Lecture 13: B+ Tree

Announcements

1. Project Part 2 extension till Friday

2. Project Part 3: B+ Tree coming out Friday

3. Poll for Nov 22nd

4. Exam Pickup: If you have questions, just want to see your exam come to office hours or drop by my office
    - Two weeks (until November 8th) for questions & concerns.

Lecture 13: B+ Tree

What you will learn about in this section

1. Recap: Indexing

2. B+ Trees: Basics

3. B+ Trees: Operations, Design & Cost

1. Recap: Indexing

Indexes: High-level

- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Search key properties
    - Any subset of fields
    - is **not** the same as *key of a relation*

- *Example:*

`Product(name, maker, price)`

On which attributes would you build indexes?

More precisely

- An *index* is a **data structure** mapping search keys to sets of rows in a database table
    - Provides efficient lookup & retrieval by search key value- usually much faster than searching through all the rows of the database table
- An index can store the full rows it points to (*primary index*) or pointers to those rows (*secondary index*)
    - We'll mainly consider secondary indexes

Operations on an Index

- Search: Quickly find all records which meet some *condition on the search key attributes*
  - More sophisticated variants as well. Why?

- Insert / Remove entries
  - Bulk Load / Delete. Why?

Indexing is one the most important features provided by a database for performance

[Activity-13.ipynb]

2. B+ Trees: Basics

What you will learn about in this section

1. B+ Trees: Basics

2. B+ Trees: Design & Cost

3. Clustered Indexes

B+ Trees

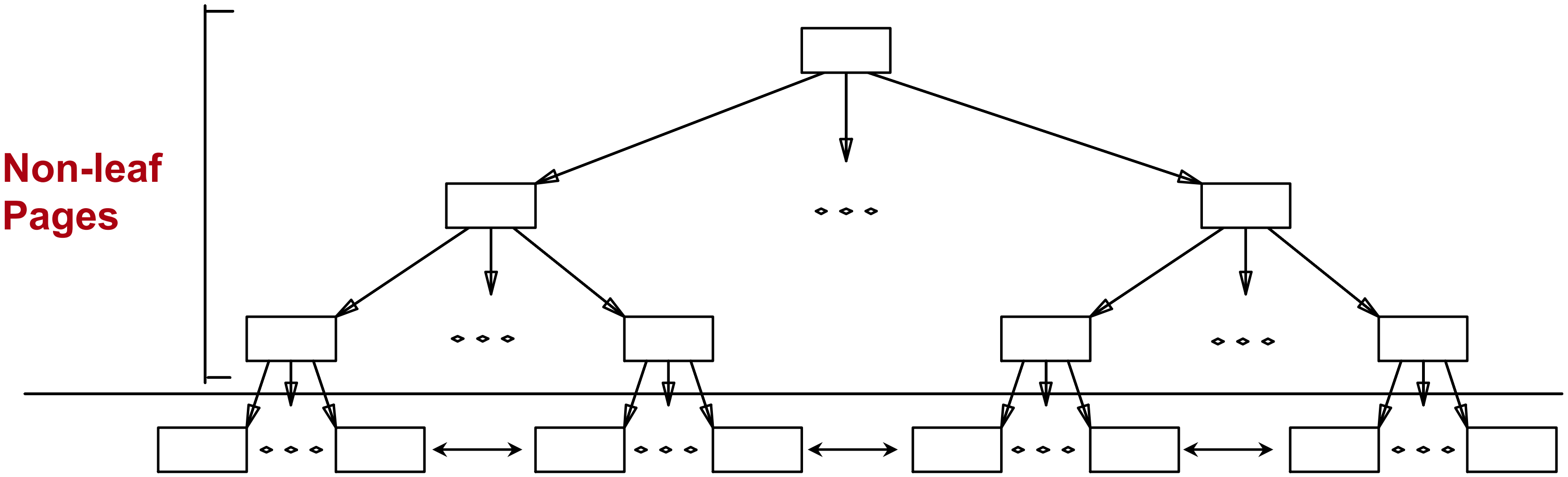
- Search trees
  - B does not mean binary!

- 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 Index

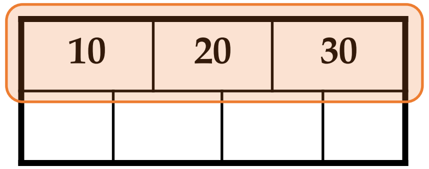
Non-leaf Pages



Leaf Pages (sorted by search key)

- Leaf pages contain data entries, and are chained (prev & next)
- Non-leaf pages have data entries

