CompSci 516

Data Intensive Computing Systems

Lecture 6

Storage and Indexing

Instructor: Sudeepa Roy

What will we learn?

- Last lecture:
  - Overview of indexing and storage
- Next:
  - Storage on disk and memory
    - Record format, Page format
  - Tree-based indexing:
    - B+ tree
    - insert, delete
  - Hash-based indexing
    - Static and dynamic (extendible hashing, linear hashing)

Reading Material

- [RG]
  - Storage: Chapter 9.4-9.7
  - Tree-based index: Chapter 10.1-10.7
  - Hash-based index: Chapter 11 (to be continued)
- [GUW]
  - Chapter 14.1 – 14.4

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Storage

Disk Space Management

- Lowest layer of DBMS software manages space on disk
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Size of a page = size of a disk block
  = data unit
- Request for a sequence of pages often satisfied by allocating contiguous blocks on disk
  - Managed by Disk-space Manager
  - Higher levels don't need to know how this is done, or how free space is managed

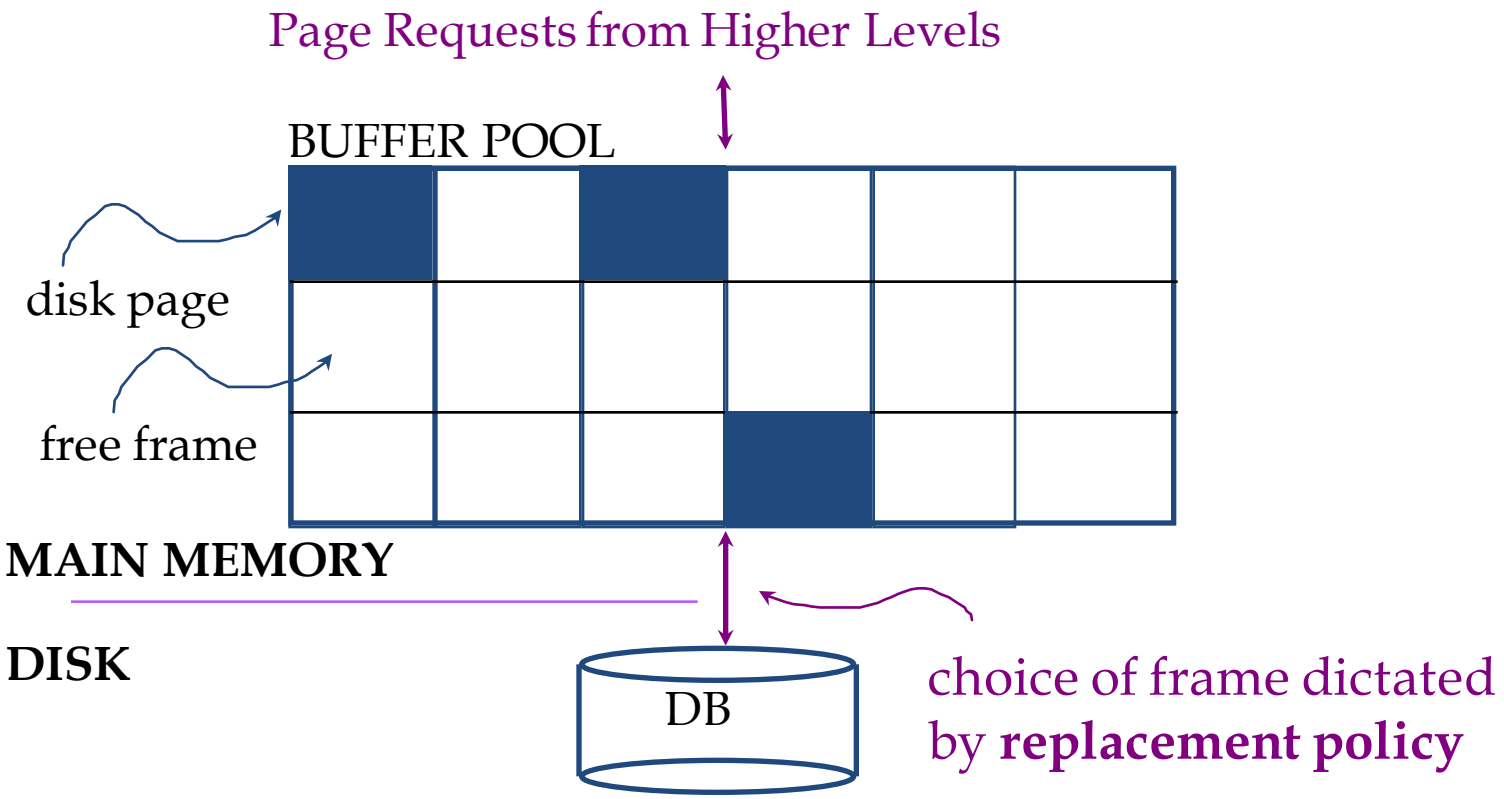
Buffer Management

Suppose

- 1 million pages in db, but only space for 1000 in memory
- A query needs to scan the entire file
- DBMS has to
  - bring pages into main memory
  - decide which existing pages to replace to make room for a new page
  - called Replacement Policy
- Managed by the Buffer manager

Buffer Management

Buffer pool = main memory is partitioned into frames
either contains a page from disk or is a free frame



- Data must be in RAM for DBMS to operate on it
- Table of <frame#, pageid> pairs is maintained

When a Page is Requested ...

For every page, store

- a dirty bit:
  - whether the page has been modified since it has been brought to memory
  - initially 0 or off

- a pin-count:
  - the number of times a page has been requested but not released
  - initially 0
  - when a page is requested, the count in incremented
  - when the requestor releases the page, count is decremented
  - buffer manager only reads a page into a frame when its pin-count is 0
  - if no page with pin-count 0, buffer manager has to wait (or a transaction is aborted -- later)

When a Page is Requested ...

- Check if the page is already in the buffer pool
- if yes, increment the pin-count of that frame
- If no,
  - Choose a frame for replacement using the replacement policy
  - If the chosen frame is dirty (has been modified), write it to disk
  - Read requested page into chosen frame
- Pin the page and return its address to the requestor

- If requests can be predicted (e.g., sequential scans), pages can be pre-fetched several pages at a time
- CC & recovery may entail additional I/O when a frame is chosen for replacement
  - Write-Ahead Log protocol : when we do Transactions

Buffer Replacement Policy

- Frame is chosen for replacement by a replacement policy

- Least-recently-used (LRU)
  - add pages with pin-count 0 to the end of a queue
- Clock
  - an efficient implementation of LRU
- First In First Out (FIFO)
- Most-Recently-Used (MRU) etc.

Buffer Replacement Policy

- Policy can have big impact on # of I/O's
- Depends on the access pattern
- Sequential flooding: Nasty situation caused by LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request in each scan causes an I/O (assume 10 frames and 11 pages in the file)
  - MRU much better in this situation (but not in all situations, of course)

DBMS vs. OS File System

- OS does disk space and buffer management:
- Why not let OS manage these tasks?

- DBMS can predict the page reference patterns much more accurately
  - can optimize
  - adjust replacement policy
  - pre-fetch pages – already in buffer + contiguous allocation
  - pin a page in buffer pool, force a page to disk (important for implementing Transactions concurrency control & recovery)

- Differences in OS support: portability issues

- Some limitations, e.g., files can't span disks

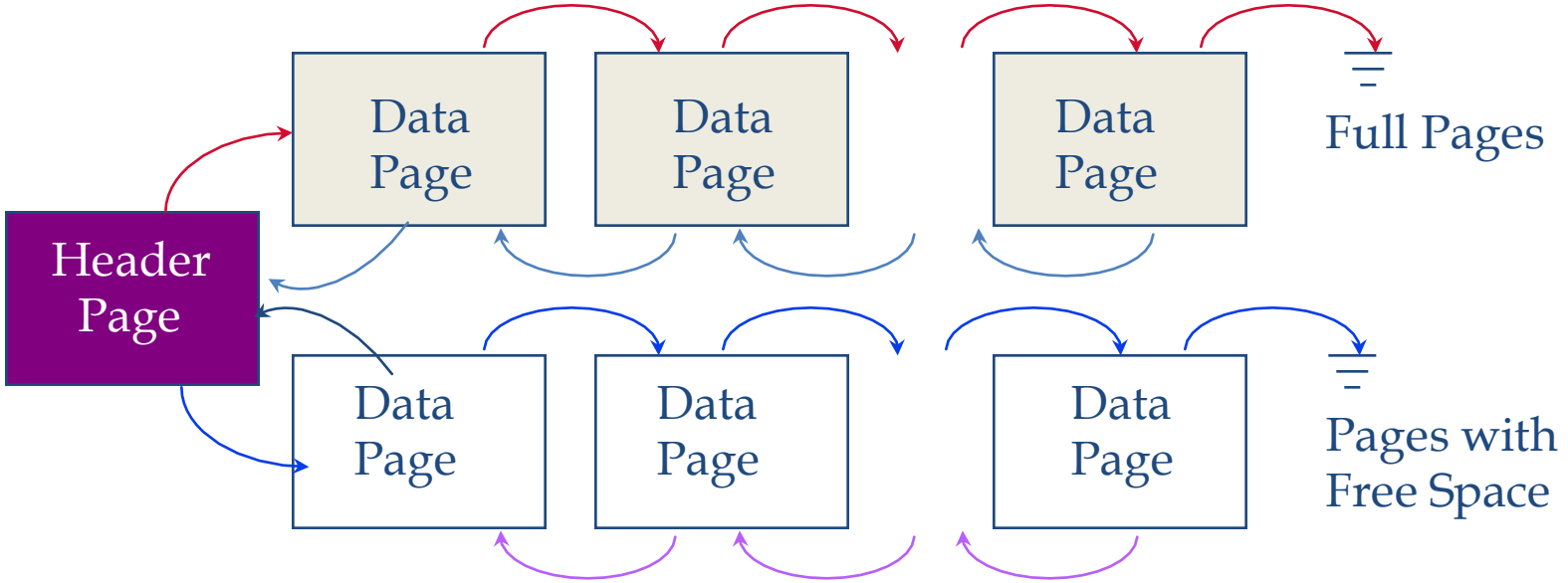
Files of Records

- Page or block is OK when doing I/O, but higher levels of DBMS operate on records, and files of records.
- FILE: A collection of pages, each containing a collection of records
- Must support:
  - insert/delete/modify record
  - read a particular record (specified using record id)
  - scan all records (possibly with some conditions on the records to be retrieved)

