CS 564: Database Management Systems

Lecture 17: B+ Tree

Xiangyao Yu

3/1/2024

Module B2 Indexes

Hash index

B+ tree index

Advanced Indexing

External sort

Outline

B+ tree data structures

B+ tree operations

- Search
- Insertion
- Deletion

Primary vs. secondary indexes

Motivation

We have the following SQL query:

```
SELECT *
FROM Sales
WHERE price > 100 ;
```

Hash index accelerates equality search

- In expectation constant I/O cost for search and insert

Tree-based index accelerates both equality and range search

- In expectation Log(N) I/O cost for search and insert

B+ Tree Index

A dynamic tree-structured index

- Adjusted to be always height-balanced
- 1 node = 1 physical page

Supports efficient equality and range search

Widely used in many DBMSs

Outline

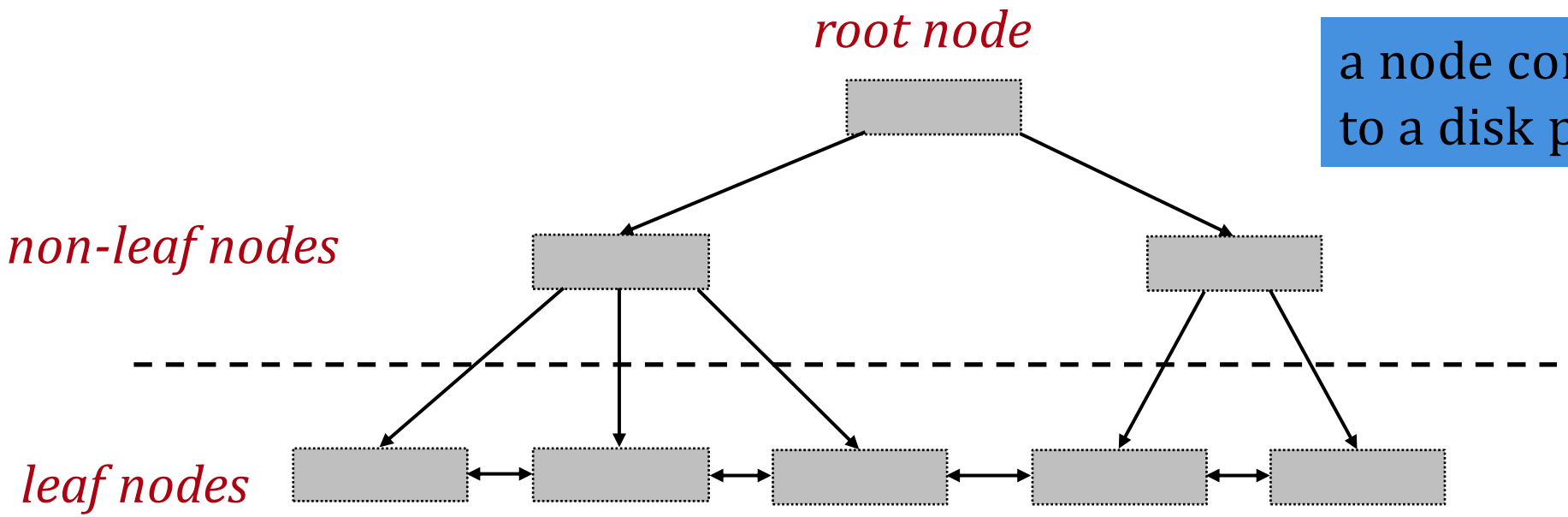
B+ tree data structures

B+ tree operations

- Search
- Insertion
- Deletion

Primary vs. secondary indexes

B+ Tree Index – Basic Structure



a node corresponds to a disk page

Index entries:

- **Exist *only* in the leaf nodes**
- **Are sorted according to the search key**

B+ Tree Index – Node

The parameter ***d*** is the *order* of the tree

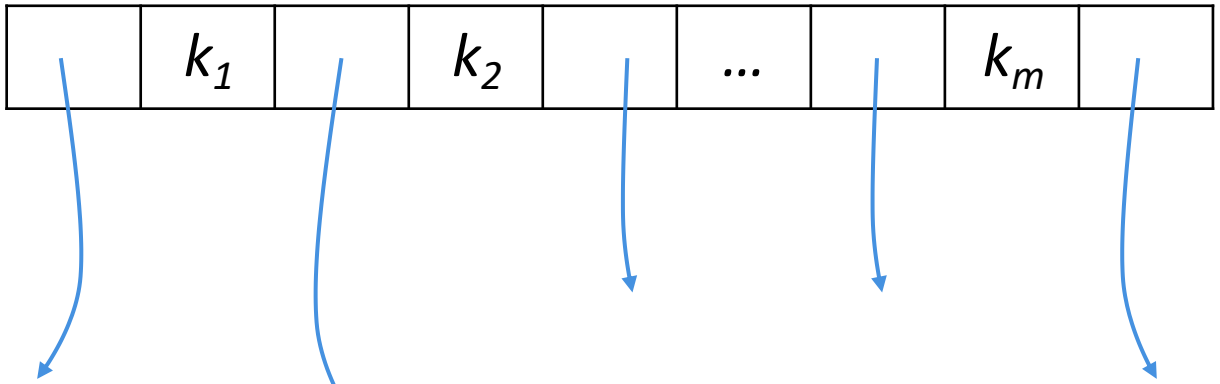
Each node contains $d \leq m \leq 2d$ entries

– Minimum 50% occupancy

With the exception of the root node, which can have $1 \leq m \leq 2d$ entries

Non-Leaf Nodes

A non-leaf (or internal) node with *m* entries has *m+1* pointers to lower-level nodes



pointer to a page with values < k_1

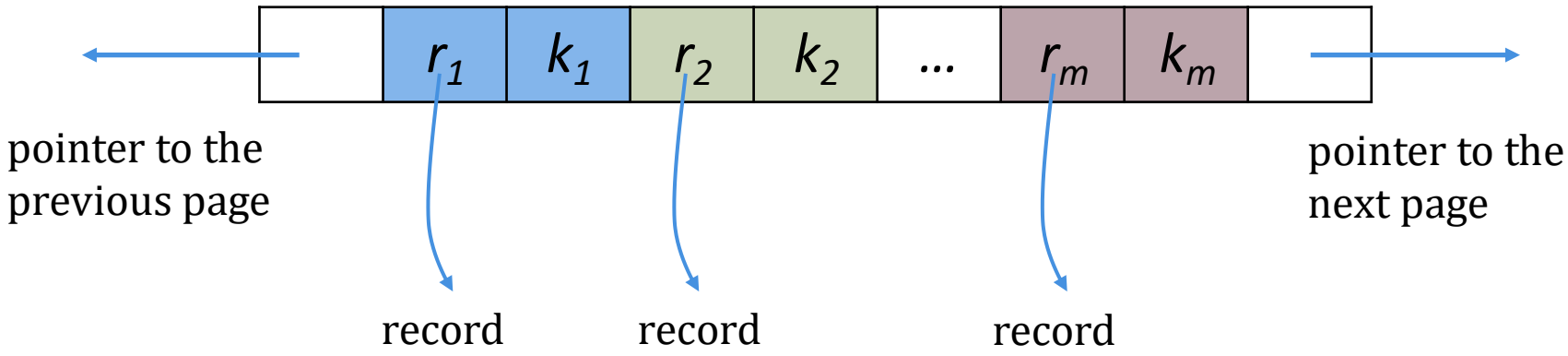
pointer to a page with $k_1 \leq$ values < k_2

