Tree-Structured Indexes

Chapter 10

WORLD of COW
By Stik

Define "normal"!

© Original Artist
Reproduction rights obtainable from
www.CartoonStock.com
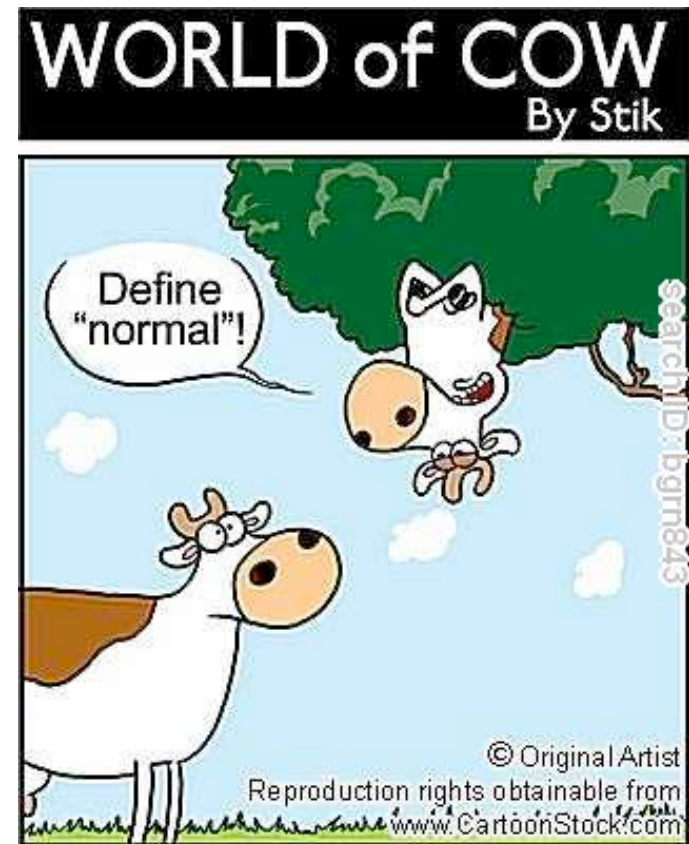
Introduction

❖ *As for any index, 3 alternatives for data entries* **k***:

  ▪ index refers to actual data record with key value **k**

  ▪ index refers to list of <**k**, rid> pairs

  ▪ index refers to list of <**k**, [rid list]>

❖ Choice is orthogonal to the *indexing technique* used to locate data entries **k***.

❖ Tree-structured indexing techniques support both *range searches* and *equality searches*.

❖ *ISAM*: static structure; *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 can be quite high (must read entire page to access one record).
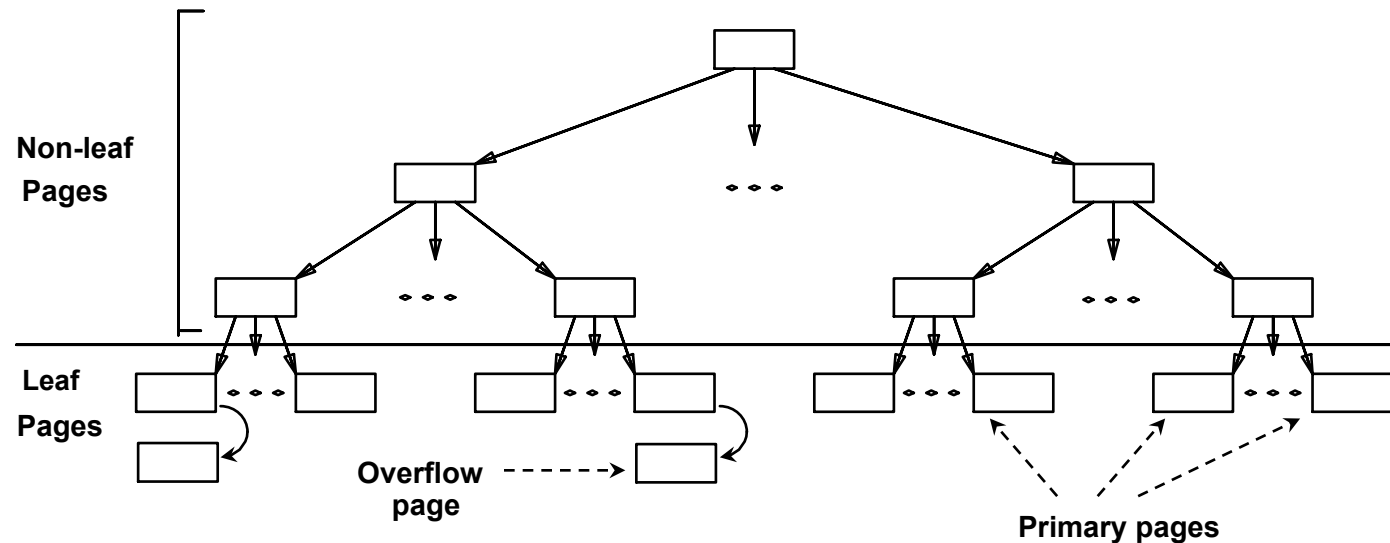
❖ Simple idea: Create an `index' file.

k1 k2		kN

Index File

Page 1	Page 2	Page 3	Page N

Data File

☞ *Can do binary search on (smaller) index file!*

ISAM – *Indexed Sequential Access Method*

index entry

P_0	K_1	P_1	K_2	P_2	$\diamond$ $\diamond$ $\diamond$	K_m	P_m
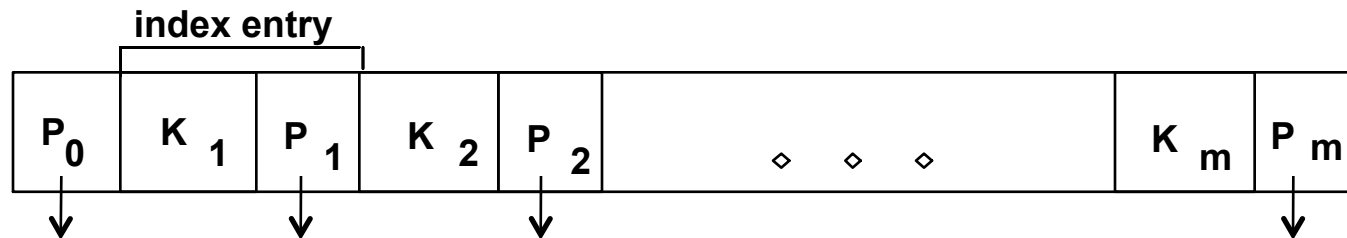
- ❖ Index file may be quite large.
- ❖ Can be applied hierarchically!

