THE B+ TREE INDEX

CS 564- Spring 2018

ACKs: *Jignesh Patel, AnHai Doan*

WHAT IS THIS LECTURE ABOUT?

The **B+ tree** index

- Basics
- Search/Insertion/Deletion
- Design & Cost

INDEX RECAP

- We have the following query:

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

- How do we organize the file to answer this query efficiently?

INDEXES

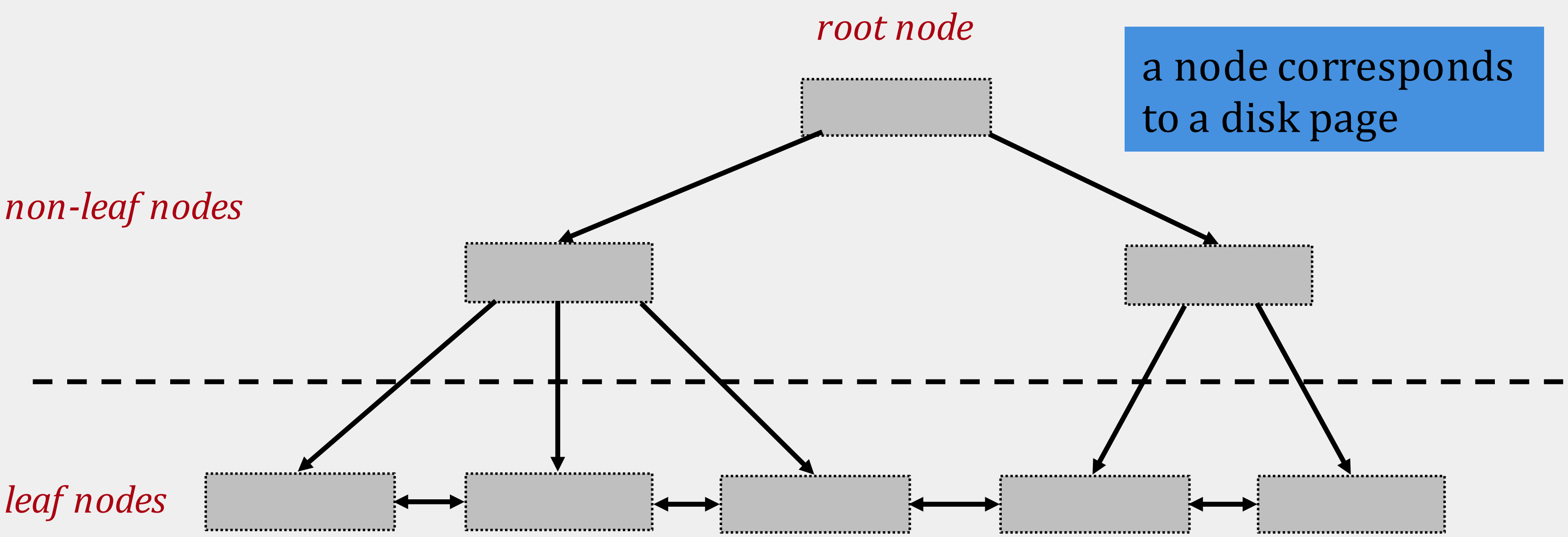
- Hash index:
  - good for equality search
  - in expectation constant I/O cost for search and insert
- B+ tree index:
  - good for **range** and **equality** search

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



data entries

- exist *only* in the leaf nodes
- are sorted according to the search key

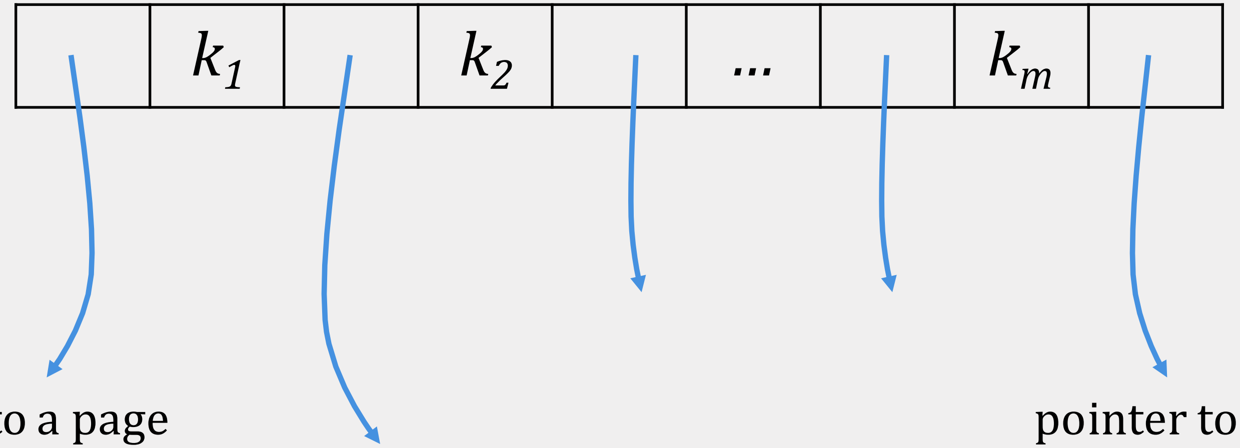
B+ TREE: NODE

- Parameter **d** is the *order* of the tree
- Each non-leaf node contains $d \leq m \leq 2d$ entries
  - minimum 50% occupancy at all times
- The root node can have $1 \leq m \leq 2d$ entries

