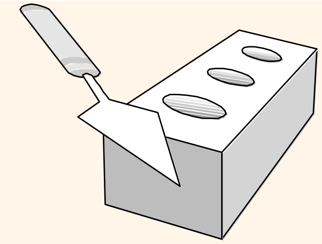
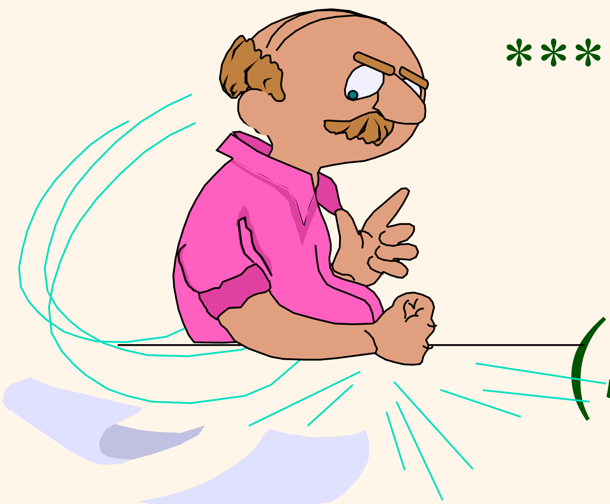


Introduction to Data Management

**** The "Flipped" Edition ****



Lecture #19
(Storage & Indexing II)

Instructor: Mike Carey
mjcarey@ics.uci.edu

Announcements

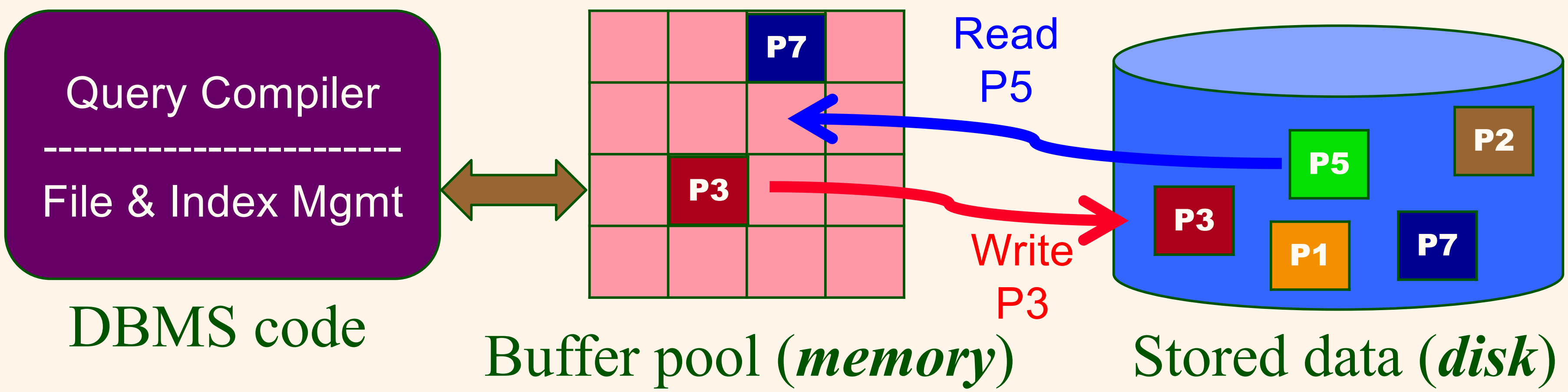
- Roadmap check:

Midterm Exam 2	***Mon, Nov 15*** (during lecture time)
Storage	Ch. 12.1-12.4, 12.6-12.7
Indexing	Ch. 14.1-14.4, 14.5
Physical DB Design	Ch. 14.6-14.7, 15.1-15.3, 15.5.3
Semistructured Data Management (*a.k.a.* NoSQL)	Ch. 8.1, AsterixDB SQL++ Primer, Couchbase SQL++ Book
Data Science 1: Advanced SQL Analytics	Ch. 5.5, 11.3
Data Science 2: Notebooks, Dataframes, and Python/Pandas	Lecture notes and Jupyter notebook
Basics of Transactions	Ch. 4.3, Ch. 17
Endterm Exam	***Fri, Dec 3*** (during lecture time)

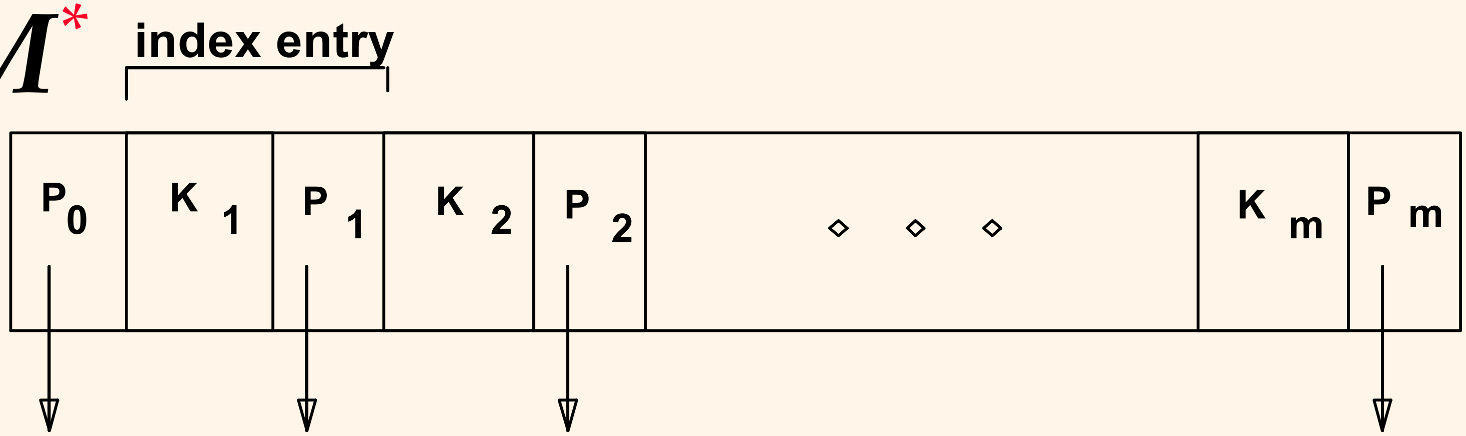
- SQL HW assignments
  - HW #6 (Advanced SQL) is underway!
  - HW #7 will come out after Midterm #2 (like before)
- Midterm #2 (relational languages) is *1 week away (!!)*
  - Same in-class Gradescope + cheat sheet plan as Midterm #1

So: Disks and Files (Reminder!)

