Carnegie Mellon University

Lecture #08

ADVANCED DATABASE SYSTEMS

OLTP Indexes (Part I)

@Andy_Pavlo // 15-721 // Spring 2018

ADMINISTRIVIA

Peloton master branch has been updated to provide easier to use debug methods.

→ Your implementation should match the behavior of the Bw-Tree.

We will be sending out information on how to access the MemSQL development machines.

TODAY'S AGENDA

T-Tree

Skip List

Bw-Tree

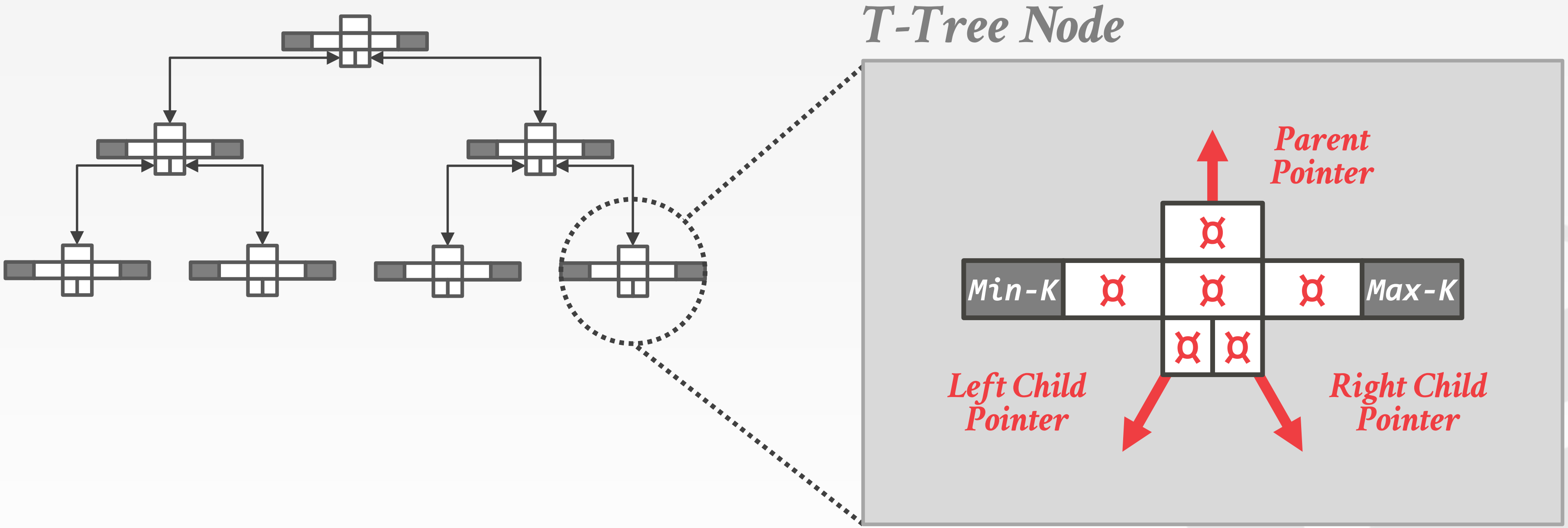
T-TREES

Based on AVL Trees. Instead of storing keys in nodes, store pointers to their original values.

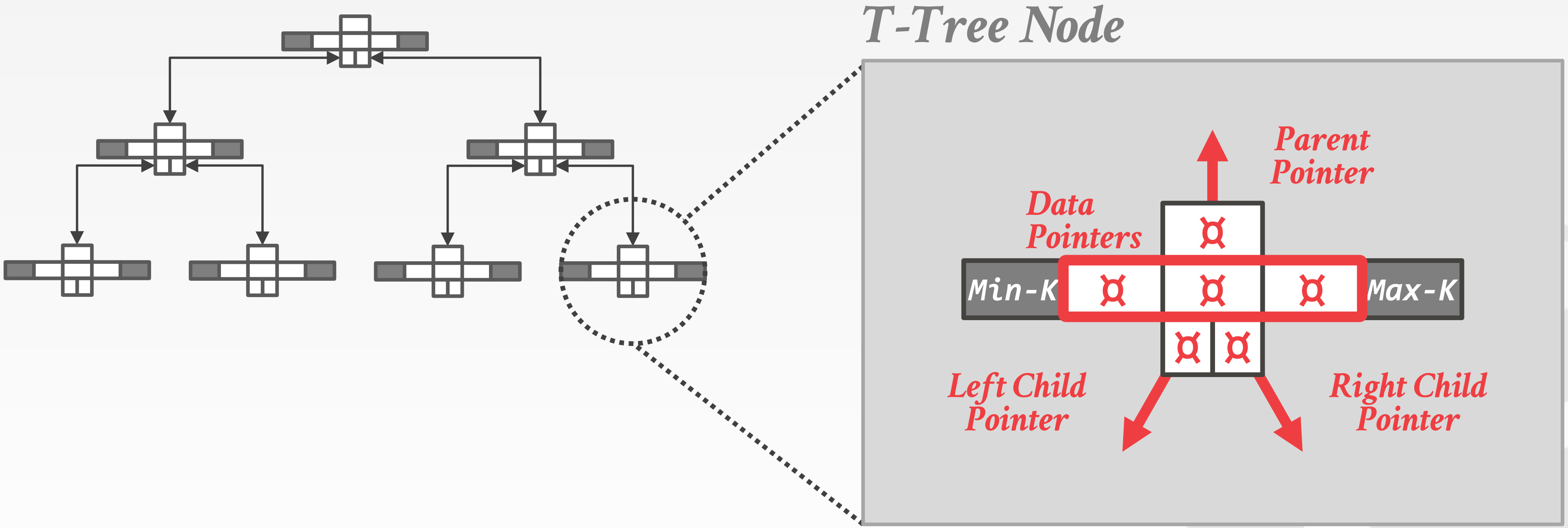
Proposed in 1986 from Univ. of Wisconsin
Used in TimesTen and other early in-memory DBMSs during the 1990s.

A STUDY OF INDEX STRUCTURES FOR MAIN MEMORY DATABASE MANAGEMENT SYSTEMS
VLDB 1986

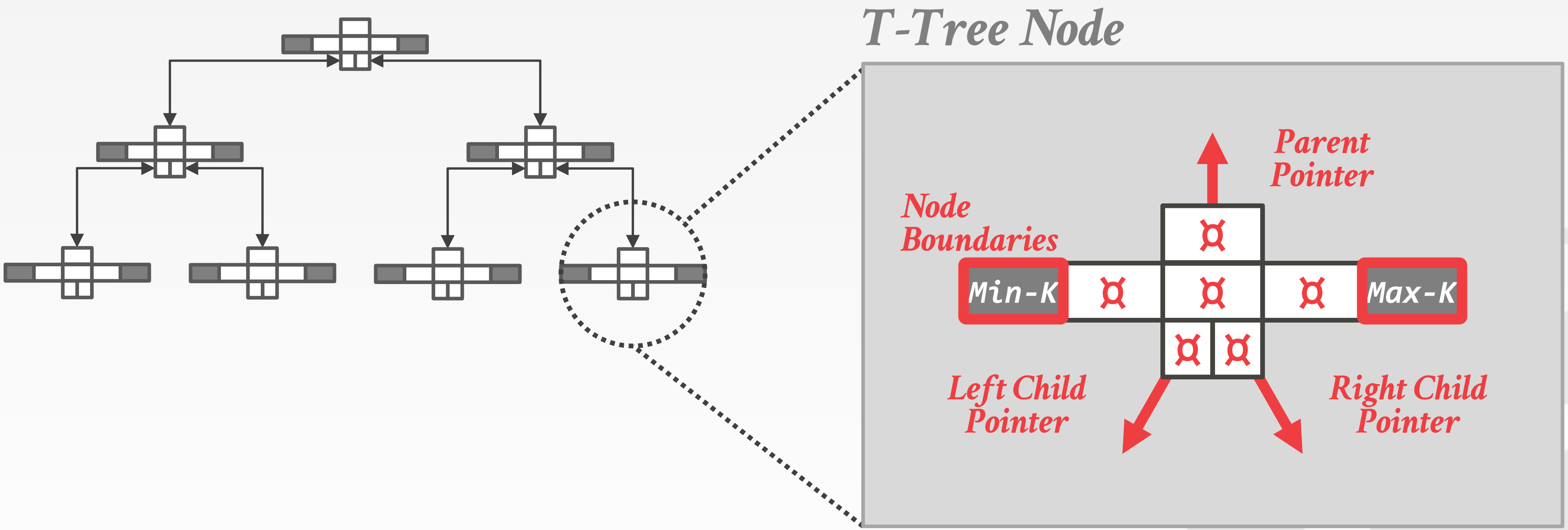
T-TREES


T-Tree Node
Parent Pointer
Min-K
Max-K
¤
¤
¤
¤
¤
¤
Left Child Pointer
Right Child Pointer

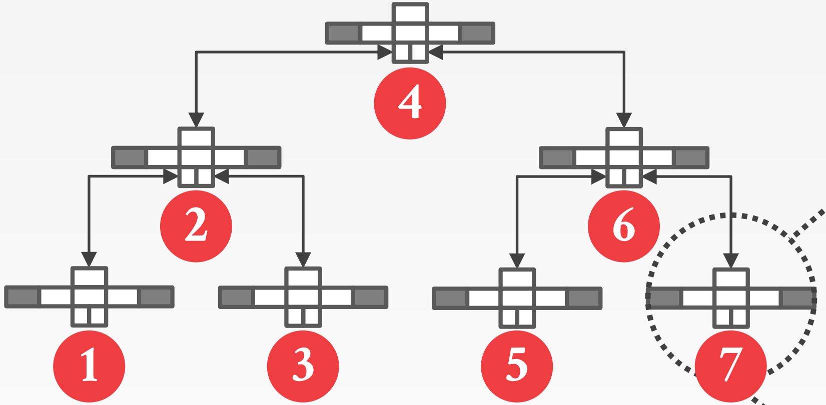
T-TREES


T-Tree Node
Parent Pointer
Data Pointers
Min-K
Max-K
Left Child Pointer
Right Child Pointer

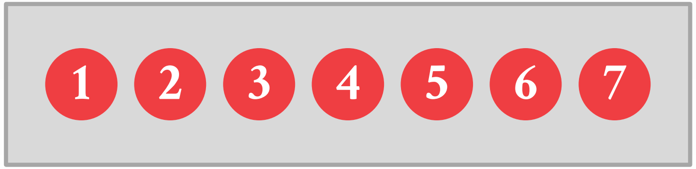
T-TREES


T-Tree Node
Parent Pointer
Node Boundaries
Min-K
¤
¤
¤
¤
¤
¤
Max-K
Left Child Pointer
Right Child Pointer

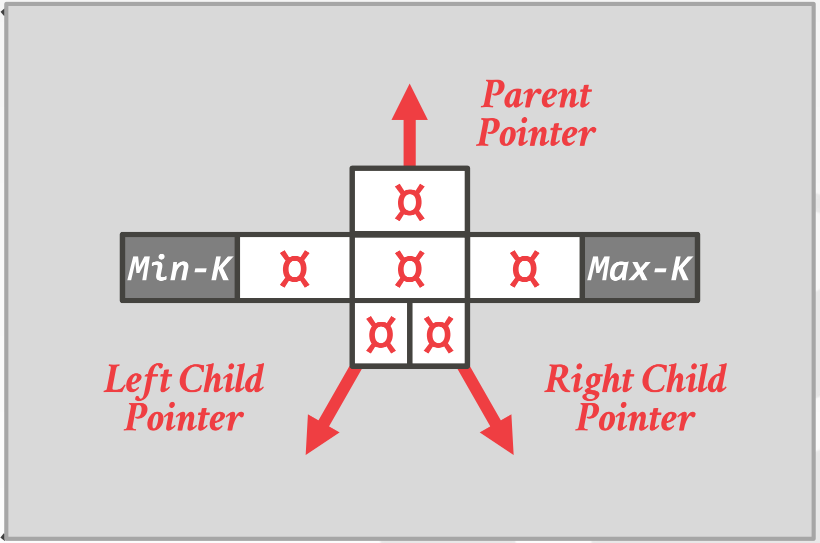
T-TREES



Key Space (Low→High)



T-Tree Node



T-TREES

Advantages

→ Uses less memory because it does not store keys inside of each node.

→ Inner nodes contain key/value pairs (like B-Tree).

Disadvantages

→ Difficult to rebalance.

→ Difficult to implement safe concurrent access.

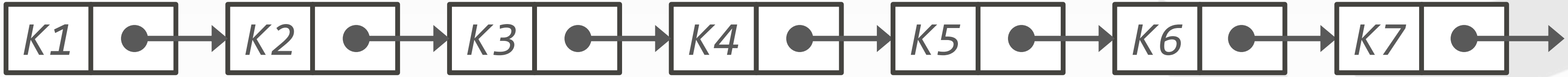
→ Have to chase pointers when scanning range or performing binary search inside of a node.

OBSERVATION

The easiest way to implement a **dynamic** order-preserving index is to use a sorted linked list.

All operations have to linear search.

→ Average Cost: O(N)

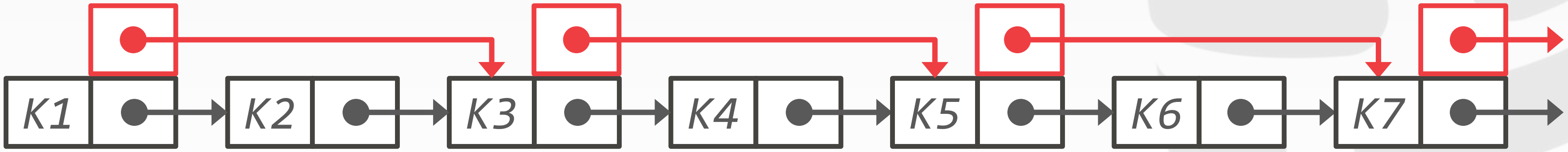


OBSERVATION

The easiest way to implement a **dynamic** order-preserving index is to use a sorted linked list.

All operations have to linear search.

→ Average Cost: O(N)

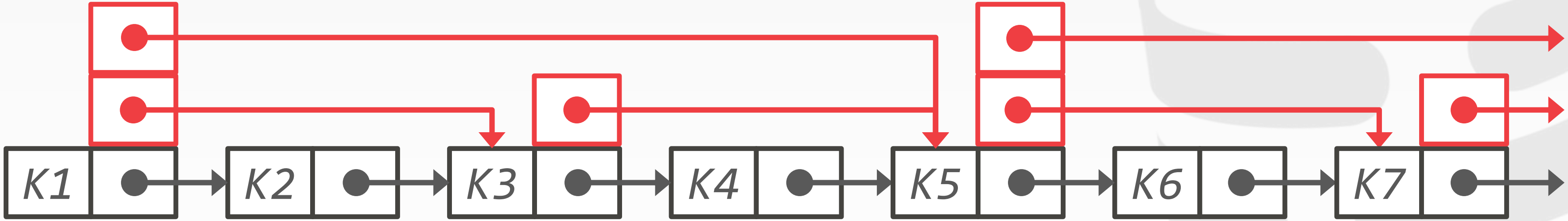


OBSERVATION

The easiest way to implement a **dynamic** order-preserving index is to use a sorted linked list.

All operations have to linear search.

→ Average Cost: O(N)



SKIP LISTS

Multiple levels of linked lists with extra pointers that **skip** over intermediate nodes.

Maintains keys in sorted order without requiring global rebalancing.

SKIP LISTS: A PROBABILISTIC ALTERNATIVE TO BALANCED TREES
CACM Volume 33 Issue 6 1990

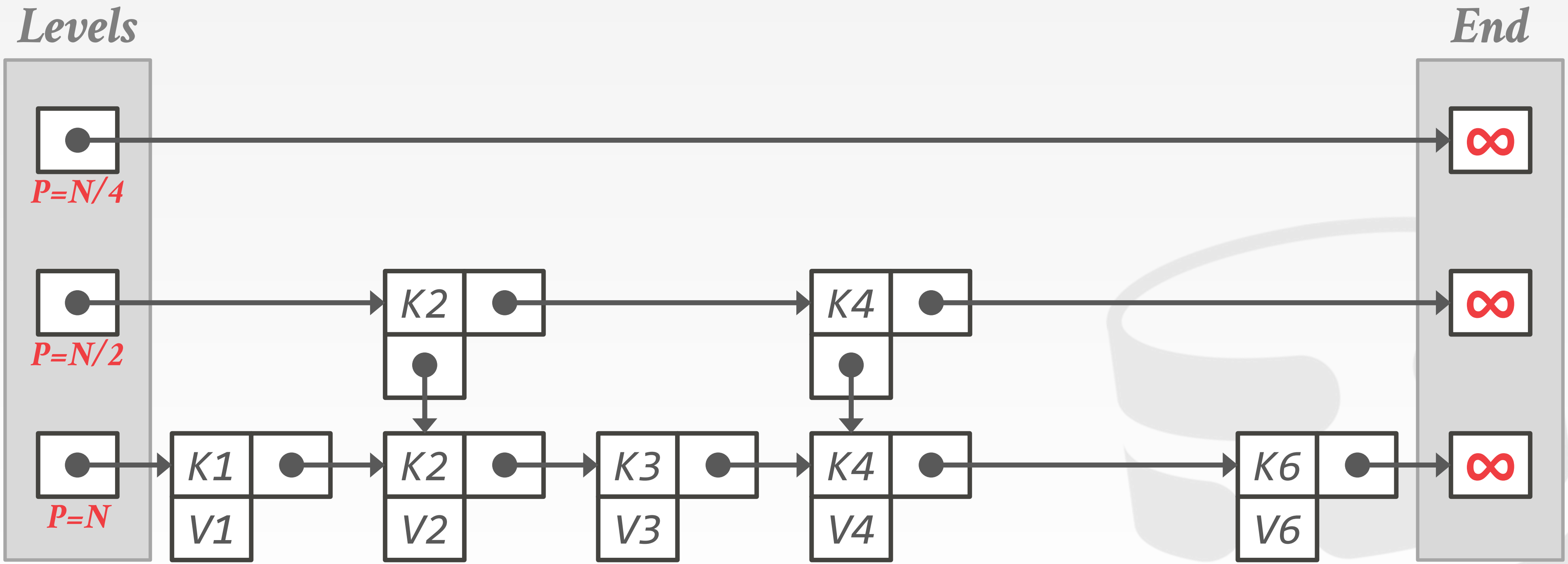
SKIP LISTS

A collection of lists at different levels

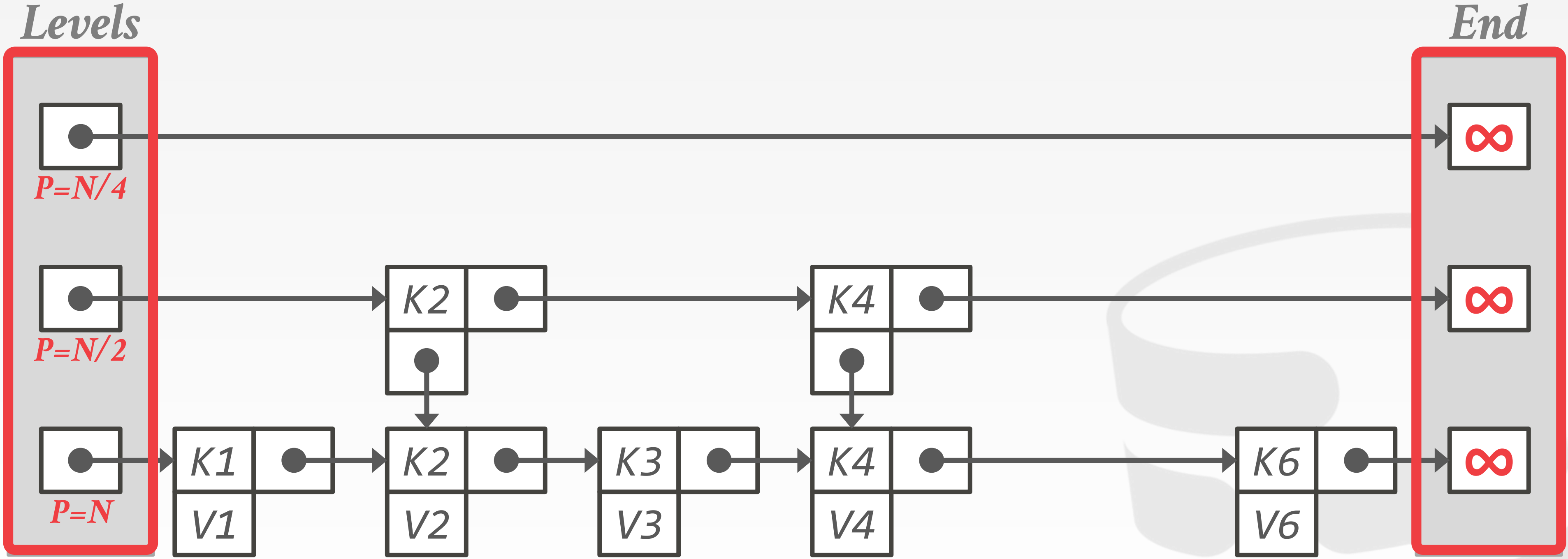
→ Lowest level is a sorted, singly linked list of all keys
→ 2nd level links every other key
→ 3rd level links every fourth key
→ In general, a level has half the keys of one below it

To insert a new key, flip a coin to decide how many levels to add the new key into.
Provides approximate O(log n) search times.