pointer to a page with values ≥ k_m

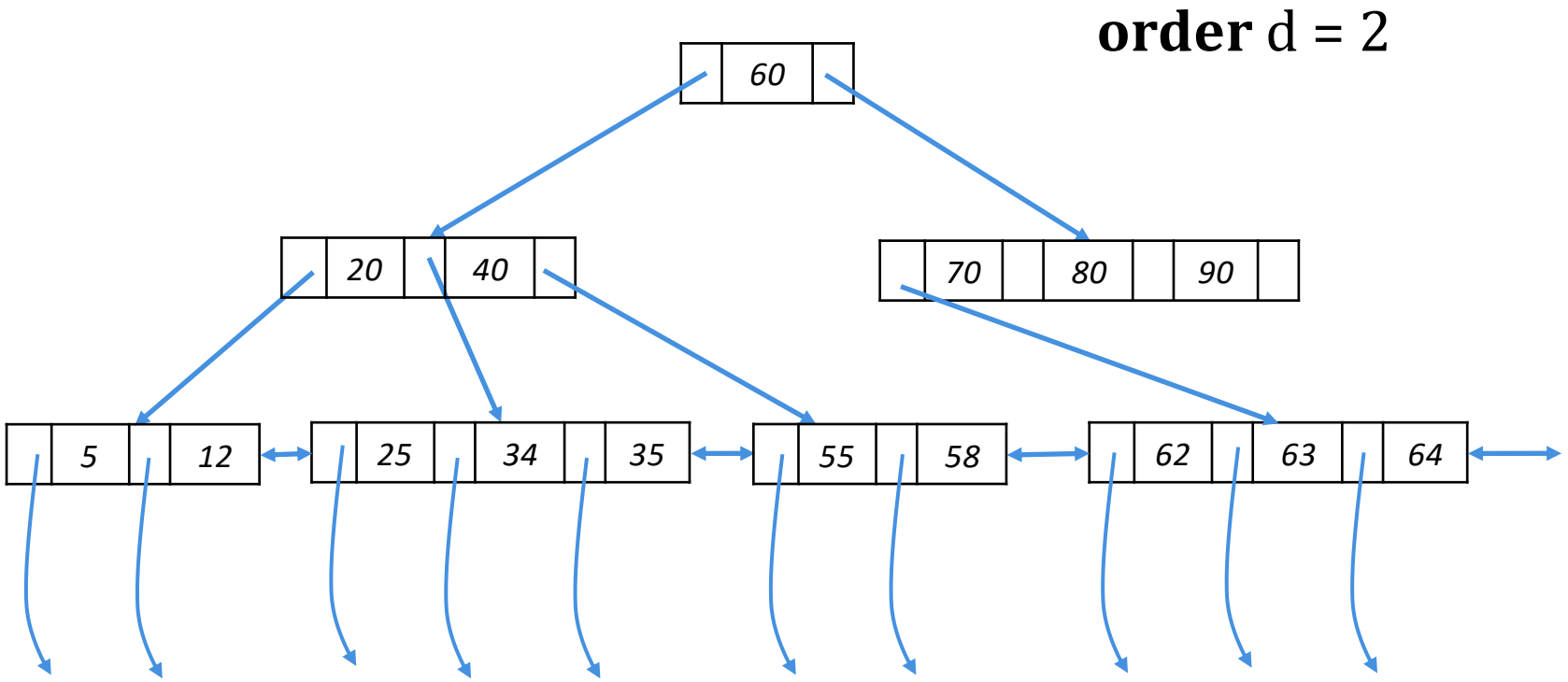
Leaf Nodes

A leaf node with *m* entries has

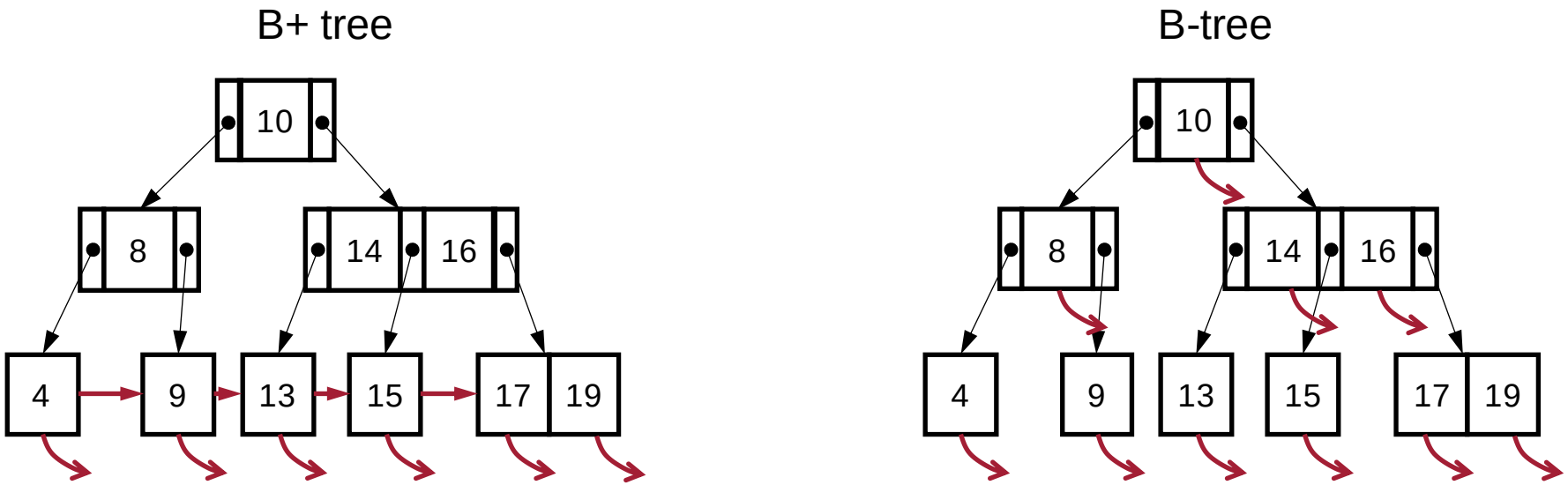
- *m* pointers to the data records (rids)
- pointers to the next and previous leaves



B+ Tree Example



B+ Tree vs. B-Tree



B-tree: data pointers stored in all nodes

B+ tree:

- Data pointers stored only in leaf nodes
- The leaf nodes are linked

Outline

B+ tree data structures

B+ tree operations

- Search
- Insertion
- Deletion

Primary vs. secondary indexes

B+ Tree Operations

A B+ tree supports the following operations:

- equality search
- range search
- insert
- delete
- bulk loading

Outline

B+ tree data structures

B+ tree operations

- **Search**
- Insertion
- Deletion

Primary vs. secondary indexes

Search

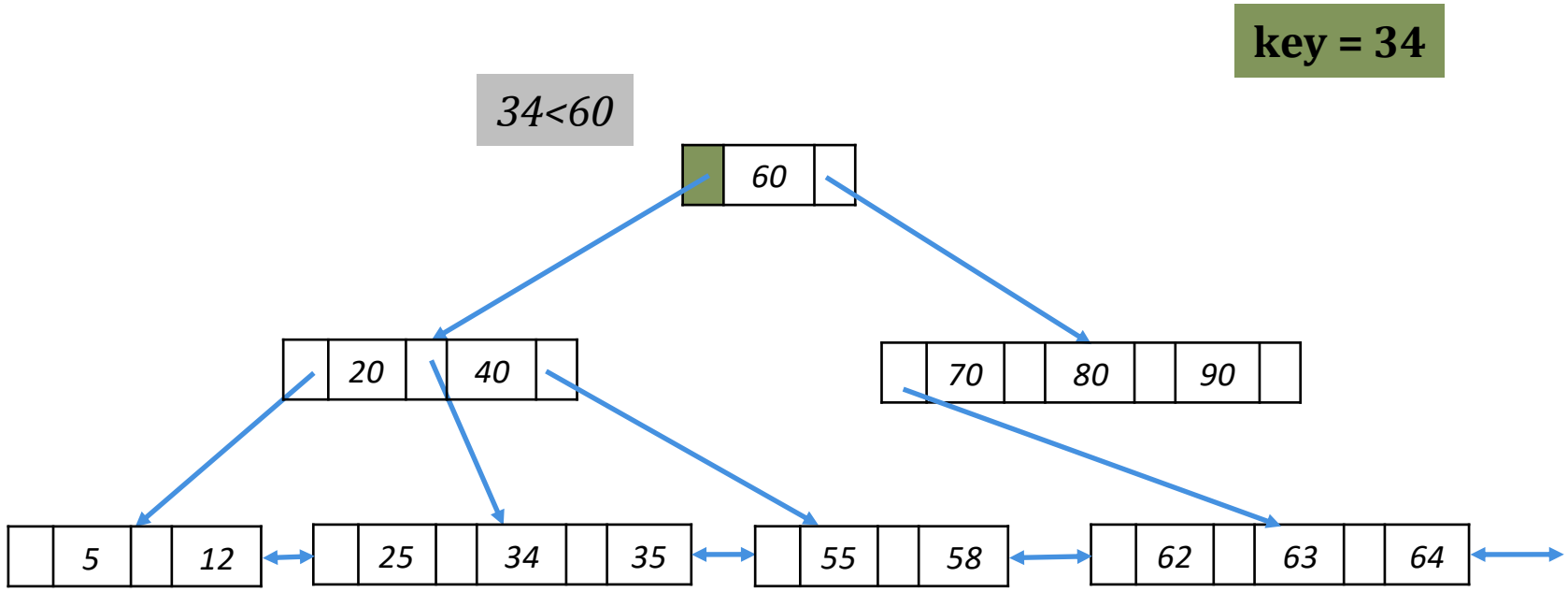
Start from the root node

Examine the index entries in non-leaf nodes to find the correct child

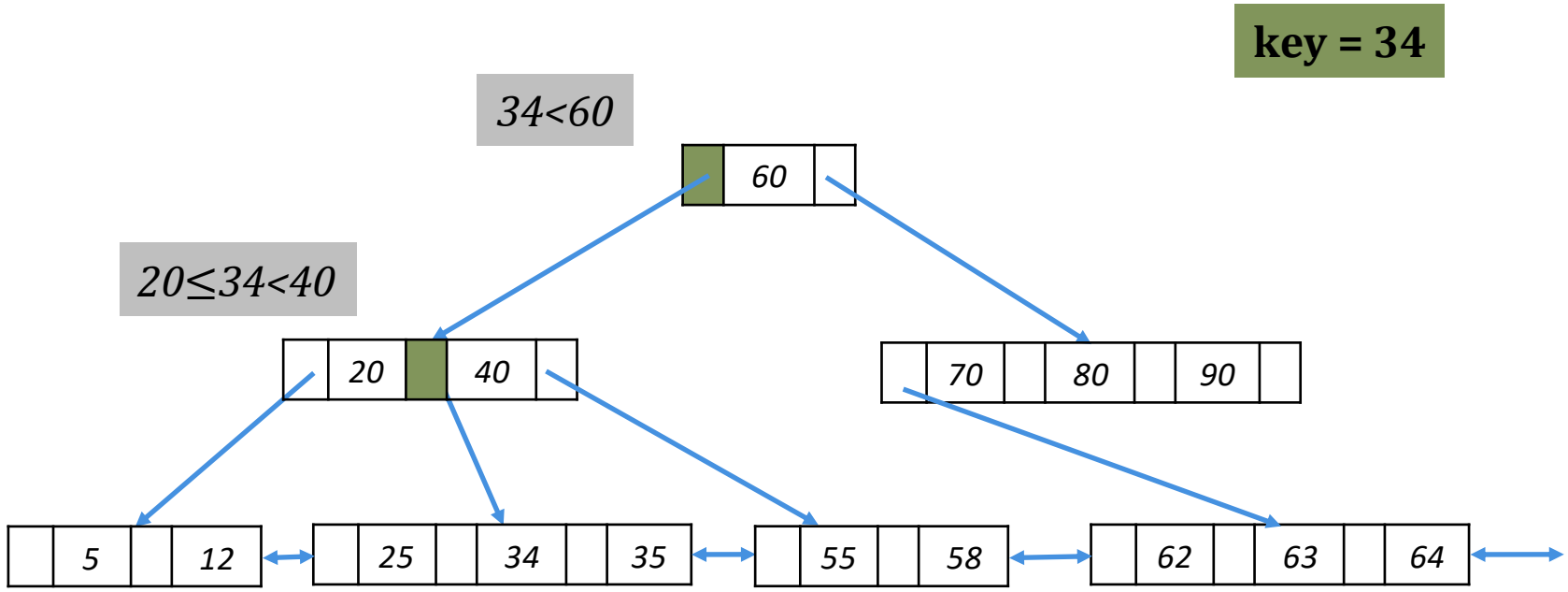
Traverse down the tree until a leaf node is reached

- For equality search, we are done
- For range search, traverse the leaves sequentially using the previous/next pointers