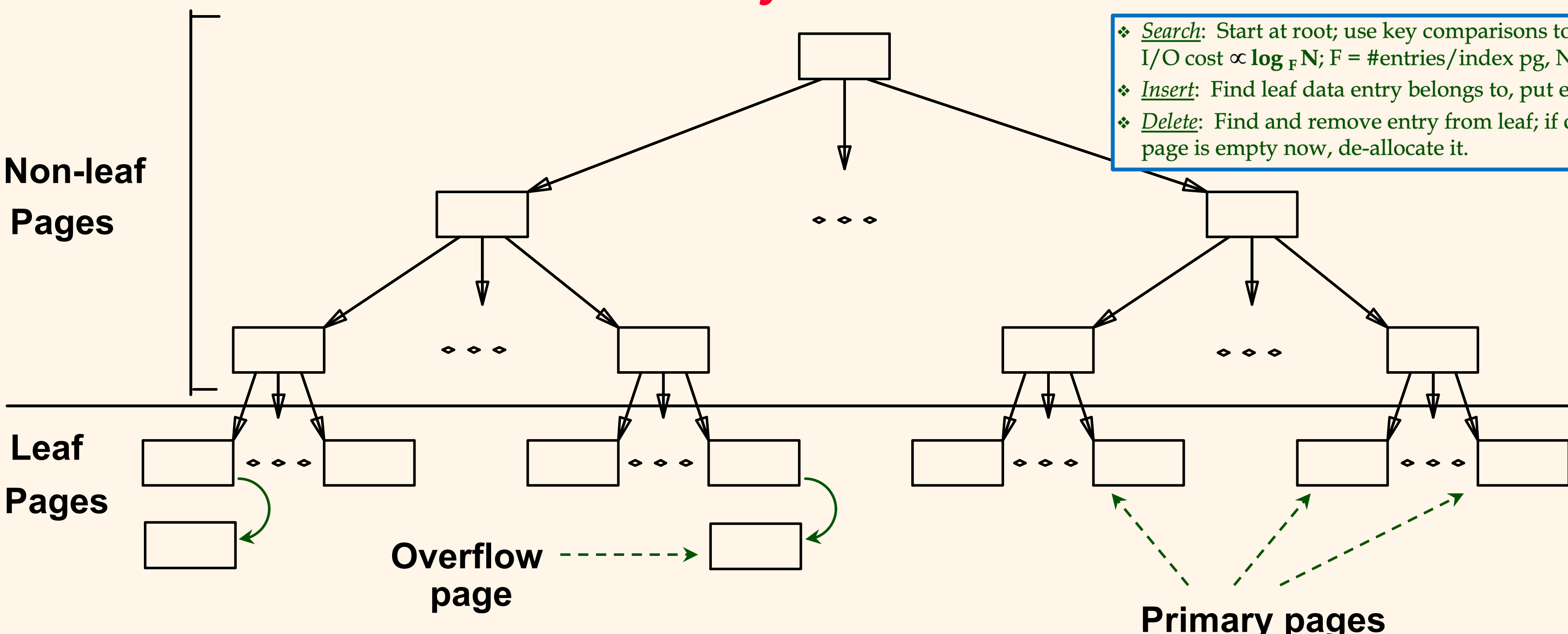
- DBMSs store persistent data on *secondary storage*.
- This has major implications for DBMS design!
  - READ: xfer data block from ***disk*** to ***memory*** (RAM).
  - WRITE: xfer data block from RAM to disk.
  - Both are ***high-cost*** operations, relative to in-memory operations, so must be considered carefully!



ISAM*



- Index file may still be quite large. But, we can apply the same idea **recursively** to address that!



- *Search*: Start at root; use key comparisons to go to leaf. I/O cost $\propto$ **log $_F$ N**; F = #entries/index pg, N = # leaf pgs
- *Insert*: Find leaf data entry belongs to, put entry there.
- *Delete*: Find and remove entry from leaf; if overflow page is empty now, de-allocate it.

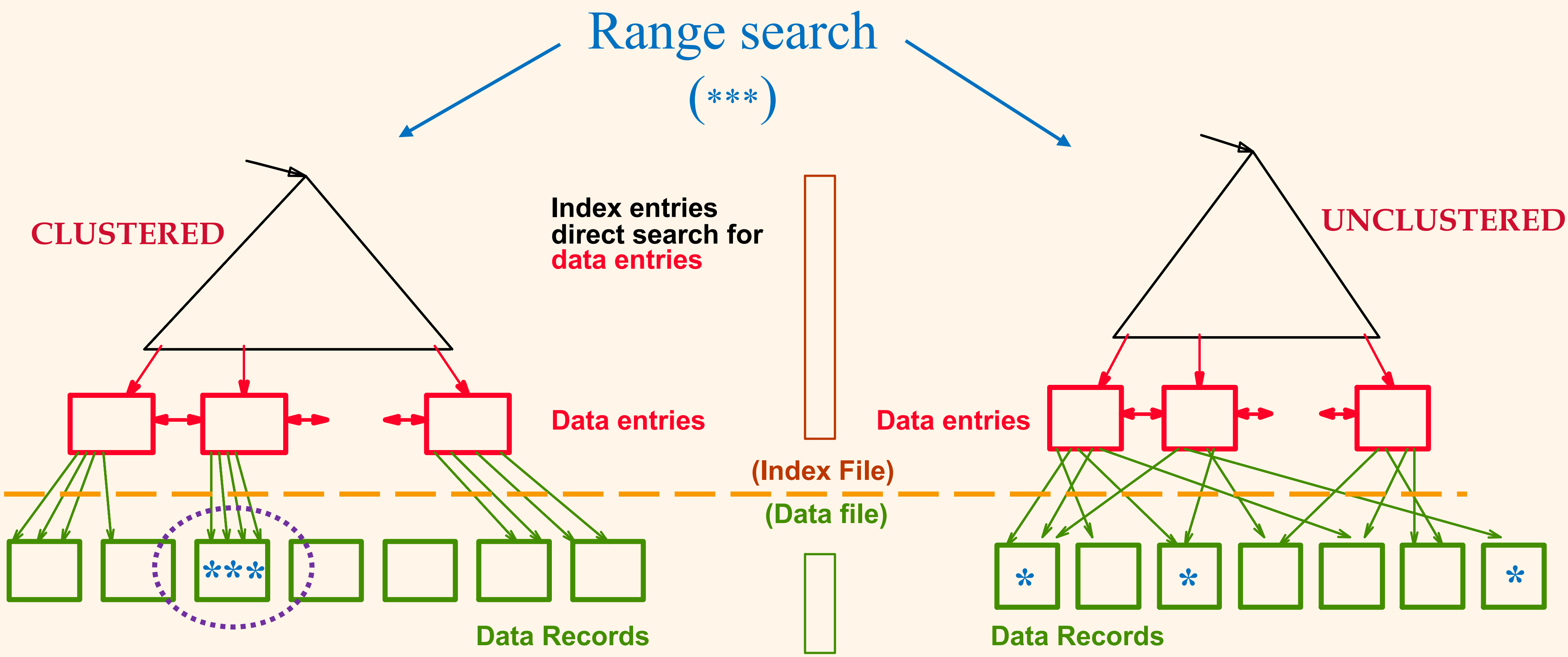
☛ *Leaf pages contain the data entries.*

**ISAM: Indexed Sequential Access Method*

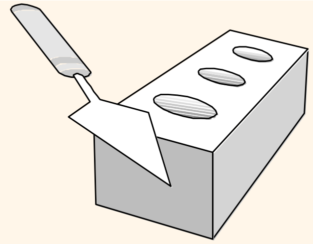
Index Classification

- *Primary* vs. *secondary*: If search key contains the primary key, it's called the *primary* index, else it's a *secondary* index.
  - *Unique* index: Search key contains a *candidate* key.
- *Clustered* vs. *unclustered*: If order of index entries is the same as, or "close to", the order of stored data records, then it's called a *clustered* index.
  - A table can be clustered on *at most one* search key (see RID ordering in example from last lecture).
  - Cost of retrieving data records via an index varies *greatly* based on whether index is clustered or not!

Clustered vs. Unclustered Indexes



Note: Clustering matters for SSD, even more so for HDD.