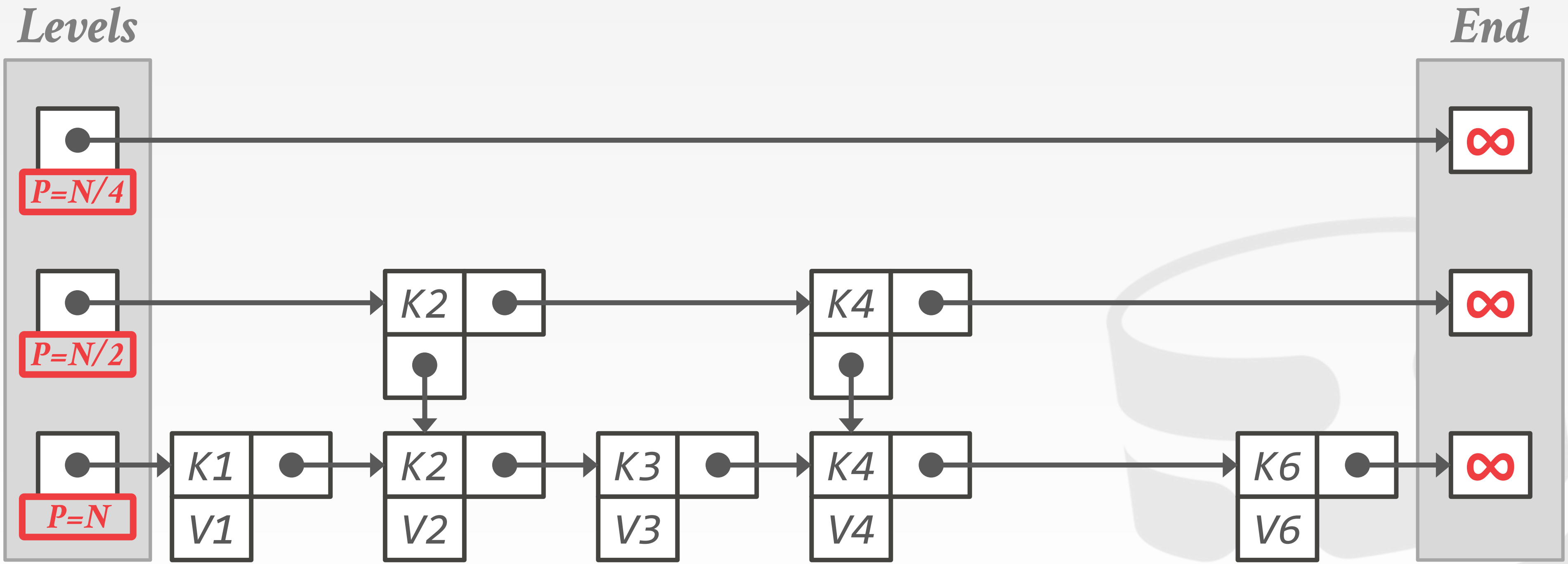
SKIP LISTS: EXAMPLE



SKIP LISTS: EXAMPLE

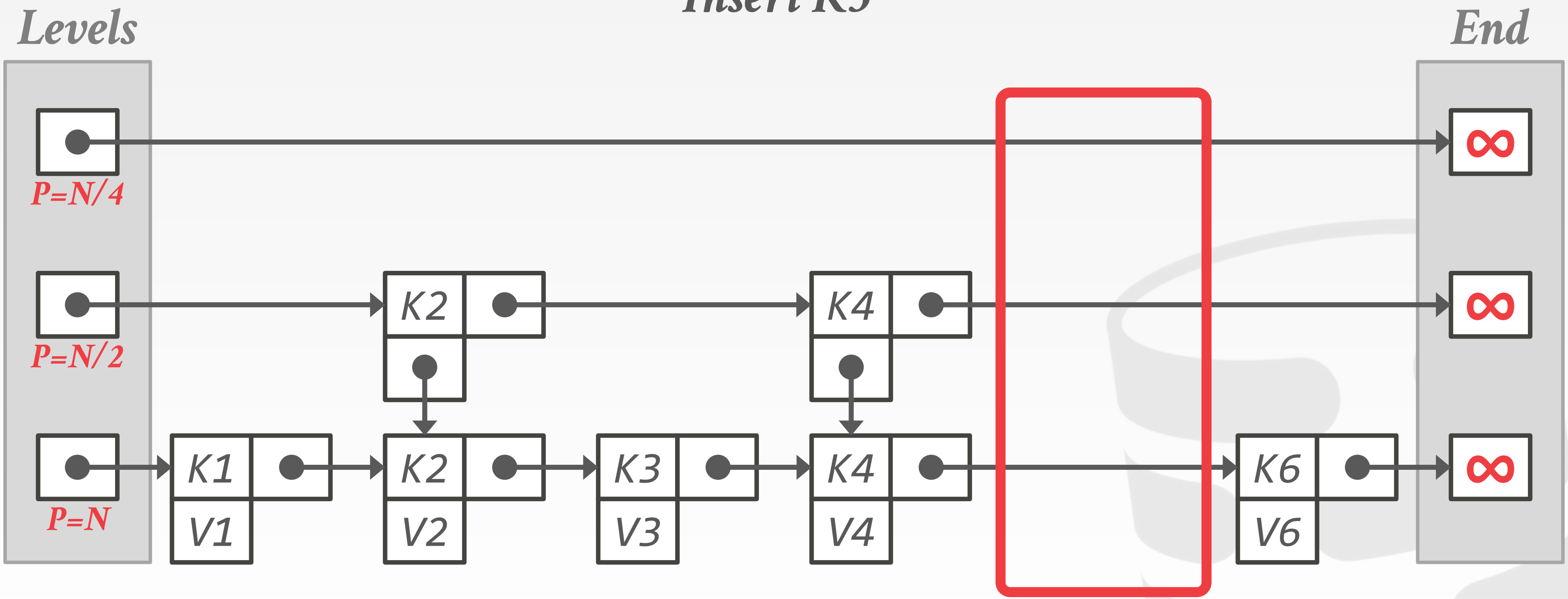


SKIP LISTS: EXAMPLE



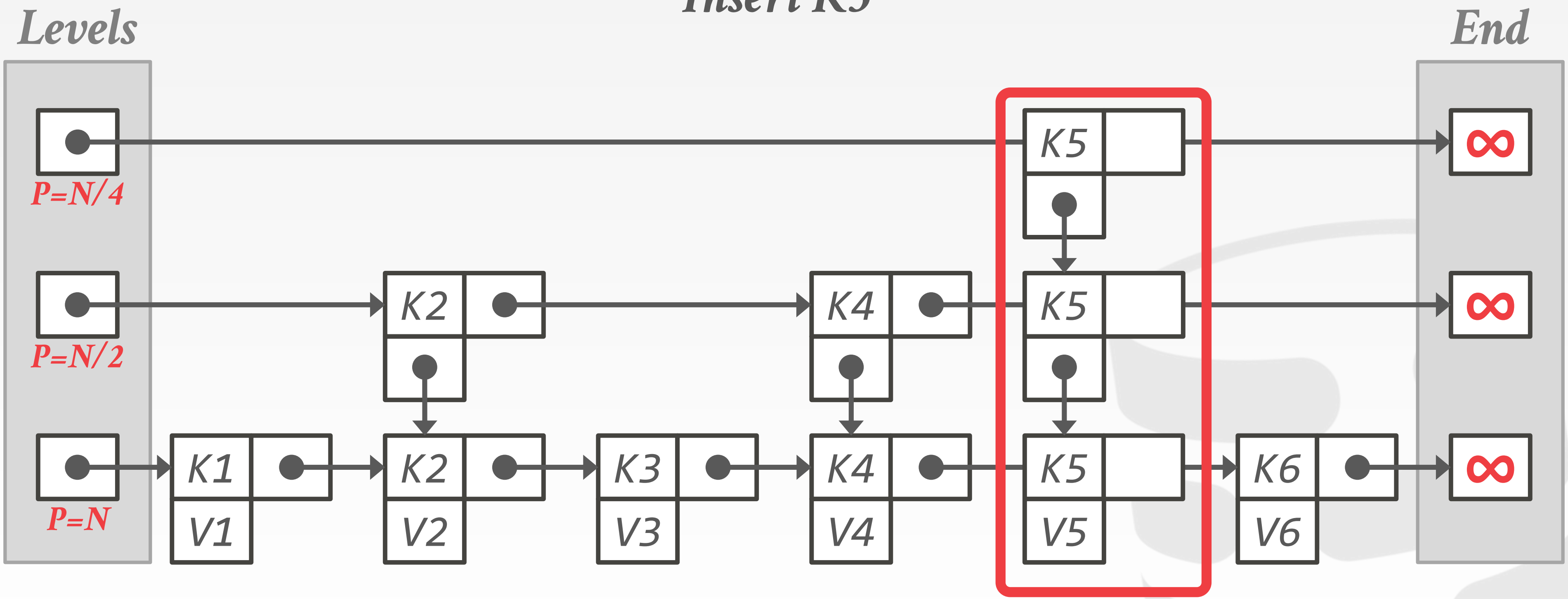
SKIP LISTS: INSERT

Insert K5



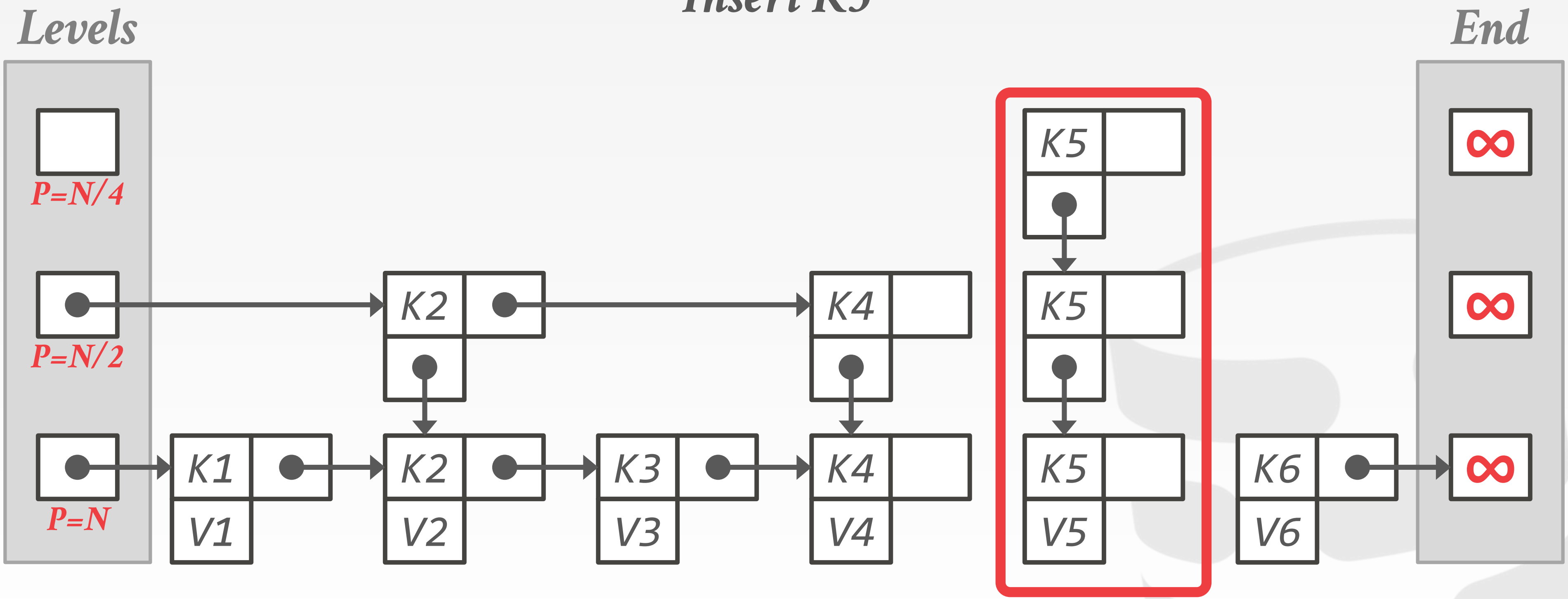
SKIP LISTS: INSERT

Insert K5



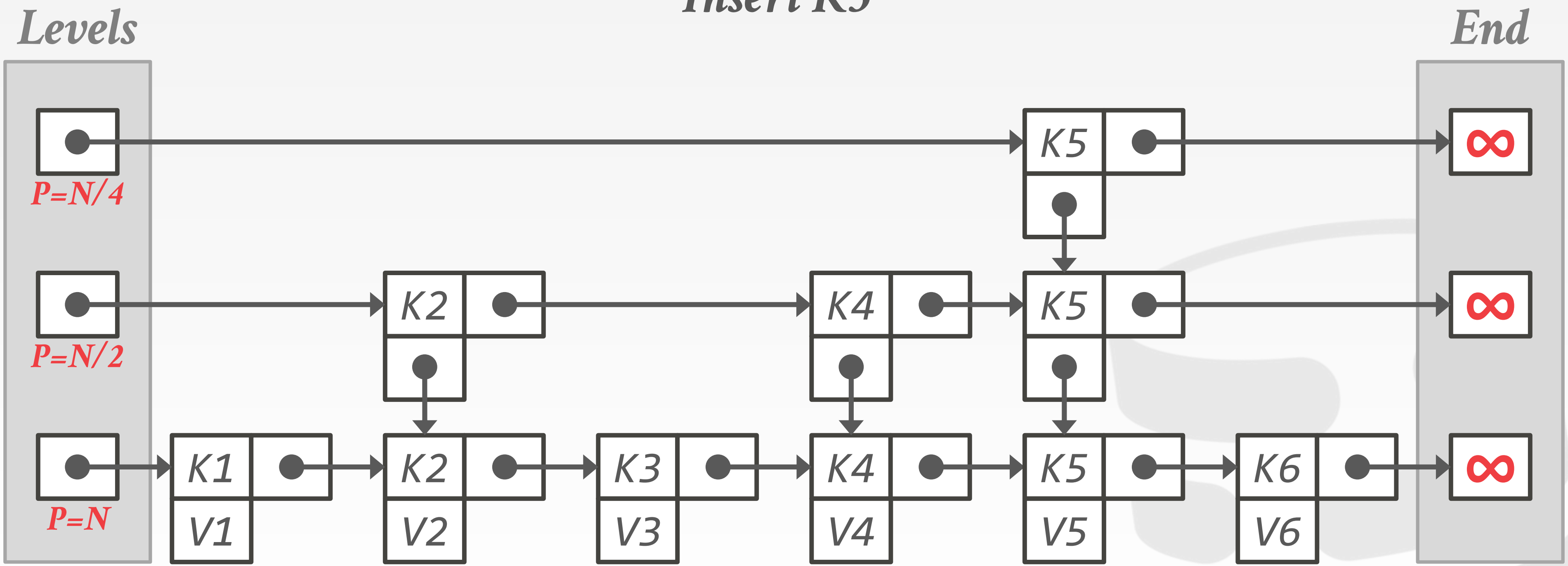
SKIP LISTS: INSERT

Insert K5



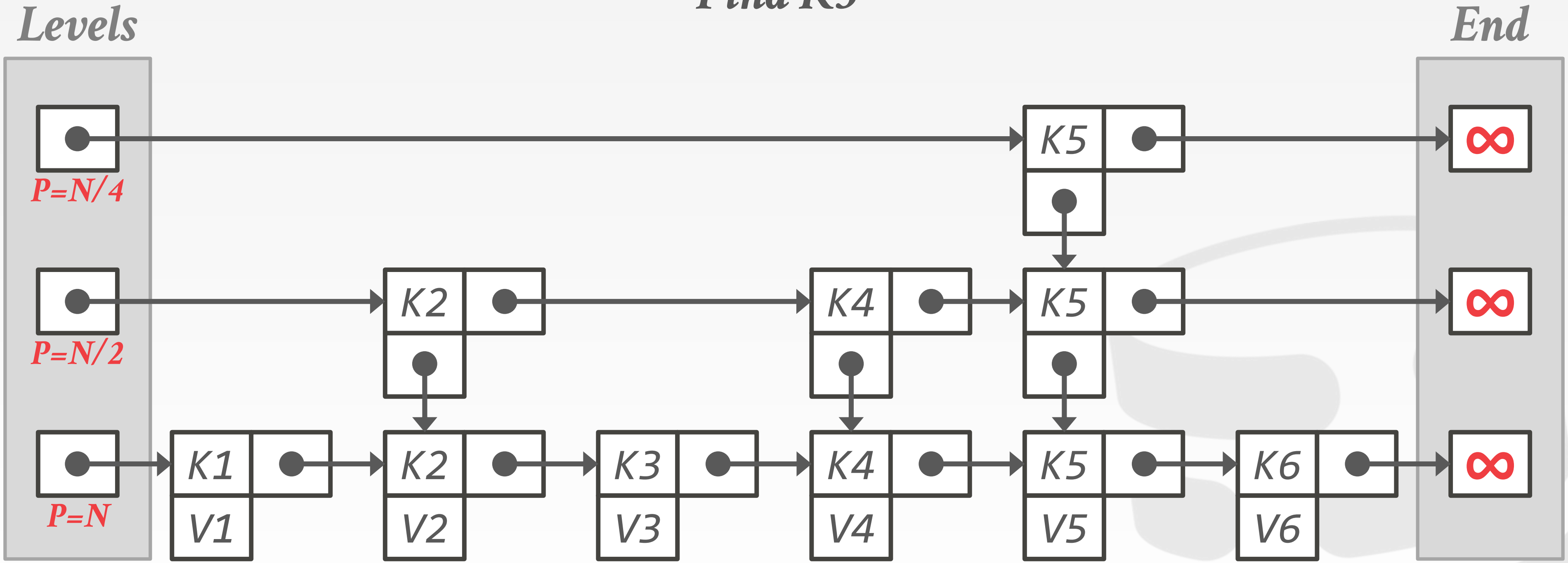
SKIP LISTS: INSERT

Insert K5



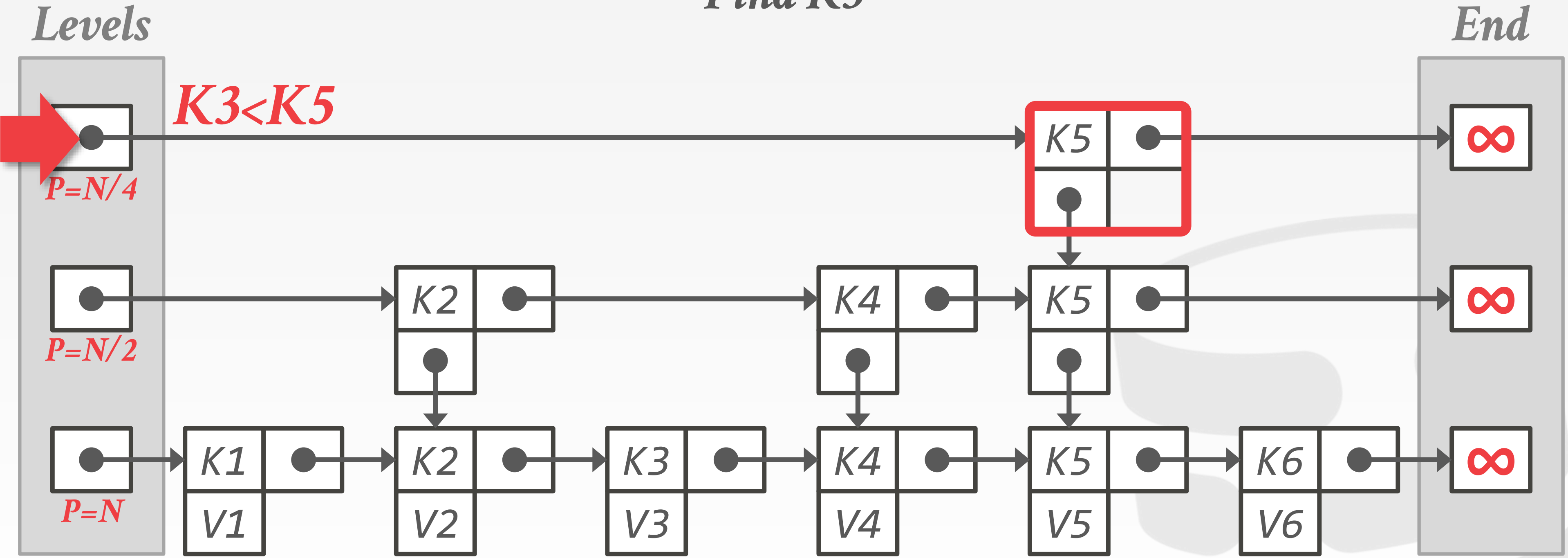
SKIP LISTS: SEARCH

Find K3



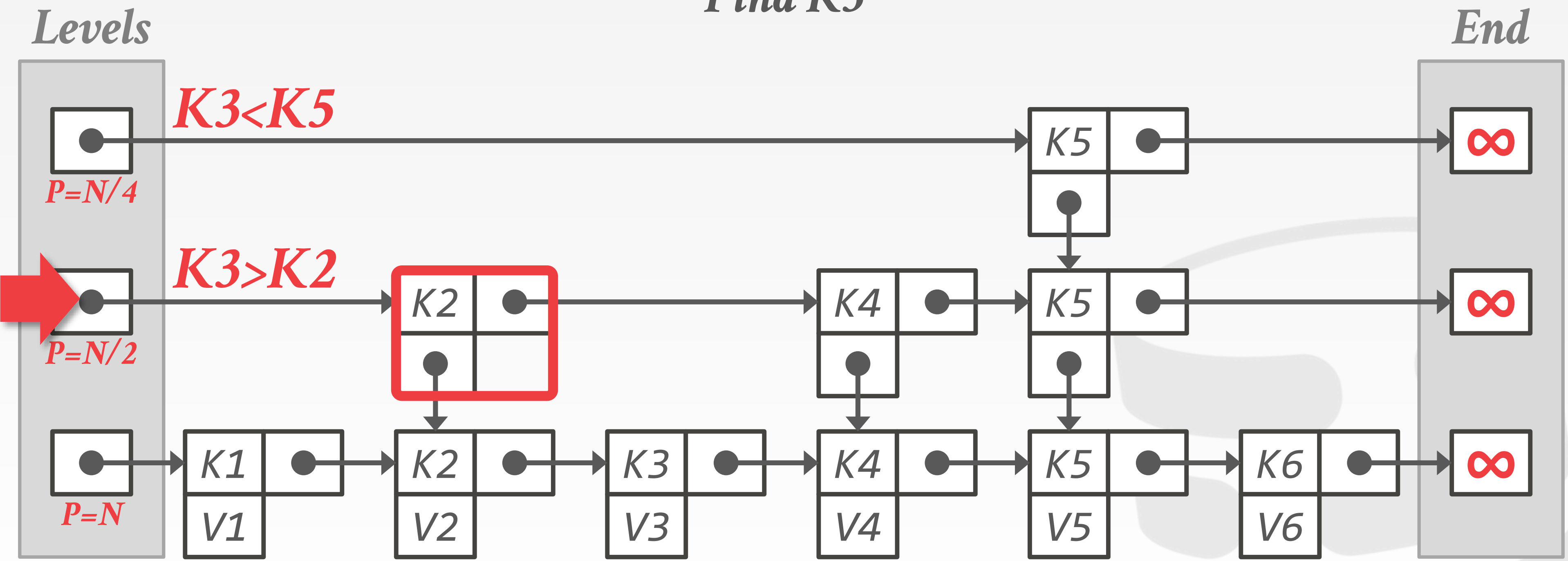
SKIP LISTS: SEARCH

Find K3



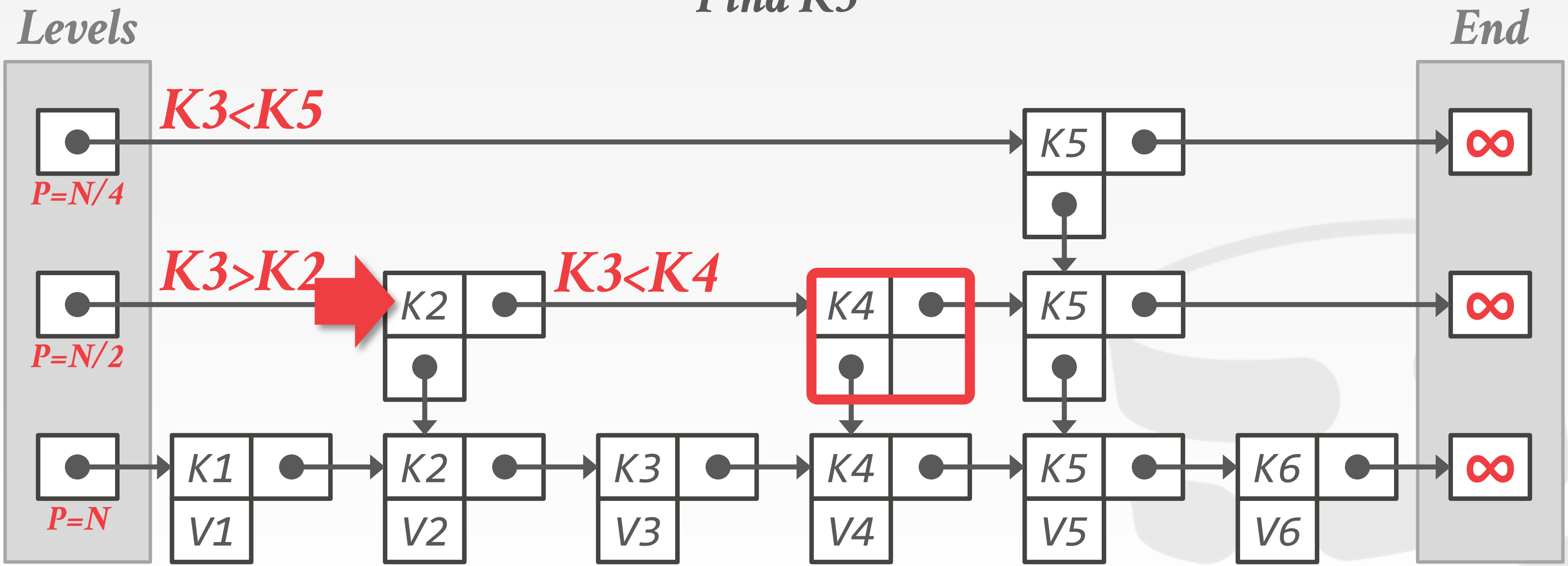
SKIP LISTS: SEARCH

Find K3



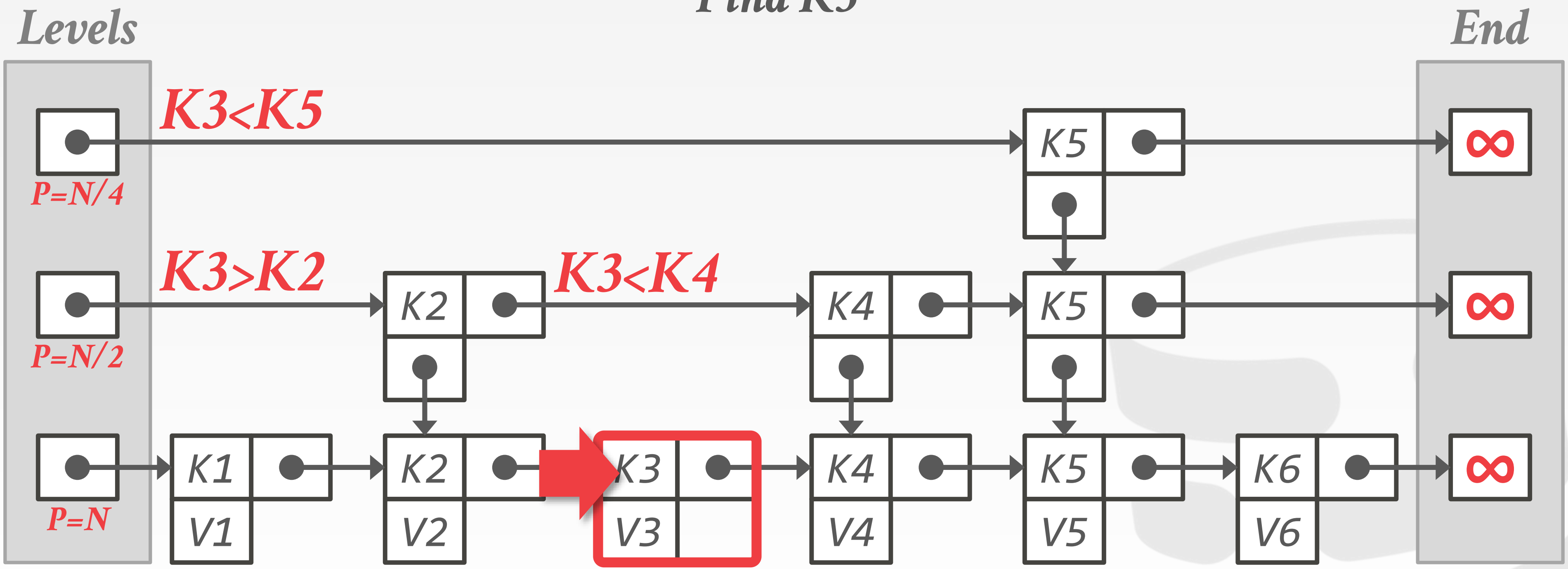
SKIP LISTS: SEARCH

Find K3



SKIP LISTS: SEARCH

Find K3



SKIP LISTS: ADVANTAGES

Uses less memory than a typical B+tree (only if you don’t include reverse pointers).

Insertions and deletions do not require rebalancing.

It is possible to implement a concurrent skip list using only CAS instructions.

CONCURRENT SKIP LIST

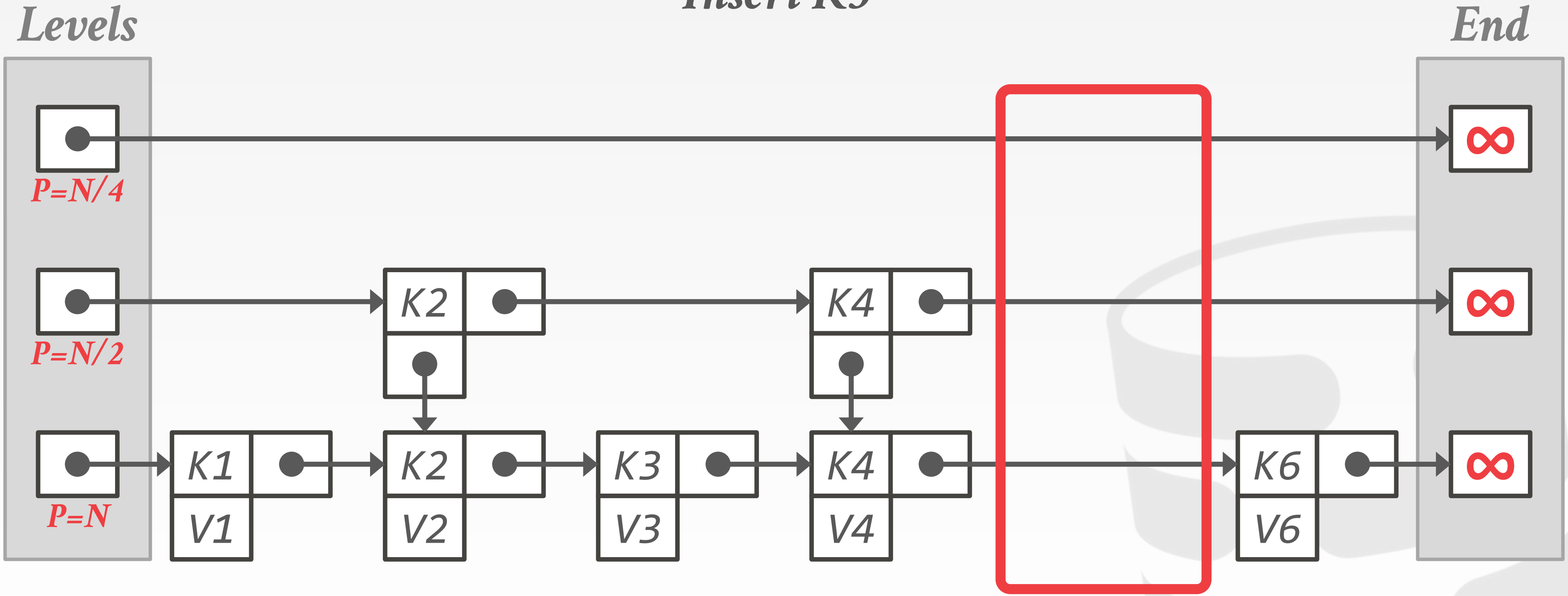
Can implement insert and delete without locks using only CaS operations.

The data structure only support links in one direction because CaS can only swap one pointer atomically.

