Database Management System

Lecture 6

Storage and File Organization

Index

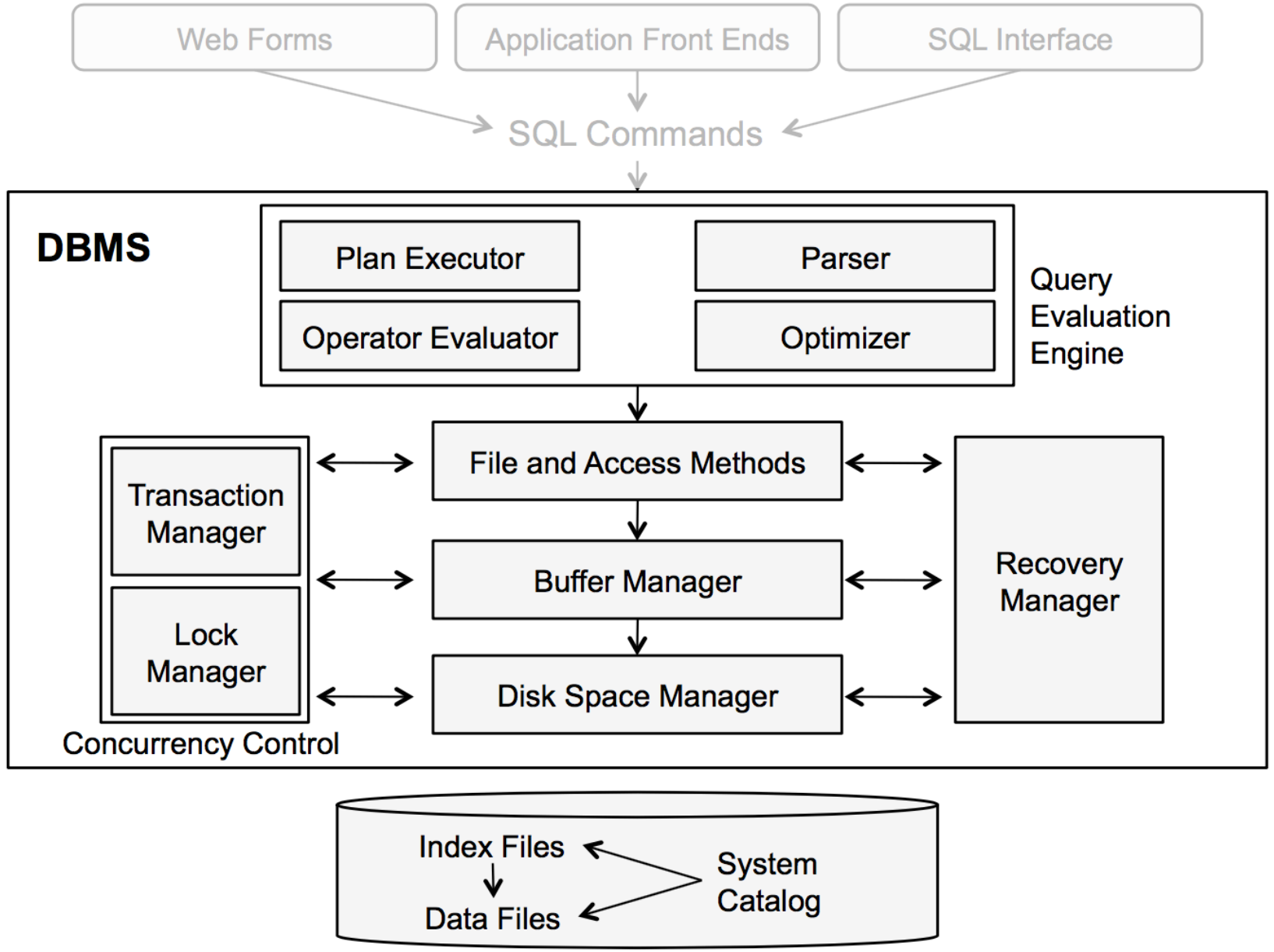
* Some materials adapted from R. Ramakrishnan, J. Gehrke and Shawn Bowers

Today's Agenda

- Storing data in the disk
- Index
  - ISAM
  - B+ tree

Storing data

Basic Database Architecture



Bird eye view on query optimization

- Given an SQL query
- Translate it into relational algebra
- Find equivalent query plans
  - different ways to order operators
  - different ways to implement each operator
- Pick a cheap plan (per estimated cost) Execute the plan ...

- We are going to learn
  - How are operators implemented?
  - How is data stored on disk?

The Plan

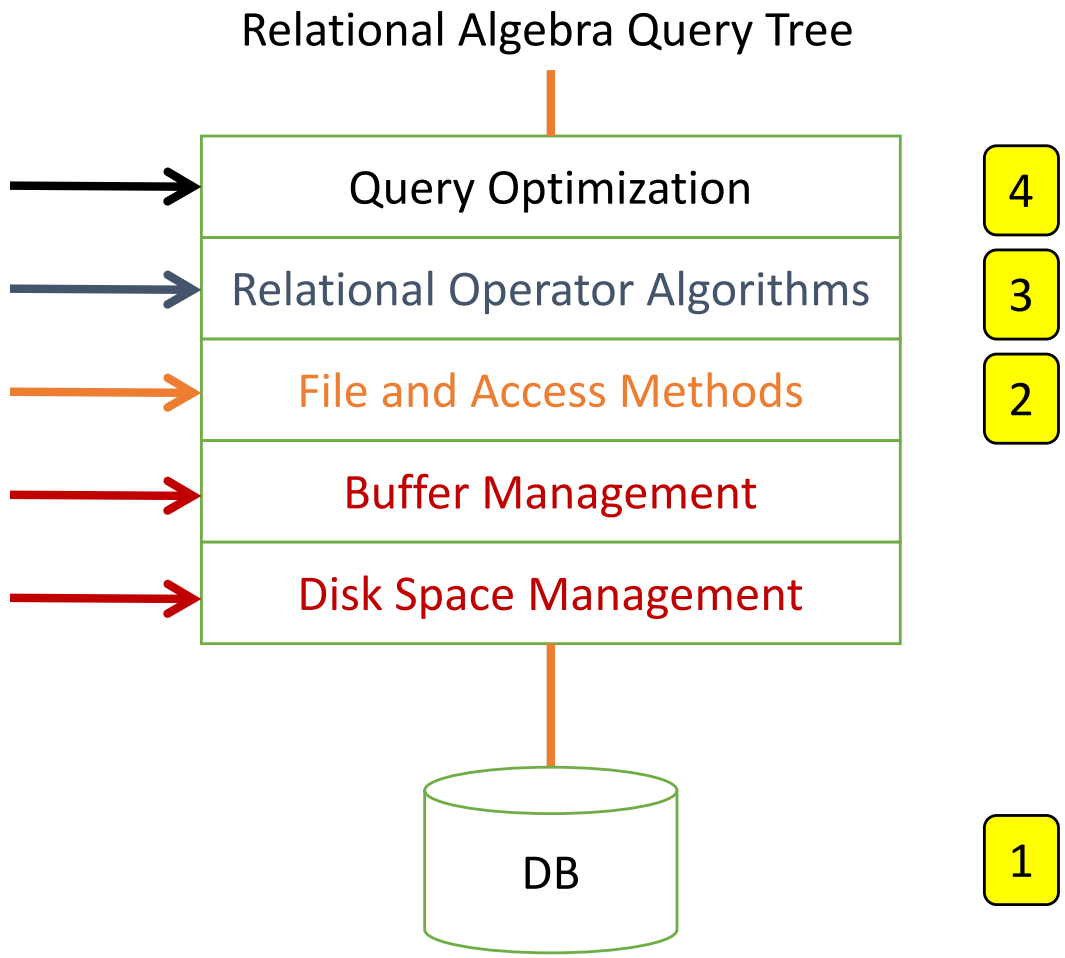
Search for a cheap plan

Join Algorithm, etc

Heap, index, etc.

O/S issues
(They may be handled by
a DBMS or by the OS)

Disk access is expensive!!



