CI-5313
Arquitectura y Administración de Bases de Datos

Clase 3 – Indices (I)

Prof. Edna Ruckhaus
Láminas Prof. José Tomás Cadenas

Cost of Operations

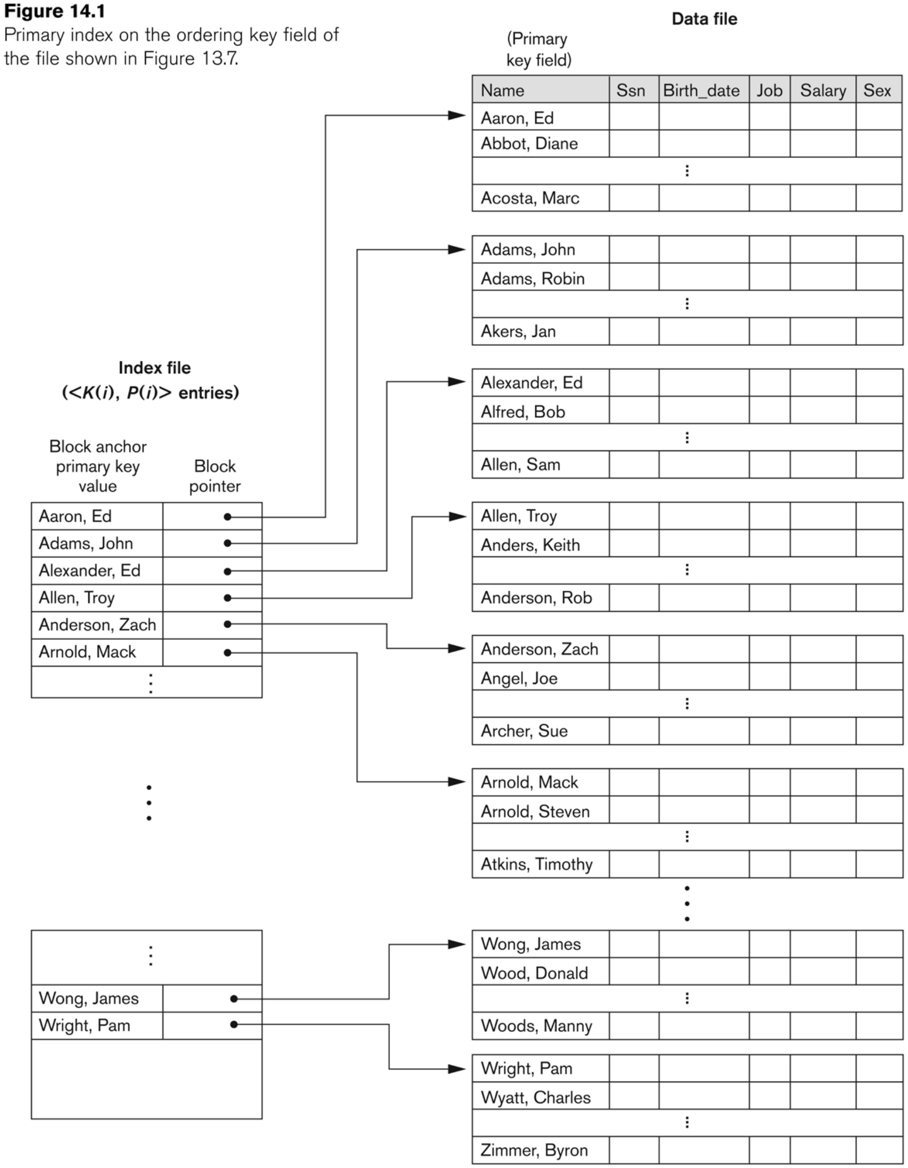
	Heap File	Sorted File
Scan all recs	**BD**	**BD**
Equality Search	**0.5 BD**	**D $\log_2$B**
Range Search	**BD**	**D ($\log_2$B + # of pages with matches)**
Insert	**2D**	**Search + BD**
Delete	**Search + D**	**Search + BD**

Index

- Data structure that organizes data records on disk.
- An index on a file speeds up selections on the search key fields for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - Search key is not the same as key (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of data entries, and supports efficient retrieval of all data entries k* with a given key value k.

Primary index on the ordering key field

Figure 14.1
Primary index on the ordering key field of the file shown in Figure 13.7.



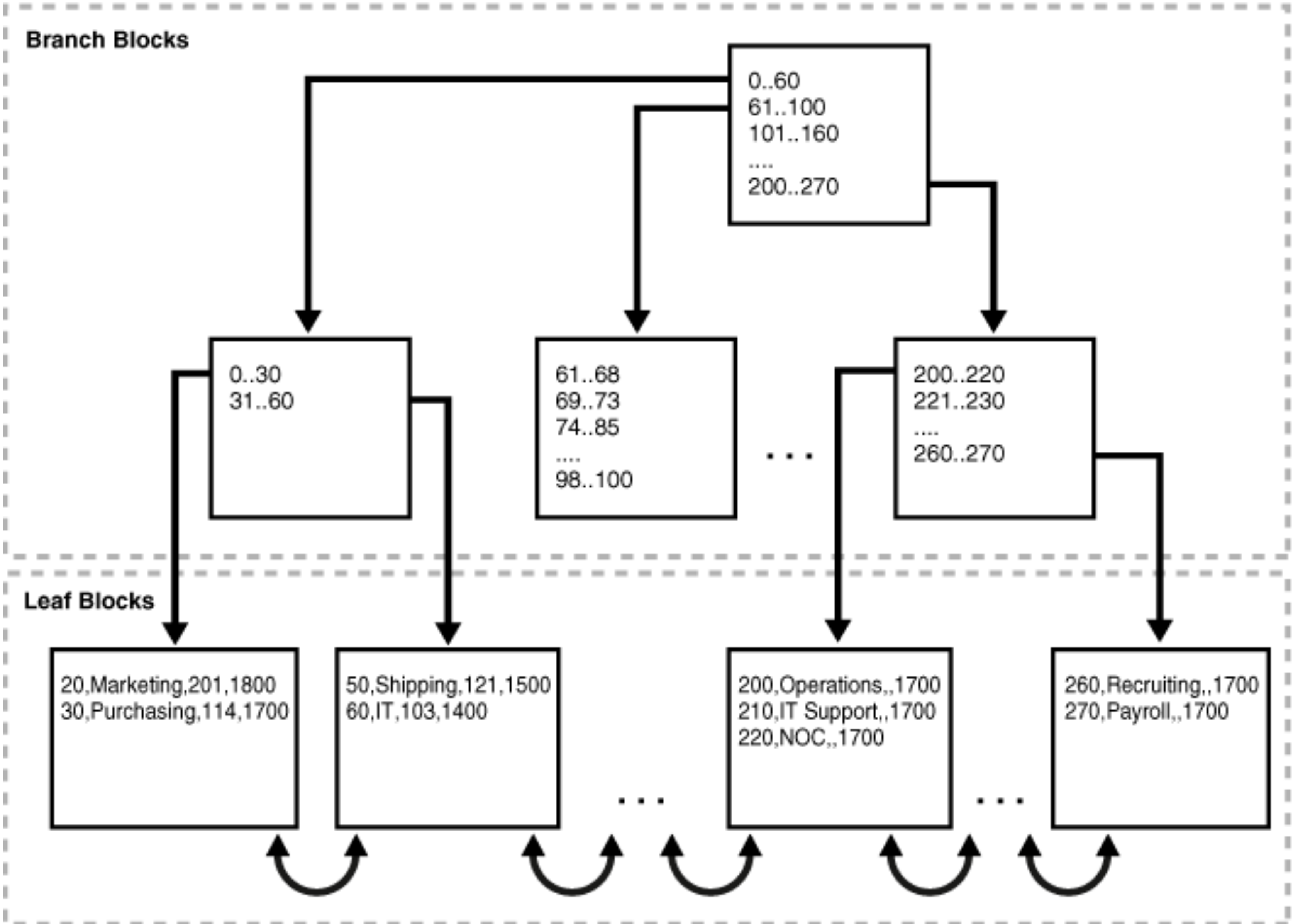
Index

- Three alternatives:
  - ① Data record with key value **k**
  - ② <**k**, rid of data record with search key value **k**>
  - ③ <**k**, list of rids of data records with search key **k**>
- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value **k**.
  - Examples of indexing techniques: B+ trees, hash-based structures
  - Typically, index contains auxiliary information that directs searches to the desired data entries

Archivo "Clustered" - Primera alternativa

- If this is used, index structure is a file organization for data records (like Heap files or sorted files).
- At most one index on a given collection of data records can use Alternative 1. (Otherwise, data records duplicated, leading to redundant storage and potential inconsistency.)
- If data records very large, # of pages containing data entries is high. Implies size of auxiliary information in the index is also large, typically.

