Comp115: Databases

Tree-structured indexing

Instructor: Manos Athanassoulis

Some Reminders

HW3 will be out soon (last homework)

Exercises will be posted for some lectures
    → reading material to help you for midterms

solutions will be posted

midterm 1 is on March 13th

SQL hands-on test (tentatively) on March 15th

project will be out this week and you will be able to work until the end of the semester

Tree-structured indexing

Intro & B⁺-Tree

Insert into a B⁺-Tree

Delete from a B⁺-Tree

Prefix Key Compression & Bulk Loading

Units

Introduction

Recall: 3 alternatives for data entries k*:

- Data record with key value **k**

- <**k**, rid of data record with search key value **k**>

- <**k**, list of rids of data records with search key **k**>

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

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

4

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 maintaining sorted file + performing binary search in a database can be quite high. Q: Why???

Simple idea: Create an "index" file.

k1 k2		kN

Index File

Page 1	Page 2	Page 3		Page N

Data File
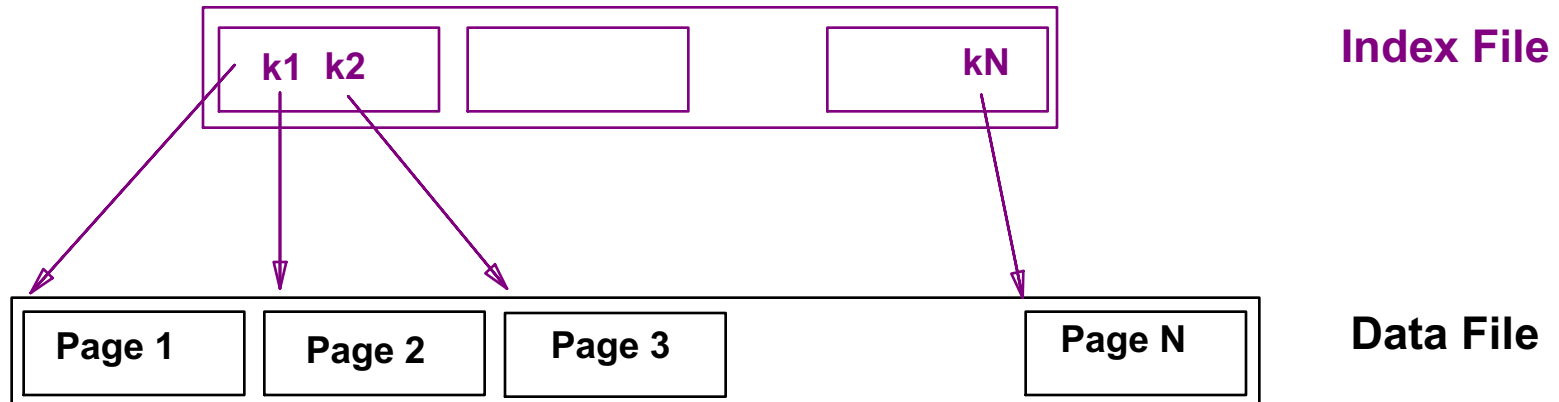
☛ *Can do binary search on (smaller) index file!*

5

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* <= 2d entries. "d" is called the *order* of the tree.

Supports <u>equality</u> and <u>range-searches</u> efficiently.

All searches go from root to leaves, in a <u style="color:red">dynamic</u> structure.

Index Entries

(Direct search)

Data Entries

6
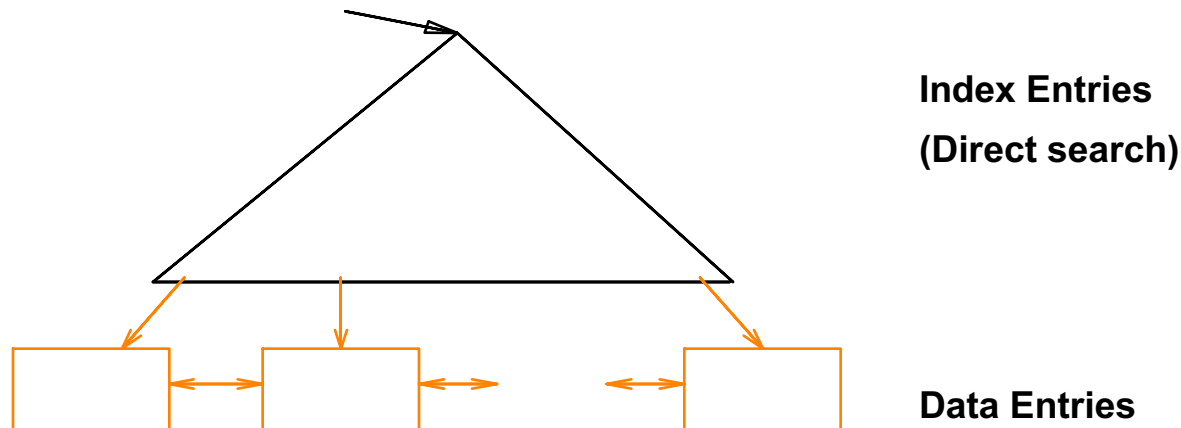
Example B+ Tree

Search begins at root, and key comparisons direct it to a leaf.

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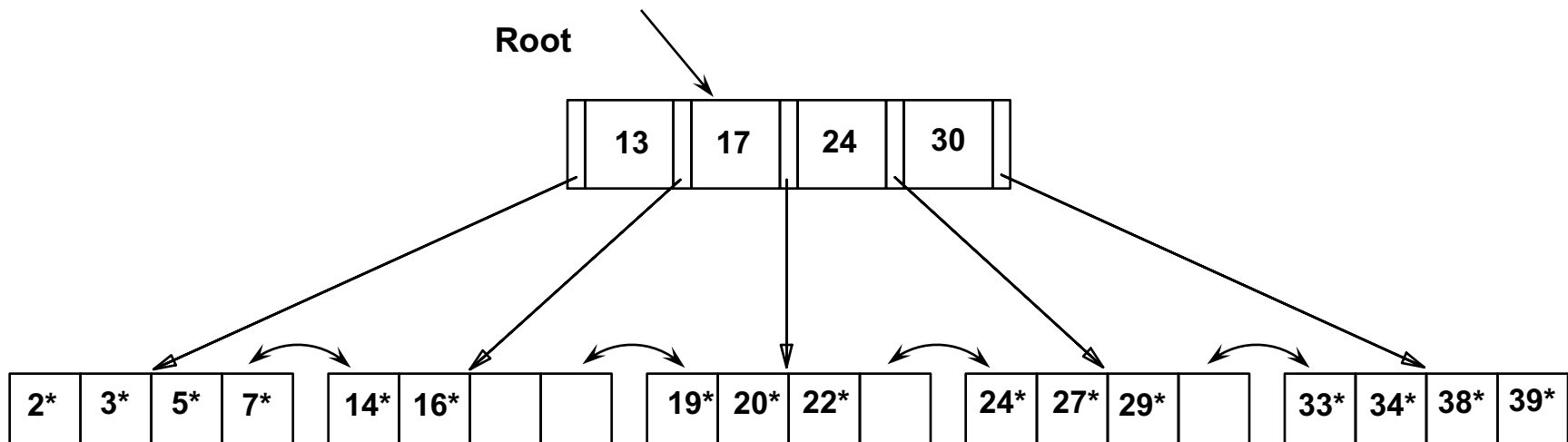
☞ *Based on the search for 15*, we <u>know</u> it is not in the tree!*

B+ Trees in Practice (cool facts!)

Typical order: 100. Typical fill-factor: 67%.

– average fanout = 2*100*0.67 = 134

Typical capacities:

– Height 4: 133^4 = 312,900,721 entries
– Height 3: 133^3 =   2,406,104 entries

Can often hold top levels in buffer pool:

– Level 1 =   1 page =   8 KB
– Level 2 =   134 pages =   1 MB
– Level 3 = 17,956 pages = 140 MB

Tree-structured indexing

Intro & B$^+$-Tree

Insert into a B$^+$-Tree

Delete from a B$^+$-Tree

Prefix Key Compression & Bulk Loading

Units

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.
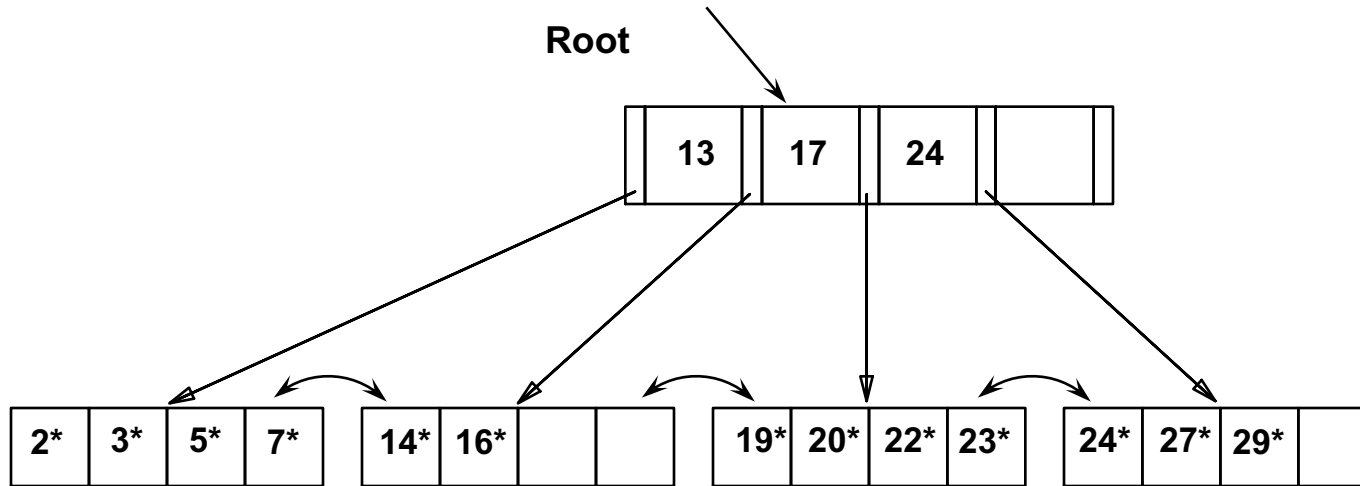 Insert index entry pointing to *L2* **into parent** of *L*.

