Review

- Suppose a doctor can work in several hospitals and receives a salary from each one. Moreover, suppose each doctor has a primary home address and several doctors can have the same primary home address. Is

  R(doctor, hospital, salary, primary_home_address)

  normalized?

- What are the functional dependencies?
  - doctor, hospital $\rightarrow$ salary
  - doctor $\rightarrow$ primary_home_address
  - doctor, hospital $\rightarrow$ primary_home_address

- The key is (doctor, hospital). Since doctor (in second FD) is a subset of the key, the table is not normalized.

- A normalized decomposition would be:
  - R1(doctor, hospital, salary)
  - R2(doctor, primary_home_address)

Disk, Storage & Access Methods

Disks and Files

- DBMS stores information on ("hard") disks.
- This has major implications for DBMS design!
  - READ: transfer data from disk to main memory (RAM).
  - WRITE: transfer data from RAM to disk.
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!

Why Not Store Everything in Main Memory?

- *Costs too much*? Not any more
  - $100 will buy you either 1 GB of RAM or 500 GB of disk today.
- *Main memory is volatile*. We want data to be saved between runs.
- *Data is also increasing at an alarming rate.*
  - "Big-Data" phenomenon
- *Memory error*
  - Larger memory means higher chances of data corruption
- Typical storage hierarchy:
  - Main memory (RAM) for currently used data.
  - SSD/Flash memory (between RAM and Disk)
  - Disk for the main database (secondary storage).
  - Tapes for archiving older versions of the data (tertiary storage).

Disks

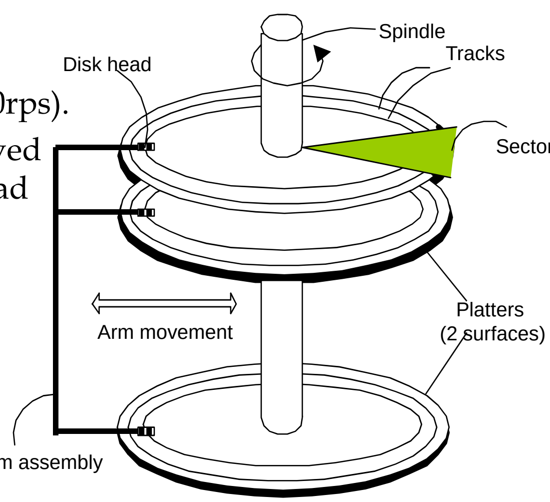
- Secondary storage device of choice.
- Main advantage over tapes: *random access* vs. *sequential*.
- Data is stored and retrieved in units called *disk blocks* or *pages*.
- Unlike RAM, time to retrieve a disk page varies depending upon location on disk.
  - Therefore, relative placement of pages on disk has major impact on DBMS performance!

Components of a Disk

The platters spin (say, 120rps).

The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a *cylinder* (imaginary!).

Only one head reads/writes at any one time.



❖ *Block size* is a multiple of *sector size* (which is fixed).

Accessing a Disk Page

- Time to access (read/write) a disk block:
  - *seek time* (moving arms to position disk head on track)
  - *rotational delay* (waiting for block to rotate under head)
  - *transfer time* (actually moving data to/from disk surface)
- Seek time and rotational delay dominate.
  - Seek time varies from about 0.3 to 10msec
  - Rotational delay varies from 0 to 4msec
  - Transfer rate is about 0.08msec per 8KB page
- Key to lower I/O cost: reduce seek/rotation delays!

Improving Access Time of Secondary Storage

- Organization of data on disk
- Disk scheduling algorithms
- Multiple disks or Mirrored disks
- Prefetching and large-scale buffering
- Algorithm design

An Example

- How long does it take to read a 2,048,000-byte file that is divided into 8,000 256-byte records assuming the following disk characteristics?

average seek time	18 ms
track-to-track seek time	5 ms
rotational delay	8.3 ms
maximum transfer rate	16.7 ms/track
bytes/sector	512
sectors/track	40
tracks/cylinder	11
tracks/surface	1,331

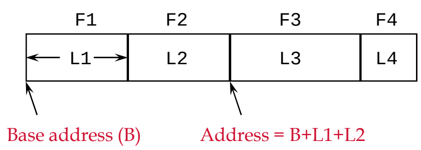
- 1 track contains 40*512 = 20,480 bytes, the file needs 100 tracks (~10 cylinders).

Design Issues

- Randomly store records
  - suppose each record is stored randomly on the disk
  - reading the file requires 8,000 random accesses
  - each access takes 18 (average seek) + 8.3 (rotational delay) + 0.4 (transfer one sector) = 26.7 ms
  - total time = 8,000*26.7 = 213,600 ms = 213.6 s
- Store on adjacent cylinders
  - read first cylinder = 18 + 8.3 + 11*16.7 = 210 ms
  - read next 9 cylinders = 9*(5+8.3+11*16.7) = 1,773 ms
  - total = 1,983 ms = 1.983 s
- Blocks in a file should be arranged sequentially on disk to minimize seek and rotational delay.

