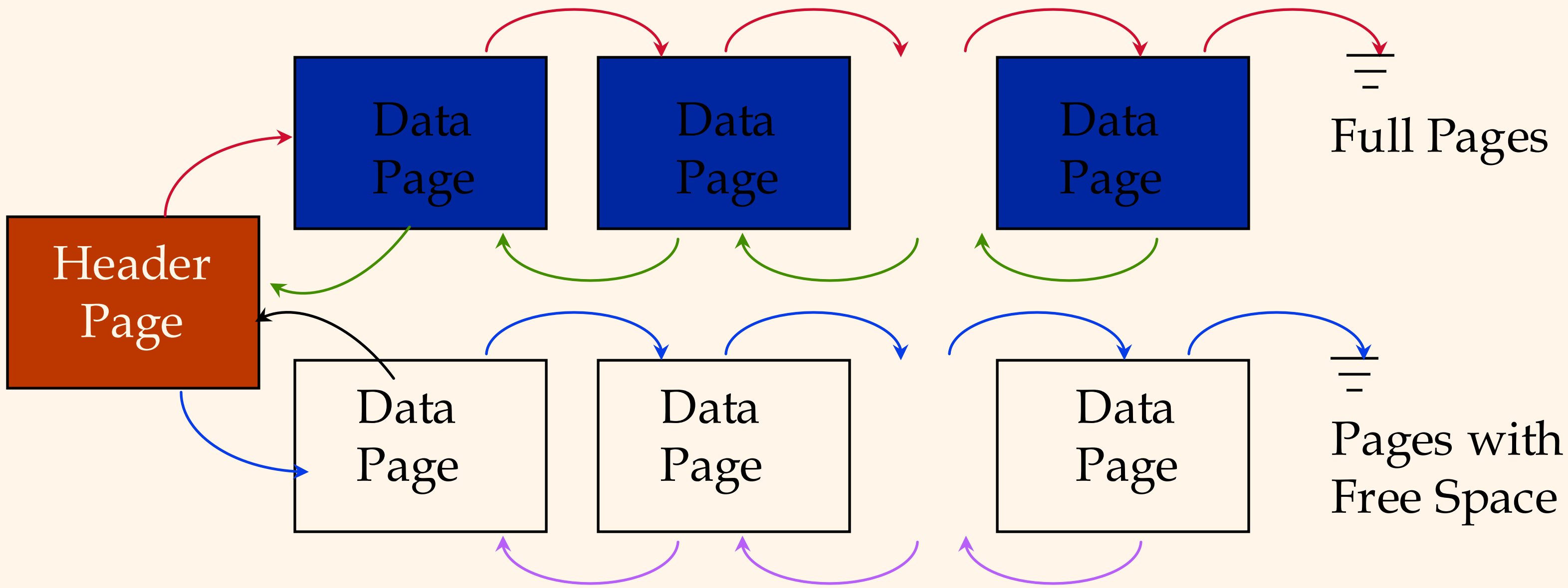
File Organization and Indexing

Lecture 7

Motivation

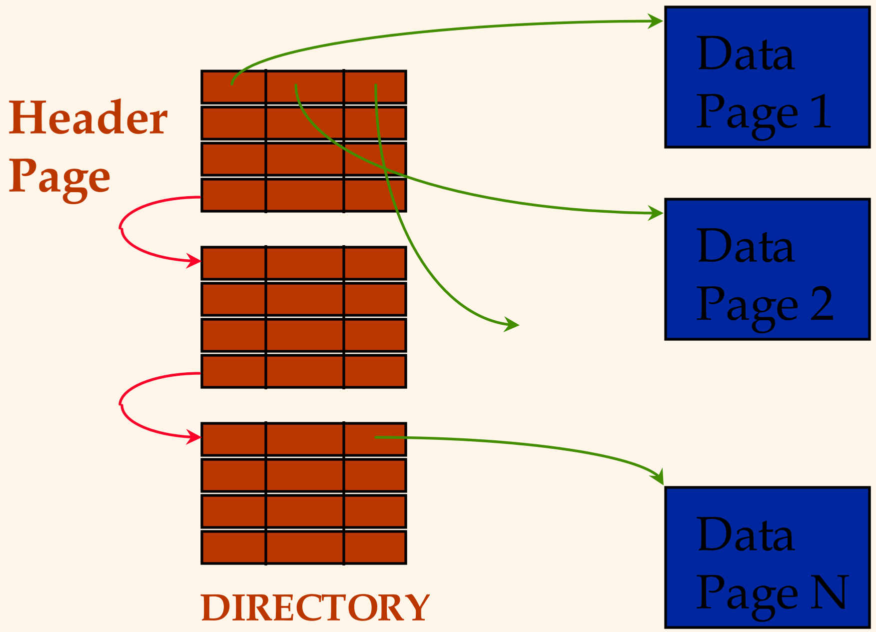
- Frequent operations
  - scan (go over all tuples)
  - sort, equality search, range search
  - insert tuple, delete tuple (modify tuple?)
- Need to speed up these operations
  - using certain file organizations and indexes
- File organizations
  - heap
  - sorted
  - hash
  - B+

Heap File Implemented as a List



- The header page id and Heap file name must be stored someplace.
- Each page contains 2 `pointers' plus data.

Heap File Using a Page Directory



- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages; linked list implementation is just one alternative.

