

Order-Preserving Trees



Lecture #09



Database Systems
15-445/15-645
Fall 2017



Andy Pavlo
Computer Science Dept.
Carnegie Mellon Univ.

ADMINISTRIVIA

Project #1 is due Monday
October 2nd @ 11:59pm

Homework #3 is due Wednesday
October 4th @ 11:59pm



STATUS

We are now going to talk about how to support the DBMS's execution engine to read/write data from pages.

Two types of data structures:

- Hash Tables
- Trees

Query Planning

Operator Execution

Access Methods

Buffer Pool Manager

Disk Manager



DATA STRUCTURES

Internal Meta-data

Core Data Storage

Temporary Data Structures

Table Indexes



TABLE INDEXES

A **table index** is a replica of a subset of a table's columns that are organized and/or sorted for efficient access using a subset of those columns.

The DBMS ensures that the contents of the table and the index are always in sync.



TABLE INDEXES

It is the DBMS's job to figure out the best index(es) to use to execute each query.

There is a trade-off on the number of indexes to create per database.

- Storage Overhead
- Maintenance Overhead



TODAY'S AGENDA

B+Tree

Skip List

Radix Tree

Extra Index Stuff



B-TREE FAMILY

There is a specific data structure called a **B-Tree**, but then people also use the term to generally refer to a class of data structures.

- **B-Tree**
- **B+Tree**
- **B^{link}-Tree**
- **B*Tree**



B+TREE

A **B+Tree** is a self-balancing tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in **O(log n)**.

- Generalization of a binary search tree in that a node can have more than two children.
- Optimized for systems that read and write large blocks of data.

The Ubiquitous B-Tree

DOUGLAS COMER

Computer Science Department, Purdue University, West Lafayette, Indiana 47907

B-trees have become, de facto, a standard for file organization. File indexes of users, directory structures, and general-purpose data items have all been proposed and implemented using B-trees. This paper reviews B-trees and shows why they have been so successful. It discusses the major variations of the B-tree, especially the B*-tree, contrasting the relative merits and costs of each implementation. It illustrates a general purpose access method which uses a B-tree.

Keywords and Phrases: B-tree, B*-tree, B'-tree, file organization, index

CR Categories: 3.73 3.74 4.33 4.34

INTRODUCTION

The secondary storage facilities available on large computer systems allow users to store, update, and recall data from large collections of information called files. A computer must retrieve an item and place it in main memory before it can be processed. In order to make good use of the computer resources, one must organize files intelligently, making the retrieval process efficient.

The choice of a good file organization depends on the kinds of retrieval to be performed. There are two broad classes of retrieval commands which can be illustrated by the following examples:

Sequential: "From our employee file, prepare a list of all employees' names and addresses," and

Random: "From our employee file, extract the information about employee J. Smith."

We can imagine a filing cabinet with three drawers of folders, one folder for each employee. The drawers might be labeled "A-G," "H-I-R," and "S-Z," while the folders

might be labeled with the employees' last names. A sequential request requires the searcher to examine the entire file, one folder at a time. On the other hand, a random request implies that the searcher, guided by the labels on the drawers and folders, need only examine one folder.

Accessing a single item in a large, randomly accessed file in a computer system is an *index* which, like the labels on the drawers and folders of the file cabinet, speeds retrieval by directing the searcher to the small part of the file containing the desired item.

Figure 1 depicts a file and its index. An index may be physically integrated with the file, or the labels on separate folders, or physically separated from the labels on the drawers. Usually the index itself is a file. If the index file is large, another index may be built on top of it to speed retrieval further, and so on. The resulting hierarchy is similar to the employee file, where the topmost index consists of labels on drawers, and the next level of index consists of labels on folders.

Natural hierarchies, like the one formed by considering last names as index entries, do not always produce the best performance.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage. The ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1979 ACM 0010-4892/79/0600-0121 \$00.75

Computing Surveys, Vol. 11, No. 2, June 1979

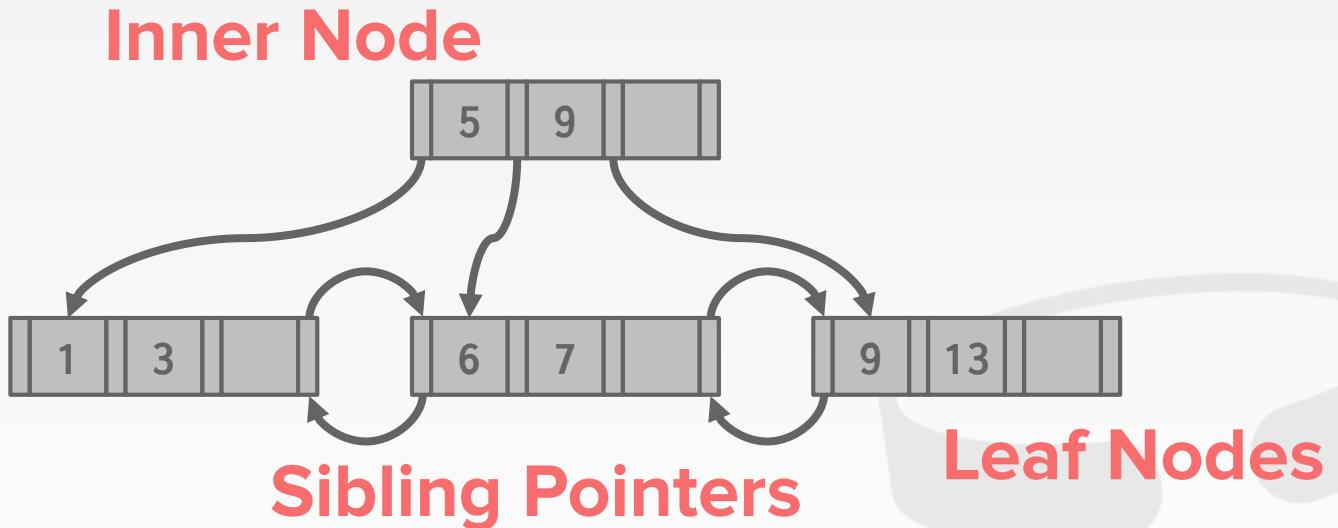
B+TREE: PROPERTIES

A B-tree is an M -way search tree with the following properties:

- It is perfectly balanced (i.e., every leaf node is at the same depth).
- Every inner node other than the root, is at least half-full
 $M/2-1 \leq \#keys \leq M-1$
- Every inner node with k keys has $k+1$ non-null children

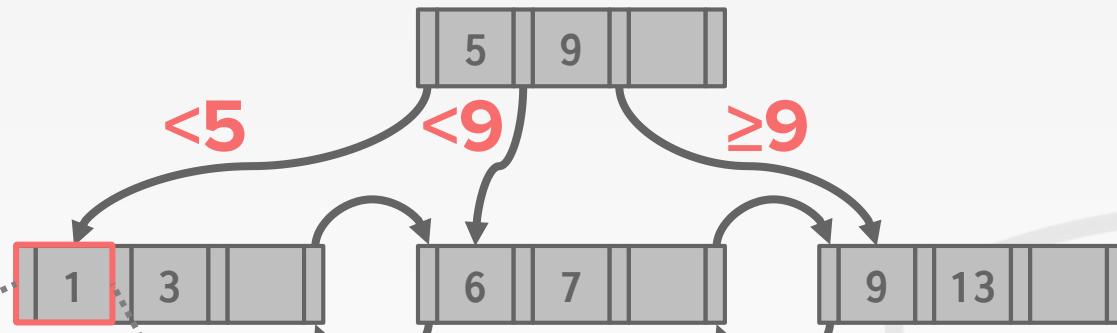


B+TREE OVERVIEW



B+TREE OVERVIEW

Inner Node



Sibling Pointers

Leaf Nodes

<value>/<key>

B+TREE NODES

Every node in the B+Tree contains an array of key/value pairs.

- The keys will always be the column or columns that you built your index on
- The values will differ based on whether the node is classified as inner nodes or leaf nodes.

The arrays are always kept in sorted order.



B+TREE: LEAF NODE VALUES

Approach #1: Record Ids

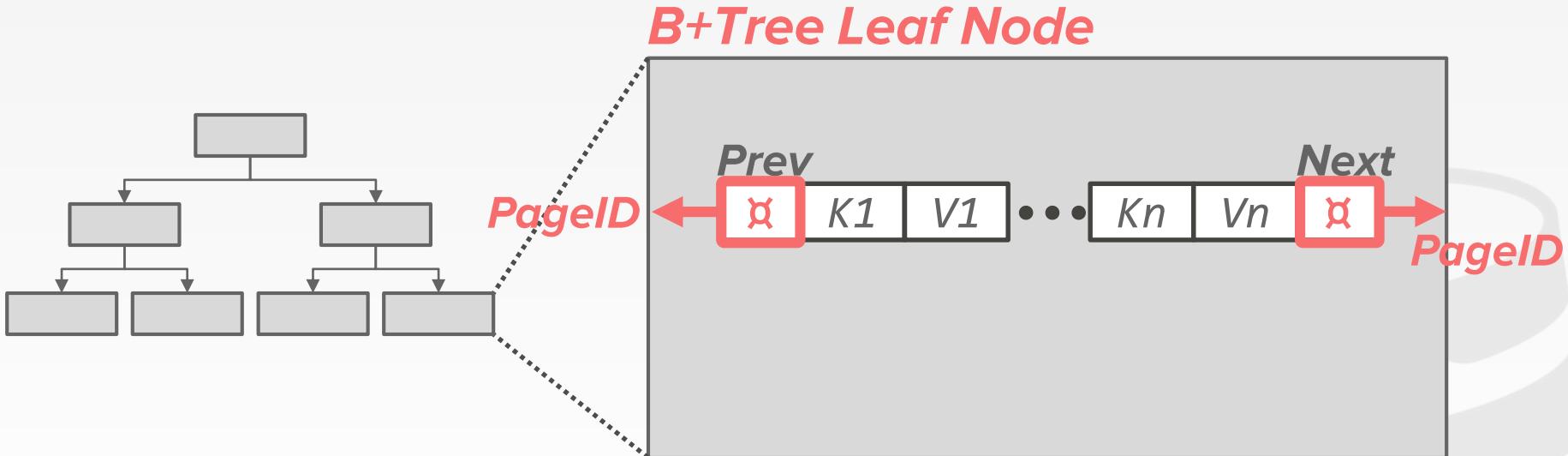
- A pointer to the location of the tuple that the index entry corresponds to.

Approach #2: Tuple Data

- The actual contents of the tuple is stored in the leaf node.
- Secondary indexes have to store the record id as their values.



B+TREE LEAF NODES

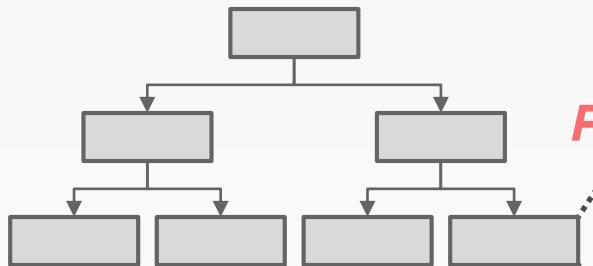


B+TREE LEAF NODES

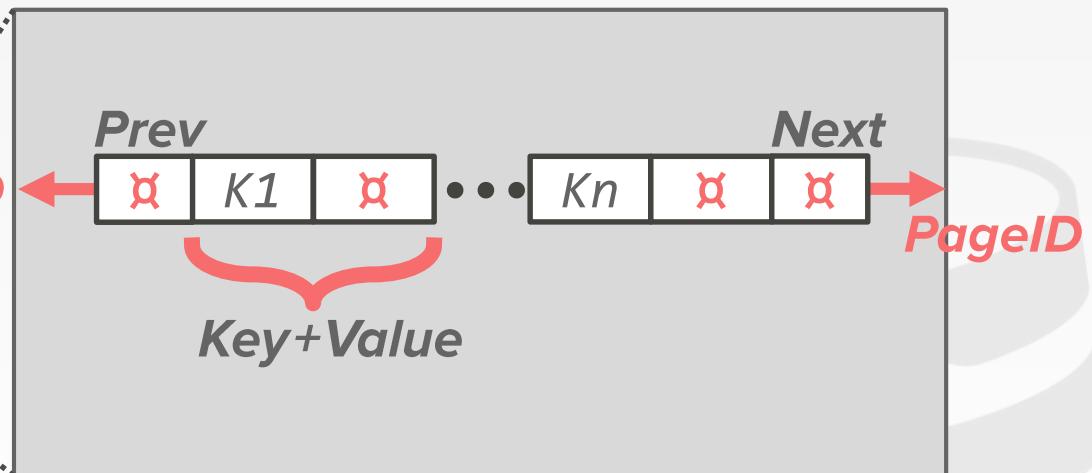
get then chunk of memory from buffer pool by pageid

values can be tuple contents or pointers to the tuples

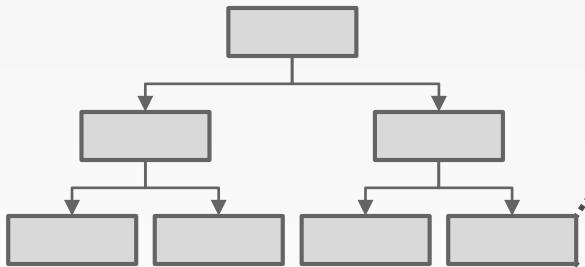
B+Tree Leaf Node



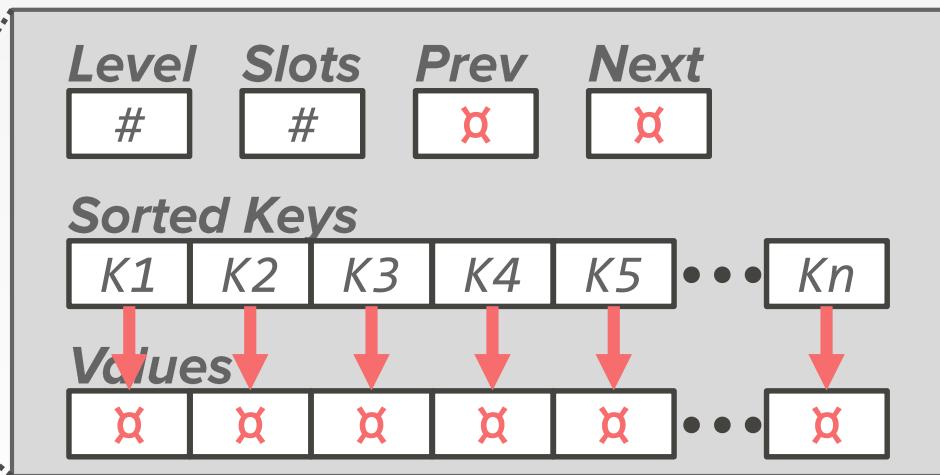
PageID



B+TREE LEAF NODES



B+Tree Leaf Node



B-TREE VS. B+TREE

The original **B-Tree** from 1972 stored keys + values in all nodes in the tree.

