I/O story

The real sorting story: What if list can't fit in memory?

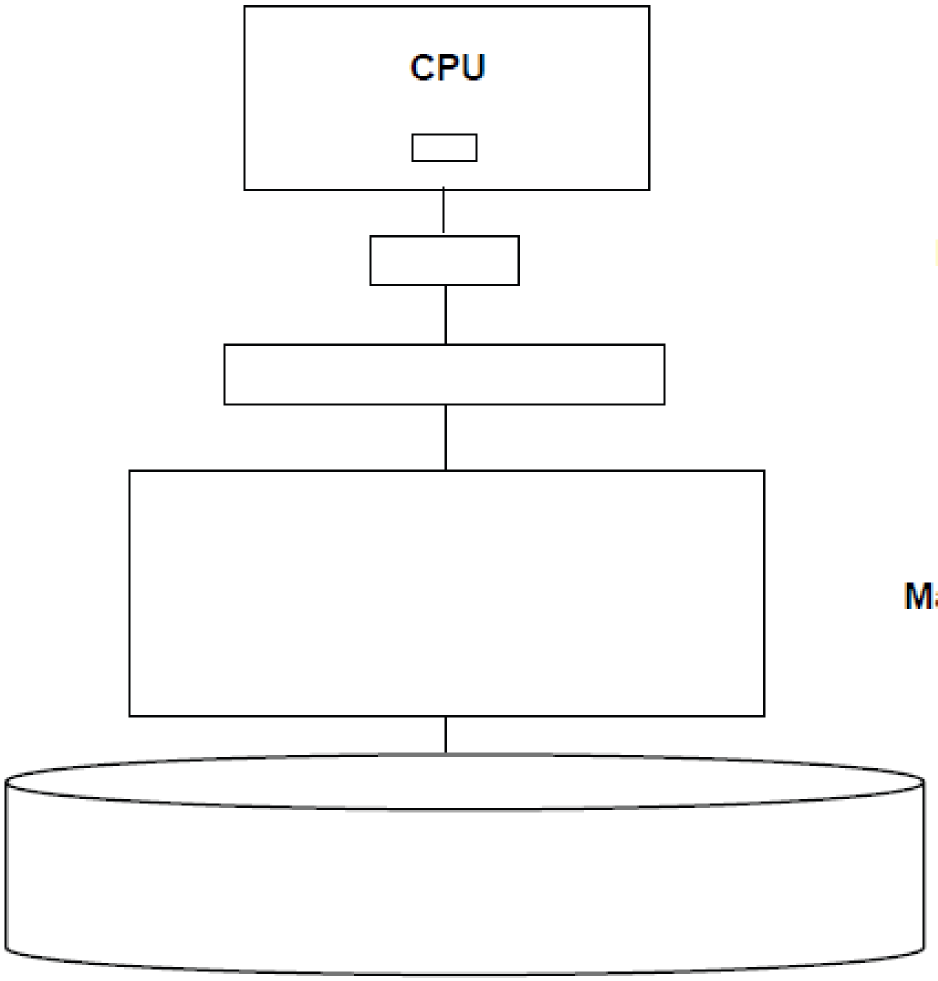
Let

- File size, N = # records to sort;
- Memory size, M = # records that can fit into internal memory;
- Block size, B = # records that can be transferred in a single block;
- P = # blocks that can be transferred concurrently

Merge sort:

$$\frac{2N}{B}\left(1 + \left\lceil \log_{M/B^{-1}} \frac{N}{M} \right\rceil\right) = O\left(\frac{N \log(N/B)}{B \log(M/B)}\right)$$

Memory Access (rough)



Cycles to access:

Registers	1
L1 Cache	2
L2 Cache	30
Main memory	250
Disk	Random: 30,000,000
	Streamed: 5000

B-Trees

Courtsey: Prof. **Krishnaprasad Thirunarayan, Wright State Univ.**
Src: cecs.wright.edu/~tkprasad/courses/cs707/L04-X-**B-Trees**.ppt

Motivation for B-Trees

- Index structures for large datasets cannot be stored in main memory
- Storing it on disk requires different approach to efficiency

- Assuming that a disk spins at 3600 RPM, one revolution occurs in 1/60 of a second, or 16.7ms
- Crudely speaking, one disk access takes about the same time as 200,000 instructions

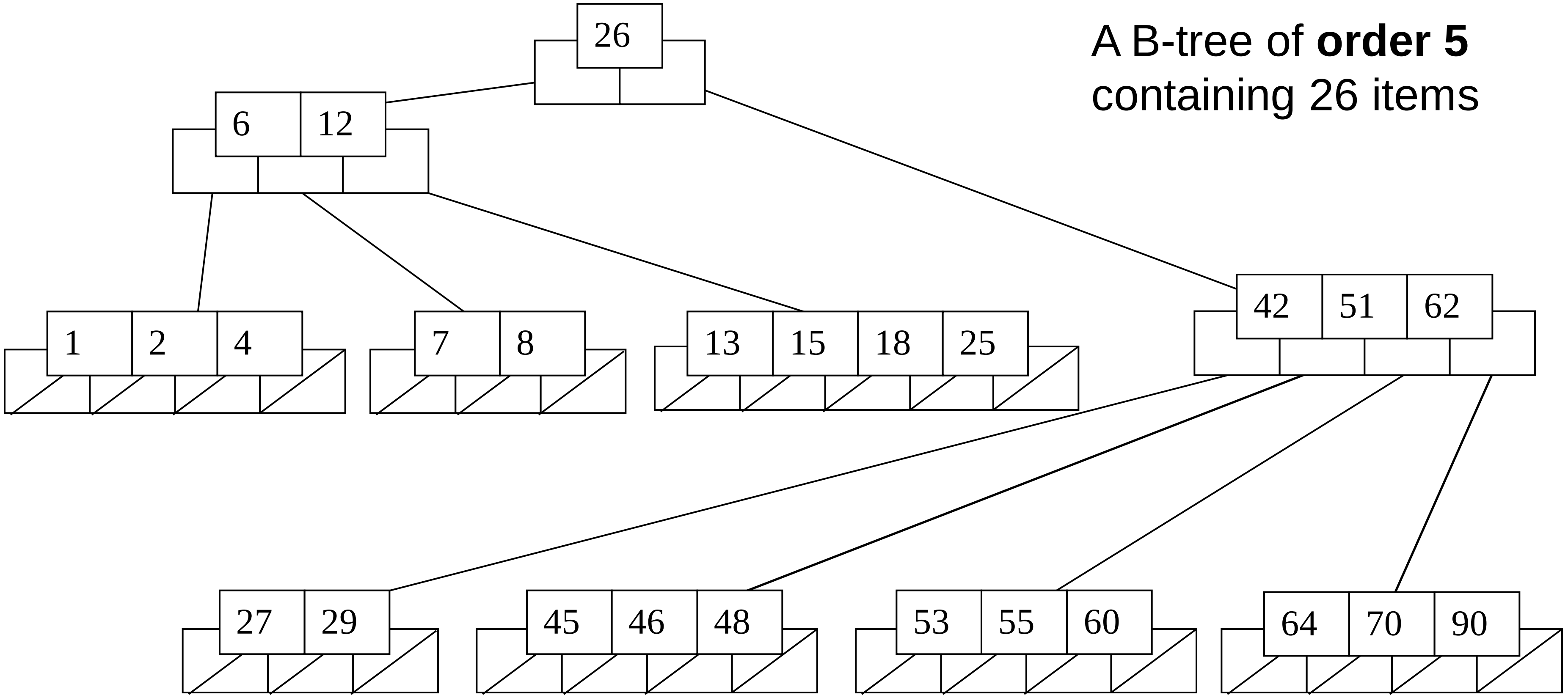
Motivation (cont.)

- Assume that we use an AVL tree to store about 20 million records
- We end up with a **very** deep binary tree with lots of different disk accesses; $\log_2 20{,}000{,}000$ is about 24, so this takes about 0.2 seconds
- We know we can't improve on the log *n* lower bound on search for a binary tree
- But, the solution is to use more branches and thus reduce the height of the tree!
  - As branching increases, depth decreases

Definition of a B-tree

- A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:
  1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
  2. all leaves are on the same level
  3. all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
  4. the root is either a leaf node, or it has from two to *m* children
  5. a leaf node contains no more than $m – 1$ keys
- The number *m* should always be odd

An example B-Tree

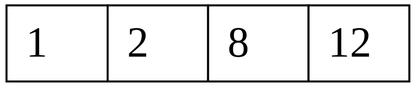


A B-tree of **order 5** containing 26 items

Note that all the leaves are at the same level

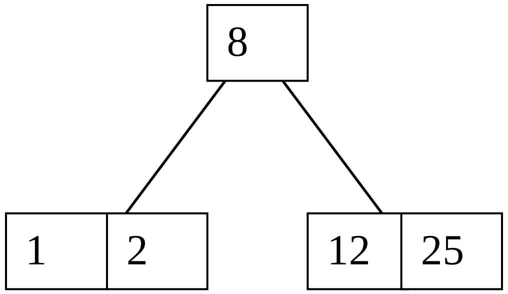
Constructing a B-tree

- Suppose we start with an empty B-tree and keys arrive in the following order:1 12 8 2 25 5 14 28 17 7 52 16 48 68 3 26 29 53 55 45
- We want to construct a B-tree of order 5
- The first four items go into the root:

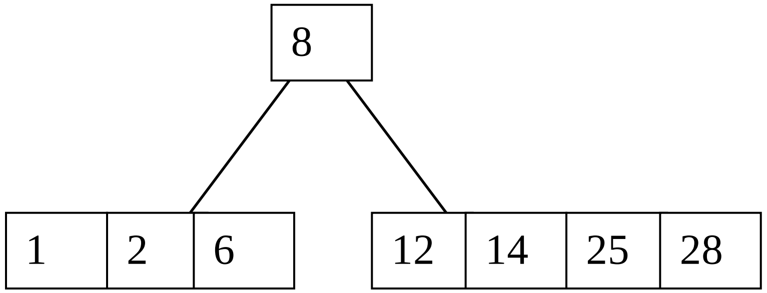


- To put the fifth item in the root would violate condition 5
- Therefore, when 25 arrives, pick the middle key to make a new root

Constructing a B-tree (contd.)

