Tree-Structured Indexes (Brass Tacks)

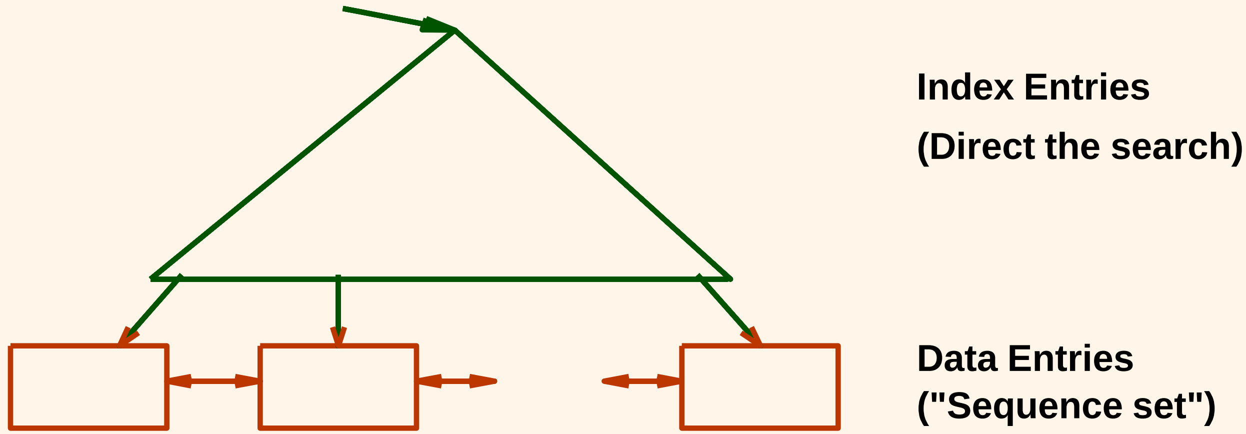
Chapter 10 Ramakrishnan and Gehrke
(Sections 10.3-10.8)

What will I learn from this set of lectures?

- How do B+trees work (for search)?
- How can I tune B+trees for performance?
- How can I maintain its balance against inserts and deletes?
- How do I build a B+tree from scratch?
- How can I handle key values that are very long – e.g., long song names or long names of people?

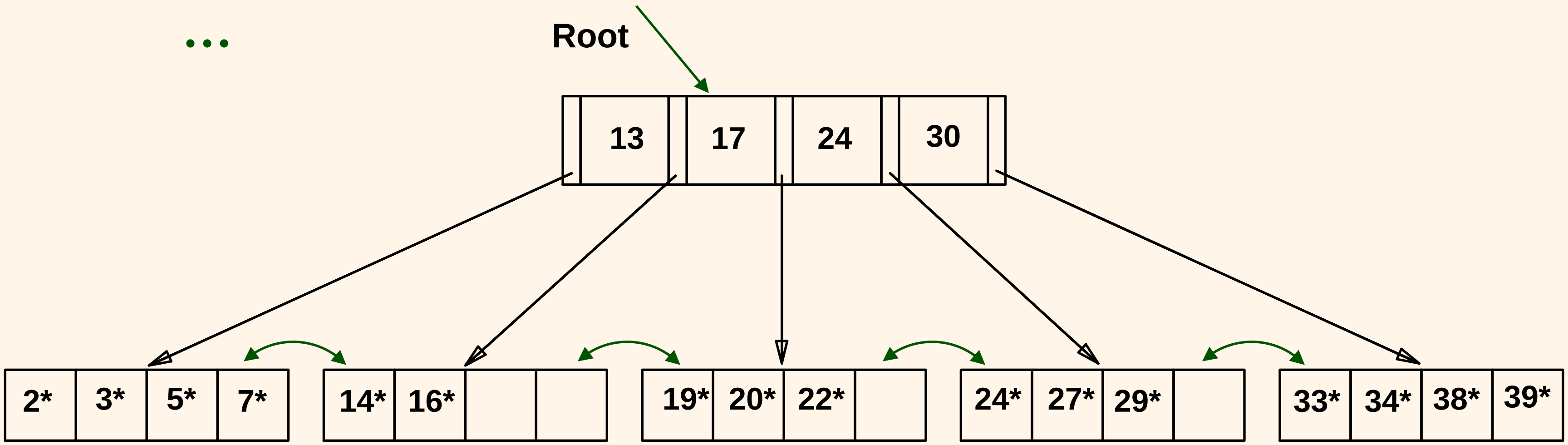
B+ Tree: The Most Widely Used Index

- 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** <= *m* <= 2**d** entries. The parameter **d** is called the *order* of the tree.
- Supports equality and range-searches efficiently.



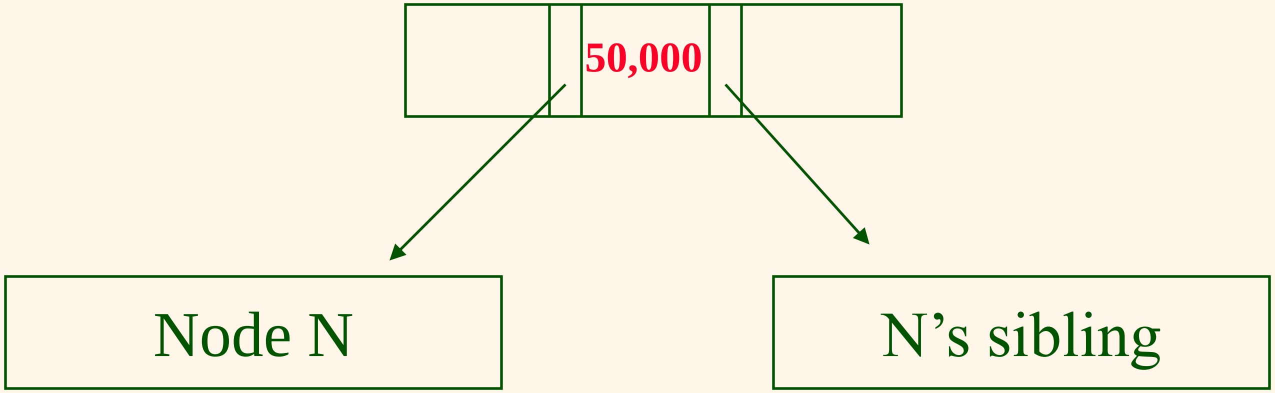
Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM)
- Binary search within a node
- Search for 5*, 15*, all data entries >= 28* ...