CONCURRENT MAINTENANCE OF SKIP LISTS
Univ. of Maryland Tech Report 1990

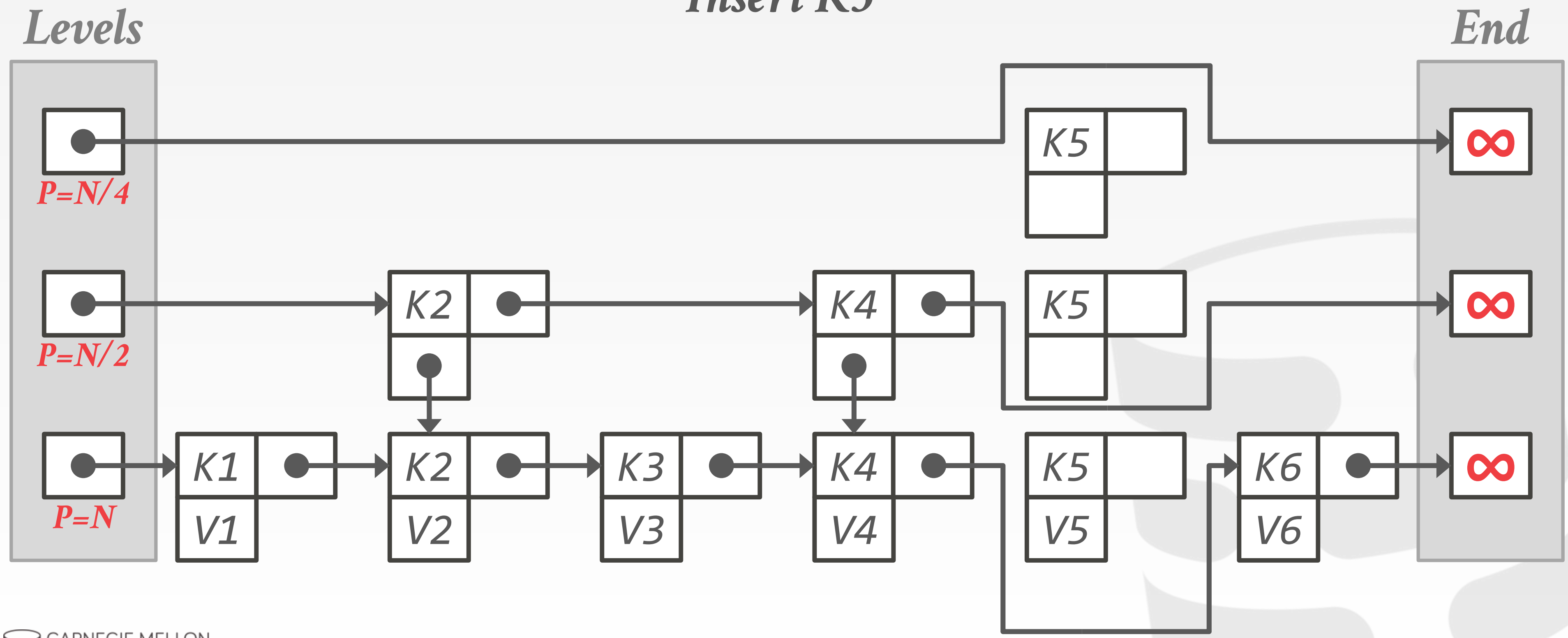
SKIP LISTS: INSERT

Insert K5



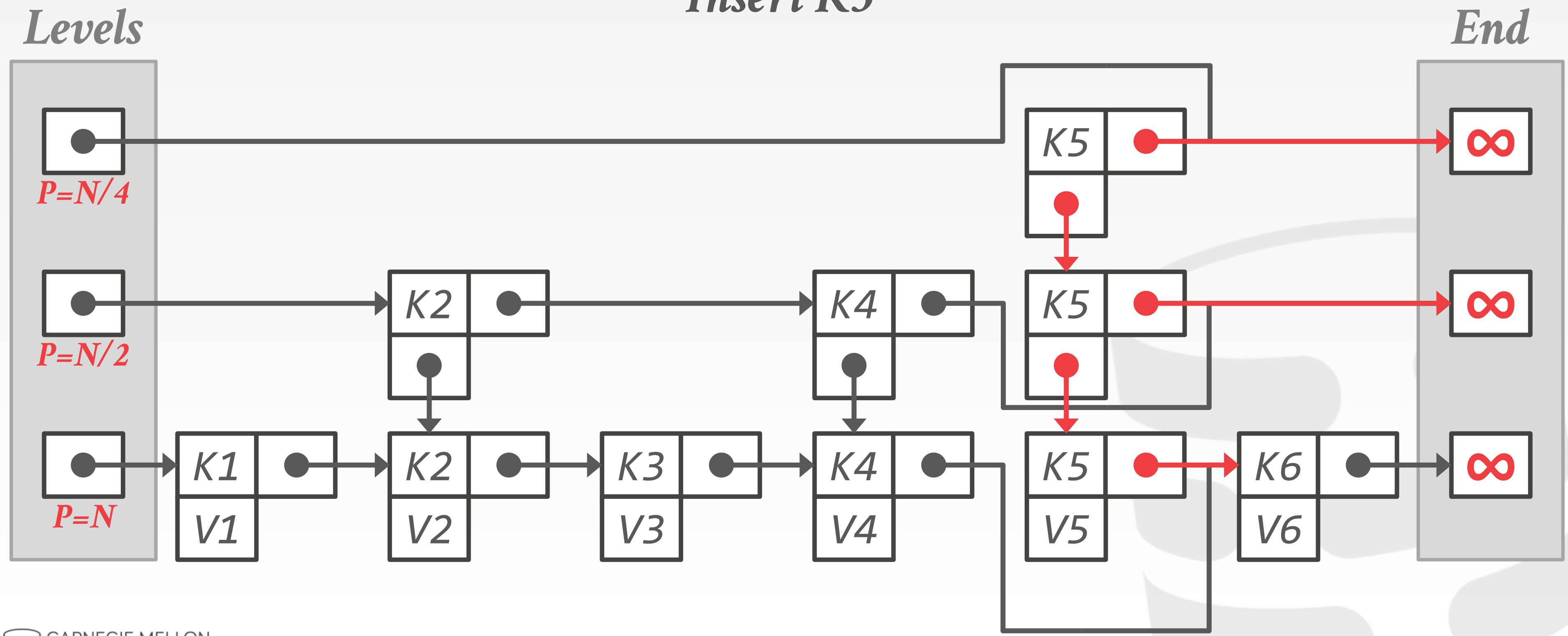
SKIP LISTS: INSERT

Insert K5



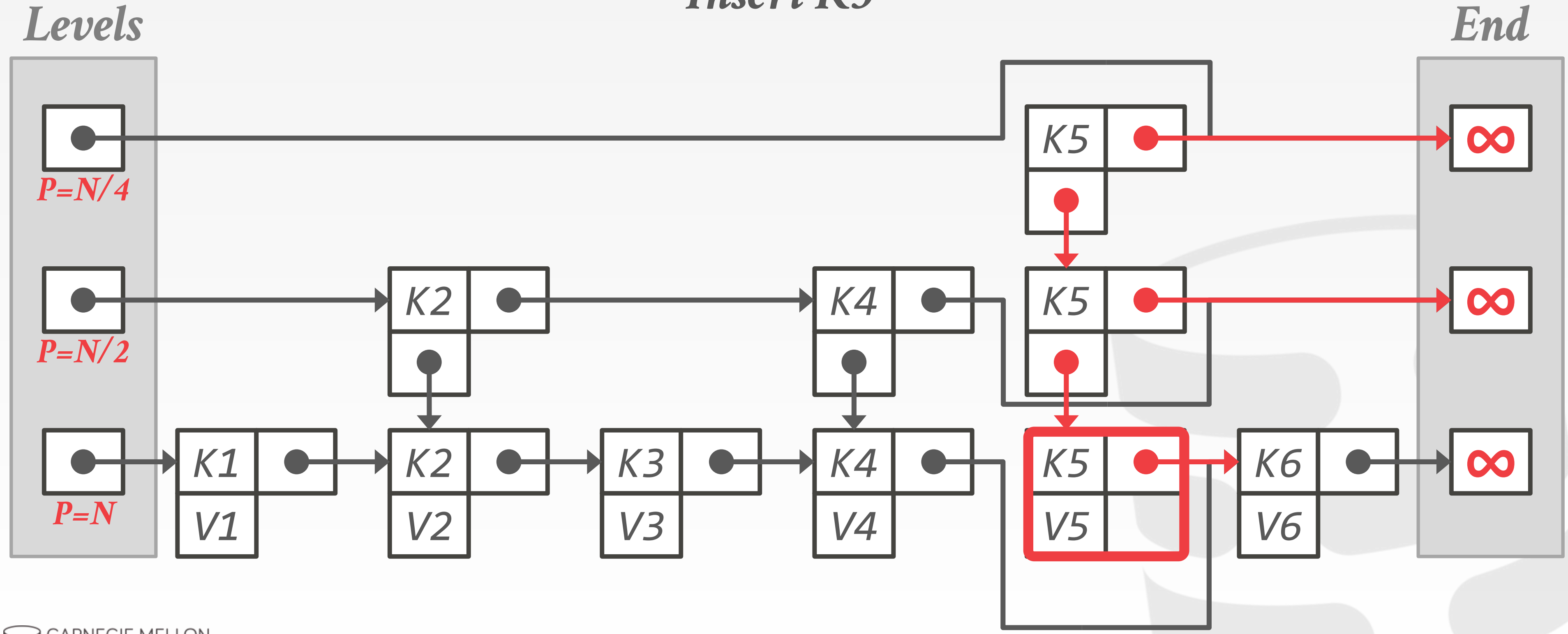
SKIP LISTS: INSERT

Insert K5



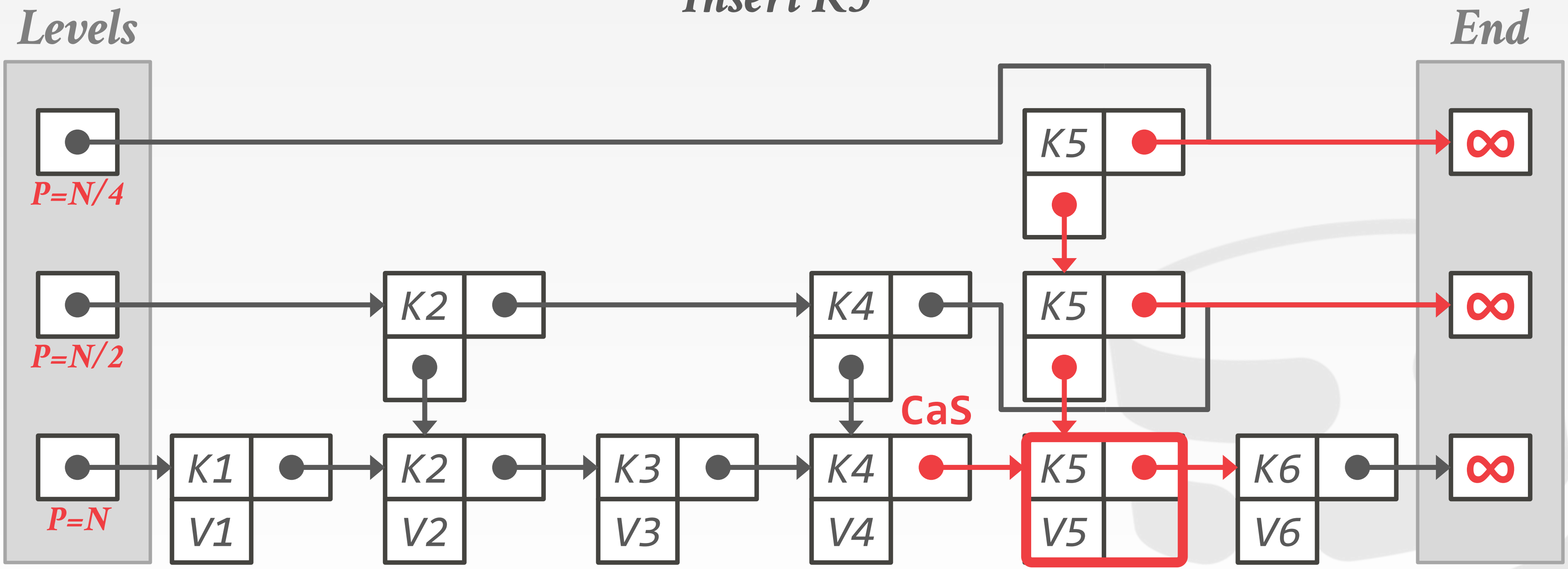
SKIP LISTS: INSERT

Insert K5



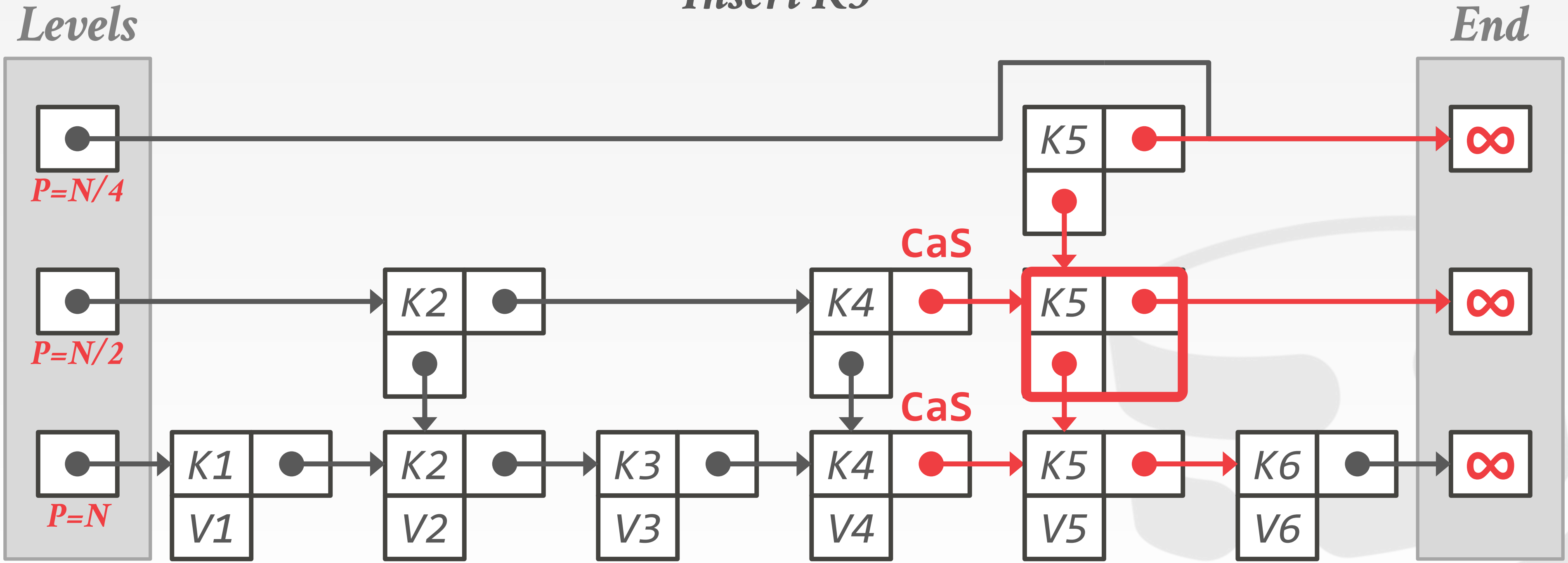
SKIP LISTS: INSERT

Insert K5

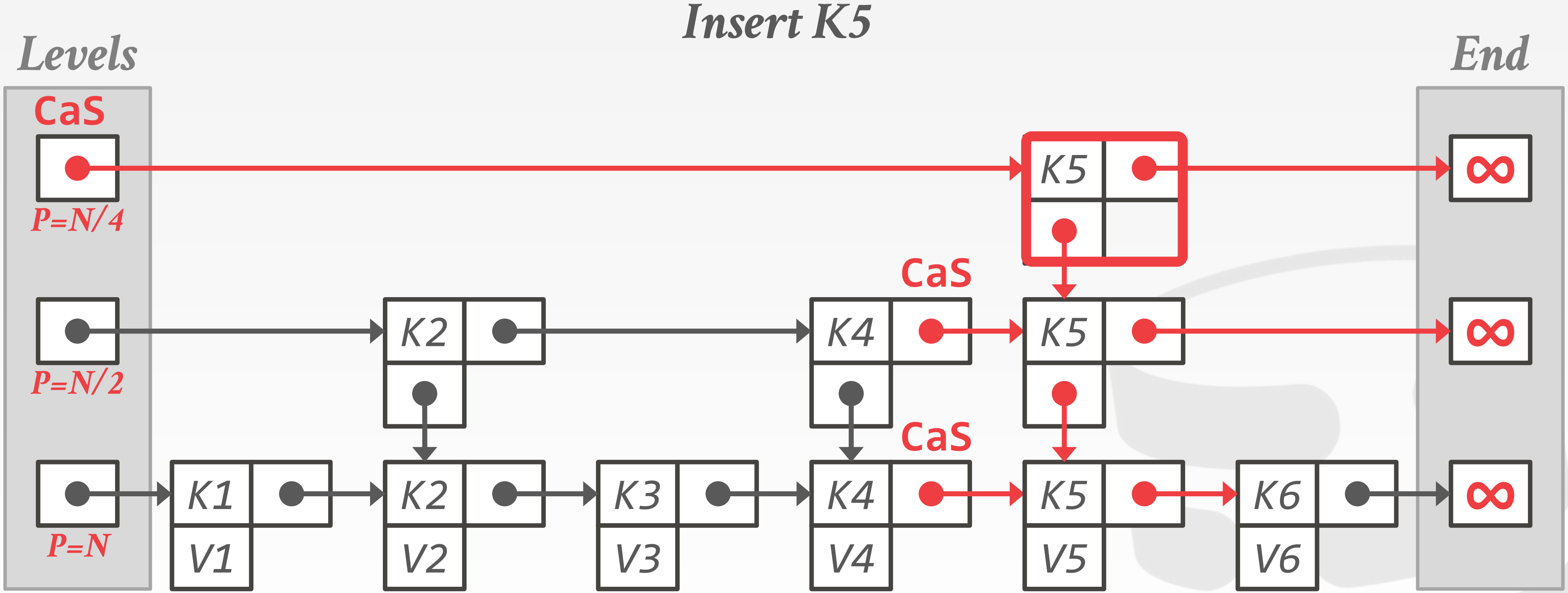


SKIP LISTS: INSERT

Insert K5



SKIP LISTS: INSERT



SKIP LISTS: DELETE

First **logically** remove a key from the index by setting a flag to tell threads to ignore.

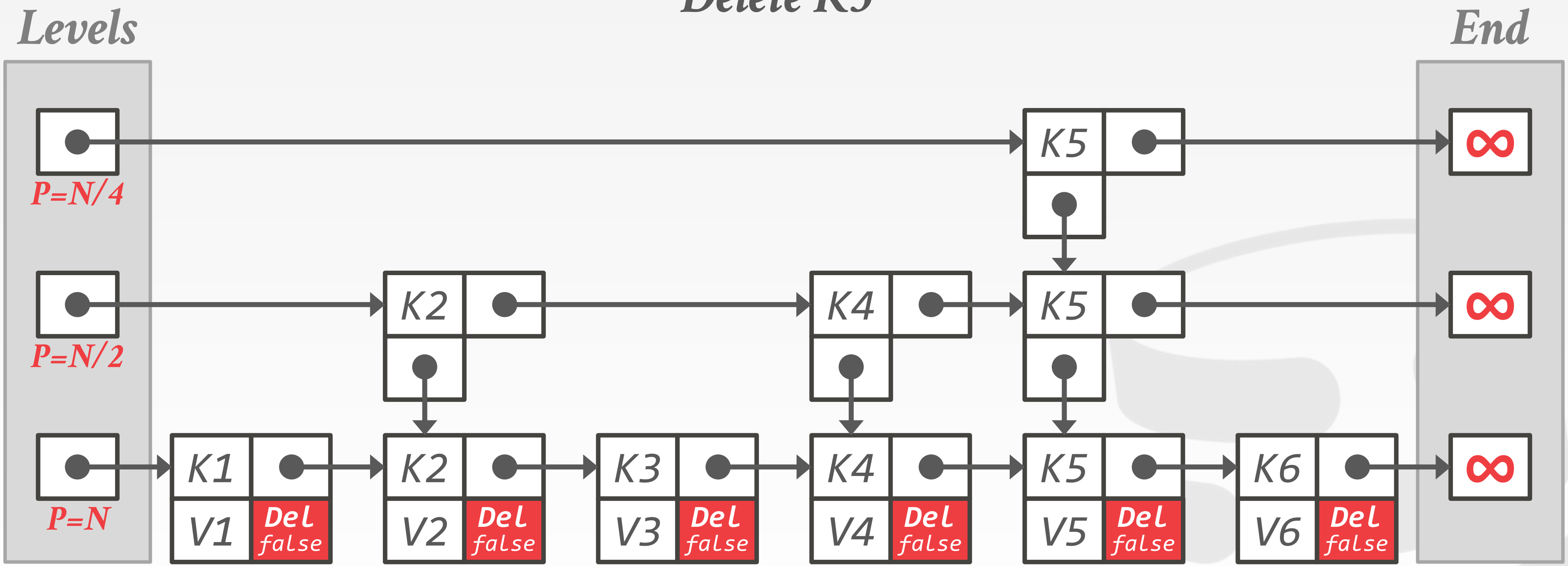
Then **physically** remove the key once we know that no other thread is holding the reference.

→ Perform CaS to update the predecessor's pointer.

Source: Stephen Tu

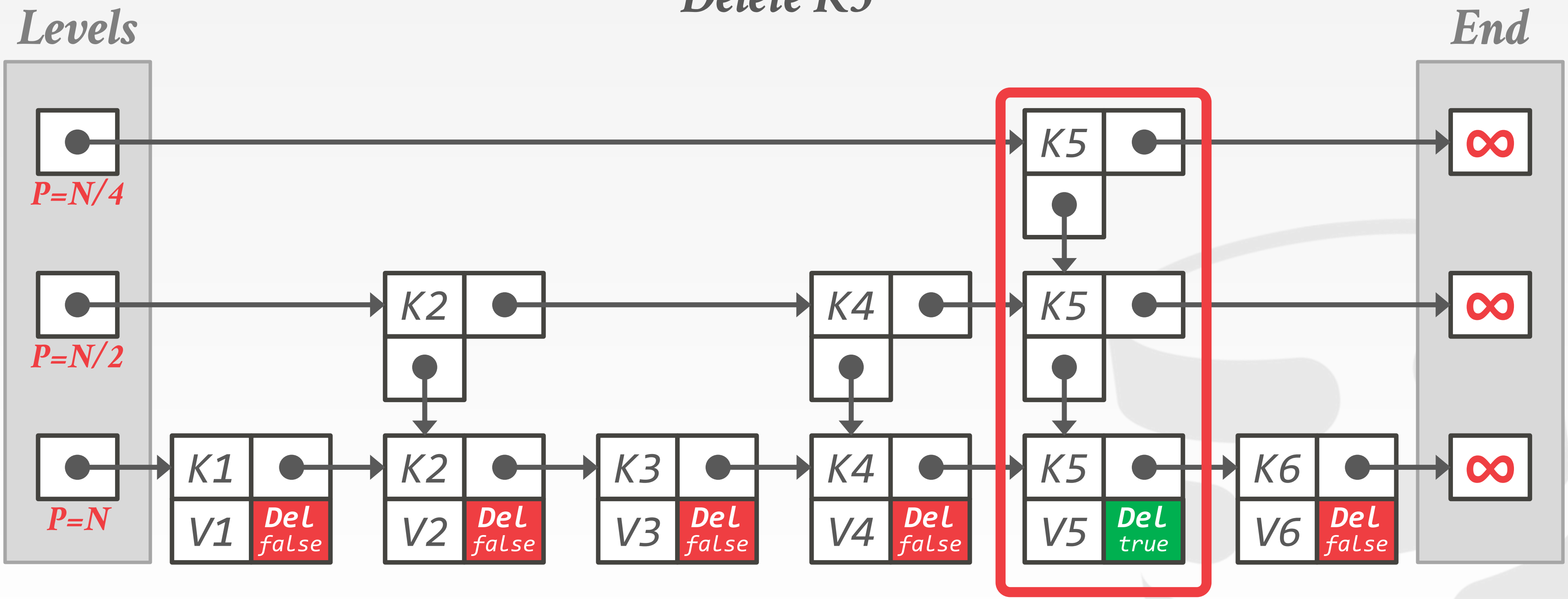
SKIP LISTS: DELETE

Delete K5



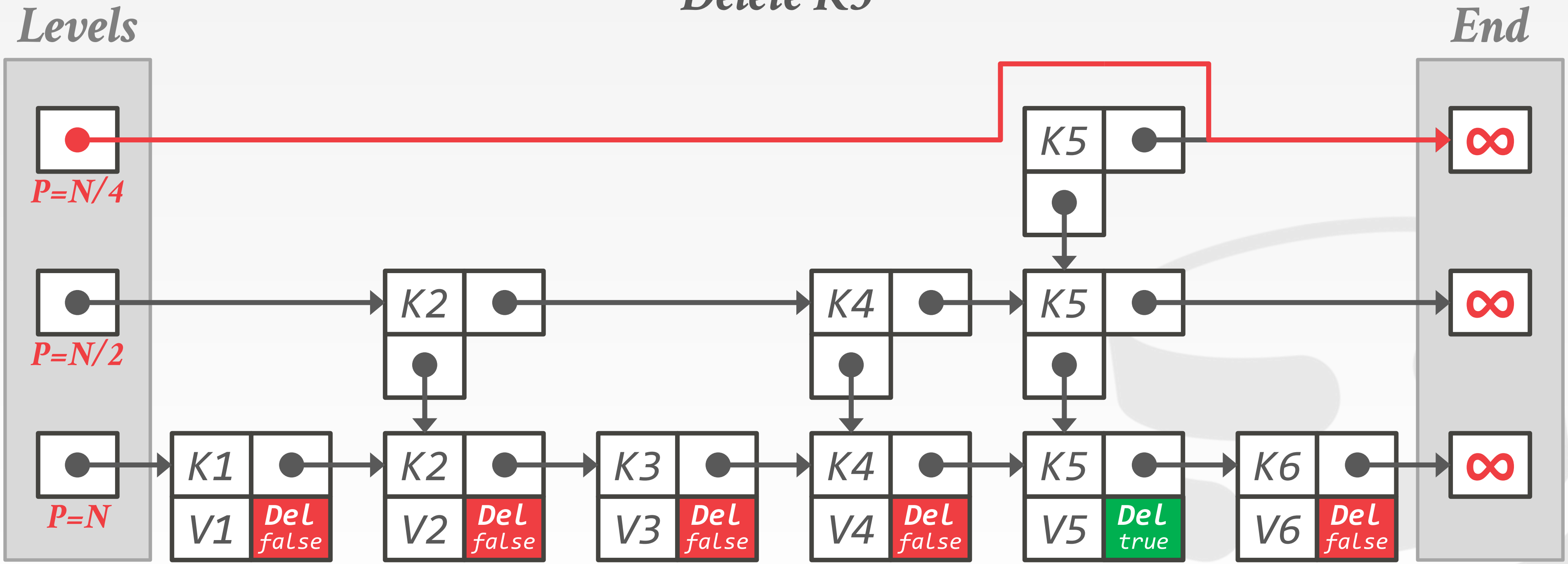
SKIP LISTS: DELETE

Delete K5

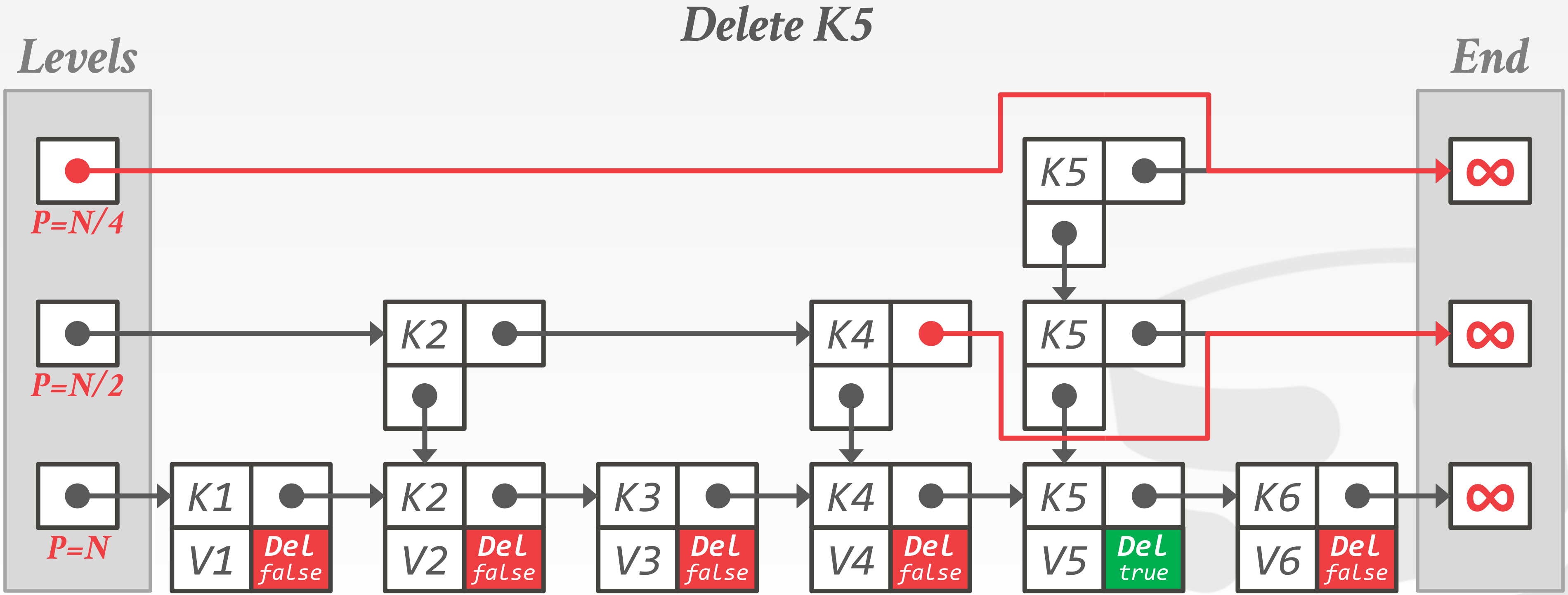


SKIP LISTS: DELETE

Delete K5

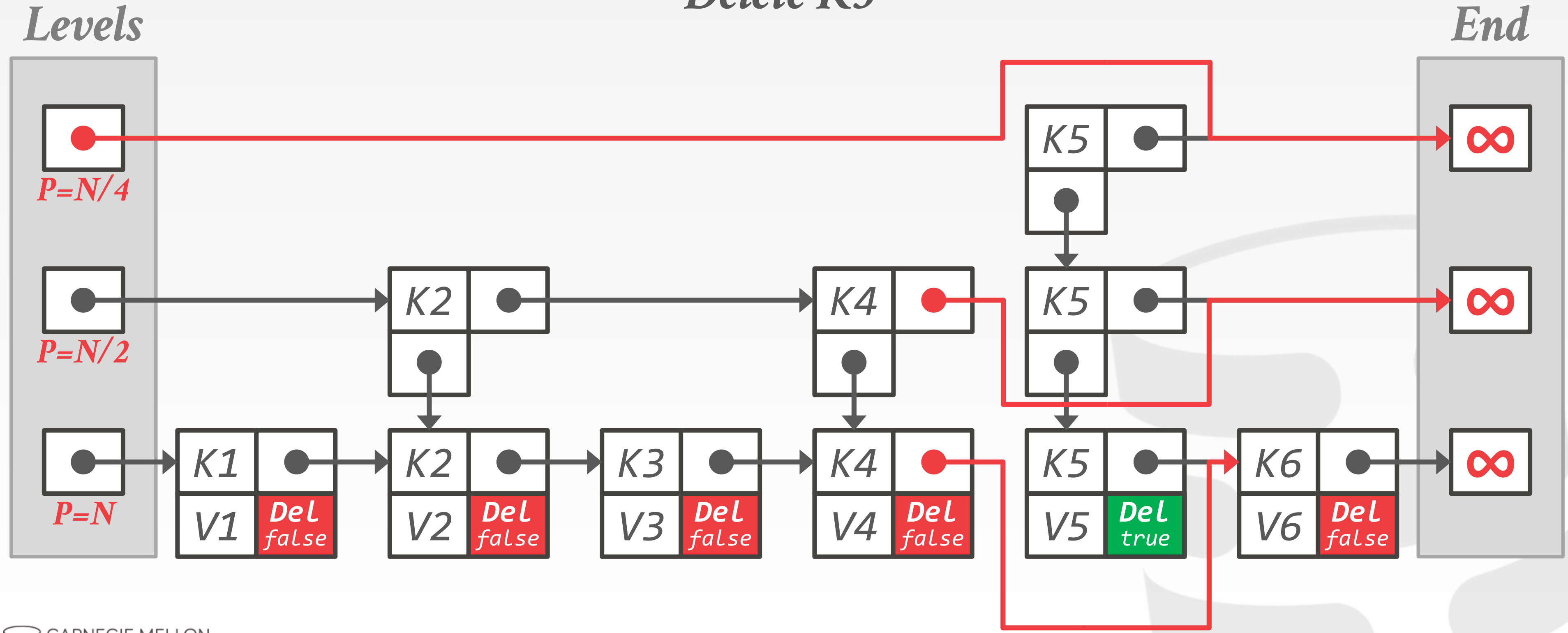


SKIP LISTS: DELETE



SKIP LISTS: DELETE

Delete K5



CONCURRENT SKIP LIST

Be careful about how you order operations.

If the DBMS invokes operation on the index, it can never “fail”

→ A txn can only abort due to higher-level conflicts.
→ If a CaS fails, then the index will retry until it succeeds.

SKIP LIST OPTIMIZATIONS

Reducing **RAND()** invocations.

Packing multiple keys in a node.

Reverse iteration with a stack.

Reusing nodes with memory pools.

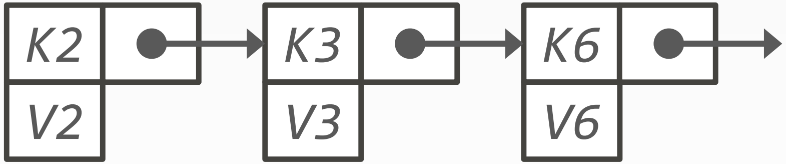
SKIP LISTS: DONE RIGHT
Ticki(?) Blog 2016

SKIP LIST: COMBINE NODES

Store multiple keys in a single node.

→ **Insert Key:** Find the node where it should go and look for a free slot. Perform CaS to store new key. If no slot is available, insert new node.

→ **Search Key:** Perform linear search on keys in each node.



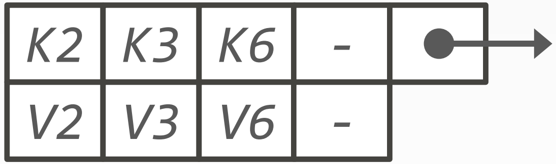
Source: Ticki

SKIP LIST: COMBINE NODES

Store multiple keys in a single node.

→ **Insert Key:** Find the node where it should go and look for a free slot. Perform CaS to store new key. If no slot is available, insert new node.

→ **Search Key:** Perform linear search on keys in each node.



K2	K3	K6	-	●
V2	V3	V6	-	

Source: Ticki

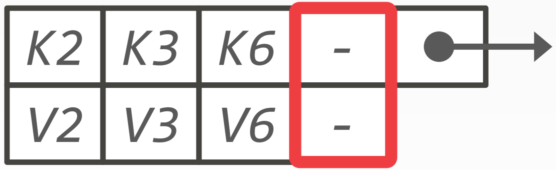
SKIP LIST: COMBINE NODES

Store multiple keys in a single node.

→ **Insert Key:** Find the node where it should go and look for a free slot. Perform CaS to store new key. If no slot is available, insert new node.

→ **Search Key:** Perform linear search on keys in each node.

Insert K4



Source: Ticki

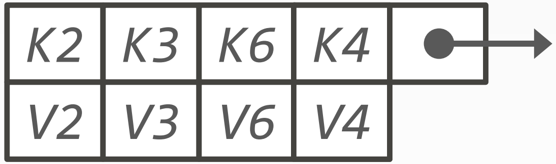
SKIP LIST: COMBINE NODES

Store multiple keys in a single node.

→ **Insert Key:** Find the node where it should go and look for a free slot. Perform CaS to store new key. If no slot is available, insert new node.

→ **Search Key:** Perform linear search on keys in each node.

Insert K4



K2	K3	K6	K4	● →
V2	V3	V6	V4	

Source: Ticki

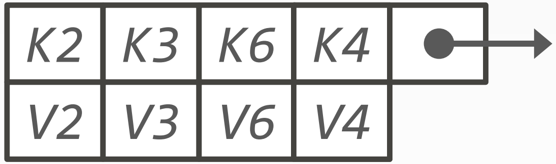
SKIP LIST: COMBINE NODES

Store multiple keys in a single node.

→ **Insert Key:** Find the node where it should go and look for a free slot. Perform CaS to store new key. If no slot is available, insert new node.

→ **Search Key:** Perform linear search on keys in each node.

