CSI2131 FILE MANAGEMENT

Prof. Lucia Moura

Winter 2003

LECTURE 1: INTRODUCTION TO FILE MANAGEMENT

Contents of today's lecture:

- Introduction to file structures
- History of file structure design
- Course contents and organization

References :

- FOLK, ZOELLICK AND RICCARDI, File Structures, 1998. Sections 1.1 and 1.2.
- Course description handout (for course contents and organization)

Introduction to File Structures

- **Data processing from a computer science perspective:**
  - Storage of data
  - Organization of data
  - Access to data
  - Processing of data

  This will be built on your knowledge of Data Structures.

- **Data Structures vs File Structures**

  Both involve :

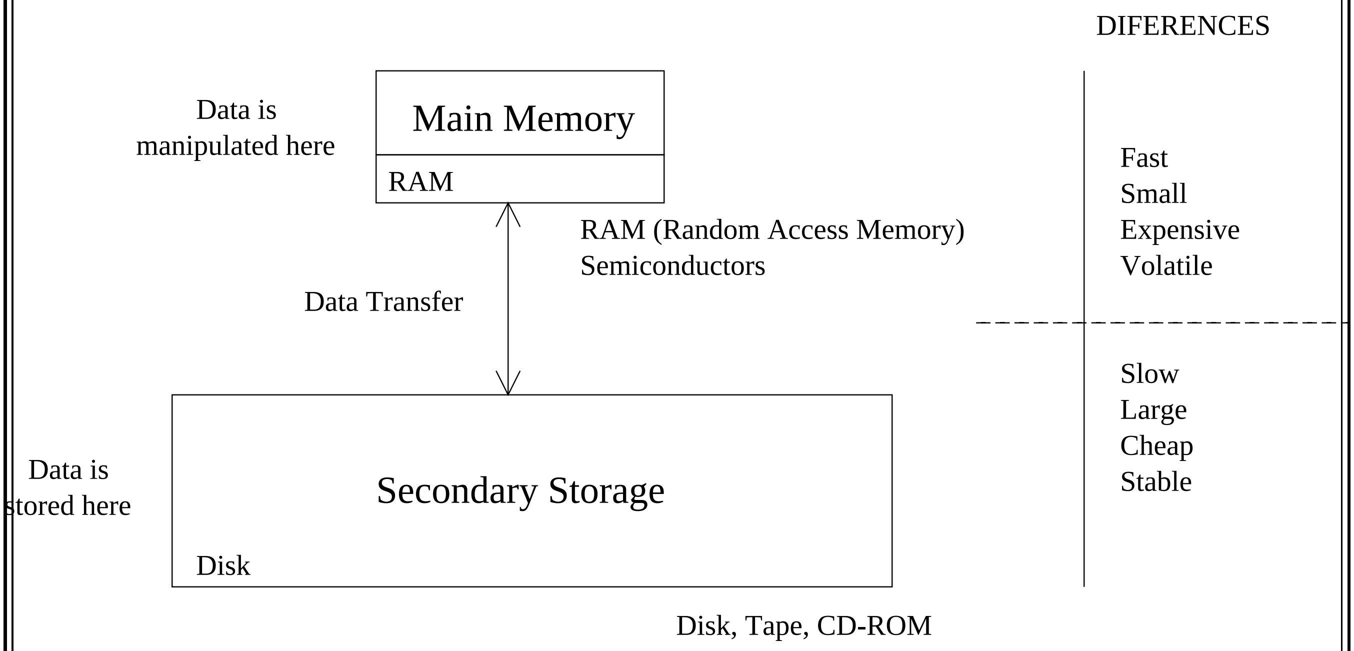
  Representation of Data
  +
  Operations for accessing data

  Difference :

  Data Structures deal with data in main memory.

  File Structures deal with data in secondary storage (Files)

Computer Architecture



How fast is main memory in comparison to secondary storage ?

Typical time for getting info from:

main memory:	$\sim$ 120 nanoseconds	$=$ 120 x 10^{-9} secs
magnetics disks:	$\sim$ 30 milliseconds	$=$ 30 x 10^{-3} secs

An analogy keeping same time proportion as above:

Looking at the index of a book: 20 secs

versus

Going to the library: 58 days

Main Memory

- Fast (since electronic)
- Small (since expensive)
- Volatile (information is lost when power failure occurs)

Secondary Storage

- Slow (since electronic and mechanical)
- Large (since cheap)
- Stable, persistent (information is preserved longer)

Goal of the file structure and what we will study in this course:

- Minimize number of trips to the disk in order to get desired information. Ideally get what we need in one disk access or get it with as few disk accesses as possible.
- Grouping related information so that we are likely to get everything we need with only one trip to the disk (e.g. name, address, phone number, account balance).

History of File Structure Design

1. In the beginning ... it was the tape
   - **Sequential access**
   - Access cost proportional to size of file
     [Analogy to sequential access to array data structure]

2. Disks became more common
   - **Direct access** [Analogy to access to position in array - binary search in sorted arrays]
   - **Indexes** were invented
     - list of keys and pointers stored in small file
     - allows direct access to a large primary file

     Great if index fits into main memory.
     As a file grows we have the same problem we had with a large primary file.

3. Tree structures emerged for main memory (1960's)
   - Binary search trees (BST's)
   - **Balanced**, self adjusting BST's : e.g. AVL trees (1963)

4. A tree structure suitable for files was invented : **B trees** (1979) and **B+ trees**

   Good for accessing millions of records with 3 or 4 disk accesses.

5. What about getting info with a single request ?

   - **Hashing Tables** (Theory developed over 60's and 70's but still a research topic)

     Good when files do not change to much in time.

   - **Extendible, dynamic hashing** (late 70's and 80's)

     One or two disk accesses even if file grows dramatically

Course Contents and Organization

- Introduction to file management. Fundamental file processing operations. (Chapters 1 and 2)
  Managing files of records. Sequential and direct access. (Chapters 4 and 5)
- Secondary storage, physical storage devices: disks, tapes and CD-ROM. (Chapter 3)
  System software: I/O system, file system, buffering. (Chapter 3)
- File compression: Huffman and Lempel-Ziv codes. Reclaiming space in files. Internal sorting, binary searching, keysorting. (Chapter 6)
- File Structures:
  - Indexing. (Chapter 7)
  - Co-sequential processing and external sorting. (Chapter 8)
  - Multilevel indexing and B trees. (Chapter 9)
  - Indexed sequential files and B+ trees. (Chapter 10)
  - Hashing. (Chapter 11)
  - Extendible hashing. (Chapter 12)

Chapters above refer to the textbook:
Folk, Zoellick and Riccardi, File Structures, 1998.

Refer to the "course description handout" for course organization.

LECTURE 2: FUNDAMENTAL FILE PROCESSING OPERATIONS

Contents of today's lecture:

- Sample programs for file manipulation
- Physical files and logical files
- Opening and closing files
- Reading from files and writing into files
- How these operations are done in C and C++
- Standard input/output and redirection

References :

- Folk, Zoellick and Riccardi, File Structures, 1998. Chapter 2

Sample programs for file manipulation

A program to display the contents of a file on the screen:

- Open file for input (reading)
- While there are characters to read from the input file :

  Read a character from the file

  Write the character to the screen
- Close the input file

A C program (which is also a valid C++ program) for doing this task:

```
// listc.cpp
#include <stdio.h>

main() {
    char ch;
    FILE * infile;

    infile = fopen("A.txt","r");

    while (fread(&ch,1,1,infile) != 0)
        fwrite(&ch,1,1,stdout);
    fclose(infile);
}
```

A C++ program for doing the same task:

```
// listcpp.cpp
#include <fstream.h>

main() {
    char ch;
    fstream infile;

    infile.open("A.txt",ios:in);
    infile.unsetf(ios::skipws);
         // set flag so it doesn't skip white space

    infile >> ch;
    while (! infile.fail())  {
        cout << ch ;
        infile >> ch ;
    }
    infile.close();
}
```

Physical Files and Logical Files

physical file: a collection of bytes stored on a disk or tape

logical file: a “channel” (like a telephone line) that connects the program to a physical file

- The program (application) sends (or receives) bytes to (from) a file through the logical file. The program knows nothing about where the bytes go (came from).

- The operating system is responsible for associating a logical file in a program to a physical file in disk or tape. Writing to or reading from a file in a program is done through the operating system.

Note that from the program point of view, input devices (keyboard) and output devices (console, printer, etc) are treated as files - places where bytes come from or are sent to.

There may be thousands of physical files on a disk, but a program only have about 20 logical files open at the same time.

The physical file has a name, for instance `myfile.txt`

The logical file has a logical name used for referring to the file inside the program. This logical name is a variable inside the program, for instance `outfile`

In C programming language, this variable is declared as follows:

```
FILE * outfile;
```

In C++ the logical name is the name of an object of the class `fstream`:

```
fstream outfile;
```

In both languages, the logical name `outfile` will be associated to the physical file `myfile.txt` at the time of **opening** the file as we will see next.

Opening Files

Opening a file makes it ready for use by the program.
Two options for opening a file :

- open an **existing** file
- create a **new** file

When we open a file we are positioned at the beginning of the file.

How to do it in C:

```
FILE * outfile;
outfile = fopen("myfile.txt", "w");
```

The first argument indicates the physical name of the file.
The second one determines the “mode”, i.e. the way, the file is opened.
The mode can be:

- `"r"`: open an existing file for input (reading);
- `"w"`: create a new file, or truncate existing one, for output;
- `"a"`: open a new file, or append an existing one, for ouput;
- `"r+"`: open an existing file for input and output;
- `"w+"`: create a new file, or truncate an existing one, for input and output;
- `"a+"`: create a new file, or append an existing one, for input and output.

How to do it in C++:

```
fstream outfile;
outfile.open("myfile.txt",ios::out);
```

The second argument is an integer indicating the mode.
Its value is set as a "bitwise or" (operator `|`) of constants defined in the class `ios`:

- `ios::in` open for input;
- `ios::out` open for output;
- `ios::app` seek to the end of file before each write;
- `ios::trunc` always create a new file;
- `ios::nocreate` fail if file doesn't exist;
- `ios::noreplace` create a new file, but fail if it already exists;
- `ios::binary` open in binary mode (rather than text mode).

Exercise: Open a physical file `"myfile.txt"` associating it to the logical file `"afile"` and with the following capabilities:

1. input and output (appending mode):
   `afile.open("myfile.txt", ios::in|ios::app);`
2. create a new file, or truncate existing one, for output:
3. open an existing file for input and output, no creation allowed:

Closing Files

This is like "hanging up" the line connected to a file.

After closing a file, the logical name is free to be associated to another physical file.

Closing a file used for output guarantees that everything has been written to the physical file.
We will see later that bytes are not sent directly to the physical file one by one; they are first stored in a buffer to be written later as a block of data. When the file is closed the leftover from the buffer is flushed to the file.

Files are usually closed automatically by the operating system at the end of program's execution.
It's better to close the file to prevent data loss in case the program does not terminate normally.

In C :

```
fclose(outfile);
```

In C++ :

```
outfile.close();
```

Reading

Read data from a file and place it in a variable inside the program.

A generic **Read** function (not specific to any programming language):

```
Read(Source_file, Destination_addr, Size)
```

Source_file: logical name of a file which has been opened
Destination_addr: first address of the memory block were data should be stored
Size: number of bytes to be read

In C (or in C++ using C streams):

```
char c;      // a character
char a[100]; // an array with 100 characters
FILE * infile;
  ⋮
infile = fopen("myfile.txt","r");
fread(&c,1,1,infile); /* reads one character */
fread(a,1,10,infile); /* reads 10 characters */
```

fread:

1st argument:	destination address (address of variable **c**)
2nd argument:	element size in bytes (a **char** occupies 1 byte)
3rd argument:	number of elements
4th argument:	logical file name

In C, read and write operations to files are supported by various functions: **fread, fget, fwrite, fput, fscanf, fprinf**.

In C++ :

```
char c;
char a[100];
fstream infile;
infile.open("myfile.txt",ios::in);
infile >> c;     // reads one character
infile.read(&c,1);
    // alternative way of reading one character
infile.read(a,10); // reads 10 bytes
```

Note that in the C++ version, the operator **>>** communicates the same info at a higher level. Since **c** is a char variable, it's implicit that only 1 byte is to be transferred.

C++ fstream also provide the **read** method, corresponding to **fread** in C.

