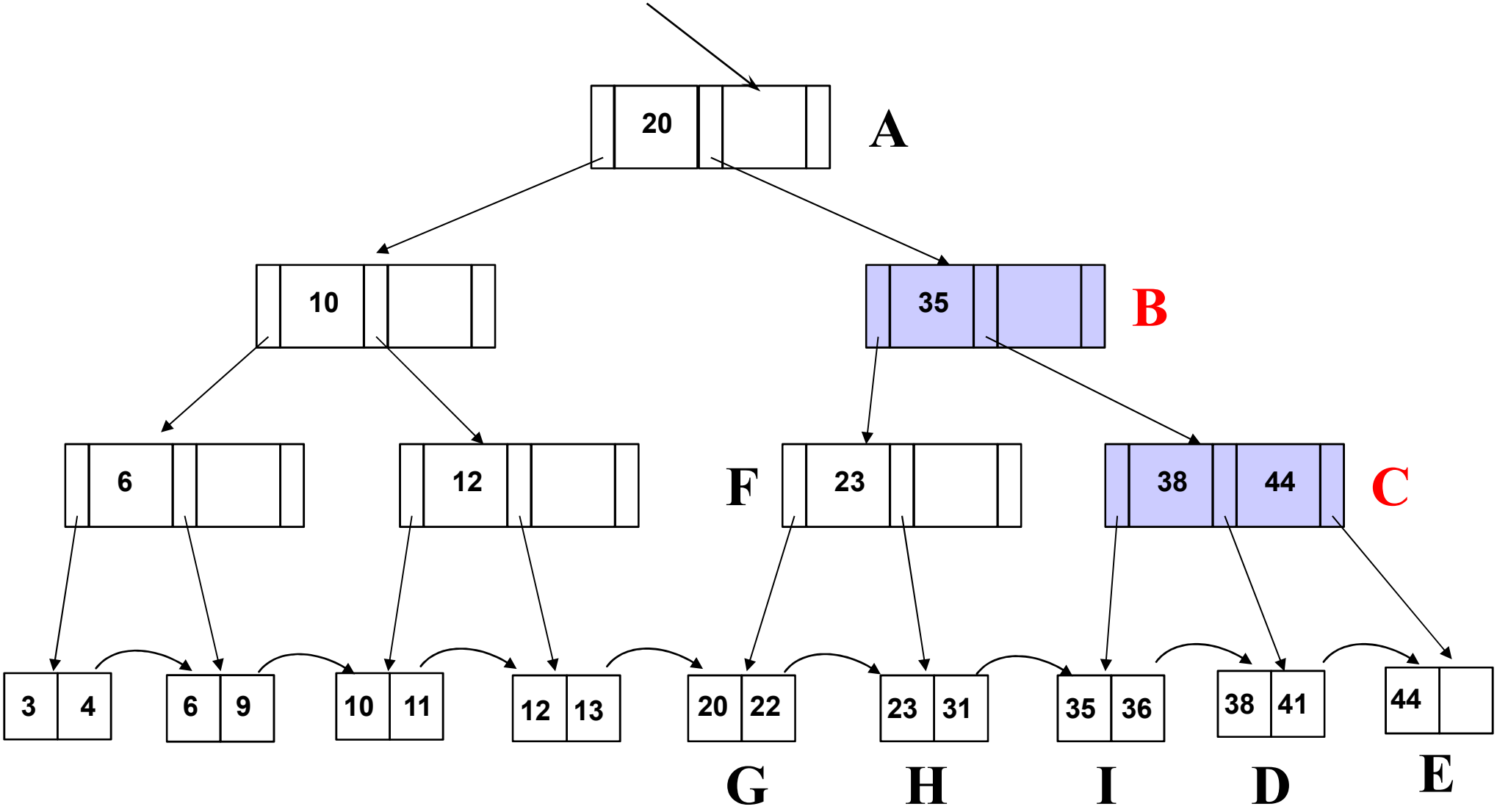
B+-tree Latching (Read 38)



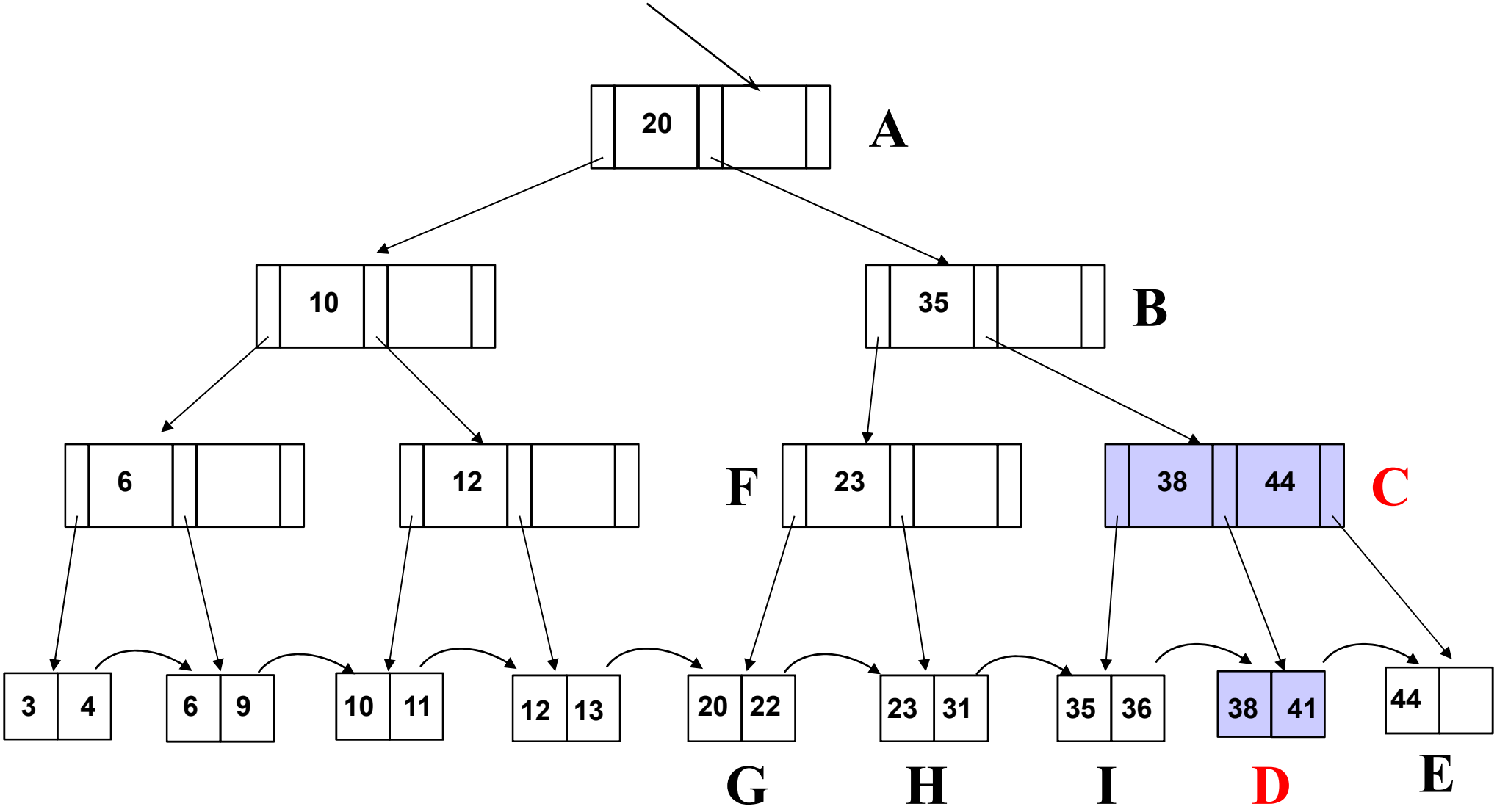
B+-tree Latching (Read 38)



