Concurrent Programming

Bw-Trees

Prof. Hyungsoo Jung

B+TREE CONCURRENCY CONTROL

We want to allow multiple threads to read and update a B+Tree 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.**

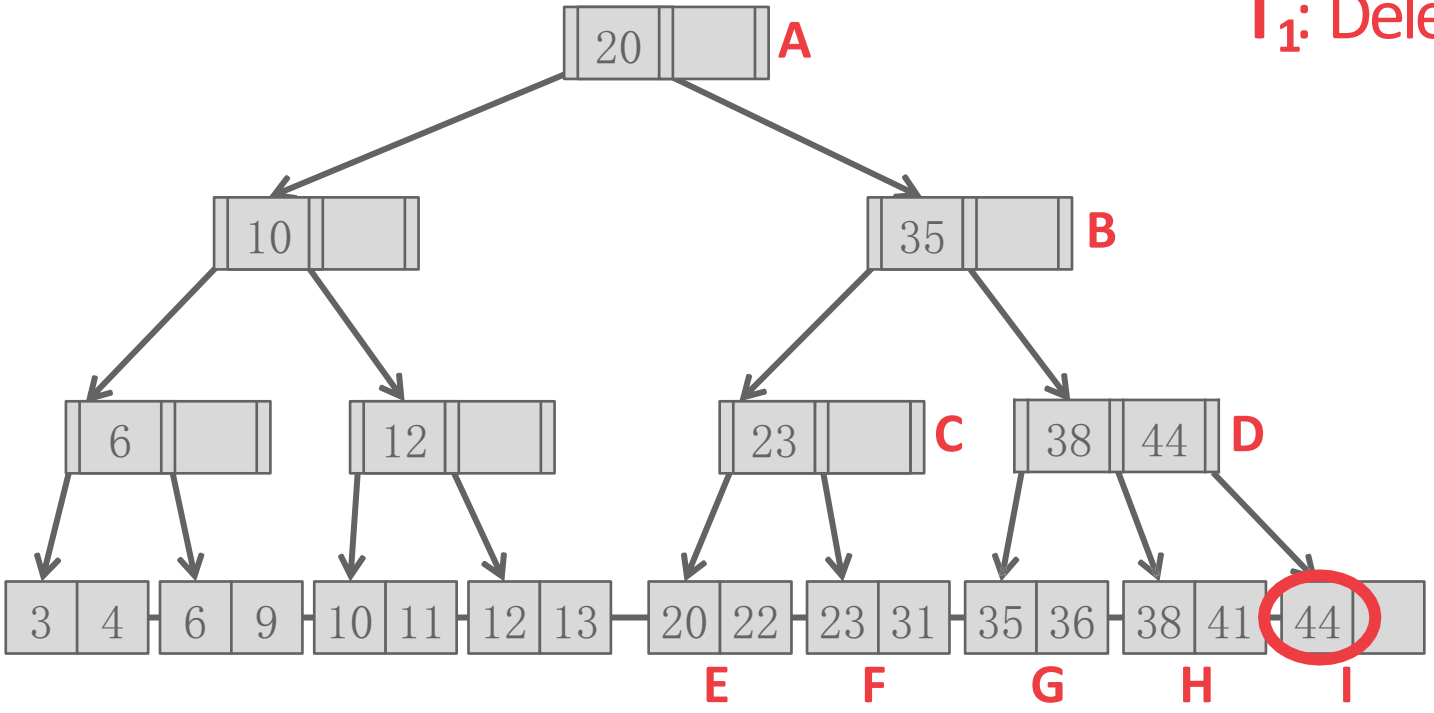
→ **One thread traversing the tree while another thread splits/merges nodes (e.g., SMO: structure modification ops).**

OBSERVATION

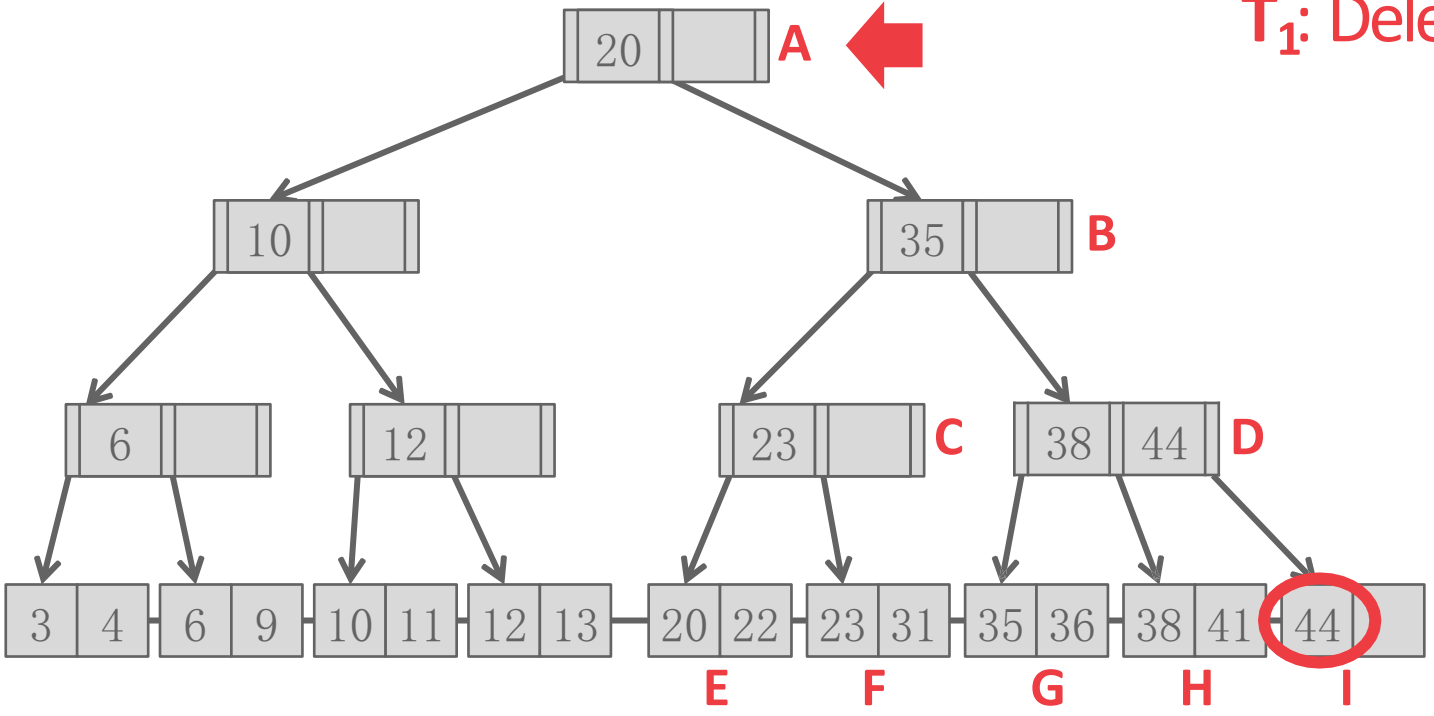
Making a data structure thread-safe is notoriously difficult in practice.

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

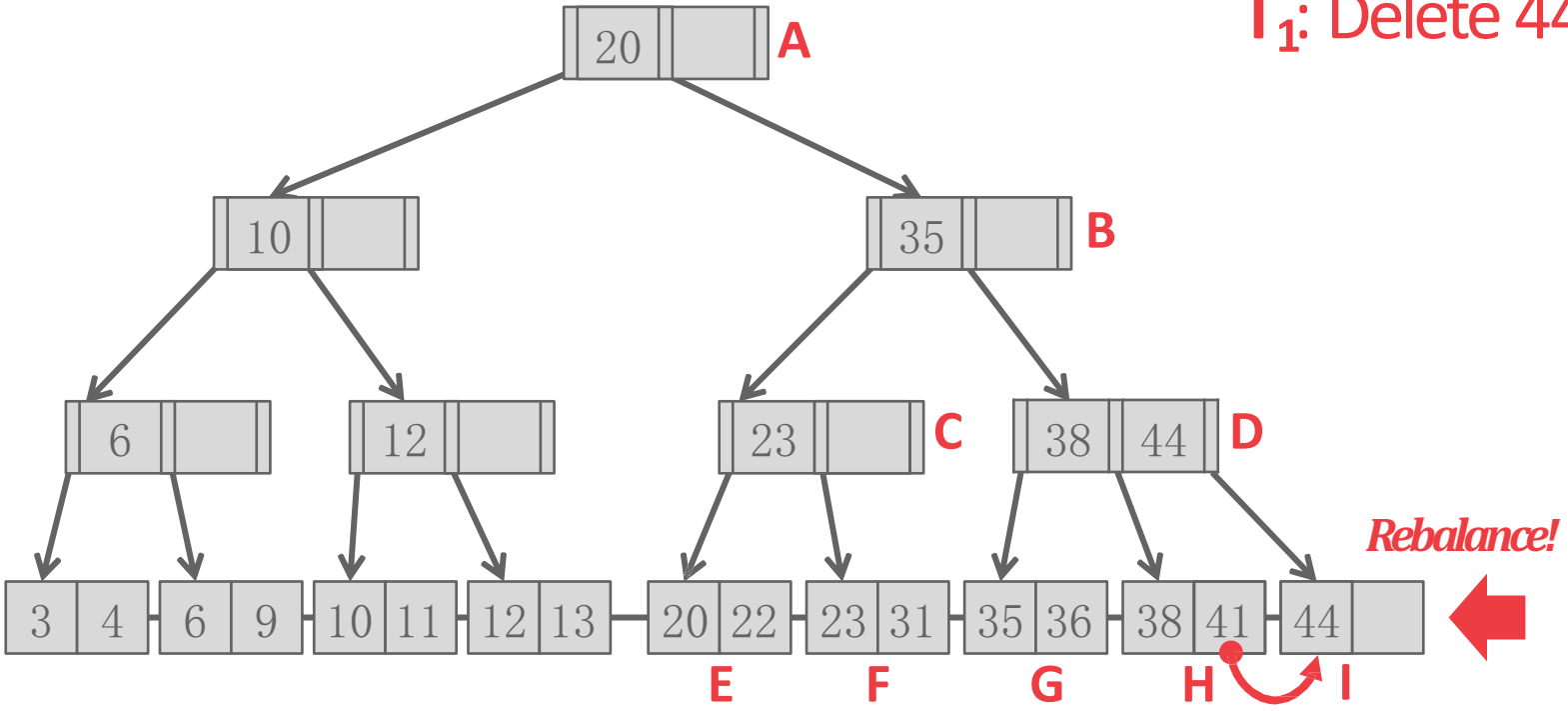
B+TREE MULTI-THREADED EXAMPLE



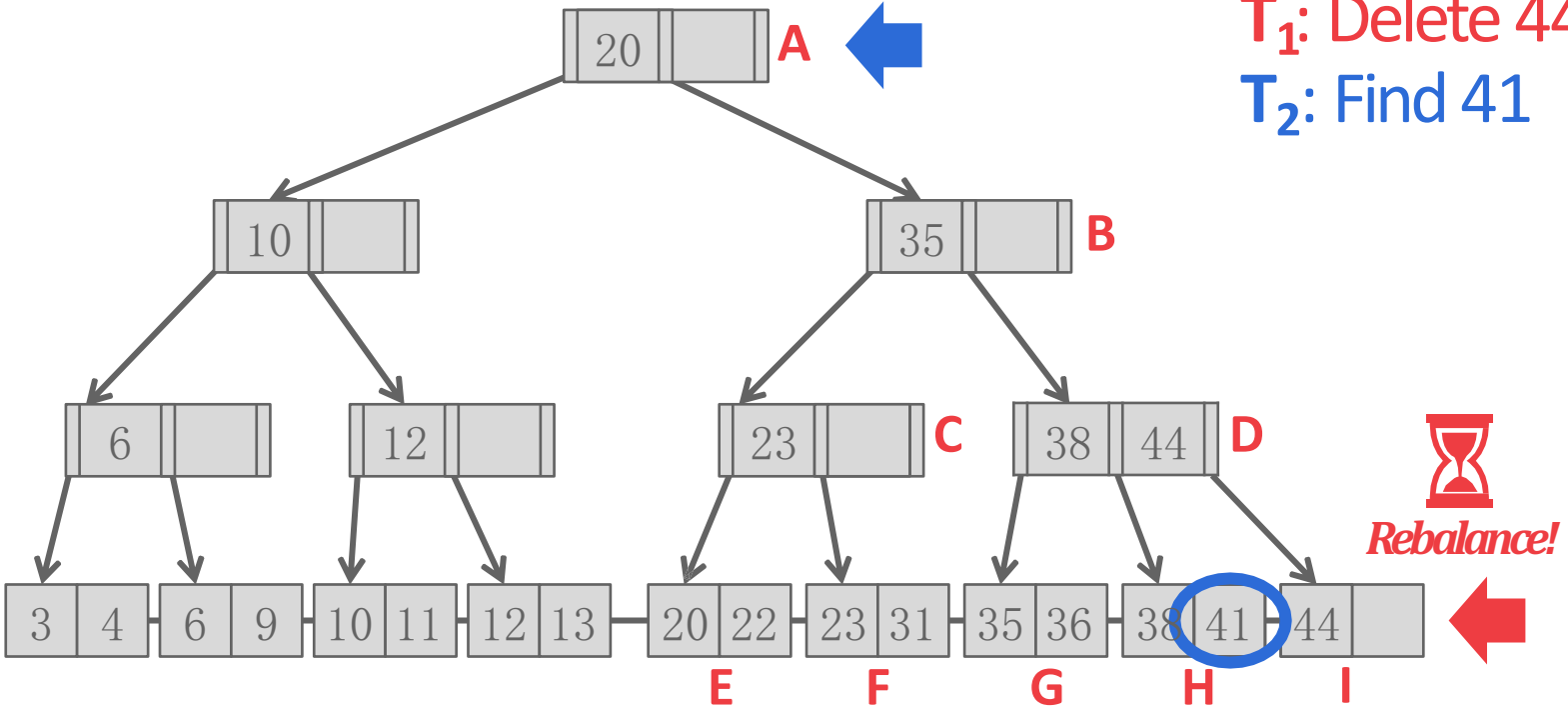
B+TREE MULTI-THREADED EXAMPLE



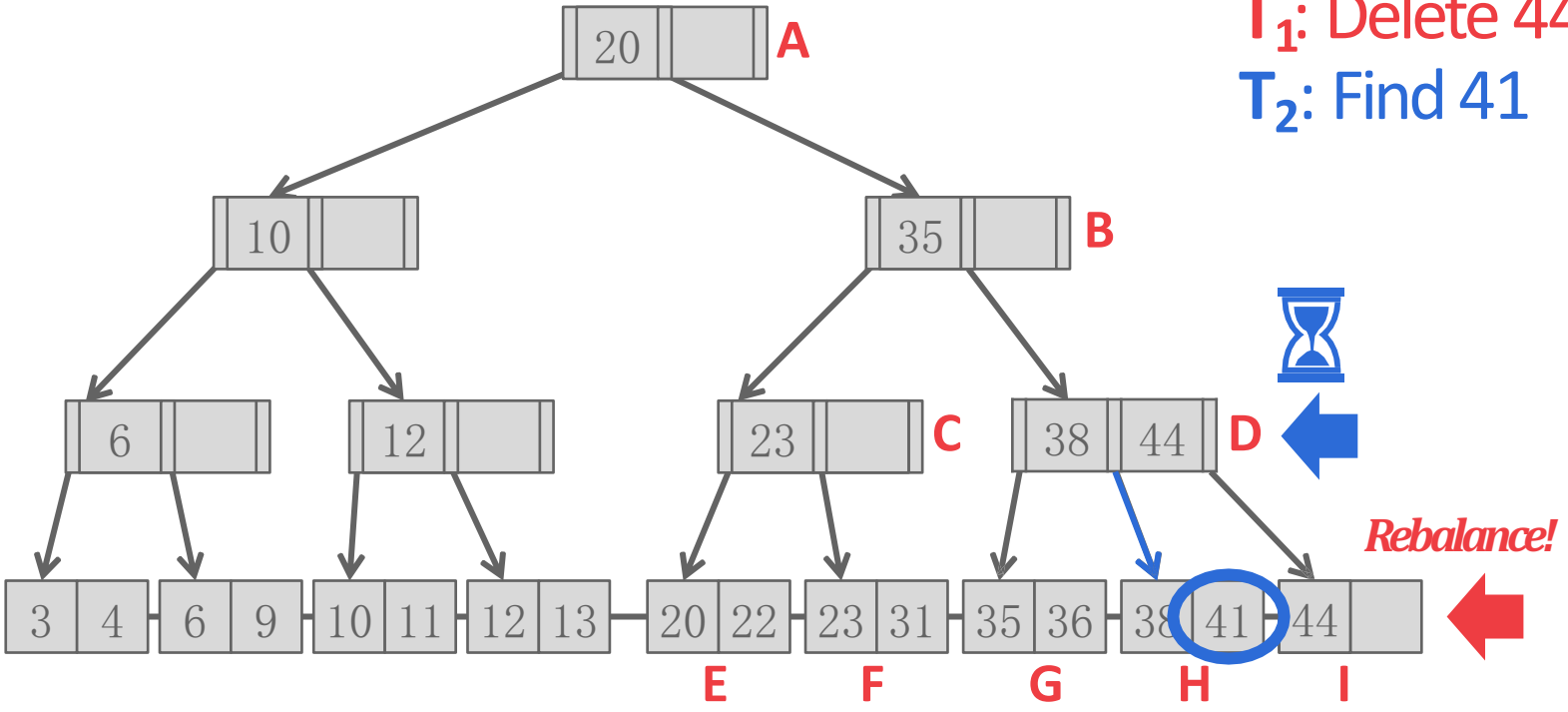
B+TREE MULTI-THREADED EXAMPLE



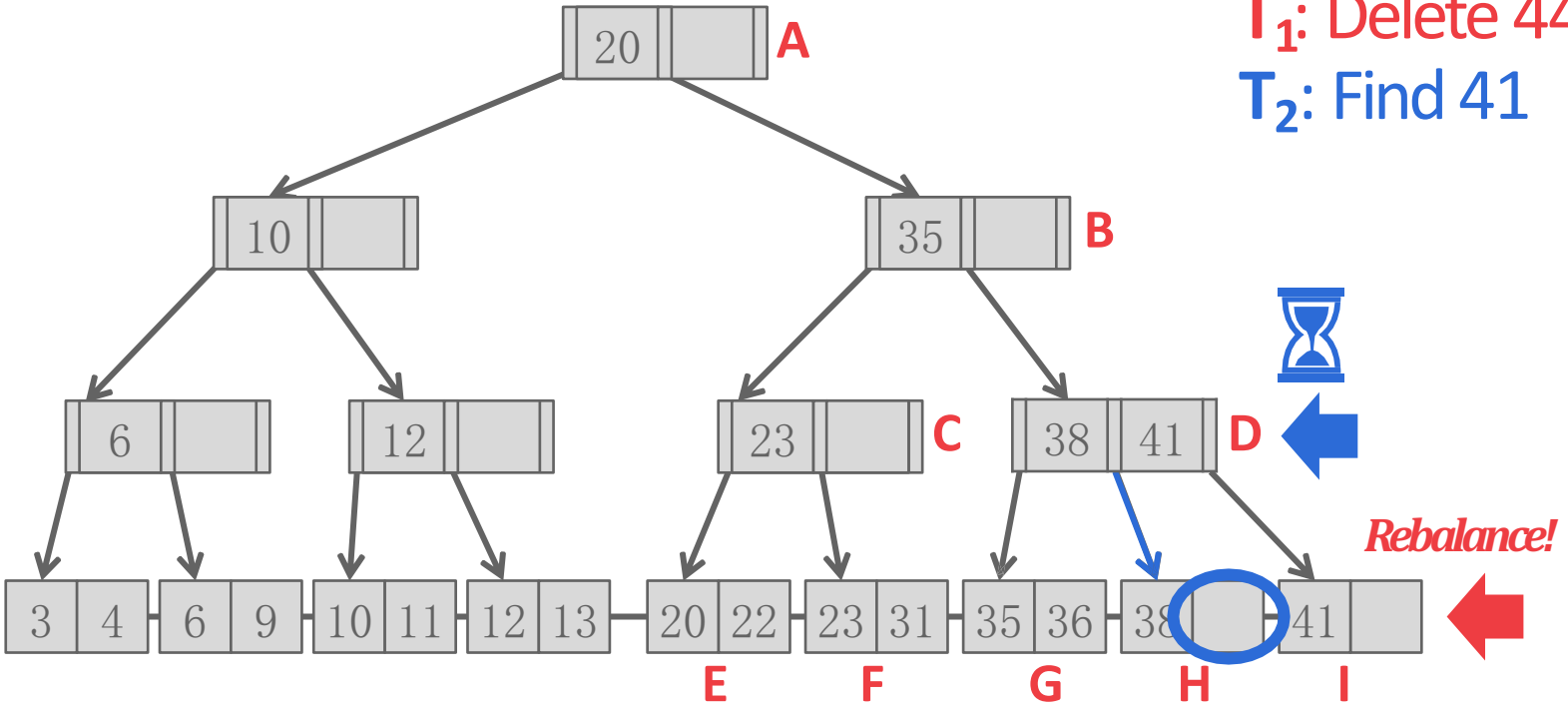
B+TREE MULTI-THREADED EXAMPLE



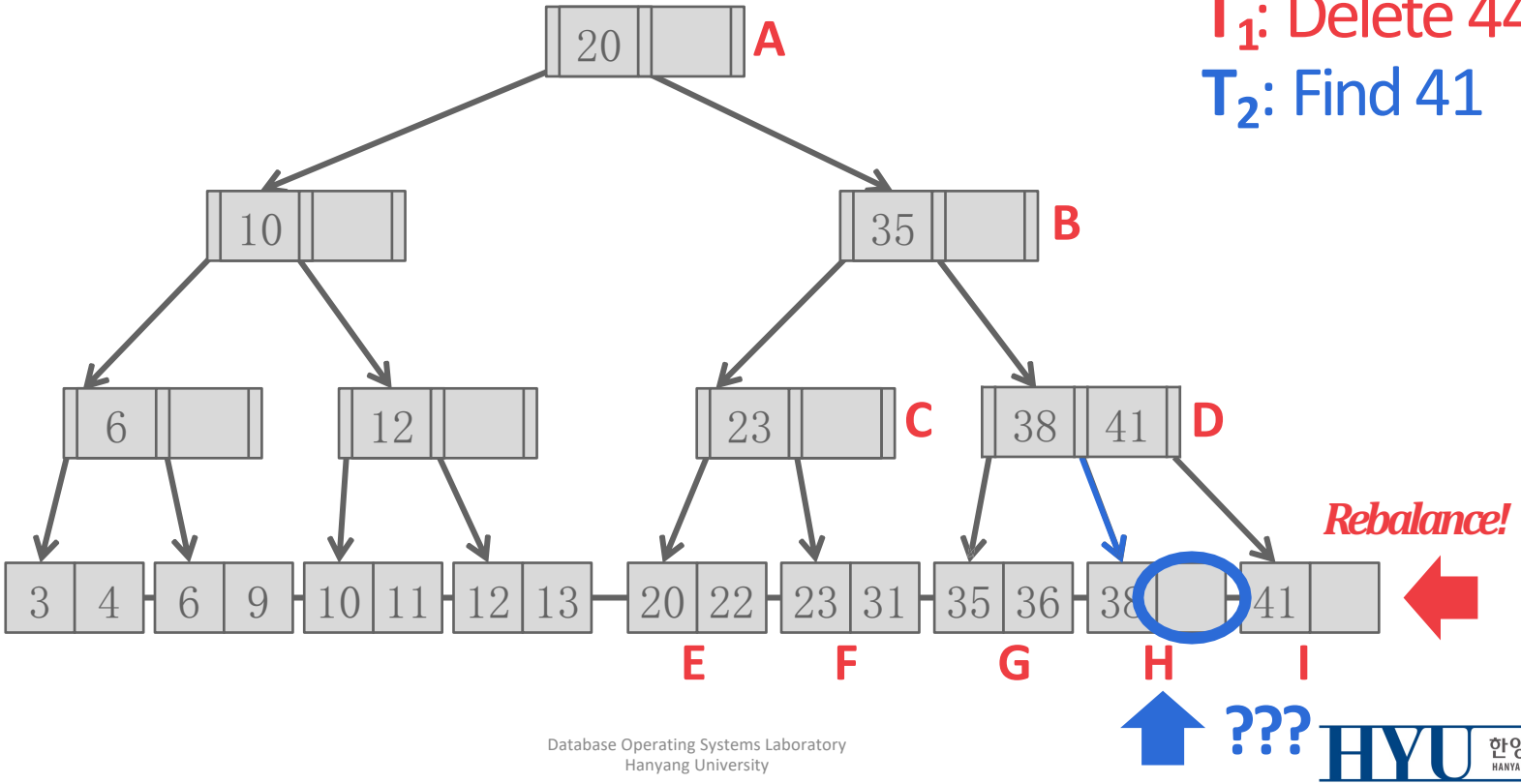
B+TREE MULTI-THREADED EXAMPLE



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

Find: Start at root and go down; repeatedly,

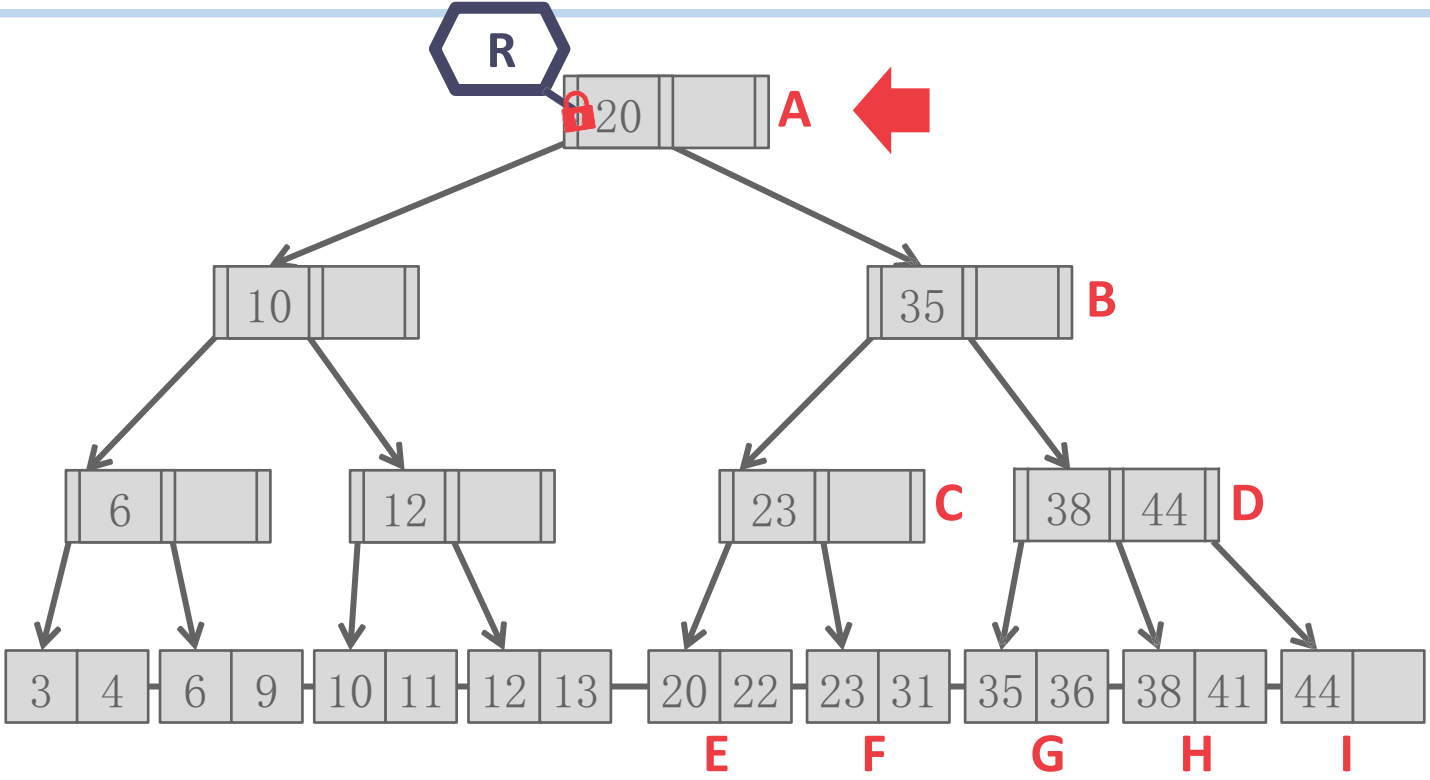
→ Acquire **R** latch on child

→ Then unlatch parent

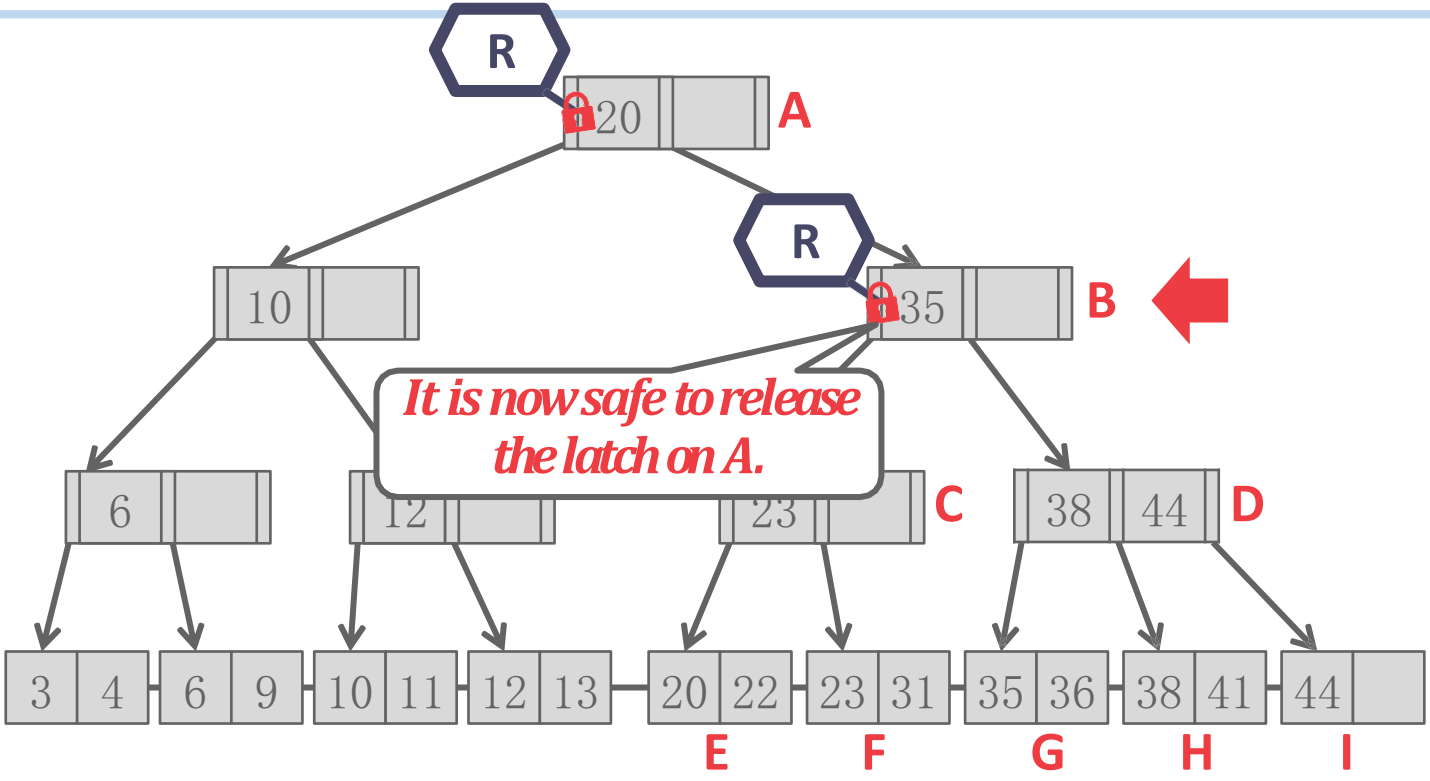
Insert/Delete: Start at root and go down, obtaining **W** latches as needed. Once child is latched, check if it is safe:

→ If child is safe, release all latches on ancestors.

