CSI2131 FILE MANAGEMENT

Prof. Lucia Moura

Winter 2005

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 :

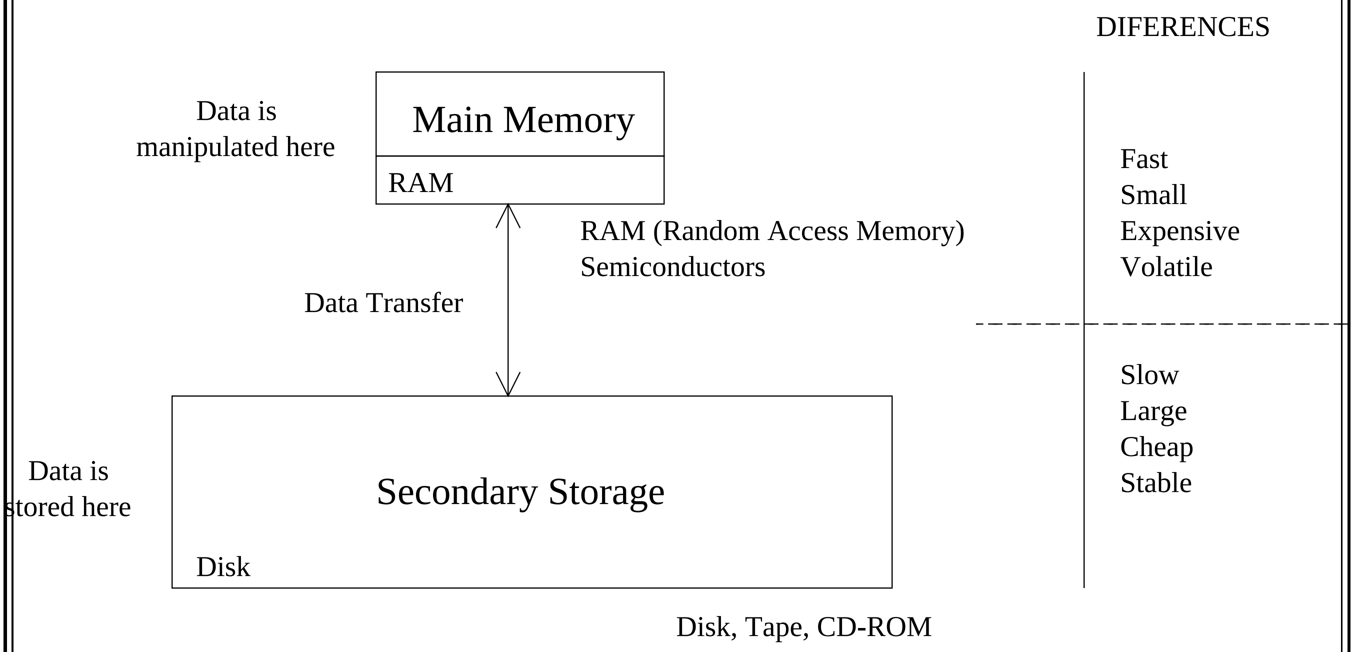
  Representation of Data
  +
  Operations for accessing data

  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: $\sim$ 60 nanoseconds $= 60 \text{ x } 10^{-9}$ secs
magnetics disks: $\sim$ 15 milliseconds $= 15 \text{ x } 10^{-3}$ 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

2. Disks became more common
   - **Direct access**
   - **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>// to use fstream class
using namespace std; // to use standard C++ library

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;
- `"rb", "wb", "ab","r+b","w+b","a+b"`: same as above but the file is open in binary mode.

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::ate` initially position at the end of file;
- `ios::trunc` always create a new file (truncate if exists);
- `ios::binary` open in binary mode (rather than text mode).

C: C++:	`r` `in`	`w` `out|trunc` or `out`	`a` `out|app`
C: C++:	`r+` `out|in`	`w+` `out|in|trunc`	`a+` `out|in|app`

All obtions above, if followed by `b` in C, would have `|binary`.

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::out|ios::app);
   ```
2. create a new file, or truncate existing one, for output:

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 <u>closed</u> 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 :

```
#include <fstream>
using namespace std;
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 records, 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:

```
058735907CARROLL19ALICE IN WONDERLAND
050381804FOLK15FILE 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`.

In C style :

Function prototype:

```
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:

```
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>
using namespace std;
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: heads, surfaces, cylinders, tracks, sectors.
- Alternative disk organization (organizing tracks by blocks)
- Nondata overhead
- The cost of a disk access
- Disk as a bottleneck

References:
Folk, Zoellick and Riccardi, File Structures, 1998.
Sections 3.1.
Online resources:

`Wikipedia:`
`http://en.wikipedia.org/wiki/Secondary_storage`

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
    CD-R = Compact Disc, Recordable
    CD-RW = Compact Disc, ReWritable
    DVD = Digital Video Disc or Digital Versatile Disc
- 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

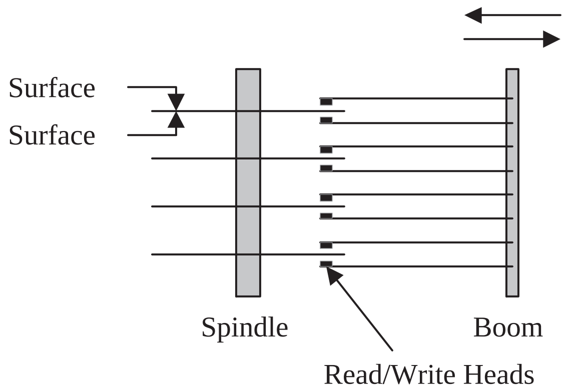
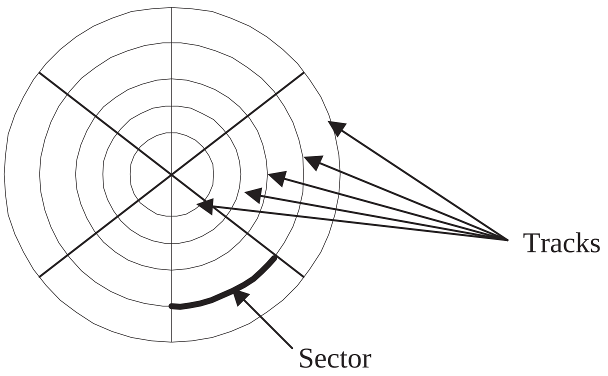


Figure 1: Disk drive with 4 platters and 8 surfaces

Looking at a surface:



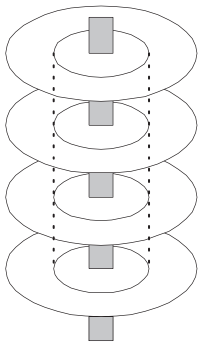
- Disk contains concentric **tracks**
- Tracks are divided into **sectors**
- A **sector** is the smallest addressable unit in a disk

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.

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.

Alternative disk organization

Organizing Tracks by Blocks

- In certain disks, 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

Every disk has some nondata overhead, that is, an amount of space used for extra stuff other than data.

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

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

From now on in these notes, unless otherwise mentioned, we will only consider sector-organized disks.

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

Note that one revolution takes (1/spindle speed) minutes; so one revolution takes 1/10000 minutes = 6 msecs =max rotat. delay

What is the average time to read one sector ?

Transfer time = revolution time/# sectors per track = 6/170 msecs ≈ 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 $\approx$ 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 (20-2000 MB for PC in 2004). 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 and Optical disks
  - Physical organization of CD-ROM
  - CD-ROM addressing and seeking
  - Disk versus CD-ROM
  - Strength and weaknesses of CD-ROM
  - Differences between CD-ROM and other optical discs: CD-R, CD-RW, DVD

References:

FOLK, ZOELLICK AND RICCARDI, File Structures, 1998. Section 3.2, 3.5 and 3.6. Online resources:

`Wikipedia:`
`http://en.wikipedia.org/wiki/Data_storage_device`
`How stuff works: http://www.howstuffworks.com/`

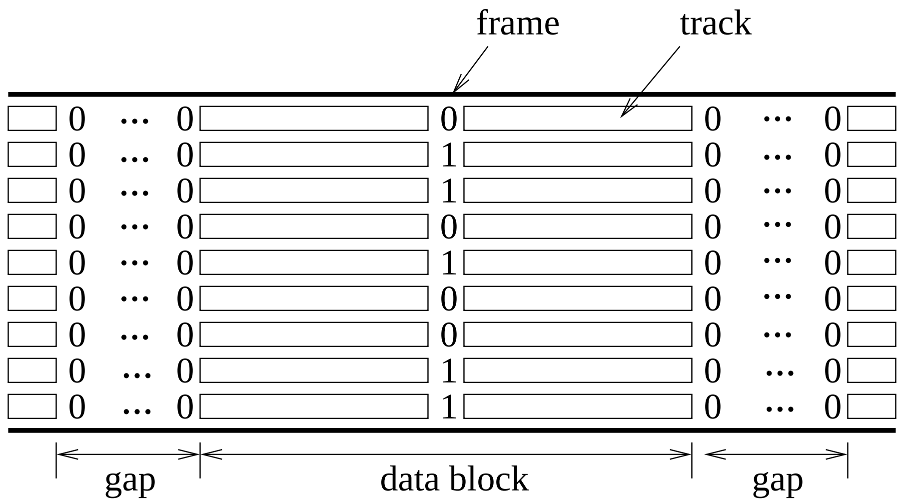
Characteristics of Magnetic Tapes

- No direct access, but very fast sequential access.
- Resistant to different environmental conditions.
- Easy to transport, store, cheaper than disk (lower cost per bit), higher bit density.
- Before, it was widely used to store application data; nowadays, it's mostly used for backups or archives (tertiary storage). This is due to greater improvement in disk density and price, over tapes.

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:



- **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**.

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)

Note: for 9-track tape bpi is the same as bytes per inch of tape.
- 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 $\sim$ 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 $\sim$ 63.3 KB/sec
B) E.T.R. = 4,545.4 bytes/inch x 200 inches/sec = 909,080 bytes/sec $\sim$ 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**.

Table: Comparison of Price Per Gigabyte (US$/GB)

Year	1994	1996	1998	2000	2002	2004
Tape Media	$5.00	$3.00	$1.80	$1.00	$0.60	$0.35
Performance Hard Disk	N/A	N/A	$80.00	$11.00	$4.50	$1.50
Enterprise Hard Disk	$8000	$1200	$400.00	$100.00	$25.00	$8.00

Table extracted from article:
Rich Harada, "is it tape and disk or tape versus disk?" Computer Technology Review, June, 2004.

Physical Organization of CD-ROM

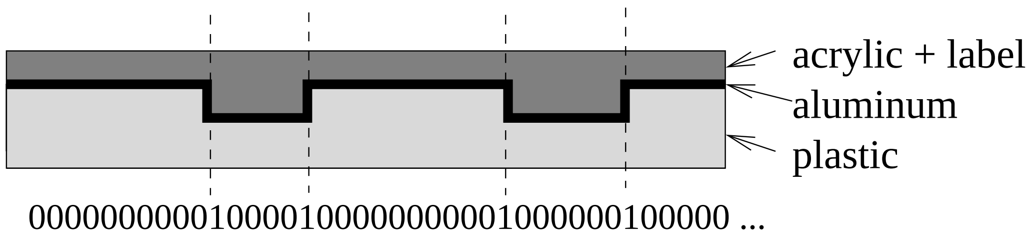
CD-ROM = Compact Disc - read only memory (write once)

Digital data is represented as a series of **pits** and **lands** on a spiral track.
pit = a little depression, forming a lower level in the track.
land = the flat part between pits, or the upper levels in the track.

The disk is made of polycarbonate plastic on which the pits are pressed, then covered with an aluminum layer, which is covered with an acrylic layer where the label is printed.

Reading a CD is done by shining a laser at the disc (from below) and detecting changing reflection patterns. Changes in height of the track causes changes in its reflectivity.

The binary data is represented by changes in height
(a common mistake is to think that pits are 0s and lands are 1s).
1 = change in height (land to pit or pit to land)
0 = a “fixed” amount of time between 1’s



Note: We cannot have two 1’s in a row!

Indeed, because of other media 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. The minimum number of bits required so that we have enough distinct code words with the desired property is 14 (with 13 bits we have only 188 such words).
The so-called **eight to fourteen modulation** encodes each sequence of 8 bits as a 14-bit code word.

Eight to fourteen modulation (EFM) encoding table:

Decimal Value	Original Bits	Translated 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
...	...	...

Because two code words in a row may have 1's too close to each other, 3 extra "merge" bits are used to separate codewords. So 17 bits are used to encode 8 bits of information.

CD-ROM addressing and Seeking

Addressing and transfer rate

Each sector contains 2352 bytes:

- 12 bytes synchronization (one 00, ten FF, one 00)
- 4 bytes ID (minute,second,sector,mode)
- 2048 bytes data
- 288 bytes for layered error correcting code

Therefore, each sector holds 2KB of encoded data.

Each second is divided into 75 seconds.
data held in a 60 Min CD : 60 min x 60 sec/min x 75 sectors/sec = 270,000 sectors = 540,000 KB $\sim$ 540 MB

The first CD-ROM drives played 75 sectors per second, leading to data transfer rate = $2 \times 75 = 150$ KB/sec = 0.15 MB/sec.
The data transfer rate of later CD drives is measured as a factor of this original data transfer rate (1X CD-ROM drive).

Earlier CD-ROM drives used Constant Linear Velocity (CLV) where the angular spin (measured in rpm) had to be continuously adjusted. This allowed both a uniform space of the pits along the spiral track and a constant transfer rate independent on the head position. But it limited the driver speed.

Newer drives use Constant Angular Velocity, as it was easier to increase drive speeds using this principle. In CAV CD-ROM drives, the transfer rate depends on the head position (more data is read per unit of time in the outer parts of the track).

Table: CD-ROM drive transfer rates *(from: www.USByte.com)*

Drive speed rating	Principle	Min rate	Max rate	Average rate
1 X	CLV	0.15 (MB/s)	0.15 (MB/s)	0.15 (MB/s)
2 X	CLV	0.30	0.30	0.30
4 X	CLV	0.60	0.60	0.60
6 X	CLV	0.90	0.90	0.90
8 X	CLV	1.20	1.20	1.20
10 X	CLV	1.50	1.50	1.50
12 X	CLV	1.80	1.80	1.80
12 X to 20 X	CAV	1.80	3.00	2.40
12 X to 24 X	CAV	1.80	3.60	2.70
12 X to 32 X	CAV	1.80	4.80	3.30
12 X to 48 X	CAV	1.80	7.20	4.50

Difficulty in Seeking

CD-ROM drives have much poorer seek performance than hard disks.

CD-ROM drives don't have cylinders, but a continuous spiral track which makes it harder to find specific information. The early CD-ROM drives had **access time** (seek time is part of it) over 300 msecs, while nowadays a high end CD-ROM would have under 100 msecs (still much slower than magnetic disks 10-15 msecs).

Comparing CD-ROM with magnetic disks

CD-ROM	**Magnetic Disks**
Sectors organized along a spiral	Sectors organized in concentric tracks
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: data is retrieved at the same speed in every track.
Disadvantage: seeking is very slow	Disadvantage: it doesn't use up all storage available

CD-ROM Strength and Weaknesses

- **Random Access:**
  Seek/access performance ($\sim$ 100 msecs) - **Slow!**

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

- **Sequential Access:**
  Data transfer rate - 4.5 MB/sec for the 48X model - **Slow!**
  Typical magnetic disk (2004) had a transfer rate of 30 MB/sec.
  But it is much faster than floppy disks (0.06 MB/sec).

- Storage capacity is $\sim$ 650-700 MB; it is good for storing texts.

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

Comparing CD-ROM with Other Optical Discs

CD-R (Compact Disc - Recordable) and have different technologies that simulates the effect of pits and lands.
Other than the writing method, these discs have the same organization and reading method as CD-ROM discs.

CD-R discs have **color dies** that can be **burned** (written) once.

The CD-R has a polycarbonate plastic with a spiral groove to guide reading and writing, coated with a thin layer of a special die, followed by a thin reflective layer and the protective coating.
A CD-R drive (CD-burner) can write on the CD-R by using the laser beam to "burn" the die to simulate pits. When reading the information, the laser operates at a low enough power not to burn the CD-R.

CD-RW discs have a phase-change material that can be written and overwritten several times.

Instead of the special die layer, this CD has a phase-changing recording layer. The laser beam can melt crystals in this recording layer into a non-crystalline amorphous phase or anneal them at a lower temperature back to the crystalline state. The difference in reflection simulates the effect of pits and lands.

DVD (Digital Video Disc or Digital Versatile Disc)

Same concept of lands and pits and spiral track as CDs.
Its capacity is much higher than CDs due to the following factors:

- higher-density:
  the track on a DVD is 2.16 times thinner than a CD and pits are more tightly spaced within a track (minimum pit length is 2.08 times smaller)
- less overhead, more data:
  wastes less space in error correction using better methods.
- multilayer storage:
  A DVD can have up to 4 layers (2 on each side). The outer layers are coated with semi-reflective gold which allows the laser to focus into the inner layers.

Capacity of DVDs:

- Single-sided, single layer ($\sim$ 4.7 GB)
- Single-sided, double layer ($\sim$ 8.5 GB)
- Double-sided, single layer ($\sim$ 9.4 GB)
- Double-sided, double layer ($\sim$ 17.1 GB)

Data transfer rates are given in multiples of 1350 KB/sec, so a 16X DVD drive has a rate of 21600 KB/sec.

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.

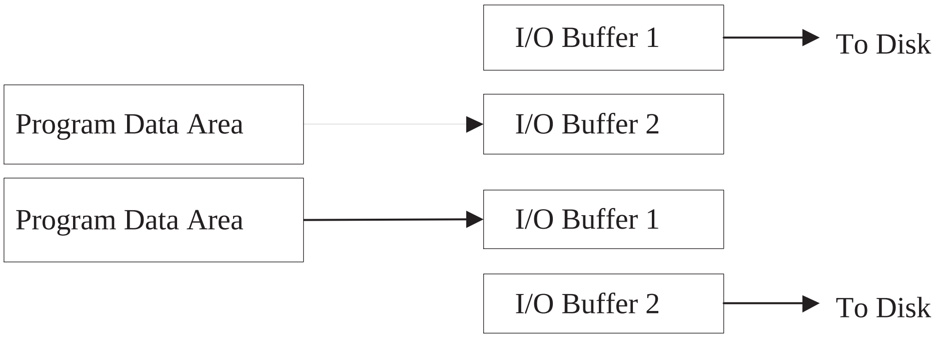
Similarly, the buffer for `output` would be written to the file only when full (or when file was closed).

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' $\rightarrow$ 0001) and decoding (0001 $\rightarrow$ '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

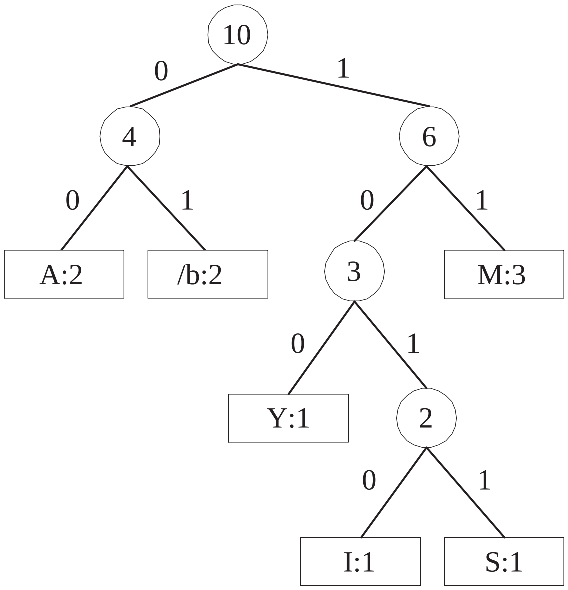
Huffman Code is a prefix code: no codeword is a prefix of any other.
(we are representing the space as "/b")

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.

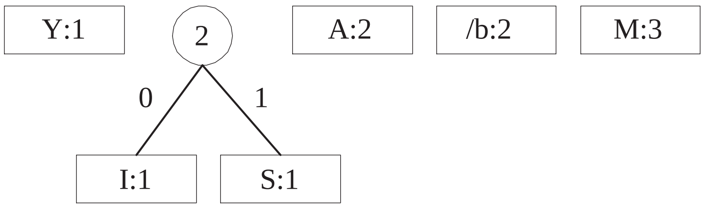
The algorithm in action

Using the letters and frequencies from the previous example:

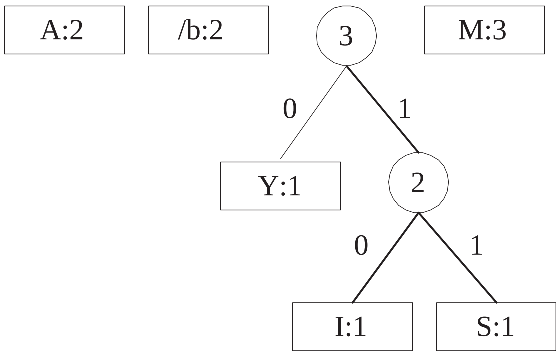
Subtrees:	I:1	S:1	Y:1	A:2	/b:2	M:3

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

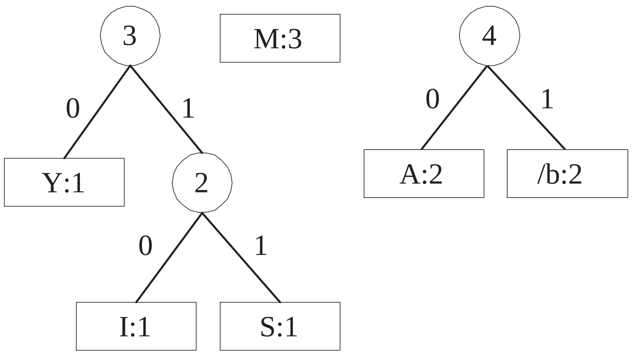
Subtrees:



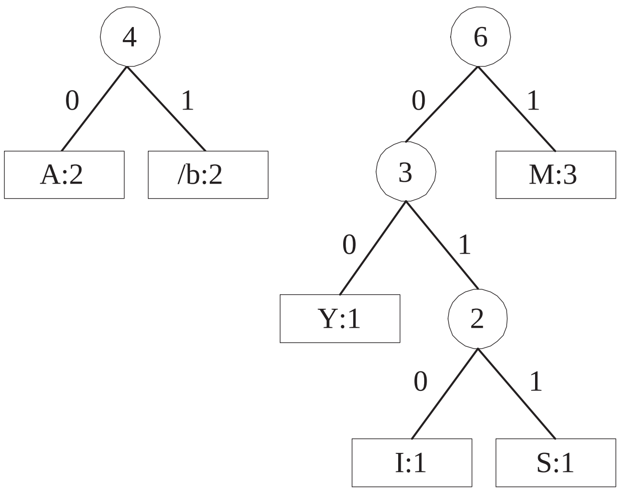
Subtrees:

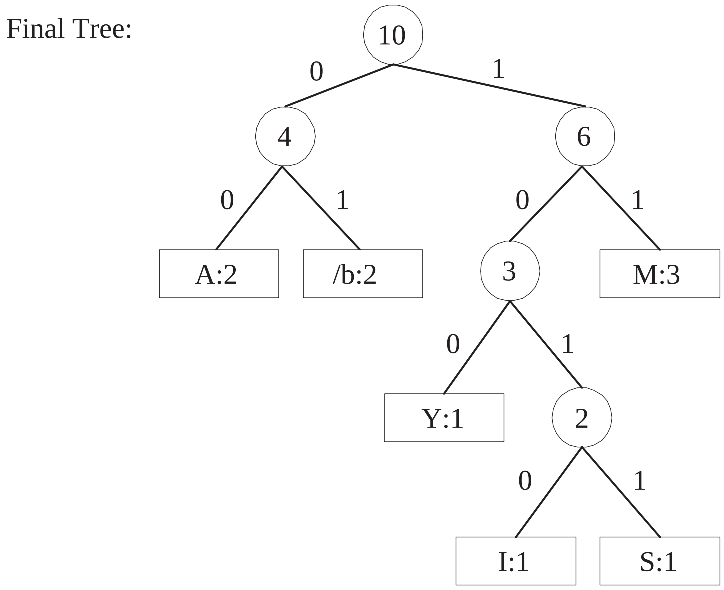


Subtrees:



Subtrees:




Final Tree:
10
0
1
4
6
0
1
0
1
A:2
/b:2
3
M:3
0
1
Y:1
2
0
1
I:1
S:1

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:

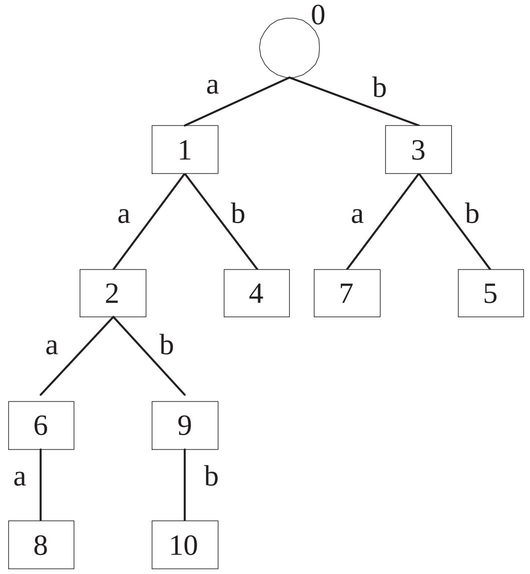
```
Index : 0 1 2  3 4  5  6   7  8    9   10
        0|a|aa|b|ab|bb|aaa|ba|aaaa|aab|aabb
```

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

Edwards|M|	Wu|F|	*-1|10|	Smith|M|	*15|30|
0	1	2	3	4
10 bytes	5 bytes	10 bytes	8 bytes	30 bytes

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)

Address of Record	
17	LON | 2312 | Symphony N.S | ...
62	RCA | 2626 | Quartet in C sharp | ...
117	WAR | 23699 | Adagio | ...
152	ANG | 3795 | Violin Concerto | ...

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: REVIEW LECTURE (NOT IN NOTES)

LECTURE 12: 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 13: 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 :

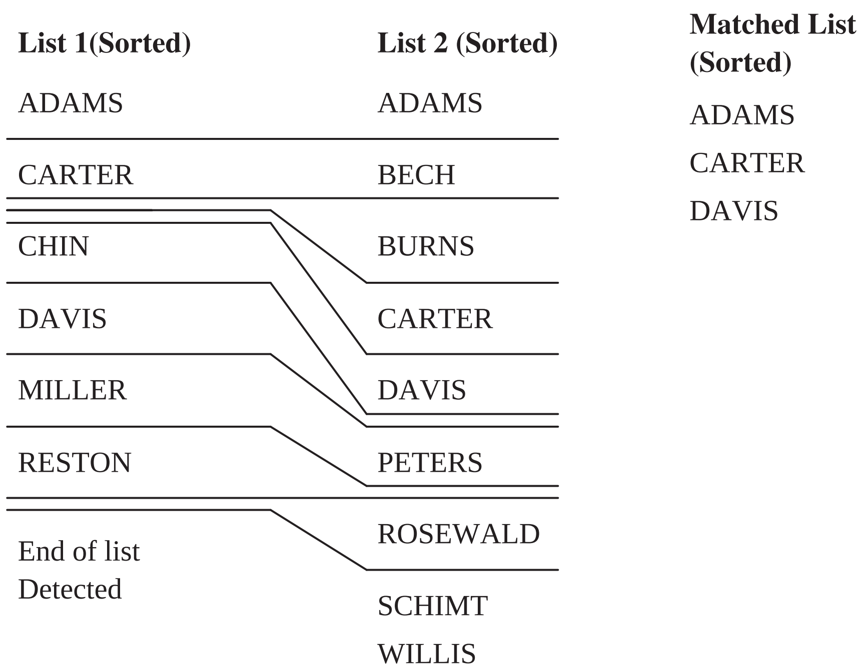
- Matching (intersection) of the items of the lists.
- Merging (union) of the items of the lists.

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

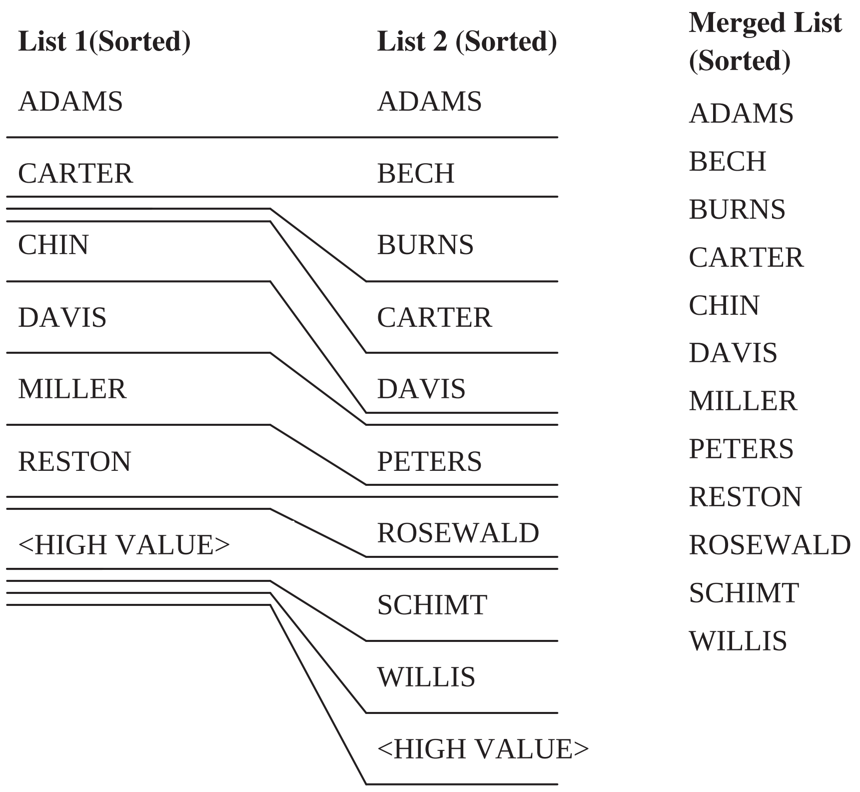


Synchronization :
`item(i)` = current item from list i

```
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.)



