CS 4440 A

Emerging Database Technologies

Lecture 7
01/29/25

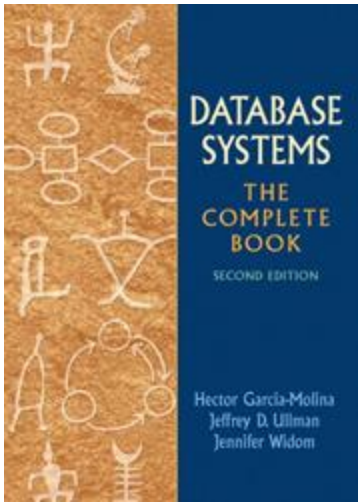
Agenda

1. B+-Trees
2. B+-Trees Cost Model
3. Log-structured Merge Tree

Reading Materials

Database Systems: The Complete Book (2nd edition)

- Chapter 14.2: B-Tree
- Chapter 14.4: Multidimensional Indexes



Acknowledgement: The following slides have been adapted from EE477 (Database and Big Data Systems) taught by Steven Whang, CS145 (Intro to Big Data Systems) taught by Peter Bailis, and CS 6530 (Advanced Database Systems) taught by Prashant Pandey.

1. B+-Tree

B+ Trees Overview

Search trees

- B does not mean binary!
- More general index structure that is commonly used in commercial DBMS's

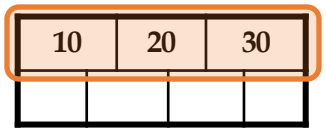
Idea in B Trees:

- Balanced, height adjusted tree
- Stores data (keys and values) in all nodes (both internal and leaf)
- Leaf nodes are independent; no connections between them

Idea in B+ Trees:

- Stores data only in leaf nodes; make leaves into a linked list (for range queries)
- Most popular variant

B+ Tree Basics

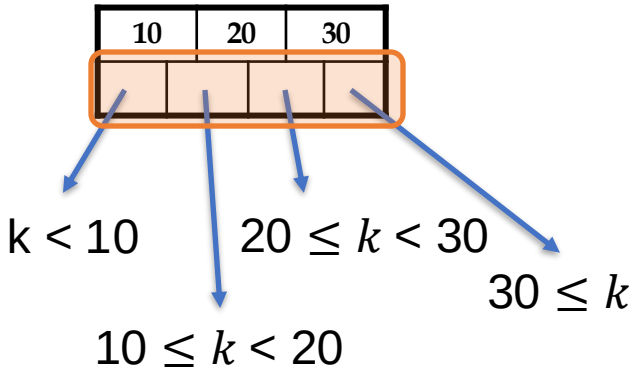


Parameter **n** = the degree

Each non-leaf (“interior”) node has between (around) $\frac{n}{2}$ and $\mathbf{n}$ keys*

*except for root node, which can have between 1 and n keys

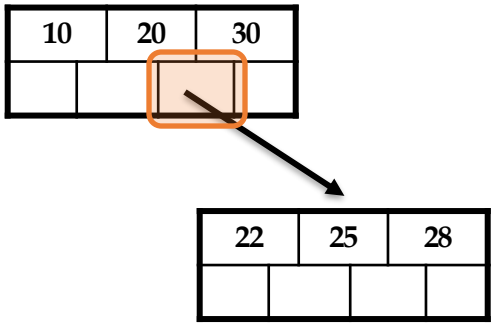
B+ Tree Basics



The n keys in a node define n+1 ranges

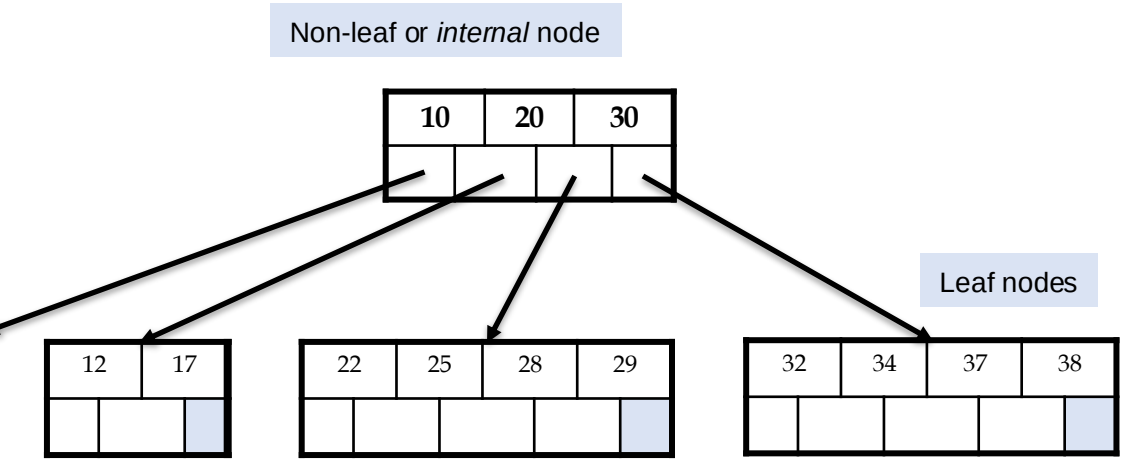
B+ Tree Basics

Non-leaf or *internal* node



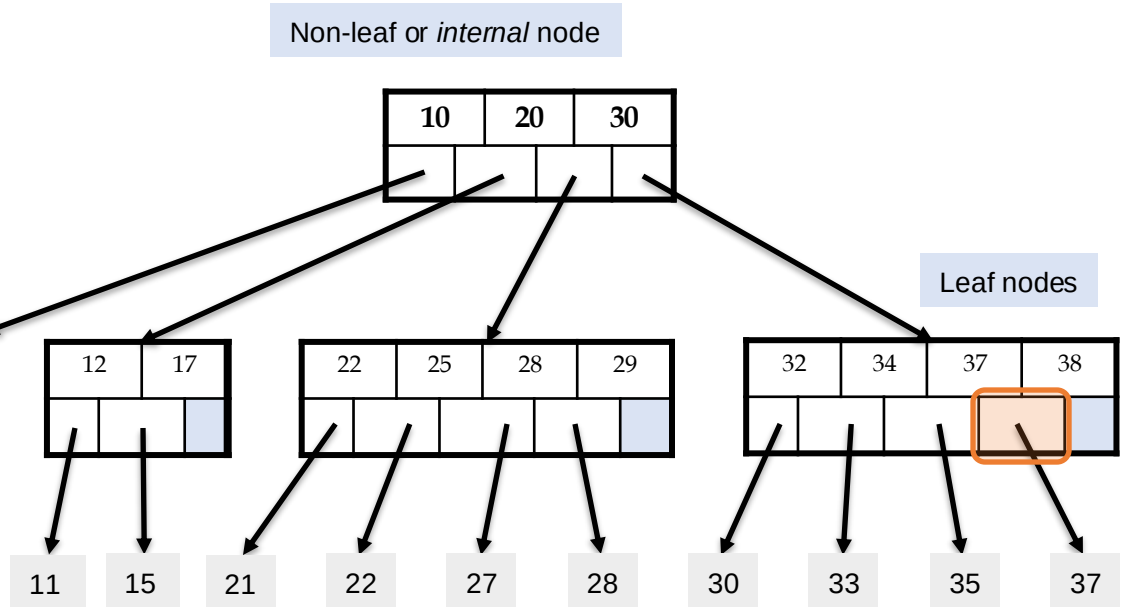
For each range, in a non-leaf node, there is a **pointer** to another node with keys in that range

B+ Tree Basics



Leaf nodes also have between n/2 and n keys, and are different in that:

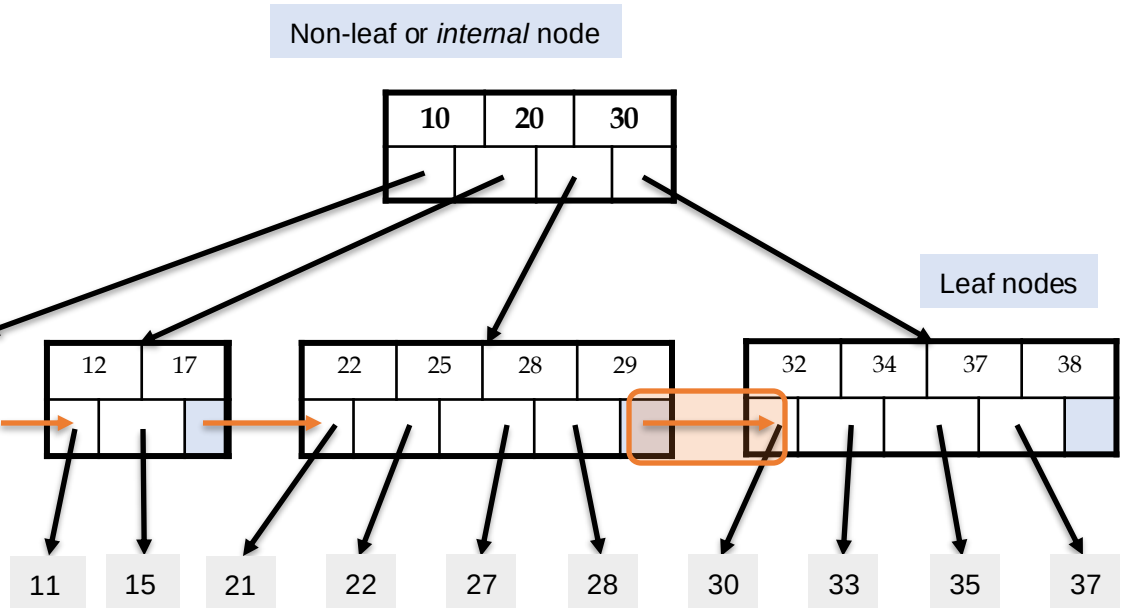
B+ Tree Basics



Leaf nodes also have between n/2 and n keys, and are different in that:

Their key slots contain pointers to data records

B+ Tree Basics

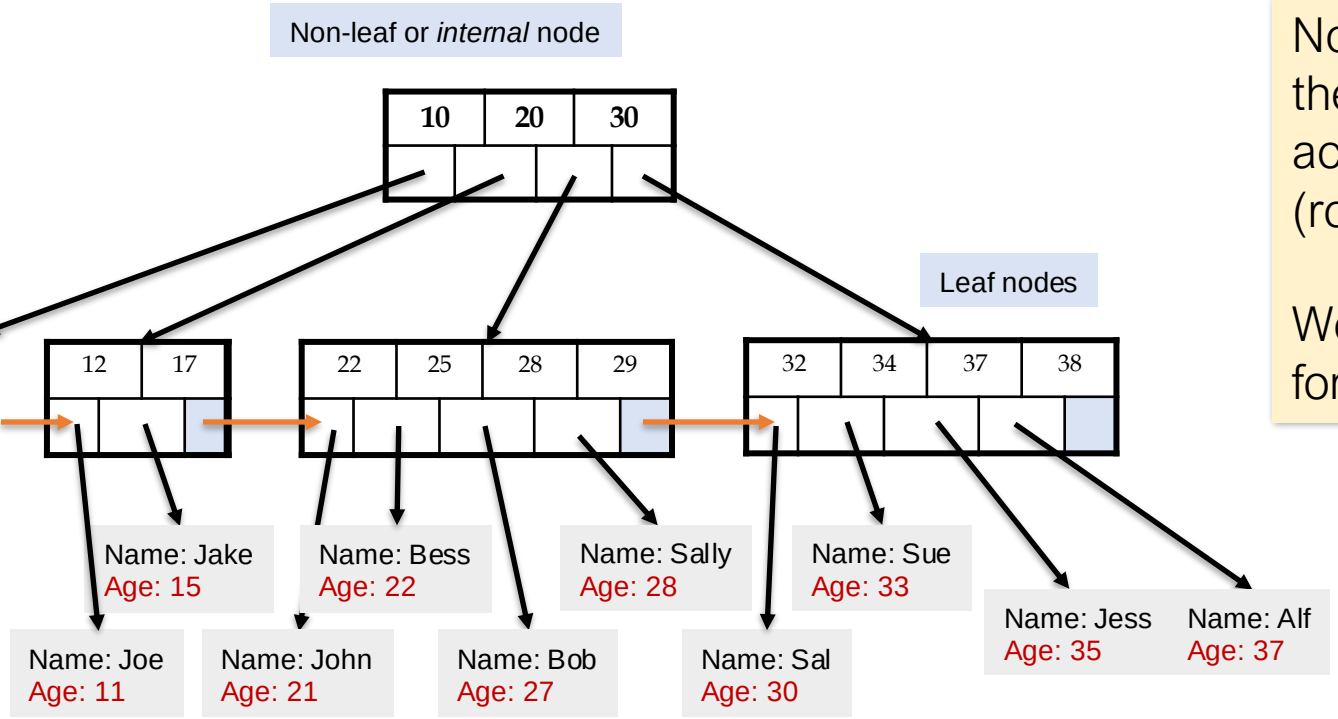


Leaf nodes also have between n/2 and n keys, and are different in that:

Their key slots contain pointers to data records

They contain a pointer to the next leaf node as well, for **faster sequential traversal**

B+ Tree Basics



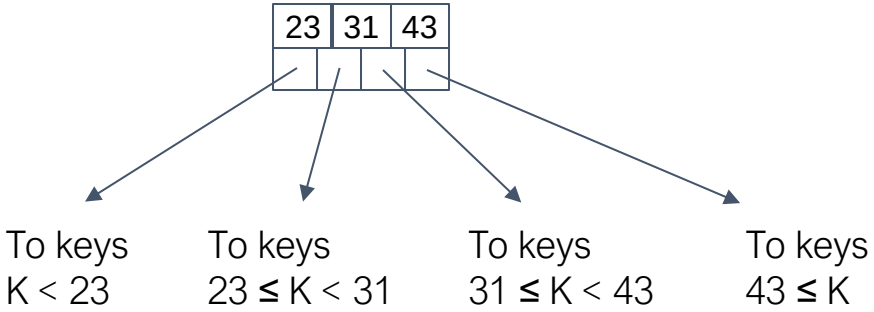
Note that the pointers at the leaf level will be to the actual data records (rows).

