



# 15-721 DATABASE SYSTEMS

## Lecture #07 – Indexing (OLTP)

---

Andy Pavlo // Carnegie Mellon University // Spring 2016

# TODAY'S AGENDA

---

Latch Implementations  
Modern OLTP Indexes

# COMPARE-AND-SWAP

---

Atomic instruction that compares contents of a memory location **M** to a given value **V**

- If values are equal, installs new given value **V'** in **M**
- Otherwise operation fails

# COMPARE-AND-SWAP

---

Atomic instruction that compares contents of a memory location **M** to a given value **V**

- If values are equal, installs new given value **V'** in **M**
- Otherwise operation fails

```
__sync_bool_compare_and_swap(&M, 20, 30)
```

# COMPARE-AND-SWAP

---

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V'** in **M**

→ Otherwise operation fails

Address



```
__sync_bool_compare_and_swap(&M, 20, 30)
```

# COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V'** in **M**

→ Otherwise operation fails

**M**  
20

Address

`__sync_bool_compare_and_swap(&M, 20, 30)`

# COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V'** in **M**

→ Otherwise operation fails

**M**  
20

`__sync_bool_compare_and_swap(&M, 20, 30)`

Address

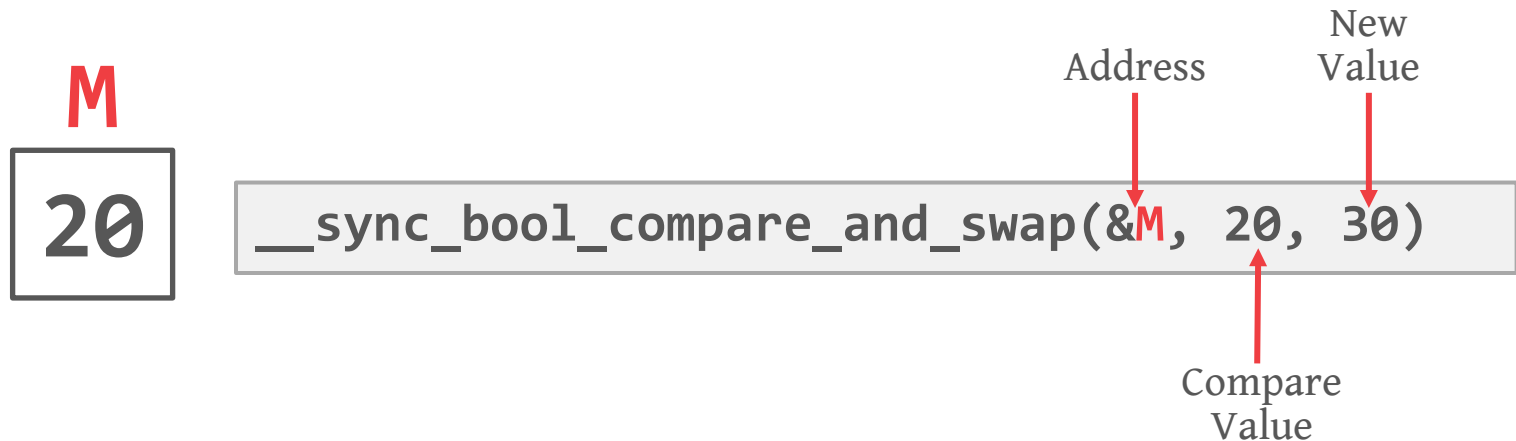
Compare  
Value

# COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V'** in **M**

→ Otherwise operation fails



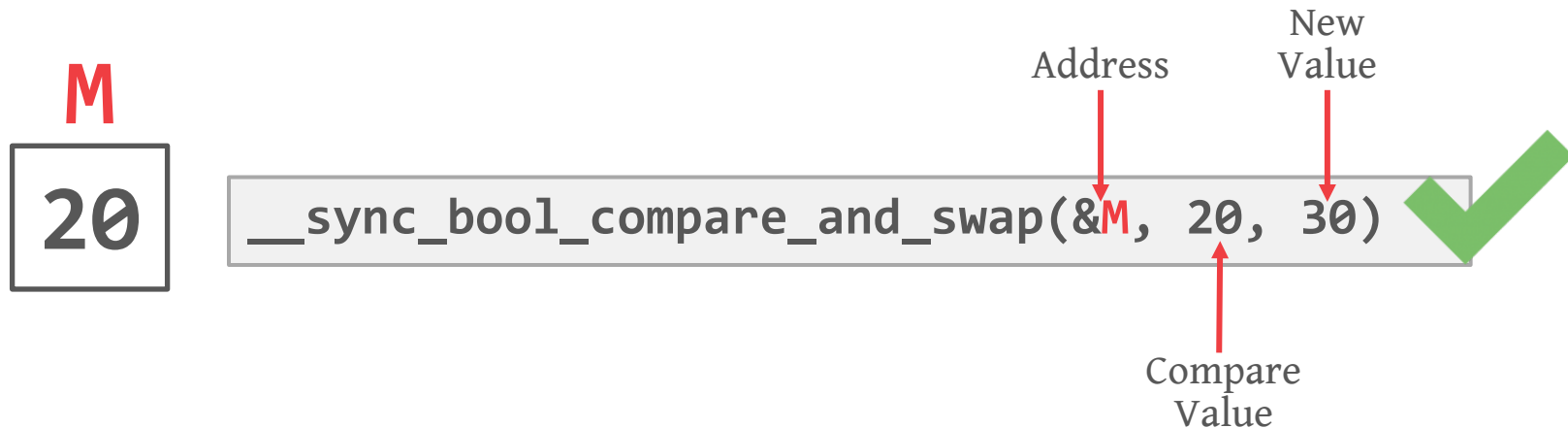


# COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V'** in **M**

→ Otherwise operation fails



# COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V'** in **M**

→ Otherwise operation fails

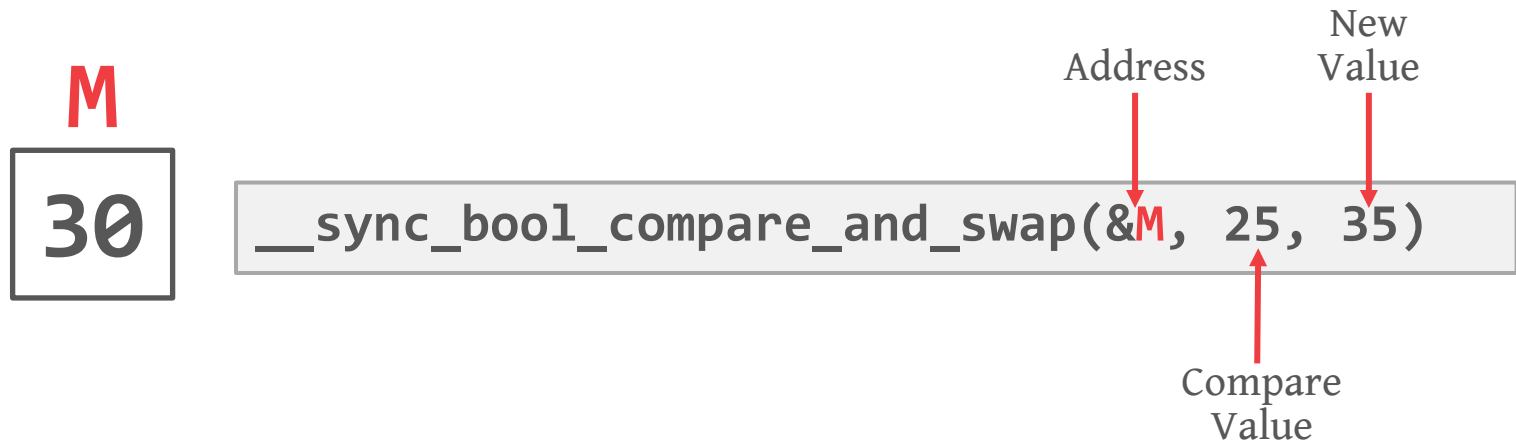


# COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V'** in **M**

→ Otherwise operation fails

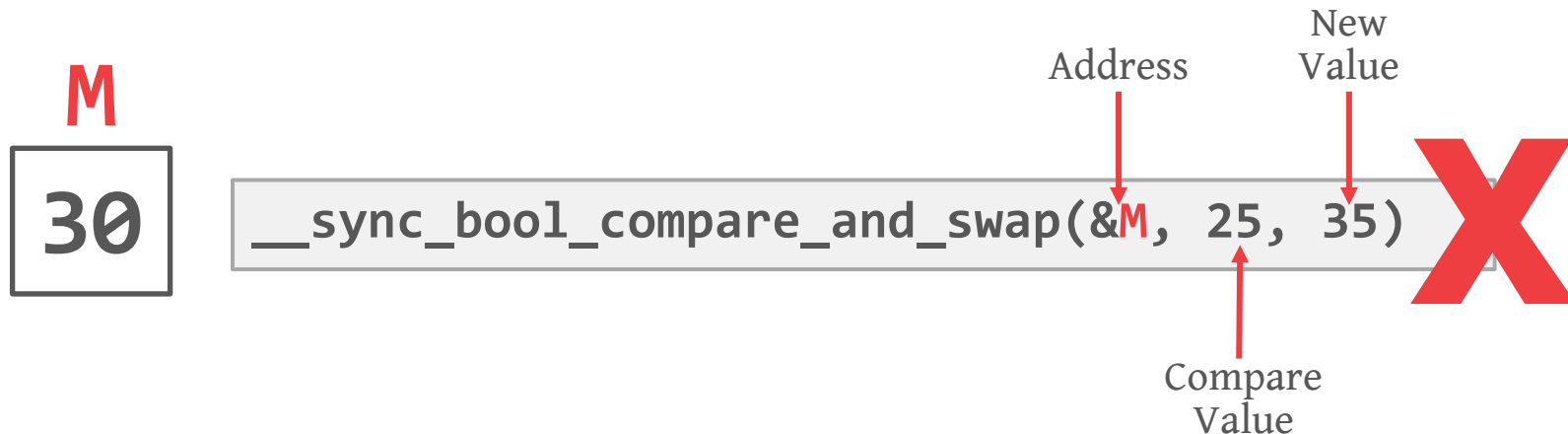


# COMPARE-AND-SWAP

Atomic instruction that compares contents of a memory location **M** to a given value **V**

→ If values are equal, installs new given value **V'** in **M**

→ Otherwise operation fails



# LATCH IMPLEMENTATIONS

---

Blocking OS Mutex

Test-and-Set Spinlock

Queue-based Spinlock

Reader-Writer Locks

# LATCH IMPLEMENTATIONS

---

## Choice #1: Blocking OS Mutex

- Simple to use
- Non-scalable (about 25ns per lock/unlock invocation)
- Example: `pthread_mutex_t` (calls `futex`)

# LATCH IMPLEMENTATIONS

---

## Choice #1: Blocking OS Mutex

- Simple to use
- Non-scalable (about 25ns per lock/unlock invocation)
- Example: `pthread_mutex_t` (calls `futex`)

```
pthread_mutex_t lock;  
:  
pthread_mutex_lock(&lock);  
// Do something special...  
pthread_mutex_unlock(&lock);
```

# LATCH IMPLEMENTATIONS

---

## Choice #1: Blocking OS Mutex

- Simple to use
- Non-scalable (about 25ns per lock/unlock invocation)
- Example: `pthread_mutex_t` (calls `futex`)

```
pthread_mutex_t lock;  
:  
pthread_mutex_lock(&lock);  
// Do something special...  
pthread_mutex_unlock(&lock);
```



# LATCH IMPLEMENTATIONS

---

## Choice #1: Blocking OS Mutex

- Simple to use
- Non-scalable (about 25ns per lock/unlock invocation)
- Example: `pthread_mutex_t` (calls `futex`)

```
pthread_mutex_t lock;  
:  
pthread_mutex_lock(&lock);  
// Do something special...  
pthread_mutex_unlock(&lock);
```

# LATCH IMPLEMENTATIONS

---

## Choice #2: Test-and-Set Spinlock (TAS)

- Very efficient (single instruction to lock/unlock)
- Non-scalable, not cache friendly
- Example: `std::atomic_flag`

# LATCH IMPLEMENTATIONS

---

## Choice #2: Test-and-Set Spinlock (TAS)

- Very efficient (single instruction to lock/unlock)
- Non-scalable, not cache friendly
- Example: `std::atomic_flag`

```
std::atomic_flag lock;  
:  
while (lock.test_and_set(...)) {  
    // Yield? Abort? Retry?  
}
```

# LATCH IMPLEMENTATIONS

---

## Choice #2: Test-and-Set Spinlock (TAS)

- Very efficient (single instruction to lock/unlock)
- Non-scalable, not cache friendly
- Example: `std::atomic_flag`

```
std::atomic_flag lock;  
:  
while (lock.test_and_set(...)) {  
    // Yield? Abort? Retry?  
}
```

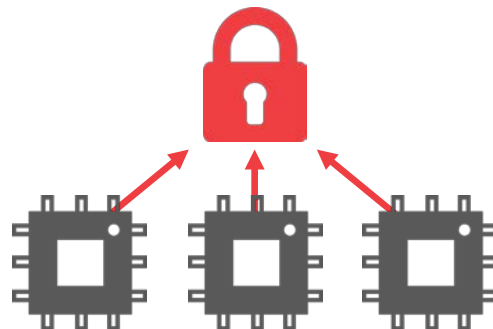


# LATCH IMPLEMENTATIONS

## Choice #2: Test-and-Set Spinlock (TAS)

- Very efficient (single instruction to lock/unlock)
- Non-scalable, not cache friendly
- Example: `std::atomic_flag`

```
std::atomic_flag lock;  
:  
while (lock.test_and_set(...)) {  
    // Yield? Abort? Retry?  
}
```



# LATCH IMPLEMENTATIONS

---

## Choice #3: Queue-based Spinlock (MCS)

- More efficient than mutex, better cache locality
- Non-trivial memory management
- Example: `std::atomic_flag`

# LATCH IMPLEMENTATIONS

---

*Mellor-Crummey and Scott*

## Choice #3: Queue-based Spinlock (MCS)

- More efficient than mutex, better cache locality
- Non-trivial memory management
- Example: `std::atomic_flag`

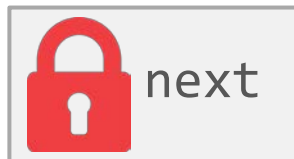
# LATCH IMPLEMENTATIONS

*Mellor-Crummey and Scott*

## Choice #3: Queue-based Spinlock (MCS)

- More efficient than mutex, better cache locality
- Non-trivial memory management
- Example: `std::atomic_flag`

*Base Lock*





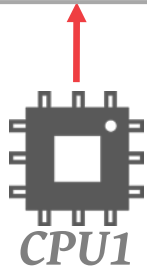
# LATCH IMPLEMENTATIONS

*Mellor-Crummey and Scott*

## Choice #3: Queue-based Spinlock (MCS)

- More efficient than mutex, better cache locality
- Non-trivial memory management
- Example: `std::atomic_flag`

*Base Lock*

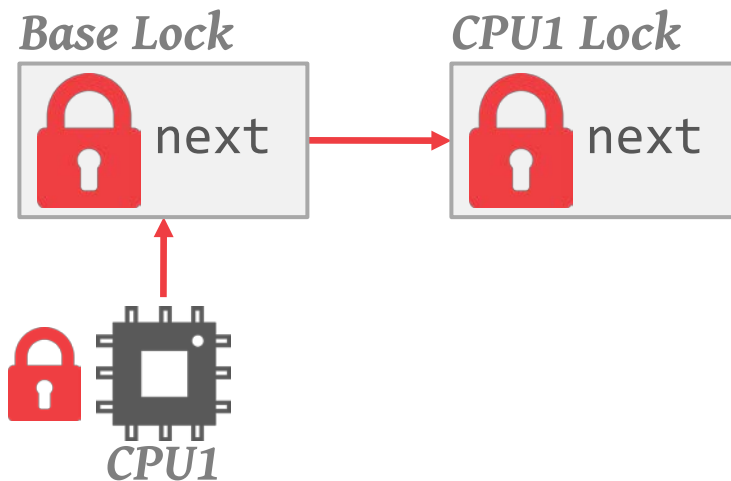


# LATCH IMPLEMENTATIONS

*Mellor-Crummey and Scott*

## Choice #3: Queue-based Spinlock (MCS)

- More efficient than mutex, better cache locality
- Non-trivial memory management
- Example: `std::atomic_flag`

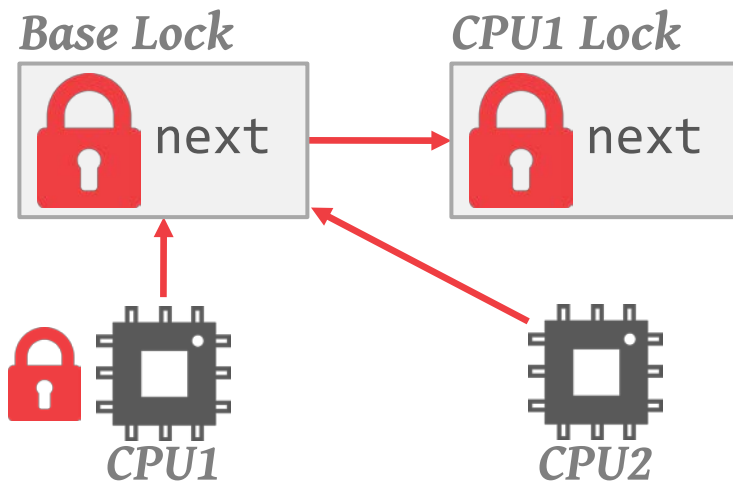


# LATCH IMPLEMENTATIONS

*Mellor-Crummey and Scott*

## Choice #3: Queue-based Spinlock (MCS)

- More efficient than mutex, better cache locality
- Non-trivial memory management
- Example: `std::atomic_flag`

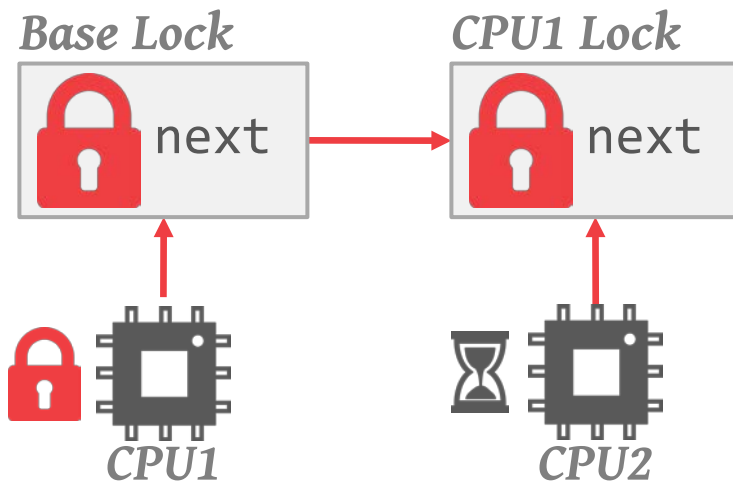


# LATCH IMPLEMENTATIONS

*Mellor-Crummey and Scott*

## Choice #3: Queue-based Spinlock (MCS)

- More efficient than mutex, better cache locality
- Non-trivial memory management
- Example: `std::atomic_flag`

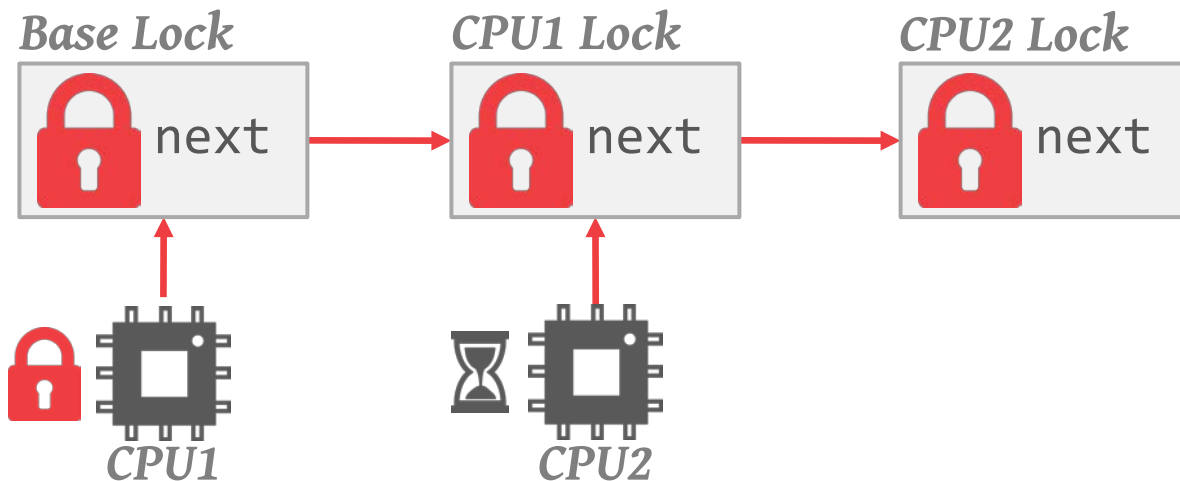


# LATCH IMPLEMENTATIONS

*Mellor-Crummey and Scott*

## Choice #3: Queue-based Spinlock (MCS)

- More efficient than mutex, better cache locality
- Non-trivial memory management
- Example: `std::atomic_flag`

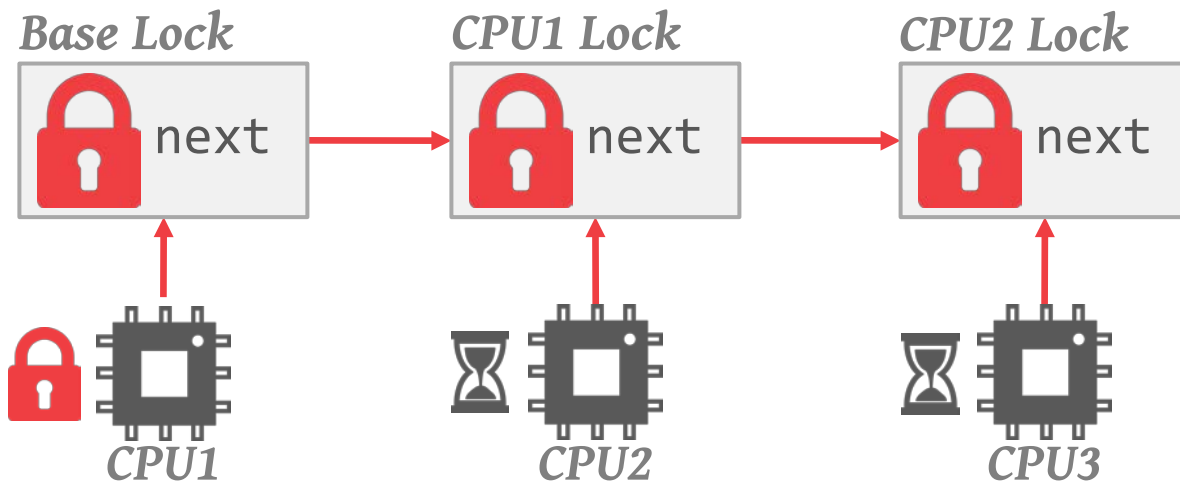


# LATCH IMPLEMENTATIONS

*Mellor-Crummey and Scott*

## Choice #3: Queue-based Spinlock (MCS)

- More efficient than mutex, better cache locality
- Non-trivial memory management
- Example: `std::atomic_flag`



# LATCH IMPLEMENTATIONS

---

## **Choice #4: Reader-Writer Locks**

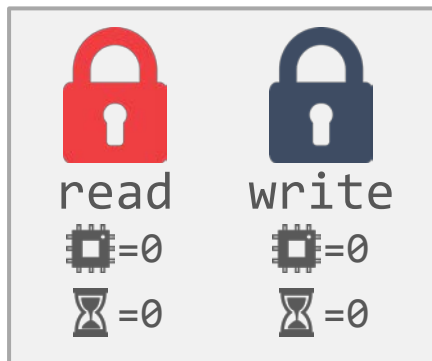
- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks

# LATCH IMPLEMENTATIONS

## Choice #4: Reader-Writer Locks

- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks

### *Latch*





# LATCH IMPLEMENTATIONS

## Choice #4: Reader-Writer Locks

- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks

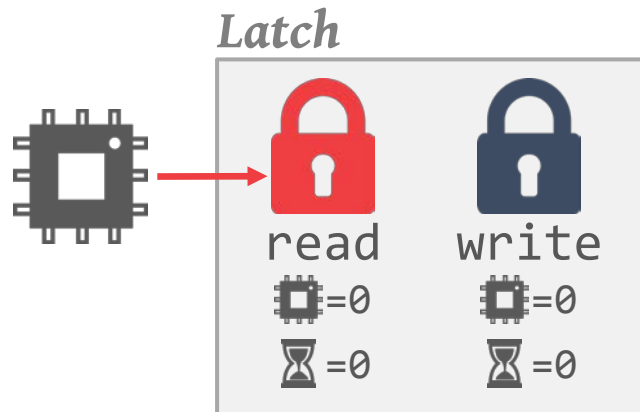
### *Latch*



# LATCH IMPLEMENTATIONS

## Choice #4: Reader-Writer Locks

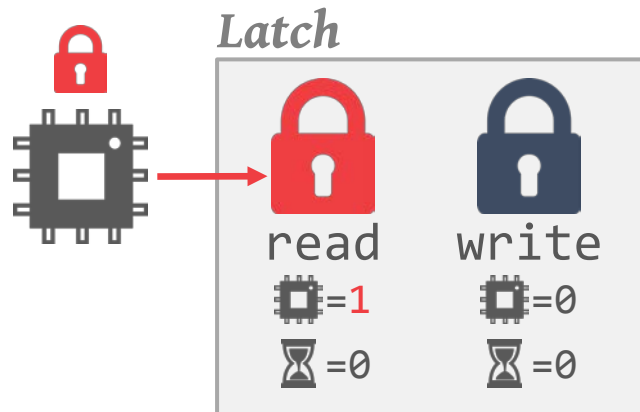
- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



# LATCH IMPLEMENTATIONS

## Choice #4: Reader-Writer Locks

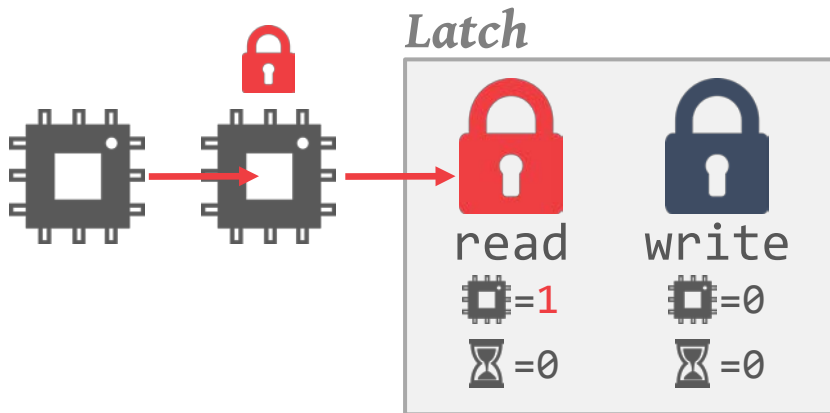
- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



# LATCH IMPLEMENTATIONS

## Choice #4: Reader-Writer Locks

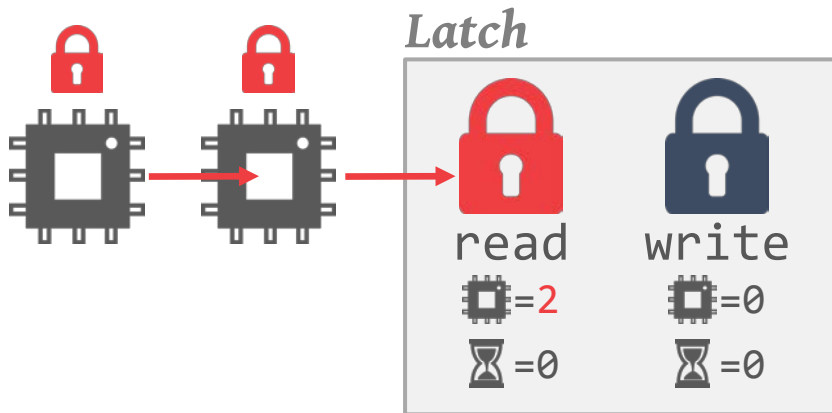
- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



# LATCH IMPLEMENTATIONS

## Choice #4: Reader-Writer Locks

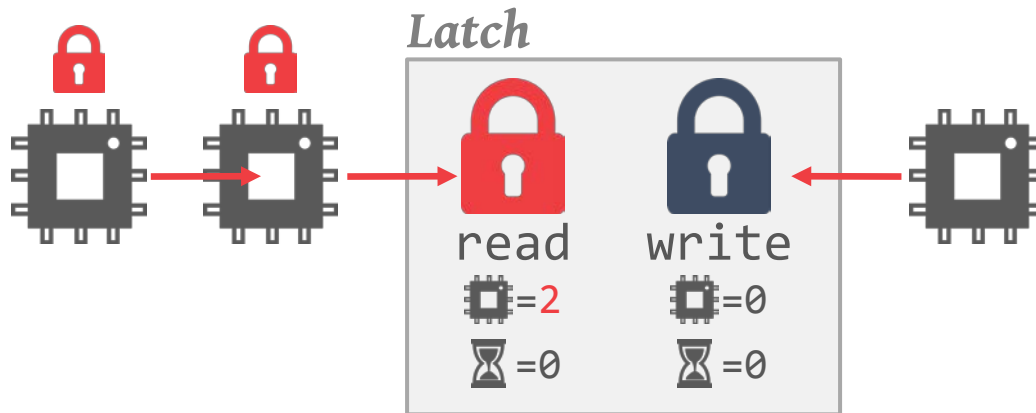
- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



# LATCH IMPLEMENTATIONS

## Choice #4: Reader-Writer Locks

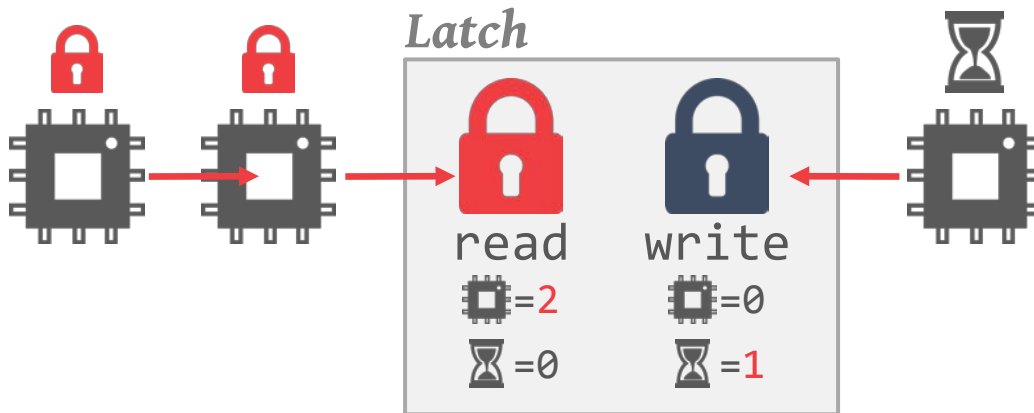
- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



# LATCH IMPLEMENTATIONS

## Choice #4: Reader-Writer Locks

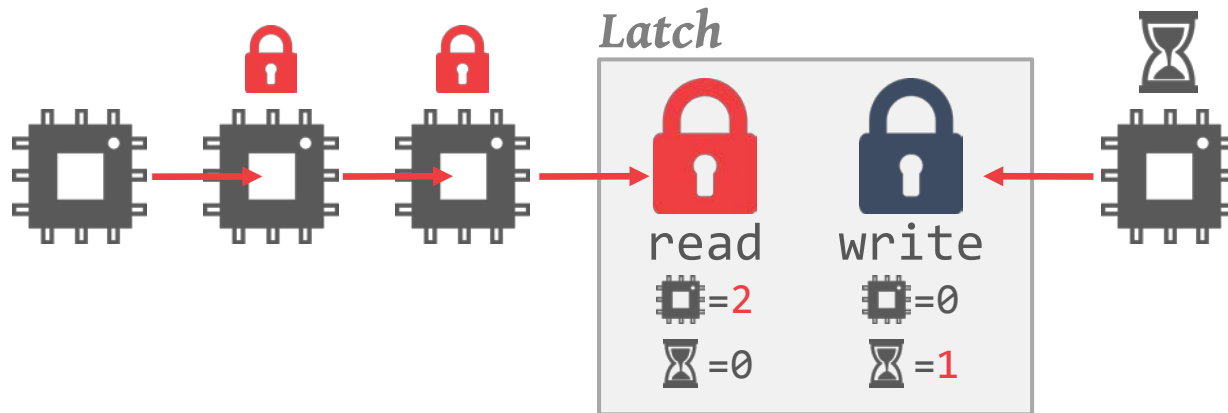
- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