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 14: 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.)

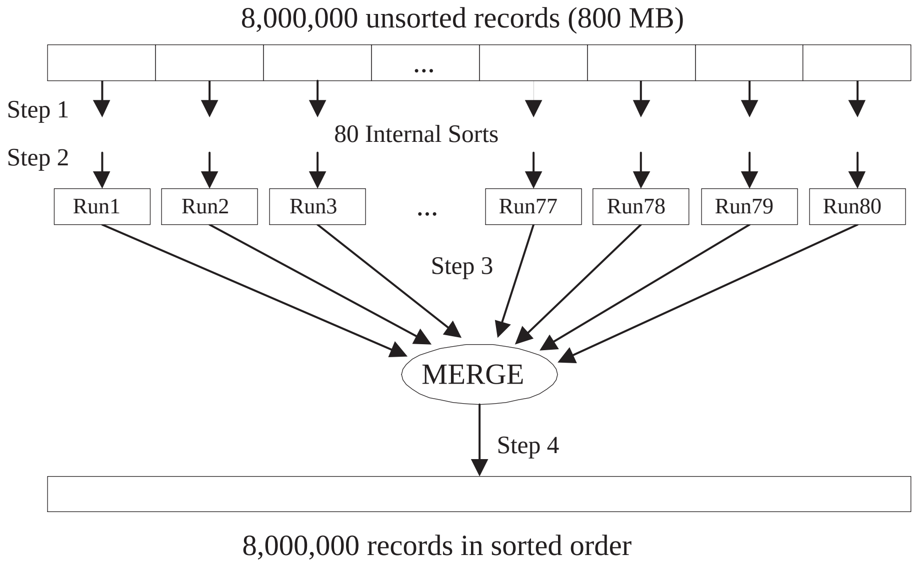
- Total file size = 800 MB
- Total number of bytes for all the keys = 80 MB
So, we cannot do internal sorting nor keysorting.

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.

LECTURE 15: HASHING I

Lecture 14 is skipped since it is a review lecture.

Contents of today's lecture:

- Introduction to Hashing
- Hash functions.
- Distribution of records among addresses, synonyms and collisions.
- Collision resolution by progressive overflow or linear probing.

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 11.1,11.2,11.3,11.5.

Motivation

Hashing is a useful searching technique, which can be used for implementing indexes. The main motivation for Hashing is improving searching time.

Below we show how the search time for Hashing compares to the one for other methods:

- Simple Indexes (using binary search): $O(\log_2 N)$
- B Trees and B+ trees (will see later): $O(\log_k N)$
- Hashing: $O(1)$

What is Hashing ?

The idea is to discover the location of a key by simply examining the key. For that we need to design a hash function.

A **Hash Function** is a function $h(k)$ that transforms a key into an address.
An address space is chosen before hand. For example, we may decide the file will have 1,000 available addresses.

If U is the set of all possible keys, the hash function is from U to {0,1,...,999}, that is

$$h : U \longrightarrow \{0, 1, ..., 999\}$$

Example :

k	**ASCII code for the first 2 letters**	`product`	**h(k) =** `product` **mod 1,000**
BALL	66, 65	$66 \times 65 = 4,290$	290
LOWELL	76, 79	$76 \times 79 = 6,004$	004
TREE	84, 82	$84 \times 82 = 6,888$	888

RRN	FILE
000	
001	
⋮	⋮
004	LOWELL
⋮	⋮
290	BALL
⋮	⋮
888	TREE
⋮	⋮
999	

There is no obvious connection between the key and the location (randomizing).

Two different keys may be sent to the same address generating a **Collision**.

Can you give an example of collision for the hash function in the previous example ?

Answer:
LOWELL, LOCK, OLIVER, and any word with first two letters L and O will be mapped to the same address:

$$h(\text{LOWELL}) = h(\text{LOCK}) = h(\text{OLIVER}) = 4.$$

These keys are called **synonyms**. The address “4” is said to be the **home address** of any of these keys.

Avoiding collisions is extremely difficult **(do you know the birthday paradox?)**, so we need techniques for dealing with it.

Ways of reducing collisions:

1. **Spread out the records** by choosing a good hash function.
2. **Use extra memory**, i.e. increase the size of the address space (Ex: reserve 5,000 available addresses rather than 1,000).
3. **Put more than one record at a single address** (use of buckets).

A Simple Hash Function

To compute this hash function, apply 3 steps :

Step 1: transform the key into a number.

```
LOWELL  =  | L | O | W | E | L | L |  |  |  |  |  |  |
ASCII code: 76  79  87  69  76  76  32 32 32 32 32 32
```

Step 2: fold and add (chop off pieces of the number and add them together) and take the mod by a prime number

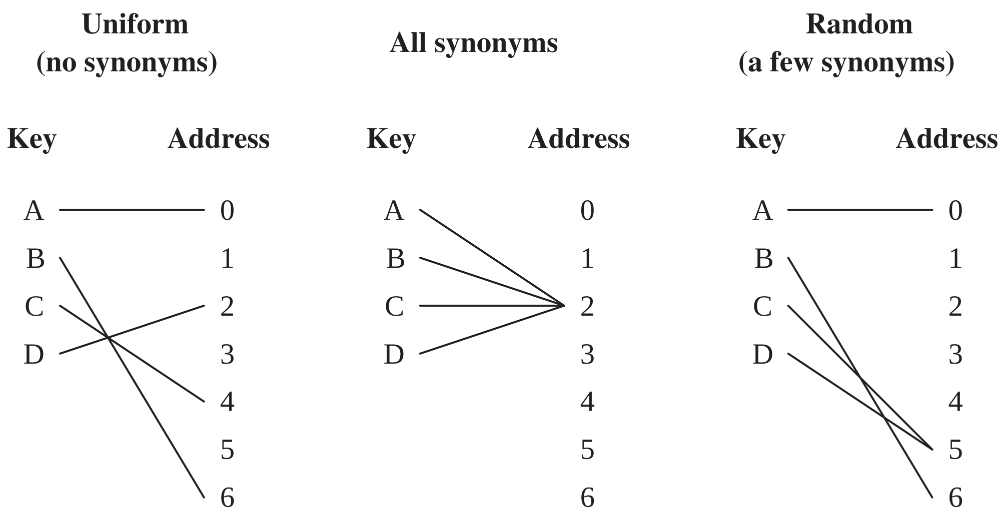
```
7679|8769|7676|3232|3232|3232|

7679+8769+7676+3232+3232+3232 = 33,820

33,820 mod 19937 = 13,883
```

Step 3: divide by the size of the address space (preferably a prime number).

```
13,883 mod 101 = 46
```

Distribution of Records among Addresses

There are 3 possibilities :



Uniform distributions are extremely rare.

Random distributions are acceptable and more easily obtainable.

Trying a **better-than-random** distribution, by preserving natural ordering among the keys :

- Examine keys for patterns.

  Ex: Numerical keys that are spread out naturally such as:
  keys are years between 1970 and 2000.
  $f(year) = (year - 1970) \bmod (2000 - 1970 + 1)$
  $f(1970) = 0, f(1971) = 1, \cdots, f(2000) = 30$

- Fold parts of the key.

  Folding means extracting digits from a key and adding the parts together as in the previous example.
  In some cases, this process may preserve the natural separation of keys, if there is a natural separation.

- Use prime number when dividing the key.

  Dividing by a number is good when there are sequences of consecutive numbers.

  If there are many different sequences of consecutive numbers, dividing by a number that has many small factors may result in lots of collisions. A prime number is a better choice.

When there is no natural separation between keys, try **randomization**.

You can using the following Hash functions:

- **Square the key and take the middle**:

  Ex: key = 453 $453^2 = 205209$
  Extract the middle = 52.
  This address is between 00 and 99.

- **Radix transformation**:

  Transform the number into another base and then divide by the maximum address.

  Ex: Addresses from 0 to 99
  key = 453 in base 11 : 382
  hash address = 382 mod 99 = 85.

Collision Resolution:Progressive Overflow

Progressive overflow/linear probing works as follows :

Insertion of key k:

- Go to the home address of k : h(k)
- If free, place the key there
- If occupied, try the next position until an empty position is found (the 'next' position for the last position is position 0, i.e. wrap around)

Example :

key k	**Home address - h(k)**
COLE	20
BATES	21
ADAMS	21
DEAN	22
EVANS	20

Complete Table:

0	
1	
2	
⋮	⋮
19	
20	
21	
22	

Table size = 23

Searching for key k:

- Go to the home address of k : h(k)
- If k is in home address, we are done.
- Otherwise try the next position until: key is found or empty space is found or home address is reached (in the last 2 cases, the key is not found)

0	DEAN
1	EVANS
2	
⋮	⋮
19	
20	COLE
21	BATES
22	ADAMS

Ex :
A search for 'EVANS' probes places : 20, 21, 22, 0, 1, finding the record at position 1.

Search for 'MOURA', if h(MOURA)=22, probes places 22, 0, 1, 2 where it concludes 'MOURA' in not in the table.

Search for 'SMITH', if h(SMITH)=19, probes 19, and concludes 'SMITH' in not in the table.

Advantage : Simplicity

Disadvantage : If there are lots of collisions, clusters of records can form, as in the previous example.

Search length

- Number of accesses required to retrieve a record.

`average search length`
`= (sum of search lengths)/(numb.of records)`

In the previous example :

0	DEAN
1	EVANS
2	
⋮	⋮
19	
20	COLE
21	BATES
22	ADAMS

key	**Search Length**
COLE	
BATES	
ADAMS	
DEAN	
EVANS	

Average search length = (1+1+2+2+5)/5 = 2.2.

Refer to figure 11.7 in page 489. It shows that a packing density up to 60% gives an average search length of 2 probes, but higher packing densities make search length to increase rapidly.

LECTURE 16: HASHING II

Contents of today's lecture:

- Predicting record distribution; packing density.
- Hashing with Buckets
- Implementation issues.

Reference: FOLK, ZOELLICK AND RICCARDI, File Structures, 1998. Sections 11.3,11.4,11.6

Predicting Record Distribution

Throughout this section we assume a random distribution for the hash function.

Let N = number of available addresses, and
r = number of records to be stored.

Let $p(x)$ be the probability that a given address will have x records assigned to it.

It is easy to see that

$$p(x) = \frac{r!}{(r-x)!x!} \left[1 - \frac{1}{N}\right]^{r-x} \left[\frac{1}{N}\right]^{x}$$

For N and r large enough this can be approximated by:

$$p(x) \sim \frac{(r/N)^x \, e^{-(r/N)}}{x!}$$

Example : $N = 1{,}000$, $r = 1{,}000$

$$p(0) \sim \frac{1^0\, e^{-1}}{0!} = 0.368$$

$$p(1) \sim \frac{1^1\, e^{-1}}{1!} = 0.368$$

$$p(2) \sim \frac{1^2\, e^{-1}}{2!} = 0.184$$

$$p(3) \sim \frac{1^3\, e^{-1}}{3!} = 0.061$$

For N addresses,
the expected number of addresses with x records is

$$N \cdot p(x).$$

Complete the numbers below for the example above:

expected # of addresses with 0 records assigned to it =
expected # of addresses with 1 records assigned to it =
expected # of addresses with 2 records assigned to it =
expected # of addresses with 3 records assigned to it =

Reducing Collision by using more Addresses

Now, we see how to reduce collisions by increasing the number of available addresses.

DEFINITION: `packing density` $= r/N$

500 records to be spread over 1000 addresses result in `packing density` $= 500/1000 = 0.5 = 50\%$.

Some questions :

1. How many addresses go unused ? *More precisely: What is the* **expected** *number of addresses with no key mapped to it?*

   $N \cdot p(0) = 1000 \cdot 0.607 = 607$

2. How many addresses have no synonyms ? *More precisely: What is the expected number of address with only one key mapped to it?*

   $N \cdot p(1) = 1000 \cdot 0.303 = 303$

3. How many addresses contain 2 or more synonyms ? *More precisely: What is the expected number of addresses with two or more keys mapped to it ?*

   $$N \cdot (p(2) + p(3) + ...) = N \cdot (1 - (p(0) + p(1)) = 1000 \cdot 0.09 = 90$$

4. Assuming that only one record can be assigned to an address, how many overflow records are expected ?

   $$1 \cdot N \cdot p(2) + 2 \cdot N \cdot p(3) + 3 \cdot N \cdot p(4) + ... =$$
   $$N \cdot [p(2) + 2 \cdot p(3) + 3 \cdot p(4) + ...] \sim 107.$$

   The justification for the above formula is that there is going to be $(i-1)$ overflow records for all the table positions that have i records mapped to it, which are expected to be as many as $N \cdot p(i)$.
   Now, there is a simpler formula derived by students from 2001:

   ```
   expected # of overflow records =
   = (#records) - (expected # of nonoverflow records)
   ```

   $$= r - (N \cdot p(1) + N \cdot p(2) + N \cdot p(3) + \ldots)$$
   $$= r - N \cdot (1 - p(0)) \quad \text{(since probabilities add up to 1)}$$
   $$= N \cdot p(0) - (N - r)$$

   ```
   =(expect. # of empty posit. for random hash function)
     - (# of empty positions for perfect hash function)
   ```

   Using this formula we get the same result as before:
   $N \cdot p(0) - (N - r) = 607 - 500 = 107$

5. What is the expected percentage of overflow records ?
   $107/500 = 0.214 = 21.4\%$

Note that using either formula, the percentage of overflow records depend only on the packing density ($PD = r/N$), and not on the individual values of N or r.
Indeed, using the formulas derived in 4., we get that the percentage of overflow records is:

$$\frac{r - N \cdot (1 - p(0))}{r} = 1 - \frac{1}{PD} \cdot (1 - p(0))$$

and the Poisson function that approximate $p(0)$ is a function of r/N which is equal to PD (for hashing without buckets).
So, hashing with packing density $PD = 50\%$ always yield 21% of records stored outside their home addresses.
Thus, we can compute the expected percentage of overflow records, given the packing density:

packing density %	% **overflow records**
10%	4.8%
20%	9.4%
30%	13.6%
40%	17.6%
50%	21.4%
60%	24.8%
70%	28.1%
80%	31.2%
90%	34.1%
100%	36.8%

Hashing with Buckets

This is a variation of hashed files in which more than one record/key is stored per hash address.

bucket = block of records corresponding to one address in the hash table.

