ECS 165B: Database System Implementation
Lecture 4

UC Davis

April 5, 2010

Acknowledgements: based on slides by Raghu Ramakrishnan and Johannes Gehrke. Presentation of RCS based on slides by David Matuszek.

Class Agenda

- Last time:
  - Quick look at DavisDB Record Manager component
  - Start review of indexing
- Today:
  - Project-related logistics: subversion, code submission
  - Finish indexing review; tree-structured indices in depth
- Reading:
  - Chapter 10 of Ramarkrishnan & Gehkre
  - (or Chapter 12 of Silberschatz et al.)

Announcements

Subversion repositories have been created, but with incorrect team assignments (Excel snafu)… ☹ Will be fixed soon…

Code submission procedure has been finalized

Textbook on reserve in Shields starting tomorrow

Office hours tomorrow @11am (TJ), Wednesday @11am (Mingmin)

Why use revision control systems?

- Scenario 1:
  - Your program is working
  - You change "just one thing"
  - Your program breaks
  - You change it back
  - Your program is still broken – *why*?

- Has this ever happened to **you**?

Why use revision control systems (2)?

- Your program worked well enough yesterday
- You made a lot of improvements last night…
  - but you haven't gotten them to work yet
- You need to turn in your program *now*

- Has this ever happened to you?

Revision control for teams

- Scenario:
  - You change one part of a program -- it works
  - Your co-worker changes another part -- it works
  - You put them together -- it doesn't work
  - Some change in one part must have broken something in the other part
  - What were all the changes?

Revision Control for Teams (2)

- Scenario:
  - You make a number of improvements to a class
  - Your co-worker makes a number of *different* improvements to the *same* class
- How can you merge these changes?

Revision control systems

- A *revision control system* (aka *version control system*) does these things:
  - Keeps multiple (older and newer) versions of source code, headers, etc
  - Requests comments regarding every change
  - Displays differences between versions
  - Detect/resolve conflicts
- Many systems out there: sccs, rcs, cvs, Visual SourceSafe, svn
  - Most popular in the past: **cvs**
  - Most popular nowadays: **svn**

Subversion commands

- **svn checkout (aka svn co)** – check out code from repository
- **svn add** – add a new file/directory to the repository
- **svn delete** – delete a file/directory from the repository
- **svn commit** – commit local changes to repository
- **svn diff** – view differences wrt current or old version
- **svn status** – see local changes
- **svn info** – get info about repository
- **svn help** – list all commands

- See **http://subversion.tigris.org**

- Graphical front-ends: TortoiseSVN (Windows), RapidSVN (cross-platform), Subclipse (eclipse plug-in)
  - Visual diffs, easier browsing of history, …

Logistics: Repository Access

Follow directions on

http://www.cs.ucdavis.edu/~green/courses/ecs165b/project.html

```
[green@pc12 ~]$ svn co file:///home/cs165b/CSIF-Proj/
   cs165b-0/svn/trunk/DavisDB
A    DavisDB/RecordFileHandle.h
A    DavisDB/FileHandle.h
A    DavisDB/PageFileManager.cpp
A    DavisDB/RecordManager.cpp
…
A    DavisDB/submit.sh
A    DavisDB/CMakeLists.txt
A    DavisDB/writeup.txt
A    DavisDB/Common.h
[green@pc12 ~]$
```

Logistics: Repository Access

- Must tell repository about new files!

```
[green@pc12 ~/DavisDB]$ svn add Foo.cpp Foo.h
A         Foo.cpp
A         Foo.h
[green@pc12 ~/DavisDB]$ svn commit -m ""
Adding         Foo.cpp
Adding         Foo.h
Transmitting file data ..
Committed revision 84.
```

- To get changes from your teammate:

```
[chenmi@pc10 ~/DavisDB]$ svn update
```

Logistics: Submitting Your Homework

[green@pc12 DavisDB]$ ./submit.sh
Usage: submit.sh <hw#>
where <hw#> is a number in the range [1,5]

Submits your project component by tagging the current version of your subversion repository as the submitted version. **It may be executed multiple times for the same <hw#>.** The most recently submitted version is the one that will be used for grading (and its timestamp will determine any late penalties). This script must be run from your subversion DavisDB directory.