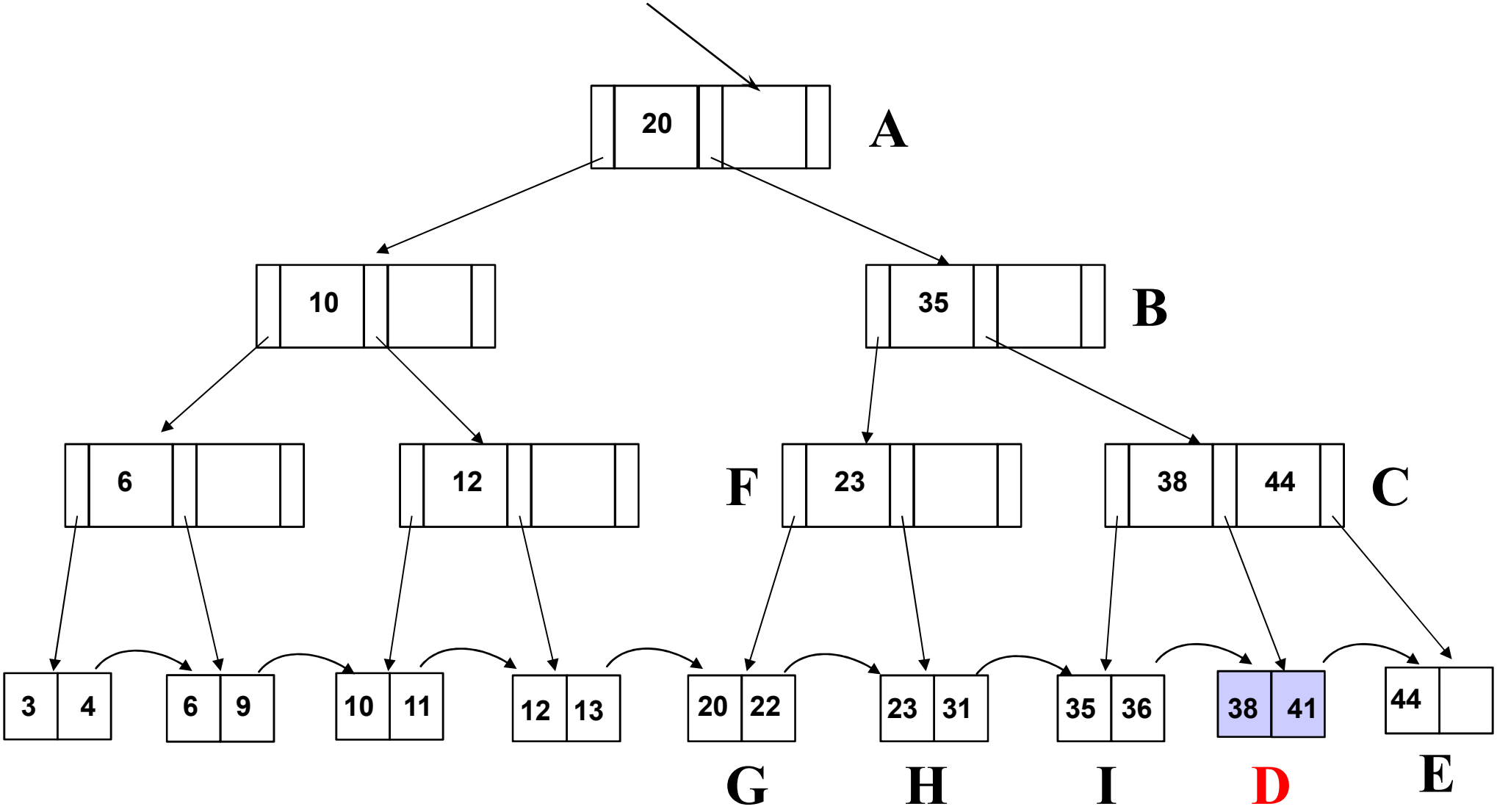
B+-tree Latching (Read 38)



