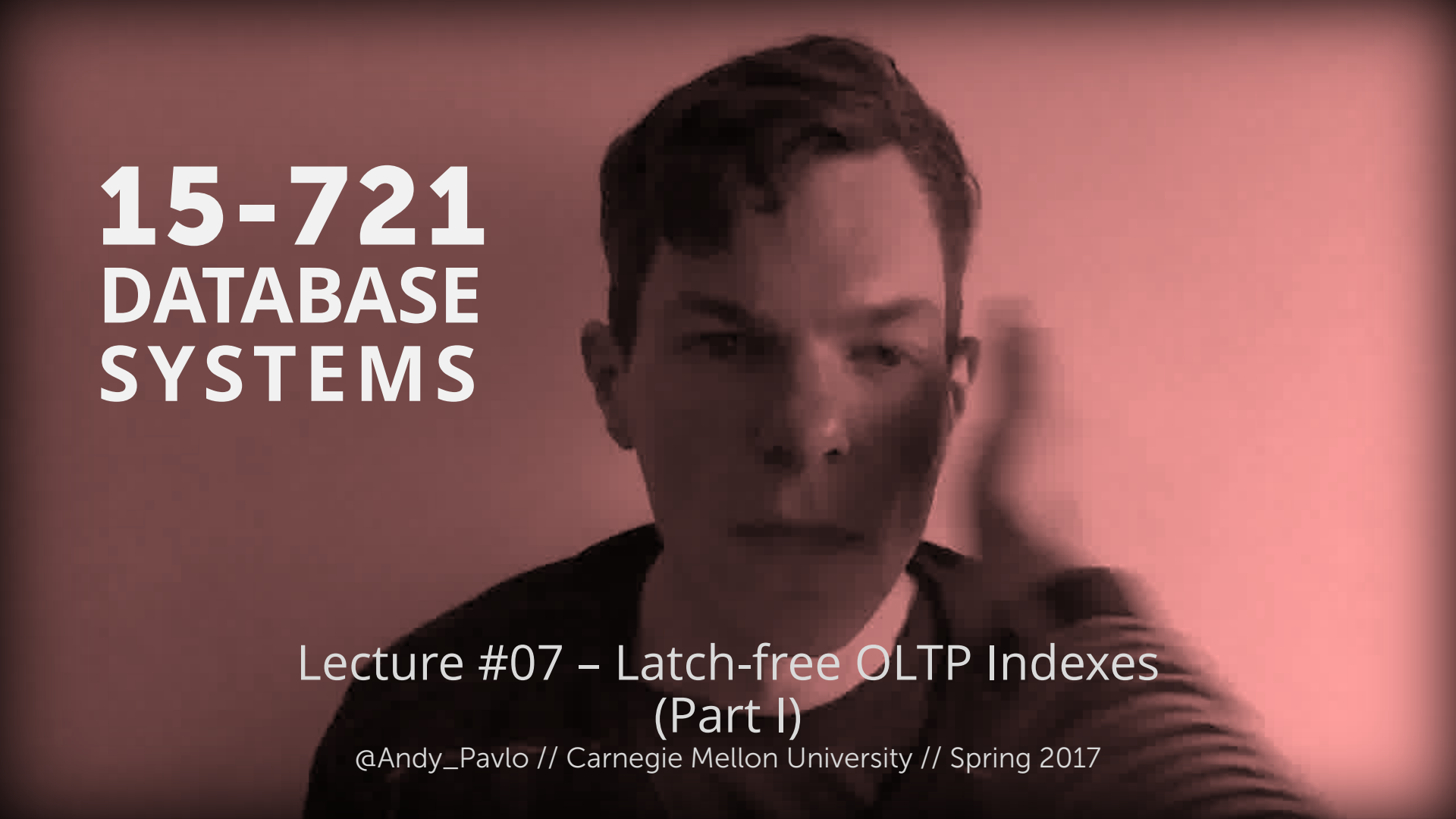


15-721 DATABASE SYSTEMS

Lecture #07 – Latch-free OLTP Indexes (Part I)

@Andy_Pavlo // Carnegie Mellon University // Spring 2017

ADMINISTRIVIA

Peloton master branch has been updated to provide cleaner test cases.

→ There is now a separate file for Skip List tests.

→ 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-Trees

Skip Lists

Index Implementation Issues

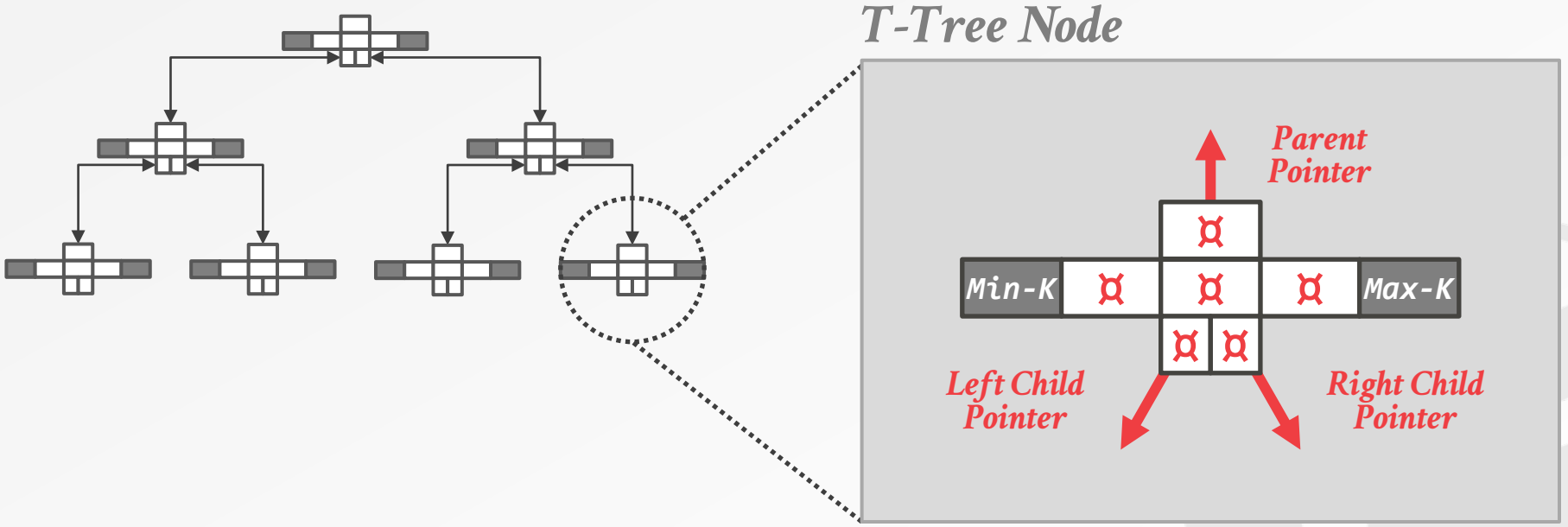
T-TREES

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

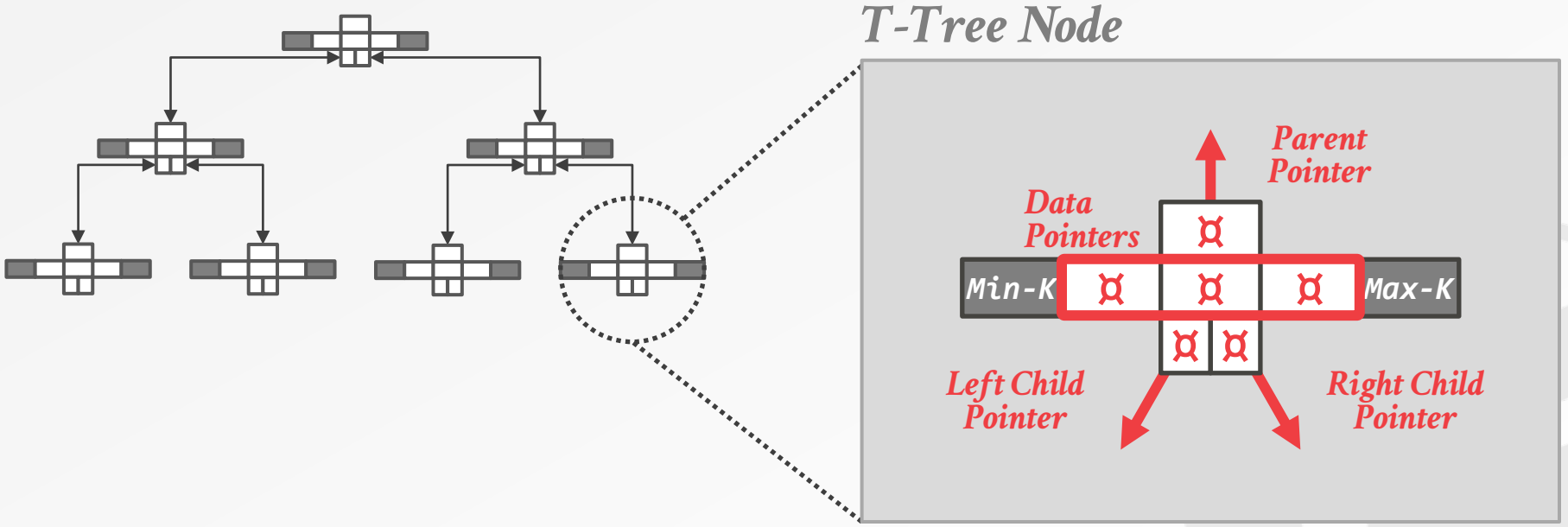
Proposed in 1986 from Univ. of Wisconsin
Used in TimesTen and other early in-memory DBMSs during the 1990s.

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

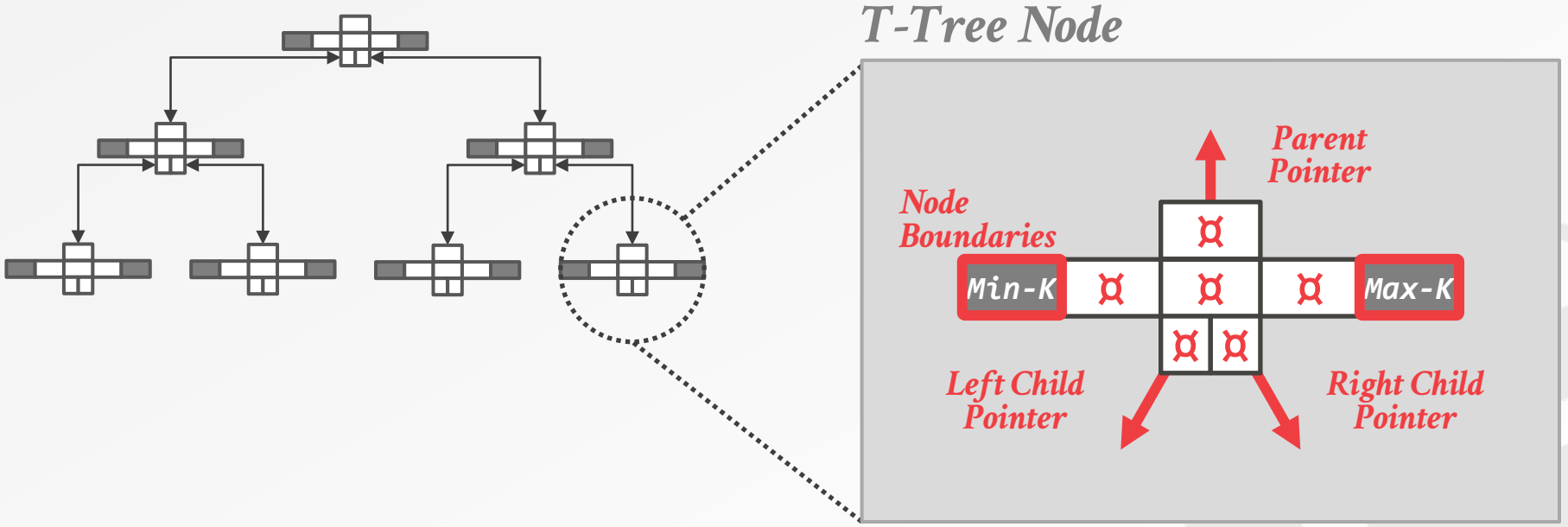
T-TREES


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

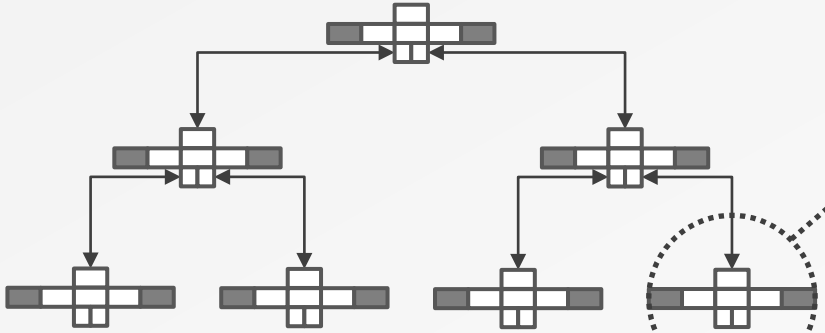
T-TREES


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

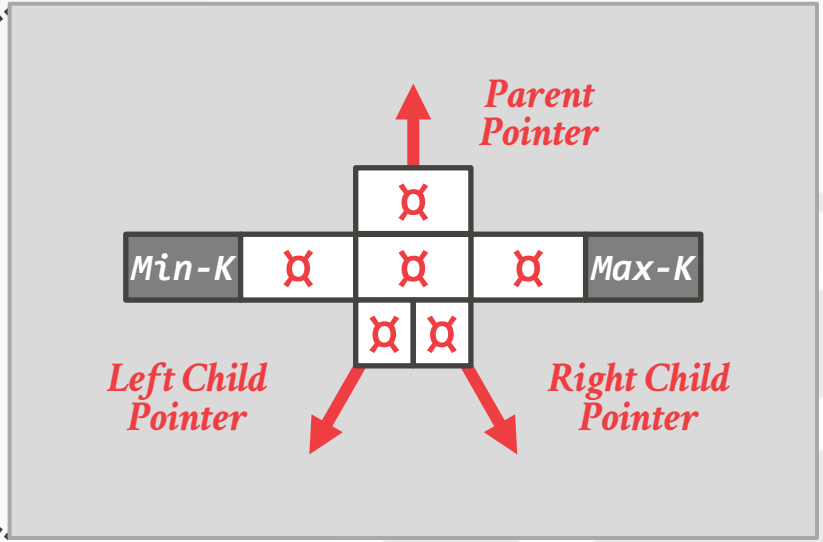
T-TREES


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

T-TREES

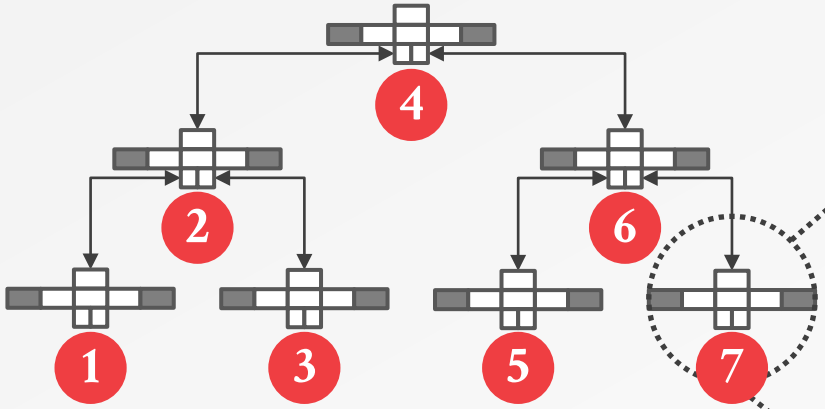


T-Tree Node

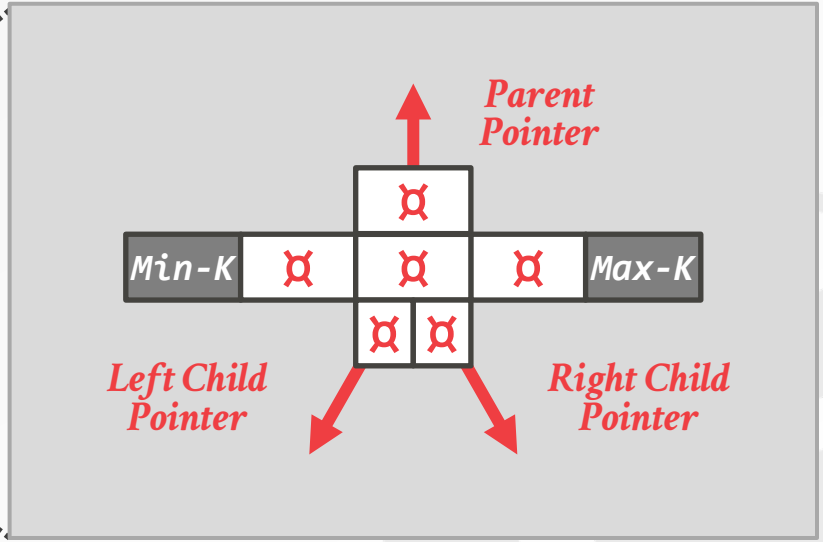


Key Space (Low→High)

T-TREES

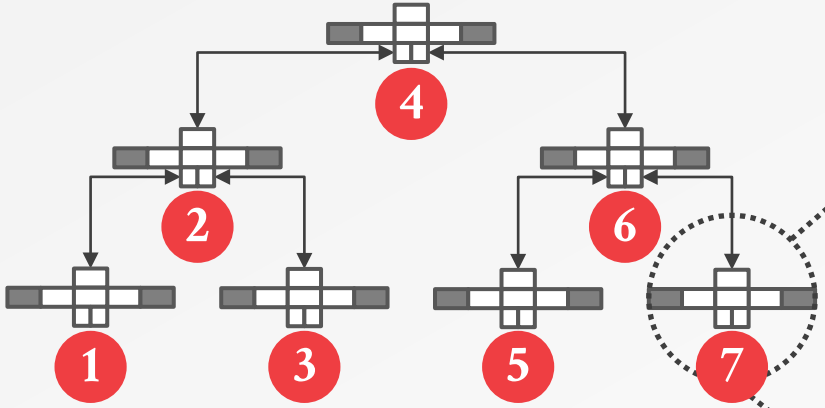


T-Tree Node

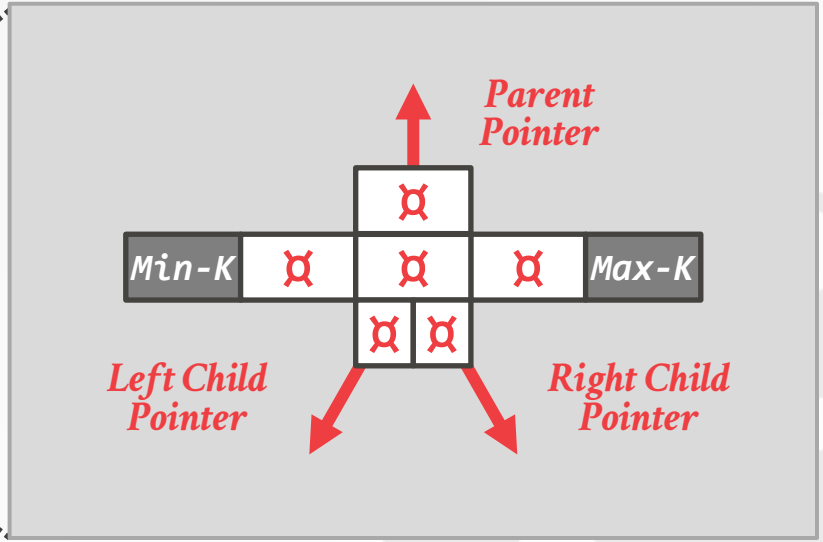


Key Space (Low→High)

T-TREES



T-Tree Node



Key Space (Low→High)

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

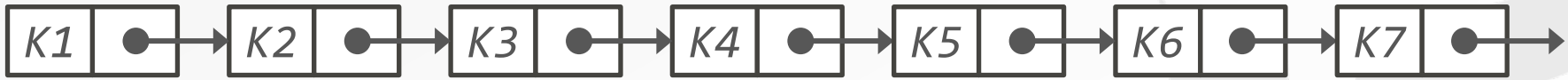
→ Difficult to rebalance.
→ Difficult to implement safe concurrent access.
→ Have to chase pointers when scanning range or performing binary search inside of a node.

