Carnegie Mellon University

Order-Preserving Trees

Lecture #09

Database Systems
15-445/15-645
Fall 2017

Andy Pavlo
Computer Science Dept.
Carnegie Mellon Univ.

ADMINISTRIVIA

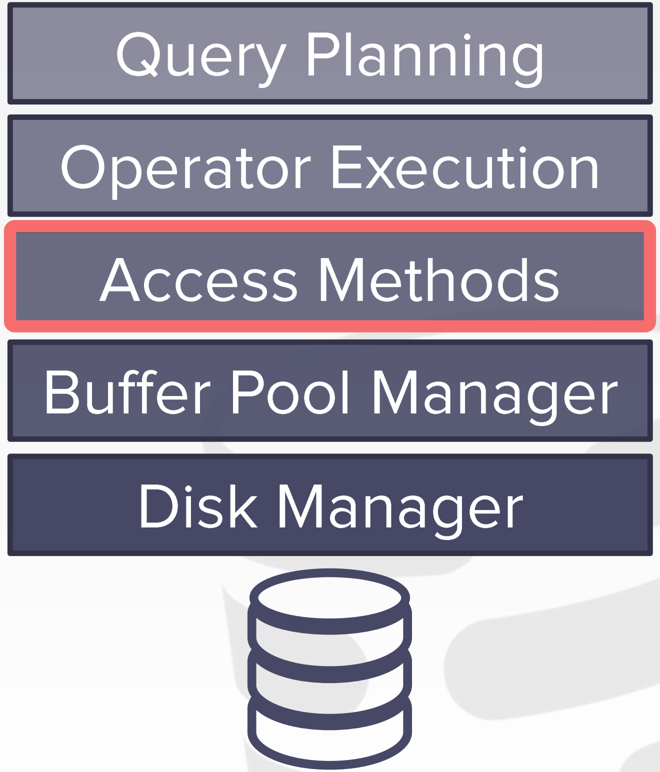
Project #1 is due Monday October 2nd @ 11:59pm

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



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.

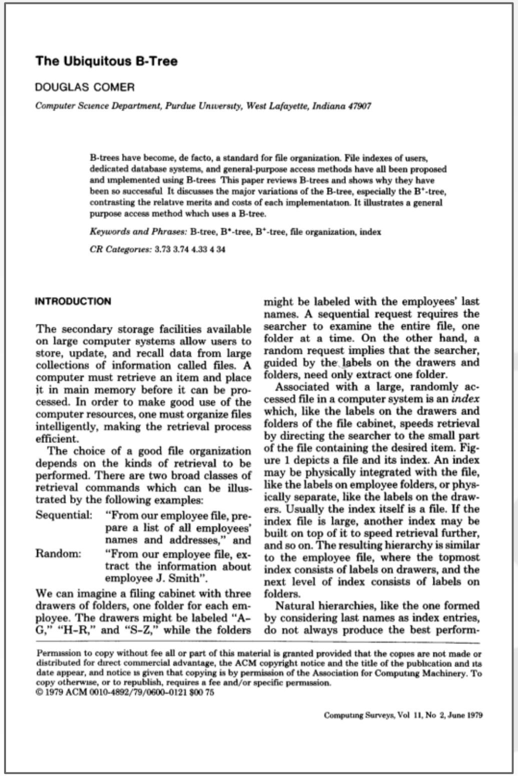
→ **B-Tree**
→ **B+Tree**
→ **B^{link}-Tree**
→ **B*Tree**

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, dedicated database systems, and general-purpose access methods 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–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 extract one folder.

Associated with 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, like the labels on employee folders, or physically separate, like 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 perform-

Computing Surveys, Vol 11, No 2, June 1979

B+TREE: PROPERTIES

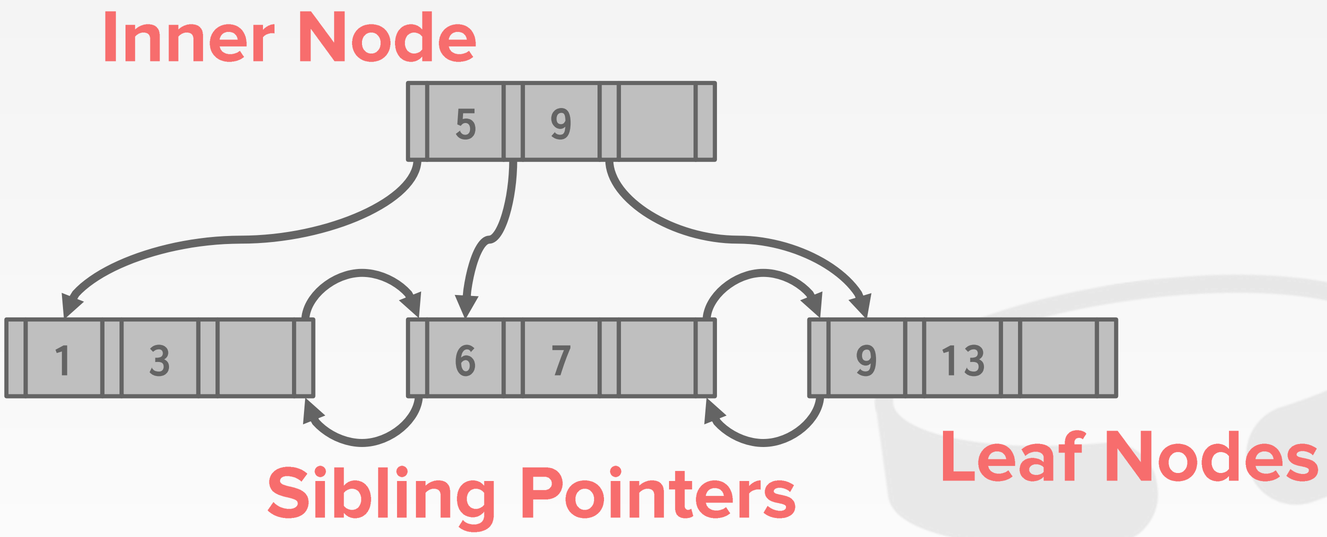
A B+tree is an ***M***-way search tree with the following properties:

→ It is perfectly balanced (i.e., every leaf node is at the same depth).

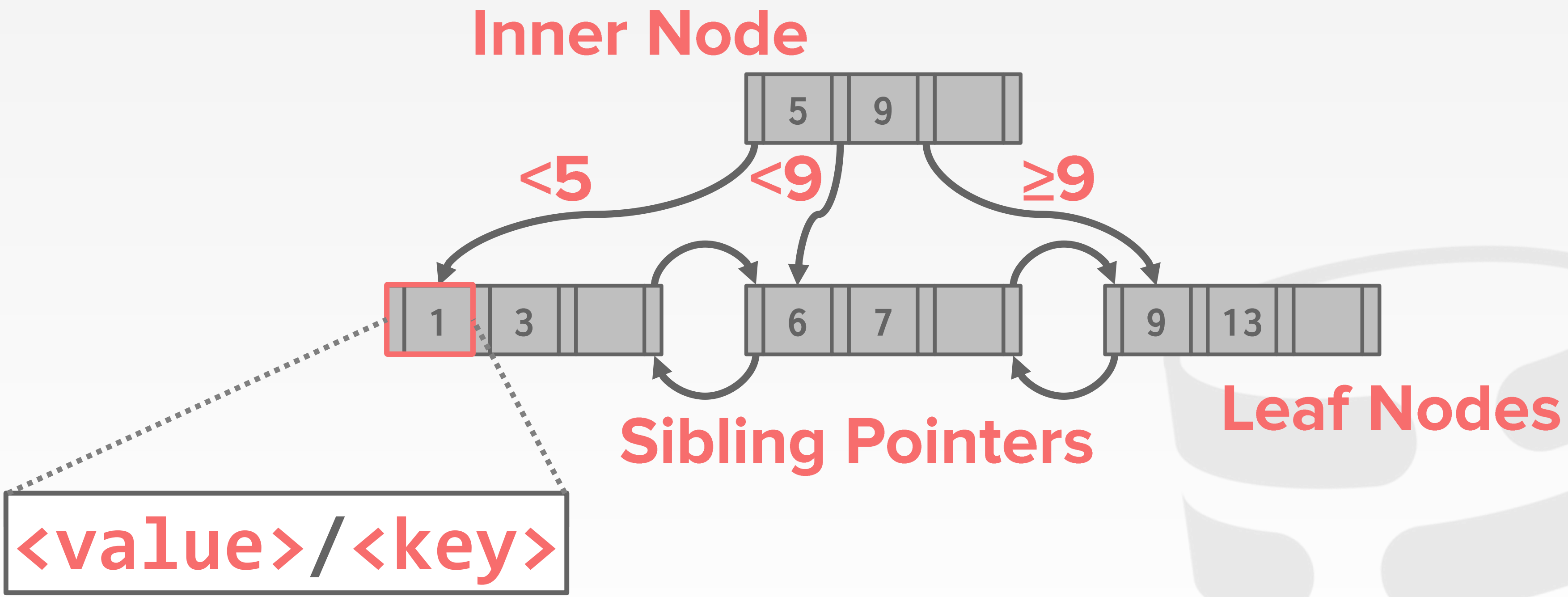
→ Every inner node other than the root, is at least half-full
`M/2-1 ≤ #keys ≤ M-1`

→ Every inner node with **k** keys has **k+1** non-null children

B+TREE OVERVIEW



B+TREE OVERVIEW



CARNEGIE MELLON
DATABASE GROUP

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

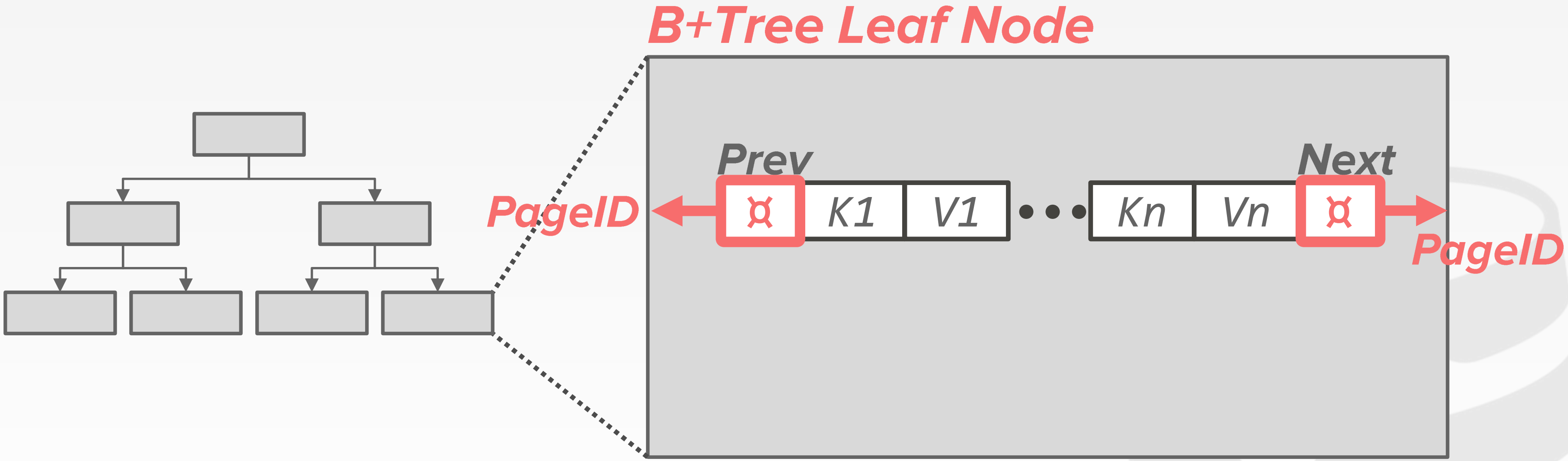
→ A pointer to the location of the tuple that the index entry corresponds to.

Approach #2: Tuple Data

→ The actual contents of the tuple is stored in the leaf node.

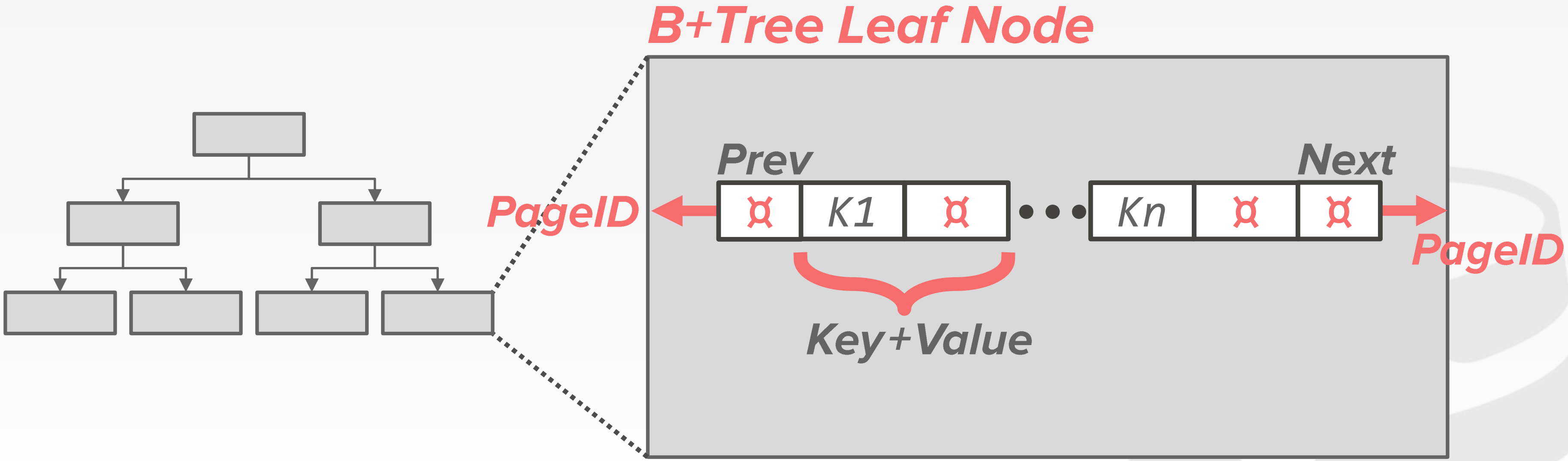
→ Secondary indexes have to store the record id as their values.