Non-leaf Pages

Leaf Pages

Overflow page

Primary pages

☛ *Leaf pages contain* *data entries (i.e. actual records or <key, rid> pairs.*

Comments on ISAM
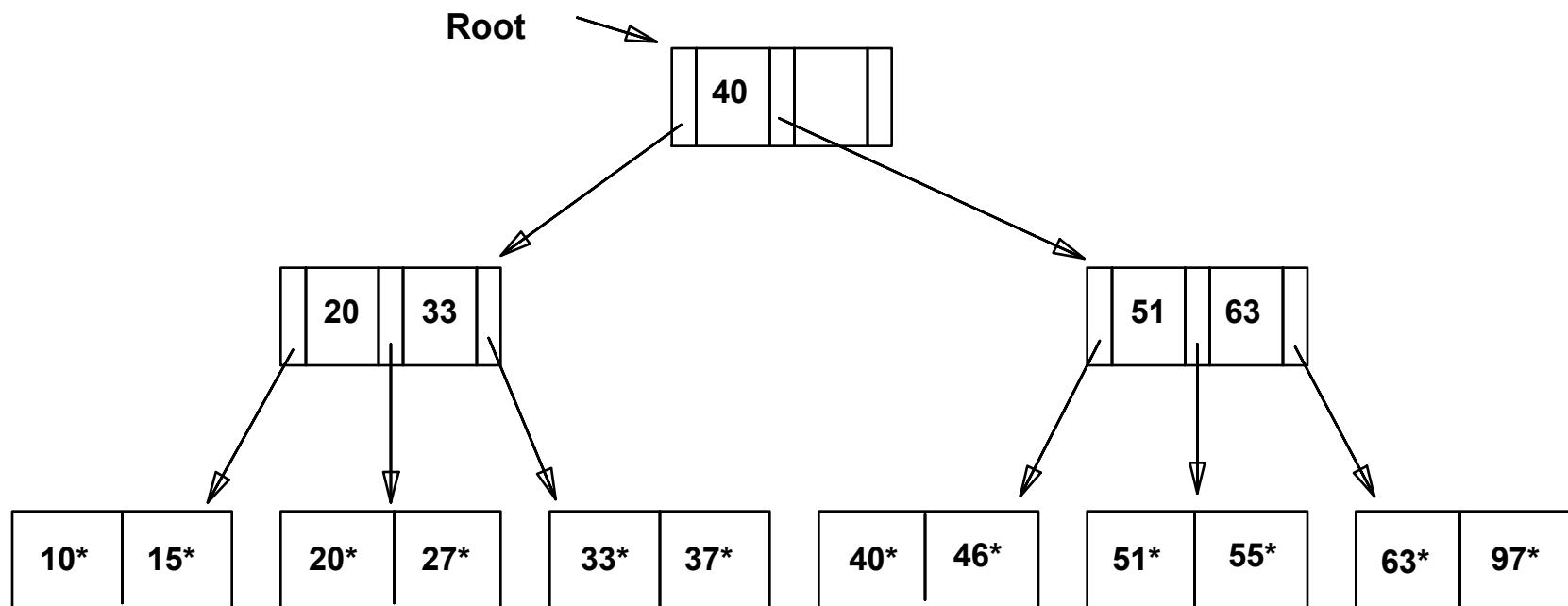
❖ *File creation*: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.

❖ *Index entries*: <search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.

❖ <u>Search</u>: Start at root; use key comparisons to go to leaf. Cost $\log_F N$
F = # entries/index pg, N = # leaf pgs

❖ <u>Insert</u>: Find leaf data entry belongs to, put it there if space is available, else allocate an overflow page, put it there, and link it in.

❖ <u>Delete</u>: Find and remove from leaf; if empty overflow page, de-allocate.

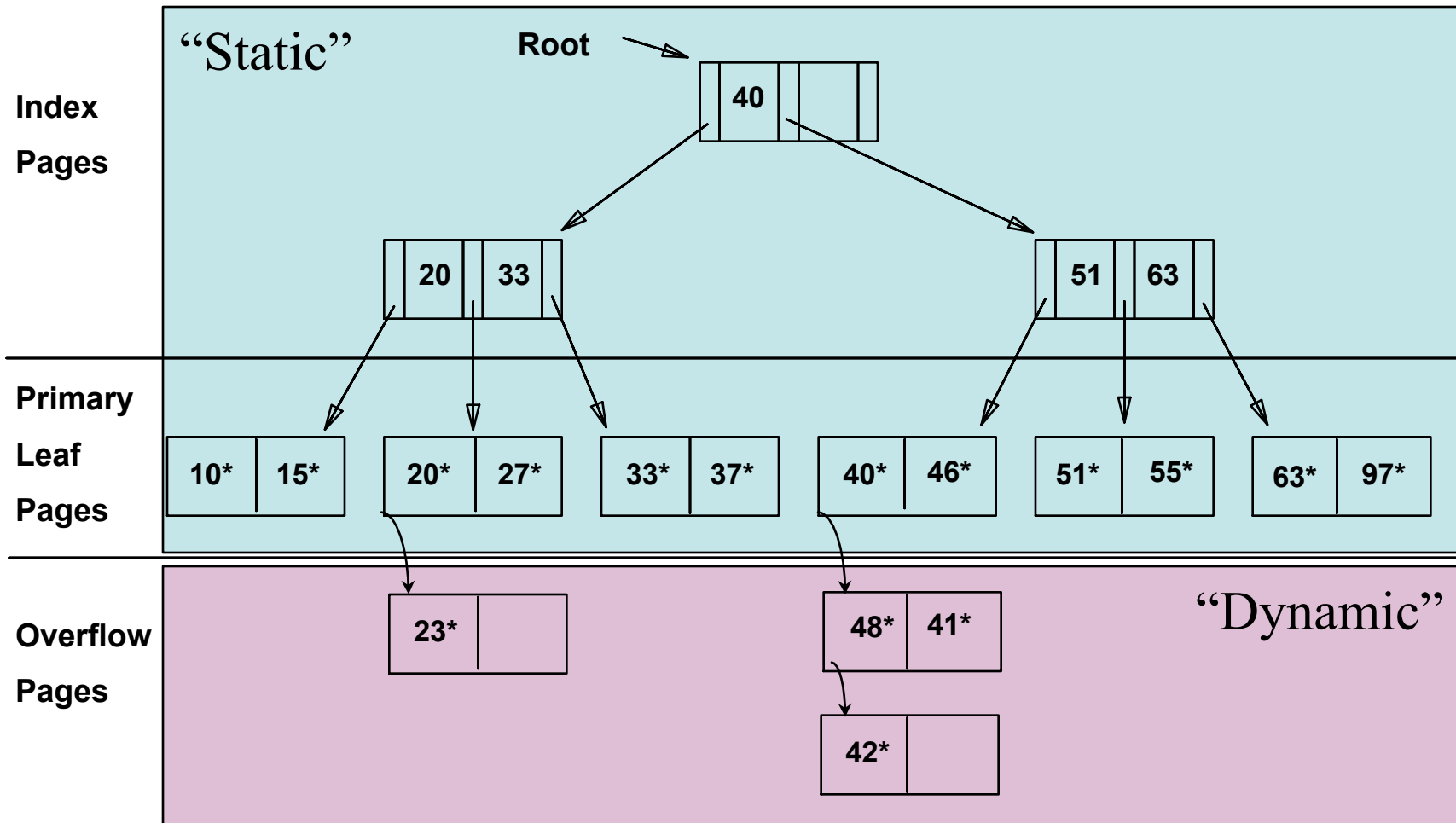☞ **Static tree structure**: *inserts/deletes affect only leaf pages.*

