Lecture 13:
B+ Trees:
An IO-Aware Index Structure

What you will learn about in this section

1. B+ Trees: Basics
2. B+ Trees: Design & Cost
3. Clustered Indexes

Building our 1st index

Person(name, age)

Query: Search for people of specific age

Design idea #1:

- Sort records by age…(fast)
- How many IO operations to search over N sorted records?
  - Simple scan: O(N)
  - Binary search: $O(\log_2 N)$

Could we get even cheaper search? E.g. go from
$\log_2 N \rightarrow \log_{200} N$?

Index Types

- B-Trees *(covered next)*
  - Very good for range queries, sorted data
  - Some old databases only implemented B-Trees
  - *We will look at a variant called **B+ Trees***

- Hash Tables
  - There are variants of this basic structure to deal with IO
  - Called ***linear*** or ***extendible hashing-*** IO aware!

These data structures are “IO aware”

Real difference between structures:
costs of ops *determines which index you pick and why*

B+ Trees

- Search trees
  - B does not mean binary!
- Idea in B Trees:
  - make 1 node = 1 physical page
  - Balanced, height adjusted tree (not the B either)
- Idea in B+ Trees:
  - Make leaves into a linked list (for range queries)

B+ Tree Basics

Person(<u>name</u>, age)

Example:
Sorted data

Name: Joe	Age: 11
Name: Jake	Age: 15
Name: John	Age: 21
Name: Bess	Age: 22
Name: Bob	Age: 27
Name: Sally	Age: 28
Name: Sal	Age: 30
Name: Sue	Age: 33
Name: Jess	Age: 35
Name: Alf	Age: 37