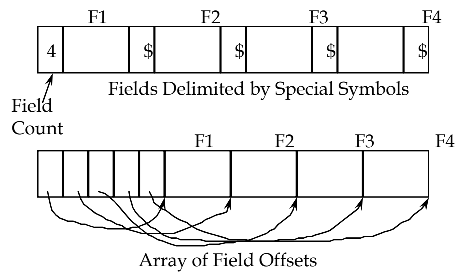
Record Formats

Fixed Length



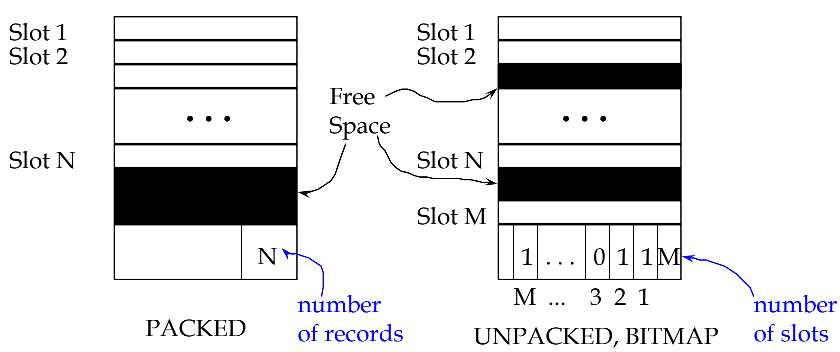
- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i*th field requires scan of record.

Variable Length: Two formats



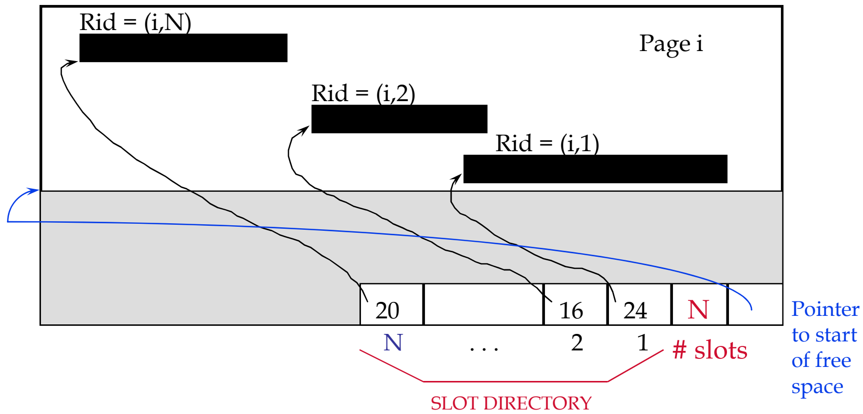
- Second offers direct access to i'th field, efficient storage of *nulls*; small directory overhead.

Page Formats: Fixed Length Records



- *Record id = <page id, slot #>. In first alternative, moving records for free space management changes rid; may not be acceptable.*

Page Formats: Variable Length Records



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

Files of Records

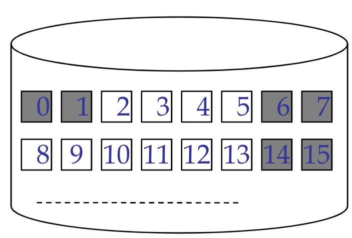
- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.
- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records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

Disk Space Management

- Many files will be stored on a single disk
- Need to allocate space to these files so that
  - disk space is effectively utilized
  - files can be quickly accessed
- Two issues
  - management of free space in a disk
    - system maintains a *free space list* -- implemented as *bitmaps or link lists*
  - allocation of free space to files
    - granularity of allocation (blocks, clusters, extents)
    - allocation methods (*contiguous, linked*)

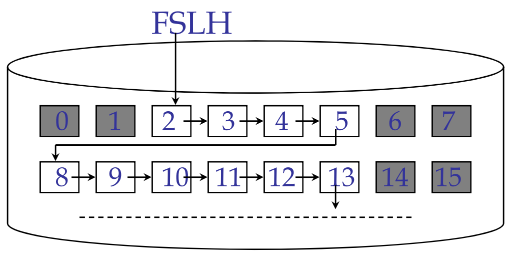
Bitmap

- each block (one or more pages) is represented by one bit
- a bitmap is kept for all blocks in the disk
  - if a block is free, its corresponding bit is 0
  - if a block is allocated, its corresponding bit is 1
- to allocate space, scan the map for 0s
- consider a disk whose blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, etc. are free. The bitmap would be
  - 110000110000001...



Link Lists

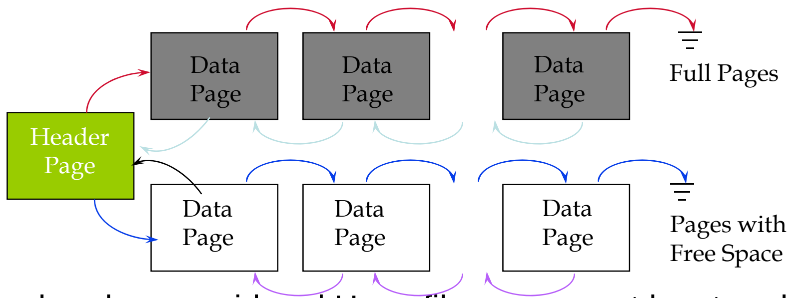
- link all the free disk blocks together
  - each free block points to the next free block
- DBMS maintains a *free space list head* (FSLH) to the first free block
- to allocate space
  - look up FSLH
  - follow the pointers
  - reset the FSLH



Unordered (Heap) Files

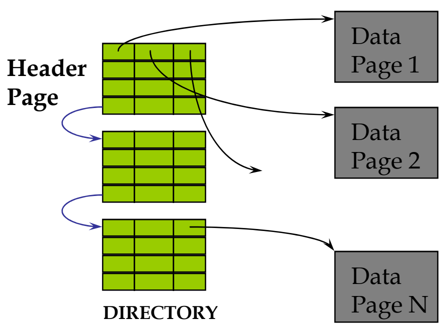
- Simplest file structure contains records in no particular order.
- As file grows and shrinks, disk pages are allocated and de-allocated.
- To support record level operations, we must:
  - keep track of the *pages* in a file
  - keep track of *free space* on pages
  - keep track of the *records* on a page
- There are many alternatives for keeping track of this.
  - We'll consider 2

