Database Applications (15-415)

DBMS Internals: Part II
Lecture 10, February 17, 2014

Mohammad Hammoud

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

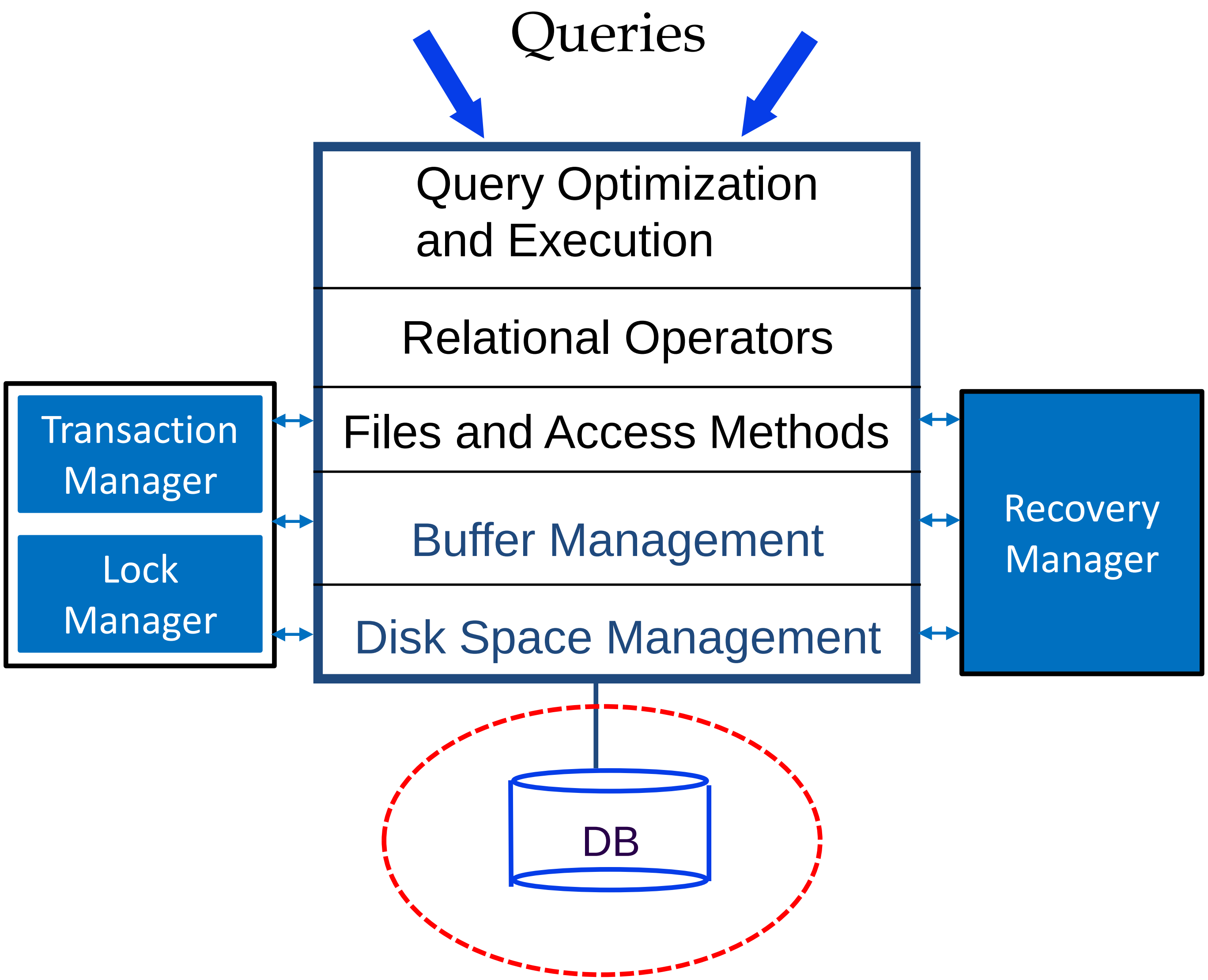
Today…

- Last Session:
  - DBMS Internals- Part I

- Today's Session:
  - DBMS Internals- Part II
    - Brief summaries of disks, disk space management, and buffer management
    - Files and Access Methods (*for today*, only file organizations and ISAM Trees)

- Announcements:
  - Project 1 is due tomorrow (Feb 18) by midnight
  - The midterm exam is on Wednesday Feb 26 (*all* material are included)

DBMS Layers


Queries
Query Optimization
and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction
Manager
Lock
Manager
Recovery
Manager
DB

Disks: A “Very” Brief Summary

- DBMSs store data in disks
  - Disks provide large, cheap and non-volatile storage

- I/O time dominates!

- The cost depends on the locations of pages on disk (*among others*)

- It is important to arrange data *sequentially* to minimize *seek* and *rotational* delays

Disks: A "Very" Brief Summary

- Disks can cause reliability and performance problems

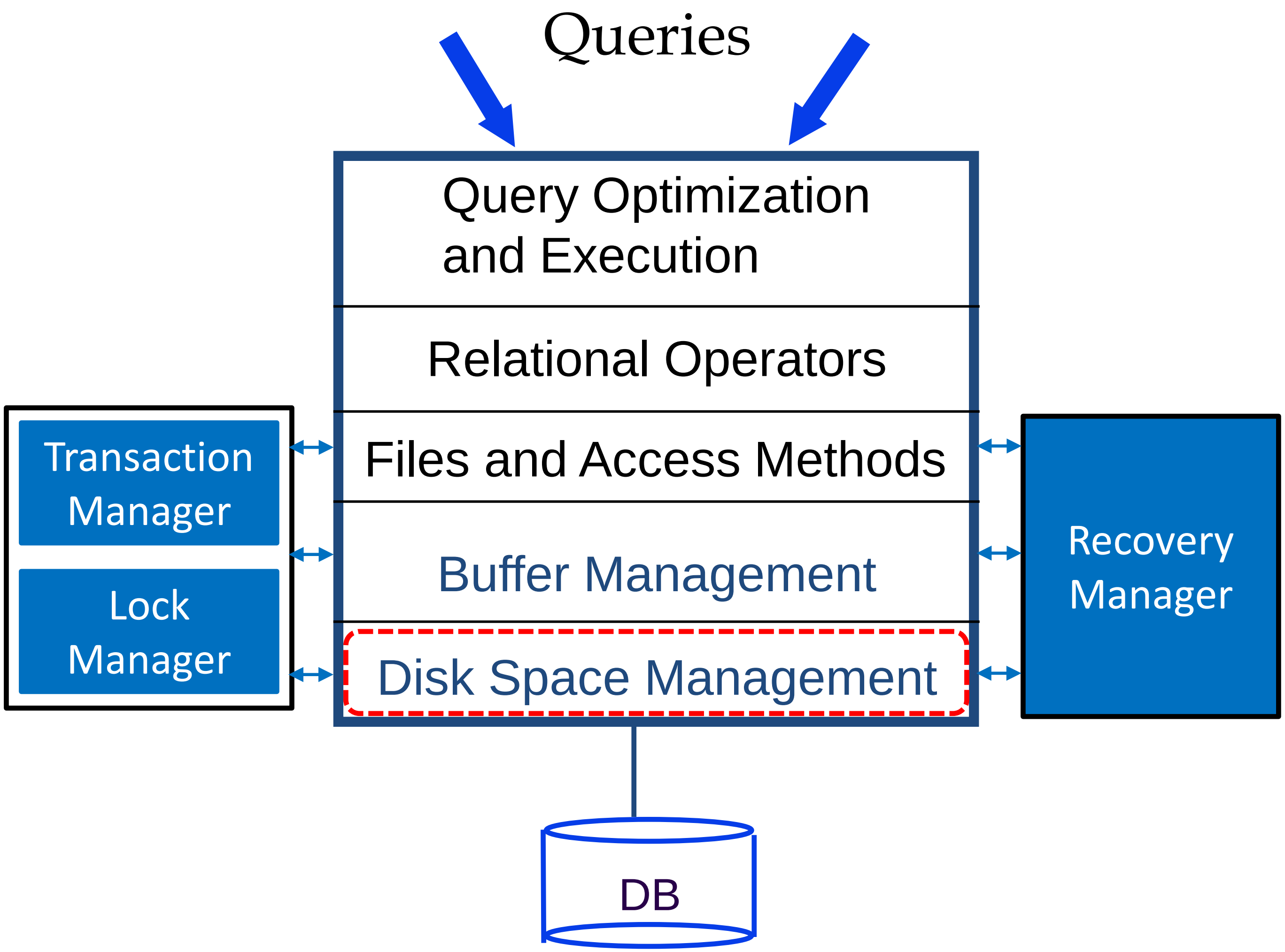
- To mitigate such problems we can adopt "multiple disks" and accordingly gain:
  1. More capacity
  2. Redundancy
  3. Concurrency

- To achieve only redundancy we apply mirroring

- To achieve only concurrency we apply striping

- To achieve redundancy *and* concurrency we apply RAID levels 2, 3, 4 or 5

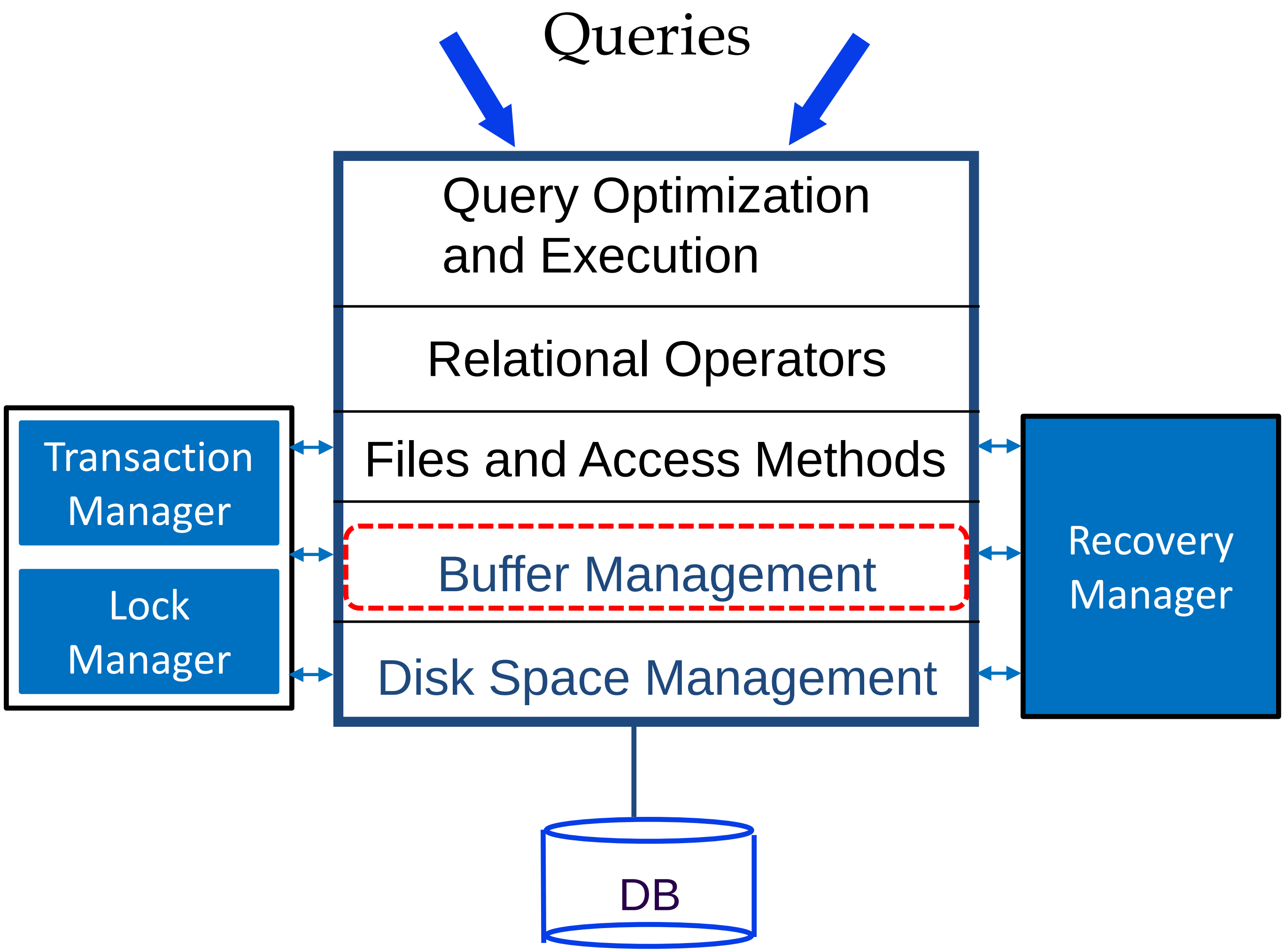
DBMS Layers


Queries
Query Optimization
and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction
Manager
Lock
Manager
Recovery
Manager
DB