B+TREE LEAF NODES

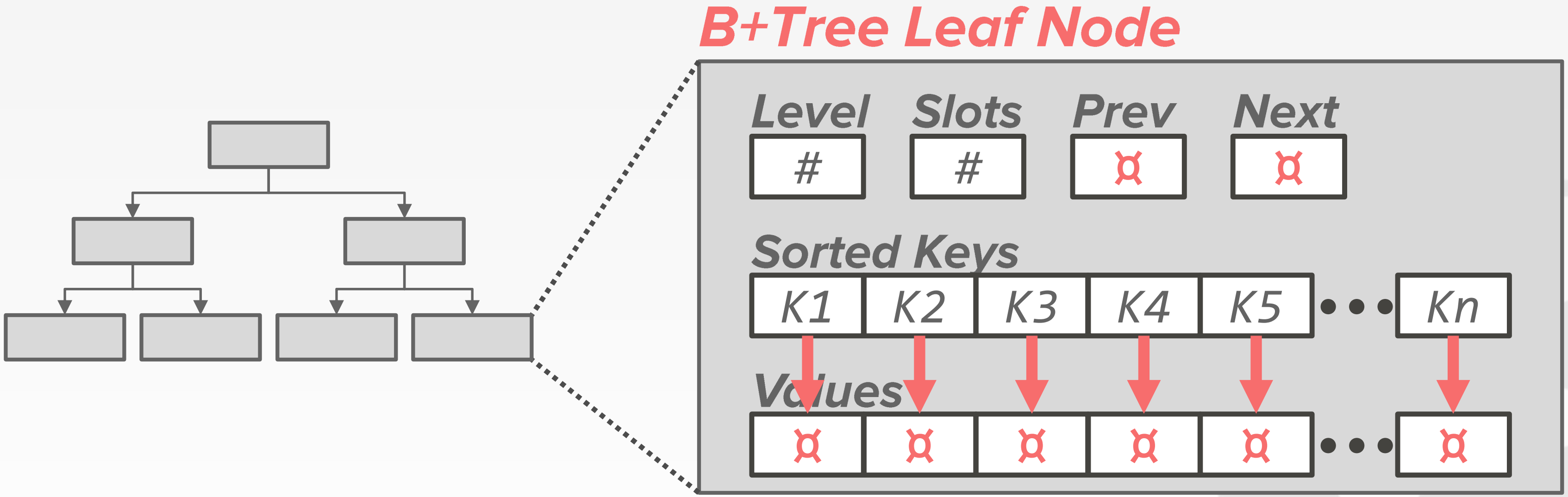


CARNEGIE MELLON DATABASE GROUP

B+TREE LEAF NODES



B+TREE LEAF NODES



CARNEGIE MELLON
DATABASE GROUP

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.

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

B+TREE: INSERT

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

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

Typical Fill-Factor: 67%.

→ Average Fanout = 2*100*0.67 = 134

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

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

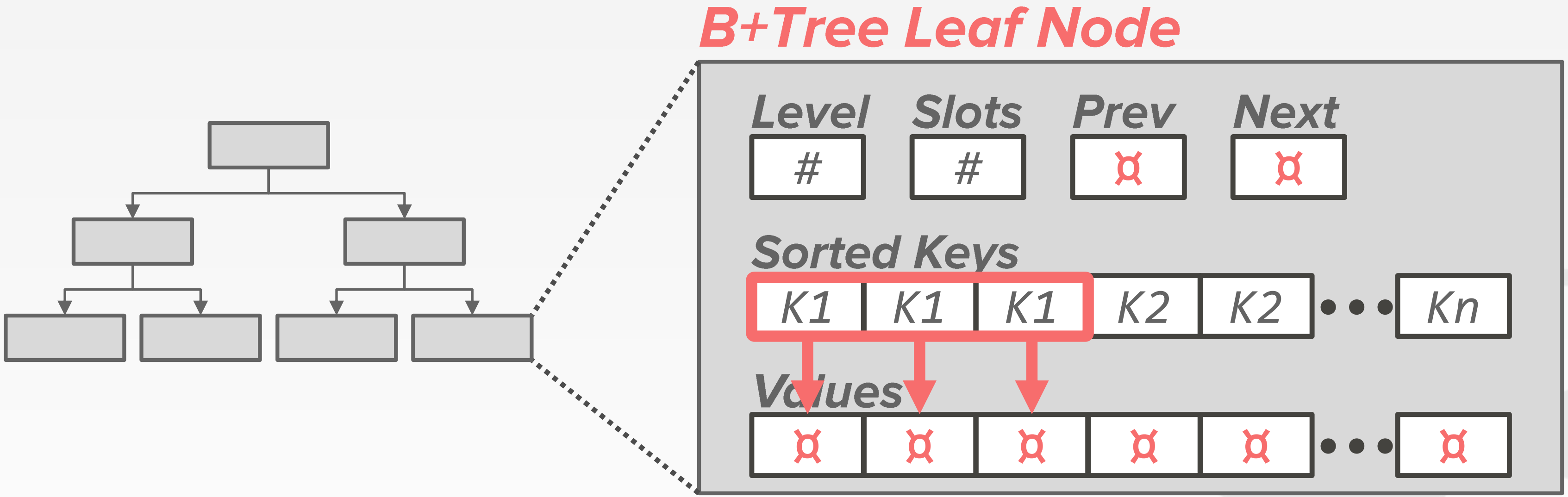
Approach #1: Duplicate Keys

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

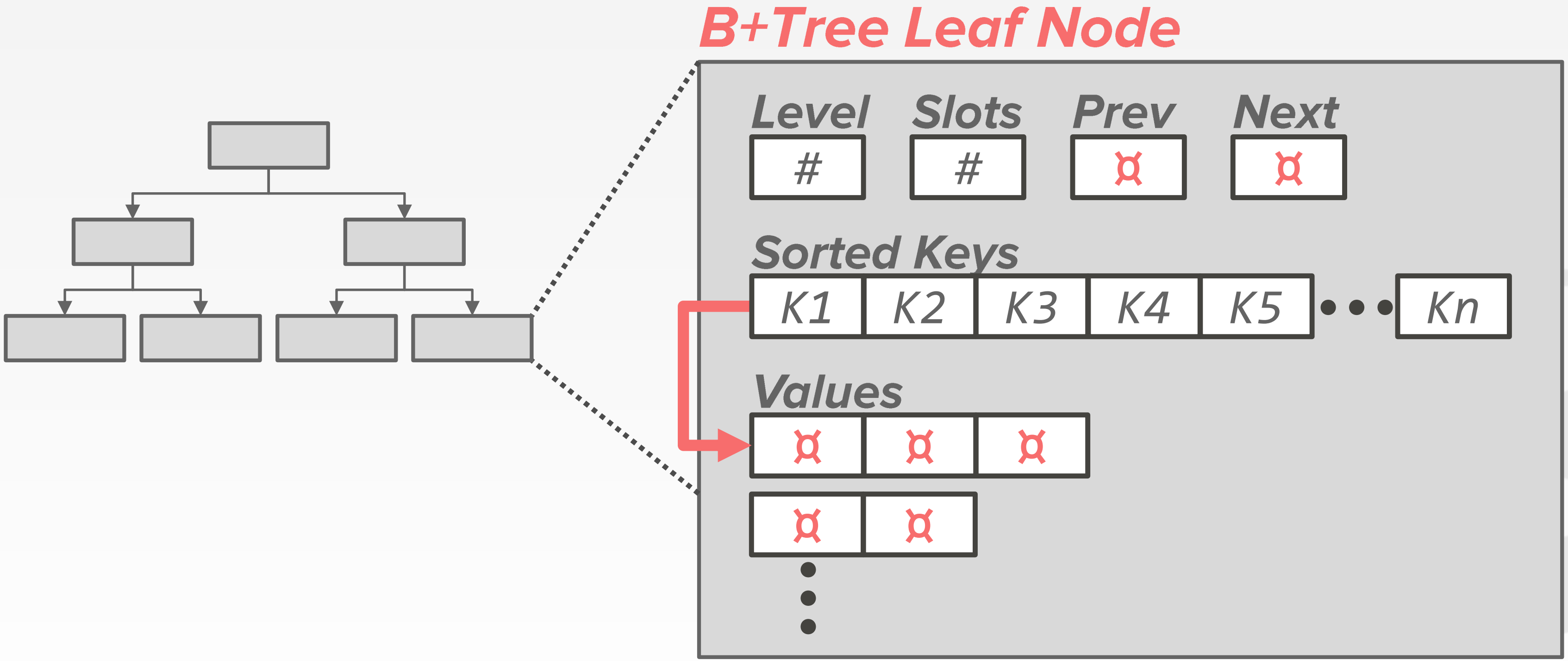
Approach #2: Value Lists

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

B+TREE: DUPLICATE KEYS



B+TREE: VALUE LISTS



B+TREE: VARIABLE LENGTH KEYS

Approach #1: Pointers

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

Approach #2: Variable Length Nodes

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

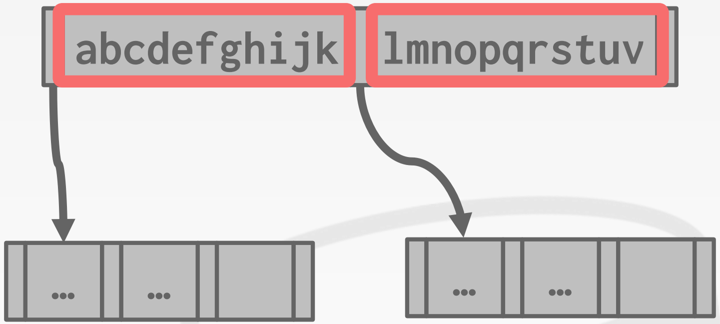
Approach #3: Key Map

→ 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.

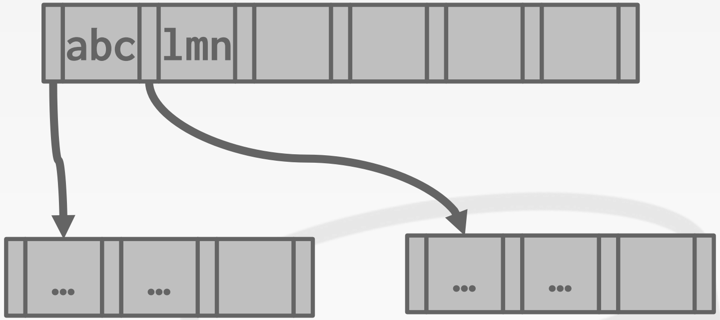


B+TREE: PREFIX COMPRESSION

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.

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

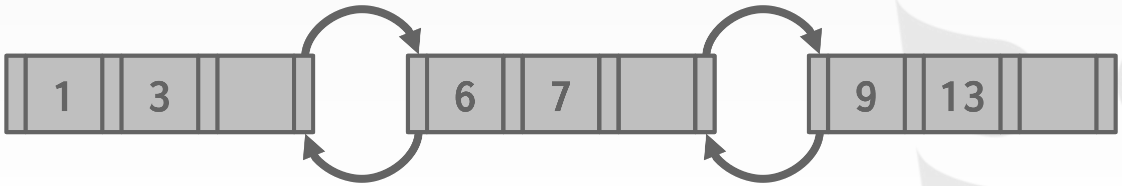
Sorted Keys: 1, 3, 6, 7, 9, 13

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



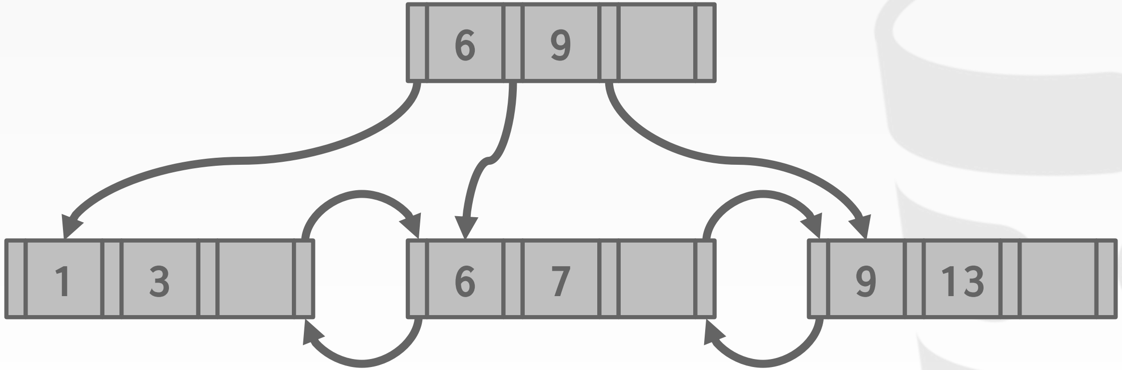
CARNEGIE MELLON DATABASE GROUP

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

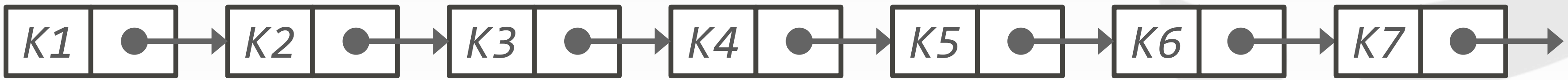


OBSERVATION

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)

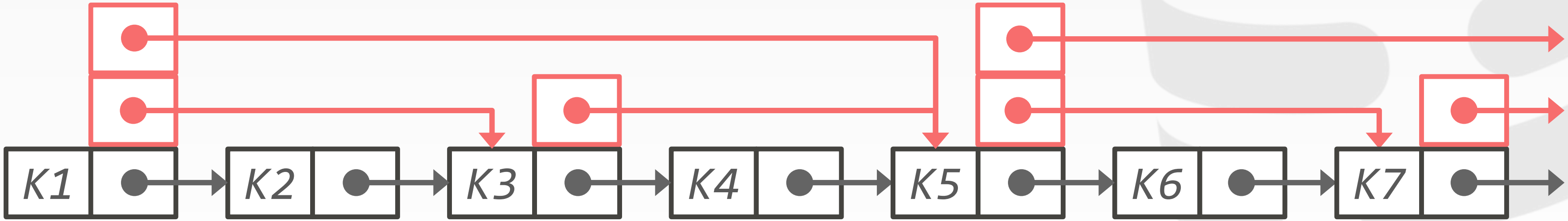


OBSERVATION

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)



CARNEGIE MELLON DATABASE GROUP

SKIP LISTS

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 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.