	k_1		k_2		…		k_m	

NON-LEAF NODES

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



pointer to a page with values $< k_1$

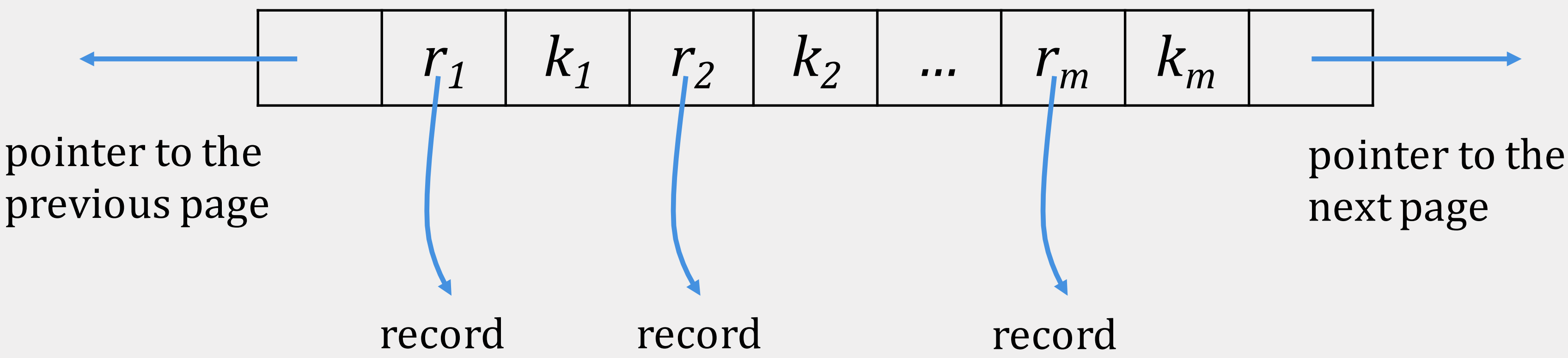
pointer to a page with $k_1 \leq$ values $< k_2$

pointer to a page with values $\geq k_m$

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 Operations

B+ TREE OPERATIONS

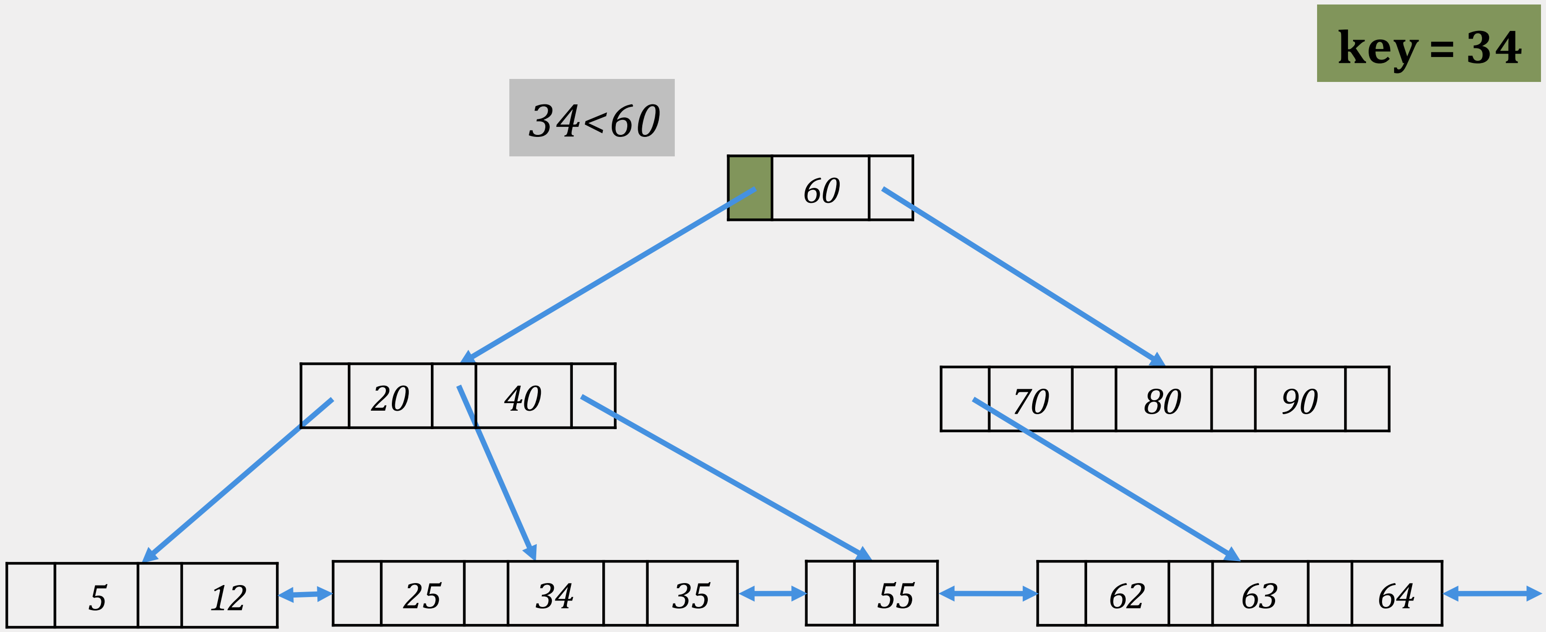
A B+ tree supports the following operations:

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

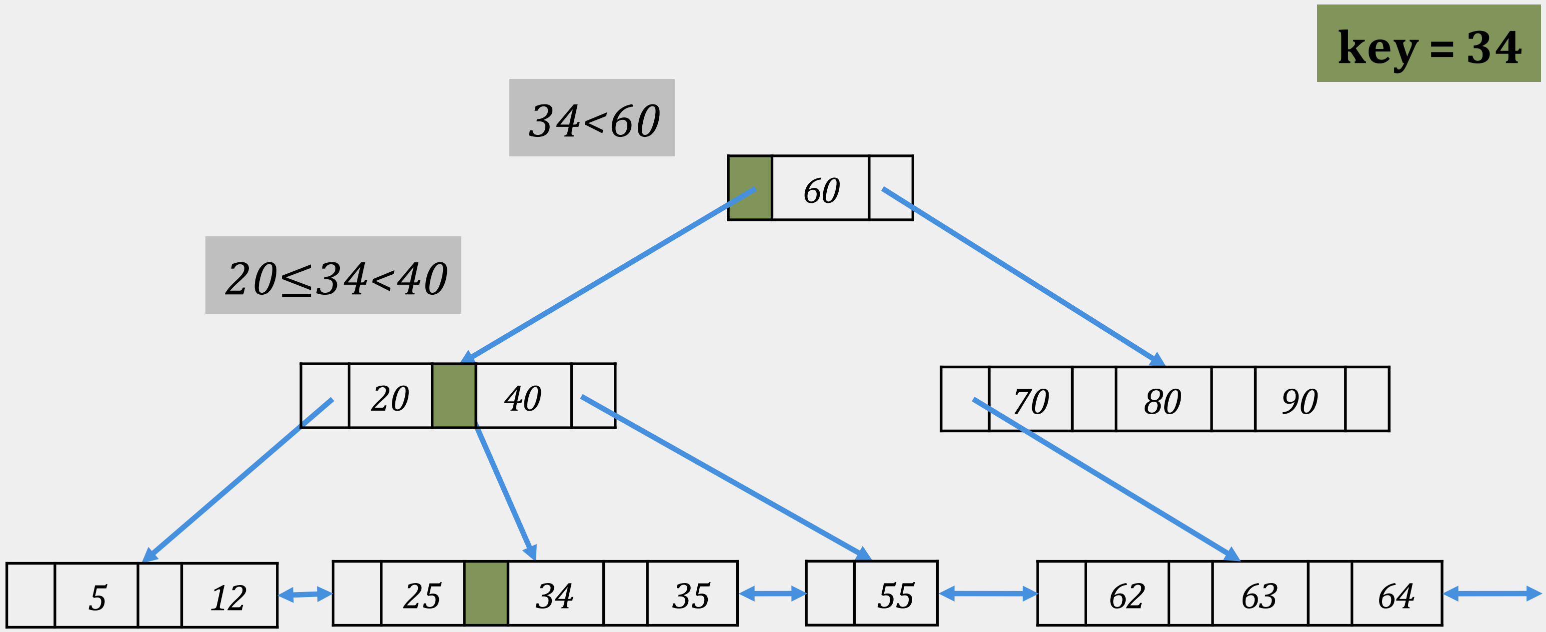
SEARCH

- 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

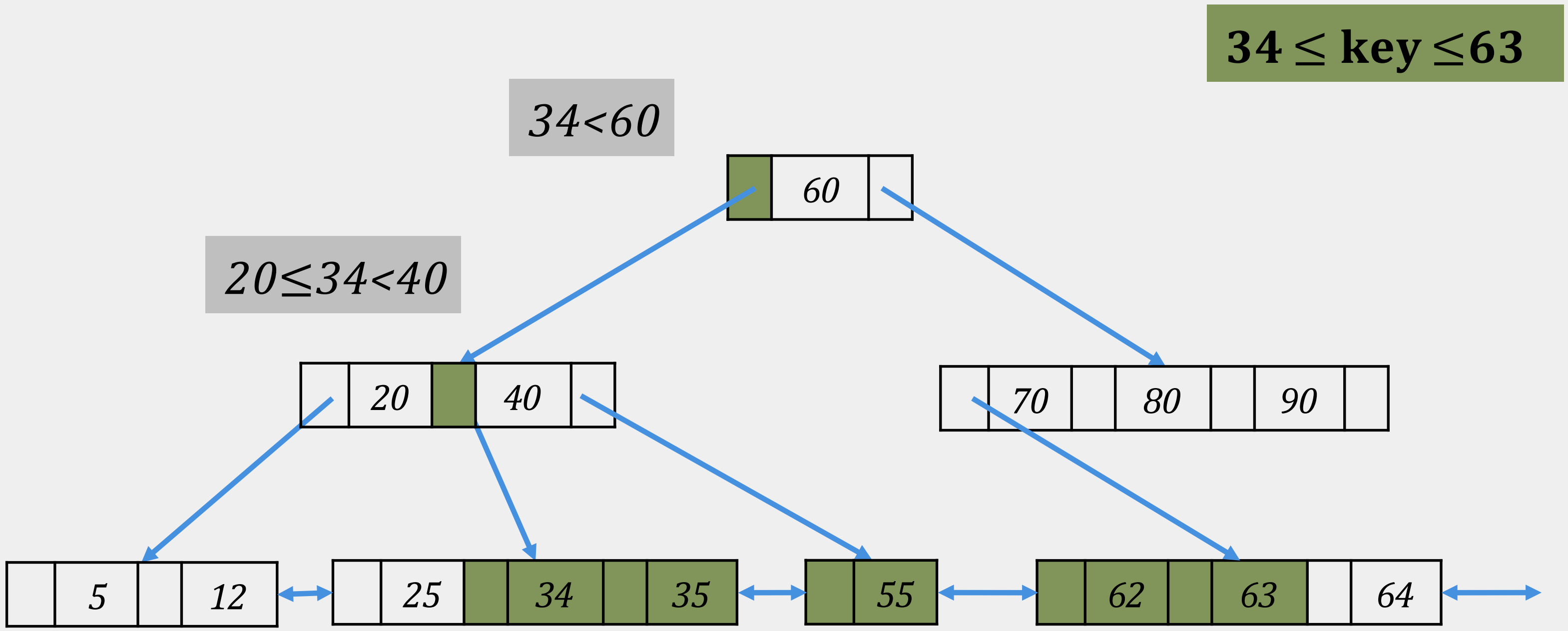


EQUALITY SEARCH: EXAMPLE



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!

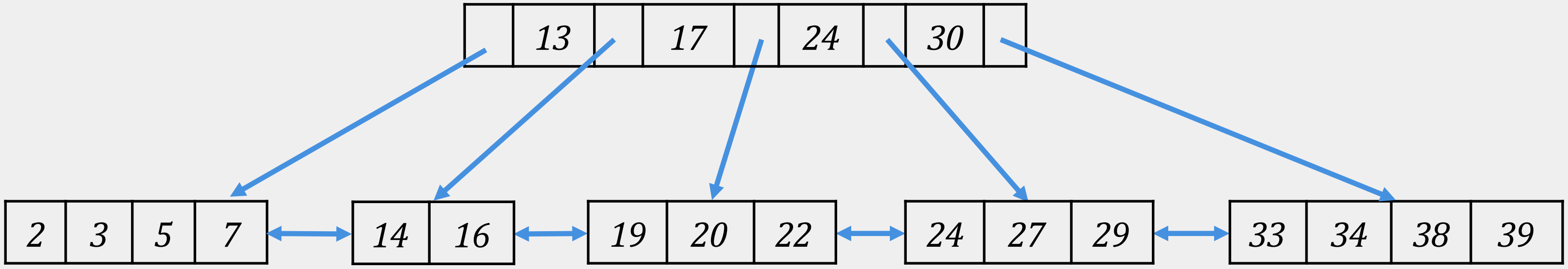