Disk Space Management: A "Very" Brief Summary

- The lowest layer of the DBMS software is the disk space manager
  - It attempts to allocate/de-allocate and read/write pages as a *contiguous* sequence of blocks on disks
  - It *abstracts* hardware details from higher DBMS layers
  - It can keep track of free blocks by maintaining a *list of free blocks* or a *bitmap* with 1 bit for each disk block
  - It typically does not rely on OS functionalities for practical (e.g., portability) and technical (e.g., addressing large amount of data) reasons

DBMS Layers

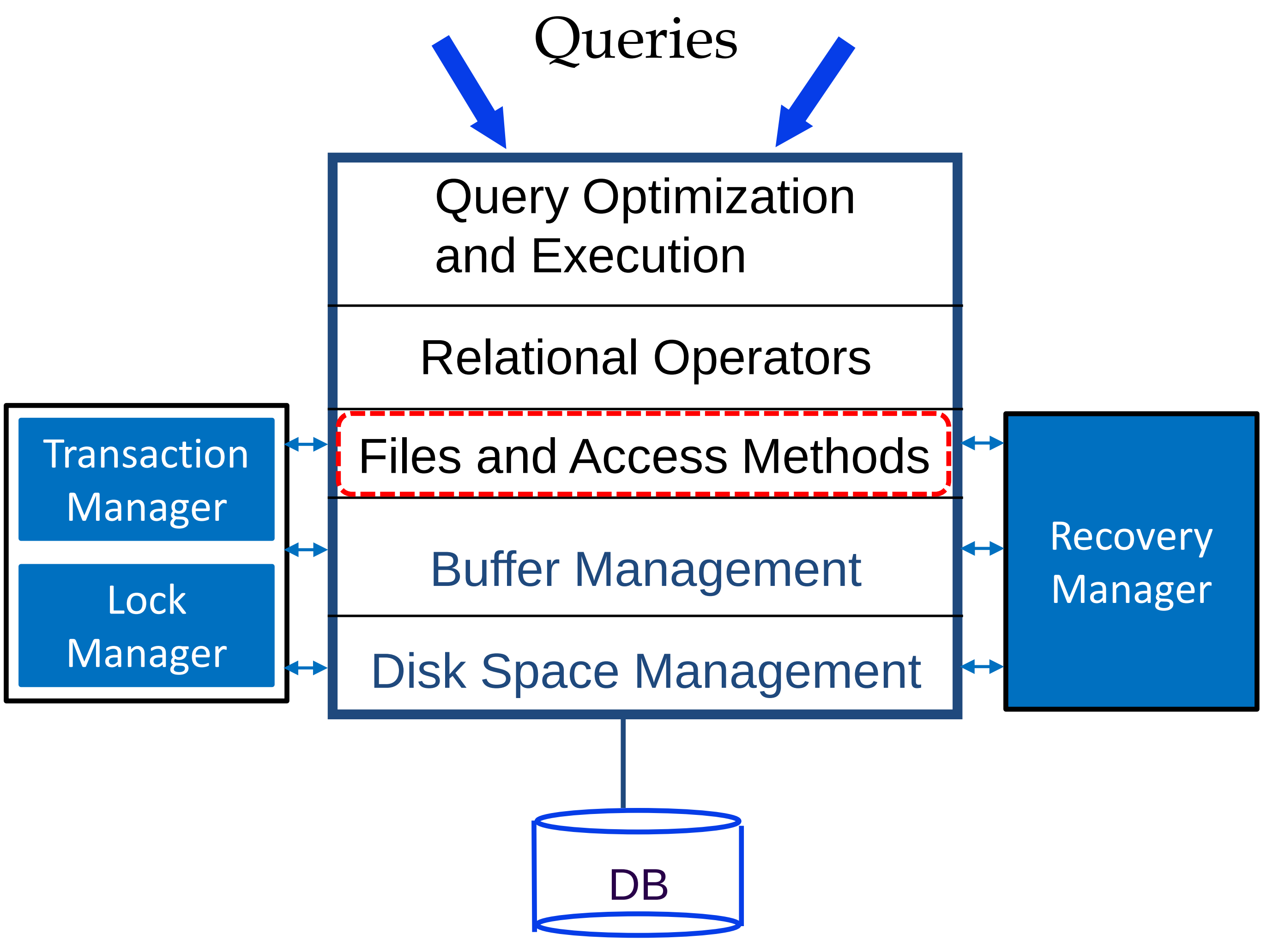


Queries
Query Optimization and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction Manager
Lock Manager
Recovery Manager
DB

Buffer Management: A "Very" Brief Summary

- The buffer manager sits on top of the disk space manager
  - It fetches pages from disks to RAM as needed in response to read/write requests
  - It hides the fact that not all data are in the RAM (similar to the classical OS virtual memory)
  - It applies effective *replacement policies* (e.g., LRU or Clock)
  - It usually does not rely on the OS functionalities for reasons like *predicting* (more accurately) *page reference patterns* and *forcing pages to disks* (required by the WAL protocol)

DBMS Layers

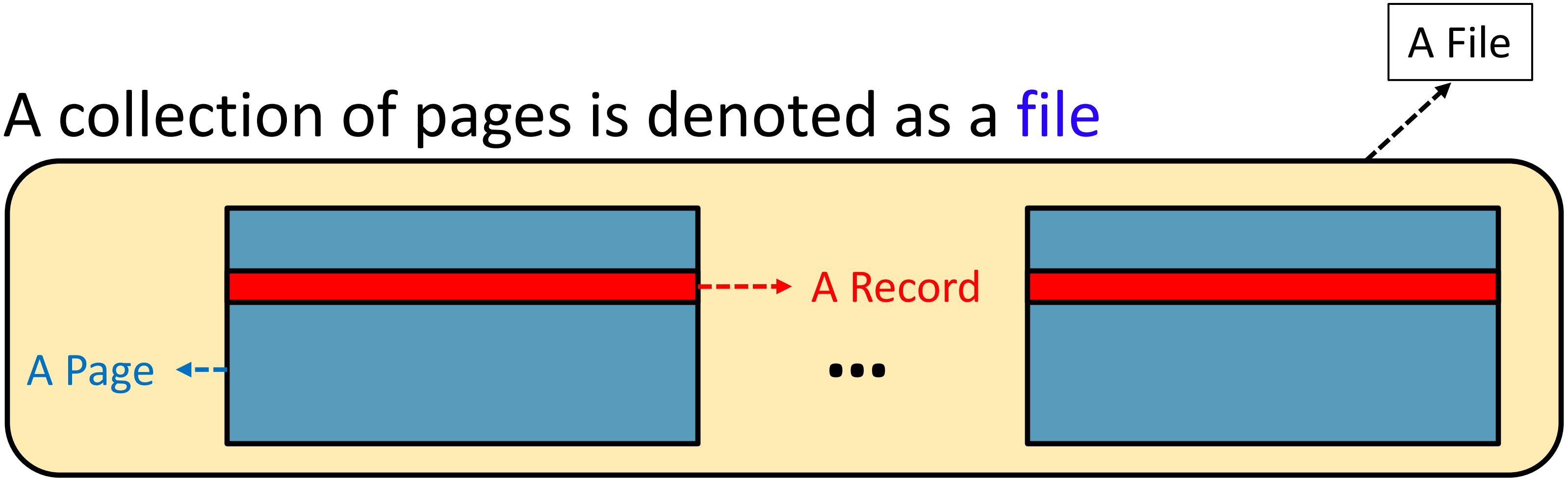

Queries
Query Optimization and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction Manager
Lock Manager
Recovery Manager
DB

Outline

- File Organizations ✓
- Why Indexing?
- Indexed Static Access Method

Records, Pages and Files

- Higher-levels of DBMSs deal with records (not pages!)
- At lower-levels, records are stored in pages
- But, a page might not fit all records of a database
  - Hence, multiple pages might be needed
- A collection of pages is denoted as a file



File Operations and Organizations

- A file is a collection of pages, each containing a collection of records

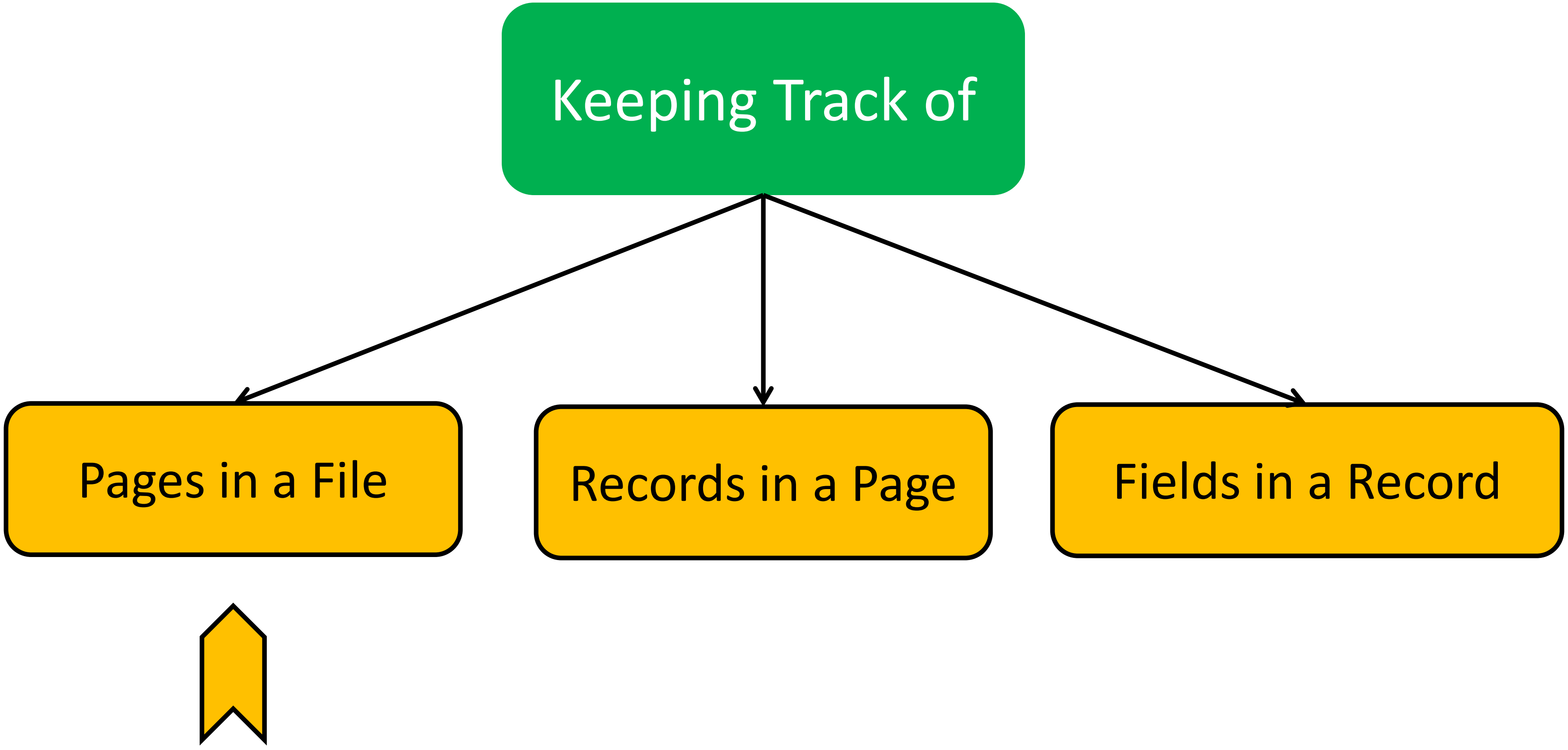
- Files must support operations like:
  - Insert/Delete/Modify records
  - Read a particular record (specified using a *record id*)
  - Scan all records (possibly with some conditions on the records to be retrieved)