MEMSQL WIREDTIGER RocksDB

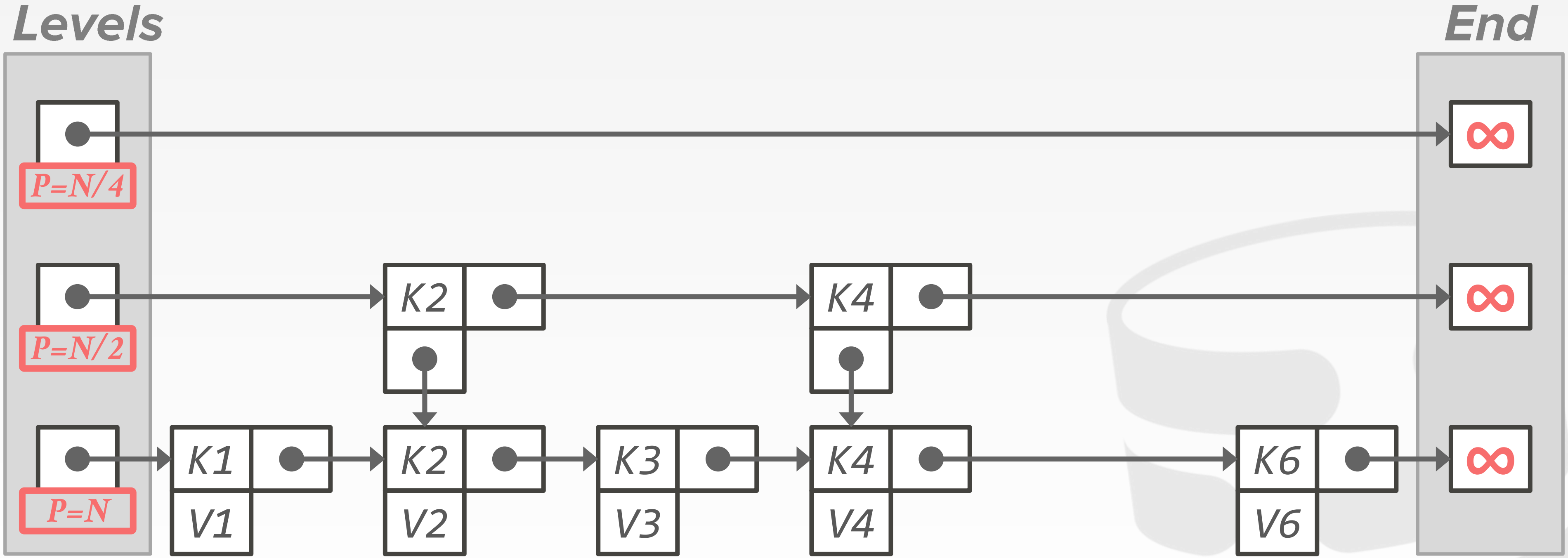
SKIP LISTS

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

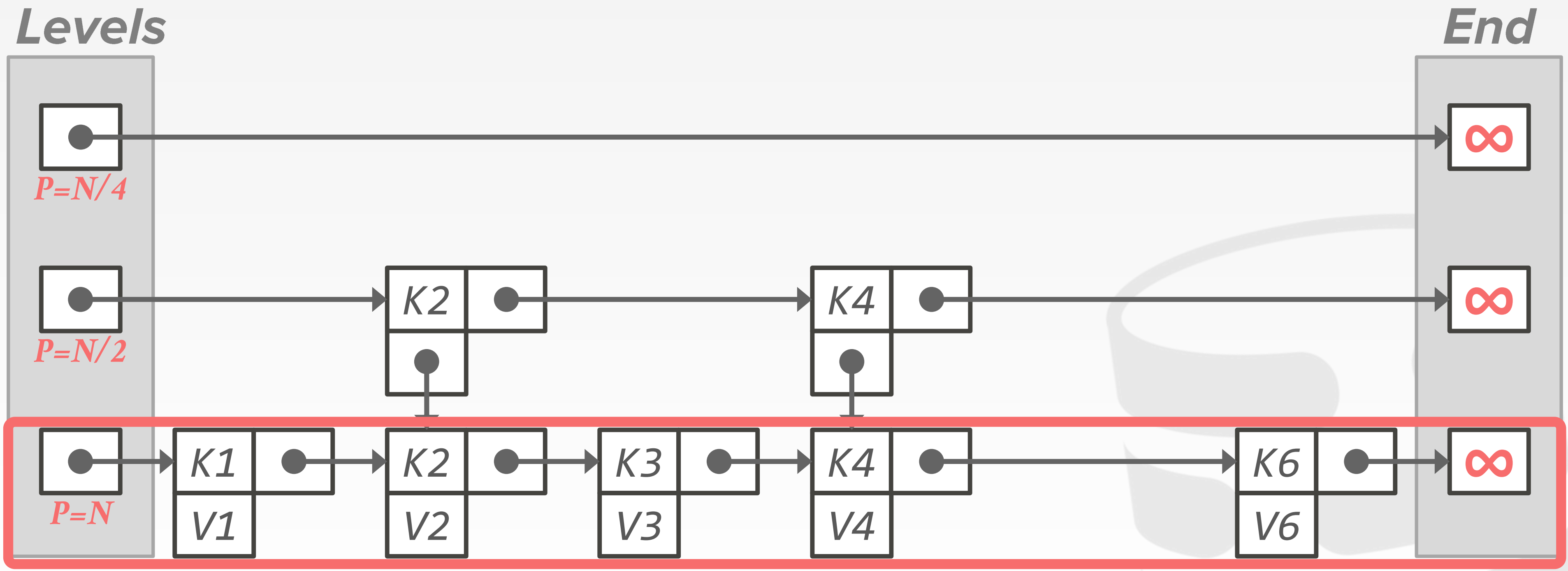
To insert a new key, flip a coin to decide how many levels to add the new key into. Provides approximate **O(log n)** search times.

