

# CSE462/562: Database Systems (Spring 23)

## Lecture 5: Access Methods and Indexing

### 2/14/2023

# Revisit of the big picture of file organization

---

- Fields → Records → Pages → Heap Files (→ Files on File System) → Storage Device
- What we support:
  - Insert a record -- easy
  - Update/delete of a record with known record ID
  - Enumerating all data records
- How do I find the student with name “Alice”?
  - Linear search for record ID
- Can we do better?
  - Binary search? Binary tree? B-tree? Hash table? Partitioning
  - Do we always need to store records as a whole?
  - Needs alternative file organization
- Sometimes also called access methods (a name that comes from mainframe OS)
  - data structures and algorithms for sequentially or randomly retrieving data by keys

# Other access methods

---

- Heap file: unordered, good for enumerating all records
- Sorted file: best for random retrieval by *search key* and/or in search key order
  - A *search key* is a set of attributes (can be a single attribute) that the underlying file is sorted w.r.t.
    - Has nothing to do with (primary/candidate) keys.
    - Implication: there may be *duplicate* keys in a sorted file
  - Must be based on files that support random access of data pages with consecutive page numbers
    - e.g., for a sorted file with M pages, page numbers are 0, 1, 2, ..., M-1.
    - Need support for random access of page i efficiently without linear traversal
- Compare the costs of record insertion/deletion/search in sorted file vs heap file?

# Cost Model for Analysis

---

- We assume fixed-length records and ignore CPU costs for simplicity:
  - **N**: the number of records
  - **B**: Number of records per page
  - **T**: Number of matching record in a search
  - Cost model: # of I/Os (also ignoring pre-fetching and/or random vs sequential access), and thus even I/O cost is loosely approximated.
  - Average-case analysis (**unless o/w specified**); based on several simplistic assumptions.
    - Good enough for knowing the overall trends.
    - Reality is a lot messier than this.
- Additional assumptions
  - Single record to insert and delete; **unless o/w specified**
  - Equality selection - exactly one match; **unless o/w specified**
  - Heap Files:
    - Insert always appends to end of file.
  - Sorted Files:
    - Two alternatives:
      - No need to compact the file after deletions.
      - Files compacted after deletions.
    - Selections on search key (the attribute(s) used for sorting).

# Cost of operations

**N:** Number of records  
**N/B:** The number of data pages  
**B:** Number of records per page  
**T:** Number of matching records

# of I/Os	Heap File	Sorted File
Scan all records	$N/B$	$N/B$
Equality Search: if we know there's only 1 matching record	$0.5N/B$ , Best Case: 1, Worst Case: $N/B$	$\log_2 (N/B)$ Best case: 1.
Range Search	$N/B$	$\log_2 (N/B) + \text{\#match pages} =$ $\log_2 (N/B) + T/B$
Insert: compact for sorted file	2	$\log_2 (N/B) + N/B$ (read + write for $0.5N/B$ pages on average)
Delete: no compact for sorted file	$0.5N/B + 1$	$\log_2 (N/B) + 1$

# Cost of operations

**N:** Number of records  
**N/B:** The number of data pages  
**B:** Number of records per page  
**T:** Number of matching records

# of I/Os	Heap File	Sorted File
Scan all records	$N/B$	$N/B$
Equality Search: if we know there's only 1 matching record	$0.5N/B$ , Best Case: 1, Worst Case: $N/B$	$\log_2 (N/B)$ Best case: 1.
Range Search	$N/B$	$\log_2 (N/B) + \text{\#match pages} =$ $\log_2 (N/B) + T/B$
Insert: compact for sorted file	2	$\log_2 (N/B) + N/B$ (read + write for $0.5N/B$ pages on average)
Delete: compact for sorted file	$0.5N/B + 1$	$\log_2 (N/B) + N/B$