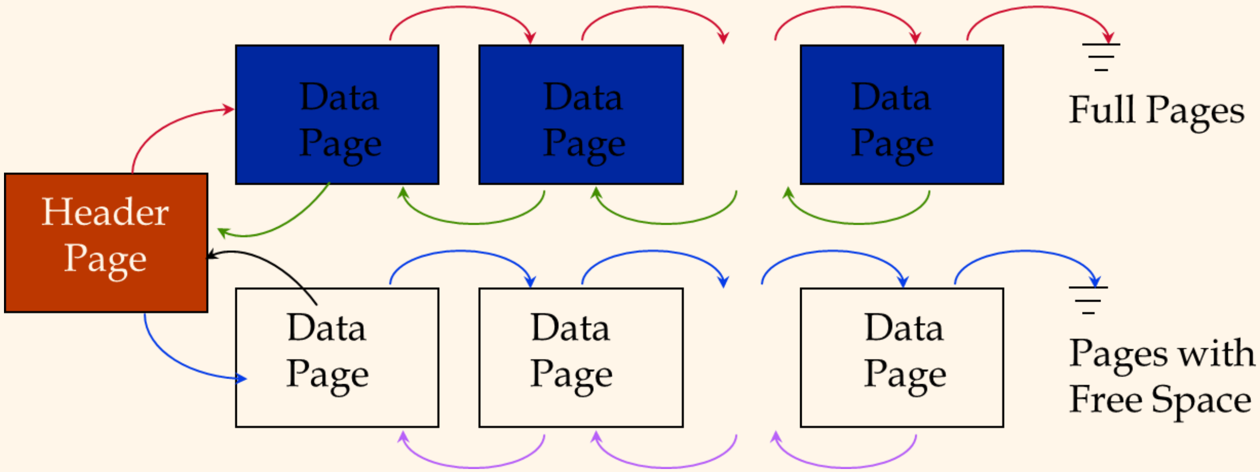
Hash Files

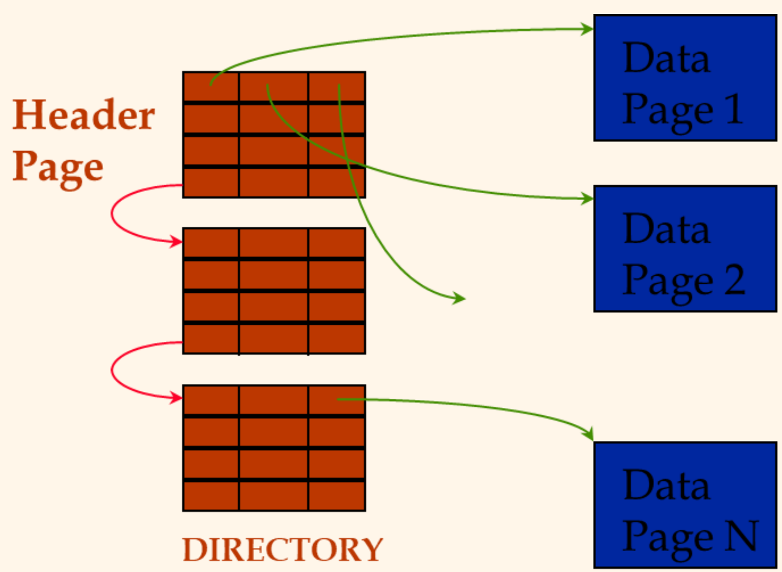
- n buckets
- Each bucket = linked list of pages
- Hash each tuple into a bucket

Sorted Files

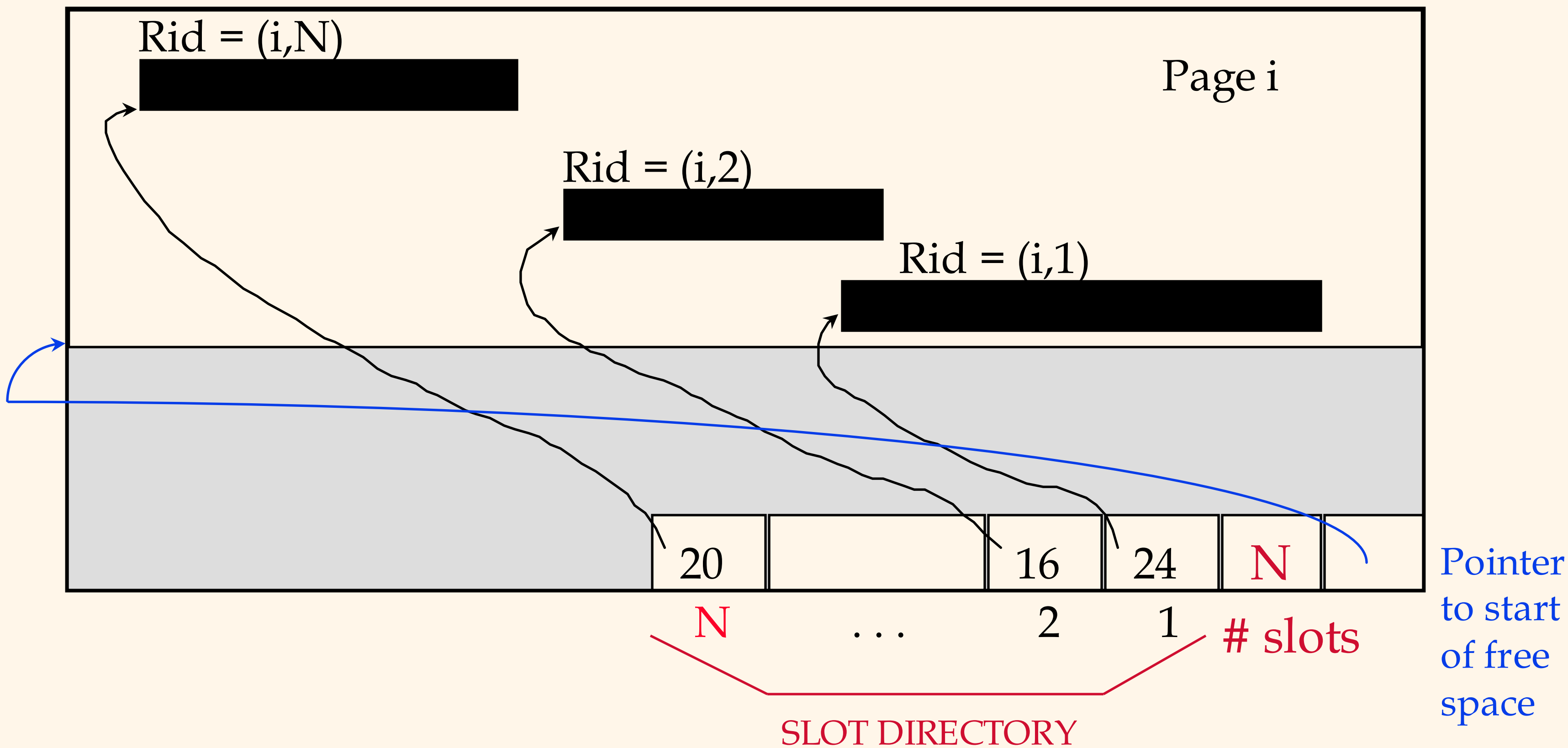
- Sort on a single attribute
- Or on a combination of attributes
- These are called "search keys" or just "keys"
  - not to be confused with keys of an entity set or table

