Maharaja Education Trust (R), Mysuru

Maharaja Institute of Technology Mysore

Belawadi, Sriranga Pattana Taluk, Mandya – 571 477

Approved by AICTE, New Delhi,
Affiliated to VTU, Belagavi & Recognized by Government of Karnataka

Lecture Notes on

FILE STRUCTURES (17IS62)

Prepared by

Department of Information Science and Engineering

Vision/ ಆಶಯ

"To be recognized as a premier technical and management institution promoting extensive education fostering research, innovation and entrepreneurial attitude"

ಸಂಶೋಧನೆ, ಆವಿಷ್ಕಾರ ಹಾಗೂ ಉದ್ಯಮಶೀಲತೆಯನ್ನು ಉತ್ತೇಜಿಸುವ ಅಗ್ರಮಾನ್ಯ ತಾಂತ್ರಿಕ ಮತ್ತು ಆಡಳಿತ ವಿಜ್ಞಾನ ಶಿಕ್ಷಣ ಕೇಂದ್ರವಾಗಿ ಗುರುತಿಸಿಕೊಳ್ಳುವುದು.

Mission/ ಧ್ಯೇಯ

- To empower students with indispensable knowledge through dedicated teaching and collaborative learning.
  ಸಮರ್ಪಣಾ ಮನೋಭಾವದ ಬೋಧನೆ ಹಾಗೂ ಸಹಭಾಗಿತ್ವದ ಕಲಿಕಾಕ್ರಮಗಳಿಂದ ವಿದ್ಯಾರ್ಥಿಗಳನ್ನು ಅತ್ಯುತ್ಕೃಷ್ಟ ಜ್ಞಾನಸಂಪನ್ನರಾಗಿಸುವುದು.
- To advance extensive research in science, engineering and management disciplines.
  ವೈಜ್ಞಾನಿಕ, ತಾಂತ್ರಿಕ ಹಾಗೂ ಆಡಳಿತ ವಿಜ್ಞಾನ ವಿಭಾಗಗಳಲ್ಲಿ ವಿಸ್ತೃತ ಸಂಶೋಧನೆಗಳೊಡನೆ ಬೆಳವಣಿಗೆ ಹೊಂದುವುದು.
- To facilitate entrepreneurial skills through effective institute - industry collaboration and interaction with alumni.
  ಉದ್ಯಮ ಕ್ಷೇತಗಳೊಡನೆ ಸಹಯೋಗ, ಸಂಸ್ಥೆಯ ಹಿರಿಯ ವಿದ್ಯಾರ್ಥಿಗಳೊಂದಿಗೆ ನಿರಂತರ ಸಂವಹನಗಳಿಂದ ವಿದ್ಯಾರ್ಥಿಗಳಿಗೆ ಉದ್ಯಮಶೀಲತೆಯ ಕೌಶಲ್ಯ ಪಡೆಯಲು ನೆರವಾಗುವುದು.
- To instill the need to uphold ethics in every aspect.
  ಜೀವನದಲ್ಲಿ ನೈತಿಕ ಮೌಲ್ಯಗಳನ್ನು ಅಳವಡಿಸಿಕೊಳ್ಳುವುದರ ಮಹತ್ವದ ಕುರಿತು ಅರಿವು ಮೂಡಿಸುವುದು.
- To mould holistic individuals capable of contributing to the advancement of the society.
  ಸಮಾಜದ ಬೆಳವಣಿಗೆಗೆ ಗಣನೀಯ ಕೊಡುಗೆ ನೀಡಬಲ್ಲ ಪರಿಪೂರ್ಣ ವ್ಯಕ್ತಿತ್ವವುಳ್ಳ ಸಮರ್ಥ ನಾಗರೀಕರನ್ನು ರೂಪಿಸುವುದು.

Department of Information Science and Engineering

VISION OF THE DEPARTMENT

To be recognized as the best centre for technical education and research in the field of information science and engineering.

MISSION OF THE DEPARTMENT

- To facilitate adequate transformation in students through a proficient teaching learning process with the guidance of mentors and all-inclusive professional activities.
- To infuse students with professional, ethical and leadership attributes through industry collaboration and alumni affiliation.
- To enhance research and entrepreneurship in associated domains and to facilitate real time problem solving.
-

PROGRAM EDUCATIONAL OBJECTIVES:

- Proficiency in being an IT professional, capable of providing genuine solutions to information science problems.
- Capable of using basic concepts and skills of science and IT disciplines to pursue greater competencies through higher education.
- Exhibit relevant professional skills and learned involvement to match the requirements of technological trends.

PROGRAM SPECIFIC OUTCOME:

Student will be able to

- **PSO1:** Apply the principles of theoretical foundations, data Organizations, networking concepts and data analytical methods in the evolving technologies.
- **PSO2:**Analyse proficient algorithms to develop software and hardware competence in both professional and industrial areas

Program Outcomes

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Course Overview

SUBJECT: FILE STRUCTURES **SUBJECT CODE: 17IS62**

It is relatively easy to come up with file structure designs that meet the general goals when the files never change. But when files grow or shrink, when information is added and deleted, it is much more difficult to handle the scenario, hence need of file structure arises. As files grew very large, unaided sequential access was not a good solution. Later Disks also allowed for direct access. Hence need of good file structure design (Such as sorting, indexing and hashing) need for the designing a solution model for a file structure problem.

File Structures describes the fundamental Concepts for storing and handling files. By using appropriate file structure and file organization, solution to be designed for real world problem. And also explains the Importance of file structure design in secondary storage devices. We learn how to apply the object oriented concepts as toolkit for various file structure problems.

Students are able to understand the Concept of file structure and secondary storage devices Students are also able to apply and analyze the appropriate design (Such as sorting, indexing and hashing) for storage and data manipulation using object oriented programming language.

Course Objectives

- Explain the fundamentals of file structures and their management.
- Measure the performance of different file structures.
- Organize different file structures in the memory.
- Demonstrate hashing and indexing techniques.

Course Outcomes

CO's	DESCRIPTION OF THE OUTCOMES
17IS62.1	Identify the appropriate concept of file structure design and secondary storage devices.
17IS62.2	Apply appropriate designs for storage and data manipulation with object oriented programming.
17IS62.3	Examine consequential processing and sorting of files using indexed methods.
17IS62.4	Analyze sorting and hashing technique for data handling.

Syllabus

SUBJECT: FILE STRUCTURES **SUBJECT CODE: 17IS62**

Topics Covered as per Syllabus	Teaching Hours
MODULE-1: INTRODUCTION	
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. **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.	**10 Hours**
MODULE-2: 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.	**10 Hours**
MODULE -3: Consequential 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. **Multi-Level Indexing and B-Trees:** The invention of B-Tree, Statement of the problem, Indexing with Binary Search Trees; Multi-Level Indexing, B-Trees, Example of Creating a B-Tree, An Object-Oriented Representation of B-Trees, B-Tree Methods; Nomenclature, Formal Definition of B-Tree Properties, Worstcase Search Depth, Deletion, Merging and Redistribution, Redistribution during insertion; B* Trees, Buffering of pages; Virtual B-Trees; Variable-length Records and keys.	**10 Hours**
MODULE-4: 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.	**10 Hours**
MODULE-5: Hashing &Extendible Hashing	
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. **Extendible Hashing:** How Extendible Hashing Works, Implementation, Deletion, Extendible Hashing Performance, Alternative Approaches.	**10 Hours**
List of 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	
List of 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.

List of URLs, Text Books, Notes, Multimedia Content, etc

1. https://www.tutorialspoint.com/dbms/dbms_file_structure.htm
2. File Structures using C++ by Dr. K.R.Venugopal

Maharaja Institute of Technology Mysore

Department of Information Science and Engineering

Index

SUBJECT: FILE STRUCTURES **SUBJECT CODE: 17IS62**

MODULE- 1

CHAPTER –1: INTRODUCTION

1.1 The Heart of File Structure Design:

Disks (Magnetic disks/Optical disks) are slow. The time it takes to get information from Random Access Memory (RAM) is about 120th billionths of second. Getting the same information from a typical disk might take 30 milliseconds, or 30 thousandths of a second. The disk access is quarter of a million times longer than the memory access. On the other hand disk provides enormous capacity at much less cost than memory and they are non-volatile. The tension between a disk's relatively slow access time and its enormous, nonvolatile capacity is the driving force behind file structure design. Good file structure design will give us access to all the data without making our application spend a lot of time waiting for the disk.

File structure *is a combination of representations for data in files and of operations for accessing the data.*

1.2 Short History of file structure design

1. Early work: Early work assumed that files were on tapes. Access was sequential and the cost of access grew in direct proportion to the size of file.

2. Emergence of Disks and indexes: Sequential access was not a good solution for large files. Disks allowed for direct access. Indexes made it possible to keep a list of keys and address of records in a small file that could be searched very quickly. With the key and address the user had direct access to the large primary file.

3. Emergence of Trees: As indexes also grew too large to they became difficult to handle**.** Sorting the indexes took too much time and reduced the performances. Idea of using tree structures to manage the indexes emerged in early 1960's. Initially BST were used for storing the records in the file. This resulted in uneven growth of trees in-turn resulted in long searches require many disk access to find a record. Than AVL trees were used. AVL trees are balanced trees. The problem of uneven growth was resolved. The AVL trees are suitable for data in memory but not for data in file. In 1970s came the idea of B trees and B+ trees which require an O ($\log_k$ N) access time. Still the efficiency was dependent on the size of the file. i.e 'N'. As the 'N' (number of records increased) the efficiency decreases.

4. Hashing: Retrieving records in a single access to a file. Ideally Hashing has an efficiency of O(1). Hashing is a good for files that do not change size greatly over time, but do not work well with dynamic files. Extendible hashing helps in overcoming the limitation of Hashing.

1.3 A Conceptual Toolkit

- ➢ Design problems & design tools •
  - Decrease the number of disk accesses by collecting data into buffers, blocks, or buckets •
  - Manage the growth of these collections by splitting them

=> Finding new ways to combine these basic tools of file design

- ➢ Conceptual tools •
  - methods of framing and addressing a design problem •
  - Each tool combines ways of representing data with specific operations

- ➢ Chapters 2 ~ 6 •
  - basic tools
- ➢ Chapters 7 ~ 11 •
  - highlights of file structure design •
  - showing how the basic tools are used to handle efficient sequential access

CHAPTER – 2: FUNDAMENTAL FILE PROCESSING OPERATIONS

2.1 Physical File and Logical File

A file that actually exists on secondary storage. It is the file as known by the computer OS and that appear in its file directory. A collection of bytes stored on a disk or tape.

Logical file

The file as seen by a program. The use of logical file allows a program to describe operations to be performed on a file without knowing what physical file will be used. A "channel" (Telephone line) that hides the details of the file location and physical format to the program. This logical file will have logical name which is what is used inside the program.