Types of Physical Storage

- Cache
  - fastest and most costly form of storage
  - volatile ... content lost if power failure, system crash, etc.
  - managed by the hardware and/or operating system
- Main Memory
  - fast access
  - in most applications, too small to store an entire DB
  - Volatile

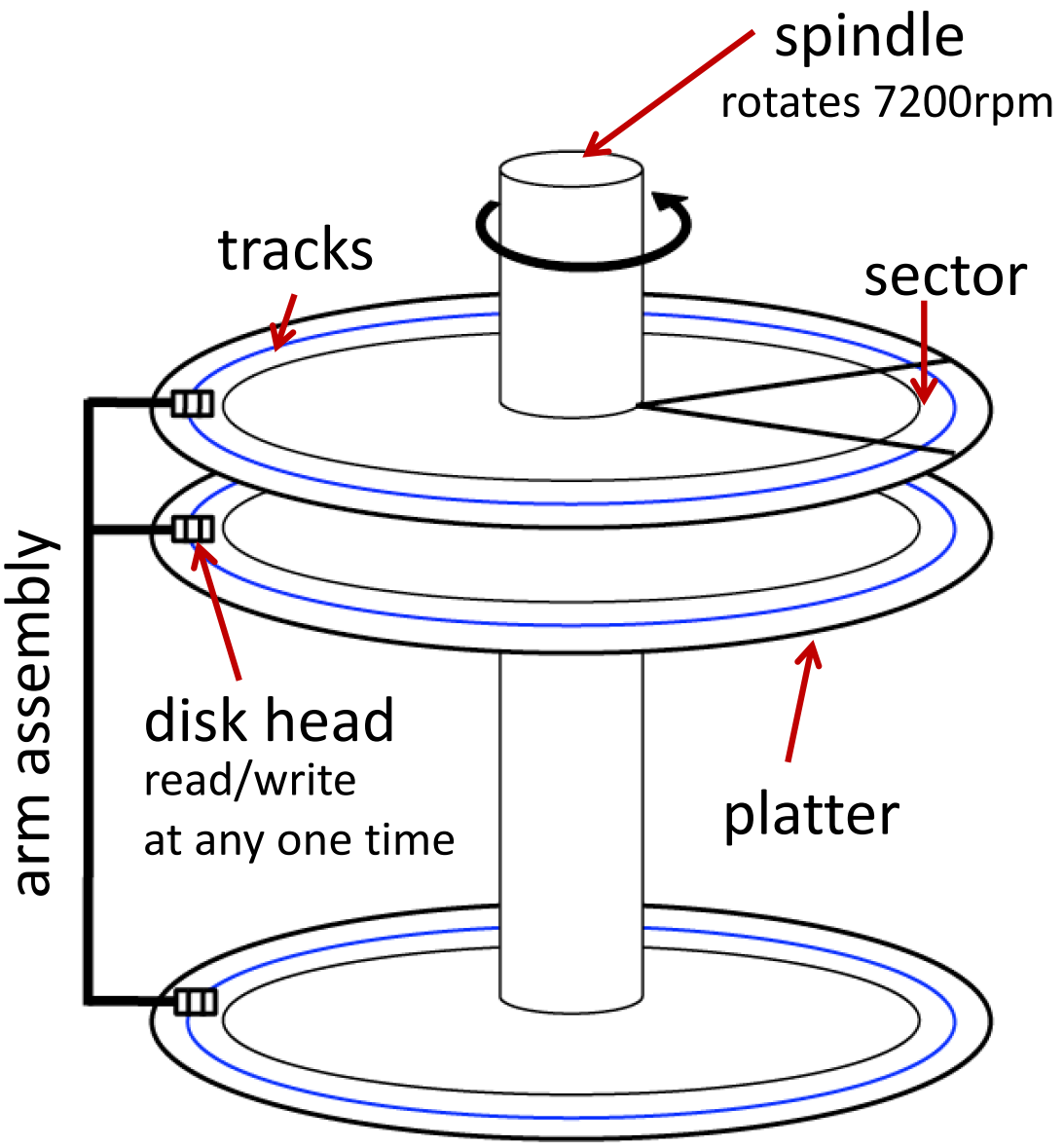
Note: many "main memory only" databases are available
... and used increasingly for applications with small storage requirements and as memory sizes increase

Types of Physical Storage

- Magnetic (“Hard”) Disk Storage
  - primary medium for long-term storage of data
  - typically can store entire database (all relations and access structures)
    - data must be moved from disk to main memory for access and written back for storage
    - direct-access, i.e., it is possible to read data on disk in any order
    - usually survives power failures and system crashes (disk failure can
    - occur, but less frequently) We focus on disk storage!

- We focus on disk storage!

Components of a Disk



To read/write a data from the disk

- Position arm (seek)
- wait for data to spin by
- read/write from the location

Components of a Disk

- Each track is made up of fixed size sectors
- page size is a multiple of sector size
  - unit of transfer
  - size depends on system and configuration
- all tracks that you can reach from one position of the arm is called cylinder

Cost of accessing data on a disk

- Time to access (read/write) data

  seek time = moving arms to position disk head on track
  rotational delay = waiting for sector to rotate under head
  transfer time = actually moving data to/from disk surface

- Key to lower I/O cost: *reduce seek & rotational delays!* you have to wait for the transfer time, no matter what

- Query cost often measured in number of page I/Os
  - often simplified to assume each page I/O costs the same
  - random I/O is more expensive than Sequential I/O

Memory Vs. Disk access Time

- Lets say disk access time (all three costs together) is about 5 milliseconds (ms)
- and memory access time is about 50 nanoseconds (ns)
  - 5 ms = 5,000,000 ns

    therefore disk access is 100,000 times slower than memory access!

Block (page) size Vs. Record Size

- The terms “block” and “page” are often used interchangeably ...
  - ... (e.g., depending on how a DBMS is implemented)
- A block generally refers to a contiguous sequence of sectors from a single track
  - unit of (physical) storage on a disk, and transfer between main memory and disk
  - a page is a “block” in logical memory ... smallest unit of transfer supported by an OS (virtual memory, paging)
  - Pages/blocks range in size (typically around 512b to 8kb)

Block (page) size Vs. Record Size

- A database system seeks to *minimize* the number of block transfers between disk and main memory
- Transfer can be reduced by keeping as many blocks as possible in main memory
  - Buffer is the portion of main memory available to store copies of disk blocks
  - Buffer manager is responsible for allocating and managing buffer space
- If possible, store file blocks sequentially:
  - Consecutive blocks on same track, followed by
  - Consecutive tracks on same cylinder, followed by
  - Consecutive cylinders adjacent to each other
  - First two incur no seek time or rotational delay, seek for third is only one track

Buffer Manager

- Program calls buffer manager when it needs blocks from disk
  - the program is given the address of the block in main memory, if it is already in the buffer
  - if block not in buffer, the buffer manager adds it ...
    - Replaces (*throws out*) other blocks to make space
    - The thrown out block is is written back to the disk if it was modified (since last write to disk)
    - Once space is allocated, the buffer manager reads in the block from disk to the buffer and returns the address

Buffer Replacement Policies

- Operating systems often replace the block *least recently used* (LRU strategy)
  - In LRU, past (use) is a predictor of future (use)
- Alternatively, queries have well-defined access patterns (e.g., sequential scans)
  - A database system can exploit user queries to predict block accesses
  - LRU can be an inefficient strategy for certain access patterns that involve (e.g., repeated) sequential scans
  - The query optimizer can provide hints on replacement strategies

Buffer Replacement Policies

Pinned block = not allowed to be written back to disk

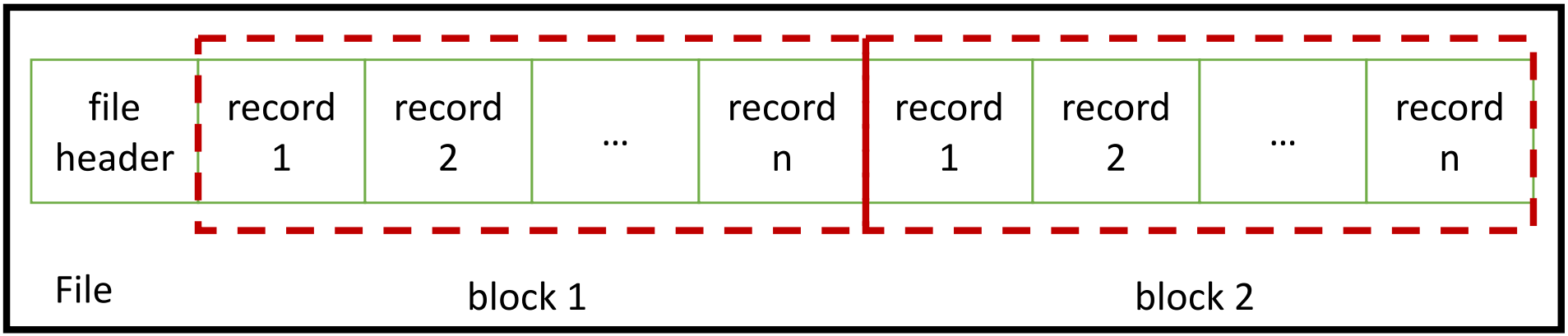
- *Most recently used* (MRU) strategy
  - Pin the block currently being processed
  - After final tuple of that block processed, the block is unpinned and becomes the most recently used block
  - Keeps older blocks around longer (good for scan problem)
- Buffer manager can use statistics regarding the probability that a request will reference a particular relation

File Organization

- A database can be stored as a collection of files
- Row-oriented storage
  - Each file is a sequence of records
  - Each record is a sequence of fields

File Organization

- Typical organization of records in files
  - Assume the record size is *fixed* (not always the case ...)
  - Each file has records of one particular type only
  - ... different files used for different relations



Fixed-Length Records

- Simple approach
  - Store record *i* starting at byte *n* * (*i* – 1), where *n* is the size of each (fixed-length) record
  - Record access is simple, but records may span blocks
- Deletion of record *i* (to avoid fragmentation)
  - Move (shift) records *i* + 1, ..., *n* to *i*, ..., *n* – 1
  - Move record *n* to *I*
  - Maintain positions of free records in a *free list*