# Other access methods

---

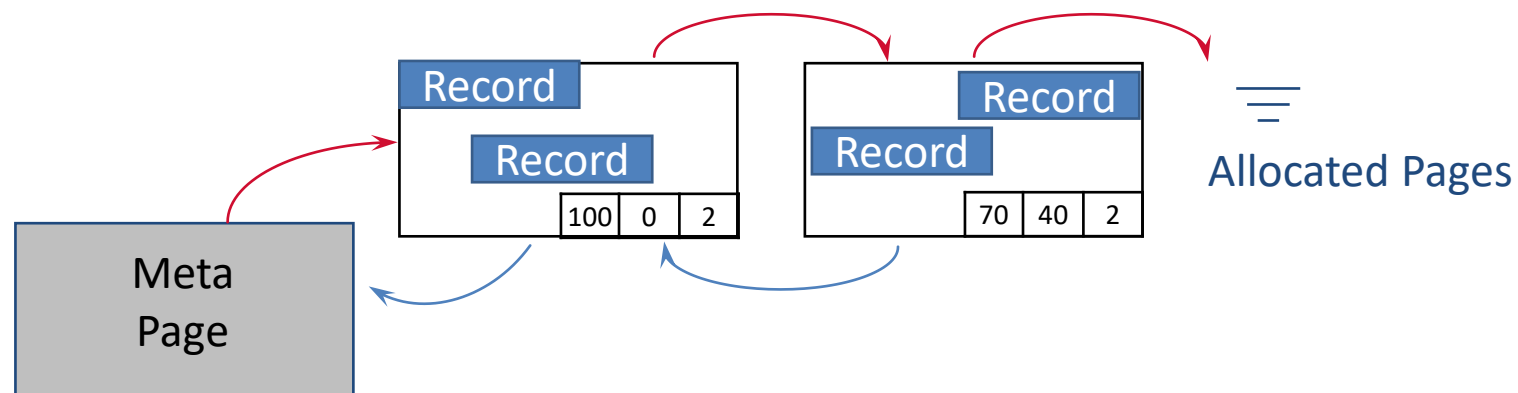
- Heap file: unordered, good for enumerating all records
- Sorted file: best for random retrieval by *search key* and/or in search key order
  - A *search key* is a set of attributes (can be a single attribute) that the underlying file is sorted w.r.t.
    - Has nothing to do with (primary/candidate) keys!
- Columnar store: store individual column/column sets in separate files
  - Also called vertical partitioning
  - Good for queries with projection -- saves I/O, SIMD friendly
- Indexes

# Index

- Sometimes, we want to retrieve records by specifying the values in one or more fields
  - Find all students in CSE
  - Find all students admitted in year 2021
- Not very efficient to handle with heap file/sorted file
  - Heap file: always need to linear scan
  - Sorted file: only (somewhat) efficient for the sorted column
    - i.e., can't use binary search on a file sorted on *major* for specific *adm\_year*

student

sid	name	major	adm_year
100	Alice	CS	2021
101	Bob	CE	2020
102	Charlie	CS	2021
103	David	CS	2020



