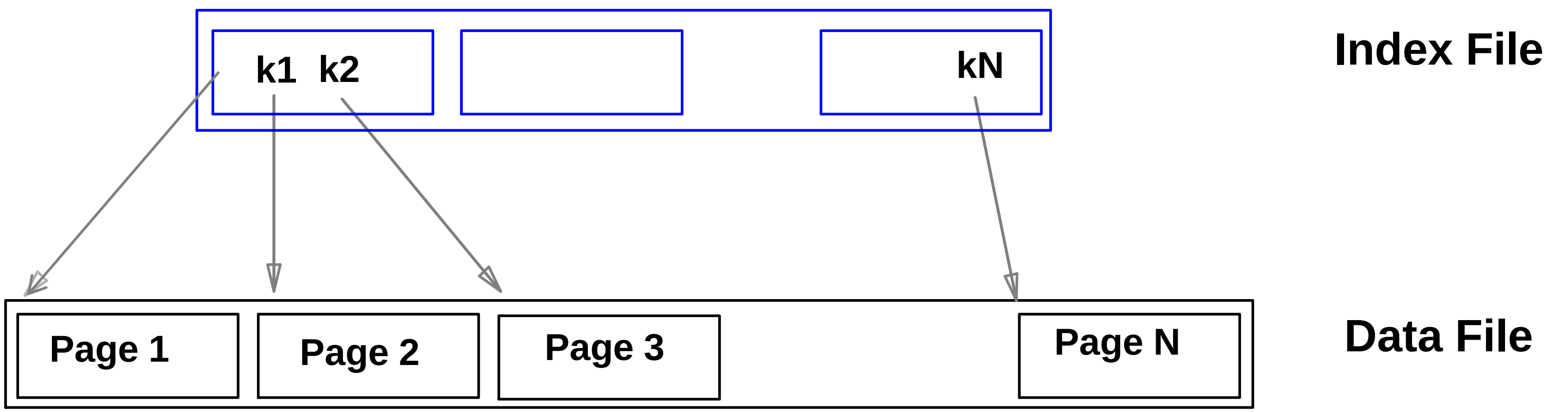
Tree-Structured Indexing

Range Searches

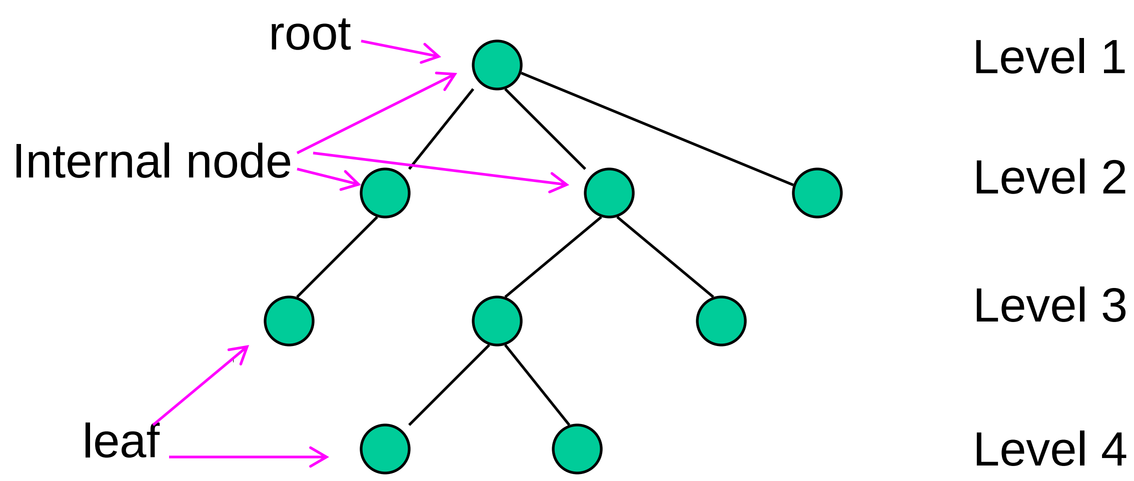
- *``Find all students with 3.0 < gpa < 3.5''*
  - If records are sorted on gpa, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.
- Simple idea: Create an `index' file.



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

B+ Tree: Most Widely Used Index

- General concept of a tree:

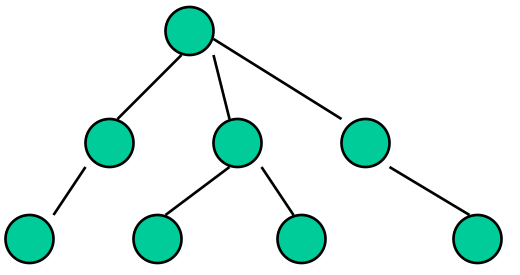


I. Height of a node: its distance to the root

II. If a higher level node is connected to a lower level node, then the higher level node is called a parent (grandparent, ancestor, etc.) of the lower level node

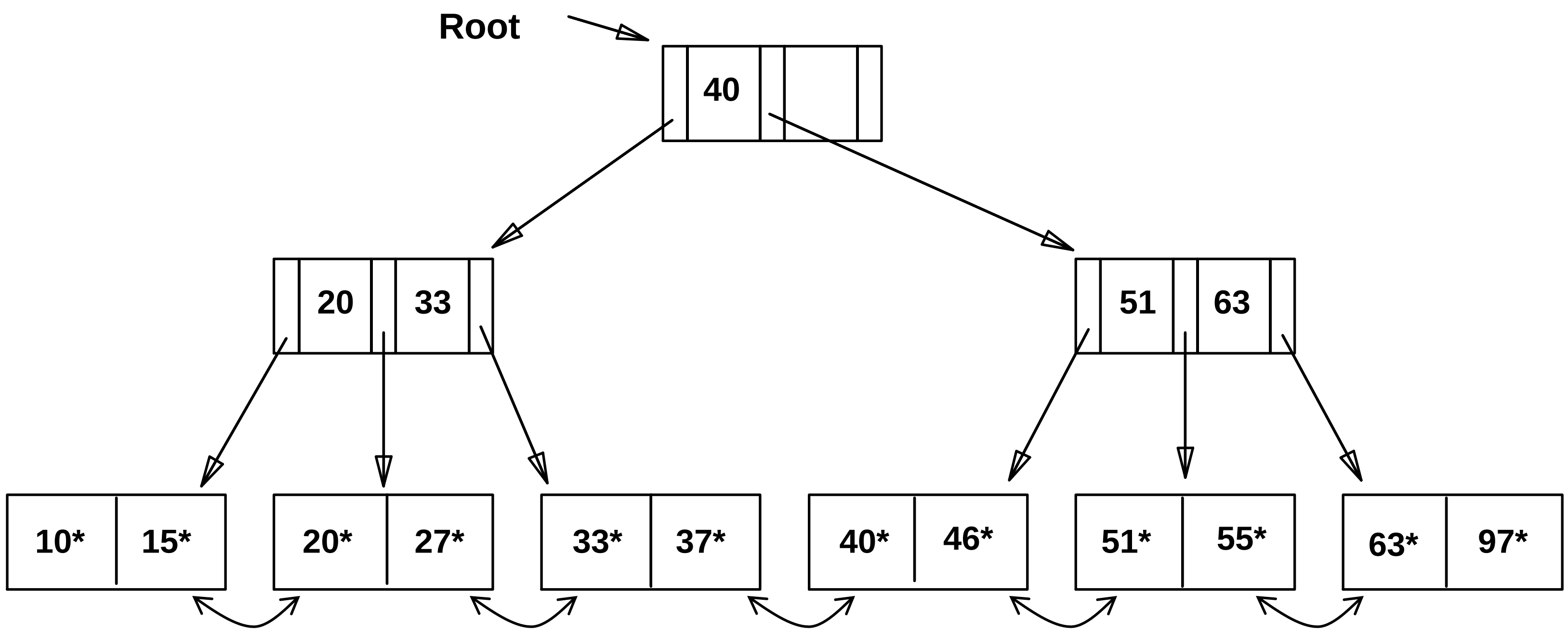
B+ Tree (cont.)

