## B+TREE

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

→ Generalization of a binary search tree, since a node can have more than two children.

→ Optimized for systems that read and write large blocks of data.

## B+TREE PROPERTIES

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

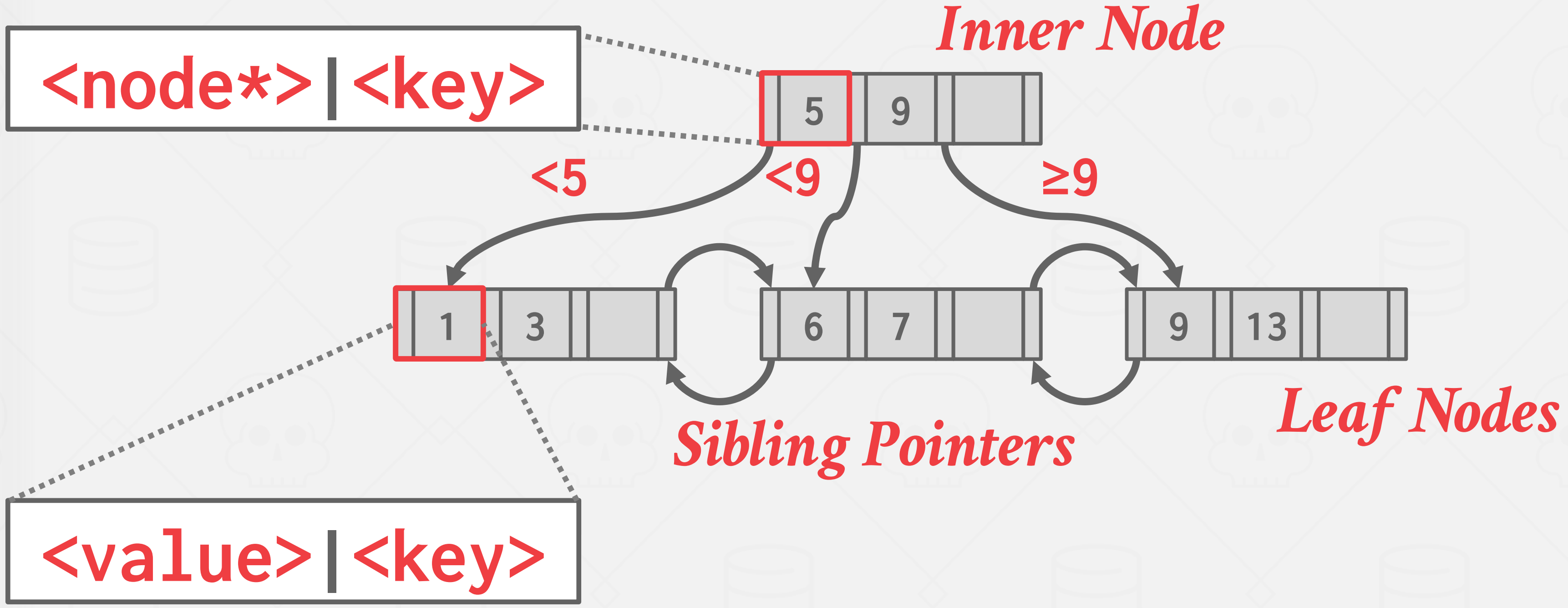
→ It is perfectly balanced (i.e., every leaf node is at the same depth in the tree)

→ Every 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 EXAMPLE



## NODES

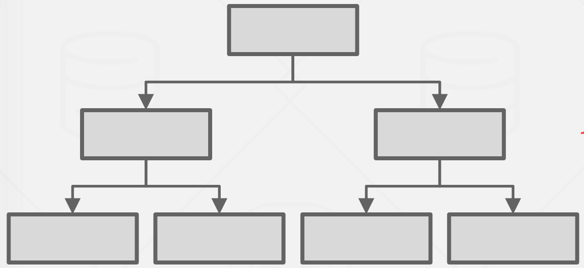
Every B+Tree node is comprised of an array of key/value pairs.