Data Pages
Index Pages
Overflow pages
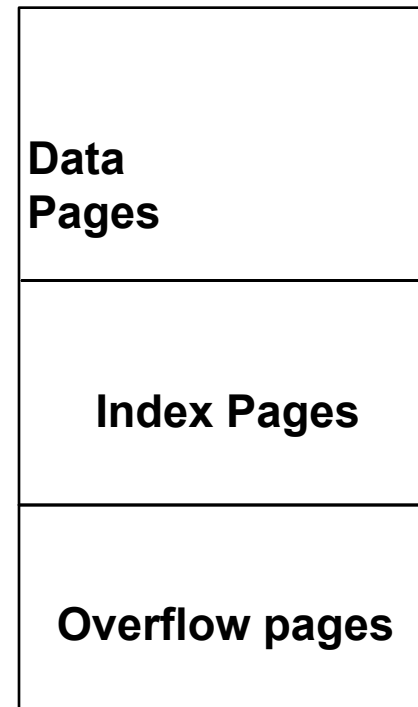
Example ISAM Tree

❖ Each node can hold 2 entries; no need for `next-leaf-page' pointers. (Why?)

Root

	40	

	20	33	

	51	63	

10*	15*

20*	27*

33*	37*

40*	46*

51*	55*

63*	97*

After Inserting 23*, 48*, 41*, 42* ...

"Static"

Root

Index Pages

| 40 | | |

| 20 | 33 | | 51 | 63 |

Primary Leaf Pages

| 10* | 15* | | 20* | 27* | | 33* | 37* | | 40* | 46* | | 51* | 55* | | 63* | 97* |

Overflow Pages

"Dynamic"

| 23* | |

| 48* | 41* |

| 42* | |

... *Then Deleting 42*, 51*, 97**
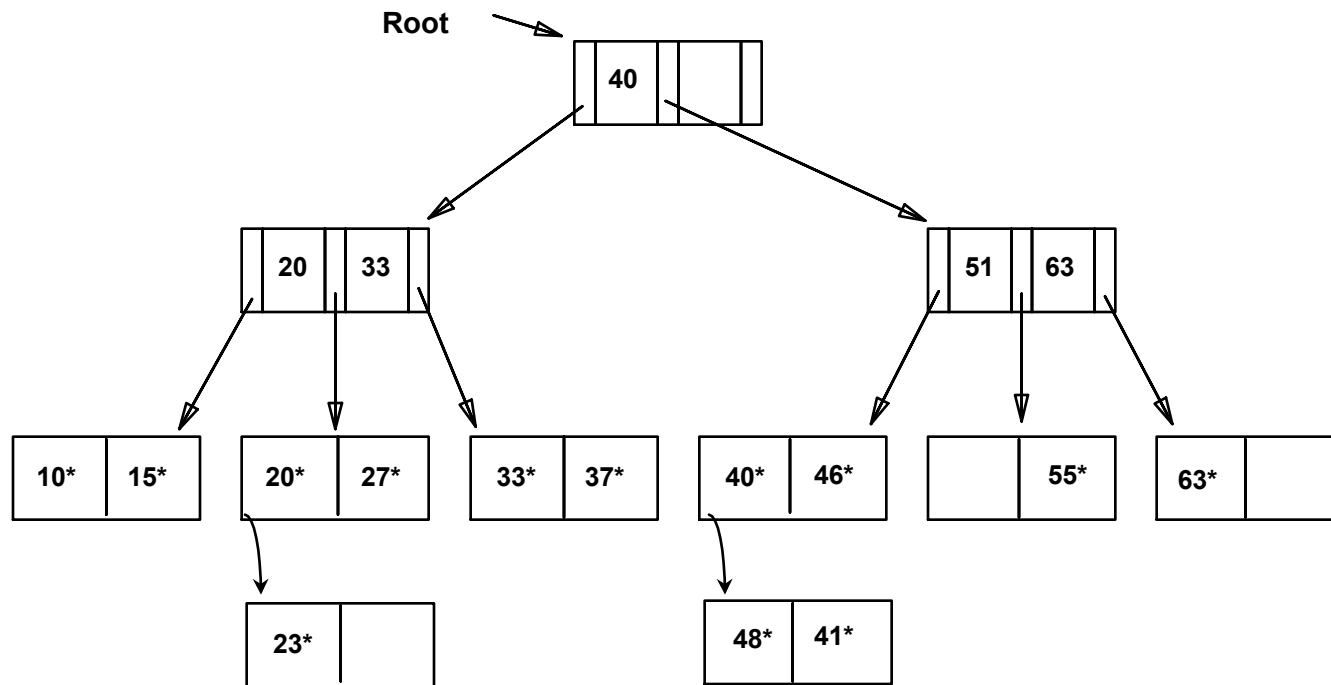
☛ *Note that 51* appears in index, but not in leaf!*
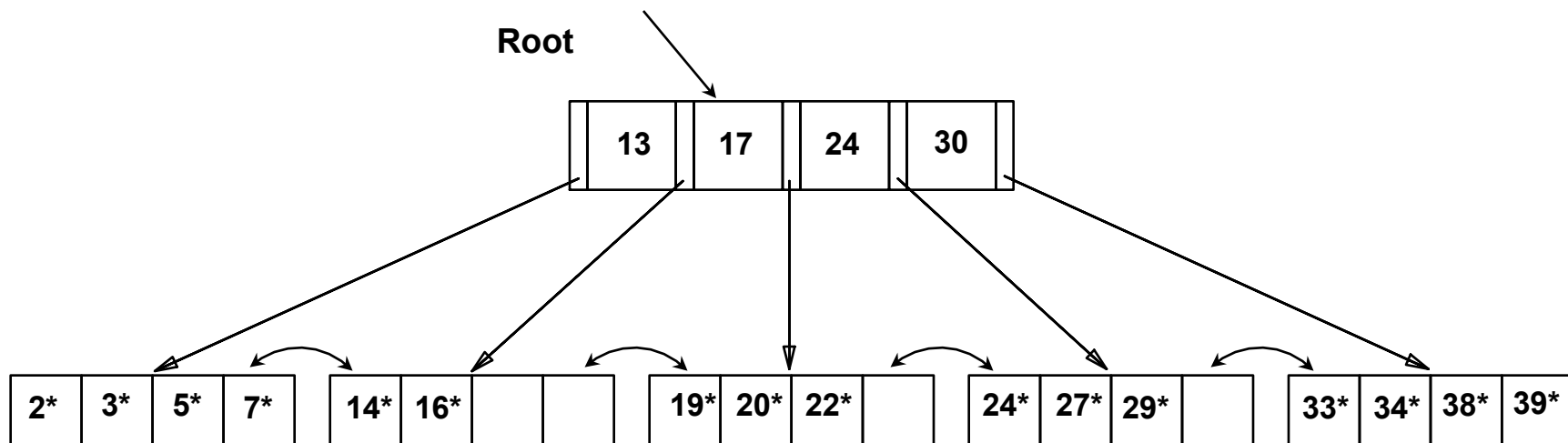
B+ Tree: Most Widely Used Index

❖ Insert/delete at $\log_F N$ cost; keep tree *balanced.*
(F = fanout, N = # leaf pages)

❖ Minimum 50% occupancy. Each internal non-root node contains **d** <= $\underline{m}$ <= 2**d** entries. The parameter **d** is called the *order* of the tree.

