Tree-Structured Indexes

courtesy of Joe Hellerstein for some slides

Jianlin Feng
School of Software
SUN YAT-SEN UNIVERSITY

Review: Files, Pages, Records

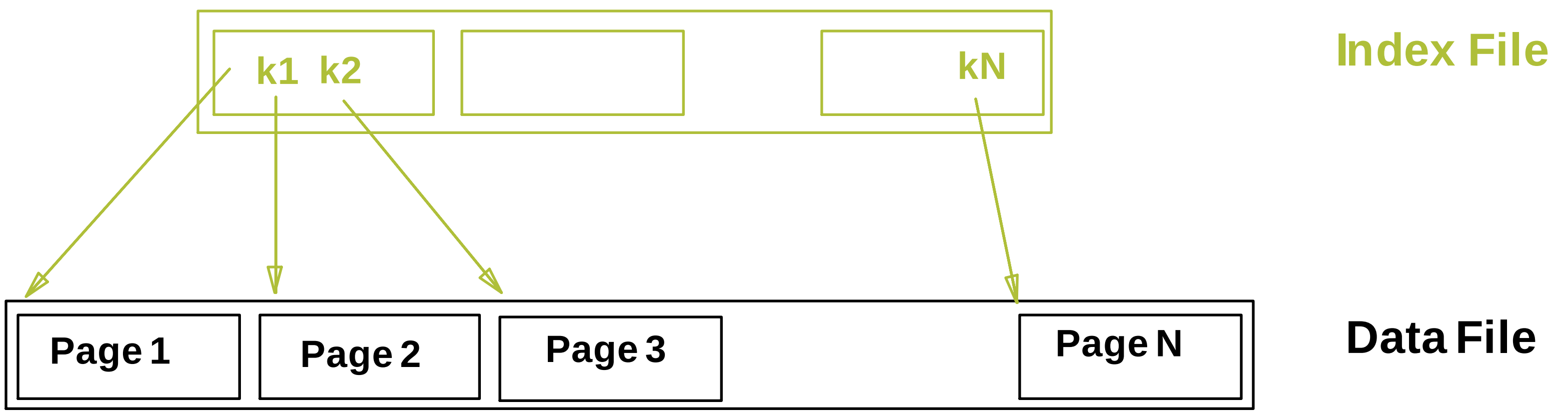
- Abstraction of stored data is “files” with “pages” of “records”.
  - Records live on *pages*
  - Physical Record ID (RID) = <page#, slot#>
  - Records can have fixed length or variable length.
- Files can be unordered (heap), sorted, or kind of sorted (i.e., “clustered”) on a *search key.*
- Indexes can be used to speed up many kinds of accesses. (i.e., “access paths”)

Tree-Structured Indexes: Introduction

- Selections of form: field <op> constant
- Equality selections (op is =)
  - Either "tree" or "hash" indexes help here.
- Range selections (op is one of <, >, <=, >=, BETWEEN)
  - "Hash" indexes don't work for these.
- More complex selections (e.g. spatial containment)
  - There are fancier trees that can do this…
- Tree-structured indexing techniques support both *range selections* and *equality selections*.
  - *ISAM*: static structure; early index technology.
  - *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

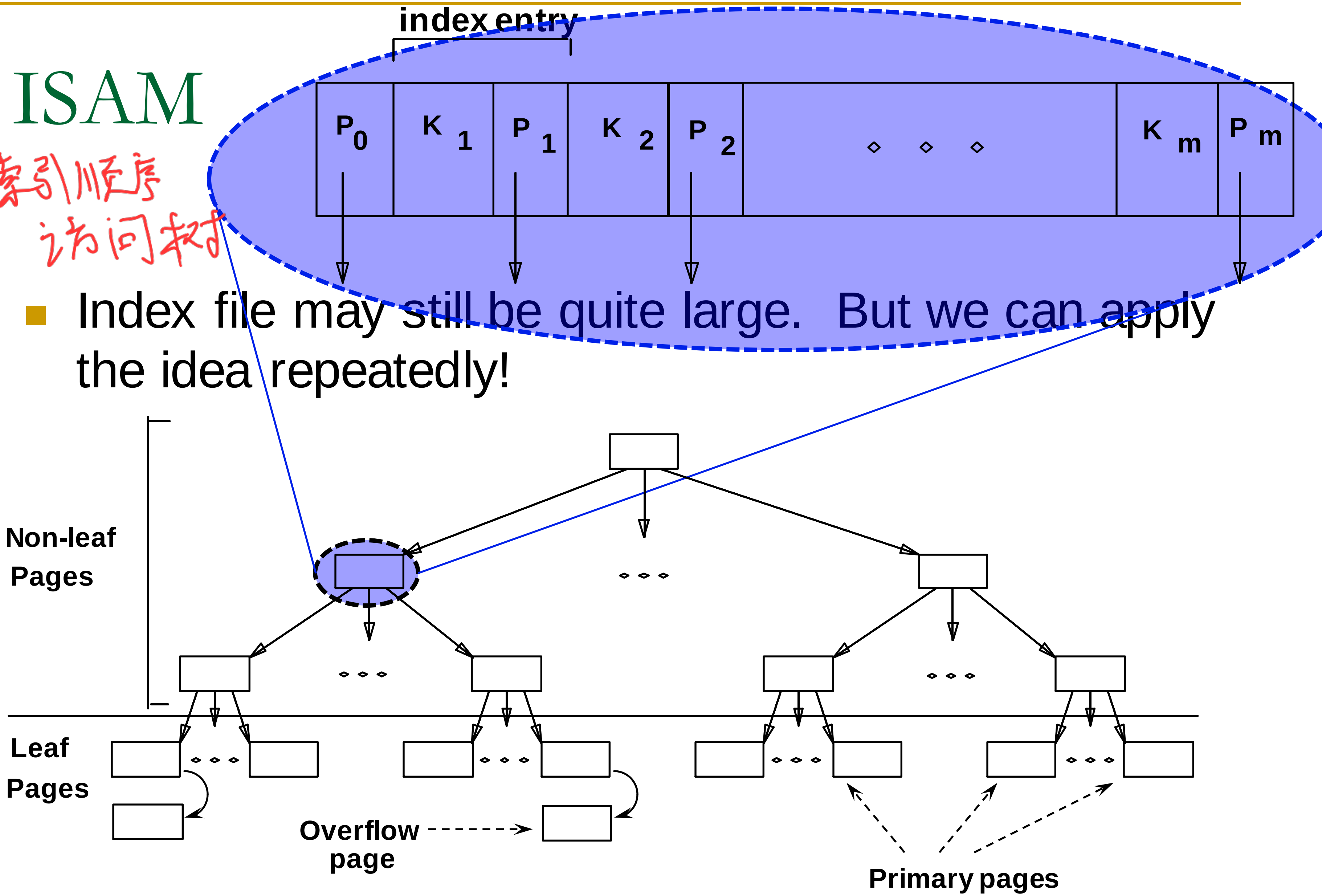
Range Searches

- *``Find all students with gpa > 3.0''*
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search in a database can be quite high.
    - Why???
- Simple idea: Create an `index' file, and then do binary search on (smaller) index file.