We might truncate these for simpler display…

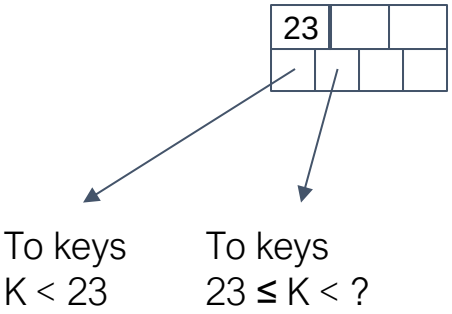
B+ Tree occupancy requirement: interior nodes

n = 3

Full



Minimal

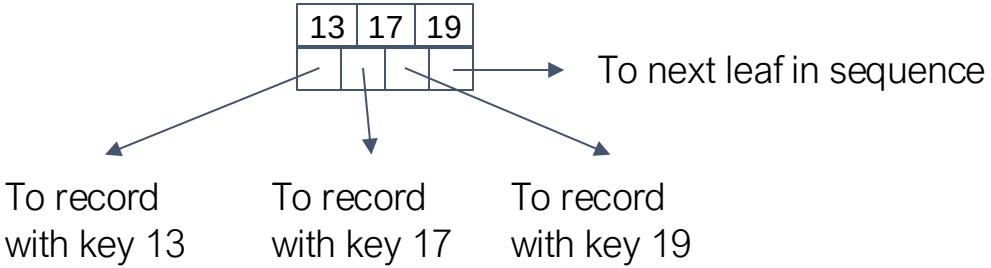


At least half of the pointers much be used

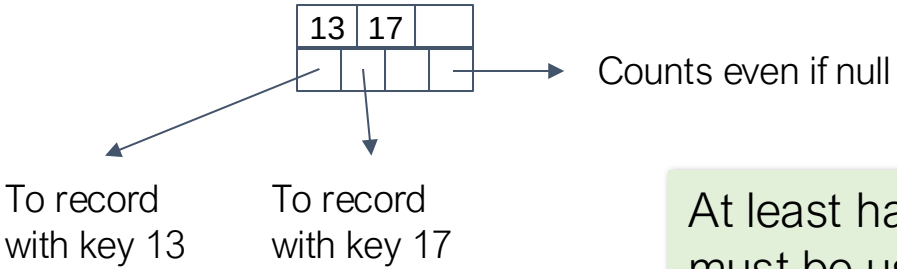
B+ Tree occupancy requirement: leaf nodes

n = 3

Full



Minimal



At least half of the keys must be used

Nodes must be “full enough”

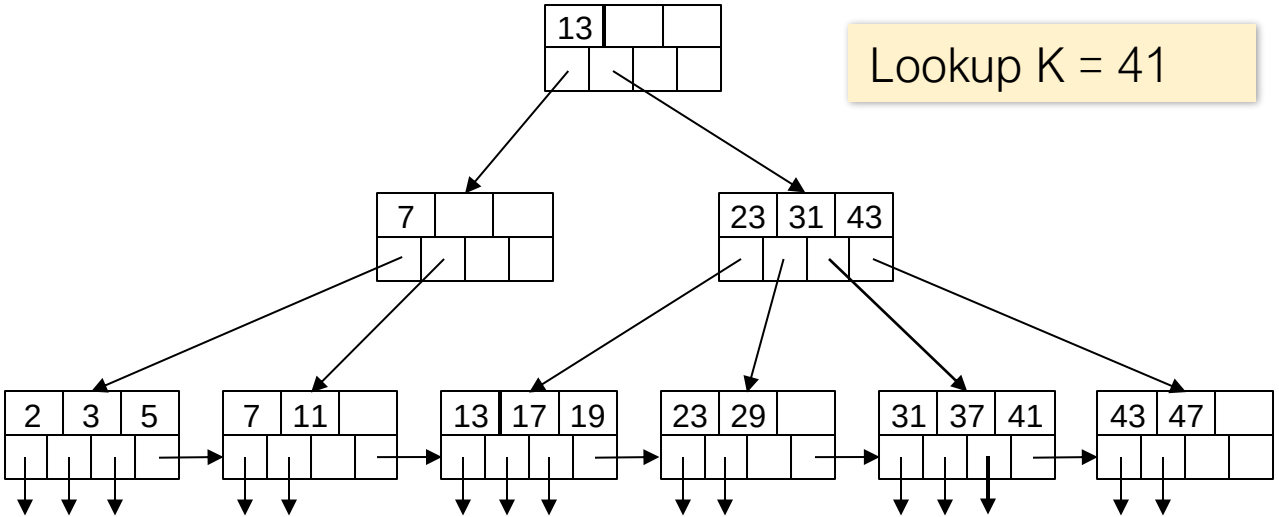
Node type	Min. # pointers	Max. # pointers	Min. # keys	Max. # keys
Interior	⌈(n + 1) / 2⌉	n + 1	⌈(n + 1) / 2⌉ - 1	n
Leaf	⌊(n + 1) / 2⌋ **	n + 1	⌊(n + 1) / 2⌋	n
Root	2 *	n + 1	1	n

* Exception: If there is only one record in the B-tree, there is one pointer in the root

