

## Announcements

- Please remember to send a mail to Deepa to register for a timeslot for your project demo by March 6, 2003
  - See Project Guidelines on class web page for more details
- Project is due on March 11, 2003
- Final Examination
  - March 18, 2003 7.30pm to 10.30pm

## Today's Lecture

- The memory hierarchy. Section 9.1
- Buffer Manager. Section 9.4
- Record formats. Section 9.7
- Page formats. Section 9.6
- Indexes. Section 8.2
- ISAM. Section 10.1 and 10.2

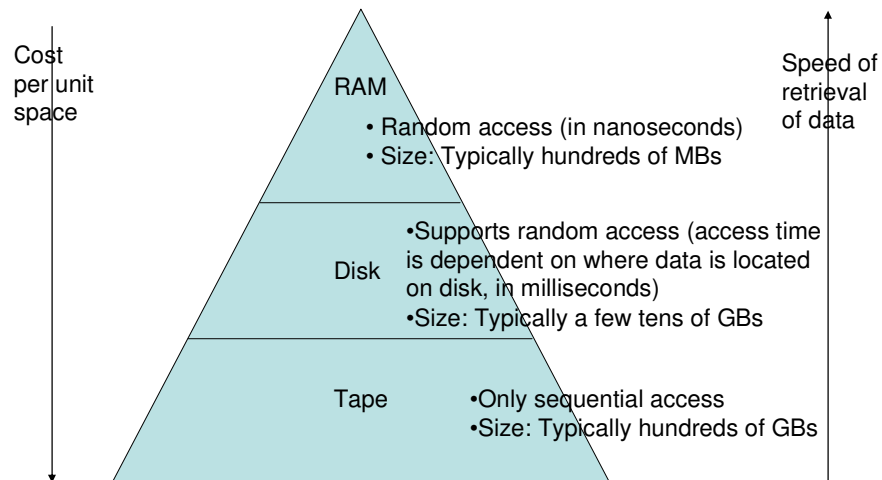
## Disk and Files

- 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. Plan your actions carefully so as to minimize the number of of READ and WRITE operations.

Winter 2003

3

## The Memory Hierarchy



Winter 2003

4

## Why Not Store Everything in Main Memory?

- *Costs too much*: Cost of RAM about 100 times the cost of the same amount of disk space.
- *Main memory is volatile*. We want data to be saved between runs. (Obviously!)
- Typical storage hierarchy:
  - Main memory (RAM) for currently used data.
  - Disk for the main database (secondary storage).
  - Tapes for archiving older versions of the data (tertiary storage)

Winter 2003

5

## Disks

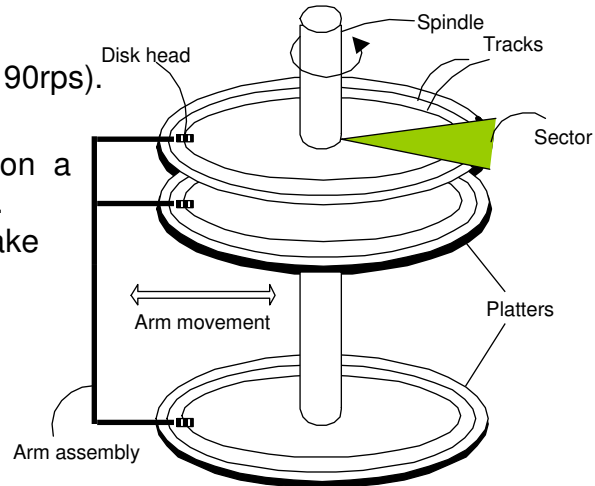
- Secondary storage device of choice.
- Main advantage over tapes: *random access* vs. *sequential*.
- Data is stored and retrieved in units called *disk blocks* or *pages*. Typical page size = 4KB, 8KB
- Unlike RAM, time to retrieve a disk page varies depending upon location on disk
  - Data is brought from disk to RAM before a DBMS can operate on it
  - Therefore, relative placement of pages on disk has major impact on DBMS performance!

Winter 2003

6

## Components of a Disk

- The platters spin (say, 90rps).
- 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).