# LATCH IMPLEMENTATIONS

## Choice #4: Reader-Writer Locks

- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks

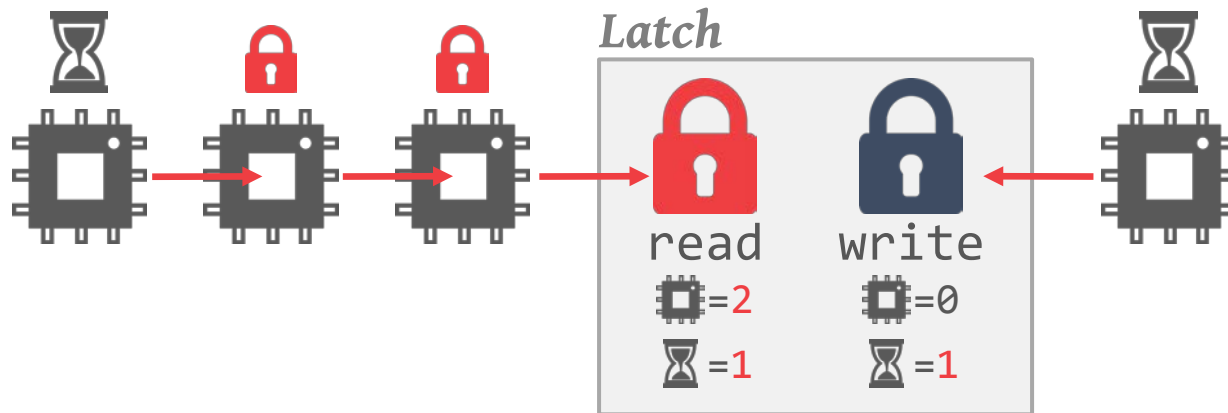




# LATCH IMPLEMENTATIONS

## Choice #4: Reader-Writer Locks

- Allows for concurrent readers
- Have to manage read/write queues to avoid starvation
- Can be implemented on top of spinlocks



# MODERN INDEXES

---

Bw-Tree (Hekaton)

Concurrent Skip Lists (MemSQL)

ART Index (HyPer)

# 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.

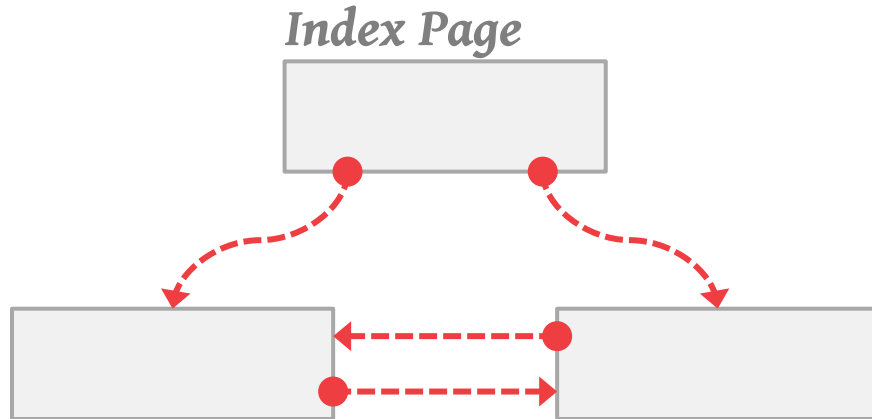



THE BW-TREE: A B-TREE FOR NEW HARDWARE  
*ICDE 2013*


# BW-TREE: MAPPING TABLE

## Mapping Table

<i>PID</i>	<i>Addr</i>
101	
102	
103	
104	



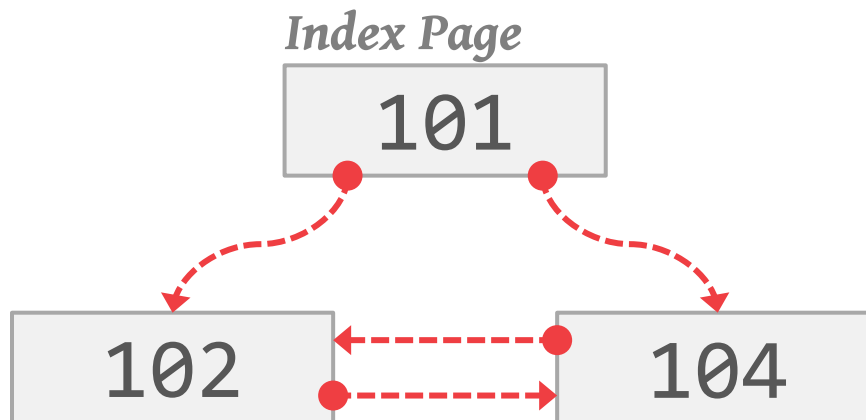
*Logical  
Pointer* 


*Physical  
Pointer* 


# BW-TREE: MAPPING TABLE

## Mapping Table

<i>PID</i>	<i>Addr</i>
101	
102	
103	
104	



*Logical  
Pointer* 

*Physical  
Pointer* 

# BW-TREE: MAPPING TABLE

## Mapping Table


PID	Addr
101	●
102	●
103	
104	●


Index Page

101

102

104

Logical  
Pointer 


Physical  
Pointer 


# BW-TREE: MAPPING TABLE

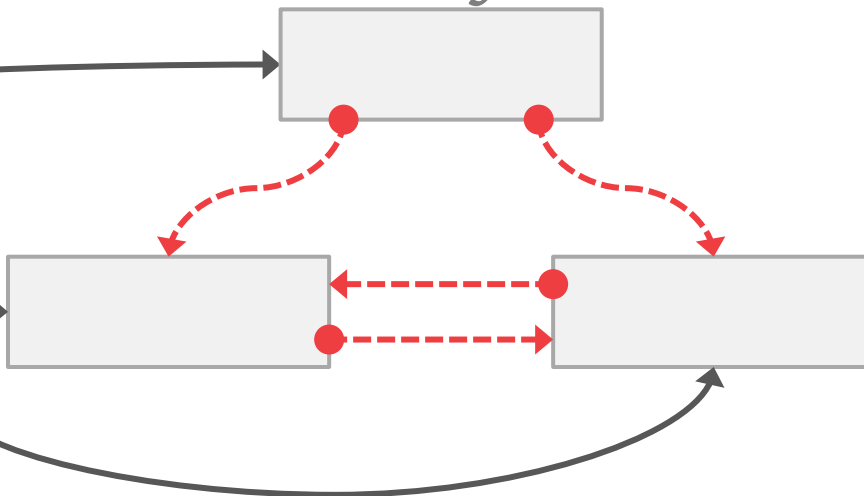
## Mapping Table

<i>PID</i>	<i>Addr</i>
101	●
102	●
103	
104	●

### Index Page

*Logical  
Pointer* 

*Physical  
Pointer* 



# BW-TREE: MAPPING TABLE

## Mapping Table

<i>PID</i>	<i>Addr</i>
101	●
102	●
103	
104	●

### Index Page



*Logical Pointer* - - - - ->  
*Physical Pointer* —————>





# BW-TREE: DELTA UPDATES

## Mapping Table

<i>PID</i>	<i>Addr</i>
101	
102	●
103	
104	

Page 102

*Logical  
Pointer* 

*Physical  
Pointer* 


# BW-TREE: DELTA UPDATES


## Mapping Table

<i>PID</i>	<i>Addr</i>
101	
102	●
103	
104	

Each update to a page produces a new delta.

Page 102

*Logical  
Pointer* 

*Physical  
Pointer* 

# BW-TREE: DELTA UPDATES


## Mapping Table


<i>PID</i>	<i>Addr</i>
101	
102	●
103	
104	

▲ Insert 50

Page 102

Each update to a page produces a new delta.

Logical  
Pointer 

Physical  
Pointer 

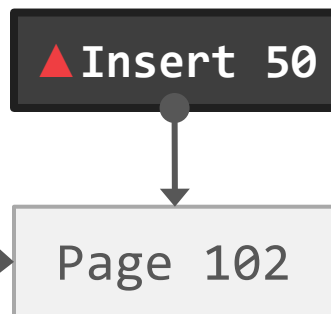
# BW-TREE: DELTA UPDATES

## Mapping Table

<i>PID</i>	<i>Addr</i>
101	
102	●
103	
104	

*Logical  
Pointer* ----->

*Physical  
Pointer* ————>




Each update to a page produces a new delta.


Delta physically points to base page.

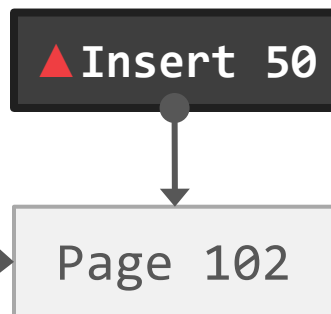
# BW-TREE: DELTA UPDATES

## Mapping Table

<i>PID</i>	<i>Addr</i>
101	
102	
103	
104	

*Logical  
Pointer* 

*Physical  
Pointer* 



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.

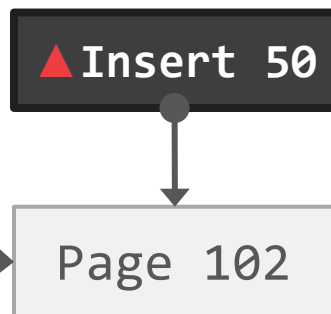
# BW-TREE: DELTA UPDATES

## Mapping Table

PID	Addr
101	
102	
103	
104	

Logical  
Pointer 

Physical  
Pointer 



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

## Mapping Table

<i>PID</i>	<i>Addr</i>
101	
102	
103	
104	

Logical  
Pointer ----->

Physical  
Pointer ----->

▲ Insert 50

Page 102

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.

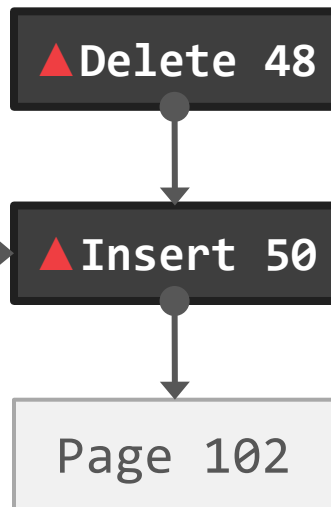
# BW-TREE: DELTA UPDATES

## Mapping Table

PID	Addr
101	
102	
103	
104	

Logical  
Pointer ----->

Physical  
Pointer ————>



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.



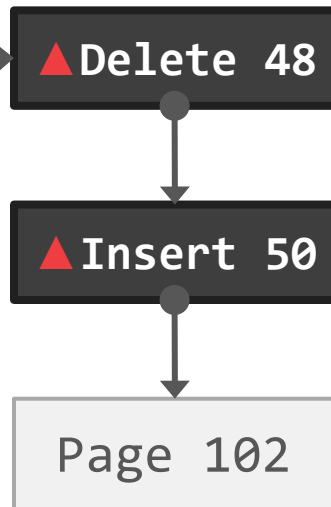
# BW-TREE: DELTA UPDATES

## Mapping Table

PID	Addr
101	
102	
103	
104	

Logical  
Pointer ----->

Physical  
Pointer ————>



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: SEARCH

## Mapping Table

<i>PID</i>	<i>Addr</i>
101	
102	
103	
104	

▲ Delete 48

▲ Insert 50

Page 102

Logical  
Pointer ----->

Physical  
Pointer ----->

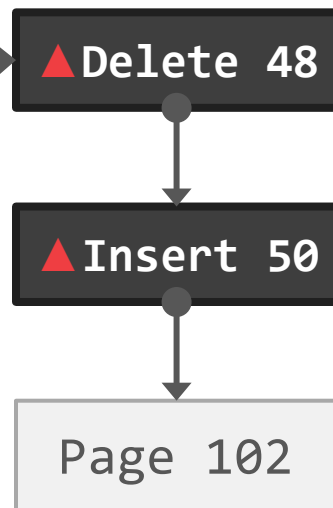
# BW-TREE: SEARCH

## Mapping Table

<i>PID</i>	<i>Addr</i>
101	
102	●
103	
104	

*Logical  
Pointer* ----->

*Physical  
Pointer* —————>



Traverse tree like a regular B+tree.

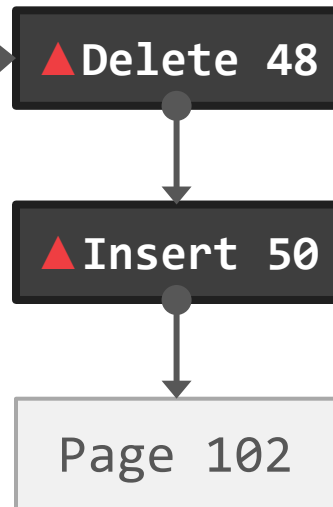
# BW-TREE: SEARCH

## Mapping Table

<i>PID</i>	<i>Addr</i>
101	
102	●
103	
104	

Logical  
Pointer ----->

Physical  
Pointer —————>



Traverse tree like a regular B+tree.

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

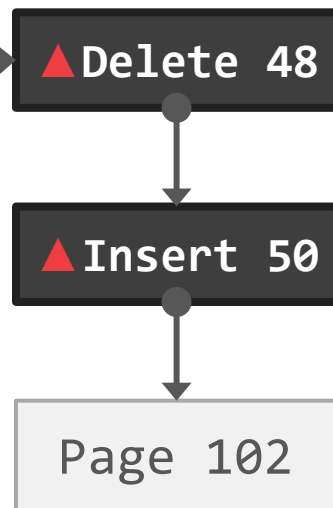
# BW-TREE: SEARCH

## Mapping Table

PID	Addr
101	
102	●
103	
104	

Logical  
Pointer ----->

Physical  
Pointer ————>



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

<i>PID</i>	<i>Addr</i>
101	
102	●
103	
104	

▲ Insert 50

Page 102

Logical  
Pointer ----->

Physical  
Pointer ————>

# BW-TREE: CONTENTION UPDATES

## Mapping Table

<i>PID</i>	<i>Addr</i>
101	
102	●
103	
104	

▲ Insert 50

Page 102

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

Logical  
Pointer ---→

Physical  
Pointer →

# BW-TREE: CONTENTION UPDATES

## Mapping Table

<i>PID</i>	<i>Addr</i>
101	
102	●
103	
104	

▲ Insert 50

Page 102

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

Logical  
Pointer →

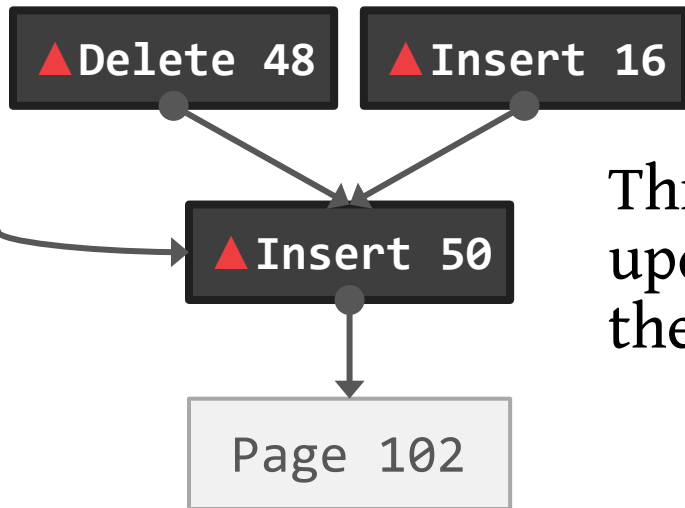
Physical  
Pointer →



# BW-TREE: CONTENTION UPDATES

## Mapping Table

PID	Addr
101	
102	●
103	
104	



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

Logical  
Pointer ----->

Physical  
Pointer ————>

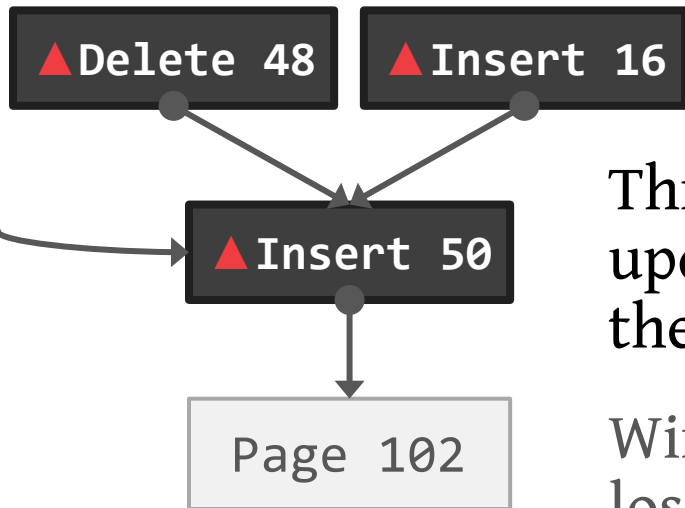
# BW-TREE: CONTENTION UPDATES

## Mapping Table

PID	Addr
101	
102	●
103	
104	

Logical  
Pointer ---→

Physical  
Pointer →



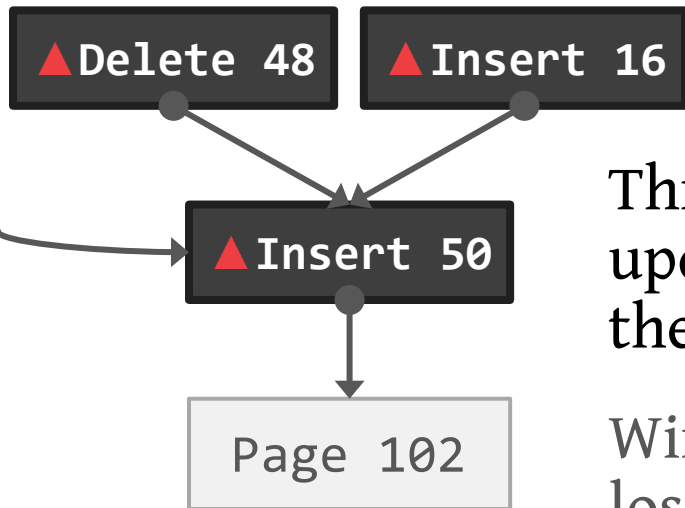
Threads may try to install updates to same state of the page.

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

Logical  
Pointer ----->

Physical  
Pointer ————>

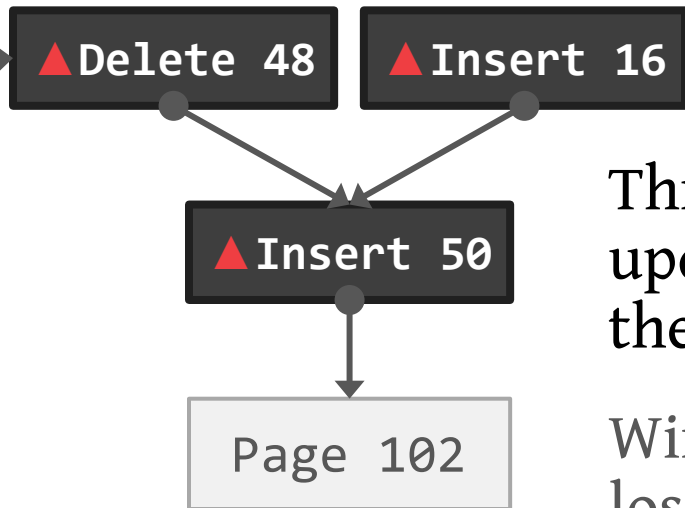
# BW-TREE: CONTENTION UPDATES

## Mapping Table

PID	Addr
101	
102	
103	
104	

Logical  
Pointer ----->

Physical  
Pointer —————>



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

Winner succeeds, any losers must retry or abort

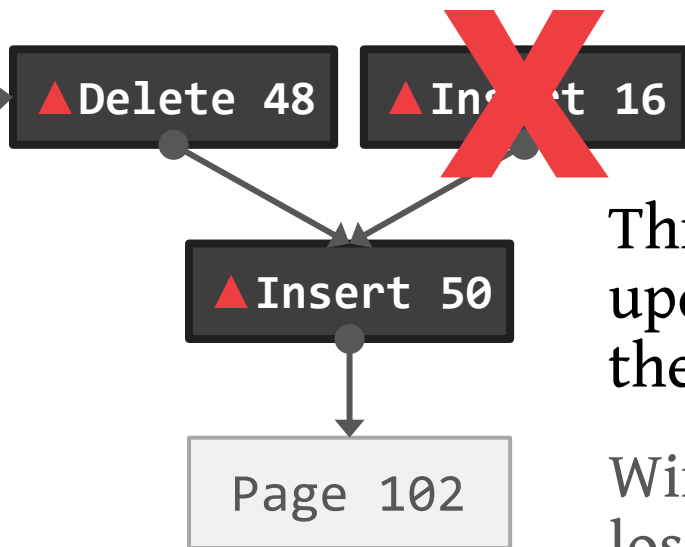
# BW-TREE: CONTENTION UPDATES

## Mapping Table

PID	Addr
101	
102	
103	
104	

Logical  
Pointer ----->

Physical  
Pointer —————>



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

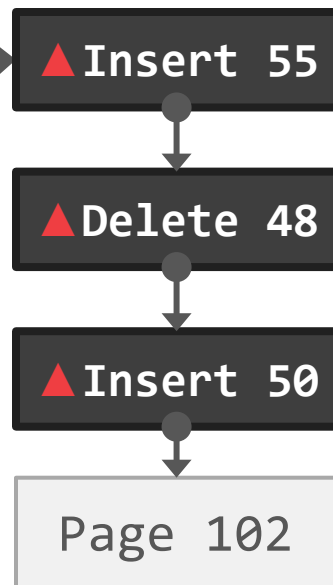
# BW-TREE: CONSOLIDATION

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	
102	
103	
104	

*Logical  
Pointer* 

*Physical  
Pointer* 



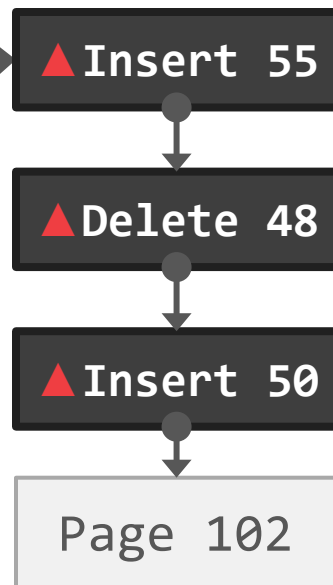
# BW-TREE: CONSOLIDATION

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	
102	
103	
104	

*Logical  
Pointer* ----->

*Physical  
Pointer* —————>



Consolidate updates by creating new page with deltas applied.



# BW-TREE: CONSOLIDATION

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	
102	
103	
104	

▲ Insert 55

▲ Delete 48

▲ Insert 50

Page 102

Consolidate updates by creating new page with deltas applied.

*Logical  
Pointer* ----->

*Physical  
Pointer* ————>

New 102

# BW-TREE: CONSOLIDATION

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	
102	
103	
104	

▲ Insert 55

▲ Delete 48

▲ Insert 50

Page 102

Consolidate updates by creating new page with deltas applied.

*Logical  
Pointer* →

*Physical  
Pointer* →

New 102

▲ Insert 50

# BW-TREE: CONSOLIDATION

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	
102	
103	
104	

▲ Insert 55

▲ Delete 48

▲ Insert 50

Page 102

Consolidate updates by creating new page with deltas applied.

*Logical  
Pointer* →

*Physical  
Pointer* →

New 102

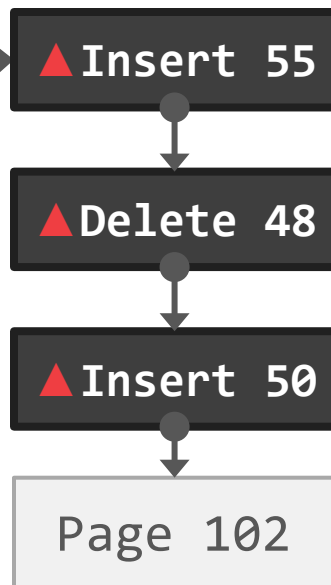
# BW-TREE: CONSOLIDATION

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	
102	
103	
104	

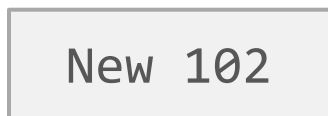
*Logical  
Pointer* ----->

*Physical  
Pointer* —————>



Consolidate updates by creating new page with deltas applied.

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



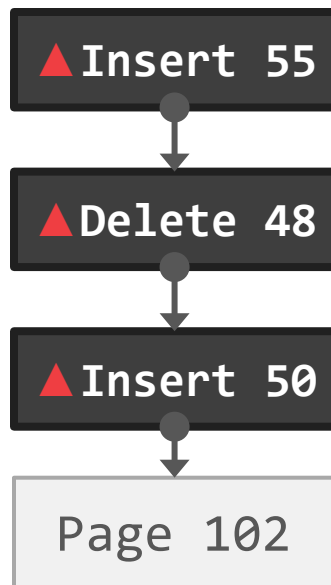
# BW-TREE: CONSOLIDATION

## Mapping Table

PID	Addr
101	
102	●
103	
104	

Logical  
Pointer ----->

Physical  
Pointer ————>



New 102

Consolidate updates by creating new page with deltas applied.

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

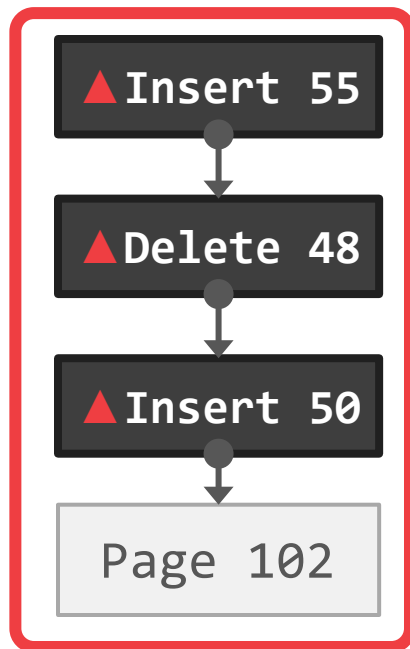
# BW-TREE: CONSOLIDATION

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	
102	●
103	
104	

*Logical  
Pointer* ----->

*Physical  
Pointer* ————>



New 102

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

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	
102	
103	
104	

*Logical  
Pointer* ----->

*Physical  
Pointer* —————>

▲ Insert 55

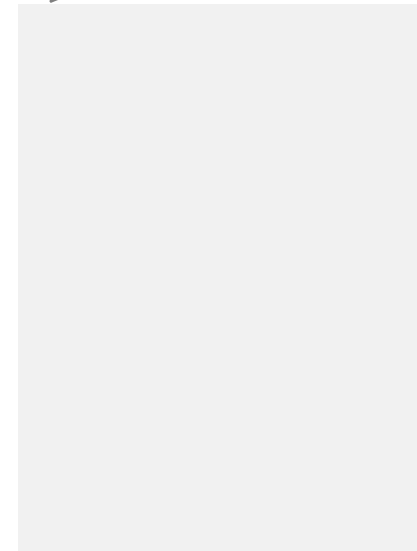
▲ Delete 48

▲ Insert 50

Page 102

New 102

*Epoch Table*





# BW-TREE: GARBAGE COLLECTION

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	
102	
103	
104	

*Logical  
Pointer* ----->

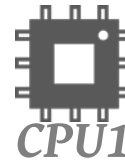
*Physical  
Pointer* —————>

▲ Insert 55

▲ Delete 48

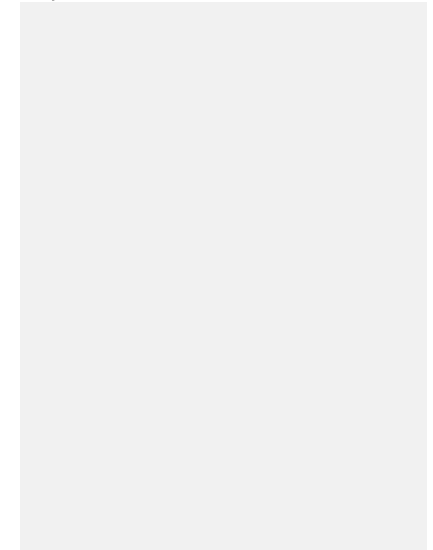
▲ Insert 50

Page 102



New 102

*Epoch Table*



# BW-TREE: GARBAGE COLLECTION

## Mapping Table

PID	Addr
101	
102	
103	
104	

Logical  
Pointer ----->

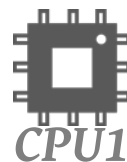
Physical  
Pointer ————>

▲ Insert 55

▲ Delete 48

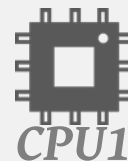
▲ Insert 50

Page 102



New 102

## Epoch Table



# BW-TREE: GARBAGE COLLECTION

## Mapping Table

PID	Addr
101	
102	
103	
104	

Logical  
Pointer ----->

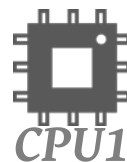
Physical  
Pointer ————>

▲ Insert 55

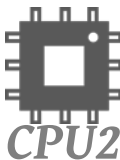
▲ Delete 48

▲ Insert 50

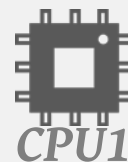
Page 102



New 102



## Epoch Table



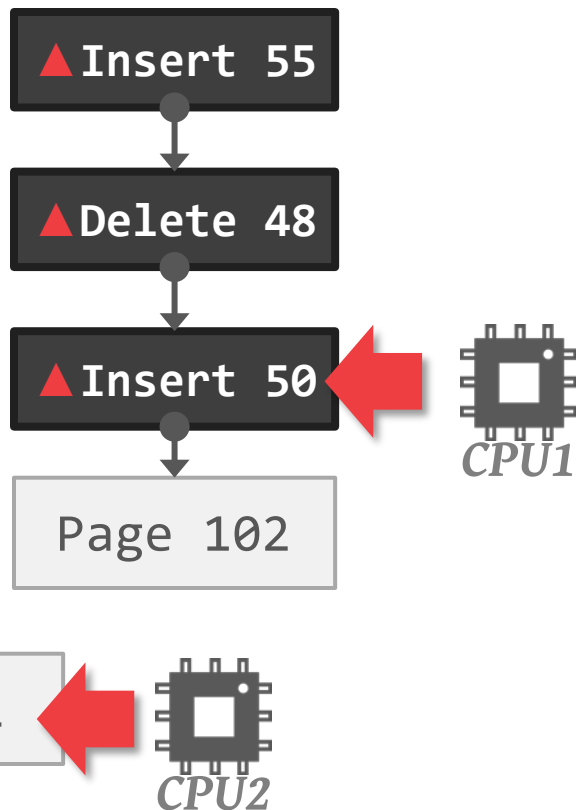
# BW-TREE: GARBAGE COLLECTION

## Mapping Table

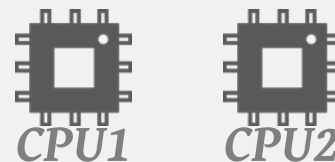
PID	Addr
101	
102	●
103	
104	

Logical  
Pointer ----->

Physical  
Pointer ————>



## Epoch Table



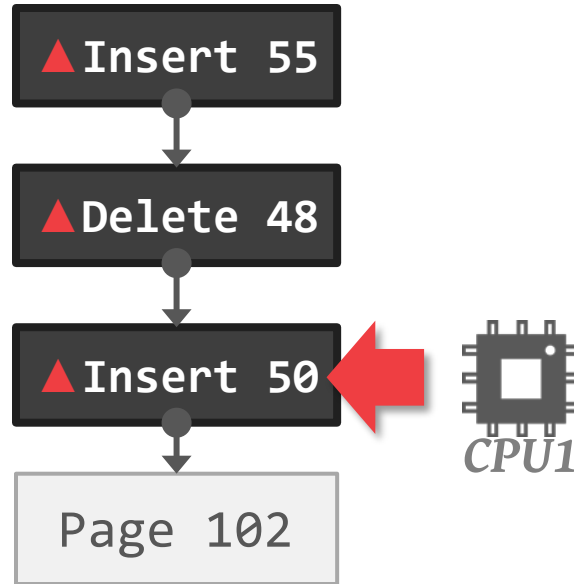
# BW-TREE: GARBAGE COLLECTION

## Mapping Table

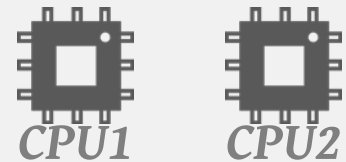
PID	Addr
101	
102	●
103	
104	

Logical  
Pointer ---→

Physical  
Pointer —→





## Epoch Table




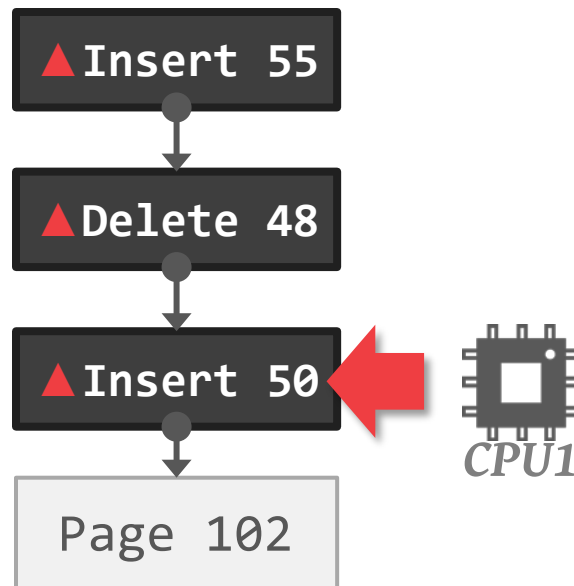
# BW-TREE: GARBAGE COLLECTION

## Mapping Table

PID	Addr
101	
102	
103	
104	

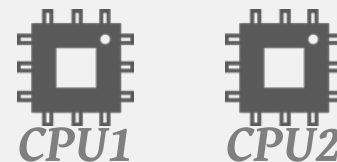
Logical  
Pointer 

Physical  
Pointer 



New 102

## Epoch Table



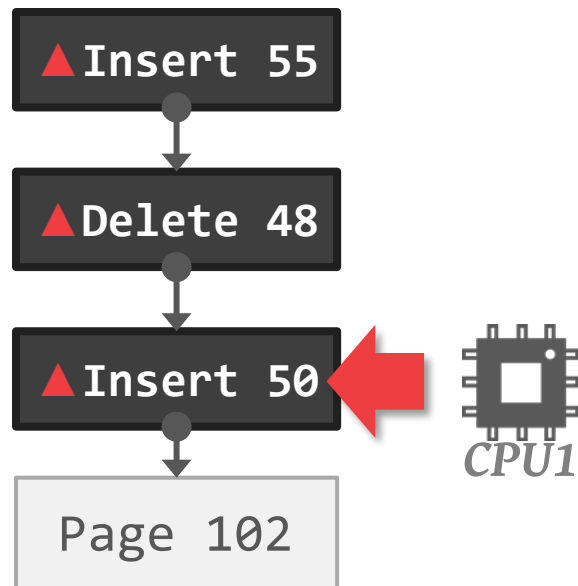
# BW-TREE: GARBAGE COLLECTION

## Mapping Table

PID	Addr
101	
102	●
103	
104	

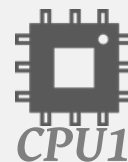
Logical  
Pointer ----->

Physical  
Pointer ————>



New 102

## Epoch Table



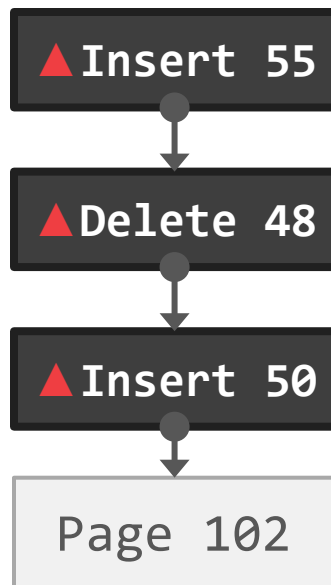
# BW-TREE: GARBAGE COLLECTION

## Mapping Table

PID	Addr
101	
102	●
103	
104	

Logical  
Pointer ----->

Physical  
Pointer —————>



New 102


## Epoch Table







# BW-TREE: GARBAGE COLLECTION

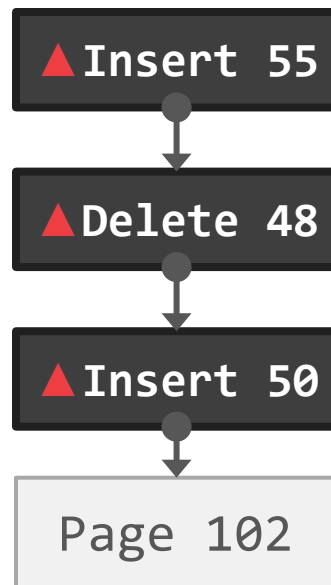
## Mapping Table

PID	Addr
101	
102	
103	
104	

Logical  
Pointer 

Physical  
Pointer 

New 102



## Epoch Table



# BW-TREE: GARBAGE COLLECTION

## Mapping Table

<i>PID</i>	<i>Addr</i>
101	
102	●
103	
104	

Logical  
Pointer ----->

Physical  
Pointer —————>

New 102

## Epoch Table



# BW-TREE: STRUCTURE MODIFICATIONS

---

Page sizes are elastic

- No hard physical threshold for splitting.
- This allows the tree to split a page when convenient.