→ More space efficient since each key only appears once in the tree.

keys in the middle is like guidepost leading to value on the leaf node

A **B+Tree** only stores values in leaf nodes. Inner nodes only guide the search process.

Keys can be appeared more than once even if keys on the leaf level have been deleted

B+TREE: INSERT

Max.Degree: max number of keys in a node which is customized

Find correct leaf **L**.

Put data entry into **L** in sorted order.

- If **L** has enough space, done!
- Else, must split **L** into **L** and a new node **L2**
 - Redistribute entries evenly, copy up middle key.
 - Insert index entry pointing to **L2** into parent of **L**.

To split inner node, redistribute entries evenly,
but push up middle key.



Source: Chris Re

B+TREE VISUALIZATION

<http://cmudb.io/btree>

<https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html>

B+TREE: DELETE

choose from left or right sibling will yield different res, in practice, always choose one direction

Start at root, find leaf **L** where entry belongs.

Remove the entry.

- If **L** is at least half-full, done!
- If **L** has only **M/2-1** entries,
 - Try to re-distribute, borrowing from sibling (adjacent node with same parent as **L**).
 - If re-distribution fails, merge **L** and sibling.

If merge occurred, must delete entry (pointing to **L** or sibling) from parent of **L**.



Source: Chris Re

B+TREES IN PRACTICE

nodes have value
in the index

Typical Fill-Factor: 67%.

→ Average Fanout = $2 * 100 * 0.67 = 134$

some queries can be done only by lookups
on index, so index pages are preferred to be
stored in the memory than the tuple pages

Typical Capacities:

→ Height 4: $1334 = 312,900,721$ entries

→ Height 3: $1333 = 2,406,104$ entries

Pages per level:

→ Level 1 = 1 page = 8 KB

→ Level 2 = 134 pages = 1 MB

→ Level 3 = 17,956 pages = 140 MB



B+TREE DESIGN CHOICES

- Merge Threshold
- Non-Unique Indexes
- Variable Length Keys
- Prefix Compression

delete -> less than half full -> merge

insert -> more than max degree -> split



B+TREE: MERGE THRESHOLD

Some DBMSs don't always merge nodes when it is half full.

Delaying a merge operation may reduce the amount of reorganization.



B+TREE: NON-UNIQUE INDEXES

Approach #1: Duplicate Keys

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

similar to hashtable collision strategy

open address

Approach #2: Value Lists

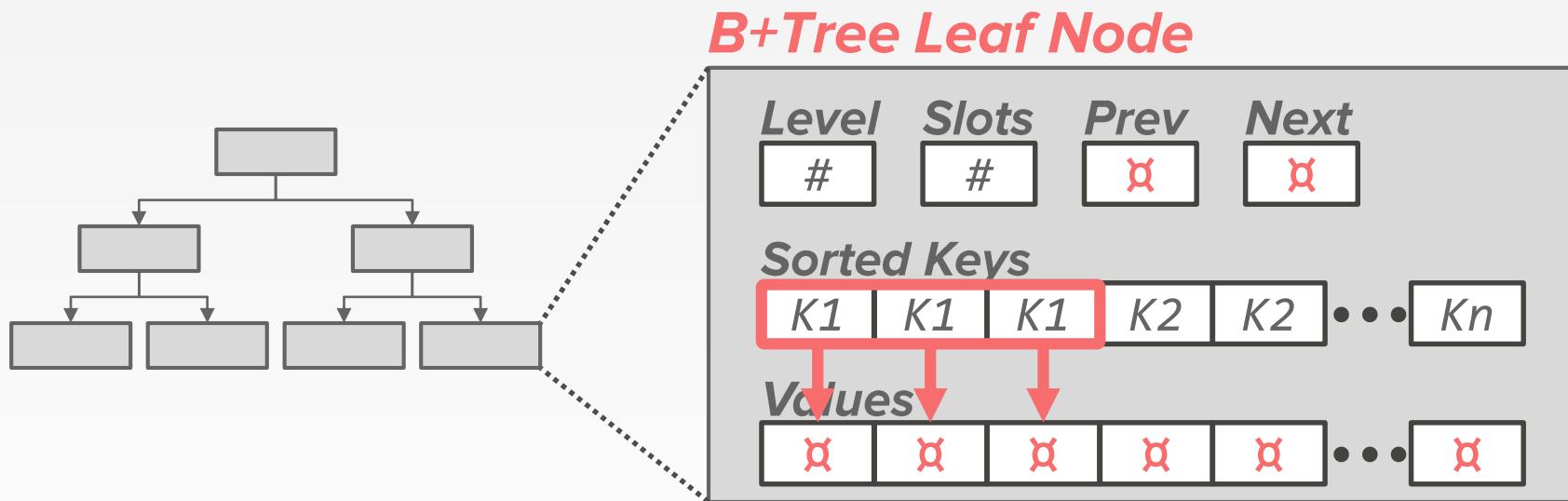
- Store each key only once and maintain a linked list of unique values.

chaining

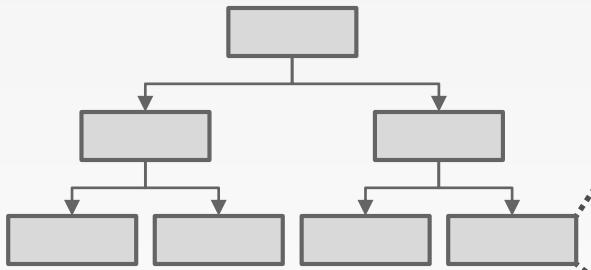
in B+ tree, leaf node actually is a page



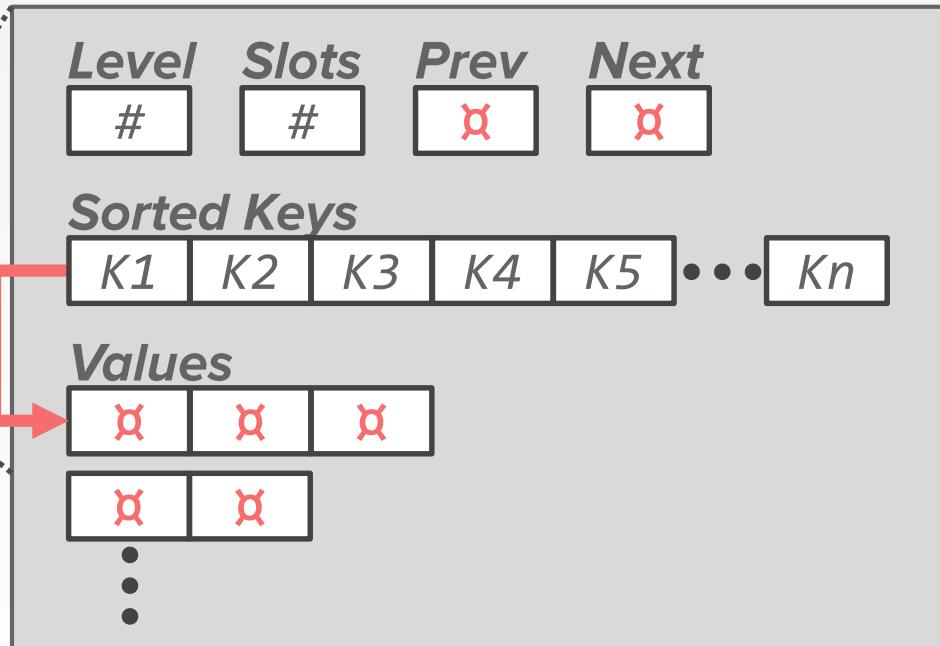
B+TREE: DUPLICATE KEYS



B+TREE: VALUE LISTS



B+Tree Leaf Node



B+TREE: VARIABLE LENGTH KEYS

question: what does b+ tree really store on leaf node?
key or key-value pair?

Approach #1: Pointers

rare

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

Approach #2: Variable Length Nodes

hard to use

- The size of each node in the B+Tree can vary.
- Requires careful memory management.

Approach #3: Key Map

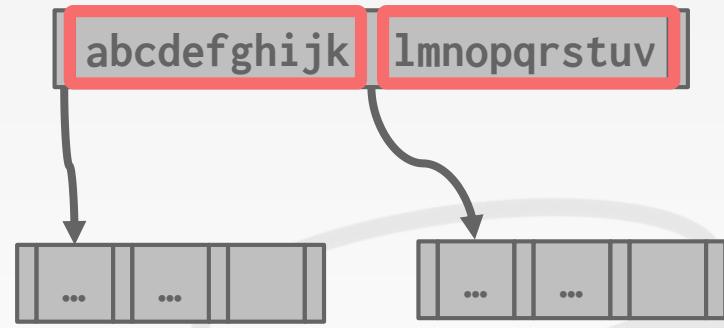
- Embed an array of pointers that map to the key + value list within the node.



B+TREE: PREFIX COMPRESSION

The keys in the inner nodes are only used to "direct traffic".
→ We don't actually need the entire key.

Store a minimum prefix that is needed to correctly route probes into the index.

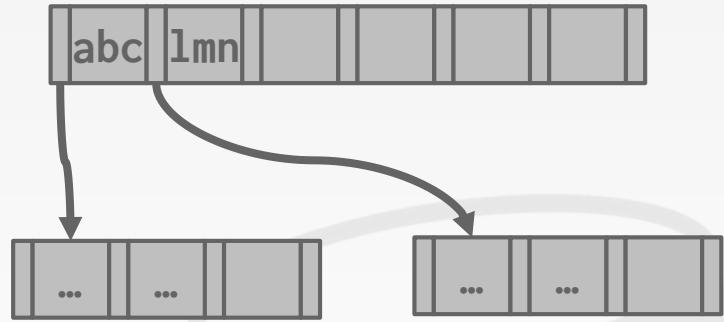


B+TREE: PREFIX COMPRESSION

full key will be stored in the leaf node

The keys in the inner nodes are only used to "direct traffic".
→ We don't actually need the entire key.

Store a minimum prefix that is needed to correctly route probes into the index.



B+TREE BULK INSERT

The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.

normal insert: from root to leaf -> top down way

bulk insert: generate b+ tree from existed sorted keys -> bottom up way



B+TREE BULK INSERT

The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.

Keys: 3, 7, 9, 13, 6, 1
Sorted Keys: 1, 3, 6, 7, 9, 13



B+TREE BULK INSERT

The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.

Keys: 3, 7, 9, 13, 6, 1

Sorted Keys: 1, 3, 6, 7, 9, 13

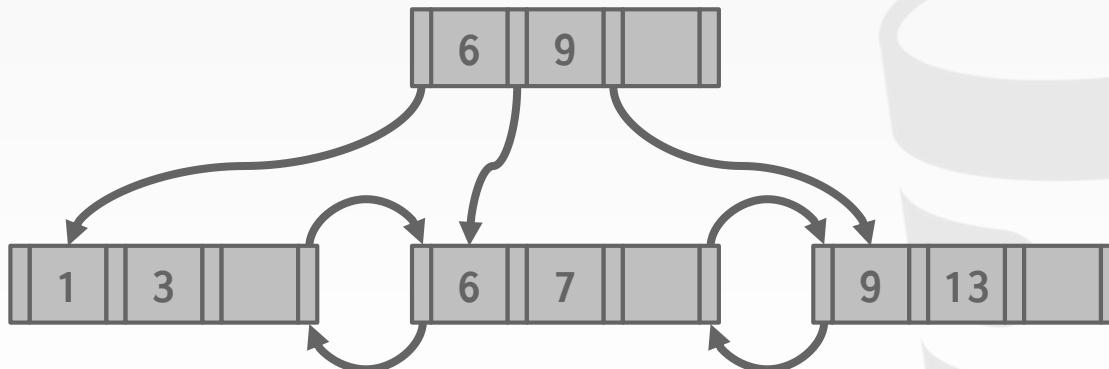


B+TREE BULK INSERT

The fastest/best way to build a B+Tree is to first sort the keys and then build the index from the bottom up.

Keys: 3, 7, 9, 13, 6, 1

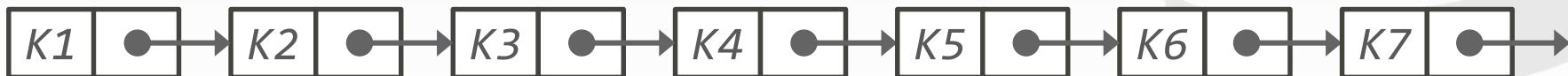
Sorted Keys: 1, 3, 6, 7, 9, 13



OBSERVATION

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

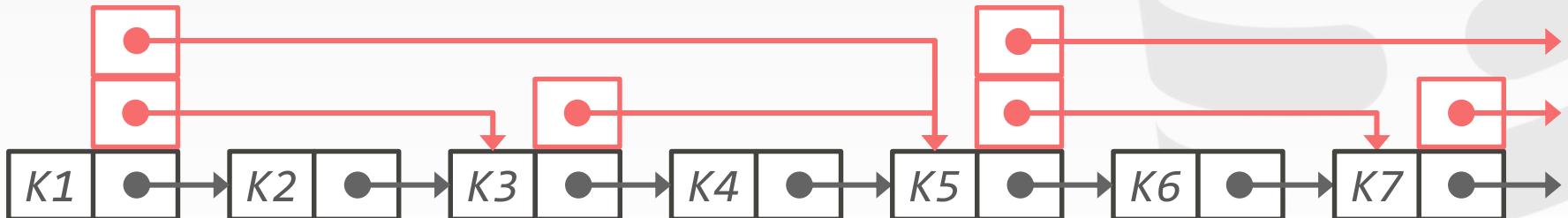
All operations have to linear search.
→ Average Cost: $O(N)$



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-level linked list

has extra entries that allow you jump over intermediate nodes
Invented in 1990.

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 search is well suited for representing almost 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 trees that are very slow to search. If it were possible to randomly permute the list of items to be inserted, trees would work well with high probability for any sequence of insertions. Unfortunately, it is not possible to do so randomly permuting the items is impractical. Balanced tree algorithms re-arrange the tree as operations are performed to maintain certain balance properties to avoid poor performance.