❖ Supports equality and range-searches efficiently.

Index Entries
(Direct search)

Data Entries
("Sequence set")
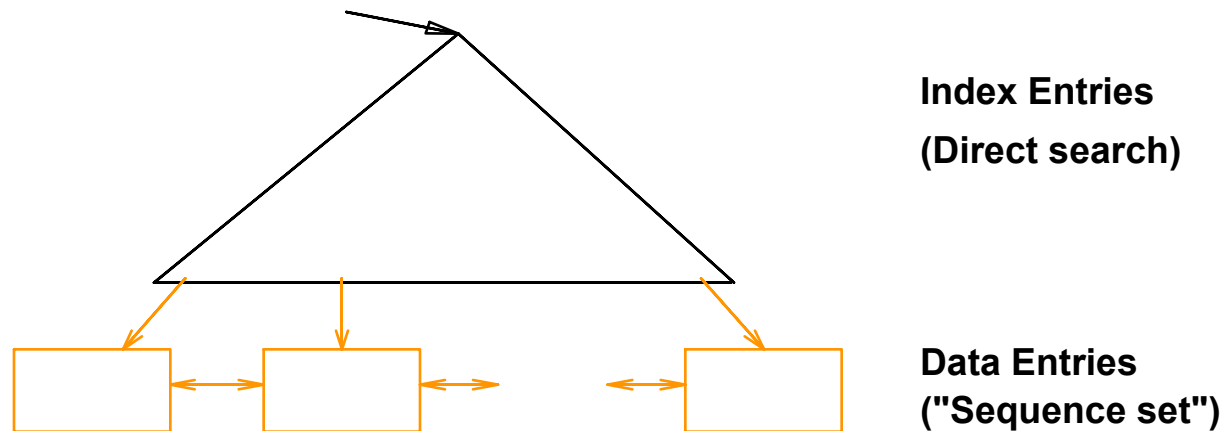
Example B+ Tree

❖ Search begins at root, and key comparisons direct it to a leaf (as in ISAM).

❖ Search for 5*, 15*, all data entries >= 24* ...

Root

| 13 | 17 | 24 | 30 |

| 2* | 3* | 5* | 7* | | 14* | 16* | | | | 19* | 20* | 22* | | | 24* | 27* | 29* | | | 33* | 34* | 38* | 39* |

☞ *Based on the search for 15*, we <u>know</u> it is not in the tree!*

B+ Trees in Practice

❖ Typical order: 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

❖ Can often hold top levels in buffer pool:

- Level 1 =   1 page =   8 Kbytes
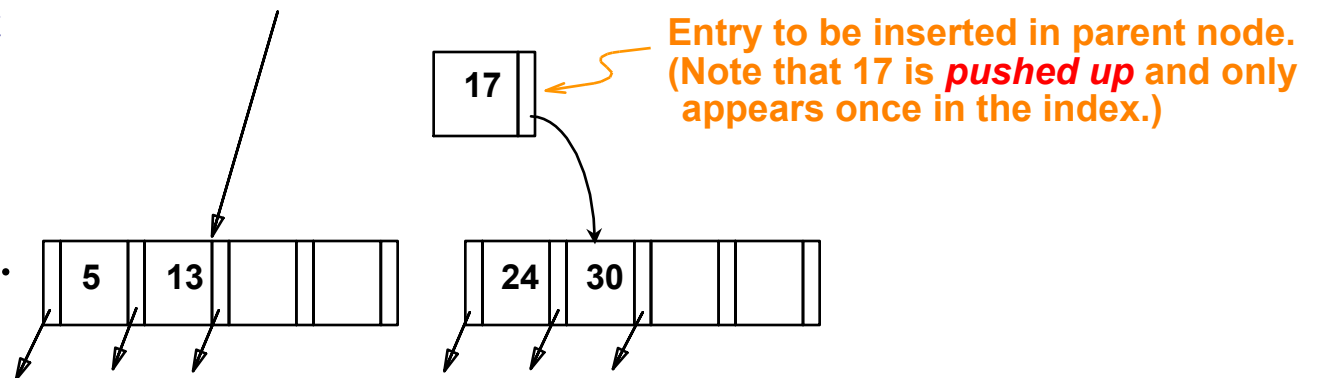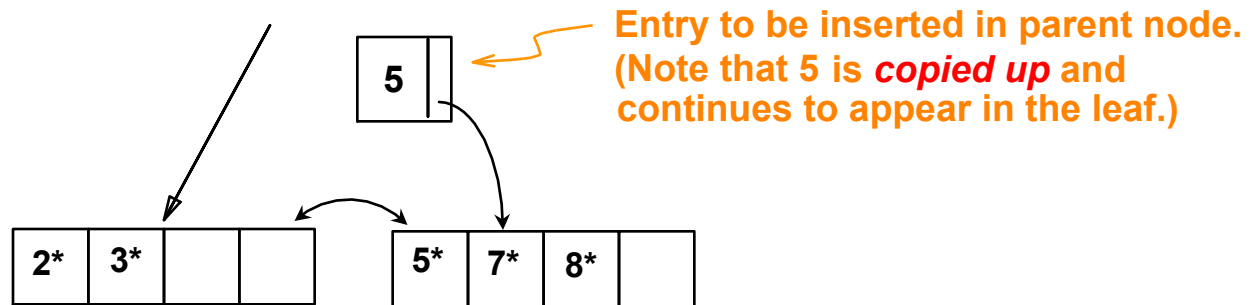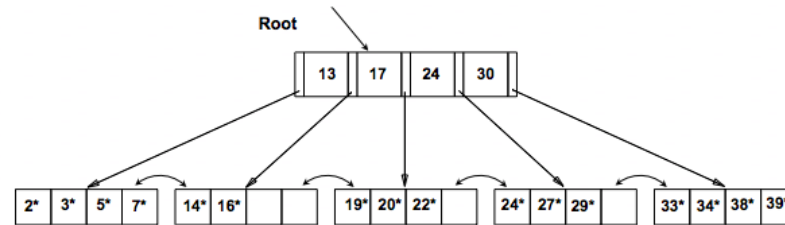- Level 2 =   133 pages =   1 Mbyte
- Level 3 = 17,689 pages = 133 Mbytes