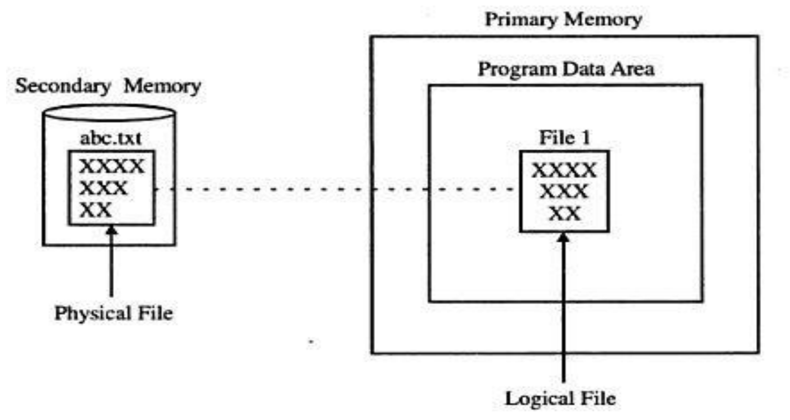


Figure 1.1 Relationship between Physical File and Logical File

2.2Opening 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 : O_TRUNC:

Append every write operation to the end of the file 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 theprogramming 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:	R W E	R W E	R W E
	1 1 1	1 0 1	0 0 1
	Owner	Group	Others

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:

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.

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

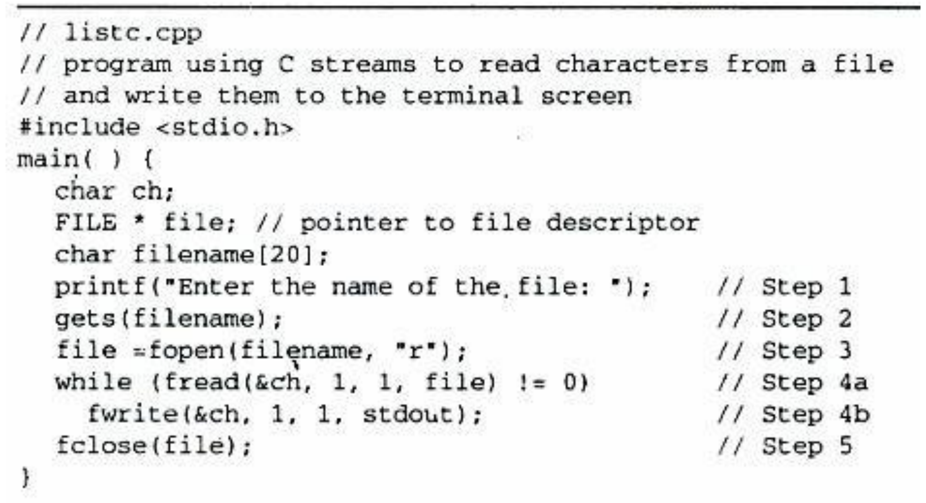
The first simple file processing program opens a file for input and reads it, character by Character, sending each character to the screen after it is read from the file. This program includes the following steps

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.

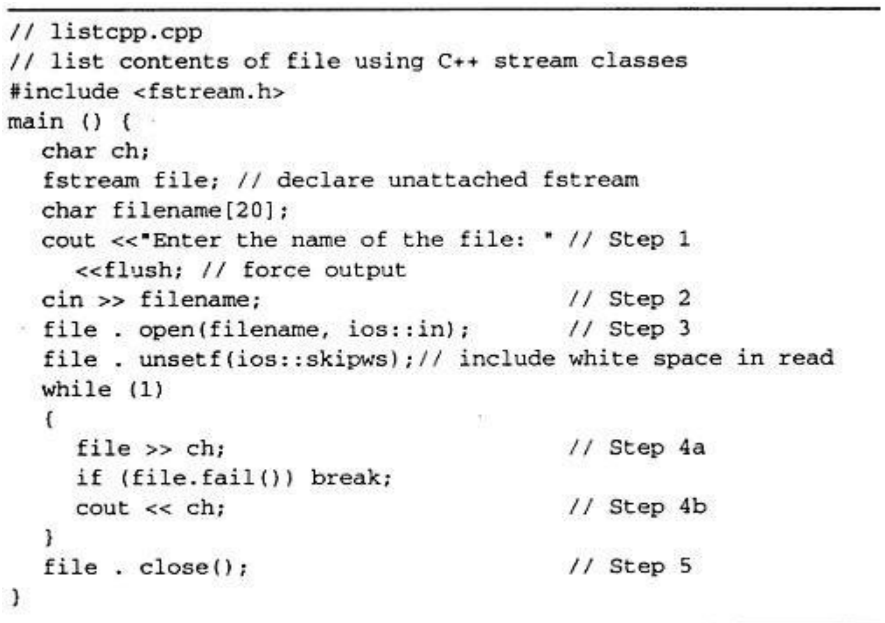
5. Close the input file.

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.

2.5.6.1 The acronym for end-of-file is EOF.

2.5.6.2 When a file reaches EOF, no more data can be read.

2.5.6.3 Data can be written at or past EOF.

2.5.6.4 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:

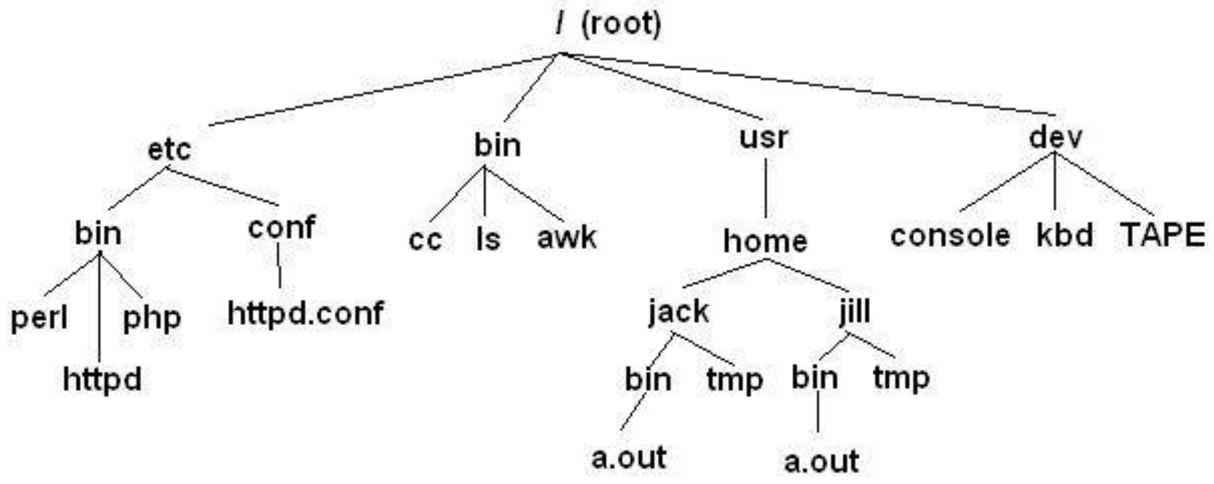
```
file.seekg(byte_offset,origin)
file.seekp(byte_offset,origin)
```

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.

1. 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

CHAPTER – 3: SECONDARY STORAGE AND SYSTEM SOFTWARE

3.1 Disks

Disks belong to direct access storage devices because they make it possible to access the data directly. Different types of disks are:

Hard disk: High capacity + Low cost per bit

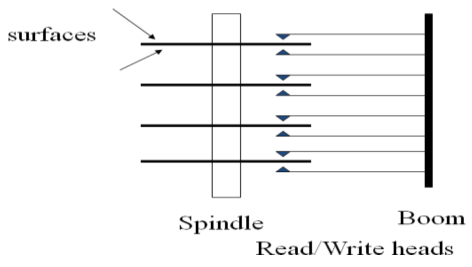
Floppy disk: cheap, slow and holds very limited data.

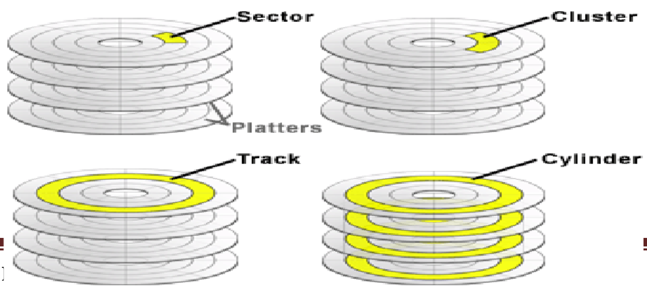
Optical disk: Read only, holds more data and can be reproduced cheaply, it is slow.

3.1.1 Organisation of Disks

The information stored on a disk is stored on the surface of one or more platters. Disks drives typically have a number of platters. The platters spin at around 5000 rpm. Platters contain contains concentric tracks on both the surface i.e top and bottom surface of platters. The tracks that are directly above and below one another forms a cylinder. The significance of the cylinder is that all the information on a single cylinder can be without moving the arm that holds the read write head. Moving the arm is called seeking and is the slowest part of accessing information from a disk. Each track is divided into number of sectors. A sector is smallest addressable unit of the disk. When a program reads a byte from the disk, the operating system locates the surface, track and sector containing that byte, the arm assembly is moved in or out to position a head on a desired track and reads the entire sector into a special area in main memory called buffer.

The bottleneck of a disk access is moving the read/write arm. So it makes sense to store a file in tracks that are below/above each other on different surfaces, rather than in several tracks on the same surface. Disk controllers: typically embedded in the disk drive, which acts as an interface between the CPU and the disk hardware. The controller has an internal cache (typically a number of MBs) that it uses to buffer data for read/write requests.





3.1.2 Estimating Capacities and space needs

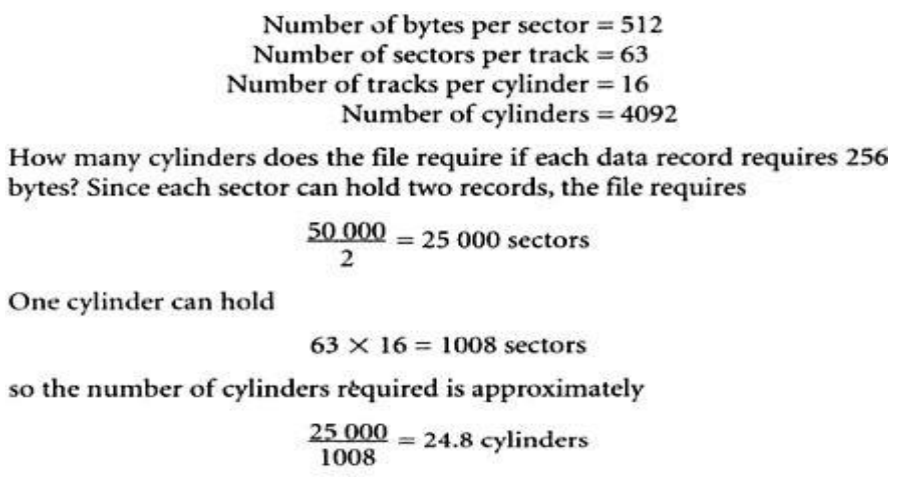
Track capacity = number of sectors/track * bytes/sector

Cylinder capacity = number of tracks/cylinder * track capacity

Drive capacity = number of cylinders * cylinder capacity

Number of cylinders = number of tracks in a surface.