** Not including the next leaf pointer

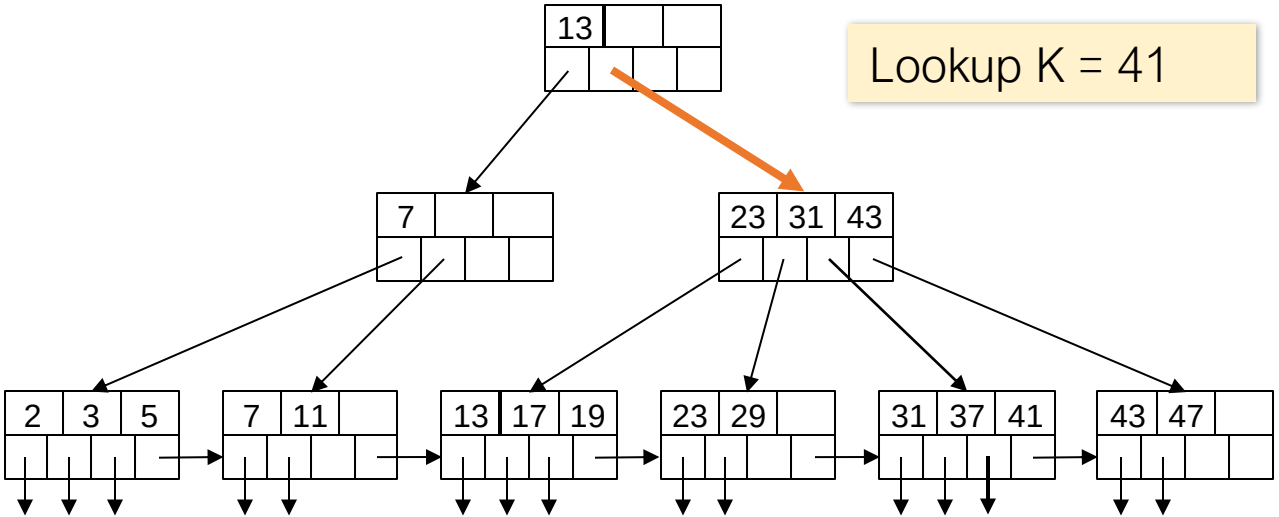
B+ Tree: Lookup

- Search for key K recursively



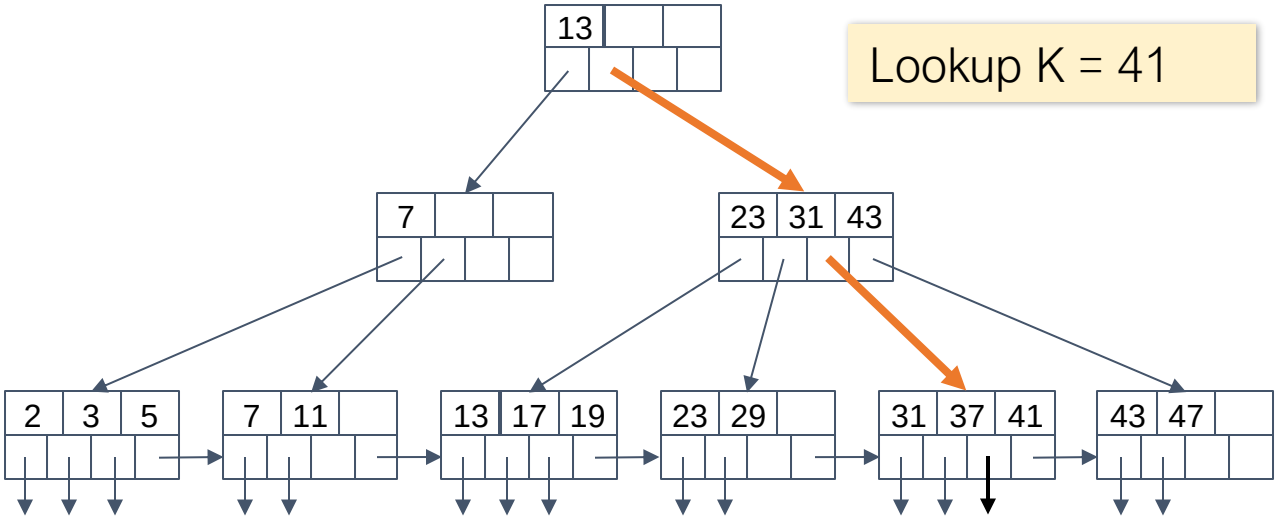
B+ Tree: Lookup

- Search for key K recursively



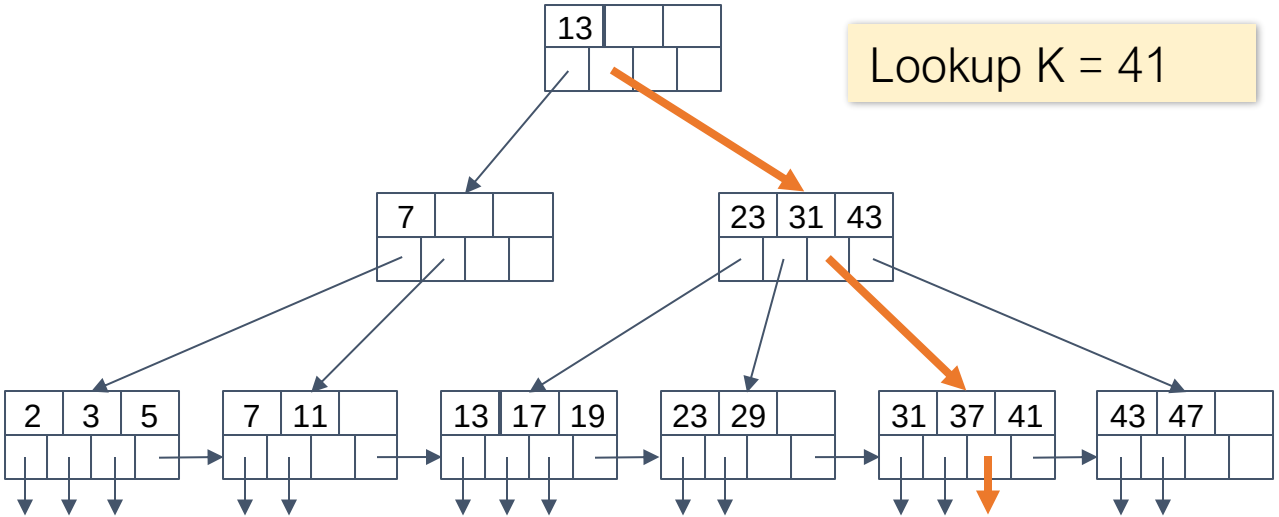
B+ Tree: Lookup

- Search for key K recursively



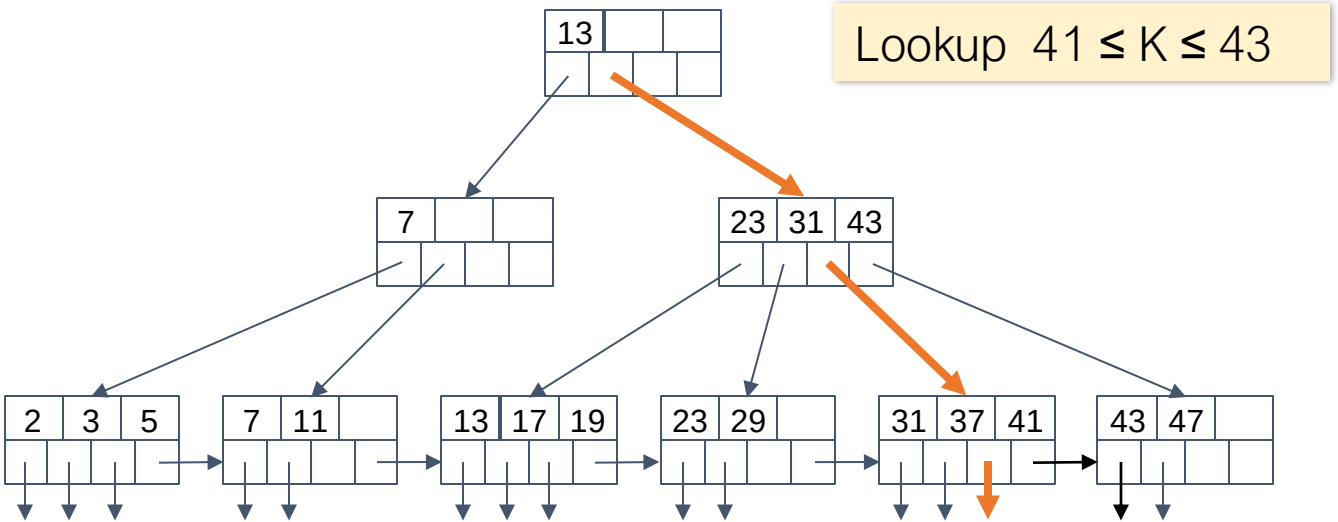
B+ Tree: Lookup

- Search for key K recursively



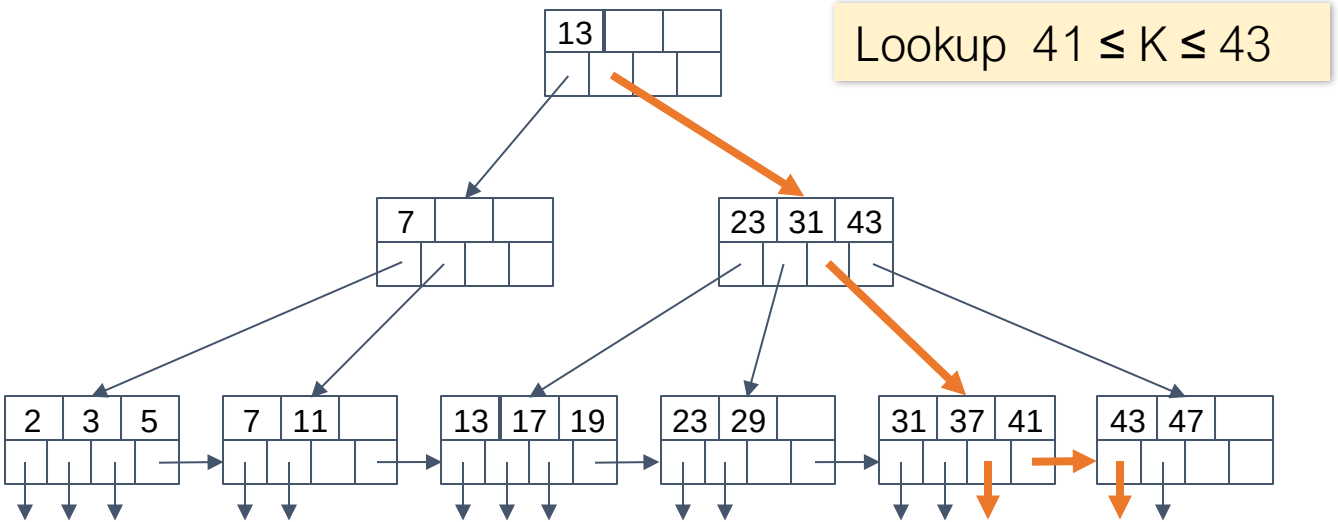
B+ Tree: Lookup

- For range query [a, b], search for key a
- Then scan leaves to right until we pass b



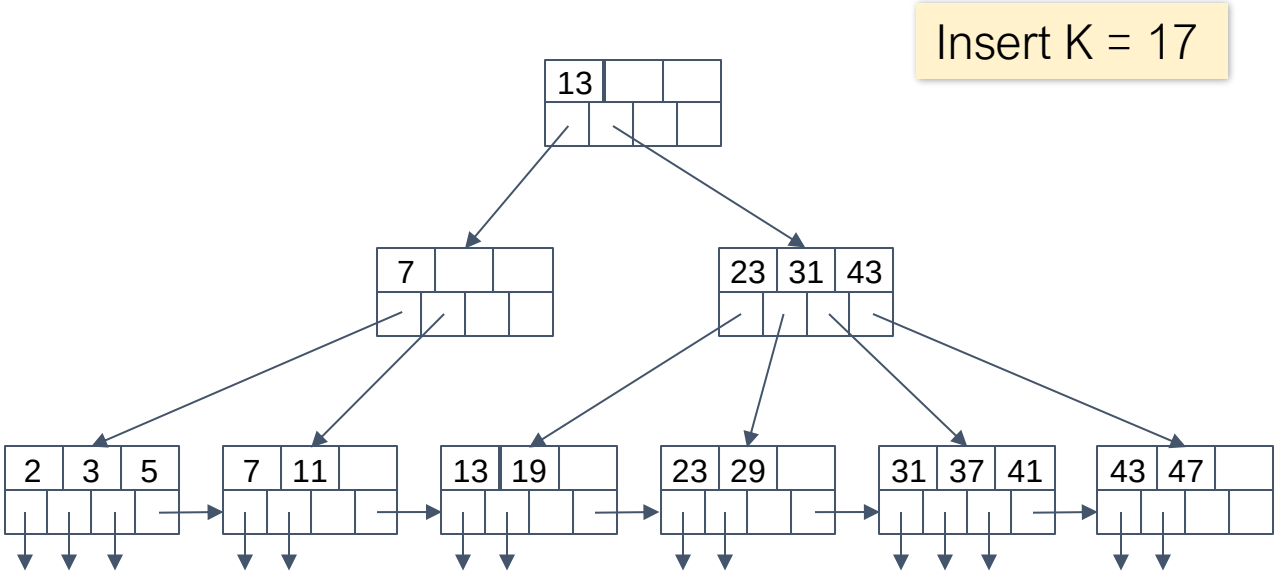
B+ Tree: Lookup

- For range query [a, b], search for key a
- Then scan leaves to right until we pass b



