CSI2131 FILE MANAGEMENT

Prof. Lucia Moura

Winter 2003

Contents of today's lecture:

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

References:
- 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

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).

How fast is main memory in comparison to secondary storage?

Typical time for getting info from:
- main memory: $\sim 120$ nanoseconds $= 120 \times 10^{-9}$ seconds
- magnetic disks: $\sim 30$ milliseconds $= 30 \times 10^{-3}$ seconds

An analogy keeping same time proportion as above:
- Looking at the index of a book: 20 secs
- Going to the library: 58 days

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 too 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)
  - Multi-level 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:

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

Lecture Notes
A C++ program for doing the same task:

```cpp
// 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();
}
```

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

```c
FILE * outfile;
```

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

```cpp
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.

---

**Physical Files and Logical Files**

- **Physical file**: a collection of bytes stored on a disk or tape
- **Logical file**: a "channel" (file 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 on a 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. The logical name is a variable inside the program, for instance `outfile`

---

**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:**

```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 output;
- "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++:
```c++
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream infile("myfile.txt");
    if (infile.fail()) {
        std::cerr << "Failed to open the file!
    } else {
        // Proceed with reading the file
    }
    return 0;
}
```
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 "afille" and with the following capabilities:

1. Input and output (appending mode):
   ```c++
   afille.open("myfile.txt");
   ```
2. Create a new file, or truncate existing one, for output:
   ```c++
   outfile.open("myfile.txt", ios::out);
   ```
3. Open an existing file for input and output, no creation allowed:
   ```c++
   ifstream infile("myfile.txt");
   ```

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):

```c
int 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 where data should be stored
- `Size`: number of bytes to be read

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

```c
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

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.

```c
In C:
fclose(outfile);
In C++:
outfile.close();
```
**Writing**

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

```c
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):

```c
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++:

```cpp
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:

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

In C++:

```cpp
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**

<table>
<thead>
<tr>
<th>purpose</th>
<th>default meaning</th>
<th>logical name in C</th>
<th>logical name in C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Output</td>
<td>Console/Screen</td>
<td>stdout</td>
<td>cout</td>
</tr>
<tr>
<td>Standard Input</td>
<td>Keyboard</td>
<td>stdin</td>
<td>cin</td>
</tr>
<tr>
<td>Standard Error</td>
<td>Console/Screen</td>
<td>stderr</td>
<td>cerr</td>
</tr>
</tbody>
</table>

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`):

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

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

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

You can also do:

```bash
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)


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:

```c
#include <fstream.h>

main() {
    char ch;
    ifstream 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>`
`02818FOLK FILE STRUCTURES <nl>`
`79733KYMUTH THE ART OF COMPUTER PROG<nl>`
`66633KYMUTH SURREAL NUMBERS <nl>`
`18395TOLKIENTHE HOBBIT <nl>`

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:
   1.3e in our file of books (field lengths are 5, 7, and 25).
   
   **87358CARROLLALICE IN WONDERLAND**
   03818FOLK FILE STRUCTURES
   79733KITH THE ART OF COMPUTER PROGR

2. Field beginning with length indicator:
   
   **058735907CARROLLALICE IN WONDERLAND**
   0603818IFOLKFILE STRUCTURES

3. Place delimiter at the end of fields:
   
   873591CARROLLALICE IN WONDERLAND!
   03818IFOLKFILE STRUCTURES!

4. Store field as keyword = value (self-describing fields):
   
   **ISBN-873591AU-CARROLL\ALICE IN WONDERLAND!**
   **ISBN-038181AU-FOLK!FILE STRUCTURES!**

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

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
  79733KITH THE ART OF COMPUTER PROGR

- Fixed-length records combined with variable-length fields:
  
  **delimited fields:**
  
  **873591CARROLLALICE IN WONDERLAND**
  03818FOLKFILE STRUCTURES
  79733KITH THE ART OF COMPUTER PROGR

- fields with length indicator:
  
  **058735907CARROLLALICE IN WONDERLAND**
  **0603818IFOLKFILE STRUCTURES**

Field Structures: advantages and disadvantages

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

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:

**873591CARROLLALICE IN WONDERLAND03818IFOLK...**

with fields with length indicator:

**058735907CARROLLALICE IN WONDERLAND0603818...**

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

3. Record beginning with length indicator.

Example:

with delimited field:

**32873591CARROLLALICE IN WONDERLAND**
**03818FOLKFILE 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: 873591CARROLL|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.