Writing

Write data from a variable inside the program into the file.
A generic `Write` function :

```
Write (Destination_File, Source_addr, Size)
```

`Destination_file`: logical file name of a file which has been opened
`Source_addr`: first address of the memory block where data is stored
`Size`: number of bytes to be written

In C (or in C++ using C streams) :

```
char c; char a[100];
FILE * outfile;
outfile = fopen("mynew.txt","w");
/* omitted initialization of c and a */
fwrite(&c,1,1,outfile);
fwrite(a,1,10,outfile);
```

In C++ :

```
char c; char a[100];
fstream outfile;
outfile.open("mynew.txt",ios::out);
/* omitted initialization of c and a */
outfile << c;
outfile.write(&c,1);
outfile.write(a,10);
```

Detecting End-of-File

When we try to read and the file has ended, the read was unsuccessful. We can test whether this happened in the following ways :

In C : Check whether `fread` returned value 0.

```
int i;
i = fread(&c,1,1,infile); // attempted to read
if (i==0) // true if file has ended
     ...
```

in C++: Check whether `infile.fail()` returns `true`.

```
infile >> c; // attempted to read
if (infile.fail()) // true if file has ended
     ...
```

Alternatively, check whether `infile.eof()` returns `true`.
Note that `fail` indicates that an operation has been unsuccessful, so it is more general than just checking for end of file.

Logical file names associated to standard I/O devices and re-direction

purpose	**default meaning**	**logical name**	
		in C	**in C++**
Standard Output	Console/Screen	`stdout`	`cout`
Standard Input	Keyboard	`stdin`	`cin`
Standard Error	Console/Screen	`stderr`	`cerr`

These streams don't need to be open or closed in the program.

Note that some operating systems allow this default meanings to be changed via a mechanism called **redirection**.

In UNIX and DOS :

Suppose that `prog.exe` is the executable program.
Input redirection (standard input becomes file `in.txt`):

```
prog.exe < in.txt
```

Output redirection (standard output becomes file `out.txt`. Note that standard error remains being console):

```
prog.exe > out.txt
```

You can also do:

```
prog.exe < in.txt > out.txt
```

LECTURE 3: MANAGING FILES OF RECORDS

Contents of today's lecture:

- Field and record organization (textbook: Section 4.1)
- Sequential search and direct access (textbook: Section 5.1)
- Seeking (textbook: Section 2.5)

Reference: FOLK, ZOELLICK AND RICCARDI, File Structures, 1998. Sections 4.1, 5.1, 2.5.

Files as Streams of Bytes

So far we have looked at a file as a stream of bytes.

Consider the program seen in the last lecture :

```
// listcpp.cpp
#include <fstream.h>
main() {
    char ch;
    fstream infile;
    infile.open("A.txt",ios:in);
    infile.unsetf(ios::skipws);
        // set flag so it doesn't skip white space
    infile >> ch;
    while (! infile.fail()) {
        cout << ch;
        infile >> ch;
    }
    infile.close();
}
```

Consider the file example: `A.txt`

```
87358CARROLLALICE IN WONDERLAND      <nl>
03818FOLK   FILE STRUCTURES          <nl>
79733KNUTH  THE ART OF COMPUTER PROGR<nl>
86683KNUTH  SURREAL NUMBERS          <nl>
18395TOLKIENTHE HOBITT               <nl>
```

(above we are representing the invisible newline character by `<nl>`)

Every stream has an associated **file position**.

- When we do `infile.open("A.txt",ios::in)` the **file position** is set at the beginning.
- The first `infile >> ch;` will read `8` into `ch` and increment the file position.
- The next `infile >> ch;` will read `7` into `ch` and increment the file position.
- The 38th `infile >> ch;` will read the newline character (referred to as `'\n'` in C++) into `ch` and increment the file position.
- The 39th `infile >> ch;` will read `0` into `ch` and increment the file position, and so on.

A file can been seen as

1. a stream of bytes (as we have seen above); or
2. a collection of records with fields (as we will discuss next ...).

Field and Record Organization

Definitions :

Record	=	a collection of related fields.
Field	=	the smallest logically meaningful unit of information in a file.
Key	=	a subset of the **fields** in a record used to identify (uniquely, usually) the record.

In our sample file "A.txt" containing information about books:
Each line of the file (corresponding to a book) is a record.
Fields in each record: ISBN Number, Author Name and Book Title.

Primary Key: a key that uniquely identifies a record.
Example of primary key in the book file:

Secondary Keys: other keys that may be used for search
Example of secondary keys in the book file:

Note that in general not every field is a key (keys correspond to fields, or combination of fields, that may be used in a search).

Field Structures

1. Fixed-length fields:
   Like in our file of books (field lengths are 5, 7, and 25).

```
87358CARROLLALICE IN WONDERLAND
03818FOLK   FILE STRUCTURES
79733KNUTH  THE ART OF COMPUTER PROGR
```

2. Field beginning with length indicator:

```
058735907CARROLL19ALICE IN WONDERLAND
050381804FOLK15FILE STRUCTURES
```

3. Place delimiter at the end of fields:

```
87359|CARROLL|ALICE IN WONDERLAND|
03818|FOLK|FILE STRUCTURES|
```

4. Store field as `keyword = value` (self-describing fields):

```
ISBN=87359|AU=CARROLL|TI=ALICE IN WONDERLAND|
ISBN=03818|AU=FOLK|TI=FILE STRUCTURES|
```

   Although the delimiter may not always be necessary here, it is convenient for separating a key value from the next keyword.

Field structures: advantages and disadvantages

Type	**Advantages**	**Disadvantages**
Fixed	Easy to Read/Store	Waste space with padding
with length indicator	Easy to jump ahead to the end of the field	Long fields require more than 1 byte to store length (when maximum size is > 256)
Delimited Fields	May waste less space than with length-based	Have to check every byte of field against the delimiter
Keyword	Fields are self describing, allows for missing fields.	Waste space with keywords

Record Structures

1. Fixed-length records.
   It can be used with fixed-length fields, but can also be combined with any of the other variable length field structures, in which case we use padding to reach the specified length.

   Examples:

   Fixed-length records combined with fixed-length fields:

```
87358CARROLLALICE IN WONDERLAND
03818FOLK   FILE STRUCTURES
79733KNUTH  THE ART OF COMPUTER PROGR
```

   Fixed-length records combined with variable-length fields:

   delimited fields:

```
87359|CARROLL|ALICE IN WONDERLAND
03818|FOLK|FILE STRUCTURES
79733|KNUTH|THE ART OF COMPUTER PROGR
```

   fields with length indicator:

   $\underline{05}$87359$\underline{07}$CARROLL$\underline{19}$ALICE IN WONDERLAND
   $\underline{05}$03818$\underline{04}$FOLK$\underline{15}$FILE STRUCTURES

2. Records with fixed number of fields (variable-length)

   It can be combined with any of the variable-length field structure.

   Examples: Number of fields per record = 3.

   with delimited fields:

```
87359|CARROLL|ALICE IN WONDERLAND|03818|FOLK|···
```

   with fields with length indicator:

```
058735907CARROLL19ALICE IN WONDERLAND0503818···
```

   In the situations above, how would the program detect that a record has ended ?

3. Record beginning with length indicator.

   Example:

   with delimited field:

```
3387359|CARROLL|ALICE IN WONDERLAND
2603818|FOLK|FILE STRUCTURES
```

   Can this method be combined with fields having length indicator or fields having keywords?

4. Use an index to keep track of addresses

   The index keeps the byte offset for each record; this allows us to search the index (which have fixed length records) in order to discover the beginning of the record.

   datafile:

```
87359|CARROLL|ALICE IN WONDERLAND|03818|FOLK|···
```

   Complete information on the index file:

   indexfile:

5. Place a delimiter at the end of the record.
   The end-of-line character is a common delimiter, since it makes the file readable at our console.

```
87358|CARROLL|ALICE IN WONDERLAND|<nl>
03818|FOLK|FILE STRUCTURES|<nl>
79733|KNUTH|THE ART OF COMPUTER PROGR|<nl>
```

Summary :

Type	**Advantages**	**Disadvantages**
Fixed Length Record	Easy to jump to the i-th record	Waste space with padding
Variable Length Record	Saves space when record sizes are diverse	Cannot jump to the i-th record, unless through an index file

Sequential Search and Direct Access

Search for a record matching a given key.

- **Sequential Search**
  Look at records sequentially until matching record is found.
  Time is in $O(n)$ for n records.

  Example when appropriate :
  Pattern matching, file with few records.

- **Direct Access**
  Being able to seek directly to the beginning of the record.
  Time is in $O(1)$ for n records.

  Possible when we know the Relative Record Number (RRN):
  First record has RRN 0, the next has RRN 1, etc.

Direct Access by RRN

Requires records of fixed length.

RRN = 30 (31st record)
record length = 101 bytes
So, byte offset = ________

Now, how to go directly to byte ________ in a file ?
By **seeking** ...

Seeking

Generic seek function :

```
Seek(Source_File, Offset)
```

Example :

```
Seek(infile, 3030)
```

Moves to byte 3030 in `file`.

<u>In C style</u> :

Function prototype:

```
int fseek(FILE *stream, long int offset, int origin);
```

`origin`: 0 = `fseek` from the <u>beginning</u> of file
1 = `fseek` from the <u>current</u> position
2 = `fseek` from the <u>end</u> of file

Examples of usage:

```
fseek(infile,0L,0); // moves to the beginning
                                //of the file

fseek(infile,0L,2); // moves to the end of the file

fseek(infile,-10L,1); // moves back 10 bytes from
                          // the current position
```

In C++ :

Object of class **fstream** has two file pointers :

- **seekg** = moves the get pointer.
- **seekp** = moves the put pointer.

General use:

```
file.seekg(byte_offset,origin);
file.seekp(byte_offset,origin);
```

Constants defined in class ios:

origin: ios::beg = **fseek** from the beginning of file
ios::cur = **fseek** from the current position
ios::end = **fseek** from the end of file

The previous examples, shown in C style, become in C++ style:

```
infile.seekg(0,ios::beg);

infile.seekg(0,ios::end);

infile.seekg(-10,ios::cur);
```

Consider the following sample program:

```
#include <fstream.h>
int main() {
  fstream myfile;
  myfile.open("test.txt",ios::in|ios::out|ios::trunc
                 |ios::binary);
  myfile<<"Hello,world.\nHello, again.";
  myfile.seekp(12,ios::beg);
  myfile<<’X’<<’X’;
  myfile.seekp(3,ios::cur);
  myfile<<’Y’;
  myfile.seekp(-2,ios::end);
  myfile<<’Z’;
  myfile.close();
  return 0;
}
```

Show **"test.txt"** after the program is executed:

| | | | | | | | | | | | | | | | | | | | | | | | | | | |

Remove **ios::binary** from the specification of the opening mode. Show **test.txt** after the program is executed under DOS:

| | | | | | | | | | | | | | | | | | | | | | | | | | | |

LECTURE 4: SECONDARY STORAGE DEVICES - MAGNETIC DISKS

Contents of today's lecture:

- Secondary storage devices
- Organization of disks
- Organizing tracks by sector
- Organizing tracks by blocks
- Nondata overhead
- The cost of a disk access
- Disk as a bottleneck

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 3.1.

Secondary Storage Devices

Since secondary storage is different from main memory we have to understand how it works in order to do good file designs.

Two major types of storage devices:

- Direct Access Storage Devices (DASDs)

  - Magnetic Disks

    Hard Disks (high capacity, low cost per bit)

    Floppy Disks (low capacity, slow, cheap)

  - Optical Disks

    CD-ROM = Compact Disc, read-only memory

  (Read-only/write once, holds a lot of data, cheap reproduction)

- Serial Devices

  - Magnetic tapes (very fast sequential access)

Organization of Disks

From now on we will use "disks" to refer to hard disks.
How disk drivers work

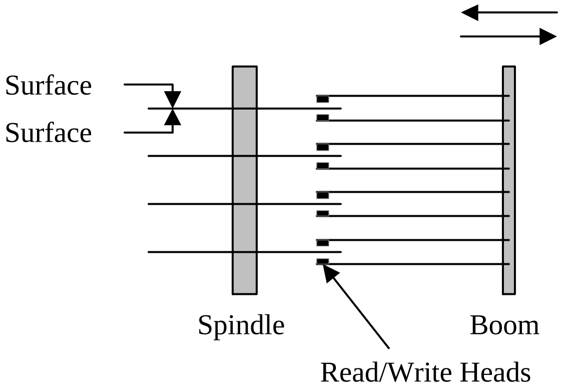
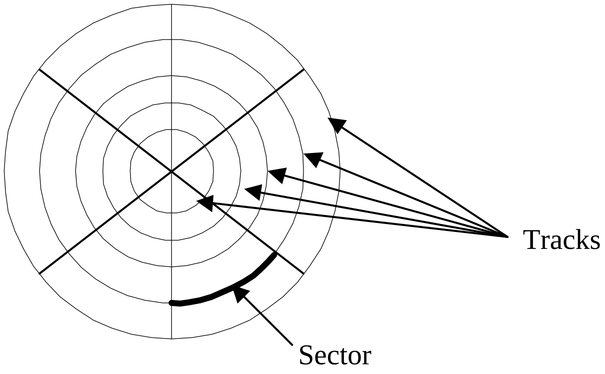


Figure 1: Disk drive with 4 platters and 8 surfaces

Looking at a surface:



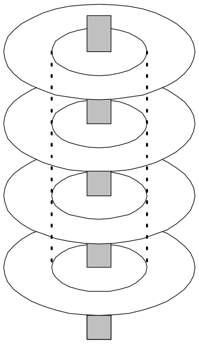
- Disk contains concentric **tracks**
- Tracks are divided into **sectors**
- A **sector** is the smallest addressable unit in a disk

When a program reads a byte from the disk, the operating system locates the surface, track and sector containing that byte, and reads the entire sector into a special area in main memory called buffer.