Sorted Files



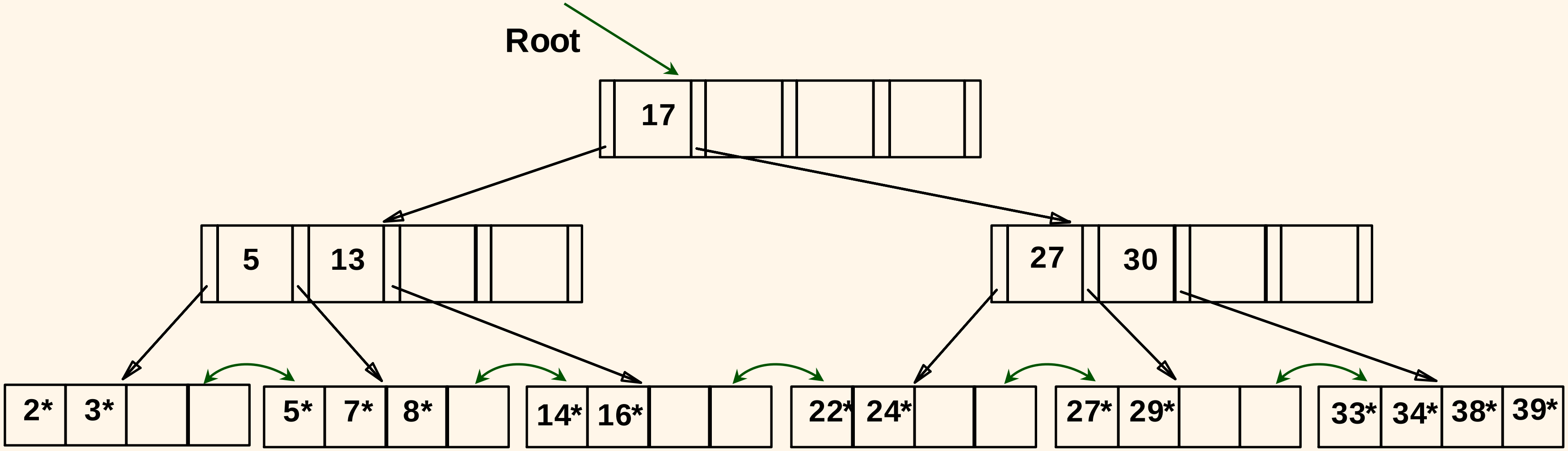


Sorted Files



B+ Tree Files

- Sorted file with a lot of pointers on top to direct search

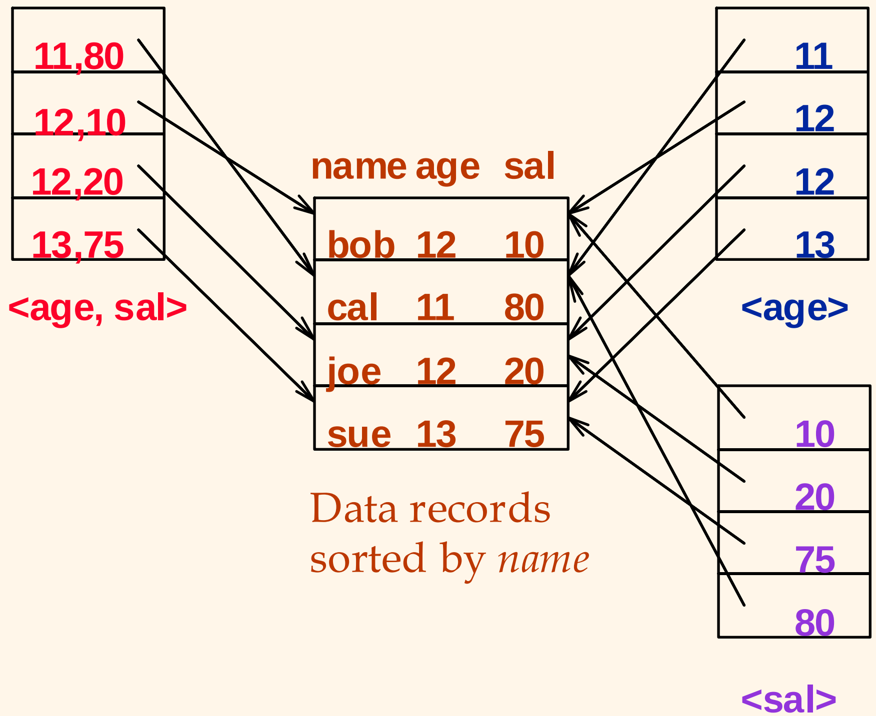


Discussion

- Frequent operations
  - scan
  - sort
  - equality / range search
  - insert / delete tuples