Inserting 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)*
    - • Allocate new node
    - • 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.
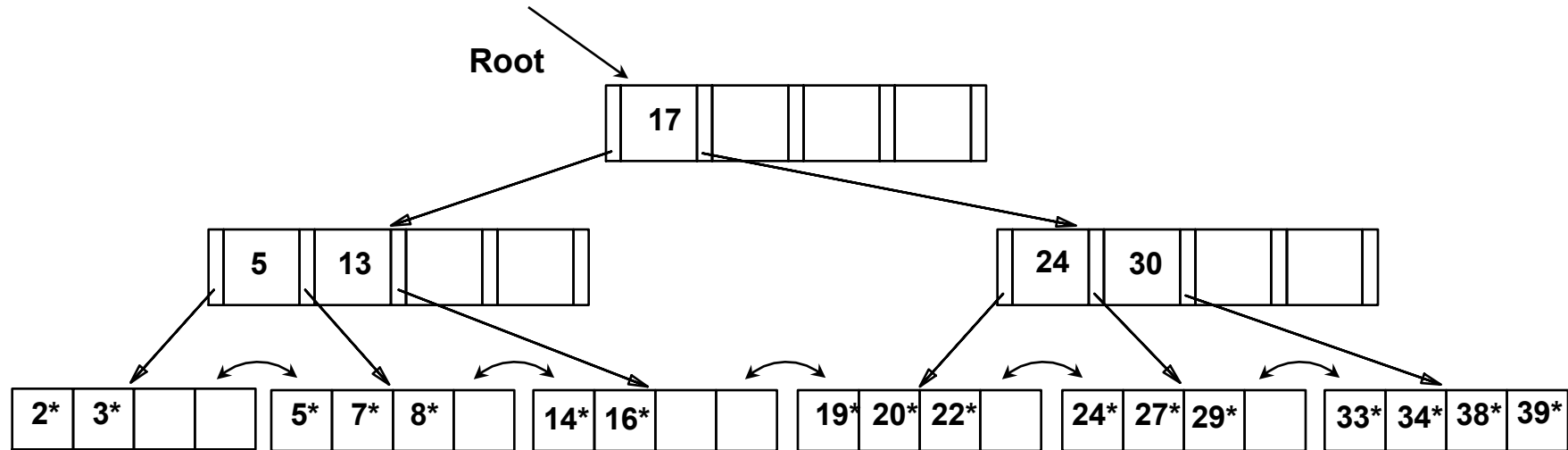 - ▪ Tree growth: gets *wider* or *one level taller at top.*

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 *push-up*; be sure you understand the reasons for this.

Root

| 13 | 17 | 24 | 30 |

| 2* | 3* | 5* | 7* | | | 14* | 16* | | | | 19* | 20* | 22* | | | 24* | 27* | 29* | | | 33* | 34* | 38* | 39* |

Entry to be inserted in parent node. (Note that 5 is *copied up* and continues to appear in the leaf.)

| 5 |

| 2* | 3* | | | | 5* | 7* | 8* | |

Entry to be inserted in parent node. (Note that 17 is *pushed up* and only appears once in the index.)

| 17 |

| 5 | 13 | | | | | 24 | 30 | | | |

*Example B+ Tree After Inserting 8**

Root

| | 17 | | | |

| | 5 | 13 | | | |

| | 24 | 30 | | | |

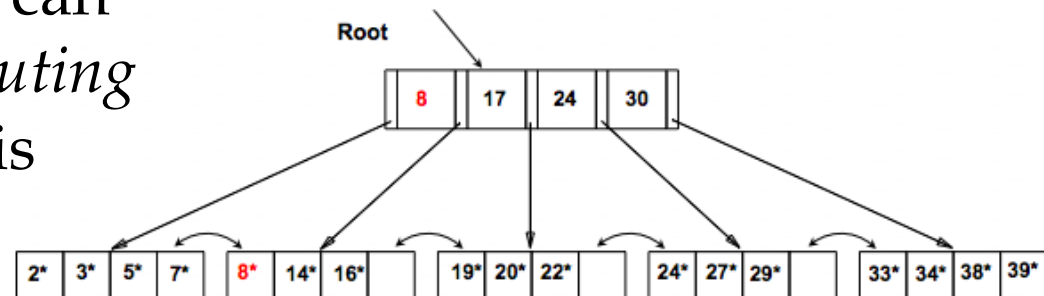| 2* | 3* | | |
5*	7*	8*	
14*	16*		
19*	20*	22*	
24*	27*	29*	
33*	34*	38*	39*

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

❖ In this example, we can avoid split by *redistributing* entries; however, this is usually not done in practice.

Root

| | 8 | 17 | 24 | 30 | |

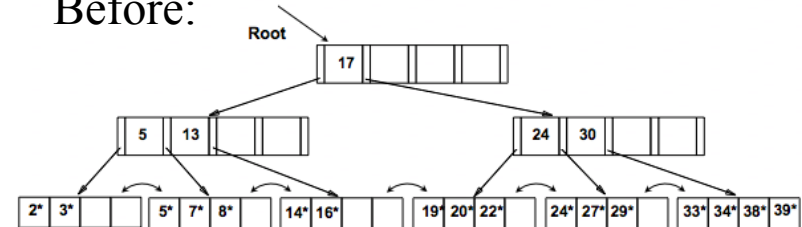| 2* | 3* | 5* | 7* |
8*	14*	16*	
19*	20*	22*	
24*	27*	29*	
33*	34*	38*	39*

Deleting a Data Entry from a B+ Tree

- ❖ Start at root, find leaf *L* with entry, if it exists.
- ❖ Remove the entry.
  - ▪ If L is at least half-full, *done!*
  - ▪ If L has only **d-1** entries,
    - • Try to re-distribute, borrowing keys from *sibling* *(adjacent node with same parent as L).*
    - • If redistribution fails, *merge* L and sibling.
- ❖ If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
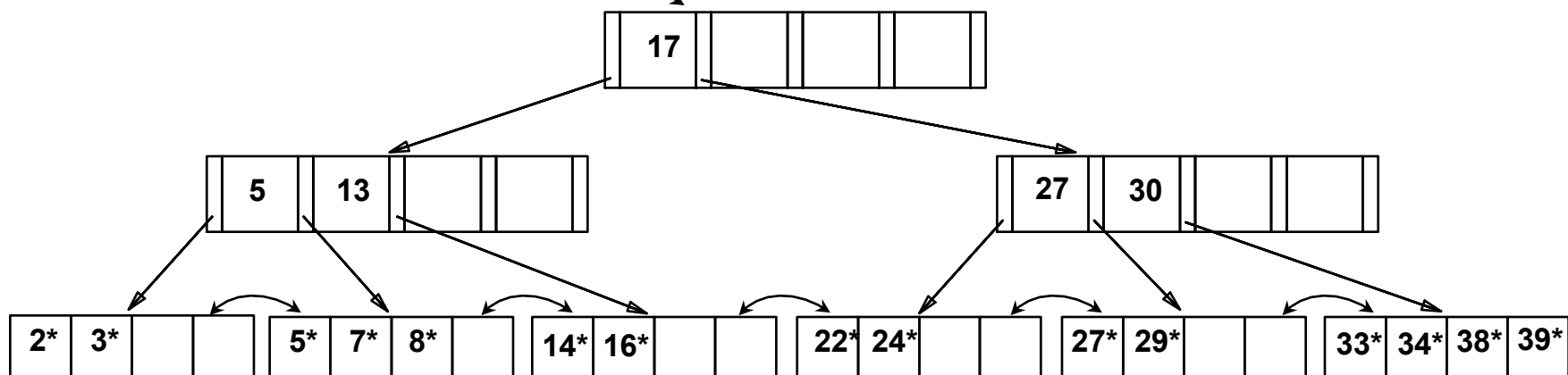- ❖ Merge could propagate to root, decreasing height.