Fixed-Length Records: Free Lists

- In the file header, store the address of the first record whose content is deleted
- Use this first record to store the address of the second available record, and so on
- These stored addresses act as “pointers” ... they “point” to the location of a record (like a linked list)
- Tricky to get right (often the case with pointers)

Variable-Length Records

- Variable-length records are often needed
  - for record types that allow a variable length for one or more fields (e.g., varchar)
  - if a file is used to store more than one relation
- Approaches for storing variable length records
- End-of-record markers
  - Fields “packed” together
  - Difficult to reuse space of deleted records (fragmentation) – No space for record to grow (e.g., due to an update)
  - ... in this case, must move the record

Variable-Length Records: Approaches

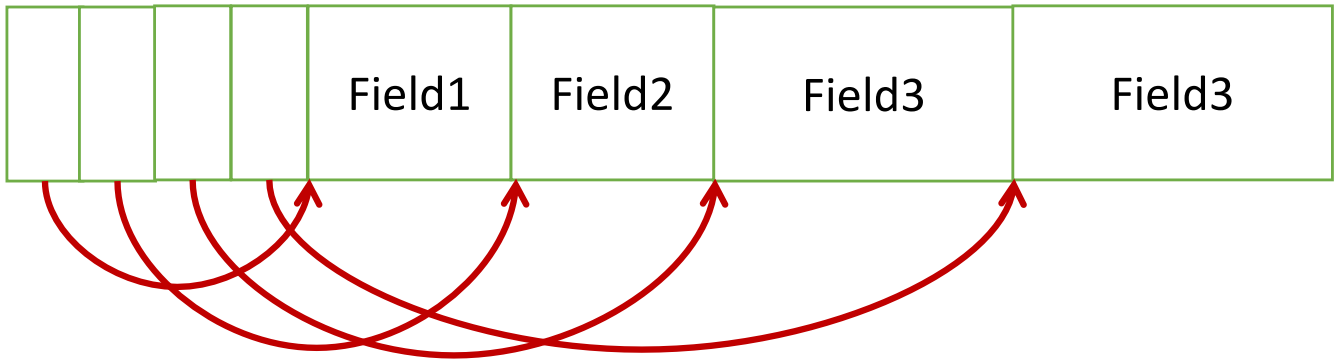
- End-of-record markers
  - Fields “packed” together
  - Difficult to reuse space of deleted records (fragmentation) – No space for record to grow (e.g., due to an update)
  - ... in this case, must move the record

Variable-Length Records: Approaches

- Field delimiters
  - Requires scan of record to get to *n*-th field value
  - Requires a field for a NULL value

Variable-Length Records: Approaches

- Each record as an array of field offsets
  - For overhead of the offset, we get direct access to any field
  - NULL values represented by assigning begin and end pointers of a field to the same address

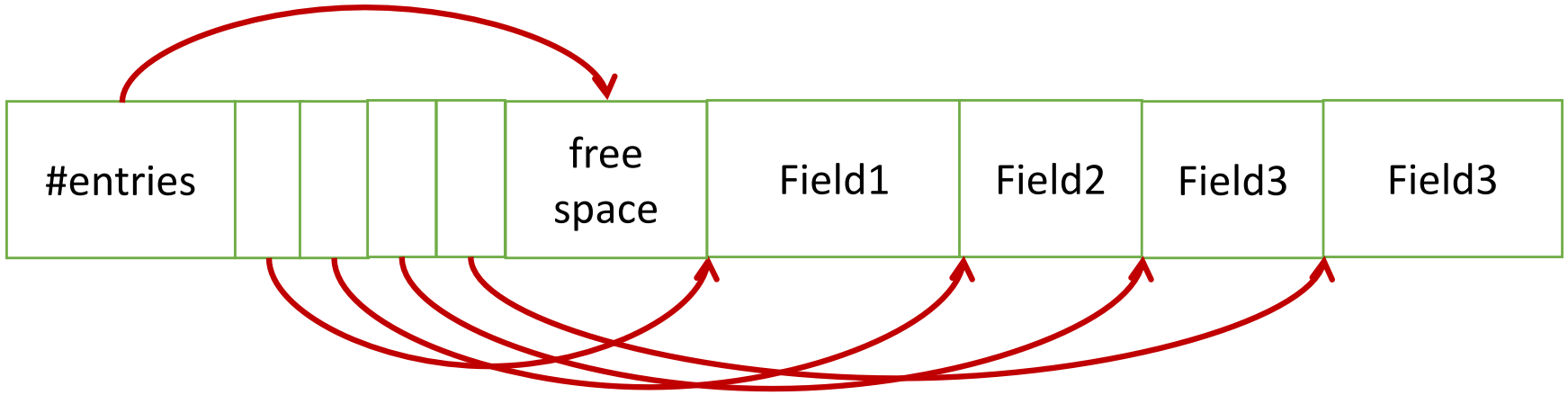


Variable-Length Records: Approaches

- Using each record as an array of field offsets can cause problems when attributes are modified
  - growth of a field requires shifting all other fields
  - a modified record may no longer fit into the block
  - a (large) record can span multiple blocks

Variable-Length Records: Approaches

- Block headers
  - maintain pointers to records
  - contain pointers to free space area
  - records inserted from end of the block
  - records can be moved around to keep them contiguous



Blocks within a file

- Heap File (unsorted file)
  - Most simple file structure
  - Records are unordered
  - Record can be placed anywhere in the file where space – Useful when scanning is the main operation
- Sequential File
  - Records are ordered according to a search key
- Hash File
  - Hash function determines which block of the file a record is placed

Indexing

What is an index?

- Lets say we have a large collection of items
  - For example, a library of “holdings”
  - A store filled with music albums (CDs)
  - Medical records in a medical office
  - All the webpages on the internet
- How can we find a particular item quickly?
  - That is, we have a “*search key*” (value entered by user)
  - We are looking for records that match the search key
- A common approach is to use some form of *indexing*

Search Keys

- What are some possible search keys for
  - Books?
  - Music Albums?
  - Medical records?
  - Web pages?

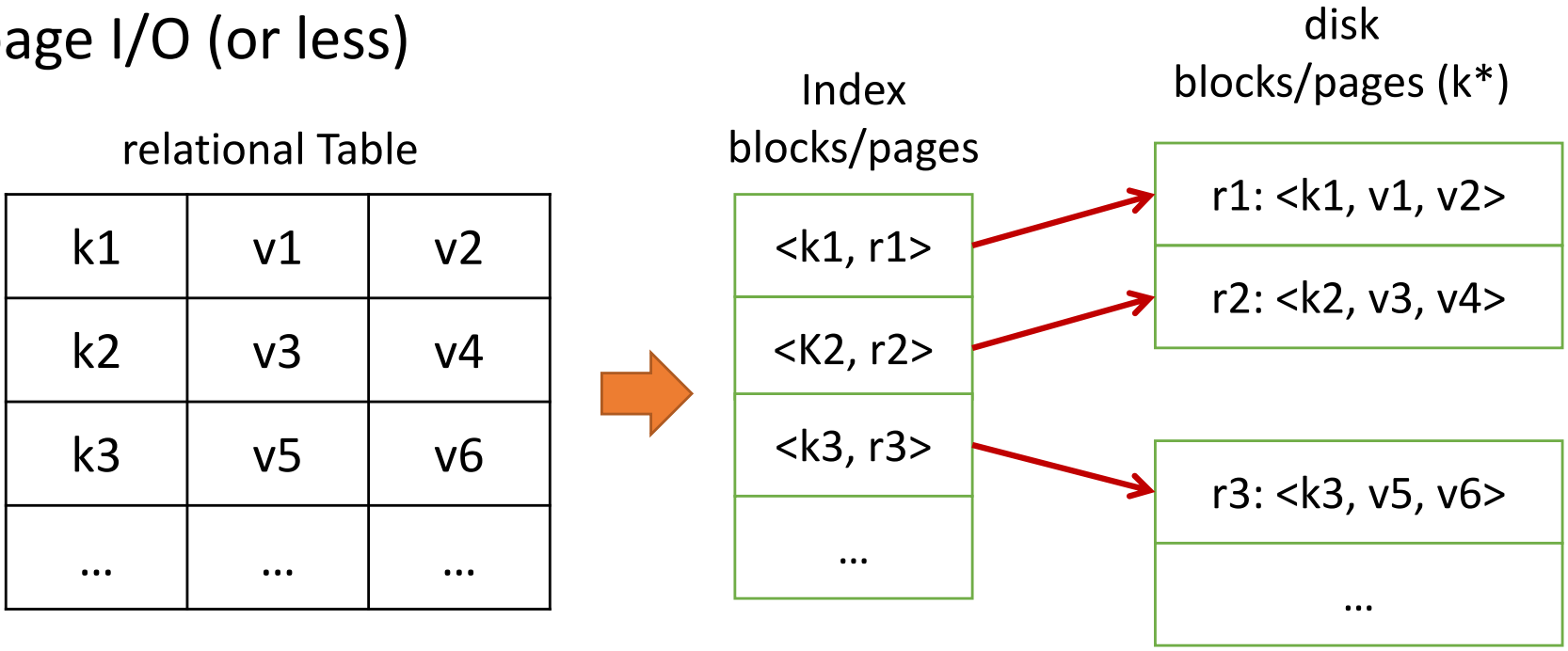
Database Indexes

- Possible Search key for Emp(ID,Name,Age,Address)

- We can build an index on any attribute
- We can build an index on any subset of attributes
  - E.g.,<Age,Name>
- We can build multiple indexes for the same table
- Can you think of a disadvantage to building many indexes?
- Note: A *Search Key* is not the same as a *Key* for a table
  - Search key values may not be unique!