The hash function gives the **Bucket Address**.

Example: for a bucket holding 3 records, insert the following keys:

key	**Home Address**
LOYD	34
KING	33
LAND	33
MARX	33
NUTT	33
PLUM	34
REES	34

0	
⋮	⋮
33	
34	

Effects of Buckets on Performance

We should slightly change some formulas:

$$\texttt{packing density} = \frac{r}{b \cdot N}$$

We will compare the following two alternatives:

1. Storing 750 data records into a hashed file with 1,000 addresses, each holding 1 record.

2. Storing 750 data records into a hashed file with 500 bucket addresses, each bucket holding 2 records.

- In both cases the packing density is 0.75 or 75%.
- In the first case r/N=0.75.
  In the second case r/N=1.50.

Estimating the probabilities as defined before:

	p(0)	**p(1)**	**p(2)**	**p(3)**	**p(4)**
1) r/N=0.75 (b=1)	0.472	0.354	0.133	0.033	0.006
2) r/N=1.50 (b=2)	0.223	0.335	0.251	0.126	0.047

Calculating the number of overflow records in each case:

1. **b=1** $(r/N{=}0.75)$:
   Number of overflow records =
   $= N \cdot [1 \cdot p(2) + 2 \cdot p(3) + 3 \cdot p(4) + \ldots]$
   $= r - N\,[p(1) + p(2) + p(3) + \ldots]$ (formula derived last class)
   $= r - N \cdot (1 - p(0))$
   $= 750 - 1000 \cdot (1 - 0.472) = 750 - 528 = 222.$
   This is about 29.6% overflow.

2. **b=2** $(r/N{=}1.5)$:
   Number of overflow records =
   $= N \cdot [1 \cdot p(3) + 2 \cdot p(4) + 3 \cdot p(5) + \ldots]$
   $= r - N \cdot p(1) - 2 \cdot N \cdot [p(2) + p(3) + \ldots]$ (formula for b=2)
   $= r - N \cdot [p(1) + 2[1 - p(0) - p(1)]$
   $= r - N \cdot [2 - 2 \cdot p(0) - p(1)]$
   $= 750 - 500 \cdot [2 - 2 \cdot (0.223) - 0.335] = 140.5 \cong 140.$
   This is about 18.7% overflow.

Indeed, the percentage of collisions for different bucket sizes is:

	Bucket Size				
Packing Density %	**1**	**2**	**5**	**10**	**100**
75%	29.6%	18.7%	8.6%	4.0%	0.0%

Refer to table 11.4 page 495 of the book to see the percentage of collisions for different packing densities and different bucket sizes.

Implementation Issues:

1. Bucket structure

   A Bucket should contain a counter that keeps track of the number of records stored in it. Empty slots in a bucket may be marked '//.../'.
   Ex: Bucket of size 3 holding 2 records:

2	JONES	/////////...///	ARNSWORTH

2. Initializing a file for hashing:

   - Decide on the **Logical Size** (number of available addresses) and on the number of buckets per address.
   - Create a file of empty buckets before storing records. An empty bucket will look like:

0	/////////...///	/////////...///	/////////...///

3. Loading a hash file:

   When inserting a key, remember to:
   - Wrap around when searching for available bucket.
   - Be careful with infinite loops when hash file is full.

LECTURE 17: HASHING III

Contents of today's lecture:

- Deletions in hashed files.
- Other collision resolution techniques:
  - double hashing,
  - chained progressive overflow,
  - chaining with separate overflow area,
  - scatter tables.
- Patterns of record access.

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 11.7,11.8,11.9

Making Deletions

Deletions in a hashed file have to be made with care.

Record	ADAMS	JONES	MORRIS	SMITH
Home Address	5	6	6	5

Hashed File using Progressive Overflow:

⋮	⋮
4	//////////
5	ADAMS
6	JONES
7	MORRIS
8	SMITH
⋮	⋮

Delete 'MORRIS'

If 'MORRIS' is simply erased, a search for 'SMITH' would be unsuccessful:

⋮	⋮	
4	//////////	← empty slot
5	ADAMS	
6	JONES	
7	//////////	← empty slot (WRONG: can't find 'SMITH' !!!)
8	SMITH	
⋮	⋮	

Search for 'SMITH' would go to home address (position 5) and when reached 7 it would conclude 'SMITH' is not in the file!

IDEA: use **TOMBSTONES**, i.e. replace deleted records with a marker indicating that a record once lived there:

⋮	⋮
4	//////////
5	ADAMS
6	JONES
7	####### ← tombstone (CORRECT: will find 'SMITH')
8	SMITH
⋮	⋮

A search must continue when it finds a tombstone, but can stop whenever an empty slot is found. A search for 'SMITH' will continue when if finds the tombstone in position 7 of the above table.

Note: Only insert a **tombstone** when the next record is occupied or is a tombstone. If the next record is an empty slot, we may mark the deleted record as empty. Why ?

Insertions should be modified to work with **tombstones**: if either an empty slot or a tombstone is reached, place the new record there.

Effects of Deletions and Additions on Performance

The presence of too many tombstones increases search length.

Solutions to the problem of deteriorating average search lengths:

1. Deletion algorithm may try to move records that follow a tombstone backwards towards its home address.
2. Complete reorganization: re-hashing.
3. Use a different type of collision resolution technique.

Other Collision Resolution Techniques

1) Double Hashing

- The first hash function determines the home address.
- If the home address is occupied, apply a second hash function to get a number c (c relatively prime to N).
- c is added to the home address to produce an overflow addresses; if occupied, proceed by adding c to the overflow address, until an empty spot is found.

Example:

k (key)	ADAMS	JONES	MORRIS	SMITH
$h_1(k)$ (home address)	5	6	6	5
$h_2(k) = c$	2	3	4	3

Hashed file using double hashing:

0	
1	
2	
3	
4	
5	ADAMS
6	JONES
7	
8	SMITH
9	
10	MORRIS

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

Suppose the above table is full, and that a key $\overline{k}$ has $h_1(\overline{k}) = 6$ and $h_2(\overline{k}) = 3$.

Question: What would be the order in which the addresses would be probed when trying to insert $\overline{k}$?

Answer: 6, 9, 1, 4, 7, 10, 2, 5, 8, 0, 3.

2) Chained Progressive Overflow

- Similar to progressive overflow, except that synonyms are linked together with pointers.
- The objective is to reduce the search length for records within clusters.

Example X: Search lengths:

Key	**Home**	**Progressive Overflow**	**Chained Progr. Overflow**
ADAMS	20	1	1
BATES	21	1	1
COLES	20	3	2
DEAN	21	3	2
EVANS	24	1	1
FLINT	20	6	3
Average Search Length :		2.5	1.7

Progressive Overflow

	data
⋮	⋮
20	ADAMS
21	BATES
22	COLES
23	DEAN
24	EVANS
25	FLINT
⋮	⋮

Chained Progressive Overflow

	data	**next**
⋮	⋮	⋮
20	ADAMS	22
21	BATES	23
22	COLES	25
23	DEAN	-1
24	EVANS	-1
25	FLINT	-1
⋮	⋮	⋮

PROBLEM: Suppose that 'DEAN' home address is 22. Since 'COLES' is there, we couldn't have a link to 'DEAN' starting in its home address!

Solution:
Two-pass loading:
- First pass: only load records that fit into their home addresses.
- Second pass: load all overflow records.

Care should be taken when deletions are done.

key	**home address**
ADAMS	20
BATES	21
COLES	20
DEAN	22
EVANS	24
FLINT	20

table after first pass:

20	ADAMS	-1
21	BATES	-1
22	DEAN	-1
23		
24	EVANS	-1
25		

table after second pass:

20	ADAMS	23
21	BATES	-1
22	DEAN	-1
23	COLES	25
24	EVANS	-1
25	FLINT	-1

3) Chaining with a Separate Overflow Area

Move overflow records to a **Separate Overflow Area**.

A linked list of synonyms start at their home address in the Primary data area, continuing in the separate overflow area.

Example X, with separate overflow area:

primary data area

20	ADAMS	0
21	BATES	1
22		
23		
24	EVANS	-1
25		

overflow area

0	COLES	2
1	DEAN	-1
2	FLINT	-1
3		
	⋮	⋮

When the packing density is higher than 1 an overflow area is **required**.

4) Scatter Tables: Indexing Revisited

Similar to chaining with separate overflow, but the hashed file contains no records, but only pointers to data records. The scatter

Example X organized as scatter table:

index (hashed)

	⋮
20	0
21	1
22	
23	
24	4
	⋮

datafile (entry-sequenced, sorted, etc.)

	data	next
0	ADAMS	2
1	BATES	3
2	COLES	5
3	DEAN	-1
4	EVANS	-1
5	FLINT	-1

Note that the data file can be organized in many different ways: sorted file, entry sequenced file, etc.

Patterns of Record Access

Twenty percent of the students send 80 percent of the e-mails. *L. M.*

Using knowledge of pattern of record access to improve performance ...

Suppose you know that 80% of the searches occur in 20% of the items.

How to use this info to try to reduce search length in a hashed file ?

- Keep track of record access for a period of time (say 1 month).
- Sort the file in descending order of access.
- Re-hash using this order.

Records more frequently searched are more likely to be **at** or **close to** their home addresses.

LECTURE 18: EXTENDIBLE HASHING I

Contents of today's lecture:

- What is extendible hashing.
- Insertions in extendible hashing.

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 12.1,12.2,12.3(overview only).

What is extendible hashing ?

- It is an approach that tries to make hashing **dynamic**, i.e. to allow insertions and deletions to occur without resulting in poor performance after many of these operations.
  Why this is not the case for ordinary hashing?
- Extendible hashing combines two ingredients:
  hashing and **tries**.
  (tries are digital trees like the one used in Lempel-Ziv)
- Keys are placed into buckets, which are independent parts of a file in disk.
  Keys having a hashing address with the same prefix share the same bucket.
  A trie is used for fast access to the buckets. It uses a prefix of the hashing address in order to locate the desired bucket.

Tries and buckets

Consider the following grouping of keys into buckets, depending on the prefix of their hash addresses:

bucket:	this bucket contains keys with hash address with prefix:
A	0
B	10
C	11

Drawing of the trie that provides and index to buckets:

```
                           +-------+
          *------------->  |   A   |
   0                       +-------+


 *
                           +-------+
   1       0   *------->   |   B   |
                           +-------+
         *
                           +-------+
           1   *------->   |   C   |
                           +-------+
```

Note: You need to connect the parents to the children in the drawing of the trie above.

Directory structure and buckets

Representing the trie as a tree would take too much space. Instead, do the following:

1) Extend the tree to a complete binary tree:

```
            0  *----+
                    +--> +-------+
         *               |   A   |
    0               +--> +-------+
            1  *----+
   *
                         +-------+
    1       0  *-------> |   B   |
                         +-------+
         *
                         +-------+
            1  *-------> |   C   |
                         +-------+
```

2) Flatten the trie up into an array:

```
   +------+      +-------+
00 |      |------>|   A   |
   +------+   +-->+-------+
01 |      |---+
   +------+      +-------+
10 |      |------>|   B   |
   +------+      +-------+
11 |      |--+
   +------+  |   +-------+
             +--->|   C   |
                  +-------+
```

How to search in extendible hashing ?

Searching for a key:

- Calculate the hash address of the key
(note that no table size is specified, so we don't take "mod").
- Check how many bits are used in the directory (2 bits in the previous example). Call i this number of bits.
- Take the least significative i bits of the hash address (in reverse order). This gives an index of the directory.
- Using this index, go to the directory and find the bucket address where the record might be.

What makes it extendible?

So far we have not discussed how this approach can be dynamic, making the table expand or shrink as records are added or deleted. The dynamic aspects are handled by two mechanisms:

- Insertions and bucket splitting.
- Deletions and bucket combination.

Bucket splitting to handle overflow

Extendible hashing solves bucket overflow by splitting the bucket into two and if necessary increasing the directory size.
When the directory size increases it doubles its size a certain number of times.

Example:

To simplify matters, let us assume the keys are numbers and the hash function returns the number itself.
For instance, $h(20) = 20$.
Bucket size: 2

numbers	1	2	3	5	6	7
binary representation:	0001	0010	0011	0101	0110	0111

```
   +-----+       +-------+
00 |     |------>| 2, 6 |
   +-----+   +-->+-------+
01 |     |---+
   +-----+       +-------+
10 |     |------>| 1, 5 |
   +-----+       +-------+
11 |     |--+
   +-----+  |    +-------+
            +--->| 3, 7 |
                 +-------+
```

```
      +-----+       +-------+
   00 |     |------>|  2, 6 |
      +-----+   +-->+-------+
   01 |     |---+
      +-----+       +-------+
   10 |     |------>|  1, 5 |
      +-----+       +-------+
   11 |     |--+
      +-----+  |    +-------+
               +--->|  3, 7 |
                    +-------+
```

Insert key 4 into the above structure:

- Since `4` has binary representation 0100, it must go into the first bucket. Since it is full, it will get split.
- The splitting will separate `4` from `2,6` since the last two bits of `4` in reverse order are `00` and the ones of `2,6` are `01`.
- the directory is prepared to accommodate the splitting, so no doubling is needed.

```
                    +-------+
               +--->|   4   |
      +-----+  |    +-------+
   00 |     |--+    +-------+
      +-----+   +-->|  2, 6 |
   01 |     |---+   +-------+
      +-----+       +-------+
   10 |     |------>|  1, 5 |
      +-----+       +-------+
   11 |     |--+
      +-----+  |    +-------+
               +--->|  3, 7 |
                    +-------+
```

Bucket splitting with increase in directory size

```
   +-----+       +-------+
00 |     |------>|  2, 6 |
   +-----+   +-->+-------+
01 |     |---+
   +-----+       +-------+
10 |     |------>|  1, 5 |
   +-----+       +-------+
11 |     |--+
   +-----+  |    +-------+
            +--->|  3, 7 |
                 +-------+
```

Insert key `9` into the above structure:

- Since `9` has binary representation 1001, it must go into the bucket indexed by 10. Since it is full, it will get split.
- The splitting will separate `1,9` from `5` since the last two bits of `1,9` in reverse order are `100` and the ones of `5` are `101`.
- The directory is not prepared to accommodate the splitting, so doubling is needed. The doubling in the directory size will add an extra bit to the directory index.