Heap File Implemented as a List



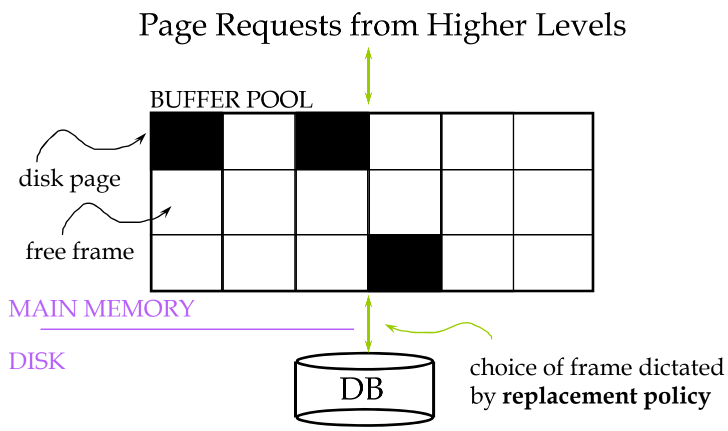
- The header page id and Heap file name must be stored someplace.
  - Database "catalog"
- Each page contains 2 `pointers' plus data.

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 HF pages*!

Buffer Management in a DBMS



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

When a Page is Requested ...

- If requested page is not in pool:
  - Choose a frame for *replacement*
  - If frame is dirty, write it to disk
  - Read requested page into chosen frame
- *Pin* the page and return its address.

If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time!

Access Methods

"If you don't find it in the index, look very carefully through the entire catalogue."

-- Sears, Roebuck, and Co., Consumer's Guide, 1897

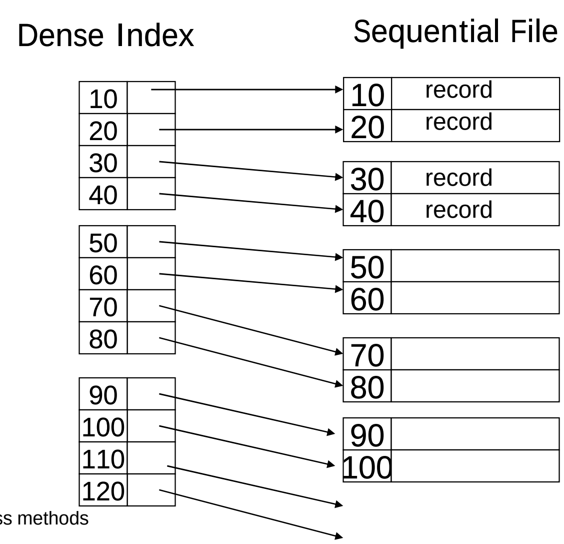
Single Record and Range Searches

- Single record retrievals
  - ``*Find student name whose matric# = 921000Y13*''
- Range queries
  - ``*Find all students with cap > 3.2*''
- Sequentially scanning the file is costly
- If data is in sorted file, do binary search to find first such student, then scan to find others.
  - cost of binary search can still be quite high.

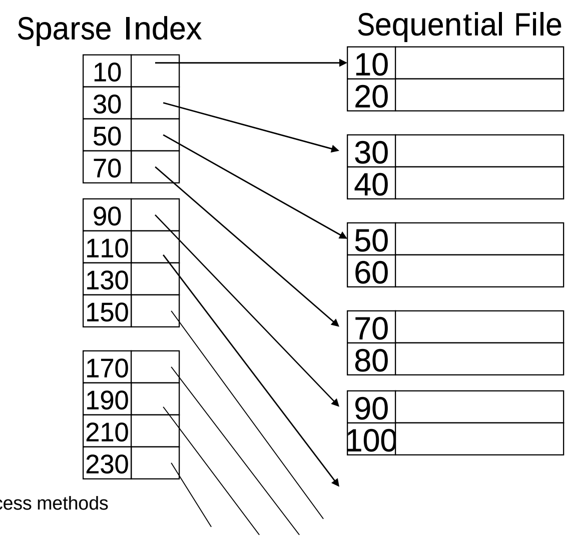
Indexes

- 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).
    - e.g., consider Student(matric#, name, addr, cap), the key is matric#, but the search key can be matric#, name, addr, cap or any combination of them
  - For each search key, you build an index

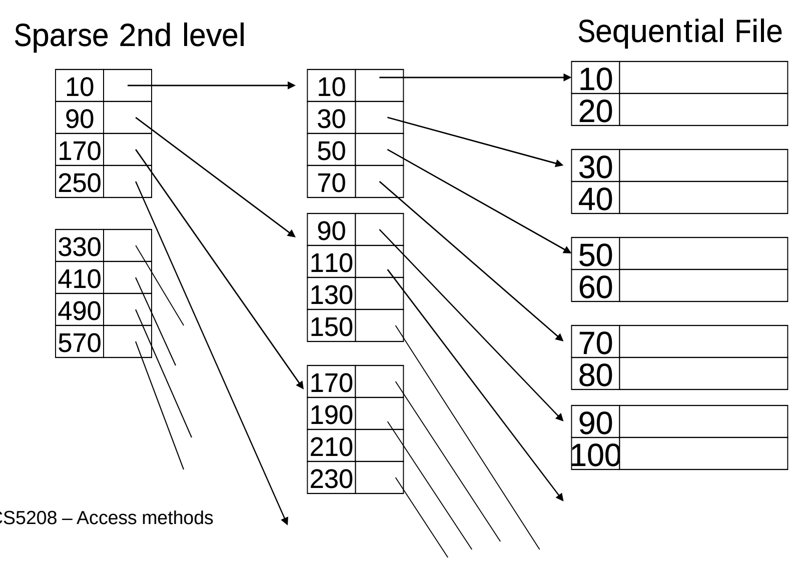
Simple Index File (Data File Sorted)



