Spring 2017

B-TREES
(LOOSELY BASED ON THE COW BOOK: CH. 10)

Motivation

Consider the following table:

```
CREATE TABLE Tweets (
   uniqueMsgID INTEGER,       -- unique message id
   tstamp      TIMESTAMP,     -- when was the tweet posted
   uid         INTEGER,       -- unique id of the user
   msg         VARCHAR (140), -- the actual message
   zip           INTEGER       -- zipcode when posted
   );
```

Consider the following query, Q1:

```
SELECT * FROM Tweets
WHERE uid = 145;
```

And, the following query, Q2:

```
SELECT * FROM Tweets
WHERE zip BETWEEN 53000 AND 54999
```

Ways to evaluate the queries, efficiently?

1. Store the table as a heapfile, scan the file. I/O Cost?
2. Store the table as a **sorted file**, binary search the file. I/O Cost?
3. Store the table as a heapfile, build an **index**, and search using the index.
4. Store the table in an **index** file. The entire tuple is stored in the index!

Index

- Two main types of indices
  - **Hash** index: good for equality search (e.g. Q1)
  - **B-tree** index: good for both range search (e.g. Q2) and equality search (e.g. Q1)
    - Generally a hash index is faster than a B-tree index for equality search
- Hash indices aim to get O(1) I/O and CPU performance for search and insert
- B-Trees have $O(\log_F N)$ I/O and CPU cost for search, insert and delete.

What is in the index

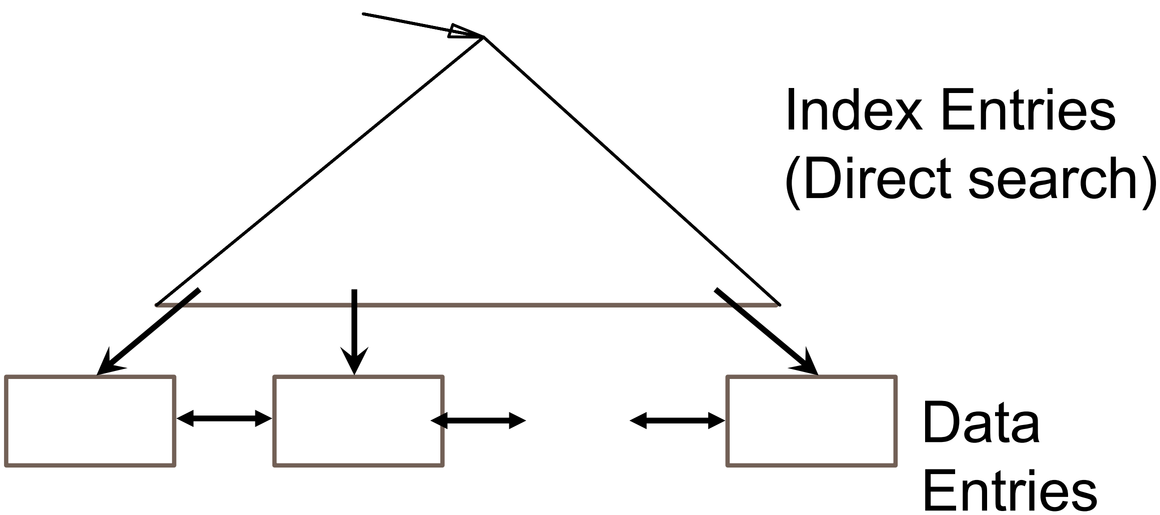
- Two things: **index key** and **some value**
  - Insert(indexKey, value)
  - Search (indexKey) -> value (s)
- What is the index key for Q1 and Q2?
- Consider Q3:

```
SELECT * FROM Tweets
WHERE uid = 145 AND
 zip BETWEEN 53000 AND 54999
```

- Value:
  - Record id
  - List of record id
  - The entire tuple!

(Ubiquitous) B+ Tree

- Height-balanced (dynamic) tree structure
- Insert/delete at $\log_F N$ cost (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root).
  Each node contains **d** <= <u>*m*</u> <= 2**d** entries.
  The parameter **d** is called the **order** of the tree.
- Supports equality and range-searches efficiently.