- Balanced tree: all leafs at the same level
  - Example:

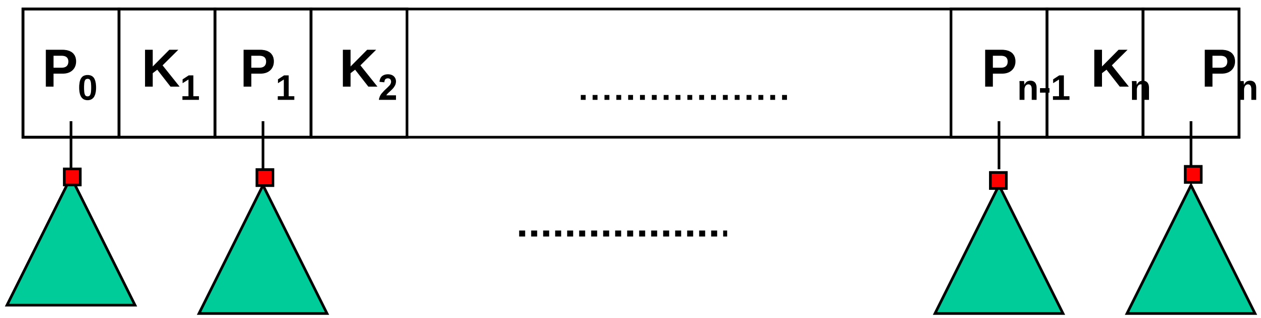


- Structure of a B+ tree:
  - It is ***balanced*:** all leaf nodes at the same level
  - It's node has a special structure

An example of a B+ tree



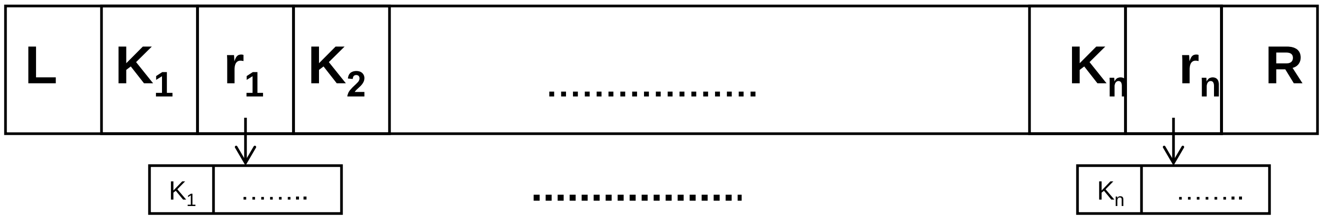
B+ tree: Internal node structure



Each P_i is a pointer to a child node, each K_i is a search key value
Pointers outnumber search key values by *exactly one.*

- **Requirements:**
  - **$K_1 < K_2 < \ldots < K_n$**
  - **If the node is not the root, we require $d \leq n \leq 2d$ where d is a pre-determined value for this B+ tree, called its *order***
  - **If the node is the root, we require $1 \leq n \leq 2d$**
  - **For any search key value K in the subtree pointed by P_i,**
    - **If $P_i = P_0$, we require $K < K_1$**
    - **If $P_i = P_1, \ldots, P_{n-1}$, we require $K_i \leq K < K_{i+1}$**
    - **If $P_i = P_n$, we require $K_n \leq K$**