For simplicity

11	15	21	22	27	28	30	33	35	37

B+ Tree Basics

10	20	30

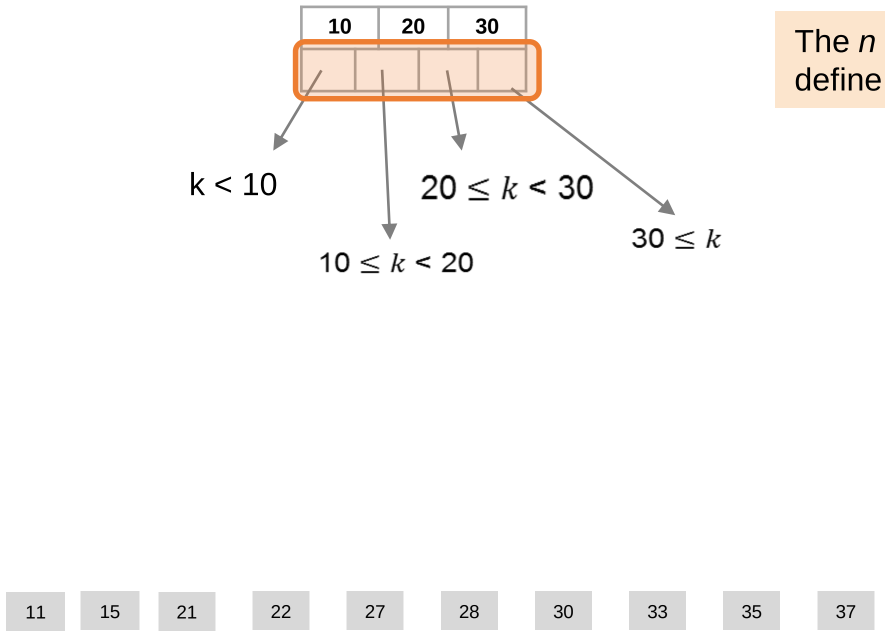
Parameter ***d*** = the degree

Each *non-leaf ("interior")* ***node*** has $\geq$ d and $\leq$ 2d ***keys****

except for root node, which can have between **1 and 2d keys*

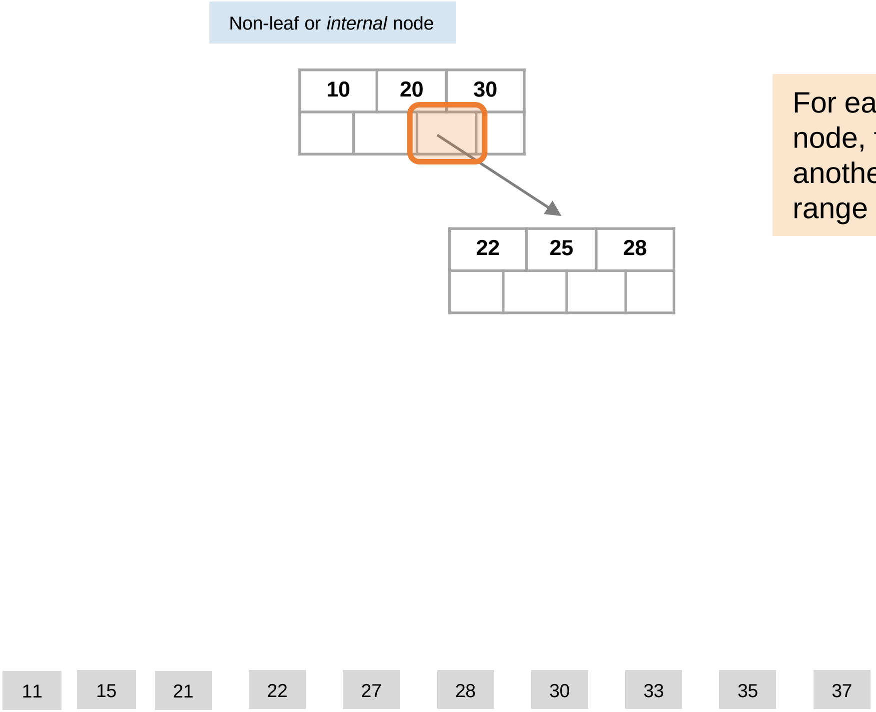
11 15 21 22 27 28 30 33 35 37

B+ Tree Basics



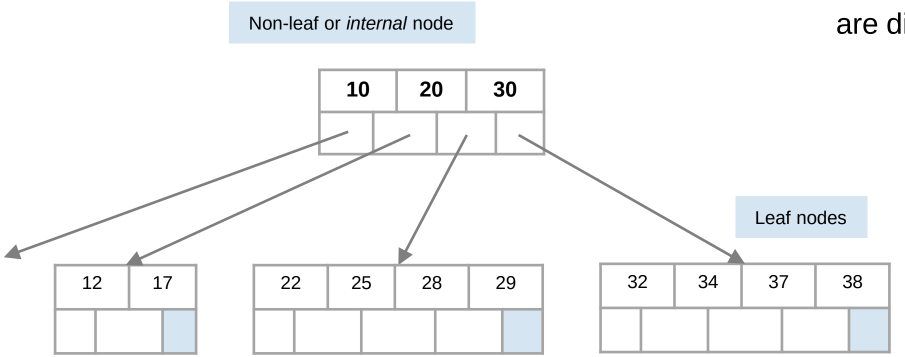
The *n* keys in a node define *n+1* ranges

B+ Tree Basics



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

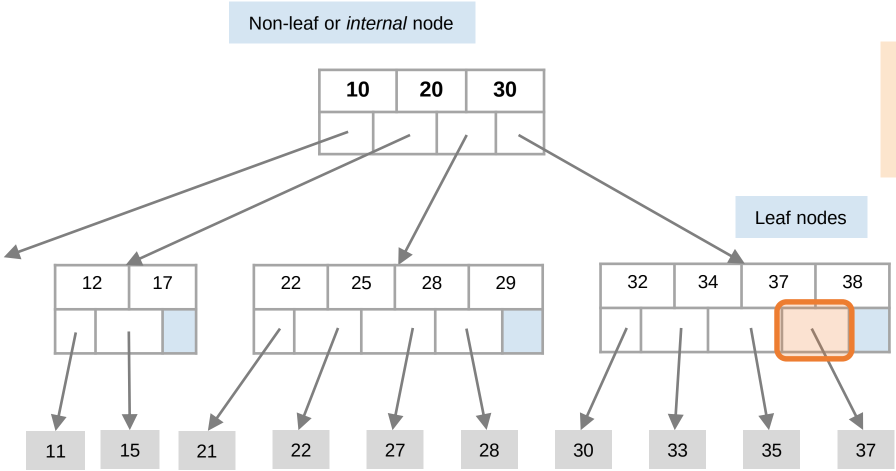
B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

