FILE STRUCTURES

Subject Code: 10IS63

PART – A

UNIT – 1 **7 Hours**

Introduction: File Structures: The Heart of the file structure Design, A Short History of File Structure Design, A Conceptual Toolkit; Fundamental File Operations: Physical Files and Logical Files, Opening Files, Closing Files, Reading and Writing, Seeking, Special Characters, The Unix Directory Structure, Physical devices and Logical Files, File-related Header Files, UNIX file System Commands; Secondary Storage and System Software: Disks, Magnetic Tape, Disk versus Tape; CD-ROM: Introduction, Physical Organization, Strengths and Weaknesses; Storage as Hierarchy, A journey of a Byte, Buffer Management, Input /Output in UNIX.

UNIT – 2 **6 Hours**

Fundamental File Structure Concepts, Managing Files of Records : Field and Record Organization, Using Classes to Manipulate Buffers, Using Inheritance for Record Buffer Classes, Managing Fixed Length, Fixed Field Buffers, An Object-Oriented Class for Record Files, Record Access, More about Record Structures, Encapsulating Record Operations in a Single Class, File Access and File Organization.

UNIT – 3 **7 Hours**

Organization of Files for Performance, Indexing: Data Compression, Reclaiming Space in files, Internal Sorting and Binary Searching, Keysorting; What is an Index? A Simple Index for Entry-Sequenced File, Using Template Classes in C++ for Object I/O, Object-Oriented support for Indexed, Entry-Sequenced Files of Data Objects, Indexes that are too large to hold in Memory, Indexing to provide access by Multiple keys, Retrieval Using Combinations of Secondary Keys, Improving the Secondary Index structure: Inverted Lists, Selective indexes, Binding.

UNIT – 4 **6 Hours**

Cosequential Processing and the Sorting of Large Files: A Model for Implementing Cosequential Processes, Application of the Model to a General Ledger Program, Extension of the Model to include Mutiway Merging, A Second Look at Sorting in Memory, Merging as a Way of Sorting Large Files on Disk.

PART - B

UNIT – 5 **7 Hours**

Multi-Level Indexing and B-Trees: The invention of B-Tree, Statement of the problem, Indexing with Binary Search Trees; Multi-Level Indexing, BTrees, Example of Creating a B-Tree, An Object-Oriented Representation of B-Trees, B-Tree Methods; Nomenclature, Formal Definition of B-Tree Properties, Worst-case Search Depth, Deletion, Merging and Redistribution, Redistribution during insertion; B* Trees, Buffering of pages; Virtual BTrees; Variable-length Records and keys.

UNIT – 6 **6 Hours**

Indexed Sequential File Access and Prefix B + Trees: Indexed Sequential Access, Maintaining a Sequence Set, Adding a Simple Index to the Sequence Set, The Content of the Index: Separators Instead of Keys, The Simple Prefix B+ Tree and its maintenance, Index Set Block Size, Internal Structure of Index Set Blocks: A Variable-order B- Tree, Loading a Simple Prefix B+ Trees, B-Trees, B+ Trees and Simple Prefix B+ Trees in Perspective.

UNIT – 7 **7 Hours**

Hashing: Introduction, A Simple Hashing Algorithm, Hashing Functions and Record Distribution, How much Extra Memory should be used?, Collision resolution by progressive overflow, Buckets, Making deletions, Other collision resolution techniques, Patterns of record access.

UNIT – 8 **6 Hours**

Extendible Hashing: How Extendible Hashing Works, Implementation, Deletion, Extendible Hashing Performance, Alternative Approaches.

Text Books:

1. Michael J. Folk, Bill Zoellick, Greg Riccardi: File Structures-An Object Oriented Approach with C++, 3rd Edition, Pearson Education, 1998. (Chapters 1 to 12 excluding 1.4, 1.5, 5.5, 5.6, 8.6, 8.7, 8.8)

Reference Books:

1. K.R. Venugopal, K.G. Srinivas, P.M. Krishnaraj: File Structures Using C++, Tata McGraw-Hill, 2008.
2. Scot Robert Ladd: C++ Components and Algorithms, BPB Publications, 1993.
3. Raghu Ramakrishan and Johannes Gehrke: Database Management Systems, 3rd Edition, McGraw Hill, 2003.

TABLE OF CONTENTS

PART – A

UNIT – 1

Introduction to the Design and Specification of File Structures

1.1 The Heart of the file structure Design

File :	A data structure on secondary storage which acts as a non-volatile container for data. File is a name given to any kind of document stored in any type of storage device which can be read by the computer. A file is identified by a name followed by a filename extension.
File Structure :	A pattern for arranging data in a file. It is a combination of representations for data in files and of operations for accessing the data.

Primary Goals for Design of File Structures and Algorithms

1) Minimize the number of disk accesses.

- If possible, transfer all information needed in one access.
- Group related information physically so it can be accessed together.

2) Maximize the space utilization

- Use compression techniques wherever possible
- Apply defragmentation procedures
- Avoid data redundancy

Problems and Concerns

- File data is frequently dynamic - that is, it changes from time to time.
- Designing file structures for changes adds complexity.
- Typical file sizes are growing.
- Solutions which work for small files may be inadequate for large files.
- File structures, algorithms, and data structures must work together.

1.2 A Short History of File Structure Design

- Earlier, the file access was sequential, and the cost of access grew in direct proportion to the size of the file. So, Indexes were added to files.
- Indexes made it possible to keep a list of keys and pointers in a smaller file that could be searched more quickly.
- Simple indexes became difficult to manage for dynamic files in which the set of keys changes. Hence tree structures were introduced.
- Trees grew unevenly as records were added and deleted, resulting in long searches requiring multiple disk accesses to find a record. Hence an elegant, self-adjusting binary tree structure called an AVL tree was developed for data in memory.

- Even with a balanced binary tree, dozens of accesses were required to find a record in moderate-sized files.
- A method was needed to keep a tree balanced when each node of the tree was not a single record, as in a binary tree, but a file block containing hundreds of records. Hence, B-Trees were introduced.
- AVL trees grow from top down as records are added, B-Trees grow from the bottom up.
- B-Trees provided excellent access performance but, a file could not be accessed sequentially with efficiency.
- The above problem was solved using B+ tree which is a combination of a B-Tree and a sequential linked list added at the bottom level of the B-Tree.
- To further reduce the number of disk accesses, hashing was introduced for files that do not change size greatly over time.
- Extendible, dynamic hashing was introduced for volatile, dynamic files which change.

2. Fundamental File Processing Operations

2.1 Physical Files and Logical Files

Physical file

A file as seen by the operating system, and which actually exists on secondary storage.

Logical file

A file as seen by a program.

- Programs read and write data from logical files.
- Before a logical file can be used, it must be associated with a physical file.
- This act of connection is called "opening" the file.
- Data in a physical file is persistent.
- Data in a logical file is temporary.
- A logical file is identified (within the program) by a program variable or constant.
- The name and form of the physical file are dependent on the operating system, not on the programming language.

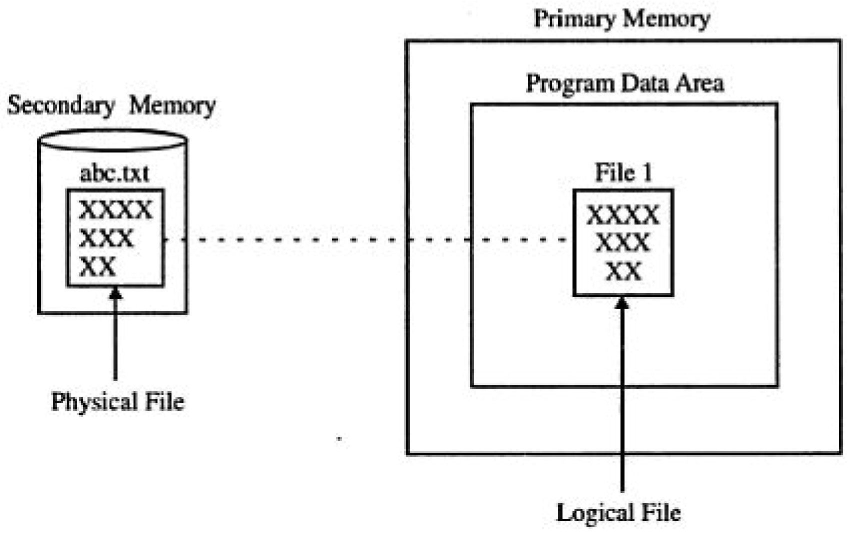


Figure 1.1 Relationship between Physical File and Logical File

2.2 Opening Files

Open: To associate a logical program file with a physical system file.

We have two options: 1) open an existing file or 2) Create a new file, deleting any existing contents in the physical file.

Opening a file makes it ready for use by the program

The C++ *open* function is used to open a file.

The open function must be supplied with (as arguments):

- The name of the physical file
- The access mode
- For new files, the protection mode

The value returned by the *open* is the fd, and is assigned to the file variable.

Function to open a file:

fd = open(filename,flags[,pmode]);

fd-file descriptor

A cardinal number used as the identifier for a logical file by operating systems such as UNIX and PC-DOS.

For handle level access, the logical file is declared as an *int*.

The handle is also known as a *file descriptor.*

Prototypes:

```
int open (const char* Filename, int Access);
int open (const char* Filename, int Access, int Protection);
```

Example:

```
int Input;
Input = open ("Daily.txt", O_RDONLY);
```

The following **flags** can be bitwise *or*ed together for the **access mode**:

O_RDONLY :	Read only
O_WRONLY :	Write only
O_RDWR :	Read or write
O_CREAT :	Create file if it does not exist
O_EXCL :	If the file exists, truncate it to a length of zero, destroying its contents. (used only with O_CREAT)
O_APPEND :	Append every write operation to the end of the file
O_TRUNC :	Delete any prior file contents

Pmode- protection mode

The security status of a file, defining who is allowed to access a file, and which access modes are allowed.

- Supported protection modes depend on the operating system, not on the programming language.
- DOS supports protection modes of:
  - Read only
  - Hidden
  - System

  for all uses of the system.
- UNIX supports protection modes of:
  - Readable
  - Writable
  - Executable

  for users in three categories:
  - Owner (Usually the user who created the file)
  - Group (members of the same group as the owner)
  - World (all valid users of the system)
- Windows supports protection modes of:
  - Readable
  - Modifiable
  - Writable
  - Executable

for users which can be designated individually or in groups.:

In Unix, the pmode is a three digit octal number that indicates how the file can be used by the owner(first digit), by members of the owner's group(second digit), and by everyone else(third digit). For example, if pmode is 0751, it is interpreted as

Example:

```
51);
```

2.3 Closing Files

close

To disassociate a logical program file from a physical system file.

- Closing a file frees system resources for reuse.
- Data may not be actually written to the physical file until a logical file is closed.
- A program should close a file when it is no longer needed.

  The C++ *close* function is used to close a file for handle level access.

  The handle close function must be supplied with (as an argument):

  - The handle of the logical file

  The value returned by the *close* is 0 if the close succeeds, and -1 if the close fails..

Prototypes:

```
int close (int Handle);
```

Example:

```
close (Input);
```

2.4 Reading and Writing

read

To transfer data from a file to program variable(s).

write

To transfer data to a file from program variable(s) or constant(s).

- The read and write operations are performed on the logical file with calls to library functions.
- For read, one or more variables must be supplied to the read function, to receive the data from the file.

- For write, one or more values (as variables or constants) must be supplied to the write function, to provide the data for the file.
- For unformatted transfers, the amount of data to be transferred must also be supplied.

2.4.1 Read and Write Functions

Reading

- The C++ *read* function is used to read data from a file for handle level access.
- The read function must be supplied with (as an arguments):
  - The source file to read from
  - The address of the memory block into which the data will be stored
  - The number of bytes to be read(byte count)
- The value returned by the *read* function is the number of bytes read.

Read function:

Prototypes:

```
int read (int Handle, void * Buffer, unsigned Length);
```

Example:

```
read (Input, &C, 1);
```

Writing

- The C++ *write* function is used to write data to a file for handle level access.
- The handle write function must be supplied with (as an arguments):
  - The logical file name used for sending data
  - The address of the memory block from which the data will be written
  - The number of bytes to be write
- The value returned by the *write* function is the number of bytes written.

  Write function:

Prototypes:

```
int write (int Handle, void * Buffer, unsigned Length);
```

Example:

```
write (Output, &C, 1);
```

2.4.2 Files with C Streams and C++ Stream Classes

- For FILE level access, the logical file is declared as a pointer to a FILE (FILE *)
- The FILE structure is defined in the stdio.h header file.

Opening

The C++ *fopen* function is used to open a file for FILE level access.

- The FILE fopen function must be supplied with (as arguments):
    - The name of the physical file
    - The access mode
- The value returned by the *fopen* is a pointer to an open FILE, and is assigned to the file variable.

fopen function:

```
file = fopen (filename, type);
```

Prototypes:

FILE * fopen (const char* Filename, char * Access);

Example:

FILE * Input;

Input = fopen ("Daily.txt", "r");

The access mode should be one of the following strings:

r

Open for reading (existing file only) in text mode

r+

Open for update (existing file only)

w

Open (or create) for writing (and delete any previous data)

w+

Open (or create) for update (and delete any previous data)

a

Open (or create) for append with file pointer at current EOF (and keep any previous data) in text mode

a+

Open (or create) for append update (and keep any previous data)

Closing

The C++ *fclose* function is used to close a file for FILE level access.

The FILE fclose function must be supplied with (as an argument):

- A pointer to the FILE structure of the logical file

The value returned by the *fclose* is 0 if the close succeeds, and &neq;0 if the close fails..

Prototypes:

int fclose (FILE * Stream);

Example:

fclose (Input);

Reading

The C++ *fread* function is used to read data from a file for FILE level access.

The FILE fread function must be supplied with (as an arguments):

- A pointer to the FILE structure of the logical file
- The address of the buffer into which the data will be read
- The number of items to be read
- The size of each item to be read, in bytes

The value returned by the *fread* function is the number of items read.

Prototypes:

size_t fread (void * Buffer, size_t Size, size_t Count, FILE * Stream);

Example:

fread (&C, 1, 1, Input);

Writing

The C++ *fwrite* function is used to write data to a file for FILE level access.

The FILE fwrite function must be supplied with (as an arguments):

- A pointer to the FILE structure of the logical file
- The address of the buffer from which the data will be written
- The number of items to be written
- The size of each item to be written, in bytes

The value returned by the *fwrite* function is the number of items written.

Prototypes:

size_t fwrite (void * Buffer, size_t Size, size_t Count, FILE * Stream);

Example:

fwrite (&C, 1, 1, Output);

2.4.3 Programs in C++ to Display the contents of a File

The first simple file processing program opens a file for input and reads it, character by character, sending each character to the screen after it is read from the file. This program includes the following steps

1. Display a prompt for the name of the input file.
2. Read the user's response from the keyboard into a variable called filename.
3. Open the file for input.
4. While there are still characters to be read from the input file,
   - Read a character from the file;
   - Write the character to the terminal screen.
5. Close the input file.

Figures 2.2 and 2.3 are C++ implementations of this program using C streams and C++ stream classes, respectively.

```
// listc.cpp
// program using C streams to read characters from a file
// and write them to the terminal screen
#include <stdio.h>
main( ) {
  char ch;
  FILE * file; // pointer to file descriptor
  char filename[20];
  printf("Enter the name of the file: ");    // Step 1
  gets(filename);                            // Step 2
  file =fopen(filename, "r");                // Step 3
  while (fread(&ch, 1, 1, file) != 0)        // Step 4a
    fwrite(&ch, 1, 1, stdout);               // Step 4b
  fclose(file);                              // Step 5
}
```

Figure 2.2 The file listing program using C streams (`listc.cpp`).

```
// listcpp.cpp
// list contents of file using C++ stream classes
#include <fstream.h>
main () {
  char ch;
  fstream file; // declare unattached fstream
  char filename[20];
  cout <<"Enter the name of the file: " // Step 1
    <<flush; // force output
  cin >> filename;                      // Step 2
  file . open(filename, ios::in);       // Step 3
  file . unsetf(ios::skipws);// include white space in read
  while (1)
  {
    file >> ch;                         // Step 4a
    if (file.fail()) break;
    cout << ch;                         // Step 4b
  }
  file . close();                       // Step 5
}
```

Figure 2.3 The file listing program using C++ stream classes (`listcpp.cpp`).

In the C++ version, the call file.unsetf(ios::skipws) causes operator >> to include white space (blanks, end-of-line,tabs, ans so on).

2.4.4 Detecting End of File

end-of-file

A physical location just beyond the last datum in a file.

- The acronym for end-of-file is EOF.
- When a file reaches EOF, no more data can be read.
- Data can be written at or past EOF.
- Some access methods set the end of file flage after a read reaches the end of file position.

Other access methods set the end of file flag after a read attempts to read beyond the end of file position.

Detecting End of File

The C++ *feof* function is used to detect when the file pointer of an fstream is **past** end of file..

The FILE *feof* function has one argument.

- A pointer to the FILE structure of the logical file

The value returned by the *feof* function is 1 if end of file is true and 0 if end of file is false.

Prototypes:

```
int feof (FILE * Stream);
```

Example:

```
if (feof (Input))
cout << "End of File\n";
```

In some languages, a function end_of_file can be used to test for end-of-file. The OS keeps track of read/write pointer. The end_of_file function queries the system to see whether the read/write pointer has moved past the last element in the file.

2.5 Seeking

The action of moving directly to a certain position in a file is called seeking.

seek

To move to a specified location in a file.

byte offset

The distance, measured in bytes, from the beginning.

- Seeking moves an attribute in the file called the *file pointer.*
- C++ library functions allow seeking.
- In DOS, Windows, and UNIX, files are organized as streams of bytes, and locations are in terms of byte count.
- Seeking can be specified from one of three reference points:
  - The beginning of the file.
  - The end of the file.

- The current file pointer position.

A seek requires two arguments

```
Seek(Source_file, Offset)
```

`Source_file`	**The logical file name in which the seek will occur.**
`Offset`	**The number of positions in the file the pointer is to be moved from the start of the file.**

Example

```
Seek(data, 373)
```

2.5.1 Seeking with C Streams

Fseek function:

```
pos = fseek(file, byte_offset, origin)
```

The C++ *fseek* function is used to move the file pointer of a file identified by its FILE structure.

The FILE fseek function must be supplied with (as an arguments):

- The file descriptor of the file(file)
- The number of bytes to move from some origin in the file(byte_offset)
- The starting point from which the byte_offset is to be taken(origin)

The Origin argument should be one of the following, to designate the reference point:

SEEK_SET: Beginning of file
SEEK_CUR: Current file position
SEEK_END: End of file

The value returned(pos) by the *fseek* function is the positon of the read/write pointer from the beginning of the file after its moved

Prototypes:

```
long fseek (FILE * file, long Offset, int Origin);
```

Example:

```
long pos;
fseek (FILE * file, long Offset, int Origin);
...
pos=fseek (Output, 100, SEEK_BEG);
```

2.5.2 Seeking with C++ Stream Classes

In C++, an object of type fstream has 2 file pointers:a get pointer for input and a put pointer for output. Two functions for seeking are

seekg: moves get pointer
seekp: moves put pointer

syntax for seek operations:

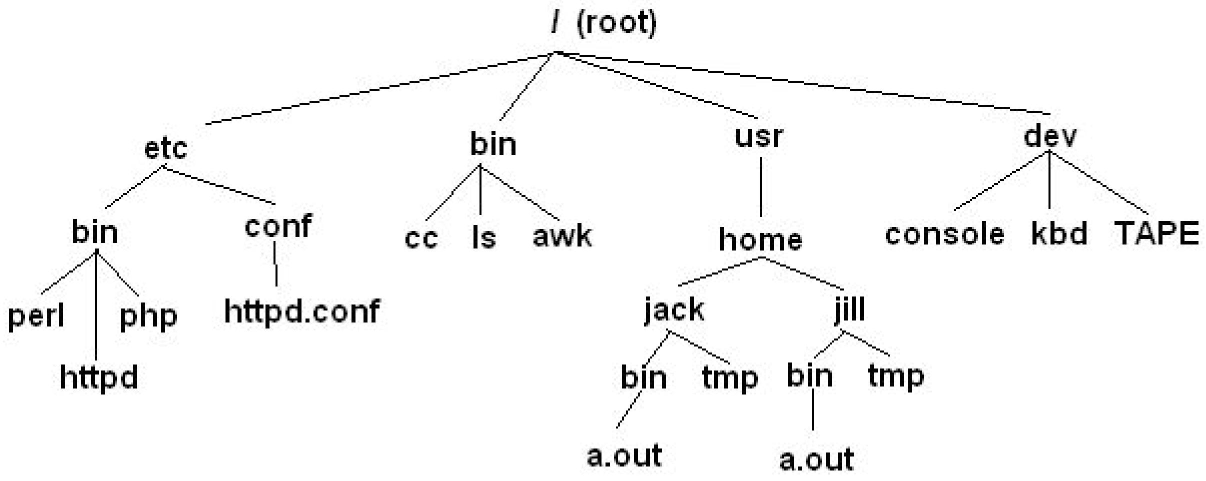
```
        file.seekg(byte_offset,origin)
        file.seekp(byte_offset,origin)
```

2.6 Special Characters in Files

- When DOS files are opened in *text* mode, the internal separator ('\n') is translated to the the external separator (<CR><LF>) during read and write.CR-carriage return, LF-Line feed
- When DOS files are opened in *binary* mode, the internal separator ('\n') is **not** translated to the the external separator (<CR><LF>) during read and write.
- In DOS (Windows) files, end-of-file can be marked by a "control-Z" character (ASCII *SUB*).
- In C++ implementations for DOS, a control-Z in a file is interpreted as end-of-file.

2.7 The Unix Directory Structure

- The Unix file system is a tree-structured organization of directories,with the root of the tree signified by the character /.
- In UNIX, the directory structure is a single tree for the entire file system.



- In UNIX, separate disks appear as subdirectories of the root (/).
- In UNIX, the subdirectories of a pathname are separated by the forward slash character (/).
- Example: /usr/bin/perl
- The directory structure of UNIX is actually a graph, since symbolic links allow entries to appear at more than one location in the directory structure.

2.8 Physical Devices and Logical Files

2.8.1 Physical devices as files

In unix, devices like keyboard and console are also files. The keyboard produces a sequence of bytes that are sent to the computer when keys are pressed. The console accepts a sequence of bytes and displays the symbols on screen.

A Unix file is represented logically by an integer-the file descriptor

A keyboard, a disk file, and a magnetic tape are all represented by integers.

This view of a file in Unix makes it possible to do with a very few operations compared to other OS.

2.8.2 The console, the keyboard and standard error

In C streams, the keyboard is called stdin(standard input),console is called stdout(standard output) error file is called stderr(standard error).

Handle	**FILE**	**iostream**	**Description**
0	stdin	Cin	Standard Input
1	stdout	Cout	Standard Output
2	stderr	Cerr	Standard Error

2.8.3 I/O redirection and Pipes

Operating systems provide shortcuts for switching between standard I/O(stdin and stdout) and regular file I/O

I/O redirection is used to change a program so it writes its output to a regular file rather than to stdout.

- In both DOS and UNIX, the standard output of a program can be redirected to a file with the > symbol.
- In both DOS and UNIX, the standard input of a program can be redirected to a file with the < symbol.

The notations for input and output redirection on the command line in Unix are

```
< file                  (redirect stdin to "file")
> file                  (redirect stdout to "file")
```

Example:

```
  list.exe > myfile
```

The output of the executable file is redirected to a file called “myfile”

pipe

Piping: using the output of one program as input to another program.

A connection between standard output of one process and standard input of a second process.

- In both DOS and UNIX, the standard output of one program can be piped (connected) to the standard input of another program with the **|** symbol.
- Example:

```
program1 | program2
```

Output of program1 is used as input for program2

2.9 File-Related Header Files

- Header files can vary with the C++ implementation.

  Stdio.h, iostream.h, fstream.h, fcntl.h and file.h are some of the header files used in different operating systems

2.10 Unix File System Commands

UNIX	**Description**
cat *filename*	Type the contents of a file
tail *filename*	Type the last ten lines of a file
cp *file1 file2*	Copy file1 to file2
mv *file1 file2*	Move(rename) file1 to file2
rm *filenames*	Delete files
chmod *mode filename*	Change the protection mode
ls	List contents of a directory
mkdir	Create directory
rmdir	Remove directory

3. Secondary Storage and Systems Software

3.1 Disks

- Compared with time for memory access, disk access is always expensive.
- Disk drives belong to a class of devices called direct access storage devices(DASDs).
- Hard-disks offer high capacity and low cost per bit(commonly used).

- Floppy disks are inexpensive, slow and hold little data.
- Removable disks use disk cartridges that can be mounted on same drive at different times.data can be accessed directly.