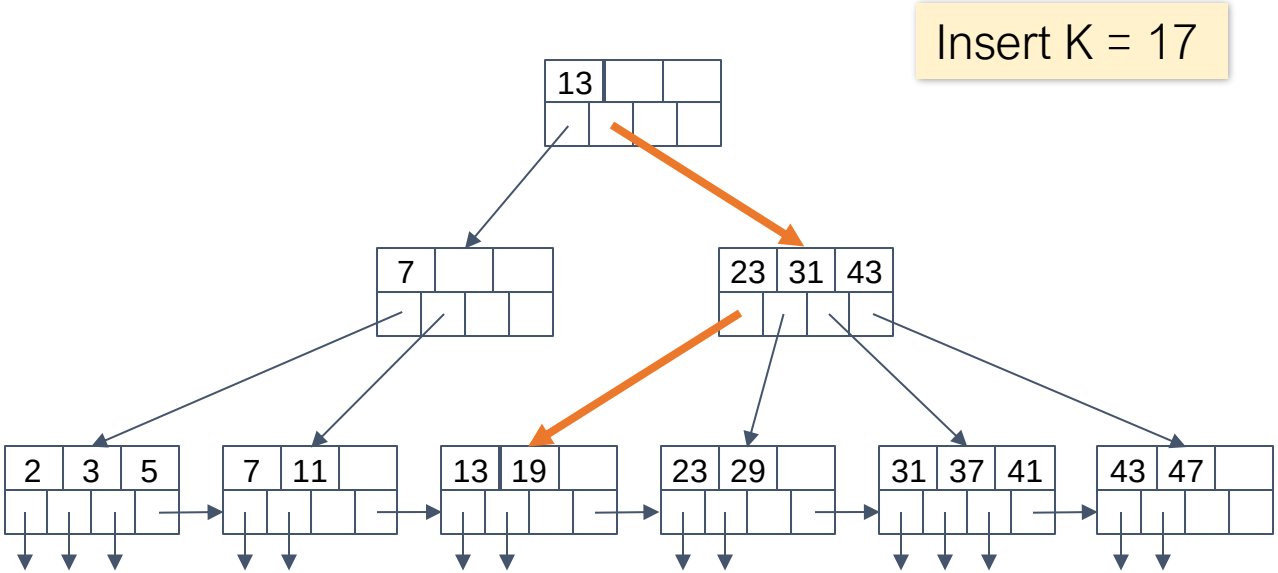
B+ Tree: Insertion

- Find place for new key in a leaf
- If there is space, put key in leaf



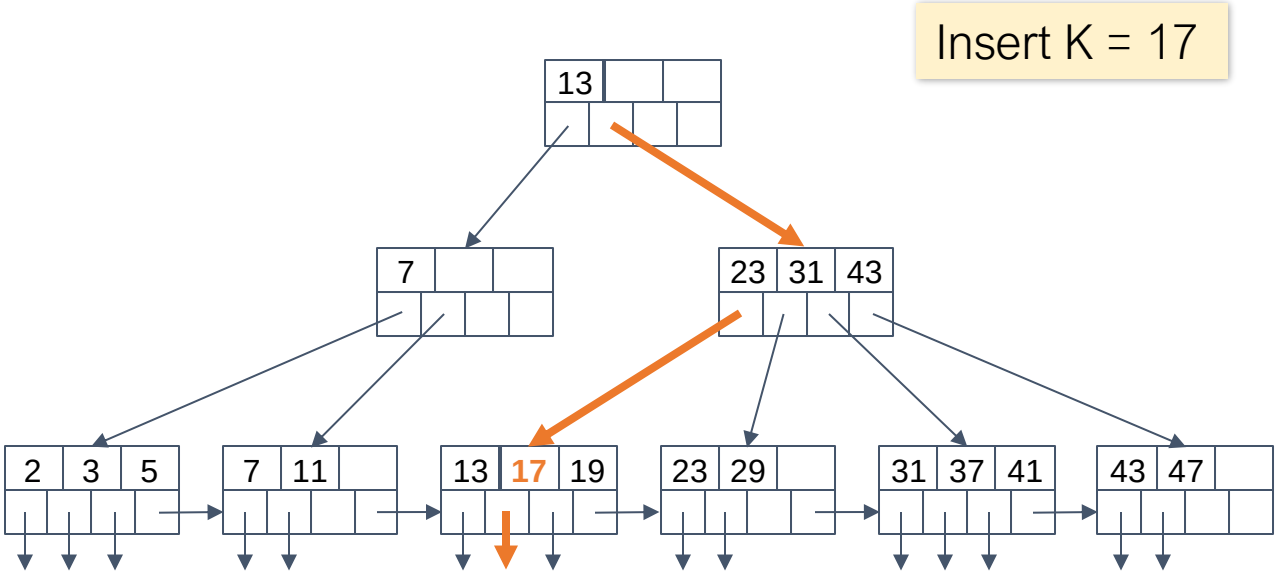
B+ Tree: Insertion

- Find place for new key in a leaf
- If there is space, put key in leaf



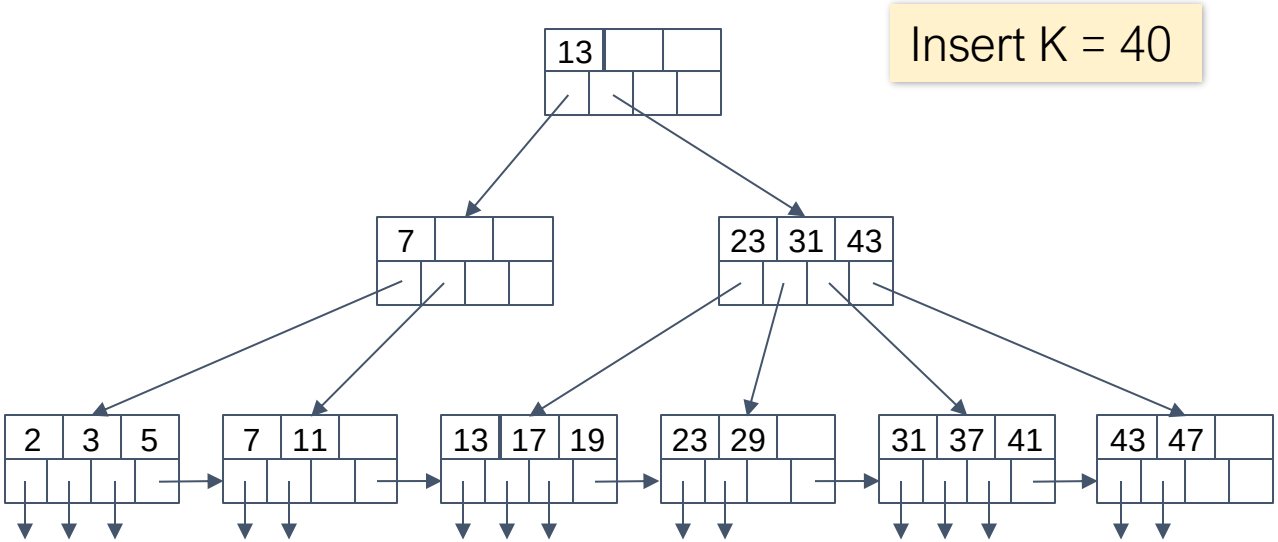
B+ Tree: Insertion

- Find place for new key in a leaf
- If there is space, put key in leaf



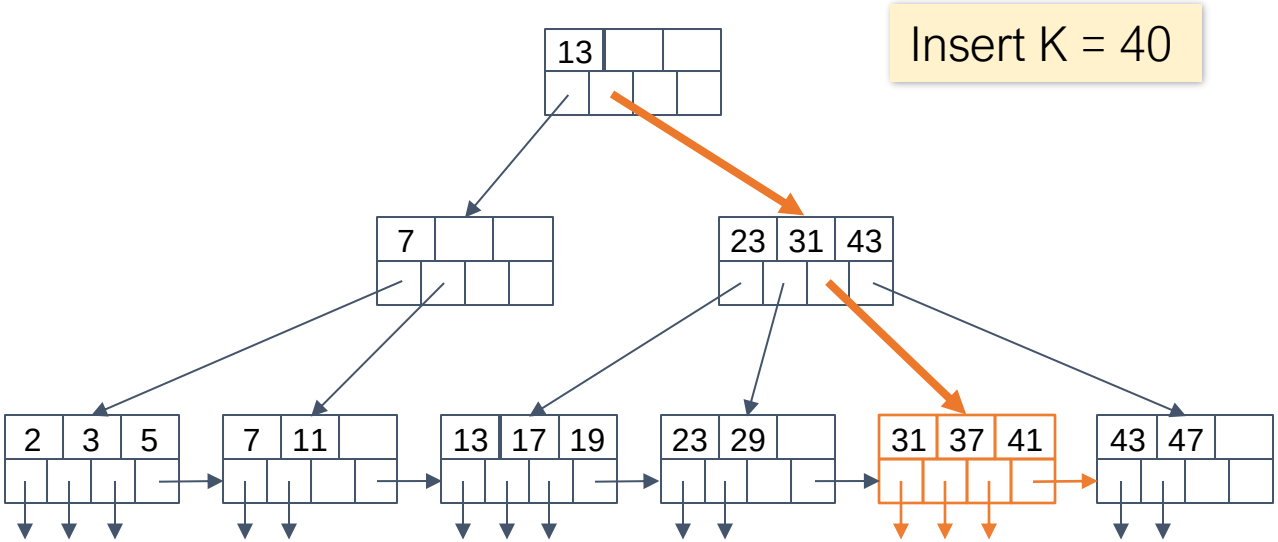
B+ Tree: Insertion

- If leaf is full, split into two and insert new pointer at a higher level recursively



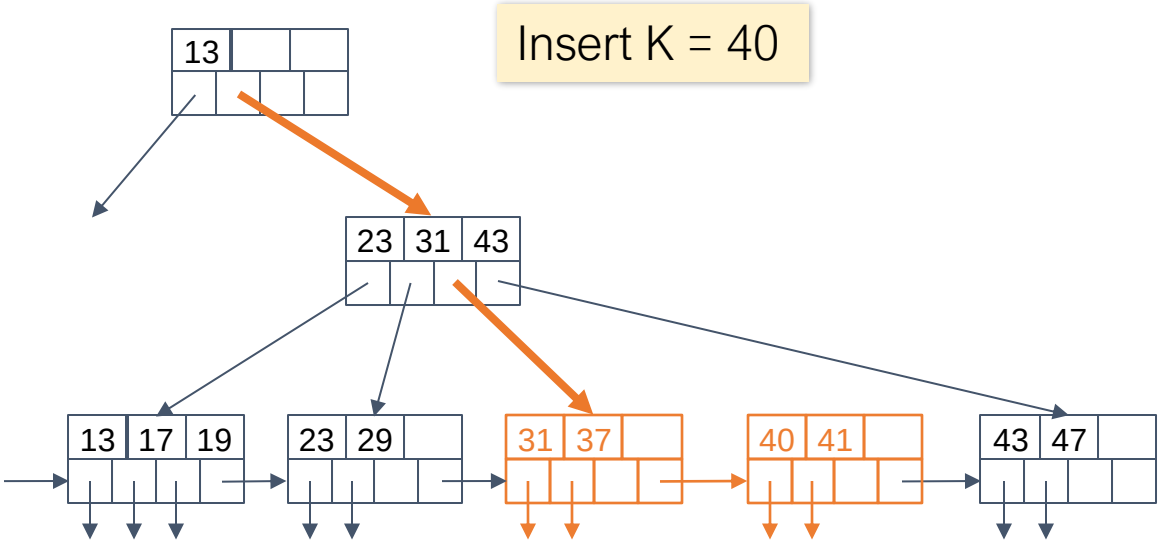
B+ Tree: Insertion

- If leaf is full, split into two and insert new pointer at a higher level recursively



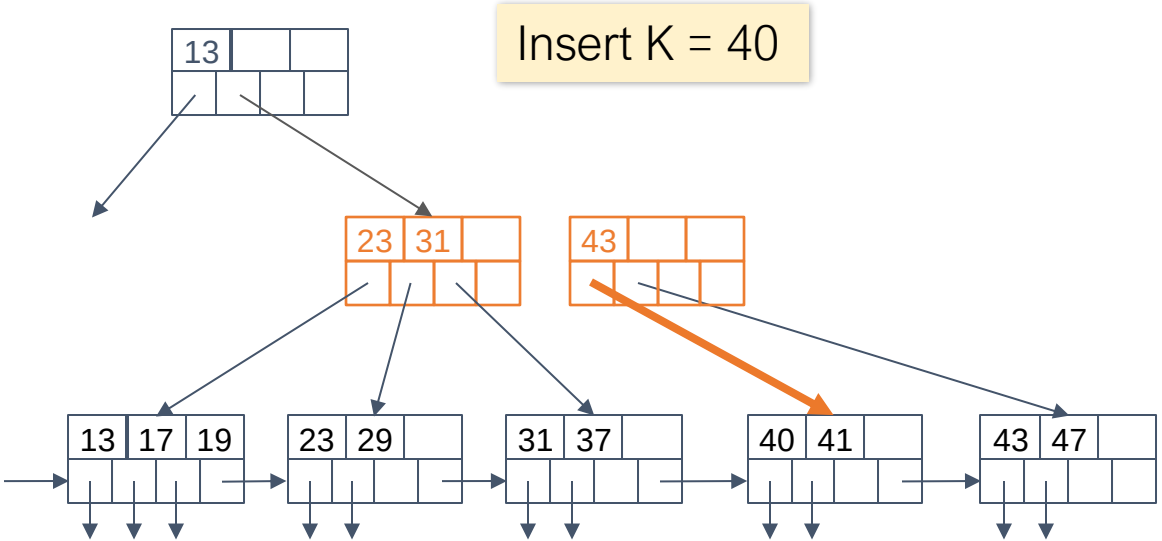
B+ Tree: Insertion

- If leaf is full, split into two and insert new pointer at a higher level recursively



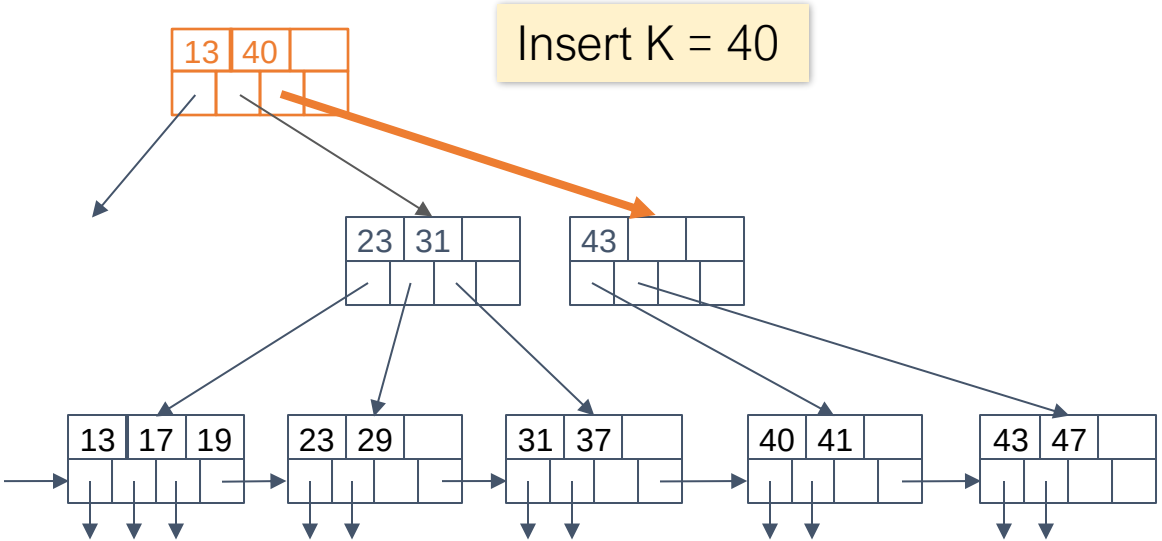
B+ Tree: Insertion

- If leaf is full, split into two and insert new pointer at a higher level recursively