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)

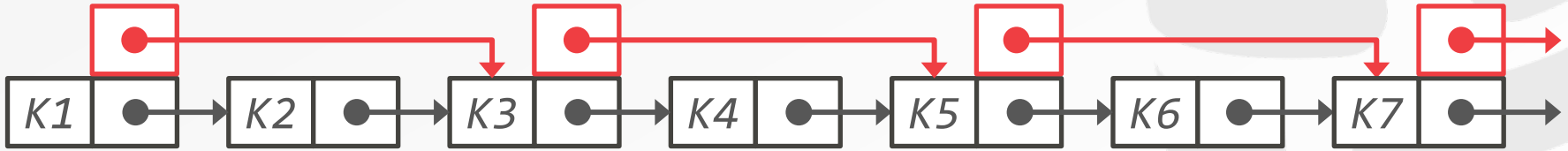


OBSERVATION

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)

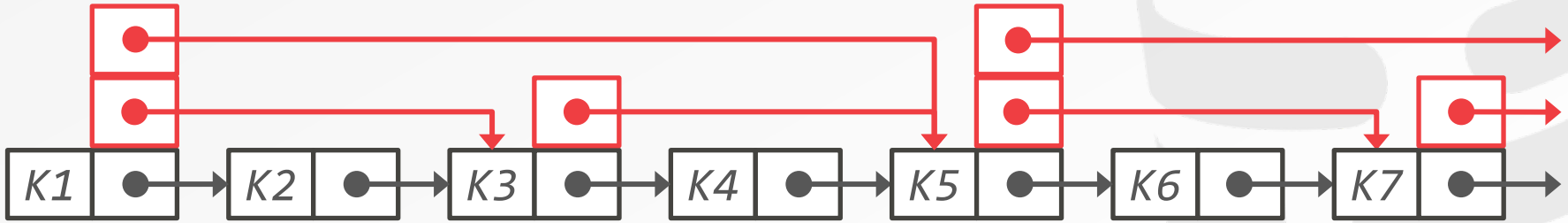


OBSERVATION

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.

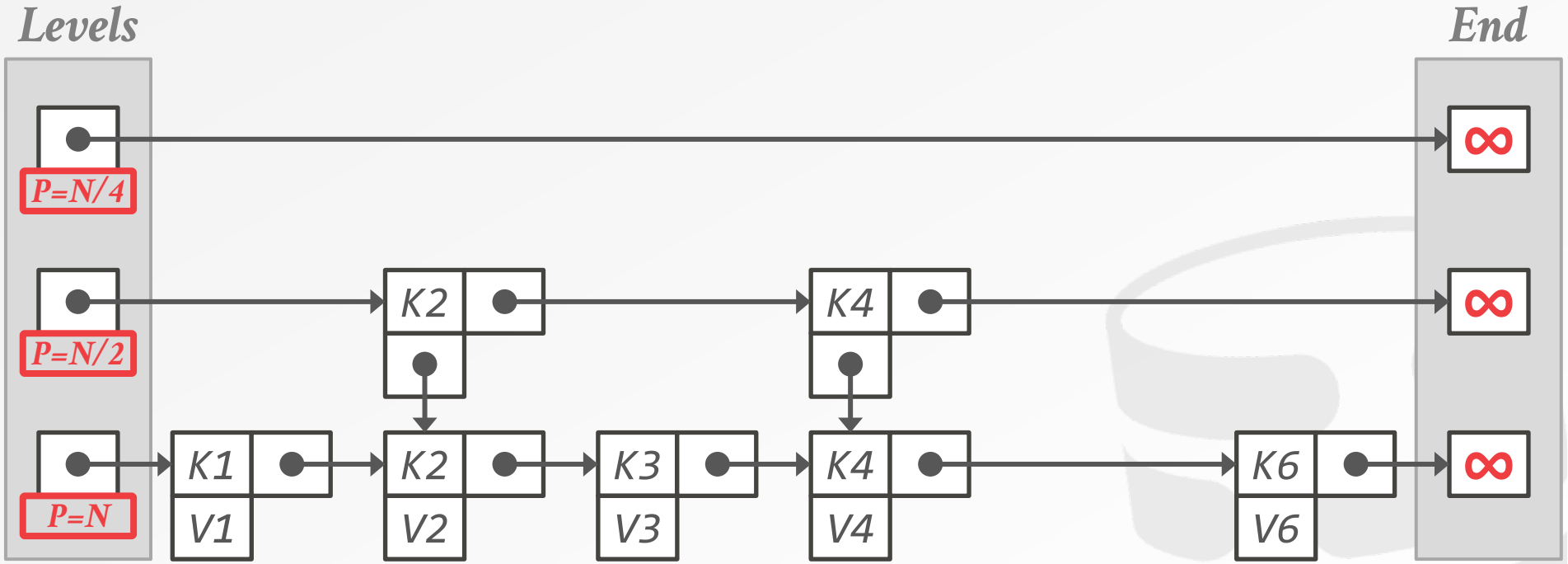
SKIP LISTS: A PROBABILISTIC ALTERNATIVE TO BALANCED TREES
CACM Volume 33 Issue 6 1990

SKIP LISTS

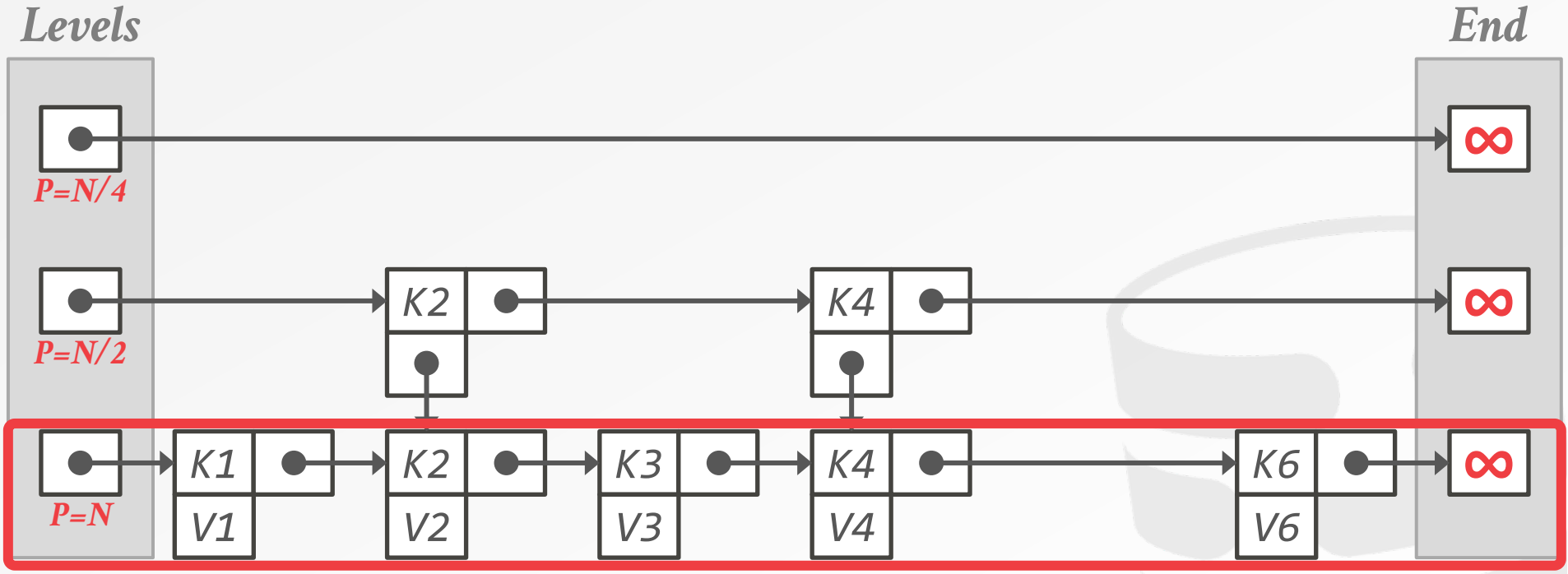
A collection of lists at different levels
→ Lowest level is a sorted, singly linked list of all keys
→ 2nd level links every other key
→ 3rd level links every fourth key
→ In general, a level has half the keys of one below it

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.