873581CARROLL|ALICE IN WONDERLAND|<n1>
03818|FOLK|FILE STRUCTURES|<n1>
797331WHAT IS THE ART OF COMPUTER PROGRAMMING|<n1>

Summary:

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Length Record</td>
<td>Easy to jump to the i-th record</td>
<td>Waste space with padding</td>
</tr>
<tr>
<td>Variable Length</td>
<td>Saves space when record sizes are diverse</td>
<td>Cannot jump to the i-th record, unless through an index file</td>
</tr>
</tbody>
</table>

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 ...

In C++:

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

- `seekg` — moves the get pointer.
- `seekp` — moves the put pointer.

**General use:**

```cpp
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:

```cpp
infile.seekg(0,ios::beg);
infile.seekg(0,ios::end);
infile.seekg(-10,ios::cur);
```

**Seeking**

Generic seek function:

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

Example:

```cpp
Seek(infile, 3030)
```
Moves to byte 3030 in file.

In C style:

Function prototype:

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

- `origin`: 0 — `fseek` from the beginning of file
- 1 — `fseek` from the current position
- 2 — `fseek` from the end of file

Examples of usage:

```cpp
fseek(infile,0L,0); // moves to the beginning of the file
fseek(infile,0L,1); // moves to the end of the file
fseek(infile,-10L,1); // moves back 10 bytes from the current position
```
Consider the following sample program:

```cpp
#include <fstream.h>
int main() {
  ofstream myfile;
  myfile.open("test.txt", ios::in|ios::out|ios::trunc |
            ios::binary);
  myfile<<"Hello, world.\r\nHello, again.\r"
       <<"X\r\nY\r\nZ\r
";
  myfile.close();
  return 0;
}
```

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

```
1111111111111111
```

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

```
1111111111111111
```

---

**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


---

**Secondary Storage Devices**

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

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

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

Looking at a surface:

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 x 63 records per track
16 x 126 = 2,016 records per cylinder
number of cylinders = \( \frac{2,016}{256} \) \( \approx \) 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

- 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?
(Note: 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.

**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/O 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.

**Data 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 non-data overhead than with sector-addressing.

Example:

**Disk characteristics:**

- Block-addressable disk drive
- Size of track = 20,000 bytes
- Non-data 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 = \left\[ \frac{20000}{100} \right\] = 150 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 = \left\[ \frac{20000}{600} \right\] = 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:

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

Example: Disk Characteristics:
- Average seek time = 8 msec
- Average rotational delay = 3 msec
- Maximum rotational delay = 6 mssec
- 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 = \left( \frac{1}{10000} \min \right) / 170 = \left( \frac{1}{10000} \times 60 \right) / 170 \text{ secs} = 6/170,000 \text{ secs} = 0.0035 \text{ mssec}

Average total time = average seek + average rotational delay + transfer time = 8 + 3 + 0.0035 = 11.0035 mssecs.

Comparing sequential access to random access

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 mssec
- Average rotational delay = 3 mssec
- Average transfer time for 1 track = 60/10000 \times 6 = 6 mssec
- Average total time per track = 8 + 3 + 6 = 17 mssec
- Total time for file = 17 mssecs x 100 = 1700 mssec

b) Reading a file accessing randomly each record
Average total time to read all records = Number of records \times average time to read a sector = 34,000 \times 11,035 mssec \approx 371,113 mssec

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)
   Ax a 8-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.
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


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

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:

```
01111101
```

- Data blocks are separated by interblock gaps.
- 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:

\[ \begin{align*}
1111111 & \\
0000000 & \\
0010000 &
\end{align*} \]

Since 000000000 cannot correspond to a valid byte, this is used to mark the \textit{inert block gap}.

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

---

\textbf{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

\[ s = n x (b + g) \]

An enormous saving by just choosing a higher blocking factor.

**Effective Recording Density (ERD)**

\[ \text{ERD} = \frac{\text{number of bytes per block}}{\text{number of inches to store a block}} \]

In previous example:

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

The Nominal Density was 6,250 bpi!

---

\textbf{Estimating Data Transmission Times}:

Nominal Rate = tape density (bpi) \times tape speed (ips)

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

Effective Transmission Rate = E.R.D. \times tape speed

In the previous example:

A) E.T.R. = 316.4 bytes/inch \times 200 inches/sec = 63,280 bytes/sec \approx 63.3 \text{ KB/sec}
B) E.T.R. = 4,545.4 bytes/inch \times 200 inches/sec = 909,080 bytes/sec = 909 \text{ KB/sec}

Note: There is a tradeoff between \textit{increasing} blocking factor for increasing speed & space utilization and \textit{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 \textit{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

<table>
<thead>
<tr>
<th>LAND</th>
<th>PIT</th>
<th>LAND</th>
<th>PIT</th>
<th>LAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 1 0 0 ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 158 words of length 13 having the desired property.

Eight to fourteen modulation (EFM) encoding table:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Original</th>
<th>Translated</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000000</td>
<td>01001000100000</td>
</tr>
<tr>
<td>1</td>
<td>0000001</td>
<td>10001000000000</td>
</tr>
<tr>
<td>2</td>
<td>0000010</td>
<td>10010000000000</td>
</tr>
<tr>
<td>3</td>
<td>0000011</td>
<td>10001000000000</td>
</tr>
<tr>
<td>4</td>
<td>0000100</td>
<td>01000100000000</td>
</tr>
<tr>
<td>5</td>
<td>0000101</td>
<td>00000100000000</td>
</tr>
<tr>
<td>6</td>
<td>0000110</td>
<td>00100000000000</td>
</tr>
<tr>
<td>7</td>
<td>0000111</td>
<td>01001000000000</td>
</tr>
<tr>
<td>8</td>
<td>0001000</td>
<td>01001000000000</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>CD-ROM</th>
<th>Magnetic Disks</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/LV = Constant Linear Velocity</td>
<td>C/AV = Constant Angular Velocity</td>
</tr>
<tr>
<td>Sectors organized along a spiral</td>
<td>Sectors organized in concentric track</td>
</tr>
<tr>
<td>Sectors have same linear length (data packed at its maximum density permitted)</td>
<td>Sectors have same angular length (data written less densely in the outer tracks)</td>
</tr>
<tr>
<td>Advantage: Takes advantage of all storage space available</td>
<td>Advantage: Operates on constant speed timing marks to select tracks</td>
</tr>
<tr>
<td>Disadvantage: Has to change rotational speed when seeking (slower towards the outside)</td>
<td>Disadvantage: It doesn’t use up all storage available</td>
</tr>
</tbody>
</table>

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 **triad-and-error**.

This slows down the performance!
CD-ROM Strength and Weaknesses:

- Seek performance (~ 500 ms) - Slow

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

- Data transfer rate = 150 KB/sec - Slow (while ~ 3000 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.

Contents of today's lecture:
- A Journey of a Byte
- Buffer Management


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.

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 input and one for output.

A new trip to get a sector of input 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:
POLF, 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 265 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 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 24 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 -
C -.-
D -..-
E .
F ..-
G --.
H ---
I .
J .-. .
K -. .
L .-.
M ..-
N -.
O ---
P .--
Q --.
R .-.
S .-
T -
U .-
V .-.
W .--
X .-.
Y .-. .
Z .--

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

<table>
<thead>
<tr>
<th>Letter</th>
<th>A</th>
<th>M</th>
<th>S</th>
<th>Y</th>
<th>/b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Code</td>
<td>00</td>
<td>1010</td>
<td>11</td>
<td>1011</td>
<td>100</td>
</tr>
</tbody>
</table>

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

Encoded message:
101001001101101100111100

25 bits rather than 80 bits (10 bytes)!

Huffman Tree (for easy decoding)

Consider the encoded message:
101001001101101100111100

- 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:

\[ B(T) = \sum_{c \in C} f(c) \cdot d_T(c), \]

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 = \( \frac{B(T)}{\text{total number of characters}} = \frac{25}{10} = 2.5 \)

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: [L1, S1, Y1, A2, B2, M3]

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

Subtrees: [Y1, A2, M3]

Subtrees: [A2, M3]

Subtrees: [3, 4]

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:

- \( \text{Q.insert(x)} \): insert \( x \) to \( Q \)
- \( \text{Q.min}() \): returns element of smallest key
- \( \text{Q.extract-min()} \): 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
\( (c_1, f[c_1]), (c_2, f[c_2]), \ldots, (c_n, f[c_n]) \)
Output: returns the Huffman Tree