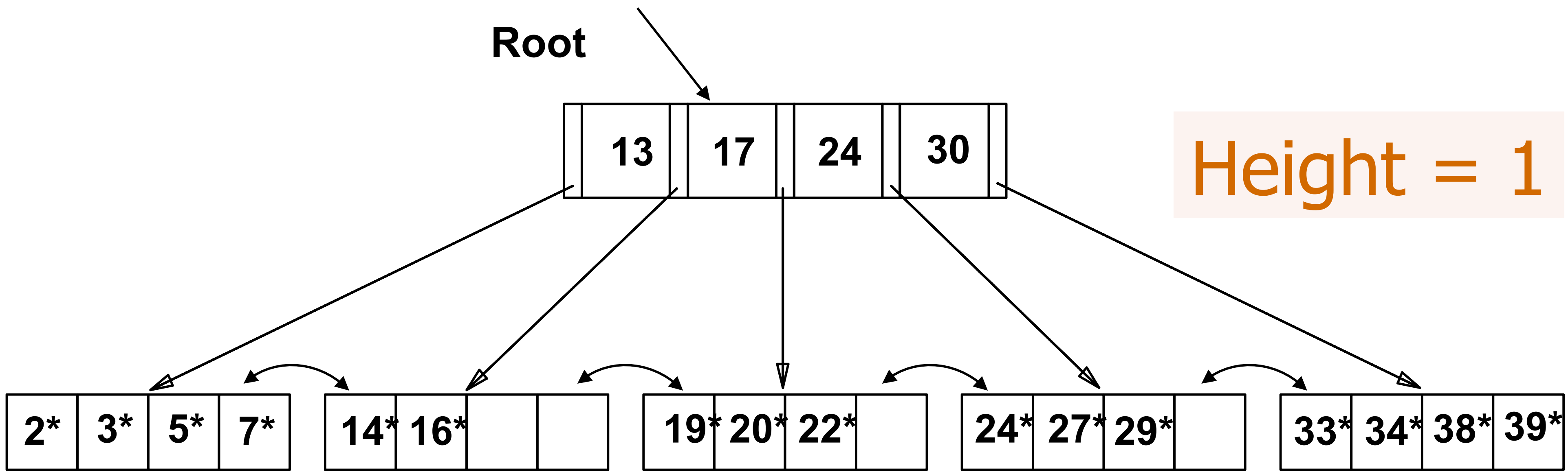


<u>Index Entries</u>
Entries in the index
(i.e. non-leaf) pages:
(search key value, pageid)

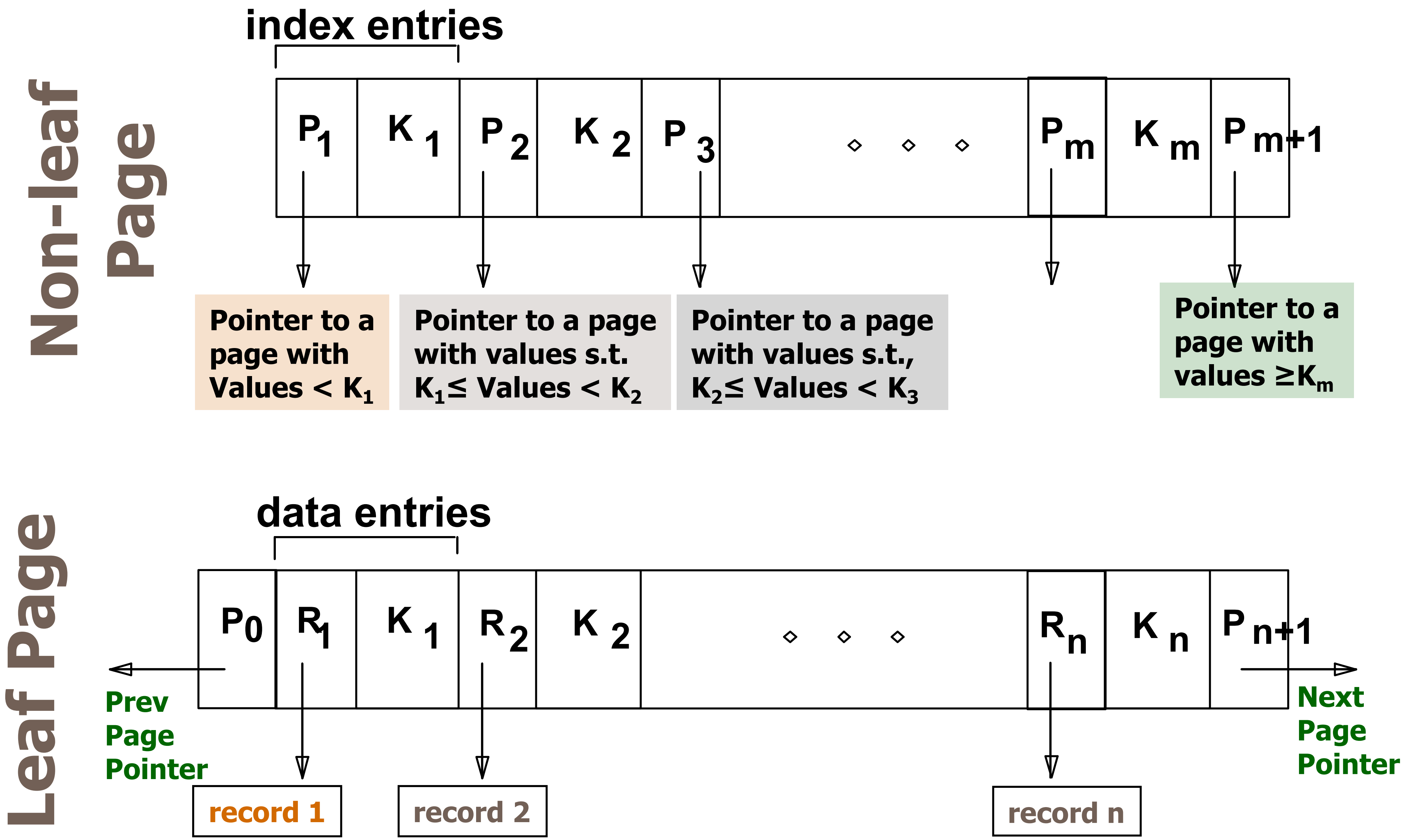
<u>Data Entries</u>
Entries in the leaf pages:
(search key value, recordid)

