THE B+ TREE INDEX

CS 564- Fall 2021

ACKs: *Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

The **B+ tree** index

- Basic principles
- Search/Insertion/Deletion
- Design choices
- I/O Cost

INDEX RECAP

We have the following SQL query:

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

Indexes help us organize the file to answer the above query efficiently!

INDEXES

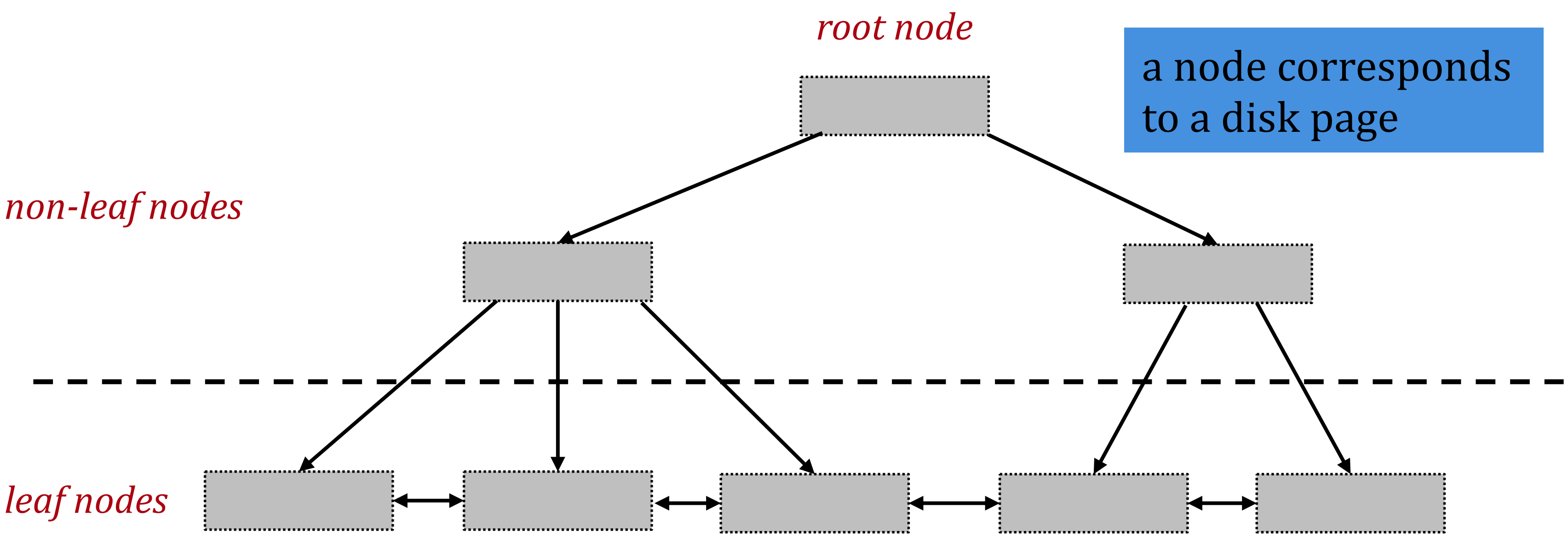
- Hash index:
  - good for equality search
  - in expectation constant I/O cost for search and insert
- B+ tree index:
  - good for range search
  - good for equality search

B+ Tree Basics

THE 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
  – SQLite uses it as the default index
  – SQL Server, DB2, …

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

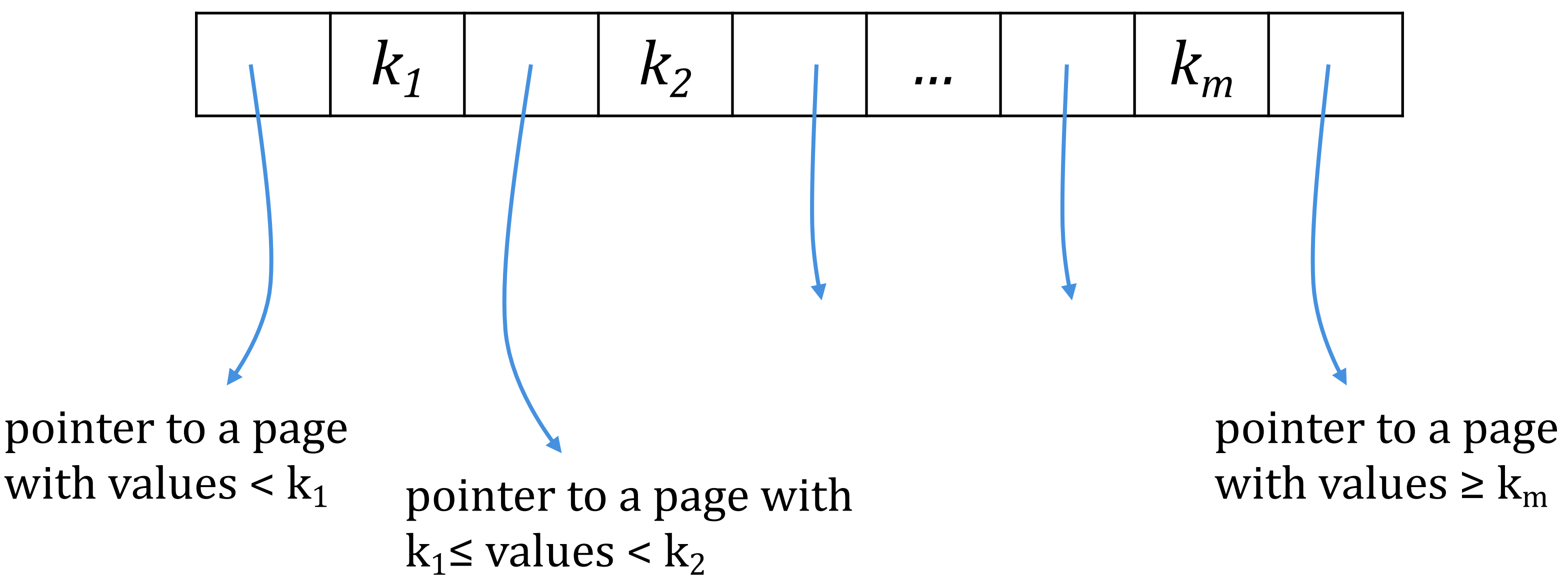
B+ TREE: NODE

- the parameter ***d*** is the *order* of the tree
- each node contains $d \leq m \leq 2d$ entries
  - – minimum 50% occupancy always
- with the exception of the root node, which can have $1 \leq m \leq 2d$ entries