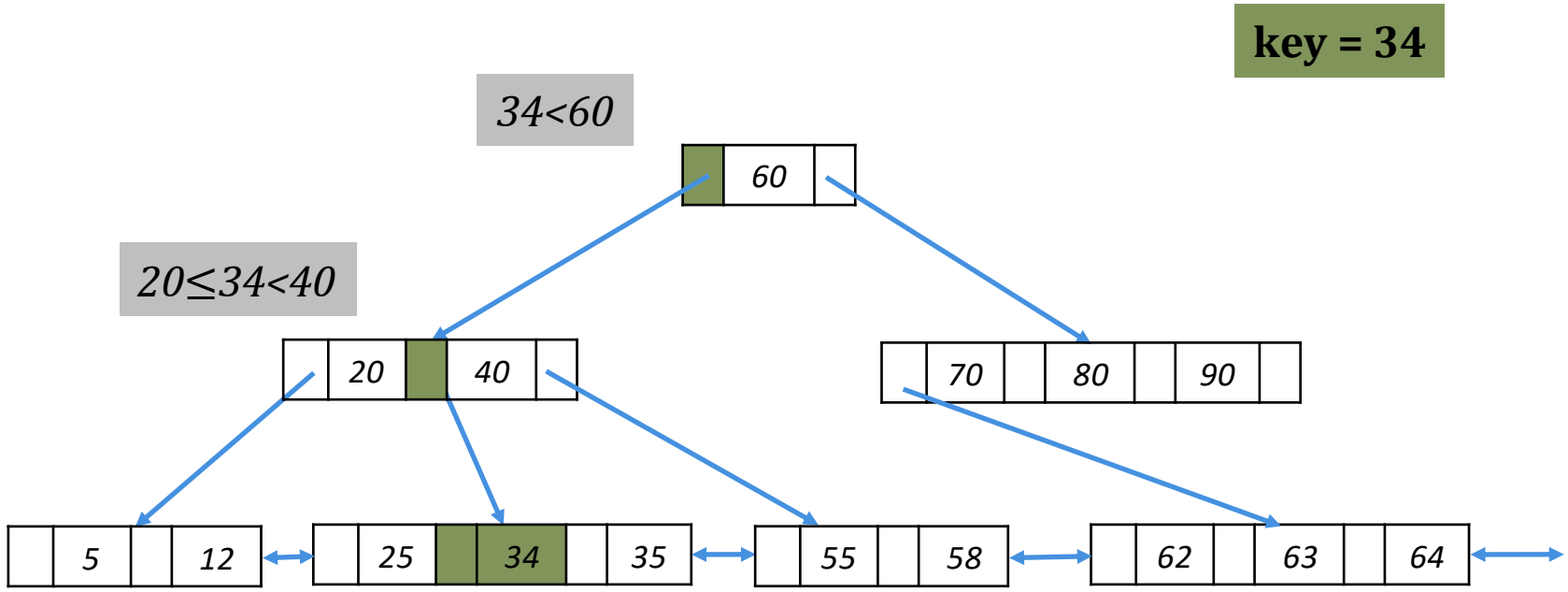
Equality Search – Example



Equality Search – Example

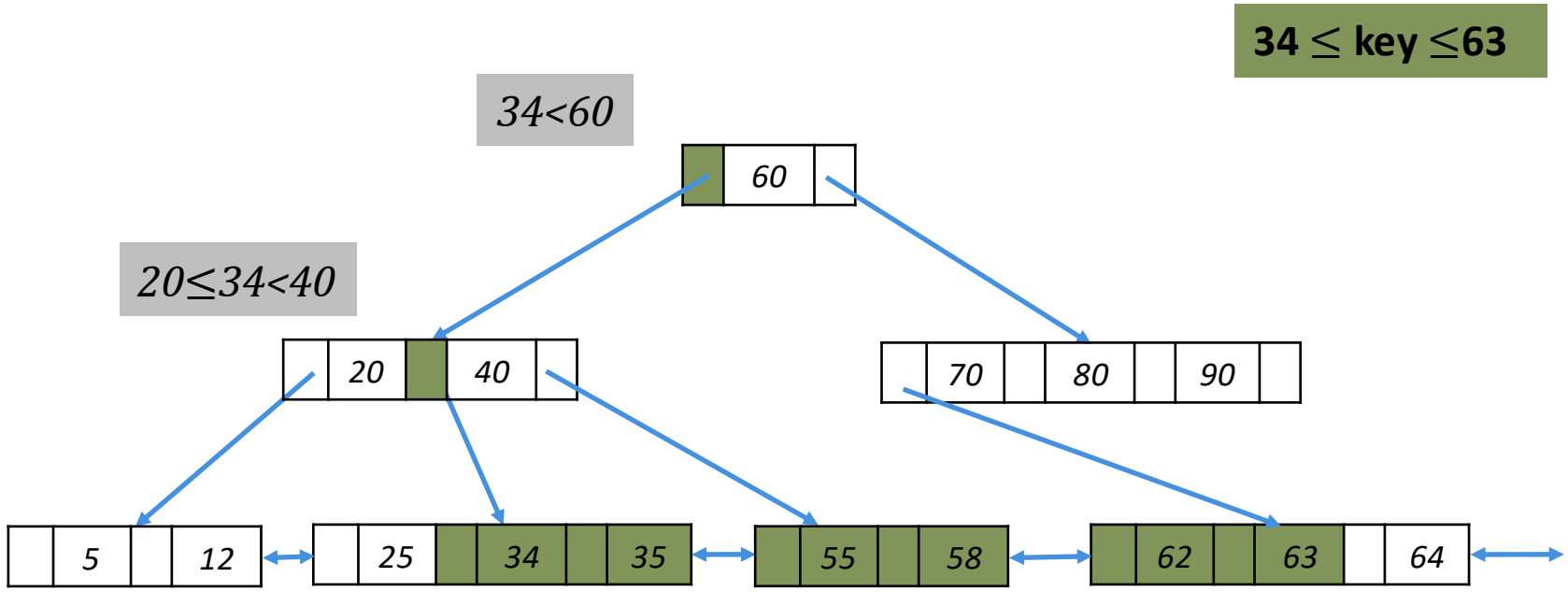


Equality Search – Example



To locate the correct data entry in the leaf node, we can do either linear or binary search

Range Search – Example



After we find the leftmost point of the range, we traverse sequentially!

Outline

B+ tree data structures

B+ tree operations

- Search
- **Insertion**
- Deletion

Primary vs. secondary indexes

Insert

Find the leaf node **L** where the entry belongs

Insert data entry in **L**

- If **L** has enough space, DONE!
- Otherwise, we must split **L** (into **L** and a new node **L'**)

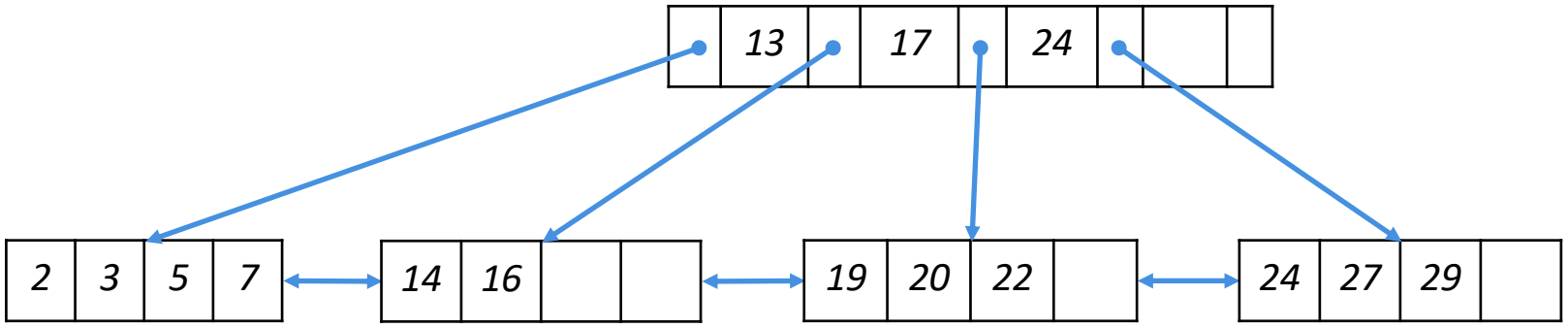
This can propagate recursively to other nodes!

- To split a non-leaf node, redistribute entries evenly, but **push up** the middle key