insert

B+Trees basics



The key 50,000 separates or discriminates between N and its sibling. Plays a crucial role in search and update maintenance. Call 50,000 the separator/discriminator of N and its sibling.

B+ Trees in Practice

- Typical order: 100. Typical fill-factor: ln 2 = 66.5% (approx).
  - average fanout = 2 x 100 x 66.5% = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,721 pages.
  - Height 3: 133^3 = 2,352,637 pages
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 MByte (approx.)
  - Level 3 = 17,689 pages = 133 MBytes (approx.)
  - Level 4 = 2,352,637 pages = 17.689 GBytes (approx.)
  - Level 5 = 312,900,721 pages = 2.352637 tera bytes! (approx.)
- For typical orders (d ~ 100-200), a shallow B+tree can accommodate very large files. How tall a B+tree do we need to cover all of Canada's taxpayer records?

B+ Trees in practice

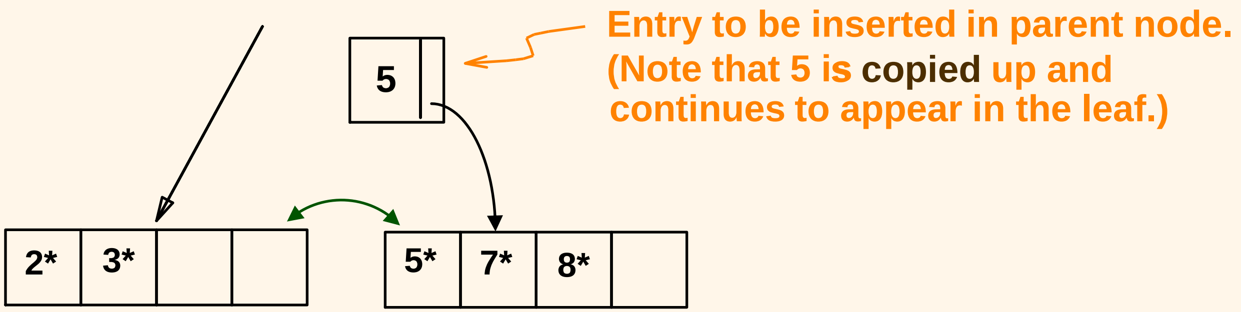
- Suppose a node is implemented as a block, a block is 4 K, a key is 12 bytes, whereas a pointer is 8 bytes.
  - For a file occupying b (logical) blocks, what is the min/max/avg height of a B+tree index?
  - If we have m bytes of RAM, how many levels of the B+tree can we prefetch to speed up performance?

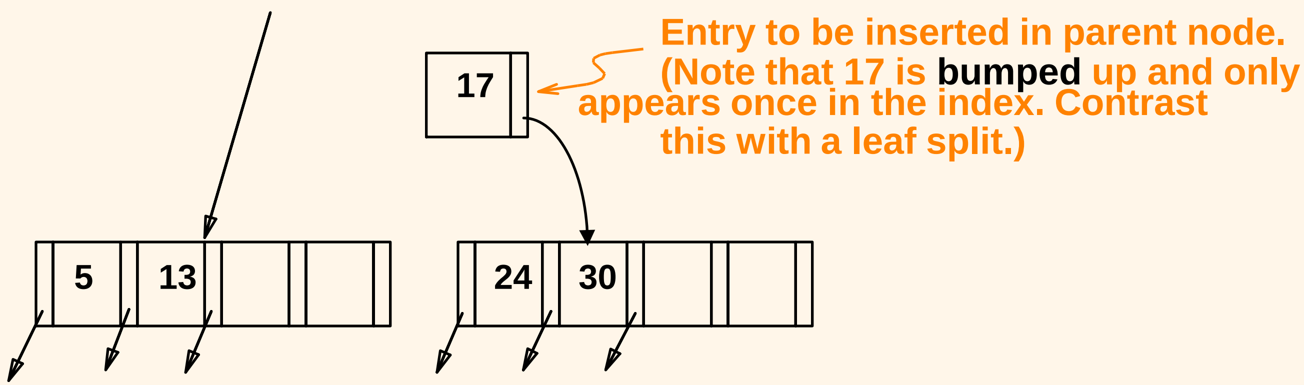
Inserting a Data Entry into a B+ Tree

- 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* (i.e., the middle key copied up) into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **bump up** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider, maybe even one level taller.*