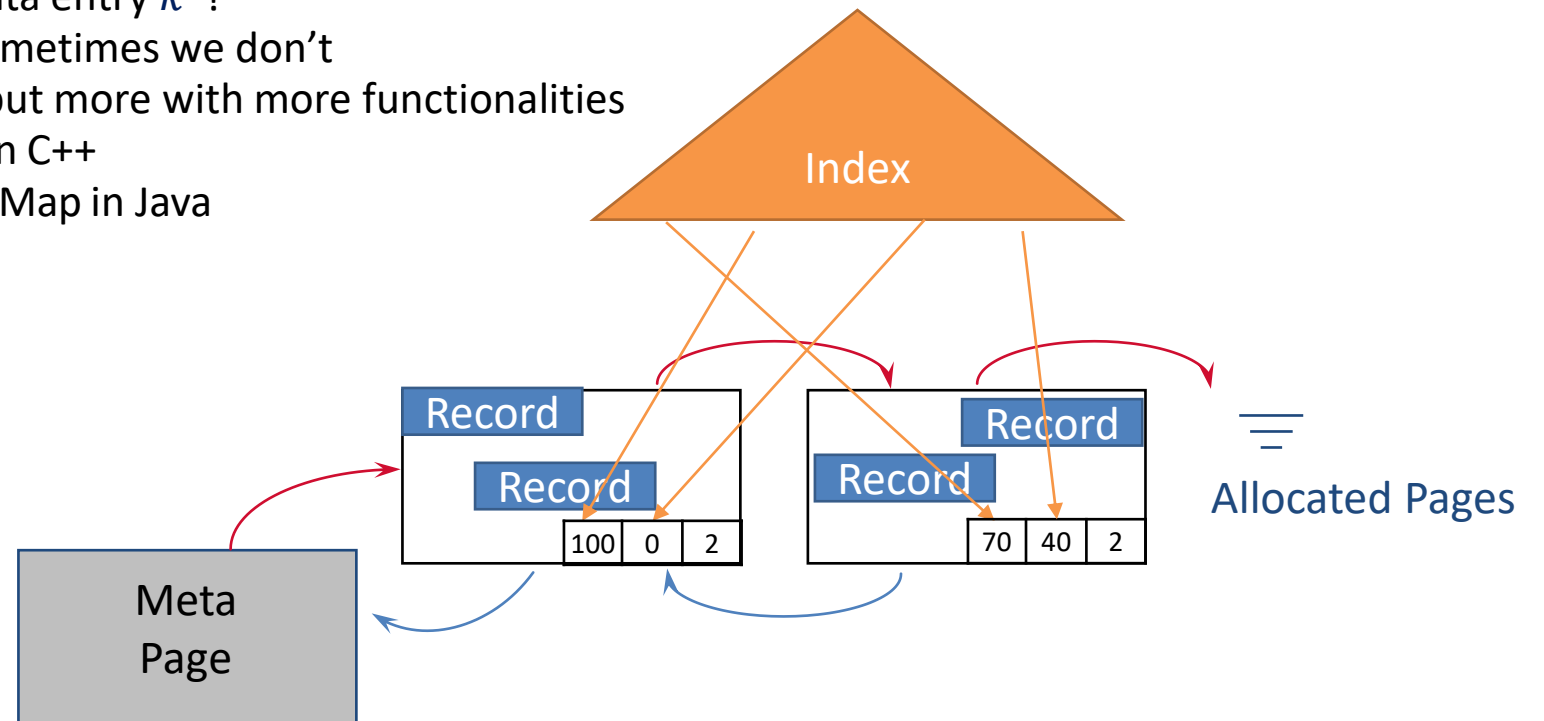


# Index

- An index: a data structure that speeds up search on a few fields on a relation
  - Maps index key  $k$  to data entry  $k^*$ 
    - Any subset of the columns of a relation can be the index key  $k$ 
      - Index key is not (candidate/primary) key; doesn't have to be unique
    - Data entry  $k^*$ 
      - e.g., the data record itself
      - Store the key  $k$  with the data entry  $k^*$ ?
        - Sometimes we do, sometimes we don't
  - Essentially an associative container, but more with more functionalities
    - `std::map/std::unordered_map` in C++
    - `java.util.TreeMap/java.util.HashMap` in Java
    - `dictionary` in Python

student

sid	name	major	adm_year
100	Alice	CS	2021
101	Bob	CE	2020
102	Charlie	CS	2021
103	David	CS	2020



# Index classification

---

- Representation of data entries in index
  - i.e., what kind of info is the index actually storing?
  - 3 alternatives
- What selections does it support
- Indexing techniques: tree/hash/other
- Lecture 10
  - Primary vs. Secondary Indexes
    - Unique indexes
  - Clustered vs. Unclustered Indexes
  - Single Key vs. Composite Indexes

# Alternatives for the data entry $k^*$ in index

---

- Three alternatives:
  - Alternative 1: the record itself (with its key  $k$ )
  - Alternative 2:  $\langle k, \text{record ID of a matching record} \rangle$
  - Alternative 3:  $\langle k, \text{list of record IDs of matching records} \rangle$
- Choice of the alternative is orthogonal to the indexing technique
  - Example of indexing techniques: B+-Tree, hash index, R-Tree, KD-Tree, and etc...
- A heap/sorted file can have multiple indexes
  - e.g., a B-tree index on `adm_year` and a hash index on `major` for the heap file of student relation
  - each usually stored as a separate file
  - usually at most *one* alternative-1 index per file (why?)

