Carnegie Mellon University

Trees Indexes (Part II)

Lecture #08

Database Systems
15-445/15-645
Fall 2018

Andy Pavlo
Computer Science
Carnegie Mellon Univ.

ADMINISTRIVIA

Project #1 is due Wednesday Sept 26th @ 11:59pm

Homework #2 is due Friday Sept 28th @ 11:59pm

Project #2 will be released on Wednesday Sept 26th. First checkpoint is due Monday Oct 8th.

TODAY'S AGENDA

Additional Index Usage

Skip Lists

Radix Trees

Inverted Indexes

IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints.

→ Primary Keys
→ Unique Constraints
→ Foreign Keys (?)

```
CREATE TABLE foo (
  id SERIAL PRIMARY KEY,
  val1 INT NOT NULL,
  val2 VARCHAR(32) UNIQUE
);
```

```
CREATE UNIQUE INDEX foo_pkey
                 ON foo (id);
```

```
CREATE UNIQUE INDEX foo_val2_key
                 ON foo (val2);
```

IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints.

→ Primary Keys
→ Unique Constraints
→ Foreign Keys (?)

```
CREATE INDEX foo_val1_key
            ON foo (val1);
```

```
CREATE TABLE foo (
  id SERIAL PRIMARY KEY,
  val1 INT NOT NULL,
  val2 VARCHAR(32) UNIQUE
);
```

```
CREATE TABLE bar (
  id INT REFERENCES foo (val1),
  val VARCHAR(32)
);
```

IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints.

→ Primary Keys
→ Unique Constraints
→ Foreign Keys (?)

```
CREATE INDEX foo_val1_key
             ON foo (val1);
```

```
CREATE TABLE foo (
  id SERIAL PRIMARY KEY,
  val1 INT NOT NULL,
  val2 VARCHAR(32) UNIQUE
);
```

```
CREATE TABLE bar (
  id INT REFERENCES foo (val1),
  val VARCHAR(32)
);
```

IMPLICIT INDEXES

Most DBMSs automatically create an index to enforce integrity constraints.

→ Primary Keys
→ Unique Constraints
→ Foreign Keys (?)

```
CREATE TABLE foo (
  id SERIAL PRIMARY KEY,
  val1 INT NOT NULL UNIQUE,
  val2 VARCHAR(32) UNIQUE
);
```

```
CREATE TABLE bar (
  id INT REFERENCES foo (val1),
  val VARCHAR(32)
);
```

PARTIAL INDEXES

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

One common use case is to partition indexes by date ranges.

→ Create a separate index per month, year.

```
CREATE INDEX idx_foo
         ON foo (a, b)
      WHERE c = 'WuTang';
```

PARTIAL INDEXES

Create an index on a subset of the entire table. This potentially reduces its size and the amount of overhead to maintain it.

One common use case is to partition indexes by date ranges.

→ Create a separate index per month, year.

```
CREATE INDEX idx_foo
          ON foo (a, b)
       WHERE c = 'WuTang';
```

```
SELECT b FROM foo
 WHERE a = 123
   AND c = 'WuTang';
```

COVERING INDEXES

If all of the fields needed to process the query are available in an index, then the DBMS does not need to retrieve the tuple.

This reduces contention on the DBMS's buffer pool resources.

```
CREATE INDEX idx_foo
          ON foo (a, b);
```

```
SELECT b FROM foo
 WHERE a = 123;
```

INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

Not part of the search key.

```
CREATE INDEX idx_foo
          ON foo (a, b)
     INCLUDE (c);
```

INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

Not part of the search key.

```
CREATE INDEX idx_foo
          ON foo (a, b)
     INCLUDE (c);
```

```
SELECT b FROM foo
 WHERE a = 123
   AND c = 'WuTang';
```

INDEX INCLUDE COLUMNS

Embed additional columns in indexes to support index-only queries.

Not part of the search key.

```
CREATE INDEX idx_foo
           ON foo (a, b)
        INCLUDE (c);
```

```
SELECT b FROM foo
 WHERE a = 123
   AND c = 'WuTang';
```

FUNCTIONAL/EXPRESSION INDEXES

The index does not need to store keys in the same way that they appear in their base table.

```
SELECT * FROM users
 WHERE EXTRACT(dow
        ↳FROM login) = 2;
```

FUNCTIONAL/EXPRESSION INDEXES

The index does not need to store keys in the same way that they appear in their base table.

```
SELECT * FROM users
 WHERE EXTRACT(dow
          ↳FROM login) = 2;
```

```
CREATE INDEX idx_user_login
    ON users (login);
```

FUNCTIONAL/EXPRESSION INDEXES

The index does not need to store keys in the same way that they appear in their base table.

You can use expressions when declaring an index.

```
SELECT * FROM users
 WHERE EXTRACT(dow
         ↳FROM login) = 2;
```

```
CREATE INDEX idx_user_login
    ON users (login);
```

FUNCTIONAL/EXPRESSION INDEXES

The index does not need to store keys in the same way that they appear in their base table.

You can use expressions when declaring an index.

```
SELECT * FROM users
 WHERE EXTRACT(dow
          ↳FROM login) = 2;
```

```
CREATE INDEX _user_login
    ON users gin);
```

```
CREATE INDEX idx_user_login
    ON users (EXTRACT(dow FROM login));
```

FUNCTIONAL/EXPRESSION INDEXES

The index does not need to store keys in the same way that they appear in their base table.

You can use expressions when declaring an index.

```
SELECT * FROM users
 WHERE EXTRACT(dow
       FROM login) = 2;
```

```
CREATE INDEX idx_user_login
    ON users (login);
```

```
CREATE INDEX idx_user_login
    ON users (EXTRACT(dow FROM login));
```

FUNCTIONAL/EXPRESSION INDEXES

The index does not need to store keys in the same way that they appear in their base table.

You can use expressions when declaring an index.

```
SELECT * FROM users
 WHERE EXTRACT(dow
  FROM login) = 2;
```

```
CREATE INDEX idx_user_login
    ON users (login);
```

```
CREATE INDEX idx_user_login
    ON users (EXTRACT(dow FROM login));
```

```
CREATE INDEX idx_user_login
    ON foo (login)
 WHERE EXTRACT(dow FROM login) = 2;
```

OBSERVATION

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

All operations have to linear search.

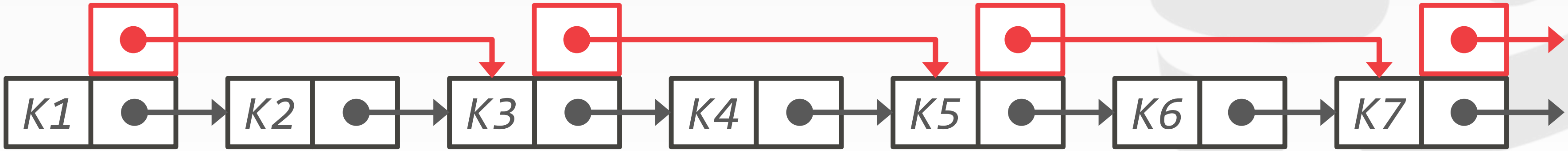
→ Average Cost: **O(N)**

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)**

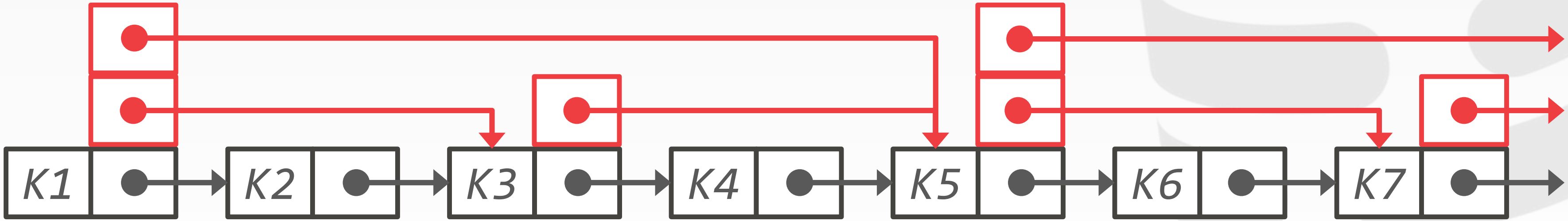


OBSERVATION

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

Skip lists are a data structure that can be used in place of balanced trees. Skip lists use probabilistic balancing rather than strictly enforced balancing and as a result the algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees.

William Pugh

Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in order, produce degenerate data structures that give very poor performance. If it were possible to randomly permute the list of items to be inserted, trees would work well with high probability for any input sequence. In most cases queries must be answered on-line, so randomly permuting the input is impractical. *Balanced* tree algorithms re-arrange the tree as operations are performed to maintain certain balance conditions and assure good performance.

Skip lists are a probabilistic alternative to balanced trees. Skip lists are balanced by consulting a random number generator. Although skip lists have bad worst-case performance, no input sequence consistently produces the worst-case performance (much like quicksort when the pivot element is chosen randomly). It is very unlikely a skip list data structure will be significantly unbalanced (e.g., for a dictionary of more than 250 elements, the chance that a search will take more than 3 times the expected time is less than one in a million). Skip lists have balance properties similar to that of search trees built by random insertions, yet do not require insertions to be random.

Balancing a data structure probabilistically is easier than explicitly maintaining the balance. For many applications, skip lists are a more natural representation than trees, also leading to simpler algorithms. The simplicity of skip list algorithms makes them easier to implement and provides significant constant factor speed improvements over balanced tree and self-adjusting tree algorithms. Skip lists are also very space efficient. They can easily be configured to require an average of $1\,^1/_3$ pointers per element (or even less) and do not require balance or priority information to be stored with each node.

SKIP LISTS

We might need to examine every node of the list when searching a linked list (*Figure 1a*). If the list is stored in sorted order and every other node of the list also has a pointer to the node two ahead it in the list (*Figure 1b*), we have to examine no more than $\lceil n/2 \rceil + 1$ nodes (where n is the length of the list).

Also giving every fourth node a pointer four ahead (*Figure 1c*) requires that no more than $\lceil n/4 \rceil + 2$ nodes be examined. If every $(2^i)^{th}$ node has a pointer 2^i nodes ahead (*Figure 1d*), the number of nodes that must be examined can be reduced to $\lceil \log_2 n \rceil$ while only doubling the number of pointers. This data structure could be used for fast searching, but insertion and deletion would be impractical.

A node that has k forward pointers is called a *level k* node. If every $(2^i)^{th}$ node has a pointer 2^i nodes ahead, then levels of nodes are distributed in a simple pattern: 50% are level 1, 25% are level 2, 12.5% are level 3 and so on. What would happen if the levels of nodes were chosen randomly, but in the same proportions (e.g., as in Figure 1*e*)? A node's i^{th} forward pointer, instead of pointing 2^{i-1} nodes ahead, points to the next node of level i or higher. Insertions or deletions would require only local modifications; the level of a node, chosen randomly when the node is inserted, need never change. Some arrangements of levels would give poor execution times, but we will see that such arrangements are rare. Because these data structures are linked lists with extra pointers that skip over intermediate nodes, I named them *skip lists*.

SKIP LIST ALGORITHMS

This section gives algorithms to search for, insert and delete elements in a dictionary or symbol table. The *Search* operation returns the contents of the value associated with the desired key or *failure* if the key is not present. The *Insert* operation associates a specified key with a new value (inserting the key if it had not already been present). The *Delete* operation deletes the specified key. It is easy to support additional operations such as "find the minimum key" or "find the next key".

Each element is represented by a node, the level of which is chosen randomly when the node is inserted without regard for the number of elements in the data structure. A *level i* node has *i forward* pointers, indexed 1 through *i*. We do not need to store the level of a node in the node. Levels are capped at some appropriate constant *MaxLevel*. The *level* of a list is the maximum level currently in the list (or 1 if the list is empty). The *header* of a list has forward pointers at levels one through *MaxLevel*. The forward pointers of the header at levels higher than the current maximum level of the list point to NIL.

