CS 2451
DBMS Implementation: Indexing and Index Structures

http://www.seas.gwu.edu/~bhagiweb/cs2541

Spring 2020

Instructor: Dr. Bhagi Narahari

General Overview

- Relational model - SQL
  - Formal & commercial query languages
- Functional Dependencies
- Normalization

- **Physical Design**
- Indexing
- Query evaluation
- Query optimization
- ….

Application Oriented

Systems Oriented

Recap: File Organization

- Tables mapped as File
  - Row is a Record
  - Column is field (in record)
- Data stored in secondary storage
  - Disks – organized as number of disk blocks
- Records mapped to disk blocks
- Size of file in disk blocks/pages: N
  - Number of records/tuples/rows: n
  - Size disk block (i.e., page): b bytes
  - Size of record (row): r bytes
  - Blocking factor p = b/r
  - File size N = n/b pages
- Efficiency/performance of a file organization
  - Time for Search, Insert, Delete

Example

- File of 1,000,000 records
- record size 200 bytes
- blocks are 4096 bytes
  - n = 1,000,000
  - r = 200
  - b = 4096
  - Blocking factor, $p = b/r$ = 4096/200 = 20
  - file size = $N = n/p$ = 1,000,000/20 = 50,000 blocks

File Organizations

- File organization determines how records are physically placed on disk
  - heap file: no particular order
  - Sorted file
  - indexed file
    - hash index
    - tree indices
- Efficiency of file organization typically measured in terms of number of disk/SSD accesses to fetch data
  - Why ?

Heap File Performance: Example

- Successful lookup: average ½ N= 25,000
  - worst case is N= n/p= 50,000 disk accesses
  - At 10ms disk access time, this is 500 seconds ~ 8 minutes!
- insertion = 2
- deletion = ½(n/p)+1 = 25,001
  - worst case = 50,001
- header page of pointers can get large
- Heap file summary: not a smart solution!

Lesson 1: better organize the records on the file

- Heap file will not cut it!
- Need to organize physical records on the file in some "smart" manner
  - Sorted file
  - Hash file

Other approaches…

- Sorted File… how long ?
  - Search time: Log (Number of disk blocks)
  - Log (50,000) blocks = 16 IF the blocks are contagious on the disk
    - Big/unrealistic assumption that records are stored in consecutive blocks on disk
- What if you are searching by another field….
  - Sorted by Number, search by name

Lesson 2: do we really need to access the entire file to answer a query ?

- Many queries reference small portion records
  - system should be able to locate these without having to search all records
  - Without having to search through the physical file of records ?
- Create another type of record (pointer?!) which contains subset of the information in the record

"Data" layout in the classroom

- 12 tables, students are the "data"
- Information for each student stored as a "index record"
  - Table Number, Name, GWID
- Assume 6 of these index records fit on one sheet
  - This sheet stored on disk as one page
- Need to find a student using their name

Storage Model 1: Information on where students are located

6 records on one page (one table seating)

Table Number	Student Name	GWID
1	Ryan	G777
1	Ryan	G778
1	Shang	G333
1	Graham	G555
1	Bryson	G234
1	Nicholas	G345

Information on where students are located

Table Number	Student Name	GWID
2	Xiaoyuan	G876
2	Oliver	G123
2	Ramim	G789
2	Linnea	G999
2	Marvin	G235
2	Genevieve	G456

Information on where students are located

Table Number	Student Name	GWID
12	Katie	G567
12	Christina	G456
12	Samuel	G321

Search Key

- When searching for records/rows/tuples, we use one of the attributes – search key
- Name is the search key
- To find a student:
  - Look up one of the sheets
  - Find table number next to the student name – this is the “address” of the student

    Table number is a pointer to the student
- Questions:
  - To find a student how many data sheets do I need to look at ?
  - To find a student how many tables do I need to visit ?

Storage Model 2: Alternate table assignments…New Datasheets.

- Suppose I assigned table numbers according to Name
  - In sorted order

Table Number	Student Name	GWID
1	Ada	G189
1	Alex	G489
1	Alyssa	G389
1	Brian	G289
1	Bryson	G234
1	Cassell	G889

Model #2 table assignments….

- Suppose I assigned table numbers according to Name
  - In sorted order

Table Number	Student Name	GWID
2	Catherine	G275
2	Chen	G475
2	Christina	G458
2	Claire	G175
2	Colin	G875
2	Dylan	G975

Model #2 table assignments….