# More on the alternatives of the data entries in index

---

- Alternative 1: actual data record (with its key  $k^*$ )
  - If this is used, it is another file organization for data records (aka index file)
  - At most *one* alt-1 index
  - *Good*: avoids record id/pointer lookups
  - *Bad*: less efficient to maintain for insertion/deletion/update
- Alternative 2 & 3
  - $\langle k, \text{record id of a matching record} \rangle$  or  $\langle k, \text{list of record ids of matching records} \rangle$ 
    - *Good*: Can have multiple alt-2/alt-3 indexes
    - *Good*: more efficient to maintain than alternative 1
    - *Bad*: additional record id/pointer lookup (usually random I/O)
      - How to work around it? Include non-key columns.
  - Alt-3 is more compact than alt-2, but the variation in data entry size can be much larger
    - Harder to deal with when they need to be split/merged
  - Alt-3: key skew could lead to extremely long record id lists
    - Workaround: split them into shorter alt-3 data entries that fit into individual data pages

# Index operations

---

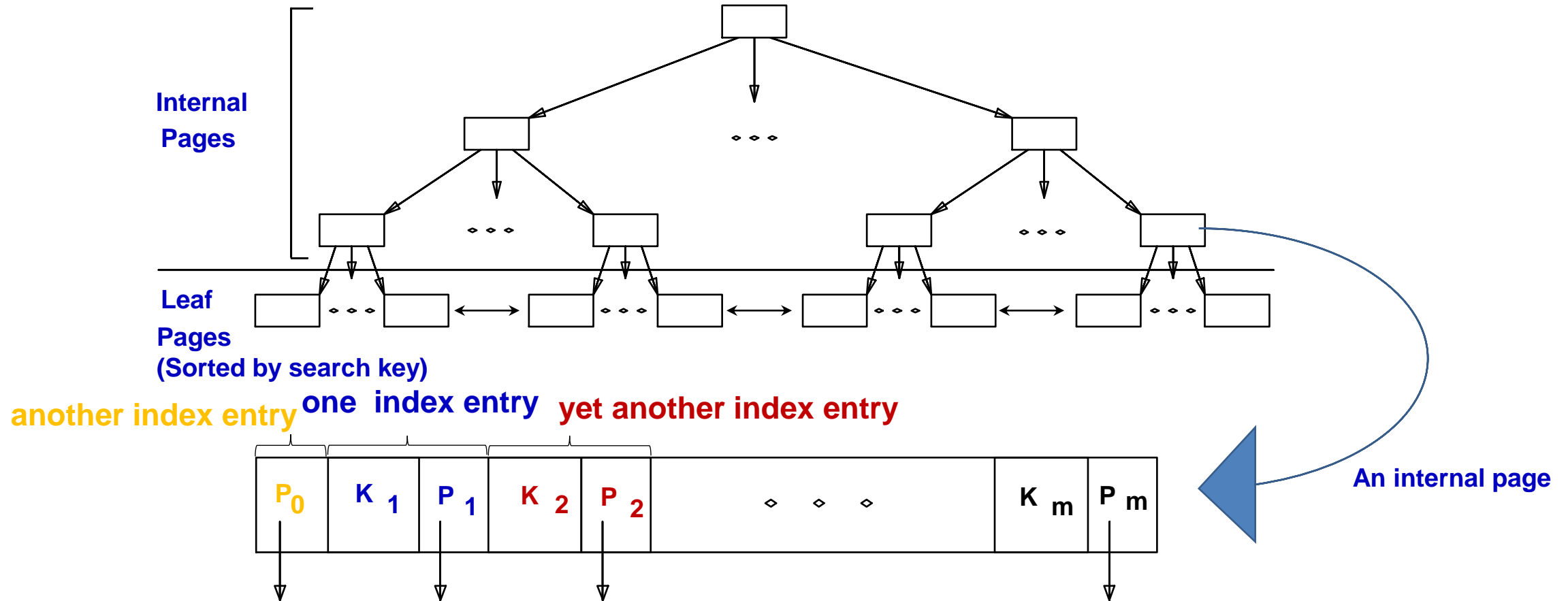
- Inserts a data entry the index
- Deletes a data entry from the index
- Updates the value of a data entry
  - Can you change the index key of a data entry?
- Search and scan
  - Point lookup: find the data entry (entries) of a *search key*
  - Range scan: enumerate all the data entries in a range of *search keys*
    - e.g.,  $\text{adm\_year} \in [2020, 2021]$ ,  $\text{adm\_year} > 2020$ ,  $\text{adm\_year} \leq 2015$
    - sometimes the search key is a subset of the index key
  - Full index scan: enumerate all data entries in an index
    - Might be useful for ordering/efficiency
  - Other search operations:
    - String prefix matching
    - 2-D, 3-D, or higher dimensional range search
    - ...