INSERT

- find correct leaf node **L**
- insert data entry in **L**
  - If **L** has enough space, DONE!
  - Else, 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

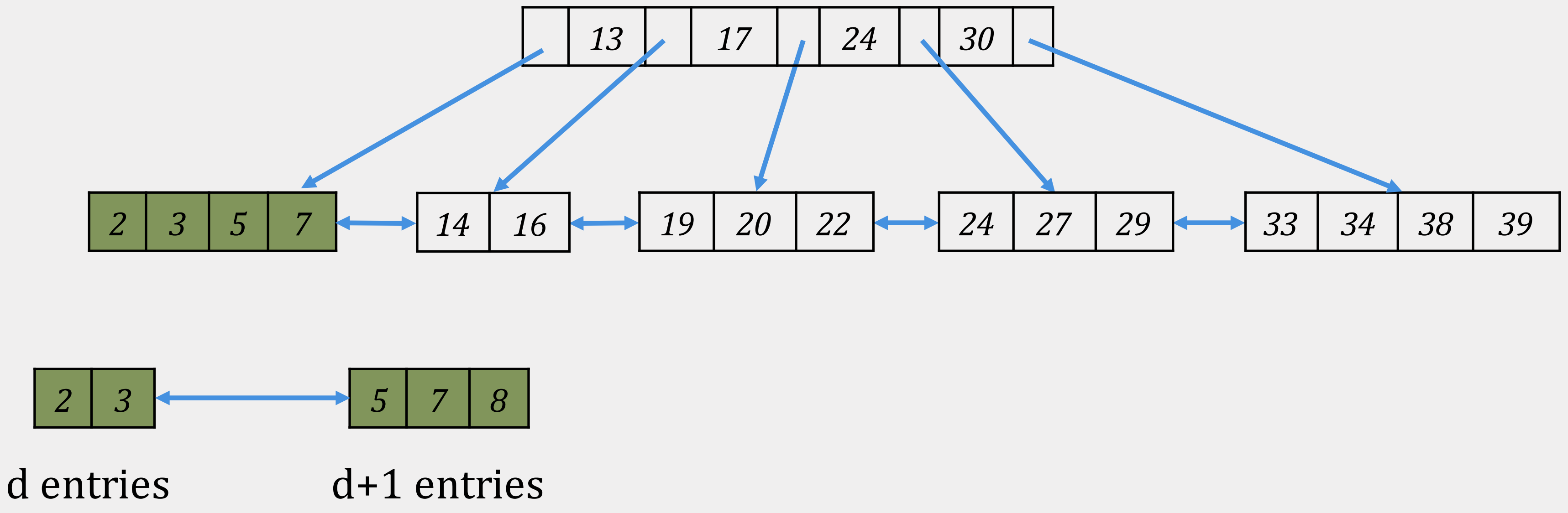


INSERT: EXAMPLE

order **d** = 2

insert 8

the leaf node is full so we must split it!