RocksDB MEMSQL WIREDTIGER

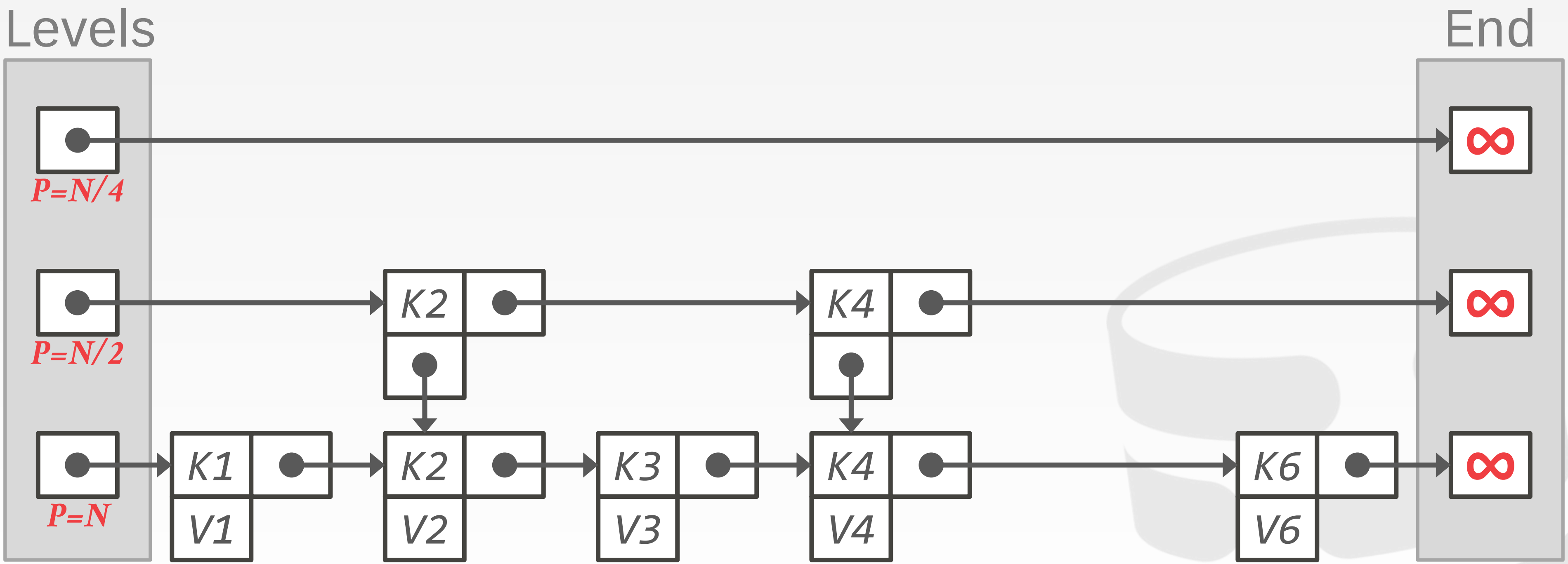
SKIP LISTS

A collection of lists at different levels

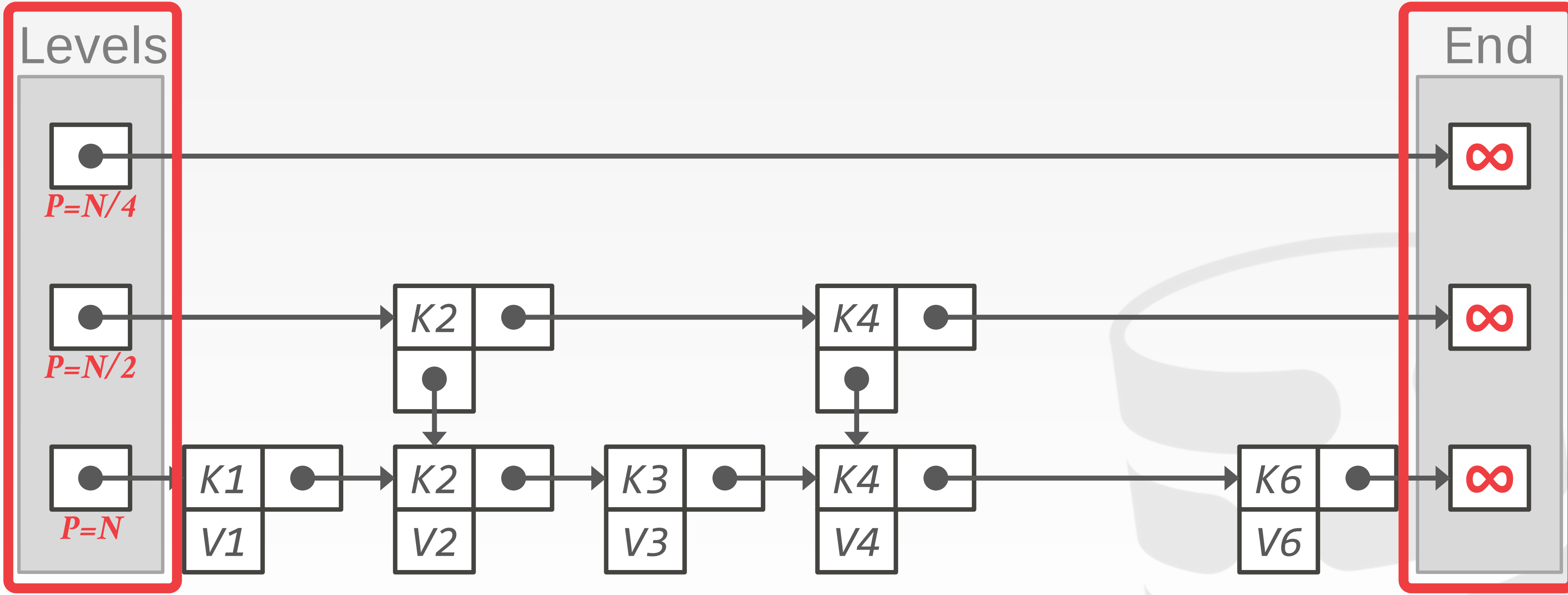
→ Lowest level is a sorted, singly linked list of all keys
→ 2nd level links every other key
→ 3rd level links every fourth key
→ In general, a level has half the keys of one below it

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.

