Indexing

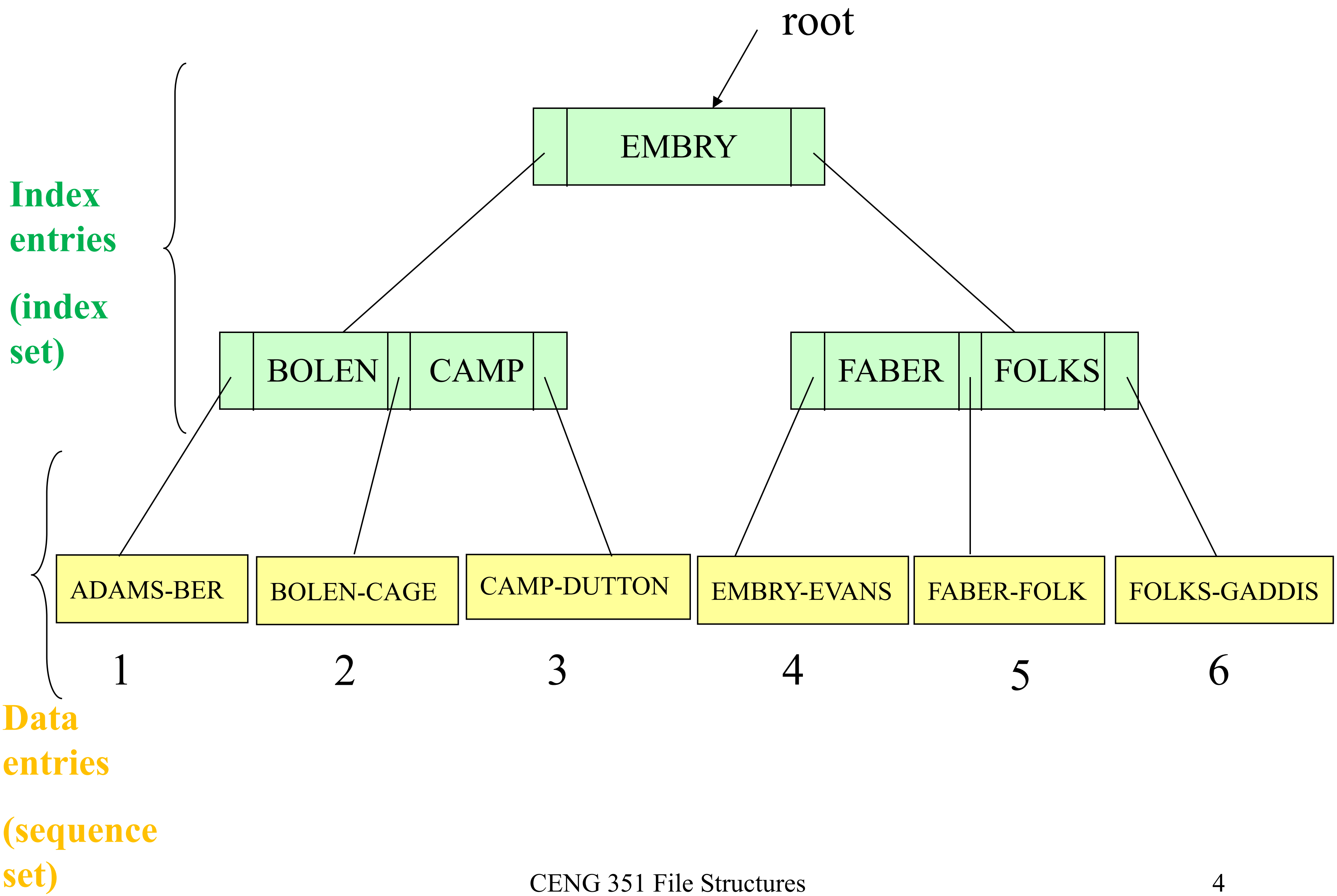
Part II

Tree indexes

- If index doesn't fit in memory:
  - Divide the index structure into blocks,
  - Organize these blocks similarly building a tree structure.
- Tree indexes:
  - B Trees
  - B+ Trees
  - Simple prefix B+ Trees
  - …

B+ Trees

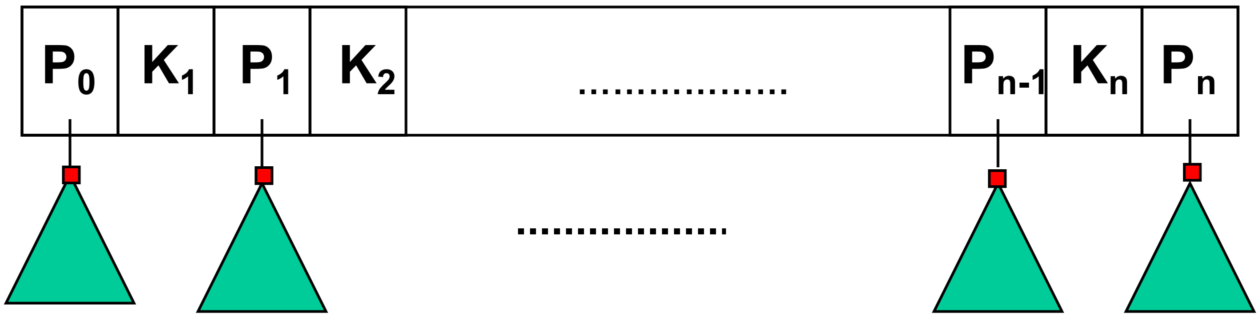
- B-tree is one of the most important data structures in computer science.
- What does B stand for? (Not binary!)
- B-tree is a multiway search tree.
- Several versions of B-trees have been proposed, but only B+ Trees have been used with large files.
- A B+tree is a B-tree in which data records are in leaf nodes, and faster sequential access is possible.


root
EMBRY
BOLEN
CAMP
FABER
FOLKS
ADAMS-BER
BOLEN-CAGE
CAMP-DUTTON
EMBRY-EVANS
FABER-FOLK
FOLKS-GADDIS
1
2
3
4
5
6
Index entries
(index set)
Data entries
(sequence set)

Formal definition of B+ Tree Properties

- Properties of a B+ Tree of order d:
  - All internal nodes (except root) have **at least d keys** and **at most 2d** keys .
  - The root has at least 2 children unless it's a leaf.
  - All leaves are on the same level.
  - An internal node with k keys has k+1 children

B+ tree: Internal/root node structure



Each P_i is a pointer to a child node; each K_i is a search key value
# of search key values = n, # of pointers = n+1

- **Requirements:**
  - **$K_1 < K_2 < \ldots < K_n$**
  - **For any search key value K in the subtree pointed by Pi,**
    If Pi = P0, we require K < K1
    If Pi = Pn, Kn <= K
    If Pi = P1, …, Pn-1, Ki <= K < Ki+1