Result of the addition of key 9 as described above:

```
    +-----+
000 |     |---+
    +-----+   +-->+-------+
001 |     |------>|  2, 6 |
    +-----+   +-->+-------+
010 |     |---|
    +-----+   |
011 |     |---+   +-------+
    +-----+    +->| 1, 9  |
100 |     |----+  +-------+
    +-----+       +-------+
101 |     |------>|   5   |
    +-----+       +-------+
110 |     |------>+-------+
    +-----+   +-->|  3,7  |
111 |     |---+   +-------+
    +-----+
```

Insertion Algorithm

- Calculate the hash function for the key and the key address in the current directory.
- Follow the directory address to find the bucket that should receive the key.
- Insert into the bucket, splitting it if necessary.
- If splitting took place, calculate i: the number of bits necessary to differentiate keys within this bucket. Double the directory as many times as needed to create a directory indexed by i bits.

Note: This algorithm does not work if such an i does not exist. That is, if there are too many keys with the same hash address ("too many" here meaning more than the size of a bucket).

Why?

Which are possible fixes?

Practice exercise

Insert the following keys, into an empty extendible hashing structure:

2,10,7,3,5,16,15,9

Show the structure after each insertion.

All the intermediate steps will be performed in lecture.
Final solution for your checking only:

```
000 and 001 point to bucket with key 16
010 and 011 point to bucket with keys 2,10
100 and 101 point to bucket with key 5,9
110 points to bucket with key 3
111 points to bucket with keys 7,15
```

LECTURE 19: EXTENDIBLE HASHING II

Contents of today's lecture:

- Insertions: a closer look at bucket splitting.
- Deletions in extendible hashing.
- Extendible hashing performance.

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 12.4,12.5.

Insertions: bucket splitting re-visited

In some cases, several splits are necessary in order to accommodate a new key to be inserted. Let us consider one such case.
Consider the following example:

```
    +-----+
000 |     |---+
    +-----+   +-->+-------+
001 |     |------>|  2,6  |  bucket A
    +-----+   +-->+-------+
010 |     |---|
    +-----+   |
011 |     |---+   +-------+
    +-----+    +->| 1, 9  |
100 |     |----+  +-------+
    +-----+       +-------+
101 |     |------>|   5   |
    +-----+       +-------+
110 |     |------>+-------+
    +-----+   +-->|  3,7  |
111 |     |---+   +-------+
    +-----+
```

Insert key 10
Bucket A must split.
Note that A has depth 1, since one digit (namely 0) determines whether a key should be placed in A.

The first split will create a new bucket A1, so that A and A1 have now depth 2.
With depth 2, two digits will be examined to determine whether a key goes to A (00*) or A1 (01*).
After redistribution, the structure becomes:

```
    +-----+       +-------+
000 |     |------>|(empty)|  bucket A
    +-----+  +--->+-------+
001 |     |--+
    +-----+       +-------+
010 |     |------>|  2,6  | bucket A1
    +-----+   +-->+-------+
011 |     |---+   +-------+
    +-----+    +->| 1, 9  |
100 |     |----+  +-------+
    +-----+       +-------+
101 |     |------>|   5   |
    +-----+       +-------+
110 |     |------>+-------+
    +-----+   +-->|  3,7  |
111 |     |---+   +-------+
    +-----+
```

A becomes empty and A1 get both keys.
After this split, the bucket pointed by 010, namely A1, is still full.
So, we must split A1.
Bucket A1 has depth 2. It will split into buckets A1 and A2, both with depth 3.

After redistribution, the structure becomes:

```
    +-----+      +-------+
000 |     |----->|(empty)|
    +-----+  +-->+-------+
001 |     |--+   +-------+
    +-----+   +->|   2   | bucket A1
    |     |---+  +-------+
010 |     |      +-------+
    +-----+   +->|   6   | bucket A2
011 |     |---+  +-------+
    +-----+      +-------+
100 |     |----->| 1, 9  |
    +-----+      +-------+
101 |     |---+  +-------+
    +-----+   +->|   5   |
110 |     |---+  +-------+
    +-----+   +->+-------+
111 |     |----->|  3,7  |
    +-----+      +-------+
```

Now the bucket pointed by 010 is not full, so that we can insert key 10 there.

After inserting key 10, the structure becomes:

```
    +-----+      +-------+
000 |     |------>|(empty)|
    +-----+  +--->+-------+
001 |     |--+    +-------+
    +-----+   +-->|  2,10 | bucket A1
    |     |---+   +-------+
010 |     |       +-------+
    +-----+   +-->|   6   | bucket A2
011 |     |---+   +-------+
    +-----+       +-------+
100 |     |------>| 1, 9  |
    +-----+       +-------+
101 |     |---+   +-------+
    +-----+   +-->|   5   |
110 |     |---+   +-------+
    +-----+   +-->+-------+
111 |     |------>|  3,7  |
    +-----+       +-------+
```

Insertion Algorithm

```
Insert (key) {

 indexKey = index in the directory where key
            should be placed
 Let A = bucket pointed by indexKey

 if (bucket A is not full) then
     Insert key into bucket A
 else  { // bucket is full so must split
   if (bucket depth is equal to directory depth) then
       { double directory size,
         re-adjusting bucket pointers;
       }
   split bucket A;
   Insert (key);  // recursive call to itself
   }
}
```

Deletions in Extendible Hashing

When we delete a key we may be able to **combine buckets**.

After the bucket combination, the directory may or may not be **collapsed**.

Combination and collapsing is a recursive process.

Bucket combination

A bucket may have a **buddy bucket**, that is a bucket that may be combined to it.
In the following example, which of the following buckets is such that when deleting a key from it, the bucket can be combined to another bucket?

```
                         bucket size = 2 records
    +-----+
000 |     | -> bucket 1 (1 record)
    +-----+
001 |     | -> bucket 2 (2 records)
    +-----+
010 |     |--+
    +-----+  +--> bucket 3 (1 record)
011 |     |--+
    +-----+
100 |     | -> bucket 4  (1 record)
    +-----+
101 |     | -> bucket 5  (2 record)
    +-----+
110 |     | -> bucket 6  (2 record)
    +-----+
111 |     | -> bucket 7  (2 record)
    +-----+
```

Answer: buckets 1&2, buckets 4&5
Why not buckets 3, 6, 7 ?

Buddy buckets

One bucket can only be combined with its **buddy bucket**.

A bucket and its buddy bucket must be distinct buckets pointed to by sibling nodes in the directory trie.

The index of the buddy bucket is obtained by switching the last bit of the bucket's index.

How to test if the bucket pointed out by index i has a buddy bucket?

```
i = 101
swap the last bit of i: iSwap = 100
if  (directory[i] != directory[iSwap]) then
     bucket pointed by direct[iSwap]
     is the buddy bucket
else there is no buddy bucket.
```

Directory collapsing

We can collapse a directory if every pair of "sibling" indexes point to the same bucket.

Check if we can collapse a directory as follows:

```
canColapse = false;
if (directory size is larger than 1) then {
    canColapse = true;
    for i=0 to (size/2 -1) do
      if (directory[2*i] != directory[2*i+1]
      then { canColapse = false;
             exit for;
           }
}
```

If `canColapse` is true then we can collapse the directory in the following way:

- Create a new directory with half the size
- Copy the pointers from positions 0 to 0, 2 to 1, . . . , (size-2) to (size/2 -1).

Deletion Algorithm

`Delete (key):`

1. Search for `key`.
2. If `key` not found, stop.
3. If `key` was found then remove `key` from its bucket `b` , in the following way:
   - physically remove `key` from bucket `b`
   - run `tryCombine(b)` in order to try to combine buckets and shrink directory.

Description of `tryCombine(b)`:

```
if (b has a buddy bucket c)  then
   if ( b.numkeys + c.numkeys) <= maxkeys {
       combine buckets b and c into bucket b.
       try to collapse the directory;
       if (directory was collapsed) then
          tryCombine(b);
       }
```

Deletion Example

Delete 5 from the following structure:

```
    +------+
000 |      |---+
    +------+   +-->+-------+
001 |      |------>|  2, 6 |
    +------+   +-->+-------+
010 |      |---|
    +------+   |
011 |      |---+   +-------+
    +------+    +->| 1, 9  |
100 |      |----+  +-------+
    +------+       +-------+
101 |      |------>|   5   |
    +------+       +-------+
110 |      |------>+-------+
    +------+   +-->|(empty)|
111 |      |---+   +-------+
    +------+
```

Search for 5 and physically remove it from its bucket `b`:

```
    +-----+
000 |     |---+
    +-----+   +-->+-------+
001 |     |------>|  2, 6 |
    +-----+   +-->+-------+
010 |     |---|
    +-----+   |
011 |     |---+   +-------+
    +-----+    +->| 1, 9  | <<<<< buddy bucket c
100 |     |----+  +-------+
    +-----+       +-------+
101 |     |------>|(empty)|  <<<< bucket b
    +-----+       +-------+
110 |     |------>+-------+
    +-----+   +-->|(empty)|
111 |     |---+   +-------+
    +-----+
```

Run `tryCombine(b)`.

Since `b` contains 0 keys and buddy bucket `c` contains 2 keys, buckets `b` and `c` can be combined...

After combining `c` and `b` we get:

```
    +-----+
000 |     |---+
    +-----+   +-->+-------+
001 |     |------>|  2, 6 |
    +-----+   +-->+-------+
010 |     |---|
    +-----+   |
011 |     |---+
    +-----+
100 |     |------>+-------+
    +-----+       |  1, 9 |  b = buckets b,c combined
101 |     |------>+-------+
    +-----+
110 |     |------>+-------+
    +-----+   +-->|(empty)|
111 |     |---+   +-------+
    +-----+
```

The directory can be collapsed...

The collapsed directory:

```
   +-----+       +-------+
00 |     |---+-->|  2, 6 |
   +-----+   |   +-------+
01 |     |---+
   +-----+       +-------+
10 |     |------>|  1, 9 | bucket b
   +-----+       +-------+
11 |     |------>+-------+
   +-----+       |(empty)| buddy bucket c
                 +-------+
```

The recursive call to `tryCombine(b)` now combines `b` and its buddy bucket `c`:

```
   +-----+       +-------+
00 |     |---+-->|  2, 6 |
   +-----+   |   +-------+
01 |     |---+
   +-----+       +-------+
10 |     |---+-->|  1, 9 | b = buckets b, c combined
   +-----+   |   +-------+
11 |     |---+
   +-----+
```

The directory can be collapsed again...

The collapsed directory:

```
  +-----+      +-------+
0 |     |---+-->|  2, 6 | buddy bucket c
  +-----+      +-------+
1 |     |--+   +-------+
  +-----+  +--->|  1, 9 | bucket b
               +-------+
```

The next recursive call to `tryCombine(b)` would detect that `b` cannot be combined with its buddy bucket `c`.

The deletion operation has been completed.

Extendible Hashing Performance

Time Performance (Worst Case):

operation	directory kept in main memory	directory kept in disk
search	1 disk access	2 disk accesses
insertion d= dir size after insertion	$O(\log d)$ disk accesses	$O(d)$ disk accesses
deletion d= dir size before deletion	$O(\log d)$ disk accesses	$O(d)$ disk accesses

The great advantage of extended hashing is that its search time is truly $O(1)$, independently from the file size.

In ordinary hashing, this complexity depends on the packing density, which could change after many insertions.

Space Performance:

1. **Space Utilization for Buckets**

   Here we describe results found by Fagin, Nievergelt, Pippinger and Strong (1979).

   Experiments and algorithm analysis by these authors have shown that the packing density for extendible hashing (space utilization for buckets) fluctuates between 0.53 and 0.93.

   They have also shown that, if:

   r = number of records
   b = block size
   N = average number of buckets

   Then,

   $$N \approx \frac{r}{b \cdot ln2}.$$

   Therefore,

   packing density $= \frac{r}{b \cdot N} \approx ln2 \approx 0.69.$

   So, we expect the average bucket utilization to be 69%.

2. **Space Utilization for the Directory**

Given r keys spread over some buckets, what is the expected size of the directory, assuming random keys are inserted into the table?

Flajolet (1983) addressed this problem by doing a careful analysis, in order to estimate the directory size.

The following table shows his findings:

b	5	10	20	...	200
r					
10^3	1.50 K	0.30 K	0.10 K		0.00 K
10^4	25.60 K	4.80 K	1.70 K		0.00 K
10^5	424.10 K	68.20 K	16.80 K		1.00 K
10^6	6.90 M	1.02 M	0.26 M		8.10 K
10^7	112.11 M	12.64 M	2.25 M		0.13 M

1 K = 10^3, 1 M = 10^6. (From Flajolet 1983)

LECTURE 20: B-TREES I

Contents of today's lecture:

- Introduction to multilevel indexing and B-trees.
- Insertions in B trees.

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 9.1-9.6.

Introduction to Multilevel Indexing and B-Trees

Problems with **simple indexes** that are kept in disk:

1. Seeking the index is still slow (binary searching):

   We don't want more than 3 or 4 seeks for a search.
   So, here $\log_2(N+1)$ is still slow:

N	$\mathbf{\log_2(N+1)}$
15 keys	4
1,000	$\sim$10
100,000	$\sim$17
1,000,000	$\sim$20

2. Insertions and deletions should be as fast as searches:
   In simple indexes, insertion or deletion take O(n) disk accesses (since index should be kept sorted)

Indexing with Binary Search Trees

We could use **balanced** binary search trees:

- **AVL Trees**

  Worst-case search is 1.44 $\log_2$(N+2)

  1,000,000 keys → 29 levels

  Still prohibitive...

- **Paged Binary Trees**

  Place subtrees of size K in a single page

  Worst-case search is $\log_{K+1}$(N+1)

K=511, N=134,217,727

Binary trees: 27 seeks
Paged binary tree: 3 seeks

This is good but there are lots of difficulties in **maintaining** (doing insertions and deletions in) a paged binary tree.