Given a disk with following characteristics



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

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

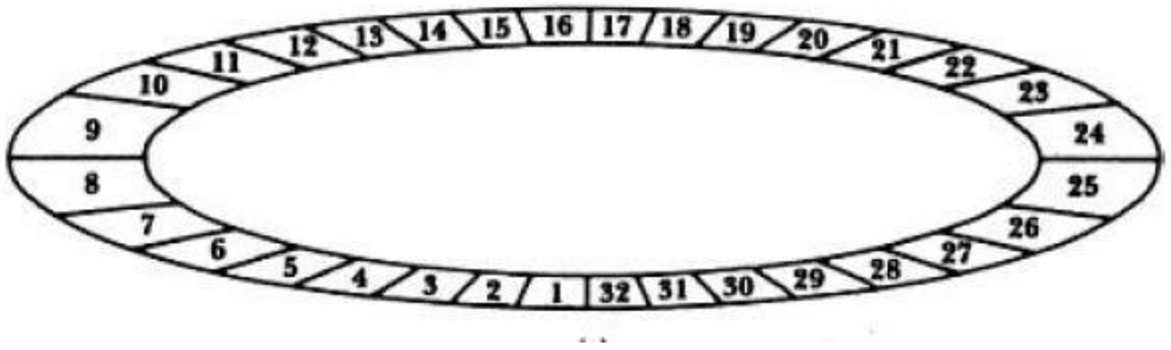
so the number of cylinders required is approximately

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

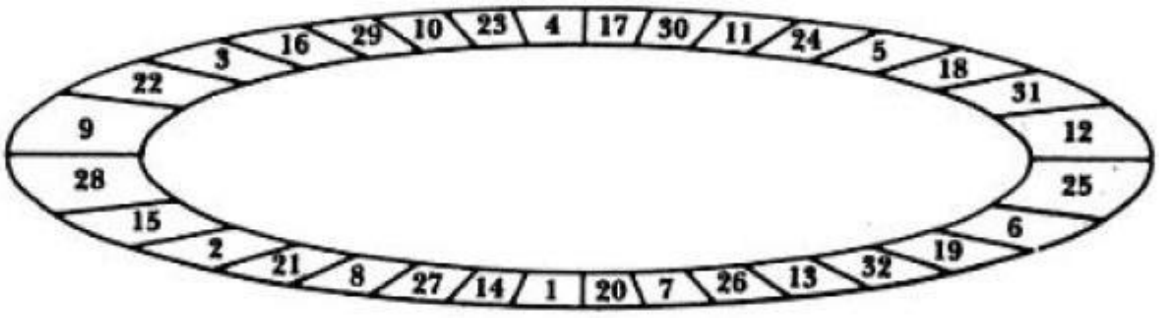
3.1.3 Organizing Tracks by sector

Two ways to organize data on disk: by sector and by 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. When you want to read a series of sectors that are all in the same track, one right after the other, we often cannot read adjacent sectors. After reading the data, it takes the disk controller a certain amount of time to process the received information before it is ready to accept more. if logically adjacent sectors are placed physically adjacent, we would miss start of the next sector while we were processing the sector that we had just read. w.r.t the given figure it takes thirty-two revolutions to read the entire 32 sectors of a track.



O/O system designers have solved this problem by interleaving the sectors: leaving an interval of several physical sectors between logically adjacent sectors. Figure below illustrates the assignment of logical sector content to the thirty-two physical sectors in a track with interleaving factor of 5. It takes five revolutions to read the entire 32 sectors of a track.

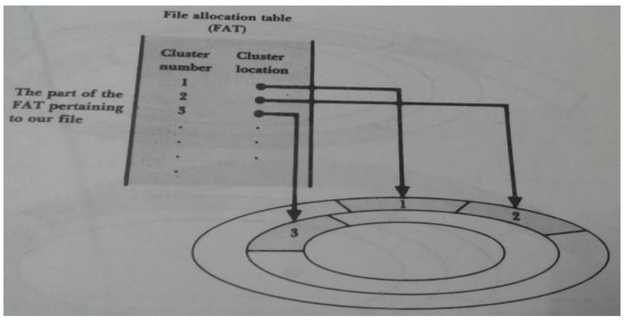


In the early 1990s, controller speeds improved so that disks can now offer 1:1 interleaving. This means that successive sectors are physically adjacent, making it possible to read entire track in a single rotation of the disk.

Clusters

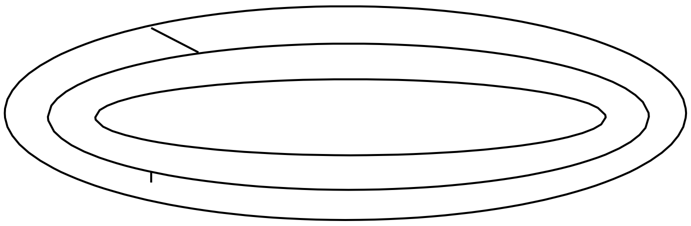
Another view of sector organization, designed to improve performance, is clusters. A cluster is a fixed number of contagious sectors. Once a given cluster has been found on a disk, all sectors in that cluster can be accessed without requiring additional seek.

To view a file as a series of clusters and still maintain the sectored view the file manager ties logical sectors to physical clusters they belong to by using a file allocation table (FAT). The FAT contains a list of all the clusters in a file. ordered accorind to the logical order of the sectors they contain. With each cluster entry in the FAT is an entry giving the physical location of the cluster.

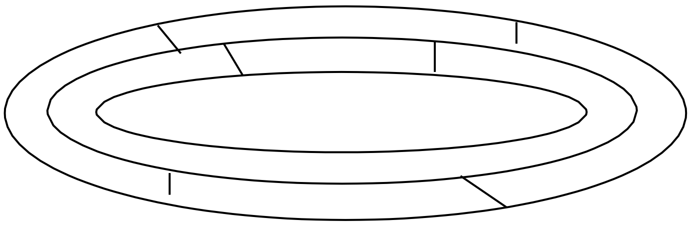

File allocation table (FAT)
Cluster number
Cluster location
The part of the FAT pertaining to our file
1
2
3
1
2
3

Extents

In case of availability of sufficient free space on a disk, it may be possible to store a file entirely in contagious clusters. Then we say that the file consists of one extent. All of its sectors, tracks and (if it is large) cylinders form one contagious whole. Then whole file can be accessed with minimum amount of seek.



File with extent 1(one)



File with extent 3

In case of non availability of enough contagious space to store entire file, the file is divided into two or more contagious parts. Each part is an extent. As the number of extents increase for a file, the file becomes more spread out on the disk, and the amount of seeking increases.

Fragementation

All of the sectors on a given disk contain same number of bytes. There are two possible organizations for records: (if the records are smaller than the sector size)

1. Store one record per sector.

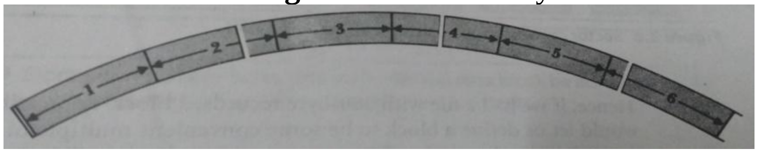
Advantage: each record canbe retrieved by reading one sector.

Disadvantage: Loss of space within each sector. (Internal Fragmentation)

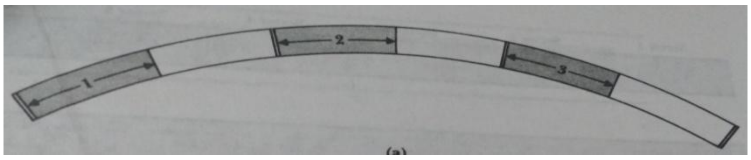
2. Store records successively (i.e one record may span across two sectors)

Advantage: No Internal fragmentation

Disadvantage: Two sectors may need to be accessed to retrieve a single record.



1 record per sector

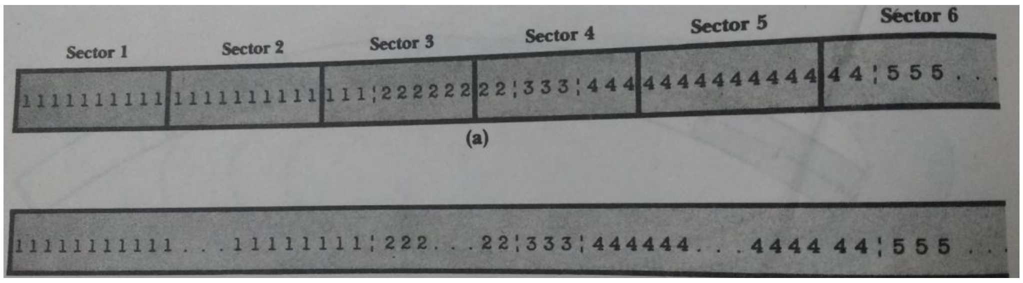


Record spanning across sectors

3.1.4 Organizing Tracks by blocks

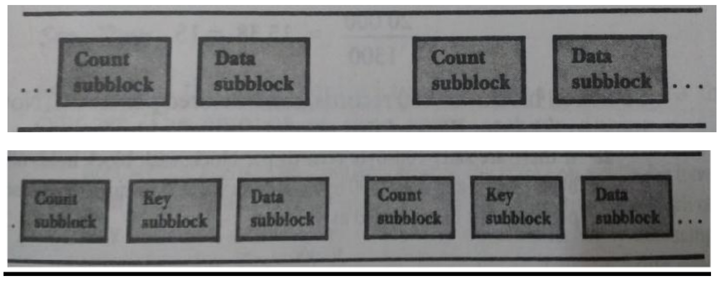
Rather than dividing tracks into sector, the tracks can be divided into blocks whose size can vary. Blocks can be either fixed or variable in length, depending on the requirements of the file structure designer and the capabilities of Operating system.

Block doesn't have record spanning across sectors and fragmentation problem of sectors, since they vary in size to fit the logical organization of data. The term blocking factor indicates the number of records that can be stored in each block.



Each block is usually accompanied by sub blocks containing extra information about the data block such as:

1. **Count sub-block:** Contains number of bytes in accompanying data block.
2. **Key sub-block:** contains the keys of all the records that are stored in the following data block.



3.1.5 Nondata Overhead

Both block and sectors require that a certain amount of space be taken up on the disk in the form of non-data overhead. Some of the overhead consists of information that is stored on the disk during preformatting.

On sector addressable disks, preformatting involves storing (at the beginning of each sector)

- Sector address.
- Track address.
- Condition of the sector (bad sector or usable).

On block organized disks:

- Sub block.
- Inter block gaps. (as shown in the above figure)

Relative to sector addressable disks block addressable disks have more non data overhead.

3.1.6 The cost of 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.

Seek Time: is the time required to move the access arm to the correct cylinder. if we are alternately accessing sectors from two files that are stored at the opposite extremes on a disk (one on the inner most cylinder, one on the outer most cylinder), seeking is very expensive. Most hard disks available today have average seek time of less than 10 milli seconds and high performance hard disks have average seek time as low as 7.5 msecs

Rotational Delay: refers to the time it takes for the disk to rotate so the sector we want is under the read/write head. Hard disk with rotation speed of 5000 rpm takes 12 msecs for one rotation. on average, the rotation delay is half a revolution, or 6 msec.