	k_1		k_2		…		k_m	

NON-LEAF NODES

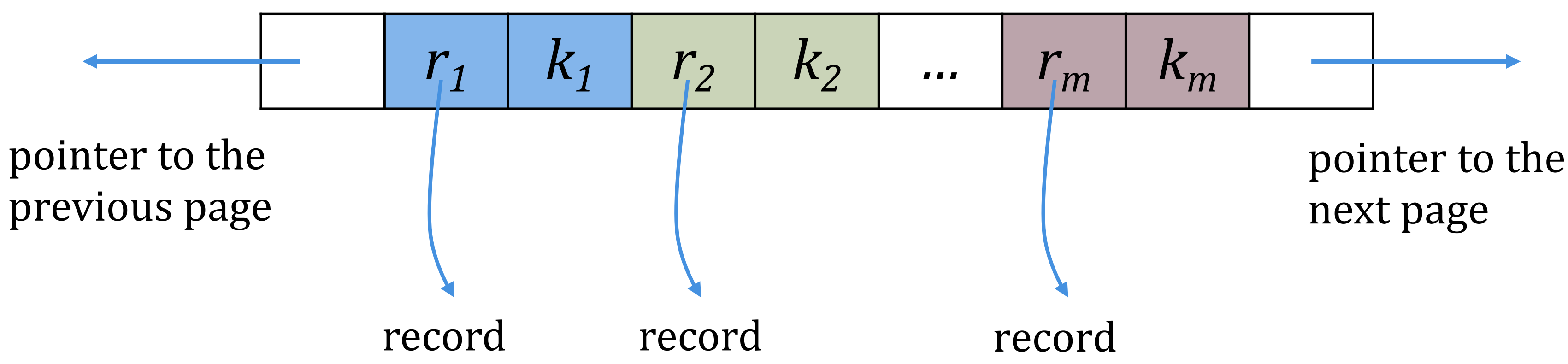
A non-leaf (or internal) node with *m* entries has *m+1* pointers to lower-level nodes