Motivation for Indexes

- Frequent operations
  - search by person name
  - search by age, sal, or (age, sal)



Data records sorted by *name*

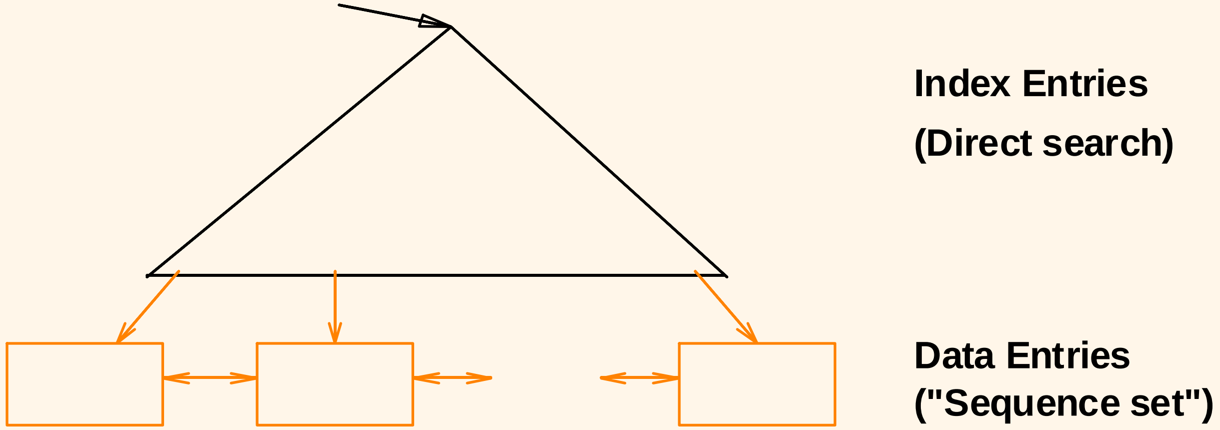
Types of Indexes

- Clustered vs unclustered indexes
- Primary vs secondary indexes

B+ Tree Index

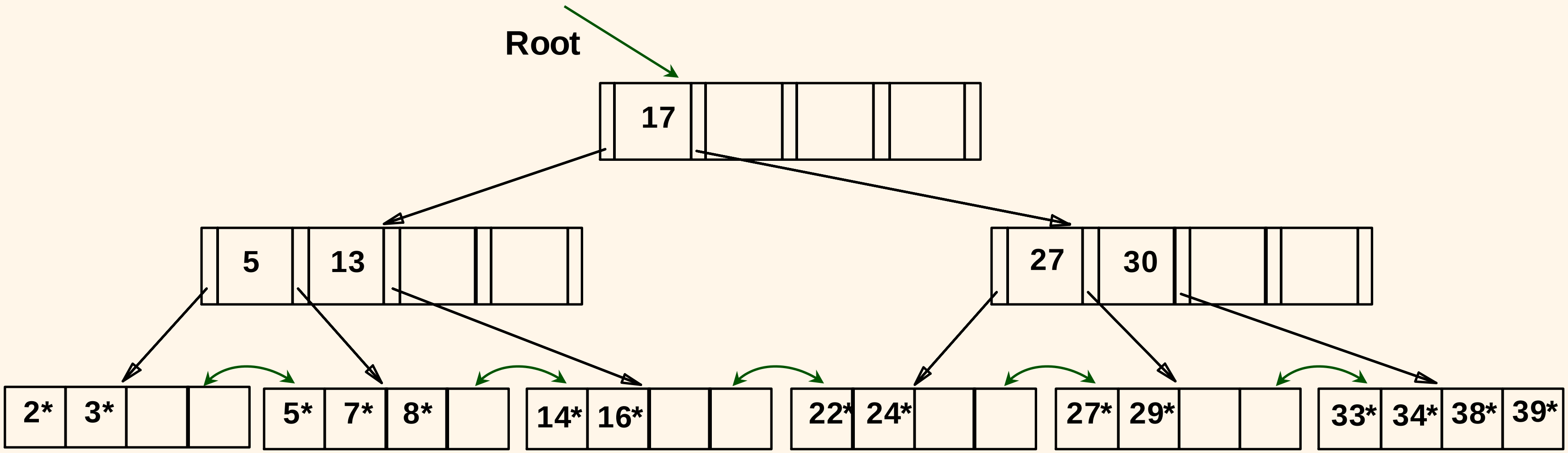
B+ Tree: Most Widely Used Index

- 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** <= <u>*m*</u> <= 2**d** entries. The parameter **d** is called the *order* of the tree.
- Supports equality and range-searches efficiently.