B+ tree: leaf node structure

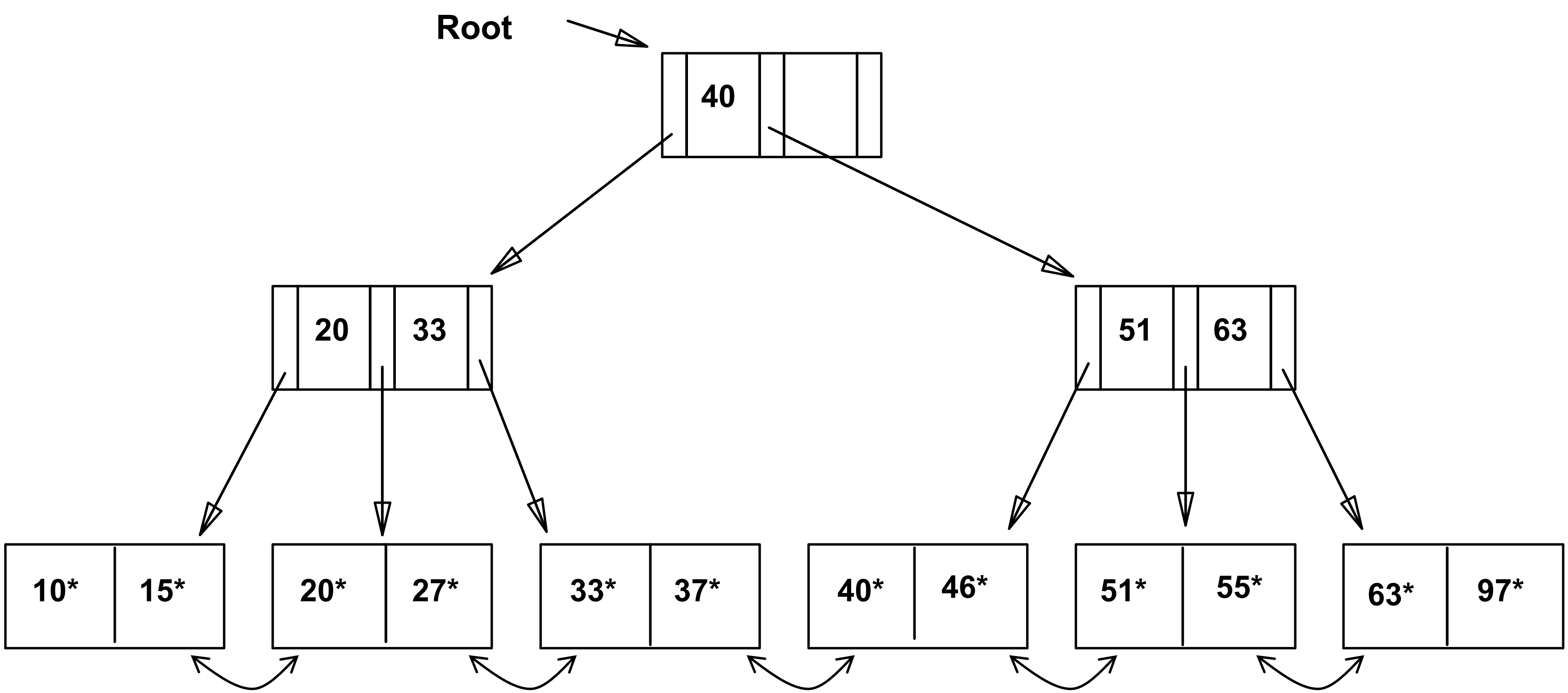
L	K_1 r_1	K_2	K_n r_n	R

....

- Pointer L points to the left neighbor; R points to the right neighbor
- $K_1 < K_2 < \ldots < K_n$
- d <=n <= 2d (d is the order of this B+ tree)
- We will use K_i* for the pair $<K_i, r_i>$ and omit L and R for simplicity