- There are several organizations of files:
  - Heap
  - Sorted
  - Indexed

Heap Files

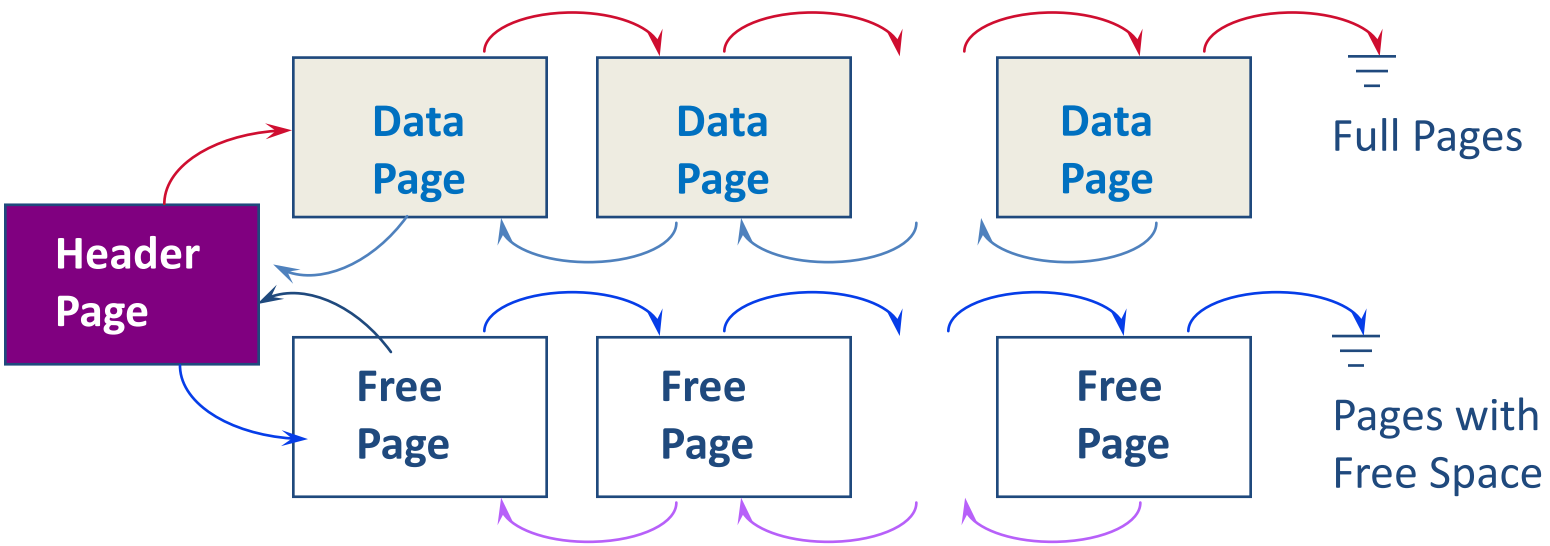
- Records in heap file pages do not follow any particular order
- As a heap file grows and shrinks, disk pages are allocated and de-allocated
- To support record level operations, we must:
  - Keep track of the *pages* in a file
  - Keep track of the *records* on a page
  - Keep track of the *fields* on a record

Supporting Record Level Operations



Heap Files Using *Lists* of Pages

- A heap file can be organized as a *doubly linked list* of pages



- The Header Page (i.e., <*heap_file_name*, *page_1_addr*> is stored in a known location on disk

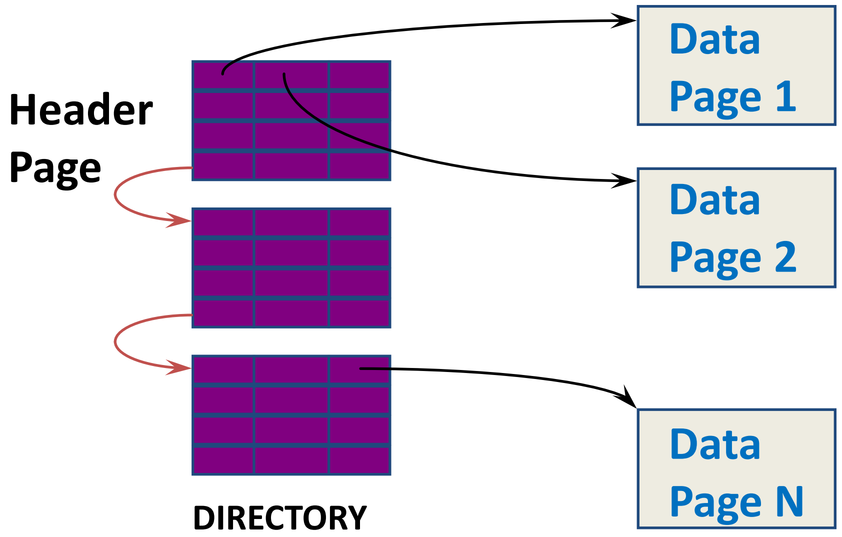
- Each page contains 2 'pointers' plus data

Heap Files Using *Lists* of Pages

- It is likely that every page has at least a few free bytes
- Thus, virtually all pages in a file will be on the free list!
- To insert a typical record, we must retrieve and examine several pages on the free list before one with *enough* free space is found
- This problem can be addressed using an alternative design known as the directory-based heap file organization

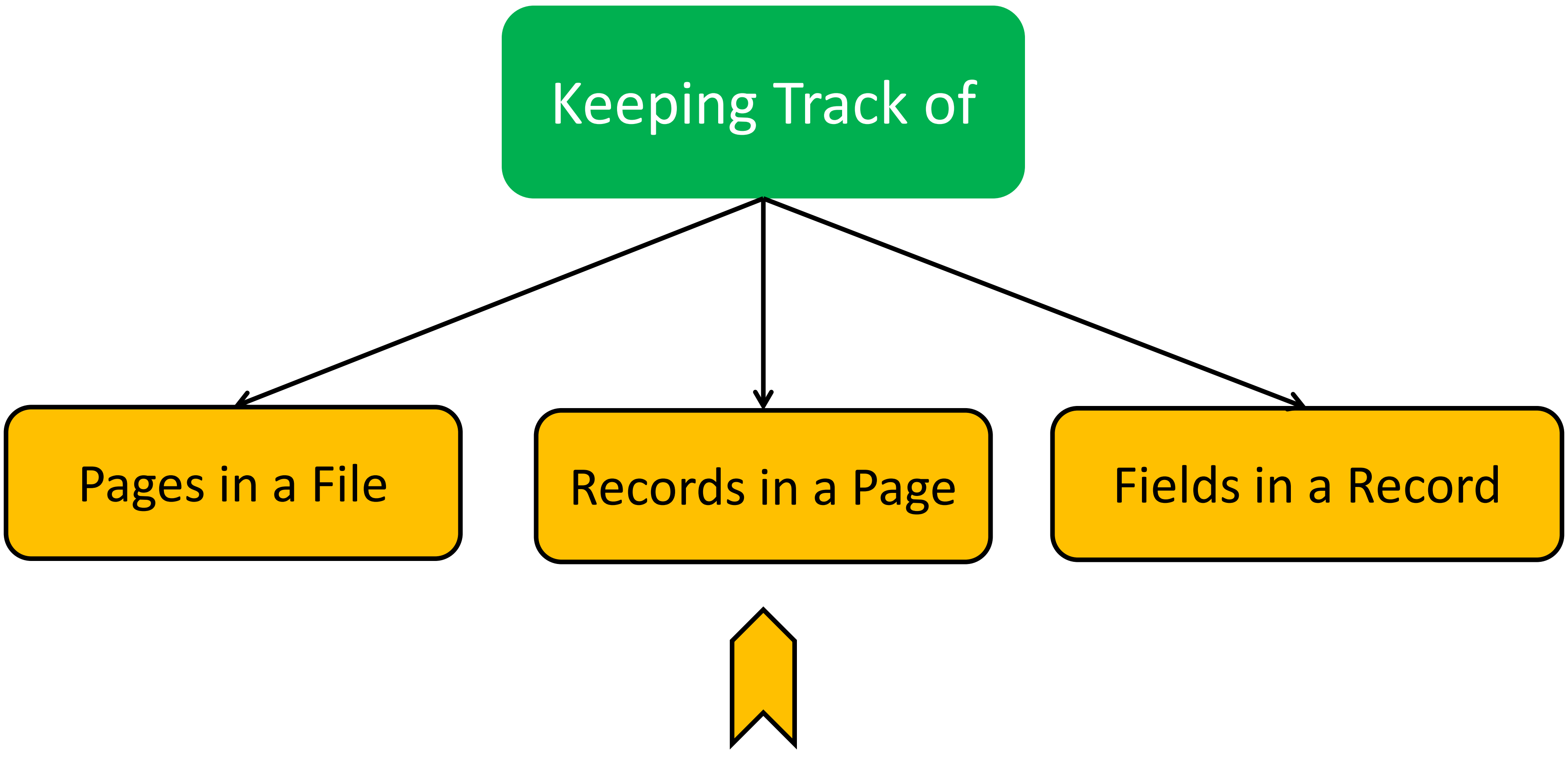
Heap Files Using *Directory* of Pages

- A directory of pages can be maintained whereby each directory entry identifies a page in the heap file



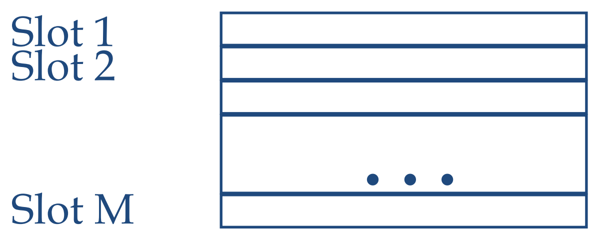
- Free space can be managed via maintaining:
  - A *bit* per entry (indicating whether the corresponding page has any free space)
  - A *count* per entry (indicating the amount of free space on the page)

Supporting Record Level Operations



Page Formats

- A page in a file can be thought of as a collection of slots, each of which contains a record

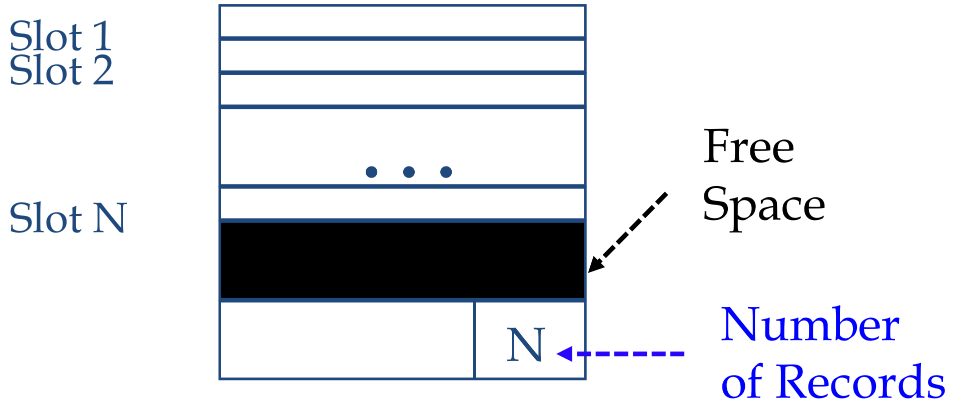


- A record can be identified using the pair <page_id, slot_#>, which is typically referred to as record id (rid)

- Records can be either:
  - Fixed-Length
  - Variable-Length

Fixed-Length Records

- When records are of fixed-length, slots become *uniform* and can be arranged *consecutively*

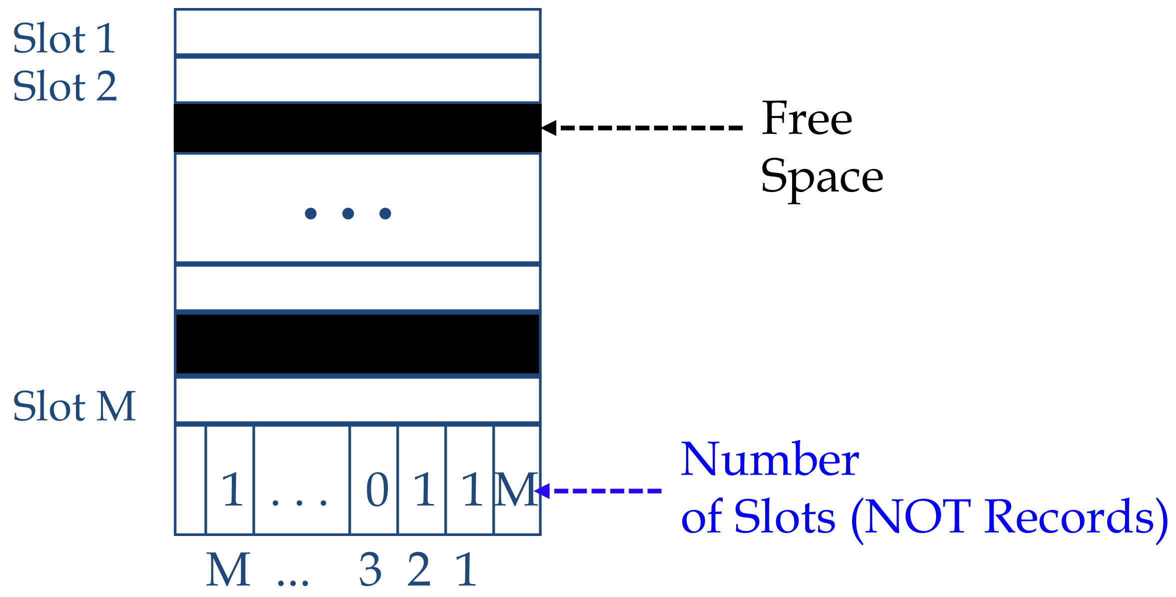


- Records can be located by simple offset calculations

- Whenever a record is *deleted*, the last record on the page is *moved* into the vacated slot
  - This changes its rid <page_id, slot_#> (*may not be acceptable!*)