Archivo "Clustered" - Primera alternativa



Index – Alternatives 2 and 3 – separate file for index

- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller
- A binary search on a single-level index yields a pointer to the file record
- Indexes can also be characterized as dense or sparse
    - A dense index has an index entry for every search key value (and hence every record) in the data file.
    - A sparse (or nondense) index, on the other hand, has index entries for only some of the search values

Index example

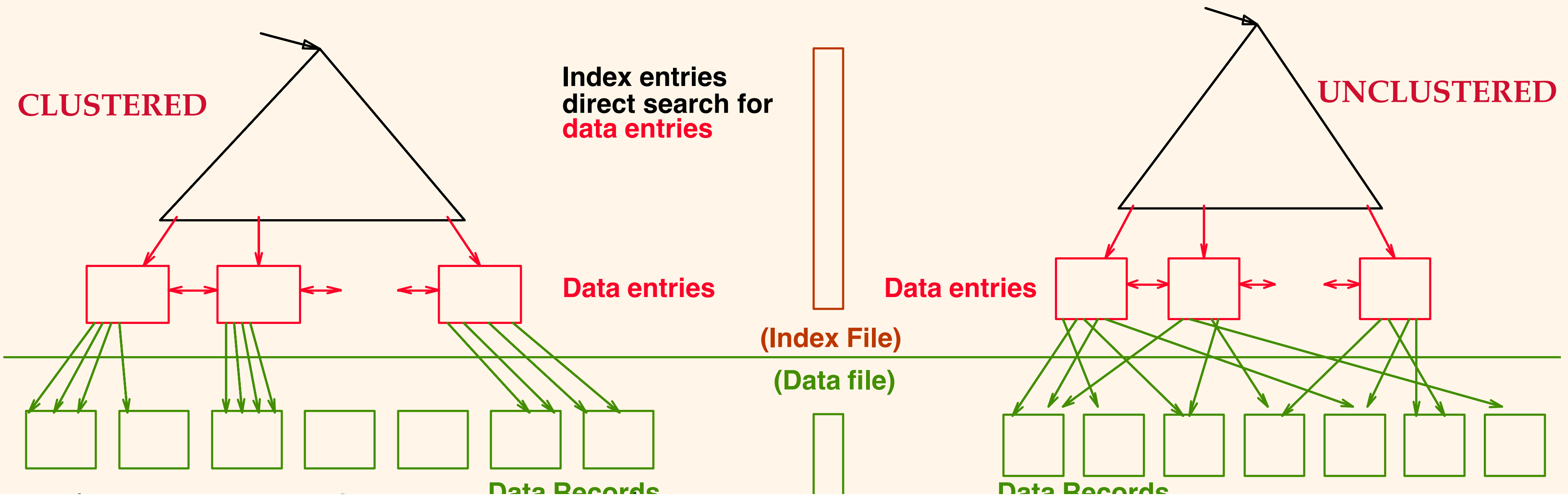
- Example: Given the following data file EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ...)
- Suppose that:
  - record size R=150 bytes   block size B=512 bytes r=30000 records
- Then, we get:
  - blocking factor Bfr= B div R= 512 div 150= 3 records/block
  - number of file blocks b= (r/Bfr)= (30000/3)= 10000 blocks
- For an index on the SSN field, assume the field size V_{SSN}=9 bytes, assume the block pointer size P_B=7 bytes. Then:
  - index entry size $R_I=(V_{SSN}+P_B)=(9+7)=16$ bytes
  - index blocking factor Bfr_I= B div R_I= 512 div 16= 32 entries/block
  - number of index blocks $b_I= (r/ Bfr_I)= (30000/32)= 938$ blocks
  - binary search needs $\log_2 b_I= \log_2 938=$ 10 block accesses
  - This is compared to an average linear search cost of:
    - (b/2)= 10000/2= 5000 block accesses
  - If the file records are ordered, the binary search cost would be:
    - $\log_2 b= \log_2 10000=$ 14 block accesses

Index taxonomies

- Dense vs. Sparse.
- Simple vs. Composite
- Single-level vs. Multi-level (trees).
- Primary vs. Secondary.
  - Primary is index on a key (unique).
  - A secondary index provides a secondary means of accessing a file for which some primary access already exists.
- Clustered vs. Unclustered.
  - If order of data records is the same as, or `close to', order of data entries, then called clustered index.
  - Alternative 1 implies clustered, but not vice-versa.
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies greatly based on whether index is clustered or not!
- Points to record vs. Points to block

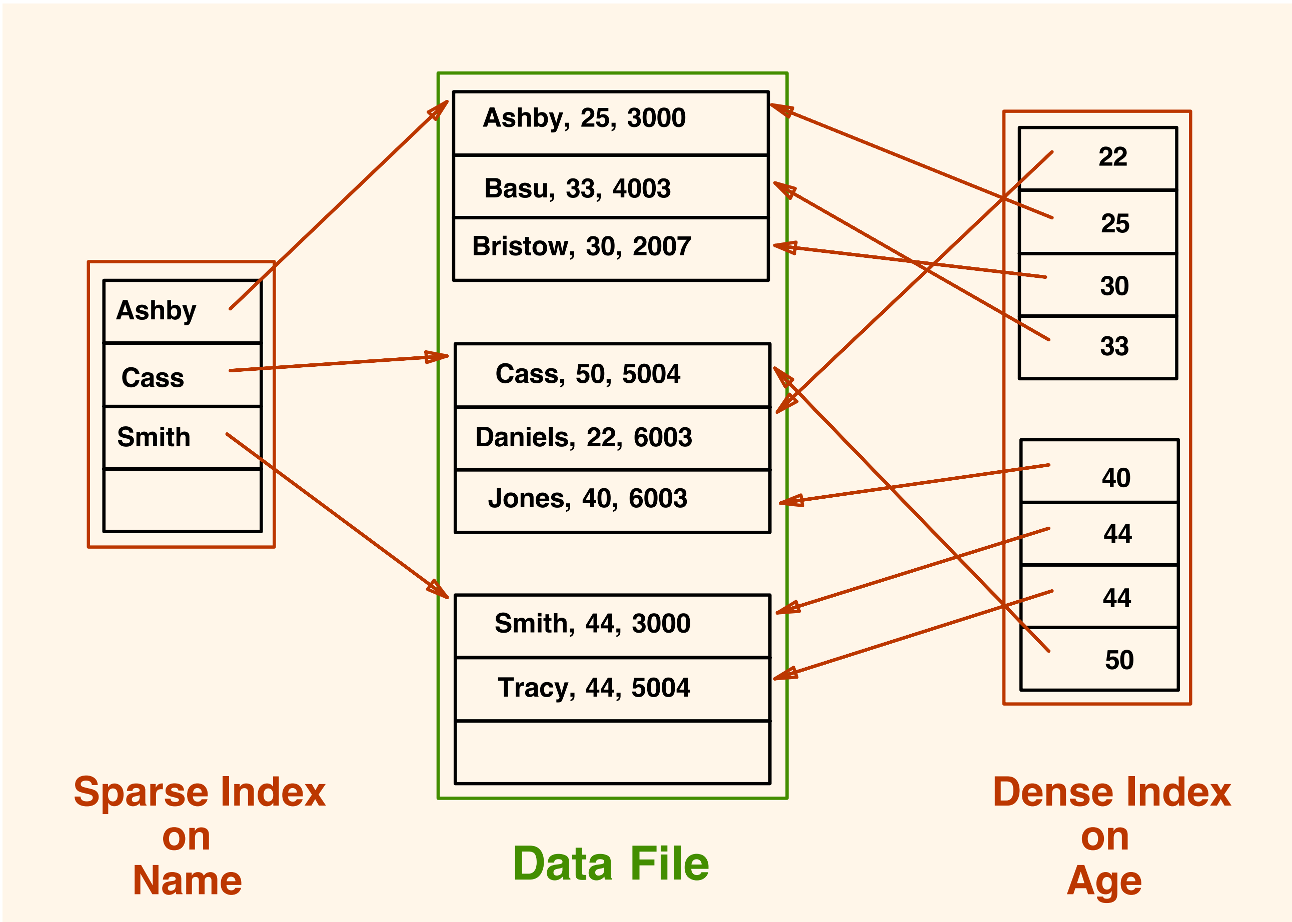
Clustered vs. Unclustered

- To build clustered index, first sort the Heap file (with some free space on each page for future inserts).
- Overflow pages may be needed for inserts. (Thus, order of data recs is `close to', but not identical to, the sort order.)



