

# CS460: Intro to Database Systems

## Class 10: Tree-Structured Indexing

Instructor: Manos Athanassoulis

<https://bu-disc.github.io/CS460/>

# Tree-structured indexing

Intro & B<sup>+</sup>-Tree

Insert into a B<sup>+</sup>-Tree

Delete from a B<sup>+</sup>-Tree

Prefix Key Compression & Bulk Loading

# Introduction

*Recall: 3 alternatives for data entries  $k^*$ :*

- Data record with key value  $k$
- $\langle k, \text{rid of data record with search key value } k \rangle$
- $\langle k, \text{list of rids of data records with search key } k \rangle$

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

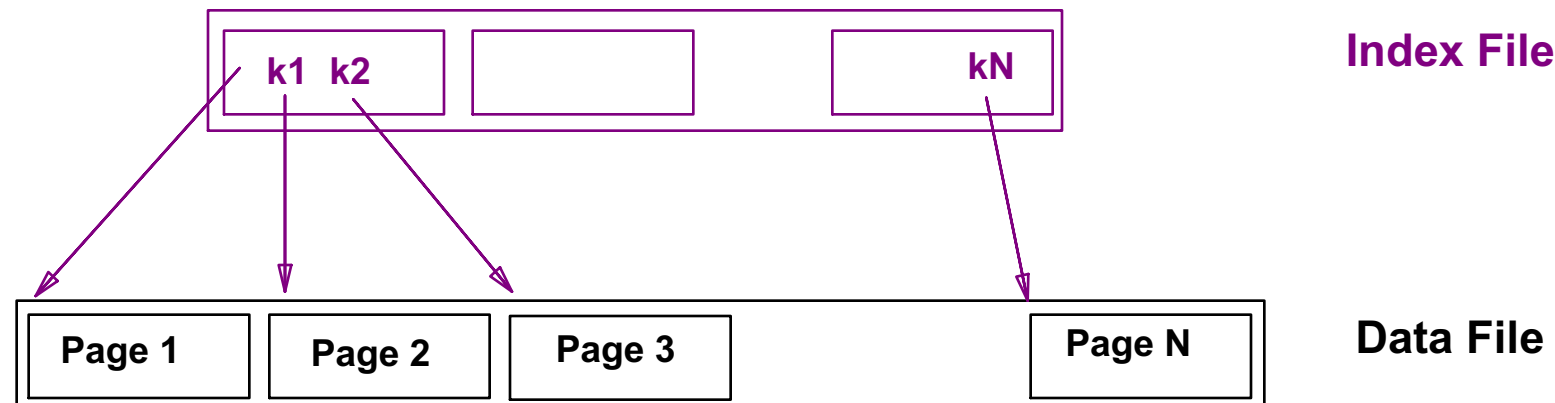
Tree-structured indexing techniques support both *range searches* and *equality searches*.

# 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.



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

# B+ Tree: The Most Widely-Used Index

Insert/delete at  $\log_F(N)$  cost; keep tree *height-balanced*.

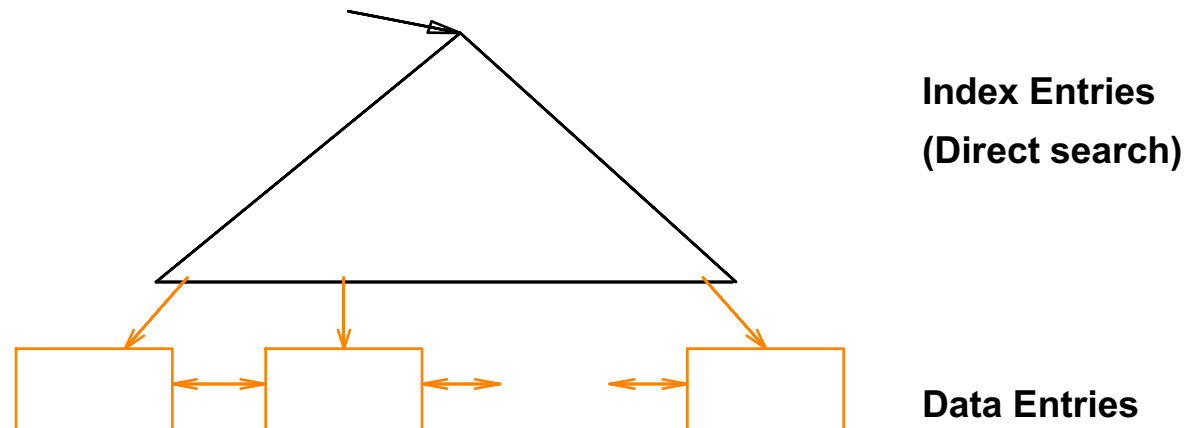
( $F$  = fanout,  $N$  = # leaf pages)

Minimum 50% occupancy (except for root).

Each node contains  $d \leq m \leq 2d$  entries. “ $d$ ” is called the *order* of the tree.

Supports equality and range-searches efficiently.

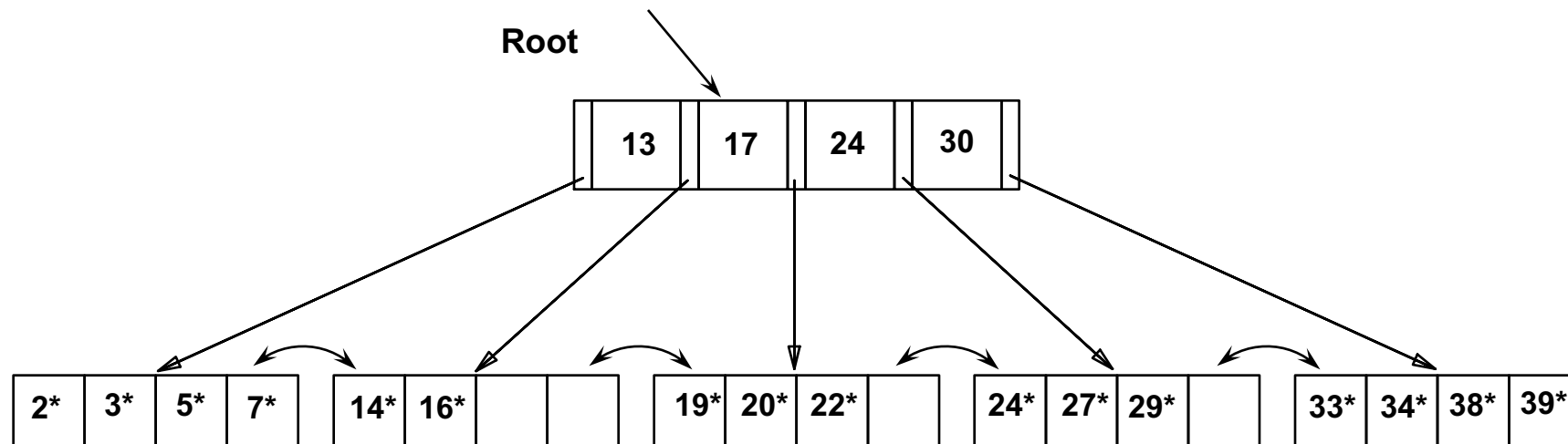
All searches go from root to leaves, in a *dynamic* structure.



# Example B+ Tree

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

Search for 5\*, 15\*, all data entries  $\geq 24^*$  ...



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

# B+ Trees in Practice (cool facts!)

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

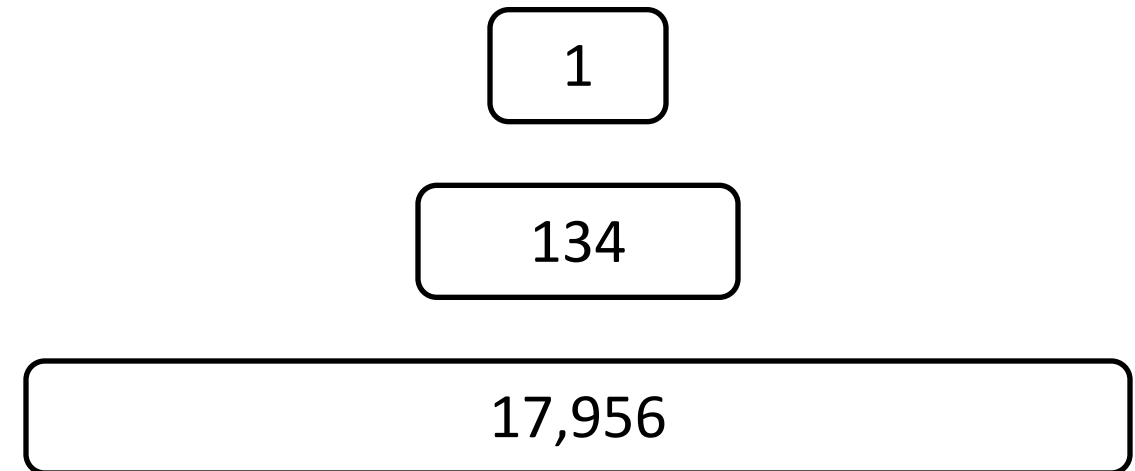
- average fanout =  $2 \cdot 100 \cdot 0.67 = 134$

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<sup>+</sup>-Tree

Insert into a B<sup>+</sup>-Tree

Delete from a B<sup>+</sup>-Tree

Prefix Key Compression & Bulk Loading



# 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  $L$ .

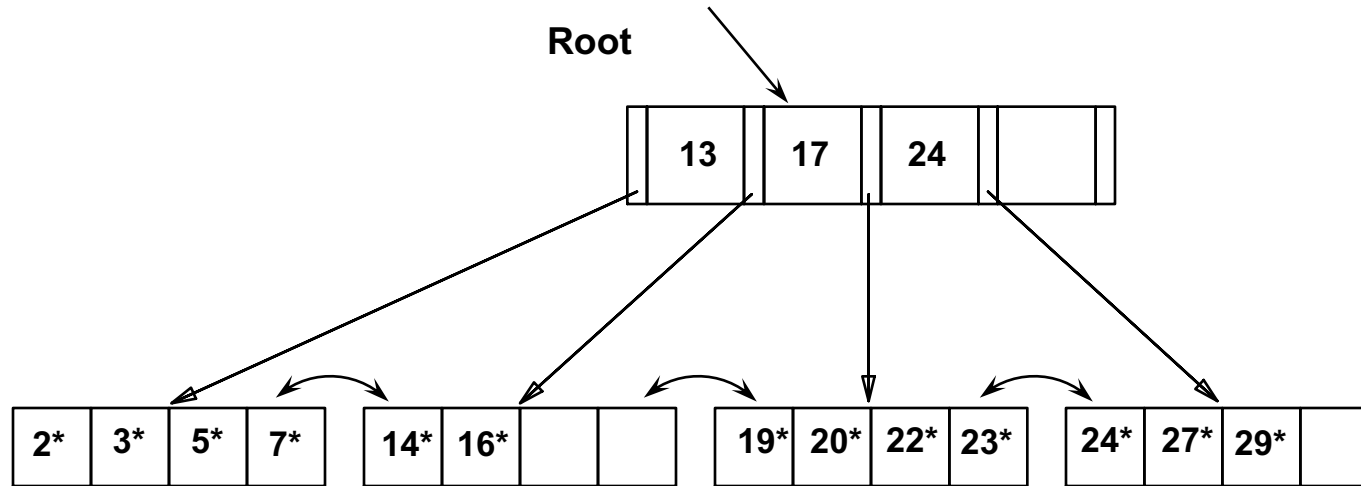
This can happen recursively

- To split index node, redistribute entries evenly, but push up middle key.  
(Contrast with leaf splits.)