Search K6

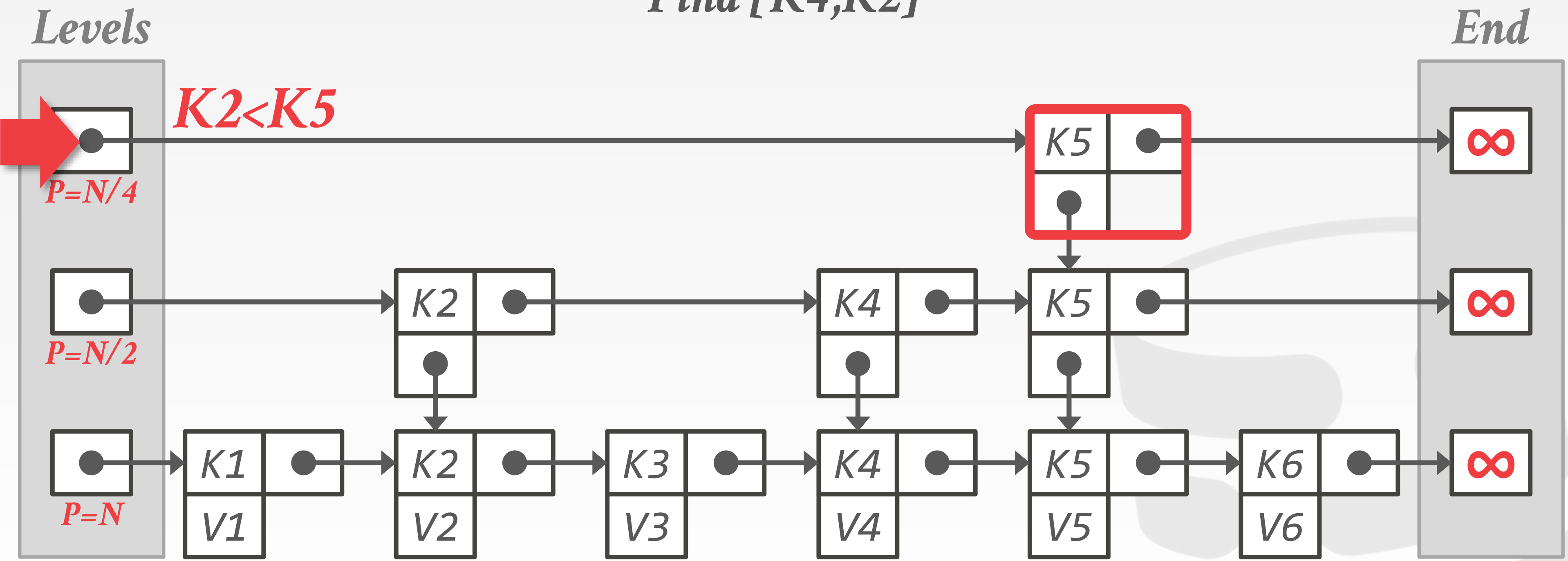


K2	K3	K6	K4	● →
V2	V3	V6	V4	

Source: Ticki

SKIP LISTS: REVERSE SEARCH

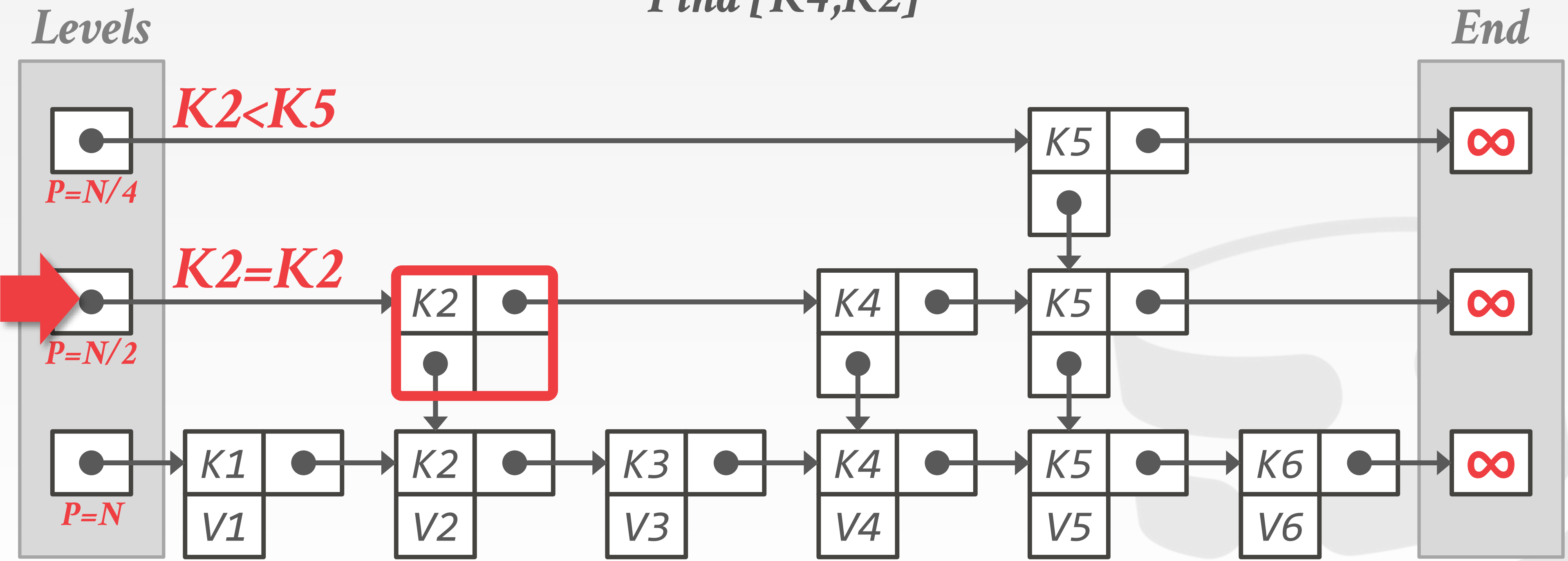
Find [K4,K2]



Source: Mark Papadakis

SKIP LISTS: REVERSE SEARCH

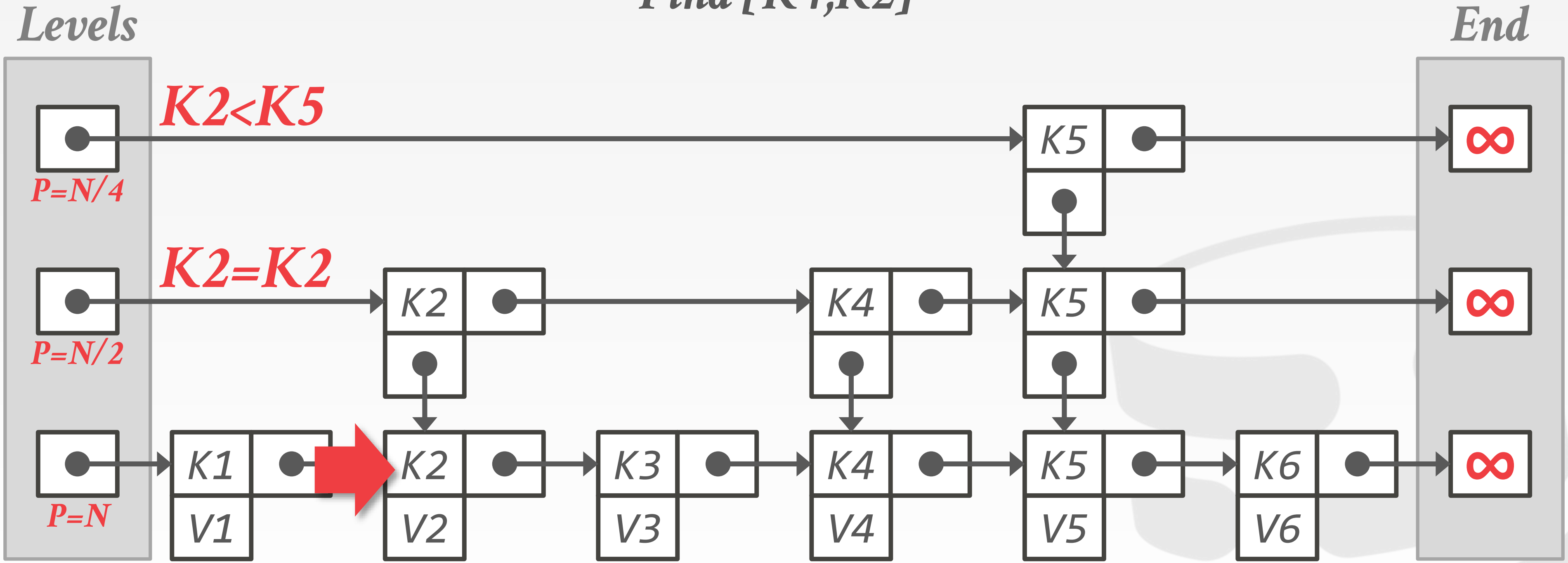
Find [K4,K2]



Source: Mark Papadakis

SKIP LISTS: REVERSE SEARCH

Find [K4,K2]

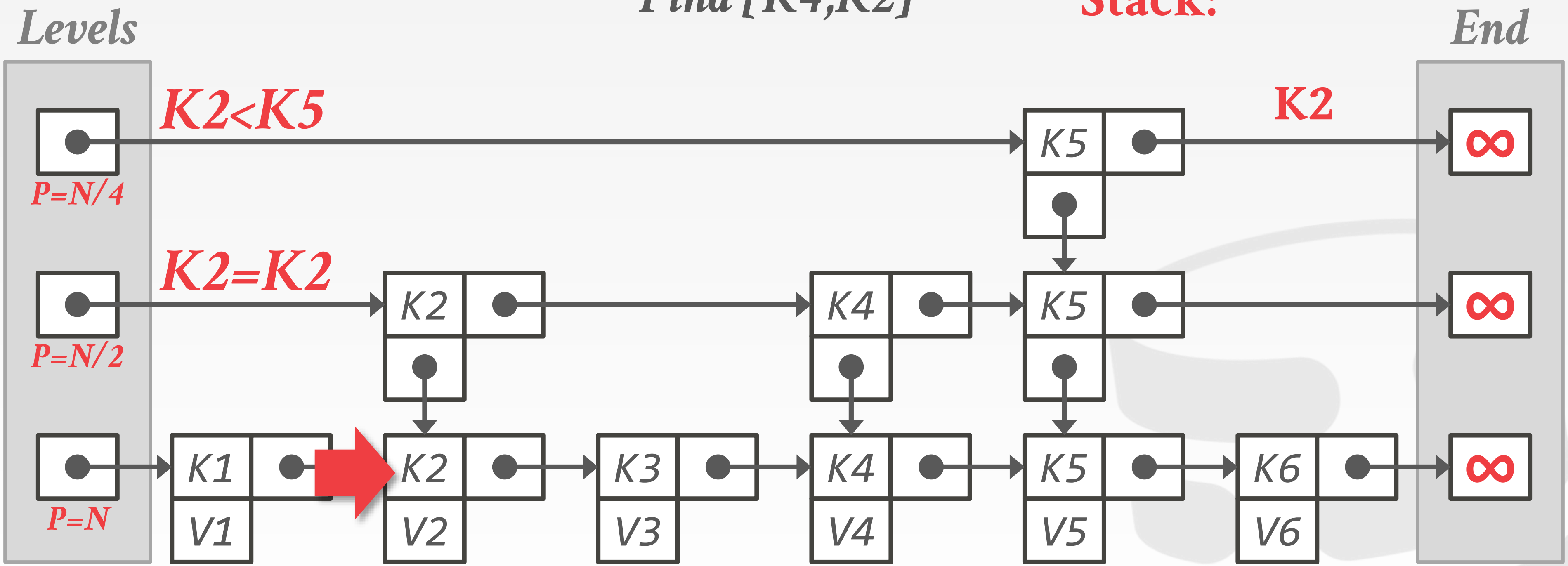


Source: Mark Papadakis

SKIP LISTS: REVERSE SEARCH

Find [K4,K2]

Stack:



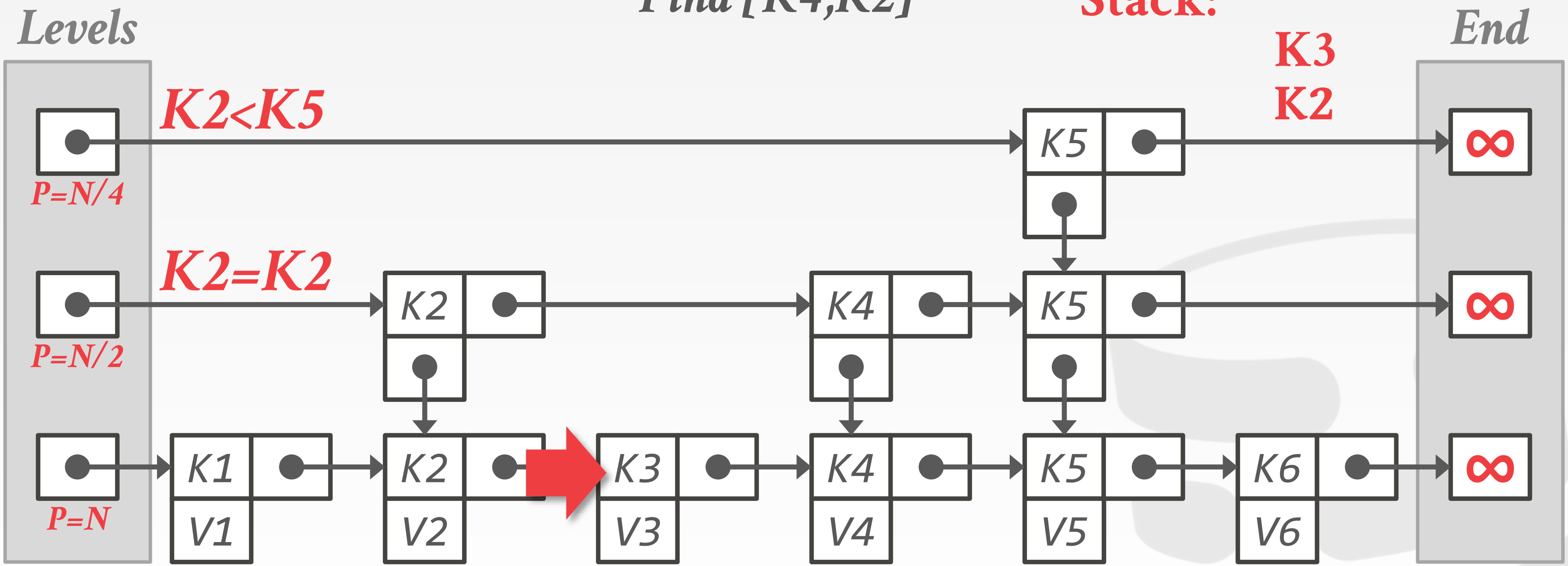
Source: Mark Papadakis

SKIP LISTS: REVERSE SEARCH

Find [K4,K2]

Stack:

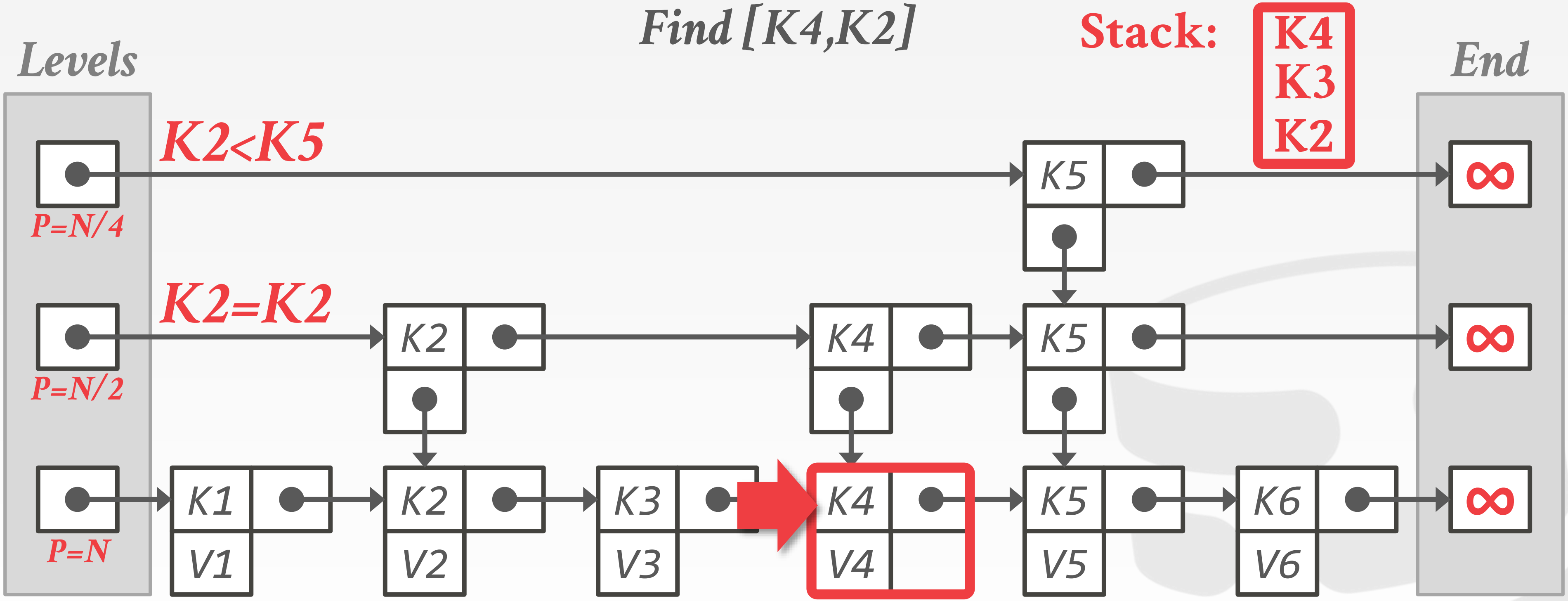
K3
K2



Source: Mark Papadakis

SKIP LISTS: REVERSE SEARCH

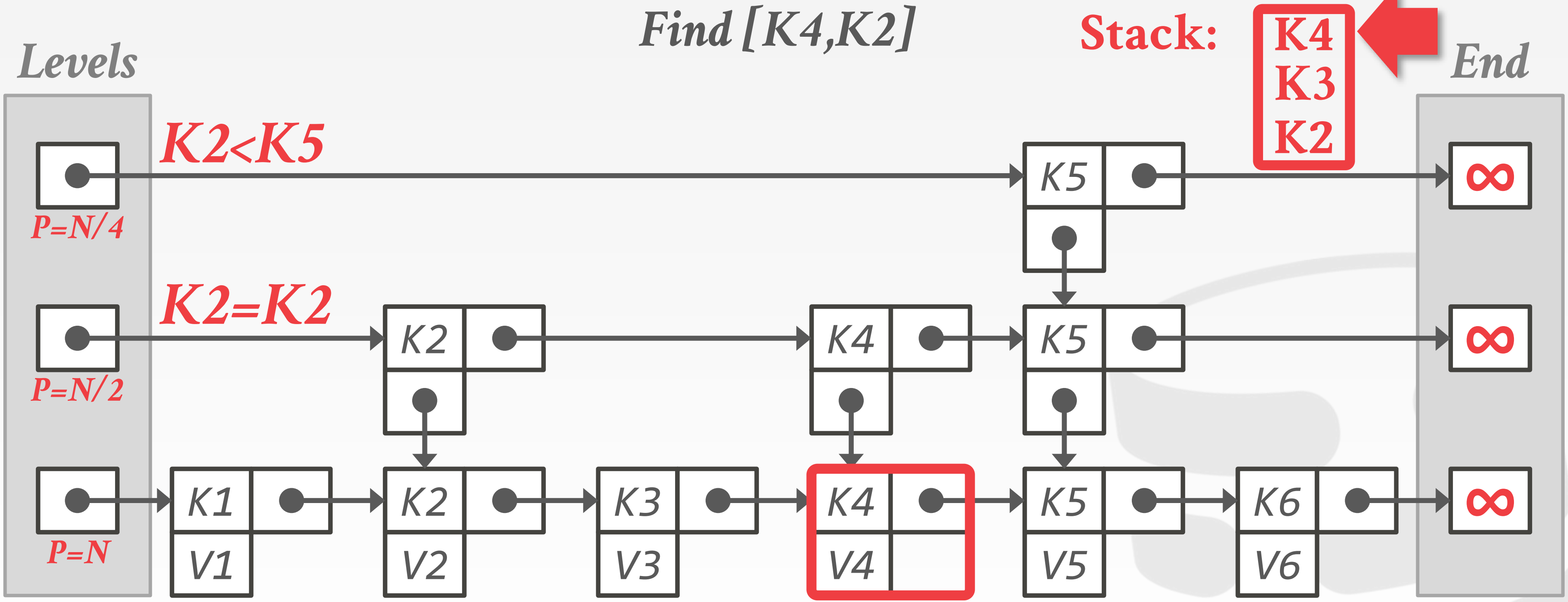
Find [K4,K2]



Source: Mark Papadakis

SKIP LISTS: REVERSE SEARCH

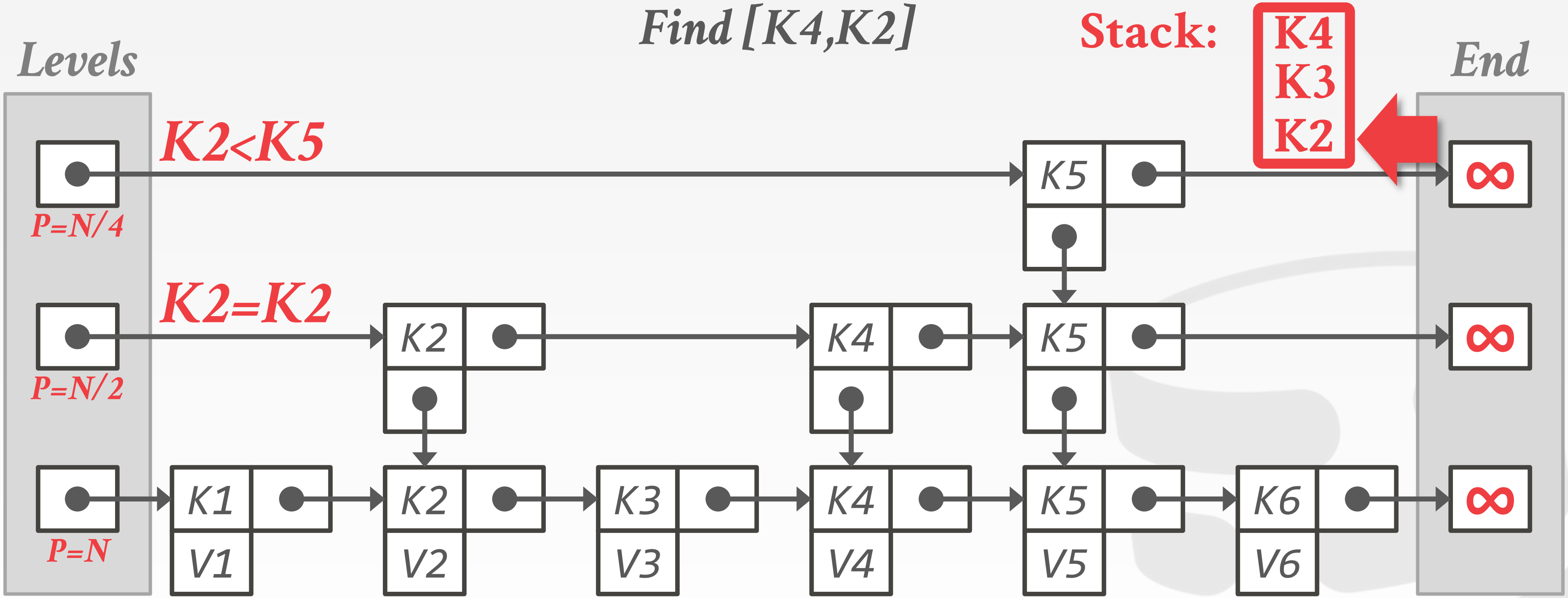
Find [K4,K2]



Source: Mark Papadakis

SKIP LISTS: REVERSE SEARCH

Find [K4,K2]



Source: Mark Papadakis

OBSERVATION

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

→ We cannot have reverse pointers in a latch-free concurrent Skip List.

→ We cannot build a latch-free B+Tree.

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

BW-TREE

Latch-free B+Tree index

→ Threads never need to set latches or block.

Key Idea #1: Deltas

→ No updates in place

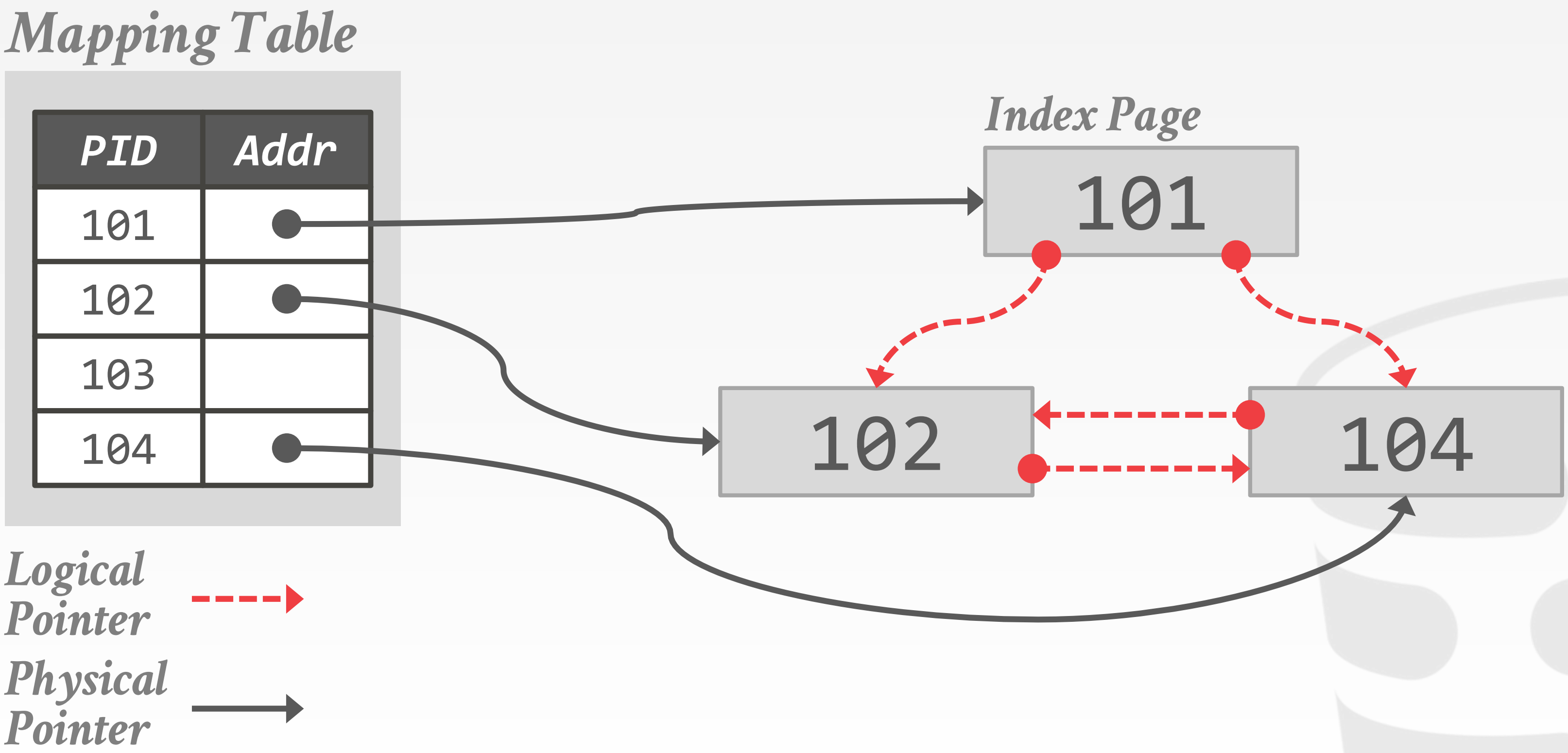
→ Reduces cache invalidation.

Key Idea #2: Mapping Table

→ Allows for CaS of physical locations of pages.

THE BW-TREE: A B-TREE FOR NEW HARDWARE
ICDE 2013

BW-TREE: MAPPING TABLE



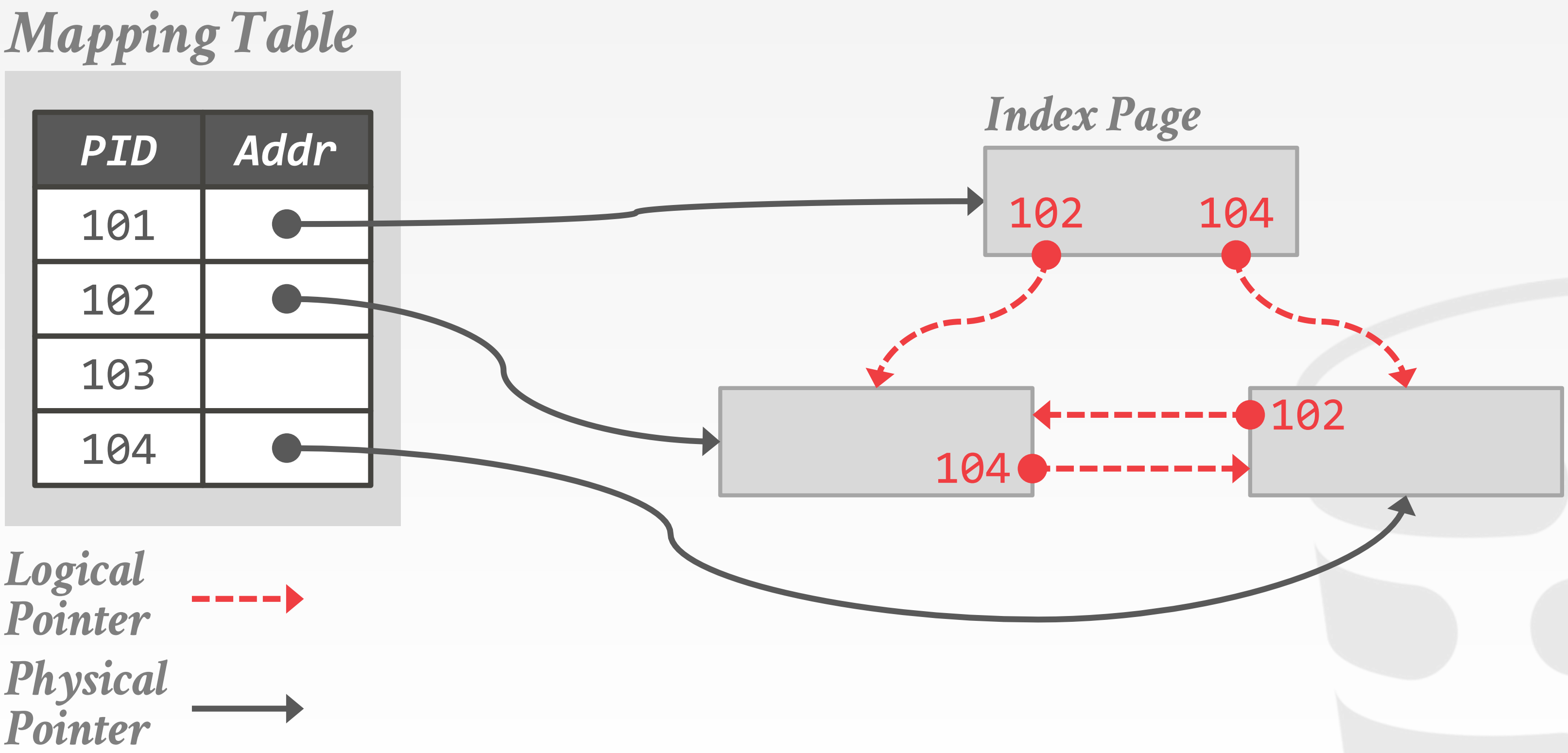
Mapping Table

PID	Addr
101	●
102	●
103	
104	●

Logical Pointer

Physical Pointer

BW-TREE: MAPPING TABLE



Mapping Table

PID	*Addr*
101	●
102	●
103	
104	●

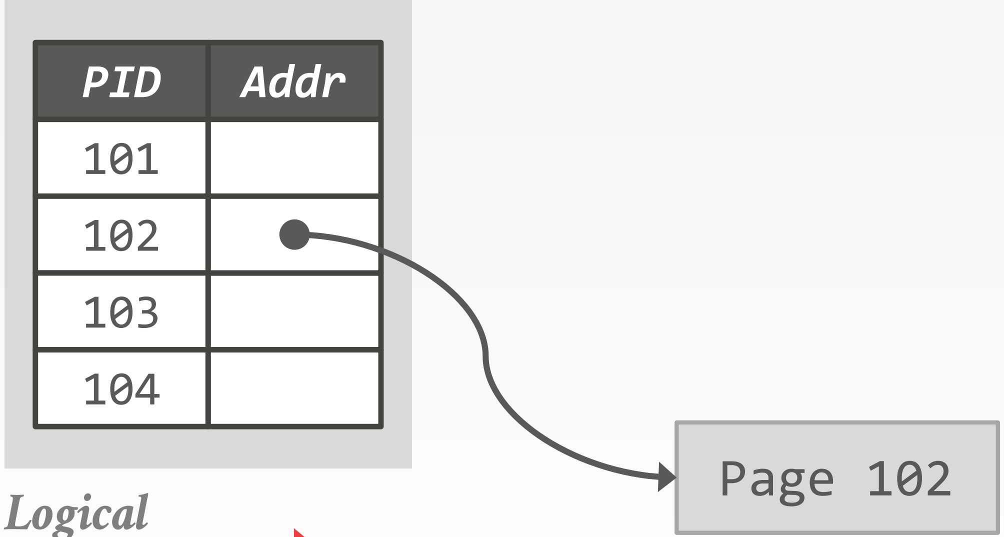
Index Page

Logical Pointer

Physical Pointer

BW-TREE: DELTA UPDATES

Mapping Table



PID	Addr
101	
102	●
103	
104	

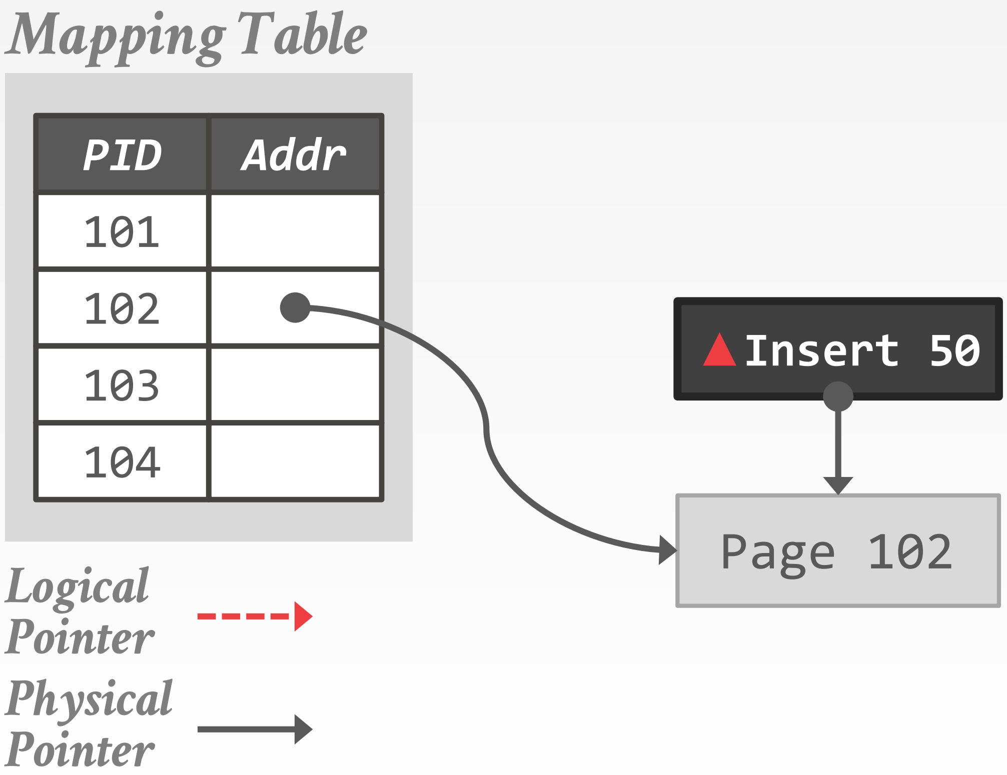
Each update to a page produces a new delta.