Index for a File (Database Table)

- An "*index*" is a data structure that speeds up selections (searching) on the search key field(s)
- An index converts a search key *k* into a data entry *k**
- Given *k**, we can get to the record(s) with the search key *k* in one page I/O (or less)



Index for a File (Database Table)

- Alternatives for data entries *k**
  - Actual data record with search value *k*
  - <*k*, rid of data record with search key *k*>
  - <*k*, list of rids of data records with search key *k*>
- Choice is orthogonal to the indexing technique used to locate data entry *k**

Most Indexes are Tree Structured

- Tree-structured indexes support
  - Range searches (e.g., gpa > 3.0)
  - Equality searches (e.g., type = ‘action’)
- Why not just store the file sorted on search key?
  - Can perform *binary search* to find matching records
  - For range search, find first record and scan forward (backward)
  - While binary search is fast O(log n) ...
  - The cost is high because we are doing the search over *disk blocks* (records stored on disk in blocks)
  - We still sort ... just not for binary search

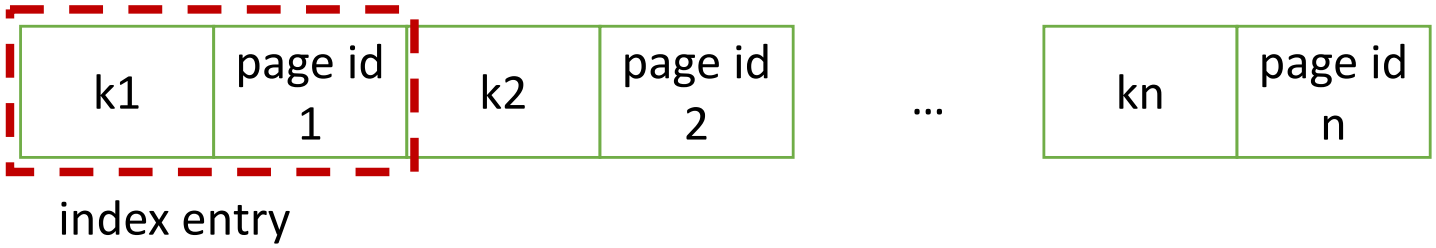
Most Indexes are Tree Structured

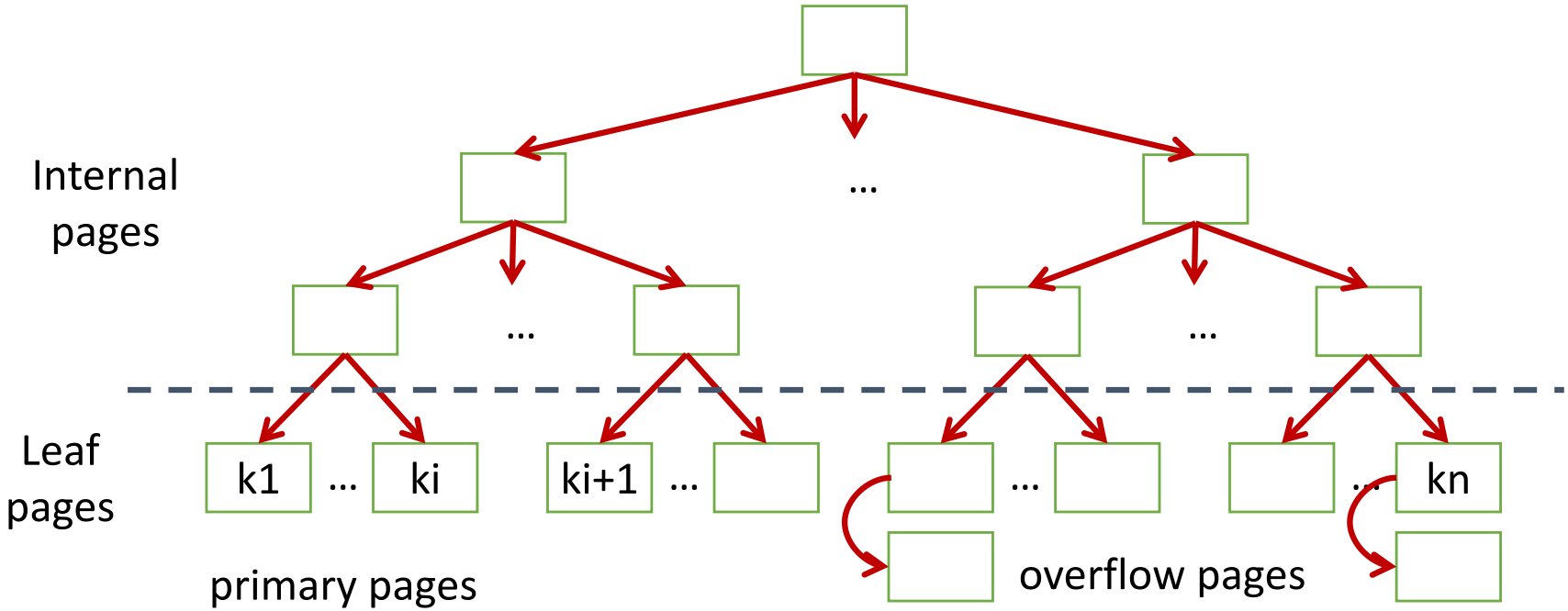
- Basic idea:
  - Store only search keys in a multi-level index
  - Search becomes much cheaper
- ISAM (indexed sequential access method)
  - Static structure (old technology)
  - Index is build just once when file is loaded
  - Uses overflow areas
  - Tree can become very unbalanced (w/ insertions & deletions)
- B+ tree (a B-tree with all data stored in leaf nodes)
  - Dynamic structure
  - Index is adjusted as records are inserted and deleted in the file
  - **Index remains balanced**

ISAM Tree Creation

- File creation
  - Allocate leaf (data) pages sequentially, sorted by search key
  - Then allocate index pages
- Index entries – <k, page id>
  - Used to direct search to data entries (in leaf pages)

ISAM Layout


k1
page id 1
k2
page id 2
...
kn
page id n
index entry


Internal pages
...
...
...
Leaf pages
k1
...
ki
ki+1
...
...
...
kn
primary pages
overflow pages

ISAM operations

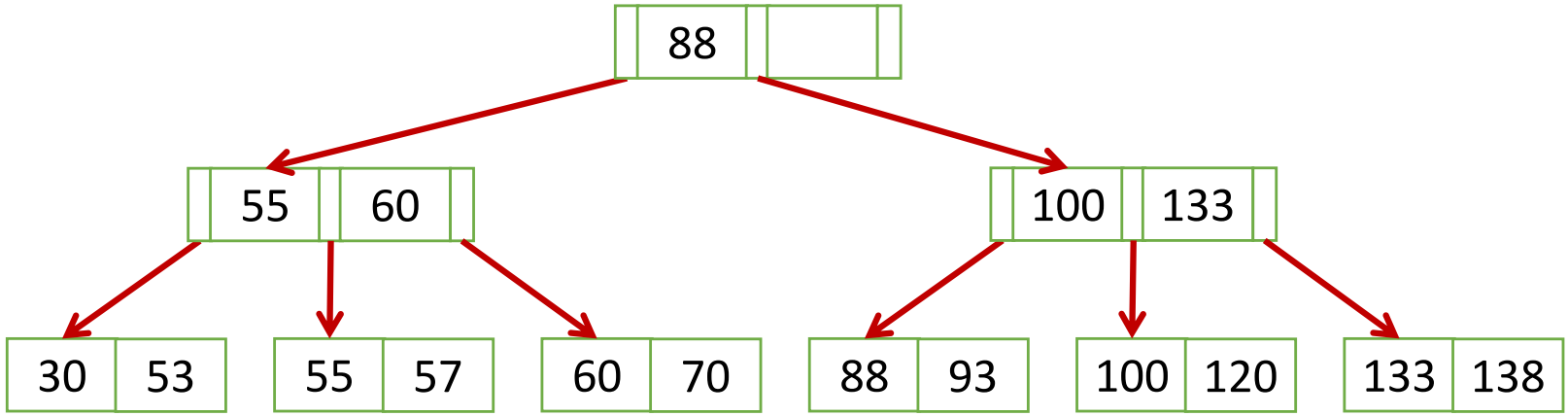
- Search
  - Start at root
  - Use key comparisons to find leaf (like binary search)
  - Cost is $O(\log_f n)$ for entries per index f and leaf pages n
  - However, inserts and deletions can create problems ...
- Insert
  - Find leaf data entry belongs to and add (or create overflow)
- Delete
  - Find and remove data entry (deallocate overflow if needed)