- Suppose I assigned table numbers according to Name
  - In sorted order

Table Number	Student Name	GWID
12	Will	G367
12	Xiaoyuan	G876
12	Yifei	G667
12	Zach	G567

Searching through new datasheets

- Using this new layout, how many datasheets do I need to look at (worst case) before I find table with the name ?
- Binary search on the datasheets: Log (12) =4
- Search Procedure for name *k*:
  - Find table number *i* with
    name *x* >= *k* AND name *y* at table *i+1 with k < y*

  Go to table i to find record with name *k* (if it exists)
- Question: Can we think of a new datasheet to capture how the tables/datasheets are laid out ?

Model #3: Another "datasheet"/Index record….

Table Number	Student Name	GWID
1	Ada	G189
2	Catherine	G275
3	Ethan	G***
4	Jake	G***
5	Katie	G***
6	Molly	G***

Where is Grady?

Table Number	Student Name	Project
7	Oliver	G***
8	Rachel	G***
9	Sam	G***
10	Stanislav	G***
11	Terry	G***
12	Will	G367

Searching through new datasheets

- Using this new layout, how many datasheets do I need?
  - Just two!!
- Search the datasheets: 2
- Once table number found, go to the one table to find "data" (student)
- This type of datasheet – Model #3- does not have entries for each student
  - **Sparse index** records
  - Able to do this because students are sorted by name

Search for student with GWID= G234

- How many accesses ?
- How many datasheets do I need to find this student ?
  - Assume “storage model 2” – students assigned to tables by sorted names

What happened ?

- Records were sorted by one attribute: name
- Search key/parameter used was a different attribute: GWID

- Sorting records by Name did not help at all when we need to search by GWID
- So what if I want to search by name and search by GWID ?
- Create TWO types of datasheets
  - One as Model 2 or 3: the single sheet with name and table number
  - Second one: 12 datasheets, sorted by GWID and for each GWID the table number

2nd Type of Data sheet/Index records….

- table numbers according to Name in sorted order
- Datasheet sorted by GWID

Table Number	Student Name	GWID
2	Oliver	G123
2	Genevieve	G133
5	Sam	G200
1	Zach	G201
3	Grady	G222
10	Alex	G288

Datasheet A0

Lesson 3: Do something with the index file

- placing additional structure on index files helps search efficiently for desired records
  - Create an index structure on the actual data file

Multi-Level Index….

- So now you have 12 datasheets for GWID
  - Datasheet is sorted by GWID
  - Time to search by GWID is log(12)=4 + 1 to go to table
- Label these datasheets A0 to A11
- Create another "level" of datasheet/index
  - Of Type B

Sheet B0

GWID	Address
G123	A0
G301	A1
G388	A2
G410	A3
G500	A4
G570	A5

Sheet B1

GWID	Address
G600	A6
G675	A7
G710	A8
G800	A9
G880	A10
G910	A11

Keep going….

- Now create another "level" of datasheet…Type C
  - GWID and pointer to Datasheet B_i
  - Tells us range of GWID values at datasheet B_i

GWID	Address
G123	B0
G600	B1

Our Multi-Level Index:

Level 0: One datasheet C0

Level 1: 2 datasheet B0 and B1

Level 2: 12 datasheets G0 to G11

(Level 3) is actual data

Time to search for GWID:

1 at level 0 + 1 at Level 1 + 1 at Level 2 (+ Table)

= No. of Levels + 1 table/disk access

GWID	Address
G123	B0
G600	B1

Our Multi-Level Index:

Level 0

GWID	Address
G123	B0
G600	B1

Level 1

GWID	Address
G123	A0
G301	A1
G388	A2
G410	A3
G500	A4
G570	A5

GWID	Address
G600	A6
G675	A7
G710	A8
G800	A9
G880	A10
G910	A11

Level 2

Table	Name	GWID
2	Oliver	G123
2	Genevie ve	G133
5	Sam	G200
1	Zach	G201
3	Grady	G222
10	Alex	G288

Table	Name	GWID
2	Oliver	G123
2	Genevie ve	G133
5	Sam	G200
1	Zach	G201
3	Grady	G222
10	Alex	G288