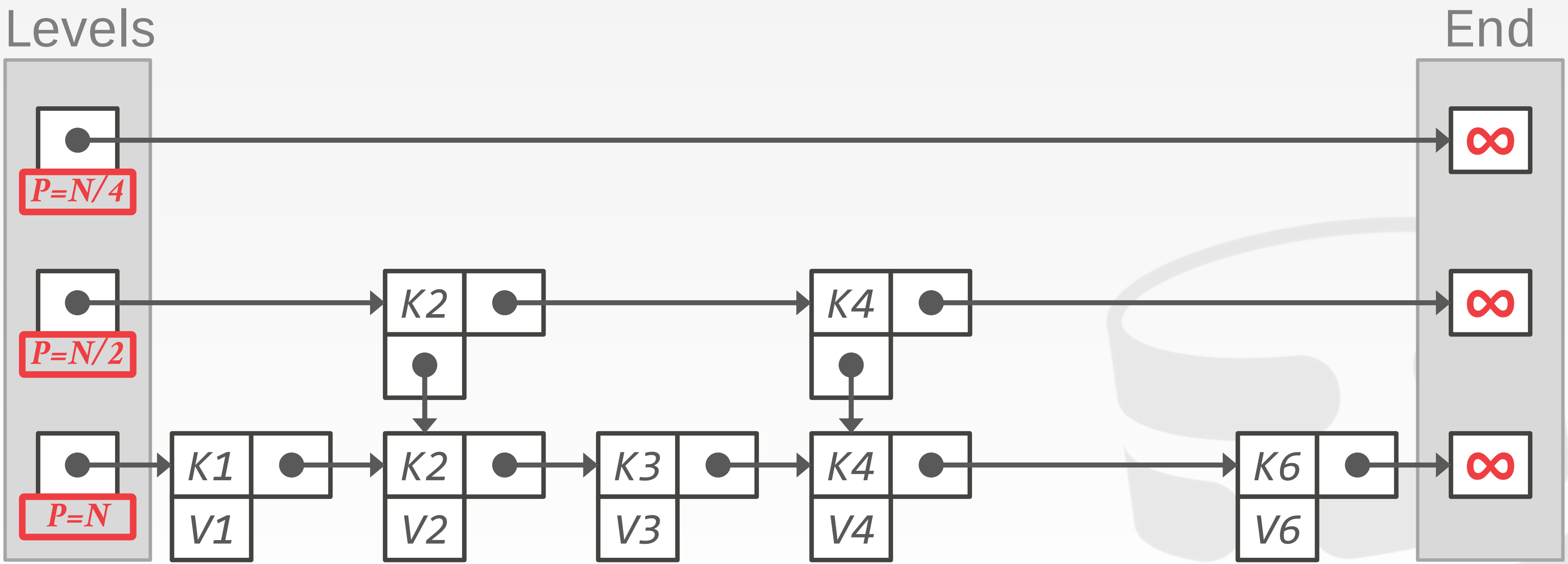
SKIP LISTS: EXAMPLE



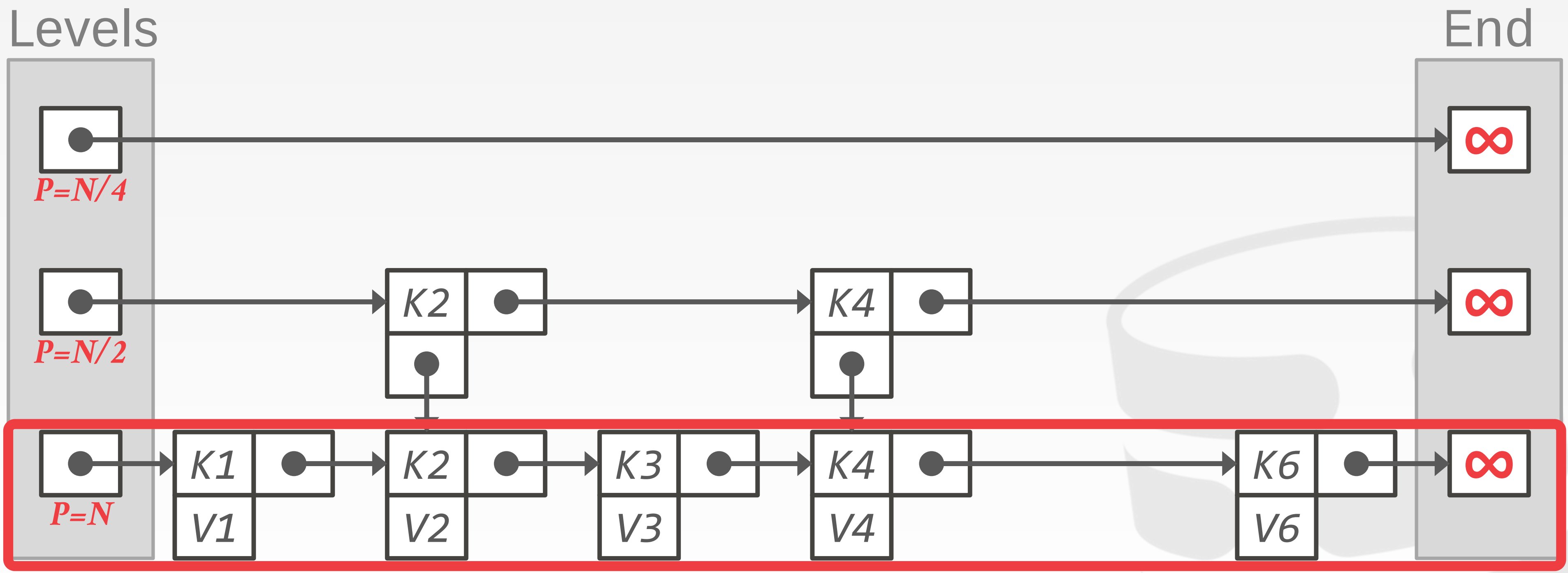
SKIP LISTS: EXAMPLE



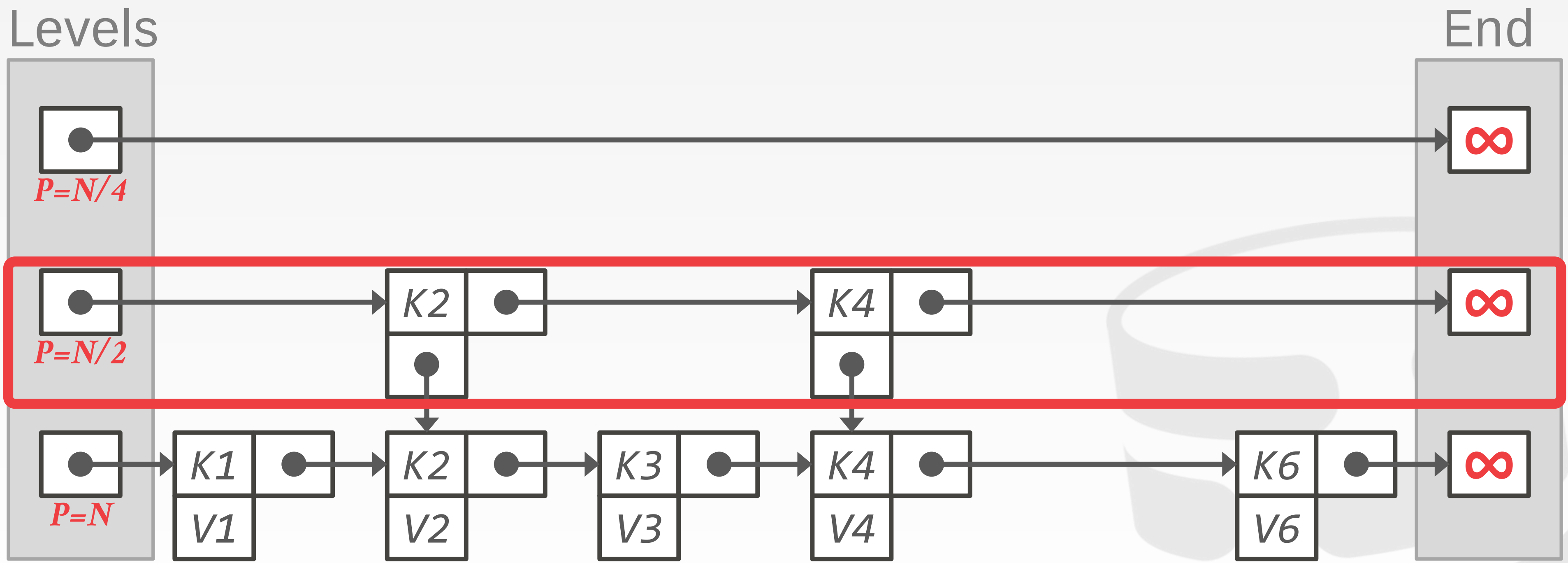
SKIP LISTS: EXAMPLE



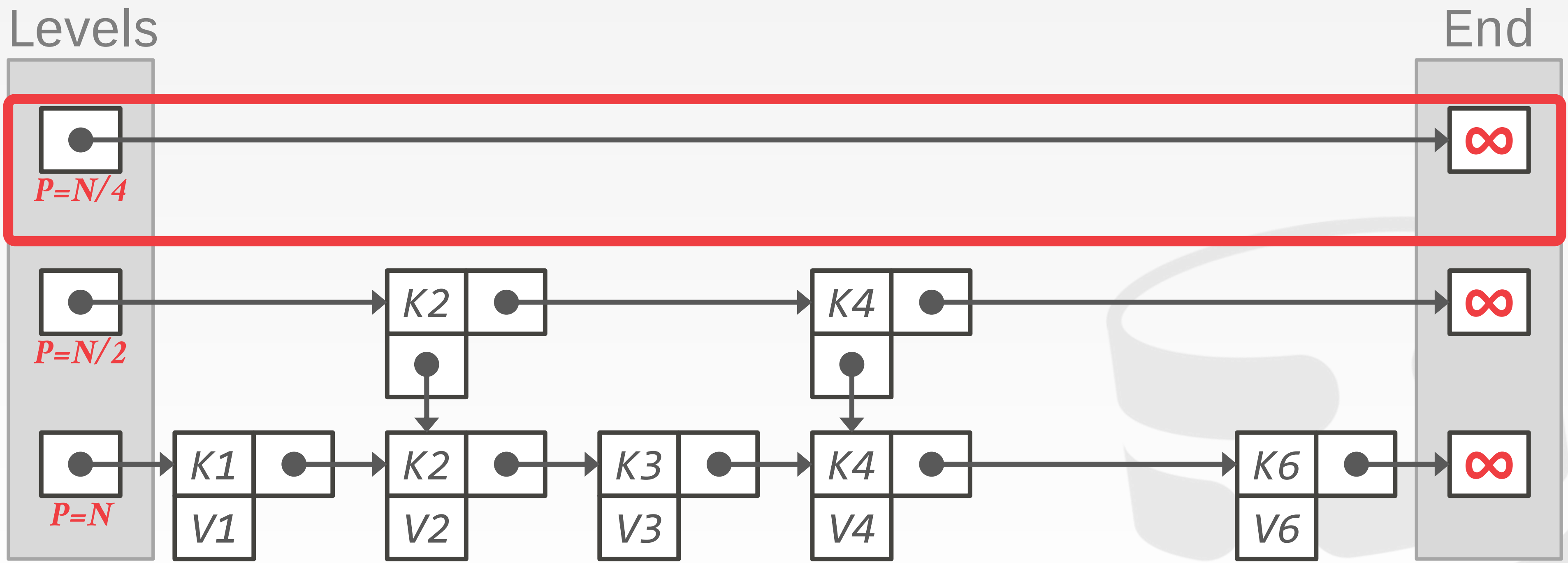
SKIP LISTS: EXAMPLE



SKIP LISTS: EXAMPLE

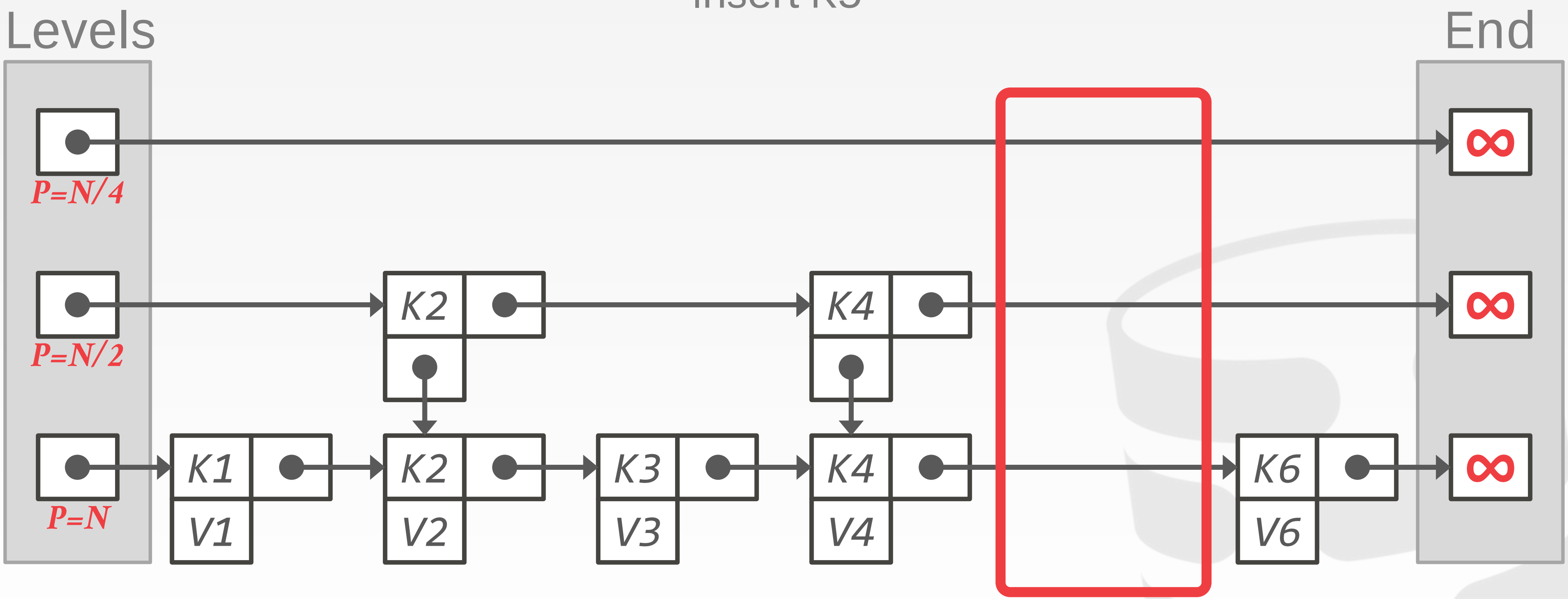


SKIP LISTS: EXAMPLE



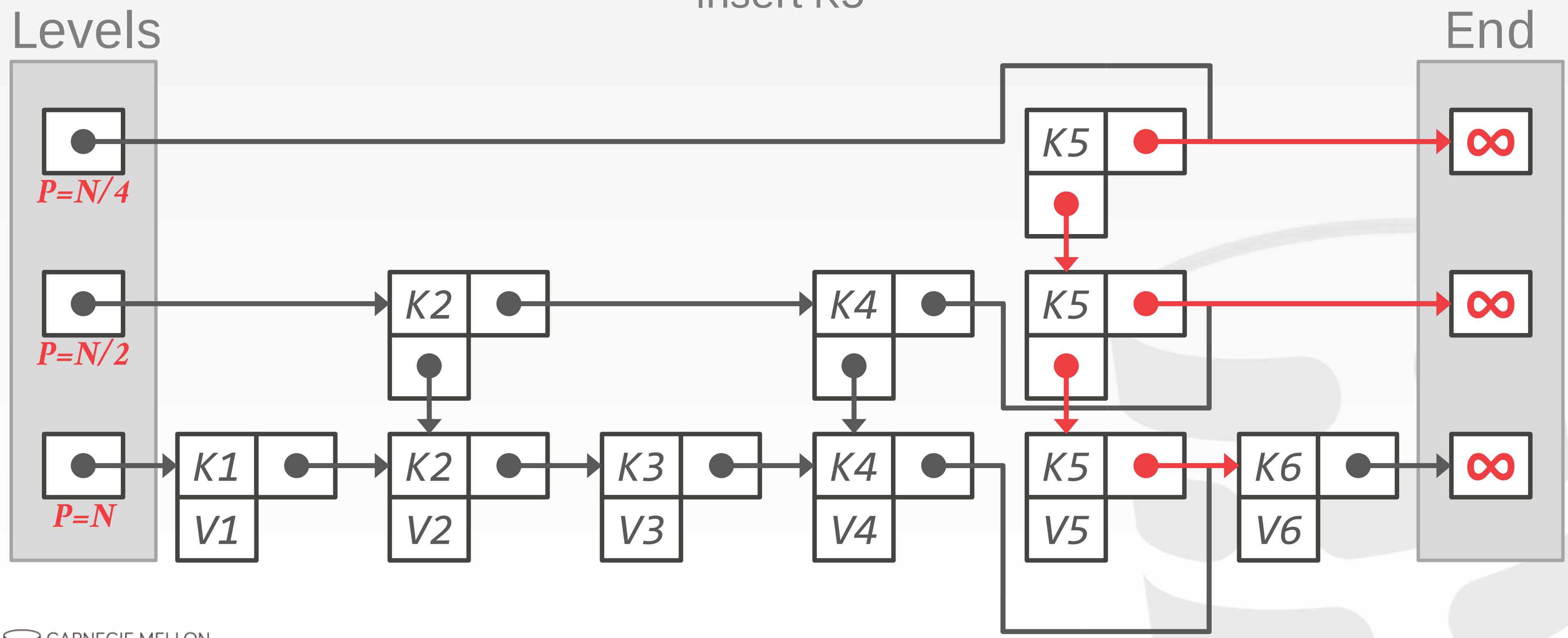
SKIP LISTS: INSERT