Skip lists are a probabilistic attempt to balanced trees. Skip lists are built by combining a random number generator. Although skip lists have had worst-case performance, no input sequence consistently produces the worst-case performance. Skip lists are also faster than balanced trees (when inserted randomly). It is very unlikely a skip list data structure will be significantly unbalanced (e.g., for a dictionary of more than 250 million entries, the chance that a skip list is unbalanced is less than 1 in 10¹⁰). The expected time is less than one in a million. Skip lists also balance properties similar to that of search trees, but by maintaining a list of pointers to the next node in the list to be searched.

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. Building skip lists is a bit more complex than building binary search trees, but the skip list algorithm makes them easier to implement and provides significant constant factor speed improvements over balanced trees. Skip lists are also space efficient. They can easily be configured to require no average of $1 + \frac{1}{2}$ pointers per element (or even less) and do not require balancing or pointer balancing to be stored with each node.

SKIP LISTS
We might need to examine every node of the list when searching for a key. We might need to move a node if the key is inserted in sorted order and all other nodes of the list also have a pointer to the node two ahead of it in the list (Figure 1a). If the list is sorted in reverse order and all other nodes of the list also have a pointer to the node two ahead of it in the list (Figure 1b), we have to examine no more than $n^2 + 1$ nodes (where n is the length of the list).

Also giving every fourth node a pointer four slots ahead (Figure 1c) requires that no more than $n^{1/2} + 2$ nodes be examined. If every 2^k node has a pointer 2^k nodes ahead (Figure 1d), the number of nodes that must be examined can be reduced to $\log_2 n + 1$. This is the same as the number of levels. This data structure could be used for fast searching, but insertion and deletion would be impractical.

It is not possible to maintain a skip list in a balanced form.

If every 2^k node has a pointer 2^k 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, etc. This is 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 k^{th} forward pointer is at least 2^k nodes away from the node. Inserting the next node of level i or higher. Insertions or deletions would require only local modifications: the level of a node, chosen randomly, would not change its position in the list. Some arrangements of levels would give poor execution times, but we will see that such arrangements are rare. Because these data structures are probabilistic, it is not possible to say that skip over intermediate nodes. I named them skip lists.

SKIP LIST ALGORITHMS
This section gives algorithms for search, insert and delete elements in a skip list symbol table. The *Search* operation returns the contents of the value associated with the specified key or *false* if the key is not present. The *Insert* operation inserts a new node with the specified key and value into the list if it had not already been present. The *Delete* operation deletes the specified key. It is easy to support additional operations such as *SearchMin*, *SearchMax*, and *SearchRange*.

Each element is represented by a node, the level of which is chosen randomly when the node is inserted without regard for the current maximum level of the list.

A *Level* node has f *forward* pointers, indexed 1 through f . We do not need to store the level of a node in the node. Levels are explicitly maintained by the *Header* node. The *Header* node of a list is the maximum level currently in the list (or 0 if the list is empty). The *Header* node has *forward* pointers to levels one through f . The *forward* pointers to higher levels are at levels higher than the current maximum level of the list point to NIL.



SKIP LISTS

skip list is a probabilistic data structure because when building the list, the entry is inserted in each level with a certain probability

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
 - advantage of random: store less data than b+ tree,
b+ tree need to store all guideposts in the intermediate nodes

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 list is like rotated b+ tree

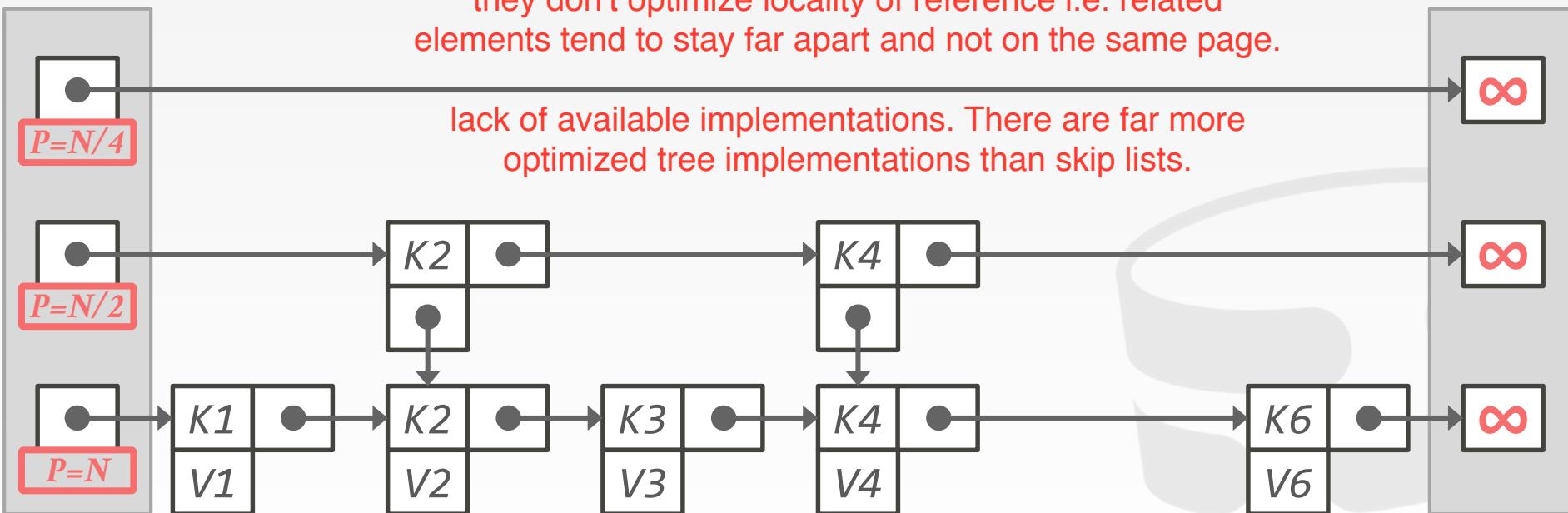
SKIP LISTS: EXAMPLE

Levels

disadvantages: skip lists are not cache friendly because they don't optimize locality of reference i.e. related elements tend to stay far apart and not on the same page.

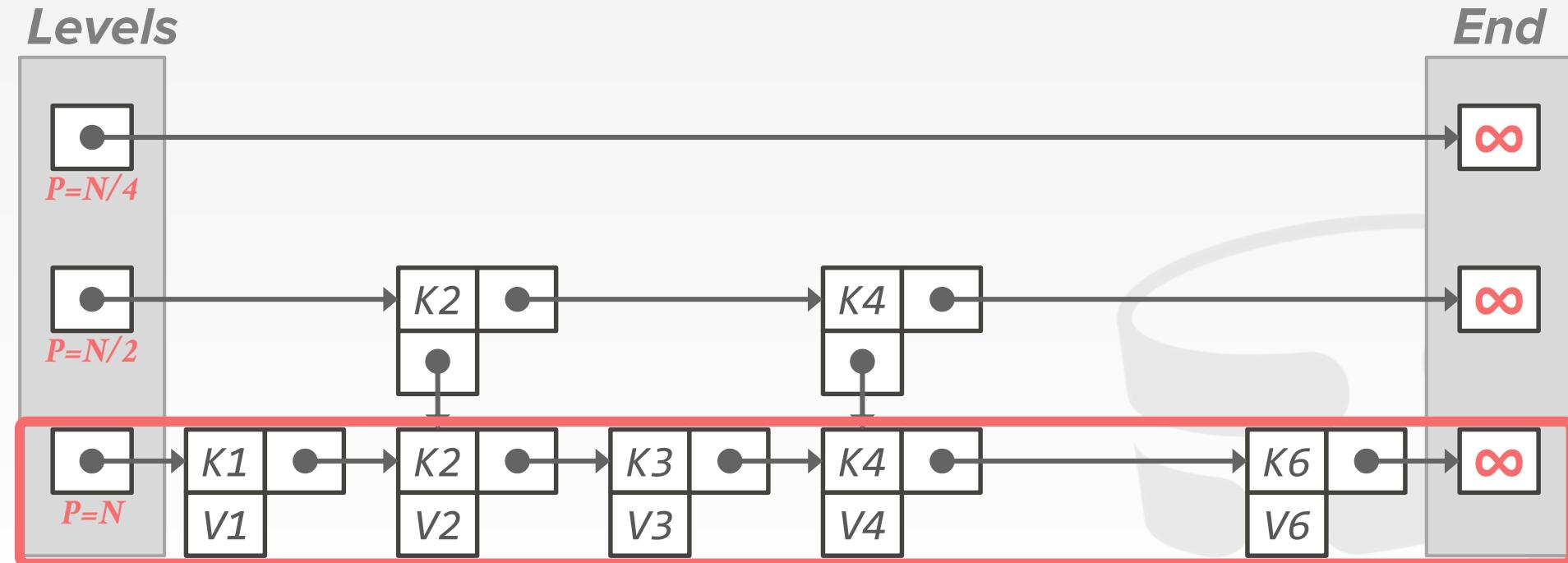
lack of available implementations. There are far more optimized tree implementations than skip lists.