After submitting, **the script will also run a test build of your project**, by checking out the submitted version into a temporary directory and executing "cmake ." then "make".

Logistics: Submitting Your Homework (2)

```
[green@pc12 DavisDB]$ ./submit.sh 1
Submitting HW1...
Submission successful.
Running a test build on the submitted code...
Test build successful.
```

Indexing Review

Sparse vs. Dense Indices

- Not described in R&G; see Silberschatz et al
- **Dense index:** has search key value and data entry for *every* record in the file
  - fast record lookup
- **Sparse index:** has search key values for only *some* records in the file (but all data entries)
  - less space, reduced maintenance costs
  - essentially, Alternative 1

Cost of Operations

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap	BD	0.5BD	BD	2D	Search +D
(2) Sorted	BD	Dlog 2B	Dlog 2 B + # matches	Search + BD	Search +BD
(3) Clustered	1.5BD	Dlog F 1.5B	Dlog F 1.5B + # matches	Search + D	Search +D
(4) Unclustered Tree index	BD(R+0.15)	D(1 + log F 0.15B)	Dlog F 0.15B + # matches	D(3 + log F 0.15B)	Search + 2D
(5) Unclustered Hash index	BD(R+0.125)	2D	BD	4D	Search + 2D

- *B* = # data pages; *R* = # of records per page; *D* = (average) time to read or write a disk pages
- High-order bit: **no one file organization is uniformly superior in all situations**
- You will need to memorize this entire matrix for Quiz #1
  - Just kidding

Tree-Structured Indices

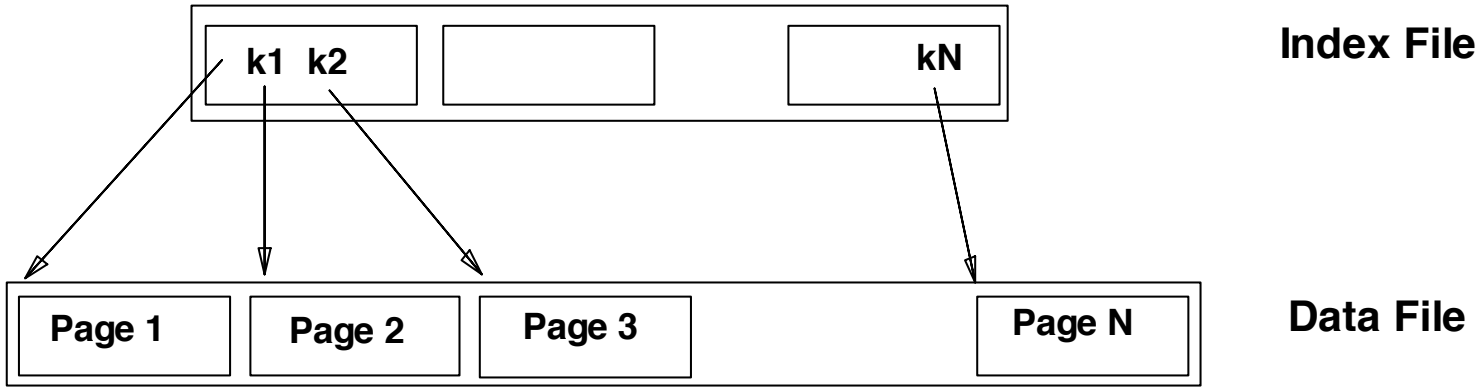
reading: Chapter 10 of Ramakrishnan and Gehrke / Chapter 12 of Silberschatz et al

Introduction

- As for any index, 3 alternatives for data entries *k**
  - Data record with key value *k*
  - <*k*, rid of data record with search key value *k*>
  - <*k*, list of rids of data records with search key *k*>
- Choice is orthogonal to the *indexing technique* used to locate data entries *k**
- Tree-structured indexing techniques support both *range searches* and *equality searches*
- *ISAM ("Indexed sequential access method")*: static structure; *B +-tree*: dynamic, adjusts gracefully under insertions and deletions