LEAF NODES

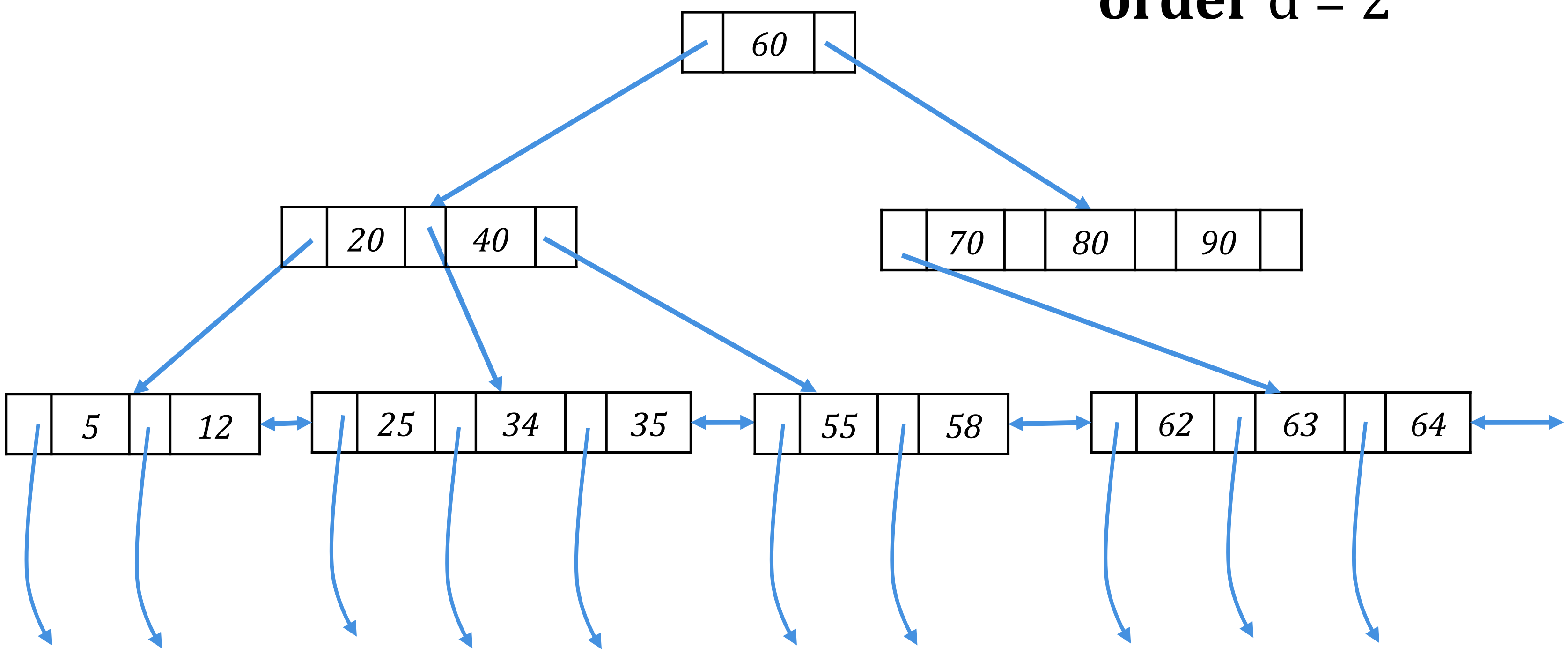
A leaf node with m entries has

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



B+ TREE EXAMPLE

order d = 2



B+ Tree Operations

B+ TREE OPERATIONS

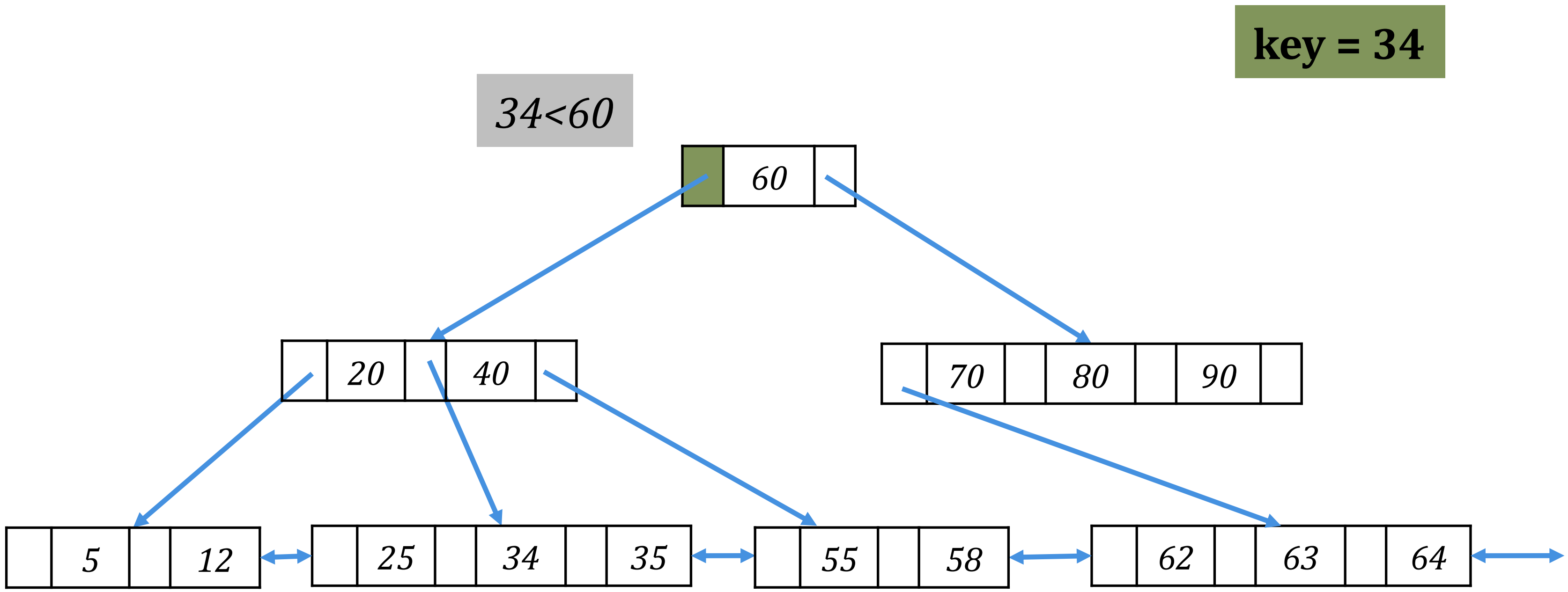
A B+ tree supports the following operations:

- equality search
- range search
- insert
- delete
- bulk loading (not covered in this lecture)