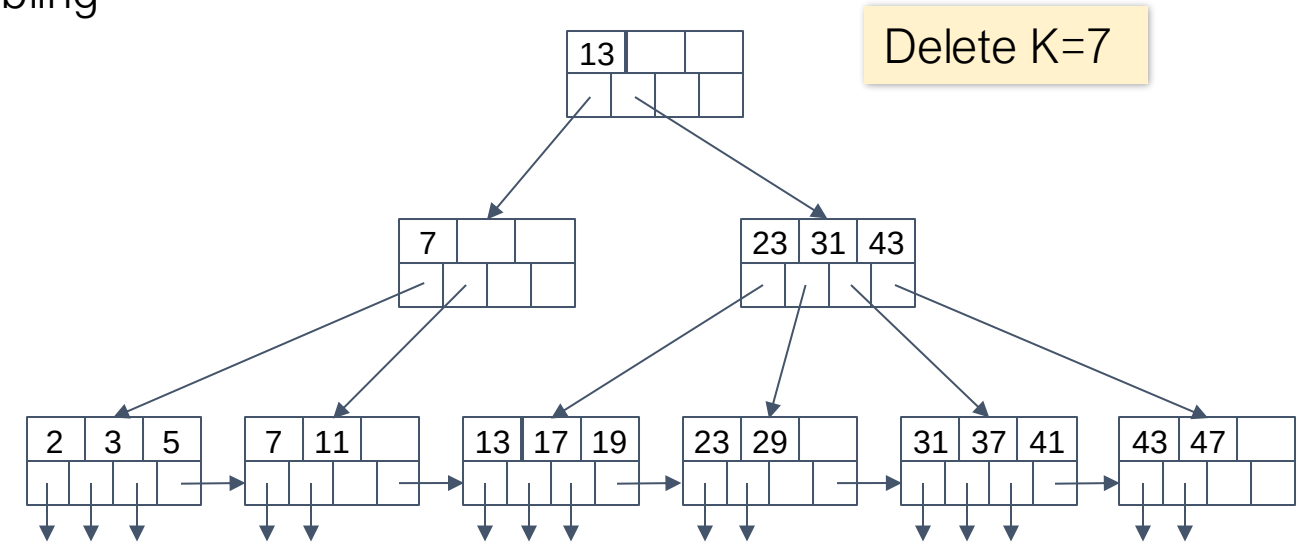
B+ Tree: Insertion

- If leaf is full, split into two and insert new pointer at a higher level recursively