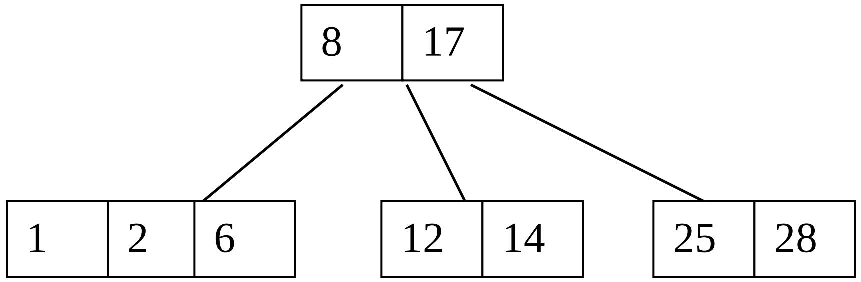


6, 14, 28 get added to the leaf nodes:

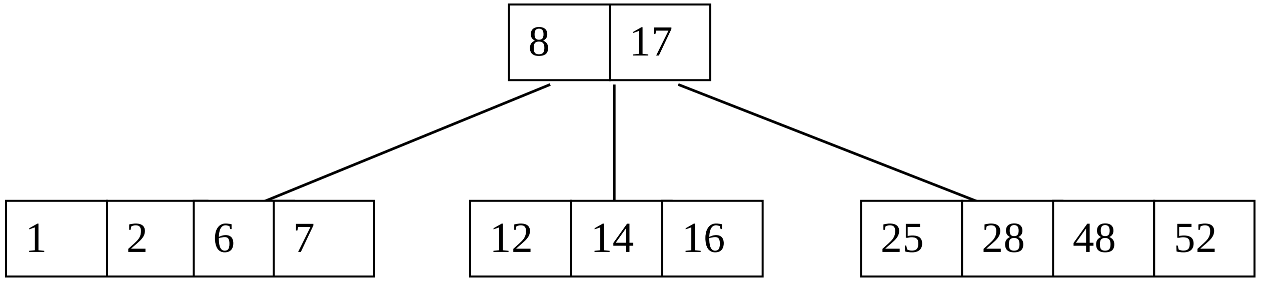


Constructing a B-tree (contd.)

Adding 17 to the right leaf node would over-fill it, so we take the middle key, promote it (to the root) and split the leaf

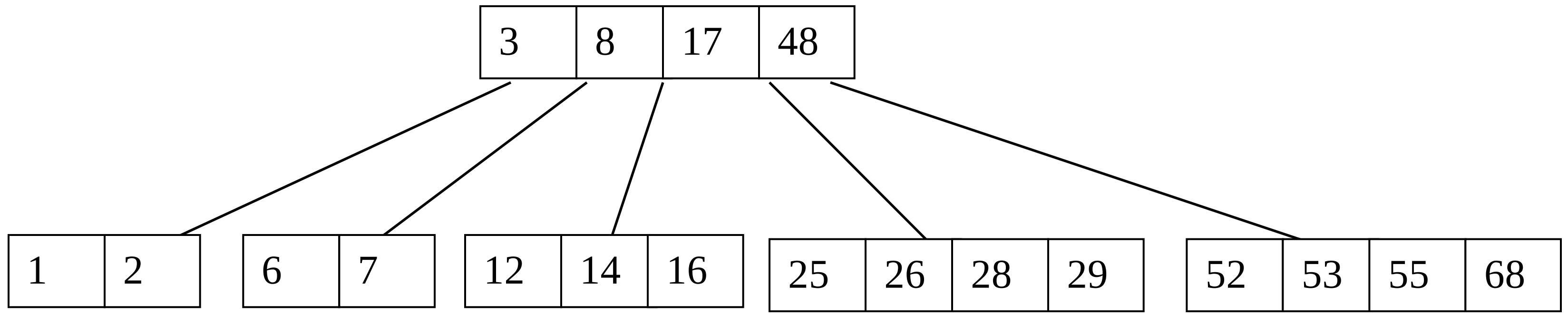


7, 52, 16, 48 get added to the leaf nodes

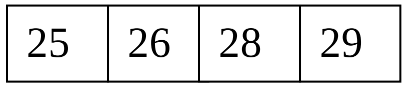


Constructing a B-tree (contd.)

Adding 68 causes us to split the right most leaf, promoting 48 to the root, and adding 3 causes us to split the left most leaf, promoting 3 to the root; 26, 29, 53, 55 then go into the leaves

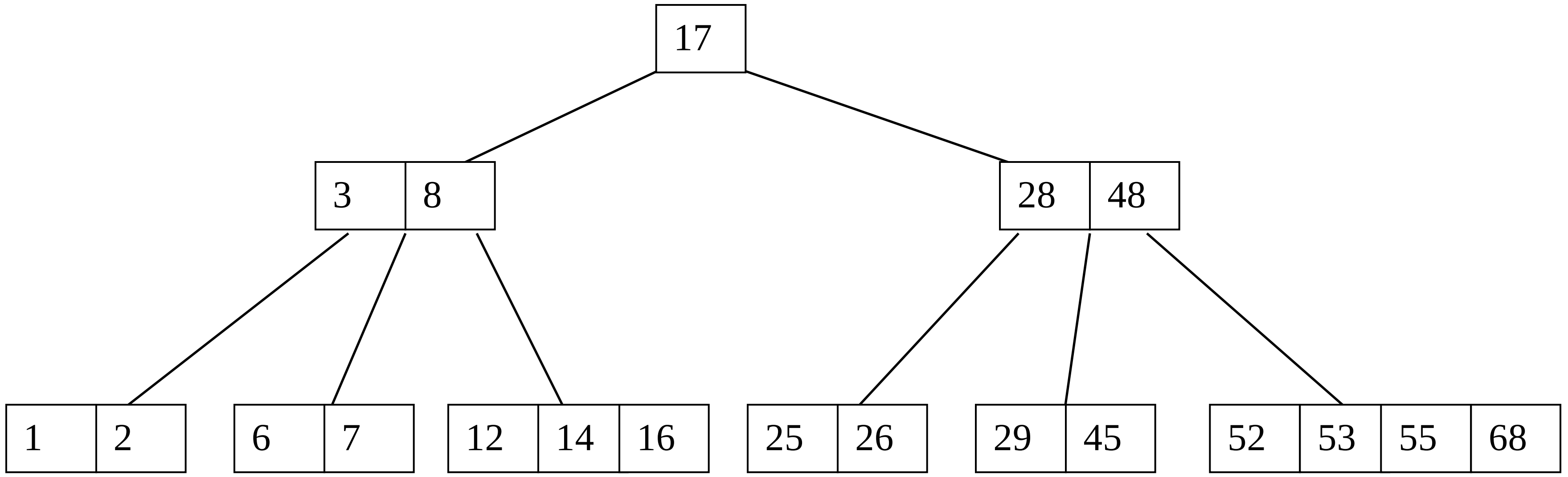


Adding 45 causes a split of



and promoting 28 to the root then causes the root to split

Constructing a B-tree (contd.)