# Index Types

---

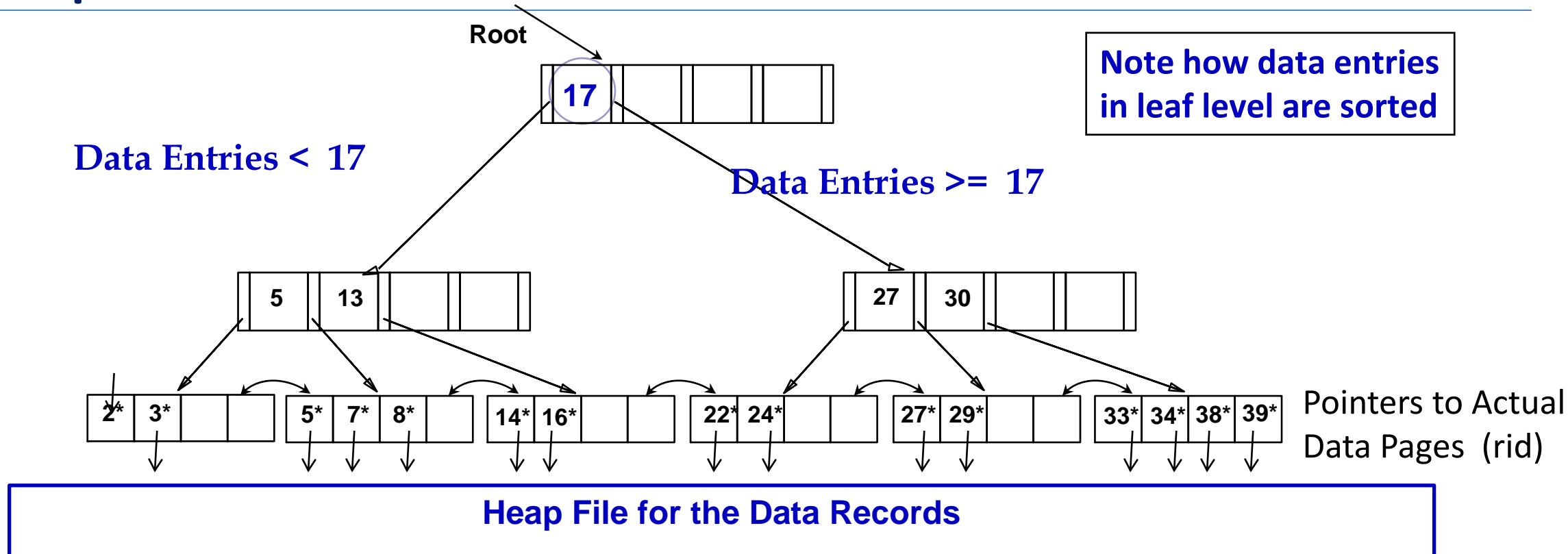
- Tree and hash indexes are the two most common categories of indexes
  - More details in the next 3-4 lectures
  - Example: B-Tree and static hash index