Example B+ Tree

- Search: Starting from root, examine index entries in non-leaf nodes, and traverse down the tree until a leaf node is reached
  - Non-leaf nodes can be searched using a binary or a linear search.
- Search for 5*, 15*, all data entries >=24*



B+-tree Page Format



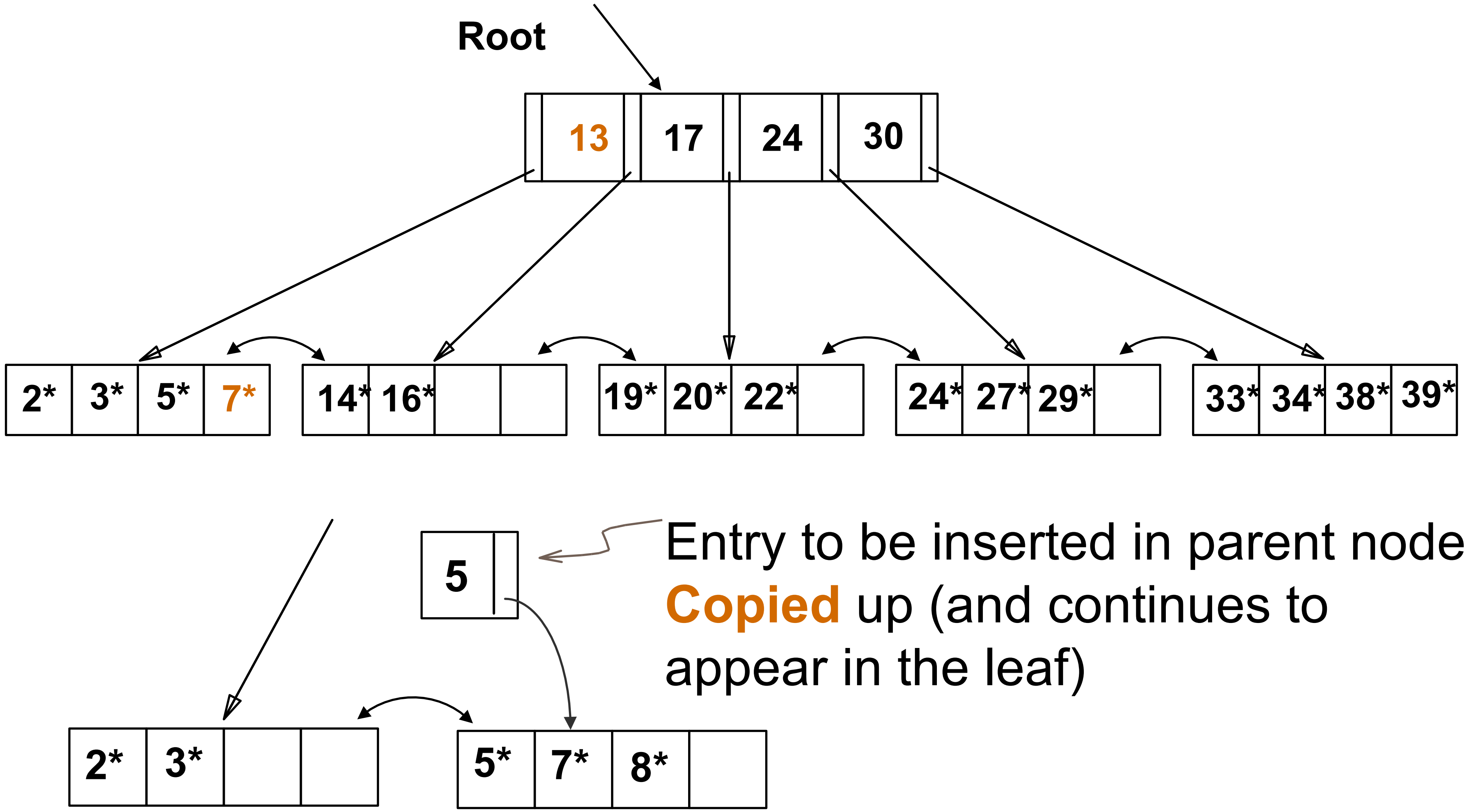
B+ Trees in Practice