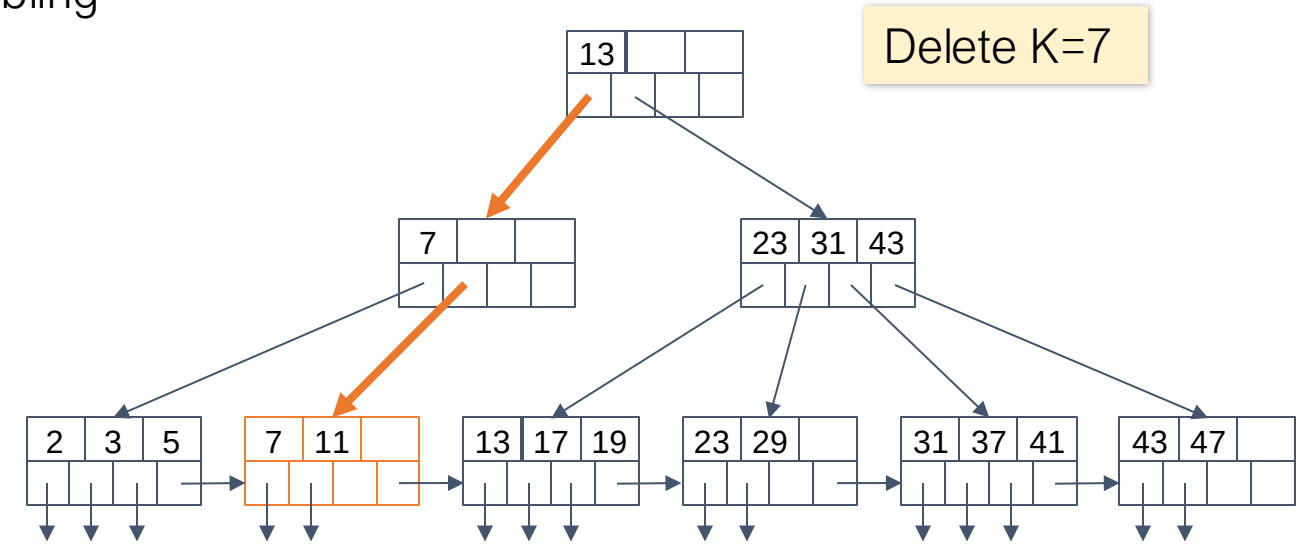
B+ Tree: Deletion

- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



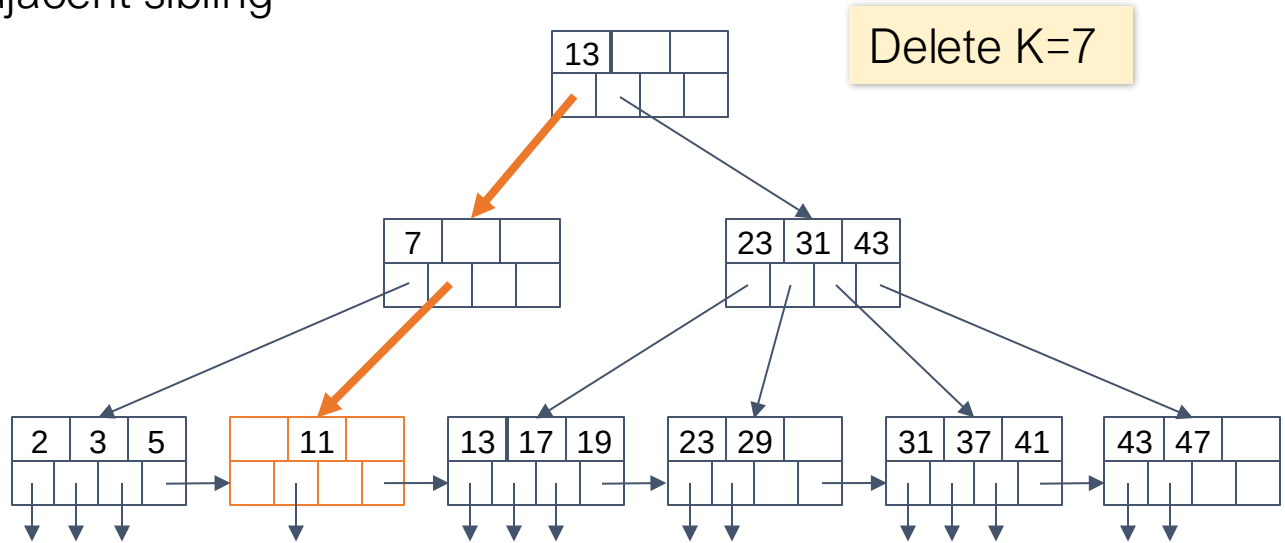
B+ Tree: Deletion

- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



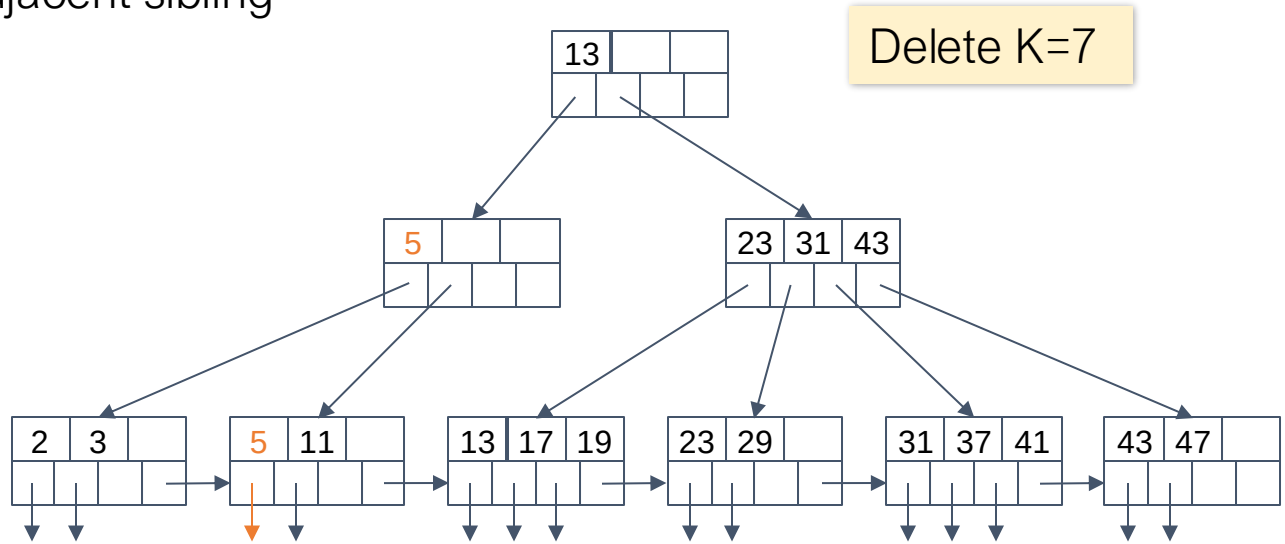
B+ Tree: Deletion

- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



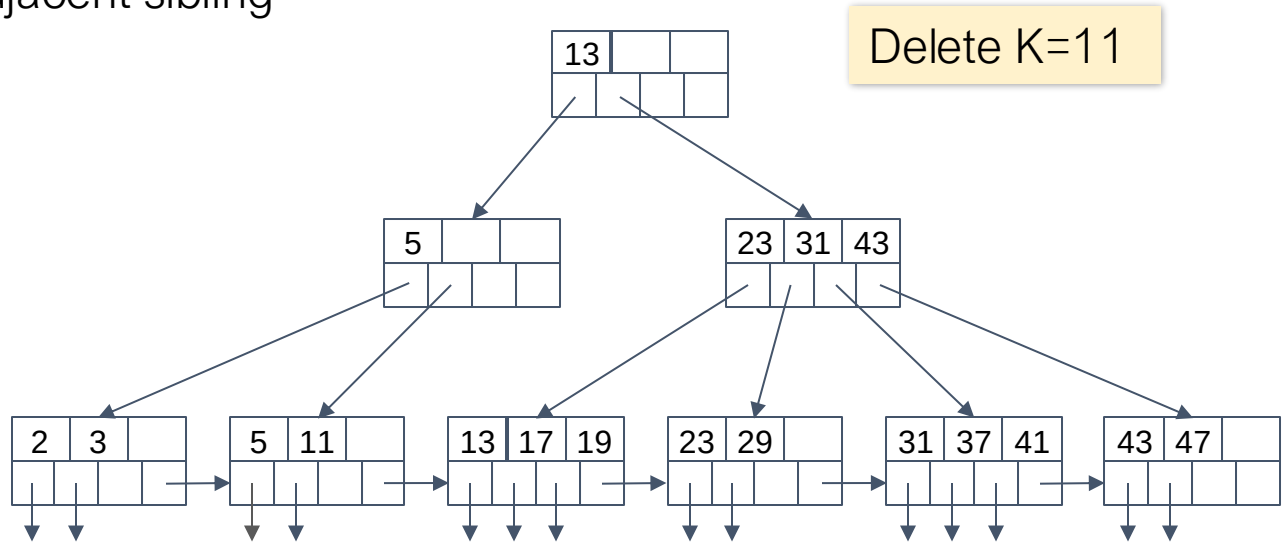
B+ Tree: Deletion

- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



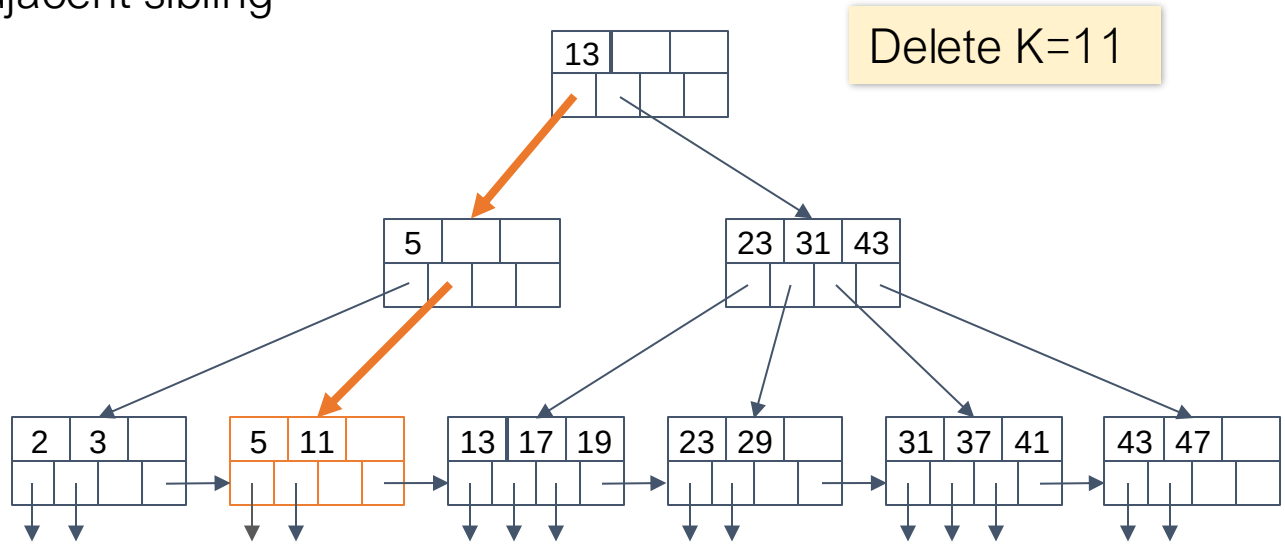
B+ Tree: Deletion

- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



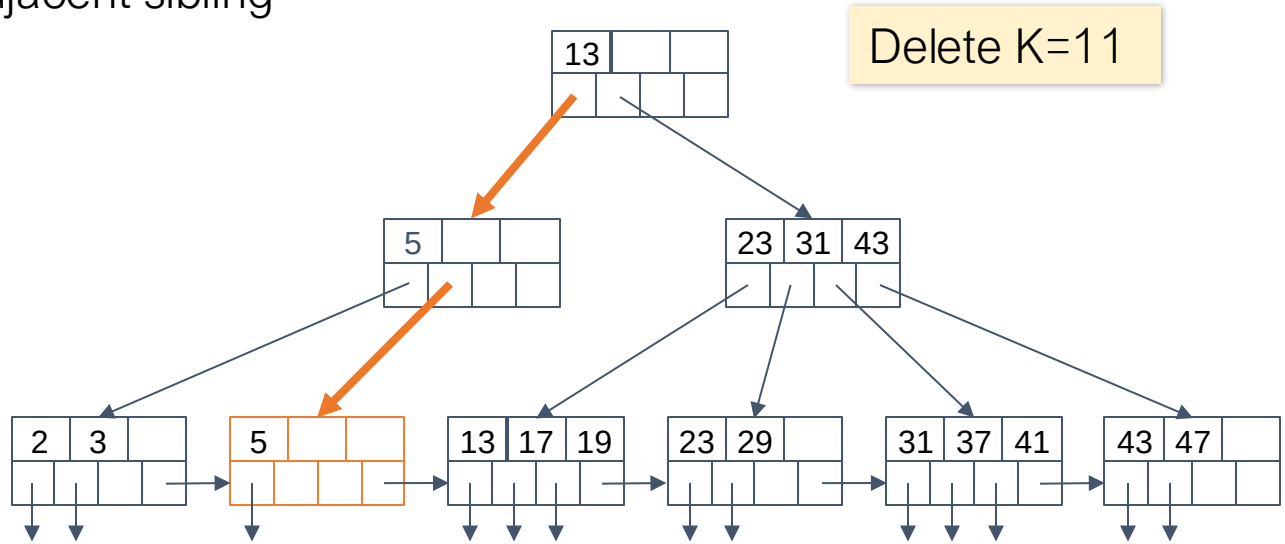
B+ Tree: Deletion

- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



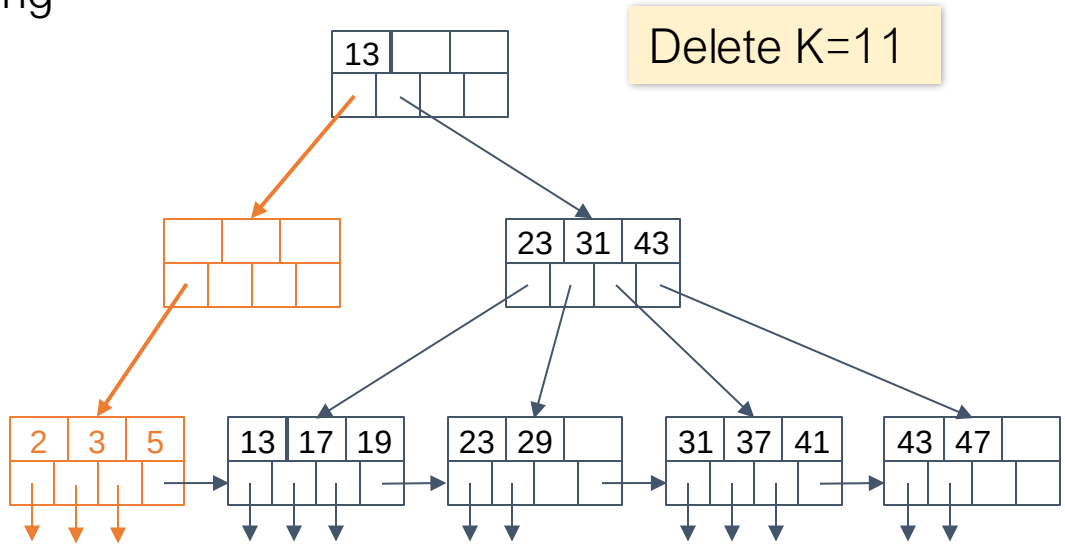
B+ Tree: Deletion

- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



B+ Tree: Deletion

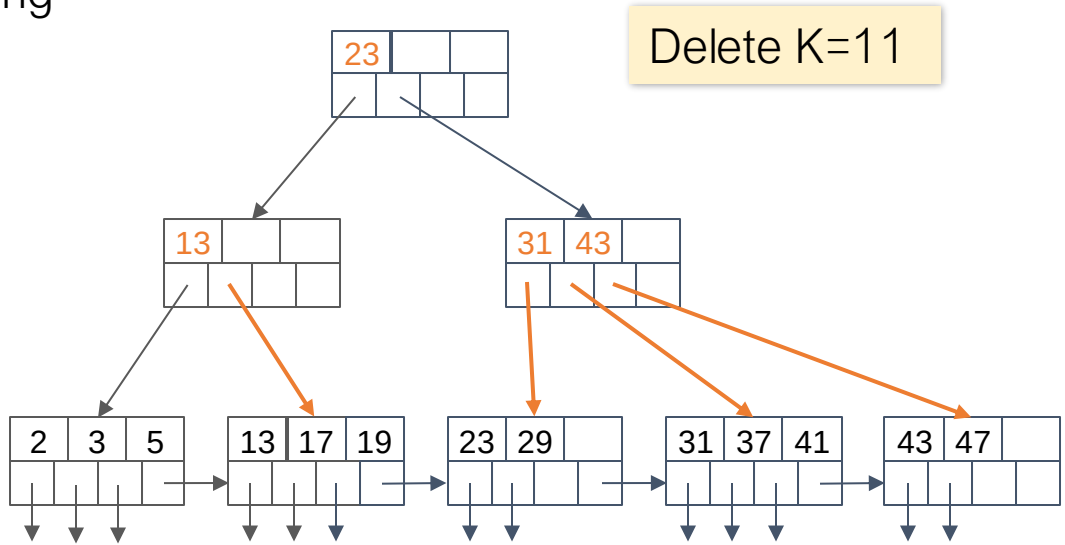
- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



B+ Tree: Deletion

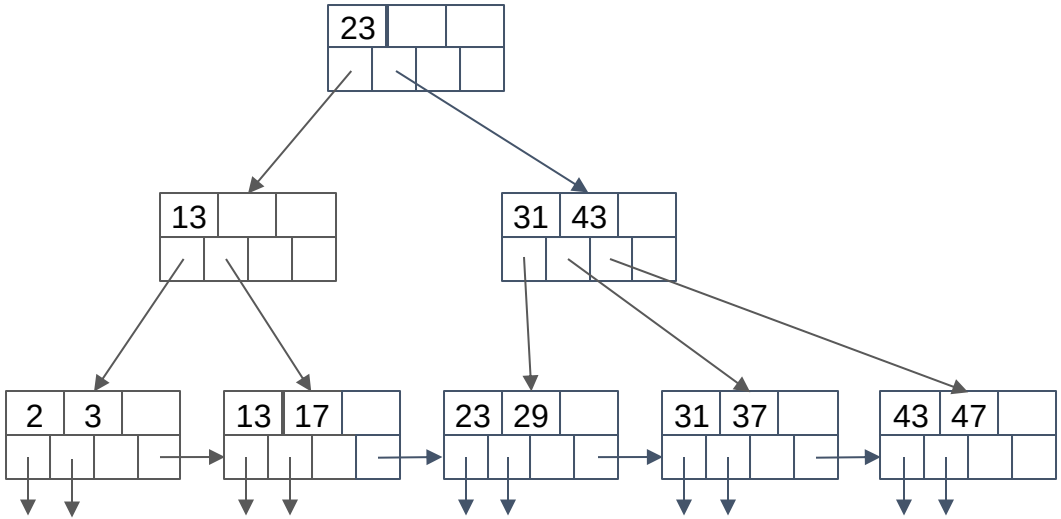
In practice, coalescing is sometimes not implemented because 1) it is hard to implement and 2) the tree will probably grow again

- Delete the key pointer from a leaf
- If the node contains too few pointers, take a pointer from or merge with adjacent sibling



In-class Exercise

- Delete K = 31



2. B+-Tree cost model

B+ Tree: High Fanout = Smaller & Lower IO

So why does B+ tree work?

As compared to binary search trees, B+ Trees have **high *fanout*** (*between d+1 and 2d+1*)

This means that the **depth of the tree is small** → getting to any element requires very few IO operations!

- Also can often store most or all of the B+ Tree in main memory!

The fanout is defined as the number of pointers to child nodes coming out of a node

Note that fanout is dynamic - we'll often assume it's constant just to come up with approximate eqns!

B+ Trees in Practice

Typical order: d=100. Typical fill-factor: 67%.

- average fanout = 133

Top levels of tree sit *in the buffer pool*:

- Level 1 = 1 page = 8 KB
- Level 2 = 133 pages = 1 MB
- Level 3 = 17,689 pages = 133 MB

Fill-factor is the percent of available slots in the B+ Tree that are filled; is usually < 1 to leave slack for (quicker) insertions

Typically, only pay for **one IO**!

Simple Cost Model for Search

Suppose:

- *f* = fanout, which is in [d+1, 2d+1] *(we'll assume it's constant for our cost model…)*
- *N* = the total number of *pages* we need to index
- *F* = fill-factor (usually ~= 2/3)

Our B+ Tree needs to have room to index *N / F* pages!

- We have the fill factor in order to leave some open slots for faster insertions

What height (*h*) does our B+ Tree need to be?

- h=1 → Just the root node- room to index f pages
- h=2 → f leaf nodes- room to index f^2 pages
- h=3 → f^2 leaf nodes- room to index f^3 pages
- …
- h → f^{h-1} leaf nodes- room to index f^h pages!

→ We need a B+ Tree of height h = $\left\lceil \log_f \frac{N}{F} \right\rceil$!