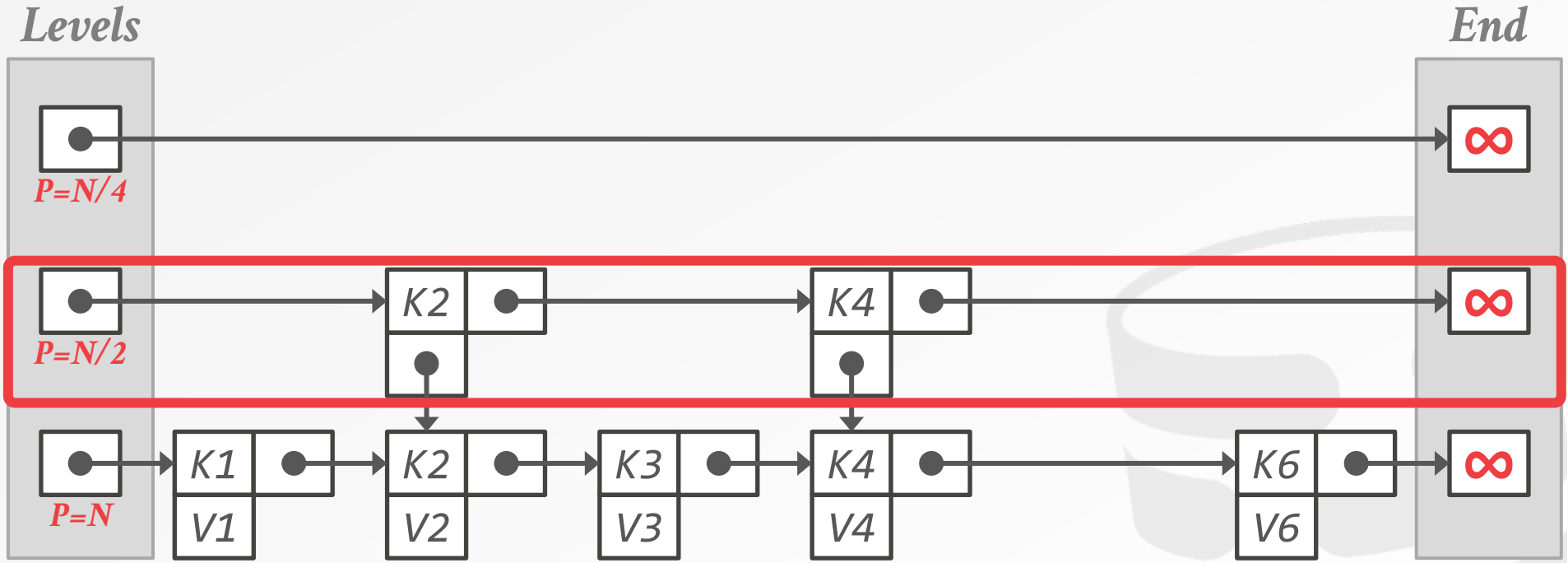
SKIP LISTS: EXAMPLE



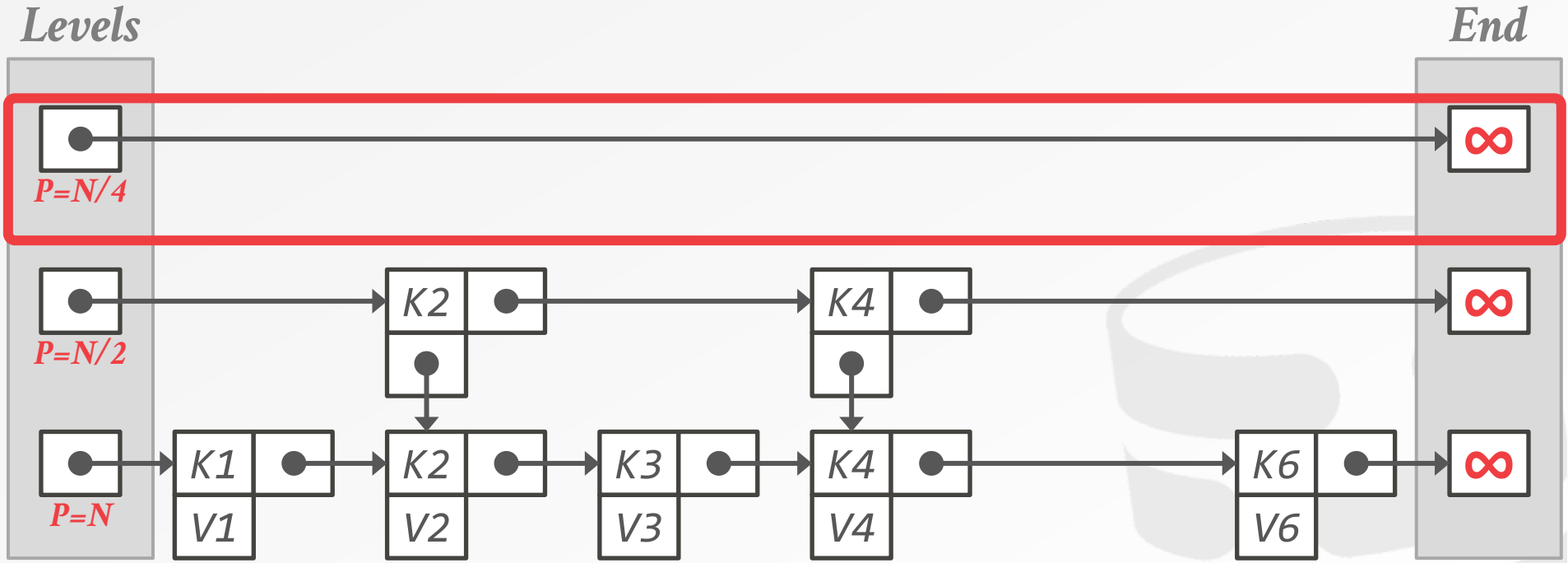
SKIP LISTS: EXAMPLE



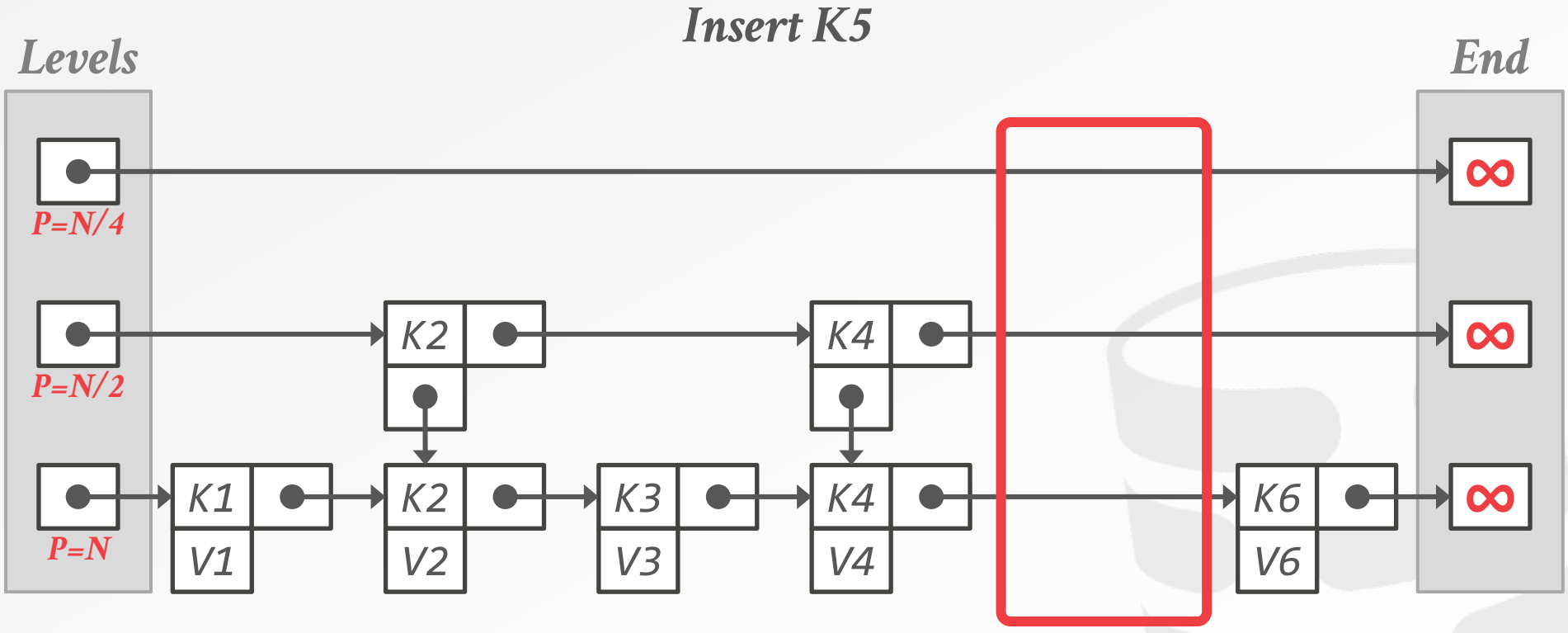
SKIP LISTS: EXAMPLE



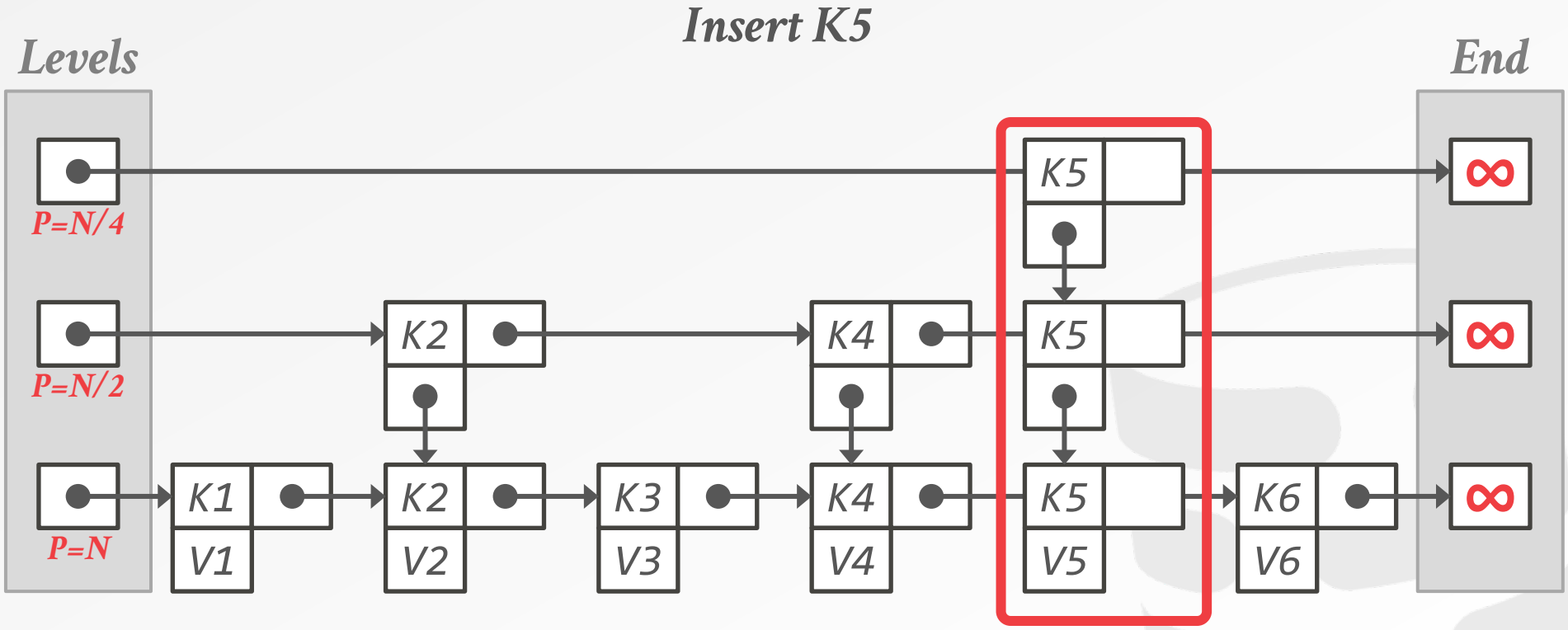
SKIP LISTS: EXAMPLE



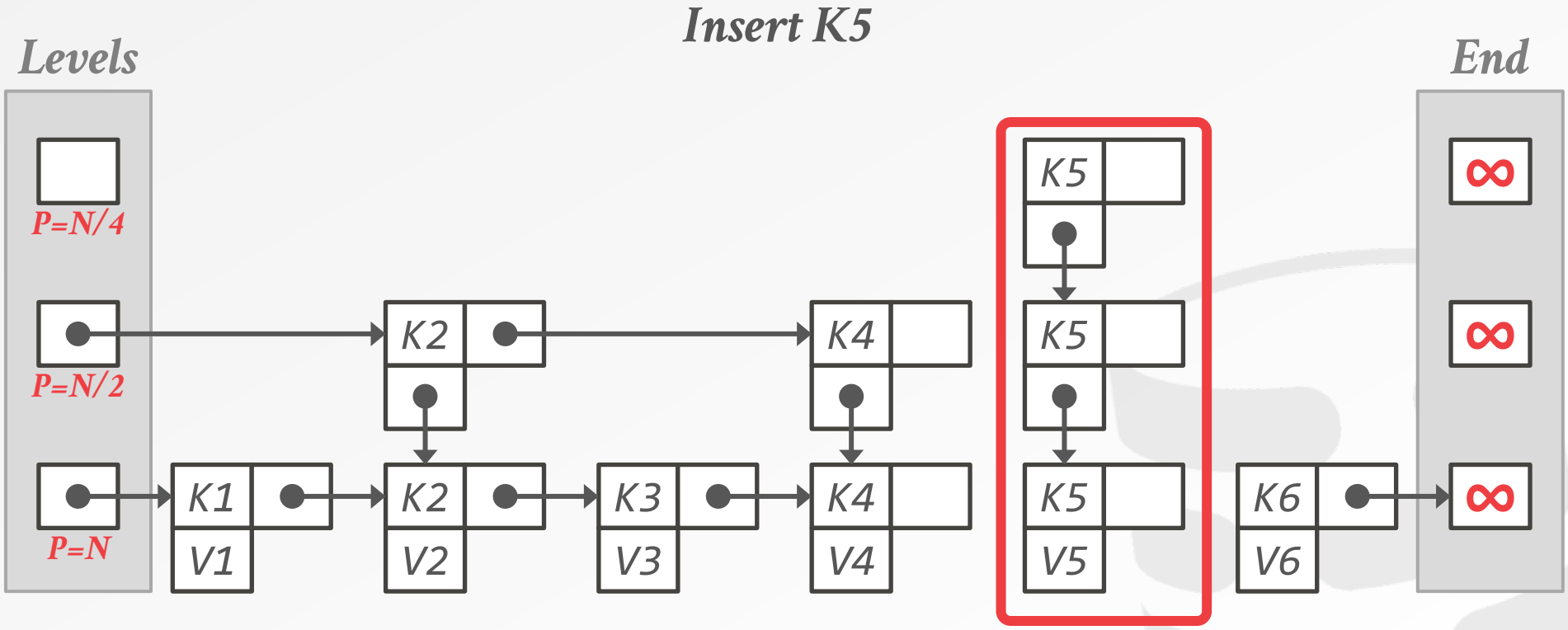
SKIP LISTS: INSERT



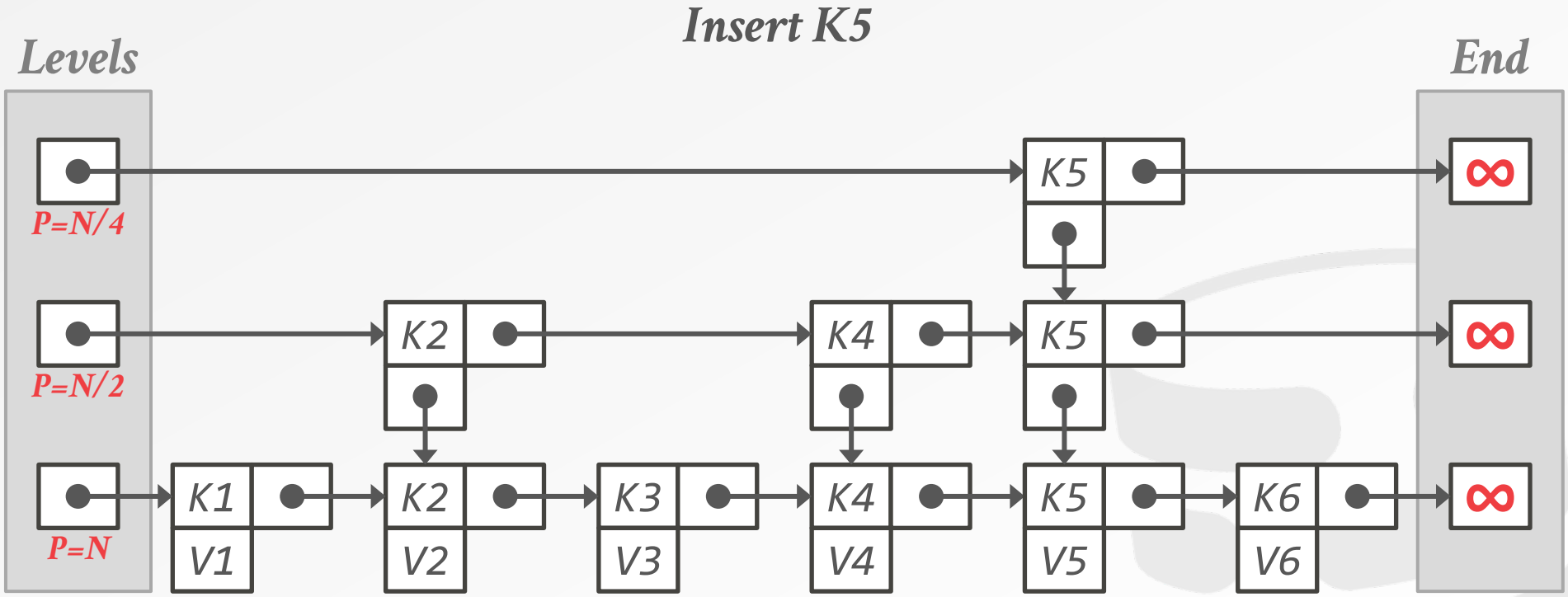
SKIP LISTS: INSERT



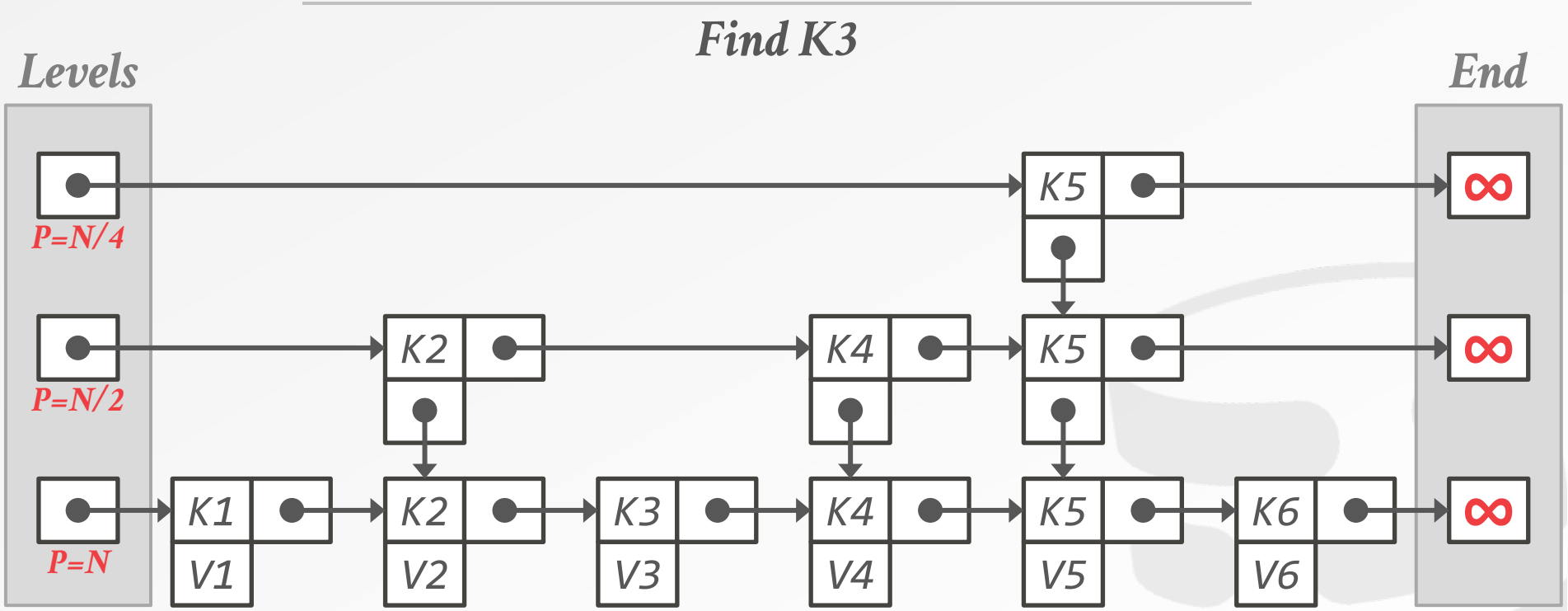
SKIP LISTS: INSERT



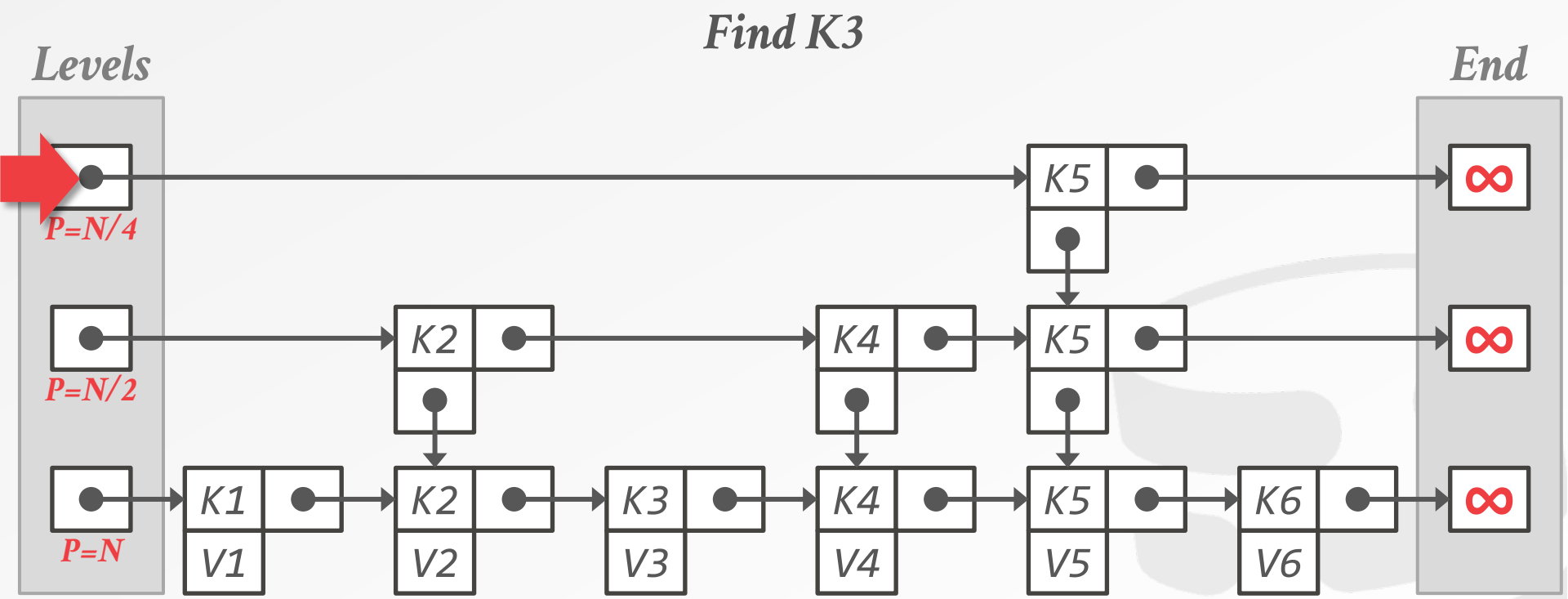
SKIP LISTS: INSERT



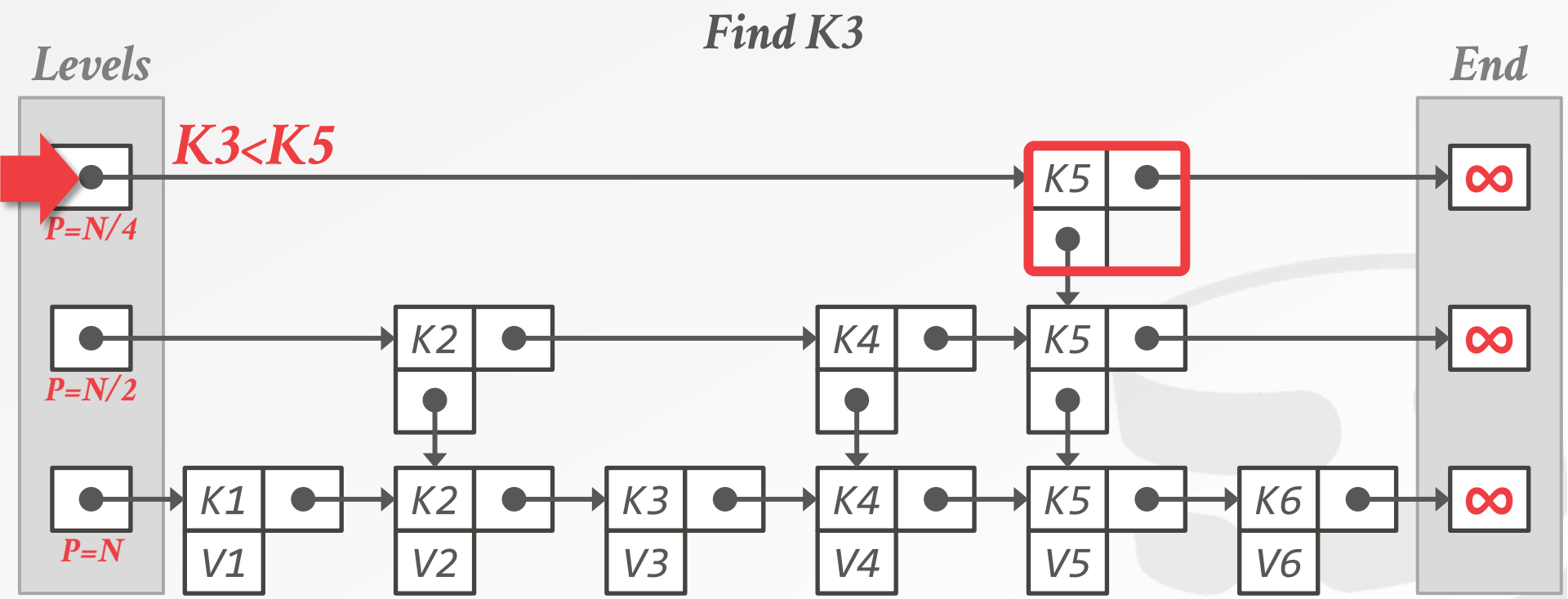
SKIP LISTS: SEARCH



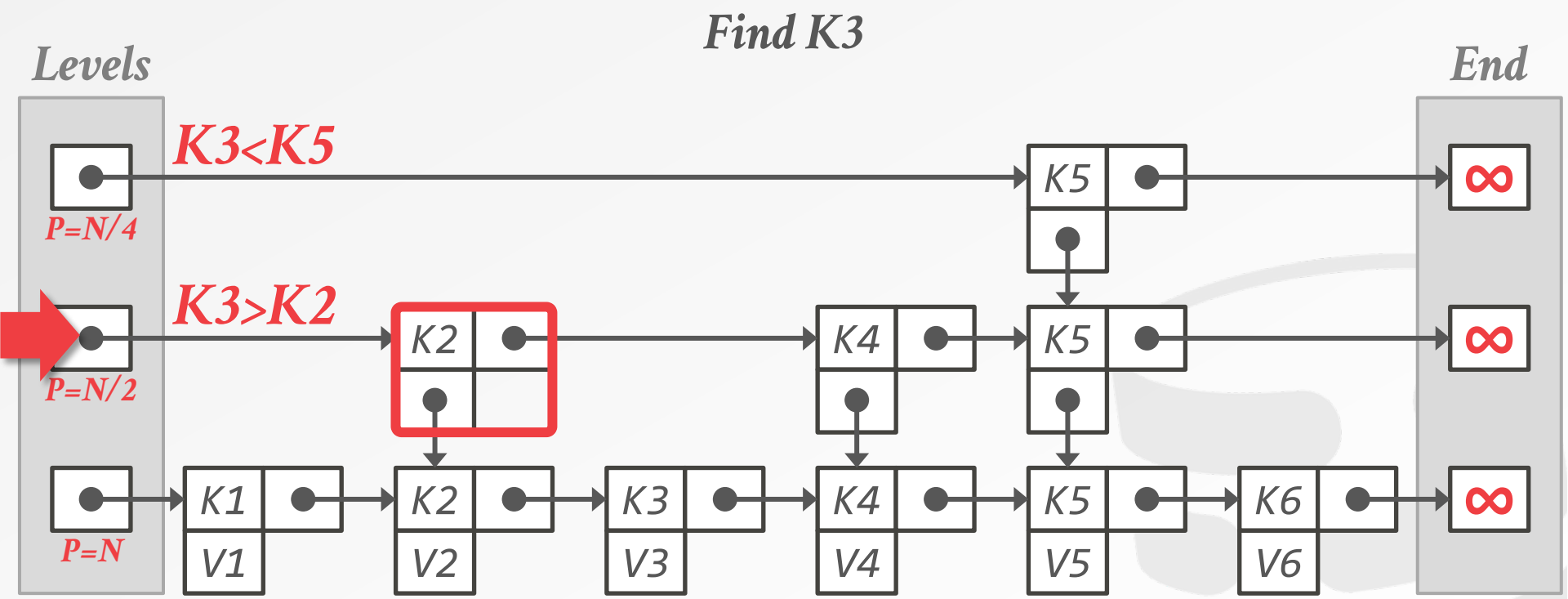
SKIP LISTS: SEARCH



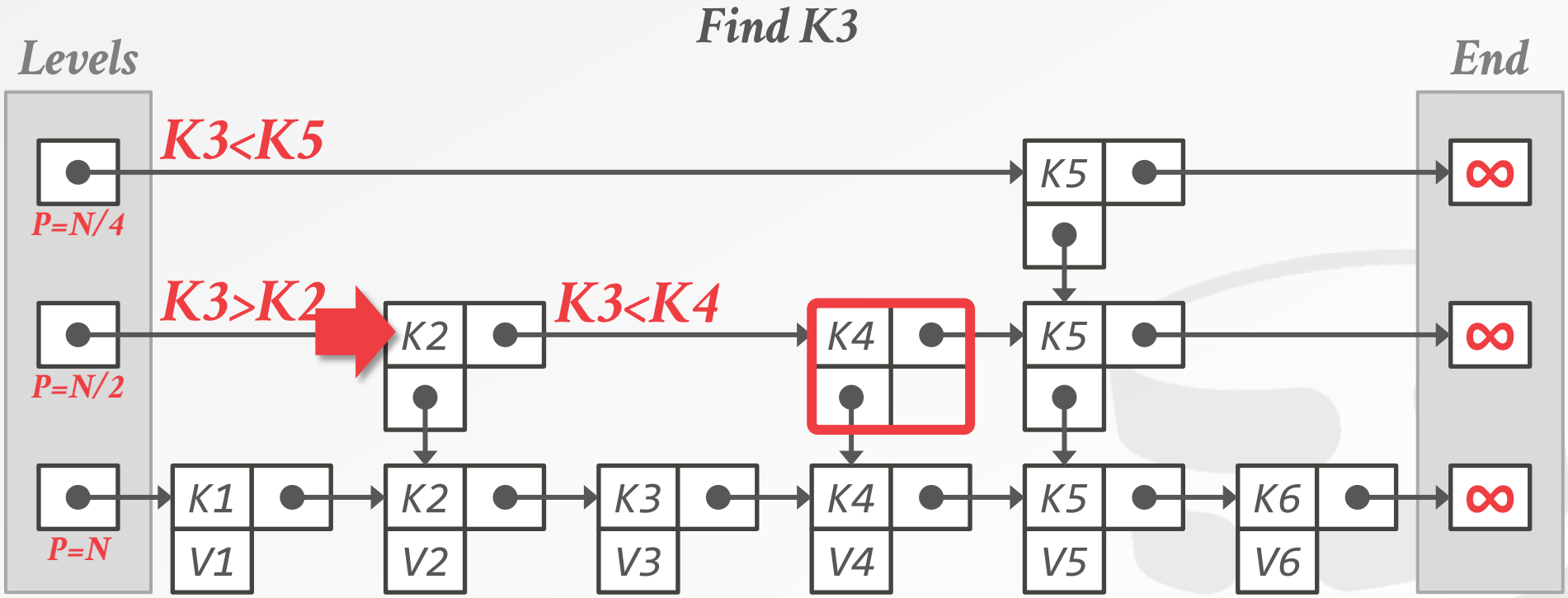
SKIP LISTS: SEARCH



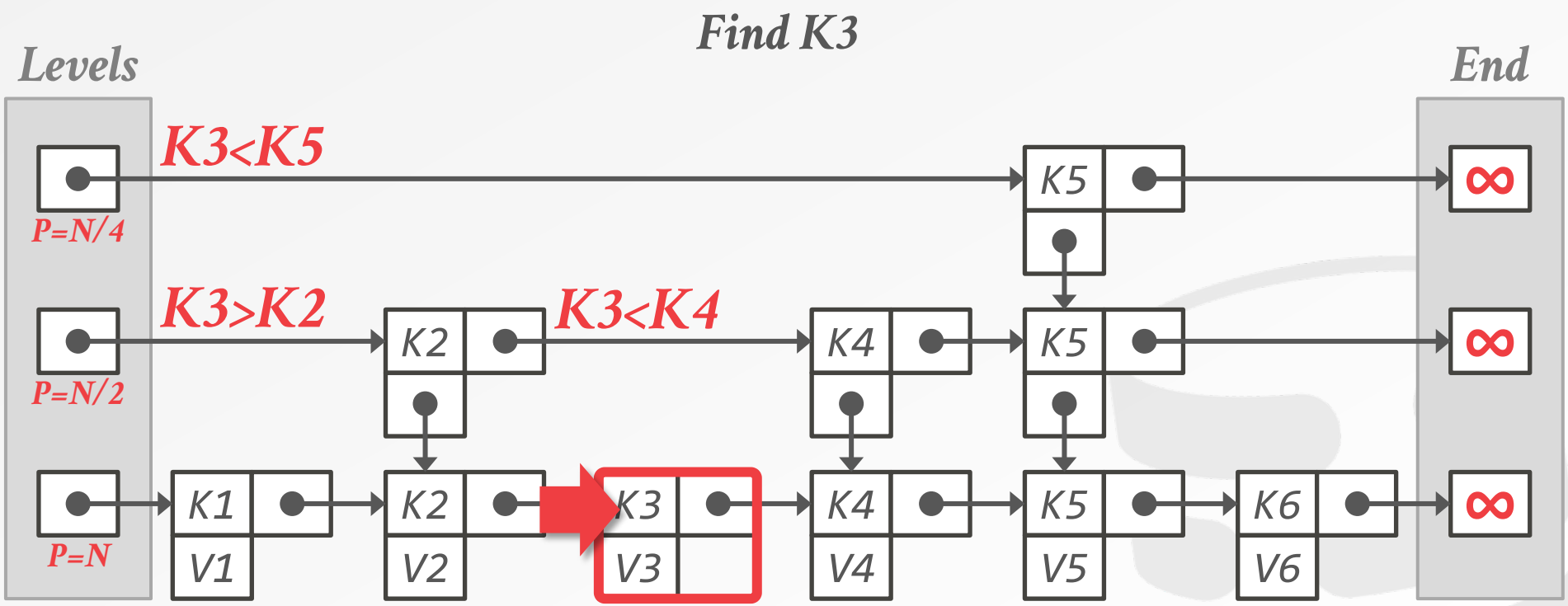
SKIP LISTS: SEARCH



SKIP LISTS: SEARCH



SKIP LISTS: SEARCH



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.