Fixed-Length Records

- Alternatively, we can handle deletions by using an array of bits



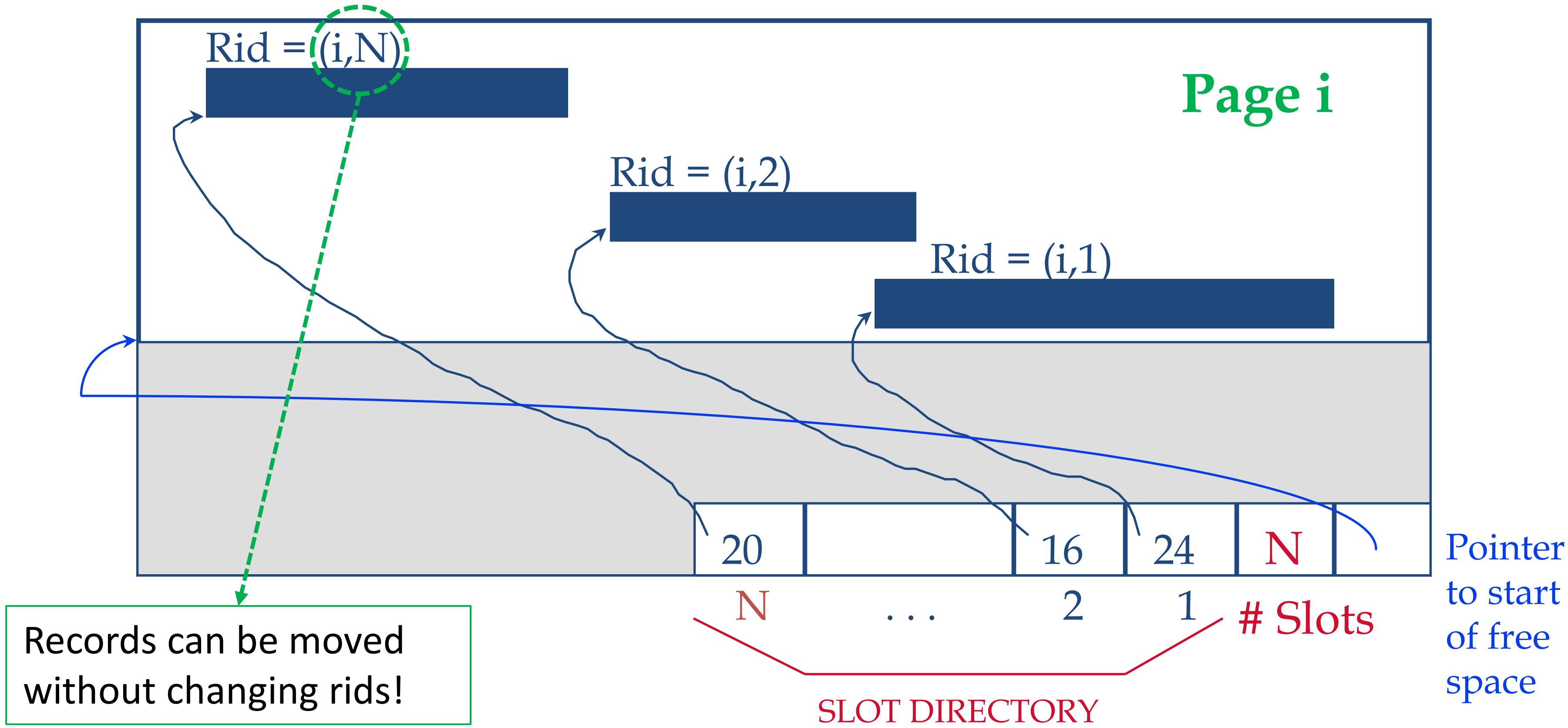
- When a record is deleted, its bit is turned off, thus, the rids of currently stored records remain the same!

Variable-Length Records

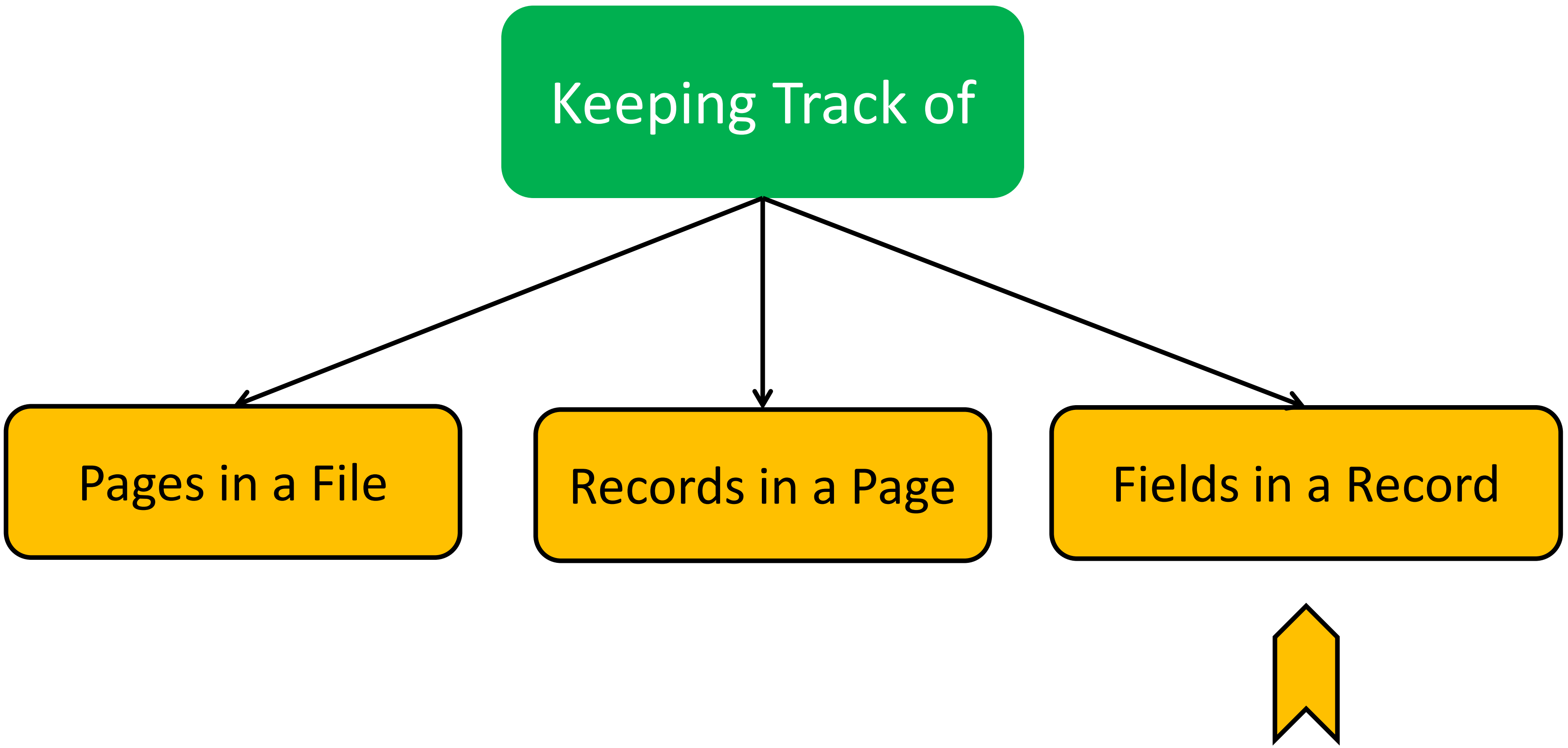
- If the records are of variable length, we cannot divide the page into a fixed collection of slots
- When a new record is to be inserted, we have to find an empty slot of “just” the right length
- Thus, when a record is deleted, we better ensure that all the free space is contiguous
- The ability of moving records “*without changing rids*” becomes crucial!

Pages with Directory of Slots

- A flexible organization for variable-length records is to maintain a directory of slots with a *<record_offset, record_length>* pair per a page



Supporting Record Level Operations

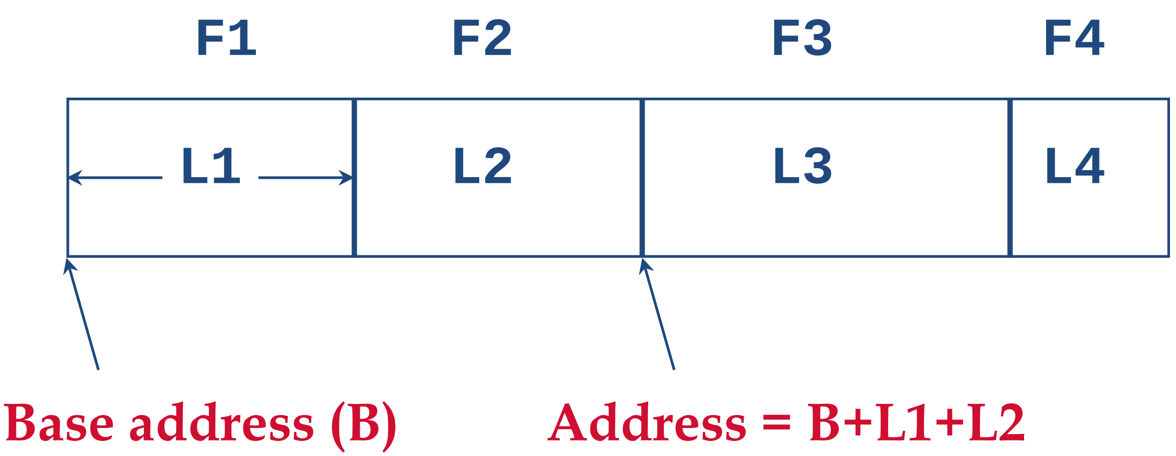


Record Formats

- Fields in a record can be either of:
  - Fixed-Length: each field has a fixed length and the number of fields is also fixed
  - Variable-Length: fields are of variable lengths but the number of fields is fixed
- Information common to all records (e.g., number of fields and field types) are stored in the system catalog

Fixed-Length Fields

- Fixed-length fields can be stored consecutively and their addresses can be calculated using information about the lengths of preceding fields



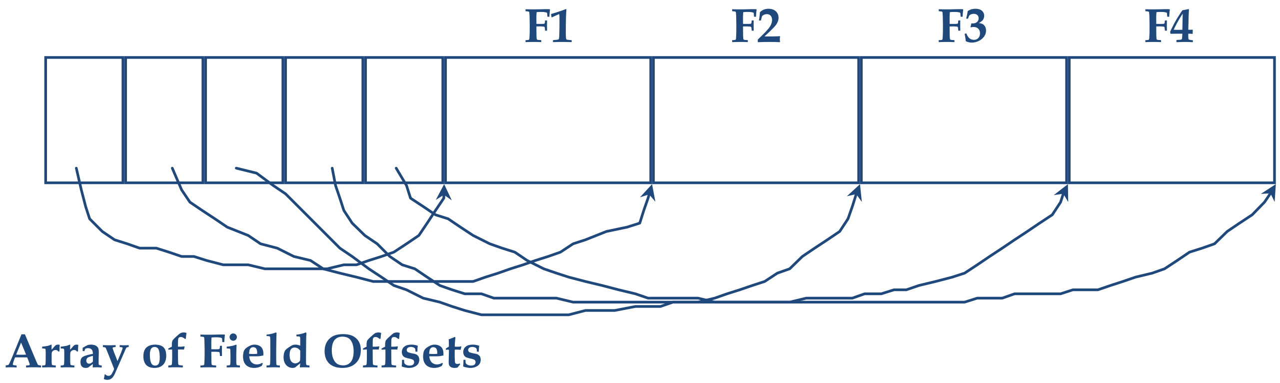
Variable-Length Fields

- There are two possible organizations to store variable-length fields
  1. Consecutive storage of fields separated by delimiters

This entails a scan of records to locate a desired field!