B+ Tree Basics



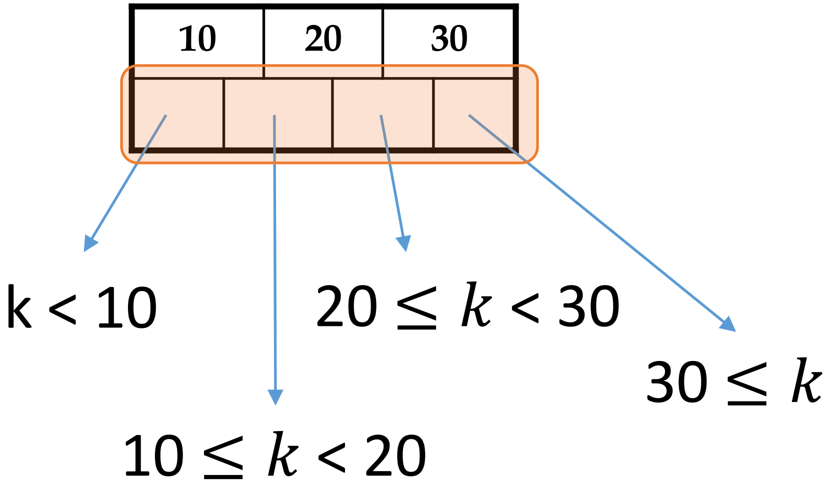
Parameter ***d*** = the order

Each *non-leaf (“interior”)* ***node*** has d $\leq m \leq$ 2d ***entries***

- ***Minimum 50% occupancy***

Root ***node*** has 1 $\leq m \leq$ 2d ***entries***

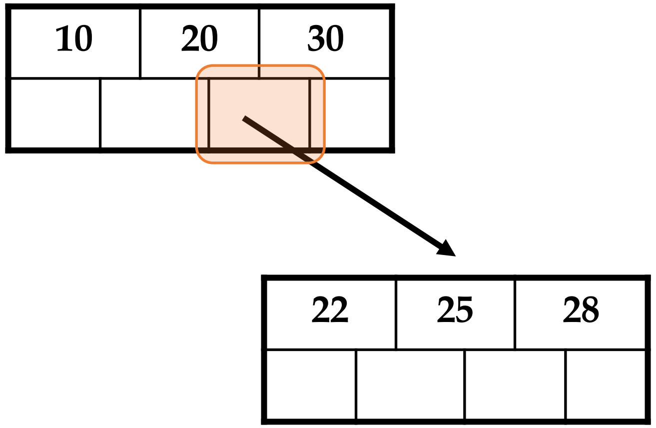
B+ Tree Basics



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

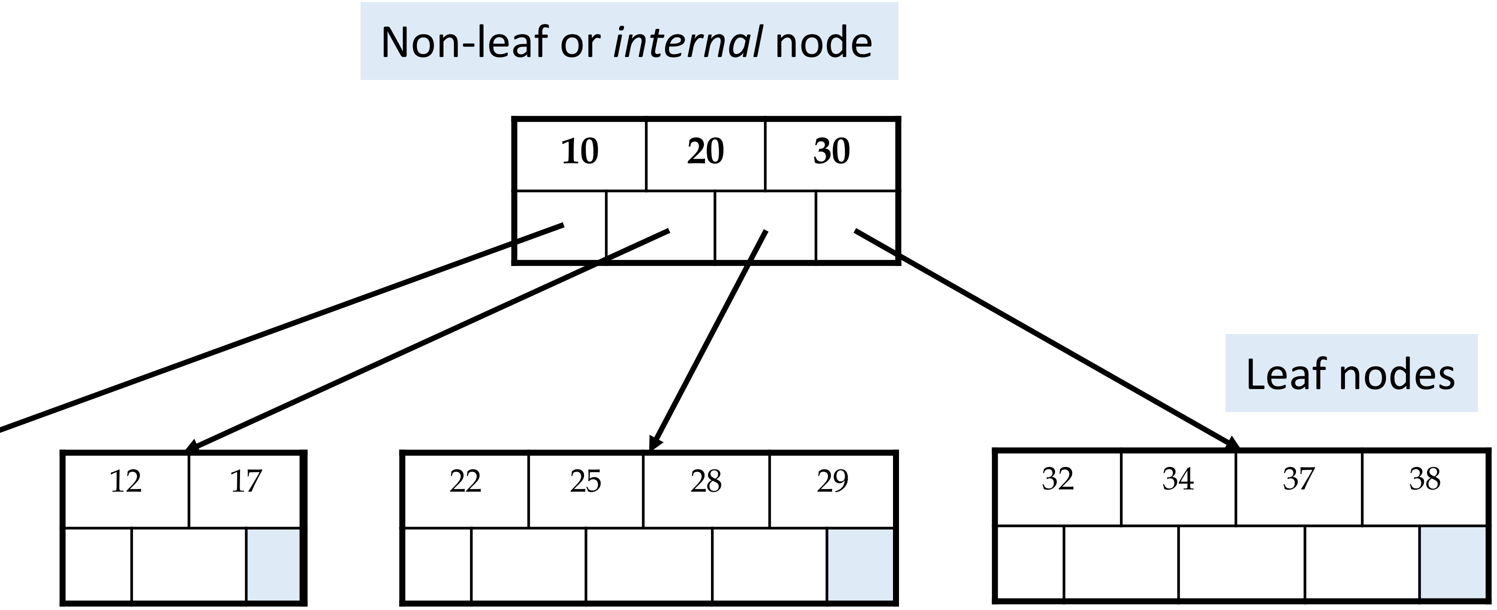
B+ Tree Basics

Non-leaf or *internal* node



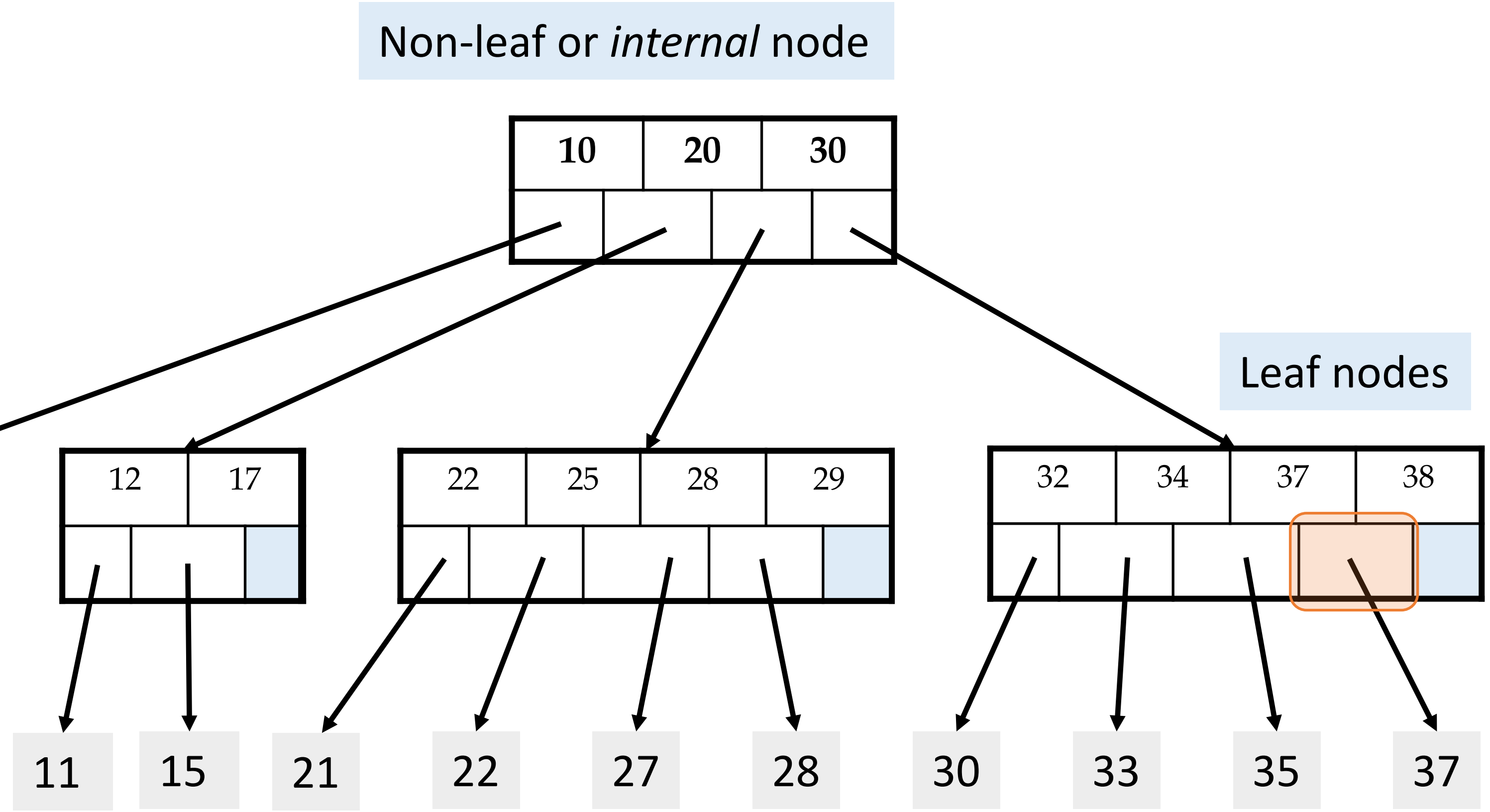
For each range, in a *non-leaf* node, there is a **pointer** to another node with entries in that range

