Carnegie Mellon University

Database Systems

Bloom Filters, Tries, Skip Lists, Inverted Indexes, Vector Indexes

15-445/645 FALL 2024 » PROF. ANDY PAVLO

ADMINISTRIVIA

Project #1 is due Sunday Sept 29th @ 11:59pm

→ Recitation + Profiling Tutorial: **@160**

→ **Extra Office Hours** on Saturday Sept 28th @ 3:00-5:00pm

→ Location: GHC 5207

→ https://ohq.eberly.cmu.edu

Homework #3 is due Sunday Sept 6th @ 11:59pm

Mid-term Exam on Wednesday Oct 9th @ 2:00pm

→ In-class in this room.

→ More info next week.

INDEXES VS. FILTERS

An **index** data structure of a subset of a table's attributes that are organized and/or sorted to the location of specific tuples using those attributes.

→ Example: B+Tree

A **filter** is a data structure that answers set membership queries; it tells you whether a key (likely) exists in a set but not where it is located.

→ Example: Bloom Filter

TODAY'S AGENDA

Bloom Filters

Skip Lists

Tries / Radix Trees

Inverted Indexes

Vector Indexes

DB Flash Talk: TiDB

BLOOM FILTERS

Probabilistic data structure (bitmap) that answers set membership queries.

→ False negatives will never occur.

→ False positives can sometimes occur.

→ See Bloom Filter Calculator.

Insert(x):

→ Use *k* hash functions to set bits in the filter to 1.

Lookup(x):

→ Check whether the bits are 1 for each hash function.

BLOOM FILTERS

Insert 'RZA'

Bloom Filter

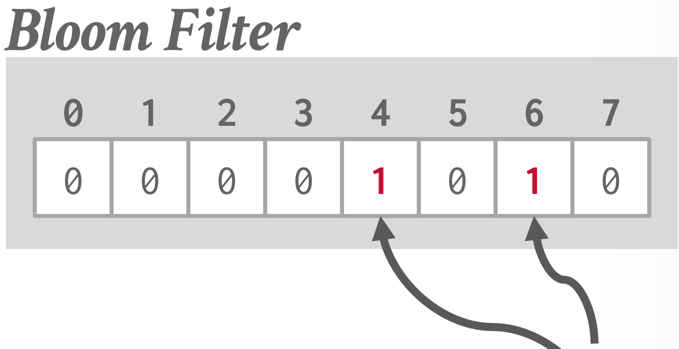
0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0

$hash_1('RZA') = 2222 \% 8 = 6$

$hash_2('RZA') = 4444 \% 8 = 4$

BLOOM FILTERS

Insert 'RZA'



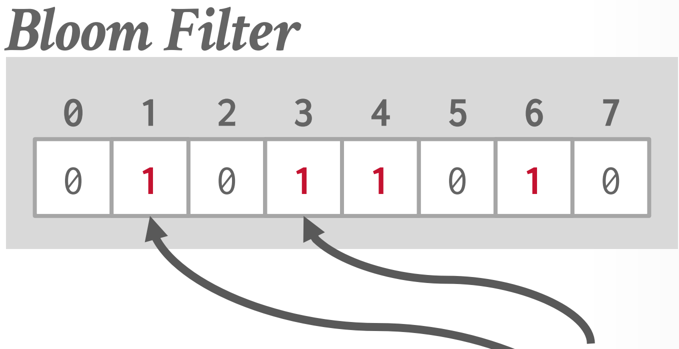
$$hash_1(\text{'RZA'}) = 2222 \% 8 = 6$$

$$hash_2(\text{'RZA'}) = 4444 \% 8 = 4$$

BLOOM FILTERS

Insert 'RZA'

Insert 'GZA'



$hash_1('GZA') = 5555 \% 8 = 3$

$hash_2('GZA') = 7777 \% 8 = 1$

BLOOM FILTERS

Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → *TRUE*

Bloom Filter

0	1	2	3	4	5	6	7
0	1	0	1	1	0	1	0

$hash_1(\text{'RZA'}) = 2222 \% 8 = 6$

$hash_2(\text{'RZA'}) = 4444 \% 8 = 4$

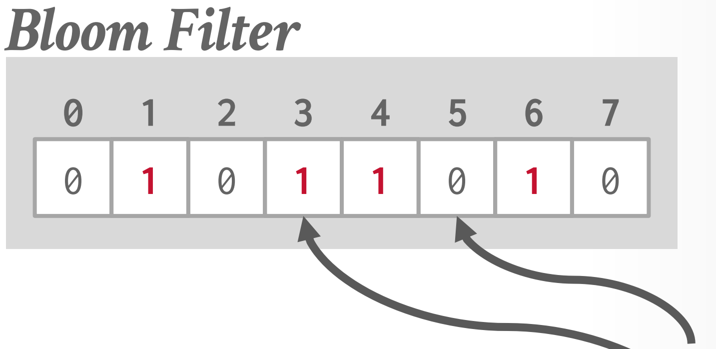
BLOOM FILTERS

Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → *TRUE*

Lookup 'Raekwon' → *FALSE*



$hash_1$('Raekwon') = 3333 % 8 = 5

$hash_2$('Raekwon') = 8899 % 8 = 3

BLOOM FILTERS

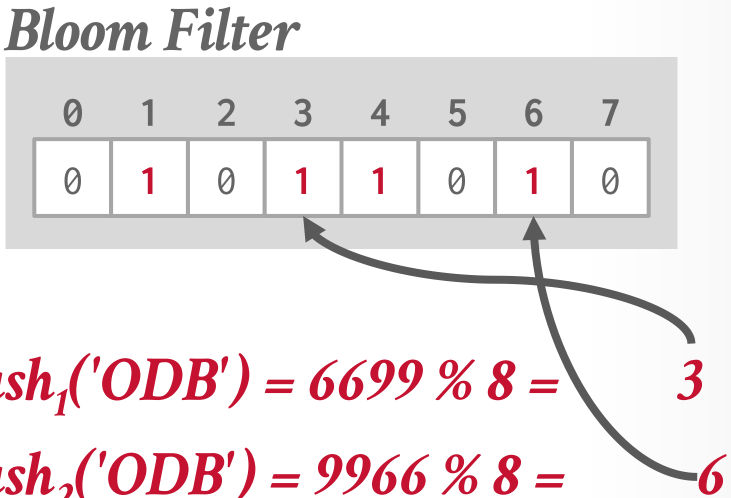
Insert 'RZA'

Insert 'GZA'

Lookup 'RZA' → *TRUE*

Lookup 'Raekwon' → *FALSE*

Lookup 'ODB' → ***TRUE***



$hash_1('ODB') = 6699 \% 8 = 3$

$hash_2('ODB') = 9966 \% 8 = 6$

OTHER FILTERS

Counting Bloom Filter

→ Supports dynamically adding and removing keys.
→ Uses integers instead of bits to count the number of occurrences of a key in a set.

Carnegie Mellon University →

Cuckoo Filter

→ Also supports dynamically adding and removing keys.
→ Uses a Cuckoo Hash Table but stores fingerprints instead of full keys.

Carnegie Mellon University →

Succinct Range Filter (SuRF)

→ Immutable compact trie that supports approximate exact matches and range filtering.

OTHER F

Counting Bloom Filter

→ Supports dynamically adding

→ Uses integers instead of bits t occurrences of a key in a set.

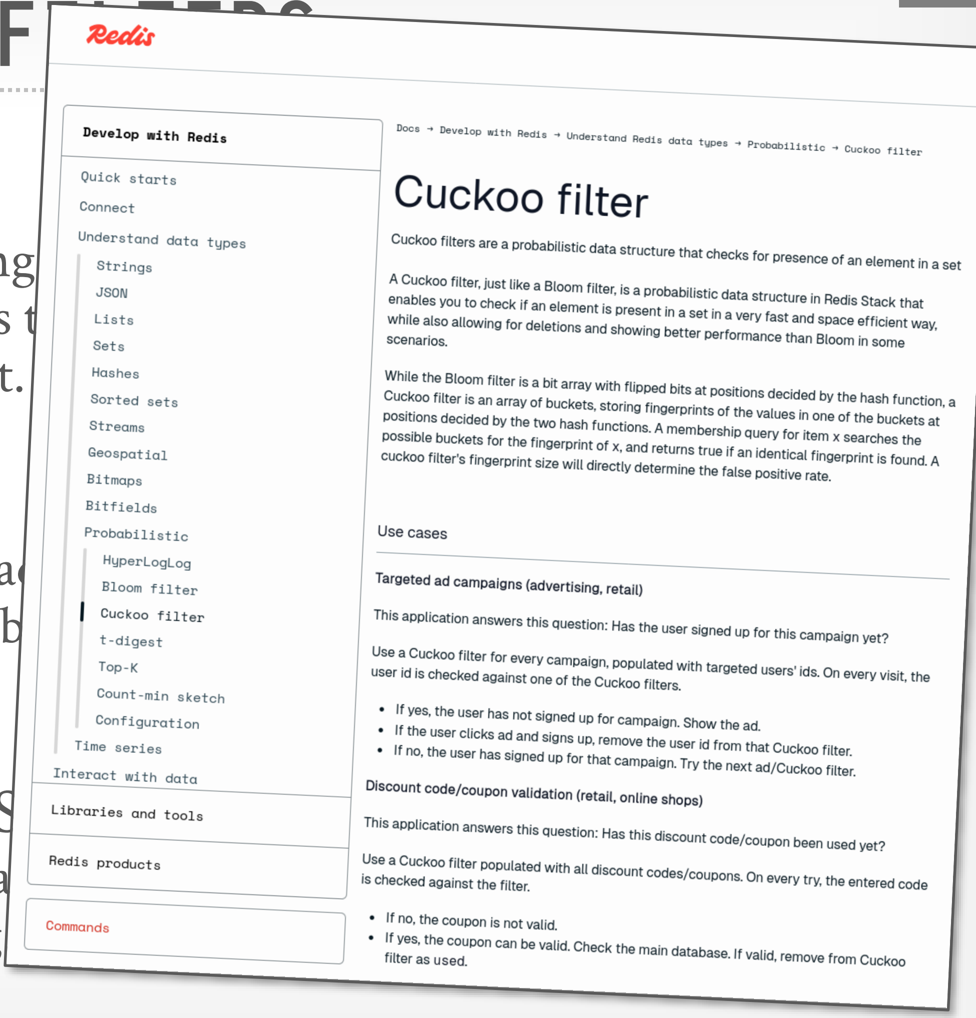
Cuckoo Filter

→ Also supports dynamically a

→ Uses a Cuckoo Hash Table b of full keys.

Succinct Range Filter (S

→ Immutable compact trie tha matches and range filtering



Redis

Develop with Redis

Quick starts
Connect
Understand data types
Strings
JSON
Lists
Sets
Hashes
Sorted sets
Streams
Geospatial
Bitmaps
Bitfields
Probabilistic
HyperLogLog
Bloom filter
Cuckoo filter
t-digest
Top-K
Count-min sketch
Configuration
Time series
Interact with data
Libraries and tools
Redis products
Commands

Docs → Develop with Redis → Understand Redis data types → Probabilistic → Cuckoo filter

Cuckoo filter

Cuckoo filters are a probabilistic data structure that checks for presence of an element in a set

A Cuckoo filter, just like a Bloom filter, is a probabilistic data structure in Redis Stack that enables you to check if an element is present in a set in a very fast and space efficient way, while also allowing for deletions and showing better performance than Bloom in some scenarios.

While the Bloom filter is a bit array with flipped bits at positions decided by the hash function, a Cuckoo filter is an array of buckets, storing fingerprints of the values in one of the buckets at positions decided by the two hash functions. A membership query for item x searches the possible buckets for the fingerprint of x, and returns true if an identical fingerprint is found. A cuckoo filter's fingerprint size will directly determine the false positive rate.

Use cases

Targeted ad campaigns (advertising, retail)

This application answers this question: Has the user signed up for this campaign yet?

Use a Cuckoo filter for every campaign, populated with targeted users' ids. On every visit, the user id is checked against one of the Cuckoo filters.

- If yes, the user has not signed up for campaign. Show the ad.
- If the user clicks ad and signs up, remove the user id from that Cuckoo filter.
- If no, the user has signed up for that campaign. Try the next ad/Cuckoo filter.

Discount code/coupon validation (retail, online shops)

This application answers this question: Has this discount code/coupon been used yet?

Use a Cuckoo filter populated with all discount codes/coupons. On every try, the entered code is checked against the filter.

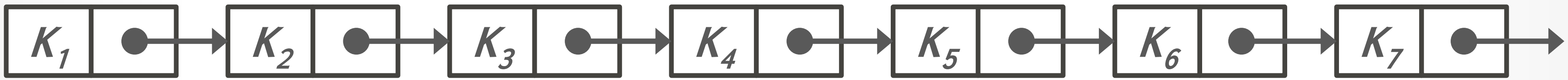
- If no, the coupon is not valid.
- If yes, the coupon can be valid. Check the main database. If valid, remove from Cuckoo filter as used.

OBSERVATION

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

All operations have to linear search.

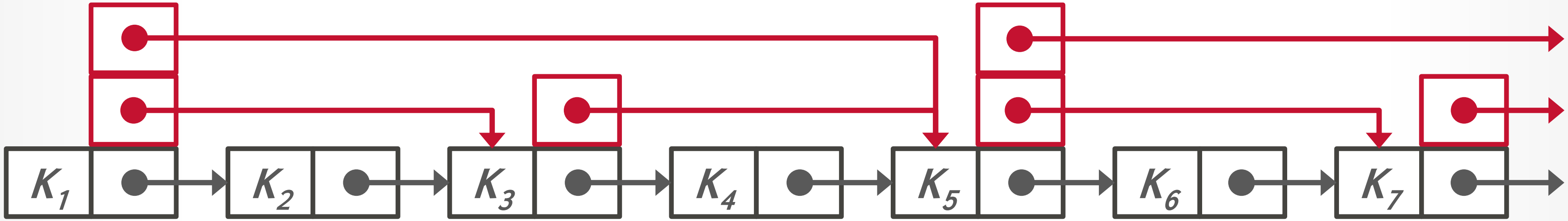
→ Average Cost: **O(n)**



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 to <u>skip</u> over entries.

→ 1st level is a sorted list of all keys.

→ 2nd level links every other key

→ 3rd level links every fourth key

→ Each level has ½ the keys of one below it

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 1e)? 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.

Maintains keys in sorted order without requiring global rebalancing.

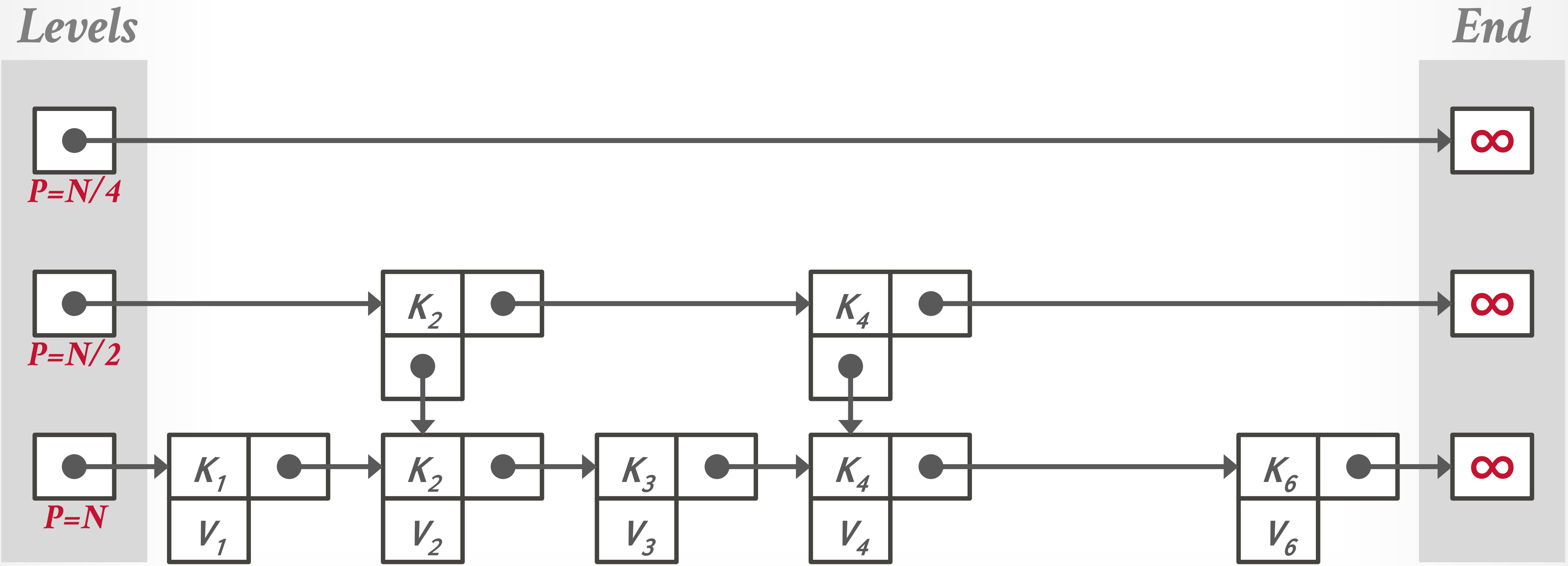
→ Approximate **O(log n)** search times.

Mostly for in-memory data structures.