- 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

B+-Tree: Inserting a Data Entry

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

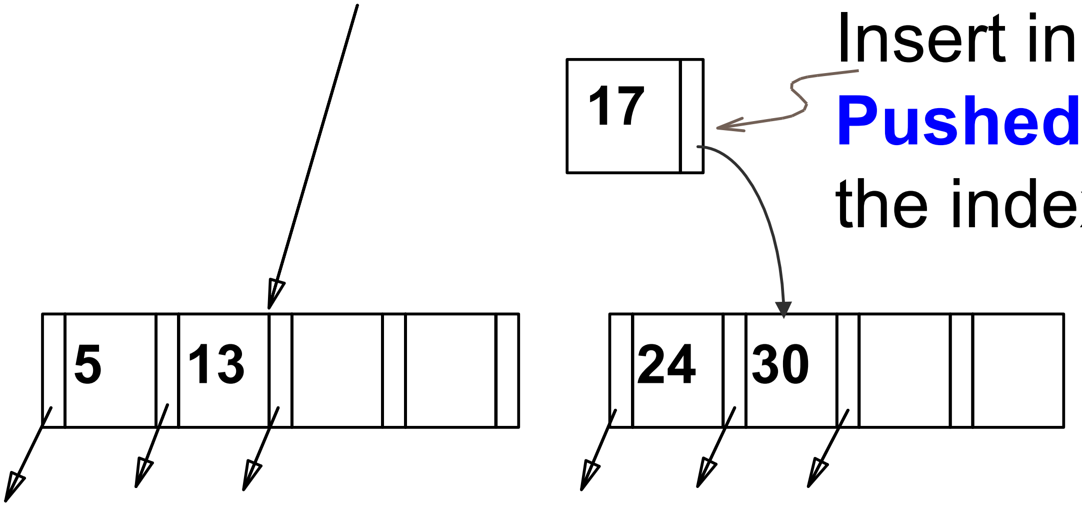
Inserting 8* into B+ Tree



Entry to be inserted in parent node

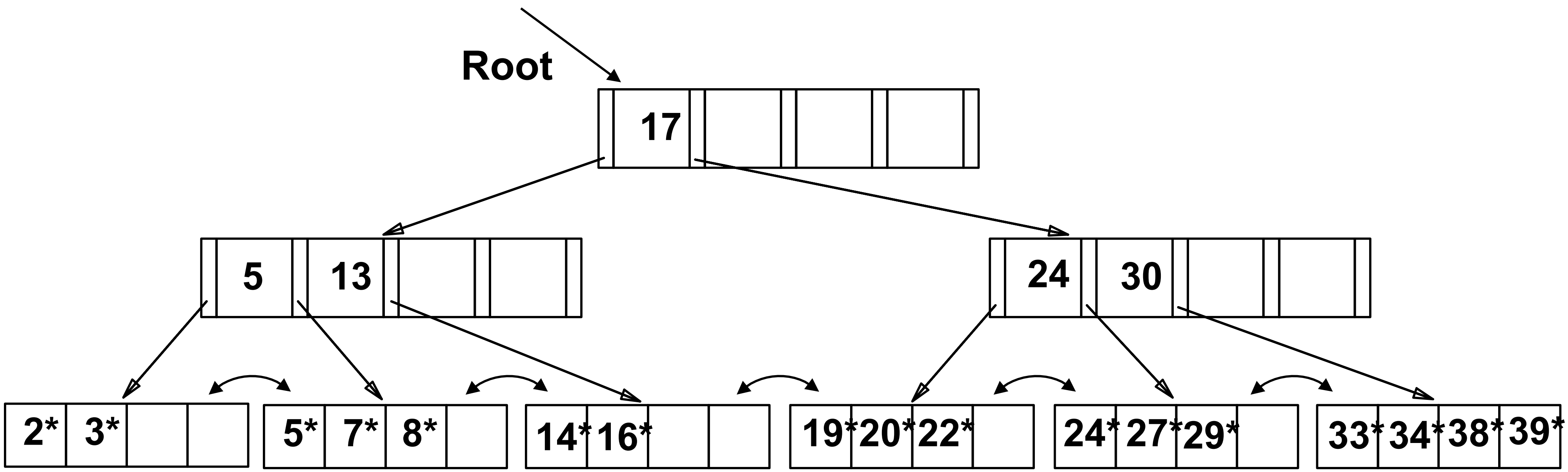
Copied up (and continues to appear in the leaf)