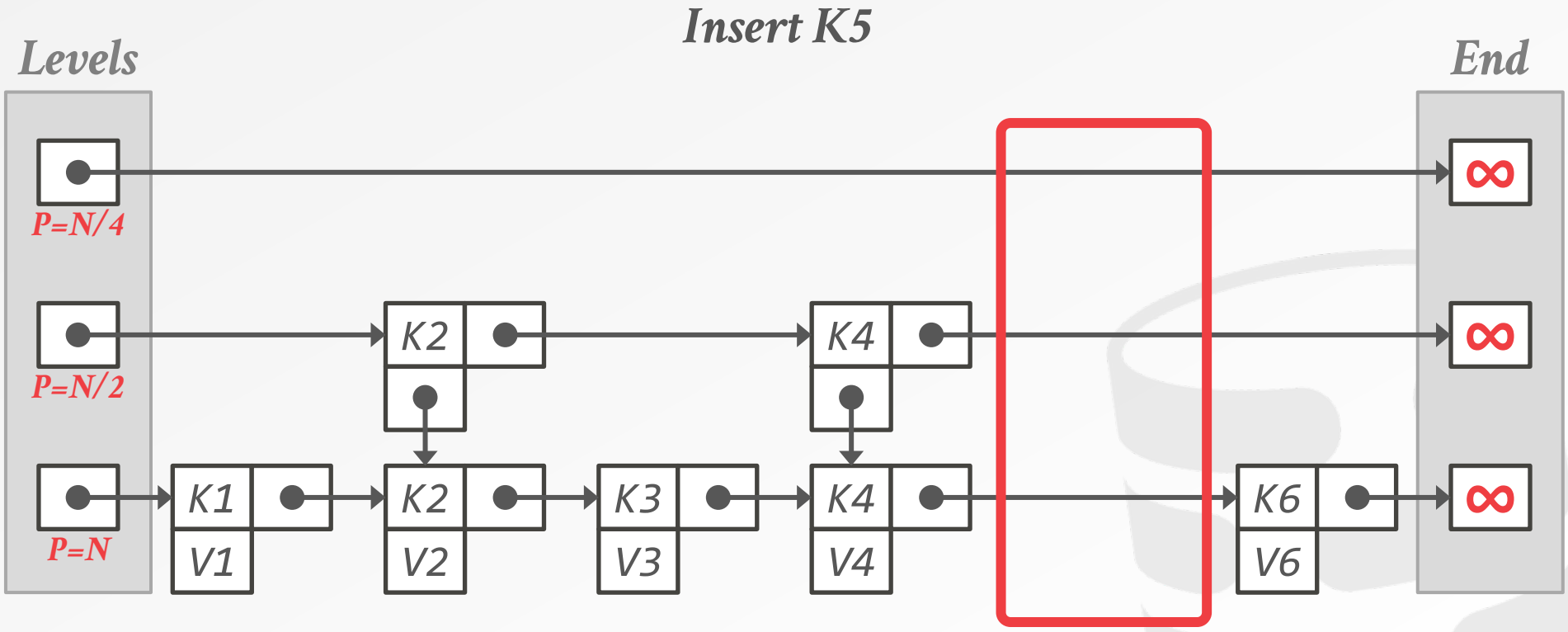
CONCURRENT SKIP LIST

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

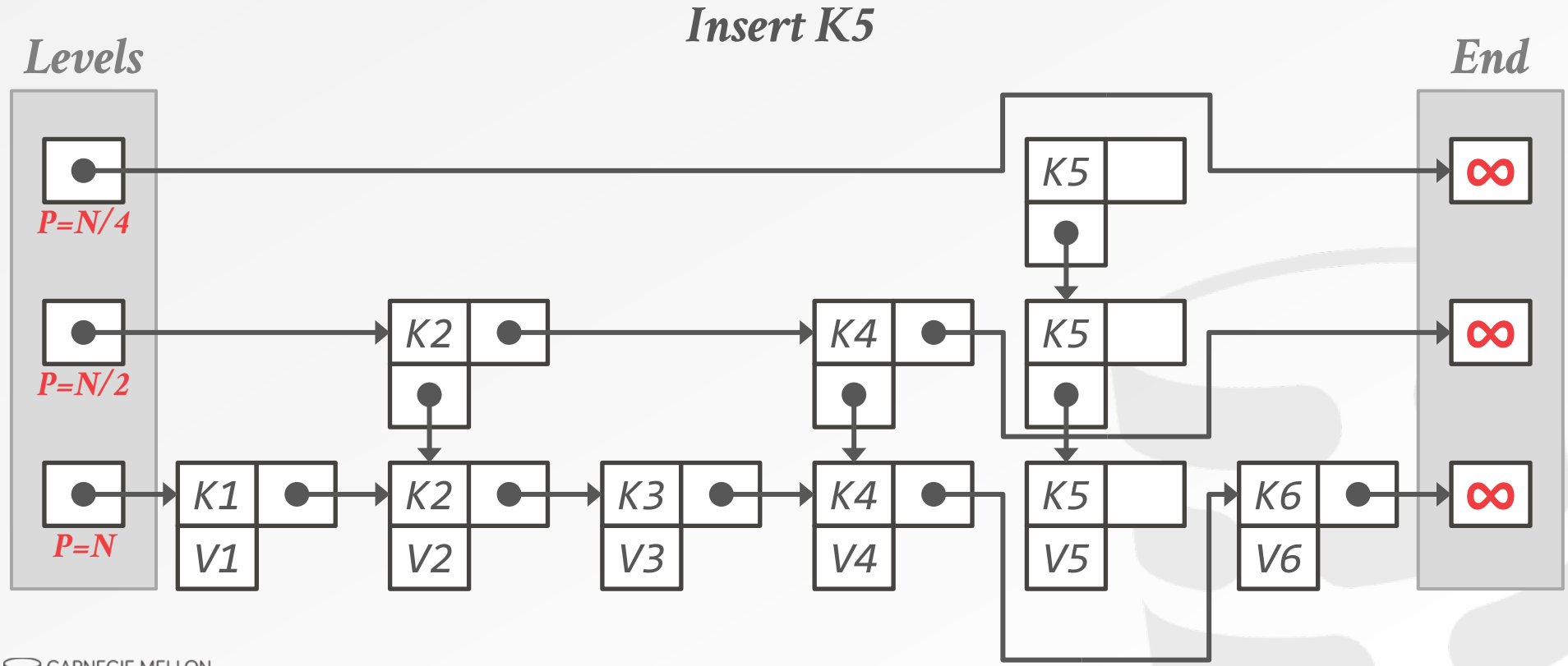
→ Only support linking in one direction

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

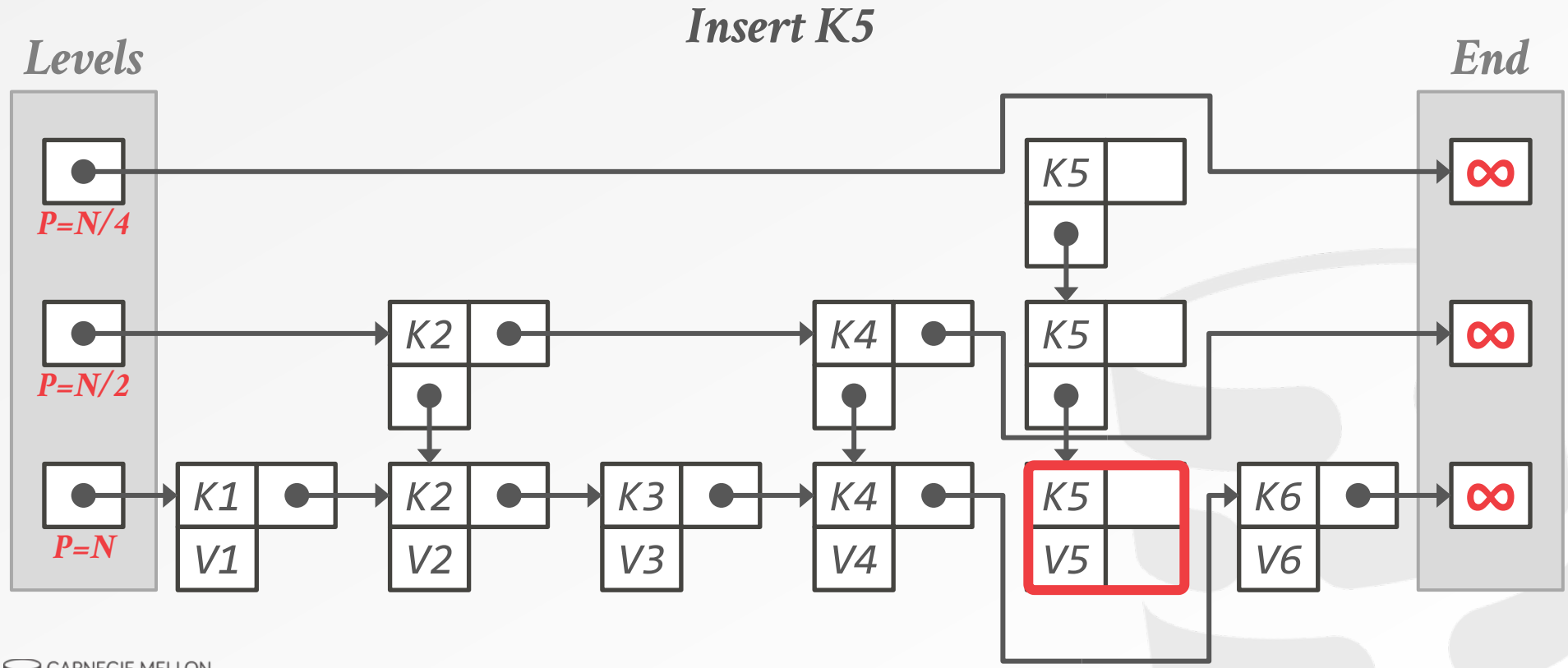
SKIP LISTS: INSERT



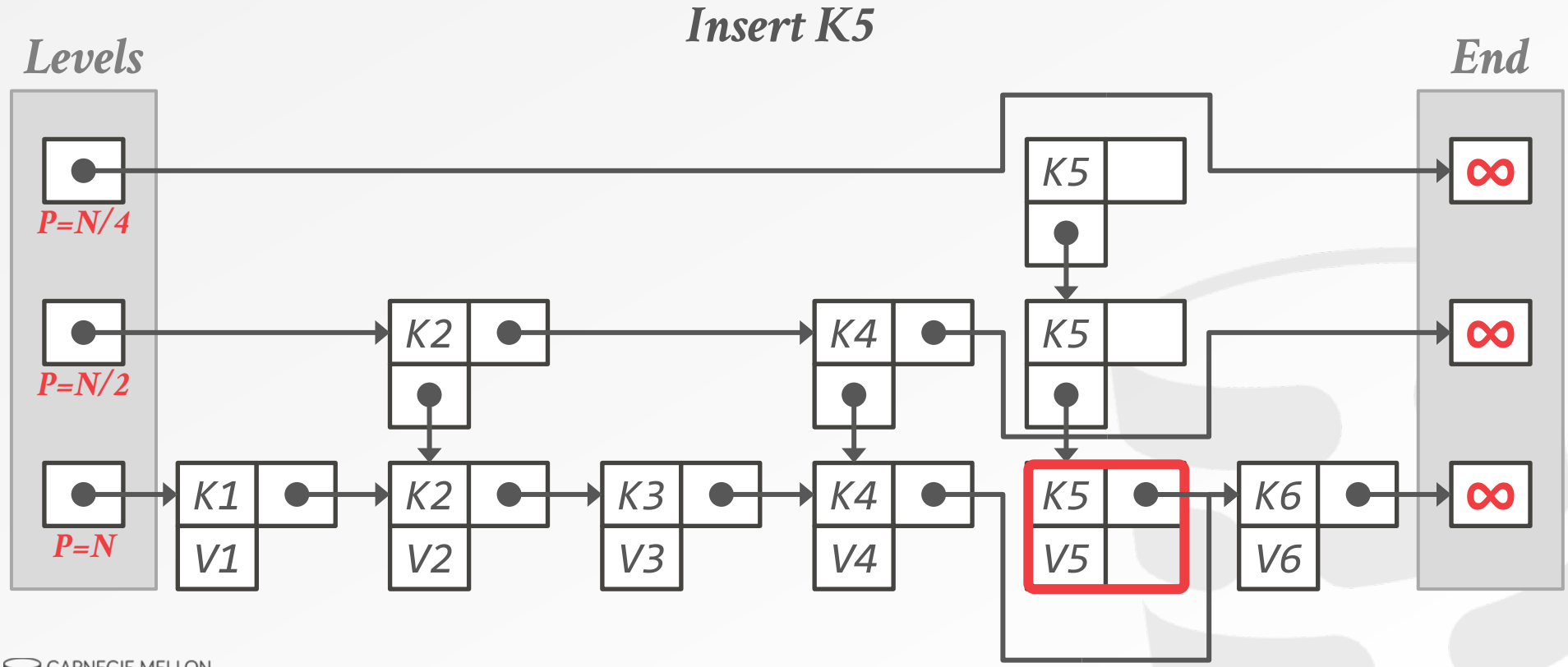
SKIP LISTS: INSERT



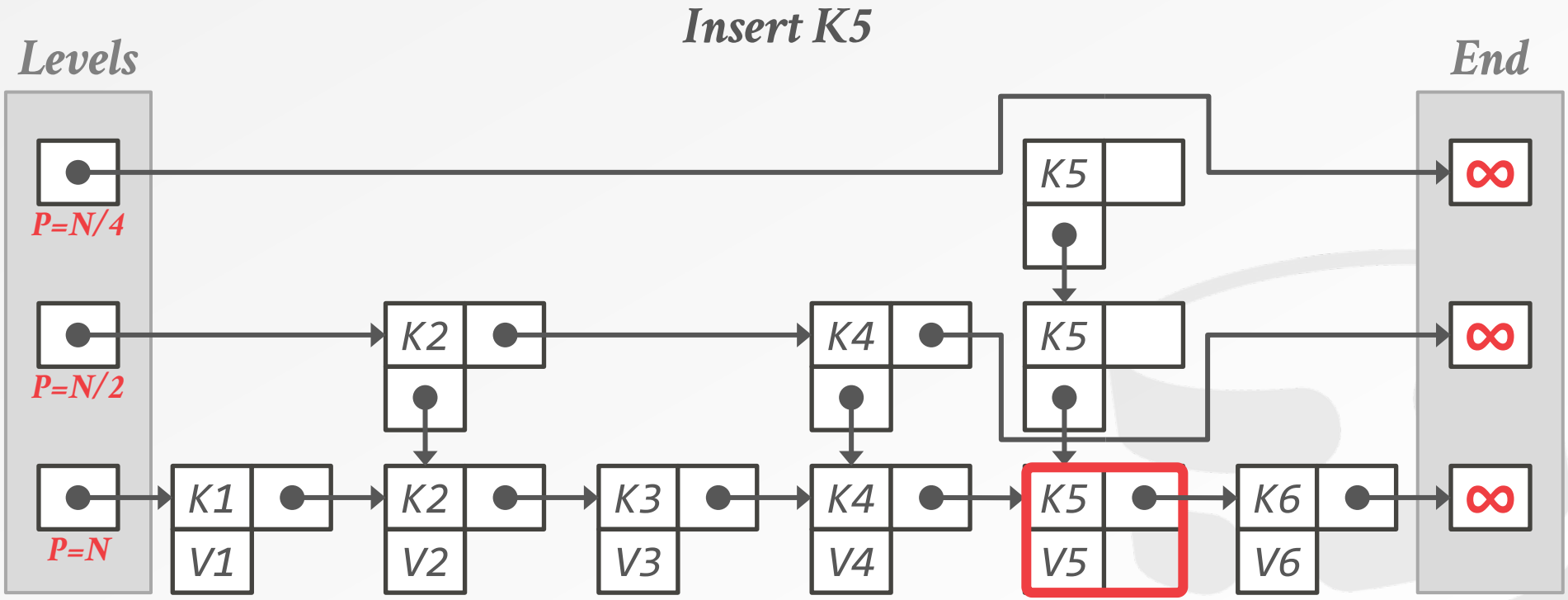
SKIP LISTS: INSERT



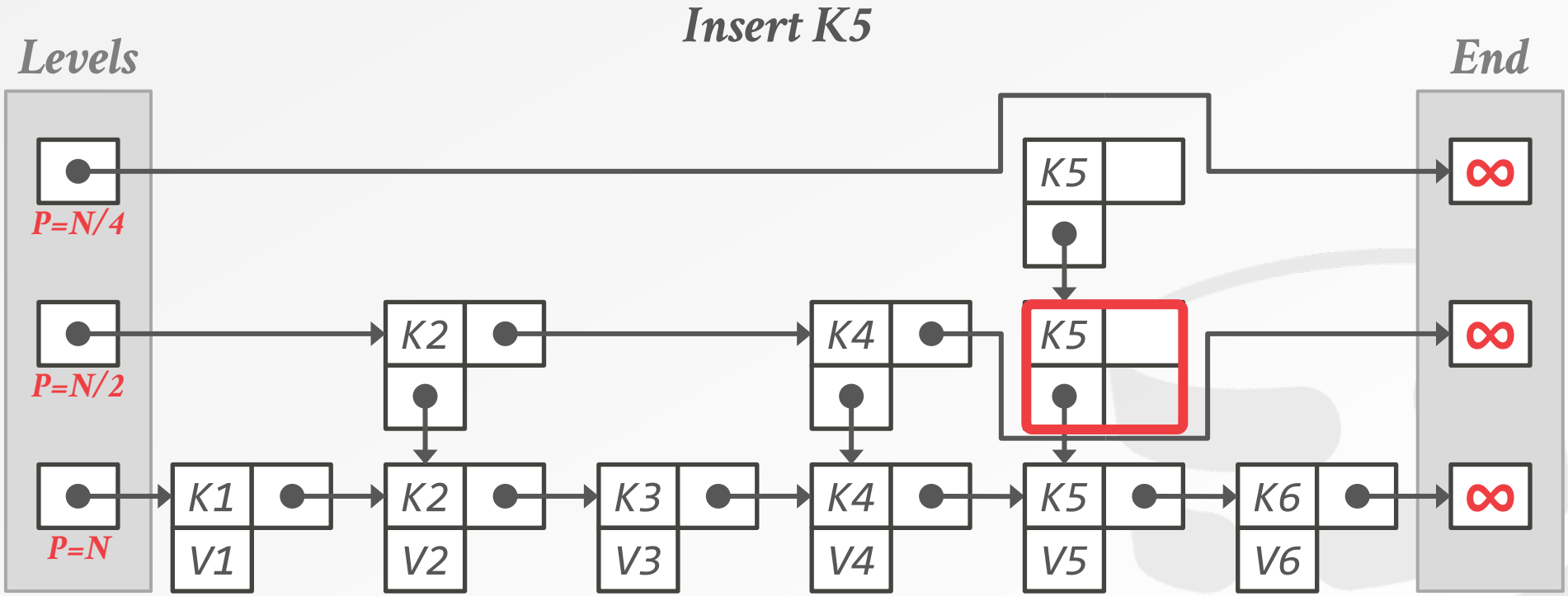
SKIP LISTS: INSERT



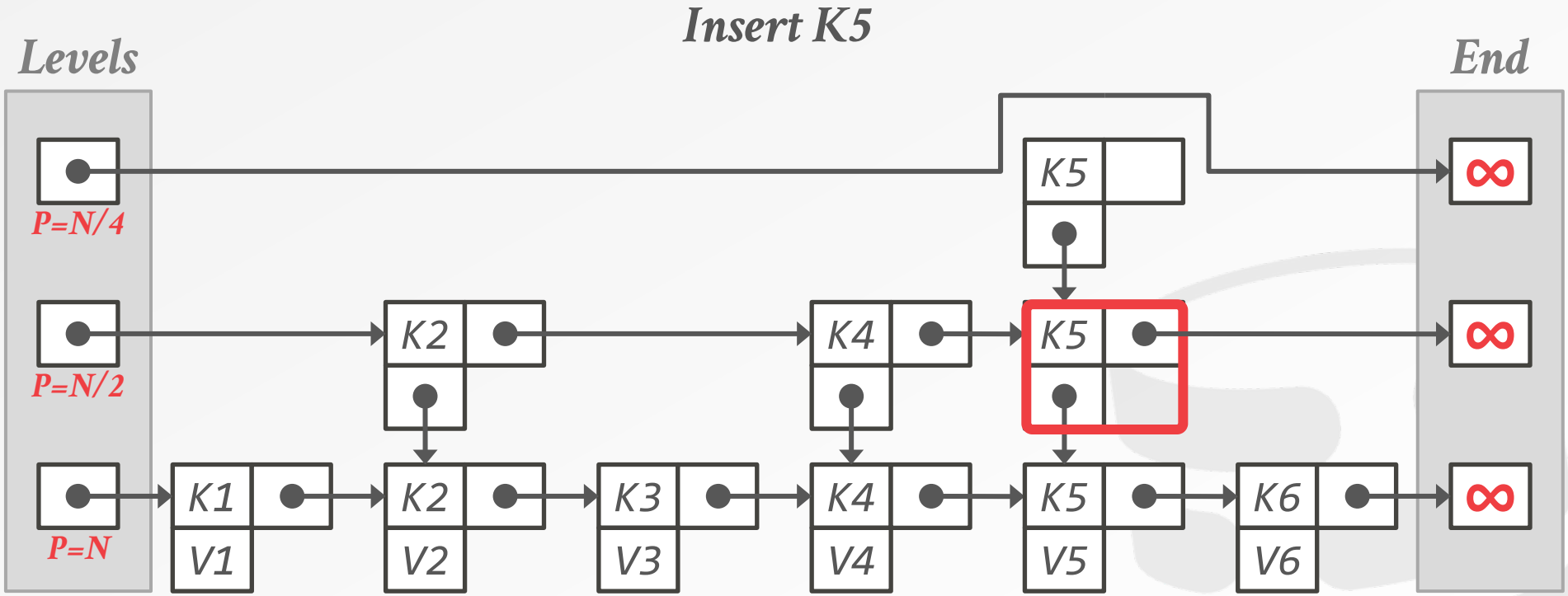
SKIP LISTS: INSERT



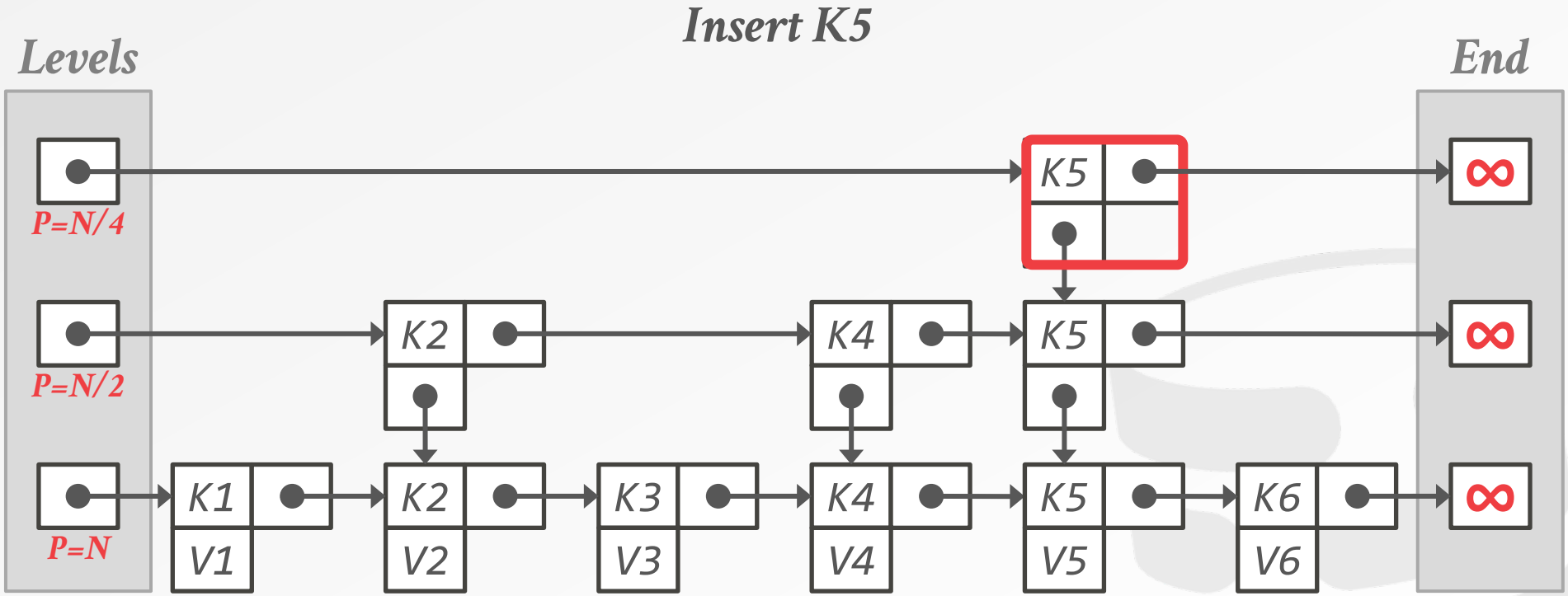
SKIP LISTS: INSERT



SKIP LISTS: INSERT



SKIP LISTS: INSERT

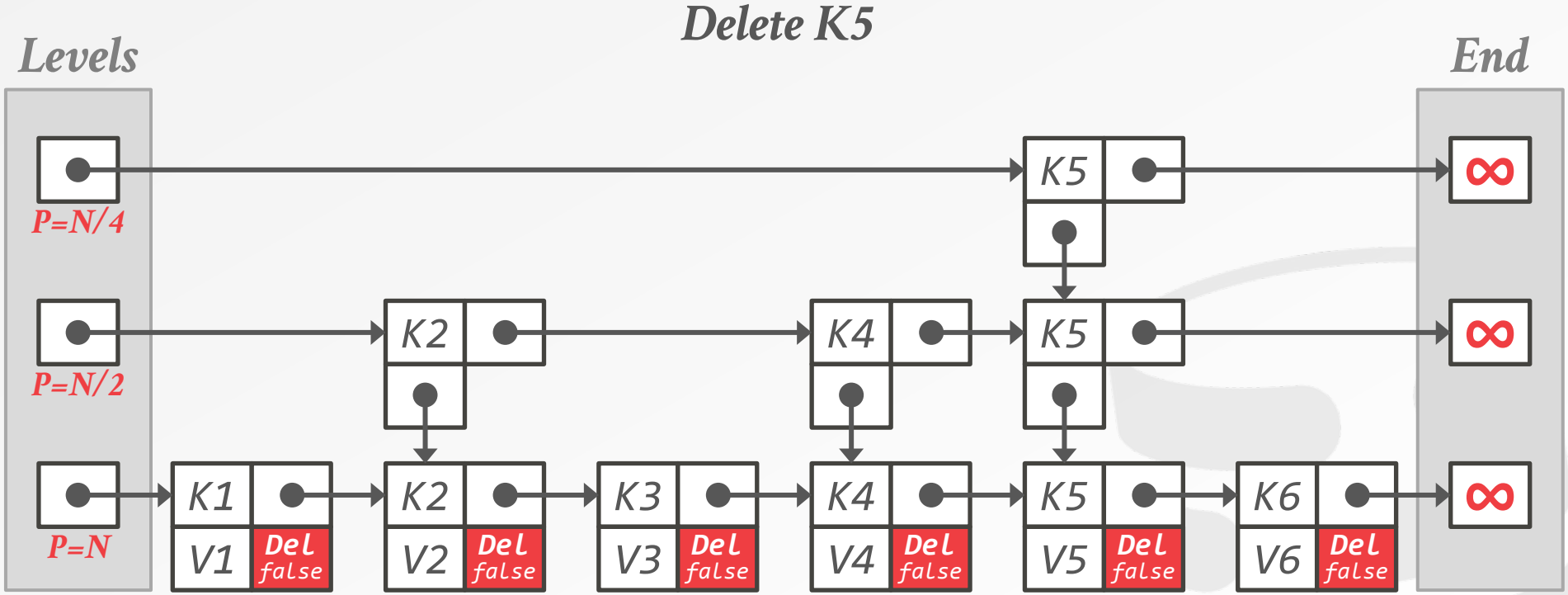


SKIP LISTS: DELETE

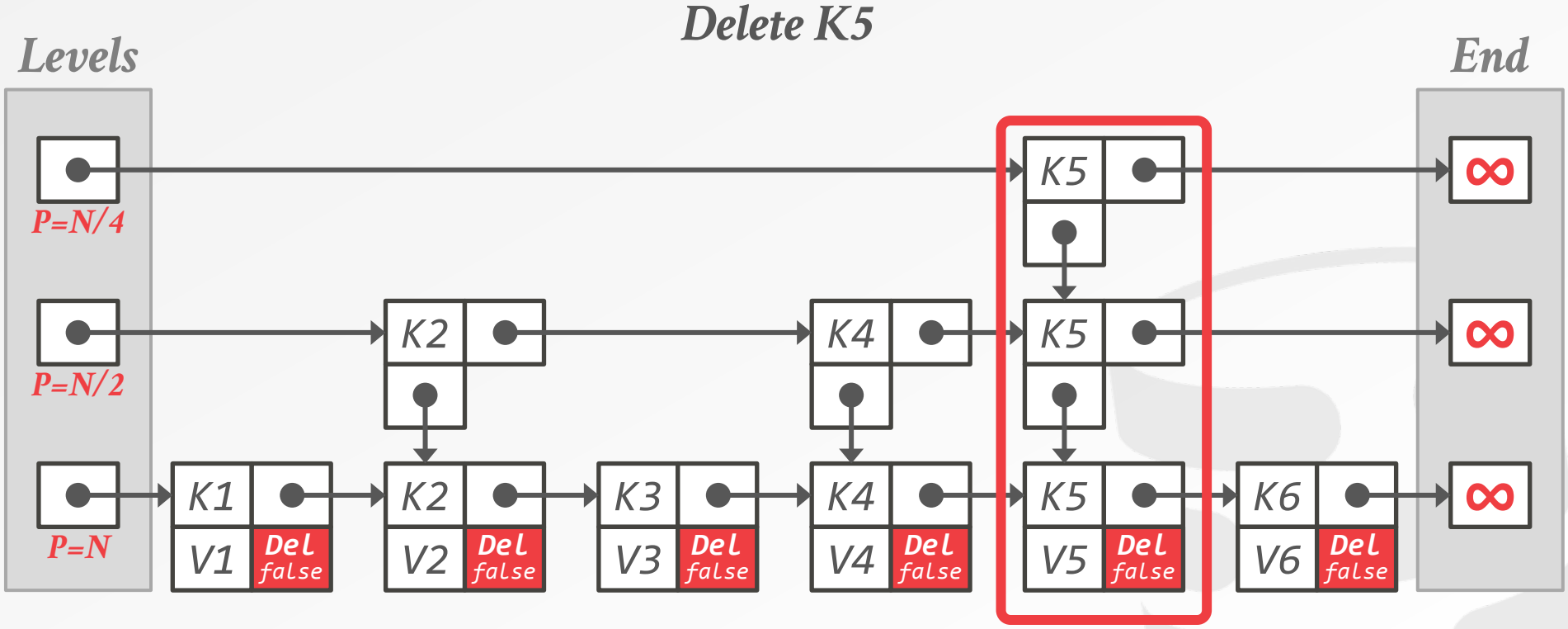
First **logically** remove a key from the index by setting a flag to tell threads to ignore.

Then **physically** remove the key once we know that no other thread is holding the reference.
→ Perform CaS to update the predecessor's pointer.