3.1.1 Organization of Disks

- The information on disk is stored on the surface of 1 or more platters.(Fig 3.1)
- The information is stored in successive **tracks** on the surface of the disk.(Fig 3.2)
- Each track is divided into **sectors**.
- A sector is the smallest addressable portion of a disk.
- Disk drives have a number of platters.
- The tracks directly above one another form a **cylinder(Fig 3.3)**
- All information on a single cylinder can be accessed without moving the arm that holds the read/write heads.
- Moving this arm is called seeking.

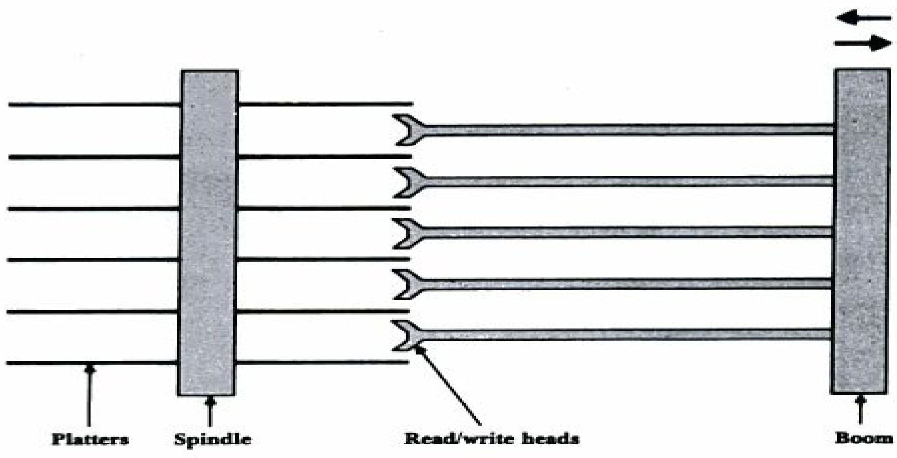


Figure 3.1 Schematic illustration of disk drive.

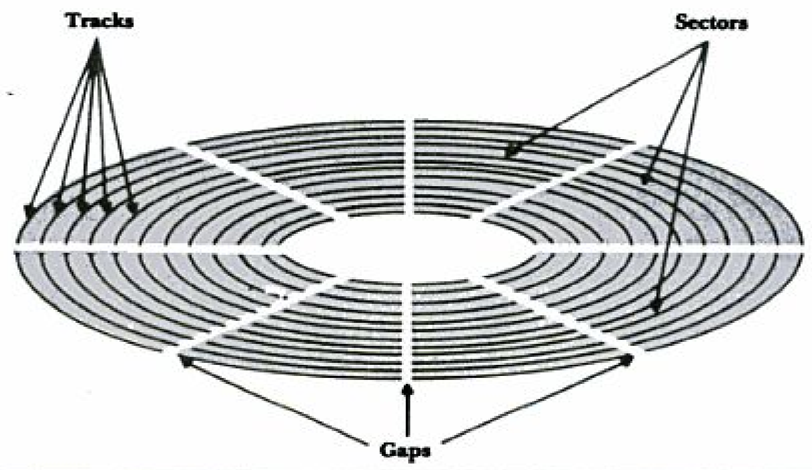


Figure 3.2 Surface of disk showing tracks and sectors.

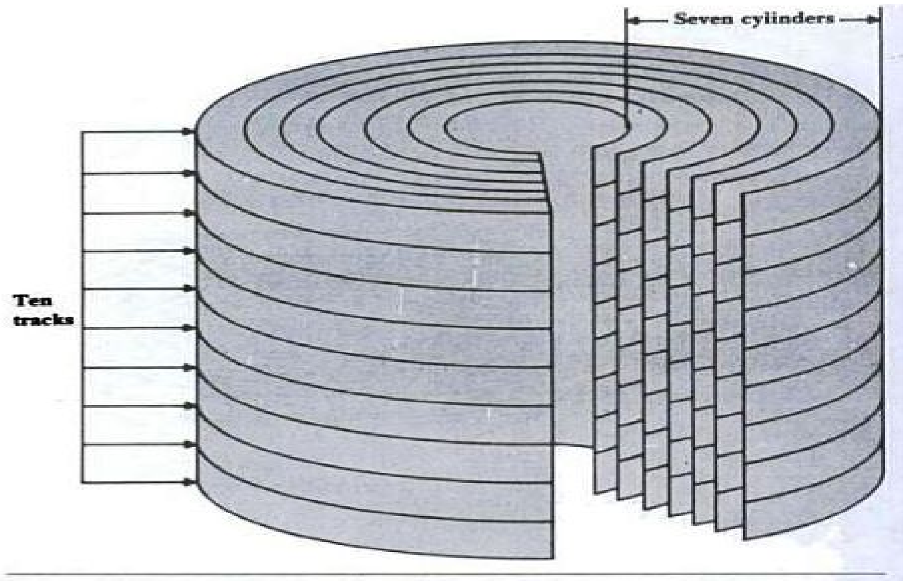


Figure 3.3 Schematic illustration of disk drive viewed as a set of seven cylinders.

3.1.2 Estimating Capacities and space needs

In a disk, each platter has 2 surfaces, so number of cylinders is same as number of tracks on a single surface.

Since a cylinder consists of a group of tracks, a track consists of a group of sectors, a sector has a group of bytes, track,cylinder and drive capacities can be computed as follows

$$\text{Track capacity} = \text{number of sectors per track} \times \text{bytes per sector}$$
$$\text{Cylinder capacity} = \text{number of tracks per cylinder} \times \text{track capacity}$$
$$\text{Drive capacity} = \text{number of cylinders} \times \text{cylinder capacity.}$$

Given a disk with following characteristics

$$\text{Number of bytes per sector} = 512$$
$$\text{Number of sectors per track} = 63$$
$$\text{Number of tracks per cylinder} = 16$$
$$\text{Number of cylinders} = 4092$$

How many cylinders does the file require if each data record requires 256 bytes? Since each sector can hold two records, the file requires

$$\frac{50\,000}{2} = 25\,000 \text{ sectors}$$

One cylinder can hold

$$63 \times 16 = 1008 \text{ sectors}$$

so the number of cylinders required is approximately

$$\frac{25\,000}{1008} = 24.8 \text{ cylinders}$$

3.1.3 Organizing Tracks by Sector

Two ways to organize data on disk: by sector and by user defined block.

The physical placement of sectors

Different views of sectors on a track:

- Sectors that are adjacent, fixed size segments of a track that happen to hold a file(Fig 3.4a). When you want to read a series of sectors that are all in the same track, one right after the other, you often cannot adjacent sectors. In Fig 3.4a, it takes thirty-two revolutions to read the entire 32 sectors of a track.
- Interleaving sectors: leaving an interval of several physical sectors between logically adjacent sectors. Fig 3.4(b) illustrates the assignment of logical sector content to the thirty-two physical sectors in a track with interleaving factor of 5. In Fig 3.4b, It takes five revolutions to read the entire 32 sectors of a track.

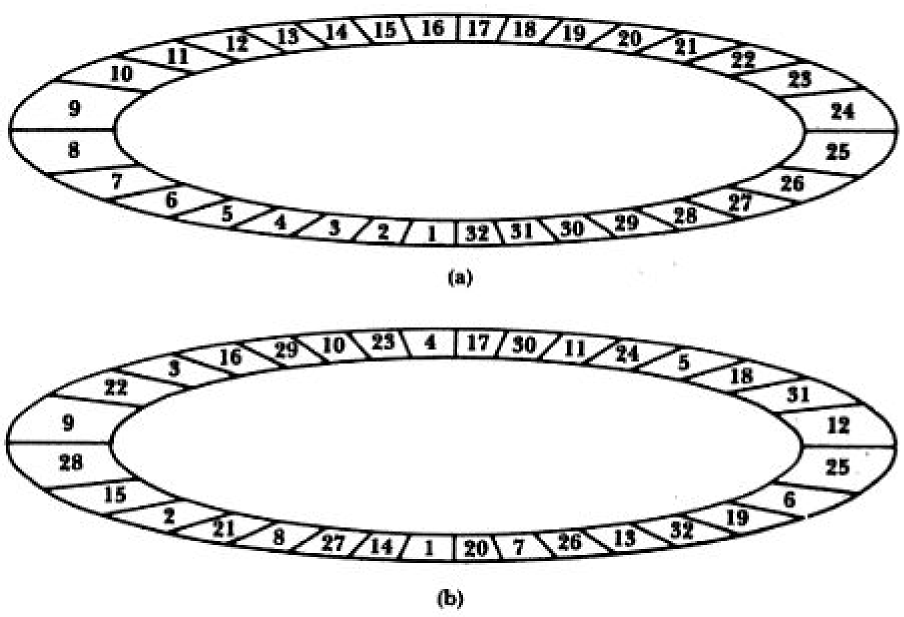


Figure 3.4 Two views of the organization of sectors on a thirty-two-sector track.

cluster

A group of sectors handled as a unit of file allocation. A cluster is a fixed number of contiguous sectors

extent

A physical section of a file occupying adjacent clusters.

fragmentation

Unused space within a file.

- Clusters are also referred to as allocation units (ALUs).
- Space is allocated to files as integral numbers of clusters.
- A file can have a single extent, or be scattered in several extents.

- Access time for a file increases as the number of separate extents increases, because of seeking.
- Defragmentation utilities physically move files on a disk so that each file has a single extent.
- Allocation of space in clusters produces fragmentation.
- A file of one byte is allocated the space of one cluster.
- On average, fragmentation is one-half cluster per file.

3.1.5 Organizing Tracks by Block

- Mainframe computers typically use variable size physical blocks for disk drives.
- Track capacity is dependent on block size, due to fixed overhead (gap and address block) per block.

3.1.6 The Cost of a Disk Access

direct access device

A data storage device which supports direct access.

direct access

Accessing data from a file by record position with the file, without accessing intervening records.

access time

The total time required to store or retrieve data.

transfer time

The time required to transfer the data from a sector, once the transfer has begun.

seek time

The time required for the head of a disk drive to be positioned to a designated cylinder.

rotational delay

The time required for a designated sector to rotate to the head of a disk drive.

- Access time of a disk is related to physical movement of the disk parts.
- Disk access time has three components: seek time, rotational delay, and transfer time.
- Seek time is affected by the size of the drive, the number of cylinders in the drive, and the mechanical responsiveness of the access arm.
- Average seek time is approximately the time to move across 1/3 of the cylinders.
- Rotational delay is also referred to as *latency.*
- Rotational delay is inversely proportional to the rotational speed of the drive.
- Average rotational delay is the time for the disk to rotate 180°.
- Transfer is inversely proportional to the rotational speed of the drive.
- Transfer time is inversely proportional to the physical length of a sector.
- Transfer time is roughly inversely proportional to the number of sectors per track.

- Actual transfer time may be limited by the disk interface.

3.1.8 Effect of Block Size

- Fragmentation waste increases as cluster size increases.
- Average access time decreases as cluster size increases.

3.1.9 Disk as a bottleneck

striping

The distribution of single files to two or more physical disk drives.

Redundant Array of Inexpensive Disks

An array of multiple disk drives which appears as a single drive to the system.

RAM disk

A virtual disk drive which actually exists in main memory.

solid state disk

A solid state memory array with an interface which responds as a disk drive.

cache

Solid state memory used to buffer and store data temporarily.

- Several techniques have been developed to improve disk access time.
- Striping allows disk transfers to be made in parallel.
- There are 6 versions, or levels, of RAID technology.
- RAID-0 uses striping.
- RAID-0 improves access time, but does not provide redundancy.
- RAID-1 uses mirroring, in which two drives are written with the same data.
- RAID-1 provides complete redundancy. If one drive fails, the other provides data backup.
- RAID-1 improves read access time, but slows write access time.
- RAM disks appear to programs as fast disk drives.
- RAM disks are volatile.
- Solid state disks appear to computer systems as fast disk drives.
- Solid state disks are used on high performance data base systems.
- Caching improves average access time.
- Disk caching can occur at three levels: in the computer main memory, in the disk controller, and in the disk drive.
- Windows operating systems use main memory caching.
- Disk controller caching requires special hardware.
- Most disk drives now contain caching memory.
- With caching, writes are typically reported as complete when the data is in the cache.

- The physical write is delayed until later.
- With caching, reads typically read more data than is requested, storing the unrequested data in the cache.
- If a read can be satisfied from data already in the cache, no additional physical read is needed.
- Read caching works on average because of program locality.

3.2 Magnetic Tape

sequential access device

A device which supports sequential access.

3.3 Disk vs Tape

Tape-based data backup infrastructures have inherent weaknesses: Tape is not a random access medium. Backed up data must be accessed as it was written to tape. Recovering a single file from a tape often requires reading a substantial portion of the tape and can be very time consuming. The recovery time of restoring from tape can be very costly. Recent studies have shown most IT administrators do not feel comfortable with their tape backups today.

The Solution

Disk-to-disk backup can help by complimenting tape backup. Within the data center, data loss is most likely to occur as a result of file corruption or inadvertent deletion. In these scenarios, disk-to-disk backup allows a much faster and far more reliable restore process than is possible with a tape device, greatly reducing the demands on the tape infrastructure and on the manpower required to maintain it. Disk-to-disk backup is quickly becoming the standard for backup since data can be backed up more quickly than with tape, and restore times are dramatically reduced.

3.4 Introduction to CD-ROM

A **CD-ROM,** an acronym of "Compact Disc Read-only memory") is a pre-pressed compact disc that contains data accessible to, but not writable by, a computer for data storage and music playback.

CD-ROMs are popularly used to distribute computer software, including video games and multimedia applications, though any data can be stored (up to the capacity limit of a disc). Some CDs hold both computer data and audio with the latter capable of being played on a CD player, while data (such as software or digital video) is only usable on a computer (such as ISO 9660 format PC CD-ROMs). These are called enhanced CDs.

- A single disc can hold more than 600 megabytes of data (~ 400 books of the textbook's size)

- CD-ROM is read only. i.e., it is a publishing medium rather than a data storage and retrieval like magnetic disks.

3.5 Physical Organization of CD-ROM

Tracks and sectors

- There is only one track, a long spiral, much like the groove in a vinyl LP.
- The track is divided up into 2353 byte sectors
- Each sector contains 2048 bytes of user data and 305 bytes of non-data overhead
- Since there is only 1 track, sectors are not addressed as they are on disks.

Sectors and the audio ancestors

- 75 sectors create 1 second of audio. 60 seconds of audio create 1 minute of audio.
- We provide a sector address in this format: mm:ss:cc, mm is minutes, ss is seconds, cc is sector #
- Humans are said to be capable of hearing sounds approximately in the range form 20Hz - 20KHz.
- You need to sample a signal at twice the rate at which you want to produce it.
- Therefore if we want to reproduce sound, we must sample at approximately twice this frequency -- about 40KHz.
- Specifically the designers of audio CD's sampled at 44.1KHz
- Each sample on an audio CD is two bytes allowing for 65,536 distinct audio levels.
- For stereo sound, we sample twice each time -- left and right channels
- The implication is that we must store (2 x 2 x 44,200) bytes per second of audio. (This is 176, 000 bytes per second).
- Divide this 176,000 bytes by the 75 sectors per second, and discover the sector capacity of 2,352 bytes

Sector format on data disks

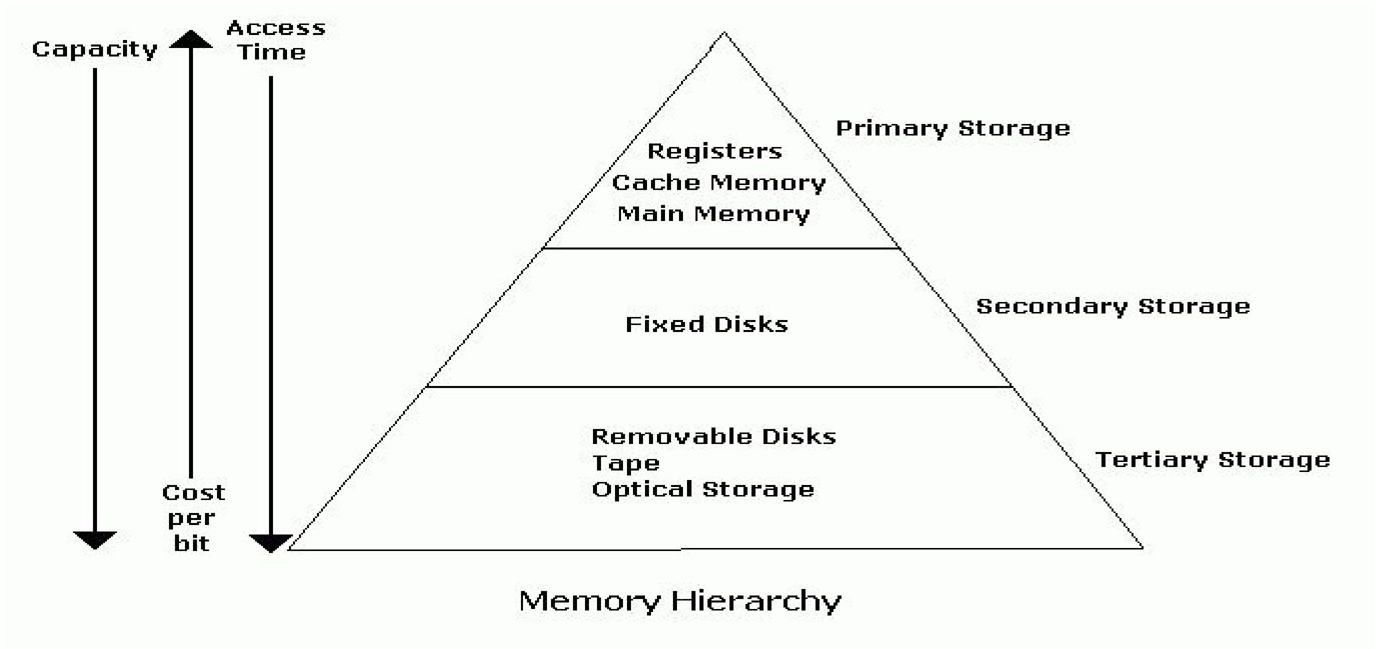
- 12 bytes synch
- 4 bytes sector ID
- 2,048 bytes user data
- 4 bytes error detection
- 8 bytes null
- 276 bytes error correction

Data is stored on the disc as a series of microscopic indentations. A laser is shone onto the reflective surface of the disc to read the pattern of pits and lands ("pits", with the gaps between them referred to as "lands"). Because the depth of the pits is approximately one-quarter to one-sixth of the wavelength of the laser light used to read the disc, the reflected beam's phase is shifted in relation to the incoming beam, causing destructive interference and reducing the reflected beam's intensity. This pattern of changing intensity of the reflected beam is converted into binary data.

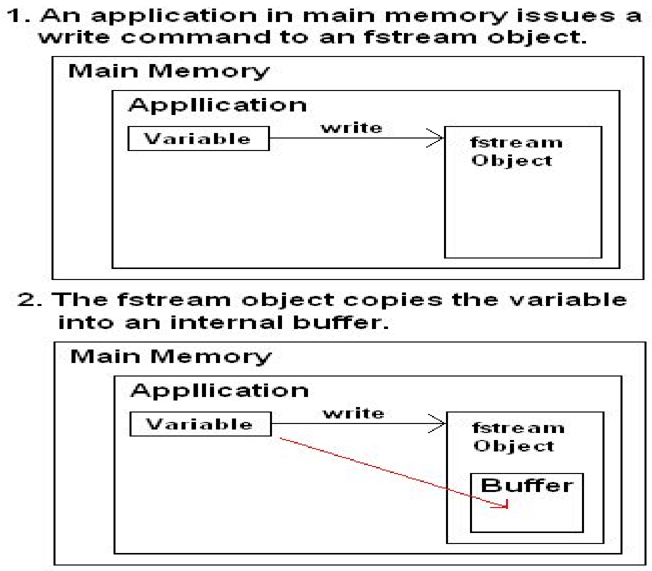
3.6 Strengths and Weaknesses

- CD-ROM Strengths: High storage capacity, inexpensive price, durability.
- CD-ROM Weaknesses: extremely slow seek performance (between 1/2 a second to a second) ==> Intelligent File Structures are critical.

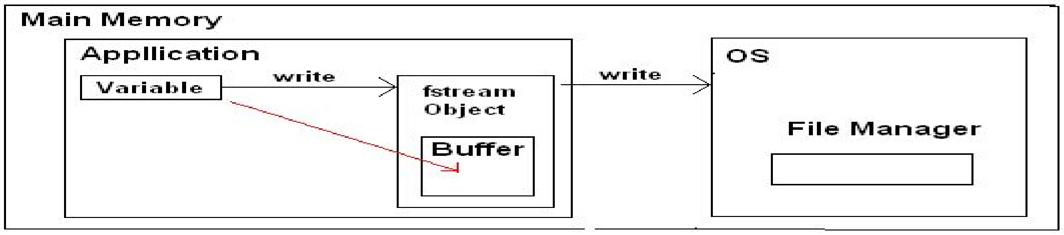
3.7 Storage as a Hierarchy



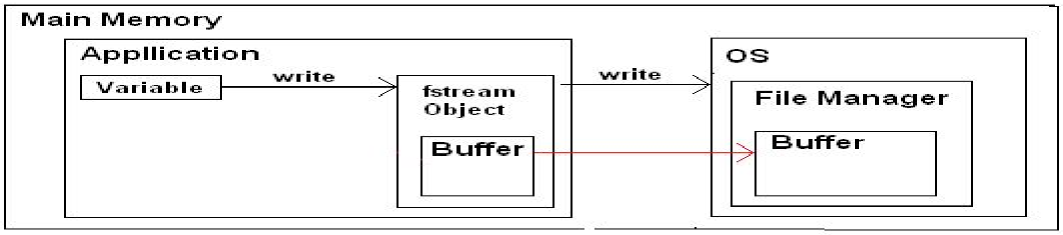
3.8 A Journey of a Byte


1. An application in main memory issues a write command to an fstream object.
Main Memory
Appllication
Variable
write
fstream Object
2. The fstream object copies the variable into an internal buffer.
Main Memory
Appllication
Variable
write
fstream Object
Buffer

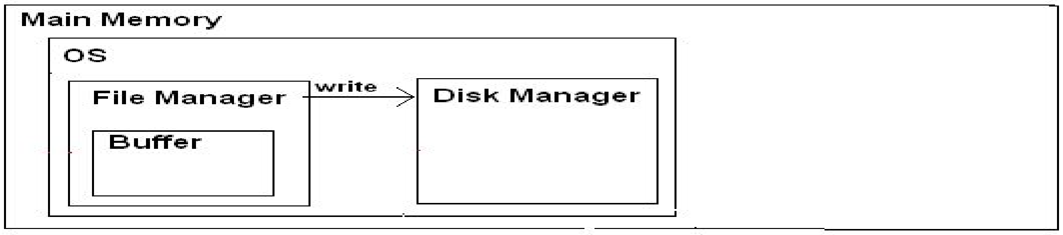
3. After repeated write, the fstream buffer fills up and the fstream object issues a write request to the operating system



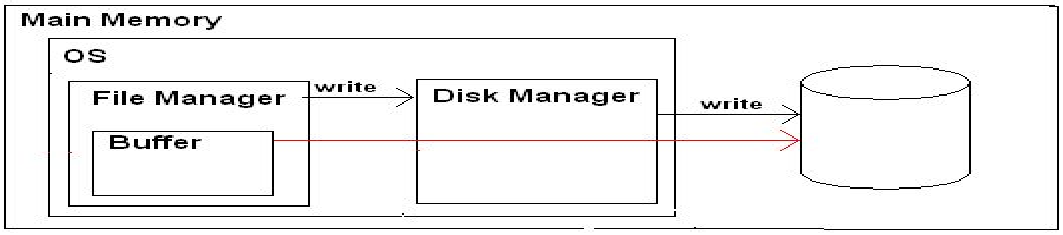
4. The operating system file manager copies the contents of the fstream buffer into an OS buffer.



5. After repeated writes, the OS buffer fills up, and the File Manager issues a write request to the OS disk manager.



6. The disk manager issues a write request to the disk drive



3.9 Buffer Management

Buffer Bottlenecks

- Assume that the system has a single buffer and is performing both input and output on one character at a time, alternatively.
- In this case, the sector containing the character to be read is constantly over-written by the sector containing the spot where the character will be written, and vice-versa.
- In such a case, the system needs more than 1 buffer: at least, one for input and the other one for output.
- Moving data to or from disk is very slow and programs may become I/O Bound ==> Find better strategies to avoid this problem.

Buffering strategies

Multiple buffering

-Double Buffering

-Buffer Pooling

Move mode and Locate mode

Scatter/Gather I/O

3.10 I/O in UNIX

block device

A device which transfers data in blocks (as opposed to character by character.)

block I/O

Input or output performed in blocks

character device

A device which transfers data character by character (as opposed to in blocks.)

character I/O

Input or output performed character by character.