Inserting 8* into B+ Tree



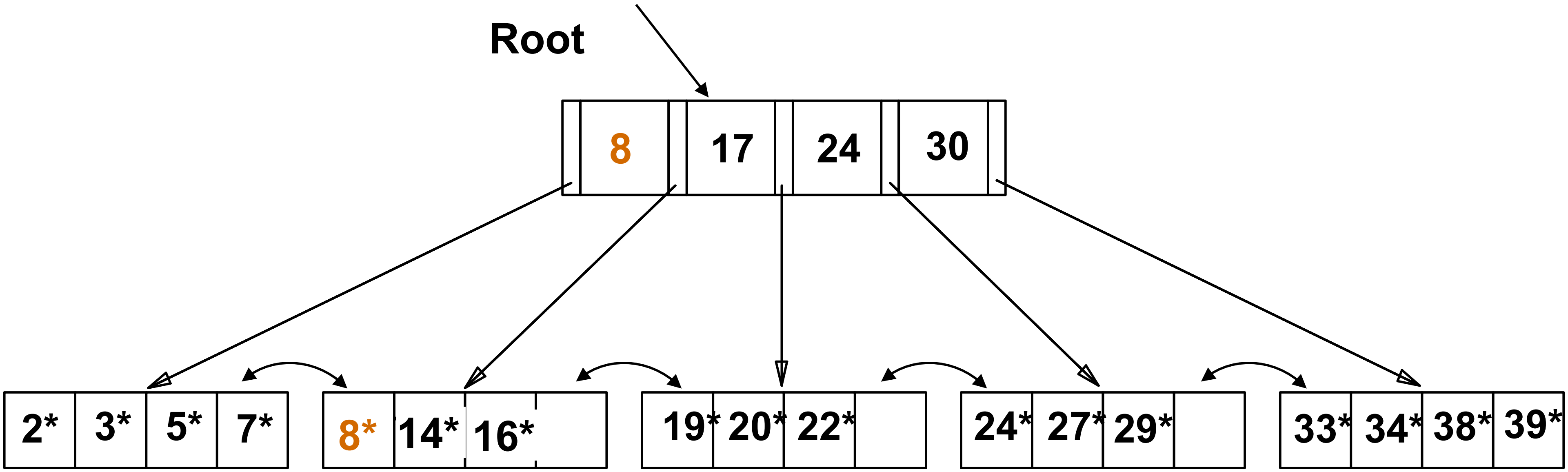
Insert in parent node.
Pushed up (and only appears once in the index)

Inserting 8* into B+ Tree



- Root was split: height increases by 1
- Could avoid split by re-distributing entries with a sibling
  - Sibling: immediately to left or right, and same parent