This can happen recursively

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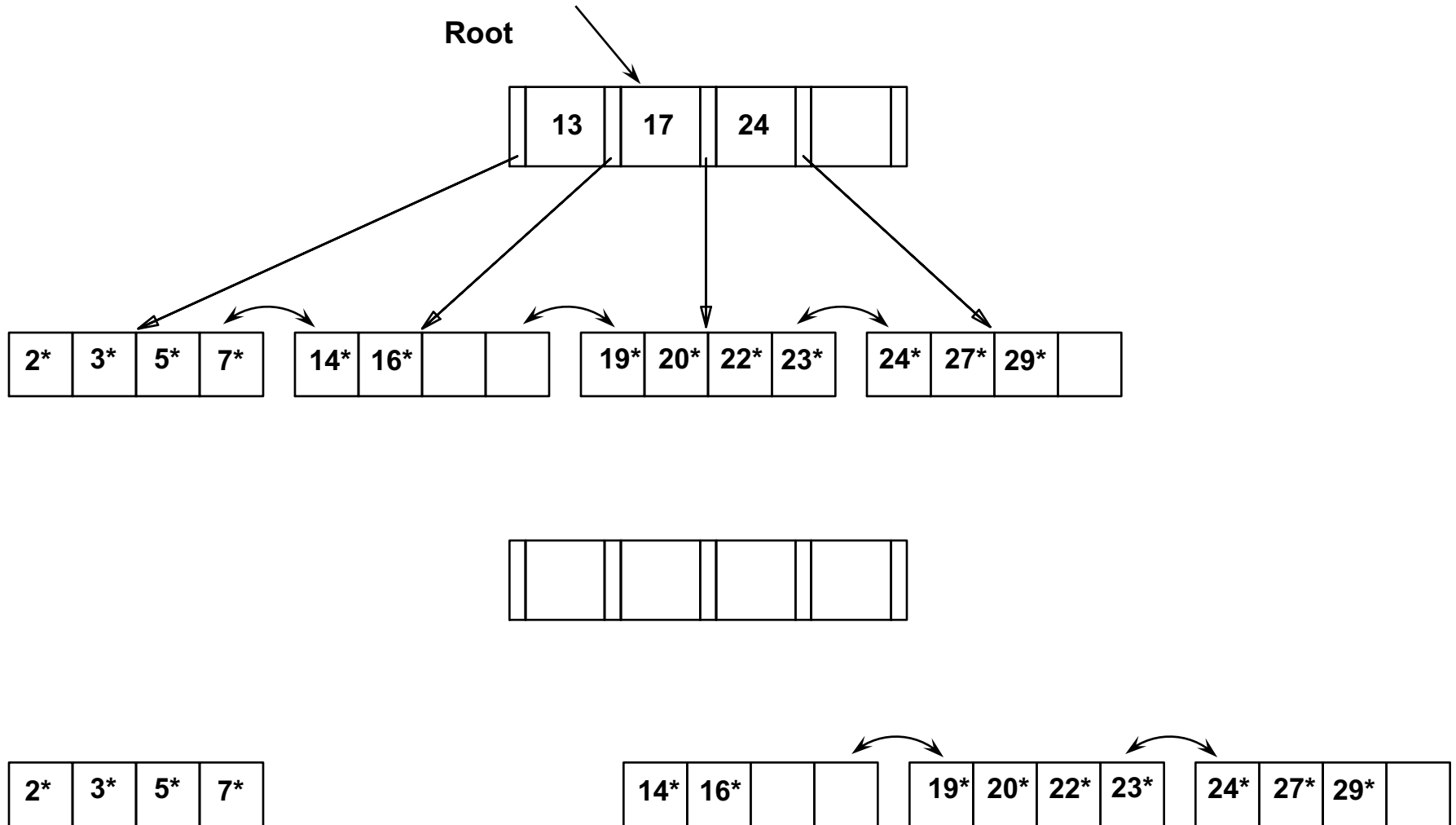
Splits "grow" tree; root split increases height.
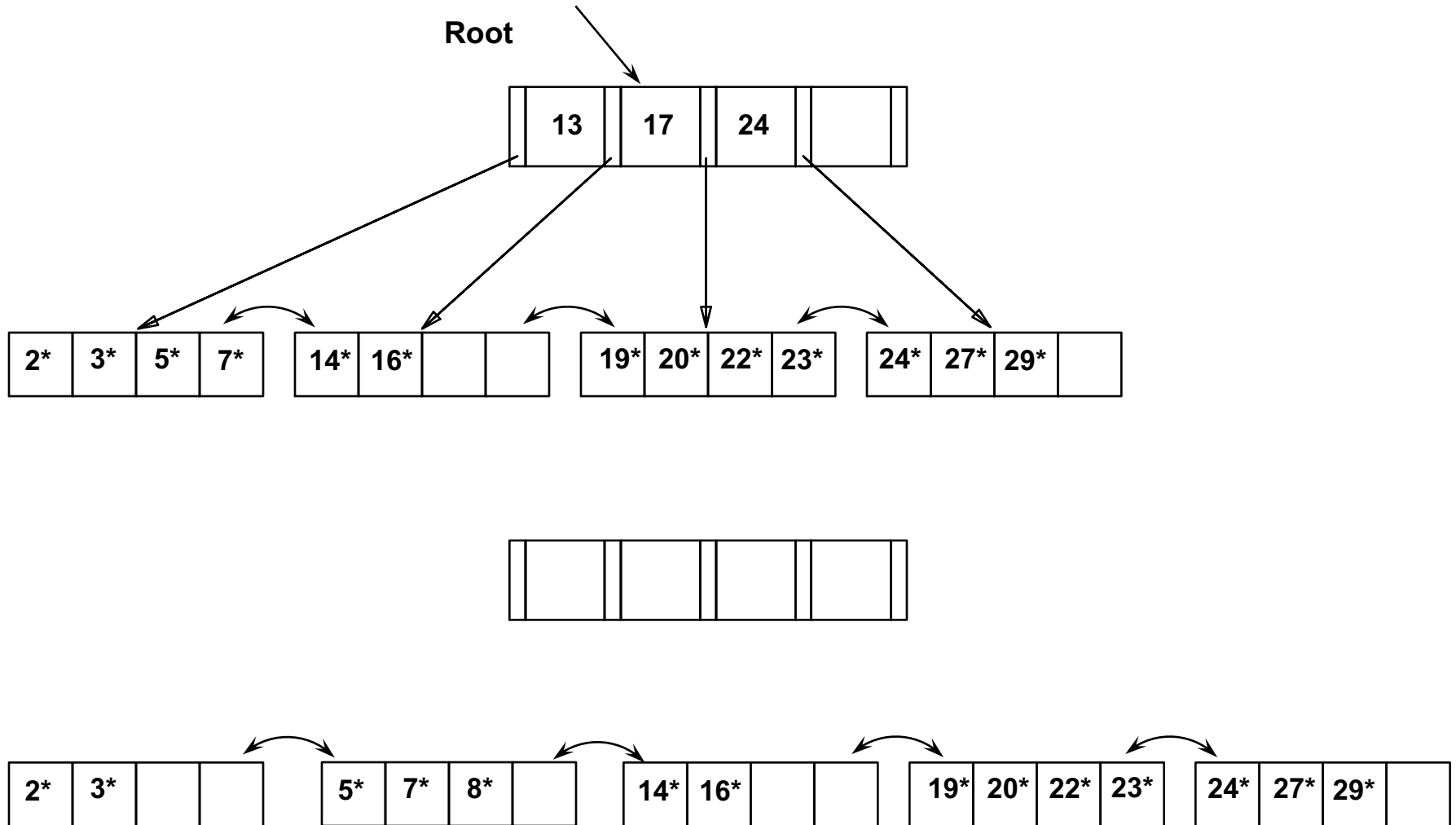
- Tree growth: gets *wider* or *one level taller at top.*

10

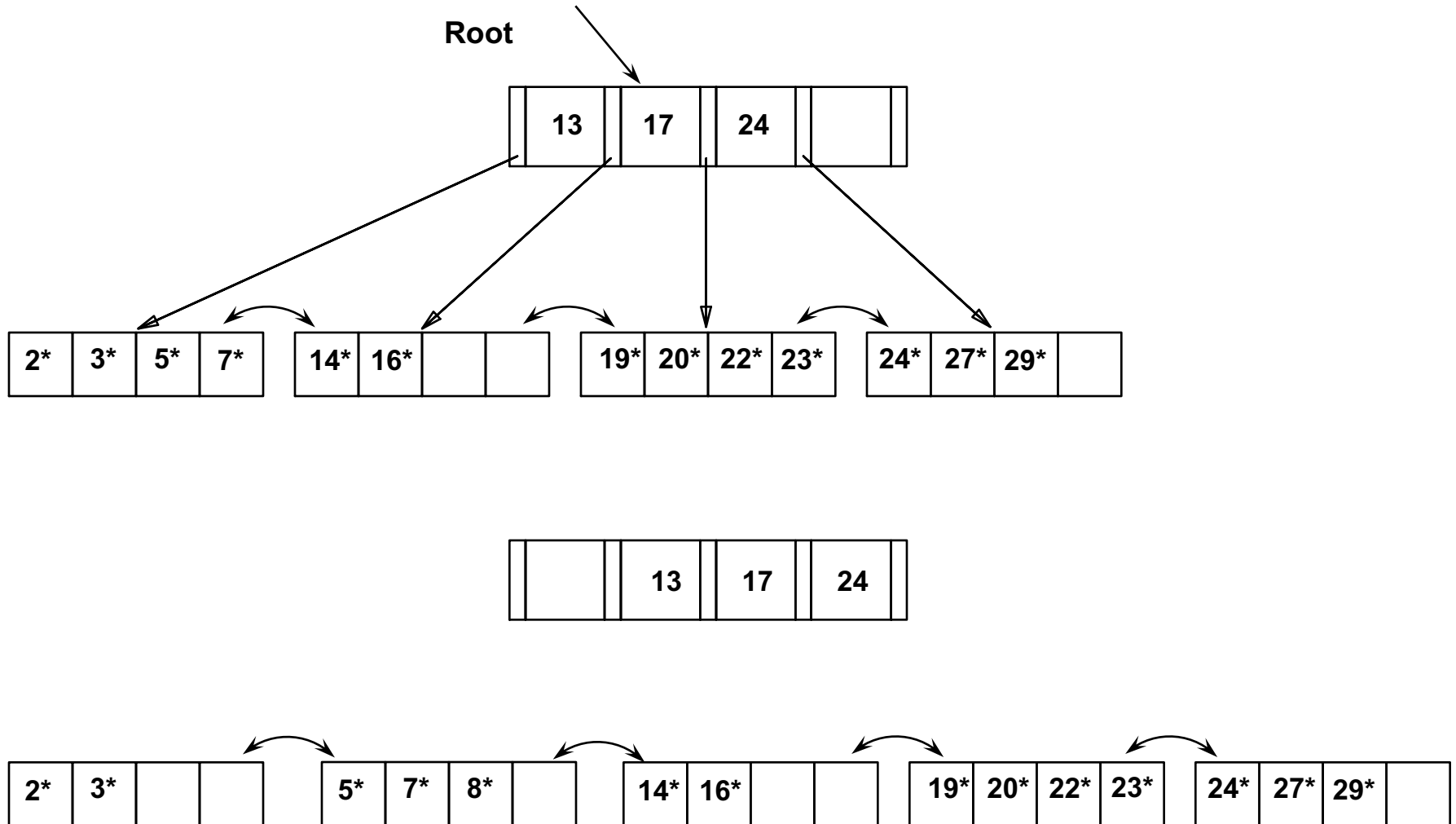# Example B+ Tree - Inserting 8*

Root

	13	17	24	

2*	3*	5*	7*		14*	16*				19*	20*	22*	23*		24*	27*	29*	

Example B+ Tree - Inserting 8*

Root

| | 13 | 17 | 24 | |

| 2* | 3* | 5* | 7* | | 14* | 16* | | | | 19* | 20* | 22* | 23* | | 24* | 27* | 29* | |

| | | | | |

| 2* | 3* | 5* | 7* | | 14* | 16* | | | | 19* | 20* | 22* | 23* | | 24* | 27* | 29* | |