→ Example: LSM MemTable

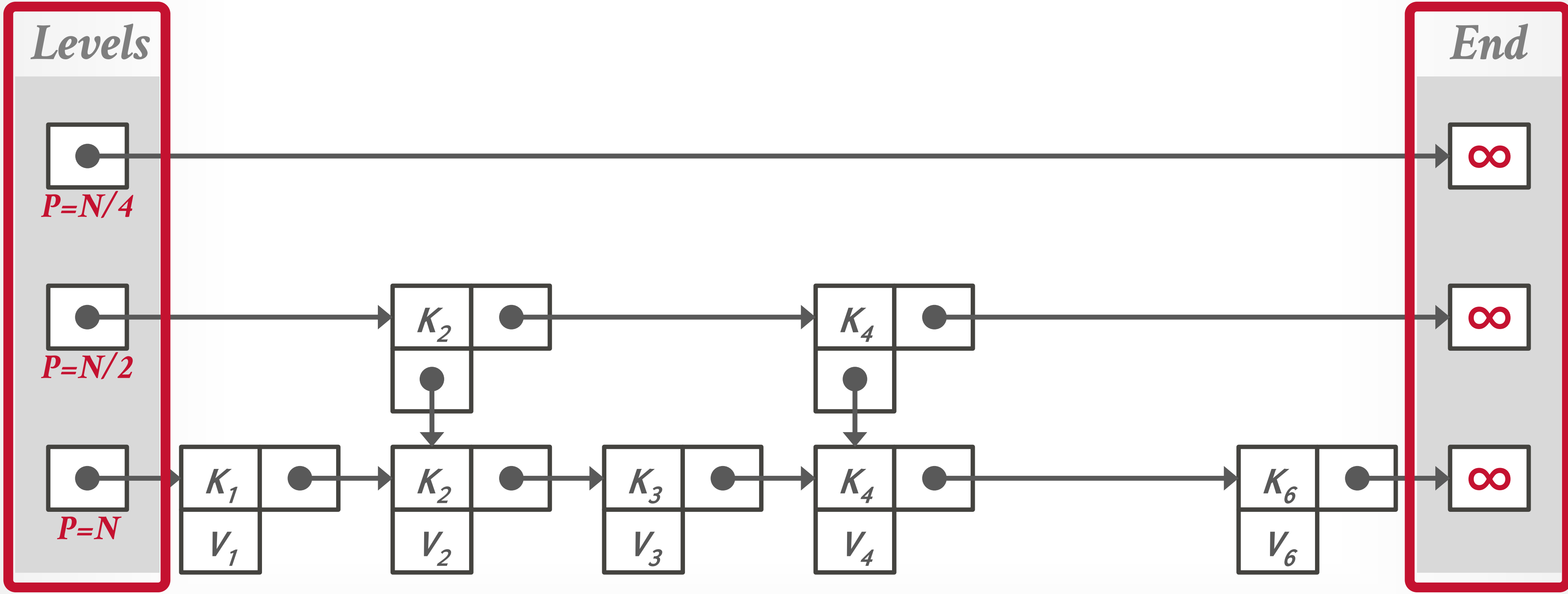
SKIP LISTS: INSERT

Insert K_5



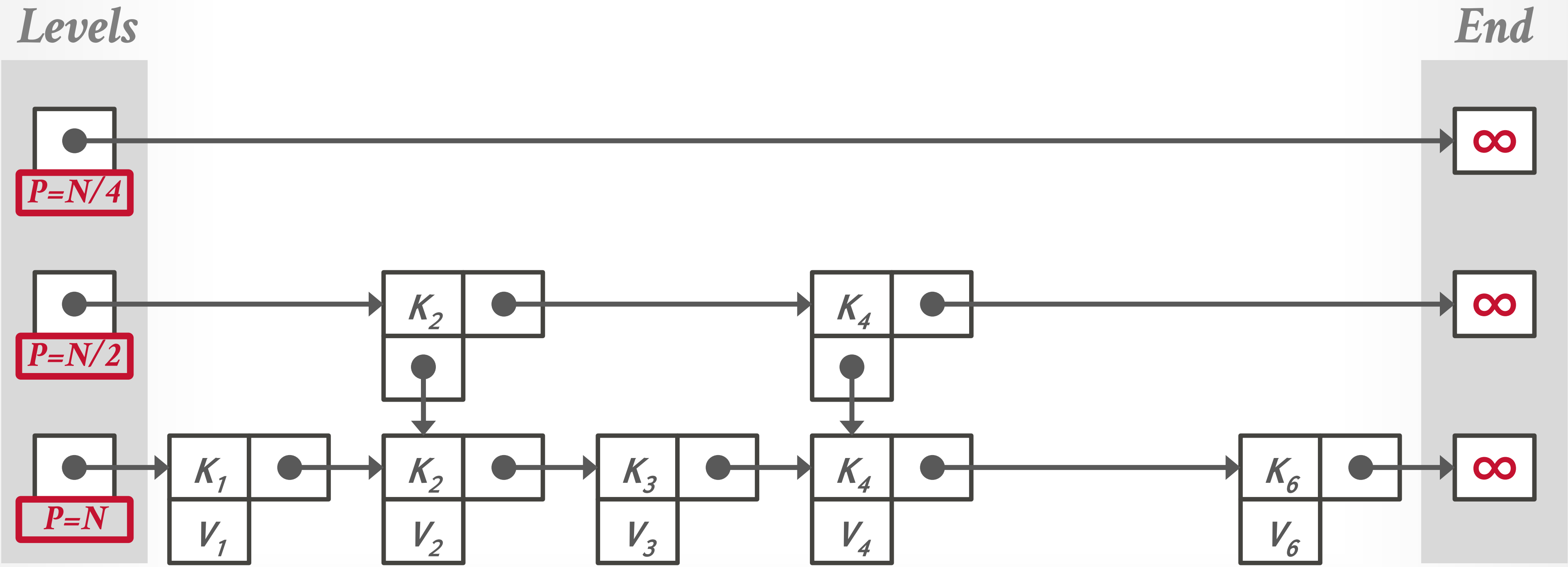
SKIP LISTS: INSERT

Insert K_5



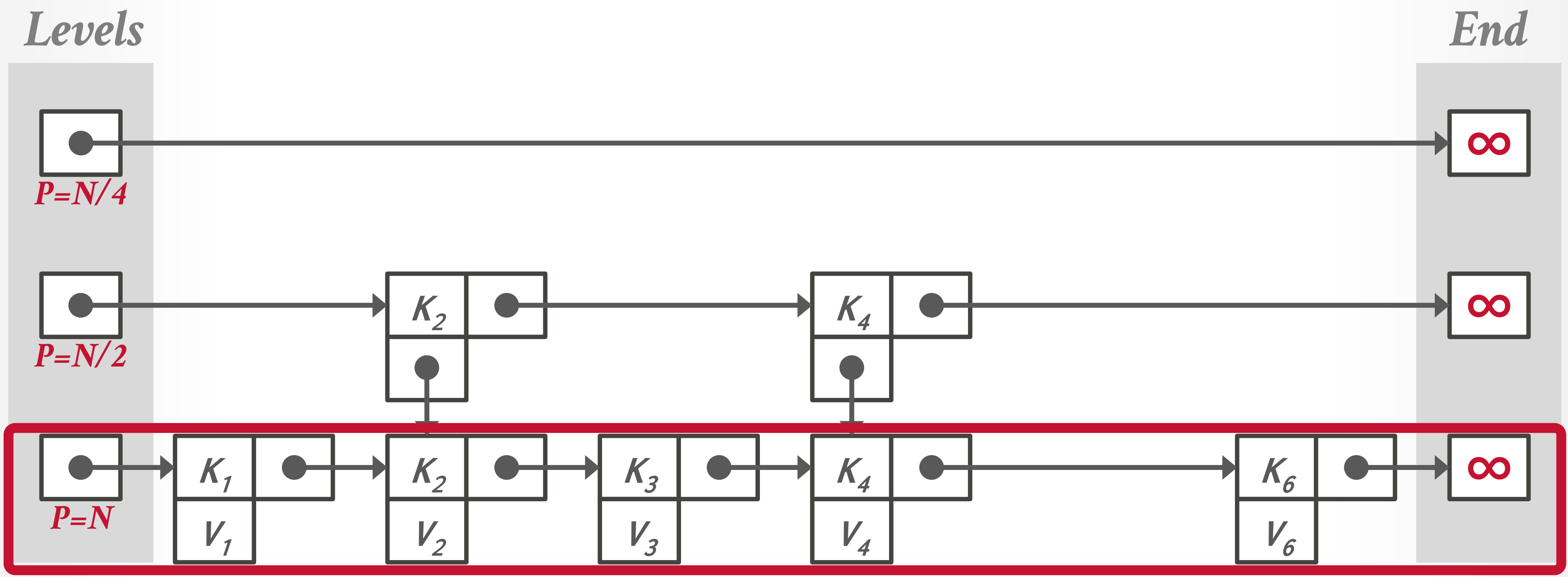
SKIP LISTS: INSERT

Insert K_5



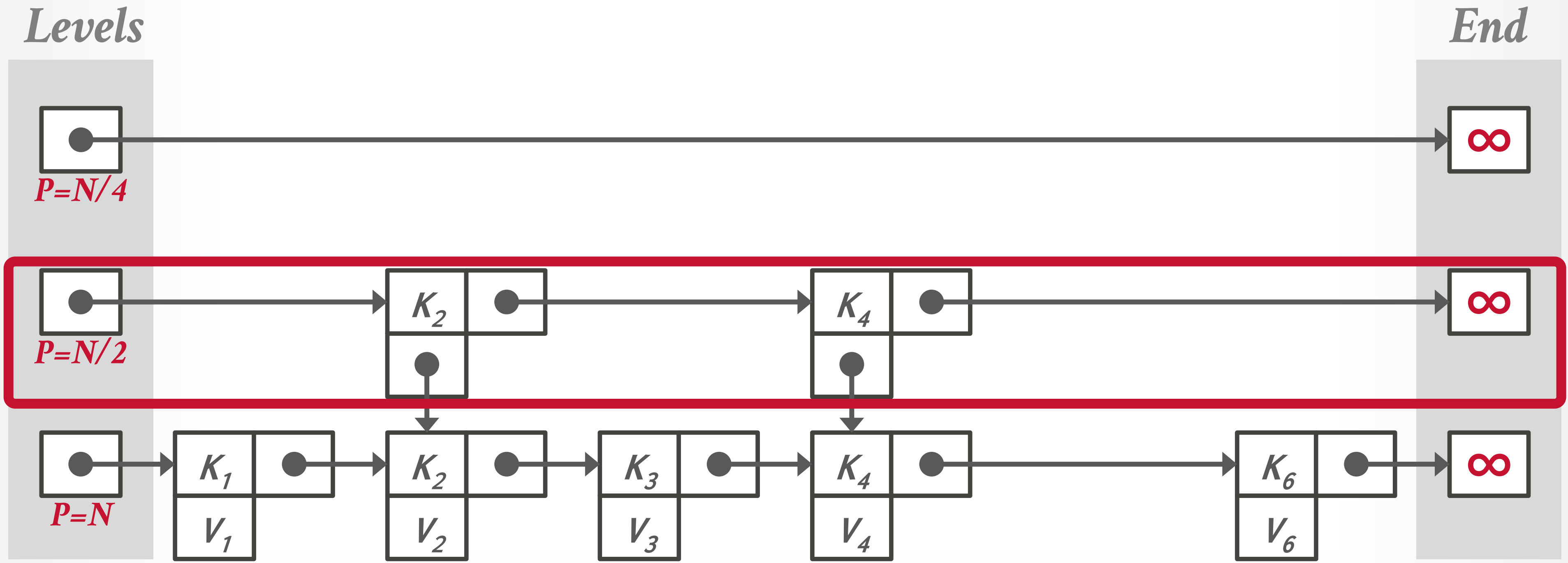
SKIP LISTS: INSERT

Insert K_5



SKIP LISTS: INSERT

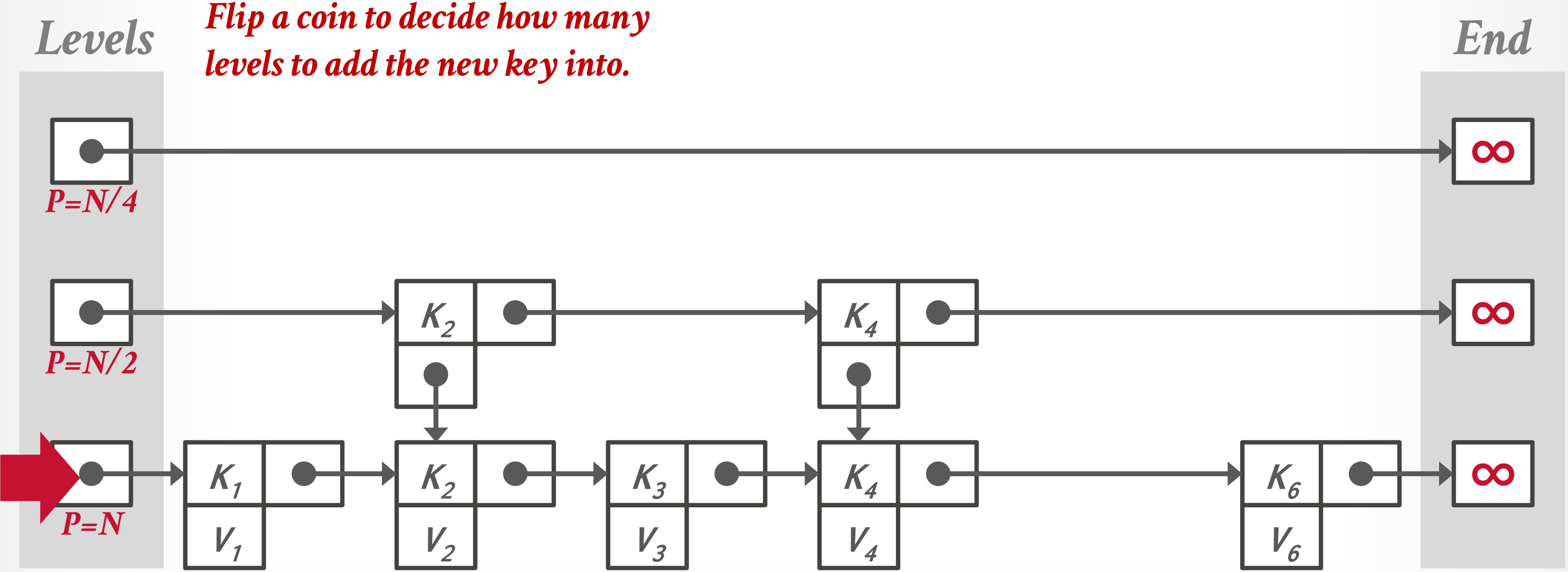
Insert K_5



SKIP LISTS: INSERT

Insert K_5

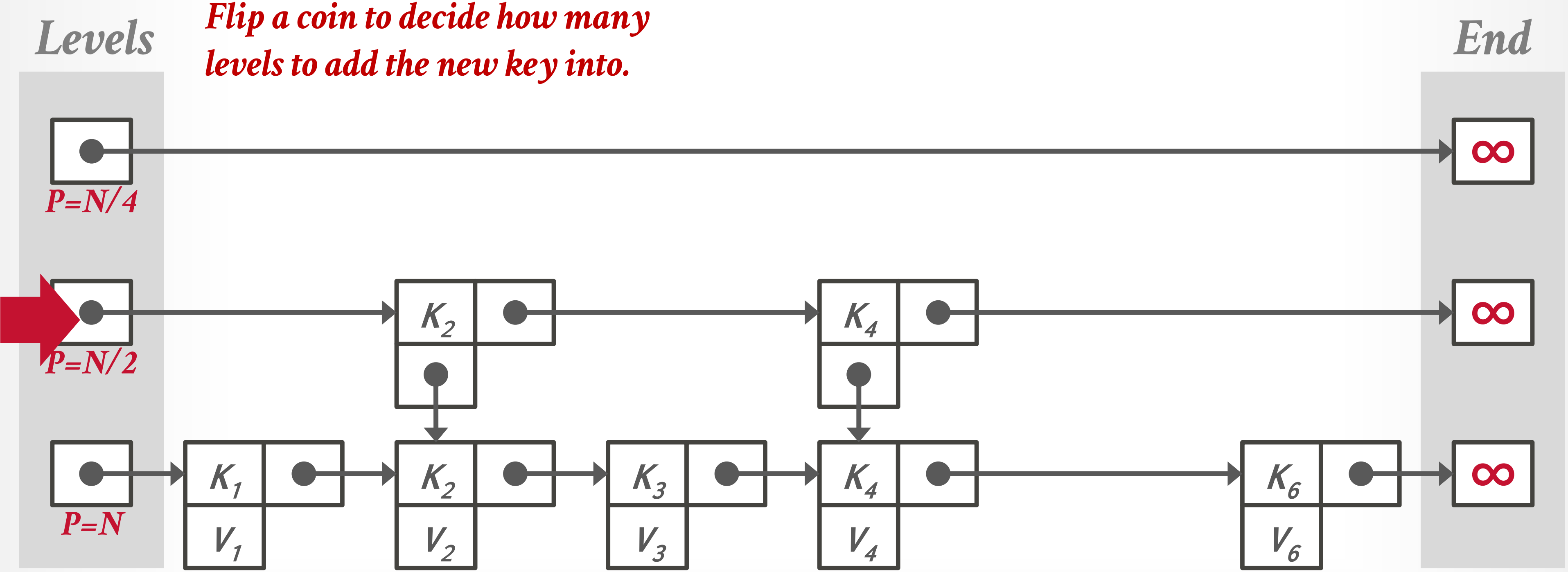
Flip a coin to decide how many levels to add the new key into.



SKIP LISTS: INSERT

Insert K_5

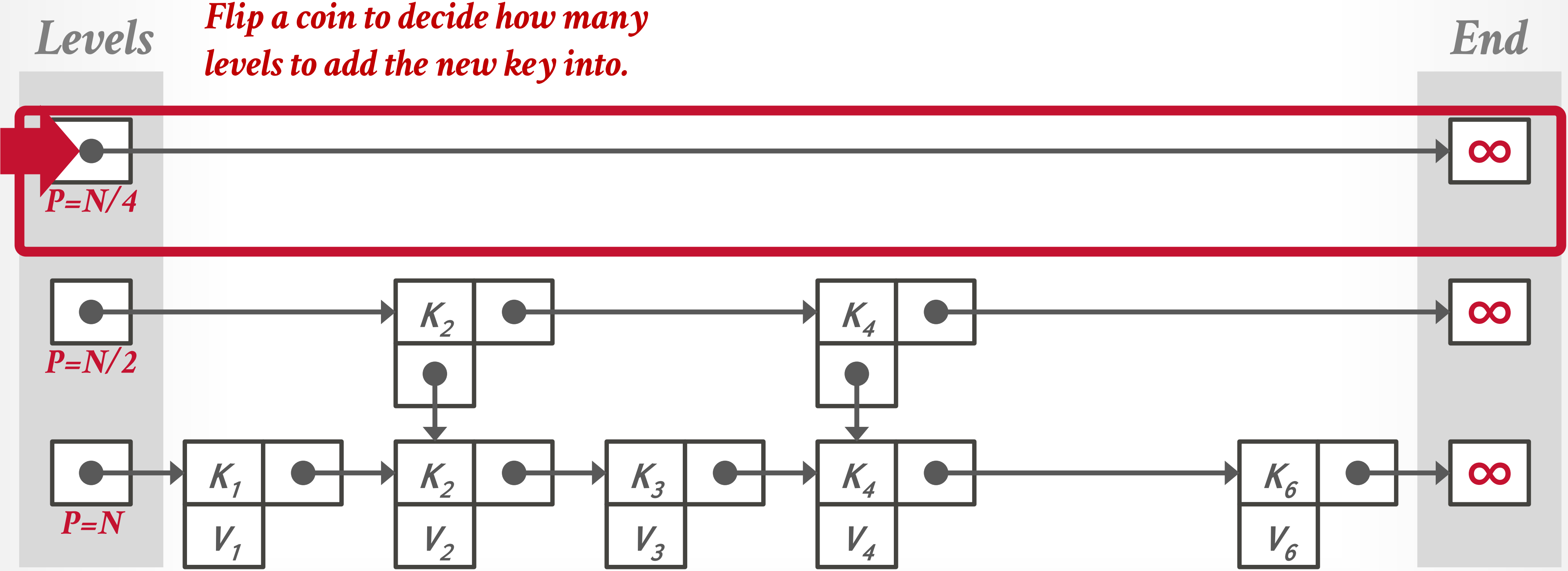
Flip a coin to decide how many levels to add the new key into.



