• Introduction

→ Disks are very slow compared with memories

→ But disks provides enormous capacity at much less cost

Example:

* getting information from RAM (120 nanoseconds)
* " " from disk (30 milliseconds)

Analogy - find an information

- RAM ↔ Index of a book (20 seconds)
- Disk ↔ Library (5M seconds, 58 days)

→ keeping the same proportion

* File Structure is a combination of representation for data in files and of operations for accessing the data. It allows application to read, write and modify data. Also supports finding the data that matches some search criteria

→ an improvement in design may make the application 100x faster

→ the details of the representation of the data and implementation determine the efficiency of the file structure.

- Ideally, we want:

→ get what we need in one access, or with as few accesses as possible

EX: binary search finds a record among 50 thousand with at most sixteen comparisons

* but 16 comparisons is too much time in disk
* we need only two or 3 comparisons at most

→ Indexes added to files

* keep lists of keys and pointers in a smaller file that could be searched more quickly

key + pointer $\xrightarrow{\text{access}}$ primary file

* Indexes can be sequential, then on alternative are Trees

* B-tree : it grows bottom up
  excellent access performance, but no longer could a file be accessed sequentially with efficiency

* Solution: adding a linked list at the bottom level of the B-tree ⇒ B^+ tree

B^+ tree = Btree + Linked list

* B^+-tree / B-tree can find one file among 1M with only three or four operations to the disk
  - as add and delete entries, performance stays the same

→ Single Request: hashing

* files that do not change size greatly over time
* hashed indexes

Indexes ⟹ Trees ⟹ Hashing

• Files

→ given the ability to create, open and close files, and to seek, read and write, we can perform operations with files (Cap 2)

• Chapter 4 - Fundamental File Structure Concepts

→ file structures make data "persistent".

* A program can create data in memory and store it in a file
* another program can read the file and recreate the data
* basic unit is the "field", which contains a single value
* Fields are organized into aggregates
  → many copies of a single field (array)
  → list of different fields (a record)

Let's see some ways/representations a file is organized.

(A) A stream file

• Suppose we want to store names and addresses of a collection of people. Input

Mary Ames	Alan Mason
123 Maple	90 Eastgate
Stillwater, OK 74075	Ada, OK 74820

→ When we list the output to our screen we see is

Ames Mary 123 Maple Stillwater Ok 74075 Mason Alan 90 East Ute Ada Ok 74820

↳ Stream of bytes containing no added information

But Once the program put all together as a single byte Stream, there is no way to get it apart again

→ we lost the integrity of the fields

→ a field is the smallest logically meaningful unit of information in a file

We need: to organize the file in some way the lets us keep the information divided into fields

(B) Field Structures

↳ there are many ways of adding structure to files. Most common:

- force the fields into a predictable lenght
- begin each field with a lenght indicator
- Place a "delimiter" at the end of the field to separate it from the next field
- use Keyword = value expressions

(01) Method: Fix the lenght of the fields

→ if we force the fields into predictable lenghts, we can pull them back out of the file simply by counting our way to the end of the file

- define a struct to hold these fixed lenghts

```
struct Person {
    char last [11];
    char first [11];
    char address [16];
    char city [16];
    char state [3];
    char zip [10];
}
```

Requires:
= 10 + 10 + 15 + 15 + 2 + 9
= 61 bytes

→ Using this structure changes our output:

FILE

Ames	Mary	123 Maple	Stillwater	OK 74075
Mason	Alan	90 Eastgate	Ada	OK 74820

→ Disadvantage: adding all the padding required to bring the fields up to a fixed lenght makes the file much larger

→ Because of this, this method is inappropriate for data that contains a lot of variability in

the lenght of the fields, such as names and addresses

- But if every field is already fixed in a lenght, or if there is very little variation in field lenghts, it is a very good solution

(02) Method: Begin each field with a Lenght Indicator

→ Another way is to store the field lenght just ahead of the field

04 Ames 04 Mary 09 123 Mopple 10 Stillwater 02 Ok 05 74075
05 Mason 04 Allan 11 90 Eastgate 03 Ada 02 Ok 05 74820

→ if the fields are not to long (less than 256 bytes) it is possible to store the lenght in a single byte at the start of each field

→ We refer to these fields as lenght-based

(03) Method: Separate the fields with Delimiters

↳ choose some special character or sequence of characters that will not appear within a field

→ in many instances "white-space characters" (blank, new line, tab) are used

→ Can also use Vertical bar, since white space can appear in the fields

Look: weitstorm.cpp

adaptor em c!

FILE [Ames | Mary | 123 Mapple | Stillwater | Ok | 74075 |
Mason | Alan | 90 Eastgate | Ada | Ok | 74820 |]

(04) Method: Use a "keyword = value" Expression to Identify Fields

File [last = Ames | first = Mary | address = 123 Mapple | city = Stillwater | state = Ok | zip = 74075 |]

↳ It has an advantage → a field provides information about itself (self-describing structure)

→ it is also good for dealing missing values.
If a field is missing, the keyword is simply not there.

→ Disadvantage: this format also wastes a lot of space, because of the keywords

(C) Record Structures

↳ Record : a set of fields that belong together when the file is viewed in terms of a higher level of organization

→ a record in a file represents a structure data object

Writting a record ↔ saving the state of an obj
reading a record ↔ restore the state of the obj

Object → data residing in memory
record → data residing in a file

→ Most often used methods for organizing records of a file :

- require that the records be a predictable number of bytes in a lenght
- require that the records be a predictable number of fields
- begin each record with a lenght indicator consisting of a count of the number of bytes

that the record contains.