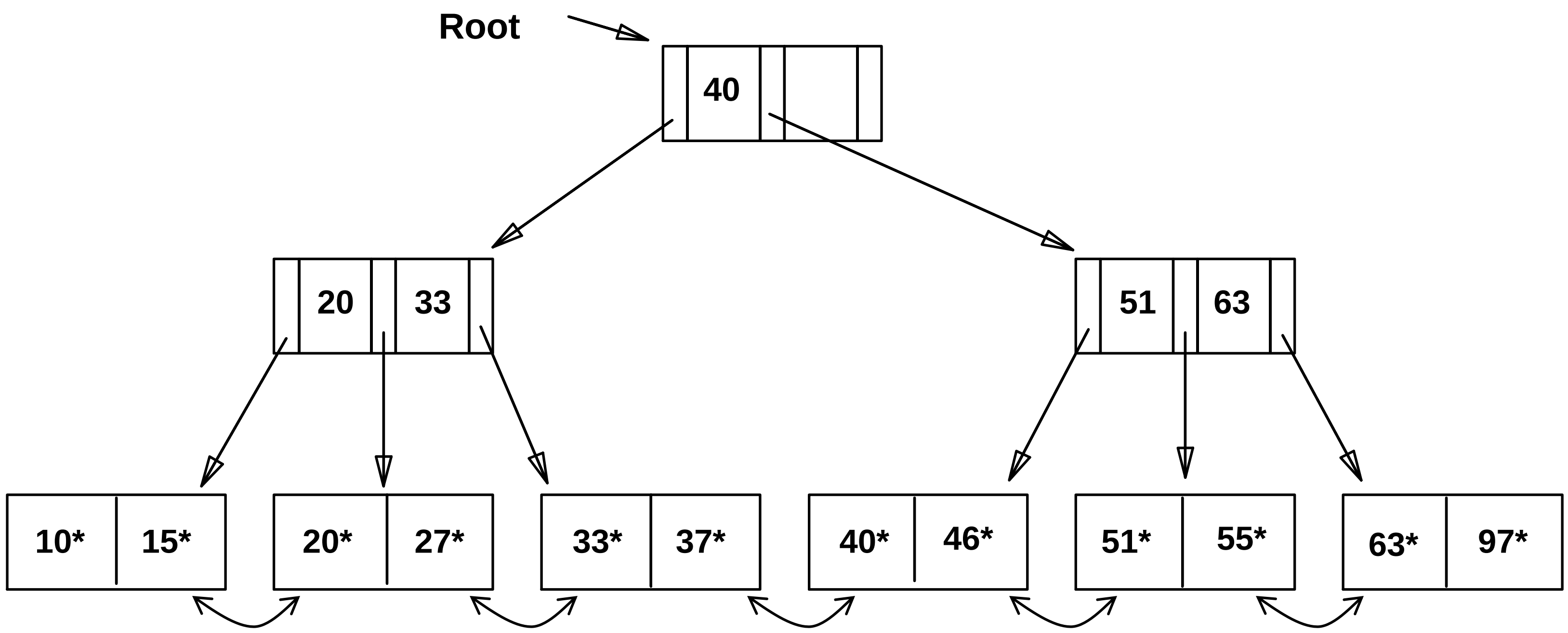
B+ tree: leaf node structure



- Each r_i is a pointer to a record that contains search key value K_i
- L points to the left neighbor, and R points to the right neighbor
- $K_1 < K_2 < \ldots < K_n$
- We require $d \leq n \leq 2d$ where 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: A B+ tree with order of 1

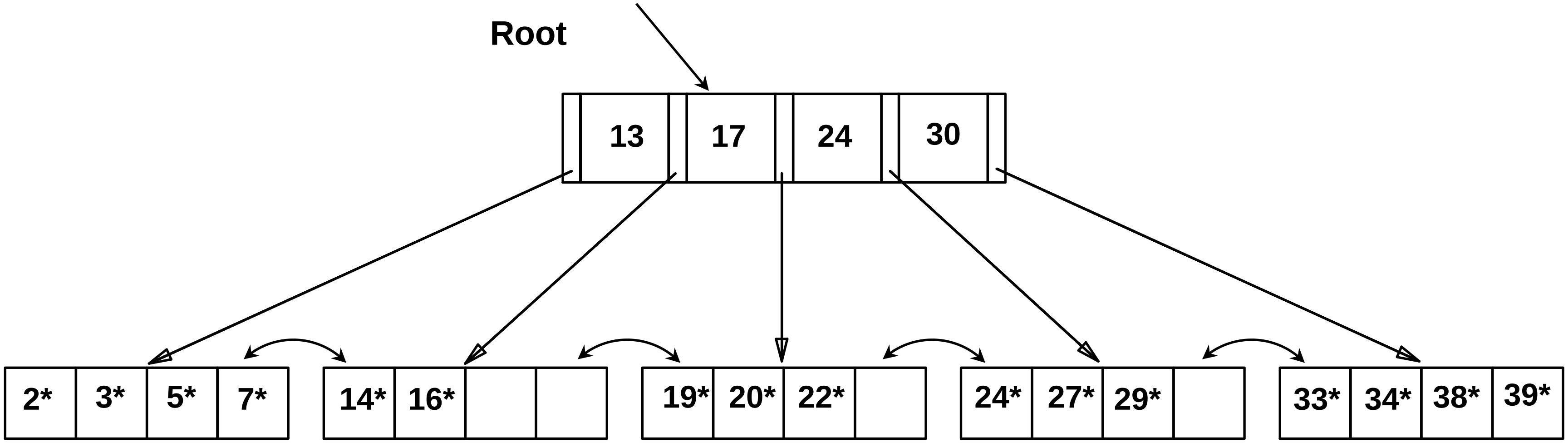
- Each node must hold at least 1 entry, and at most 2 entries



- Given search key values 27, 51, 64, how to find the rids?
  - Search begins at the root, and key comparisons direct it to a leaf

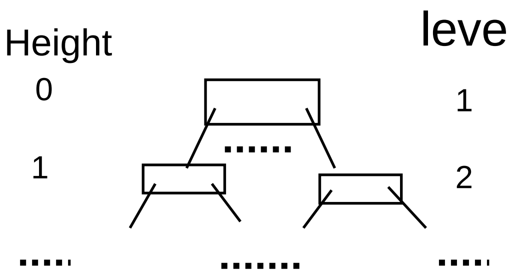
Example: a B+ tree with order 2

- Search for 5*, 15*, 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.



Cost for searching a value in B+ tree

- In general nodes are pages
- Let H be the height of the B+ tree: need to read H+1 pages to reach a leaf node
- Let F be the (average) number of pointers in a node (for internal node, called *fanout*)
  - Level 1 = 1 page = F^0 page
  - Level 2 = F pages = F^1 pages
  - Level 3 = Fx F pages = F^2 pages
  - Level H+1 = …….. = F^H pages (i.e., leaf nodes)
  - Suppose there are D data entries. So there are D/(F-1) leaf nodes
  - D/(F-1) = F^H. That is, H = $\log_F(\frac{D}{F-1})$