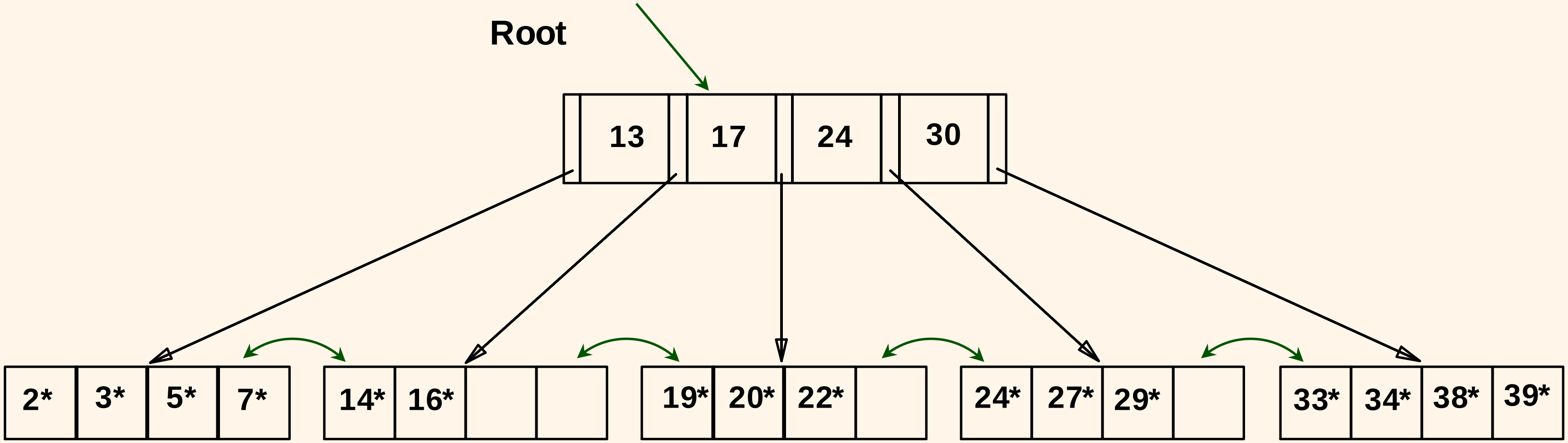
Example B+ Tree

- d = 2, each entry is a number



Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf
- Search for 5*, 15*, all data entries >= 24* ...