End



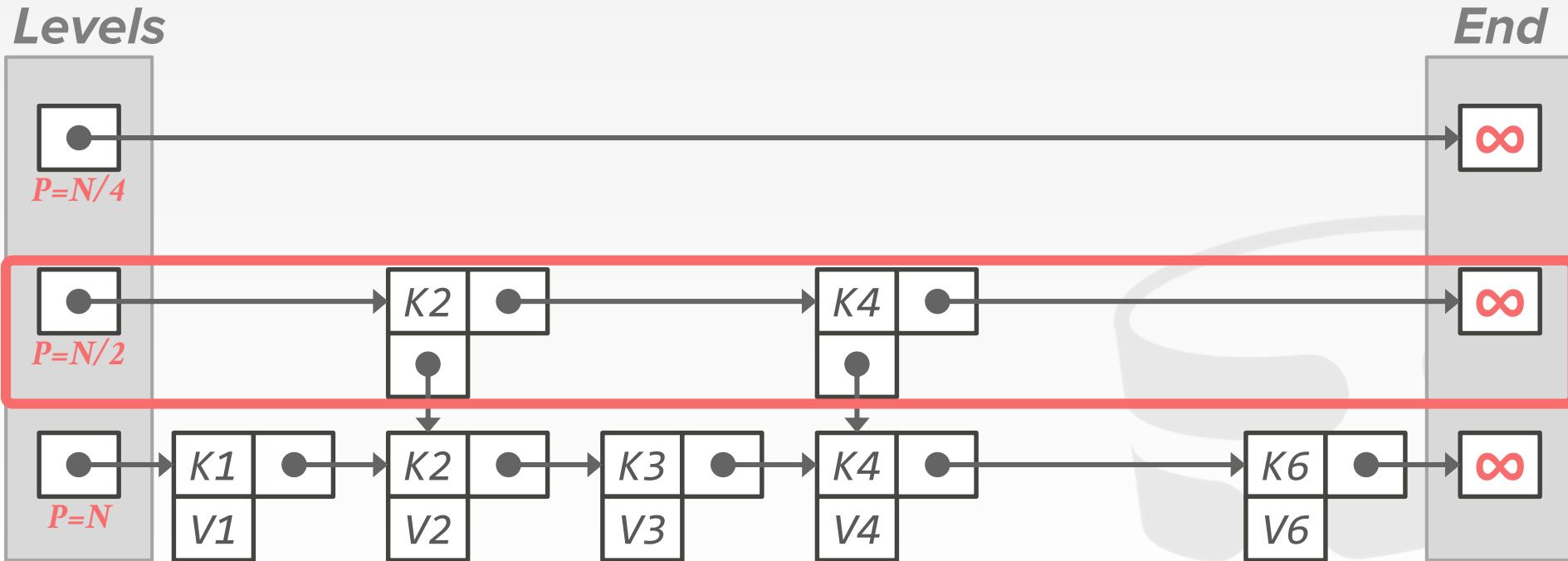
SKIP LISTS: EXAMPLE

Levels



SKIP LISTS: EXAMPLE

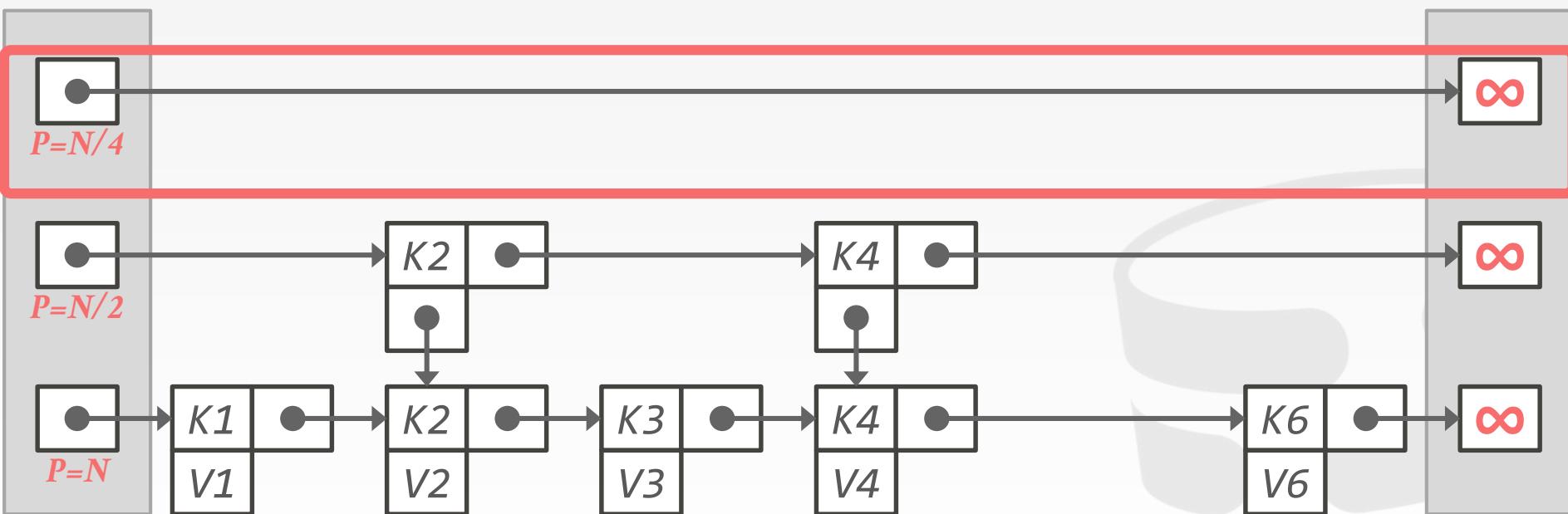
Levels



SKIP LISTS: EXAMPLE

Levels

End

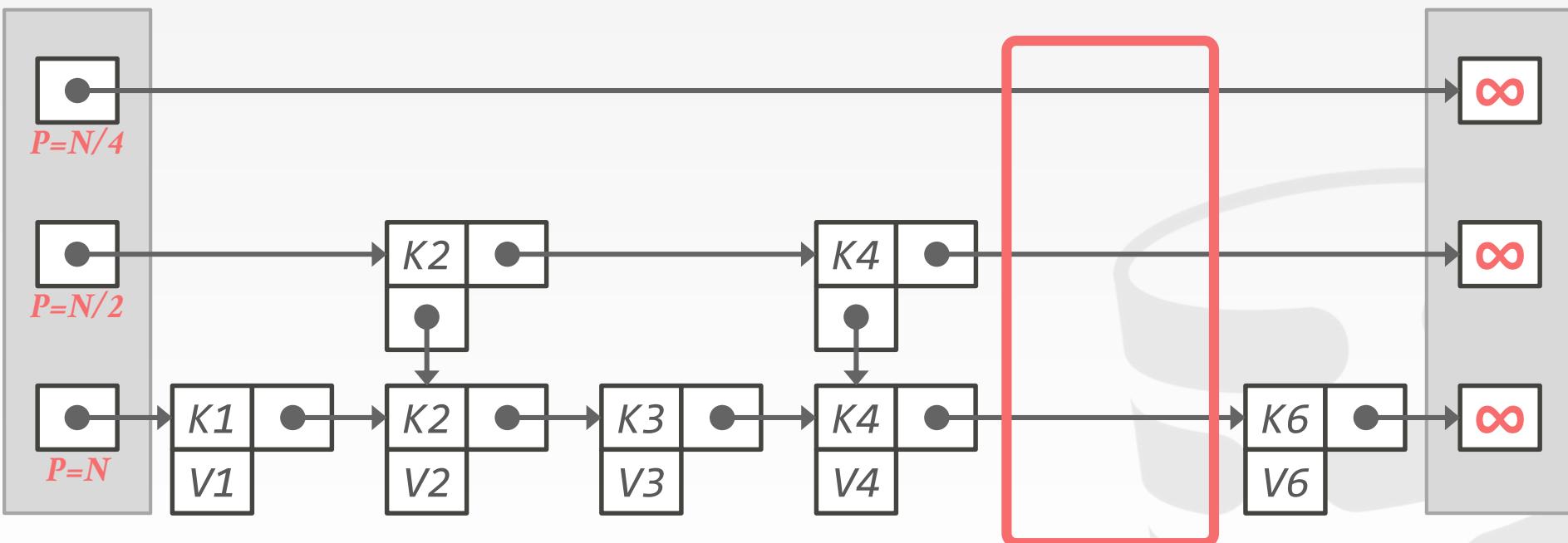


SKIP LISTS: INSERT

Insert K5

Levels

End

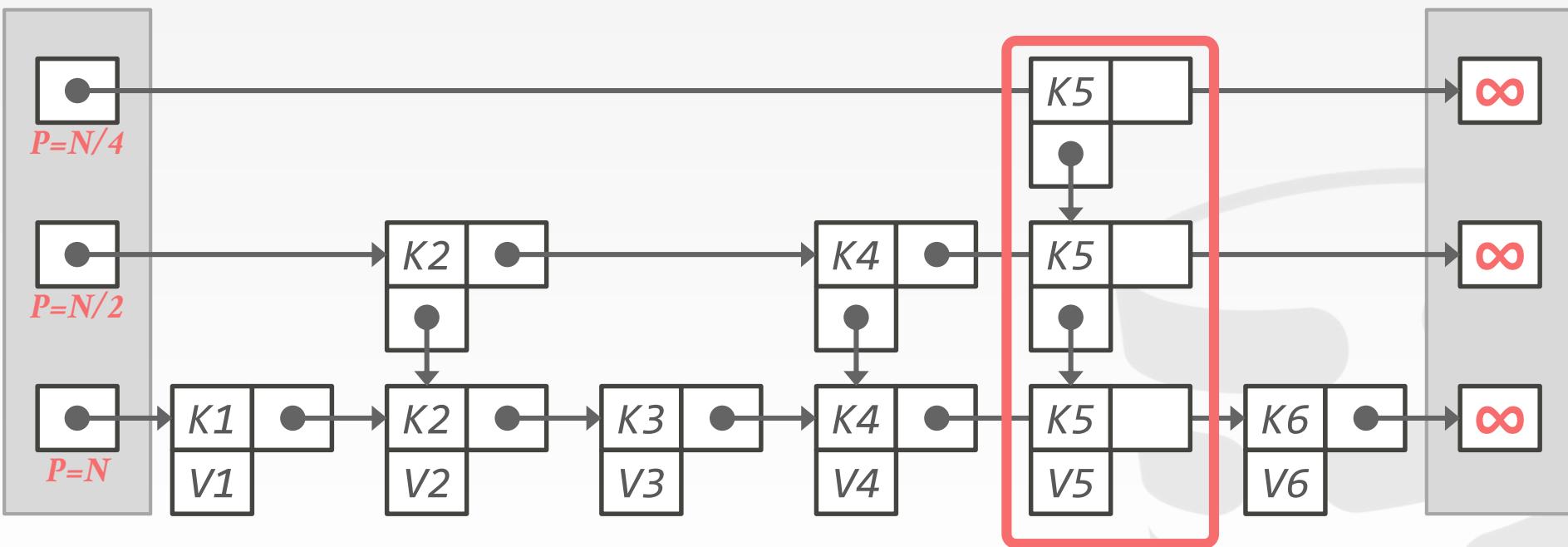


SKIP LISTS: INSERT

Insert K5

Levels

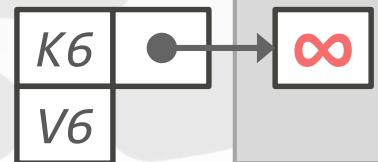
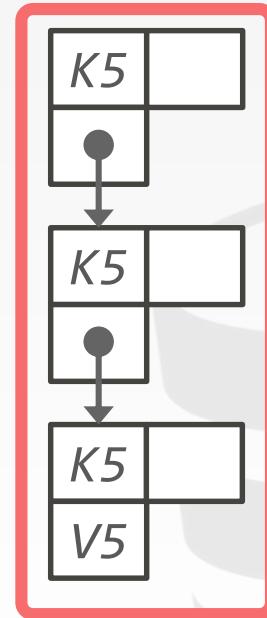
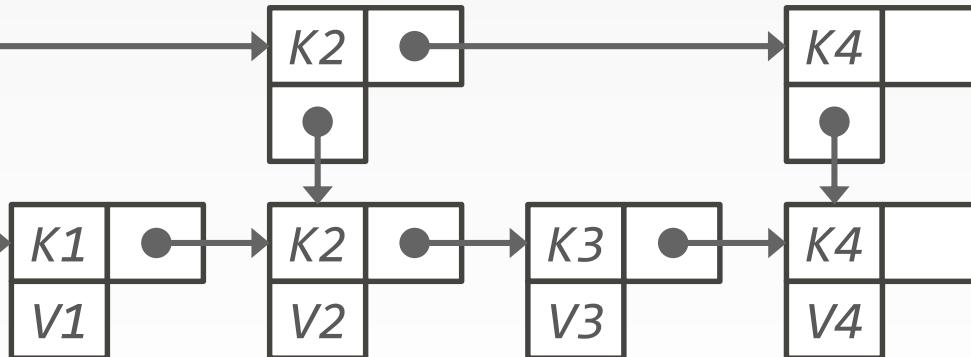
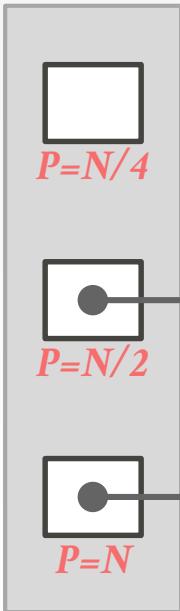
End