Multi-Level Indices and Tree Structures

- Organize the index records as a tree
- But need to make sure it is balanced so we get log(N) height
- Degree of tree ?
  - In our examples each node had 6 children (except for root)
- So it is a k-ary tree…but what is k ? How do we choose ?
- Systems answer: How many index records fit on one disk page ? This is your degree !
  - Since we have to fetch a disk page, might as well pack it with maximum degree possible

Let's summarize:

- We have 'data' stored on disk blocks
  - Students sitting at tables
- We created new 'data' and stored them on a page
  - The tables with assignments and the datasheets (pages of paper)
- This new data is called an **index**
- To find the disk block/page with the data, we can search the index data
  - Looking at the datasheets to find the table where a student sits
- If the data was sorted, then we can organize the index data and do a binary search on it
- If data was sorted, then we can 'compress' the amount of index data and create a sparse index

Some observations…

- When the physical ordering of the data corresponded to the search key, we ended up with one type of index and could create a set of index records that did not have an entry to each record
  - Search key was name, and students sorted by name
- When the search key did not correspond to the key (attribute) used to sort/order the physical records, we needed an index record for each record
  - Search key was student GWID, and we needed entries for all students (and needed to store these on 12 datasheets)
- We can have BOTH indices for the same file
  - Index/datasheet using GWID number, and Index/datasheet using Name
  - This will support queries that search by Name or search by GWID

Some Database Definitions….Indexing

- An ***index*** is a data structure that allows for fast lookup of records in a file.
- An index may also allow records to be retrieved in sorted order.
- Indexing is important for file systems and databases as many queries require only a small amount of the data in a file.

Index Terminology

- The ***data file*** is the file that actually contains the records.
- The ***index file*** is the file that stores the index information.
- The ***search key*** is the set of attributes stored by the index to find the records in the data file.
  - Note that the search key does not have to be unique - more than one record may have the same search key value.
- An ***index entry*** is one index record that contains a search key value and a pointer to the location of the record with that value.

Types of Indexes

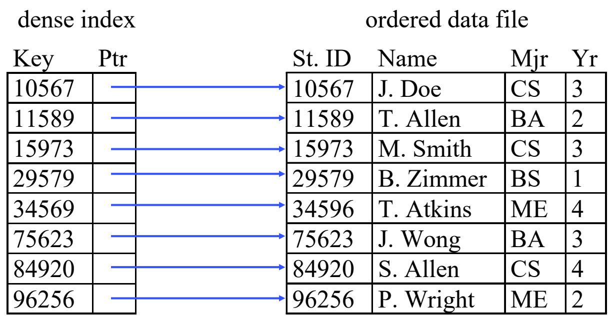
- There are several different types of indexes:
  - Indexes on ***ordered*** versus ***unordered*** files
    - An ordered file is sorted on the search key. Unordered file is not.
  - ***Dense*** versus ***sparse*** indexes
    - A dense index has an index entry for every record in the data file.
    - A sparse index has index entries for only some of the data file records (often indexes by blocks).
  - ***Primary*** (clustering) indexes versus ***secondary*** indexes
    - A primary index sorts the data file by its search key. The search key **DOES NOT** have to be the same as the primary key.
    - A secondary index does not determine the organization of the data file.
  - ***Single-level*** versus ***multi-level*** indexes
    - A single-level index has only one index level.
    - A multi-level index has several levels of indexes on the same file.

Evaluating Index Methods

- Index methods can be evaluated for functionality, efficiency, and performance.
- The ***functionality*** of an index can be measured by the types of queries it supports. Two query types are common:
  - exact match on search key
  - query on a range of search key values
- The ***performance*** of an index can be measured by the time required to execute queries and update the index.
  - Access time, update, insert, delete time
- The ***efficiency*** of an index is measured by the amount of space required to maintain the index structure.

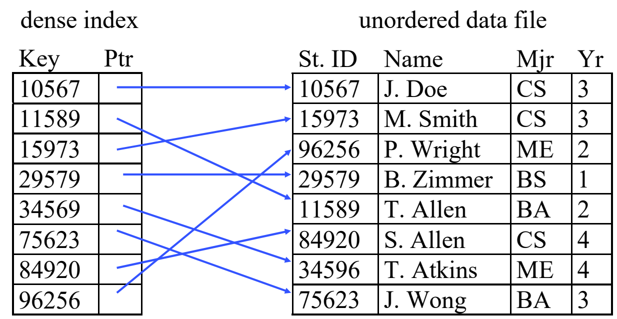
Primary Index on Ordered File

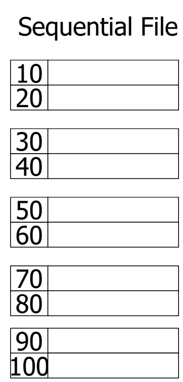
Dense, primary, single-level index on an ordered file.