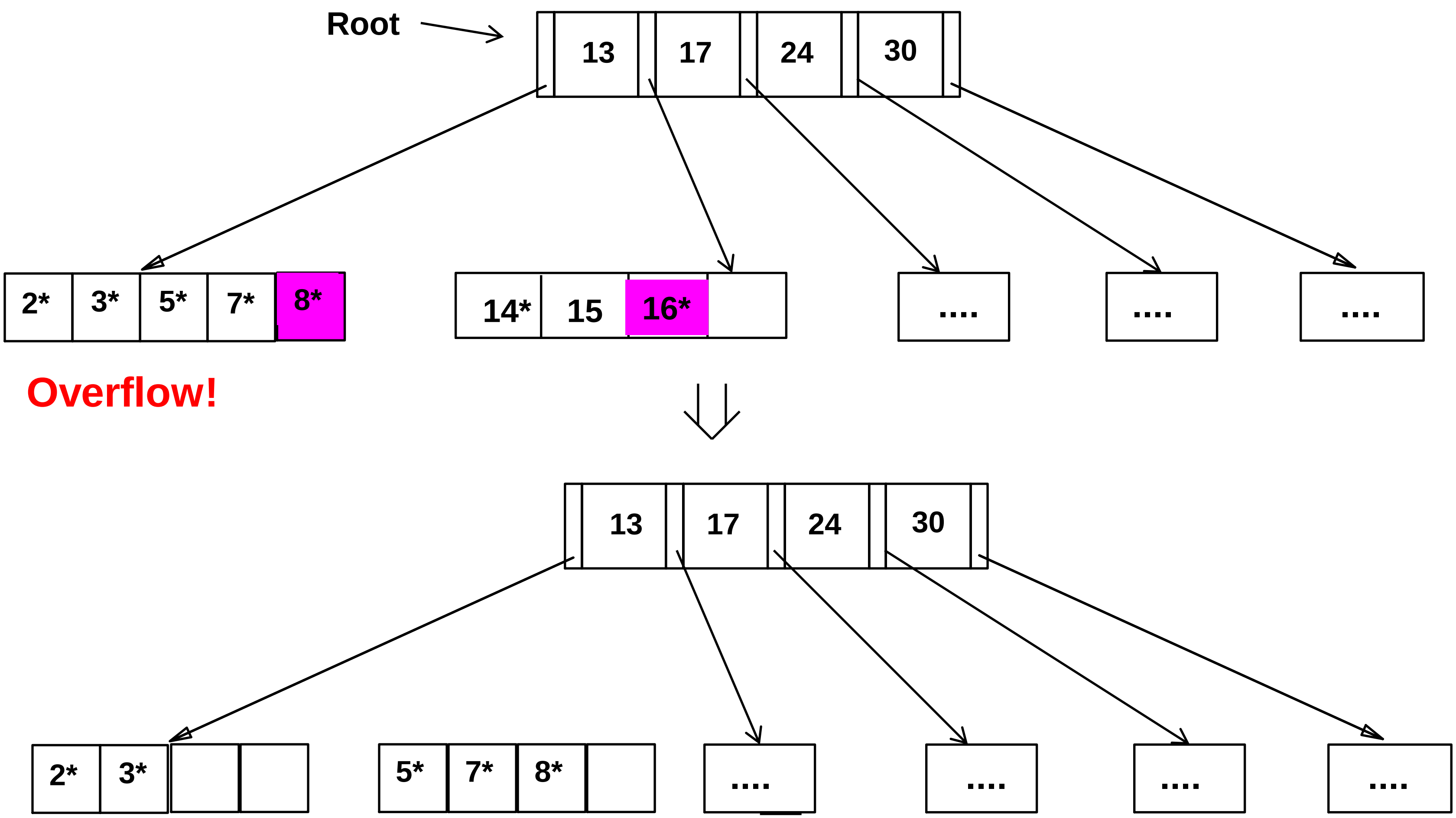
B+ Trees in Practice

- Typically, a node is a page
- Typical order: 100. Typical fill-factor: 67%.
  - 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

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, 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 an internal 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.*

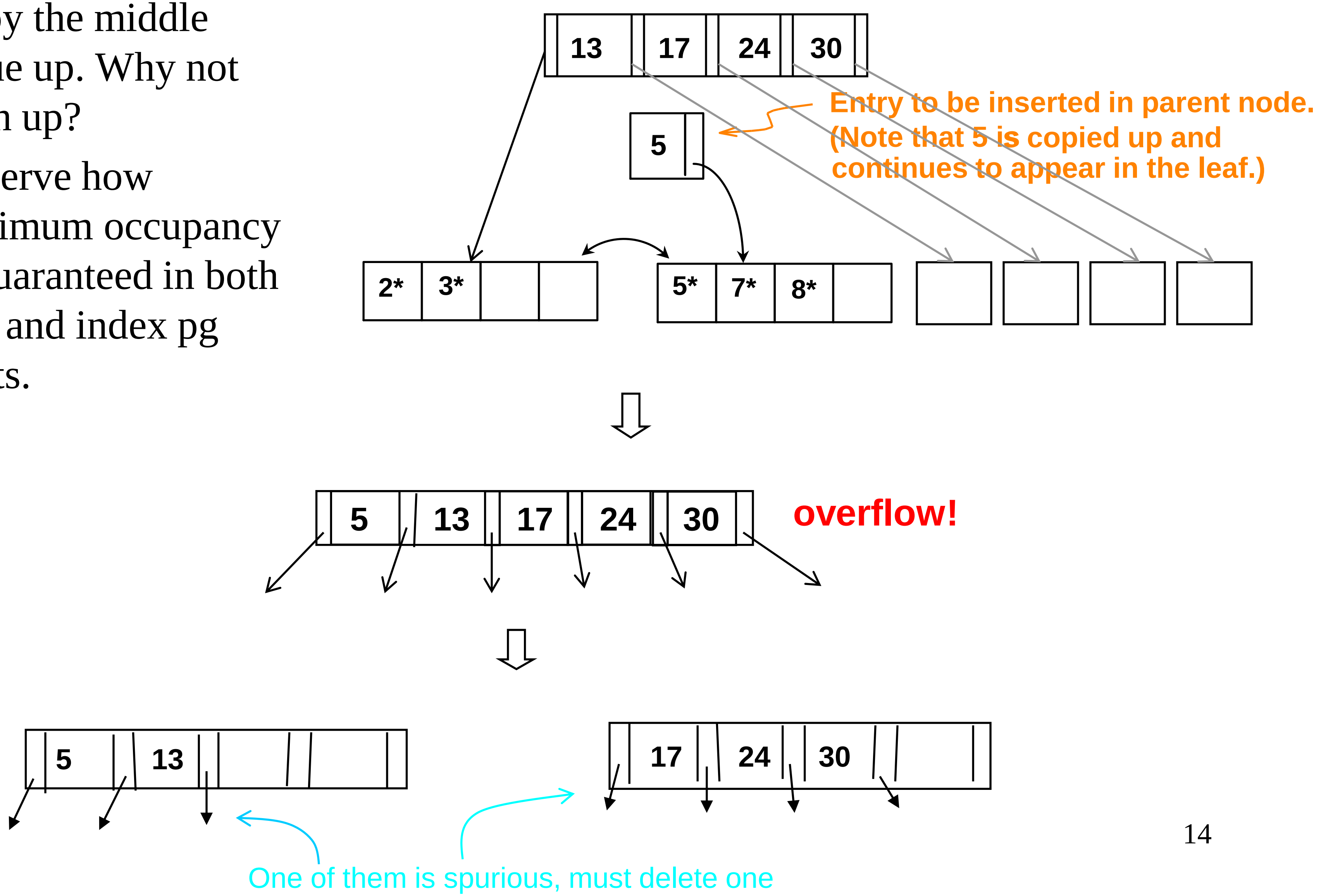
Inserting 16*, 8* into Example B+ tree



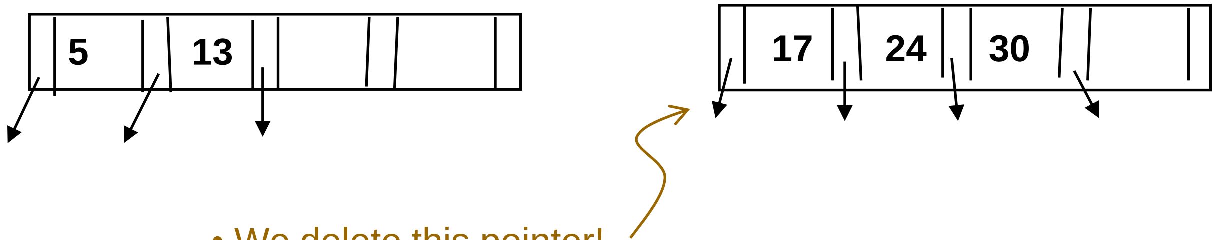
One more child generated, must add one more pointer to its parent, thus one more key value as well.