- Disks are block devices.
- Keyboards, displays, and terminals are character devices.

UNIT-2

Fundamental File Structures Concepts

4.1 Field and Record Organization

4.1.1 A Stream File

- In the Windows, DOS, UNIX, and LINUX operating systems, files are not internally structured; they are streams of individual bytes.

F	r	e	d		F	l	i	n	t	s	t	o	n	e	4	4	4	4		G	r	a	n	**...**

- The only file structure recognized by these operating systems is the separation of a text file into lines.
  - For Windows and DOS, two characters are used between lines, a carriage return (ASCII 13) and a line feed (ASCII 10);
  - For UNIX and LINUX, one character is used between lines, a line feed (ASCII 10);
- The code in applications programs can, however, impose internal organization on stream files.

Record Structures

record

A subdivision of a file, containing data related to a single entity.

field

A subdivision of a record containing a single attribute of the entity which the record describes.

stream of bytes

A file which is regarded as being without structure beyond separation into a sequential set of bytes.

4.2 Using Classes to Manipulate Buffers

- Within a program, data is temporarily stored in variables.
- Individual values can be aggregated into structures, which can be treated as a single variable with parts.
- In C++, classes are typically used as as an aggregate structure.
- C++ Person class (version 0.1):

```
class Person {
  public:
    char FirstName [11];
    char LastName[11];
    char Address [21];
    char City [21];
    char State [3];
    char ZIP [5];
};
```

- With this class declaration, variables can be declared to be of type Person. The individual fields within a Person can be referred to as the name of the variable and the name of the field, separated by a period (.).
- C++ Program:

```
#include
class Person {
  public:
    char FirstName [11];
    char LastName[11];
    char Address [31];
    char City [21];
    char State [3];
    char ZIP [5];
};

void Display (Person);
int main () {
  Person Clerk;
  Person Customer;
  strcpy (Clerk.FirstName, "Fred");
  strcpy (Clerk.LastName, "Flintstone");
  strcpy (Clerk.Address, "4444 Granite Place");
  strcpy (Clerk.City, "Rockville");
  strcpy (Clerk.State, "MD");
  strcpy (Clerk.ZIP, "00001");
  strcpy (Customer.FirstName, "Lily");
  strcpy (Customer.LastName, "Munster");
  strcpy (Customer.Address, "1313 Mockingbird Lane");
  strcpy (Customer.City, "Hollywood");
  strcpy (Customer.State, "CA");
  strcpy (Customer.ZIP, "90210");
  Display (Clerk);
  Display (Customer);
}
void Display (Person Someone) {
  cout << Someone.FirstName << Someone.LastName
       << Someone.Address << Someone.City
       << Someone.State << Someone.ZIP;
}
```

- In memory, each Person will appear as an aggregate, with the individual values being parts of the aggregate:

Person					
Clerk					
FirstName	**LastName**	**Address**	**City**	**State**	**ZIP**
Fred	Flintstone	4444 Granite Place	Rockville	MD	0001

- The output of this program will be:

```
FredFlintstone4444 Granite PlaceRockvilleMD00001LilyMunster1313
Mockingbird LaneHollywoodCA90210
```

- Obviously, this output could be improved. It is marginally readable by people, and it would be difficult to program a computer to read and correctly interpret this output.

4.3 Using Inheritance for Record Buffer Classes

- **Inheritance**
  The implicit inclusion of members of a parent class in a child class.

Delineation of Records in a File

fixed length record

A record which is predetermined to be the same length as the other records in the file.

Record 1	**Record 2**	**Record 3**	**Record 4**	**Record 5**

- The file is divided into records of equal size.
- All records within a file have the same size.
- Different files can have different length records.
- Programs which access the file must know the record length.
- Offset, or position, of the nth record of a file can be calculated.

- There is no external overhead for record separation.
- There may be internal fragmentation (unused space within records.)
- There will be no external fragmentation (unused space outside of records) except for deleted records.
- Individual records can always be updated in place.
- Example (80 byte records):
- 0 66 69 72 73 74 20 6C 69 6E 65 0 0 1 0 0 0 first line......
- 10 0 0 0 0 0 0 0 0 FF FF FF FF 0 0 0 0 ....
- 20 68 FB 12 0 DC E0 40 0 3C BA 42 0 78 FB 12 0 h.....@.<.B.x...
- 30 CD E3 40 0 3C BA 42 0 8 BB 42 0 E4 FB 12 0 ..@.<.B...B.....
- 40 3C 18 41 0 C4 FB 12 0 2 0 0 0 FC 3A 7C 0 <.A..........:|.
- 50 73 65 63 6F 6E 64 20 6C 69 6E 65 0 1 0 0 0 second line.....
- 60 0 0 0 0 0 0 0 0 FF FF FF FF 0 0 0 0 ....
- 70 68 FB 12 0 DC E0 40 0 3C BA 42 0 78 FB 12 0 h.....@.<.B.x...
- 80 CD E3 40 0 3C BA 42 0 8 BB 42 0 E4 FB 12 0 ..@.<.B...B.....
- 90 3C 18 41 0 C4 FB 12 0 2 0 0 0 FC 3A 7C 0 <.A..........:|.
- Advantage: the offset of each record can be calculated from its record number. This makes direct access possible.
- Advantage: there is no space overhead.
- Disadvantage: there will probably be internal fragmentation (unusable space within records.)

Delimited Variable Length Records

variable length record

A record which can differ in length from the other records of the file.

delimited record

A variable length record which is terminated by a special character or sequence of characters.

delimiter

A special character or group of characters stored after a field or record, which indicates the end of the preceding unit.

Record 1	#	**Record 2**	#	**Record 3**	#	**Record 4**	#	**Record 5**	#

- The records within a file are followed by a delimiting byte or series of bytes.
- The delimiter cannot occur within the records.
- Records within a file can have different sizes.

- Different files can have different length records.
- Programs which access the file must know the delimiter.
- Offset, or position, of the nth record of a file cannot be calculated.
- There is external overhead for record separation equal to the size of the delimiter per record.
- There should be no internal fragmentation (unused space within records.)
- There may be no external fragmentation (unused space outside of records) after file updating.
- Individual records cannot always be updated in place.
- Algorithms for Accessing Delimited Variable Length Records
- Code for Accessing Delimited Variable Length Records
- Code for Accessing Variable Length Line Records
- Example (Delimiter = ASCII 30 (IE) = RS character:
- `0  66 69 72 73 74 20 6C 69 6E 65 1E 73 65 63 6F 6E  first line.secon`
- `10  64 20 6C 69 6E 65 1E                             d line.`
- Example (Delimiter = '\n'):
- `0  46 69 72 73 74 20 28 31 73 74 29 20 4C 69 6E 65  First (1st) Line`
- `10   D  A 53 65 63 6F 6E 64 20 28 32 6E 64 29 20 6C  ..Second (2nd) l`
- `20  69 6E 65  D  A                                   ine..`
- Disadvantage: the offset of each record cannot be calculated from its record number. This makes direct access impossible.
- Advantage: there is space overhead for the length prefix.
- Advantage: there will probably be no internal fragmentation (unusable space within records.)

Length Prefixed Variable Length Records

110	**Record 1**	**40**	**Record 2**	**100**	**Record 3**	**80**	**Record 4**	**70**	**Record 5**

- The records within a file are prefixed by a length byte or bytes.
- Records within a file can have different sizes.
- Different files can have different length records.
- Programs which access the file must know the size and format of the length prefix.
- Offset, or position, of the nth record of a file cannot be calculated.
- There is external overhead for record separation equal to the size of the length prefix per record.
- There should be no internal fragmentation (unused space within records.)

- There may be no external fragmentation (unused space outside of records) after file updating.
- Individual records cannot always be updated in place.
- Algorithms for Accessing Prefixed Variable Length Records
- Code for Accessing PreFixed Variable Length Records
- Example:

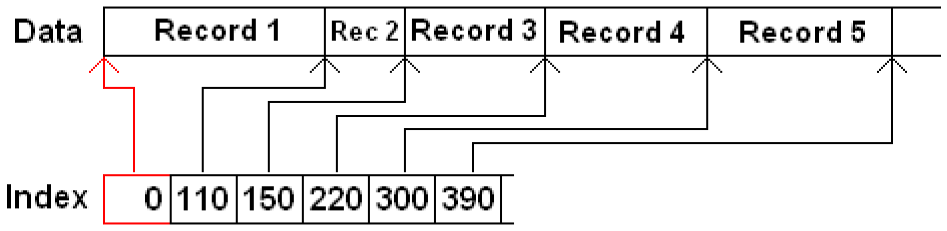
-
  ```
   0  A  0 46 69 72 73 74 20 4C 69 6E 65  B  0 53 65  ..First Line..Se
  ```

-
  ```
  10  63 6F 6E 64 20 4C 69 6E 65 1F  0 54 68 69 72 64  cond Line..Third
  ```

-
  ```
  20  20 4C 69 6E 65 20 77 69 74 68 20 6D 6F 72 65 20   Line with more
  ```

-
  ```
  30  63 68 61 72 61 63 74 65 72 73                     characters
  ```
- Disadvantage: the offset of each record can be calculated from its record number. This makes direct access possible.
- Disadvantage: there is space overhead for the delimiter suffix.
- Advantage: there will probably be no internal fragmentation (unusable space within records.)

Indexed Variable Length Records



- An auxiliary file can be used to point to the beginning of each record.
- In this case, the data records can be contiguous.
- If the records are contiguous, the only access is through the index file.
- Code for Accessing Indexed VariableLength Records
- Example:

  Index File:

-
  ```
   0  12  0  0  0 25  0  0  0 47  0  0  0          ....%...G...
  ```
-

  Data File:

-
  ```
   0  46 69 72 73 74 20 28 31 73 74 29 20 53 74 72 69  First (1st) Stri
  ```

-
  ```
  10  6E 67 53 65 63 6F 6E 64 20 28 32 6E 64 29 20 53  ngSecond (2nd) S
  ```

-
  ```
  20  74 72 69 6E 67 54 68 69 72 64 20 28 33 72 64 29  tringThird (3rd)
  ```

-
  ```
  30  20 53 74 72 69 6E 67 20 77 68 69 63 68 20 69 73   String which is
  ```

-
  ```
  40  20 6C 6F 6E 67 65 72                              longer
  ```

- Advantage: the offset of each record is be contained in the index, and can be looked up from its record number. This makes direct access possible.
- Disadvantage: there is space overhead for the index file.
- Disadvantage: there is time overhead for the index file.
- Advantage: there will probably be no internal fragmentation (unusable space within records.)
- The time overhead for accessing the index file can be minimized by reading the entire index file into memory when the files are opened.

Fixed Field Count Records

- Records can be recognized if they always contain the same (predetermined) number of fields.

Delineation of Fields in a Record

Fixed Length Fields

Field 1	**Field 2**	**Field 3**	**Field 4**	**Field 5**

- Each record is divided into fields of correspondingly equal size.
- Different fields within a record have different sizes.
- Different records can have different length fields.
- Programs which access the record must know the field lengths.
- There is no external overhead for field separation.
- There may be internal fragmentation (unused space within fields.)

Delimited Variable Length Fields

Field 1	**!**	**Field 2**	**!**	**Field 3**	**!**	**Field 4**	**!**	**Field 5**	**!**

- The fields within a record are followed by a delimiting byte or series of bytes.
- Fields within a record can have different sizes.
- Different records can have different length fields.
- Programs which access the record must know the delimiter.
- The delimiter cannot occur within the data.
- If used with delimited records, the field delimiter must be different from the record delimiter.
- There is external overhead for field separation equal to the size of the delimiter per field.
- There should be no internal fragmentation (unused space within fields.)

Length Prefixed Variable Length Fields

12	**Field 1**	**4**	**Field**	**10**	**Field 3**	**8**	**Field 4**	**7**	**Field 5**

			2						

- The fields within a record are prefixed by a length byte or bytes.
- Fields within a record can have different sizes.
- Different records can have different length fields.
- Programs which access the record must know the size and format of the length prefix.
- There is external overhead for field separation equal to the size of the length prefix per field.
- There should be no internal fragmentation (unused space within fields.)

Representing Record or Field Length

- Record or field length can be represented in either binary or character form.
- The length can be considered as another hidden field within the record.
- This length field can be either fixed length or delimited.
- When character form is used, a space can be used to delimit the length field.
- A two byte fixed length field could be used to hold lengths of 0 to 65535 bytes in binary form.
- A two byte fixed length field could be used to hold lengths of 0 to 99 bytes in decimal character form.
- A variable length field delimited by a space could be used to hold effectively any length.
- In some languages, such as strict Pascal, it is difficult to mix binary values and character values in the same file.
- The C++ language is flexible enough so that the use of either binary or character format is easy.

Tagged Fields

- Tags, in the form "Keyword=Value", can be used in fields.
- Use of tags does not in itself allow separation of fields, which must be done with another method.
- Use of tags adds significant space overhead to the file.
- Use of tags does add flexibility to the file structure.
- Fields can be added without affecting the basic structure of the file.
- Tags can be useful when records have sparse fields - that is, when a significant number of the possible attributes are absent.

Byte Order

- The byte order of integers (and floating point numbers) is not the same on all computers.
- This is hardware dependent (CPU), not software dependent.

- Many computers store numbers as might be expected: $40_{10} = 28_{16}$ is stored in a four byte integer as 00 00 00 28.
- PCs reverse the byte order, and store numbers with the least significant byte first: $40_{10} = 28_{16}$ is stored in a four byte integer as 28 00 00 00.
- On most computers, the number 40 would be stored in character form in its ASCII values: 34 30.
- IBM mainframe computers use EBCDIC instead of ASCII, and would store "40" as F4 F0.

Managing Files of Records

5.1 Record Access

5.1.1 Record Keys

key

A value which is contained within or associated with a record and which can be used to identify the record.

canonical form

A standard form for a key into which a nonstandard form of the key can be transformed algorithmically, for comparison purposes.

primary key

A key which uniquely identifies the records within a file.

secondary key

A search key other than the primary key.

- Search keys should be converted to canonical form before being used for a search.
- Examples: All upper case, all lower case, no dashes in phone number or SSN, etc.
- Primary keys should uniquely identify a single record.
- Ideally, the primary key should identify the entity to which a record corresponds, but not be an attribute of the identity. That is, a primary key should be *dataless.*
- Secondary keys are typically not unique.
- Algorithms and structures for handling secondary keys should not assume uniqueness.

5.1.2 A Sequential Search

sequential access

Accessing data from a file whose records are organized on the basis of their successive physical positions.

sequential search

A search which reads each record sequentially from the beginning until the record or records being sought are found.

- A sequential search is *O(n)*; that is, the search time is proportional to the number of items being searched.
- For a file of 1000 records and unique random search keys, an average of 500 records must be read to find the desired record.
- For an unsuccessful search, the entire file must be examined.
- Sequential is unsatisfactory for most file searches.
- Sequential search is satisfactory for certain special cases:
    - Sequential search is satisfactory for *small* files.
    - Sequential search is satisfactory for files that are searched only infrequently.
    - Sequential search is satisfactory when a high percentage of the records in a file will match.
    - Sequential search is required for unstructured text files.

5.1.3 Unix Tools for Sequential Processing

- Unix style tools for MS-DOS are available from the KU chat BBS website.
- Linux style tools for MS-DOS are available from the Cygwin website.
- The *cat* (con**cat**enate) utility can be used to copy files to standard output.
- The *cat* (con**cat**enate) utility can be used to combine (concatenate) two or more files into one.
- The *grep* (general regular expression) prints lines matching a pattern.
- The grep manual (man page) is available on line (Shadow Island.)
- The *wc* (word count) utility counts characters, words, and lines in a file.
- wc and other utilities are also available from National Taiwan University.

5.1.4 Direct Access

Accessing data from a file by record position with the file, without accessing intervening records.

relative record number

An ordinal number indicating the position of a record within a file.

- Direct access allows individual records to be read from different locations in the file without reading intervening records.

5.2 More about Record Structures

5.2.1 Choosing a Record Structure and Record Length

5.2.2 Header Records

header record: A record placed at the beginning of a file which contains information about the organization or the data of the file.

self describing file

A file which contains metadata describing the data within the file and its organization.

- Header records can be used to make a file self describing.
- 0 10 0 56 2 0 44 1C 0 0 0 0 0 0 0 0 0 ..V..D..........
- 10 2C 0 5 4E 61 6E 63 79 5 4A 6F 6E 65 73 D 31 ,..Nancy.Jones.1
- 20 32 33 20 45 6C 6D 20 50 6C 61 63 65 8 4C 61 6E 23 Elm Place.Lan
- 30 67 73 74 6F 6E 2 4F 4B 5 37 32 30 33 32 34 0 gston.OK.720324.
- 40 6 48 65 72 6D 61 6E 7 4D 75 6E 73 74 65 72 15 .Herman.Munster.
- 50 31 33 31 33 20 4D 6F 63 6B 69 6E 67 62 69 72 64 1313 Mockingbird
- 60 20 4C 61 6E 65 5 54 75 6C 73 61 2 4F 4B 5 37 Lane.Tulsa.OK.7
- 70 34 31 31 34 34 0 5 55 68 75 72 61 5 53 6D 69 41144..Uhura.Smi
- 80 74 68 13 31 32 33 20 54 65 6C 65 76 69 73 69 6F th.123 Televisio
- 90 6E 20 4C 61 6E 65 A 45 6E 74 65 72 70 72 69 73 n Lane.Enterpris
- A0 65 2 43 41 5 39 30 32 31 30 e.CA.90210
- The above dump represents a file with a 16 byte (10 00) header, Variable length records with a 2 byte length prefix, and fields delimited by ASCII code 28.

5.2.3 Adding Header Records to C++ Buffer Classes

file organization method

The arrangement and differentiation of fields and records within a file.

file-access method

The approach used to locate information in a file.

- File organization is static.
- Design decisions such as record format (fixed, variable, etc.) and field format (fixed, variable, etc.) determine file organization.
- File access is dynamic.
- File access methods include sequential and direct.
- File organization and file access are not functionally independent; for example, some file organizations make direct access impractical.

5.5 Beyond Record Structures

5.5.1 Abstract Data Models for File Access

5.3.2 Headers and Self-Describing Files

header record

A record placed at the beginning of a file which contains information about the organization or the data of the file.

self describing file

A file which contains metadata describing the data within the file and its organization.

- Files often begin with headers, which describe the data in the file, and the organization of the file.

File Header

Header
Data

- The header can contain such information as:
  - The record format (fixed, prefixed, delimited, etc.)
  - The field format (fixed, prefixed, delimited, etc.)
  - The names of each field.
  - The type of data in each field.
  - The size of each field.
  - Etc.
- Header records can be used to make a file self describing.

```
•   0  10  0 56  2  0 44 1C  0  0  0  0  0  0  0  0  0  ..V..D..........
•  10  2C  0  5 4E 61 6E 63 79  5 4A 6F 6E 65 73  D 31  ,..Nancy.Jones.1
•  20  32 33 20 45 6C 6D 20 50 6C 61 63 65  8 4C 61 6E  23 Elm Place.Lan
•  30  67 73 74 6F 6E  2 4F 4B  5 37 32 30 33 32 34  0  gston.OK.720324.
•  40   6 48 65 72 6D 61 6E  7 4D 75 6E 73 74 65 72 15  .Herman.Munster.
•  50  31 33 31 33 20 4D 6F 63 6B 69 6E 67 62 69 72 64  1313 Mockingbird
•  60  20 4C 61 6E 65  5 54 75 6C 73 61  2 4F 4B  5 37   Lane.Tulsa.OK.7
•  70  34 31 31 34 34  0  5 55 68 75 72 61  5 53 6D 69  41144..Uhura.Smi
•  80  74 68 13 31 32 33 20 54 65 6C 65 76 69 73 69 6F  th.123 Televisio
•  90  6E 20 4C 61 6E 65  A 45 6E 74 65 72 70 72 69 73  n Lane.Enterpris
   A0  65  2 43 41  5 39 30 32 31 30                    e.CA.90210
```

- The above dump represents a file with a 16 byte (10 00) header, Variable length records with a 2 byte length prefix, and fields delimited by ASCII code 28 ($1C_{16}$). The actual data begins at byte 16 (10_{16}).

5.3.3 Metadata

metadata

Data which describes the data in a file or table.

5.3.4 Mixing Object Types in One File

5.3.5 Representation Independent File Access

5.3.6 Extensibility

extensibility

Having the ability to be extended (e.g., by adding new fields) without redesign.

5.4 Portability and Standardization

portability

The ability to be easily accessed by different systems and applications.

5.4.1 Factors Affecting Portability

5.4.2 Achieving Portability

File Access

- File organization is static.
- File access is dynamic.

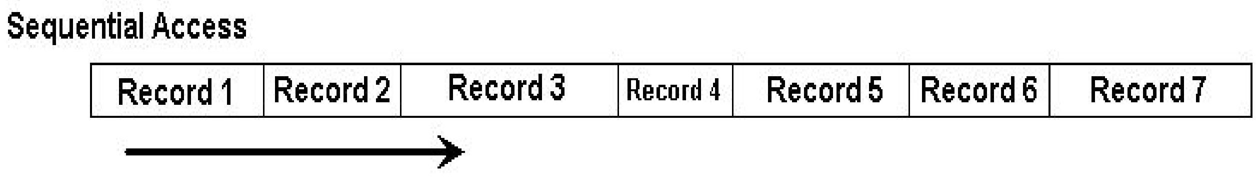
Sequential Access

sequential access

Access of data in order.

Accessing data from a file whose records are organized on the basis of their successive physical positions.

- Sequential access processes a file from its beginning.



- All operating systems support sequential access of files.
- Sequential access is the fastest way to read or write all of the records in a file.
- Sequential access is slow when reading a singlt random record, since all the preceeding records must be read.

Direct Access

direct access

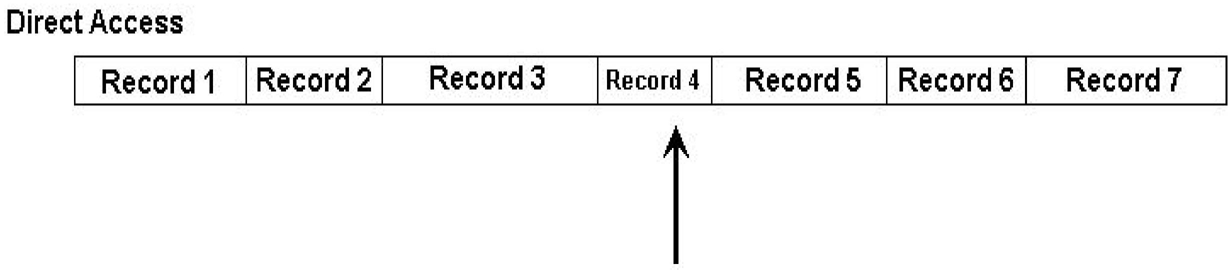
Access of data in arbitrary order, with variable access time.

Accessing data from a file by record position with the file, without accessing intervening records.

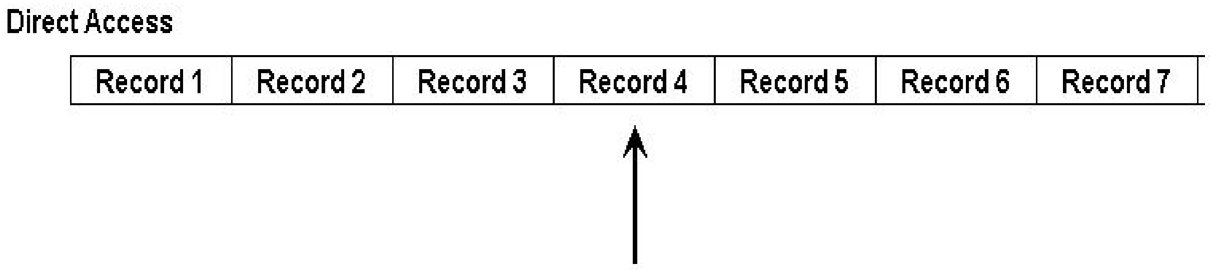
relative record number

An ordinal number indicating the position of a record within a file.

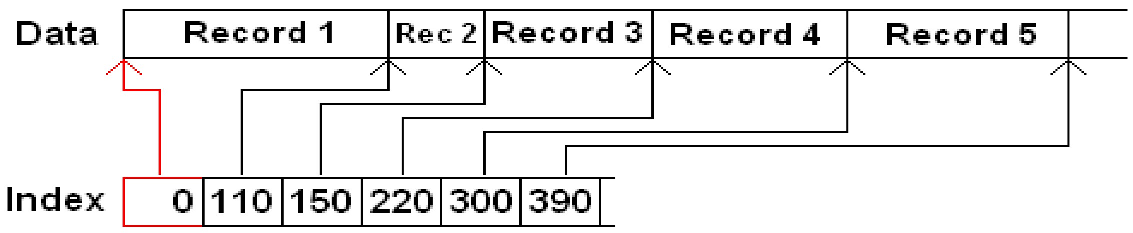
- Direct access processes single records at a time by position in the file.