Logical Pointer - - - -▶

Physical Pointer ——▶

Source: Justin Levandoski

BW-TREE: DELTA UPDATES



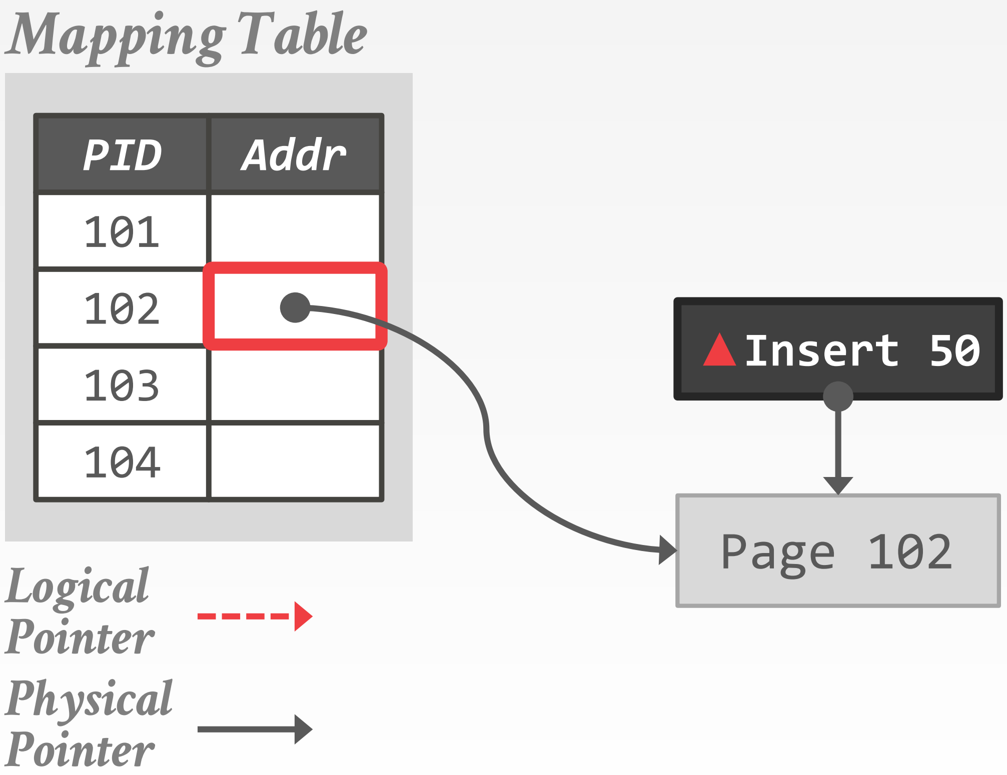
PID	Addr
101	
102	
103	
104	

Each update to a page produces a new delta.

Delta physically points to base page.

Source: Justin Levandoski

BW-TREE: DELTA UPDATES



Mapping Table

PID	Addr
101	
102	●
103	
104	

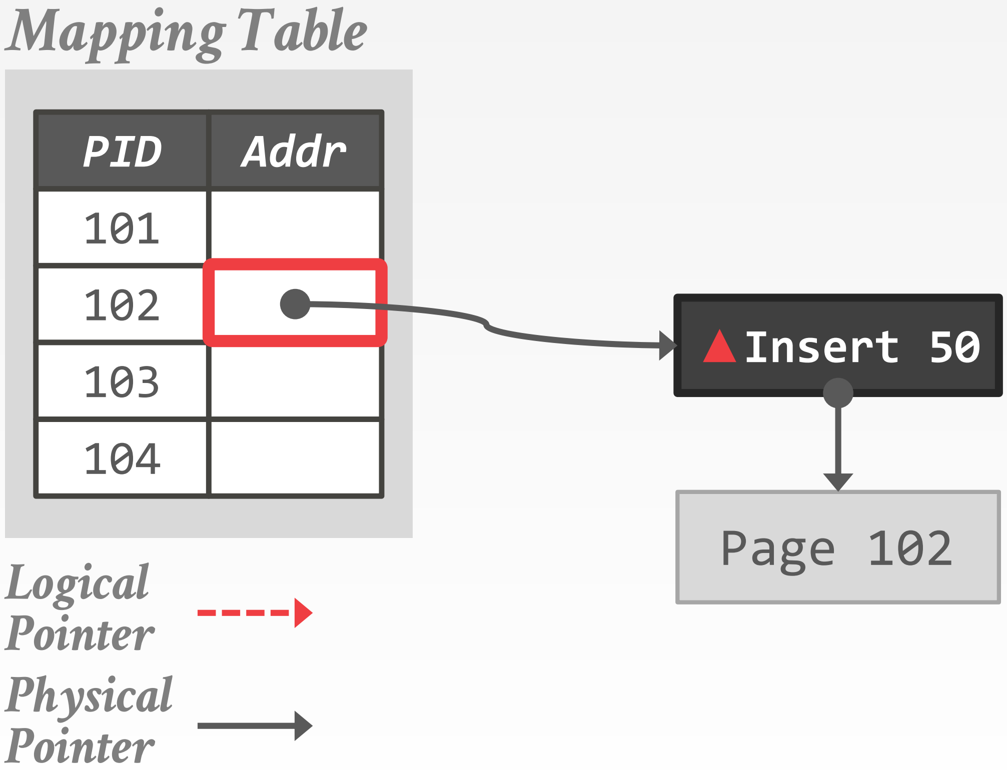
Each update to a page produces a new delta.

Delta physically points to base page.

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

Source: Justin Levandoski

BW-TREE: DELTA UPDATES



Mapping Table

PID	*Addr*
101	
102	●
103	
104	

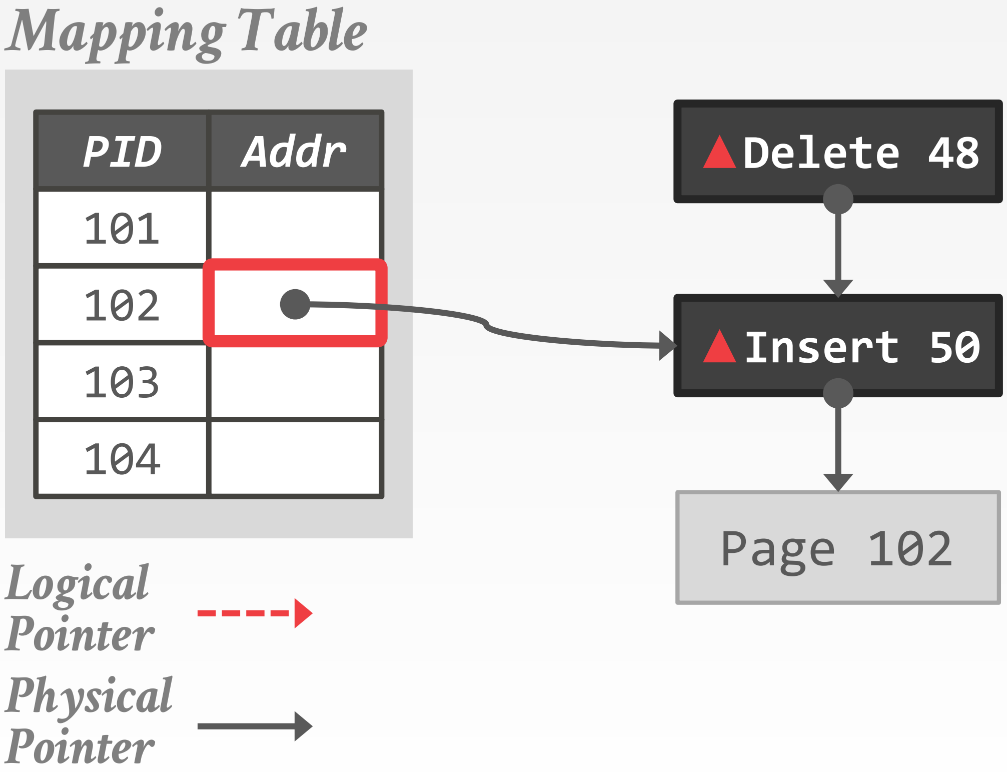
Each update to a page produces a new delta.

Delta physically points to base page.

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

Source: Justin Levandoski

BW-TREE: DELTA UPDATES



Mapping Table

PID	Addr
101	
102	●
103	
104	

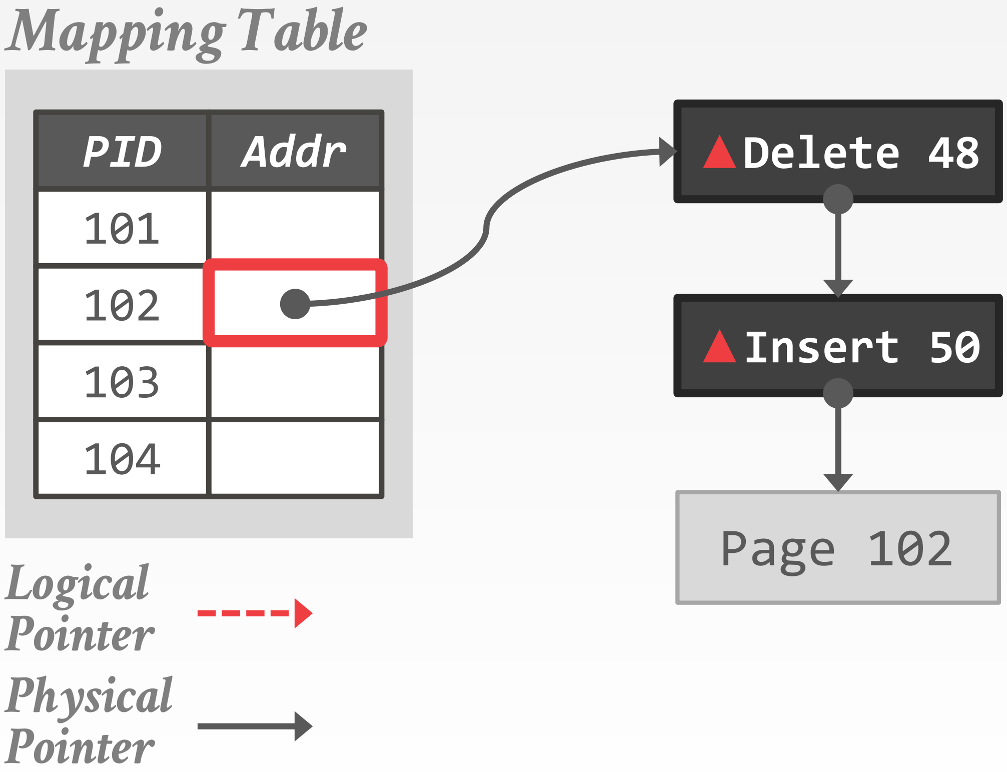
Each update to a page produces a new delta.

Delta physically points to base page.

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

Source: Justin Levandoski

BW-TREE: DELTA UPDATES



PID	Addr
101	
102	●
103	
104	

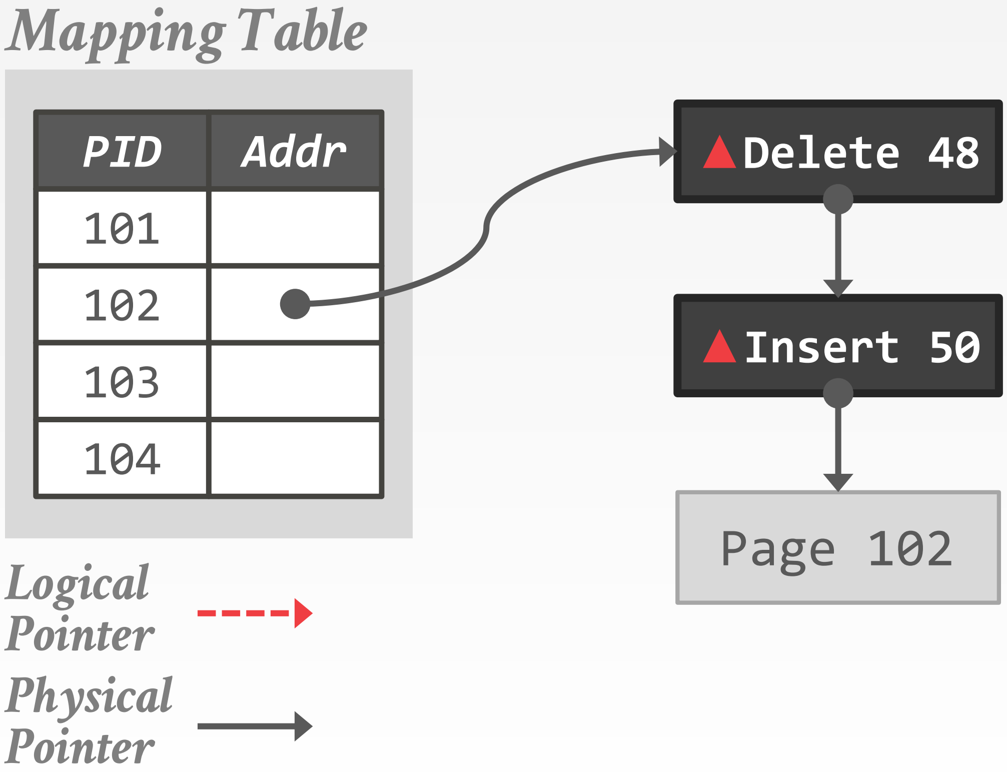
Each update to a page produces a new delta.

Delta physically points to base page.

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

Source: Justin Levandoski

BW-TREE: SEARCH

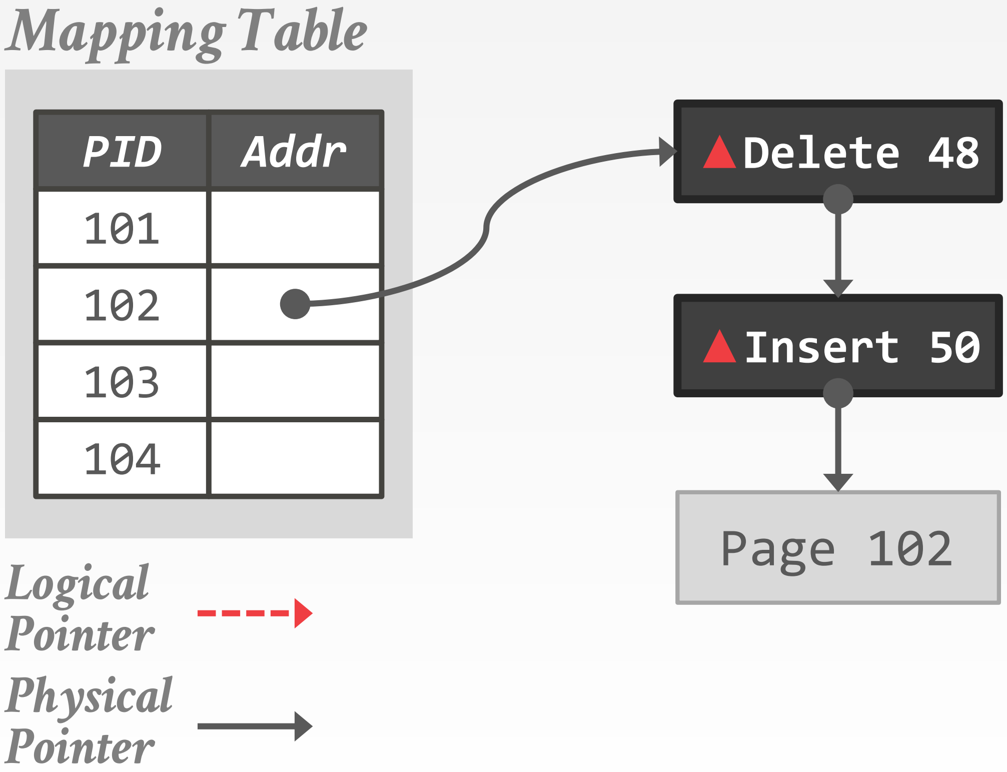


PID	Addr
101	
102	●
103	
104	

Traverse tree like a regular B+tree.

If mapping table points to delta chain, stop at first occurrence of search key.

BW-TREE: SEARCH



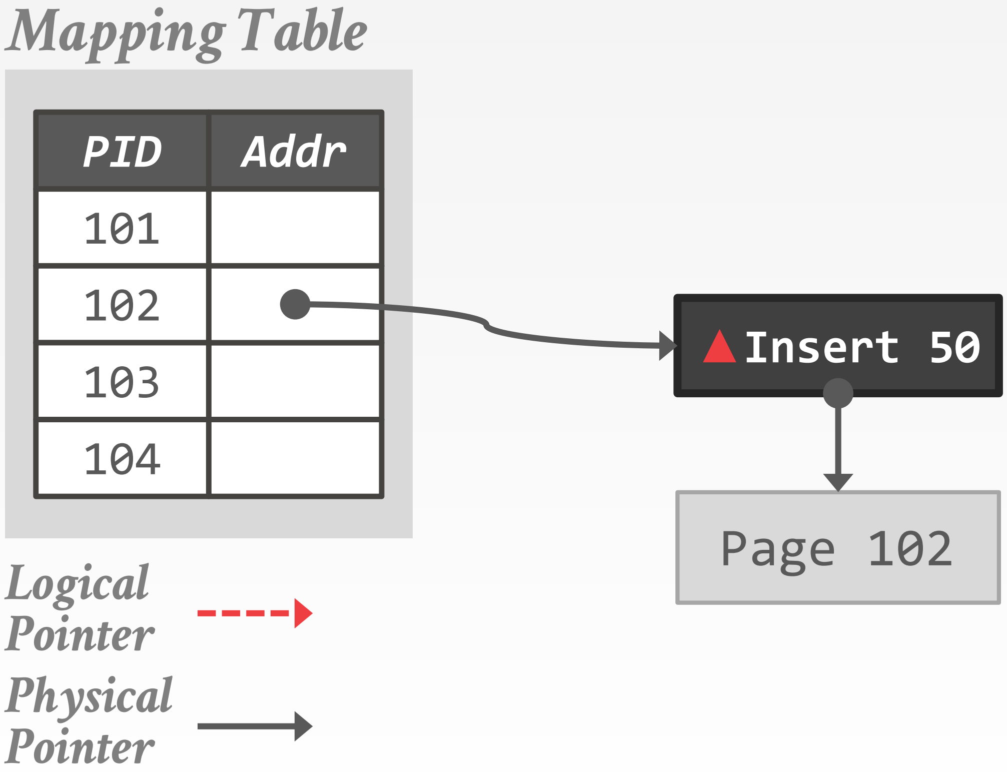
PID	Addr
101	
102	●
103	
104	

Traverse tree like a regular B+tree.

If mapping table points to delta chain, stop at first occurrence of search key.

Otherwise, perform binary search on base page.

BW-TREE: CONTENTION UPDATES

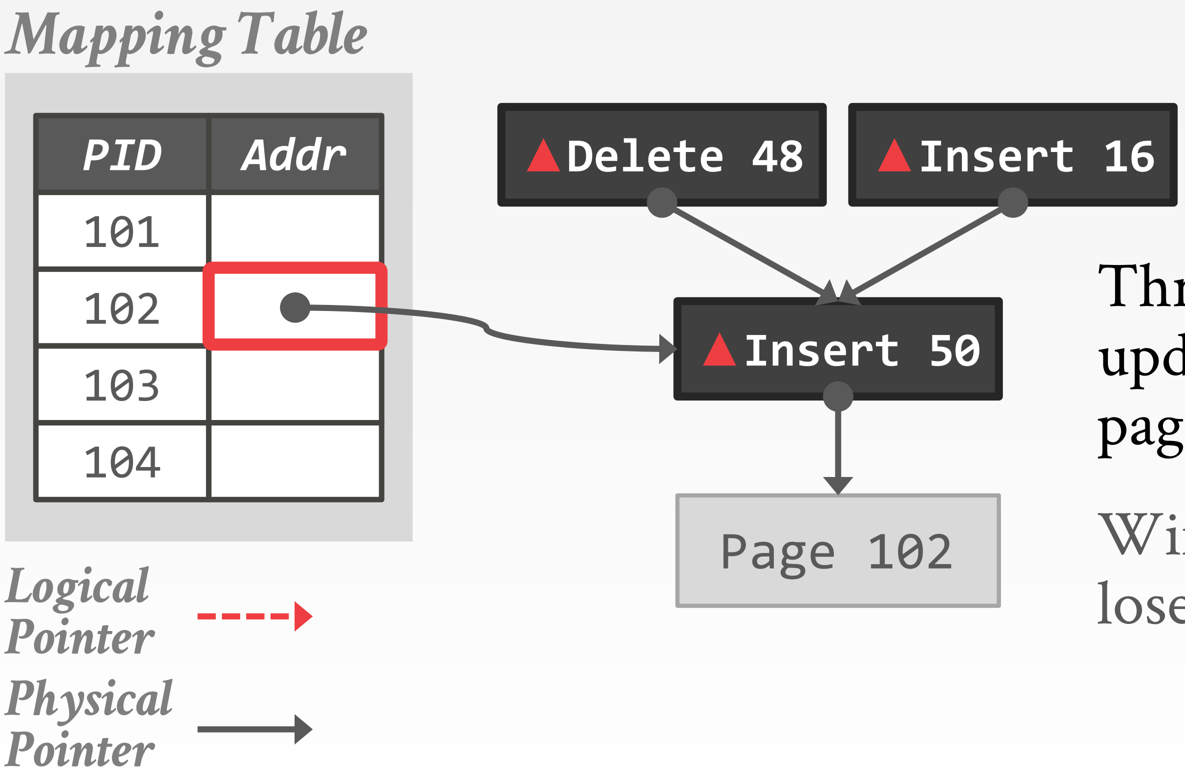


Mapping Table

PID	Addr
101	
102	●
103	
104	

Threads may try to install updates to same state of the page.

BW-TREE: CONTENTION UPDATES



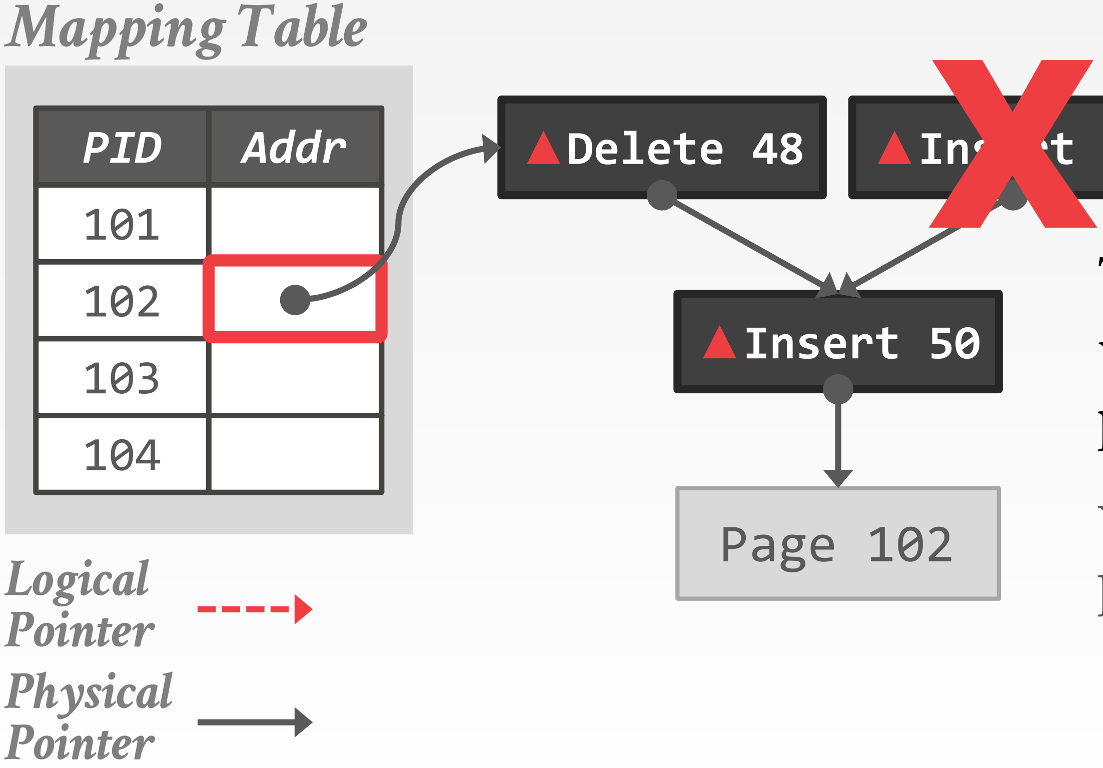
Mapping Table

PID	*Addr*
101	
102	●
103	
104	

Threads may try to install updates to same state of the page.

Winner succeeds, any losers must retry or abort

BW-TREE: CONTENTION UPDATES



Mapping Table

PID	*Addr*
101	
102	
103	
104	

Threads may try to install updates to same state of the page.

Winner succeeds, any losers must retry or abort

BW-TREE: DELTA TYPES

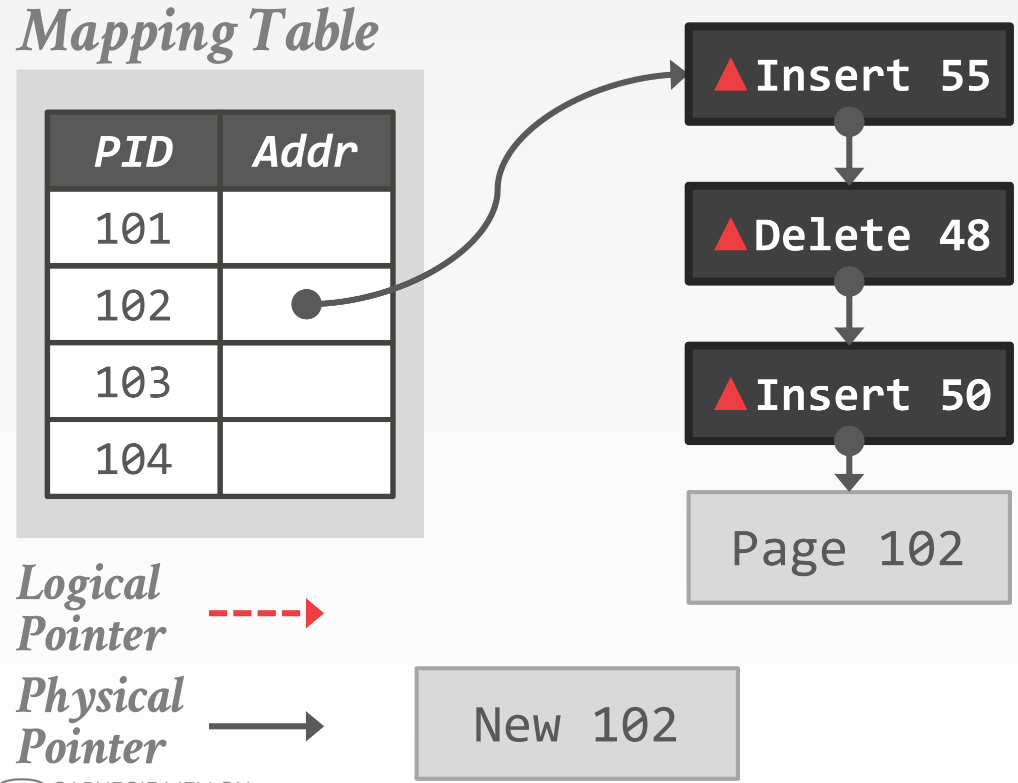
Record Update Deltas

→ Insert/Delete/Update of record on a page

Structure Modification Deltas

→ Split/Merge information

BW-TREE: CONSOLIDATION

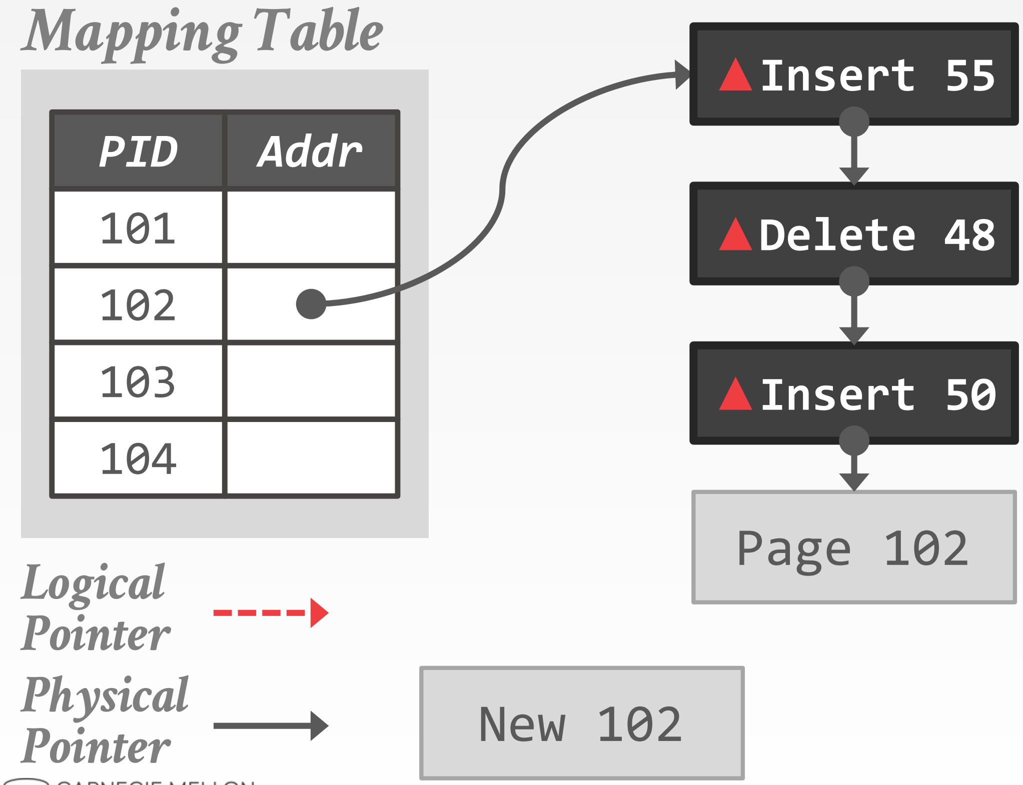


Mapping Table

PID	*Addr*
101	
102	●
103	
104	

Consolidate updates by creating new page with deltas applied.