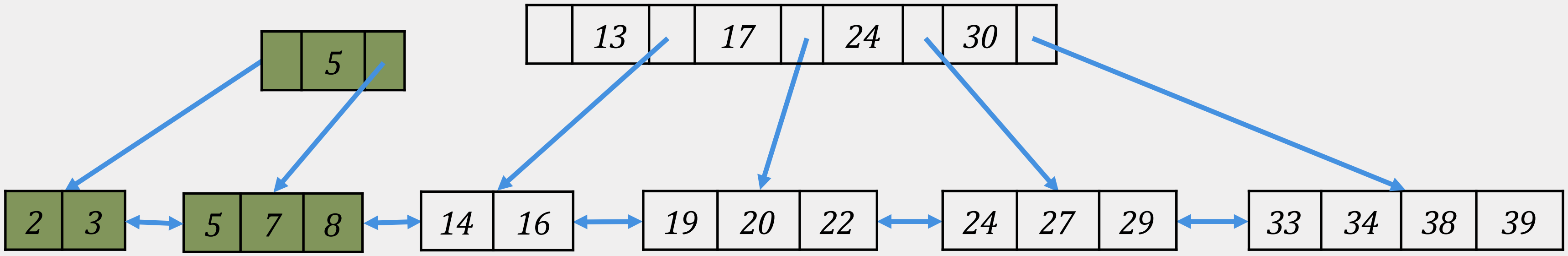


INSERT: EXAMPLE

order **d** = 2

insert 8

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

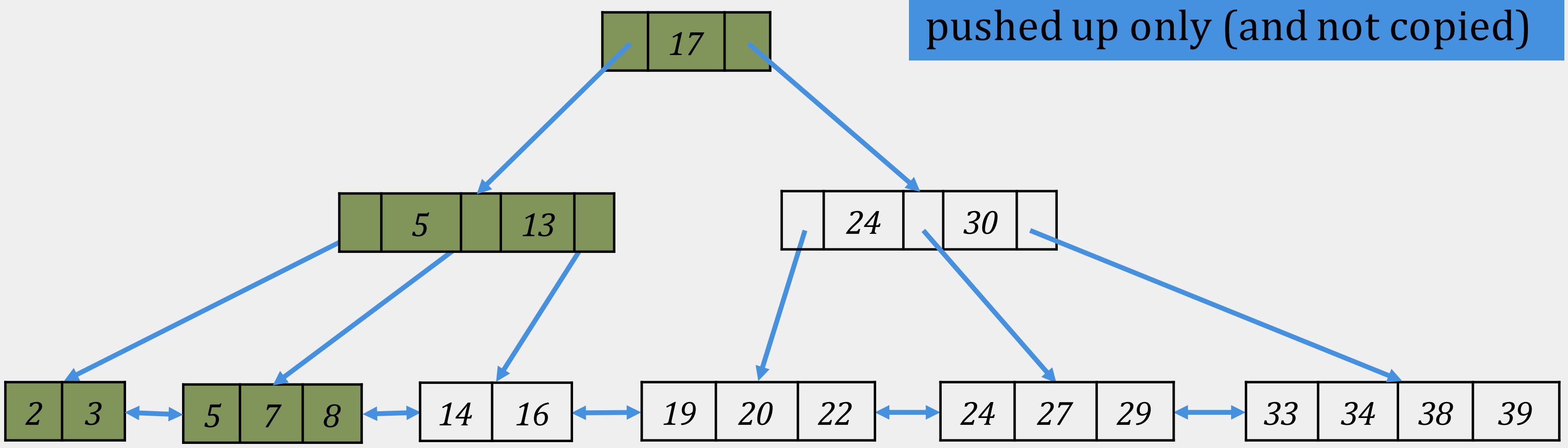


INSERT: EXAMPLE

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:

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

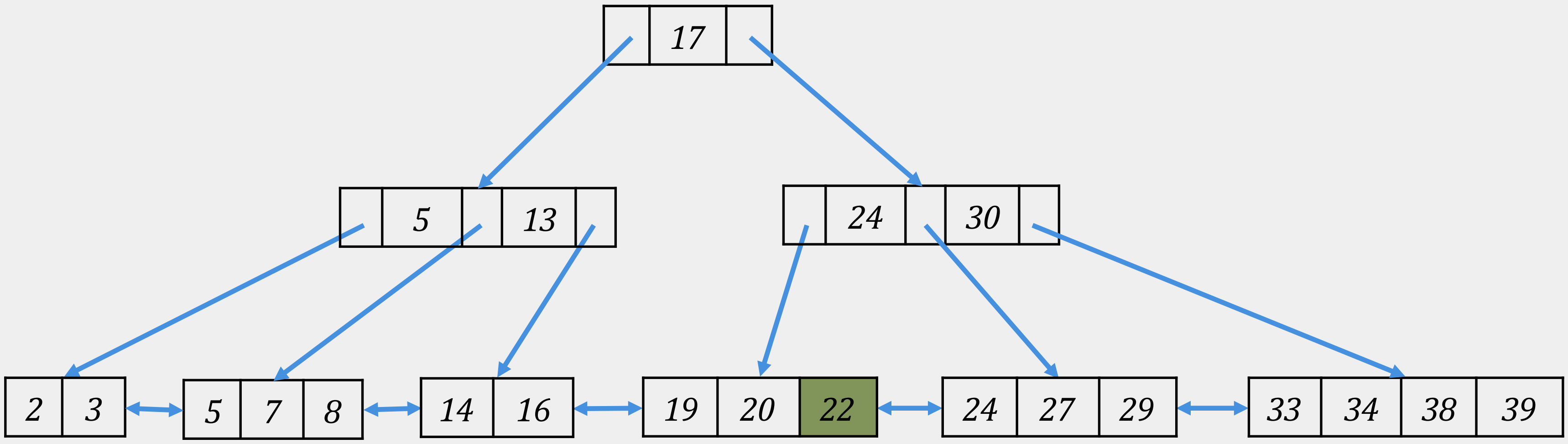
B+ TREE: DELETE

- find 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 **re-distribute**, borrowing from sibling
    - If re-distribution 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