B+ Tree Basics



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

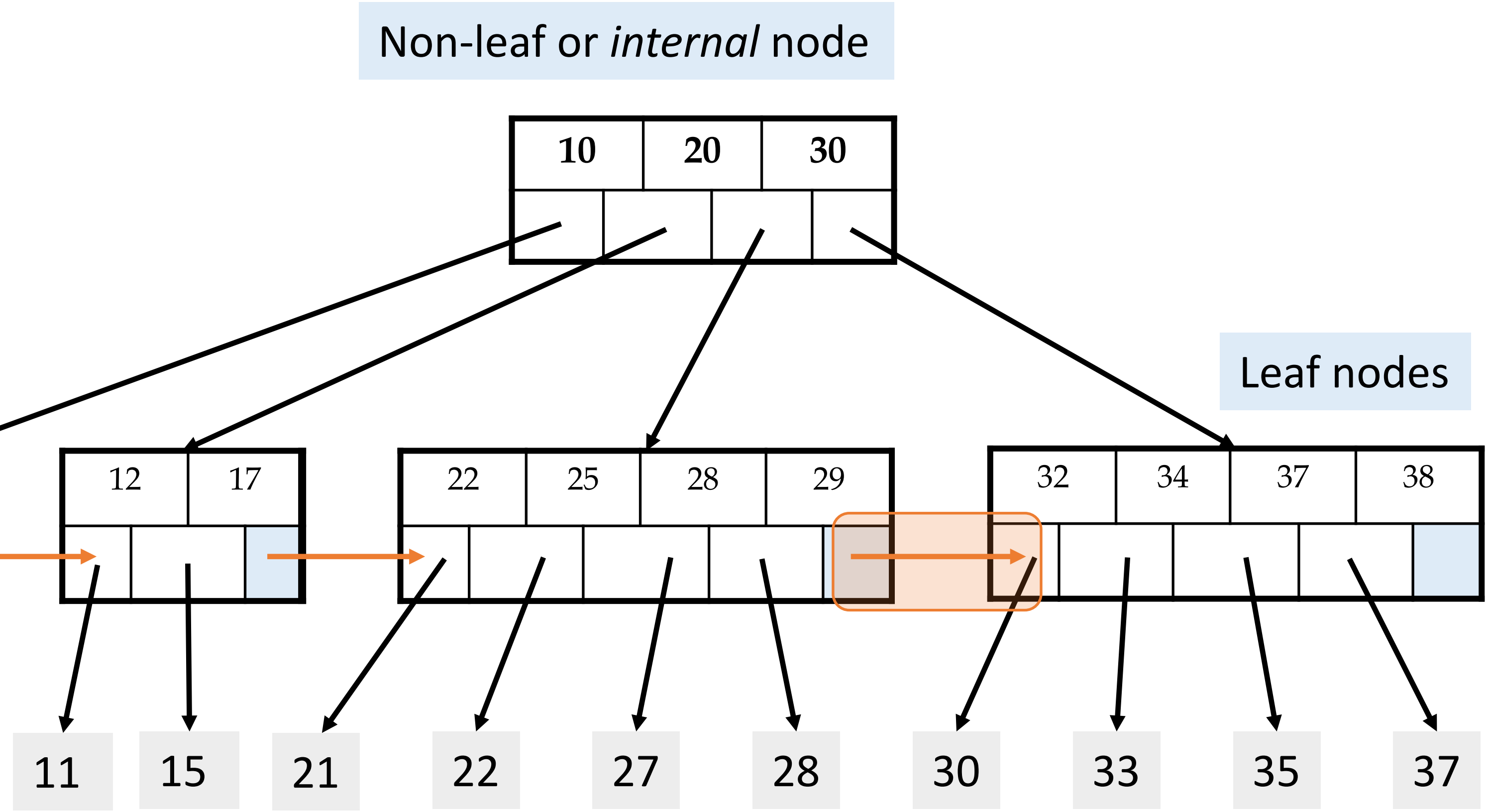
B+ Tree Basics



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

Their entry slots contain pointers to data records

B+ Tree Basics

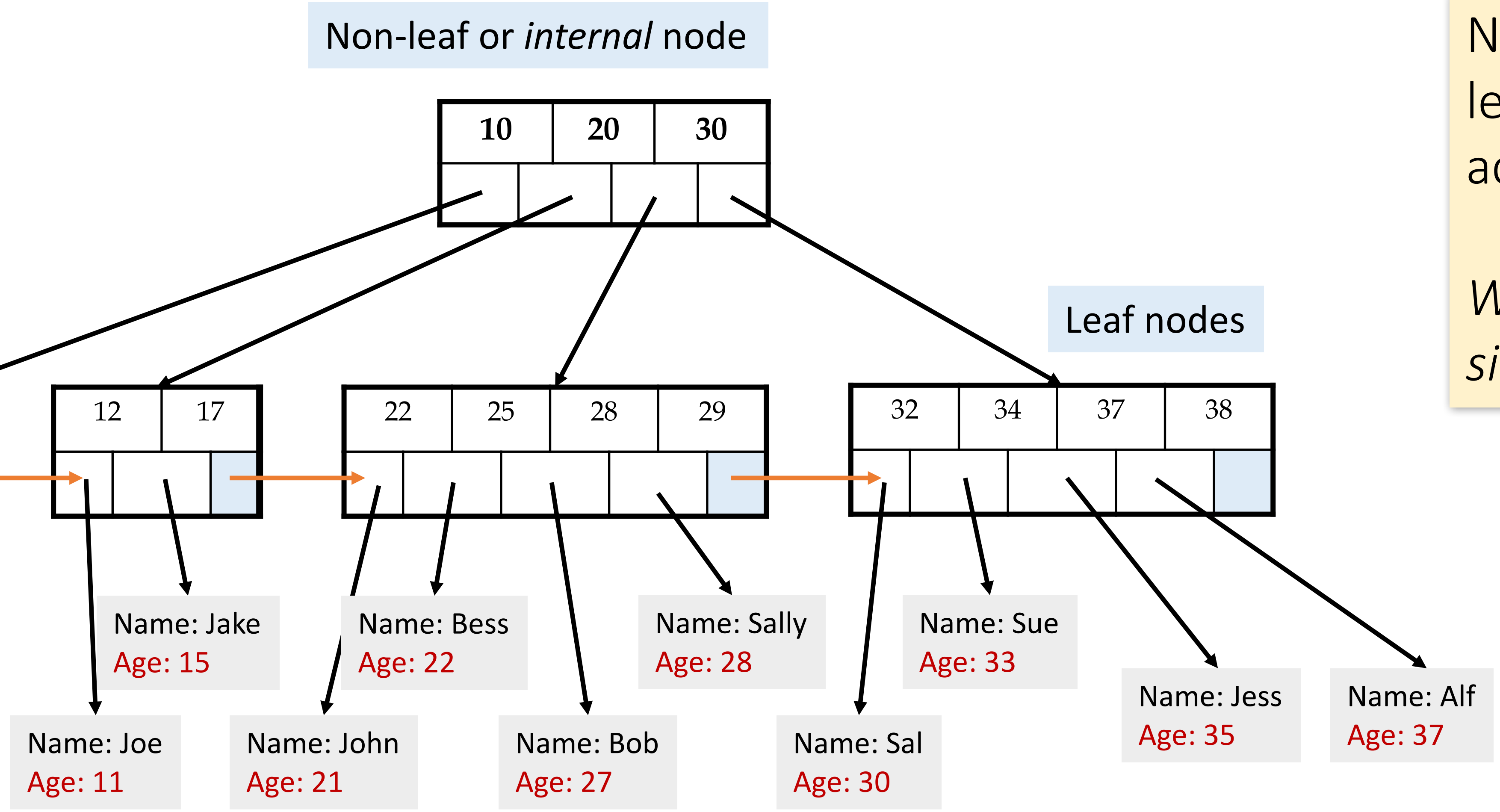


Leaf nodes also have between *d* and *2d* entries, and are different in that:

Their entry slots contain pointers to data records

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

B+ Tree Basics

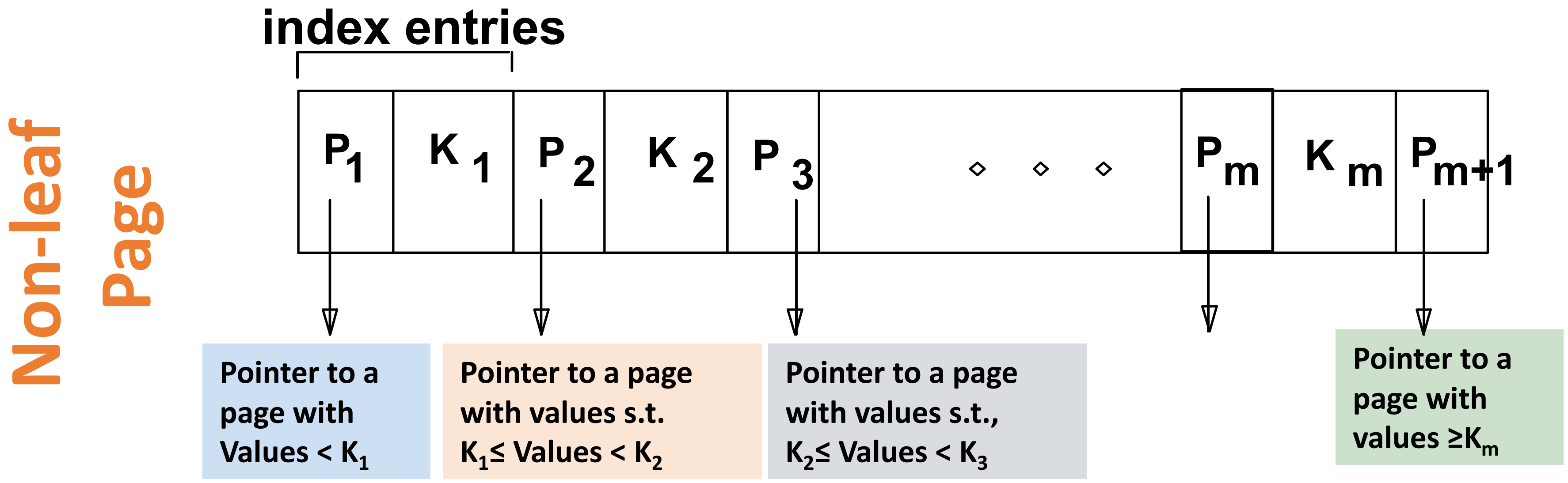


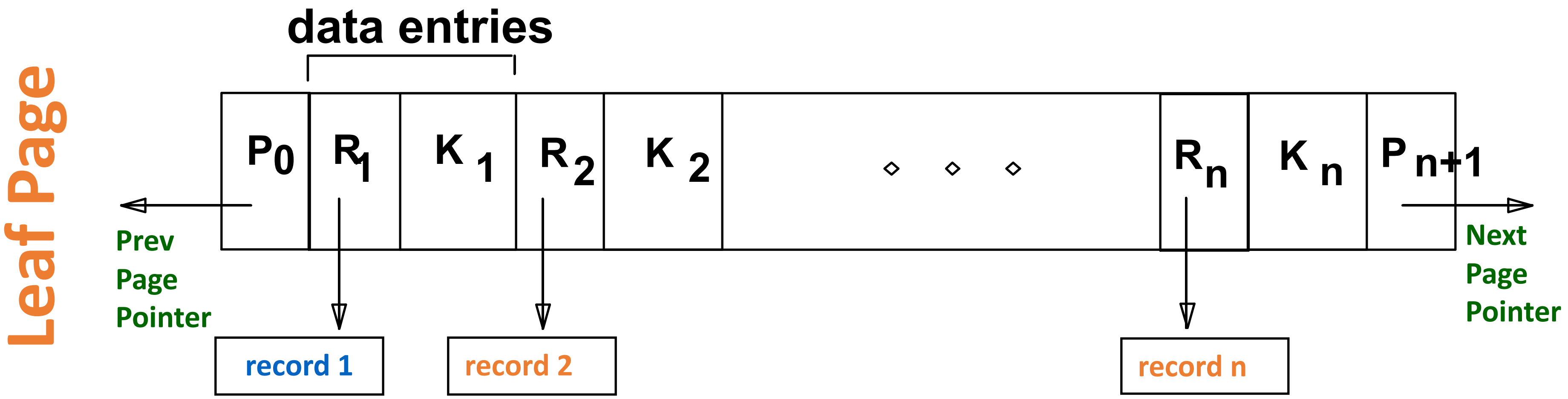
Note that the pointers at the leaf level will be to the actual data records (rows).

We might truncate these for simpler display (as before)…

Height = 1

B+ Tree Page Format





3. B+ Trees: Operations, Design & Cost

B+ Tree operations

A B+ tree supports the following operations:

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

Searching a B+ Tree

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

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

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

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

B+ Tree: Search

- start from root
- examine index entries in non-leaf nodes to find the correct child
- traverse down the tree until a leaf node is reached
- non-leaf nodes can be searched using a binary or a linear search

B+ Tree Exact Search Animation

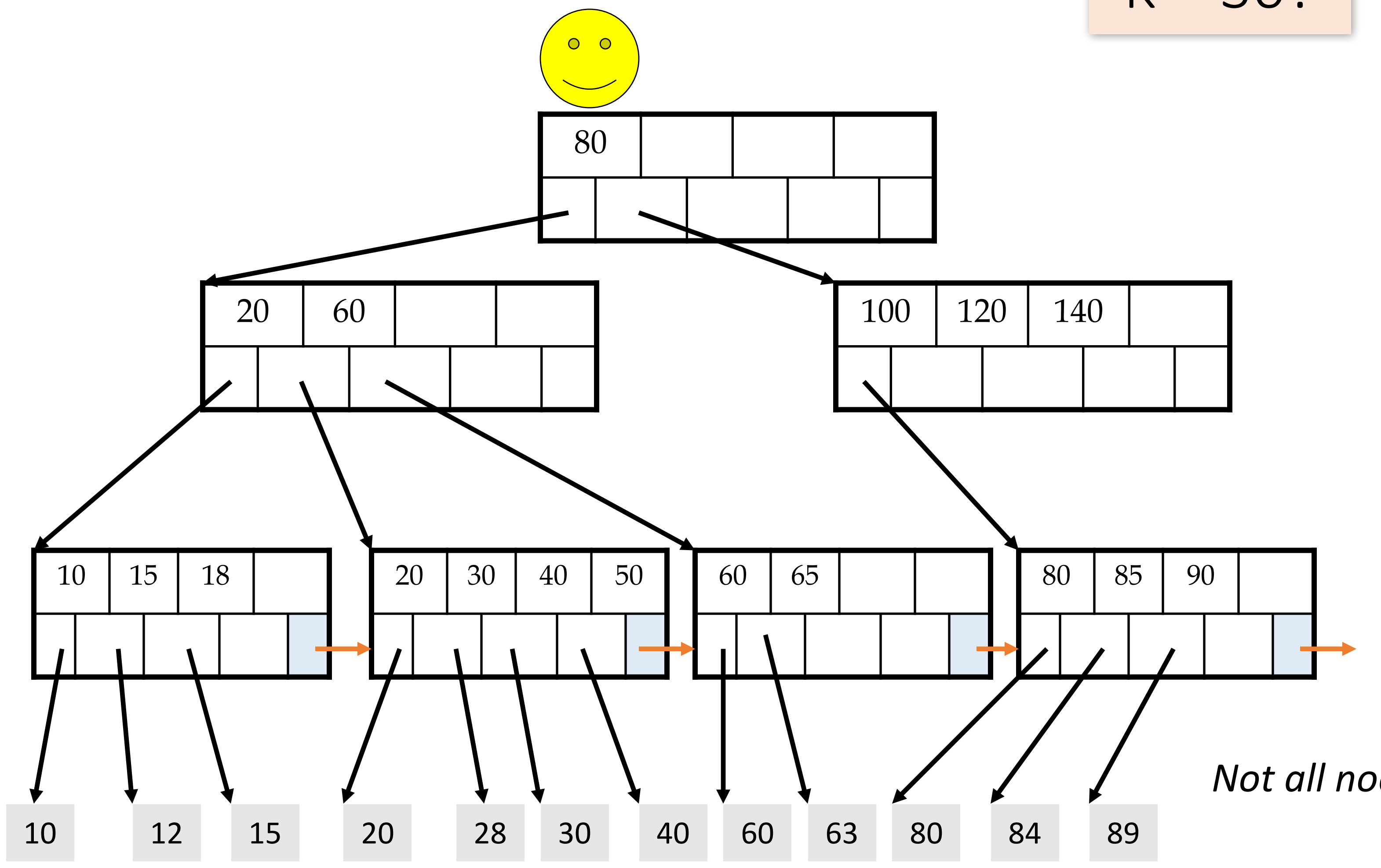
K = 30?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