Inserting into a B-Tree

- Attempt to insert the new key into a leaf
- If this would result in that leaf becoming too big, split the leaf into two, promoting the middle key to the leaf's parent
- If this would result in the parent becoming too big, split the parent into two, promoting the middle key
- This strategy might have to be repeated all the way to the top
- If necessary, the root is split in two and the middle key is promoted to a new root, making the tree one level higher

Exercise in Inserting a B-Tree

- Insert the following keys to a 5-way B-tree:
- 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56
-

Removal from a B-tree

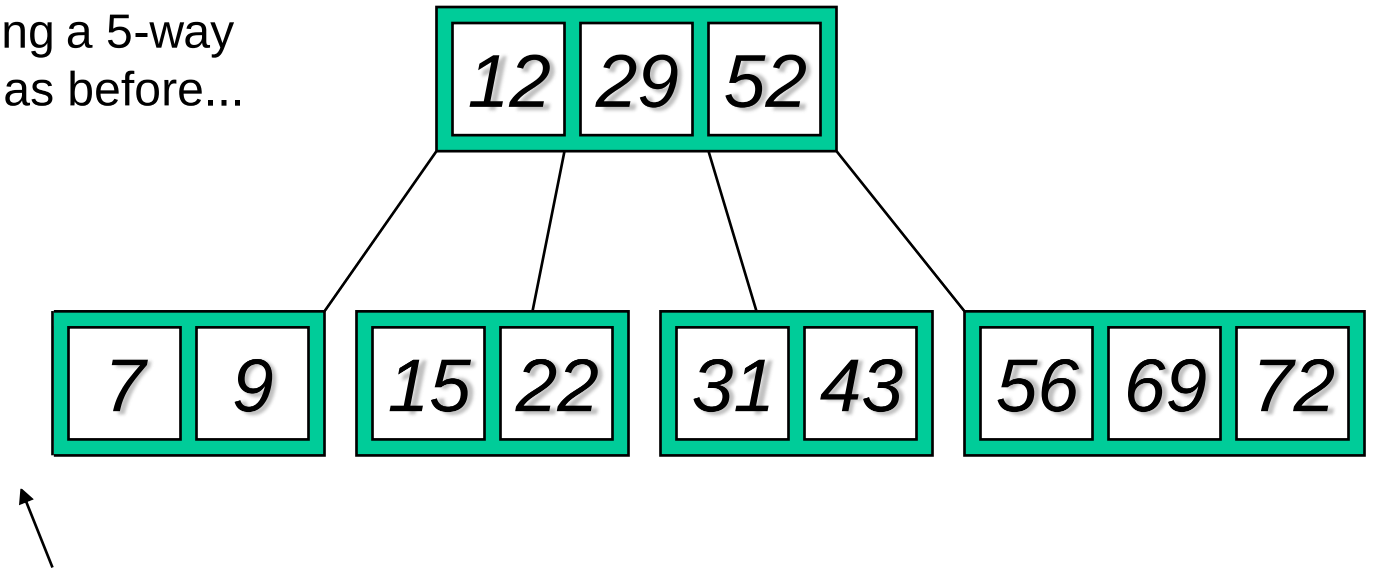
- During insertion, the key always goes *into* a *leaf*. For deletion we wish to remove *from* a leaf. There are three possible ways we can do this:
- 1 - If the key is already in a leaf node, and removing it doesn't cause that leaf node to have too few keys, then simply remove the key to be deleted.
- 2 - If the key is *not* in a leaf then it is guaranteed (by the nature of a B-tree) that its predecessor or successor will be in a leaf -- in this case we can delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.

Removal from a B-tree (2)

- If (1) or (2) lead to a leaf node containing less than the minimum number of keys then we have to look at the siblings immediately adjacent to the leaf in question:
  - 3: if one of them has more than the min. number of keys then we can promote one of its keys to the parent and take the parent key into our lacking leaf
  - 4: if neither of them has more than the min. number of keys then the lacking leaf and one of its neighbours can be combined with their shared parent (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leave the parent with too few keys then we repeat the process up to the root itself, if required

Type #1: Simple leaf deletion

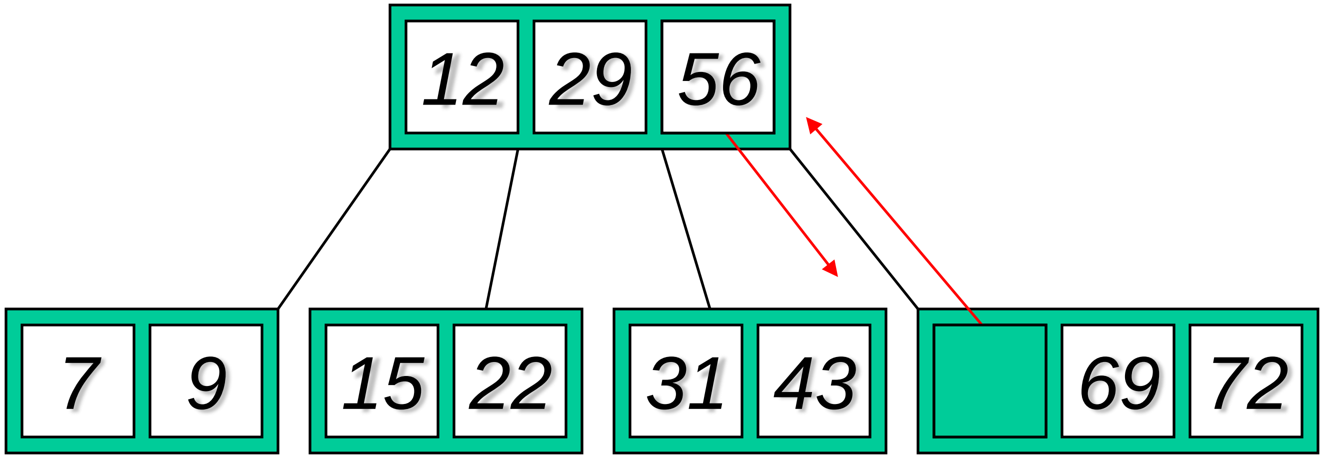
Assuming a 5-way B-Tree, as before...



Delete 2: Since there are enough keys in the node, just delete it

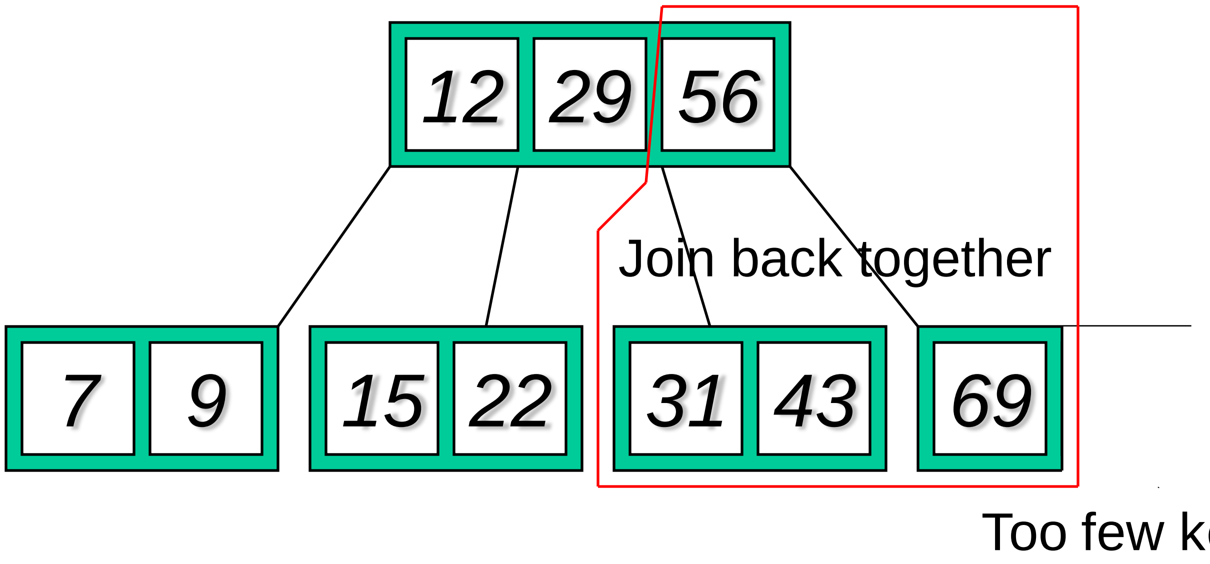
Note when printed: this slide is animated

Type #2: Simple non-leaf deletion



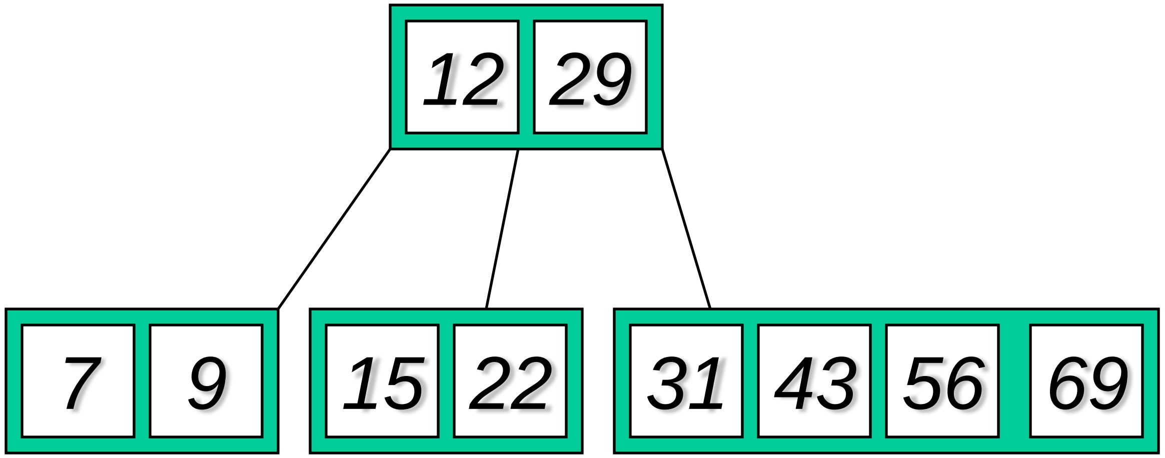
Note when printed: this slide is animated