Winter 2003

7

## Accessing a Disk Page

- Access time for a disk block depends on the location of the block on disk
  - Access time = seek time + rotational delay + transfer time
  - *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 1 to 20msec
  - Rotational delay varies from 0 to 10msec
  - Transfer rate is about 1msec per 4KB page
- Key to lower I/O cost: reduce seek/rotation delays!

Winter 2003

Hardware vs. software solutions

8

## Arranging Pages on Disk

- “*Next*” block concept:
  - blocks on same track, followed by
  - blocks on same cylinder, followed by
  - blocks on adjacent cylinder
- Blocks in a file should be arranged sequentially on disk (by “next”), to minimize seek and rotational delay.
- For a sequential scan, *pre-fetching* several pages at a time is a big win!

Winter 2003

9

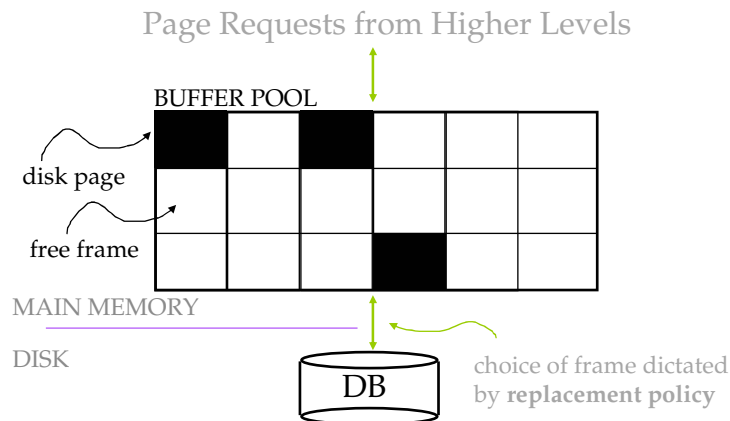
## Disk Space Management

- Lowest layer of DBMS software manages space on disk.
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Request for a *sequence* of pages must be satisfied by allocating the pages sequentially on disk! Higher levels don’t need to know how this is done, or how free space is managed.

Winter 2003

10

## Buffer Management in a DBMS



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

Winter 2003

11

## When a Page is Requested ...

- Every page has a pin count and a dirty bit
- Pin count – the number of times a page has been requested but not yet released
- Dirty bit – indicates whether a page has been modified
- When a page is requested:
  - Check buffer pool to see if page is already in buffer pool.
  - If requested page is in buffer pool, then increment pin count of that page (pin that page) and return the (main memory) address of the frame containing the desired page.

Winter 2003

12

## More on Buffer Management

- If requested page is not in buffer pool:
  - Choose a frame for *replacement*
  - If frame (to be replaced) is dirty, write it to disk
  - Read requested page into chosen frame
  - Pin the page and return its address.
- Requestor of page must unpin it when the requestor is done with the page, and indicate whether page has been modified through the dirty bit.
- A page is a candidate for replacement iff *pin count* = 0.

Winter 2003

13

## Buffer Replacement Policy

- Frame is chosen for replacement by a *replacement policy*:
  - Least-recently-used (LRU), Clock, MRU etc.
- Policy can have big impact on # of I/O's; depends on the *access pattern*.
- Sequential flooding: Nasty situation caused by LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O. MRU much better in this situation (but not in all situations, of course).

Winter 2003

14

## DBMS vs. OS File System

OS does disk space & buffer management: why not let OS manage these tasks?

- A DBMS can predict page reference patterns much more accurately than in an OS environment
  - Page reference patterns are generated by higher-level operations (e.g., sequential scan or particular implementations of RA)
  - Ability to predict page references allows for better buffer management (know what pages to evict, what pages to prefetch even before the page requests are made)
- Also requires the ability to force a page to disk for concurrency control and crash recovery purposes

Winter 2003

15

## Files of Records

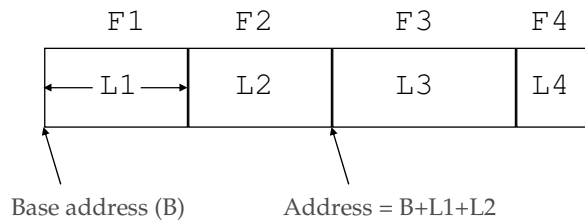
- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.
- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records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