Example B+ Tree - Inserting 8*

Root

| | 13 | 17 | 24 | |

| 2* | 3* | 5* | 7* | | 14* | 16* | | | | 19* | 20* | 22* | 23* | | 24* | 27* | 29* | |

| | | | | |

| 2* | 3* | | | | 5* | 7* | 8* | | | 14* | 16* | | | | 19* | 20* | 22* | 23* | | 24* | 27* | 29* | |

Example B+ Tree - Inserting 8*

Root

| | 13 | 17 | 24 | |

| 2* | 3* | 5* | 7* | | 14* | 16* | | | | 19* | 20* | 22* | 23* | | 24* | 27* | 29* | |

| | | 13 | | 17 | | 24 | |

| 2* | 3* | | | | 5* | 7* | 8* | | | 14* | 16* | | | | 19* | 20* | 22* | 23* | | 24* | 27* | 29* | |

14

Example B+ Tree - Inserting 8*

Root

	13	17	24	

| 2* | 3* | 5* | 7* | | 14* | 16* | | | | 19* | 20* | 22* | 23* | | 24* | 27* | 29* | |

	5	13	17	24	

| 2* | 3* | | | | 5* | 7* | 8* | | | 14* | 16* | | | | 19* | 20* | 22* | 23* | | 24* | 27* | 29* | |

15

Example B+ Tree - Inserting 21*

Root

| 5 | 13 | 17 | 24 |

| 2* | 3* | | | 5* | 7* | 8* | | 14* | 16* | | | 19* | 20* | 22* | 23* | 24* | 27* | 29* | |

| 5 | 13 | 17 | 24 |

| 2* | 3* | | | 5* | 7* | 8* | | 14* | 16* | | | 19* | 20* | 22* | 23* | | 24* | 27* | 29* | |

16

Example B+ Tree - Inserting 21*

Root

| 5 | 13 | 17 | 24 |

| 2* | 3* | | | 5* | 7* | 8* | | 14* | 16* | | | 19* | 20* | 22* | 23* | 24* | 27* | 29* | |

| 5 | 13 | 17 | 24 |

| 2* | 3* | | | 5* | 7* | 8* | | 14* | 16* | | | 19* | 20* | | | 21* | 22* | 23* | | 24* | 27* | 29* | |

17

Example B+ Tree - Inserting 21*

Root

| 5 | 13 | 17 | 24 |

| 2* | 3* | | | | 5* | 7* | 8* | | | 14* | 16* | | | | 19* | 20* | 22* | 23* | | 24* | 27* | 29* | |

| 5 | 13 | 17 | |

| 21 | 24 | | |

| 2* | 3* | | | | 5* | 7* | 8* | | 14* | 16* | | | | 19* | 20* | | | 21* | 22* | 23* | | 24* | 27* | 29* | |

18

Example B+ Tree - Inserting 21*

Root

| 5 | 13 | 17 | 24 |

| 2* | 3* | | | | 5* | 7* | 8* | | | 14* | 16* | | | | 19* | 20* | 22* | 23* | | 24* | 27* | 29* | |

Root

| 17 | | | |

| 5 | 13 | | | | 21 | 24 | | |

| 2* | 3* | | | | 5* | 7* | 8* | | | 14* | 16* | | | | 19* | 20* | | | | 21* | 22* | 23* | | | 24* | 27* | 29* | |

Example B+ Tree

Root

17

5 **13**

21 **24**

2* **3***

5* **7*** **8***

14* **16***

19* **20***

21* **22*** **23***

24* **27*** **29***

Notice that root was split, leading to increase in height.
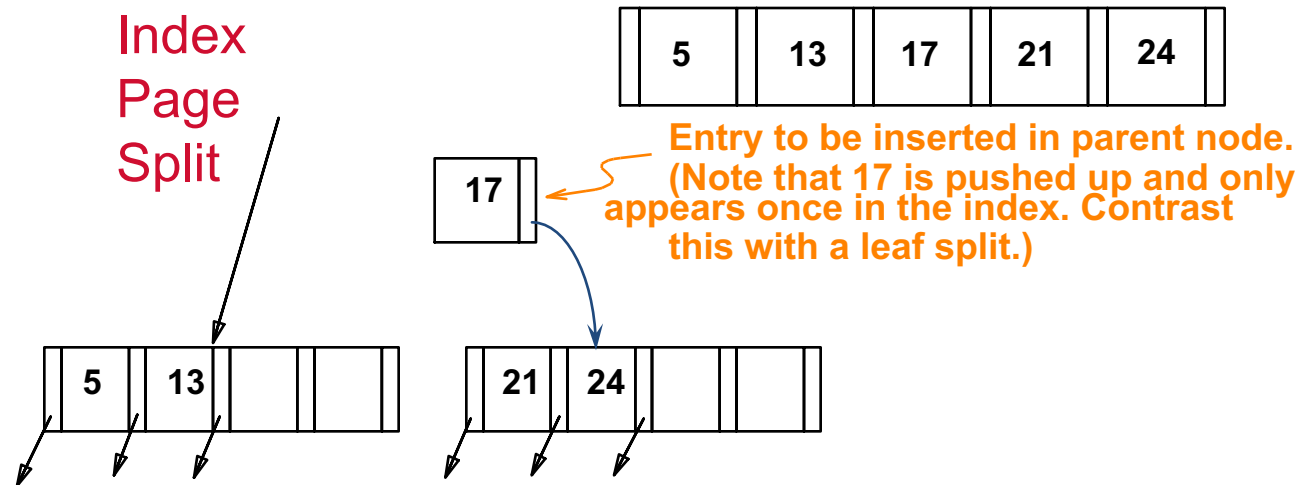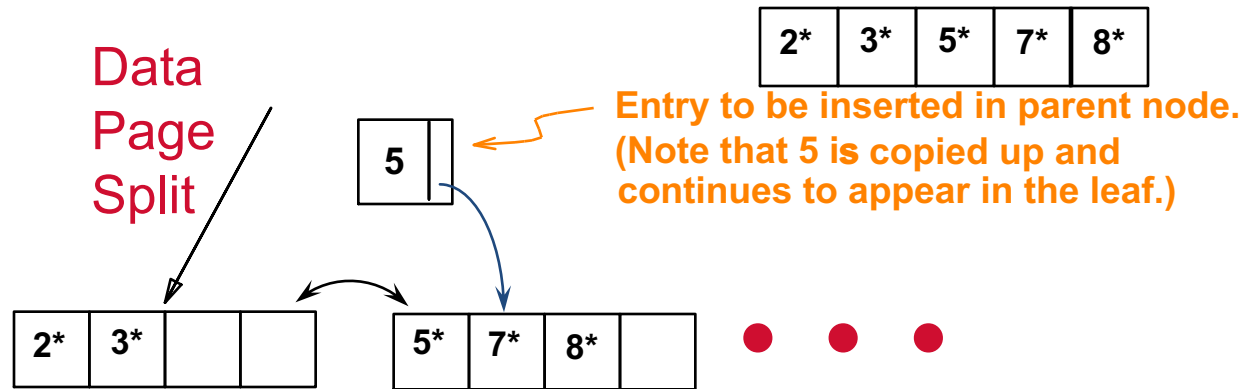
In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.
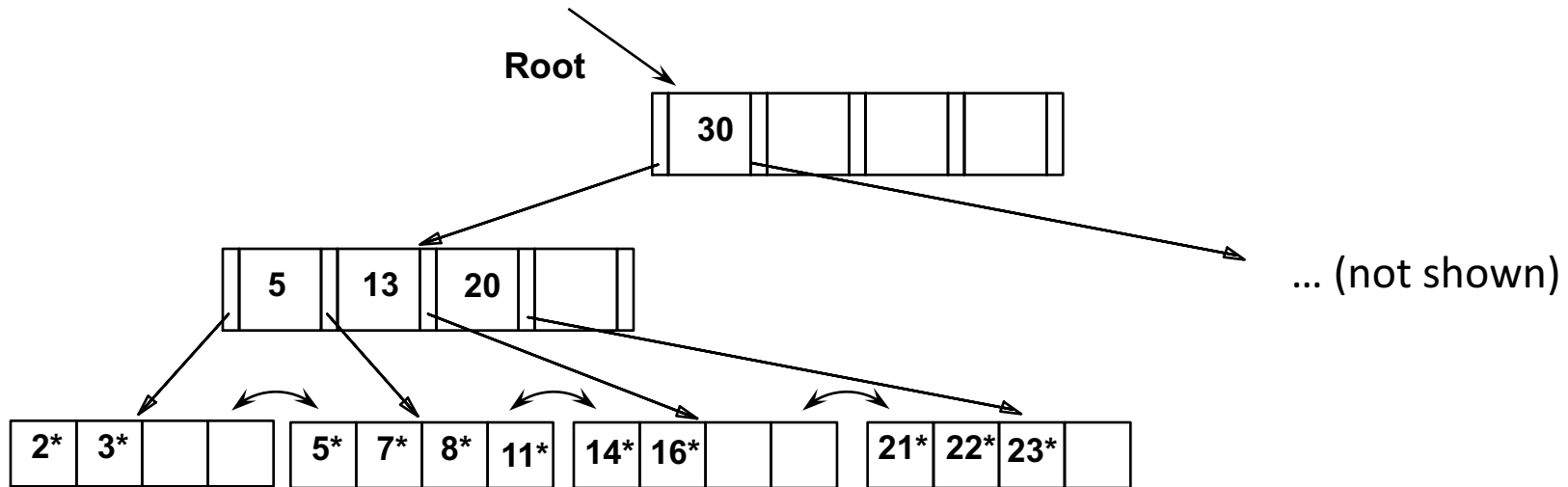
20