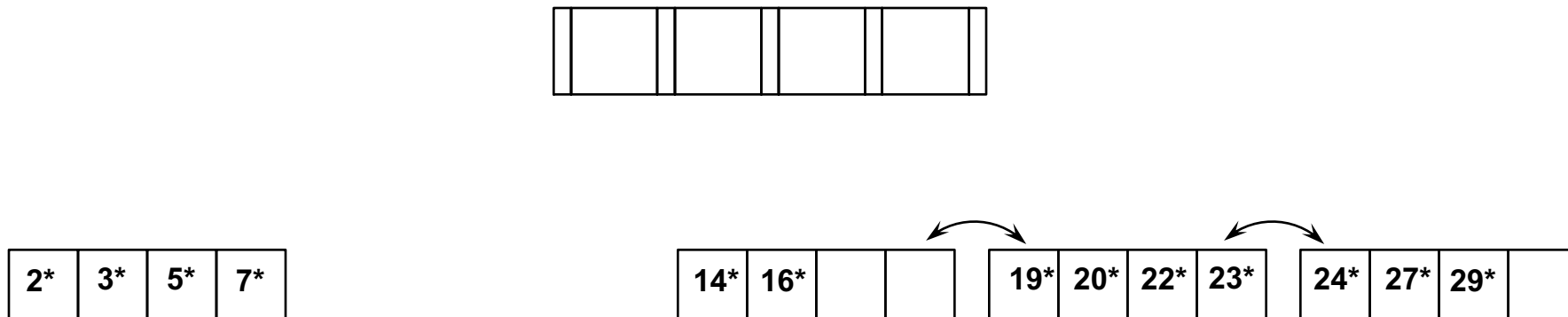
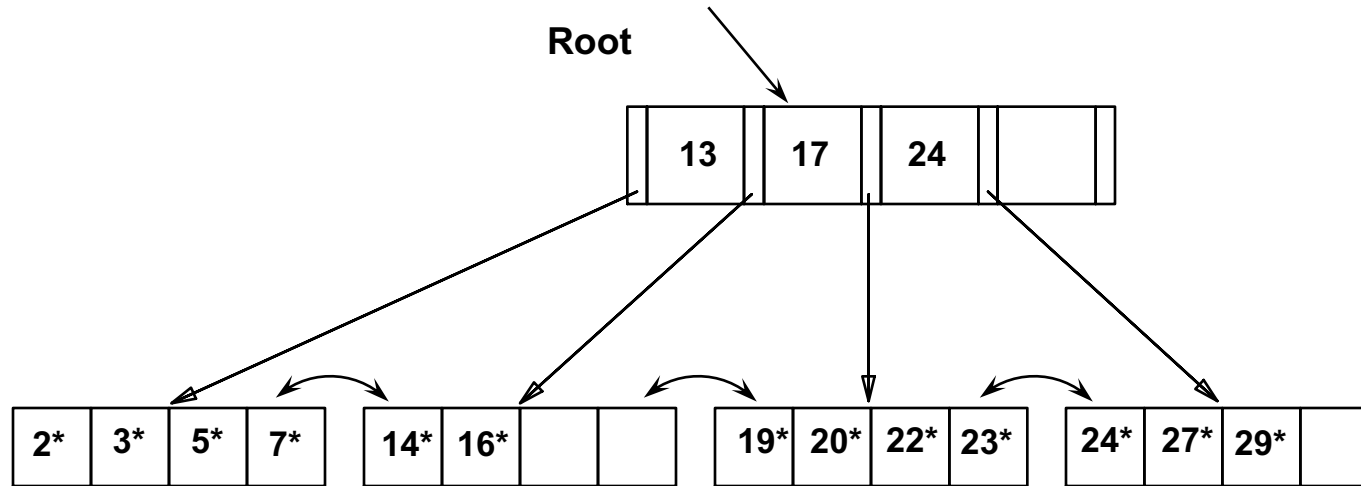
Splits “grow” tree; root split increases height.

- Tree growth: gets wider or one level taller at top.

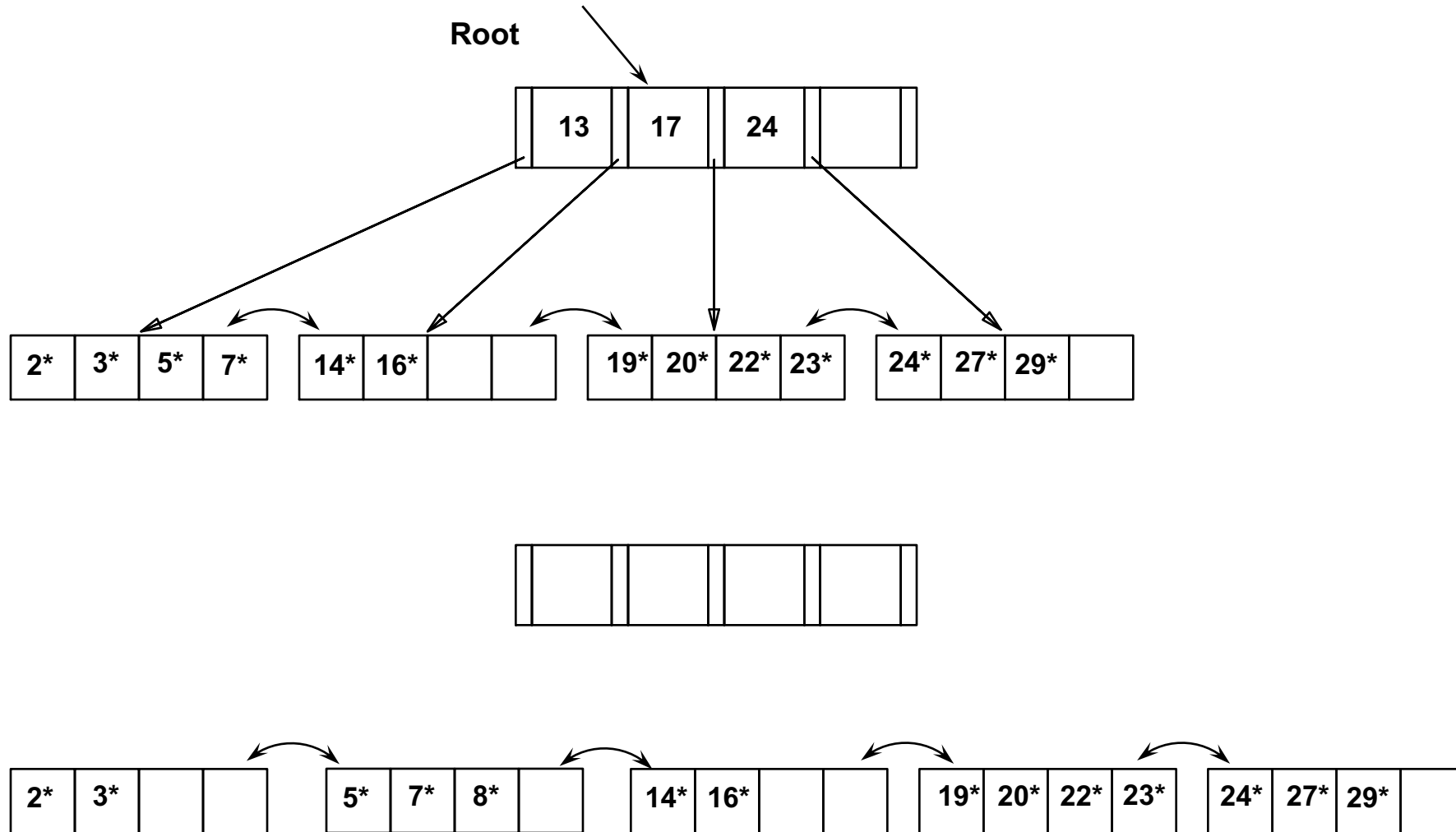
# Example B+ Tree - Inserting 8\*



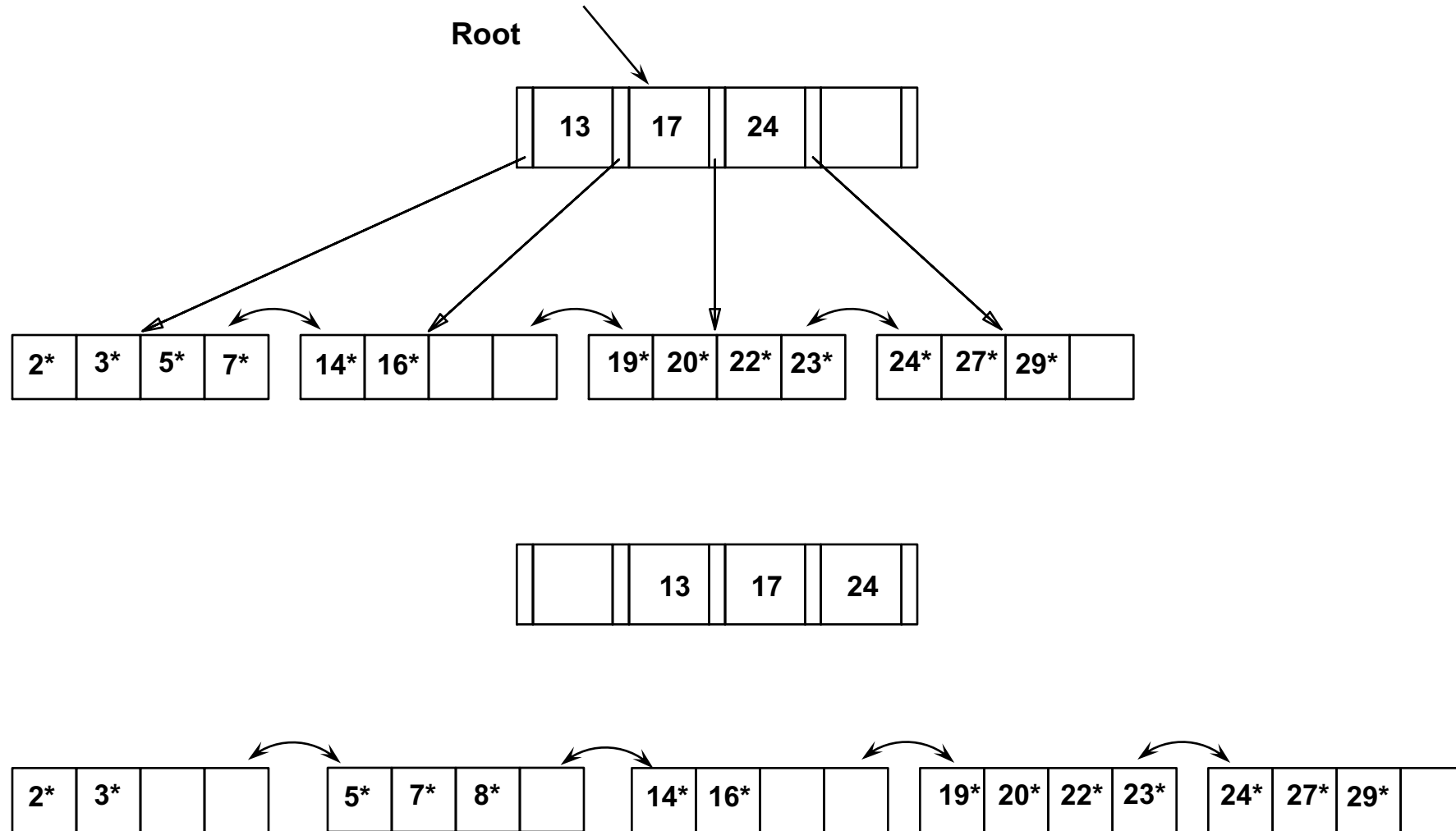
# Example B+ Tree - Inserting 8\*



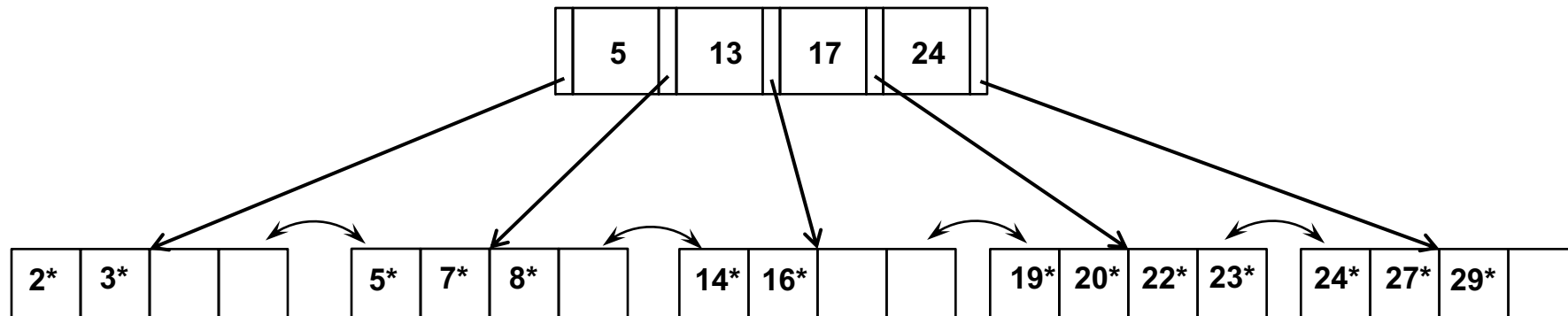
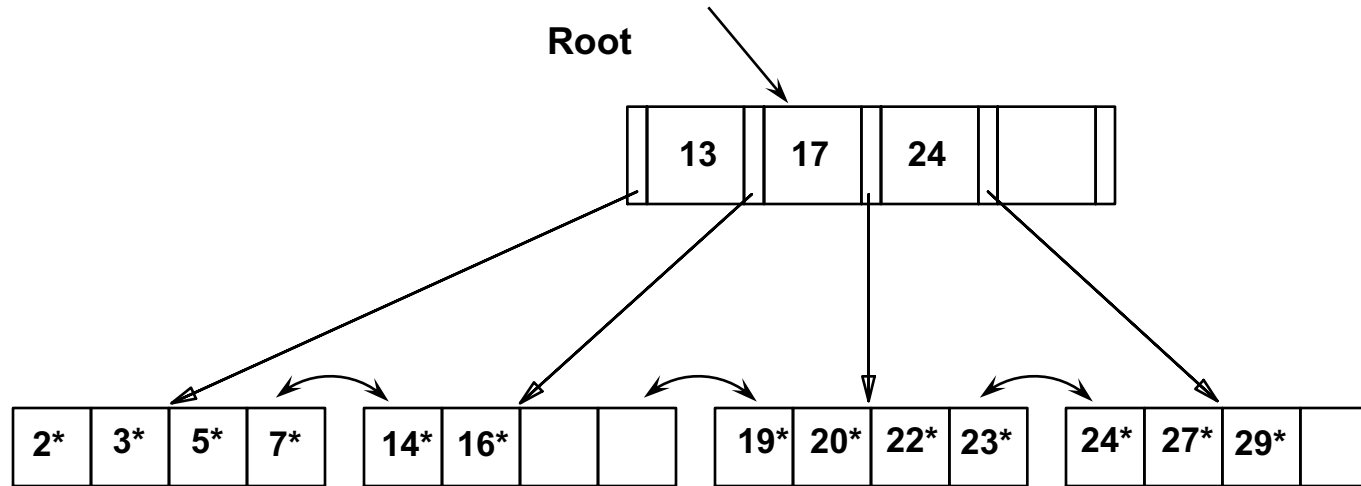
# Example B+ Tree - Inserting 8\*