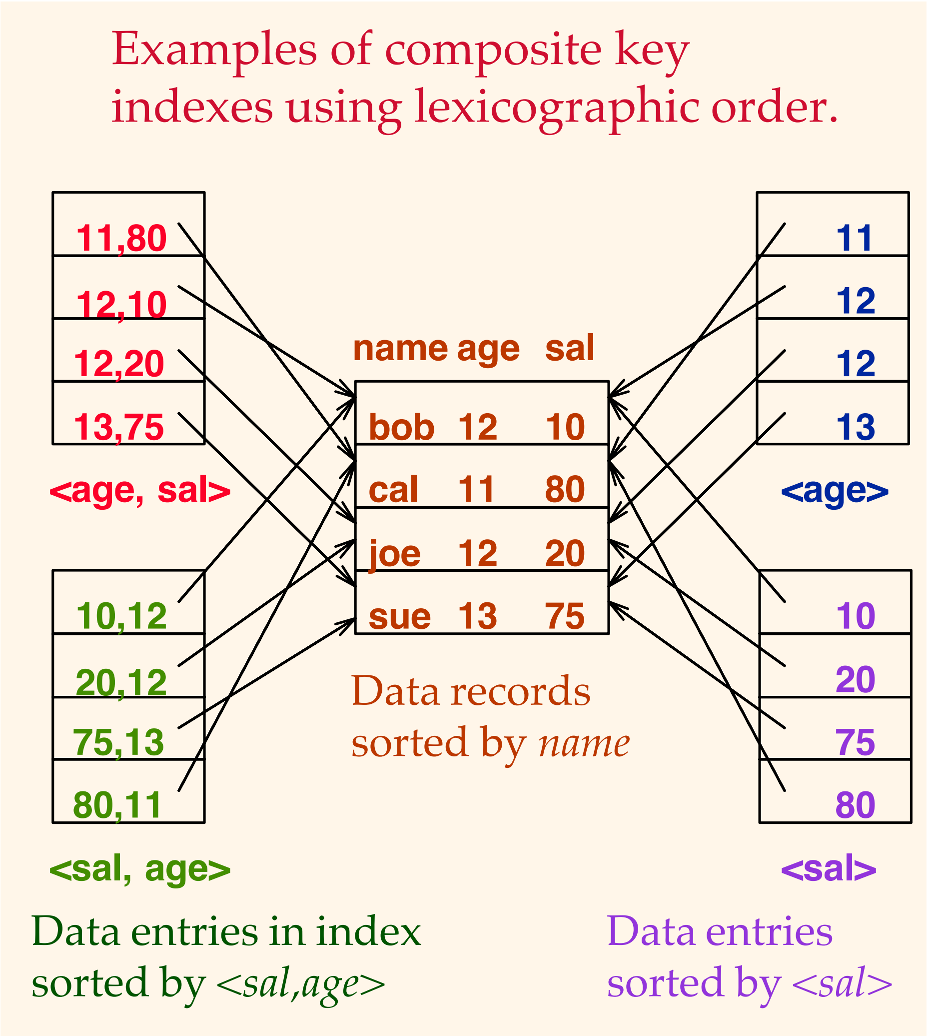
Dense vs. Sparse

- Dense vs. Sparse.



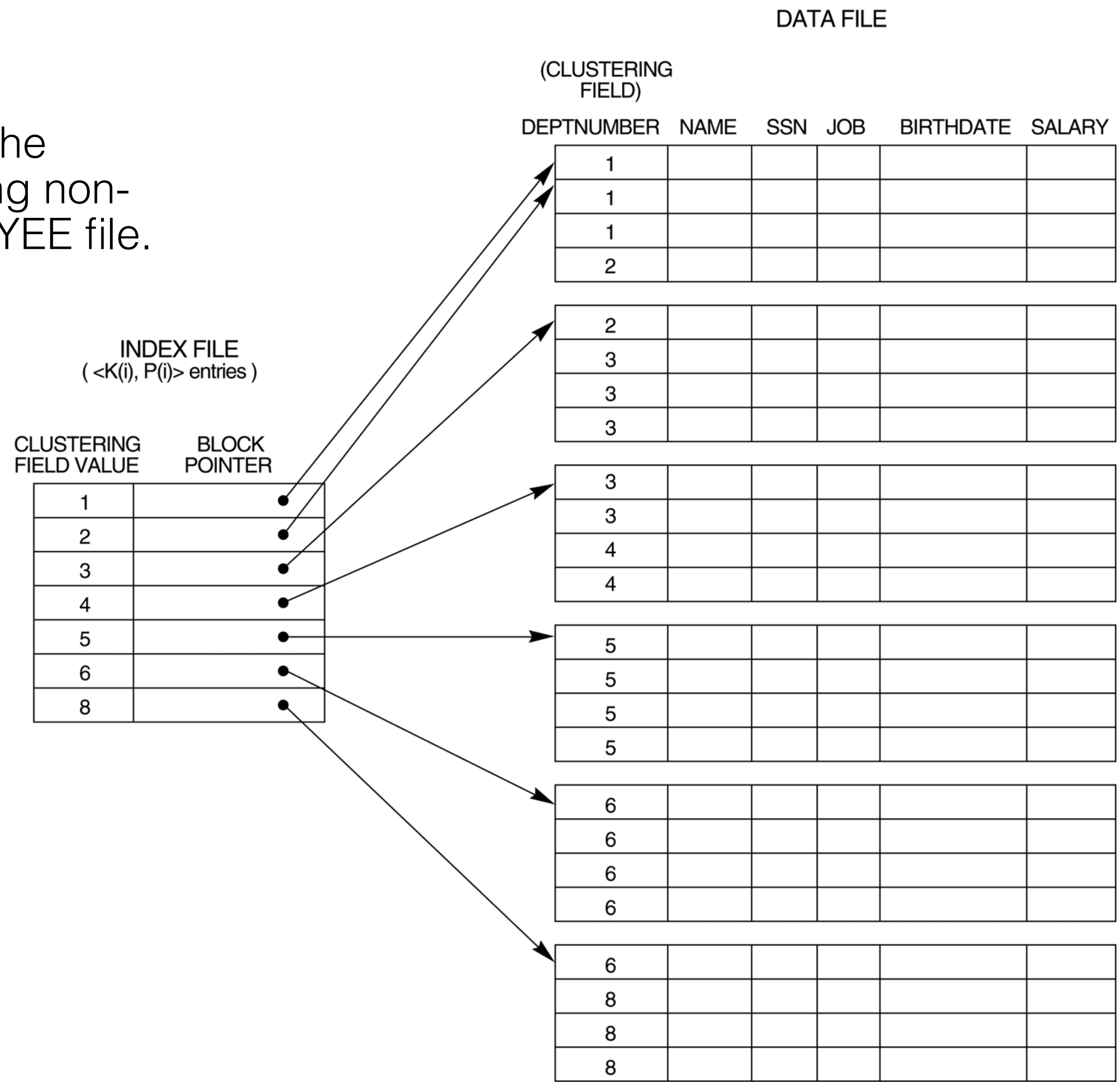
- Alternative 1 is sparse or dense?
- A sparse index is clustered or unclustered?

Example – Category in each taxonomy?



Example – Category in each taxonomy?

A clustering index on the DEPTNUMBER ordering non-key field of an EMPLOYEE file.