Example: Data vs. Index Page Split

minimum occupancy is guaranteed in both leaf and index page splits

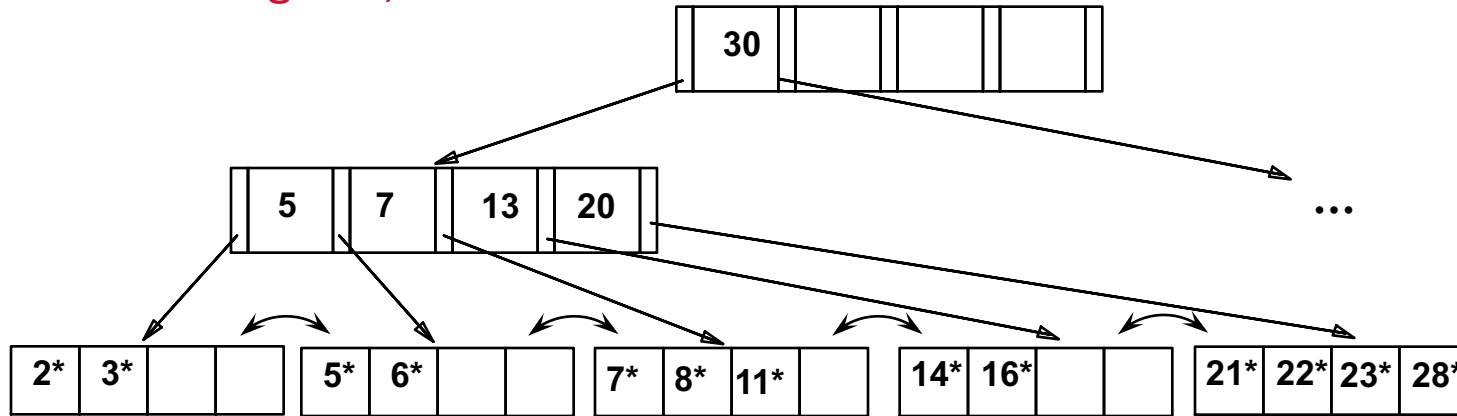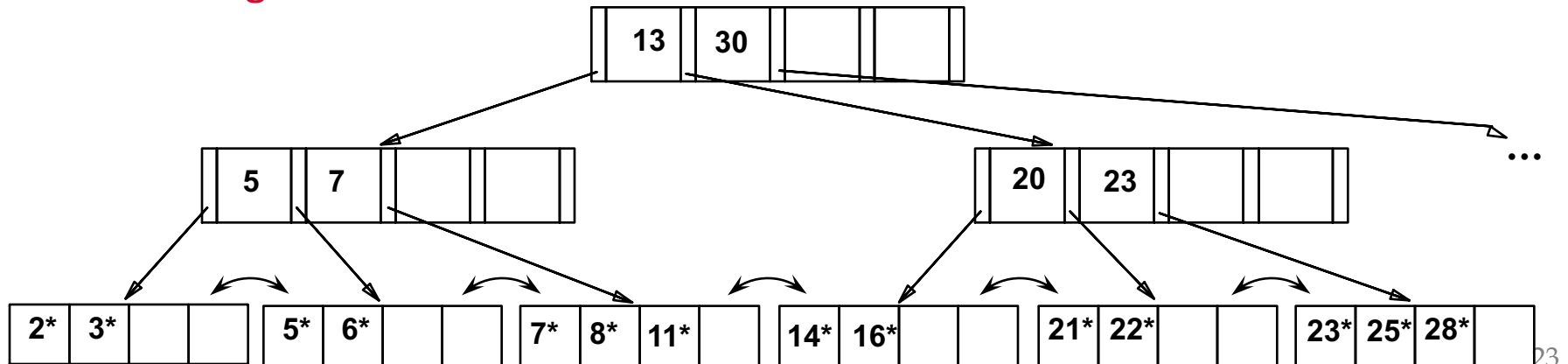*copy-up* for data page splits

push-up for index page split

Data Page Split

2*	3*	5*	7*	8*

5	

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

2*	3*		

5*	7*	8*	

● ● ●

Index Page Split

5	13	17	21	24

17	

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.)

5	13		

21	24		

Now you try…

Root

| | 30 | | | |

… (not shown)

| | 5 | | 13 | 20 | | |

| 2* | 3* | | |

| 5* | 7* | 8* | 11* |

| 14* | 16* | | |

| 21* | 22* | 23* | |

Insert the following data entries (in order): 28*, 6*, 25*

Answer…

After inserting 28*, 6*

| | 30 | | | | | |

| | 5 | 7 | 13 | 20 | |

...

| 2* | 3* | | | | 5* | 6* | | | | 7* | 8* | 11* | | 14* | 16* | | | | 21* | 22* | 23* | 28* |

After inserting 25*

| | 13 | 30 | | | | |

...

| | 5 | 7 | | | | | | 20 | 23 | | | |

| 2* | 3* | | | | 5* | 6* | | | | 7* | 8* | 11* | | 14* | 16* | | | | 21* | 22* | | | | 23* | 25* | 28* | |

23

Tree-structured indexing

Intro & B⁺-Tree

Insert into a B⁺-Tree

Delete from a B⁺-Tree

Prefix Key Compression & Bulk Loading

Units

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).*
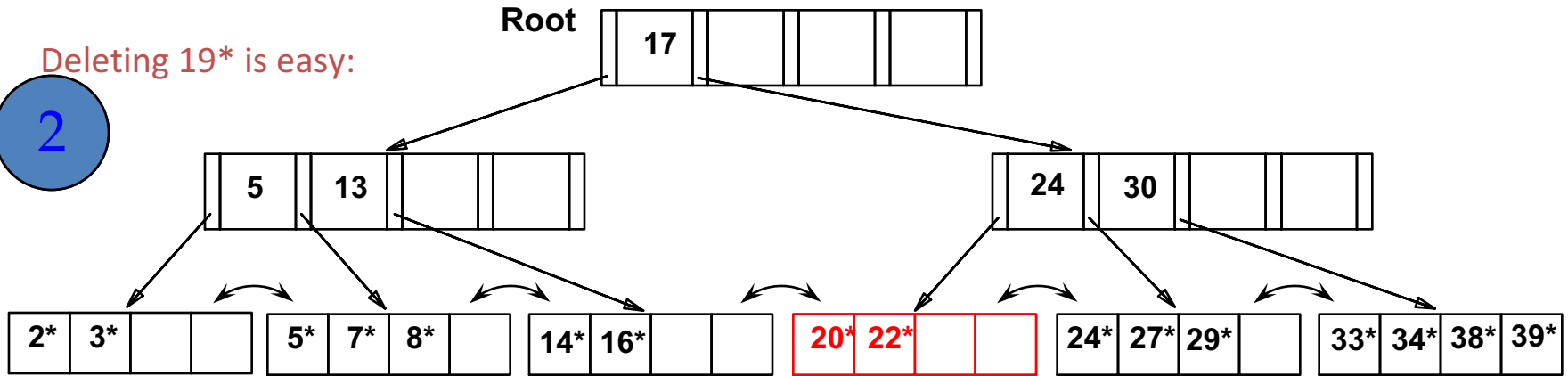 - If re-distribution fails, *merge* L and sibling.

If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
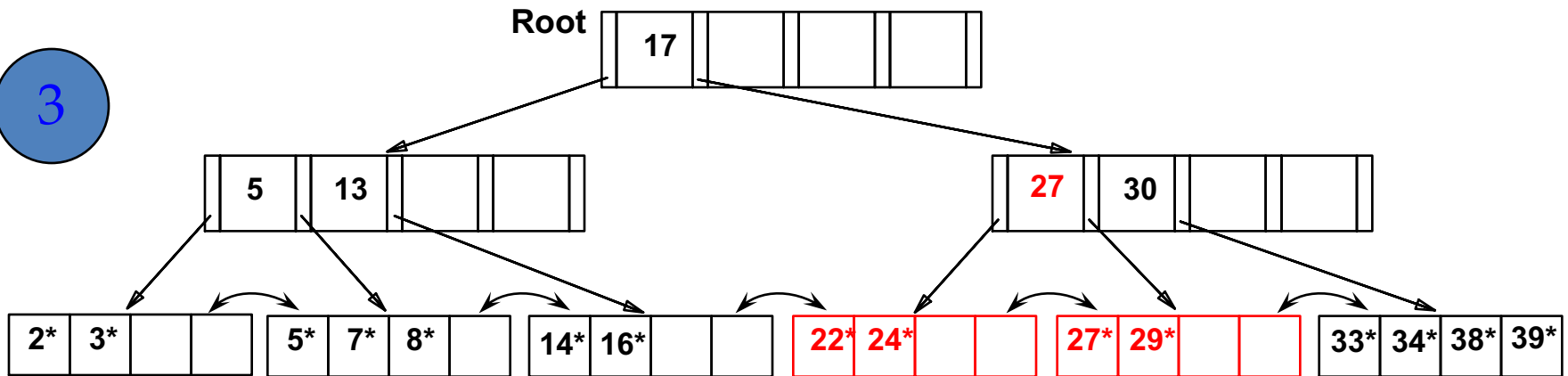
Merge could propagate to root, decreasing height.

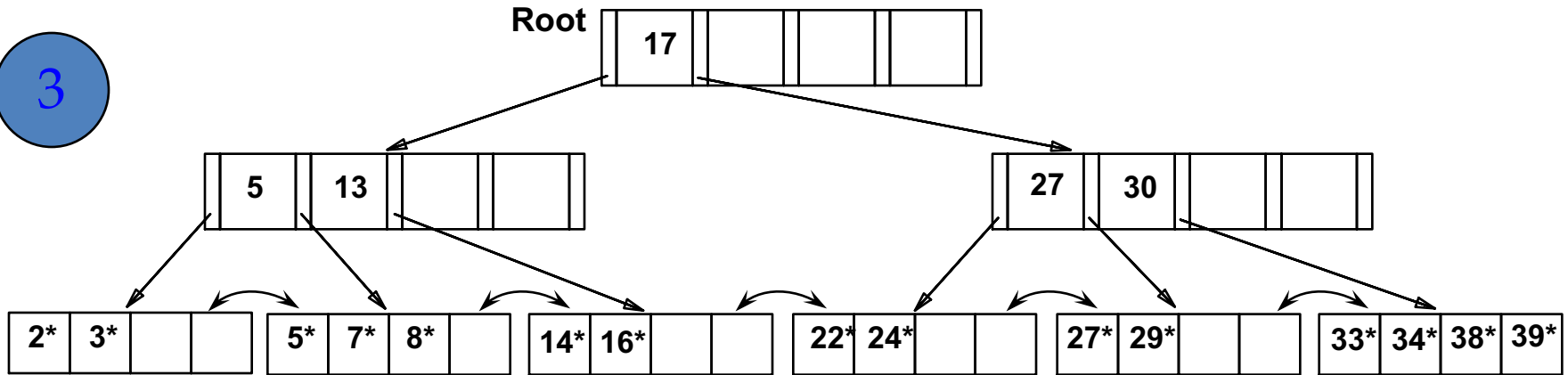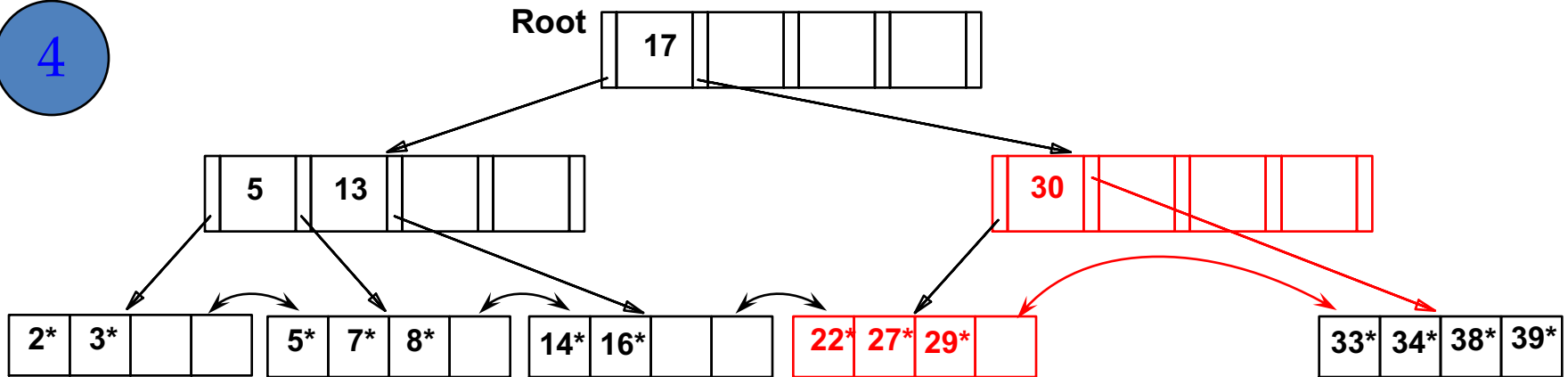# Example: Delete 19* & 20*