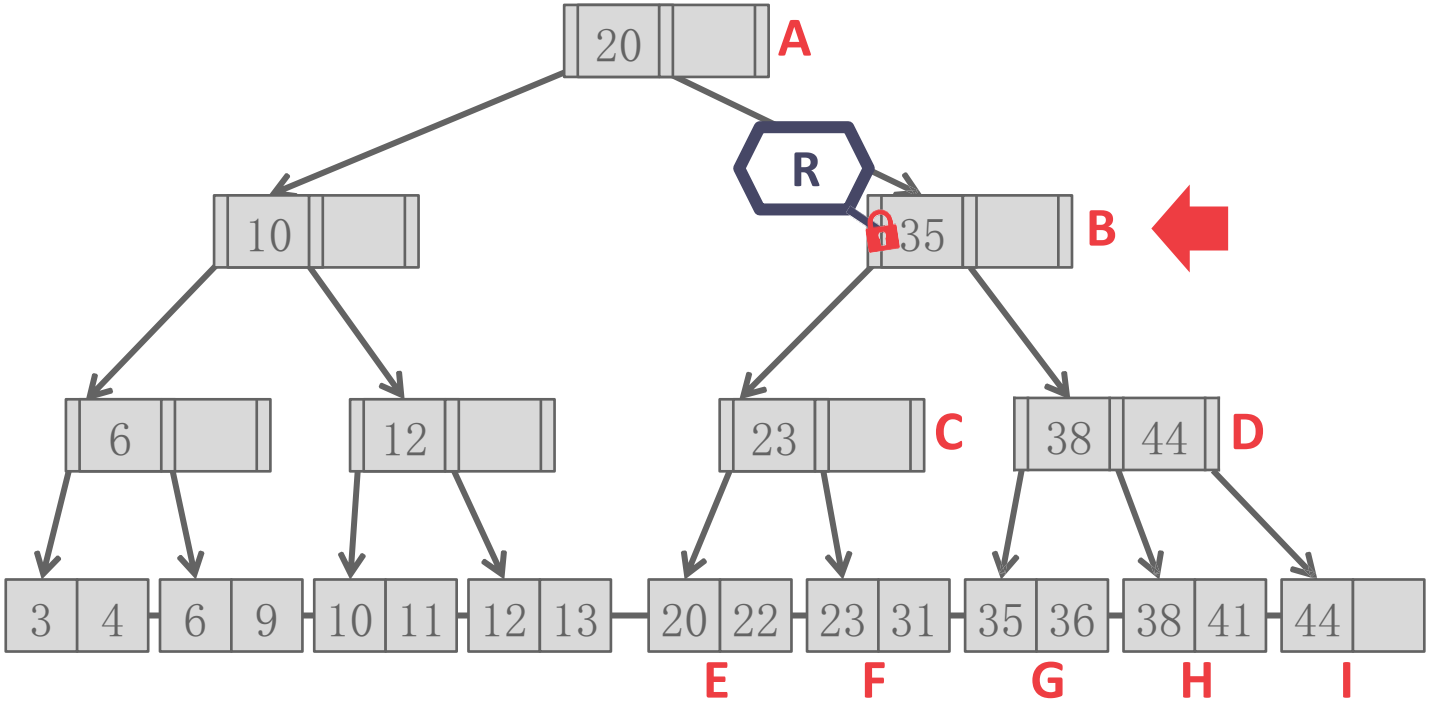
EXAMPLE #1 – FIND 38



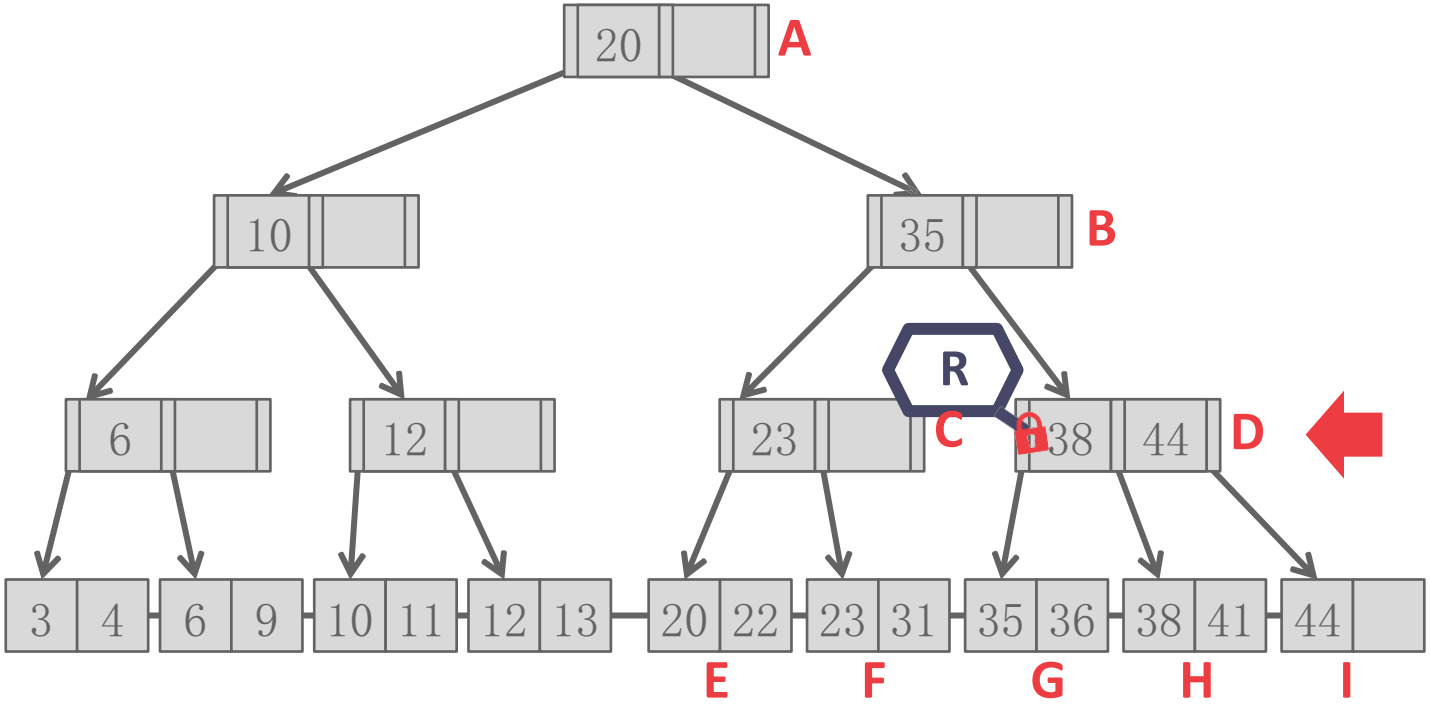
EXAMPLE #1 – FIND 38



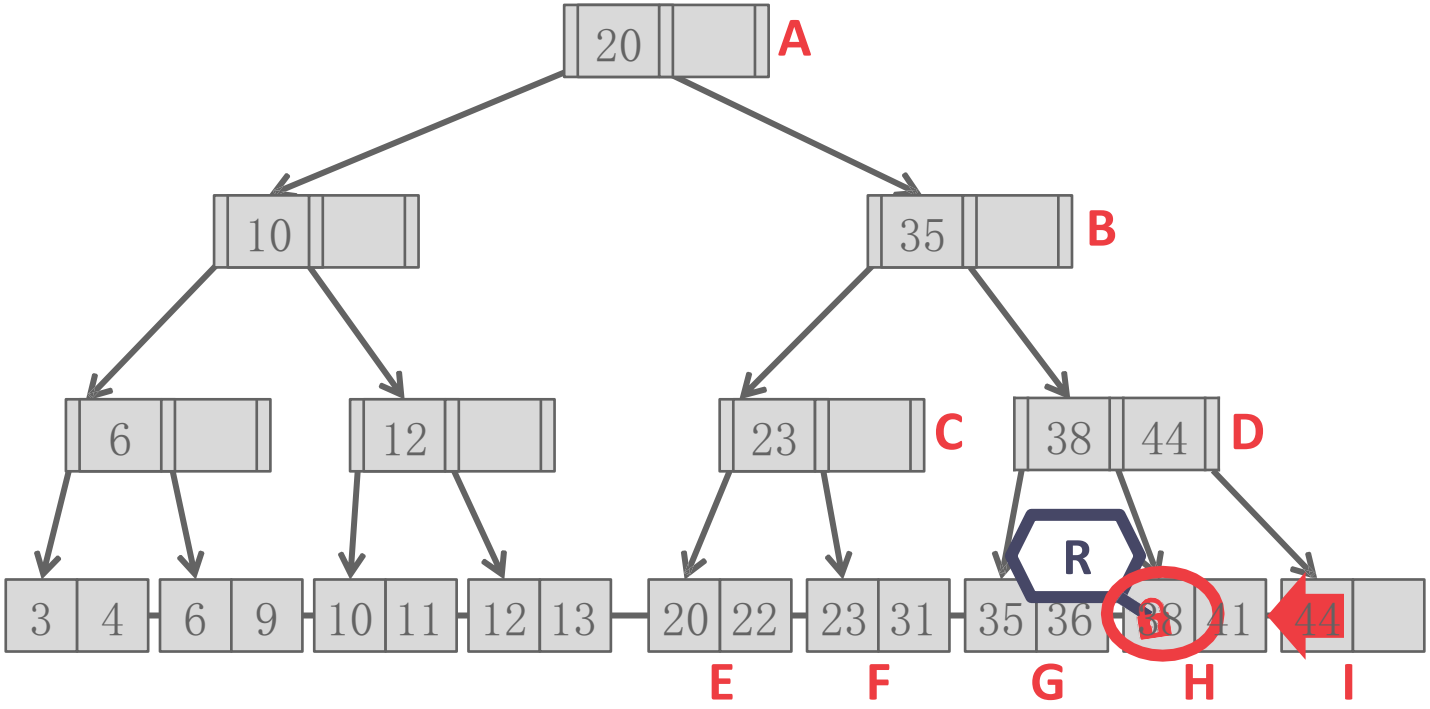
EXAMPLE #1 – FIND 38



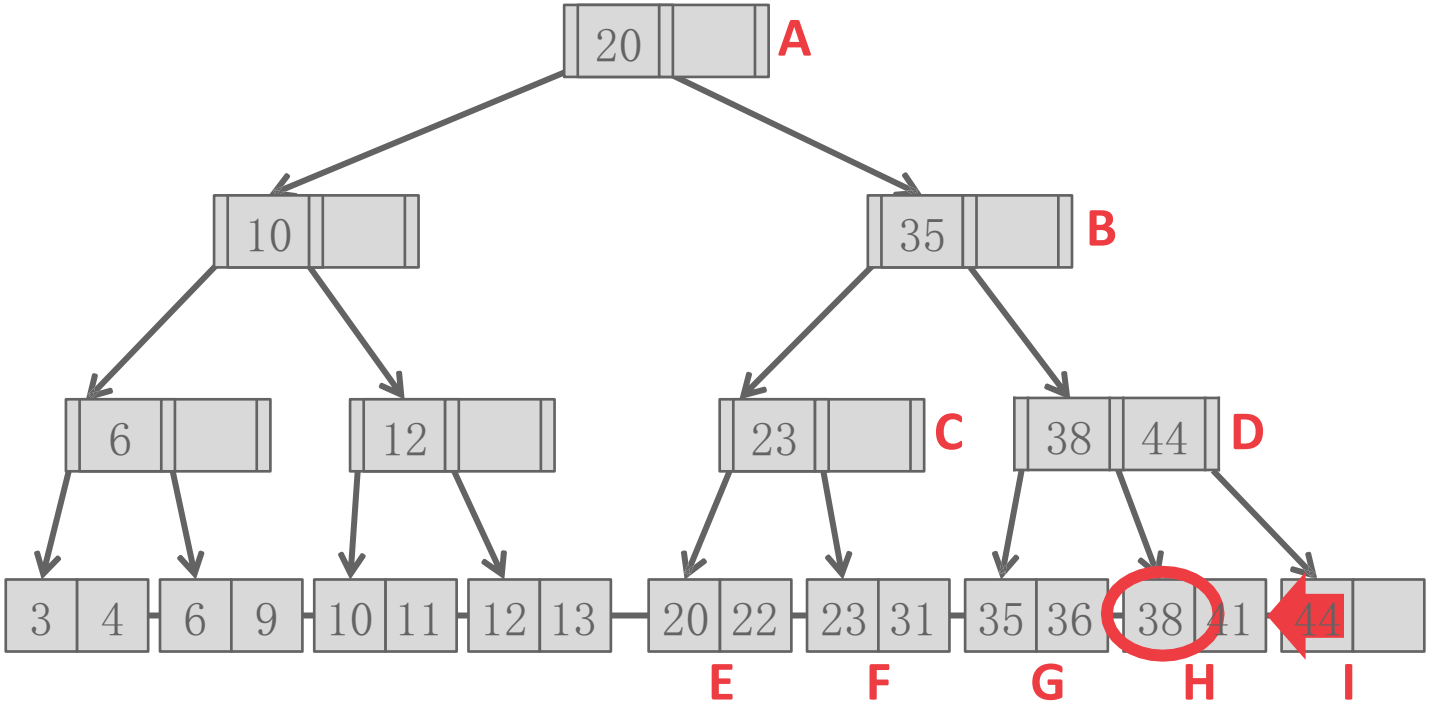
EXAMPLE #1 – FIND 38



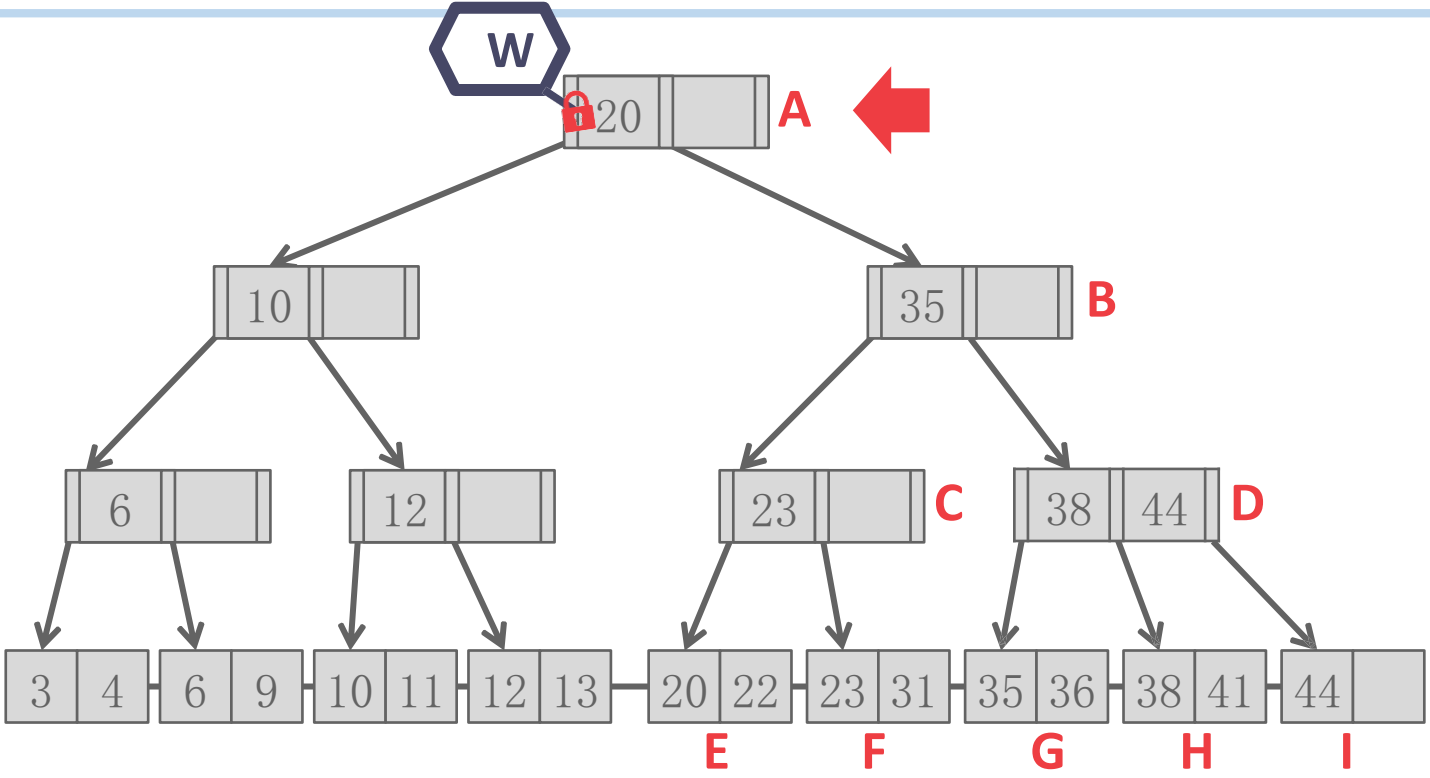
EXAMPLE #1 – FIND 38



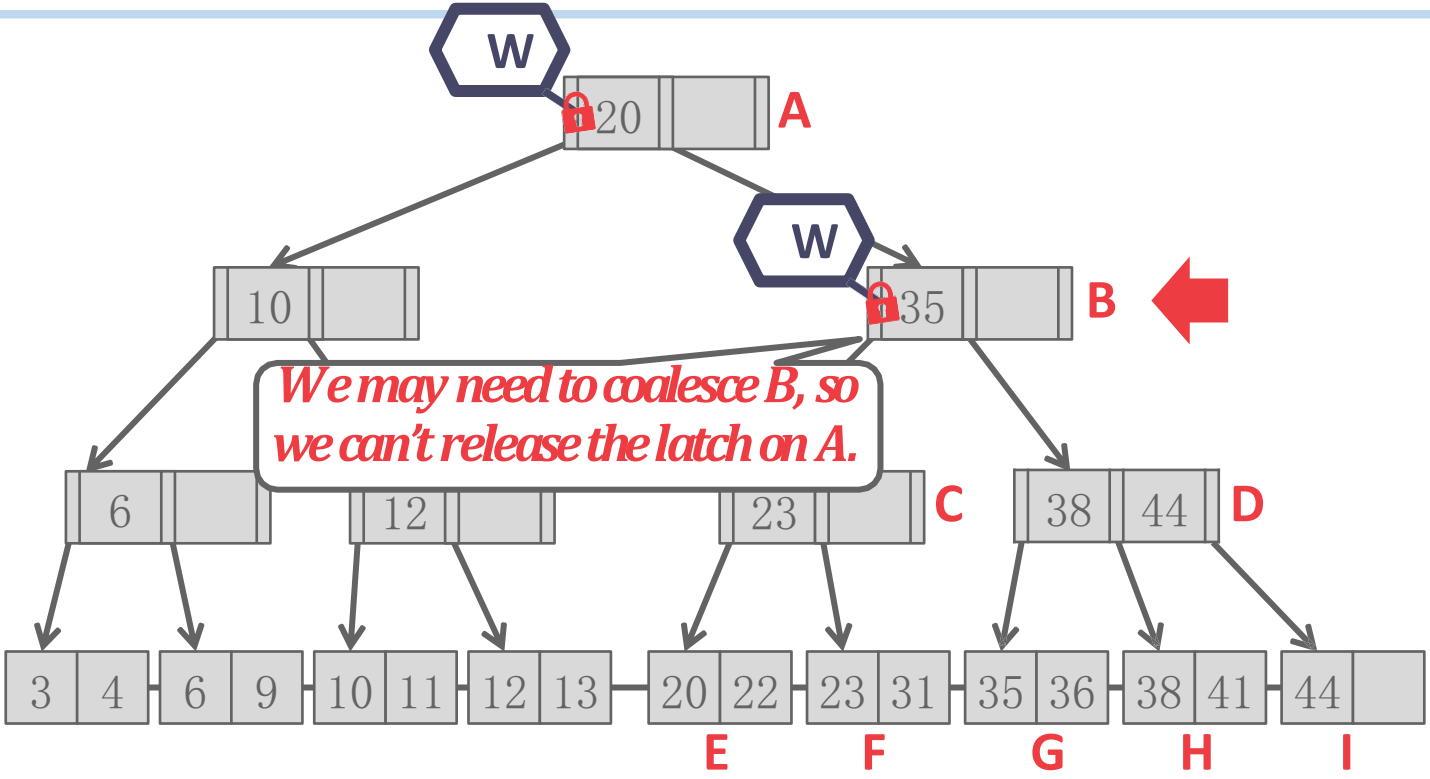
EXAMPLE #1 – FIND 38



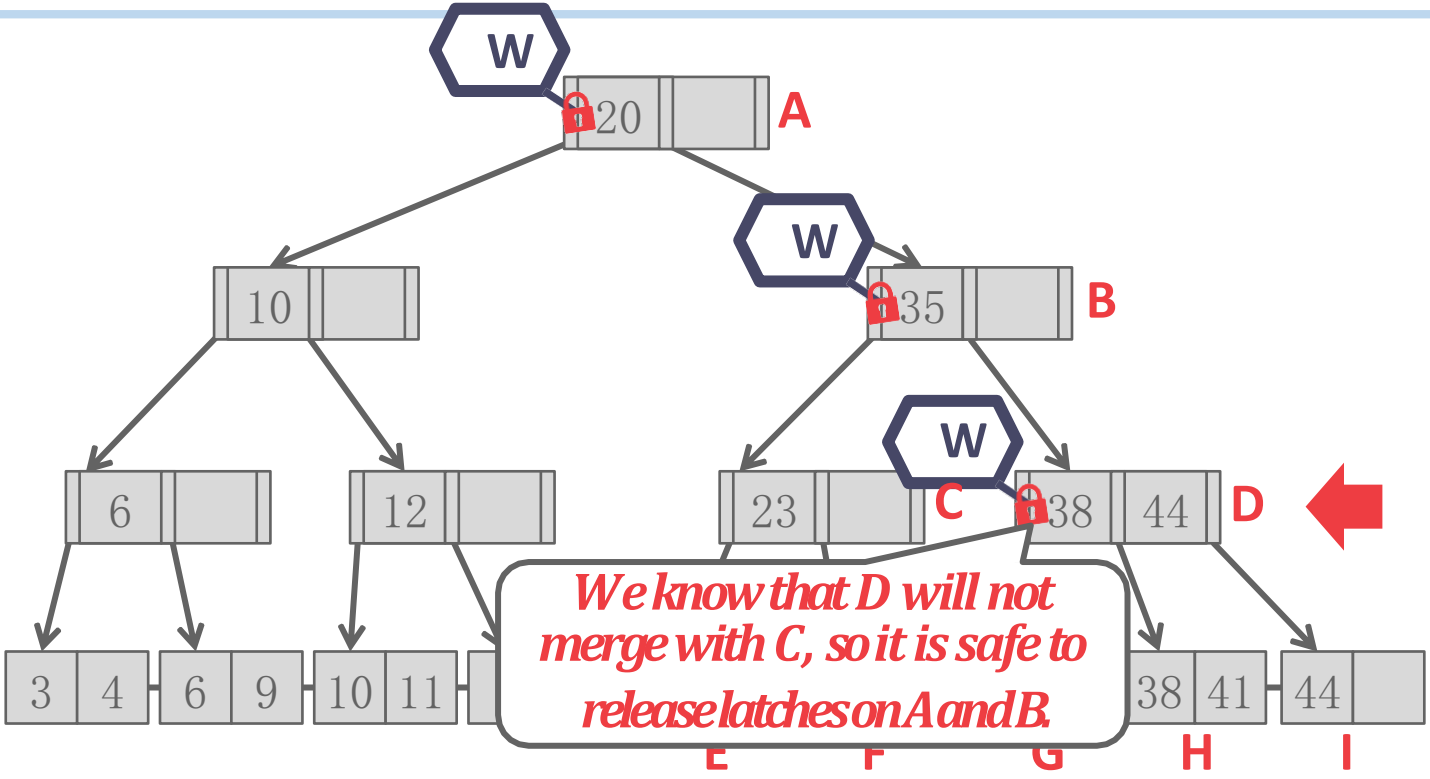
EXAMPLE #2 – DELETE 38



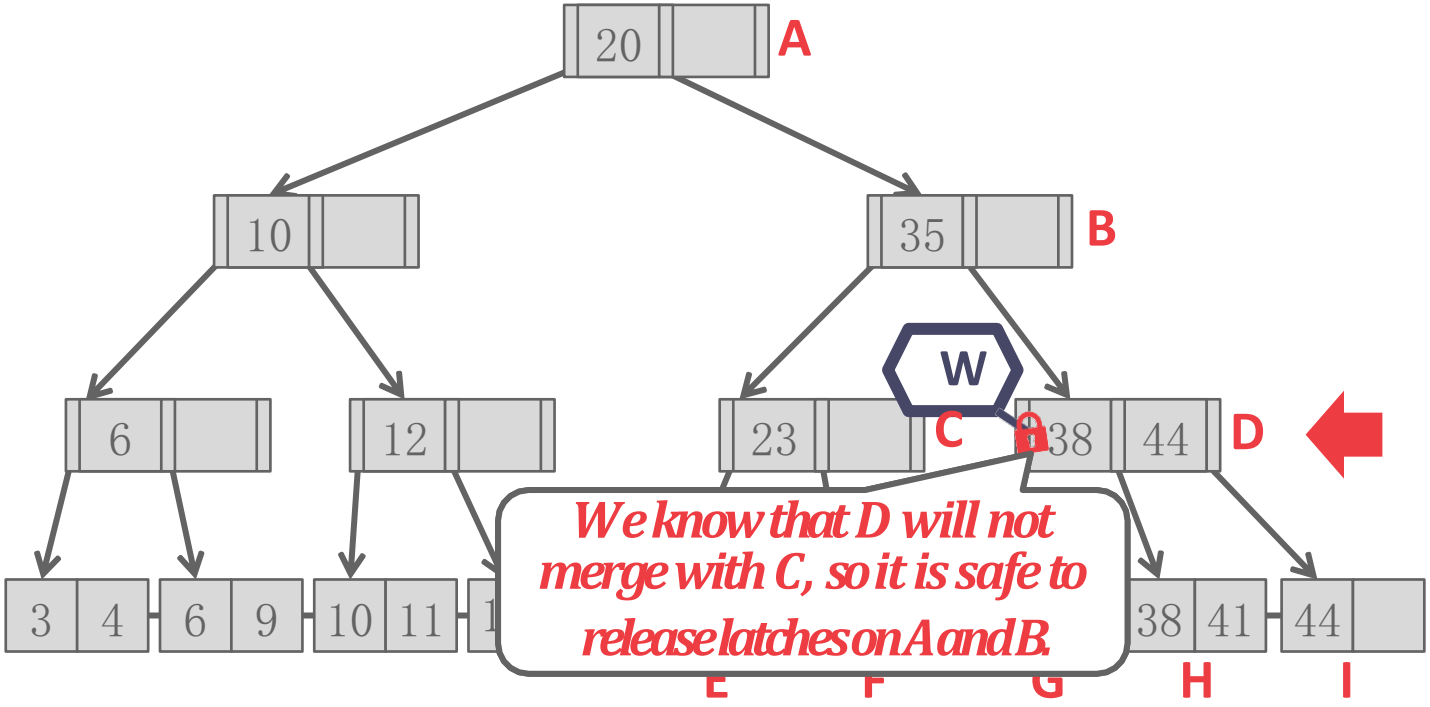
EXAMPLE #2 – DELETE 38



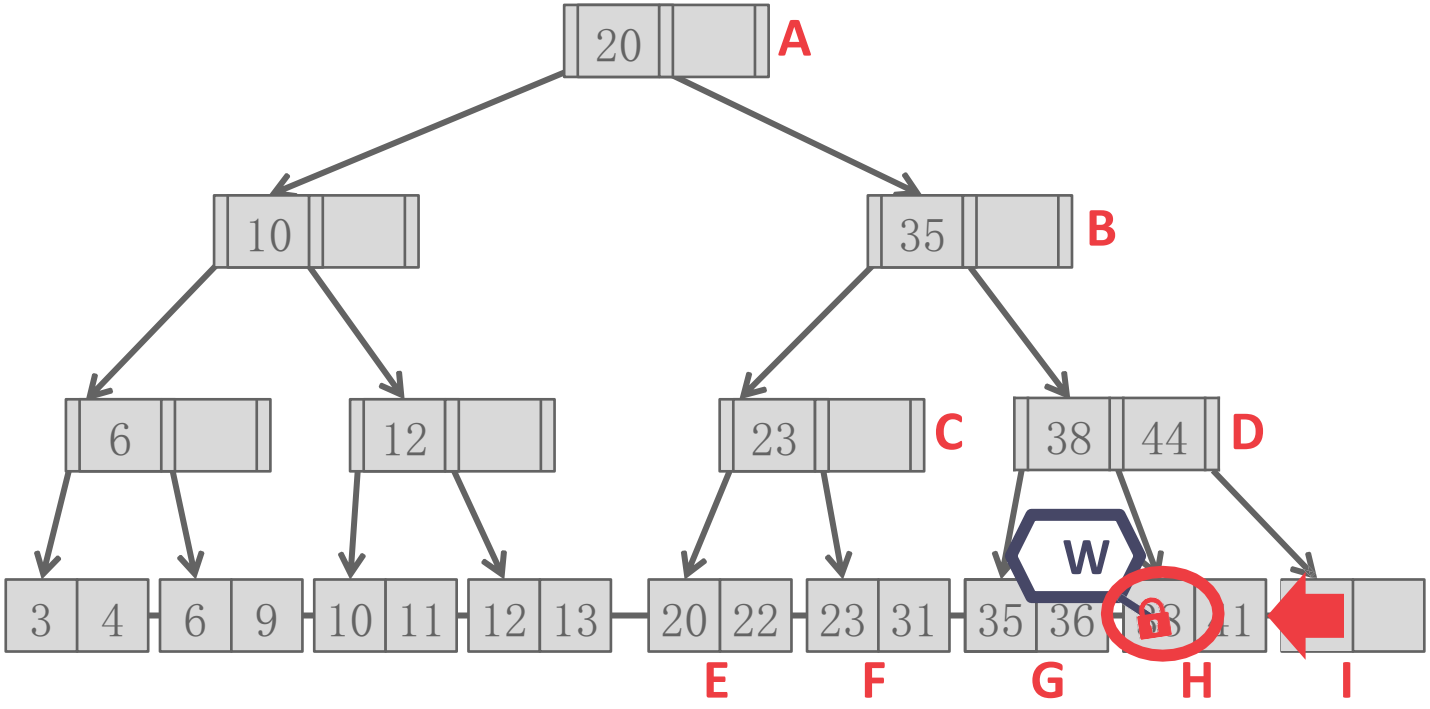
EXAMPLE #2 – DELETE 38



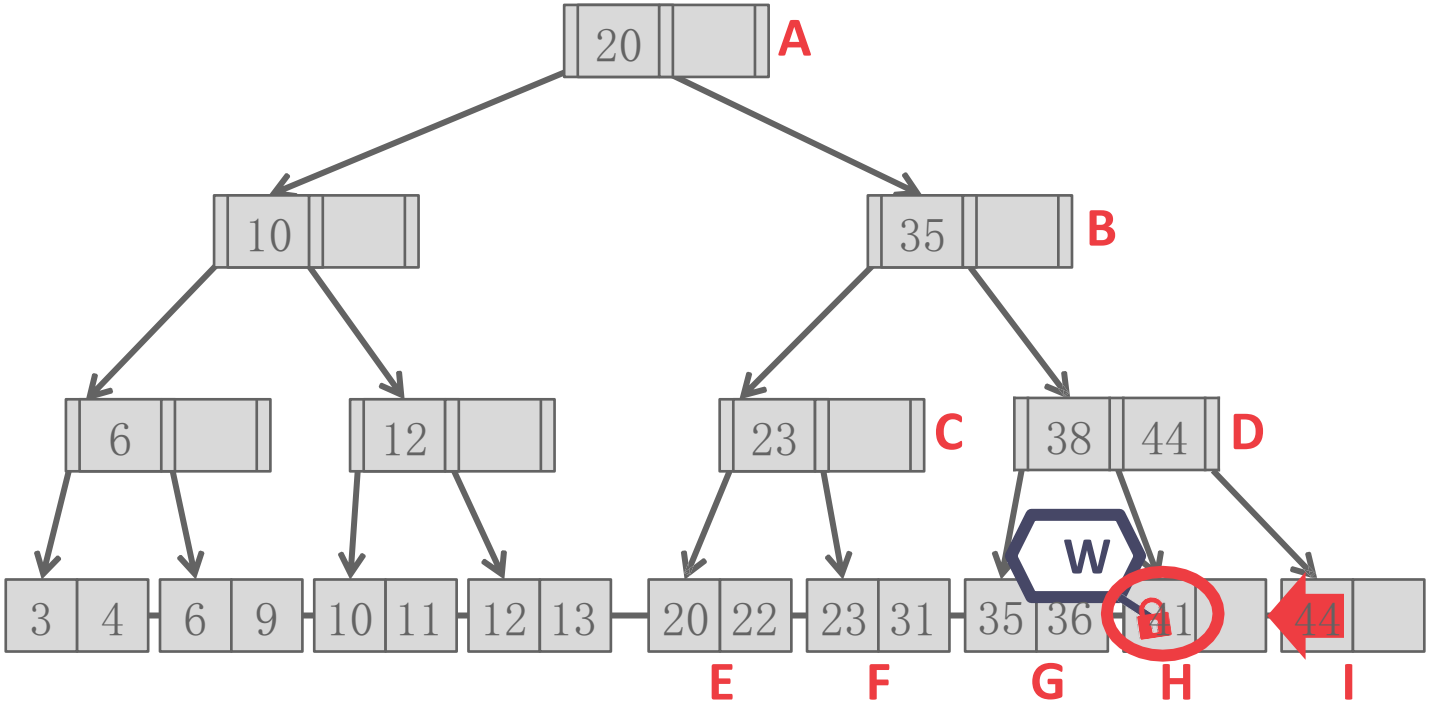
EXAMPLE #2 – DELETE 38



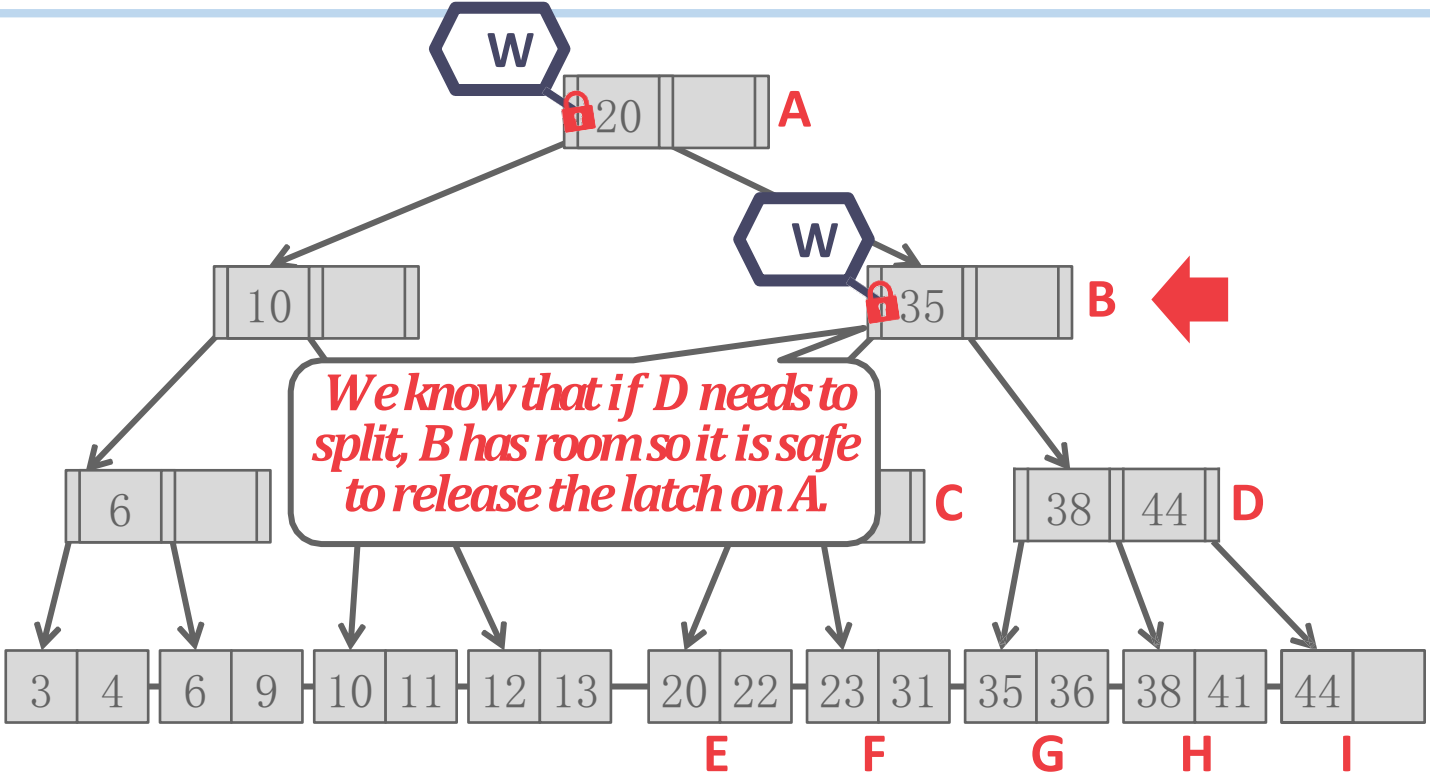
EXAMPLE #2 – DELETE 38



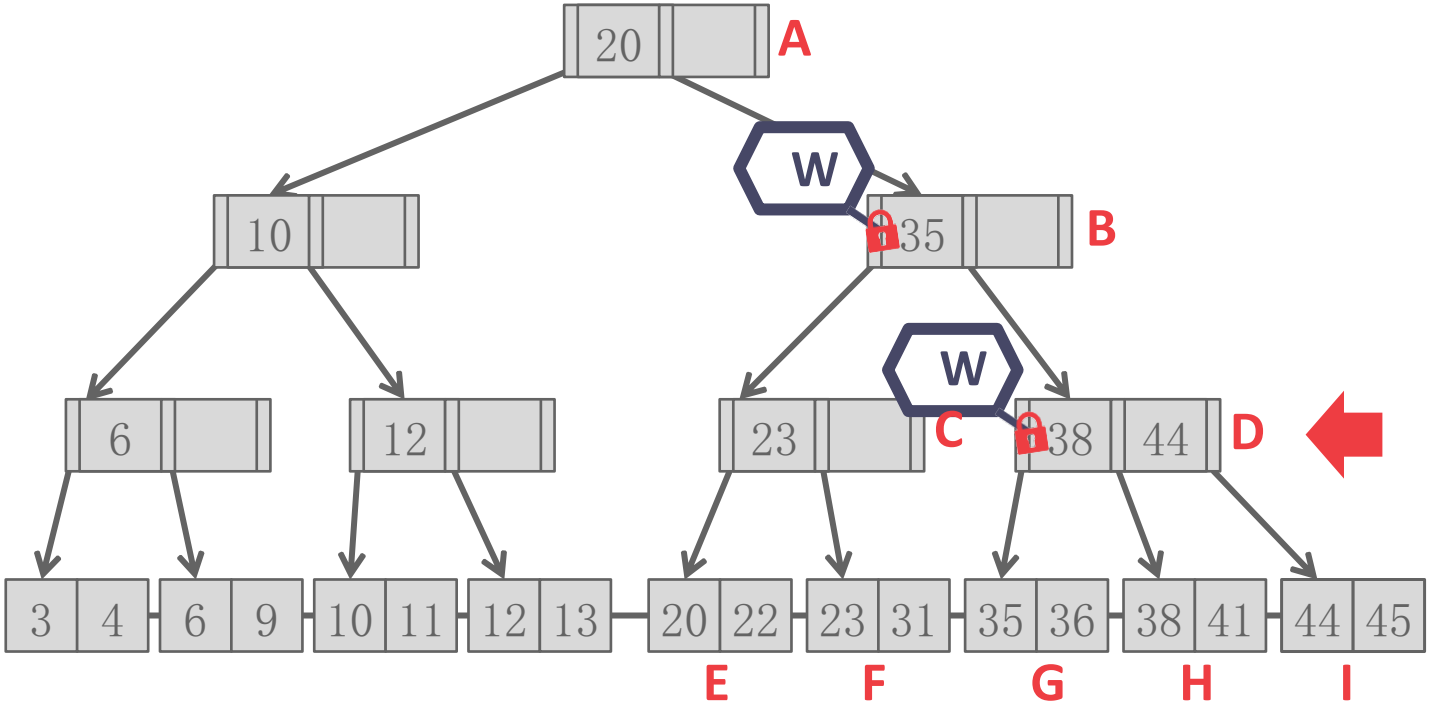
EXAMPLE #2 – DELETE 38



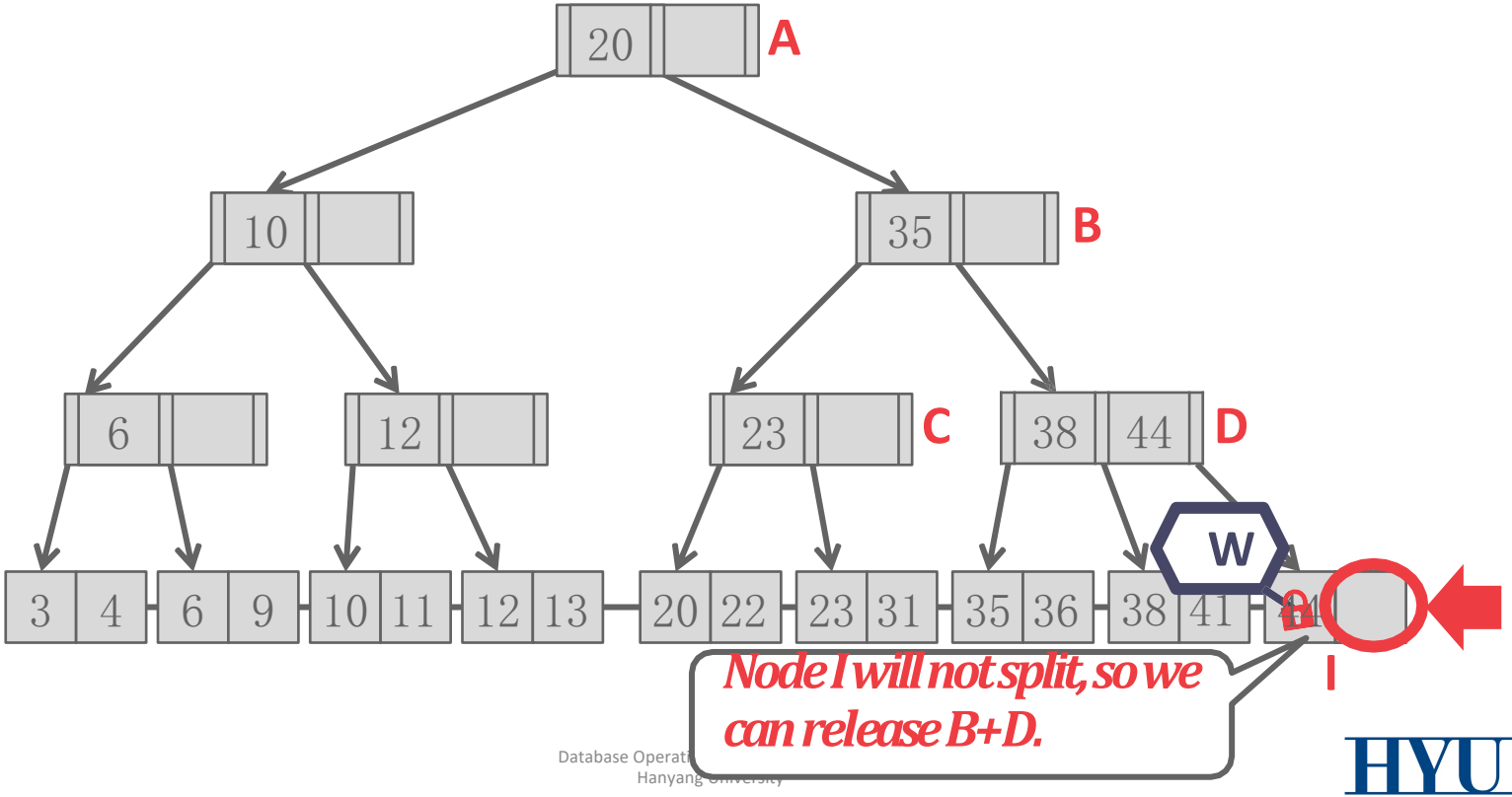
EXAMPLE #3 – INSERT 45



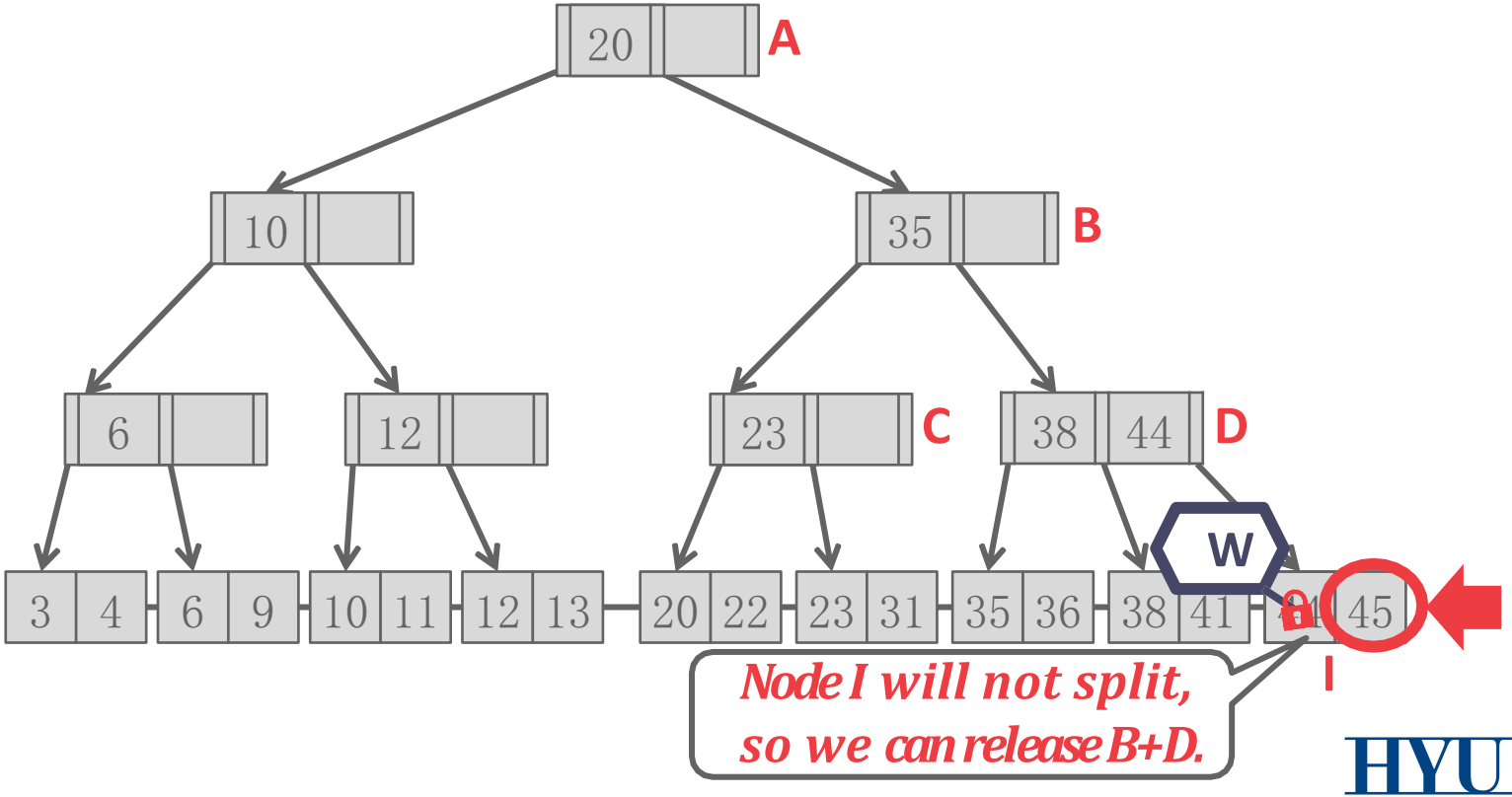
EXAMPLE #3 – INSERT 45



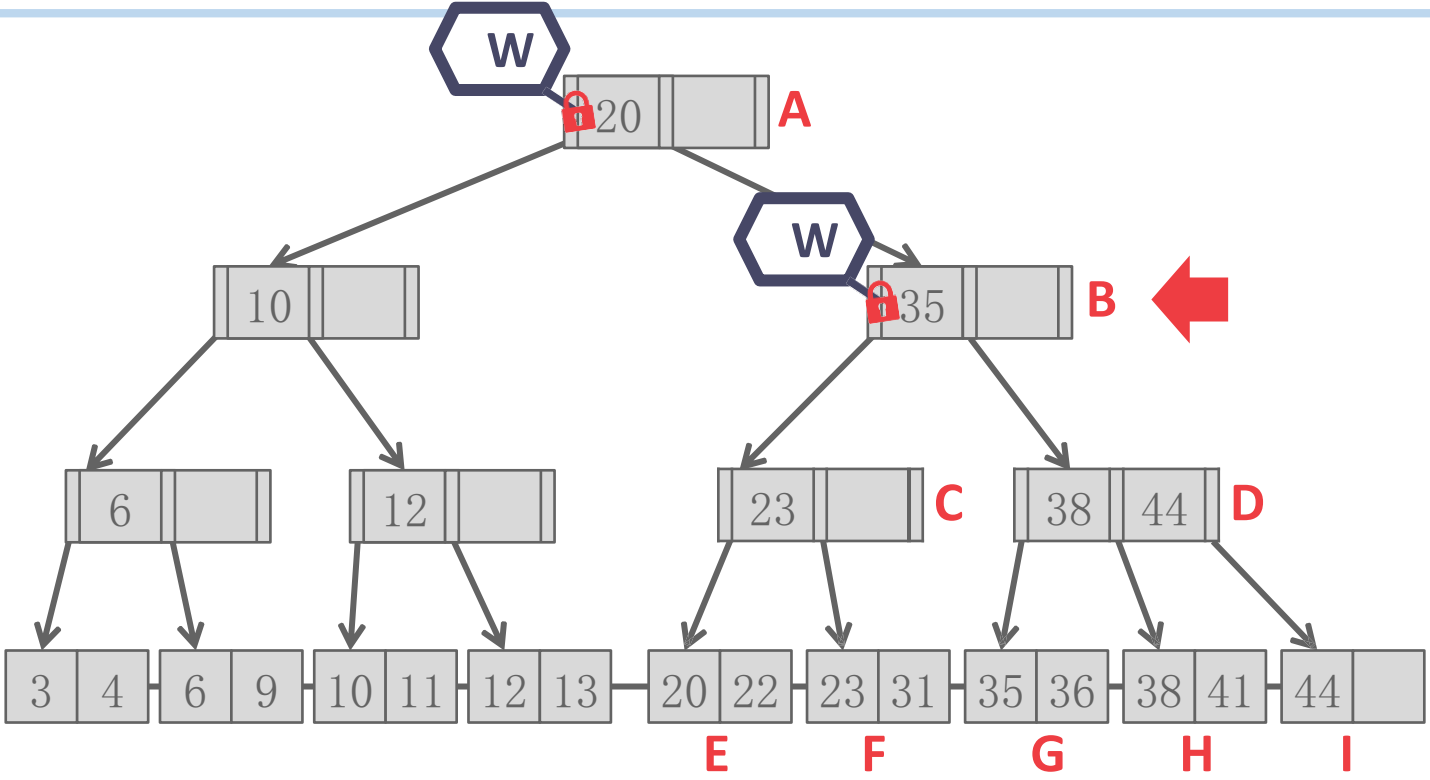
EXAMPLE #3 – INSERT 45



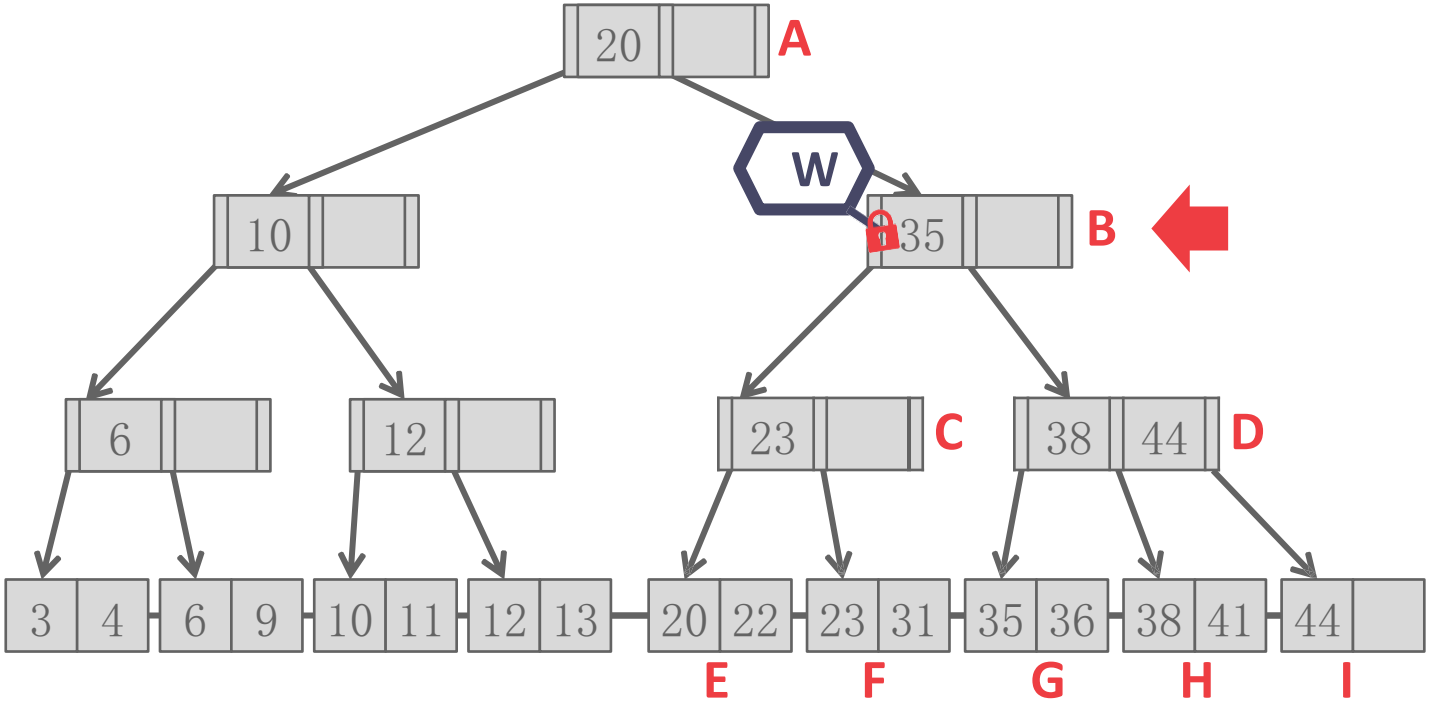
EXAMPLE #3 – INSERT 45