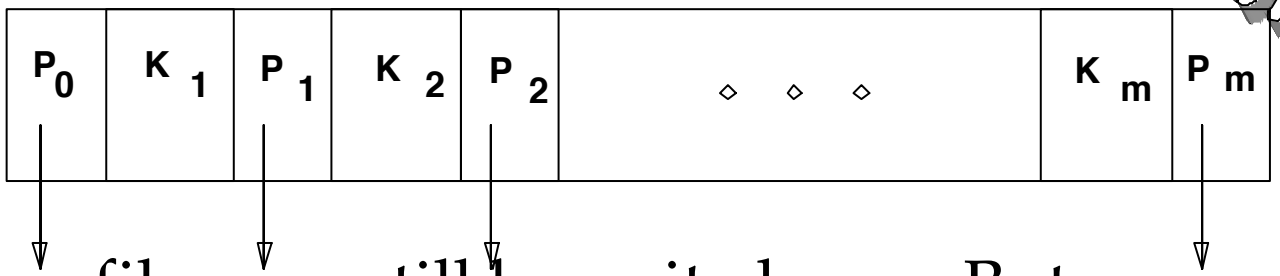
Range Searches

- "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
- Simple idea: create an "index" file

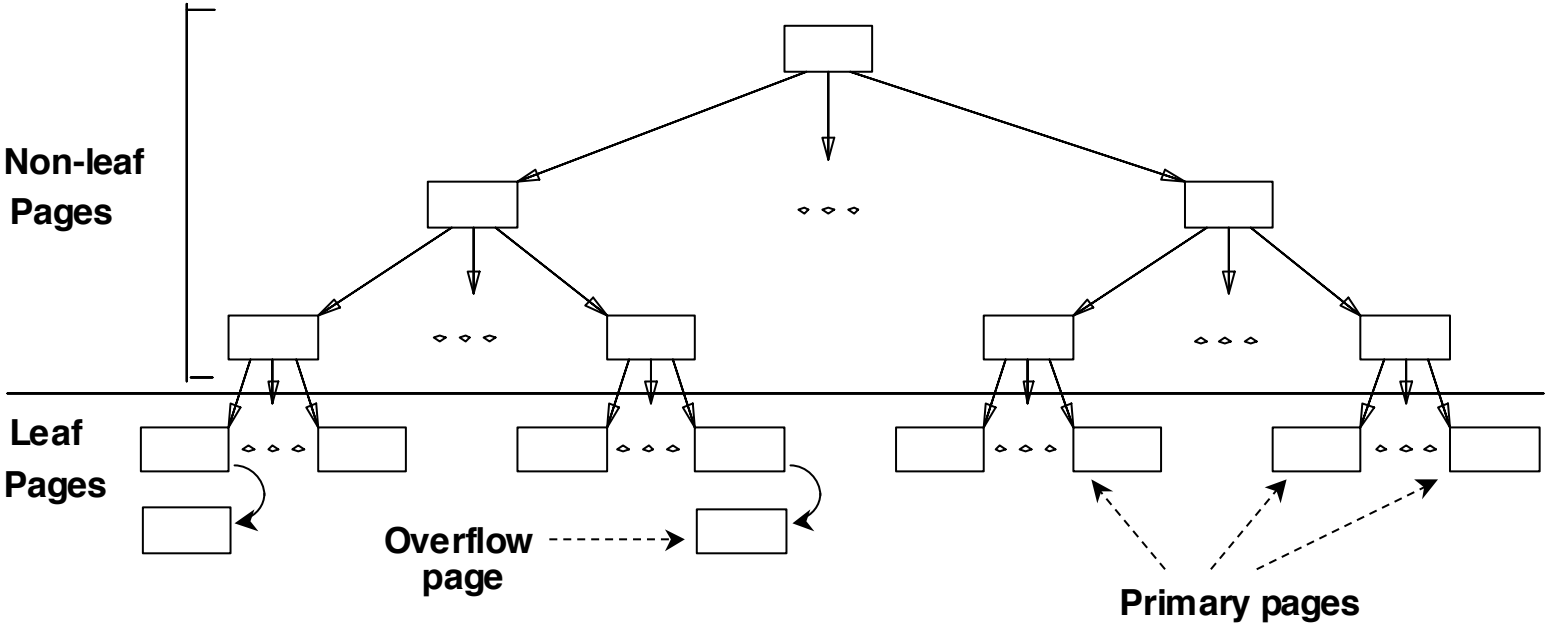


  - can do binary search on (smaller) index file!

