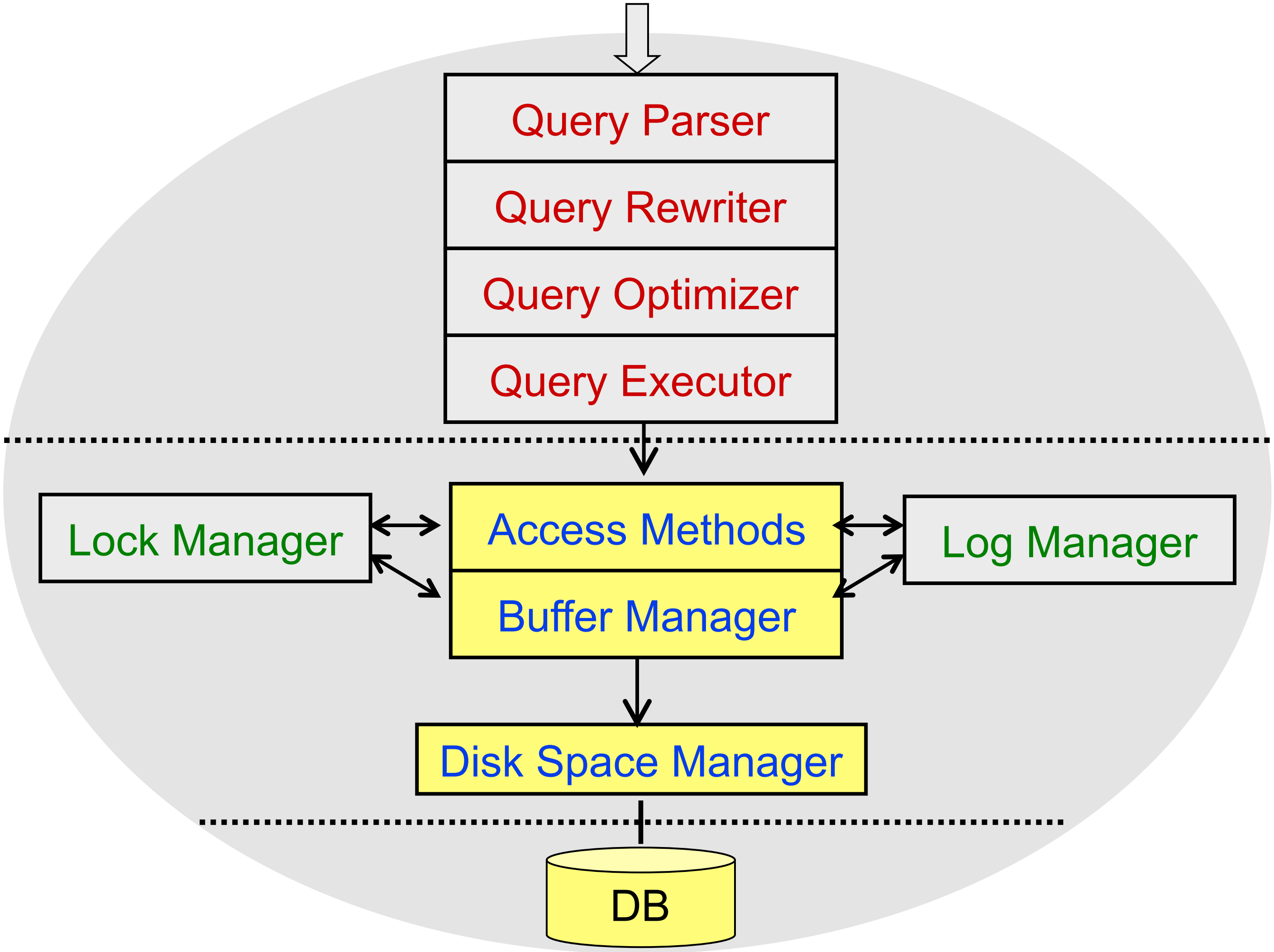
Disks, Files, and Indexes

Yanlei Diao

Slides Courtesy of R. Ramakrishnan and J. Gehrke

DBMS Architecture



Outline

- Disks, Disk Space Manager
- Disk-Resident Data Structures
  - Files of records
  - Indexes
    - Tree indexes: B+ tree
    - Hash indexes

Memory Hierarchy

- Main Memory (RAM)
  - Random access, fast, usually volatile
  - Main memory for currently used data
- Magnetic Disk
  - Random access, relatively slow, nonvolatile
  - Persistent storage for all data in the database.
- Tape
  - Sequential scan (read the entire tape to access the last byte), nonvolatile
  - For archiving older versions of the data.

Disks and DBMS Design

- A database is stored on disks. This has major implications on DBMS design!
  - READ: transfer data from disk to RAM for **data processing**.
  - WRITE: transfer data (new/modified) from RAM to disk for **persistent storage**.
  - Both are high-cost operations relative to in-memory operations, so must be planned carefully!

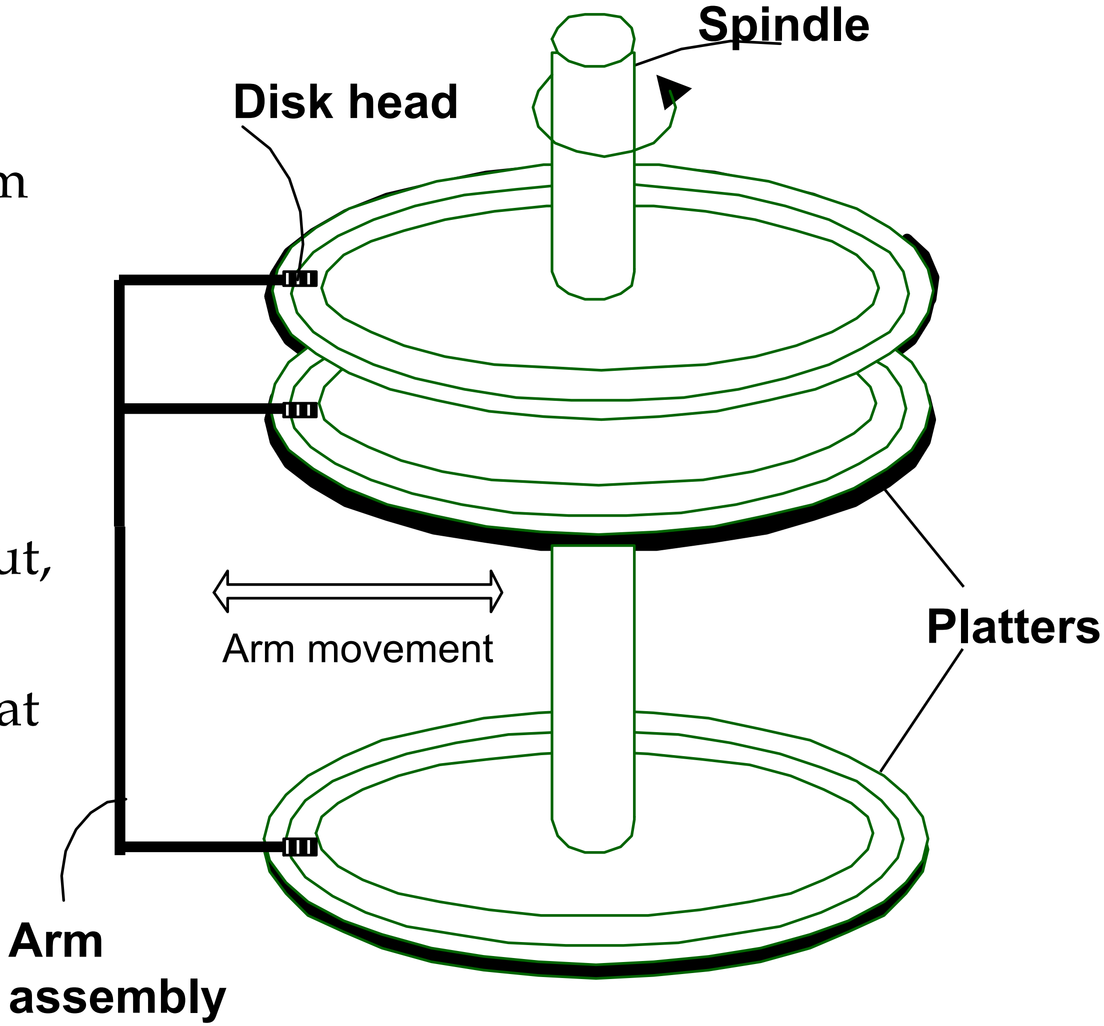
Basics of Disks

- Unit of storage and retrieval: *disk block* or *page*.
  - A contiguous sequence of bytes.
  - Size is a DBMS parameter, 4KB or 8KB.
- Unlike RAM, **time to retrieve a page** varies!
  - It depends upon the location on disk.
  - Relative placement of pages on disk has major impact on DBMS performance!

Components of a Disk

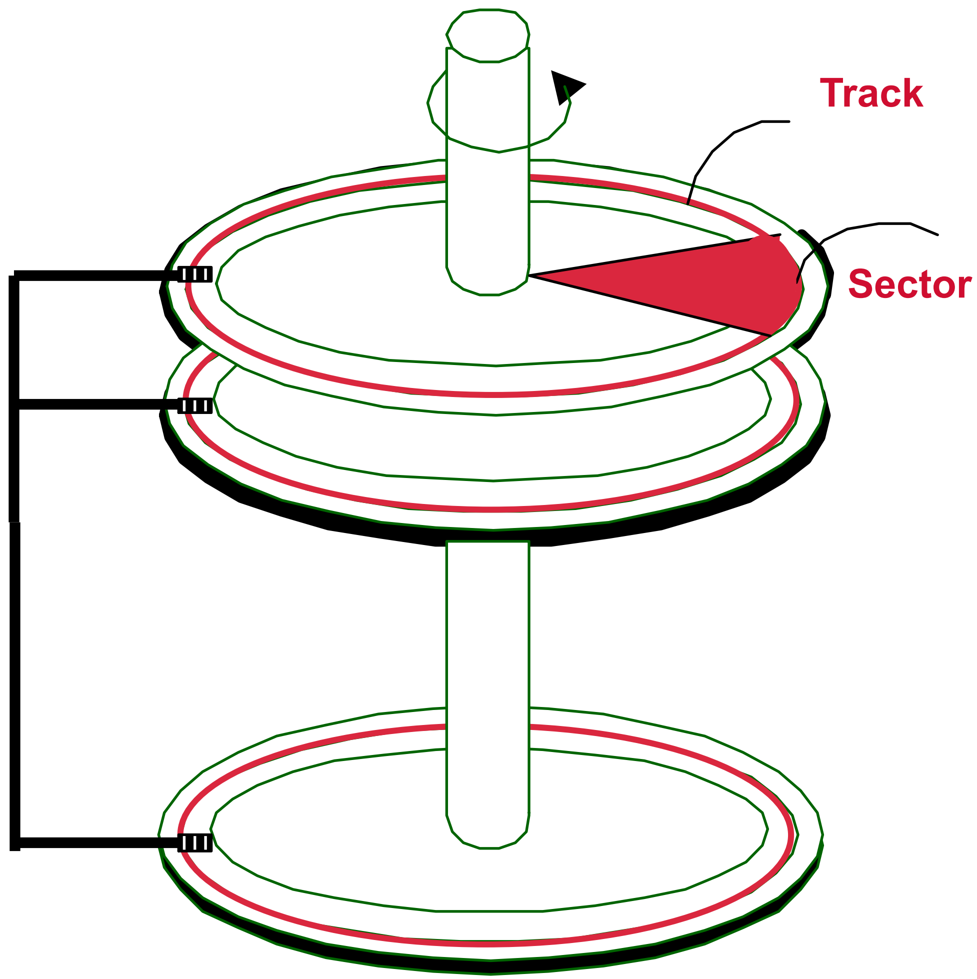
- *Spindle*, *Platters*
  E.g. spin at 7200 or 15,000 rpm (revolutions per minute)

- *Disk heads*, *Arm assembly*
  - Arm assembly moves in or out, e.g., 2-10ms
  - Only one head reads/writes at any one time.