Inserting 8* into Example B+ Tree (order 2)

- Copy the middle value up. Why not push up?
- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.

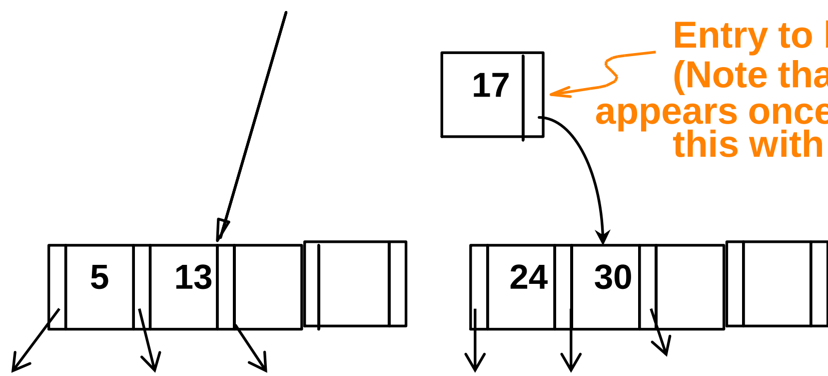


Insertion into B+ tree (cont.)



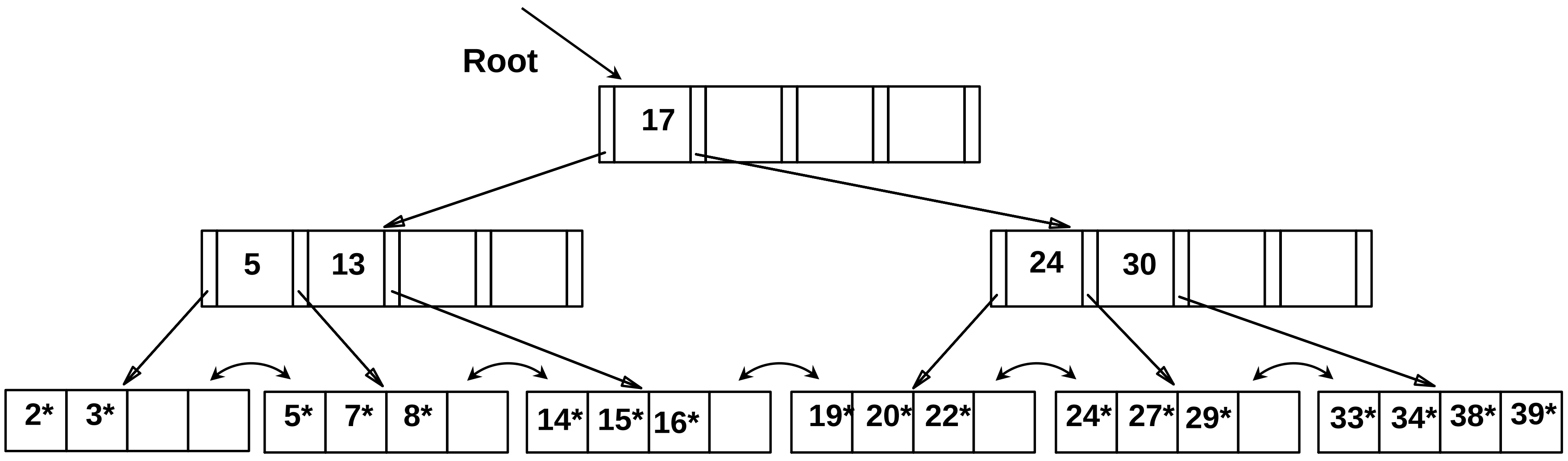
- We delete this pointer!
- But then we should also delete 17
- On the other hand, a value must be inserted into its parent.
- Therefore, we insert 17 to its parent

- **This explains why we must push up the middle entry, instead of copying it up, when we split an internal node.**