Indexing in PostgreSQL

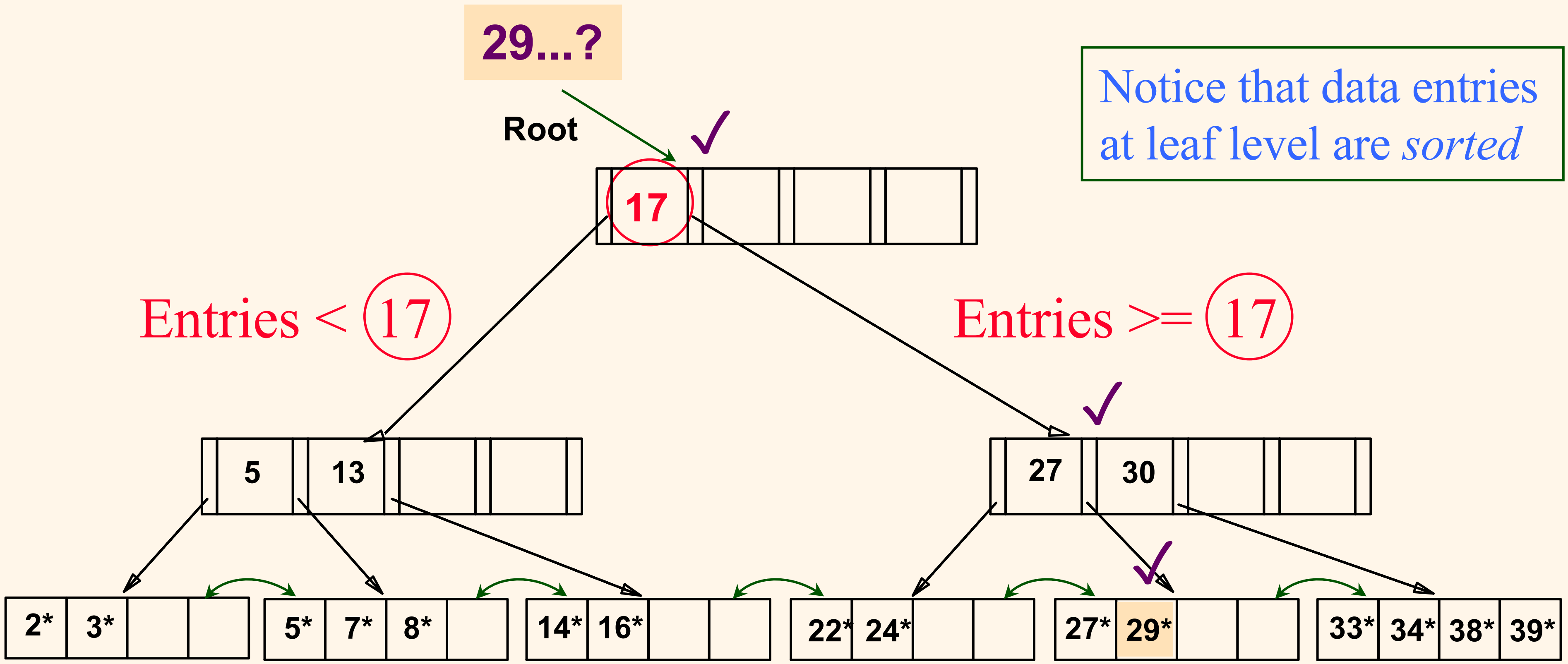
Choices are `btree`, `hash`, `gist`, and `gin`. The default method is `btree`.