SKIP LISTS: EXAMPLE

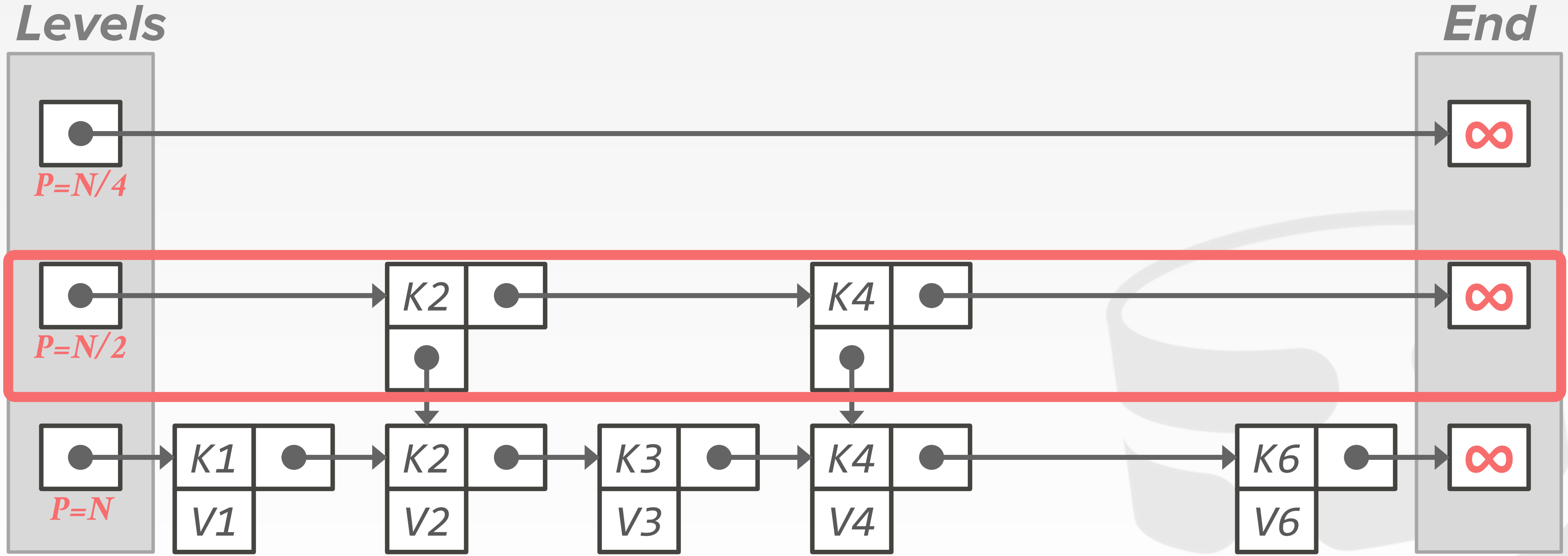


CARNEGIE MELLON
DATABASE GROUP

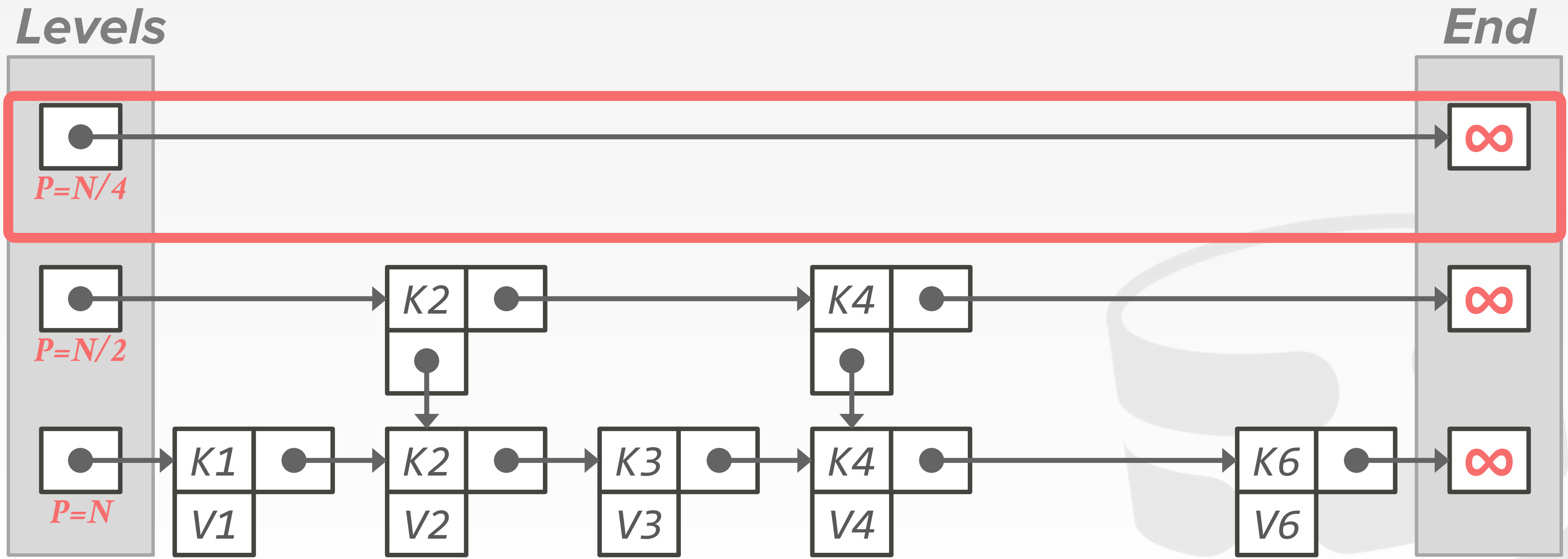
SKIP LISTS: EXAMPLE



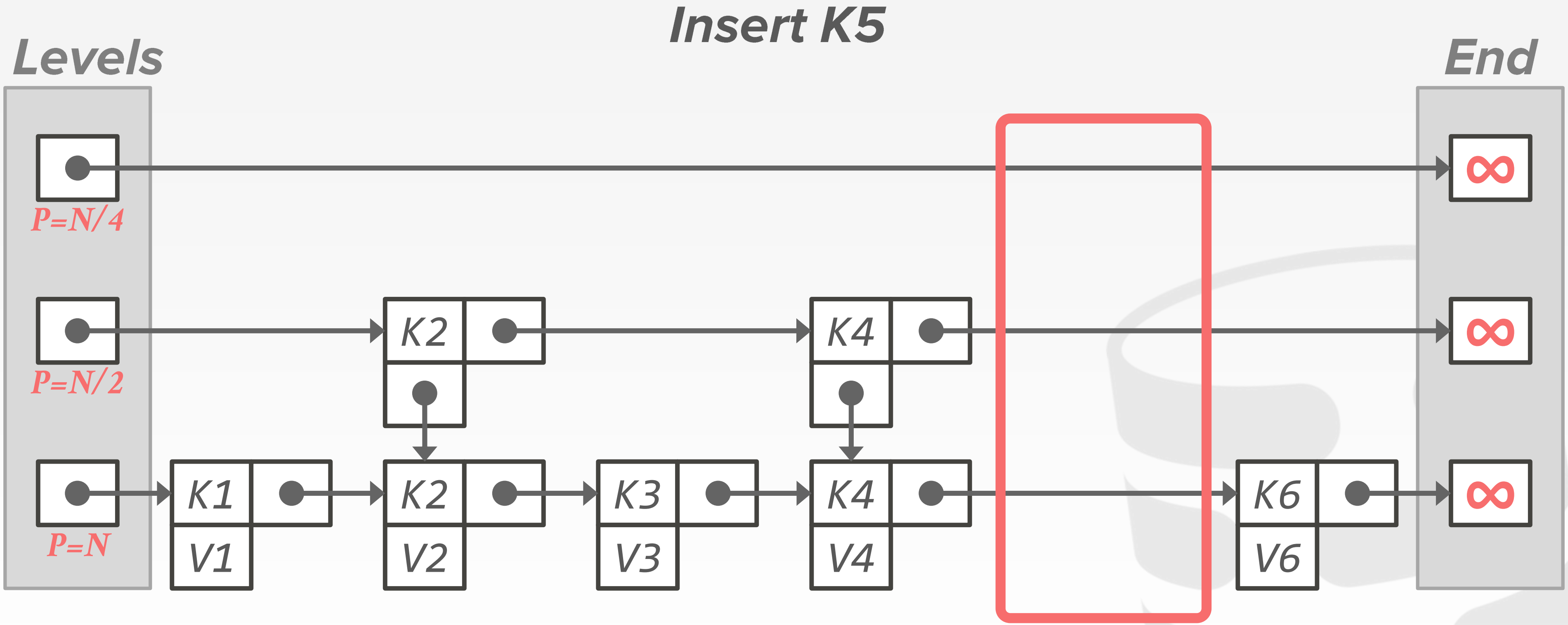
SKIP LISTS: EXAMPLE



SKIP LISTS: EXAMPLE

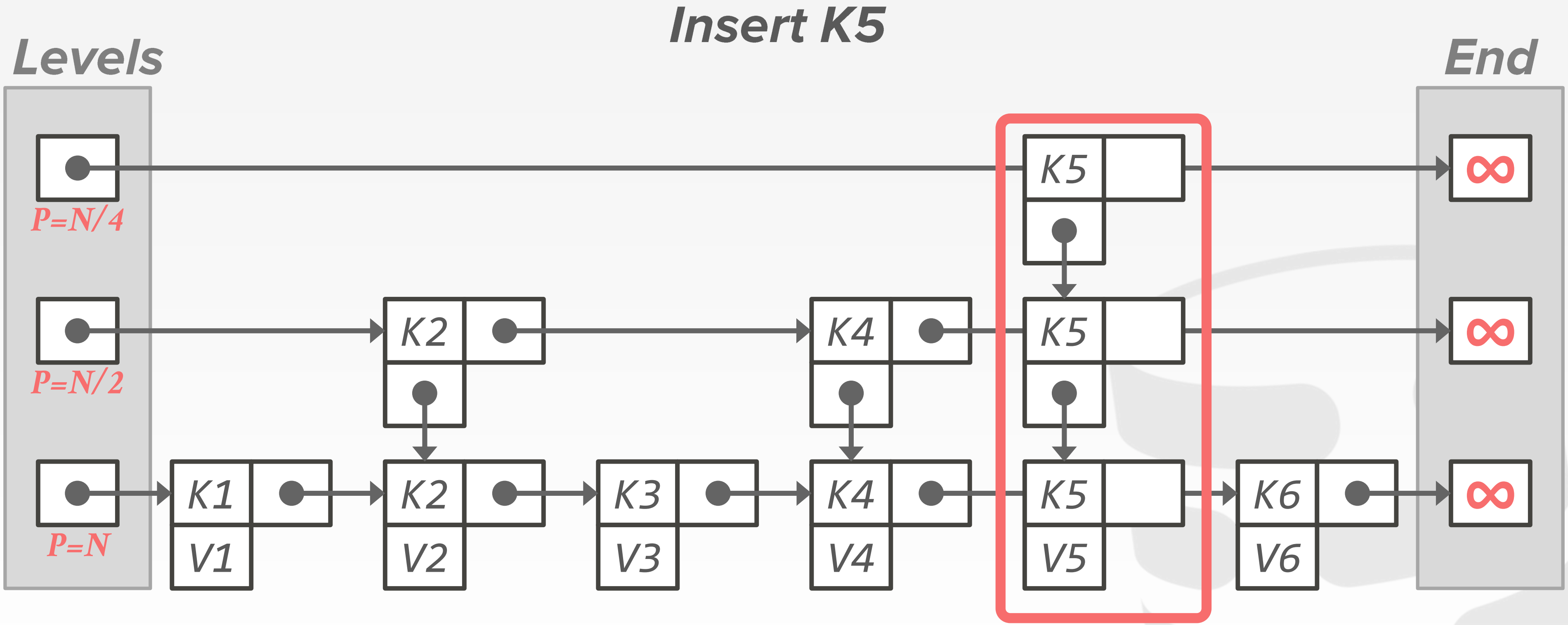


SKIP LISTS: INSERT

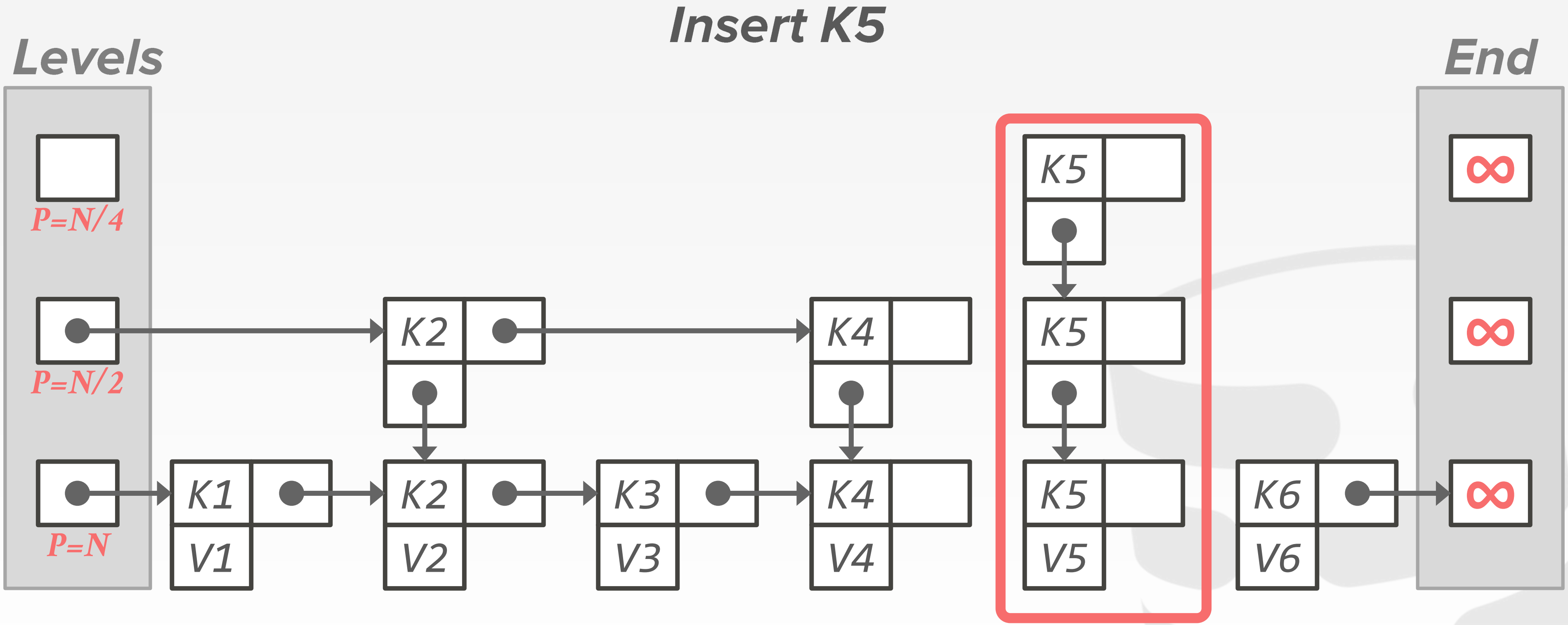


CARNEGIE MELLON DATABASE GROUP

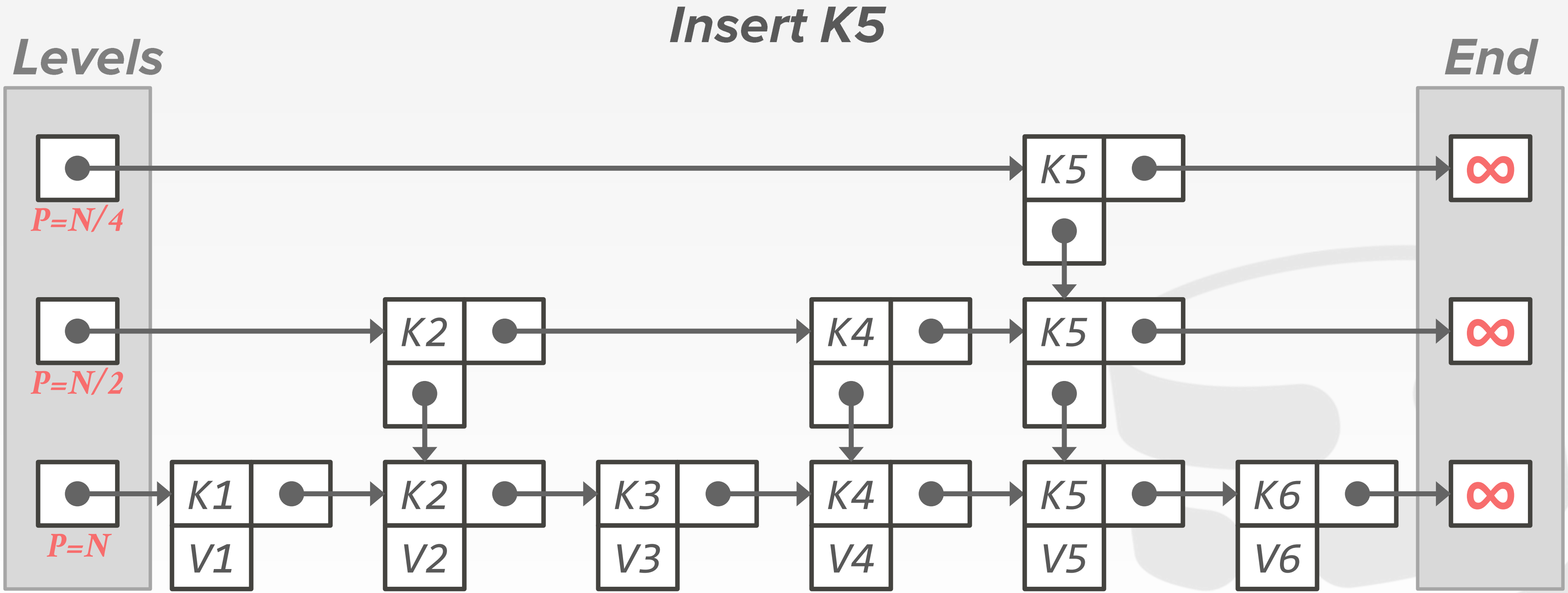
SKIP LISTS: INSERT



SKIP LISTS: INSERT

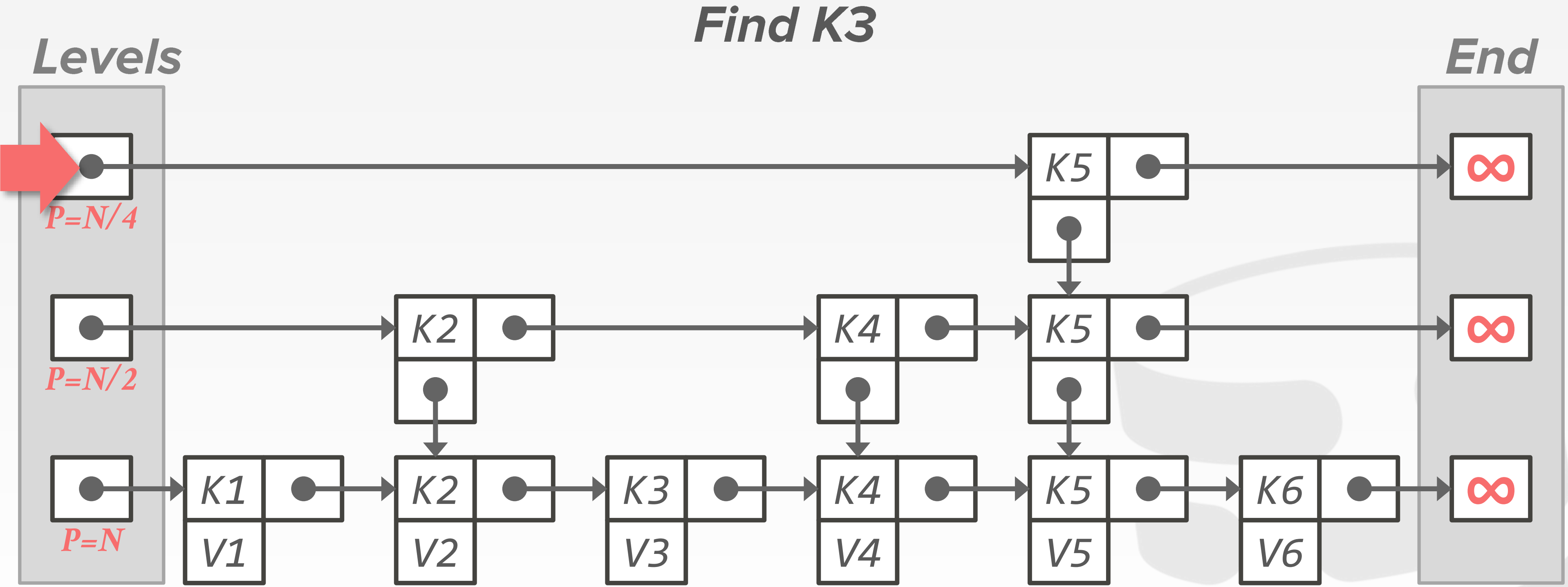


SKIP LISTS: INSERT