SKIP LISTS: INSERT

Insert K5

Levels



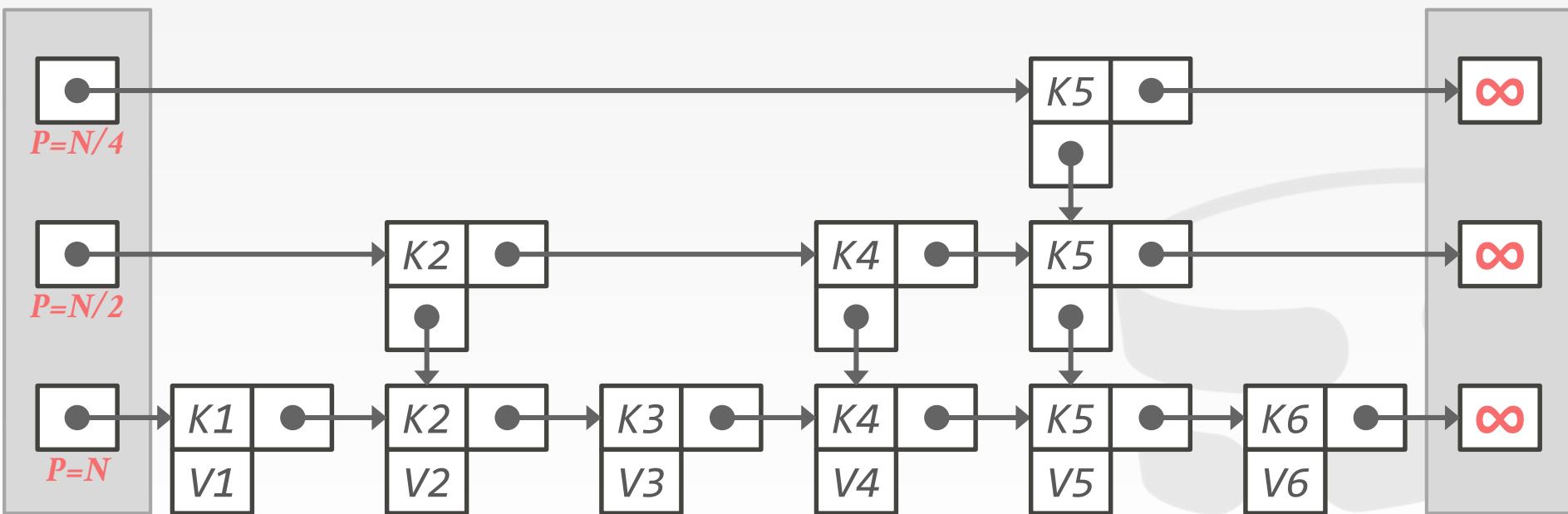
End

SKIP LISTS: INSERT

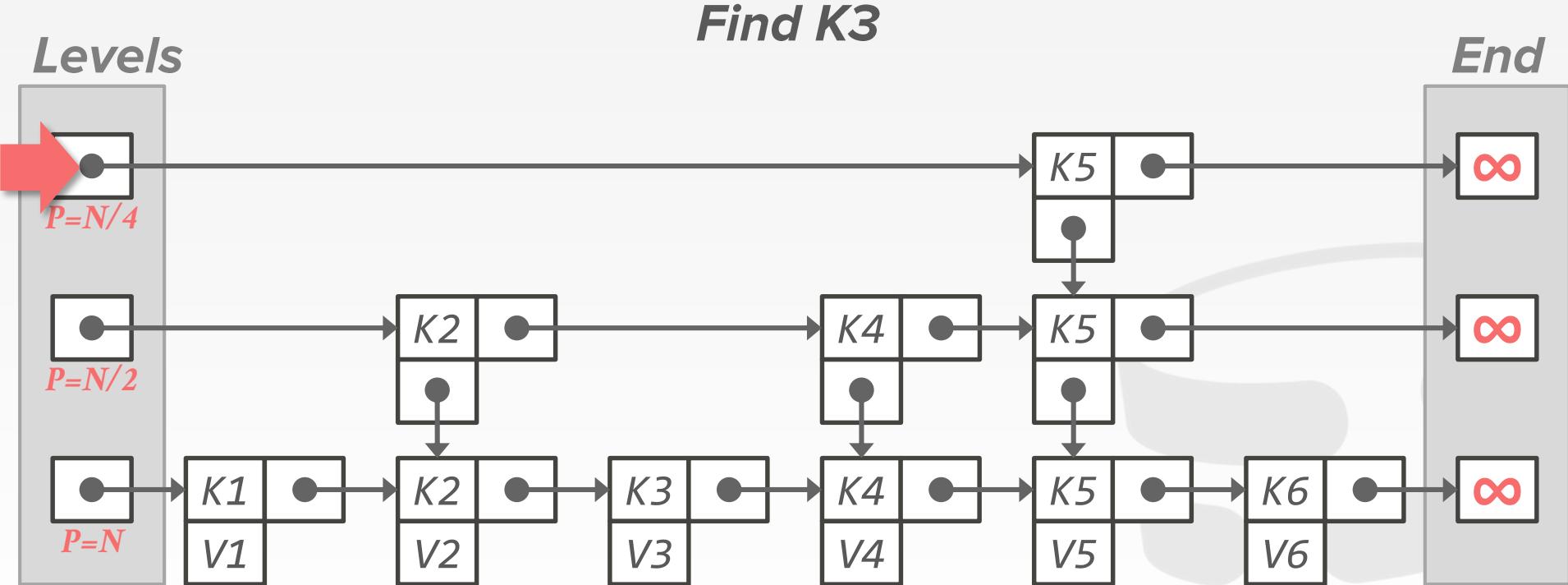
Insert K5

Levels

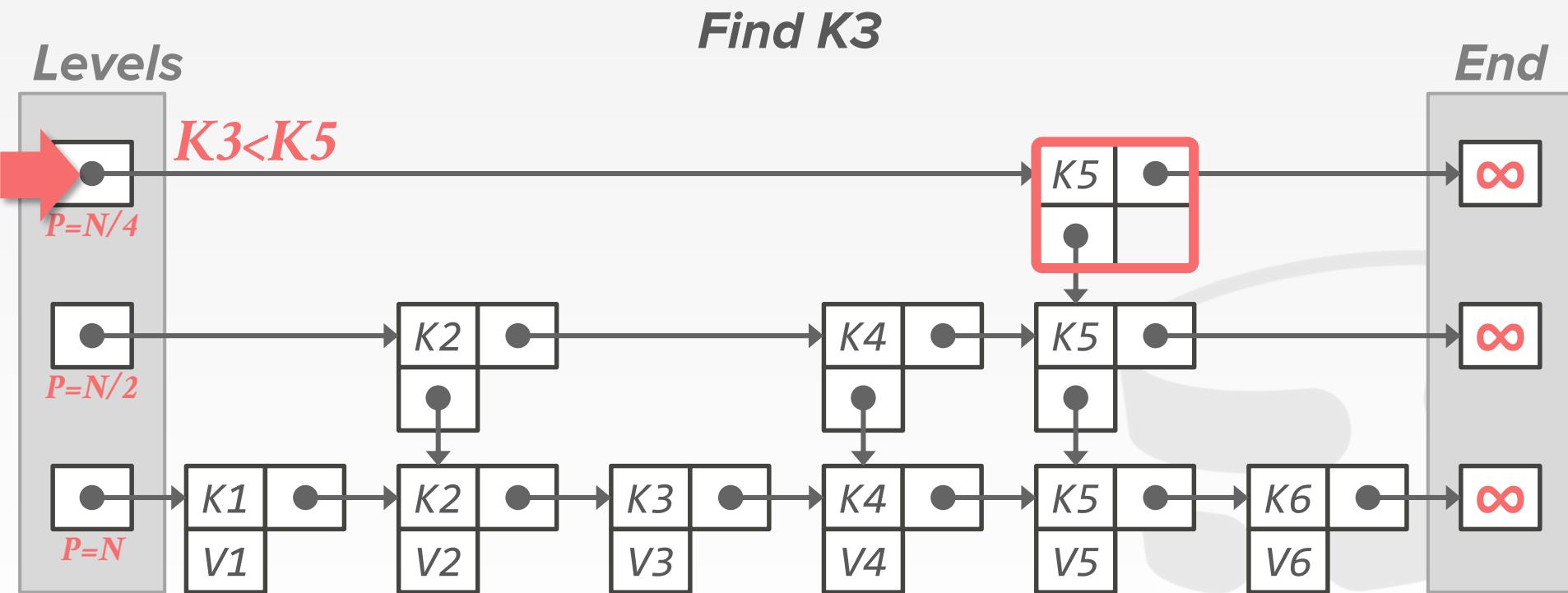
End



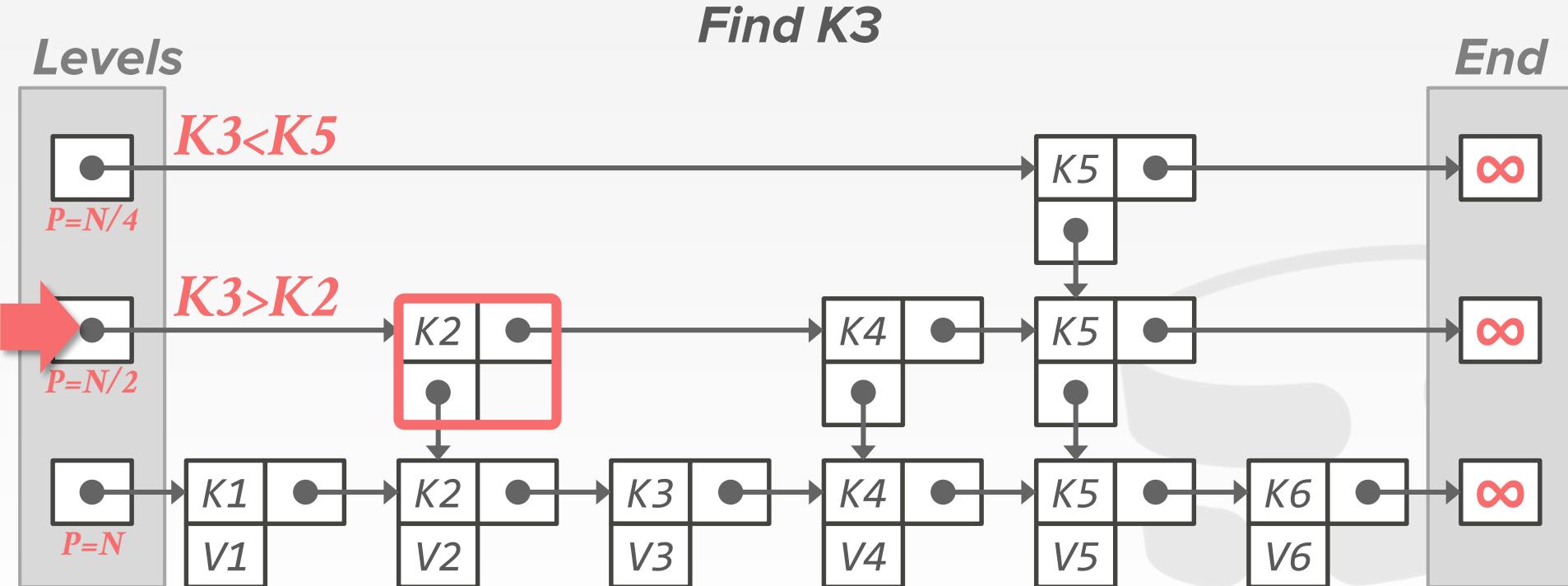
SKIP LISTS: SEARCH



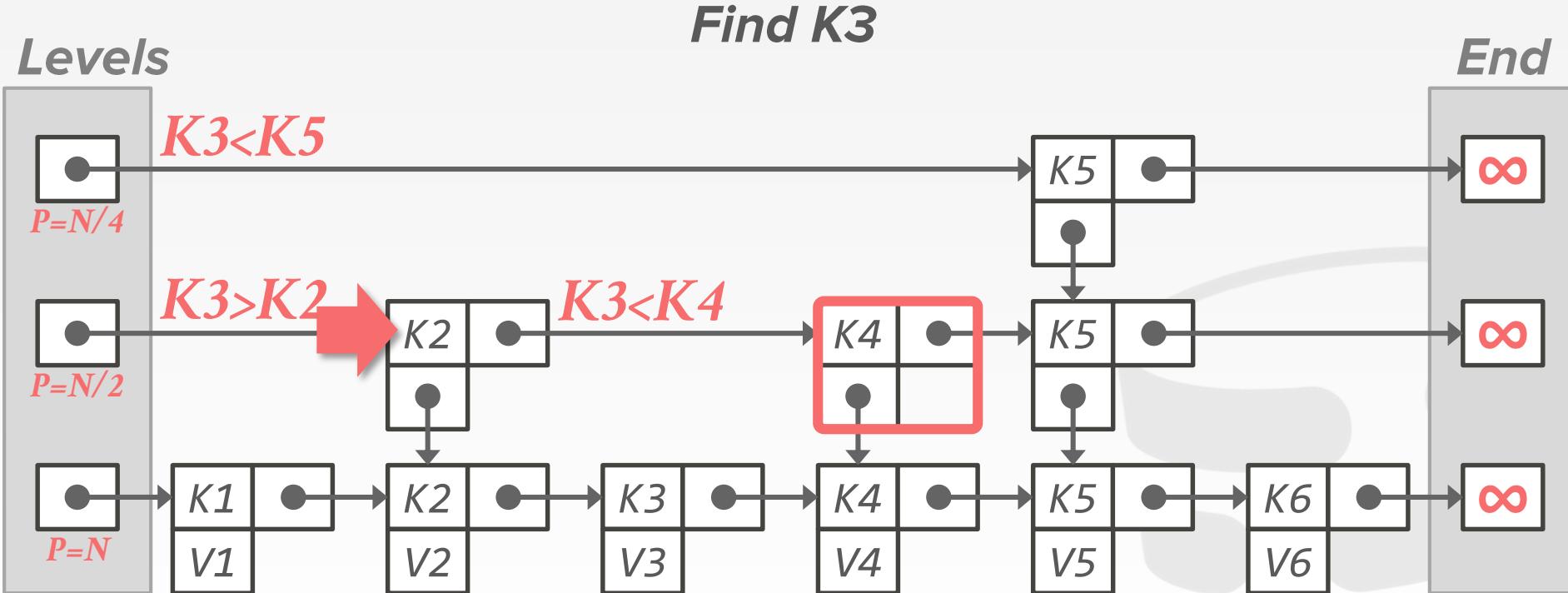
SKIP LISTS: SEARCH



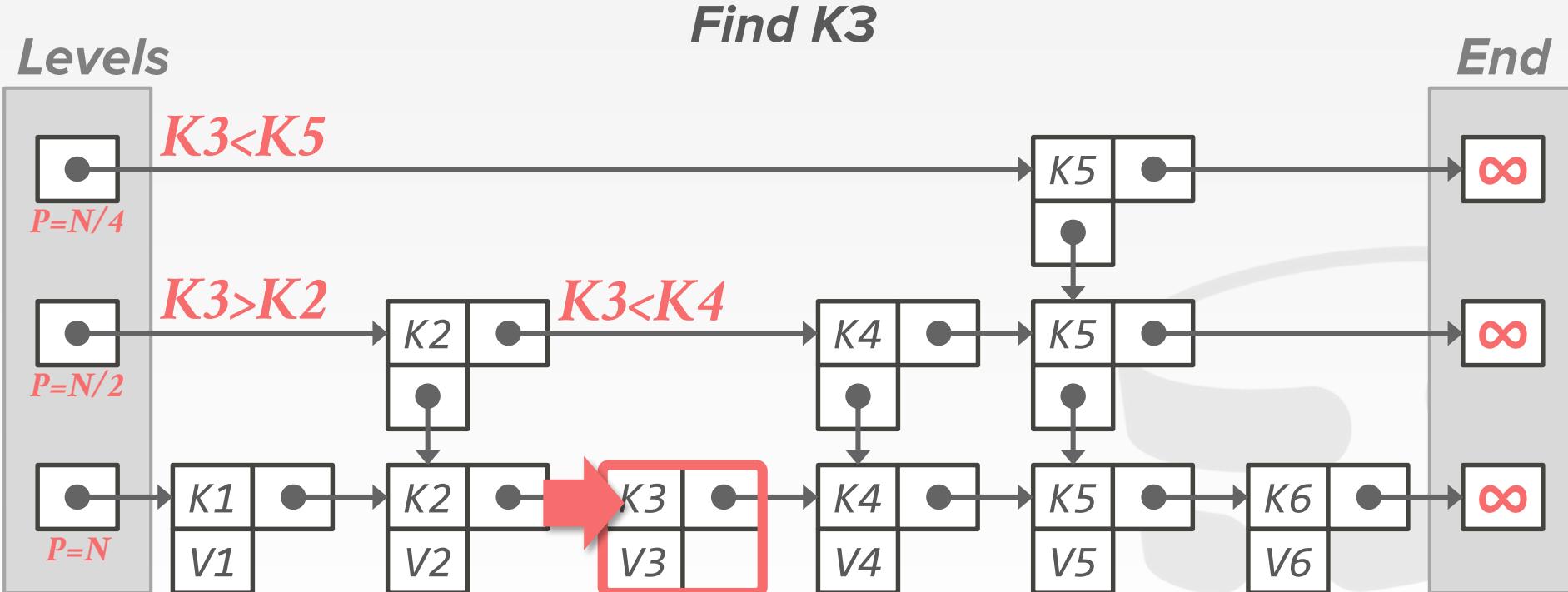
SKIP LISTS: SEARCH



SKIP LISTS: SEARCH



SKIP LISTS: SEARCH



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.