Not all nodes pictured

B+ Tree Range Search Animation

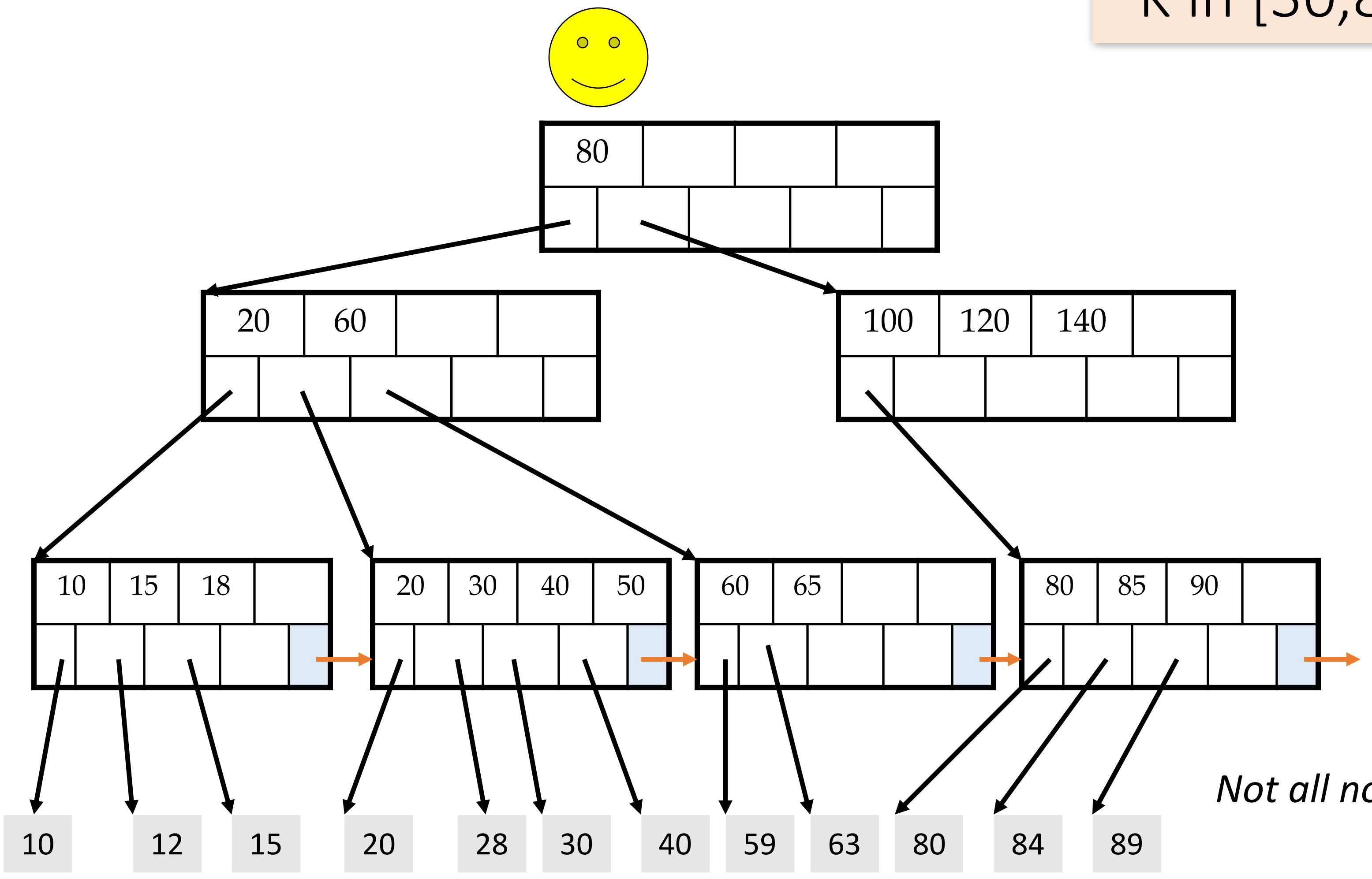
K in [30,85]?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!