Insert Example

order **d** = 2

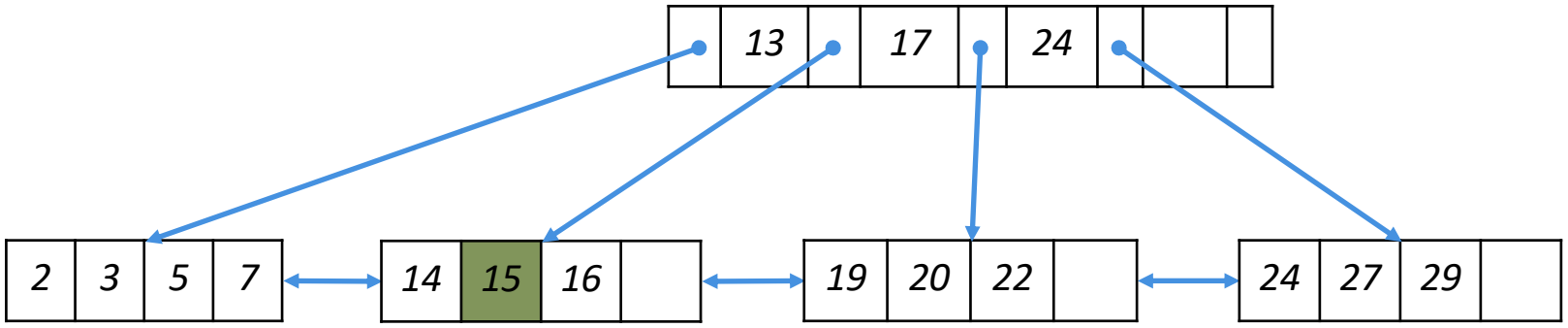
Insert 15



Insert Example

order **d** = 2

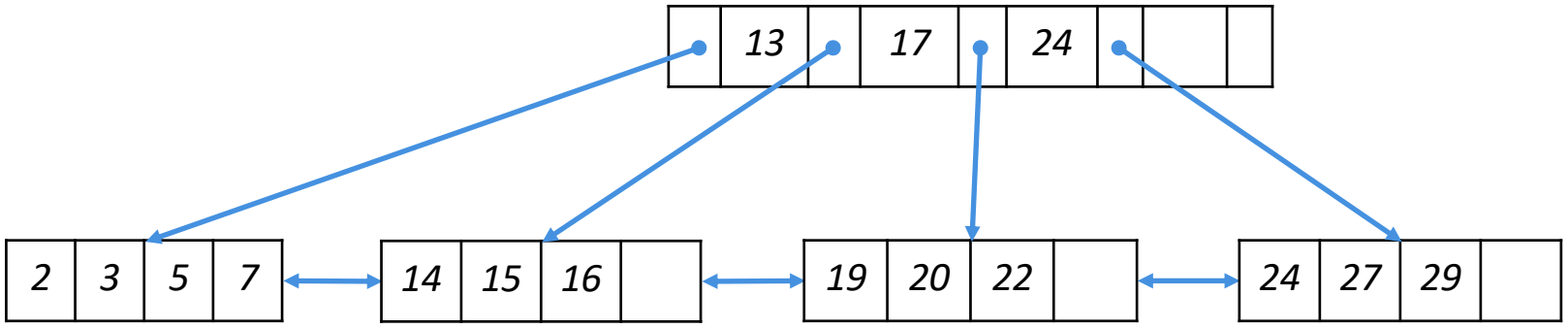
Insert 15



Insert Example

order **d** = 2

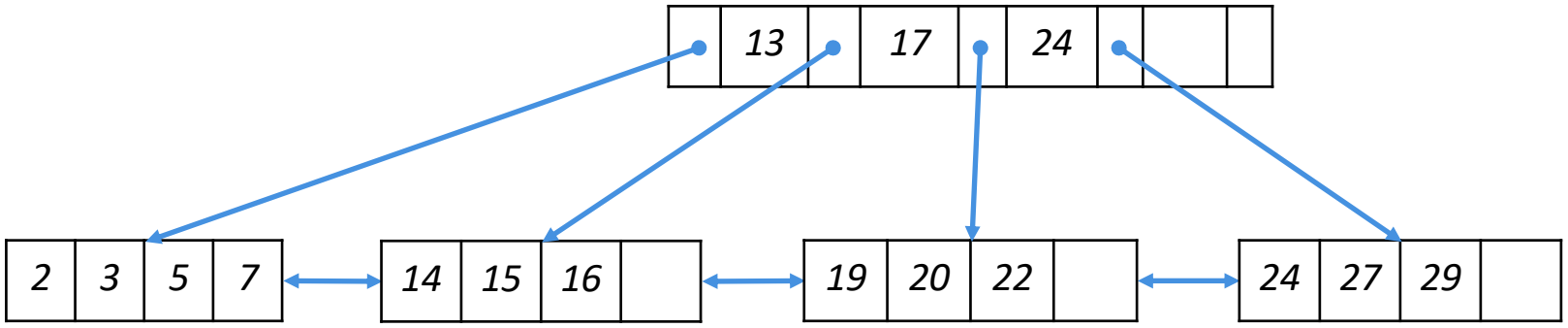
Insert 8



Insert Example

order **d** = 2

Insert 8



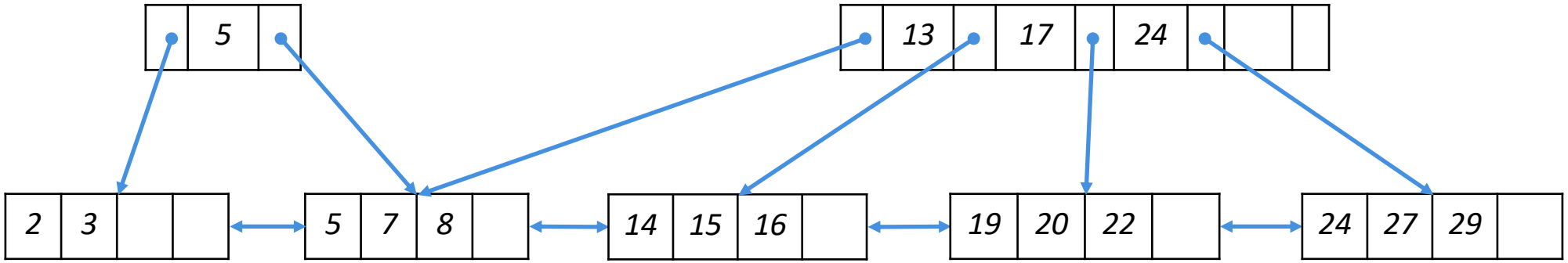
the leaf node is full so we must split it!

Insert Example

order **d** = 2

Insert 8

The middle key (***5***) must be inserted to the parent node

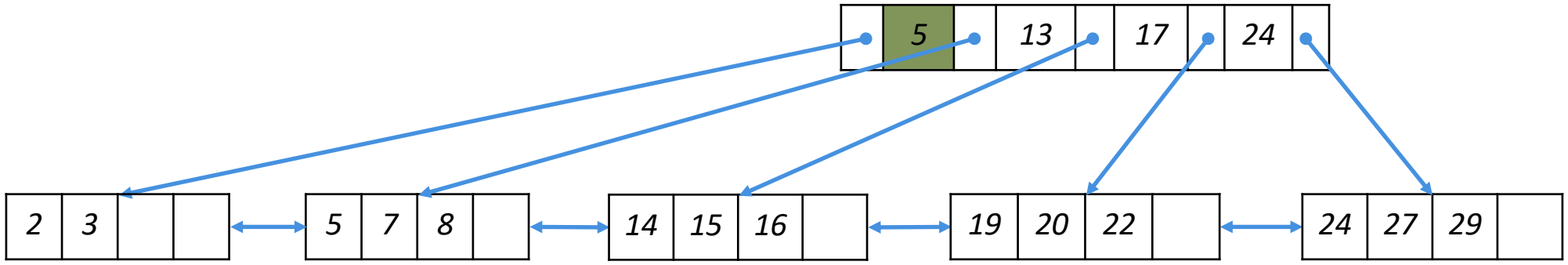


Insert Example

order **d** = 2

Insert 8

The middle key (***5***) must be inserted to the parent node

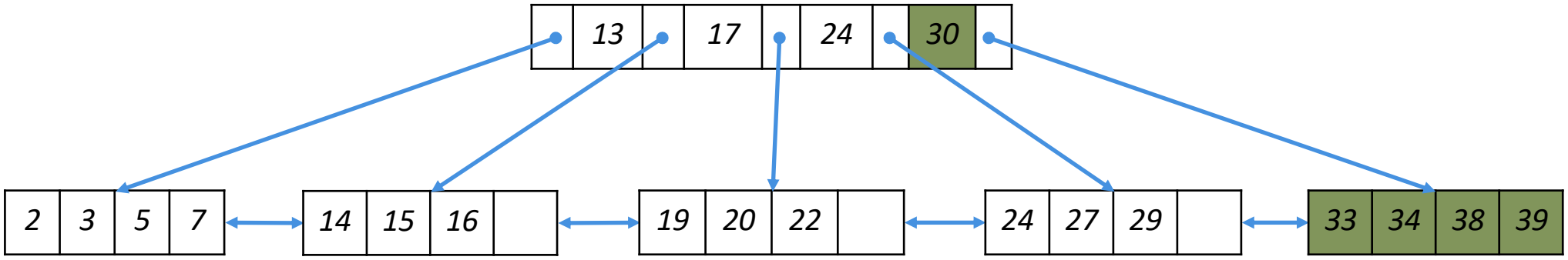


Insert Example – Parent Node Split

order **d** = 2

What if the parent node is already full before the insertion?

Insert 8



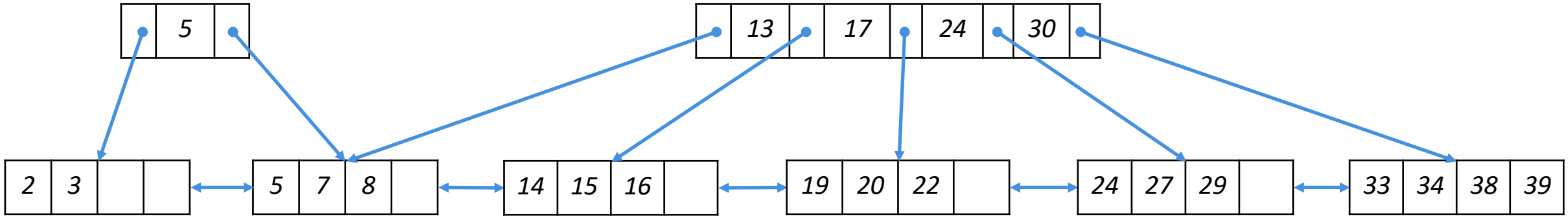
Insert Example – Parent Node Split

order **d** = 2

What if the parent node is already full before the insertion?