The bottleneck of a disk access is moving the read/write arm. So, it makes sense to store a file in tracks that are below/above each other in different surfaces, rather than in several tracks in the same surface.

Cylinder = the set of tracks on a disk that are directly above/below each other.

A cylinder



All the information on a cylinder can be accessed without **moving the read/write arm (Seeking)**.

number of cylinders = number of tracks in a surface
track capacity = number of sector per track x bytes per sector
cylinder capacity = number of surfaces x track capacity
drive capacity = number of cylinders x cylinder capacity

Solve the following problem:

File characteristics:

- Fixed-length records
- Number of records = 50,000 records
- Size of a record = 256 bytes

Disk characteristics:

- Number of bytes per sector = 512
- Number of sectors per track = 63
- Number of tracks per cylinder = 16
- Number of cylinders = 4092

Q: How many cylinders are needed?

A:
2 records per sector
2×63 records per track
$16 \times 126 = 2{,}016$ records per cylinder
number of cylinders $= \frac{50{,}000}{2{,}016} \sim 24.8$ cylinders

Note: A disk might not have this many physically contiguous cylinders available. This file may be spread over hundreds of cylinders.

Organizing Tracks by Sector

The Physical Placement of Sectors

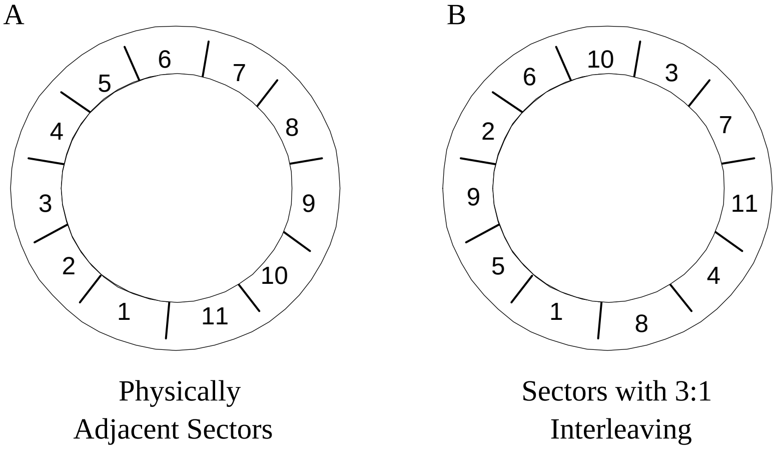


Figure 2: Interleaving

- Files are often stored in adjacent sectors, but "logically adjacent" does not necessarily mean "physically adjacent"
- **For some disks** : when it reads a sector, it takes some time to get ready to read the next sector. **Solution: interleaving**

Example: Suppose we need to read consecutively the sectors of a track in order: sector 1, sector 2, sector 3, ..., sector 11. Suppose the disk cannot read consecutive sectors.

- Without interleaving (Figure -A):
  How many revolutions to read the disk?
- With 3:1 interleaving (Figure -B):
  How many revolutions to read the disk?

(Now most disk controllers are fast and need no interleaving)

Clusters, Extents and Fragmentation

The **file manager** is the part of the operating system responsible for managing files. The file manager maps the **logical parts** of the file into their **physical location**.

A **cluster** is a fixed number of contiguous sectors (if there is interleaving these sectors are not physically contiguous).
The **file manager** allocates an integer number of **clusters** to a file.

Ex: Sector size: 512 bytes, Cluster size: 2 sectors

If a file contains 10 bytes, a cluster is allocated (1024 bytes). There may be unused space in the last cluster of a file. This unused space contributes to internal fragmentation.

Clusters are good since they improve sequential access: reading bytes sequentially from a cluster can be done in one revolution, seeking only once.
The file manager maintains a file allocation table (FAT) containing for each cluster in the file its location in disk.

An **extent** is a group of contiguous clusters. If a file is stored in a single extent then seeking (movement of read/write head) is done only once. If there is not enough contiguous clusters to hold a file, the file is divided into 2 or more extents.

Fragmentation

1) Due to records not fitting exactly in a sector

Ex: Record size = 200 bytes, Sector size = 512 bytes
To avoid that a record span 2 sectors we can only store 2 records in this sector (112 bytes go unused per section). This extra unused space contributes to fragmentation.
The alternative is to let a record span two sectors, but then two sectors must be read when we need to access this record.

2) Due to the use of clusters

If the file size is not a multiple of the cluster size, then the last cluster will be partially used.

Choice of cluster size: some operating systems allow the system administrator to choose cluster size.

When to use **large cluster size**?
When disk contain large files likely to be processed sequentially.
Ex: Updates in a master file of bank accounts (in batch mode)

What about **small cluster size**?
When disks contain small files and/or files likely to be accessed randomly
Ex: Online updates for airline reservation.

Organizing Tracks by Blocks

- Disk tracks may be divided into user-defined **blocks** rather than into sectors.
  (Note: Here, “block” is not used as a synonym of sector or group of sectors)
- The amount transferred in a single I/0 operation can vary depending on the needs of the software designer (not hardware)
- A block is usually organized to contain an integral number of logical records.

Blocking Factor = number of records stored in each block in a file.

No internal fragmentation, no record spanning two blocks.

A block typically contains subblocks:

- Count subblock: contains the number of bytes in a block.
- Key subblock (optional): contains the key for the last record in the data subblock (the disk controller can search for key without loading it in main memory)
- Data subblock: contains the records in this block.

Nondata Overhead

Amount of space used for extra stuff other than data.

Sector-Addressable Disks = At the beginning of each sector some info is stored, such as sector address, track address, condition (if sector is defective); there is some gap between sectors.

Block-Organized Disks subblocks and interblock gaps is part of the extra stuff; more nondata overhead than with sector-addressing.

Example:
Disk characteristics:
- Block-addressable disk drive
- Size of track = 20,000 bytes
- Nondata overhead per block = 300 bytes

File characteristics:
- Record size = 100 bytes

Q: How many records can be stored per track for the following blocking factors?

(1) Blocking factor = 10

(2) Blocking factor = 60

Solution for example:

Disk characteristics:
- Block-addressable disk drive
- Size of track = 20,000 bytes
- Nondata overhead per block = 300 bytes

File characteristics:
- Record size = 100 bytes

(1) Blocking factor = 10

Size of data subblock = 1,000
Nondata overhead = 300
# of blocks that can fit in a track = $\lfloor \frac{20{,}000}{1{,}300} \rfloor = \lfloor 15.38 \rfloor = 15$ blocks
# of records per track = 150 records

(2) Blocking factor = 60

Size of data subblock = 6,000
Nondata overhead = 300
# of blocks that can fit in a track = $\lfloor \frac{20{,}000}{6{,}300} \rfloor = 3$ blocks
# of records per track = 180 records

The Cost of a Disk Access

The time to access a sector in a track on a surface is divided into:

Time Component	**Action**
Seek time	Time to move the read/write arm to the correct cylinder
Rotational delay (or latency)	Time it takes for the disk to rotate so that the desired sector is under the read/write head
Transfer time	Once the read/write head is positioned over the data, this is the time it takes for transferring data

Example: Disk Characteristics:

- Average seek time = 8 msec
- Average rotational delay = 3 msec
- Maximum rotational delay = 6 msecs
- Spindle speed = 10,000 rpm
- Sectors per track = 170 sectors
- Sector size = 512 bytes

What is the average time to read one sector ?

Transfer time = revolution time/# sectors per track = (1/10,000)min/170 = (1/10,000 x 60)/170 secs = 6/170,000 secs = 6/170 msecs $\approx$ 0.035 msecs

Average total time = average seek + average rotational delay + transfer time = 8 + 3 + 0.035 = 11.035 msecs.

Comparing sequential access to random access

Same disk as before:
- Average seek time = 8 msec
- Average rotational delay = 3 msec
- Maximum rotational delay = 6 msecs
- Spindle speed = 10,000 rpm
- Sectors per track = 170 sectors
- Sector size = 512 bytes

File characteristics:
- Number of records = 34,000
- Record size = 256 bytes
- File occupies 100 tracks dispersed randomly

a) Reading File Sequentially
- Average seek time = 8 msecs
- Average rotational delay = 3 msecs
- Average transfer time for 1 track = 60/10,000 = 6 msecs
- Average total time per track = 8 + 3 + 6 = 17 msecs
- Total time for file = 17 msecs x 100 = 1.7 secs

b) Reading a file accessing randomly each record
Average total time to read all records = Number of records x average time to read a sector = 34,000 x 11.035 msecs ≈ 371.1 secs

Note: typo in page 63 of the book in a similar calculation.

Disk as a bottleneck

Processes are often **disk-bound**, i.e. network and CPU have to wait a long time for the disk to transmit data.

Various techniques to solve this problem

1. **Multiprocessing** (CPU works on other jobs while waiting for the disk)
2. **Disk Striping**
   Putting different blocks of the file in different drives.
   Independent processes accessing the same file may not interfere with each other (parallelism).
3. **RAID** (Redundant array of independent disks)
   Ex: in an eight-drive RAID the controller breaks each block into 8 pieces and place one in each disk drive (at the same position in each drive).
4. **RAM Disk** (memory disk)
   Piece of main memory used to simulate a disk (difference: speed and volatility). Used to simulate floppy disks.
5. **Disk Cache**
   Large block of memory configured to contain pages of data from a disk (typical size = 256 KB). When data is requested from disk, check cache first. If data is not there go to the disk and replace some page already in cache with page from disk containing the data.

LECTURE 5: SECONDARY STORAGE DEVICES - MAGNETIC TAPES AND CD-ROM

Contents of today's lecture:

- Magnetic Tapes
  - Characteristics of magnetic tapes
  - Data organization on 9-track tapes
  - Estimating tape length requirements
  - Estimating data transmission times
  - Disk versus tape
- CD-ROM
  - Physical Organization of CD-ROM
  - CD-ROM Strengths and Weaknesses

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Section 3.2, 3.5 and 3.6.

Characteristics of Magnetic Tapes

- No direct access, but very fast sequential access.
- Resistant to different environmental conditions.
- Easy to transport, store, cheaper than disk.
- Before, it was widely used to store application data; nowadays, it's mostly used for backups or archives (tertiary storage).

Data Organization on Nine-Track Tapes

In a tape, the **logical position** of a byte within a file is the same as its **physical position** in the file (sequential access).

Nine-track tape:

```
-----------------------------------------------------
______|0     0|__________|0|__________|0     0|______
______|0     0|__________|1|__________|0     0|______
______|0     0|__________|1|__________|0     0|______
______|0     0|__________|0|__________|0     0|______
______|0     0|__________|1|__________|0     0|______
______|0     0|__________|0|__________|0     0|______
______|0     0|__________|0|__________|0     0|______
______|0     0|__________|1|__________|0     0|______
______|0     0|__________|1|__________|0     0|______
-----------------------------------------------------
      |<-Gap->|  <-- Data Block -->   |<-Gap->|
```

- **Data blocks** are separated by interblock GAPS.
- 9 parallel tracks (each is a sequence of bits)
- A **frame** is a 1-bit slice of the tape corresponding to 9 bits (one in each track) which correspond to 1 byte plus a **parity bit**.

In the example above, the byte stored in the frame that is shown is: `01101001`. The parity bit is `1`, since we are using **odd parity**, i.e., the total number of bits is odd.

Complete the parity bit in the examples below:

```
11111111

00000000

00100000
```

Since `000000000` cannot correspond to a valid byte, this is used to mark the **interblock gap**.

So, if we say that this tape has 6,250 **bits per inch** (bpi) per track, indeed it stores 6,250 **bytes per inch** when we take into account the 9 tracks.

Estimating Tape Length Requirements

Performance of tape drives can be measured in terms of 3 quantities:
- Tape density = 6250 bpi (bits per inch per track)
- Tape speed = 200 inches per second (ips)
- Size of interblock gap = 0.3 inch

File characteristics:
- Number of records = 1,000,000
- Size of record = 100 bytes

How much tape is needed?

It depends on the blocking factor (how many records per data block). Let us compute the space requirement in two cases:
A) Blocking factor = 1
B) Blocking factor = 50

Space requirement (s)
b = length of data block (in inches)
g = length of interblock gap (in inches)
n = number of data blocks

s = n x (b + g)

A) Blocking factor = 1

b = block size/tape density = 100 bytes/6250 bpi= 0.016 inch
n = 1,000,000 (recall blocking factor is 1)
s = 1,000,000 x (0.016 + 0.3) inch = 316,000 inches $\sim$ 26,333 feet
(Absurd to have the length of the data block smaller than the interblock gap!)

B) Blocking factor = 50

b = 50 x 100 bytes/6,250 bpi = 0.8 inch
n = 1,000,000 records/50 records per block = 20,000 blocks
s = 20,000 x (0.8 + 0.3) inch = 22,000 inches $\approx$ 1,833 feet

An enormous saving by just choosing a higher blocking factor.