Another Clustering Index Example

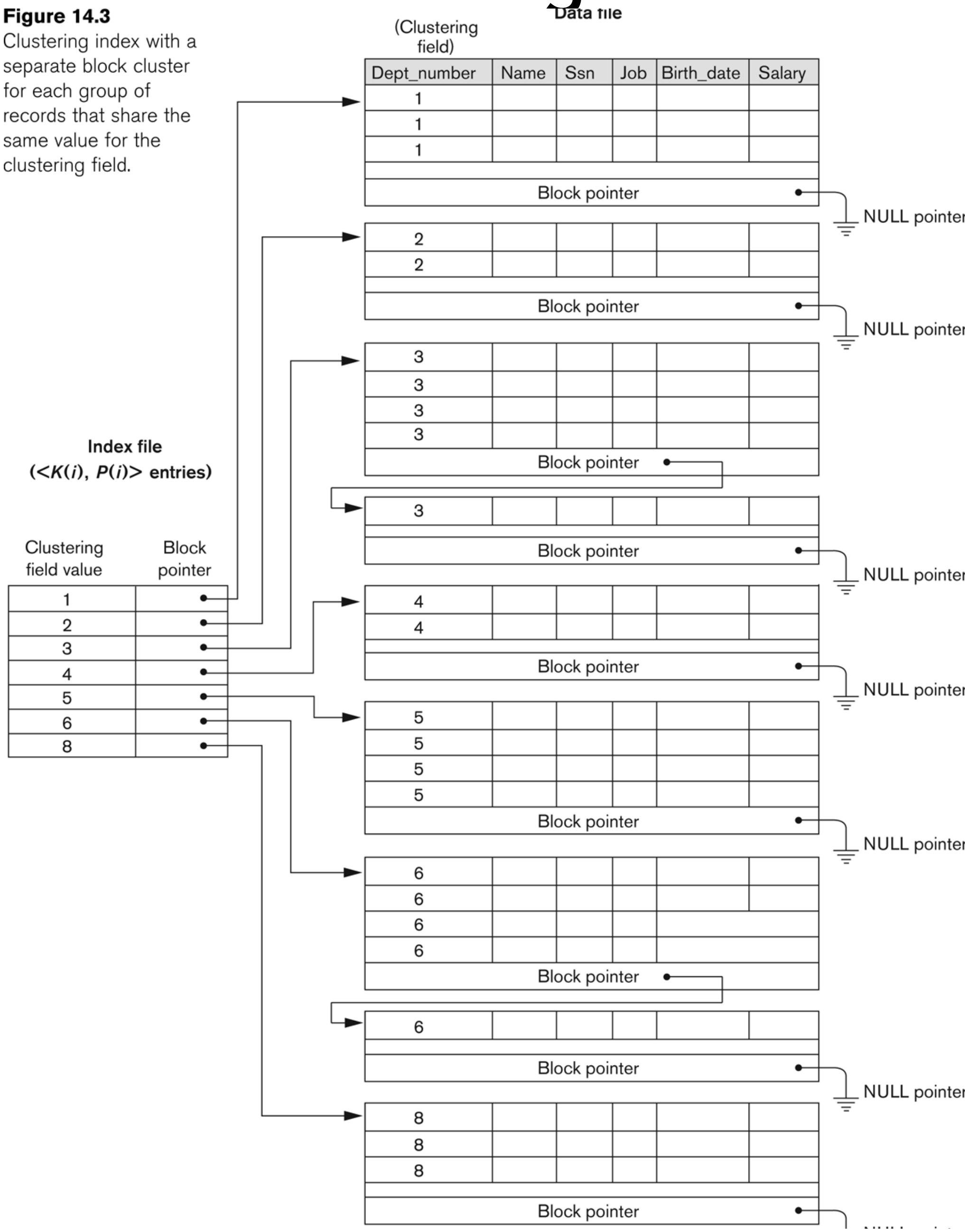
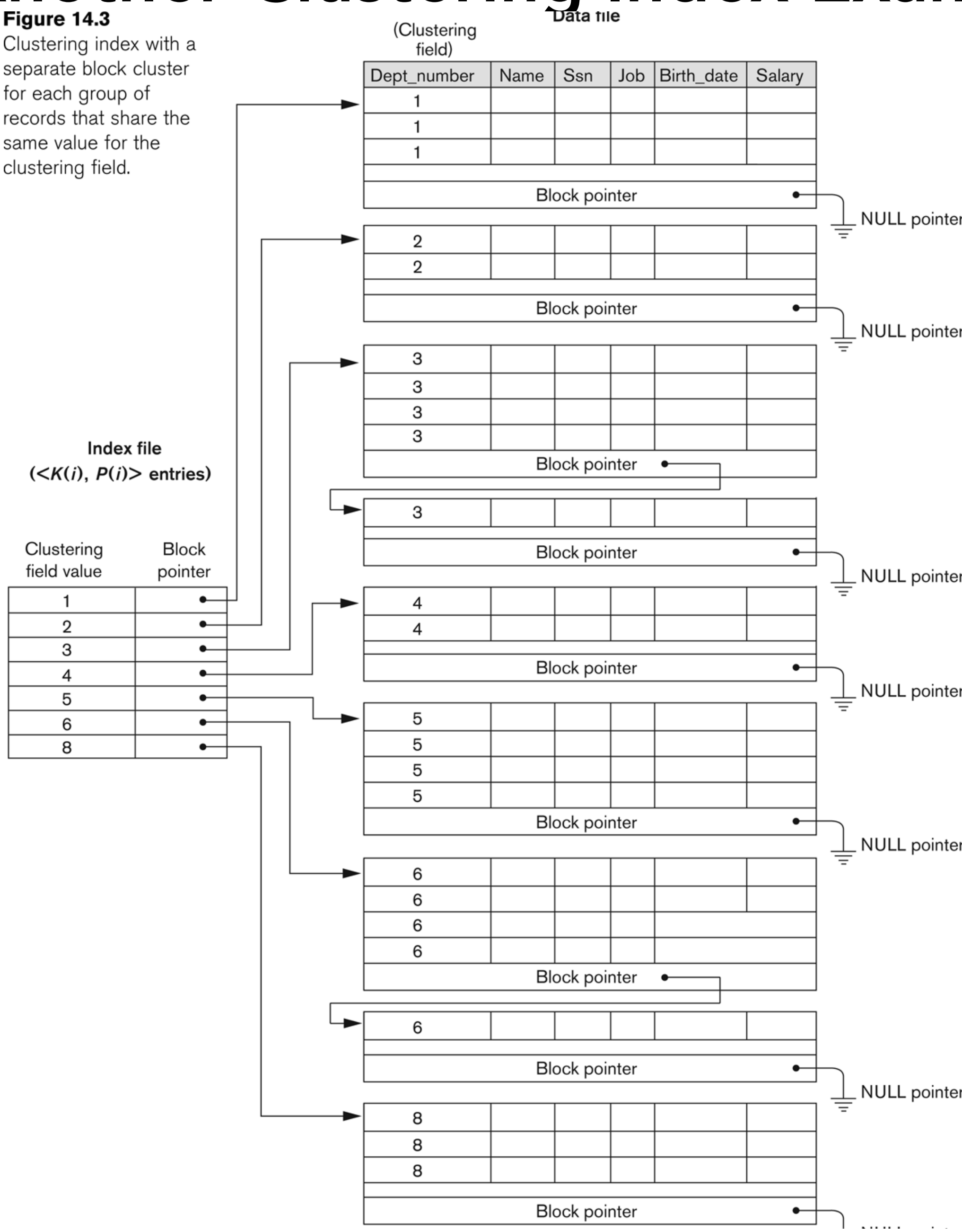


Figure 14.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

Another Clustering Index Example

Figure 14.3
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.