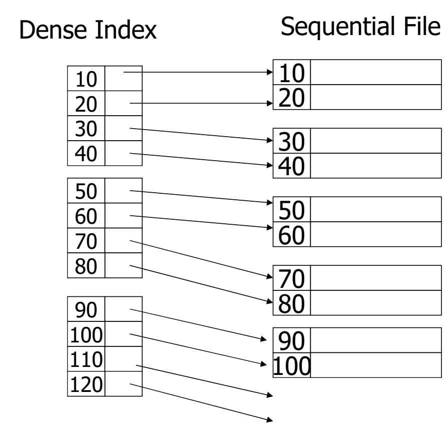
(Secondary) Index on Unordered File

Dense, single-level index on an unordered file.



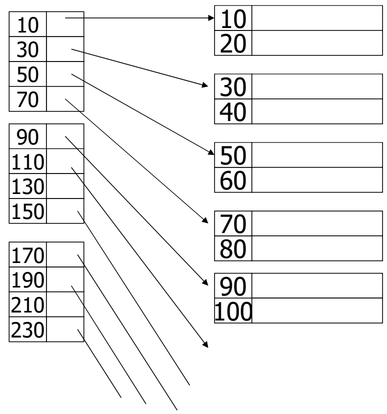


Primary Index…Dense index



Primary Sparse Index

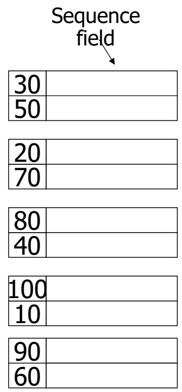
Sequential File



Secondary Indexes

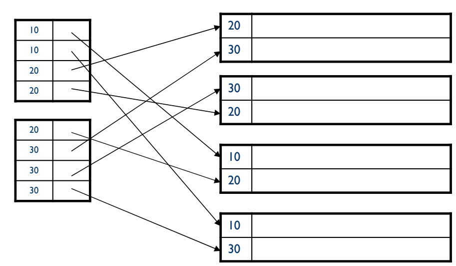
- Applications:
  - index other attributes than primary key
  - index unsorted files (heap files)
  - index clustered data

Secondary indexes

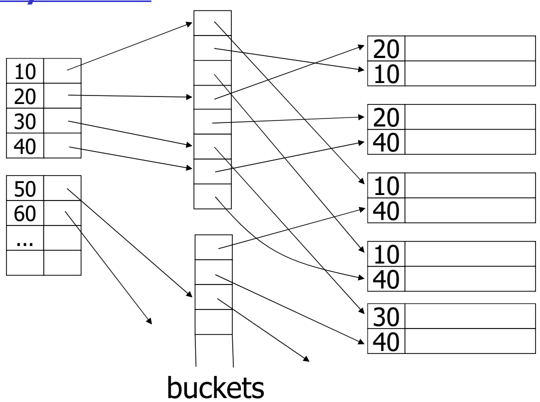


Secondary Indexes

- To index other attributes than primary key
- Always dense (why ?)



Bucket of pointersDuplicate values & secondary indexes



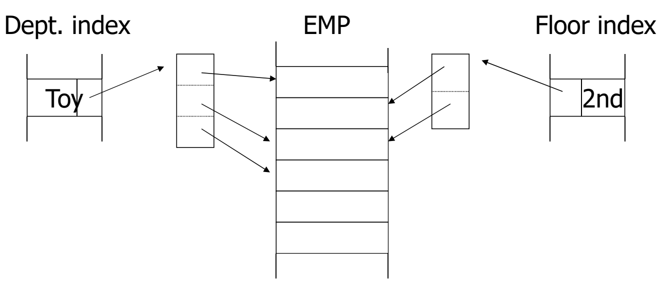
45

Why "bucket" idea is useful

Indexes	Records
Name: primary	EMP (name,dept,floor,...)
Dept: secondary	
Floor: secondary	