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*



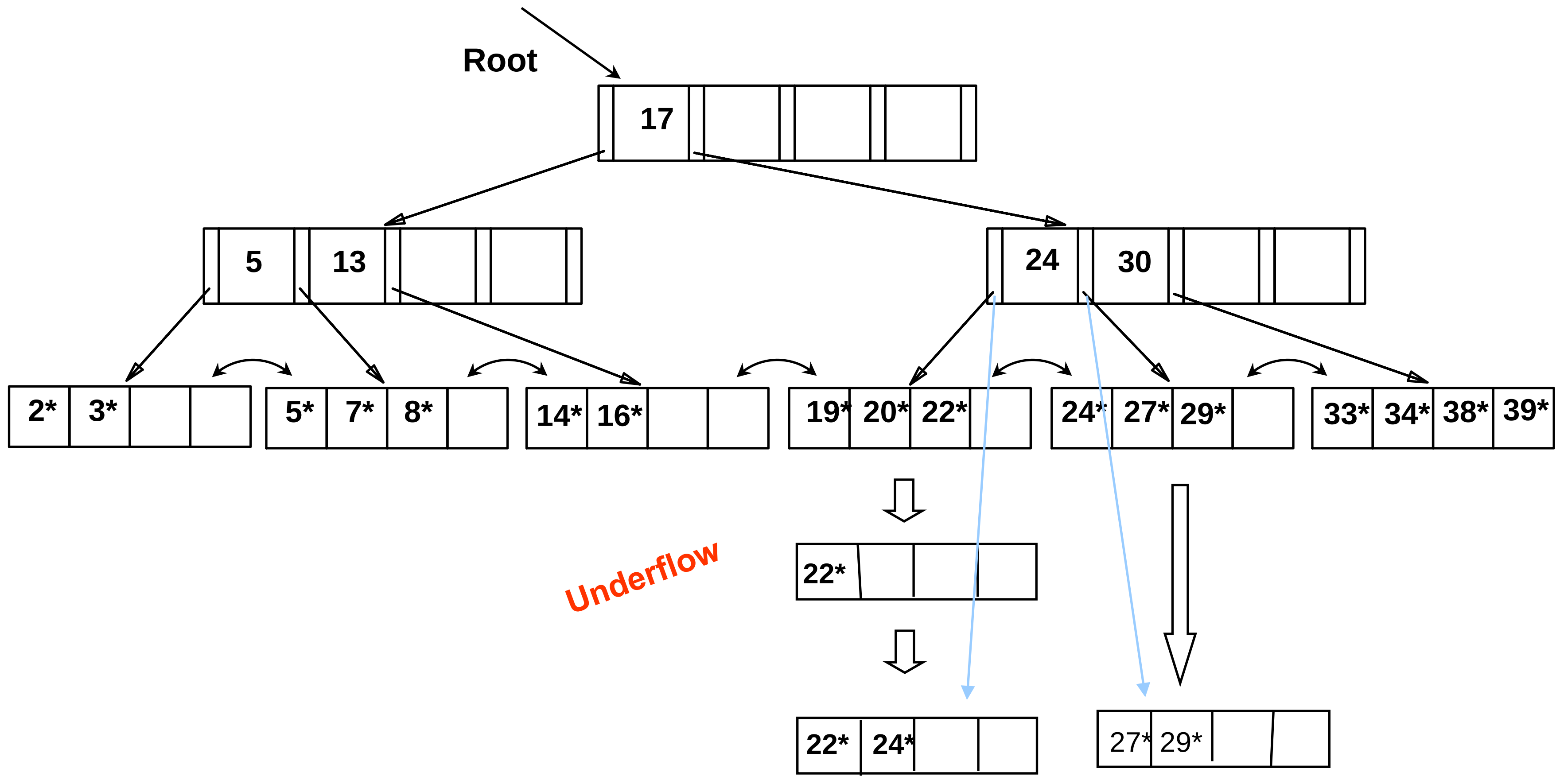
- Notice that root was split, leading to increase in height.

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

Deleting a Data Entry from a B+ Tree

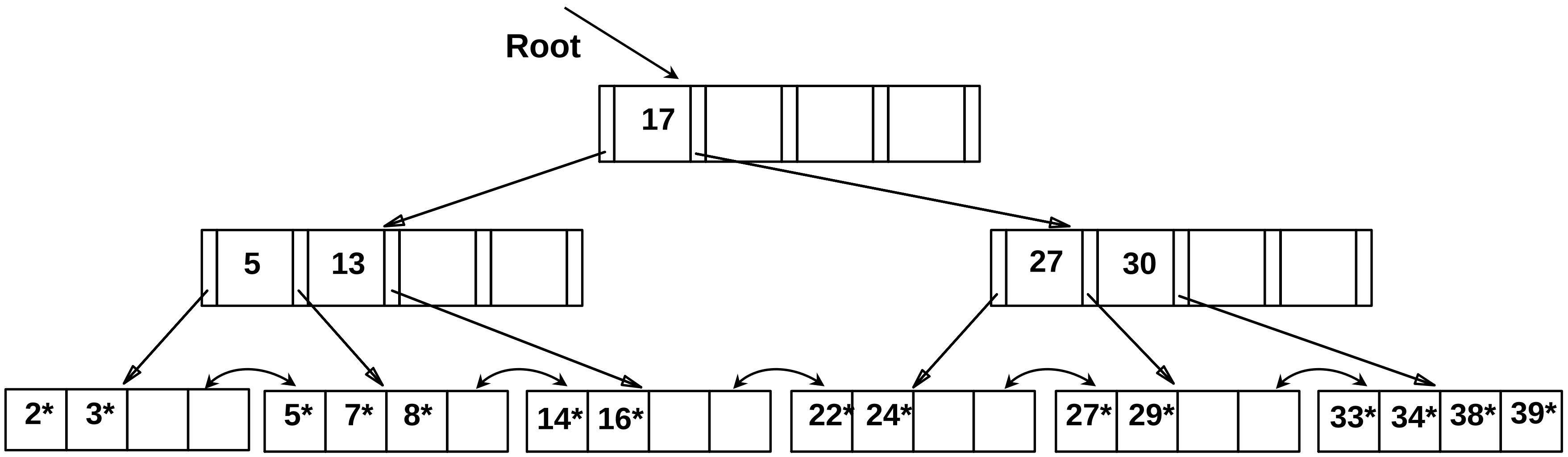
- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling* *(adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *L* and sibling.
- If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