SKIP LISTS: INSERT

Insert K_5

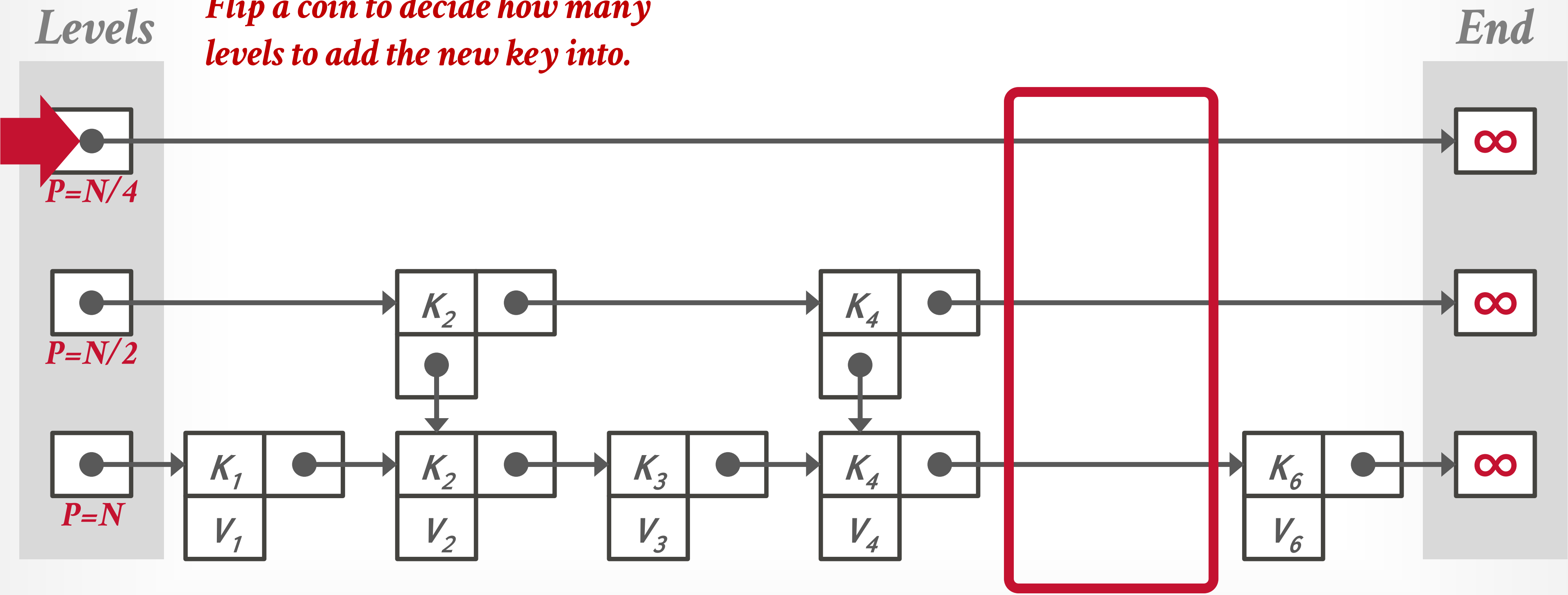
Flip a coin to decide how many levels to add the new key into.



SKIP LISTS: INSERT

Insert K_5

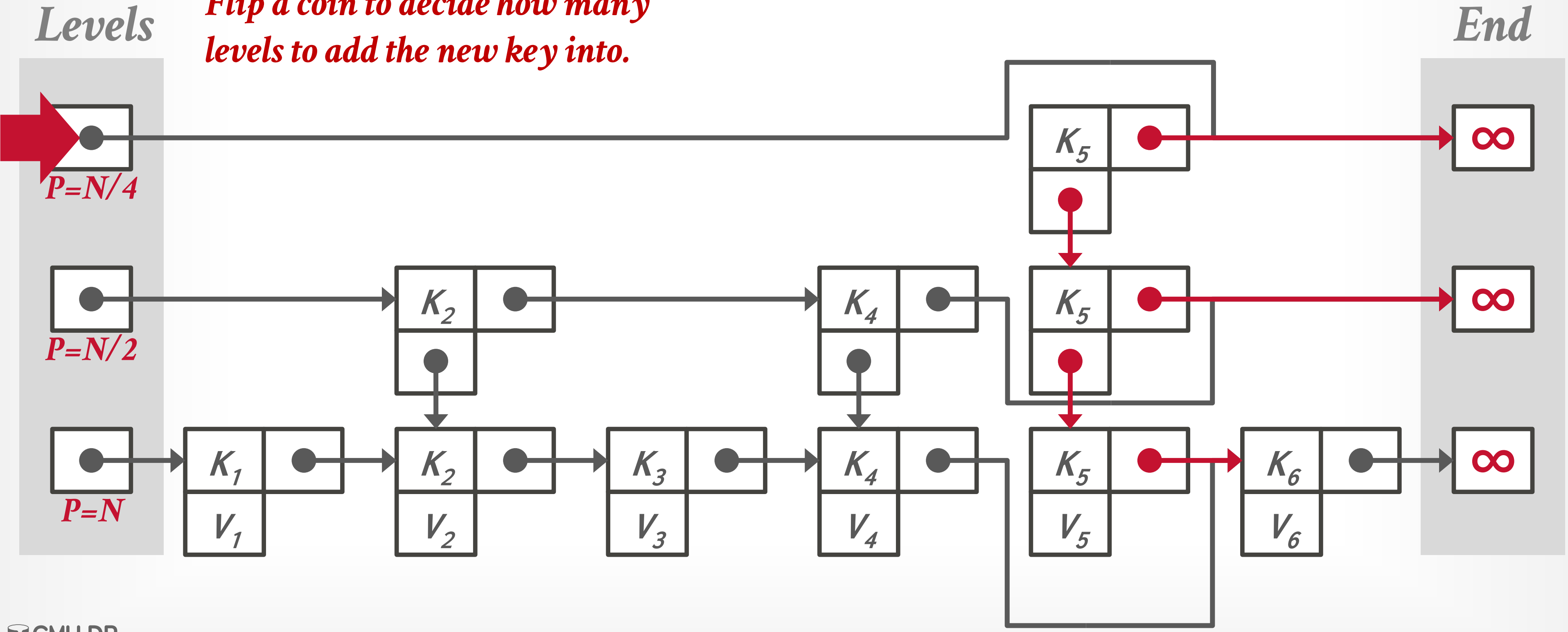
Flip a coin to decide how many levels to add the new key into.



SKIP LISTS: INSERT

Insert K_5

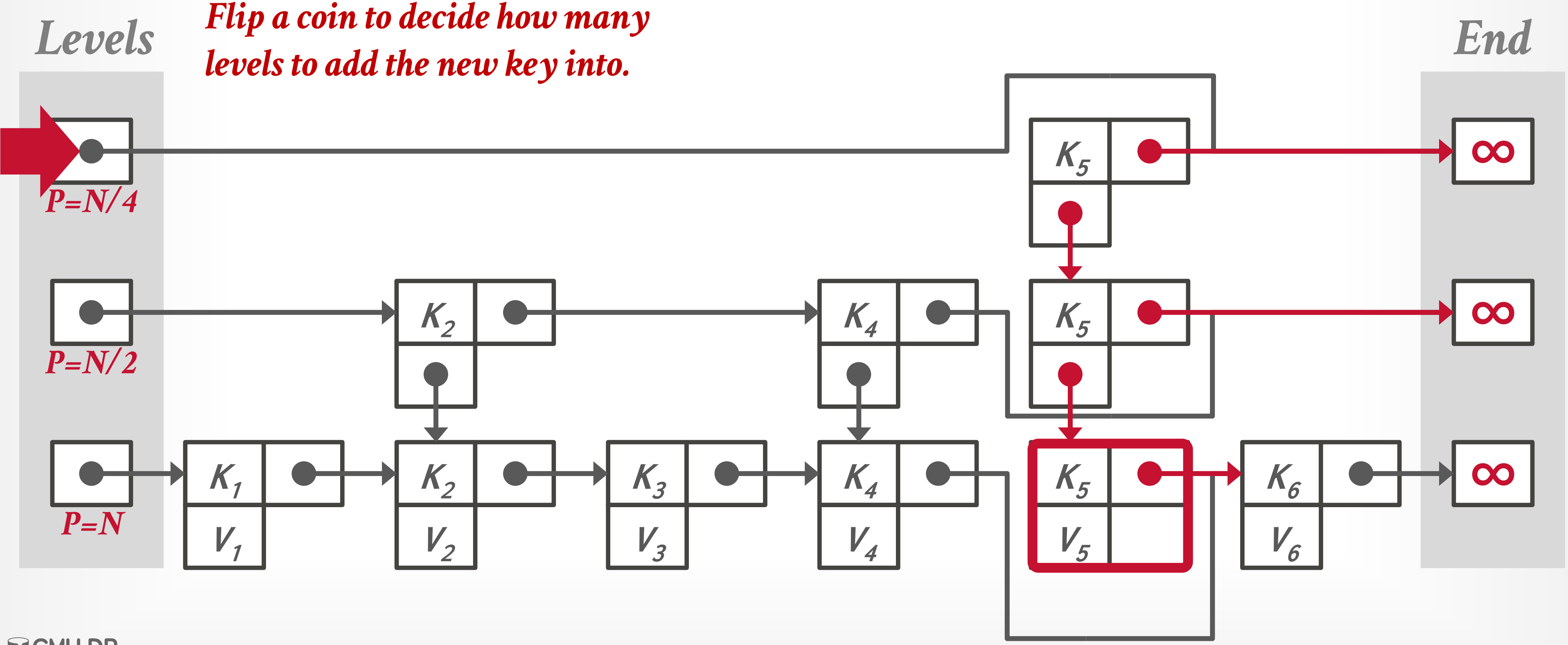
Flip a coin to decide how many levels to add the new key into.



SKIP LISTS: INSERT

Insert K_5

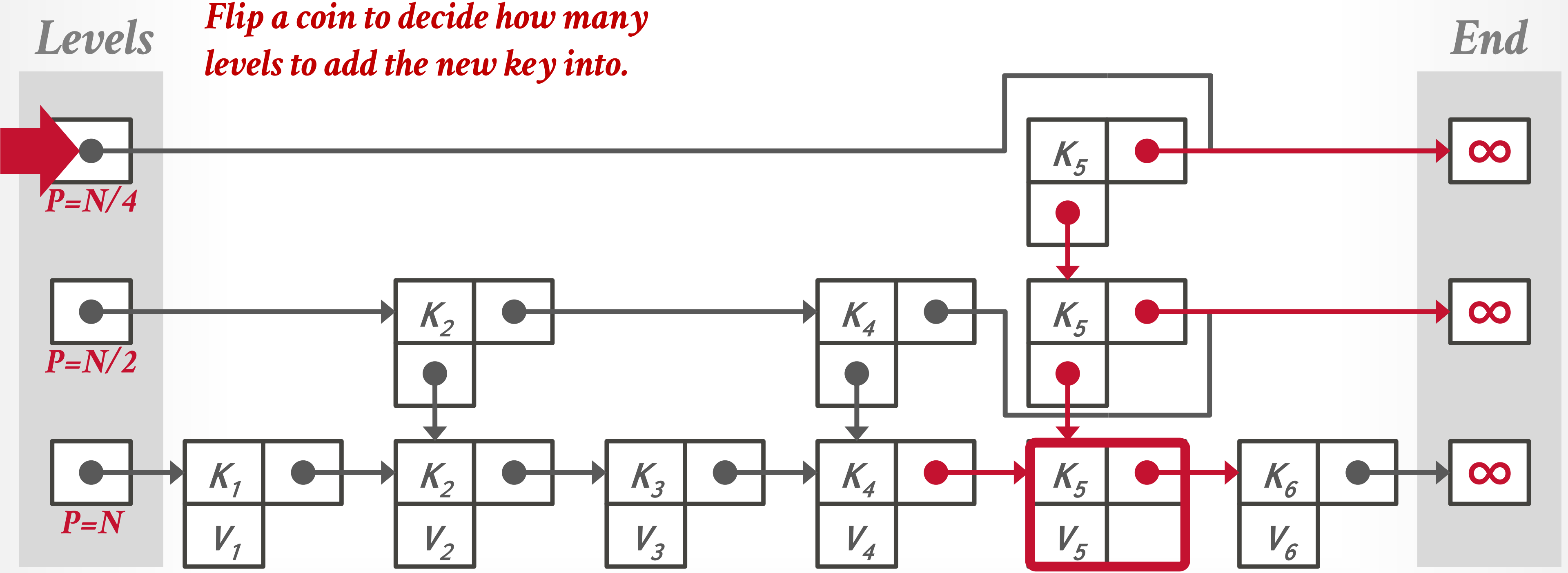
Flip a coin to decide how many levels to add the new key into.



SKIP LISTS: INSERT

Insert K_5

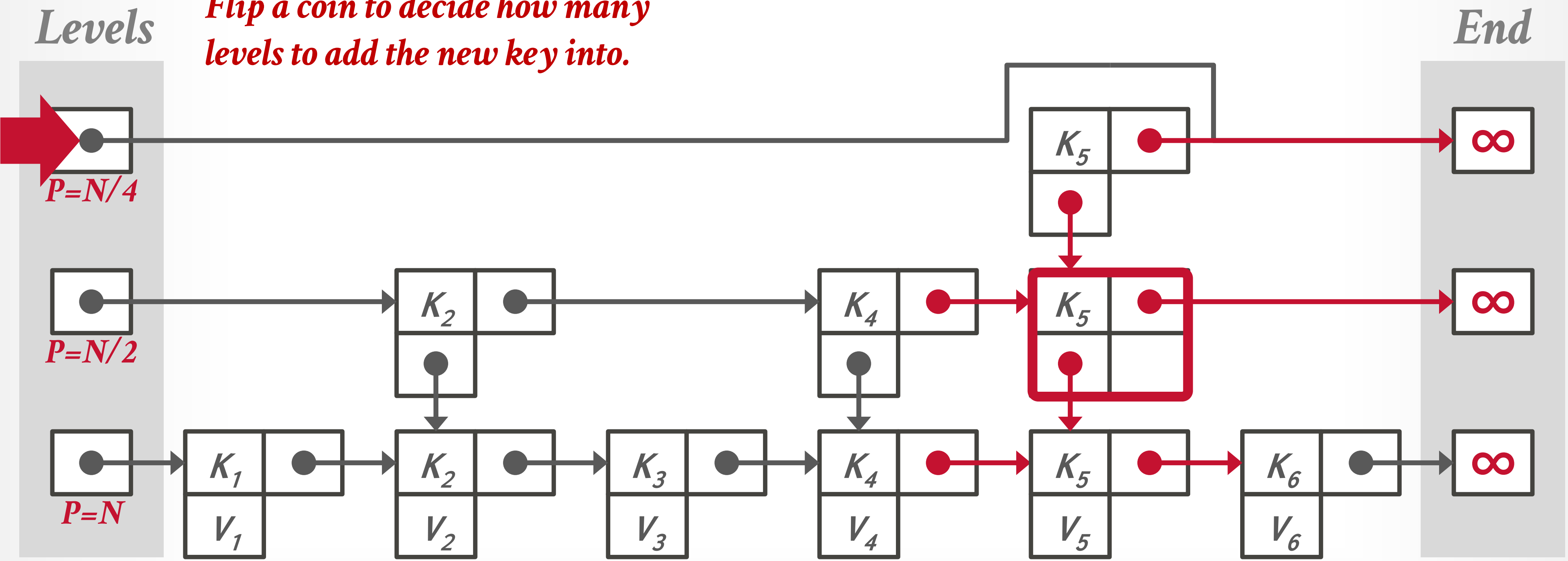
Flip a coin to decide how many levels to add the new key into.