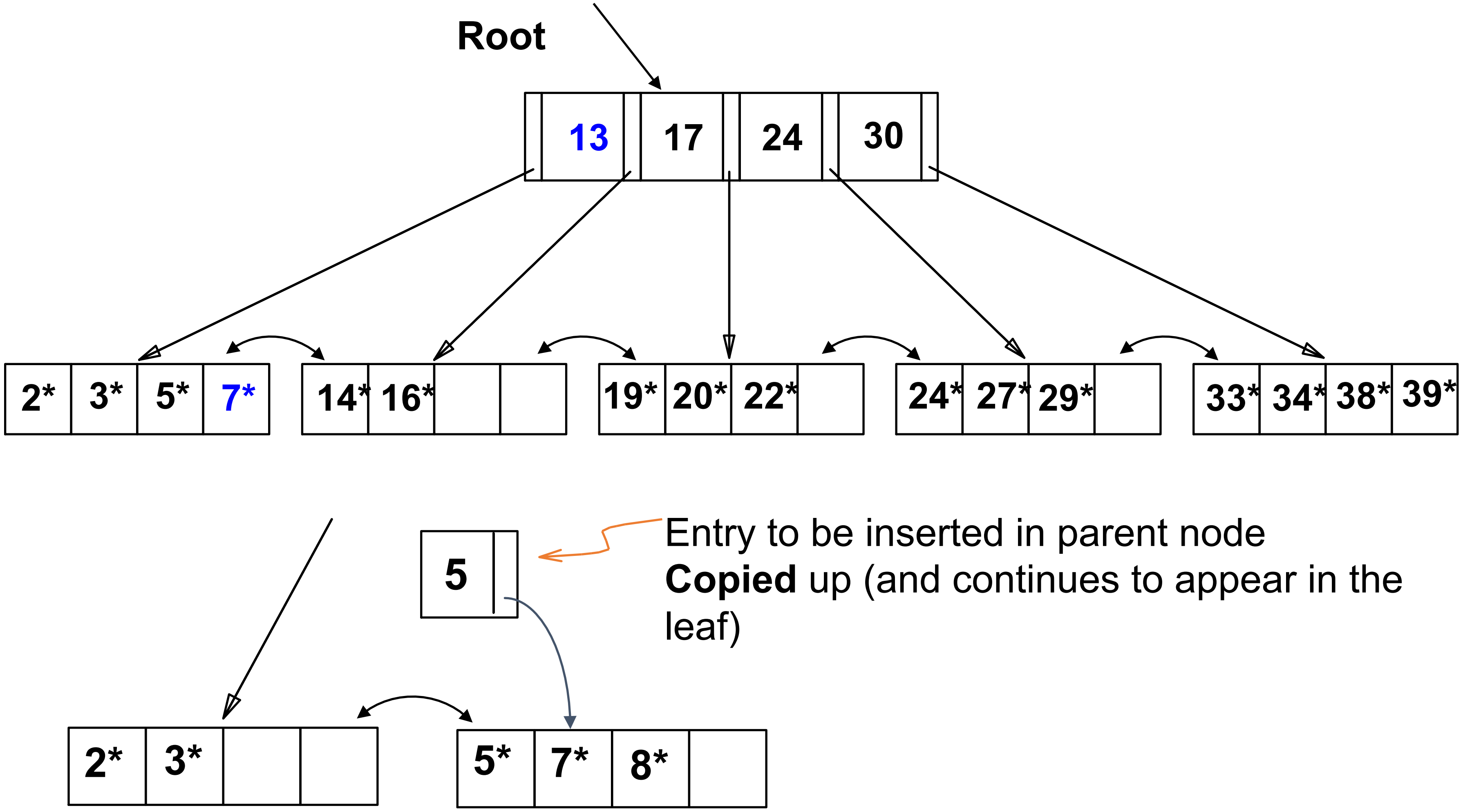


Not all nodes pictured

B+ Tree: Insert

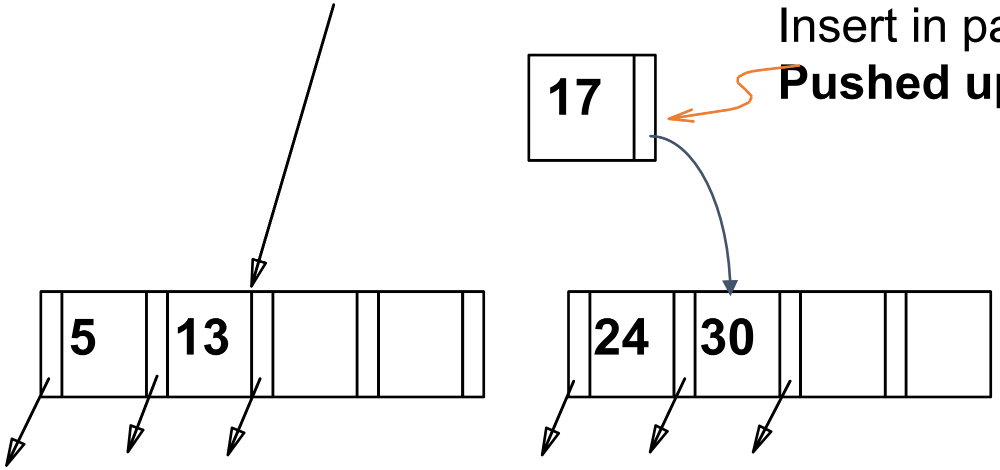
- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must ***split*** *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split non-leaf node, redistribute entries evenly, but **pushing up** the middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Inserting 8* into B+ Tree



Entry to be inserted in parent node
Copied up (and continues to appear in the leaf)

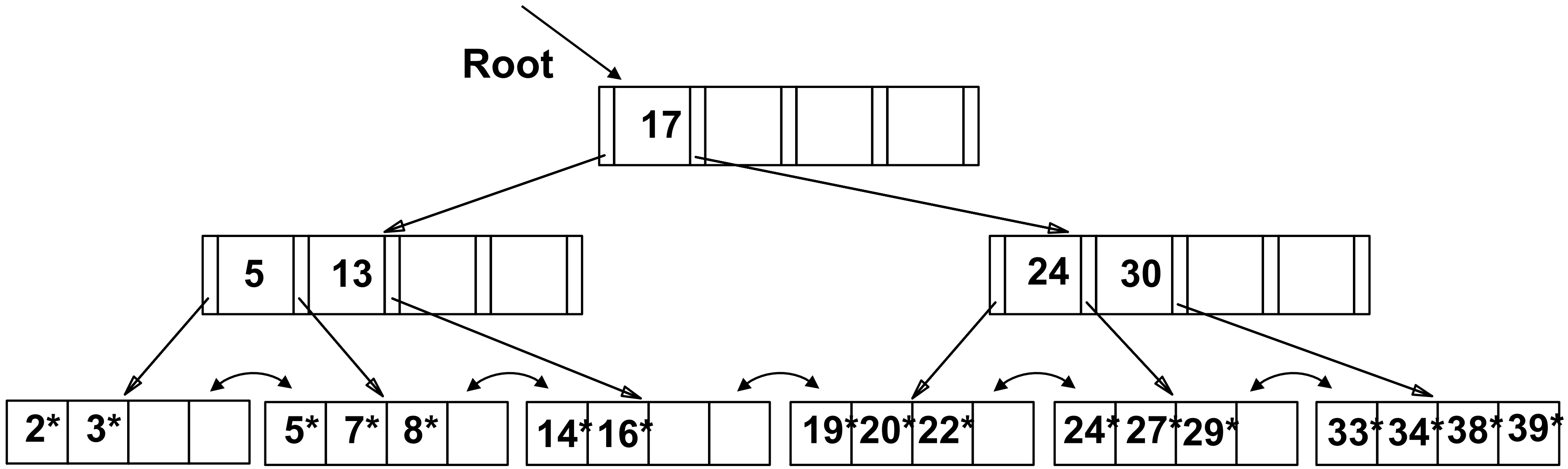
Inserting 8* into B+ Tree



Insert in parent node.
Pushed up (and only appears once in the index)

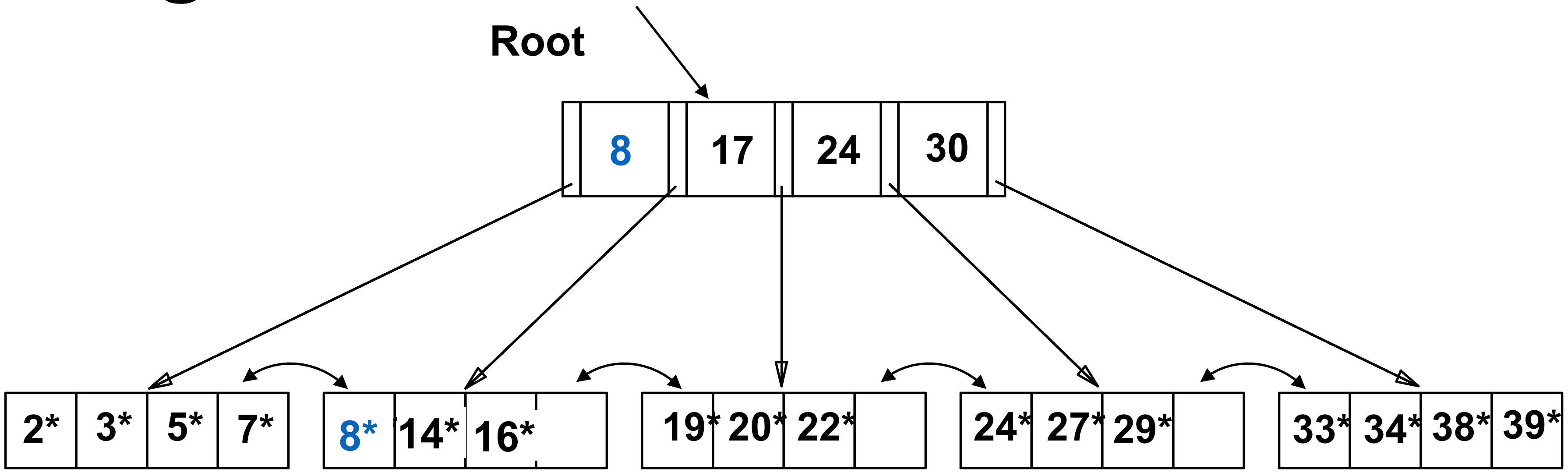
Minimum occupancy is guaranteed in both leaf and index page splits

Inserting 8* into B+ Tree



- Root was split: height increases by 1
- Could avoid split by re-distributing entries with a sibling
  - Sibling: immediately to left or right, and same parent

Inserting 8* into B+ Tree



- Re-distributing entries with a **sibling**
  - Improves page occupancy
  - Usually not used for non-leaf node splits. Why?
    - Increases I/O, especially if we check both siblings
    - Better if split propagates up the tree (rare)
    - Use only for leaf level entries as we have to set pointers