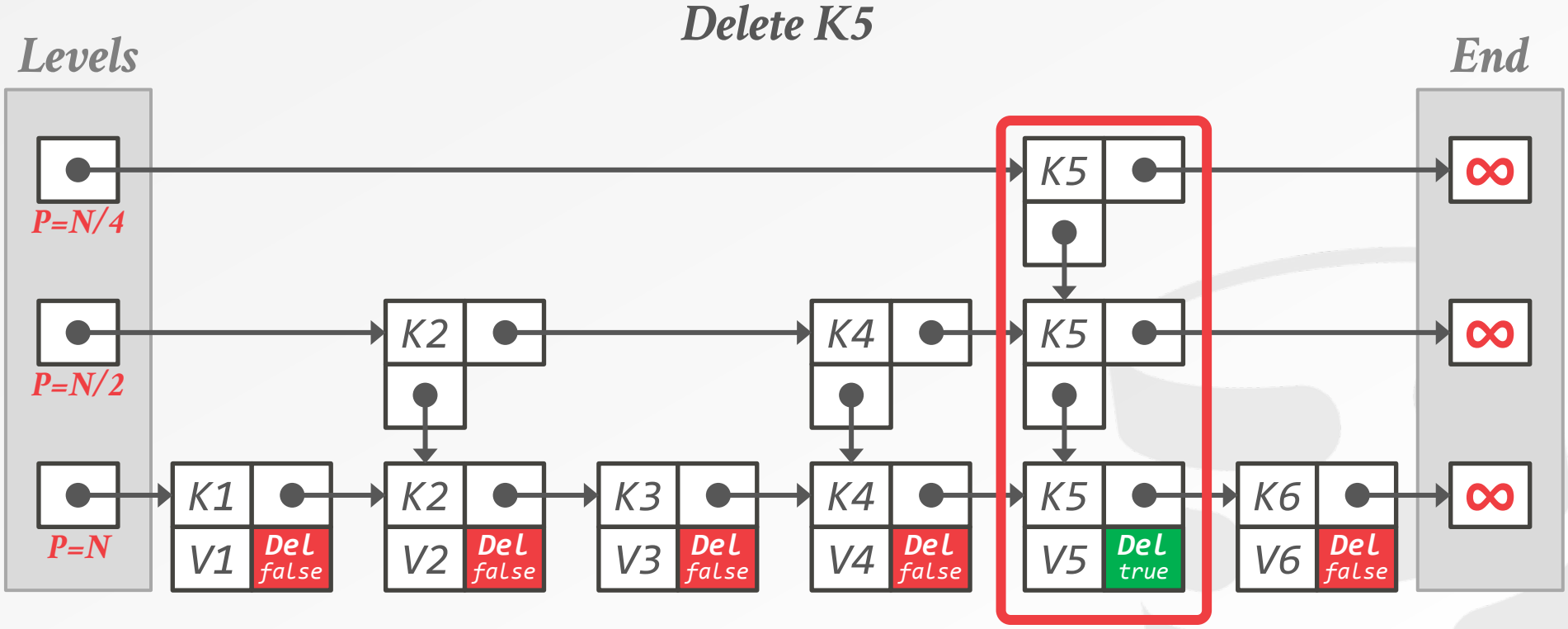
SKIP LISTS: DELETE



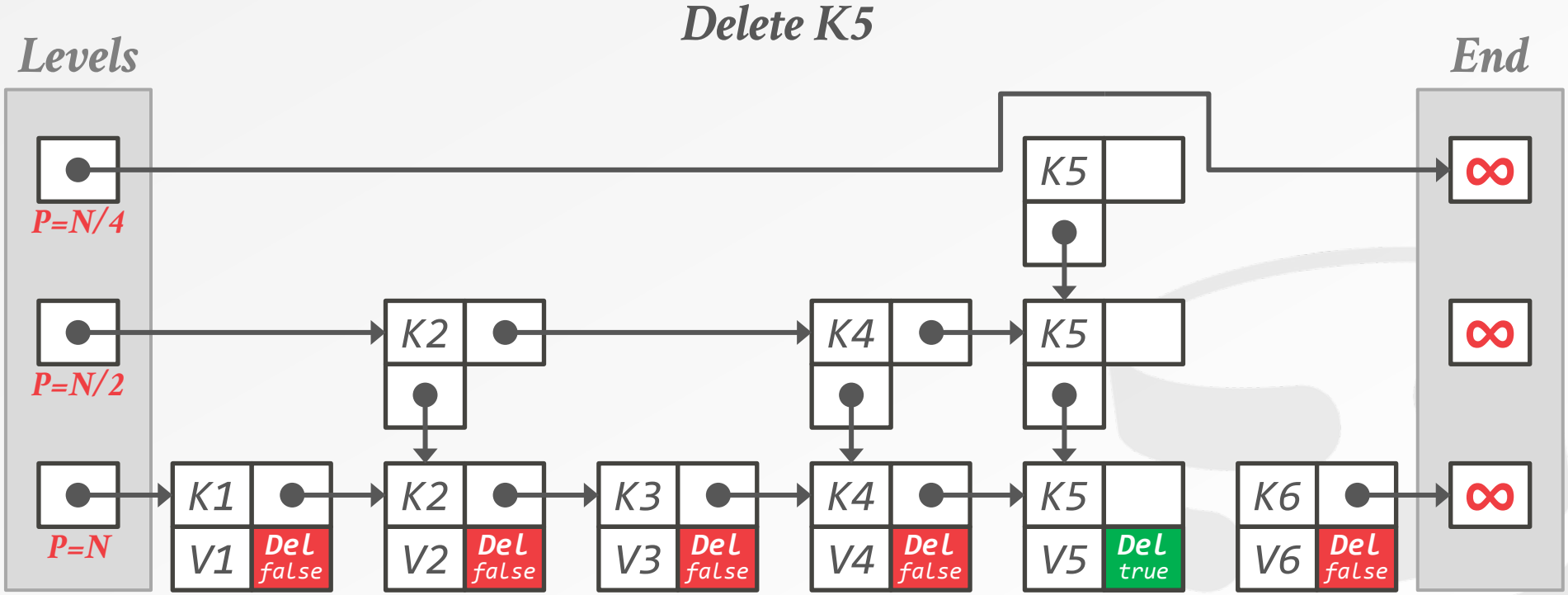
SKIP LISTS: DELETE



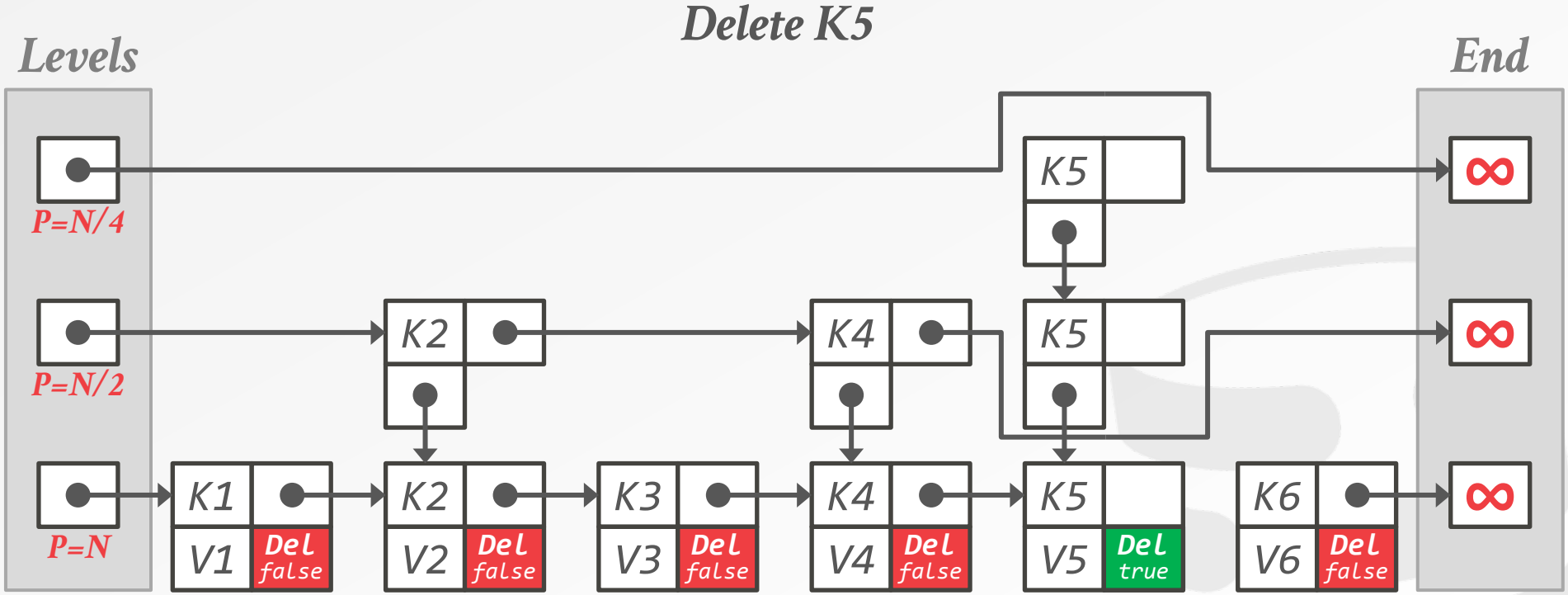
SKIP LISTS: DELETE



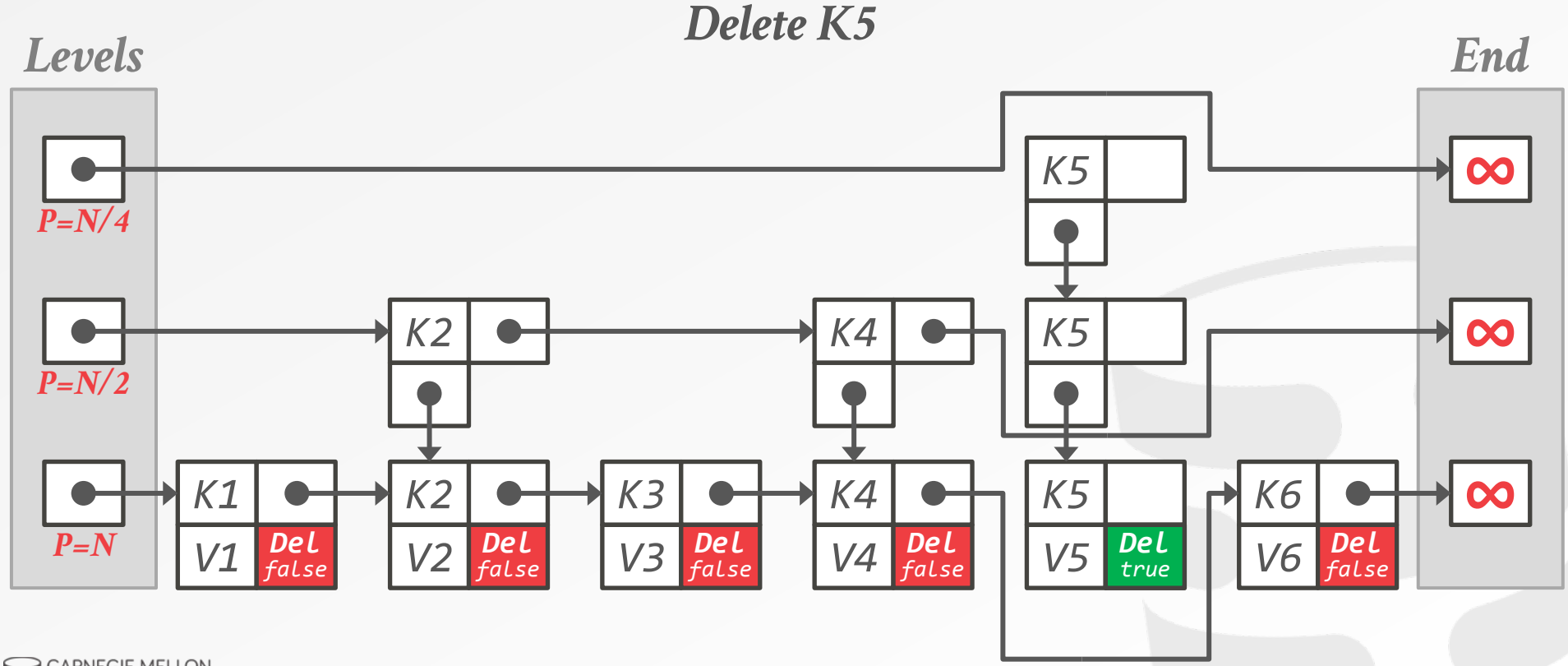
SKIP LISTS: DELETE



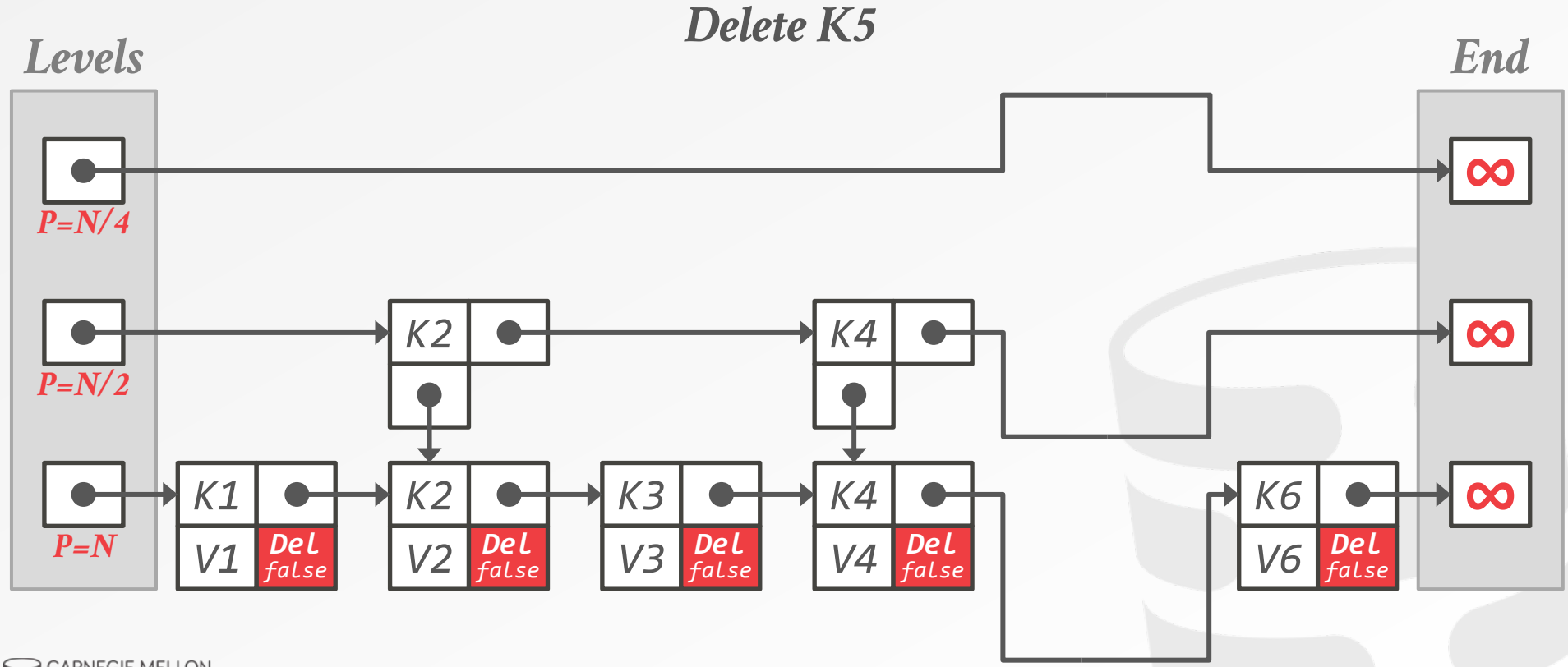
SKIP LISTS: DELETE



SKIP LISTS: DELETE



SKIP LISTS: DELETE



CONCURRENT SKIP LIST

Be careful about how you order operations.

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

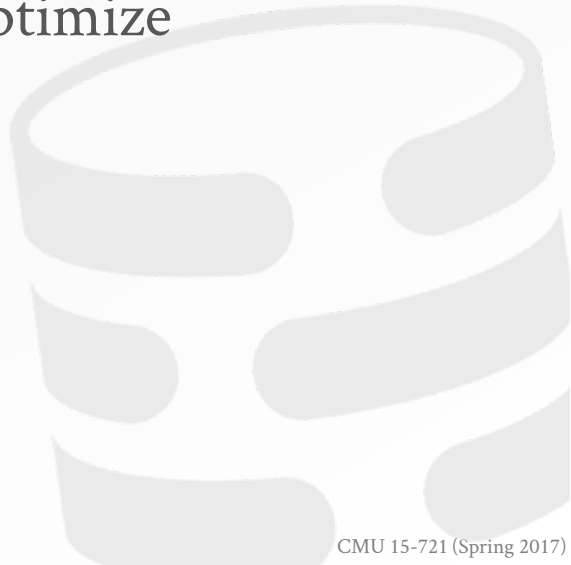
→ A txn can only abort due to higher-level conflicts.
→ If a CaS fails, then the index will retry until it succeeds.

SKIP LISTS: DISADVANTAGES

Invoking random number generator multiple times per insert is slow.

Not cache friendly because they do not optimize locality of references.

Reverse search is non-trivial.



SKIP LIST OPTIMIZATIONS

Reducing `RAND()` invocations.

Packing multiple keys in a node.

Reverse iteration with a stack.

Reusing nodes with memory pools.

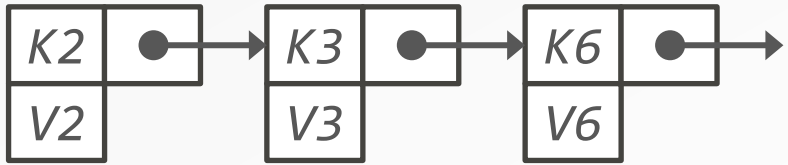
SKIP LISTS: DONE RIGHT
Ticki(?) Blog 2016

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.



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.

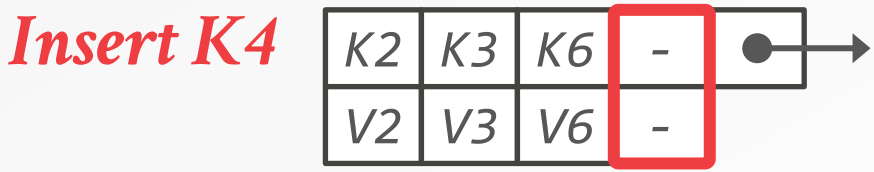
→ **Search Key:** Perform linear search on keys in each node.

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.



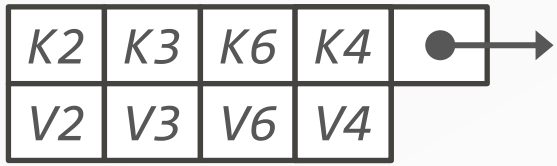
SKIP LIST: COMBINE NODES

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



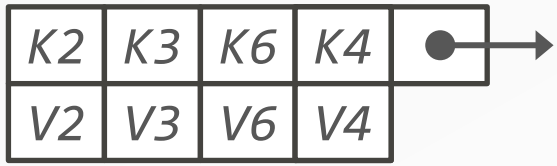
SKIP LIST: COMBINE NODES

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.

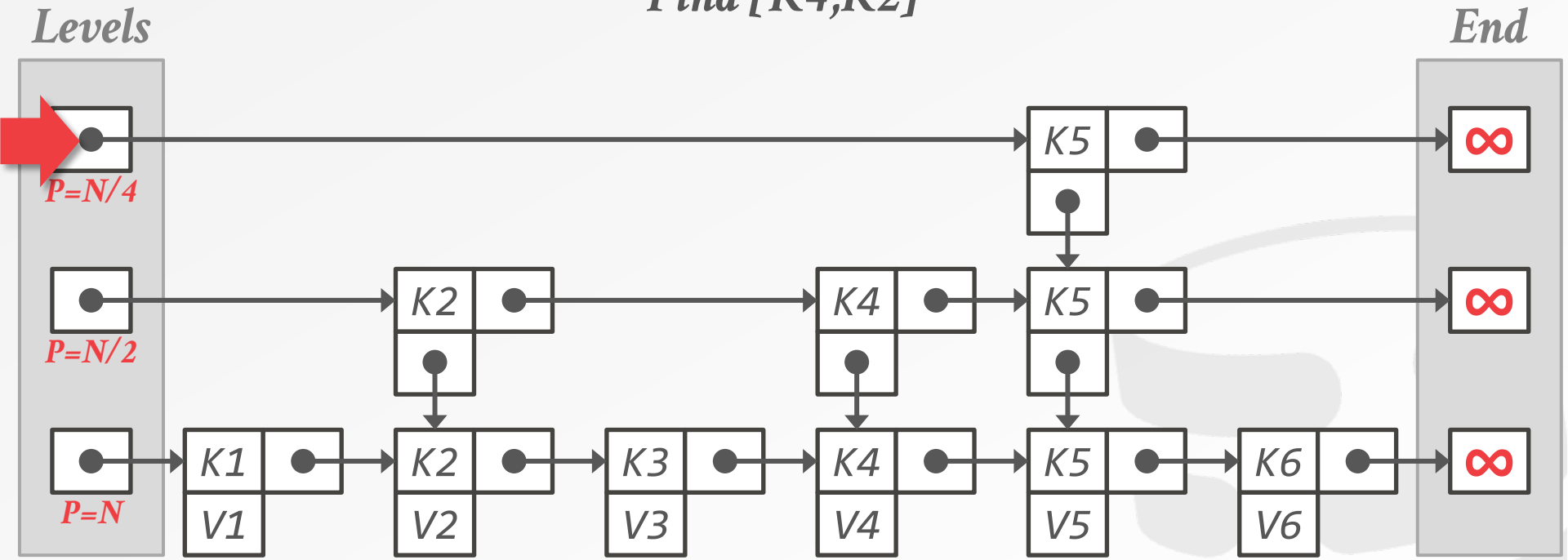
→ **Search Key:** Perform linear search on keys in each node.