ISAM

索引顺序
访问树

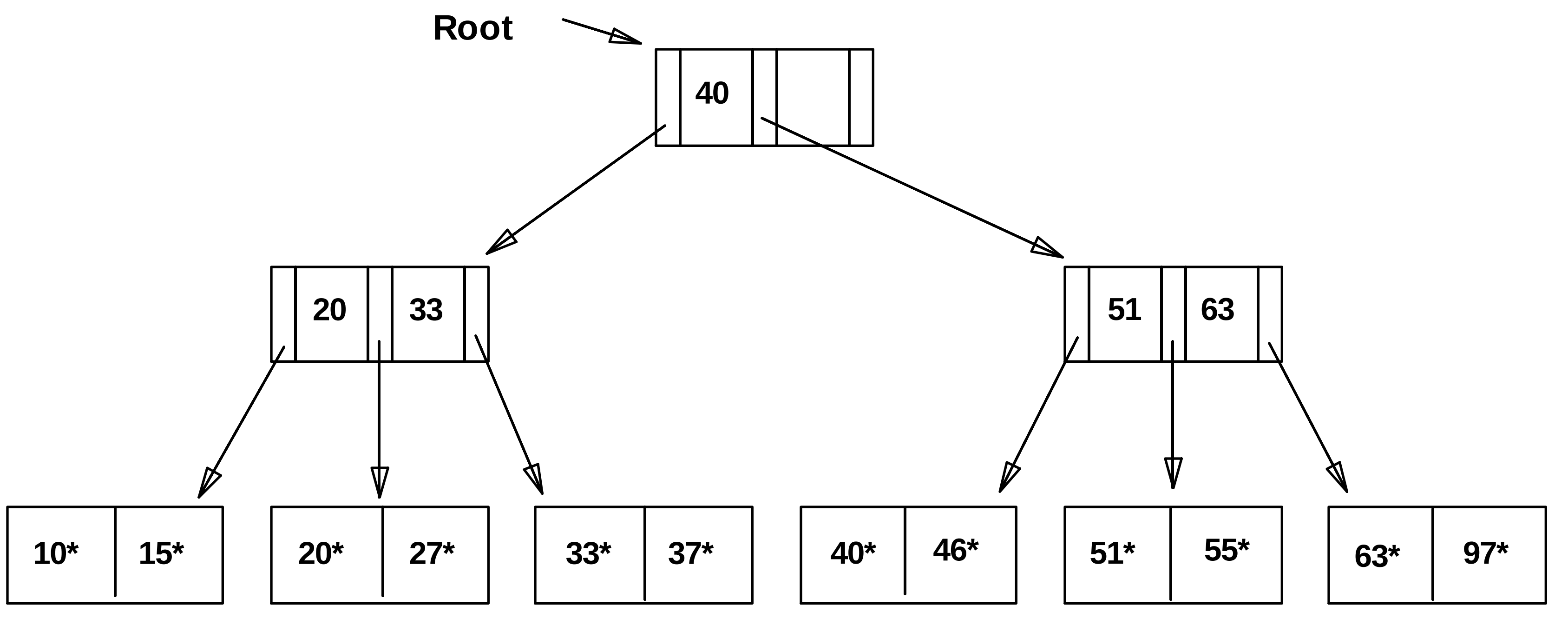


- Index file may still be quite large. But we can apply the idea repeatedly!

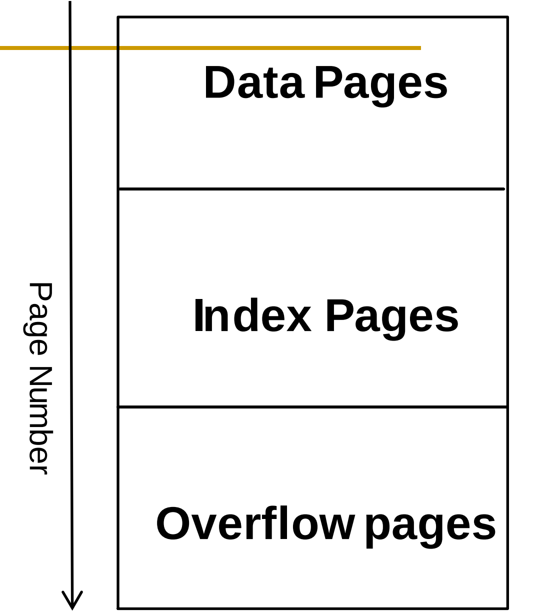
- *Leaf pages contain data entries.*

Example ISAM Tree

- ***Index entries*: <search key value, page id>, they direct search for data entries *in leaves.***
- **Example where each node can hold 2 entries;**