Data on Disk

- ❖ A platter consists of *tracks*.
  - single-sided platters
  - double-sided platters
- ❖ Tracks under heads make a *cylinder* (imaginary!)
- ❖ Each track is divided into *sectors* (whose size is fixed).
- ❖ *Block* (*page*) *size* is a multiple of *sector size* (DBMS parameter).



Accessing a Disk Page

- Time to access (read/write) a disk block:
  1. *seek time* (moving arms to position a disk head on a track)
  2. *rotational delay* (waiting for a block to rotate under the head)
  3. *transfer time* (actually moving data to/from disk surface)
- Seek time and rotational delay dominate.
  - *seek time*: 2 to 10 msec
  - *rotational delay*: 0 to 10 msec
  - *transfer rate*: <1msec/page, or 10's-100's megabytes/sec
- Key to lower I/O cost: reduce seek/rotation delays! Hardware vs. software solutions?

Arranging Pages on Disk

- Software solution uses the *'next'* block concept:
  - blocks on the same track, followed by
  - blocks on the same cylinder, followed by
  - blocks on an adjacent cylinder

- Pages in a *file* should be arranged sequentially on disk (by `next'), to minimize seek and rotational delay.
  - Scan of the file is a *sequential scan*.

Disk Space Manager

- Lowest layer of DBMS managing space on disk. Higher levels call it to:
  - allocate/de-allocate a page
  - allocate/de-allocate a sequence of pages
  - read/write a page

- Requests for a sequence of pages are satisfied by *allocating the pages sequentially* on disk!
  - Higher levels don't need to know any details.

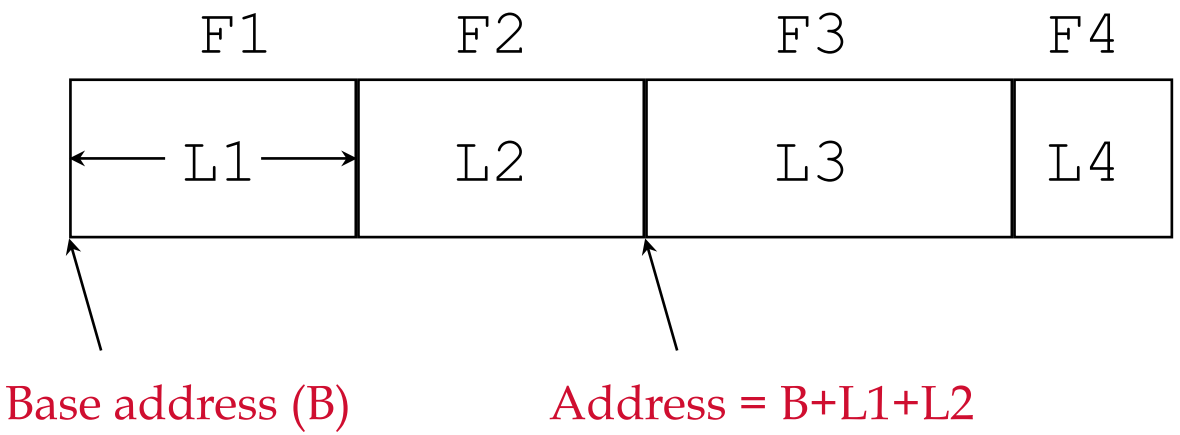
Outline

- Disks, Disk Space Manager
- Disk-Resident Data Structures
  - Files of records
  - Indexes
    - Tree indexes: B+ tree
    - Hash indexes

File of Records

- Abstraction of disk-resident data for query processing: *a file of records residing on multiple pages*
  - A number of *fields* are organized in a *record*
  - A collection of records are organized in a *page*
  - A collection of pages are organized in a *file*

Record Format: Fixed Length

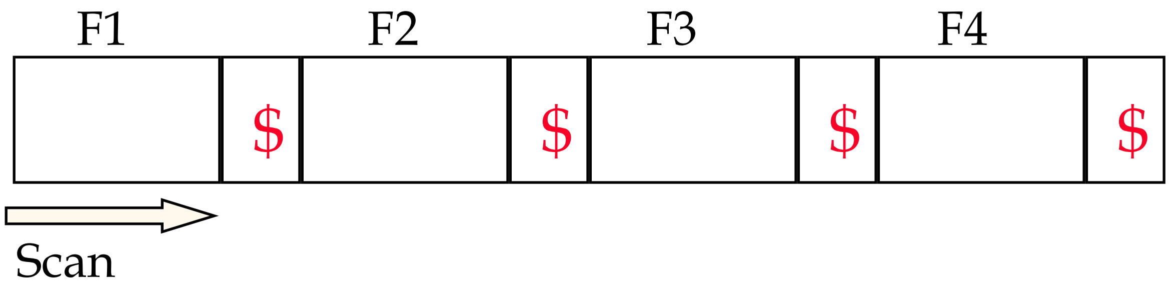


- Record type: the *number of fields* and *type of each field* (defined in the schema), stored in *system catalog.*
- Fixed length record: (1) the number of fields is fixed, (2) each field has a fixed length.
- Store fields consecutively in a record. How do we find *i'th* field of the record?

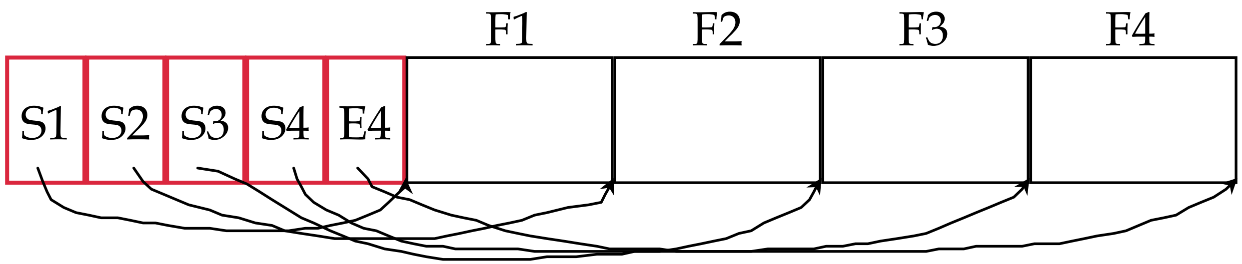
Record Format: Variable Length

- Variable length record: (1) number of fields is fixed, (2) some fields are of variable length

Fields Delimited by Special Symbols



Array of Field Offsets



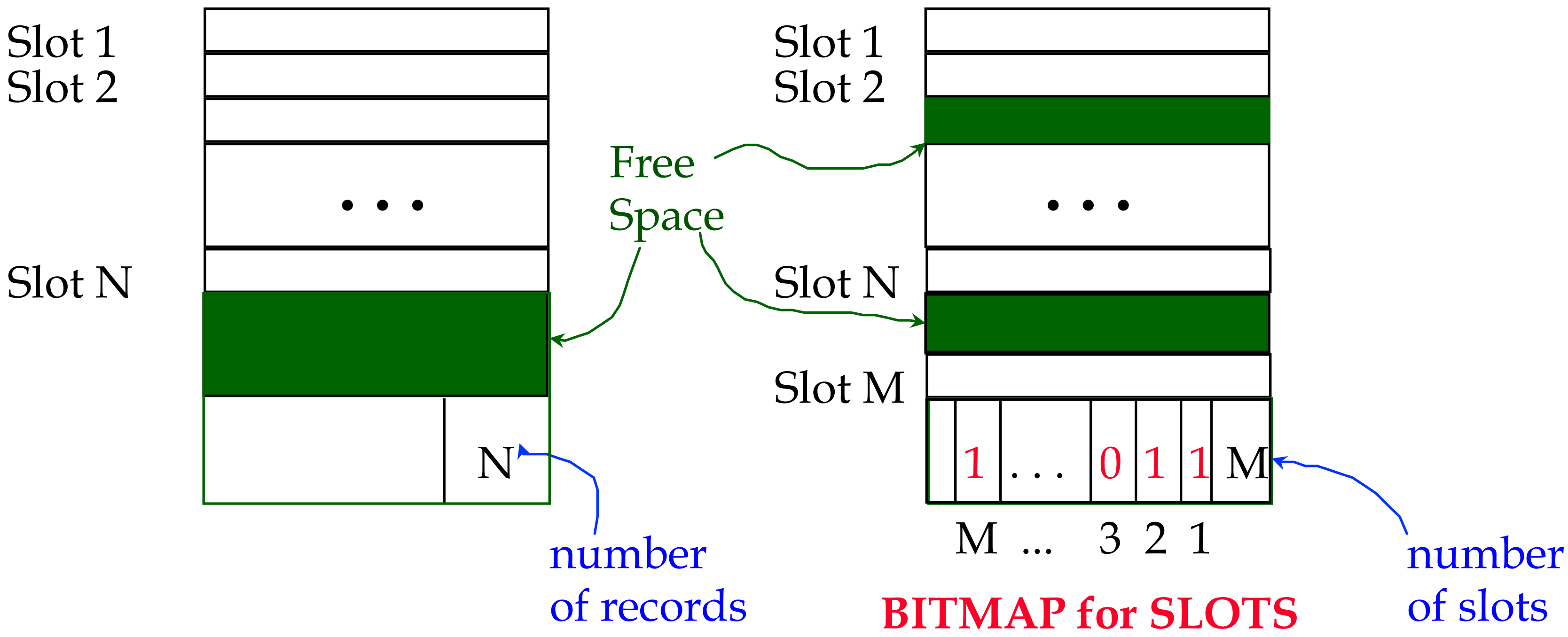
2nd choice offers direct access to i' th field; but small directory overhead.

Page Format

- How to store a collection of records on a *page*?

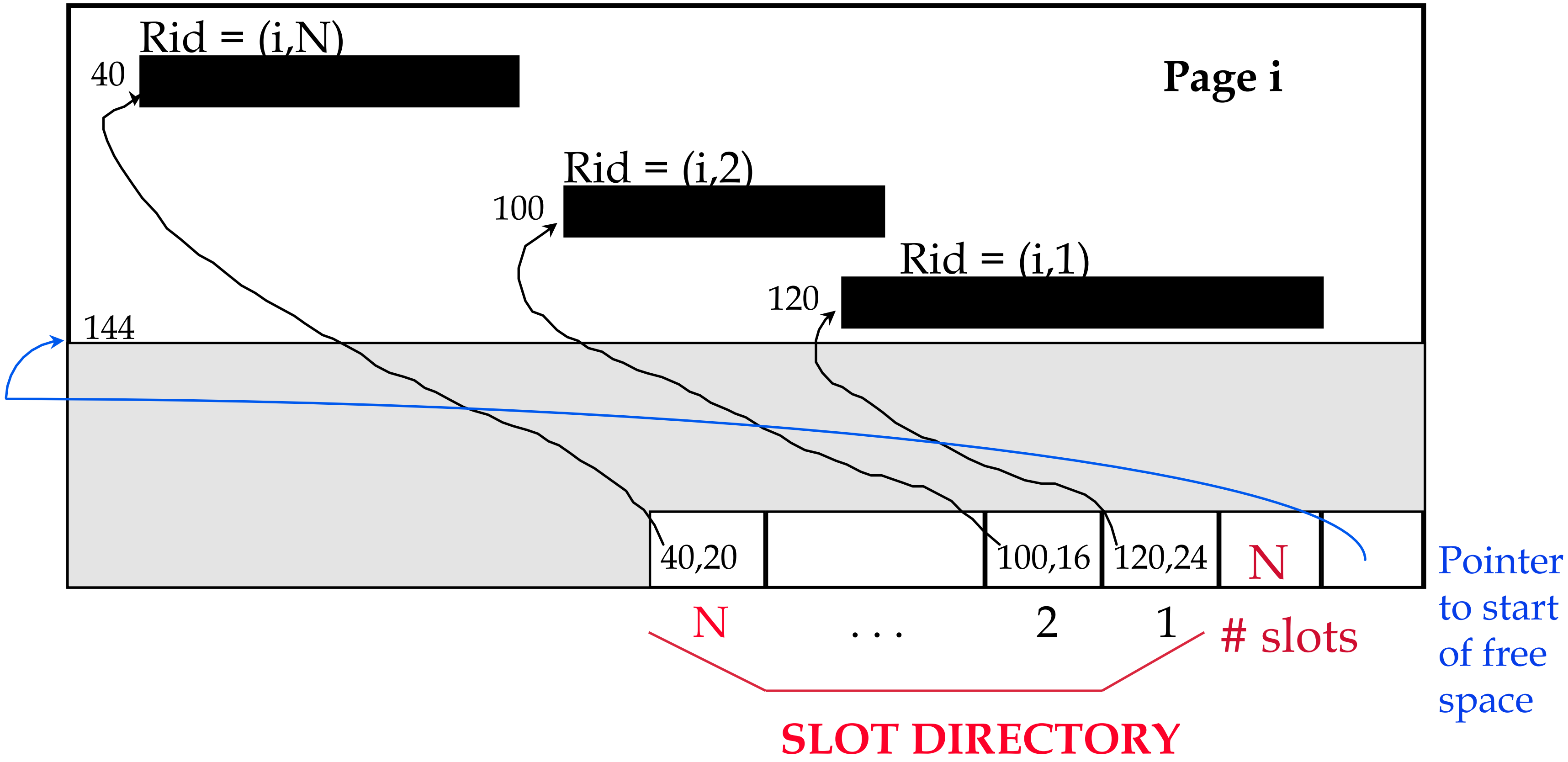
- View a page as a collection of *slots*, one for each record.
- A record is identified by *rid* = <page id, slot #>
  - Record ids (rids) are used in indexes. More on this later…