Variable-Length Fields

- There are two possible organizations to store variable-length fields
  1. Consecutive storage of fields separated by delimiters
  2. Storage of fields with an array of integer offsets



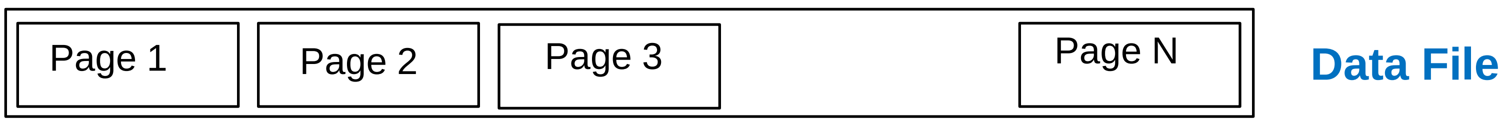
This offers *direct access* to a field in a record and stores NULL values efficiently!

Outline

- File Organizations
- Why Indexing? ✓
- Indexed Static Access Method

Motivation

- Consider a file of student records *sorted* by GPA

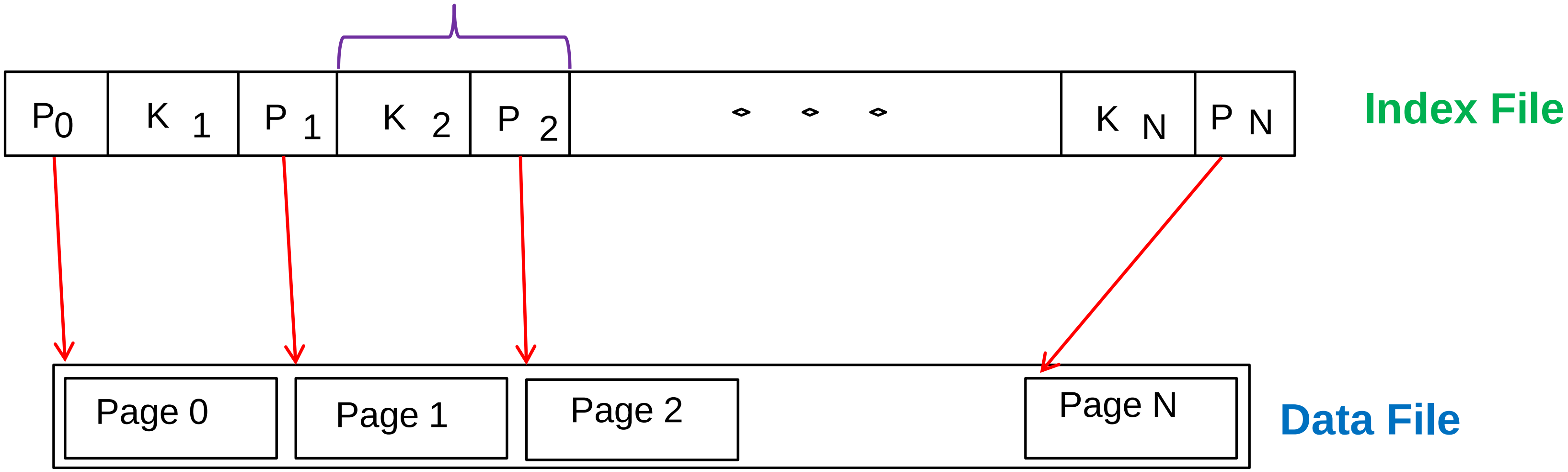


- How can we answer a *range selection* (E.g., *"Find all students with a GPA higher than 3.0"*)?
  - What about doing a *binary search* followed by a *scan*?
    - Yes, but…
  - What if the file becomes "very" large?
    - Cost is proportional to the number of pages fetched
    - Hence, may become very slow!

Motivation

- What about creating an *index file* (with one record per page) and do binary search there?

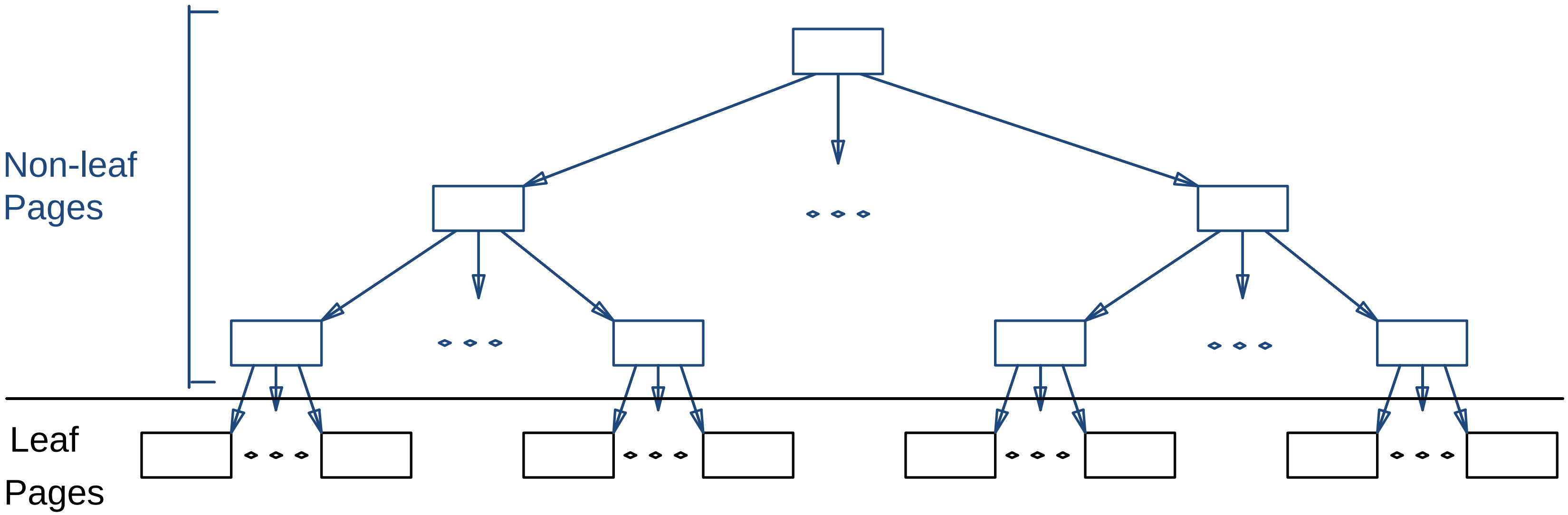
Index Entry = <first key on the page, pointer to the page>



- But, what if the index file becomes also “very” large?

Motivation

- Repeat recursively!



Each tree page is a disk page and all data records reside (*if chosen to be part of the index*) in ONLY leaf pages

How else data records can be stored?

Where to Store Data Records?

- In general, *3 alternatives* for “data records” (**k*)** can be adopted:
  - Alternative (1): K* is an actual data record with key **k**
  - Alternative (2): K* is a <**k**, **rid>** pair, where rid is the record id of a data record with search key **k**
  - Alternative (3): K* is a <**k**, rid-list> pair, where rid-list is a list of rids of data records with search key **k**

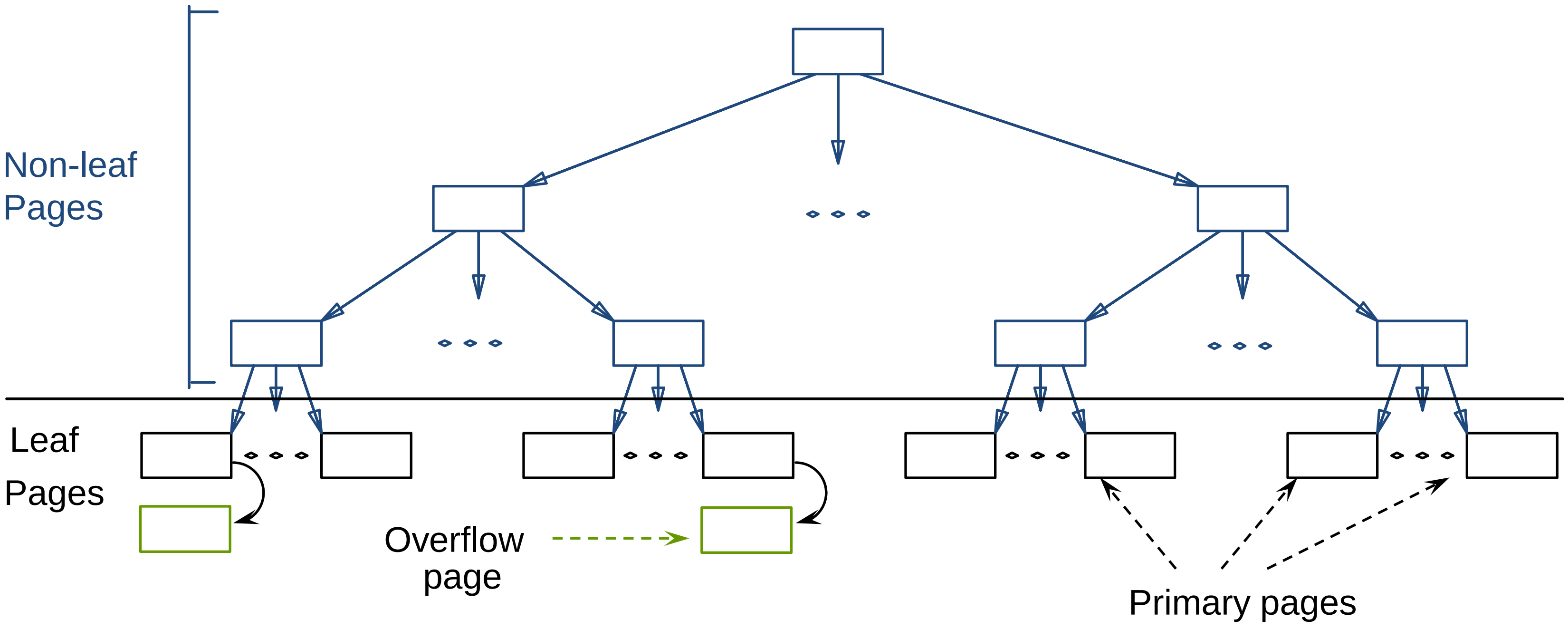
Where to Store Data Records?

- In general, *3 alternatives* for "data records" (**k***) can be adopted:
  - **A (1)**: Leaf pages contain the actual data (i.e., the data records)
  - **A (2)**: Leaf pages contain the <key, rid> pairs and actual data records are stored in a separate file
  - **A (3)**: Leaf pages contain the <key, rid-list> pairs and actual data records are stored in a separate file

The choice among these alternatives is orthogonal to the *indexing technique.*

ISAM Trees: Page Overflows

- Now, what if there are a lot of insertions?