Fast Insertions & Self-Balancing

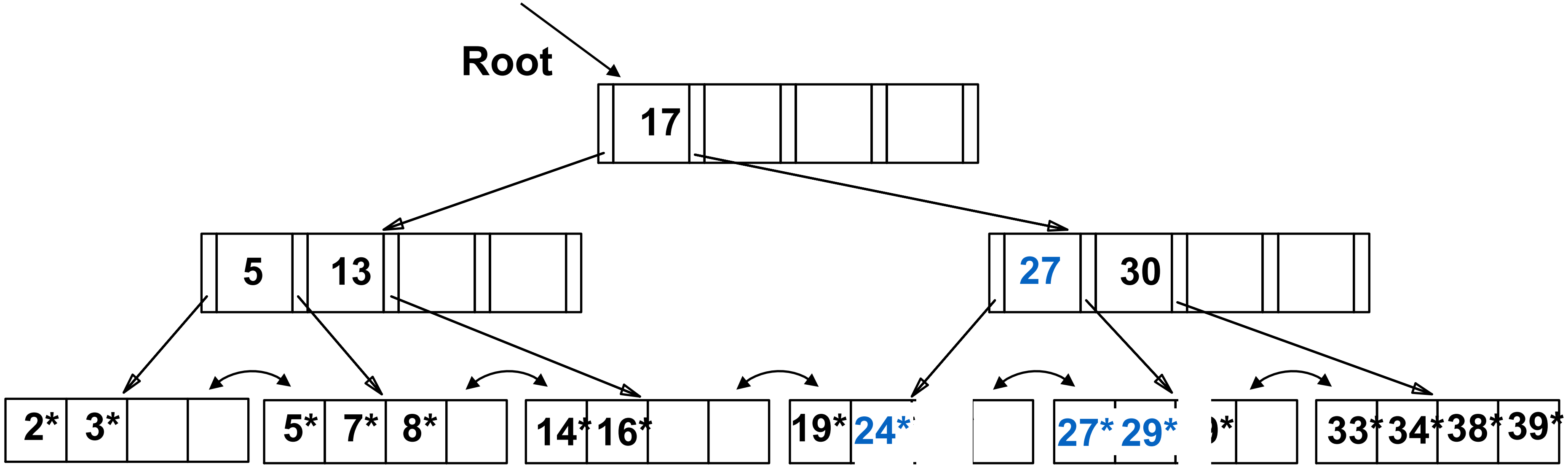
- The B+ Tree insertion algorithm has several attractive qualities:
  - **~ Same cost as exact search**
  - ***Self-balancing:*** B+ Tree remains **balanced** (with respect to height) even after insert

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

B+ Tree: Deleting a data entry

- Start at root, find leaf *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 (adjacent node with same parent as L)*.
    - If re-distribution fails, ***merge*** *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could **propagate** to root, decreasing height.

Deleting 22* and 20*

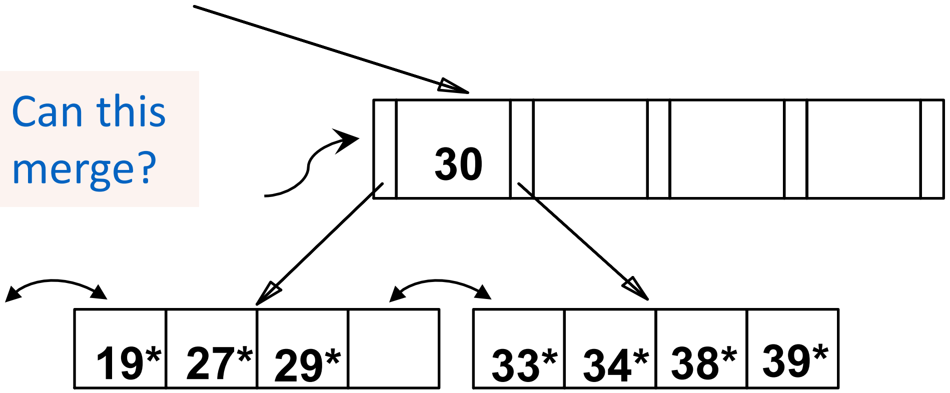


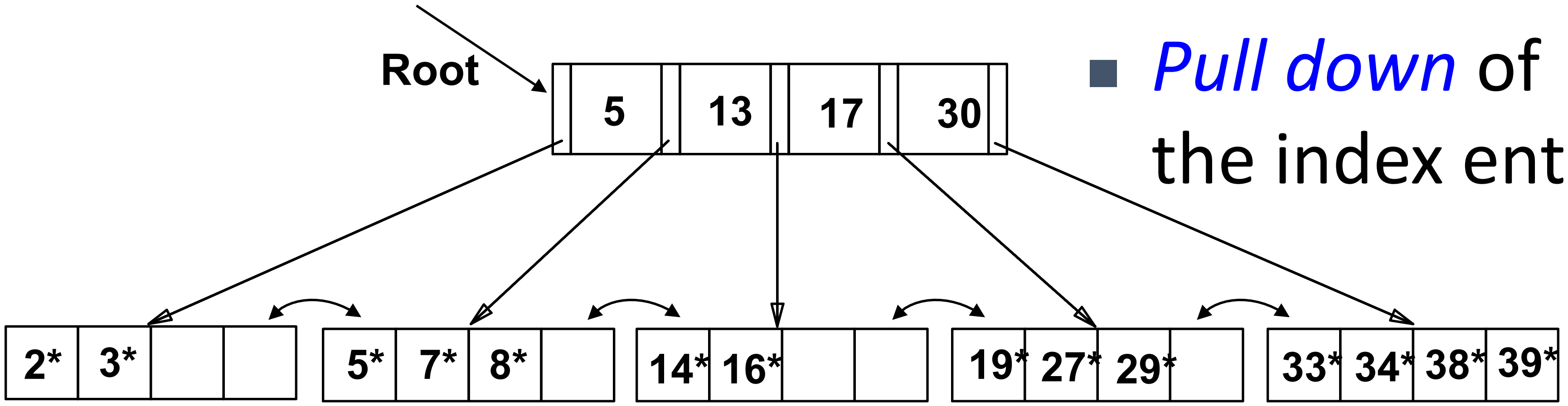
- Deleting 22* is easy.
- Deleting 20* is done with re-distribution. Notice how the middle key is **copied up**.

… And then deleting 24*

- Must merge.
- In the non-leaf node, ***toss*** the index entry with key value = 27

Can this merge?