Effective Recording Density (ERD)

ERD = number of bytes per block / number of inches to store a block

In previous example :

A) Blocking factor =1: E.R.D. = 100/0.316 $\sim$ 316.4 bpi
B) Blocking factor =50: E.R.D. = 5,000/1.1 $\sim$ 4,545.4 bpi

The **Nominal Density** was 6,250 bpi!

Estimating Data Transmission Times

Nominal Rate = tape density (bpi) x tape speed (ips)

In a 6,250 - bpi , 200 - ips tape :
Nominal Rate = 6,250 bytes/inch x 200 inches/second =
= 1,250,000 bytes/sec ∼ 1,250 KB/sec

Effective Transmission Rate = E.R.D. x tape speed

In the previous example:
A) E.T.R. = 316.4 bytes/inch x 200 inches/sec = 63,280 bytes/sec ∼ 63.3 KB/sec
B) E.T.R. = 4,545.4 bytes/inch x 200 inches/sec = 909,080 bytes/sec ∼ 909 KB/sec
Note : There is a tradeoff between **increasing** blocking factor for increasing speed & space utilization and **decreasing** it for reducing the size of the I/O buffer.

Disk versus Tape

In the past : Disks and Tapes were used for secondary storage: disks preferred for random access and tapes for sequential access.
Now :
Disks have taken over most of secondary storage (lower cost of disk and lower cost of RAM which allows large I/O buffer). Tapes are mostly used for **tertiary storage**.

Physical Organization of CD-ROM

Compact Disc - read only memory (write once)

- Data is encoded and read optically with a laser
- Can store around 600 MB data

Digital data is represented as a series of **Pits** and **Lands**.

Pit = a little depression, forming a lower level in the track
Land = the flat part between pits, or the upper levels in the track

Reading a CD is done by shining a laser at the disc and detecting changing reflections patterns.

1 = change in height (land to pit or pit to land)
0 = a "fixed" amount of time between 1's

```
       LAND       PIT       LAND       PIT   LAND
... ------------+     +-------------+       +---- ...
                |_____|             |_______|

... 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 1 0 0 0 1 0 0 ...
```

Changes in height in the track are detected as changes of intensity of the reflected light.
Note: We cannot have two 1's in a row!

Indeed, because of other limitations there must be at least two and at most ten 0's between two 1's.

Therefore, each of the 256 bytes must be encoded into a sequence of bits that has every pair of 1's separated by at least two zeros. There are exactly 267 binary words of length 14 that satisfy this property; 256 of them were chosen to represent every possible byte in the so-called eight to fourteen modulation. We could not encode bytes using 13 bits since there are only 188 words of length 13 having the desired property.

Eight to fourteen modulation (EFM) encoding table:

Decimal	Original	Translated
Value	Bits	Bits
0	00000000	01001000100000
1	00000001	10000100000000
2	00000010	10010000100000
3	00000011	10001000100000
4	00000100	01000100000000
5	00000101	00000100010000
6	00000110	00010000100000
7	00000111	00100100000000
8	00001000	01001001000000
...	...	...

Note that: Since 0's are represented by the **length of time** between transitions, we must travel at **constant linear velocity** on the tracks.

Comparing CD-ROM with magnetic disks

CD-ROM	**Magnetic Disks**
CLV = Constant Linear Velocity	CAV = Constant Angular Velocity
Sectors organized along a spiral	Sectors organized in concentric track
Sectors have same linear length (data packed at its maximum density permitted)	Sectors have same angular length (data written less densely in the outer tracks)
Advantage: takes advantage of all storage space available	Advantage: operates on constant speed, timing marks to delimit tracks
Disadvantage: has to change rotational speed when seeking (slower towards the outside)	Disadvantage: it doesn't use up all storage available

CD-ROM addressing and poor Seek performance

Addressing

1 second of play time is divided up into 75 **sectors**.
Each sector holds 2KB.

60 Min CD :
60 min x 60 sec/min x 75 sectors/sec = 270,000 sectors = 540,000 KB ~ 540 MB

A **sector** is addressed by :
Minute : Second : Sector
16:22:34
16 min, 22 sec, 34th sector

Difficulty in Seeking

- To read address of a sector it must be at the correct speed
- But knowing the correct speed depends on the ability to read the address info!

The **drive control mechanism** solves this problem by trial-and-error.
This slows down the performance!

CD-ROM Strength and Weaknesses

- Seek performance (∼ 500 msecs) - Slow

  Our old analogy :
  20 secs (RAM)
  58 days (Magnetic Disks)
  2.5 years (CD-ROM)

- Data transfer rate - 150 KB/sec - Slow (while ∼ 3,000 KB/sec for magnetic disks), but 5 times faster than floppy disks.

- Storage capacity is ∼ 600 MB; good for storing texts.

- Read-only access (publishing medium). File structure designer can take advantage of that.

Things changed nowadays :

- Most drives use CAV or combination of CAV and CLV
- Other types of compact discs :
  - CD-R = compact disc-recordable
  - CD-RW = compact disc-rewritable

  They use different technologies which simulates the effect of Pits and Lands.

LECTURE 6: A JOURNEY OF A BYTE AND BUFFERING

Contents of today's lecture:

- A Journey of a Byte
- Buffer Management

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Section 3.8 and 3.9.

Complementary reading: Section 3.10 "I/O in Unix", if interested.

A Journey of a Byte

Suppose in our program we wrote :

```
...
outfile << c;
...
```

This causes a call to the file manager, the part of the operating system responsible for input/output operations.

The O/S (File Manager) makes sure the byte is written to the disk.

Pieces of software/hardware involved in I/O operations :

- **Application program**
  - requests the I/O operation (`outfile << c;`)
- **Operating system/file manager**
  - keeps tables for all opened files (types of accesses allowed, FAT with each file's corresponding clusters, etc)
  - brings appropriate sector to buffer
  - writes **byte** to buffer
  - gives instruction to I/O processor to write data from this buffer into correct place in disk.

  Note: the operating system is a program running in CPU and working on RAM (it copied the content of variable `c` into the appropriate buffer). The **buffer** is an exact image of a cluster in disk.
- **I/O Processor** (a separate chip in the computer; it runs independently of CPU so that it frees up CPU to other tasks - I/O and internal computing can overlap)
  - Finds a time when drive is available to receive data, and puts data in proper format for the disk.
  - Sends data to the disk controller
- **Disk Controller** (a separate chip on the disk circuit board)
  - Controller instructs the drive to move the read/write head to the proper track, waits for proper sector to come under the read/write head, then sends the **byte** to be deposited on the surface of the disk.

Refer to Figure 3.21 at page 90 of the textbook.

Buffer Management

Buffering means working with large chunks of data in main memory so that the number of accesses to secondary storage is reduced.

Today we will discuss the **System I/O Buffers**.

Note that the application program may have its own “buffer” - i.e. a place in memory (variable, object) that accumulates large chunks of data to be later written to disk as a chunk.

Recommended Reading :
Chapter 4.2 - using classes to manipulate buffers. This has nothing to do with the system I/O buffer which is beyond the control of the program and is manipulated by the operating system.

System I/O Buffers

Buffer Bottlenecks What if the O/S used only one I/O buffer ?

Consider a piece of program that reads from a file and writes into another, character by character:

```
      while(1) {
(1)      infile >> ch;
(2)      if (infile.fail()) break;
(3)      outfile << ch;
      }
```

Suppose that the next character to be read from **`infile`** is physically stored in sector `X` of the disk. Suppose that the place to write the next character to **`outfile`** is sector `Y`.

With a single buffer :

- When line (1) is executed, sector `X` is brought to the buffer and `ch` receives the correct character.
- When line (3) is executed, sector `Y` must be brought to the buffer. Sector `Y` is brought and `ch` is deposited to the right position.
- Now line (1) is executed again. Suppose we did not reach the end of sector `X` yet. Then, sector `X` must be brought again to the buffer, so the current content of the buffer must be written to sector `Y` before this is done.

And so on.

This could be solved if there were more buffers available!

Most operating systems have an input buffer and an output buffer.
One buffer could be used for `infile` and one for `outfile`.
A new trip to get a sector of `infile` would only be done after all the bytes in the previous sector had been read.

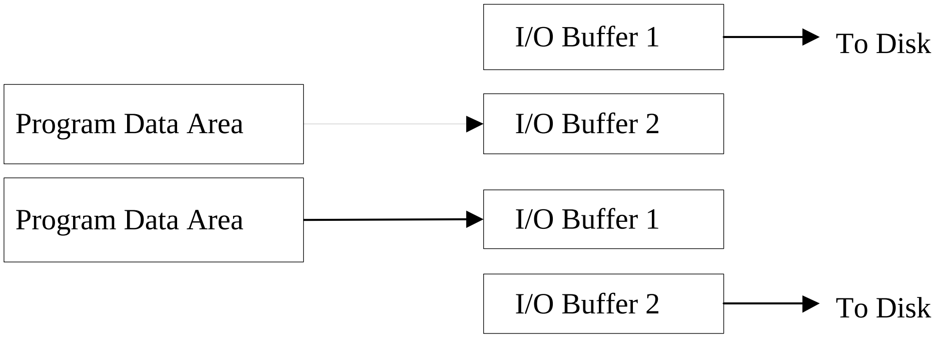
Similarly, the buffer for `output` would be written to the file only when full (or when file was closed).

Question : If the sector size is 512 bytes. How many extra trips to the disk we have to do if we have only 1 buffer in comparison to two or more, **in our previous program** ?

Buffering Strategies

1. Multiple Buffering

Double buffering: Two buffers + I/O-CPU overlapping



Several buffers may be employed in this way (multiple buffering).

Some operating systems use a buffering scheme called **buffer pooling** :

- There is a pool of buffers.
- When a request for a sector is received by the O/S, it first looks to see if that sector is in some of the buffers.
- If not there, then it brings the sector to some free buffer. If no free buffer exists then it must choose an occupied buffer, write its current contents to the disk, and then bring the requested sector to this buffer.

Various schemes may be used to decide which buffer to choose from the buffer pool. One effective strategy is the **Least Recently Used (LRU)** Strategy: when there are no free buffers, the least recently used buffer is chosen.

2. Move Mode and Locate Mode

Move Mode

Situation in which data must be always moved from system buffer to program buffer (and vice-versa).

Locate Mode

This refers to one of the following two techniques, in order to avoid unnecessary moves.
The file manager uses system buffers to perform all I/O, but provides its location to the program, using a pointer variable.
The file manager performs I/O directly between the disk and the program's data area.

3. Scatter/Gather I/O

We may want to have data separated into more than one buffer (for example: a block consisting of header followed by data).

Scatter Input: a single read call identifies a collection of buffers into which data should be scattered.

Similarly, we may want to have several buffers gathered for output.

Gather Output: a single write call gathers data from several buffers and writes it to output.

LECTURE 7: DATA COMPRESSION I

Contents of today's lecture:

- Introduction to Data Compression
- Techniques for Data Compression
  - Compact Notation
  - Run-length Encoding
  - Variable-length codes: Huffman Code

References:
FOLK, ZOELLICK AND RICCARDI, File Structures, 1998. Section 6.2 (Data Compression).
CORMEN, LEISERSON, RIVEST AND STEIN, Introduction to Algorithms, 2001, 2nd ed. Section 16.3 (Huffman codes).

Data Compression = Encoding the information in a file in such a way that it takes less space.

Using Compact Notation

Ex: File with fields: lastname, province, postal code, etc. Province field uses 2 bytes (e.g. ‘ON’, ‘BC’) but there are only 13 provinces and territories which could be encoded by using only 4 bits (compact notation).

16 bits are encoded by 4 bits (12 bits were redundant, i.e. added no extra information)

Disadvantages:

- The field “province” becomes unreadable by humans.
- Time is spent encoding (‘ON’ → 0001) and decoding (0001 → ‘ON’).
- It increases the complexity of software.

Run-length Encoding

Good for files in which sequences of the same byte may be frequent.

Example: Figure 6.1 in page 205 of the textbook: image of the sky.

- A pixel is represented by 8 bits.
- Background is represented by the pixel value 0.

The idea is to avoid repeating, say, 200 bytes equal to 0 and represent it by (0, 200).

If the same value occurs more than once in succession, substitute by 3 bytes:

- a special character - run length code indicator (use 1111 1111 or FF in hexadecimal notation)
- the pixel value that is repeated (FF is not a valid pixel anymore)
- the number of times the value is repeated (up to 256 times)

Encode the following sequence of Hexadecimal bytes:

```
22  23  24  24  24  24  24  24  24  25
26  26  26  26  26  26  25  24
```

Run-length encoding:

```
22  23  FF  24  07  25  FF  26  06  25  24
```

18 bytes reduced to 11.

Variable-Length Codes and Huffman Code

Example of a variable-length code:

Morse Code (two symbols associated to each letter)

A	._
B	_...
...	
E	.
F	.._.
...	
T	_
U	.._
...	