Search K6

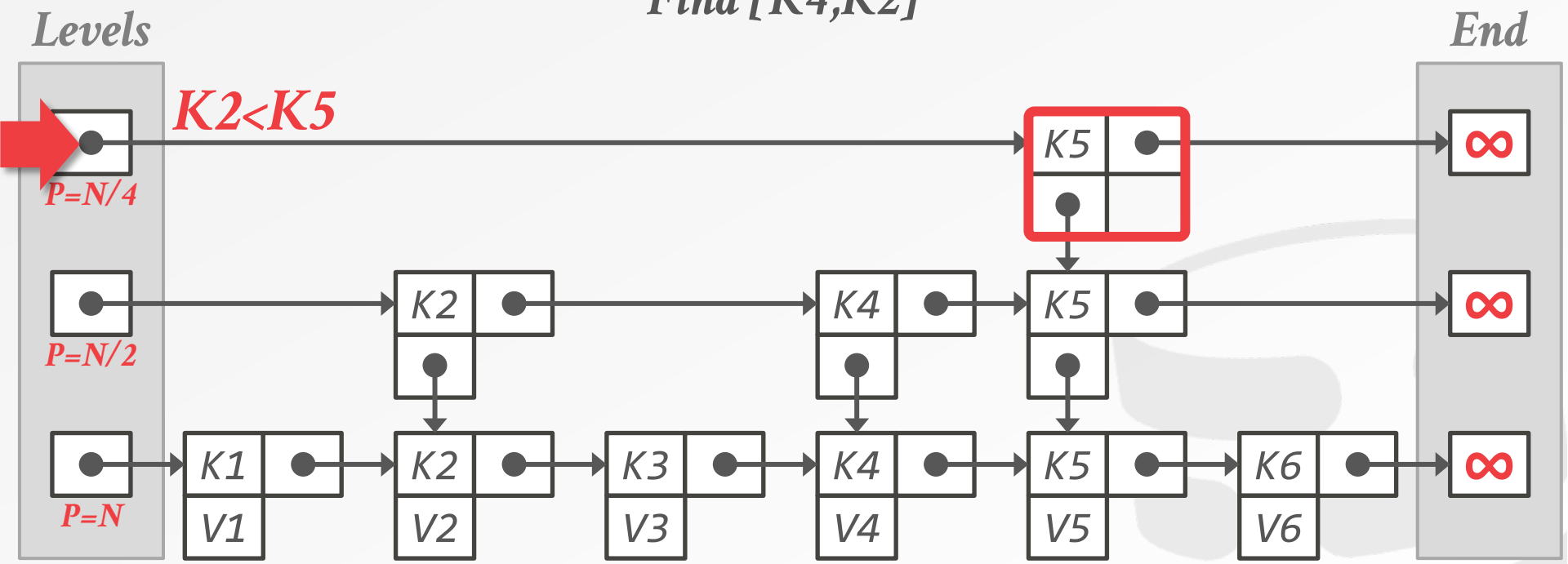


SKIP LISTS: REVERSE SEARCH

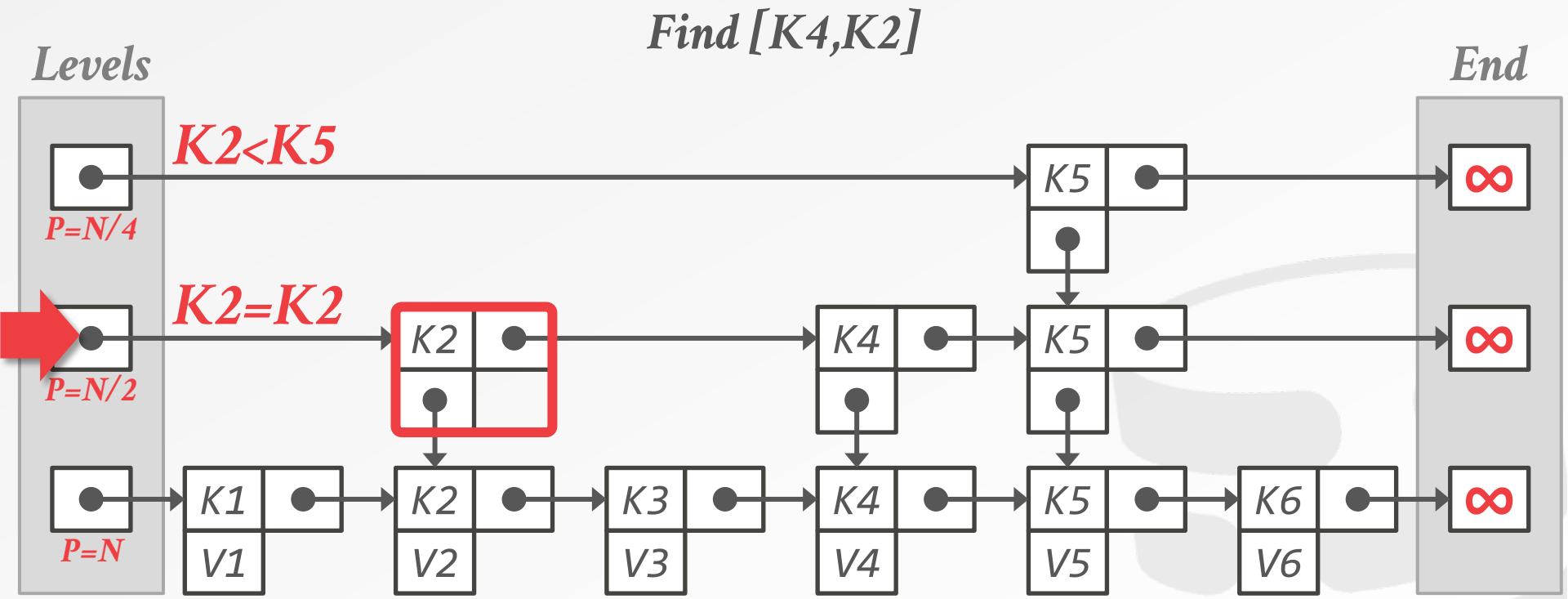
Find [K4,K2]