Transfer time: Once the data we want is under the read/write head, it can be transferred.

3.1.7 Effect of Block Size on Performance: A Unix Example

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

3.1.8 Disks as Bottleneck

Processes are often disk bound. i.e the cpu often has to wait long period of time for the disk to transmit data. Then the cpu processes the data. Solution to handle disk bottleneck are:

Solution 1: Multi-programming (CPU works on other jobs while waiting for the disk)

Solution 2: Stripping: Disk stripping involves splitting the parts of a file and storing on several different drives, then letting the separate drives deliver parts of the file to CPU simultaneously(It achieves parallelism)

Solution 3: RAID: Redundant array of independent disks

Solution 4: RAM disks: Simulate the behaviour of mechanical disk in main memory (provides faster access)

Solution 5: Disk cache: Large block of main memory configured to contain pages of data from a disk. First check cache for required data if not available, then go to disk and replace some page in cache with the page from the disk containing the required data.

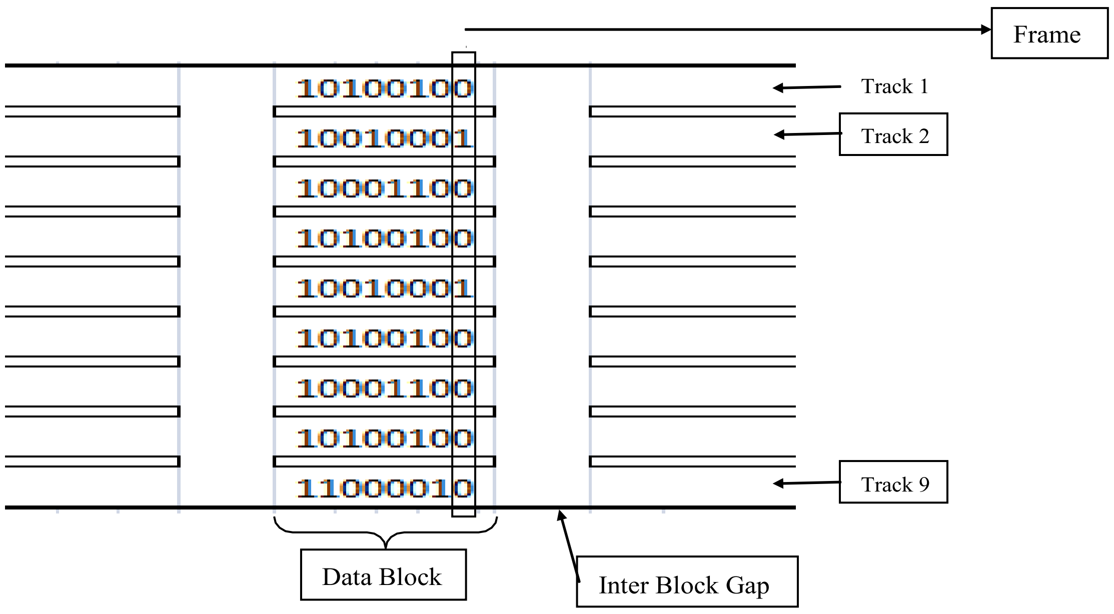
3.2 MAGNETIC TAPES

Magnetic tape units belong to a class of devices that provide no direct accessing facility but can provide very rapid sequential access to data. Tapes are compact stand up well under different environment conditions, easy to store and transport. Tapes are widely used to store application data. Currently tapes are used as archival storage.

3.2.3 Organization Of Data On Nine Track Tape:

Since tapes are accessed sequentially there is no need for addresses to identify location of data on tape. ON a tape logical position of a byte within a file corresponds directly to its physical position relative to the start of the file.

The surface of the typical tape can be seen as a set of parallel tracks, each of which is a sequence of bits. In a nine track tape the nine bits that are at the corresponding position in the nine respective tracks are taken to constitute one byte, plus a parity bit. So a byte can be thought of as a one bit wide slice of tape. Such a slice is called frame. Parity bit is not part of the data. It is used to check the validity of the data.



Frames are organized into data blocks of variable size separated by interblock gaps (Long enough to start/accelerate and stop/decelerate). Tapes cannot start and stop instantaneously.

3.2.4 Estimating Tape Length Requirements

Length of the magnetic tape is given by - s

s = n x (b+g) n=number of data blocks, g=inter block gap, b=physical length of data block

3.2.5 Estimating Data Transmission Times

Effective transmission rate = effective recording density (bpi) x tape speed (ips)

For problem related to Magnetic Tapes refer to Class notes.

3.3 Disk versus Tapes

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

CD-ROM: Compact disk read only memory. A single disk can hold approximately 700MB of data. CD-ROM is read only. It is publishing medium rather than a data storage and retrieval like magnetic disks.

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

CD-ROM is the child of CD audio. Audio discs are designed to play music, not to provide fast, random access to data. This biases CD toward having storage capacity and moderate data transfer rates and against decent seek performance**.**

3.5.1 Reading Pits and Lands:

CD ROMs are stamped from a glass master disk which has a coating that is changed by the laser beam. When the coating is developed, the areas hit by the laser beam turn into pits along the track followed by the beam. The smooth unchanged areas between the pits are called lands.

When we read the CD we focus a beam of laser light on the track as it makes under the optical pickup (read/write head). The pits scatter the light but the lands reflect the light back to optical pickup. High and low intensity reflected light is the signal used to reconstruct the original digital information.

Nominal transmission rate = tape density (bpi) x tape speed (ips)

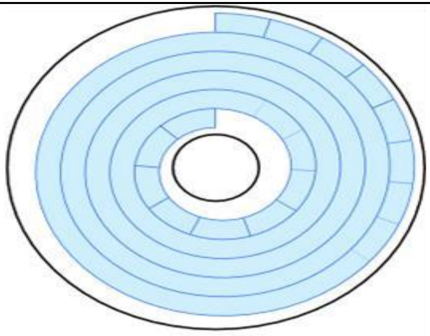
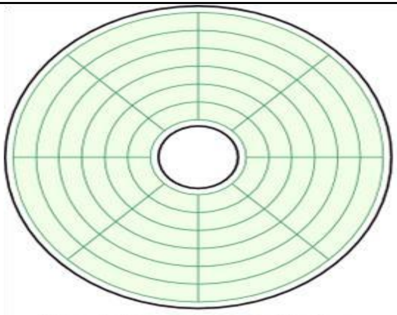
1's are represented by transition from pit to land and back again. Every time the light intensity changes we get 1. The 0s (Zeros) are represented by the amount of time between the transitions. The longer between transitions, the more 0s (Zeros) we have.

Given this scheme it is not possible to have two adjacent 1's – 1's are always separated by 0s.In face due to the limits of the resolution of the optical pickup, there must be at least two 0s between any pair of 1s. This means that the raw patterns of 8 bits 1s and 0s has to be translated so that at least 2 0s (zeros) separate consecutive 1s.

This translation is done using EFM (Eight to Fourteen Modulation) encoding lookup table. The EFM transition scheme refers lookup table, turns the original 8 bits of data into 14 expanded bits that can be represented in the pits and lands on the disc.

3.5.2 Constant Linear Velocity and Constant Angular Velocity

CLV	CAV
The data is stored on a single spiral track that winds for almost 3 miles from the centre to the outer edge of the disc.	The data is stored on 'n' number of concentric tracks and pie shaped sectors.
All the sectors take same amount of space.	Inner sectors take less space compared to outer sectors.
Storage capacity of all the sectors is same.	Storage capacity of all the sectors is same.

All the sectors are written at maximum density (Constant data density)	Writes data less densely in outer sectors and more densely at inner sectors. (Variable data density)
Due to Constant data density space is not wasted in either inner or outer sectors.	Due to Variable data density space is wasted in outer sectors.
Constant data density implies that disc has to spin more slowly when reading data at outer sectors compared to reading at the inner (center) sectors. (Variable speed of disc rotation)	Variable data density implies that disc rotates at constant speed irrespective of reading from inner sectors or outer sectors.
Poor seeking performance	Seeking is fast compared to CLV
 Constant linear velocity disc	 Constant angular velocity disc

3.5.3 Addressing

In CD audio each sector is of 2 Kilo Bytes. 75 sectors create 1 second of audio playback. According to original Philips/Song standards, a CD whether used for audio or CD-ROM, contains at least one hour of playing time. That means the disc is capable of holding at least 540000 kilo bytes of data.

1 second = 75 sectors

1 minute = 4500 sectors
60 minutes - ? (60x4500 = 270000 (Sectors)

60 seconds = ? (60x75 = 4500 sectors)

270000 sectors x 2 kilo bytes = 540000 kilo byes => 540 mega bytes.

Sectors are addressed by min:sec:sector (example- 34th Sector of 10th second 16th minute is addressed as 16:10:34)

3.5.4 Structure of a CD-ROM sector:

12 bytes Synch	4 bytes sector ID	2048 bytes user data	4 bytes error detection	8 bytes null	276 bytes error correction

3.6 CD-ROM strength and Weaknessess

3.6.1 Seek Performance: Random access performance is very poor. Current magnetic disk technology has an average random data access time of about 30 msecs whereas CD ROM takes 500 msecs to even 1 second.

3.6.2 Data Transfer rate: Not terribly slow not very fast**.** It has modest transfer rate of 75 sectors or 150 kilo bytes per second. Its about 5 times faster than transfer rate of floopy discs and an order of magnitude slower than rate for good Winchester disks.

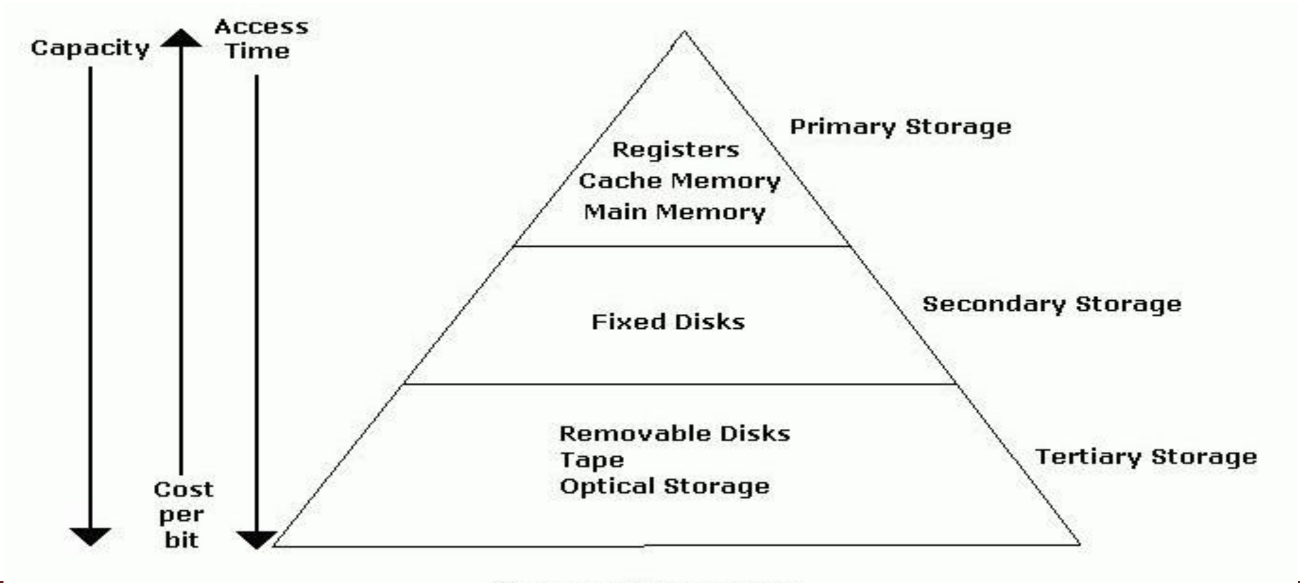
3.6.3 Storage Capacity: Holds approximately 700 MB of data. Large storage capacity for text data. Enables us to build indexes and other support structures that cann help overcome some of the limitations associated with CD-ROM's poor seek performance.