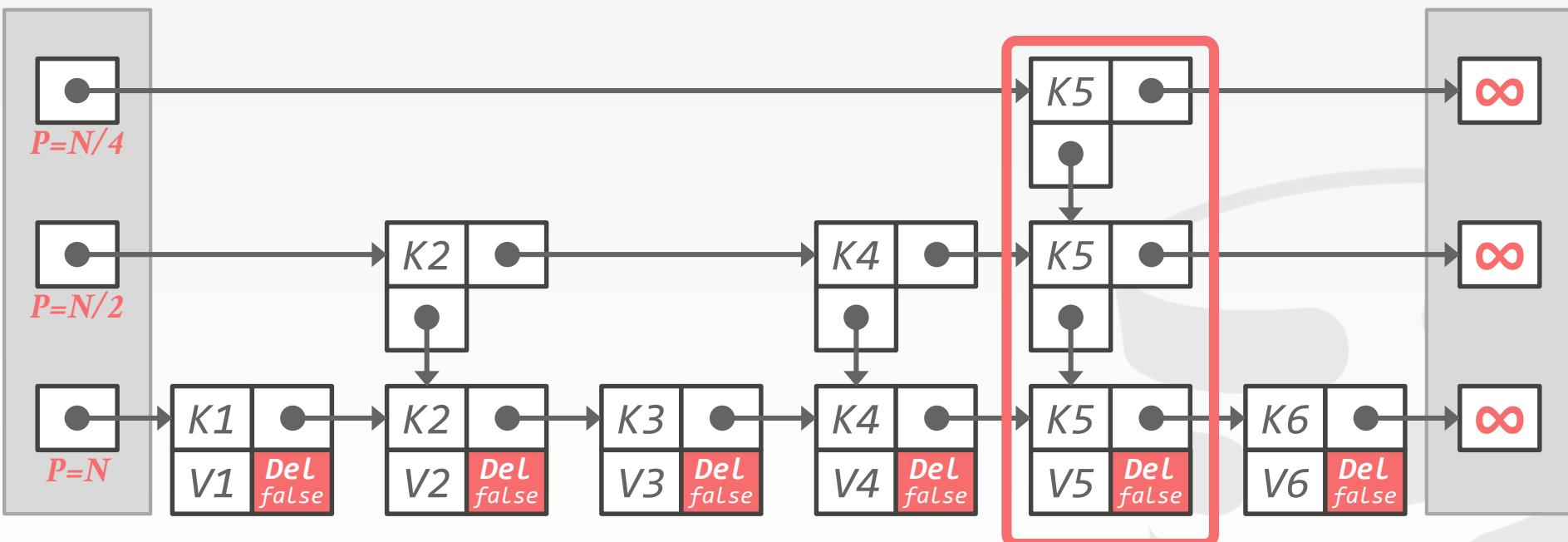


SKIP LISTS: DELETE

Delete K5

Levels

End

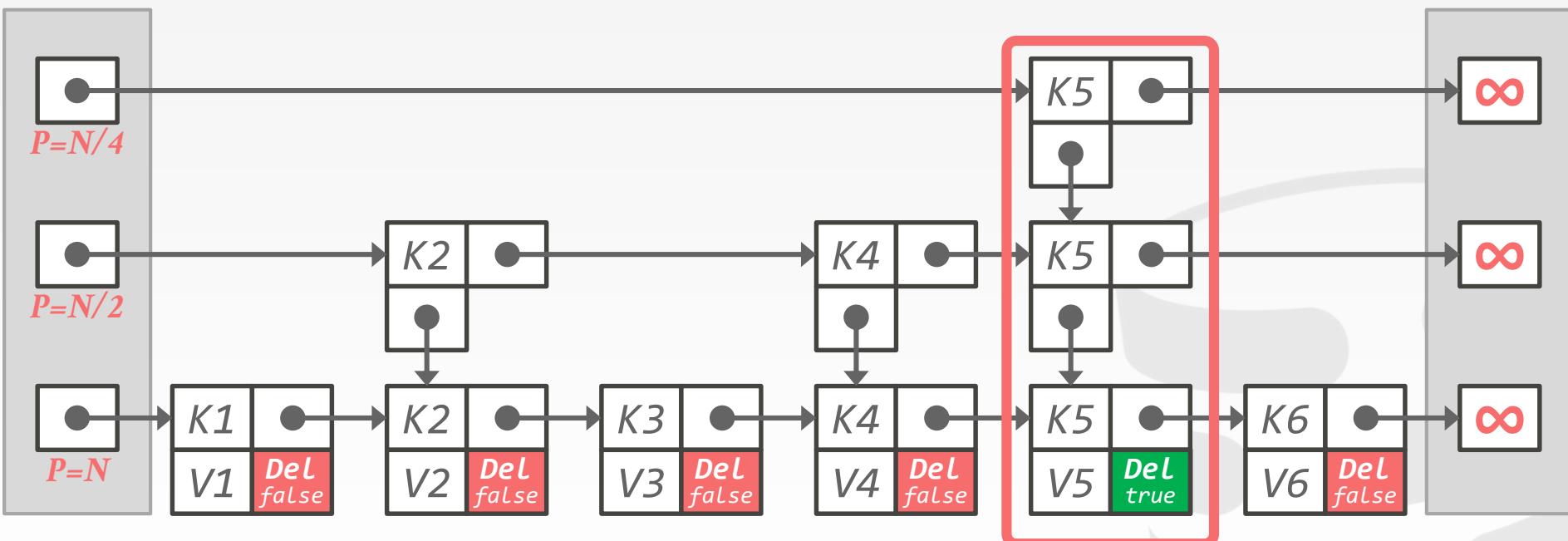


SKIP LISTS: DELETE

Delete K5

Levels

End

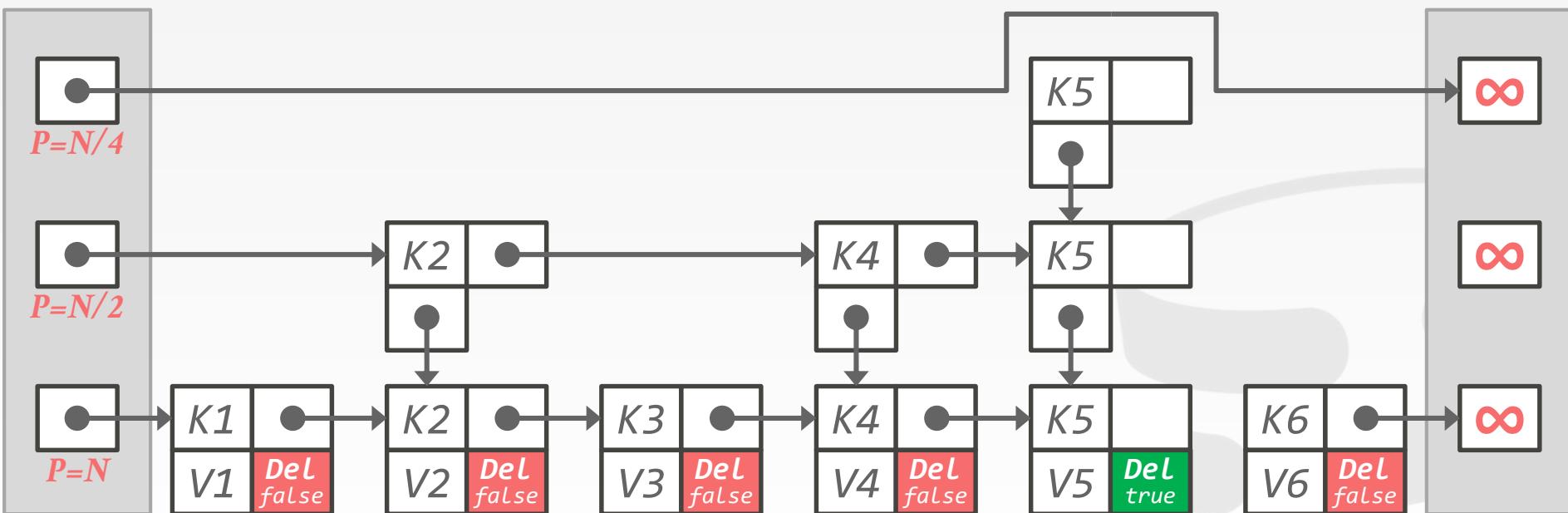


SKIP LISTS: DELETE

Delete K5

Levels

End

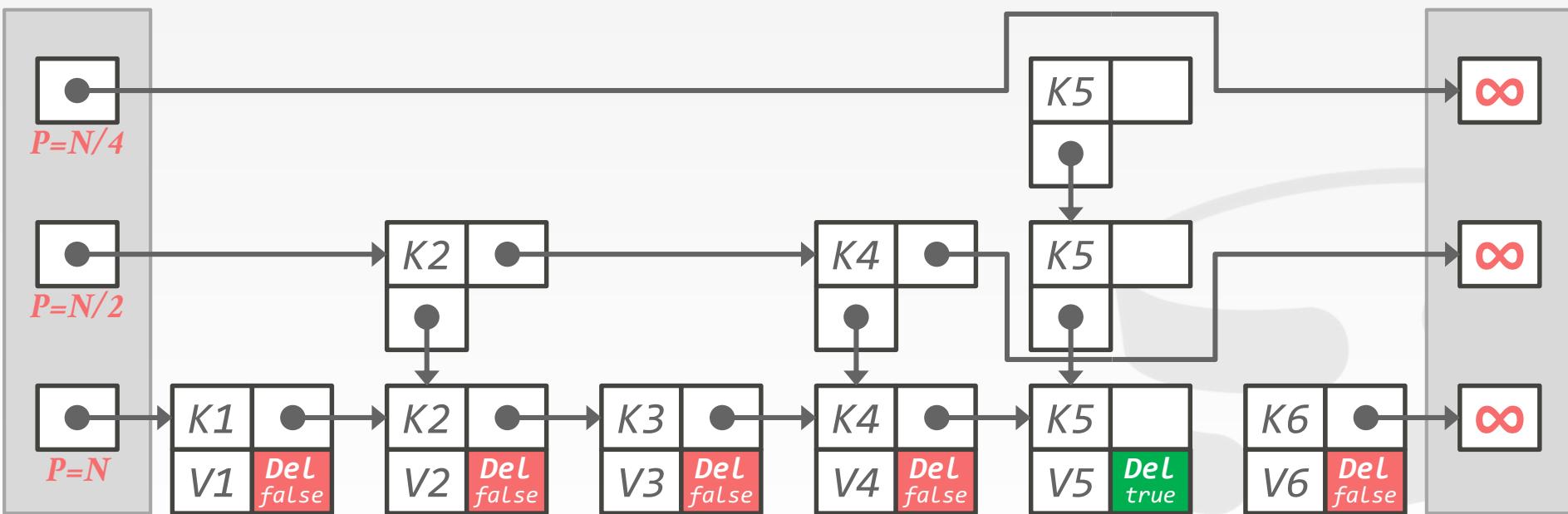


SKIP LISTS: DELETE

Delete K5

Levels

End

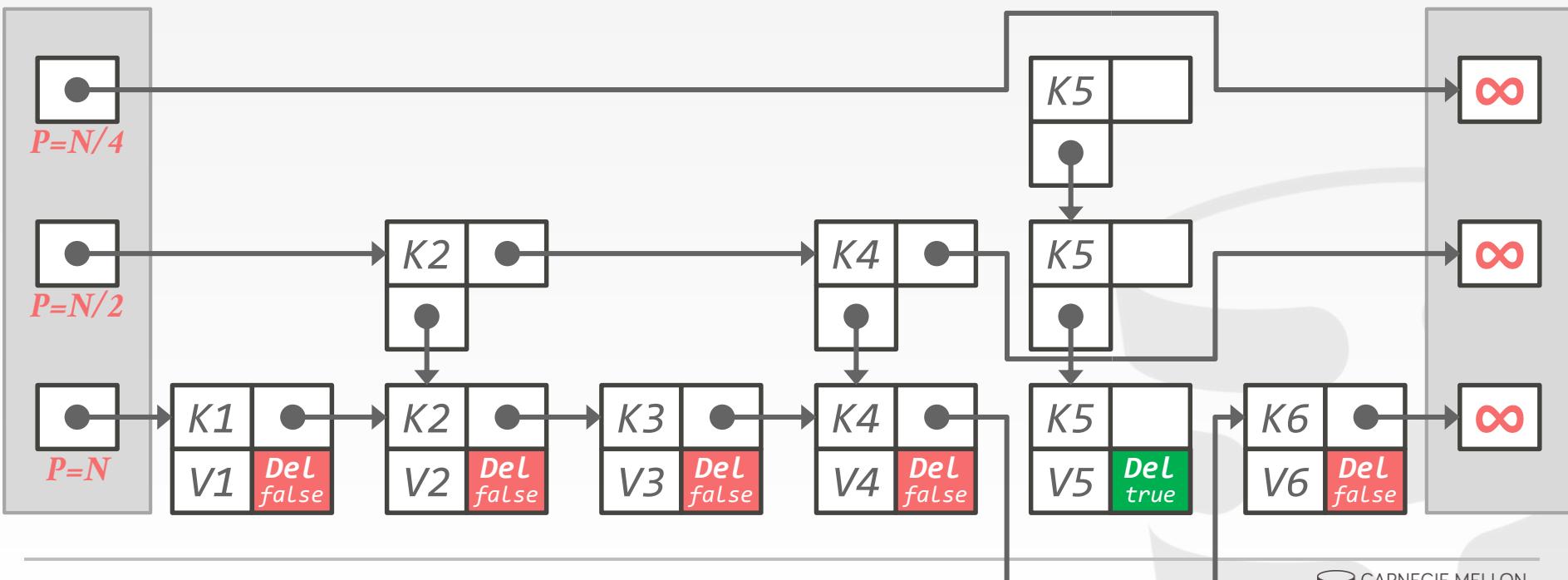


SKIP LISTS: DELETE

Delete K5

Levels

End

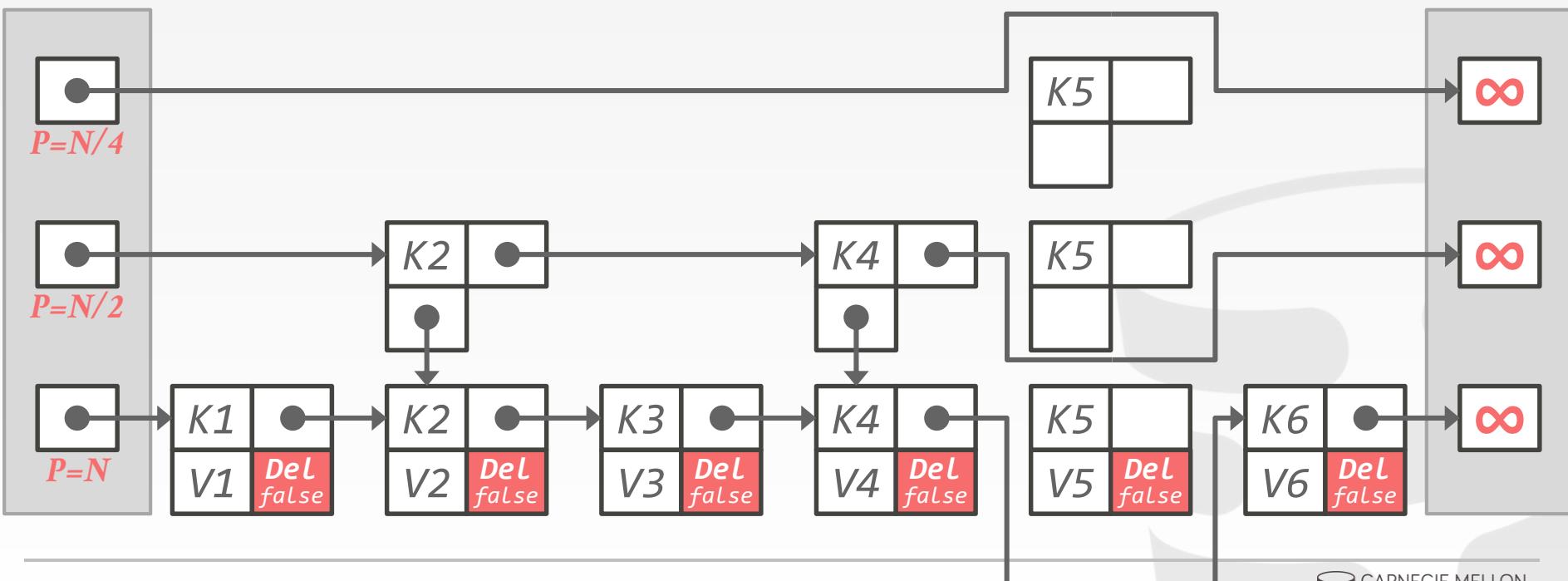


SKIP LISTS: DELETE

Delete K5

Levels

End



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.



SKIP LISTS: DISADVANTAGES

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

Invoking random number generator multiple times per insert is slow.

Reverse search is non-trivial.