Multilevel Indexing

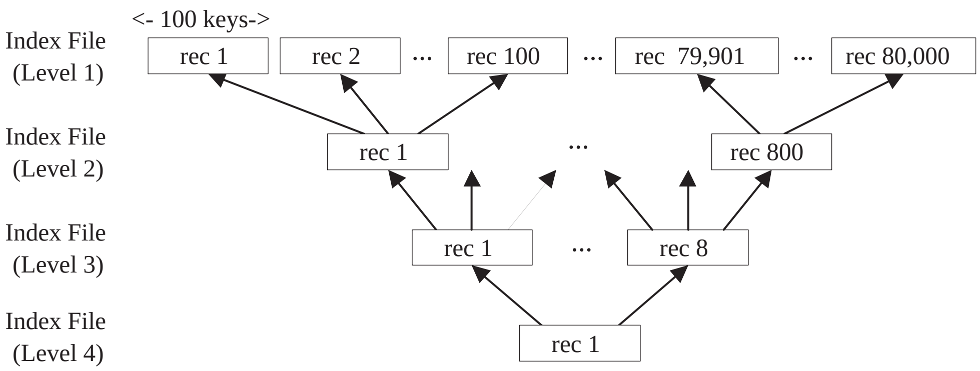
Consider our 8,000,000 example with keysize = 10 bytes.
Index file size = 80 MB
Each **record** in the **index** will contain 100 pairs (key, reference)
A **simple index** would contain: 80,000 records. Too expensive to search (still $\sim$ 16 seeks)

Multilevel Index

Build an index of an index file:

How:

- Build a simple index for the file, sorting keys using the method for external sorting previously studied.
- Build an index for this index.
- Build another index for the previous index, and so on.

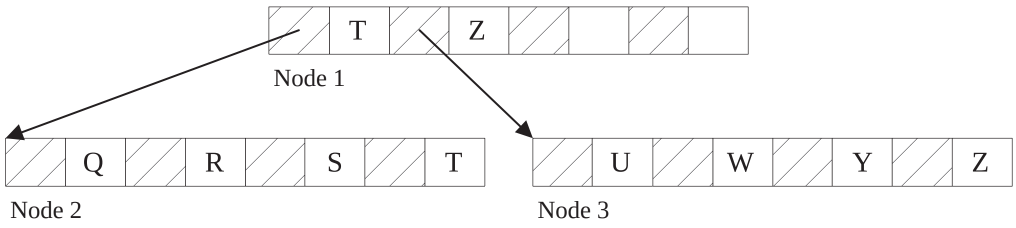


Note: That the index of an index stores the largest key in the record it is pointing to.

B-Trees - Working Bottom-Up

- Again an index record may contain 100 keys.
- An index record may be half full (each index record may have from 50 to 100 keys).
- When insertion in an index record causes it to overfull:
  - Split record in two
  - "Promote" the largest key in one of the records to the upper level

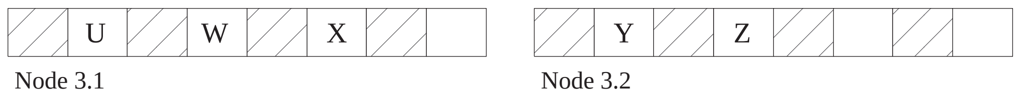
Example for order = 4 (instead of 100).



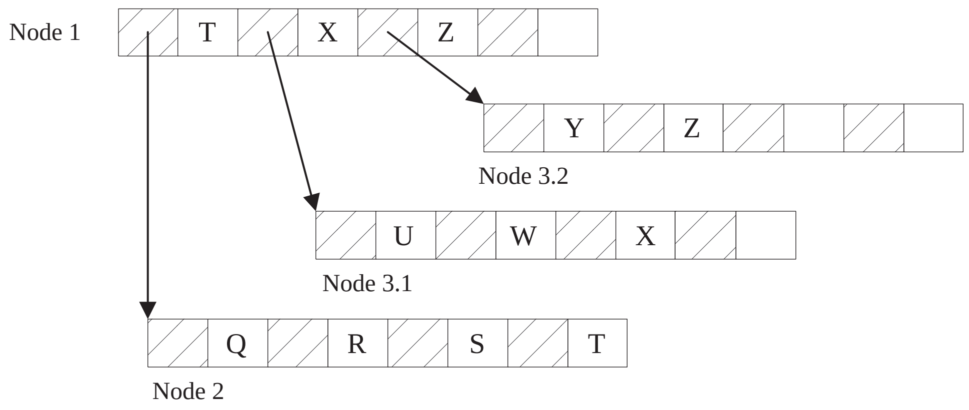
Inserting X

X is between T and Z: insertion in node 3 splits it and generates a promotion of node X.

Spliting :



Promoting largest of Node 3.1.

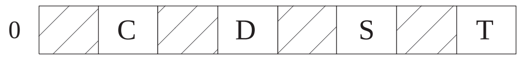


Important: If Node 1 was full, this would generate a new split-promotion of Node 1. This could be propagated up to the root.

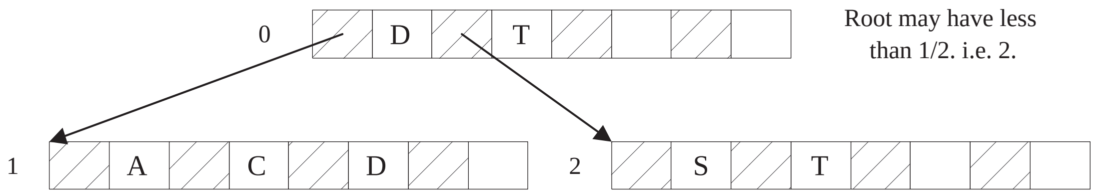
An example showing insertions:

Inserting keys: order = 4

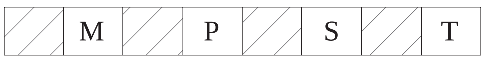
C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J, Y, Q, Z, F, X, V



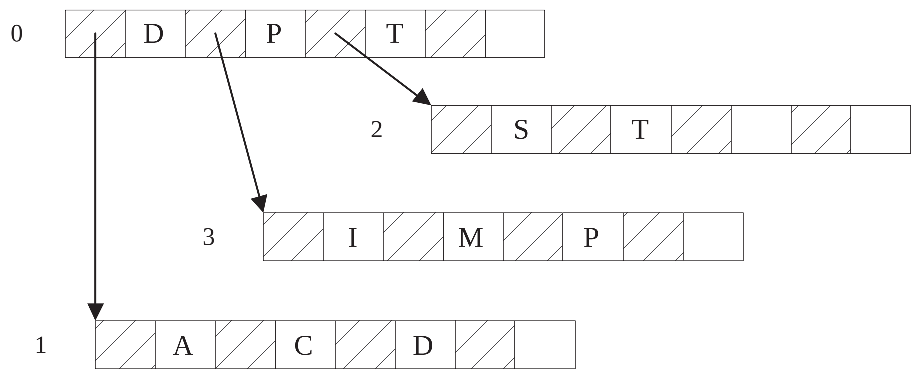
Inserting A: Split and promotion



Inserting M,P only changes Node 2 to :



But inserting “I” Splits and promotes “P”.



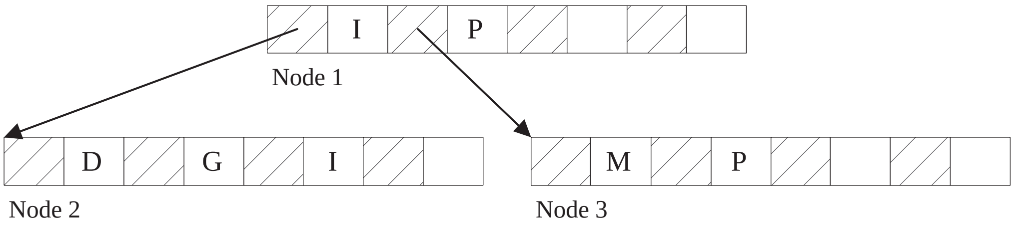
There is room for one more node at level 2. After that the root may get split!

Complete the above example as an exercise.

Important: At each level, no more than 2 nodes are affected. We got search and updates with cost equal to the height of the tree !!!

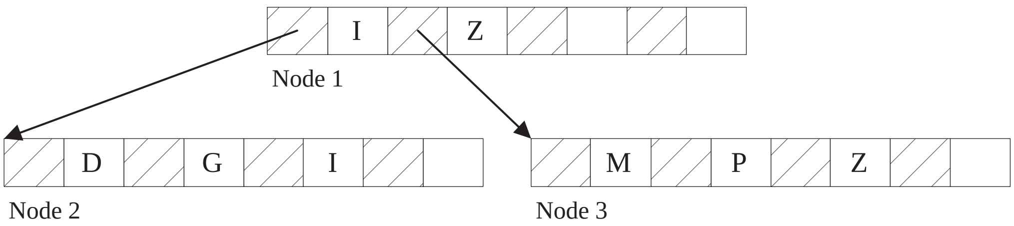
Note regarding insertions in B-trees

Special case of larger key:



Inserting Z

Z is larger than P but P is larger in Node 1, so the place for Z is Node 3.



B-Tree Properties

Properties of a B-tree of order m:

1. Every node has a maximum of m children.
2. Every node, except for the root and the leaves, has at least $\lceil m/2 \rceil$ children.
3. The root has at least two children (unless it is a leaf).
4. All the leafs appear on the same level.
5. The leaf level forms a complete index of the associated data file.

Worst-case search depth

The worst-case depth occurs when every node has the minimum number of children.

Level	Minimum number of keys (children)
1 (root)	2
2	$2 \cdot \lceil m/2 \rceil$
3	$2 \cdot \lceil m/2 \rceil \cdot \lceil m/2 \rceil = 2 \cdot \lceil m/2 \rceil^2$
4	$2 \cdot \lceil m/2 \rceil^3$
...	...
d	$2 \cdot \lceil m/2 \rceil^{d-1}$

If we have N keys in the leaves:

$$N \geq 2 \cdot \lceil m/2 \rceil^{d-1}$$

So, $d \leq 1 + \log_{m/2}(N/2)$

For $N = 1,000,000$ and order $m = 512$, we have

$$d \leq 1 + \log_{256} 500,000$$
$$d \leq 3.37$$

There is at most 3 levels in a B-tree of order 512 holding $1,000,000$ keys.

LECTURE 21: B-TREES II

Contents of today's lecture:

- Outline of `Search` and `Insert` algorihtms
- Deletions in B trees.

Reference: Folk, Zoellick and Riccardi, File Structures, 1998. Sections 9.8, 9.9, 9.10, 9.11, 9.12

Outline of Search and Insert algorithms

`Search (keytype key)`

1) Find leaf: find the leaf that could contain `key`, loading all the nodes in the path from root to leaf into an array in main memory.
2) Search for `key` in the leaf which was loaded in main memory.

`Insert (keytype key, int datarec_address)`

1) Find leaf (as above).

2) Handle special case of new largest key in tree: update largest key in all nodes that have been loaded into main memory and save them to disk.

3) Insertion, overflow detection and splitting on the update path:
   `currentnode` = leaf found in step 1)
   `recaddress` = `datarec_address`

   3.1) Insert the pair (`keys, recaddress`) into `currentnode`.

   3.2) If it caused overflow

   - Create `newnode`
   - Split contents between `newnode, currentnode`
   - Store `newnode, currentnode` in disk
   - If no parent node (root), go to step 4)
   - `currentnode` becomes parent node
   - `recaddress` = address in disk of `newnode`
   - `key` = largest key in new node
   - Go back to 3.1)

4) Creation of a new root if the current root was split.

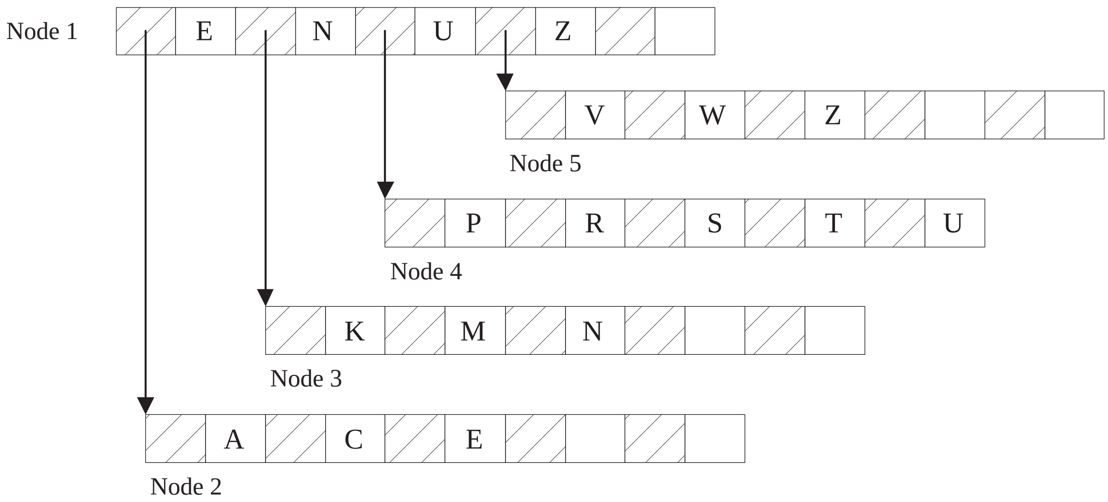
   Create root node pointing to `newnode` and `currentnode`. Save new root to disk.

Deletions in B-Trees

The rules for deleting a key K from a node n in a B-tree:

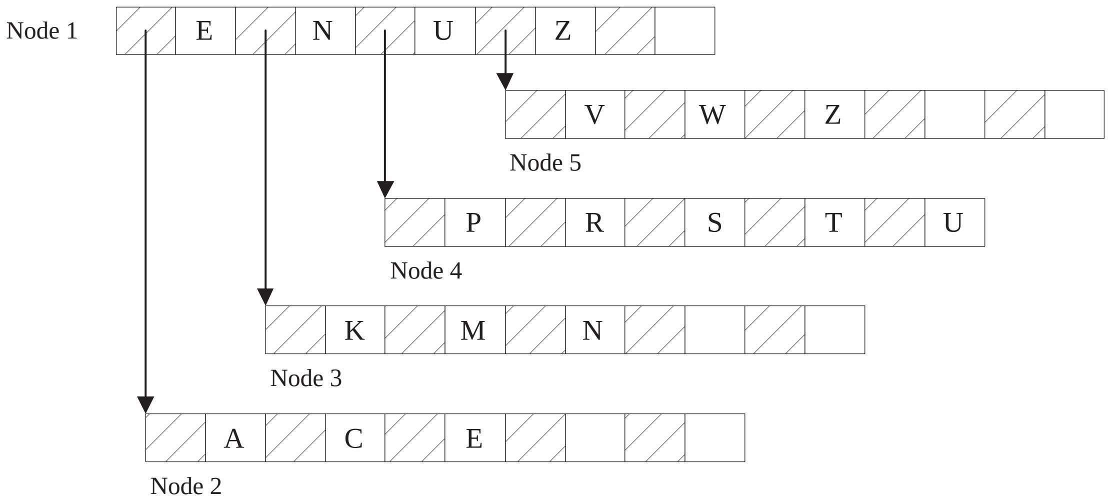
1. If n has more than the minimum number of keys and K is not the largest key in n, simply delete K from n.
2. If n has more than the minimum number of keys and K is the largest key in n, delete K from n and modify the higher level indexes to reflect the new largest key in n.
3. If n has exactly the minimum number of keys and one of the siblings has "few enough keys", **merge** n with its sibling and delete a key from the parent node.
4. If n has exactly the minimum number of keys and one of the siblings has extra keys, **redistribute** by moving some keys from a sibling to n, and modify higher levels to reflect the new largest keys in the affected nodes.