Inserting 8* into B+ Tree

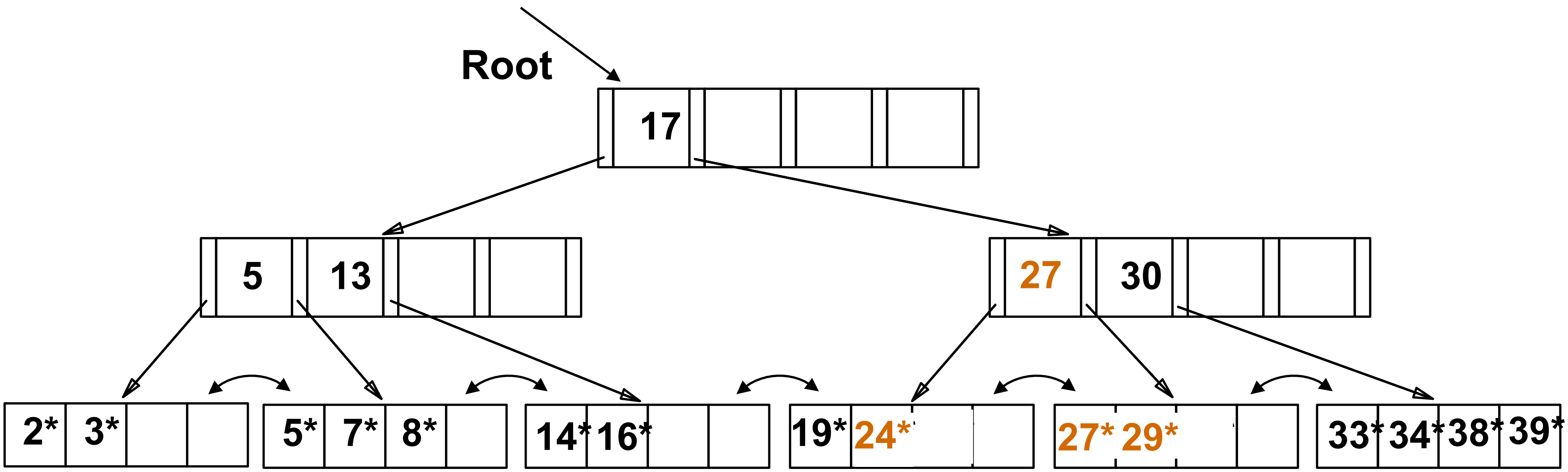


- Re-distributing entries with a **sibling**
  - Improves page occupancy
  - Usually not used for non-leaf node splits. Why?
    - Increases I/O, especially if we check both siblings
    - Better if split propagates up the tree (rare)
    - Use only for leaf level entries as we have to set pointers

B+-Tree: Deleting a Data Entry

- 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.