Delete 19* and 20*

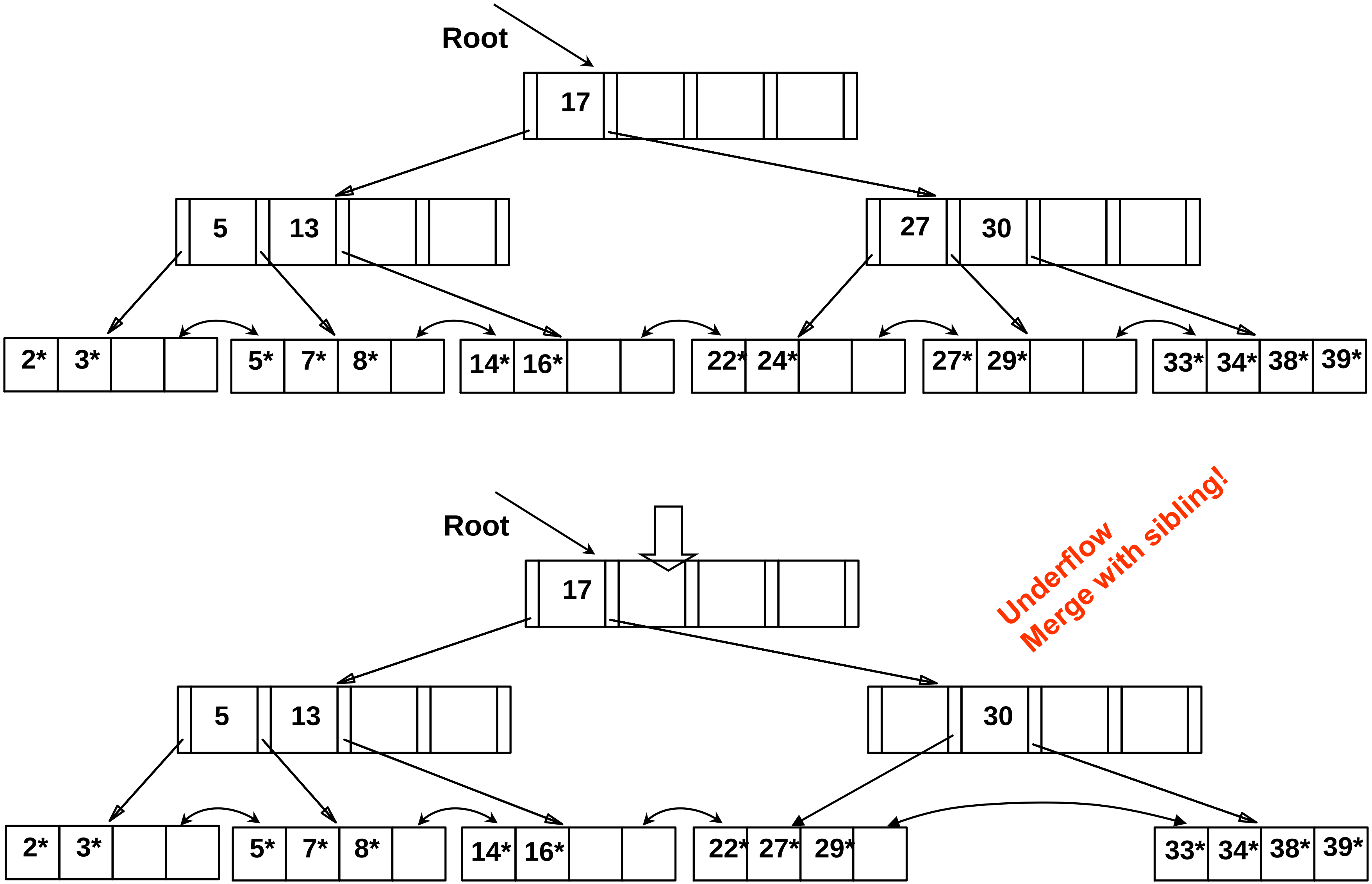


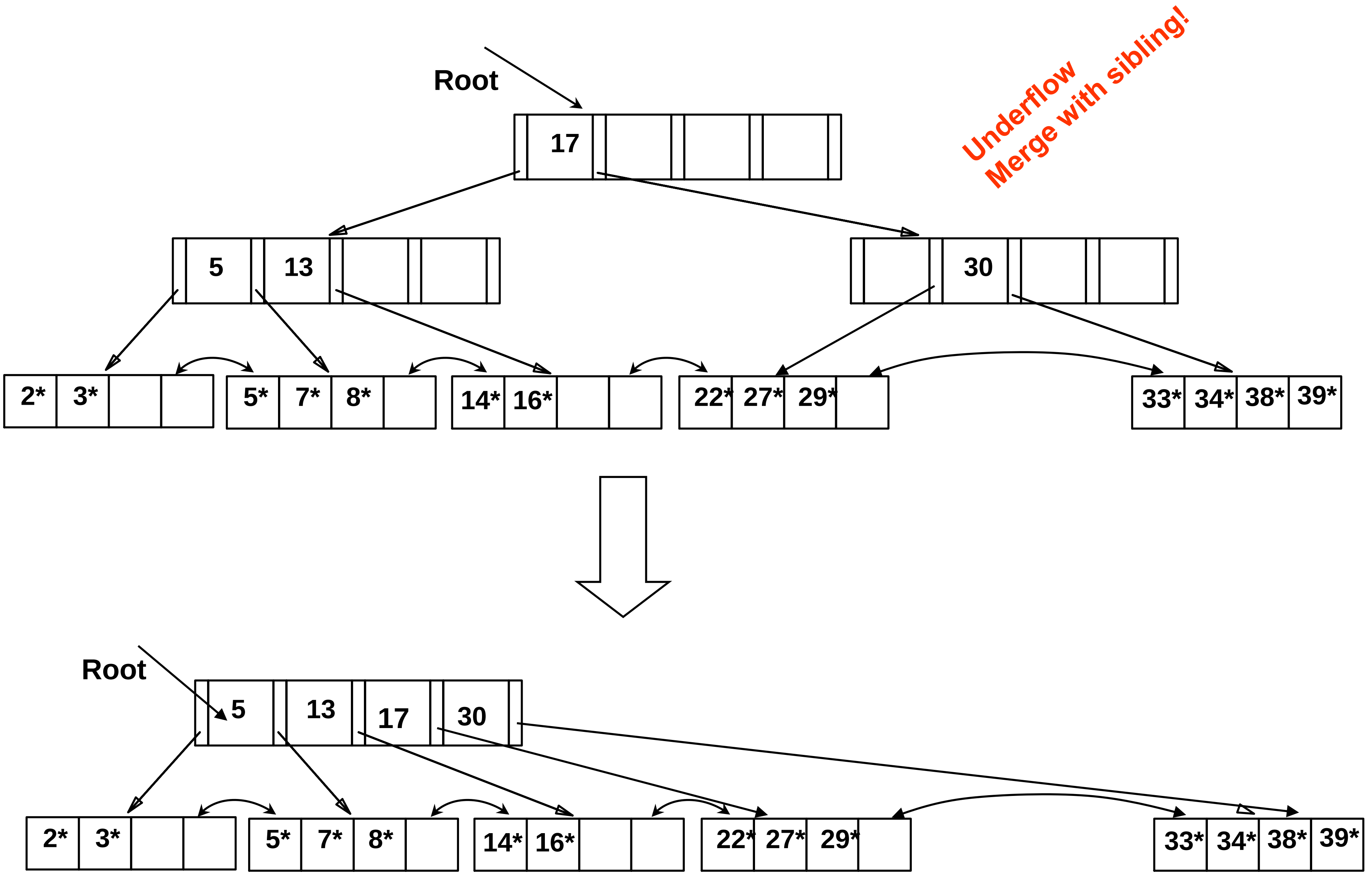
Have we still forgot something?

Deleting 19* and 20* (cont.)