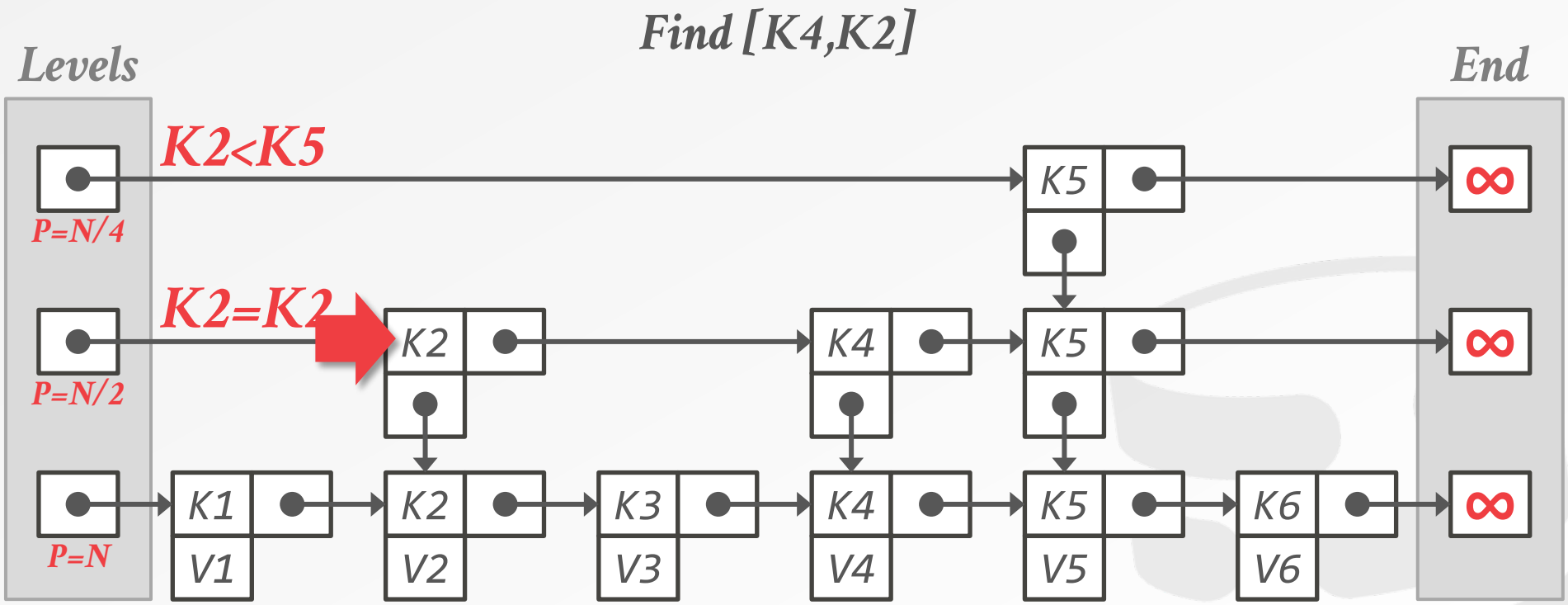
SKIP LISTS: REVERSE SEARCH



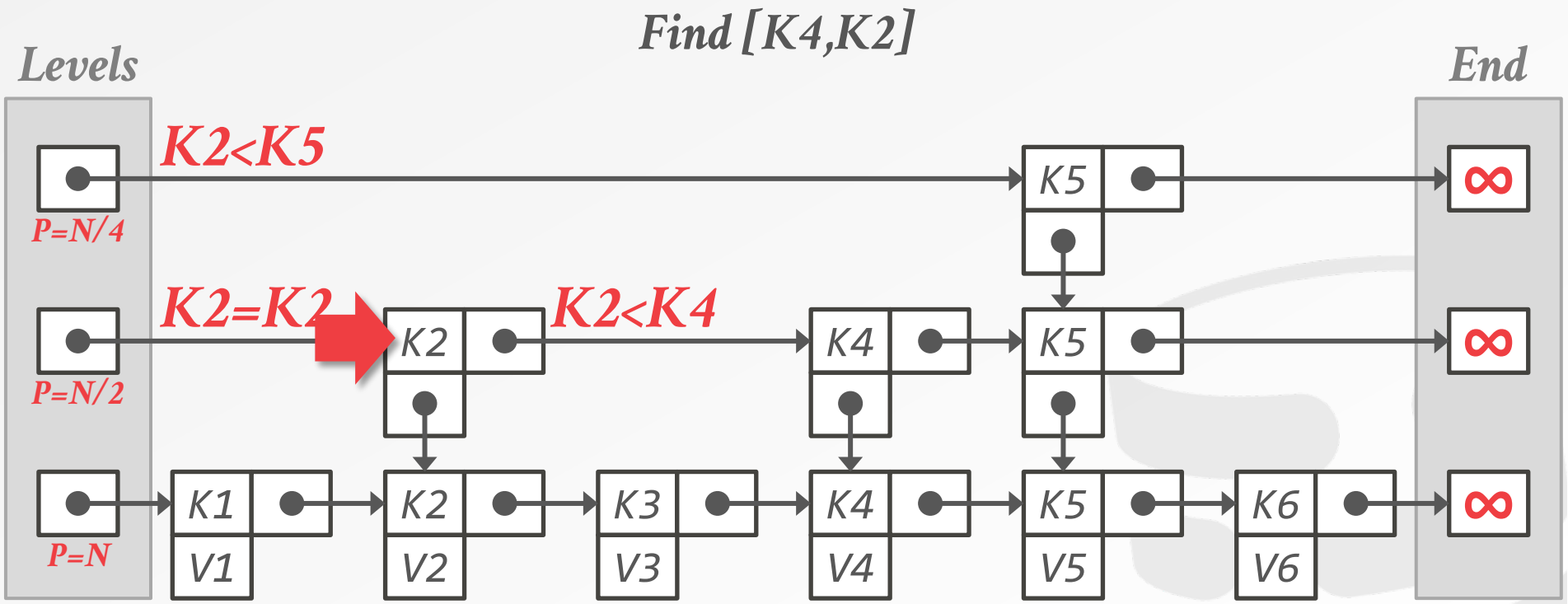
SKIP LISTS: REVERSE SEARCH



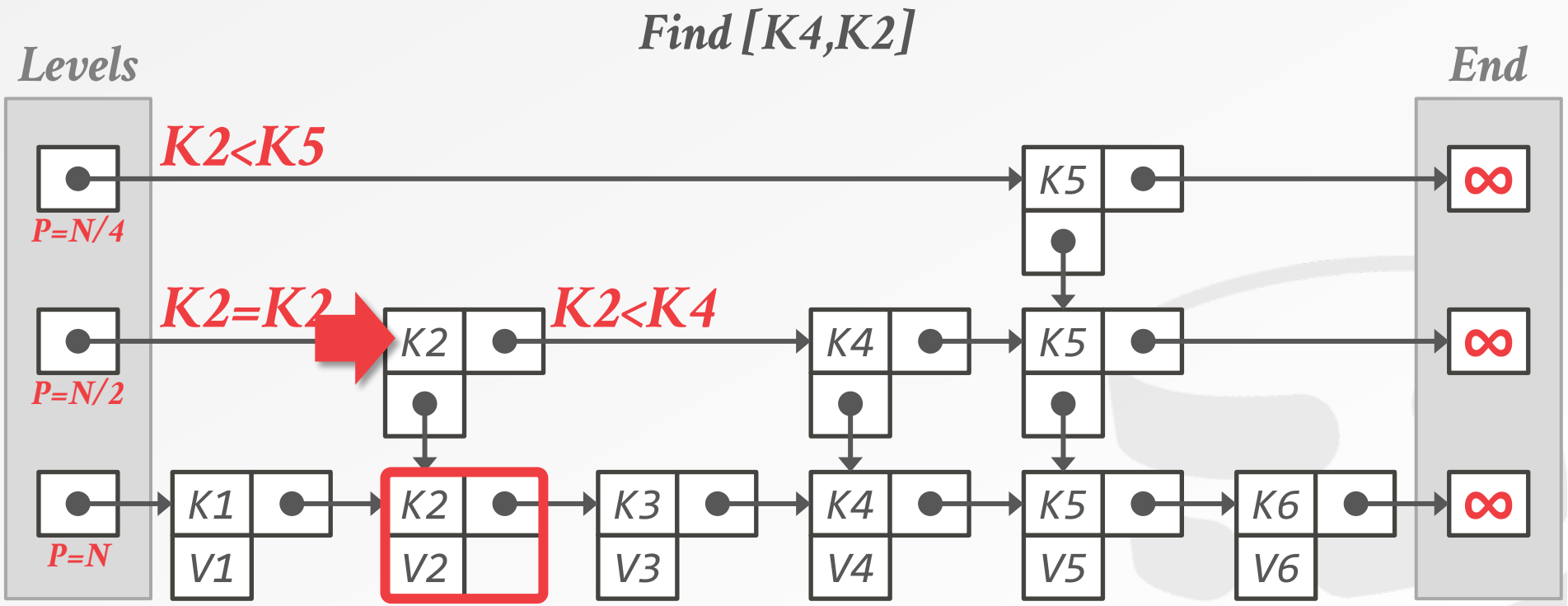
SKIP LISTS: REVERSE SEARCH



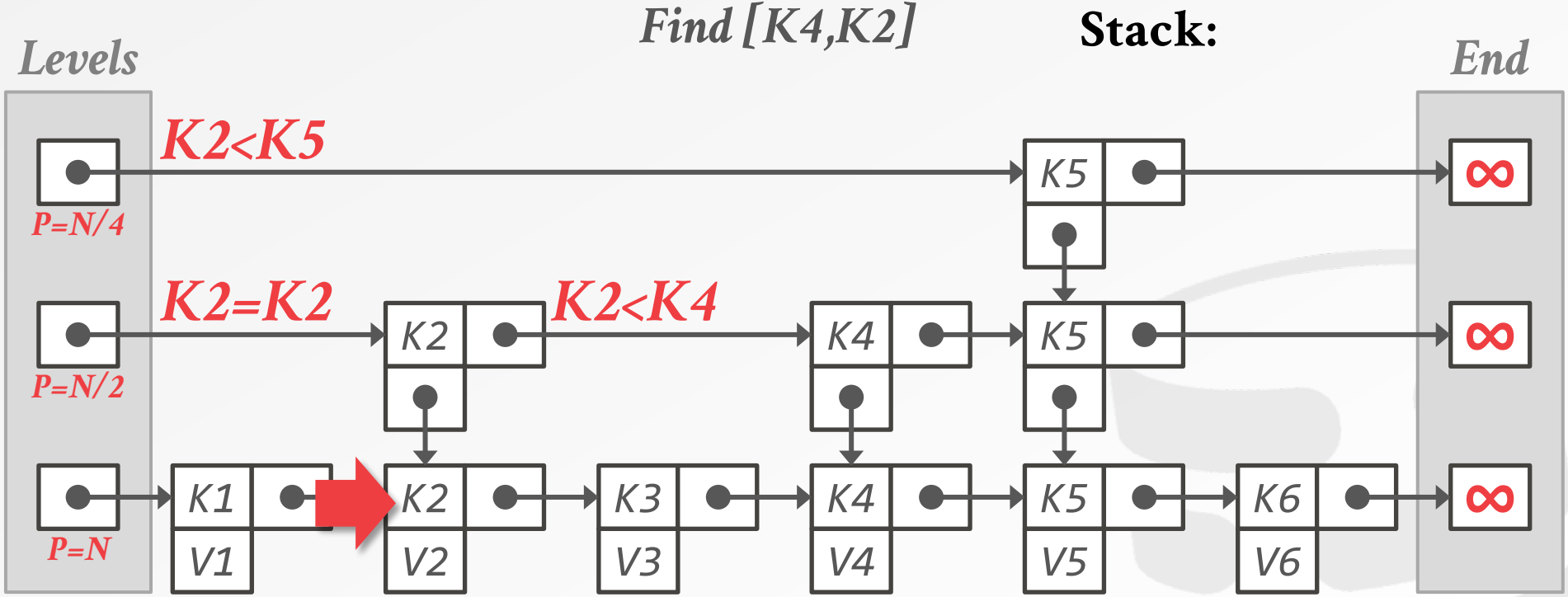
SKIP LISTS: REVERSE SEARCH



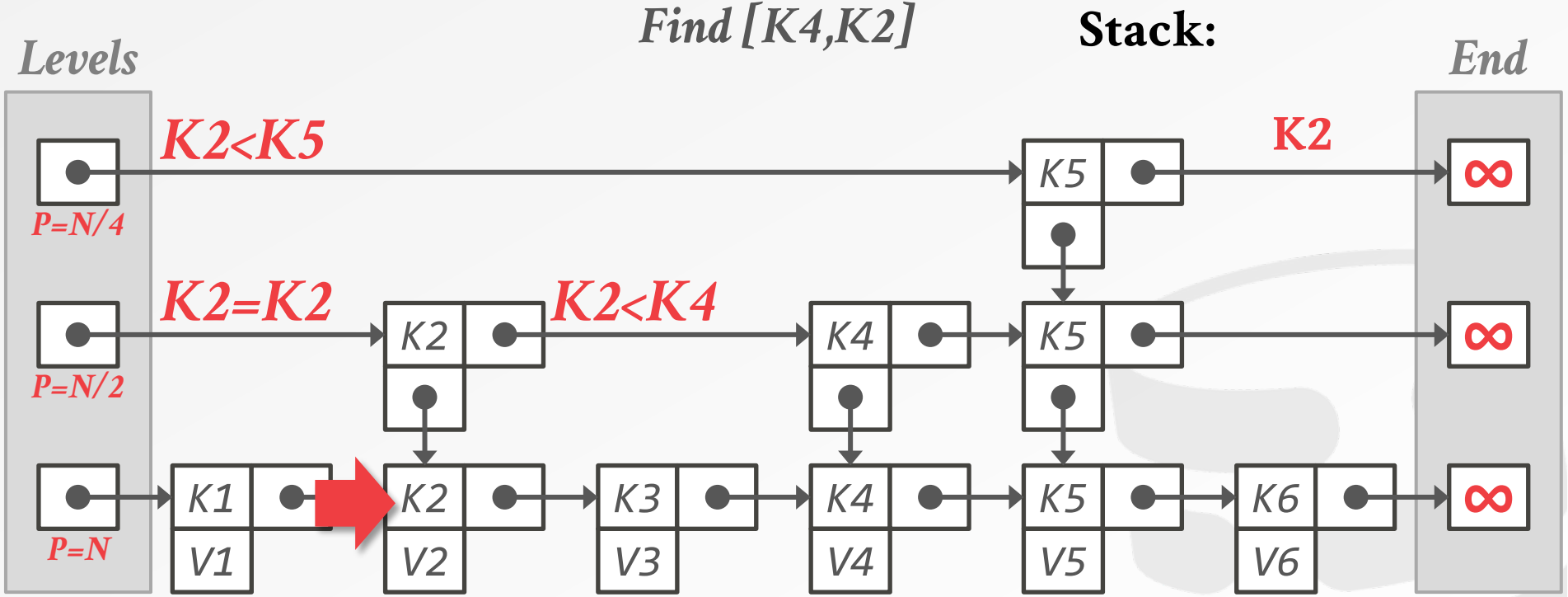
SKIP LISTS: REVERSE SEARCH



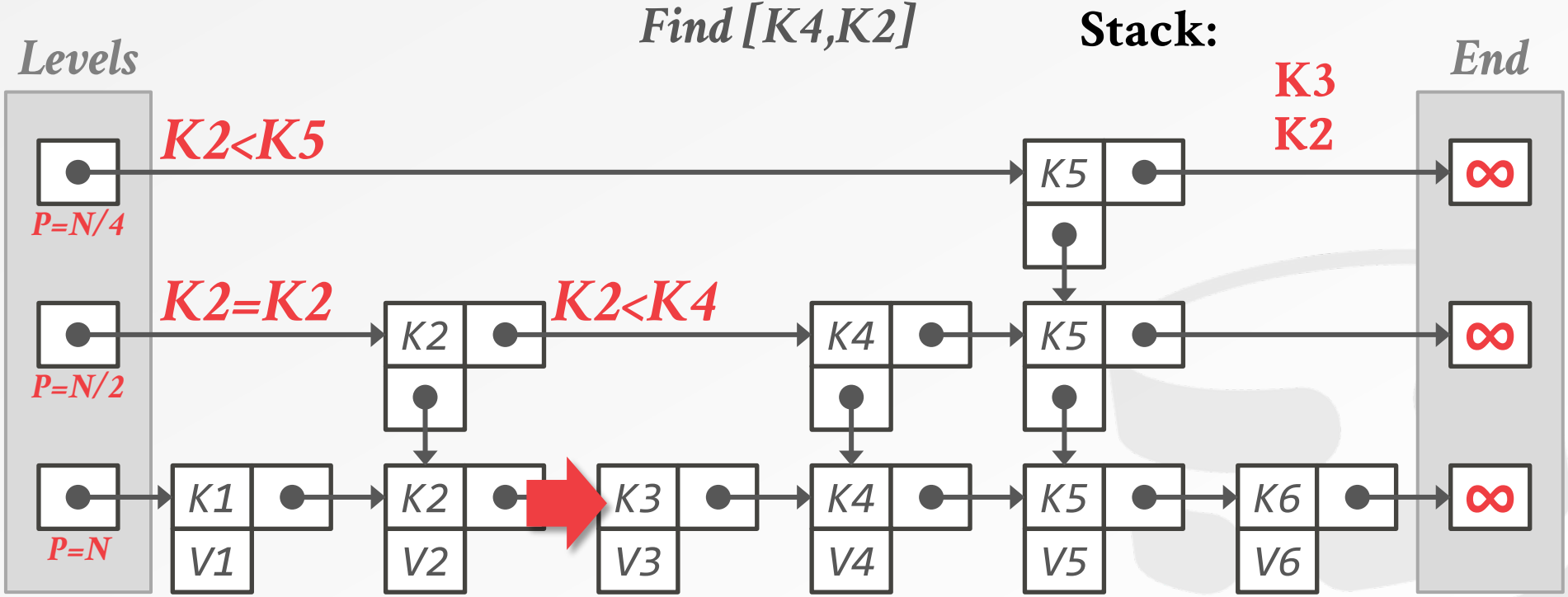
SKIP LISTS: REVERSE SEARCH



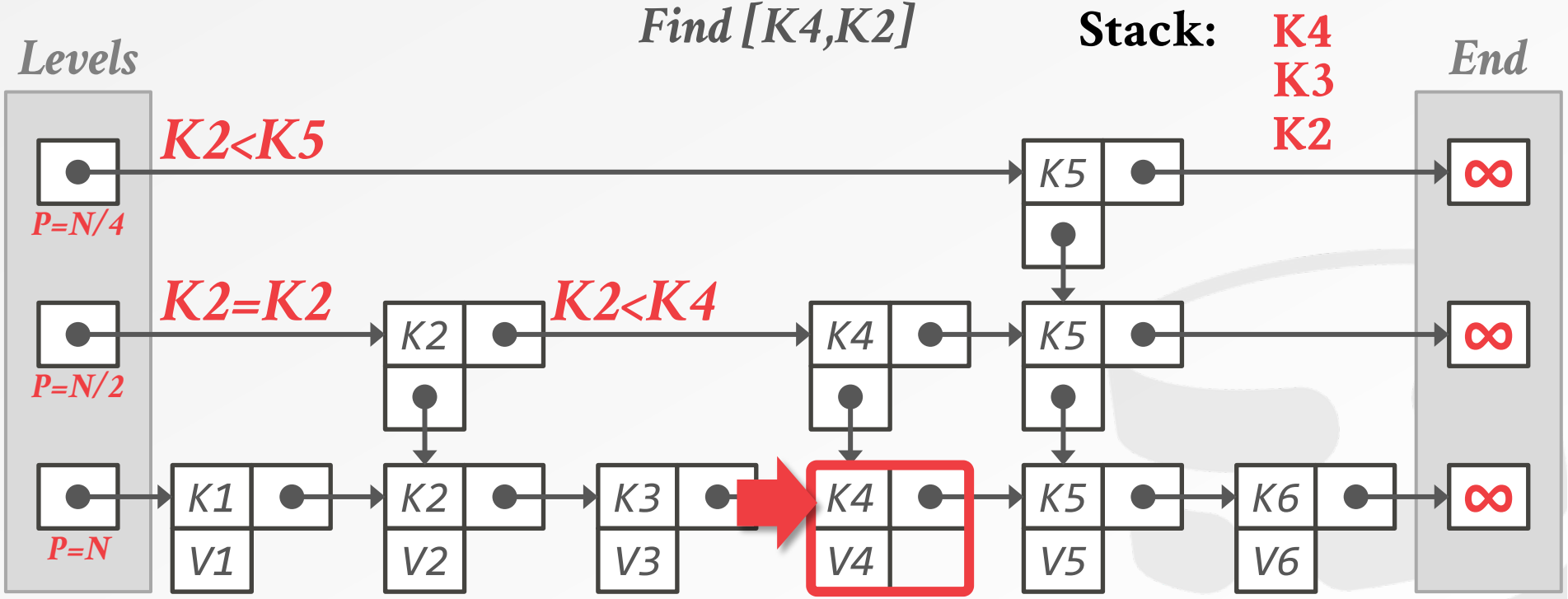
SKIP LISTS: REVERSE SEARCH



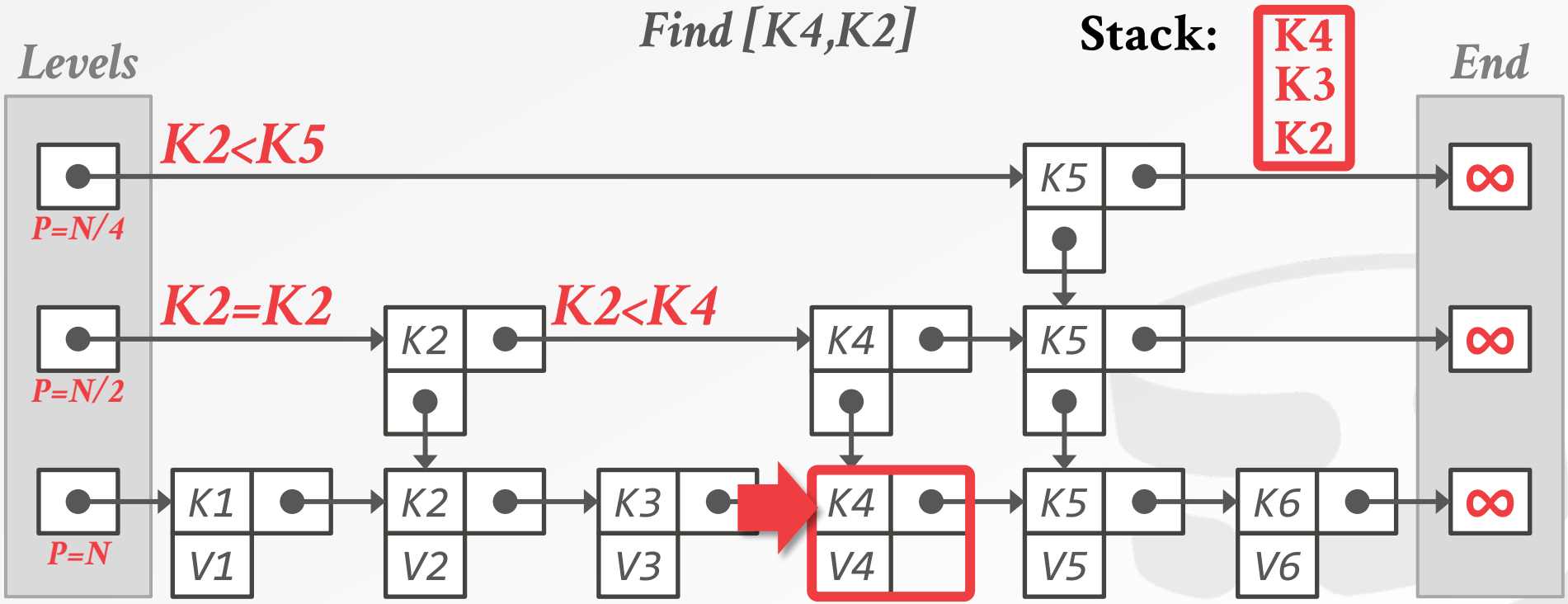
SKIP LISTS: REVERSE SEARCH



SKIP LISTS: REVERSE SEARCH



SKIP LISTS: REVERSE SEARCH



INDEX IMPLEMENTATION ISSUES

Memory Pools

Garbage Collection

Non-Unique Keys

Variable-length Keys

MEMORY POOLS

We don't want to be calling `malloc` and `free` anytime we need to add or delete a node.

If all the nodes are the same size, then the index can maintain a pool of available nodes.

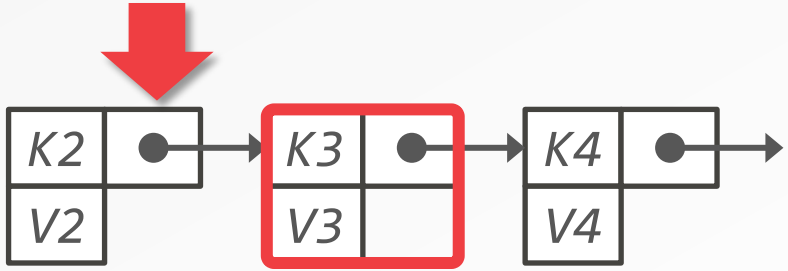
→ **Insert**: Grab a free node, otherwise create a new one.
→ **Delete:** Add the node back to the free pool.

Need some policy to decide when to retract the pool size.

GARBAGE COLLECTION

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

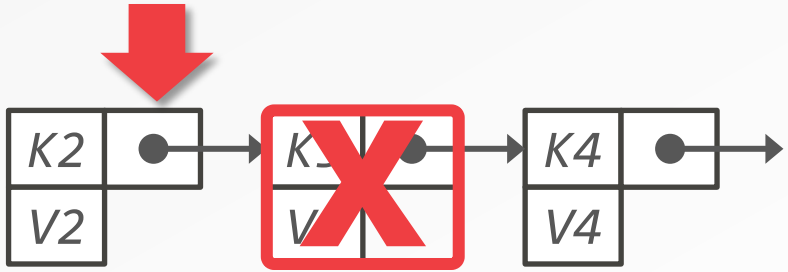
→ Reference Counting
→ Epoch-based Reclamation
→ Hazard Pointers
→ Many others…



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

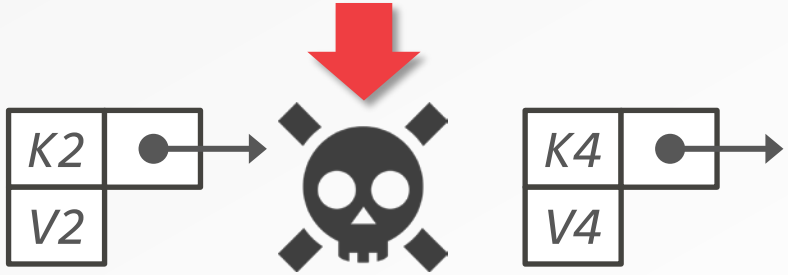
We need to know when it is safe to reclaim memory for deleted nodes in a latch-free index.