- Mainfreme and midrange operating systems support direct access of files.
- The Windows, DOS, UNIX, and Linux operating systems do not natively support direct access of files.
- When using Windows, DOS, UNIX, and Linux operating systems applications must be programmed to use direct access.
- Direct access is slower than sequential when reading or writing all of the records in a file.
- Direct access is fast when reading a singlt random record, since the preceeding records are ignored.
- Direct access allows individual records to be read from different locations in the file without reading intervening records.
- When files are organized with fixed length records, the location of a record in a file can be calculated from its relative record number, and the file can be accessed using the seek functions.



- ByteOffset = (RRN - 1) × RecLen
- When files have variable length records supported by an index, the records can be accessed directly through the index, with the use of the seek function.



- For direct access to be useful, the relative record number of the record of interest must be known.
- Direct access is often used to support keyed access.

Keyed Access

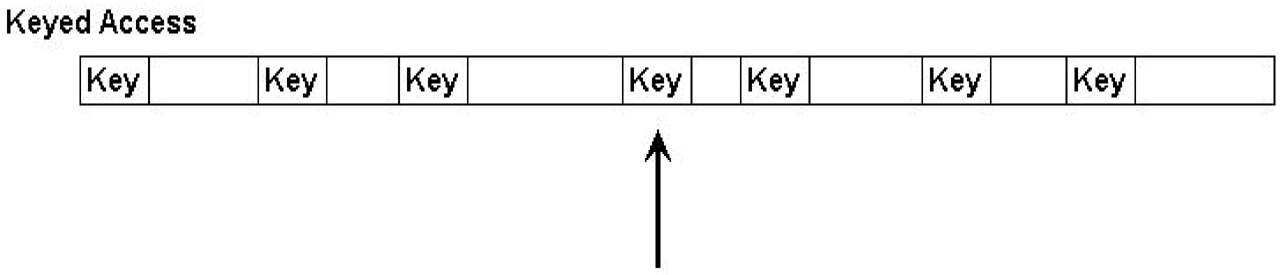
keyed access

Accessing data from a file by an alphanumeric key associated with each record.

key

A value which is contained within or associated with a record and which can be used to identify the record.

- Keyed access processes single records at a time by record key.



- Mainfreme and midrange operating systems support keyed access of files.
- The Windows, DOS, UNIX, and Linux operating systems do not natively support keyed access of files.
- When using Windows, DOS, UNIX, and Linux operating systems applications must be programmed to use keyed access.
- Keyed access will be covered in more detail in later chapters.

metadata

Data which describes the data in a file or table.

UNIT- 3

Organizing Files for Performance, Indexing

6.1. Data Compression

- Compression can reduce the size of a file, improving performance.
- File maintenance can produce fragmentation inside of the file. There are ways to reuse this space.
- There are better ways than sequential search to find a particular record in a file.
- Keysorting is a way to sort medium size files.
- We have already considered how important it is for the file system designer to consider how a file is to be accessed when deciding how to create fields, records, and other file structures. In this chapter, we continue to focus on file organization, but the motivation is different. We look at ways to organize or reorganize files in order to improve performance.
- In the first section, we look at how to organize files to make them smaller. Compression techniques make file smaller by encoding them to remove redundant or unnecessary information.

data compression

The encoding of data in such a way as to reduce its size.

redundancy reduction

Any form of compression which removes only redundant information.

- In this section, we look at ways to make files smaller, using data compression. As with many programming techniques, there are advantages and disadvantages to data compression. In general, the compression must be reversed before the information is used. For this tradeoff,
  - Smaller files use less storage space.
  - The transfer time of disk access is reduced.
  - The transmission time to transfer files over a network is reduced.

  But,
  - Program complexity and size are increased.
  - Computation time is increased.
  - Data portability may be reduced.
  - With some compression methods, information is unrecoverably lost.
  - Direct access may become prohibitably expensive.
  - Data compression is possible because most data contains redundant (repeated) or unnecessary information.

- Data compression is possible because most data contains redundant (repeated) or unnecessary information. Reversible compression removes only redundant information, making it possible to restore the data to its original form. Irreversible compression goes further, removing information which is not actually necessary, making it impossible to recover the original form of the data.
- Next we look at ways to reclaim unused space in files to improve performance. Compaction is a batch process that we can use to purge holes of unused space from a file that has undergone many deletions and updates. Then we investigate dynamic ways to maintain performance by reclaiming space made available by deletions and updates of records during the life of the file.

6.1.1 Compact Notation

The replacement of field values with an ordinal number which index an enumeration of possible field values.

- Compact notation can be used for fields which have an effectively fixed range of values.
- Compact notation can be used for fields which have an effectively fixed range of values. The *State* field of the *Person*record, as used earler, is an example of such a field. There are 676 (26 x 26) possible two letter abbreviations, but there are only 50 states. By assigning an ordinal number to each state, and storing the code as a one byte binary number, the field size is reduced by 50 percent.
- No information has been lost in the process. The compression can be completely reversed, replacing the numeric code with the two letter abbreviation when the file is read. Compact notation is an example of redundancy reduction.
- On the other hand, programs which access the compressed data will need additional code to compress and expand the data. An array can used as a translation table to convert between the numeric codes and the letter abbreviations. The translation table can be coded within the program, using literal constants, or stored in a file which is read into the array by the program.
- Since a file using compact notation contains binary data, it cannot be viewed with a text editor, or typed to the screen. The use of delimited records is prohibitively expensive, since the delimiter will occur in the compacted field.

6.2.2 Run Length Encoding

run-length encoding

An encoding scheme which replaces runs of a single symbol with the symbol and a repetition factor.

- Run-length encoding is useful only when the text contains long runs of a single value.
- Run-length encoding is useful for images which contain solid color areas.

- Run-length encoding may be useful for text which contains strings of blanks.
- Example:
- uncompressed text (hexadecimal format):
- 40 40 40 40 40 40 43 43 41 41 41 41 41 42
- compressed text (hexadecimal format):
- FE 06 40 43 43 FE 05 41 42

  where FE is the compression escape code, followed by a length byte, and the byte to be repeated.

6.2.3 Variable Length Codes

variable length encoding

An encoding scheme in which the codes for differenct symbols may be of different length.

huffman code

A variable length code, in which each code is determined by the occurence frequency of the corresponding symbol.

prefix code

A variable length code in which the length of a code element can be determined from the first bits of the code element.

- The optimal Huffman code can be different for each source text.
- Huffman encoding takes two passes through the source text: one to build the code, and a second to encode the text by applying the code.
- The code must be stored with the compressed text.
- Huffman codes are based on the frequency of occurrence of characters in the text being encoded.
- The characters with the highest occurence frequency are assigned the shortest codes, minimizing the average encoded length.
- Huffman code is a prefix code.
- Example:
- Uncompressed Text:
- abdeacfaag (80 bits)
- Frequencies: a 4 e 1
- b 1 f 1
- c 1 g 1
- d 1
- code: a 1 e 0001
- b 010 f 0010

- c 011 g 0011
- d 0000
- Compressed Text (binary):
- 1010000000110110010110011 (26 bits)
- Compressed Text (hexadecimal):
- A0 1B 96 60

6.2.4 Irreversible Compression Techniques

irreversible compression

Any form of compression which reduces information.

reversible compression

Compression with no alteration of original information upon reconstruction.

- Irreversible compression goes beyond redundancy reduction, removing information which is not actually necessary, making it impossible to recover the original form of the data.
- Irreversible compression is useful for reducing the size of graphic images.
- Irreversible compression is used to reduce the bandwidth of audio for digital recording and telecommunications.
- JPEG image files use an irreversible compression based on cosine transforms.
- The amount of information removed by JPEG compression is controllable.
- The more information removed, the smaller the file.
- For photographic images, a significant amount of information can be removed without noticably affecting the image.
- For line graphic images, the JPEG compression may introduce aliasing noise.
- GIF images files irreversibly compress images which contain more than 256 colors.
- The GIF format only allows 256 colors.
- The compression of GIF formatting is reversible for images which have fewer than 256 colors, and lossy for images which have more than 256 colors.
- Recommendation:
  - Use JPEG for photographic images.
  - Use GIF for line drawings.

6.2.5. Compression in UNIX

- The UNIX *pack* and *unpack* utilities use Huffman encoding.
- The UNIX *compress* and *uncompress* utilities use Lempil-Ziv encoding.
- Lempil-Ziv is a variable length encoding which replaces strings of characters with numbers.
- The length of the strings which are replaced increases as the compression advances through the text.

- Lempel-Ziv compression does not store the compression table with the compressed text. The compression table can be reproduced during the decompression process.
- Lempel-Ziv compression is used by "zip" compression in DOS and Windows.
- Lempel-Ziv compression is a redundancy reduction compression - it is completely reversible, and no information is lost.
- The ZIP utilities actually support several types of compression, including Lempil-Ziv and Huffman.

6.3 Reclaiming Space in Files

6.3.1 Record Deletion and Storage Compaction

external fragmentation

Fragmentation in which the unused space is outside of the allocated areas.

compaction

The removal of fragmentation from a file by moving records so that they are all physically adjacent.

- As files are maintained, records are added, updated, and deleted.
- The problem: as records are deleted from a file, they are replaced by unused spaces within the file.
- The updating of variable length records can also produce fragmentation.
- Compaction is a relatively slow process, especially for large files, and is not routinely done when individual records are deleted.

6.3.2 Deleting Fixed-Length Records for Reclaiming Space Dynamically

linked list

A container consisting of a series of nodes, each containing data and a reference to the location of the logically next node.

avail list

A list of the unused spaces in a file.

stack

A last-in first-out container, which is accessed only at one end.

Record 1	**Record 2**	**Record 3**	**Record 4**	**Record 5**

- Deleted records must be marked so that the spaces will not be read as data.
- One way of doing this is to put a special character, such as an asterisk, in the first byte of the deleted record space.

Record 1	**Record 2**	*	**Record 4**	**Record 5**

- If the space left by deleted records could be reused when records are added, fragmentation would be reduced.
- To reuse the empty space, there must be a mechanism for finding it quickly.

- One way of managing the empty space within a file is to organize as a linked list, known as the *avail list*.
- The location of the first space on the avail list, the head pointer of the linked list, is placed in the header record of the file.
- Each empty space contains the location of the next space on the avail list, except for the last space on the list.
- The last space contains a number which is not valid as a file location, such as -1.
-

Header	**Slot 1**	**Slot 2**	**Slot 3**	**Slot 4**	**Slot 5**	**Slot 6**
3	*** -1**	**Record 2**	*** 1**	**Record 4**	**Record 5**	**Record 6**

- If the file uses fixed length records, the spaces are interchangable; any unused space can be used for any new record.
- The simplest way of managing the avail list is as a stack.
- As each record is deleted, the old list head pointer is moved from the header record to the deleted record space, and the location of the deleted record space is placed in the header record as the new avail list head pointer, pushing the new space onto the stack.

Header	**Slot 1**	**Slot 2**	**Slot 3**	**Slot 4**	**Slot 5**	**Slot 6**
5	*** -1**	**Record 2**	*** 1**	**Record 4**	*** 3**	**Record 6**

- When a record is added, it is placed in the space which is at the head of the avail list.
- The push process is reversed; the empty space is popped from the stack by moving the pointer in the first space to the header record as the new avail list head pointer.

Header	**Slot 1**	**Slot 2**	**Slot 3**	**Slot 4**	**Slot 5**	**Slot 6**
3	*** -1**	**Record 2**	*** 1**	**Record 4**	**Record 5**	**Record 6**

- With fixed length records, the relative record numbers (RRNs) can be used as location pointers in the avail list.

6.3.3. Deleting Variable-Length Records

- If the file uses variable length records, the spaces not are interchangable; a new record will not fit just any unused space.
- With variable length records, the byte offset of each record can be used as location pointers in the avail list.
- The size of each deleted record space should also be placed in the space.

6.3.4. Storage Fragmentation

coalescence

The combination of two (or more) physically adjacent unused spaces into a single unused space.

internal fragmentation

Fragmentation in which the unused space is within the allocated areas.

6.3.5. Placement Strategies

placement strategy

A policy for determining the location of a new record in a file.

first fit

A placement strategy which selects the first space on the free list which is large enough.

best fit

A placement strategy which selects the smallest space from the free list which is large enough.

worst fit

A placement strategy which selects the largest space from the free list (if it is large enough.)

First Fit

Header	Slot @50	Slot @120	Slot @200	Slot @300	Slot @370	Slot @430
370	* -1 70	Record	* 50 100	Record	* 200 60	Record

- The simplest placement strategy is *first fit*.
- With first fit, the spaces on the avail list are scanned in their logical order on the avail list.
- The first space on the list which is big enough for a new record to be added is the one used.
- The used space is delinked from the avail list, or, if the new record leaves unused space, the new (smaller) space replaces the olf space.
- Adding a 70 byte record, only the first two entries on the list are checked:

Header	Slot @50	Slot @120	Slot @200	Slot @230	Slot @300	Slot @370	Slot @430
370	* -1 70	Record	* 5030	Record	Record	* 200 60	Record

- As records are deleted, the space can be added to the head of the list, as when the list is managed as a stack.

Best Fit

- The *best fit* strategy leaves the smallest space left over when the new record is added.
- There are two possible algorithms:
    1. Manage deletions by adding the new record space to the head of the list, and scan the entire list for record additions.
    2. Manage the avail list as a sorted list; the first fit on the list will then be the best fit.
- Best Fit, Sorted List:

Header	**Slot @50**	**Slot @120**	**Slot @200**	**Slot @300**	**Slot @370**	**Slot @430**
370	*** 200 70**	**Record**	*** -1 100**	**Record**	*** 50 60**	**Record**

- Adding a 65 byte record, only the first two entries on the list are checked:

Header	**Slot @50**	**Slot @120**	**Slot @200**	**Slot @300**	**Slot @370**	**Slot @430**
370	**Record**	**Record**	*** -1 100**	**Record**	*** 20060**	**Record**

- Best Fit, Unsorted List:

Header	**Slot @50**	**Slot @120**	**Slot @200**	**Slot @300**	**Slot @370**	**Slot @430**
200	*** 370 70**	**Record**	*** 50 100**	**Record**	*** -1 60**	**Record**

- Adding a 65 byte record, all three entries on the list are checked:

Header	**Slot @50**	**Slot @120**	**Slot @200**	**Slot @300**	**Slot @370**	**Slot @430**
200	**Record**	**Record**	*** 370 100**	**Record**	*** -1 60**	**Record**

- The 65 byte record has been stored in a 70 byte space; rather than create a 5 byte external fragment, which would be useless, the 5 byte excess has become internal fragmentation within tbe record.

Worst Fit

- The *worst fit* strategy leaves the largest space left over when the new record is added.
- The rational is that the leftover space is most likely to be usable for another new record addition.

- There are two possible algorithms:
  1. Manage deletions by adding the new record space to the head of the list, and scan the entire list for record additions.
  2. Manage the avail list as a reverse sorted list; the first fit on the list, which will be the first entry, will then be the worst fit.
- Worst Fit, Sorted List:

Header	**Slot @50**	**Slot @120**	**Slot @200**	**Slot @300**	**Slot @370**	**Slot @430**
200	*** 370 70**	**Record**	*** 50 100**	**Record**	*** -1 60**	**Record**

- Adding a 65 byte record, only the first entry on the list are checked:

Header	**Slot @50**	**Slot @120**	**Slot @200**	**Slot @235**	**Slot @300**	**Slot @370**	**Slot @430**
50	*** 370 70**	**Record**	*** -1 35**	**Record**	**Record**	*** 20060**	**Record**

- Worst Fit, Unsorted List:

Header	**Slot @50**	**Slot @120**	**Slot @200**	**Slot @300**	**Slot @370**	**Slot @430**
200	*** -1 70**	**Record**	*** 370 100**	**Record**	*** 50 60**	**Record**

- Adding a 65 byte record, all three entries on the list are checked:

Header	**Slot @50**	**Slot @120**	**Slot @200**	**Slot @235**	**Slot @300**	**Slot @370**	**Slot @430**
200	*** -1 70**	**Record**	*** 37035**	**Record**	**Record**	*** 50 60**	**Record**

- **Commonalities**
- Regardless of placement strategy, when the record does not match the slot size (as is usually the case), there are two possible actions:
  1. Create a new empty slot with the extra space, creating external fragmentation.
  2. Embed the extra space in the record, creating internal fragmentation.

6.4 Finding Things Quickly: An Introduction to Internal Sorting and Binary Searching

6.4.1 Finding Things in Simple Field and Record Files

6.4.2 Search by Guessing: Binary Search

binary search

A search which can be applied to an ordered linear list to progressively divide the possible scope of a search in half until the search object is found.

- Example: Search for 'M' in the list:
- A B C D E F G H I J K L M N O P Q R S
- Compare 'M' to the middle li in the list:
- A B C D E F G H I **J** K L M N O P Q R S
- 'M' > 'J': Narrow the search to the last half. (Eliminate the first half.)
- A B C D E F G H I J **K L M N O P Q R S**
- Compare 'M' to the middle li in the remainder of the list:
- A B C D E F G H I J **K L M N O P Q R S**
- 'M' < 'O': Narrow the search to the first half of the remainder. (Eliminate the last half.)
- A B C D E F G H I J **K L M N** O P Q R S
- Compare 'M' to the middle li in the remainder of the list:
- A B C D E F G H I J **K L M N** O P Q R S
- 'M' > 'L': Narrow the search to the last half. (Eliminate the first half.)
- A B C D E F G H I J K L **M N** O P Q R S
- Compare 'M' to the middle li in the remainder of the list:
- A B C D E F G H I J K L **M N** O P Q R S
- 'M' == 'M': The search is over.

6.4.3 Binary Search versus Sequential Search

- A binary search of *n* items requires $\lfloor \log_2 n \rfloor + 1$ comparisons at most.
- A binary search of *n* items requires $\lfloor \log_2 n \rfloor + 1/2$ comparisons on average.
- Binary searching is $O(\log_2 n)$.
- Sequential search of *n* items requires n comparisons at most.
- A successful sequential search of *n* items requires n / 2 comparisons on average.
- A unsuccessful sequential search of *n* items requires n comparisons.
- Sequential searching is $O(n)$.
- Binary searching is only possible on ordered lists.

6.4.4 Sorting a Disk File in Memory

internal sort

A sort performed entirely in main memory.

- The algorithms used for internal assume fast random access, and are not suitable for sorting files directly.
- A small file which can be entirely read into memory can be sorted with an internal sort, and then written back to a file.

6.4.5 The limitations of Binary Searching and Internal Sorting

- Binary searching requires more than one or two accesses.

- More than one or two accesses is too many.
- Keeping a file sorted is very expensive.
- An internal sort works only on small files.

6.5 Keysorting

keysort

A sort performed by first sorting keys, and then moving records.

6.5.1 Description of the Method

- Read each record sequentially into memory, one by one
- George Washington 1789 1797 None
- John Adams 1797 1801 Fed
- Thomas Jefferson 1801 1809 DR
- James Madison 1809 1817 DR
- James Monroe 1817 1825 DR
- John Q Adams 1825 1829 DR
- Andrew Jackson 1829 1837 Dem
- Martin Van Buren 1837 1841 Dem
- William Harrison 1841 1841 Whig
- Save the key of the record, and the location of the record, in an array (*KEYNODES*).
- Washington George 1
- Adams John 2
- Jefferson Thomas 3
- Madison James 4
- Monroe James 5
- Adams John Q 6
- Jackson Andrew 7
- Van Buren Martin 8
- Harrison William 9
- After all records have been read, internally sort the *KEYNODES* array of record keys and locations.
- Adams John 2
- Adams John Q 6
- Harrison William 9
- Jackson Andrew 7
- Jefferson Thomas 3
- Madison James 4
- Monroe James 5

- Van Buren Martin 8
- Washington George 1
- Using the *KEYNODES* array, read each record back into memory a second time using direct access.
- Write each record sequentially into a sorted file.
- John Adams 1797 1801 Fed
- John Q Adams 1825 1829 DR
- William Harrison 1841 1841 Whig
- Andrew Jackson 1829 1837 Dem
- Thomas Jefferson 1801 1809 DR
- James Madison 1809 1817 DR
- James Monroe 1817 1825 DR
- Martin Van Buren 1837 1841 Dem
- George Washington 1789 1797 None

6.5.2 Limitations of the Keysort Method

- Keysort is only possible when the *KEYNODES* array is small enough to be held in memory.
- Each record must be read twice: once sequentially and once directly.
- The direct reads each require a seek.
- If the original file and the output file are on the same physical drive, there will also be a seek for each write.
- Keysorting is a way to sort medium size files.

6.5.3 Another Solution: Why Bother to Write the File Back?

- Rather than actually sorting the data file, the *KEYNODES* array can be written to a disk file (sequentially), and will function as an index.
- The next chapter examines indexes.

6.5.4 Pinned Records

pinned record

A record which cannot be moved without invalidating existing references to its location.

Indexing

7.1 What is an Index?

index

A structure containing a set of entries, each consisting of a key field and a reference field, which is used to locate records in a data file.

key field

The part of an index which contains keys.

reference field

The part of an index which contains information to locate records.

- An index imposes order on a file without rearranging the file.
- Indexing works by indirection.

7.2 A Simple Index for Entry-Sequenced Files

simple index

An index in which the entries are a key ordered linear list.

- Simple indexing can be useful when the entire index can be held in memory.
- Changes (additions and deletions) require both the index and the data file to be changed.
- Updates affect the index if the key field is changed, or if the record is moved.
- An update which moves a record can be handled as a deletion followed by an addition.

7.3 Using Template Classes in C++ for Object I/O

7.4 Object Oriented Support for Indexed, Entry Sequenced Files

entry-sequenced file

A file in which the record order is determined by the order in which they are entered.

- The physical order of records in the file may not be the same as order of entry, because of record deletions and space reuse.
- The index should be read into memory when the data file is opened.

7.5 Indexes That are too Large to Hold in Memory

- Searching of a simple index on disk takes too much time.
- Maintaining a simple index on disk in sorted order takes too much time.
- Tree structured indexes such as B-trees are a scalable alternative to simple indexes.
- Hashed organization is an alternative to indexing when only a primary index is needed.

7.6 Indexing to Provide Access by Multiple Keys

secondary key

A search key other than the primary key.

secondary index

An index built on a secondary key.

- Secondary indexes can be built on any field of the data file, or on combinations of fields.
- Secondary indexes will typically have multiple locations for a single key.
- Changes to the data may now affect multiple indexes.

- The reference field of a secondary index can be a direct reference to the location of the entry in the data file.
- The reference field of a secondary index can also be an indirect reference to the location of the entry in the data file, through the primary key.
- Indirect secondary key references simplify updating of the file set.
- Indirect secondary key references increase access time.

7.7 Retrieval Using Combinations of Secondary Keys

- The search for records by multiple keys can be done on multiple index, with the combination of index entries defining the records matching the key combination.
- If two keys are to be combined, a list of entries from each key index is retrieved.
- For an "or" combination of keys, the lists are merged.
- I.e., any entry found in either list matches the search.
- For an "and" combination of keys, the lists are matched.
- I.e., only entries found in both lists match the search.

7.8 Improving the Secondary Index Structure: Inverted Lists

inverted list

An index in which the reference field is the head pointer of a linked list of reference items.

7.9 Selective Indexes

selective index

An index which contains keys for only part of the records in a data file.

7.10 Binding

binding

The association of a symbol with a value.

locality

A condition in which items accessed temporally close are also physically close.

UNIT-4

Cosequential Processing and the Sorting of Large Files

8.1 An Object Oriented Model for Implementing Cosequential Processes

cosequential operations

Operations which involve accessing two or more input files sequentially and in parallel, resulting in one or more output files produced by the combination of the input data.

8.1.1 Considerations for Cosequential Algorithms

- Initialization - What has to be set up for the main loop to work correctly.
- Getting the next item on each list - This should be simple and easy, from the main algorithm.
- Synchronization - Progress of access in the lists should be coordinated.
- Handling End-Of-File conditions - For a match, processing can stop when the end of any list is reached.
- Recognizing Errors - Items out of sequence can "break" the synchronization.

8.1.2 Matching Names in Two Lists

match

The process of forming a list containing all items common to two or more lists.

8.1.3 Cosequential Match Algorithm

- Initialize (open the input and output files.)
- Get the first item from each list.
- While there is more to do:
  - Compare the current items from each list.
  - If the items are equal,
    - Process the item.
    - Get the next item from each list.
    - Set *more* to true iff none of this lists is at end of file.
  - If the item from list *A* is less than the item from list *B*,
    - Get the next item from list *A*.
    - Set *more* to true iff list *A* is not at end-of-file.
  - If the item from list *A* is more than the item from list *B*,
    - Get the next item from list *B*.
    - Set *more* to true iff list *B* is not at end-of-file.
- Finalize (close the files.)