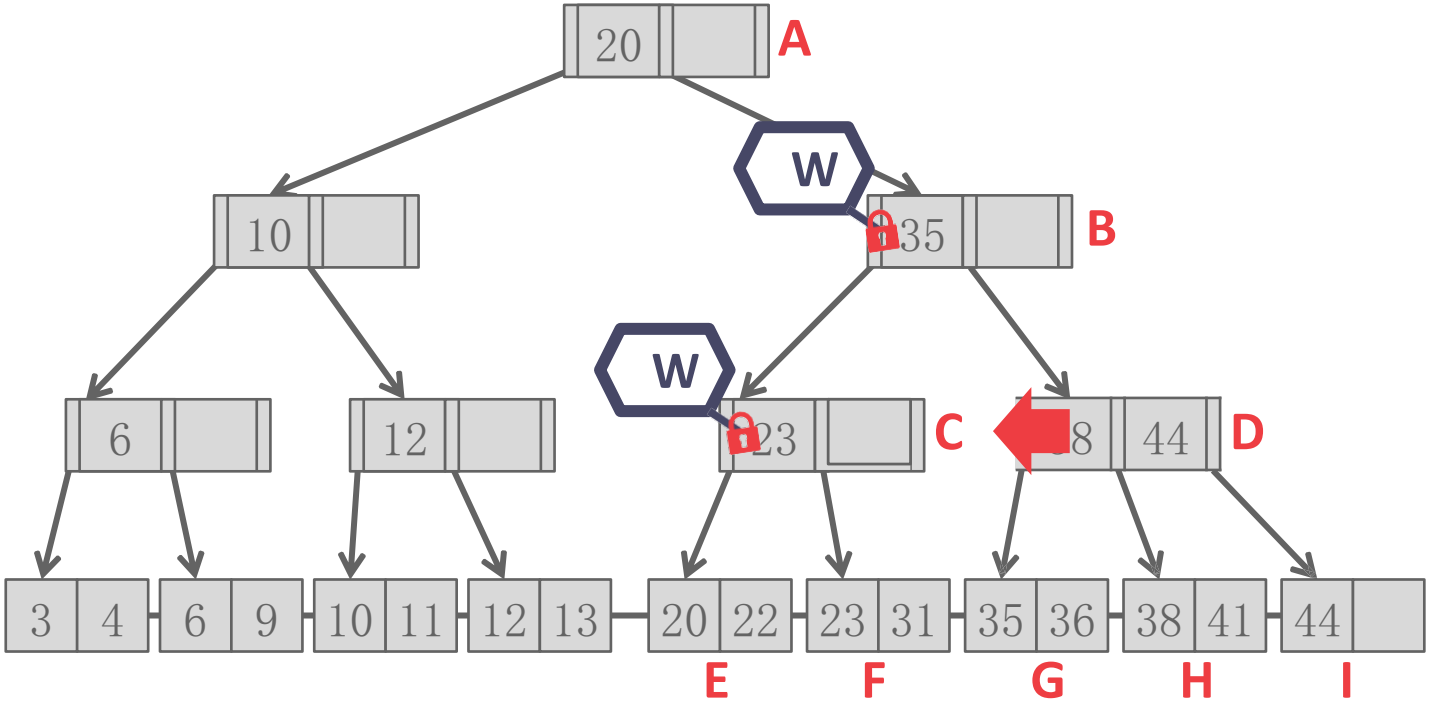
EXAMPLE #4 – INSERT 25



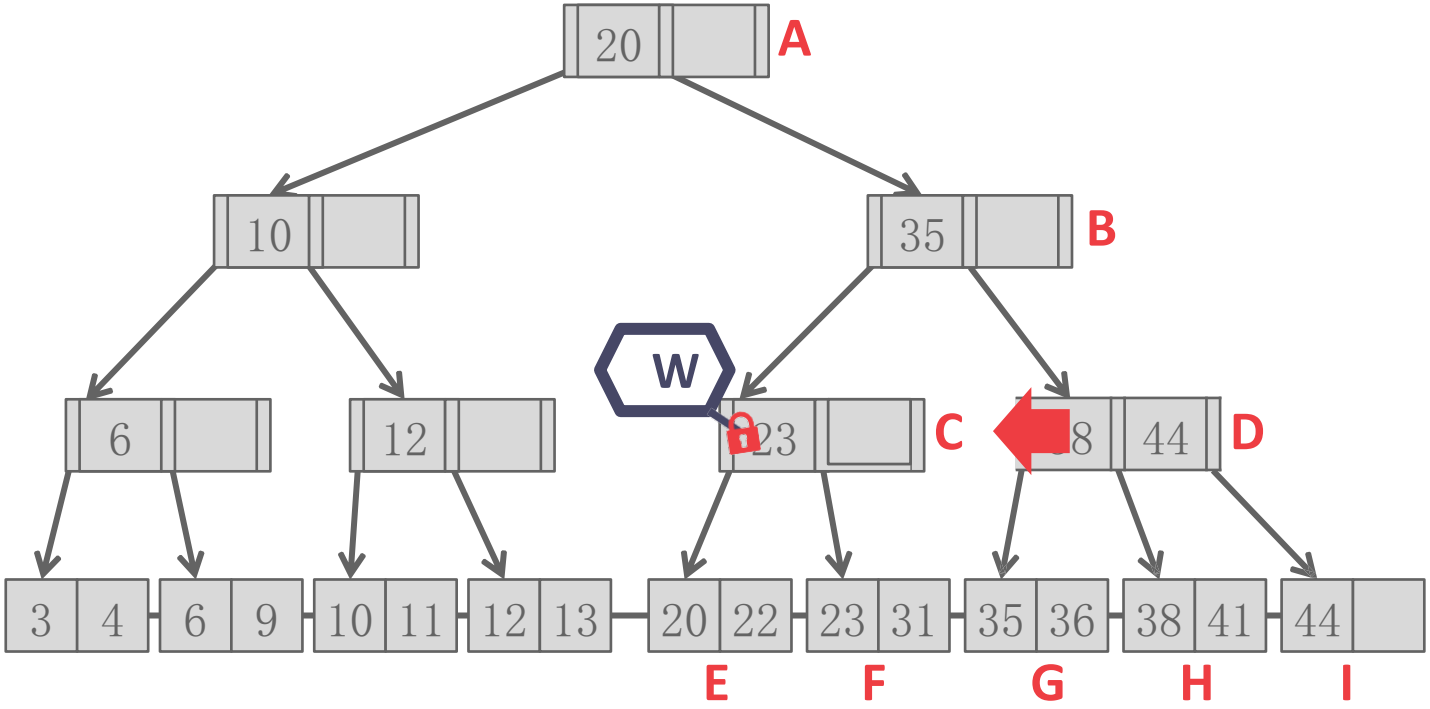
EXAMPLE #4 – INSERT 25



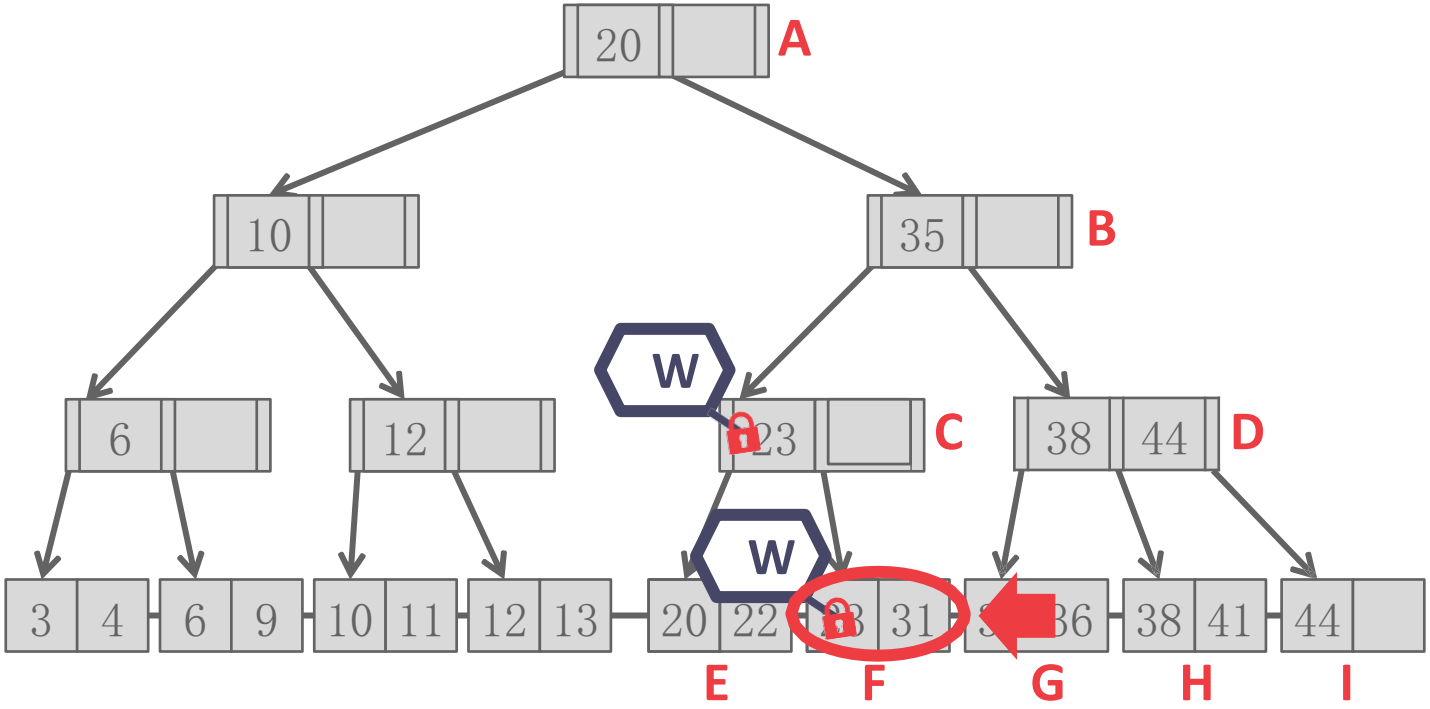
EXAMPLE #4 – INSERT 25



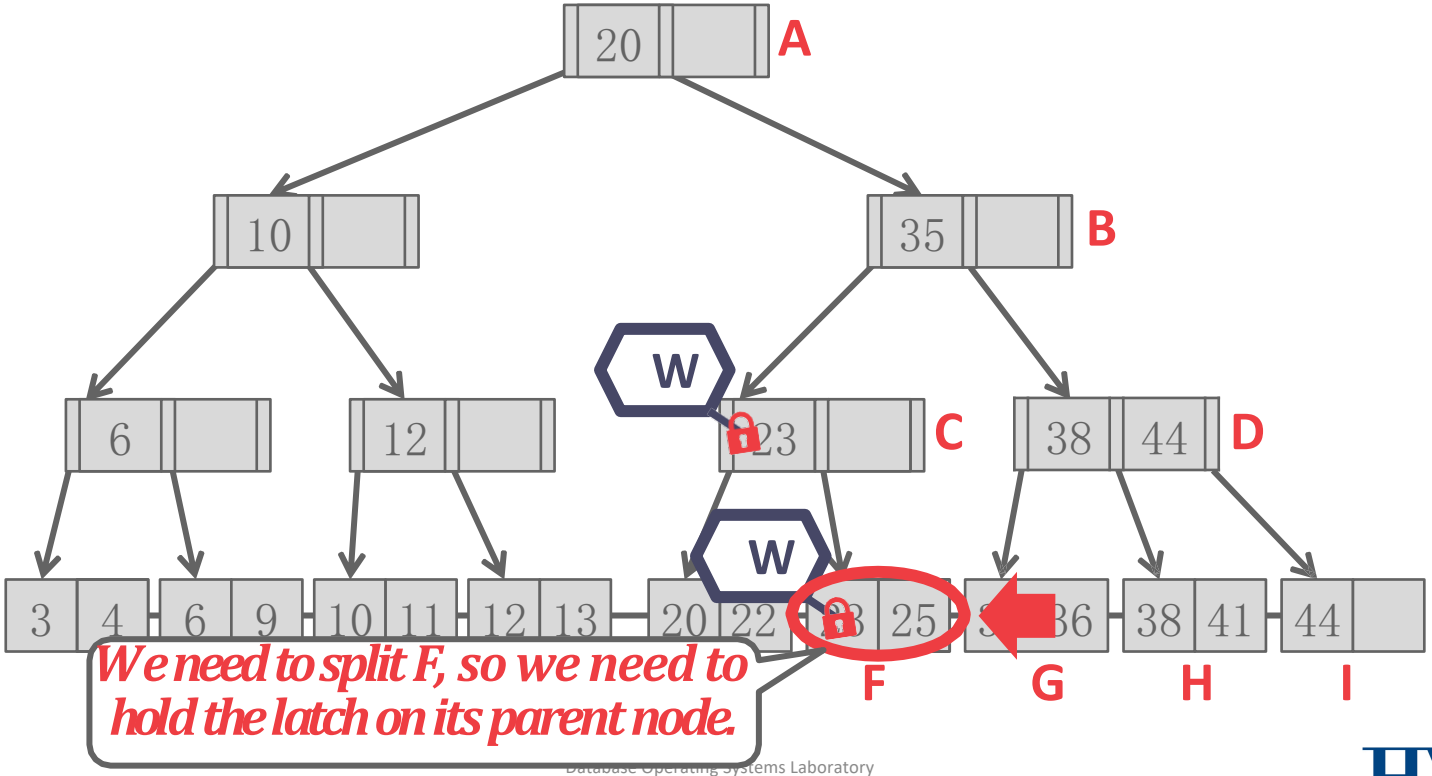
EXAMPLE #4 – INSERT 25



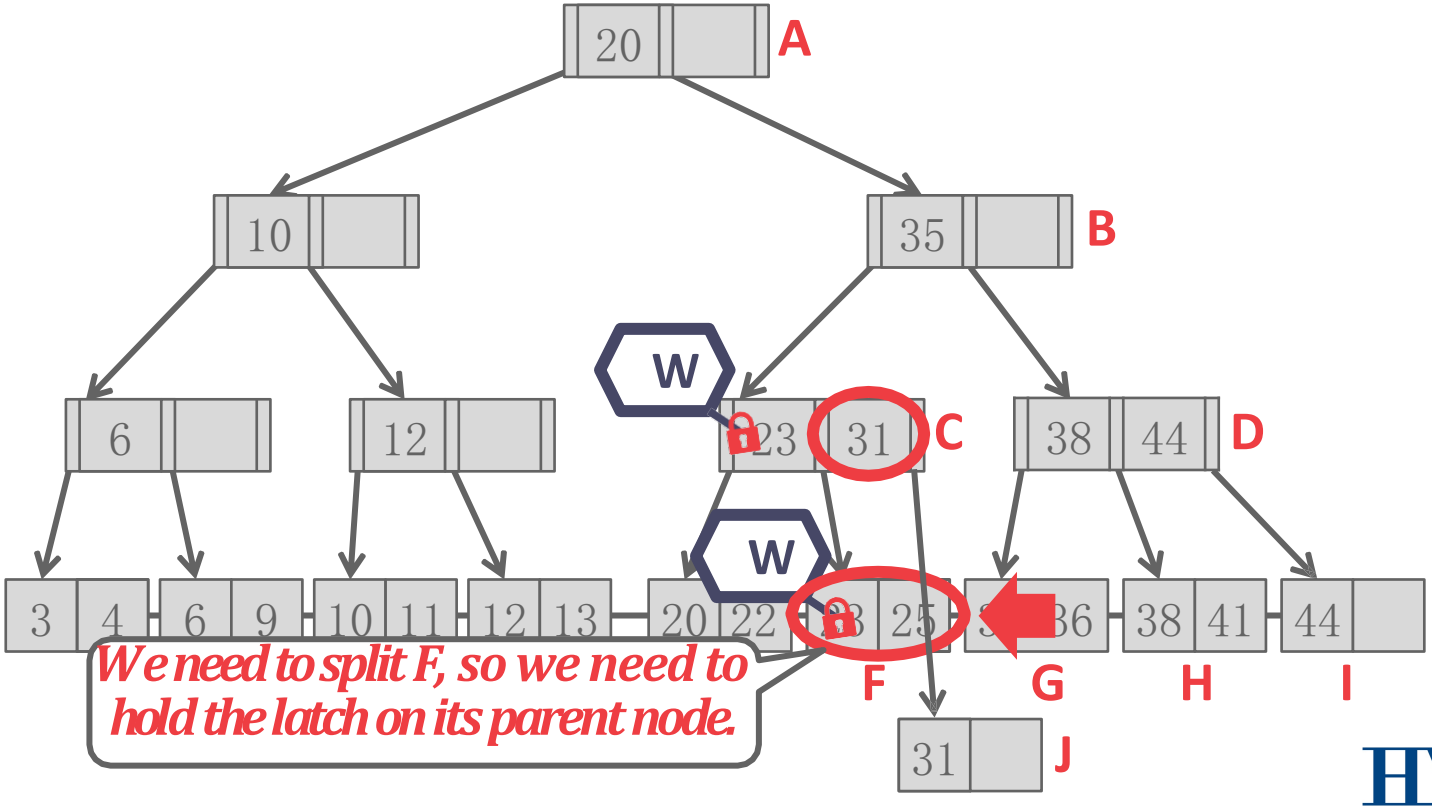
EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25



EXAMPLE #4 – INSERT 25



OBSERVATION

Because CaS only updates a single address at a time, this limits the design of our data structures

→ We cannot build a latch-free B+Tree because we need to update multiple pointers on split/merge operations.

What if we had an indirection layer that allowed us to update multiple addresses atomically?

BW-TREE

Latch-free B+Tree index built for the Microsoft Hekaton project.

Key Idea #1: Deltas

→ No updates in place

→ Reduces cache invalidation.

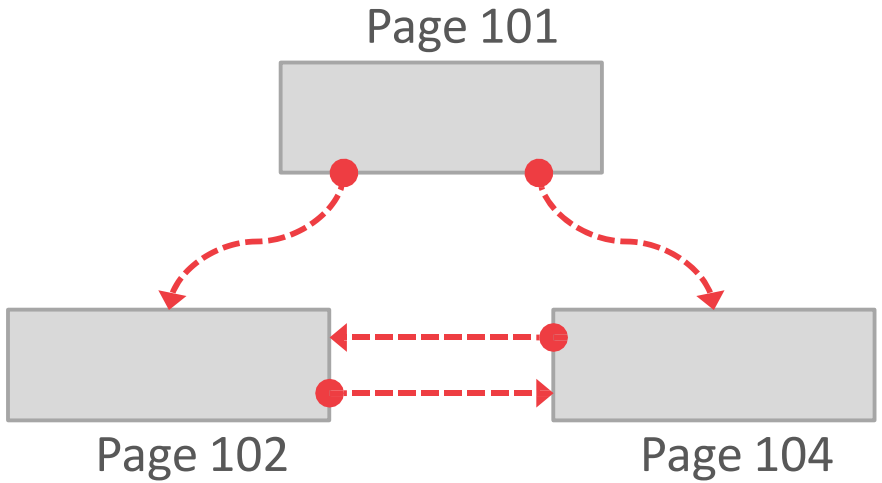
Key Idea #2: Mapping Table

→ Allows for CaS of physical locations of pages.

BW-TREE: MAPPING TABLE

Mapping Table

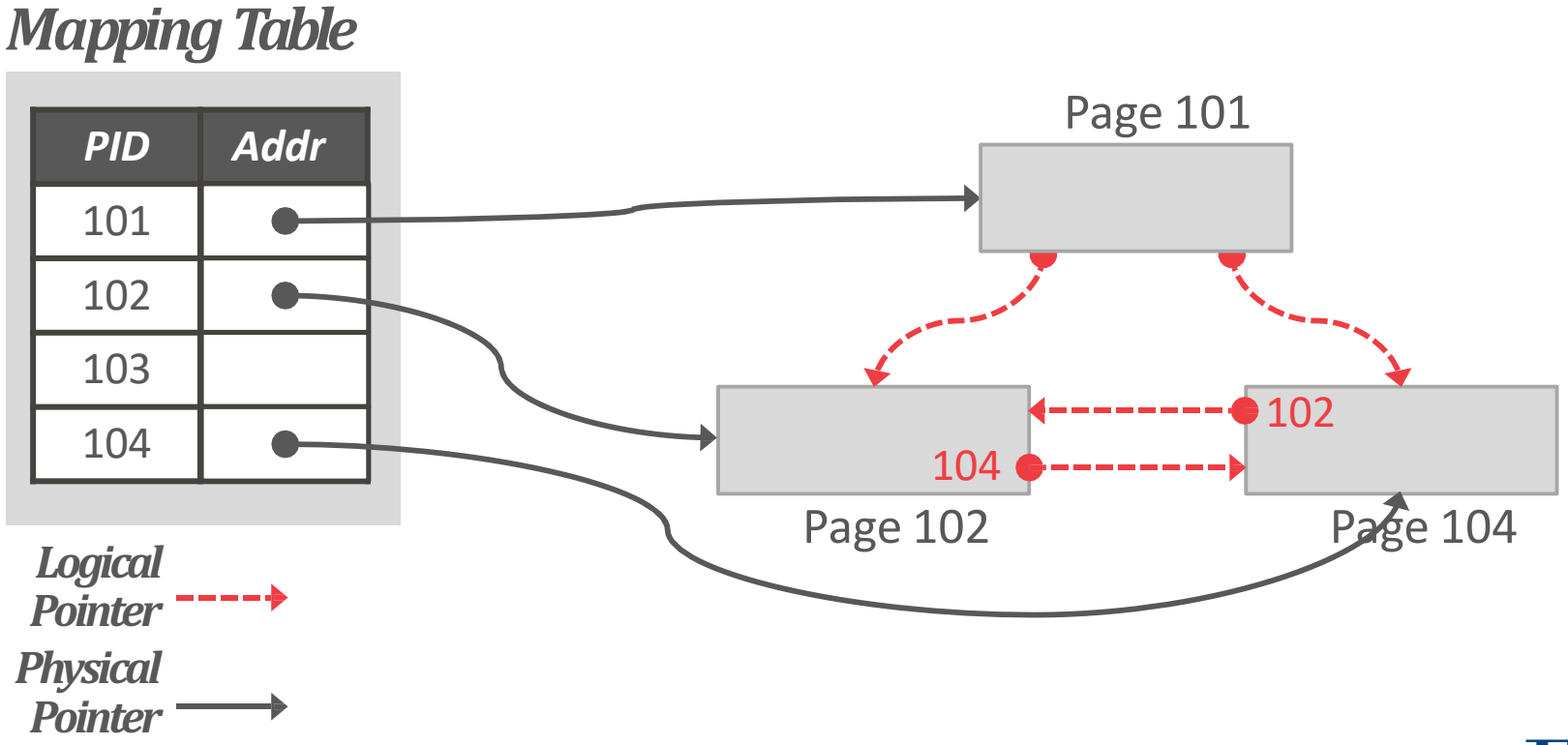