Insert K5



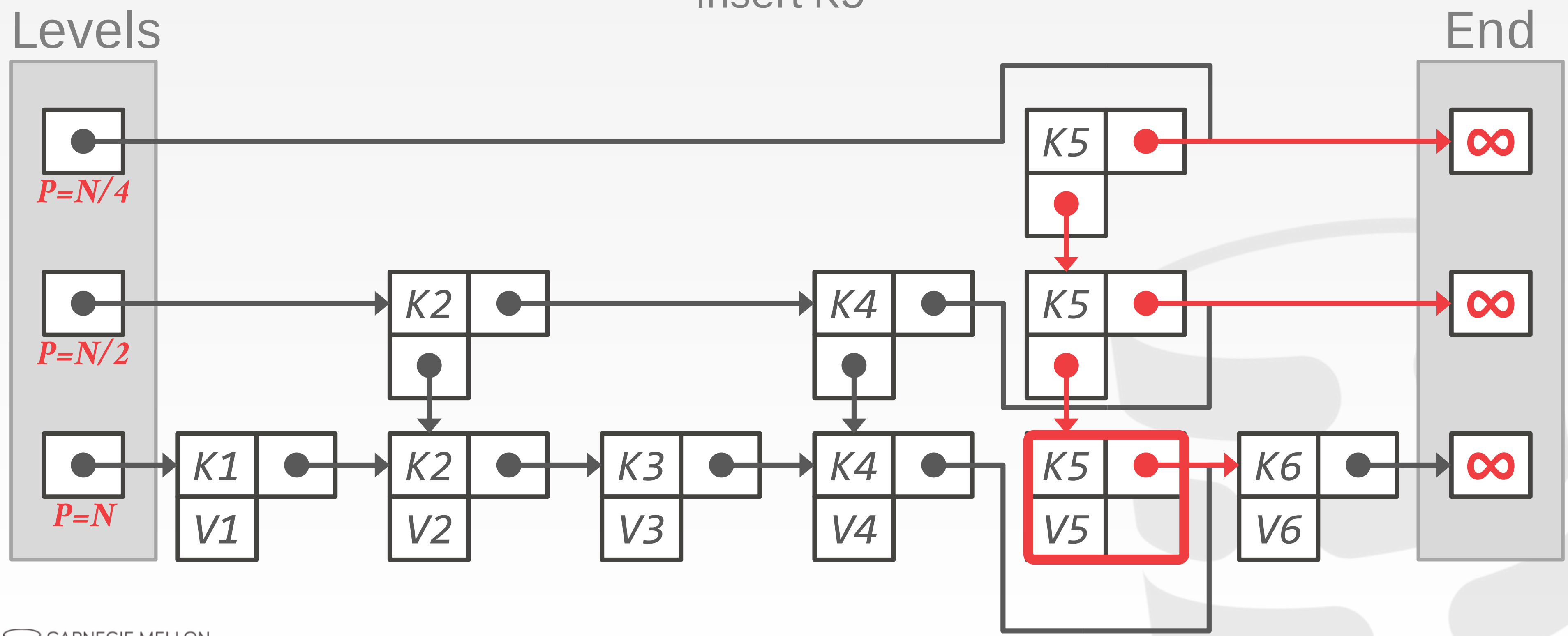
SKIP LISTS: INSERT

Insert K5



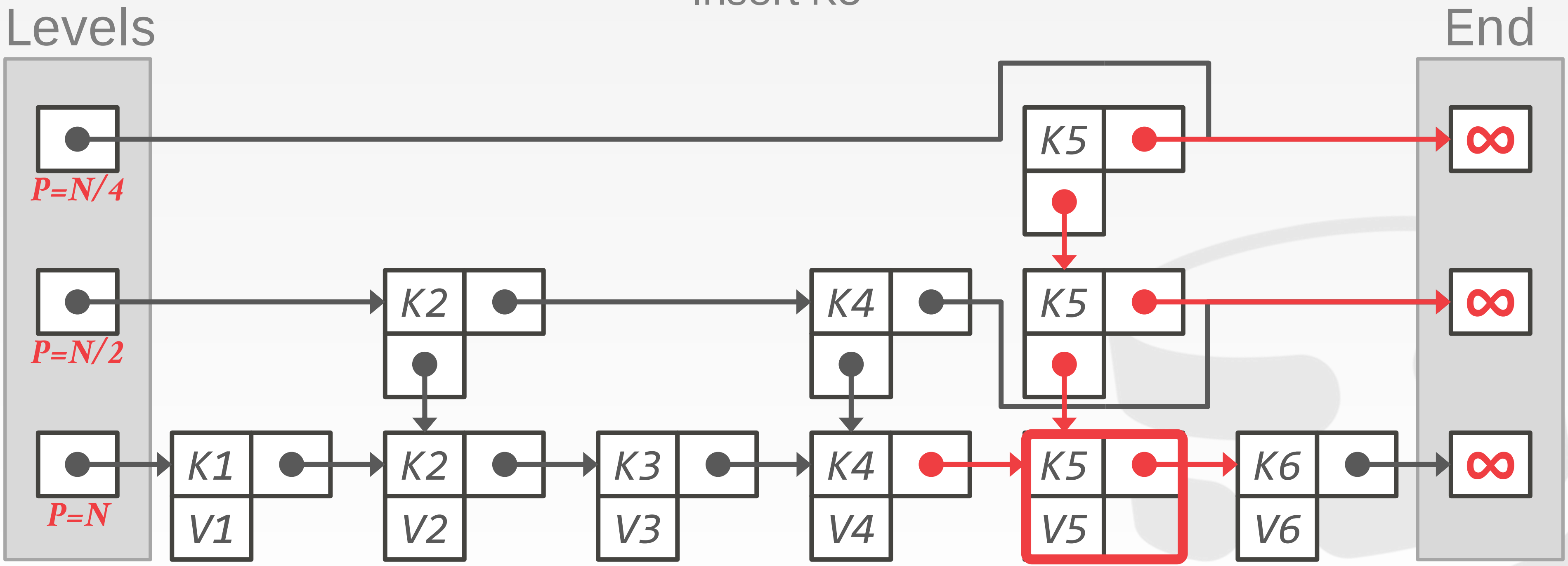
SKIP LISTS: INSERT

Insert K5



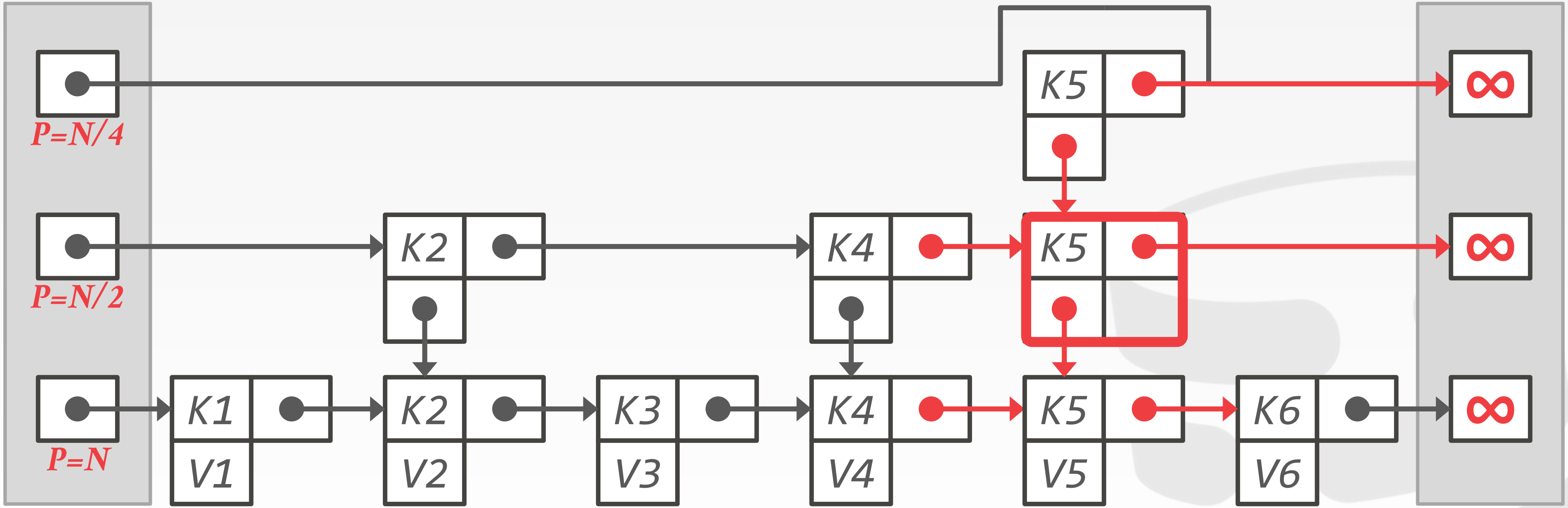
SKIP LISTS: INSERT

Insert K5



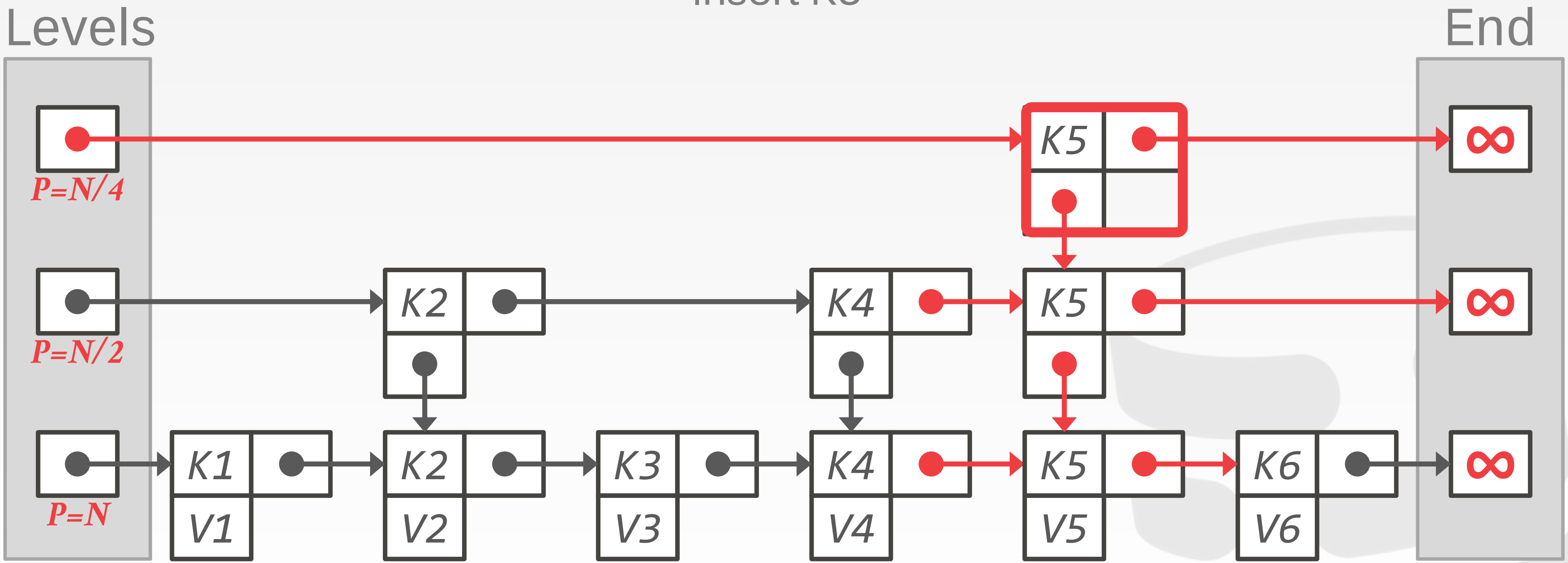
SKIP LISTS: INSERT

Insert K5



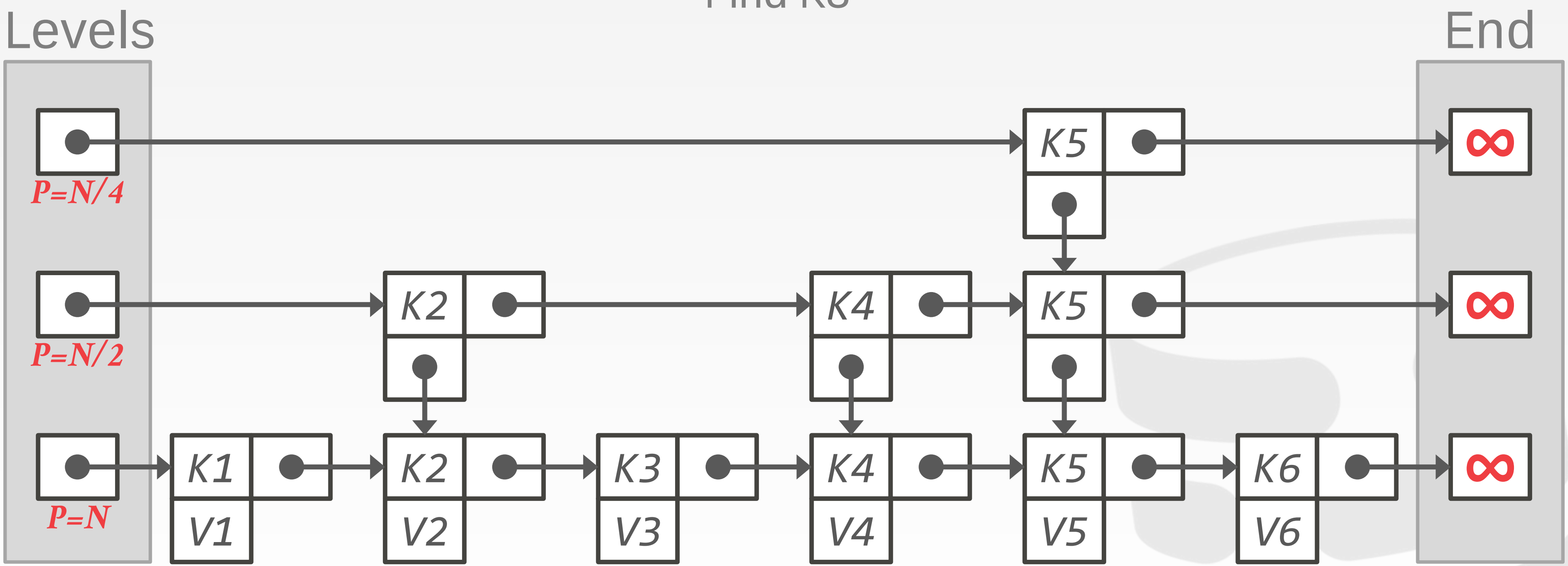
SKIP LISTS: INSERT

Insert K5



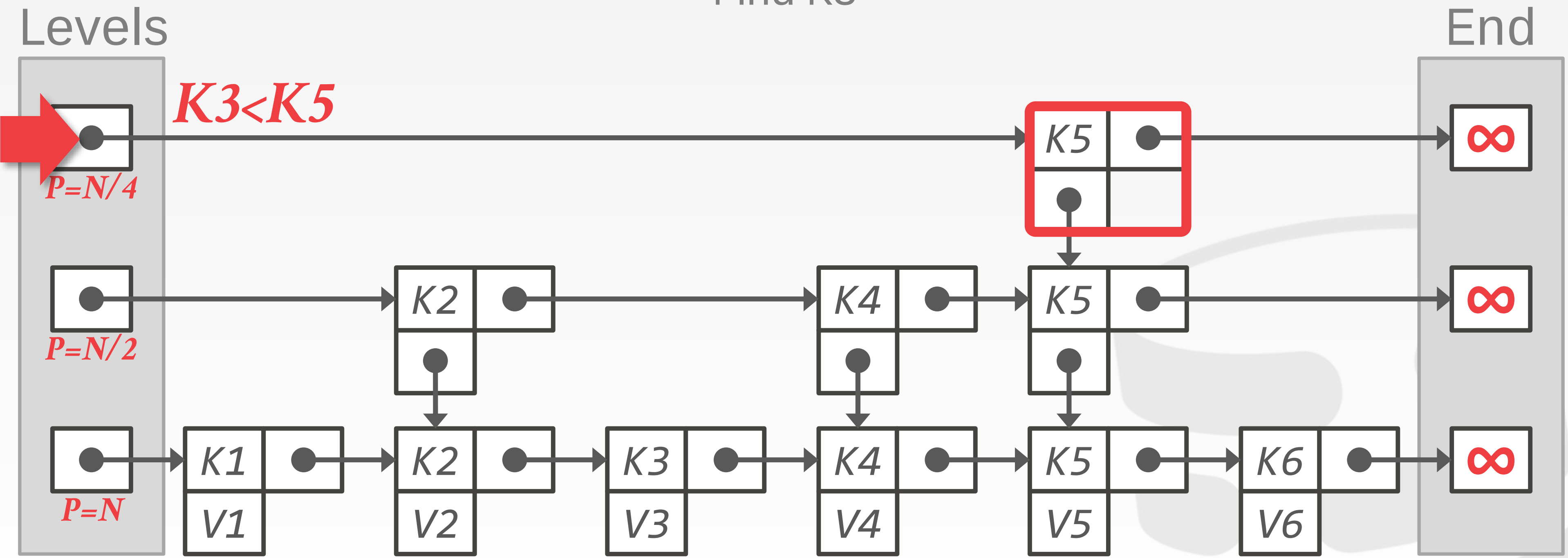