Example

Assume each node holds 2 entries

Keys are 53, 55, 60, 70, 88, 30, 57, 88, 100, 93, 120, 138, 138

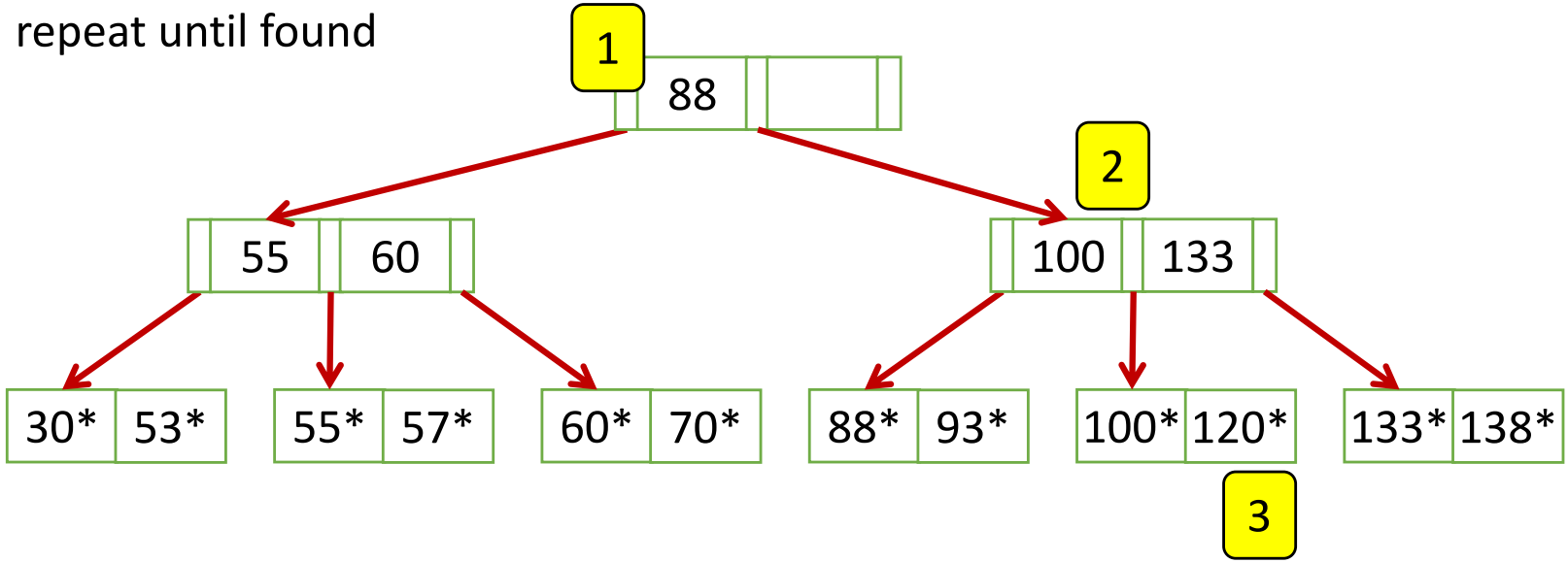
1. Sort the keys
2. Create index



Example: Search

Search 120

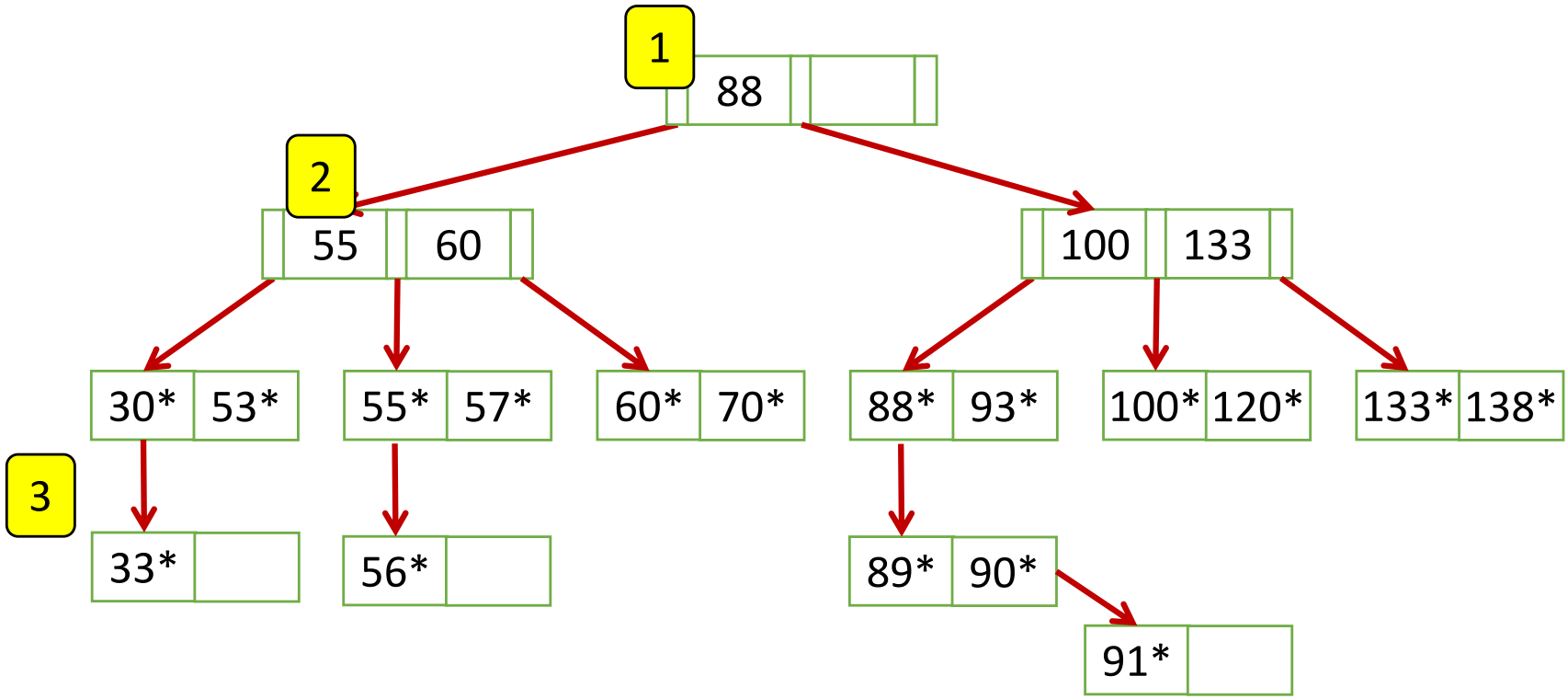
1. visit the head of the tree
    1. larger than follow right,
    2. smaller than follow left
    3. in between take the middle
    4. repeat until found



Example: Insert

Insert 33*, 56*, 89*,90*,91*

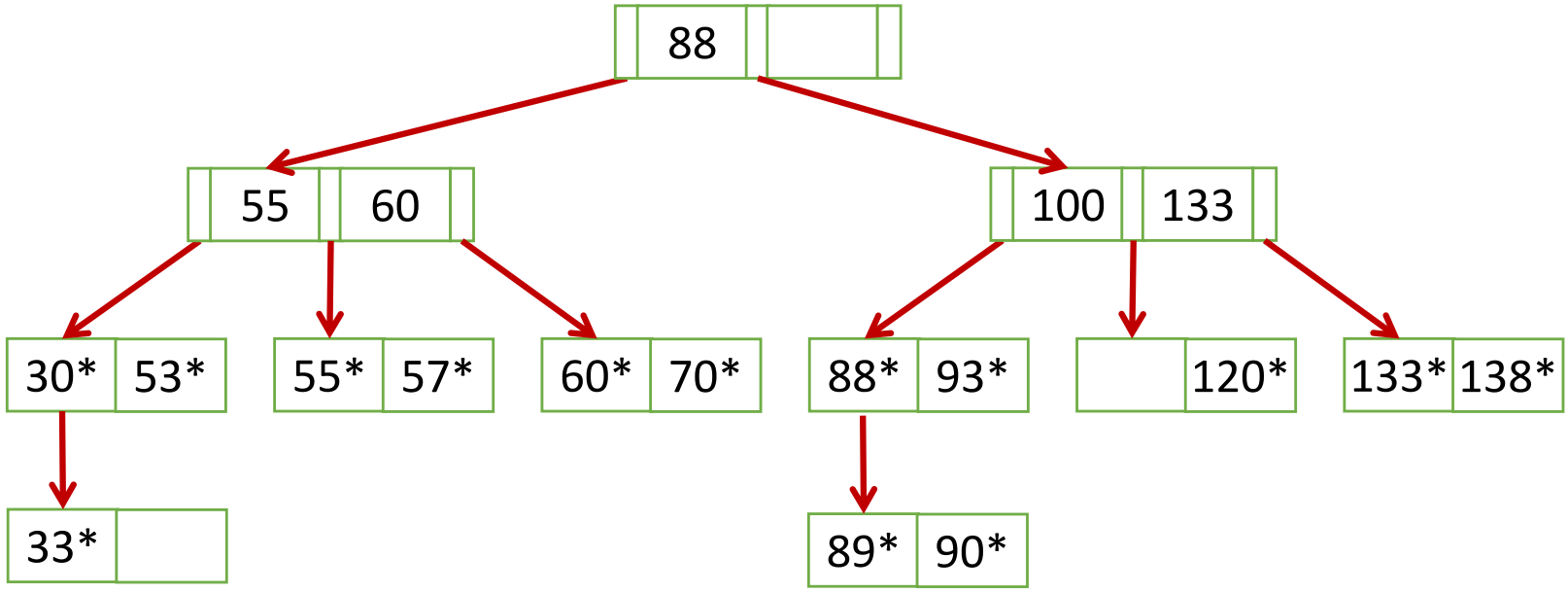
1. same as search
2. if the node is full make overflow