PID	Addr
101	
102	
103	
104	



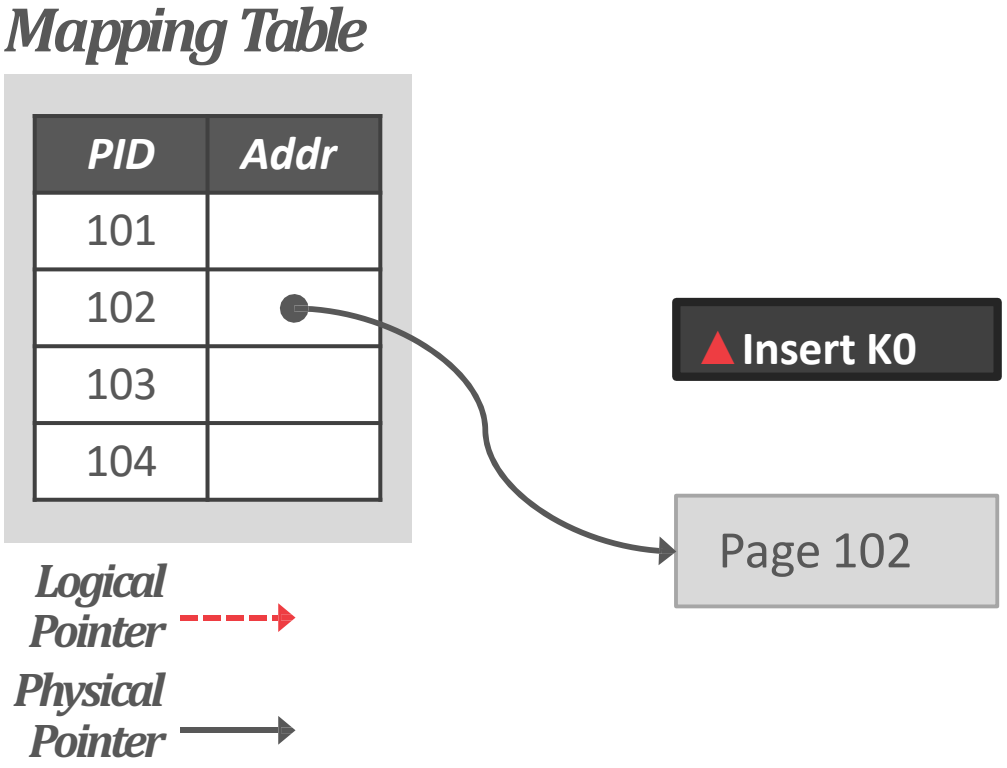
Logical Pointer ⇢

Physical Pointer →

BW-TREE: MAPPING TABLE

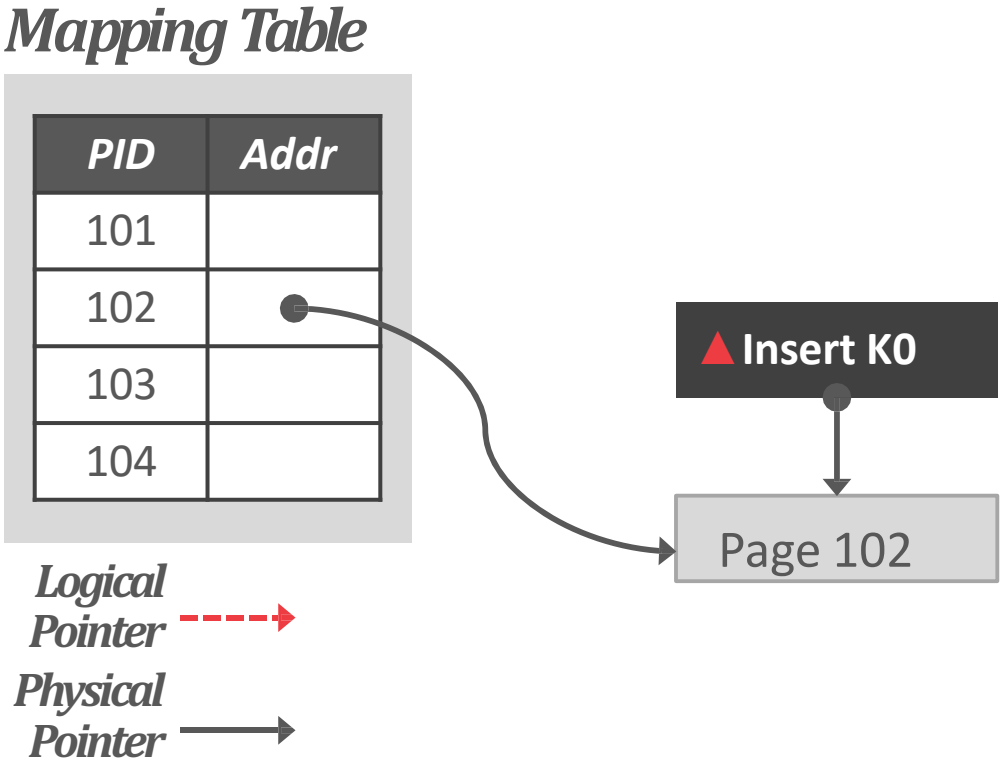


BW-TREE: DELTA UPDATES



Each update to a page produces a new delta.

BW-TREE: DELTA UPDATES

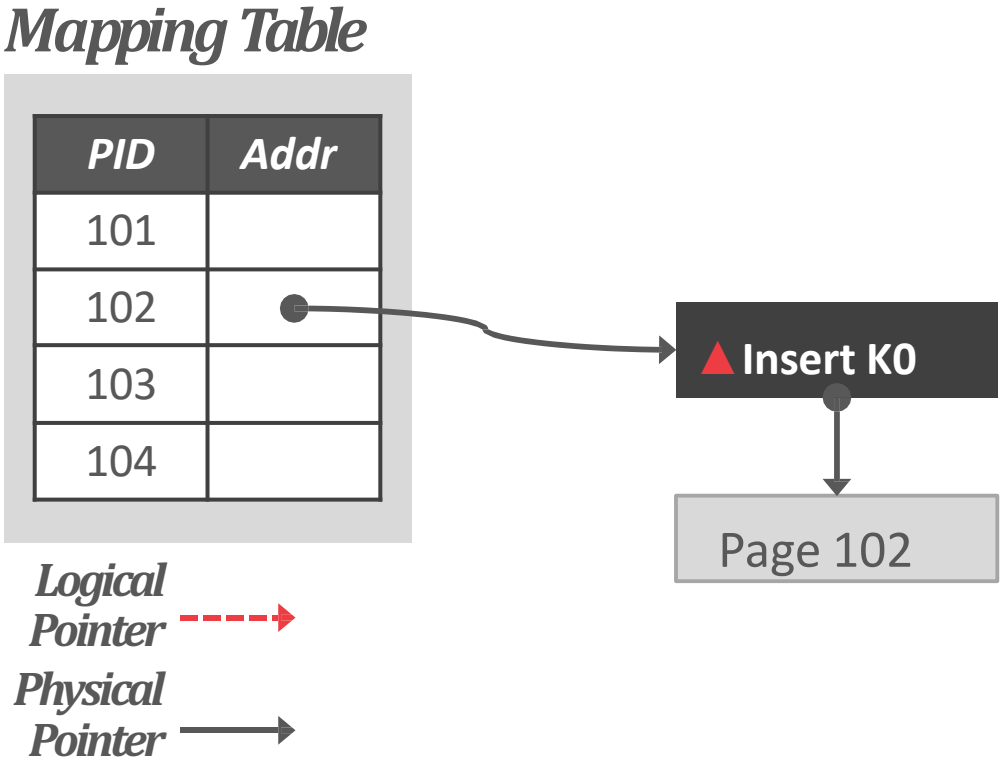


Each update to a page produces a new delta.

Delta physically points to base page.

Install delta address in physical address slot of mapping table using CaS.

BW-TREE: DELTA UPDATES

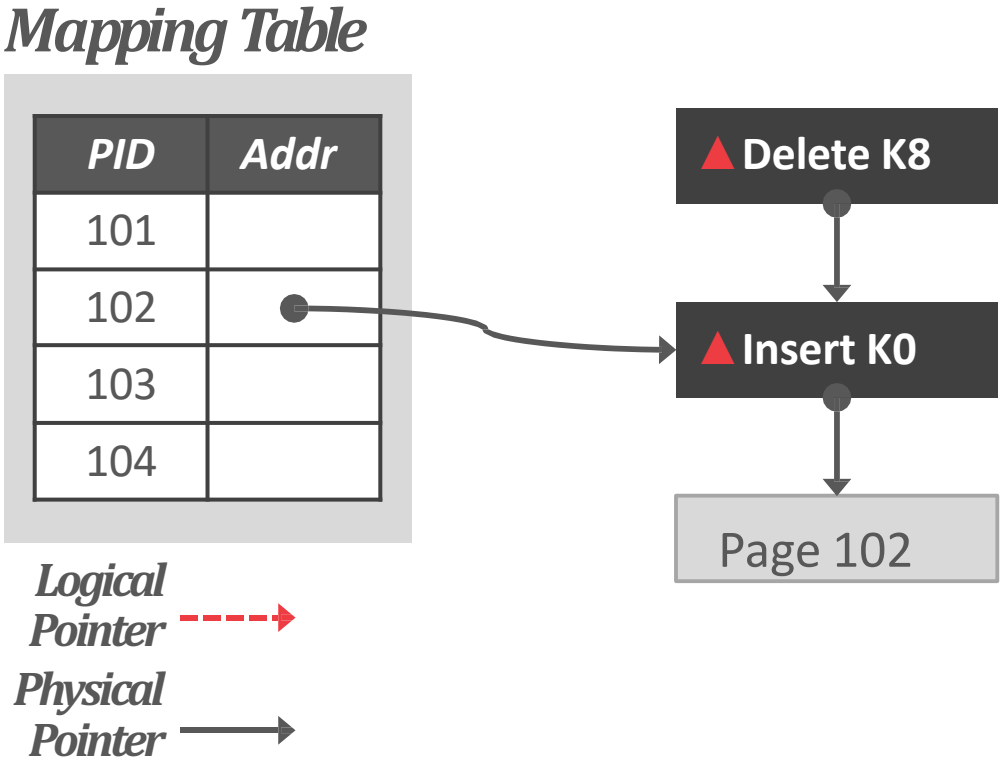


Each update to a page produces a new delta.

Delta physically points to base page.

Install delta address in physical address slot of mapping table using CaS.

BW-TREE: DELTA UPDATES

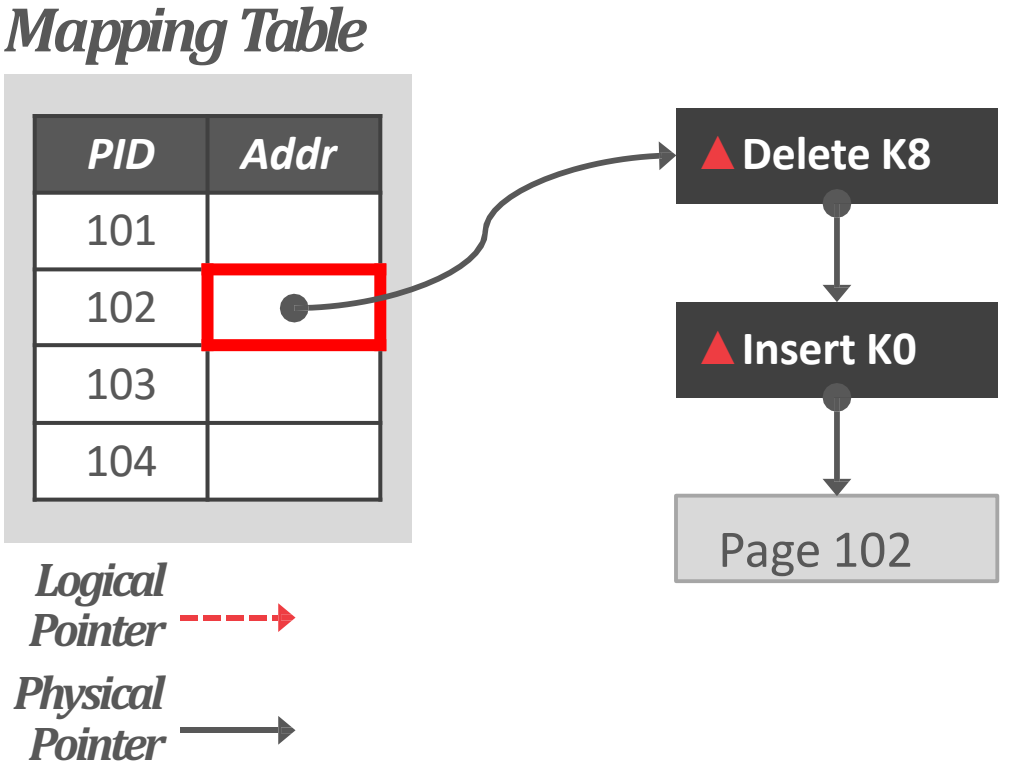


Each update to a page produces a new delta.

Delta physically points to base page.

Install delta address in ph ysical address slot of mapping table using CaS.

BW-TREE: DELTA UPDATES

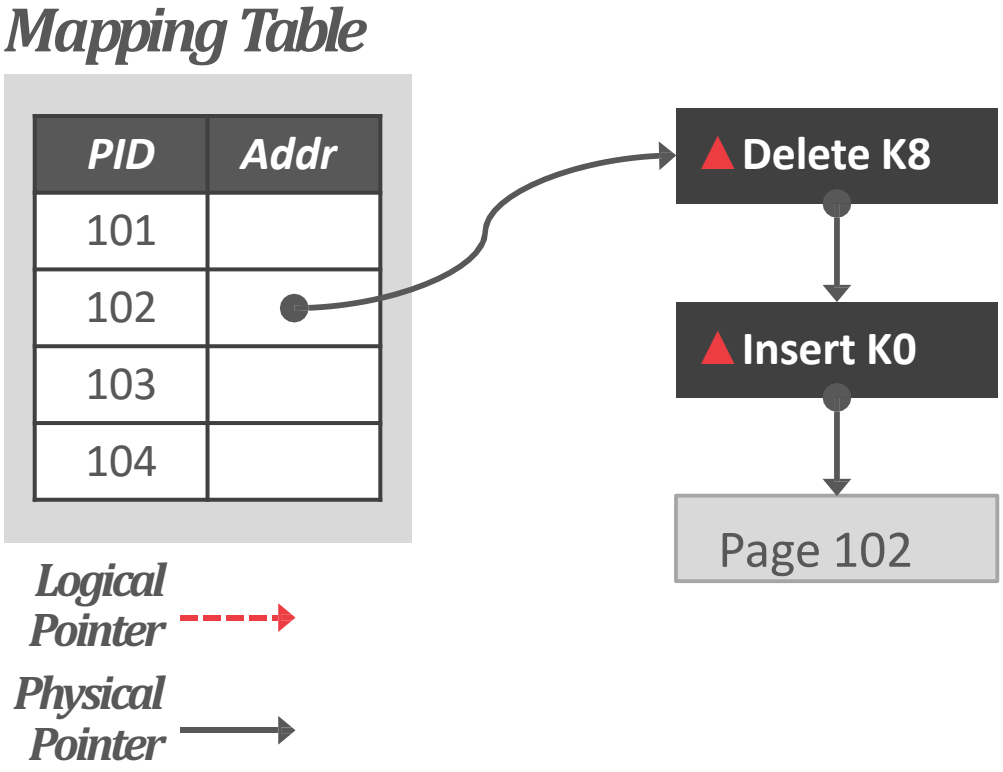


Each update to a page produces a new delta.