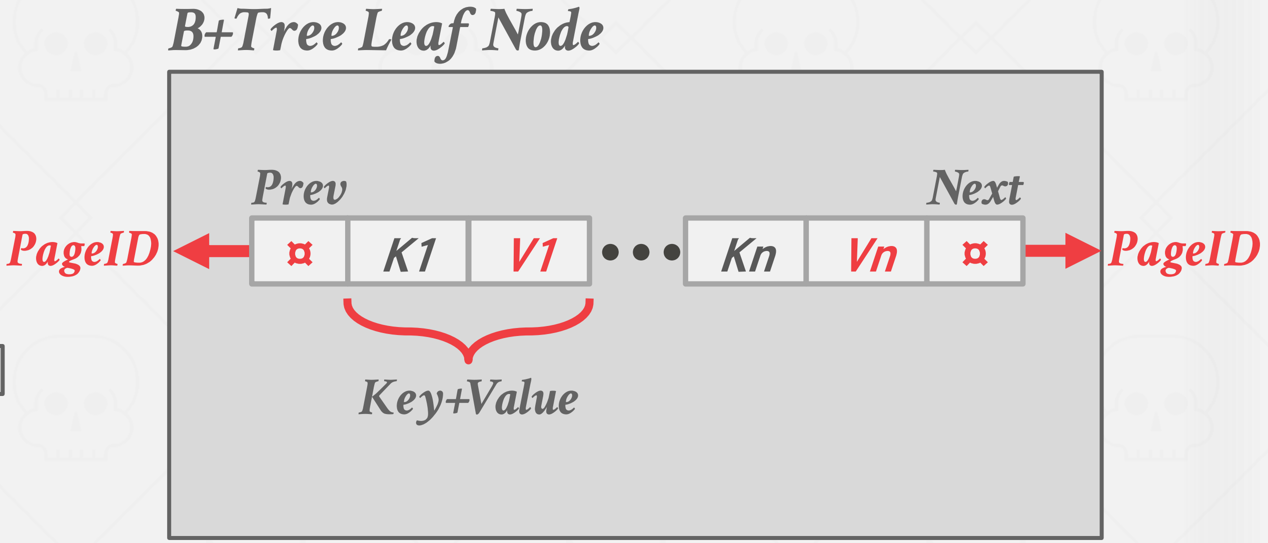
→ The keys are derived from the attribute(s) that the index is based on.

→ The values will differ based on whether the node is classified as an **inner node** or a **leaf node.**

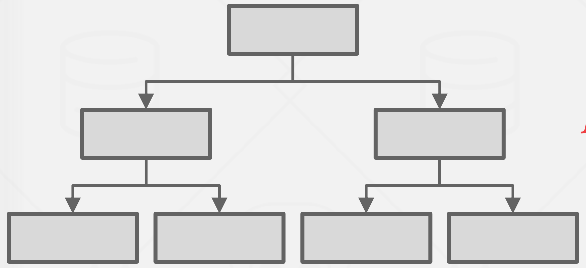
The arrays are (usually) kept in sorted key order.