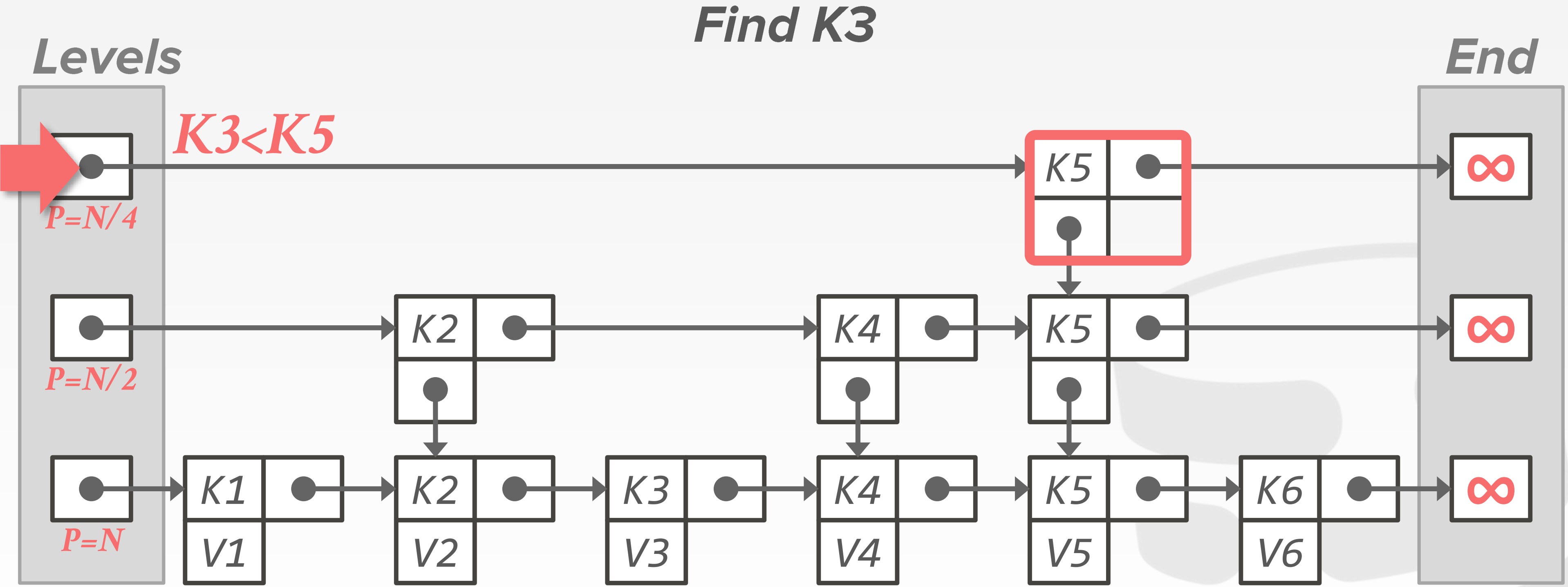


CARNEGIE MELLON DATABASE GROUP

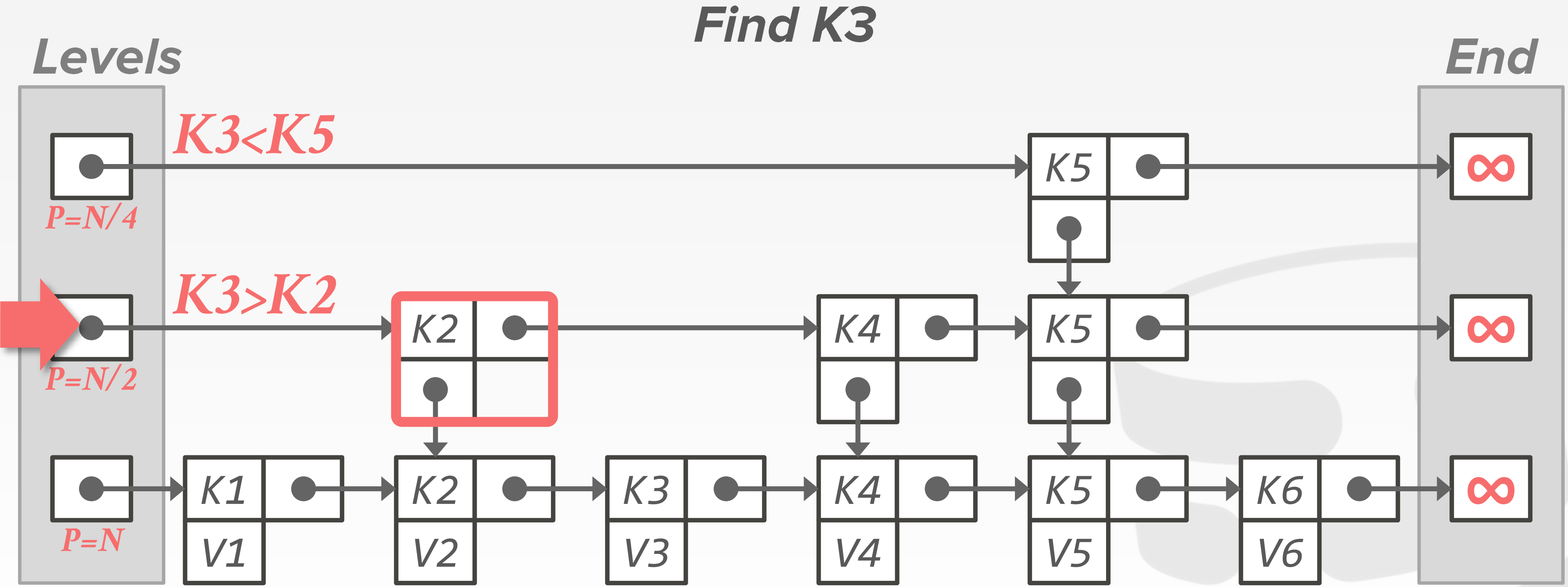
SKIP LISTS: SEARCH



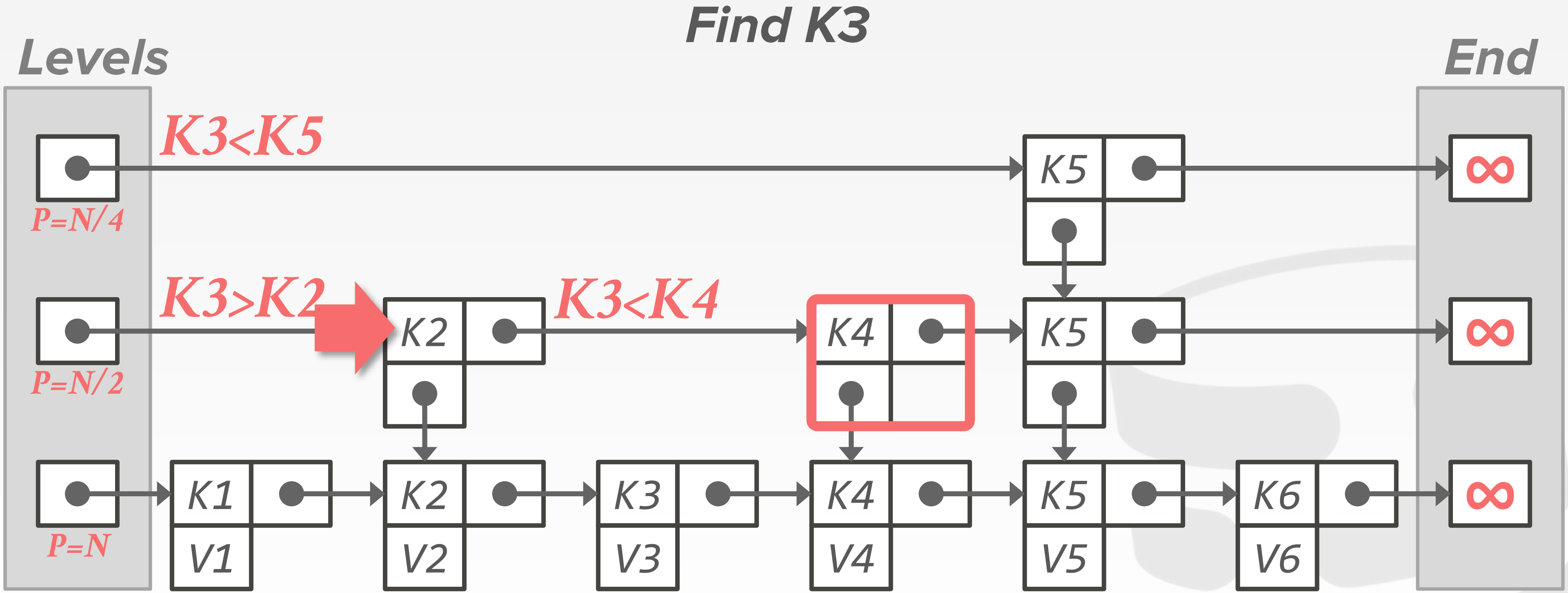
SKIP LISTS: SEARCH



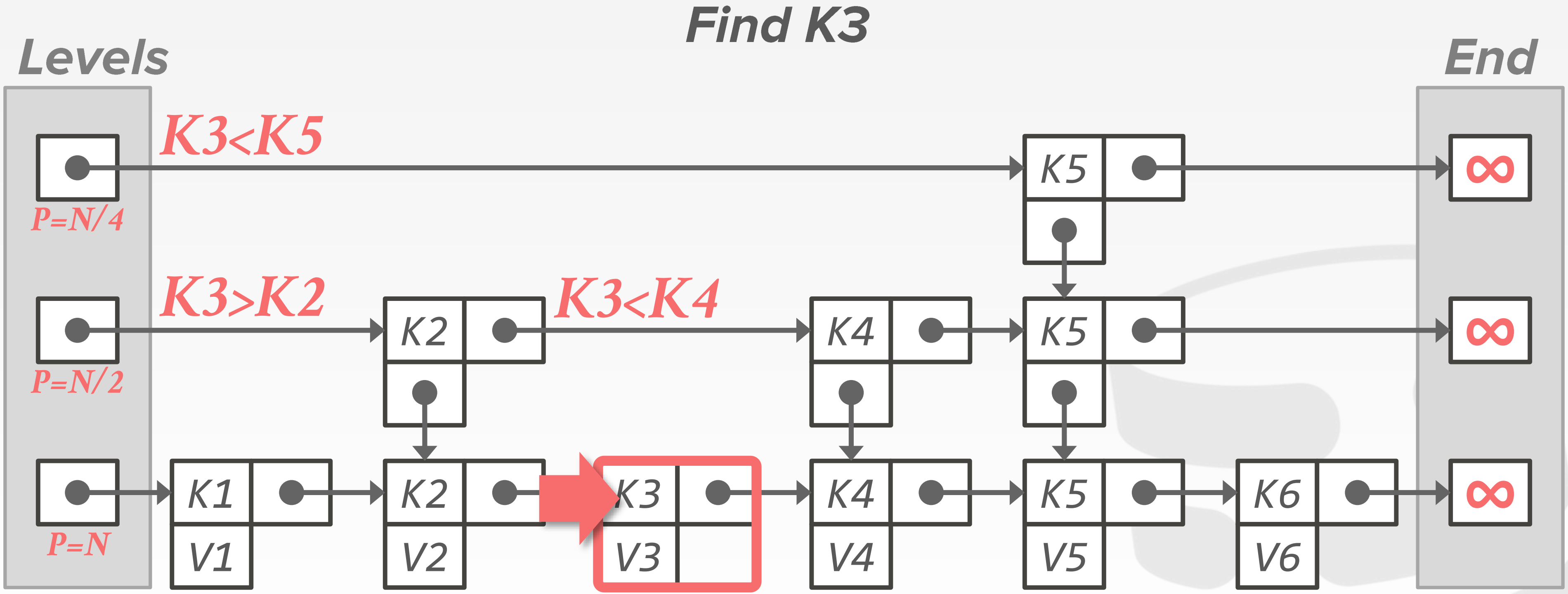
SKIP LISTS: SEARCH



SKIP LISTS: SEARCH



SKIP LISTS: SEARCH



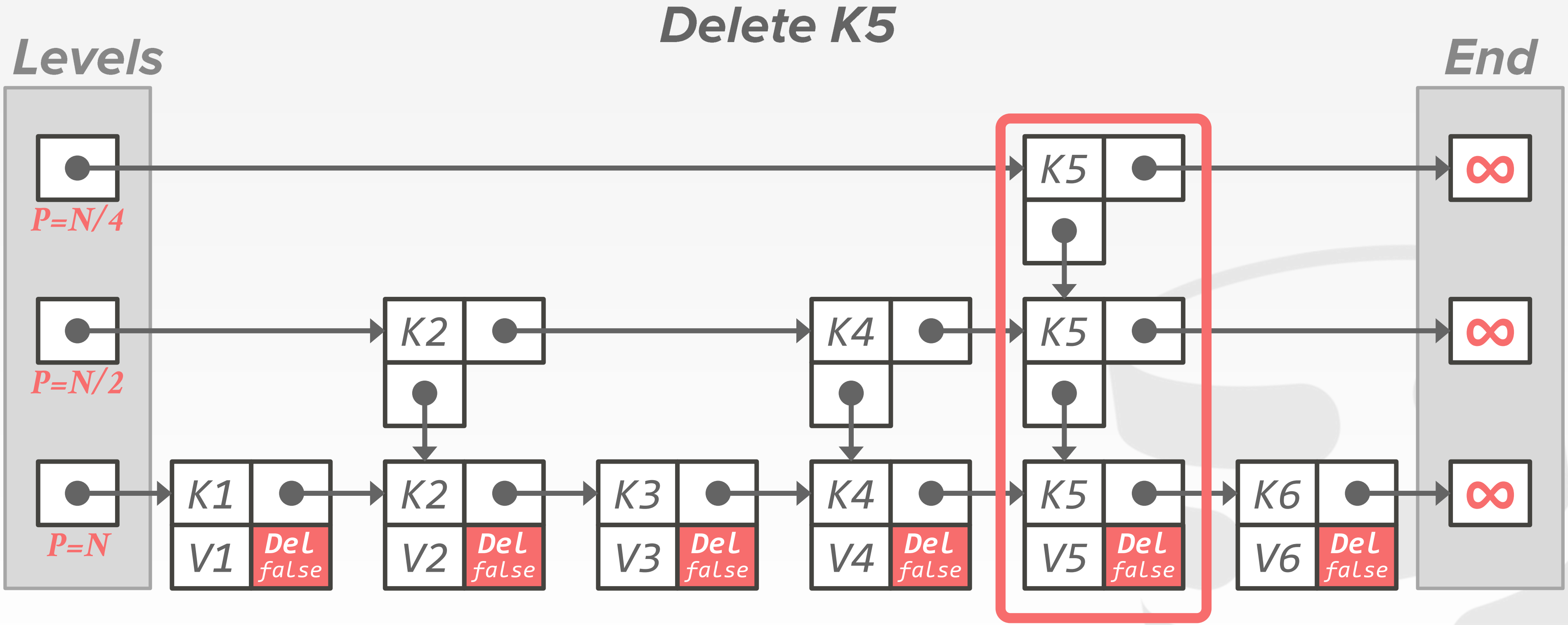
SKIP LISTS: DELETE

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

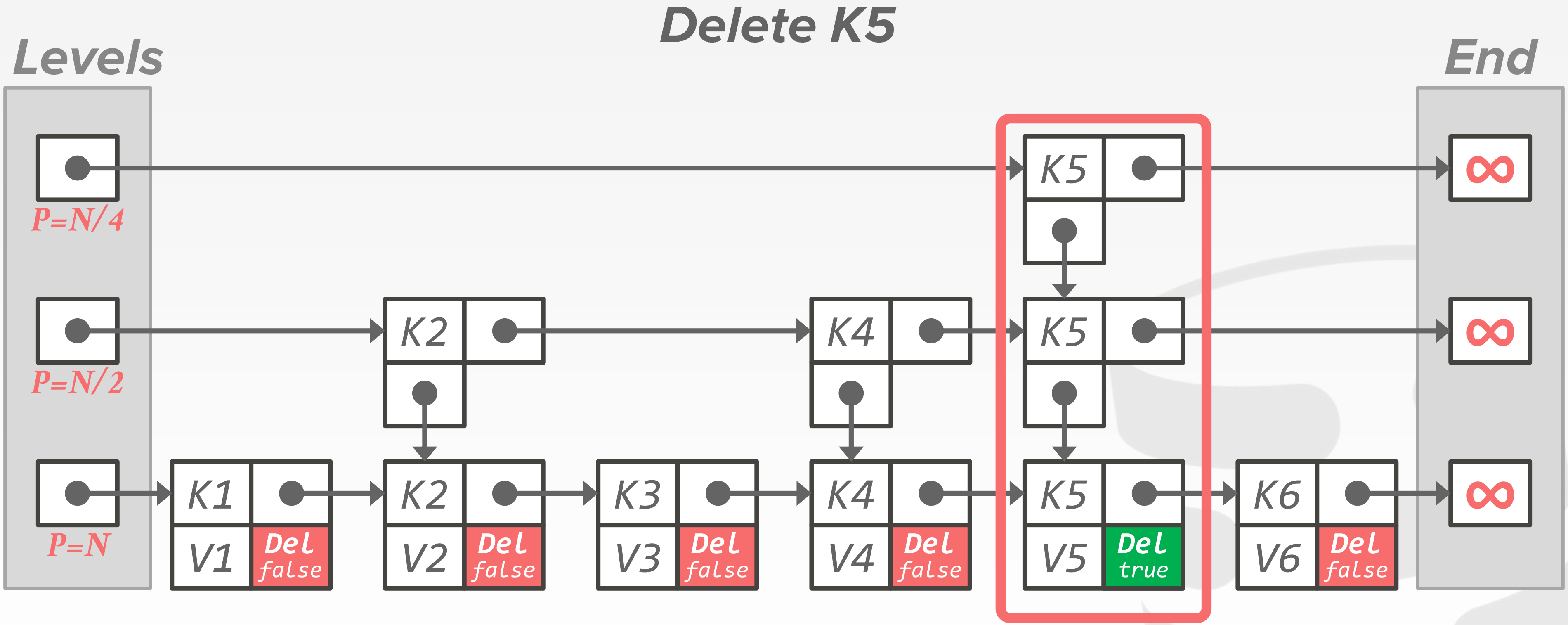
Then **physically** remove the key once we know that no other thread is holding the reference.