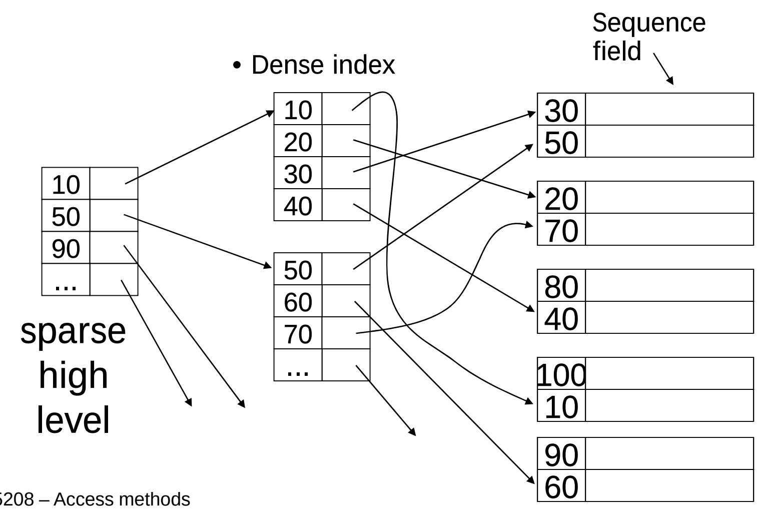
Simple Index File (Cont)



Simple Index File (Cont)



Secondary indexes



CS5208 – Access methods 29

Conventional indexes

Advantages:

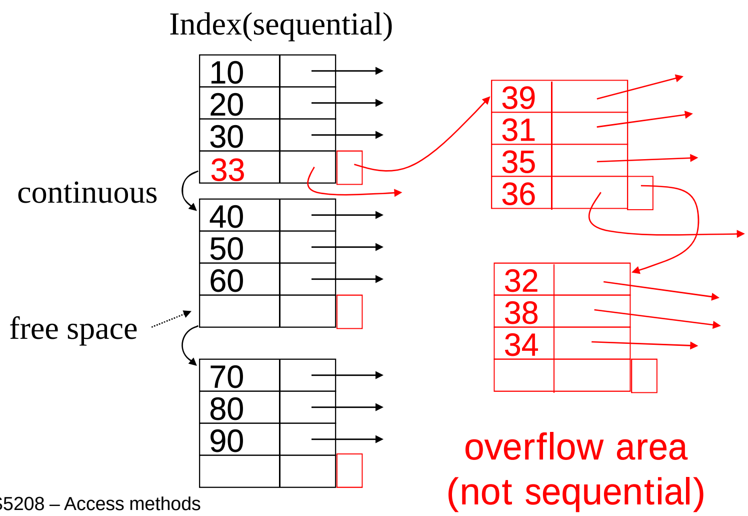
- Simple
- Index is sequential file
- Good for scans

Disadvantages:

- Inserts expensive, and/or
- Lose sequentiality & balance

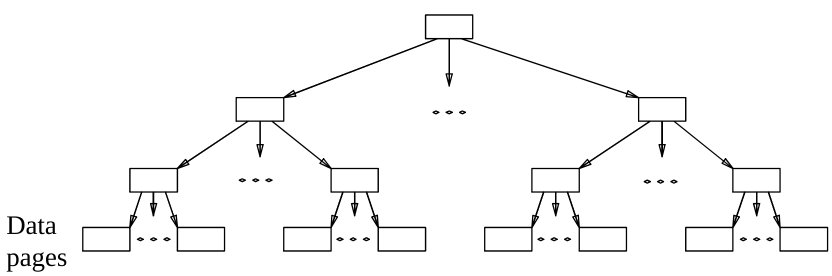
CS5208 – Access methods 30

Example



Tree-Structured Indexing

- Tree-structured indexing techniques support both *range searches* and *equality searches*

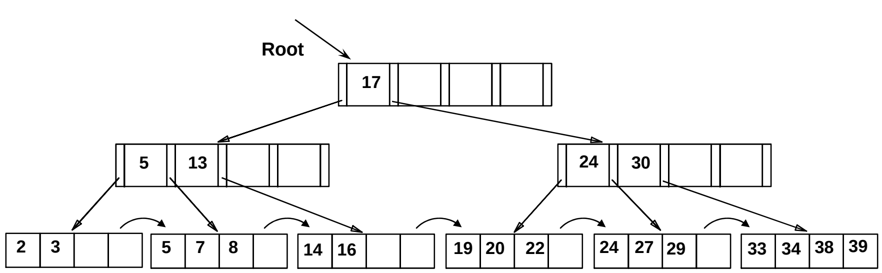


B^+ Tree: The Most Widely Used Index

- *Height-balanced.*
  - Insert/delete at $\log_F N$ cost (F = fanout, N = # leaf pages);
- Grow and shrink dynamically.
- Minimum 50% occupancy (except for root).
  - Each node contains **d** <= *m* <= 2**d** entries. The parameter **d** is called the *order* of the tree.
  - *Order* (**d**) concept replaced by physical space criterion in practice (`*at least half-full*').
- `next-leaf-pointer' to chain up the leaf nodes.
- Data entries at leaf are sorted.