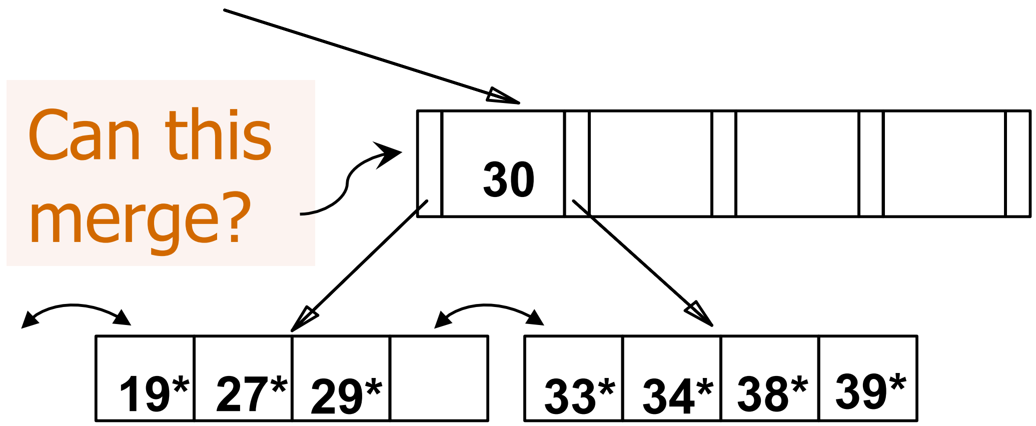
Deleting 22* and 20*

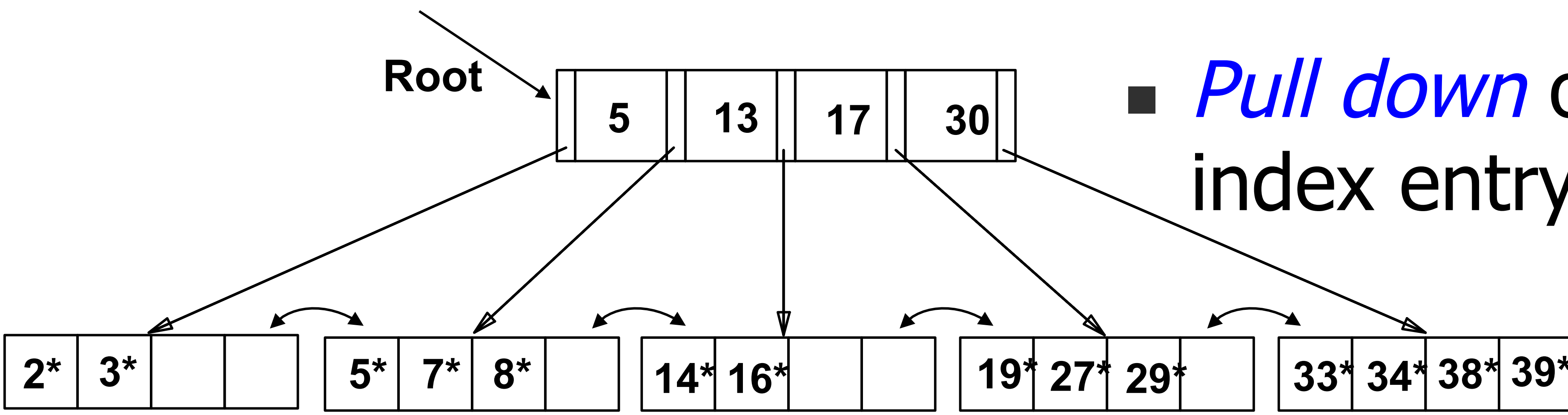


- Deleting 22* is easy.
- Deleting 20* is done with re-distribution. Notice how the middle key is **copied up**.