SKIP LISTS: INSERT

Insert K_5

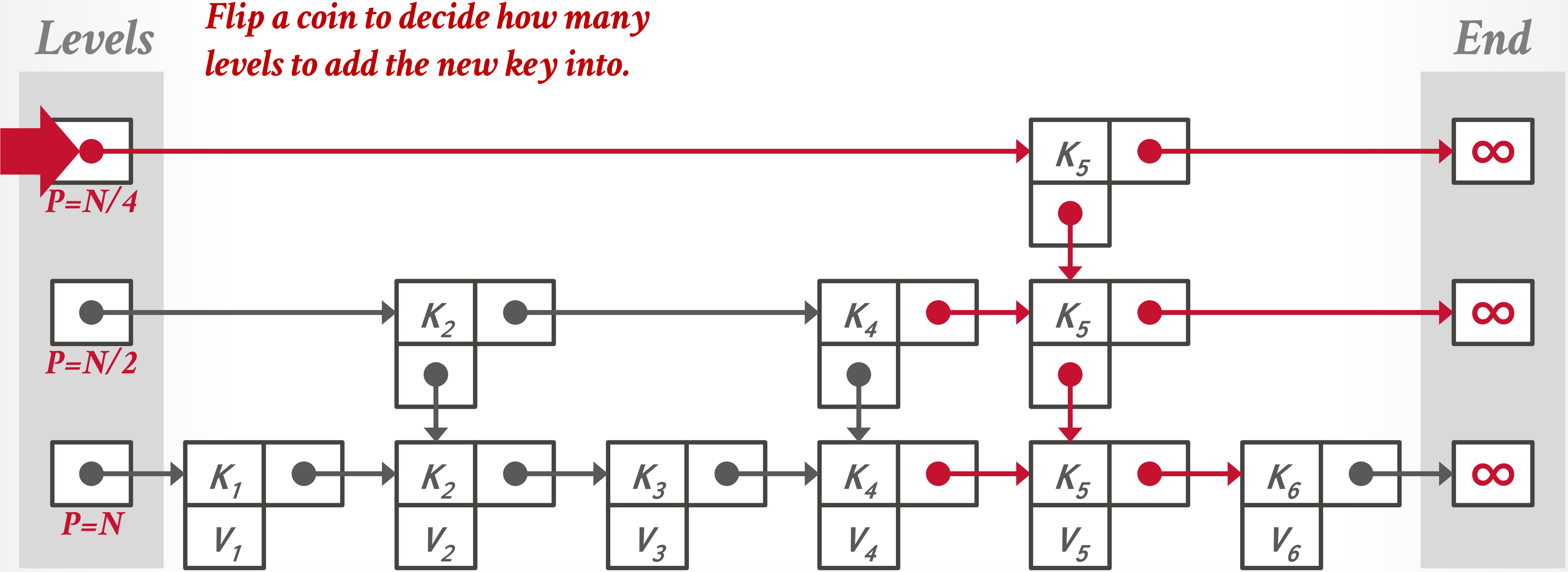
Flip a coin to decide how many levels to add the new key into.



SKIP LISTS: INSERT

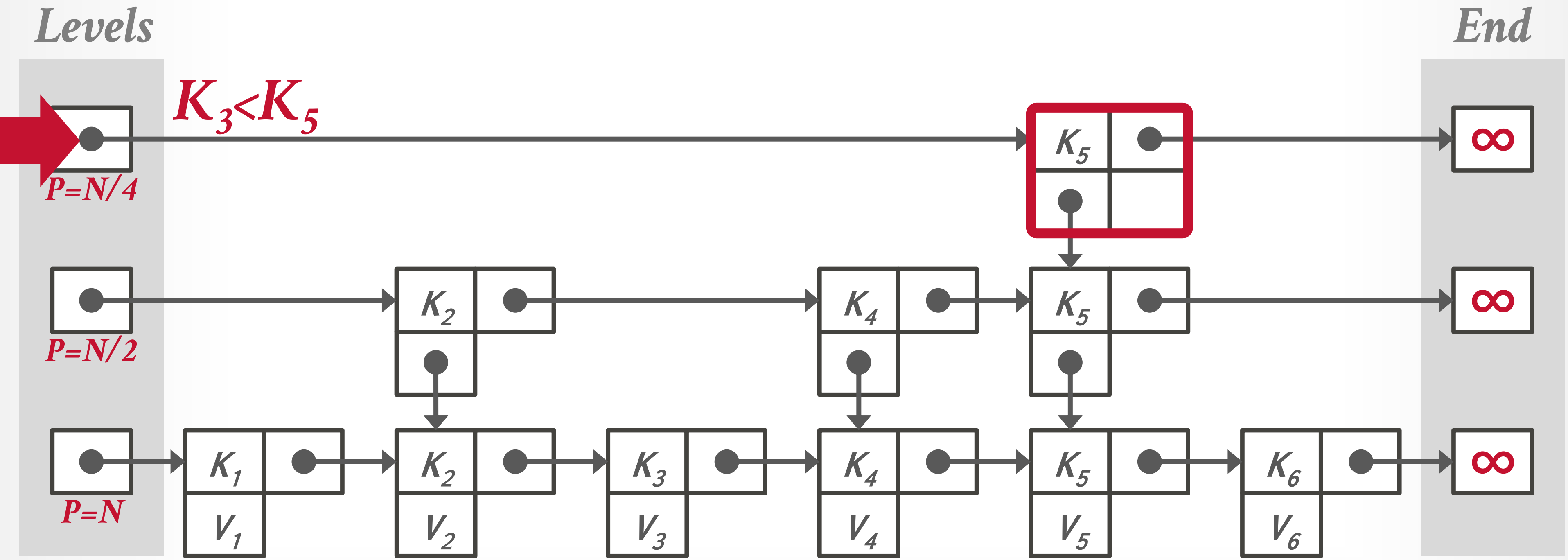
Insert K_5

Flip a coin to decide how many levels to add the new key into.



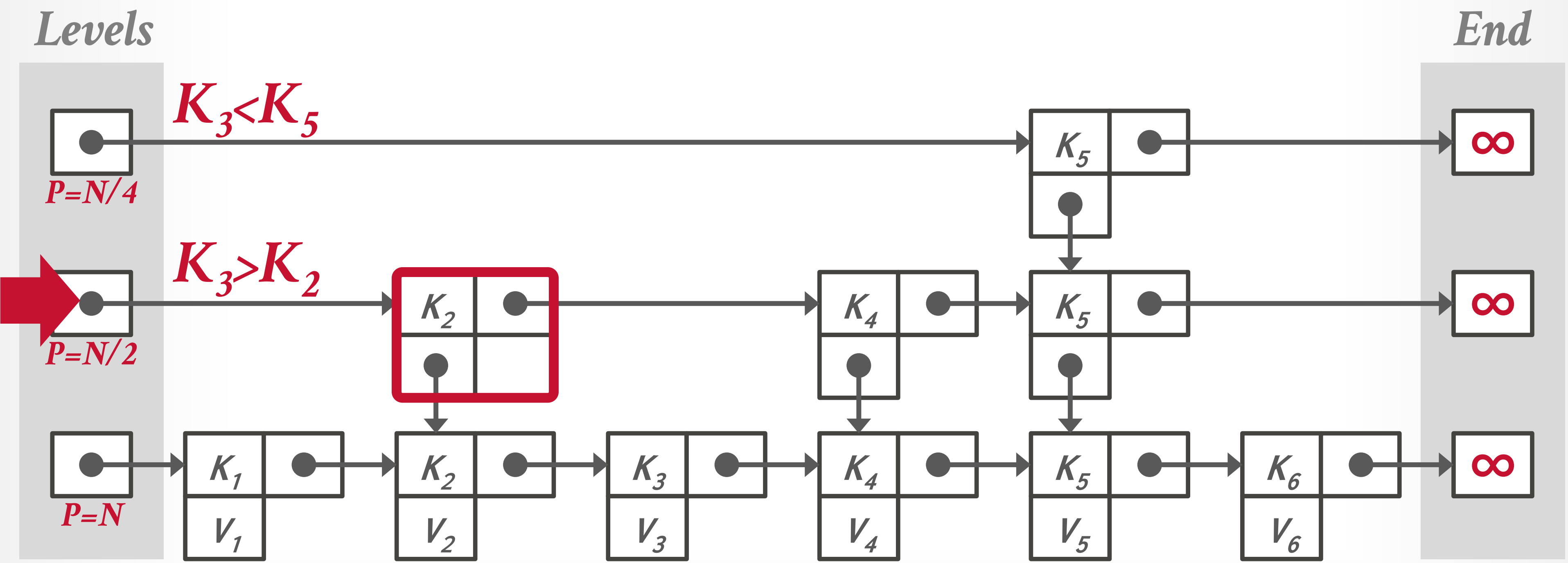
SKIP LISTS: SEARCH

Find K_3



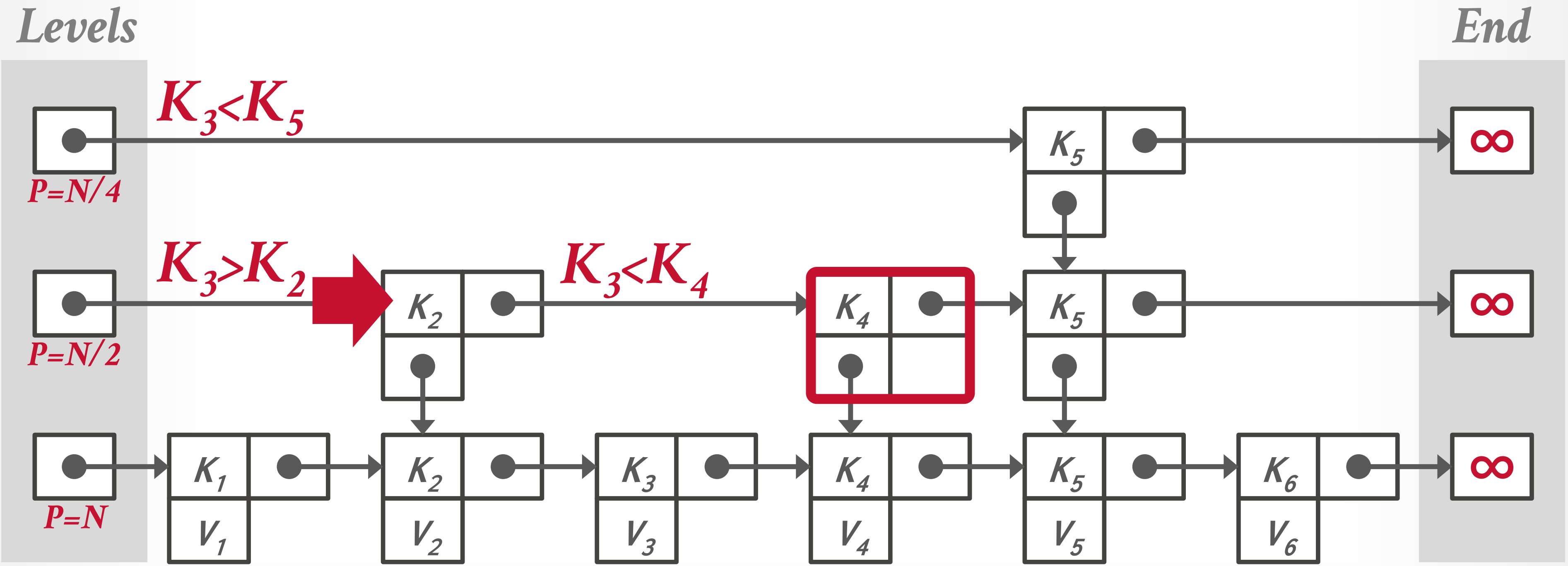
SKIP LISTS: SEARCH

Find K_3



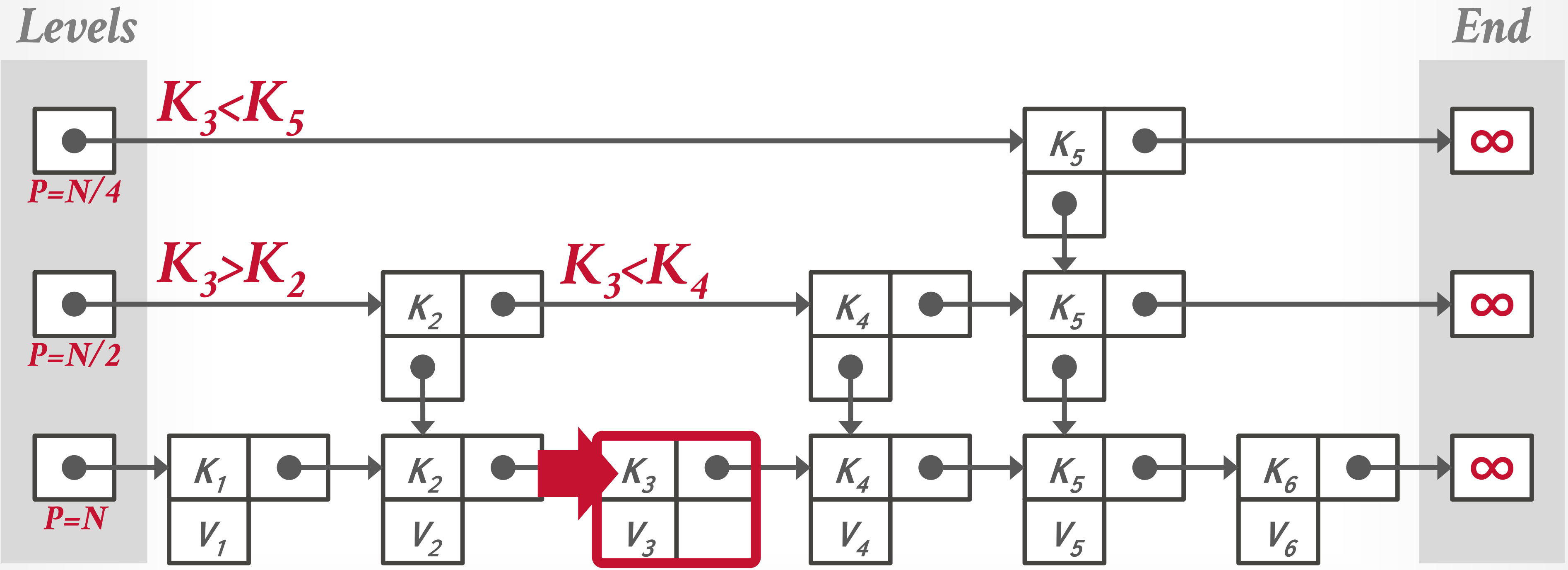
SKIP LISTS: SEARCH

Find K_3



SKIP LISTS: SEARCH

Find K_3



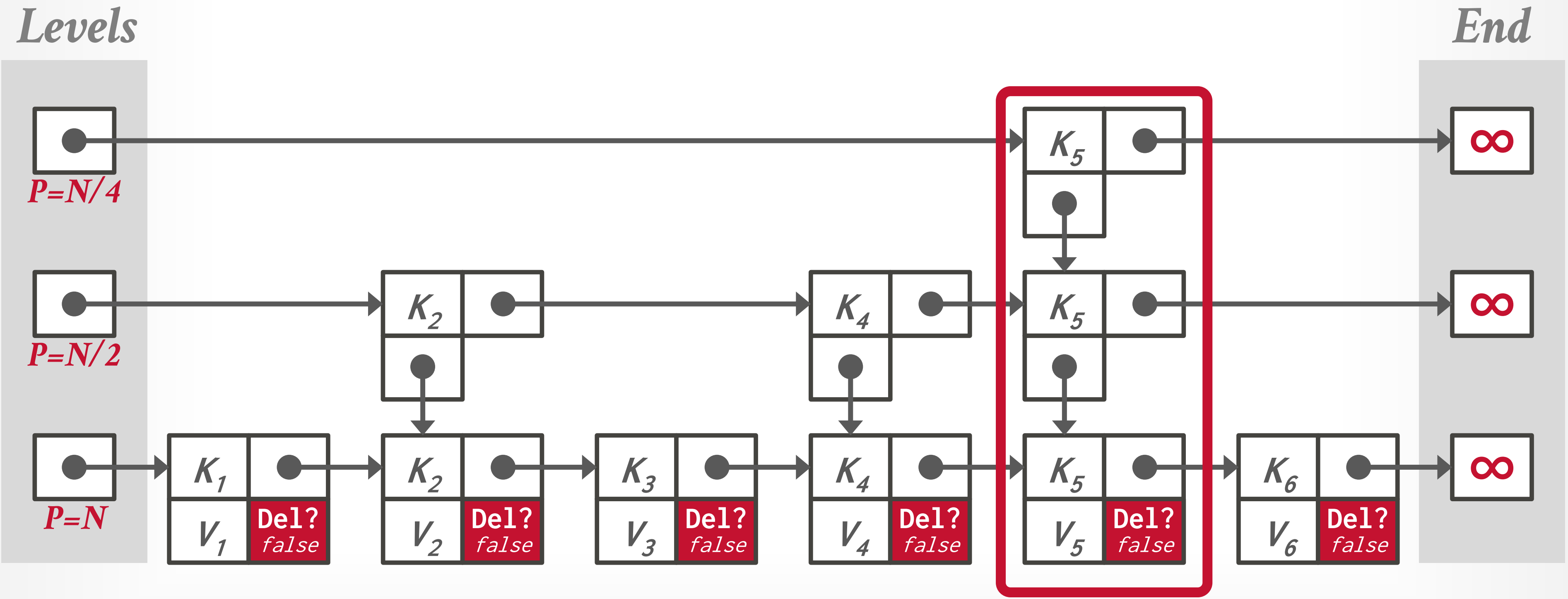
SKIP LISTS: DELETE

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.