This structure is referred to as *Indexed Sequential Access Method* (ISAM)

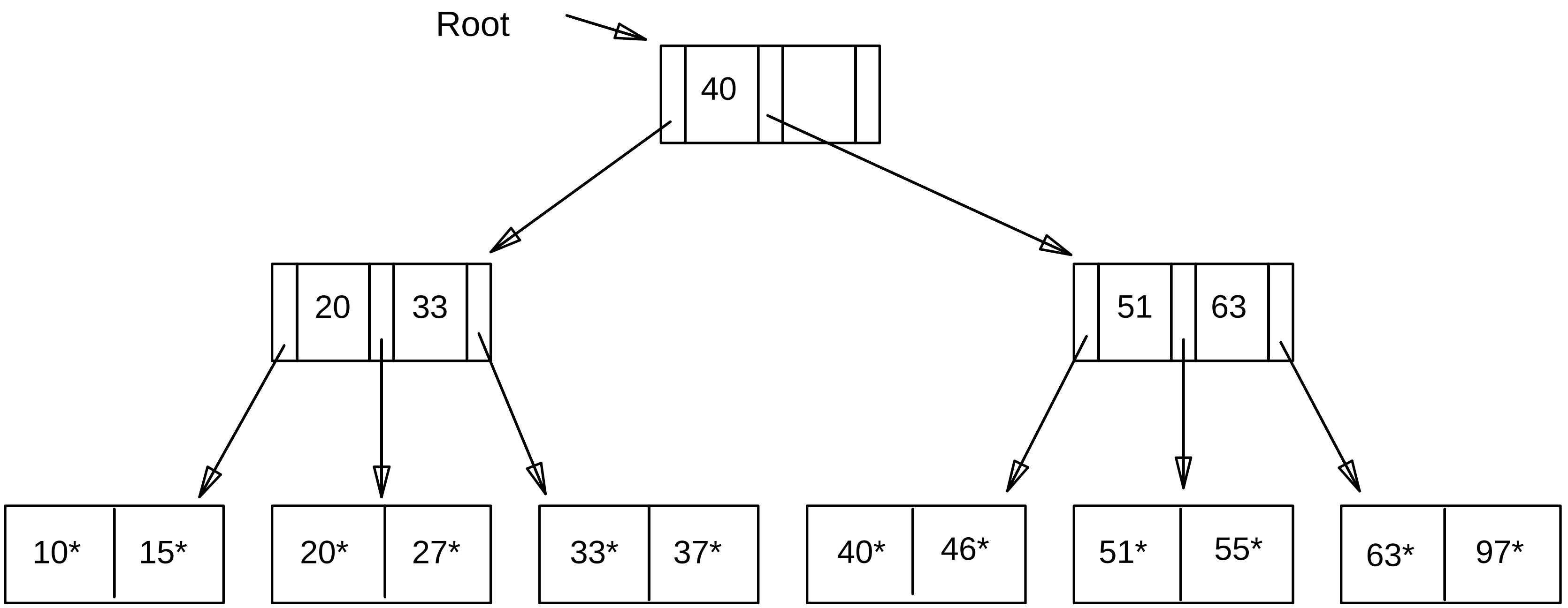
Outline

- Files and File Organizations
- Why Indexing?
- Indexed Static Access Method ✓

ISAM File Creation

- How to create an ISAM file?
  - All leaf pages are allocated *sequentially* and *sorted* on the search key value
  - If Alternative (2) or (3) is used, the data records are created and sorted before allocating leaf pages
  - The non-leaf pages are subsequently allocated

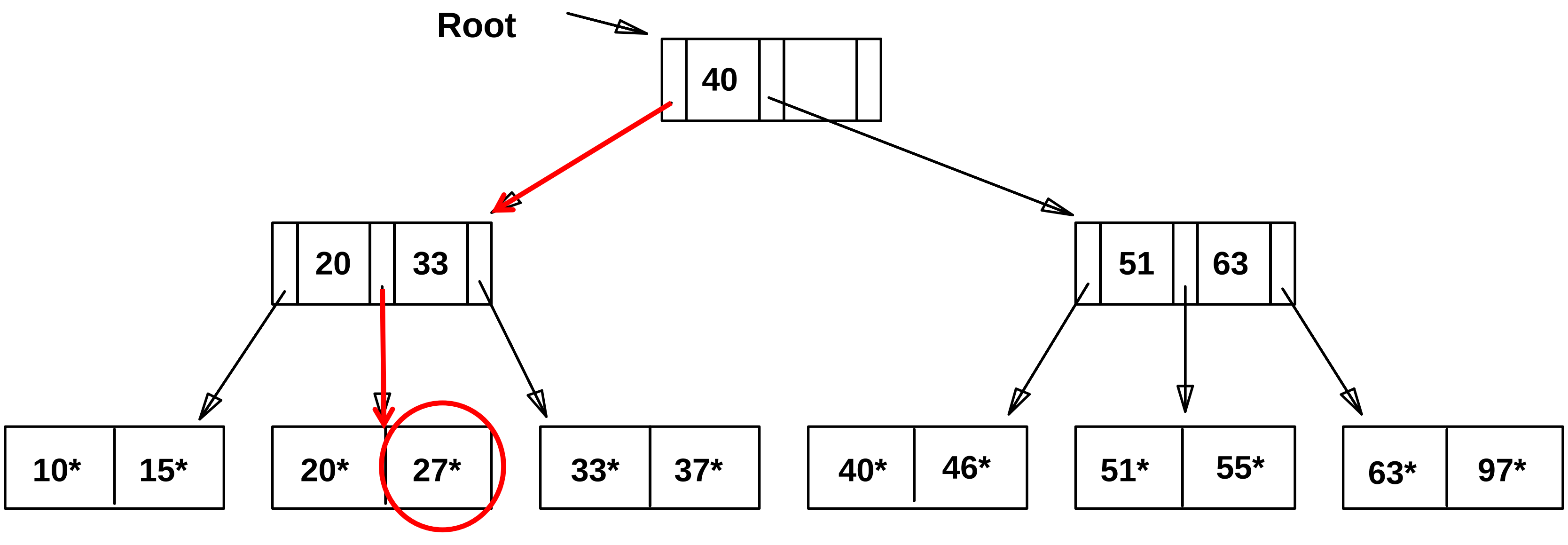
An Example of ISAM Trees



2 Entries Per Page.

ISAM: Searching for Entries

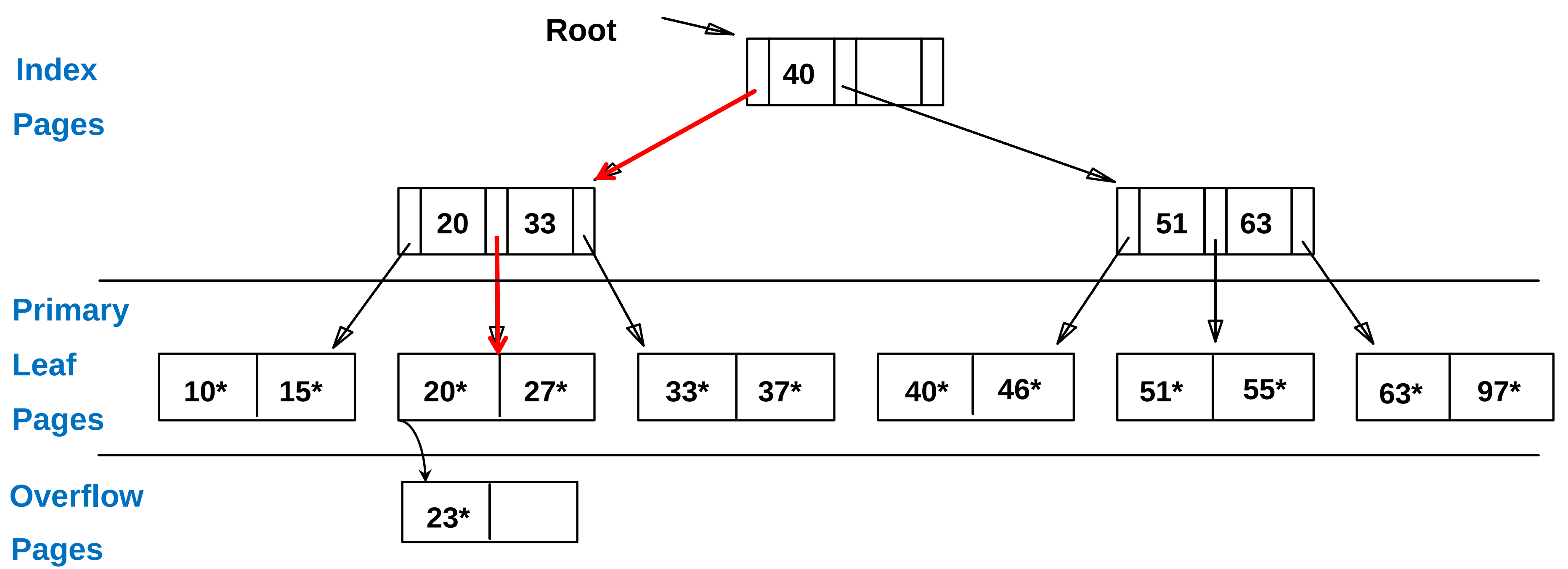
- Search begins at root, and key comparisons direct it to a leaf
- Search for 27*



ISAM: Inserting Entries

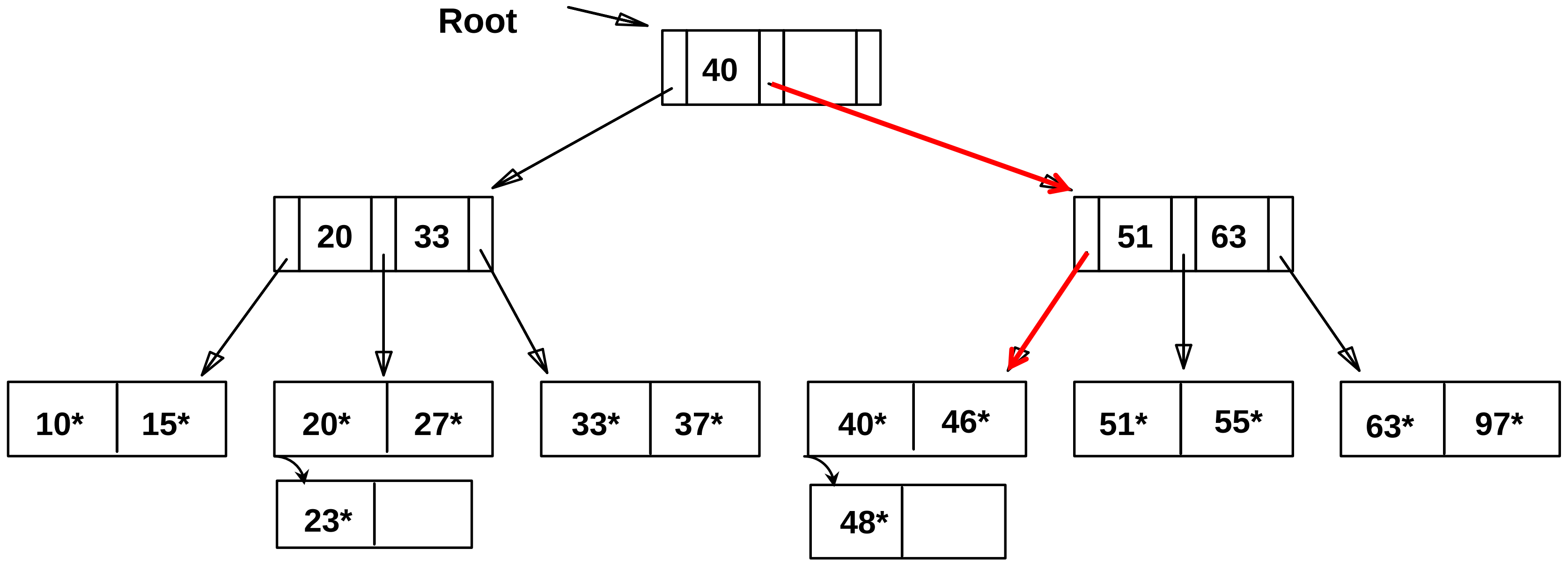
- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

- Insert 23*



ISAM: Inserting Entries

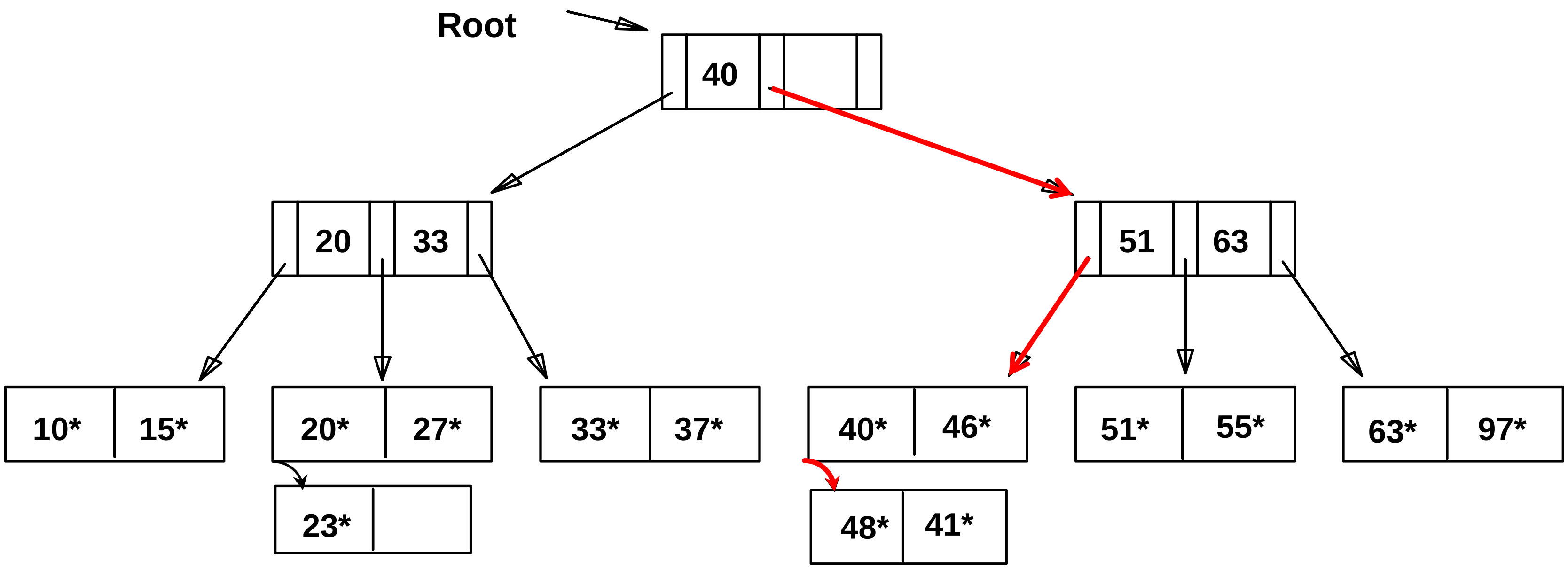
- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)
- Insert 48*