RADIX TREE

for skip list and b+ tree, whole comparison between keys is inevitable

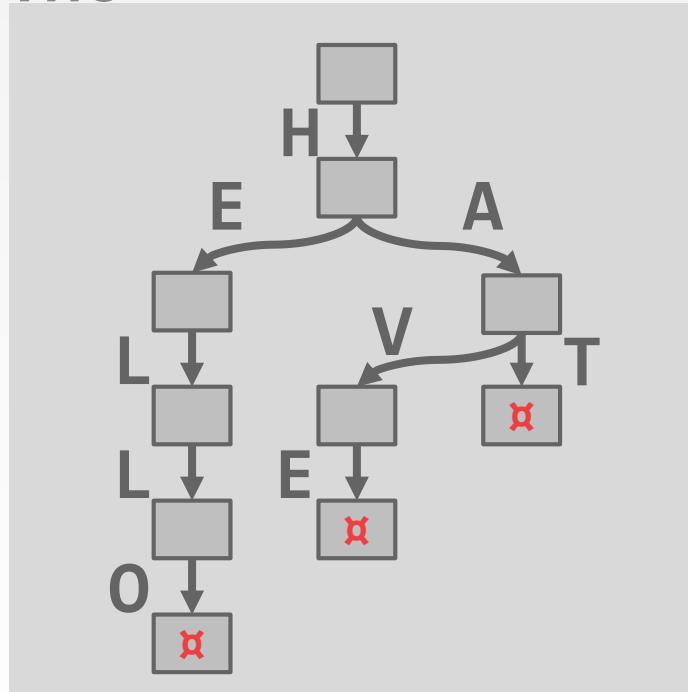
Uses digital representation of keys to examine prefixes one-by-one instead of comparing entire key.