Simple Cost Model for Search

Note that if we have ***B*** available buffer pages, by the same logic:

- We can store $\boldsymbol{L_B}$ levels of the B+ Tree in memory
- where $\boldsymbol{L_B}$ *is the number of levels such that the sum of all the levels' nodes fit in the buffer:*
  - $B \geq 1 + f + \cdots + f^{L_B - 1} = \sum_{l=0}^{L_B - 1} f^l$

In summary: to do exact search:

- We read in one page per level of the tree
- However, levels that we can fit in buffer are free!
- Finally we read in the actual record

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + 1$

where $B \geq \sum_{l=0}^{L_B - 1} f^l$

Simple Cost Model for Search

To do range search, we just follow the horizontal pointers

The IO cost is that of loading additional leaf nodes we need to access + the IO cost of loading each *page* of the results- we phrase this as “Cost(OUT)”

IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + Cost(OUT)$

where $B \geq \sum_{l=0}^{L_B - 1} f^l$

3. Log-Structured Merge Tree

Problem with B+-Trees

Optimized for reads, not for writes (insert, update, delete)

Write amplification:

- ratio of the amount of data written to the storage device versus the amount of data written to the database
- Inserting and deleting a record could require updating multiple pages on disk (many random disk I/Os)

Log-structured Merge Tree (LSM Tree)

Proposed by O'Neil,Cheng, and Gawlick in 1996

Uses write-optimized techniques to significantly speed up inserts

Used by many NoSQL systems:

- e.g., Bigtable, Cassandra, Dynamo, HBase, RocksDB, InfluxDB

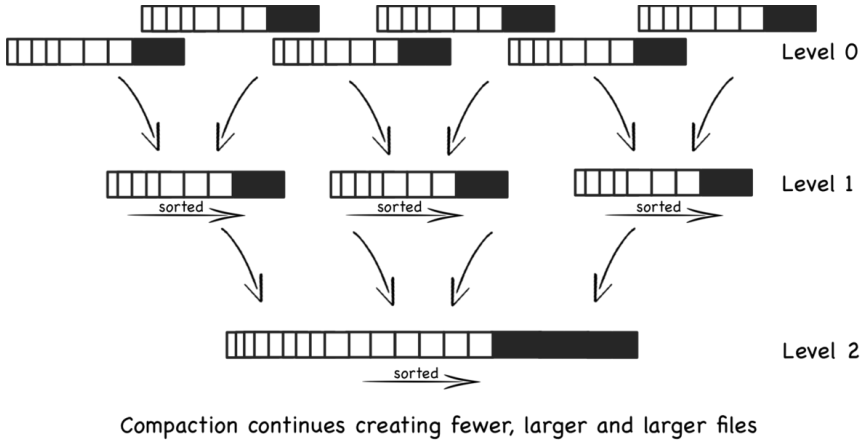


Image source: https://en.wikipedia.org/wiki/Log-structured_merge-tree

LSM Tree: Overview

Log-structured · Merge · Tree

Log-structured

All data is written sequentially, regardless of logical ordering

Merge

As data evolves, sequentially written runs of key-value pairs are merged

- Runs of data are indexed for efficient lookup
- Merges happen only after much new data is accumulated

Tree

The hierarchy of key-value pair runs form a tree

- Searches start at the root, progress downwards

LSM Tree: Overview [O'Neil, Cheng, Gawlick '96]

An LSM-tree comprises a hierarchy of levels of increasing size

- All data inserted into in-memory tree (C_0); no I/O cost at this step
- Larger on disk levels ($C_{i>0}$) hold data that does not fit into memory

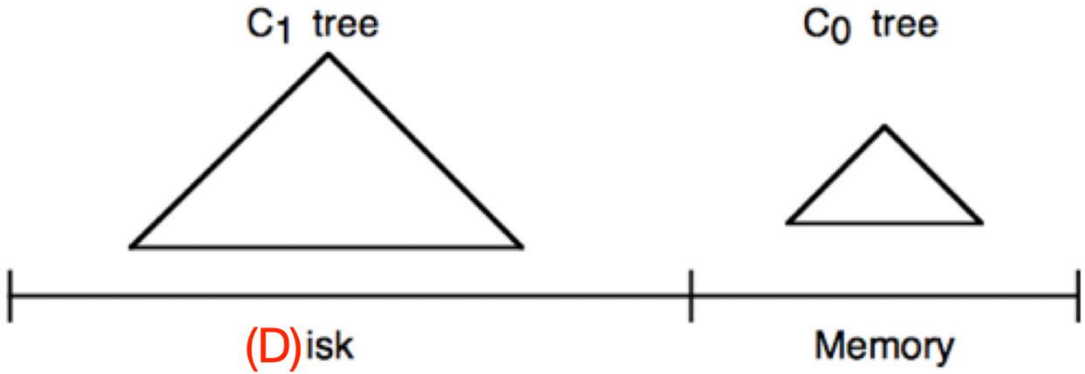


Figure 2.1. Schematic picture of an LSM-tree of two components

LSM Tree: Overview [O'Neil, Cheng, Gawlick '96]

When a level exceeds its size limit, its data is merged and rewritten

- Also called "compaction"
- Higher level is always merged into next lower level (C_i merged with C_{i+1})
- Merging always proceeds top down

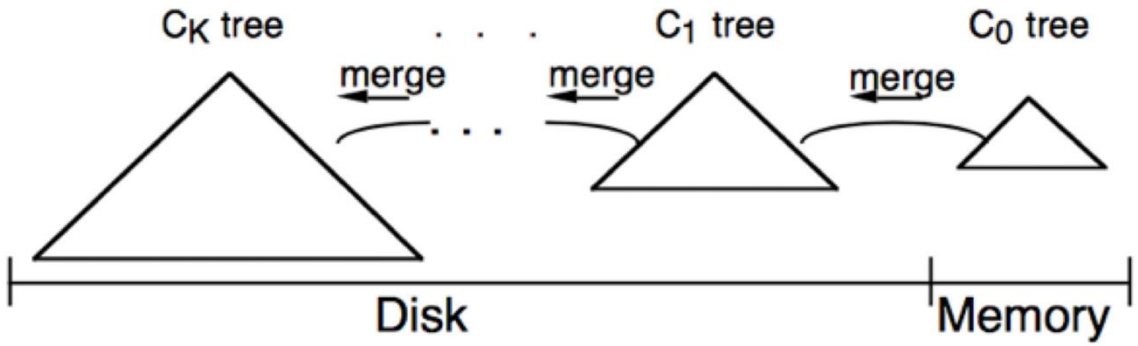


Figure 3.1. An LSM-tree of K+1 components

LSM Tree: Inserts

New data is written to an in-memory buffer (MemTable)

- typically organized as a balanced tree (like a Red-Black tree) to maintain sorted order
- Append-only log

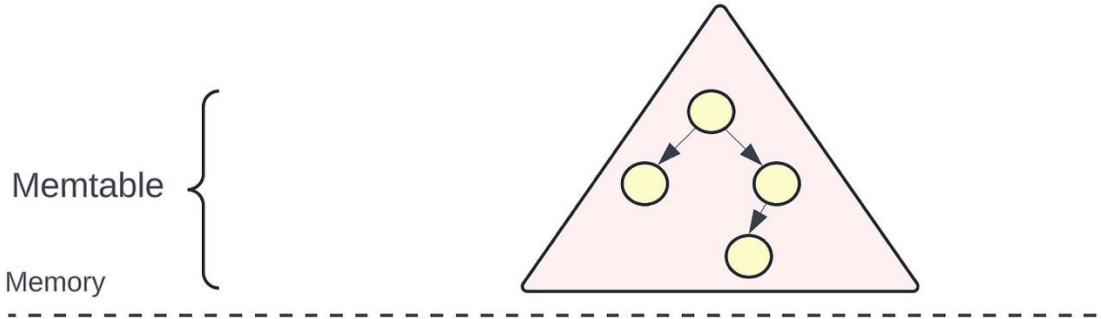


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

LSM Tree: Inserts

New data is written to an in-memory buffer (MemTable)

- typically organized as a balanced tree (like a Red-Black tree) to maintain sorted order

When the MemTable fills up, it's flushed to disk as a new SSTable file (the “run”) in Level 0

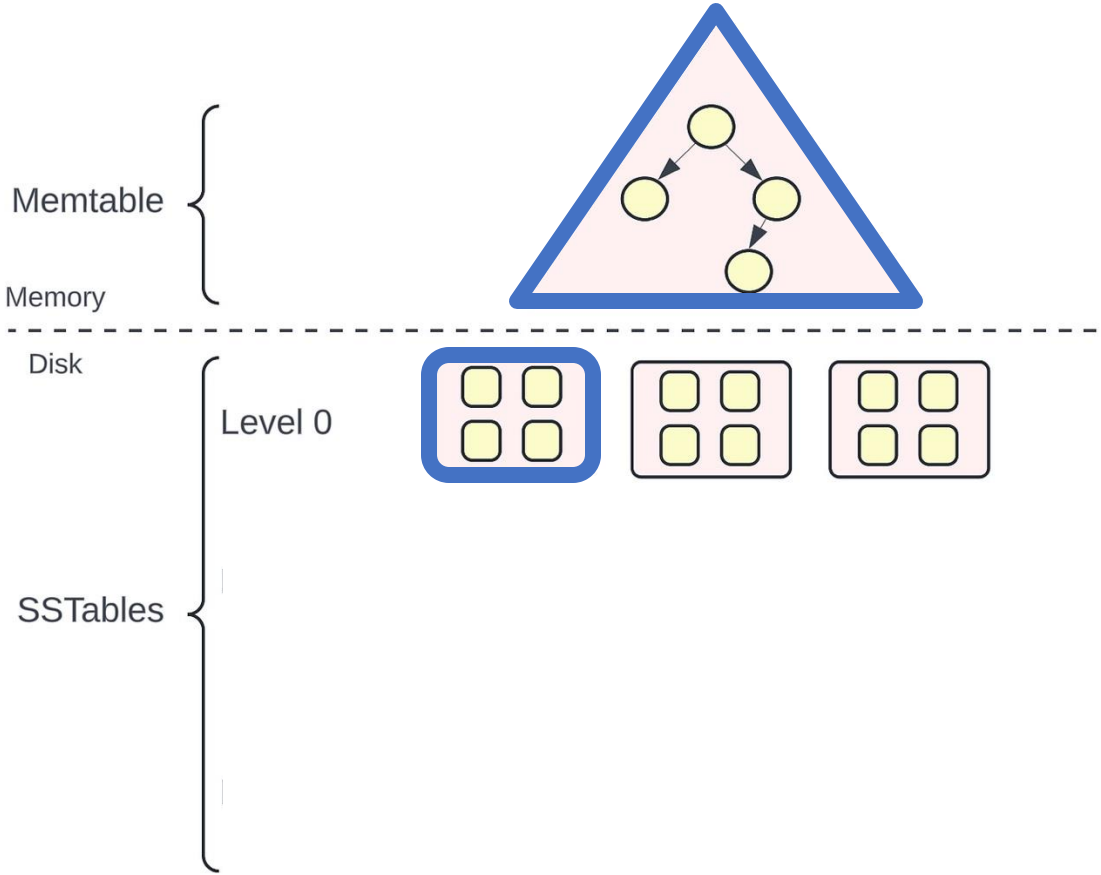


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

LSM Tree: Inserts

Once too many L0 files exist, compaction merges them into L1

- k-way merge sort
- Input: L0 files + L1 files with overlapping key ranges
- Output: new L1 files with non-overlapping ranges