46

Query: Get employees in (Toy Dept) ∧ (2nd floor)



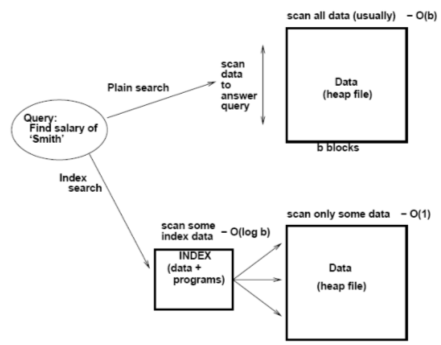
→ **Intersect toy bucket and 2nd Floor bucket to get set of matching EMP's**

47

Lesson 3: Do something with the index file

- placing additional structure on files helps search efficiently for desired records
  - Create an index structure on the actual data file

Indexed Files



Performance using Indexing

- Recall our earlier example:
  - 1,000,000 records, and 50,000 disk blocks
  - Heap file: 50,000 accesses worst case
  - Sorted file of 50,000 blocks: log (50,000)= 16 accesses
- Index record stores Key field 10 bytes, rec. pointer 10 bytes
  - Index record: 20 bytes
  - 4096 page size, we get blocking factor <u>200 index records per page</u>
- Sparse index: 50,000 disk blocks = 50,000 index records=
  - 250 disk blocks to store index records
- Search using binary search on index records to find the pointer to the data block: log (250)=8
- Then fetch the data block: 1 read
- Total: 8+1 = 9 accesses
  - Faster than sorted file

What if the index file itself gets very large….

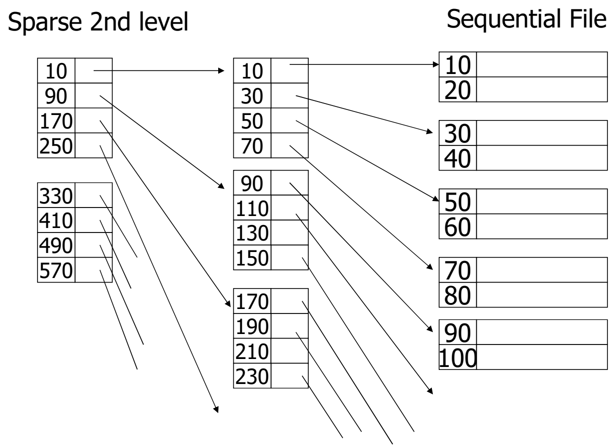
- Example: 1,000,000 records in data file
- Dense index: 1,000,000 index records
- Index record is 20 bytes and 200 fit on disk block
- Therefore, 5000 disk blocks to store index file
- Searching through 5000 disk blocks = 5000 disk reads…not efficient
- Aha...you can sort the index blocks and get log 5000 = 13 disk reads + 1 more to read the data block...this is still slow !
  - A 20ms per disk read, this is over 300ms
  - And assume disk blocks are contigious

- So ???

Organize the index file….

- If index file is large, then create a second level index to this file

2 Level Primary Index



Performance of 2 level Index File

- For sparse index:
- have 250 disk blocks containing index records at Level 1
- We need 250 index entries at level 2:
  - 200 index records per block – 250/200 = 2 blocks to store level 2 index
- To search:
  - Search Level 2 index to find block at Level 1: 2 disk reads
  - Get block from Level 1: 1 disk read
  - Finally, get the data block: 1 disk read
- Total: 4 disk reads

2 Level Dense Index

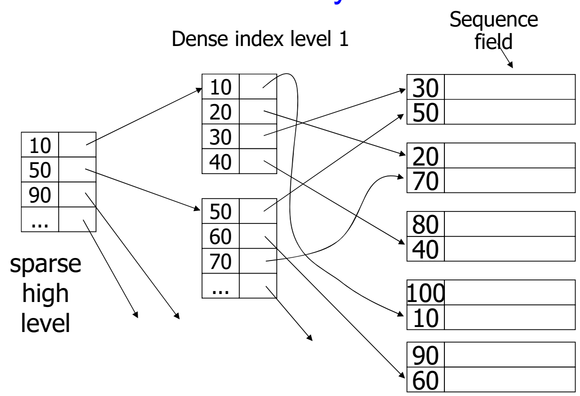
- Recall: 5000 disk blocks at Level 1 secondary index
- How many index records at Level 2
  - 5000 index records
  - With 200 per block, we get 25 disk blocks at Level 2

Multi-level Index - Question

Does the 2nd level index of a dense level 1 index have to be a dense index ?