Page Format: *Fixed Length Records*



☛ *If we move records for free space management, we may change rids! Unacceptable for performance.*

Page Format: *Variable Length Records*



- *Compaction: get all slots whose offset is not -1, sort by start address, move their records up in sorted order. No change of rids!*

Files of Records

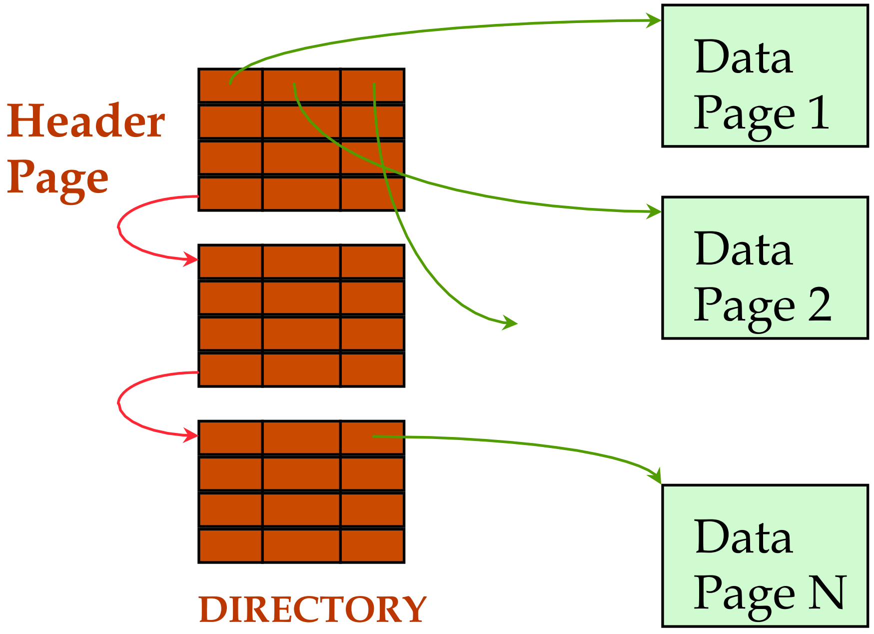
- *File*: a collection of pages, each containing a collection of records. Typically, one file for each relation.
  - *Updates*: insert/delete/modify records
  - *Index scan*: read a record given a *record id* – more later
  - *Sequential scan*: scan all records (possibly with some conditions on the records to be retrieved)

- Files in DBMS versus Files in OS?

Heap (Unordered) Files

- *Heap file*: contains records in no particular order.
- As a file grows and shrinks, disk pages are allocated and de-allocated.
- To support record-level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on a page

Heap File Using a Page Directory



- A directory entry per page: a pointer to the page, # free bytes on the page.
- The directory is a collection of pages; a linked list is one implementation.
  - **Much smaller** than the linked list of all data pages.
- Search for space for insertion: fewer I/Os.

Outline

- Disks, Disk Space Manager
- Disk-Resident Data Structures
  - Files of records
  - Indexes
    - Tree indexes: B+ tree
    - Hash indexes

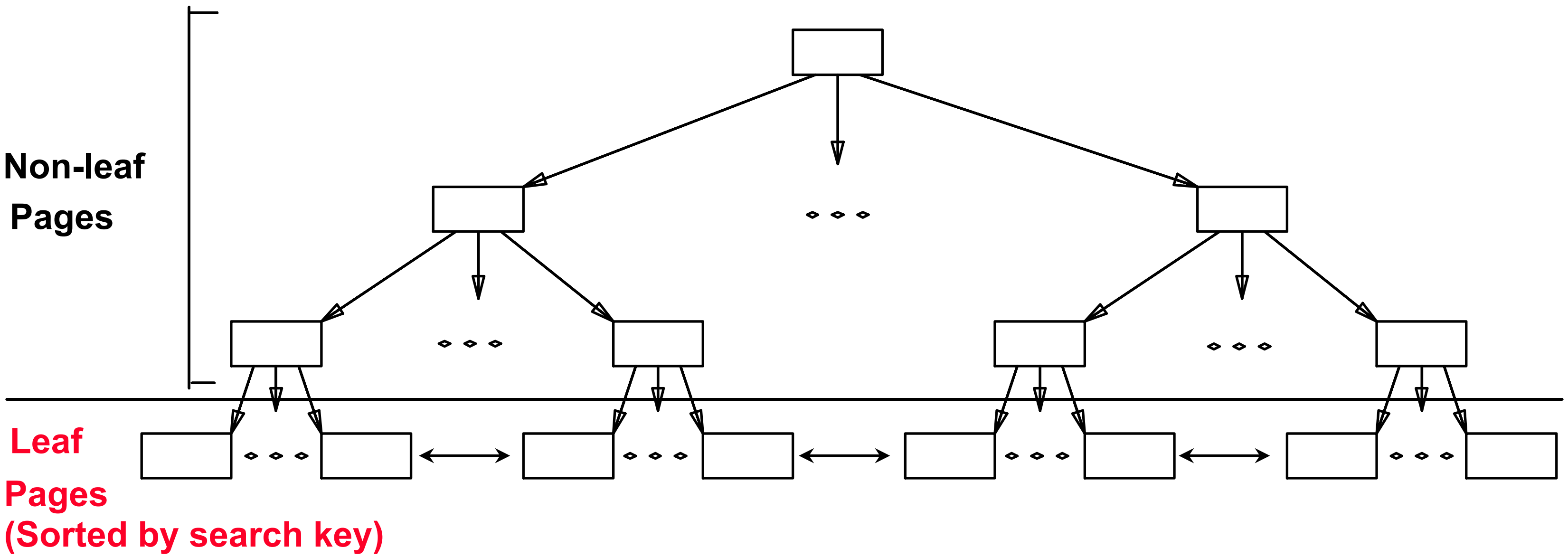
Access Methods

- Routines that manage disk-based data structures.
- *File of records*:
  - Abstraction of external storage for query processing
  - (1) *Sequential scan*; (2) Locate a record using *record id (rid)*
    - E.g., retrieve all sailor records, or a record w. (page 4, slot 2)
- *Indexes*:
  - Auxiliary data structures
  - *Associative access*: given a value in the *index search key*, find the (record ids of) records with this value.
    - E.g., find all sailors with rating > 5.

Outline

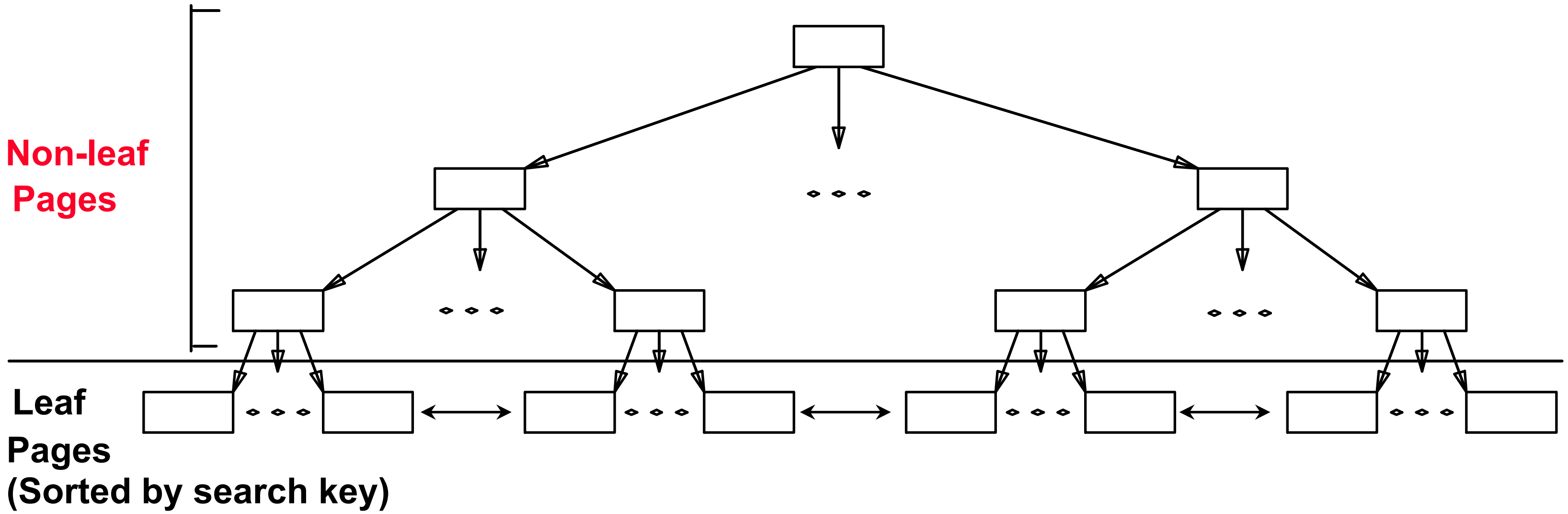
- Disks, Disk Space Manager
- Disk-Resident Data Structures
  - Files of records
  - Indexes
    - Tree indexes: B+ tree
    - Hash indexes

B+ Tree Indexes

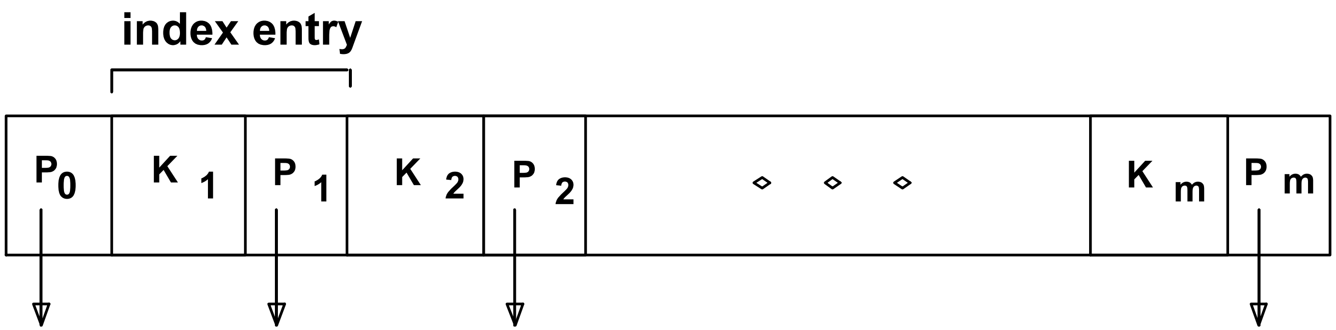


- Leaf pages contain *data entries*:
  - Data entries are *sorted* by the search key value
  - Leaf pages are chained using prev & next pointers