Source: Stephen Tu

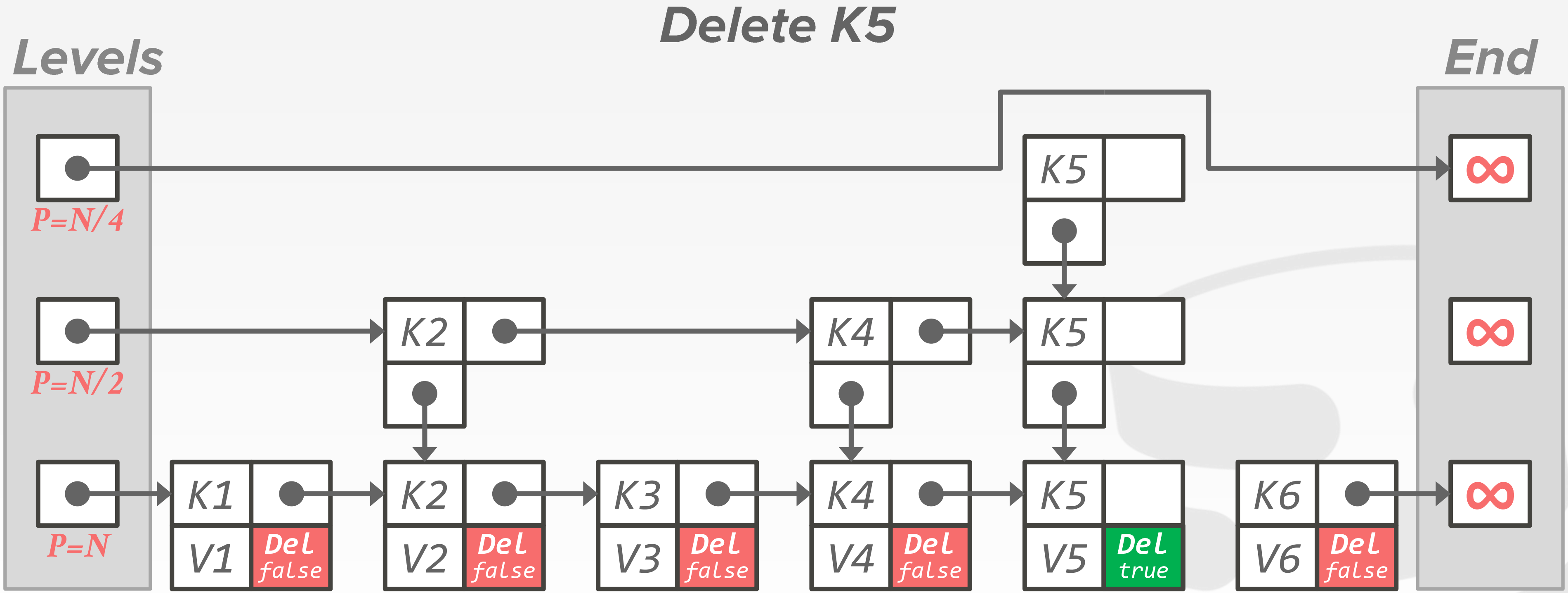
SKIP LISTS: DELETE



SKIP LISTS: DELETE

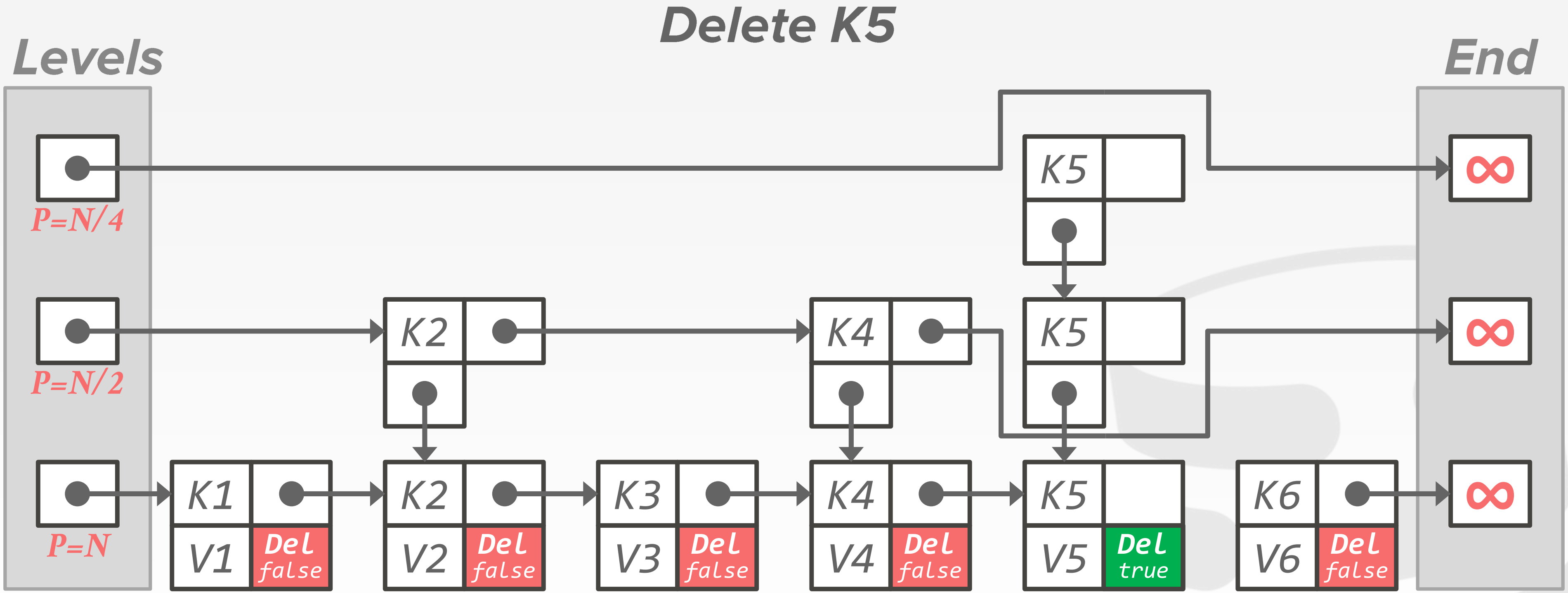


SKIP LISTS: DELETE



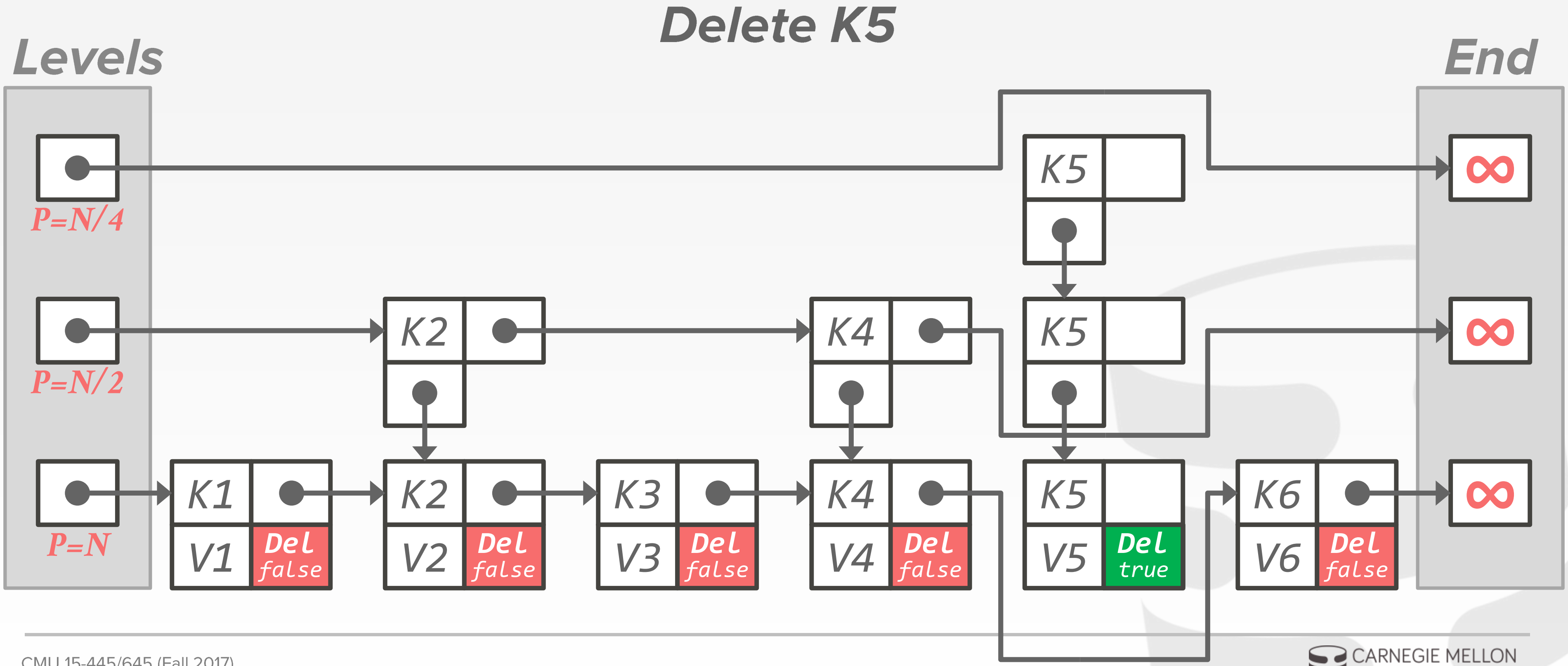
CARNEGIE MELLON
DATABASE GROUP

SKIP LISTS: DELETE



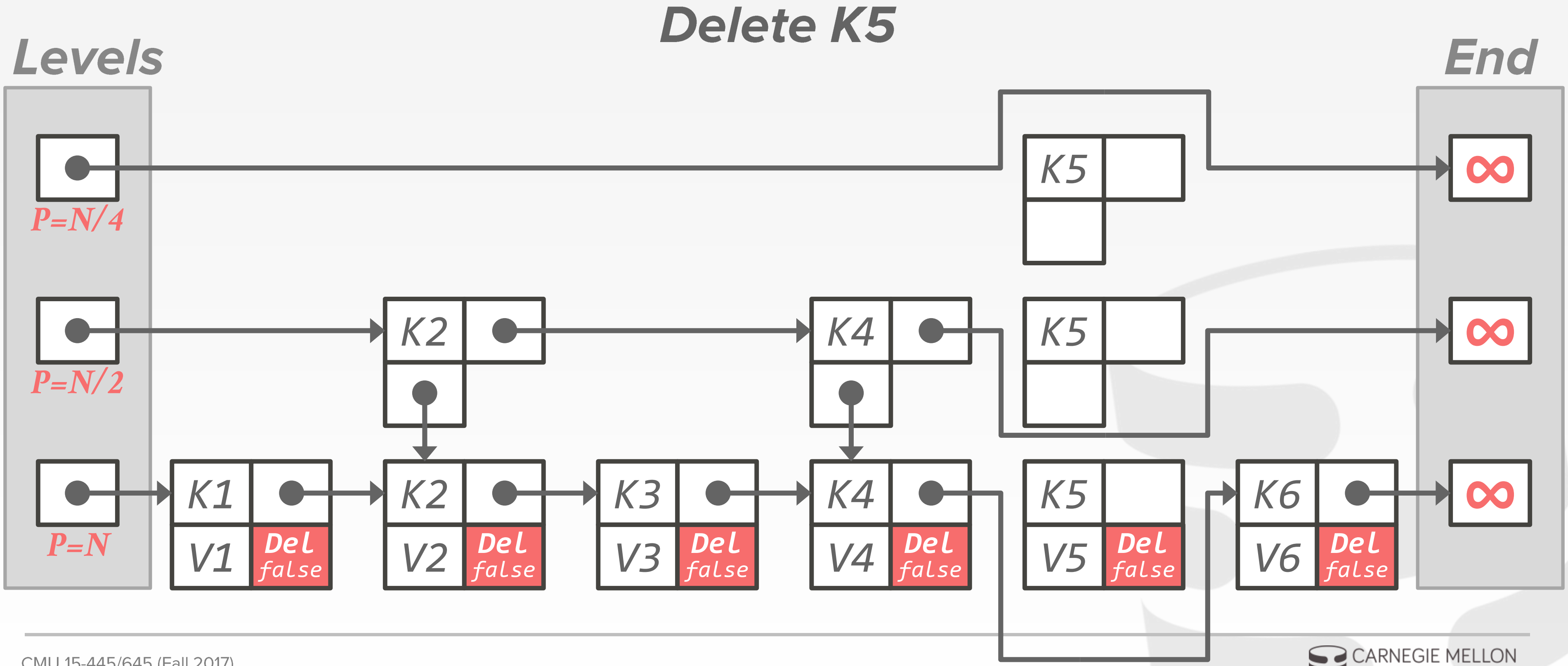
CARNEGIE MELLON DATABASE GROUP

SKIP LISTS: DELETE



CARNEGIE MELLON DATABASE GROUP

SKIP LISTS: DELETE



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.