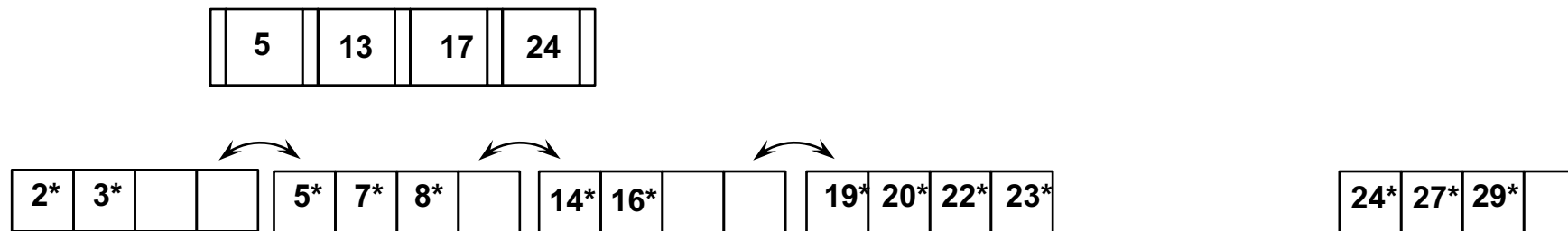
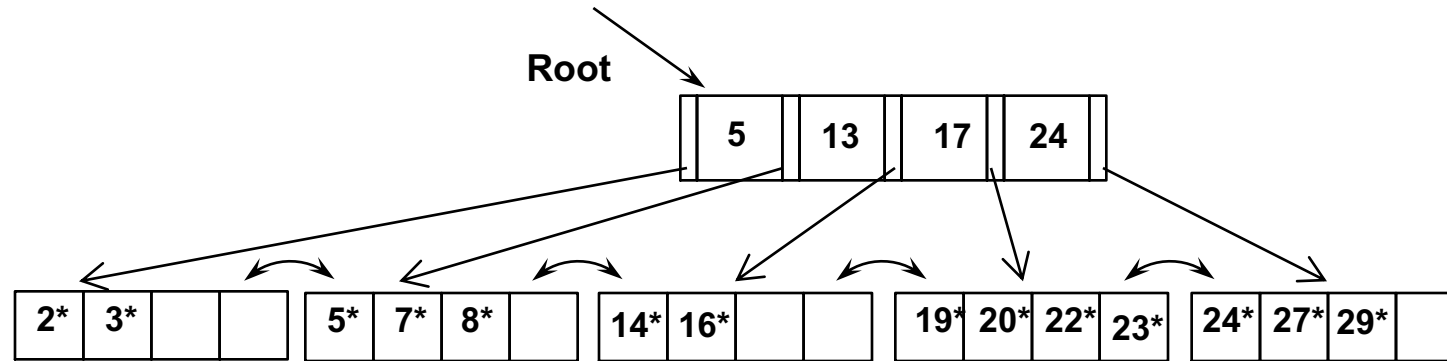
# Example B+ Tree - Inserting 8\*



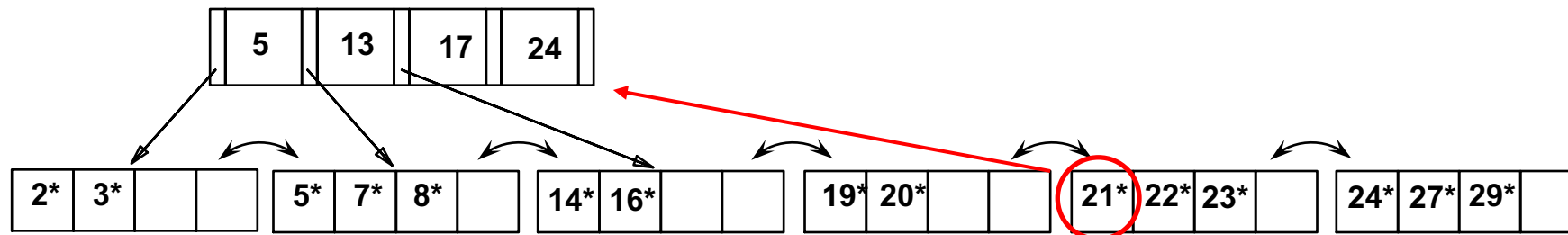
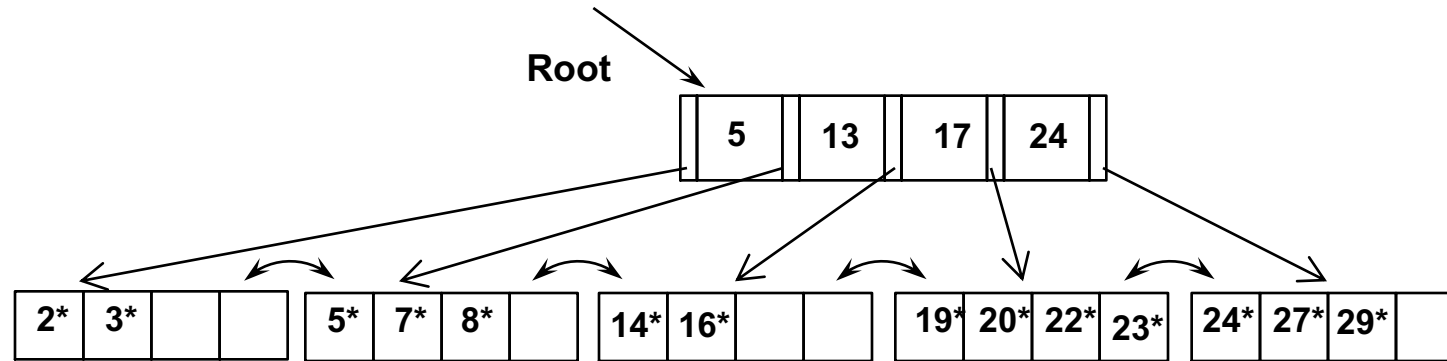
# Example B+ Tree - Inserting 8\*



# Example B+ Tree - Inserting 21\*

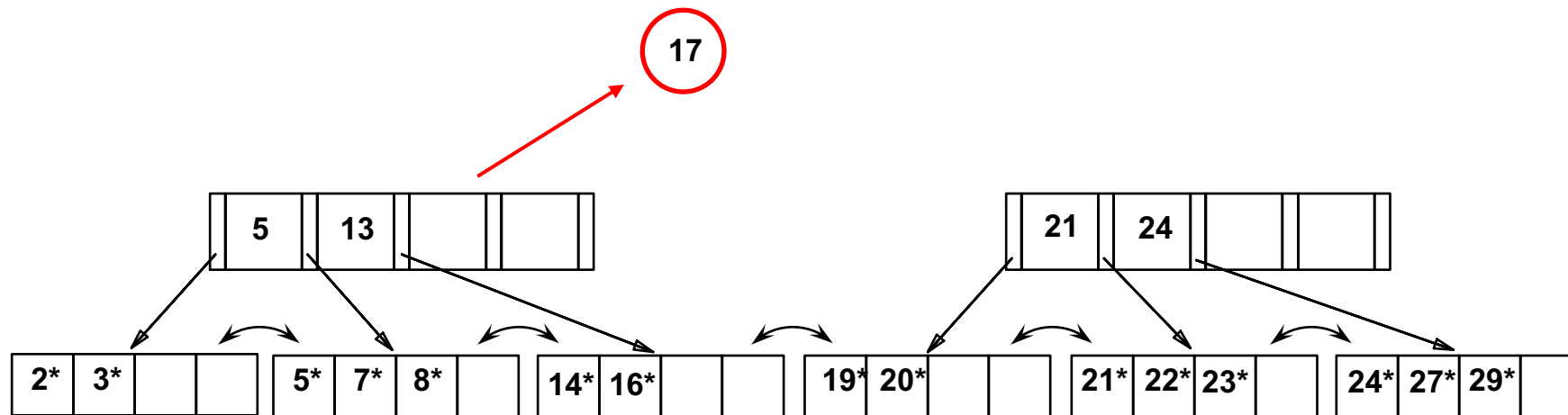
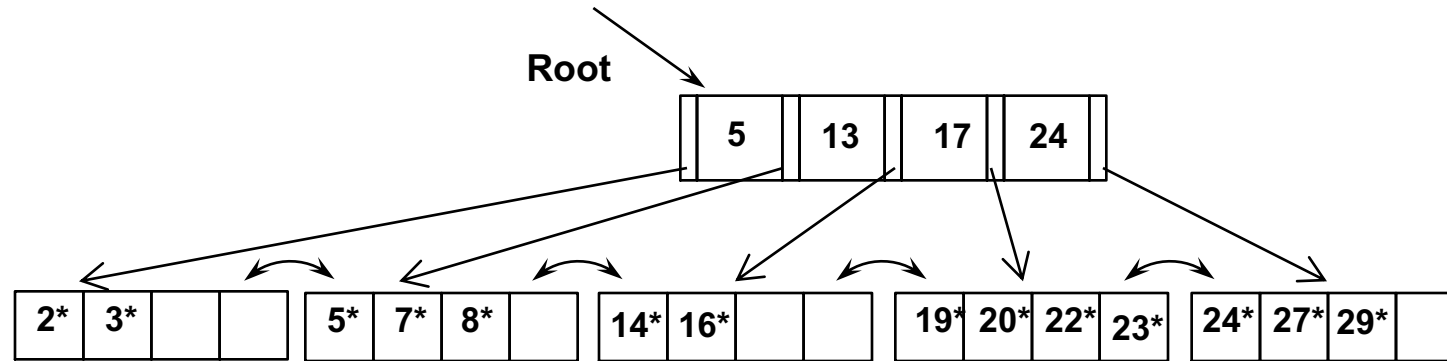


# Example B+ Tree - Inserting 21\*

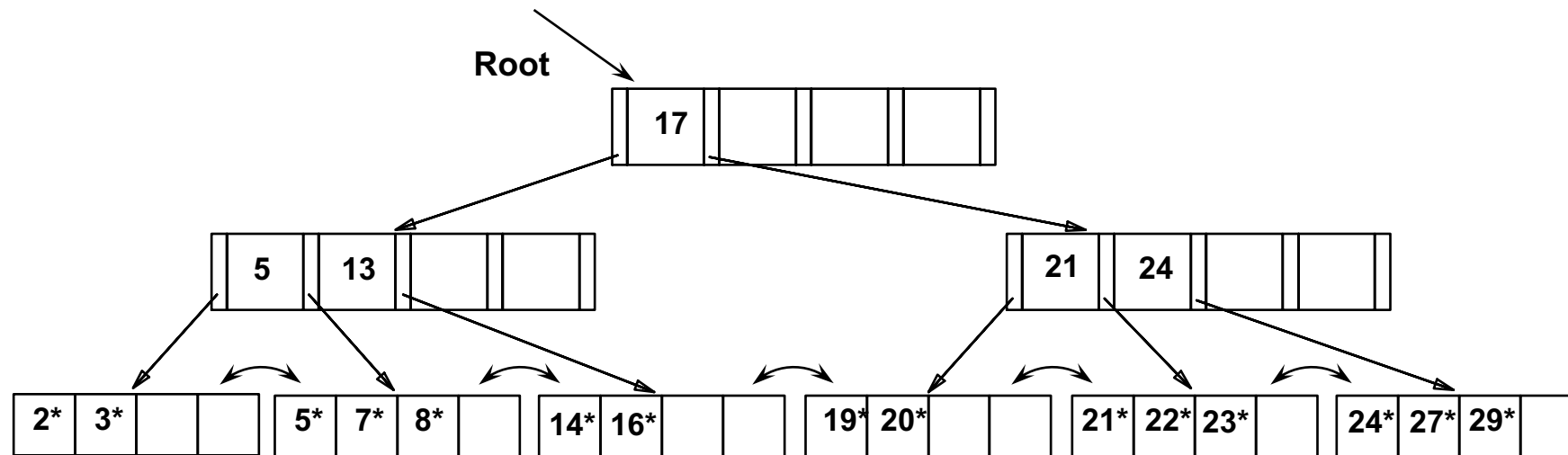
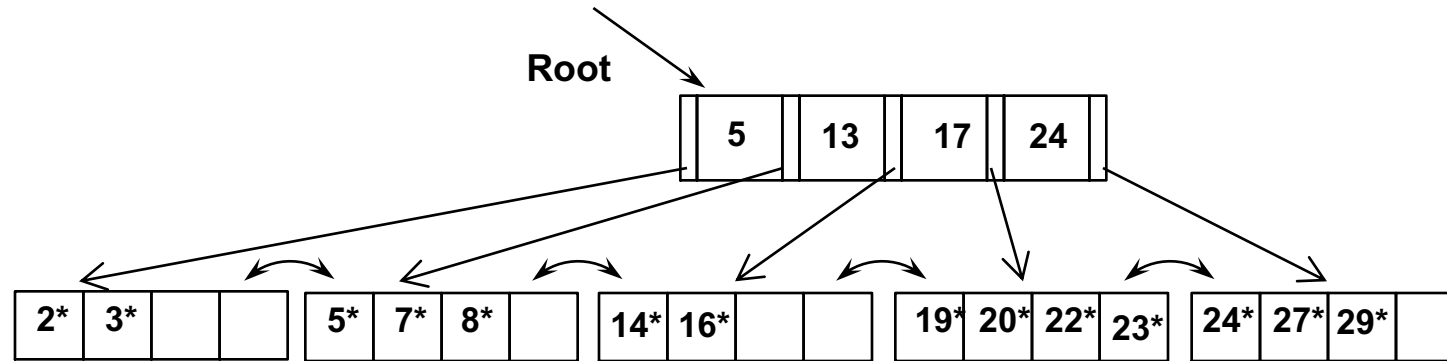