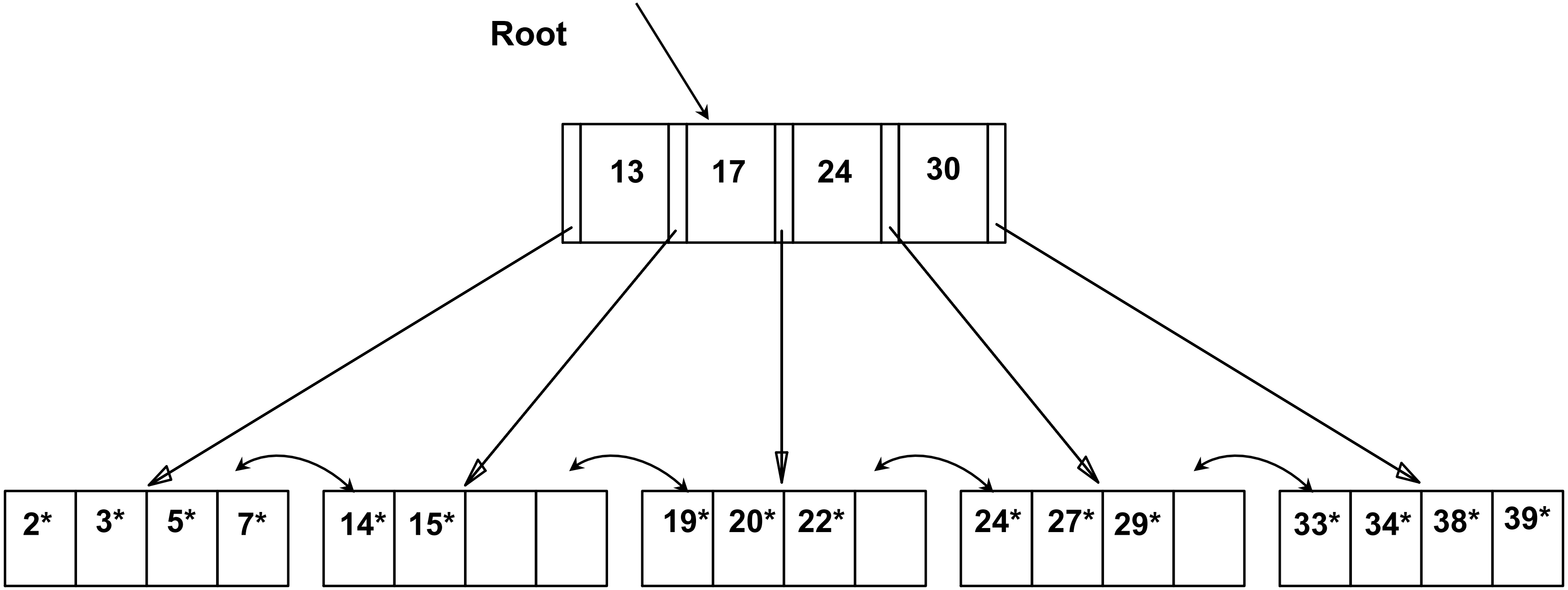
Example: B+ tree with order of 1

- Each node must hold at least 1 entry, and at most 2 entries



Example: Search in a B+ tree order 2

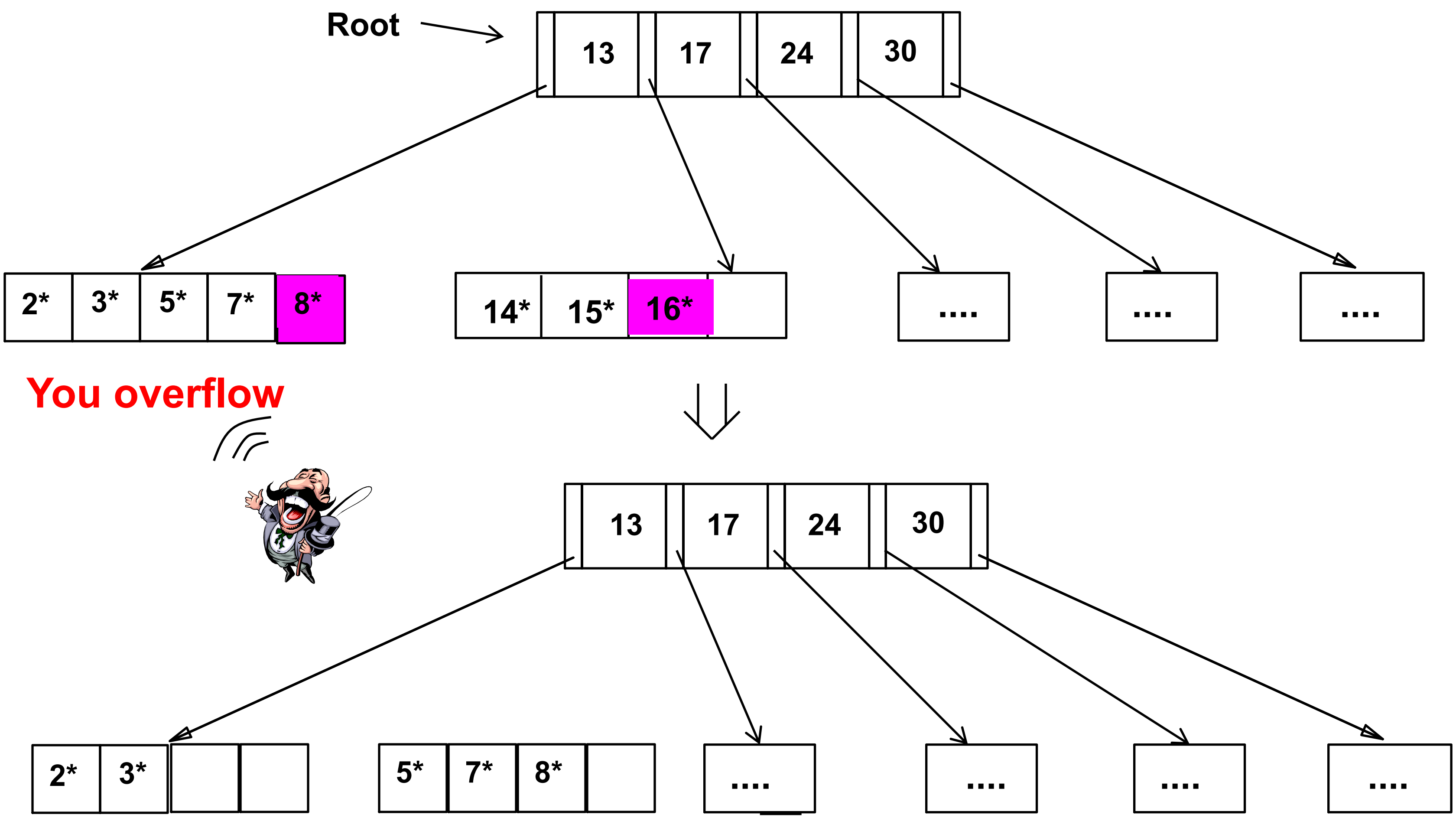
- Search: how to find the records with a given search key value?
  - Begin at root, and use key comparisons to go to leaf
- Examples: search for 5*, 16*, all data entries >= 24* ...
  - The last one is a range search, we need to do the sequential scan, starting from the first leaf containing a value >= 24.



How to Insert a Data Entry into a B+ Tree?

- Let’s look at several examples first.

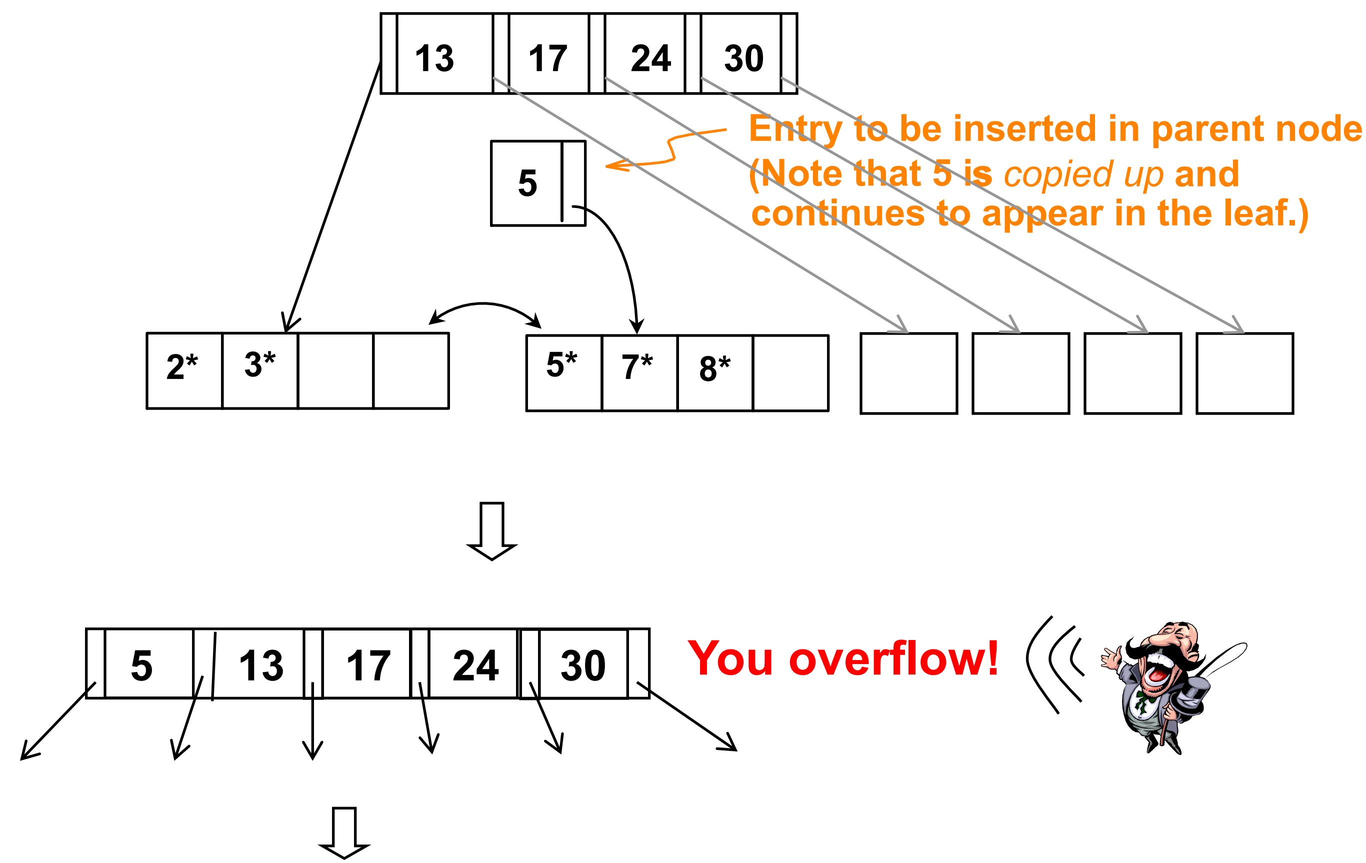
Inserting 16*, 8* into Example B+ tree



One new child (leaf node) generated; must add one more pointer to its parent, thus one more key value as well.

Inserting 8* (cont.)