Tree supports “half-split” without latching

- Install split at child level by creating new page
- Install new separator key and pointer at parent level

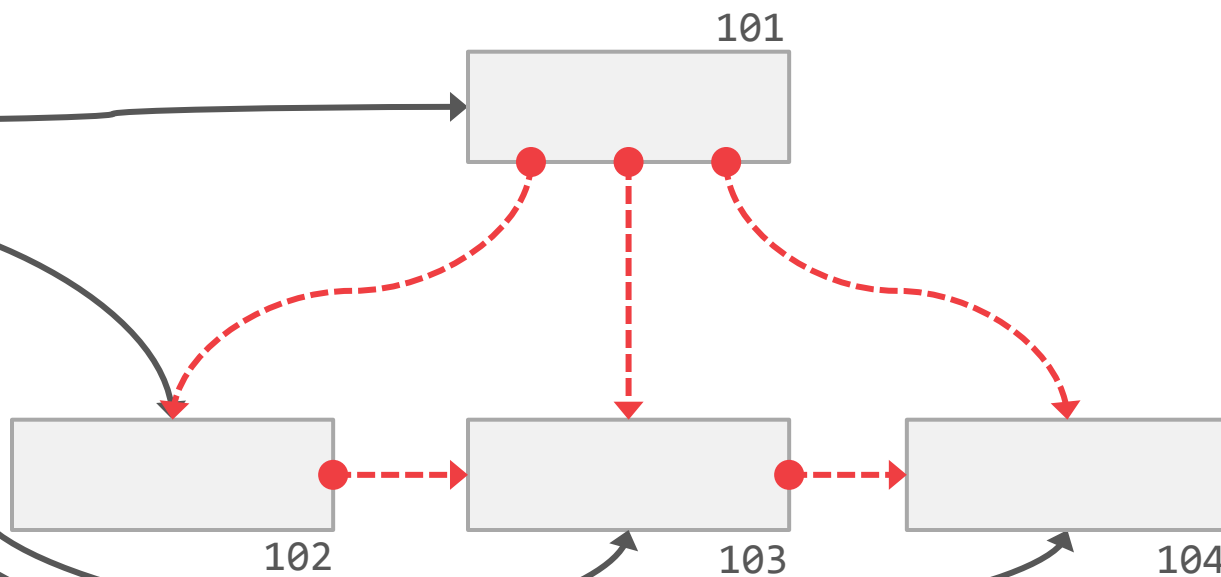
# BW-TREE: STRUCTURE MODIFICATIONS

## Mapping Table

<i>PID</i>	<i>Addr</i>
101	●
102	●
103	●
104	●
105	

Logical  
Pointer - - - - - →


Physical  
Pointer ————— →




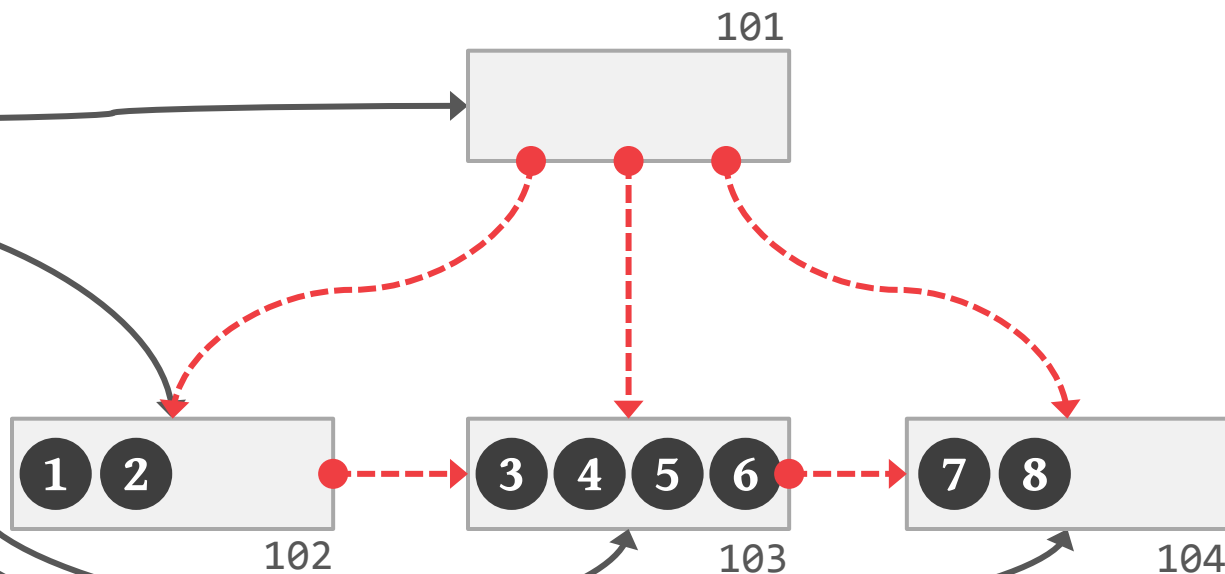
# BW-TREE: STRUCTURE MODIFICATIONS

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	●
102	●
103	●
104	●
105	

*Logical  
Pointer* 

*Physical  
Pointer* 



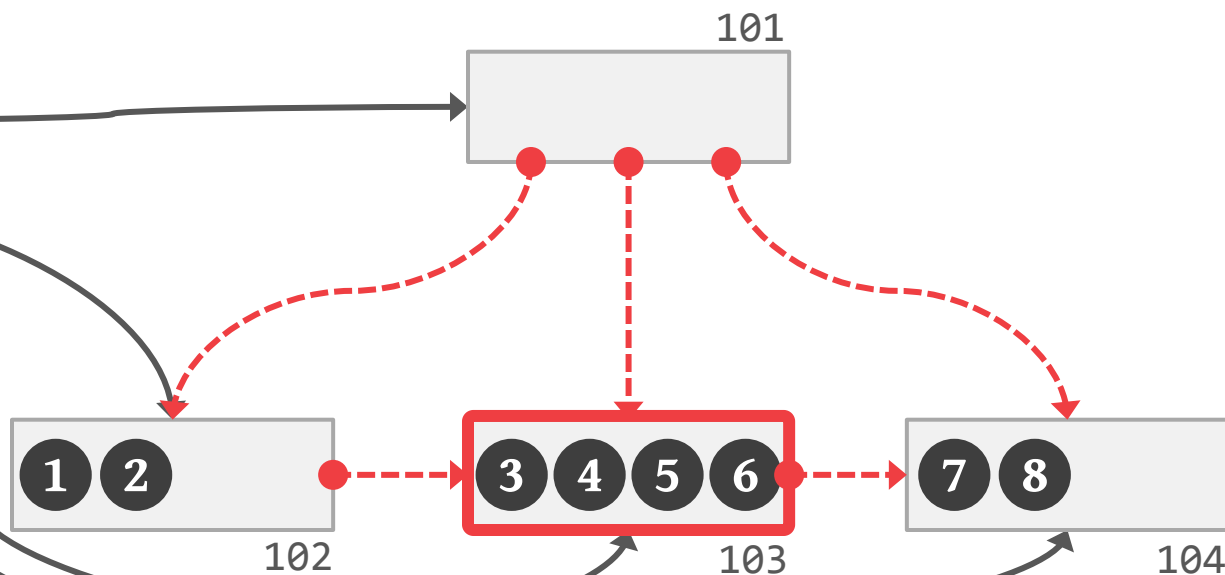
# BW-TREE: STRUCTURE MODIFICATIONS

## Mapping Table

PID	Addr
101	●
102	●
103	●
104	●
105	

Logical  
Pointer - - - - ->


Physical  
Pointer —————>




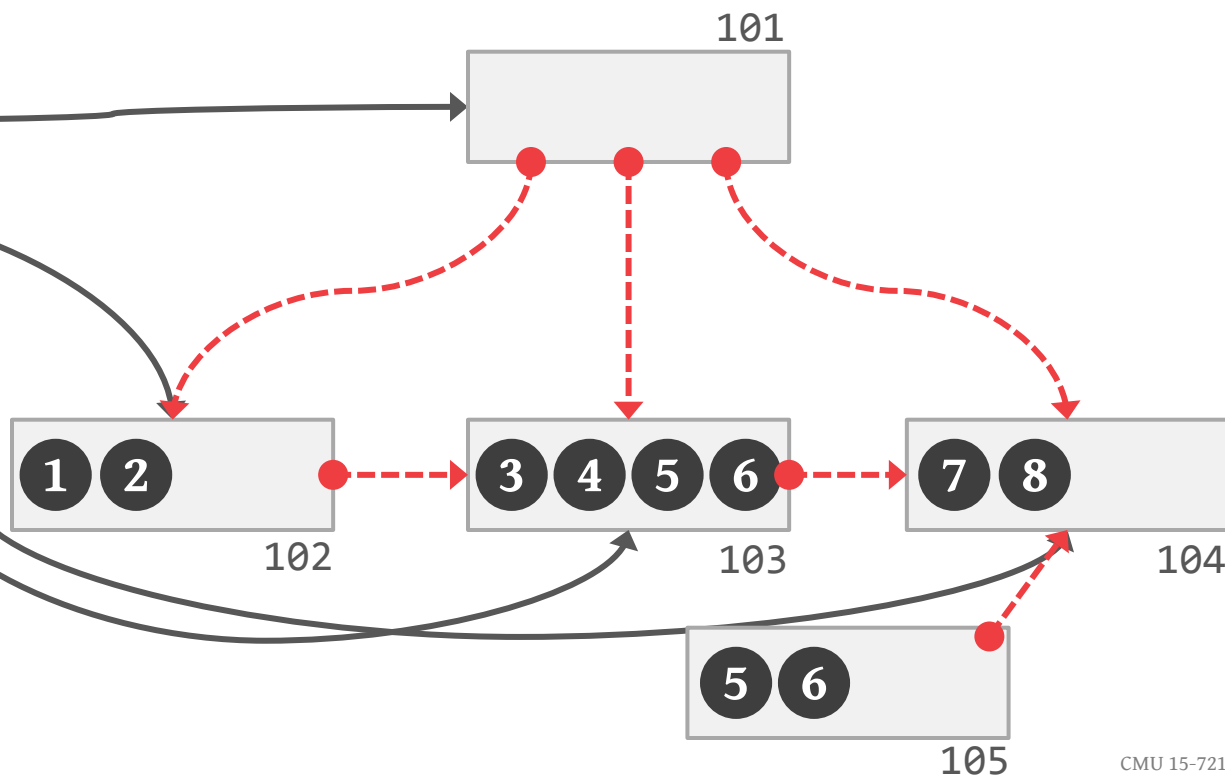
# BW-TREE: STRUCTURE MODIFICATIONS

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	●
102	●
103	●
104	●
105	

*Logical  
Pointer* 

*Physical  
Pointer* 



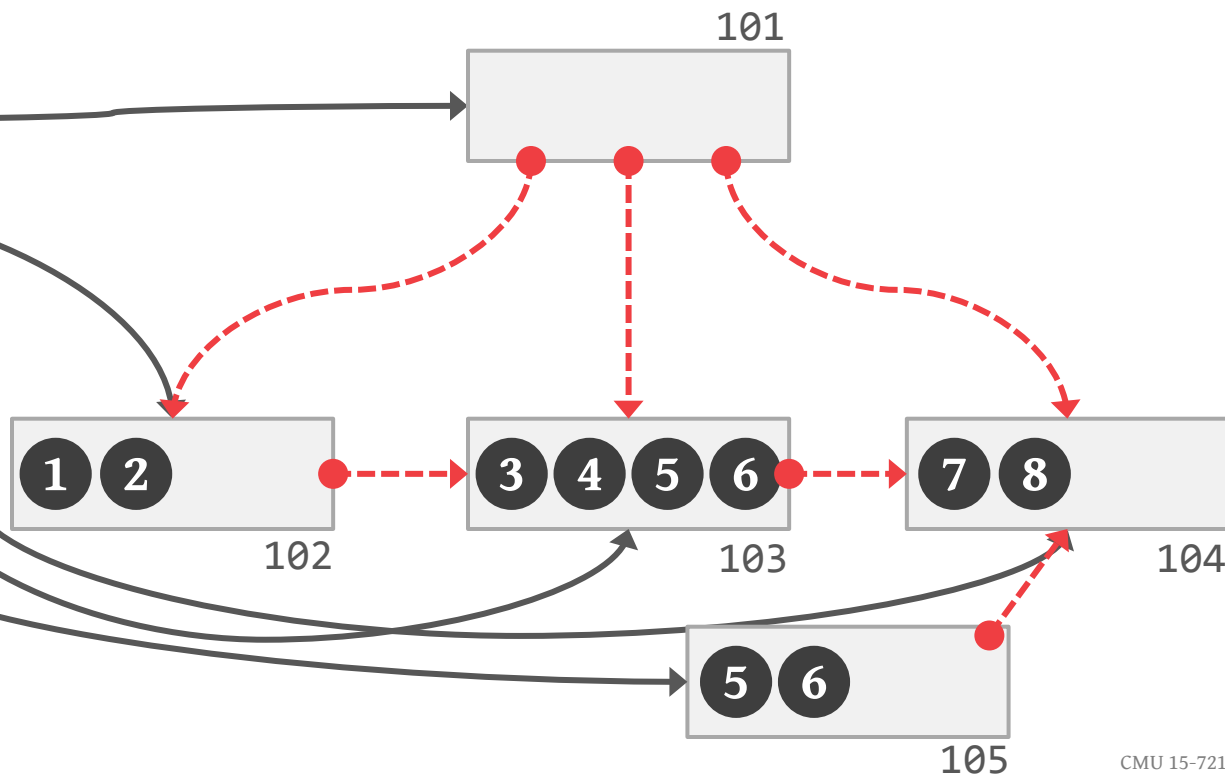
# BW-TREE: STRUCTURE MODIFICATIONS

## Mapping Table

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

Logical  
Pointer ---→

Physical  
Pointer —→





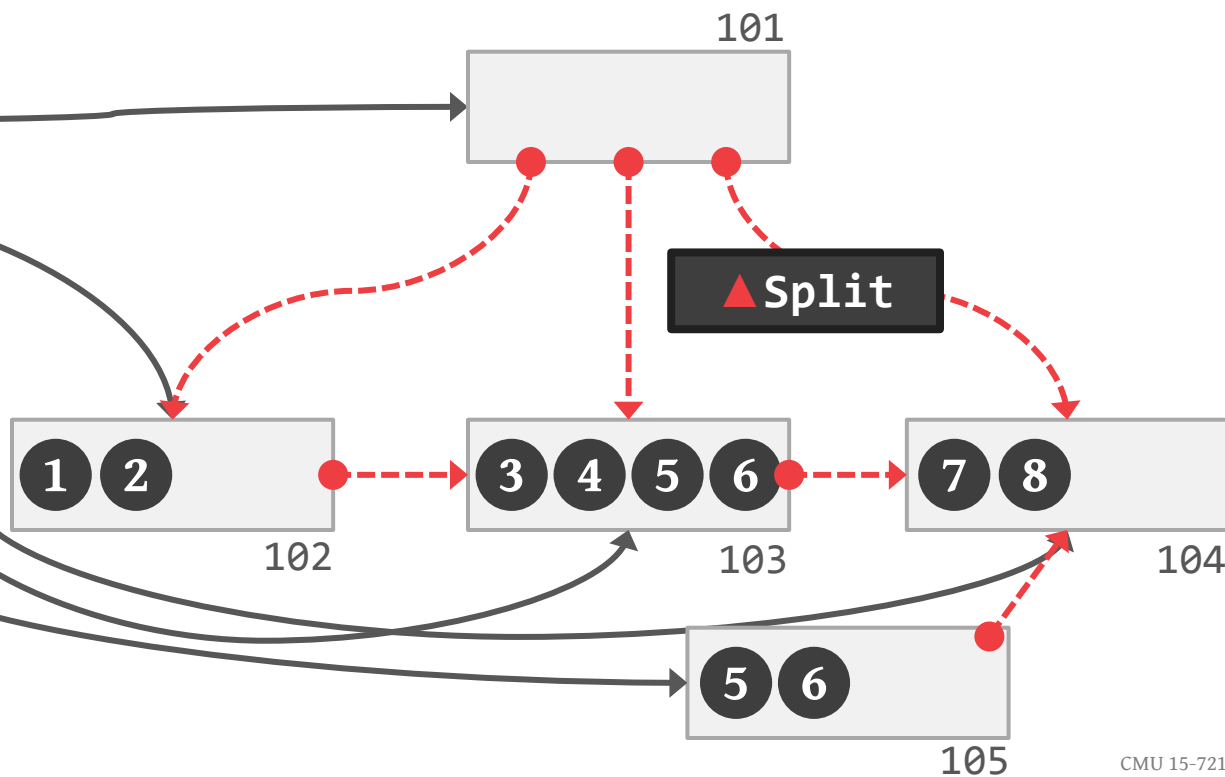
# BW-TREE: STRUCTURE MODIFICATIONS

## Mapping Table

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

Logical  
Pointer - - - - - →


Physical  
Pointer ————— →




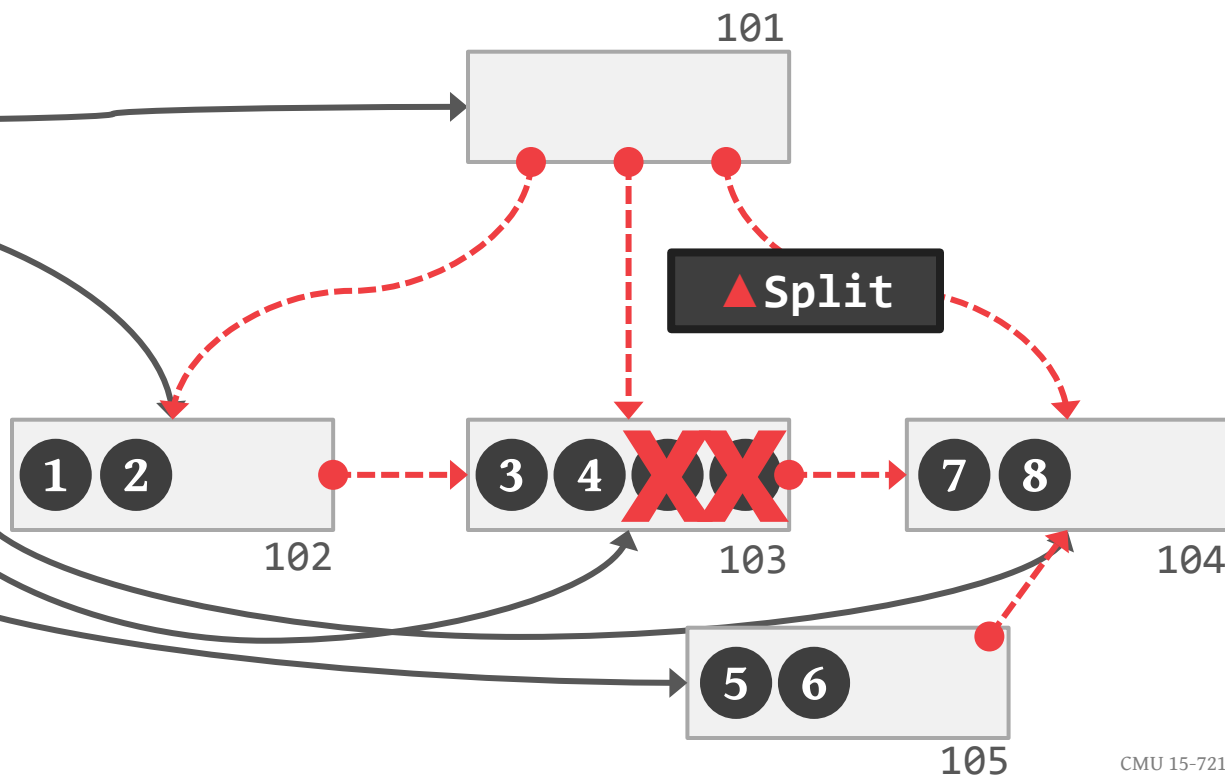
# BW-TREE: STRUCTURE MODIFICATIONS

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	●
102	●
103	●
104	●
105	●

*Logical  
Pointer* 

*Physical  
Pointer* 



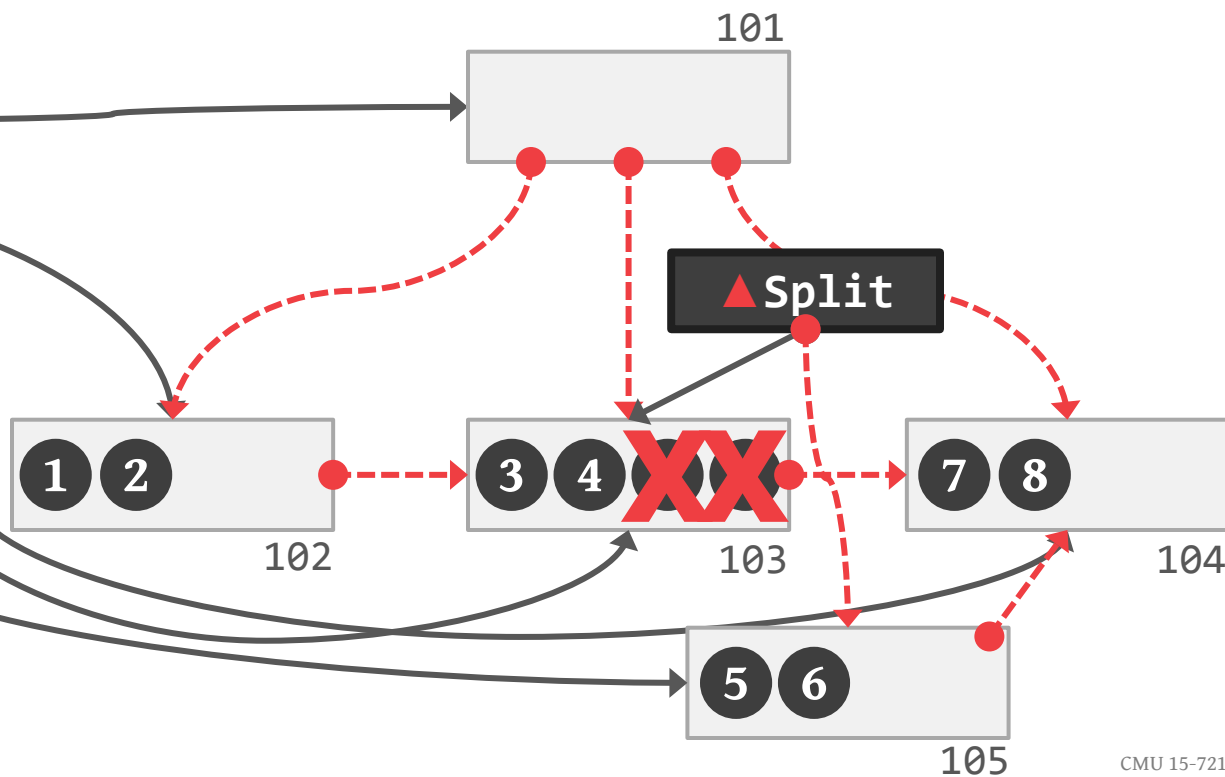
# BW-TREE: STRUCTURE MODIFICATIONS

## Mapping Table

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

Logical  
Pointer - - - - ->

Physical  
Pointer —————>



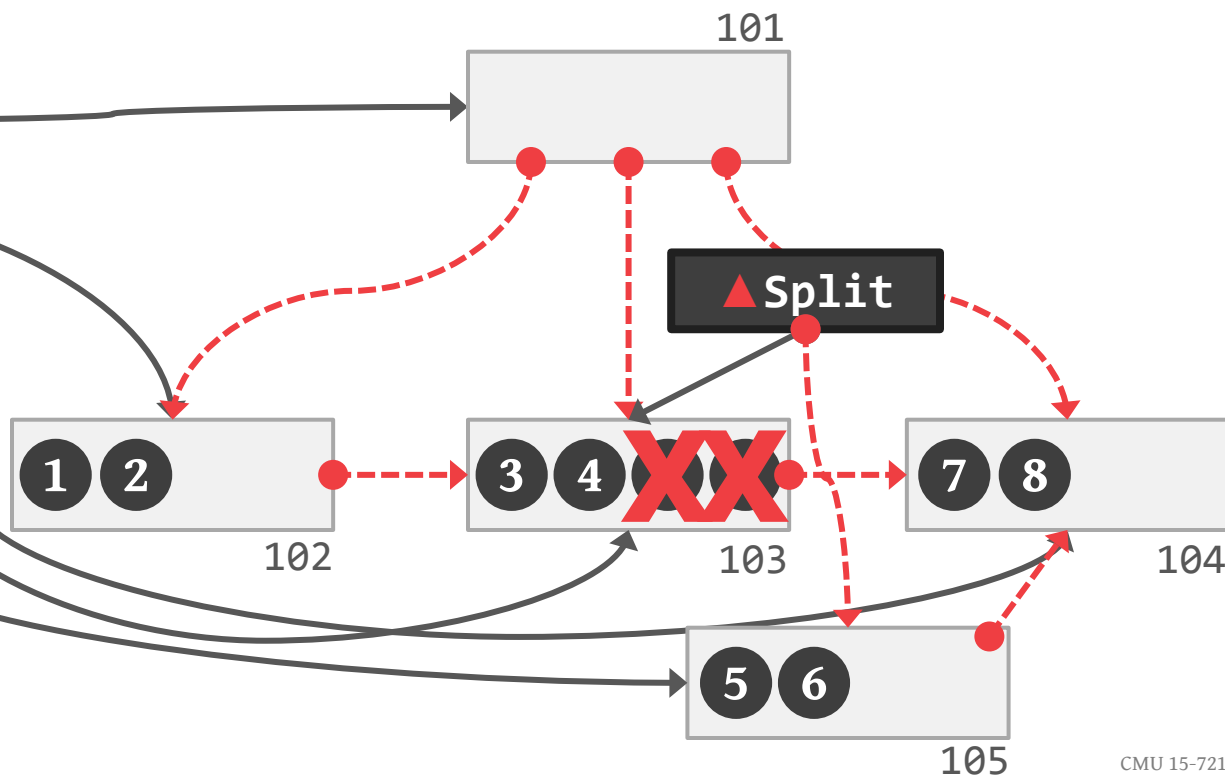
# BW-TREE: STRUCTURE MODIFICATIONS

## Mapping Table

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

Logical  
Pointer ---→


Physical  
Pointer —→




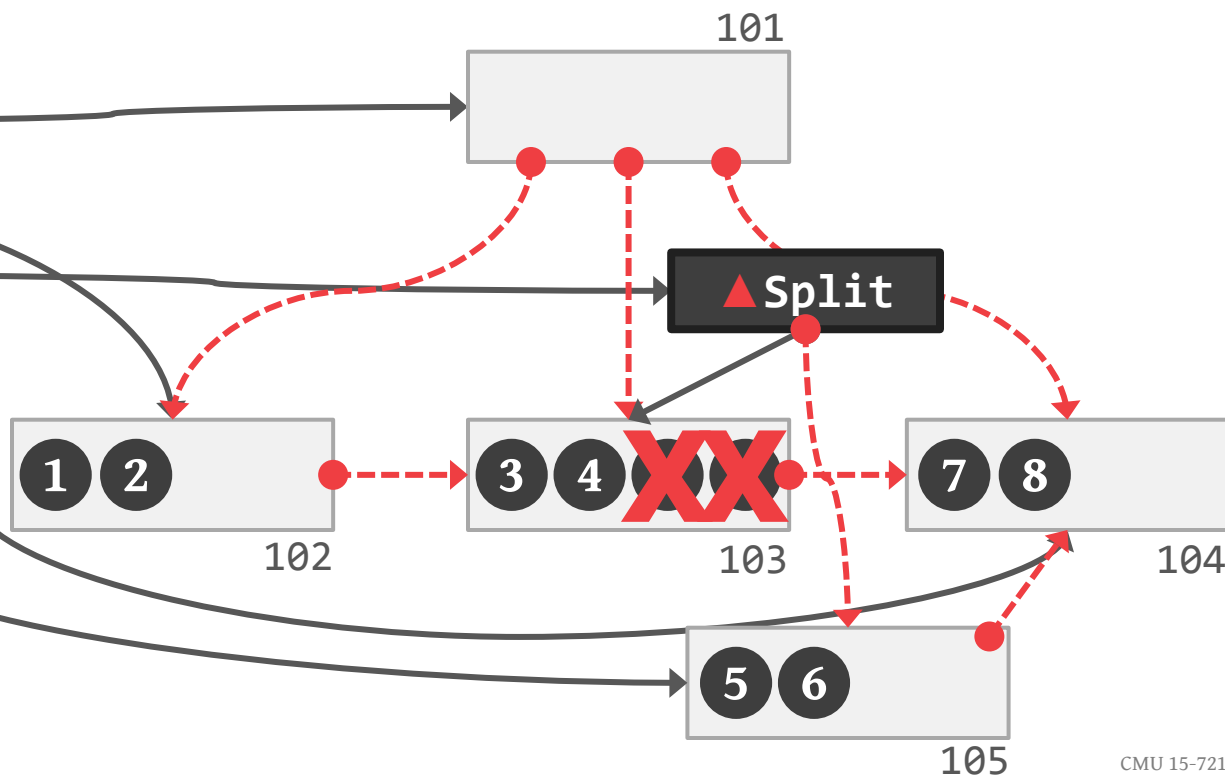
# BW-TREE: STRUCTURE MODIFICATIONS

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	●
102	●
103	●
104	●
105	●

*Logical  
Pointer* 

*Physical  
Pointer* 



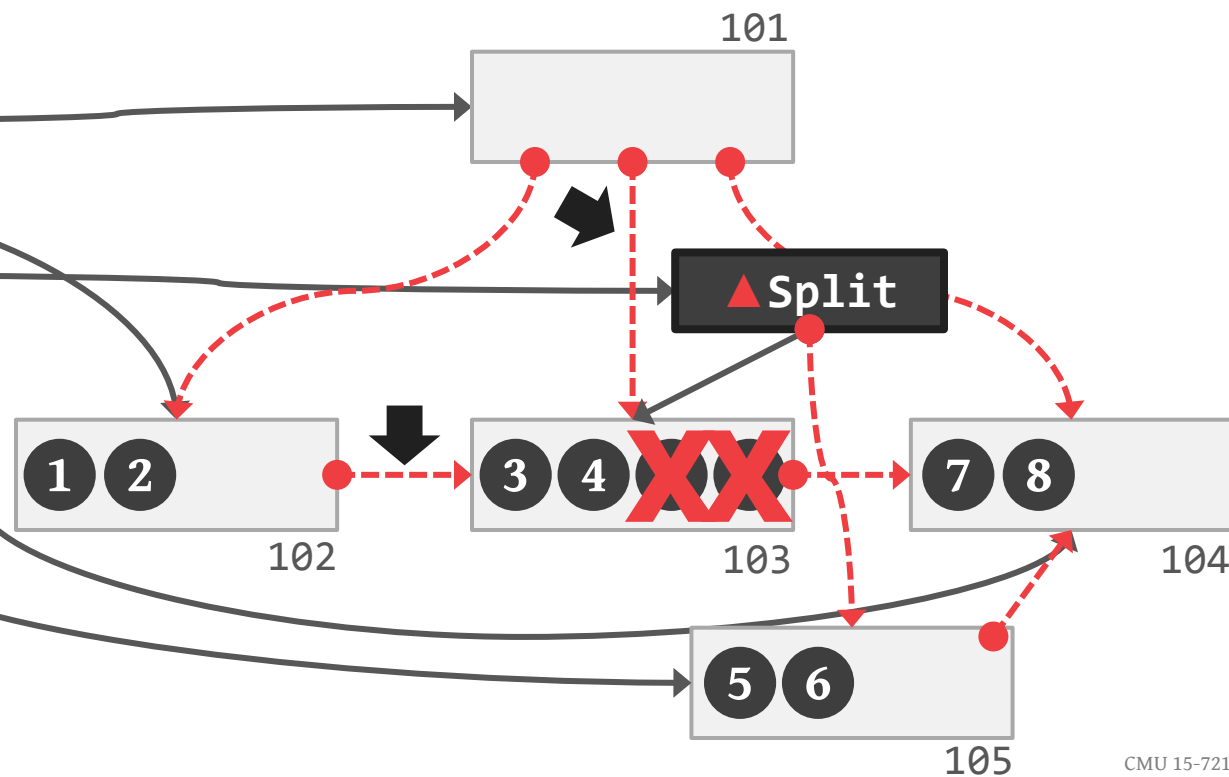
# BW-TREE: STRUCTURE MODIFICATIONS

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	●
102	●
103	●
104	●
105	●

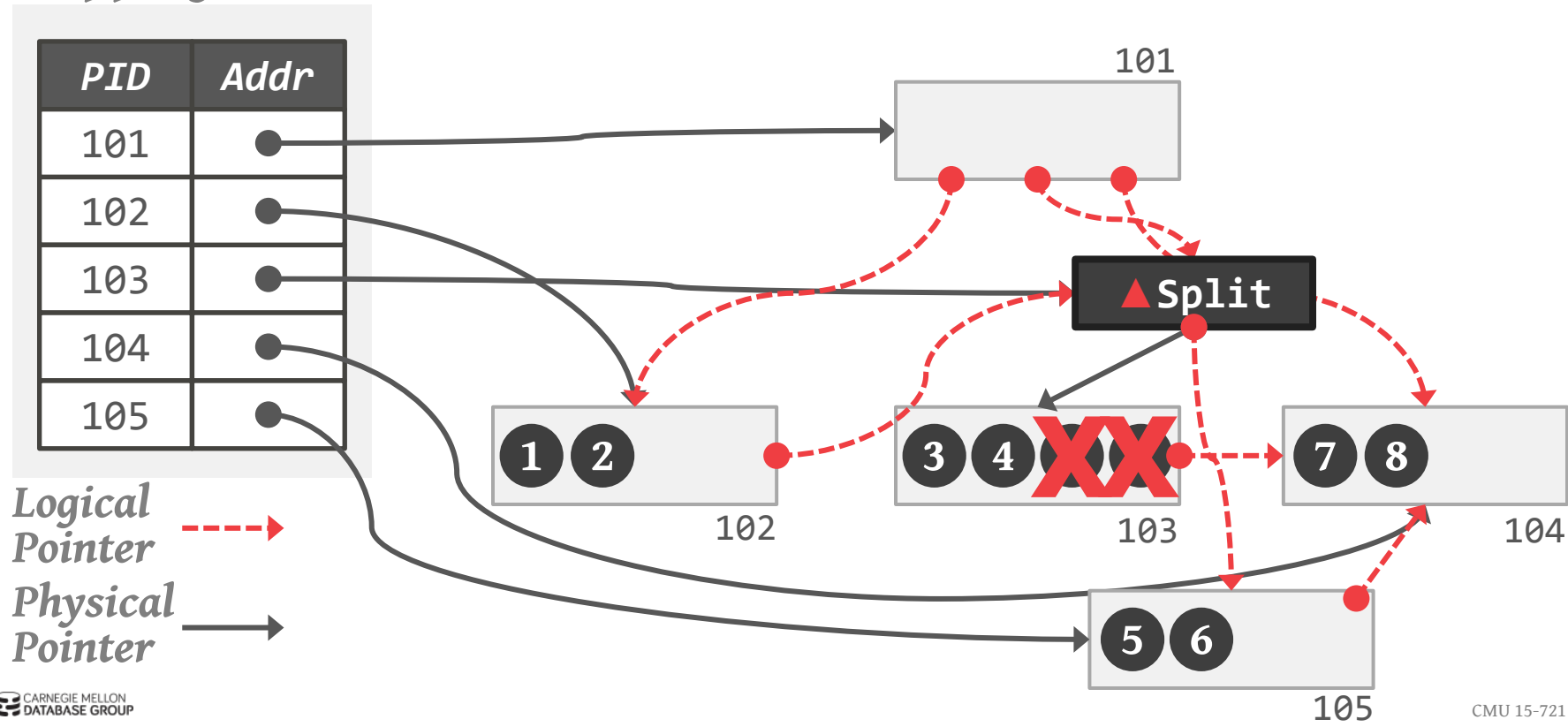
*Logical  
Pointer* ---→

*Physical  
Pointer* →



# BW-TREE: STRUCTURE MODIFICATIONS


## Mapping Table




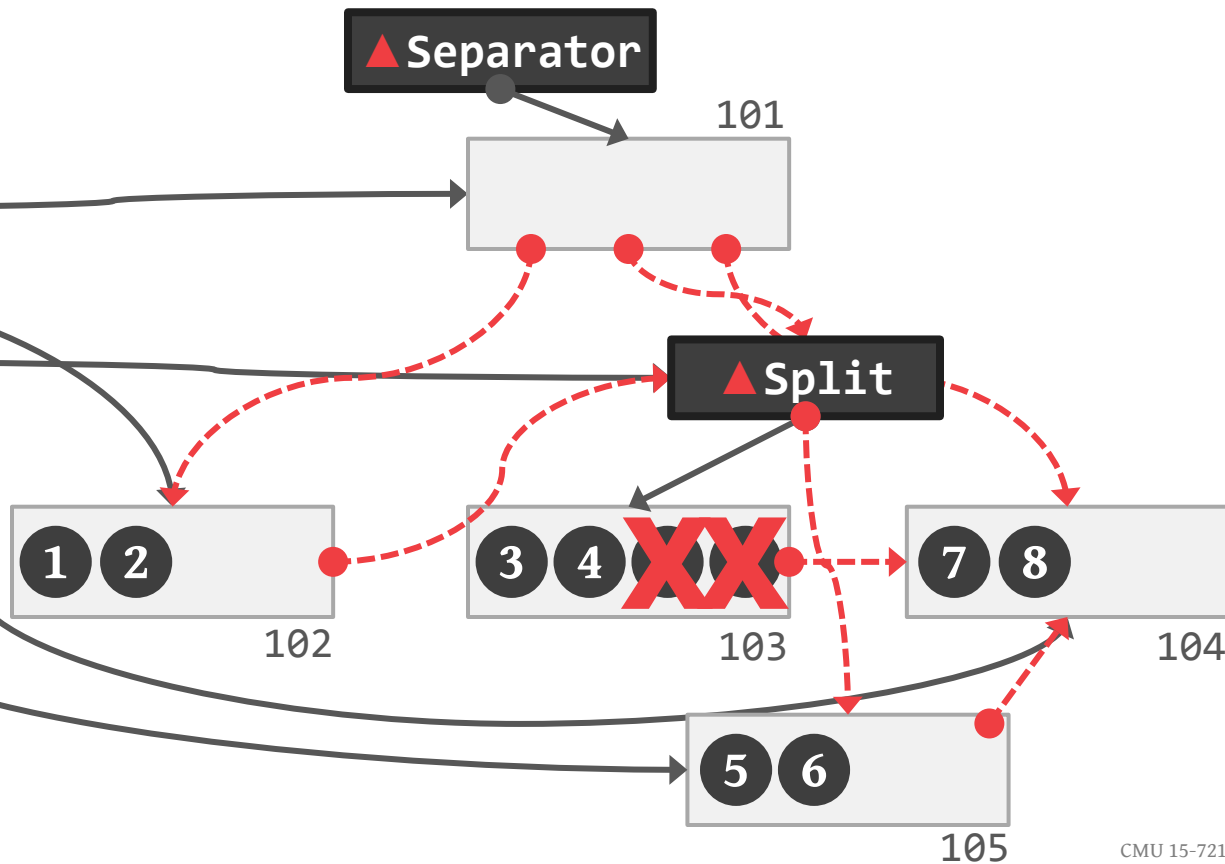
# BW-TREE: STRUCTURE MODIFICATIONS

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	●
102	●
103	●
104	●
105	●

*Logical  
Pointer* 

*Physical  
Pointer* 







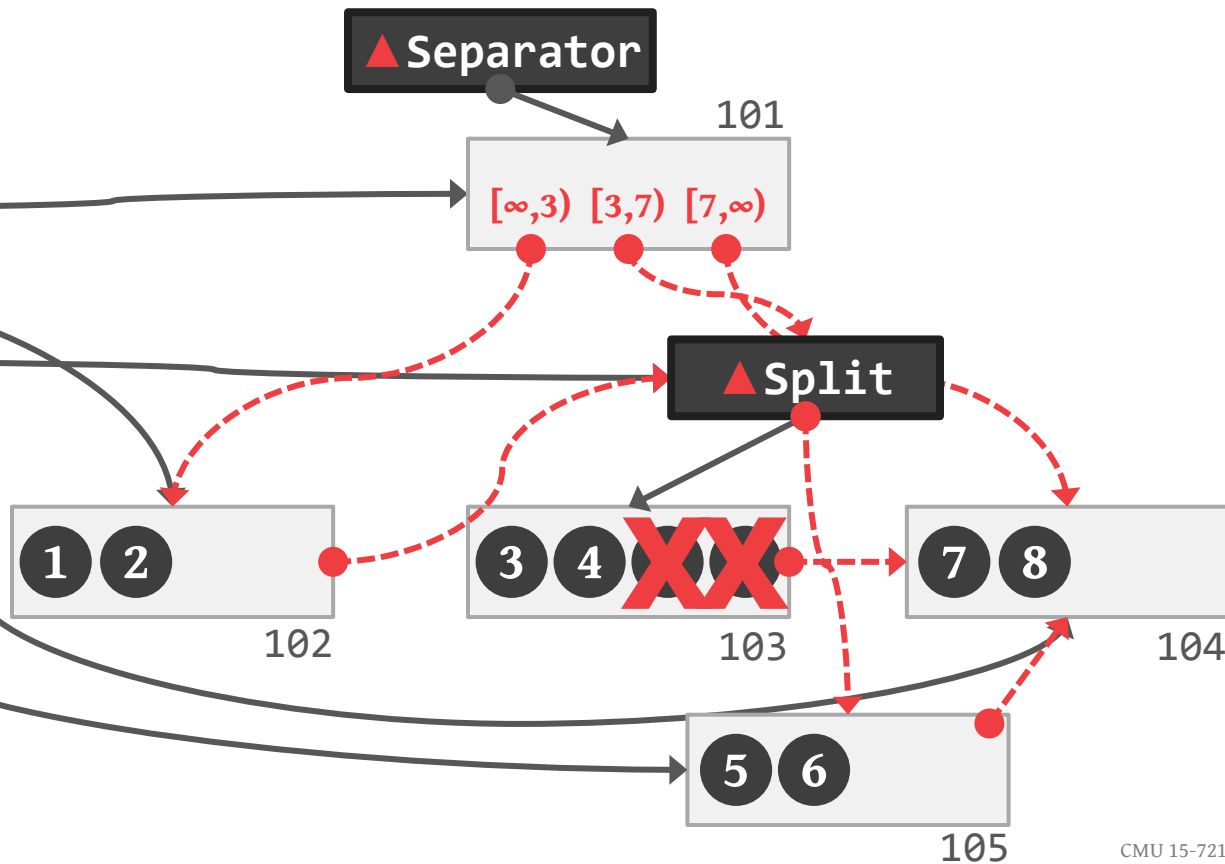
# BW-TREE: STRUCTURE MODIFICATIONS

*Mapping Table*

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

*Logical  
Pointer* 


*Physical  
Pointer* 




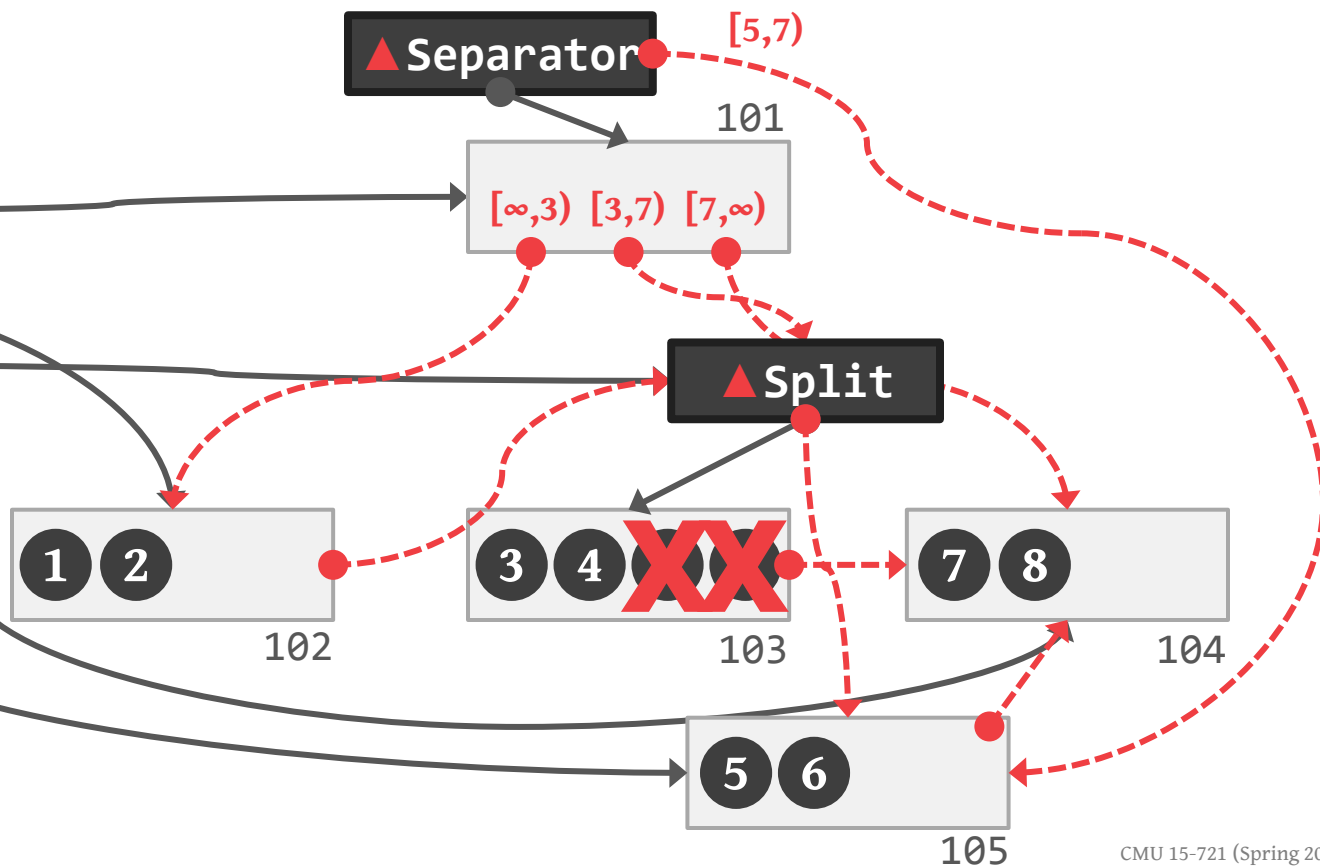
# BW-TREE: STRUCTURE MODIFICATIONS

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	●
102	●
103	●
104	●
105	●

*Logical  
Pointer* 

*Physical  
Pointer* 



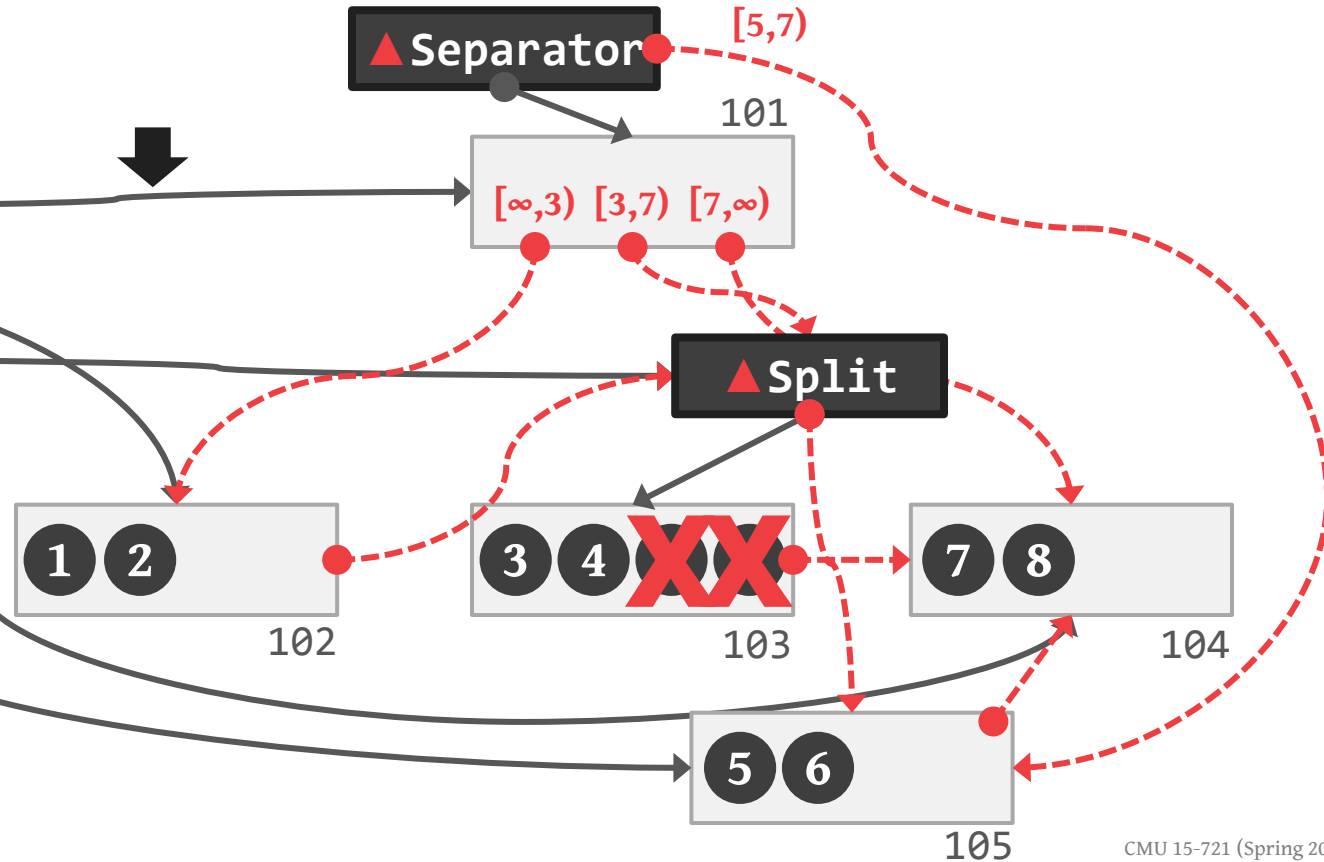
# BW-TREE: STRUCTURE MODIFICATIONS

*Mapping Table*

<i>PID</i>	<i>Addr</i>
101	●
102	●
103	●
104	●
105	●

*Logical  
Pointer* ---


*Physical  
Pointer* —




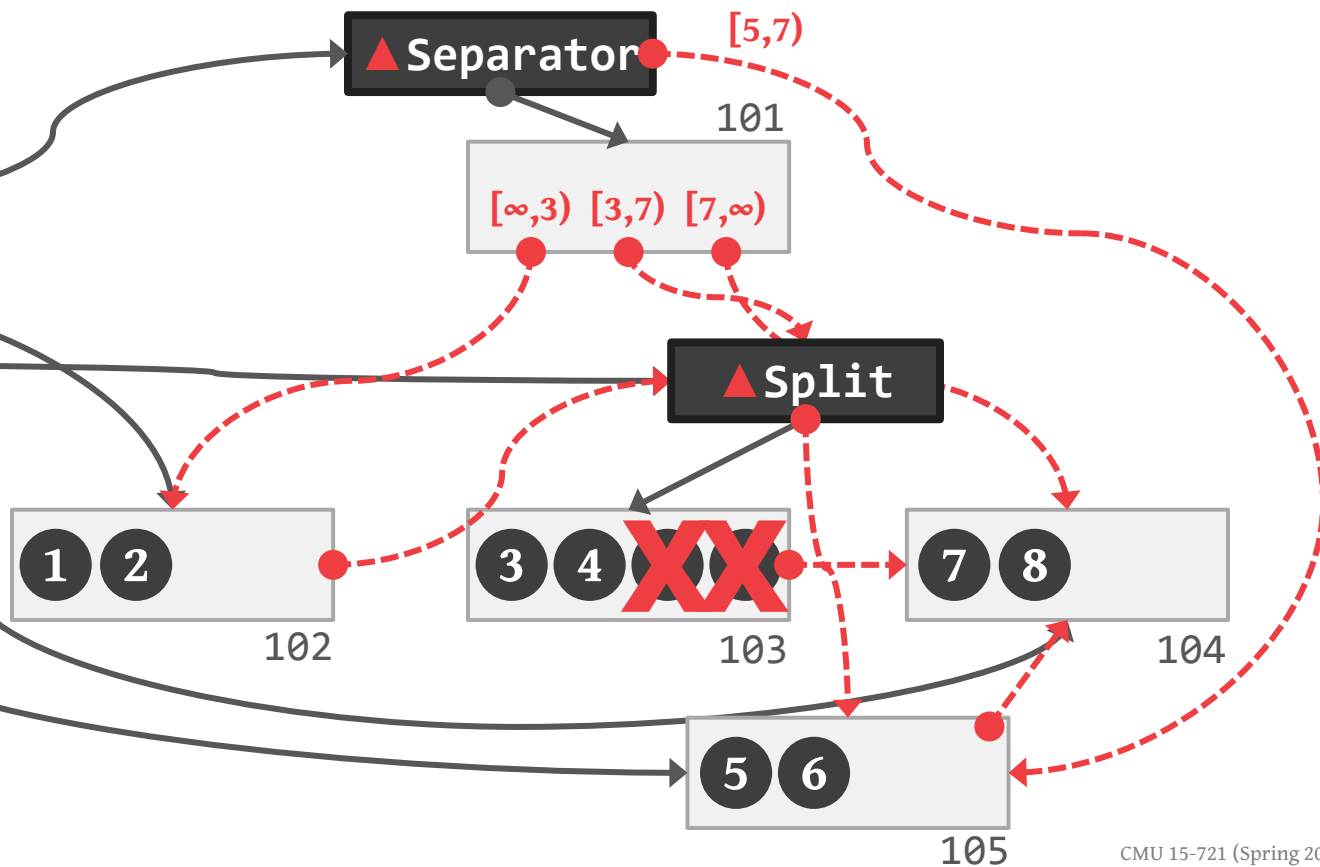
# BW-TREE: STRUCTURE MODIFICATIONS

*Mapping Table*

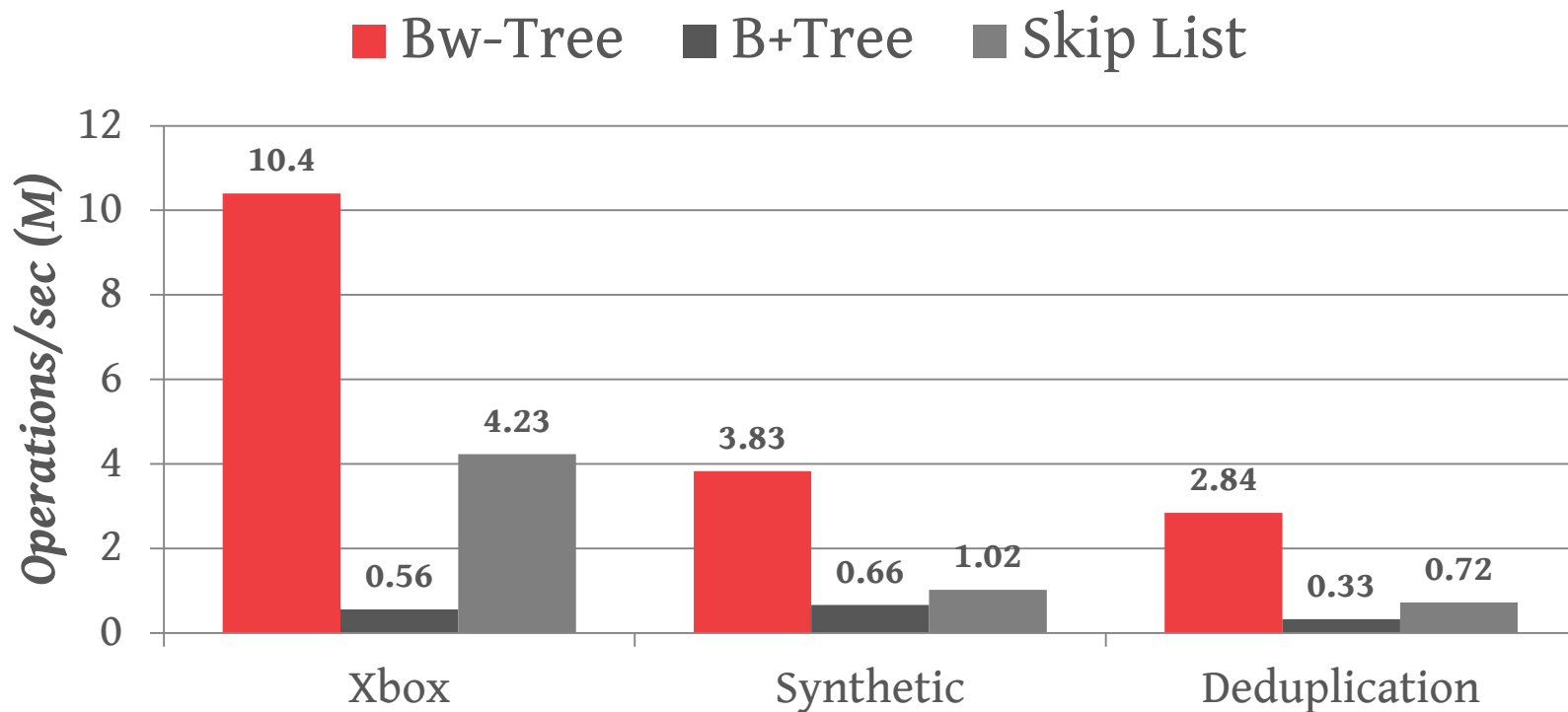
<i>PID</i>	<i>Addr</i>
101	●
102	●
103	●
104	●
105	●

*Logical  
Pointer* 

*Physical  
Pointer* 



# BW-TREE: PERFORMANCE



Source: [Justin Levandoski](#)

CMU 15-721 (Spring 2016)

# CONCURRENT SKIP LIST

---

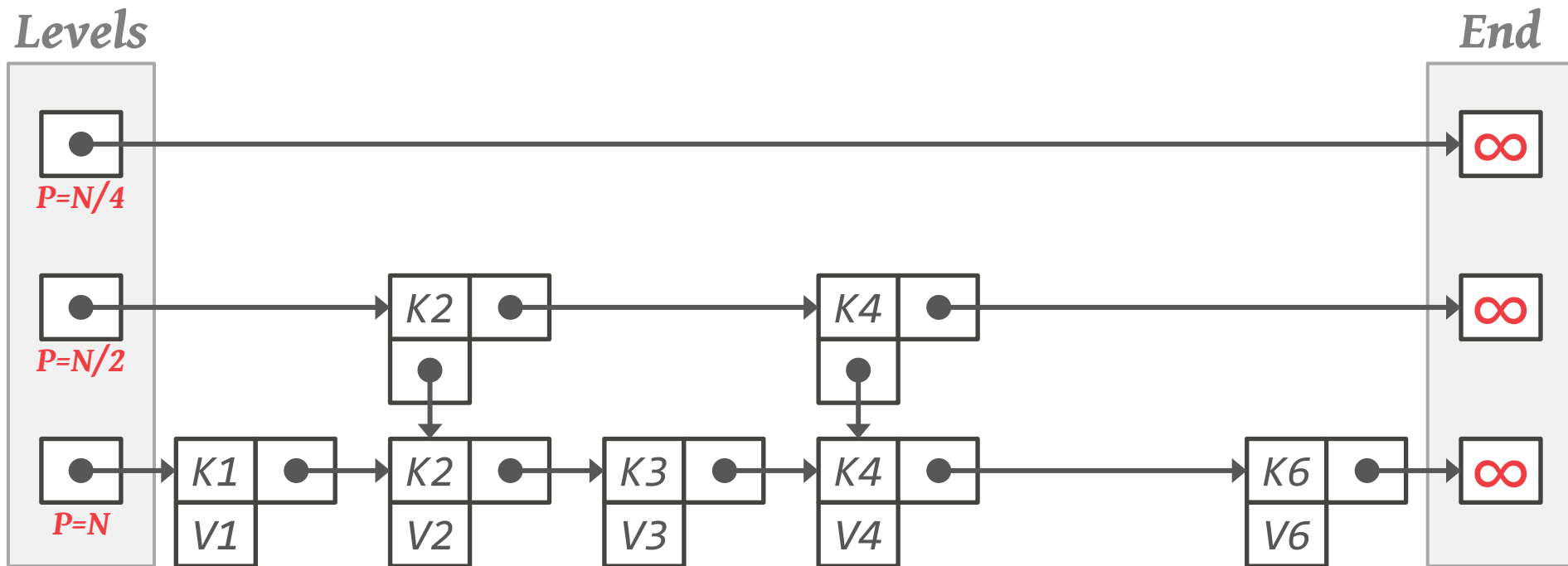
Can implement insert and delete without locks using only CAS operations.

Perform lazy deletion of towers.