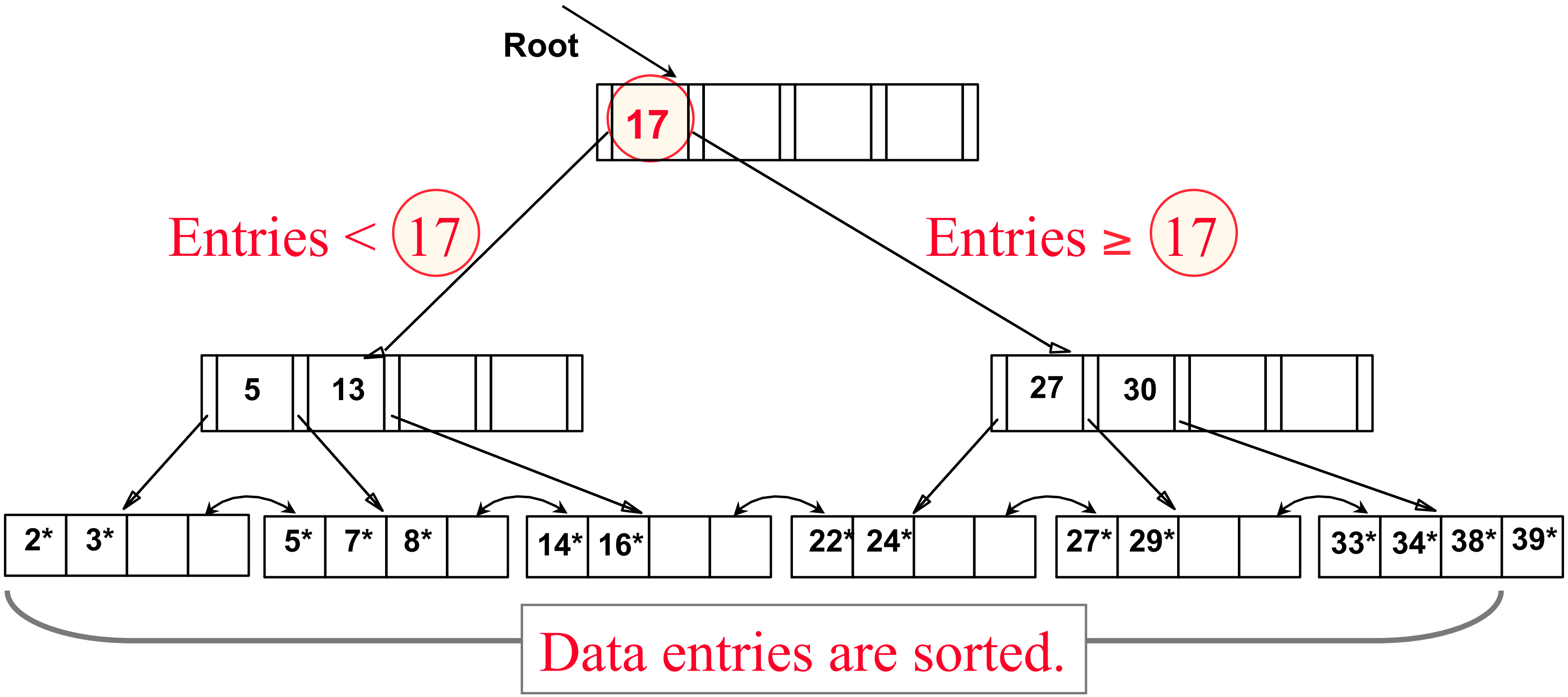
B+ Tree Indexes



❖ Non-leaf pages have *index entries*, used **only** to direct searches.

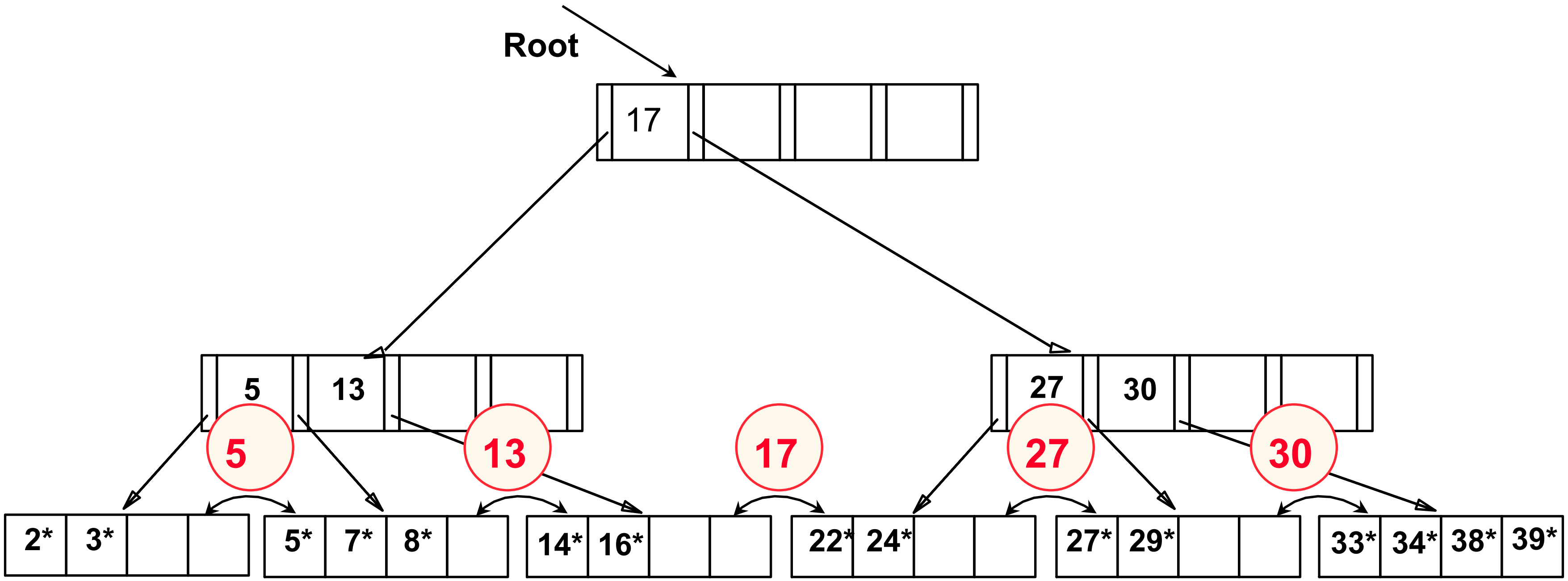


Example B+ Tree



- *Equality selection*: find 28*? 29*?
- *Range selection*: find all > 15* and < 30*
- *Insert/delete*: Find the data entry in a leaf, then change it. More later…

Example B+ Tree



(1) Index Classification

Alternatives for Data Entry **k*** in Indexes:

- In a data entry k*, we can store:
  - Alternative 1: <**k**, data record with search key value **k**>
  - Alternative 2: <**k**, rid of a record with search key value **k**>
  - Alternative 3: <**k**, list of rids of records with search key **k**>

- Choice of an *alternative for data entries* is orthogonal to an *indexing technique* used.
  - Indexing techniques: B+ tree, hashing, …

Alternative 1 for Data Entries

- Alternative 1 (*primary index*):
  - Data records are physically stored in leaf pages.
  - Given a collection of data records (*no replication*), at most one index can use Alternative 1.
  - If data records are large, the num. of leaf pages is large.
    - The non-leaf part of the index (used to direct searches) can also be large, hence slowing down search.

Alternatives 2, 3 for Data Entries

- Alternatives 2 and 3 (*secondary index*):
  - <k, rid(s)>: store rid, not the record.
  - Data entries are typically much smaller than data records.
    - Index < data file. Index search structure is compact.
  - Alternative 3 is more compact than Alternative 2.
    - In the presence of multiple entries of the same key value!
  - Alternative 3 leads to variable sized data entries, even if search keys are of fixed length.

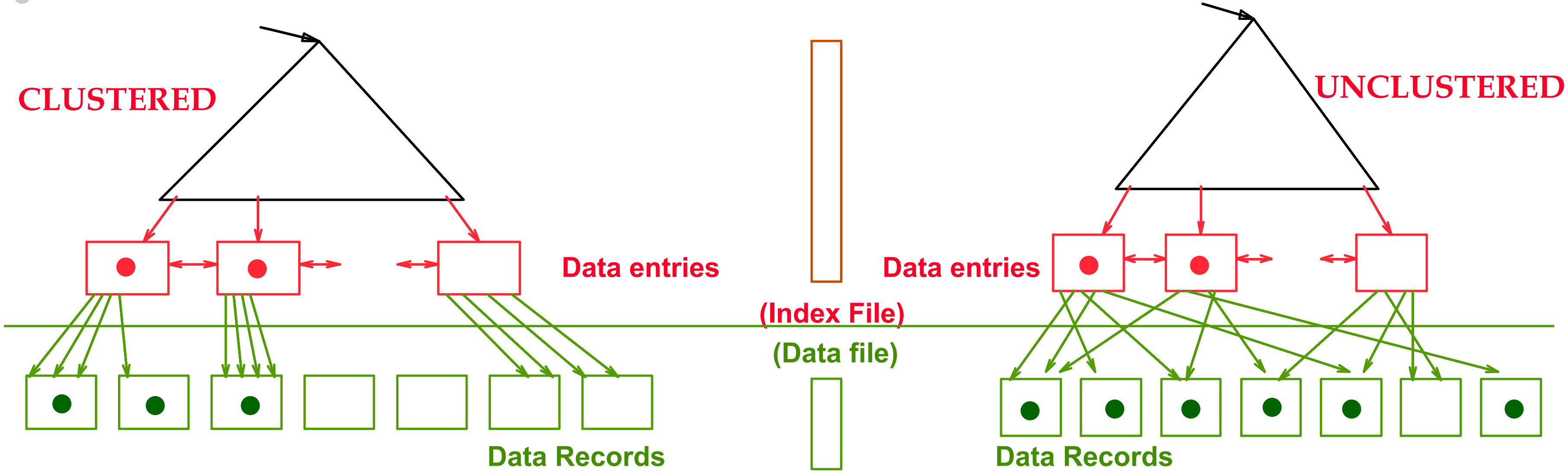
Index Classification (Contd.)

- *Clustered* index: order of data records in a file is the same as or `close to' order of (sorted) data entries in the index.
  - The data file is (almost) sorted by the index's search key.
  - A data file can have at most one clustered index.

  - Alternative 1 with a tree index is always clustered.
  - Alternatives 2 and 3 are clustered only if data records are sorted on the search key field.

- *Unclustered* index: otherwise
  - Multiple unclustered indexes on a data file.

Clustered vs. Unclustered Indexes



- Retrieve a range of data records matching a search key value:
  - Clustered index: fast, with one or a few I/Os.
  - Unclustered index: can be slow, toughing many pages of data records. 1 I/O per data record in the worst case.