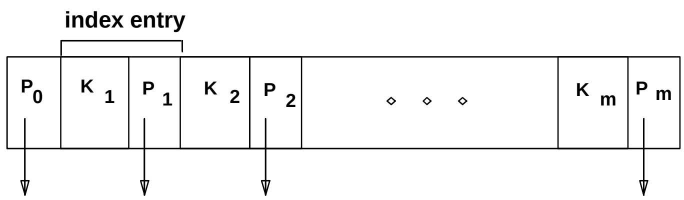
Example B+ Tree

- Each node can hold 4 entries (order = 2)

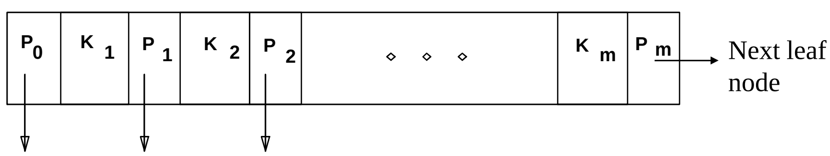


Node structure

- Non-leaf nodes

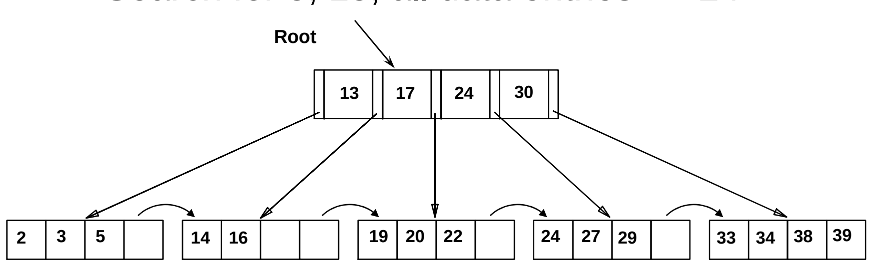


- Leaf nodes



Searching in B+ Tree

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



Based on the search for 15, we <u>know</u> it is not in the tree!*

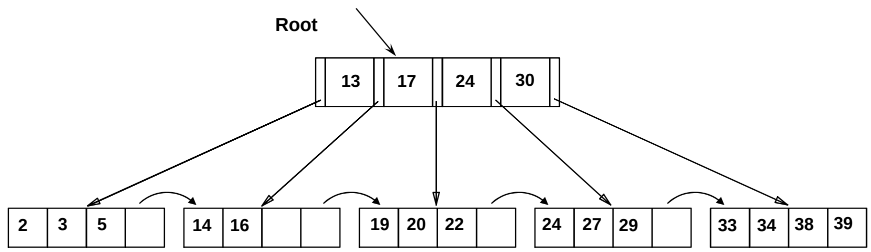
B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities (root at Level 1, and has 1 entry):
  - Level 5: 133^4 = 312,900,700 records
  - Level 4: 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