Example: Delete

Delete 100*, 91*, 56*

1. same as search
2. delete the entry



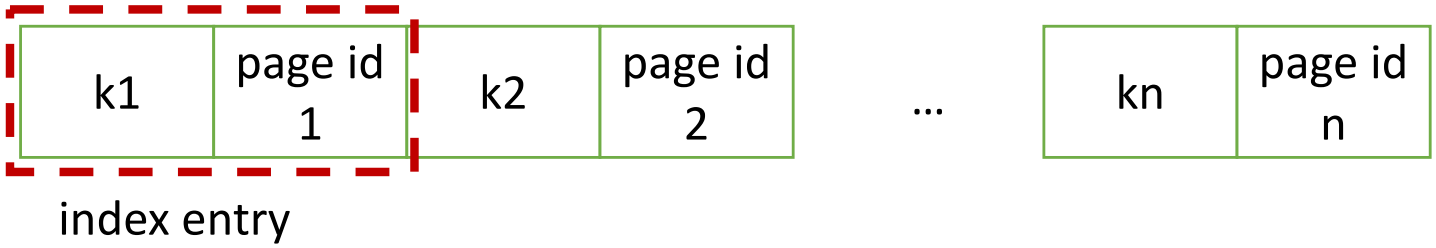
B+ Tree

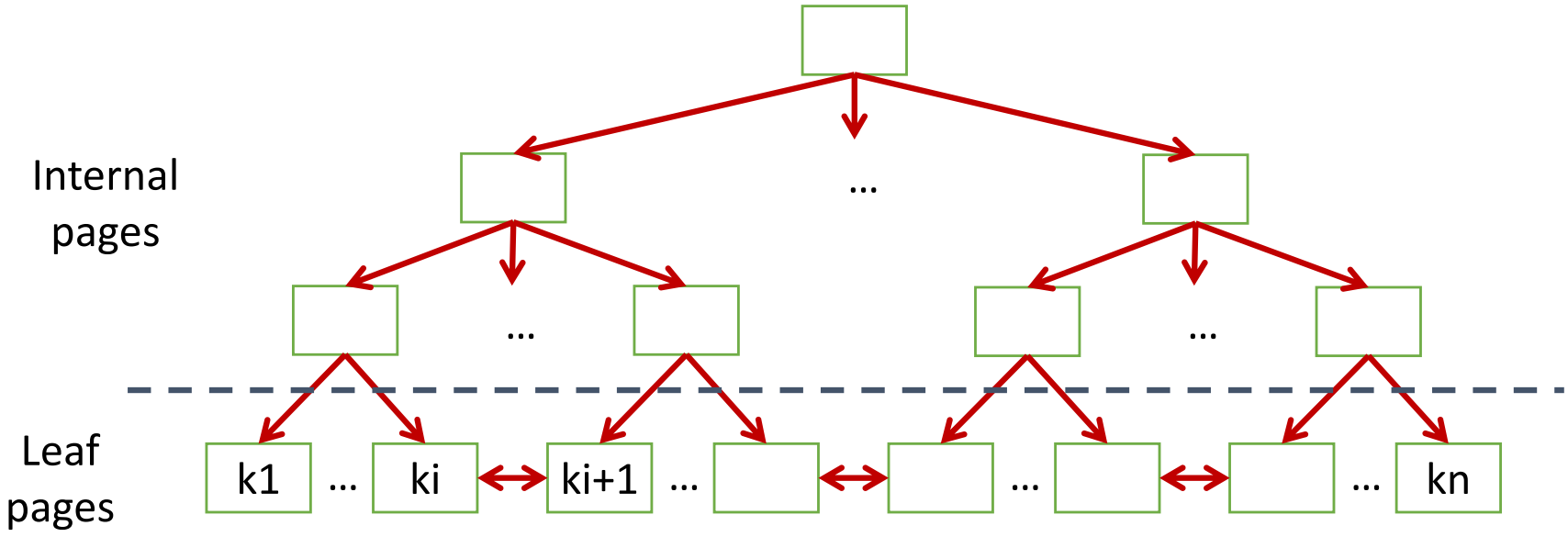
- Ensures the tree stays balanced
  - Insert, delete, search are $O(\log_f n)$
  - Where *f* denotes the “fanout”
- Minimum 50% occupancy (except for root)
  - Each node contains d <= m <= 2d entries
  - The parameter **d is called the order of the tree**
- Maintains a doubly linked list of data-entry pages
- Supports equality and range searches efficiently

B+ Tree: Most widely used index

- Ensures the tree stays balanced
  - Insert, delete, search are $O(\log_f n)$
  - Where *f* denotes the “fanout”
- Minimum 50% occupancy (except for root)
  - Each node contains d <= m <= 2d entries
  - The parameter d is called the order of the tree
- Maintains a doubly linked list of data-entry pages
- Supports equality and range searches efficiently

B+tree Layout



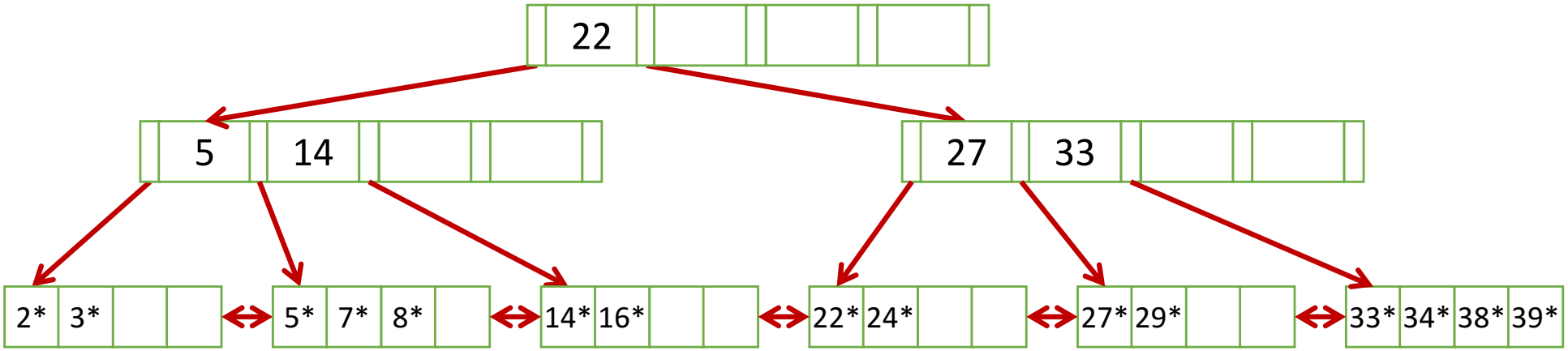


Example: Search

Suppose we have 2, 3, 5, 7, 8, 14, 16, 22, 4, 27, 29, 33, 34, 38, 39

How many page I/Os are required to find

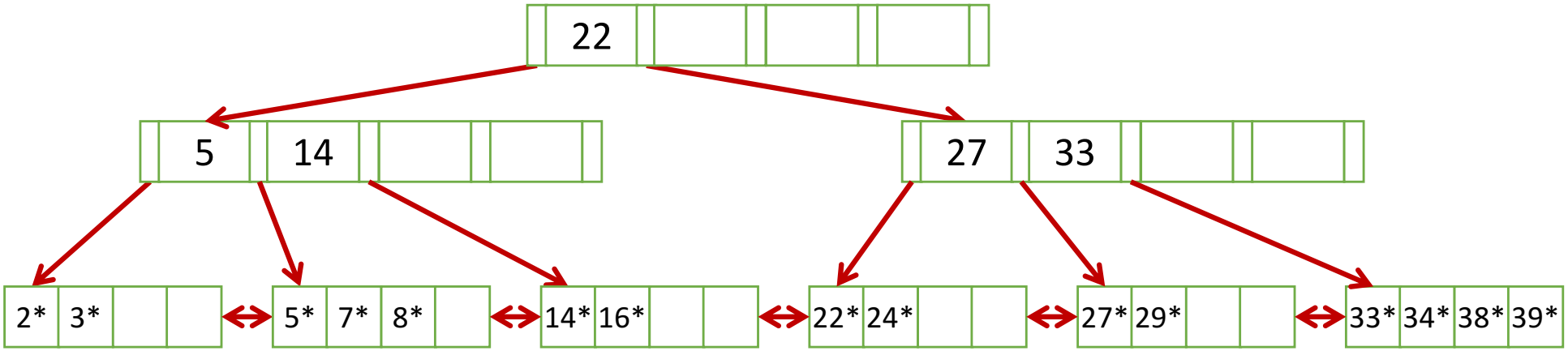
- entry 24*
- entry 30*
- all data entries >15* and <30*