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

Before:

Root

| 17 | | | |

| 5 | 13 | | | | 24 | 30 | | |

| 2* | 3* | | | | 5* | 7* | 8* | | | 14* | 16* | | | | 19* | 20* | 22* | | 24* | 27* | 29* | | 33* | 34* | 38* | 39* |

Root

| 17 | | | |

| 5 | 13 | | | | 27 | 30 | | |

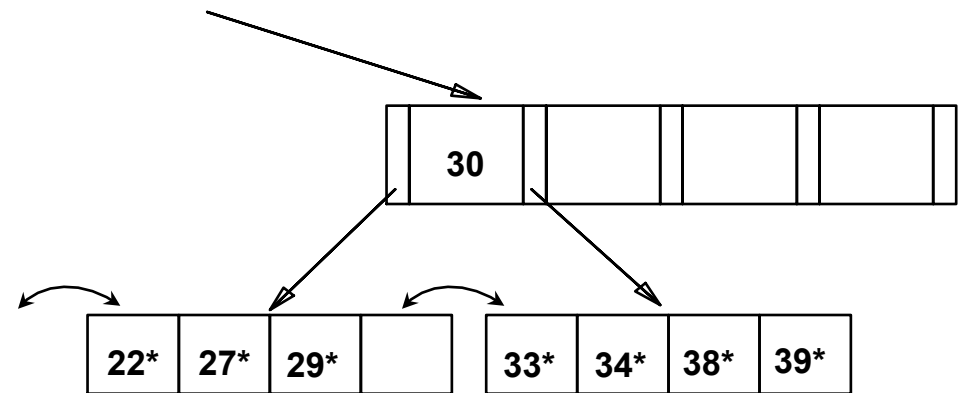| 2* | 3* | | | | 5* | 7* | 8* | | | 14* | 16* | | | | 22* | 24* | | | 27* | 29* | | | 33* | 34* | 38* | 39* |

❖ Deleting 19* is easy.

❖ Deleting 20* is done with redistribution. Notice how middle key is *copied up*.
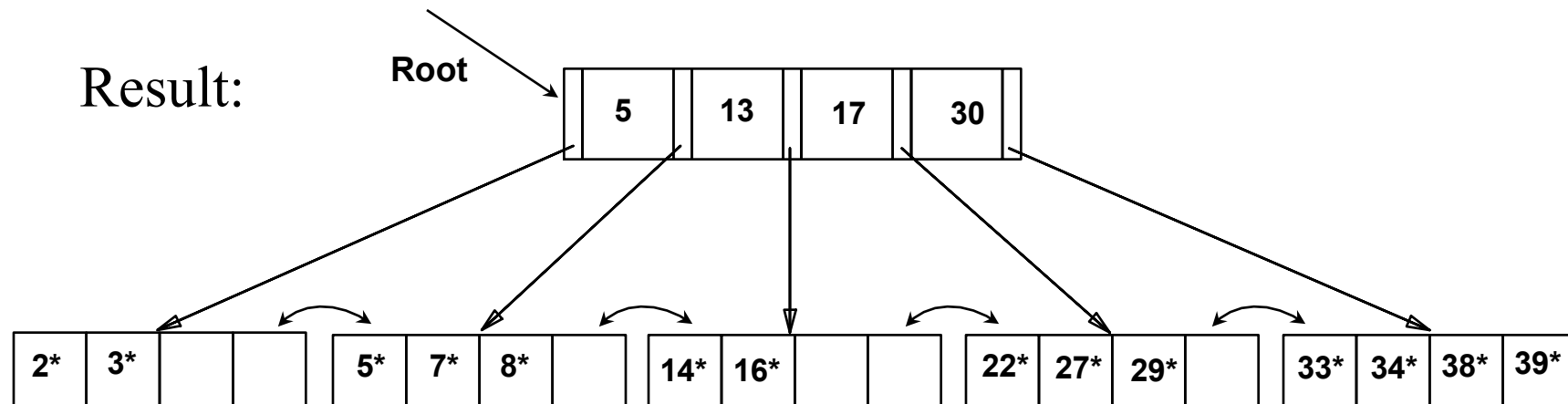
... *And Then Deleting 24**

- ❖ Must merge (twice).
- ❖ Observe `*toss'* of index entry (27) in first merge and `*pull down'* of index entry (17) in second.

	30						

22*	27*	29*	

33*	34*	38*	39*

Result:

Root

5	13	17	30

2*	3*		

5*	7*	8*	

14*	16*		

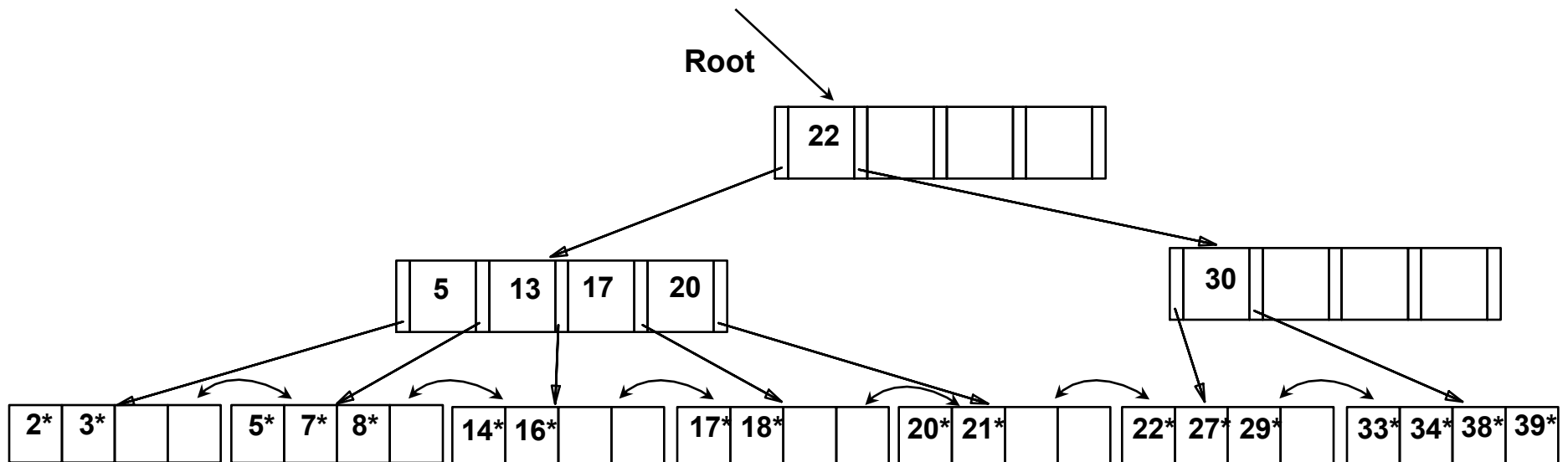| 22*	27*	29*	

33*	34*	38*	39*

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 redistribute entry from left child of root to right child.