3.6.4 Read-Only Access: CD- ROM is a publishing medium**,** a storage device that cannot be changes after manufacture. This provide significant advantages such as:

a. We never have to worry about updating.
b. This simplifies the file structure and also optimizes our index structures and other aspects of file organization.

3.6.5 Asymmetric reading and writing: For most media, files are written and read using the same computer system. Often reading and writing are both interactive so there is a need to provide quick response to the user. CD-ROM is different. We create the files to be placed on the disc once, then we distribute the disc, and it is accessed thousands of times.

3.7 Storage as a Hierarchy

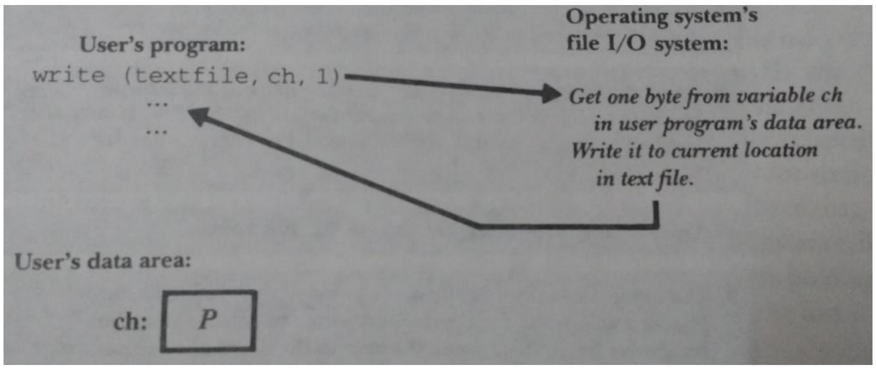


Memory Hierarchy

3.8 A JOURNEY OF A BYTE

What happens when the following statement in the application program is executed?

write(fd,ch,1)



User's program:
write (textfile, ch, 1)
...
...
Operating system's file I/O system:
Get one byte from variable ch in user program's data area.
Write it to current location in text file.
User's data area:
ch: P

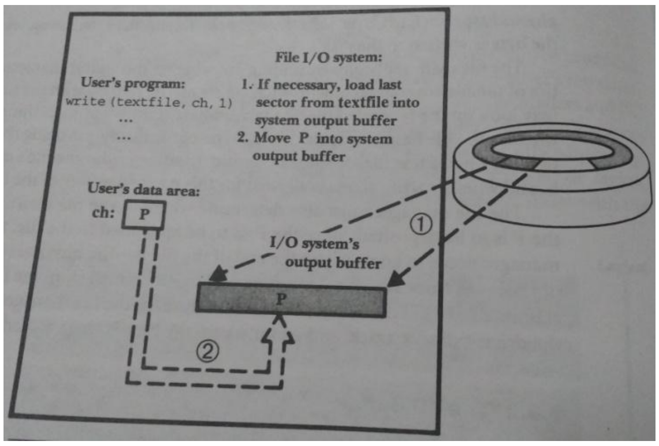
1. The program asks the operating system to write the contents of the variable c to the next available position in the file.

2. The operating system passes the job on to the file manager.

3. The file manager looks up for the given file in a table containing information about it, such as whether the file is open and available for use, what types of access are allowed, if any, and what physical file the logical name fd corresponds to.

4. The file manager searches a file allocation table for the physical location of the sector that is to contain the byte.

5. The file manager makes sure that the last sector in the file has been stored in a system I/O buffer in RAM, than deposits the 'P' into its proper position in the buffer.



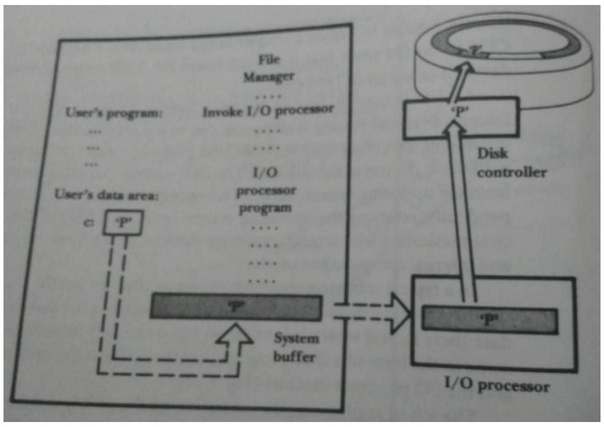
File I/O system:
User's program:
write (textfile, ch, 1)
...
...
1. If necessary, load last sector from textfile into system output buffer
2. Move P into system output buffer
User's data area:
ch: P
①
I/O system's output buffer
P
②

6. The file manager gives instructions to the I/O processor abou where the byte is stored in RAM and where it needs to be sent on the disk.

7. The I/O processor finds a time when the drive is available to receive the data and puts the data in proper format for the disk. It may also buffer the data to send it out in chunks of the proper size for the disk.

8. The I/O processor sends the data to the disk controller.

9. The controller instructs the drive to move the read/write head to the proper track, waits for the desired sector to come under the read/write head, than sends the byte to the drive to be deposited bit by bit on the surface of the disk.



3.9 BUFFER MANAGEMENT

Buffer •

- the part of main memory available for storage of copies of disk blocks •
- not controlled by programmers, but by the operating system

Buffering: Buffering involves working with a large chunk of data in memory so the number of accesses to secondary storage can be reduced.

3.9.1 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. Strategies to avoid this problem:

3.9.2 Buffering strategies

-Multiple buffering

- Double Buffering

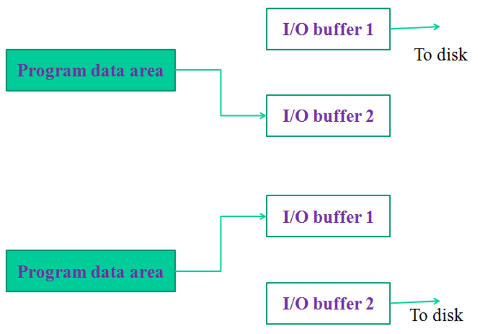
-Buffer Pooling

Move mode and Locate mode

Scatter/Gather I/O

Multiple buffering:

Suppose that a program is only writing to a disk and that it is I/O bound. The CPU wants to be filling a buffer at the same time that I/O is being performed. If two buffers are used and I/O-CPU overlapping is permitted, the CPU can be filling one buffer while the contents of the other are being transmitted to disk. When both tasks are finished, the roles of the buffers can be exchanged. This method is called double buffering. This technique need not be restricted to two buffers.

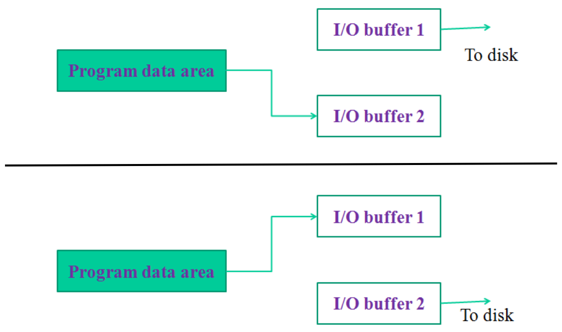


Some file system use a buffering scheme called buffer pooling. Buffer pooling:
There is a pool of buffers. When a request for a sector is received, O.S. first looks to see that sector is in some buffer. If not there, it brings the sector to some free buffer. If no free buffer exists, it must choose an occupied buffer. (Usually LRU strategy is used).

Double buffering: •

- the method of swapping the roles of two buffers after each output (or input) operation
  - O.S. : operating on one buffer •
  - I/O system : loading or emptying the other buffer

Overlapping



Buffer pooling •

- when a system buffer is needed, it is taken from a pool of available buffers and used •
- buffer selection for replacement : least recently used (LRU)

Buffer handling (by file manager)

1. **Move mode** •

- data : system buffer (RAM) <=> program buffer (RAM)
    - (= program's data area) •
- take the amount of time

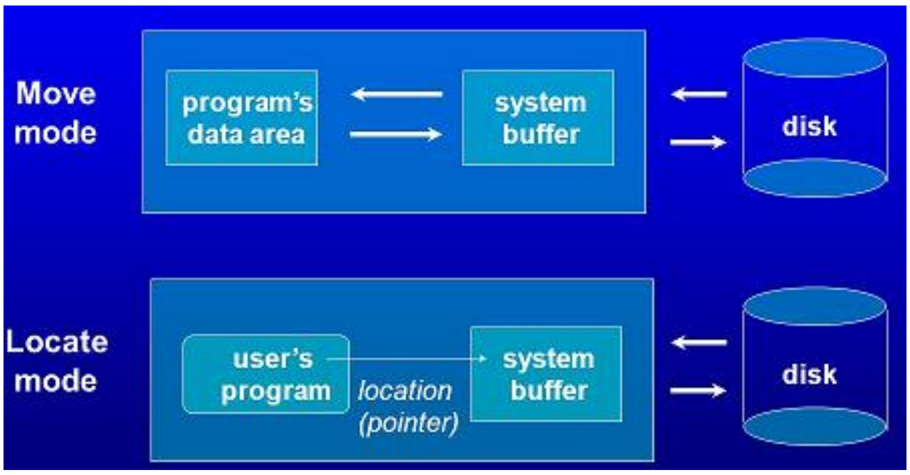
2. **Locate mode**

(1) data : secondary storage <=> program's data area

- no extra move

(2) use system buffers to handle all I/O, but provide the program with the locations of the system buffers