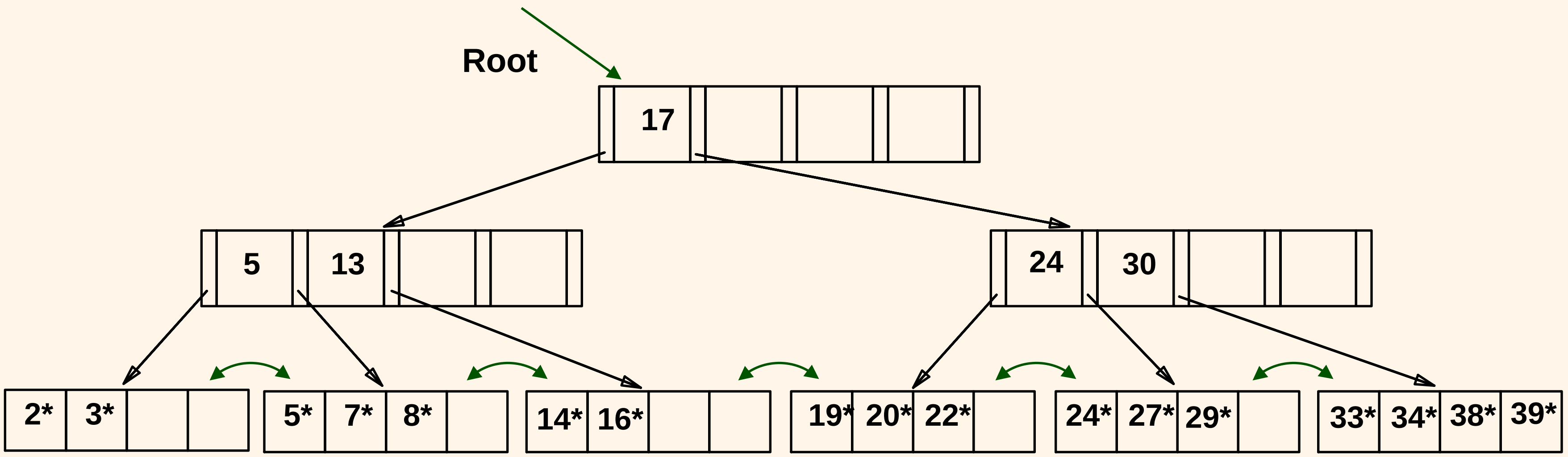
Inserting 8 into Example B+ Tree*

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between *copy-up* and *bump-up*; Why do we handle leaf page split and index page split differently?





Example B+ Tree After Inserting 8*



- ❖ Notice that root was split, leading to increase in height.
- ❖ In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

What would it look like?

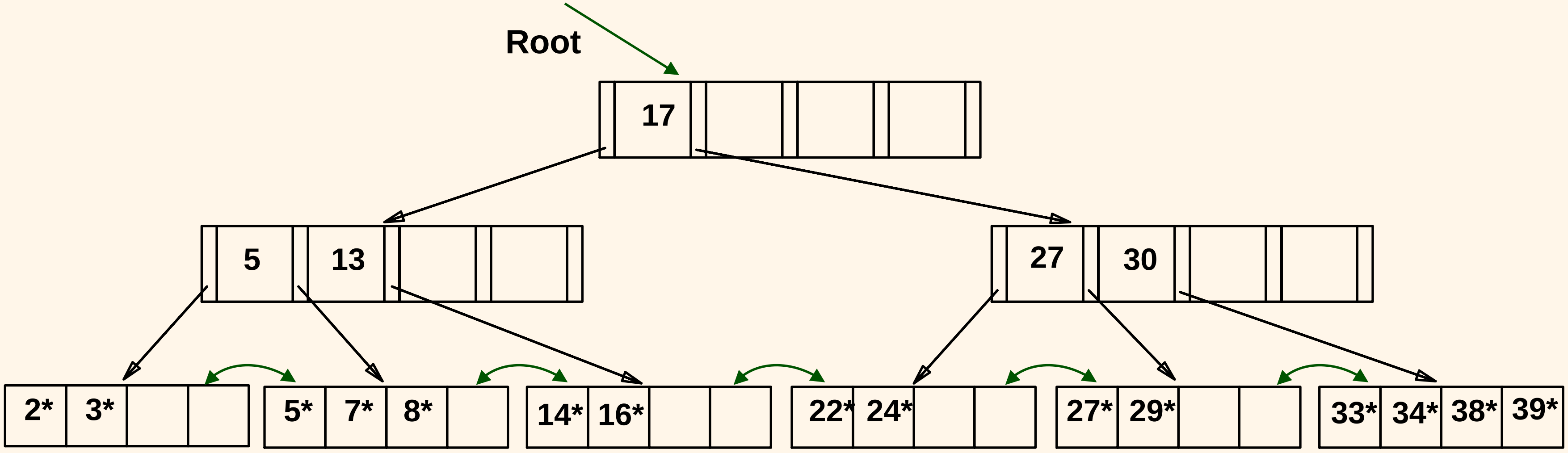
delete

Deleting a Data Entry from a B+ Tree

- 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)*. Adjust the key that separates L and its sibling.
    - If re-distribution fails, *merge* *L* and sibling.
- If merge occurred, must delete separator entry (discriminating *L* & its sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

Question: Can L's occupancy ever drop below d-1?