ISAM

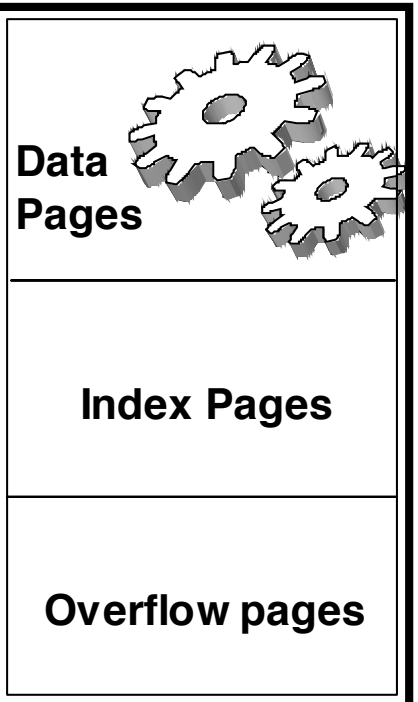


- Index file may still be quite large. But we can apply the idea repeatedly!



* *Leaf pages contain data entries.*

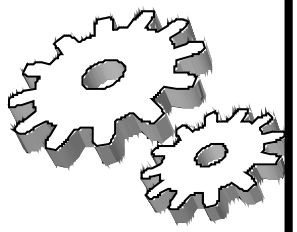
Comments on ISAM



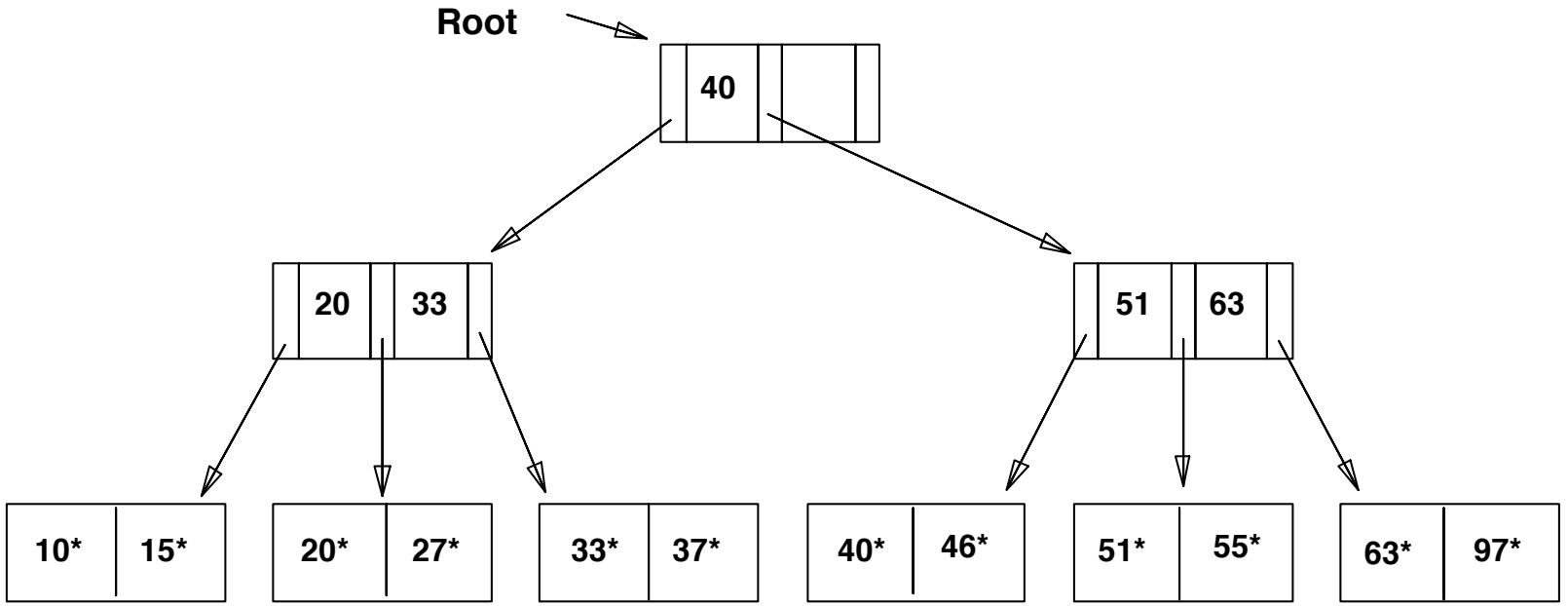
- *File creation*: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.
- *Index entries*: <search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.
- *Search*: Start at root; use key comparisons to go to leaf. Cost $\propto \log_F N$; F = # entries/index pg, N = # leaf pgs
- *Insert*: Find leaf data entry belongs to, and put it there.
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.