→ 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 actually 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.

Source: Stephen Tu

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.

NON-UNIQUE INDEXES

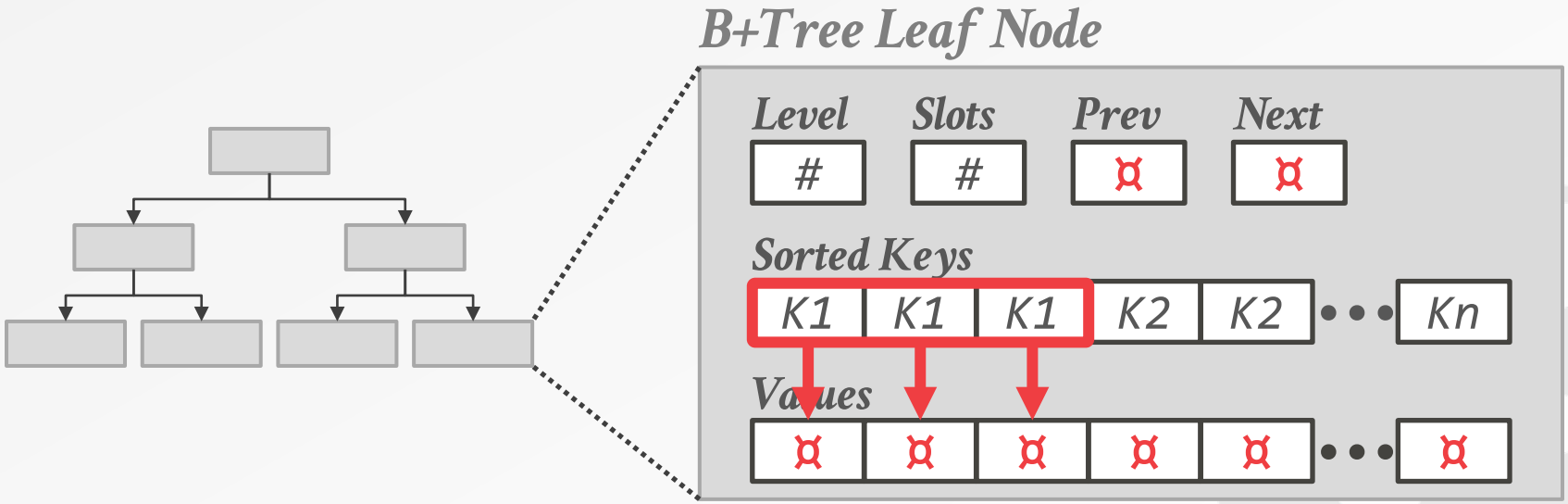
Approach #1: Duplicate Keys

→ Use the same node layout but store duplicate keys multiple times.

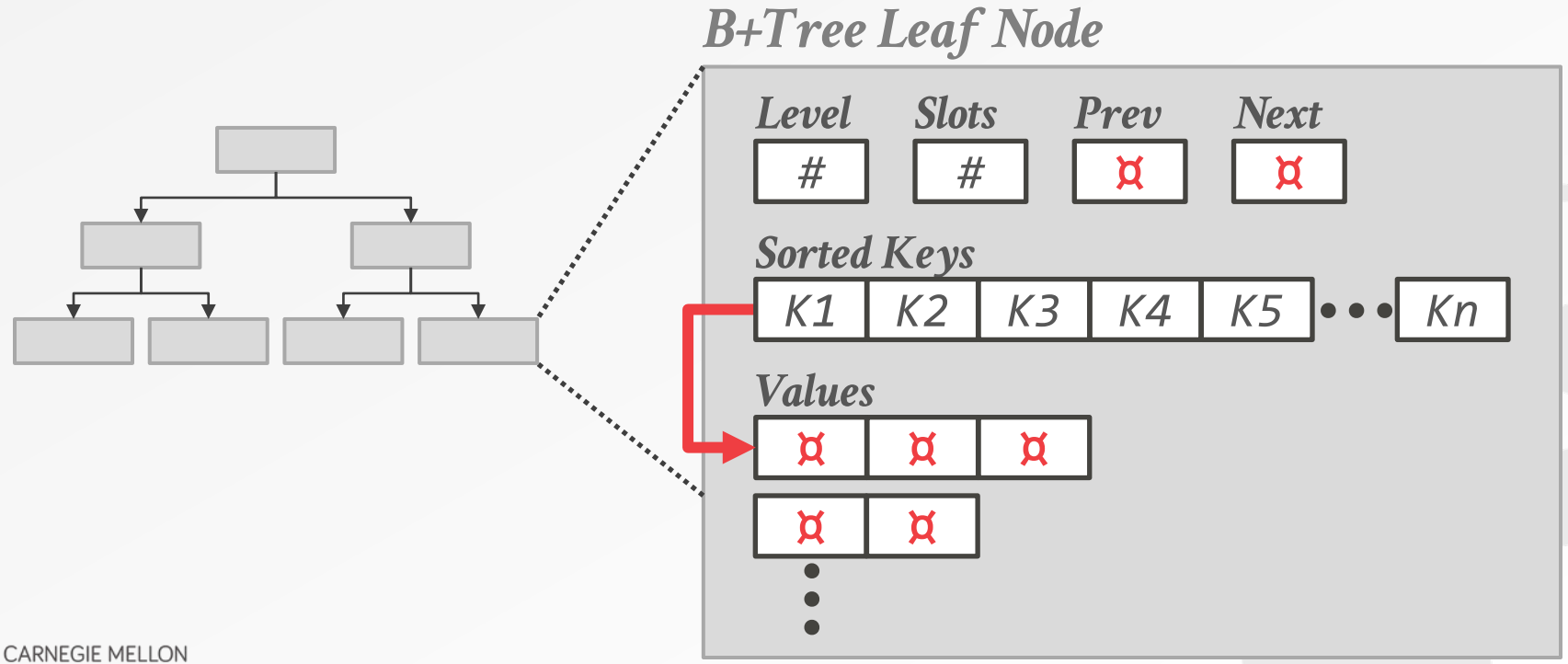
Approach #2: Value Lists

→ Store each key only once and maintain a linked list of unique values.

DUPLICATE KEYS



VALUE LISTS


B+Tree Leaf Node
Level
Slots
Prev
Next
#
#
Sorted Keys
K1
K2
K3
K4
K5
Kn
Values

VARIABLE LENGTH KEYS

Approach #1: Pointers

→ Store the keys as pointers to the tuple's attribute.

Approach #2: Variable Length Nodes

→ The size of each node in the index can vary.
→ Requires careful memory management.

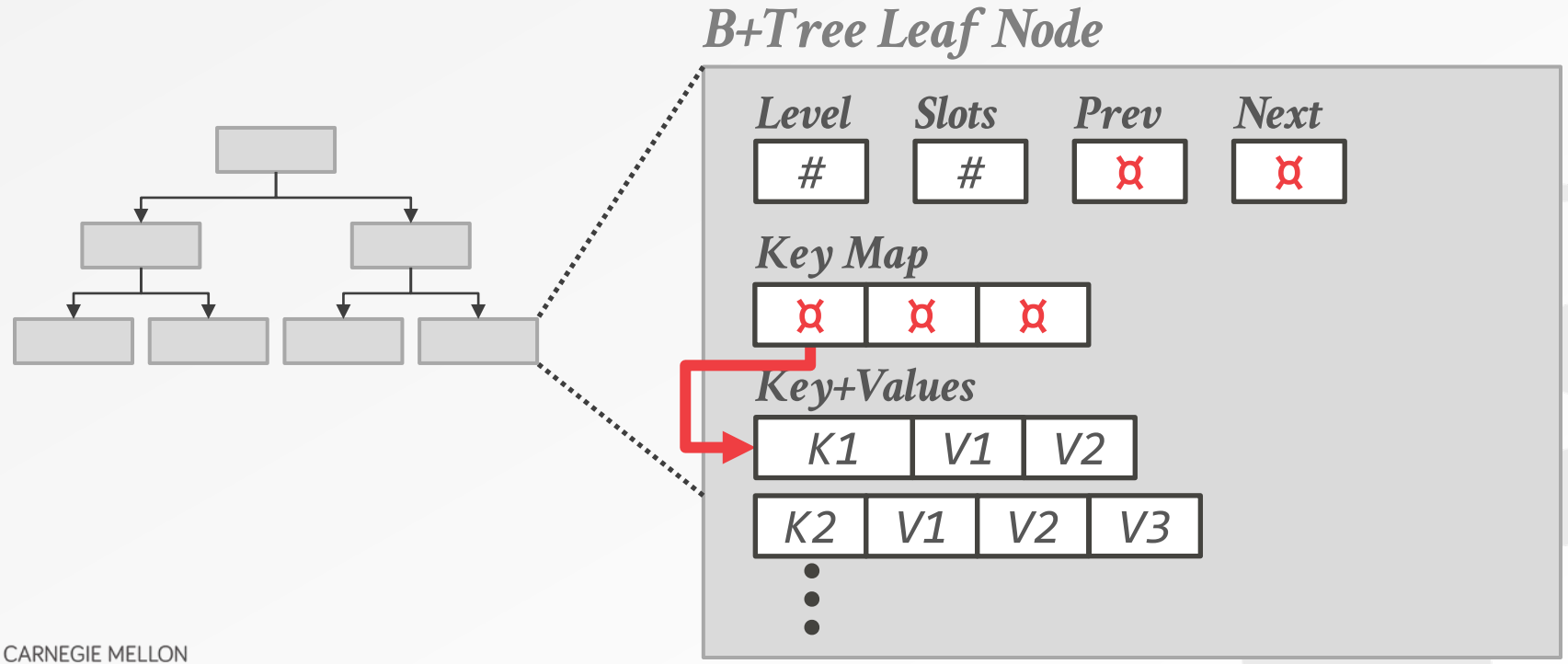
Approach #3: Padding

→ Always pad the key to be max length of the key type.

Approach #4: Key Map

→ Embed an array of pointers that map to the key + value list within the node.

KEY MAP


B+Tree Leaf Node
Level
Slots
Prev
Next
#
#
Key Map
Key+Values
K1
V1
V2
K2
V1
V2
V3

PARTING THOUGHTS

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

Skip List is really easy to implement.

Concurrent Skip List is more tricky.

Epoch garbage collection is more cache friendly.

NEXT CLASS

More OLTP Indexes
→ Microsoft Bw-Tree
→ HyPer ART

Crash course on performance testing.