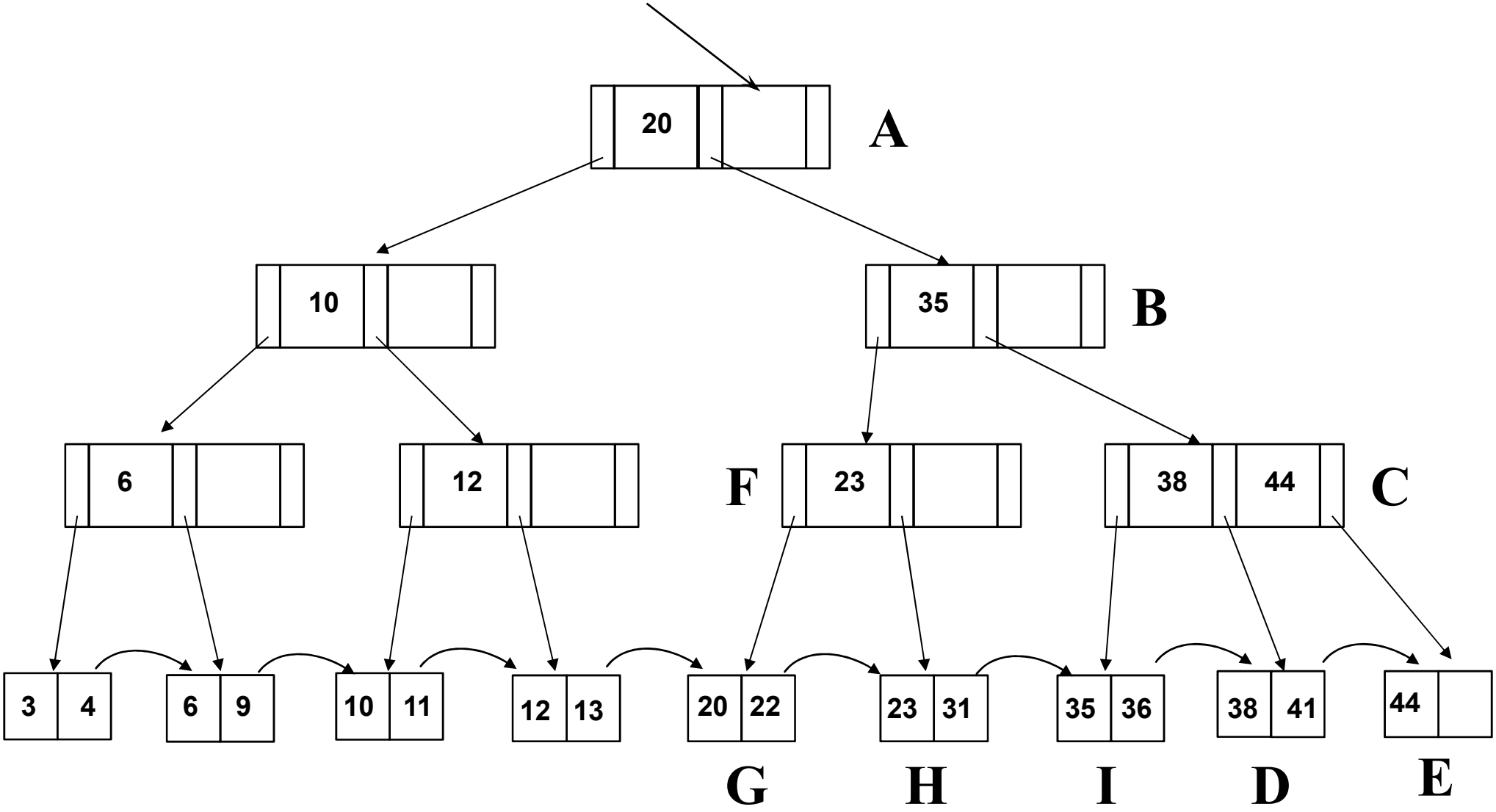
B+-tree Latching (Read 38)



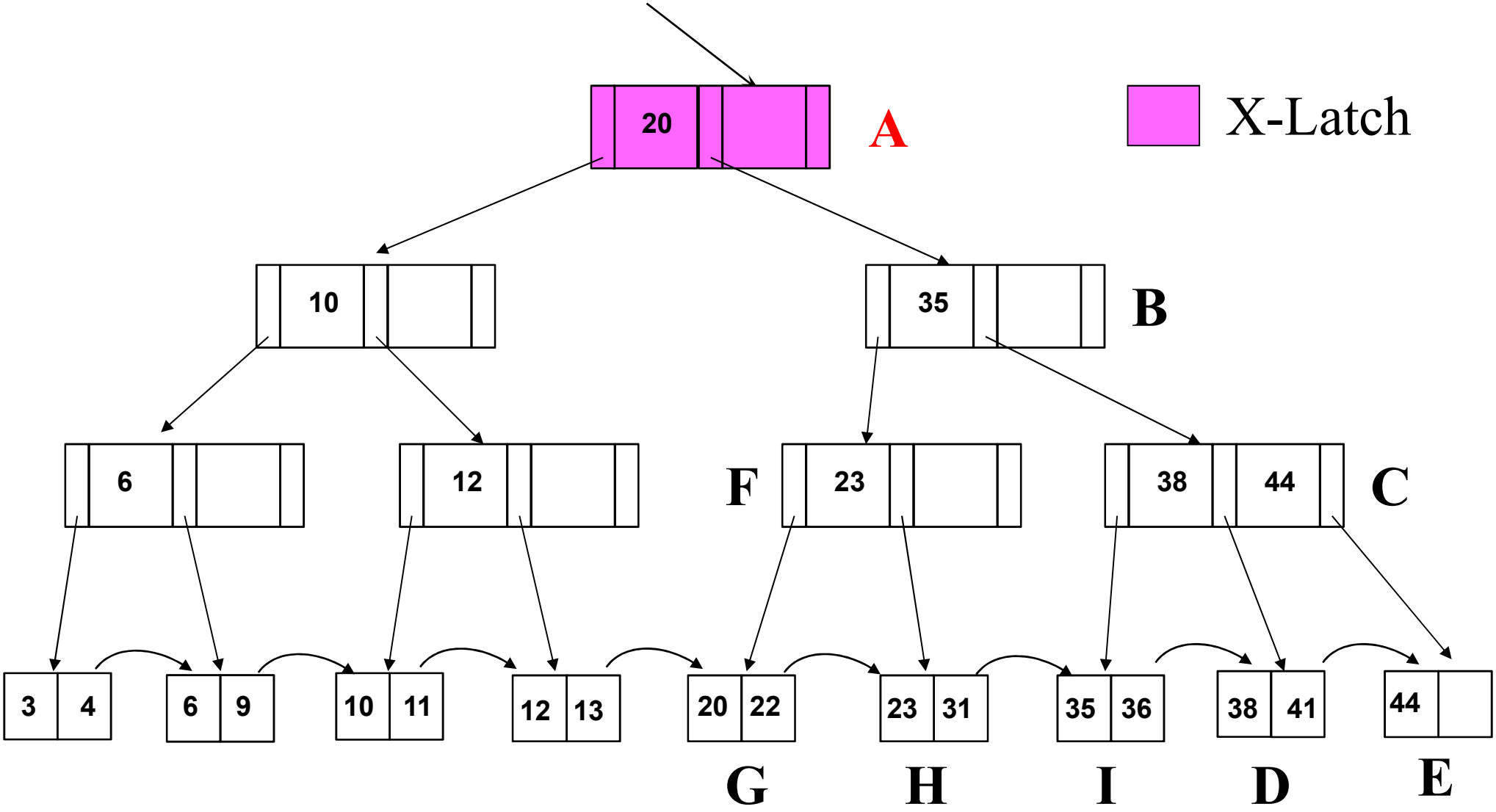
B+-tree Latching (Read 38)