Delta physically points to base page.

Install delta address in physical address slot of mapping table using CaS.

BW-TREE: DELTA UPDATES

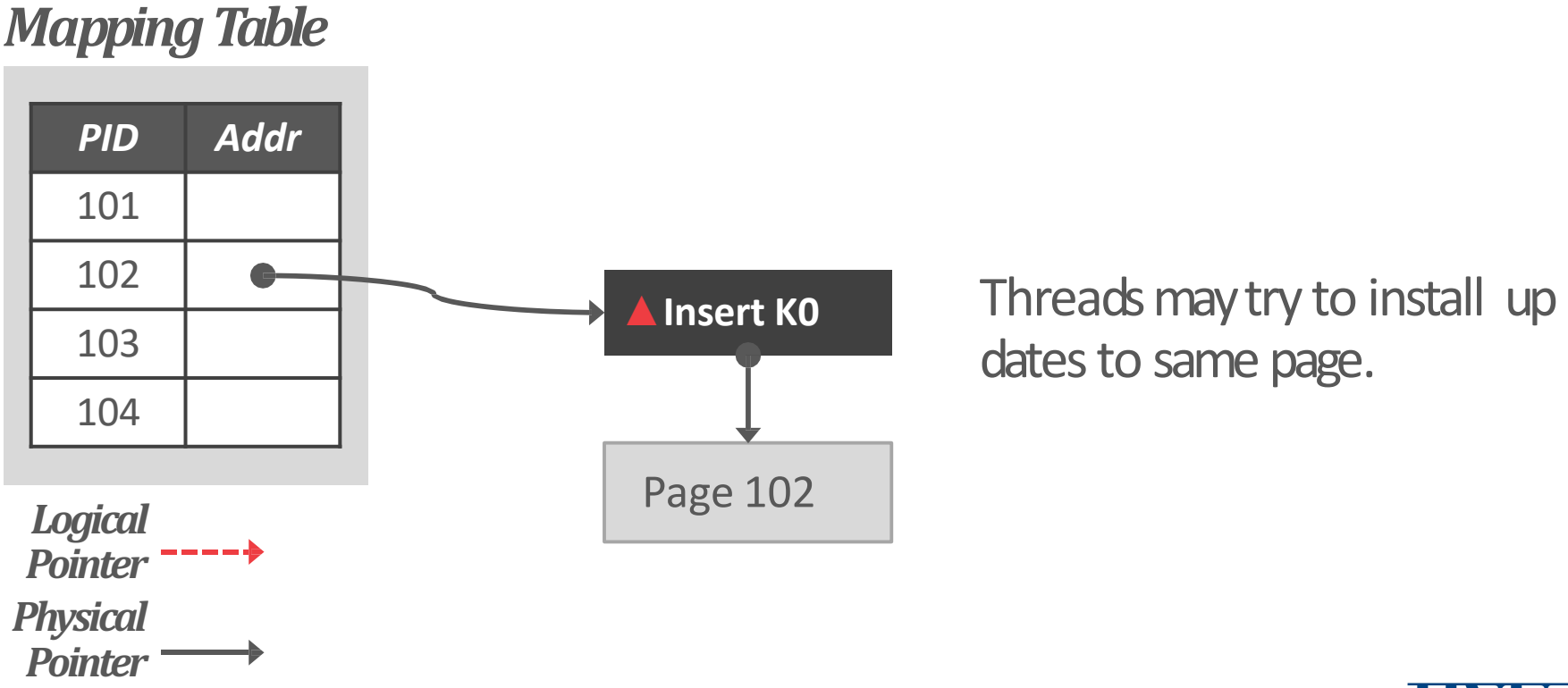


Traverse tree like a regular B +tree.

If mapping table points to d elta chain, stop at first occu rrence of search key.

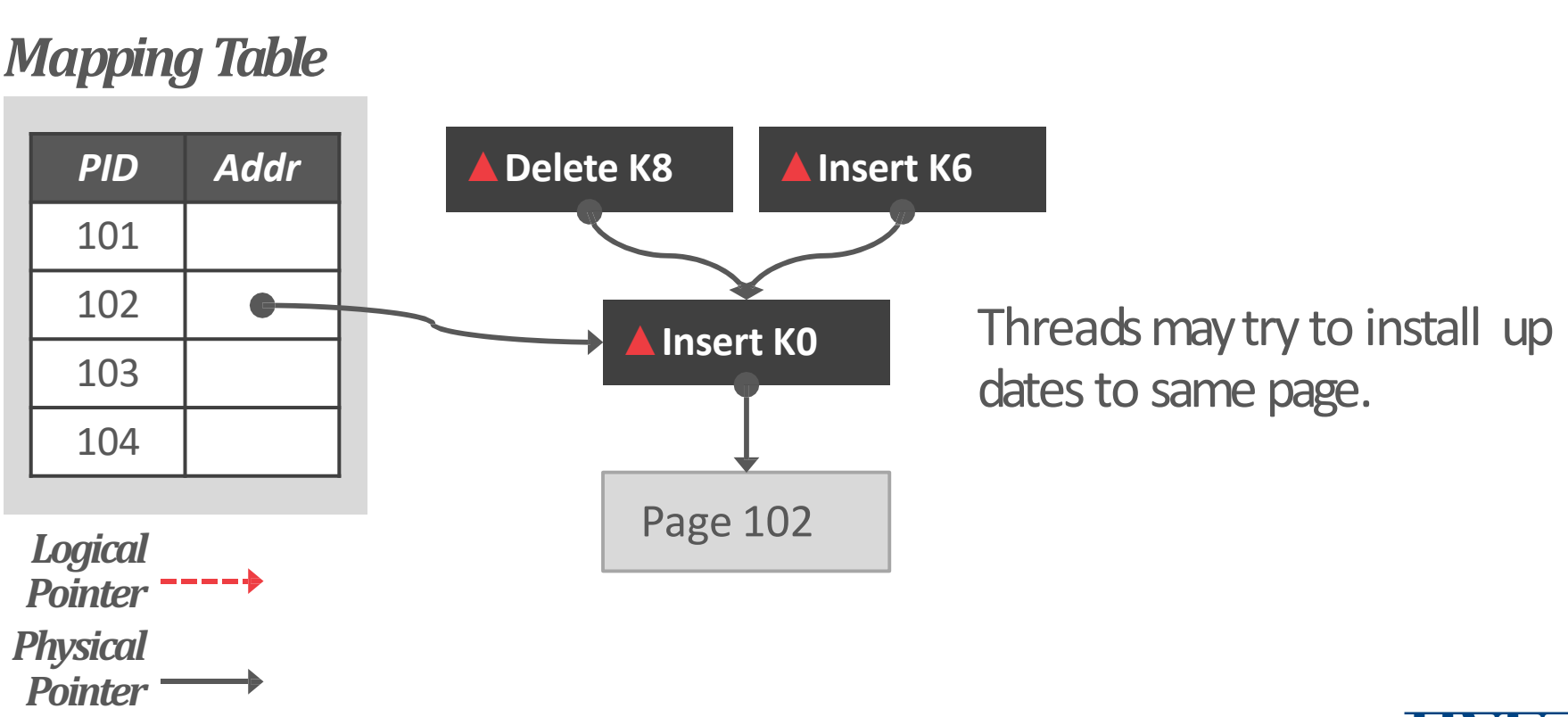
Otherwise, perform binary search on base page.

BW-TREE: CONTENTION UPDATES



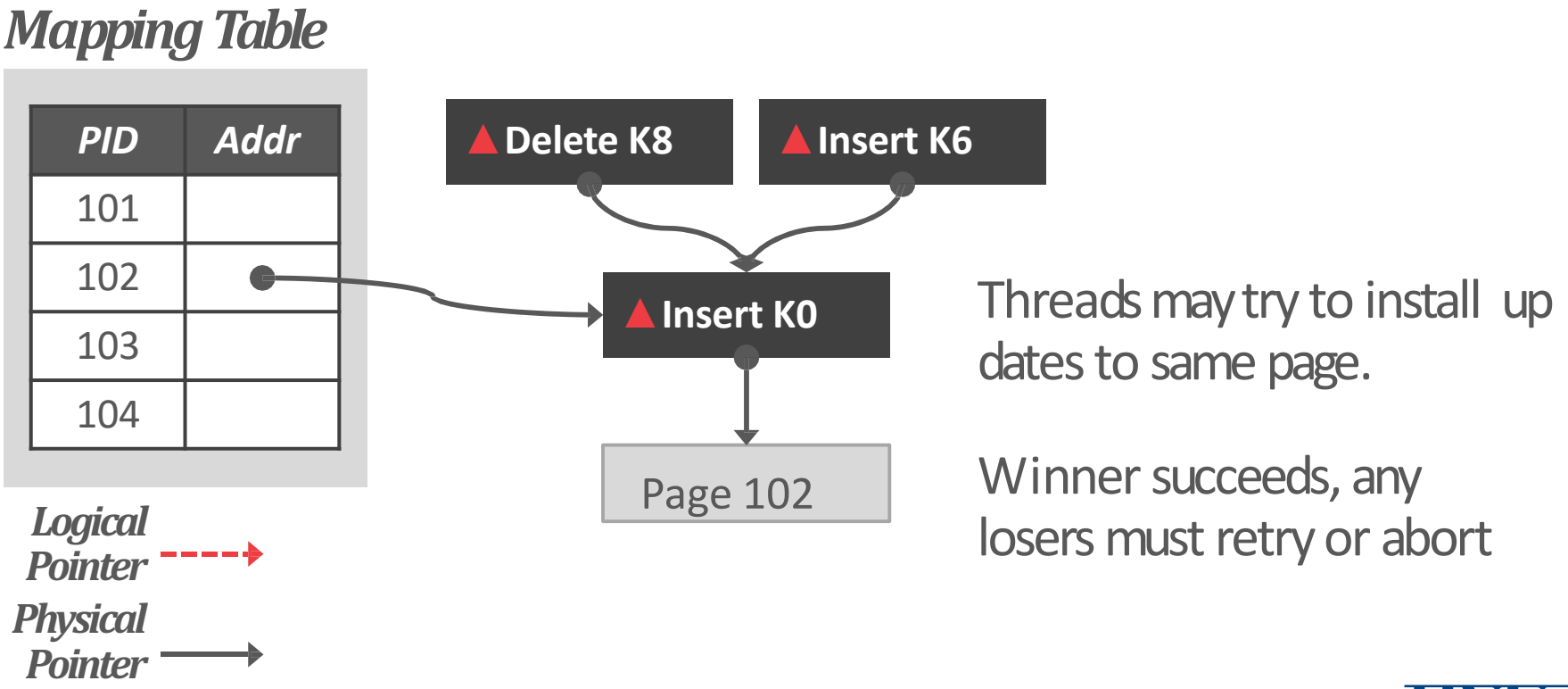
Threads may try to install up dates to same page.

BW-TREE: CONTENTION UPDATES



Threads may try to install up dates to same page.

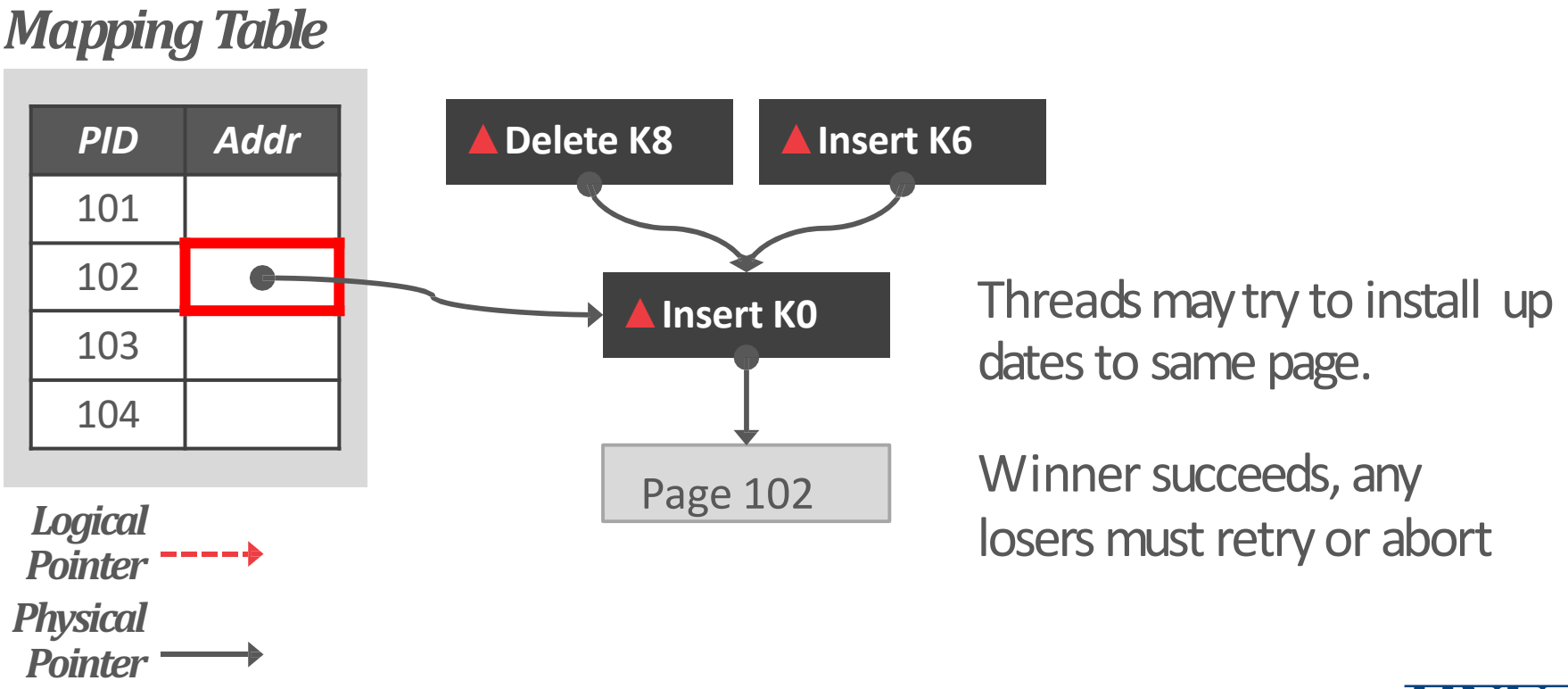
BW-TREE: CONTENTION UPDATES



Threads may try to install up dates to same page.

Winner succeeds, any losers must retry or abort

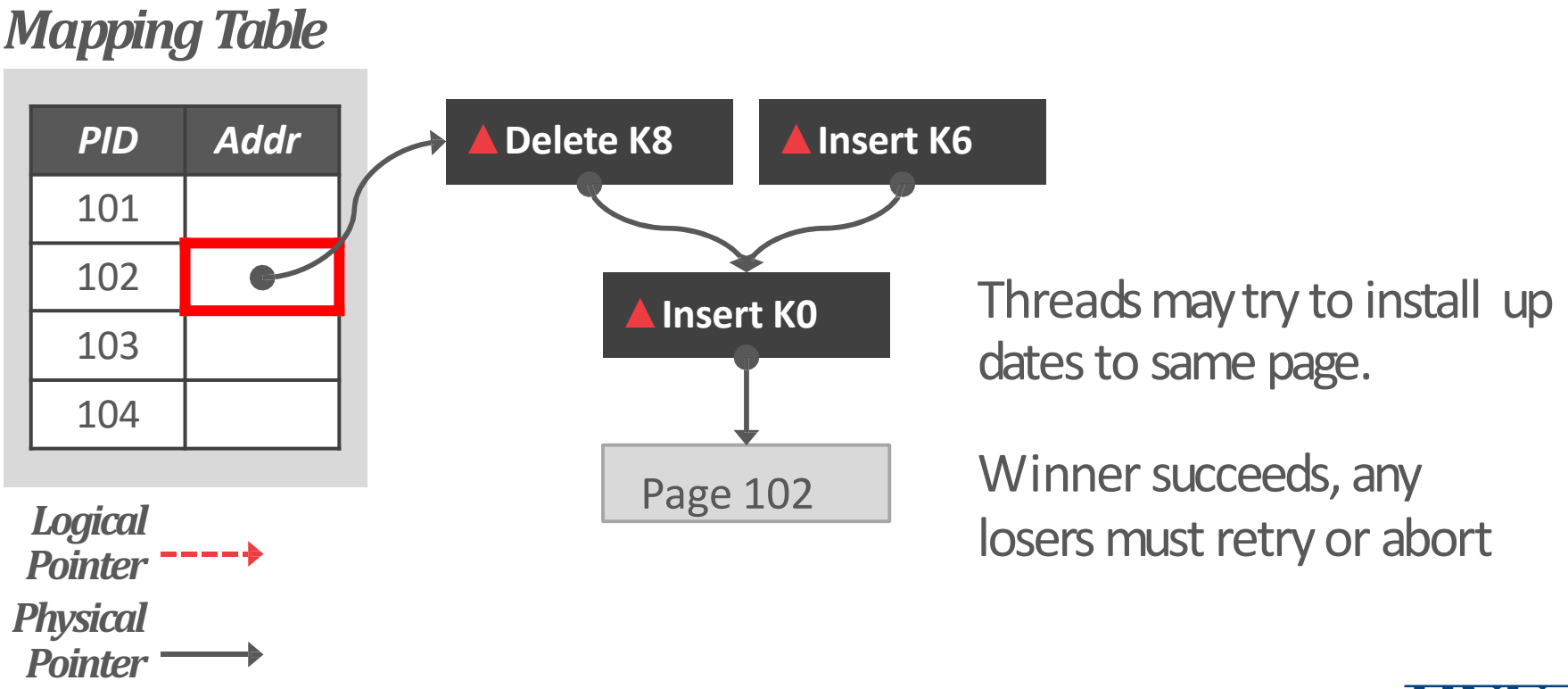
BW-TREE: CONTENTION UPDATES



Threads may try to install up dates to same page.

Winner succeeds, any losers must retry or abort

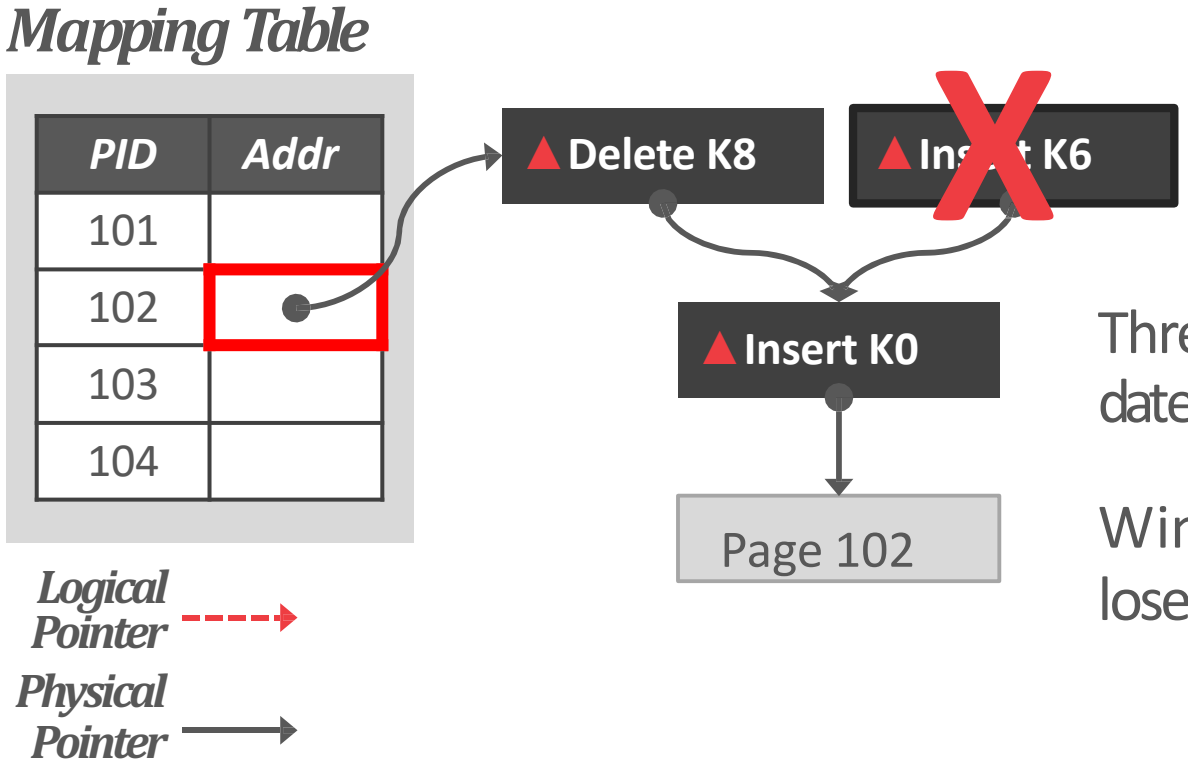
BW-TREE: CONTENTION UPDATES



Threads may try to install up dates to same page.

Winner succeeds, any losers must retry or abort

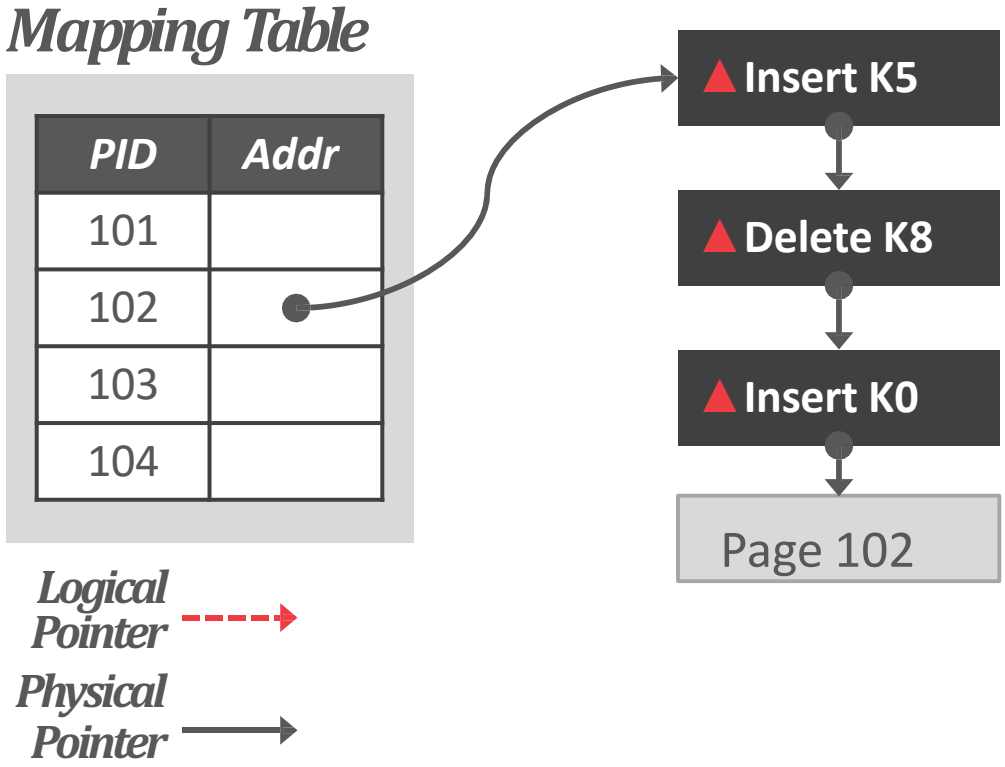
BW-TREE: CONTENTION UPDATES



Threads may try to install up dates to same page.

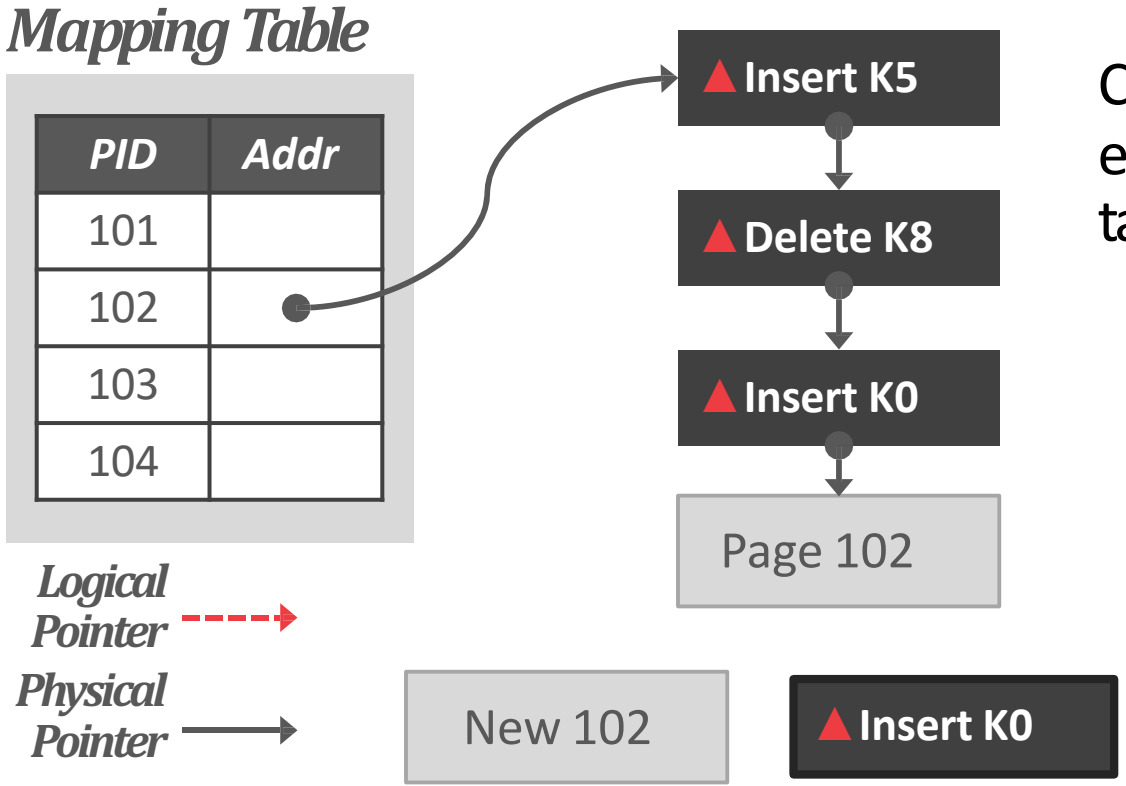
Winner succeeds, any losers must retry or abort

BW-TREE: CONSOLIDATION



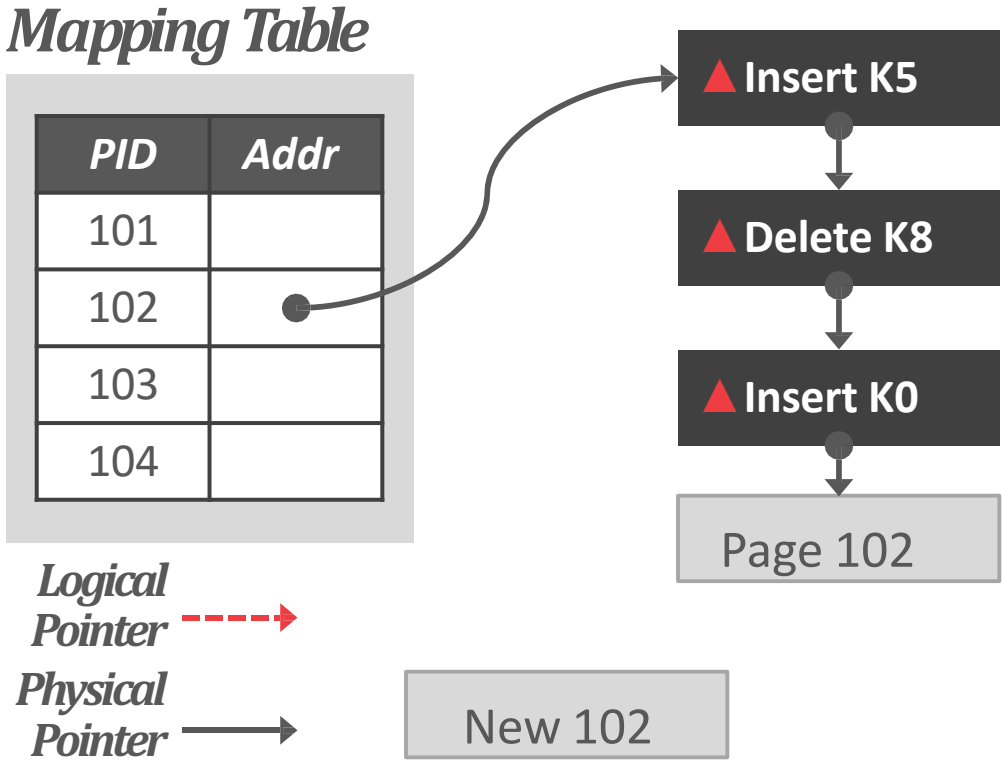
Consolidate updates by cr eating new page with del tas applied.

BW-TREE: CONSOLIDATION



Consolidate updates by cr eating new page with del tas applied.