Type #4: Too few keys in node and its siblings



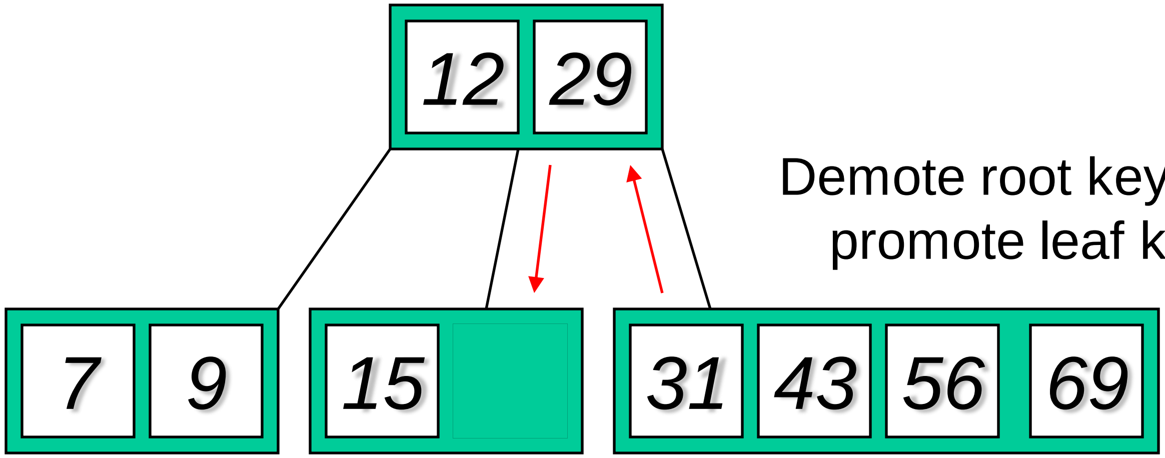
Note when printed: this slide is animated

Type #4: Too few keys in node and its siblings



Note when printed: this slide is animated

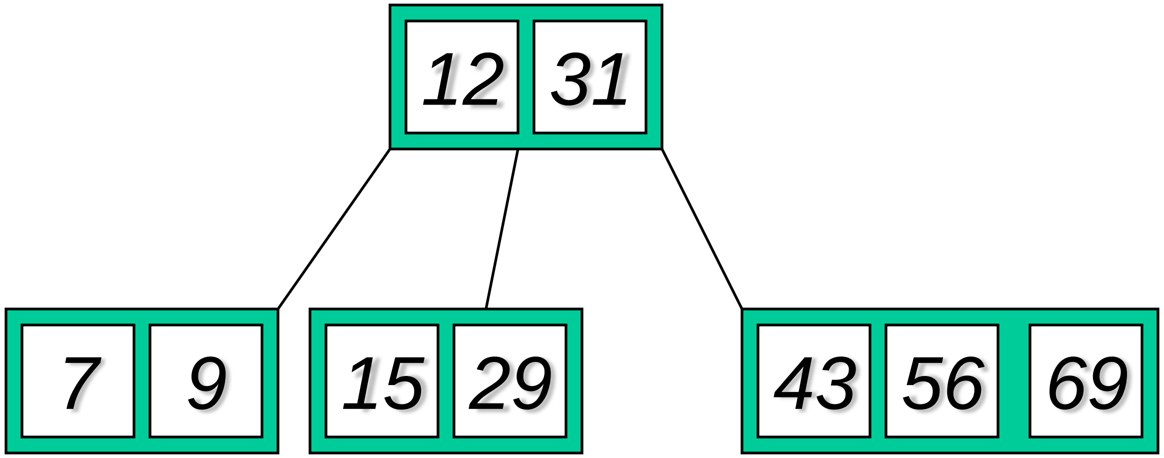
Type #3: Enough siblings



Demote root key and promote leaf key

Note when printed: this slide is animated

Type #3: Enough siblings



Note when printed: this slide is animated

Exercise in Removal from a B-Tree

- Given 5-way B-tree created by these data (last exercise):
- 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56
- Add these further keys: 2, 6,12
- Delete these keys: 4, 5, 7, 3, 14
-

Analysis of B-Trees

- The maximum number of items in a B-tree of order m and height h:

root	$m - 1$
level 1	$m(m - 1)$
level 2	$m^2(m - 1)$
. . .	
level h	$m^h(m - 1)$

- So, the total number of items is

$$(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) =$$

$$[(m^{h+1} - 1)/ (m - 1)]\ (m - 1) = \boldsymbol{m^{h+1} - 1}$$

- When $m = 5$ and $h = 2$ this gives $5^3 - 1 = 124$

Reasons for using B-Trees

- When searching tables held on disc, the cost of each disc transfer is high but doesn't depend much on the amount of data transferred, especially if consecutive items are transferred
  - If we use a B-tree of order 101, say, we can transfer each node in one disc read operation
  - A B-tree of order 101 and height 3 can hold $101^4 – 1$ items (approximately 100 million) and any item can be accessed with 3 disc reads (assuming we hold the root in memory)
- If we take *m* = 3, we get a **2-3 tree**, in which non-leaf nodes have two or three children (i.e., one or two keys)
  - B-Trees are always balanced (since the leaves are all at the same level), so 2-3 trees make a good type of balanced tree

Comparing Trees

- Binary trees
  - Can become *unbalanced* and *lose* their good time complexity (big O)
  - AVL trees are strict binary trees that *overcome the balance problem*
  - Heaps remain balanced but only *prioritise* (not order) the keys

- Multi-way trees
  - B-Trees can be *m*-way, they can have any (odd) number of children
  - One B-Tree, the 2-3 (or 3-way) B-Tree, *approximates* a permanently balanced binary tree, exchanging the AVL tree's balancing operations for insertion and (more complex) deletion operations

B+ Trees

- No data in internal nodes
- Leaf nodes have data, internal nodes have just keys.
- Keys are used for directing a search to the proper leaf.
- If a target key is less than a key in an internal node, then the pointer just to its left is followed.
- If a target key is greater or equal to the key in the internal node, then the pointer to its right is followed.
- B+ Tree combines features of Indexed Sequential Access Method and B Trees.

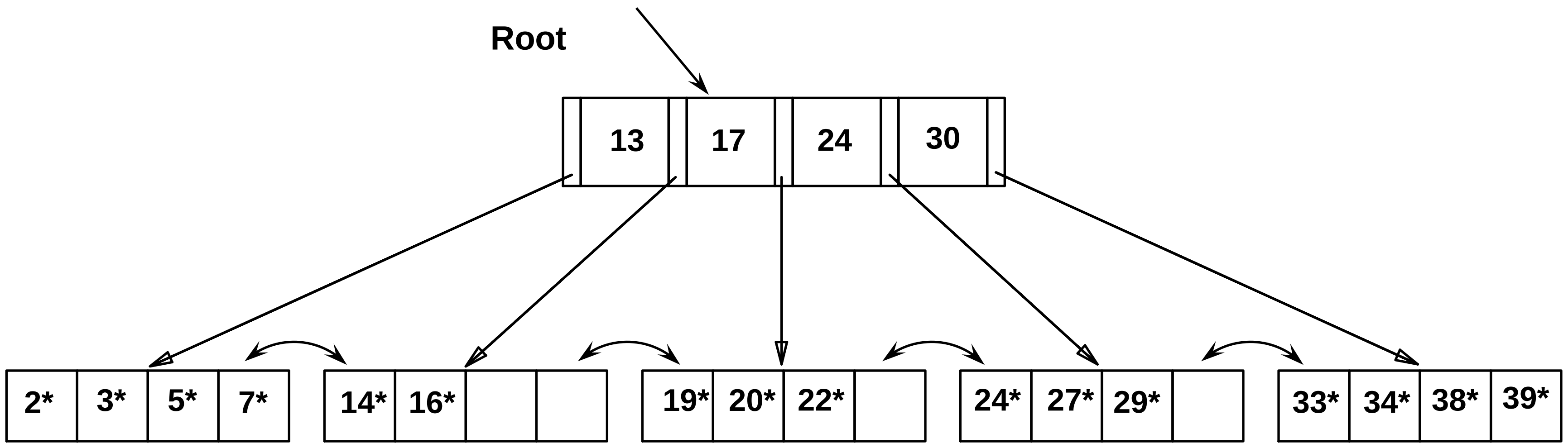
B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 3: 133^3 = 2,352,637 entries
  - Height 4: 133^4 = 312,900,700 entries
- 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