Dense Scondary Index Example

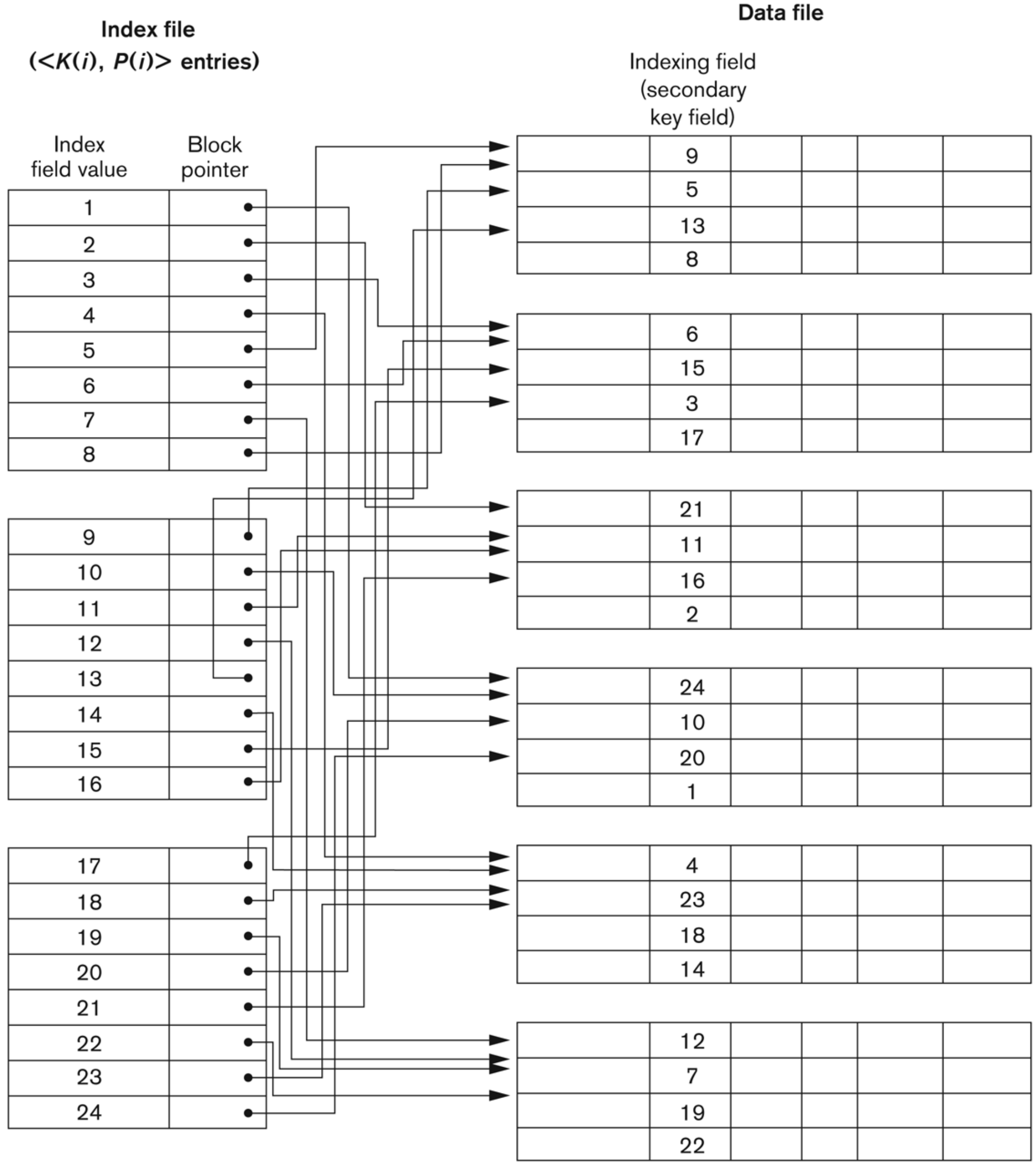


Figure 14.4
A dense secondary index (with block pointers) on a nonordering key field of a file.

Secondary Index Example

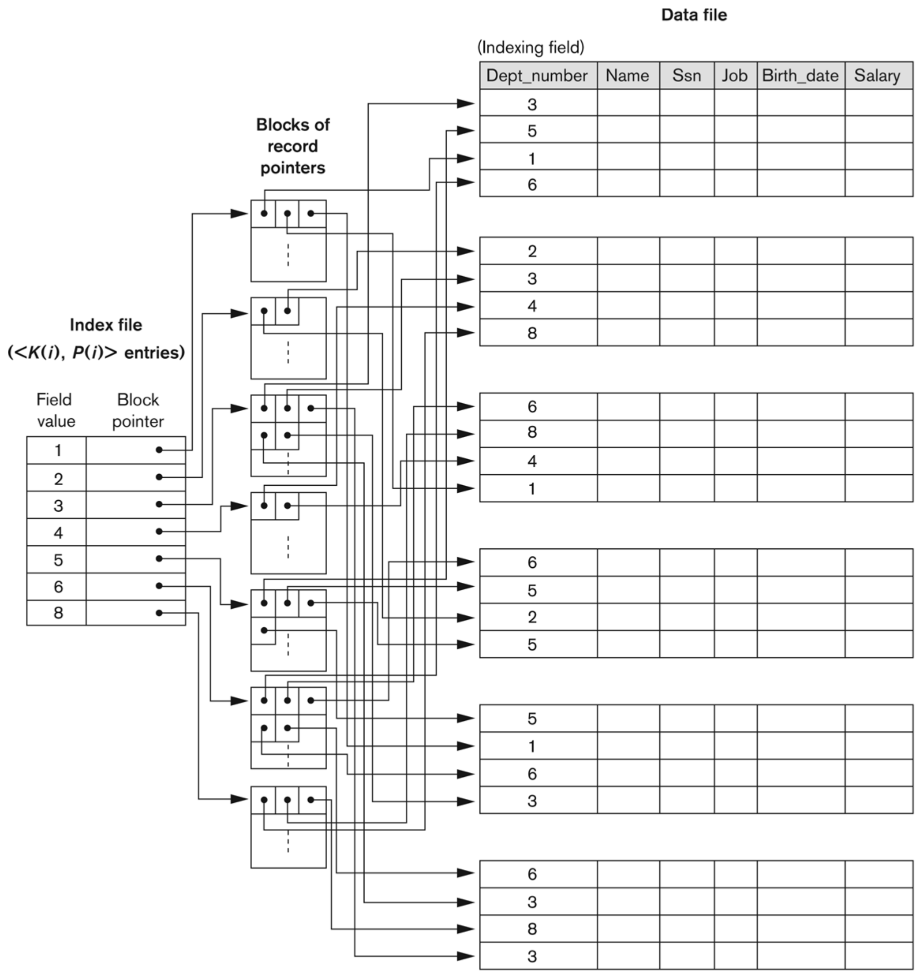
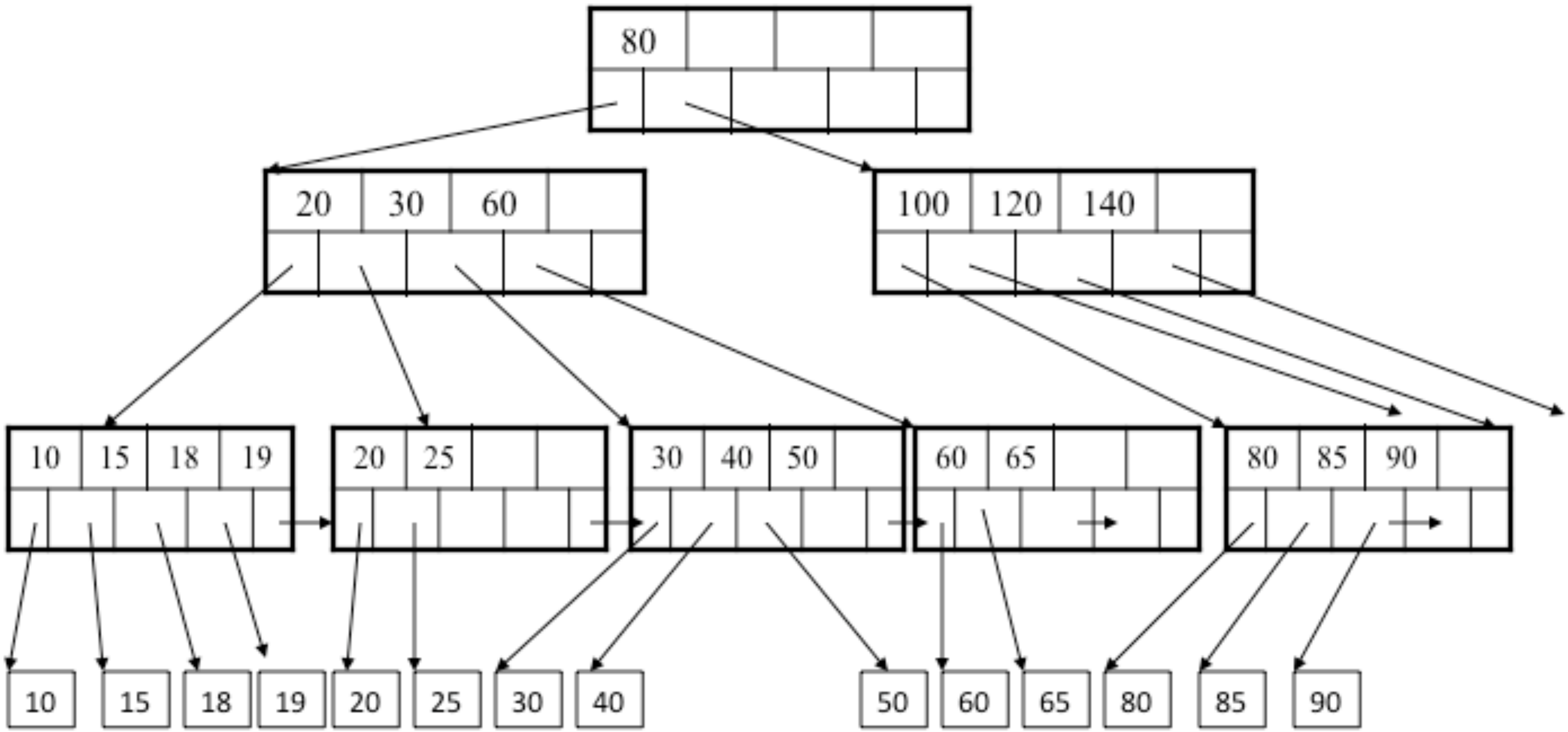