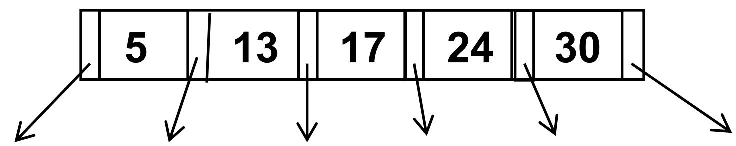
- Copy up the middle value (leaf split)



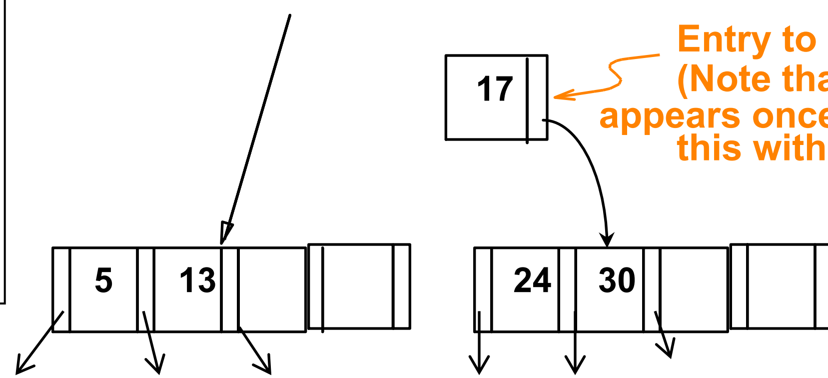
Insertion into B+ tree (cont.)

- Understand difference between copy-up and push-up

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.

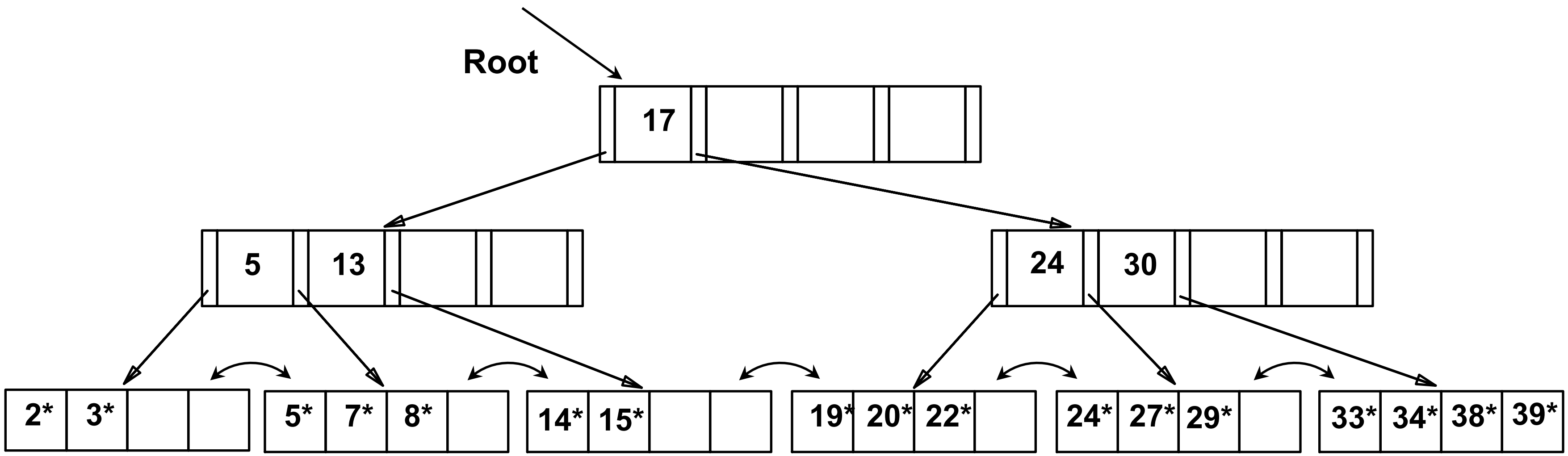


We split this node, redistribute entries evenly, and push up middle key.



Entry to be inserted in parent node. (Note that 17 is pushed up and only appears once in the index. Contrast this with a leaf split.)

Example B+ Tree After Inserting 8*



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

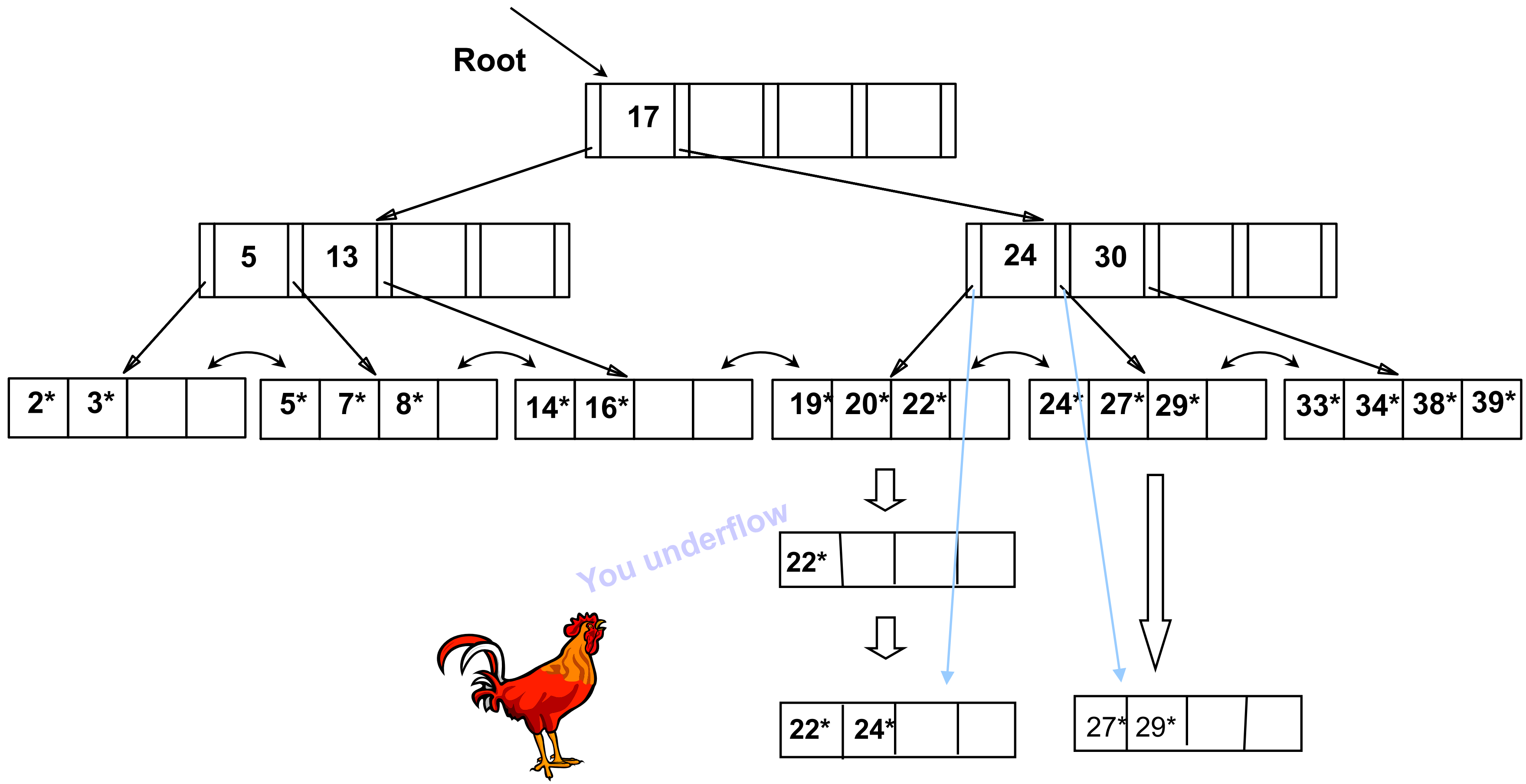
Inserting a Data Entry into a B+ Tree: Summary