Unordered (Heap) Files

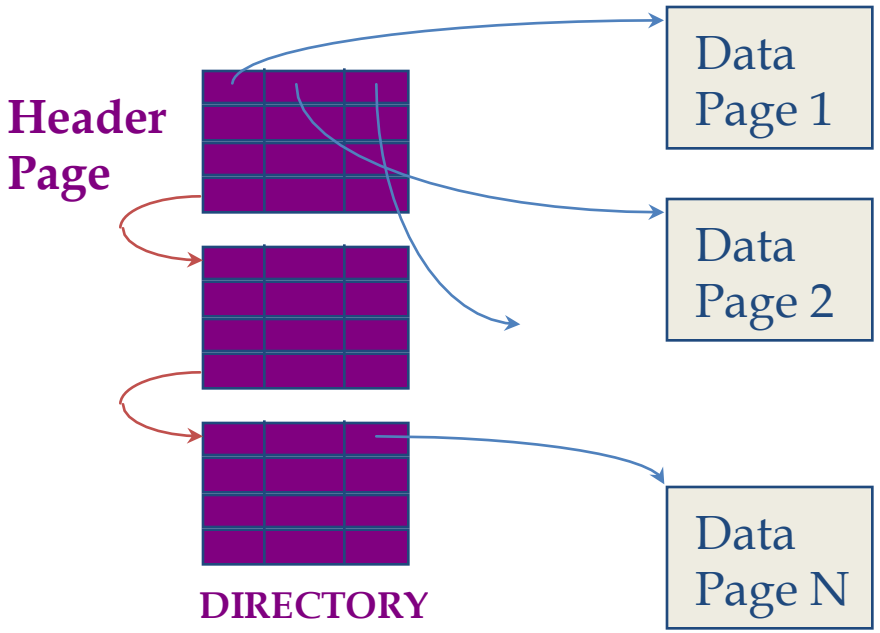
- Simplest file structure contains records in no particular order.
- As file grows and shrinks, disk pages are allocated and de-allocated.
- To support record level operations, we must:
  - keep track of the pages in a file
  - keep track of free space on pages
  - keep track of the *records* on a page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

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
- Problem:
  - to insert a new record, we may need to scan several pages on the free list to find one with sufficient space

Heap File Using a Page Directory

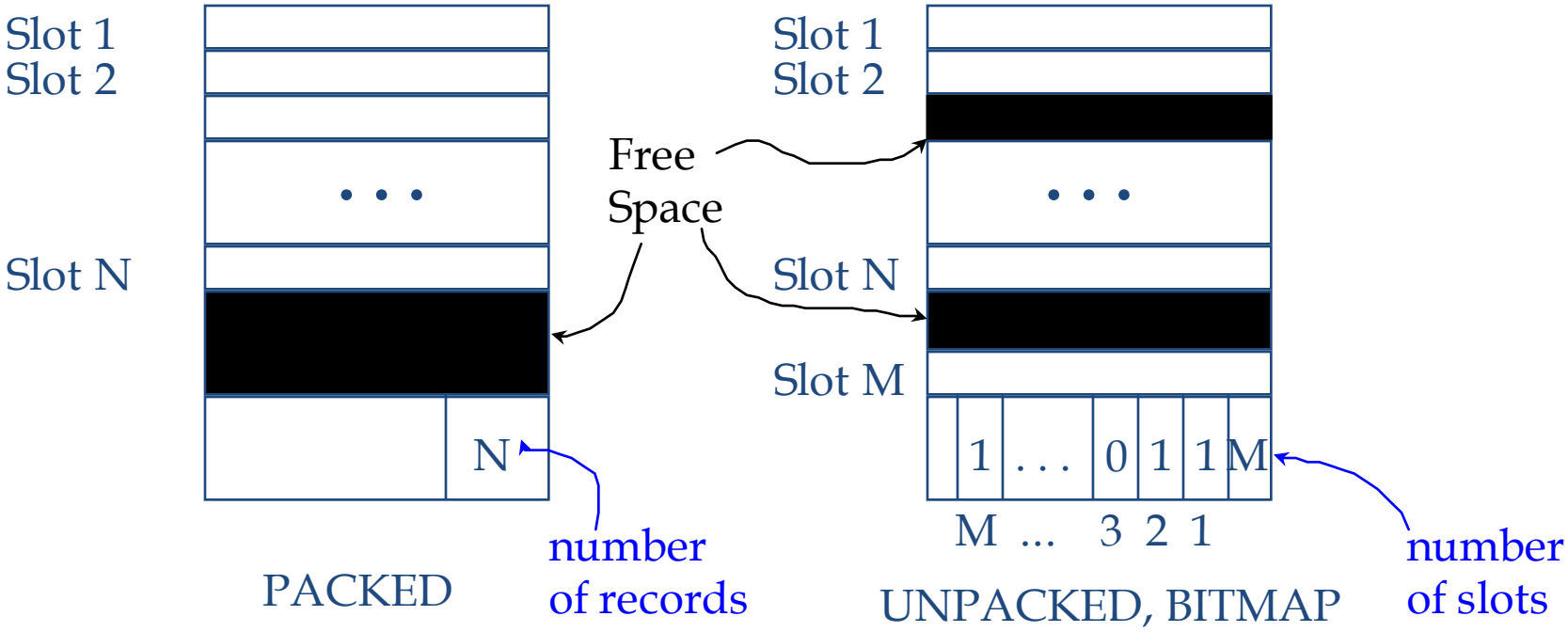


- The entry for a page can include the number of free bytes on the page.
- The directory is a collection of pages; linked list implementation is just one alternative.
  - Much smaller than linked list of all heap file pages!

How do we arrange a collection of records on a page?

- Each page contains several slots
  - one for each record
- Record is identified by <page-id, slot-number>
- Fixed-Length Records
- Variable-Length Records
- For both, there are options for
  - Record formats (how to organize the fields within a record)
  - Page formats (how to organize the records within a page)

Page Formats: Fixed Length Records

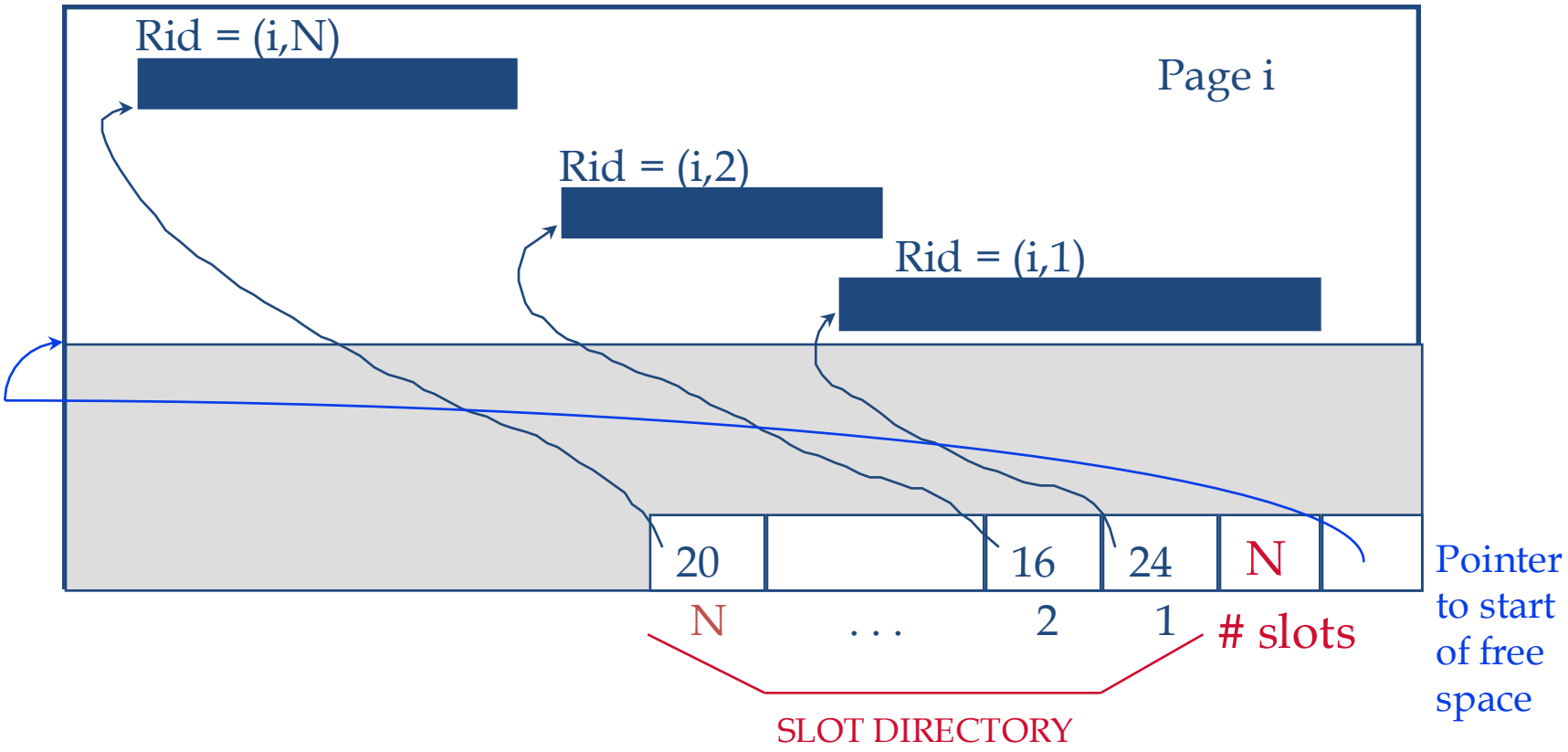


- Record id = <page id, slot #>
- Packed: moving records for free space management changes rid; may not be acceptable
- Unpacked: use a bitmap – scan the bit array to find an empty slot
- Each page also may contan additional info like the id of the next page (not shown)