BW-TREE: CONSOLIDATION



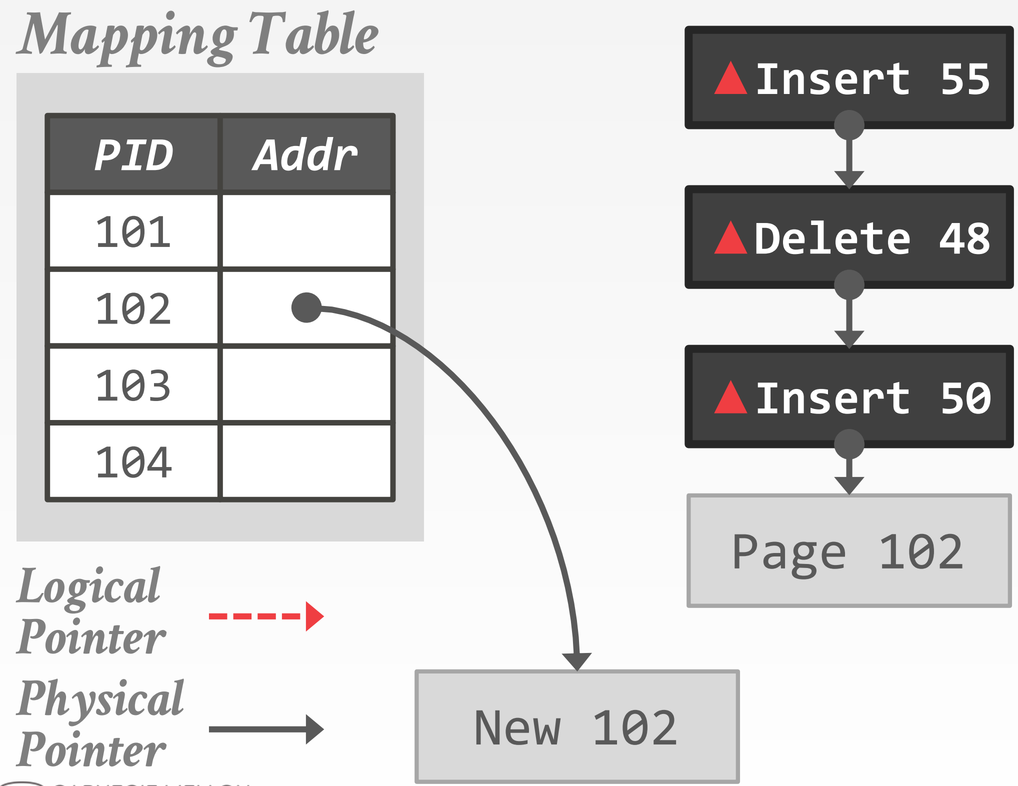
Mapping Table

PID	*Addr*
101	
102	●
103	
104	

Consolidate updates by creating new page with deltas applied.

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

BW-TREE: CONSOLIDATION



Mapping Table

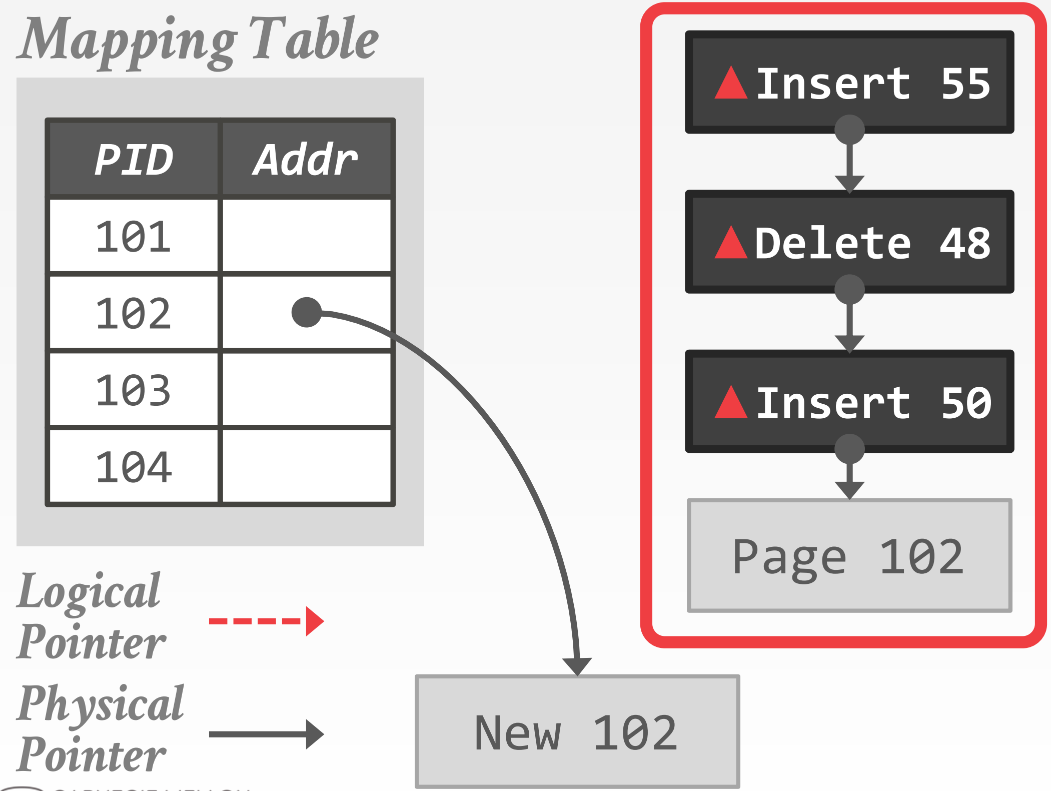
PID	*Addr*
101	
102	●
103	
104	

Consolidate updates by creating new page with deltas applied.

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

BW-TREE: CONSOLIDATION

Mapping Table



PID	*Addr*
101	
102	●
103	
104	

Consolidate updates by creating new page with deltas applied.

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

Old page + deltas are marked as garbage.

BW-TREE: GARBAGE COLLECTION

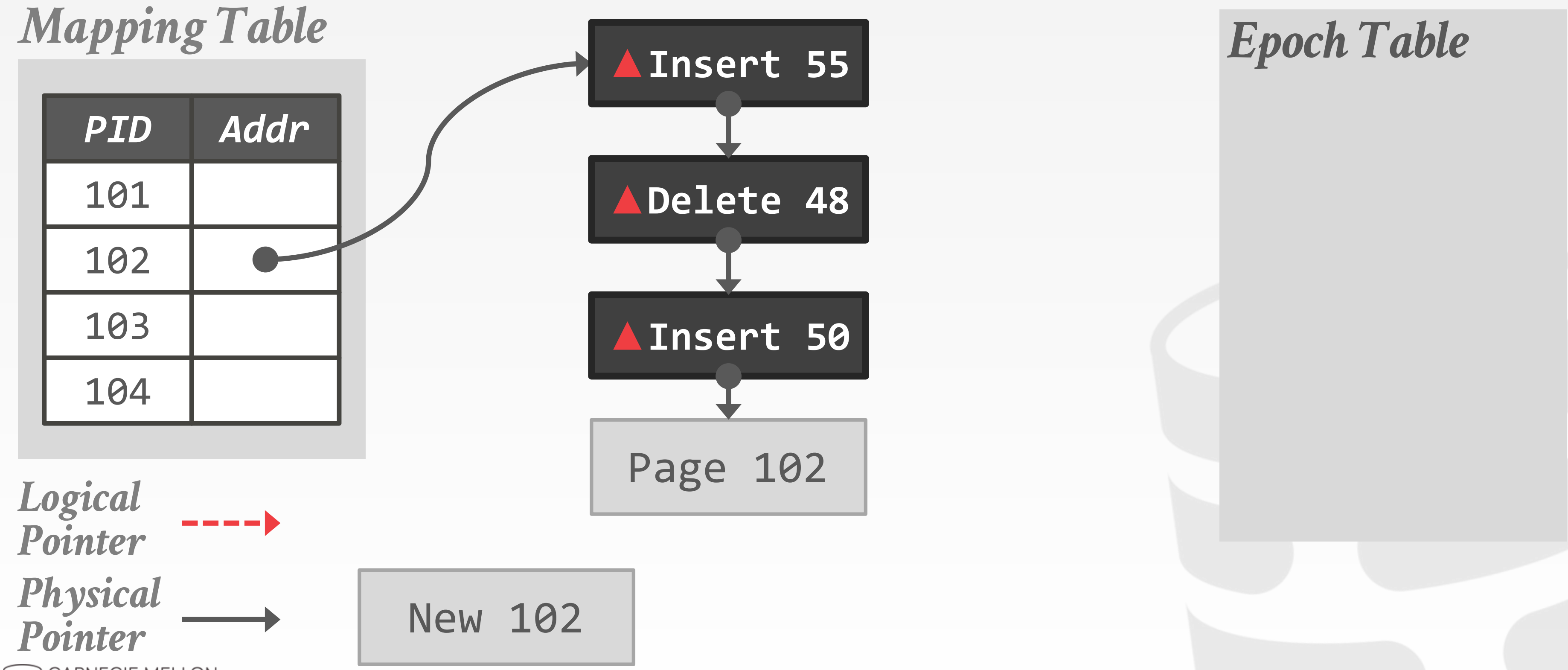
Operations are tagged with an **epoch**

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

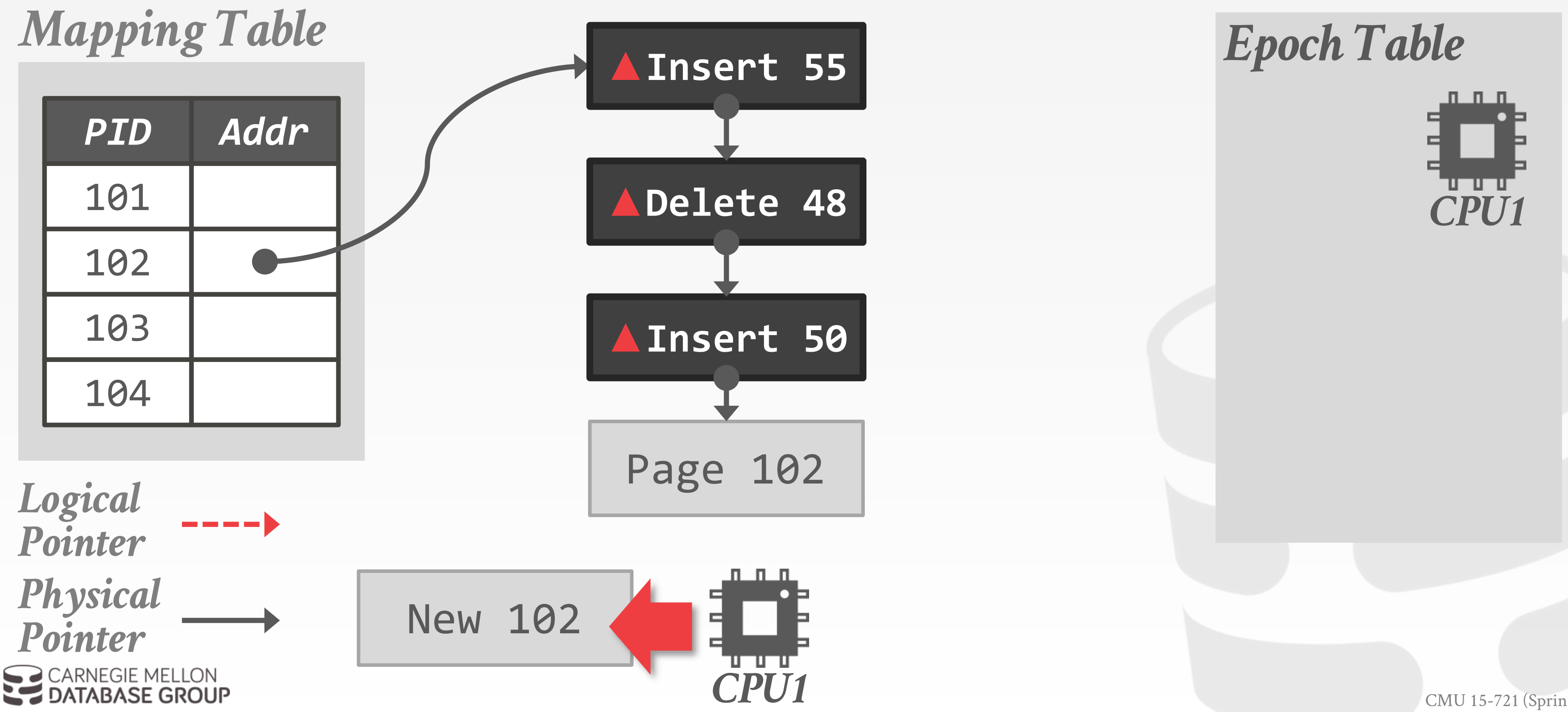
→ Thread joins an epoch prior to each operation and post objects that can be reclaimed for the current epoch (not necessarily the one it joined)

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

BW-TREE: GARBAGE COLLECTION



BW-TREE: GARBAGE COLLECTION



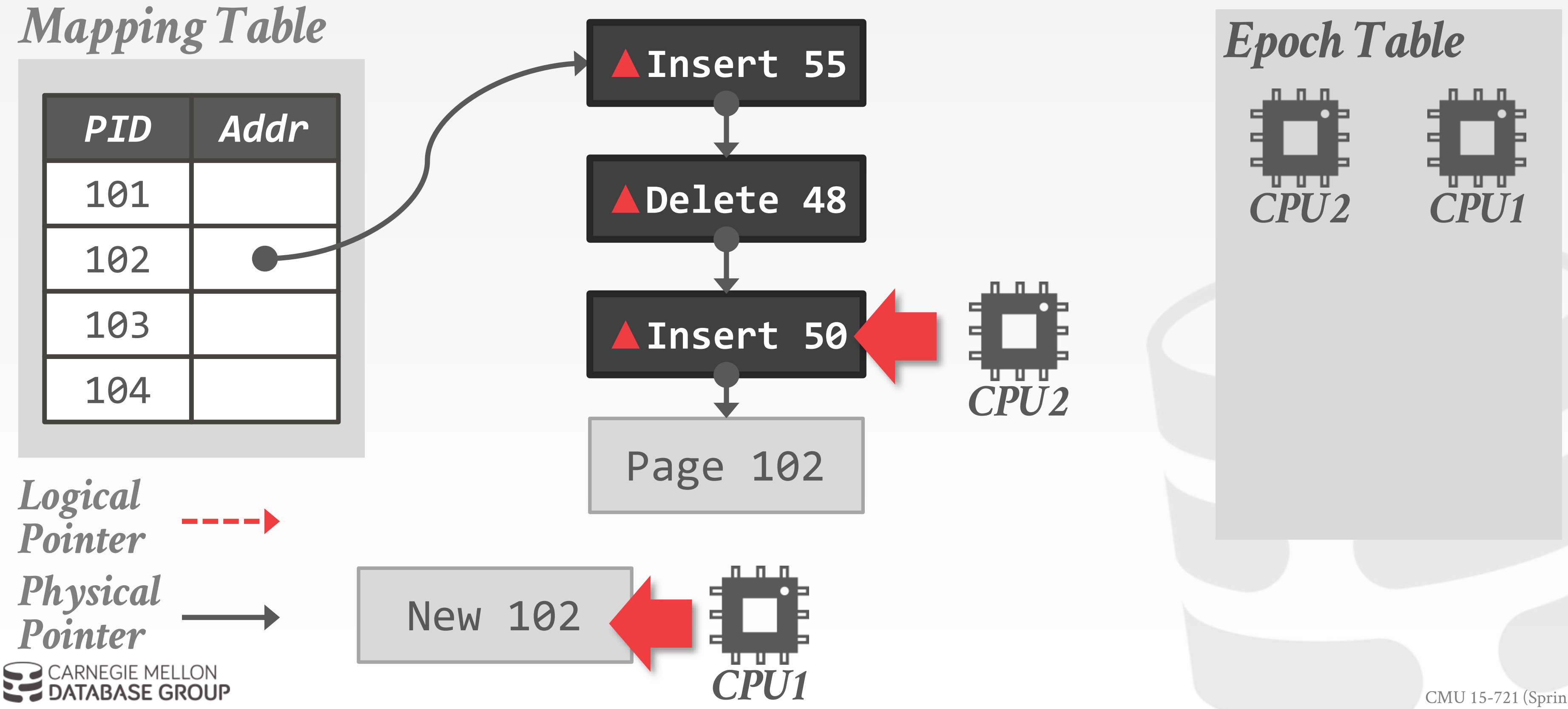
Mapping Table

PID	*Addr*
101	
102	●
103	
104	

Logical Pointer

Physical Pointer

BW-TREE: GARBAGE COLLECTION



Mapping Table

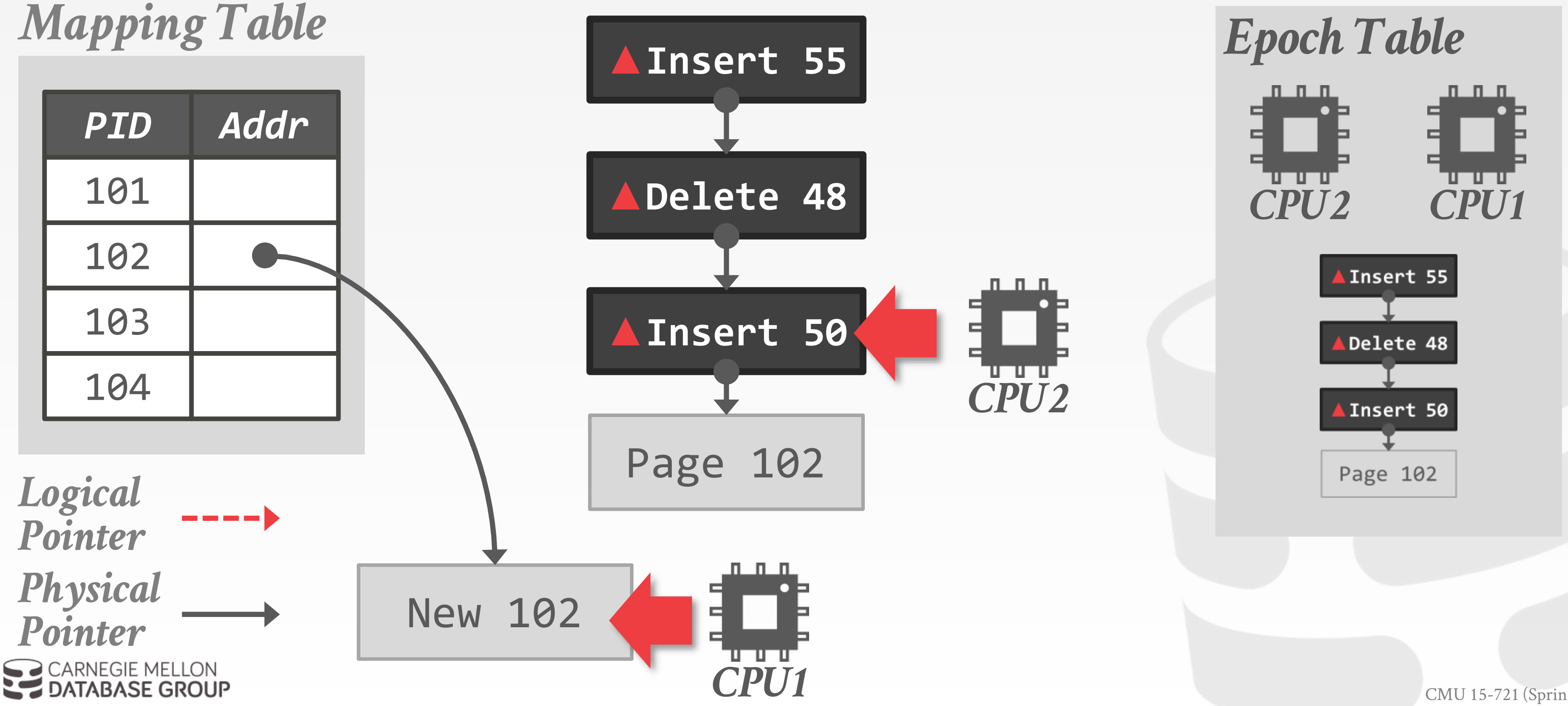
PID	Addr
101	
102	●
103	
104	

Epoch Table

Logical Pointer

Physical Pointer

BW-TREE: GARBAGE COLLECTION



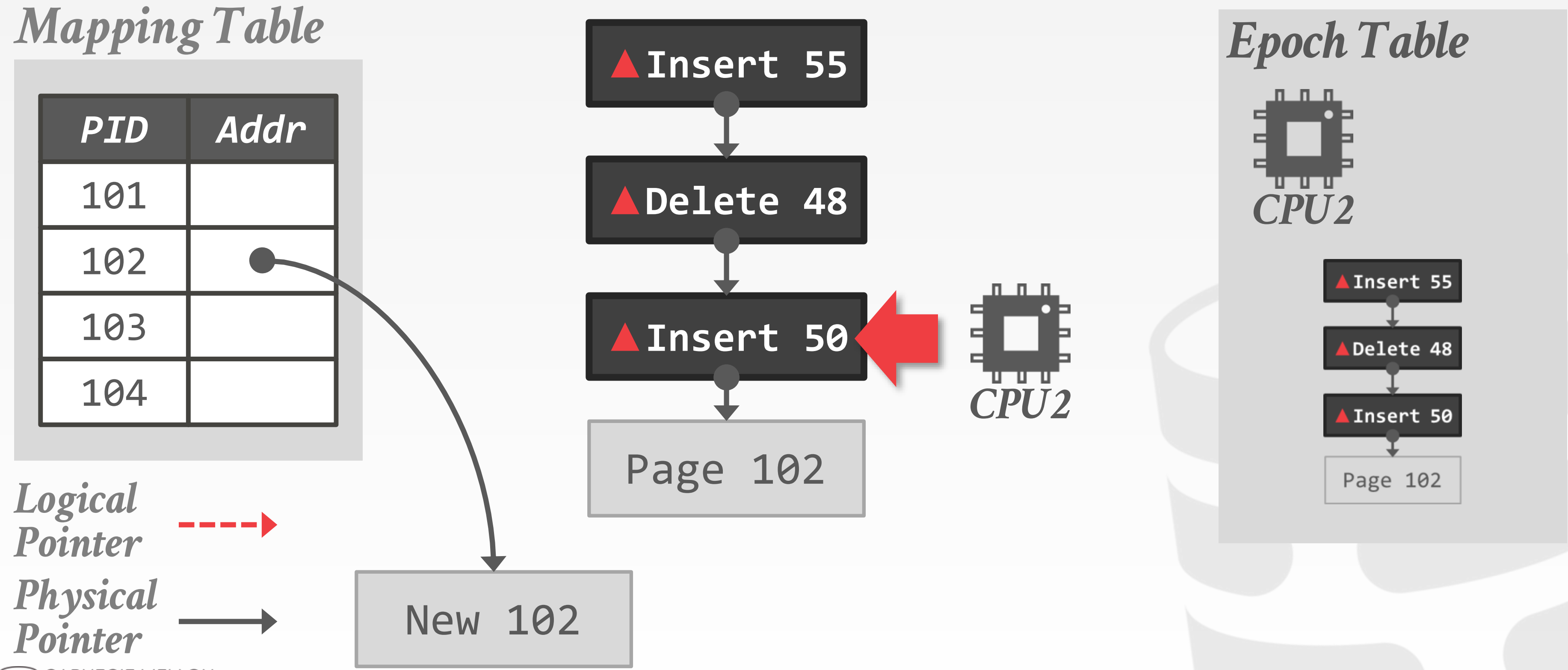
Mapping Table

PID	*Addr*
101	
102	●
103	
104	

Logical Pointer

Physical Pointer

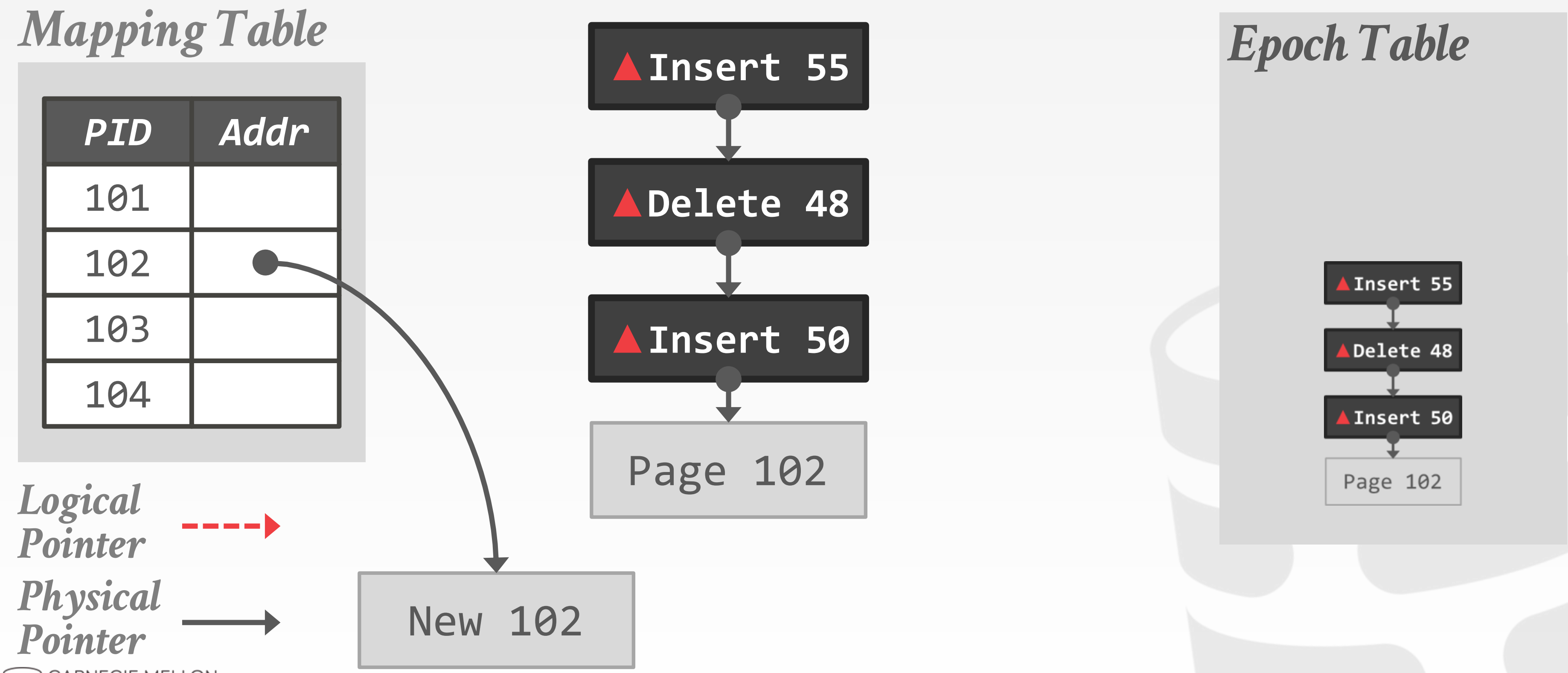
BW-TREE: GARBAGE COLLECTION



Mapping Table

PID	Addr
101	
102	●
103	
104	

BW-TREE: GARBAGE COLLECTION



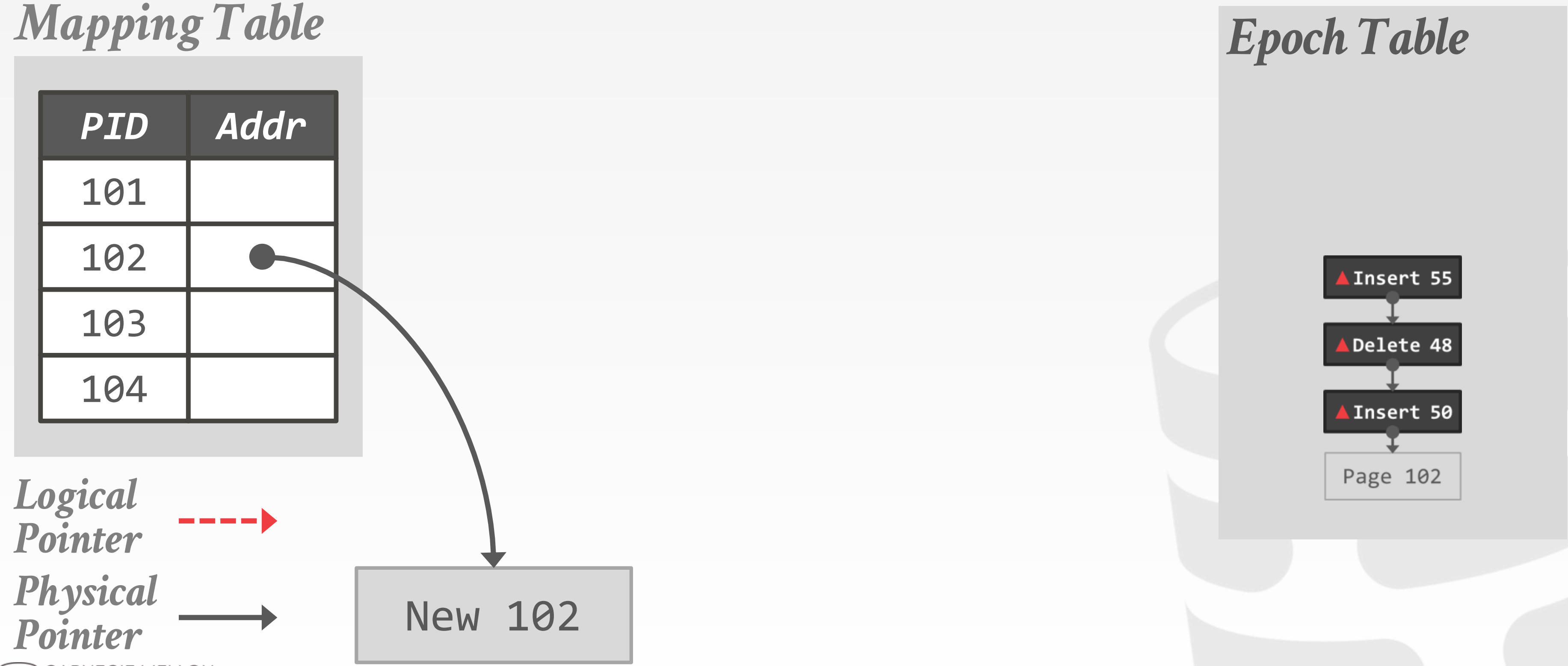
Mapping Table

PID	Addr
101	
102	●
103	
104	

Logical Pointer - - - ▶

Physical Pointer ——▶

BW-TREE: GARBAGE COLLECTION



Mapping Table

PID	Addr
101	
102	●
103	
104	

Logical Pointer

Physical Pointer

BW-TREE: STRUCTURE MODIFICATIONS

Split Delta Record

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

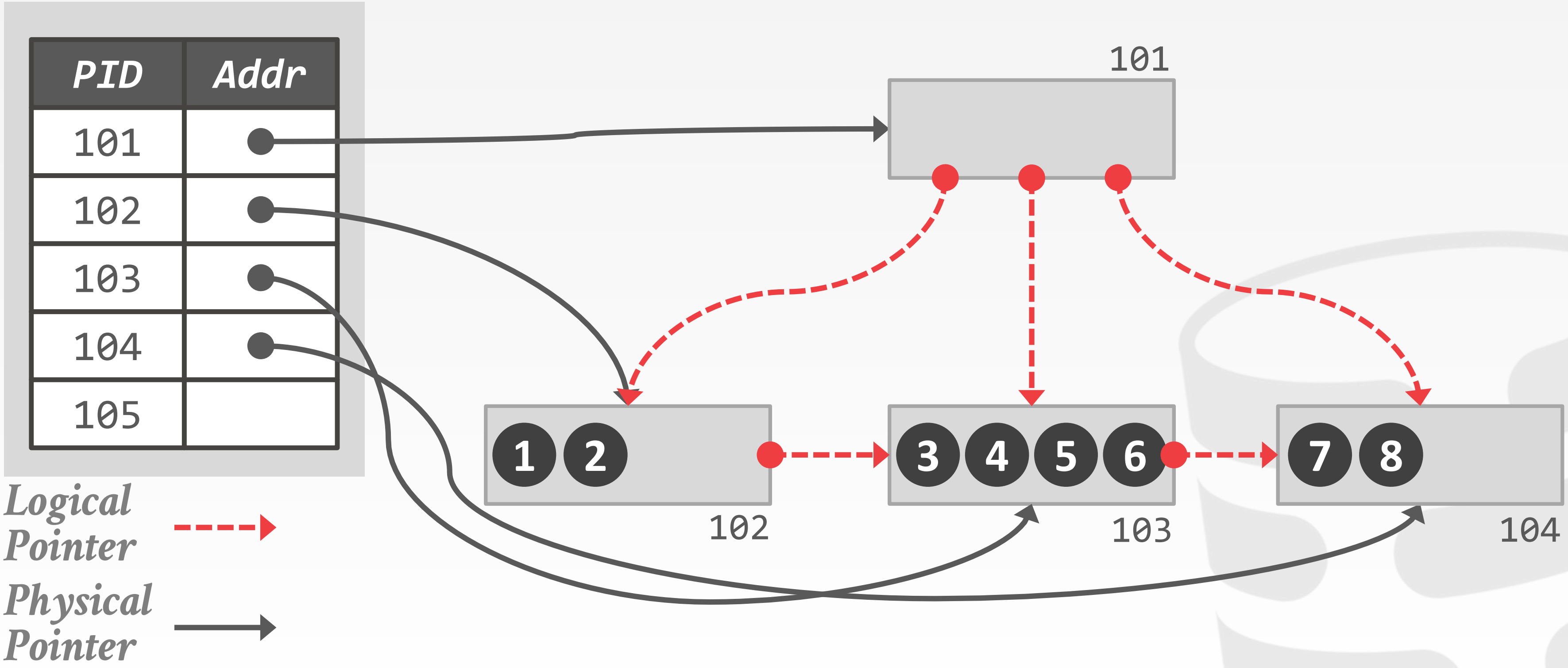
→ Use a logical pointer to the new page.