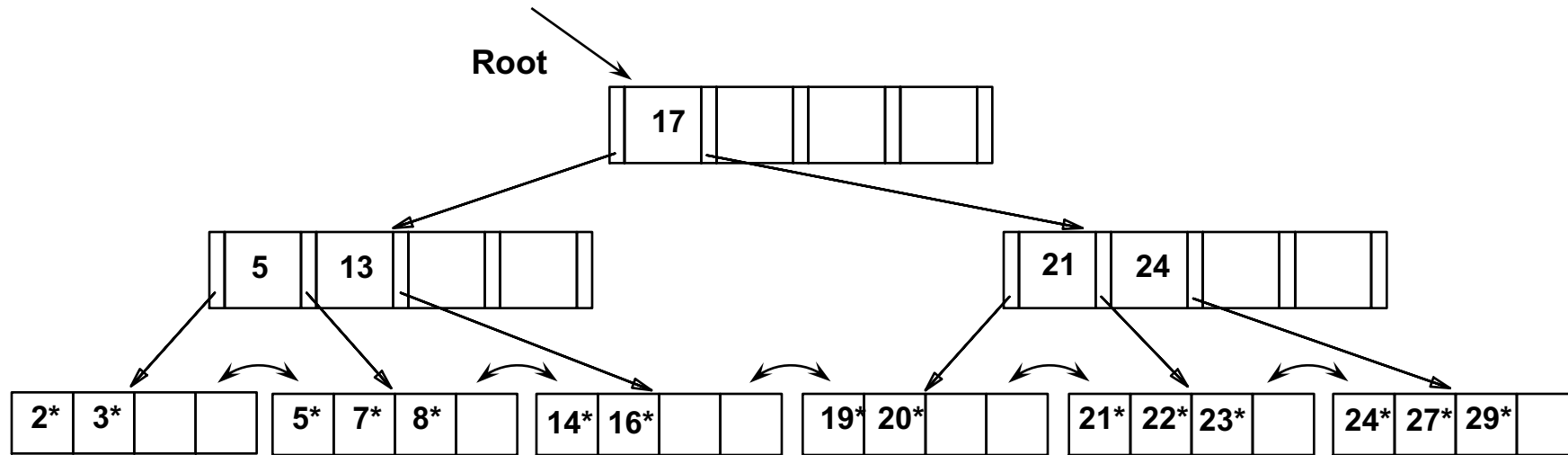
# Example B+ Tree - Inserting 21\*



# Example B+ Tree - Inserting 21\*



# Example B+ Tree



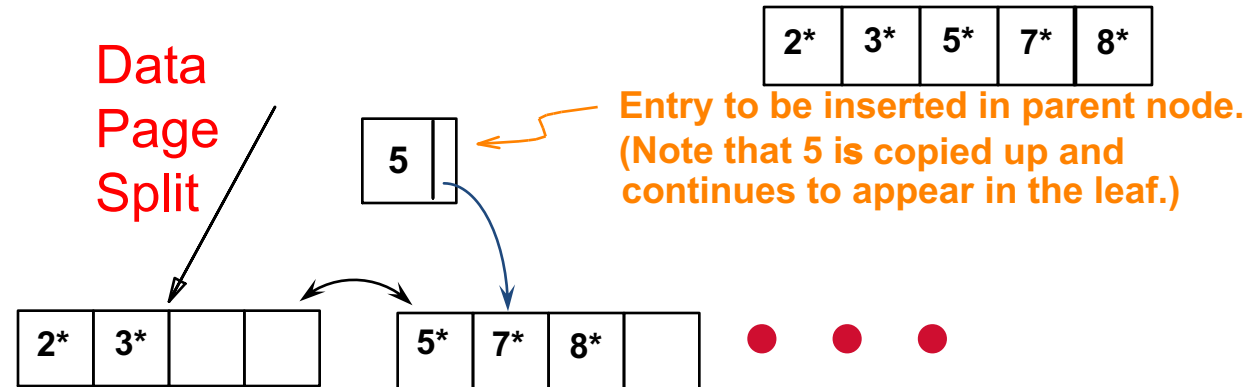
Notice that root was split, leading to increase in height.

In this example, we can avoid split by re-distributing entries;  
however, this is usually not done in practice.

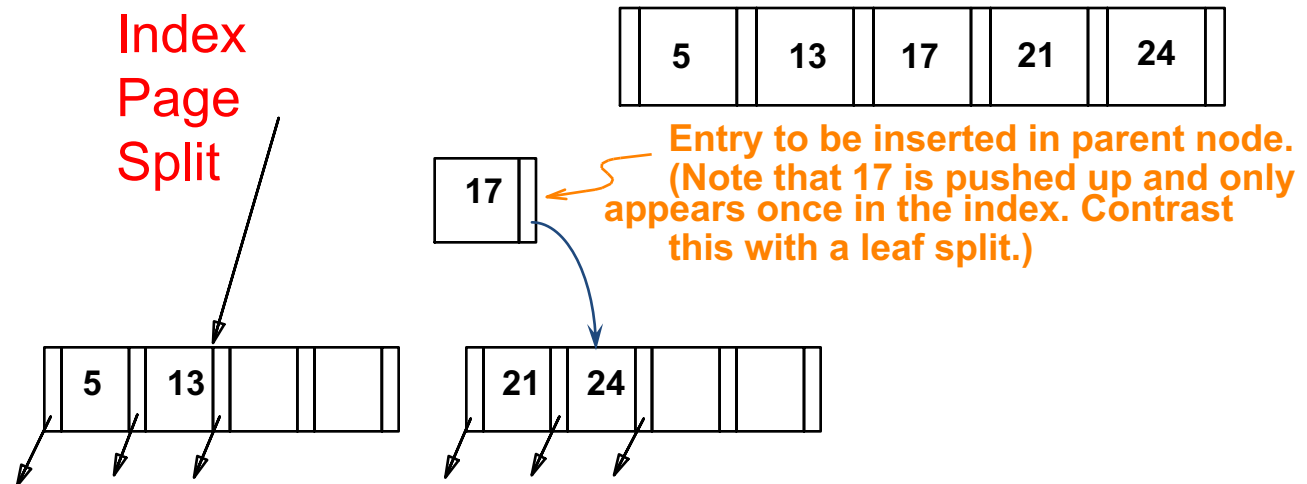
# Example: Data vs. Index Page Split

minimum occupancy is guaranteed in both leaf and index page splits

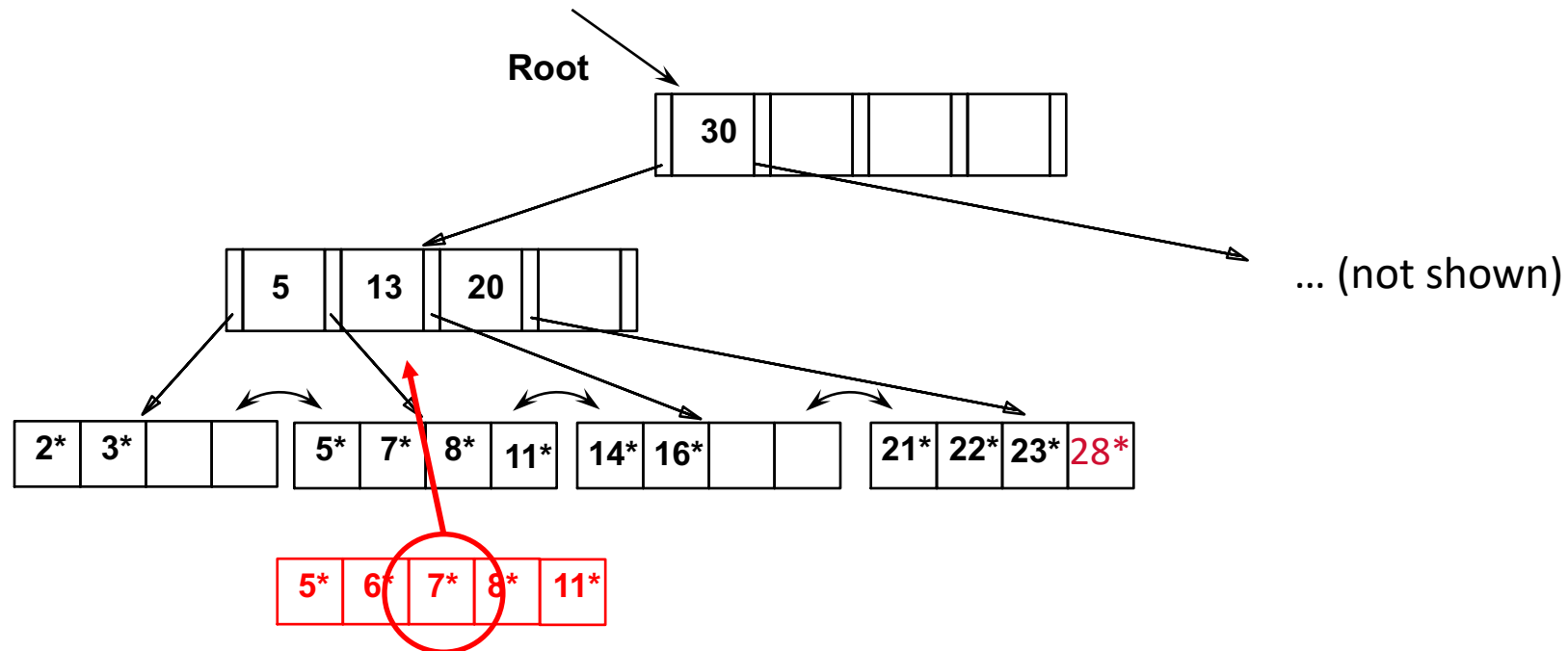
*copy-up* for data page splits



*push-up* for index page split



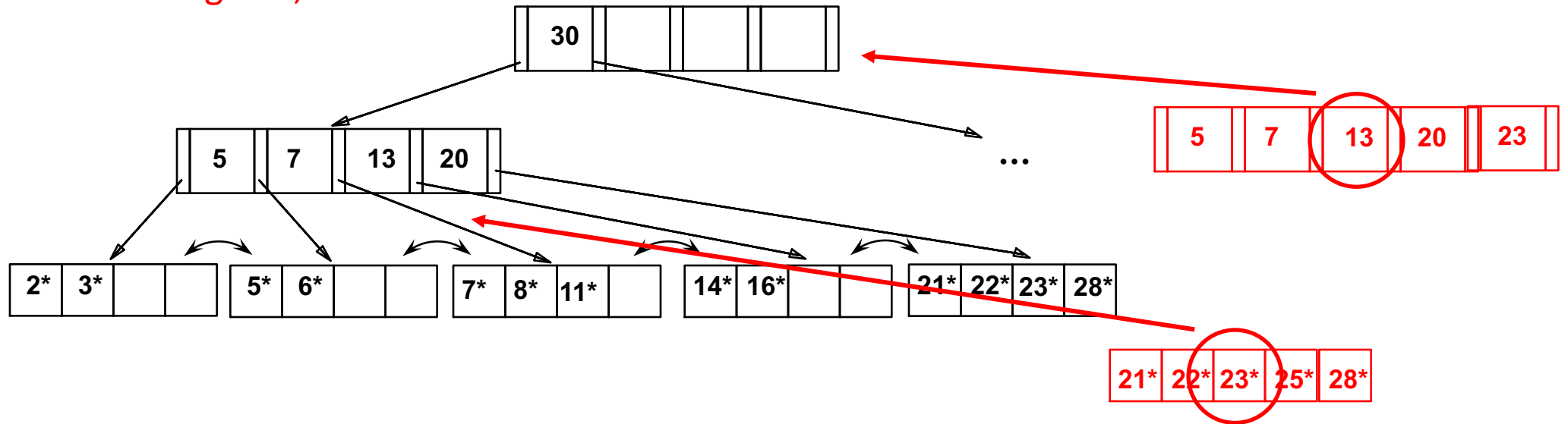
# Now you try...