- program can operate directly on data in the I/O buffer



➢ Block (header + data) •

headers ... in one buffer

data ... in a different buffer

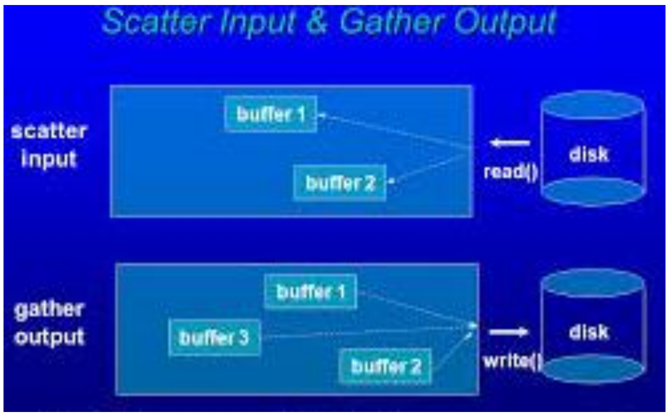
➢ two-step process

(i) read whole block into a single big buffer

(ii) move the different parts to their own buffers

➢ Two methods (to reduce the running time)

1. scatter input : readv( ) •
    - a single READ call identifies a collection of buffers into which data from a single block is to be scattered
2. gather output : writev( ) •
    - several buffers can be gathered and written with a single WRITE call



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

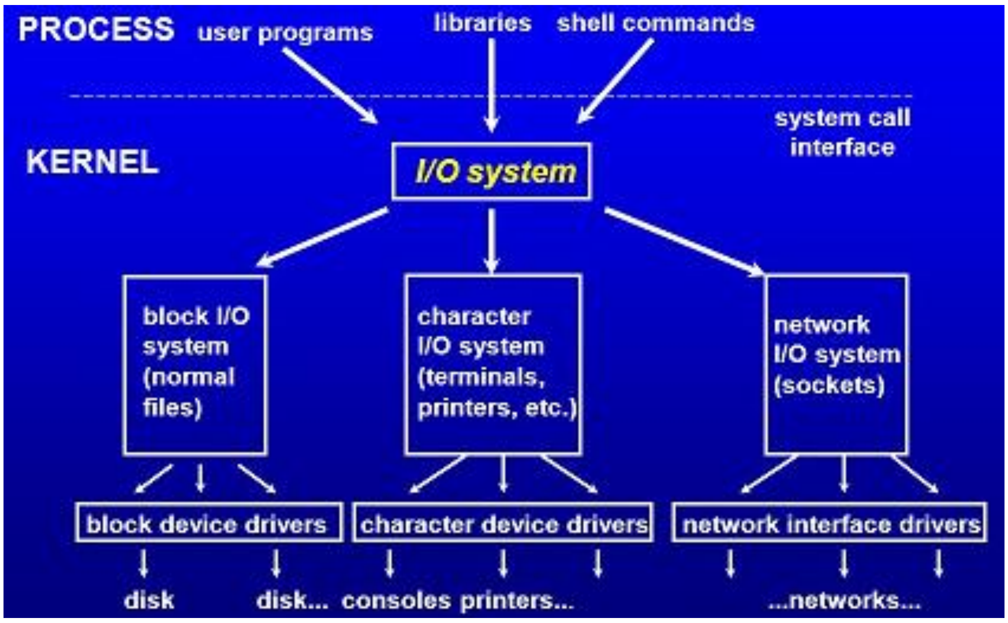
Input or output performed character by character.

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

3.10.1 The Kernal

Kernel I/O Structure

1. topmost layer •
   - deal with data in logical, structural terms (e.g., name, a body of text, an image, an array of numbers, or some other logical entity) •
   - application view on a file •
   - processes ... shell routines (cat, tail), user programs, library routines (scanf(), fread())
2. bottom layers •
   - carry out the task of turning the logical object into a collection of bits on a physical device •
   - system view on a file •
   - UNIX Kernel ... views all I/O as operating on a sequence of bytes



❖ Journey of a byte •

➢ write (fd, &ch, 1); (vs. fprintf(): library call)

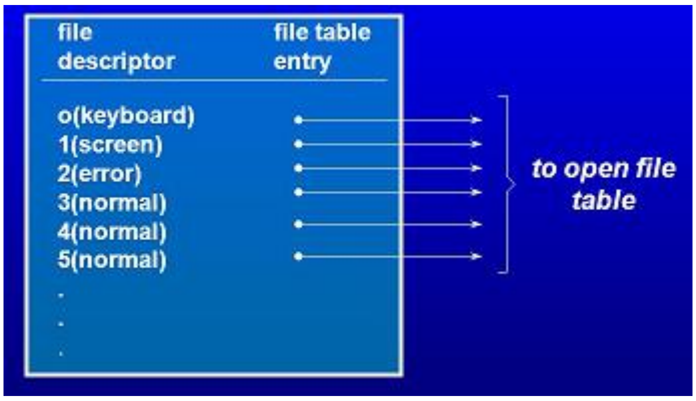
(i) the system call instructs the kernel to write a character to a file

(ii) the kernel I/O system begins by connecting the file descriptor (fd) to some file or device (scan a series of 4 tables) •

- file descriptor table •
- open file table •
- file allocation table •
- table of index nodes (inode table)

❖ File descriptor table •

- owned by the process (your program) •
- associates each of the file descriptors used by a process with an entry in the open file table •
- every process has its own descriptor table •
- includes entries for all opened files, including stdin, stdout, stderr



❖ Open file table •

- owned by the kernel •
- contains entries (called "file structures") for every open file •
- R/W mode, the number of processes currently using it, offset of next access, pointers to generic functions (i.e., read and write routines), inode table entry, etc. •
- transitory (during opened)

❖ Two methods of file open

1. several different processes can refer to the same open file table entry •
   - two processes : dependent

2. the same file can be opened by two separate open() statements (i.e., two separate entries) •
   - two processes : independent

❖ File allocation table (FAT) •

- owned by the kernel •
- a list(index) of the disk blocks that make up the file •
- dynamic tree - like structure, not a simple linear array

❖ inode (index node) table •

- owned by the kernel •
- when a file is opened, a copy of its inode is usually loaded into the inode table in memory for rapid access

❖ Index node (inode) •

- permanent (during the existence of files), so kept on disk with the file •
- information about file position, file size, owner's id, permissions, block count, etc.

  => once the kernel's I/O system has the inode information, it invokes an I/O processor program (≡ device driver)

❖ Device driver •

- appropriate for the type of data, the type of operation, and the type of device that is to be written •
- see that your data is moved from its buffer to its proper place on disk

3.10.2 Linking File Names to Files

- ❖ Directory •
  - a small file that contains, for each file, a file name together with a pointer (called "hard link") to the file's inode on disk •
  - file name : 14 bytes, inode number : 2 bytes
- ❖ Hard link •
  - an entry in a directory that connects a file name to the inode of the corresponding file •
  - several file names can point to the same inode •
  - when a file name is deleted, file itself is not deleted, but it s hard-link count is decremented by one
- ❖ Soft link (or symbolic link) •
  - an entry in a directory that gives the pathname of a file •
  - links a file name to another file name rather than to an actual file (i.e., an indirect pointer to a file) •
  - can refer to a directory or even to a file in a different file system •
  - UNIX 4.3 BSD (not System V)
- ❖ Shell commands •
  - ln file-name1 file-name2 (cf., link()) •
    - o two file names have the same inode number •
  - rm file-name, … •
    - o it removes directory entries & links •
  - cp file-name1 file-name2 •
    - o two file names have different inode numbers •
  - mv file-name1 file-name2 •
    - o file name is changed, but with the same inode number

3.10.3 Normal Files, Special Files, and Sockets

Three types of files

1. Normal files :• files that this text is about
2. Special files: • represent a stream of characters and control signals that drive some devices (e.g., line printer or graphic device)
3. Sockets: • abstractions that serve as endpoints for interprocess communication

=> many of the same routines can be used to access any of them (e.g., open() and write() system calls)

3.10.4 Block I/O

Three I/O systems

1. block I/O system •
    - for block-oriented device like a disk or a tape •
    - access blocks randomly •
    - sequence of bytes <=> block •
    - block size = 512 bytes(common sector size) -> 1024 bytes
2. character I/O system •
    - for character-oriented device like a keyboard or a printer •
    - read and write streams of data, not blocks

3. network I/O system

3.10.5 Device Drivers

❖ Device driver (≡ I/O processor program) •

- a separate set of routines for each peripheral device •
- performs the actual I/O between the I/O buffer and the device

❖ Block I/O device driver •

- take a block from a buffer, destined for one of physical blocks, and see that it gets deposited in the proper physical place on the device

=> Block I/O part of kernel need not know anything about the specific device it is writing to

3.10.6 The Kernel and File systems

UNIX file system ⊄ Kernel's I/O system •

- a collection of files, together with secondary information about the files in the system •
- includes the directory structure, the directories, ordinary files, and the inodes that describes the files

Separation (file system ⊂ kernel)

1. all parts of a file system •
    - reside on disk •
    - some parts are brought into memory by the kernel as needed
2. kernel •
    - reside in memory Advantages of separation •
    - we can tune a file system to a particular device or usage pattern independently of how the kernel views files •
    - we can have separate file systems that are organized differently, perhaps on different devices, but are accessible by the same kernel (e.g., file system on CD-ROM)

CHAPTER – 4: FUNDAMENTAL FILE STRUCTURE CONCEPTS

The different types of records are:

- Fixed Length Records.
- Variable Length Records.

Fixed length record: The length of all the records in the file is fixed i.e the length is predefined and altogether the length of the record will not exceed the predefined length.

Variable length record: The length of records is not fixed. Each records are of different length in the file.

4.1 Field & Record Organization

When we are building a file structures, we are making it possible to make data persistent. That is, one program can store data from memory to a file, and terminate. Later, another program can retrieve the data from the file, and process it in memory. In this chapter, we look at file structures which can be used to organize the data within the file, and at the algorithms which can be used to store and retrieve the data sequentially.

Field

A filed is the smallest logically meaningful unit of information in a file. A field is a logical notion.

4.1.1: A stream files

- 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 characters is used between lines, and a line feed (ASCII 10);
- The code in applications programs can, however, impose internal organization on stream files.

4.1.2: Field Structures:

There are many ways of adding structure to files to maintain the identity of fields. Four of the most common methods are:

- ➢ Fix the length of fields.
- ➢ Begin each field with a length indicator.
- ➢ Separate the fields with delimiters.

- Use a "Keyword = Value" Expression to identify fields.

Method 1: Fix the length of fields.

In this method length of each field in a record is fixed. For example consider the figure below:

10 bytes	8 bytes	5 Bytes	12 bytes	15 Bytes	20 Bytes
Ames	Mary	123	Maple	Stillwater	OK74075
Mason	Alan	90	Eastgate	Ada	OK74820

Here we have fixed the field 1 size as 10 bytes, field 2 as 8 bytes, field 3 as 5 bytes and so on which results in total length of record to be 70 bytes. While reading the record the first 10 bytes that is read is treated as field 1 the next 8 bytes are treated as field 2 and so on.

Disadvantage of this method is padding of each and every field to bring it to pre-defined length which makes the file much larger. Rather than using 4 bytes to store "Ames" we are using 10 bytes. We can also encounter problems with data that is too long to fit into the allocated amount of space.

Method 2: Begin each field with a length indicator.

- 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.

```
04Ames04 Mary0312305Maple10Stillwater07OK74075
05Mason04Alan02 9008Eastgate03Ada07OK74820
```

Method 3: Separate the fields with delimiters.