- 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, put middle key in L2
    - **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.*

Deleting a Data Entry from a B+ Tree

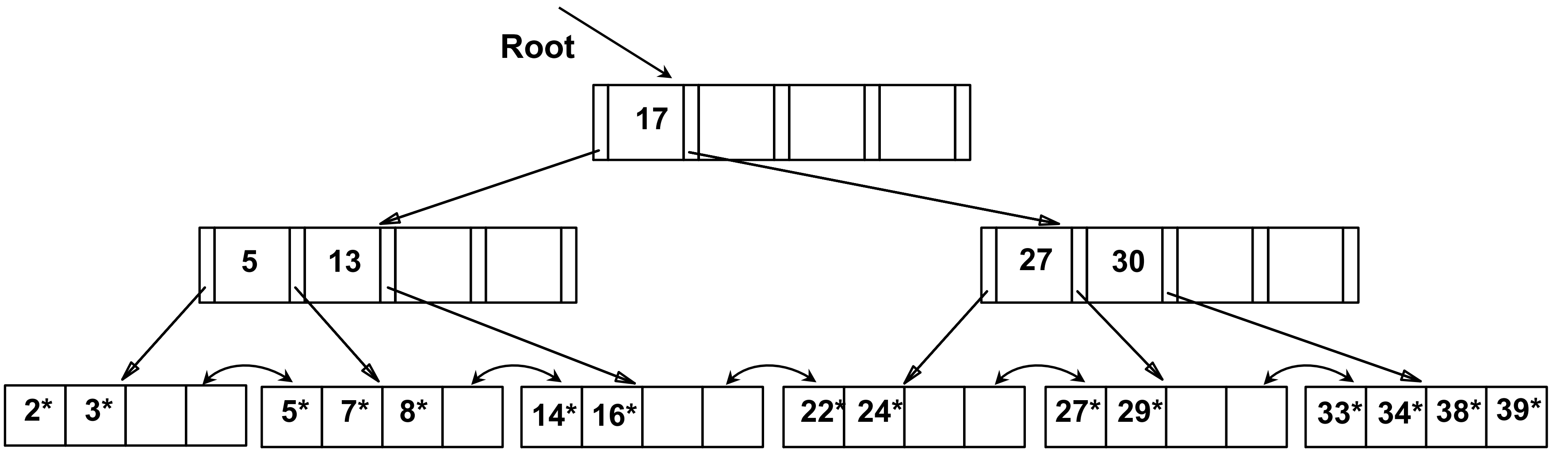
- Examine examples first …

Delete 19* and 20*

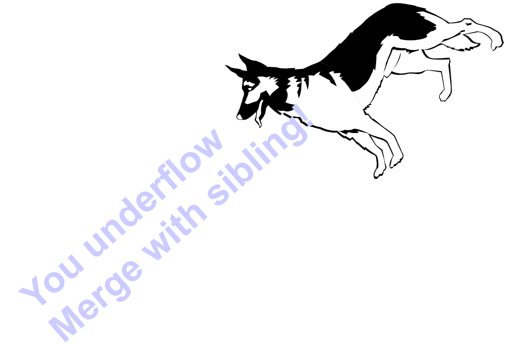


Have we still forgotten something?

Deleting 19* and 20* (cont.)

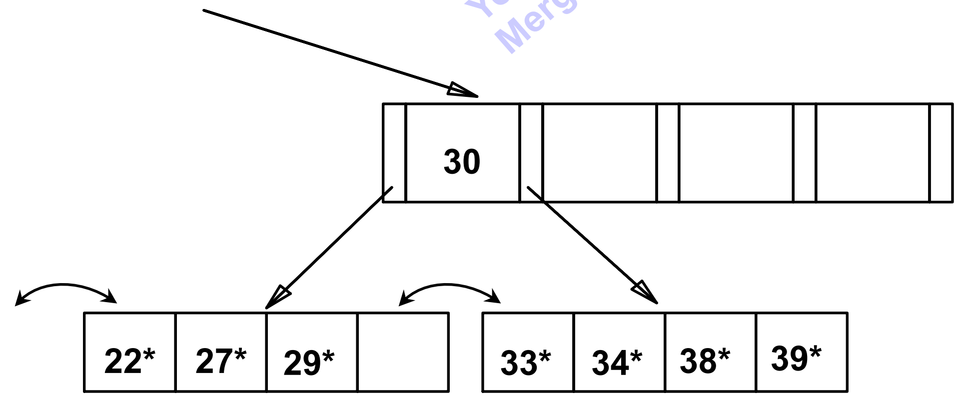


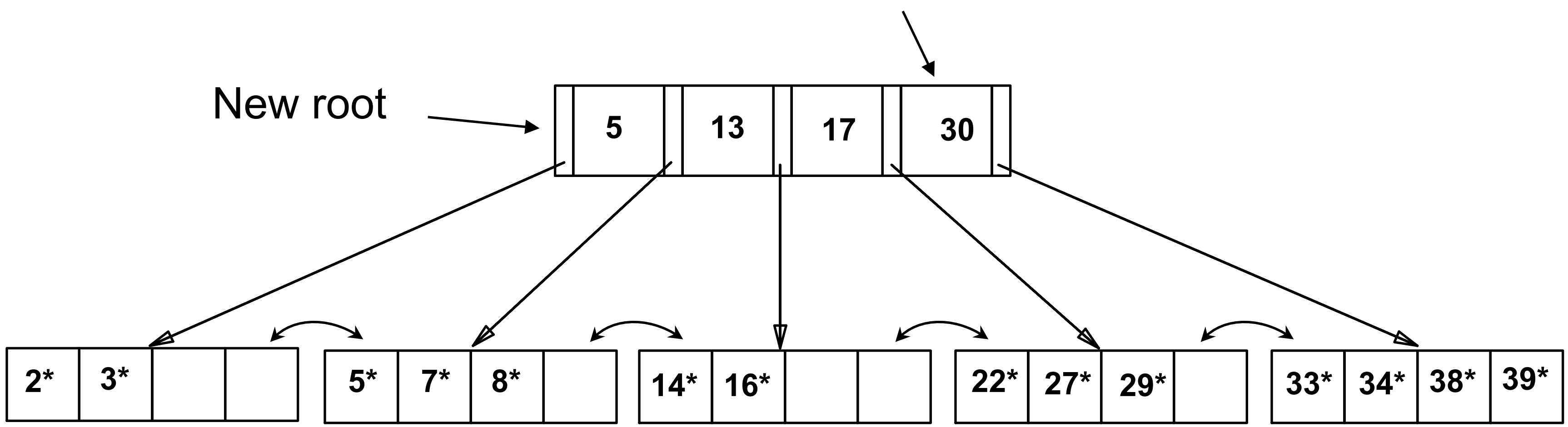
- Notice how 27 is *copied up*.
- But can we move it up?
- Now we want to delete 24
- Underflow again! But can we redistribute this time?



Deleting 24*

- Observe the two leaf nodes are merged, and 27 is discarded from their parent, but …
- Observe *`pull down'* of index entry (below).