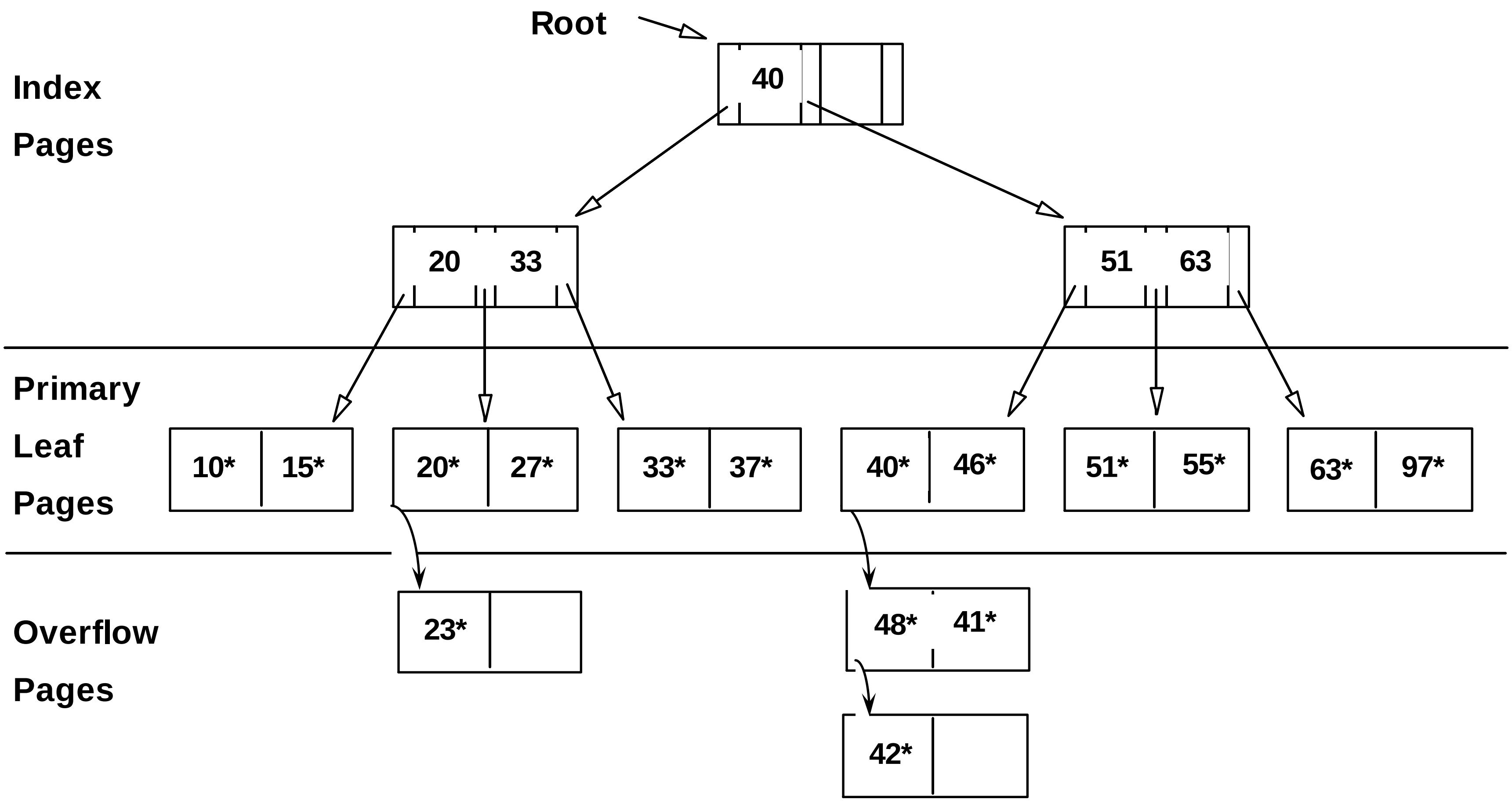
ISAM is a STATIC Structure



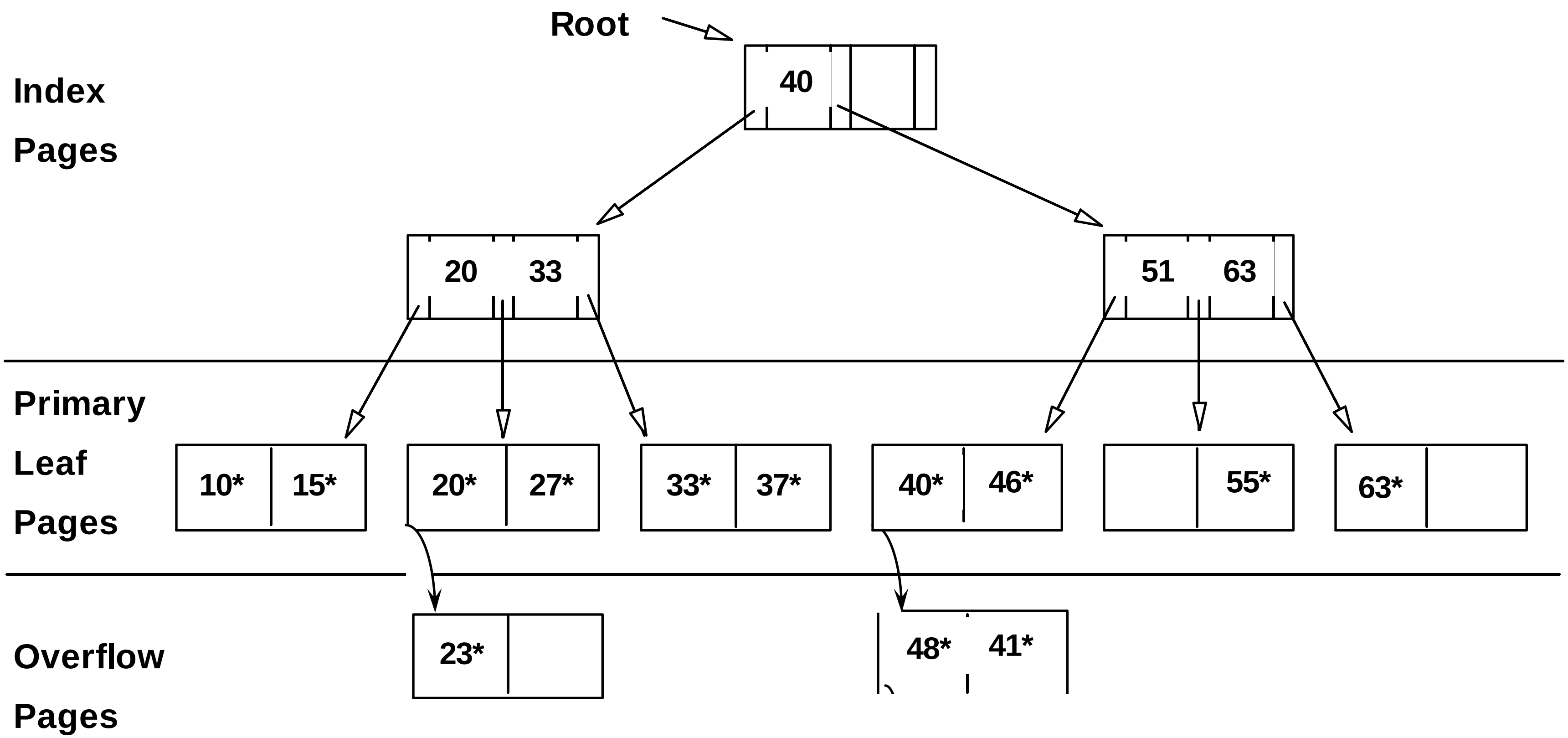
- *File creation*:
  - Leaf (data) pages allocated sequentially, sorted by search key
  - then index pages
  - then overflow pgs.
- *Search*: Start at root; use key comparisons to go to leaf.
- Cost = $\log_F N$
  - F = # entries/page (i.e., fanout)
  - N = # leaf pgs
  - no need for `next-leaf-page' pointers. (Why?)
- *Insert*: Find leaf that data entry belongs to, and put it there. Overflow page if necessary.
- *Delete*: Seek and destroy! If deleting a tuple empties an overflow page, de-allocate it and remove from linked-list.

Static tree structure: *inserts/deletes affect only leaf pages*.

Example: Insert 23*, 48*, 41*, 42*



... then Deleting 42*, 51*, 97*



- *Note that 51* appears in index levels, but not in leaf!*

B+ Tree Structure (1)

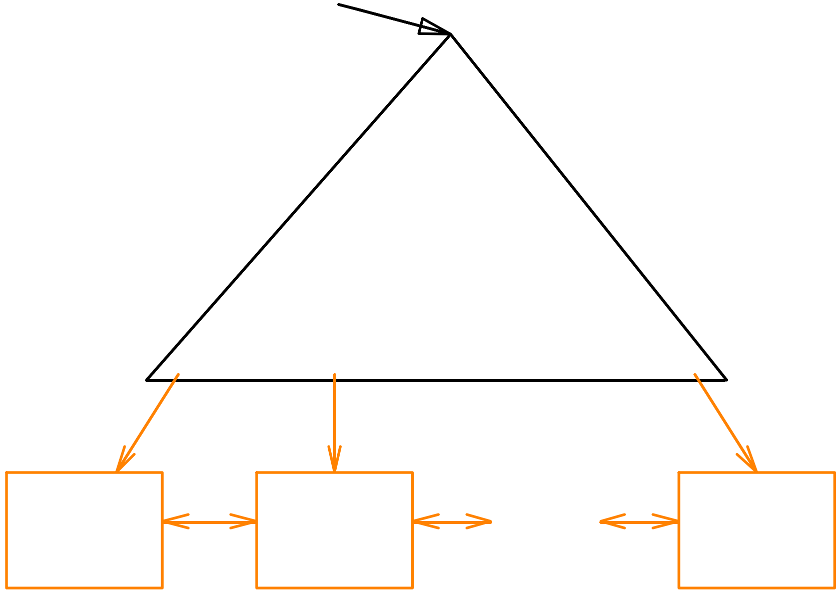
- The ROOT node contains between 1 and 2**d** index entries.
  - The parameter **d** is called the *order* of the tree.
  - An index entry is a pair of < key, page id>
  - the ROOT is a leaf or has at least two children.

- Each internal node contains m (**d** ≤ m ≤ 2**d**) index entries.
  - Each internal node has $m+1$ children.

- Each leaf node contains m (**d** ≤ m ≤ 2**d**) data entries
  - A data entry is one of <key, record> or <key, RID> or <key, list of RIDs>

B+ Tree Structure (2)

- Each path from the ROOT to any leaf has the same length.
  - Length is the number of nodes in a path.
- Supports equality and range-searches efficiently.