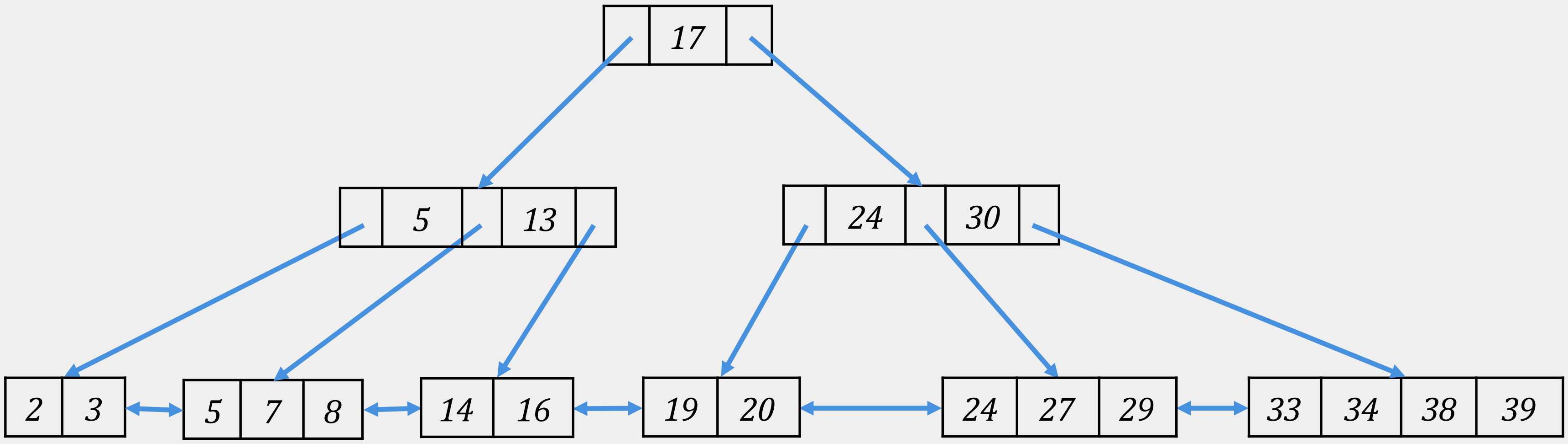


since by deleting 22 the node remains half-full, we simply remove it

DELETE : EXAMPLE 1

order **d** = 2

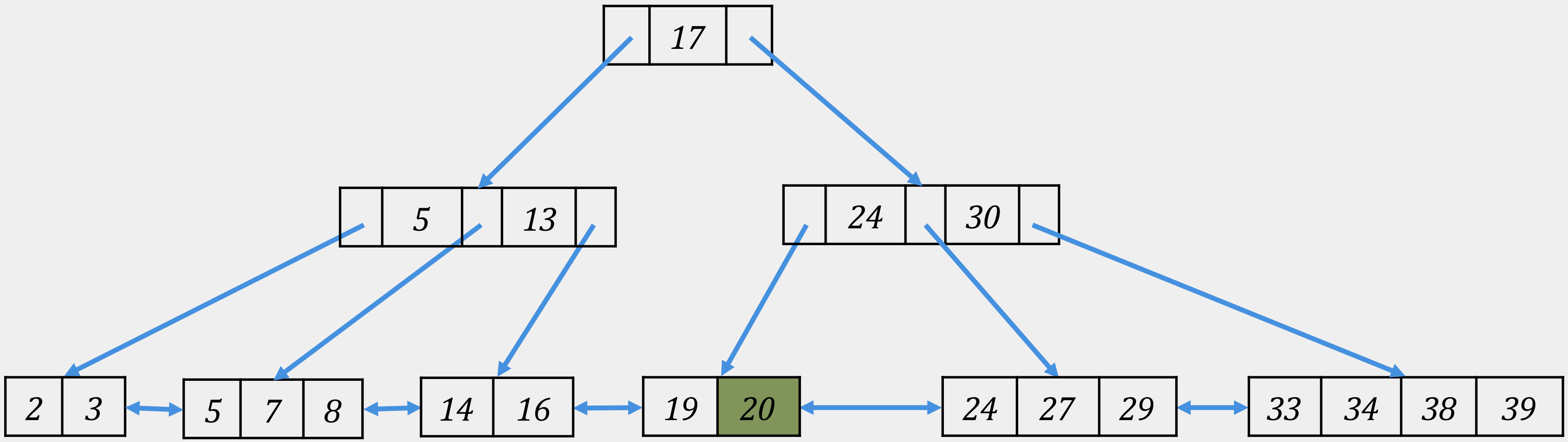
delete 22



DELETE : EXAMPLE 2

order **d** = 2

delete 20