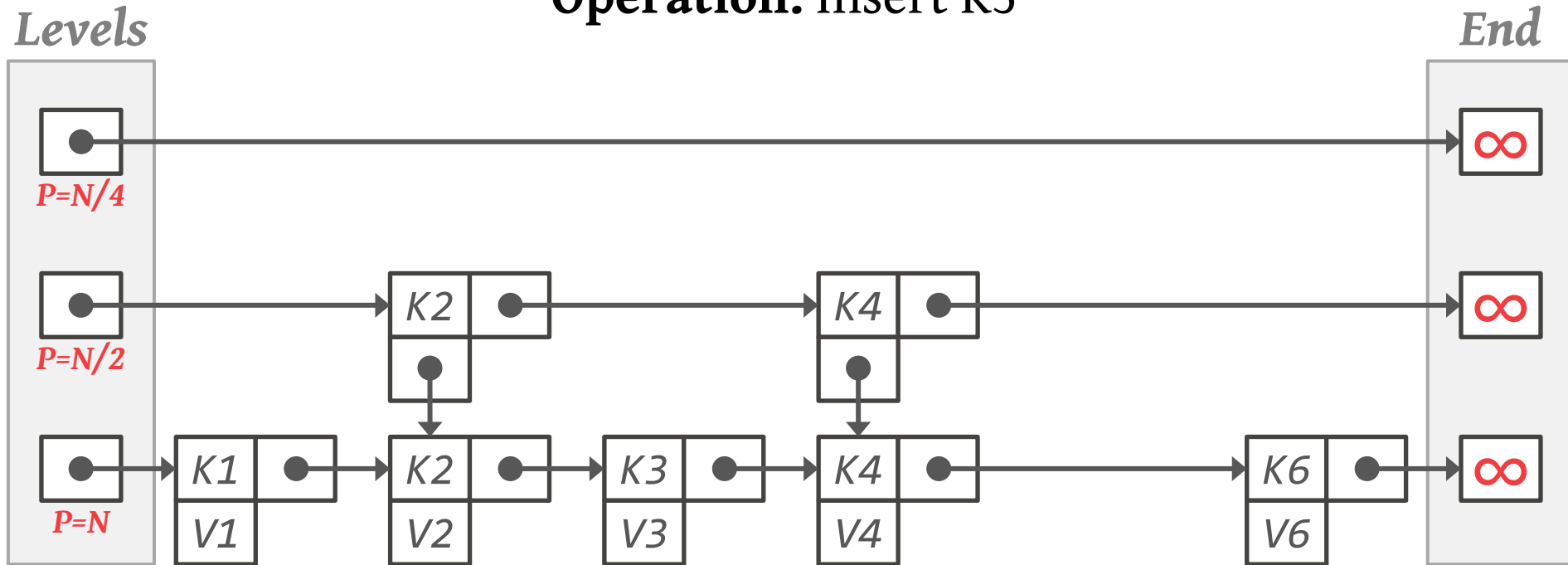
CONCURRENT MAINTENANCE OF SKIP LISTS  
*Univ. of Maryland Tech Report 1990*