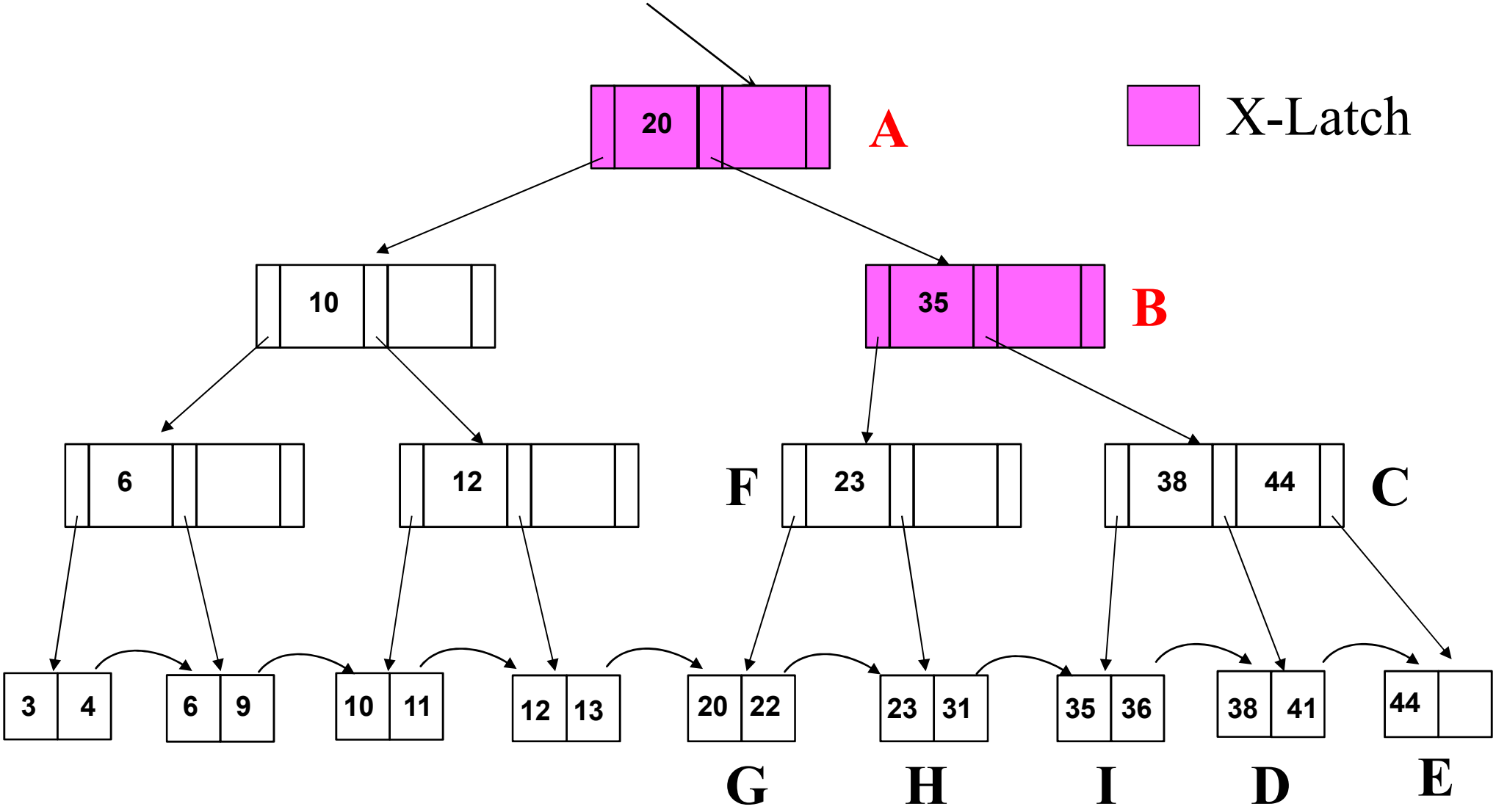
B+-tree Latching (Insert 45)



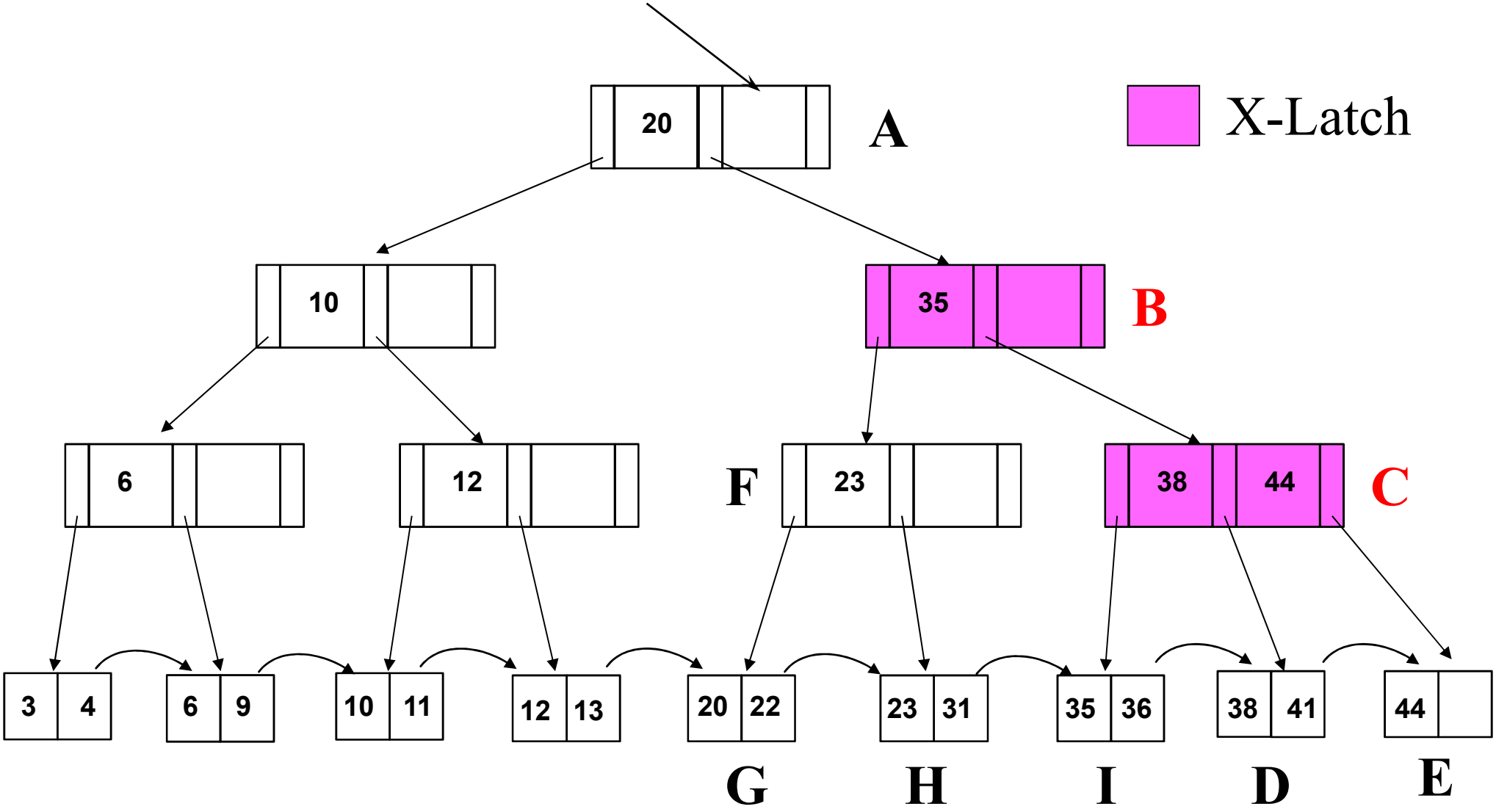
B+-tree Latching (Insert 45)