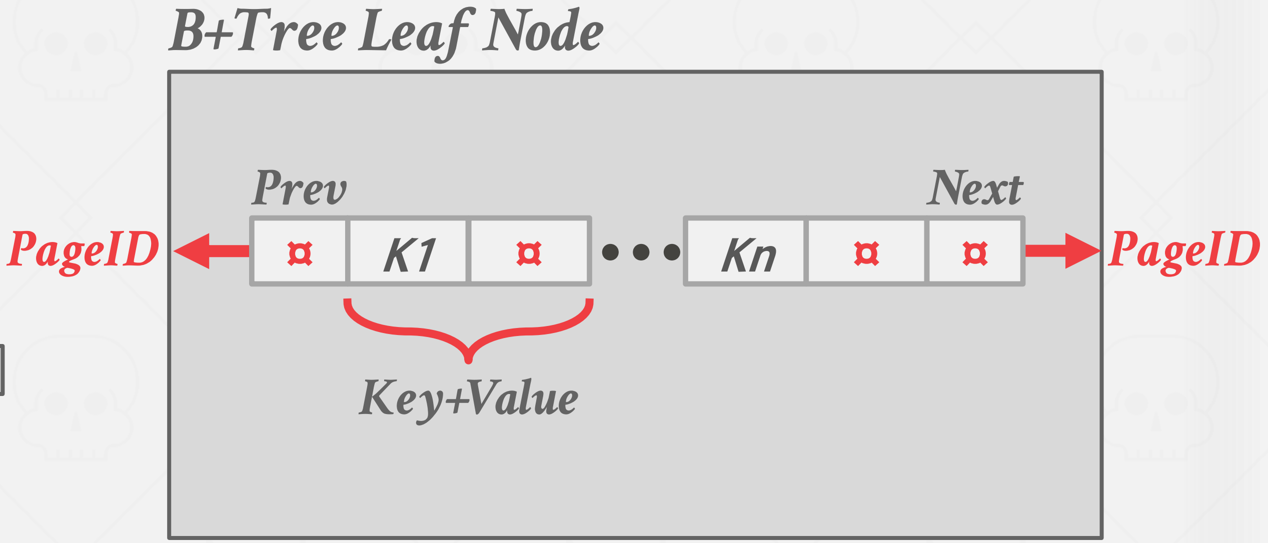
## B+TREE LEAF NODES



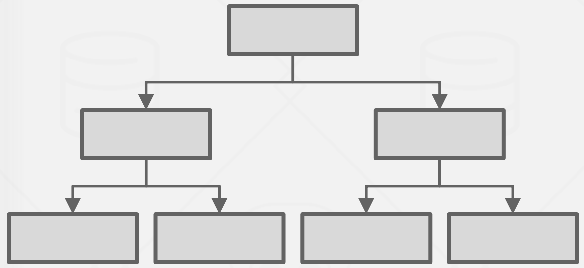


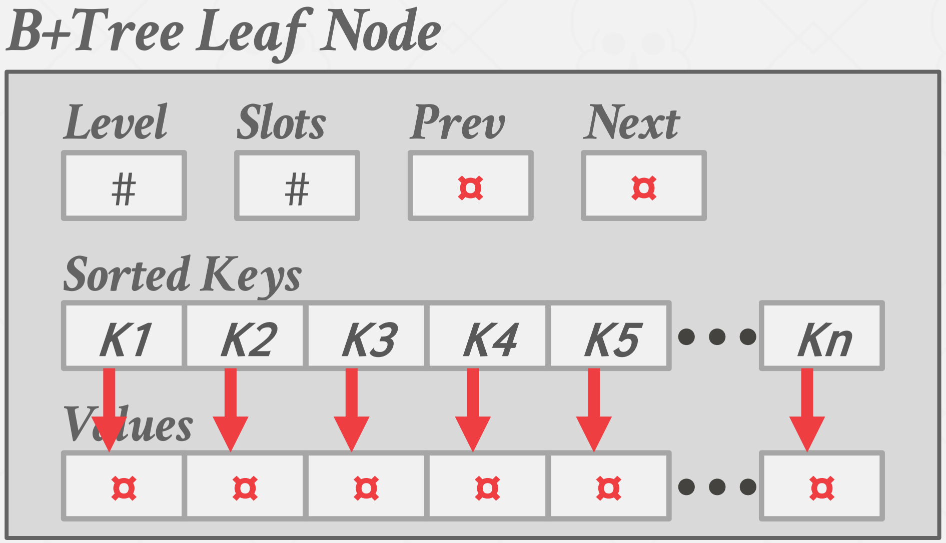
# B+TREE LEAF NODES





## B+TREE LEAF NODES





## B-TREE VS. B+TREE

The original **B-Tree** from 1972 stored keys and 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 node **L**.
Insert data entry into **L** in sorted order.

If **L** has enough space, done!