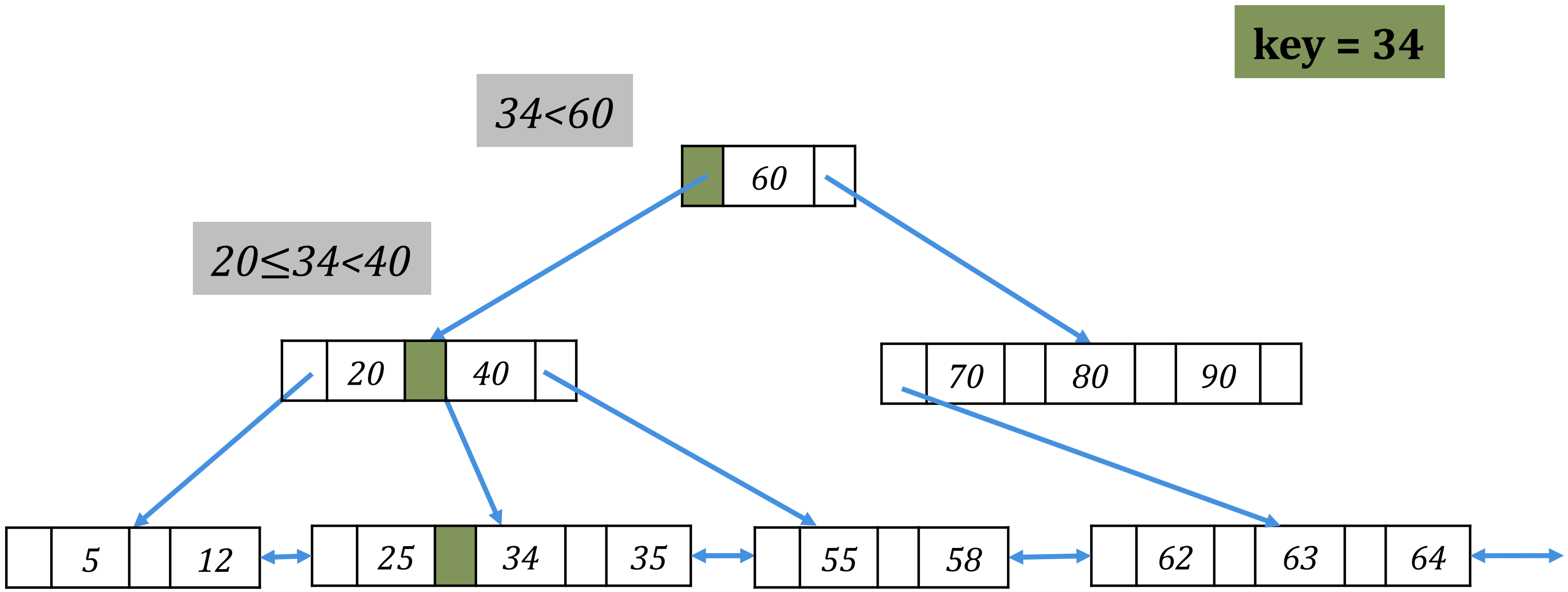
SEARCH

- 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

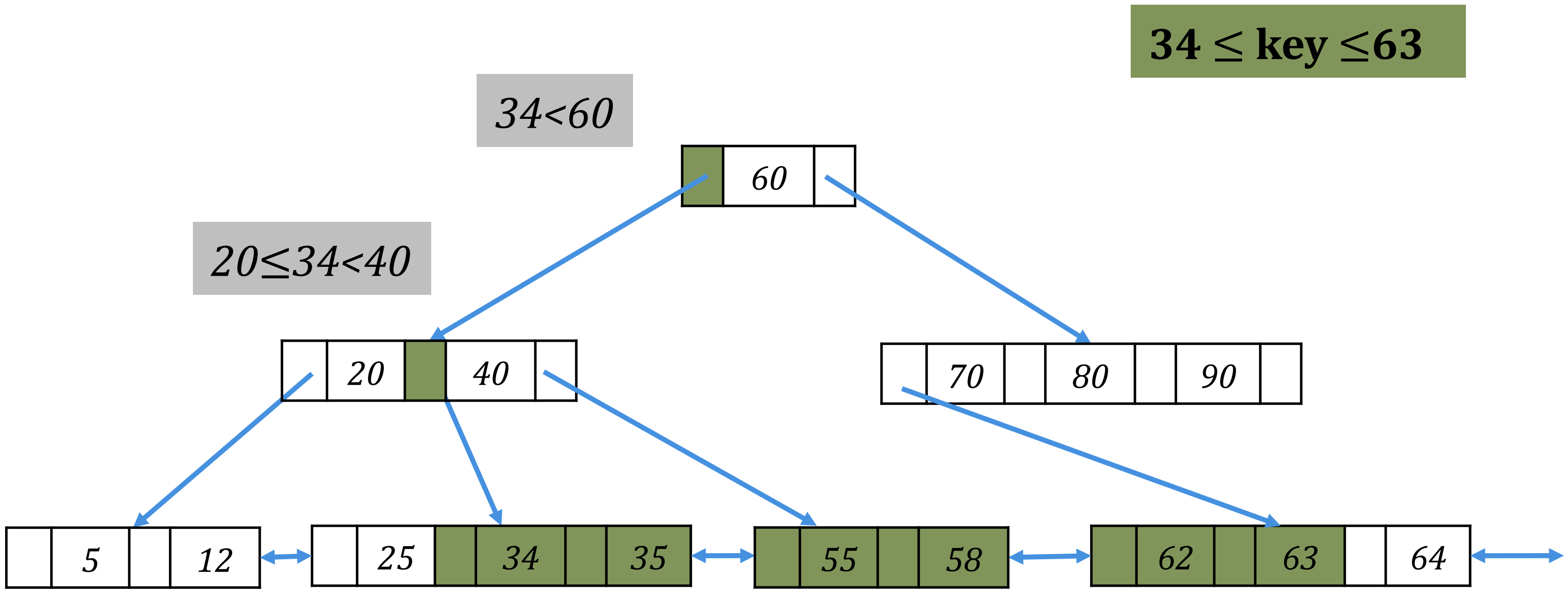


EQUALITY SEARCH: EXAMPLE



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!

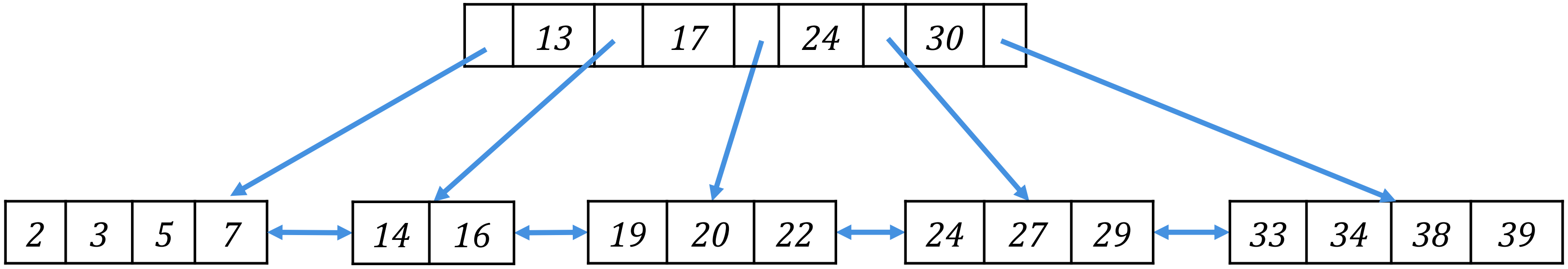
INSERT

- 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'**)
    - redistribute entries evenly, **copy up** the middle key
    - insert index entry pointing to **L'** into parent of **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 8

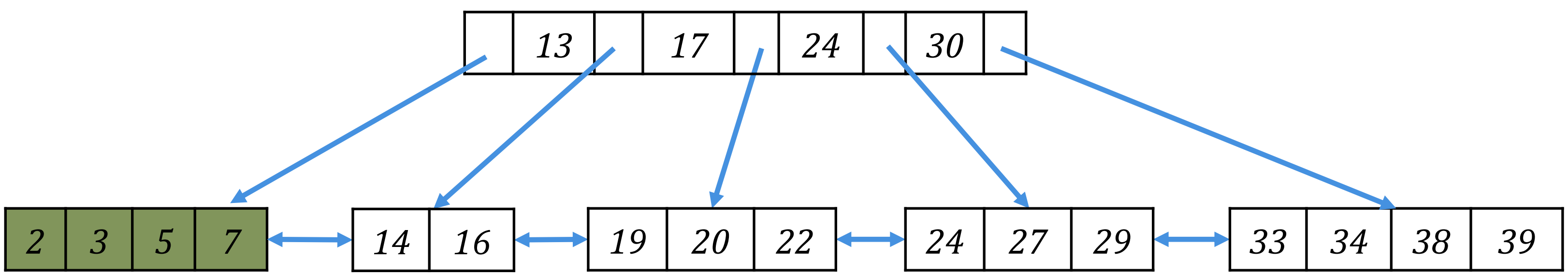


INSERT: EXAMPLE

order **d** = 2

insert 8

the leaf node is full so we must split it!