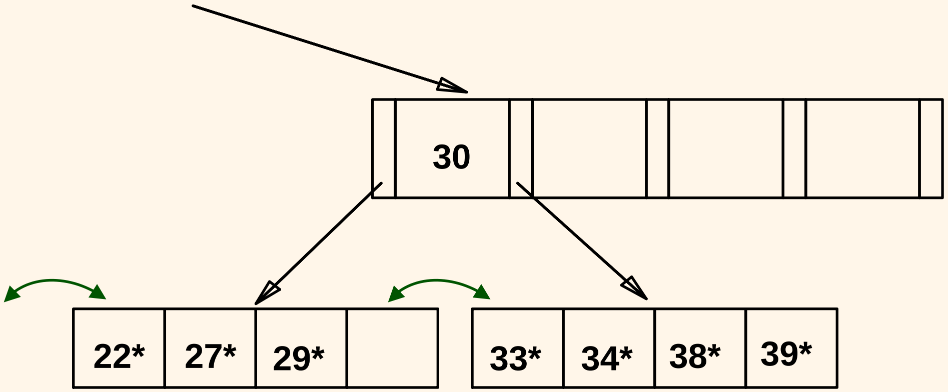
Example Tree After (Inserting 8* Then) Deleting 19* and 20* ...

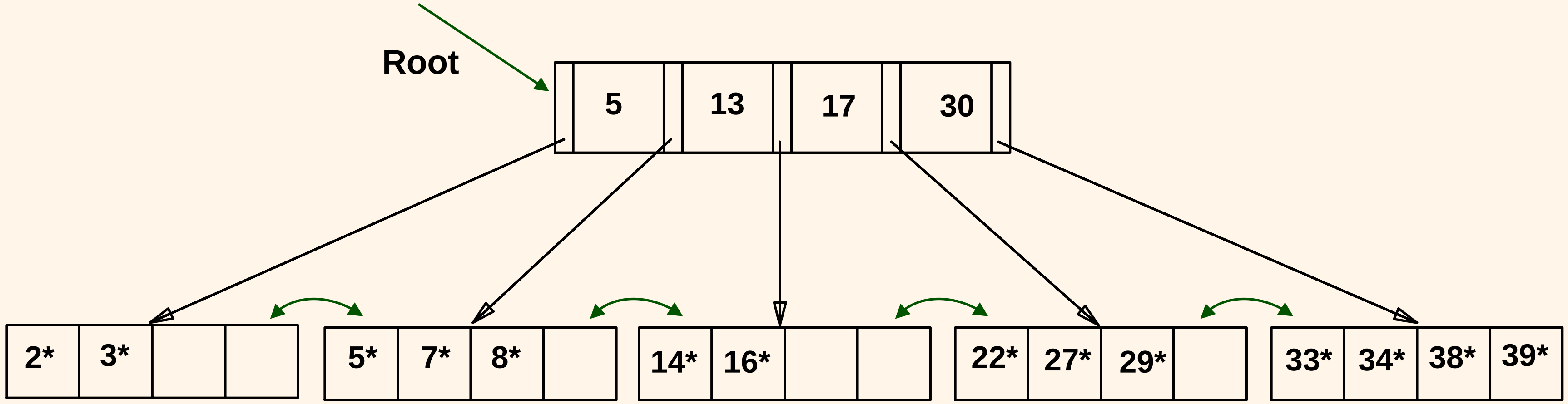


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how new middle key is *copied up*.

... And Then Deleting 24*

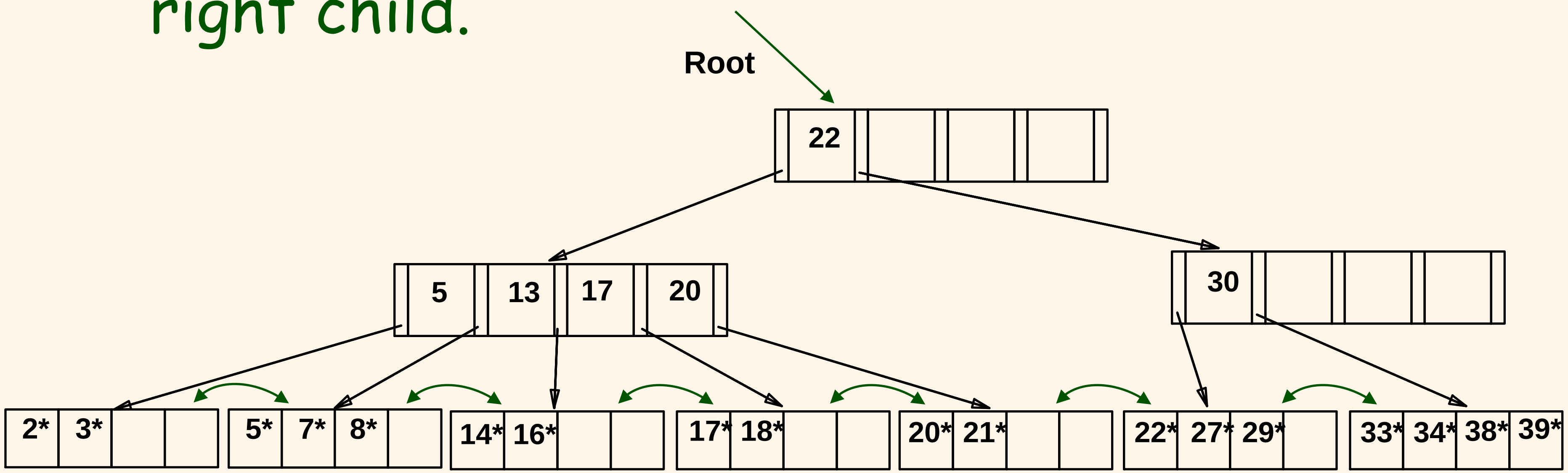
- Must merge.
- Observe `*toss*' of index entry (on right), and `*pull down*' of index entry (below).