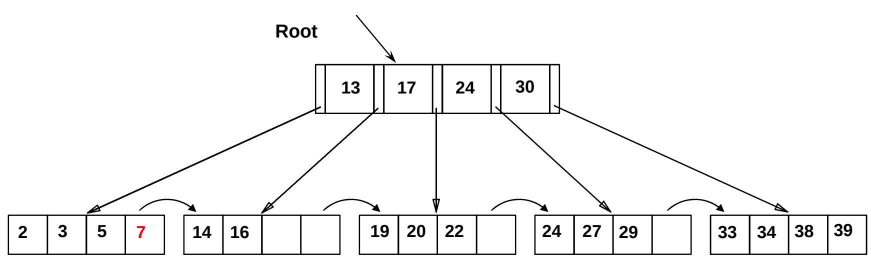
Inserting a Data Entry into a B+ Tree

- 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.*

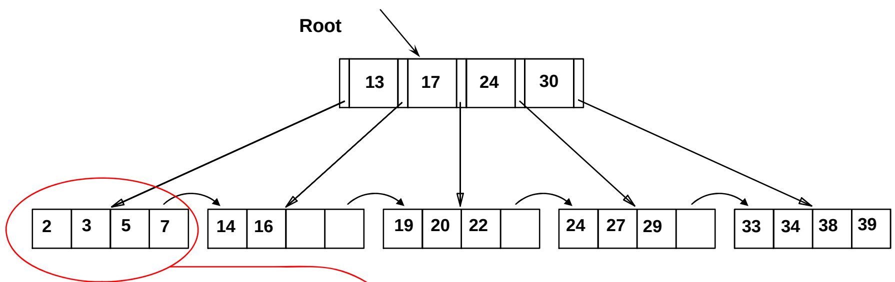
Inserting 7 & 8 into Example B+ Tree



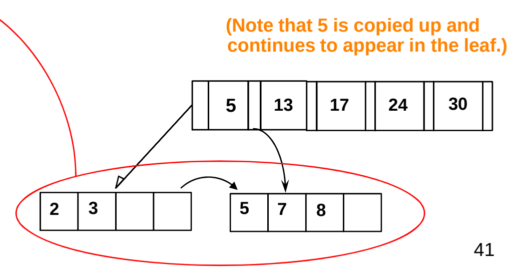
Inserting 7 & 8 into Example B+ Tree



Inserting 7 & 8 into Example B+ Tree

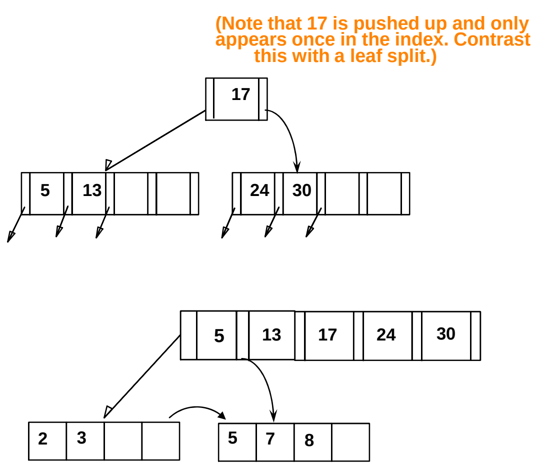


- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.

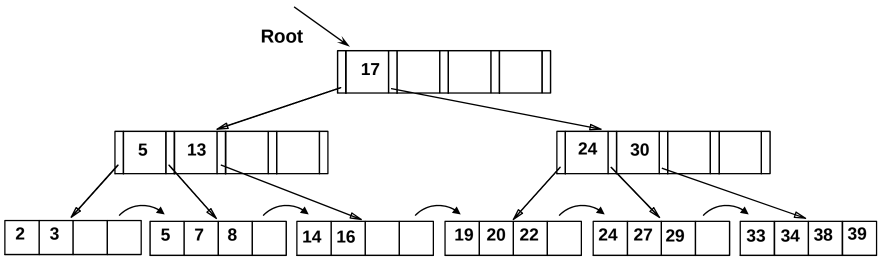


Insertion (Cont)

- Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.



Example B+ Tree After Inserting 8

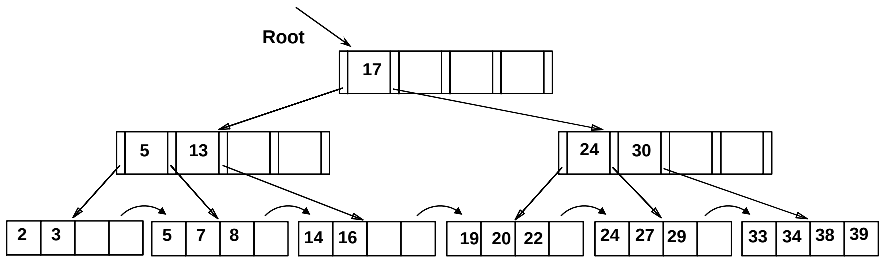


- Notice that root was split, leading to increase in height.
- In this example, we can avoid splitting by re-distributing entries; however, this is usually not done in practice. Why?

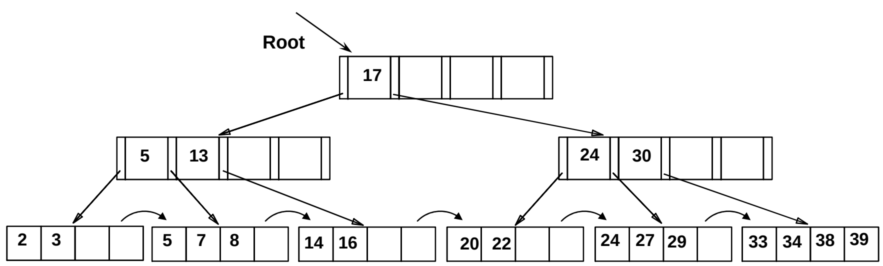
Deleting a Data Entry from a B+ Tree

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **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.

Example Tree After (Inserting 8, Then) Deleting 19

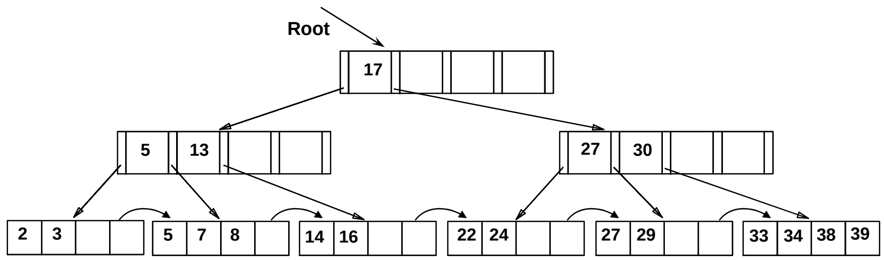


Example Tree After (Inserting 8, Then) Deleting 19



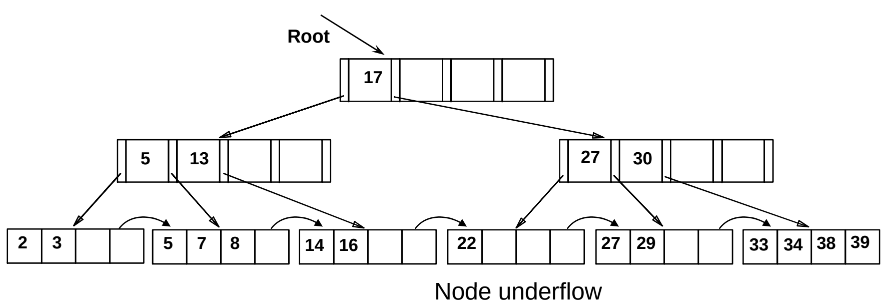
- Deleting 19 is easy.

Example Tree After Deleting 20 ...