d entries

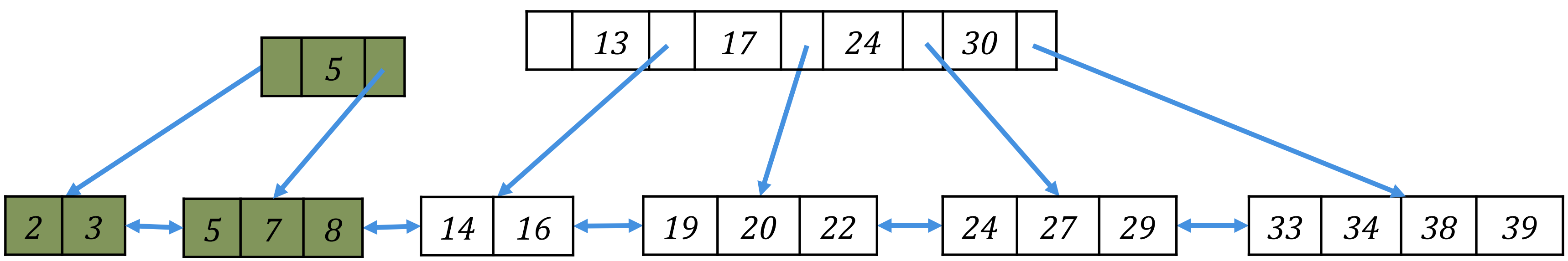
d+1 entries

INSERT: EXAMPLE

order **d** = 2

insert 8

the middle key (***5***) must be copied up, but the root node is full as well!

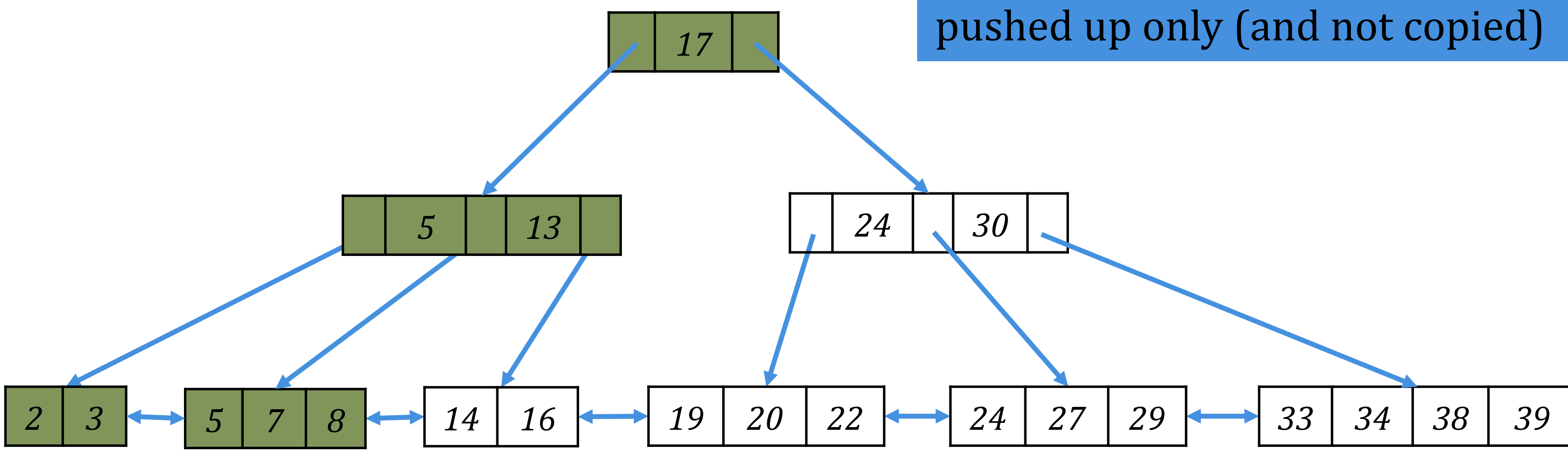


INSERT: EXAMPLE

order **d** = 2

insert 8

the new middle key (***17***) is now pushed up only (and not copied)



INSERT PROPERTIES

The B+ tree insertion algorithm has several attractive qualities:

- about the same cost as exact search
- it is ***self-balancing:*** the tree remains balanced (with respect to height) even after multiple insertions

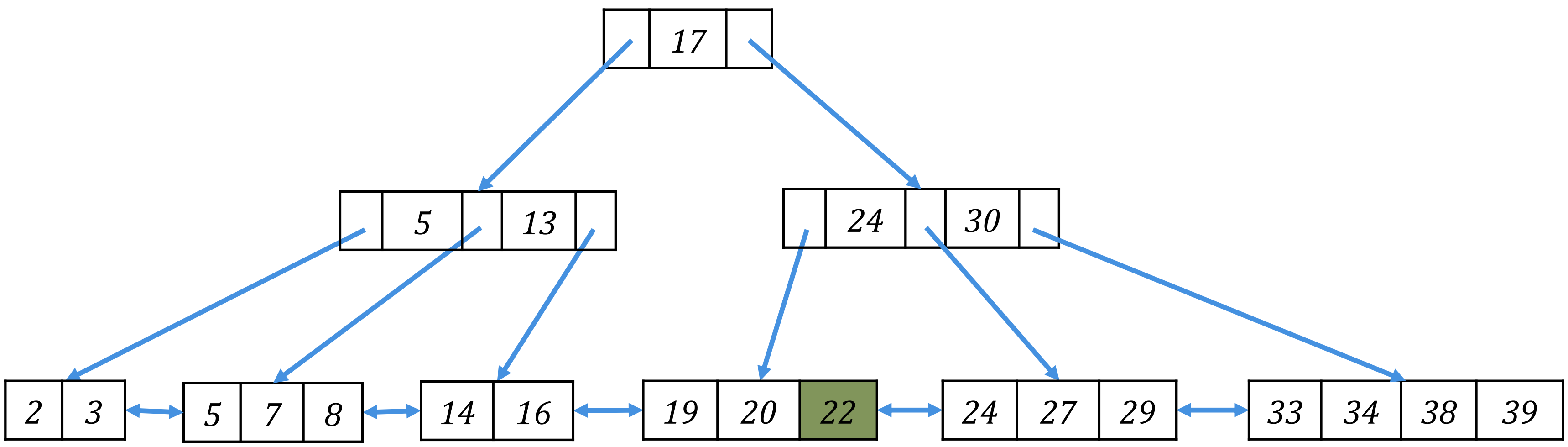
DELETE

- find the leaf node **L** where 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 1