SKIP LISTS: SEARCH

Find K3



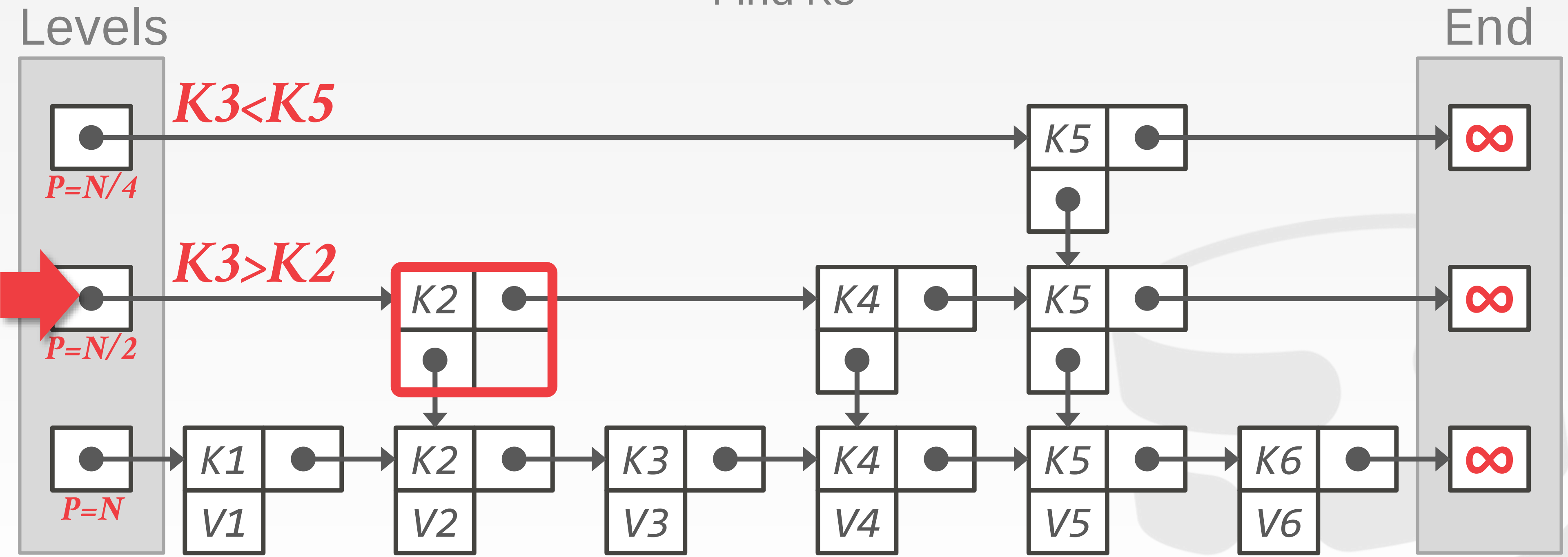
SKIP LISTS: SEARCH

Find K3



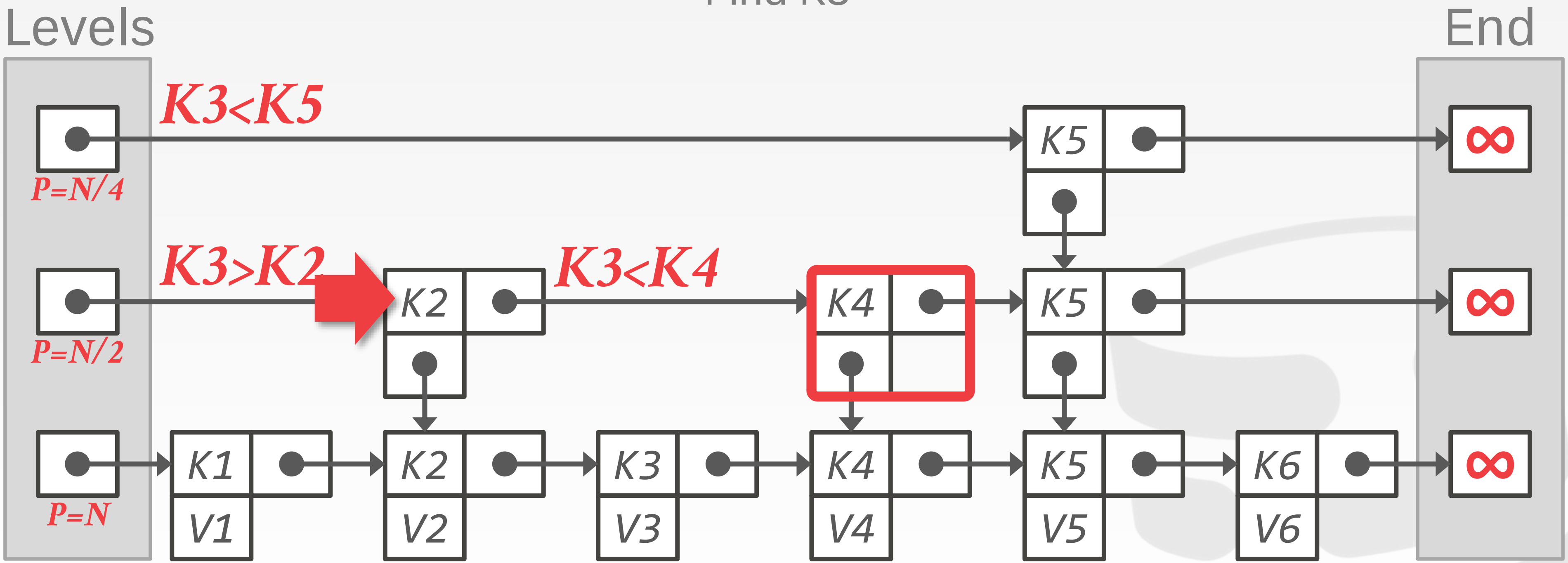
SKIP LISTS: SEARCH

Find K3



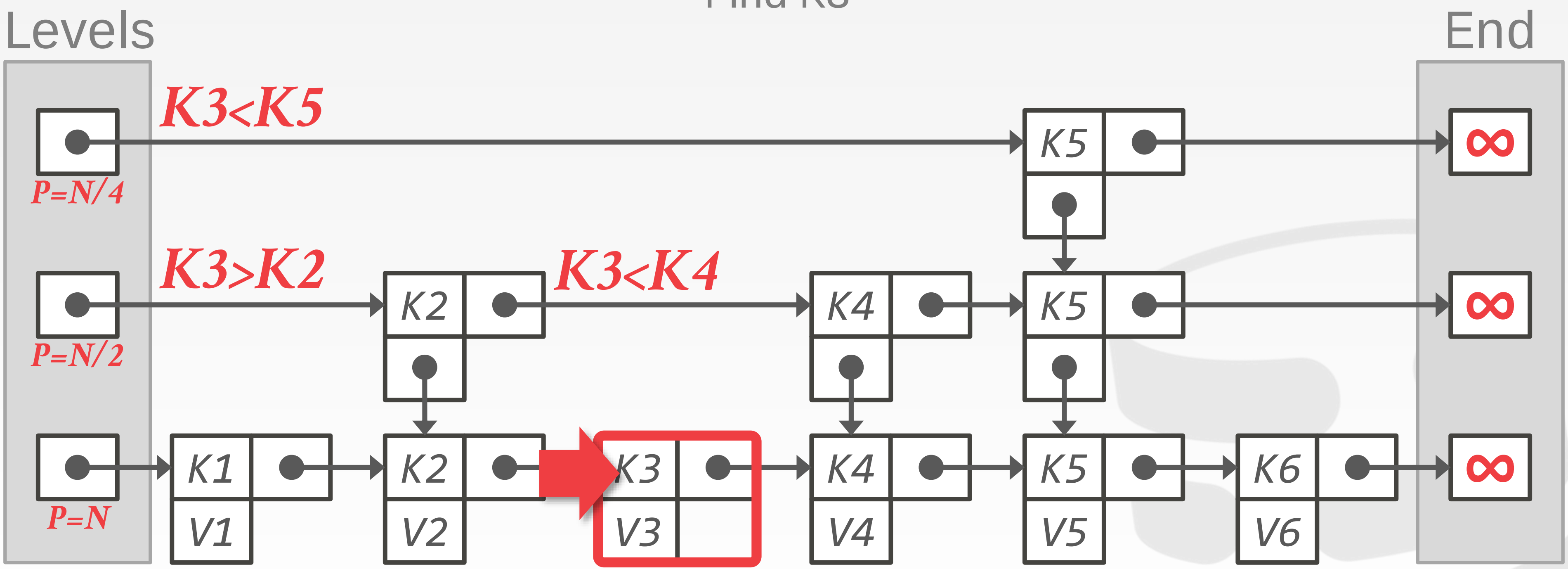
SKIP LISTS: SEARCH

Find K3



SKIP LISTS: SEARCH

Find K3



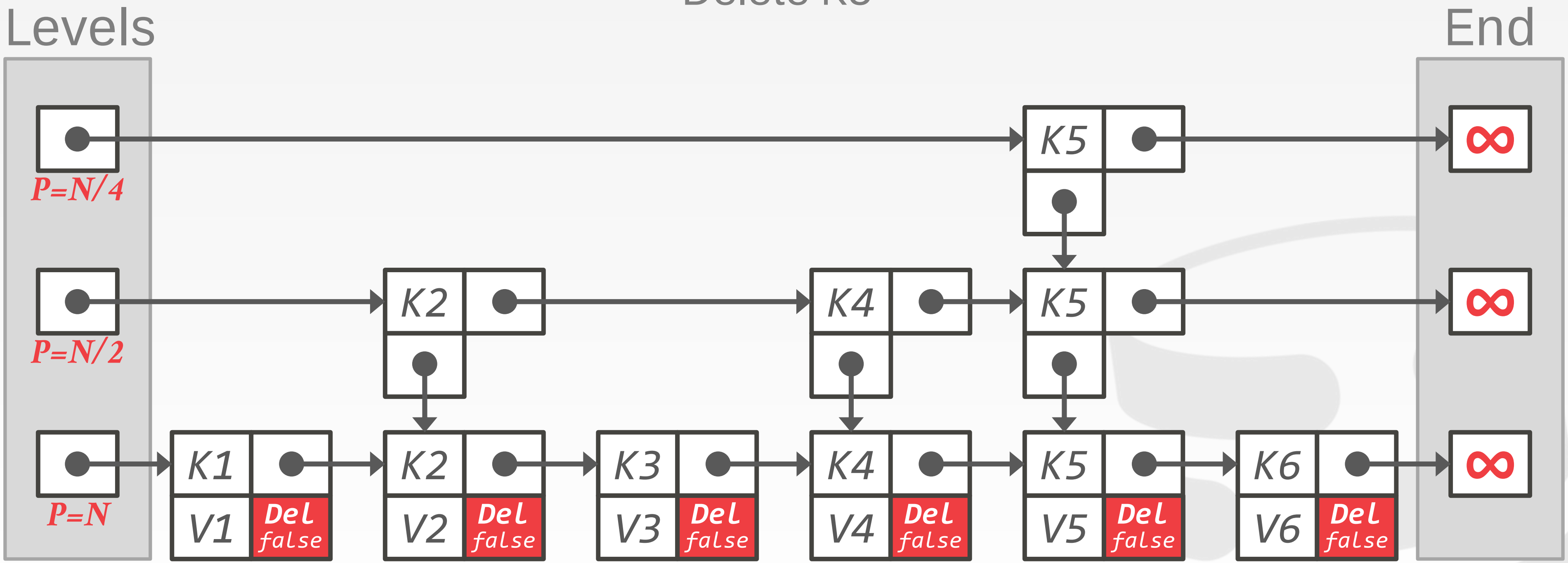
SKIP LISTS: DELETE

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.