# SKIP LISTS: INSERT



# SKIP LISTS: INSERT

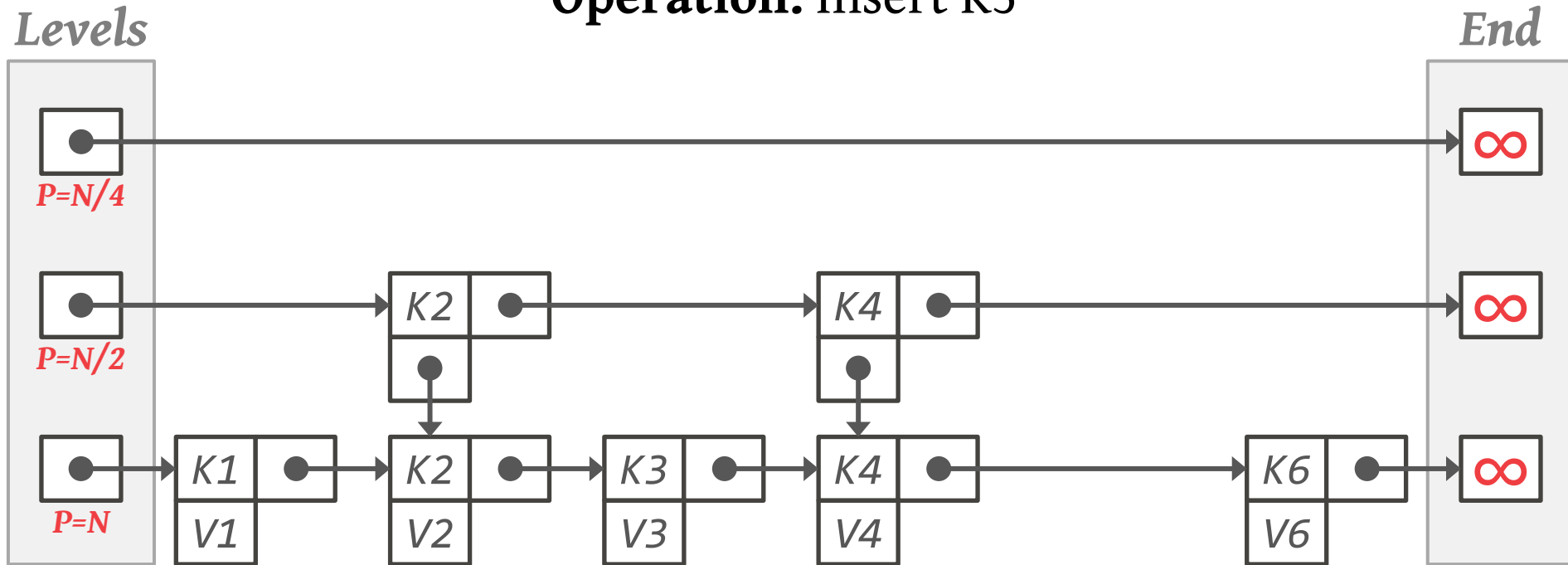
**Operation: Insert K5**





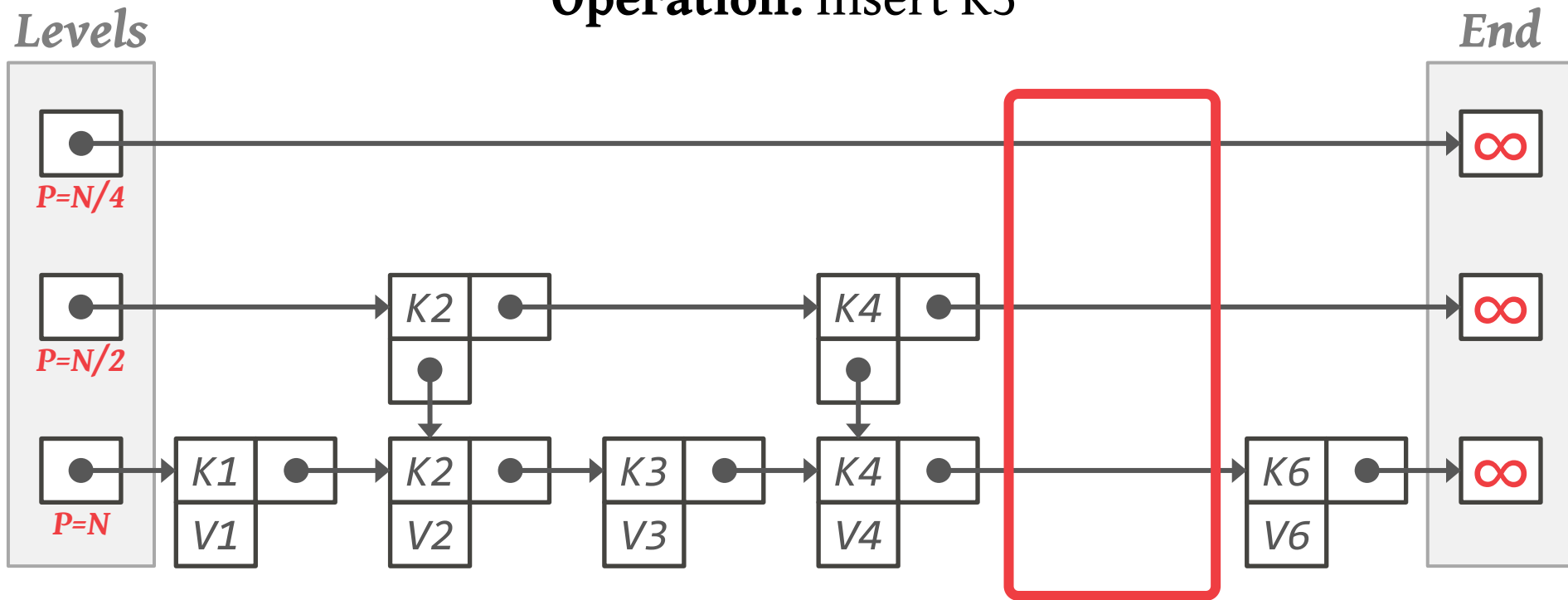
# SKIP LISTS: INSERT

**Operation: Insert K5**



# SKIP LISTS: INSERT

**Operation: Insert K5**

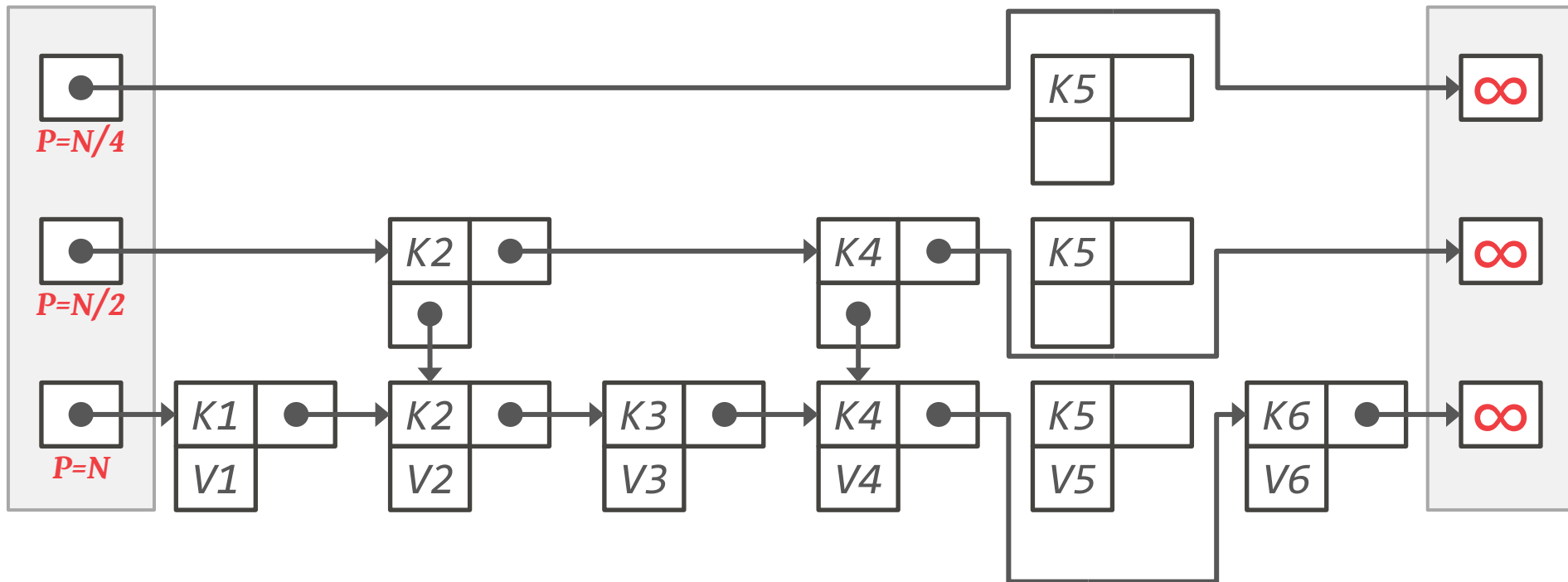


# SKIP LISTS: INSERT

**Operation: Insert K5**

*Levels*

*End*

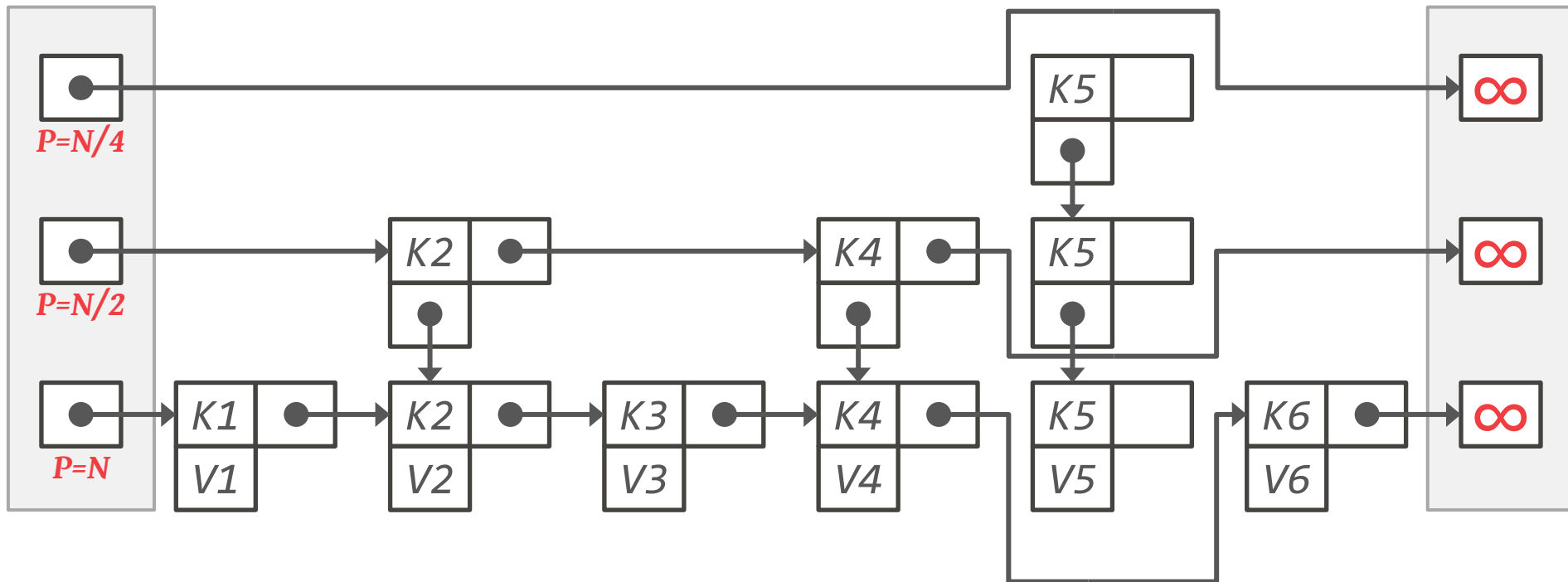


# SKIP LISTS: INSERT

**Operation: Insert K5**

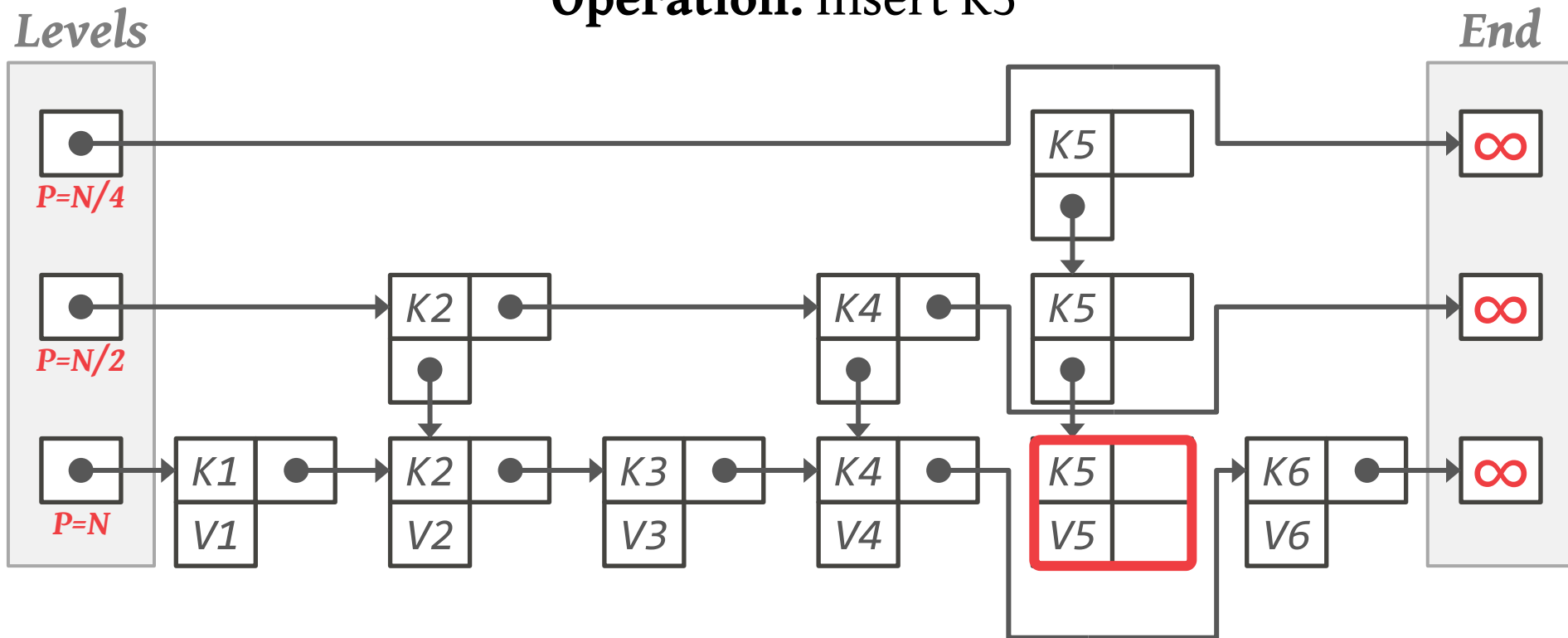
*Levels*

*End*



# SKIP LISTS: INSERT

**Operation: Insert K5**

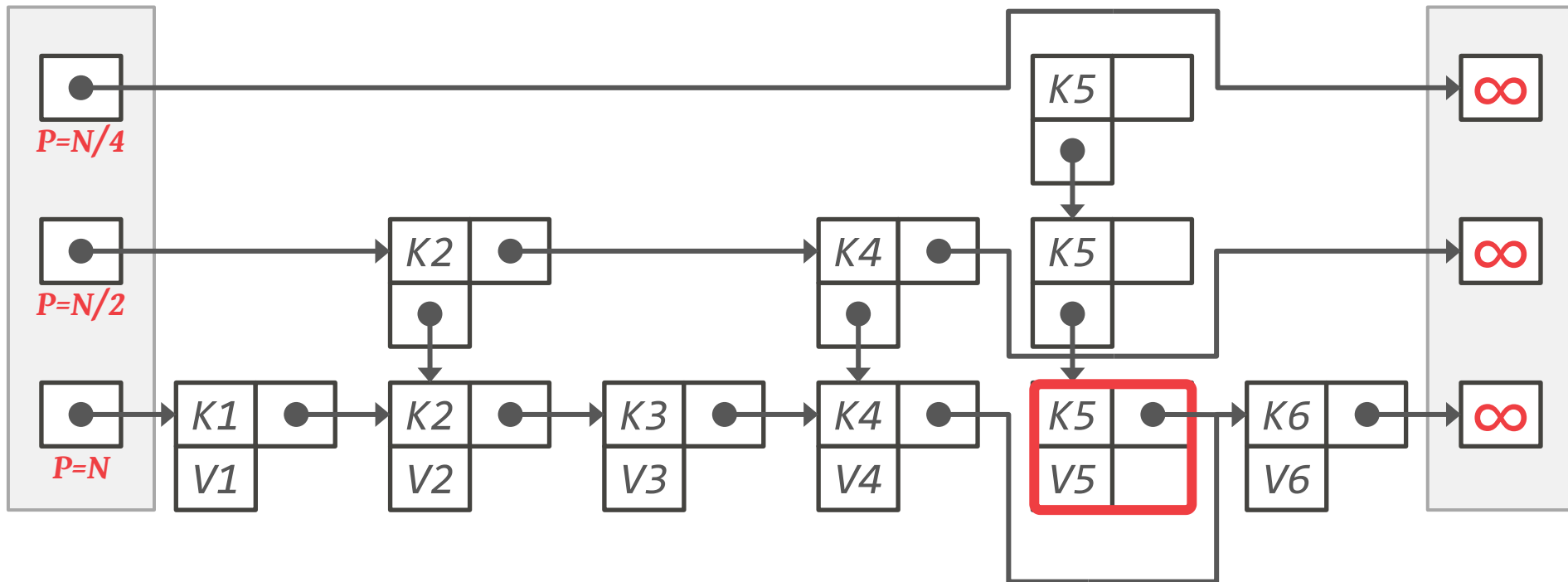


# SKIP LISTS: INSERT

**Operation: Insert K5**

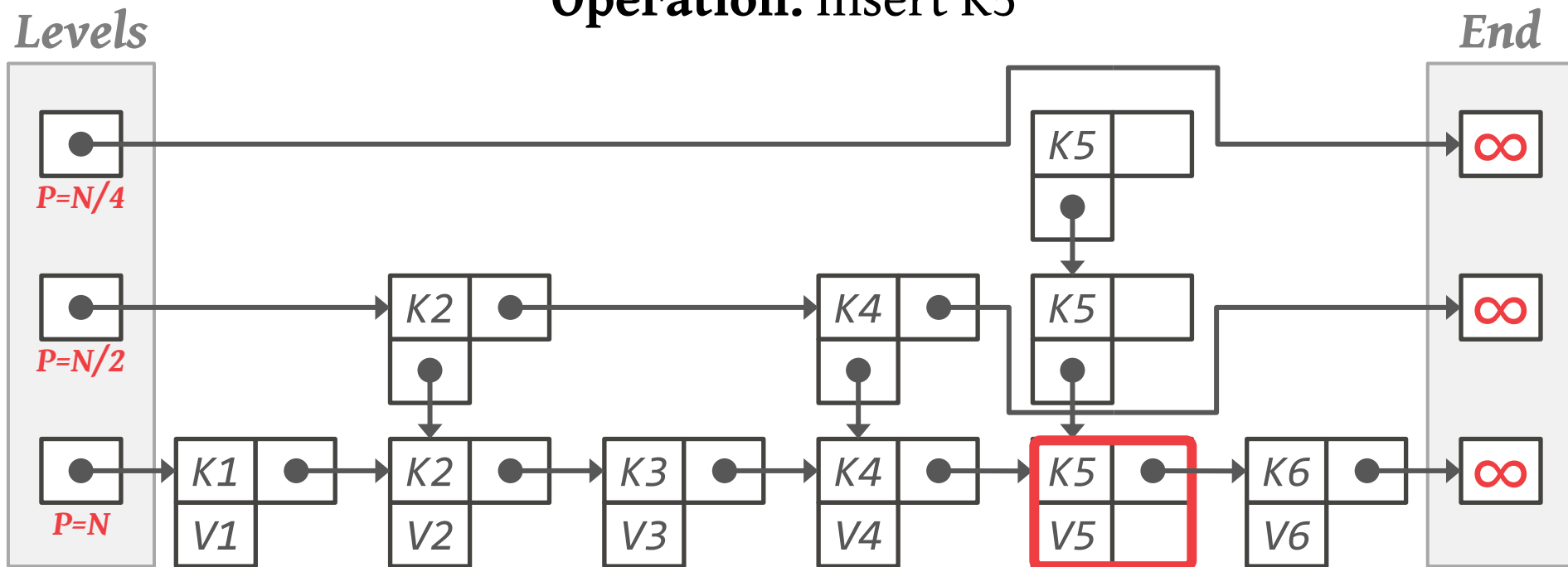
*Levels*

*End*



# SKIP LISTS: INSERT

**Operation: Insert K5**

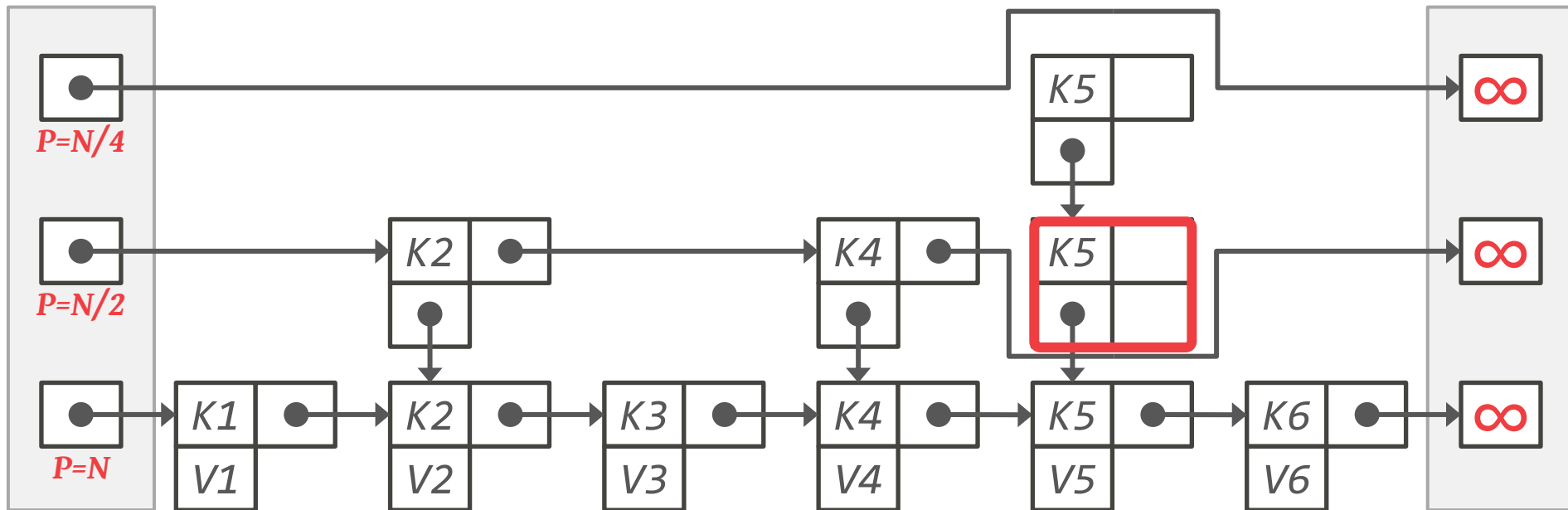


# SKIP LISTS: INSERT

**Operation: Insert K5**

*Levels*

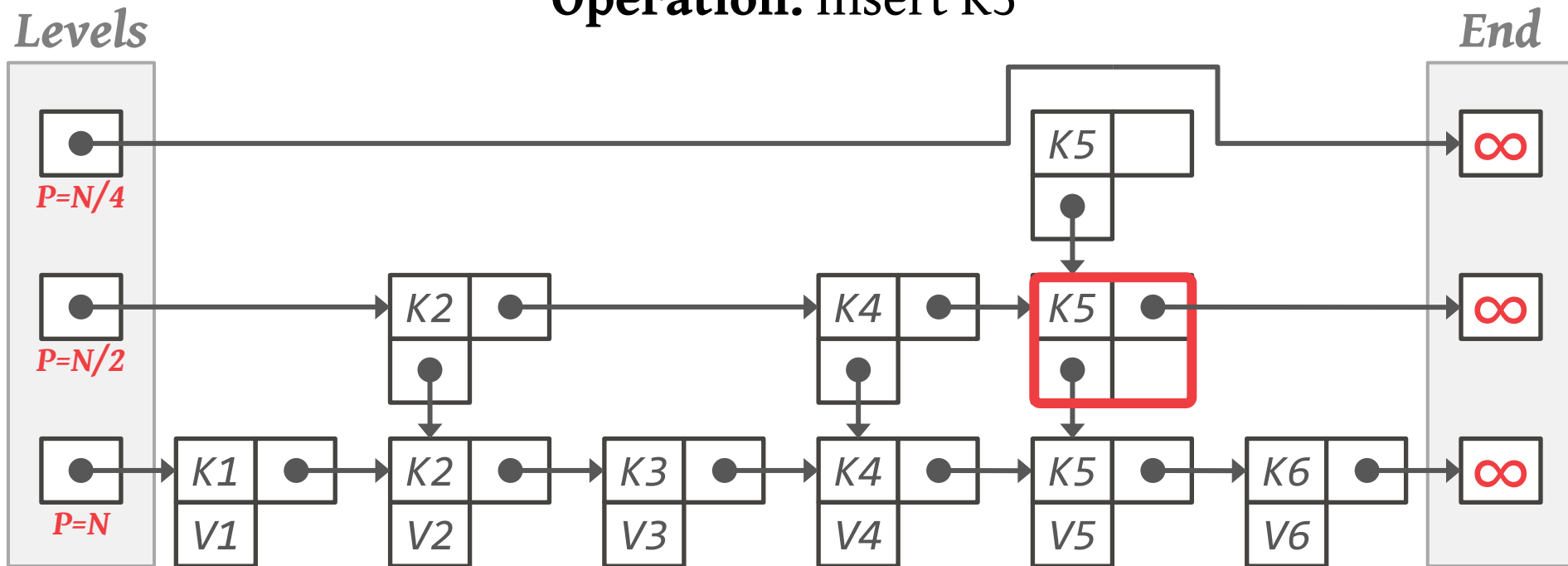
*End*





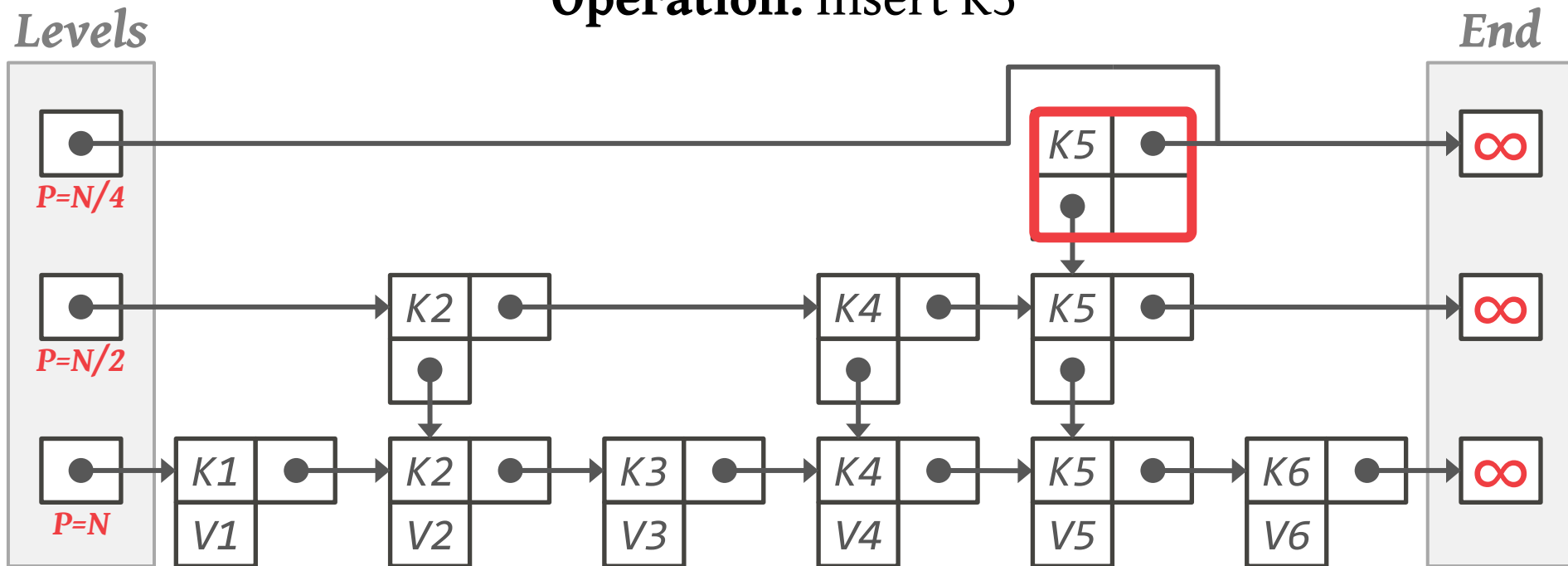
# SKIP LISTS: INSERT

**Operation: Insert K5**



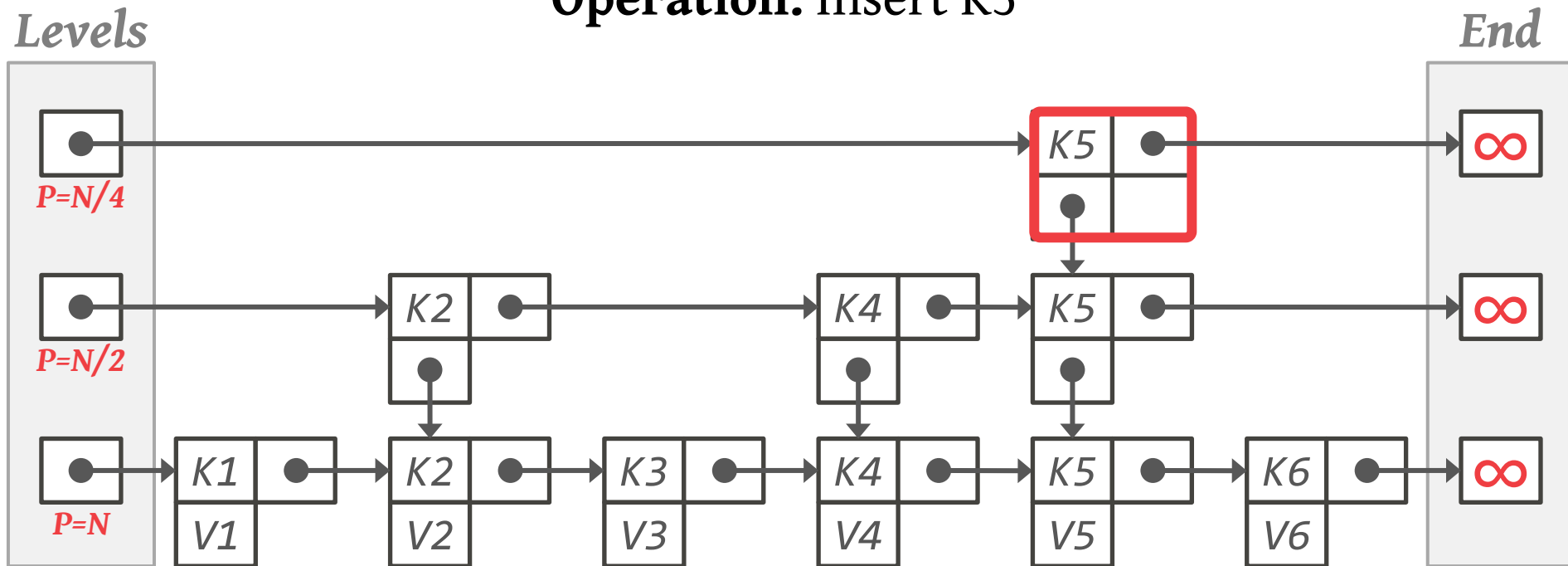
# SKIP LISTS: INSERT

**Operation: Insert K5**



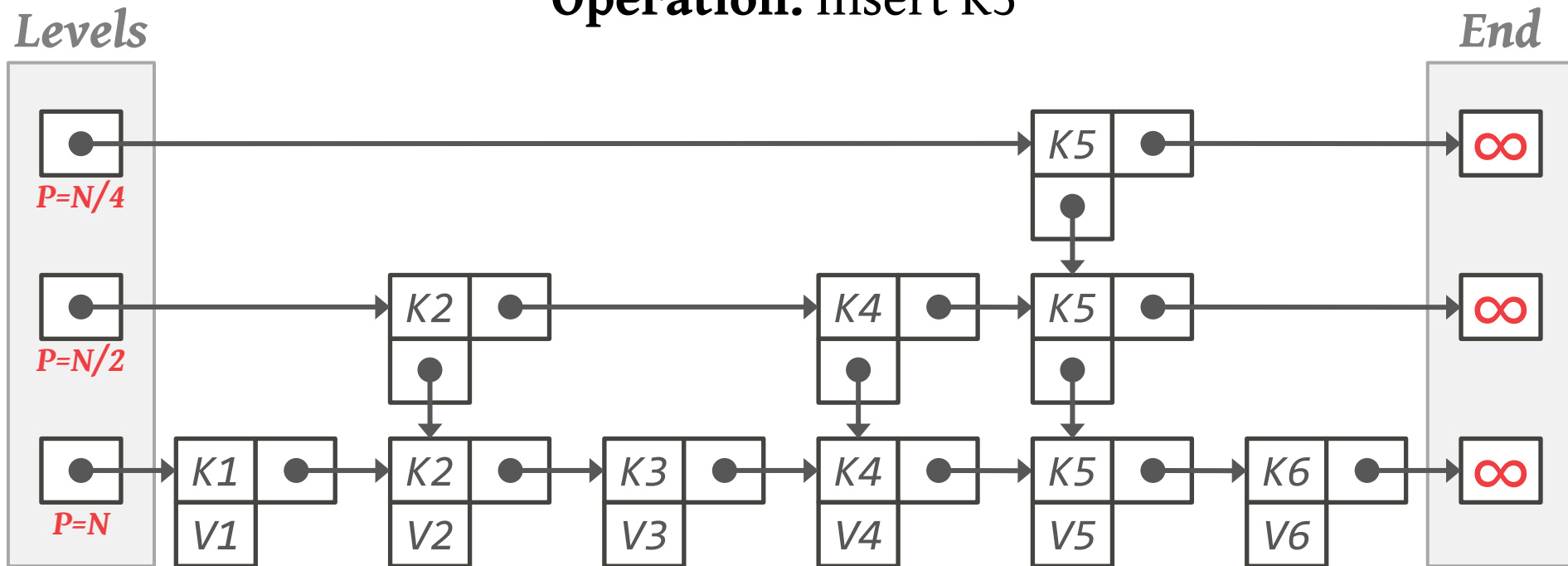