Index Entries

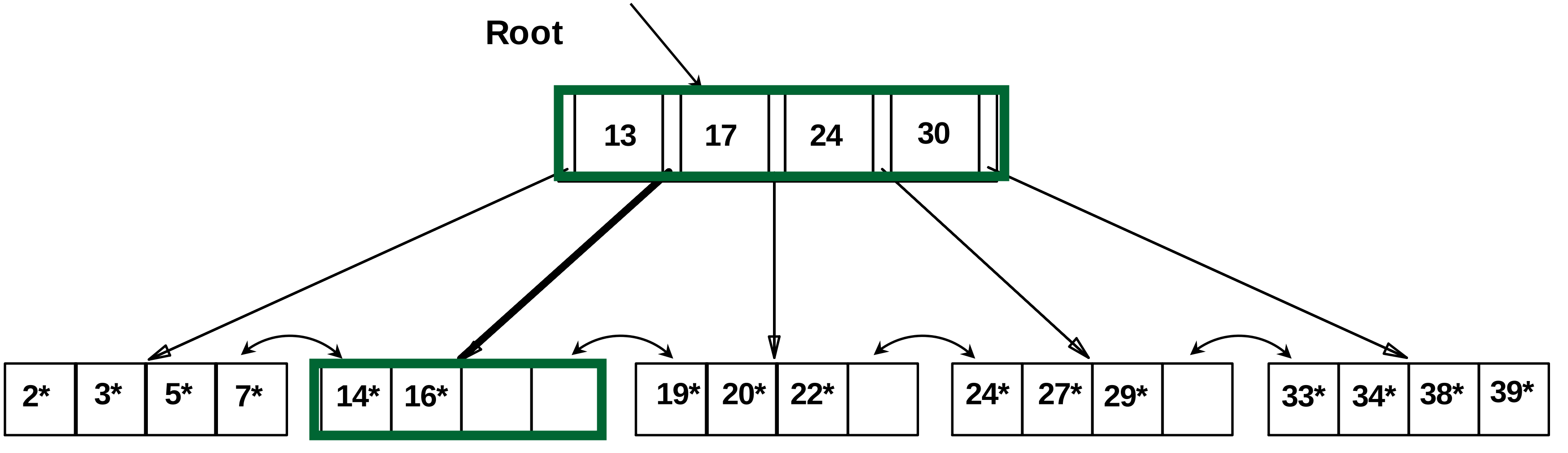
(Direct search)

Data Entries

("Sequence set")

B+ Tree Equality Search

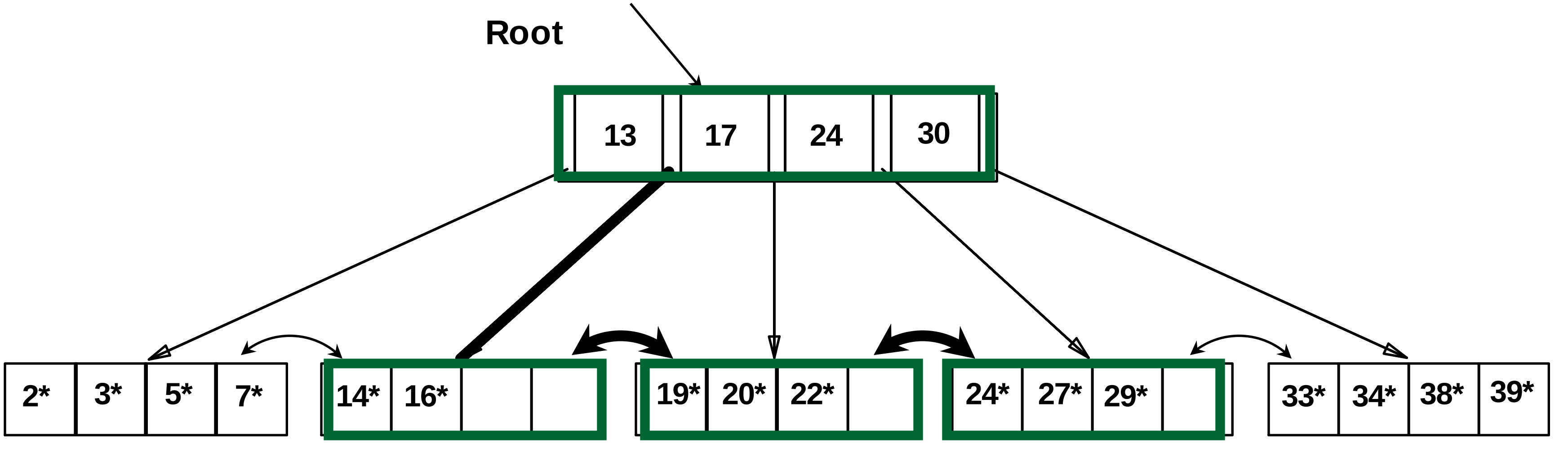
- Search begins at root, and key comparisons direct it to a leaf.
- Search for 15*…



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

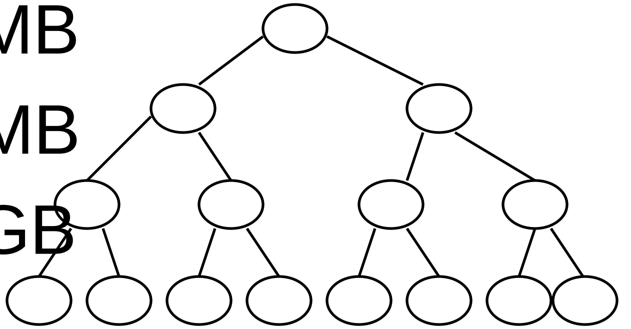
B+ Tree Range Search

- Search all records whose ages are in [15,28].
  - Equality search 15*.
  - Follow sibling pointers.



B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 KB
  - Level 2 = 133 pages = 1 MB
  - Level 3 = 17,689 pages = 145 MB
  - Level 4 = 2,352,637 pages = 19 GB

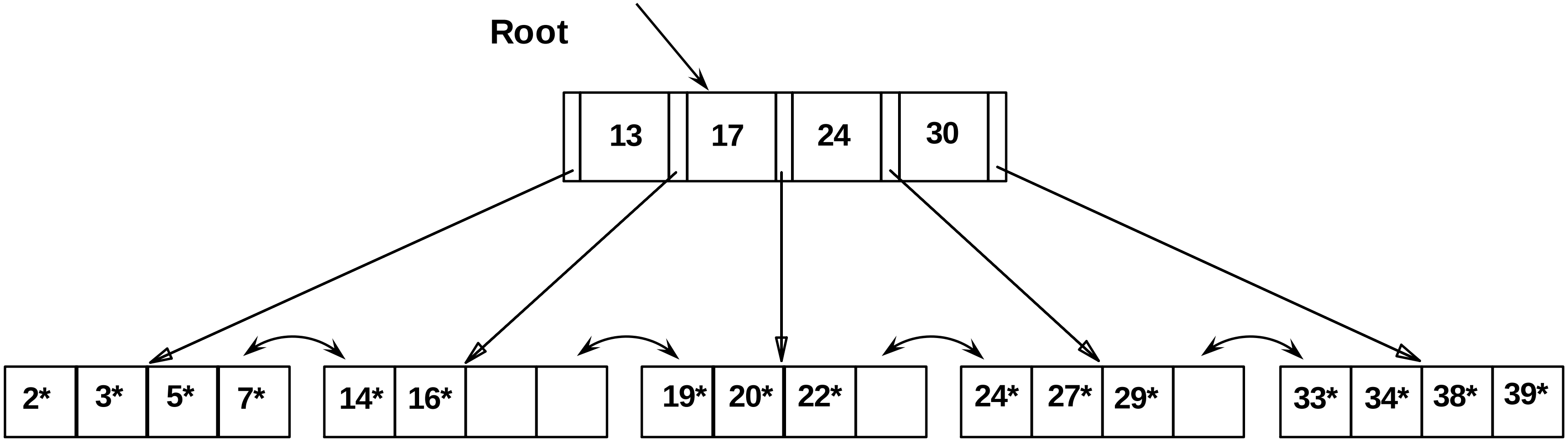


- With 1 MB buffer, can locate one record in 19 GB (or 0.3 billion records) in two I/Os!