Winter 2003

16

## Record Formats: Fixed Length

A record with 4 fields



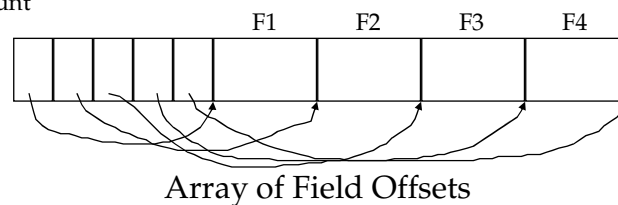
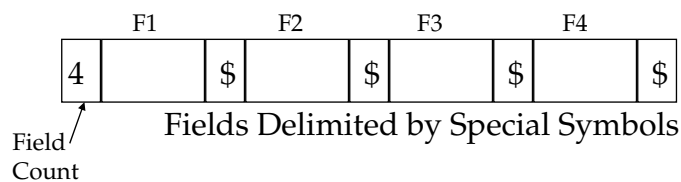
- Information about field types same for all records in a file; stored in *system catalogs*.

Winter 2003

17

## Record Formats: Variable Length

- Two alternative formats (# fields is fixed):

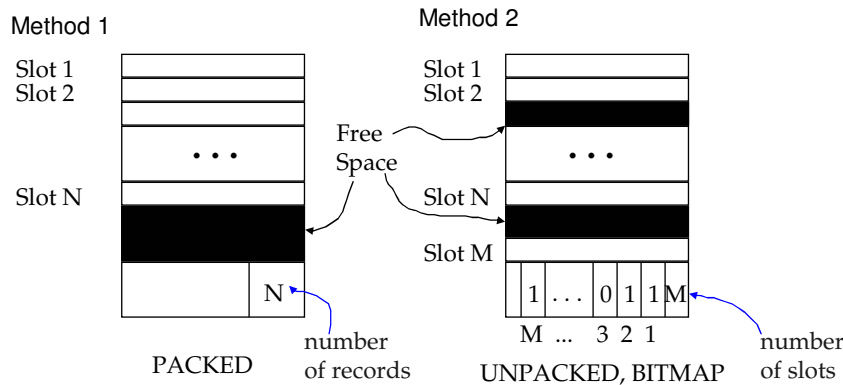


- Second offers direct access to *i*'th field, efficient storage of *nulls* (special *don't know* value); small directory overhead.

Winter 2003

18

## 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.

Winter 2003

19

## Variable-length records

- If records are of variable length, the previous method does not work well
  - A new record may not fit well into existing slots
  - Allocate bigger slots → waste of space
  - Allocate small slots → many records may not fit
- Therefore, allocate sufficient space for a new record and perform space management to fill up “holes” when records are deleted
- Idea: maintain a directory of slots

Winter 2003

20

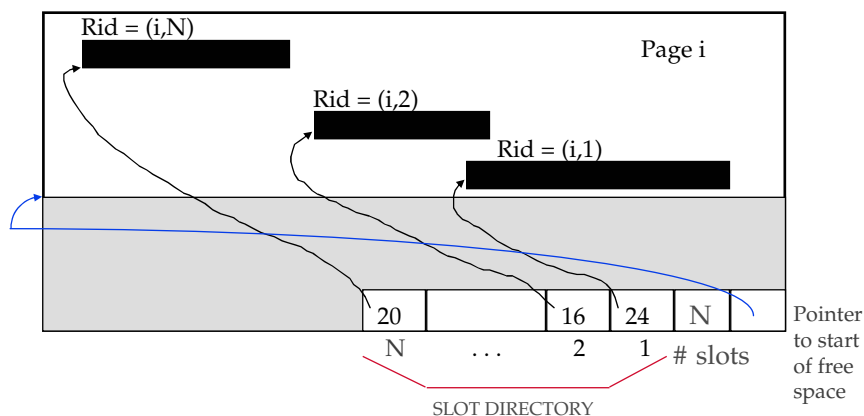
## Directory of slots

- <record offset, record length> pair per slot
  - Record offset – pointer to slot
- Records can be moved around without affecting the record id (<page id, slot number>)
  - Only the record offset stored in the slot changes
- Deletion: set record offset to -1 in slot directory
- Insertion:
  - Look for a free slot in slot directory, create one if necessary
  - If free space is large enough to hold record, insert record into free space and update slot directory
  - Otherwise, reclaim space (compaction) and repeat above