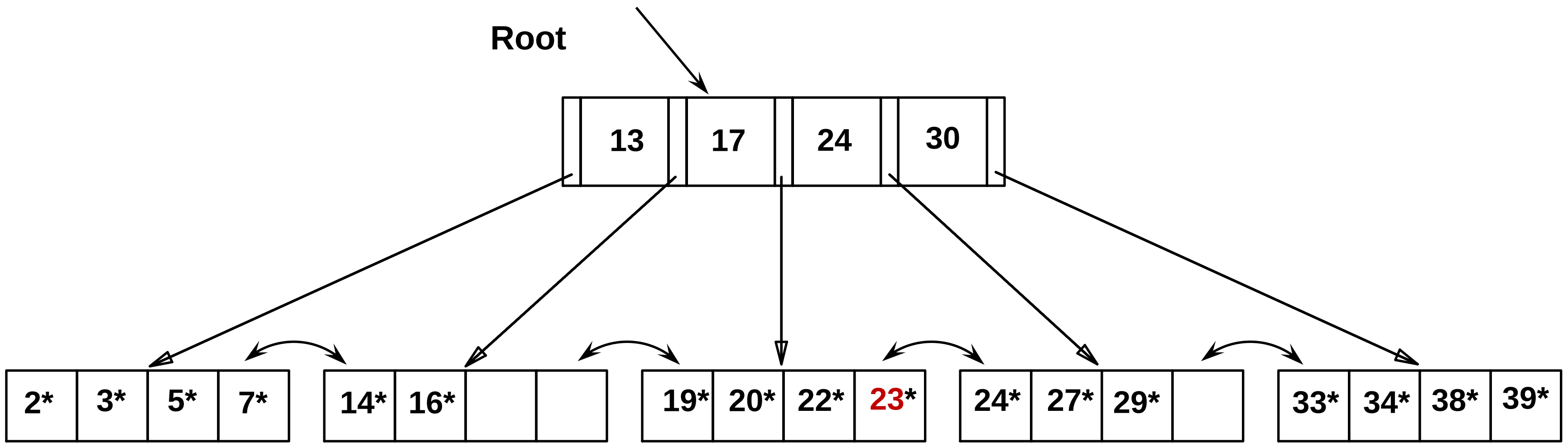
B+ Trees: Summary

- Searching:
  - $\log_d(n)$ – Where *d* is the order, and *n* is the number of entries
- Insertion:
  - Find the leaf to insert into
  - If full, split the node, and adjust index accordingly
  - Similar cost as searching
- Deletion
  - Find the leaf node
  - Delete
  - May not remain half-full; must adjust the index accordingly

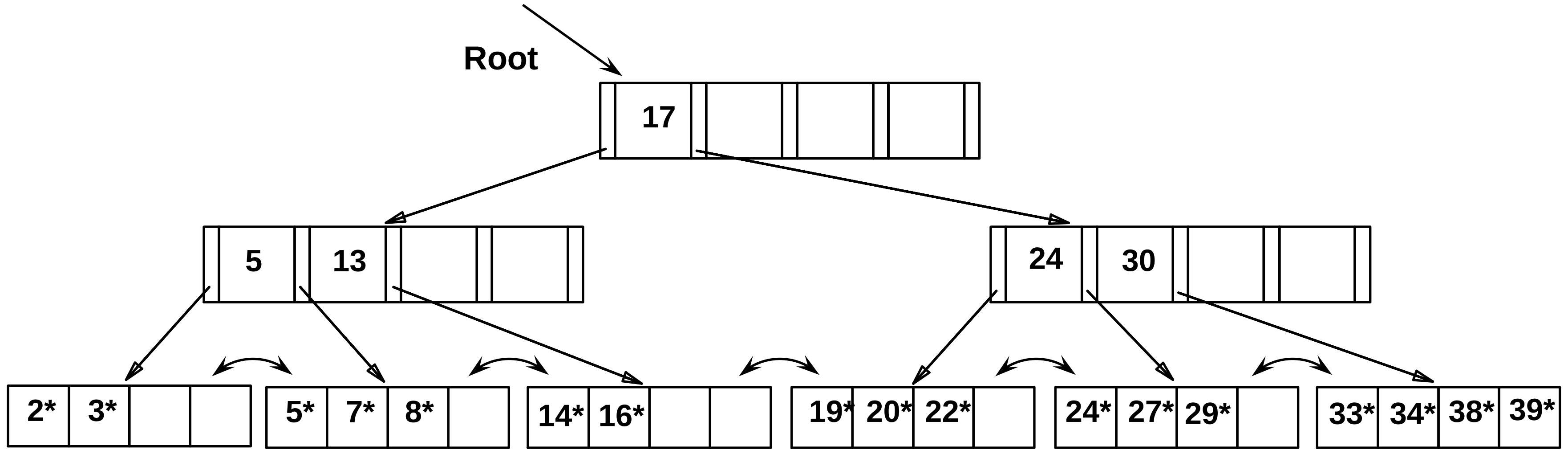
Insert 23*



No splitting required.



Example B+ Tree - Inserting 8*



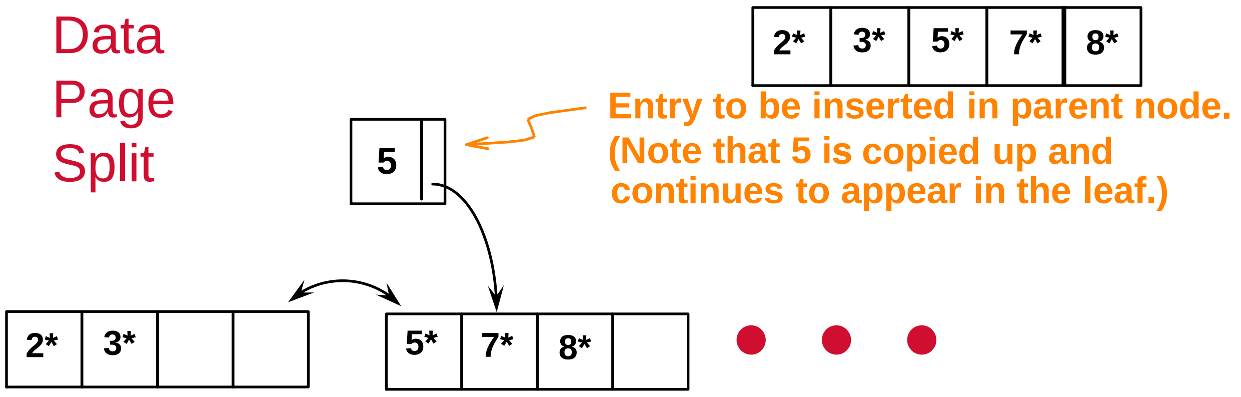
❖ Notice that root was split, leading to increase in height.

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

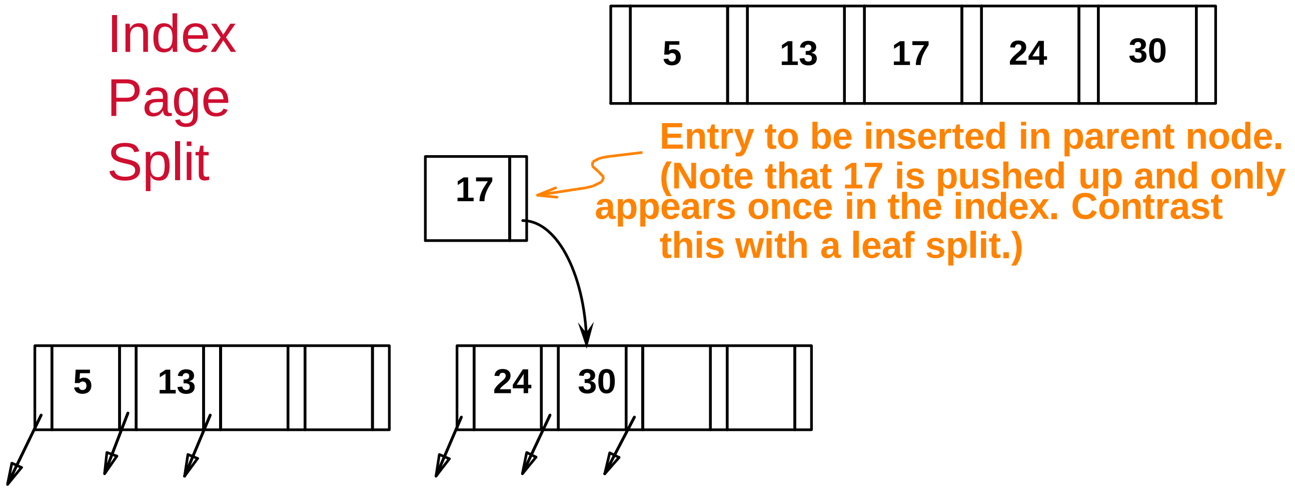
Data vs. Index Page Split (from previous example of inserting "8")