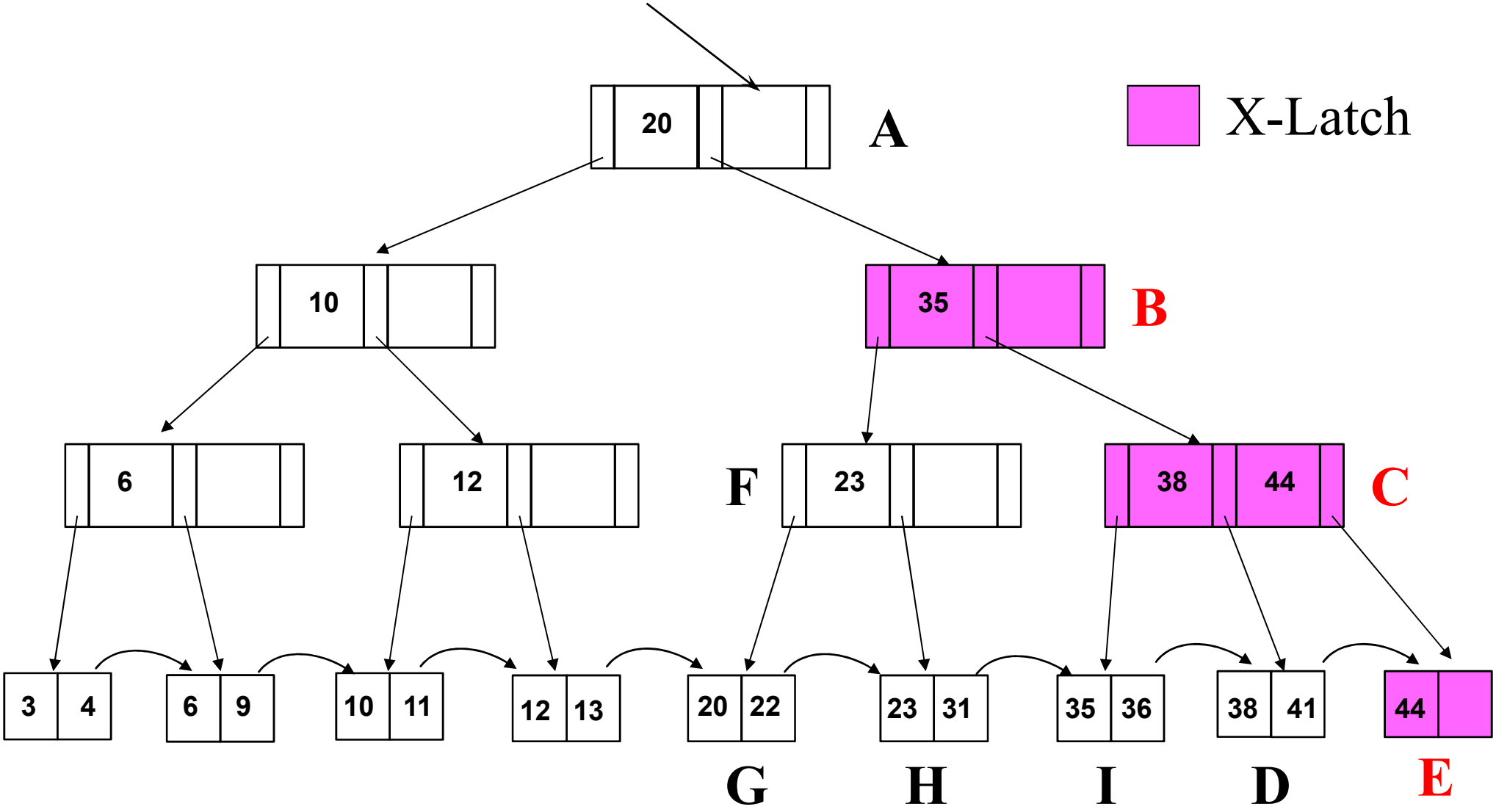
B+-tree Latching (Insert 45)



B+-tree Latching (Insert 45)