Otherwise, split **L** keys 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 – DELETE

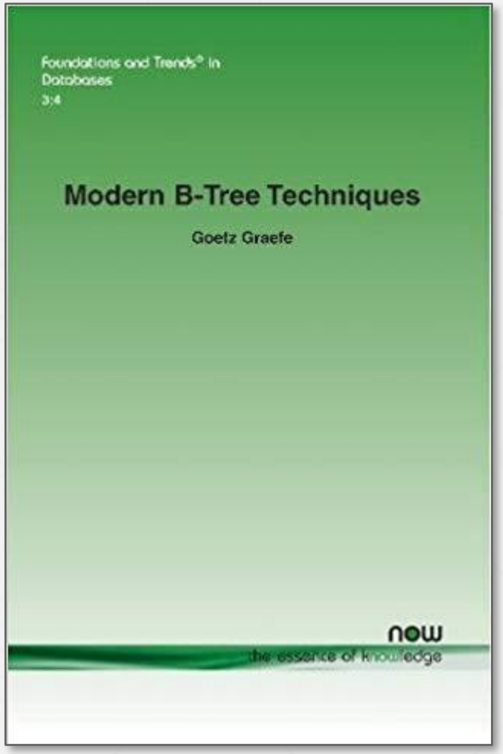
Start at root, find leaf **L** where entry belongs.
Remove the entry.
If **L** is at least half-full, done!
If **L** has only **M/2-1** entries,
→ Try to re-distribute, borrowing from sibling (adjacent node with same parent as **L**).
→ If re-distribution fails, merge **L** and sibling.

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

Source: Chris Re

## B+TREE DESIGN CHOICES

Node Size
Merge Threshold
Variable-Length Keys
Intra-Node Search



## NODE SIZE

The slower the storage device, the larger the optimal node size for a B+Tree.
→ HDD: ~1MB
→ SSD: ~10KB
→ In-Memory: ~512B

Optimal sizes can vary depending on the workload
→ Leaf Node Scans vs. Root-to-Leaf Traversals

## MERGE THRESHOLD

Some DBMSs do not always merge nodes when they are half full.

Delaying a merge operation may reduce the amount of reorganization.

It may also be better to just let smaller nodes exist and then periodically rebuild entire 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 index can vary.
→ Requires careful memory management.

**Approach #3: Padding**

→ Always pad the key to be max length of the key type.

**Approach #4: Key Map / Indirection**

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

# INTRA-NODE SEARCH

**Approach #1: Linear**

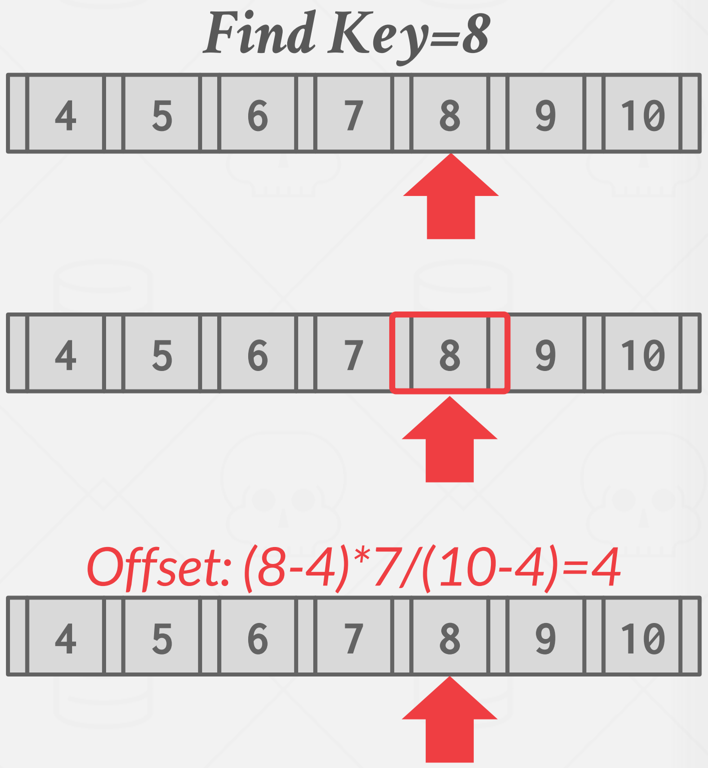
→ Scan node keys from beginning to end.

**Approach #2: Binary**

→ Jump to middle key, pivot left/right depending on comparison.

**Approach #3: Interpolation**

→ Approximate location of desired key based on known distribution of keys.