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

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

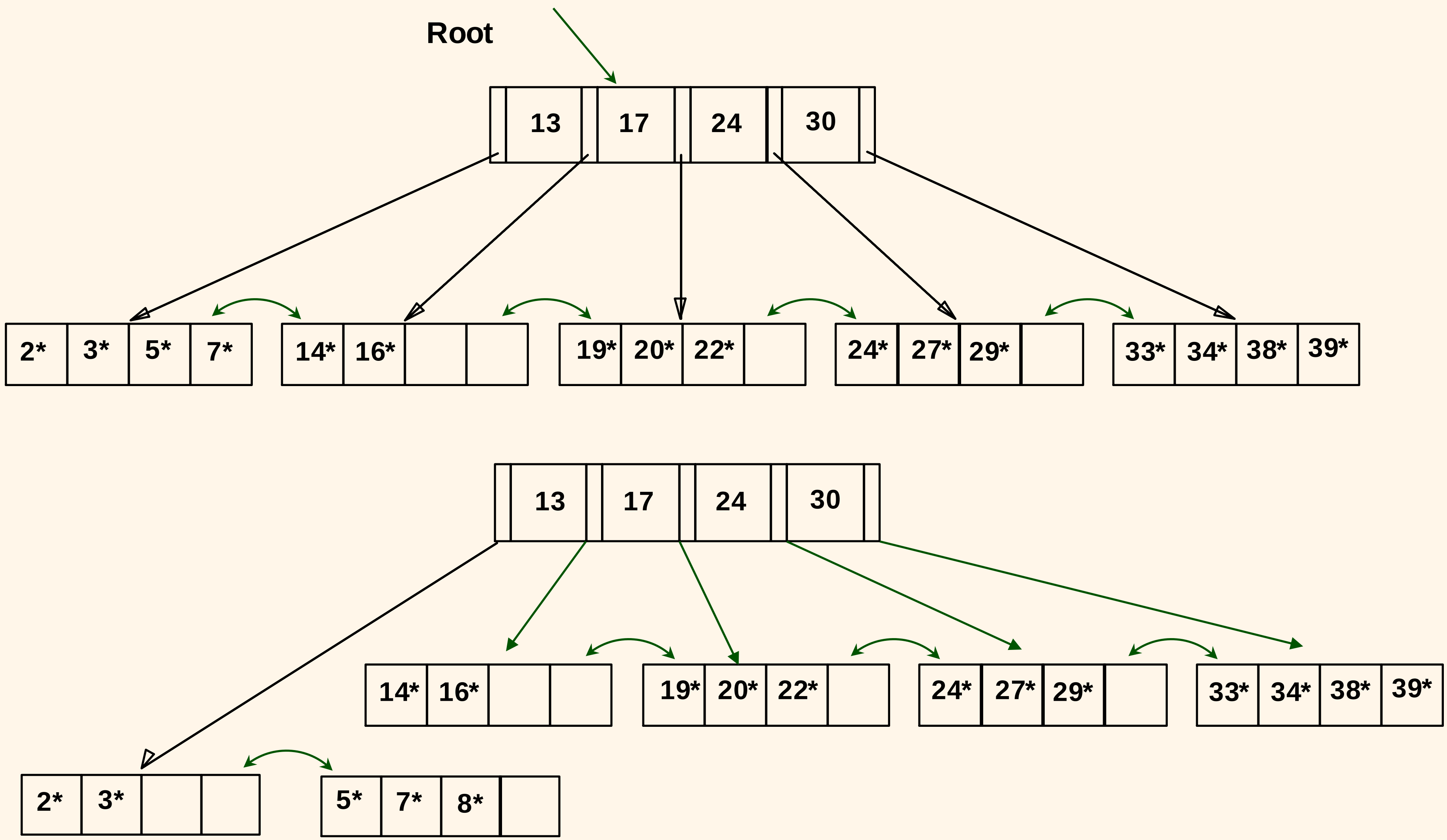
Inserting a Data Entry into a B+ Tree

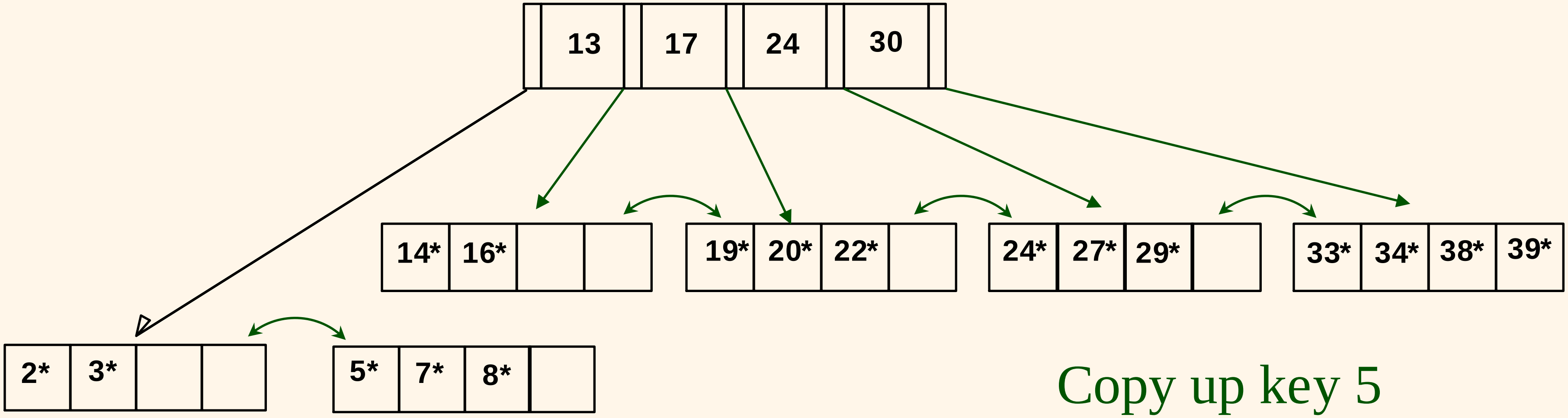
- Find correct leaf *L*.
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To 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.*

Important Tip to Get This Done Right

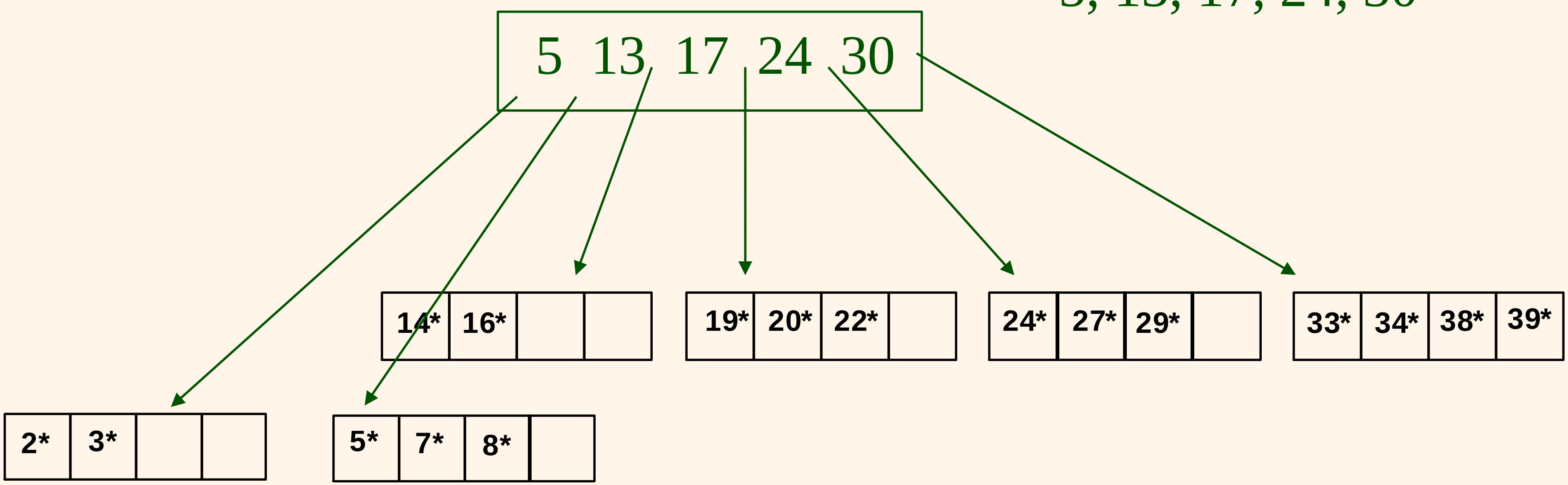
- on the exam
- focus on creating the pages, get them right
- don't worry about the pointers
- once you have the pages right, you can easily create the pointers