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table [ USING method ]
    ( { column | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace ]
    [ WHERE predicate ]
```

***Ex*:** **CREATE UNIQUE INDEX** title_idx **ON** films (title);

https://www.postgresql.org/docs/14/sql-createindex.html)

Next Up: ***B+ Trees***



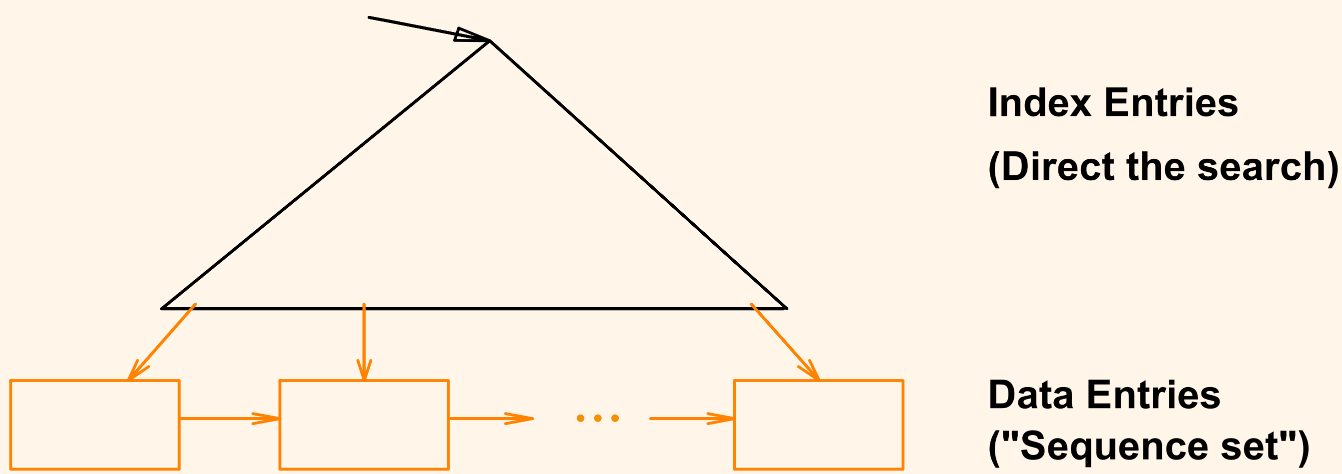
Notice that data entries at leaf level are *sorted*

$k^* = (k, \mathrm{I}(k))$, *e.g.*, (29, *RID(s)*)

Note: Just *3 page reads* to get from root to (*any*) leaf here!

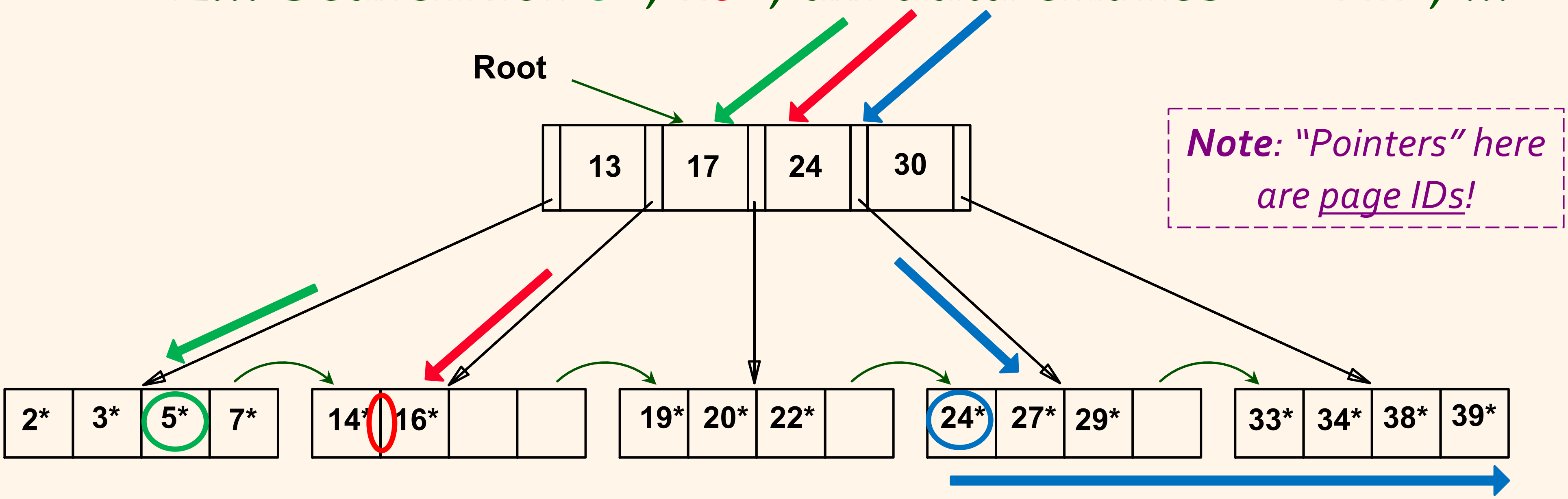
B+ Tree: *Most Widely Used Index!*

- Insert/delete at log $_F$ N cost; keep tree ***height-balanced***. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains **d** <= $\underline{m}$ <= 2**d** entries. The (mythical) **d** is called the *order* of the B+ tree.
- Supports equality and range-searches efficiently.



Example B+ Tree

- Search begins at root, and key comparisons direct the search to a leaf (as in ISAM).