```java
Make priority queue Q using c_1, c_2, ..., c_n;
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[l] + f[r];
    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.

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:

```
'aaabbbbaaabaaaaaabaaabb
```

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' has been seen, we now see 'aa'.
'b' has been seen, we now see 'ab'.
```

Note that this is a dynamic method.

---

**Practice Exercise**

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

```
'aaabbcdddddeab'
```

```
'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

- 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: 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:

```
<q> <a> <00> <b> <01> <b> <01> <c> <01> <q> <00> <q> <00> <b> <1001> <b>
```

Note that \(<q>\> 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 convenience.

Total number of bits in the encoded example:

\[
(10 \times 8) + (0 + 1 + 2 \times 2 + 4 \times 3 + 2 \times 4) = 105 \text{ bits}
\]

The original message was represented using \(24 \times 8 = 192\) bits.
Decompressing

1 2 3 4 5 6 7 8 9 10
0a1a10b1b1b2a2a16a12b19b

-------------
| previous laded |
|  pointer  |  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  a  a  b  ab  bb  a  a  a  a  ba  a  a  a  a  ab  aabb
```
Lecture 9: Reclaiming Space in Files

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 deletion the file may look like:

<table>
<thead>
<tr>
<th>List head</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edwards</td>
<td>Williams</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>-1</td>
<td>Smith</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

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>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>10 bytes</td>
<td>5</td>
<td>10 bytes</td>
<td>8 bytes</td>
</tr>
</tbody>
</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?

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).

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;
LECTURE 10: BINARY SEARCHING,
KEYSORTING AND INDEXING

A binary search\(^1\) algorithm in C++:

```cpp
class FixedRecordFile{
    public:
        int NumRecs();
        int ReadByRN(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.ReadByRN(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 appears sometimes found in the textbook.

Binary Search versus Sequential Search:

- Binary Search: \(O(\log n)\)
- 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
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.

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.

A Simple Index for Entry-Sequence File

Records (Variable Length)

<table>
<thead>
<tr>
<th>Address of Record</th>
<th>Record</th>
<th>artist</th>
<th>album</th>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>LN</td>
<td>2312</td>
<td>Symphony No. 9</td>
<td>...</td>
</tr>
<tr>
<td>62</td>
<td>RCA</td>
<td>2626</td>
<td>Quartet in C Sharp</td>
<td>...</td>
</tr>
<tr>
<td>117</td>
<td>WAR</td>
<td>23699</td>
<td>Adagio</td>
<td>...</td>
</tr>
</tbody>
</table>

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

Index:

<table>
<thead>
<tr>
<th>key</th>
<th>Reference field</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANG</td>
<td>152</td>
</tr>
<tr>
<td>RCA</td>
<td>62</td>
</tr>
<tr>
<td>WAR</td>
<td>7099</td>
</tr>
</tbody>
</table>

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.)