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

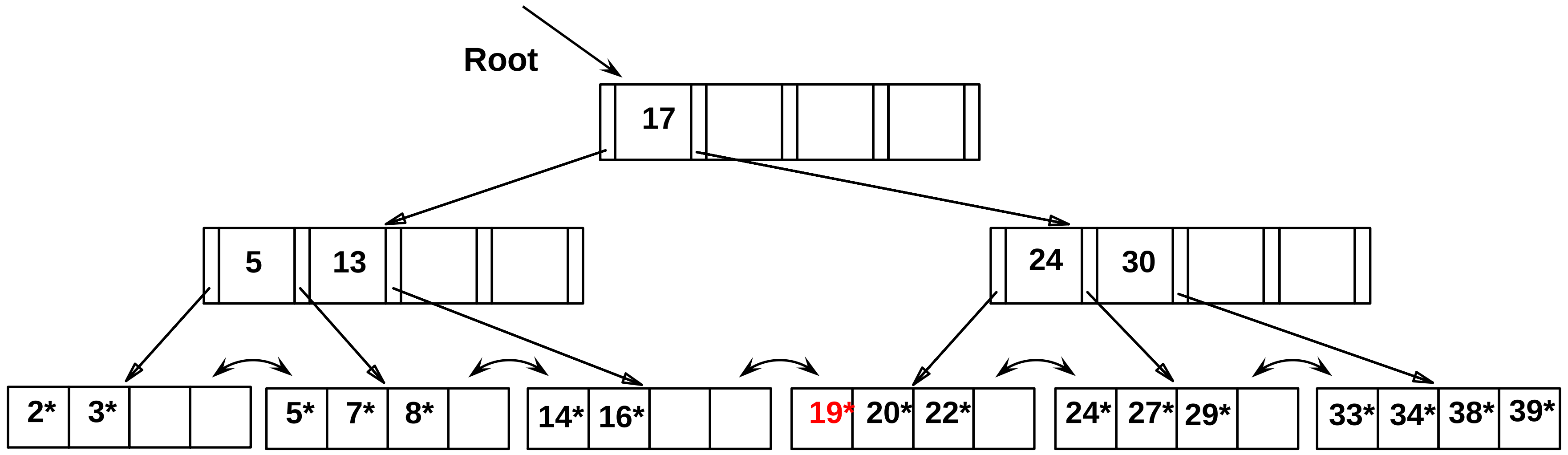
Data Page Split


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*

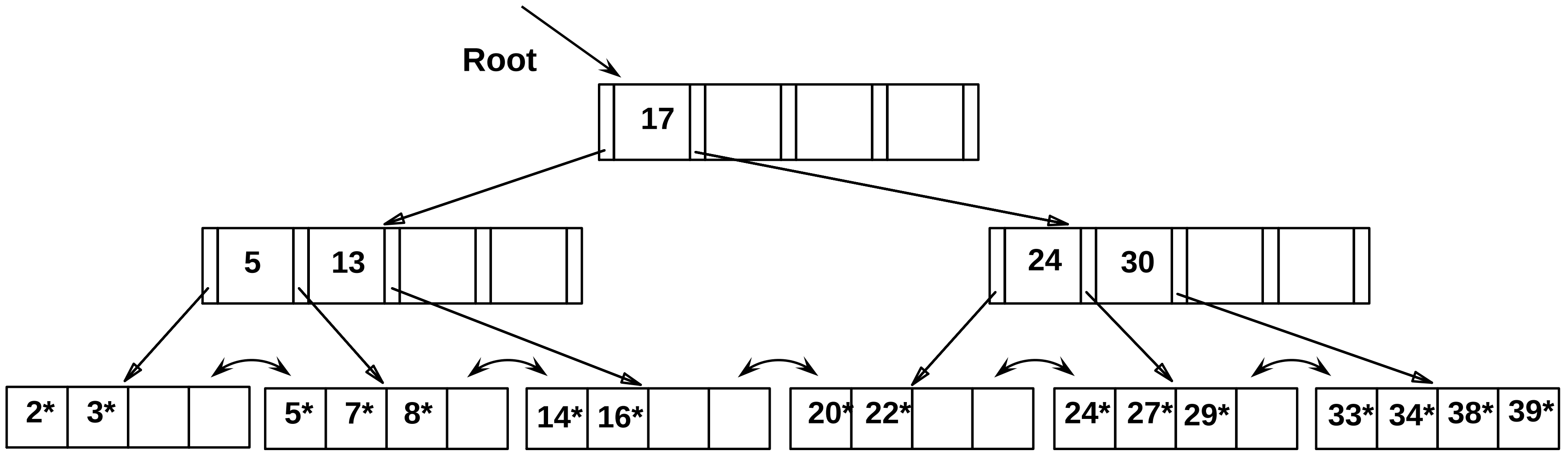
Index Page Split


5
13
17
24
30
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.)
17
5
13
24
30

Delete 19*

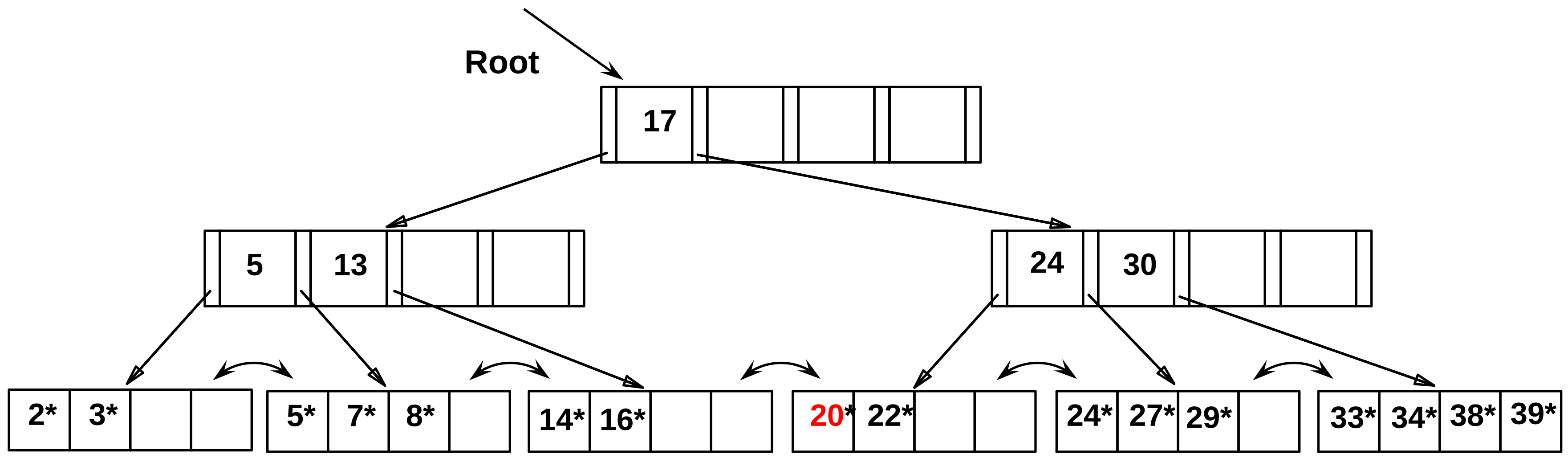


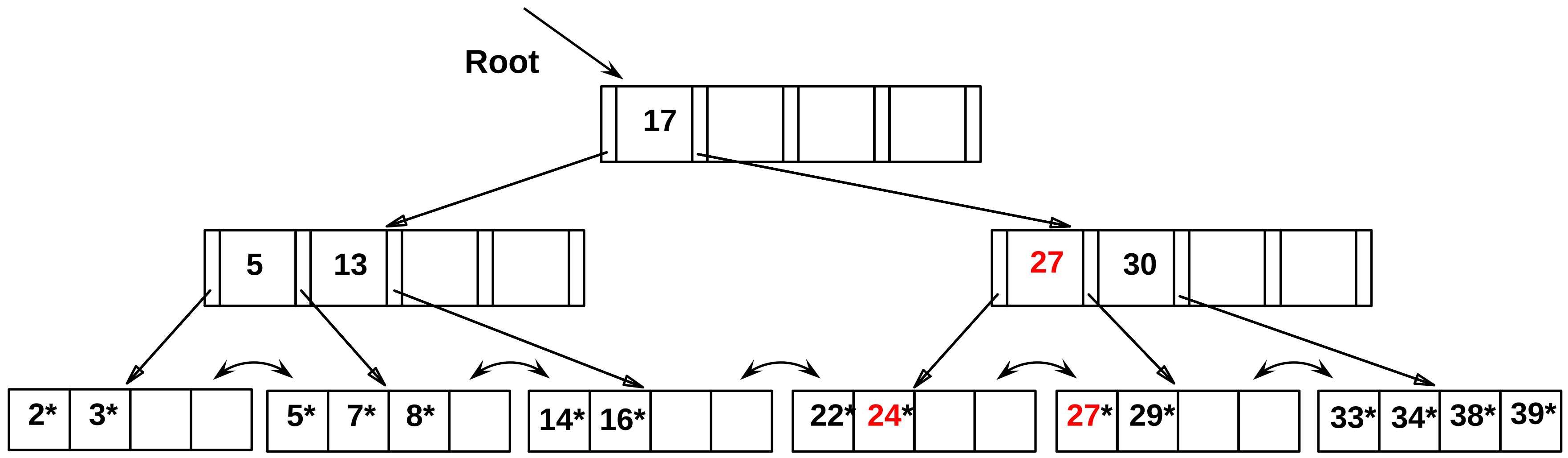
Root
17
5
13
24
30
2* 3*
5* 7* 8*
14* 16*
19* 20* 22*
24* 27* 29*
33* 34* 38* 39*



Root
17
5
13
24
30
2* 3*
5* 7* 8*
14* 16*
20* 22*
24* 27* 29*
33* 34* 38* 39*

Delete 20* ...