## OPTIMIZATIONS

Prefix Compression

Deduplication

Suffix Truncation

Pointer Swizzling

Bulk Insert

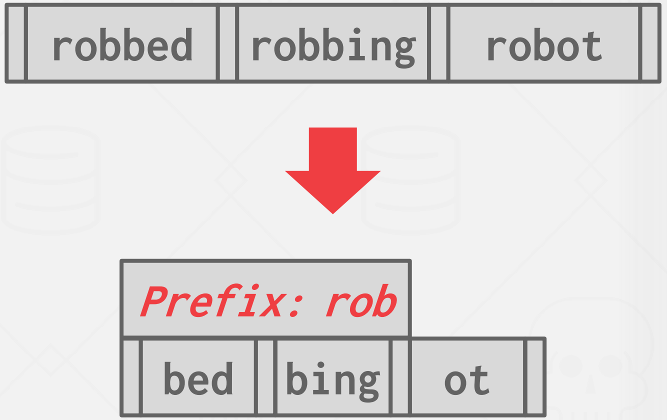
Buffer Updates

Many more…

## PREFIX COMPRESSION

Sorted keys in the same leaf node are likely to have the same prefix.

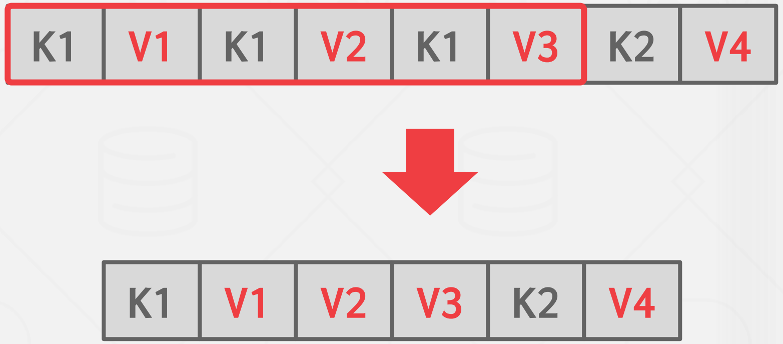
Instead of storing the entire key each time, extract common prefix and store only unique suffix for each key.
→ Many variations.



## DEDUPLICATION

Non-unique indexes can end up storing multiple copies of the same key in leaf nodes.

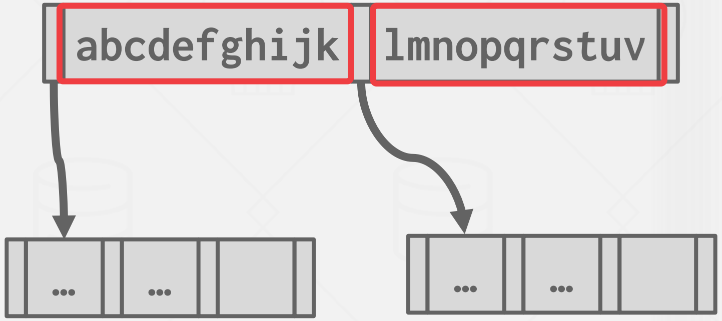
The leaf node can store the key once and then maintain a list of tuples with that key (similar to what we discussed for hash tables).



## SUFFIX TRUNCATION

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

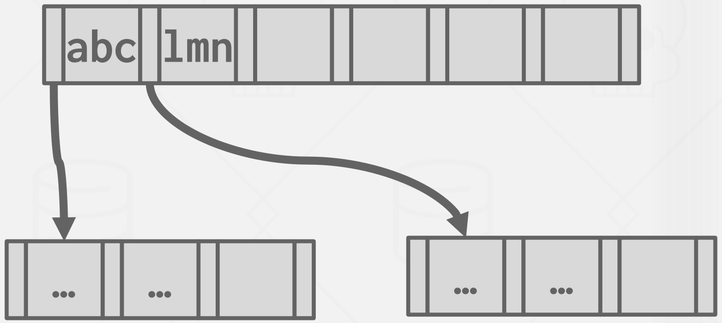
Store a minimum prefix that is needed to correctly route probes into the index.