# Tree-based indexes



- Leaf pages contain data entries, and are chained (prev & next page ids)
- Internal pages have index entries; only used to direct searches
- Good for equality and range selection
  - Results are ordered by index key

# Example: B-Tree index



- Technically, this is the B+-Tree index, not the original B-Tree
  - Difference: B+-Tree only stores **keys** rather than **data entries** in internal nodes
  - But most DBMS uses B+-Tree, but use the term B-Tree...



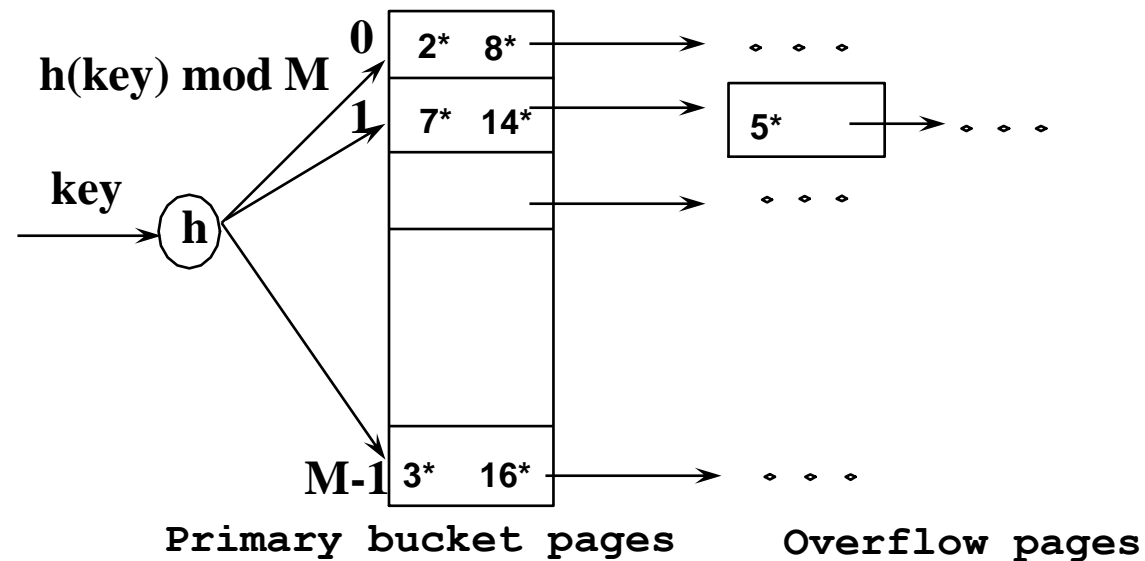
# Hash-based indexes

---

- Good for equality selections.
- Index is a collection of buckets.
  - Bucket = *primary page* plus zero or more *overflow pages*.
  - Buckets contain data entries.
- *Hashing function*  $h$ :  $h(r)$  = bucket in which (data entry for) record  $r$  belongs.  
 $h$  looks at the *index key* fields of  $r$ .
  - *No need for “index entries” in this scheme.*

# Example: static hashing index

- Fixed number of primary pages = # of buckets (denoted as **M**)
  - allocated sequentially; never de-allocated
  - allocate overflow pages if needed
- **$h(k) \% M$**  = the bucket id for a data entry with *index key*  $k$ .



# Summary

---

- This lecture
  - Access methods and indexing
  - Index classification
- HW1 solution released today
- Next time
  - Hash indexes