8.1.4 Cosequential Match Code

- void Match (char * InputName1,
-         char * InputName2,

```
          char * OutputName) {
/* Local Declarations                    */
OrderedFile Input1;
OrderedFile Input2;
OrderedFile Output;
int Item1;
int Item2;
int more;

/* Initialization                        */
cout << "Data Matched:" << endl;
Input1.open (InputName1, ios::in);
Input2.open (InputName2, ios::in);
Output.open (OutputName, ios::out);

/* Algorithm                             */
Input1 >> Item1;
Input2 >> Item2;
more = Input1.good() && Input2.good();
while (more) {
  cout << Item1 << ':' << Item2 << " => " << flush;  /* DEMO only */
  if (Item1 < Item2) {
    Input1 >> Item1;
    cout << '\n';                        /* DEMO only */
  } else if (Item1 > Item2) {
    Input2 >> Item2;
    cout << '\n';                        /* DEMO only */
  } else {
    Output << Item1 << endl;
    cout << Item1 << endl;               /* DEMO only */
    Input1 >> Item1;
    Input2 >> Item2;
  }
  more = Input1.good() && Input2.good();
}

```

```
•   /* Finalization                    */
•   Input1.close ();
•   Input2.close ();
•   Output.close ();
•  }
•
```

8.1.5 OrderedFile Class Declaration

```
•  class OrderedFile : public fstream {
•   public:
•     void        open       (char * Name, int Mode);
•     int         good       (void);
•     OrderedFile & operator >> (int & Item);
•   private:
•     int last;
•     static int HighValue;
•     static int LowValue;
•  };
```

8.1.6 OrderedFile Class Implementation

```
•  int OrderedFile::HighValue = INT_MAX;
•  int OrderedFile::LowValue = INT_MIN;
•  void OrderedFile :: open (char * Name, int Mode) {
•   fstream :: open (Name, Mode);
•   last = LowValue;
•  }
•
•  OrderedFile & OrderedFile :: operator >> (int & Item) {
•   fstream::operator >> (Item);
•   if (eof()) {
•     Item = HighValue;
•   } else if (Item < last) {
•     Item = HighValue;
•     cerr << "Sequence Error\n";
•   }
•   last = Item;
•   return *this;
•  }
```

```
•
• int OrderedFile :: good () {
•   return fstream::good() && (last != HighValue);
• }
•
```

8.1.7 Match *main* Function

```
• int main (int, char * []) {
•   /* Local Declarations                    */
•   char OutputName [50];
•   char InputName2 [50];
•   char InputName1 [50];
•
•   /* Initialization                        */
•   cout << "Input name 1?  ";
•   cin >> InputName1;
•   cout << "Input name 2?  ";
•   cin >> InputName2;
•   cout << "Output name?  ";
•   cin >> OutputName;
•
•   /* Algorithm                             */
•   Match (InputName1, InputName2, OutputName);
•
•   /* Report to system                      */
•   return 0;
• }
•
```

8.1.8 Merging Two Lists

merge

The process of forming a list containing all items in any of two or more lists.

8.1.9 Cosequential Merge Algorithm

- Initialize (open the input and output files.)
- Get the first item from each list.
- While there is more to do:
  - Compare the current items from each list.
  - If the items are equal,

        - Process the item.
        - Get the next item from each list.
    - If the item from list *A* is less than the item from list *B*,
        - Process the item from list *A*.
        - Get the next item from list *A*.
    - If the item from list *A* is more than the item from list *B*,
        - Process the item from list *B*.
        - Get the next item from list *B*.
    - Set *more* to false iff all of this lists are at end of file.
- Finalize (close the files.)

8.1.10 Cosequential Merge Code

```
void Merge (char * InputName1,
            char * InputName2,
            char * OutputName) {
 /* Local Declarations                    */
 OrderedFile Input1;
 OrderedFile Input2;
 OrderedFile Output;
 int Item1;
 int Item2;
 int more;

 /* Initialization                    */
 cout << "Data Merged:" << endl;
 Input1.open (InputName1, ios::in);
 Input2.open (InputName2, ios::in);
 Output.open (OutputName, ios::out);

 /* Algorithm                         */
 Input1 >> Item1;
 Input2 >> Item2;
 more = Input1.good() || Input2.good();
 while (more) {
   cout << Item1 << ':' << Item2 << " => " << flush;  /* DEMO only */
   if (Item1 < Item2) {
     Output << Item1 << endl;
•      cout << Item1 << endl;           /* DEMO only */
•      Input1 >> Item1;
•    } else if (Item1 > Item2) {
•      Output << Item2 << endl;
•      cout << Item2 << endl;           /* DEMO only */
•      Input2 >> Item2;
•    } else {
•      Output << Item1 << endl;
•      cout << Item1 << endl;           /* DEMO only */
•      Input1 >> Item1;
•      Input2 >> Item2;
•    }
•    more = Input1.good() || Input2.good();
•   }
•
•   /* Finalization                  */
•   Input1.close ();
•   Input2.close ();
•   Output.close ();
•  }
•
```

8.1.11 Merge *main* Function

```
•  int main (int, char * []) {
•   /* Local Declarations                 */
•   char OutputName [50];
•   char InputName2 [50];
•   char InputName1 [50];
•
•   /* Initialization                  */
•   cout << "Input name 1?  ";
•   cin >> InputName1;
•   cout << "Input name 2?  ";
•   cin >> InputName2;
•   cout << "Output name?  ";
•   cin >> OutputName;
•
```

- /* Algorithm */
- Merge (InputName1, InputName2, OutputName);
-
- /* Report to system */
- return 0;
- }
-

8.1.12 General Cosequential Algorithm

- Initialize (open the input and output files.)
- Get the first item from each list
- While there is more to do:
  - Compare the current items from each list
  - Based on the comparison, appropriately process one or all items.
  - Get the next item or items from the appropriate list or lists.
  - Based on the whether there were more items, determine if there is more to do.
- Finalize (close the files.)

high value

A value greater than any valid key.

low value

A value less than any valid key.

sequence checking

Verification of correct order.

synchronization loop

The main loop of the cosequential processing model, which is responsible for synchronizing inputs.

8.1.13 Cosequential Algorithm Summary

- Two or more input files are to be processed in a parallel fashion to produce one or more output files.
  - In some cases, an output file may be the same as one of the input files.
- Each file is sorted on one or more key fields,and all files are ordered in the same way on the same fields.
  - It is not necessary that all files have the same record structure
- In some cases, a HighValue must exist which is greater thanall valid key values, and a LowValue must exist which is less than all valid key values.

    - This is not an absolute necessity, but it does allow the algorithms to be simplified.
- Records are to be processed in logical sorted order.
    - Physical ordering is not necessary, but does improve efficiency significantly.
- For each file, only one current record is consideredin the main loop.
    - Subprograms may look ahead or back.
- Records can only be manipulated in main memory.
    - This is true for all file processing.

8.2 Application of the Model to a General Ledger Program

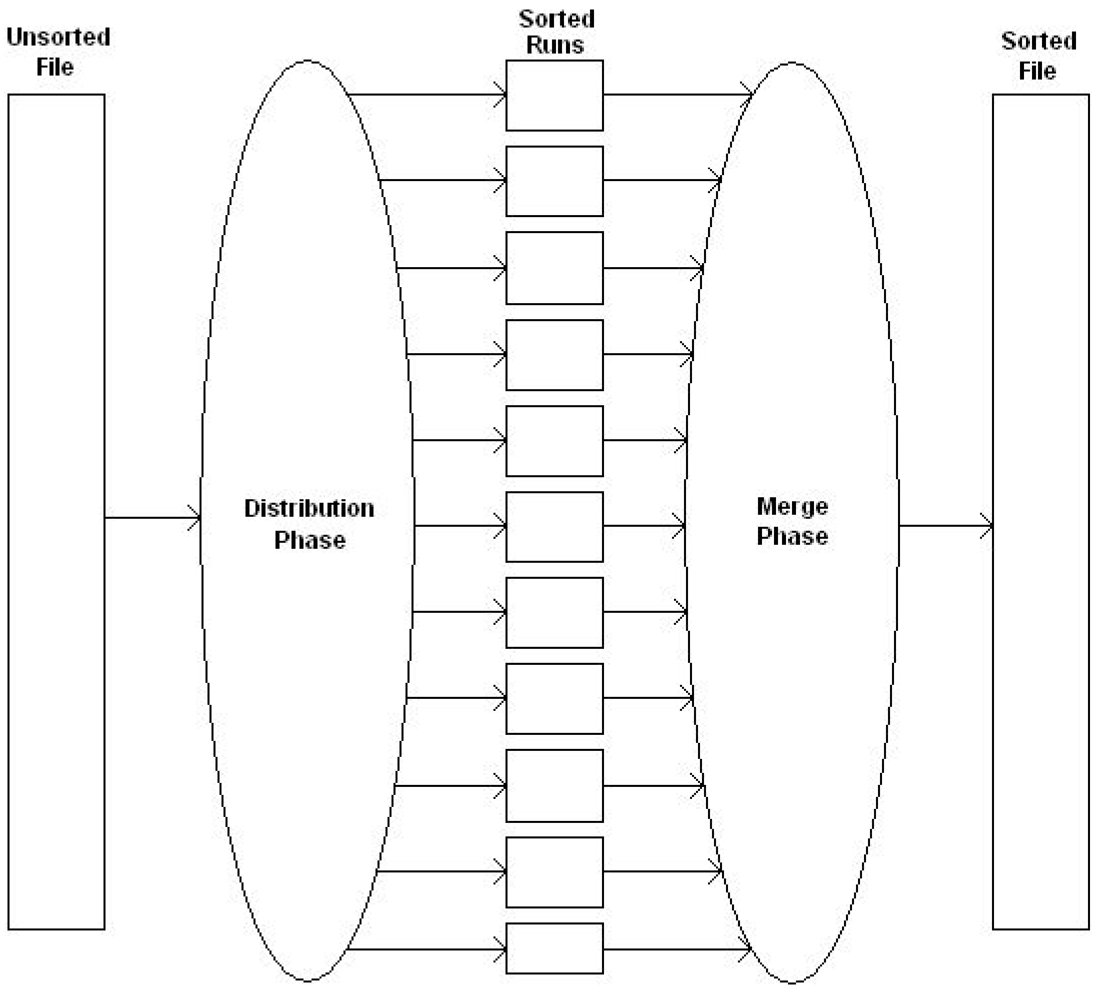
Transaction Posting

8.3 Extension of the Model to Include Multiway Merging

selection tree

A binary tree in which the parent of two nodes is the lesser (or greater) of each pair of nodes, making the root the least (or greatest.)

8.4 External Sorting

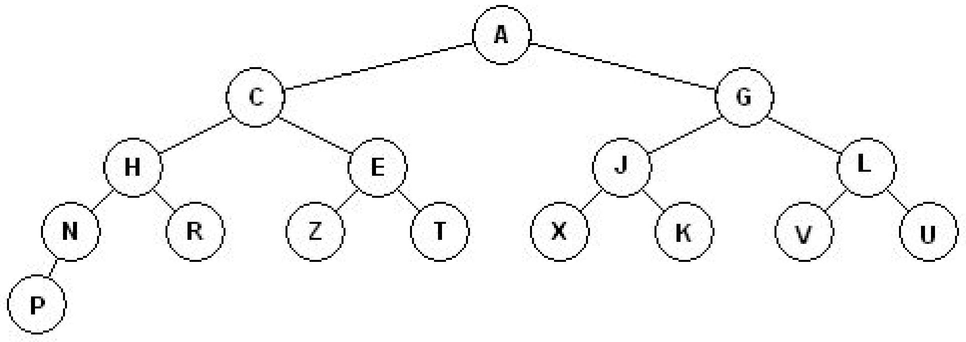


run

In a merge sort, an ordered subset of the data.

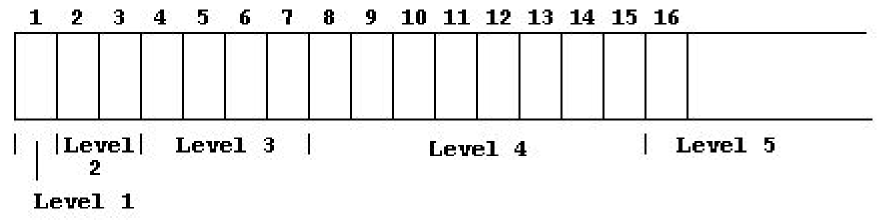
8.5 Heaps

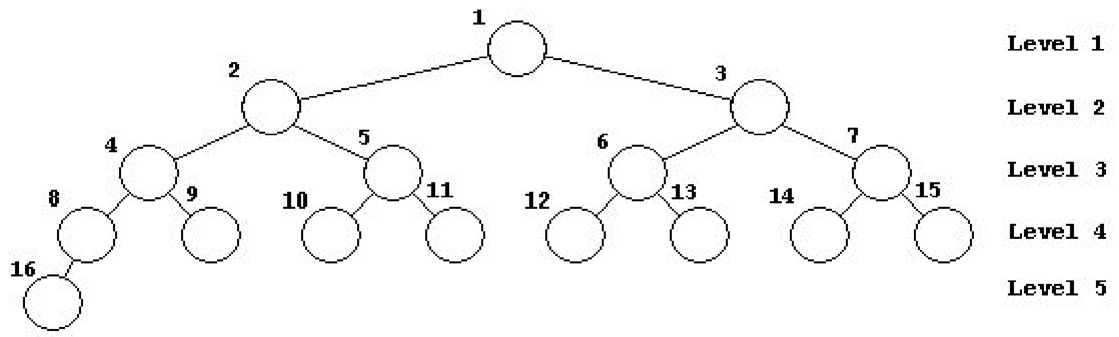
8.5.1 Logical Structure of a Heap



- A heap is a version of binary tree.
- A heap is ordered along each branch, from the root to any of the leaves.
- A heap is not ordered in the left-right method of a binary search tree.
- A heap is not ordered horizontally, within the tree levels.
- A heap is ordered vertically, from the leaves to the root.
- A heap is a complete binary tree.
- All levels of a heap are filled, except for the lowest.
- The lowest level is filled from left to right.
- The root of a heap is always the lowest (or highest) item in the heap.

8.5.2 Physical Structure of a Heap





- The nodes of a heap are stored physically in an array.
- The root of the tree is in the first element of the array.
- The children of the node at location *n* are at locations *2n* and *2n + 1*.
- The parent of the node at location *n* is at location $\lfloor n/2 \rfloor$.

8.5.3 Building a Heap

- A heap is built by inserting elements at the end of the heap, as a new leaf, and then "sifting up," sorting the branch from the new leaf to the root.
- Example:
- Insert *R*: **R**
- Insert *L*: **R L**
-

R

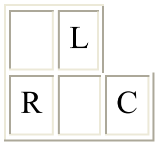
L

- Sift *L*: **L R**
-

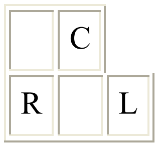
L

R

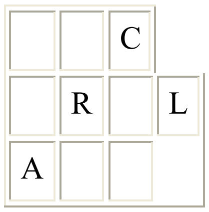
- Insert *C*: **L R C**
-



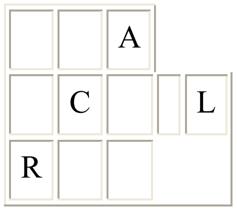
- Sift *C*: **C R L**
-

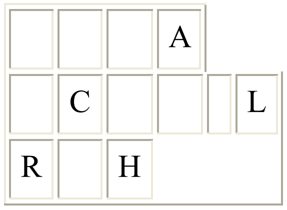


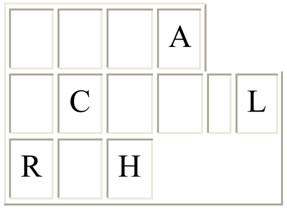
- Insert *A*: **C R L A**
-

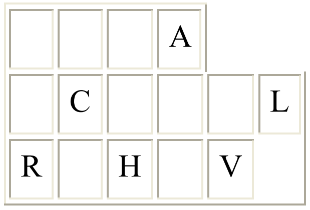


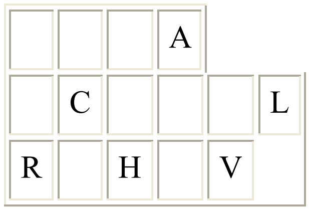
- Sift *A*: **A C L R**

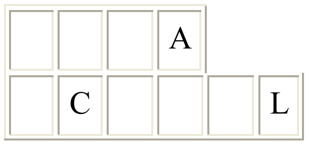
- 

- Insert *H*: **A C L R H**
- 

- Sift *H*: **A C L R H**
- 

- Insert *V*: **A C L R H V**
- 

- Sift *V*: **A C L R H V**
- 

- Insert *E*: **A C L R H V E**
- 

R		H		V		E

- Sift *E*: **A C E R H V L**
-

			A			
	C				E	
R		H		V		L

8.4 A Second Look at Sorting in Memory

heapsort

A sort based on insertions and removals from a heap

- Algorithm:
  - Build a heap from the data.
  - While there are items in the heap.
    - Remove the root from the heap.
    - Replace the root with the last leaf of the heap.
    - Sift the root down to restore the array to a heap.

8.4.1 Removing items from a heap

- Initial Heap: **A C E R H V L**

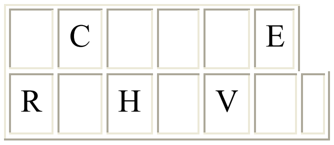
			A			
	C				E	
R		H		V		L

- Remove *A*: **~~A~~ C E R H V L**
-

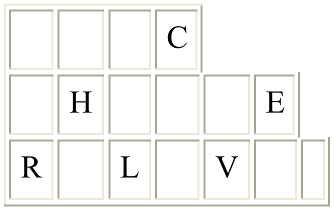
	C				E	
R		H		V		L

- Move Up *L*: **L C E R H V**
-

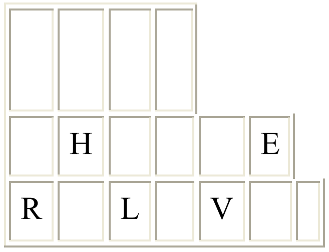
			L



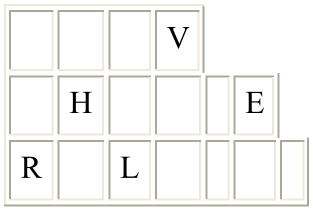
- Sift Down *L*: **C H E R L V**
-



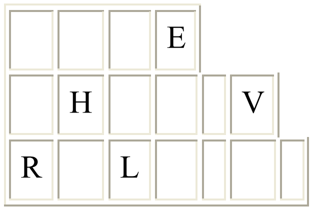
- Remove *C*: **~~C~~ H E R L V**
-



- Move Up *V*: **V H E R L**
-



- Sift Down *V*: **E H V R L**
-



8.7 Using Heapsort for the Distribution Phase of a Merge Sort

- Heapsort Algorithm:
    - Allocate Space for a heap of *n* records.
    - While more records in input:
        - While the heap is not full and there are more records:
            - Read a record from the input
            - Insert the record into the heap

        - Open a run
        - While the heap is not empty:
            - Remove the root record from the heap
            - Write the record to the run
        - Close the run

- This process will produce equal size runs of *n* records each.
- For an input file of *N* records, there will be $\lceil N/n \rceil$ runs.

8.8 Using Replacement Selection for the Distribution Phase of a Merge Sort

replacement selection

An algorithm for creating the initial runs of a mergesort which is based on a heapsort, which adds new records to the heap when possible, to lengthen the runs.

- Replacement Selection Algorithm:
    - Allocate Space for a heap of *n* records.
    - While the heap is not full and there are more records:
        - Read a record from the input
        - Insert the record into the heap
    - While the heap is not empty:
        - Open a run
        - While the heap is not empty:
            - Remove the root record from the heap
            - Write the record to the run
            - Read a record from the input
            - If the last record written is less than or equal to the new record read,
                - Insert the new record into the heap
            - If the last record written is greater than the new record read,
                - Insert the new record into the same array as the heap, but do not insert it into the heap
        - Close the run
        - Form a new heap from the records in the array.

8.8.1 Replacement Selection Algorithm Example

- Initial Array (All heap):

Array **A C E R H**

Heap

- Remove *A* from the heap and write it to the run:

Array

C H E R	
Heap	

Run

A

- Read *J* from the input and add it to the heap (J > A):

Array

C H E R J
Heap

Run

A

- Remove *C* from the heap and write it to the run:

Array

E H J R	
Heap	

Run

A C

- Read *B* from the input and add it to the array, just past the heap (B < C):

Array

E H J R	**B**
Heap	

Run

A C

- Remove *E* from the heap and write it to the run:

Array

H R J	**B**
Heap	

Run

A C E

- Read *T* from the input and add it to the heap (T > E):

Array

H R J T	**B**
Heap	

Run

A C E

- Remove *H* from the heap and write it to the run:

Array

J R T	**B**
Heap	

Run

A C E H

- Read *L* from the input and add it to the heap (L > H):

Array

J L T R	**B**
Heap	

Run

A C E H

- Remove *J* from the heap and write it to the run:

Array

L R T	**B**
Heap	

Run

A C E H J

- Read *D* from the input and add it to the array, just past the heap (D < J):

Array

L R T	**D B**
Heap	

Run

A C E H J

- Remove *L* from the heap and write it to the run:

Array

R T	**D B**
Heap	

Run

A C E H J L

- Read *V* from the input and add it to the heap (V > L):

Array

R T V	**D B**
Heap	

Run

A C E H J L

- Remove *R* from the heap and write it to the run:

Array

T V	**D B**
Heap	

Run

A C E H J L R

- Read *K* from the input and add it to the array, just past the heap (K < R):

Array

T V	**K D B**
Heap	

Run

A C E H J L R

- Remove *T* from the heap and write it to the run:

Array

V	**K D B**
Heap	

Run

A C E H J L R T

- Read *N* from the input and add it to the array, just past the heap (N < T):

Array

V	**N K D B**
Heap	

Run

A C E H J L R T

- Remove *V* from the heap and write it to the run:

Array

N K D B

Run

A C E H J L R T V

- Read *Q* from the input and add it to the array, just past the (null) heap (Q < V):

Array **Q N K D B** Run **A C E H J L R T V**

- Convert the array to a heap and start a new run:

Array **B D N Q K** Run **A C E H J L R T V** Run

Heap

- Repeat

8.8.2 Advantages of Replacement Sort over Heapsort

- Replacement sort runs will be, on average, twice as long as heapsort runs, using the same size array.
- There will be half as many replacement sort runs, on average, as heapsort runs, using the same size array.

8.5 Merging as a Way of Sorting Large Files on Disk

8.5.1 Single Step merging

k-way merge

A merge of order k.

order of a merge

The number of input lists being merged.

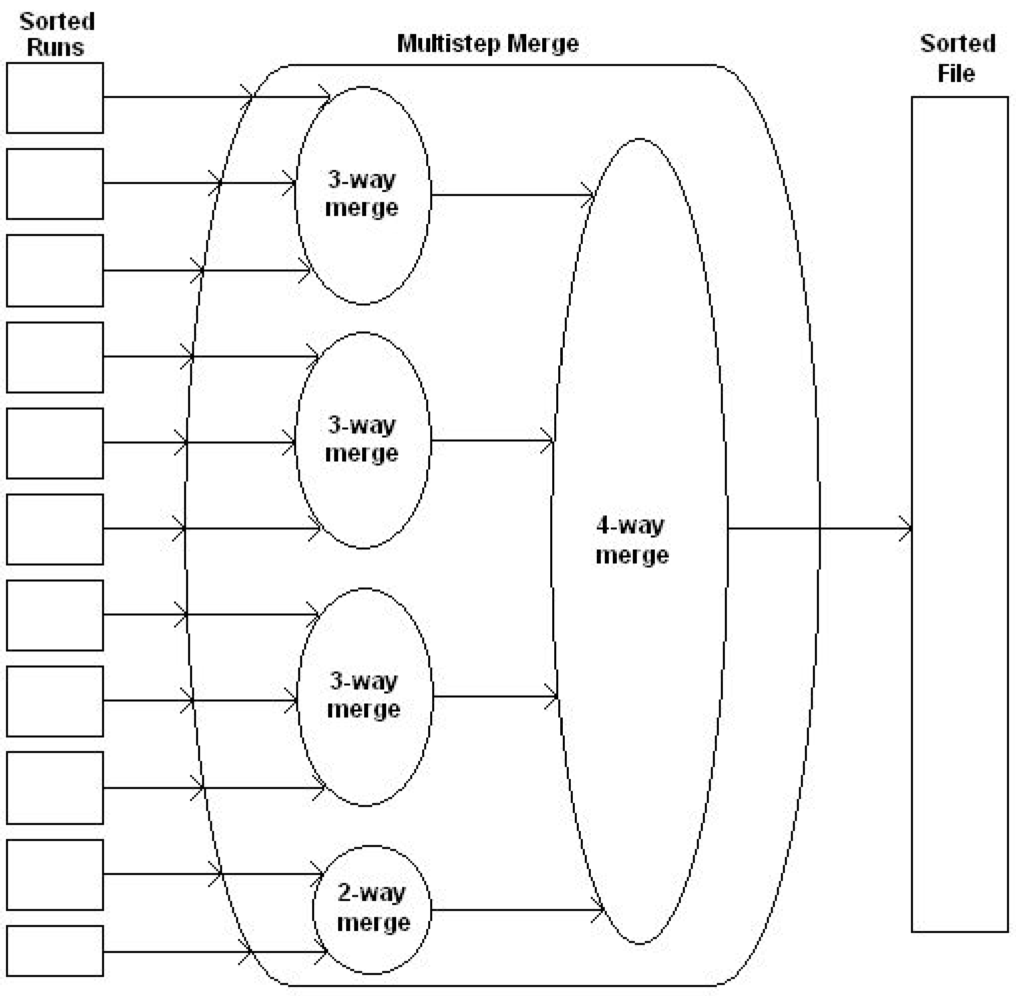
- If the distribution phase creates *k* runs, a single k-way merge can be used to produce the final sorted file.
- A significant amount of seeking is used by a k-way merge, assuming the input runs are on the same disk.