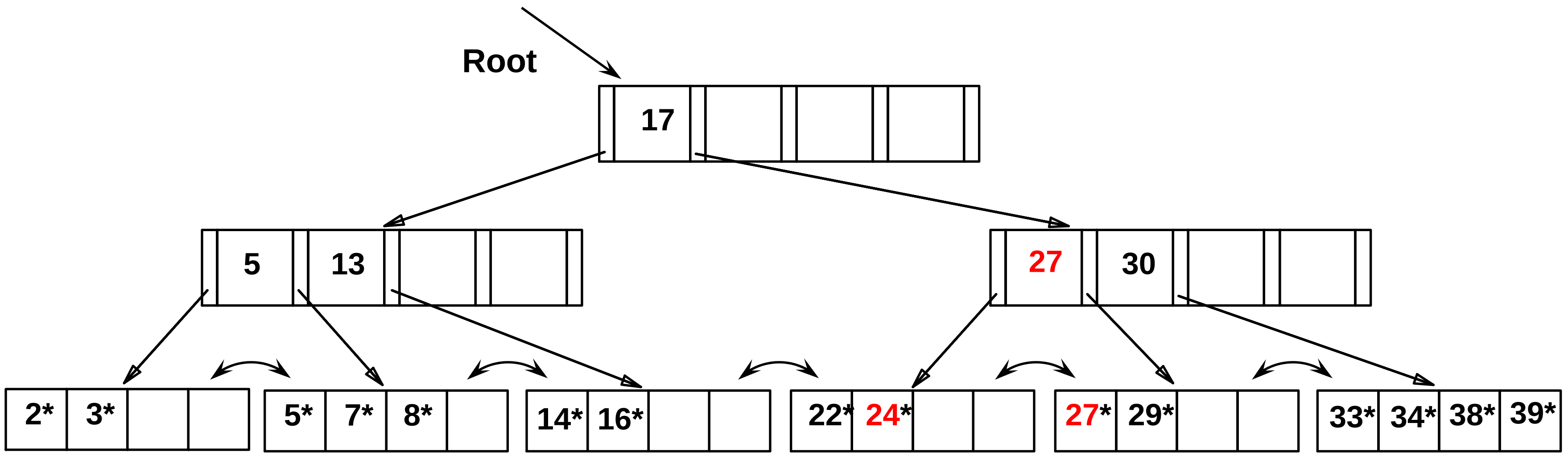

Root
17
5
13
24
30
2*
3*
5*
7*
8*
14*
16*
20*
22*
24*
27*
29*
33*
34*
38*
39*

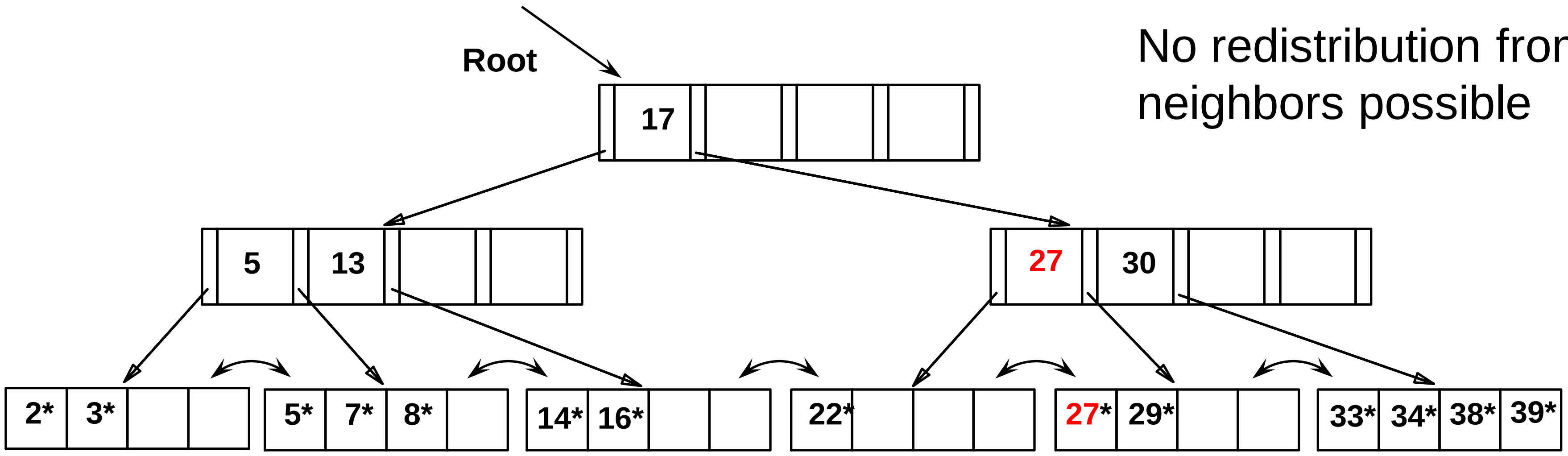

Root
17
5
13
27
30
2*
3*
5*
7*
8*
14*
16*
22*
24*
27*
29*
33*
34*
38*
39*

Delete 19* and 20* ...

- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.
- Further deleting 24* results in more drastic changes

Delete 24* ...