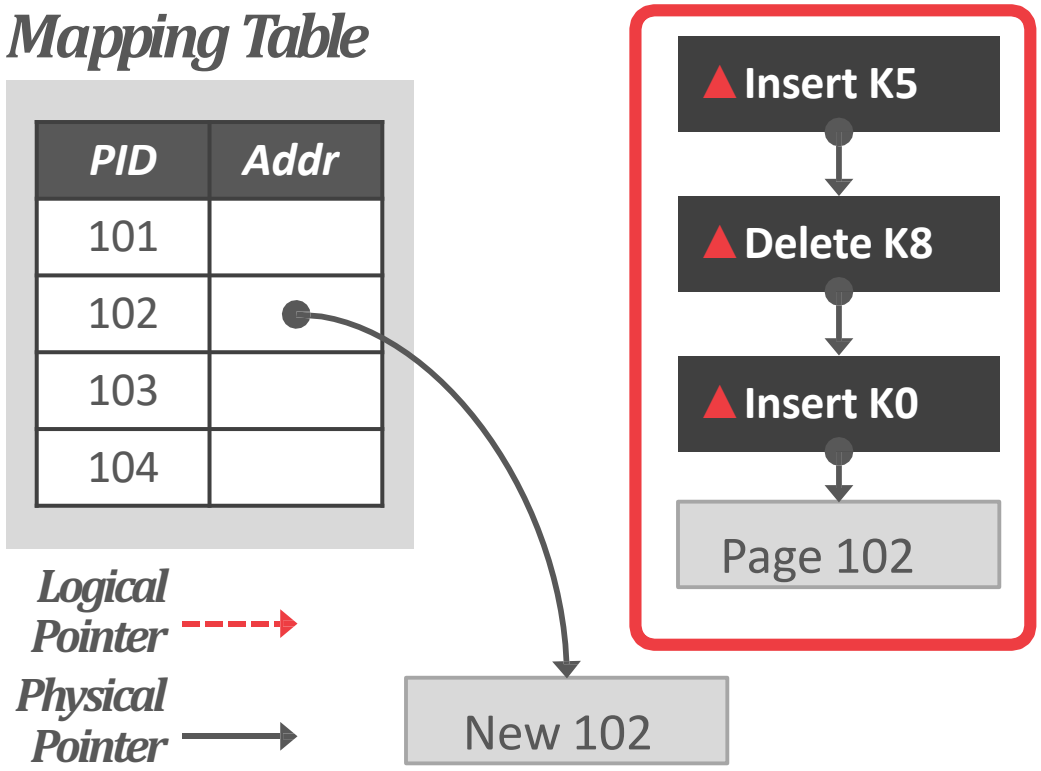
BW-TREE: CONSOLIDATION



Consolidate updates by cr eating new page with del tas applied.

CaS-ing the mapping table address ensures no deltas are missed.

BW-TREE: CONSOLIDATION



Consolidate updates by cr eating new page with del tas applied.

CaS-ing the mapping table ad dress ensures no deltas are missed.

Old page + deltas are marked as garbage.

GARBAGE COLLECTION

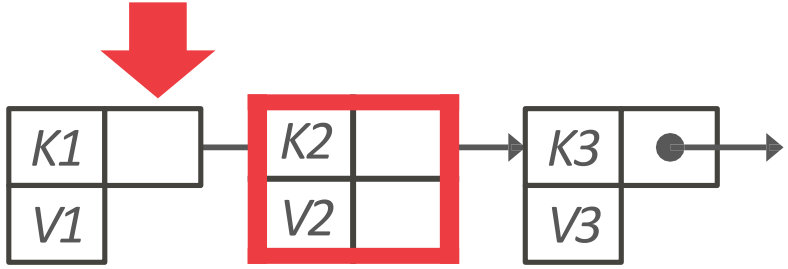
We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

→ Reference Counting

→ Epoch-based Reclamation

→ Hazard Pointers

→ Many others…



GARBAGE COLLECTION

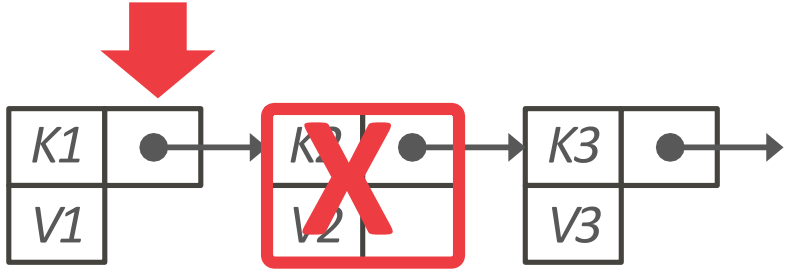
We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

→ Reference Counting

→ Epoch-based Reclamation

→ Hazard Pointers

→ Many others…



GARBAGE COLLECTION

We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

→ Reference Counting

→ Epoch-based Reclamation

→ Hazard Pointers

→ Many others…

GARBAGE COLLECTION

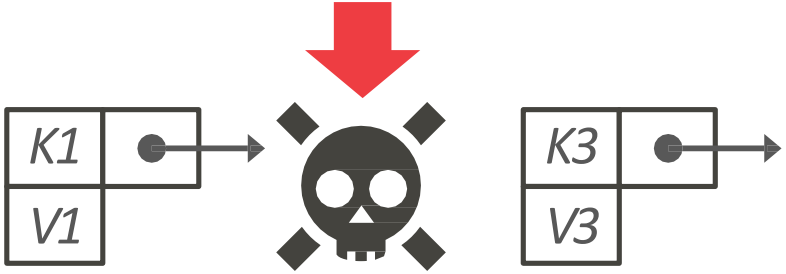
We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

→ Reference Counting

→ Epoch-based Reclamation

→ Hazard Pointers

→ Many others…



REFERENCE COUNTING

Maintain a counter for each node to keep track of the number of threads that are accessing it.

→ Increment the counter before accessing.

→ Decrement it when finished.

→ A node is only safe to delete when the count is zero.

This has bad performance for multi-core CPUs

→ Incrementing/decrementing counters causes a lot of cache coherence traffic.

OBSERVATION

We don’t care about the actual value of the reference counter. We only need to know when it reaches zero.

We don’t have to perform garbage collection immediately when the counter reaches zero.

EPOCH GARBAGE COLLECTION

Maintain a global **epoch** counter that is periodically updated (e.g., every 10 ms).

→ Keep track of what threads enter the index during an epoch and when they leave.

Mark the current epoch of a node when it is marked for deletion.

→ The node can be reclaimed once all threads have left that epoch (and all preceding epochs).

Also known as ***Read-Copy-Update*** (RCU) in Linux.

BW-TREE: GARBAGE COLLECTION

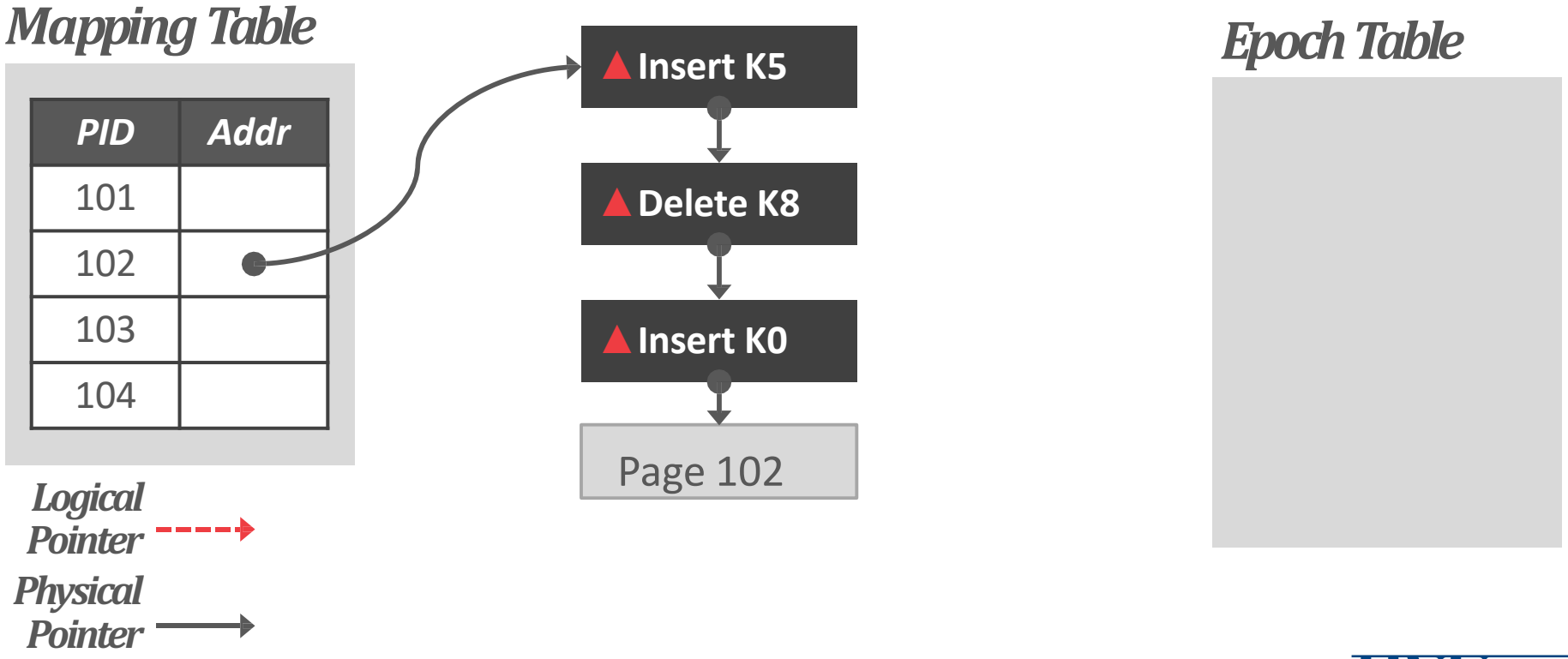
Operations are tagged with an **epoch**

→ Each epoch tracks the threads that are part of it and the objects that can be reclaimed.

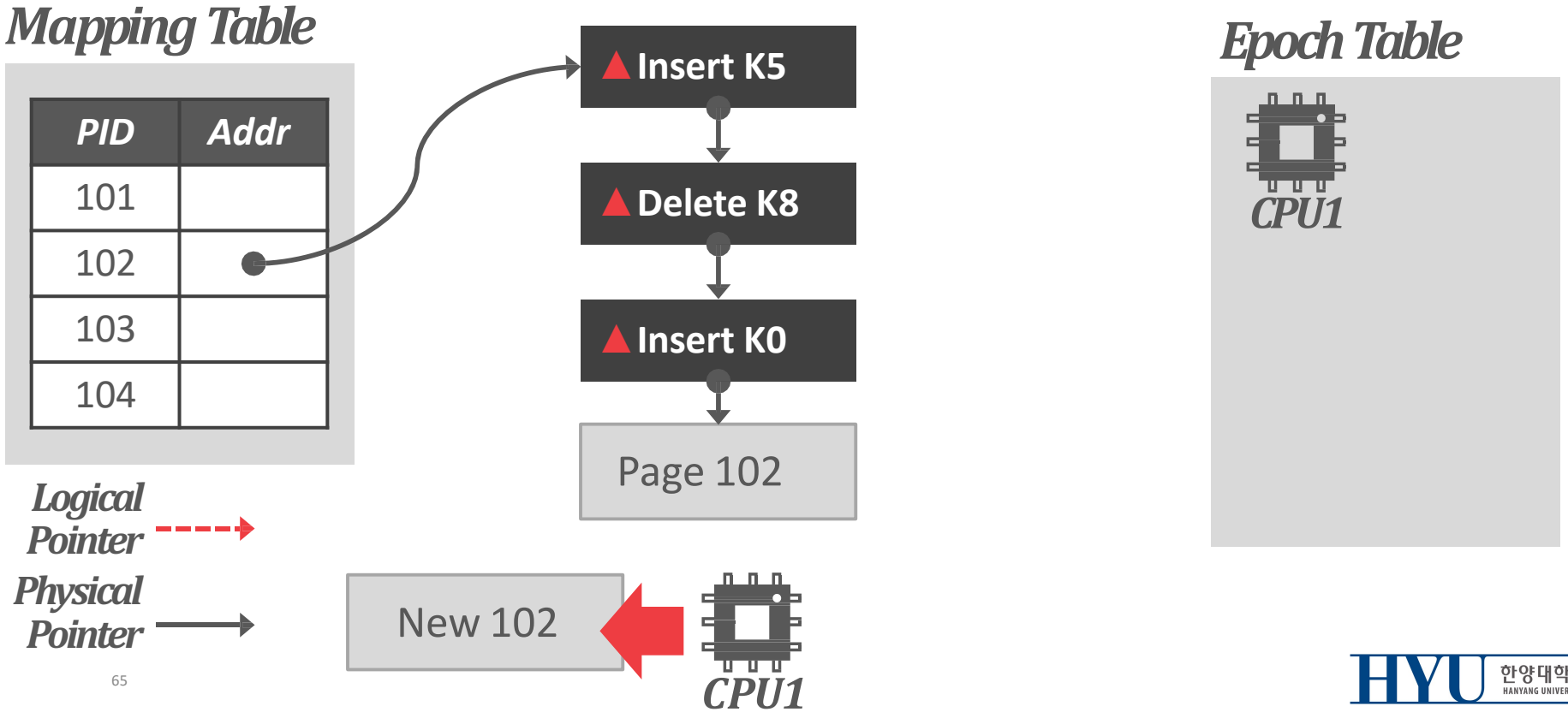
→ Thread joins an epoch prior to each operation and post objects that can be reclaimed for the current epoch (not necessarily the one it joined)

Garbage for an epoch reclaimed only when all threads have exited the epoch.