Insert 8

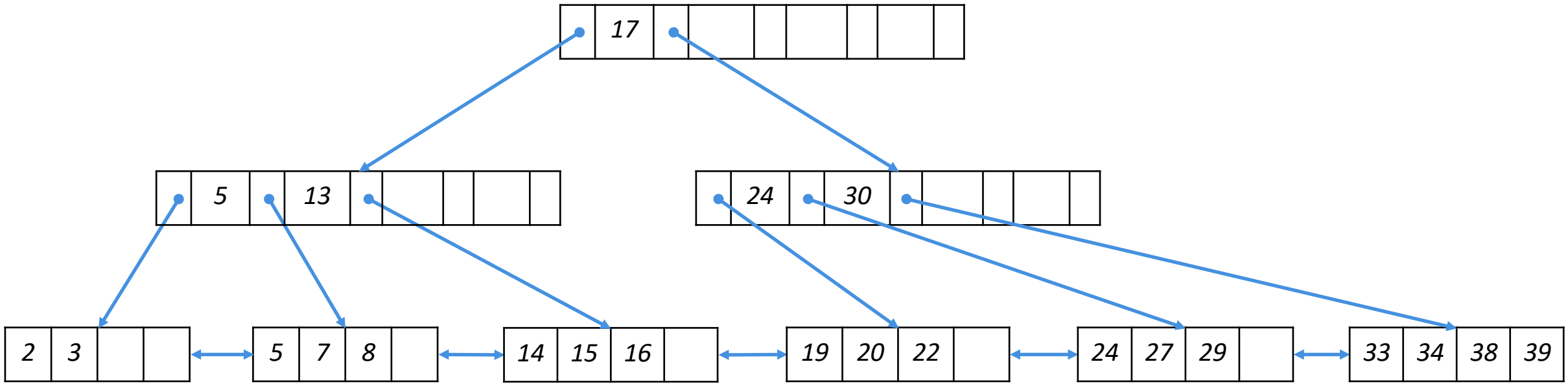
The parent node must also be split



Insert Example – Parent Node Split

order **d** = 2

Insert 8



Insert Properties

The B+ tree insertion algorithm has several attractive qualities:

- About the same cost as equality search
- ***Self-balancing:*** the tree remains balanced (with respect to height) even after multiple insertions

Outline

B+ tree data structures

B+ tree operations

- Search
- Insertion
- **Deletion**

Primary vs. secondary indexes

Delete

Find the leaf node **L** where the entry belongs

Remove the entry

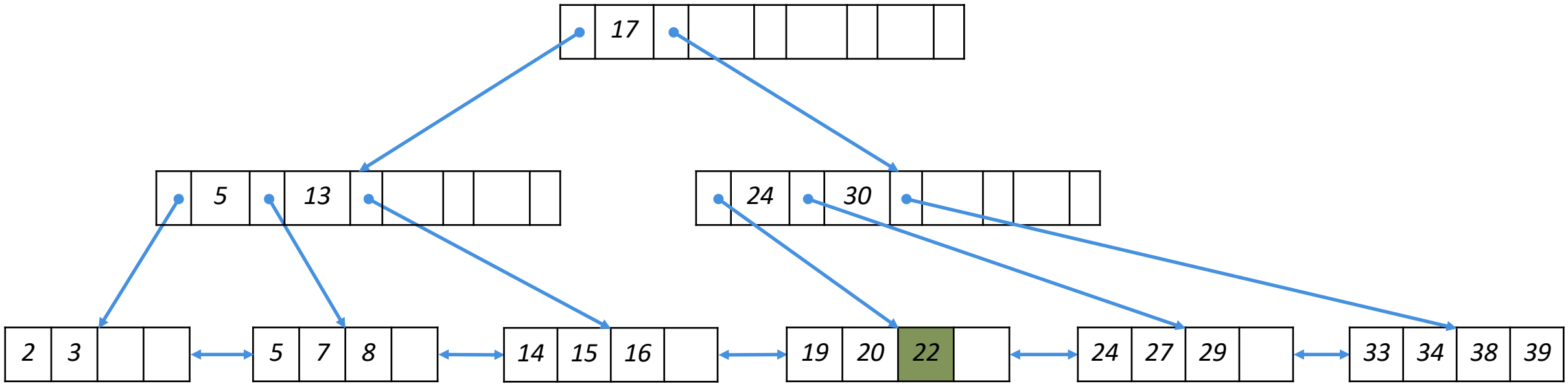
- If **L** is at least half-full, DONE!
- If **L** has only *d-1* entries,
  - Try to **redistribute** borrowing entries from a neighboring sibling
  - If redistribution fails, **merge** **L** and sibling