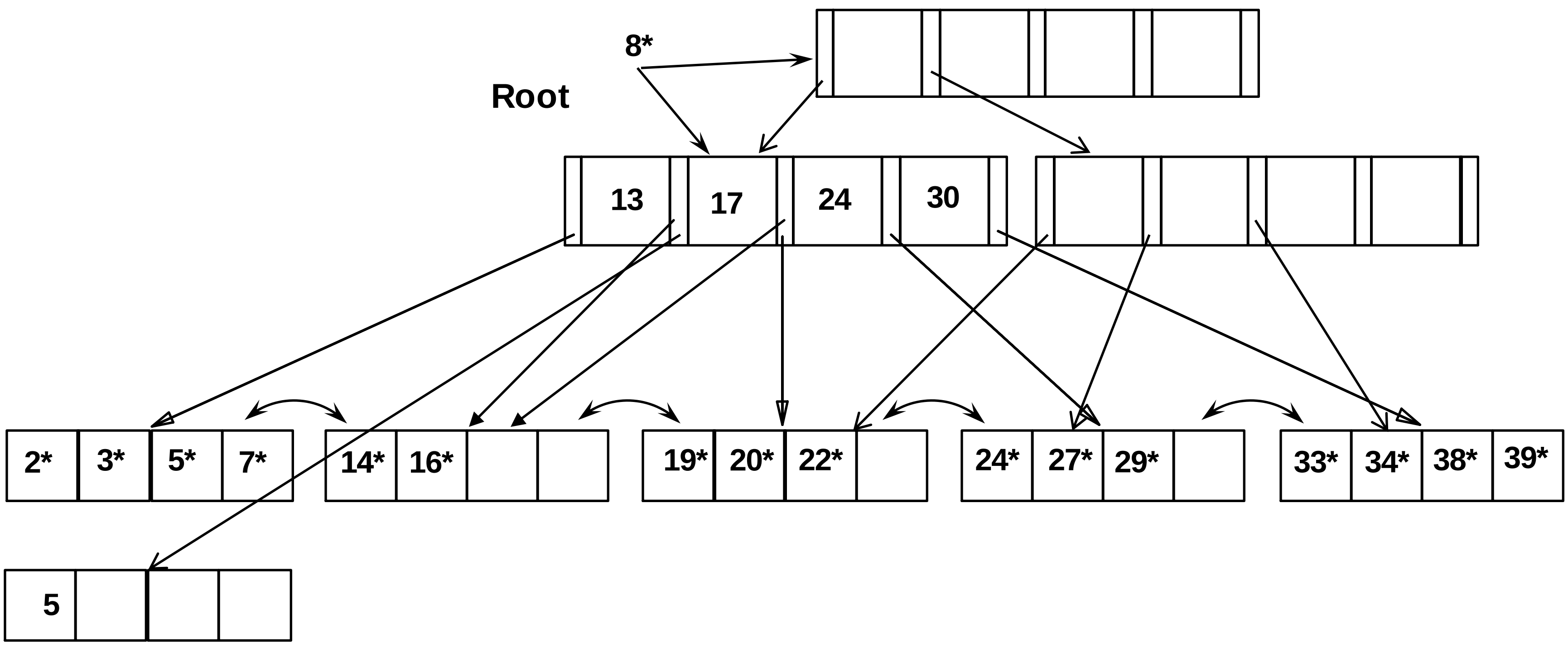
Inserting a Data Entry into a B+ Tree

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

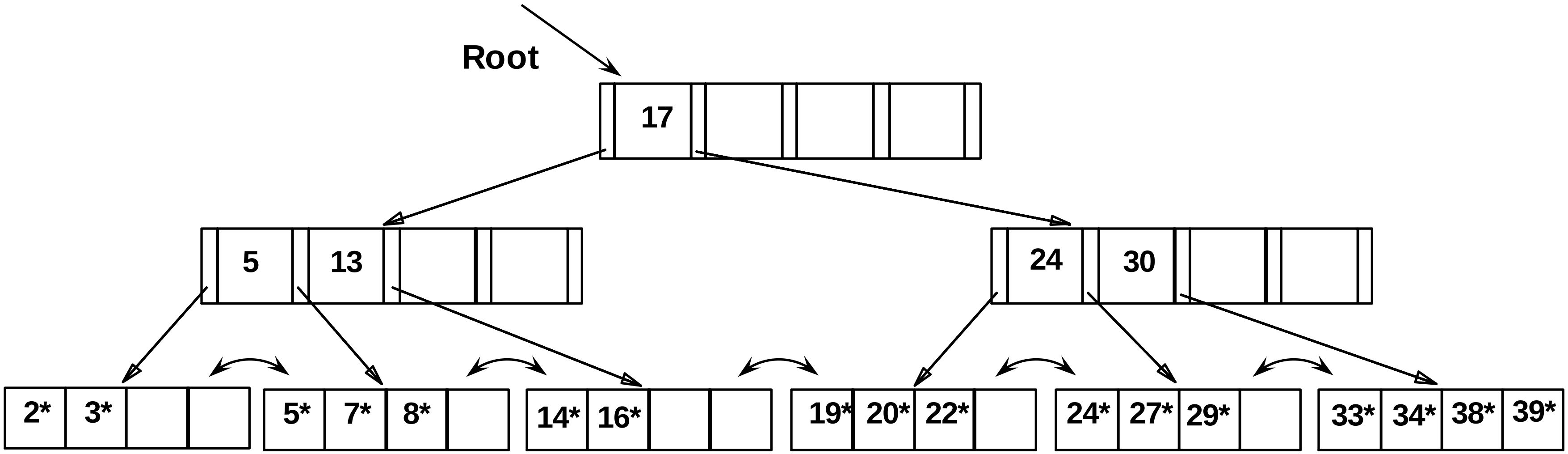
Example B+ Tree – Inserting 8*



Animation: Insert 8*



Final B+ Tree - Inserting 8*

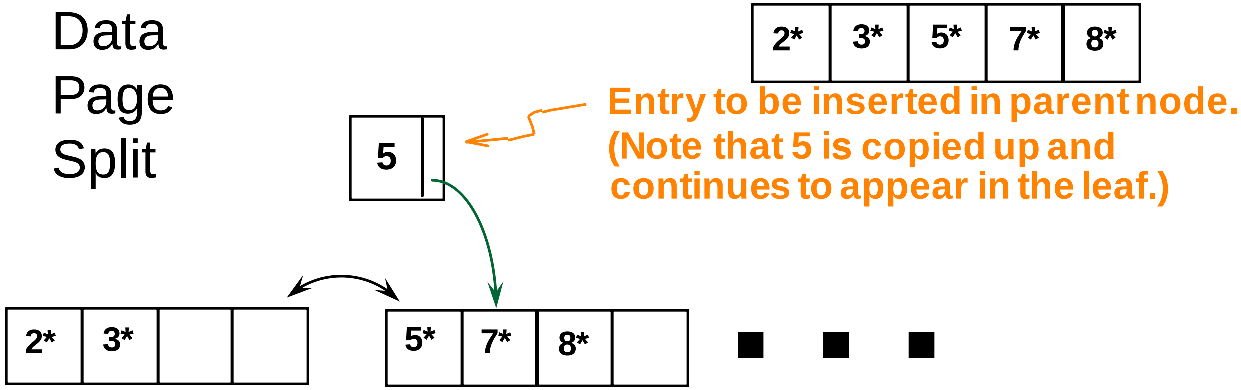


- Notice that root was split, leading to increase in height.
- In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