Deleting 19* is easy:

2

Root | 17

5 | 13

24 | 30

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

3

Root | 17

5 | 13

27 | 30

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

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

… and then deleting 24*

3

Root

| 17 | | | |

| 5 | 13 | | |

| 27 | 30 | | |

| 2* | 3* | | |

| 5* | 7* | 8* | |

| 14* | 16* | | |

| 22* | 24* | | |

| 27* | 29* | | |

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

4

Root

| 17 | | | |

| 5 | 13 | | |

| 30 | | | | |

| 2* | 3* | | |

| 5* | 7* | 8* | |

| 14* | 16* | | |

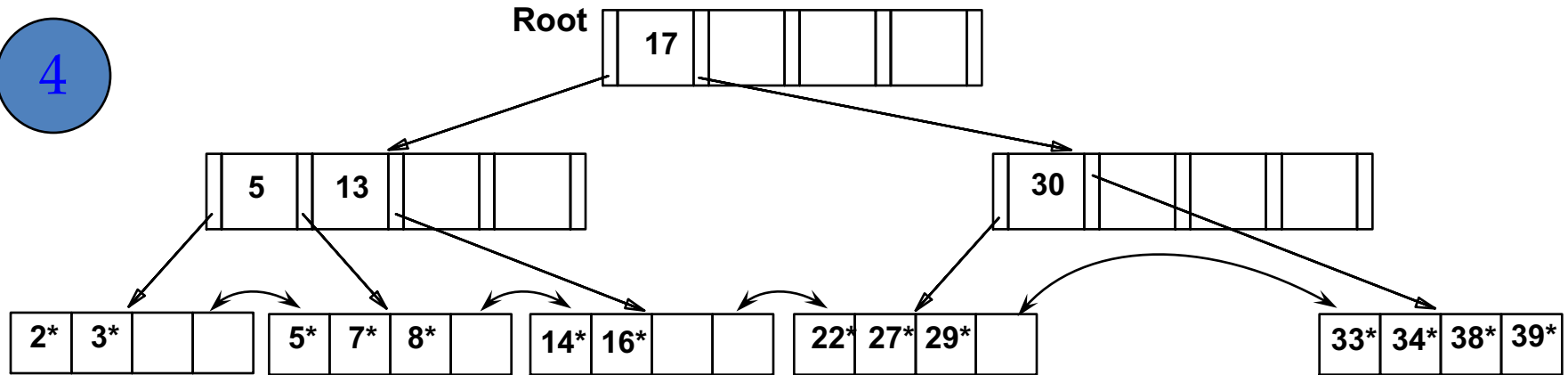| 22* | 27* | 29* | |

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

Must merge leaves   **… but are we done??**

... merge non-leaf nodes, shrink tree

4

Root | 17 | | | |

| 5 | 13 | | |

| 30 | | | |

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

5

Root | 5 | 13 | 17 | 30 |

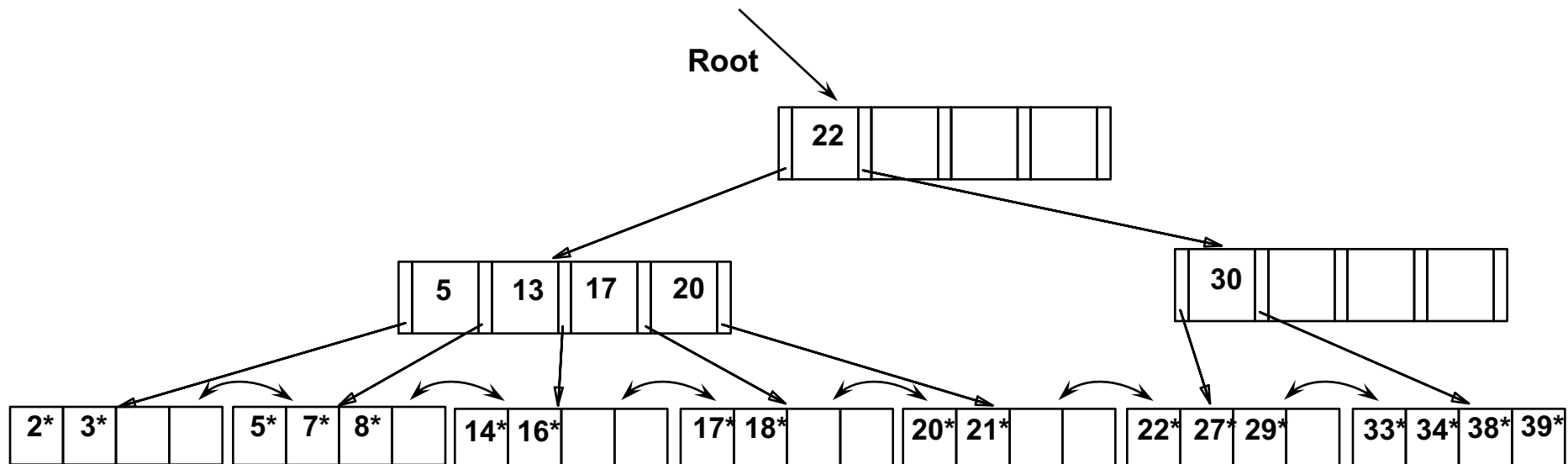| 2* | 3* | | | | 5* | 7* | 8* | | | 14* | 16* | | | | 22* | 27* | 29* | | | 33* | 34* | 38* | 39* |
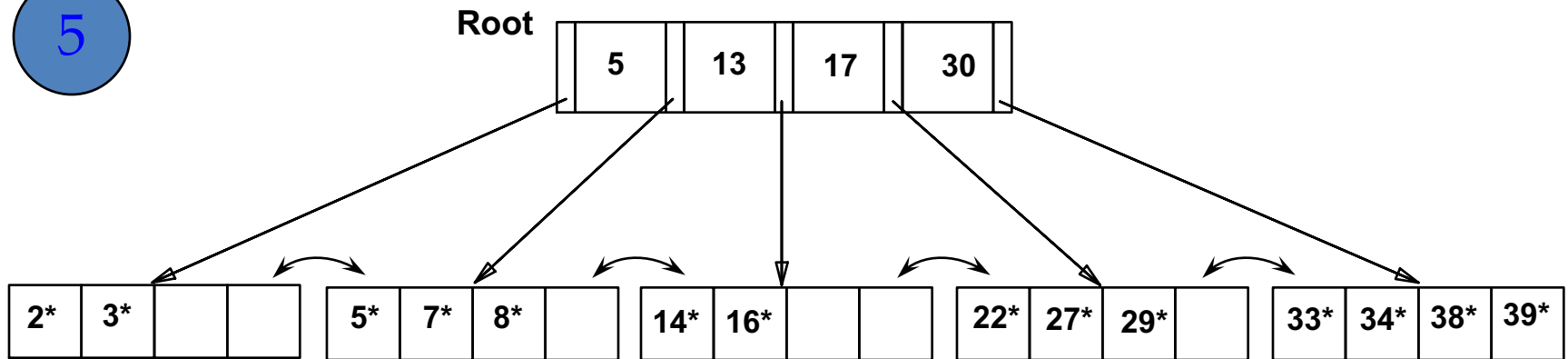
Example of non-leaf re-distribution

Tree is shown below *during deletion* of 24*.
What could be a possible initial tree?

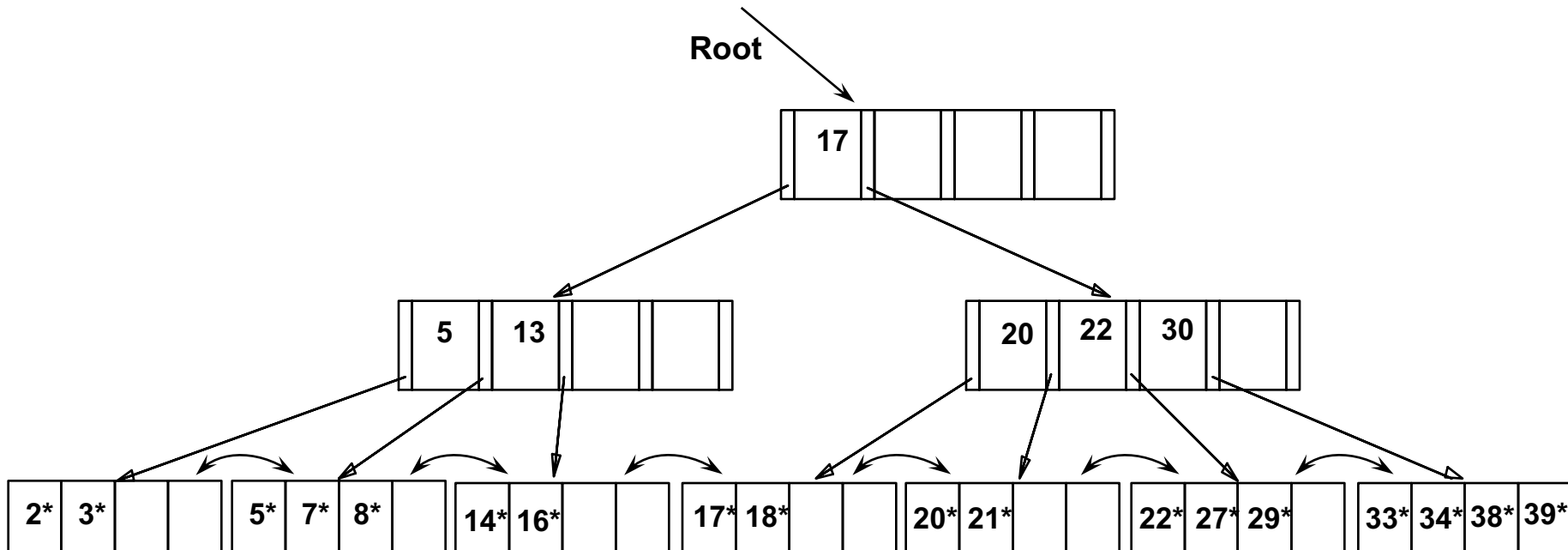In contrast to previous example, can re-distribute entry from left child of root to right child.