- ***Ex:*** Search for 5*, 15*, all data entries >= 24*, ...



☛ *Based on the search for 15*, we know it is not in the tree.*

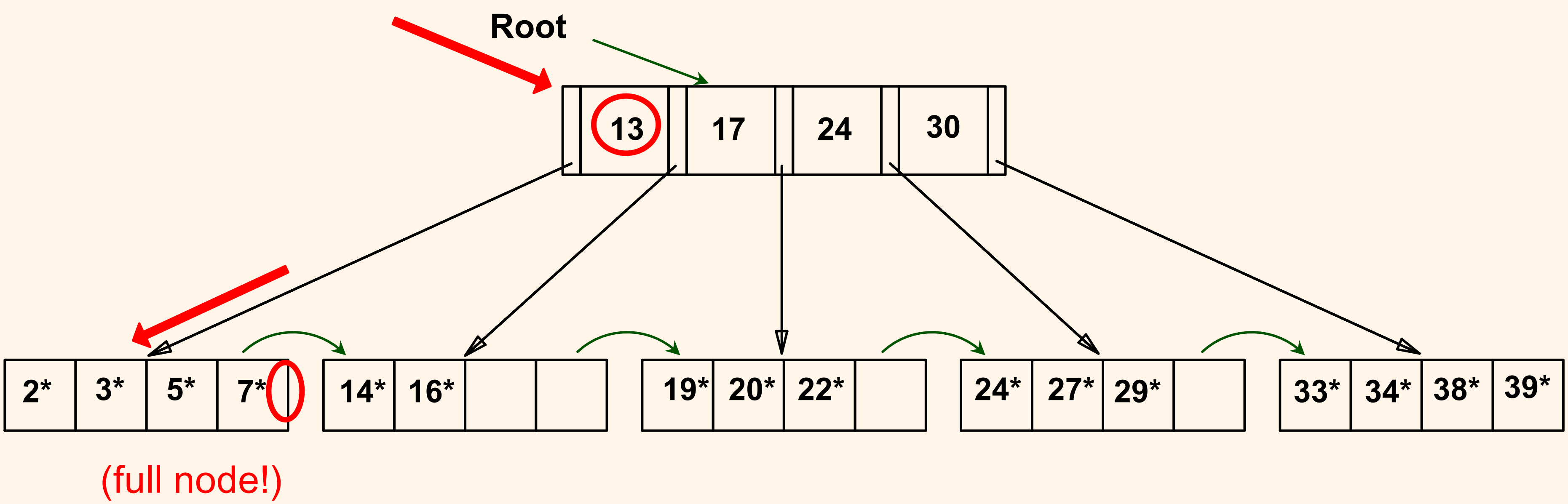
☛ *Range searches find the starting point, then scan across the leaves.*

Inserting a Data Entry into a B+ Tree

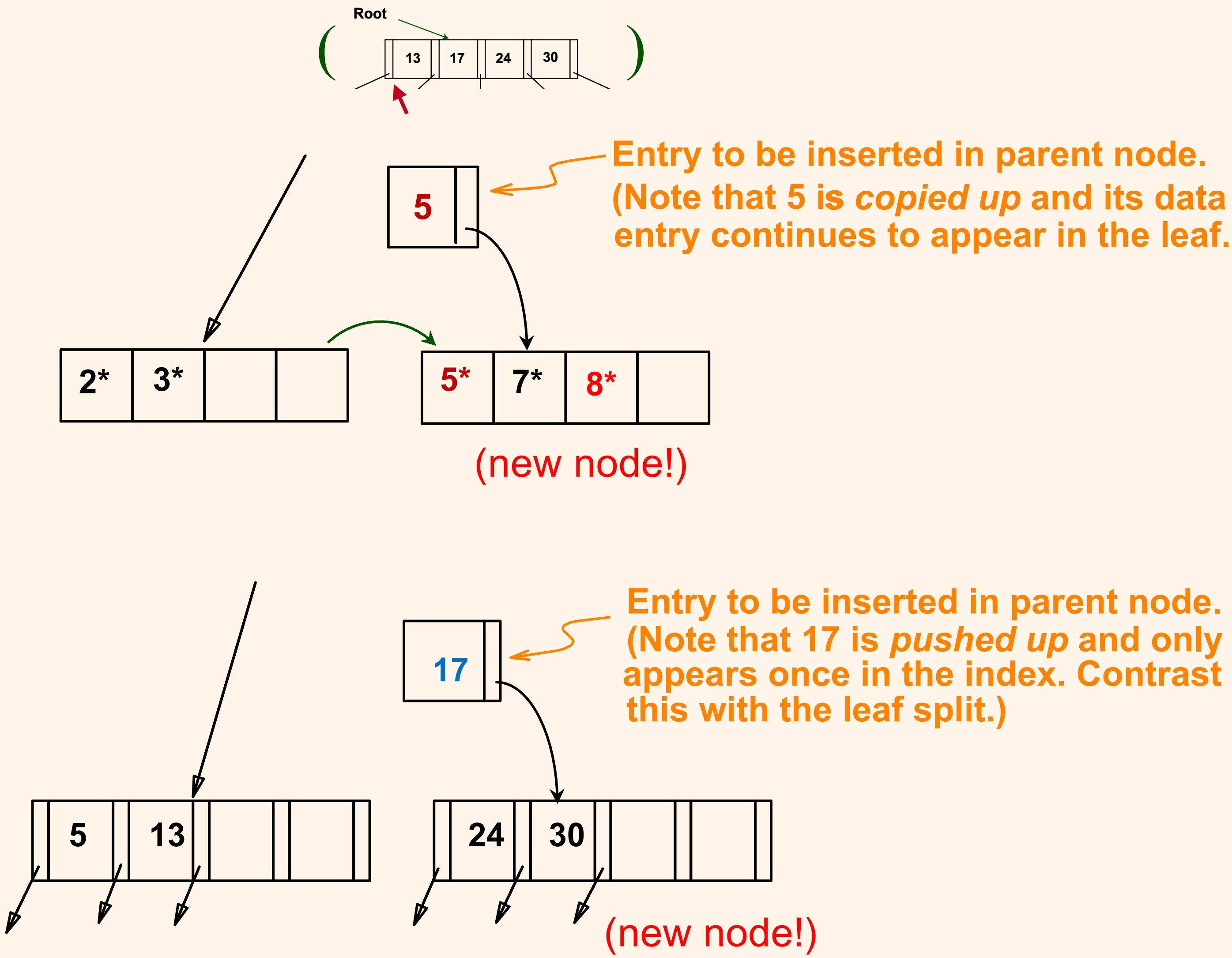
- Find correct leaf *L* (by ***searching*** for the new k).
- Put new data entry (k*, *a.k.a.* (k, I(k)) in leaf *L*.
  - If *L* has enough space, *done*! (Most likely case!)
  - Else, must *split* leaf *L* *(into L and a new leaf node L2)*
    - Redistribute entries "evenly" and copy up the middle key.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

*Example B+ Tree: Insert 8**

- Search begins at root, key comparisons direct the search to leaf (the insert's target page).
- ***Ex:*** Search for 8*

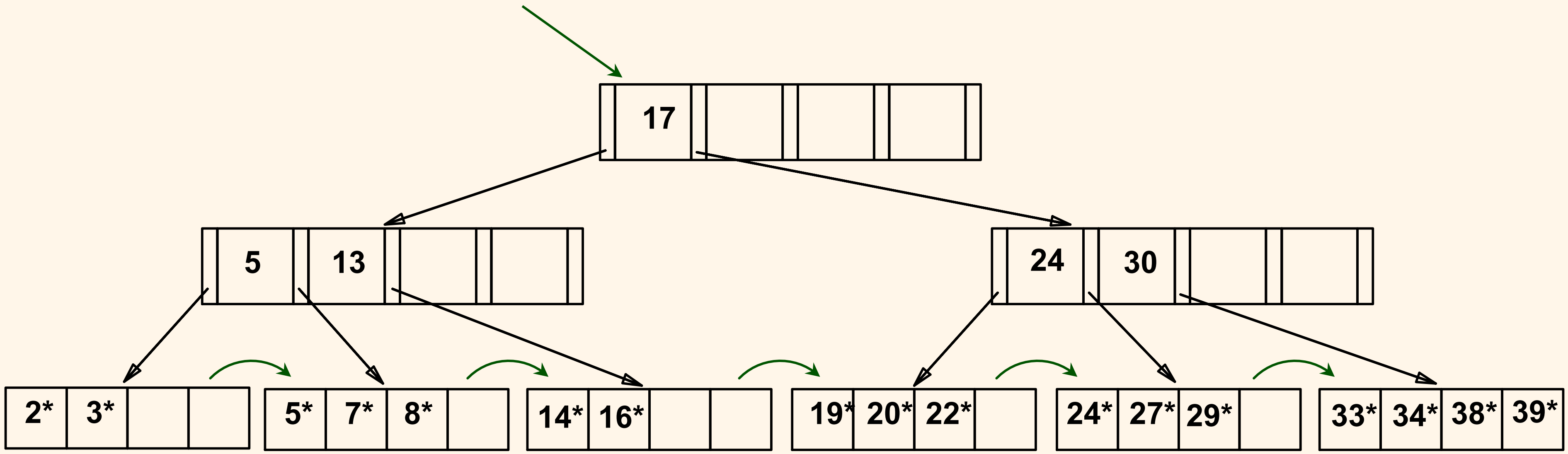


Inserting 8* into Example B+ Tree



- Observe how the minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this!

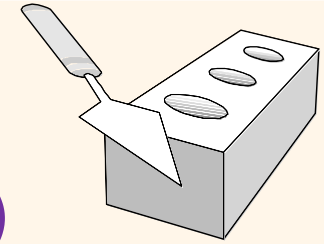
*Example B+ Tree **After** Inserting 8**