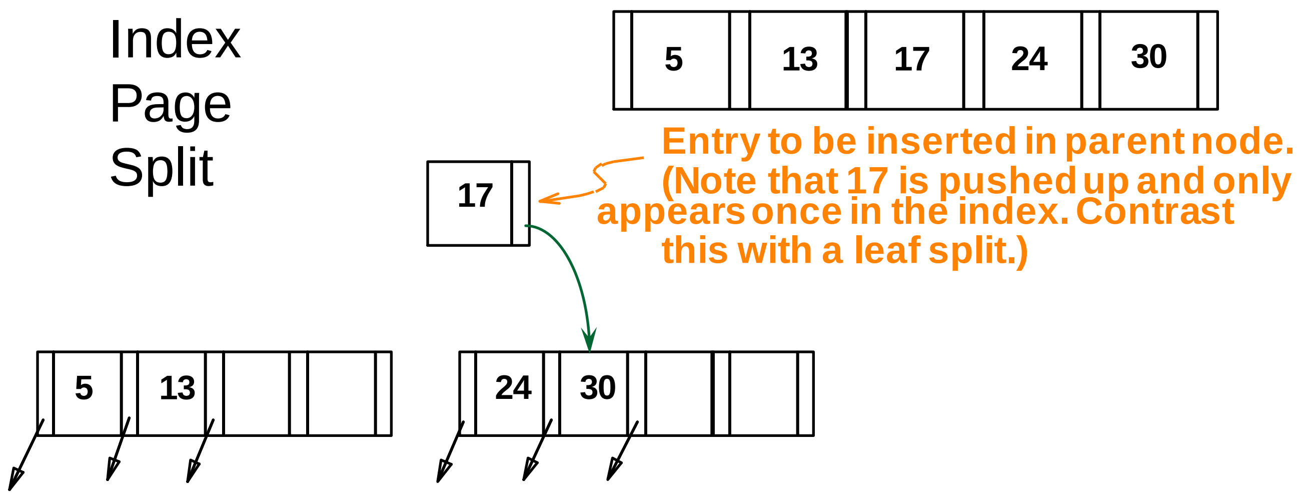
Data vs. Index Page Split (from previous example of inserting "8*")

- Observe how 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.

Data Page Split



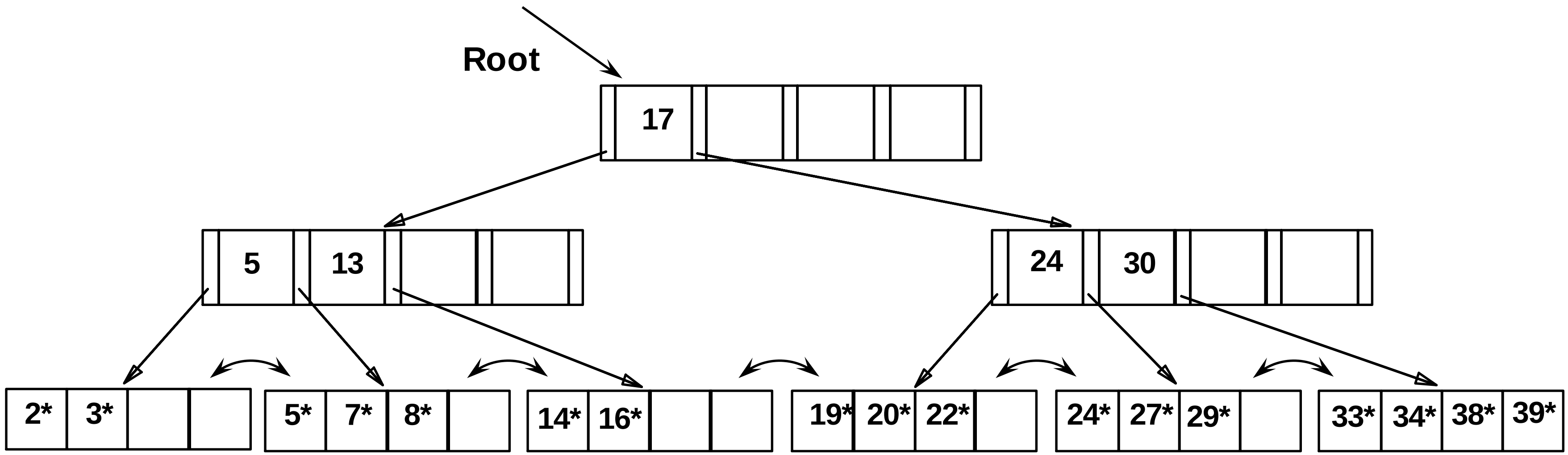
Index Page Split



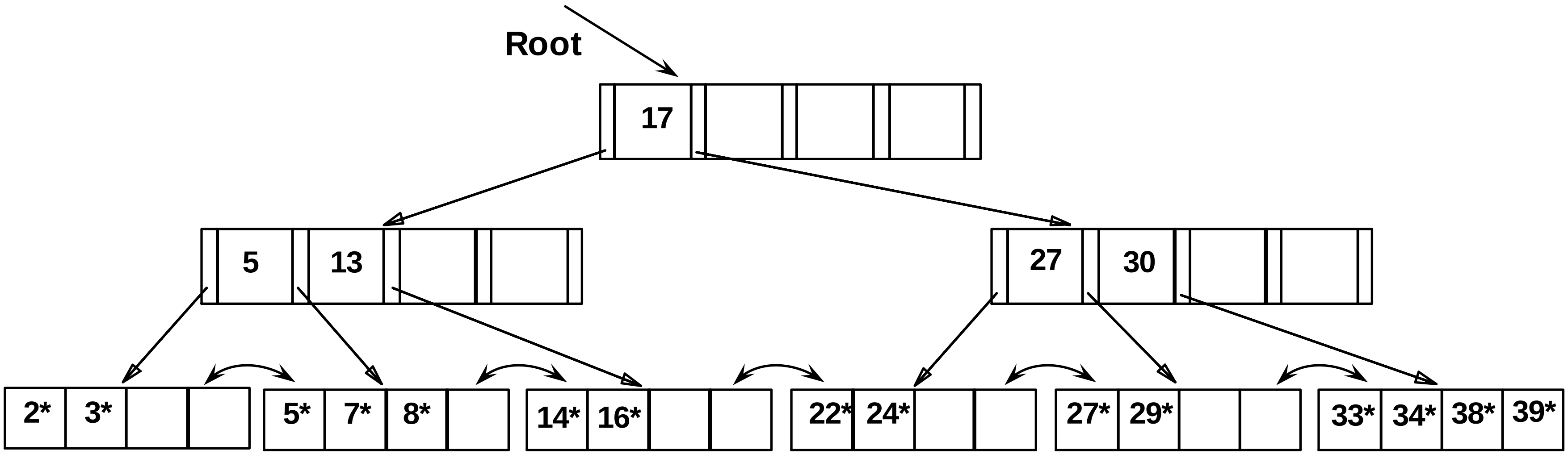
Deleting a Data Entry from a B+ Tree

- 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)*.
    - 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.

Example Tree (including 8*) Delete 19* and 20* ...



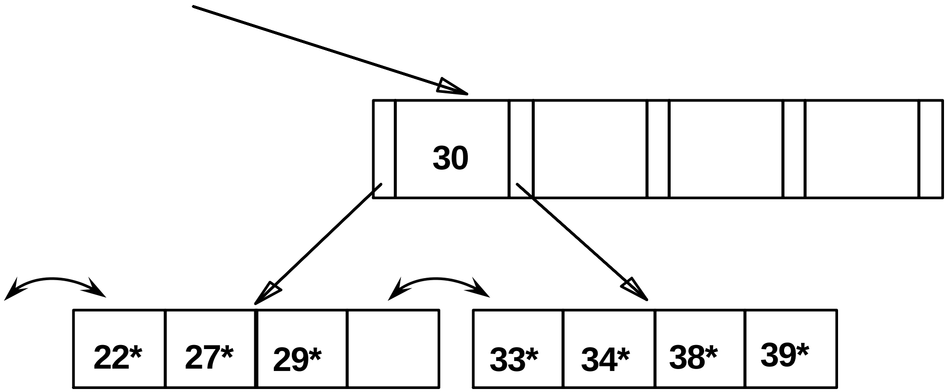
Example Tree (including 8*) Delete 19* and 20* ...