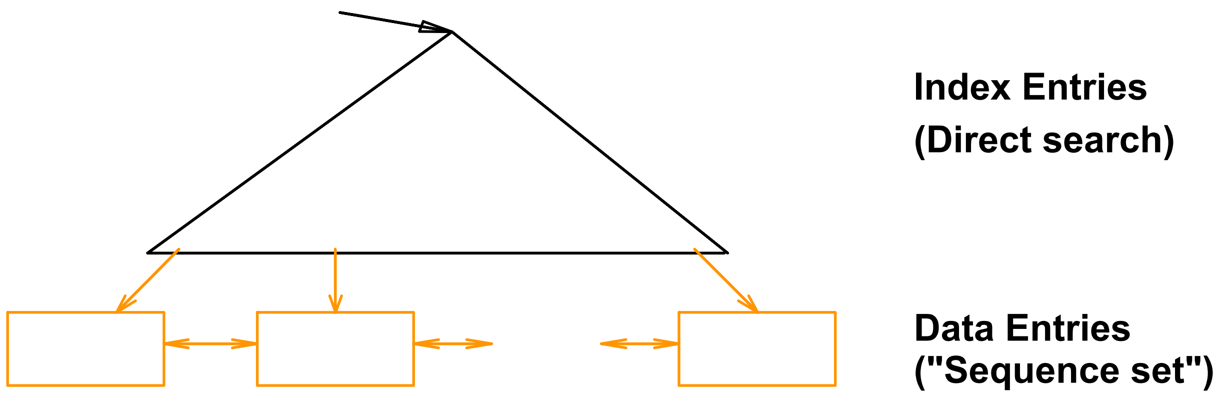
(2) Properties of B+ Tree

- Height-balanced given arbitrary inserts/deletes.
  - *Fanout*: num. of child pointers of a non-leaf node

    F = avg. fanout
  - *Height*: N = num. of leaf pages

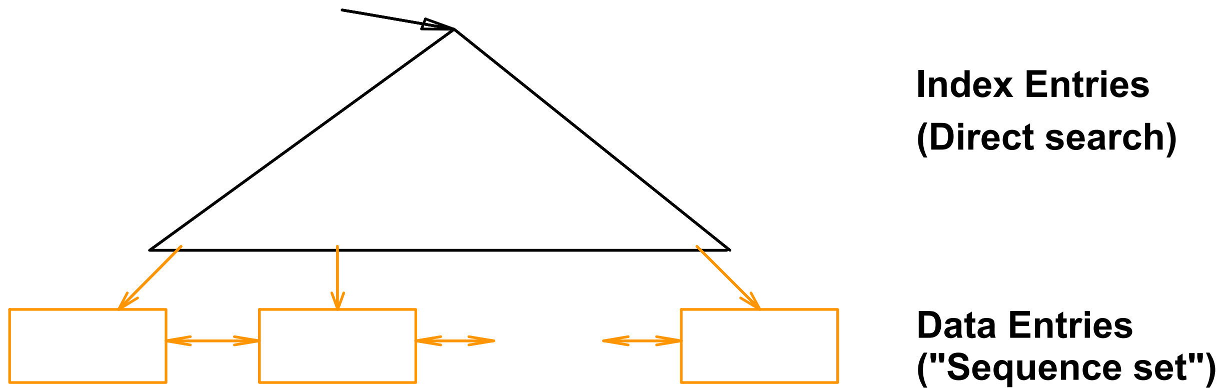
    $H = \mathrm{Log}_F N$

    (Root: level 0, …, Leaf: level H)



Properties of B+ Tree

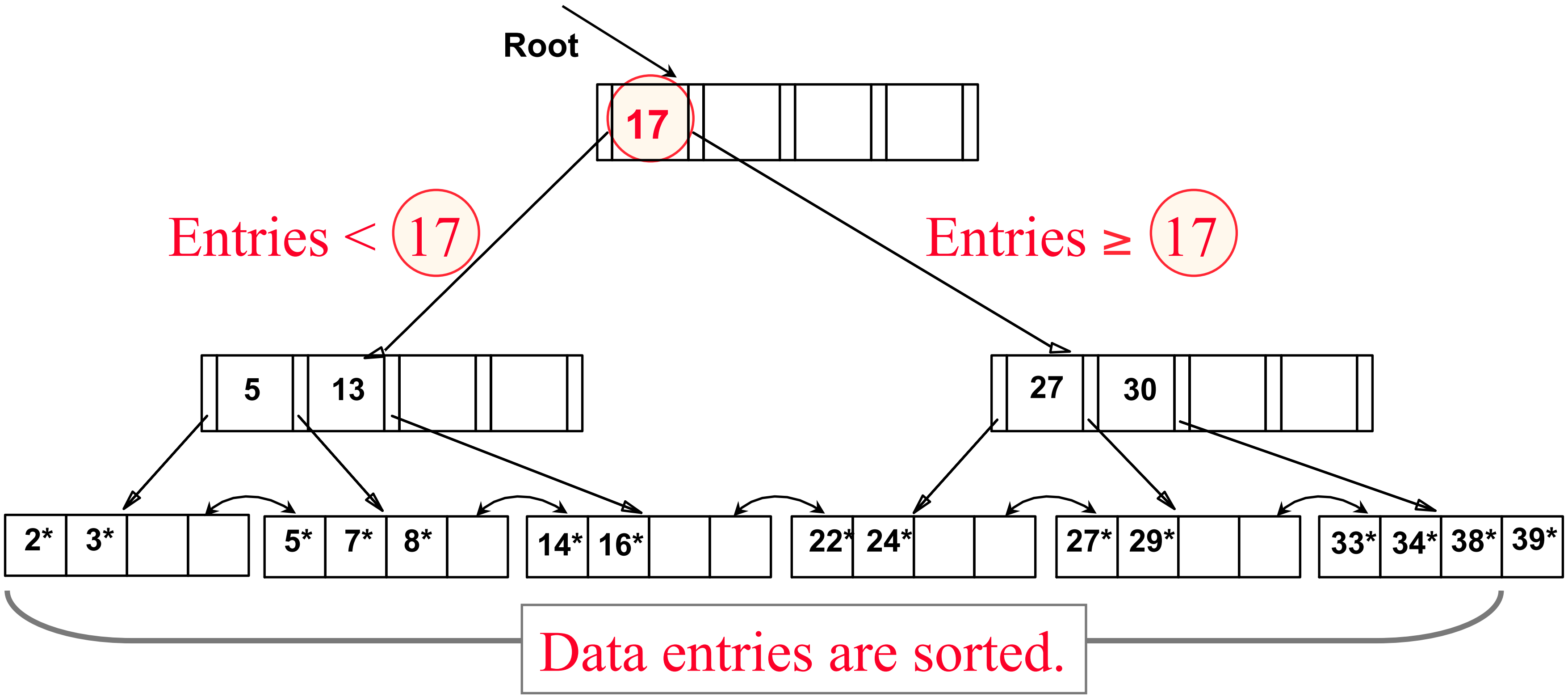
- Minimum 50% occupancy (except for the root).
  - *Order* of the tree (n): max num. of keys in a node.
    - Can be computed using the page size, key size, pointer size.
  - Each non-root node is at least half full, containing [$\lfloor n/2 \rfloor$, n] entries.
  - Root node can have [1, n] entries.



B+ Trees in Practice

- Typical order is 200. Typical fill-factor (occupancy) is 67%.
  - Average fanout = 133
  - Level 0=1 page; Level 1=133 pages; Level 2=133^2 pages…
- Typical capacities:
  - Height 3: 133^3 = 2,352,637 records
  - Height 4: 133^4 = 312,900,700 records
- Can often hold top levels in buffer pool:
  - Level 0 = 1 page = 8 KBytes
  - Level 1 = 133 pages = 1 MByte
  - Level 2 = 17,689 pages = 133 MBytes

(3) Searches in a B+ Tree



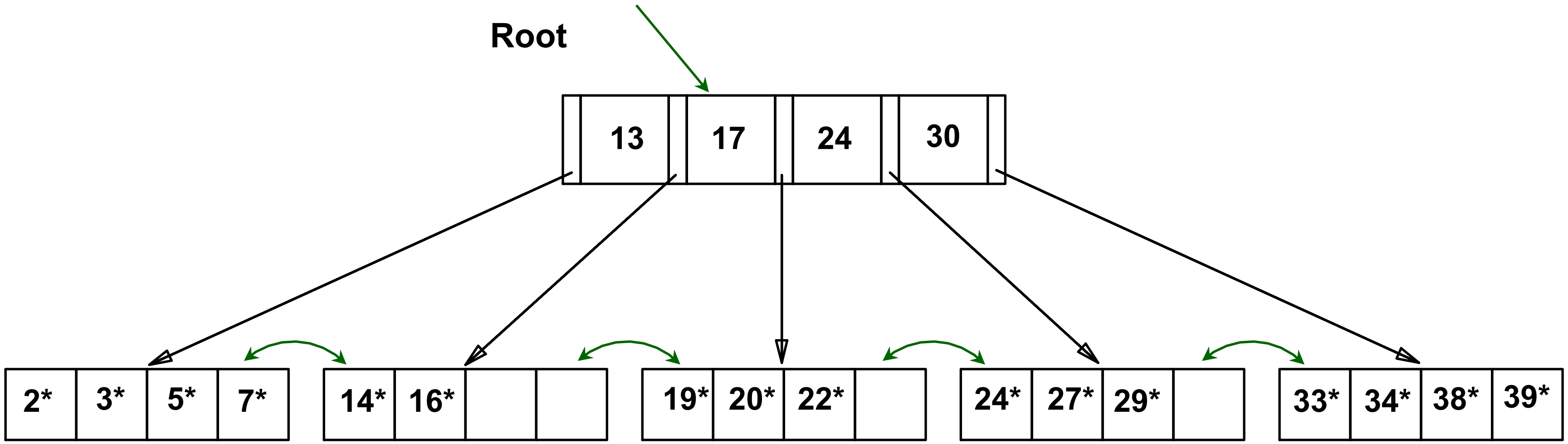
- Search begins at root, key comparisons direct it to a leaf.
- *Equality selection*: find 28*? 29*?
- *Range selection*: find all > 15* and < 30*

(4) Inserting a Data Entry into a B+ Tree

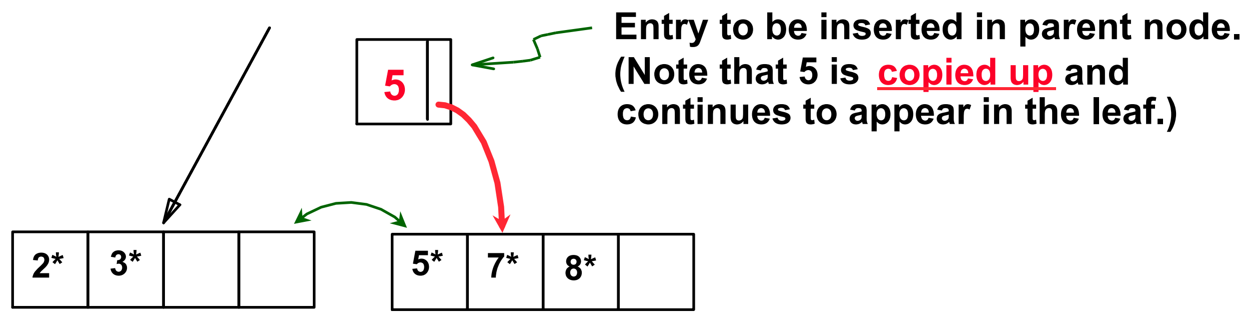
- Find correct leaf *L* via a top-down search.
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, ***copy up*** middle key *k*, insert (*k*, pointer to *L2)* into parent of *L*.
  - Splitting can happen recursively to non-leaf nodes
    - Redistribute entries evenly, but ***push up*** middle key. (Contrast with leaf splits.)
- Splits "grow" the tree!
  - First *wider*, then *one level taller* when the root splits.