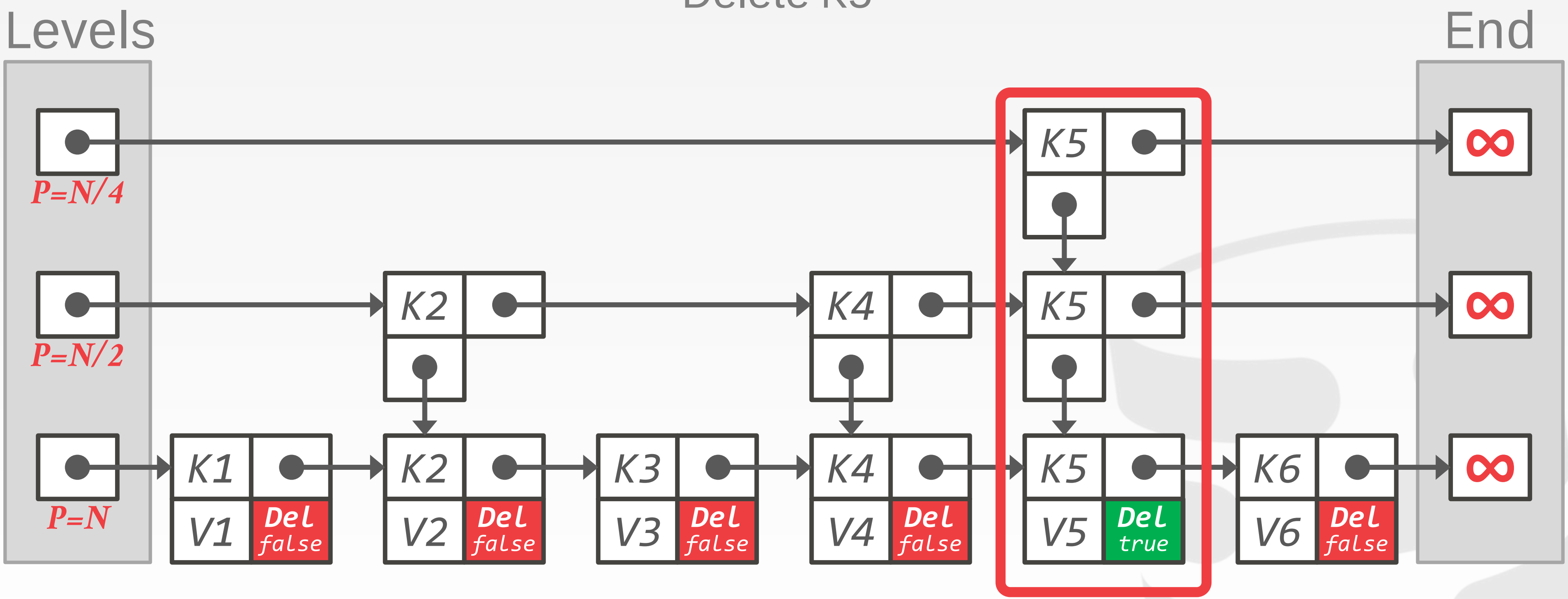
SKIP LISTS: DELETE

Delete K5



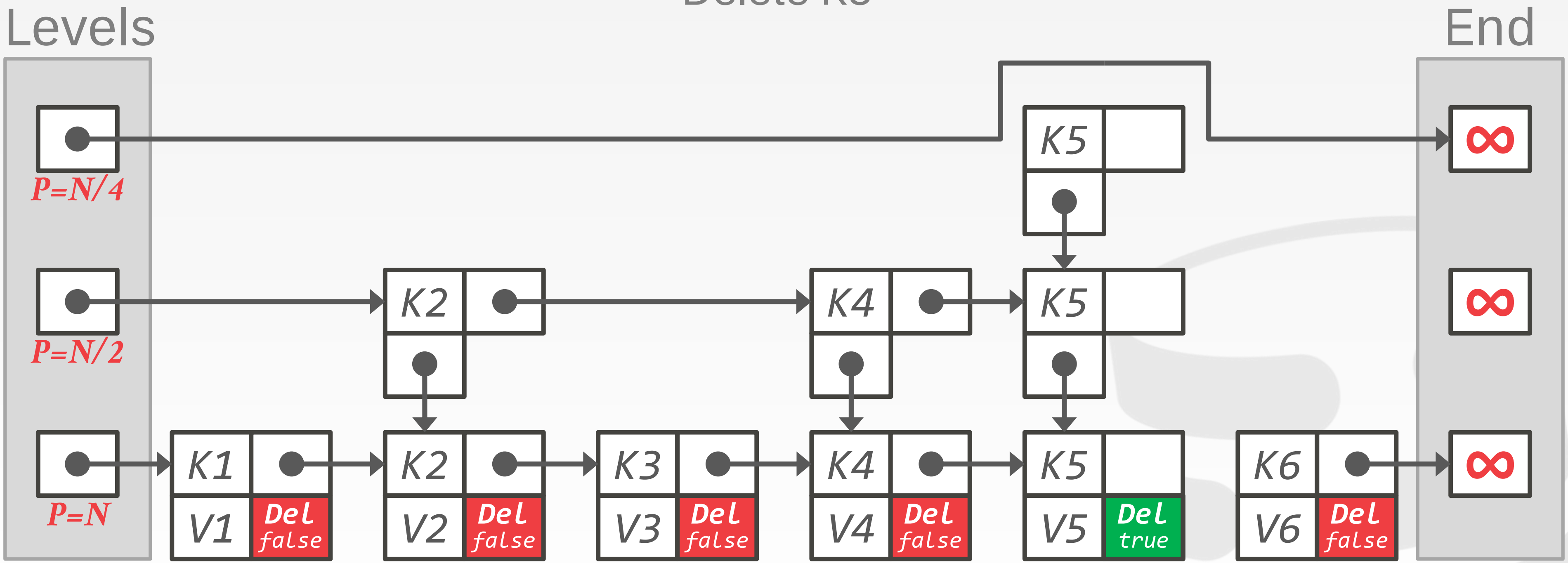
SKIP LISTS: DELETE

Delete K5



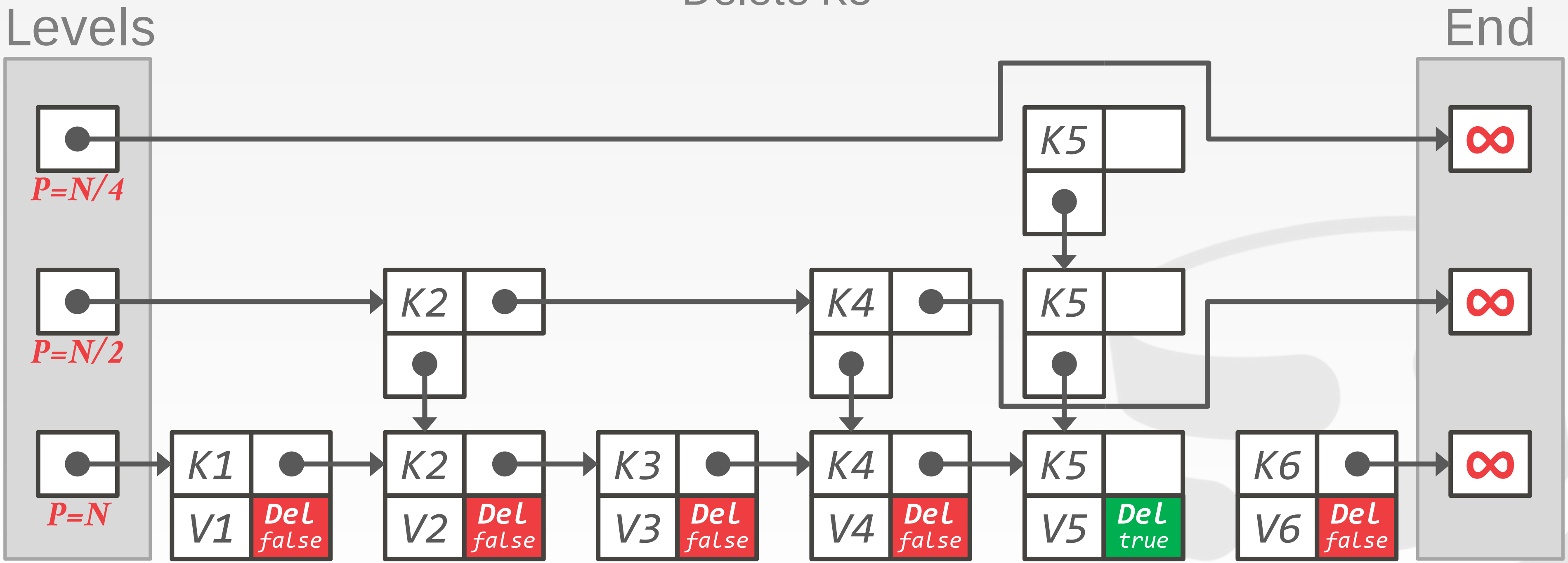
SKIP LISTS: DELETE

Delete K5



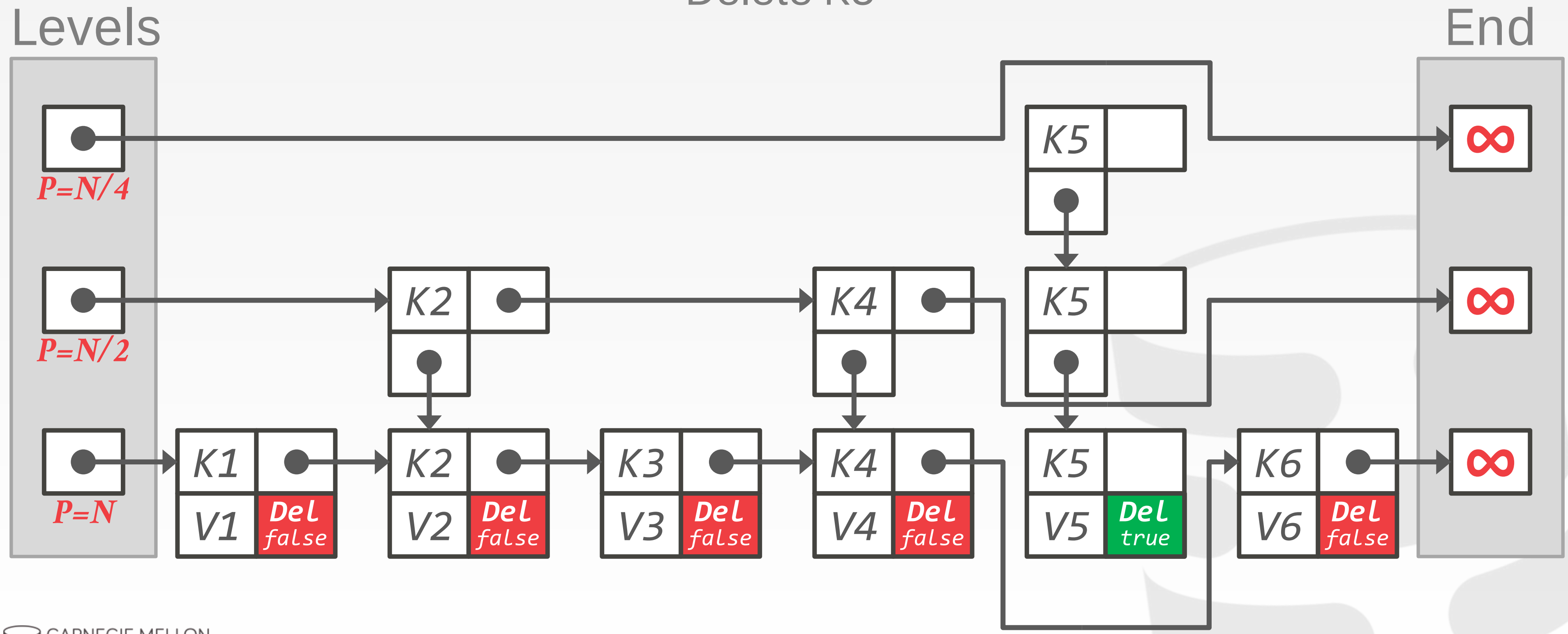
SKIP LISTS: DELETE

Delete K5



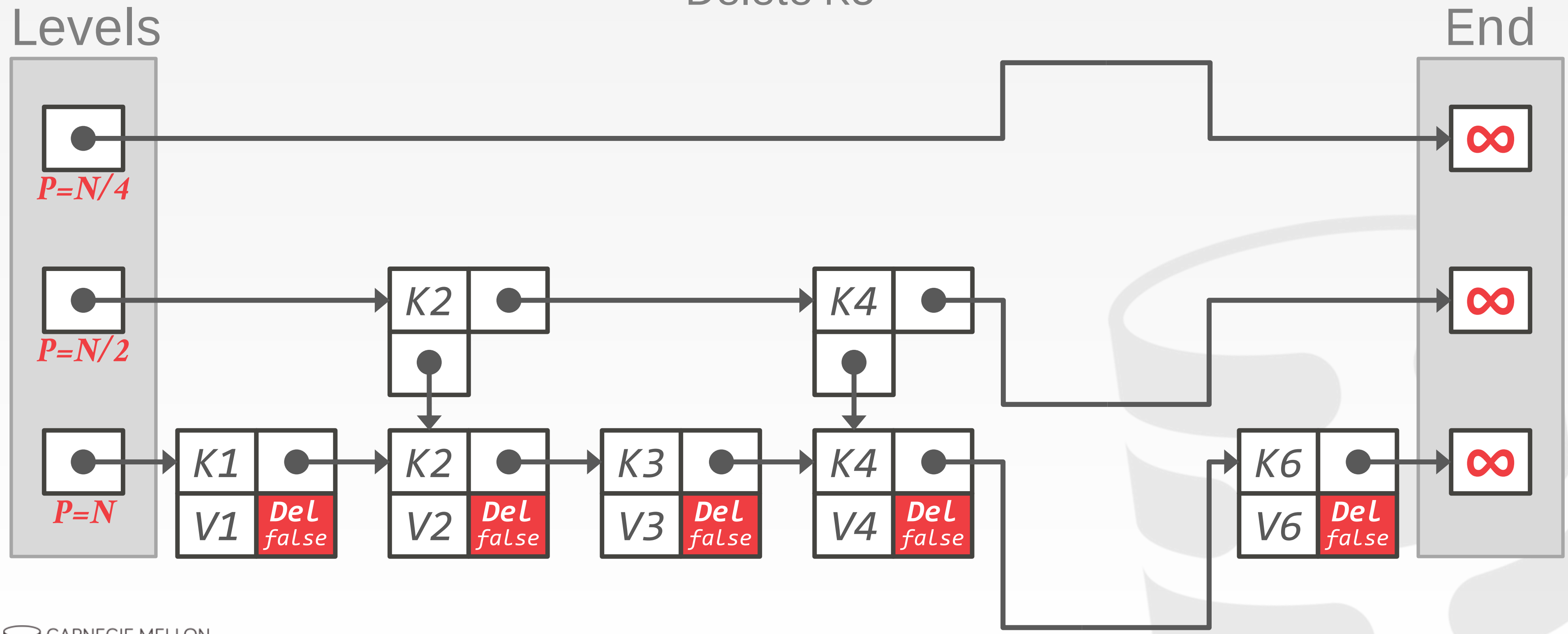
SKIP LISTS: DELETE

Delete K5



SKIP LISTS: DELETE

Delete K5



SKIP LISTS

Advantages:

→ Uses less memory than a typical B+Tree if you don't include reverse pointers.
→ Insertions and deletions do not require rebalancing.

Disadvantages:

→ Not disk/cache friendly because they do not optimize locality of references.
→ Reverse search is non-trivial.

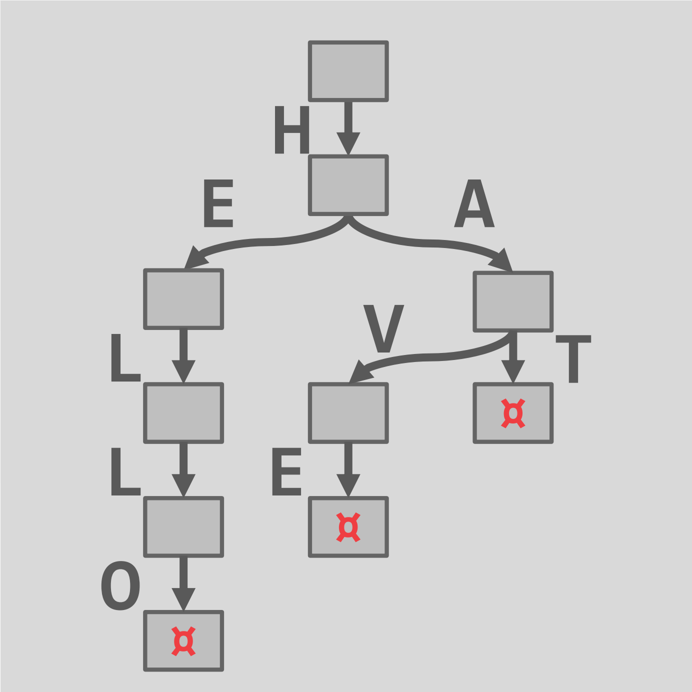
RADIX TREE

Represent keys as individual digits. This allows threads to examine prefixes one-by-one instead of comparing entire key.

→ 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.