Contents of today's lecture:

- Indexing

**Reference:** FOLK, ZOELICK 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:

<table>
<thead>
<tr>
<th>Secondary key</th>
<th>Primary key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beethoven 1965</td>
<td>ANG3795</td>
</tr>
<tr>
<td>Beethoven 192901</td>
<td>DG139201</td>
</tr>
<tr>
<td>Beethoven 198907</td>
<td>DG18807</td>
</tr>
<tr>
<td>Beethoven RCA2626</td>
<td>Beethoven</td>
</tr>
<tr>
<td>Corea WAR2069</td>
<td>Corea</td>
</tr>
<tr>
<td>Dvorak COL3809</td>
<td>Dvorak</td>
</tr>
<tr>
<td>Prokofiev LON2312</td>
<td>Prokofiev</td>
</tr>
</tbody>
</table>

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 clean up 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 reorganize 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:

<table>
<thead>
<tr>
<th>Matches from composer index</th>
<th>Matches from title index</th>
<th>Matched list (logical “and”)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANG3795</td>
<td>ANG3795</td>
<td>ANG3795</td>
</tr>
<tr>
<td>DG139201</td>
<td>COL31809</td>
<td>DG18807</td>
</tr>
<tr>
<td>DG18807</td>
<td>DG18807</td>
<td></td>
</tr>
</tbody>
</table>

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 is 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 index as an index containing one entry for each key and a pointer to a linked list of references.

<table>
<thead>
<tr>
<th>Secondary Key Index File</th>
<th>LABEL ID List File</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Beethoven</td>
<td>3 LON2312</td>
</tr>
<tr>
<td>1 Corea</td>
<td>2 WAR2309</td>
</tr>
<tr>
<td>2 Dvorak</td>
<td>5 ANG3795</td>
</tr>
<tr>
<td>3 Prokofiev</td>
<td>7 COL31809</td>
</tr>
<tr>
<td>6 DG139201</td>
<td>4 DG18807</td>
</tr>
<tr>
<td>7 ANG36130</td>
<td>0 RCA3626</td>
</tr>
</tbody>
</table>

Beethoven is a secondary key that appears in records identified by the LABEL ID: ANG3795, DG139201, DG18807 and RCA3626 (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.
- Rearrangemet 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:
- Loss 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.
Contents of today’s lecture:

- Consequential processing (Section 8.1).
- Application: a general ledger program (Section 8.2)


**Consequential Processing**

Consequential 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 alphabetical order.
   - Sorting large files (break into small pieces, sort each piece and then merge them).

**Matching the Names in Two Lists**

<table>
<thead>
<tr>
<th>List 1 (Sorted)</th>
<th>List 2 (Sorted)</th>
<th>Matched List (Sorted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAMS</td>
<td>ADAMS</td>
<td></td>
</tr>
<tr>
<td>CARTER</td>
<td>BEICH</td>
<td></td>
</tr>
<tr>
<td>CHIN</td>
<td>BURNS</td>
<td></td>
</tr>
<tr>
<td>DAVIS</td>
<td>CARTER</td>
<td></td>
</tr>
<tr>
<td>MILLER</td>
<td>DAVIS</td>
<td></td>
</tr>
<tr>
<td>RESTON</td>
<td>PETERS</td>
<td></td>
</tr>
<tr>
<td>End of list</td>
<td>ROSEWALD</td>
<td>SCHMIT</td>
</tr>
<tr>
<td>Detected</td>
<td></td>
<td>WILLIS</td>
</tr>
</tbody>
</table>

**Synchronization:**

\[ \text{item}(i) = \text{current item from list 1} \]

if \[ \text{item}(1) < \text{item}(2) \] then

get next item from list 1

if \[ \text{item}(1) > \text{item}(2) \] then

get next item from list 2

if \[ \text{item}(1) = \text{item}(2) \] 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.)**