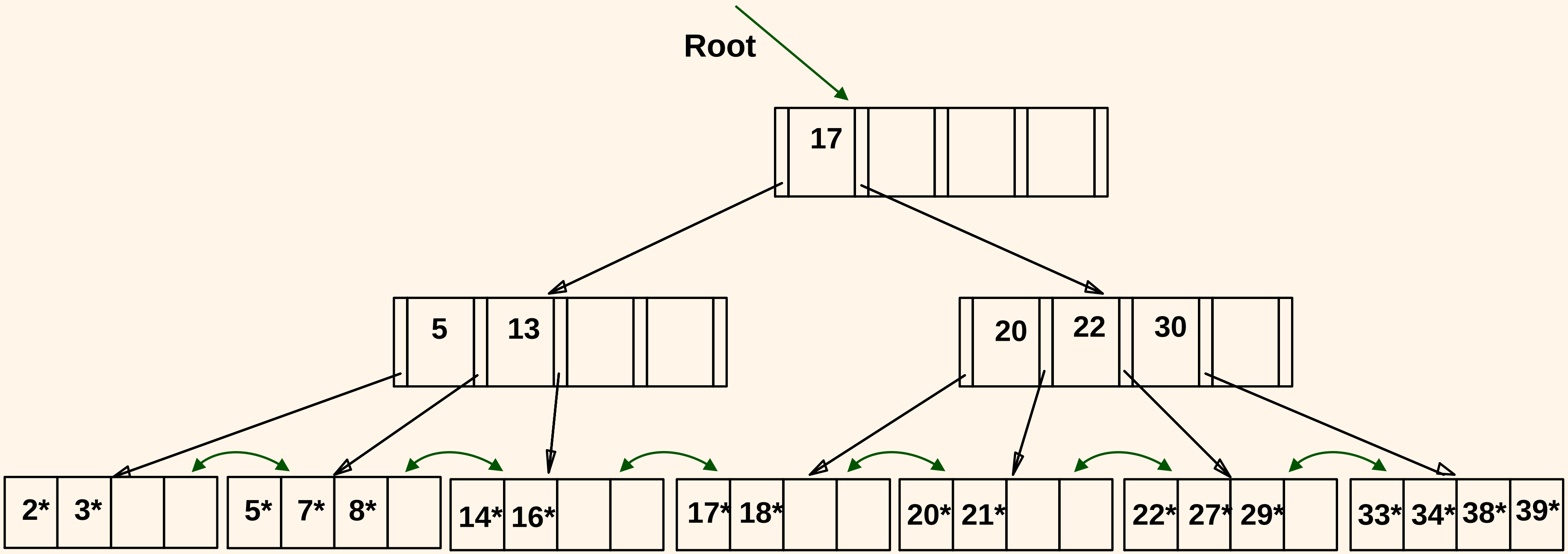
Example of Non-leaf Redistribution

- Tree is shown below ***during*** *deletion* of 24*. (What could be a possible initial tree?)
- In contrast to previous example, can re-distribute entry from left child of root to right child.



After Re-distribution

- Intuitively, entries are re-distributed by "*pushing through*" the splitting entry (i.e., separator) in the parent node.
- It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.



Optimization 1: Prefix Key Compression

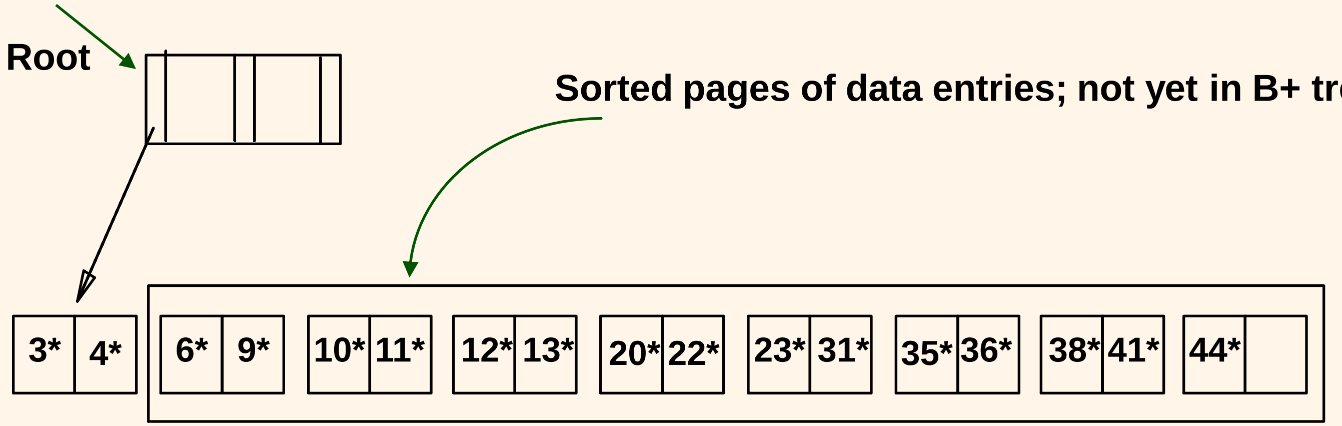
- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic'; can often compress them.
  - E.g., If we have adjacent **index entries** with search key values *Dannon Yogurt*, *David Smith* and *Devarakonda Murthy*, we can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)
    - Is this correct? Not quite! What if there is a **data entry** *Davey Jones*? (Can only compress *David Smith* to *Davi*)
    - In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.
- Insert/delete must be suitably modified.
- This idea works for any field/attribute whose data type is a long string: e.g., customer code, shipping tracking number, etc.

Analyzing insert/delete time.