Example: Update

insert/modify/delete require finding data entry in leaf
When inserting, if the page is full, the page has to split

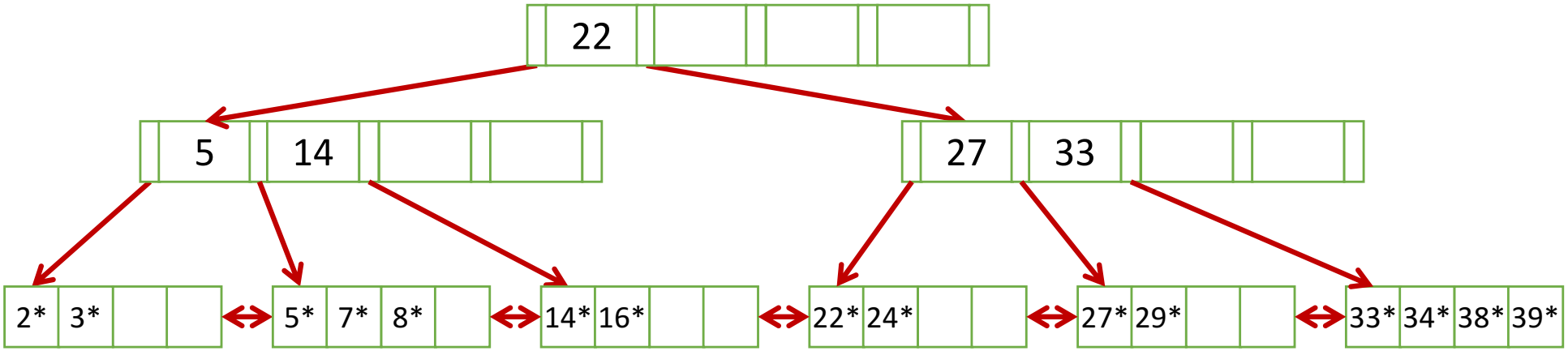
- one entry is added to parent
- changes may have snow ball effect
- root is special
- this maintains the tree balance



Example: delete

Delete 7, 14

1 find 7



Combine (merge) pages on delete to maintain 50% full constraint

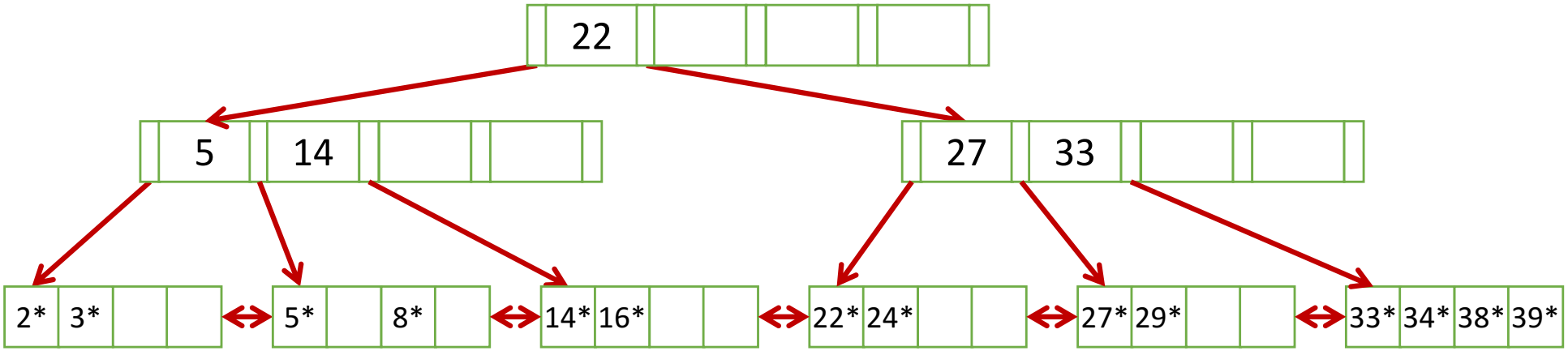
- – Changes also "bubble" up the tree

Example: delete

Delete 7, 14

1 find 7

2 Delete 7



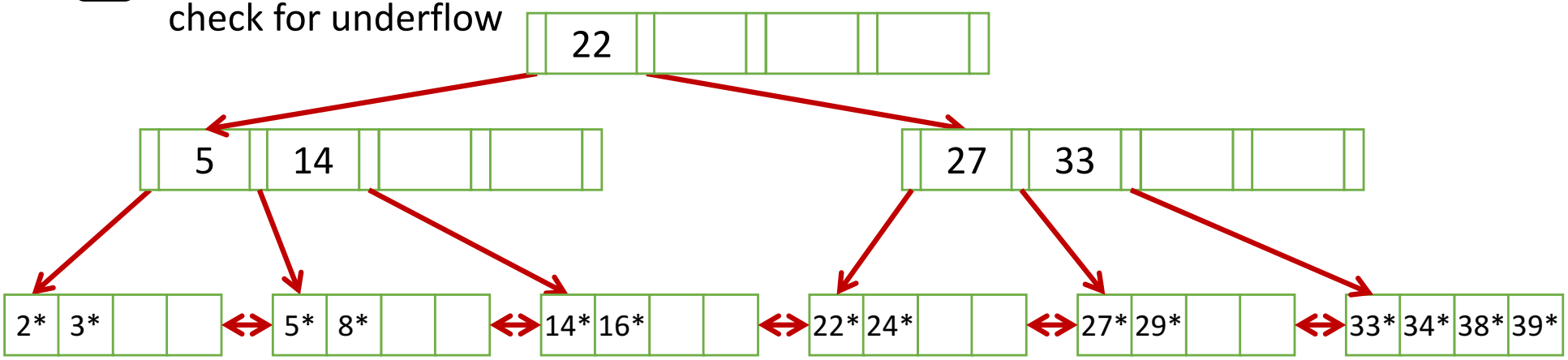
Combine (merge) pages on delete to maintain 50% full constraint