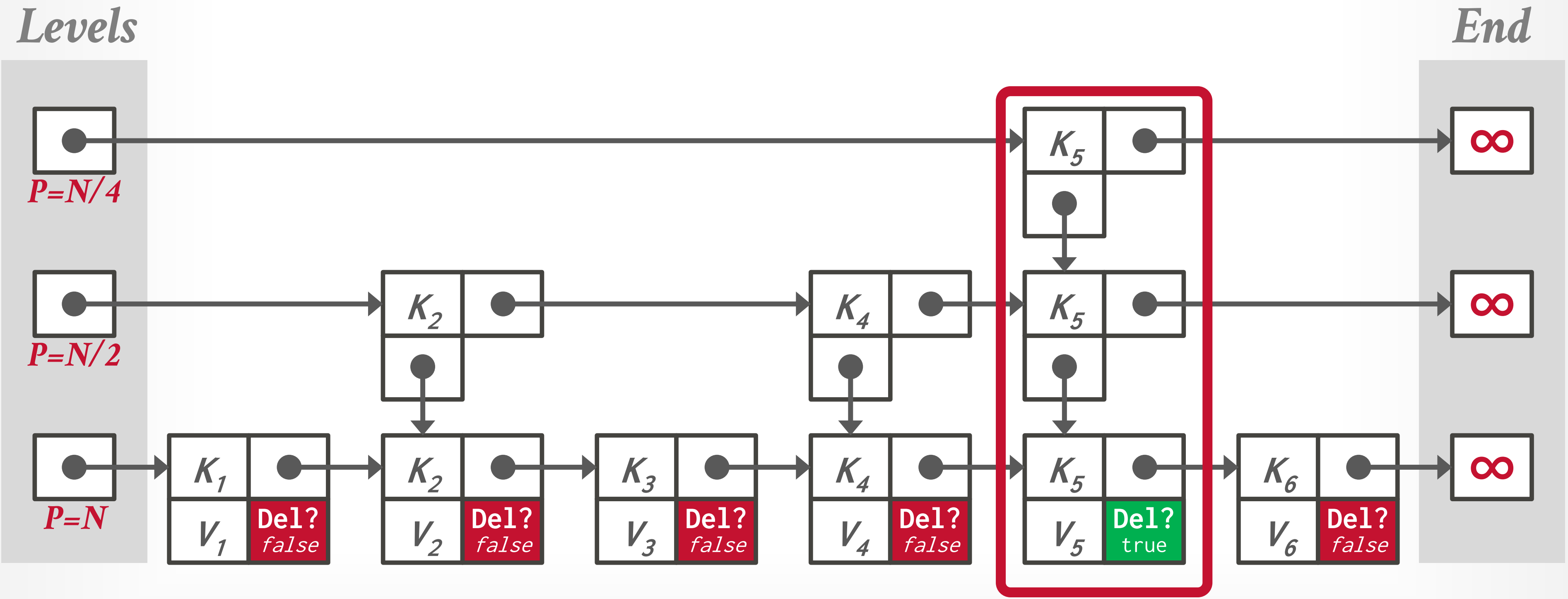
SKIP LISTS: DELETE

Delete K_5



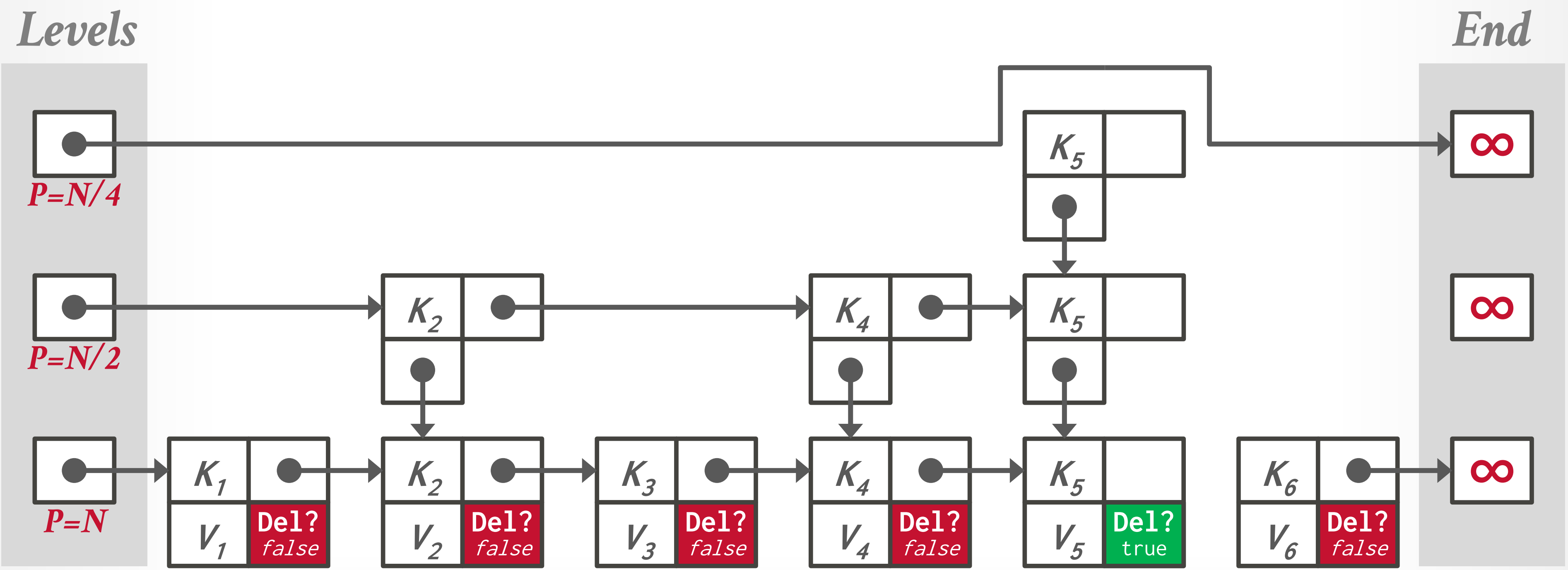
SKIP LISTS: DELETE

Delete K_5



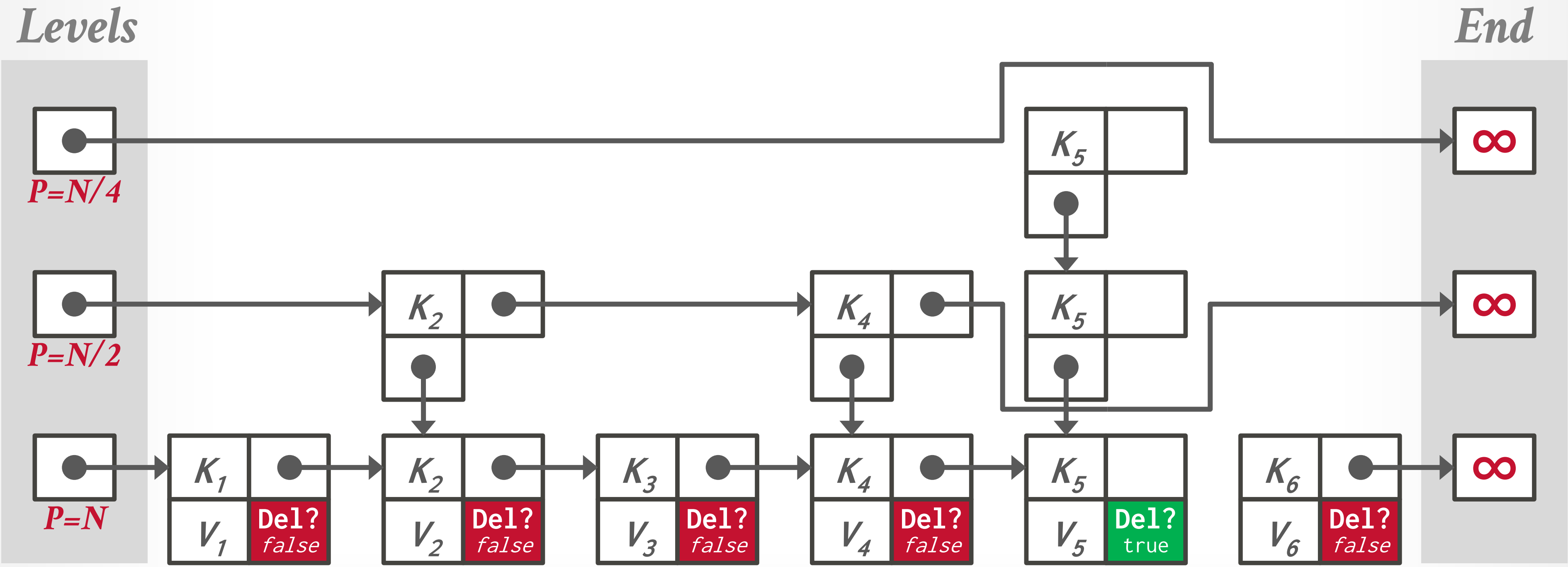
SKIP LISTS: DELETE

Delete K_5



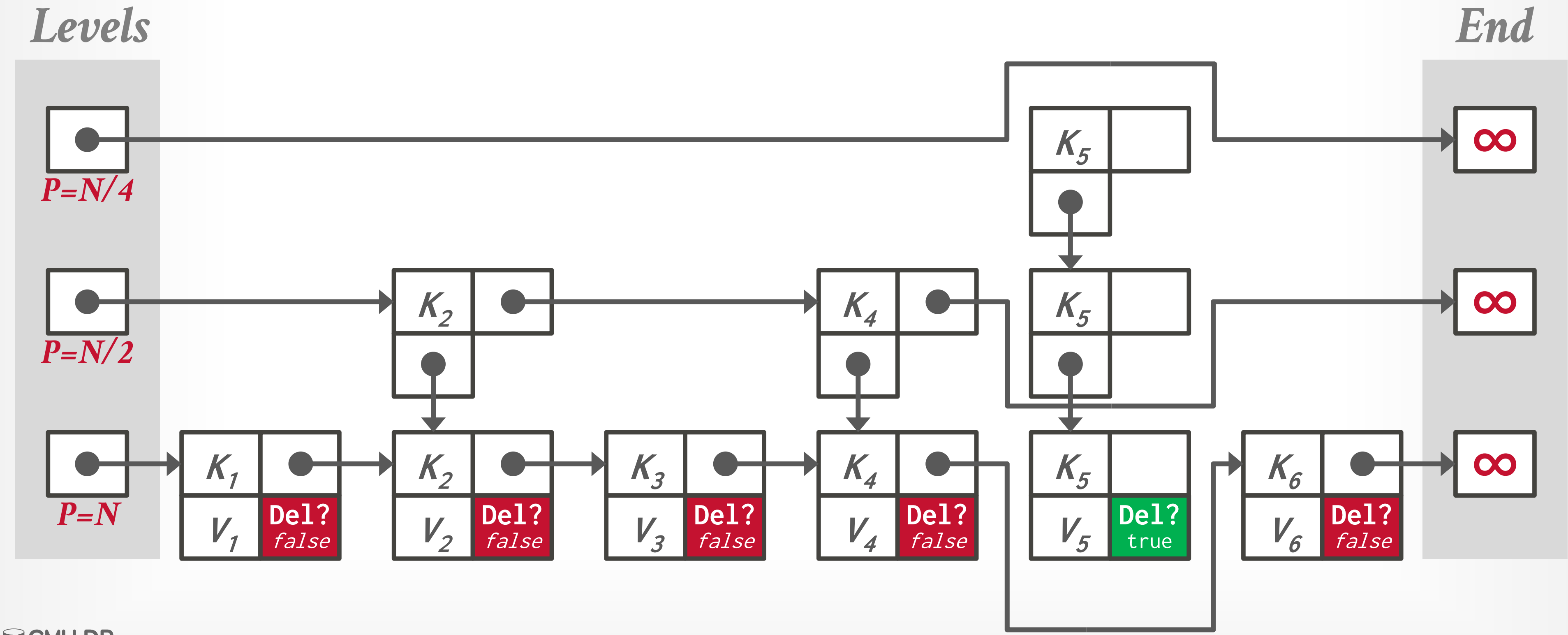
SKIP LISTS: DELETE

Delete K_5



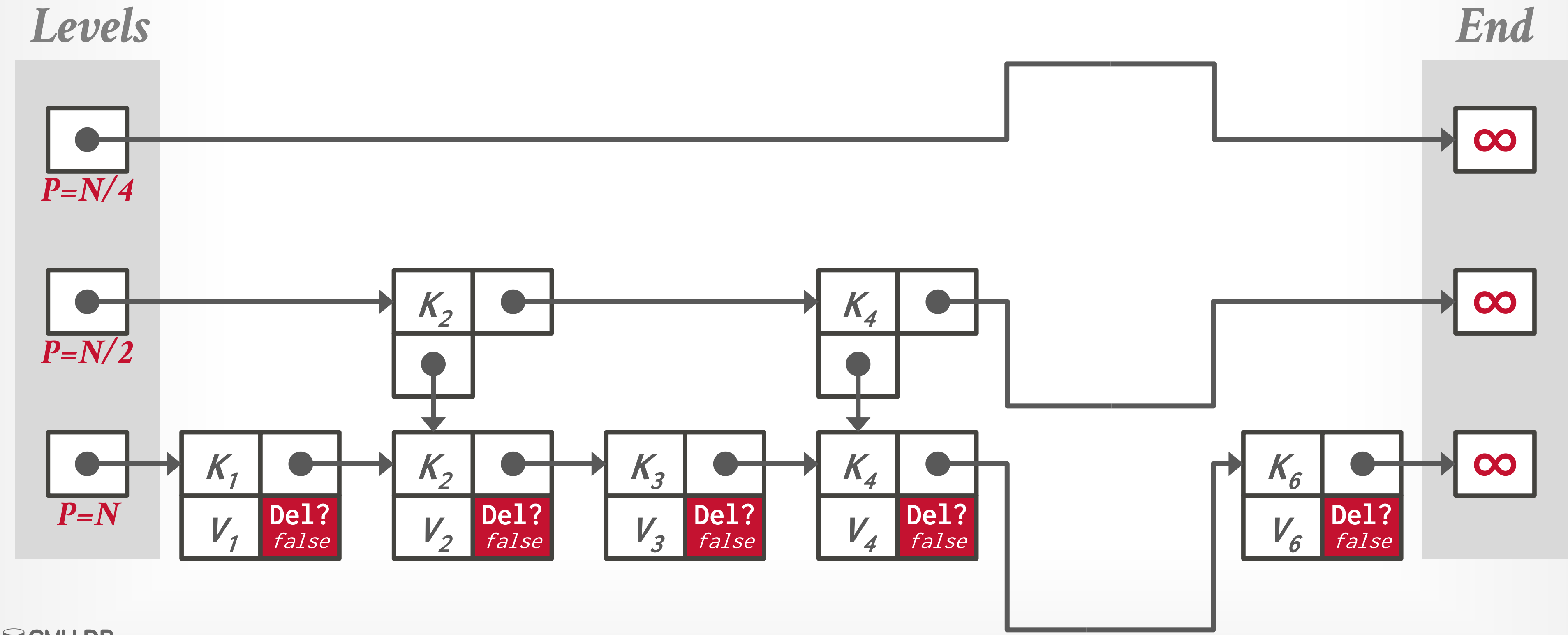
SKIP LISTS: DELETE

Delete K_5



SKIP LISTS: DELETE

Delete K_5



SKIP LISTS

Advantages:

→ Uses less memory than a typical B+Tree if you do <u>not</u> 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.

OBSERVATION

The inner node keys in a B+Tree cannot tell you whether a key exists in the index. You must always traverse to the leaf node.

This means that you could have (at least) one buffer pool page miss per level in the tree just to find out a key does not exist.

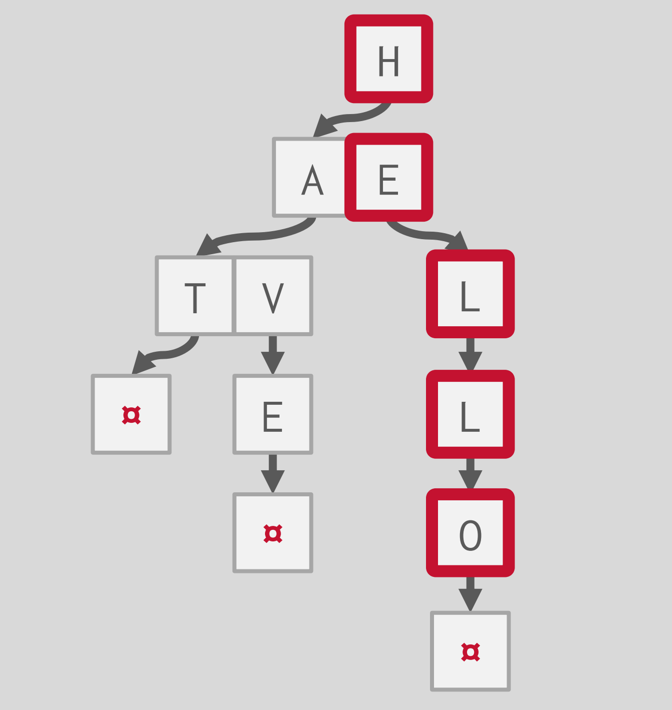
TRIE INDEX

Use a digital representation of keys to examine prefixes one-by-one.
→ aka ***Digital Search Tree***, ***Prefix Tree***.

Shape depends on keys and lengths.
→ Does not depend on existing keys or insertion order.
→ Does not require rebalancing operations.

All operations have **O(k)** complexity where **k** is the length of the key.
→ Path to a leaf node represents a key.
→ Keys are stored implicitly and can be reconstructed from paths.

Keys: HELLO, HAT, HAVE



TRIE KEY SPAN

The **span** of a trie level is the number of bits that each partial key / digit represents.

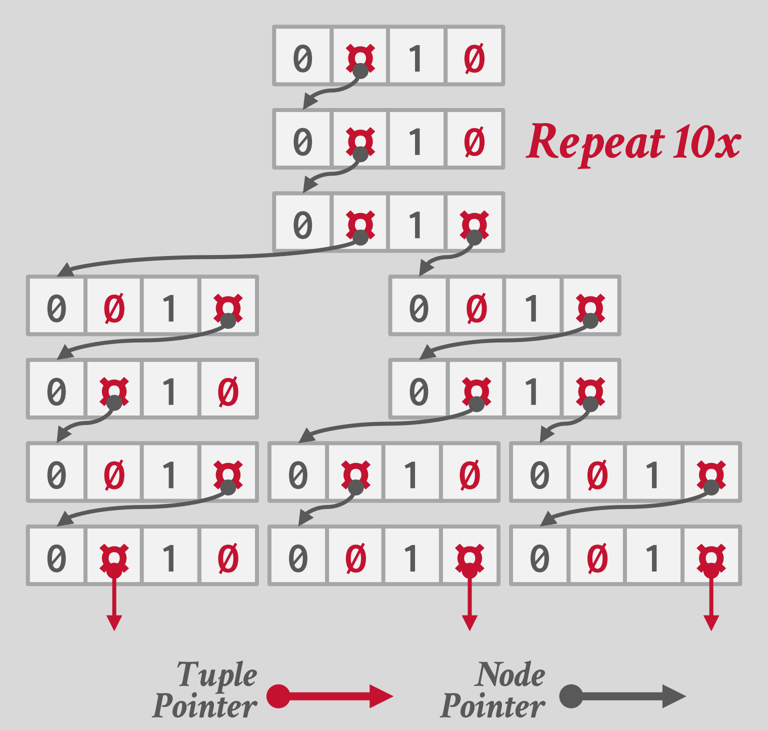
→ If the digit exists in the corpus, then store a pointer to the next level in the trie branch. Otherwise, store null.

This determines the **fan-out** of each node and the physical **height** of the tree.