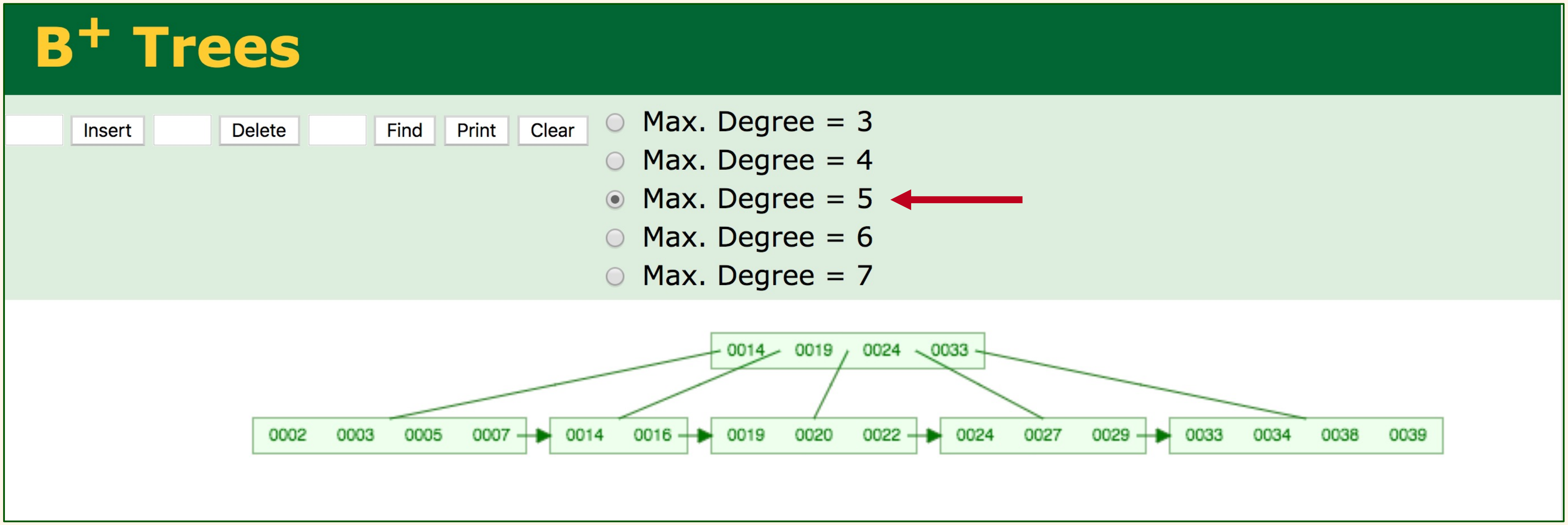


❖ Notice that root was split, leading to ***increase in height***.

❖ In this example, could avoid leaf split by redistributing entries; but, not usually done in practice. (***Q:*** Any idea why?)

Let's Go Live…! *(Come to class!)*

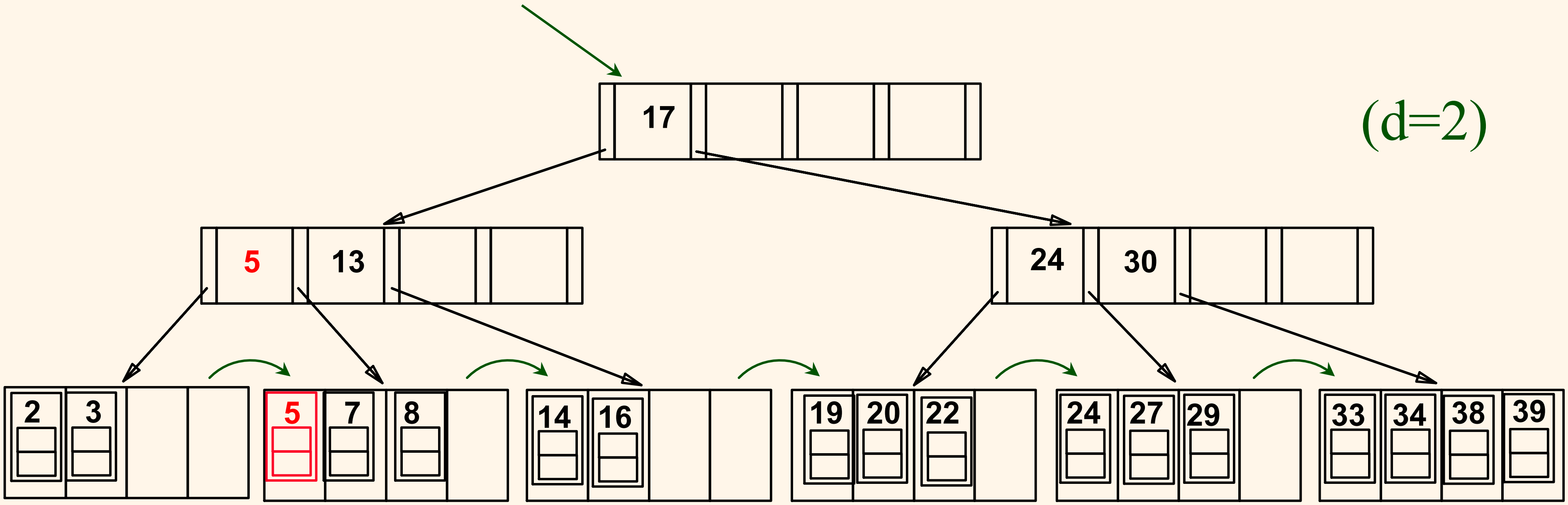




Note: Very cool online B+ tree viz tool available (☺)

- https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
- Only slight differences from our defs (e.g., key 13 above → 14)
- Their "*Max. Degree*" is our 2d+1 (limit of ***5 ptrs/node*** above)
- **Insert**: 2, 3, 14, 16, 19, 20, 24, 27, 33, 34, 5, 7, 38, 39, 22, 29

A Star Is (Un-)Born...*

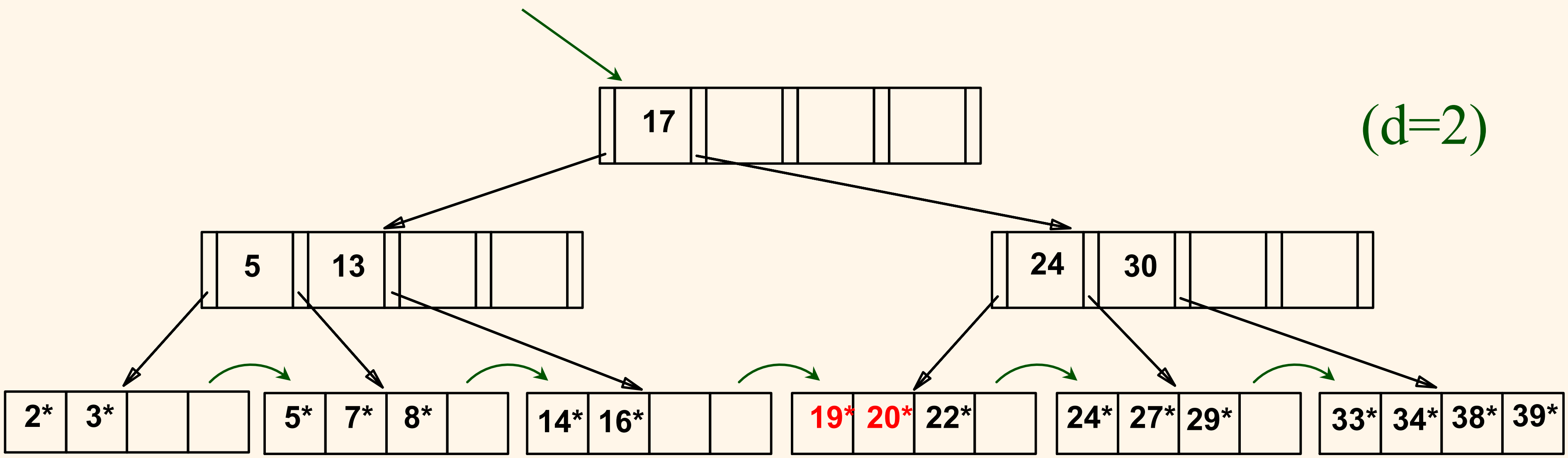


- Hopefully the picture above clarifies what's been more vaguely denoted by the * notation...!
- Again: the leaves are where the ***data*** (like 5*, *a.k.a.* I(5)) actually lives!

Deleting a Data Entry from a B+ Tree

- Start at root, find leaf *L* where entry should be.
- Remove the entry.