11 15 21 22 27 28 30 33 35 37

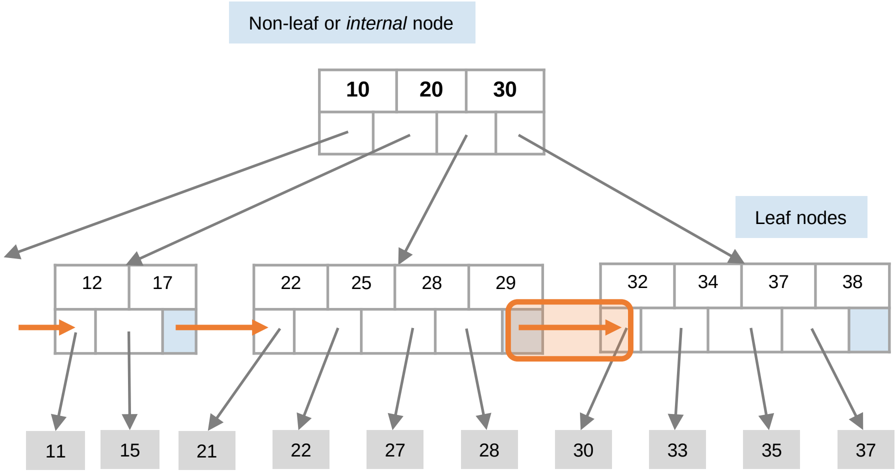
B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

Their key slots contain pointers to data records

B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

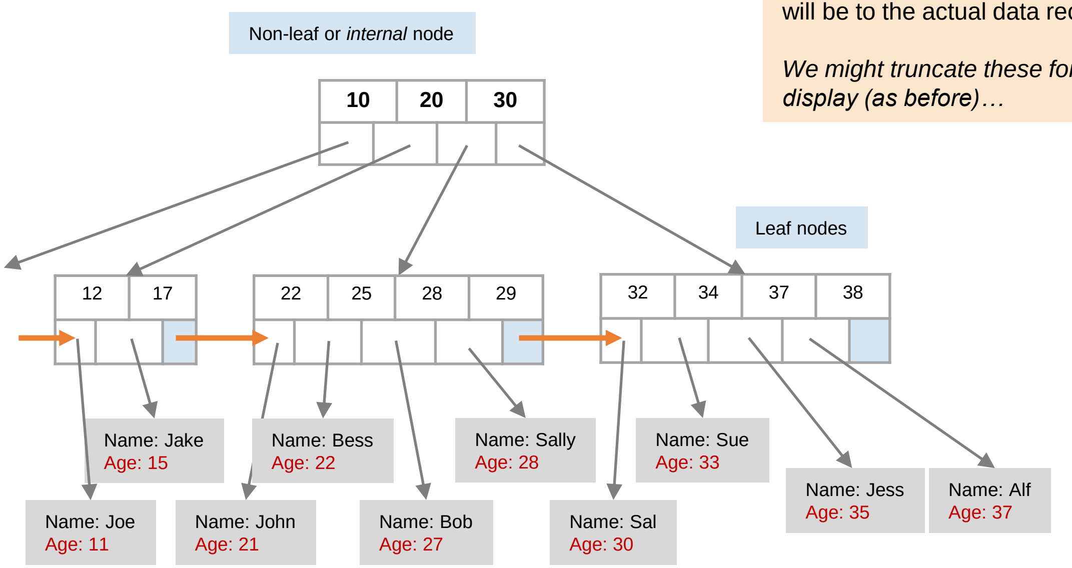
Their key slots contain pointers to data records