Example of Non-leaf Redistribution
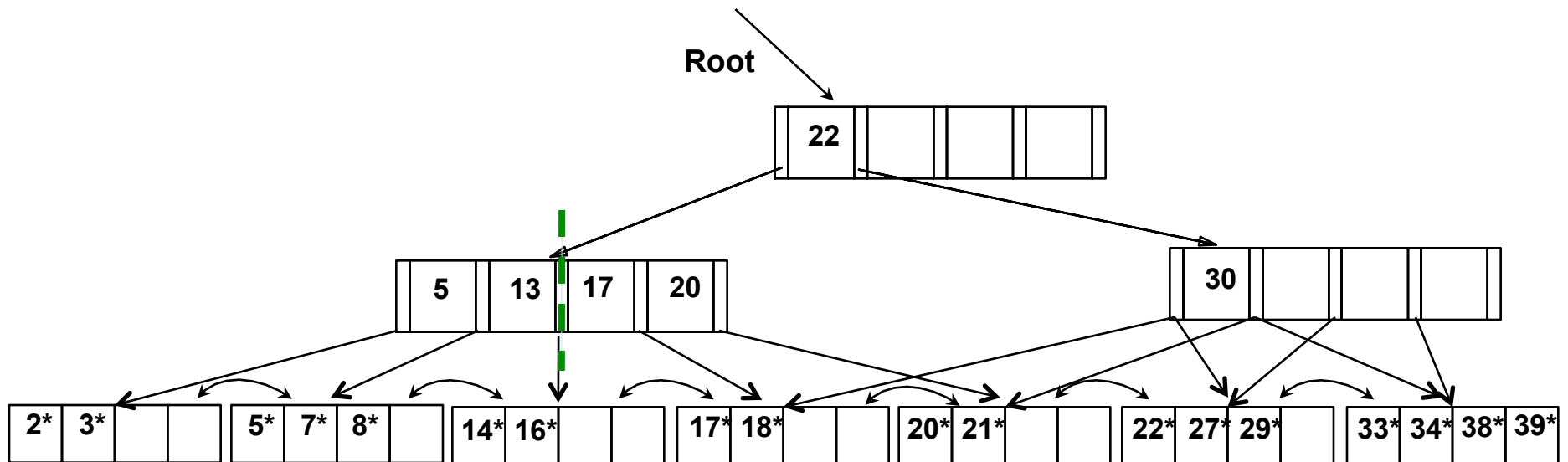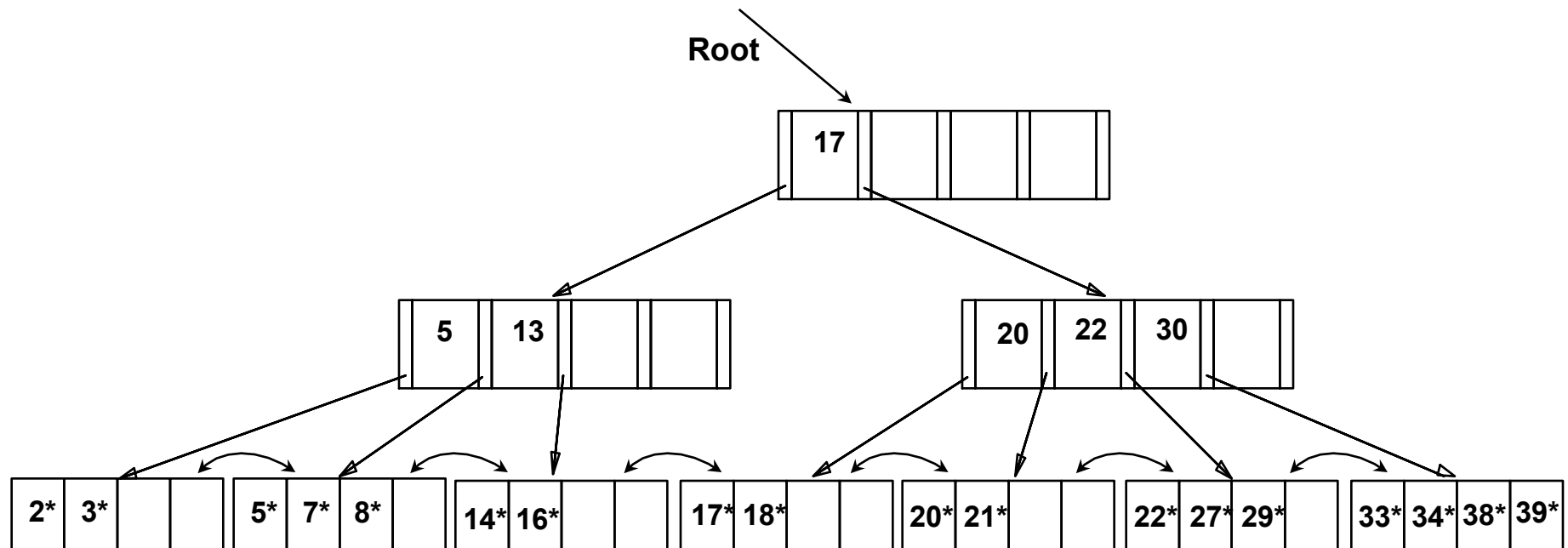
❖ Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)

❖ In contrast to previous example, can redistribute entry from left child of root to right child.

Root

| 22 | | | |

| 5 | 13 | 17 | 20 |

| 30 | | | |

| 2* | 3* | | | | 5* | 7* | 8* | | 14* | 16* | | | | 17* | 18* | | | | 20* | 21* | | | | 22* | 27* | 29* | | 33* | 34* | 38* | 39* |

After Redistribution

❖ Intuitively, entries are redistributed 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.
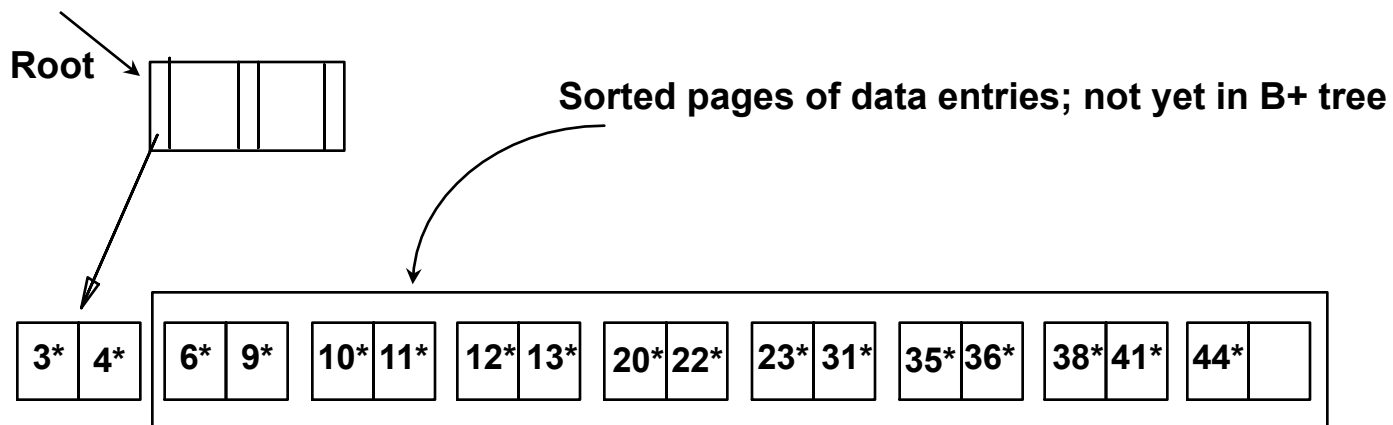
Root

| 17 | | | |

| 5 | 13 | | |

| 20 | 22 | 30 | |

| 2* | 3* | | |

| 5* | 7* | 8* |

| 14* | 16* | | |

| 17* | 18* | | |

| 20* | 21* | | |

| 22* | 27* | 29* |

| 33* | 34* | 38* | 39* |

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.