  - If L is still at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to redistribute by borrowing from <u>*sibling*</u> *(adjacent node with same parent as L)*.
    - If re-distribution fails, <u>*merge*</u> *L* and sibling.
- If merge occurred, must delete search-guiding entry (pointing to *L* vs. sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

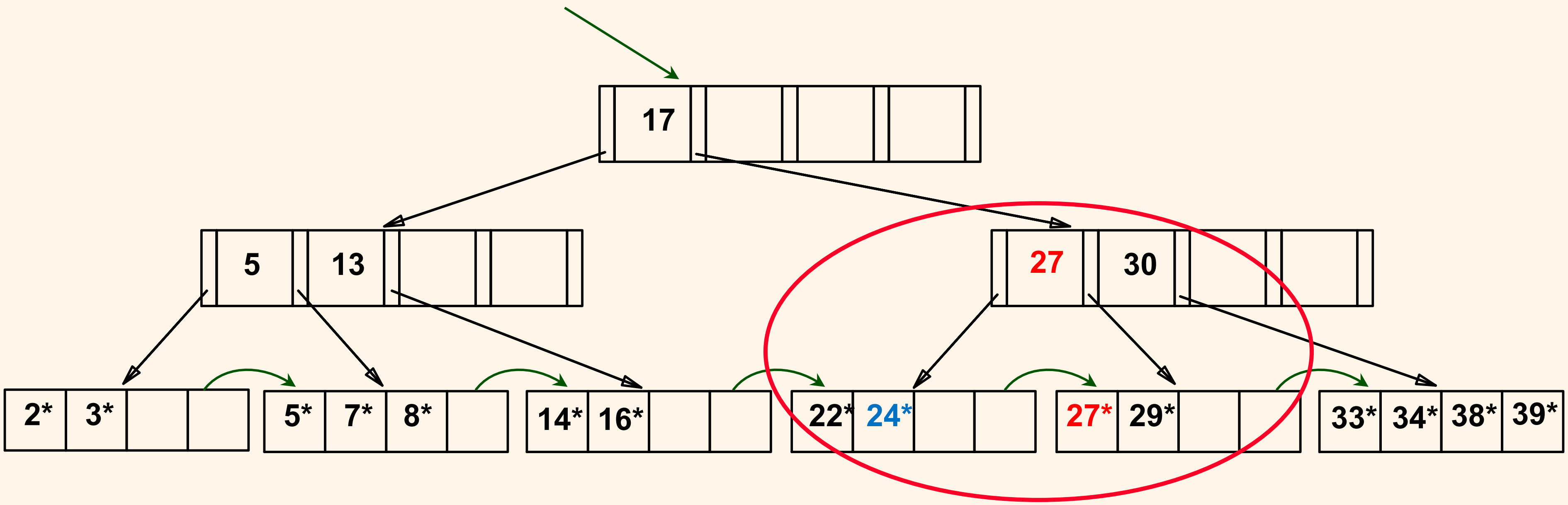
(Our previous B+ Tree, including 8*)



(d=2)

(Let's delete: 19, 20)

Tree After Deleting 19 and 20* ...*



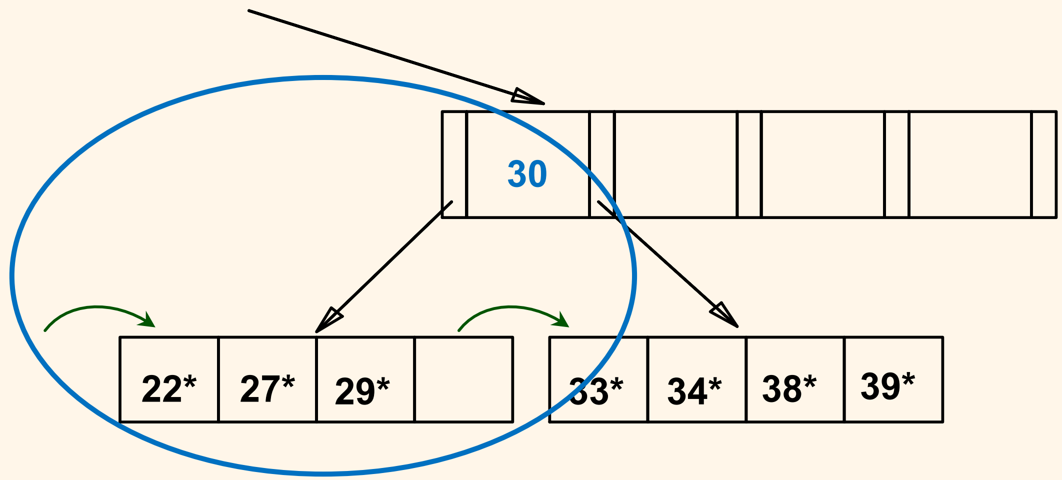
- Deleting 19* is easy.
- Deleting 20* is done with redistribution. Notice how middle leaf key (27) was *copied up*.

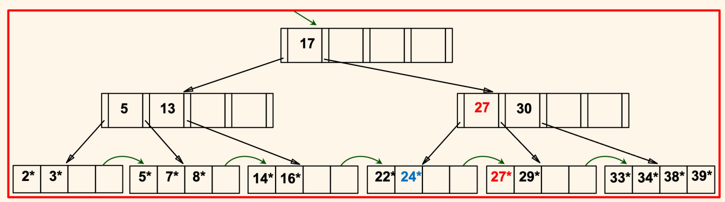
... Now Let's Delete 24*

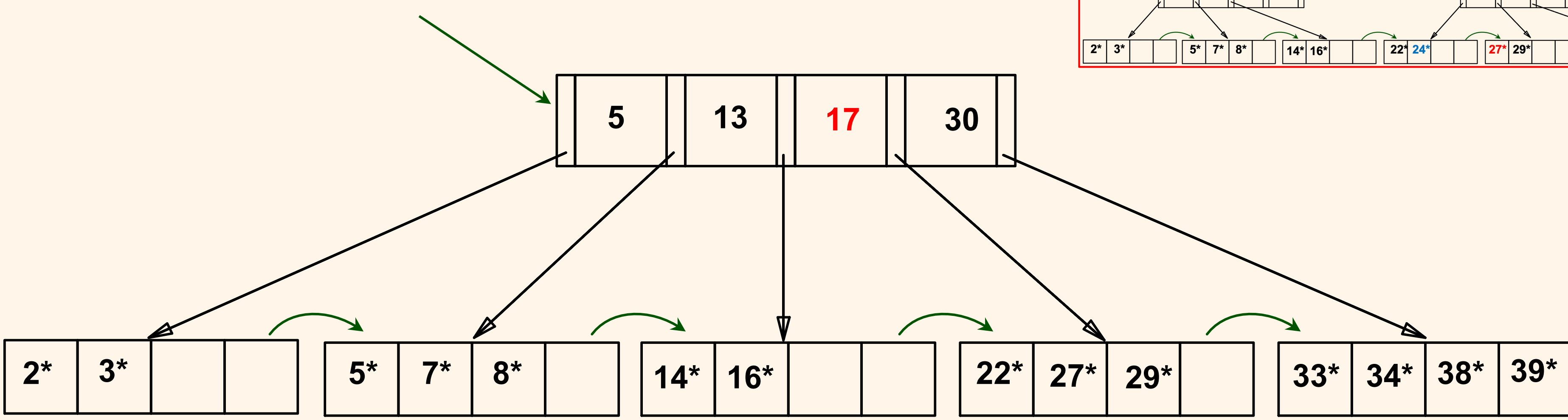
- Must merge this time.
- Observe "*toss*" of index entry (between), and "*pull down*" of index entry (below).

(~~27~~)

(17)

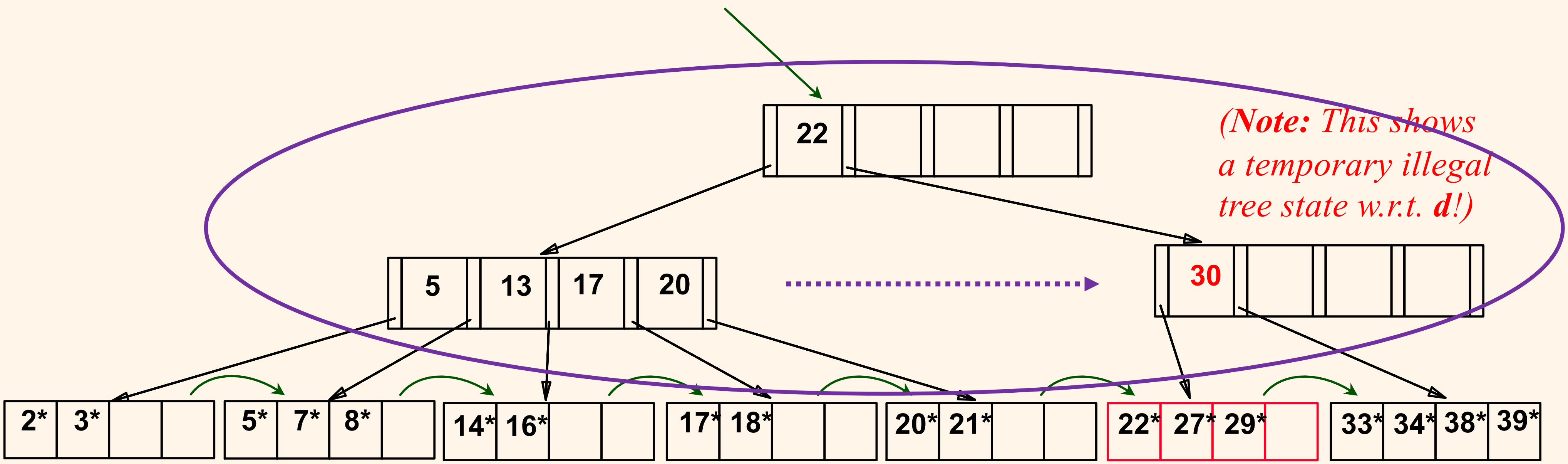






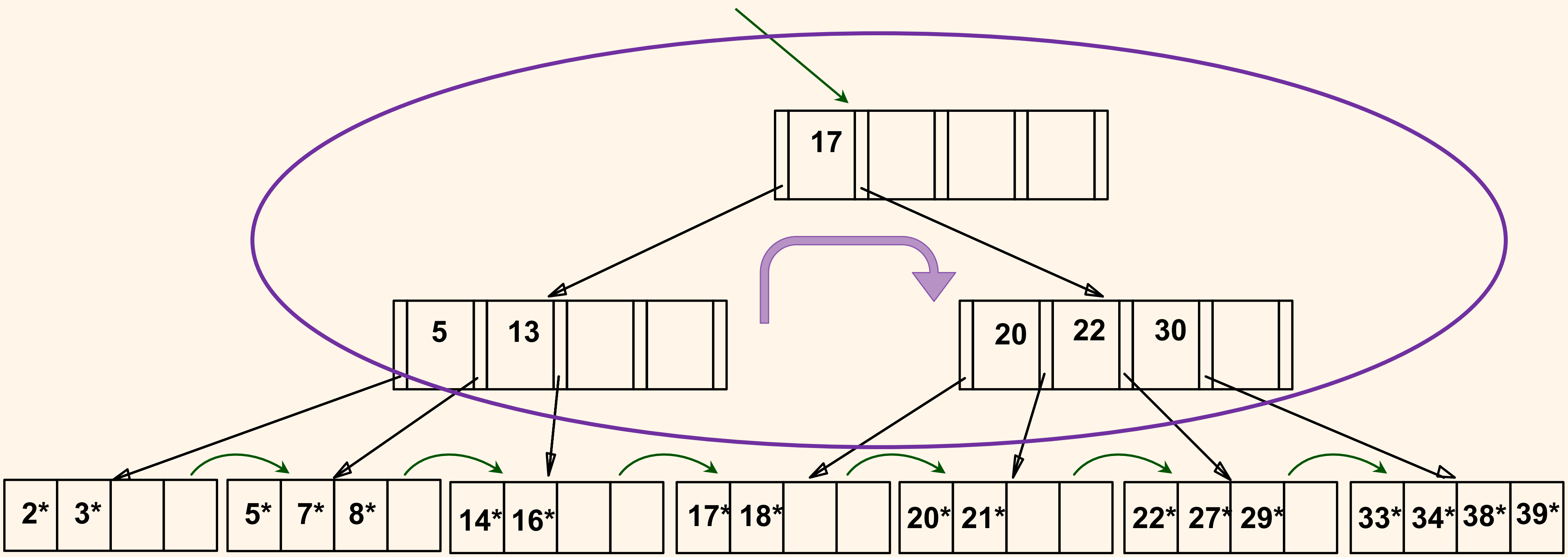
Example of Non-leaf Redistribution