Winter 2003

21

## Page Formats: Variable Length Records



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

Winter 2003

22

## Indexes

- IO cost is important in database operations
- Key to efficiency is to organize files of data records on disk so that IO costs can be minimized
- Clearly, a relation can always be stored as it is (as a heap file). Whenever a record is needed, scan the file to retrieve the desired record.
  - Inefficient (obviously!)
- An **index** is a data structure that organizes data records on disk to optimize certain kinds of retrieval operations

Winter 2003

23

## Index

- Search key  $k^*$  – the value which the search is based upon
  - records that matches the search key are the desired records
- Data entry – records that are stored in an index file
- Data record – the actual data record (which may or may not reside in an index file)

Winter 2003

24

## Alternatives for Data Entry $k^*$ in Index

- Three alternatives for what to store as a data entry in an index:
  - 1) actual data record (together with search key  $k^*$ )
  - 2)  $\langle k, \text{rid} \rangle$  where  $\text{rid}$  is the record id of data record with search key value  $k$
  - 3)  $\langle k, \text{list of rids} \rangle$  where list of rids contain all data records with search key  $k$
- Alternative 3 is better than 2 in terms of space but each list of rids is of variable length
- At most one index should use alternative 1. We do not want to store data records repeatedly

Winter 2003

25

## Remark

- Choice of alternative for data entries is orthogonal to the indexing technique used to locate data entries with a given key value  $k$ .
  - Examples of indexing techniques: B+ trees, hash-based structures
  - Typically, index contains auxiliary information that directs searches to the desired data entries

Winter 2003

26

## Index Classification

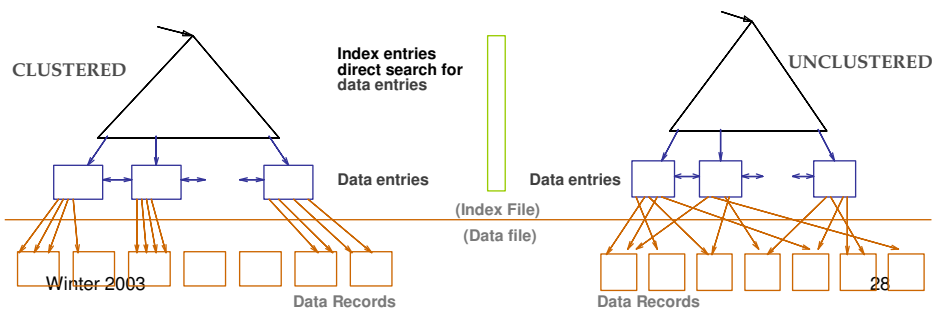
- Primary vs. secondary: If search key contains primary key, then called primary index. Otherwise, called secondary index.
  - Unique index: Search key contains a candidate key
- Clustered vs. unclustered: If order of data records is the same as, or `close to`, order of data entries, then called clustered index.
  - Alternative 1 implies clustered, but not vice-versa.
  - A file can be clustered on at most one search key.
  - Cost of retrieving data records through index varies *greatly* based on whether index is clustered or not!

Winter 2003

27

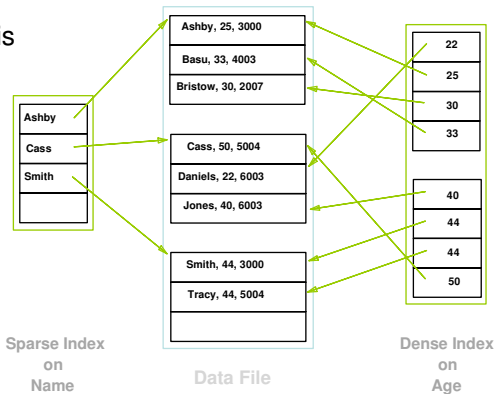
## Clustered vs. Unclustered Index

- Suppose that Alternative (2) is used for data entries, and that the data records are stored in a Heap file.
  - To build clustered index, first sort the Heap 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.)