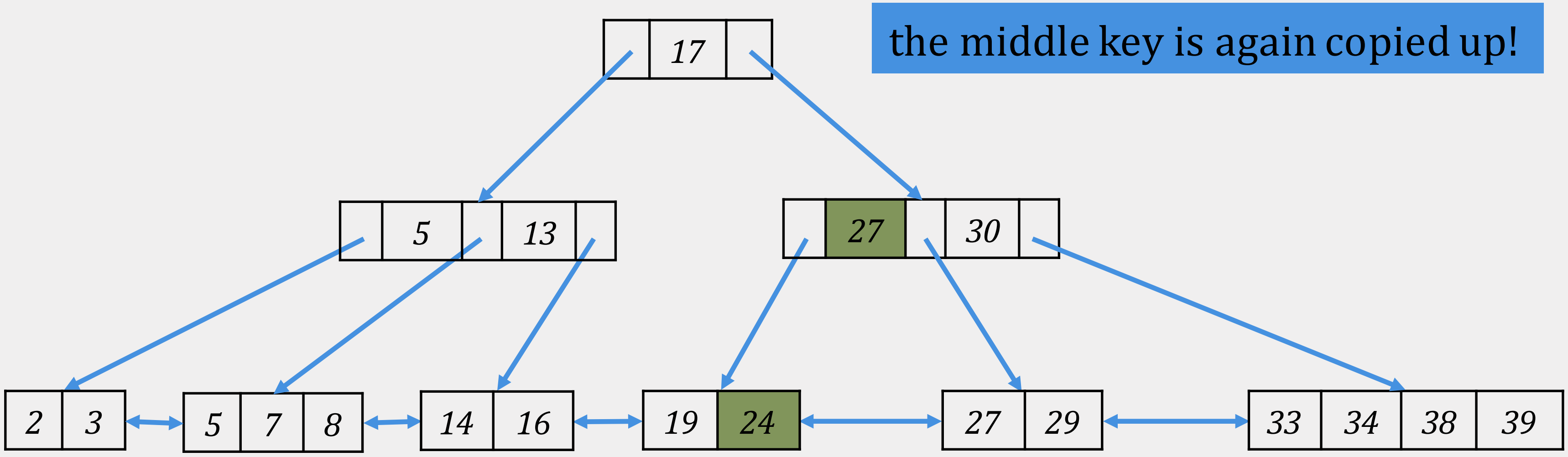


by removing 20 the node is not half-full anymore, so we attempt to redistribute!

DELETE : EXAMPLE 2

order **d** = 2

delete 20



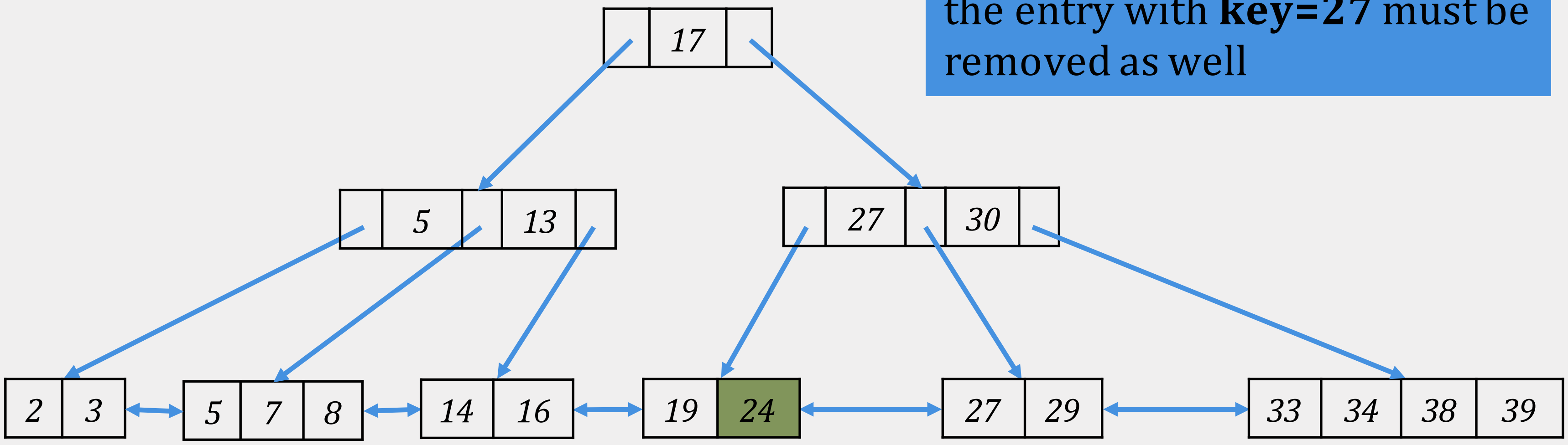
the middle key is again copied up!