Previous Example

Inserting 8*

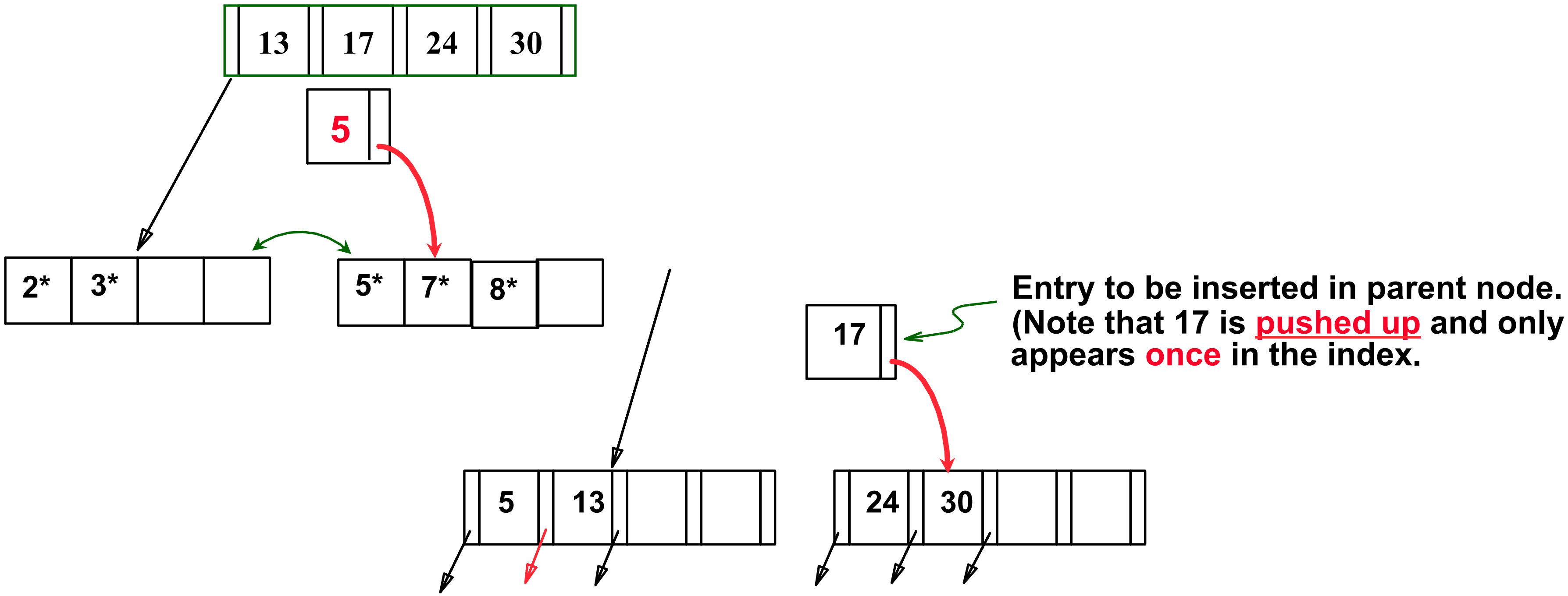


Inserting 8 into Example B+ Tree*



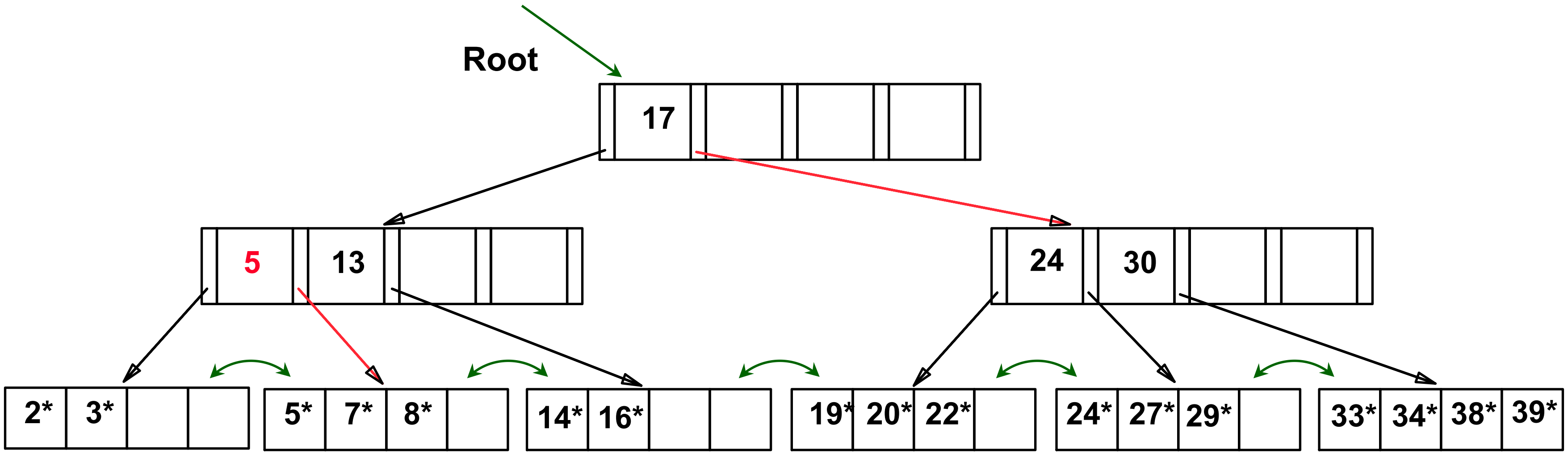
- Minimum occupancy is guaranteed in node splits.
- Copy up: key value of an inserted entry **must appear in a leaf node**!

Inserting 8 into Example B+ Tree*



- Note difference between *copy-up* and *push-up*. Reasons?
- Push up: Any key value can appear **at most once in non-leaf nodes**!

*Example B+ Tree After Inserting 8**



❖ Root was split, leading to increase in height!

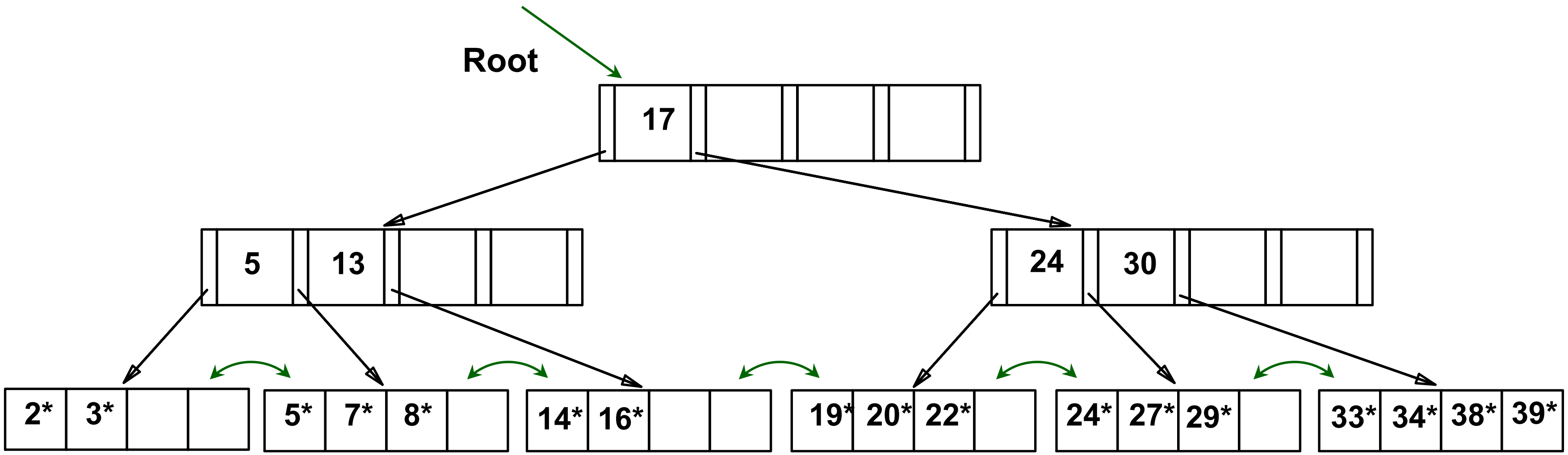
(5) Deleting a Data Entry from a B+ Tree

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only $\lfloor \mathbf{n/2} \rfloor$ **- 1** entries,
    - Try to *re-distribute*, borrowing from *sibling* *(adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *L* and sibling. Must delete index entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

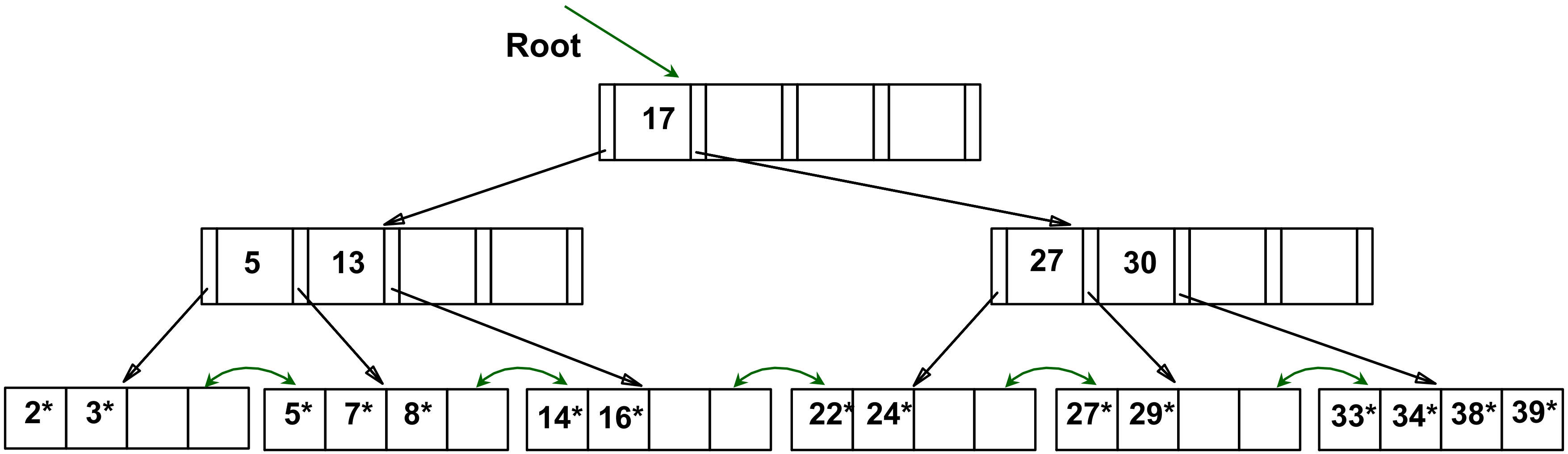
Current B+ Tree

Delete 19*

Delete 20*



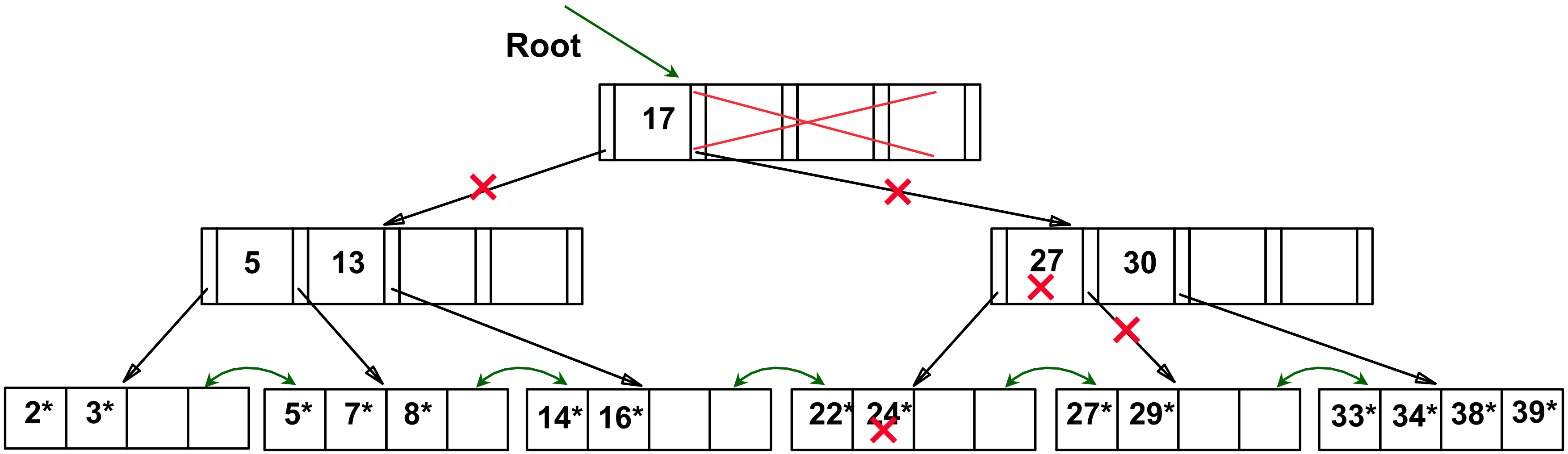
Example Tree After Deleting 19, 20*...*



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

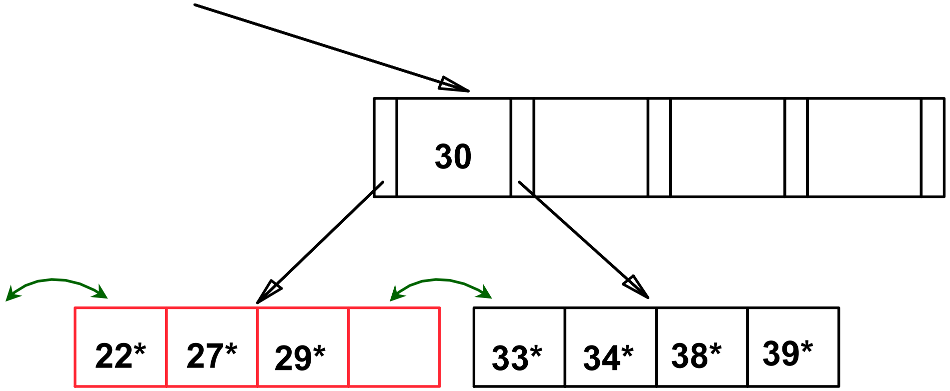
New B+ Tree ...