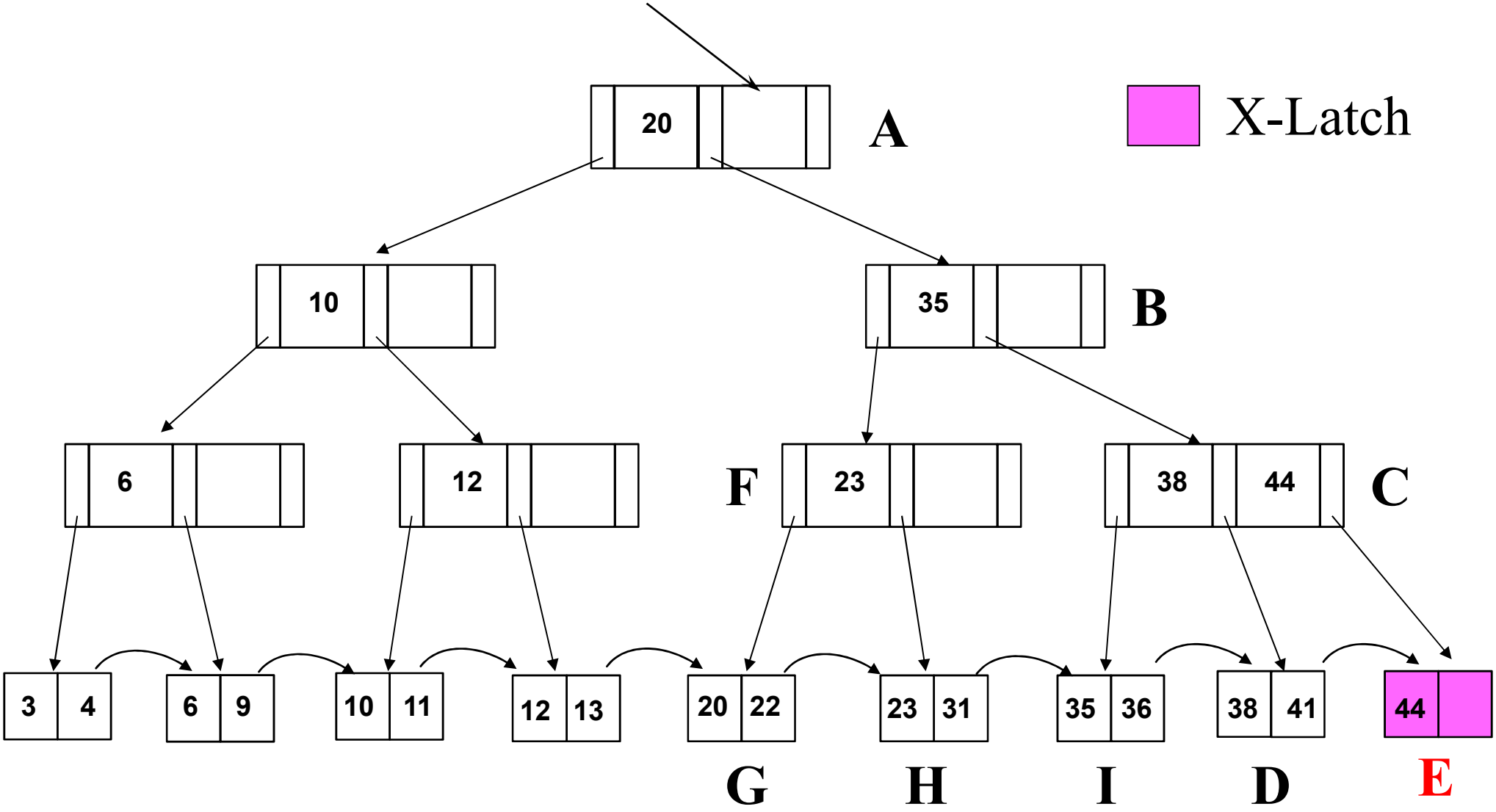


B+-tree Latching (Insert 45)

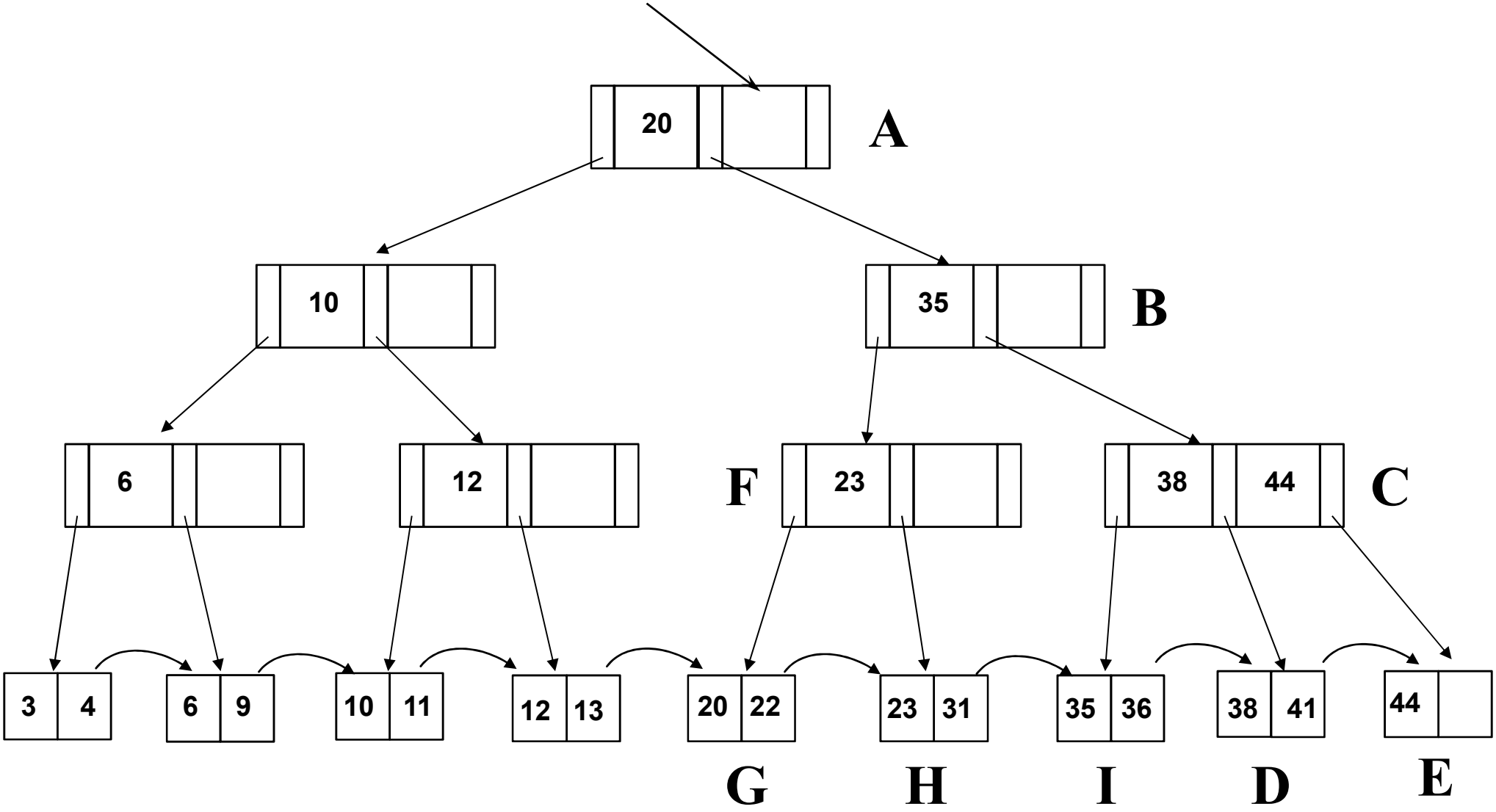


NOTE: B is not released. Why?