* **Static tree structure**: *inserts/deletes affect only leaf pages*.

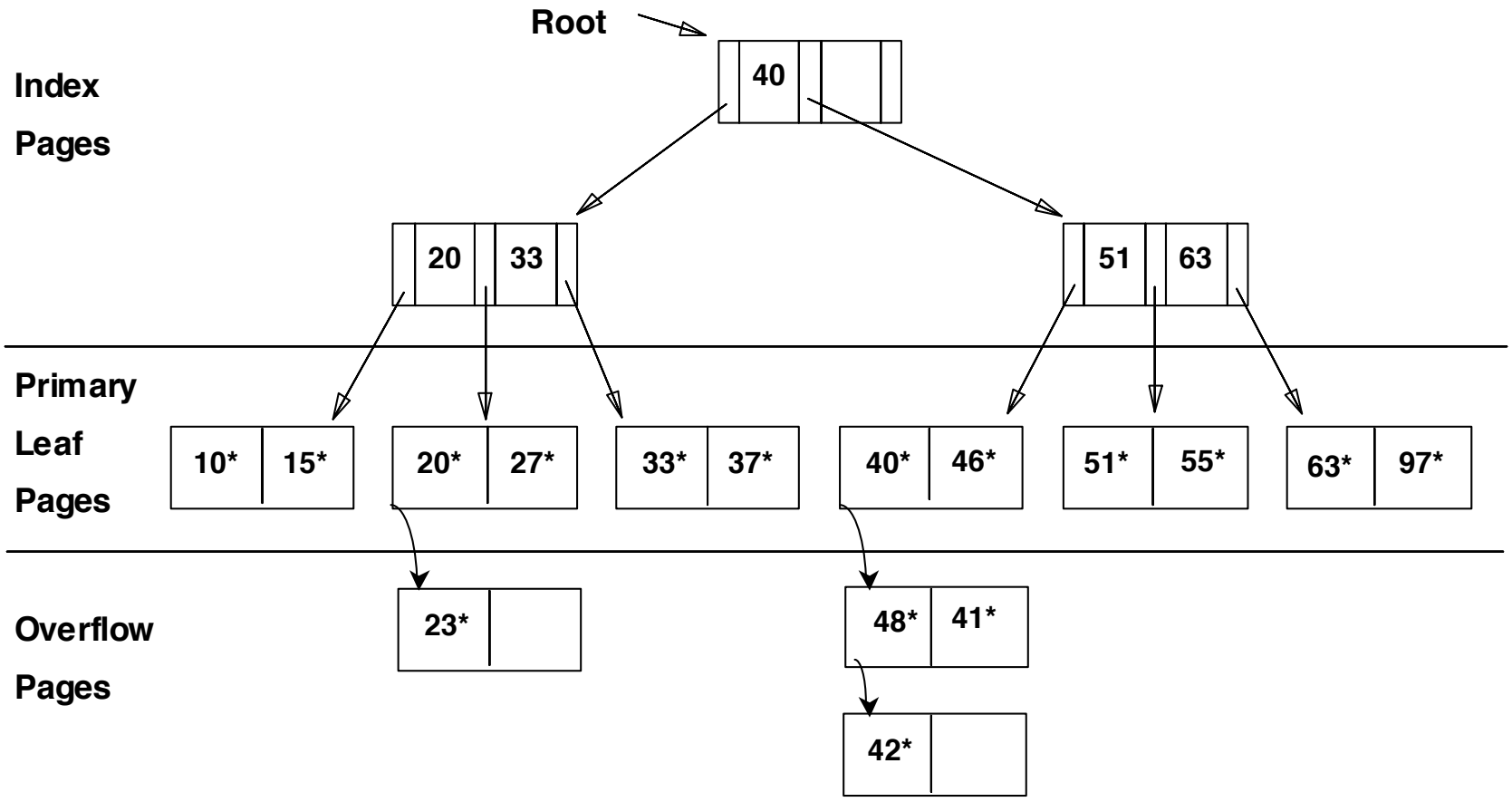
Example ISAM Tree



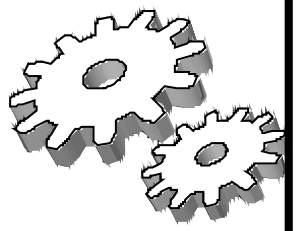
- Each node can hold 2 entries; no need for `next-leaf-page' pointers. (Why?)