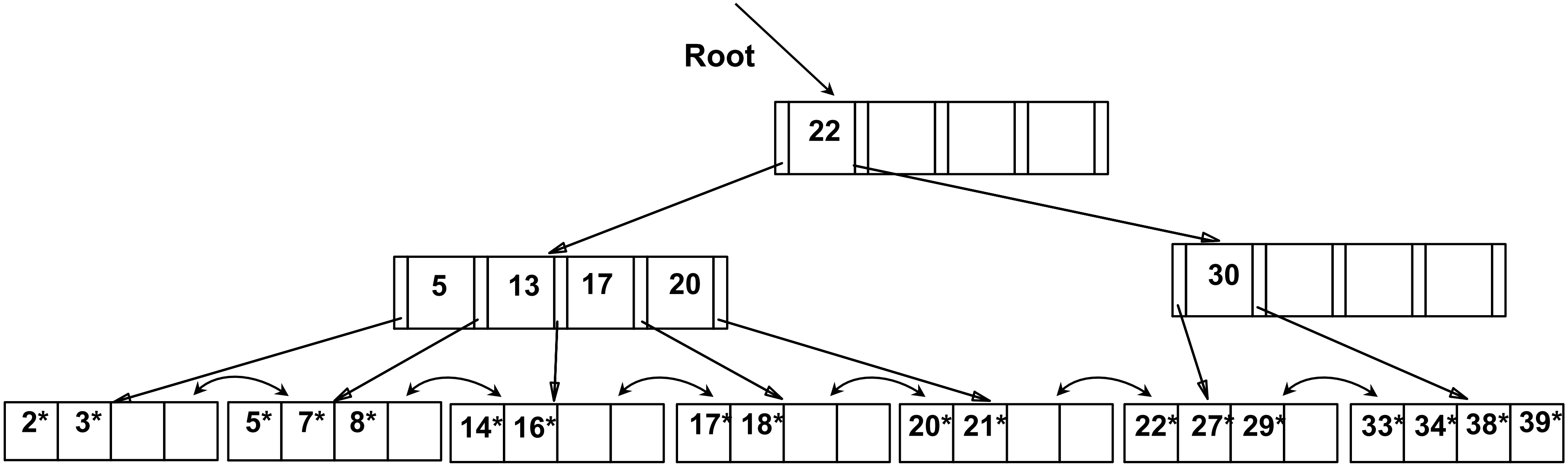


Deleting a Data Entry from a B+ Tree: Summary

- 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.

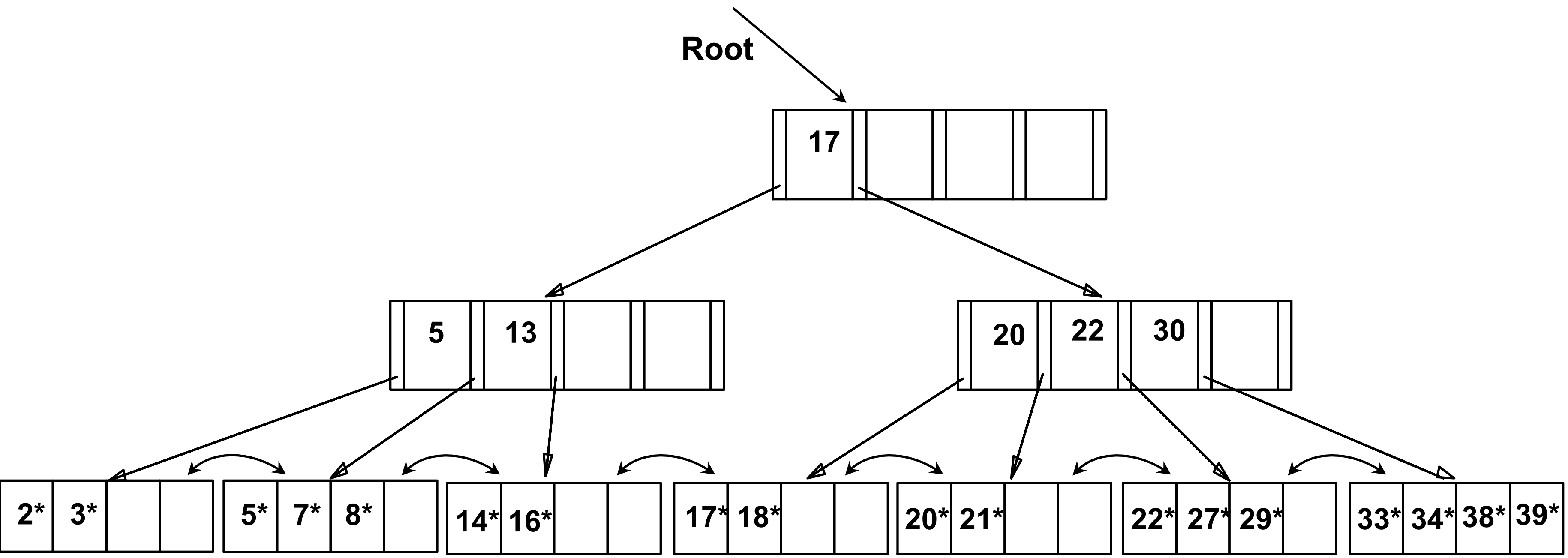
Example of Non-leaf Re-distribution

- 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

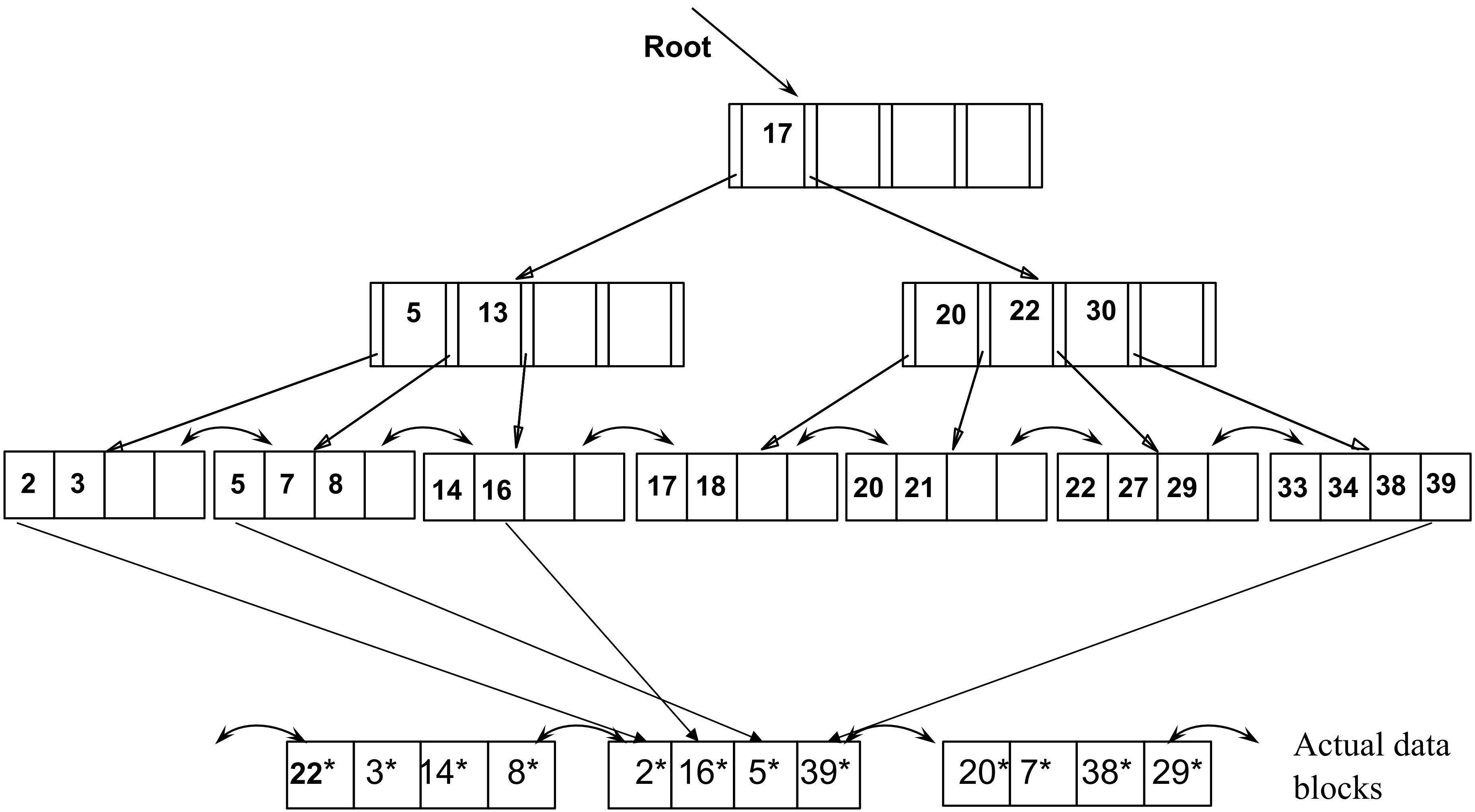
- Intuitively, entries are re-distributed by *`pushing through'* the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.