Page Formats: Variable Length Records

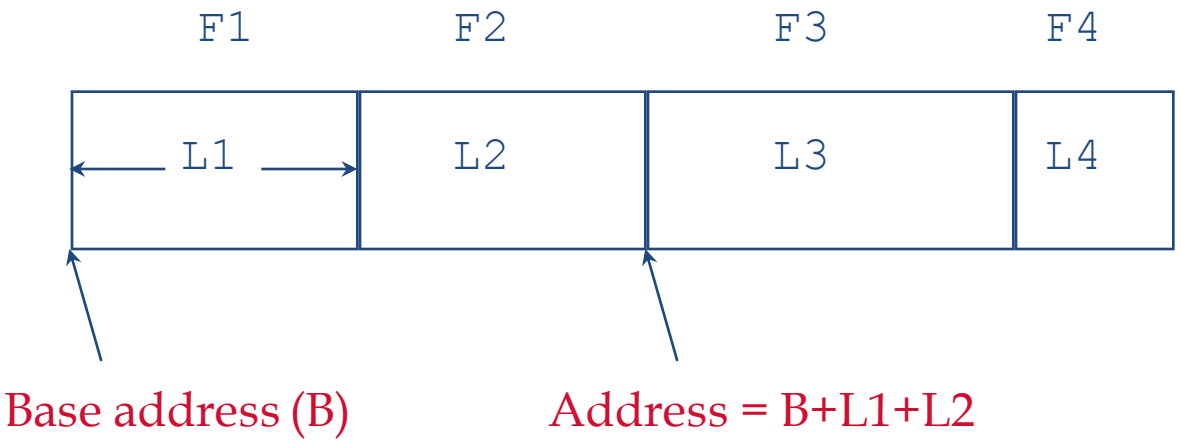
- Need to find a page with the right amount of space
  - Too small – cannot insert
  - Too large – waste of space
- if a record is deleted, need to move the records so that all free space is contiguous
  - need ability to move records within a page
- Can maintain a directory of slots (next slide)
  - <record-offset, record-length>
  - deletion = set record-offset to -1
- Record-id rid = <page, slot-in-directory> remains unchanged

Page Formats: Variable Length Records



- Can move records on page without changing rid; so, attractive for fixed-length records too.

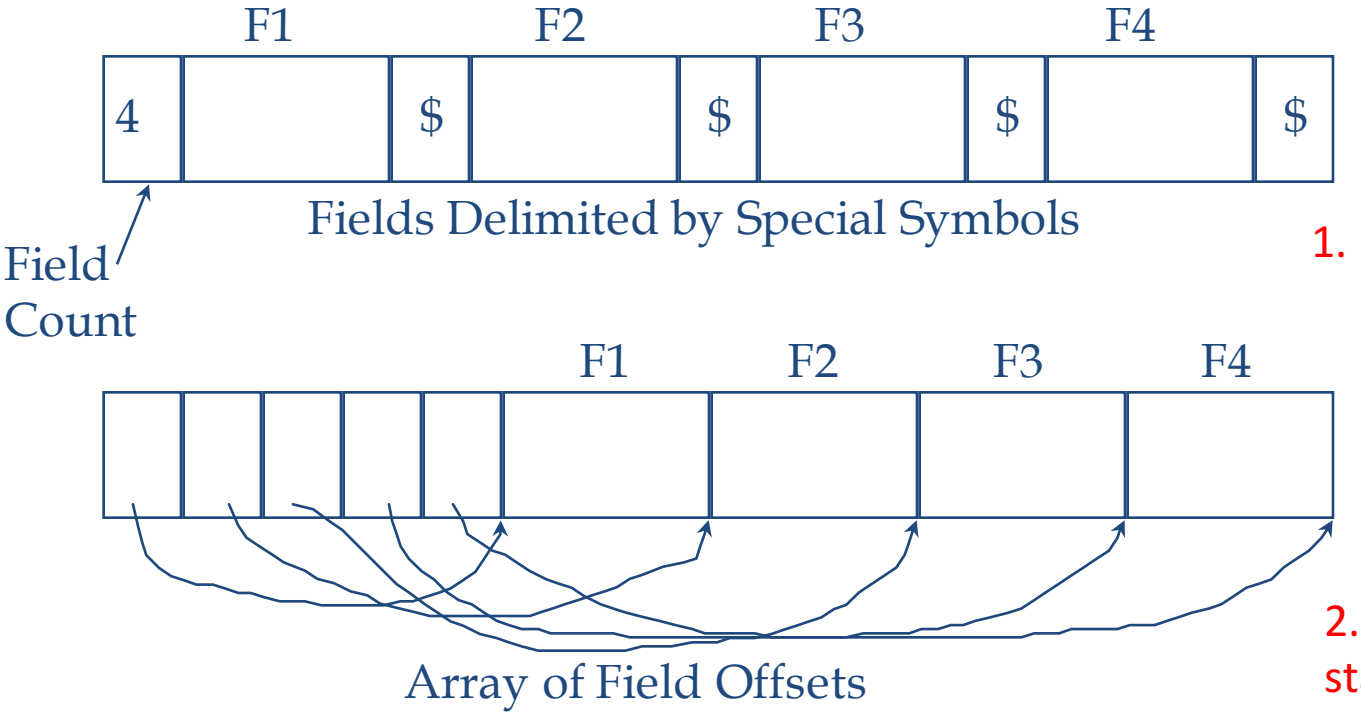
Record Formats: Fixed Length



- Each field has a fixed length
  - for all records
  - the number of fields is also fixed
  - fields can be stored consecutively
- Information about field types same for all records in a file
  - stored in system catalogs.
- Finding i-th field does not require scan of record
  - given the address of the record, address of a field can be obtained easily

Record Formats: Variable Length

- Cannot use fixed-length slots for records
- Two alternative formats (# fields is fixed):



- Second offers direct access to i-th field, efficient storage of nulls (special don't know value); small directory overhead
- Modification may be costly (may grow the field and not fit)

System Catalogs

- For each index:
  - structure (e.g., B+ tree) and search key fields
- For each relation:
  - name, file name, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.
- (described in [RG] 12.1)

Catalogs are themselves stored as relations!

Tree-based Index and B^+-Tree

Recall from the previous lecture….

- For an index,
  - Search key = k
  - Data entry (stored in the index file) = k*
  - Data entry points to the data record with search key k
- Three alternatives for data entries **k***:
  1. The entire data record with key value **k**
  2. <**k**, rid>
  3. <**k**, list-of-rids>
- The above choice is orthogonal to the indexing technique used to locate data entries **k***
  - Tree-based
  - Hash-based

Recall from the previous lecture….

Different types of indexes:

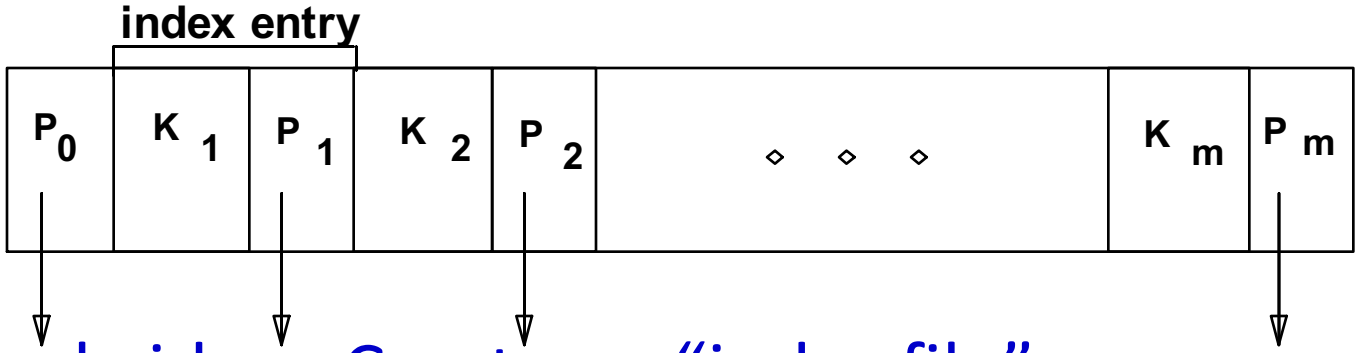
- Clustered vs. unclustered index
  - how data records are organized within a page
- Primary vs. secondary index
  - primary = index on primary key
  - unique index
- Tree- based index supports range searches and equality searches
  - hash-based supports equality searches, but may have less overhead

- Today: how these indexes are implemented and used

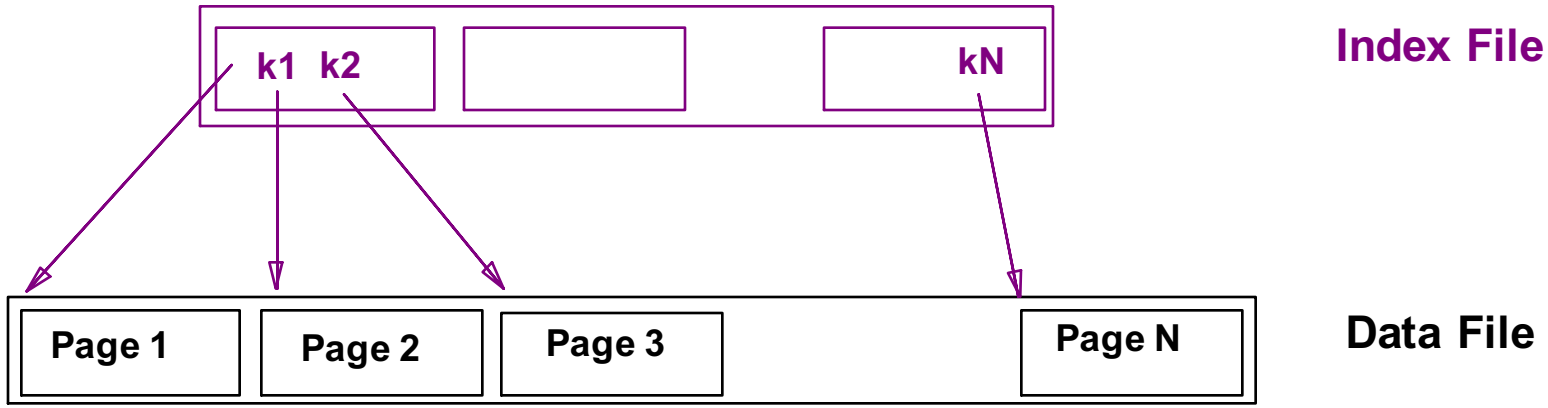
Range Searches

- ``*Find all students with gpa > 3.0*''
  - If data is in sorted file, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.