- 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

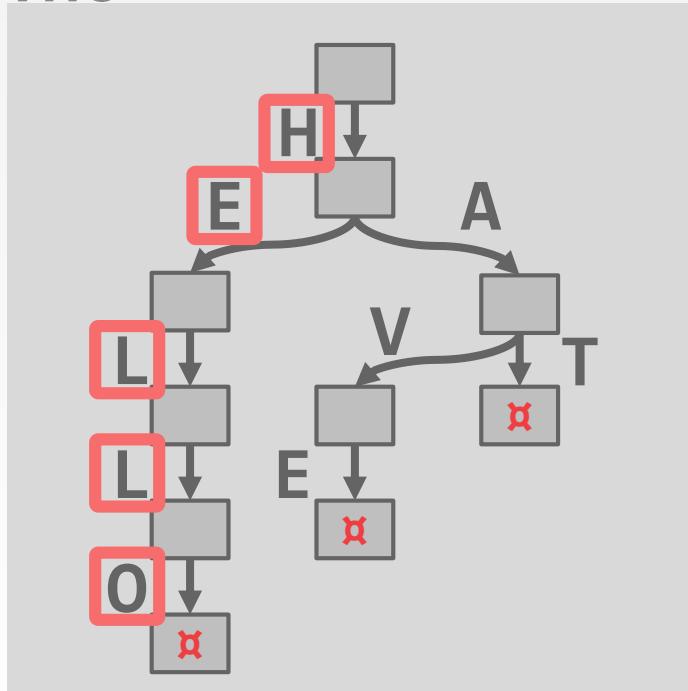
Trie



Keys: HELLO, HAT, HAVE

TRIE VS. RADIX TREE

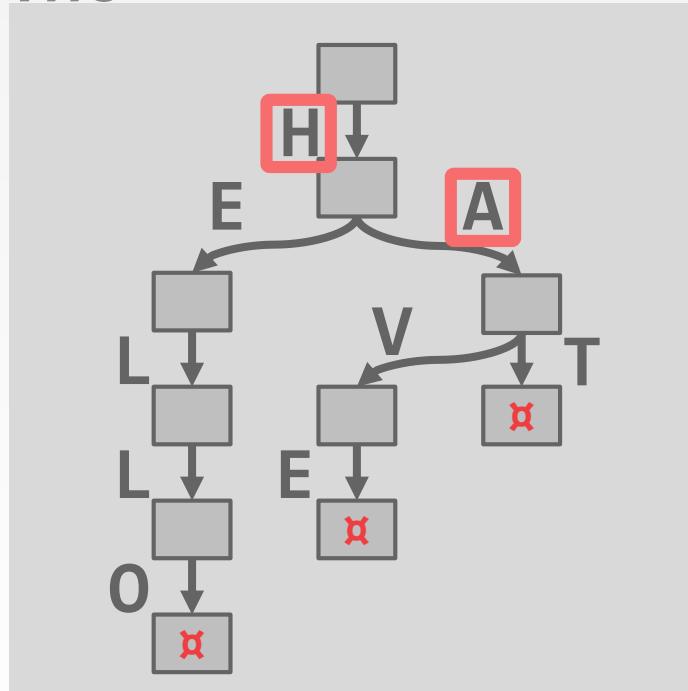
Trie



Keys: **HELLO**, HAT, HAVE

TRIE VS. RADIX TREE

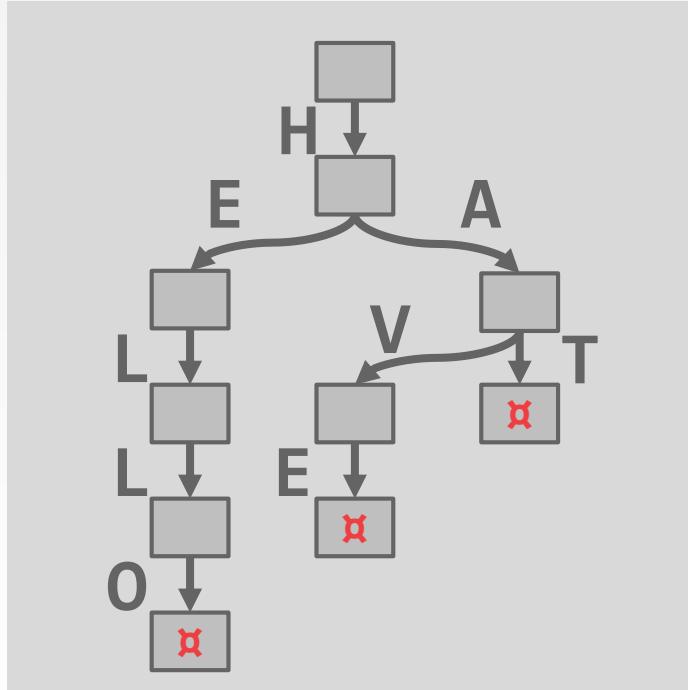
Trie



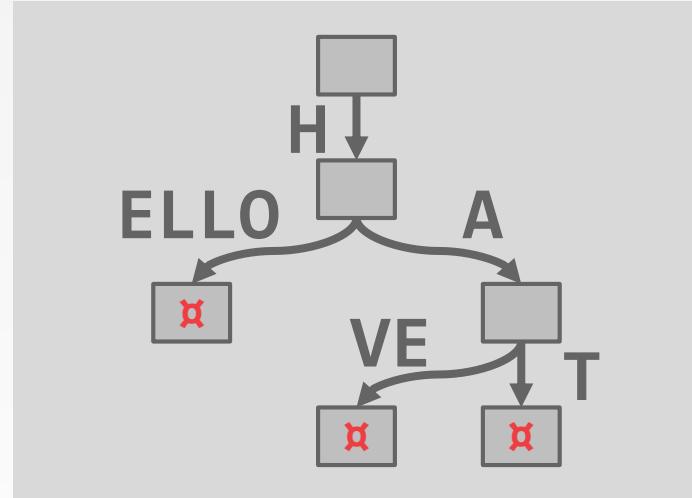
Keys: HELLO, **HAT, HAVE**

TRIE VS. RADIX TREE

Trie



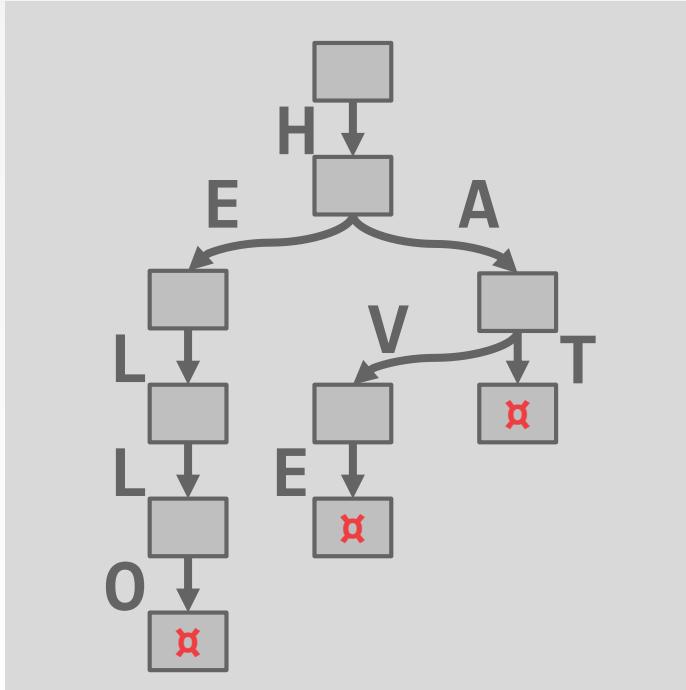
Radix Tree



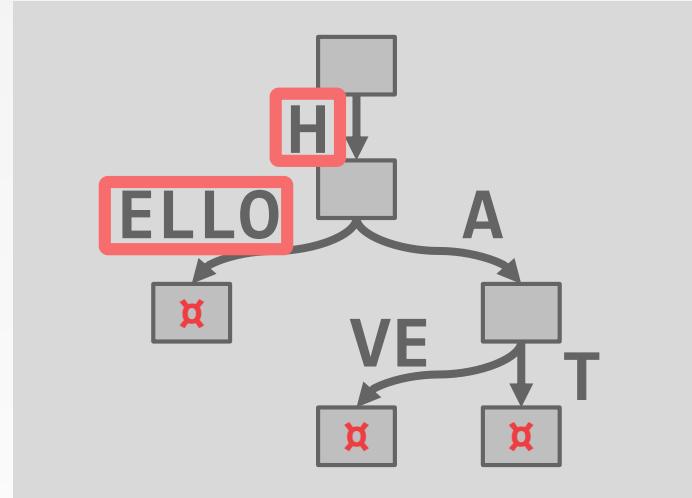
Keys: HELLO, HAT, HAVE

TRIE VS. RADIX TREE

Trie

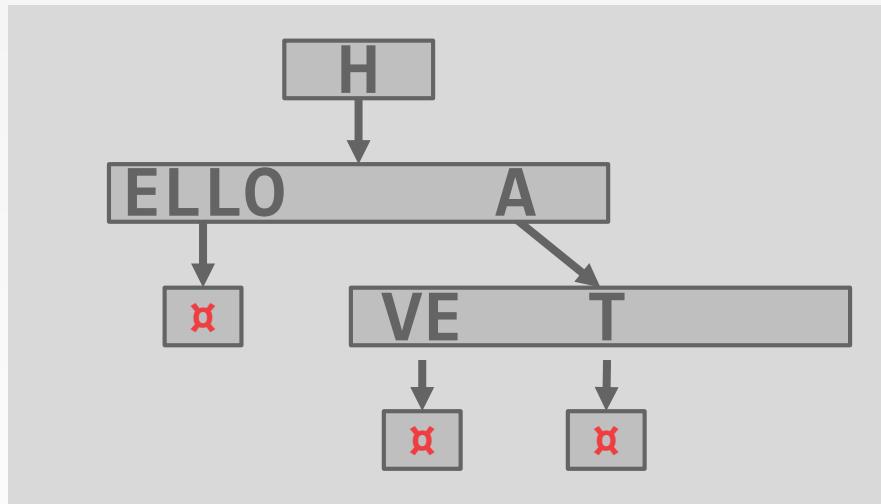


Radix Tree

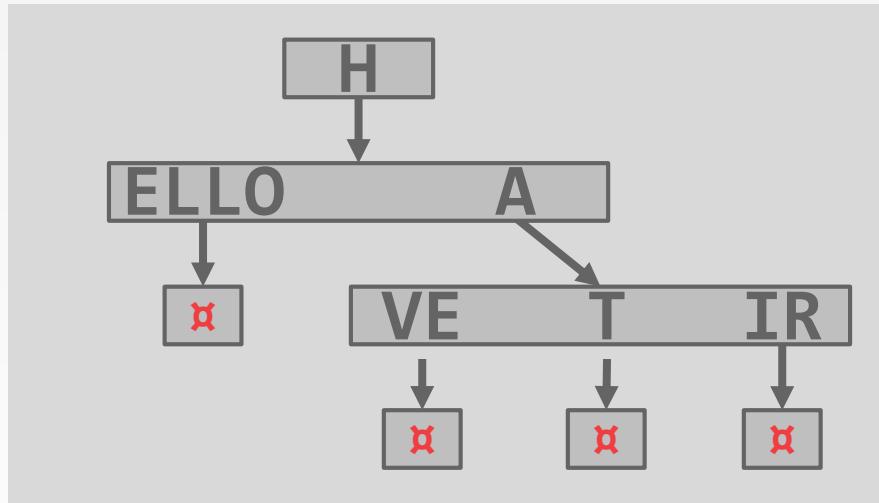


Keys: **HELLO**, HAT, HAVE

RADIX TREE: MODIFICATIONS



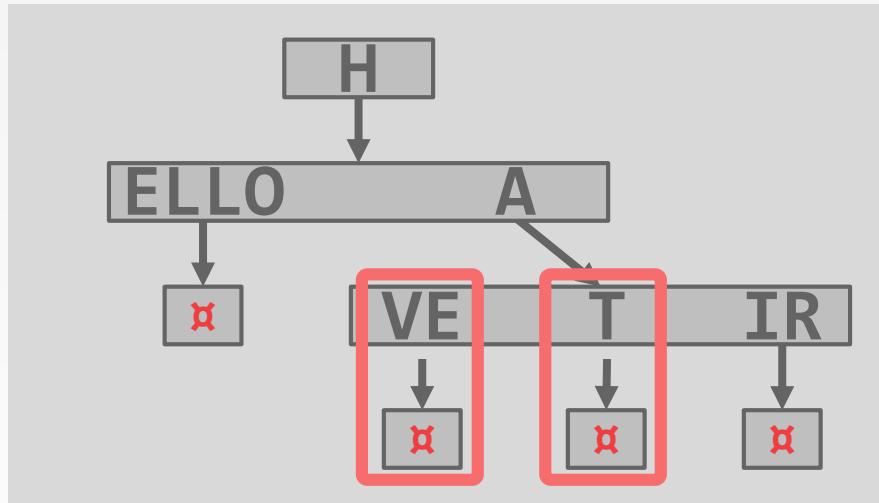
RADIX TREE: MODIFICATIONS



Insert HAIR



RADIX TREE: MODIFICATIONS

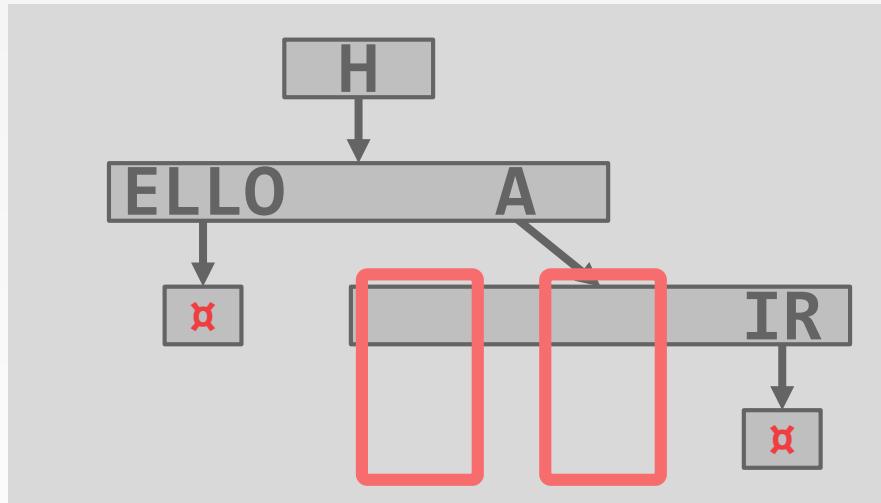


Insert HAIR

Delete HAT, HAVE



RADIX TREE: MODIFICATIONS

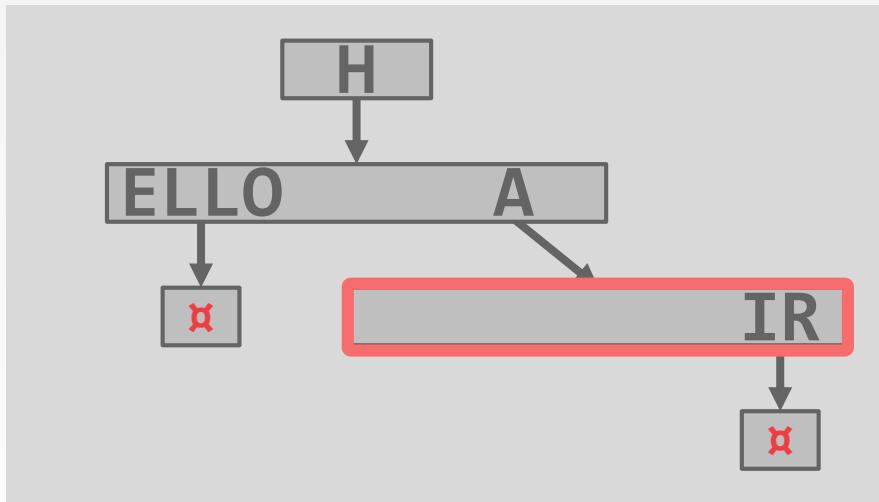


Insert HAIR

Delete HAT, HAVE



RADIX TREE: MODIFICATIONS

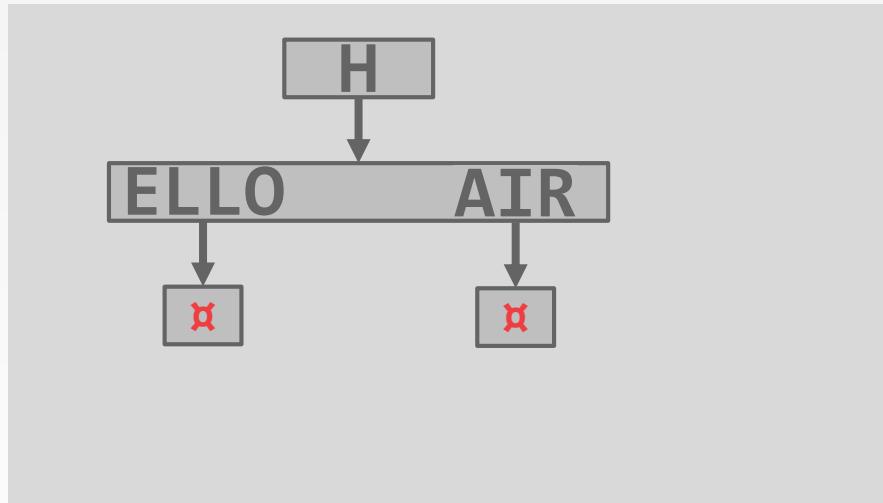


Insert HAIR

Delete HAT, HAVE

merge up with parent node

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 convert little endian to big endian before saving to DBMS
- **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

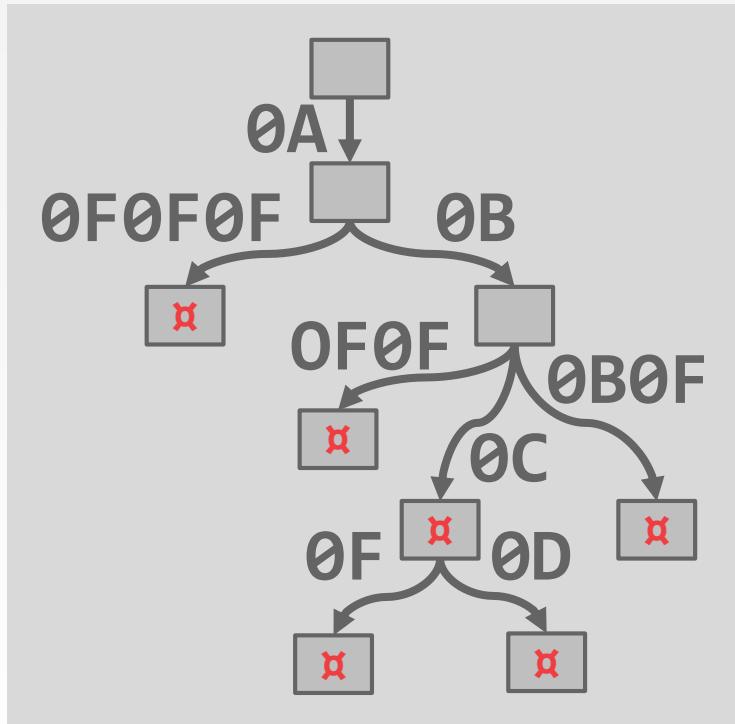


Big Endian



Little Endian

RADIX TREE: BINARY COMPARABLE KEYS



Int Key: 168496141

Hex Key: 0A 0B 0C 0D

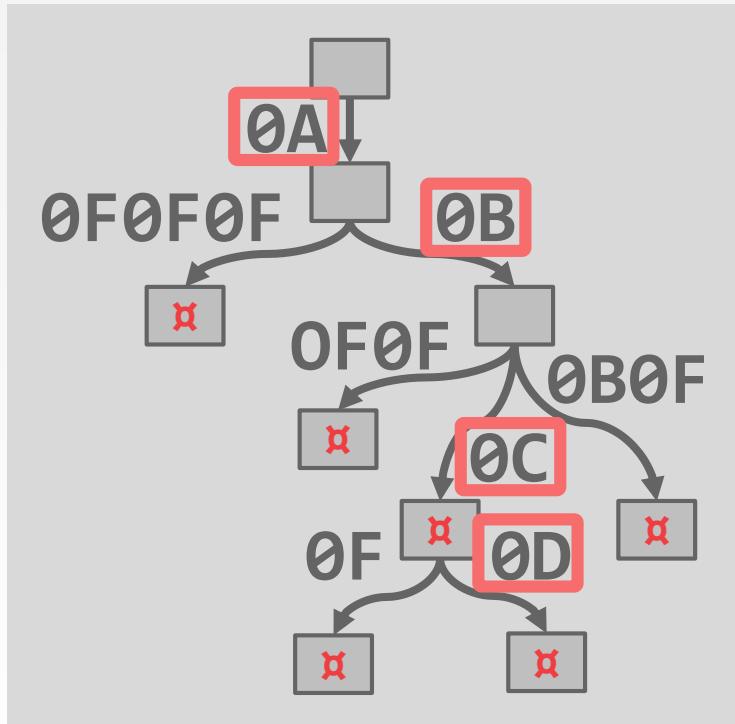


Big Endian



Little Endian

RADIX TREE: BINARY COMPARABLE KEYS



Int Key: 168496141

Hex Key: 0A 0B 0C 0D



BigEndian

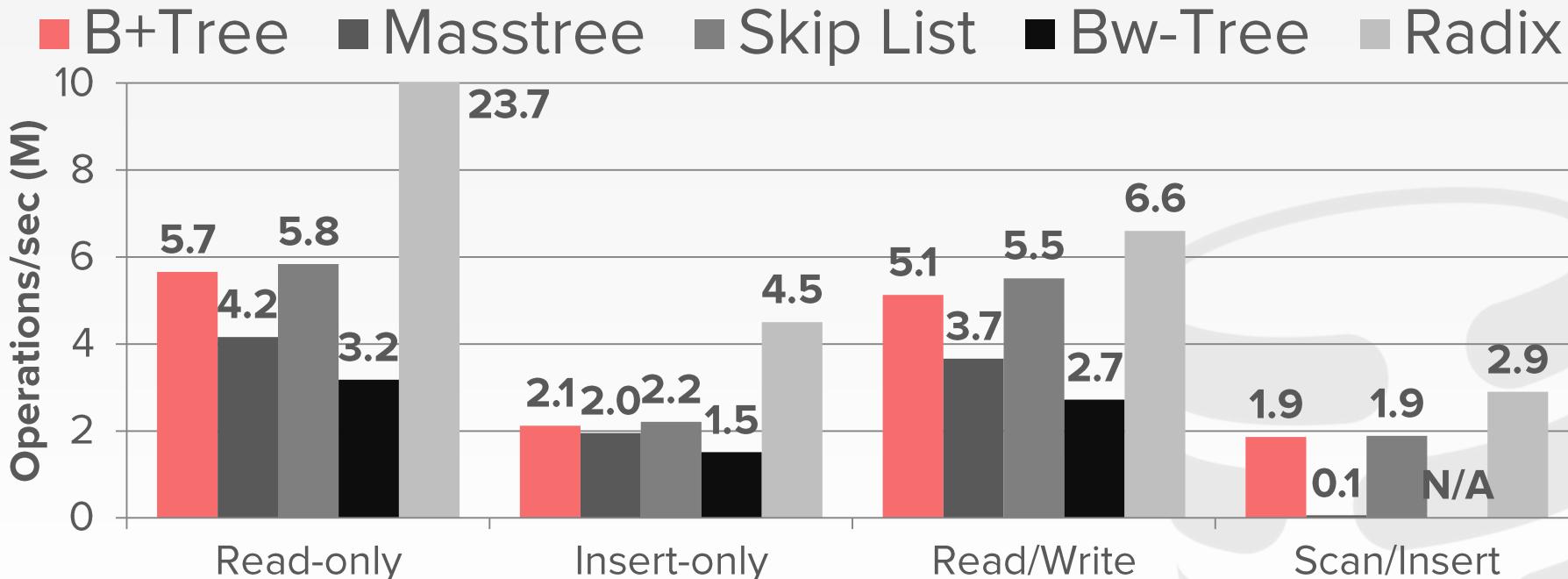


LittleEndian

SINGLE-THREADED PERFORMANCE

in memory. radix is fast because
the comparison is just bit-wise

Data Set: 30m Random 64-bit Integers



SELECTION CONDITIONS

The DBMS can use a B+Tree index if the query provides all of the attributes in a prefix of the search key.

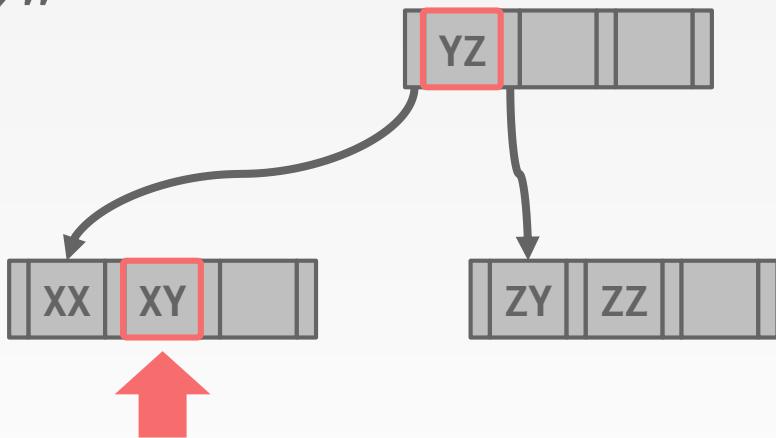
→ Index on $\langle a, b, c \rangle$ matches $(a=5 \text{ AND } b=3)$,
but not $b=3$.

For Hash index, we must have all attributes in search key.



B+TREE PREFIX SEARCH

Find "XY"

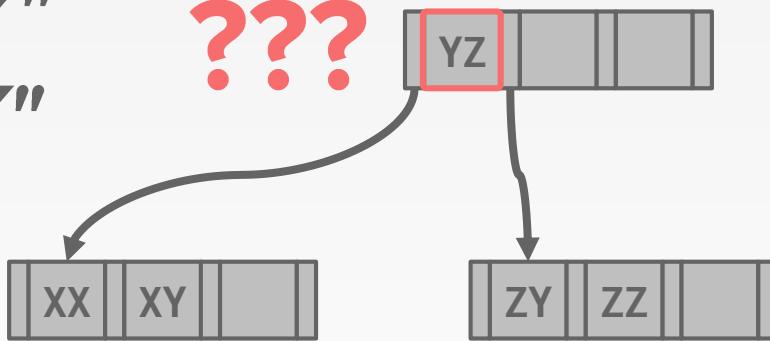


B+TREE PREFIX SEARCH

Find "XY"

???

Find "_Y"



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.

```
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.

```
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.

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

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

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.

```
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)
```

```
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'
```

CONCLUSION

The venerable B+Tree is always a good choice for your DBMS.

Skip Lists and Radix Trees have some interesting properties.

We will cover lock free data structures in 15-721.



NEXT CLASS

Query Processing

→ How to use what we've talked about
so far to actually execute queries!