- 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 fields 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.
- Here the external overhead for field separation is equal to the size of the delimiter per field

- Here in the example we have used "|" as delimiter.

```
Ames|Mary|23|Maple|Stillwater|OK74075
Mason|Alan|90|Eastgate|Ada|OK74820
```

Method 4: Use a "Keyword = Value" Expression to identify fields.

- It is the structure in which a field provides information about itself (Self describing).
- It is easy to tell which fields are contained in a file.
- It is also good format for dealing with missing fields. If a field is missing, this format makes it obvious, because the keyword is simply not there.
- This format wastes lot of space for storing keywords.

```
Lname=Ames | Fname=Mary | Address=123 Maple | City= Stillwater | State = OK | Zip=74075
Lname=Mason | Fname=Alan | Address=90 Eastgate | City = Ada | State = OK | Zip = 74820
```

4.1.3: Reading a stream of fields

4.1.4: Record Structures:

A record can be defined as a set of fields that belong together when the file is viewed in terms of a higher level of organization.

Following are some of the most often used methods for organizing the records of a file.

- ➢ Make records a predictable number of bytes.
- ➢ Make records a predictable number of fields.
- ➢ Begin each record with a length indicator.
- ➢ Use an index to keep track of the address.
- ➢ Place a delimiter at the end of each record.

Method 1: Make records a predictable number of bytes.

- All records within a file have the same size.
- 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.
- There are three ways of achieving this method. They are as shown below.

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

Fixed Length records with fixed Length fields

```
Ames|Mary|23|Maple|Stillwater|OK74075| ←——Unused space ——→
Mason|Alan|90|Eastgate|Ada|OK74820| ←—— Unused space ——→
```

Fixed length records with variable length fields

Method 2: Make record a predictable number of fields.

- All the records in the a file contains fixed number of fields.
- In the figure below each record is of 6 fields.

```
Ames|Mary|23|Maple|Stillwater|OK74075| Mason|Alan|90|Eastgate|Ada|OK74820| …
```

6 Fields 6 Fields

Method 3: Begin each record with a length indicator

- 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.

```
40Ames|Mary|23|Maple|Stillwater|OK74075|36 Mason|Alan|90|Eastgate|Ada|OK74820| …
↑                                       ↑
```

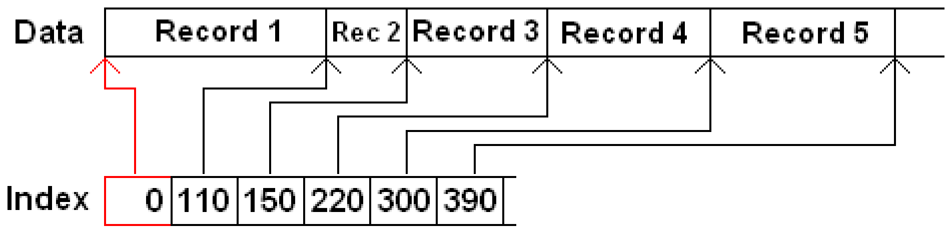
Length of 1st record (Length Indicator)

Length of 2nd record (Length Indicator)

Method 4: Use an index to keep track of the address.

- An auxiliary file can be used to point to the beginning of each record.
- In this case, the data records can be contiguous.
- Disadvantage: there is space overhead for the index file.
- Disadvantage: there is time overhead for the index file.
- The time overhead for accessing the index file can be minimized by reading the entire index file into memory when the files are opened.

- Advantage: there will probably be no internal fragmentation.
- 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.



Method 5: Place a delimiter at the end of each record.

- Records are terminated by a special character or sequence of characters called delimiter.
- Programs which access the record must know the delimiter.
- The delimiter cannot be part of data.
- If used with delimited fields, the field delimiter must be different from the record delimiter.
- Here the external overhead for field separation is equal to the size of the delimiter per record.
- In the following figure we use "|" as field delimiter and "#" as record delimiter.

Ames|Mary|23|Maple|Stillwater|OK74075| #Mason|Alan|90|Eastgate|Ada|OK74820|# …

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.

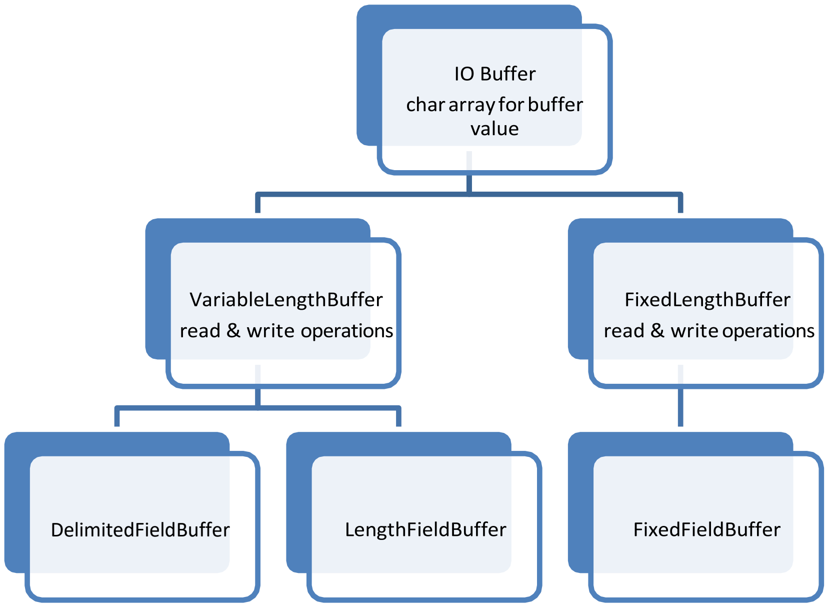
4.3: Using Inheritance for record buffer classes

4.3.1: Inheritance in the C++ stream class

One or more base classes define members and methods, which are then used by subclasses. Our discussion in this section deals with fstream class which is embedded in a class hierarchy that contains many other classes. The read operations, including the extraction operators are defined in class istream. The write operations are defined in class ostream. Class iostream, inherits from istream and ostream.

4.3.2: A Class Hierarchy for Record Buffer Objects

The members and methods that are common to all of the three buffer classes are included in base



class IOBuffer. Example the members in IOBuffer class like BufferSize, MaxBytes, NextByte are inherited in its derived class VariableLengthBuffer and FixedLengthBuffer. Similarly methods read(), Write(), Pack(), Unpak() of IOBuffer are inherited by derived classes Variable Length Buffer and Fixed Length Buffer. (Refer Text book for class definition).

Here the member functions read(), Write(), Pack(), Unpack() of class IO Buffer are virtual functions so that subclasses Variable Length Buffer and Fixed Length Buffer define its own implementation. This means that the class IO Buffer does not include an implementation of the method. The reading and writing of variable length records are included in the class Variable Length Buffer. Packing and Unpacking delimited fields is defined in the class Delimited Field Buffer which is derived from Variable Length Buffer.

4.4: Managing Fixed-Length, Fixed-Field Buffers

- Class Fixed Length Buffer is the subclass of IO Buffer that supports read and write of fixed-length records.
- Since each record is of fixed size, it is not required to specify record size explicitly.

4.5: An object-oriented class for record files

The class BufferFile encapsulates the file operations of open, create, close, read, write, and seek in a single object. Each BufferFile object is attached to a buffer. The read and write operations move data between file and buffer. The use of BufferFile adds a level of protection to our file operations. Once a disk file is connected to a BufferFile object, it can be manipulated only with the related buffer.

CHAPTER – 5: ***Managing Files of Records***

5.1: Record Access

5.1.1: Record keys

- When looking for an individual record, it is convenient to identify the record with a key based on record contents.
- The key should be unique so that duplicate entries can be avoided.

For example, in the previous section example we might want to access the "Ames record" or the "Mason record" rather than thinking in terms of the "first record" or "second record".

- When we are looking for a record containing the last name Ames we want to recognize it even if the user enters the key in the form "AMES", "ames" or "Ames". To do this we must define a standard form of keys along with associated rules and procedures for converting keys into this standard form. This is called as **canonical form of the key.**

5.1.2: Sequential Search

Reading through the file, record by record, looking for a record with a particular key is called **sequential searching**. In general the work required to search sequentially for a record in a file with 'n' records in proportional to n: i.e the sequential search is said to be of the order O(n).

This efficiency is tolerable if the searching is done on the date present in the main memory, but not for, which has to be extracted from secondary storage device, due to high delay involved in accessing the data. Instead of extracting the records from the secondary storage device one at a time sequential we can access some set of records at once from the hard disk, store it in main memory and do the comparisons. This is called as **record blocking.**

In some cases sequential search is superior like:

Repetitive hits: Searching for patterns in ASCII files.

Searching records with a certain secondary key value.

Small Search Set: Processing files with few records.

Devices/media most hospitable to sequential access: tape, binary file on disk.

5.1.3:Unix Tools For Sequential Processing

Some of the UNIX commands which perform sequential access are:

Cat: displays the content of the file sequentially on the console.

%cat filename Example:

```
%cat myfile
Ames        Mary    123     Maple       Stillwater      OK74075
Mason       Alan     90     Eastgate    Ada             OK74820
```

wc: counts the number of lines, words, and characters in the file

%wc filename Example:

```
%wc  myfile  2   14  76
```

grep: (generalized regular expression) Used for pattern matching

%grep string filename Example:

```
% grep Ada myfile
Mason       Alan     90     Eastgate    Ada    OK74820
```

5.1.4: Direct Access:

- The most radical alternative to searching sequentially through a file for a record is a retrieval mechanism known as direct access.
- The major problem with direct access is knowing where the beginning of the required record is.
  - ❖ One way to know the beginning of the required record or byte offset of the required record is maintaining a separate index file.
  - ❖ The other way is by relative record number **RRN**. If a file is a sequence of records, the RRN of a record gives its position relative to the beginning of the file. The first record in a file has RRN 0, the next has RRN 1, and so forth.
- To support direct access by RRN, we need to work with records of fixed length. If the records are all the same length, we can use a record's RRN to calculate the byte offset of the start of the record relative to the start of the file.

For example: if we are interested in the record with an RRN of 546 and our file has fixed length record size of 128 bytes per record, we can calculate the byte offset of a record with an RRN of n is

Byte offset = 546 * 128 = 69888

In general **Byte offset = n * r** where r is length of record and n is the record number.

- C++ can use seekg and seekp methods to jump to the byte that begins the record.