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

DELETE : EXAMPLE 3

order **d** = 2

delete 24



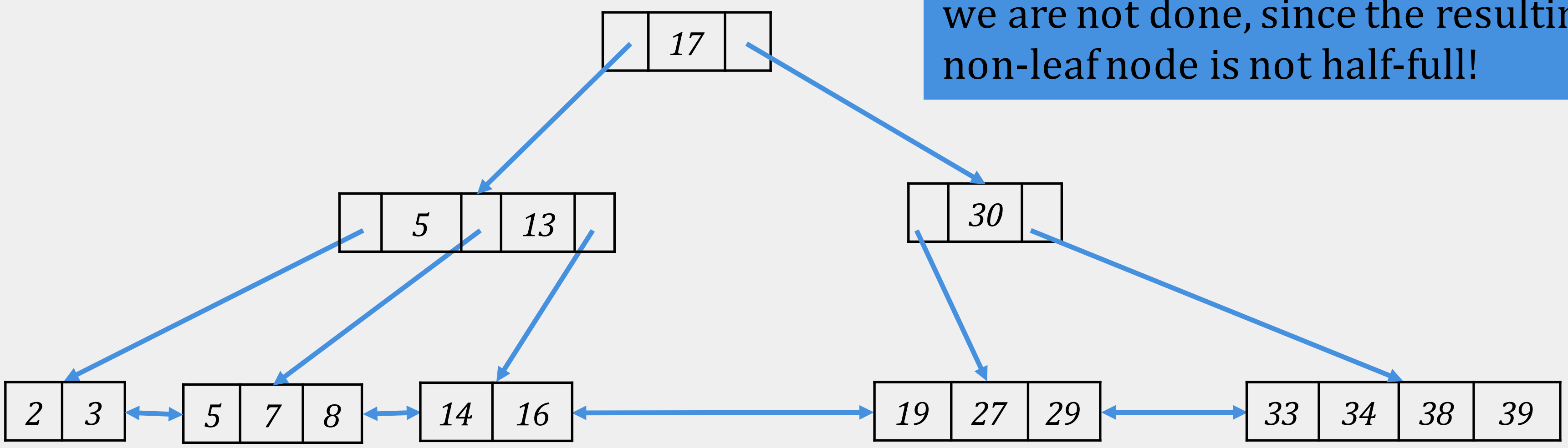
the entry with **key=27** must be removed as well

in this case, we have to merge nodes!

DELETE : EXAMPLE 3

order **d** = 2

delete 24



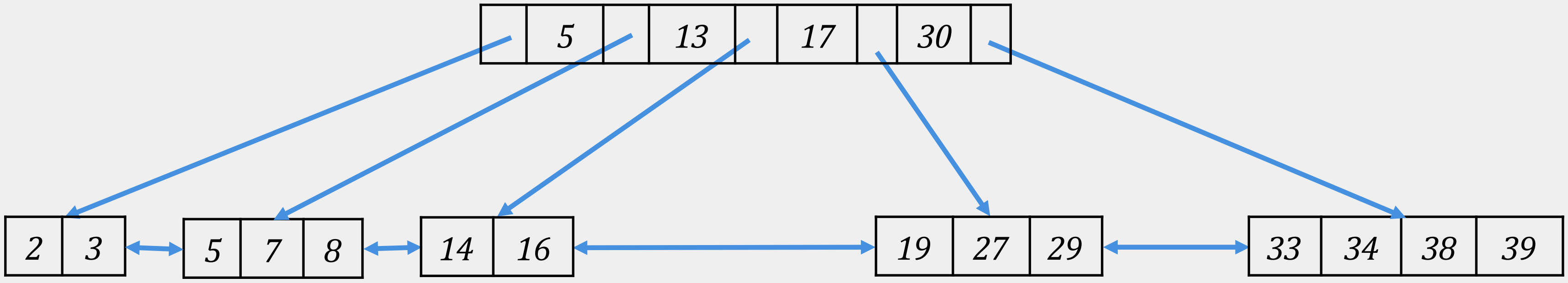
we are not done, since the resulting non-leaf node is not half-full!

DELETE : EXAMPLE 3

order **d** = 2

delete 24

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



B+ TREE: DELETE

- Redistribution of entries can also be 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 data 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 DESIGN

How large is ***d***?

- Example
  - key size = 4 bytes
  - pointer size = 8 bytes
  - block size = 4096 bytes

We want each *node* to fit on a single *block/page*

$$2d \cdot 4 + (2d + 1) \cdot 8 \leq 4096$$

$$\boldsymbol{d} \leq 170$$

B+ TREE: FANOUT

<u>Fanout</u>: the number of pointers to child nodes coming out of a node

- compared to binary trees (fanout =*2*), B+ trees have a high fanout (between *d+1* and *2d+1*)
- high fanout -> smaller depth -> less I/O per search
- The fanout of B+ trees is dynamic, but we will often assume it is constant to come up with approximate equations

B+ TREES IN PRACTICE

- typical order: **d** = 100
- typical fill factor = 67%
  - average node fanout = 133
- typical B+ tree capacities:
  - *height 4*: 133^4 = 312,900,700 records
  - *height 3*: 133^3 = 2,352,637 records
- It can often store the top levels in buffer pool:
  - *level 1* = 1 page = 8 KB
  - *level 2* = 133 pages = 1 MB
  - *level 3* = 17,689 pages = 133 MB

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

COST MODEL FOR SEARCH

Parameters:

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

We need to index N/F pages. For different heights:

- h = 1 -> f pages
- h = 2 -> f^2 pages
- h = k -> f^k pages

height must be $h = \left\lceil \log_f \frac{N}{F} \right\rceil$

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

$$\text{I/O cost } = \left\lceil log_f \frac{N}{F} \right\rceil - L_B + 1$$

If we have ***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

$$\text{I/O cost } = \left\lceil log_f \frac{N}{F} \right\rceil - L_B + OUT$$

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