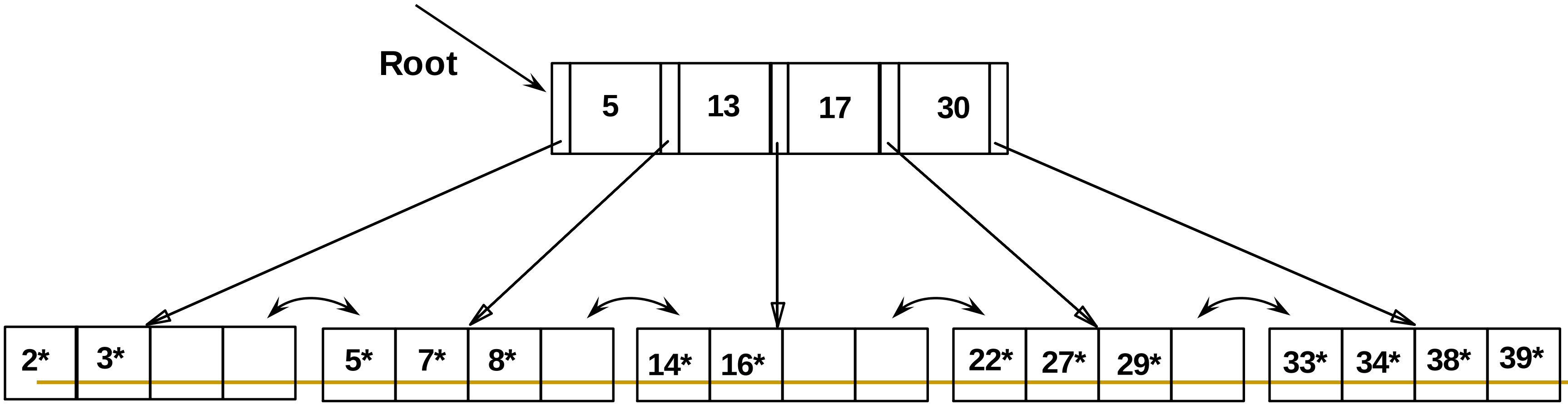


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

... And Then Deleting 24*

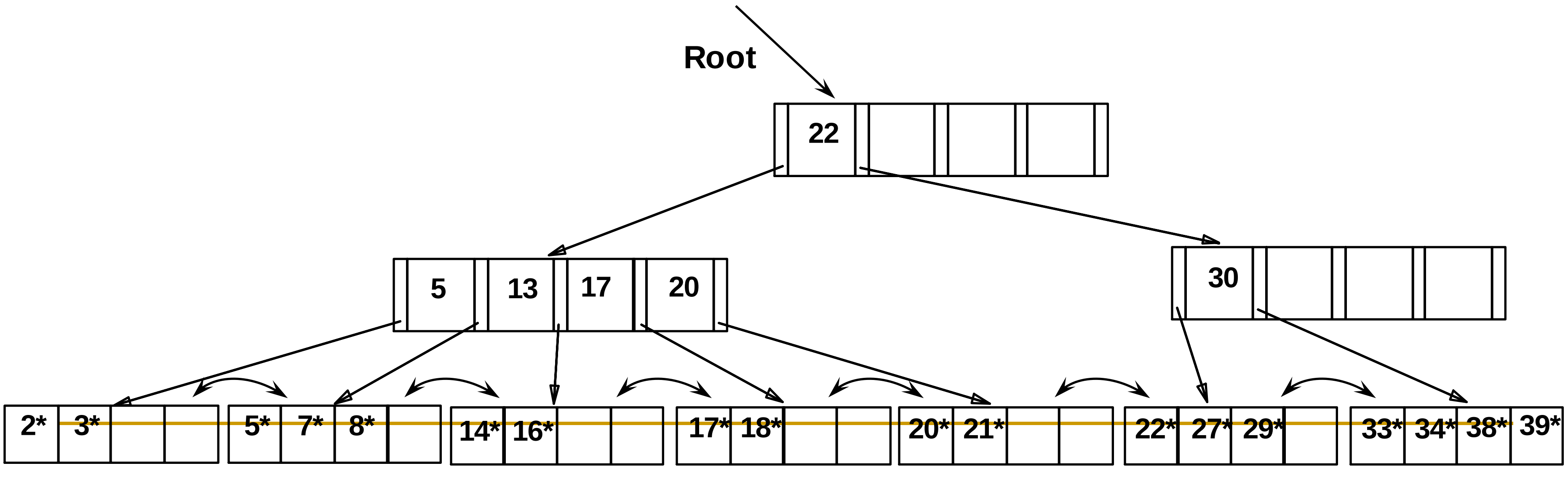
- Must merge.
- Observe *`toss'* of index entry (on right), and *`pull down'* of index entry (below).





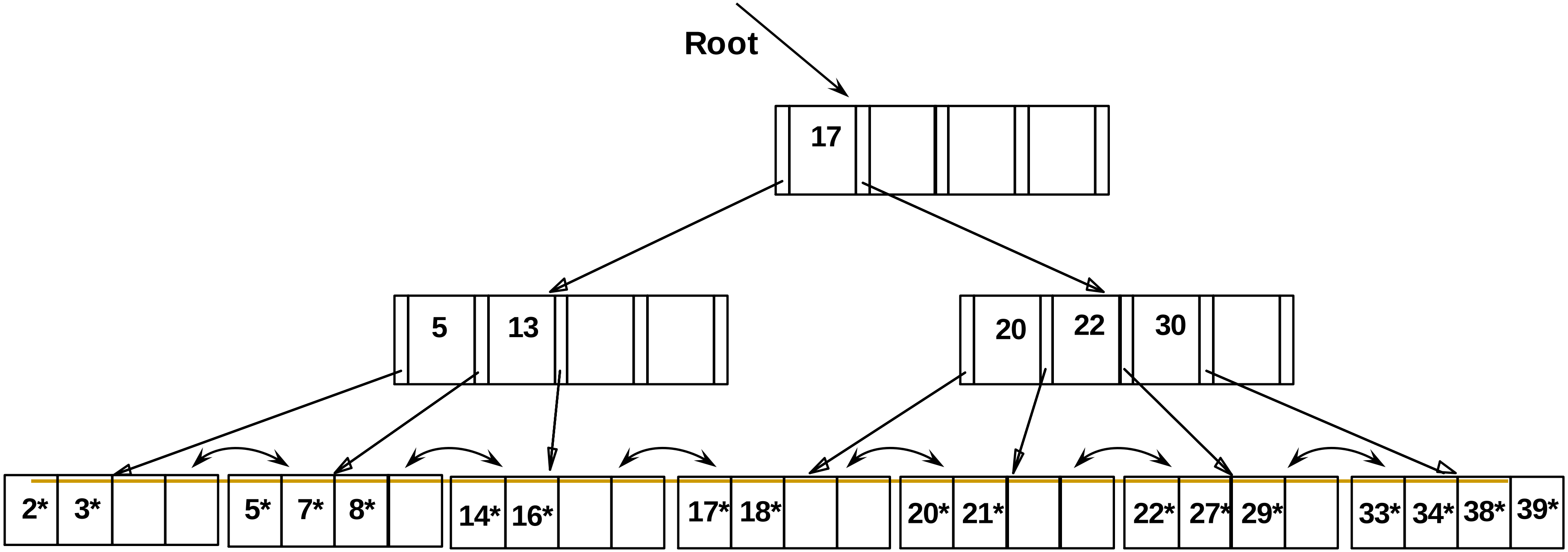
Example of Non-leaf Re-distribution

- 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.