→ ***n***-way Trie = Fan-Out of ***n***

TRIE KEY SPAN

1-bit Span Trie

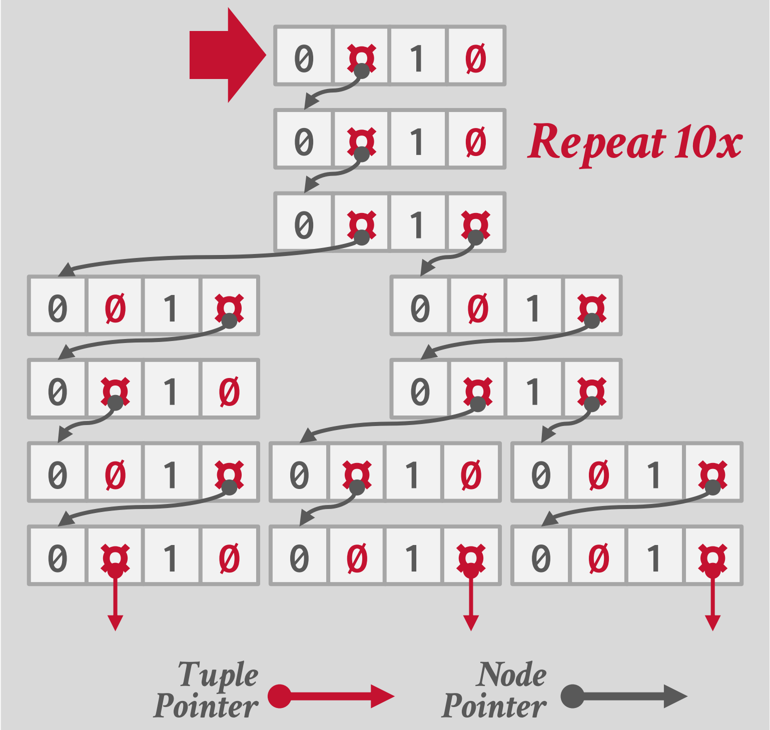


Keys: K10,K25,K31

K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie

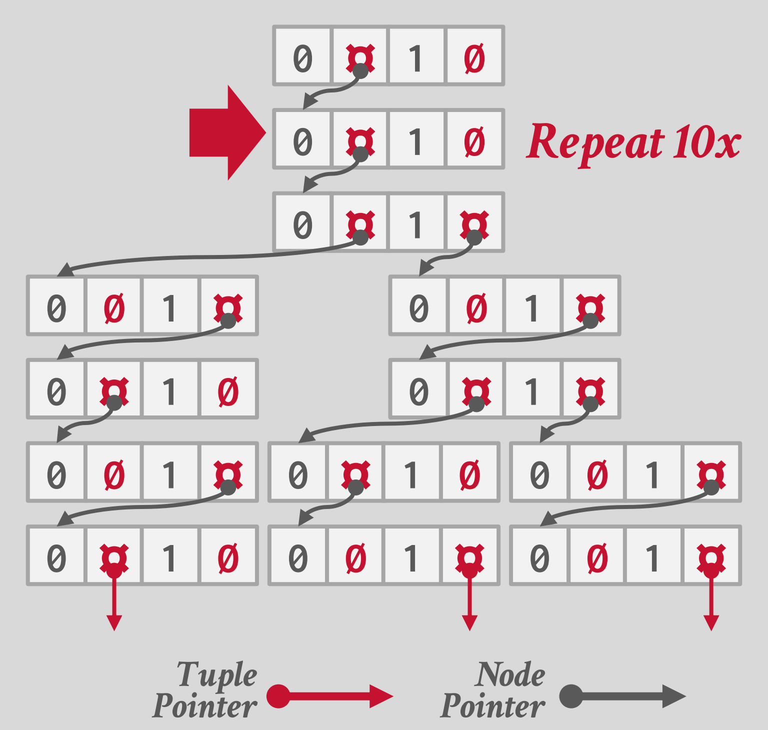


Keys: K10,K25,K31

K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie

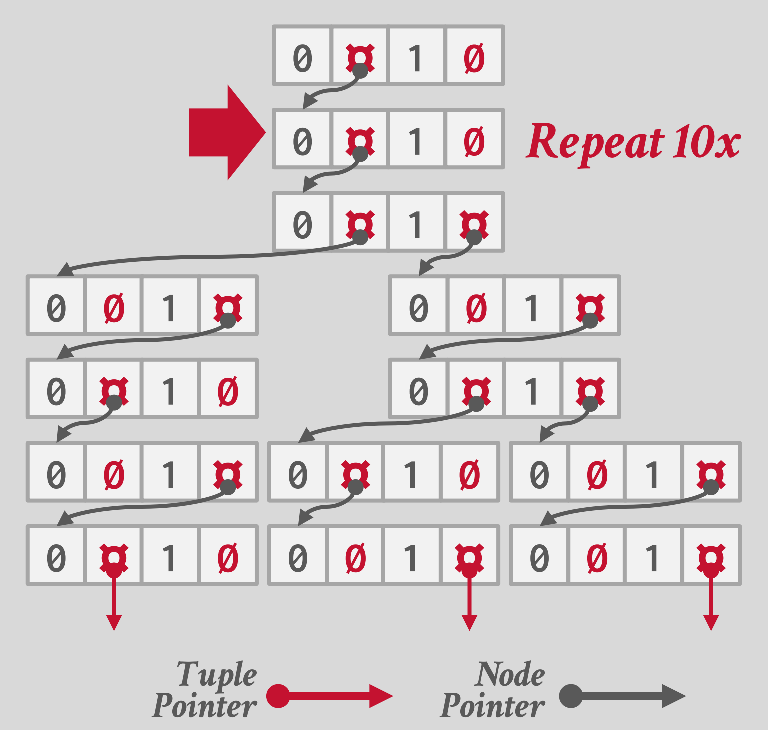


Keys: K10,K25,K31

K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie

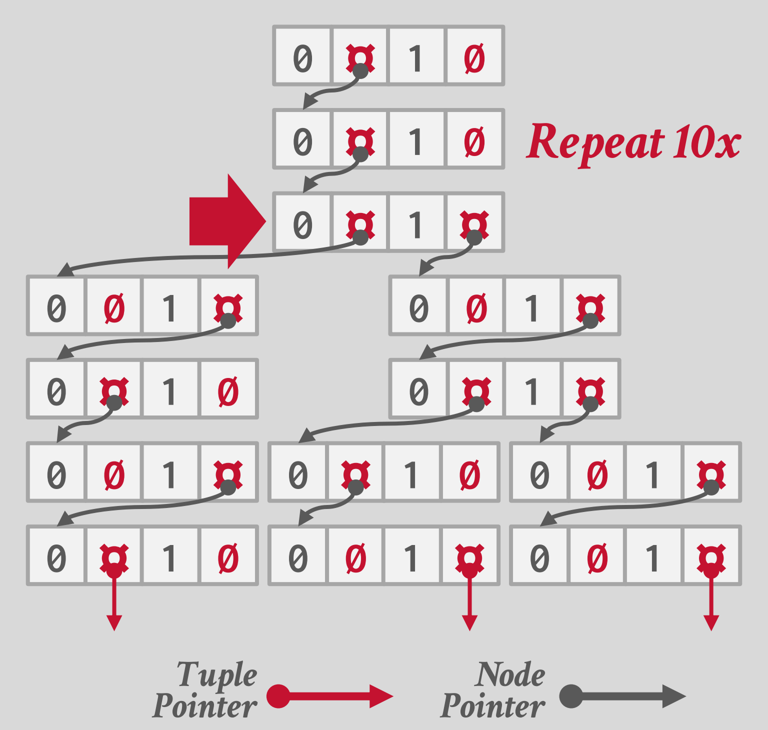


Keys: K10,K25,K31

K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie

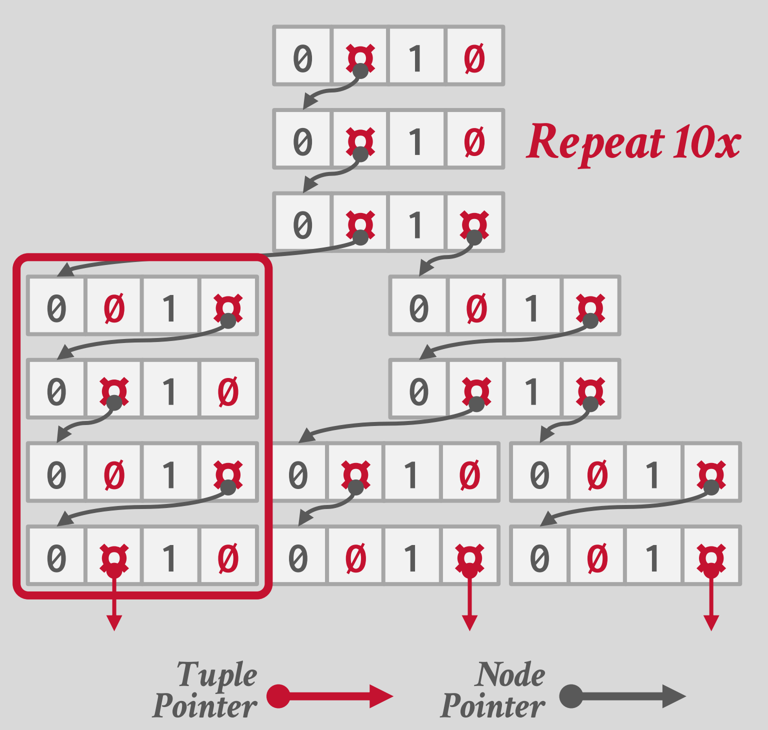


Keys: K10,K25,K31

K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie



Keys: K10,K25,K31

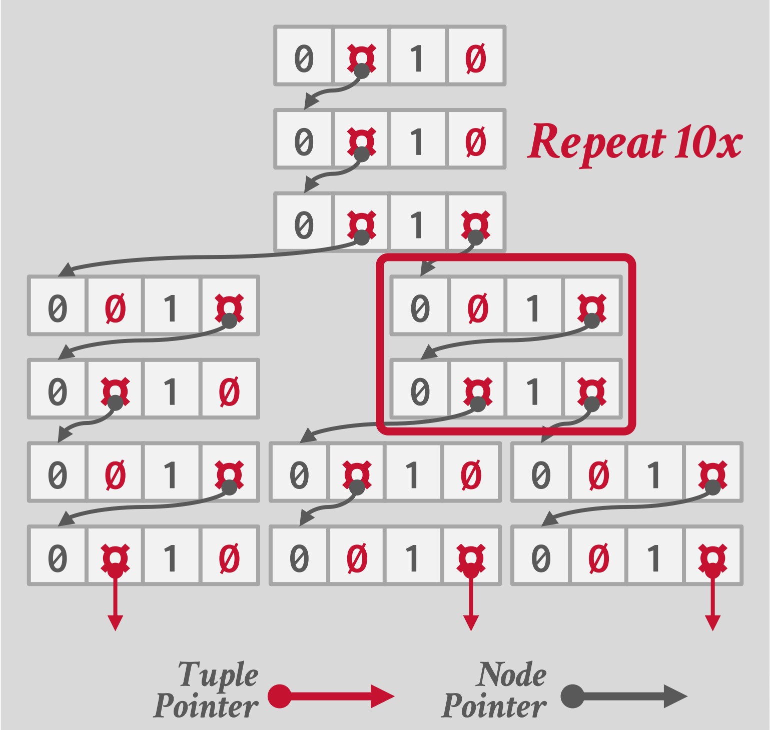
K10→ 00000000 0000**1010**

K25→ 00000000 00011001

K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie

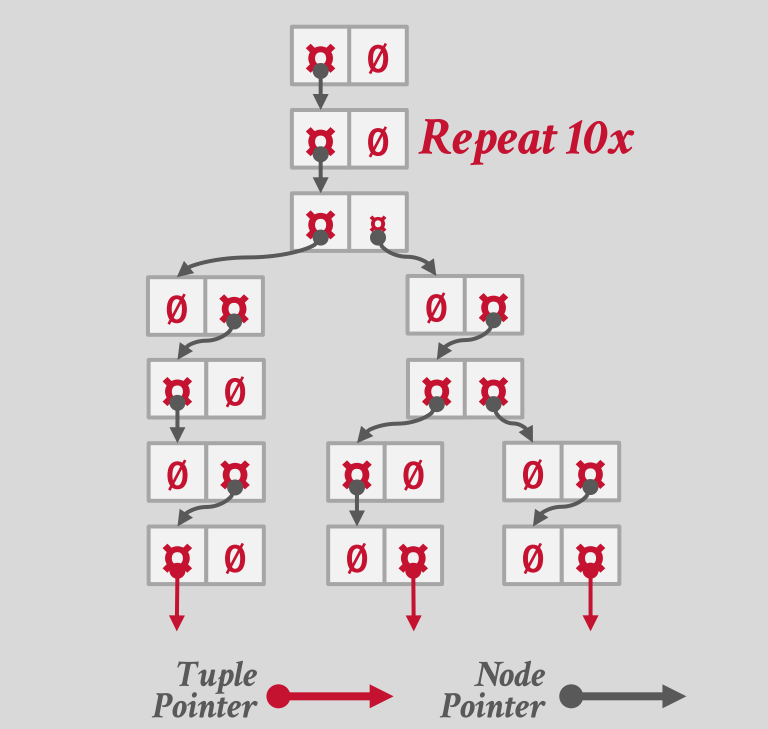


Keys: K10,K25,K31

K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie

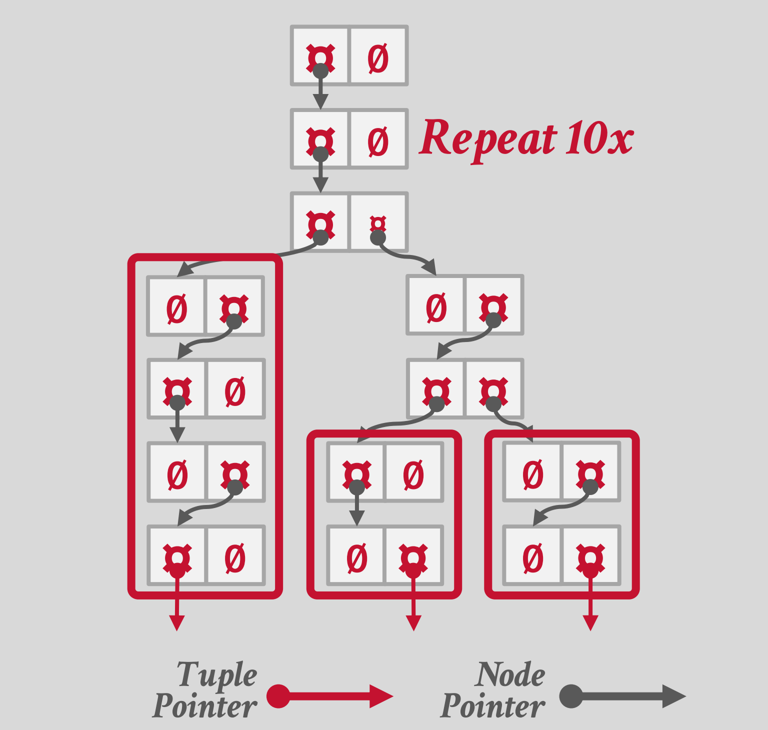


Keys: K10,K25,K31

K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111

TRIE KEY SPAN

1-bit Span Trie

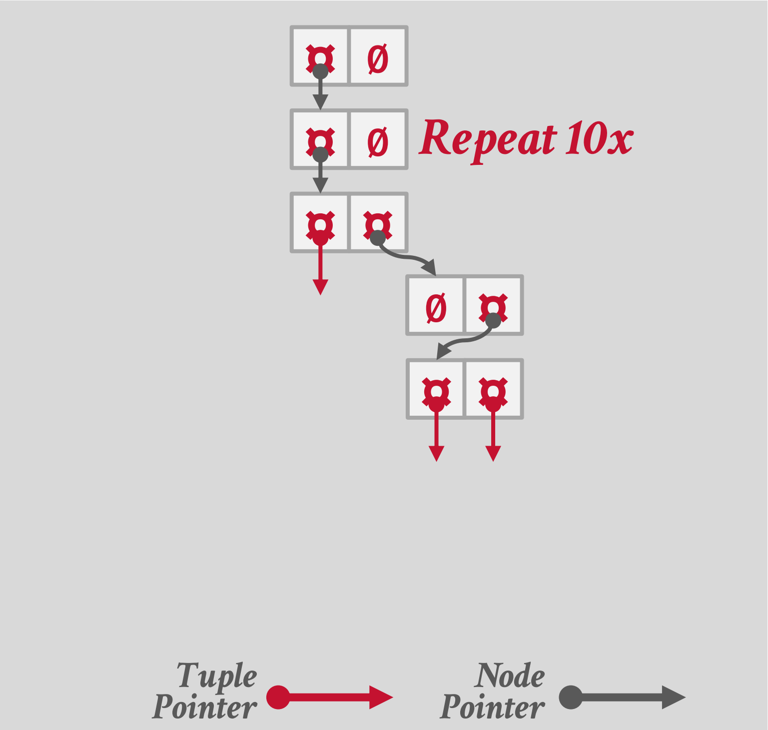


Keys: K10,K25,K31

K10→ 00000000 00001010
K25→ 00000000 00011001
K31→ 00000000 00011111

RADIX TREE

1-bit Span Radix Tree

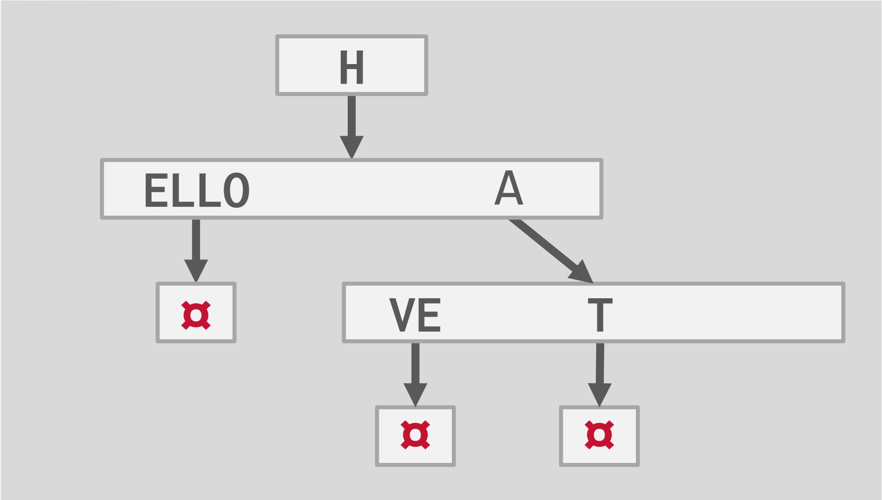


Vertically compressed trie that compacts nodes with a single child.