8.9.2 Multistep Merging

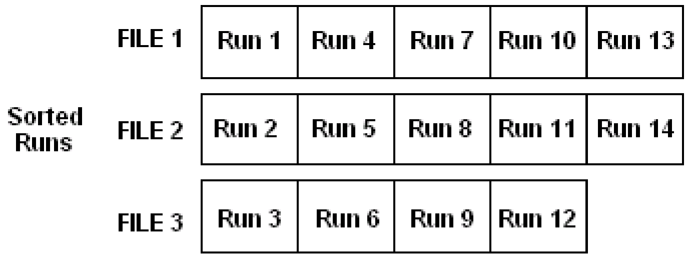
multistep merge

A merge which is carried out in two or more stages, with the output of one stage being the input to the next stage.

- A multistep merge increases the number of times each record will be read and written.
- Using a multistep merge can decrease the number of seeks, and reduce the overall merge time.



8.6 Sorting Files on Tape


Sorted Runs

FILE 1	Run 1	Run 4	Run 7	Run 10	Run 13
FILE 2	Run 2	Run 5	Run 8	Run 11	Run 14
FILE 3	Run 3	Run 6	Run 9	Run 12	

- When sorting with tape, multiple runs are placed in a single file.

balanced merge

A multistep merge which uses the same number of output files as input files.

8.7 Sort-Merge Packages

8.8 Sorting and Cosequential Processing in UNIX

Transaction Posting

Data - ledger.txt - General Ledger

```
101  Checking Account #1        1032.57  2114.56  5219.23
102  Checking Account #2         541.78  3096.17  1321.20
505  Advertising Expense         25.00    25.00    25.00
510  Auto Expense               195.40   307.92   501.12
515  Bank Charges                 0.00     0.00     0.00
520  Books & Publications        27.95    27.95    87.40
525  Interest Expense           103.50   255.20   380.27
535  Miscellaneous Expense       12.45    17.87    23.97
540  Office Expense              37.50   105.25   138.37
545  Postage & Shipping Expense  21.00    27.63    57.45
550  Rent Expense               500.00  1000.00  1500.00
555  Supplies Expense           112.00   167.50  2441.80
```

Data - journal.txt - Daily Journal

```
101  1271  04/02/96  Auto Expense             -78.70
510  1271  04/02/96  Tune Up, Minor Repair     78.70
101  1272  04/02/96  Rent                    -500.00
550  1272  04/02/96  April Rent               500.00
101  1273  04/04/96  Advertising              -87.50
505  1273  04/04/96  Newspaper Ad              87.50
```

Data - sorted.txt - Sorted Journal

```
101  1271  04/02/96  Auto Expense             -78.70
101  1272  04/02/96  Rent                    -500.00
101  1273  04/04/96  Advertising              -87.50
505  1273  04/04/96  Newspaper Ad              87.50
510  1271  04/02/96  Tune Up, Minor Repair     78.70
550  1272  04/02/96  April Rent               500.00
```

Report - summary.txt - Monthly Summary

```
101  Checking Account #1
   1271  04/02/96  Auto Expense             -78.70
   1272  04/02/96  Rent                   -500.00
   1273  04/04/96  Advertising             -87.50
   1274  04/04/96  Auto Expense             -31.83
Beginning Balance:  5219.23   Ending Balance: 4521.20

102  Checking Account #2
    670  04/02/96  Office Supplies         -32.78
Beginning Balance:  1321.20   Ending Balance: 1288.42
```

Source Code - posting.cpp

```
int main () {
  Transiaction  Entry;
  Transiactions Journal;
  Account       CurrentAccount;
  Accounts      Ledger;
  Summary       Report;
  bool          More

  Ledger.Open ("ledger.txt");
  Journal.Open ("sorted.txt");
  Report.Open ("summary.txt");

  Ledger >> CurrentAccount;
  Journal >> Entry;
  CurrentAccount.Extend (Entry);
  Report.Head (CurrentAccount);
  More = Journal.good () || Ledger.good ();

  while (More) {
    if (Entry.Account == CurrentAccount.Number) {
      CurrentAccount.Post (Entry);
      Report << Entry;
      Journal >> Entry;
      More = Journal.good ();
    } else if (Entry.Account > CurrentAccount.Number) {
      Report.Foot (CurrentAccount);
      Ledger.Update (CurrentAccount);
      Ledger >> CurrentAccount.
      if (Ledger.good()) {
        CurrentAccount.Extend (Entry);
        Report.Head (CurrentAccount);
      } else {
        More = false;
      }
    } else { /*Entry.Account < CurrentAccount.Number */
      Report.Error (Entry);
      Journal >> Entry;
      More = Journal.good ();
    }
  }

  Report.Foot (CurrentAccount);

  Journal.Close ();
  Ledger.Close ();
  Report.Close ();

  return 0;
}
```

Source Code - transactions.h

```
#include "Account.h";

class Accounts: public RecordFile {
  public:
    Accounts & operator >> (Account &);
  bool      Update     (Account &);
}
```

Source Code - transaction.h

```
class Transaction {
 public:
  unsigned int   Account     (unsigned int = 0);
  double         Amount      ();
  double         Amount      (double);
  char *         Date        (char * = NULL);
  char *         Description (char * = NULL);
  unsigned int   Month       (unsigned int = 0);
}
```

Source Code - accounts.h

```
class Transactions: public RecordFile {
 public:
  Transactions & operator >> (Transaction &);
}
```

Source Code - account.h

```
class Account {
 public:
  unsigned int   Account     (unsigned int = 0);
  char *         Description (char * = NULL);
  double         Balance     (unsigned int);
  double         Balance     (unsigned int, double);
  bool           Extend      (Transaction &);
}

}
```

Source Code - account.cpp

```
#include "Account.h"

unsigned int Account::Account (unsigned int Value = 0) {
 if (Value > 0) {
  sprintf (Field [0], "%d", Value);
 }
 return atou (Field [0]);
```

```
}

char * Account::Description (char * Value) {
 if (Value != NULL) {
   sprintf (Field [1], "%s", Value);
  }
 return Field [1];
}

double Account::Balance (unsigned int Month) {
 return atod (Field [3 + Month];
}

double Account::Balance (unsigned int Month, double Value) {
 sprintf (Field [3 + Month], "%f", Value);
 return atod (Field [3 + Month];
}

bool Account::Extend (Transaction & T) {
 unsigned int Month;

 Month = T.Month ();
 Balance (Month, Balance (Month - 1));
}

bool Account::Post (Transaction & T) {
 unsigned int Month;
 Month = T.Month ();
 Balance (Month, Balance (Month) + T.Amount());
}
```

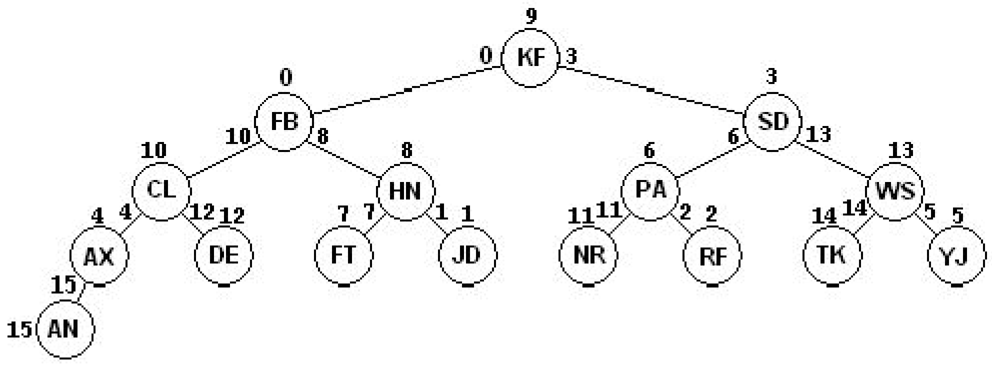
UNIT-5
Multilevel Indexing and B-Trees

9.1 Introduction: The Invention of B-Trees

9.2 Statement of the Problem

- Searching an index must be faster than binary searching.
- Inserting and deleting must be as fast as searching.

9.3 Indexing with Binary Search Trees



binary tree

A tree in which each node has at most two children.

leaf

A node at the lowest level of a tree.

height-balanced tree

A tree in which the difference between the heights of subtrees in limited.

- Binary trees grow from the top down: new nodes are added as new leaves.
- Binary trees become unbalanced as new nodes are added.
- The imbalance of a binary tree depends on the order in which nodes are added and deleted.
- Worst case search with a balanced binary tree is $\log_2 (N + 1)$ compares.
- Worst case search with an unbalanced binary tree is $>N$ compares.

9.3.1 AVL Trees

AVL Tree

A binary tree which maintains height balance (to HB(1)) by means of localized reorganizations of the nodes.

- AVL trees maintain *HB(1)* with local rotations of the nodes.
- AVL trees are not perfectly balanced.

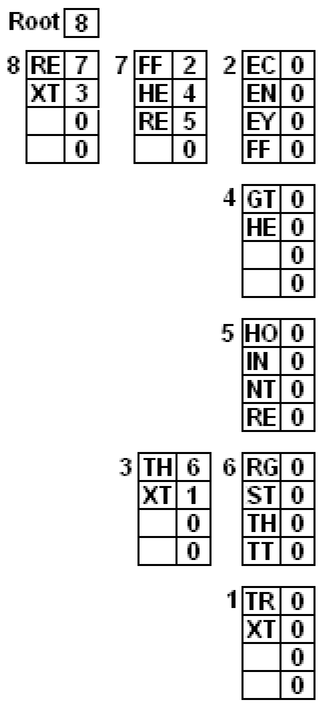
- Worst case search with an AVL tree is 1.44 $\log_2 (N + 2)$ compares.

9.3.2 Paged Binary Trees

- Worst case search with a balanced paged binary tree with page size M is $\log_{M+1} (N + 1)$ compares.
- Balancing a paged binary tree can involve rotations across pages, involving physical movement of nodes.

9.3.3 Problems with Paged Binary Trees

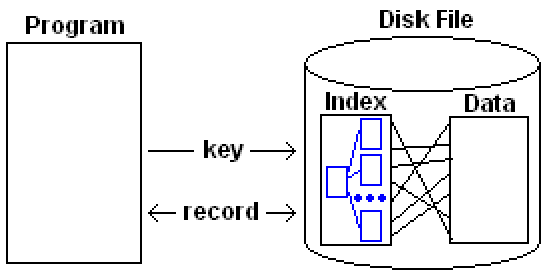
9.4 Multilevel Indexing: A Better Approach to Tree Indexes



9.5 B-Trees: Working up from the Bottom

B-tree

A multiway tree in which all insertions are made at the leaf level. New nodes at the same level are created when required by node overflow, and new nodes at the parent level are created when required by the creation of new nodes. leaf level when



- B-trees grow from the bottom up.
- B-trees are always automatically balanced.

paged index

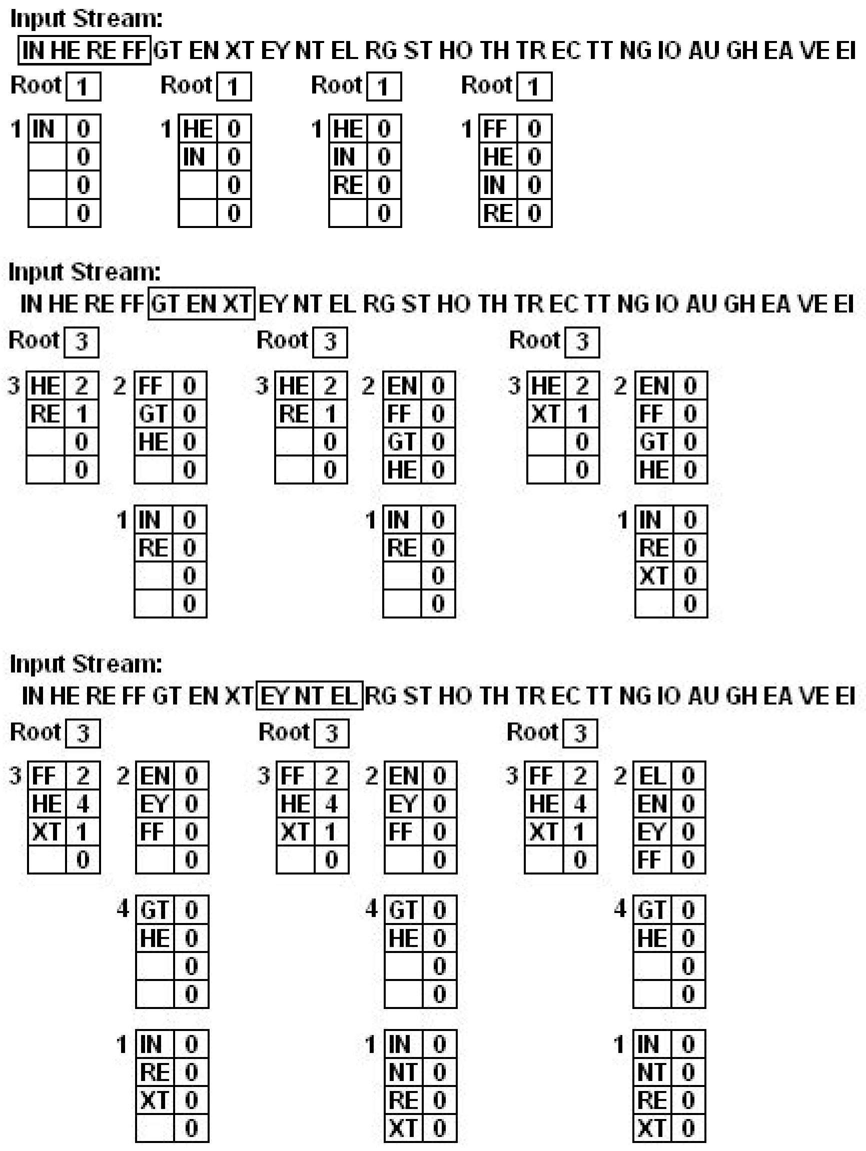
An index organized as pages, or blocks, each of which holds multiple keys.

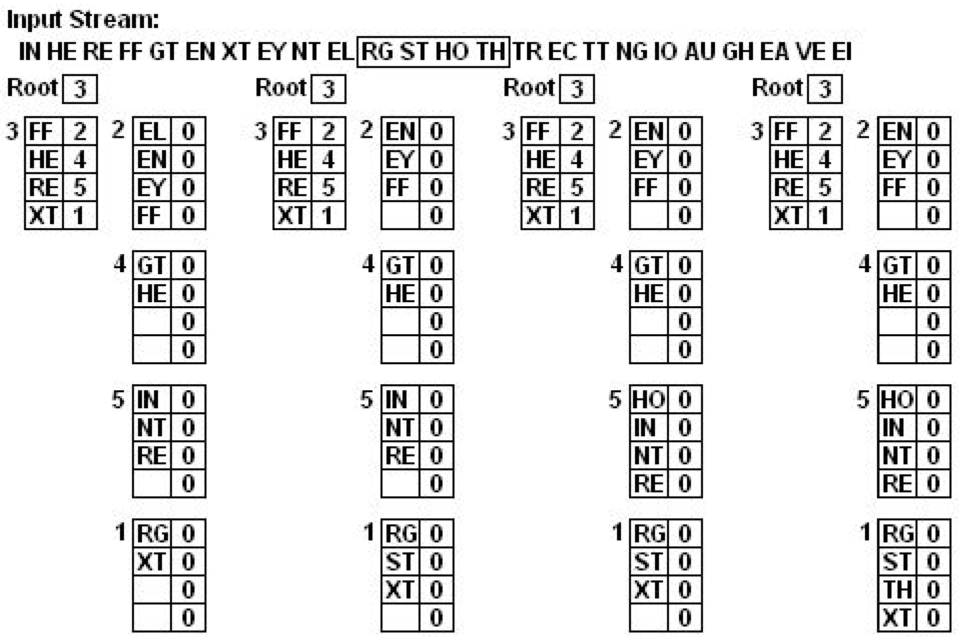
splitting

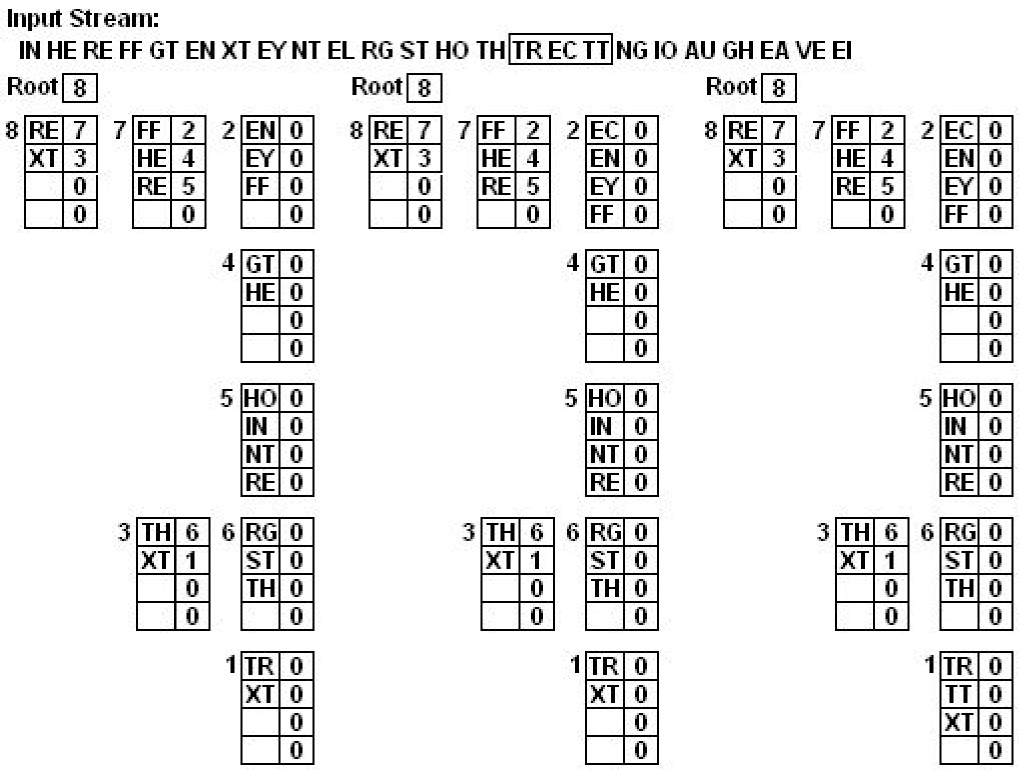
Creation of a new node when a node overflows, with the partial distribution of the contents of the overflowing node to the new node.

- New B-tree pages are created when a leaf overflows and is split.

9.6 Example of Creating a B-Tree


Input Stream:
IN HE RE FF GT EN XT EY NT EL RG ST HO TH TR EC TT NG IO AU GH EA VE EI
Root 1
Input Stream:
IN HE RE FF GT EN XT EY NT EL RG ST HO TH TR EC TT NG IO AU GH EA VE EI
Root 3
Input Stream:
IN HE RE FF GT EN XT EY NT EL RG ST HO TH TR EC TT NG IO AU GH EA VE EI
Root 3


Input Stream:
IN HE RE FF GT EN XT EY NT EL RG ST HO TH TR EC TT NG IO AU GH EA VE EI
Root 3
Root 3
Root 3
Root 3


Input Stream:
IN HE RE FF GT EN XT EY NT EL RG ST HO TH TR EC TT NG IO AU GH EA VE EI
Root 8
Root 8
Root 8

9.7 B-Tree Methods: Search, Insert, and Others

9.7.1 Searching

9.7.2 Insertion

9.7.3 Create, Open, and Close

9.8 B-Tree Nomenclature

order of a B-tree

The maximum number of children which a B-tree supports.

9.9 Formal Definition of B-Tree Properties

9.9.1 For a B-Tree of Order *n*

- Every page has a maximum of *n* children.
- Every page, except for the root and the leaves, has a minimum of $\lceil n/2 \rceil$ children.
- The root has a minimum of 2 children (unless it is a leaf.)
- All of the leaves are on the same level.
- The leaf level forms a complete, ordered index of the associated data file.

9.10 Worst Case Search Depth

- Worst case search with a B-tree of order M is $\lfloor 1 + \log_{M/2} (N) \rfloor$ compares.

9.11 Deletion, Merging, and Redistribution

9.11.1 Deletion

merging

The combination of the contents of two nodes into a single node, with the deletion of one.

9.11.2 Redistribution

9.12 Redistribution during Insertion: A Better Way to Improve Storage Utilization

redistribution

The movement of contents between adjacent nodes to equalize the loading.

9.13 B^* Trees

B* tree

A B-tree in which each node is at least 2/3 full.

9.13.1 For a B^* Tree of Order *n*

- Every page has a maximum of *n* children.
- Every page, except for the root and the leaves, has a minimum of $\lceil (n-1)/2 \rceil$ children.
- The root has a minimum of 2 children (unless it is a leaf.)
- All of the leaves are on the same level.
- The leaf level forms a complete, ordered index of the associated data file.

9.14 Buffering of B-Trees: Virtual B-Trees

virtual B-tree

A B-tree which is accessed by routines which cache pages in anticipation of later requests.

9.14.1 LRU Replacement

- LRU = Least Recently Used
- Using caching of pages, in a virtual B-tree, can significantly reduce the average number of disk accesses required for index searching.

9.14.2 Replacement Based on Page Height

9.14.3 Importance of Virtual B-Trees

9.15 Variable Length Records and Trees

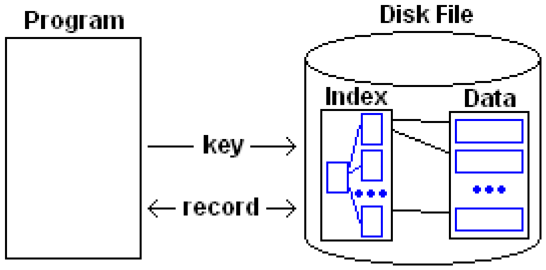
UNIT-6

Indexed Sequential File Access and Prefix B+Trees

10.1 Indexed Sequential Access

indexed sequential access

Access which can be either indexed or sequential.