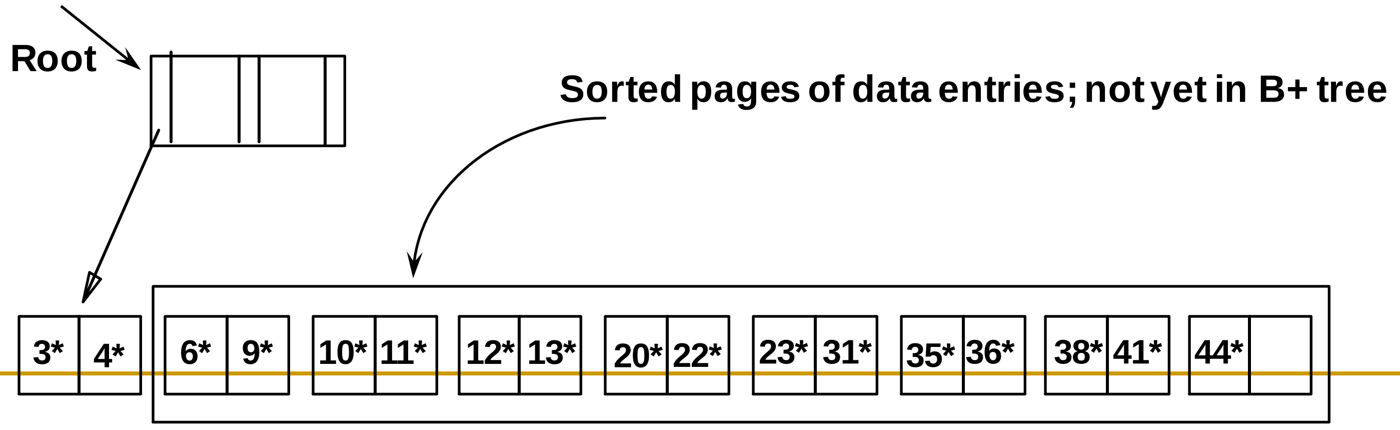
After Re-distribution

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



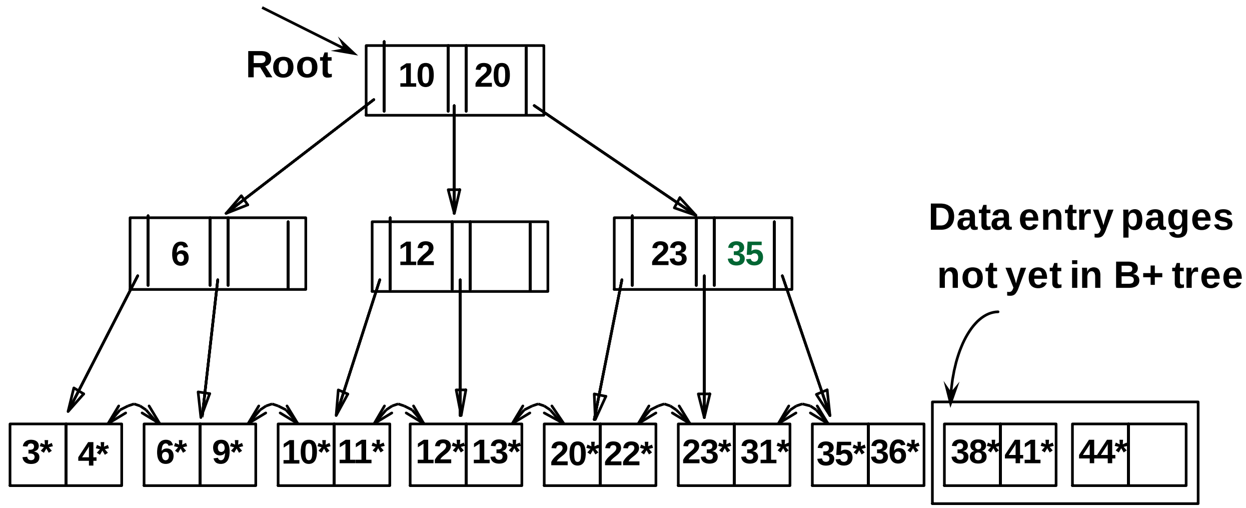
Bulk Loading of a B+ Tree

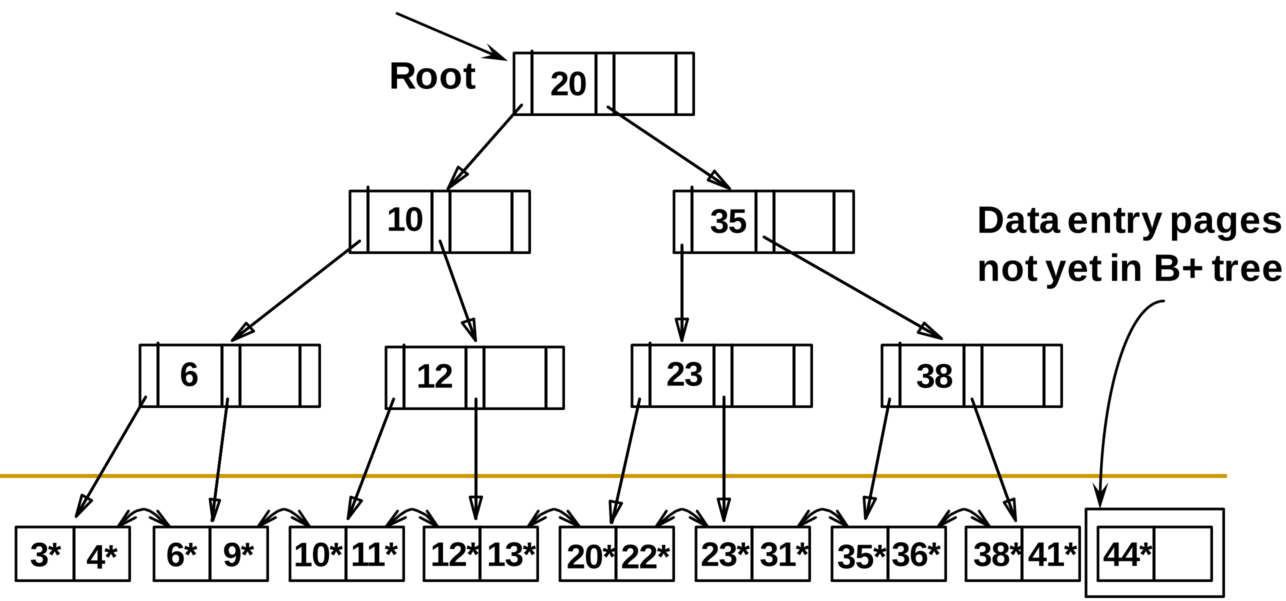
- Given: large collection of records
- Desire: B+ tree on some field
- Bad idea: repeatedly insert records
  - Slow, and poor leaf space utilization . Why?
- *Bulk Loading* can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



Bulk Loading (Contd.)

- 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.





Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.
- Option 2: *Bulk Loading*
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course).
  - Can control “fill factor” on pages.

A Note on `Order'

- *Order* (**d**) makes little sense with variable-length entries
- Use a physical criterion in practice (`*at least half-full*').
  - Index pages often hold many more entries than leaf pages.
  - Variable sized records and search keys:
    - different nodes have different numbers of entries.
  - Even with fixed length fields, Alternative (3) gives variable length
- Many real systems are even sloppier than this --- only reclaim space when a page is *completely* empty.

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 rarely more than 3 or 4.
  - Typically, 67% occupancy on average.
  - Usually preferable to ISAM; adjusts to growth gracefully.
  - If data entries are data records, splits can change rids!

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.
- B+ tree widely used because of its versatility.
  - One of the most optimized components of a DBMS.