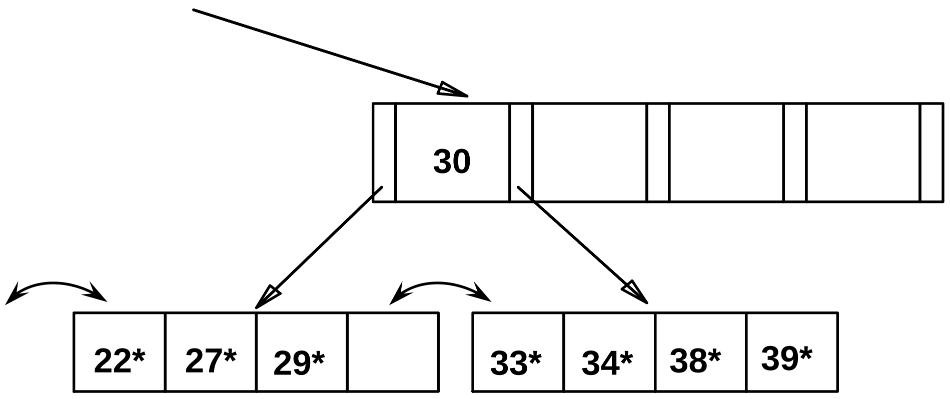


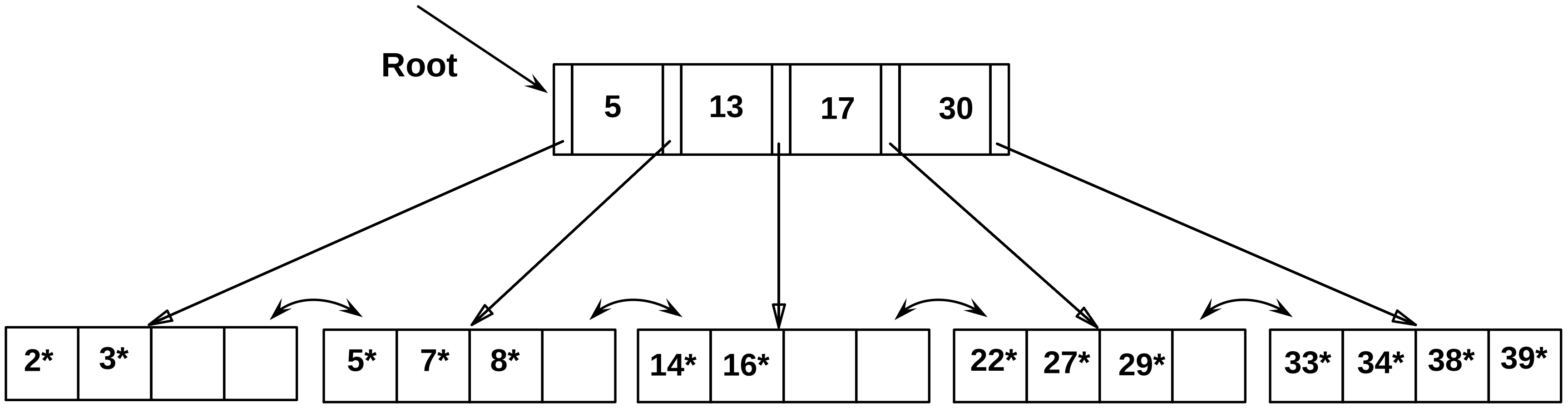


No redistribution from neighbors possible

Deleting 24*

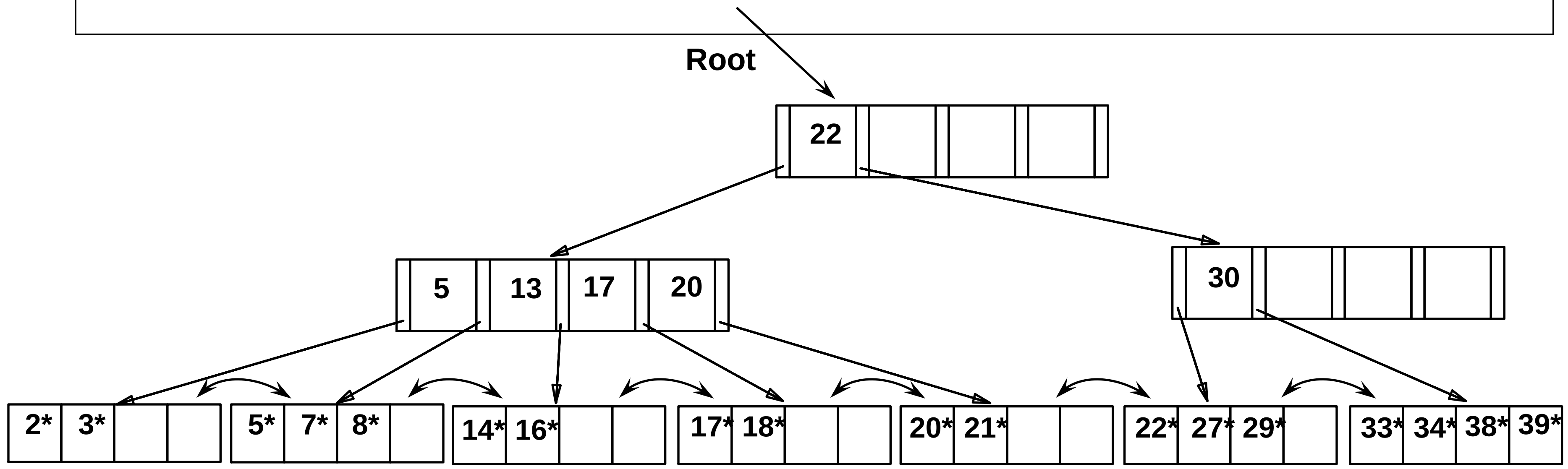
- Must merge. [22] [27,29]
- Reordering [5 13] [17] [30] as below





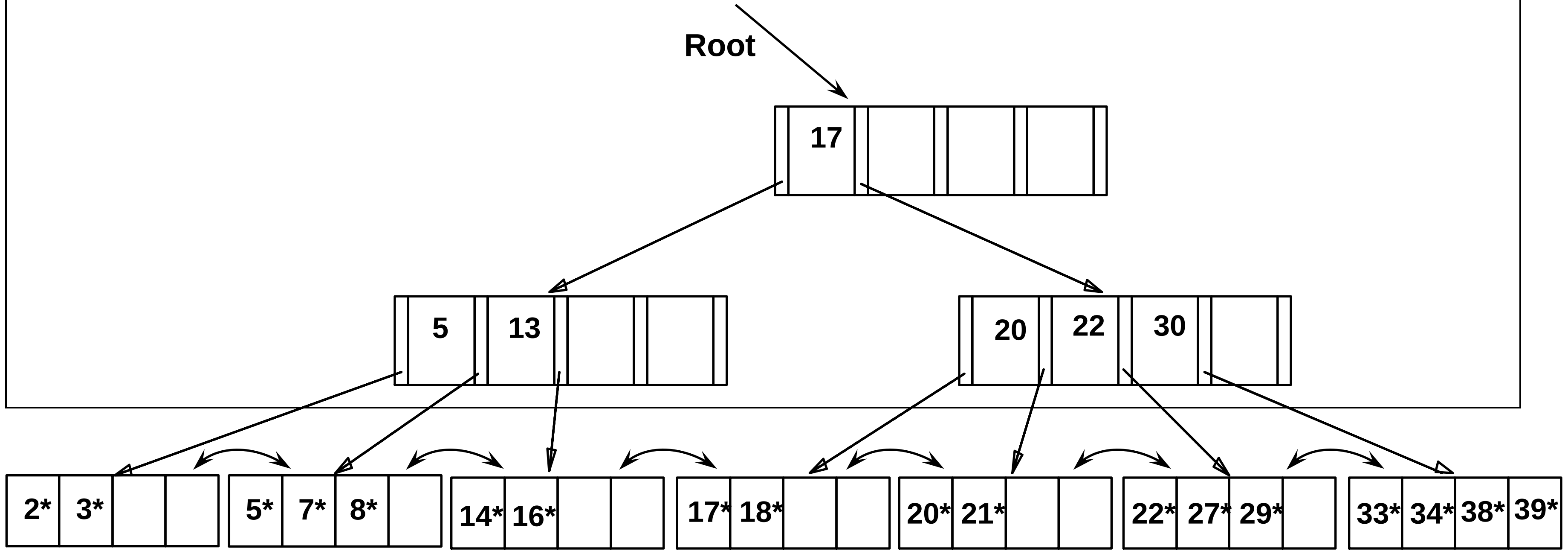
Example of Non-leaf Re-distribution

- 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

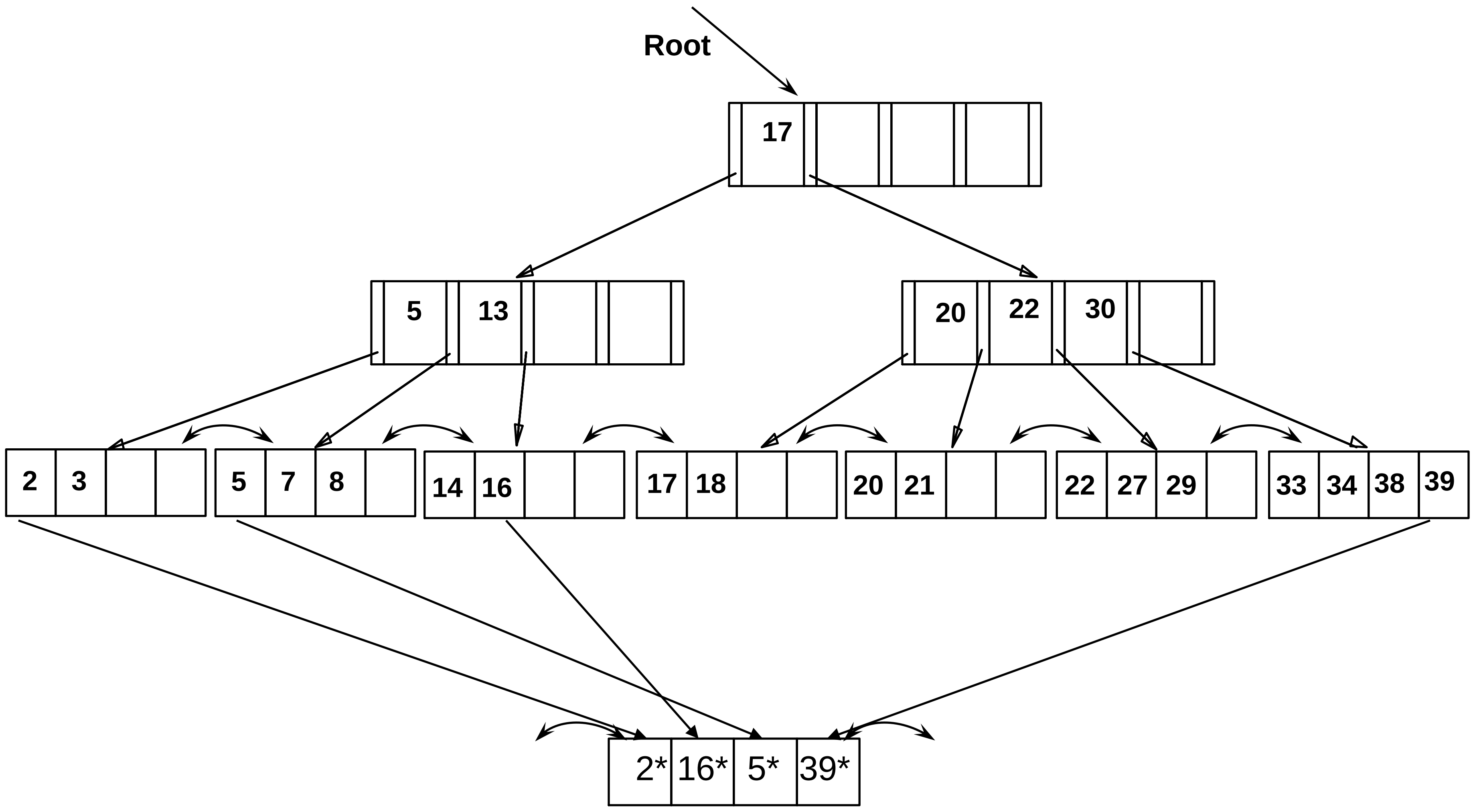
- Intuitively, entries are re-distributed by `*pushing through*' the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.



Primary vs Secondary Index

- Note: We were assuming the data items were in sorted order
  - This is called *primary* index
- *Secondary* index:
  - Built on an attribute that the file is not sorted on.

A Secondary B+-Tree index



Primary vs Secondary Index

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