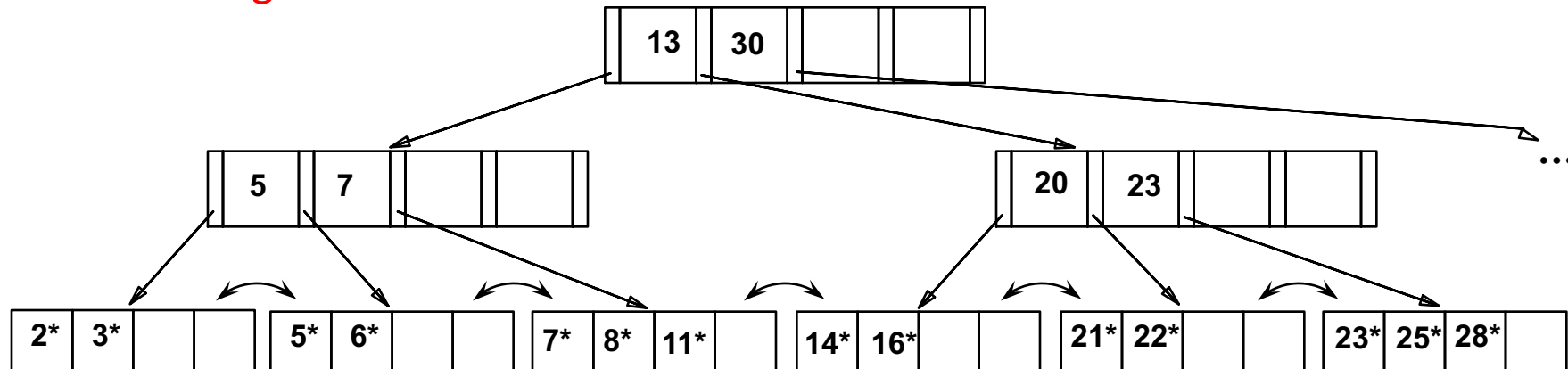
Insert the following data entries (in order): 28\*, 6\*, 25\*

# Answer...

After inserting 28\*, 6\*



After inserting 25\*



# Tree-structured indexing

Intro & B<sup>+</sup>-Tree

Insert into a B<sup>+</sup>-Tree

Delete from a B<sup>+</sup>-Tree

Prefix Key Compression & Bulk Loading

# 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.

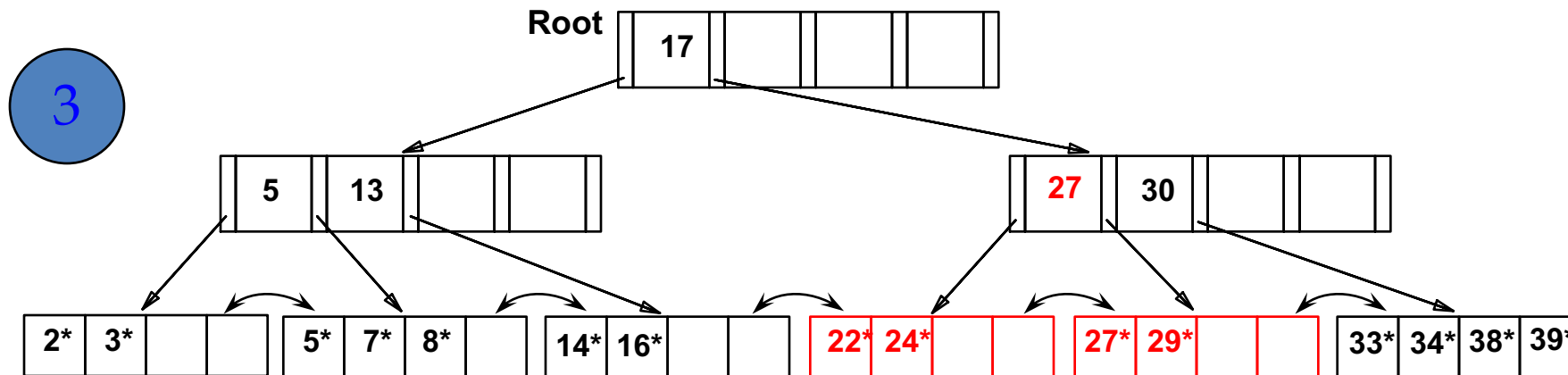
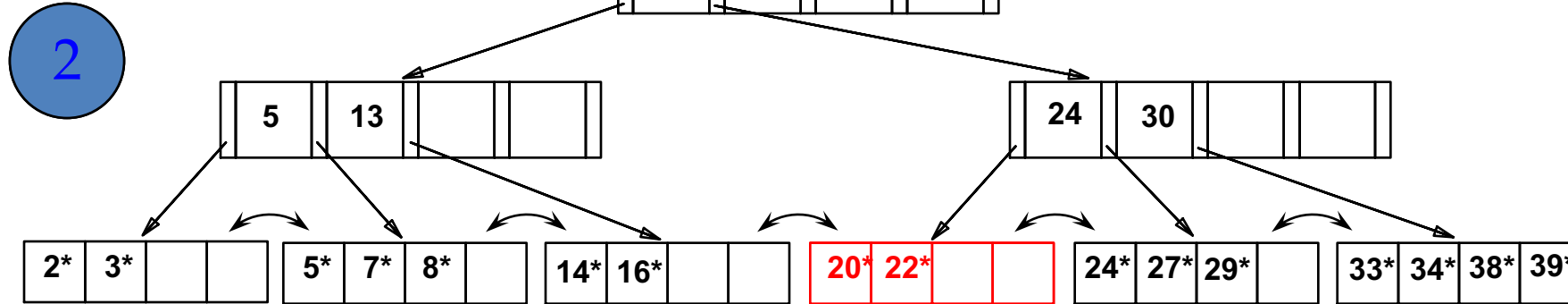
If merge occurred, must delete entry (pointing to  $L$  or sibling) from parent of  $L$ .

Merge could propagate to root, decreasing height.



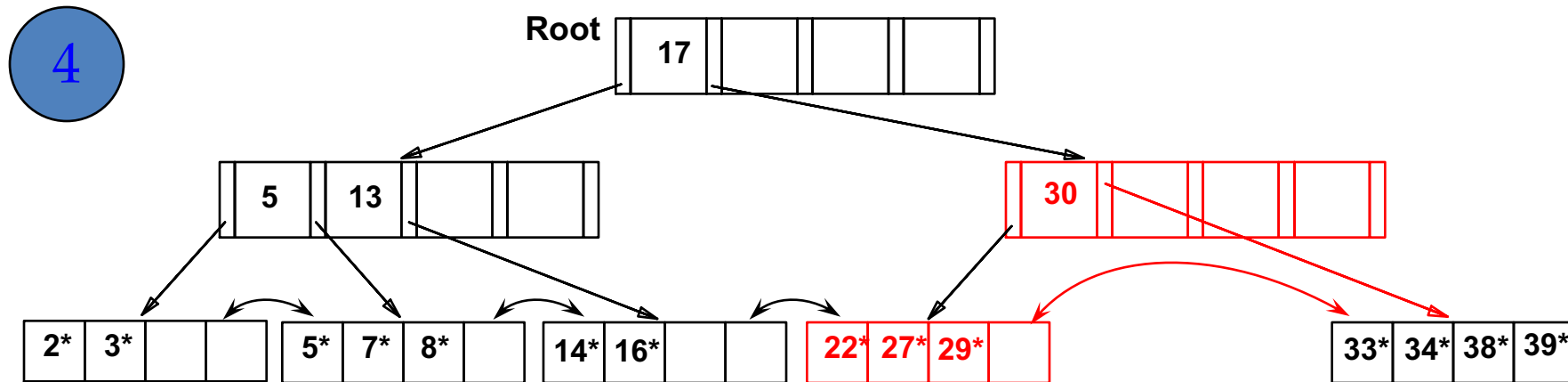
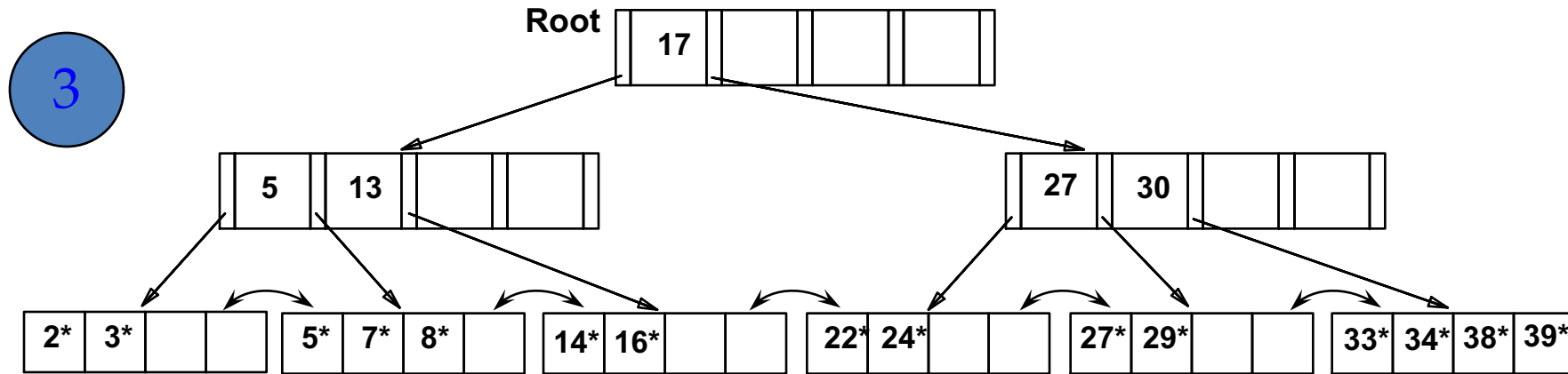
# Example: Delete 19\* & 20\*

Deleting 19\* is easy:



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

# ... and then deleting 24\*

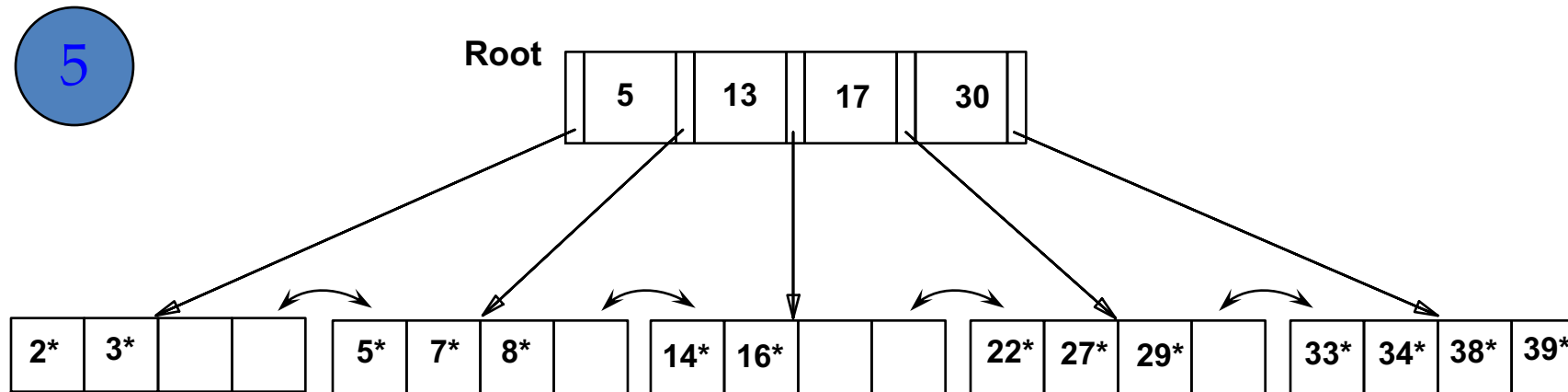
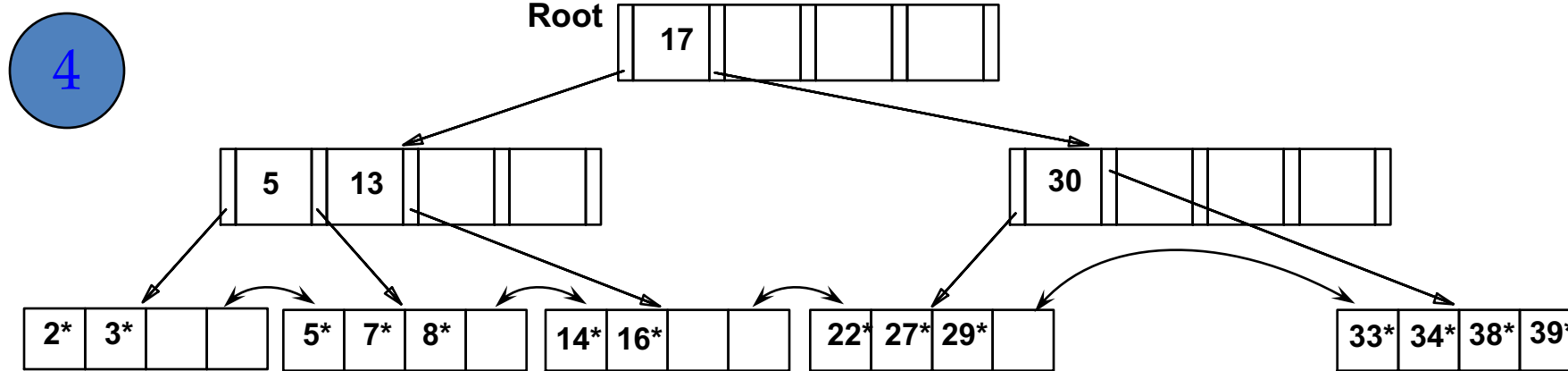


Must **merge** leaves

... but are we done??