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* |

29

After Re-distribution

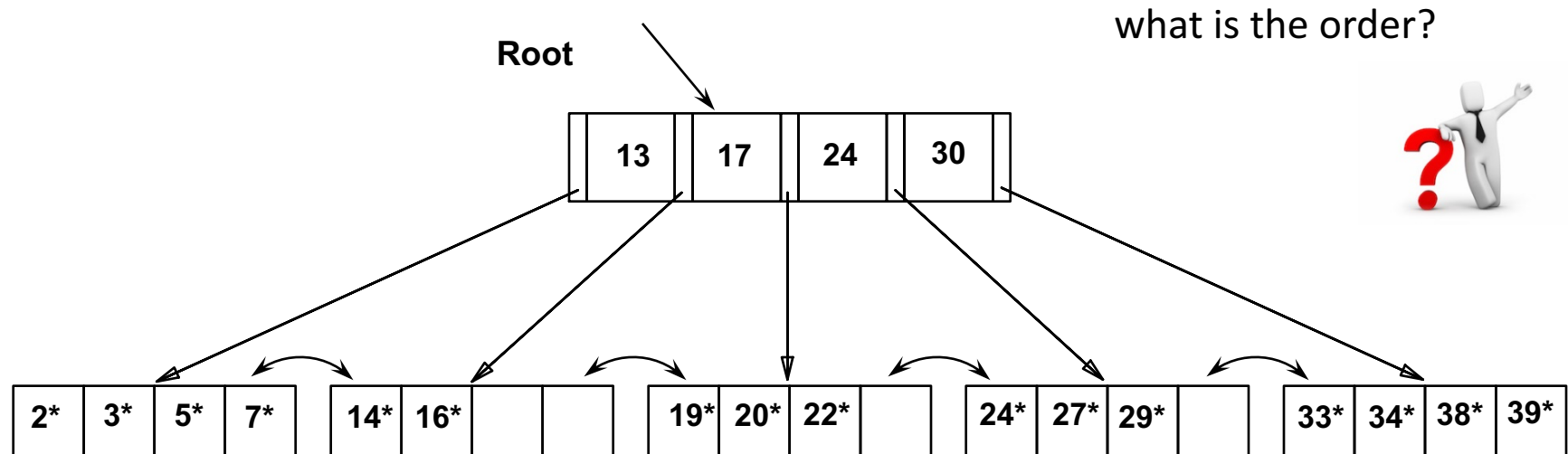Intuitively, entries are re-distributed by "*pushing through*" the splitting entry in the parent node.

[it suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration]

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* |

Reminders

begin at root, compare keys to reach the leaf
"order" *d* means d to 2*d elements

what is the order?

Root

13	17	24	30

2*	3*	5*	7*

14*	16*		

19*	20*	22*	

24*	27*	29*	

33*	34*	38*	39*

31

Tree-structured indexing

Intro & B$^+$-Tree

Insert into a B$^+$-Tree

Delete from a B$^+$-Tree

Prefix Key Compression & Bulk Loading

Units

Prefix Key Compression

why?

we want to increase fan-out

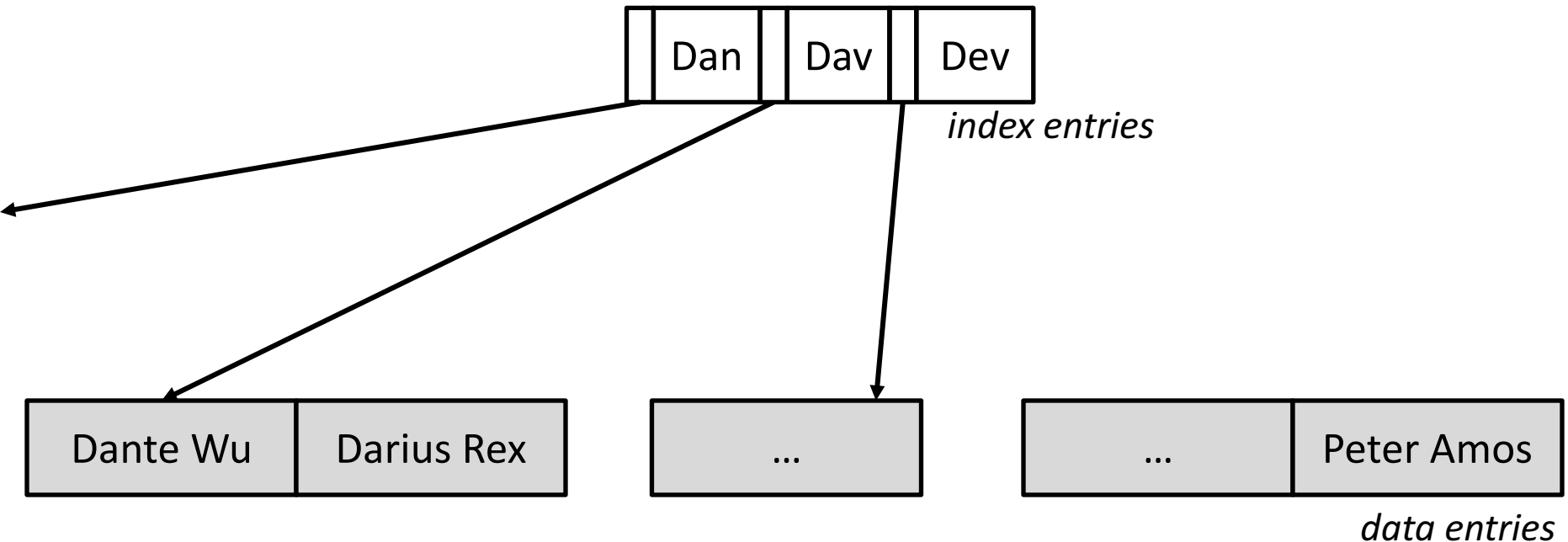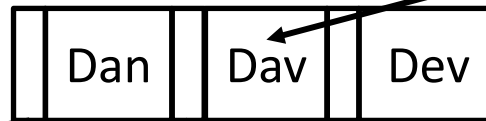key values in index entries (internal nodes) are used to "direct traffic"

	Daniel Lee		Davey Smith		Devarakonda ...

index entries

Dante Wu	Darius Rex

...

...	Peter Amos

data entries

33

Prefix Key Compression

we want to increase fan-out

why?

key values in index entries (internal nodes) are used to "direct traffic"

	Dan		Dav		Dev	

index entries

Dante Wu	Darius Rex		...		...	Peter Amos

data entries

34
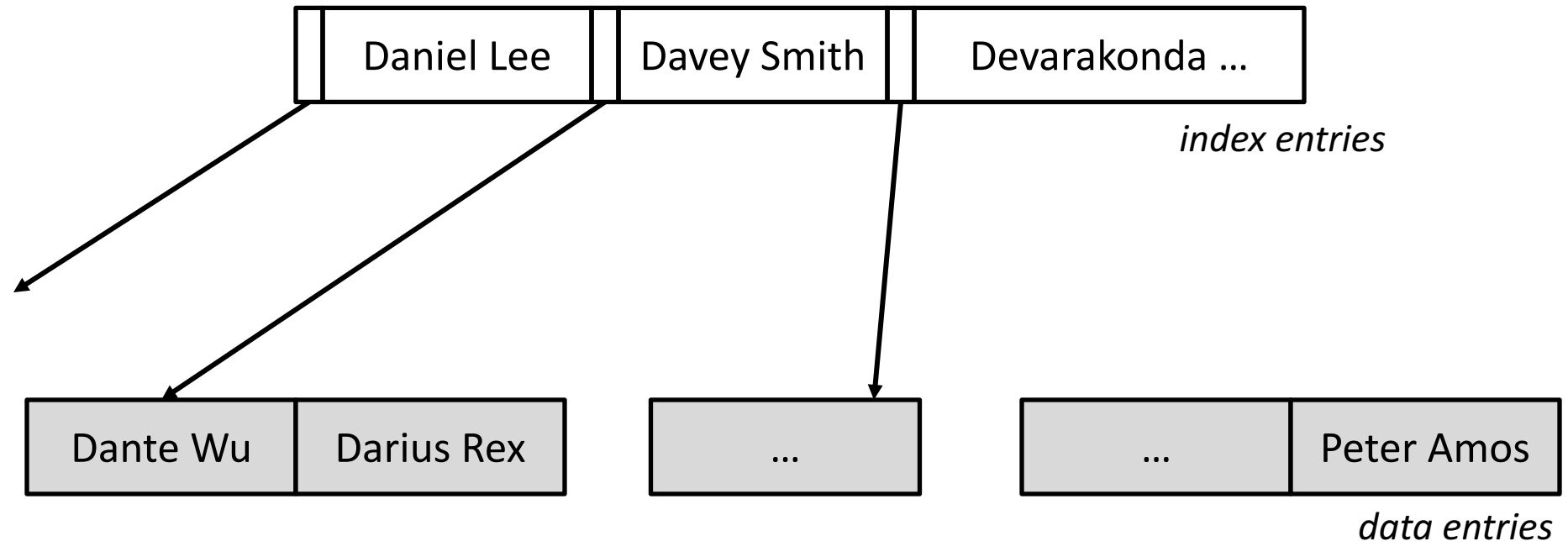
Prefix Key Compression

we want to increase fan-out   why?
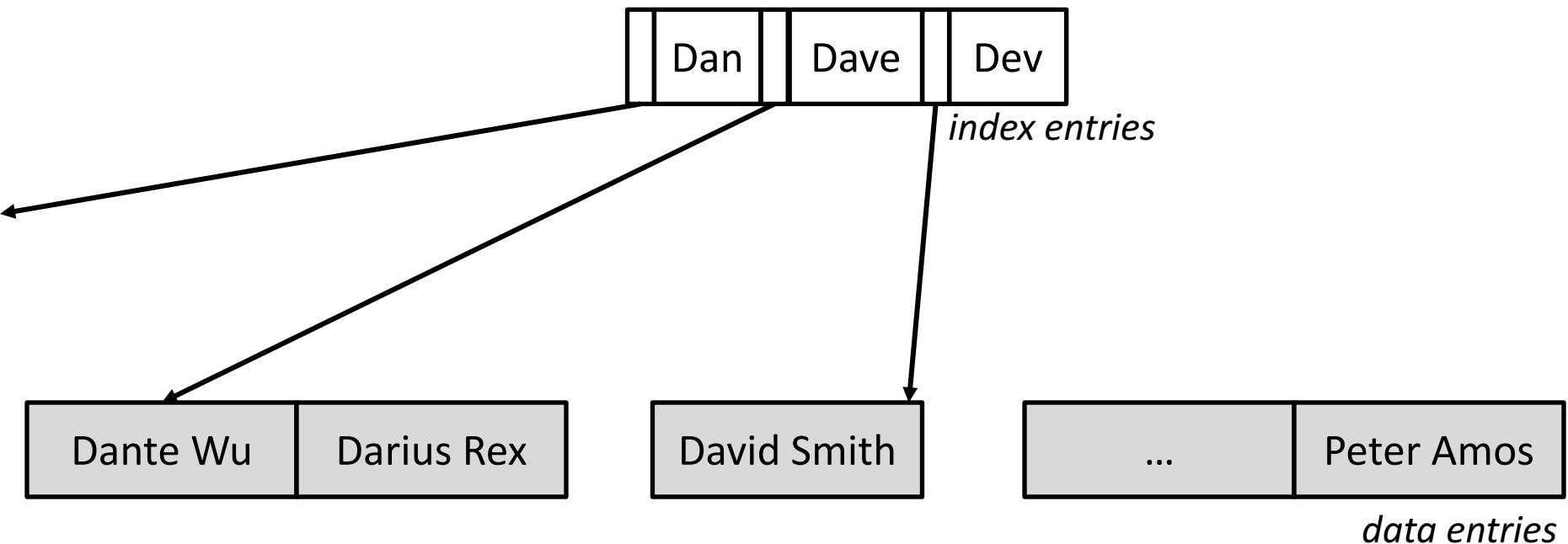
key values in index entries (internal nodes) are used to "direct traffic"

remember: Davey Smith
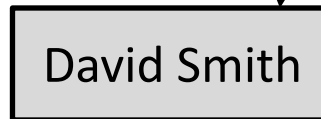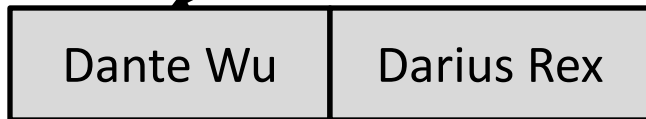
	Dan		Dav		Dev

index entries

is it ok now?