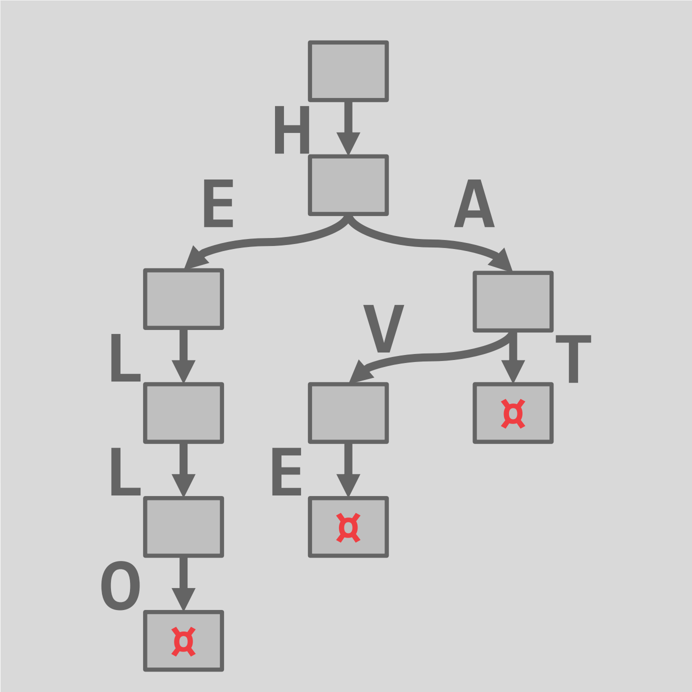
RADIX TREE

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

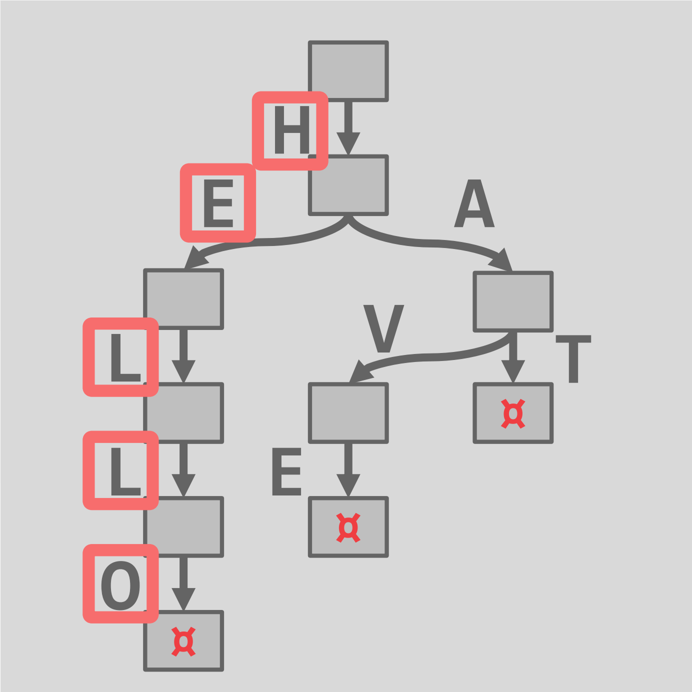
Trie



Keys: HELLO, HAT, HAVE

TRIE VS. RADIX TREE

Trie

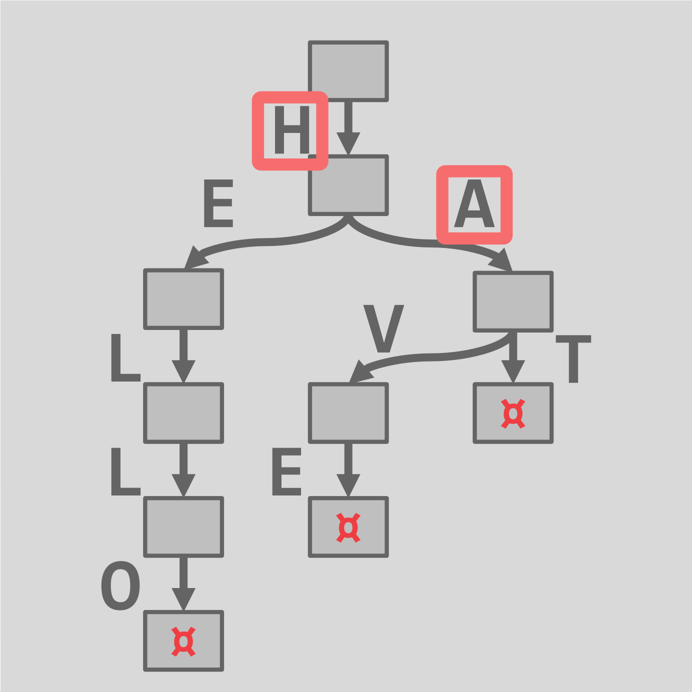


Keys: HELLO HAT, HAVE

CARNEGIE MELLON DATABASE GROUP

TRIE VS. RADIX TREE

Trie

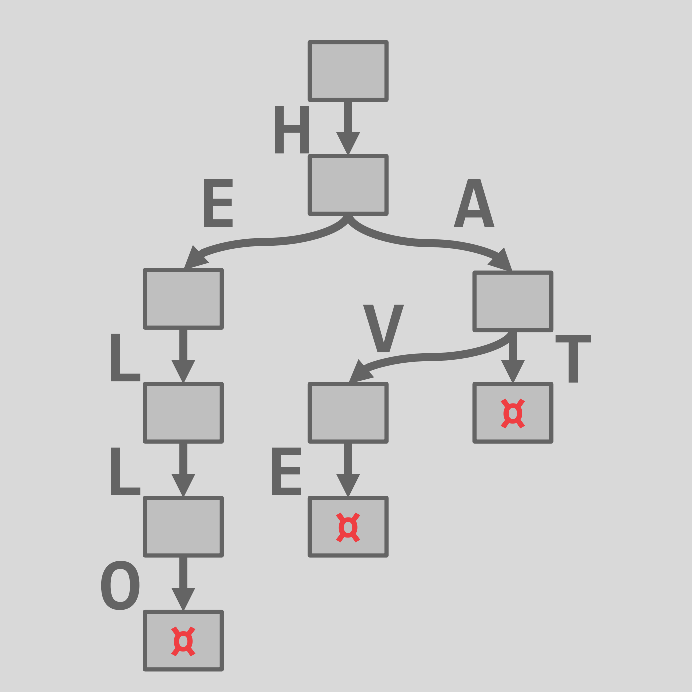


Keys: HELLO, HAT, HAVE

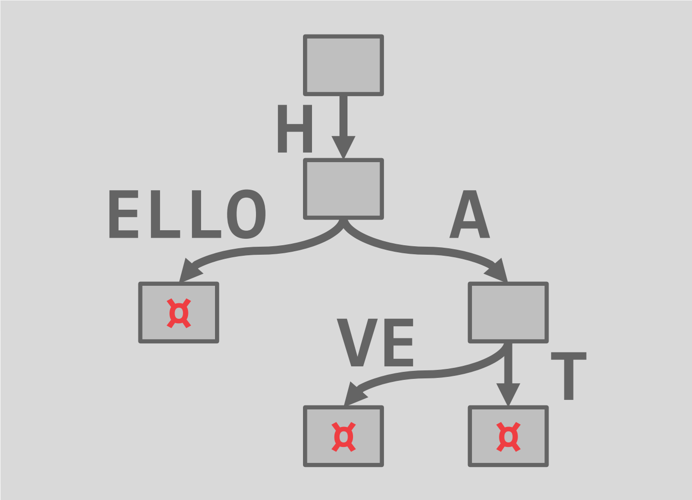
CARNEGIE MELLON DATABASE GROUP

TRIE VS. RADIX TREE

Trie



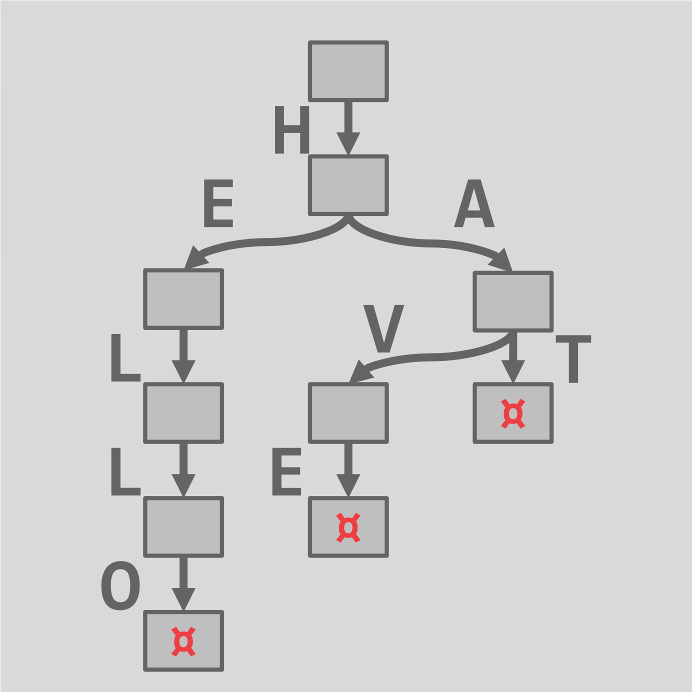
Radix Tree



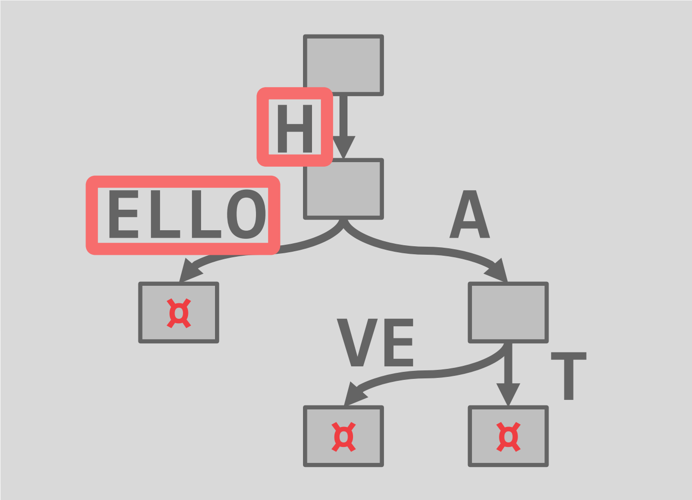
Keys: HELLO, HAT, HAVE

TRIE VS. RADIX TREE

Trie



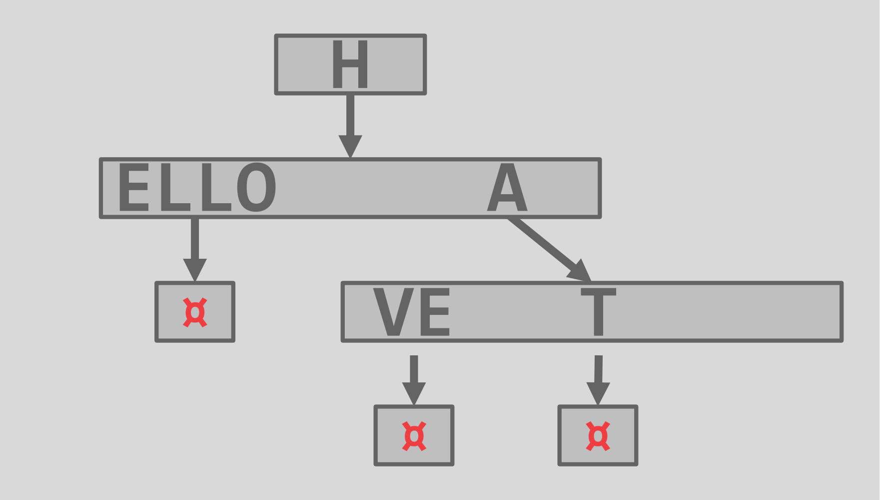
Radix Tree



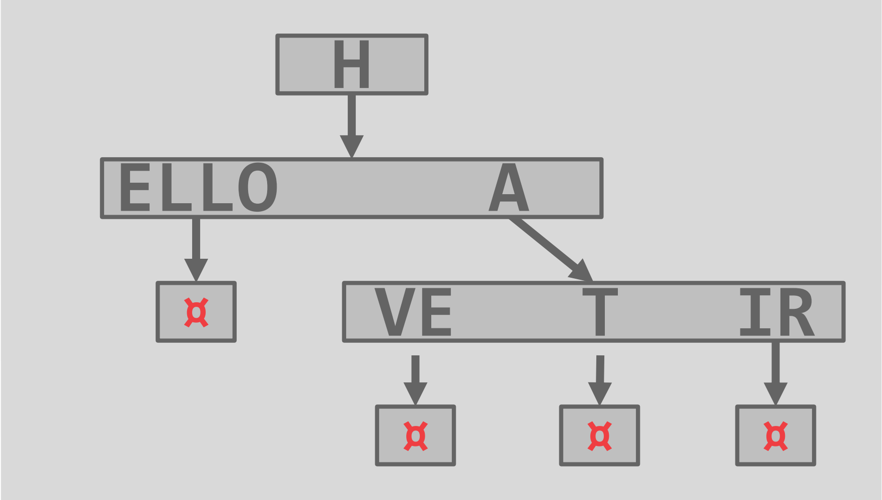
Keys: HELLO HAT, HAVE

CARNEGIE MELLON
DATABASE GROUP

RADIX TREE: MODIFICATIONS



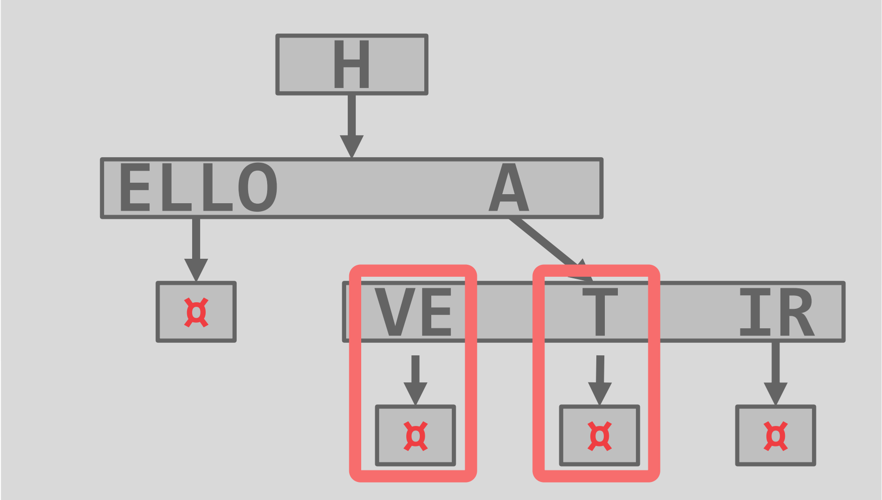
RADIX TREE: MODIFICATIONS



Insert HAIR

CARNEGIE MELLON DATABASE GROUP

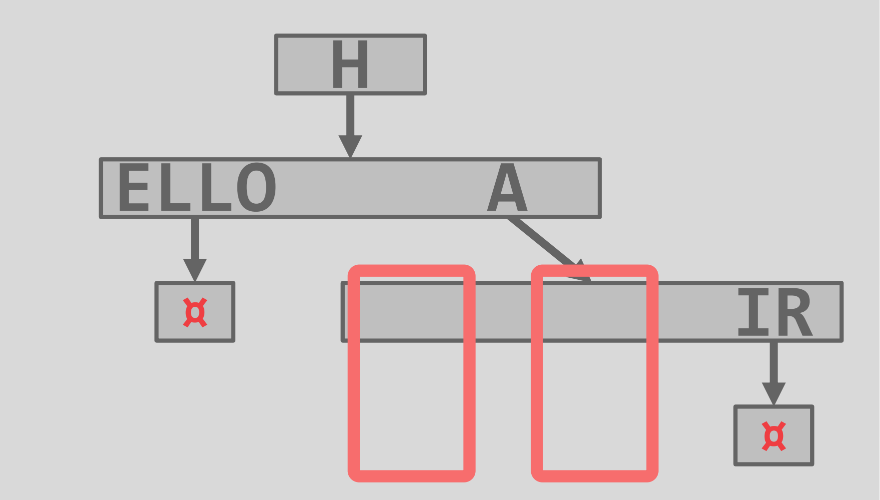
RADIX TREE: MODIFICATIONS



Insert HAIR

Delete HAT, HAVE

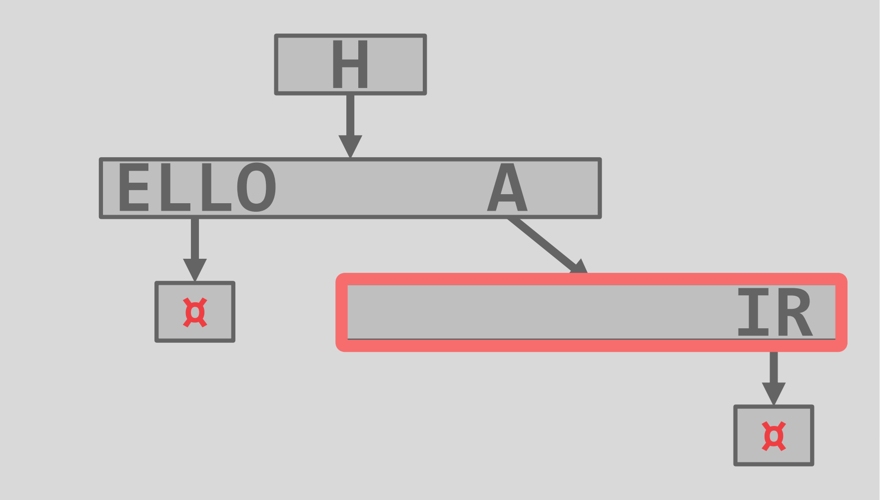
RADIX TREE: MODIFICATIONS



Insert HAIR