- Example: If level 1 has 5000 index records, should level 2 index have index entries for each of these 5000 records?

2 Level Secondary indexes



Performance of 2 level Dense index

- 5000 disk blocks at Level 1 secondary index
- Create sparse index at Level 2, with 5000 index records
  - With 200 per block, we get 25 disk blocks at Level 2
- To Search:
  - Search 25 blocks at Level 2: log(25)= 5 disk reads and find the index block to retrieve from Level 1
  - Fetch the level 1 index block
  - Find the data pointer and fetch the data: 1 disk access
  - Total: 7 disk reads

With secondary indexes:

- Lowest level is dense
- Other levels are sparse

Also: Pointers are record pointers

(not block pointers)

59

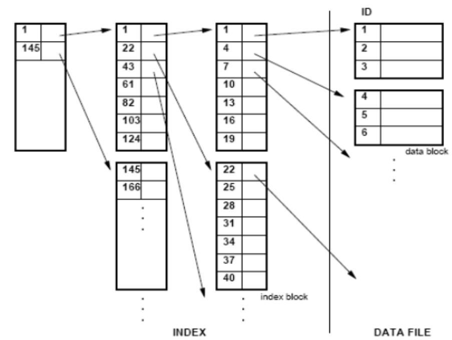
Index Maintenance

- As the data file changes, the index must be updated as well.
- The two operations are ***insert*** and ***delete***.
- Maintenance of an index is similar to maintenance of an ordered file. The only difference is the index file is smaller.
- Techniques for managing the data file include:
  - 1) Using overflow blocks
  - 2) Re-organizing blocks by shifting records
  - 3) Adding or deleting new blocks in the file
- These same techniques may be applied to both sparse and dense indexes.

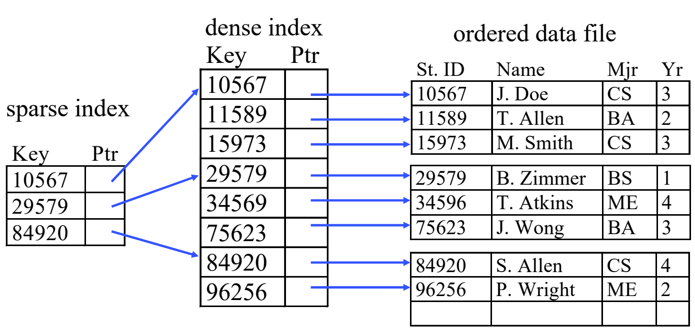
Multi level Index Files

- If index file is large, then create a second level index to this file
- If second level index is also large, then create a third level index to the second level index
- If third level index is also large, then create a 4th level…
- If fourth .....
- When do you stop ?
- When the final index is ONE disk block/page !!

Multilevel Index



Multi-level Index on an Ordered File



Multi-level Index

- A ***multi-level index*** has more than one index level for the same data file.
  - Each level of the multi-level index is smaller, so that it can be processed more efficiently.
  - The first level of a multi-level index may be either sparse or dense, but all higher levels must be sparse. Why?
- Having multiple levels of index increases the level of indirection, but is often quicker because the upper levels of the index may be stored entirely in memory.
  - However, index maintenance time increases with each level.

Multilevel Index for the Dense index

- Level 1 had 5000 disk blocks to store the index records
  - For 1,000,000 data records and 1,000,000 index records
- Level 2 had 250 disk blocks to store the index records to Level 1
  - To store 5000 index records pointing to the 5000 disk blocks of Level 1 index
- Level 3 we need 250 index records, and therefore 2 disk blocks
- Level 4 we need 2 index records and therefore one disk block – the root node !
- Time: Read once from each level and then from data file
  - 5 reads

Implementing Index structures: Tree Structures and Multilevel Index

- Natural correspondence between the two
- What if we use "standard" binary search trees ?
- Need concept of "balanced" tree
  - B+ trees : variation of B-trees tailored for DBMS operations
- Have you looked at the following during discussion of data structures:
  - 2-3 trees or search trees
  - Hash tables