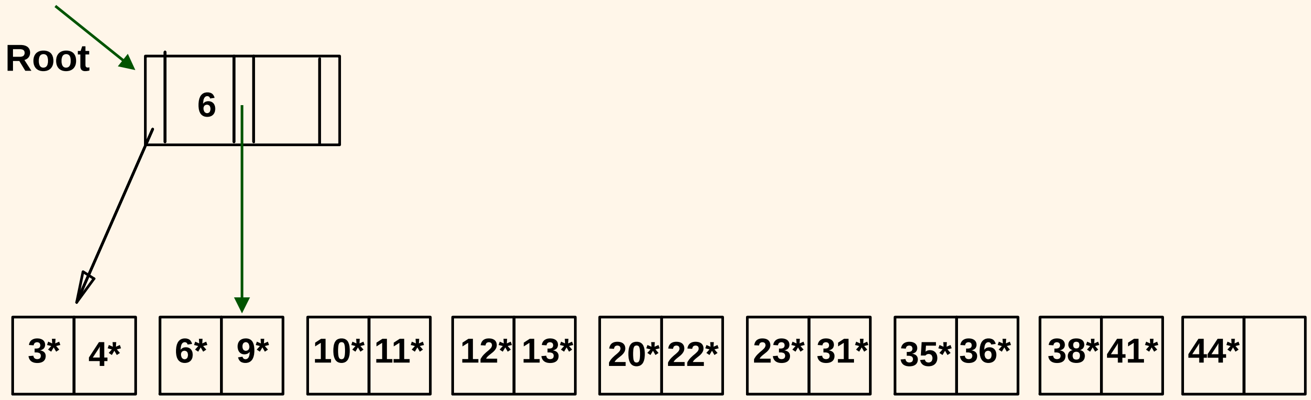
- Given a B+tree index with height h, what can you say about best case, average case, and worst case I/O?
- It must have something to do with h.
- But is it exactly h?
- What exactly constitutes best, averege, and worst case?

Optimization 2: Bulk Loading of a B+ Tree

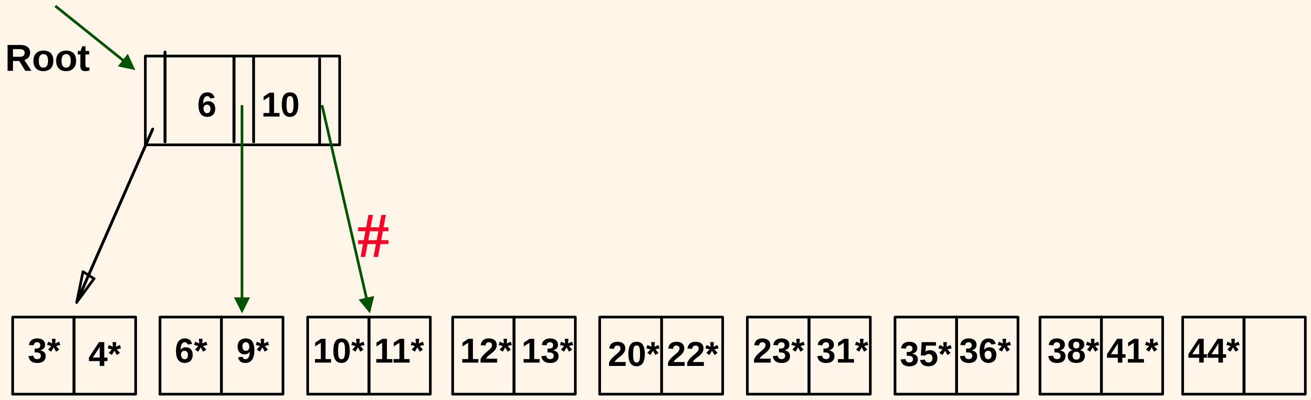
- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- *Bulk Loading* can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



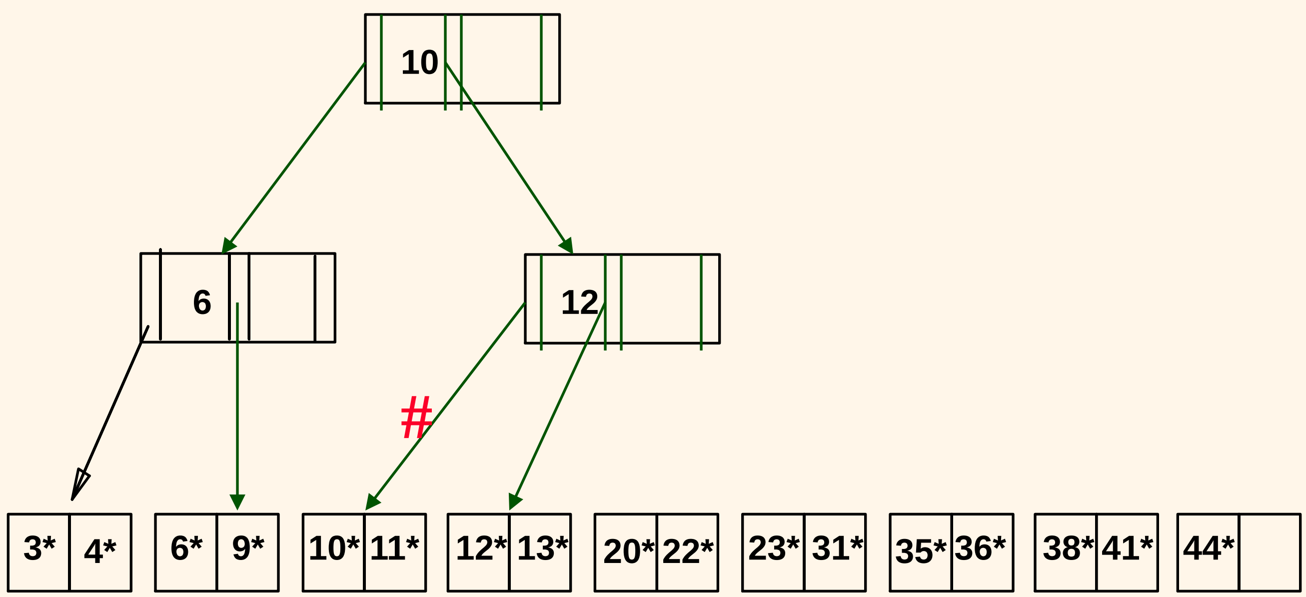
Bulk Loading (contd.)