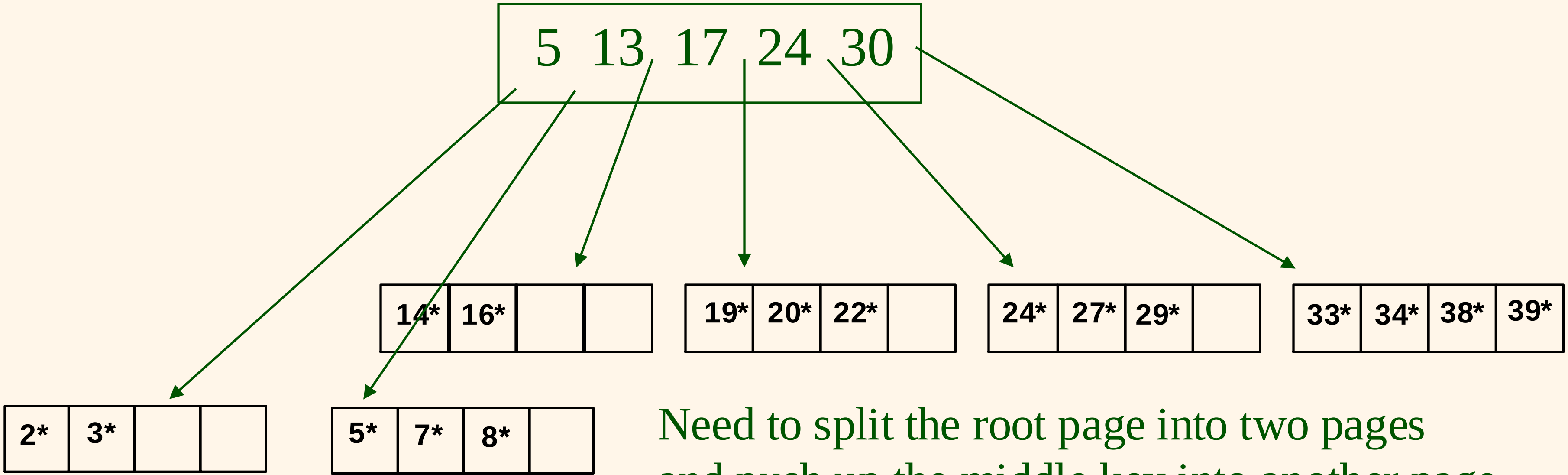
*Example: Insert 8**



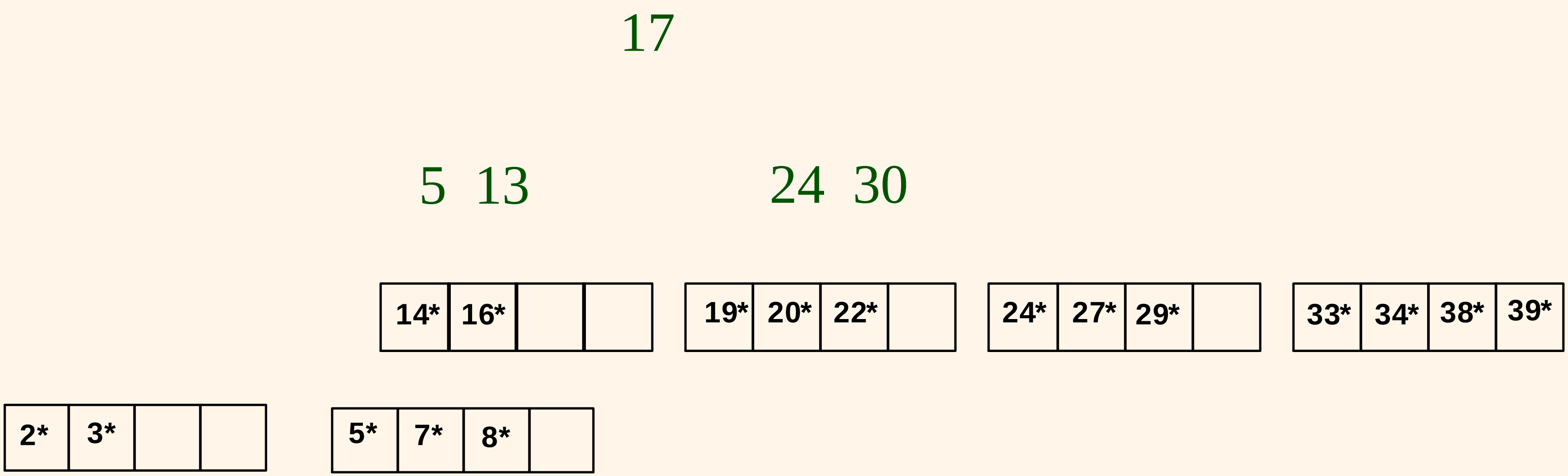


Copy up key 5
So page will have
5, 13, 17, 24, 30

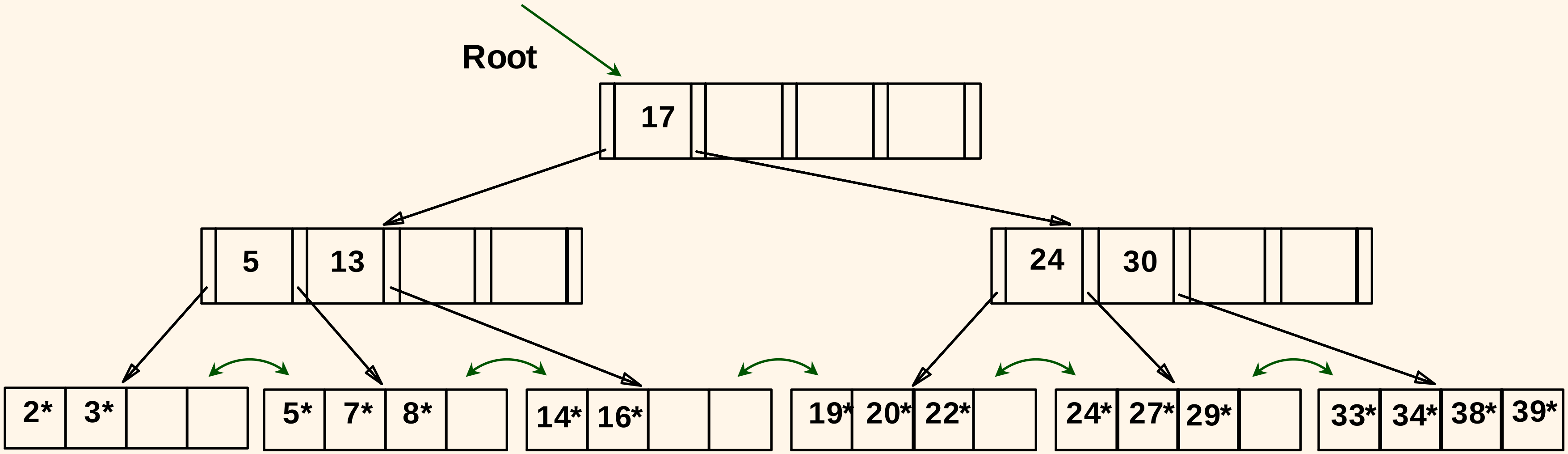




Need to split the root page into two pages
and push up the middle key into another page