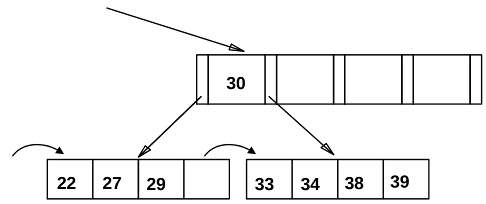
- Deleting 20 is done with re-distribution. Notice how middle key is *copied up*.

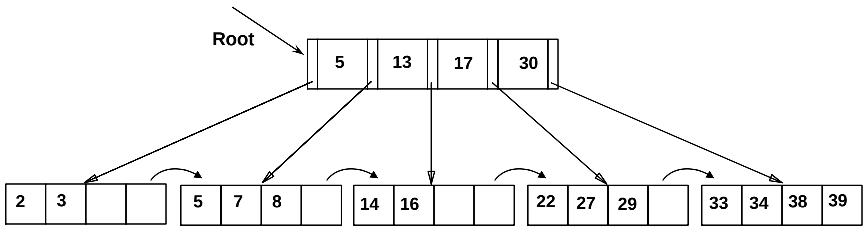
Example Tree After Deleting 24 ...



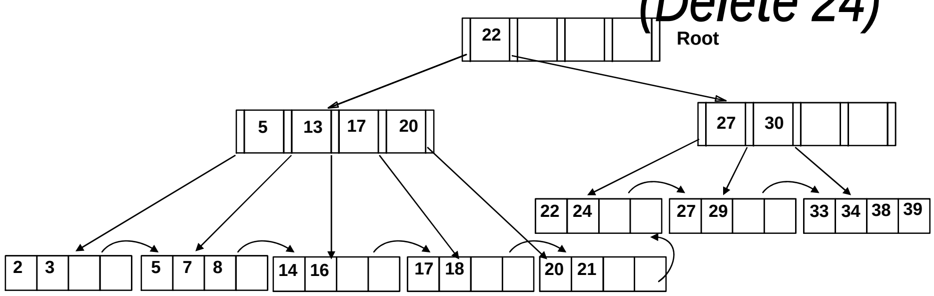
... And Then Deleting 24

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

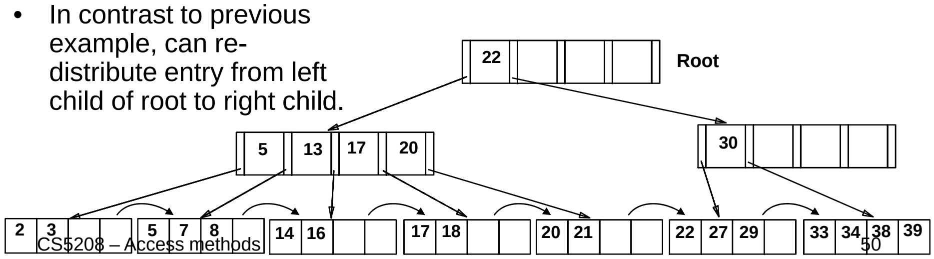




Example of Non-leaf Re-distribution (Delete 24)

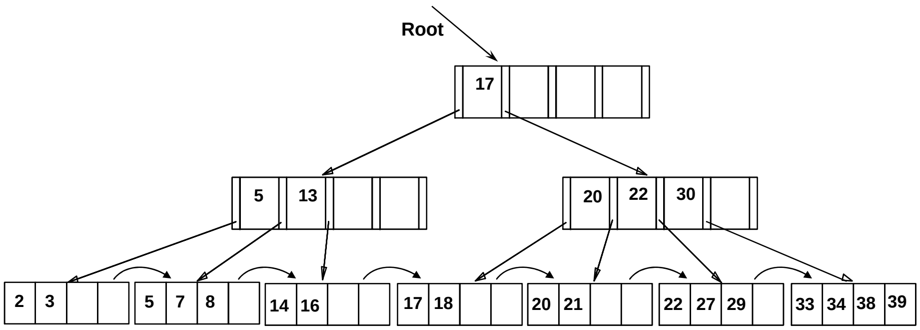


- In contrast to previous example, can re-distribute entry from left child of root to right child.



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.

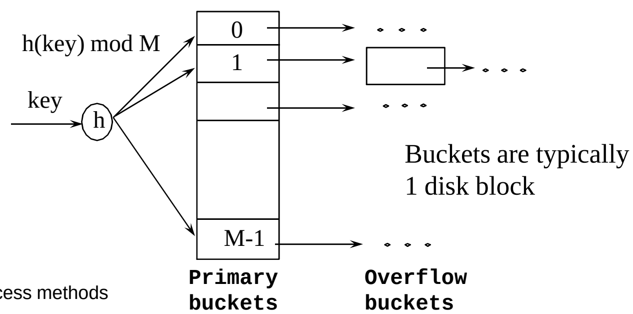


Hash-based Index

- *Hash-based* indexes
  - (Ideally) best for *equality selections*
  - Performance degenerate for skewed data distributions
  - Inefficient for range searches
    - Depends on hash function used
- Static and dynamic hashing techniques exist

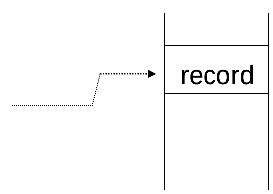
Static Hashing

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

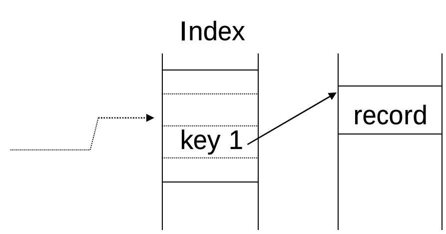


Two alternatives

(1) key $\rightarrow$ h(key)



(2) key $\rightarrow$ h(key)



- Alt (2) for “secondary” search key

Static Hashing (Cont.)