Primary vs Secondary Index

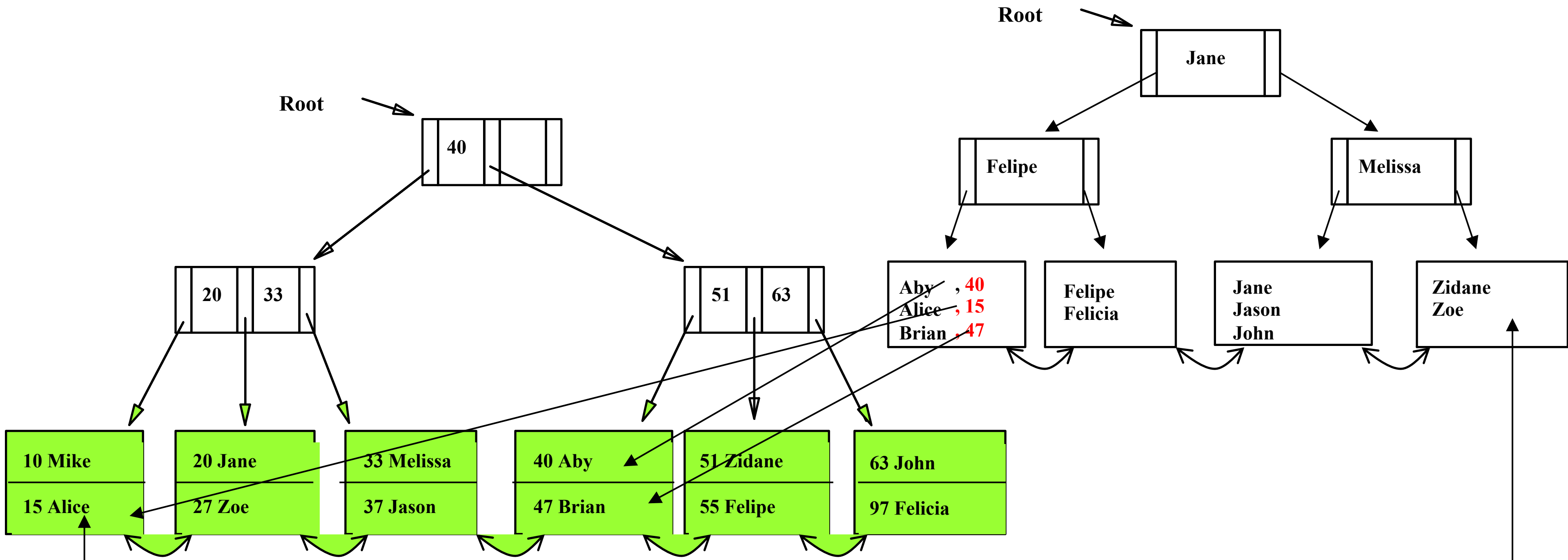
- Note: We were assuming the data items were in sorted order
  - This is called *primary/clustered B+tree* index
- *Secondary B+tree* index:
  - Built on an attribute that the file is not sorted on.
- Can have many different indexes on the same file.

A Secondary B+-Tree index



A file organized as (or, has) a **Primary B+-Tree** index on *ssn*

The same file also has a **Secondary B+-Tree** index on *name*



•As **15***, we store the **actual data record** with key value 15 (**Alternative-1**)

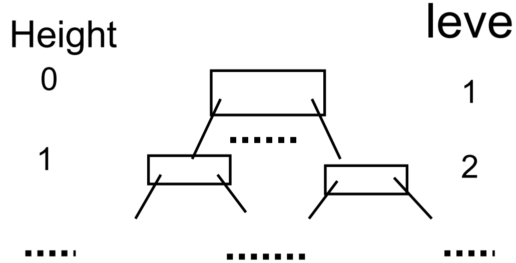
• In this case, the leaf nodes can be larger, say a few blocks (pages), called **buckets**

•We have k* as <key, rid> (**Alternative 2**)

•We can have rid's as pointers, or use PK as rid. We show both above

Cost for searching a value in B+ tree

- Assumptions:
  - Each interior node is a disk block
  - Each leaf node is also a disk block and data entries (K*) are of the form <key, ptr>. There are D data entries.
  - Let F be the average number of pointers in a node (for internal nodes, it is called *fanout,* **i.e.,** avg. number of children)
- Observe: Let H be the height of the B+ tree: we need to read H+1 nodes (blocks) to reach a data entry in a leaf node
- How do we find H?
  - Level 1 = 1 page = F^0 page
  - Level 2 = F pages = F^1 pages
  - Level 3 = F * F pages = F^2 pages
  - Level H+1 = …….. = F^H pages (i.e., leaf nodes)
  - F pointers → F-1 keys, so there must be D/(F-1) leaf nodes
  - D/(F-1) = F^H. That is, $H = \log_F(\frac{D}{F-1})$



B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 66%.
  - average fanout = 133 (i.e, # of pointers in internal node)
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes
- Suppose there are 1,000,000,000 data entries.
  - $H = \log_{133}(1000000000/132) < 4$
  - The cost is 5 pages read

Cost Computation: Another Example

Leaves would store the **actual records**

- A <u>primary B+ tree index</u> on key field giftID.
- 2.500.000 gift records, each record: 400 bytes.
- giftID: 12 bytes, address pointer: 4 bytes
- A <u>bucket</u> can hold 500 records
  - So we have larger leaf nodes (called **buckets**), as we store actual records
  - No claim for interior nodes, assume each is a block!
- B+ tree will have a fill factor of 50% [min occupancy]
- B (block size): 1600
- s: 10 ms, r: 5 ms, btt: 1 ms.

a) No of index nodes and their total size

We need to find i) fanout of the nodes, and ii) no of leaves.