Dante Wu	Darius Rex		David Smith		...	Peter Amos

data entries

35

Prefix Key Compression

we want to increase fan-out

why?

key values in index entries (internal nodes) are used to "direct traffic"

	Dan	Dave	Dev

index entries

Dante Wu	Darius Rex

David Smith

...	Peter Amos

data entries

36

Prefix Key Compression

we want to increase fan-out

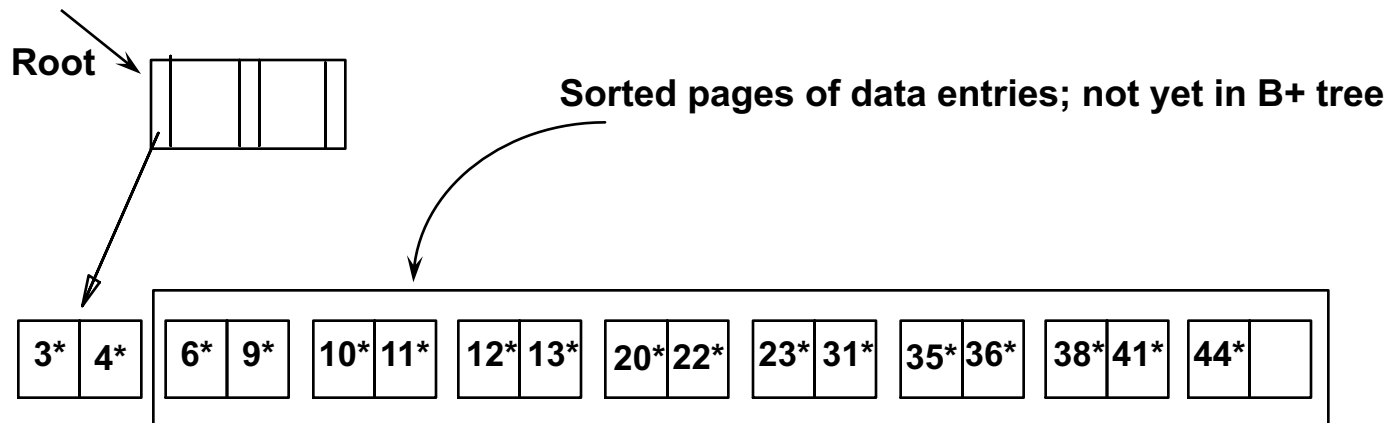key values in index entries (internal nodes) are used to "direct traffic"

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 <u>repeatedly inserting records is very slow</u>.
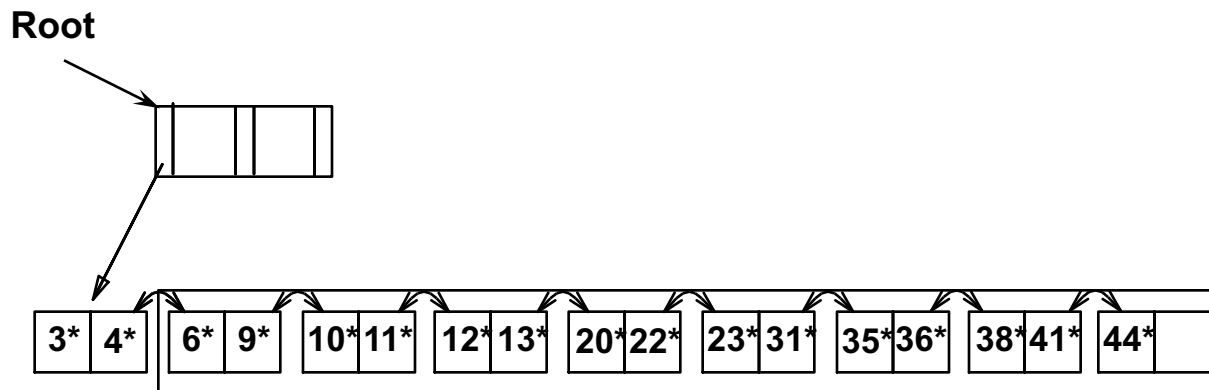
Bulk Loading can be done much more efficiently.

Initialization: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.

Root

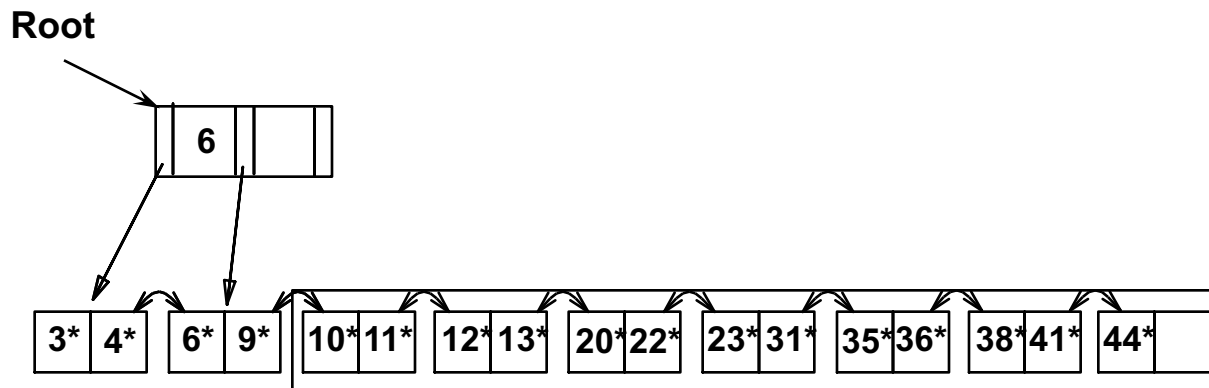**Sorted pages of data entries; not yet in B+ tree**

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

38

Bulk Loading (Contd.)

where to insert: into right-most index page just above leaf level
what to insert: the left-most value of the new leaf

Root

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

what to do when full? when this fills up, splits node
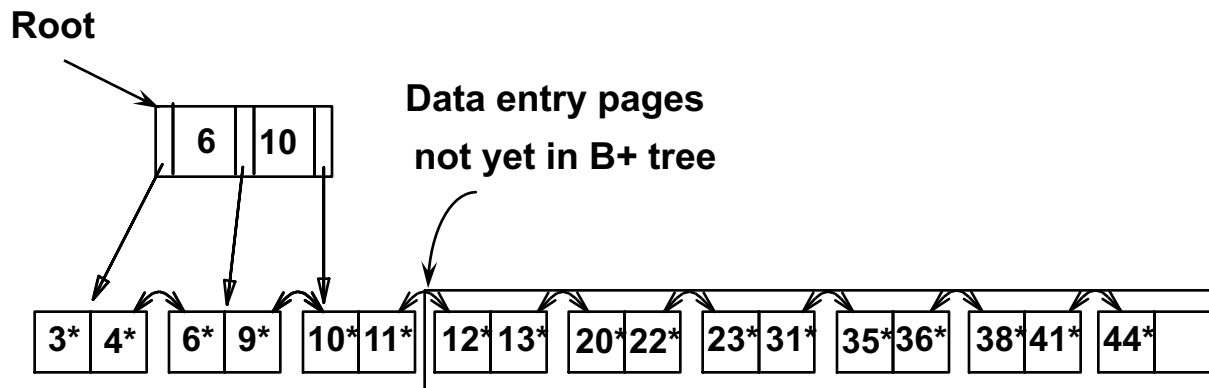(if needed split may go up right-most path to the root)

39

Bulk Loading (Contd.)

where to insert: into right-most index page just above leaf level
what to insert: the left-most value of the new leaf

Root

| | 6 | | |

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

what to do when full? when this fills up, splits node
(if needed split may go up right-most path to the root)

Bulk Loading (Contd.)

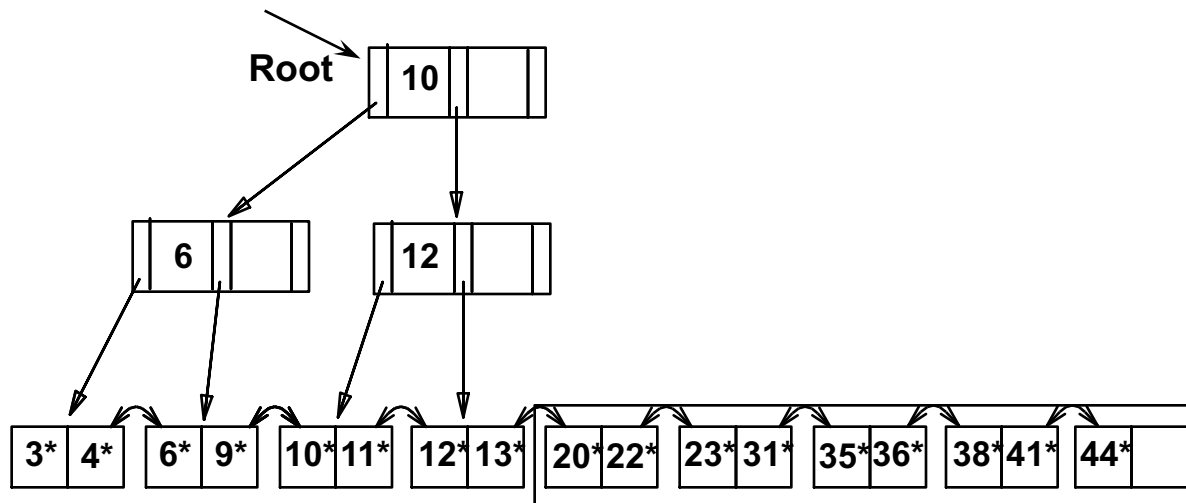**where to insert**: into right-most index page just above leaf level
what to insert: the left-most value of the new leaf

Root

Data entry pages
not yet in B+ tree

| 6 | 10 |

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

what to do when full? when this fills up, splits node
(if needed split may go up right-most path to the root)
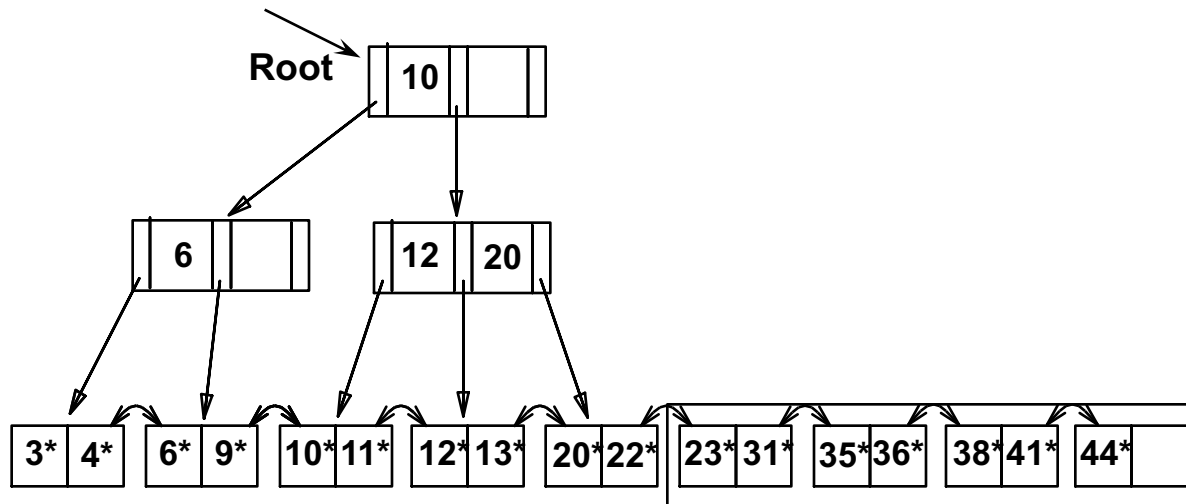
41