If a merge occurred, we must delete an entry from the parent of **L**

Delete – Example

order **d** = 2

Delete 22

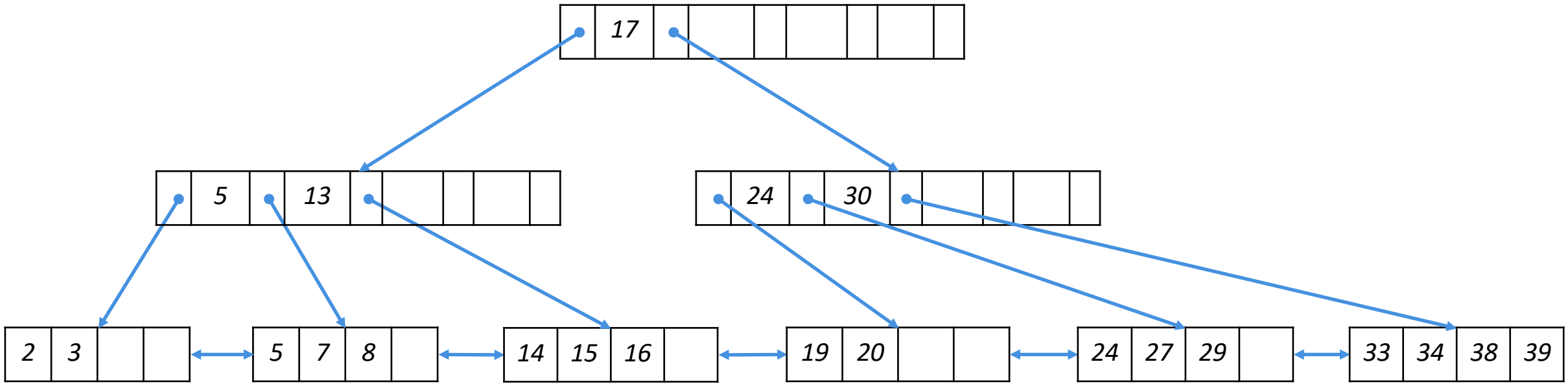


Since by deleting 22 the node remains half-full, we simply remove it

Delete – Example

order **d** = 2

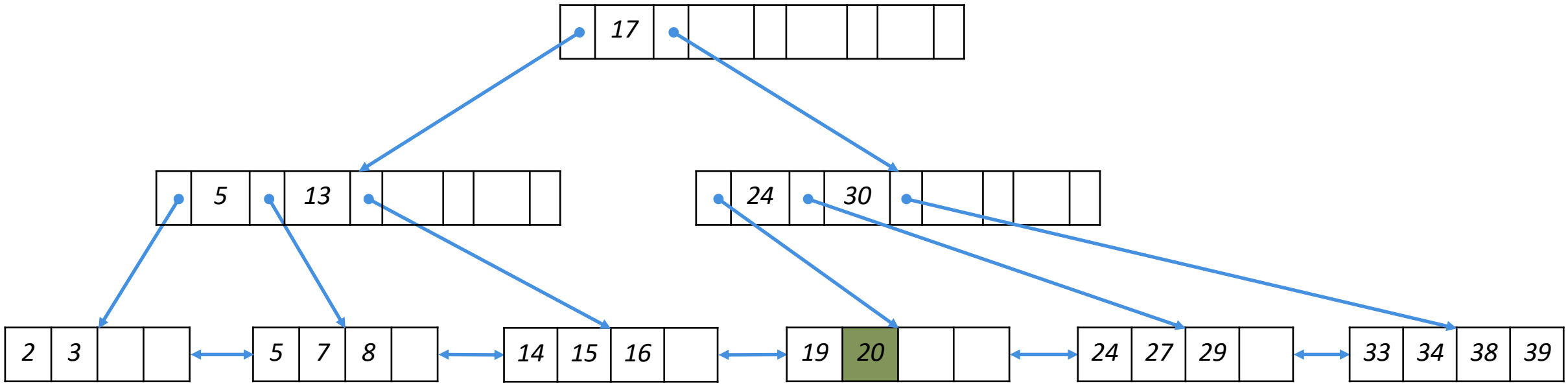
Delete 22



Delete – Example

order **d** = 2

Delete 20

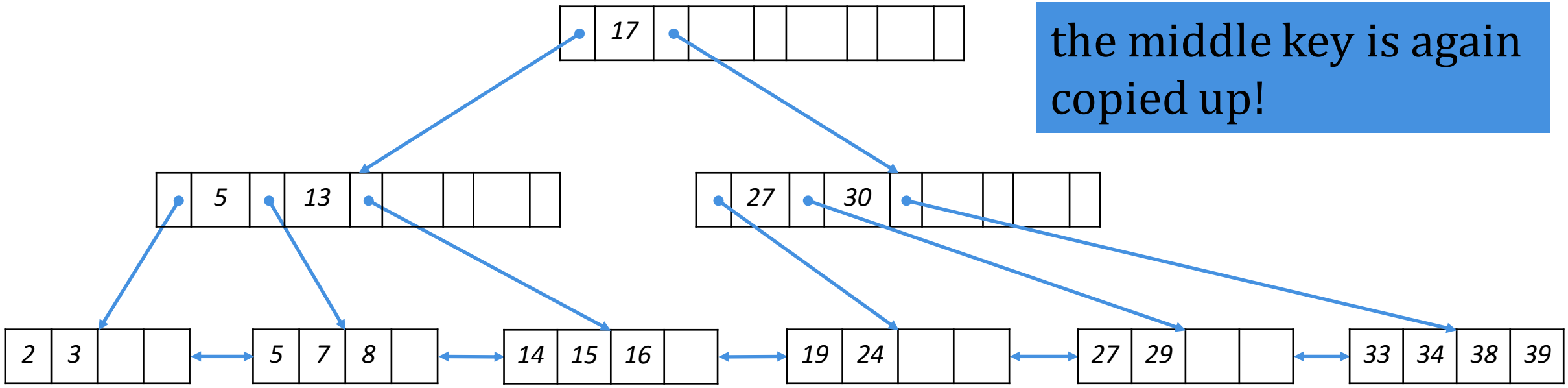


by removing 20 the node is not half-full anymore, so we attempt to redistribute!

Delete – Example

order **d** = 2

Delete 20



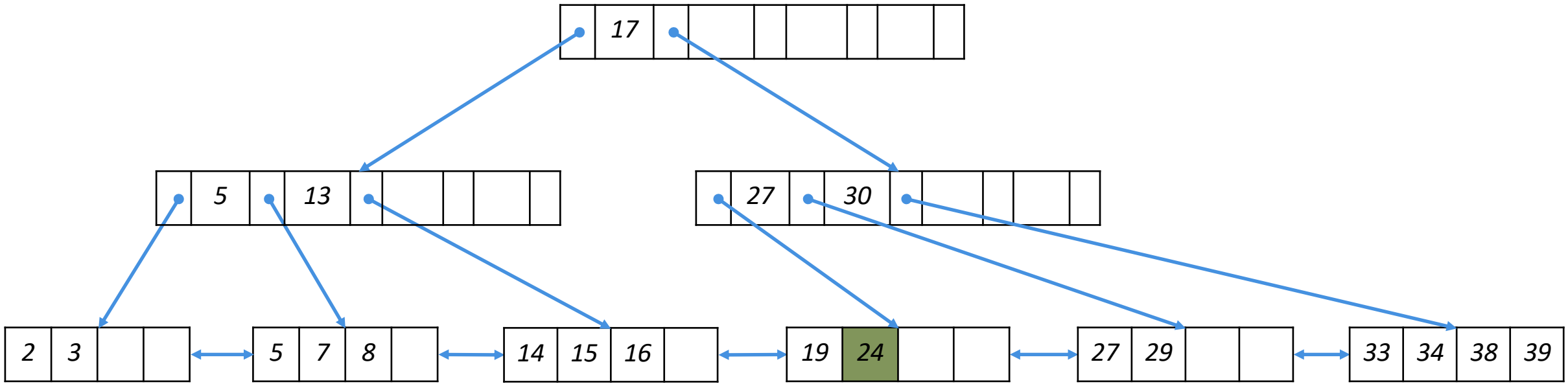
the middle key is again copied up!

by removing 20 the node is not half-full anymore, so we attempt to redistribute!

Delete – Example

order **d** = 2

Delete 24

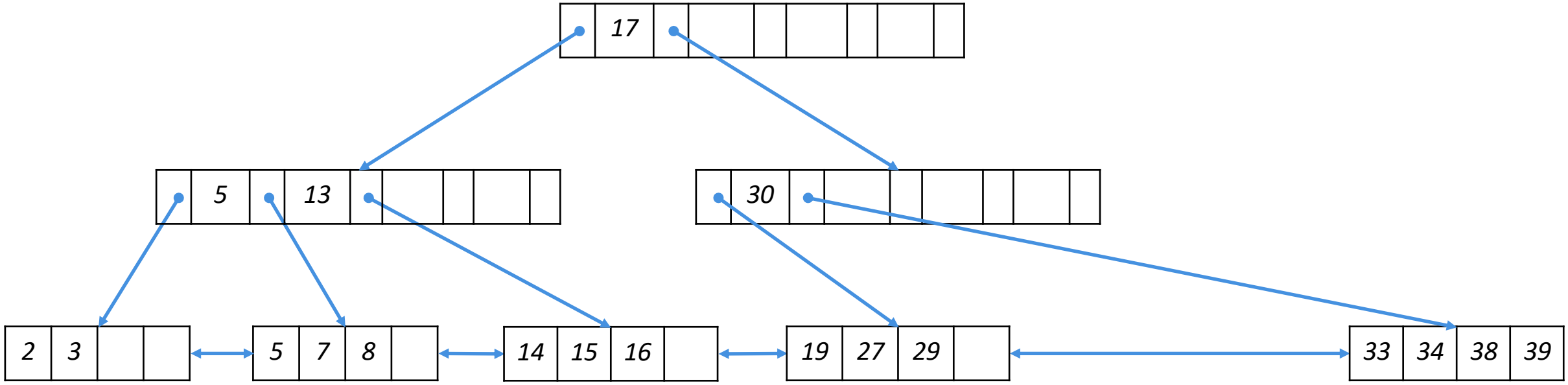


in this case, we have to merge nodes!

Delete – Example

order **d** = 2

Delete 24

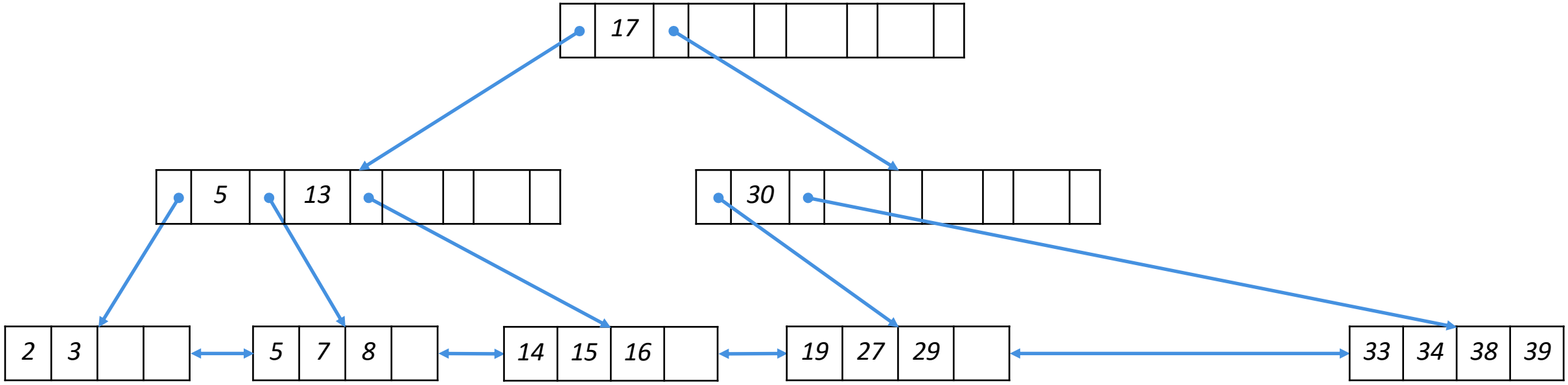


in this case, we have to merge nodes!