Consider the following example of a B-tree of **order 5** (minimum allowed in node is 3 keys)

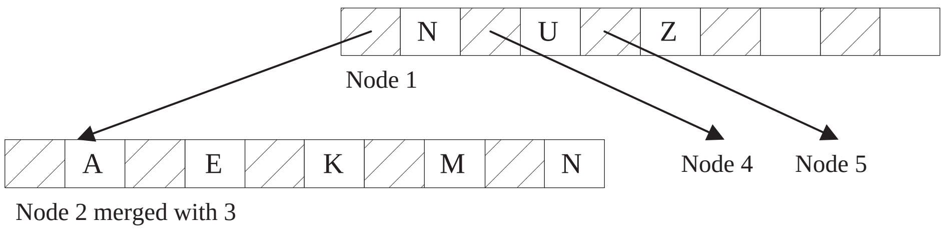


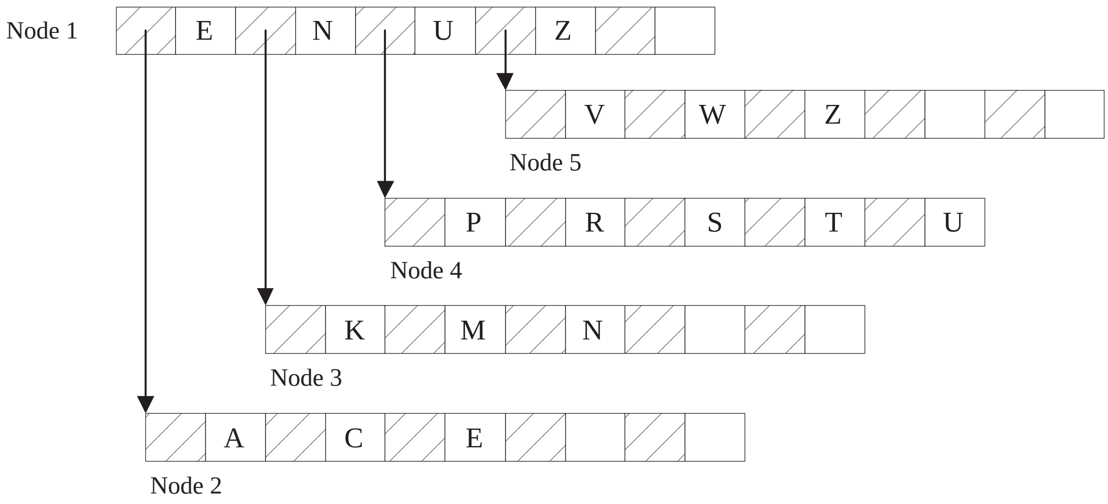
We consider the following 5 alternative modifications on the previous tree:

- Deleting "T" falls into case 1)
- Deleting "U" falls into case 2)

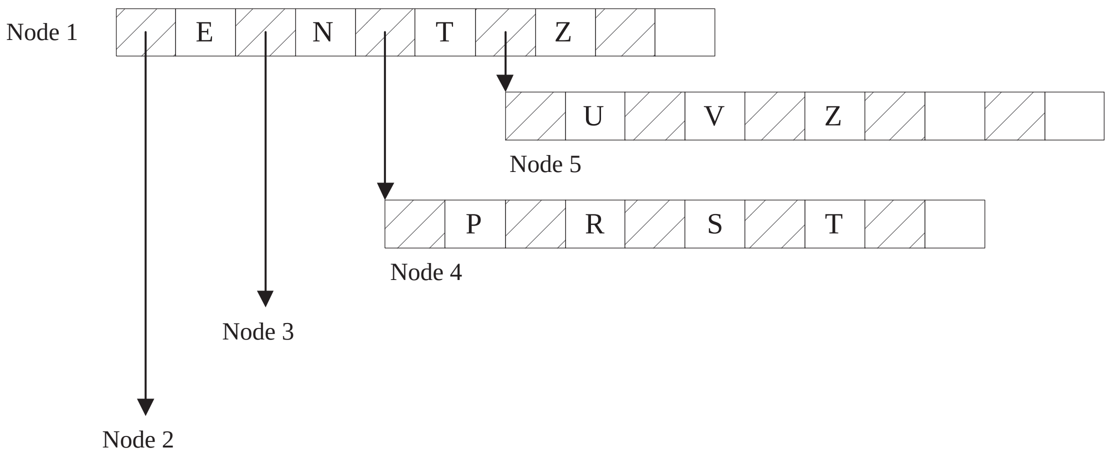


- Deleting "C" falls into case 3) merging with sibling:





- Deleting “W” falls into case 4) redistribution with sibling:



- Deleting “M” allows for two possibilities: case 3) or 4)
  - Merge Node 3 with Node 2; or
  - Redistribute keys between Node 3 and Node 4

Note that “sibling” here refers only to nodes that have the same parent and are **next to each other**.

LECTURE 22: B+ TREES I

Contents of today's lecture:

- Maintaining a sequence set.
- A simple prefix B+ tree.

Reference : Folk, Zoellick and Riccardi, File Structures, 1998. Sections 10.1 - 10.5

Motivation

Some applications require two views of a file :

Indexed view :	**Sequential view :**
Records are indexed by a key	Records can be sequentially accessed in order by key
Direct, indexed access	Sequential access (physically contiguous records)
Interactive, random access	Batch processing (Ex: co-sequential processing)

Example of applications

- Student record system in a university :
  - Indexed view : access to individual records
  - Sequential view : batch processing when posting grades or when fees are paid
- Credit card system :
  - Indexed view : interactive check of accounts
  - Sequential view : batch processing of payment slips

We will look at the following two aspects of the problem :

1. Maintaining a **sequence set** : keeping records in sequential order
2. Adding an **index set** to the sequence set

Maintaining a Sequence Set

Sorting and re-organizing after insertions and deletions is out of question.
We organize the sequence set in the following way:

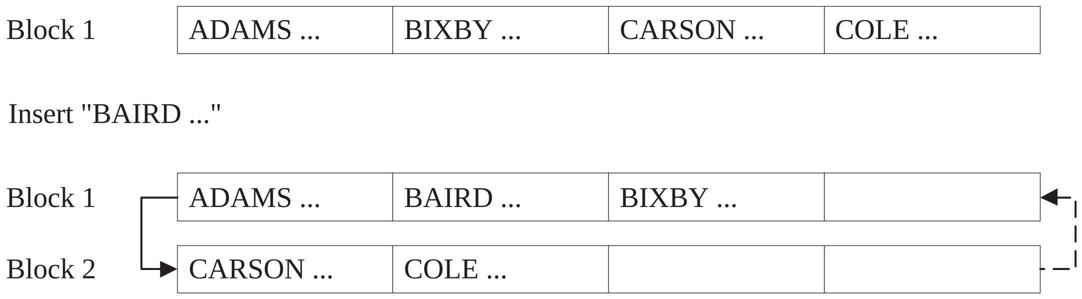
- Records are grouped in **blocks**
- Blocks should be **at least half full**.
- **Link fields** are used to point to the preceding block and the following block (similarly to doubly linked lists)
- Changes (insertion/deletion) are localized into blocks by performing :
  - **Block Splitting** when **insertion** causes overflow
  - **Block Merging** or **Redistribution** when **deletion** causes underflow

Example:

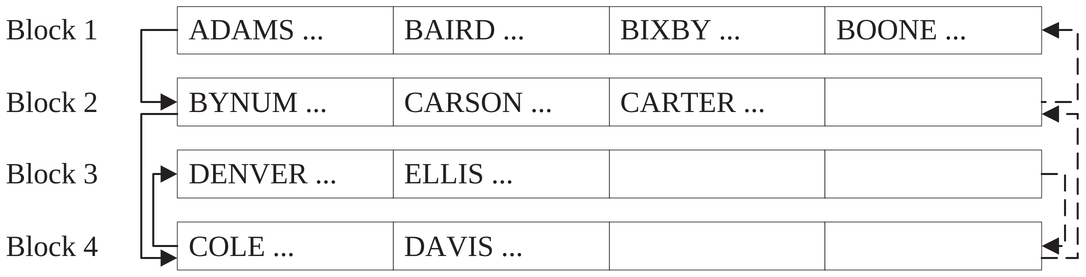
Block size $= 4$

key : **Last Name**

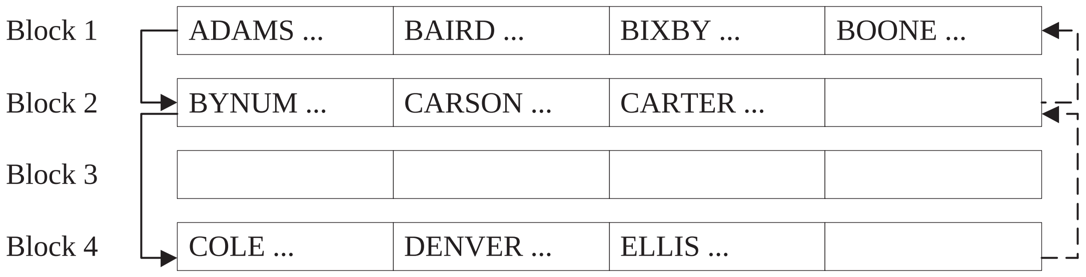
• Insertion with overflow:



- **Deletion with merging:**



Delete "DAVIS ..." (Merging)



Block 3 is available for re-use

Delete ‘BYNUM’:
Just remove it from Block 2
Then, delete ‘CARTER’ :
We can either merge Block 2 and 4 or redistribute records among Blocks 1 and 2.

- **Deletion with redistribution:**

When previous and next blocks are full then redistribution is the only option.

Advantages and disadvantages of the scheme described

Advantages:

- No need to re-organize the whole file after insertions/deletions.

Disadvantages:

- File takes more space than unblocked files (since blocks may be half full).
- The order of the records is not necessarily **physically** sequential (we only guarantee physical sequentiality within a block).

Choosing Block Size

Consider :

- Main memory constraints (must hold at least 2 blocks)
- Avoid seeking within a block (Ex: in sector formatted disks choose block size equal to cluster size).

Adding an Index Set to the Sequential Set

Index will Contain Separators Instead of Keys

Choose the **Shortest Separator** (a prefix)

```
Block     Range of Keys  Separator
-----     -------------  ---------
  1       ADAMS - BERNE
                         BO
  2       BOLEN - CAGE
                         CAM
  3       CAMP  - DUTTON
                         E
  4       EMBRY - EVANS
                         F
  5       FABER - FOLK
                         FOLKS
  6       FOLKS - GADDIS
```

How can we find a separator for `key1`= "CAGE" and `key2` ="CAMP"?

Find the smallest prefix of `key2` that is not a prefix of `key1`.
In this example, the separator is "CAM".

The Simple Prefix B+ Tree

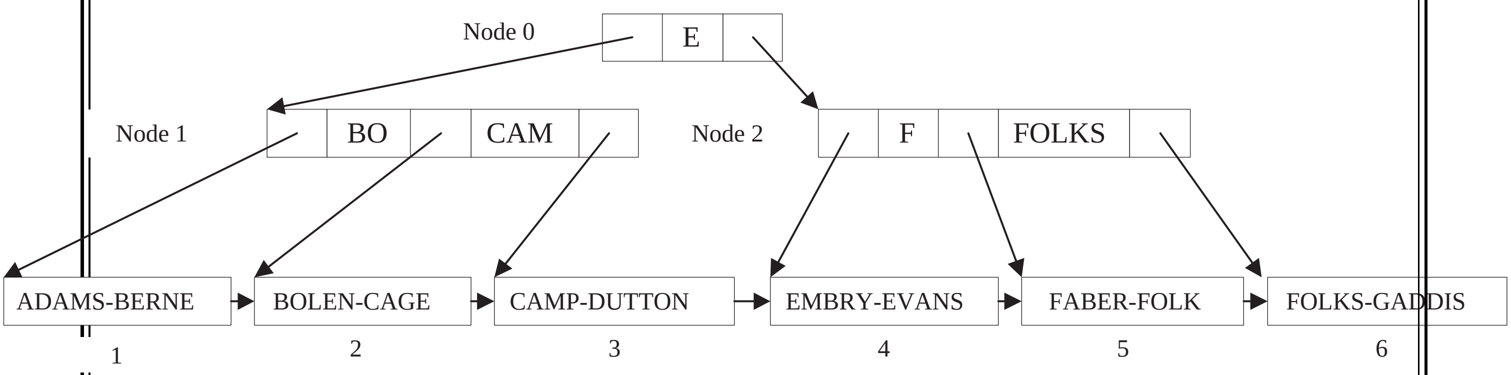
The **simple prefix B+ tree** consists of :

- **sequence set** (as previously seen).
- **index set**: similar to a B-tree index, but storing the shortest separators (prefixes) for the sequence set.

Note : If a node contains N separators, it will contain N+1 children. Using separators slightly modifies the operations in the B-tree index.

Example :

Order of the index set is 3 (i.e. maximum of 2 separators and 3 children). Note: The order is usually much larger, but we made it small for this example.



Search in a simple prefix B+ tree: Search for "EMBRY":

- Retrieve Node 0 (root).
- "EMBRY" > "E", so go right, and retrieve Node 2.
- Since "EMBRY" < "F" go left, and retrieve block number 4.
- Look for the record with key "EMBRY" in block number 4.

LECTURE 23: B+ TREES II

Contents of today's lecture:

- Simple Prefix B+ Tree Maintenance: Insertions and Deletions

Reference : Folk, Zoellick and Riccardi, File Structures, 1998. Sections 10.6 - 10.7 (overview 10.8 - 10.11).

From B trees to B+ trees

A clarification regarding the book's treatment of B-trees and B+ trees: the transition from understanding B trees to understanding the index set of the simple prefix B+ tree may be facilitated by the following interpretation.

- Look at the B + tree index set as a B-tree in which the smallest element in a child is stored at the parent node in order:

key1, pointer1, key2, pointer2, ..., keyN, pointerN.