<table>
<thead>
<tr>
<th>List 1 (Sorted)</th>
<th>List 2 (Sorted)</th>
<th>Merged List (Sorted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADAMS</td>
<td>ADAMS</td>
<td>ADAMS</td>
</tr>
<tr>
<td>CARTER</td>
<td>BEICH</td>
<td>CARTER</td>
</tr>
<tr>
<td>CHIN</td>
<td>BURNS</td>
<td>CHIN</td>
</tr>
<tr>
<td>DAVIS</td>
<td>CARTER</td>
<td>DAVIS</td>
</tr>
<tr>
<td>MILLER</td>
<td>DAVIS</td>
<td>MILLER</td>
</tr>
<tr>
<td>RESTON</td>
<td>PETERS</td>
<td>RESTON</td>
</tr>
<tr>
<td>&lt;HIGH VALUE&gt;</td>
<td>ROSEWALD</td>
<td>SCHMIT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>WILLIS</td>
</tr>
</tbody>
</table>

**Modify the synchronization slightly:**

if \[ \text{item}(1) < \text{item}(2) \] then

output \[ \text{item}(1) \] to output list

get next item from list 1

if \[ \text{item}(1) > \text{item}(2) \] then

output \[ \text{item}(2) \] to output list

get next item from list 2

if \[ \text{item}(1) = \text{item}(2) \] 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 VALUE>` 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
   ```

Sample Ledger Fragment

<table>
<thead>
<tr>
<th>Account Number</th>
<th>Account Title</th>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>checking account #1</td>
<td>1032.00</td>
<td>2114.00</td>
<td>5219.00</td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>checking account #2</td>
<td>543.00</td>
<td>3094.17</td>
<td>1321.20</td>
<td></td>
</tr>
<tr>
<td>510</td>
<td>auto expense</td>
<td>195.00</td>
<td>307.00</td>
<td>301.00</td>
<td></td>
</tr>
<tr>
<td>540</td>
<td>office expense</td>
<td>57.00</td>
<td>105.20</td>
<td>138.37</td>
<td></td>
</tr>
<tr>
<td>550</td>
<td>rent</td>
<td>600.00</td>
<td>1000.00</td>
<td>1500.00</td>
<td></td>
</tr>
</tbody>
</table>

Sample Journal Entry

<table>
<thead>
<tr>
<th>Account Number</th>
<th>Check Number</th>
<th>Date</th>
<th>Description</th>
<th>Debit/Credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>1271</td>
<td>April 2, 01</td>
<td>Auto expense</td>
<td>-79.00</td>
</tr>
<tr>
<td>510</td>
<td>1271</td>
<td>April 2, 01</td>
<td>Tune-up</td>
<td>79.00</td>
</tr>
<tr>
<td>101</td>
<td>1272</td>
<td>April 3, 01</td>
<td>Rent</td>
<td>-500.00</td>
</tr>
<tr>
<td>550</td>
<td>1272</td>
<td>April 3, 01</td>
<td>Rent for April</td>
<td>500.00</td>
</tr>
<tr>
<td>102</td>
<td>670</td>
<td>April 4, 01</td>
<td>Office expense</td>
<td>-32.00</td>
</tr>
<tr>
<td>510</td>
<td>670</td>
<td>April 4, 01</td>
<td>Printer cartridge</td>
<td>32.00</td>
</tr>
<tr>
<td>101</td>
<td>1273</td>
<td>April 5, 01</td>
<td>Auto expense</td>
<td>-31.00</td>
</tr>
<tr>
<td>510</td>
<td>1273</td>
<td>April 5, 01</td>
<td>Oil change</td>
<td>31.00</td>
</tr>
</tbody>
</table>
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
    - Print description of transaction
    - Read next transaction record
  }
  if item(1) > item(2) {
    Transaction with no master:
    - Print error message
    - Read next transaction record
  }
}

---

**Contents of today’s lecture:**

- Consequent 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).

**Consequent 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 minitem 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 ≤ 8).

- Line(1) does a sequential search on item[1], item[2], ..., item[K].
  Running time: O(K)
- Line(5) just replaces line 1 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 ≤ M because of possible repetitions.)

Solution 1
Line(1): K · N steps
Line(5), counting all executions: M · 1 steps
Total time: O(K · N + M) ≤ O(K · M)

Solution 2
Line(1): 1 · N steps
Line(5), counting all executions: M · 2 · log K steps
Total time: O(N + M · log K) = O((log K) · 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 key sorting.

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 multi-way 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
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.