Separator Delta Record

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

BW-TREE: STRUCTURE MODIFICATIONS

Mapping Table



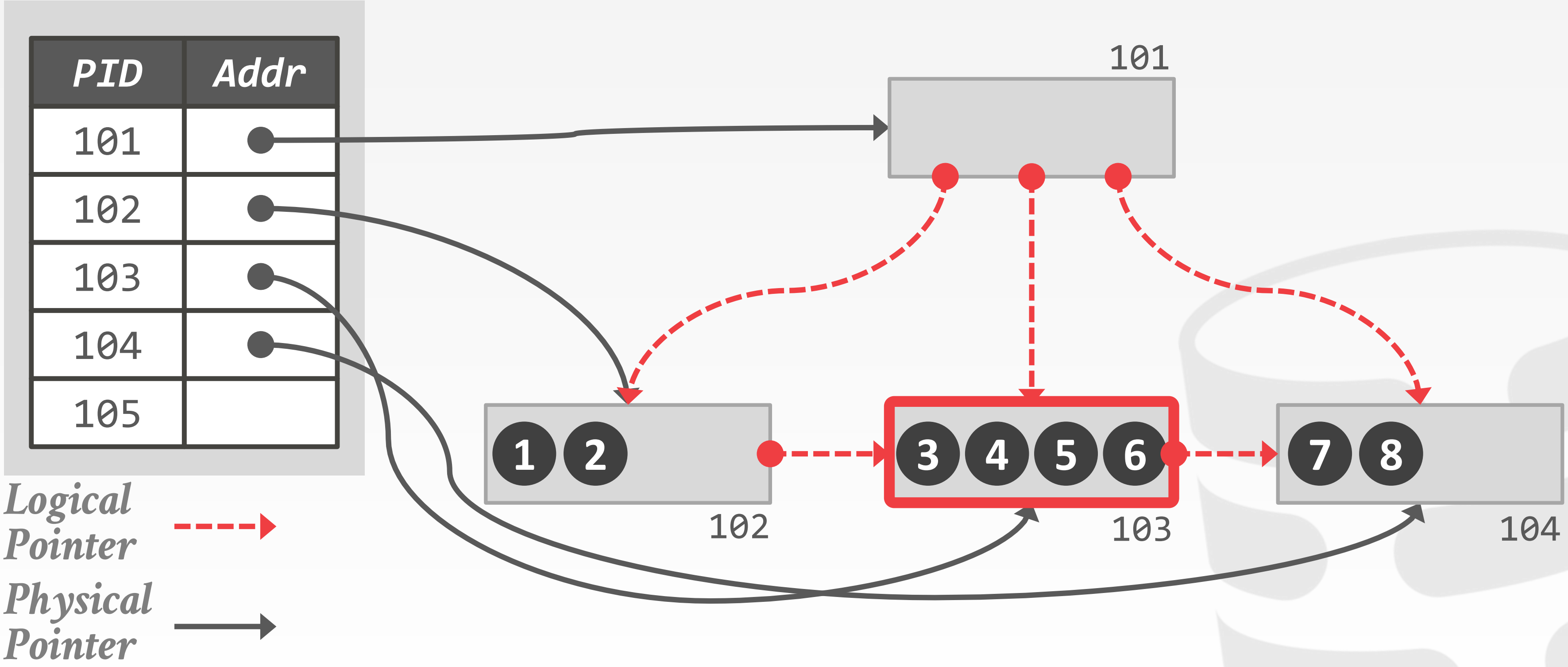
PID	*Addr*
101	●
102	●
103	●
104	●
105	

Logical Pointer

Physical Pointer

BW-TREE: STRUCTURE MODIFICATIONS

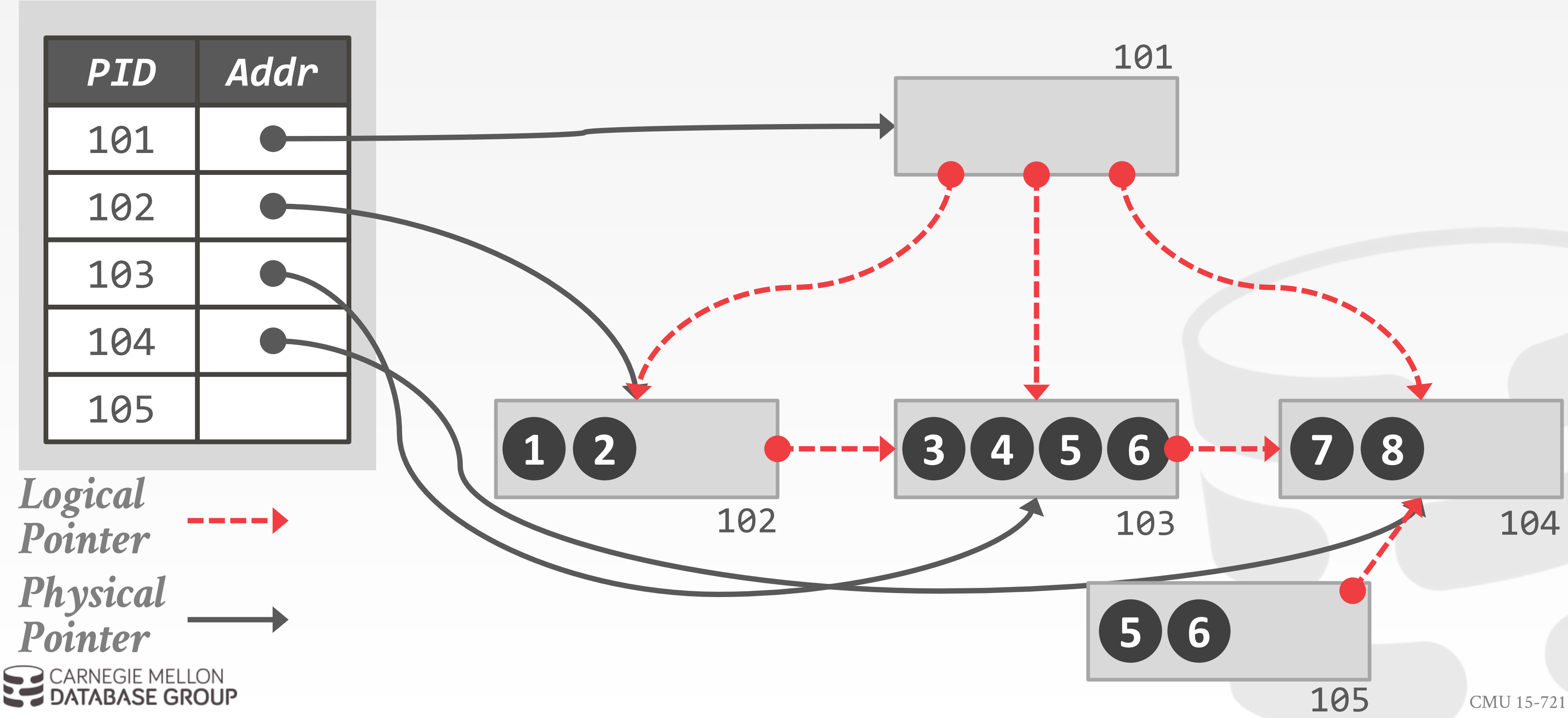
Mapping Table



PID	Addr
101	●
102	●
103	●
104	●
105	

BW-TREE: STRUCTURE MODIFICATIONS

Mapping Table

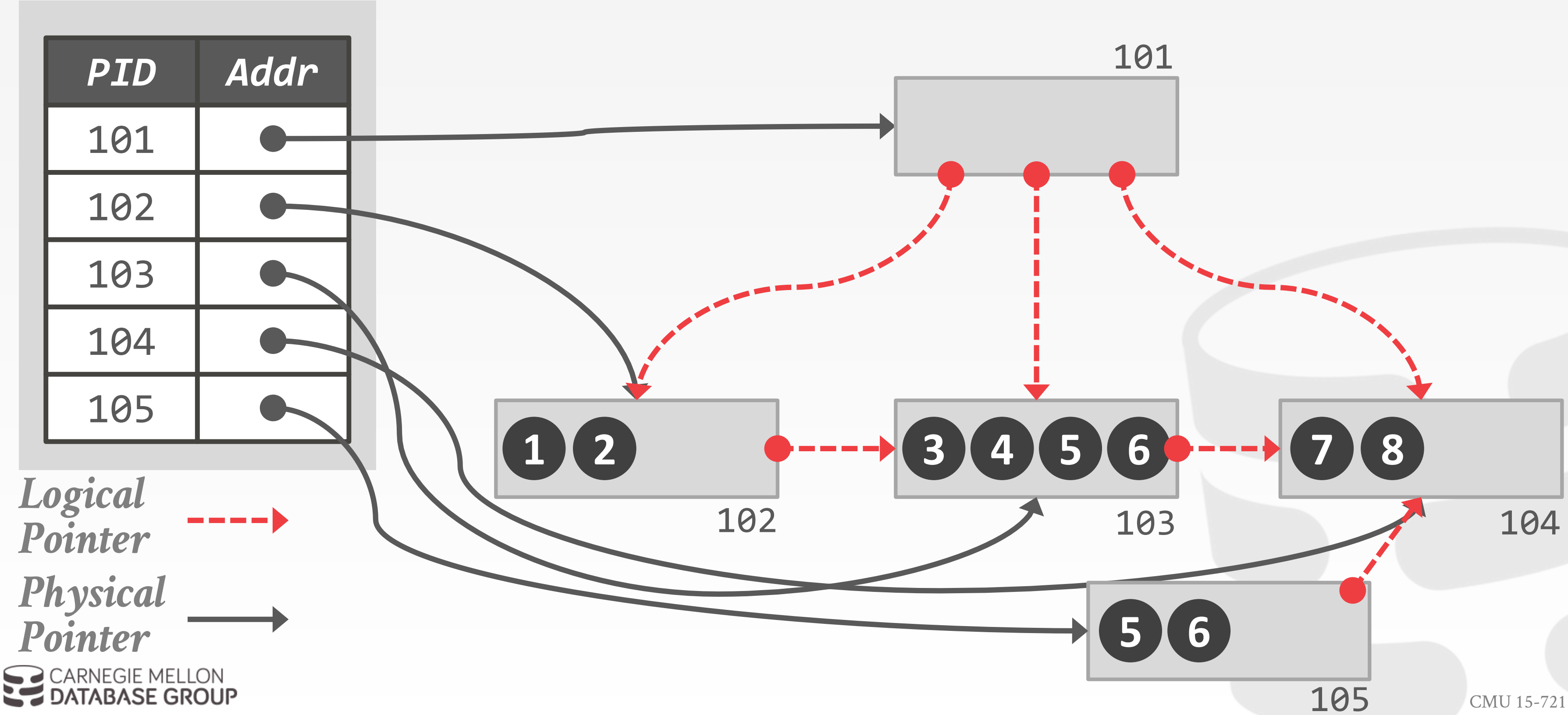


PID	Addr
101	●
102	●
103	●
104	●
105	

Logical Pointer

Physical Pointer

BW-TREE: STRUCTURE MODIFICATIONS



Mapping Table

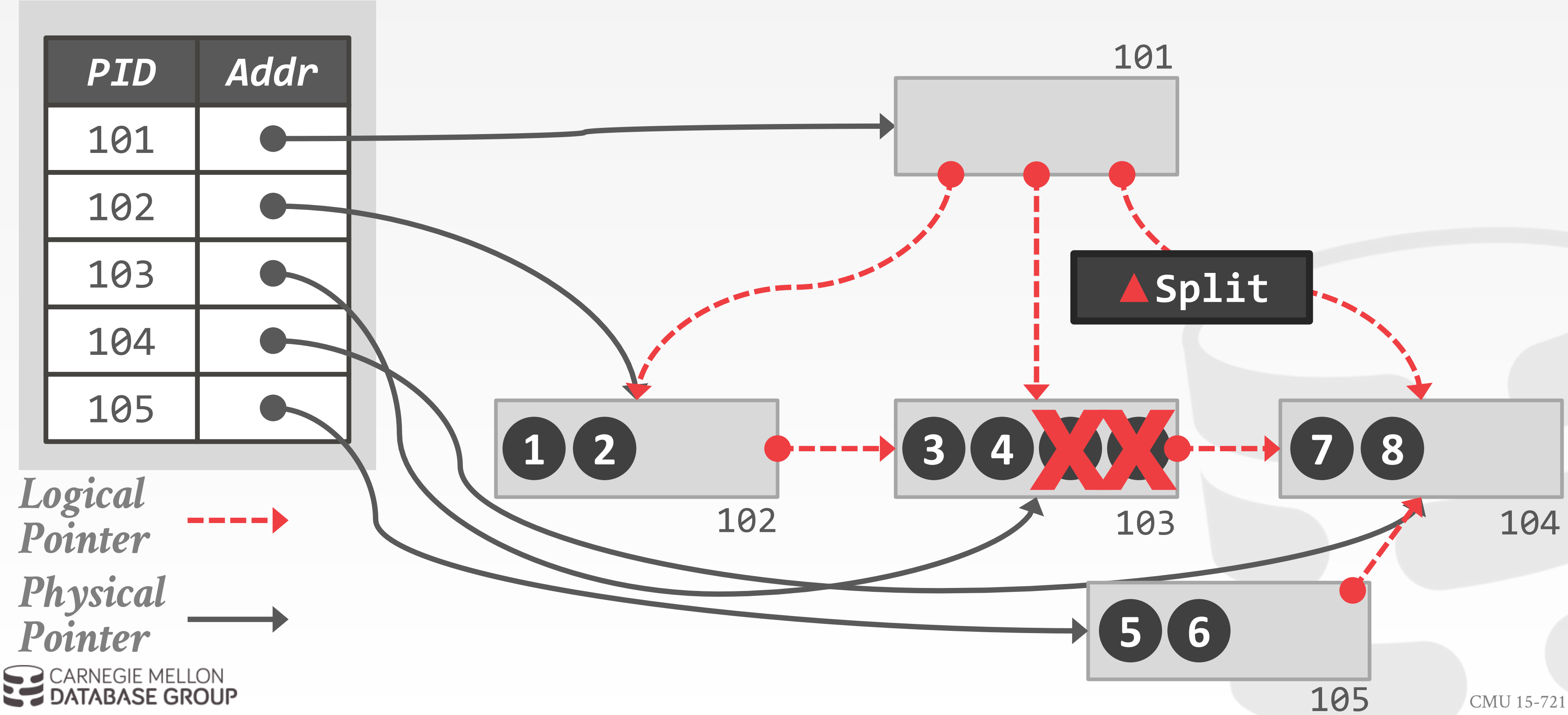
PID	*Addr*
101	●
102	●
103	●
104	●
105	●