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 (as before)…



Some finer points of B+ Trees

Searching a B+ Tree

- For exact key values:
  - Start at the root
  - Proceed down, to the leaf

- For range queries:
  - As above
  - *Then sequential traversal*

```
SELECT name
FROM   people
WHERE  age = 25
```

```
SELECT name
FROM   people
WHERE  20 <= age
  AND  age <= 30
```

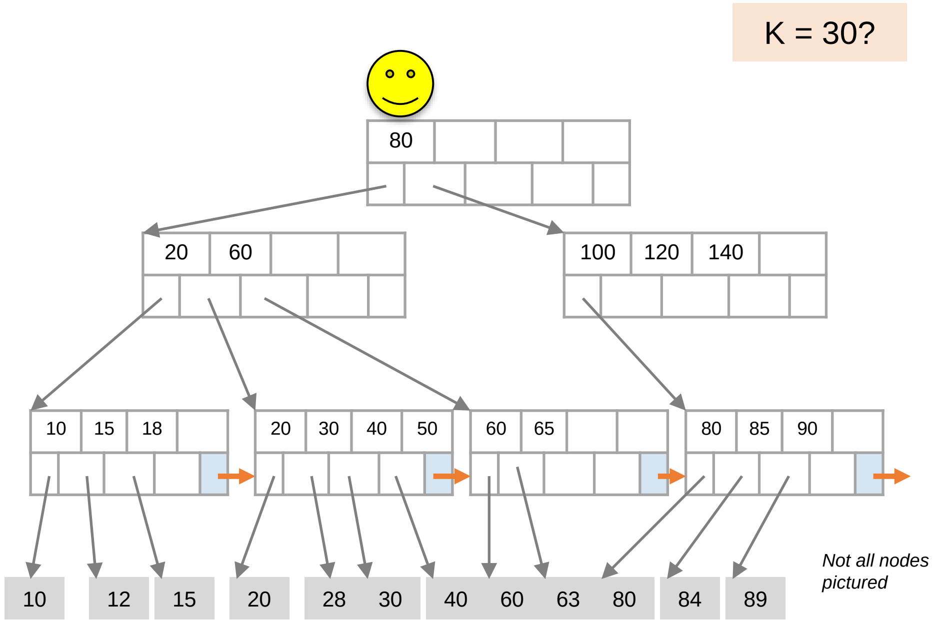
B+ Tree Exact Search Animation

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



B+ Tree Range Search Animation

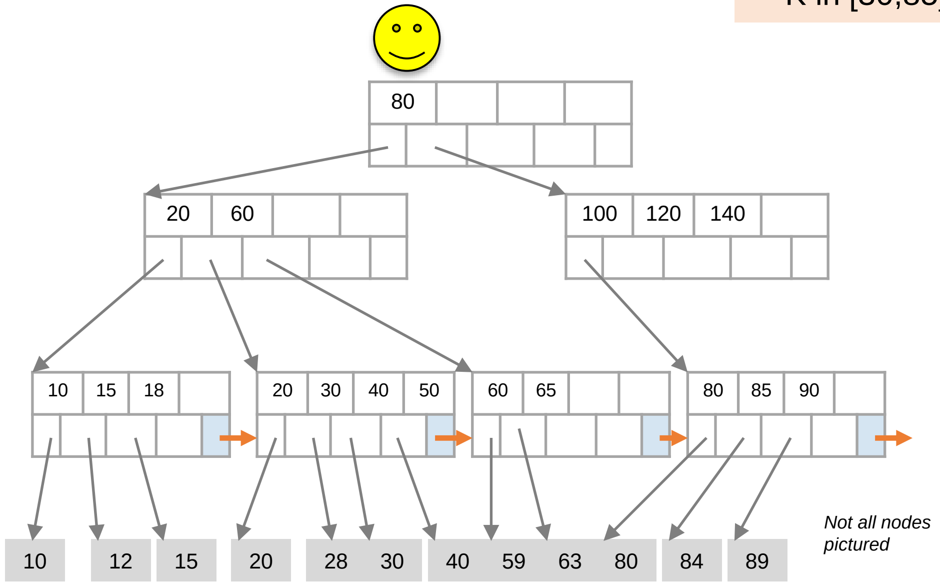
30 < 80

30 in [20,60)

30 in [30,40)

To the data!