- Buckets may contain *data records or pointers.*
  - Unless otherwise stated, we assume the former.
- Hash fn works on *search key* field of record *r.* Must distribute values over range 0 ... M-1.
  - **h**(*key*) = (a * *key* + b) mod M usually works well.
    - a and b are constants
    - **h** has to be tuned for different applications.
- Long overflow chains can develop and degrade performance.
  - *Extendible* and *Linear Hashing*: Dynamic techniques to fix this problem.

Within a bucket or a chain of buckets:

- Do we keep keys sorted?
- Yes, if CPU time critical
  & Inserts/Deletes not too frequent

EXAMPLE 2 records/bucket

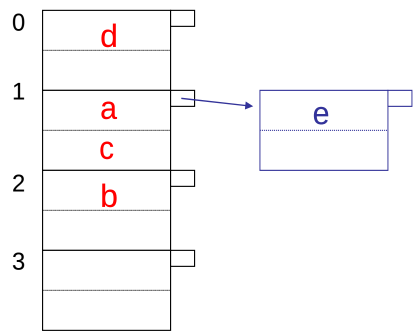
INSERT:

h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0



h(e) = 1

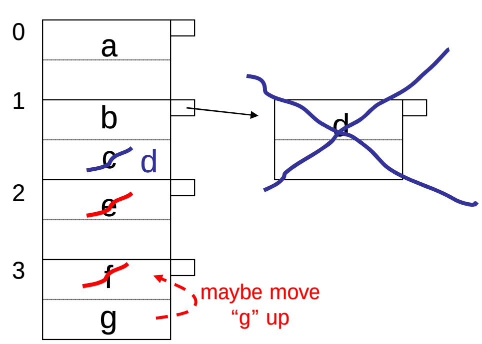
EXAMPLE: deletion

Delete:

e

f

c



Rule of thumb:

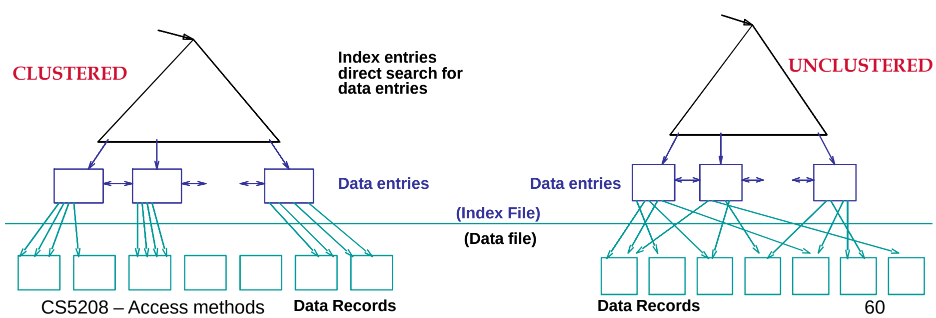
- Try to keep space utilization between 50% and 80%

  Utilization = # keys used/total # keys that fit

- If < 50%, waste space
- If > 80%, overflows significant
  - Depends on how good hash function is & on #keys/bucket
- How to cope with growth?
  - Overflows and reorganization
  - Dynamic hashing

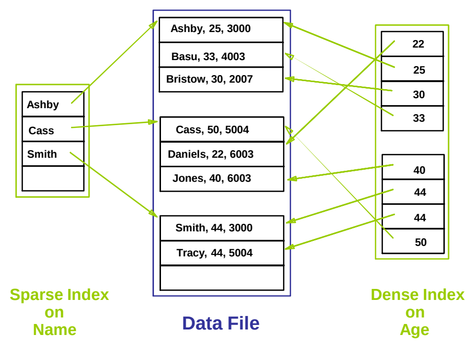
Clustered vs. Unclustered Index

- Suppose the data file is unsorted.
  - To build clustered index, first sort the data 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.)



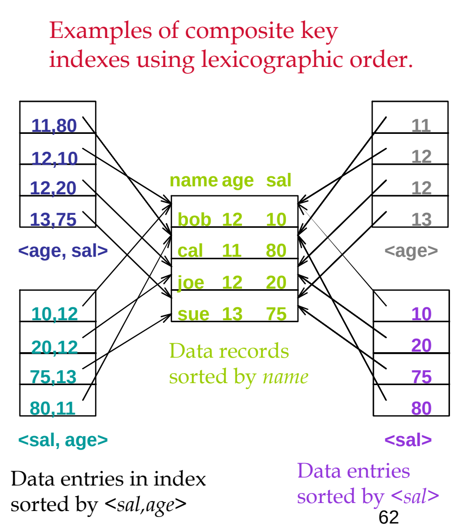
Dense vs. Sparse

- If there is at least one data entry per search key value (in some data record), then dense.
  - Every sparse index is clustered!
  - Sparse indexes are smaller.



Multi-attribute Indexes

- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=12 & sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age=12 & sal > 10 (use <age, sal>)
    - age < 12 & sal = 10 (use <age,sal> may fetch more records than desired)
- Data entries in index sorted by search key to support range queries.
  - Lexicographic order, or
  - Spatial order
- There are also multi-attribute indexing structures (e.g., R-trees)



Summary

- Is it always beneficial to use an index for data retrieval?

- Is it beneficial to build indexes on ALL attributes of a table?