TRIE VS. RADIX TREE

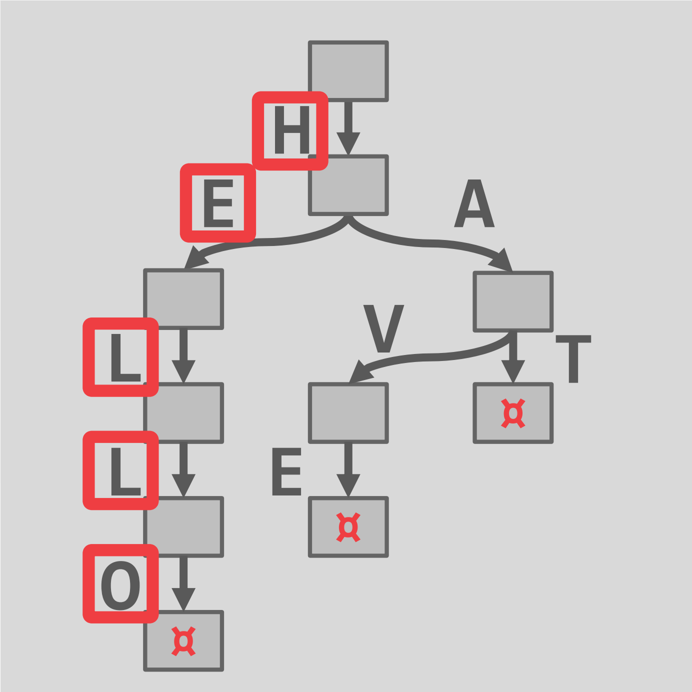
Trie



Keys: **HELLO, HAT, HAVE**

TRIE VS. RADIX TREE

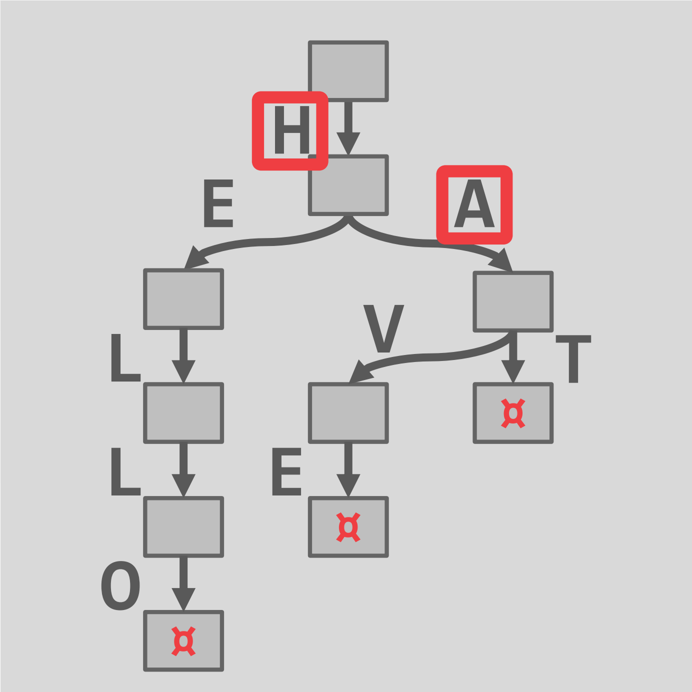
Trie



Keys: **HELLO HAT, HAVE**

TRIE VS. RADIX TREE

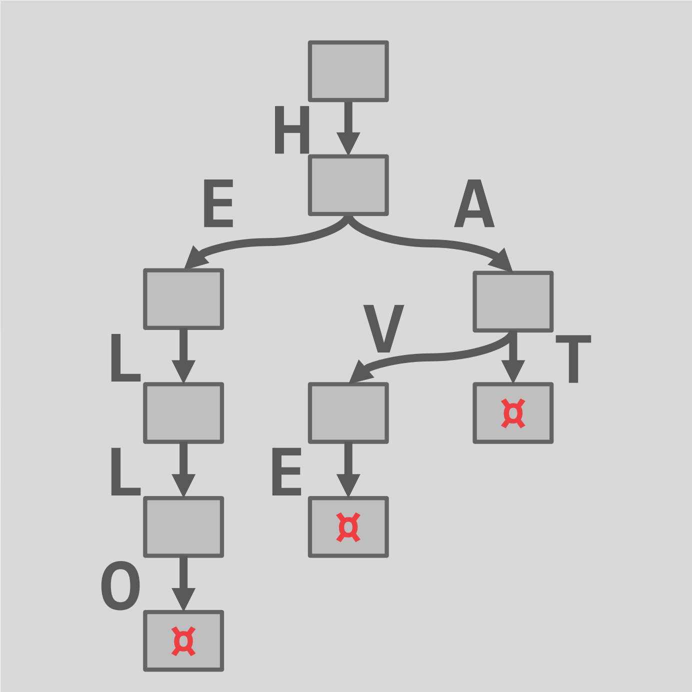
Trie



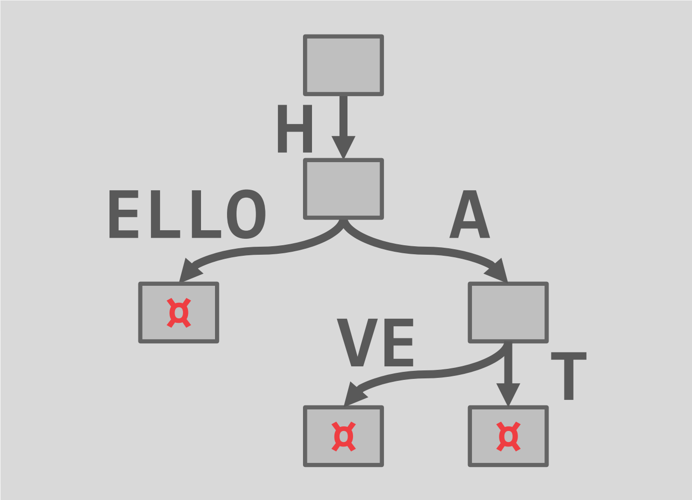
Keys: **HELLO, HAT, HAVE**

TRIE VS. RADIX TREE

Trie



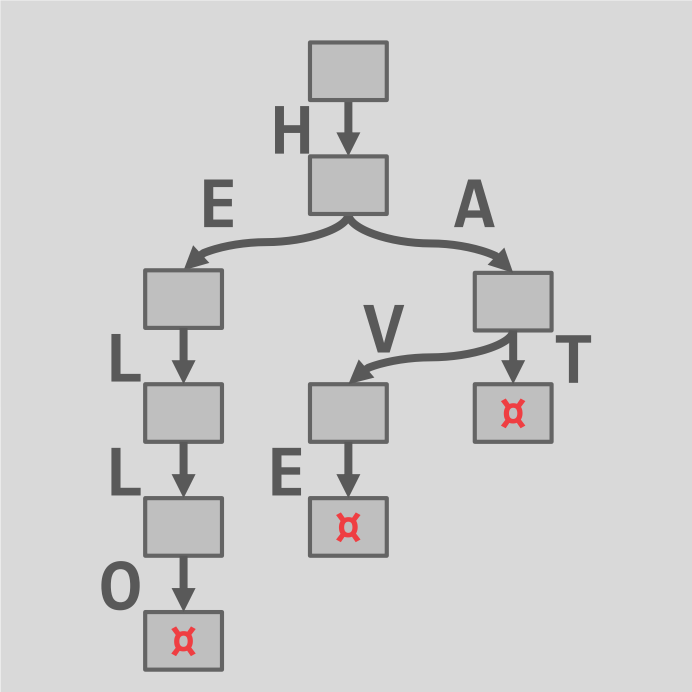
Radix Tree



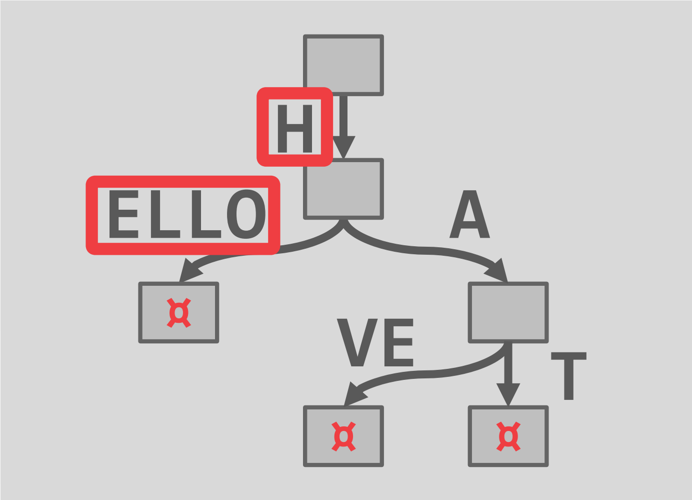
Keys: **HELLO, HAT, HAVE**

TRIE VS. RADIX TREE

Trie

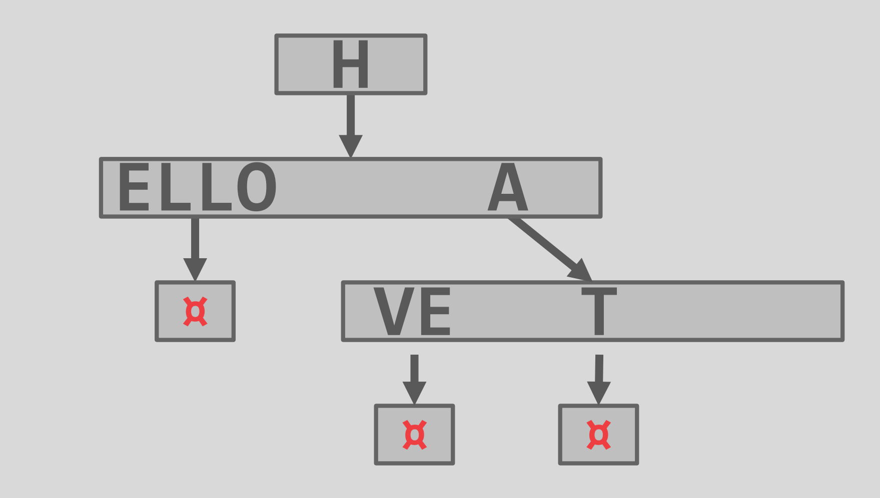


Radix Tree

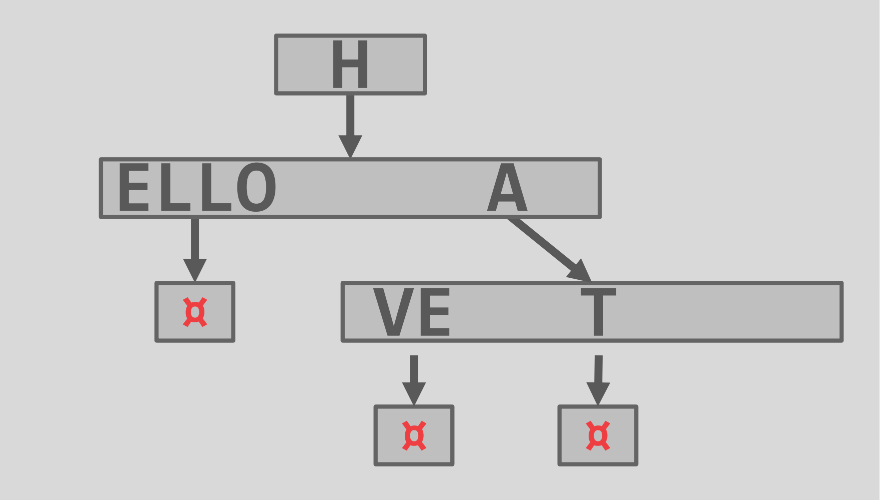


Keys: HELLO HAT, HAVE

RADIX TREE: MODIFICATIONS

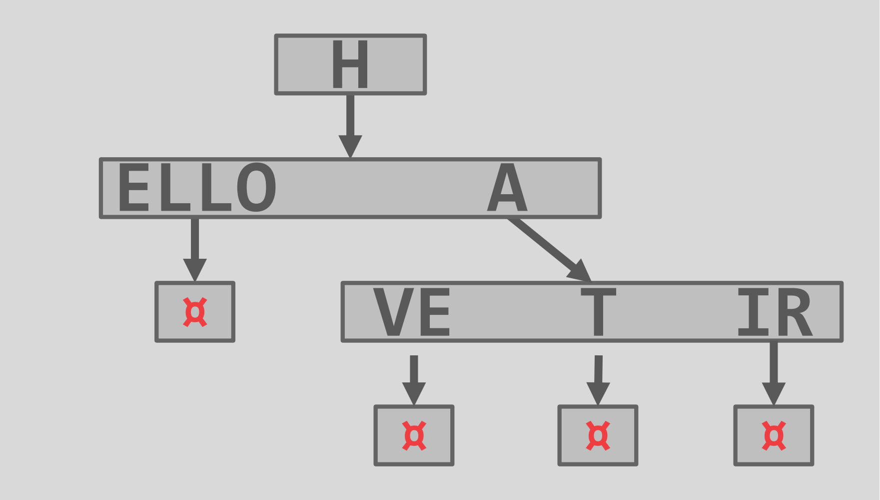


RADIX TREE: MODIFICATIONS



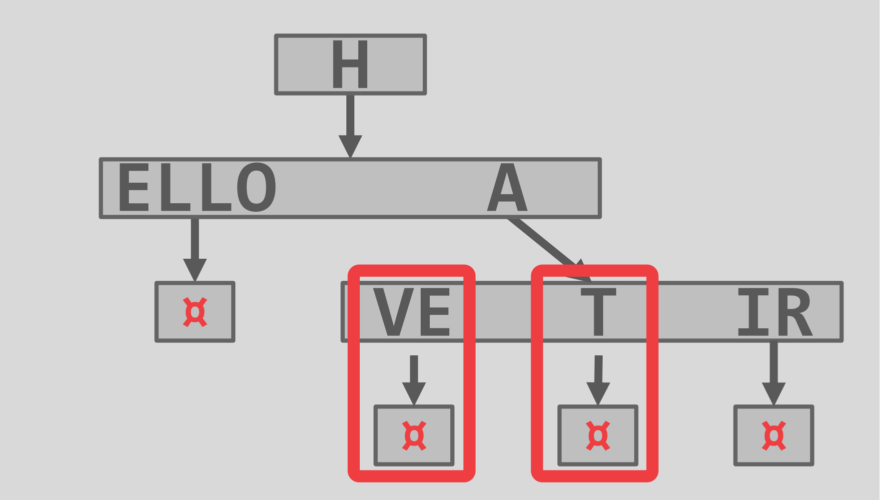
Insert **HAIR**

RADIX TREE: MODIFICATIONS



Insert **HAIR**

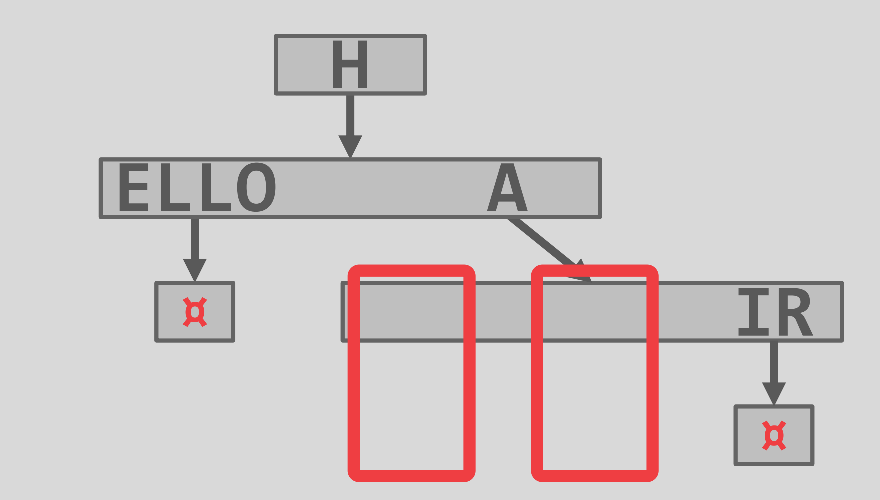
RADIX TREE: MODIFICATIONS



Insert **HAIR**

Delete **HAT**, **HAVE**

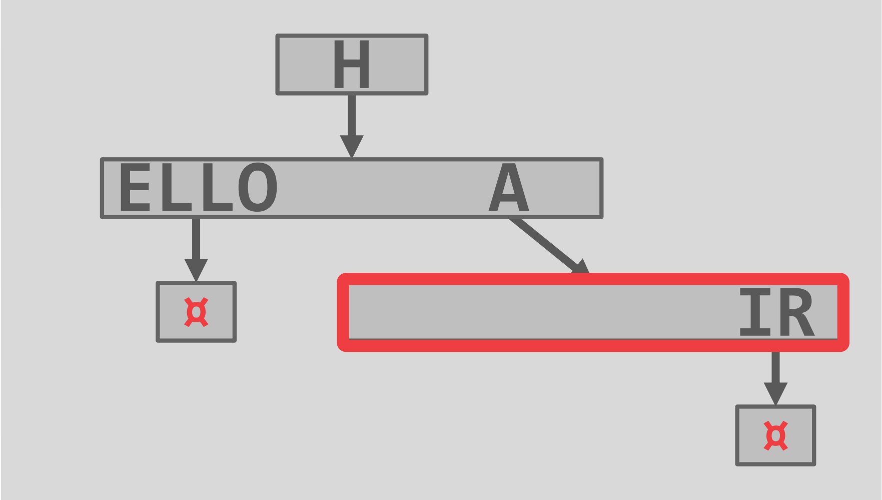
RADIX TREE: MODIFICATIONS



Insert **HAIR**

Delete **HAT**, **HAVE**

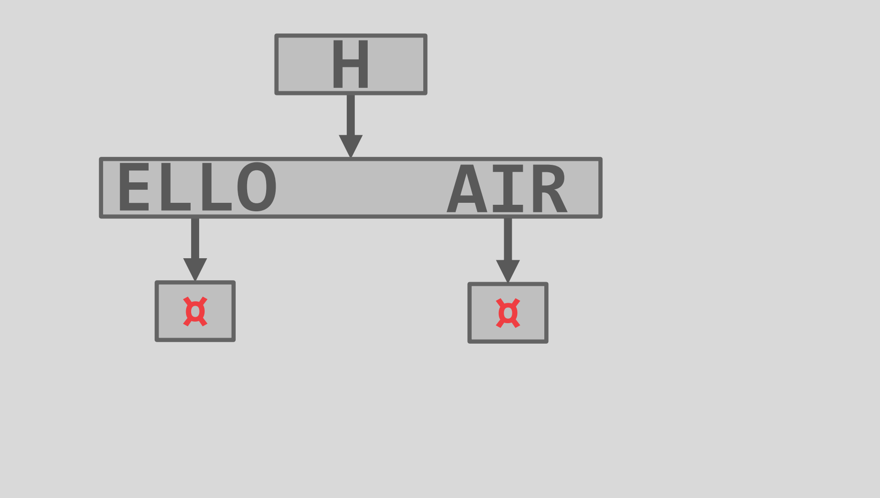
RADIX TREE: MODIFICATIONS



Insert **HAIR**

Delete **HAT**, **HAVE**

RADIX TREE: MODIFICATIONS



Insert **HAIR**

Delete **HAT**, **HAVE**

RADIX TREE: 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.

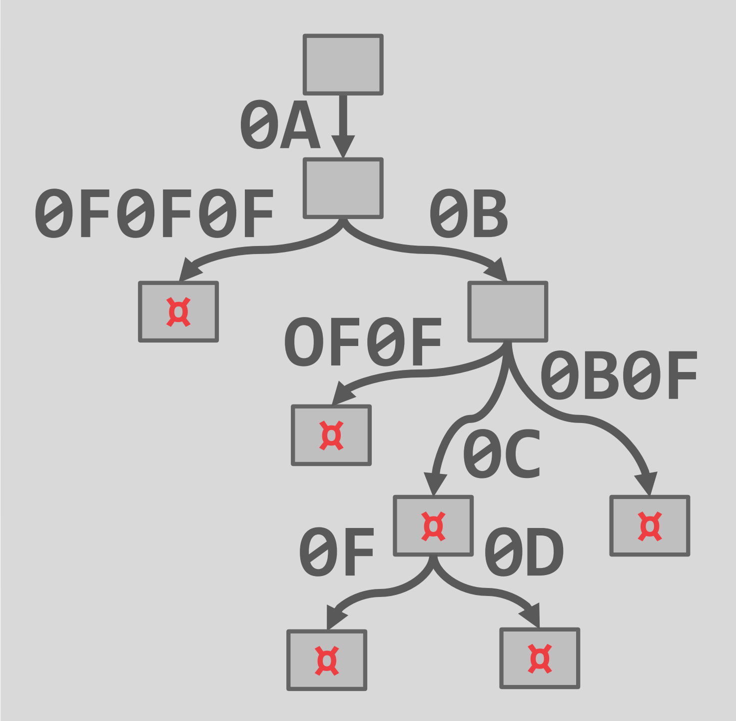
RADIX TREE: BINARY COMPARABLE KEYS

Int Key: **168496141**

↓

Hex Key: **0A 0B 0C 0D**

RADIX TREE: BINARY COMPARABLE KEYS



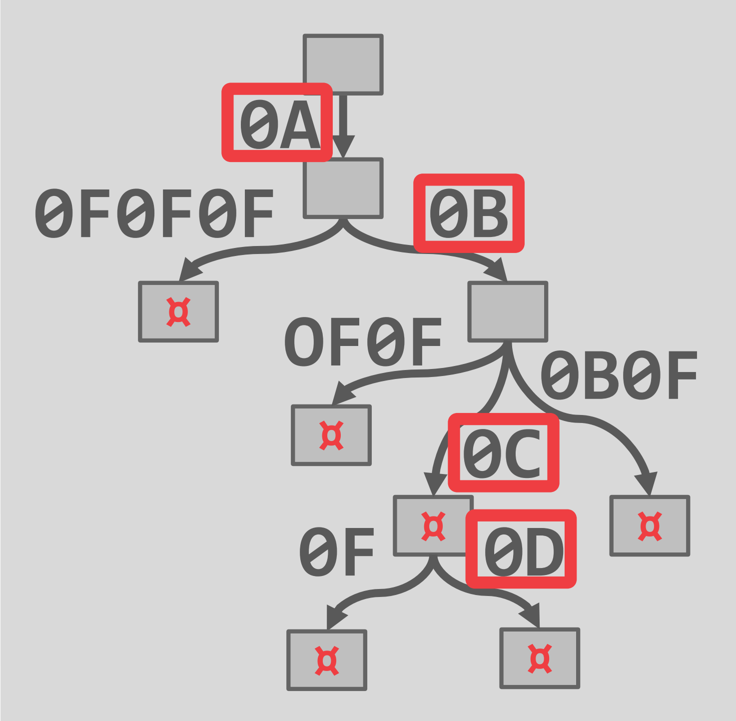
Int Key: **168496141**

Hex Key: **0A 0B 0C 0D**

Big Endian

Little Endian

RADIX TREE: BINARY COMPARABLE KEYS

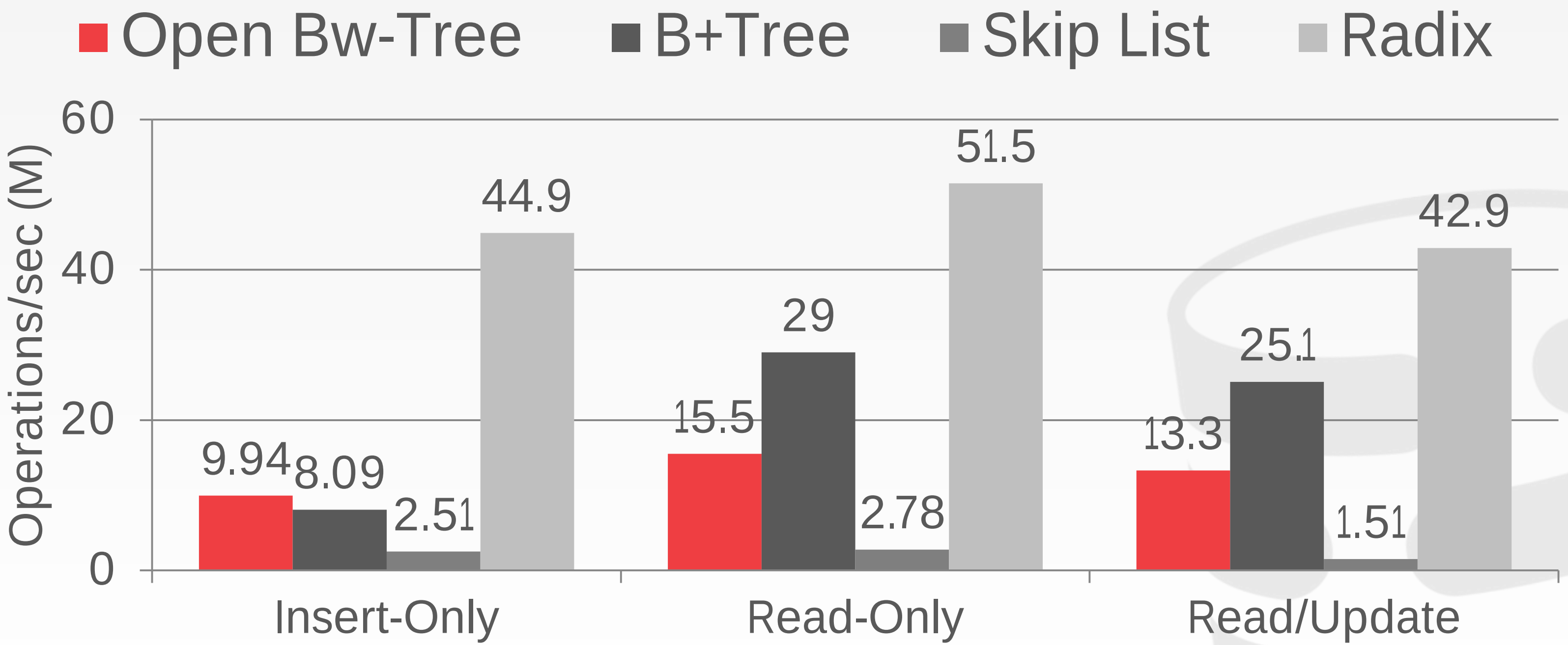


Int Key: **168496141**

Hex Key: **0A 0B 0C 0D**

IN-MEMORY TABLE INDEXES

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



Source: Ziqi Wang

OBSERVATION

The tree indexes that we've discussed so far are useful for "point" and "range" queries:

→ Find all customers in the 15217 zip code.

→ Find all orders between June 2018 and September 2018.