Delete – Example

order **d** = 2

Delete 24



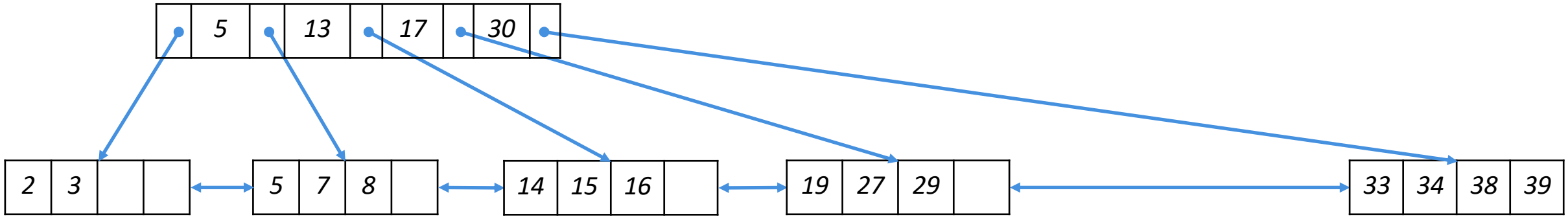
We are not done, since the resulting non-leaf node is not half-full!

Delete – Example

order **d** = 2

Delete 24

Merge non-leaf nodes



We are not done, since the resulting non-leaf node is not half-full!

More on Delete

Redistribution of entries is also possible for the non-leaf nodes

We can also try to redistribute using *all siblings*, and not only the neighboring one

In real systems, deletion is often implemented without adjusting the tree structure, because files typically grow rather than shrink

Outline

B+ tree data structures

B+ tree operations

- Search
- Insertion
- Deletion

Primary vs. secondary indexes

Primary vs. Secondary Index

If the search key contains the primary key, it is called a primary index

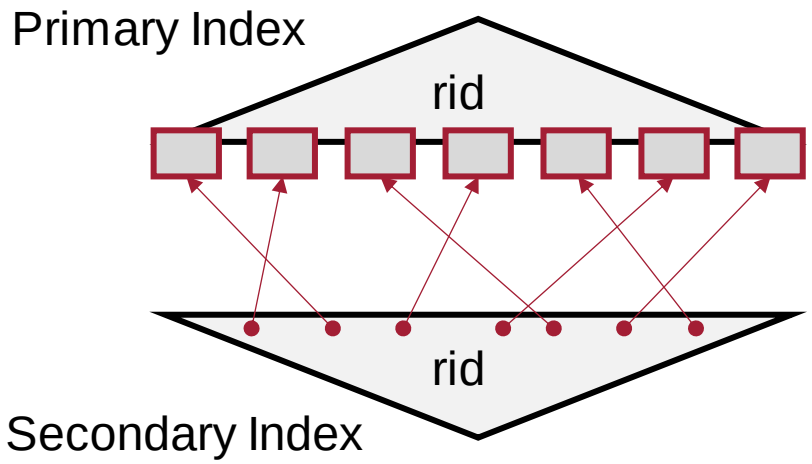
- In a primary index, there are no duplicates for a value of the search key
- There can only be one primary index!

Any other index is called a secondary index

If the search key contains a candidate key, it is called a unique index

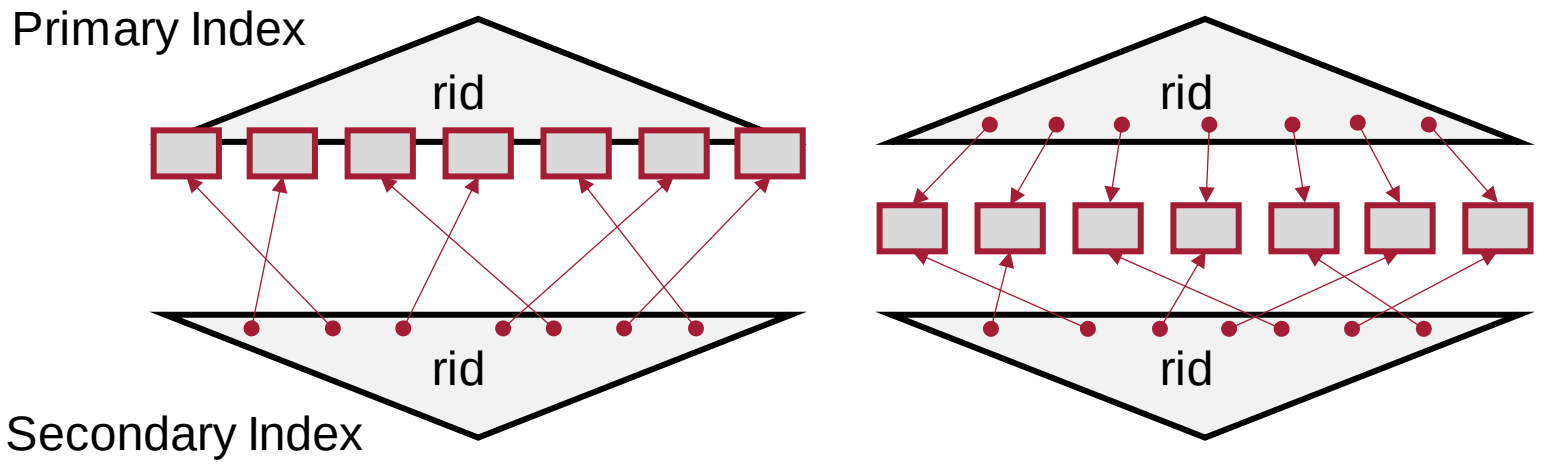
- A primary index is also a unique index

Alternative Secondary Index Design



Primary index stores records, secondary index stores rid

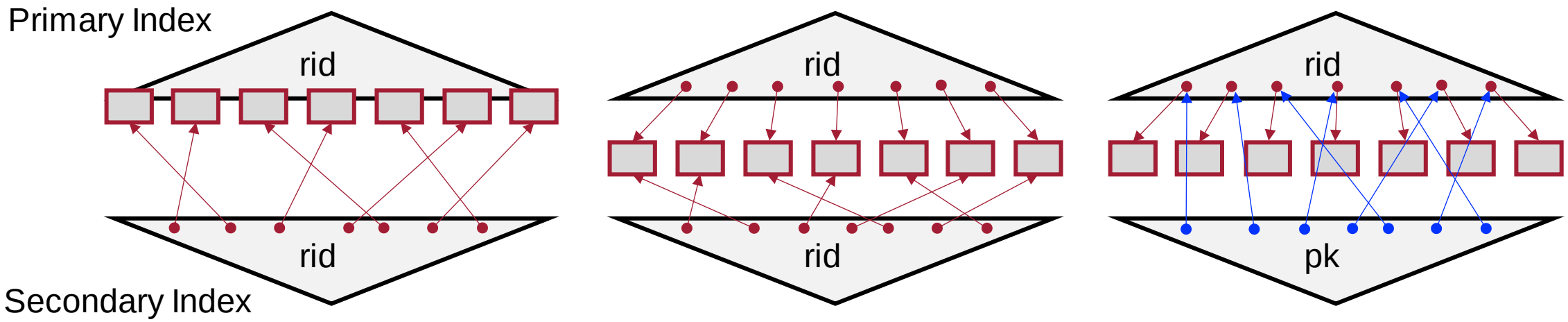
Alternative Secondary Index Design



Primary index stores records, secondary index stores rid

Both primary and secondary indexes store rid

Alternative Secondary Index Design



Primary index stores records, secondary index stores rid

Both primary and secondary indexes store rid

Secondary index store the primary key

- Only primary index is changed when updating rid

Summary

B+ tree data structures

B+ tree operations

- Search
- Insertion
- Deletion

Primary vs. secondary indexes