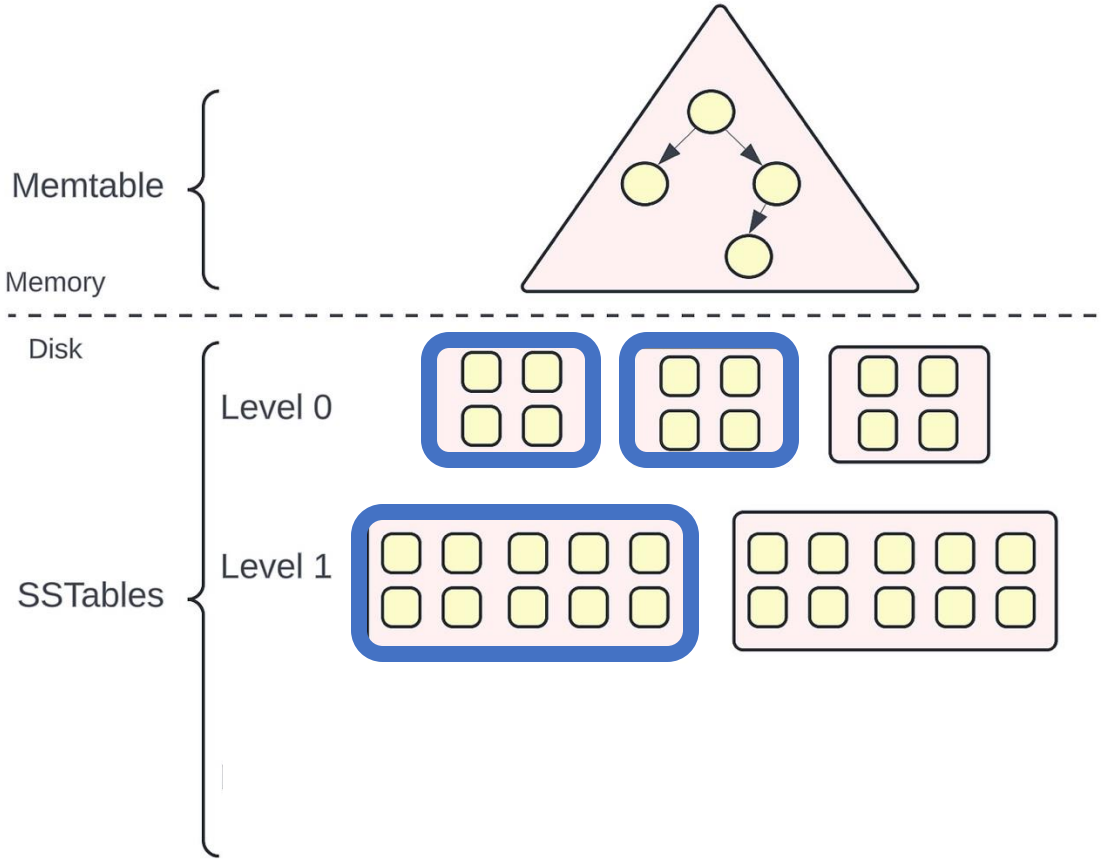


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

LSM Tree: Inserts

Once too many L0 files exist, compaction merges them into L1

Similarly, when there are too many L1 files, compaction merges them into L2

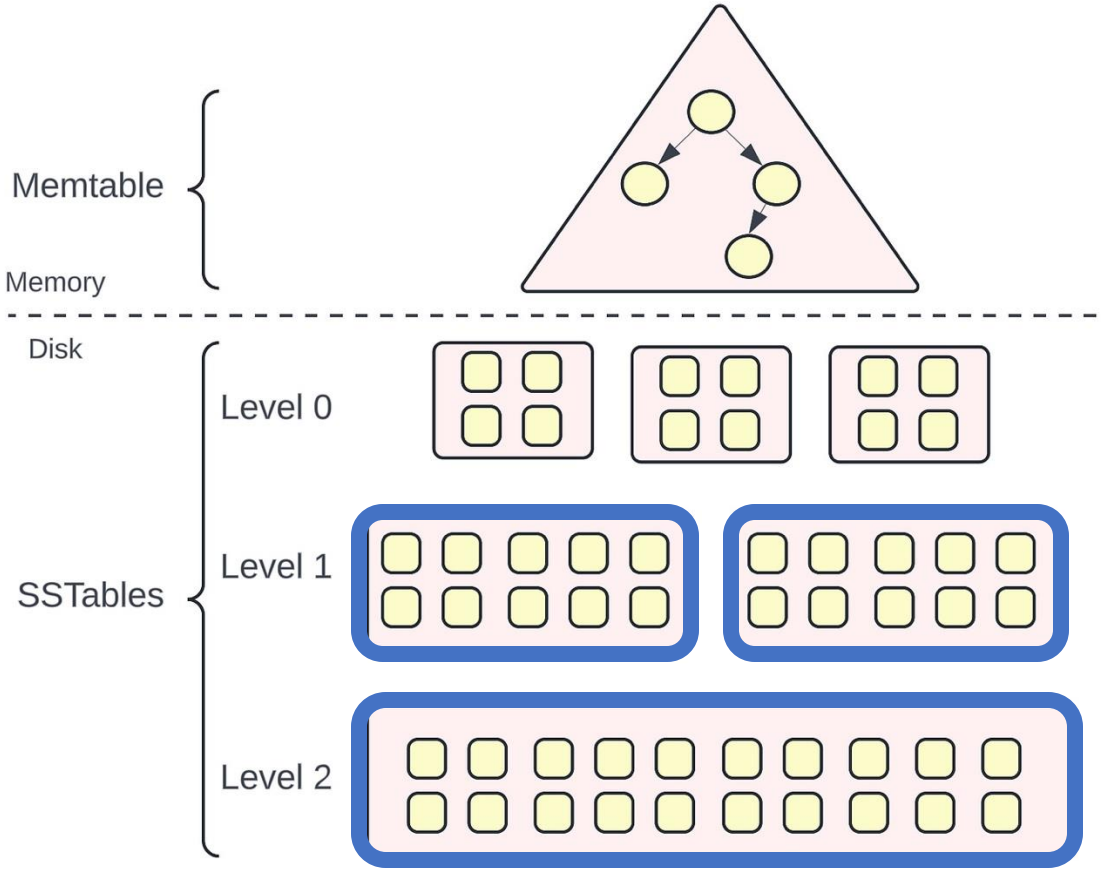


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

LSM Tree: Inserts

Once too many L0 files exist, compaction merges them into L1

Similarly, when there are too many L1 files, compaction merges them into L2

Different compaction strategies:

- Size tiered compaction strategy (bigger files in deeper levels)
- Leveled compaction strategy (more files per level)

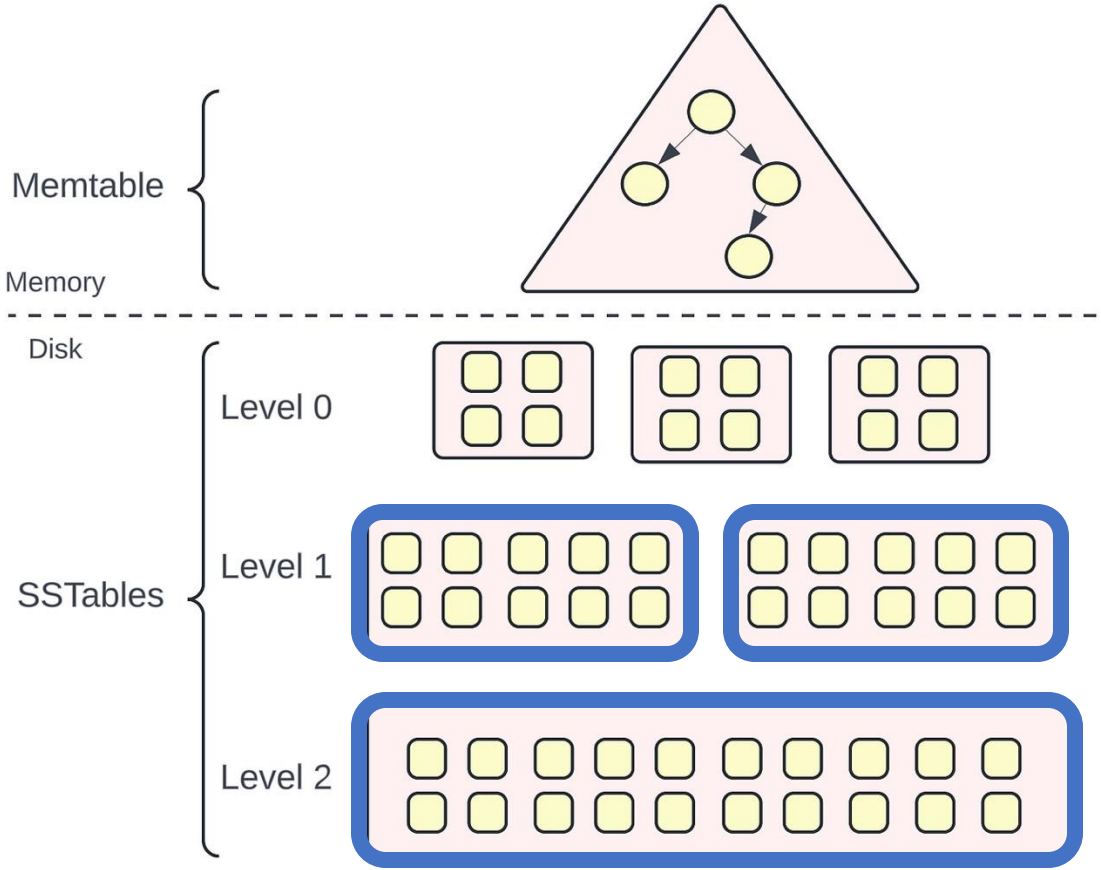


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

LSM Tree: Lookups

Check Memtables

Check SSTables:

- Bloom Filter:
  - check existence of key
- Summary Tables:
  - e.g., store the range of keys in each SSTable
  - Skip irrelevant files

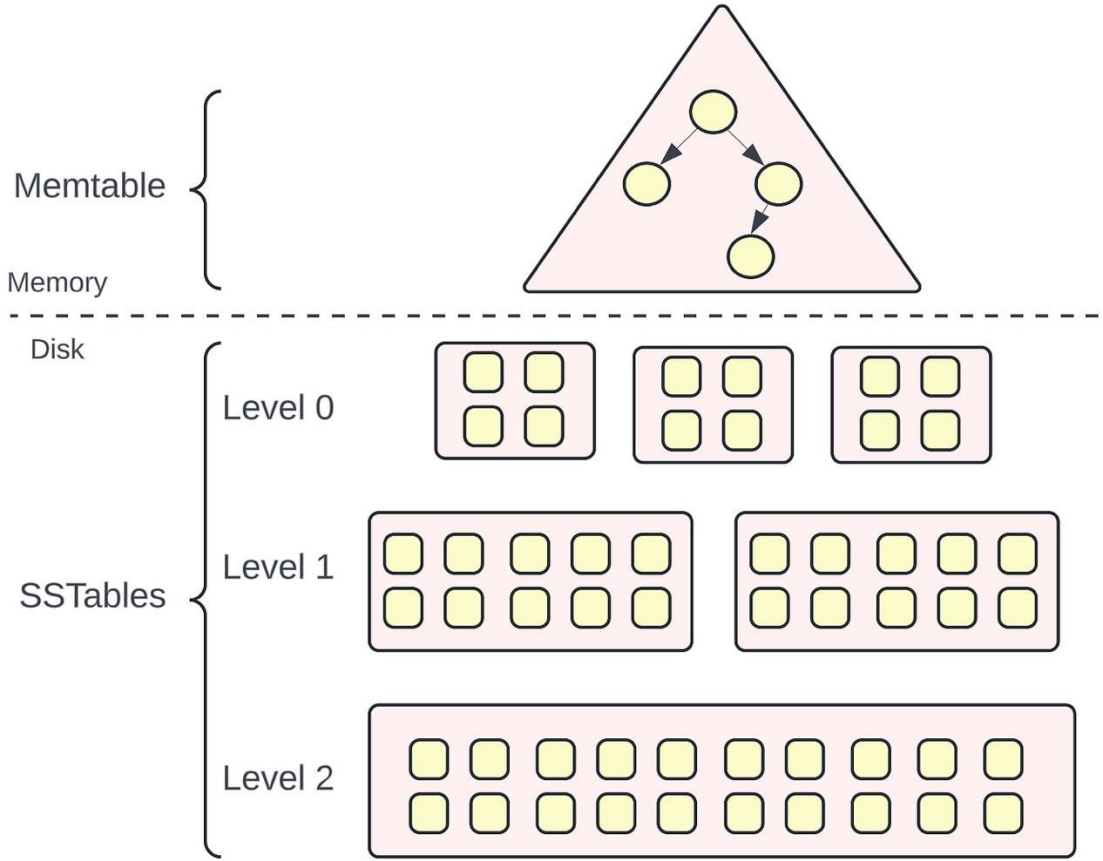


Image source: https://vivekbansal.substack.com/p/what-is-lsm-tree

Summary: B+ Tree vs LSM Tree

	B+ Tree	LSM Tree
Structure	Balanced tree with internal nodes and leaves	Multiple levels, memtable, and SSTables

Summary: B+ Tree vs LSM Tree

	B+ Tree	LSM Tree
Structure	Balanced tree with internal nodes and leaves	Multiple levels, memtable, and SSTables
Reads	Fast lookups and range queries	Potential for slower reads (multiple SSTables may need to be checked)

Summary: B+ Tree vs LSM Tree

	B+ Tree	LSM Tree
Structure	Balanced tree with internal nodes and leaves	Multiple levels, memtable, and SSTables
Reads	Fast lookups and range queries	Potential for slower reads (multiple SSTables may need to be checked)
Writes	Slower, as it can involve rebalancing, splitting and merging nodes	Faster, append-only for the initial writes

Summary: B+ Tree vs LSM Tree

	B+ Tree	LSM Tree
Structure	Balanced tree with internal nodes and leaves	Multiple levels, memtable, and SSTables
Reads	Fast lookups and range queries	Potential for slower reads (multiple SSTables may need to be checked)
Writes	Slower, as it can involve rebalancing, splitting and merging nodes	Faster, append-only for the initial writes
Typical Use Cases	Relational databases, read-heavy workloads	NoSQL databases, log-structured systems, write-heavy workloads