Index file format



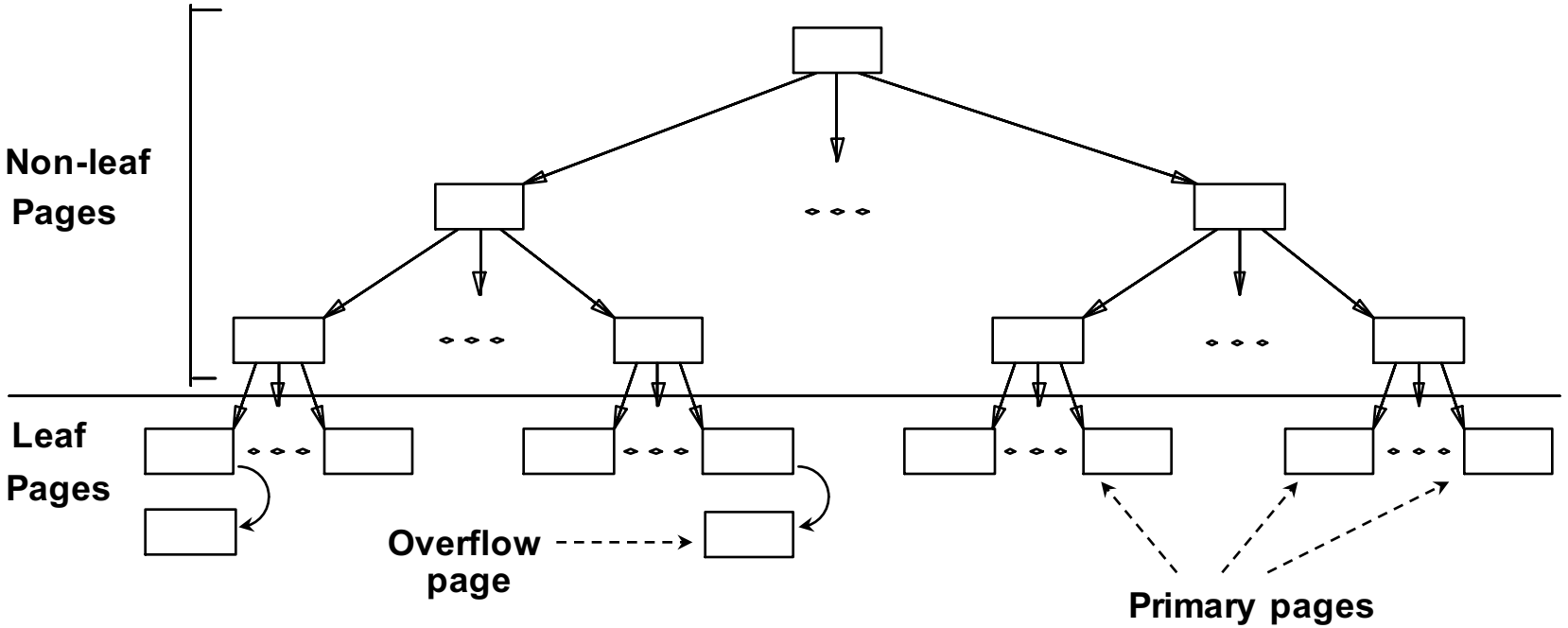
- Simple idea: Create an "index file"
  - <first-key-on-page, pointer-to-page>, sorted on keys



Can do binary search on (smaller) index file
but may still be expensive: apply this idea repeatedly

Indexed Sequential Access Method (ISAM)

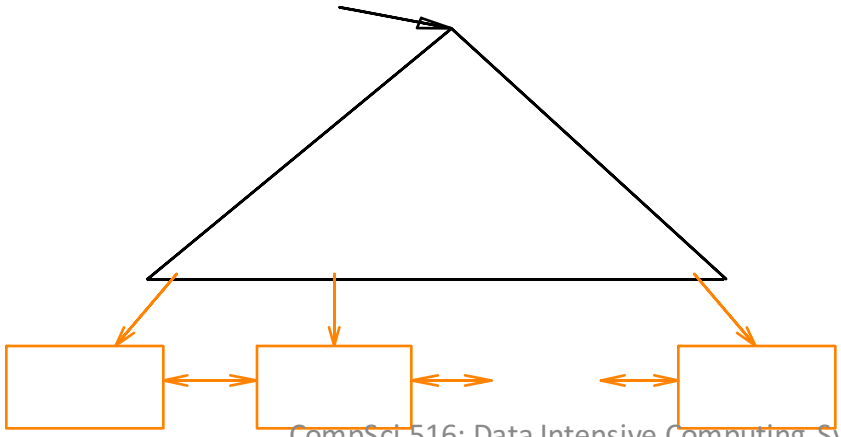
- Leaf-pages contain data entry – also some overflow pages
- DBMS organizes layout of the index – a static structure
- If a number of inserts to the same leaf, a long overflow chain can be created
  - affects the performance



Leaf pages contain data entries.

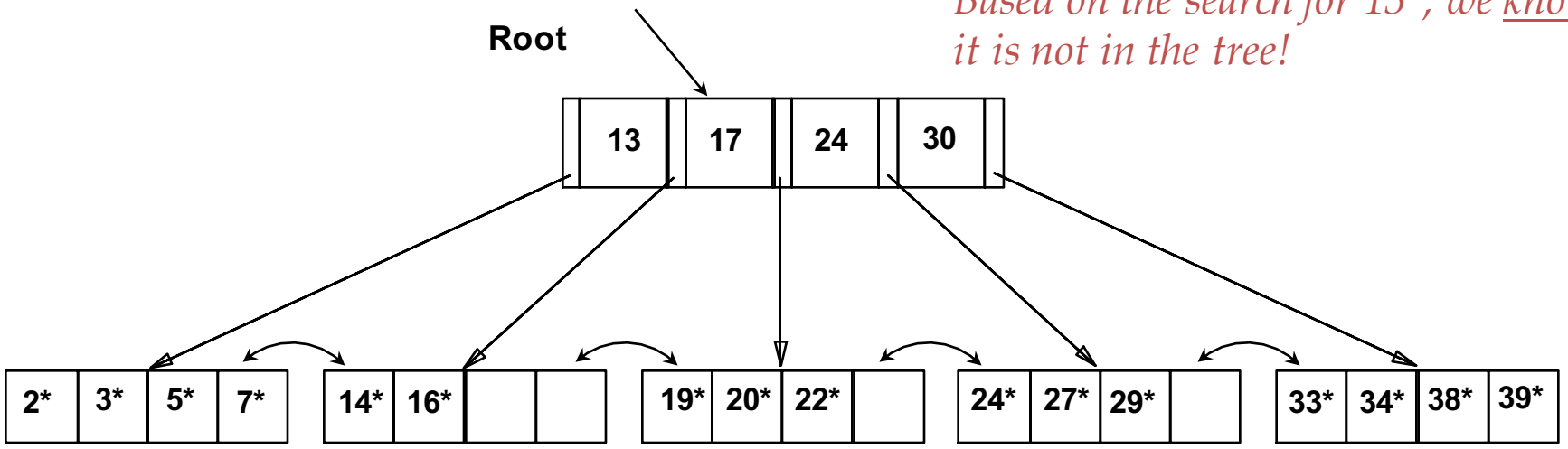
B+ Tree

- Most Widely Used Index
  - a dynamic structure
- Insert/delete at $\log_F N$ cost = height of the tree
  - F = fanout, N = no. of leaf pages
  - tree is maintained height-balanced
- Minimum 50% occupancy
  - Each node contains **d** <= m <= 2**d** entries
  - Root contains 1 <= m <= 2d entries
  - The parameter **d** is called the order of the tree
- Supports equality and range-searches efficiently



Example B+ Tree

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



B+ Trees in Practice

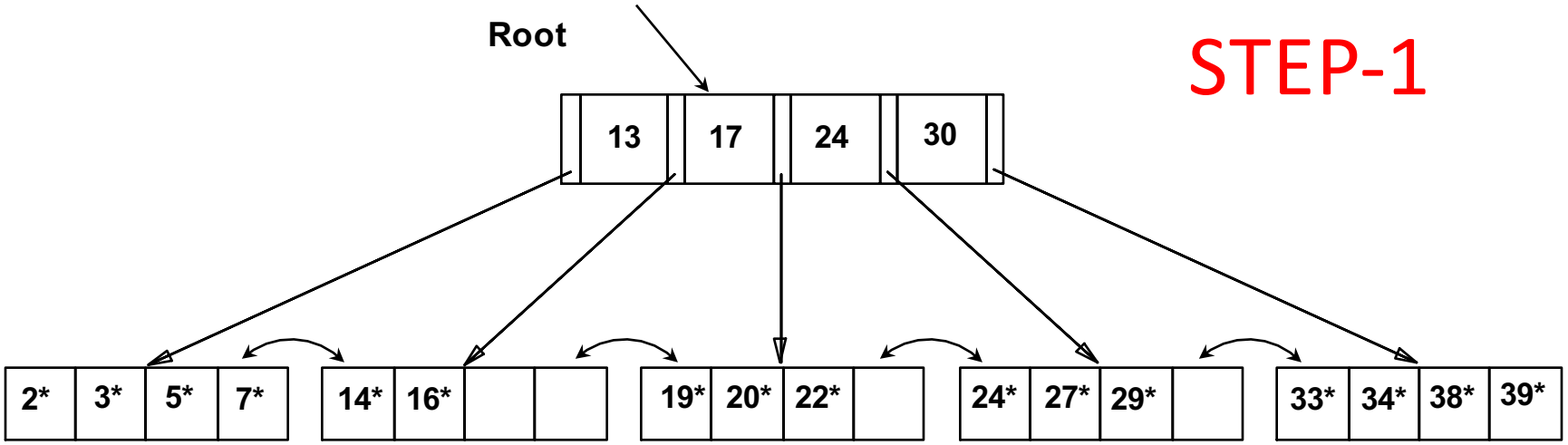
- Typical order: d = 100. Typical fill-factor: 67%.
  - average fanout F = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level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