10.2 Maintaining a Sequence Set

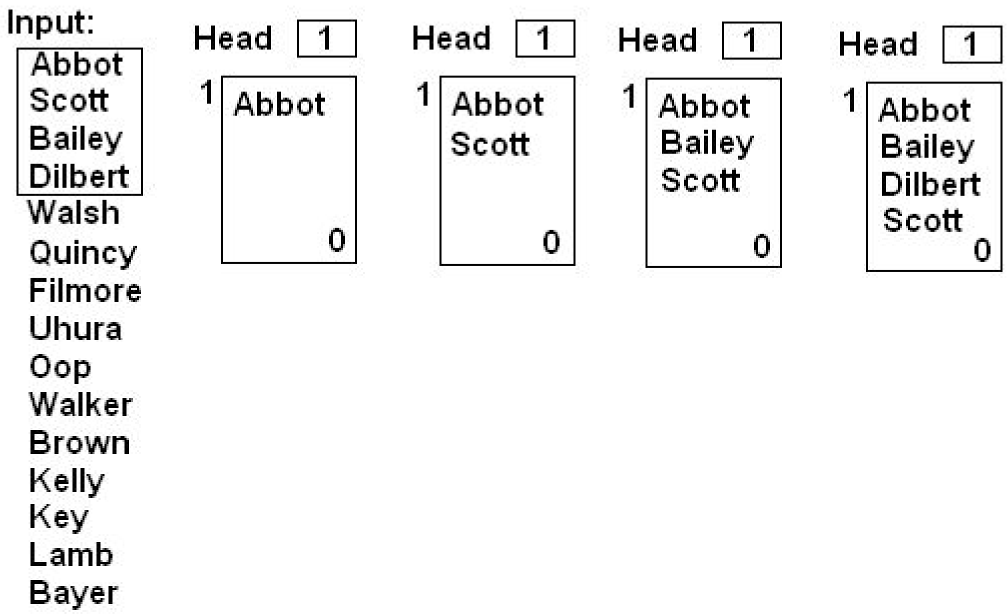
sequence set

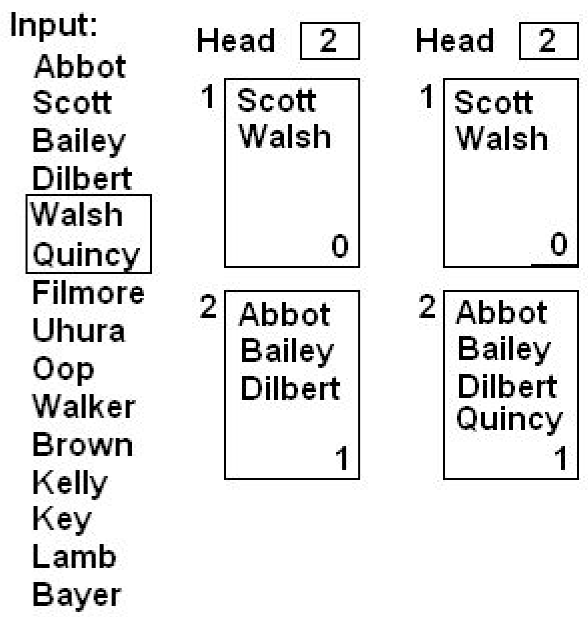
The part of a B+ tree which contains the data records, in key sequential order.

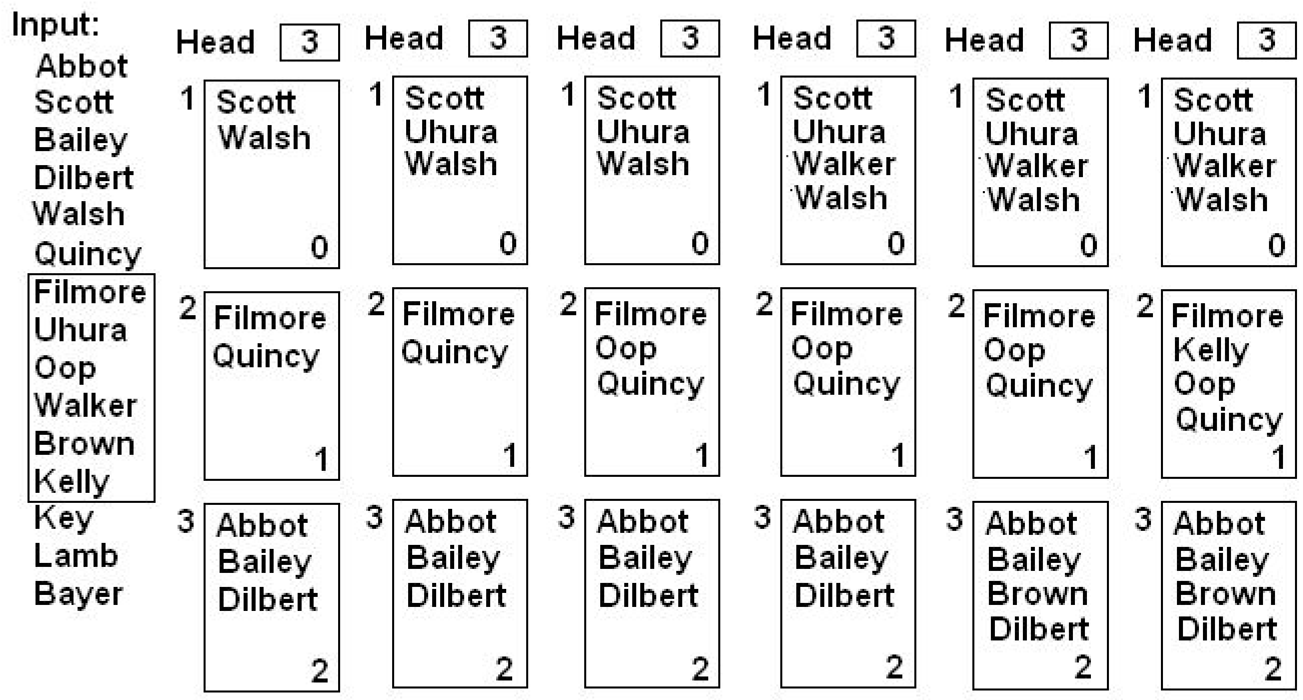
10.2.1 The Use of Blocks

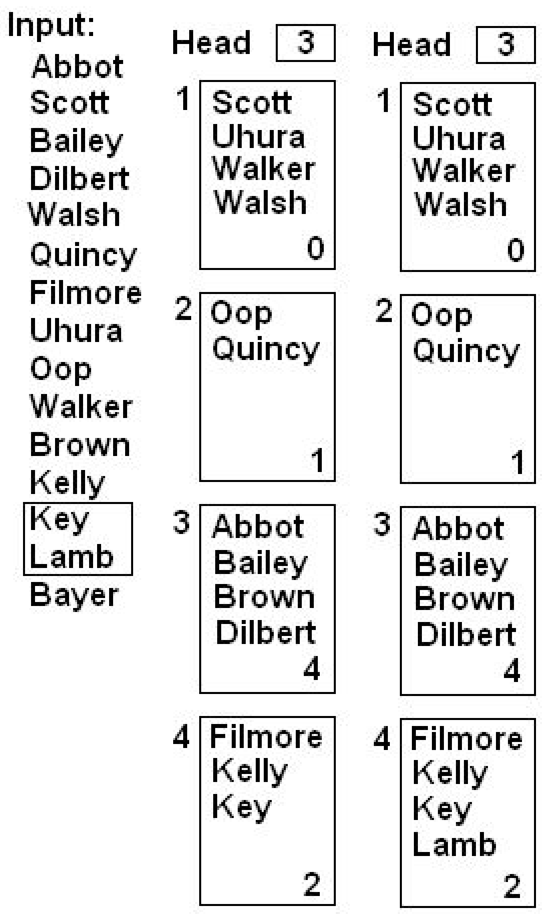
- The sequence set is an ordered list, divided into pages.
- The sequence set is physically ordered within the pages, and logically ordered between pages.
- The pages of the sequence set are split and merged, similarly to the pages of a B-tree.

10.2.2 Sequence Set Example




Input:
Abbot
Scott
Bailey
Dilbert
Walsh
Quincy
Filmore
Uhura
Oop
Walker
Brown
Kelly
Key
Lamb
Bayer
Head 2
1 Scott Walsh 0
2 Abbot Bailey Dilbert 1
Head 2
1 Scott Walsh 0
2 Abbot Bailey Dilbert Quincy 1


Input:
Abbot
Scott
Bailey
Dilbert
Walsh
Quincy
Filmore
Uhura
Oop
Walker
Brown
Kelly
Key
Lamb
Bayer
Head 3
1 Scott Walsh 0
2 Filmore Quincy 1
3 Abbot Bailey Dilbert 2
Head 3
1 Scott Uhura Walsh 0
2 Filmore Quincy 1
3 Abbot Bailey Dilbert 2
Head 3
1 Scott Uhura Walsh 0
2 Filmore Oop Quincy 1
3 Abbot Bailey Dilbert 2
Head 3
1 Scott Uhura Walker Walsh 0
2 Filmore Oop Quincy 1
3 Abbot Bailey Dilbert 2
Head 3
1 Scott Uhura Walker Walsh 0
2 Filmore Oop Quincy 1
3 Abbot Bailey Brown Dilbert 2
Head 3
1 Scott Uhura Walker Walsh 0
2 Filmore Kelly Oop Quincy 1
3 Abbot Bailey Brown Dilbert 2



10.2.3 Choice of Block Size

- The block size should be large enough minimize the number of blocks in the file.
- The block size should be small enough to allow several blocks to be held in memory at the same time.
- The block size should be small enough to allow "quick" access
- The block size should be small enouth to be read without an internal seek.
- The cluster (ALU) size is often a good choice.

10.3 Adding a Simple Index to the Sequence Set

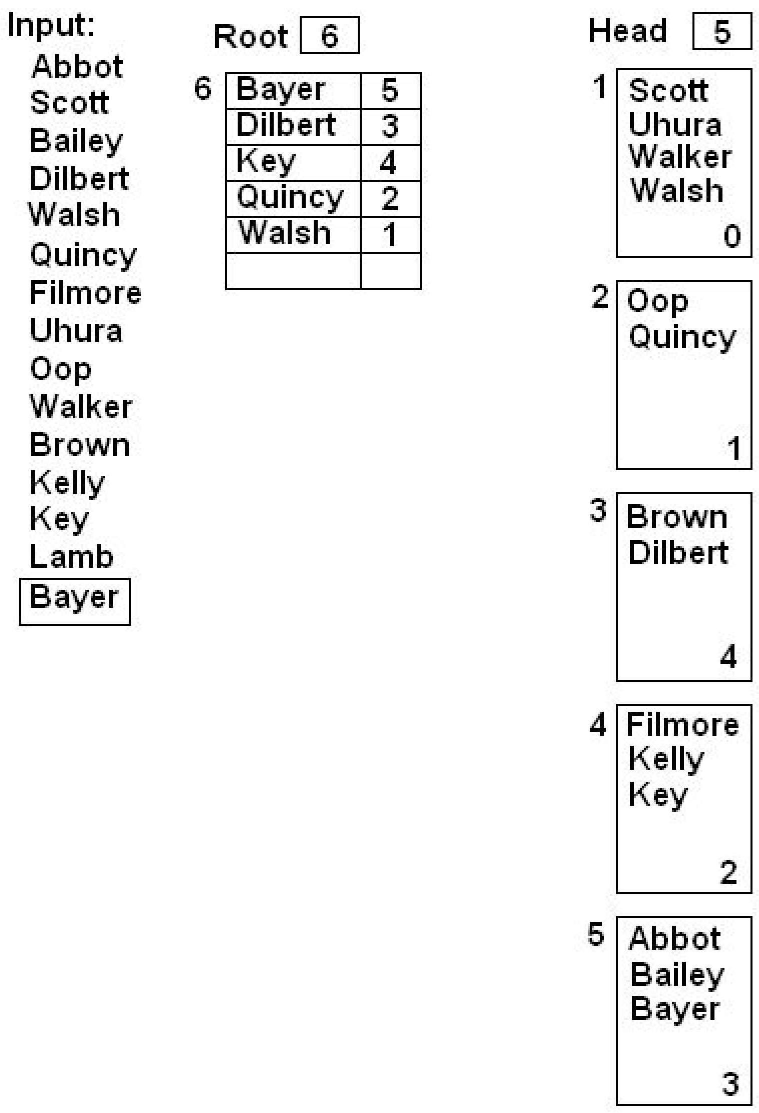
index set

The index of a B+ tree, organized as a B-tree.

B+ tree

A file structure consisting of a sequence set and an index set.

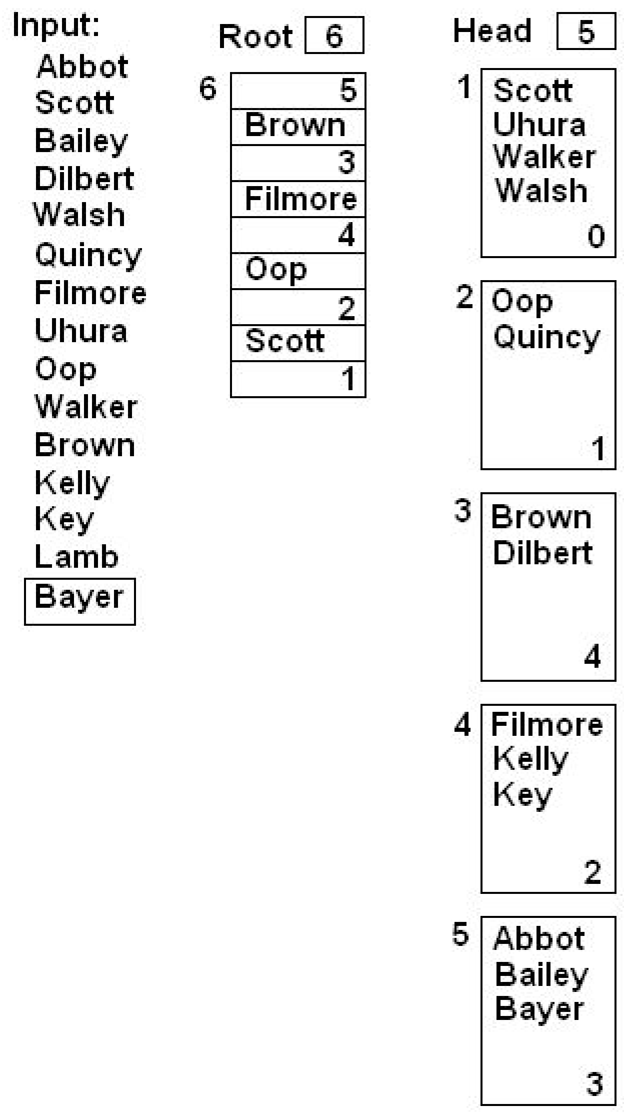
- The index set contains one entry per sequence set page.



10.4 The Content of the Index: Separators Instead of Keys

separator

A value which can be compared to a key to determine the proper block of an index.

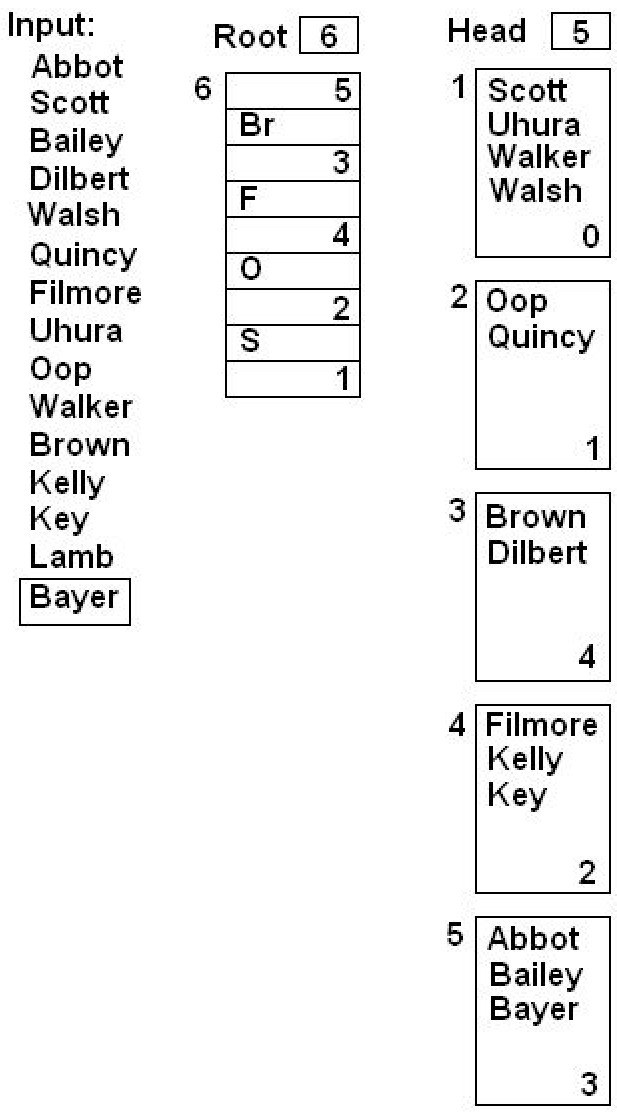


variable order

Order which does not always have the same value.

shortest separator

The shortest value which can be compared to a key to determine the proper block of an index.

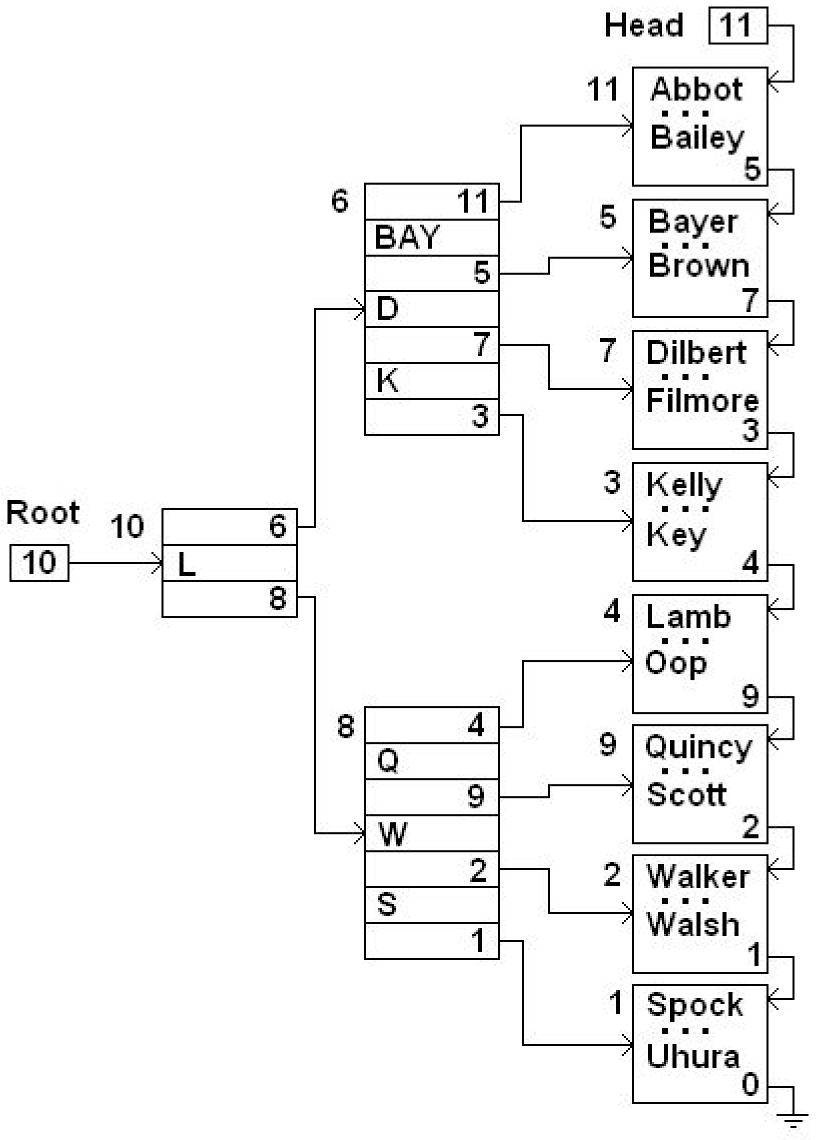


- The index entries determine which page of the sequence set *may* contain the search key.

10.5 The Simple Prefix B+Tree

simple prefix B+ tree

A B+ tree in which the index set contains simple prefix separators.



10.6 Simple Prefix B+Tree Maintenance

10.6.1 Changes Localized to Single Blocks in the Sequence Set

- Additions, deleteions, and updates in the sequence set which affect only a single block do not affect the index set.

10.6.2 Changes involving multiple blocks in the sequence set

- When additions to the sequence set result in a split in the sequence set, a new separator must be inserted into toe index set.
- When deletions from the sequence set result in a merger in the sequence set, a separator must be deleted from the sequence set.
- When changes in the sequence set result in redistribution, there must be a corresponding change in the affected separators in the index set.

10.7 Index Set Block Size

- Index set blocks should be large enough to make the b-tree of the index set of high order.
- Index set blocks shoud be small enough to hold several in memory simultaneously.
- Index set blocks should always be an integral number of sectors.
- Index set block should probably be an integral number of clusters, to minimize seeking.
- It is usually convenient to make the index set pages and the sequence set blocks the same size; the two can then be easily comingled in a single file.

10.8 Internal Structure of Index Set Blocks: A Variable Order B-Tree

10.9 Loading a Simple Prefix B+Tree

- When loading a B+ tree from an existing file, it is often convenient to load the load the sequence set sequentially, filling each block to a specified fill factor, and making appropriate entries into the index set as the file is loaded.
- The previously described structure is a simple prefixe B+tree.
- It is also, of course, possible to build a B+tree index set from complete keys rather than from separators. This is a B+Tree.
- A B+tree index set would contain one entry per gap in the sequence set, as for a prefix B+tree. The difference would be that these entries would be complete keys, rather than shortest separators.

10.10 B-Trees, B*Trees, and B+Trees in Perspective

- B trees, B^*trees, and simple prefix B+trees have been described.
- Many other versions of each have been implemented, and are in use. The basic principles remain the same, though.
- Obviously, only one B+tree can be implemented on a single data set. This would normally be the primary index.
- Many B trees or B^* trees can be implemented on a single data set. These would normally be secondary indexes.

UNIT-7
Hashing

11.1 Introduction

- Key driven file access should be *O*(1) - that is, the time to access a record should be a constant which does not vary with the size of the dataset.
- Indexing can be regarded as a table driven function which translates a key to a numeric location.
- Hashing can be regarded as a computation driven function which translates a key to a numeric location.

hashing

The transformation of a search key into a number by means of mathematical calculations.

randomize

To transform in an apparently random way.

- Hashing uses a repeatable pseudorandom function.
- The hashing function should produce a uniform distribution of hash values.

uniform distribution

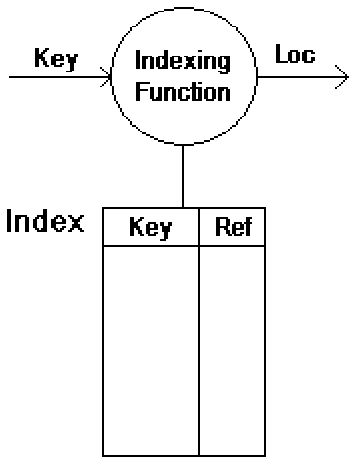
A randomization in which each value in a range has an equal probability.

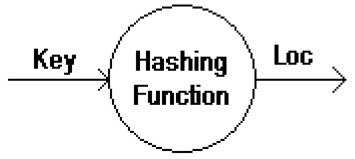
- For each key, the result of the hashing function is used as the home address of the record.

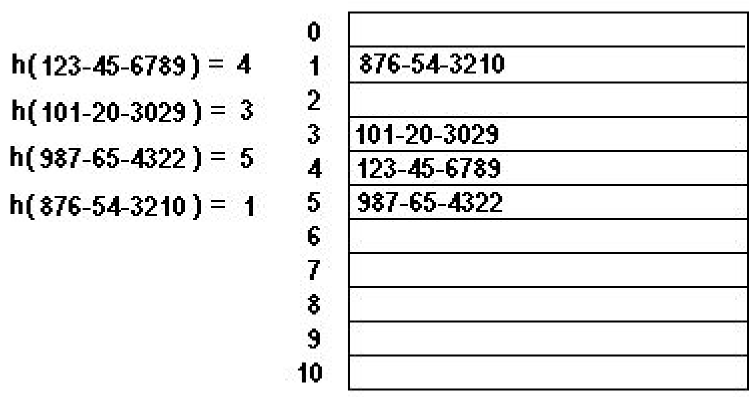
home address

The address produced by the hashing of a record key.

- Under ideal conditions, hashing provides *O*(1) key driven file access.







11.2 Hashing Algorithms

Modulus

- Modulus - the key is divided by the size of the table, and the remainder is used as the hash function.
- Example:
  Key = 123-45-6789
  123456789 % 11 = 5
  h(123-45-6789) = 5
- Modulus functions work better when the divisor is a prime number, or at least not a composite of small numbers.

Fold and Add

- **fold and add**
  A hashing techique which separates a long key field into smaller parts which are added together.
- Example:
  Key = 123-45-6789
  123
  456
  $\underline{+789}$
  1368
  h(123-45-6789) = 1368
- Fold and add is used for long keys.

Mid-square

- **mid-square**
  hashing method which squares the key and the uses the middle digits of the result.
- Example:
  Key = 123-45-6789
  123456789^2 = **$15241578\underline{750}190521$**
  h(123-45-6789) = 8750

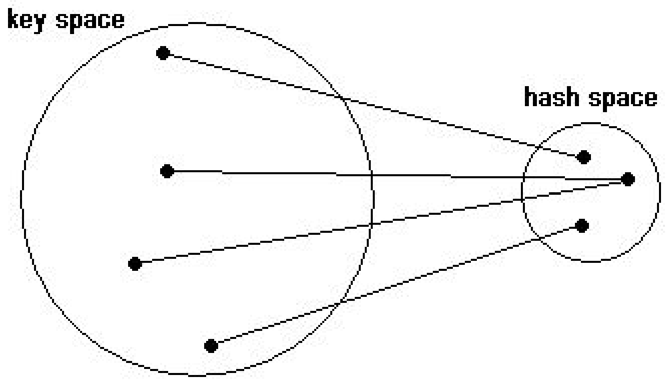
Combined methods

- Practical hashing functions often combine techniques.
- Example:
  Key = 123-45-6789
  123
  456
  +789
  1368

  1368 % 11 = 4
  h(123-45-6789) = 4
- For non-numeric keys, the key is simply treated as though it were a number, using its internal binary representation.
- Example:
  Key = "Kemp"
  "Kemp" = $4B656D70_{16} = 1264938352_{10}$

11.3 Hashing Functions and Record Distributions

- The size of the key space is typically much larger than the space of hashed values.
- This means that more than one key will map to the same hash value.