# SKIP LISTS: INSERT

**Operation: Insert K5**



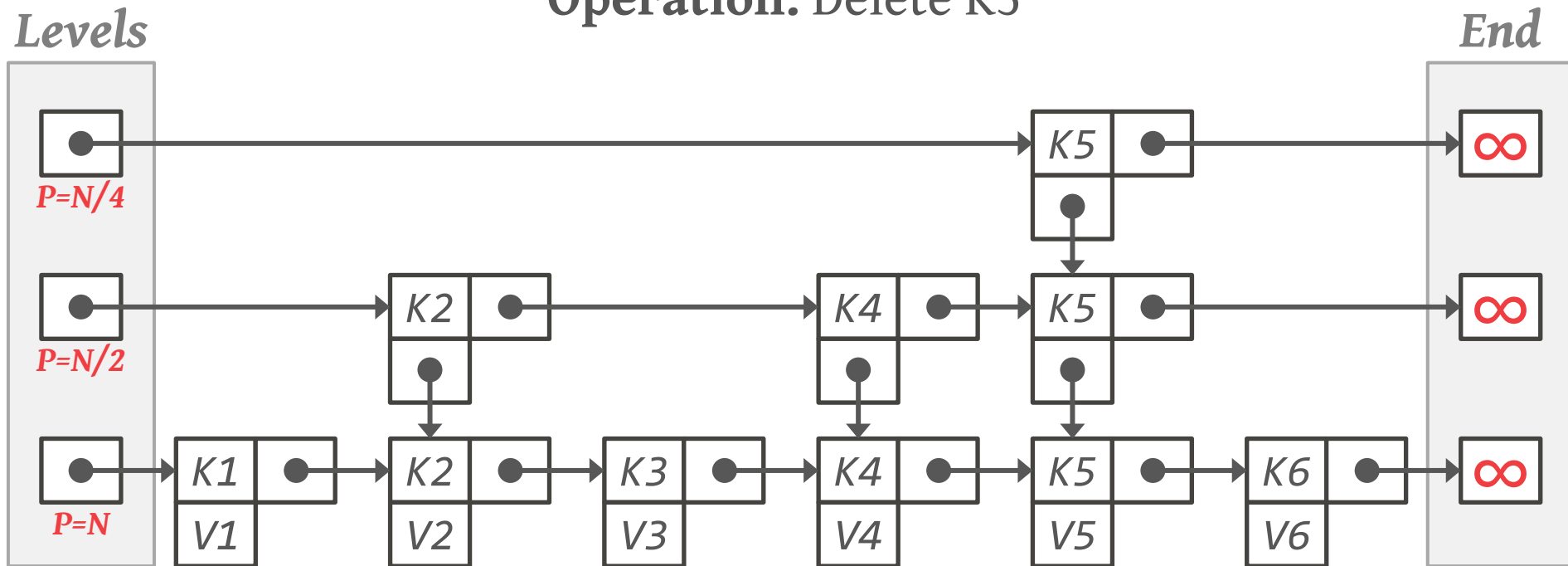
# SKIP LISTS: INSERT

**Operation: Insert K5**



# SKIP LISTS: DELETE

Operation: Delete K5

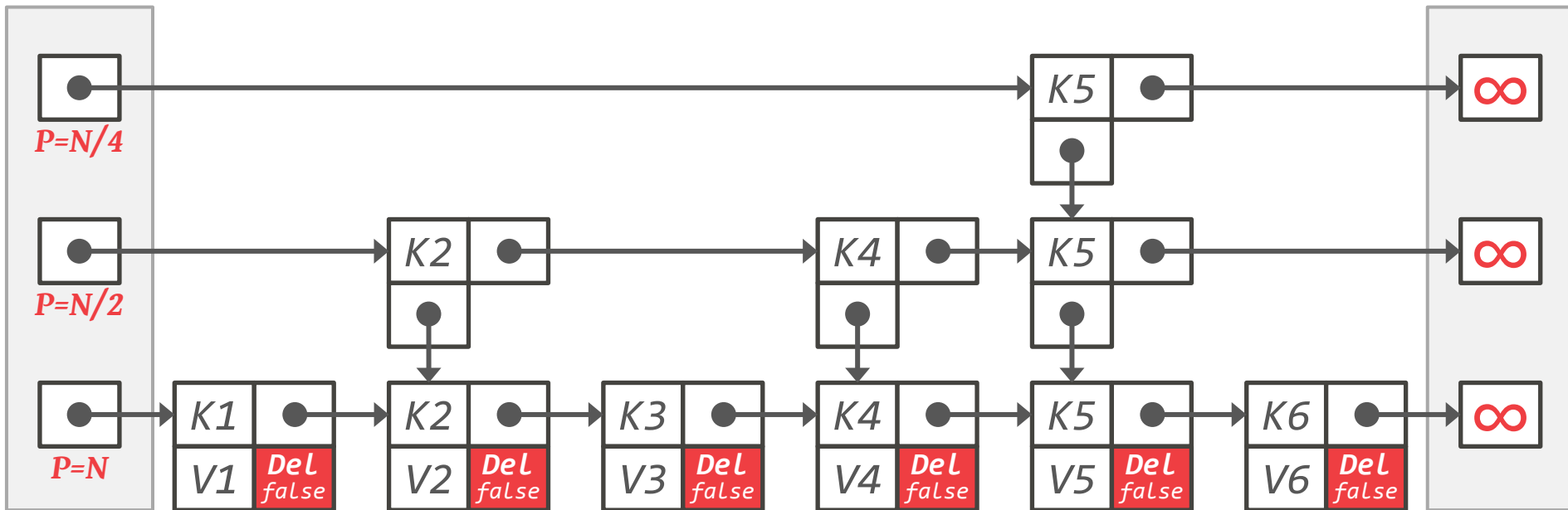


# SKIP LISTS: DELETE

Operation: Delete K5

Levels

End

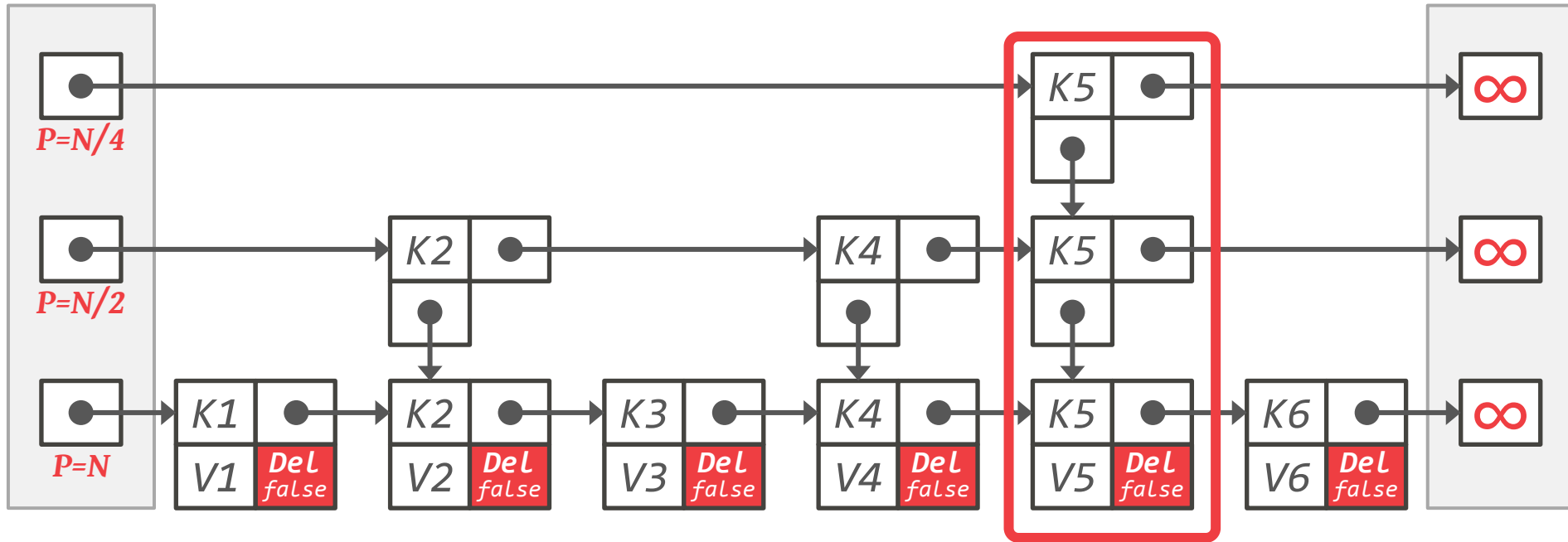


# SKIP LISTS: DELETE

Operation: Delete K5

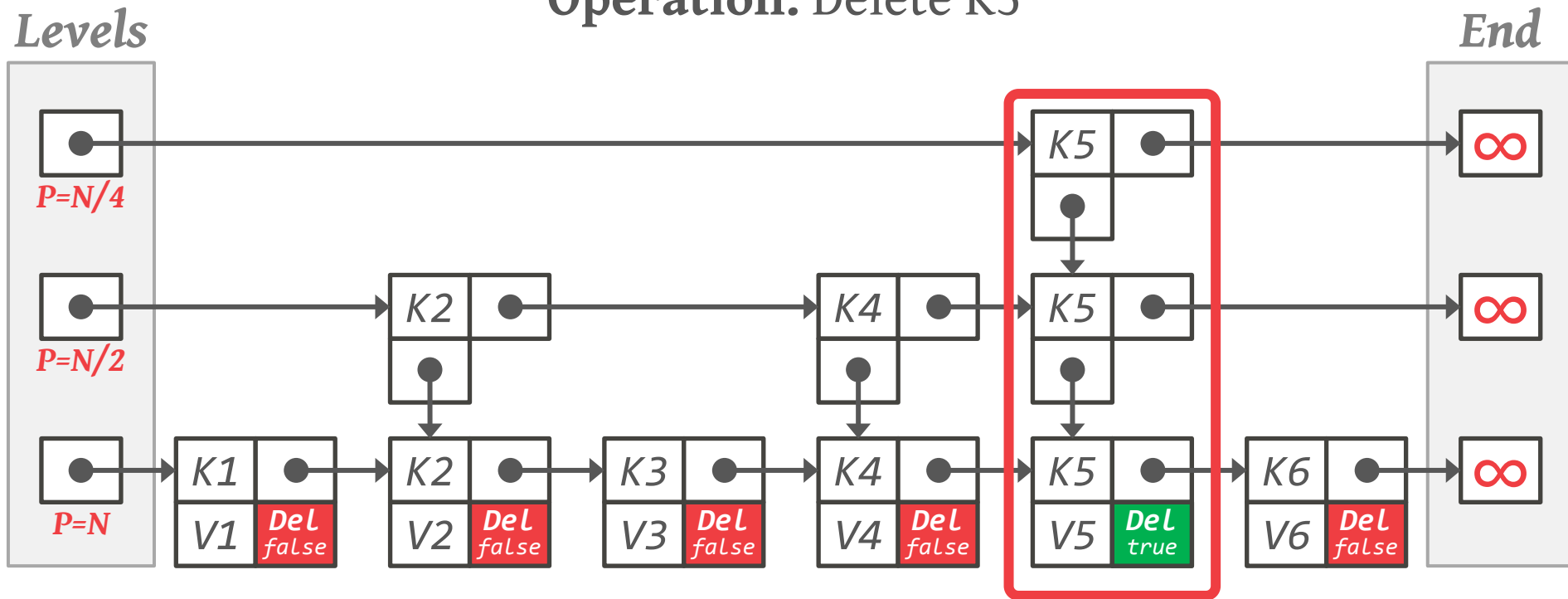
Levels

End



# SKIP LISTS: DELETE

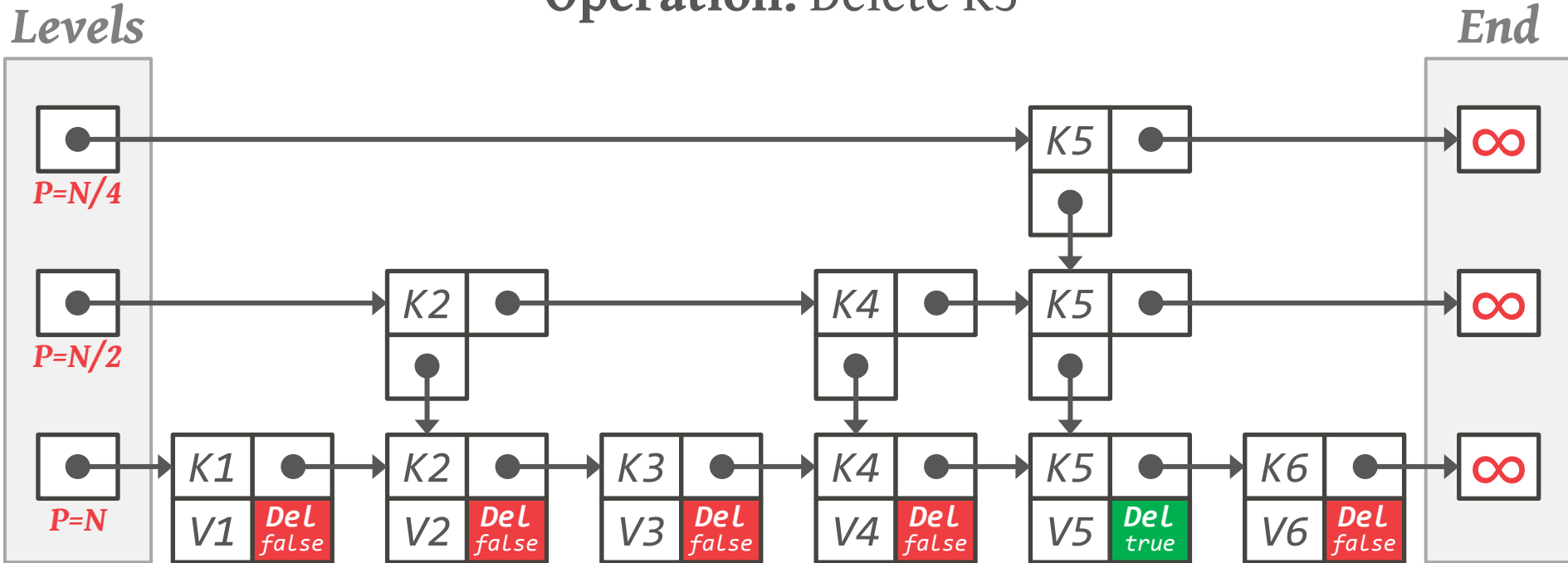
Operation: Delete K5





# SKIP LISTS: DELETE

## Operation: Delete K5

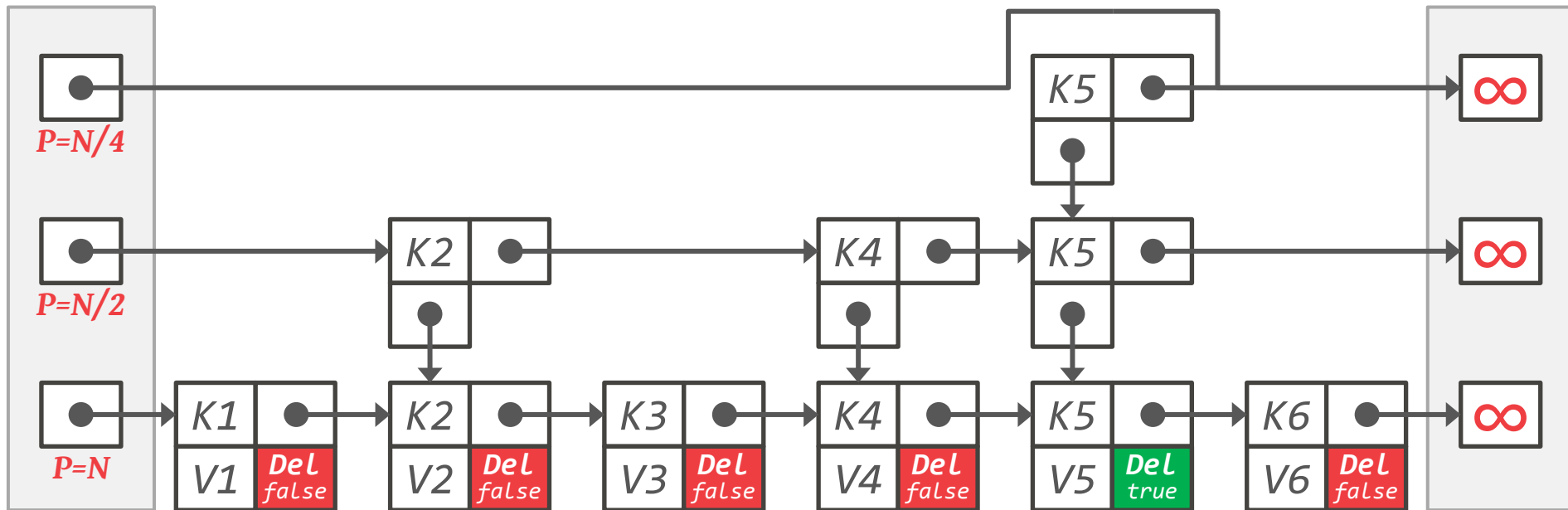


# SKIP LISTS: DELETE

Operation: Delete K5

Levels

End

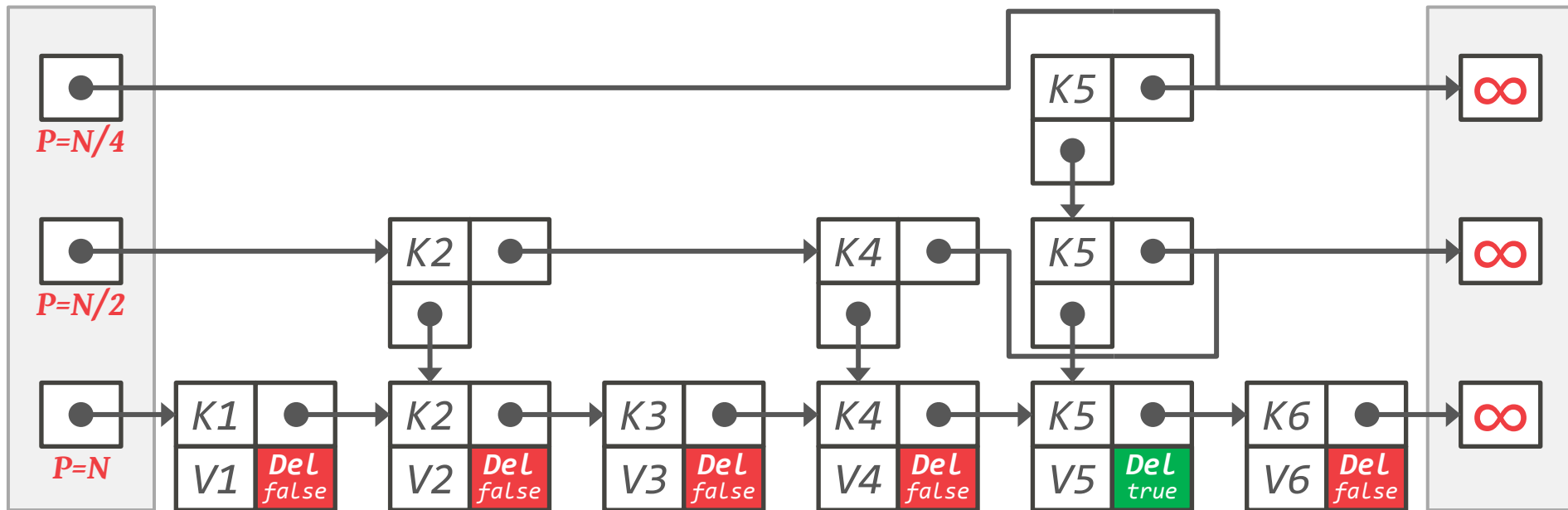


# SKIP LISTS: DELETE

Operation: Delete K5

Levels

End

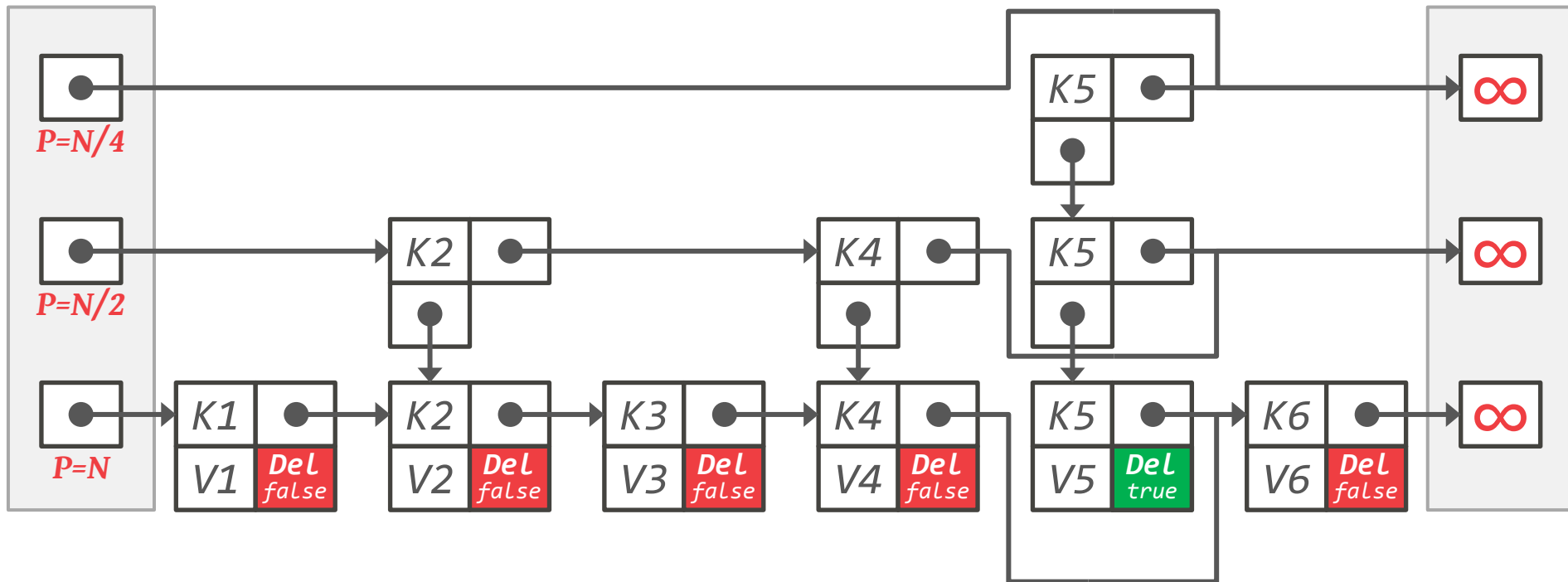


# SKIP LISTS: DELETE

Operation: Delete K5

Levels

End

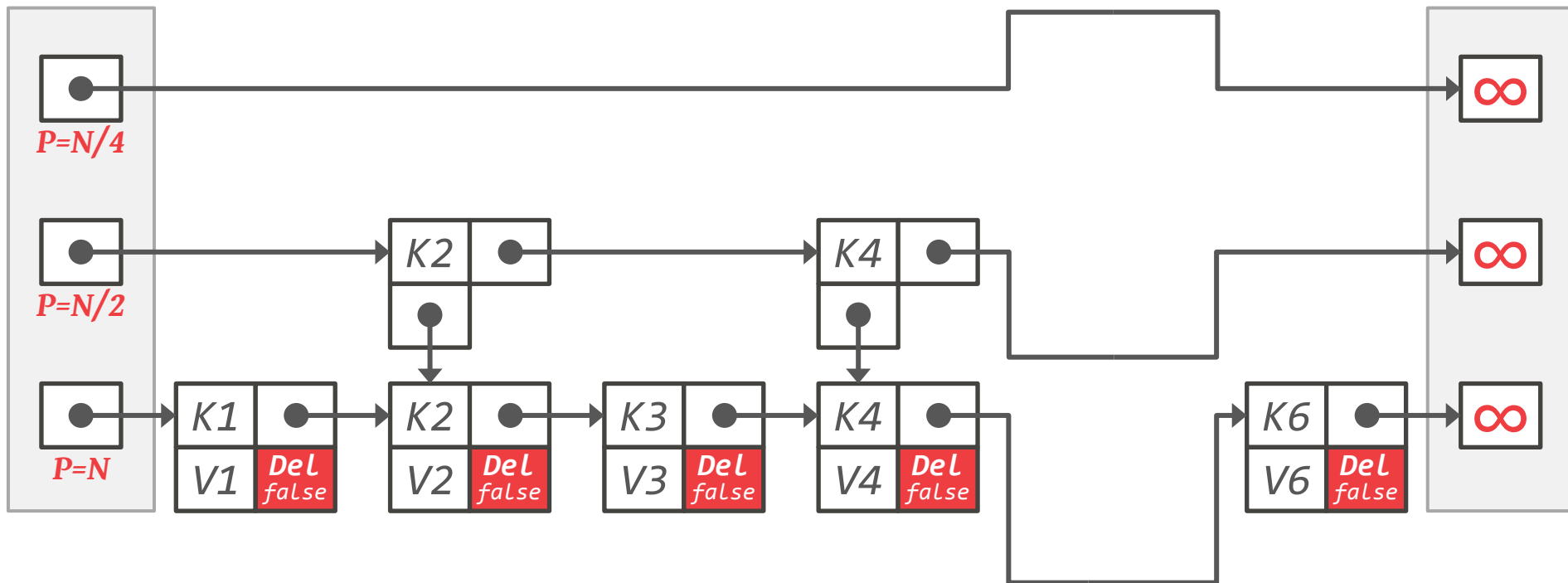


# SKIP LISTS: DELETE

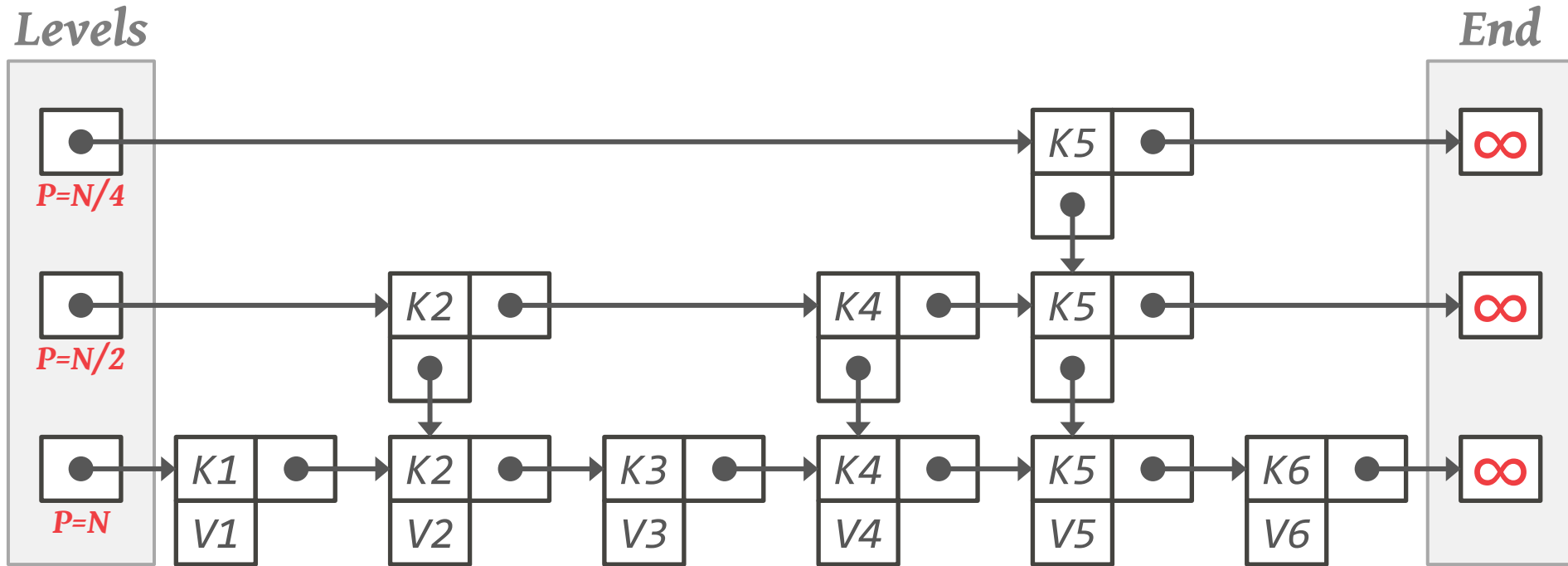
Operation: Delete K5

Levels

End



# SKIP LISTS: REVERSE SEARCH

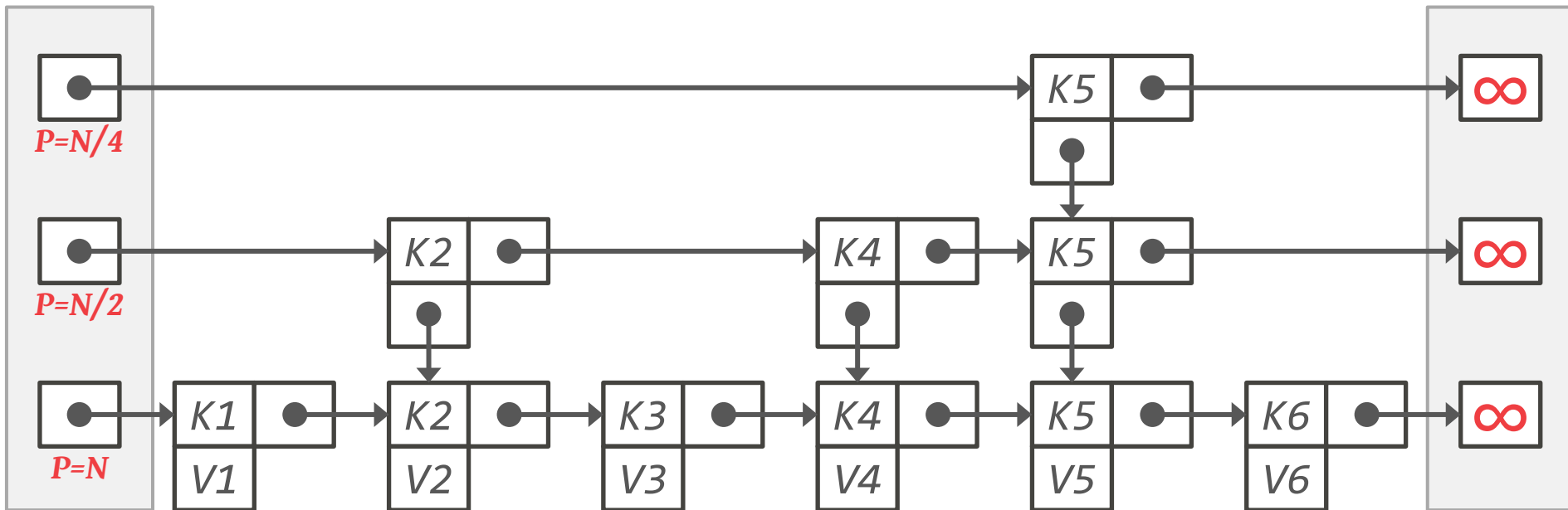


# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

*Levels*

*End*



Source: [Mark Papadakis](#)