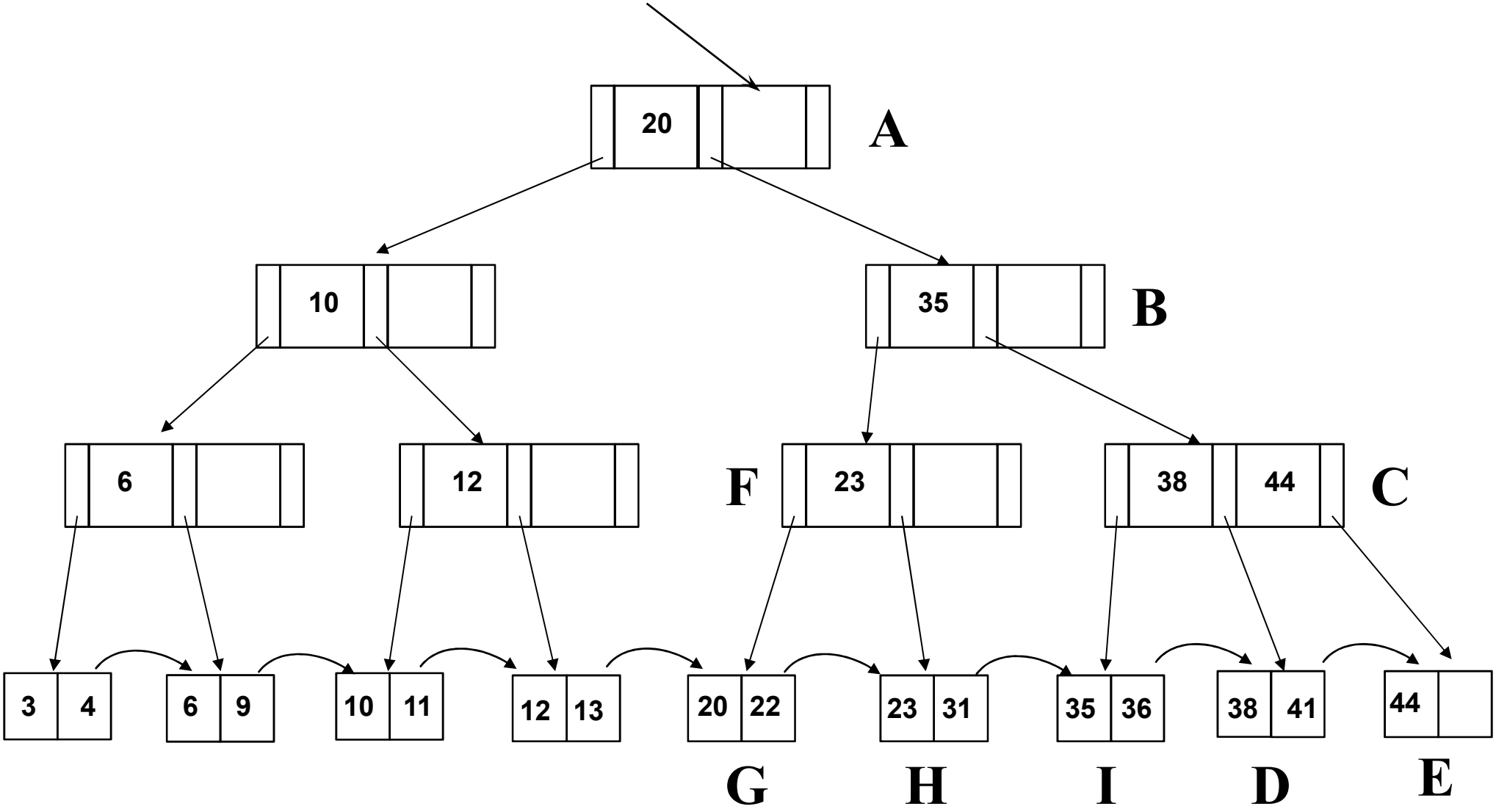
B+-tree Latching (Insert 45)



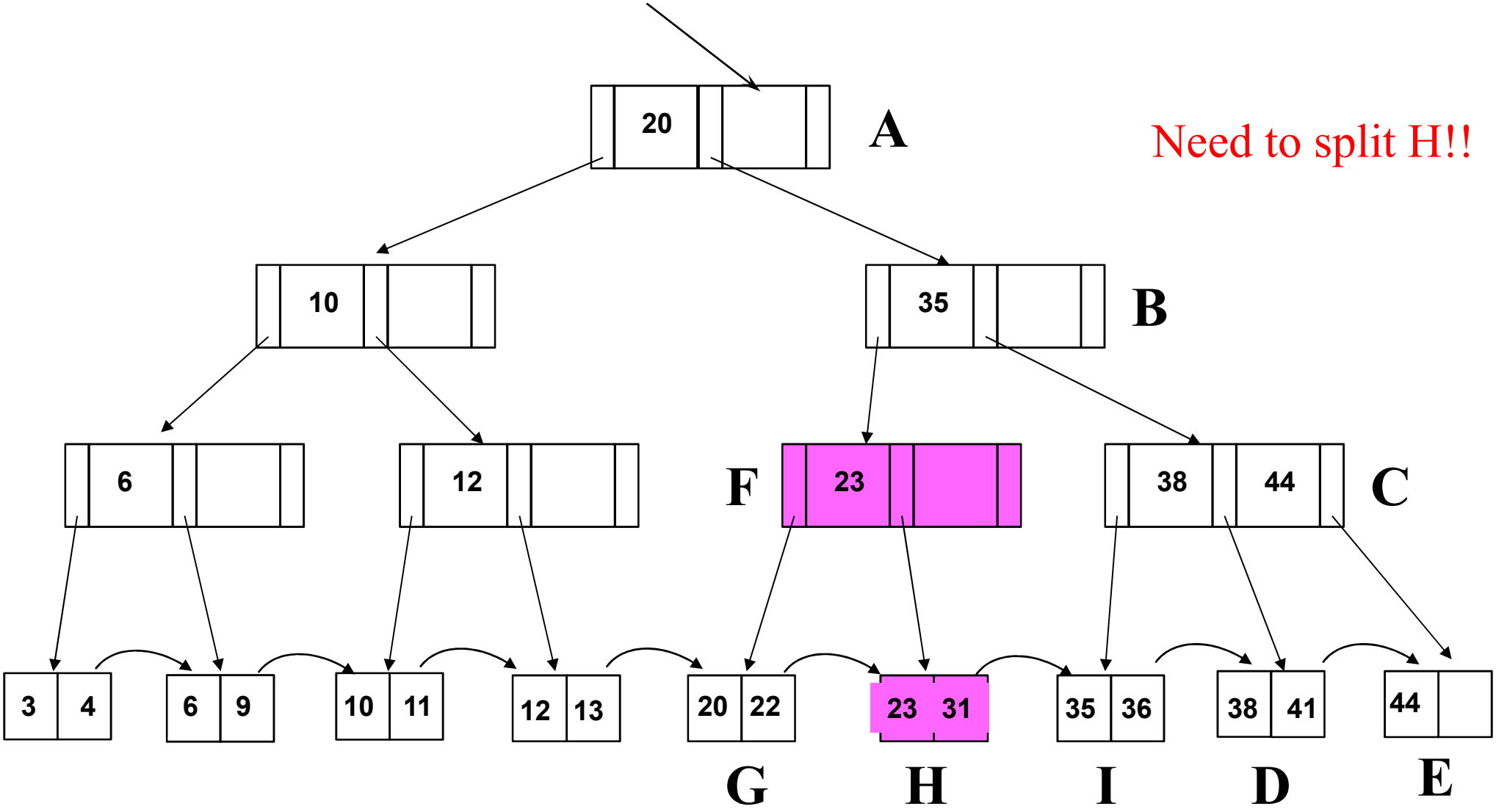
B+-tree Latching (Delete 38)