K in [30,85]?



B+ Tree Design

- How large is ***d***?

- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 64k bytes

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

  2d x 4 + (2d+1) x 8 <= 64k → ***d ~= 2730***

  (keys) (pointers)

B+ Tree: High Fanout = Smaller & Lower IO

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

- Hence the **depth of the tree is small** → getting to any element requires very few IO operations!
  - Also can often store most/all of B+ Tree in RAM!

- A TiB = 2^{40} Bytes. What is the height of a B+ Tree (with fill-factor = 1) that indexes it (with 64K pages)?
  - $(2*2730 + 1)^h = 2^{40} \rightarrow \boldsymbol{h = 4}$

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
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Top levels of tree sit *in the buffer pool*:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

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

- Let:
  - ***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 $B \geq \sum_{l=0}^{L_B - 1} f^l$

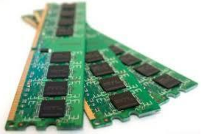
Simple Cost Model for Search

- 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)”

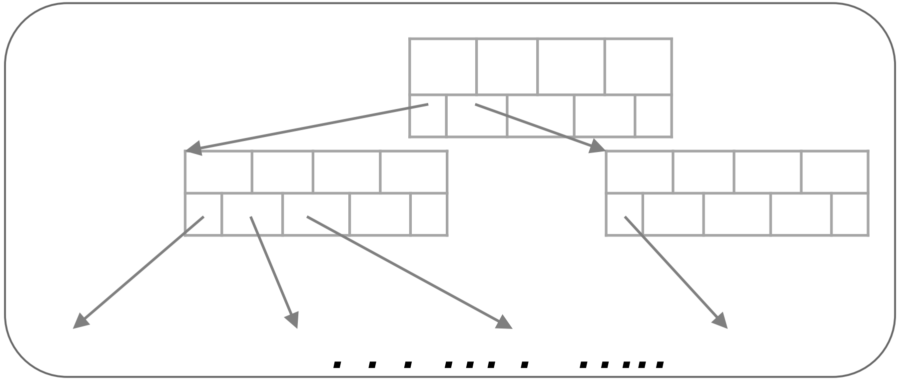
IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + Cost(OUT)$

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

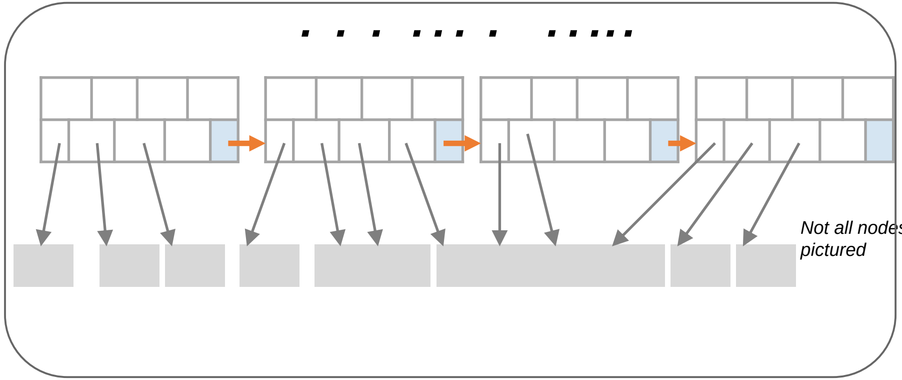
Search cost of B+ Tree (on RAM + Disk)