Inserting a Data Entry into a B+ Tree

See later,
now through examples

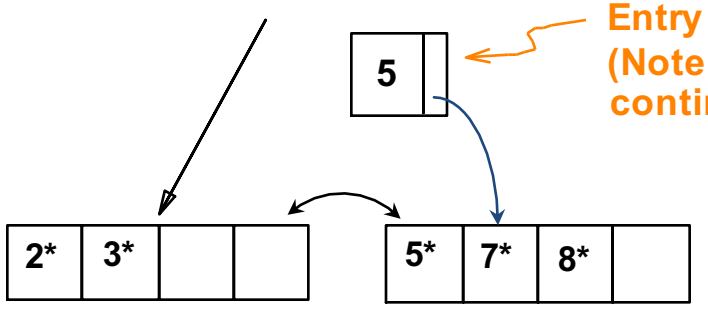
- Find correct leaf L
- Put data entry onto L
  - If L has enough space, done
  - Else, must split L
    - into L and a new node L2
    - Redistribute entries evenly, copy up middle key.
    - Insert index entry pointing to L2 into parent of L.
- This can happen recursively
  - To split index node, redistribute 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 8* into Example B+ Tree



STEP-1

- Copy-up: 5 appears in leaf and the level above
- Observe how minimum occupancy is guaranteed

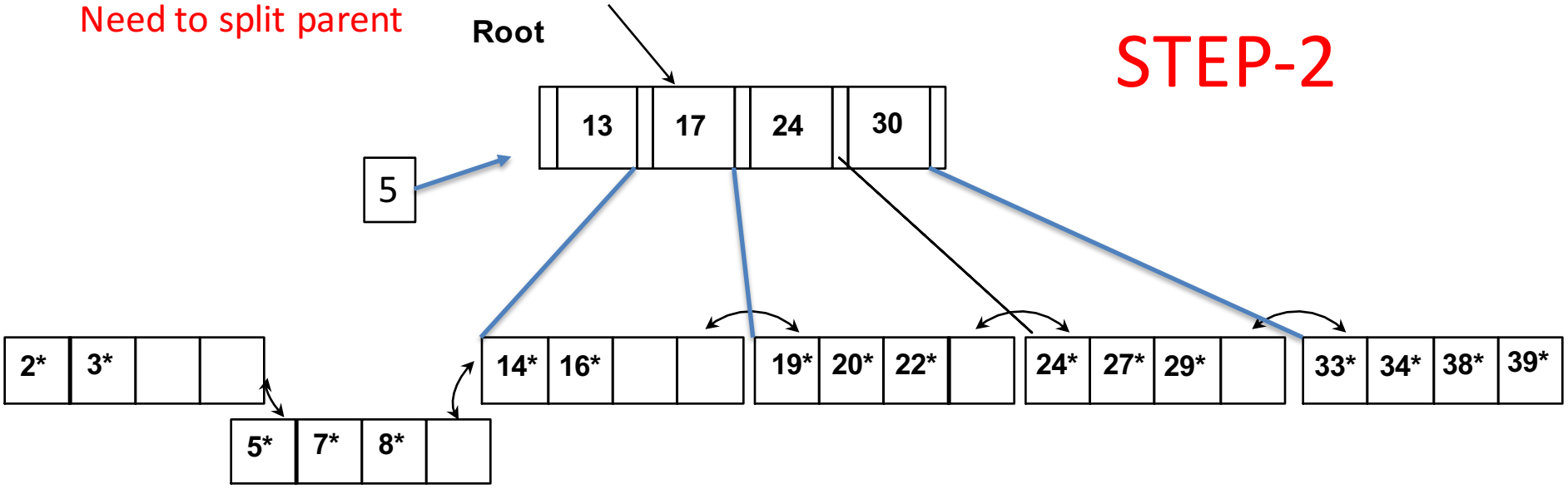


Entry to be inserted in parent node. (Note that 5 is copied up and continues to appear in the leaf.)

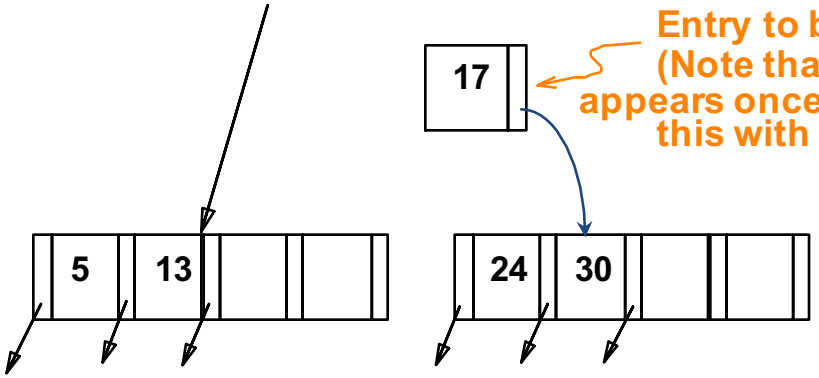
Inserting 8* into Example B+ Tree

Need to split parent

STEP-2

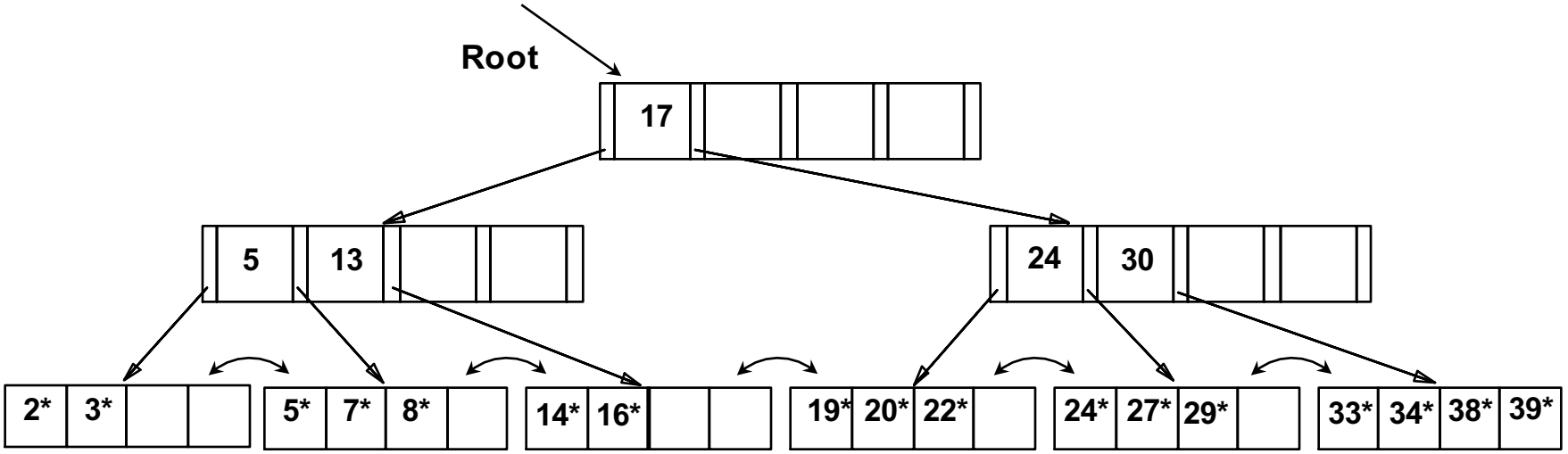


- Note difference between copy-up and push-up
- What is the reason for this difference?
- All data entries must appear as leaves, no such requirement for indexes



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 split by re-distributing entries; however, this is usually not done in practice.

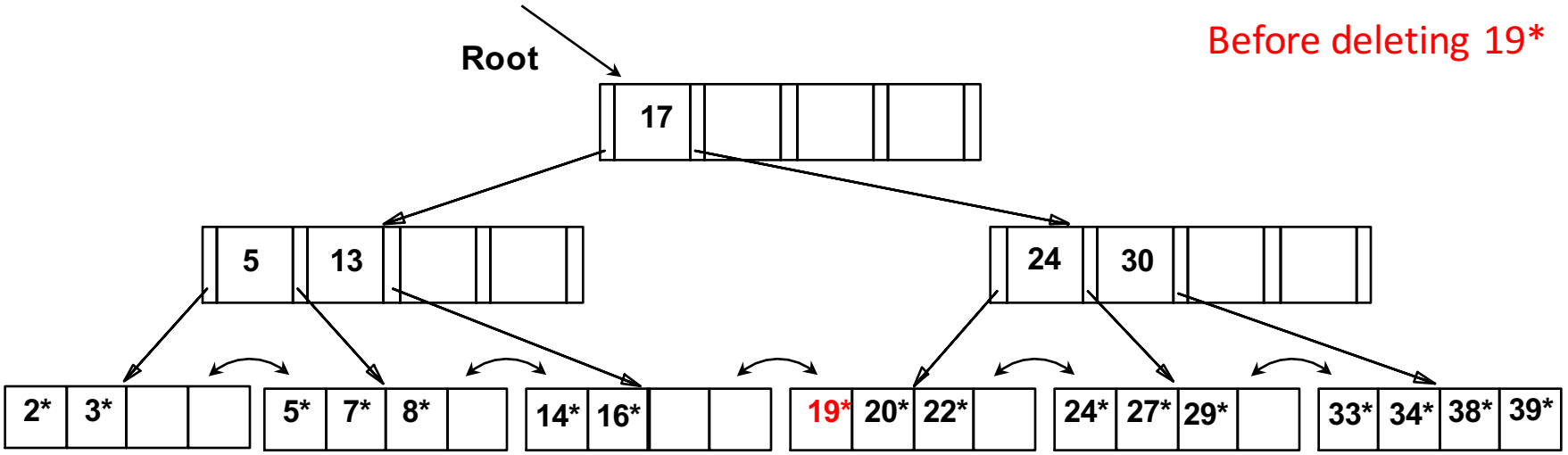
Deleting a Data Entry from a B+ Tree

Each non-root node contains **d** <= m <= 2**d** entries

- 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.