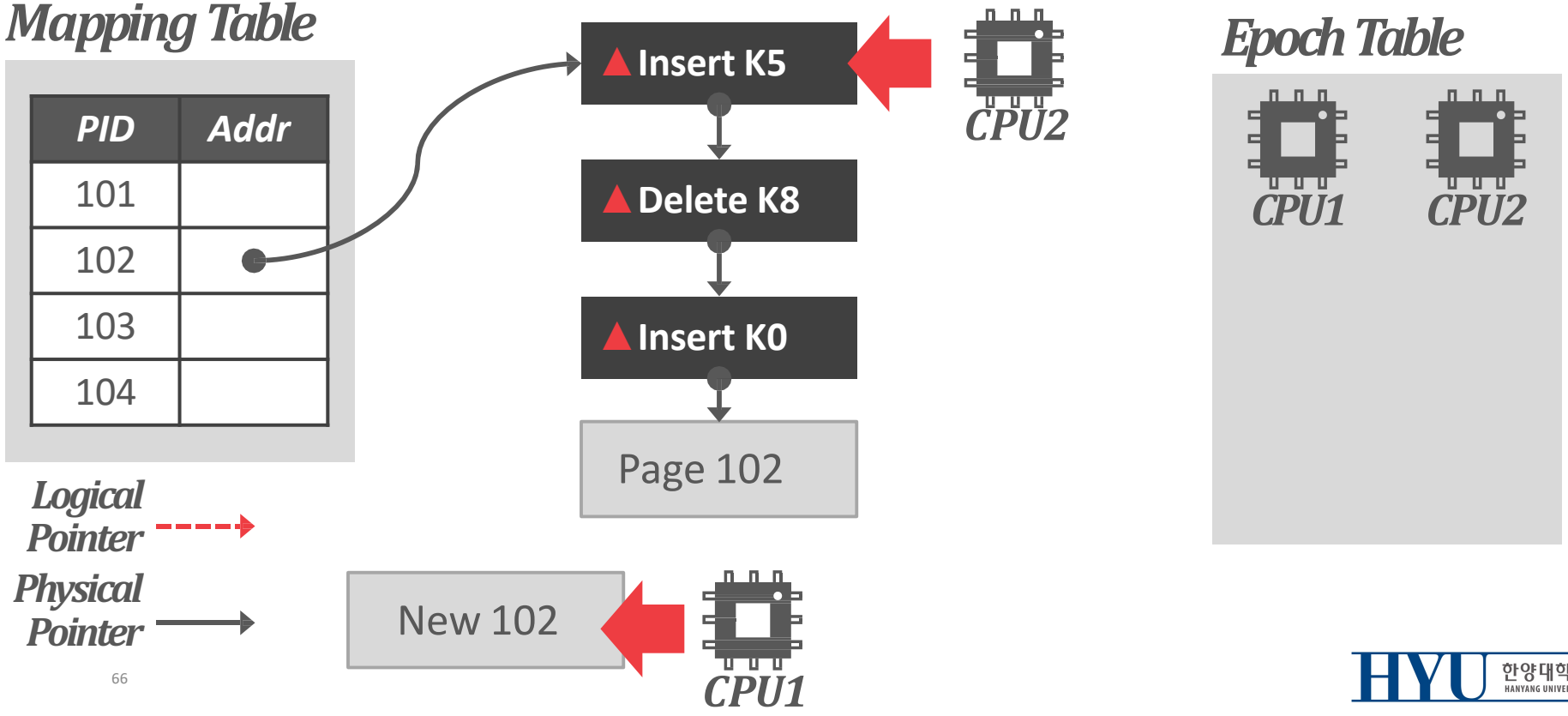
BW-TREE: GARBAGE COLLECTION



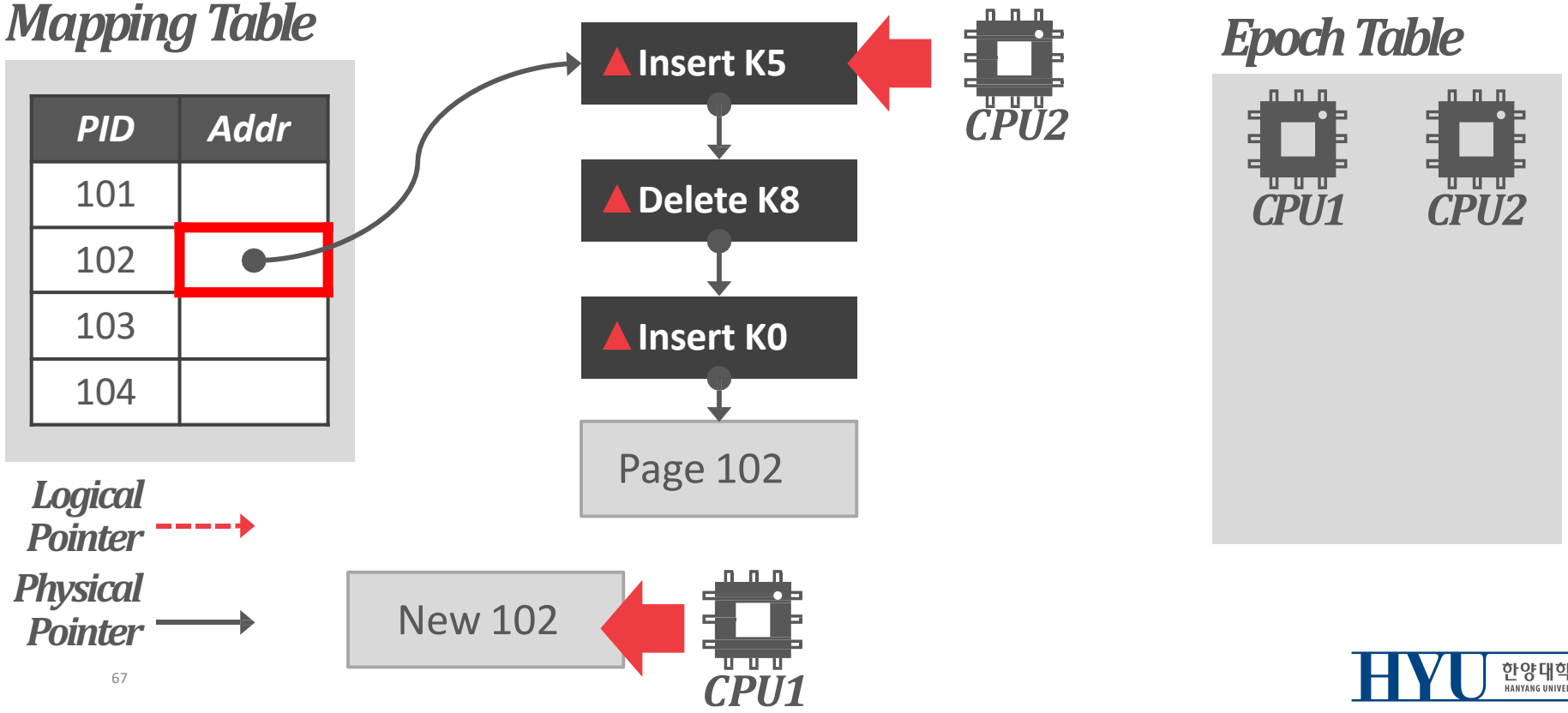
BW-TREE: GARBAGE COLLECTION



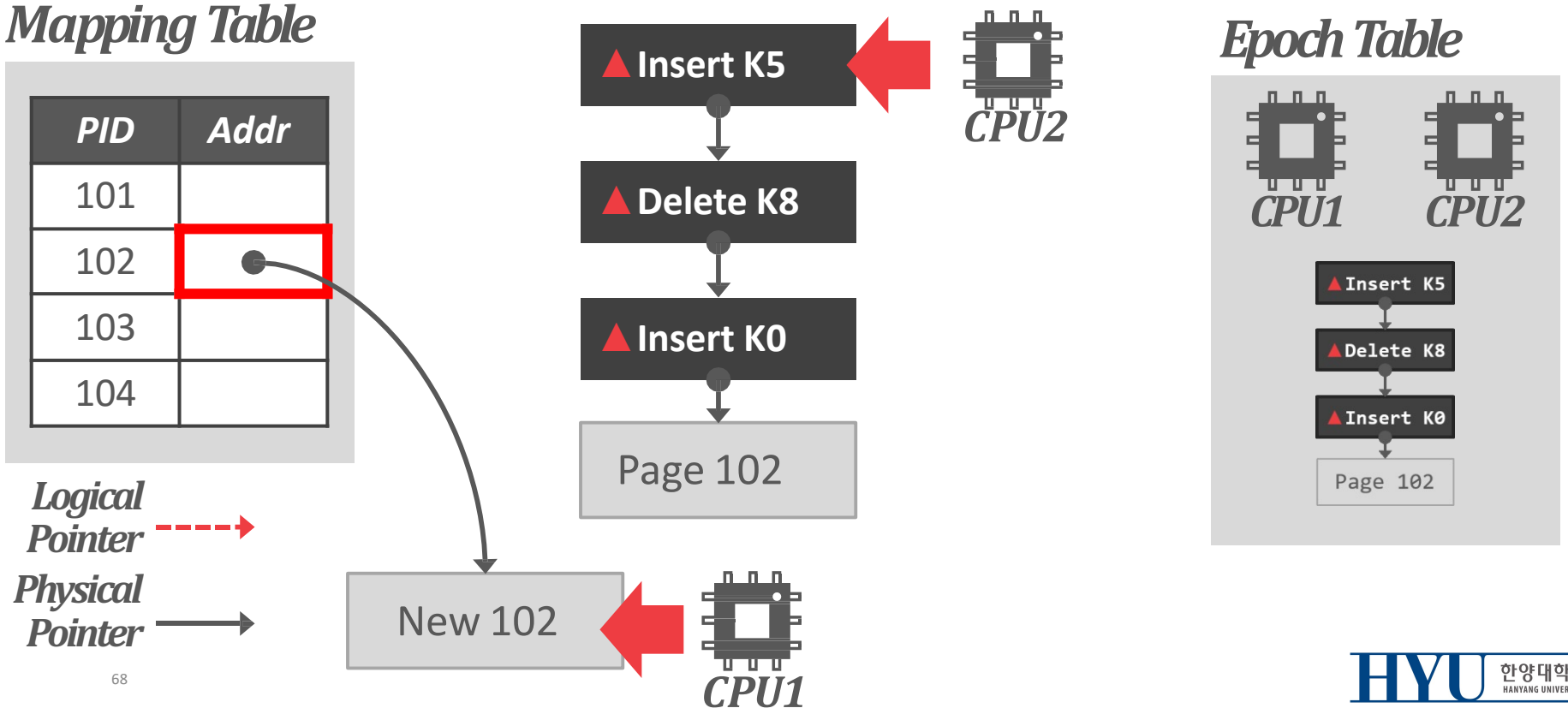
BW-TREE: GARBAGE COLLECTION



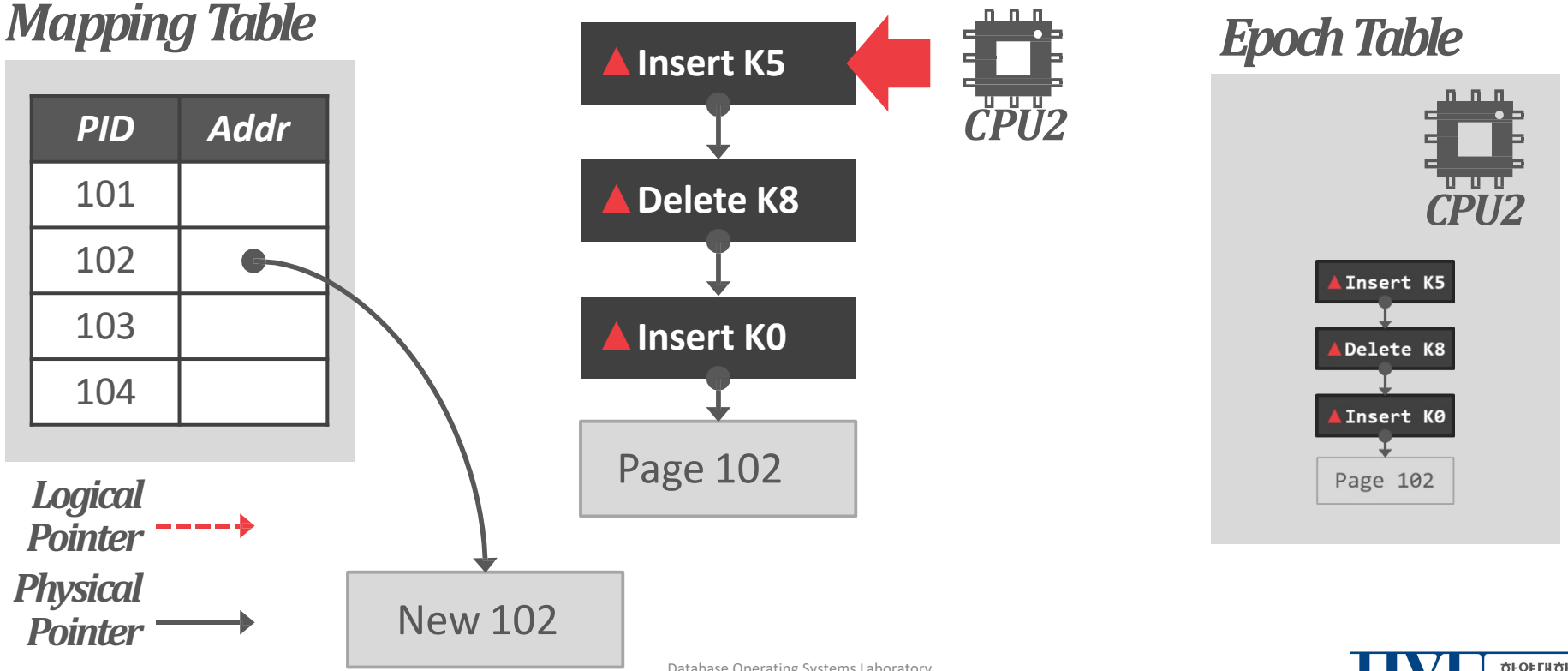
BW-TREE: GARBAGE COLLECTION



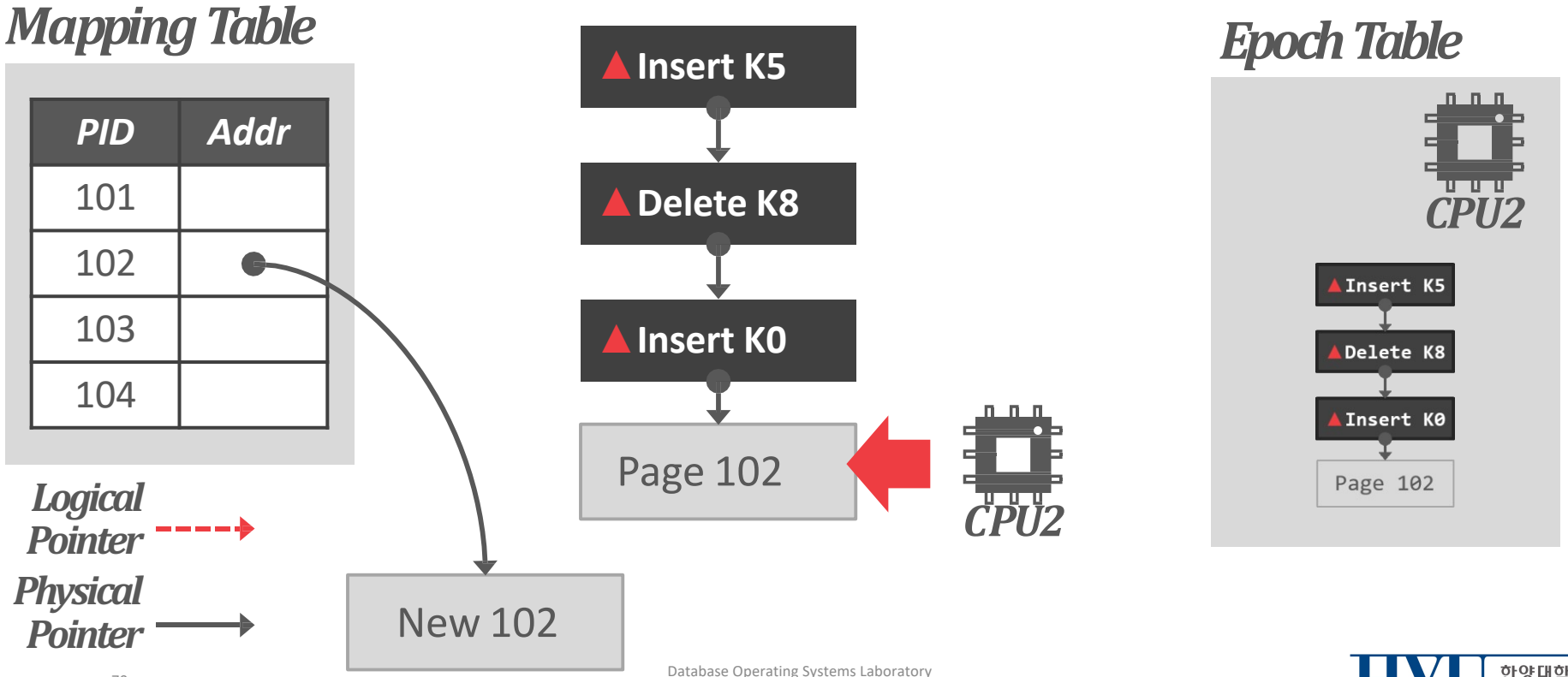
BW-TREE: GARBAGE COLLECTION



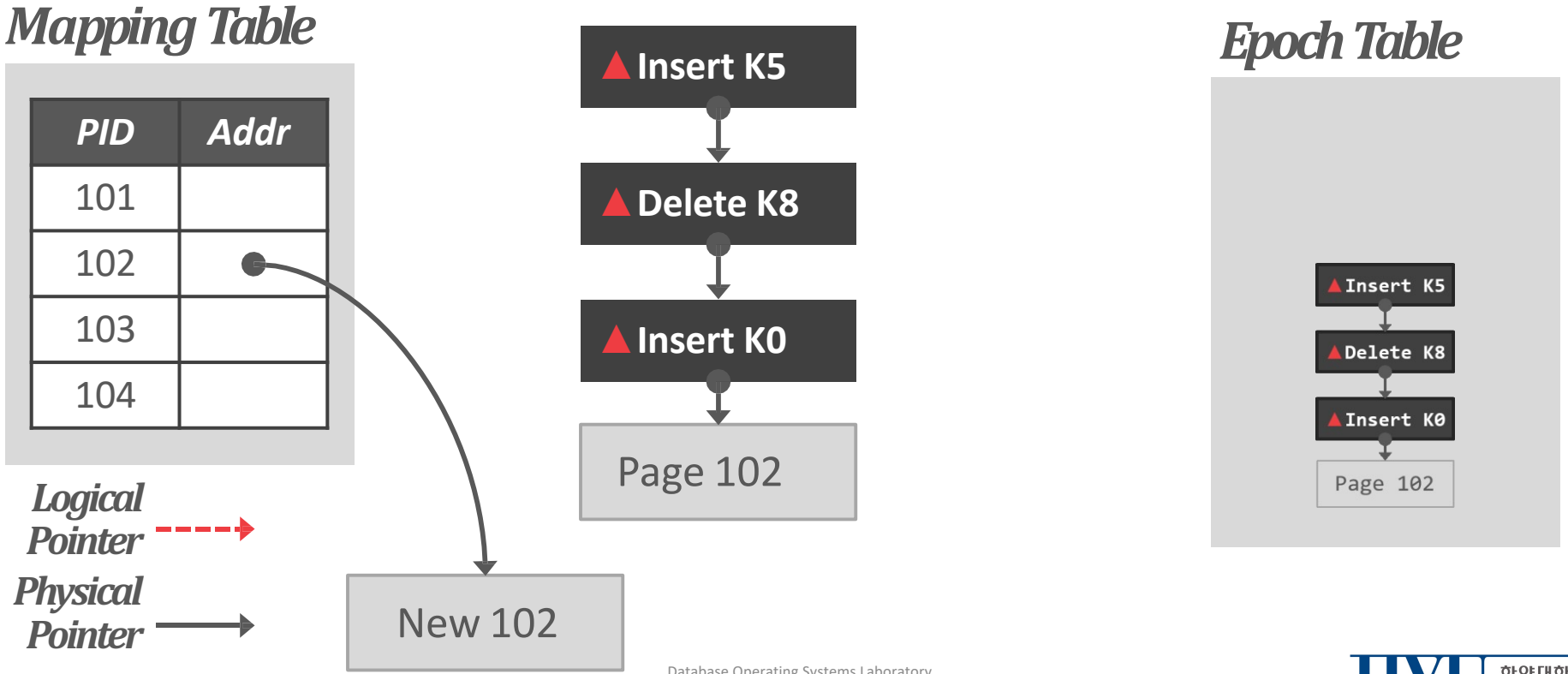
BW-TREE: GARBAGE COLLECTION



BW-TREE: GARBAGE COLLECTION



BW-TREE: GARBAGE COLLECTION



BW-TREE: GARBAGE COLLECTION

Mapping Table

PID	Addr
101	
102	
103	
104	

Insert K5

Delete K8

Insert K0

Page 102

Epoch Table

Logical Pointer

Physical Pointer

New 102

BW-TREE: STRUCTURE MODIFICATIONS

Split Delta Record

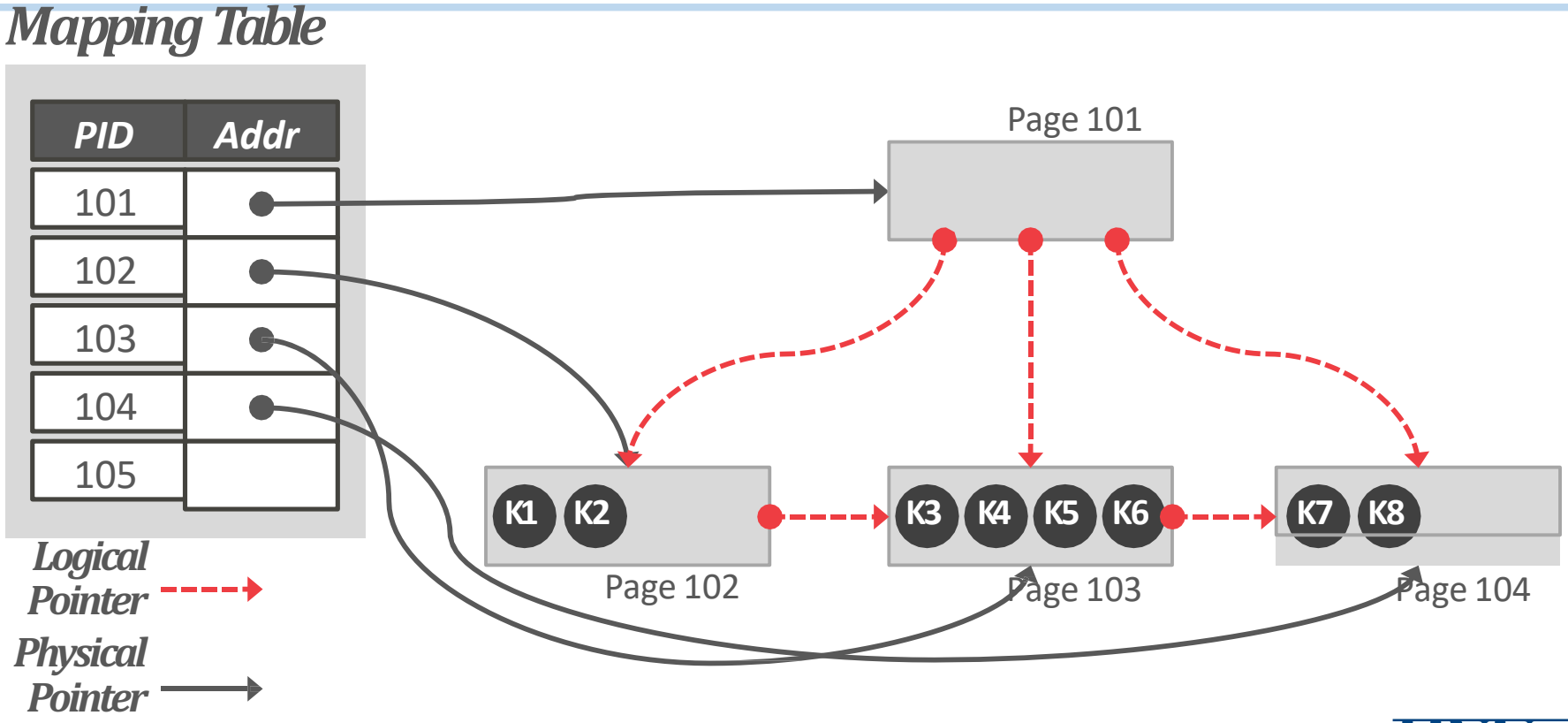
→ Mark that a subset of the base page's key range is now located at another page.

→ Use a logical pointer to the new page.

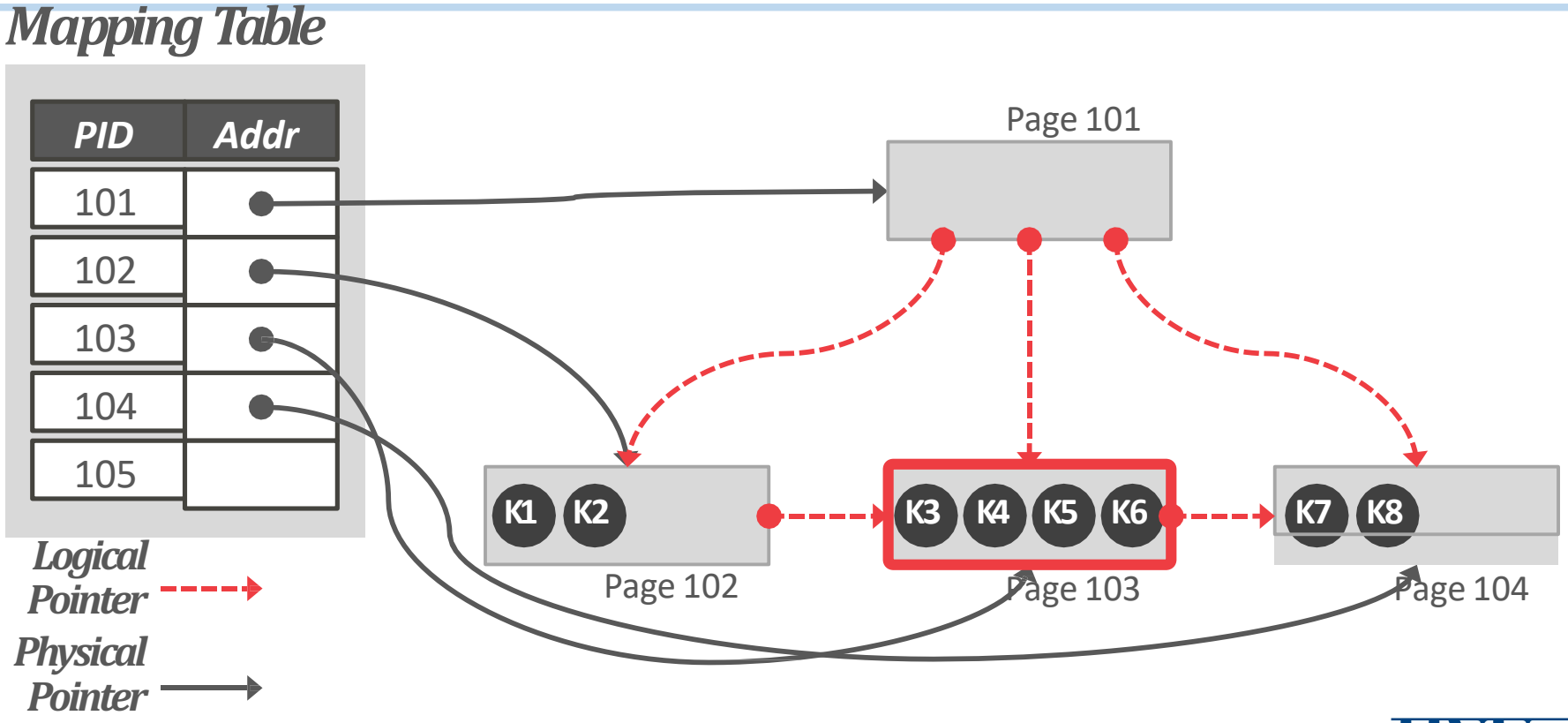
Separator Delta Record

→ Provide a shortcut in the modified page's parent on what ranges to find the new page.