order **d** = 2

delete 22

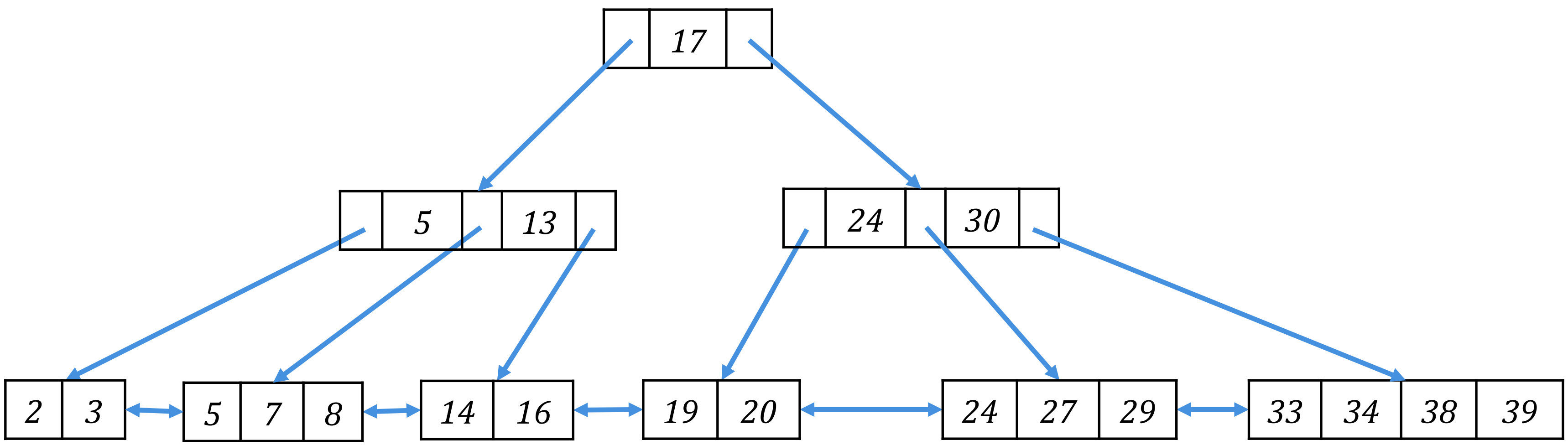


since by deleting 22 the node remains half-full, we simply remove it

DELETE : EXAMPLE 1

order **d** = 2

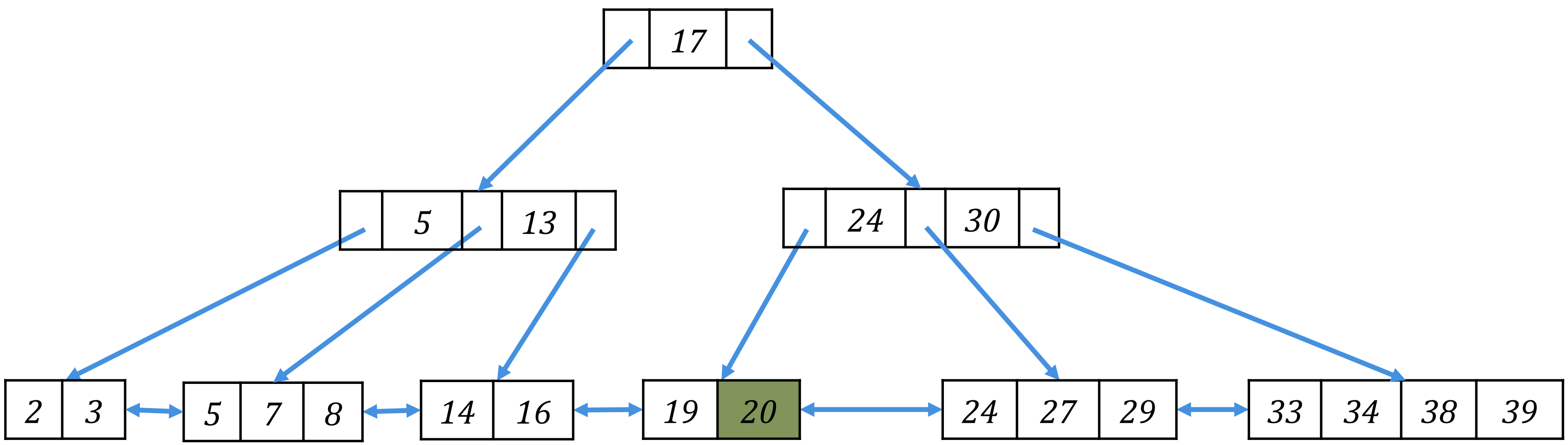
delete 22



DELETE : EXAMPLE 2

order **d** = 2

delete 20

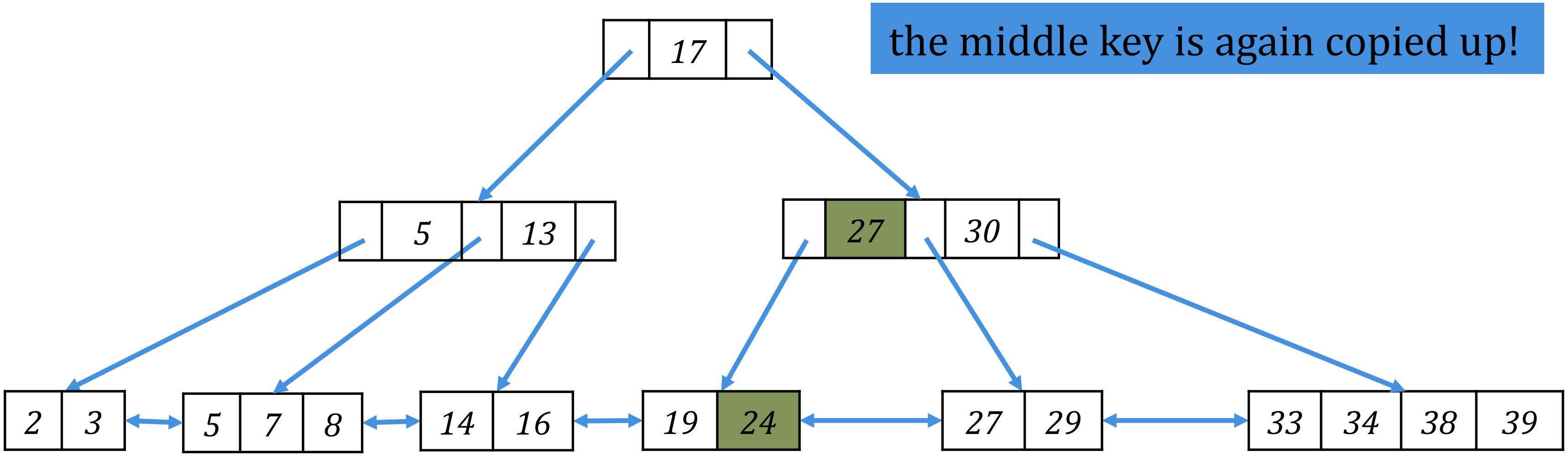


by removing 20 the node is not half-full anymore, so we attempt to redistribute!