Bulk Loading (Contd.)

where to insert: into right-most index page just above leaf level
what to insert: the left-most value of the new leaf

what to do when full? when this fills up, splits node
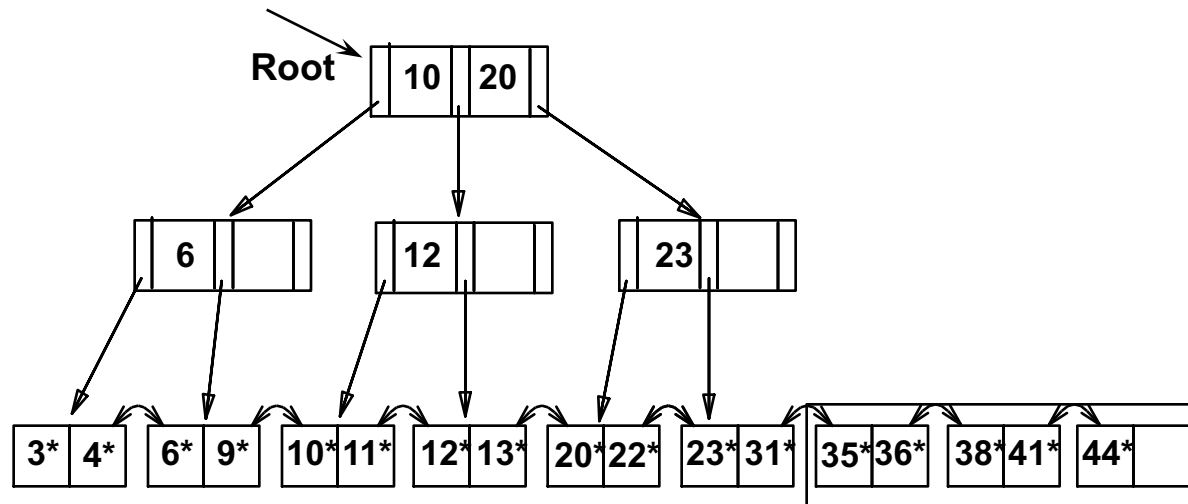(if needed split may go up right-most path to the root)

42

Bulk Loading (Contd.)

where to insert: into right-most index page just above leaf level
what to insert: the left-most value of the new leaf

what to do when full? when this fills up, splits node
(if needed split may go up right-most path to the root)

43

Bulk Loading (Contd.)

where to insert: into right-most index page just above leaf level
what to insert: the left-most value of the new leaf

what to do when full? when this fills up, splits node
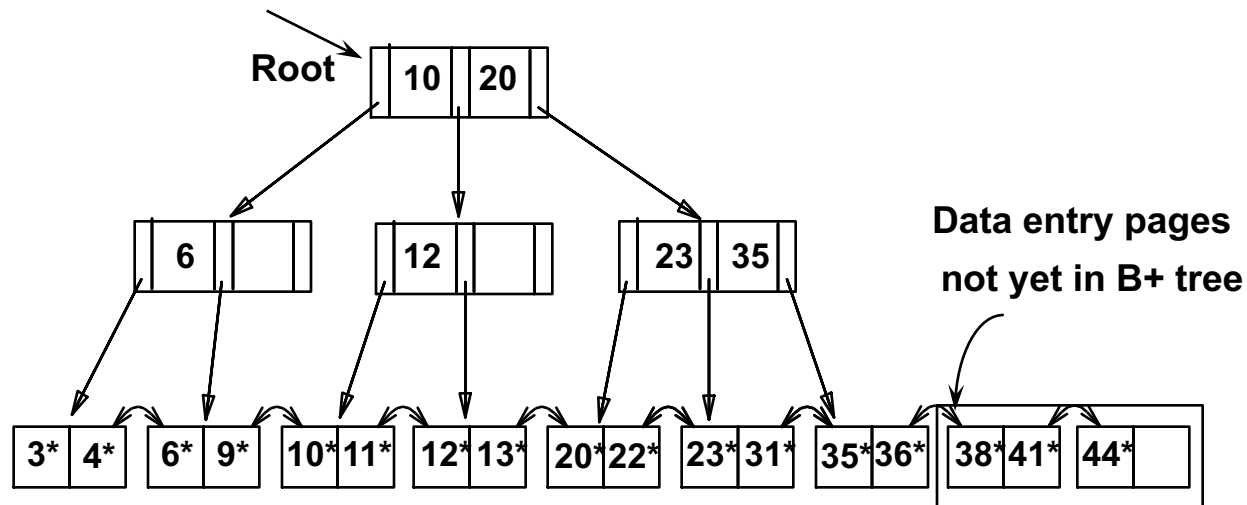(if needed split may go up right-most path to the root)

44

Bulk Loading (Contd.)

where to insert: into right-most index page just above leaf level
what to insert: the left-most value of the new leaf

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* | |

what to do when full? when this fills up, splits node
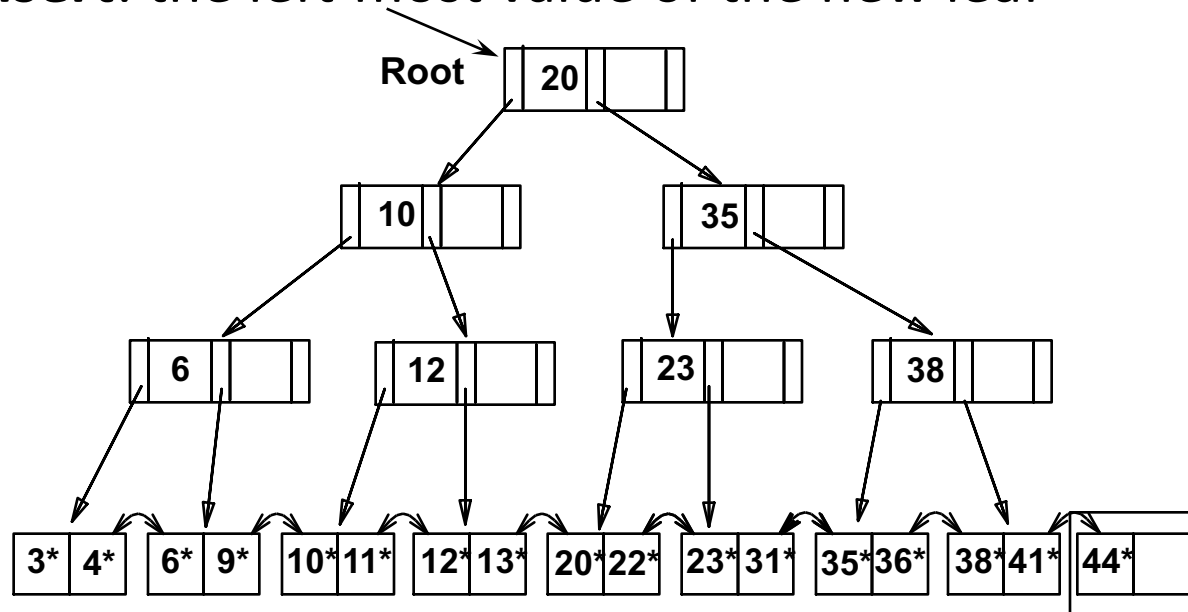(if needed split may go up right-most path to the root)

Bulk Loading (Contd.)

where to insert: into right-most index page just above leaf level
what to insert: the left-most value of the new leaf

what to do when full? when this fills up, splits node
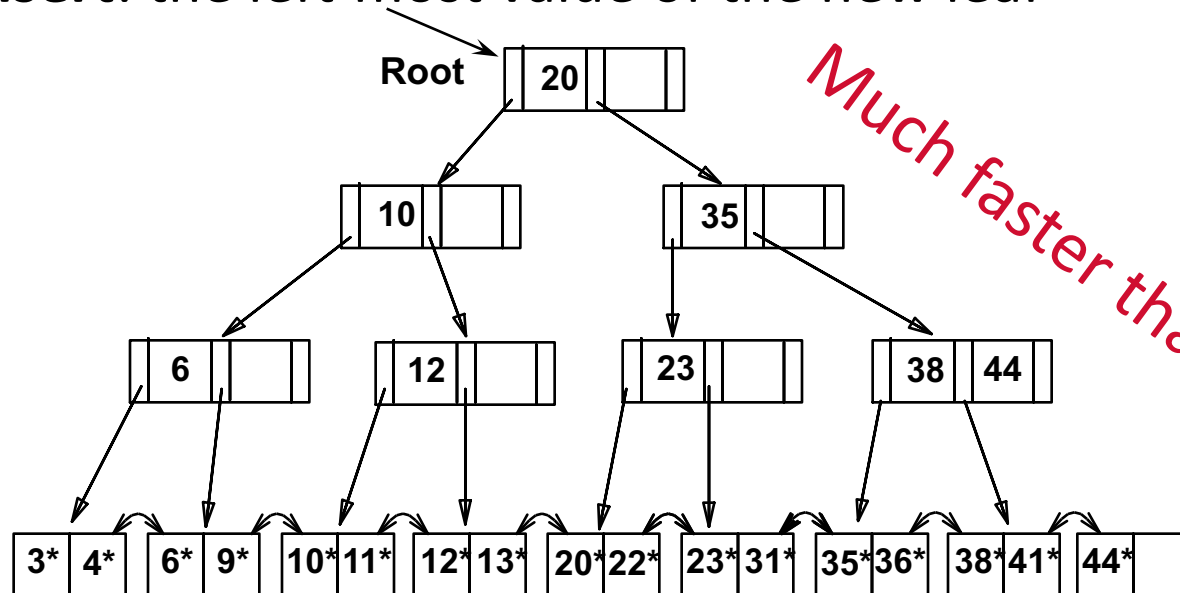(if needed split may go up right-most path to the root)

46

Bulk Loading (Contd.)

where to insert: into right-most index page just above leaf level
what to insert: the left-most value of the new leaf

Much faster than inserts!!!

what to do when full? when this fills up, splits node
(if needed split may go up right-most path to the root)

Summary of Loading Options

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) 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 different 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)).

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.

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.
- – Typically, 67% occupancy on average.
- – If data entries are data records, splits can change rids!

B+ Trees

"It could be said that the world's information is at our fingertips because of B-trees"

Goetz Graefe

Google (prev. Microsoft, HP Fellow)
ACM Software System Award

51