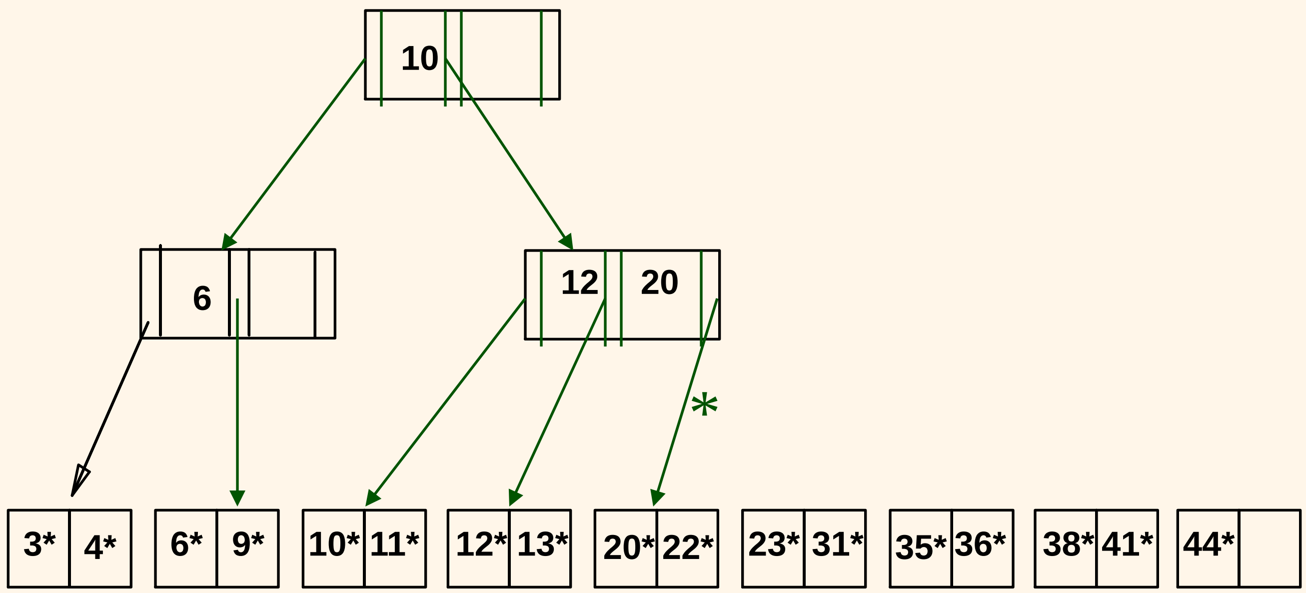
Bulk Loading (contd.)



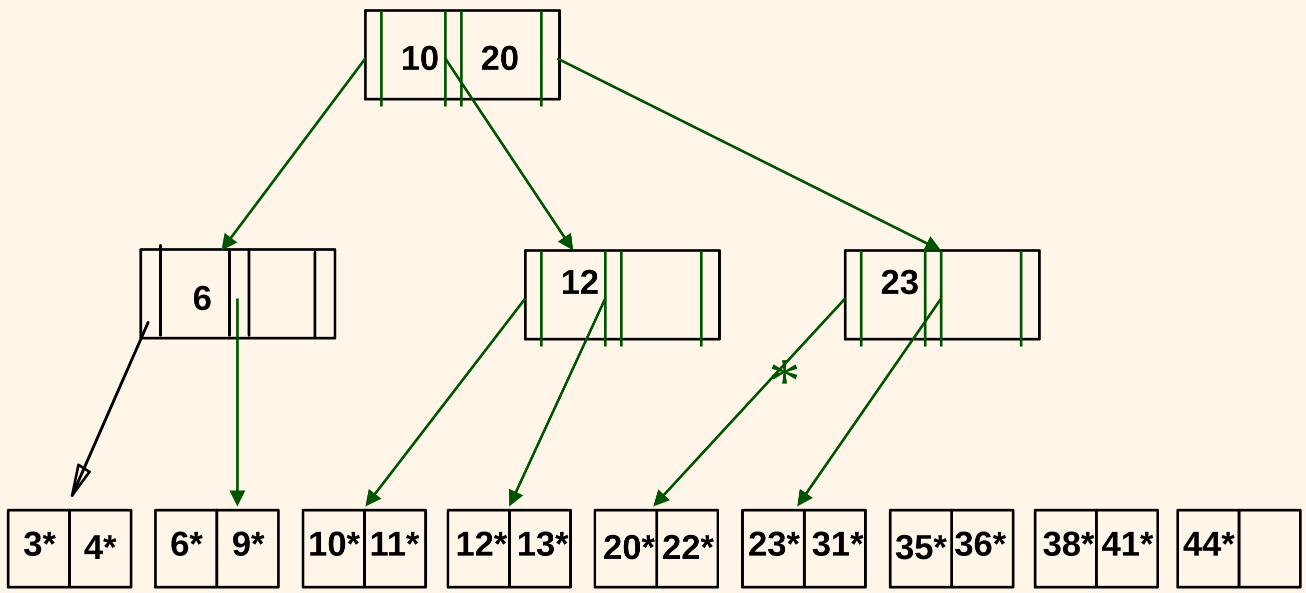
Bulk Loading (contd.)