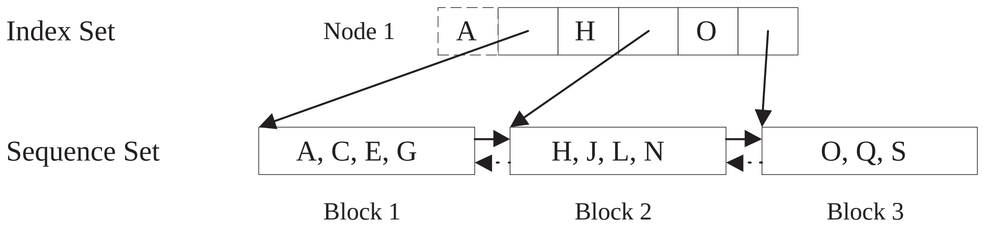
- Then, remember that each key is a separator. And remove key1 from the node's representation. It may help if you think that "key1" is still there, but is "invisible" for understanding purposes.

With this in mind, it should become clear that the updates on the B + tree index set are the same as in regular B-trees.

Simple Prefix B+ Tree Maintenance

Example:

- Sequence set has blocking factor 4
- Index set is a B tree of order 3

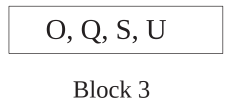


Note that "A" is not really there.

1. Changes which are local to single blocks in the sequence set

   Insert "U":

   - Go to the root
   - Go to the right of "O"
   - Insert "U" to block 3:
     The only modification is



   - There is no change in the index set

Delete "O":

- Go to the root
- Go to the right of "O"
- Delete "O" from block 3:
  The only modification is

Q, S, U

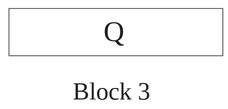
Block 3

There is no change in the index set: "O" is still a perfect separator for blocks 2 and 3.

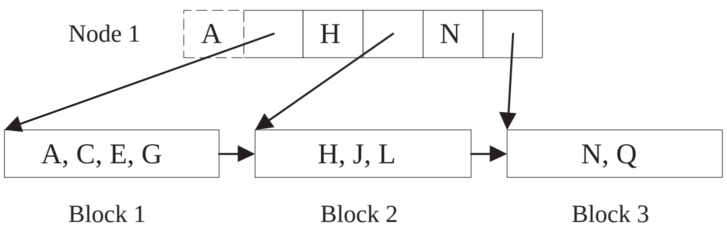
2. Changes involving multiple blocks in the sequence set.

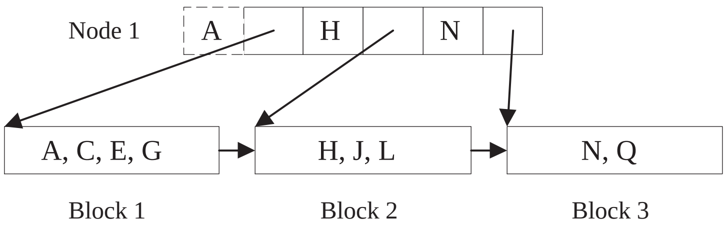
Delete “S” and “U”:

Now block 3 becomes less than 1/2 full (underflow)



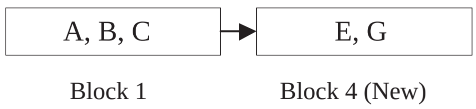
Since block 2 is full, the only option is **re-distribution** bringing a key from block 2 to block 3:
We must update the separator “O” to “N”.
The new tree becomes:



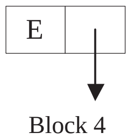


Insert "B":

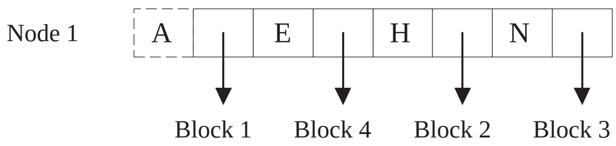
- Go to the root
- Go to the left of "H" to block 1
- Block 1 would have to hold A,B,C,E,G
- Block 1 is split:



Promote new separator "E" together with pointer to new block 4

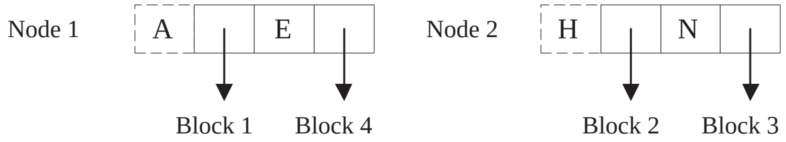


We wished to have:

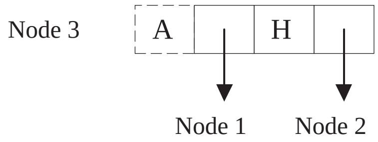


But the order of the index set is 3 (3 pointers, 2 keys).

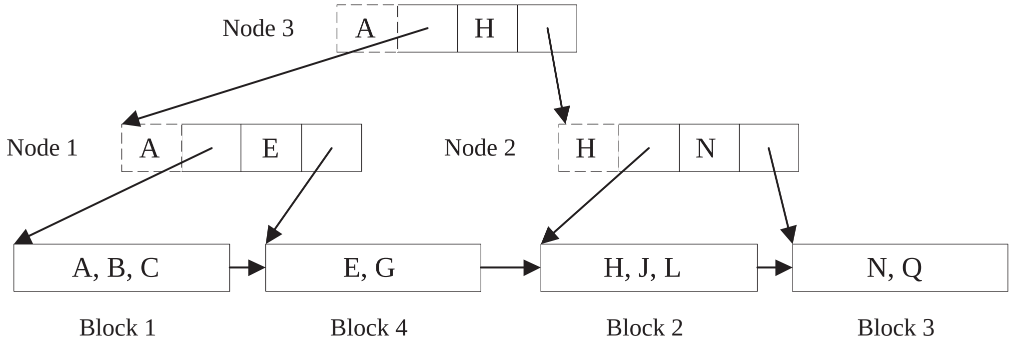
So this causes node to split:



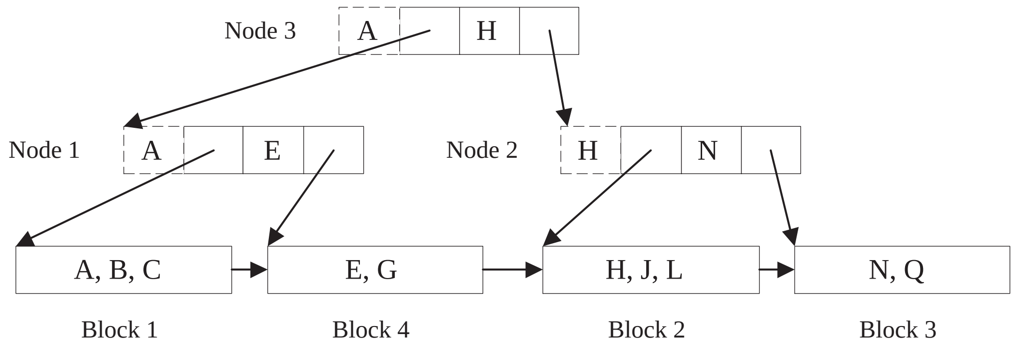
Create a new root to point to both nodes:



The new tree is:

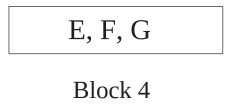


Remember that "A" is not really present in nodes 1 an 3 and that "H" is not really present in node 2.



Insert "F"

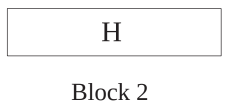
- Go to root
- Go to left of "H"
- Go to right of "E" in Node 1
- Insert "F" in block 4
- Block 4 becomes:



- Index set remains unchanged

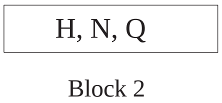
Delete "J" and "L"

Block 2 would become:

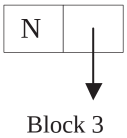


But this is an underflow.
One may get tempted to redistribute among blocks 4 and 2: E, F, G and H would become E, F and G, H.
Why this is not possible ?
Block 4 and block 2 are not siblings! They are cousins.

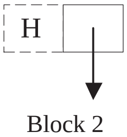
The only sibling of block 2 is block 3.
Redistribution in not possible between H and N,Q, so the only possibility is **merging** blocks 2 and 3 :



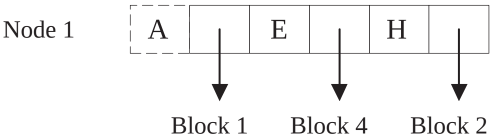
- Send block 3 to `AVAIL LIST`
- Remove the following from node 2



- This causes an underflow in Node 2

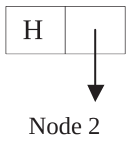


- The only possibility is a merge with its sibling (Node 1):

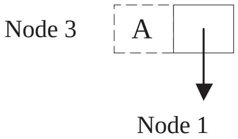


- In our interpretation “H” becomes “visible”; In the textbook’s interpretation “H” is brought down from the root.
- Send node 2 to `AVAIL LIST`

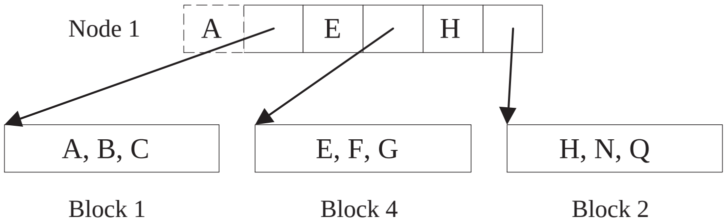
- Now remove



from the root (Node 3). This causes underflow on the root:



- Underflow on the root causes removal of the root (Node 3) and Node 1 becomes the new root :



Blocks were reunited as a big happy family again !!

Compare it with the original tree.

Note : Remember that a B+ tree may be taller, so that splittings or mergings of nodes may propagate for several levels up to the root.

Advanced Observations for Meditation

1. Usually a node for the index set has the same physical size (in bytes) than a block in the sequence set. In this case, we say "index set block" for a node.
2. Usually sequence set blocks and index set blocks are mingled inside the same file.
3. The fact that we use separators of variables length suggests the use of B trees of variable order. The concepts of underflow and overflow become more complex in this case.
4. Index set blocks may have a complex internal structure in order to store variable length separators and allow for binary search on them (see Figure 10.12 on page 442).
5. Building a B+ tree from an existing file already containing many records can be done more efficiently than doing a sequence of insertions into an initially empty B+ tree. This is discussed in Section 10.9 (Building a Simple Prefix B+ Tree).
6. Simple prefix B+ trees or regular B+ trees are very similar. The difference is that the latter stores actual keys rather than shortest separators or prefixes.

LECTURE 24: PRACTICE EXERCISES (NO NOTES)

LECTURE 25: COURSE OVERVIEW

Fundamental file processing operations

- open, close, read, write, seek (file as a stream of bytes)

Secondary Storage Devices and System Software

- how different secondary storage devices work (tapes, magnetic disks, CD-ROM)
- the role of different basic software and hardware in I/O (operating system (file manager), I/O processor, disk controller)
- buffering at the level of the system I/O buffers.

Logical view of files and file organization

- file as a collection of records (concepts of records, fields, keys)
- record and field structures
- sequential access; direct access (RRN, byte offset).

Organizing files for performance

- data compression
- reclaiming space in files: handling deletions, AVAIL LIST, etc.
- sorting and searching: internal sorting, binary searching, keysorting.

Cosequential processing

- main characteristics: **sequential access** to input and output files, and **co-ordinated access** of input files.
- main types of processing: matching (intersection) and merging (union).
- main types of application: the merging step in an external sorting method; posting of transactions to a master file.

Indexing

- Primary and secondary key indexes.
- Maintenance of indexed files; different index organizations.
- Inverted lists for secondary indexes.
- Alternative ways of organizing an index:
  - Simple index
    keeping index sorted by key (additions and deletions are expensive: about $O(n)$ disk accesses)
  - B trees and B+ trees
    improvement in time: searches, insertions and deletions in about $O(\log_k n)$ disk accesses, where k is the order of the tree and n is the number of records for B trees or the number of blocks of records for B+ trees.
  - Hashed index: searches, insertions and deletions in constant expected time, i.e. expected time $O(1)$, provided that the hash function disperses the keys well (approximately random).

B trees and B+ trees

- We started from the problem of maintenance of simple indexes.
- B trees: multi-level index that work from bottom up and guarantees searches and updates in about $O(\log_k n)$ disk accesses. More precisely, the worst case search time, insertion time and deletion time (in number of disk accesses) is in the worst-case proportional to the height of the B-tree. The maximum B tree height is $1 + \log_{\lceil k/2 \rceil} n/2$, where n is the number of keys and k is the order of the B tree.
- Then, we discussed the problem of having both an indexed and a sequential view of the file: B+ trees.
- B+ tree = sequence set + index set
  The index set is a B tree; the sequence set is like a doubly linked list of **blocks** of records.
- simple prefix B+ trees: B+ trees in which we store separators (short prefixes) rather than keys, in the index set.
- We learned B trees and B+ trees maintenance and operations.

Hashing and Extedible Hashing

- We have discussed **static hashing** techniques: expected time for access is $O(1)$ when when hash function is good and file doesn't change too much (not many deletions/additions).
- We have discussed: hash functions, record distribution and search length, the use of buckets, collision resolution techniques (progressive overflow, chained progressive overflow, chaining with a separate overflow area, scatter tables, patterns of record access).
- We have discussed **extendible hashing**. In extendible hashing, hashing is modified to become self-adjusting, allowing for the desired expected $O(1)$ access even when the file grows a lot; the address space changes after a lot of insertions or deletions. A good hash function is still required.
- If you cannot predict which one could be a good hash function for a dataset, it is preferable to use a B tree or B+ tree (e.g. general purpose data base management systems). If the hash function is not good, hashing may lead to an $O(n)$ performance (e.g. assignment 3), which is prohibitive, while B trees and B+ trees have a guaranteed worst case performance which is quite good.

What's next: Database Management Systems CSI3317

Different layers:

Database management systems
File systems
Physical storage devices

Each layer hides details of the lower level.
This course focused in **file processing**. In order to do file processing efficiently we studied some key issues concerning **physical storage devices**.
CSI3317 will focus on **database management systems**.

As the number of users and applications in an organization grows, file processing evolves into database processing.
A **database management system** is a large software that maintains the data of an organization and mediates between the data and the application programs.
Each application program asks the DBMS for data, the DBMS figures out the best way to locate the data.
The applications are coded without knowing the physical organization of the data.
File processing is done by the DBMS rather than the application program.
The application program describes the database at a high-level, conceptual view; this high-level description is called a data model.
One of the most popular data models is the relational data model; SQL is a commercially used, standard relational-database language.

Difficulties and issues handled by a DBMS:

- data independence: changes in file organization should not require changes in the application programs; example of common changes: add new fields to records of a file, add an index to a file, add and remove secondary indexes.
- data sharing: care must be taken when several users may be reading and modifying the data.
- data integrity: ensuring consistency of data (when data is updated, related data must be updated)

You you learn a lot about this in CSI 3317 ...