i) fanout: Assume , **n keys (n+1) ptrs** can fit to an index node:

n x 12 + (n+1) x 4 = 1600 bytes → 16n = 1596 / 16 → n= 99

So at most 99 keys in a node (2d= 99, d (tree order) is floor(99/2))

Tree fill factor 50%; max 99 keys x 50% = 49 keys at most

fanout: 49 + 1 = 50 ptrs per node

ii) no of leaves:

500 rec/leaf * fill factor (50%) = 250 recs/leaf

2.5M records / 250 = 10000 leaf nodes (i.e., buckets)

a) No of index nodes and their total size

- Tree height = $\log_{50} 10000 = 3$
- So, there are H+1 = 4 levels

Level 4: 10000 leaf nodes (data buckets)

Level 3: ceil (10000 / 50 ptrs) = 200 nodes

Level 2: ceil (200/50) = 4 nodes

Level 1: ceil (4/50) = 1 node (root)

Index nodes: 1 + 4 + 200 = 205

Total Size: 205 x 1600 bytes

b) Time cost of reading an arbitrary record

- Three has H=3, so 4 levels
- At the first 3 levels, we fetch index nodes:

3 x (s + r+ btt) = 3x (10 +5+1) = 48 ms

- At the fourth level we fetch the leaf node (data bucket)
  - But how many blocks is a data bucket?
  - (500 recs x 400 bytes/rec) / 1600 = 125 blocks
  - So, cost s + r+ 125 x btt = 10+ 5+ 125 x 1 = 140 ms
- Total cost: 48 + 140 = 188 ms

c) Cost of reading all records in sorted manner

- Reach to leftmost leaf node, as before:
- at the first 3 levels, we fetch index nodes:

3 x (s + r+ btt) = 3x (10 +5+1) = 48 ms

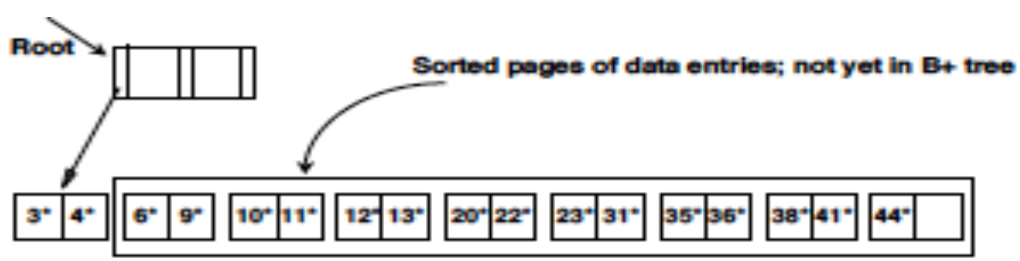
- Read all the leaf nodes (using doubly linked list pointers)
  - 10000 (s + r + 125 x btt)
- Think: What if this is a secondary B+ tree and we store <key, ptr> pairs at leaf nodes (data buckets)?

Terminology

- **Blocking Factor:** the number of records which can fit in a leaf node.
- **Fan-out :** the average number of children of an internal node.
- A B+tree index can be used either as a primary index or a secondary index.
  - **Primary index:** determines the way the records are actually stored (also called a sparse index, clustered index)
  - **Secondary index:** the records in the file are not grouped in blocks according to keys of secondary indexes (also called a dense index)

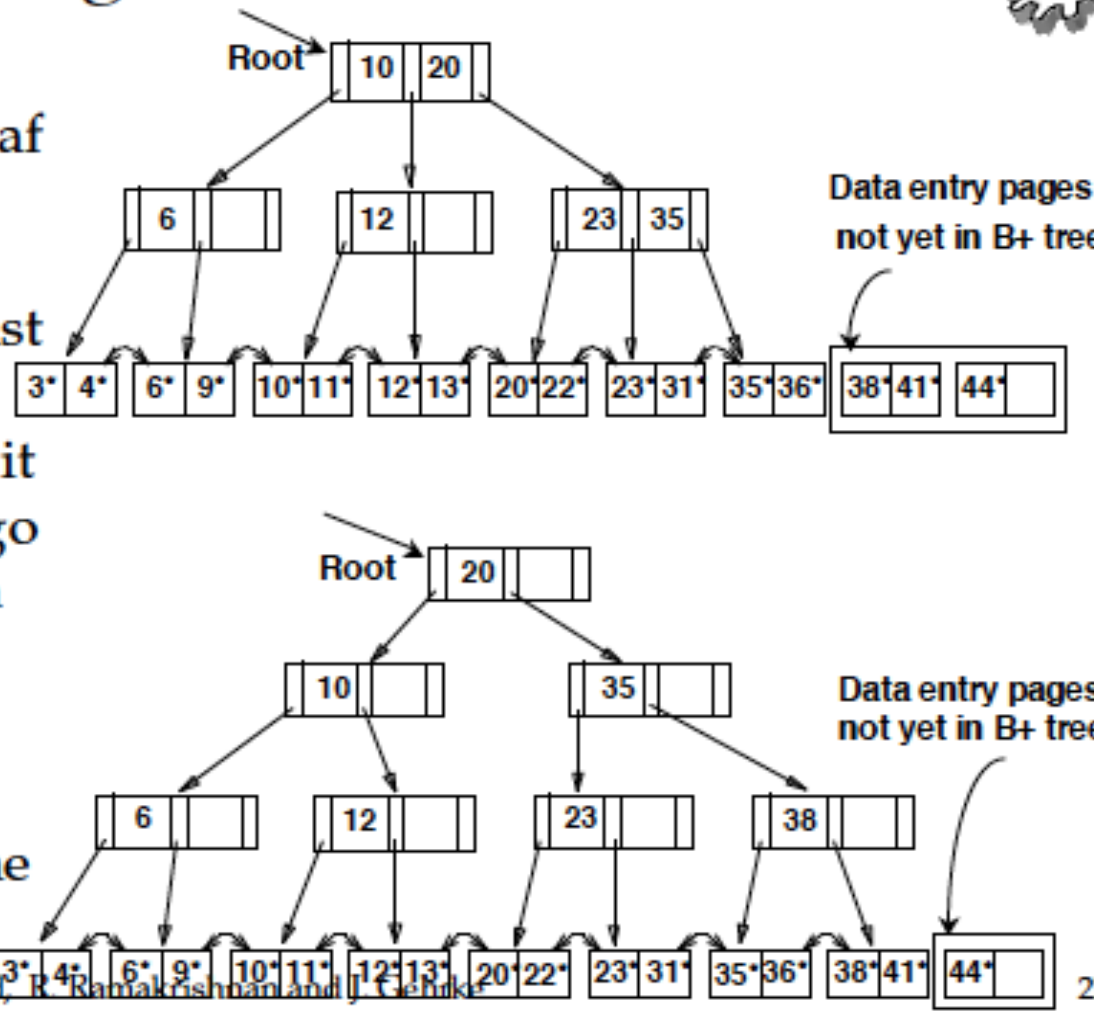
Bulk Loading of a B+ Tree

- 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.)

- 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!



Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke 22

Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced; High fanout (**F**) means depth rarely more than 3 or 4.
  - Almost always better than maintaining a sorted file.
  - Typically, 67% occupancy on average.
  - If data entries are data records, splits can change rids!
- Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.

More…

- Hash-based Indexes
  - Static Hashing
  - Extendible Hashing
  - Linear Hashing
- Grid-files
- R-Trees
- etc…
- A nice animation site for B+ trees: https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html