Since E and T are the most frequent letters, they are associated to the shortest codes (. and - respectively)

Huffman Code

Huffman Code is a variable length code, but unlike Morse Code the encoding depends on the frequency of letters occurring in the data set.

Example of Huffman Code:
Suppose the file content is:

I		A	M		S	A	M	M	Y

Total: 10 characters

Letter	A	I	M	S	Y	/b
Frequency	2	1	3	1	1	2
Code	00	1010	11	1011	100	01

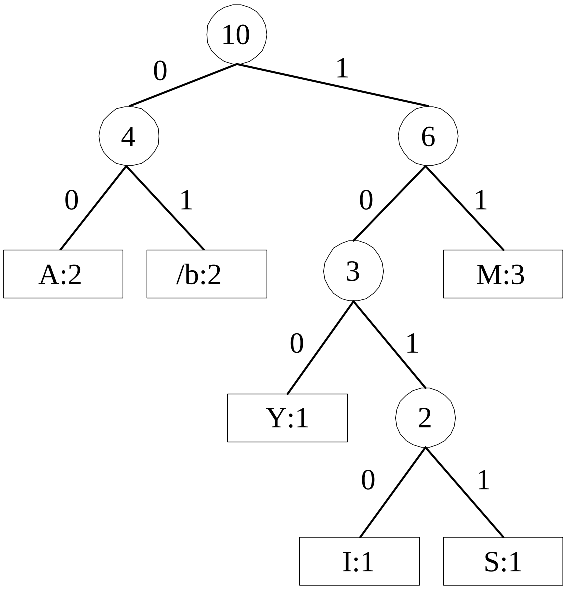
Huffman Code is a prefix code: no codeword is a prefix of any other.
(we are representing the space as “/b”)

Encoded message:

```
1010010011011011001111100
```

25 bits rather than 80 bits (10 bytes)!

Huffman Tree (for easy decoding)



Consider the encoded message:

```
101001001101...
```

- Interpret the 0's as "go left" and the 1's as "go right".
- A **codeword** for a character corresponds to the path from the root of the Huffman tree to the leaf containing the character.

Following the labeled edges in the Huffman tree we decode the above message.

```
1010    leads us to I
01      leads us to /b
00      leads us to A
11      leads us to M
01      leads us to /b
etc.
```

Properties of Huffman Tree

- Every internal node has 2 children;
- Smaller frequencies are further away from the root;
- The 2 smallest frequencies are siblings;
- The number of bits required to encode the file is minimized:

$$B(T) = \sum_{(c \in C)} f(c).d_T(c),$$

where:
$B(T)$ = number of bits needed to encode the file using tree T,
$f(c)$ = frequency of character c,
$d_T(c)$ = length of the codeword for character c.

In our example:

$$B(T) = 2 \times 2 + 1 \times 4 + 3 \times 2 + 1 \times 4 + 1 \times 3 + 2 \times 2 = 25$$

What is the average number of bits per encoded letter ?

Average number of bits per letter =
= B(T)/total number of characters = 25/10 = 2.5

The way Huffman Tree is constructed guarantees that B(T) is as small as possible!

How is the Huffman Tree constructed ?

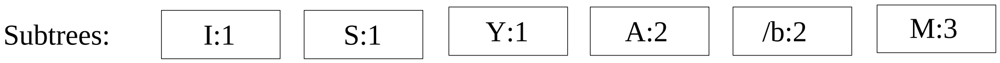
The weight of a node is the total frequency of characters under the subtree rooted at the node.

Originally, form subtrees which represent each character with their frequencies as weights.

The algorithm employs a **Greedy Method** that always merges the subtrees of smallest weights forming a new subtree whose root has the sum of the weight of its children.

The algorithm in action

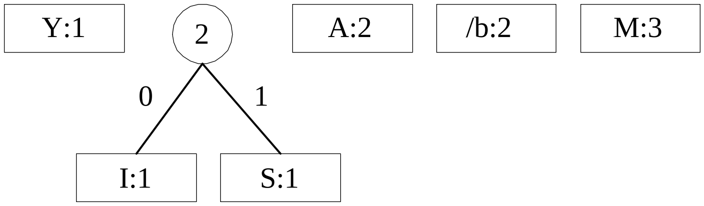
Using the letters and frequencies from the previous example:



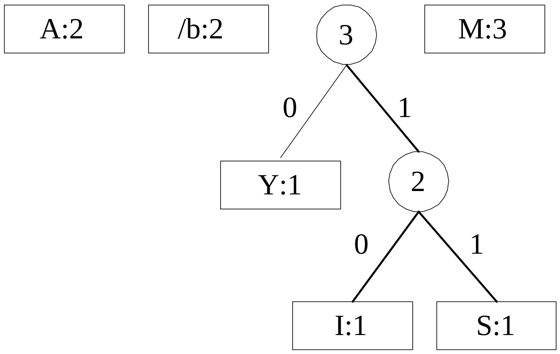
Subtrees: I:1 S:1 Y:1 A:2 /b:2 M:3

Merge the two subtrees of smallest weight (break ties arbitrarily) ...

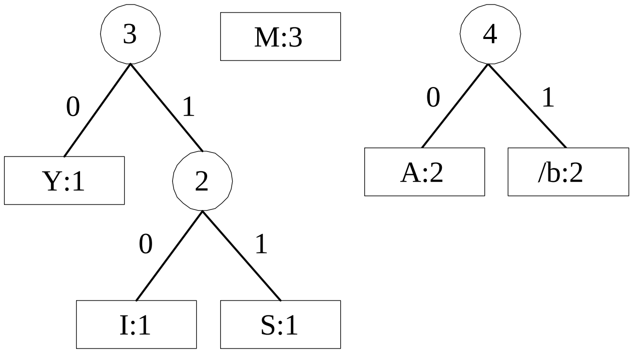
Subtrees:



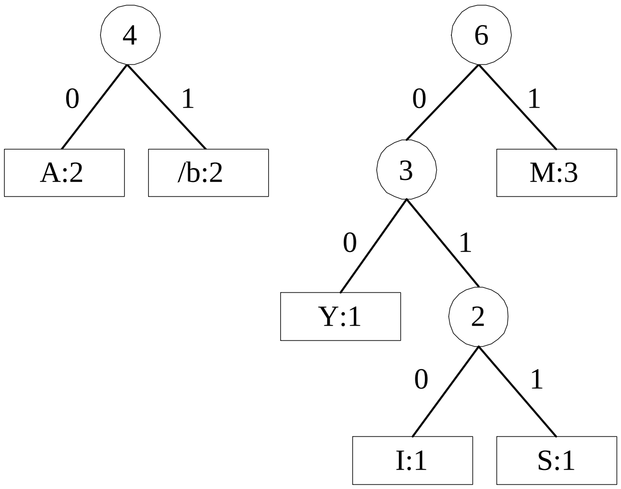
Subtrees:



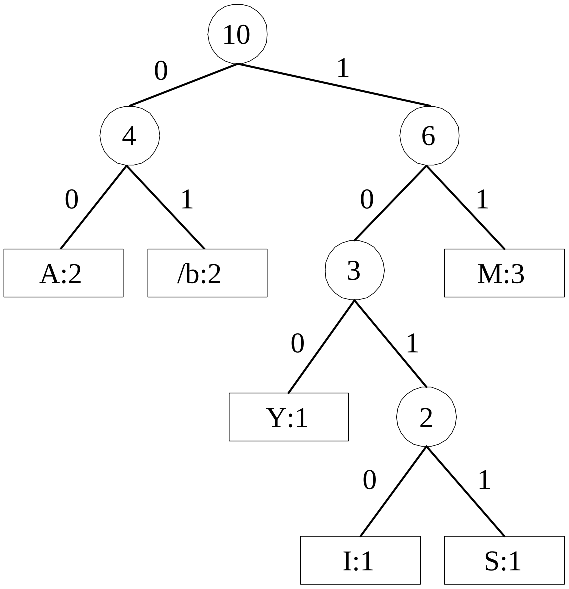
Subtrees:



Subtrees:



Final Tree:



Pseudo-Code for Huffman Algorithm:

A **priority queue** `Q` is used to identify the smallest-weight subtrees to merge. A priority queue provides the following operations:

- `Q.insert(x)`: insert `x` to `Q`
- `Q.minimum()`: returns element of smallest key
- `Q.extract-min()`: removes and returns the element with smallest key

Possible implementations of a priority queue:
Linked lists: Each of the three operations can be done in O(n)
Heaps: Each of the three operations can be done in O(log n)

Pseudo-Code: `Huffman`

Input: characters and their frequencies
`(c1, f[c1]), (c2, f[c2]), ..., (cn, f[cn])`
Output: returns the Huffman Tree

```
Make priority queue Q using c1, c2, ..., cn;
 for i = 1 to n-1 do {
     z = allocate new node;
     l = Q.extract-min();
     r = Q.extract-min();
     z.left = l;
     z.right = r;
     f[z] = f[r] + f[l];
     Q.insert(z);
 }
 return Q.extract-min();
```

What is the running time of this algorithm if the priority queue is implemented as a ...

1. **Linked List ?**
   - `Make priority queue` takes $O(n)$.
   - `extract-min` and `insert` takes $O(n)$.
   - Loop iterates $n-1$ times.

   Total time: $O(n^2)$

2. **Heap (Array Heap) ?**
   - `Make priority queue` takes $O(n \cdot \log n)$ or $O(n)$.
   - `extract-min` and `insert` takes $O(\log n)$.
   - Loop iterates $n-1$ times.

   Total time: $O(n \cdot \log n)$.

- **Pack** and **unpack** commands in Unix use Huffman Codes byte-by-byte.
- They achieve 25 - 40% reduction on text files, but is not so good for binary files that have more uniform distribution of values.

LECTURE 8: DATA COMPRESSION II

Contents of today's lecture:

- Techniques for Data Compression
  - Lempel-Ziv codes.

Reference: This notes.

Lempel-Ziv Codes

There are several variations of Lempel-Ziv Codes. We will look at LZ78.

Ex: Commands `zip` and `unzip` and Unix `compress` and `uncompress` use Lempel-Ziv codes.

Let us look at an example for an alphabet having only two letters:

`aaababbbaaabaaaaaaabaabb`

Rule : Separate this stream of characters into pieces of text, so that each piece is the shortest string of characters that we have not seen yet.

`a|aa|b|ab|bb|aaa|ba|aaaa|aab|aabb`

1. We see `"a"`.
2. `"a"` has been seen, we now see `"aa"`.
3. We see `"b"`.
4. `"a"` has been seen, we now see `"ab"`.
5. `"b"` has been seen, we now see `"bb"`.
6. `"aa"` has been seen, we now see `"aaa"`.
7. `"b"` has been seen, we now see `"ba"`.
8. `"aaa"` has been seen, we now see `"aaaa"`.
9. `"aa"` has been seen, we now see `"aab"`.
10. `"aab"` has been seen, we now see `"aabb"`.

Note that this is a dynamic method!

Index the pieces from 1 to n. In the previous example:

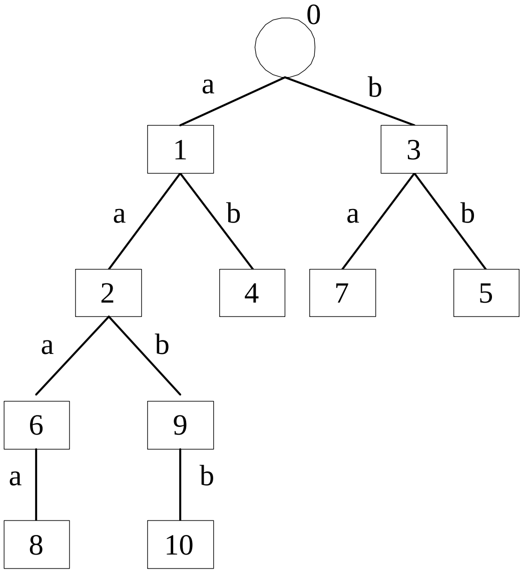
```
Index : 0 1 2  3 4  5  6   7  8    9   10
        0|a|aa|b|ab|bb|aaa|ba|aaaa|aab|aabb
```

0 = Null string

Encoding :

```
Index : 1  2  3  4  5  6  7  8  9  10
        0a|1a|0b|1b|3b|2a|3a|6a|2b|9b
```

Since each piece is the concatenation of a piece already seen with a new character, the message can be encoded by a previous index plus a new character.
Indeed a digital tree can be built when encoding:



When a node is inserted the code for the current piece becomes the parent node combined with the new character.
Note that this tree is not binary in general. Here, it is binary because the alphabet has only 2 letters.