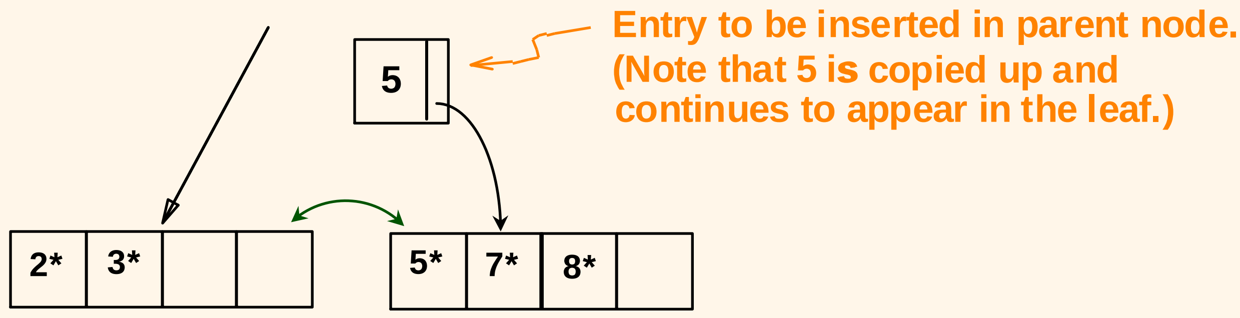
*Example B+ Tree After Inserting 8**

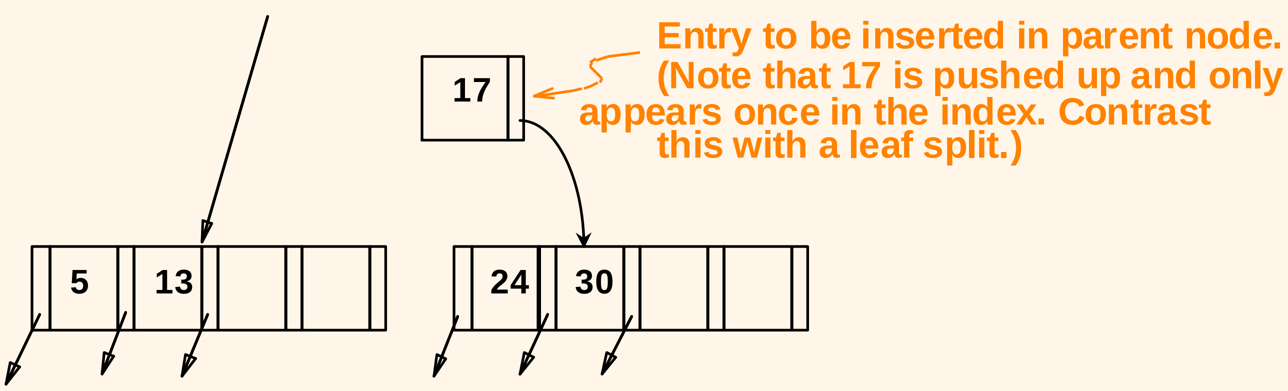


❖ Notice that root was split, leading to increase in height.

Inserting 8 into Example B+ Tree*

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.

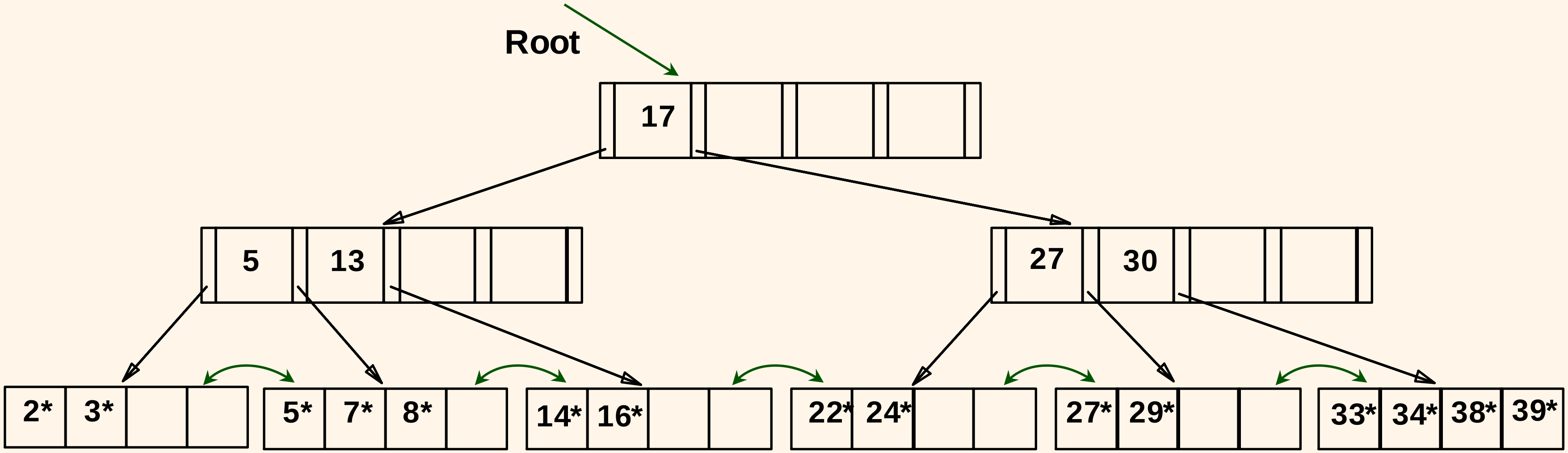




Deleting a Data Entry from a B+ Tree

- 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 Tree After (Inserting 8, Then) Deleting 19* and 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.
- Observe *`toss'* of index entry (on right), and *`pull down'* of index entry (below).