- New/different example B+ tree is shown below *during deletion* of an entry 24*
- In contrast to previous example, can redistribute entry from root's left child to its right child.



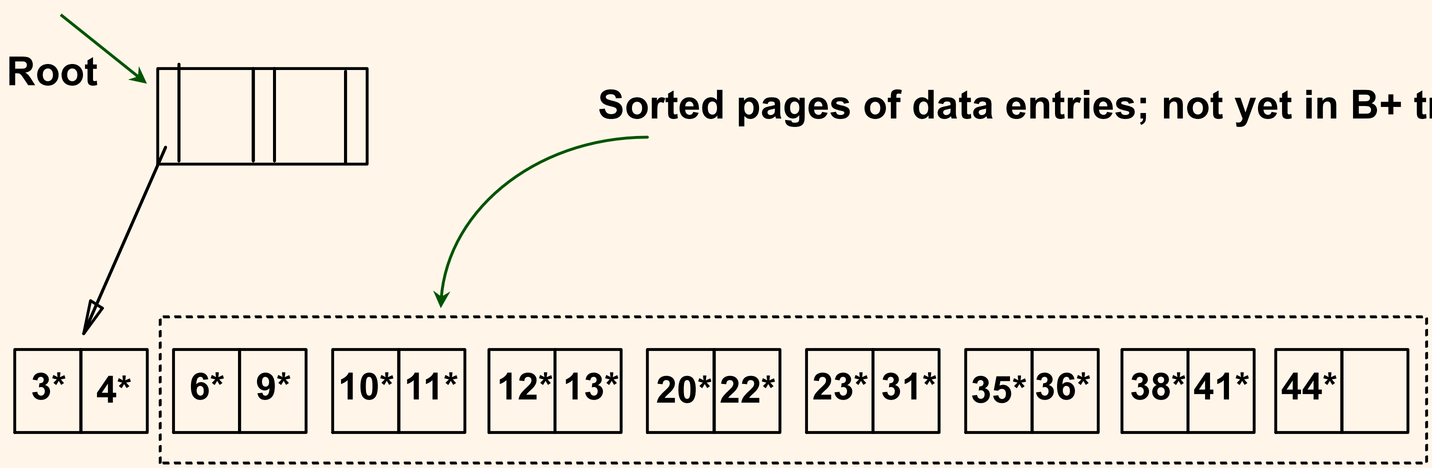
After Redistribution

- Intuitively, entries are redistributed by "*pushing through*" (or "rotating", if you prefer) the splitting entry in the parent node.



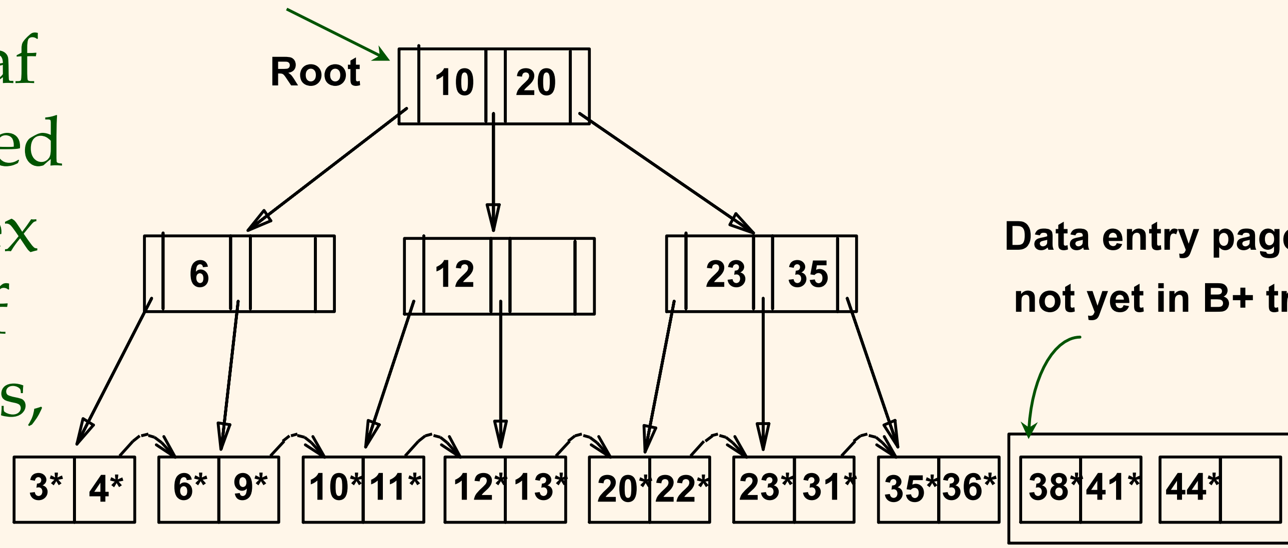
Bulk Loading of a B+ Tree

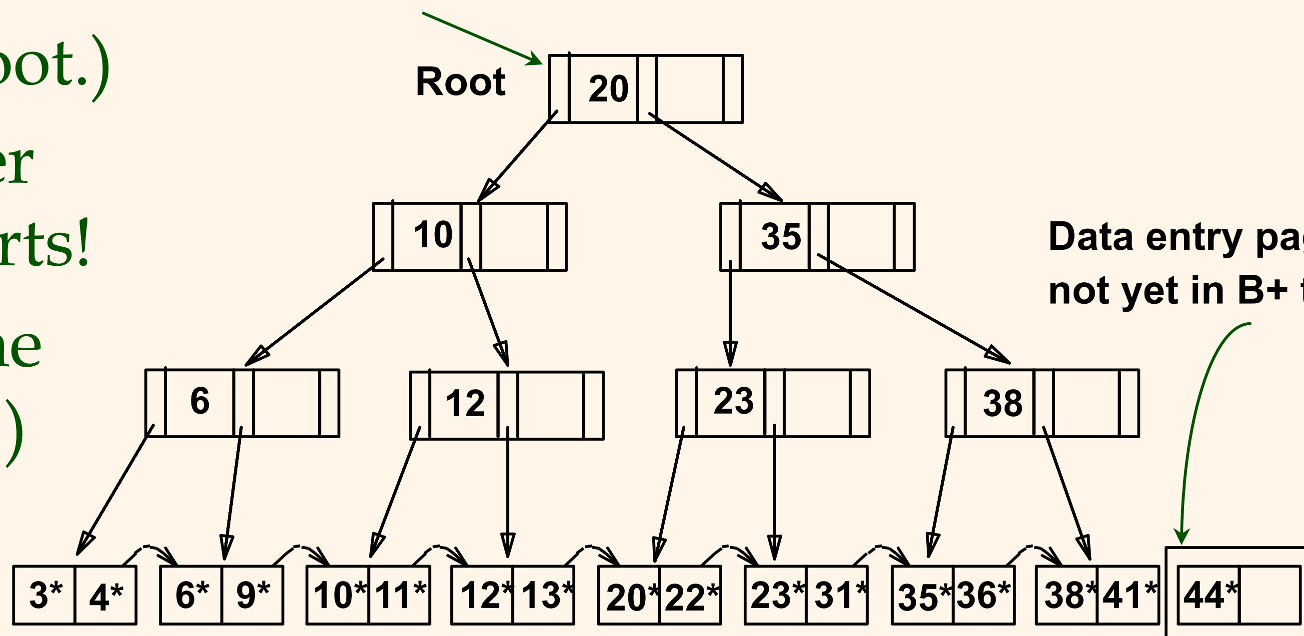
- If we have a large collection of records, and we want to create a B+ tree on some field, to do so by repeatedly inserting records would be slow.
- *Bulk Loading* can be done much more efficiently!
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



Bulk Loading (cont'd.)

- Index entries for leaf pages always entered into right-most index page just above leaf level. When one fills, it splits. (A split may go up the right-most path to the root.)
- Much (much) faster than repeated inserts!
- Can also control the leaf" fill factor" (%)





To Be Continued...