Delete HAT, HAVE

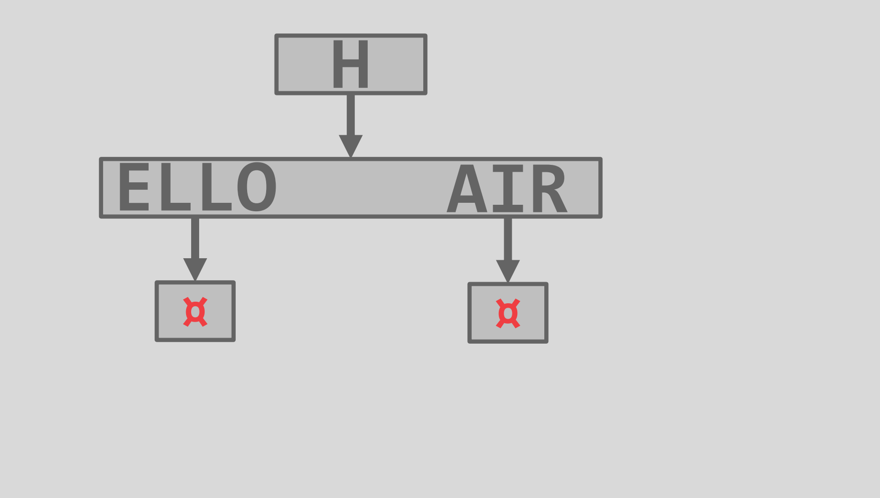
RADIX TREE: MODIFICATIONS



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.

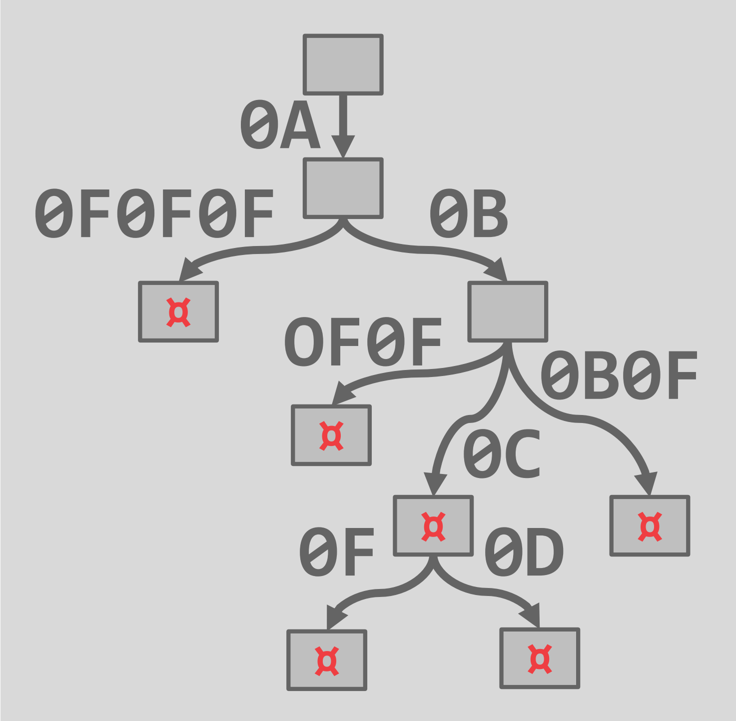
RADIX TREE: BINARY COMPARABLE KEYS

Int Key: 168496141

Hex Key: 0A 0B 0C 0D

Big Endian *Little Endian*

RADIX TREE: BINARY COMPARABLE KEYS

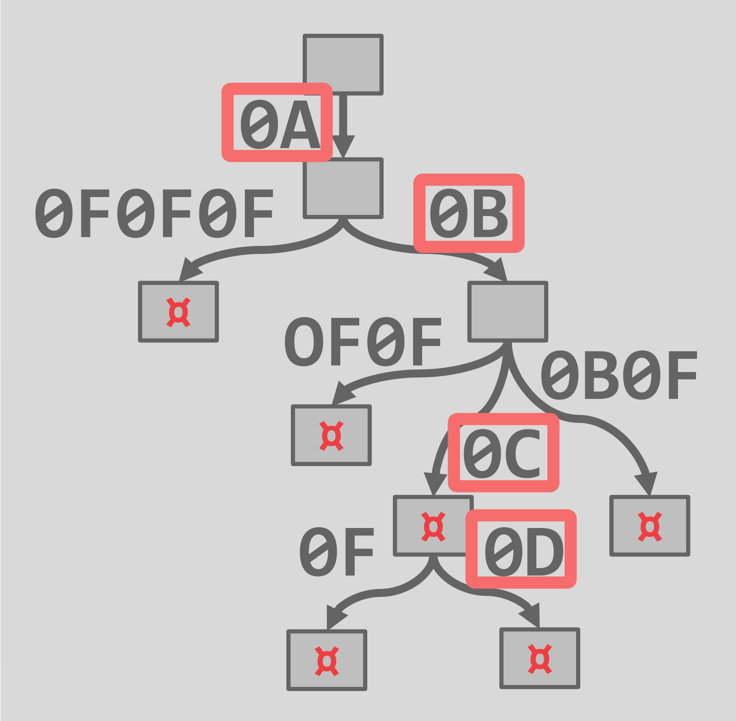


Int Key: 168496141

Hex Key: 0A 0B 0C 0D

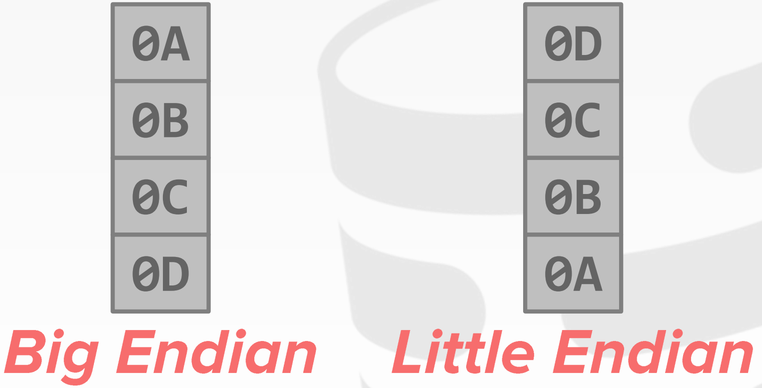
Big Endian *Little Endian*

RADIX TREE: BINARY COMPARABLE KEYS



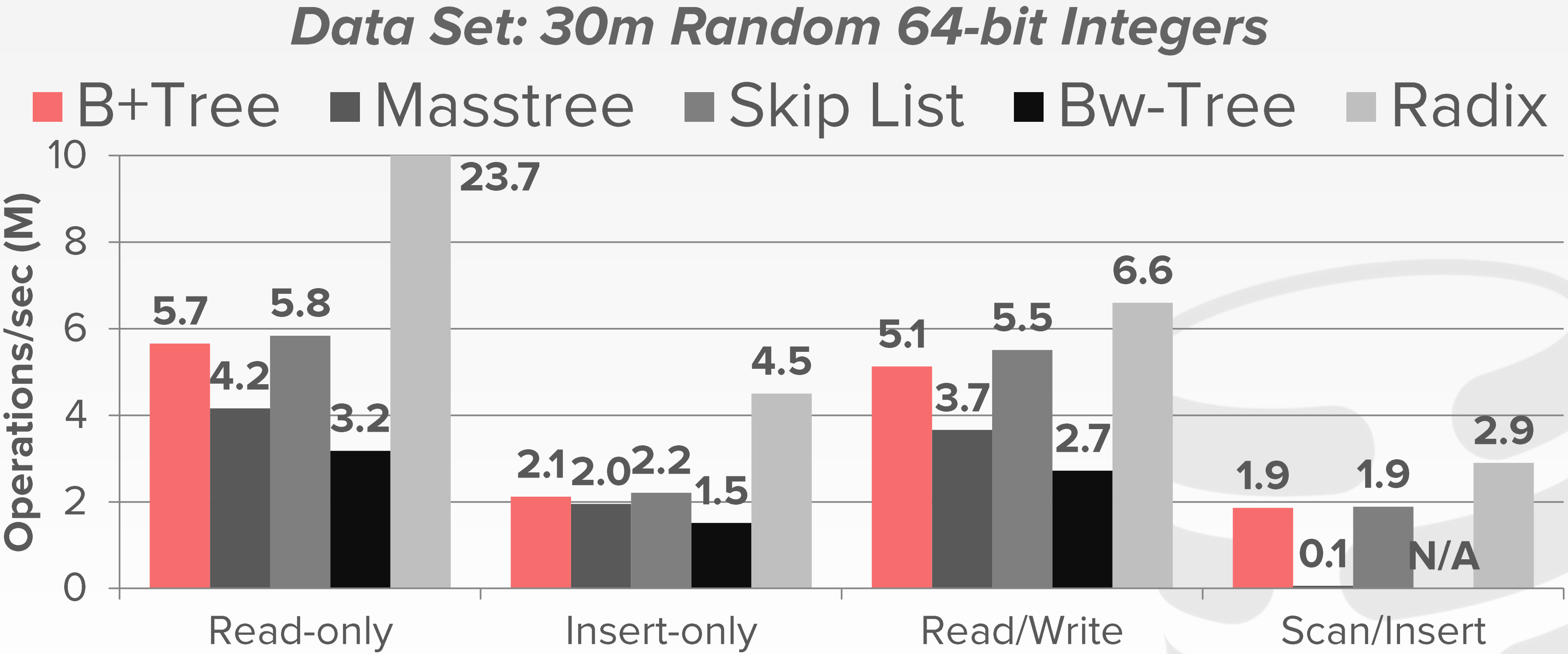
Int Key: 168496141

Hex Key: 0A 0B 0C 0D



Source: Huanchen Zhang

SINGLE-THREADED PERFORMANCE



CARNEGIE MELLON DATABASE GROUP

SELECTION CONDITIONS

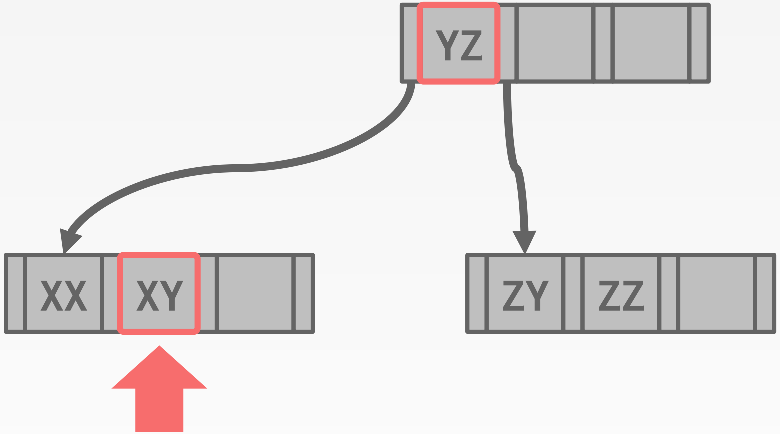
The DBMS can use a B+Tree index if the query provides all of the attributes in a prefix of the search key.

→ Index on `<a,b,c>` matches `(a=5 AND b=3)`, but not `b=3`.

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

B+TREE PREFIX SEARCH

Find "XY"

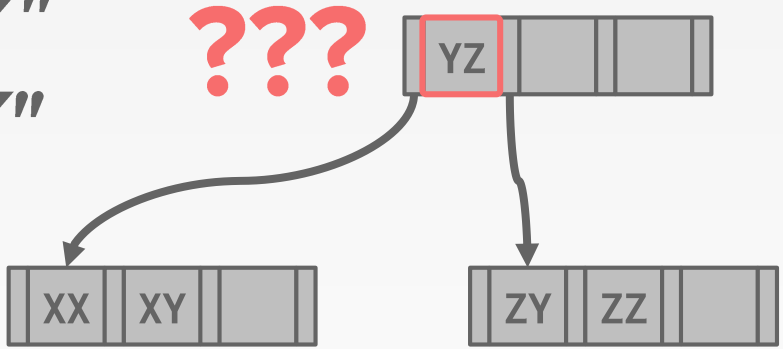


CARNEGIE MELLON DATABASE GROUP

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!