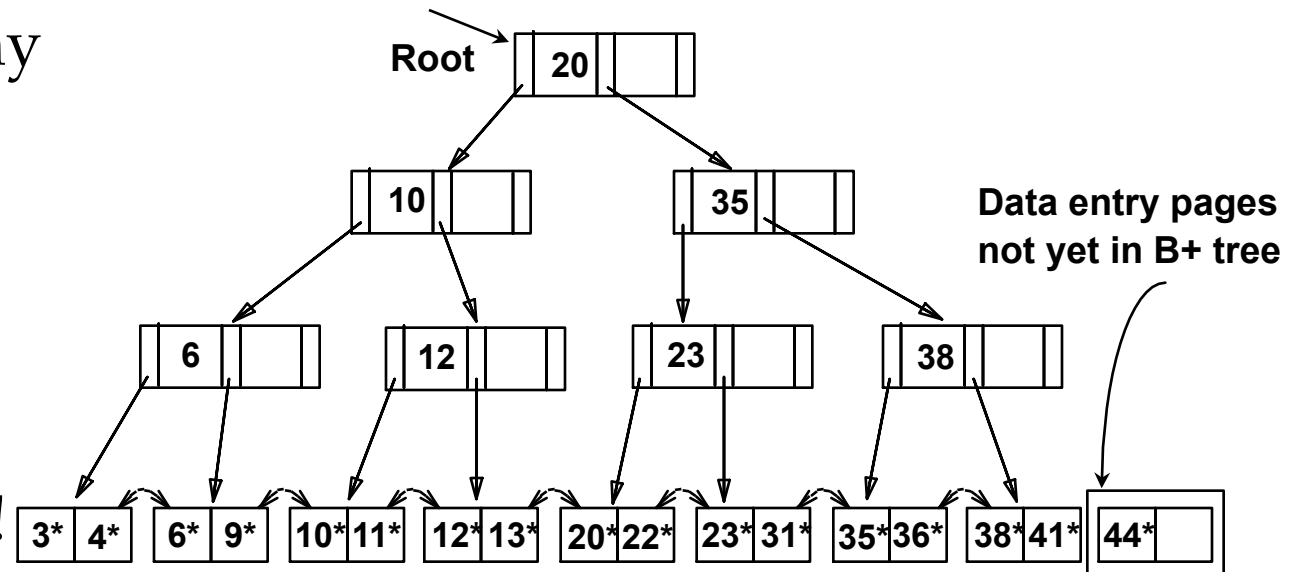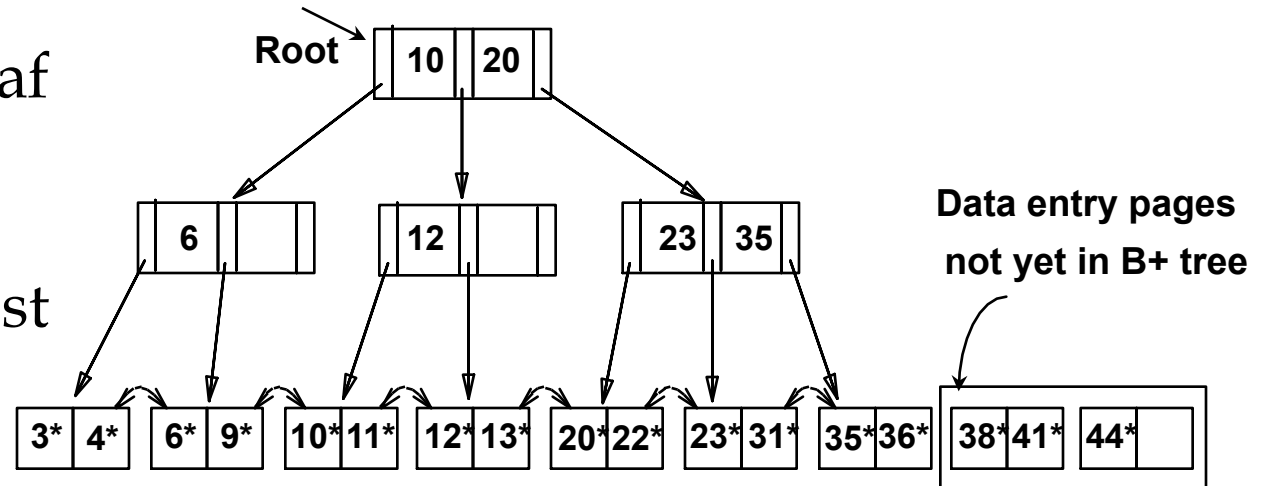❖ Insert/delete must be suitably modified.

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.

Root

Sorted pages of data entries; not yet in B+ tree

| 3* | 4* | | 6* | 9* | | 10* | 11* | | 12* | 13* | | 20* | 22* | | 23* | 31* | | 35* | 36* | | 38* | 41* | | 44* | |

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 if one considers locking!

Root → [10 | 20]

[6 | |] [12 | |] [23 | 35]

Data entry pages not yet in B+ tree

[3* 4*] [6* 9*] [10* 11*] [12* 13*] [20* 22*] [23* 31*] [35* 36*] [38* 41* | 44*]

Root → [20 | |]

[10 | |] [35 | |]

[6 | |] [12 | |] [23 | |] [38 | |]

Data entry pages not yet in B+ tree

[3* 4*] [6* 9*] [10* 11*] [12* 13*] [20* 22*] [23* 31*] [35* 36*] [38* 41* | 44*]

Summary of Bulk Loading

❖ Option 1: multiple inserts.

- Slow.
- Does not give 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.

A Note on "Order"

❖ *Order* (**d**) concept replaced by physical space criterion in practice (`at least half-full`).

- Index pages can typically hold many more entries than leaf pages.

- Variable sized records and search keys mean differnt nodes will contain different numbers of entries.

- Even with fixed length fields, multiple records with the same search key value (*duplicates*) can lead to variable-sized data entries (if we use Alternative (3)).

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.
- Almost always better than maintaining a sorted file.

Summary (Contd.)

- Typically, 67% occupancy on average.
- Usually preferable to ISAM, modulo *locking* considerations; adjusts to growth gracefully.
- If data entries are data records, splits can change rids!

❖ 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.