Bulk Loading (contd.)

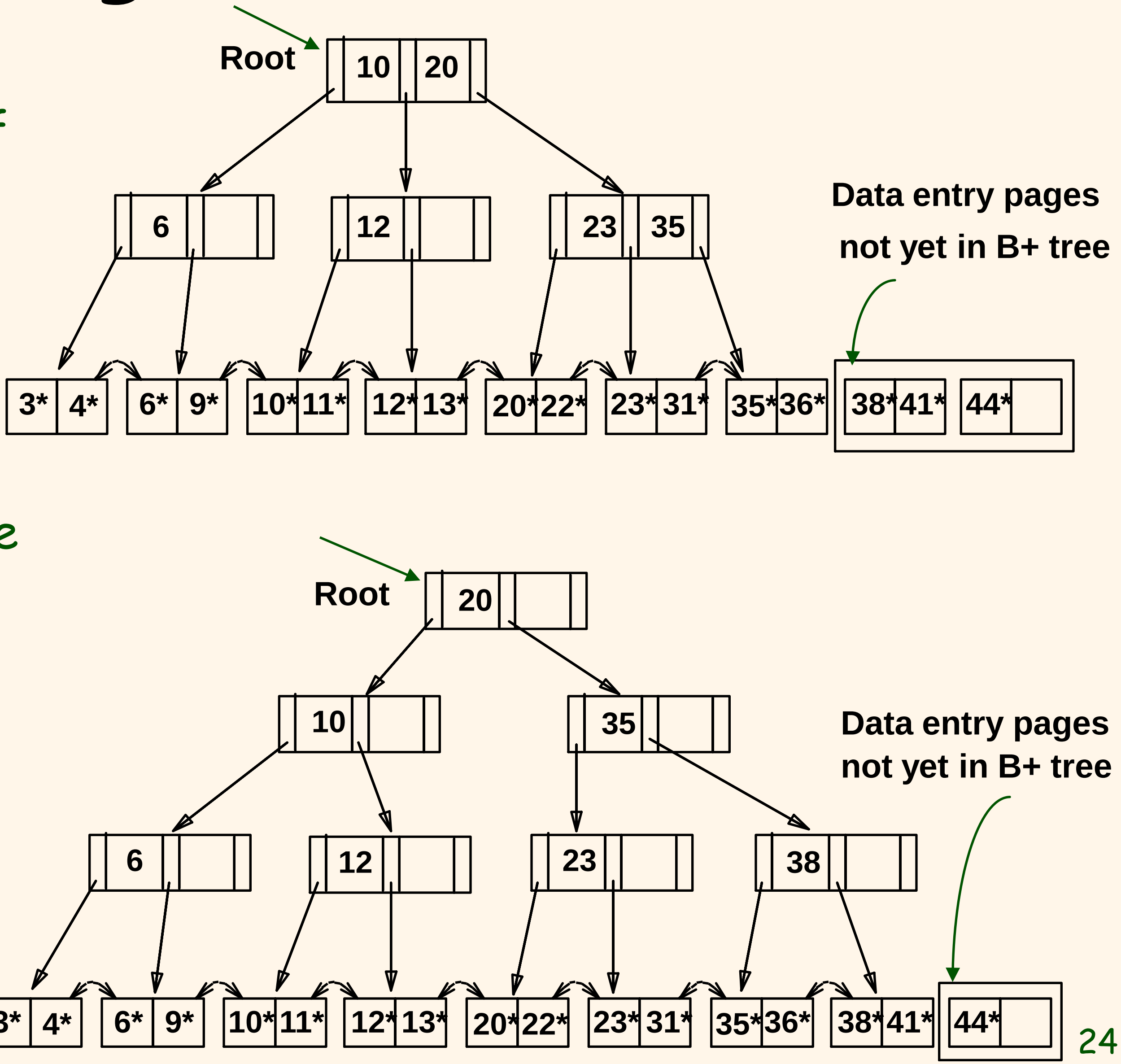


Bulk Loading (contd.)



Bulk Loading (Contd.)

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits.
  (Split may go up right-most path to the root.)
- Much faster than repeated inserts, especially when one considers locking!



Bulk Loading

- How would you estimate time taken to build a B+tree index from scratch?
  - Naïve approach of repeated insertions.
  - Bulk loading.
- Naïve approach: each insert is potentially a random block seek; cannot read/insert next data block until after previous block has been inserted. Very slow.
- Bulk loading: dominant factor – sorting data (or data entries as appropriate). Followed by another sequential read (w/ proper buffer management).
  - Explored in exercises.

Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not ensure sequential storage of leaves.
- Option 2: bulk loading
  - Has advantages for concurrency control.
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course).
  - Can control "fill factor" on pages.

Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- ISAM is a static structure.
  - Only leaf pages modified; overflow pages needed.
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced; $\log_F N$ cost.
  - High fanout (**F**) means depth (i.e., height) rarely more than 3 or 4.
  - Almost always better than simply maintaining a sorted file.

Summary (Contd.)

- Key compression increases fanout, reduces height.
- Bulk loading can be much faster than repeated inserts for creating a B+ tree on a large data set.
- Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.