Logical Pointer

Physical Pointer

BW-TREE: STRUCTURE MODIFICATIONS

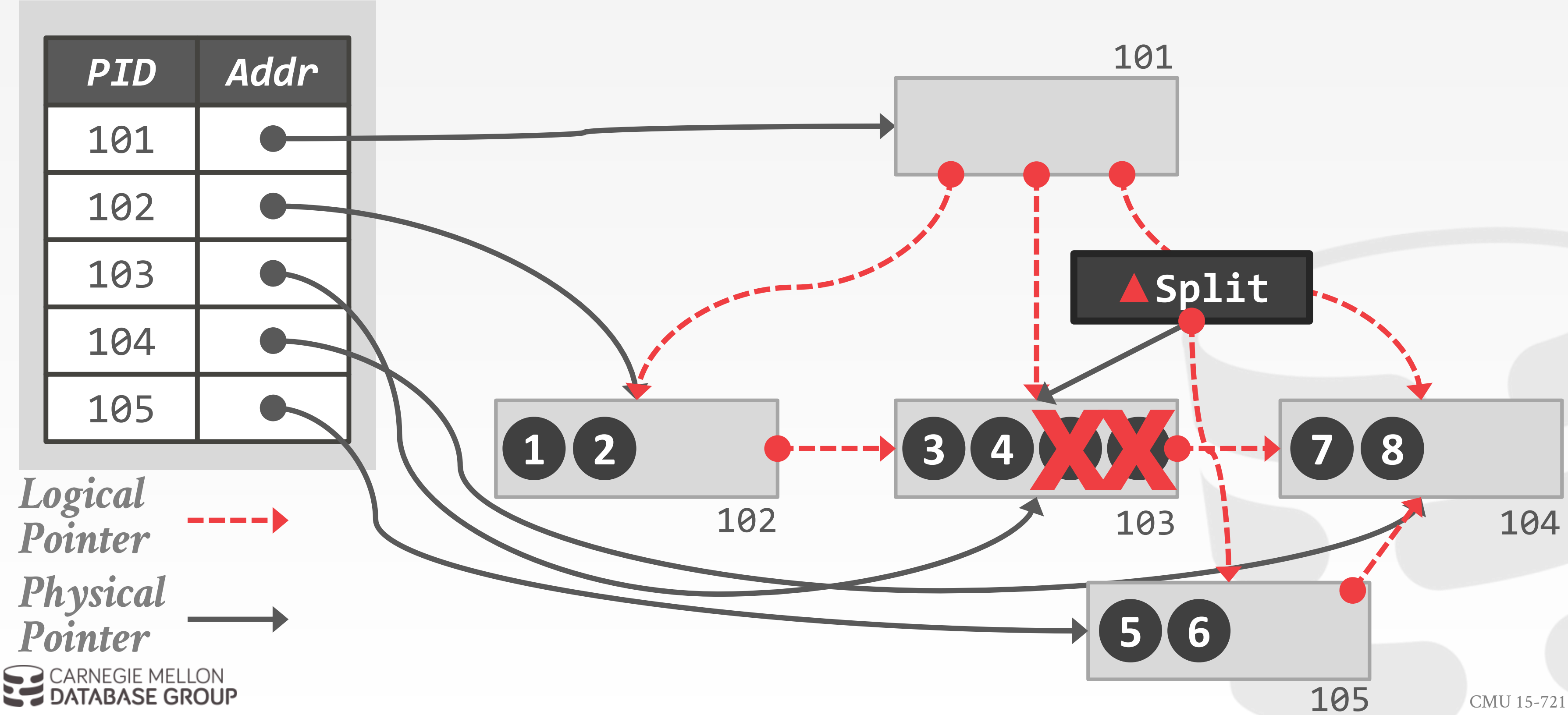
Mapping Table



PID	*Addr*
101	●
102	●
103	●
104	●
105	●

BW-TREE: STRUCTURE MODIFICATIONS

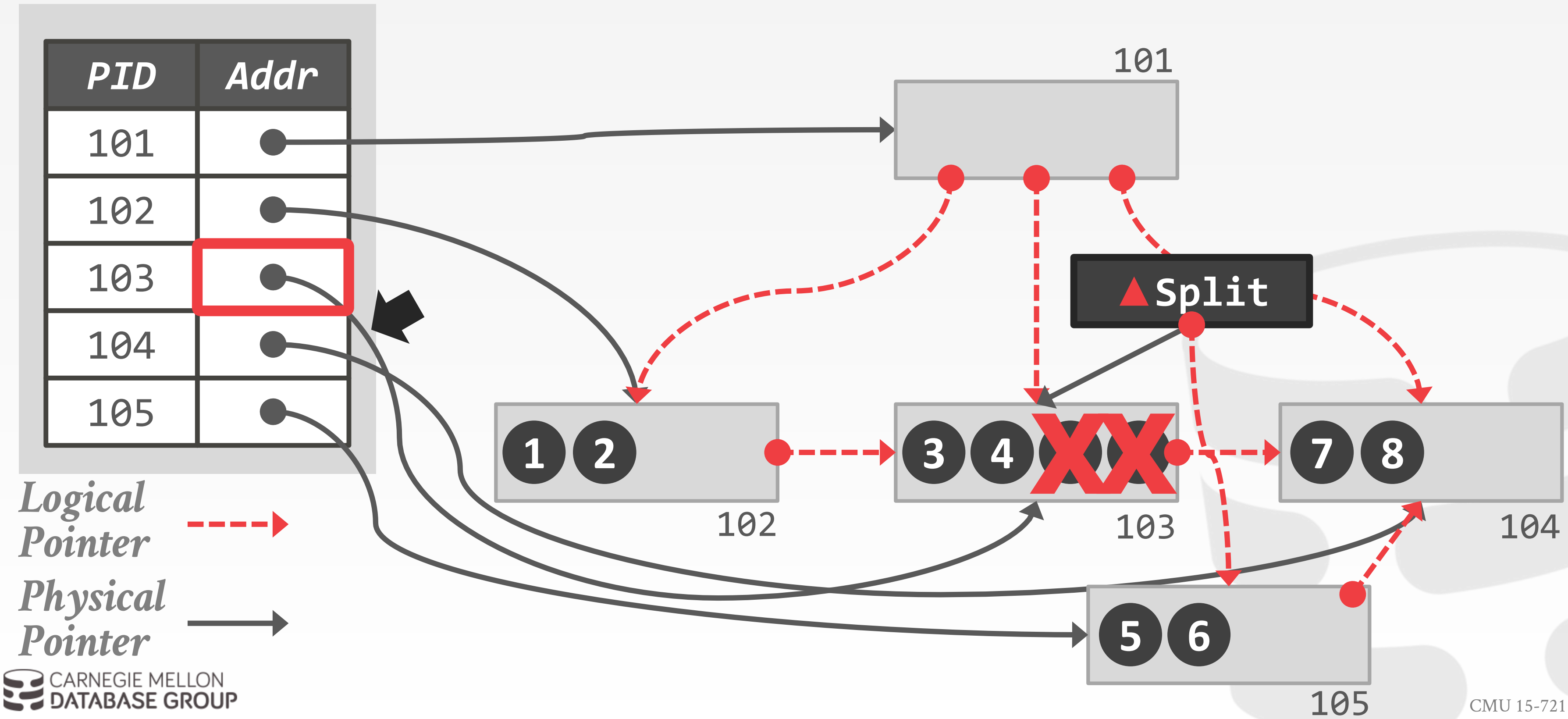
Mapping Table



PID	Addr
101	●
102	●
103	●
104	●
105	●

BW-TREE: STRUCTURE MODIFICATIONS

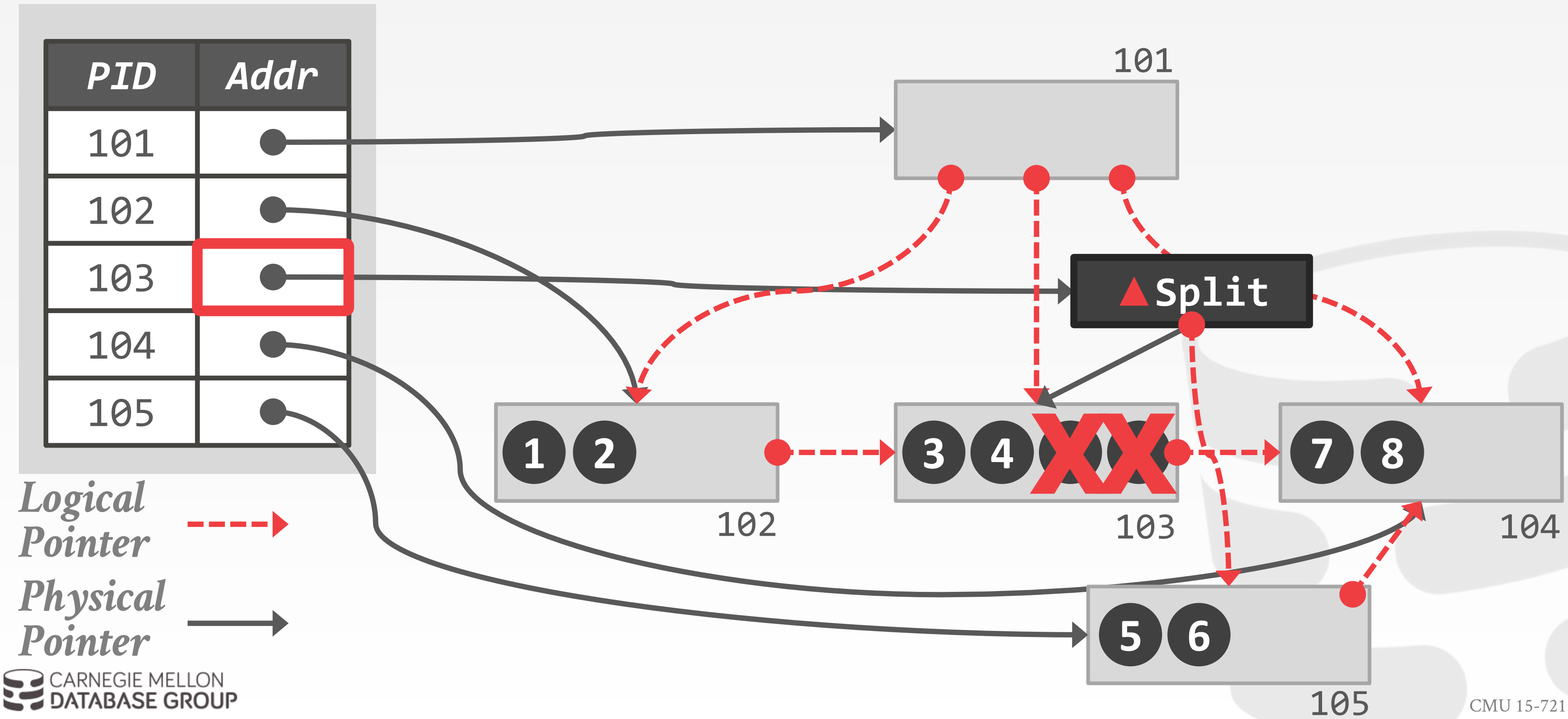
Mapping Table



PID	*Addr*
101	●
102	●
103	●
104	●
105	●

BW-TREE: STRUCTURE MODIFICATIONS

Mapping Table

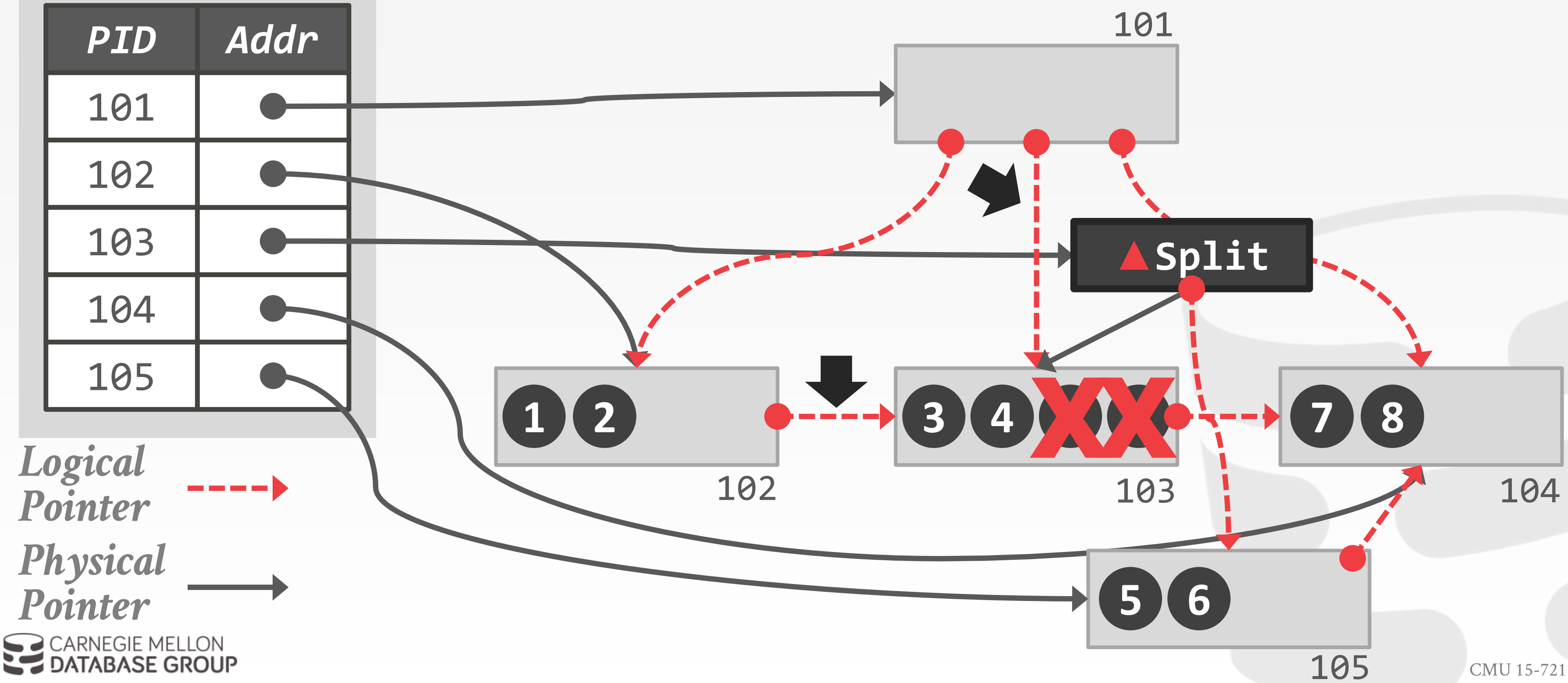


PID	Addr
101	●
102	●
103	●
104	●
105	●

BW-TREE: STRUCTURE MODIFICATIONS

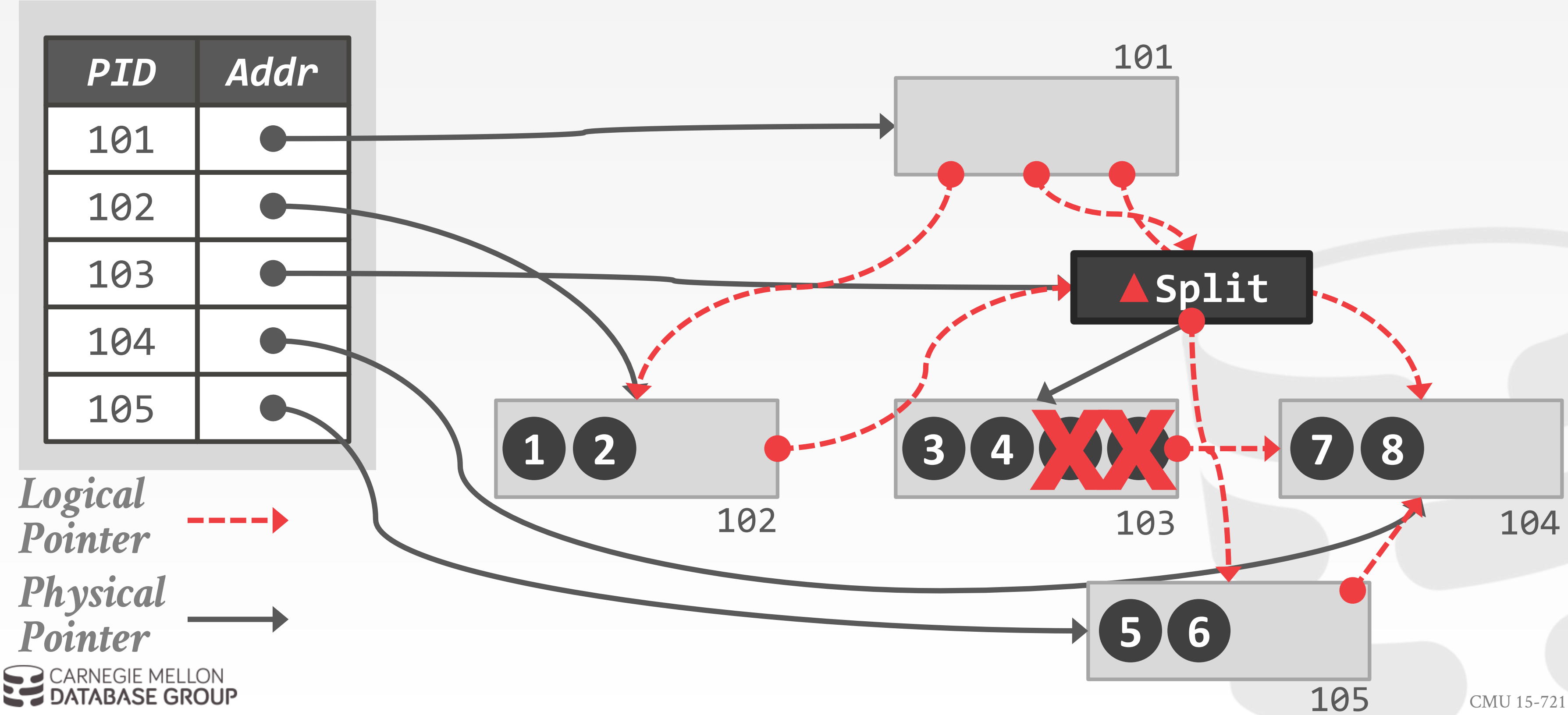
Mapping Table

PID	Addr
101	●
102	●
103	●
104	●
105	●



BW-TREE: STRUCTURE MODIFICATIONS

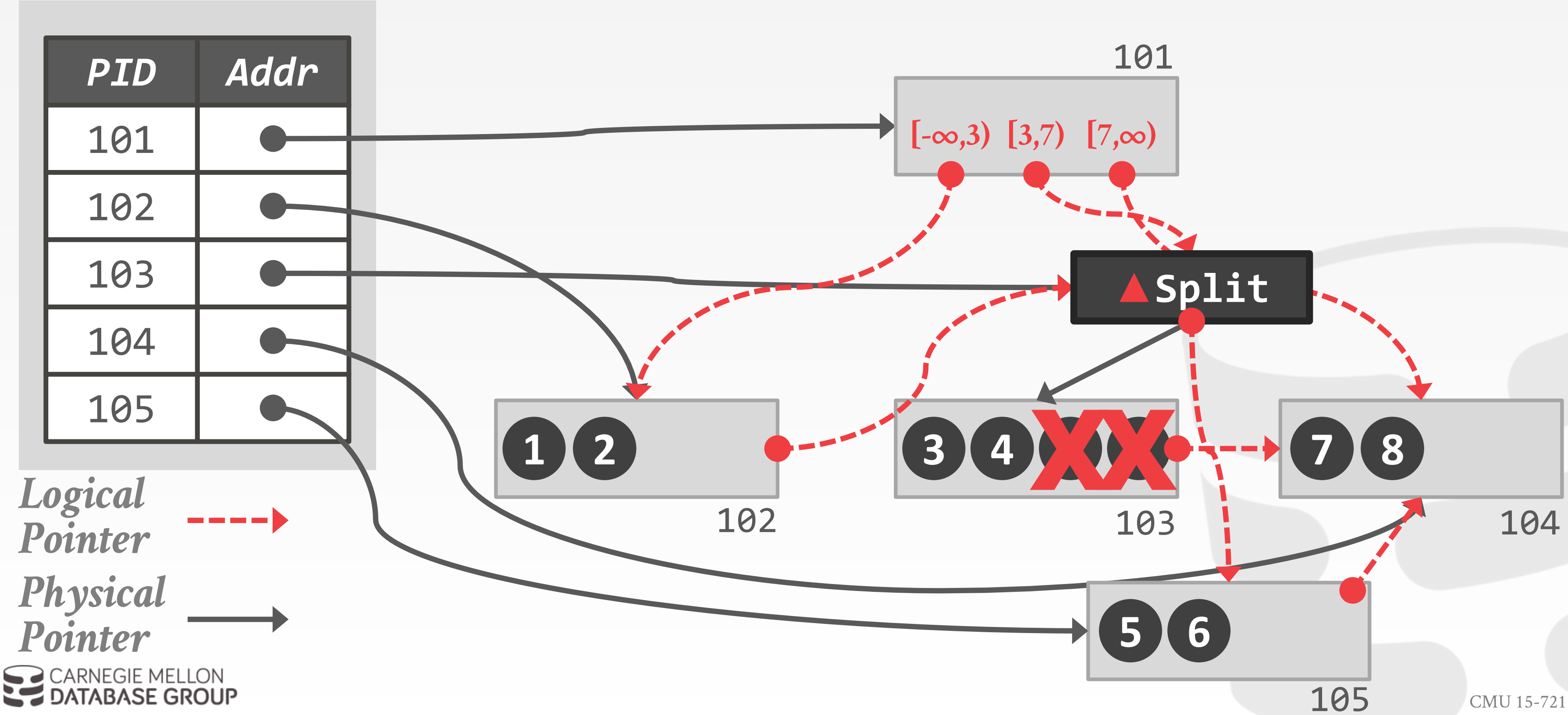
Mapping Table



PID	*Addr*
101	●
102	●
103	●
104	●
105	●

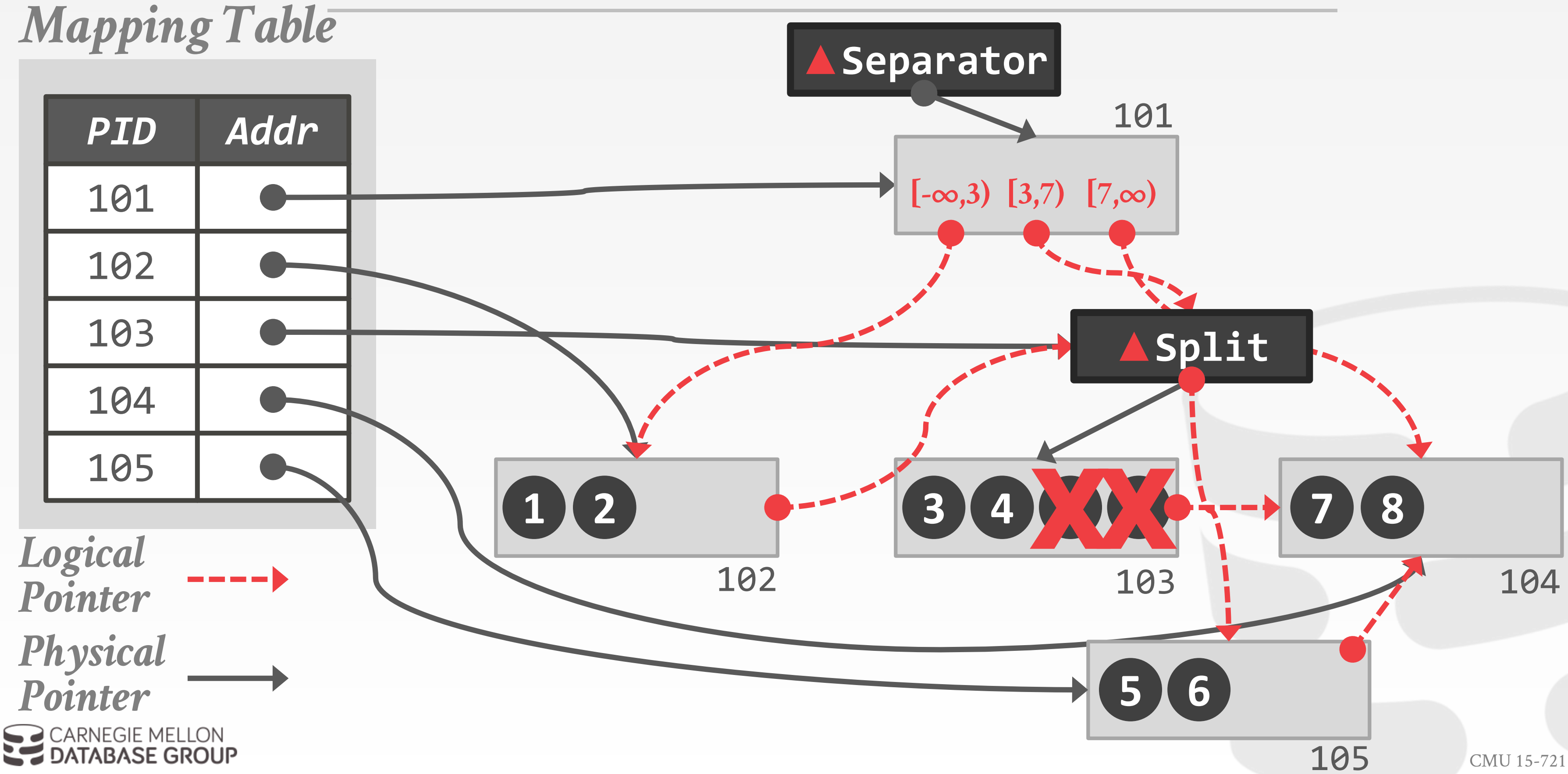
BW-TREE: STRUCTURE MODIFICATIONS

Mapping Table



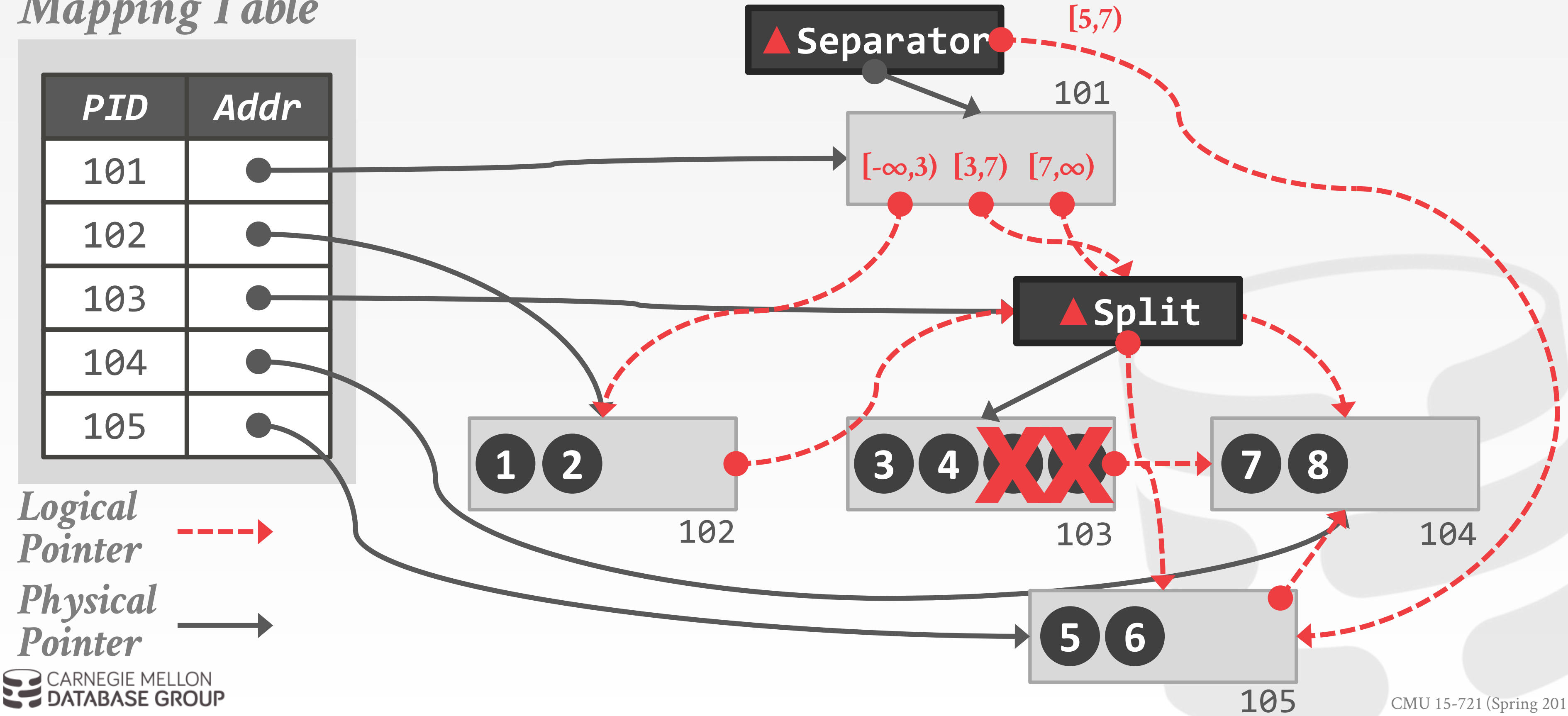
PID	Addr
101	●
102	●
103	●
104	●
105	●

BW-TREE: STRUCTURE MODIFICATIONS



BW-TREE: STRUCTURE MODIFICATIONS

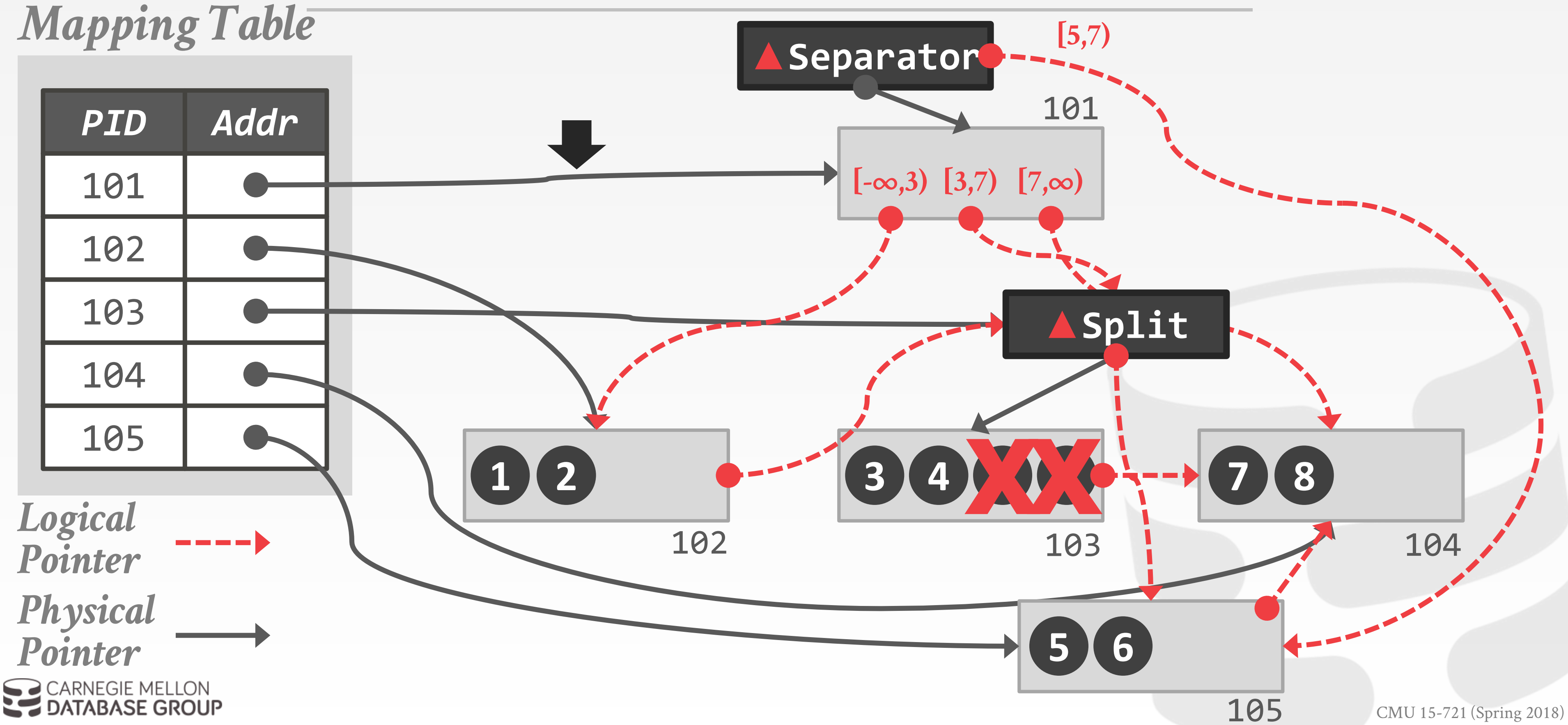
Mapping Table



PID	*Addr*
101	●
102	●
103	●
104	●
105	●

CARNEGIE MELLON
DATABASE GROUP

BW-TREE: STRUCTURE MODIFICATIONS



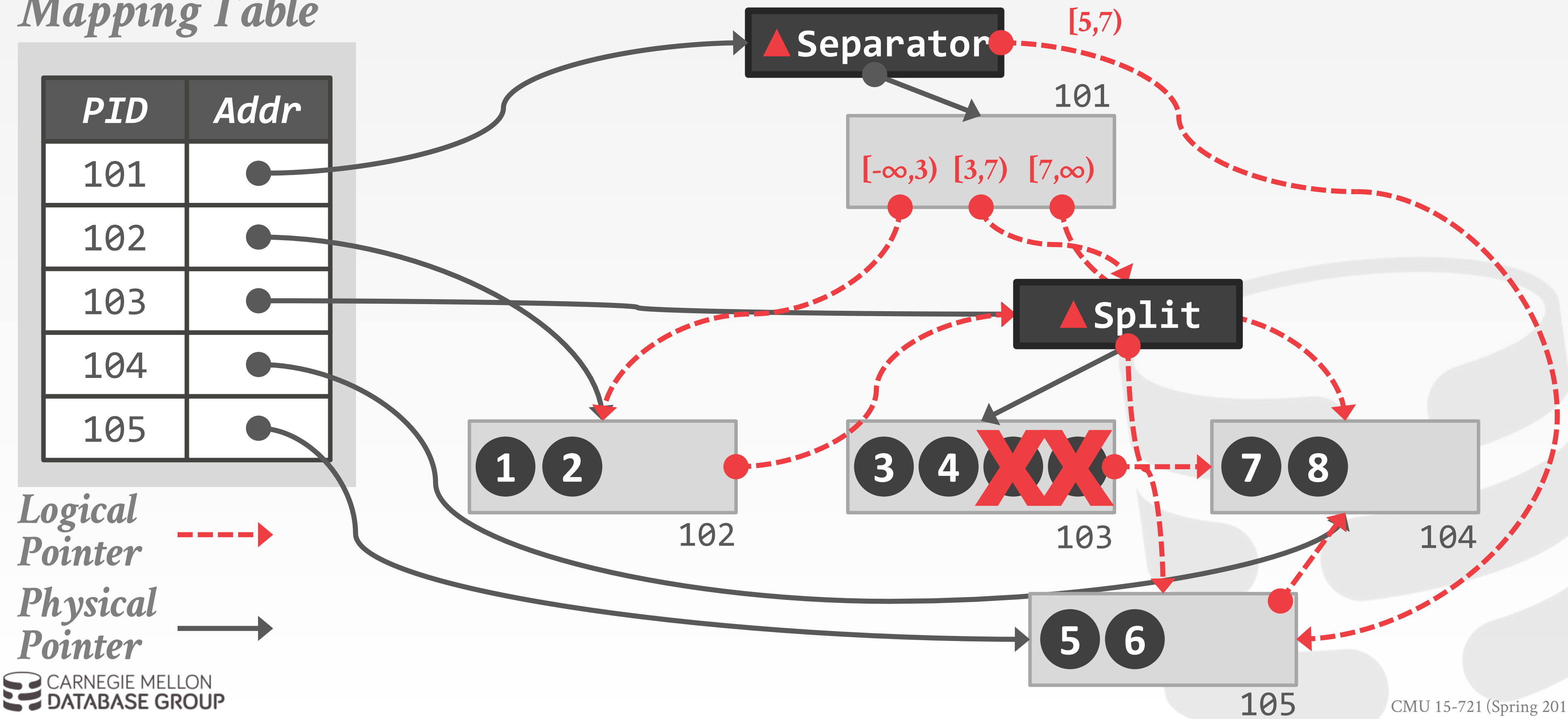
Mapping Table

PID	*Addr*
101	●
102	●
103	●
104	●
105	●

BW-TREE: STRUCTURE MODIFICATIONS

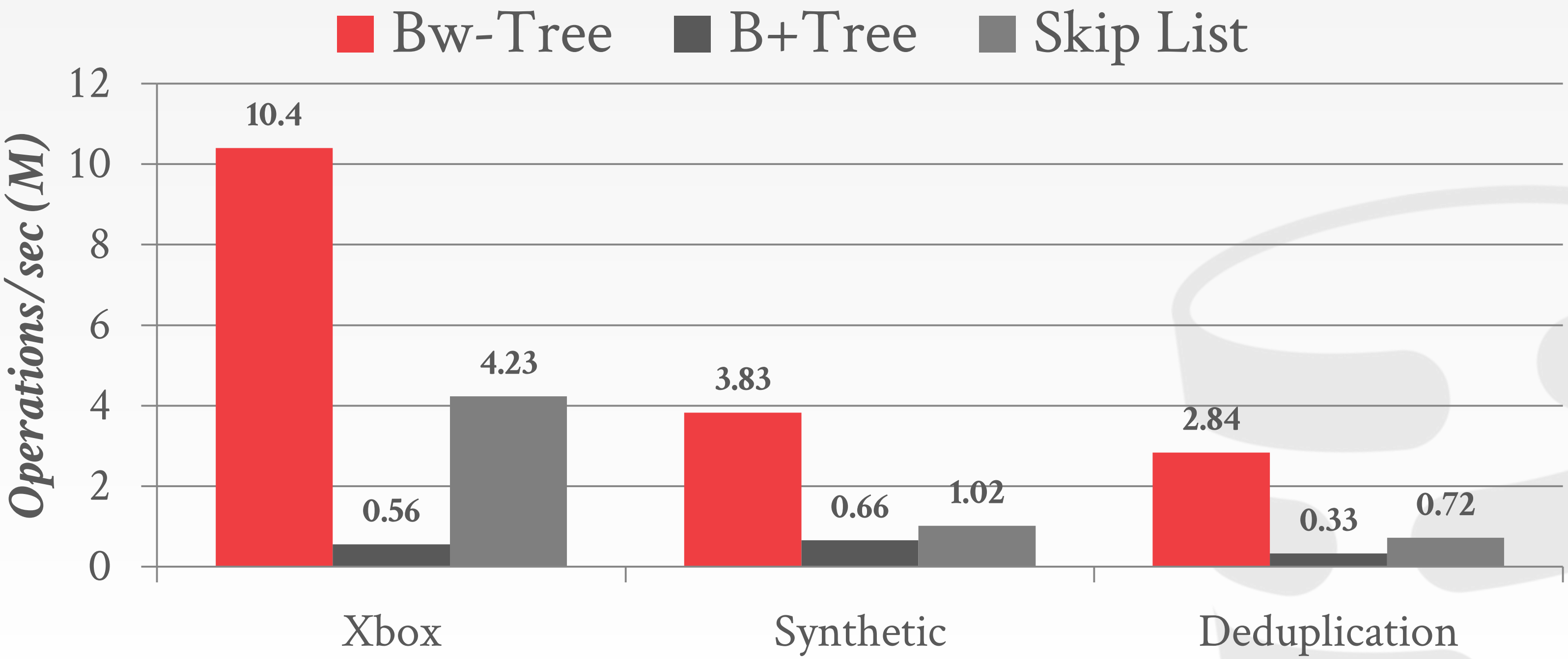
Mapping Table

PID	Addr
101	●
102	●
103	●
104	●
105	●



BW-TREE: PERFORMANCE

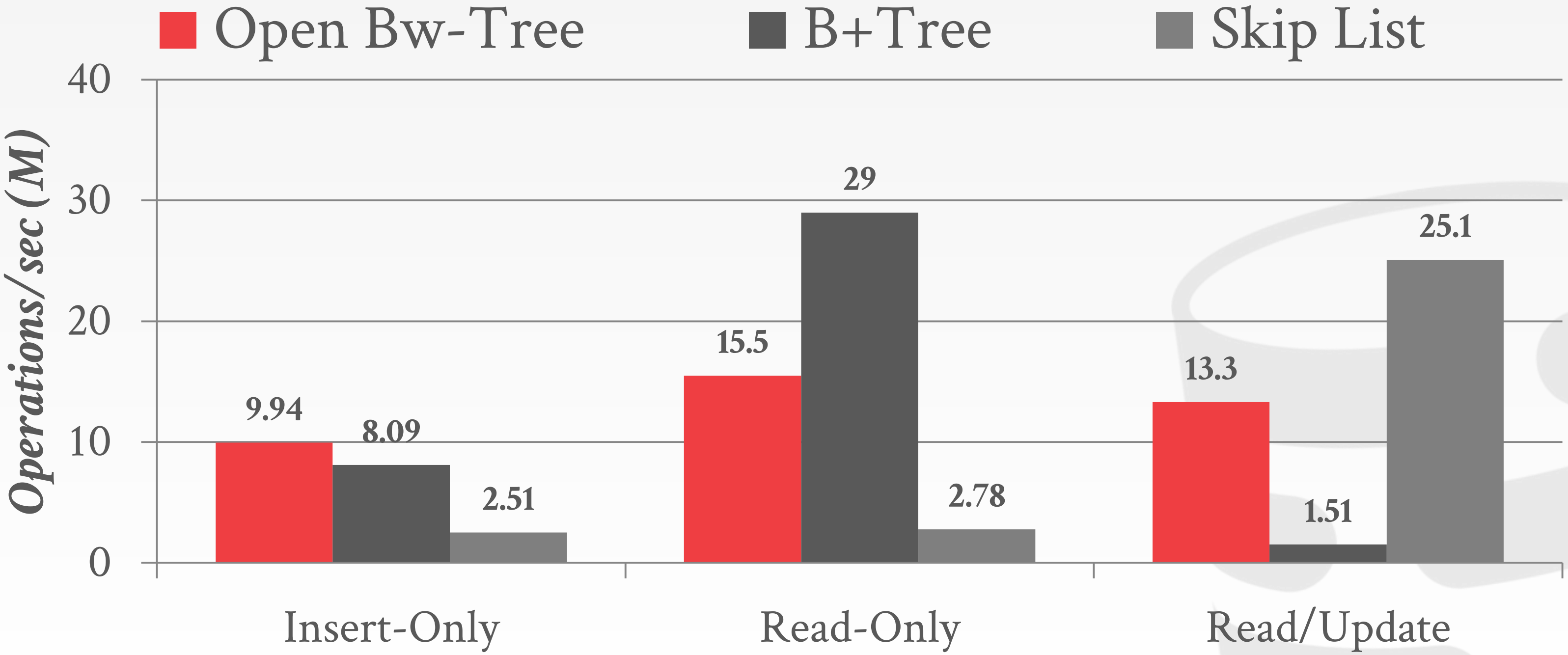
Processor: 1 socket, 4 cores w/ 2×HT



Source: Justin Levandoski

BW-TREE: PERFORMANCE

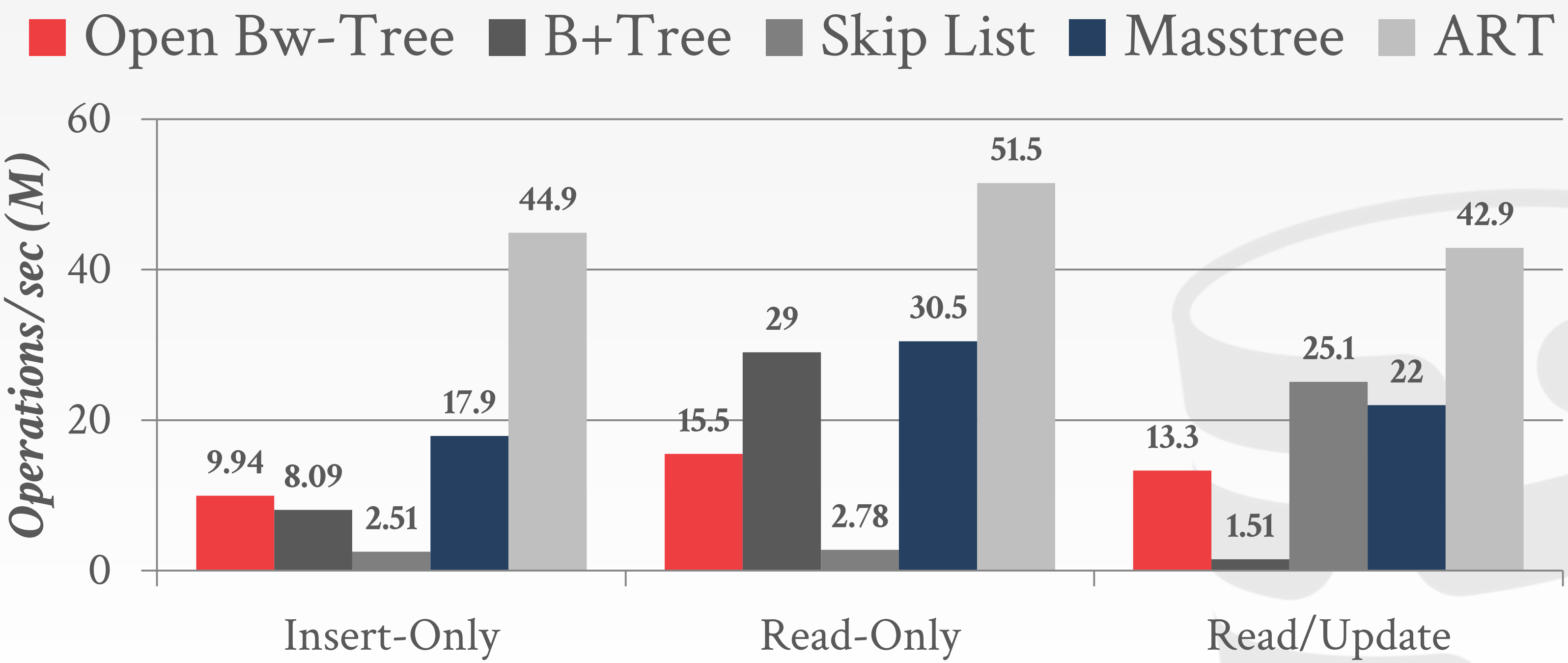
Processor: 1 socket, 10 cores w/ 2×HT
Workload: 50m Random Integer Keys (64-bit)



Source: Ziqi Wang

BW-TREE: PERFORMANCE

Processor: 1 socket, 10 cores w/ 2×HT
Workload: 50m Random Integer Keys (64-bit)



Source: Ziqi Wang

PARTING THOUGHTS

Managing a concurrent index looks a lot like managing a database.

Non-concurrent Skip List is easy to implement.

A Bw-Tree is hard to implement.

NEXT CLASS

Let's add latches back in our OLTP indexes!

Other implementation issues.

Crash course on performance testing.