Practice Exercises

Encode (using Lempel-Ziv) the file containing the following characters, drawing the corresponding digital tree:

```
"aaabbcbcdddeab"
```

```
"I AM SAM. SAM I AM."
```

Bit Representation of Coded Information

How many bits are necessary to represent each integer with index n ? The integer is at most $n-1$, so the answer is: at most the number of bits to represent the number $n-1$.

```
1   2   3   4   5   6   7   8   9   10
0a|1a|0b|1b|3b|2a|3a|6a|2b|9b
```

Index 1: no bit (always start with zero)
Index 2: at most 1, since previous index can be only 0 or 1.
Index 3: at most 2, since previous index is between 0-2.
Index 4: at most 2, since previous index is between 0-3.
Index 5-8: at most 3, since previous index is between 0-7
Index 9-16: at most 4, since previous index is between 0-15

Each letter is represented by 8 bits. Each index is represented using the largest number of bits possibly required for that position. For the previous example, this representation would be as follows:

```
<a>1<a>00<b>01<b>011<b>010<a>011<a>110<a>0010<b>1001<b>
```

Note that `<a>` and `<b>` above should be replaced by the ASCII code for `a` and `b`, which uses 8 bits. We didn't replace them for clarity and conciseness.

Total number of bits in the encoded example :
$(10 \times 8) + (0 + 1 + 2 \times 2 + 4 \times 3 + 2 \times 4) = 105$ bits

The original message was represented using $24 \times 8 = 192$ bits.

Decompressing

```
1  2  3  4  5  6  7  8  9  10
0a|1a|0b|1b|3b|2a|3a|6a|2b|9b
```

	previous pointer	added character
0	-	-
1	0	a
2	1	a
3	0	b
4	1	b
5	3	b
6	2	a
7	3	a
8	6	a
9	2	b
10	9	b

As the table is constructed line by line, we are able to decode the message by following the pointers to previous indexes which are given by the table. Try it, and you will get:

```
a aa b ab bb aaa aaa ba aaaa aab aabb
```

Decode the following Lempel-Ziv encoded file:

```
|0M|0A|0K|0E|0 |0L|2K|4 |0F|7E|
```

decoded message:

number of bits in original message:

number of bits in encoded message:

Decode the following Lempel-Ziv encoded file:

```
|0T|0H|0A|1 |0S|3M|0 |0I|7A|0M|0,|1H|3T|4S|6 |8 |6!|
```

decoded message:

number of bits in original message:

number of bits in encoded message:

Irreversible Compression

All previous techniques : we preserve all information in the original data.

Irreversible compression is used when some information can be sacrificed.

Example :
Shrinking an image from 400-by-400 pixels to 100-by-100 pixels.
1 pixel in the new image for each 16 pixels in the original message.

It is less common than reversible compression.

Final Notes

In UNIX:

- **pack** and **unpack** use Huffman codes byte-by-byte.
25-40% for text files, much less for binary files (more uniform distribution)

- **compress** and **uncompress** use Lempel-Ziv.

LECTURE 9: RECLAIMING SPACE IN FILES

Contents of today's lecture:

Reclaiming space in files
Reference: FOLK, ZOELLICK AND RICCARDI, File Structures, 1998. Section 6.2

Motivation

Let us consider a file of records (fixed length or variable length).

We know how to create a file, add records to a file and modify the content of a record. These actions can be performed physically by using the various basic file operations we have seen (open a file for writing or appending, going to a position of a file using "seek" and writing the new information there).

What happens if records need to be deleted ?

There is no basic operation that allows us to "remove part of a file". Record deletion should be taken care by the program responsible for file organization.

Strategies for record deletion

How to delete records and reuse the unused space ?

1) Record Deletion and Storage Compaction

Deletion can be done by “marking” a record as deleted.

Ex.: - Place ‘*’ at the beginning of the record; or
- Have a special field that indicates one of two states: “deleted” or “not deleted”.

Note that the space for the record is not released, but the program that manipulates the file must include logic that checks if record is deleted or not.

After a lot of records have been deleted, a special program is used to squeeze the file - this is called **Storage Compaction**.

2) Deleting Fixed-Length Records and Reclaiming Space Dynamically

How to use the space of deleted records for storing records that are added later ?

Use an "AVAIL LIST", a linked list of available records.

- a header record (at the beginning of the file) stores the beginning of the AVAIL LIST (-1 can represent the null pointer).
- when a record is deleted, it is marked as deleted and inserted into the AVAIL LIST. The record space is in the same position as before, but it is logically placed into AVAIL LIST.

Ex.: After deletions the file may look like :

List head → 4

Edwards	Williams	*-1	Smith	*2	Sethi
0	1	2	3	4	5

If we add a record, it can go to the first available spot in the AVAIL LIST (RRN=4).

3) Deleting Variable Length Records

Use an AVAIL LIST as before, but take care of the variable-length difficulties.

The records in AVAIL LIST must store its size as a field.
RRN can not be used, but exact byte offset must be used.

List head → 33

<table>
<tr><th>Edwards|M|</th><th>Wu|F|</th><th>*-1|10|</th><th>Smith|M|</th><th>*15|30|</th></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>10 bytes</td><td>5 bytes</td><td>10 bytes</td><td>8 bytes</td><td>30 bytes</td></tr>
</table>

Addition of records must find a large enough record in AVAIL LIST.

Placement Strategies for New Records

There are several strategies for selecting a record from AVAIL LIST when adding a new record:

1. First-Fit Strategy

   - AVAIL LIST is not sorted by size.
   - First record large enough to hold new record is chosen.

   Example:
   AVAIL LIST: size=10, size=50, size=22, size=60
   record to be added: size=20

   Which record from AVAIL LIST is used for the new record ?

2. Best-Fit Strategy

   - AVAIL LIST is sorted by size.
   - Smallest record large enough to hold new record is chosen.

   Example:
   AVAIL LIST: size=10, size=22, size=50, size=60
   record to be added: size=20

   Which record from AVAIL LIST is used for the new record ?

3. Worst-Fit Strategy
   - AVAIL LIST is sorted by decreasing order of size.
   - Largest record is used for holding new record; unused space is placed again in AVAIL LIST.

   Example:
   AVAIL LIST: size=60, size=50, size=22, size=10
   record to be added: size=20
   Which record from AVAIL LIST is used for the new record ?

When choosing between strategies we must consider two types of fragmentation within a file:

Internal Fragmentation: wasted space within a record.
External Fragmentation: space is available at AVAIL LIST, but it is so small that cannot be reused.

For each of the following approaches, which type of fragmentation arises, and which placement strategy is more suitable?
When the added record is smaller than the item taken from AVAIL LIST:

- leave the space unused within the record
  type of fragmentation:
  suitable placement strategy:
- return the unused space as a new available record to AVAIL LIST
  type of fragmentation:
  suitable placement strategy:

Ways of combating external fragmentation:

- **Coalescing the holes** : if two records in AVAIL LIST are adjacent, combine them into a larger record.
- Minimize fragmentation by using one of the previously mentioned **placement strategies** (for example: worst-fit strategy is better than best-fit strategy in terms of external fragmentation when unused space is returned to Avail List).

LECTURE 10: BINARY SEARCHING, KEYSORTING AND INDEXING

Contents of today's lecture:

- Binary Searching (Chapter 6.3.1 - 6.3.3),
- Keysorting (Chapter 6.4)
- Introduction to Indexing (Chapter 7.1-7.3)

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 6.3.1-6.3.3,6.4,7.1-7.3

Binary Searching

Let us consider fixed-length records that must be searched by a **key value**.

If we knew the RRN of the record identified by this key value, we could jump directly to the record (using "seek").

In practice, we do not have this information and we must search for the record containing this key value.

If the file is not sorted by the key value we may have to look at every possible record before we find the desired record.

An alternative to this is to maintain the file **sorted by key value** and use **binary searching**.

A binary search[1] algorithm in C++ :

```
class FixedRecordFile{
   public:
     int NumRecs();
     int ReadByRRN(RecType & record, int RRN);
};
class KeyType {
   public:
     int operator==(KeyType &);
     int operator>(KeyType &);
};
class RecType {
   public:
     KeyType key();
};
int BinarySearch(FixedRecordFile & file, RecType & obj,
                 KeyType & key) {
   int low=0; int high=file.NumRecs() -1;
   while (low <= high) {
     int guess = (high + low)/2;
     file.ReadByRRN(obj,guess);
     if (obj.key() == key) return 1;
     if (obj.key() > key)  high = guess - 1;
     else low = guess + 1;
   }
   return 0;  //did not find key
}
```

[1] this algorithm corrects some mistakes found in the textbook.

Binary Search versus Sequential Search :

Binary Search : O(log$_2$n)
Sequential Search : O(n)

If file size is doubled, sequential search time is doubled, while binary search time increases by 1.

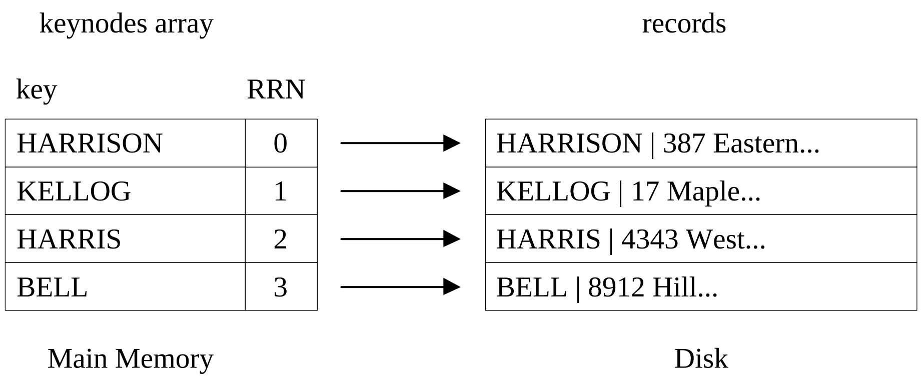
Keysorting

Suppose a file needs to be sorted, but it is too big to fit into main memory.

To sort the file, we only need the **keys**. Suppose that all the keys fit into main memory.

Idea:

- Bring the keys to main memory plus corresponding RRN
- Do internal sorting of keys
- Rewrite the file in sorted order



keynodes array

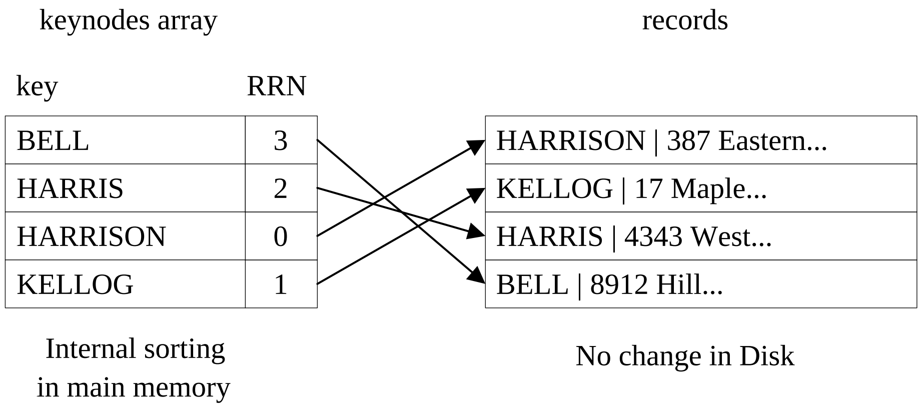
key	RRN
HARRISON	0
KELLOG	1
HARRIS	2
BELL	3

Main Memory

records

HARRISON | 387 Eastern...
KELLOG | 17 Maple...
HARRIS | 4343 West...
BELL | 8912 Hill...

Disk



keynodes array

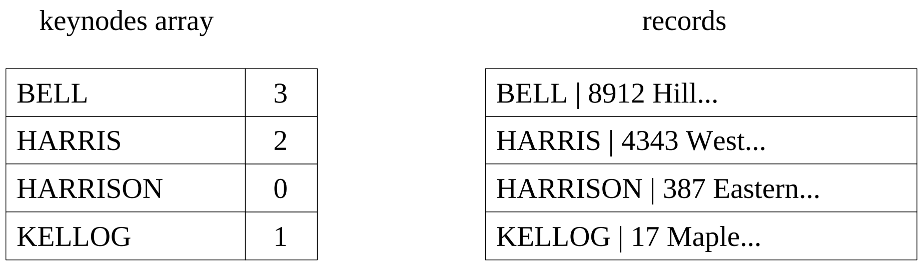
key	RRN
BELL	3
HARRIS	2
HARRISON	0
KELLOG	1

Internal sorting
in main memory

records

HARRISON | 387 Eastern...
KELLOG | 17 Maple...
HARRIS | 4343 West...
BELL | 8912 Hill...

No change in Disk



keynodes array

BELL	3
HARRIS	2
HARRISON	0
KELLOG	1

records

BELL | 8912 Hill...
HARRIS | 4343 West...
HARRISON | 387 Eastern...
KELLOG | 17 Maple...

create new sorted file to
replace previous

How much effort we must do (in terms of disk accesses) ?

- Read file sequentially once
- Go through each record in random order (seek)
- Write each record once (sequentially)

Why bother to write the file back?

Use keynode array to create an **index file** instead.

index file

BELL	3
HARRIS	2
HARRISON	0
KELLOG	1

records

HARRISON | 387 Eastern...
KELLOG | 17 Maple...
HARRIS | 4343 West...
BELL | 8912 Hill...