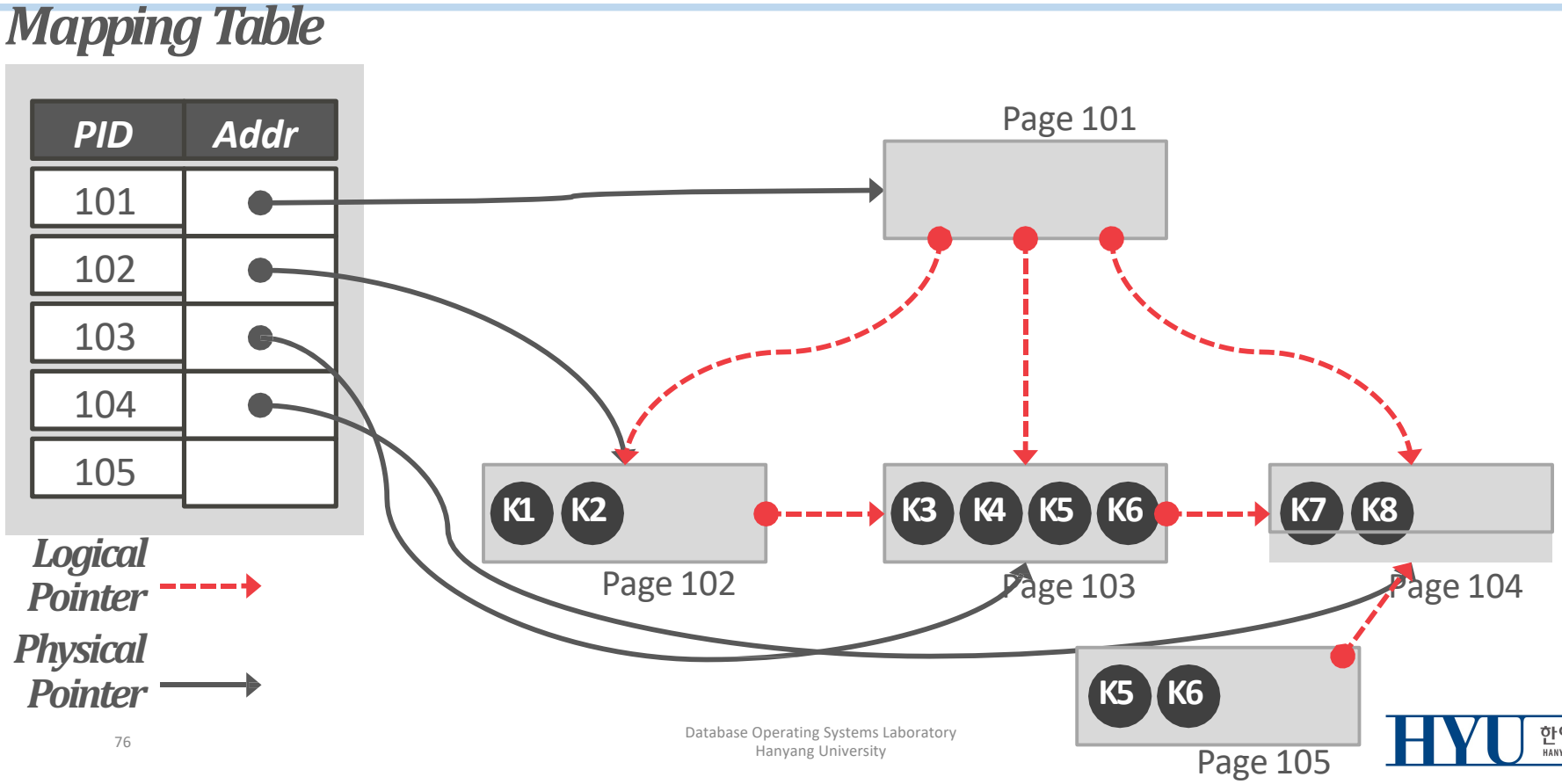
BW-TREE: STRUCTURE MODIFICATIONS



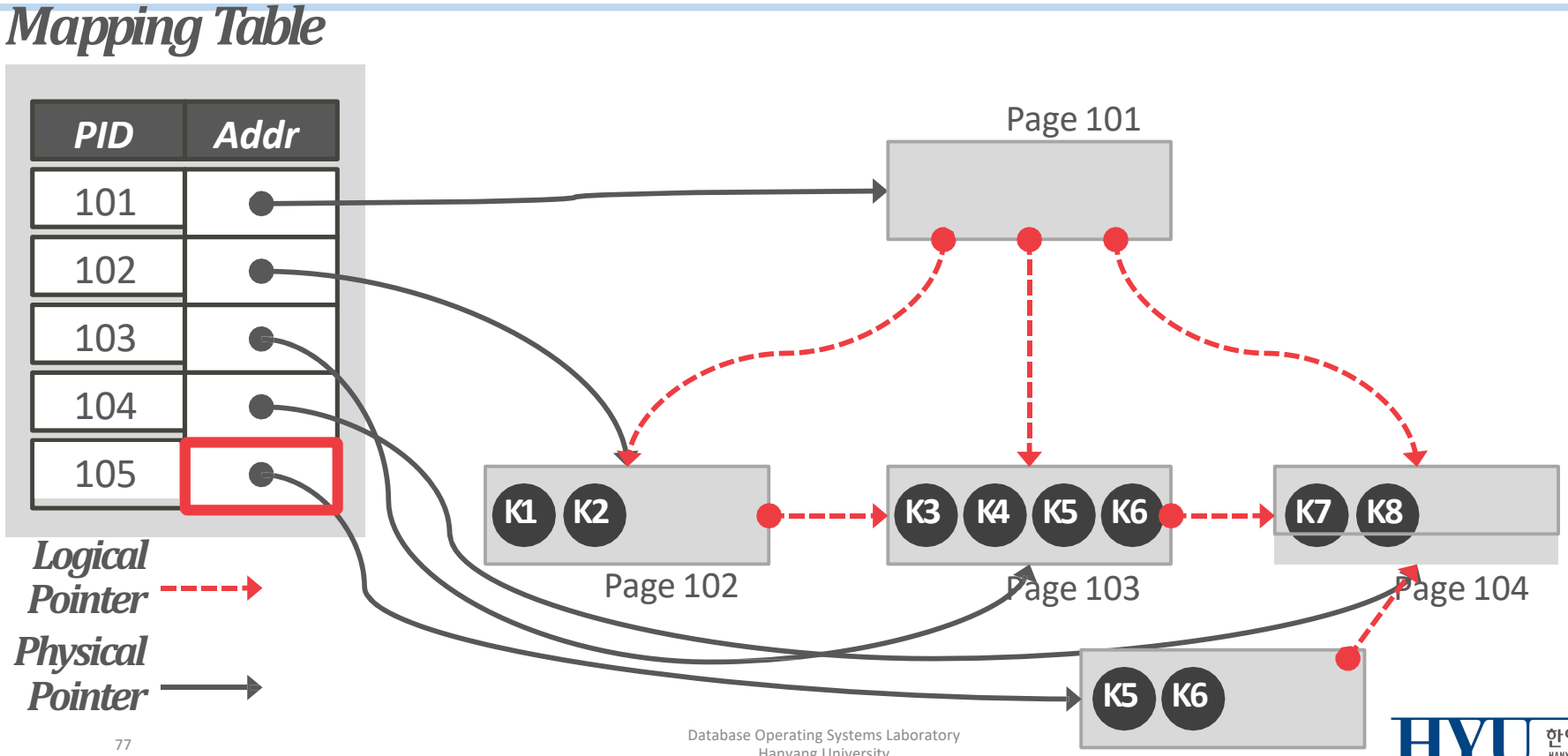
BW-TREE: STRUCTURE MODIFICATIONS



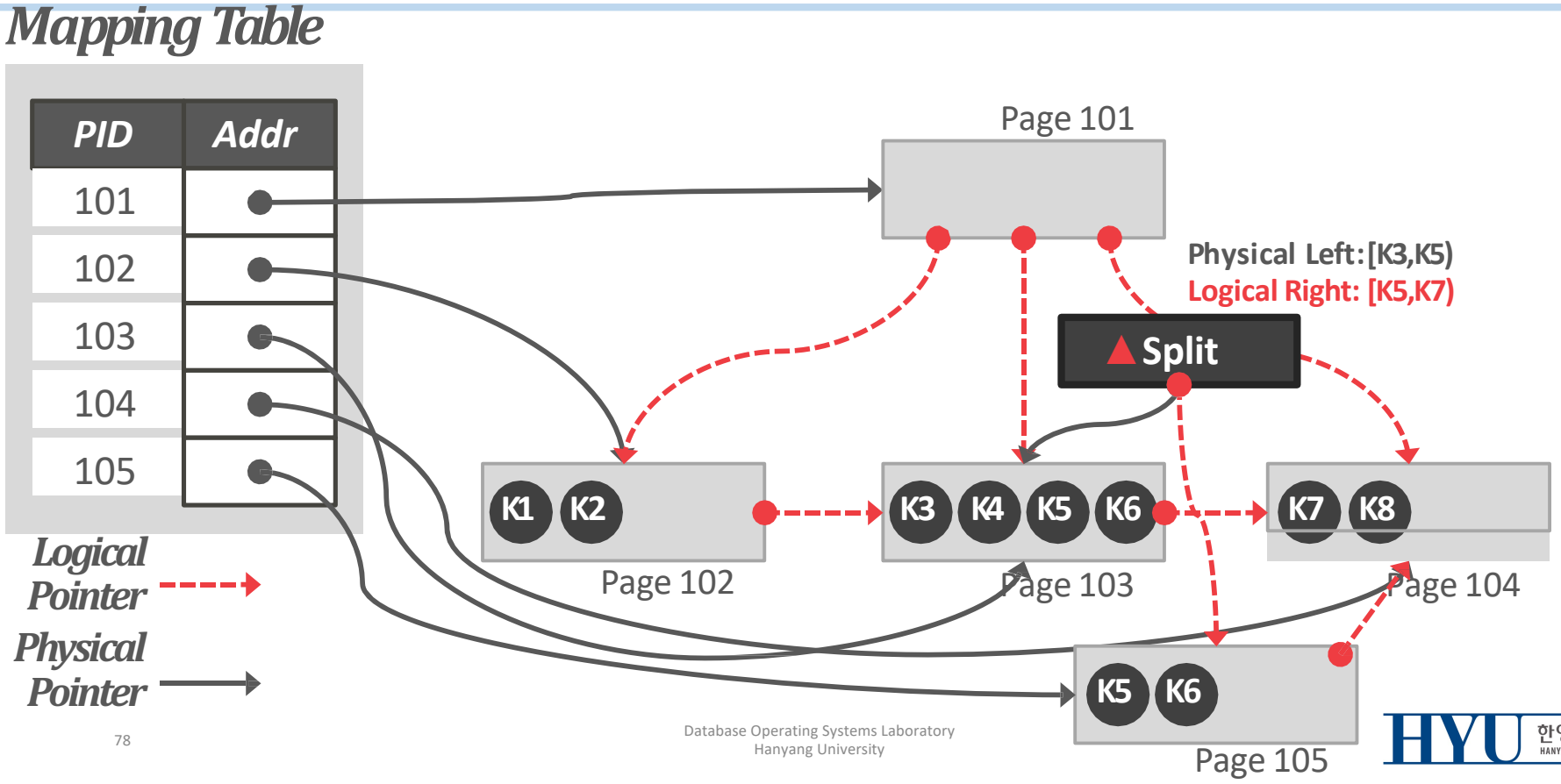
BW-TREE: STRUCTURE MODIFICATIONS



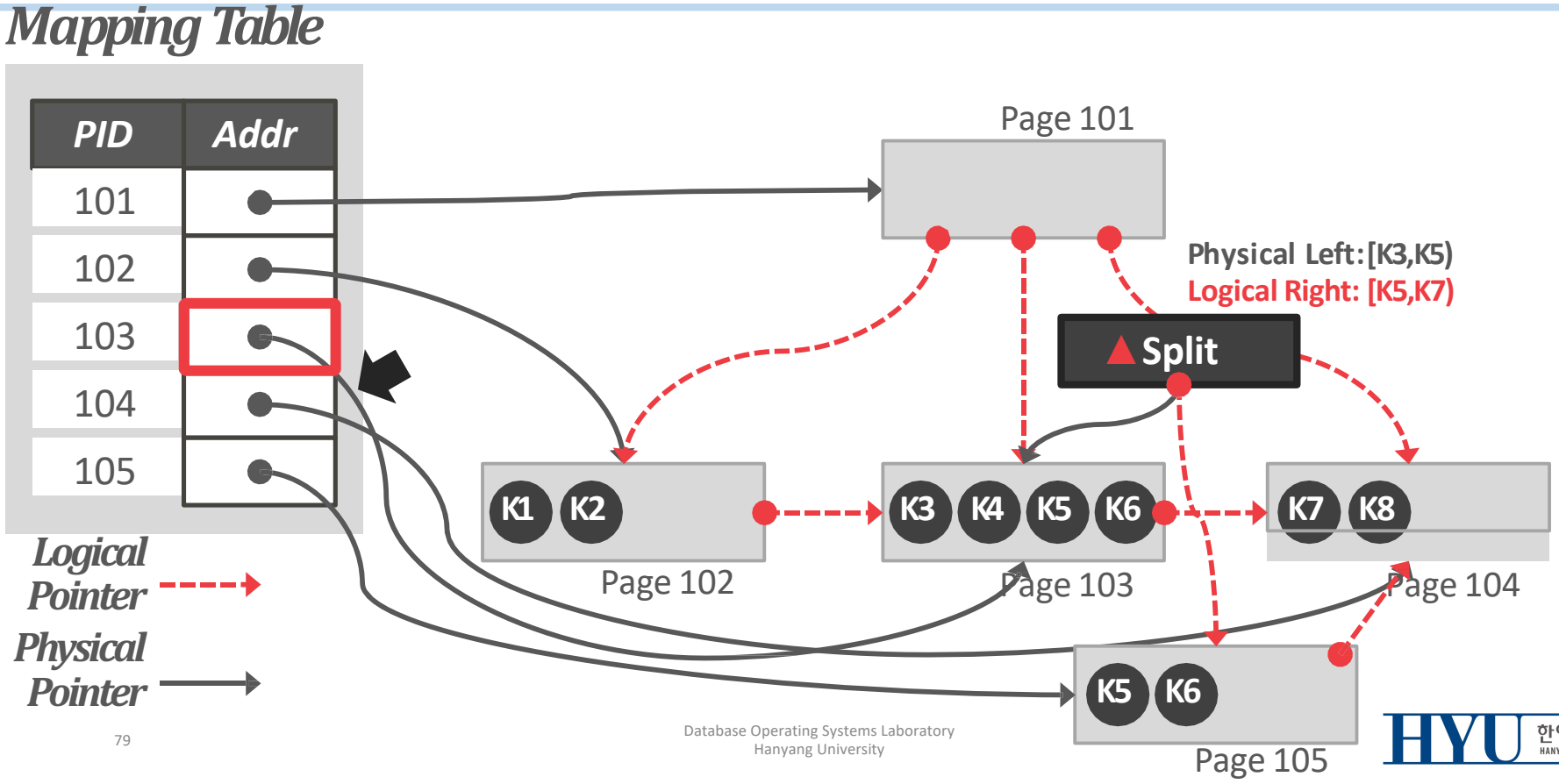
BW-TREE: STRUCTURE MODIFICATIONS



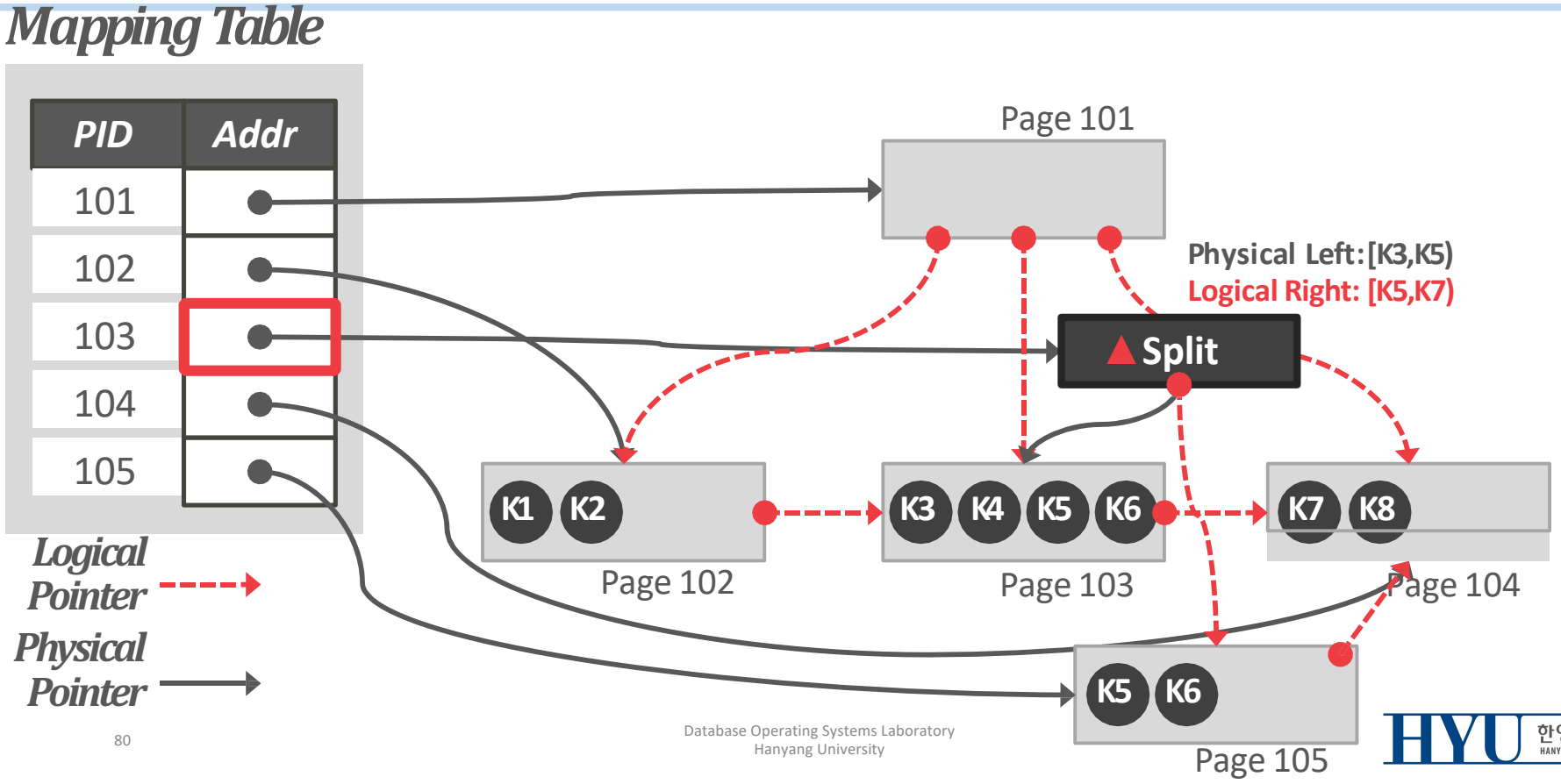
BW-TREE: STRUCTURE MODIFICATIONS



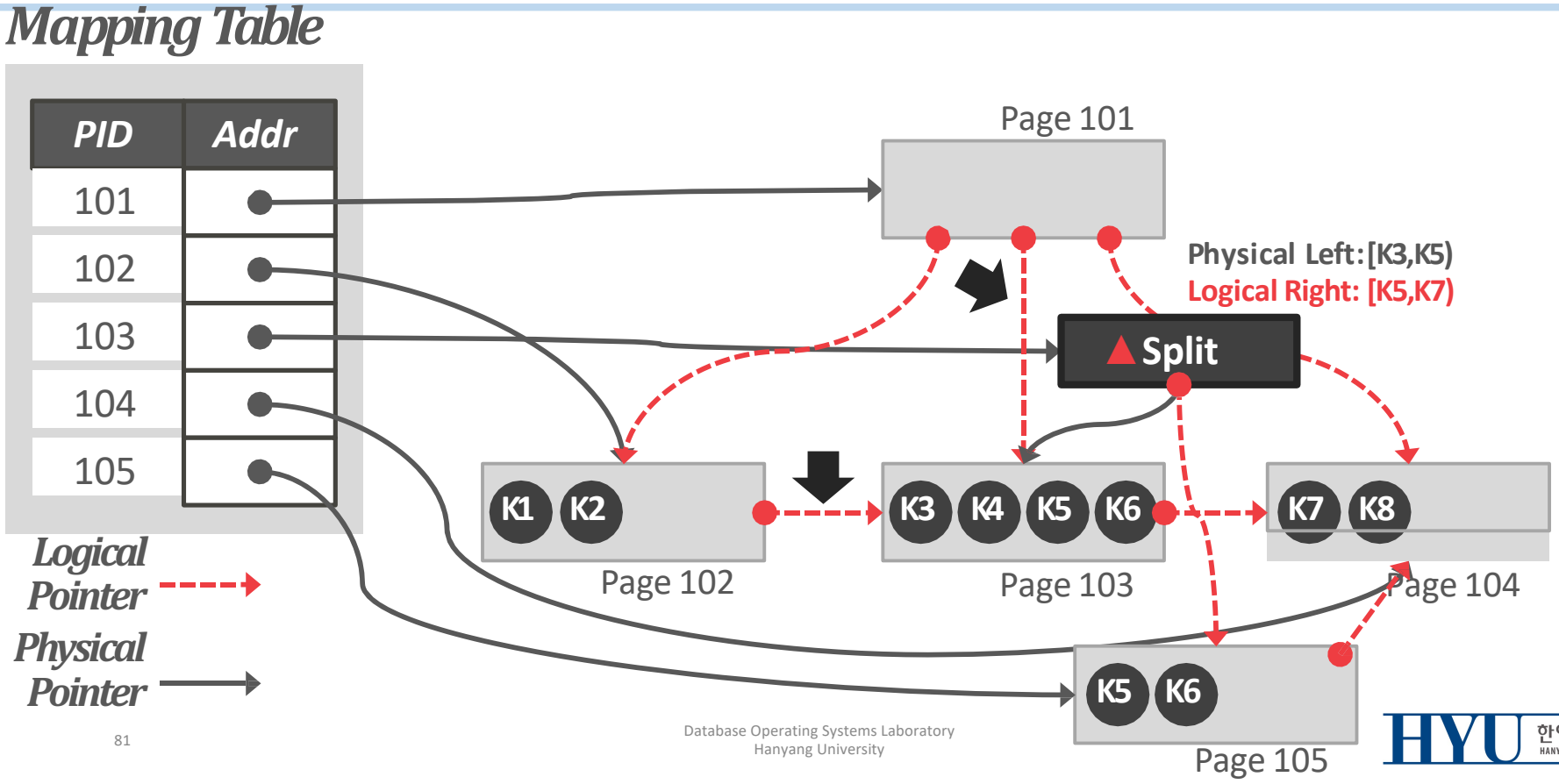
BW-TREE: STRUCTURE MODIFICATIONS



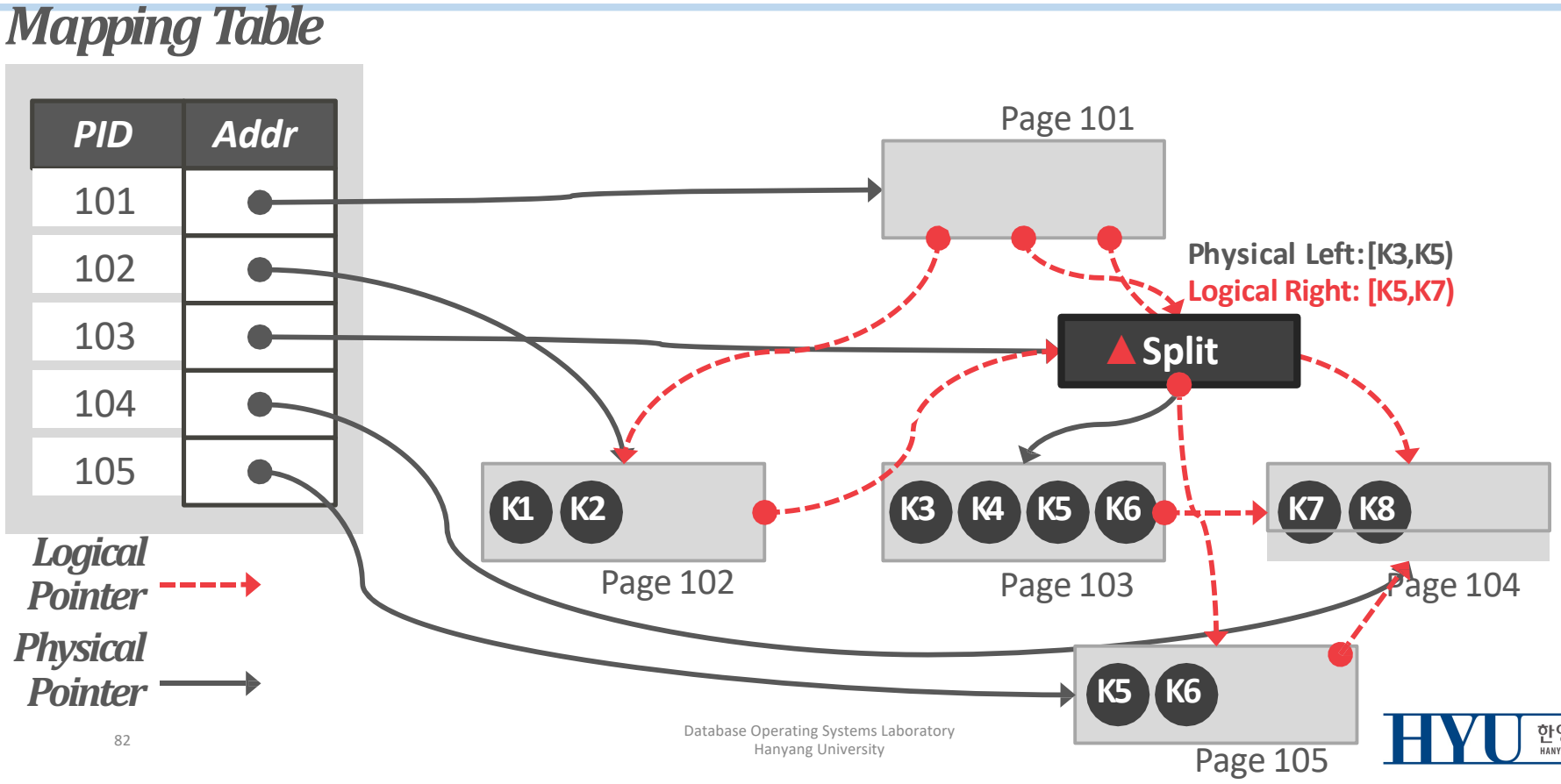
BW-TREE: STRUCTURE MODIFICATIONS



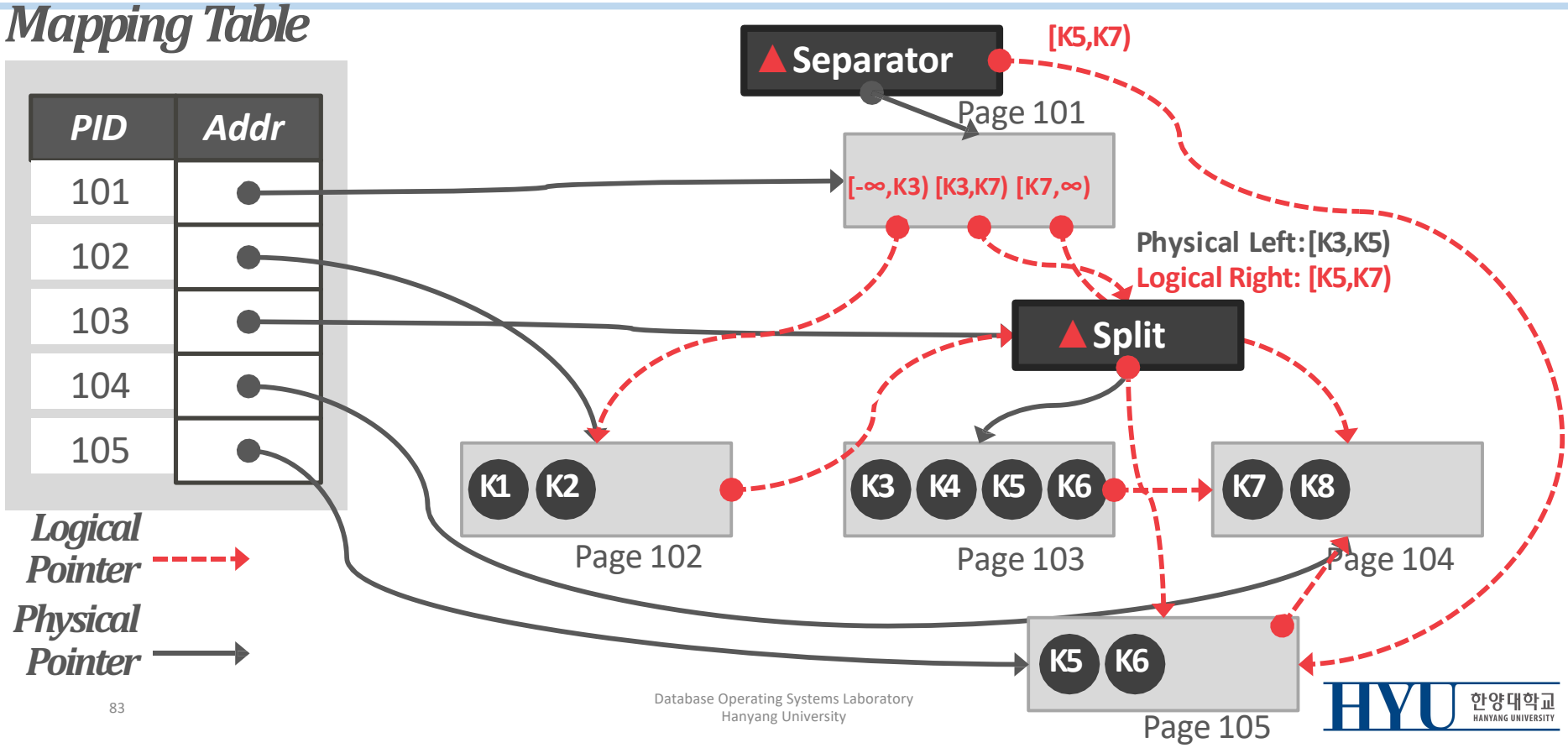
BW-TREE: STRUCTURE MODIFICATIONS



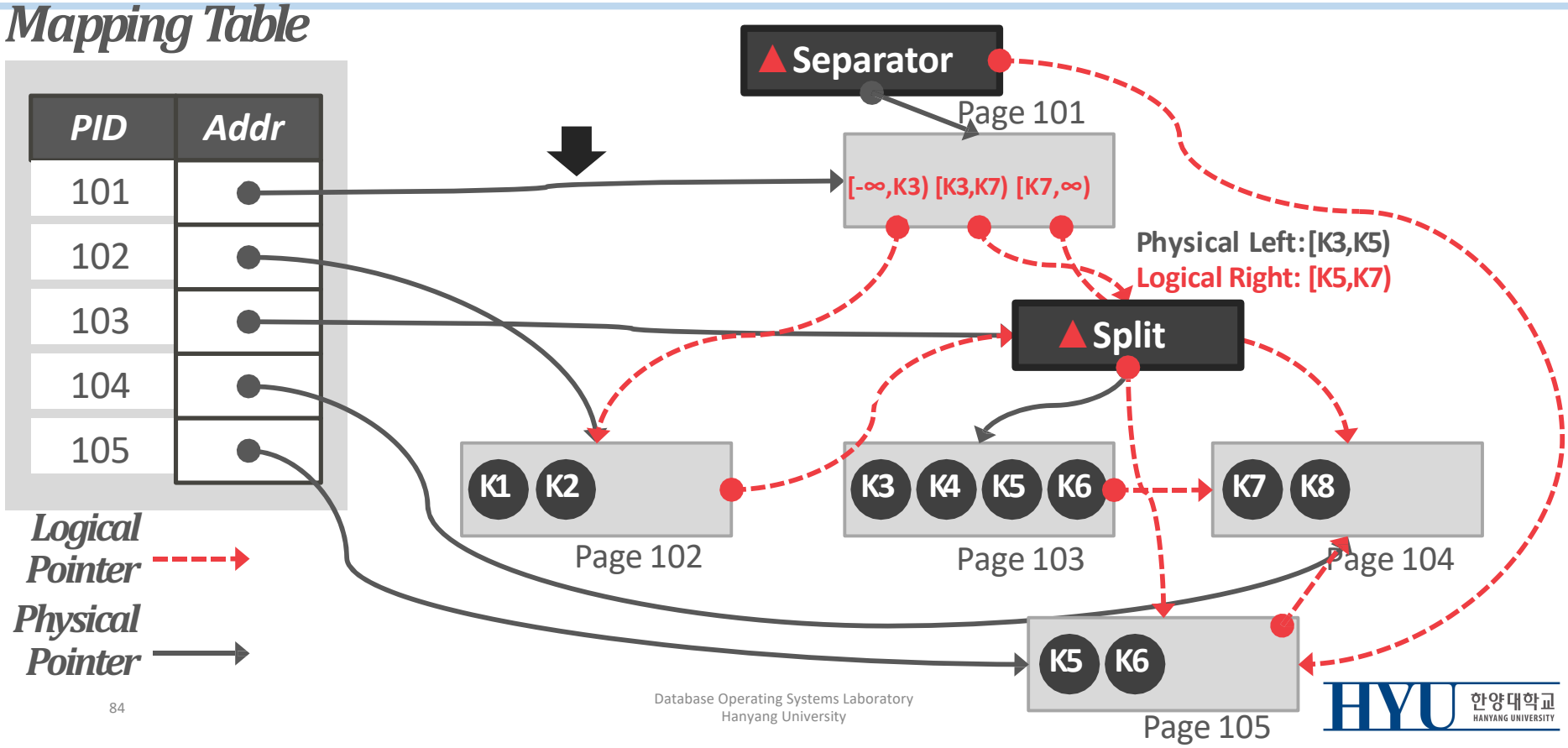
BW-TREE: STRUCTURE MODIFICATIONS



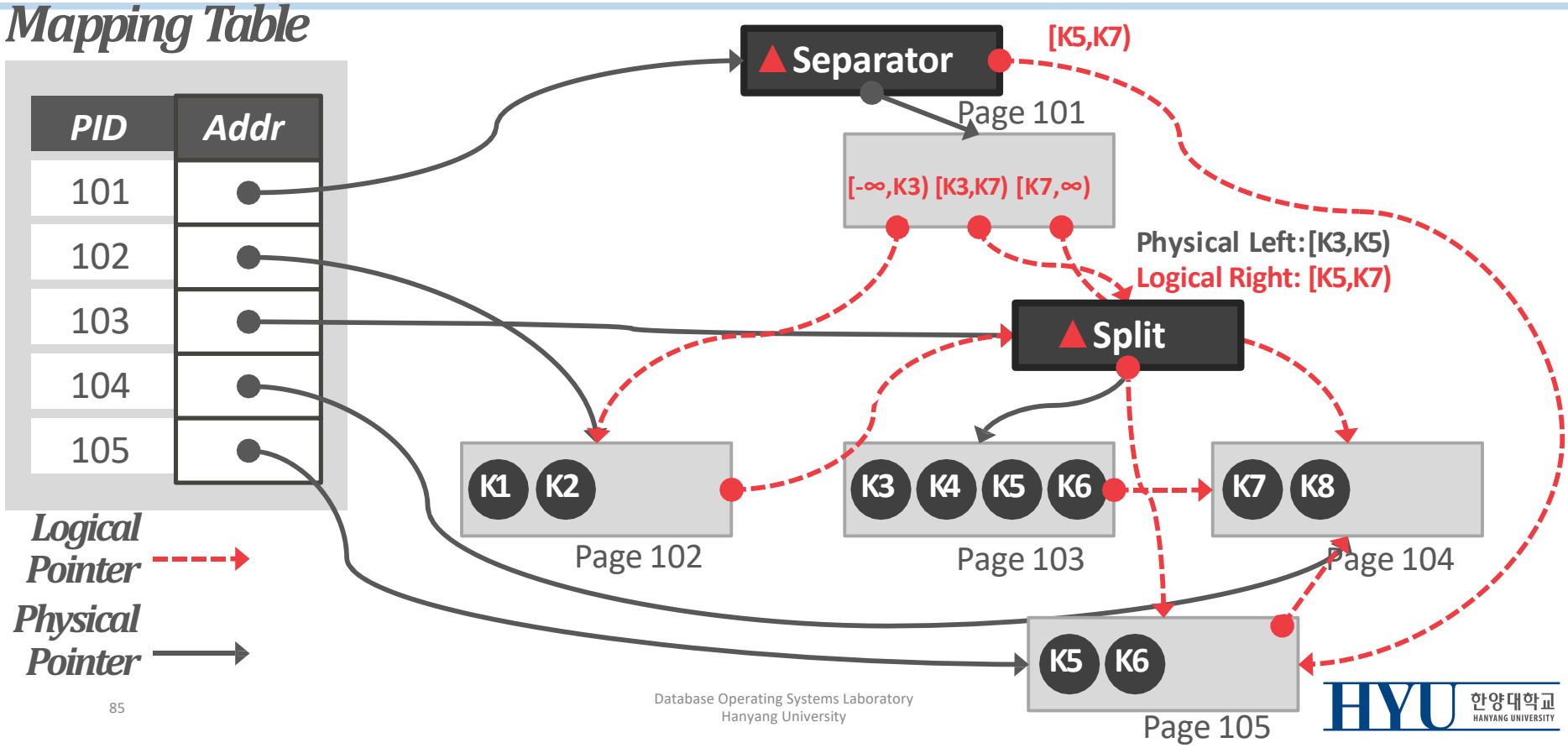
BW-TREE: STRUCTURE MODIFICATIONS



BW-TREE: STRUCTURE MODIFICATIONS



BW-TREE: STRUCTURE MODIFICATIONS



OPEN BW-TREE OPIMIZATIONS

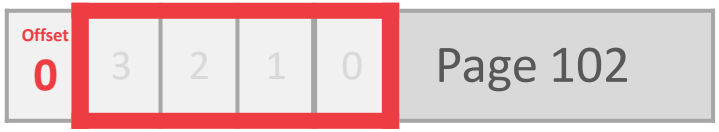
Optimization #1: Pre-Allocated Delta Records

→ Store the delta chain directly inside of a page.

→ Avoids small object allocation, list traversal.

Mapping Table

PID	Addr
102	



Delta Slots

BUILDING THE BW- TREE TAKES MORE THAN JUST BUZZ WORDS
SIGMOD 2018

OPEN BW-TREE OPIMIZATIONS

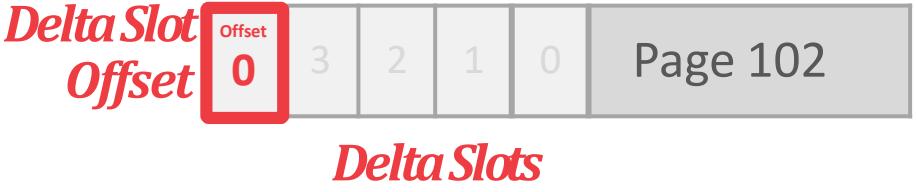
Optimization #1: Pre-Allocated Delta Records

→ Store the delta chain directly inside of a page.

→ Avoids small object allocation, list traversal.

Mapping Table

PID	Addr
102	



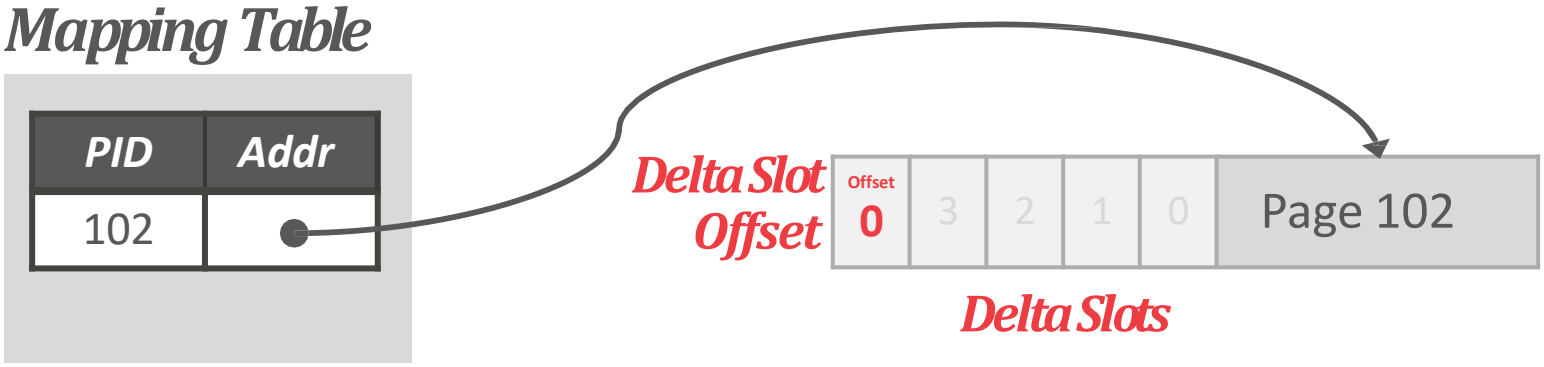
BUILDING THE BW- TREE TAKES MORE THAN JUST BUZZ WORDS
SIGMOD 2018

OPEN BW-TREE OPIMIZATIONS

Optimization #1: Pre-Allocated Delta Records

→ Store the delta chain directly inside of a page.

→ Avoids small object allocation, list traversal.

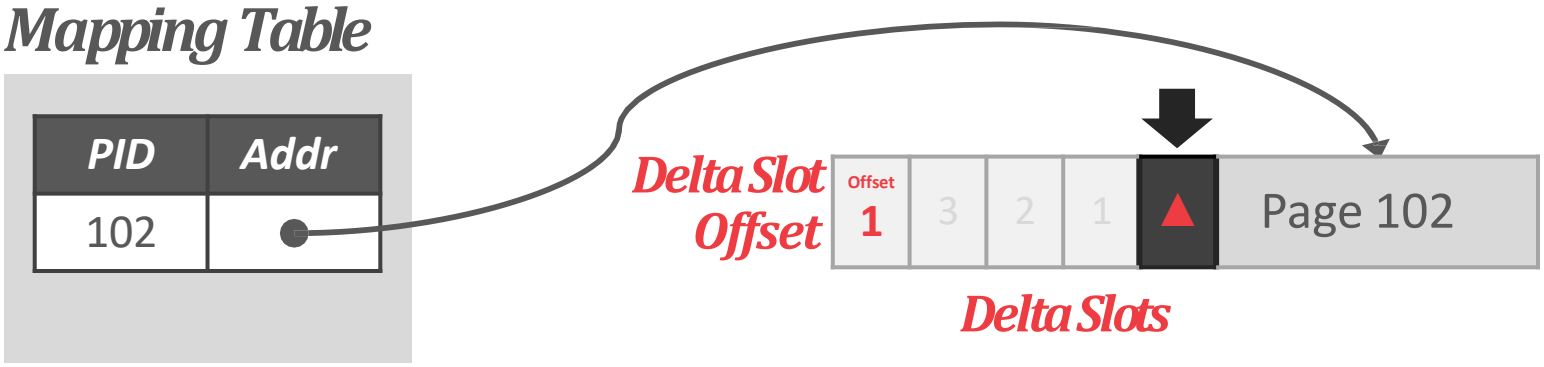


OPEN BW-TREE OPIMIZATIONS

Optimization #1: Pre-Allocated Delta Records

→ Store the delta chain directly inside of a page.

→ Avoids small object allocation, list traversal.



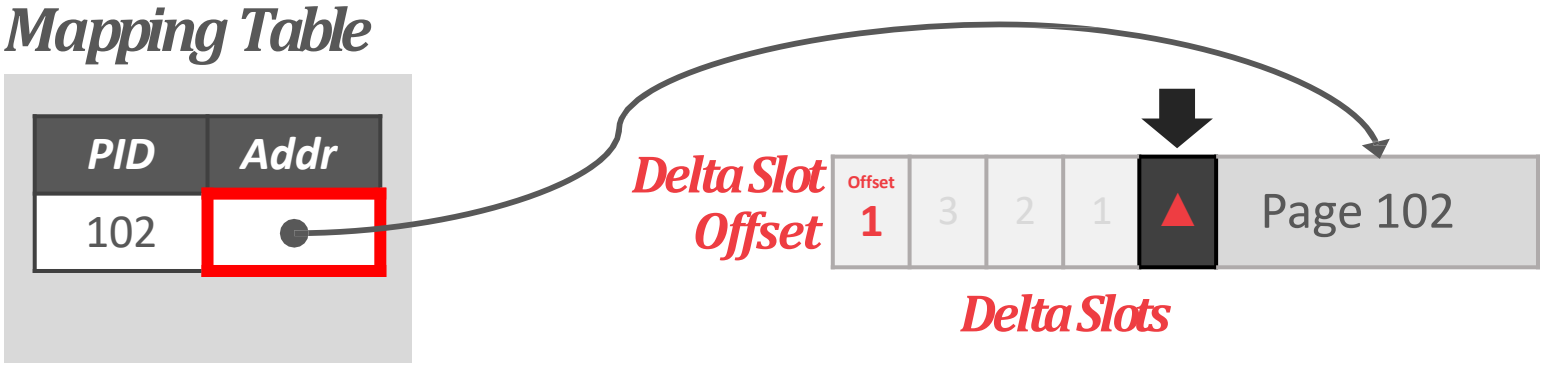
BUILDING THE BW- TREE TAKES MORE THAN JUST BUZZ WORDS
SIGMOD 2018

OPEN BW-TREE OPIMIZATIONS

Optimization #1: Pre-Allocated Delta Records

→ Store the delta chain directly inside of a page.

→ Avoids small object allocation, list traversal.

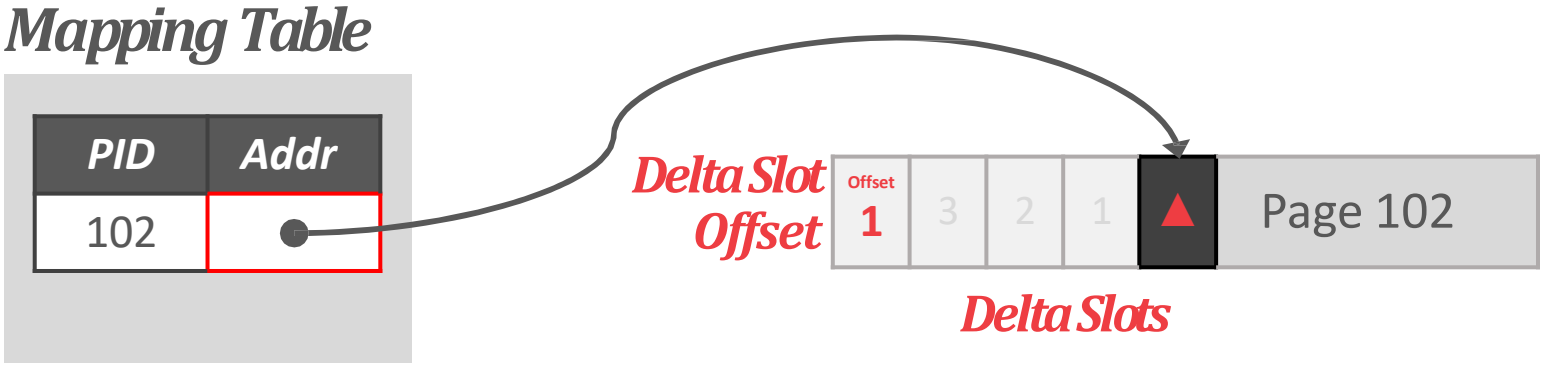


OPEN BW-TREE OPIMIZATIONS

Optimization #1: Pre-Allocated Delta Records

→ Store the delta chain directly inside of a page.

→ Avoids small object allocation, list traversal.



BUILDING THE BW- TREE TAKES MORE THAN JUST BUZZ WORDS
SIGMOD 2018

OPEN BW-TREE OPIMIZATIONS

Optimization #2: Mapping Table Expansion

→ Fastest associative data structure is a plain array.

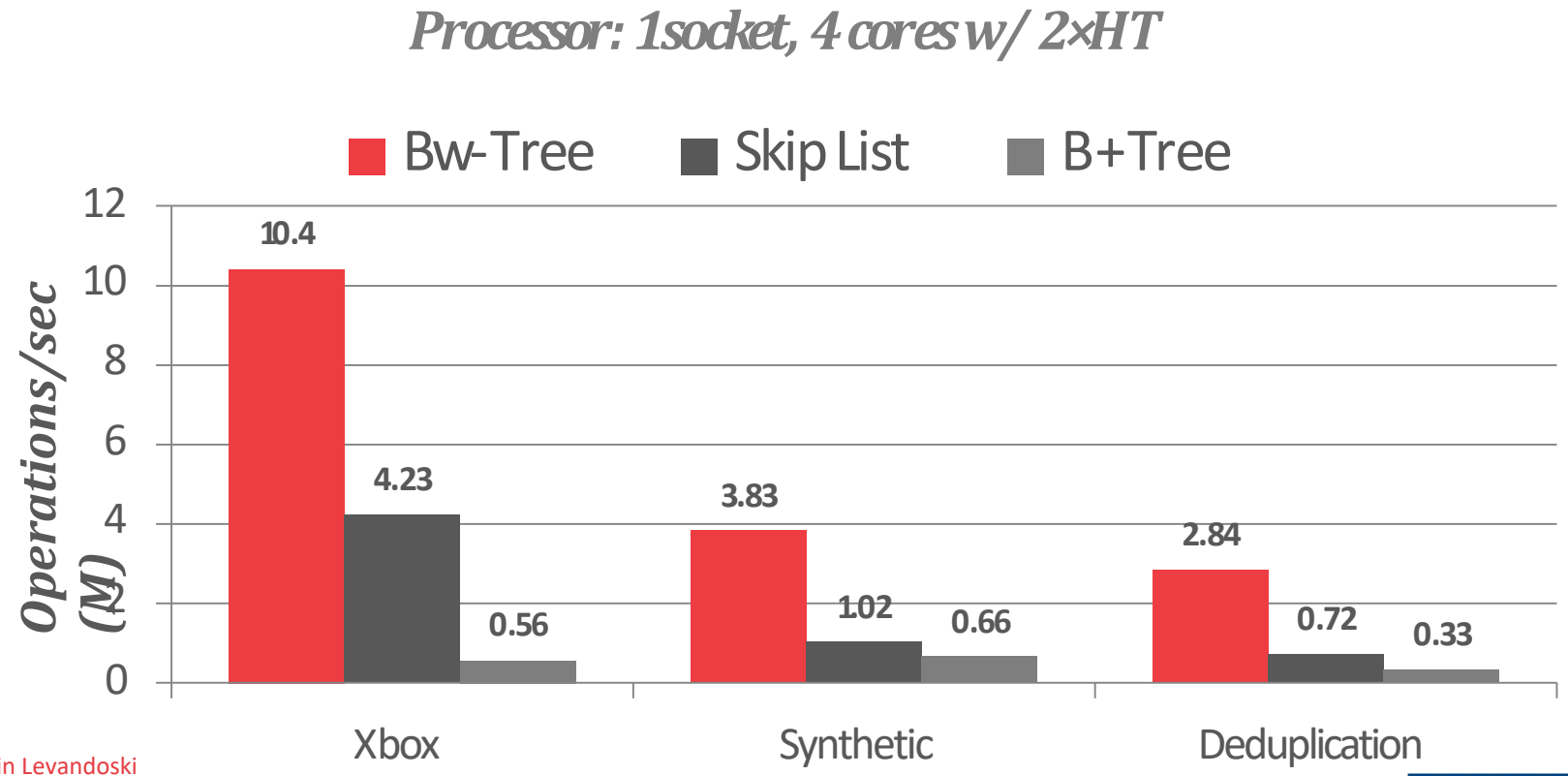
→ Allocating the full array for each index is wasteful

Use virtual memory to allocate the entire array without backing it with physical memory.

→ Only need to allocate physical memory when threads access higher offsets in the array.

BUILDING THE BW- TREE TAKES MORE THAN JUST BUZZ WORDS
SIGMOD 2018

BW-TREE: PERFORMANCE



Source: Justin Levandoski

Thank You