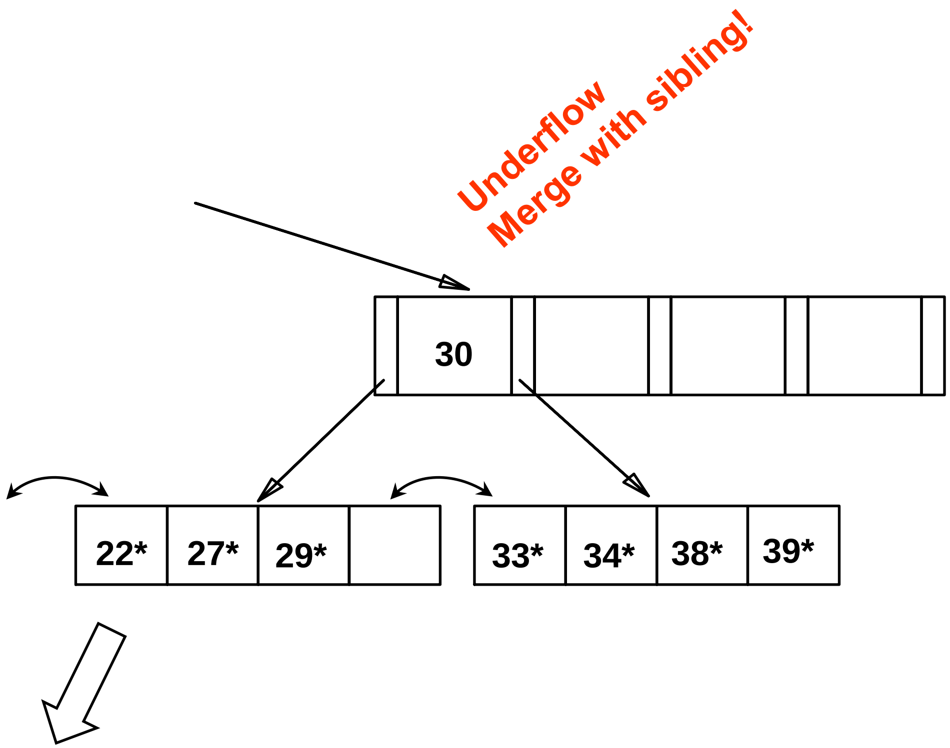
- Notice how 27 is *copied up*.
- But can we move it up?
- Now we want to delete 24
- Underflow again! But can we redistribute this time?

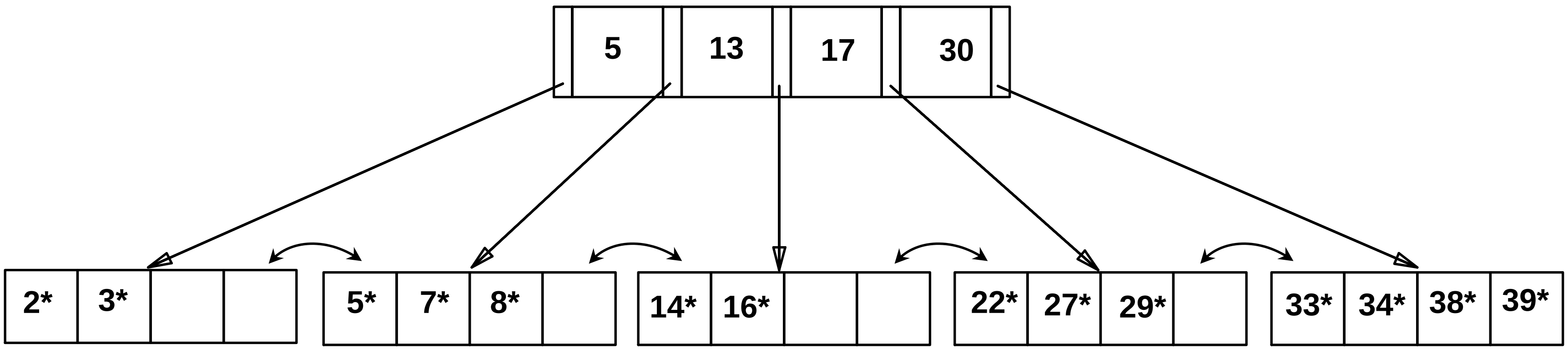

Root
17
5
13
27
30
2* 3*
5* 7* 8*
14* 16*
22* 24*
27* 29*
33* 34* 38* 39*
Root
17
5
13
30
2* 3*
5* 7* 8*
14* 16*
22* 27* 29*
33* 34* 38* 39*
Underflow
Merge with sibling!


Root
17
Underflow
Merge with sibling!
5
13
30
2*
3*
5*
7*
8*
14*
16*
22*
27*
29*
33*
34*
38*
39*
Root
5
13
17
30
2*
3*
5*
7*
8*
14*
16*
22*
27*
29*
33*
34*
38*
39*

Deleting 24*

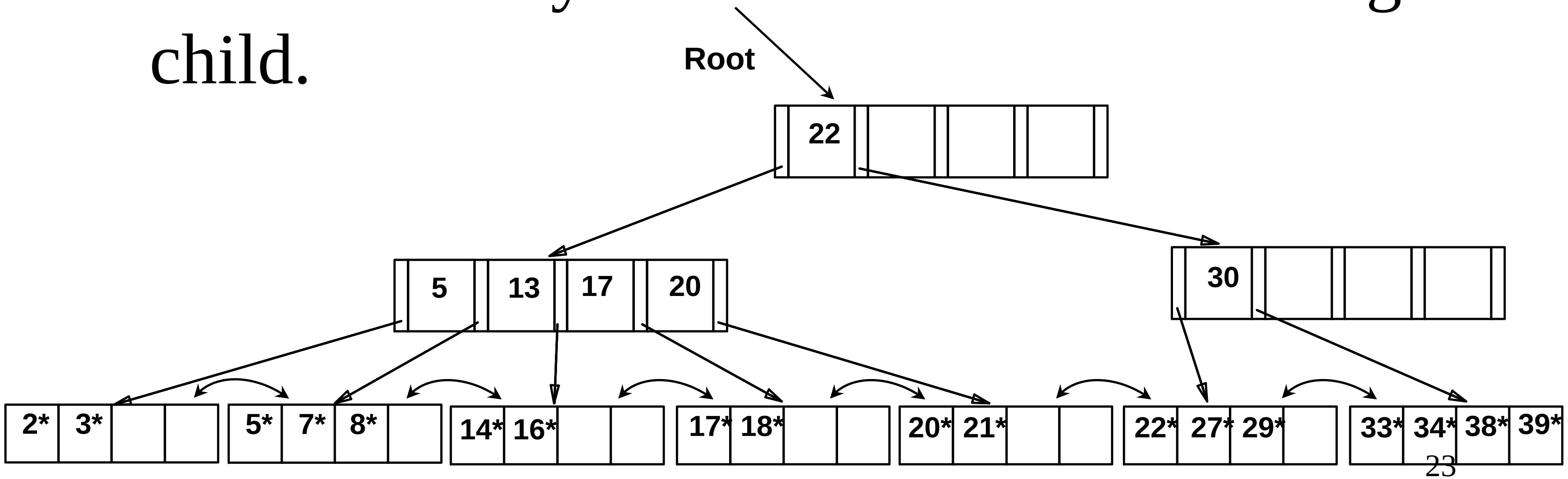
- Observe the two leaf nodes are merged, and 27 is discarded from their parent.
- Observe *`pull down'* of index entry (below).





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.