## SUFFIX TRUNCATION

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

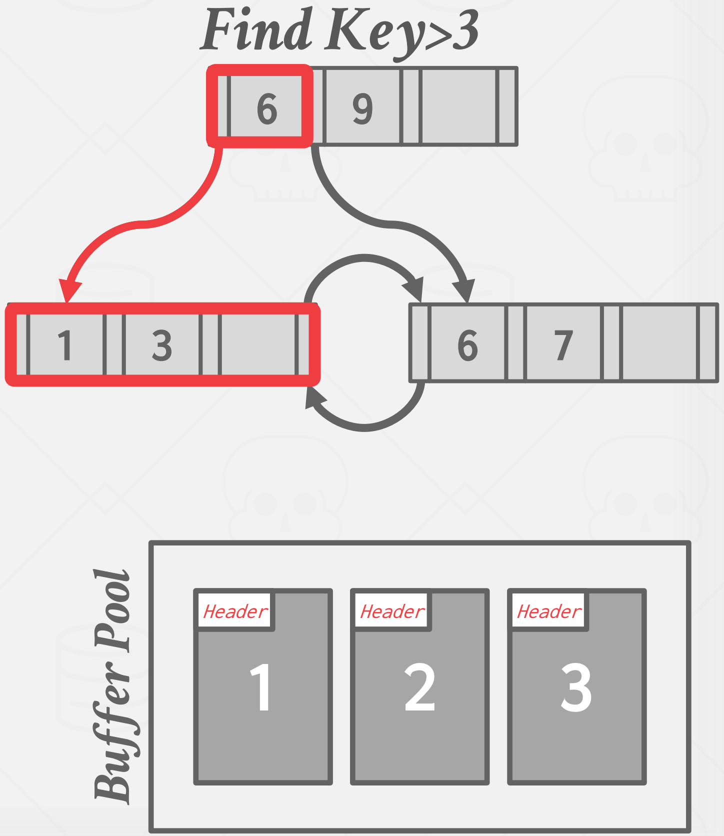
Store a minimum prefix that is needed to correctly route probes into the index.



## POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

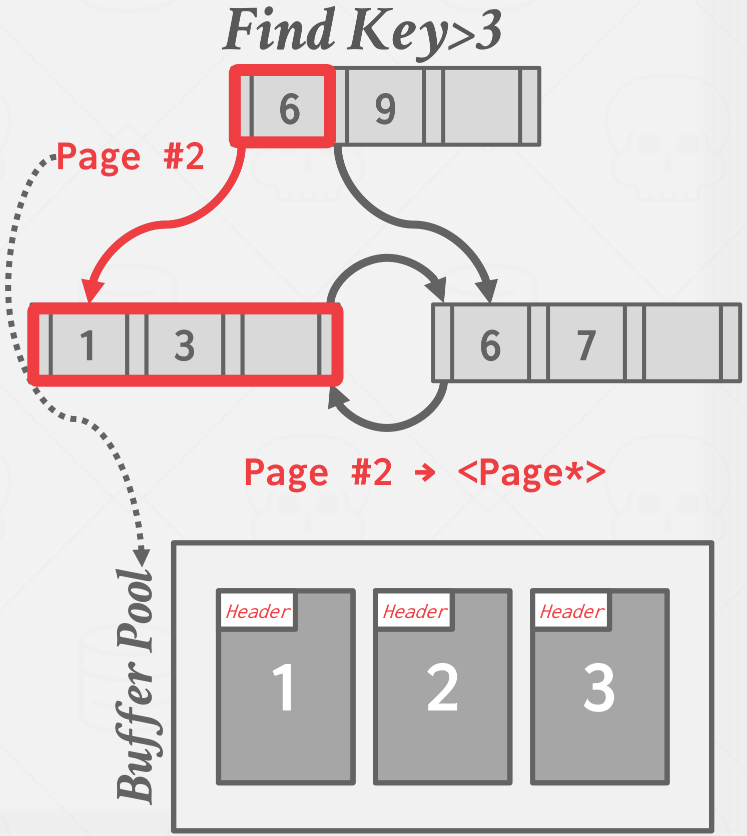
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



## POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

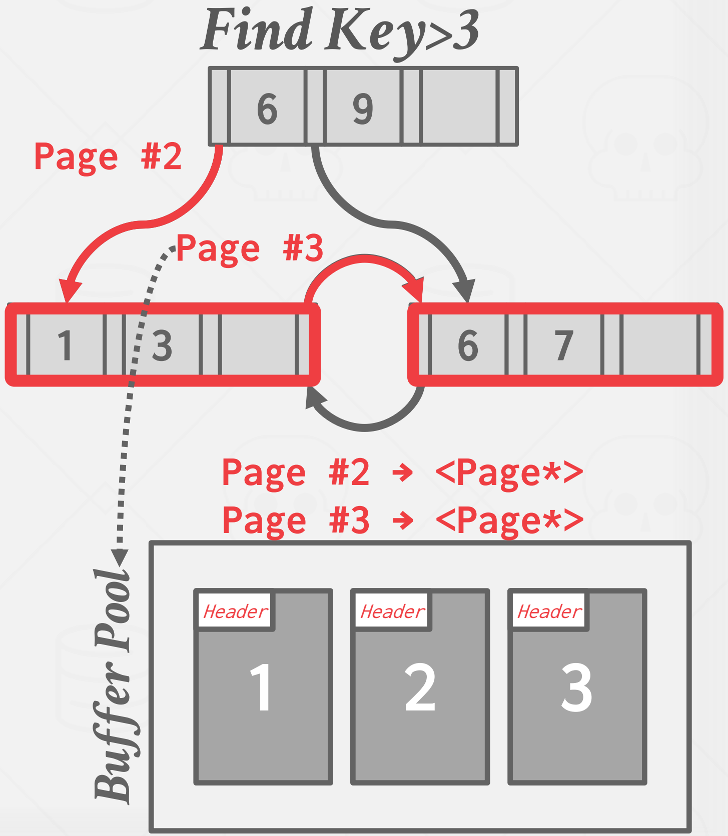
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



## POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

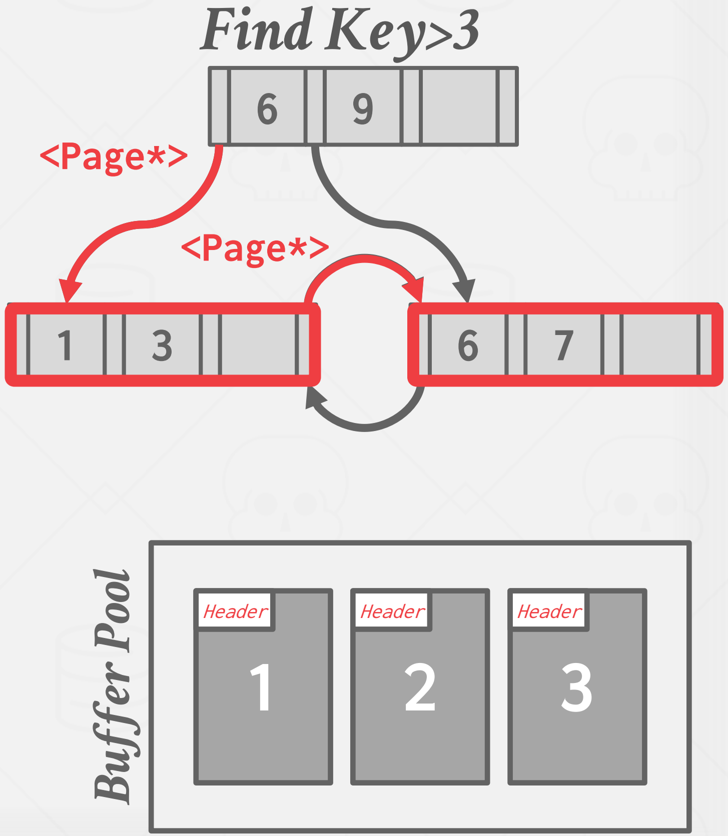
If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



## POINTER SWIZZLING

Nodes use page ids to reference other nodes in the index. The DBMS must get the memory location from the page table during traversal.

If a page is pinned in the buffer pool, then we can store raw pointers instead of page ids. This avoids address lookups from the page table.



## BULK INSERT

The fastest way to build a new B+Tree for an existing table 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