## Index Classification (Cont.)

- Dense vs. Sparse: If there is at least one data entry per search key value (in some data record), then dense.
  - Alternative 1 always leads to dense index.
  - Every sparse index is clustered!
  - Sparse indexes are smaller; however, some useful optimizations are based on dense indexes.



Winter 2003

29

## Tree Structured Indexing

- We will learn about two index structures: ISAM and B+ trees
- ISAM (Index Sequential Access Method) and B+ trees provide efficient support for range searches.
- Also provide efficient support for insertions and deletions, equality selections (although in this case, hash-based indexes are usually better)

Winter 2003

30

## ISAM vs. B+ trees

- ISAM is a static index structure that is effective when the file is not frequently updated
- B+ tree is a dynamic index structure that adjusts to updates gracefully
  - Most widely used index structure due to its adaptability to updates and support for both equality and range queries

Winter 2003

31

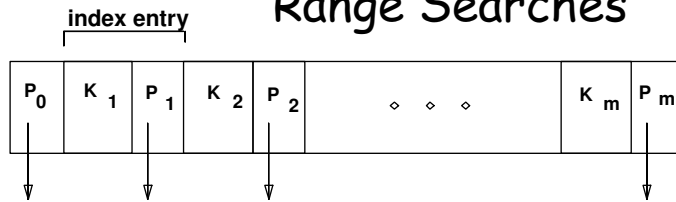
## Range Searches

- Consider a file of student records sorted by gpa
- *“Find all students with gpa > 3.0”*
  - do binary search to find first such student, then scan to find others.
  - cost of binary search can be quite high since cost is proportional to the number of pages fetched
- Simple idea: Create an *“index”* file.

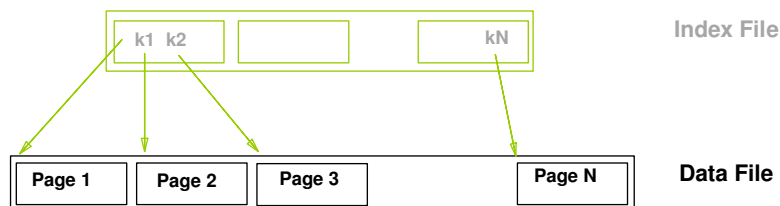
Winter 2003

32

## Range Searches



- Can do binary search on (smaller) index file to find the largest  $k_i$  where  $k_i \leq 3.0$



Winter 2003

33

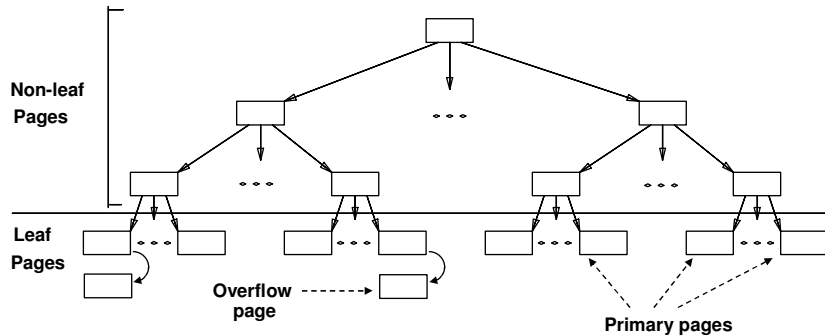
## Range Searches

- Because the size of each entry in the index file is usually much smaller than the size of a page, the index file is much smaller than the data file.
- So, this method is likelier to be more efficient than a simple binary search on the data file
- Tree Indexing:
  - Index file may still be quite large. But we can apply the previous idea repeatedly until the smallest index structure fits one page
  - Leads to a tree-index structure with several pages of non-leaf pages

Winter 2003

34

# ISAM



- Leaf pages contain data entries.
- Suppose no overflow records, and we have  $2^{20}$  data entries stored in  $2^{14}$  primary pages ( $2^6$  data entries per page).
- Each index page holds  $2^7$  index entries. Height of the tree is 2. Therefore, after 3 page IOs, we can locate the right data entry!