... And Then Deleting 24*

- Must merge.
- In the non-leaf node, *toss the* index entry with key value = 27

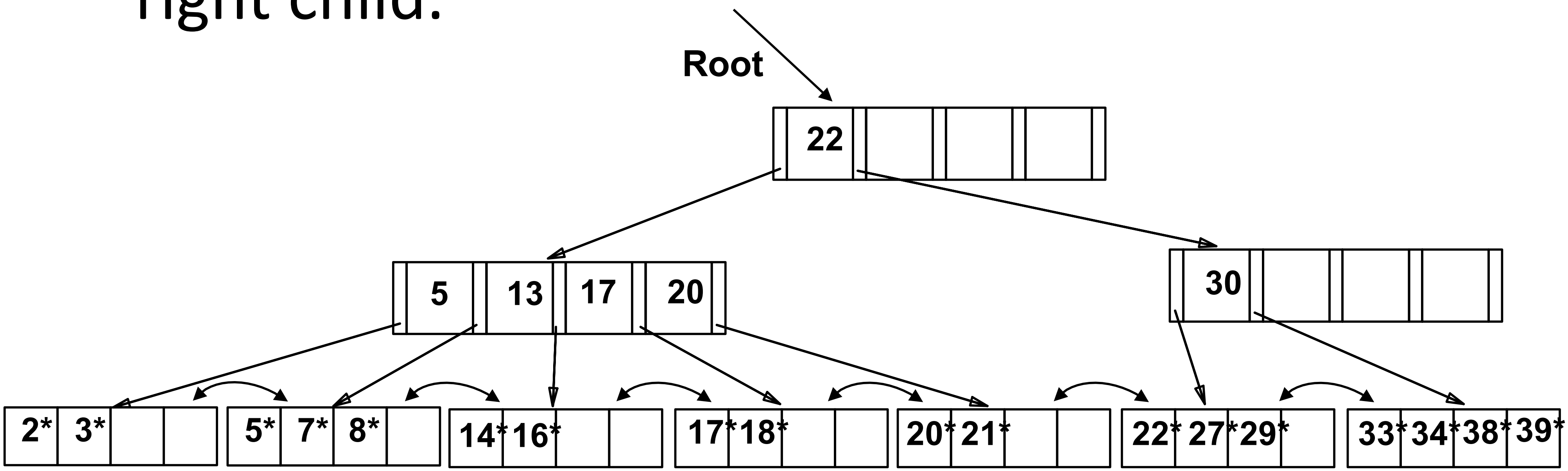




- *Pull down* of index entry

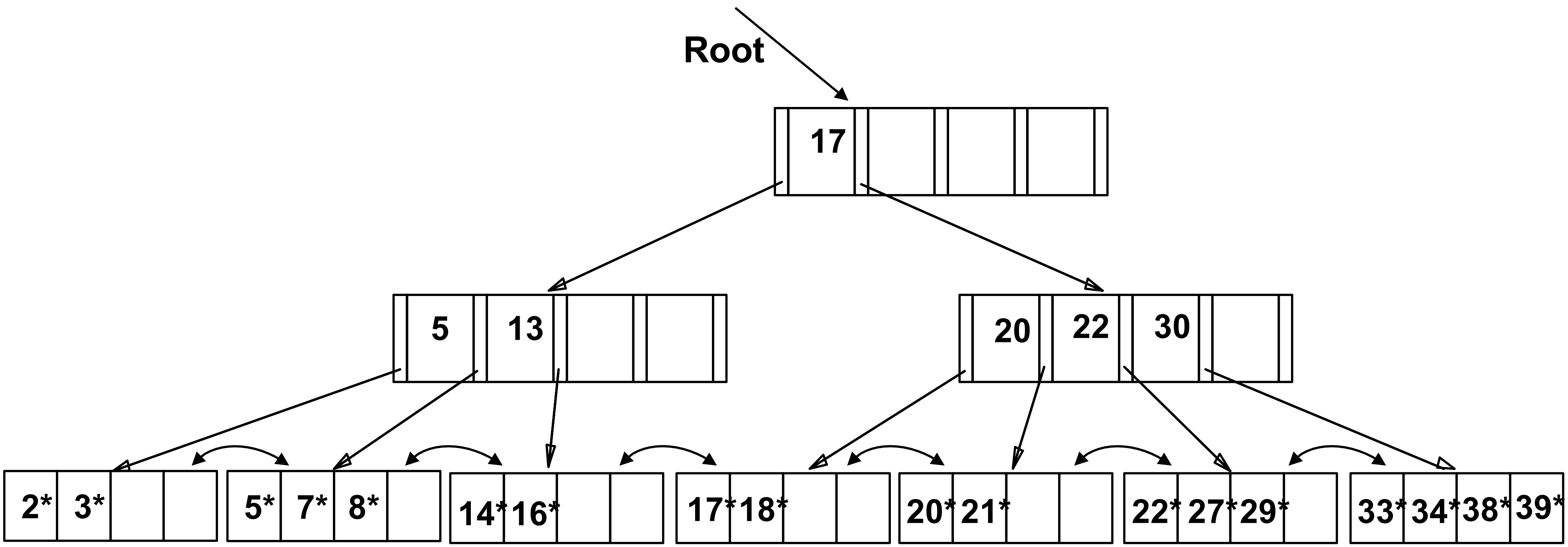
Non-leaf Re-distribution

- Tree *during deletion* of 24*.
- Can re-distribute entry from left child of root to right child.



After Re-distribution

- Rotate through the parent node
- It suffices to re-distribute index entry with key 20; For illustration 17 also re-distributed



B+-Tree Deletion

- Try redistribution with **all** siblings first, then merge. Why?
  - Good chance that redistribution is possible (large fanout!)
  - Only need to propagate changes to parent node
  - Files typically grow not shrink!

Duplicates

- Duplicate Keys: many data entries with the same key value
- Solution 1:
  - All entries with a given key value reside on a single page
  - Use overflow pages!
- Solution 2:
  - Allow duplicate key values in data entries
  - Modify search
  - Use RID to get a **unique** (composite) key!
- Use list of rids instead of a single rid in the leaf level
  - Single data entry could still span multiple pages

A Note on Order

- *Order* (**d**) concept replaced by physical space criterion in practice (*at least half-full*).
  - Index (i.e. non-leaf) pages can typically hold many more entries than leaf pages.
    - Leaf pages could have actual data records
  - 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 (e.g. list of rids).

ISAM - Indexed Sequential Access Method

- A *static* B+-tree
  - When the index is created, build a B+-tree on the relation
  - Updates and deletes don't change the non-leaf pages.
  - Use overflow pages. Leaf pages could be empty!
- Search Cost: $\text{Log}_F N$ + # overflow pages