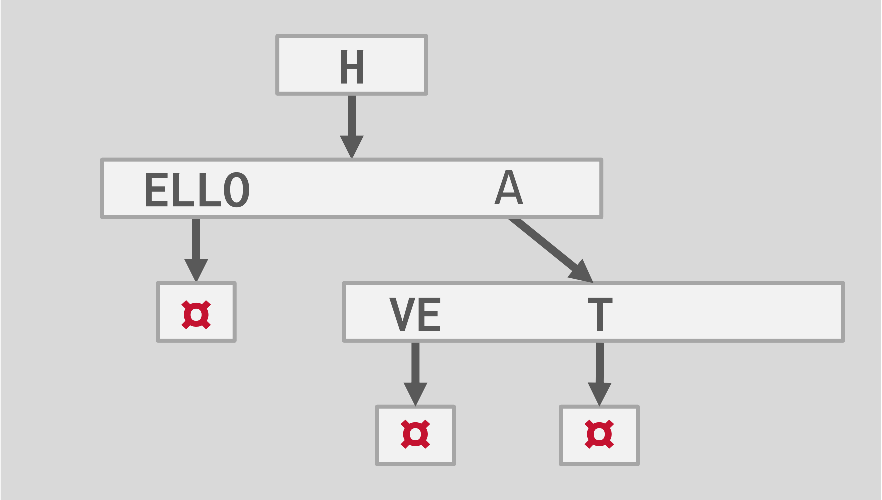
→ Also known as ***Patricia Tree***.

Can produce false positives, so the DBMS always checks the original tuple to see whether a key matches.

RADIX TREE: MODIFICATIONS

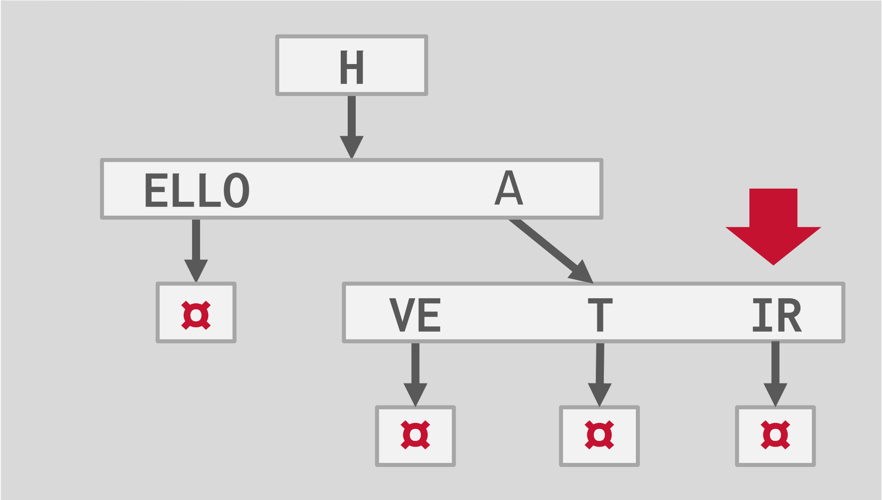


RADIX TREE: MODIFICATIONS



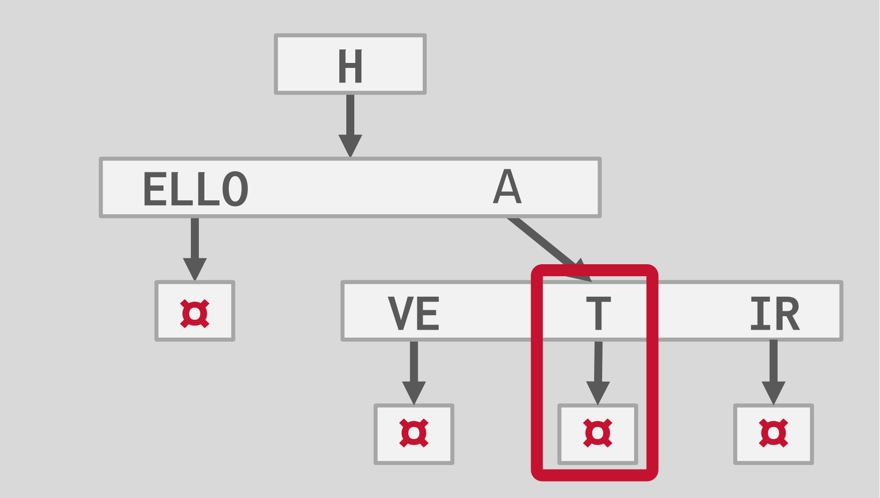
Insert HAIR

RADIX TREE: MODIFICATIONS



Insert HAIR

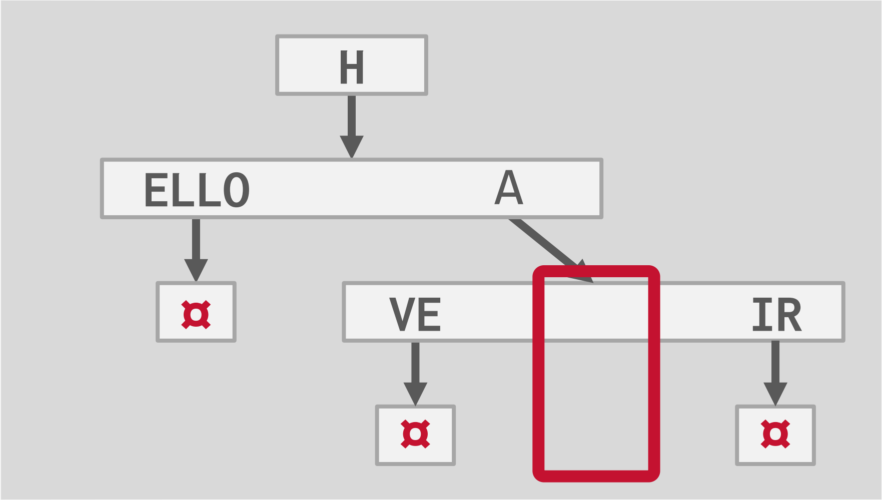
RADIX TREE: MODIFICATIONS



Insert HAIR

Delete HAT

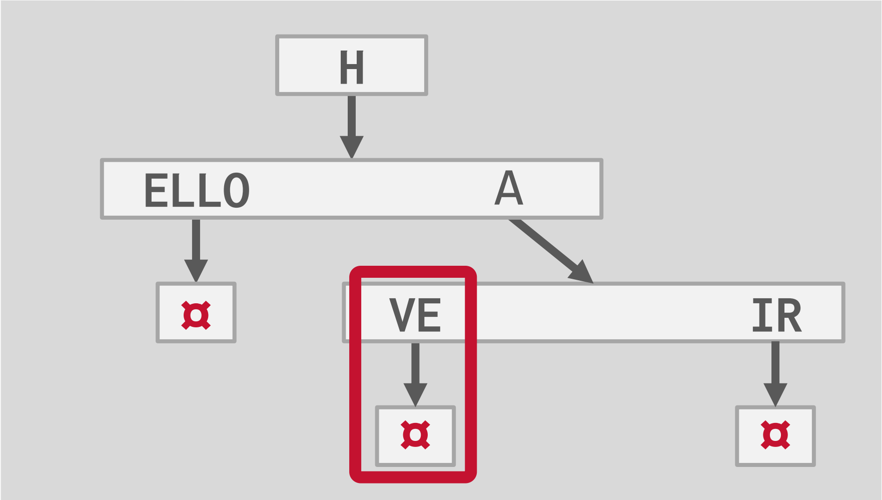
RADIX TREE: MODIFICATIONS



Insert HAIR

Delete HAT

RADIX TREE: MODIFICATIONS

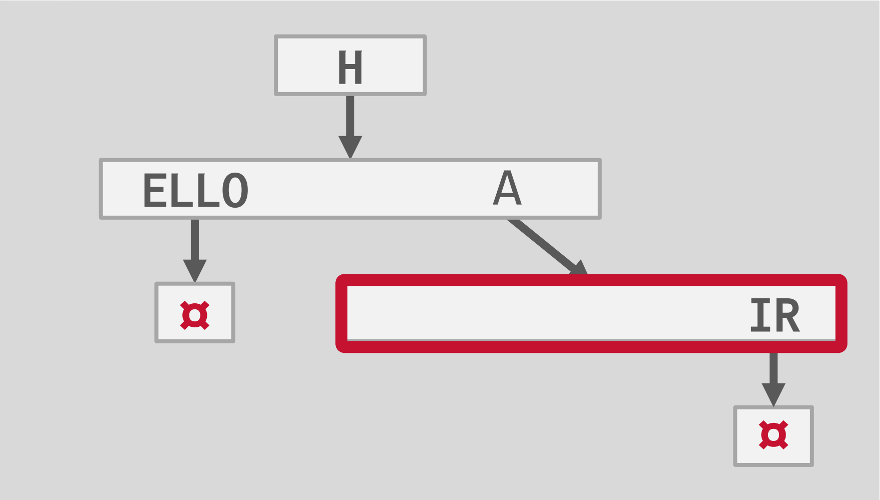


Insert HAIR

Delete HAT

Delete HAVE

RADIX TREE: MODIFICATIONS

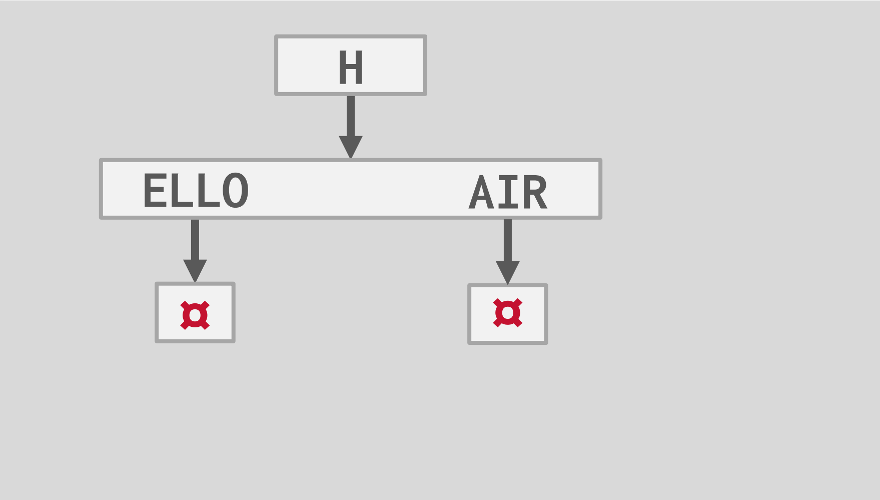


Insert HAIR

Delete HAT

Delete HAVE

RADIX TREE: MODIFICATIONS



Insert HAIR

Delete HAT

Delete HAVE

OBSERVATION

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

→ Find all customers in the 15217 zipcode.
→ Find all orders between June 2024 and September 2024.

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

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

OBSERVATION

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

→ Find all customers in the 15217 zipcode.

→ Find all orders between June 2024 and September 2024.

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

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

revisions(id,content,…)

id	content
11	Wu-Tang Clan is an American hip hop musical collective formed in Staten Island, New York City, in 1992...
22	Carnegie Mellon University (CMU) is a private research university in Pittsburgh, Pennsylvania. The institution was established in 1900 by Andrew Carnegie...
33	In computing, a database is an organized collection of data or a type of data store based on the use of a database management system (DBMS), the software...
44	Andrew Pavlo, best known as Andy Pavlo, is an associate professor of Computer Science at Carnegie Mellon University. He conducts research on database...

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

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

INVERTED INDEX

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

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

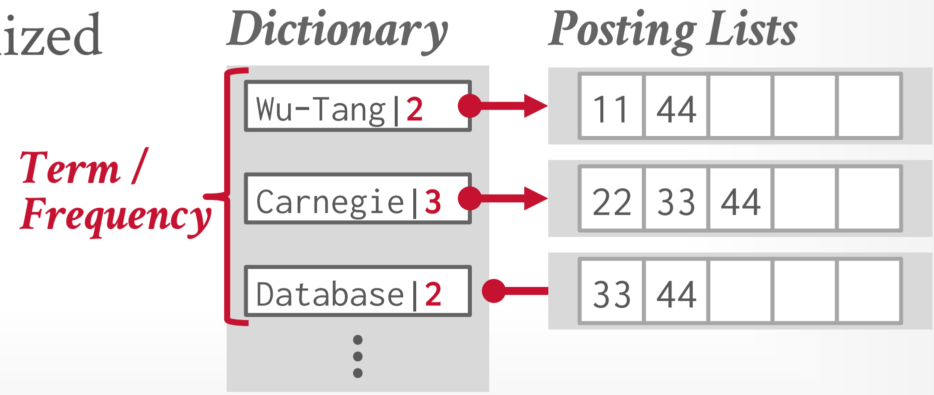
→ Originally called a ***concordance*** (1200s).

Many major DBMSs support these natively. But there are also specialized DBMSs and libraries.

Lucene, elasticsearch, Solr, Xapian, OpenSearch, Sphinx, tantivy, vespa, splunk>

revisions(id,content,…)

id	content
11	Wu-Tang Clan is an American hip hop musical collective formed in Staten Island, New York City, in 1992...
22	Carnegie Mellon University (CMU) is a private research university in Pittsburgh, Pennsylvania. The institution was established in 1900 by Andrew Carnegie...
33	In computing, a database is an organized collection of data or a type of data store based on the use of a database management system (DBMS), the software...
44	Andrew Pavlo, best known as Andy Pavlo, is an associate professor of Computer Science at Carnegie Mellon University. He conducts research on database...



INVERTED INDEX: LUCENE

Uses a **Finite State Transducer** for determining offset of terms in dictionary.

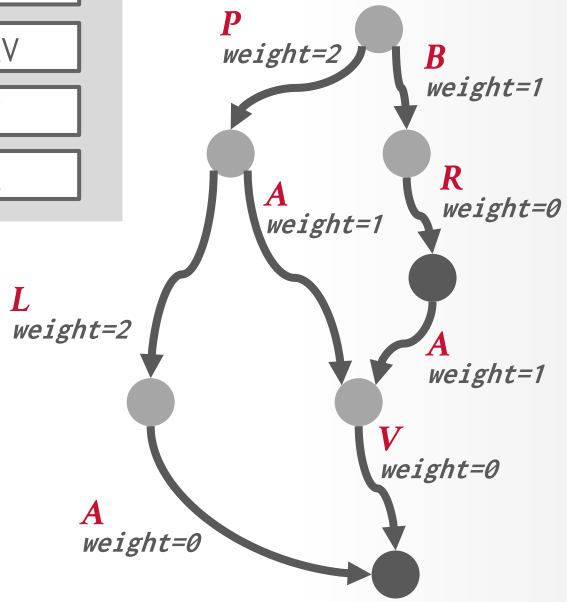
Incrementally create dictionary segments and then merge them in the background.

→ Uses compression methods we previously discussed (e.g., delta, bit packing).

→ Also supports precomputed aggregations for terms and occurrences.

Dictionary

1	BR
2	BRAV
3	PAV
4	PLA



INVERTED INDEX: LUCENE

Uses a **Finite State Transducer** for determining offset of terms in dictionary.

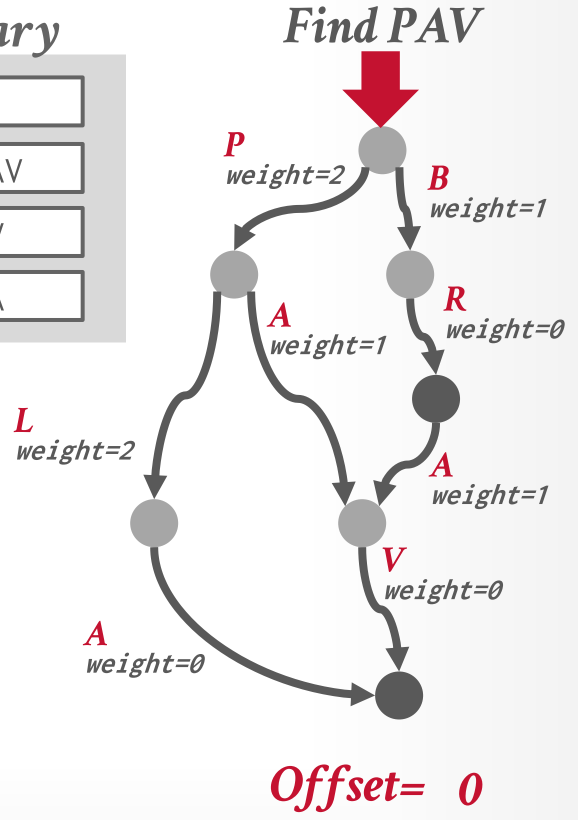
Incrementally create dictionary segments and then merge them in the background.

→ Uses compression methods we previously discussed (e.g., delta, bit packing).

→ Also supports precomputed aggregations for terms and occurrences.

Dictionary

1	BR
2	BRAV
3	PAV
4	PLA



INVERTED INDEX: LUCENE

Uses a **Finite State Transducer** for determining offset of terms in dictionary.

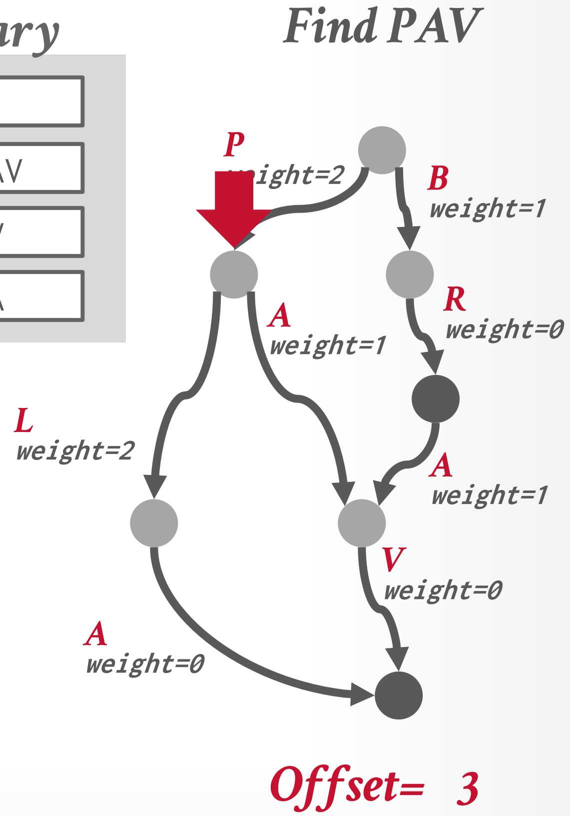
Incrementally create dictionary segments and then merge them in the background.