# ... merge non-leaf nodes, shrink tree

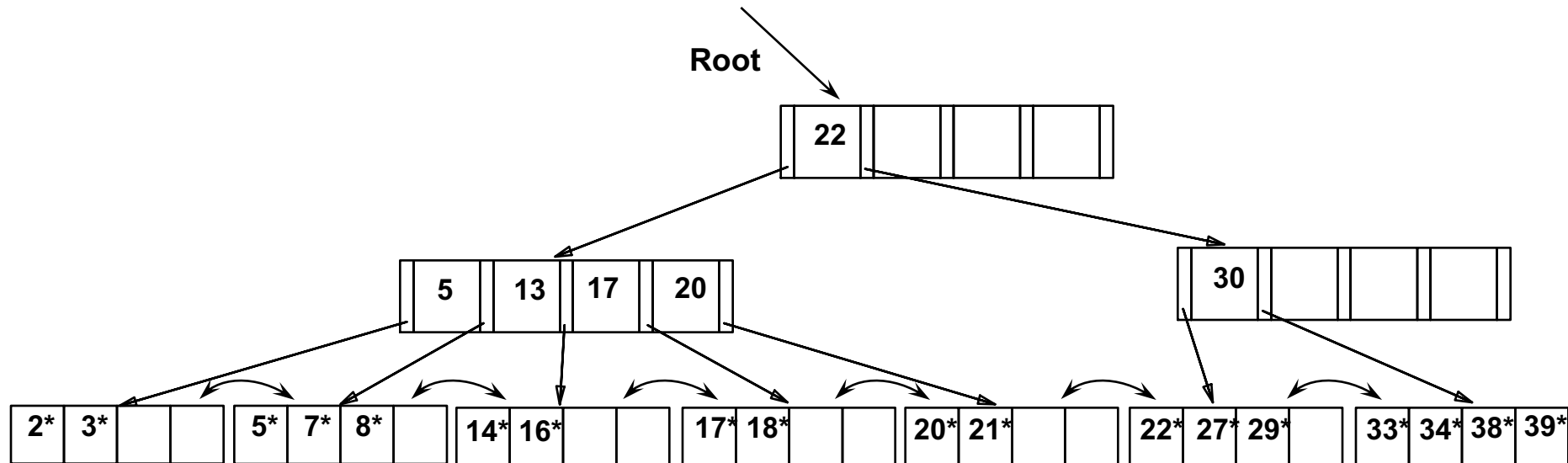


# Example of non-leaf re-distribution

Tree is shown below *during deletion* of 24\*.

*What could be a possible initial tree?* 

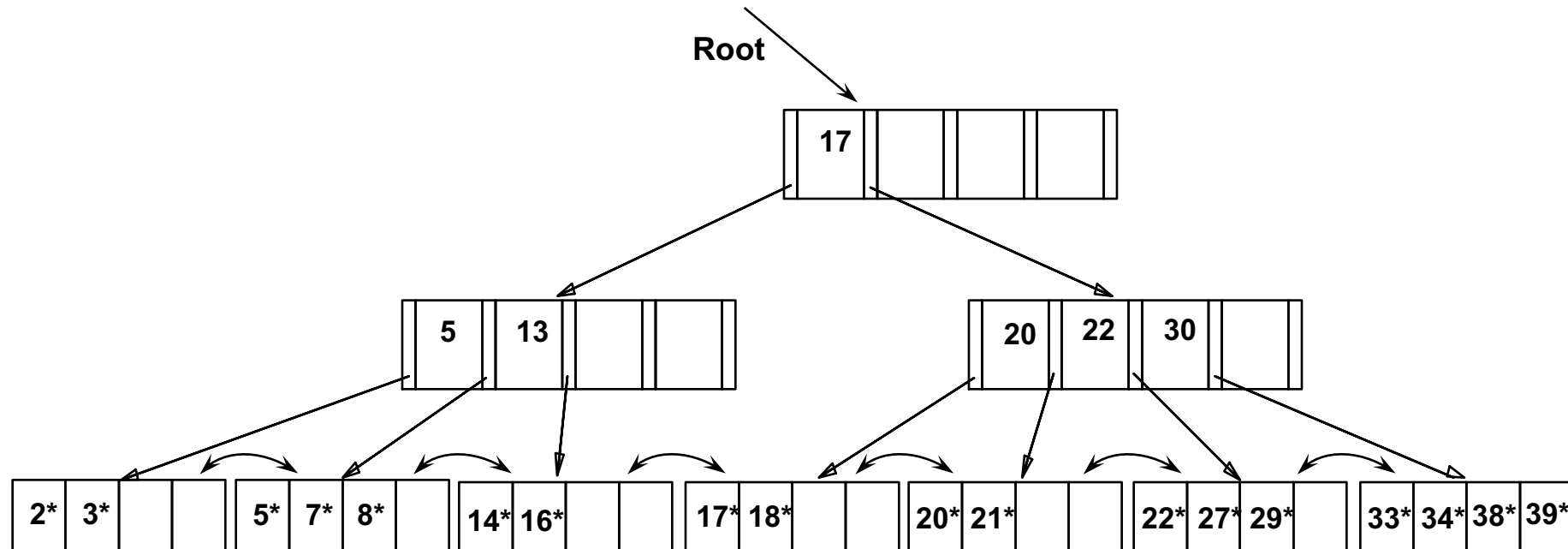
In contrast to previous example, can re-distribute entry from left child of root to right child.



# After Re-distribution

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

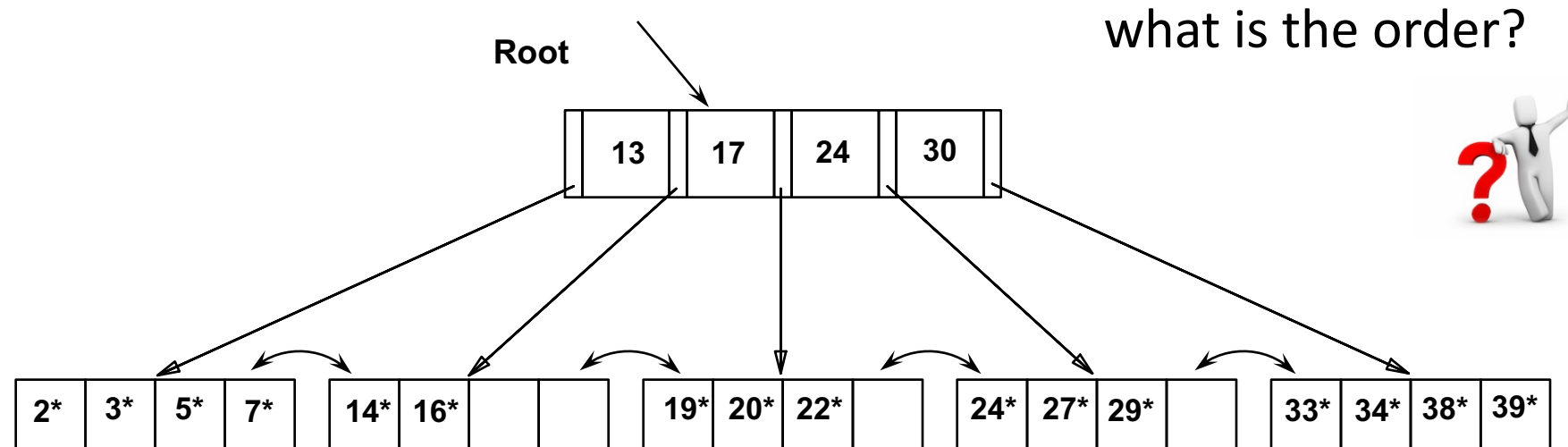
it suffices to re-distribute index entry with key 20;  
we have re-distributed 17 as well for illustration