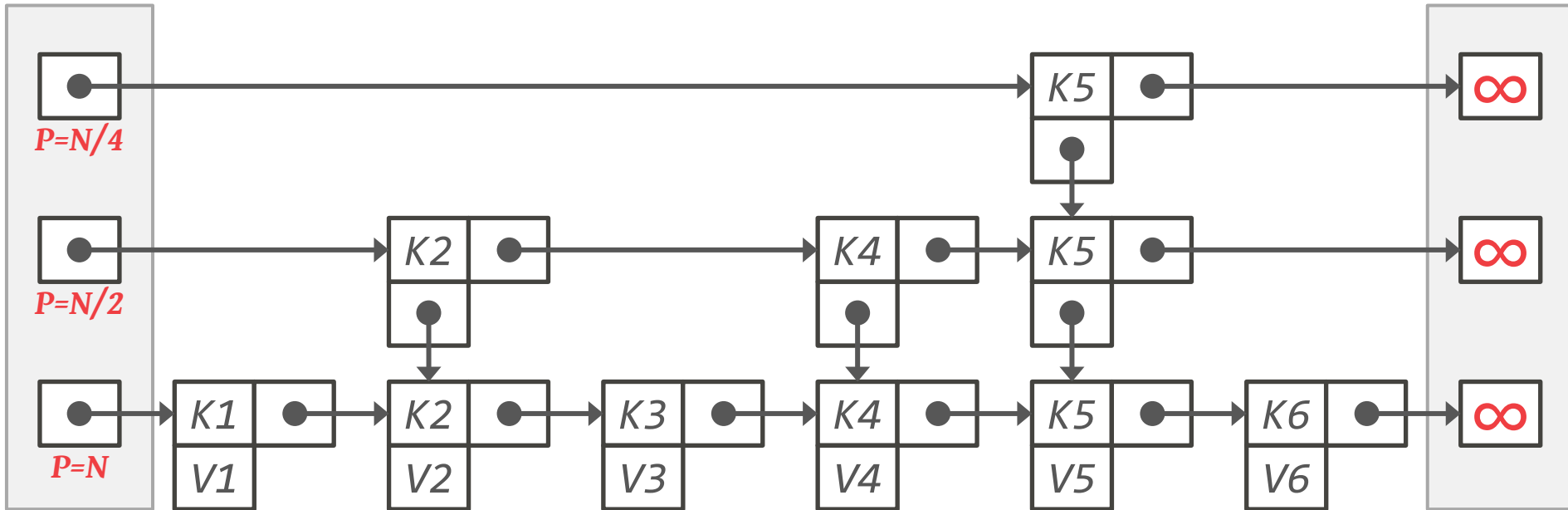
CMU 15-721 (Spring 2016)

# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

*Levels*

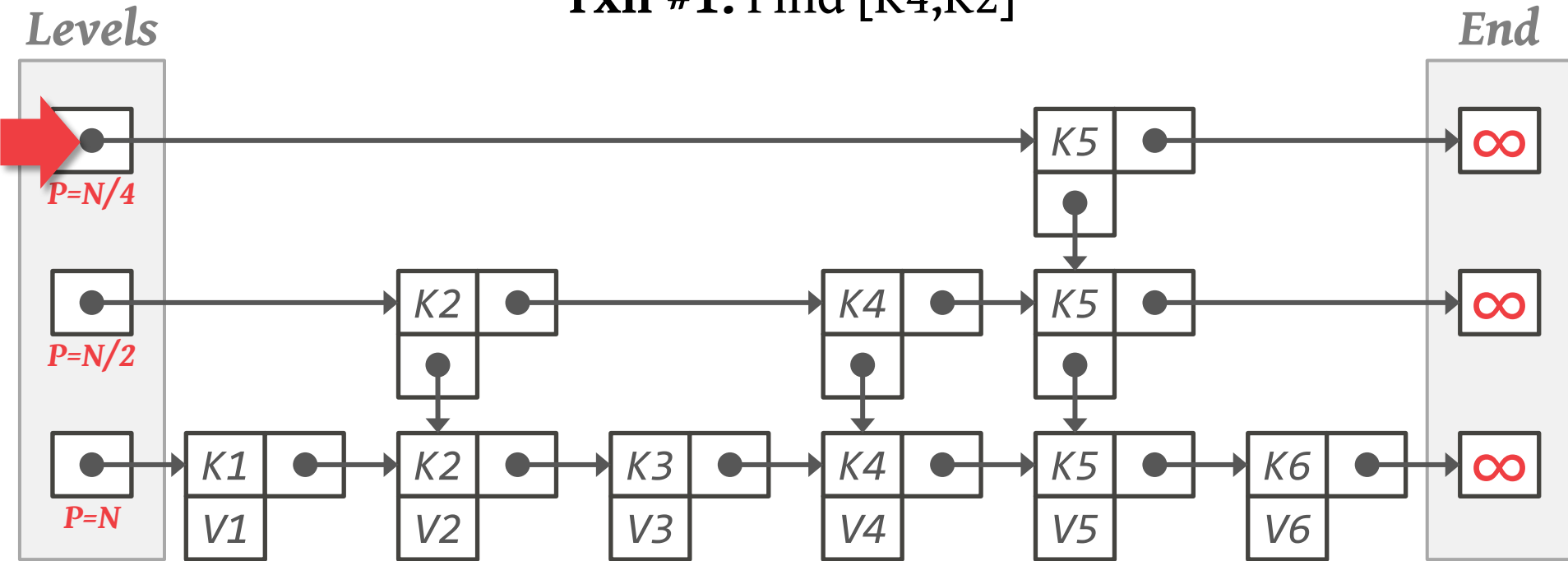
*End*





# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

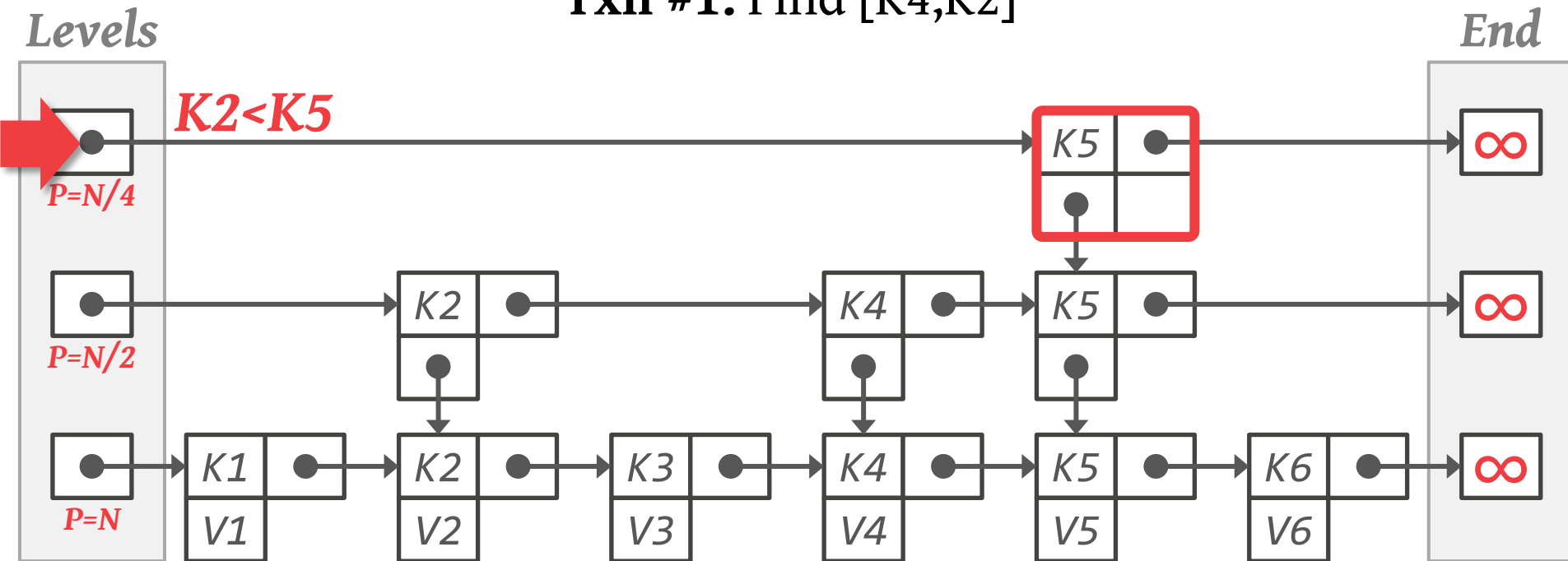


Source: [Mark Papadakis](#)

CMU 15-721 (Spring 2016)

# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

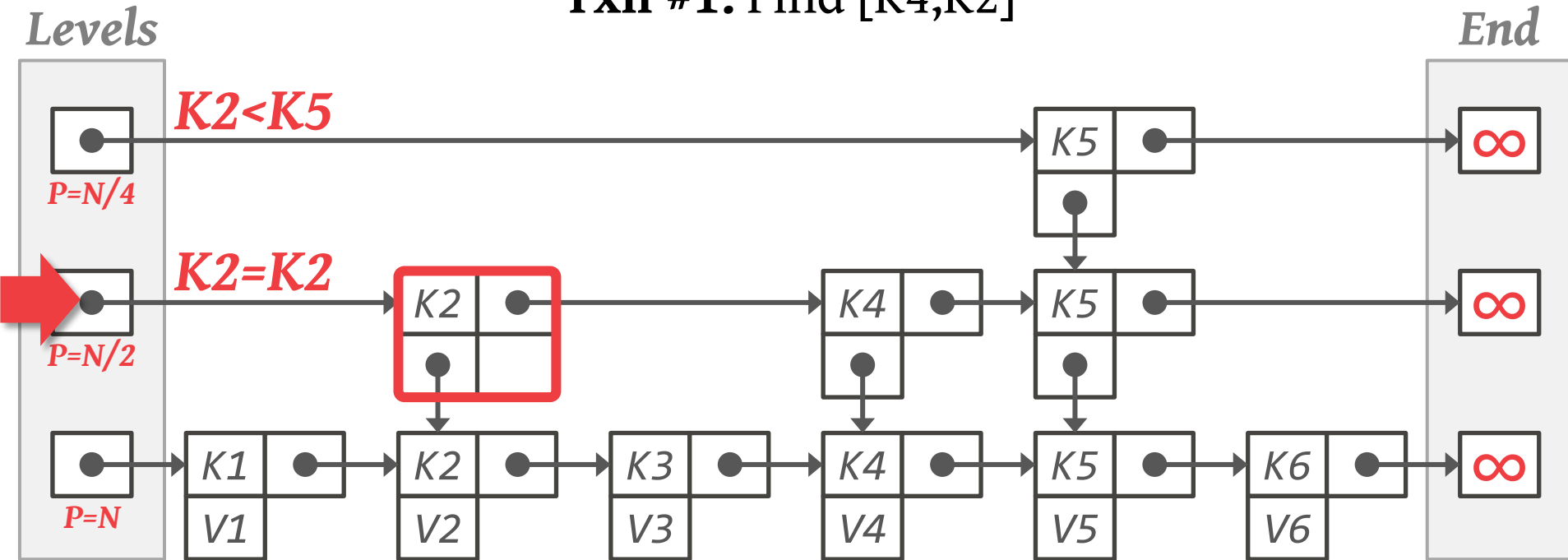


Source: [Mark Papadakis](#)

CMU 15-721 (Spring 2016)

# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

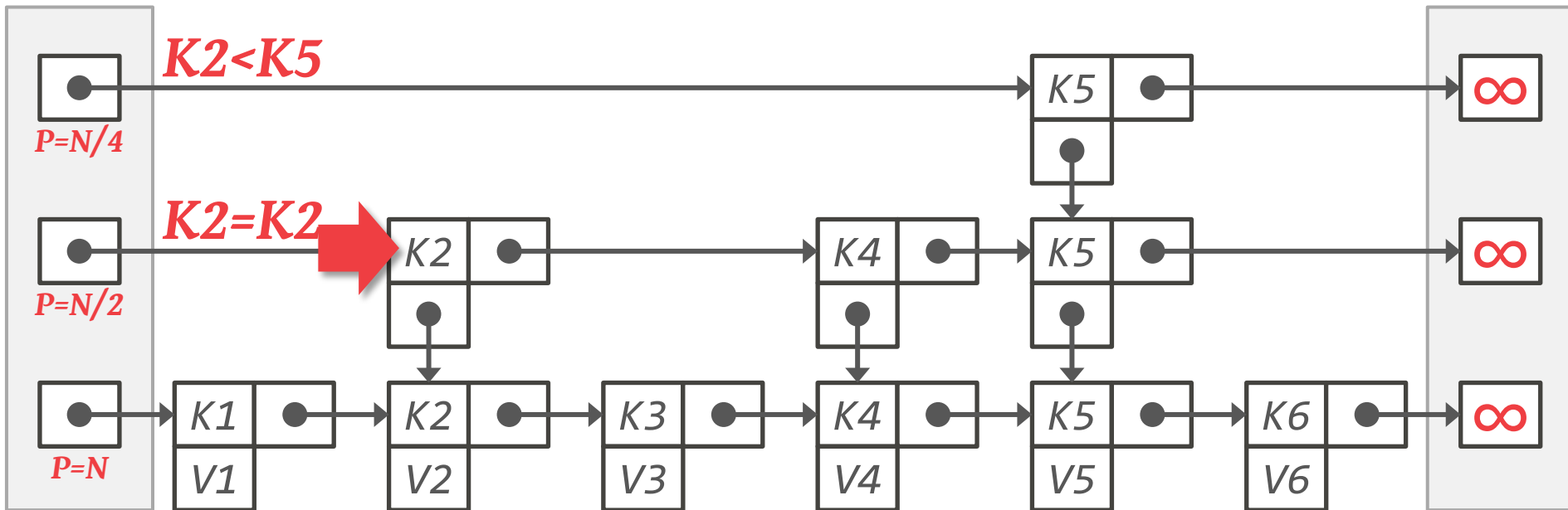


# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

*Levels*

*End*

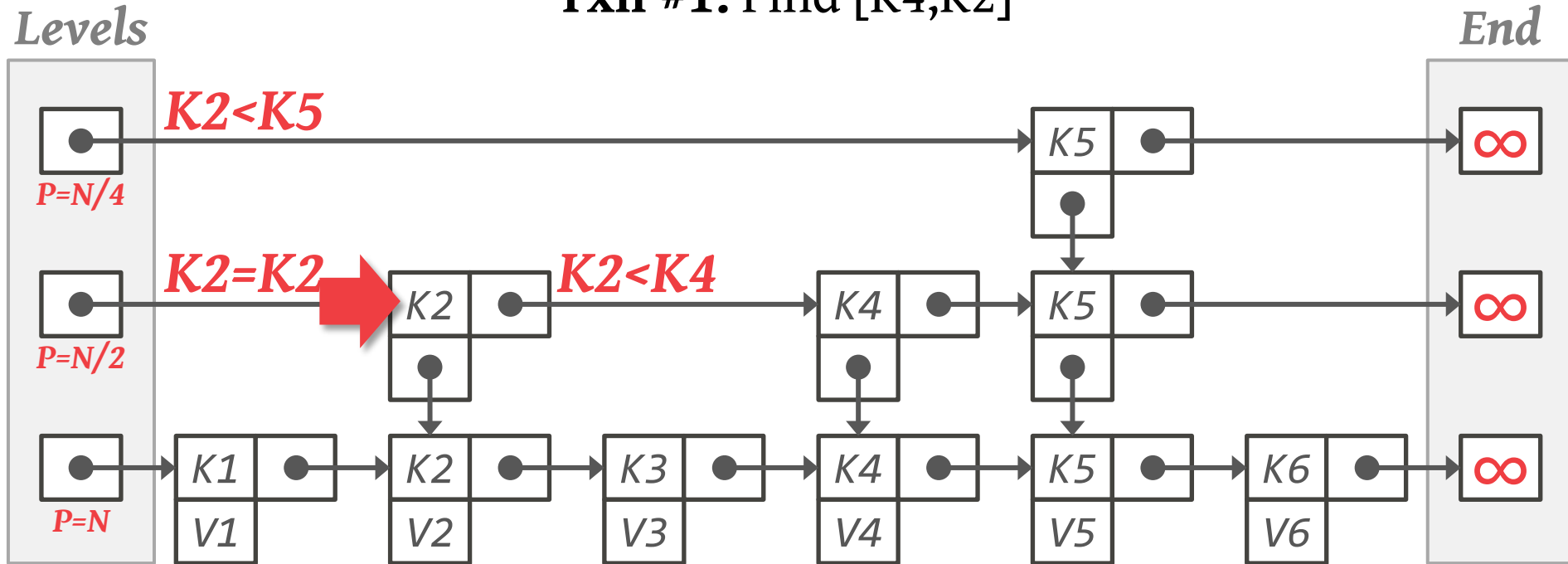


Source: [Mark Papadakis](#)

CMU 15-721 (Spring 2016)

# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

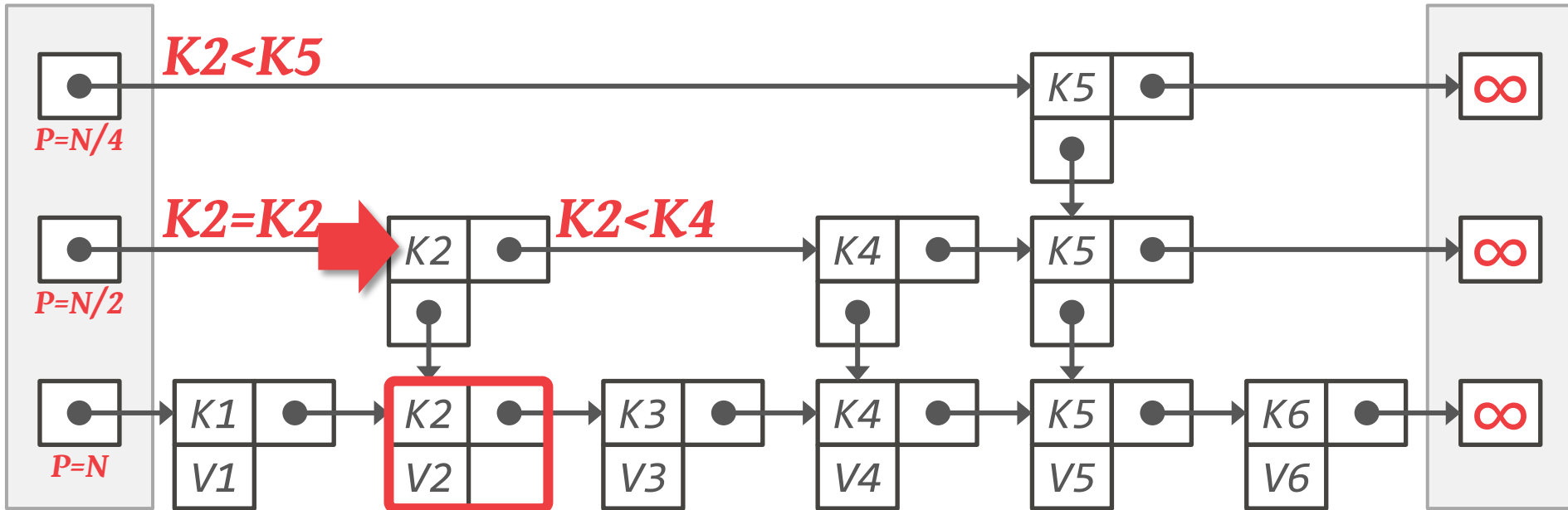


# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

*Levels*

*End*

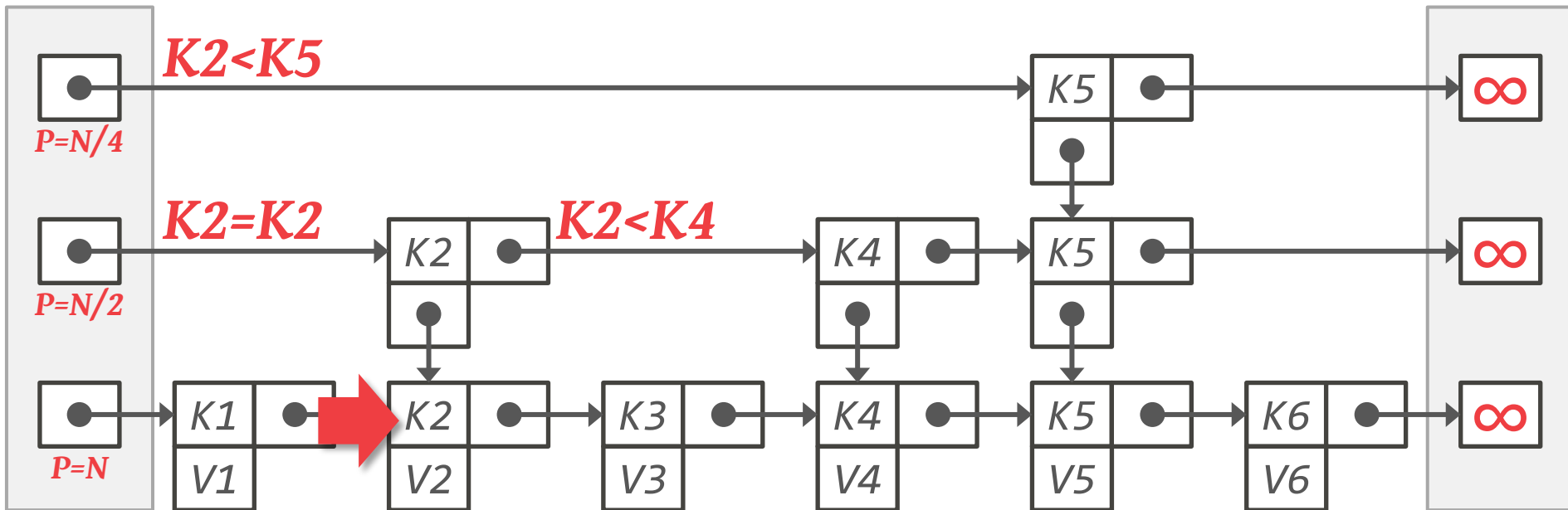


# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

*Levels*

*End*



Source: [Mark Papadakis](#)

CMU 15-721 (Spring 2016)

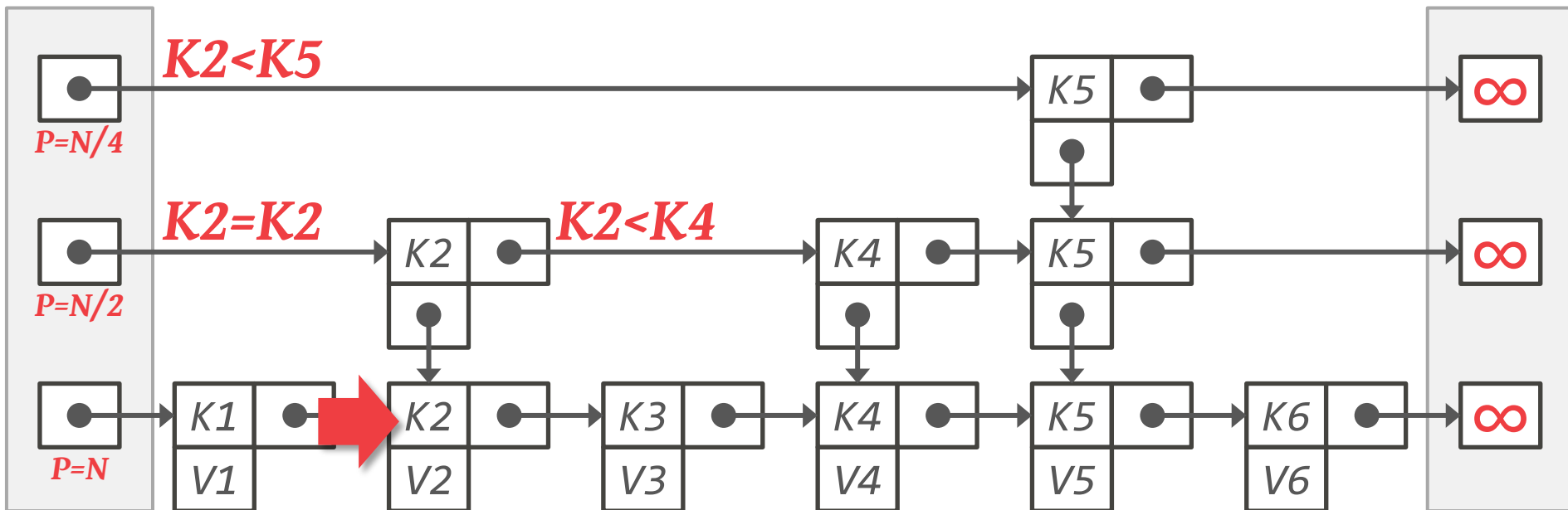
# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

**Stack:**

*End*

*Levels*



Source: [Mark Papadakis](#)

CMU 15-721 (Spring 2016)



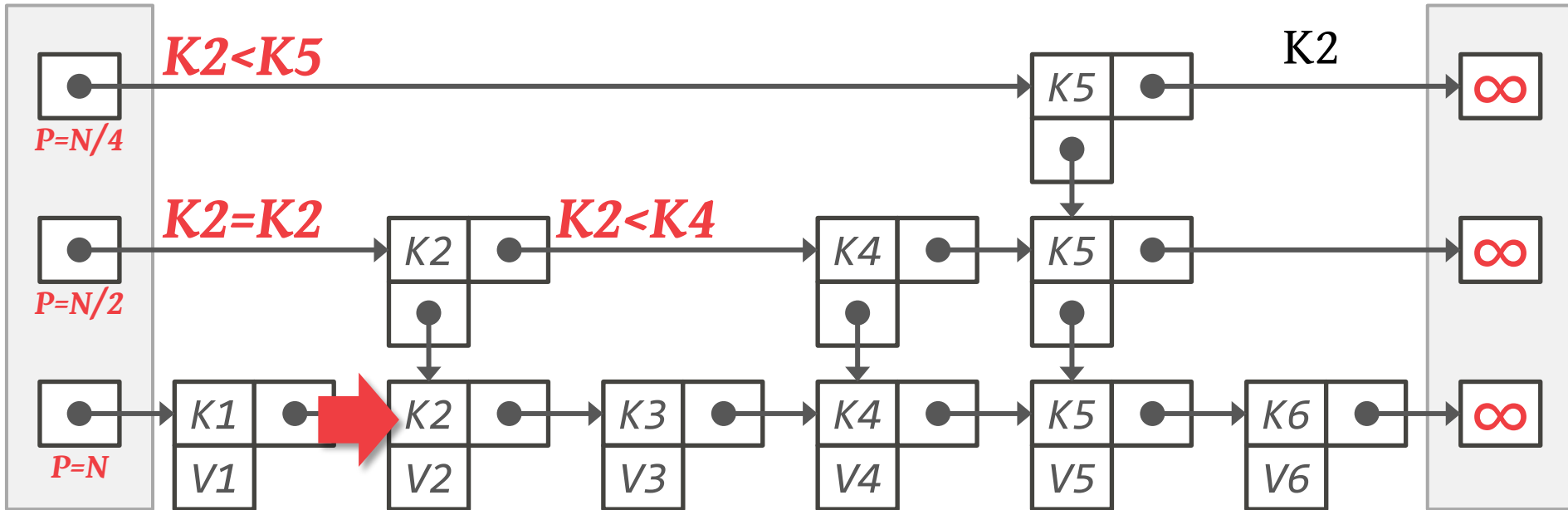
# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

**Stack:**

*End*

*Levels*



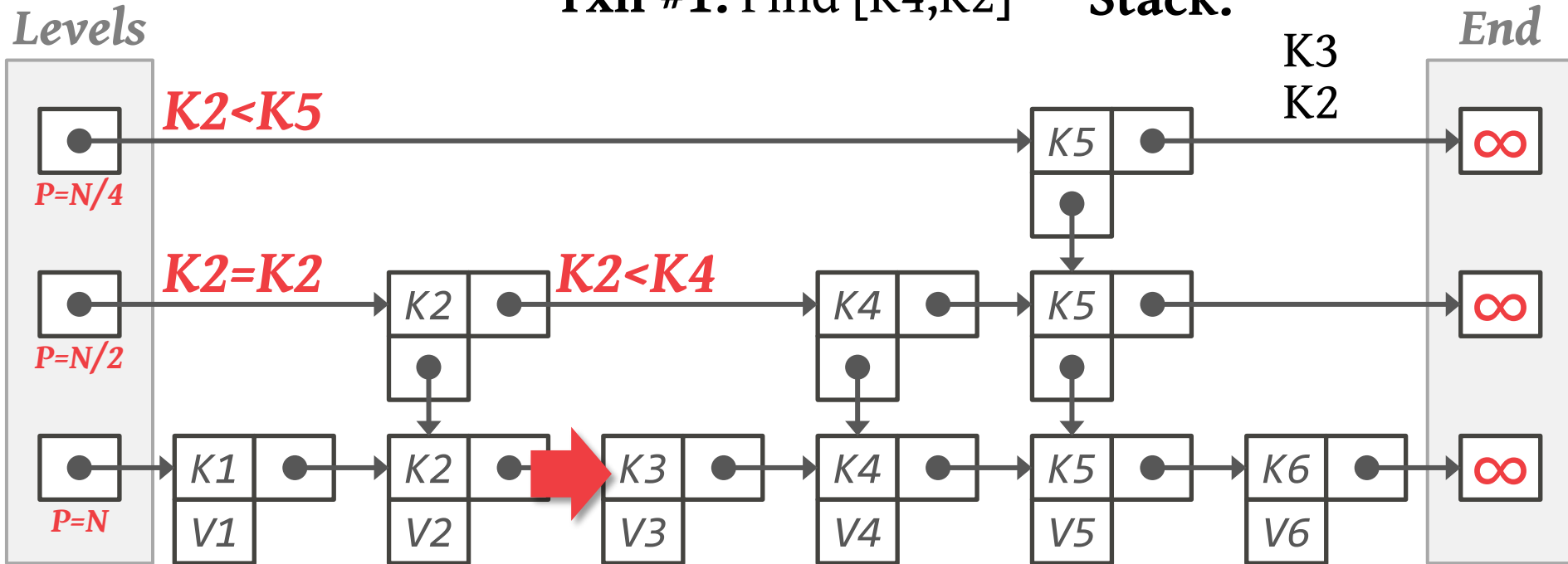
# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

**Stack:**

*End*

K3  
K2



Source: [Mark Papadakis](#)