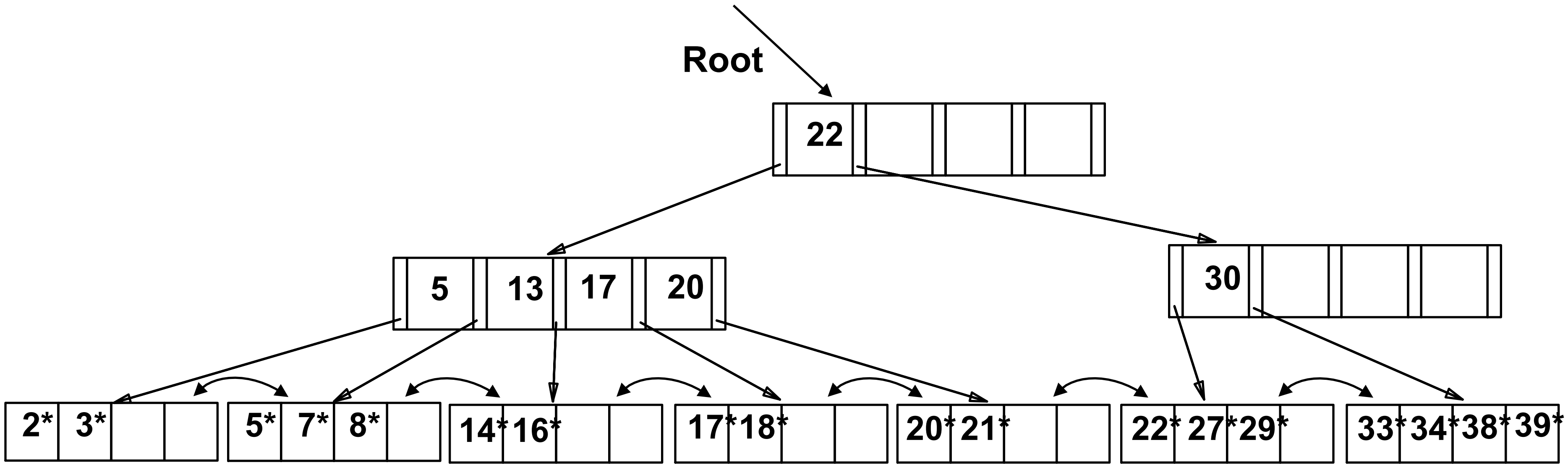




- *Pull down* of the index entry

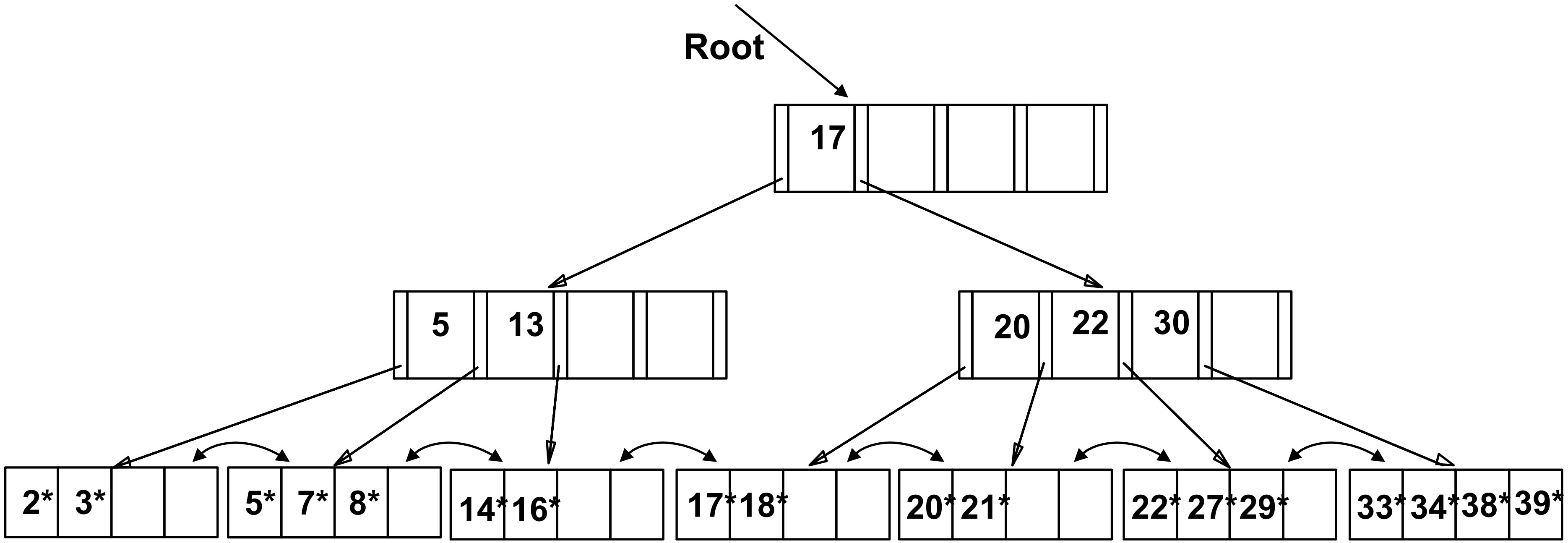
Non-leaf Re-distribution

- Tree *during deletion* of 24*.
- Can re-distribute entry from left child of root to right child.



After Re-distribution

- Rotate through the parent node
- It suffices to re-distribute index entry with key 20; For illustration 17 also re-distributed



B+ Tree deletion

- Try redistribution with **all** siblings first, then merge. Why?
  - Good chance that redistribution is possible (large fanout!)
  - Only need to propagate changes to parent node
  - Files typically grow not shrink!

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
  - Use RID to get a **unique** (composite) key!
- Use list of rids instead of a single rid in the leaf level
  - Single data entry could still span multiple pages

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 x 4 + (2d+1) x 8 <= 4096 → ***d <= 170***

NB: Oracle allows 64K = 2^16 byte blocks
→ d <= 2730

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

The known universe contains ~10^{80} particles… what is the height of a B+ Tree that indexes these?

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

<u>Fill-factor</u> 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)"

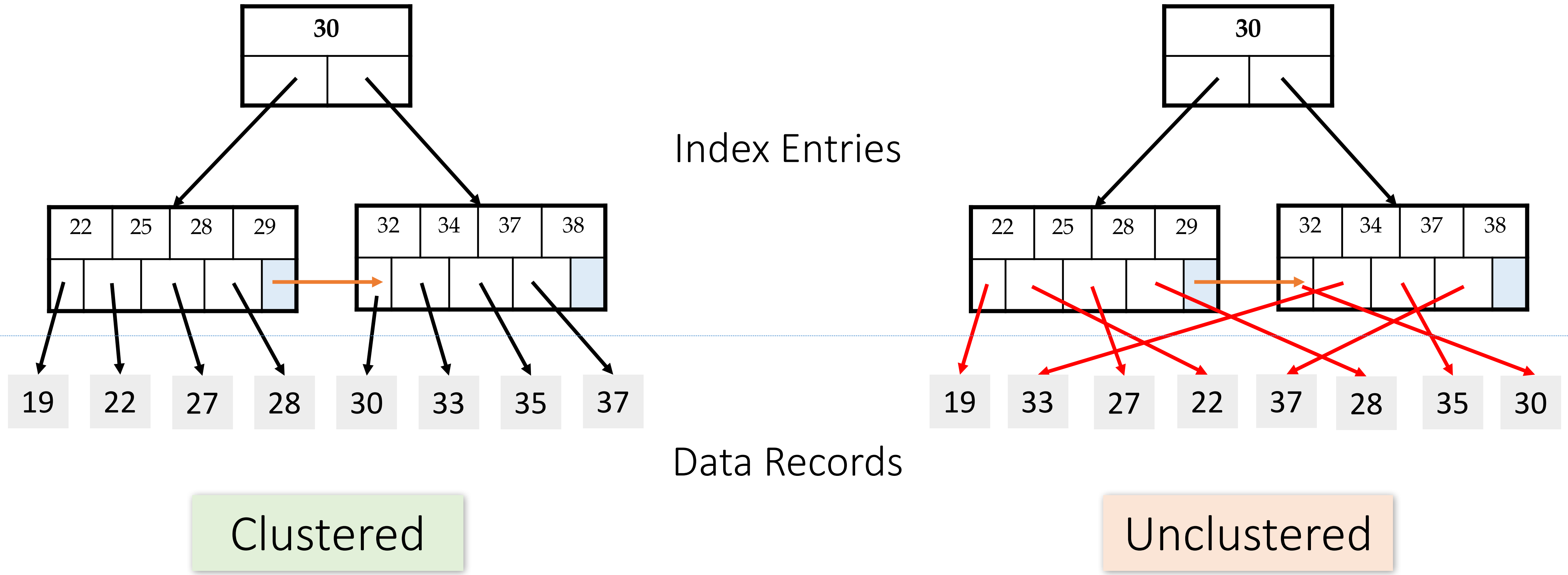
IO Cost: $\left\lceil \log_f \frac{N}{F} \right\rceil - L_B + Cost(OUT)$

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

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



Index Entries

Data Records

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 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