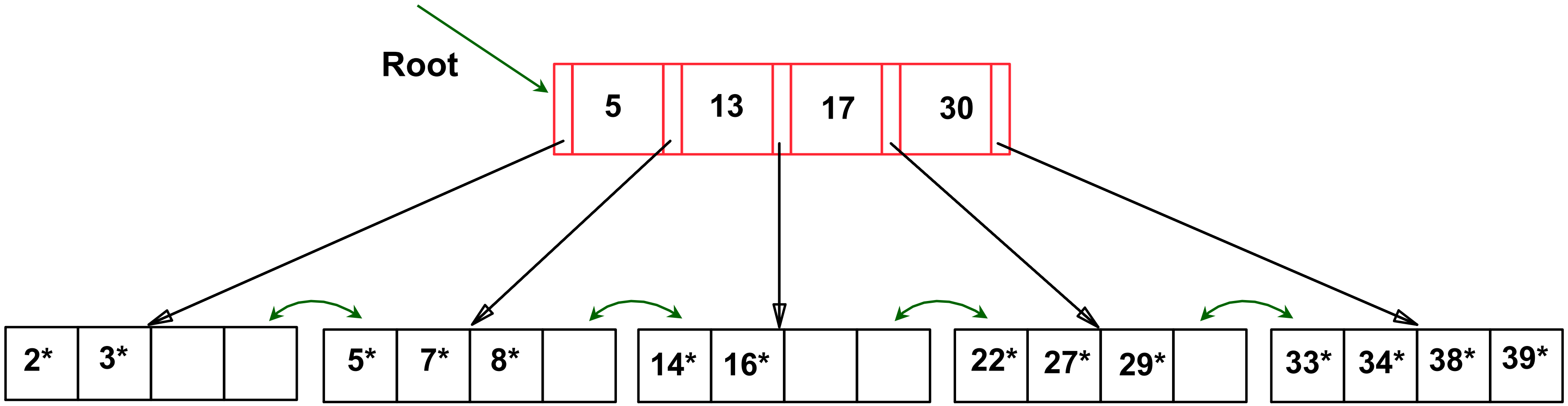
Delete 24*



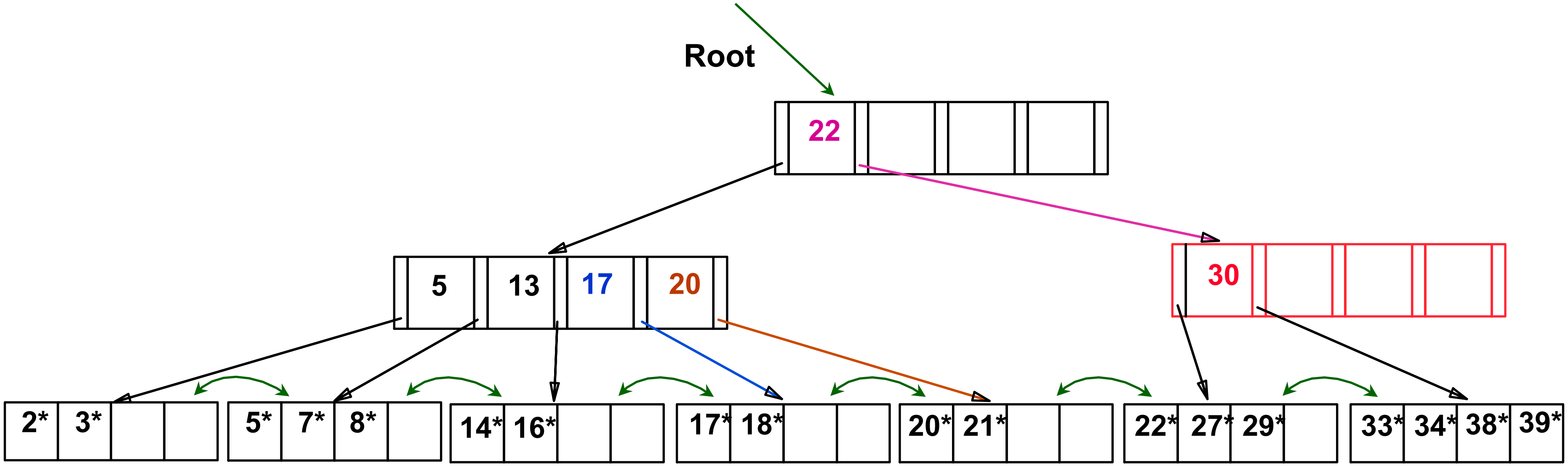
... And Then Deleting 24*

- Must merge nodes.
- <u>*Toss*</u> index entry (right)
- <u>*Pull down*</u> of index entry (below).



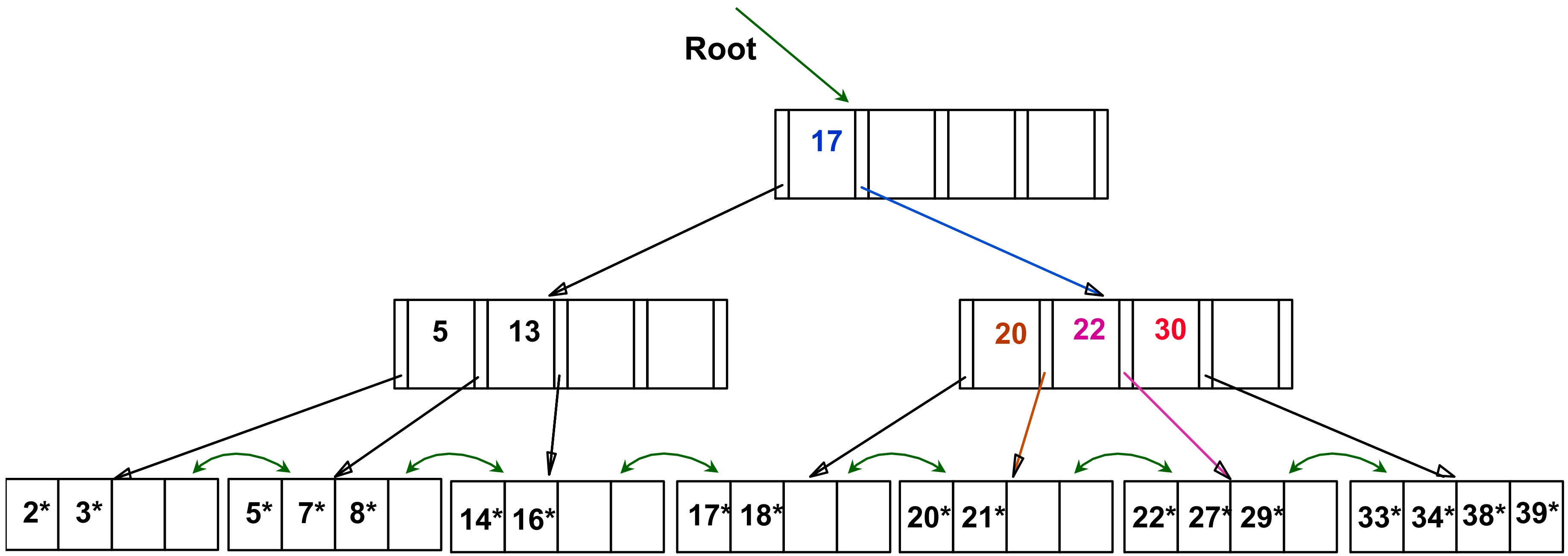


Example of Non-leaf Re-distribution



- 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.
- It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.

Outline

- Disks, Disk Space Manager
- Disk-Resident Data Structures
  - Files of records
  - Indexes
    - Tree indexes: B+ tree
    - Hash indexes

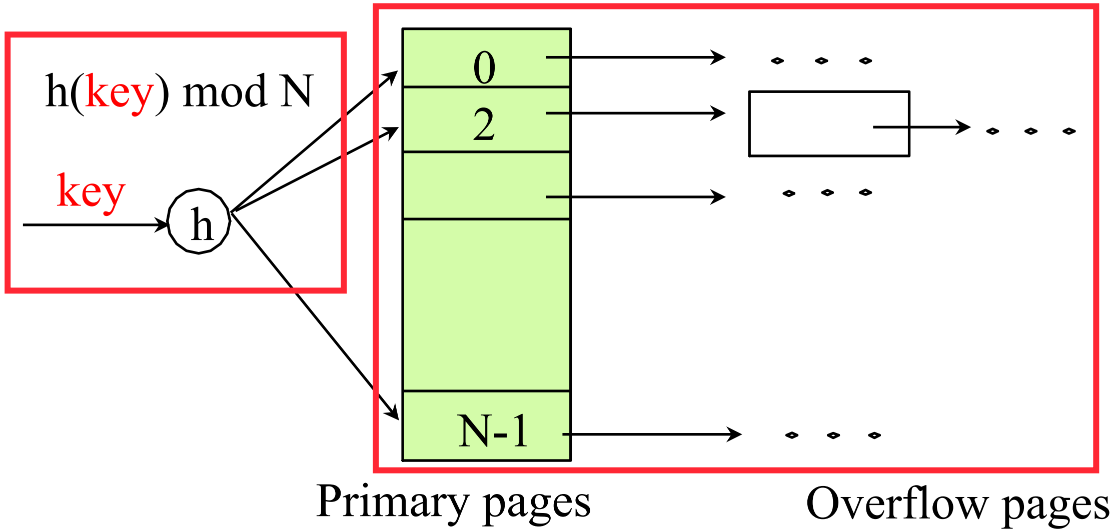
Hash Indexes

- *Hash-based* indexes are best for *equality selections*. ***Cannot*** support range searches.
  - E.g., retrieve a student with id '123' or all students at age=20.

- Static and dynamic hashing techniques exist.
  - Trade-offs similar to ISAM vs. B+ trees.

- As for any index, 3 alternatives for data entries **k***:
  - <**k**, data record with key value **k**>
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>

Static Hashing



- **h**(*k*) mod N = bucket to which data entry with key *k* belongs. k1≠k2 can lead to the same bucket.
- Static structure: # buckets (N) fixed
  - *Primary pages*: allocated sequentially, never de-allocated;
  - *Overflow pages*: allocated/de-allocated if needed.

Static Hashing (Contd.)