# Reminders

begin at root, compare keys to reach the leaf

“order”  $d$  means  $d$  to  $2^d$  elements



# Tree-structured indexing

Intro & B<sup>+</sup>-Tree

Insert into a B<sup>+</sup>-Tree

Delete from a B<sup>+</sup>-Tree

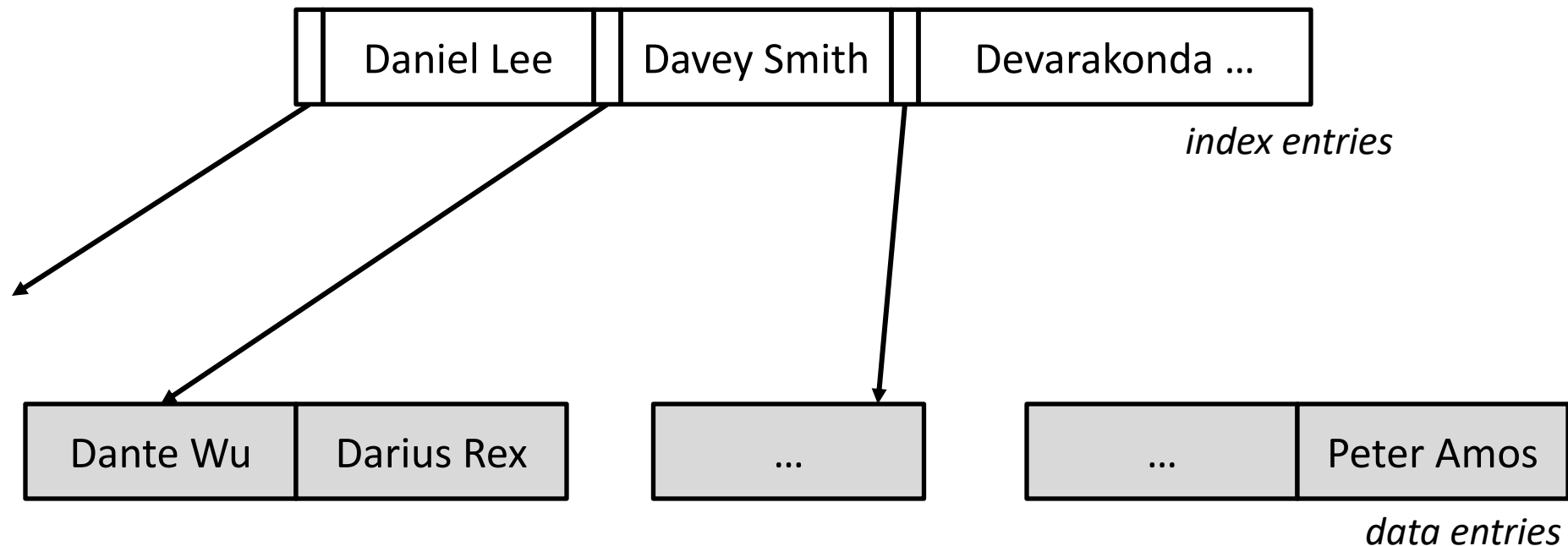
Prefix Key Compression & Bulk Loading

# Prefix Key Compression

we want to increase fan-out



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



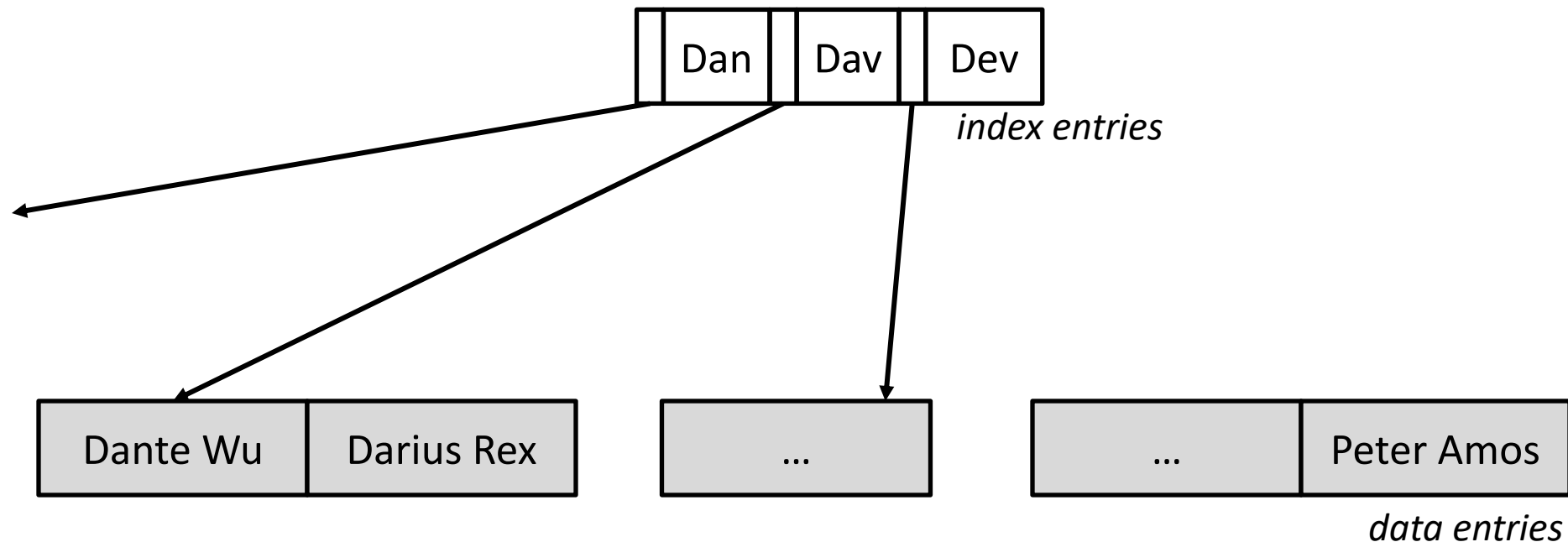


# Prefix Key Compression

we want to increase fan-out



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

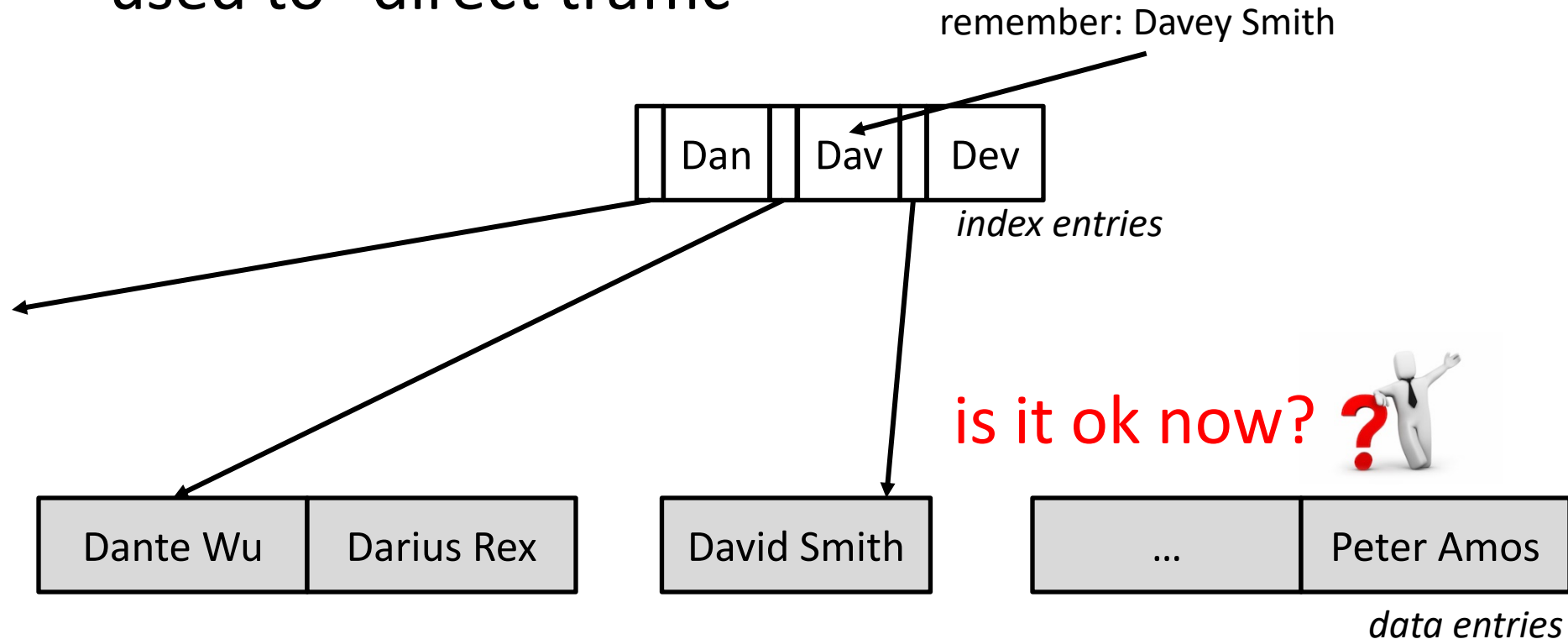


# Prefix Key Compression

we want to increase fan-out

why? 

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



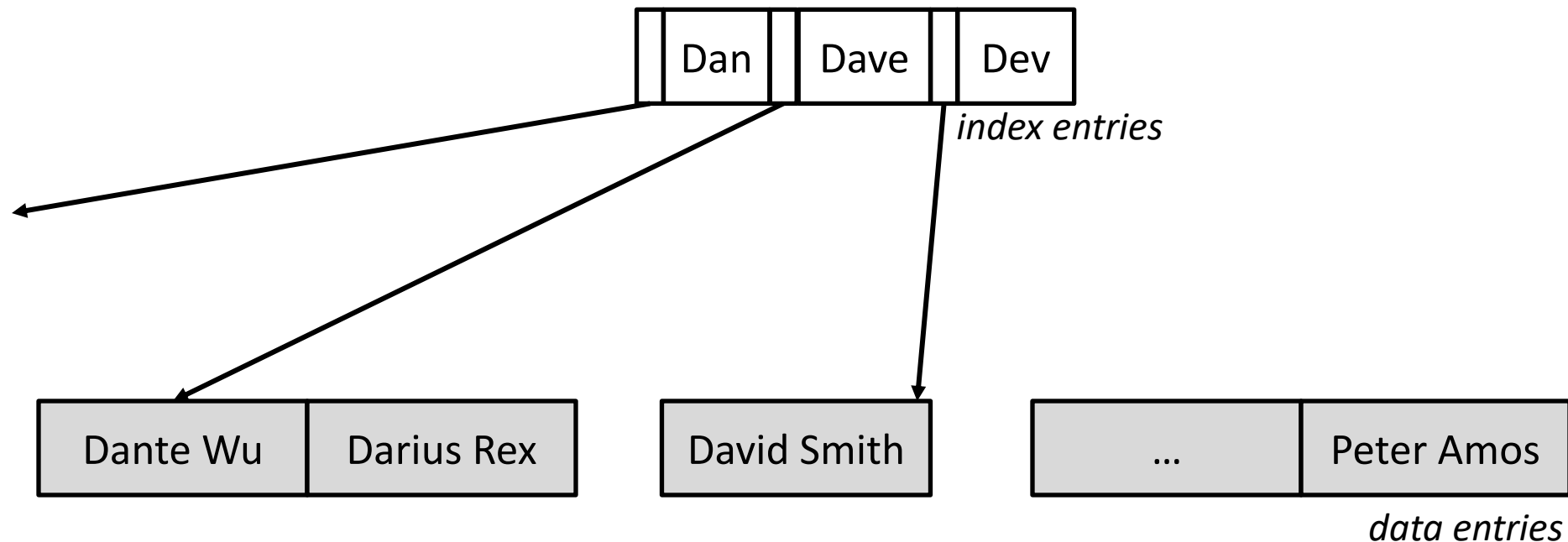
is it ok now? 

# Prefix Key Compression

we want to increase fan-out



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



# Prefix Key Compression

we want to increase fan-out

keys in index entries (internal nodes) are used to “direct traffic”

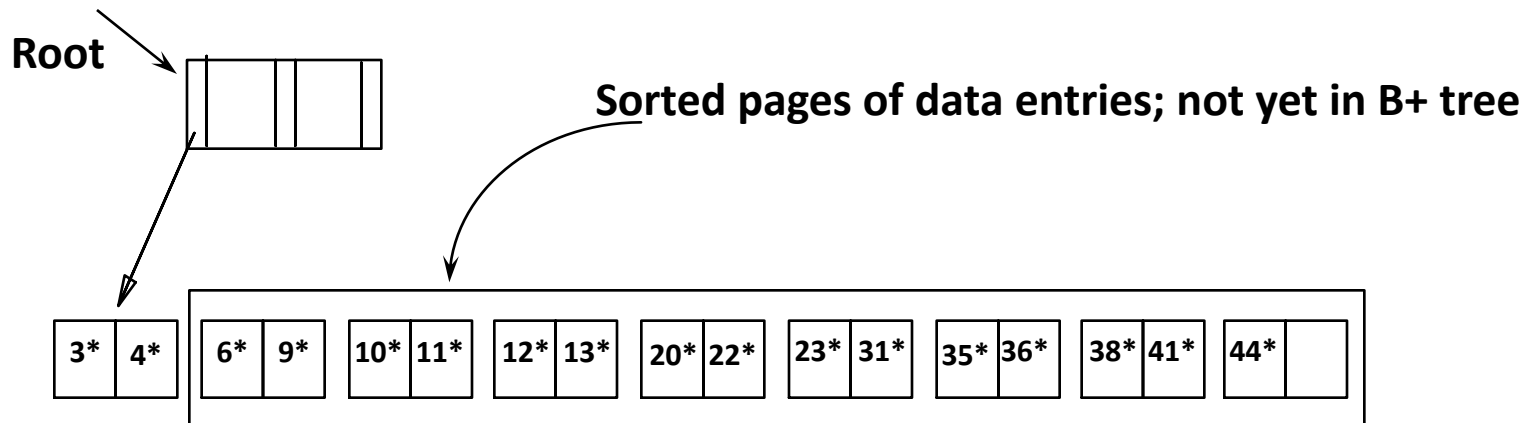
insert/delete must be suitably modified

# 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 repeatedly inserting records is very slow.

Bulk Loading can be done much more efficiently.

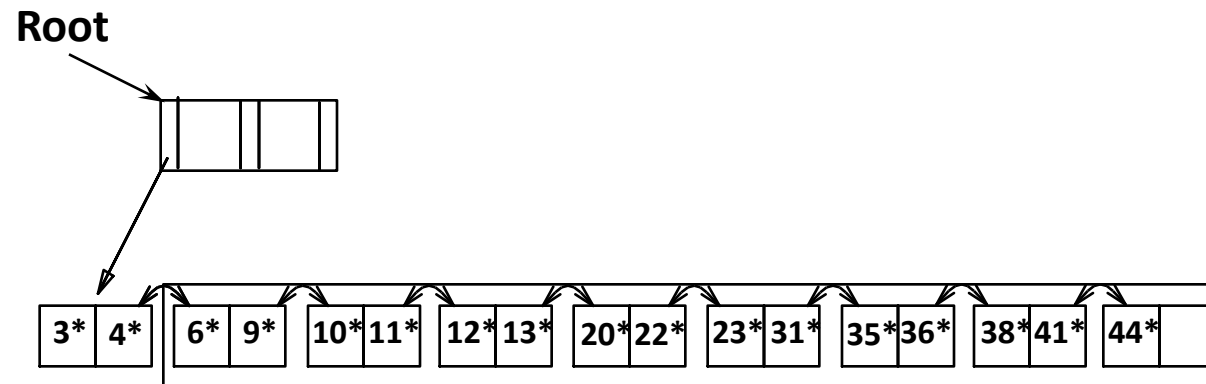
*Initialization:* Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



# Bulk Loading (Contd.)

**where to insert:** into right-most index page just above leaf level

**what to insert:** the left-most value of the new leaf

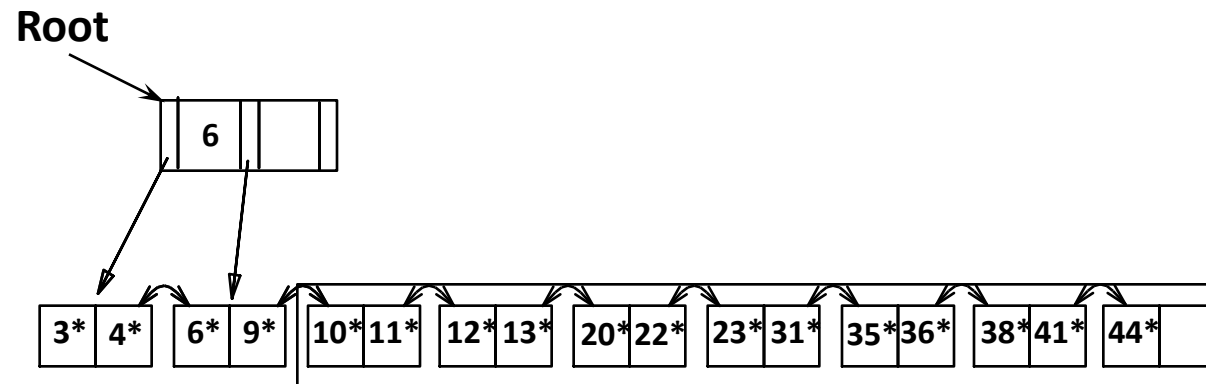


**what to do when full?** when this fills up, splits node  
(if needed split may go up right-most path to the root)

# 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

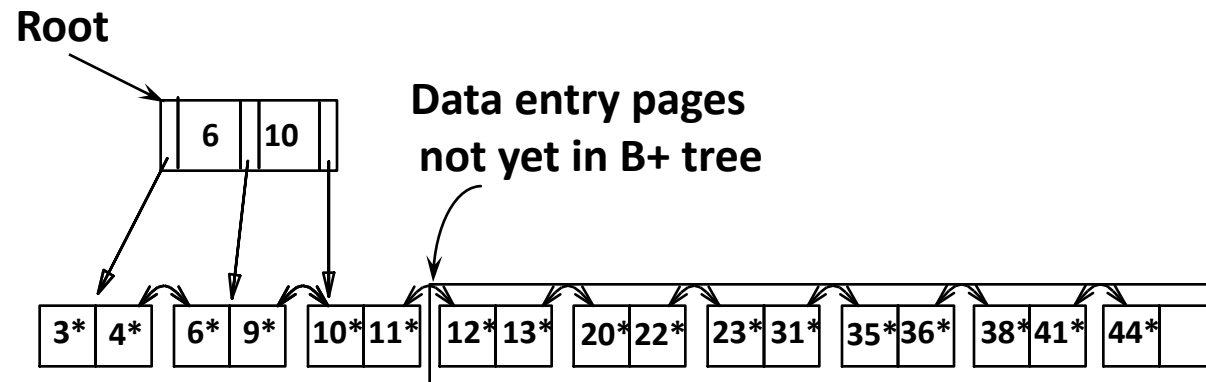


**what to do when full?** when this fills up, splits node  
(if needed split may go up right-most path to the root)

# 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



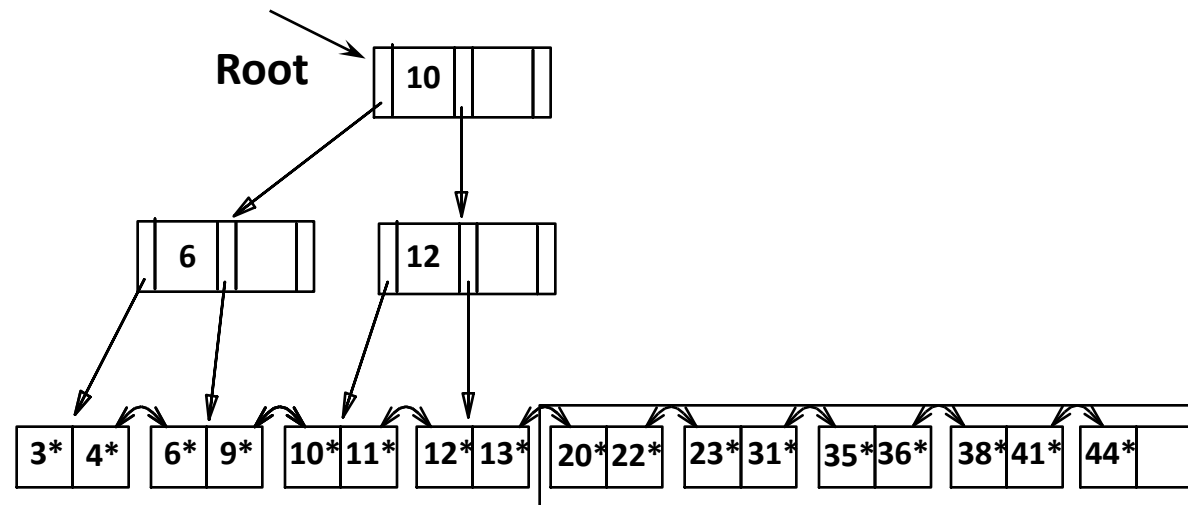
**what to do when full?** when this fills up, splits node  
(if needed split may go up right-most path to the root)