See later,
now through examples

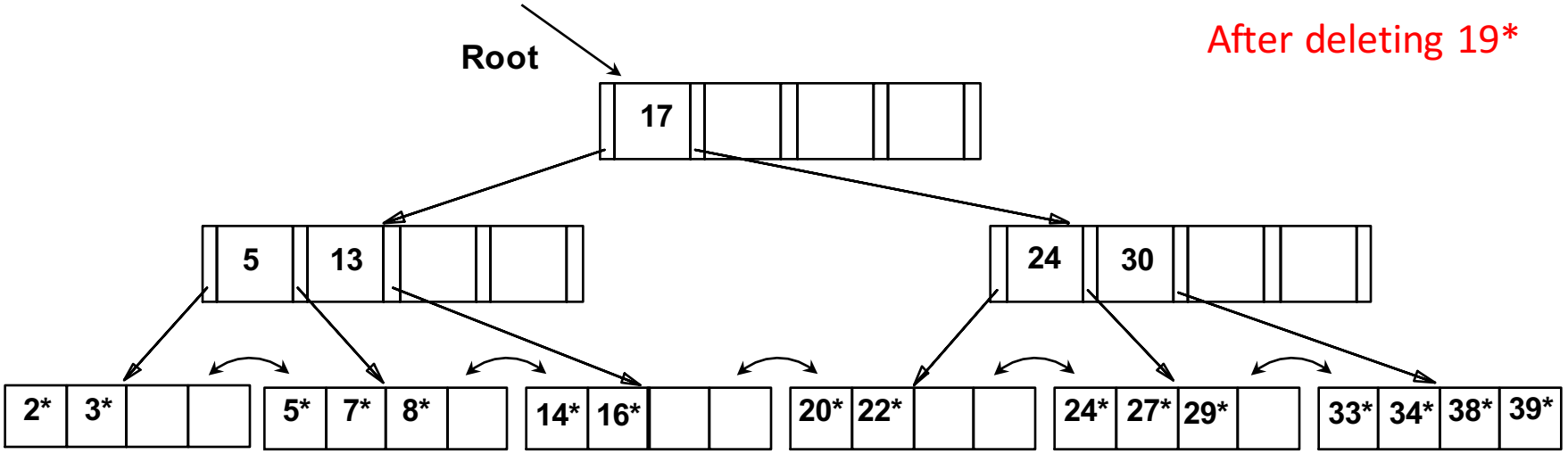
Example Tree: Delete 19*



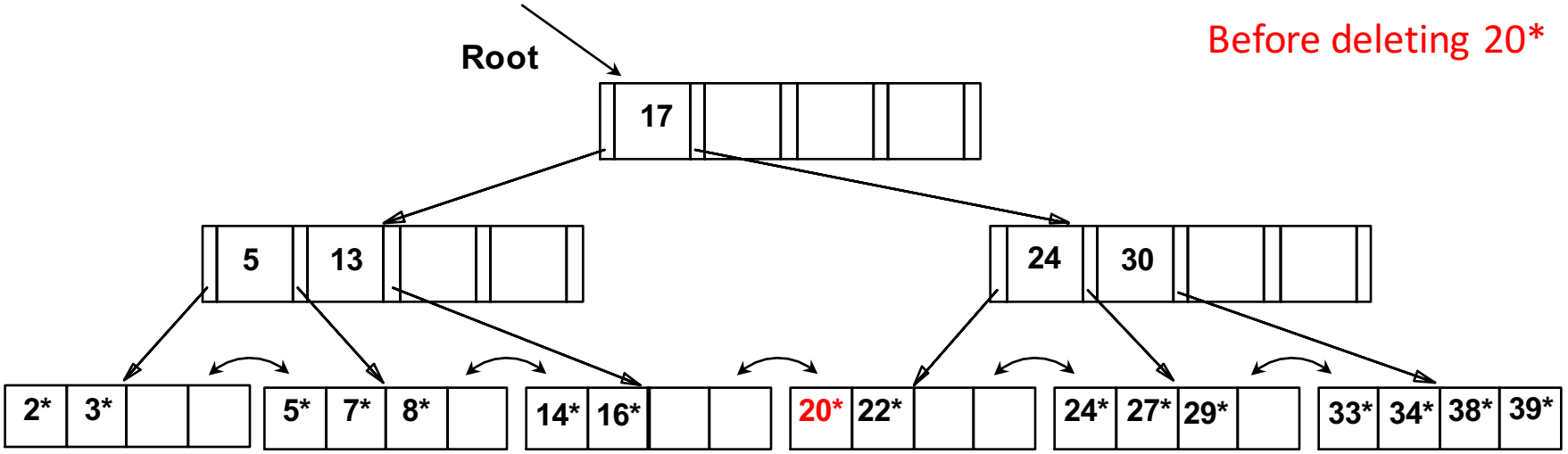
Before deleting 19*

- We had inserted 8*
- Now delete 19*
- Easy

Example Tree: Delete 19*

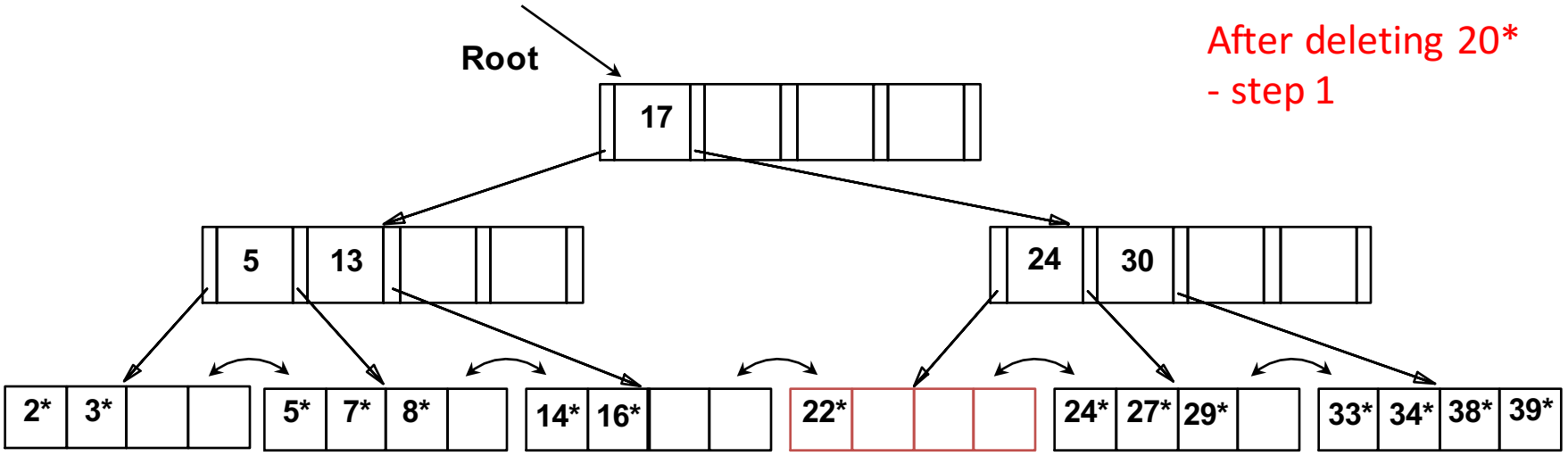


Example Tree: Delete 20*



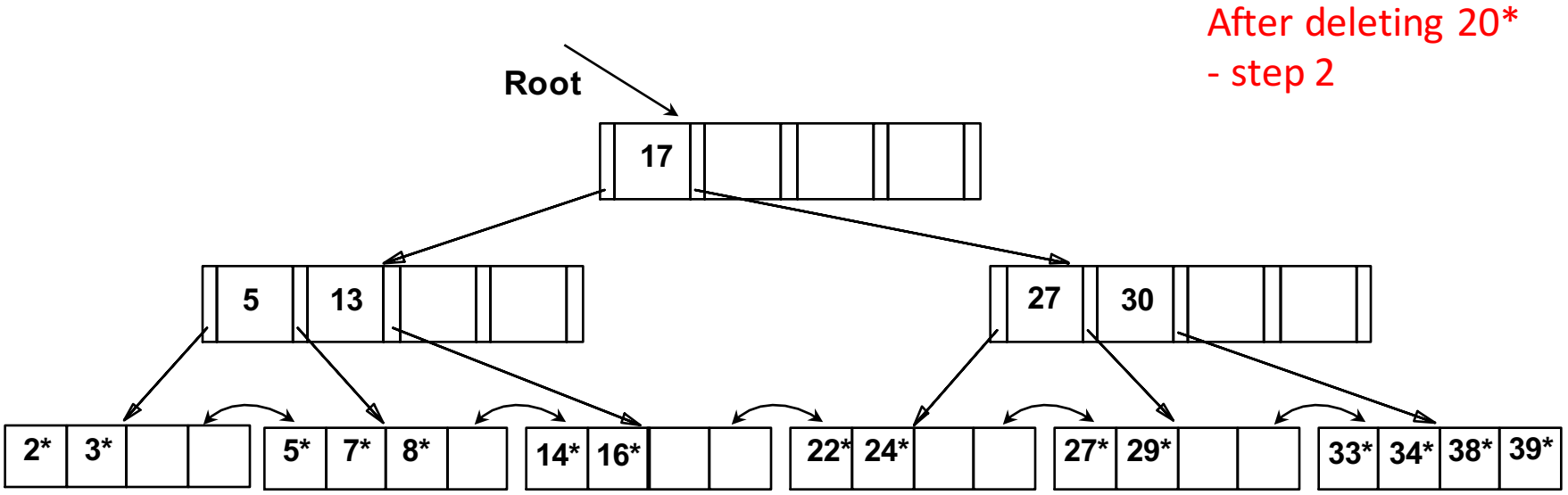
Before deleting 20*

Example Tree: Delete 20*



- < 2 entries in leaf-node
- Redistribute

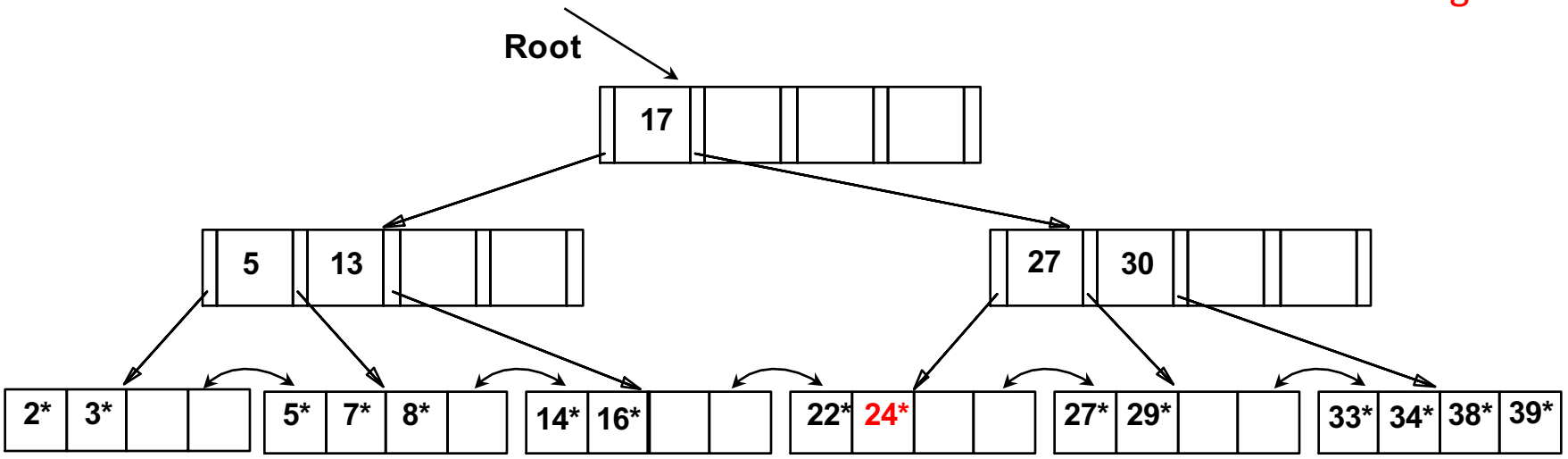
Example Tree: Delete 20*



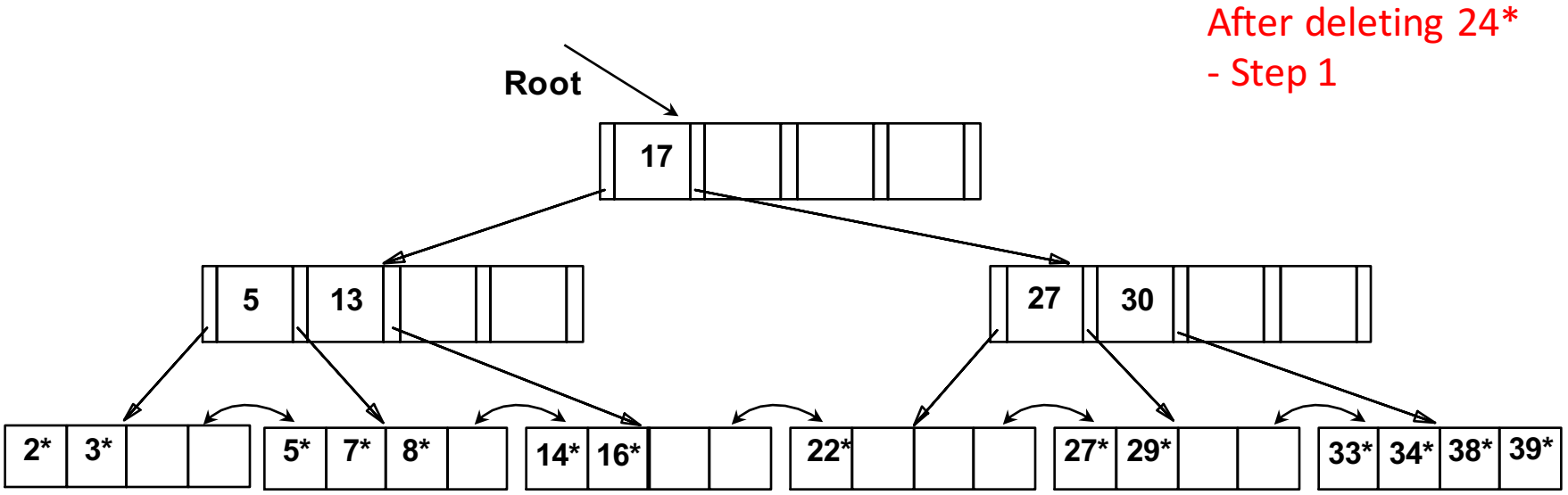
- Notice how middle key is copied up