Collisions

- **synonyms**
  Keys which hash to the same value.
- **collision**
  An attempt to store a record at an address which does not have sufficient room
- **packing density**
  The ratio of used space to allocated space.
- For simple hashing, the probability of a synonym is the same as the packing density.

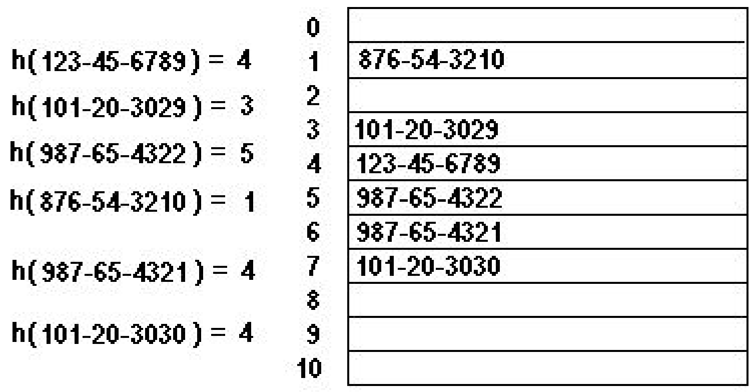
11.4 How Much Extra Memory Should be Used?

- Increasing memory (i.e., increasing the size of the hash table) will decrease collisions.

11.5 Collision Resolution by Progressive Overflow

- **progressive overflow**

  A collision resolution technique which places overflow records at the first empty address after the home address
- With progressive overflow, a sequential search is performed beginning at the home address.
- The search is continued until the desired key or a blank record is found.
- Progressive overflow is also referred to as *linear probing.*



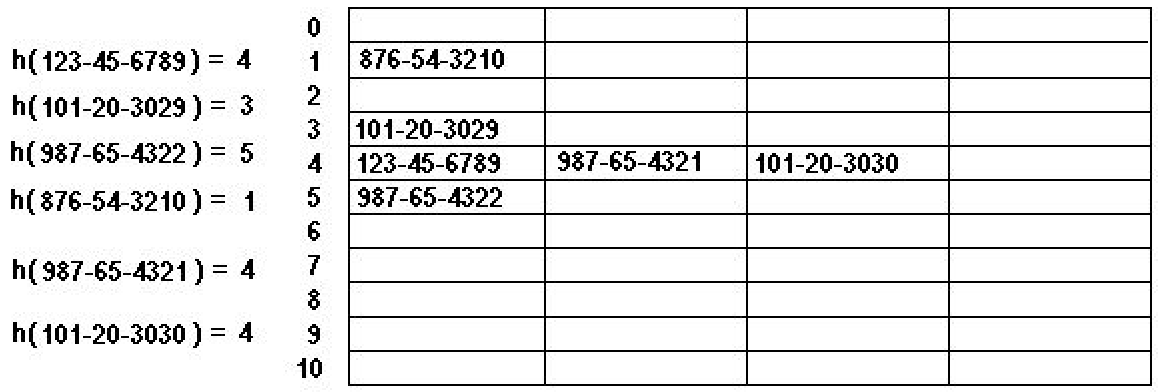
h(123-45-6789) = 4
h(101-20-3029) = 3
h(987-65-4322) = 5
h(876-54-3210) = 1
h(987-65-4321) = 4
h(101-20-3030) = 4

0	
1	876-54-3210
2	
3	101-20-3029
4	123-45-6789
5	987-65-4322
6	987-65-4321
7	101-20-3030
8	
9	
10	

11.6 Storing more than One Record per Address: Buckets

- **bucket**

  An area of a hash table with a single hash address which has room for more than one record.
- When using buckets, an entire bucket is read or written as a unit. (Records are not read individually.)
- The use of buckets will reduce the average number of probes required to find a record.



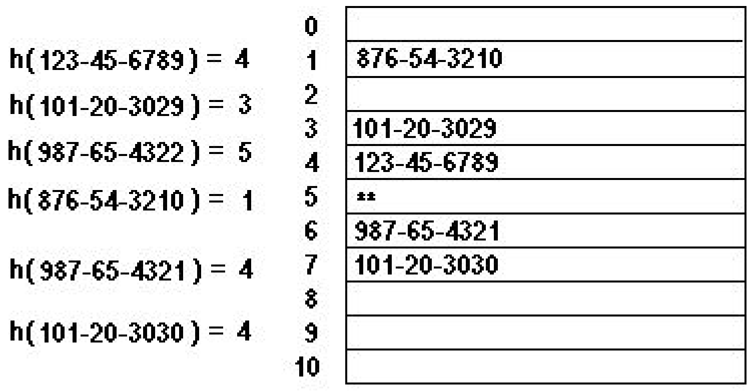
h(123-45-6789) = 4
h(101-20-3029) = 3
h(987-65-4322) = 5
h(876-54-3210) = 1
h(987-65-4321) = 4
h(101-20-3030) = 4

0				
1	876-54-3210			
2				
3	101-20-3029			
4	123-45-6789	987-65-4321	101-20-3030	
5	987-65-4322			
6				
7				
8				
9				
10				

11.7 Making Deletions

- **tombstone**

  A marker placed on a deleted record.
- Using a tombstone avoids terminating a probe prematurely.
- Locations containing a tombstone can be be used for records being added.

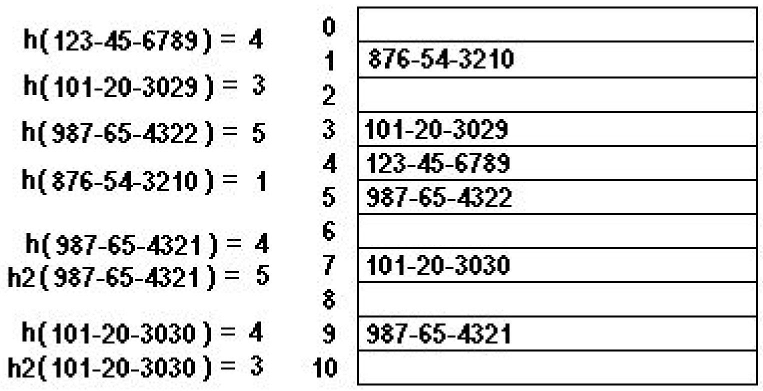


11.8 Other Collision Resolution Techniques

Double Hashing

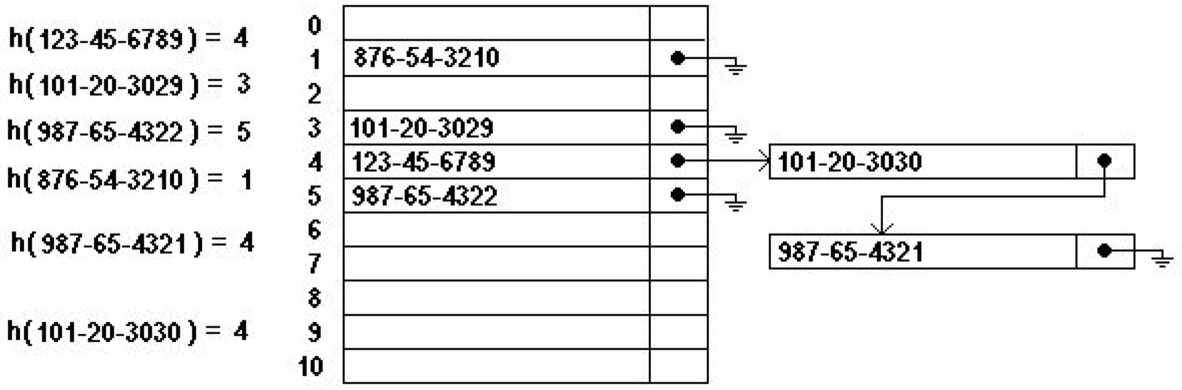
- Double hashing is similar to progressive overflow.
- The second hash value is used as a stepping distance for the probing.
- The second hash value should never be one. (Add one.)
- The second hash value should be relatively prime to the size of the table. (This happens if the table size is prime.)
- **double hashing**

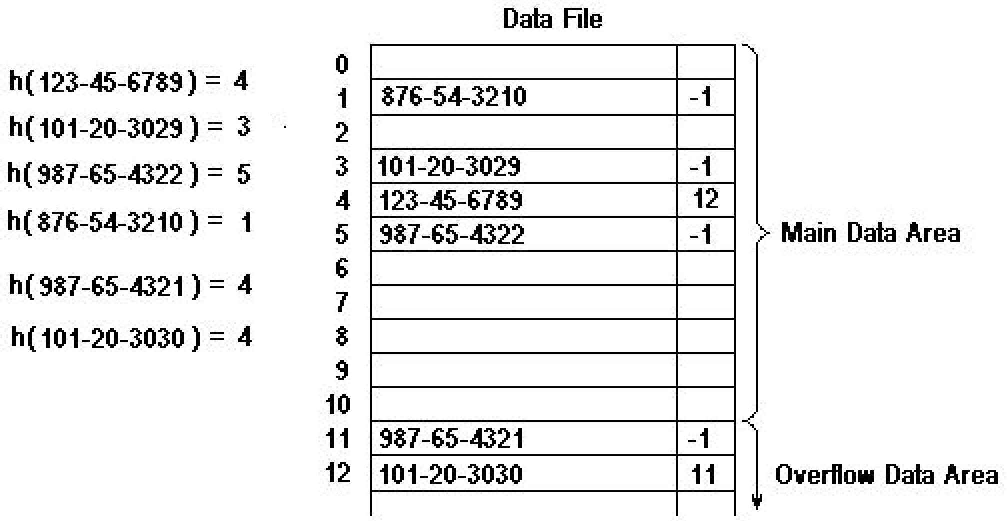
  A collision resolution scheme which applies a second hash function to keys which collide, to determine a probing distance.
- The use of double hashing will reduce the average number of probes required to find a record.
- Linear probing and double hashing are both referred to as *open addressing* collision handling methods.

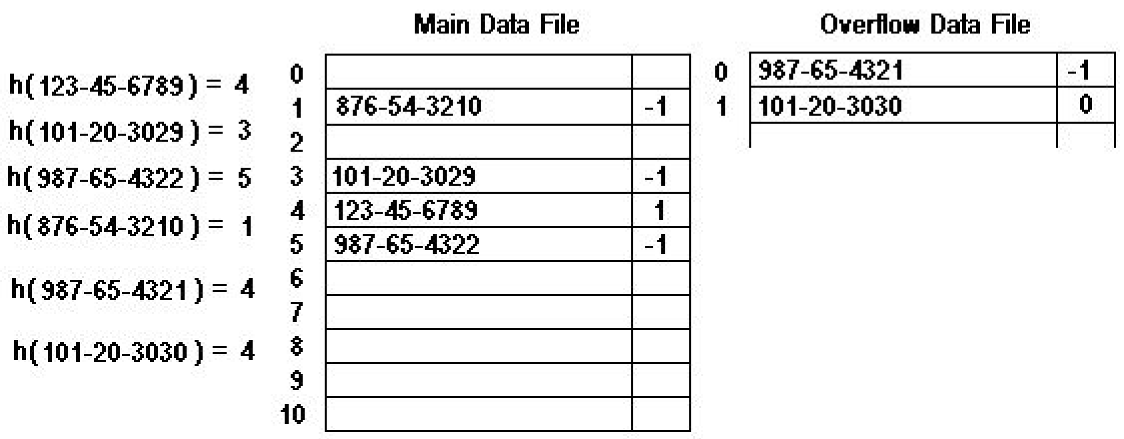


Open Chaining

- Open chaining forms a linked list, or chain, of synonyms.
- The overflow records can be kept in the same file as the hash table itself:
- The overflow records can be kept in a separate file:

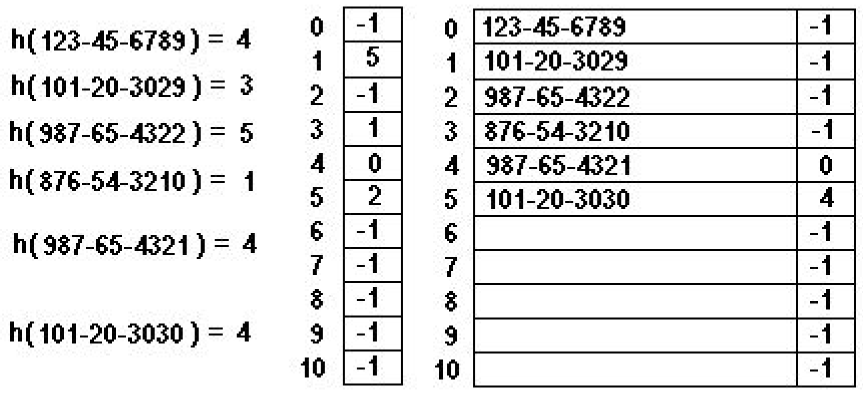






Scatter Tables

- If all records are moved into a separate "overflow" area, with only links being left in the hash table, the result is a *scatter table*.
- A scatter table scatter table is smaller than an index for the same data.



11.9 Patterns of Record Access

- Loading items in decreasing order of retrieval frequncy will improve the average access time.
- The idea is that records accessed more frequently can be accessed more quickly.
- This will lower the average retrieval time.

UNIT-8

Extendible Hashing

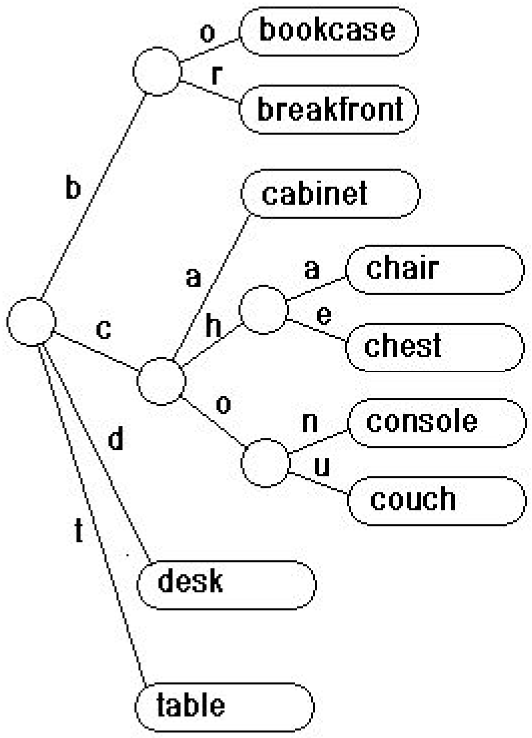
12.2 How Extendible Hashing Works

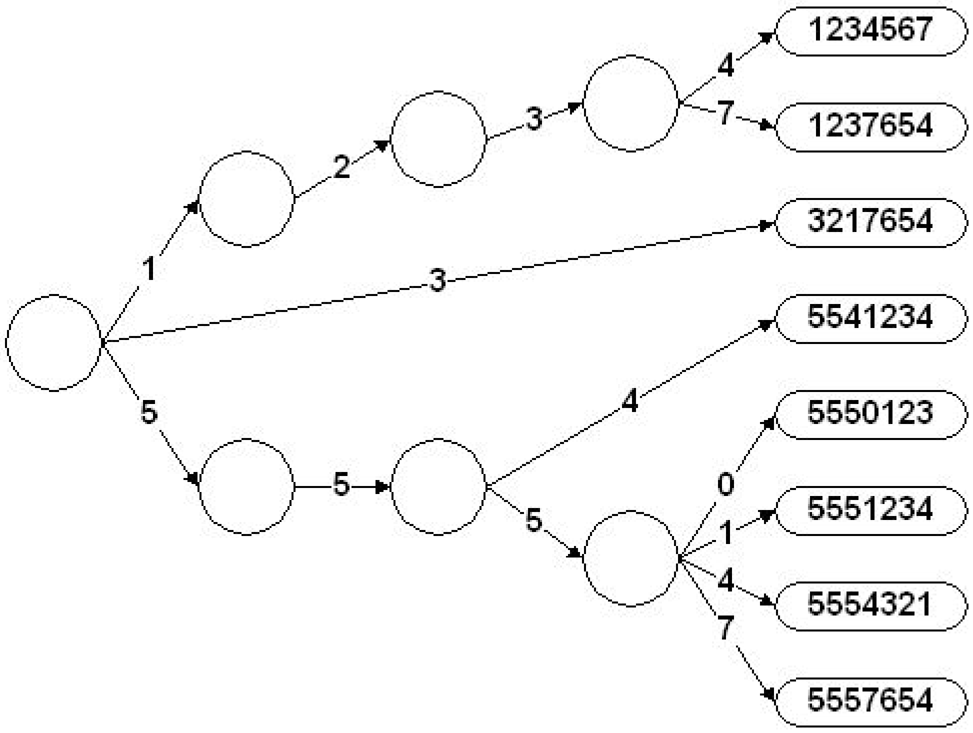
Tries

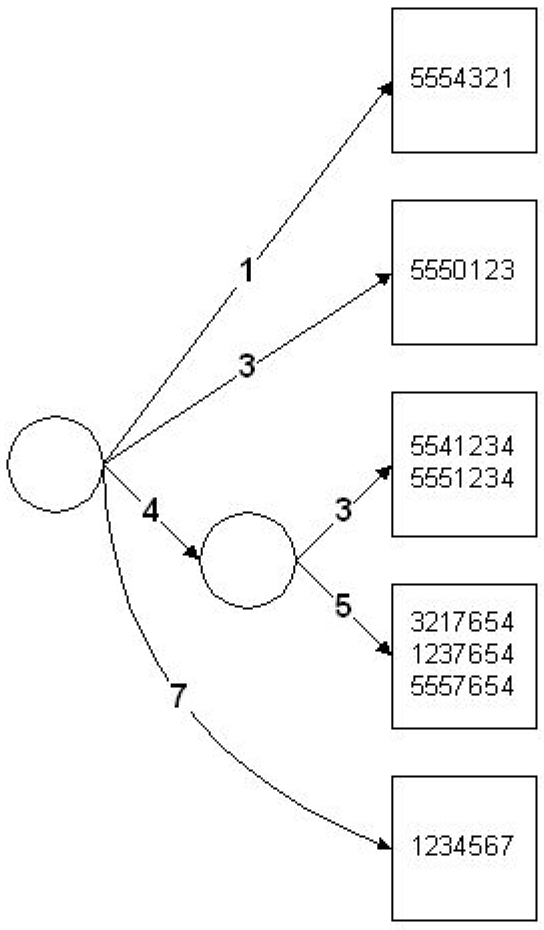
- **trie**

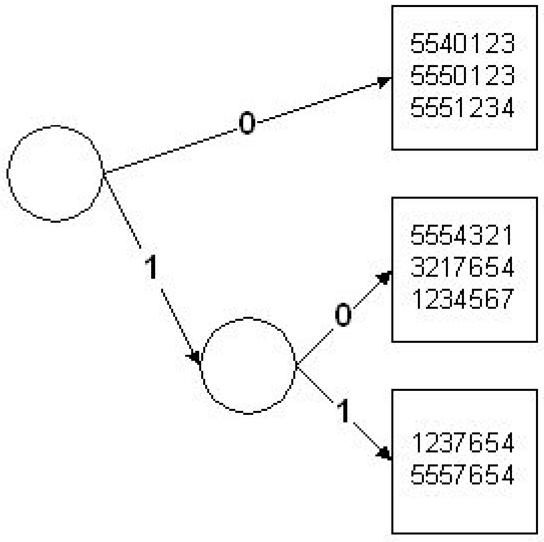
  A search tree in which the child of each node is determined by subsequent charaters of the key.
- An alphabetic (radix 26) trie potentially has one child node for each letter of the alphabet.
- A decimal (radix 10) trie has up to 10 children for each note.
- The trie can be shortened by the use of buckets.
- The bucket distribution can be balanced by the use of hashing.

Key	**Hash**
5554321	100111001
5550123	10111010
5541234	100111100
5551234	1011110
3217654	100111101
1237654	10011011
5557654	101110011
1234567	1101001


b
o
r
bookcase
breakfront
c
a
h
o
cabinet
a
e
chair
chest
n
u
console
couch
d
desk
t
table


1
2
3
4
7
1234567
1237654
3
3217654
5
5
4
5541234
5
0
1
4
7
5550123
5551234
5554321
5557654

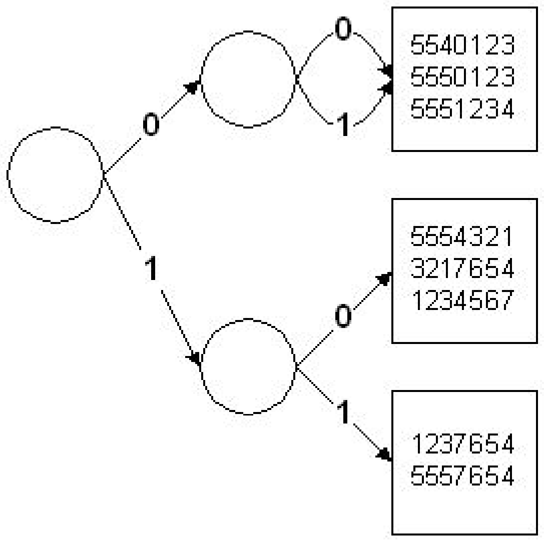


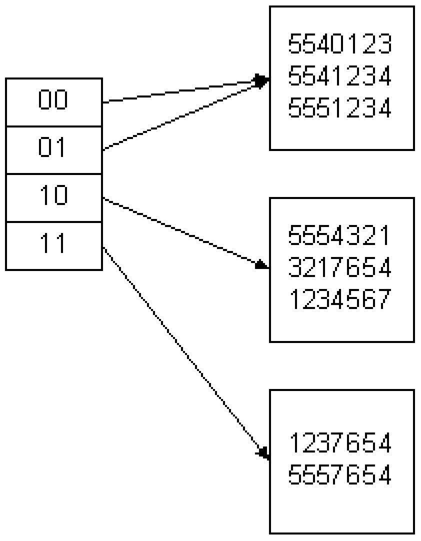


Turning the Tries into a Directory

- **extendible hashing**

  An application of hashing that works well with files that over time undergo substantial changes in size.

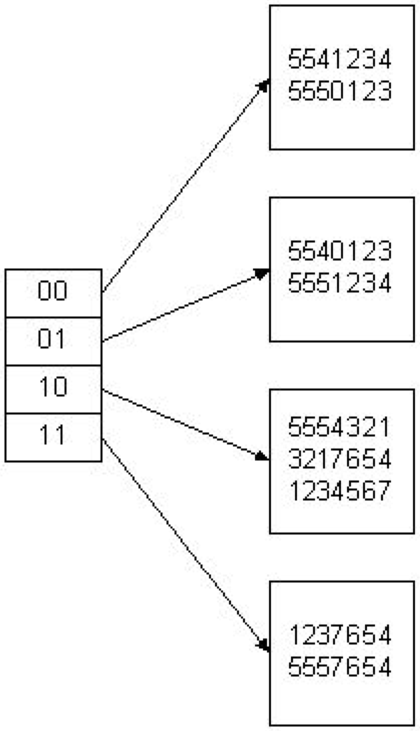


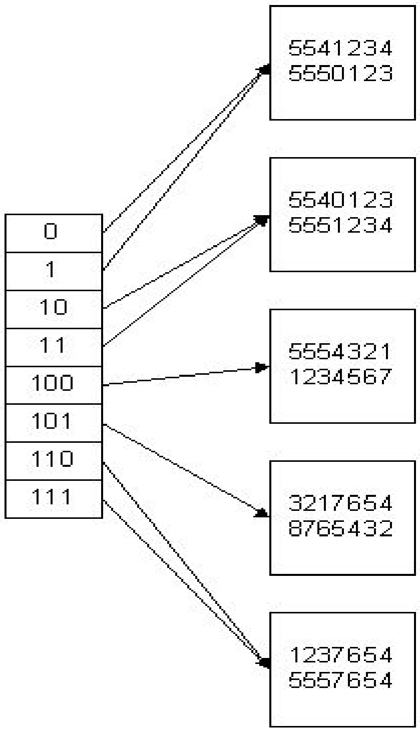


Splitting to Handle Overflow

- **splitting**

  Creation of a new node when a node overflows, with the partial distribution of the contents of the overflowing node to the new node.





Linear Hashing

linear hashing

An application of hashing in which the address space is extended by one bucket each time an overflow occurs