- Hash fn on the *search key* distributes values over [0 ... N-1].
  - **h**(*key*) mod N = (a * *key* + b) mod N
  - a and b are constants; a lot is known about how to tune **h**

- Buckets contain data entries in a chain of pages.
  - Long overflow chains degrade performance.
  - Dynamic techniques fix this problem.

Extendible Hashing

- When bucket (primary page) becomes full, why not re-organize file by *doubling* num. of buckets?
  - Reading and writing all pages is expensive!

- **Idea**: use a *directory of buckets*. When bucket is full:

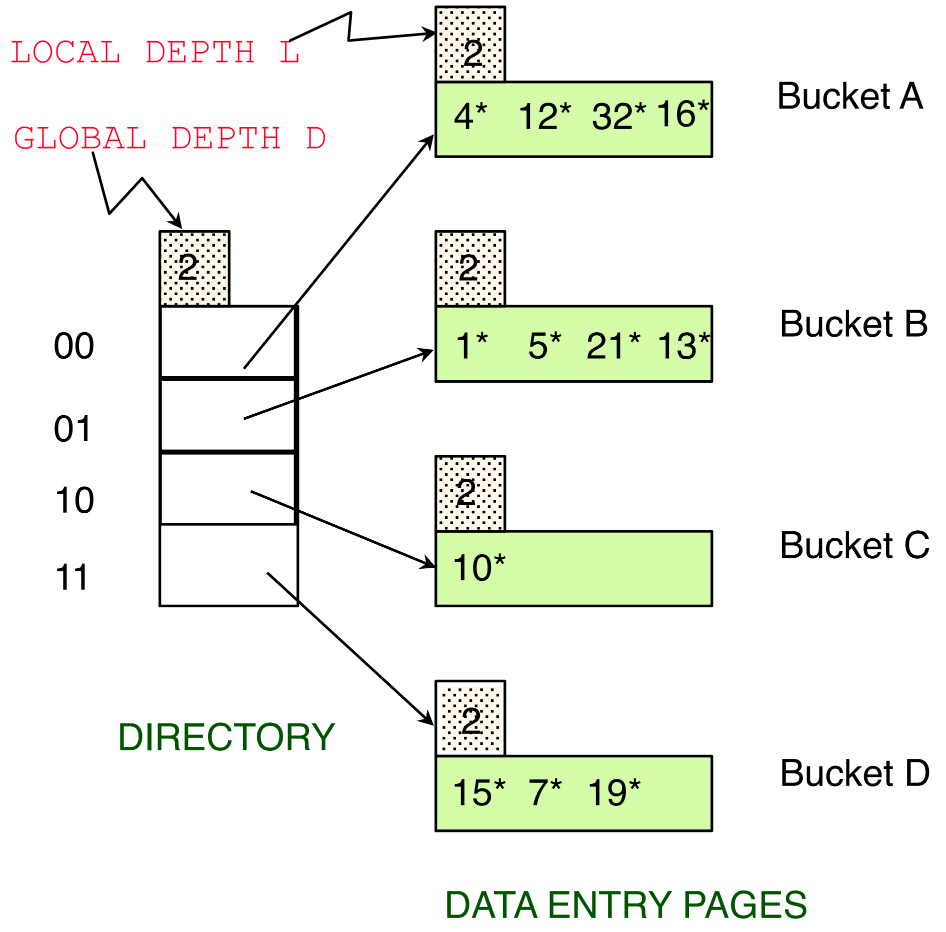
  1) *double the directory,*

  2) *split just the bucket that overflowed.*

  - Directory much smaller than file, so doubling is cheap.
  - Only one page of data entries is split. *No overflow page*!
  - Trick lies in how hash function is adjusted.

Example

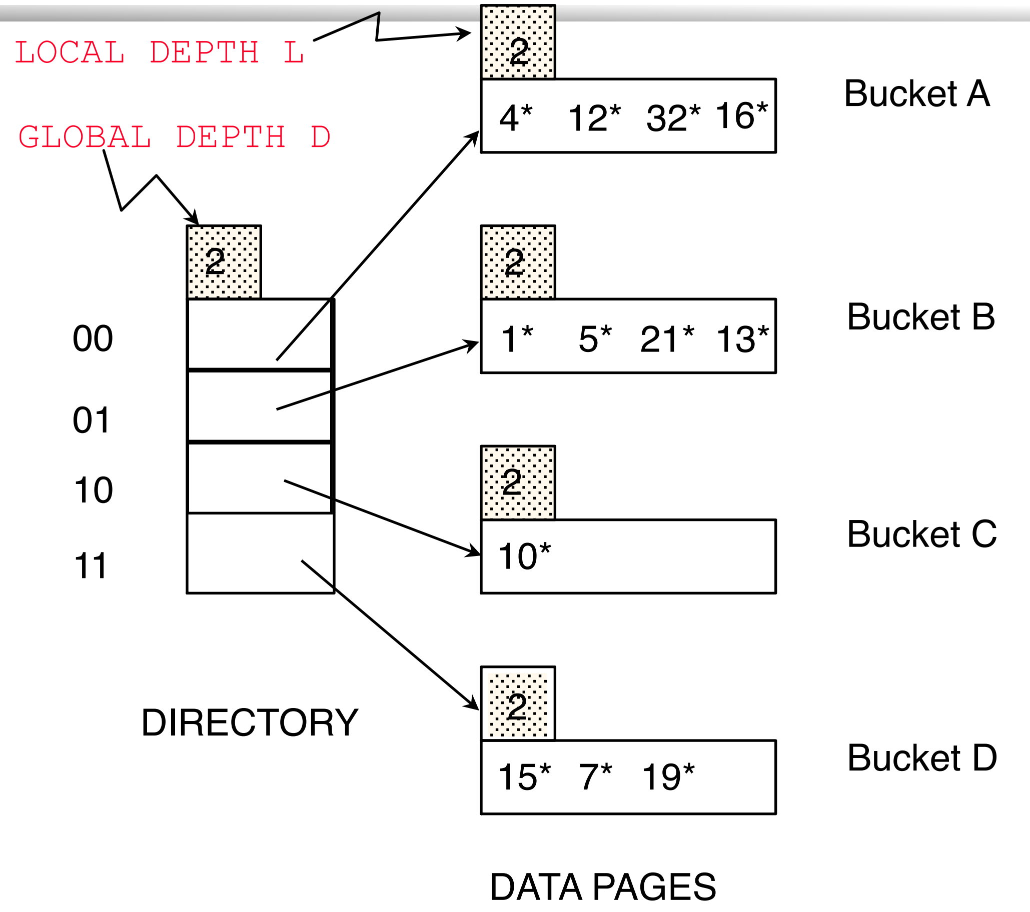
- Directory is array of size N=4, *global depth* D = 2.
- To find bucket for *key*:
  1) get **h**(*key*),
  2) take last \`*global depth*' # bits of **h**(*key*), i.e., mod 2^D.
     - If **h**(*key*) = 5 = binary 101,
     - Take last 2 bits, go to bucket pointed to by 01.
- Each bucket has *local depth* L ($L \leq D$) for splits!



Inserts

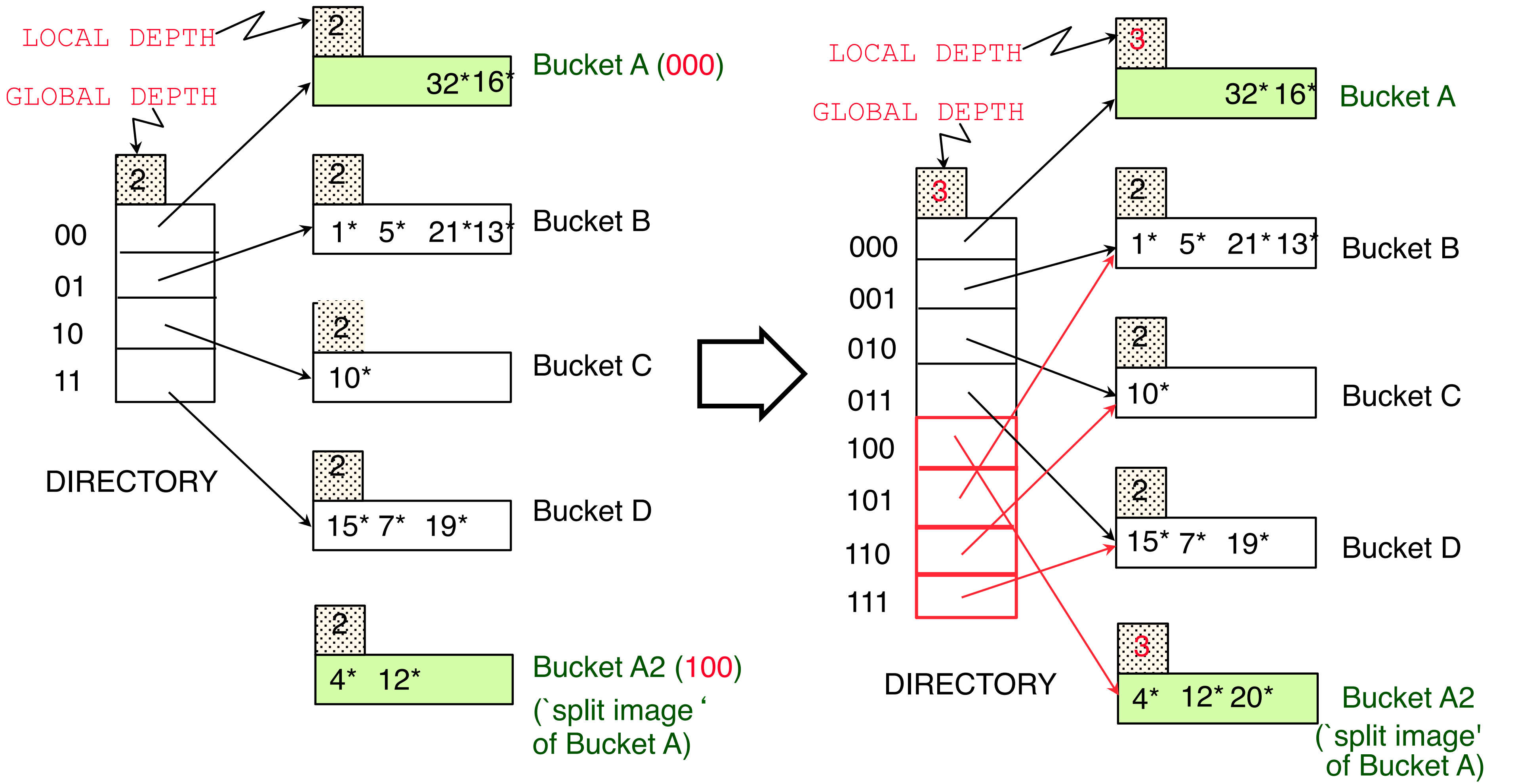
❖ If bucket is full, *split* it:
- *Allocate new page,*
- *Re-distribute,*
- If needed, *double* directory.

❖ Double the directory if *global depth* D = *local depth* L
- Split if D = L.
- Otherwise, don't.



Insert *k** with h(*k*)=20?

*Insert **h**(k)=20 (Causes Doubling)*



Points to Note

- 20 = binary 10100. Last **3** bits needed to distinguish A, A2.
  - *Global depth D of directory*: Max # of bits needed to tell which bucket an entry belongs to.
  - *Local depth L of a bucket*: Actual # of bits needed to determine if an entry belongs to this bucket.

- Bucket split causes directory doubling if before insertion, *L* of bucket = *D of directory*.

Deletes

- Remove a data entry from bucket
  - If bucket is empty, can be merged with `split image'.
  - If each directory entry points to same bucket as its split image, can halve directory.
  - If assume more inserts than deletes, do nothing…

Comments on Extendible Hashing

- *Access cost*: If directory fits in memory, equality search takes one I/O to access the bucket; else two.
- *Skews*: If the distribution of *hash values* is skewed, directory can grow large. An example?
- *Duplicates*: Entries with *same key value* need overflow pages!