Figure 14.5
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

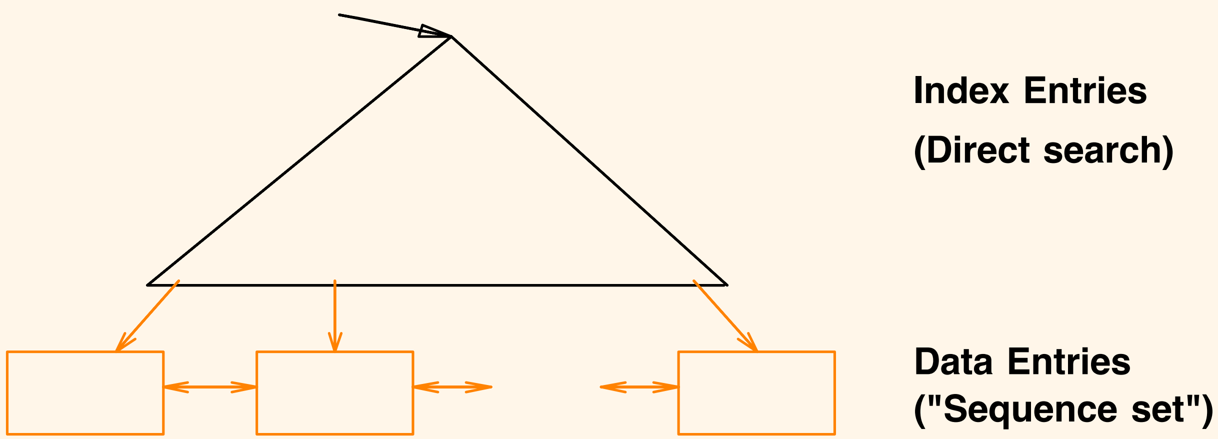
B+ Tree



80
20
30
60
100
120
140
10
15
18
19
20
25
30
40
50
60
65
80
85
90
10
15
18
19
20
25
30
40
50
60
65
80
85
90

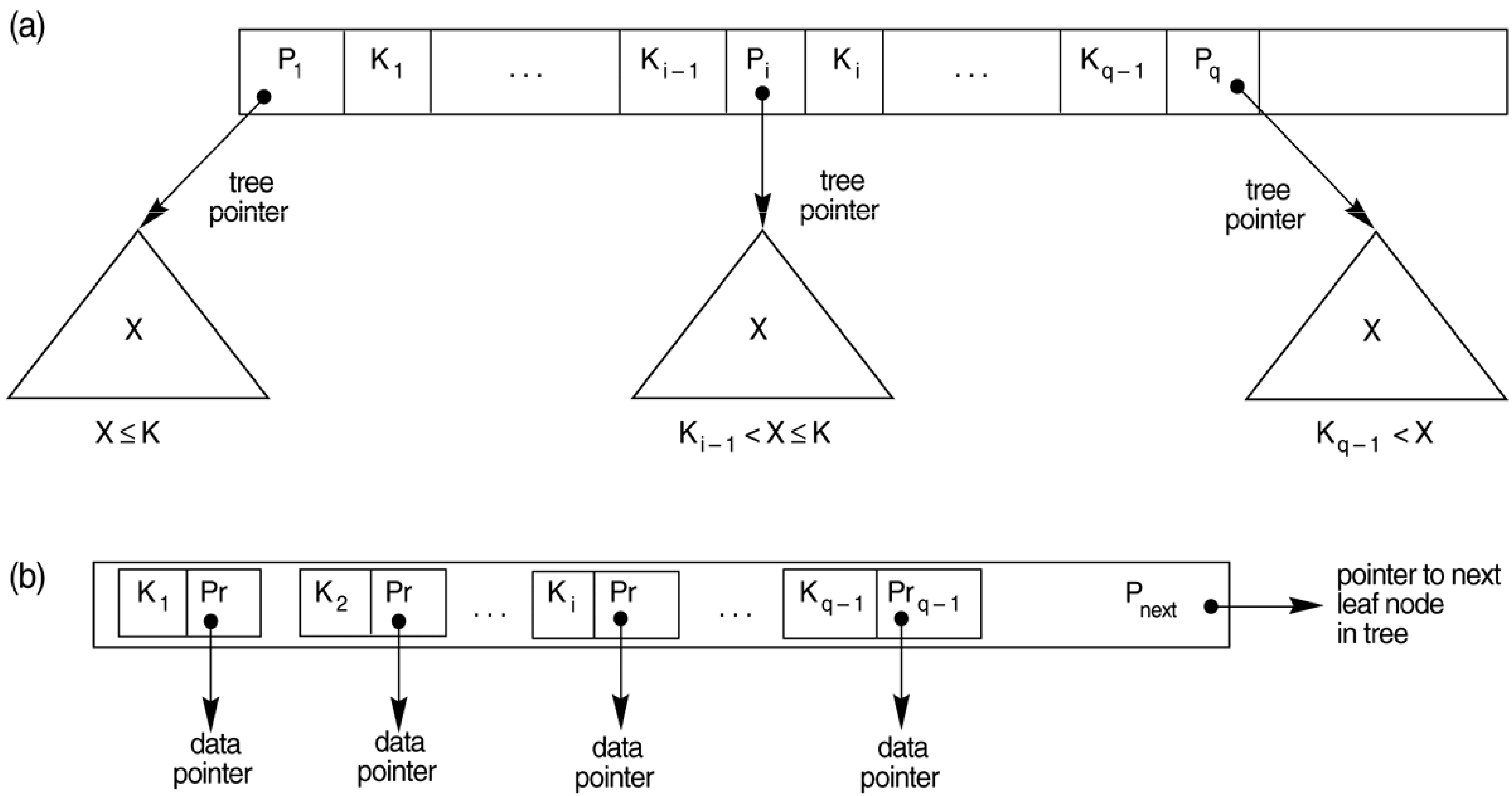
B+ Tree

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



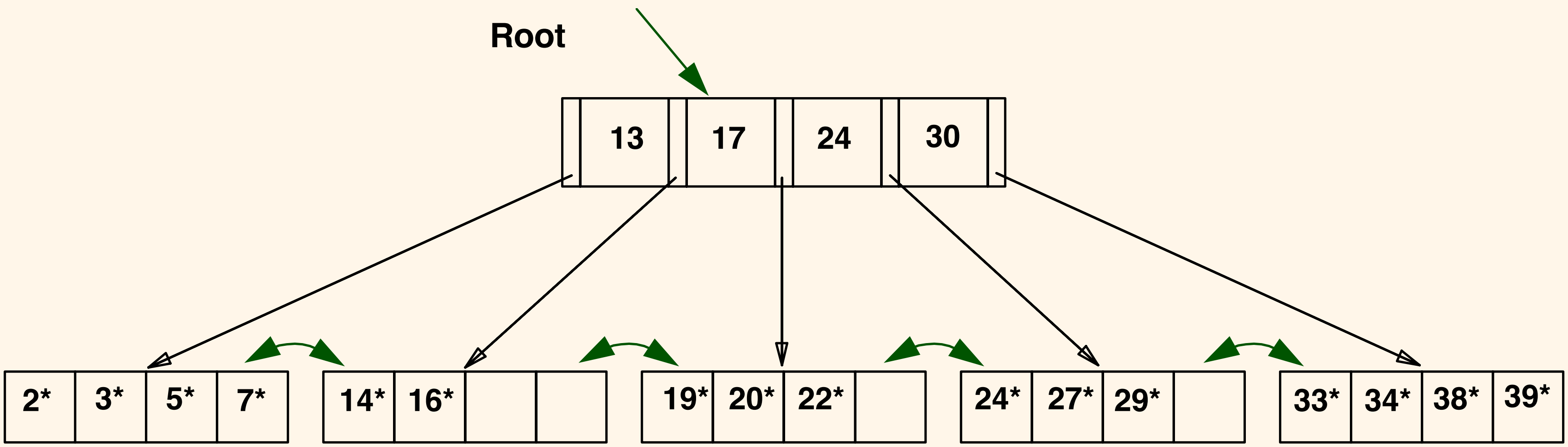
B+ Tree

- FIGURE 14.11 The nodes of a B+-tree
  - (a) Internal node of a B+-tree with q –1 search values.
  - (b) Leaf node of a B+-tree with q – 1 search values and q – 1 data pointers.