# 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

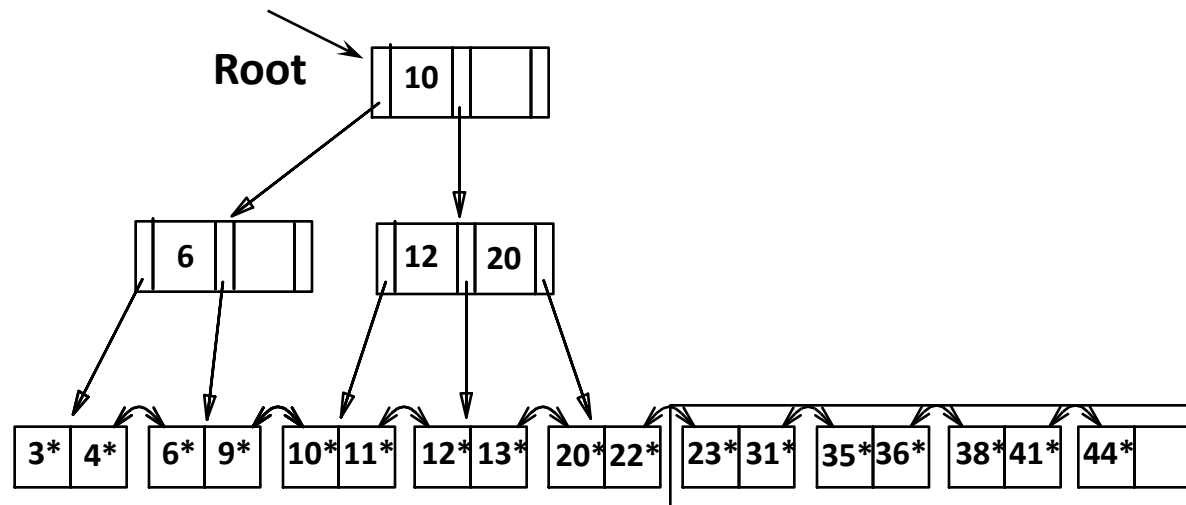


**what to do when full?** when this fills up, splits node  
(if needed split may go up right-most path to the root)

# 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

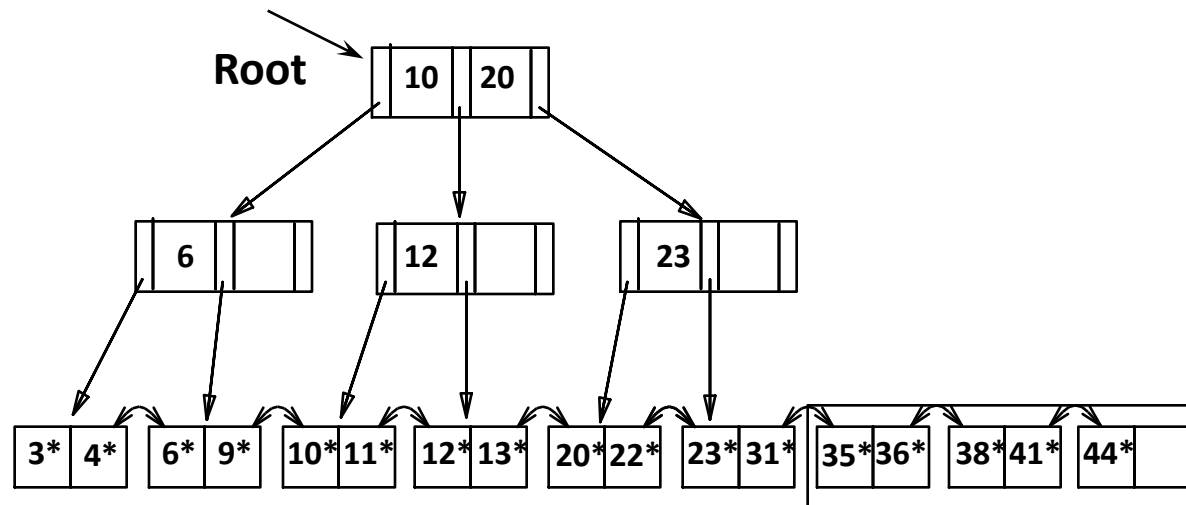


**what to do when full?** when this fills up, splits node  
(if needed split may go up right-most path to the root)

# 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

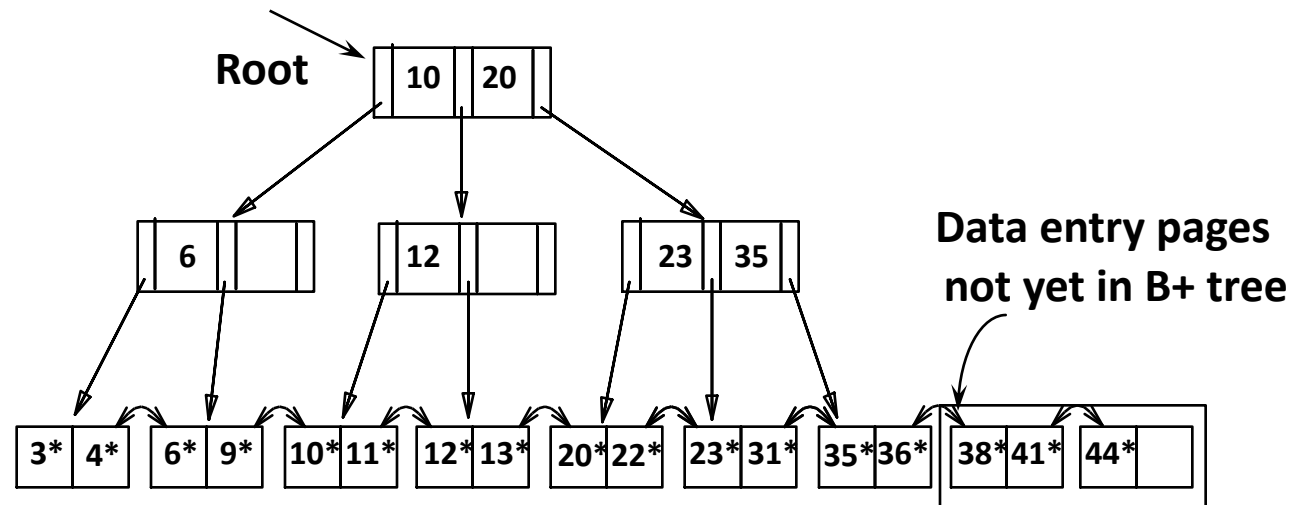


**what to do when full?** when this fills up, splits node  
(if needed split may go up right-most path to the root)

# Bulk Loading (Contd.)

**where to insert:** into right-most index page just above leaf level

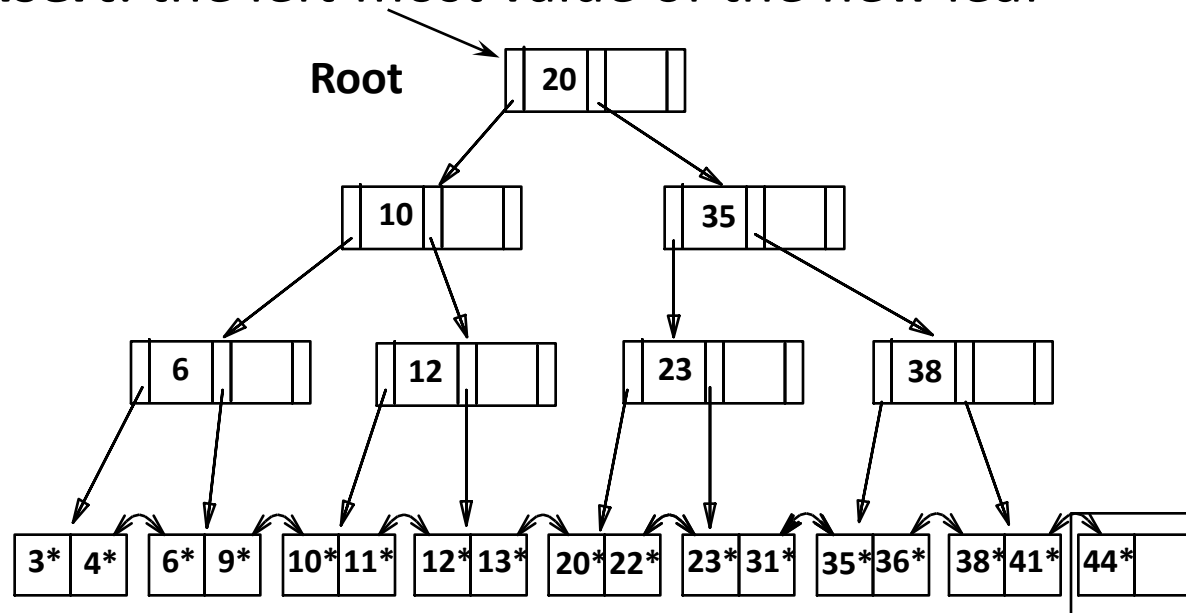
**what to insert:** the left-most value of the new leaf



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

# 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

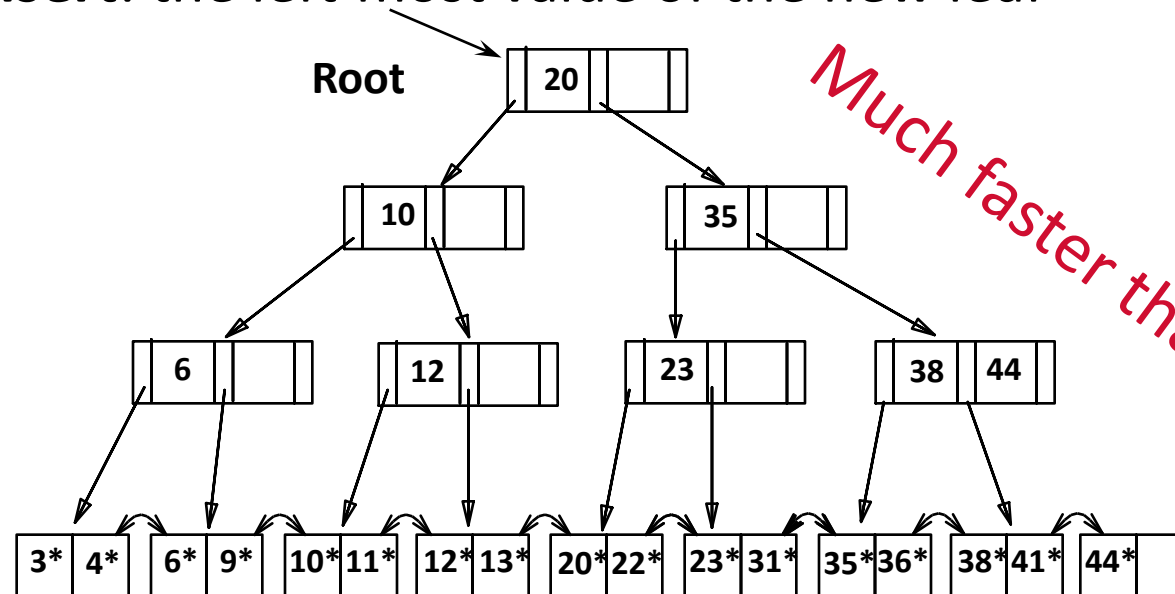


**what to do when full?** when this fills up, splits node  
 (if needed split may go up right-most path to the root)

# 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)

# 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**