DELETE : EXAMPLE 2

order **d** = 2

delete 20



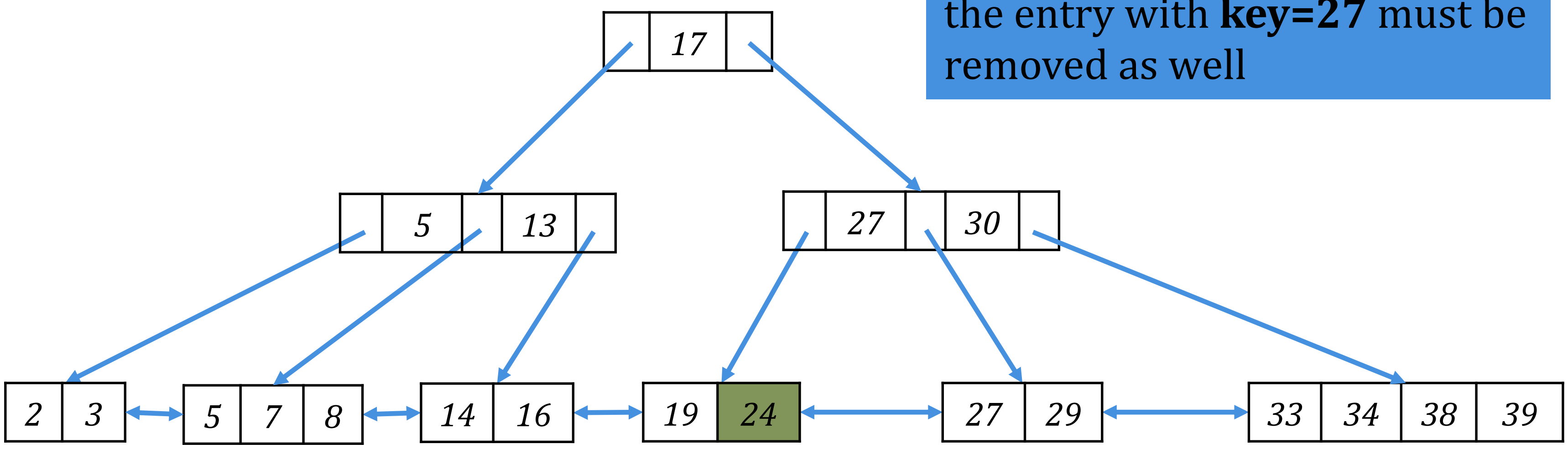
the middle key is again copied up!

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

DELETE : EXAMPLE 3

order **d** = 2

delete 24



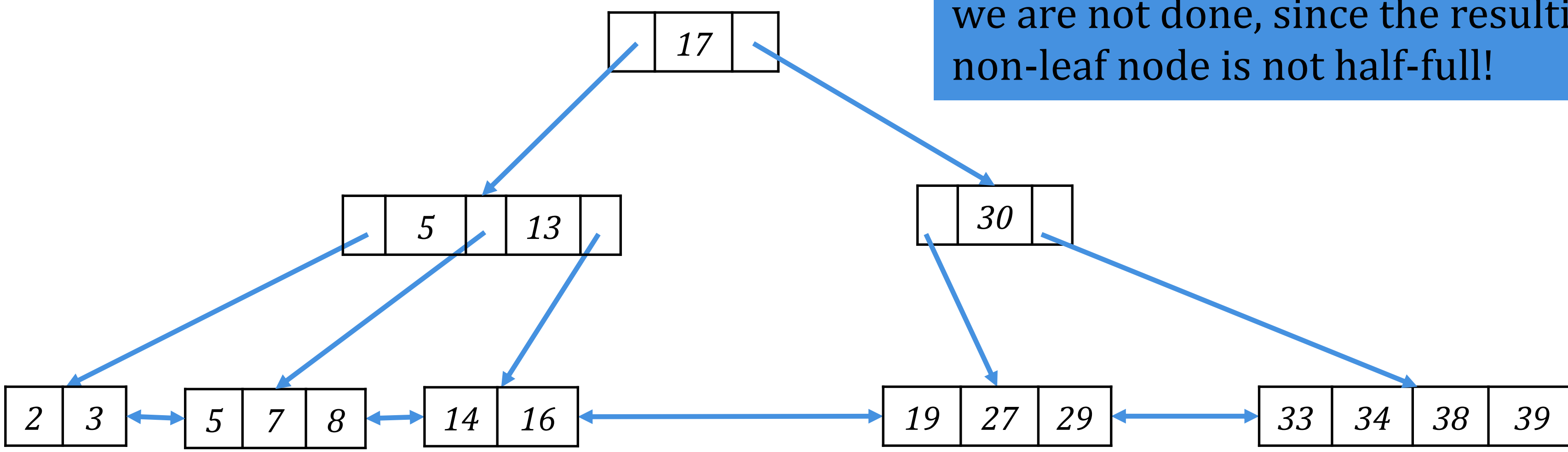
the entry with **key=27** must be removed as well

in this case, we have to merge nodes!