Leave file unchanged

This is called INDEXING !!

Pinned Records

Remember that in order to support deletions we used `AVAIL LIST`, a list of available records.
The `AVAIL LIST` contains info on the physical information of records. In such a file a record is said to be **pinned**.

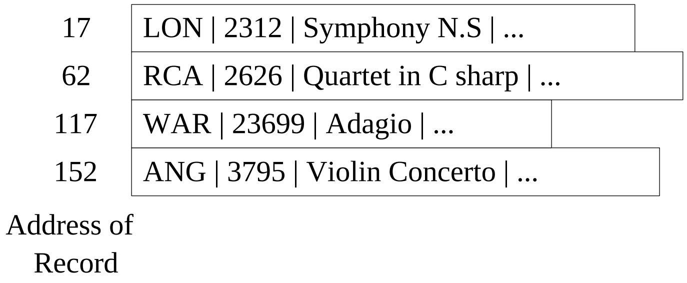
If we use an **index file** for sorting, the `AVAIL LIST` and positions of records remain unchanged. This is convenient.

Introduction to Indexing

- Simple indexes use simple arrays.
- An index lets us **impose order on a file** without rearranging the file.
- Indexes provide **multiple access paths** to a file - **multiple indexes** (library catalog providing search for author, book and title).
- An index can provide keyed access to variable-length record files.

A Simple Index for Entry-Sequenced File

Records (Variable Length)



17 LON | 2312 | Symphony N.S | ...

62 RCA | 2626 | Quartet in C sharp | ...

117 WAR | 23699 | Adagio | ...

152 ANG | 3795 | Violin Concerto | ...

Address of Record

Primary key = company label + record ID (LABEL ID).

Index :

key	Reference field
ANG3795	152
LON2312	17
RCA2626	62
WAR23699	117

- Index is sorted (main memory).
- Records appear in file in the order they entered.

How to search for a recording with given LABEL ID ?
"Retrieve recording" operation :

- Binary search (in main memory) in the index : find LABEL ID, which leads us to the reference field.
- Seek for record in position given by the reference field.

Two issues to be addressed :

- How to make a persistent index (i.e. how to store the index into a file when it is not in main memory).
- How to guarantee that the index is an accurate reflection of the contents of the file. (This is tricky when there are lots of additions, deletions and updates.)

LECTURE 11: INDEXING

Contents of today's lecture:

- Indexing

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 7.4 - 7.6, 7.7 - 7.10

Indexing

Operations in order to Maintain an Indexed File

1. Create the original empty index and data files.
2. Load the index file into memory before using it.
3. Rewrite the index file from memory after using it.
4. Add data records to the data file.
5. Delete records from the data file.
6. Update records in the data file.
7. Update the index to reflect changes in the data file.

We will take a closer look at operations 3-7.

Rewrite the Index File from Memory

When the data file is closed, the index in memory needs to be written to the index file.

An important issue to consider is what happens if the rewriting does not take place (power failures, turning the machine off, etc.)

Two important safeguards:

- Keep an status flag stored in the header of the index file. The status flag is "on" whenever the index file is not up-to-date. When changes are performed in the index in main memory the status flag in the file is turned on. Whenever the file is rewritten from main memory the status flag is turned off.
- If the program detects the index is out-of-date it calls a procedure that reconstruct the index from the data file.

Record Addition

This consists of appending the data file and inserting a new record in the index. The rearrangement of the index consists of “sliding down” the records with keys larger than the inserted key and then placing the new record in the opened space.

Note: This rearrangement is done in main memory.

Record Deletion

This should use the techniques for reclaiming space in files (Chapter 6.2) when deleting from the data file. We must delete the corresponding entry from the index:

- Shift all records with keys larger than the key of the deleted record to the previous position (in main memory); or
- Mark the index entry as deleted.

Record Updating

There are two cases to consider:

- The update changes the value of the key field:
  Treat this as a deletion followed by an insertion
- The update does not affect the key field:
  If record size is unchanged, just modify the data record. If record size changes treat this as a delete/insert sequence.

Indexes too Large to Fit into Main Memory

The indexes that we have considered before could fit into main memory. If this is not the case, we have the following problems:

- Binary searching of the index file is done on disk, involving several "seeks".
- Index rearrangement (record addition or deletion) requires shifting on disk.

Two main alternatives:

- Hashed organization (Chapter 11) (When speed is a top priority)
- Tree-structured (multilevel) index such as B-trees and B+ trees (Chapter 9,10) (It allows keyed and ordered sequential access).

But a simple index is still useful, even in secondary storage:

- It allows binary search to obtain a keyed access to a record in a variable-length record file.
- Sorting and maintaining an index is less costly than sorting and maintaining the data file, since the index is smaller.
- We can rearrange keys, without moving the data records when there are pinned records.

Indexing to Provide Access by Multiple Keys

In our recording file example, we built an index for LABEL ID key.
This is the **primary key**.
There may be **secondary keys**: title, composer and artist.

We can build **secondary key indexes**.

Composer index:

Secondary key	**Primary key**
Beethoven	ANG3795
Beethoven	DG139201
Beethoven	DG18807
Beethoven	RCA2626
Corea	WAR23699
Dvorak	COL31809
Prokofiev	LON2312

Note that in the above index the secondary key reference is to the primary key rather than to the byte offset.

This means that the primary key index must be searched to find the byte offset, but there are many advantages in postponing the binding of a secondary key to an specific address.

Record Addition

When adding a record, an entry must also be added to the secondary key index.
Store the field in **Canonical Form** (say capital letters, with pre-specified maximum length).
There may be duplicates in secondary keys. Keep duplicates in sorted order of primary key.

Record Deletion

Deleting a record implies removing all the references to the record in the primary index and in all the secondary indexes. This is too much rearrangement, specially if indexes cannot fit into main memory.
Alternative:

- Delete the record from the data file and the primary index file reference to it. Do not modify the secondary index files.
- When accessing the file through a secondary key, the primary index file will be checked and a deleted record can be identified.

This results in a lot of saving when there are many secondary keys.

The deleted record still occupy space in the secondary key indexes.
If a lot of deletions occur, we can periodically cleanup these deleted records from the secondary key indexes.

Record Updating

There are three types of updates :

- Update changes the secondary key :

  We have to rearrange the secondary key index to stay in sorted order.

- Update changes the primary key :

  Update and reorder the primary key index; update the references to primary key index in the secondary key indexes (it may involve some re-ordering of secondary indexes if secondary key occurs repeated in the file).

- Update confined to other fields :

  This won't affect secondary key indexes. The primary key index may be affected if the location of record changes in data file.

Retrieving Rec's using Combinations of Secondary Keys

Secondary key indexes are useful in allowing the following kinds of queries :

- Find all recording with composer "BEETHOVEN".
- Find all recording with title "Violin Concerto".
- Find all recording with composer "BEETHOVEN" **and** title "Symphony No.9".

This is done as follows :

Matches from composer index	**Matches from title index**	**Matched list (logical "and")**
ANG3795 DG139201 DG18807 RCA2626	ANG3795 COL31809 DG18807	ANG3795 DG18807

Use the matched list and primary key index to retrieve the two recordings from the file.

Improving the Secondary Index Structure: Inverted Lists

Two difficulties found in the proposed secondary index structures :

- We have to rearrange the secondary index file even if the new record to be added in for an existing secondary key.
- If there are duplicates of secondary keys then the key field is repeated for each entry, wasting space.

Solution 1

Make the secondary key index record consist of secondary key + array of references to records with secondary key.

Problems :

- The array will take a maximum length and we may have more records.
- We may have lots of unused spaces in some of the arrays (wasting space in internal fragmentation).

Solution 2 : Inverted Lists

Organize the secondary key index as an index containing one entry for each key and a pointer to a **linked list** of references.

Secondary Key Index File

0	Beethoven	3
1	Corea	2
2	Dvorak	5
3	Prokofiev	7

LABEL ID List File

0	LON2312	-1
1	RCA2626	-1
2	WAR23699	-1
3	ANG3795	6
4	DG18807	1
5	COL31809	-1
6	DG139201	4
7	ANG36193	0

Beethoven is a secondary key that appears in records identified by the LABEL IDs: ANG3795, DG139201, DG18807 and RCA2626 (check this by following the links in the linked list).

Advantages:

- Rearrangement of the secondary key index file is only done when a new composer's name is added or an existing composer's name is changed. Deleting or adding recordings for a composer only affects the **LABEL ID list file**. Deleting all recordings by a composer can be done by placing a "-1" in the reference field in the secondary index file.
- Rearrangement of the secondary index file is quicker since it is smaller.
- Smaller need for rearrangement causes a smaller penalty associated with keeping the secondary index file in disk.
- The **LABEL ID list file** never needs to be sorted since it is entry sequenced.
- We can easily reuse space from deleted records from the **LABEL ID list file** since its records have fixed-length.

Disadvantages :

- Lost of "locality" : labels of recordings with same secondary key are not contiguous in the **LABEL ID list file** (seeking). To improve this, keep the **LABEL ID list file** in main memory, or, if too big, use paging mechanisms.

Selective Indexes

We can build selective indexes, such as :
Recordings released prior to 1970, recordings since 1970.

This may be useful in queries involving boolean "and" operations :
"Retrieve all the recordings by Beethoven released since 1970".

Binding

In our example of indexes, when does the binding of the index to the physical location of the record happens ?

For the primary index, binding is at the time the file is constructed. For the secondary index, it is at the time the secondary index is used.

Advantages of postponing binding (as in our example):

- We need small amount of reorganization when records are added/deleted.
- It is a safer approach : important changes are done in one place rather than in many places.

Disadvantages :

- It results in slower access times (binary search in secondary index plus binary search in primary index).

When to use a **tight binding** ?

- When data file is nearly static (little or no adding, deleting or updating of records).
- When rapid retrieval performance is essential. Example : Data stored in CD-ROM should use tight binding.

When to use the **bind-at-retrieval** system?

- When record additions, deletions and updates occur more often.

LECTURE 12: COSEQUENTIAL PROCESSING

Contents of today's lecture:

- Cosequential processing (Section 8.1),
- Application: a general ledger program (Section 8.2)

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Section 8.1-8.2.

Cosequential Processing

Cosequential processing involves the **coordinated processing** of **two or more sequential lists** to produce a single output list.

The two main types of resulting output lists are :

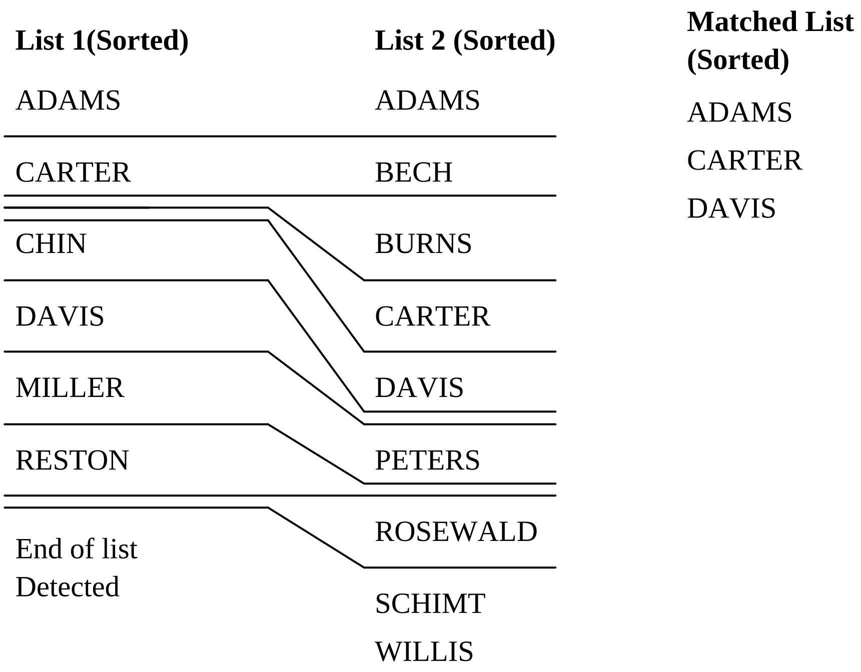
- Matching (intersection) of the items of the lists.
- Merging (union) of the items of the lists.

Examples of applications :

1. Matching :
   Master file - bank account info (account number, person name, account balance) - sorted by account number
   Transaction file - updates on accounts (account number, credit/debit info).

2. Merging :
   Merging two class lists keeping alphabetic order.
   Sorting large files (break into small pieces, sort each piece and then merge them).