Example Tree: … And Then Delete 24*

Before deleting 24*



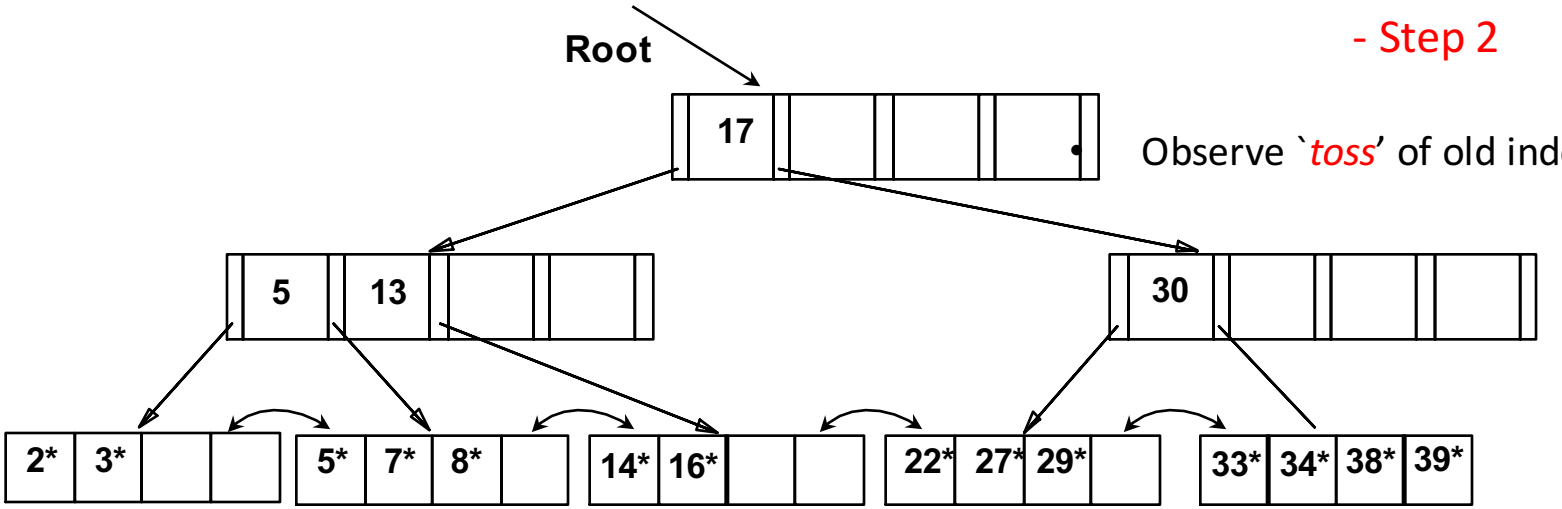
Example Tree: … And Then Delete 24*



- Once again, imbalance at leaf
- Can we borrow from sibling(s)?
- No – d-1 and d entries
- Need to merge

Example Tree: ... And Then Delete 24*

After deleting 24*
- Step 2

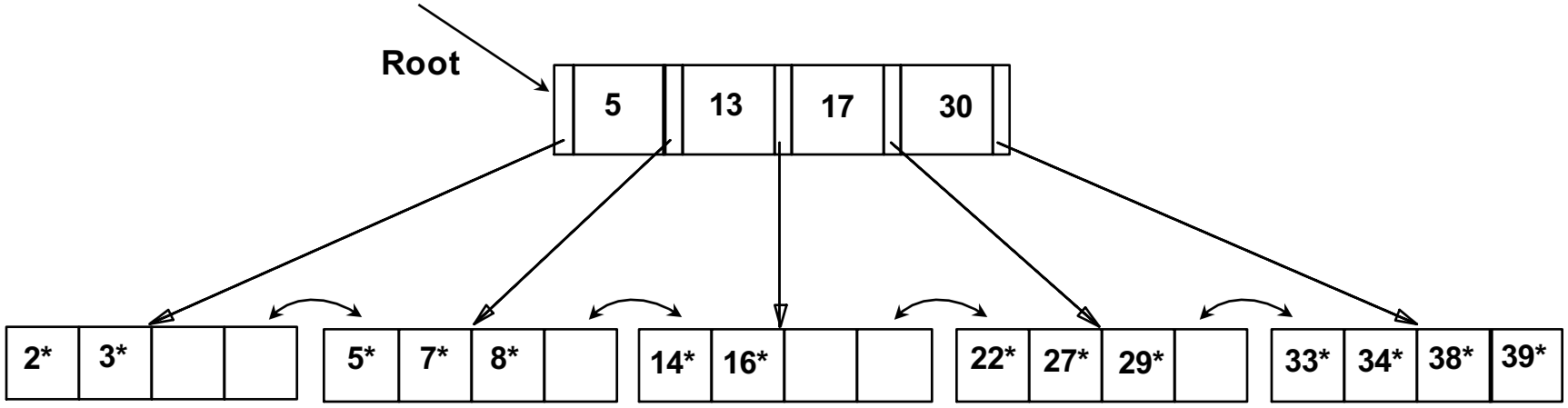


Observe `*toss*' of old index entry 27

- Imbalance at parent
- Merge again
- But need to "pull down" root index entry

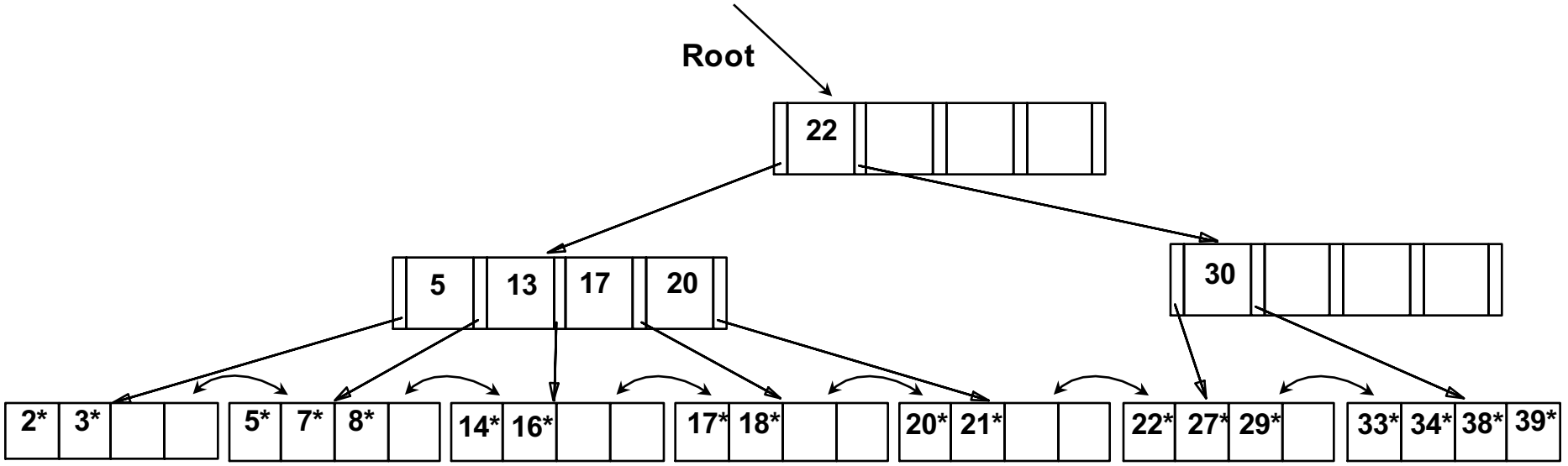
because, three index 5, 13, 30
but five pointers to leaves