- use a second file to keep track of the beginning byte address of each record
- place a delimiter at the end of each record to separate it from the next record

(01) Method: Fixed-Lenght Records (149 pg)

↳ A fixed-length record file is one with each record contains the same number of bytes

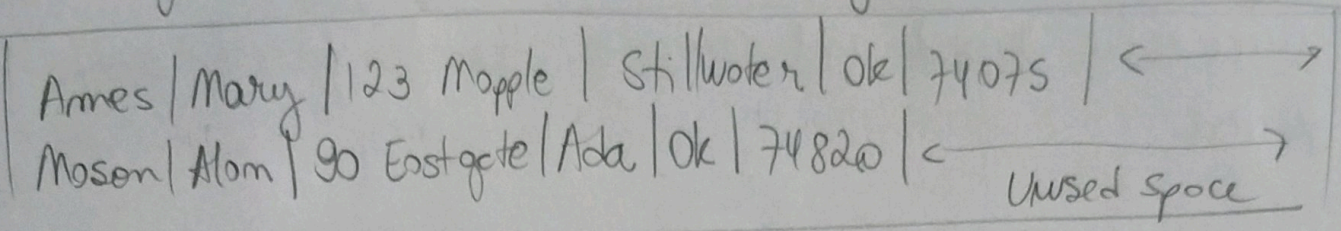
- analogous to the first method for fields
- are among the most used methods
- fixing the number of bytes in a record does not imply that the size or number of fields in the record must be fixed

EX:

a) Counting bytes fixed-length records with fixed-lenght fields

Ames	Mary	123 Maple	Stillwater	OK74075
Mason	Alan	90 Eastgate	Ada	OK74820

b) fixed-lenght records with varidole-length records



(02) Method: Make records a Predictable Number of Fiels

↳ we specify that it will contain a fixed number of fields

c) six-fields per record:

[Ames | Mary | 123 Mapple | Stillwater | Ok | 74075 | Moson ...]

(03) Begin each record with a Lenght Indicator

↳ beginning each record with a field containing an integer that indicates how many bytes there are in the record

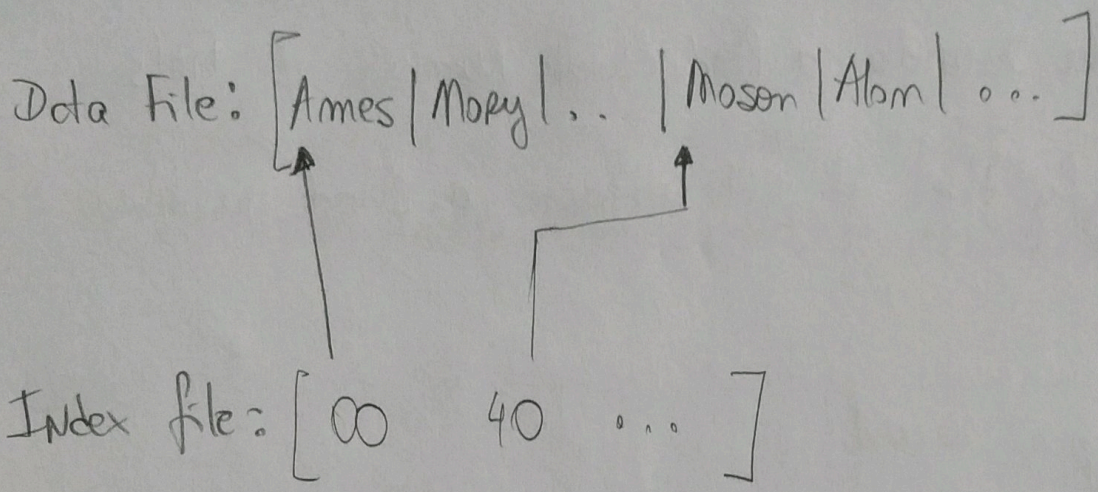
[40 Ames | Mary | 123 Mople | Stillwater | Ok | 74075 | 36 Moson | Alom ...]

(04) Method: Use an Index to keep Track of Addresses

↳ use an index to keep a byte offset for each record

↳ byte offsets allow us to find the beginning of each successive record and compute the lenght of each record

Ex:



(05) Method: Place a delimiter at the End of each Record

↳ at a record level, works the same as with fields

↳ a common choice of record delimiter is the end of line character (\n), or #

[Ames|Mary|...|OK|74075|#Mason|Alan|...]

Obs: If we want to put a lenght indicator at the beginning of every record (before any other fields), we must know the sum of the lenghts of the fields before we can begin writing the record to the file (buffer)

Obs 2: represent record lenght can be in the form of a 2-byte binary integer before each record.

EX:

File [40 AMES|MARY|...|74075|36 Mason|Alan|...|74820]

↳ each record has a record lenght field preceding the data fields. this field is delimited by a blank.

Obs 3:

① Mixing Number and Characters : Use of a File Dump

↳ File dumps give us the ability to look inside a file at the actual bytes that are stored there.

↳ Considering previous examples, the first record has 40 characters, including delimiters.
the actual bytes stored in the file are:

	Dec	Hex	ASCII
a) 40 stored as ASCII char:	40	34 | 30	4 0
b) 40 stores as 2-byte Integer	40	00 | 28	to "("

- Summary

↳ the lowest level of a file organization is a stream of bytes

↳ fields: fundamental pieces of information

↳ fields are grouped together to form records

↳ Recognizing fields and records requires that we impose structure on the data in the file

↳ there are many ways to separate one field from the next and one record from the next:

- fix the lenght of each record
- begin each field or record with a count of the number of bytes that it contains
- Use delimiters to mark the division between entities

Record structure with a lenght indicator at the beginning of each record to develop programs for writing and reading a simple file with objects.

141 - 176