- – Changes also "bubble" up the tree

Example: delete

Delete 7, 14

1. find 7
2. Delete 7
3. Shift 8;
   check for underflow



Combine (merge) pages on delete to maintain 50% full constraint

- – Changes also "bubble" up the tree

Example: delete

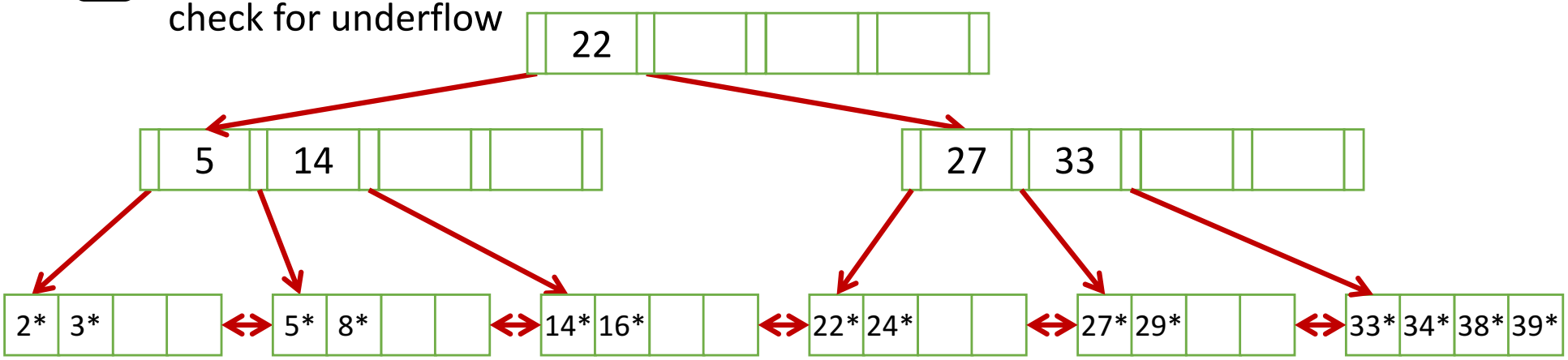
Delete 7, 14

1 find 7

2 Delete 7

3 Shift 8;
check for underflow

4 find 14



Combine (merge) pages on delete to maintain 50% full constraint

- – Changes also "bubble" up the tree

Example: delete

Delete 7, 14

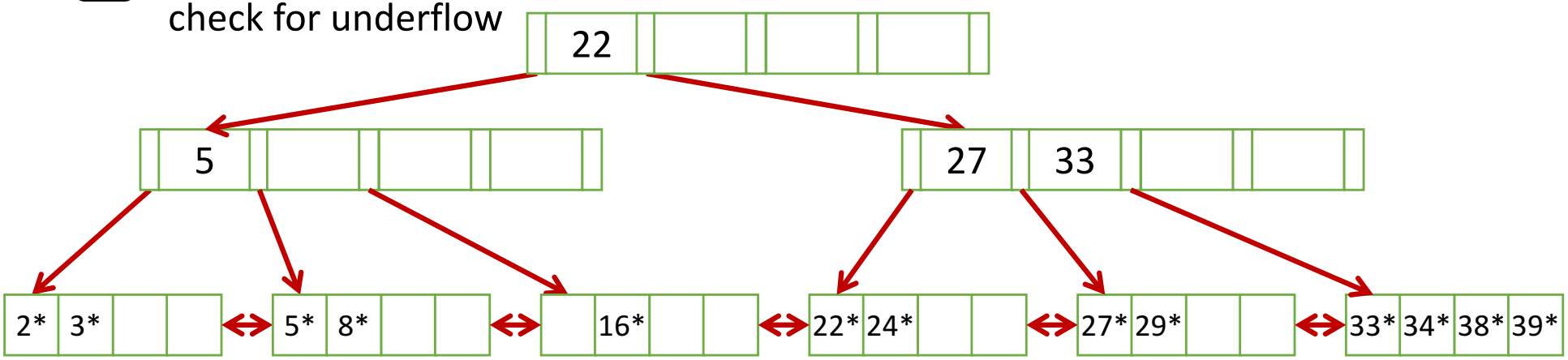
1 find 7

2 Delete 7

3 Shift 8;
check for underflow

4 find 14

5 Delete 14;
parent is also deleted



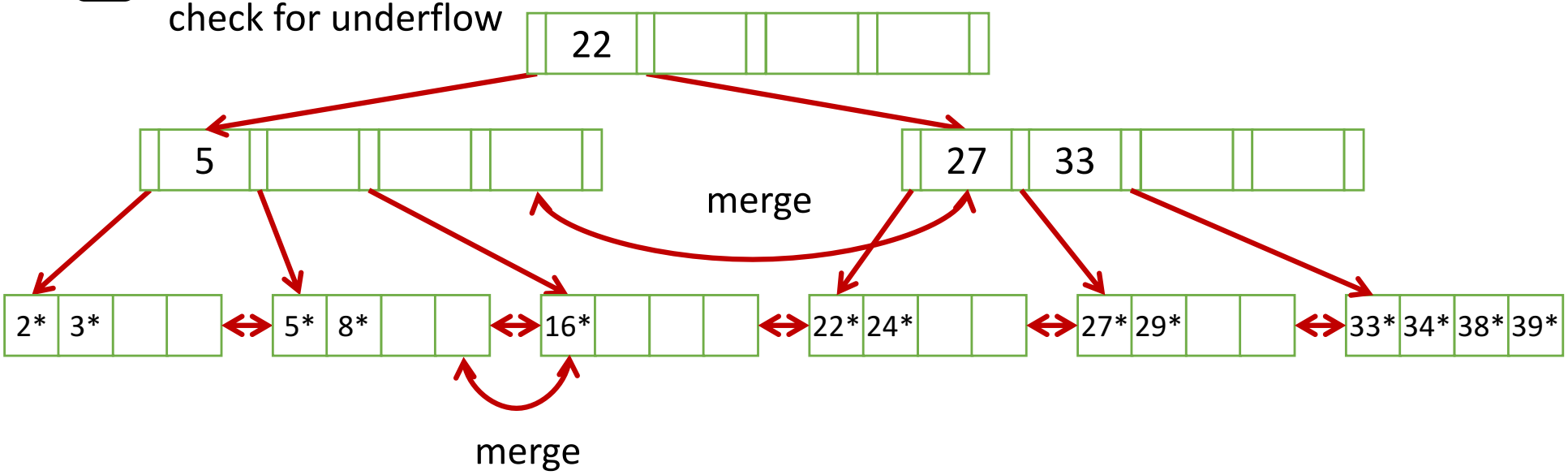
Combine (merge) pages on delete to maintain 50% full constraint

- – Changes also "bubble" up the tree

Example: delete

Delete 7, 14

1. find 7
2. Delete 7
3. Shift 8; check for underflow
4. find 14
5. Delete 14; parent is also deleted
6. Detected underflow; merge



Combine (merge) pages on delete to maintain 50% full constraint

- – Changes also "bubble" up the tree

Example: delete

Delete 7, 14

1 find 7

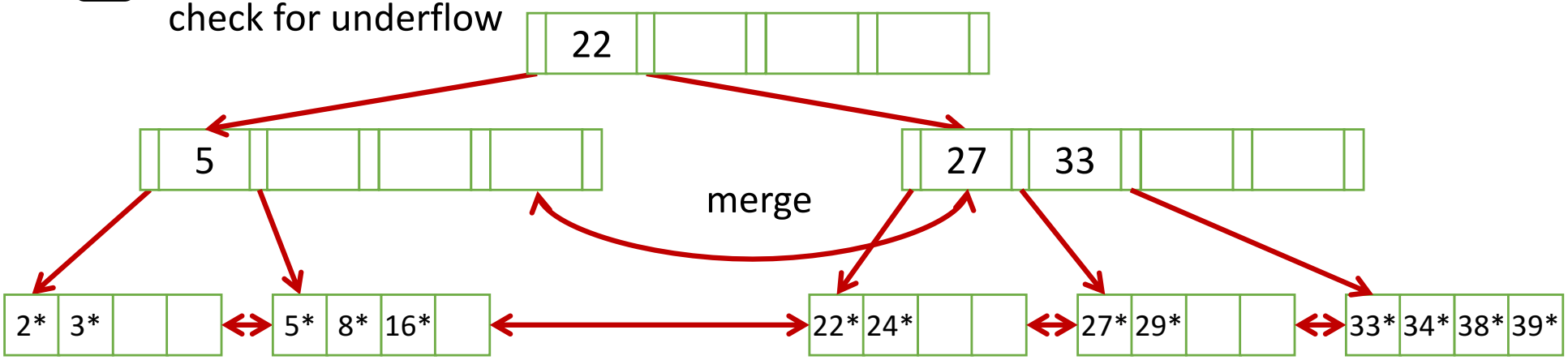
2 Delete 7

3 Shift 8;
check for underflow

4 find 14

5 Delete 14;
parent is also deleted

6 Detected underflow;
merge



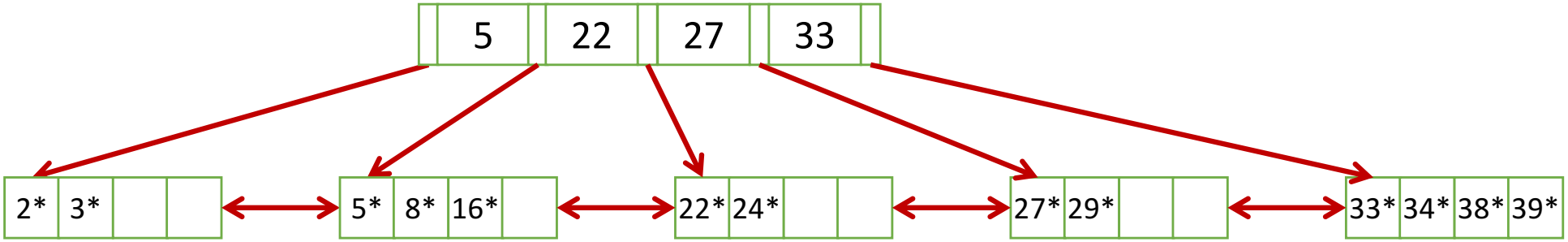
Combine (merge) pages on delete to maintain 50% full constraint

- – Changes also "bubble" up the tree

Example: delete

Delete 7, 14

1. find 7
2. Delete 7
3. Shift 8; check for underflow
4. find 14
5. Delete 14; parent is also deleted
6. Detected underflow; merge



Combine (merge) pages on delete to maintain 50% full constraint

- – Changes also "bubble" up the tree

Notes on B+ tree

- Creating a B+ tree by incremental inserts
  - Is really slow!
  - Instead, we can “bulk” load the items
  - This means build the tree bottom up from the sorted data entries (leaves)
  - Results in fewer I/Os to build the index

- Important to increase fan out ... why?

Costs associated with indexes

- If you define an index in your database you will incur three costs
  - Additional space to store the index
  - Updates will be slower (rebalance the tree) – More optimization choices
- The advantage is many queries will run faster
  - But need to determine if it makes sense with your workload
  - i.e., does the trade-off of query versus update time make sense for your application

To do

- Review
  - Ch. 8
  - Ch 9: intro 9.1, 9.3-9.7
  - Ch. 10: Intro, 10.1-10.6