Winter 2003

35

# ISAM

- File creation: Leaf (data) pages allocated sequentially, sorted by search key; then index pages allocated, then space for overflow pages.
- Index entries:  $\langle \text{search key value, page id} \rangle$ ; they `direct' search for data entries, which are in leaf pages.
- Search: Start at root; use key comparisons to go to leaf. Cost  $\log_F N + 1$ ;  $F = \# \text{ entries/index pg}$ ,  $N = \# \text{ leaf pgs}$
- 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.

Data Pages

Index Pages

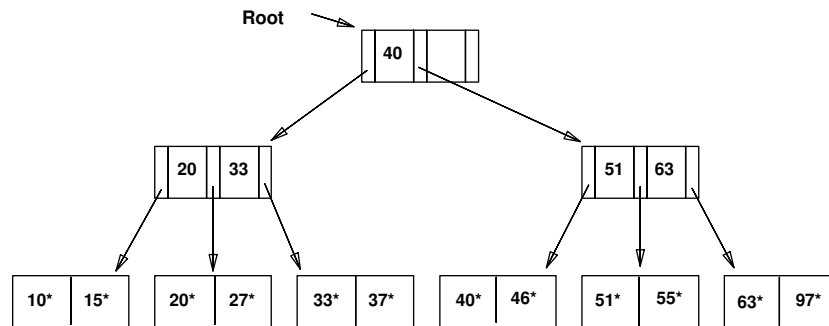
Overflow pages

Winter 2003

36

## Example ISAM Tree

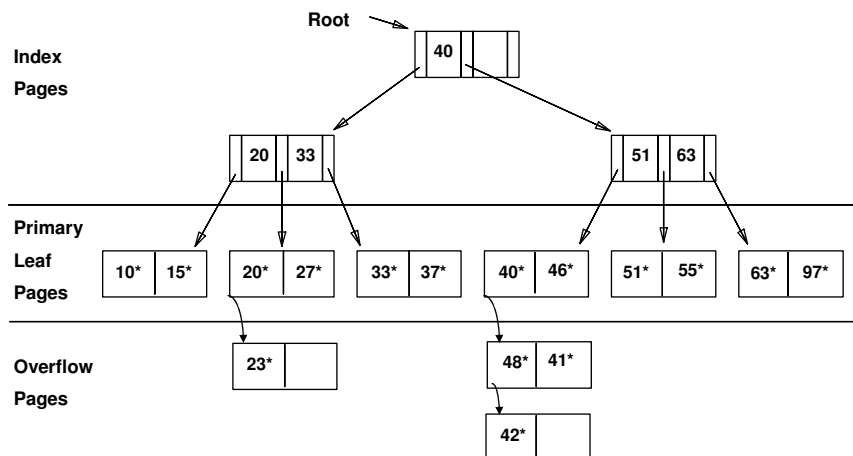
- $k^*$  denotes a data entry with search key value  $k$
- Leaf pages: Each node can hold 2 data entries; no need for 'next-leaf-page' pointers. (Why?)



Winter 2003

37

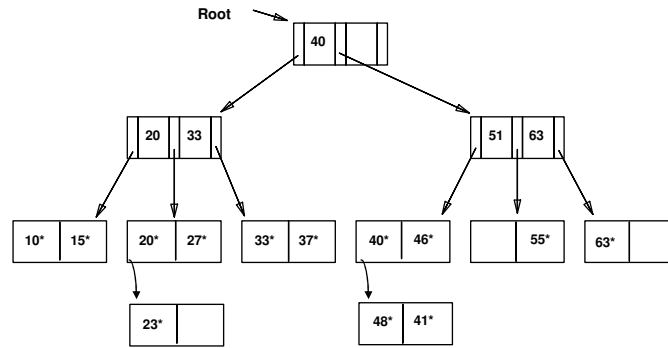
## After Inserting $23^*$ , $48^*$ , $41^*$ , $42^*$ ...



Winter 2003

38

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



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