B+ Tree Search

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- 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+ Tree in practice

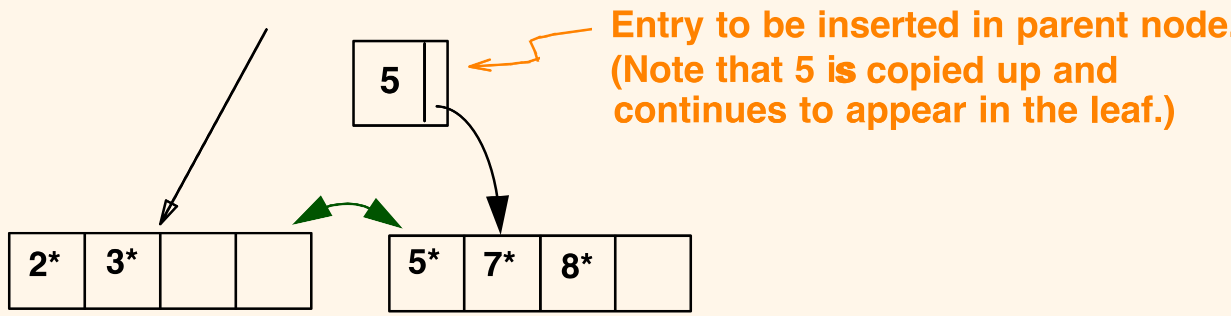
- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

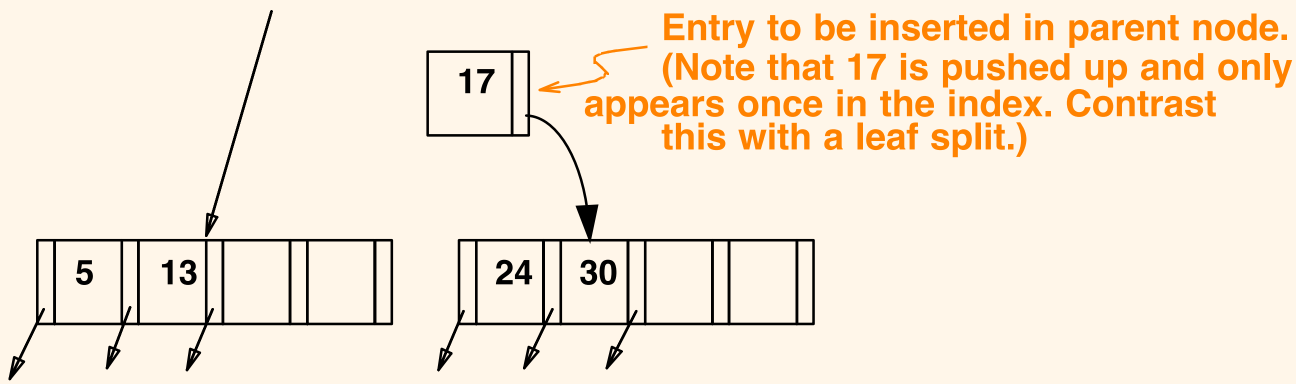
Inserting B+ Tree

- 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.*

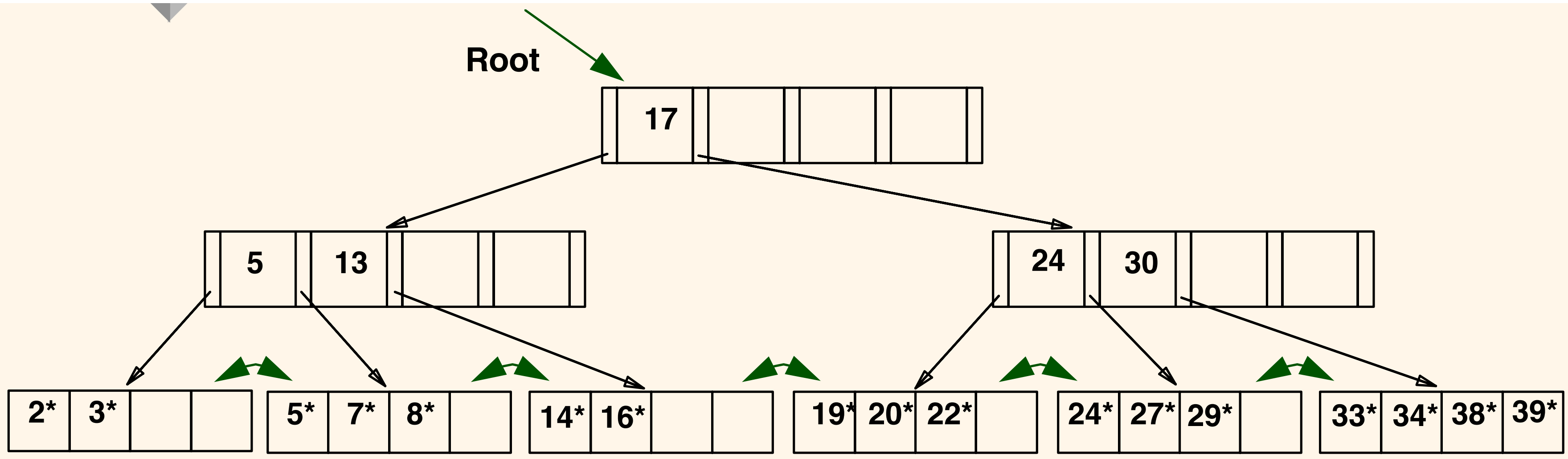
Inserting B+ Tree

- 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.





Inserting node 8 B+ Tree



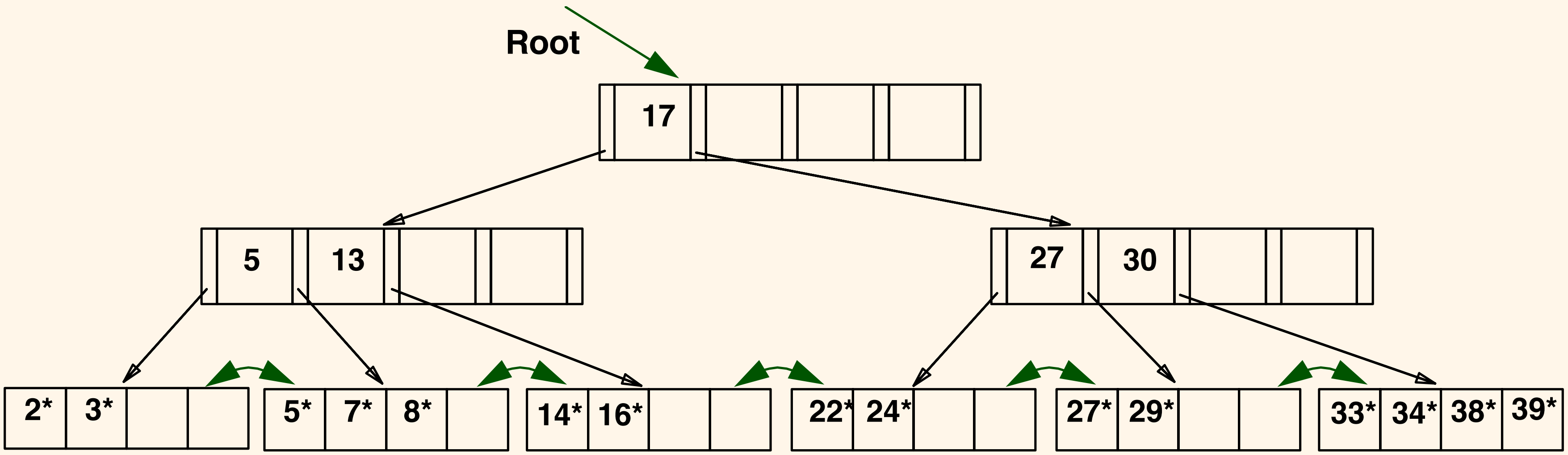
❖ Notice that root was split, leading to increase in height.

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

Deleting 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.