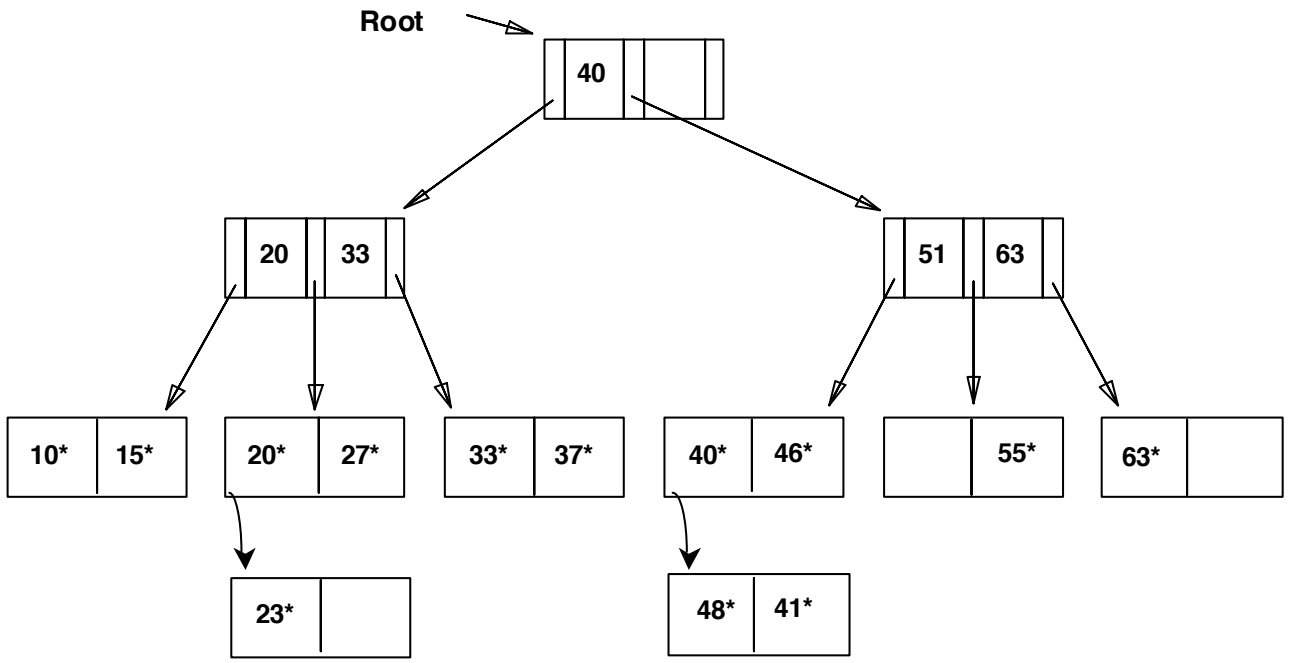


After Inserting 23, 48*, 41*, 42*...*



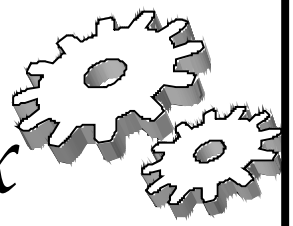
... Then Deleting 42*, 51*, 97*



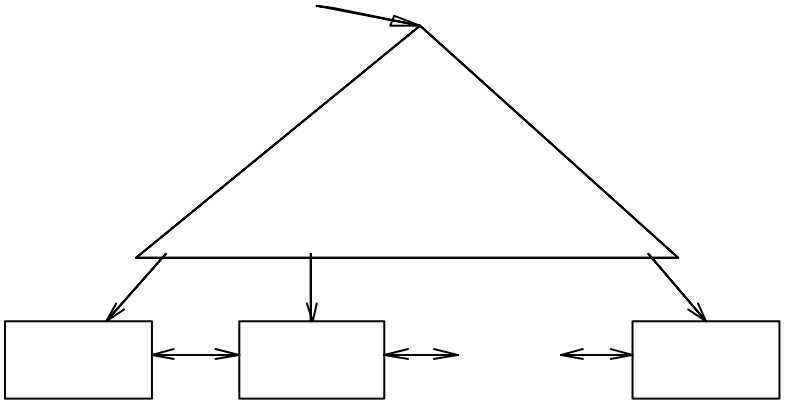


** Note that 51* appears in index levels, but not in leaf!*

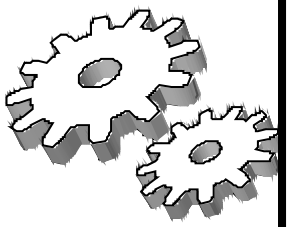
B+ Tree: Most Widely Used Index



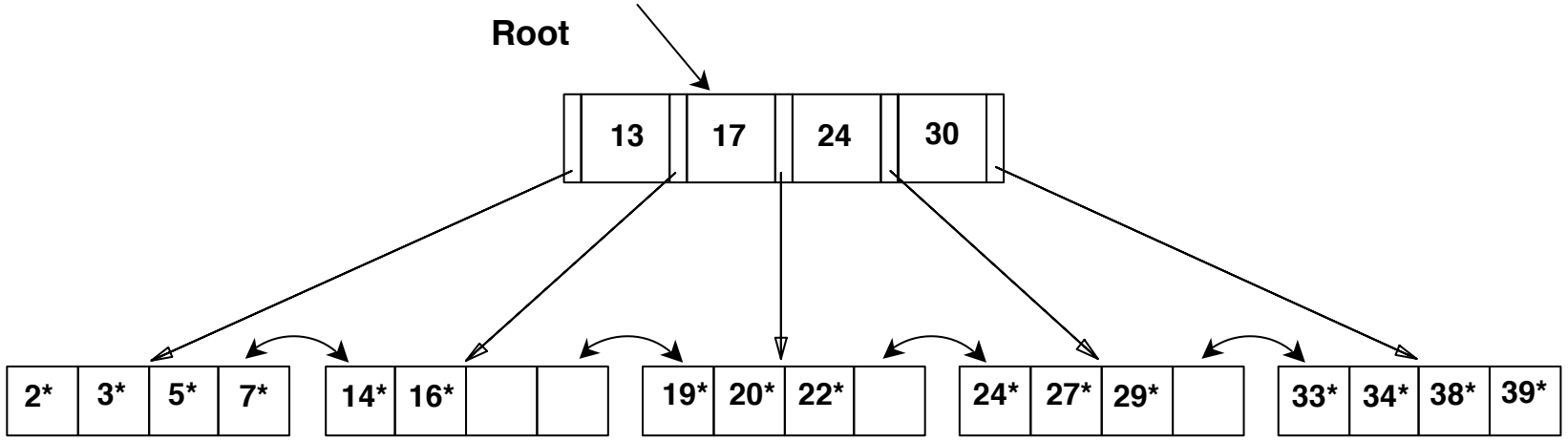
- Insert/delete at $\log_F N$ cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains **d** <= *m* <= 2**d** entries. The parameter **d** is called the *order* of the tree.
- Supports equality and range-searches efficiently.



Example B+ Tree

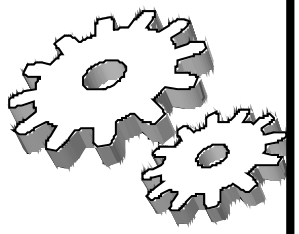


- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...



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

B+ Trees in Practice



- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 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