Matching the Names in Two Lists



List 1(Sorted)

ADAMS
CARTER
CHIN
DAVIS
MILLER
RESTON
End of list
Detected

List 2 (Sorted)

ADAMS
BECH
BURNS
CARTER
DAVIS
PETERS
ROSEWALD
SCHIMT
WILLIS

Matched List (Sorted)

ADAMS
CARTER
DAVIS

Synchronization :

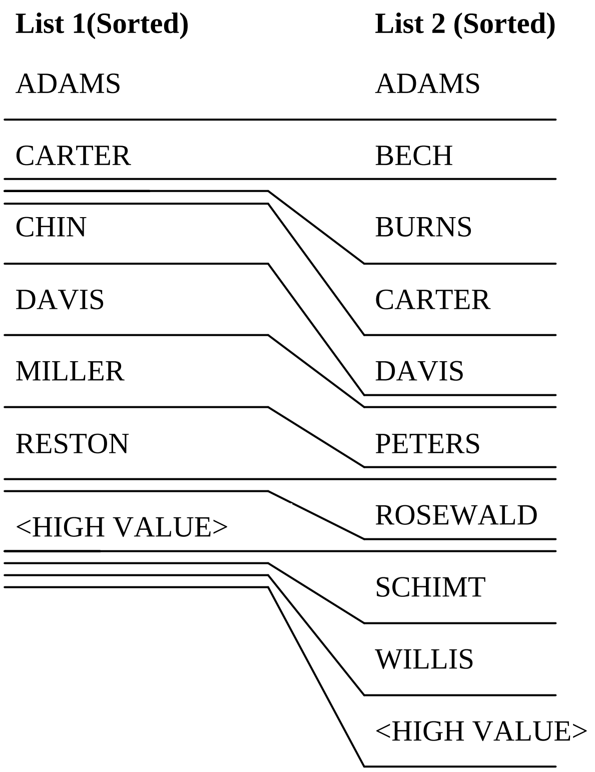
`item(i)` = current item from list i

```
if item(1) < item(2) then
     get next item from list 1
if item(1) > item(2) then
     get next item from list 2
if item(1) = item(2) then
     output the item to output list
     get next item from list 1 and list 2
```

Handling End-of-File/End-of-List Condition

Halt when we get to the end of **either** list 1 **or** list 2.

Merging the Names from Two Lists (Elimin. Repetit.)



List 1(Sorted)

ADAMS
CARTER
CHIN
DAVIS
MILLER
RESTON
<HIGH VALUE>

List 2 (Sorted)

ADAMS
BECH
BURNS
CARTER
DAVIS
PETERS
ROSEWALD
SCHIMT
WILLIS
<HIGH VALUE>

Merged List (Sorted)

ADAMS
BECH
BURNS
CARTER
CHIN
DAVIS
MILLER
PETERS
RESTON
ROSEWALD
SCHIMT
WILLIS

Modify the synchronization slightly :

```
if item(1) < item(2) then
     output item(1) to output list
     get next item from list 1
if item(1) > item(2) then
     output item(2) to output list
     get next item from list 2
if item(1) = item(2) then
     output the item to output list
     get next item from list 1 and list 2
```

Handling End-of-File/End-of-List Condition

1. Using a `<HIGH VALE>` as in the previous example:

   By storing `<HIGH VALUE>` in the current item for the list that finished, we make sure the contents of the other list is flushed to the output list.
   The **stopping criteria** is changed to :
   Halt when we get to the end of **both** list 1 **and** list 2.

2. Reducing the number of comparisons:

   We can perform a similar algorithm with less comparisons **without** using a `<HIGH VALUE>` as described above.
   The **stopping criteria** becomes:
   When we get to the end of **either** list 1 **or** list 2, we halt the program.

   Finalization: flush the unfinished list to the output list.

```
while (list 1 did not finish)
   output item(1) to output list
   get next item from list 1

while (list 2 did not finish)
   output item(2) to output list
   get next item from list 2
```

Cosequential Processing: A General Ledger Program

Ledger = A book containing accounts to which debits and credits are posted from books of original entry.

Problem: design a general ledger posting program as part of an accounting system.

Two files are involved in this process:

Master File: ledger file
- monthly summary of account balance for each of the book-keeping accounts.

Transaction File: journal file
- contains the monthly transactions to be posted to the ledger.

Once the journal file is complete for a given month, the journal must be **posted** to the ledger.

Posting involves associating each transaction with its account in the ledger.

Sample Ledger Fragment

Account Number	Account Title	Jan	Feb	Mar	Apr
101	checking account #1	1032.00	2114.00	5219.00	
102	checking account #2	543.00	3094.17	1321.20	
510	auto expense	195.00	307.00	501.00	
540	office expense	57.00	105.25	138.37	
550	rent	500.00	1000.00	1500.00	
⋮	⋮	⋮	⋮	⋮	⋮

Sample Journal Entry

Account Number	Check Number	Date	Description	Debit/Credit
101	1271	April 2, 01	Auto expense	- 79.00
510	1271	April 2, 01	Tune-up	79.00
101	1272	April 3, 01	Rent	- 500.00
550	1272	April 3, 01	Rent for April	500.00
102	670	April 4, 01	Office expense	- 32.00
540	670	April 4, 01	Printer cartridge	32.00
101	1273	April 5, 01	Auto expense	- 31.00
510	1273	April 5, 01	Oil change	31.00
⋮	⋮	⋮	⋮	⋮

Sample Ledger Printout

```
101 Checking account #1
    1271    | April 2, 01 | Auto expense     -  79.00
    1272    | April 3, 01 | Rent             - 500.00
    1273    | April 5, 01 | Auto expense     -  31.00
   Prev. Bal.:    5,219.00       New Bal.:   4,609.00

102 Checking account #2
     ⋮
510 Auto expense
     ⋮
540 Office expense
     ⋮
550 Rent
     ⋮
```

How to implement the Posting Process?

- Use account number as a **key** to relate journal transactions to ledger records.
- Sort the journal file.
- Process ledger and sorted journal **co-sequentially**.

Tasks to be performed:

- Update ledger file with the current balance for each account.
- Produce printout as in the example.

From the point of view of ledger account :
Merging (unmatched accounts go to printout)

From the point of view of journal account:
Matching (unmatched accounts in journal constitute an error)

The posting method is a combined merging/matching.

Ledger Algorithm

```
Item(1): always stores the current master record
Item(2): always stores the current transactions record

- Read first master record
- Print title line for first account
- Read first transactions record
While (there are more masters
       or there are more transactions) {
   if item(1) < item(2) then {
     Finish this master record:
     - Print account balances, update master record
     - Read next master record
     - If read successful, then print title line for
       new account           }
   if item(1) = item(2) {
     Transaction matches master:
     - Add transaction amount to the account balance
       for new month
     - Print description of transaction
     - Read next transaction record  }
   if item(1) > item(2) {
     Transaction with no master:
     - Print error message
     - Read next transaction record  }
}
```

LECTURE 13: COSEQUENTIAL PROCESSING - SORTING LARGE FILES

Contents of today's lecture:

- Cosequential Processing and Multiway Merge,
- Sorting Large Files (external sorting)

Reference : Folk, Zoellick and Riccardi, File Structures, 1998. Sections 8.3, 8.5 (up to 8.5.3).

Cosequential processing and Multiway Merging

K-way merge algorithm : merge K sorted input lists to create a single sorted output list.

We will adapt our 2-way merge algorithm :

- Instead of `List1` and `List2` keep an array of lists : `List[1], List[2], ..., List[K].`
- Instead of `item(1)` and `item(2)` keep an array of items : `item[1], item[2], ..., item[K].`

Merging Eliminating Repetitions

We modify our synchronization step :

```
if item(1) < item(2) then ...
if item(1) > item(2) then ...
if item(1) = item(2) then ...
```

As follows :

```
(1) minitem = index of minimum item in item[1],
              item[2],..., item[K]
(2) output item[minitem] to output list
(3) for i=1 to K do
(4)    if item[i]=item[minitem] then
(5)       get next item from List[i]
```

If there are no repeated items among different lists, lines (3)-(5) can be simplified to :

```
get next item from List[minitem]
```

Different ways of implementing the method :

Solution 1 : when the number of lists is small (say K $\leq$ 8).

- `Line(1)` does a sequential search on `item[1], item[2], ..., item[K]`.
  Running time : O(K)
- `Line(5)` just replaces `item[i]` with newly read item.
  Running time : O(1)

Solution 2 : when the number of lists is large.
Store current items `item[1], item[2], ...,item[K]` into priority queue (say, an array heap).

- `Line(1)` does a `min` operation on the array-heap.
  Running time : $O(1)$
- `Line(5)` performs a `extract-min` operation on the array-heap :
  Running time : $O(\log K)$
  and an `insert` on the array-heap
  Running time : $O(\log K)$

The detailed analysis of both algorithm is somewhat involved.

Let N = Number of items in output list
M = Number of items summing up all input lists
(Note $N \leq M$ because of possible repetitions.)

Solution 1
`Line(1)`: $K \cdot N$ steps
`Line(5)`, counting all executions: $M \cdot 1$ steps
Total time: $O(K \cdot N + M) \subseteq O(K \cdot M)$

Solution 2
`Line(1)` : $1 \cdot N$ steps
`Line(5)`, counting all executions : $M \cdot 2 \cdot \log K$ steps
Total time : $O(N + M \cdot \log K) = O((\log K) \cdot M)$

Merging as a Way of Sorting Large Files

- Characteristics of the file to be sorted:

  8,000,000 records

  Size of a record = 100 bytes

  Size of the key = 10 bytes

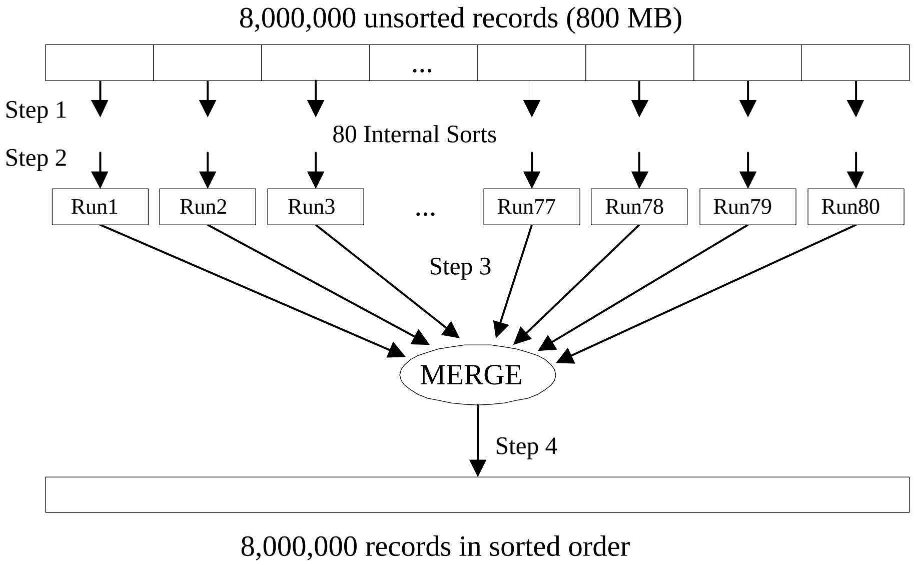
- Memory available as a work area : 10 MB (Not counting memory used to hold program, operating system, I/O buffers, etc.)

- Total file size = 800 MB
- Total number of bytes for all the keys = 80 MB
So, we cannot do internal sorting nor keysorting.

Idea :

1. Forming runs: bring as many records as possible to main memory, do internal sorting and save it into a small file. Repeat this procedure until we have read all the records from the original file.
2. Do a multiway merge of the sorted files.

   In our example, what could be the size of a run ?
   Available memory = 10 MB = 10,000,000 bytes
   Record size = 100 bytes
   Number of records that can fit into available memory = 100,000 records
   Number of runs = 80 runs



I/O operations are performed in the following times:

1. Reading each record into main memory for sorting and forming the runs.

2. Writing sorted runs to disk.

   The two steps above are done as follows:
   Read a chunk of 10 MEGS; Write a chunk of 10 MEGS
   (Repeat this 80 times)

   In terms of basic disk operations, we spend :
   For reading : 80 seeks[2] + transfer time for 800 MB
   Same for writing.

[2] Each chunk is read right after we wrote the previous run, so there is an initial seeking.

3. Reading sorted runs into memory for merging. In order to minimize “seeks” read one chunk of each run, so 80 chunks. Since the memory available is 10 MB each chunk can have 10,000,000/80 bytes = 125,000 bytes = 1,250 records

   How many chunks to be read for each run?
   size of a run/ size of a chunk = 10,000,000 / 125,000 = 80

   Total number of basic “seeks” = Total number of chunks (counting all the runs) is 80 runs x 80 chunks/run = 80^2 chunks.

   Reading each chunk involves **basic seeking**.

4. When writing a sorted file to disk, the number of basic seeks depends on the size of the output buffer: bytes in file/ bytes in output buffer.

   For example, if the output buffer contains 200 K, the number of basic seeks is : 200,000,000 / 200,000 = 4,000.

From steps 1-4 as the number of records (N) grows, step 3 dominates the running time.

There are ways of reducing the time for the bottleneck step (step 3):

- Allocate more hardware (e.g disk drives, memory)
- Perform the merge in more than one step - this reduces the order of each merge and increases the run sizes.
- Algorithmically increase the length of each run.
- Find ways to overlap I/O operations.

For details in the above steps see sections : 8.5.4 - 8.5.11.