→ Uses compression methods we previously discussed (e.g., delta, bit packing).

→ Also supports precomputed aggregations for terms and occurrences.

Dictionary

1	BR
2	BRAV
3	PAV
4	PLA



INVERTED INDEX: LUCENE

Uses a **Finite State Transducer** for determining offset of terms in dictionary.

Incrementally create dictionary segments and then merge them in the background.

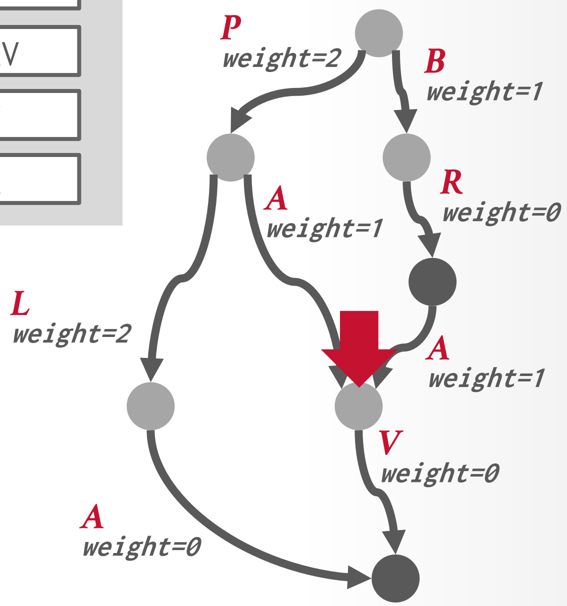
→ Uses compression methods we previously discussed (e.g., delta, bit packing).

→ Also supports precomputed aggregations for terms and occurrences.

Dictionary

1	BR
2	BRAV
3	PAV
4	PLA

Find PAV



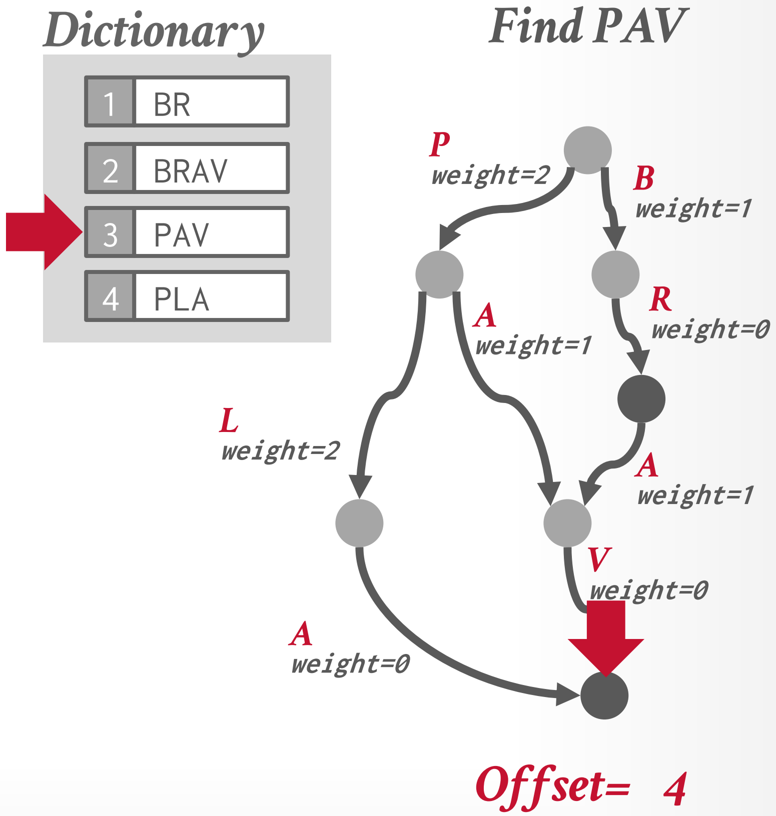
Offset= 4

INVERTED INDEX: LUCENE

Uses a **Finite State Transducer** for determining offset of terms in dictionary.

Incrementally create dictionary segments and then merge them in the background.

→ Uses compression methods we previously discussed (e.g., delta, bit packing).
→ Also supports precomputed aggregations for terms and occurrences.

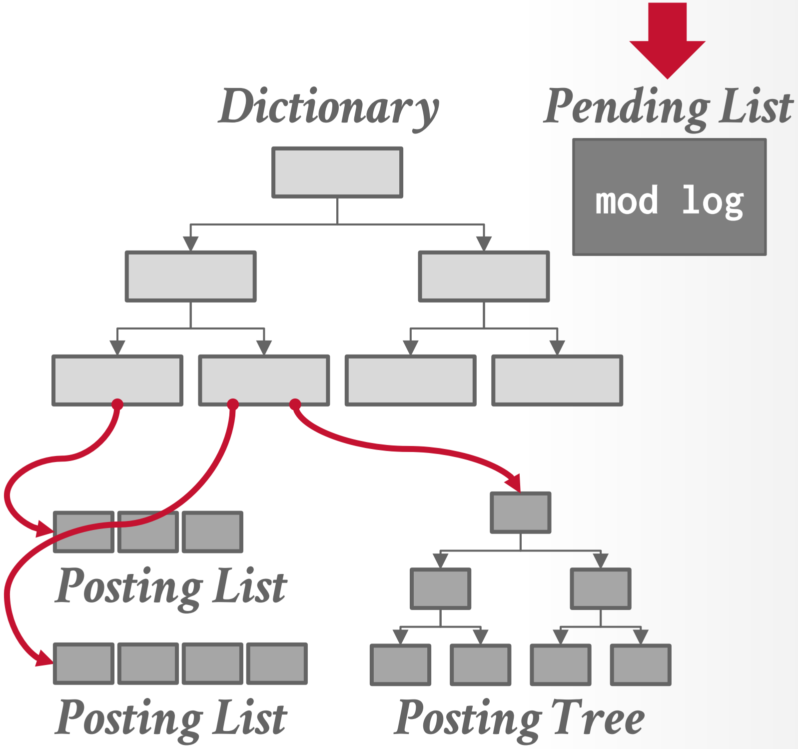


INVERTED INDEX: POSTGRESQL

PostgreSQL's **Generalized Inverted Index** (GIN) uses a B+Tree for the term dictionary that map to a posting list data structure.

Posting list contents varies depending on number of records per term:
→ **Few**: Sorted list of record ids.
→ **Many**: Another B+Tree of record ids.

Uses a separate "pending list" log to avoid incremental updates.

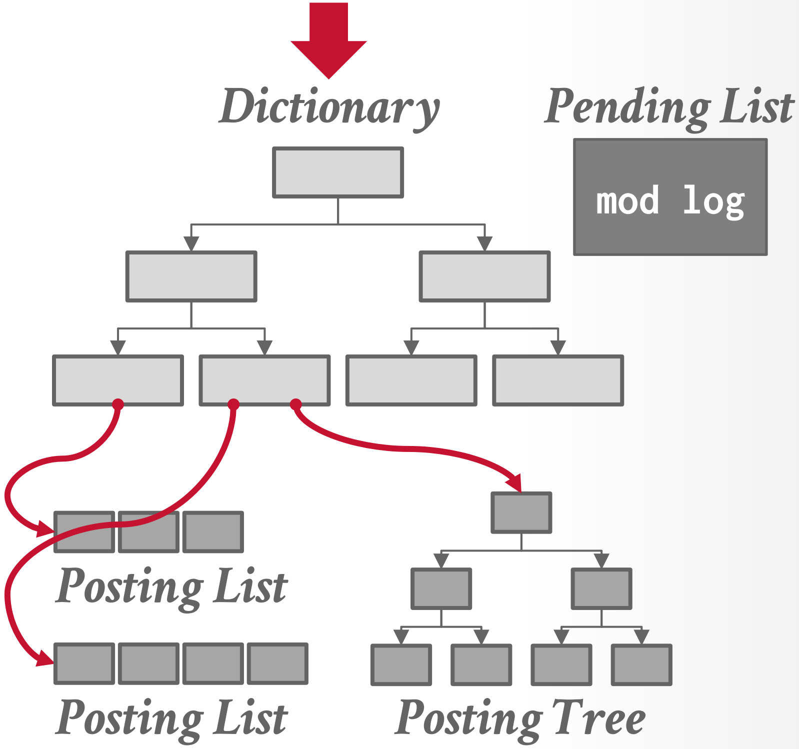


INVERTED INDEX: POSTGRESQL

PostgreSQL's **Generalized Inverted Index** (GIN) uses a B+Tree for the term dictionary that map to a posting list data structure.

Posting list contents varies depending on number of records per term:
→ **Few**: Sorted list of record ids.
→ **Many**: Another B+Tree of record ids.

Uses a separate "pending list" log to avoid incremental updates.



OBSERVATION

Inverted indexes search data based on its contents.

→ There is a little magic to tweak terms based on linguistic models.

→ Example: Normalization ("`Wu-Tang`" matches "`Wu Tang`").

Instead of searching for records containing exact keywords (e.g., "Wu-Tang"), an application may want search for records that are related to topics (e.g., "hip-hop groups with songs about slinging").

VECTOR INDEXES

Specialized data structures to perform nearest-neighbor searches on **embeddings**.

→ An embedding is an array of floating point numbers.
→ May also need to filter data before / after vector searches.

The correctness of a query depends on whether the result "feels right".

Pinecone Weaviate milvus qdrant
marqo CloseVector VectorDB turbopuffer <(°O°)>

VECTOR INDEXES

Specialized data structures to perform nearest-neighbor searches on **embeddings**.

→ An embedding is an array of floating point numbers.

→ May also need to filter data before / after vector searches.

The correctness of a query depends on whether the result "feels right".

Pinecone Weaviate milvus qdrant

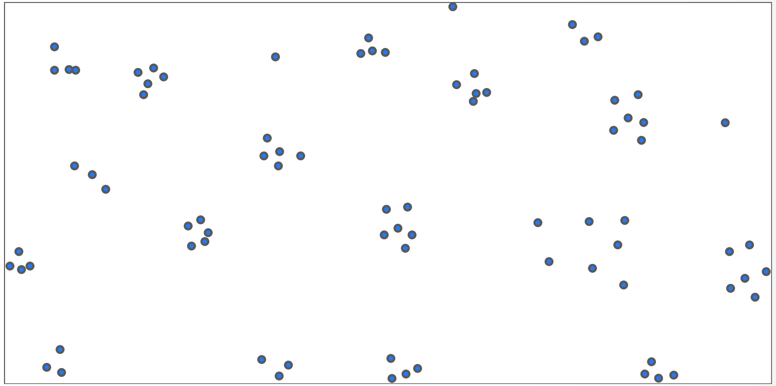
marqo CloseVector VectorDB turbopuffer <(°O°)>

VECTOR INDEXES: INVERTED FILE

Partition vectors into smaller groups using a clustering algorithm.

To find a match, use same clustering algorithm to map into a group, then scan that group's vectors.

→ Also check nearby groups to improve accuracy.



Preprocess / quantize vectors to reduce dimensionality.

Example: IVFFlat

Source: Chi Zhang

VECTOR INDEXES: INVERTED FILE

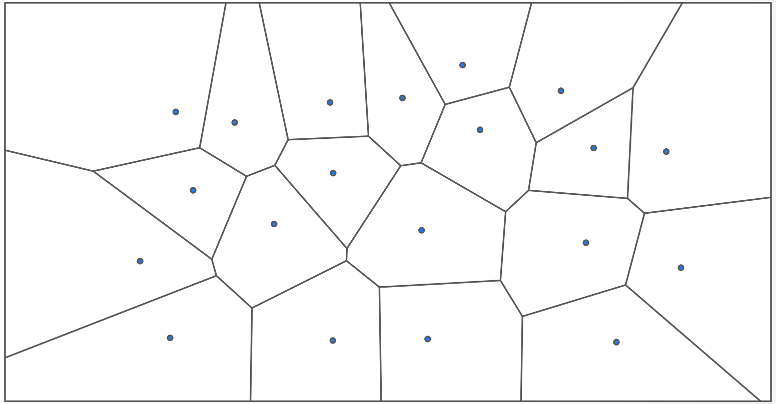
Partition vectors into smaller groups using a clustering algorithm.

To find a match, use same clustering algorithm to map into a group, then scan that group's vectors.

→ Also check nearby groups to improve accuracy.

Preprocess / quantize vectors to reduce dimensionality.

Example: IVFFlat



Source: Chi Zhang

VECTOR INDEXES: INVERTED FILE

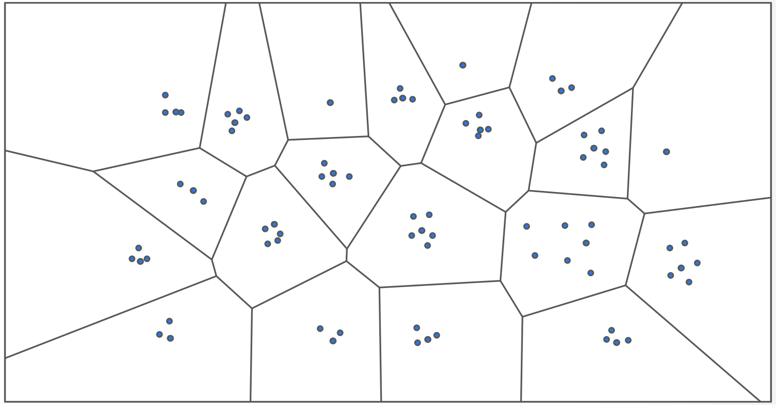
Partition vectors into smaller groups using a clustering algorithm.

To find a match, use same clustering algorithm to map into a group, then scan that group's vectors.

→ Also check nearby groups to improve accuracy.

Preprocess / quantize vectors to reduce dimensionality.

Example: IVFFlat



Source: Chi Zhang

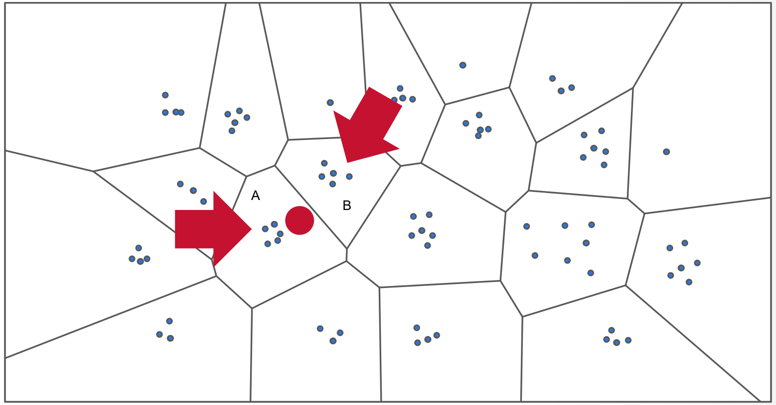
VECTOR INDEXES: INVERTED FILE

Partition vectors into smaller groups using a clustering algorithm.

To find a match, use same clustering algorithm to map into a group, then scan that group's vectors.
→ Also check nearby groups to improve accuracy.

Preprocess / quantize vectors to reduce dimensionality.

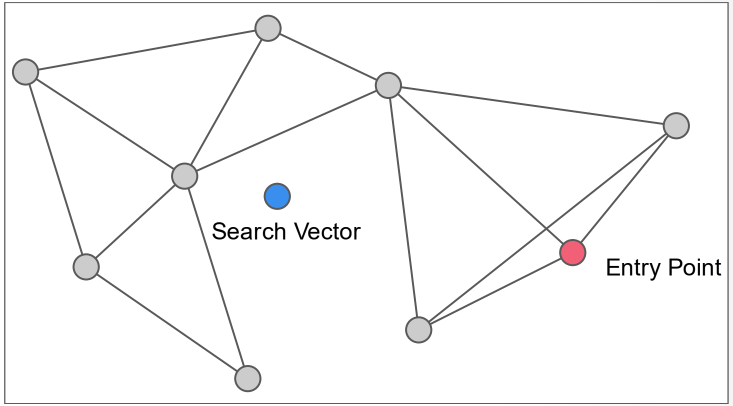
Example: IVFFlat



Source: Chi Zhang

VECTOR INDEXES: NAVIGABLE SMALL WORLDS

Build a graph where each node represents a vector and it has edges to its ***n*** nearest neighbors.
→ Can use multiple levels of graphs (HNSW)



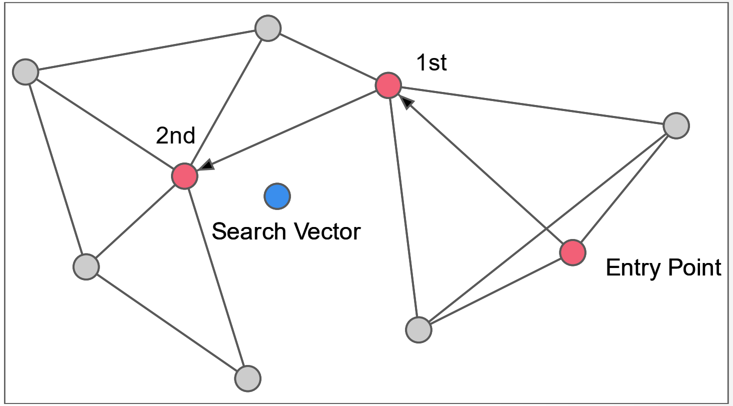
To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Example: Faiss, hnswlib

Source: Chi Zhang

VECTOR INDEXES: NAVIGABLE SMALL WORLDS

Build a graph where each node represents a vector and it has edges to its ***n*** nearest neighbors.
→ Can use multiple levels of graphs (HNSW)



To find a match for a given vector, enter the graph and then greedily choose the next edge that moves closer to that vector.

Example: Faiss, hnswlib

Source: Chi Zhang

CONCLUSION

We will see filters again this semester.

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!