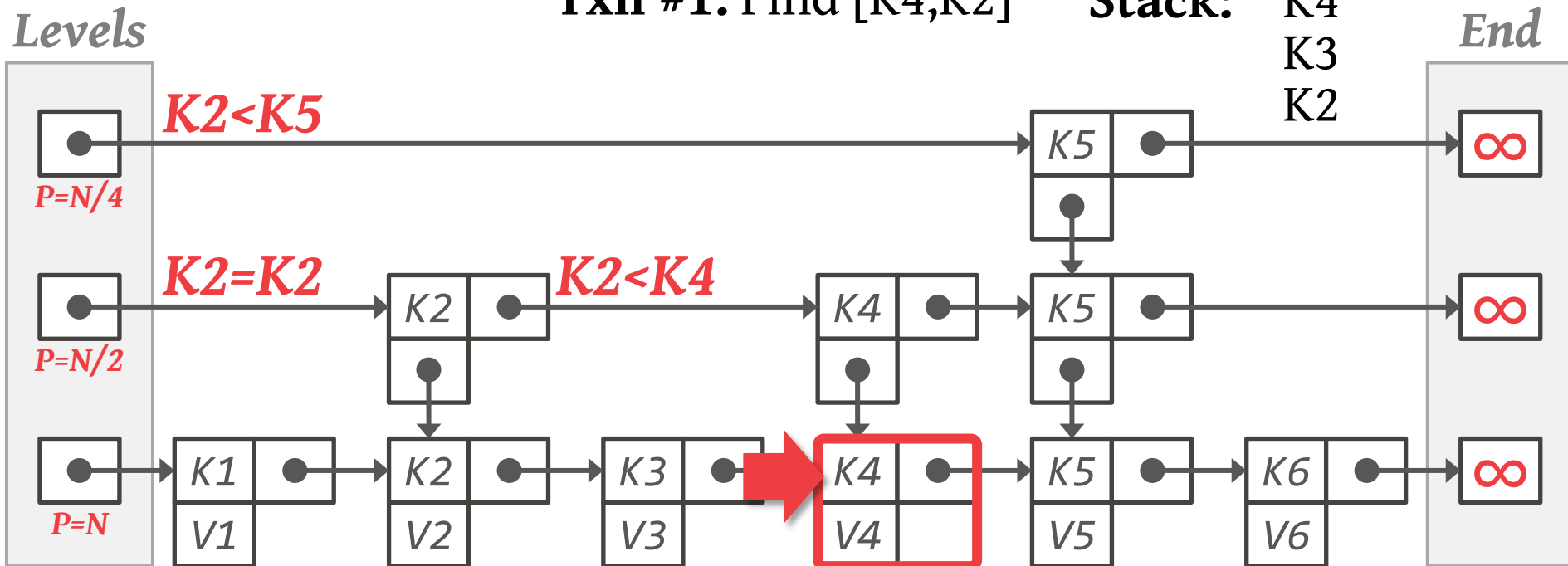
CMU 15-721 (Spring 2016)

# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

**Stack:** K4  
K3  
K2

*End*

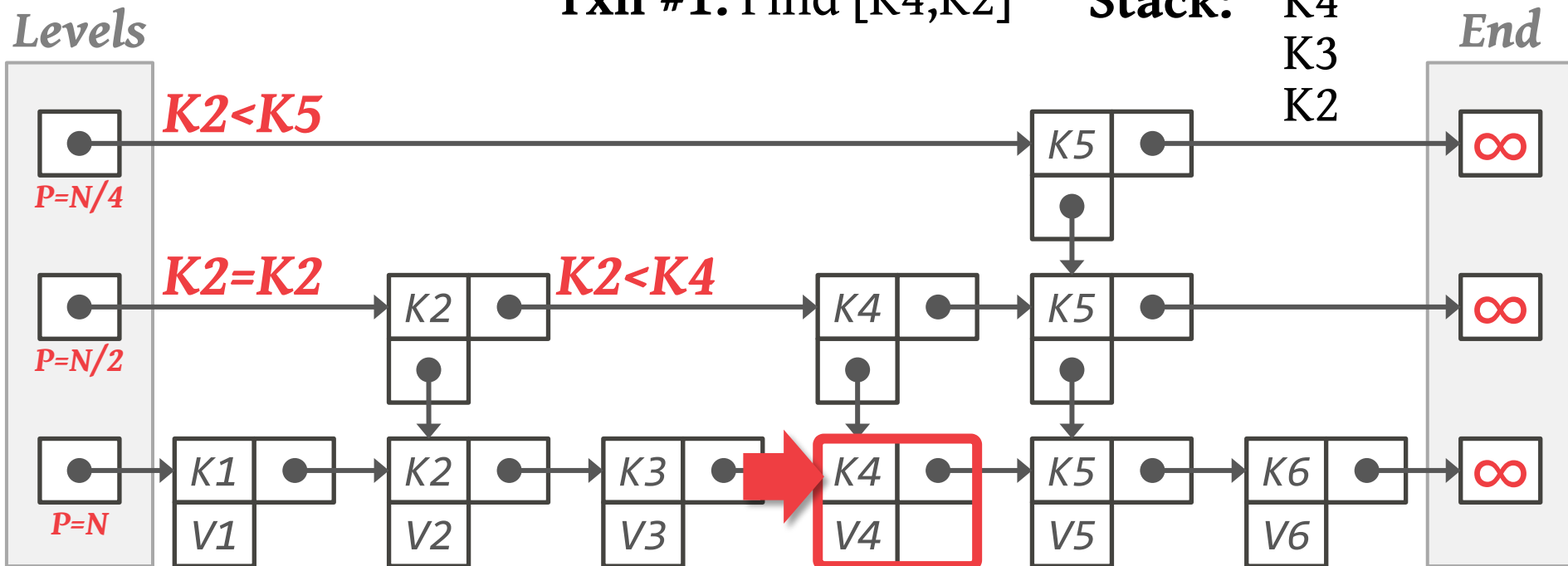


# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find  $[K4, K2]$**

**Stack:** K4  
K3  
K2

*End*



Source: [Mark Papadakis](#)

CMU 15-721 (Spring 2016)

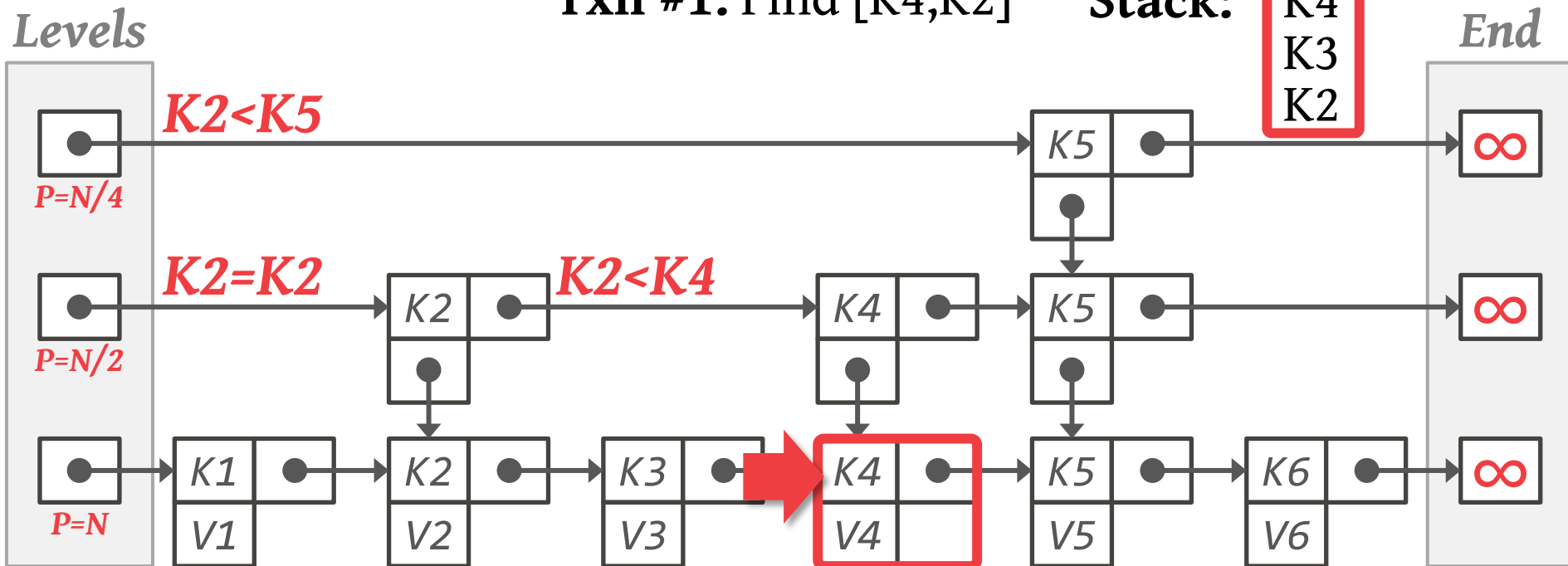
# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

**Stack:**

K4  
K3  
K2

*End*



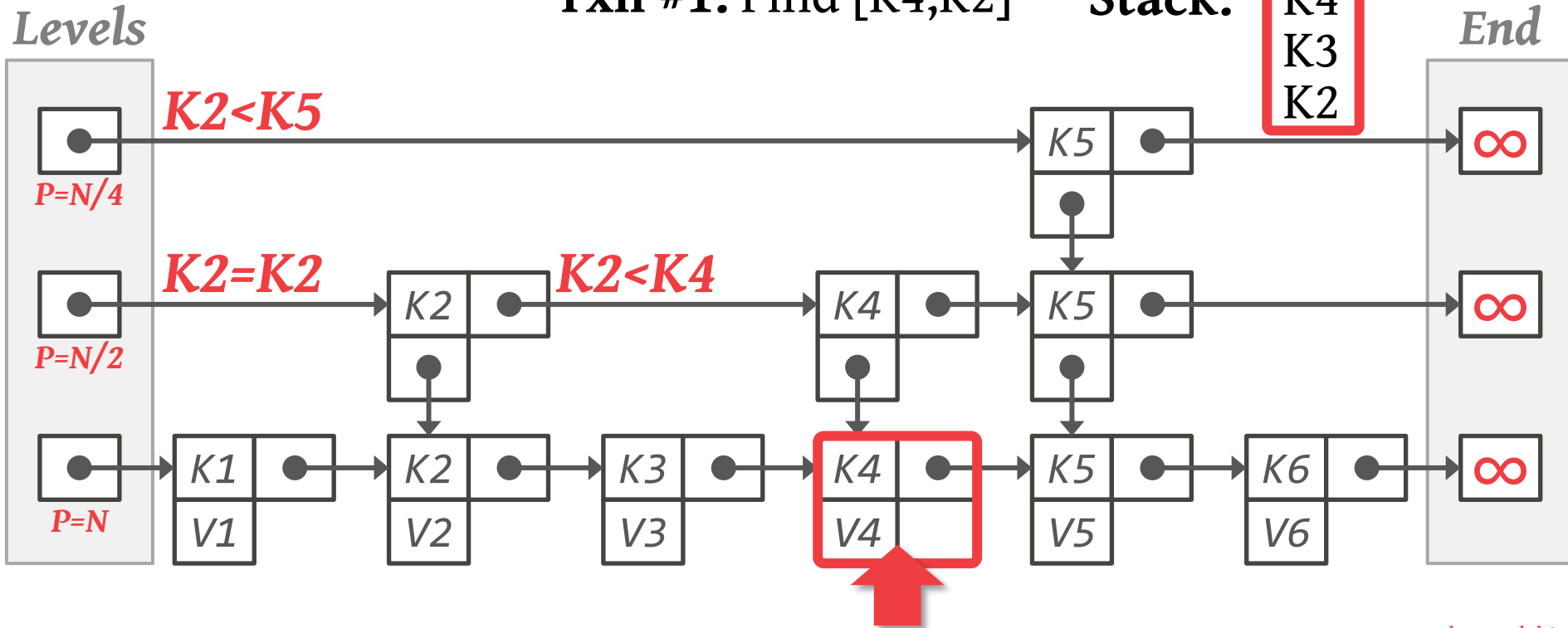
# SKIP LISTS: REVERSE SEARCH

**Txn #1: Find [K4,K2]**

**Stack:**

K4  
K3  
K2

*End*



Source: [Mark Papadakis](#)

CMU 15-721 (Spring 2016)

# SKIP LISTS: REVERSE SEARCH

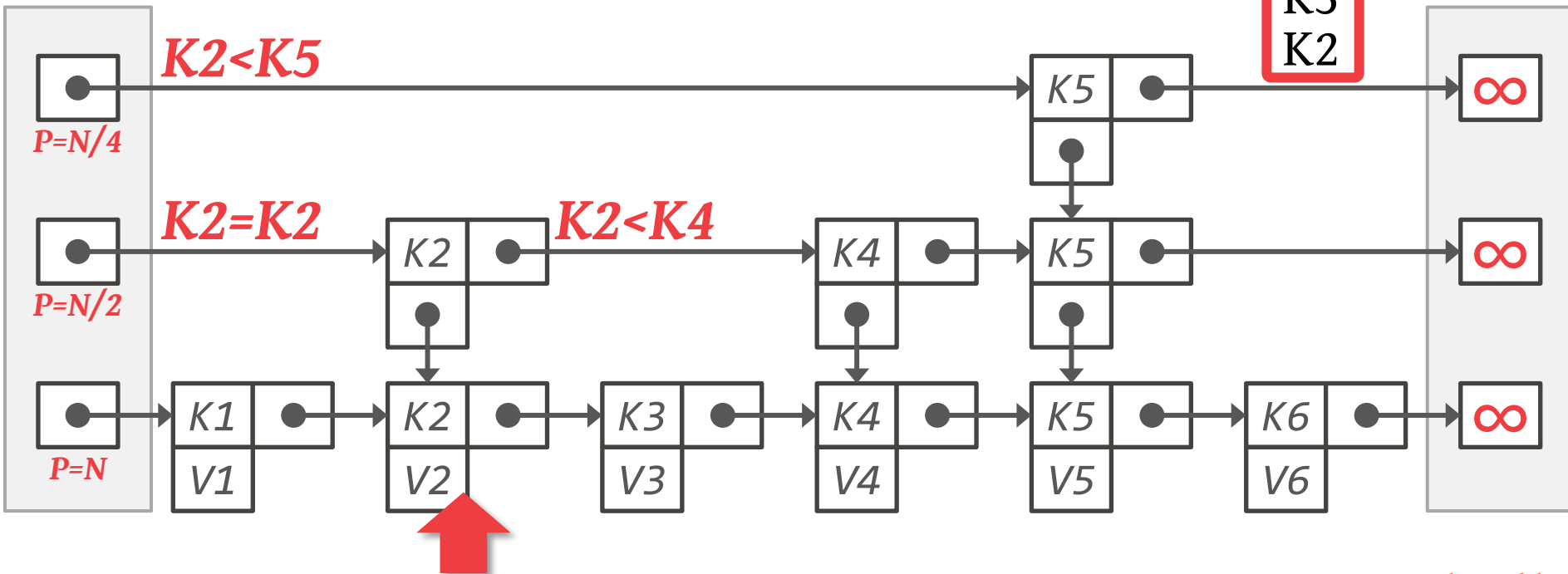
**Txn #1: Find [K4,K2]**

**Stack:**

K4  
K3  
K2

*End*

*Levels*



Source: [Mark Papadakis](#)

CMU 15-721 (Spring 2016)

# ADAPATIVE RADIX TREE (ART)

---

Uses a digital representation of keys to examine key prefixes one-by-one instead of comparing the entire key.

Radix trees properties:

- The height of the tree depends on the length of keys.
- Does not require rebalancing
- The path to a leaf node represents the key of the leaf
- Keys are stored implicitly and can be reconstructed from paths.

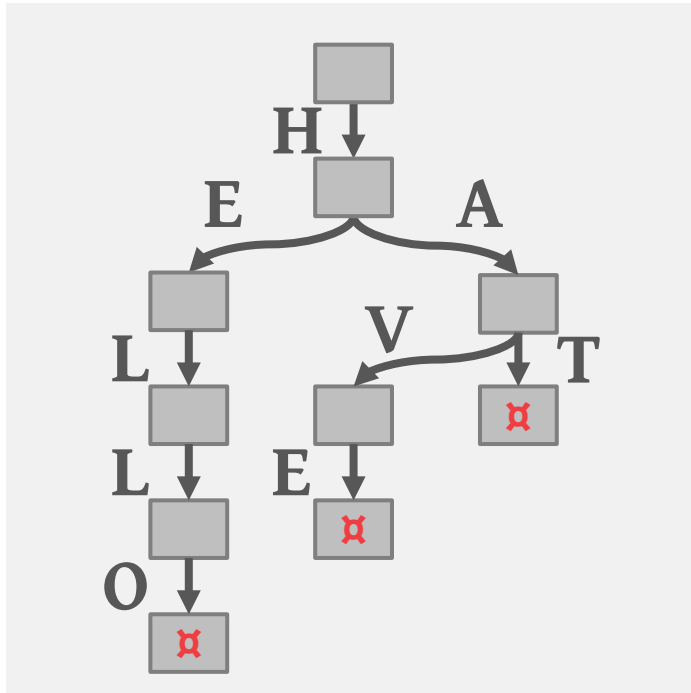


THE ADAPTIVE RADIX TREE: ARTFUL  
INDEXING FOR MAIN-MEMORY DATABASES  
*ICDE 2013*

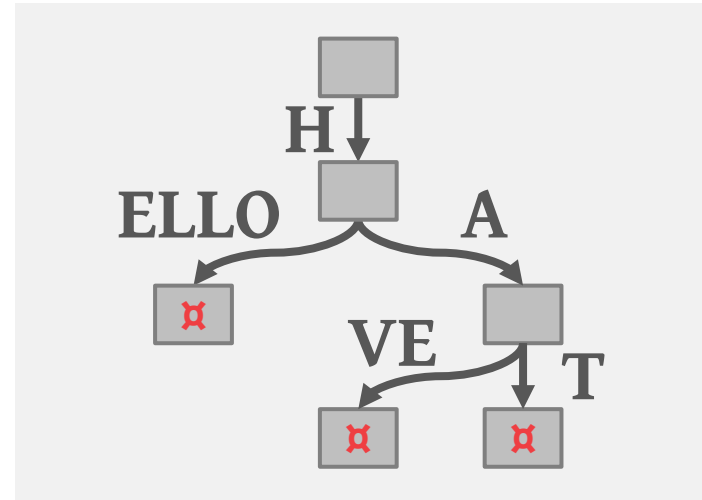


# TRIE VS. RADIX TREE

*Trie*

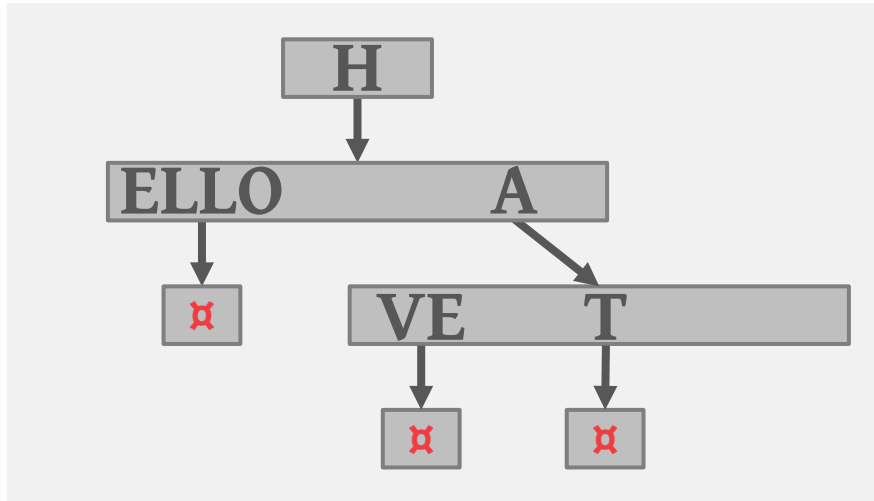


*Radix Tree*

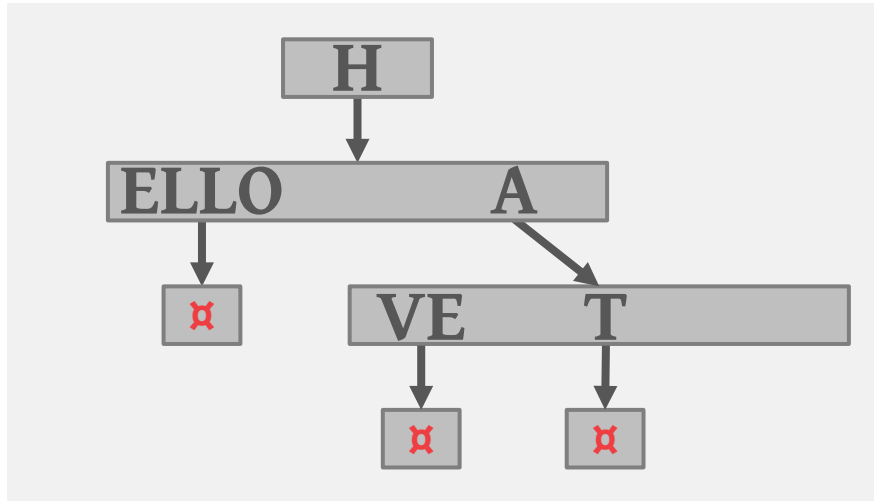


Keys: hello, hat, have

# ART INDEX: MODIFICATIONS

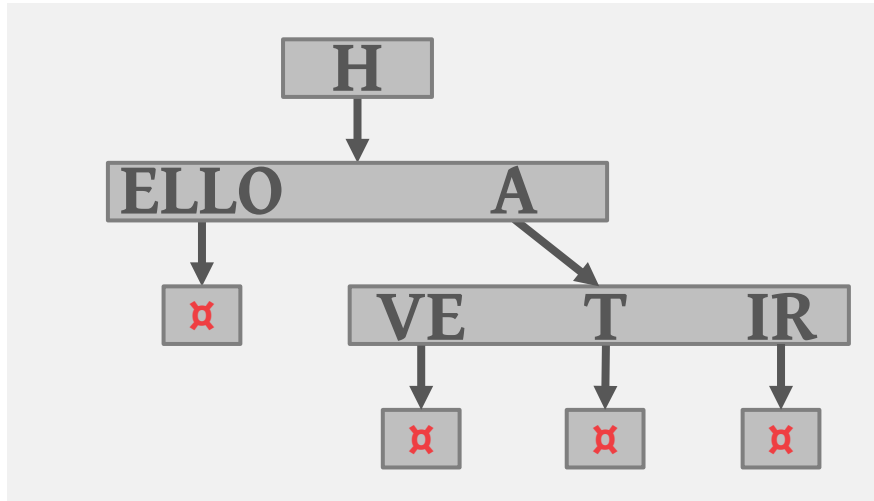


# ART INDEX: MODIFICATIONS



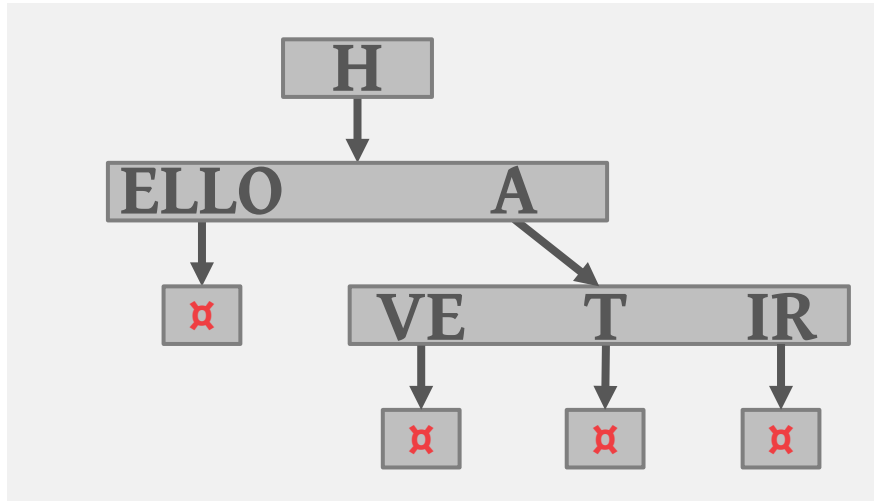
**Operation:** Insert hair

# ART INDEX: MODIFICATIONS



**Operation:** Insert hair

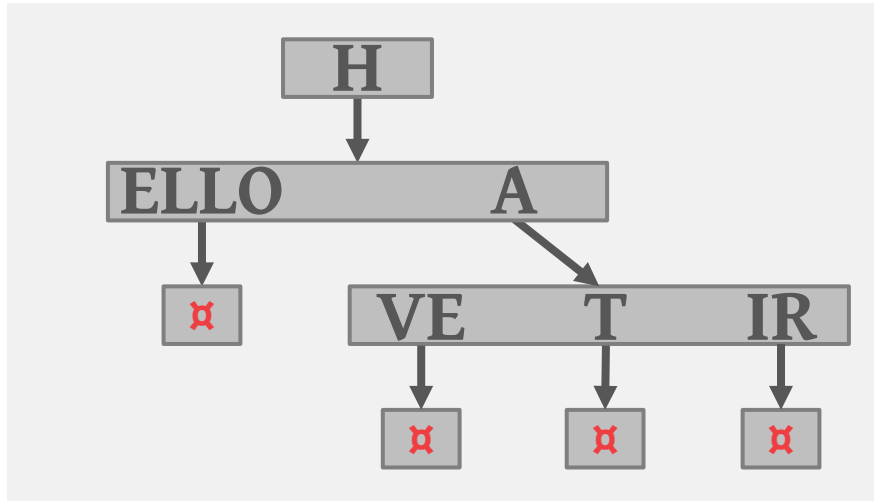
# ART INDEX: MODIFICATIONS



**Operation:** Insert hair

**Operation:** Delete hat, have

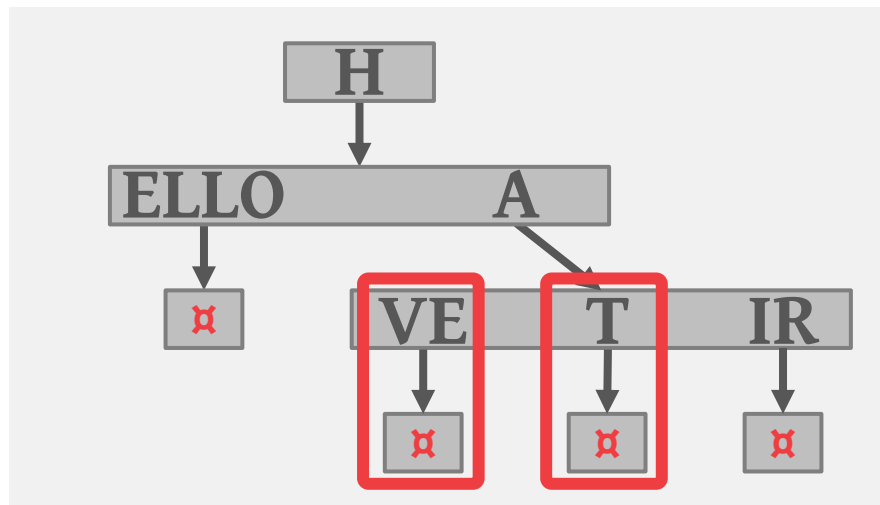
# ART INDEX: MODIFICATIONS



**Operation:** Insert hair

**Operation:** Delete hat, have

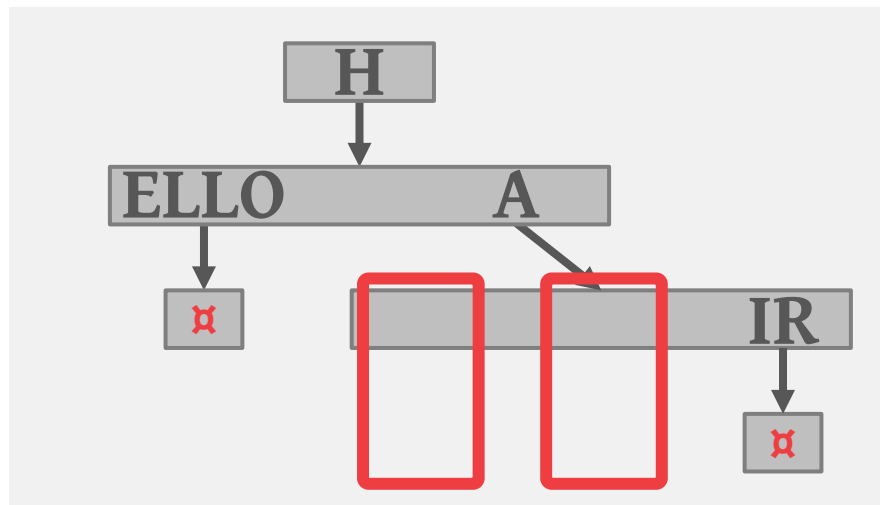
# ART INDEX: MODIFICATIONS



**Operation:** Insert hair

**Operation:** Delete hat, have

# ART INDEX: MODIFICATIONS

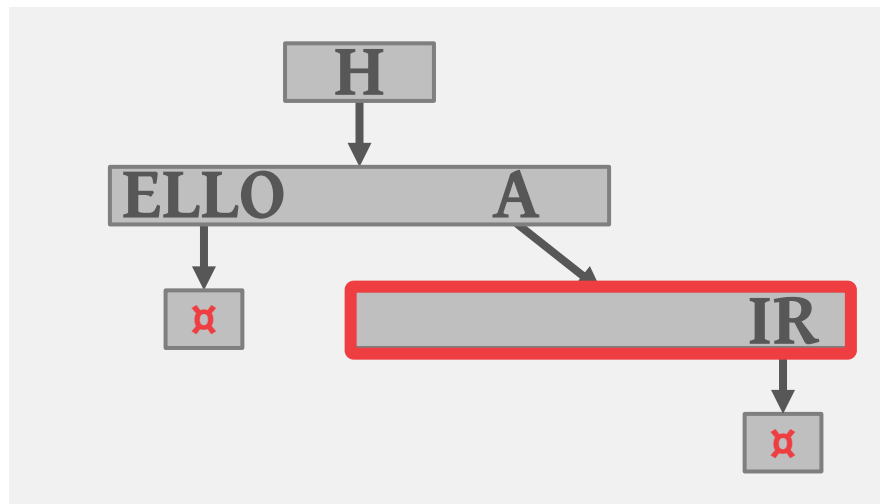


**Operation:** Insert hair

**Operation:** Delete hat, have



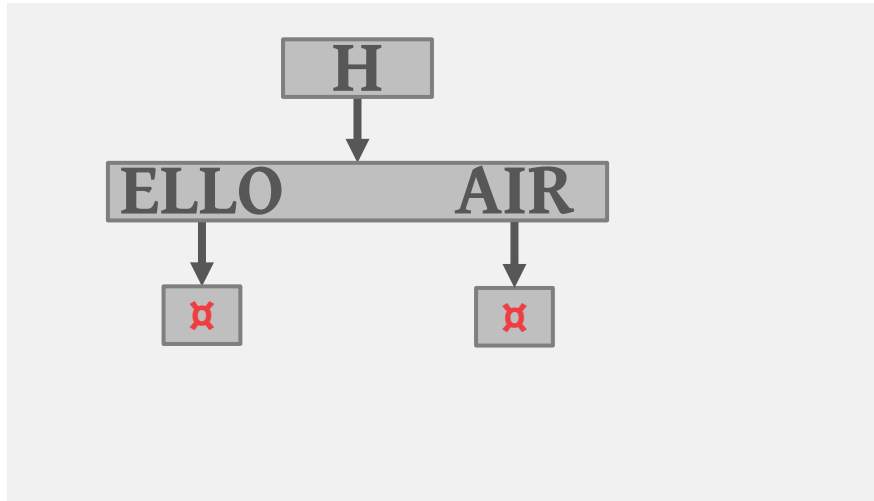
# ART INDEX: MODIFICATIONS



**Operation:** Insert hair

**Operation:** Delete hat, have

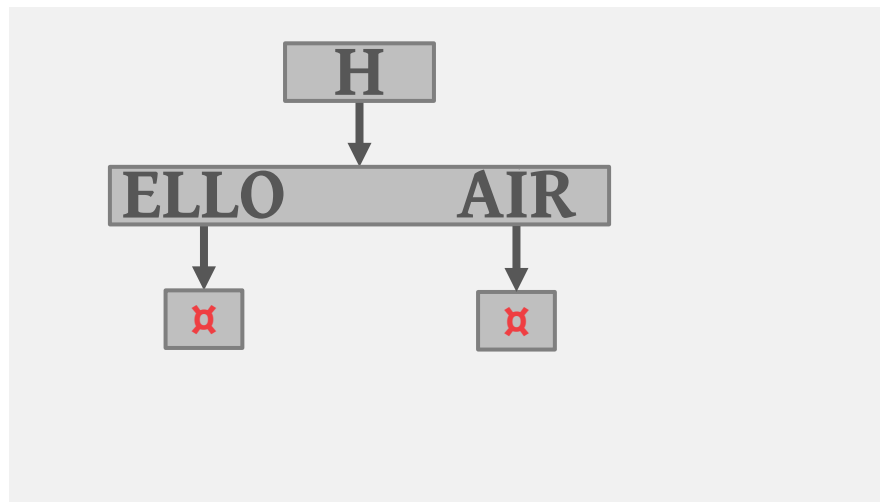
# ART INDEX: MODIFICATIONS



**Operation:** Insert hair

**Operation:** Delete hat, have

# ART INDEX: MODIFICATIONS



**Operation:** Insert hair

**Operation:** Delete hat, have

Note: The ART index described in 2013 is not latch-free.

# ART INDEX: BINARY COMPARABLE KEYS

---

Not all attribute types can be decomposed into binary comparable digits for a radix tree.

- **Unsigned Integers:** Byte order must be flipped for little endian machines.
- **Signed Integers:** Flip two's-complement so that negative numbers are smaller than positive.
- **Floats:** Classify into group (neg vs. pos, normalized vs. denormalized), then store as unsigned integer.
- **Compound:** Transform each attribute separately.

## PARTING THOUGHTS

---

Bw-Tree is probably the most dank database data structure in recent years.

Skip List is really easy to implement.

# NEXT CLASS

---

Indexing for OLAP workloads.

→ More from Microsoft Research...

Project #2 Announcement