Final Example Tree



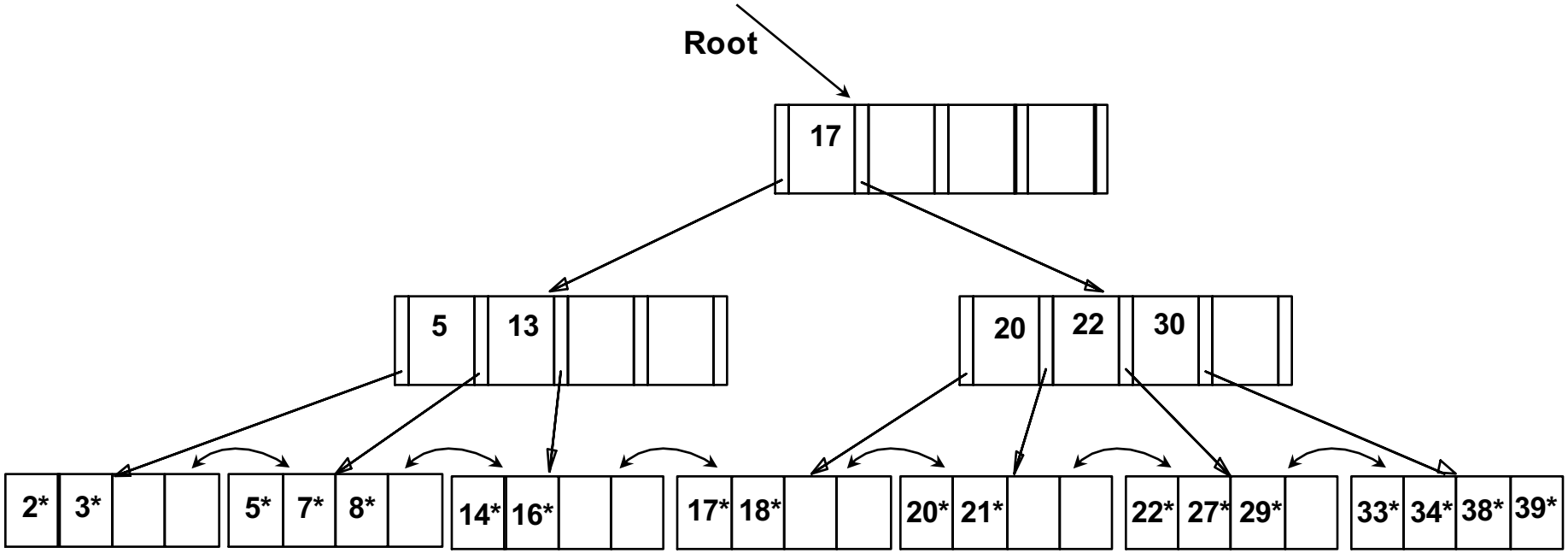
Example of Non-leaf Re-distribution

- An intermediate tree is shown
- 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.



Duplicates

- First Option:
  - The basic search algorithm assumes that all entries with the same key value resides on the same leaf page
  - If they do not fit, use overflow pages (like ISAM)
- Second Option:
  - Several leaf pages can contain entries with a given key value
  - Search for the left most entry with a key value, and follow the leaf-sequence pointers
  - Need modification in the search algorithm
- if k* = <k, rid>, several entries have to be searched
  - Or include rid in k – becomes unique index, no duplicate
  - If k* = <k, rid-list>, some solution, but if the list is long, again a single entry can span multiple pages

A Note on `Order'

- *Order* (**d**)
  - denotes minimum occupancy
  - replaced by physical space criterion in practice (`*at least half-full*')
  - Index pages can typically hold many more entries than leaf pages
  - 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 (if we use Alternative (3))

Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- ISAM is a static structure.
  - Only leaf pages modified; overflow pages needed.
  - Overflow chains can degrade performance unless size of data set and data distribution stay constant.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced; $\log_F N$ cost.
  - High fanout (**F**) means depth rarely more than 3 or 4.
  - Almost always better than maintaining a sorted file
  - Most widely used index in database management systems because of its versatility.
  - One of the most optimized components of a DBMS.

Hash-based Indexing

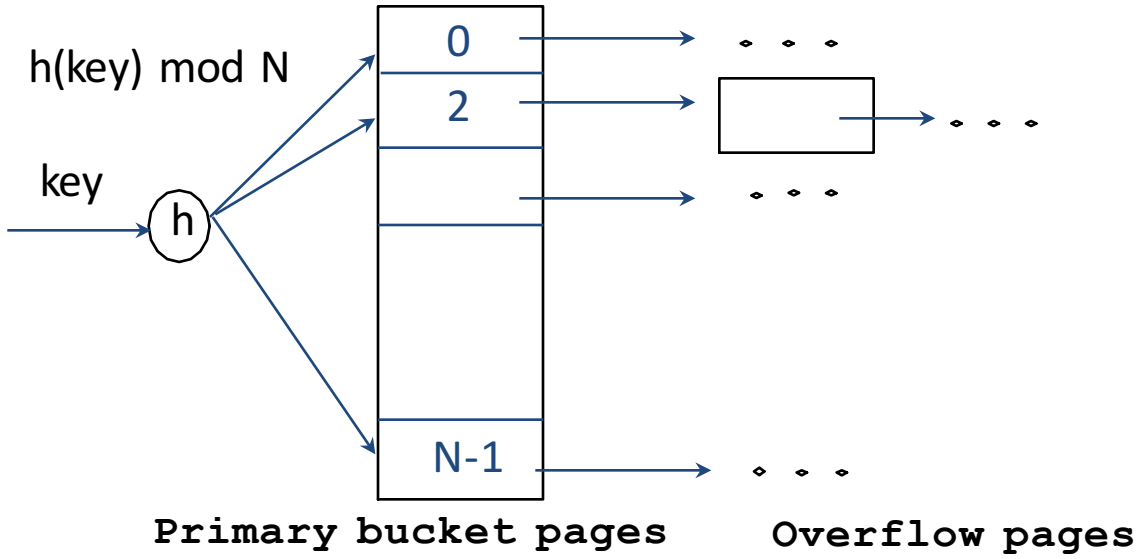
Introduction

- Hash-based indexes are best for equality selections
  - Cannot support range searches
  - But useful in implementing relational operators like join (later)

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

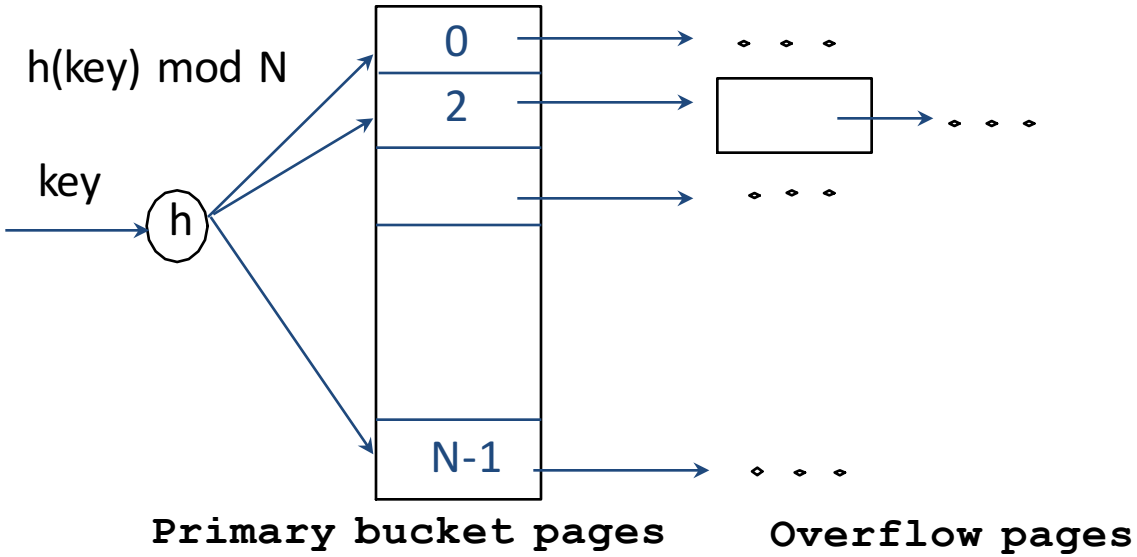
Static Hashing

- Pages containing data = a collection of buckets
  - each bucket has one primary page, also possibly overflow pages
  - buckets contain data entries k*



Static Hashing

- # primary pages fixed
  - allocated sequentially, never de-allocated, overflow pages if needed.
- h(k) mod N = bucket to which data entry with key k belongs
  - N = # of buckets



Static Hashing

- Hash fn works on search key field of record r
  - Must distribute values over range 0 ... N-1.
  - h(key) = (a * key + b) usually works well.
  - a and b are constants – chosen to tune h
- Advantage:
  - #buckets known – pages can be allocated sequentially
  - search needs 1 I/O,
  - insert/delete needs 2 I/O if no overflow page
- Disadvantage:
  - Long overflow chains can develop and degrade performance
- Solutions:
  - keep some pages say 80% full initially
  - Rehash if overflow pages (can be expensive)
  - or use Dynamic Hashing

Dynamic Hashing Techniques

- Extendible Hashing
- Linear Hashing

To be continued in the next lecture