They are **not** good at keyword searches:

→ Find all Wikipedia articles that contain the word "Pavlo"

WIKIPEDIA EXAMPLE

```
CREATE TABLE useracct (
  userID INT PRIMARY KEY,
  userName VARCHAR UNIQUE,
  ⋮
);
```

```
CREATE TABLE pages (
  pageID INT PRIMARY KEY,
  title VARCHAR UNIQUE,
  latest INT
  ↳REFERENCES revisions (revID),
);
```

```
CREATE TABLE revisions (
  revID INT PRIMARY KEY,
  userID INT REFERENCES useracct (userID),
  pageID INT REFERENCES pages (pageID),
  content TEXT,
  updated DATETIME
);
```

WIKIPEDIA EXAMPLE

If we create an index on the content attribute, what does that actually do?

```
CREATE INDEX idx_rev_cntnt
    ON revisions (content);
```

This doesn't help our query.

Our SQL is also not correct...

```
SELECT pageID FROM revisions
 WHERE content LIKE '%Pavlo%';
```

INVERTED INDEX

An ***inverted index*** stores a mapping of words to records that contain those words in the target attribute.

→ Sometimes called a ***full-text search index***.

→ Also called a ***concordance*** in old (like really old) times.

The major DBMSs support these natively.
There are also specialized DBMSs.

elasticsearch

Solr

Sphinx

QUERY TYPES

Phrase Searches

→ Find records that contain a list of words in the given order.

Proximity Searches

→ Find records where two words occur within ***n*** words of each other.

Wildcard Searches

→ Find records that contain words that match some pattern (e.g., regular expression).

DESIGN DECISIONS

Decision #1: What To Store

→ The index needs to store at least the words contained in each record (separated by punctuation characters).
→ Can also store frequency, position, and other meta-data.

Decision #2: When To Update

→ Maintain auxiliary data structures to "stage" updates and then update the index in batches.

CONCLUSION

B+Trees are still the way to go for tree indexes.

Inverted indexes are covered in CMU 11-442.

We did not discuss geo-spatial tree indexes:

→ Examples: R-Tree, Quad-Tree, KD-Tree
→ This is covered in CMU 15-826.

NEXT CLASS

How to make indexes thread-safe!