Read 1st levels
Into RAM buffer



Rest of index on disk



$1 + f + f^2 + f^3 + \ldots <= B$

Keep 1st L_B levels in RAM of size B

Algorithm: B+ Search
- Read 1 page per level
- Pages in RAM are free
- Read 1 page for record

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

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

Fast Insertions & Self-Balancing

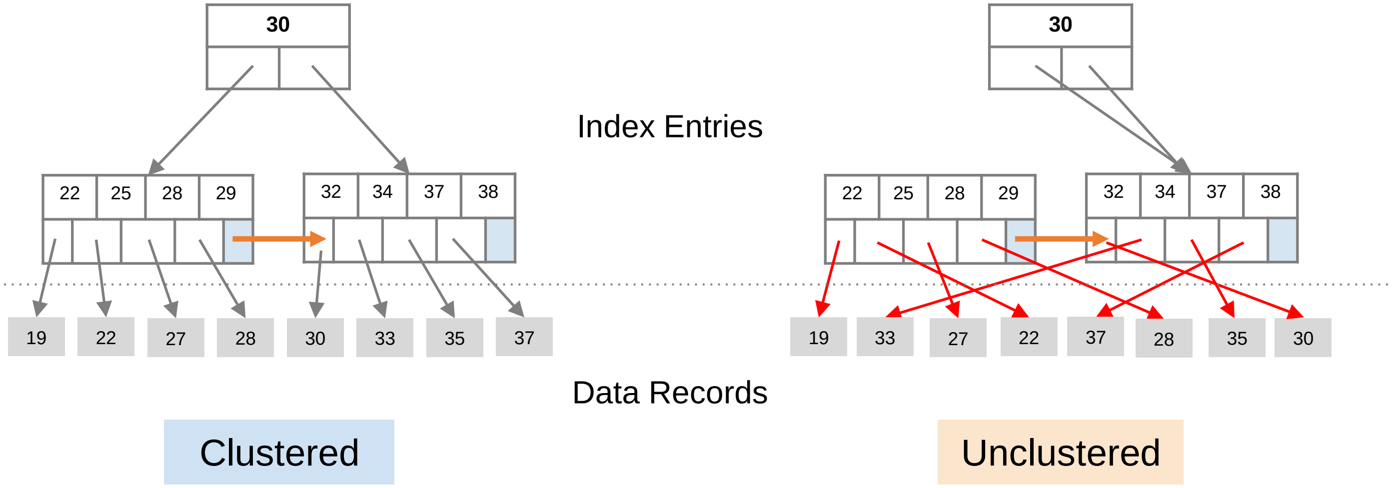
- We won’t go into specifics of B+ Tree insertion algorithm, but has several attractive qualities:
  - **~ Same cost as exact search**
  - ***Self-balancing:*** B+ Tree remains **balanced** (with respect to height) even after insert

B+ Trees also (relatively) fast for single insertions!
However, can become bottleneck if many insertions (if fill-factor slack is used up…)

Clustered Indexes

An index is ***clustered*** if the underlying data is ordered in the same way as the index's data entries.

Clustered vs. Unclustered Index



Clustered vs. Unclustered Index

- Recall that for a disk with block access, **sequential IO is much faster than random IO**
- For exact search, no difference between clustered / unclustered
- For range search over R values: difference between **1 random IO + R sequential IO**, and **R random IO**:
  - A random IO costs ~ 10ms (sequential much much faster)
  - For R = 100,000 records- **difference between ~10ms and ~17min!**

Summary

- We covered an algorithm + some optimizations for sorting larger-than-memory files efficiently
  - An ***IO aware*** algorithm!

- We create **indexes** over tables in order to support ***fast (exact and range) search*** and ***insertion*** over ***multiple search keys***

- **B+ Trees** are one index data structure which support very fast exact and range search & insertion via ***high fanout***
  - ***Clustered vs. unclustered*** makes a big difference for range queries too

THANK YOU!