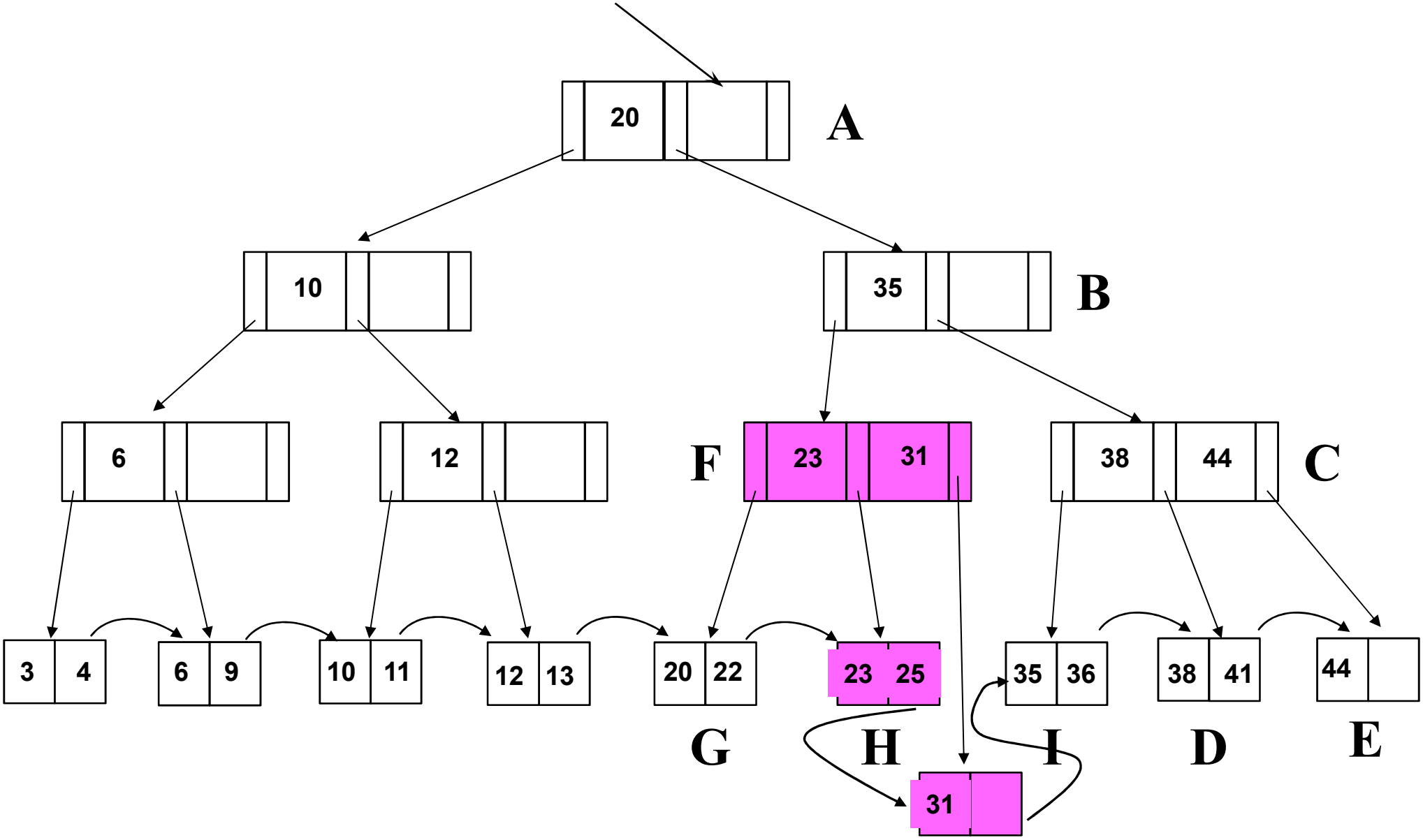
B+-tree Latching (Insert 25)



B+-tree Latching (Insert 25)



B+-tree Latching (Insert 25)



A Better B+-tree Latching Scheme

- Observation
  - Whenever we insert/delete, we apply a W latch on the root node
    - This is bad for achieving high degree of concurrency
  - Split/merge operations are uncommon (tree nodes typically have high fanout)
- Acquire R latch until leaf level; if the leaf is unsafe, then restart and revert to the basic latch coupling scheme (first pass to leaf wasted)

B+-tree Latching: Tree traversal

- When a Xact traverses a tree top-down, it either acquires the latch of its child node or wait
  - If the desired latch is unavailable, it waits
- There won't be any deadlocks

B+-tree Latching: Leaf traversal

- When a Xact traverses from one leaf node to another, deadlocks are possible (if you allow traversal from left-to-right, and right-to-left)
  - Index latches do not support deadlock detection or avoidance
- *So, a "no-wait" scheme must be adopted*, i.e., the Xact/thread will abort, release all latches it held and restart

Summary

- 2PL is *not* used for index locking
- Latch coupling is used to facilitate B+-tree concurrency
- Deletion can be done “efficiently” at the expense of violating the minimum utilization requirement