DELETE : EXAMPLE 3

order **d** = 2

delete 24



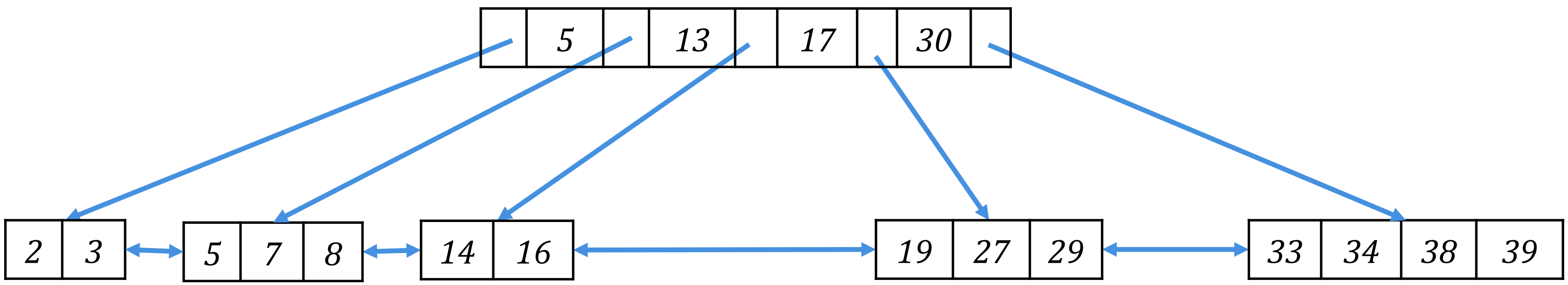
we are not done, since the resulting non-leaf node is not half-full!

DELETE : EXAMPLE 3

order **d** = 2

delete 24

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

DUPLICATES

Duplicate keys: many index entries with the same key value

- Solution #1
  - All entries with a given key value reside on a single page
  - Use overflow pages
- Solution #2
  - Allow duplicate key values in data entries
  - Modify search operation

B+ Tree Design & Cost

B+ TREE: FAN-OUT

fan-out f: the number of pointers to child nodes coming out of a non-leaf node

- compared to binary trees (fan-out $=2$), B+ trees have a high fan-out ($d+1 \leq f \leq 2d+1$)
- The fan-out of B+ trees is dynamic, but we will typically assume it is constant for our cost model

B+ TREE: FILL-FACTOR

<u>fill-factor</u> ***F***: the percent of available slots in the B+ tree that are filled

- it is usually < 1 to leave slack for (quicker) insertions!
- typical fill factor F = 2/3

B+ TREE: HEIGHT

height ***h***: the number of levels of the non-leaf nodes

- the height is at least 1 (root node)
- high fan-out -> smaller height-> less I/O per search
- typical heights of B+ trees: 3 or 4

B+ TREE: EXAMPLE

- page size **P** = 4000 bytes
- search key size = 30 bytes
- address size = 10 bytes
- fill-factor **F** = 2/3
- number of records = 2,000,000

- We assume that the index entries store only the search key and the address of tuple
- We assume no duplicate entries

B+ TREE: EXAMPLE

What is the order **d** and fan-out **f** ?

- each non-leaf node stores up to 2**d** values of the key + (2**d**+1) addresses for the children pages
- to fit this into a single page, we must have:

$$2d \cdot 30 \ + (2d + 1) \cdot 10 \ \leq \ 4000$$
$$\boldsymbol{d} \leq 50$$

- since a maximum capacity node has (2**d**+1) = 101 children, and the fill-factor is 2/3, the fan-out is $\boldsymbol{f} = 101 \ * \frac{2}{3} = 67$

B+ TREE: EXAMPLE

How many leaf pages are in the B+ tree?

- we assume for simplicity that each leaf page stores only pairs of (key, address)
- each pair needs 30+10 = 40 bytes
- to store 2,000,000 such pairs with fill-factor **F** = 2/3, we need:

$$\#leaves = (2{,}000{,}000 * 40)/(4{,}000 * \boldsymbol{F}) = 30{,}000$$

B+ TREE: EXAMPLE

What is the height **h** of the B+ tree?

- we calculated that we need to index **N** = 30,000 pages
- **h** = 1 -> indexes **f** pages
- **h** = 2 -> indexes $\mathbf{f}^2$ pages
- …
- **h** = k -> indexes $\mathbf{f}^k$ pages

height must be $\boldsymbol{h} = \lceil log_f N \rceil$

for our example, $\boldsymbol{h} = \lceil log_{67} 30{,}000 \rceil = 3$

B+ TREE: EXAMPLE

What is the total size of the tree?

- #pages = 1 + 67 + 67^2 + 30,000 = 34,557
- the top levels of the B+ tree do not take much space and can be kept in the buffer pool
  - *level 0* = 1 page ~ 4 KB
  - *level 1* = 67 pages ~ 268 KB
  - *level 2* = 4,489 pages ~ 18 MB

COST MODEL FOR SEARCH

To do equality search:

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

I = 0 if the record is stored at the leaf node, otherwise I = 1

$$\text{I/O cost } = \boldsymbol{h} - L_B + 1 + I$$

If we have $\boldsymbol{B}$ available buffer pages, we can store $\boldsymbol{L}_\mathrm{B}$ levels of the B+ Tree in memory:

- $\boldsymbol{L}_\mathrm{B}$ is the number of levels such that the sum of all the levels' nodes fit in the buffer:

$$B \geq 1 + f + \cdots + f^{L_B - 1}$$

COST MODEL FOR SEARCH

To do range search:

- we read one page per level of the tree
- levels that we can fit in buffer are free!
- we read sequentially the pages in the range

$$\text{I/O cost } = \mathrm{h} - L_B + OUT$$

Here, OUT is the I/O cost of loading the additional leaf nodes we need to access + the I/O cost of loading each ***page*** of the results