ISAM: Inserting Entries

- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

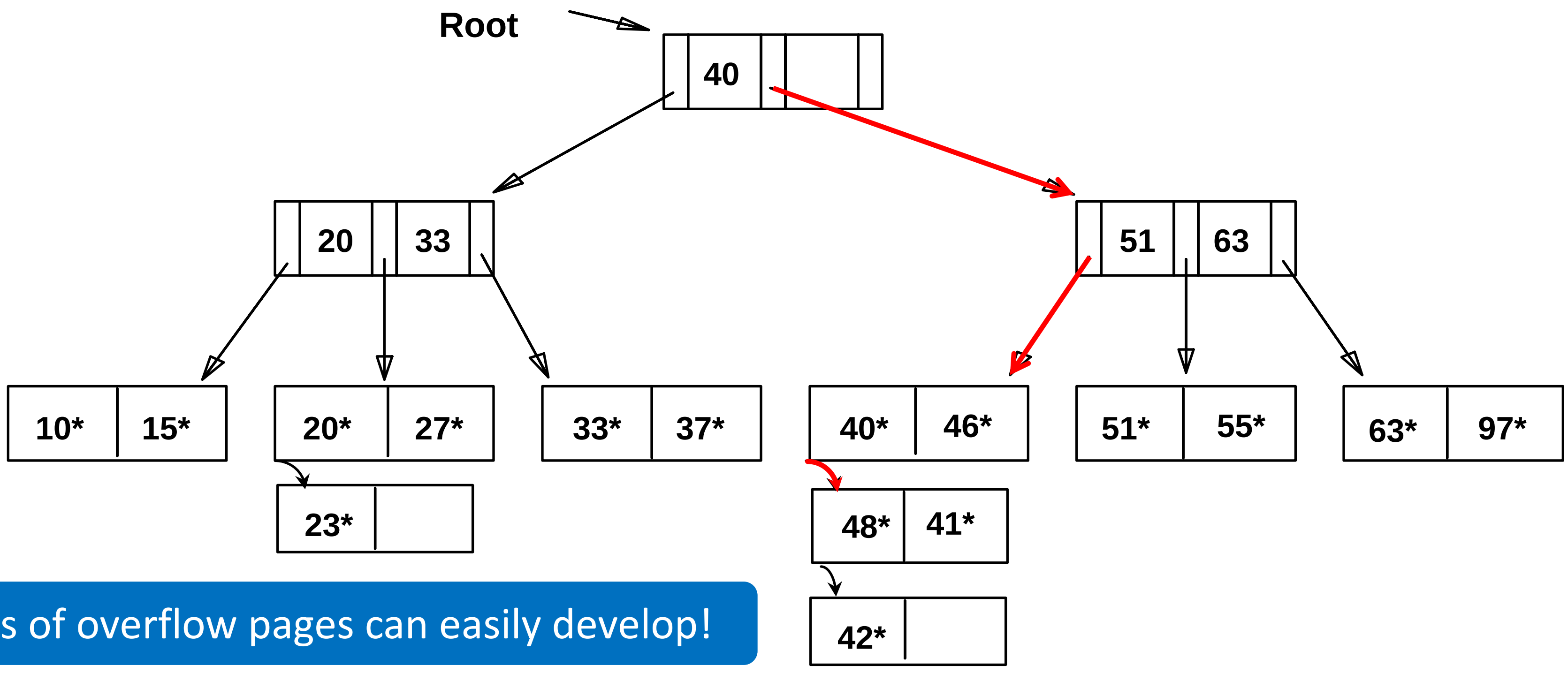
- Insert 41*



ISAM: Inserting Entries

- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

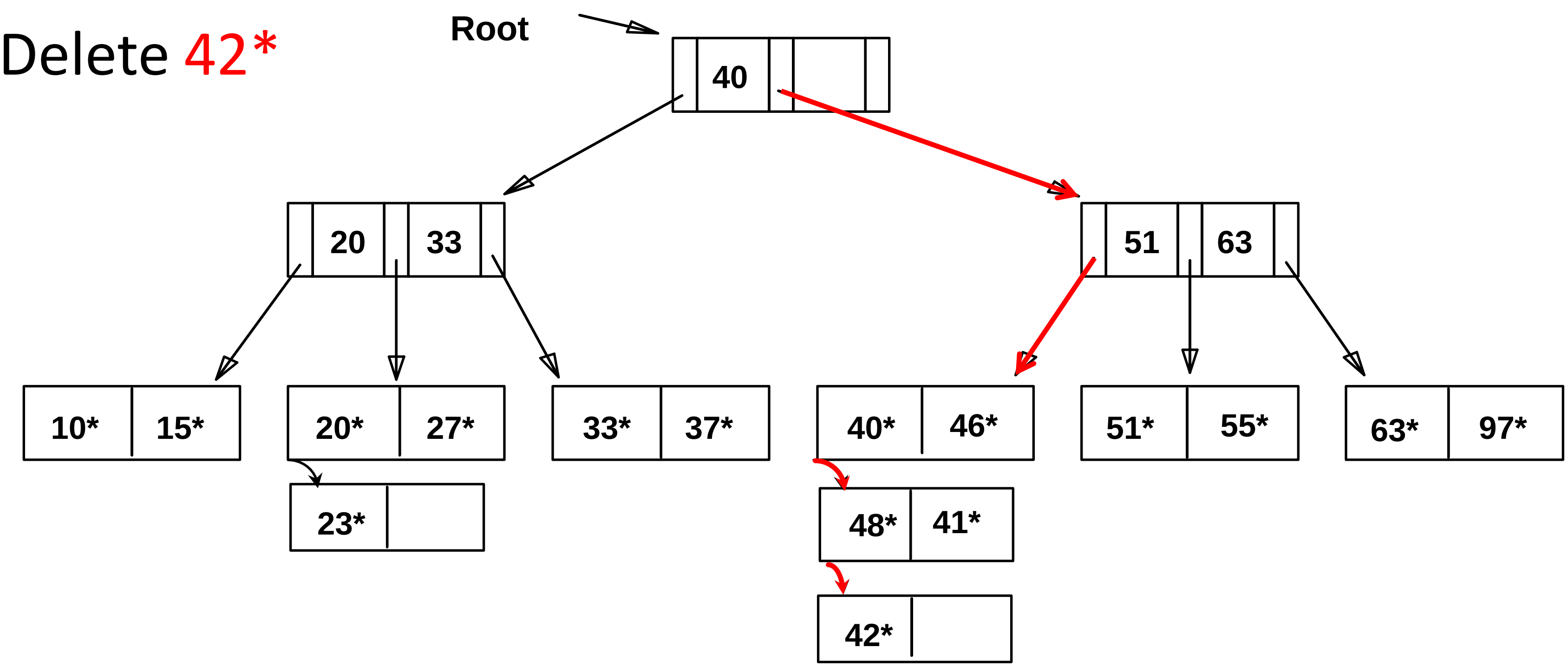
- Insert 42*



Chains of overflow pages can easily develop!

ISAM: Deleting Entries

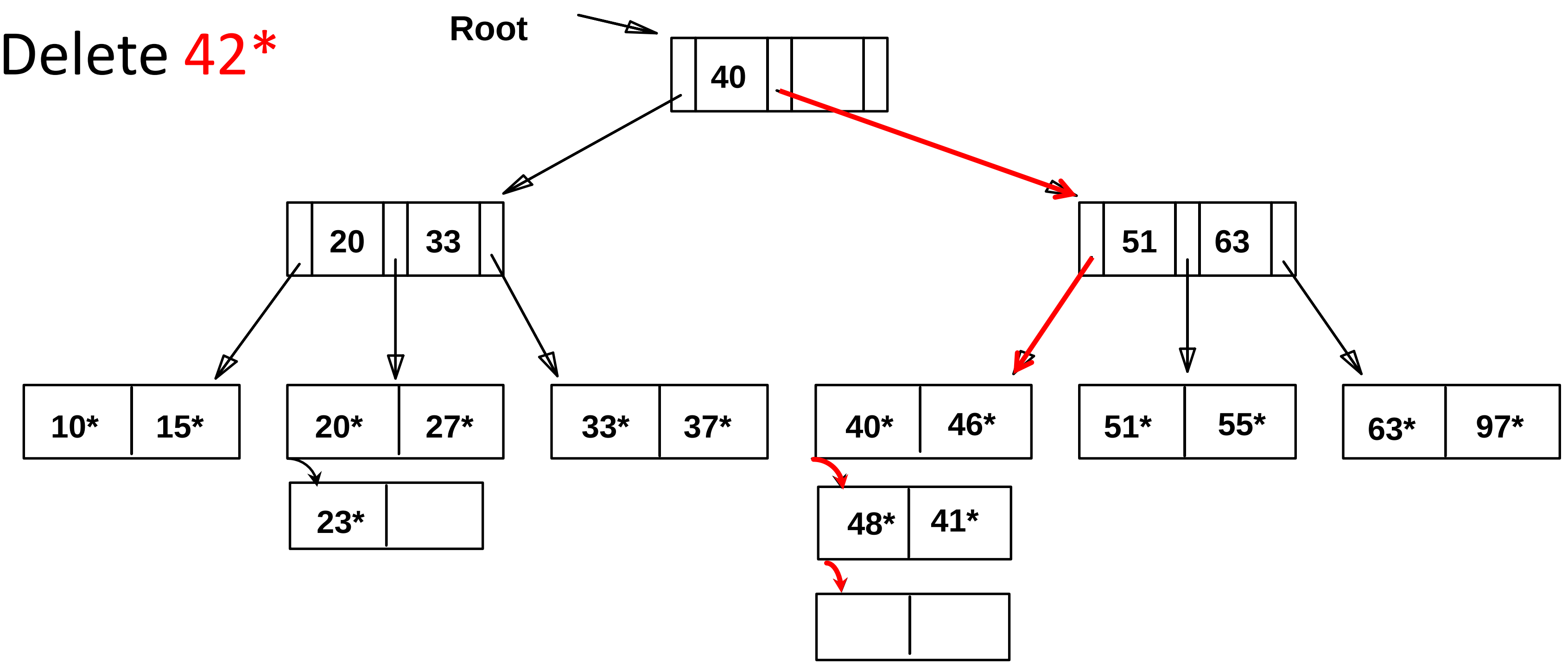
- The appropriate page is determined as for a search, and the entry is deleted (with ONLY overflow pages removed when becoming empty)
- Delete 42*



ISAM: Deleting Entries

- The appropriate page is determined as for a search, and the entry is deleted (with ONLY overflow pages removed when becoming empty)