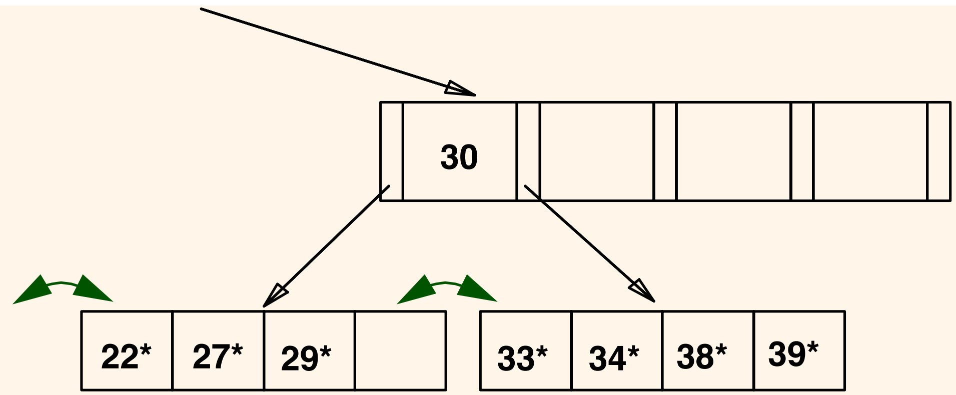
Deleting B+ Tree

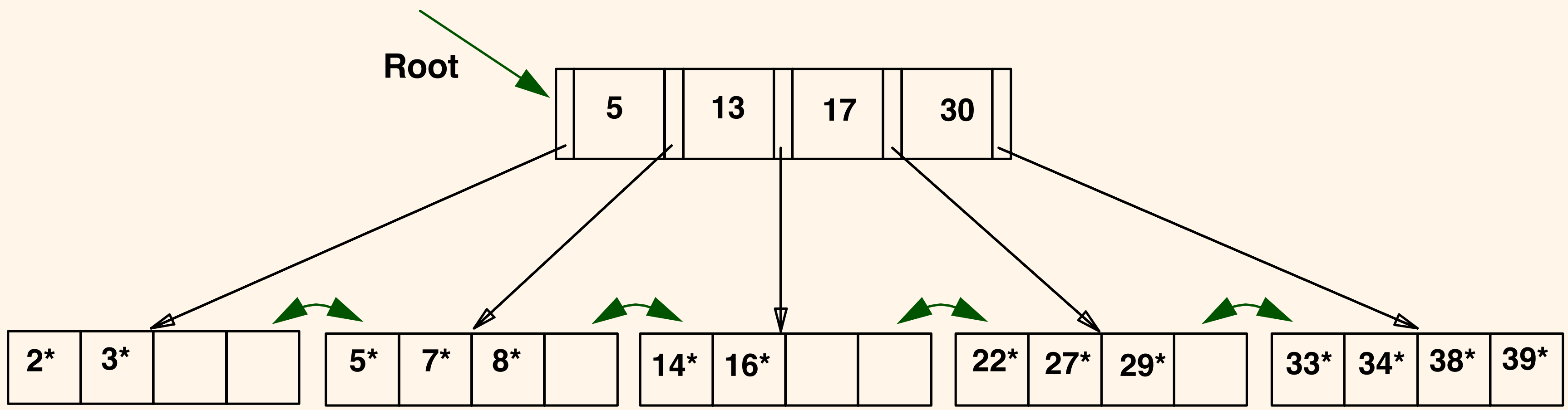


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

Deleting 24 B+ Tree
toss 27, pull down 17

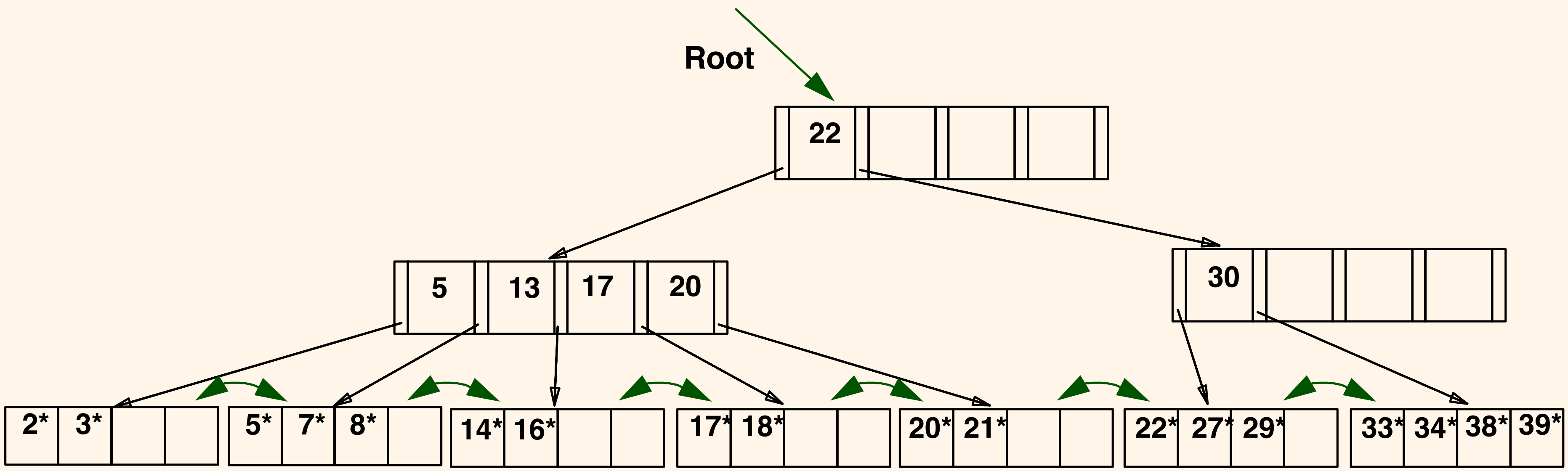
- Must merge.
- Observe *`toss'* of index entry (on right), and *`pull down'* of index entry (below).





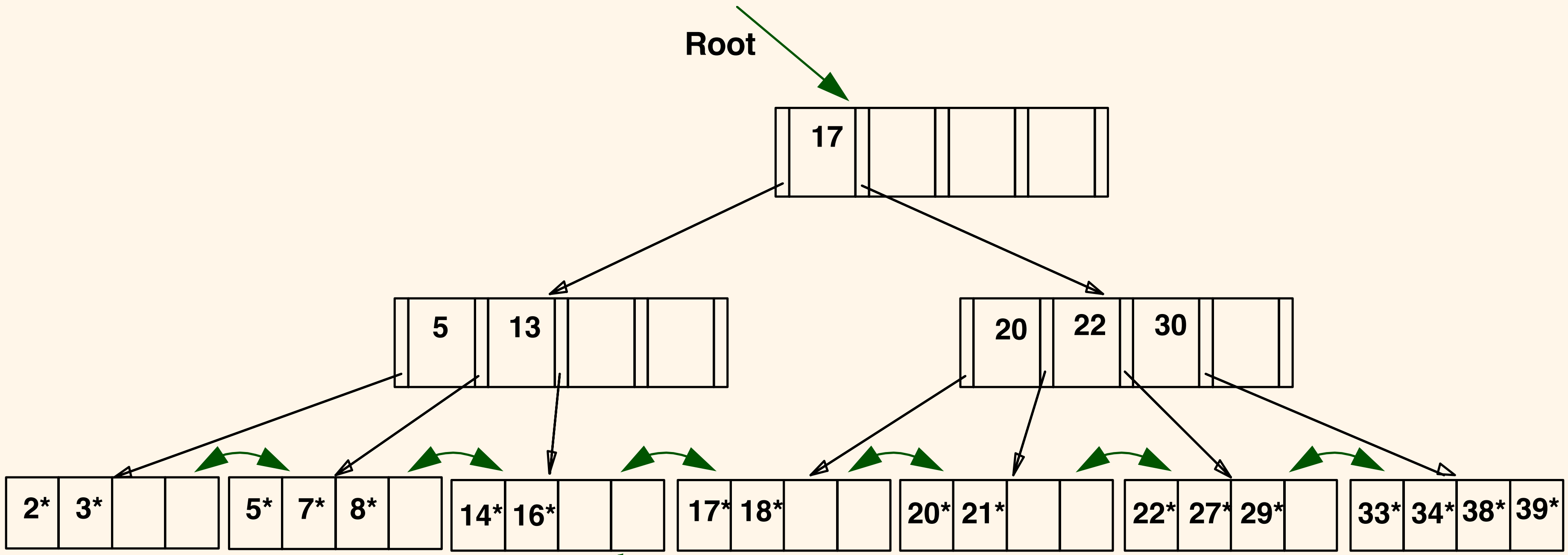
Example of non-leaf redistribution

- 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 redistribution

- 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.



http://www.cs.usfca.edu/~galles/visualization/BPlusTree.html