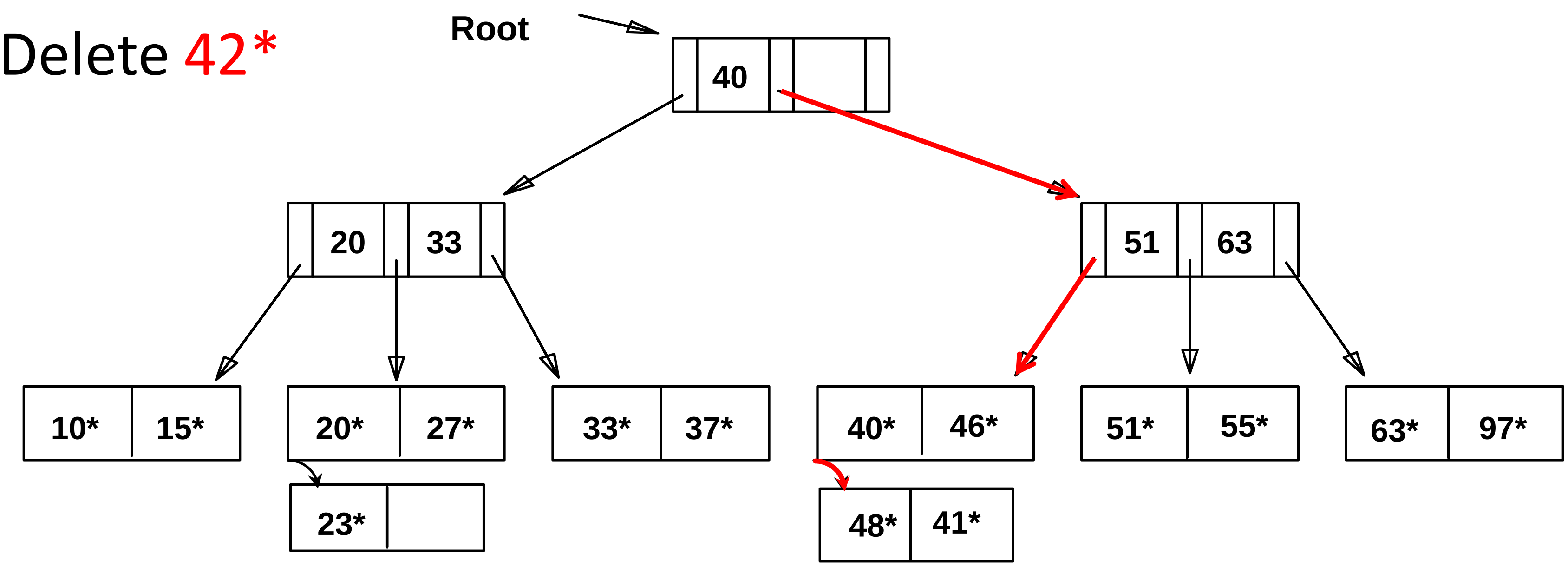
- Delete 42*



ISAM: Deleting Entries

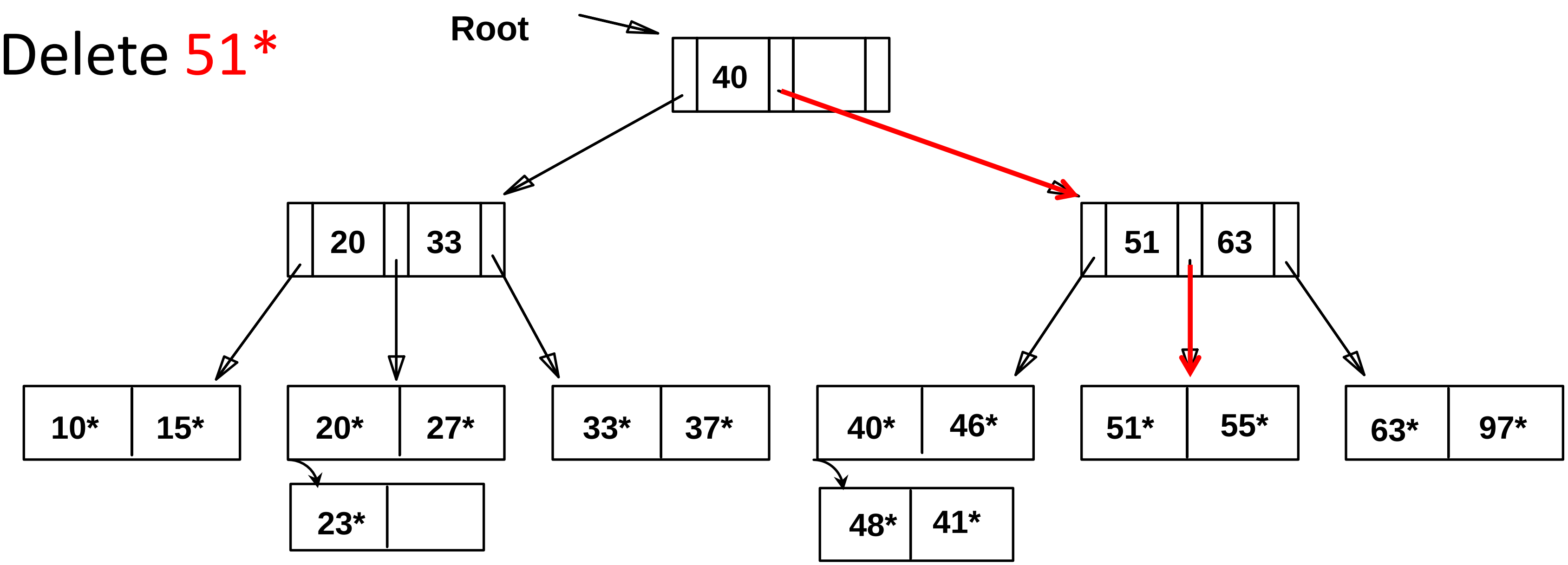
- The appropriate page is determined as for a search, and the entry is deleted (with ONLY overflow pages removed when becoming empty)

- Delete 42*



ISAM: Deleting Entries

- The appropriate page is determined as for a search, and the entry is deleted (with ONLY overflow pages removed when becoming empty)
- Delete 51*

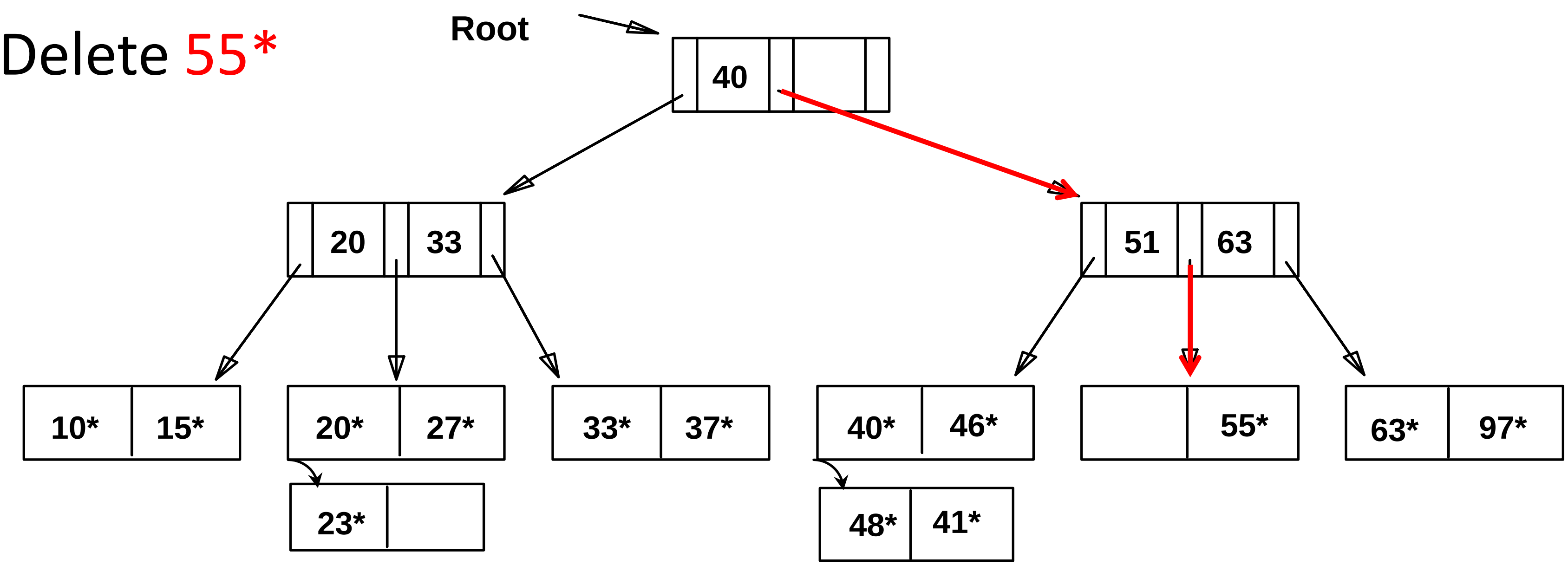


Note that 51 still appears in index levels, but not in leaf!

ISAM: Deleting Entries

- The appropriate page is determined as for a search, and the entry is deleted (with ONLY overflow pages removed when becoming empty)

- Delete 55*

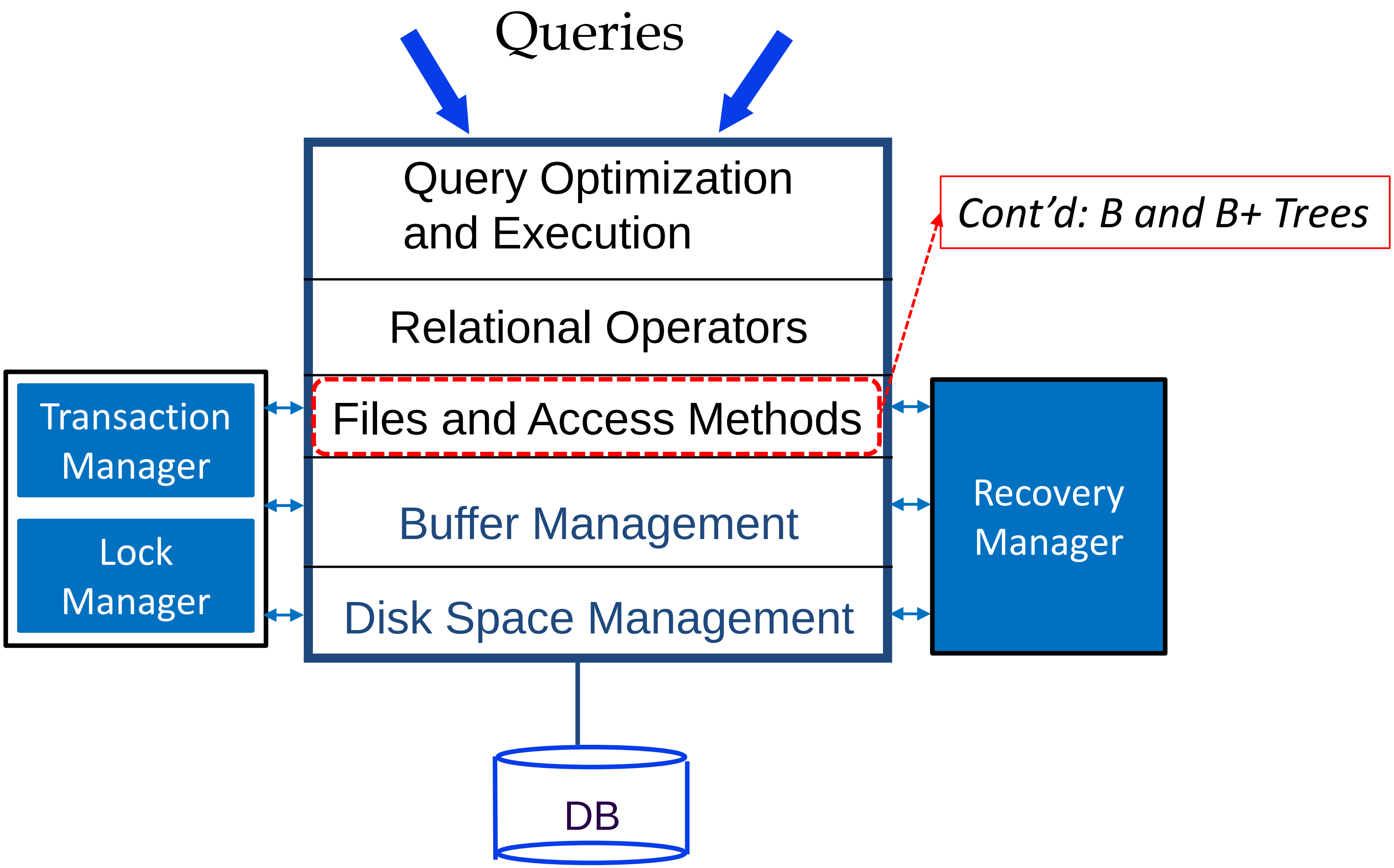


Primary pages are NOT removed, even if they become empty!

ISAM: Some Issues

- Once an ISAM file is created, insertions and deletions affect only the contents of leaf pages (i.e., ISAM is a *static* structure!)
- Since index-level pages are *never* modified, there is no need to *lock* them during insertions/deletions (critical for concurrency!)
- Long overflow chains can develop easily
  - The tree can be initially set so that ~20% of each page is free
- If the data distribution and size are relatively static, ISAM might be a good choice to pursue!

Next Class



Queries
Query Optimization
and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
Transaction
Manager
Lock
Manager
Recovery
Manager
DB
Cont'd: B and B+ Trees