5.2 More about Record Structures

5.2.1Choosing a Record Structure and Record Length

- The simplest record formats for direct access by RRN is the use of fixed length records.
- Fixed length records can provide good performance and good space utilization when the data vcomes in fixed-size quantities (Ex: pin codes, Date of Birth, etc..)
- Two general approaches for organizing fields within a fixed-length record are:

(i) Use of fixed Length fields within fixed length record

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

(ii) Use of variable length fields within fixed length record

```
Ames|Mary|23|Maple|Stillwater|OK74075| ←——Unused space ——→
Mason|Alan|90|Eastgate|Ada|OK74820| ←—— Unused space ——→
```

5.2.2 Header Records

Header Records

It is often necessary or useful to keep track of some general information about a file to assist in future use of the file. A header record is often placed at the beginning of the file to hold information such as number of records, type of records the file contains, size of the file, date and time of file creation, modification etc. Header records make a file self describing object, freeing the software that accesses the file from having to know a priori everything about its structure. The header record usually has a different structure than the data record.

self describing file

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

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.

```
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.

5.3: Encapsulating record I/O operations in a single class:

- For making objects persistent, a good object oriented design should provide operations to read and write objects directly.
- The write operations include, pack into a buffer and write the buffer to a file.
- In class RecordFile, read operations takes an object of some class and writes it to a file.

  For this C++ templates are used.

5.4: File Access and File Organization

- File organization is static.
  - Fixed Length Records.
  - Variable Length Records.
- File access is dynamic.
  - Sequential Access
  - Direct Access.

****END OF MODULE - 1****

MODULE - 2

CHAPTER – 6: ORGANIZATION OF FILES FOR PERFORMANCE

Introduction

- 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.

6.1 Data Compression

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.
- 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 Using a different Notation (Compact Notation)

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.

6.1.2 Suppressing Repeating Sequences

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.

6.1.3 Assigning Variable Length Codes

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:	a4	e1	b1	f1	c1	g1	d1

Code: a 1 e 0001 b 010 f 0010 c 011 g 0011 d 0000

Compressed Text (binary):

10100000000110110010110011 (26 bits)

Compressed Text (hexadecimal):

A0 1B 96 60

6.1.4 Irreversible Compression Techniques

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.

6.1.5 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 co mpressed 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.

6.2 Reclaiming Space in Files

6.2.1 Record Deletion and Storage Compaction

External fragme ntation: 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.

```
ABC|123 Mysore…….
DCV|333 Bangalore
…
XXX|36 Mysore…….
```

Fig 1: Before the second record is marked as deleted

```
ABC|123 Mysore…….
*CV|333 Bangalore …
XXX|36 Mysore…….
```

Fig 2: After the second record is marked as deleted

```
ABC|123 Mysore…….
XXX|36 Mysore…….
```

Fig 3: storage compaction

6.2.2 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.

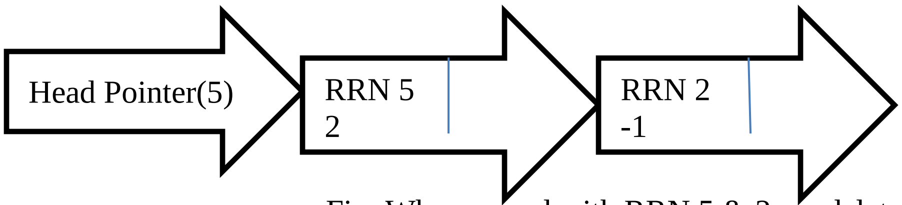


Fig: When record with RRN 5 & 2 are deleted.

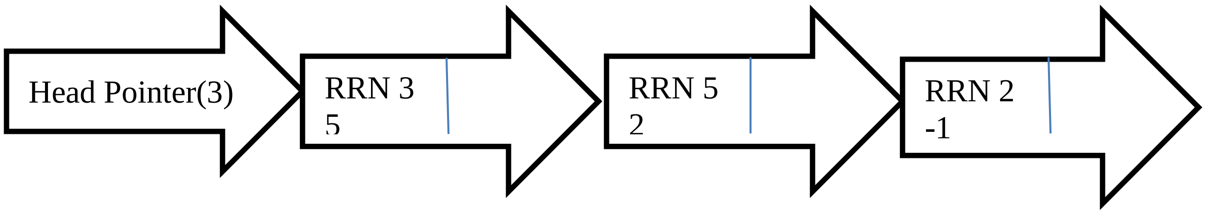


Fig: When record with RRN 3 is also deleted.

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 interchangeable; 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	New Record	Record 6

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

6.2.3 Deleting Variable-Length Records

- If the file uses variable length records, the spaces not are interchangeable; a new record may not fit just in 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.

HEAD.FIRST_AVAIL: -1

18 Ames|Mary|123 USA|17 James|John|75 UK|24 Folk|
Michael|150 London

Fig: Original sample file stored in variable length format with byte count.

HEAD.FIRST_AVAIL: 21

18 Ames|Mary|123 USA|17 *|-1-----------------|24 Folk|
Michael|150 London

Fig: sample file after deletion of second record

Adding and removing records:

- Here, we cannot access the avail list as a stack since the avail list differ in size. We search through the avail list for a record slot that is the right size. ("big enough").
- Fig shows removal of a record from avail list.

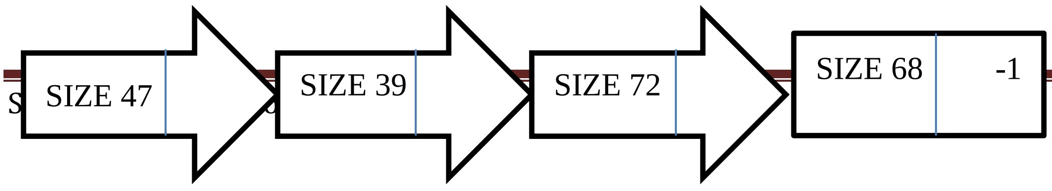
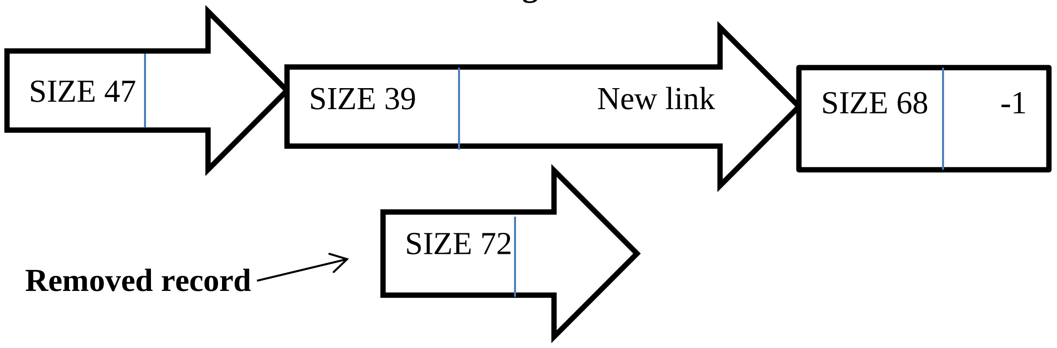


Fig : Before deletion



6.2.4 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 records.

External fragmentation: Fragmentation in which the unused space is outside or between individual records.

There are three ways to deal with external fragmentation

- Storage compaction
- Coalesing the holes
- Use a clever placement strategy

6.2.5 Placement Strategies

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 old 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	*50| 30	New 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	* 200| 60	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 the 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**	**New Record**	**Record**	*** 200 |60**	**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	* 370| 35	New 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.

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

- The cost of seeking is very high.
- This cost has to be taken into consideration when determining a strategy for searching (also for sorting) a file for particular piece of information.

6.3.1 Finding things in simple field and Record file

Here, we are interested to retrieve records based on their key value.

6.3.2 Search by Guessing: 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:

  ABCDEFGHIJKLMNOPQRS
- Compare 'M' to the middle in the list:

  ABCDEFGHI**J**KLMNOPQRS
- 'M' > 'J': Narrow the search to the last half. (Eliminate the first half.)

  ABCDEFGHIJ**KLMNOPQRS**
- Compare 'M' to the middle li in the remainder of the list:

  ABCDEFGHIJ**KLMNOPQRS**
- 'M' < 'O': Narrow the search to the first half of the remainder. (Eliminate the last half.)

  ABCDEFGHIJ**KLMN**OPQRS
- Compare 'M' to the middle li in the remainder of the list:

ABCDEFGHIJ**KLMN**OPQRS

- 'M' > 'L': Narrow the search to the last half. (Eliminate the first half.)

  ABCDEFGHIJKL**MN**OPQRS

- Compare 'M' to the middle li in the remainder of the list:

  ABCDEFGHIJKL**MN**OPQRS

  'M' == 'M': The search is over.

6.3.3 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.

6.3.4 Sorting a Disk File in Memory

Internal sort: A sort performed entirely in main memory.

- The algorithms used for internal sort 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.

6.3.5 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.

6.4 Keysorting

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

6.4.1 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 (*KE YNODES*).

- Washington George 1
- Adams John 2
- Jefferson Thomas 3
- Madison James 4
- Monroe James 5
- Adams John 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
- 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.

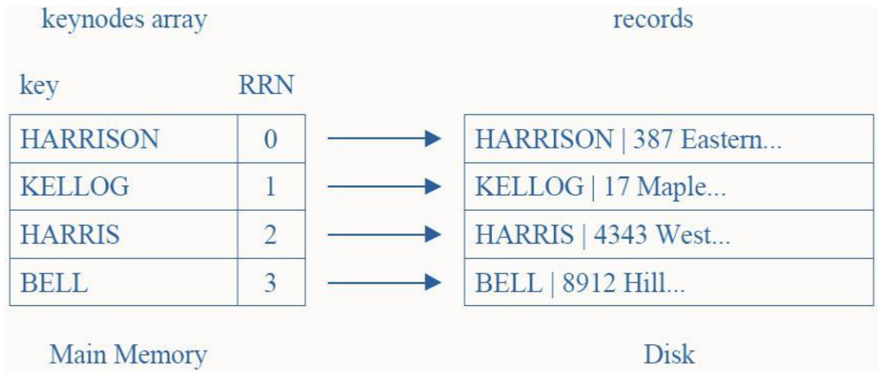


Fig: Before sorting keys

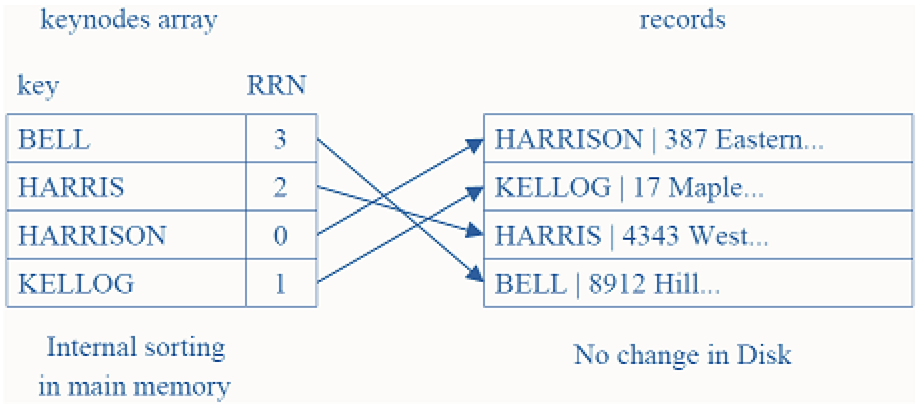


Fig: After sorting keys

6.4.2 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.

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

- Rather than actually sorting the data file, the *KE YNODES* array can be written to a disk file (sequentially), and will function as an index.

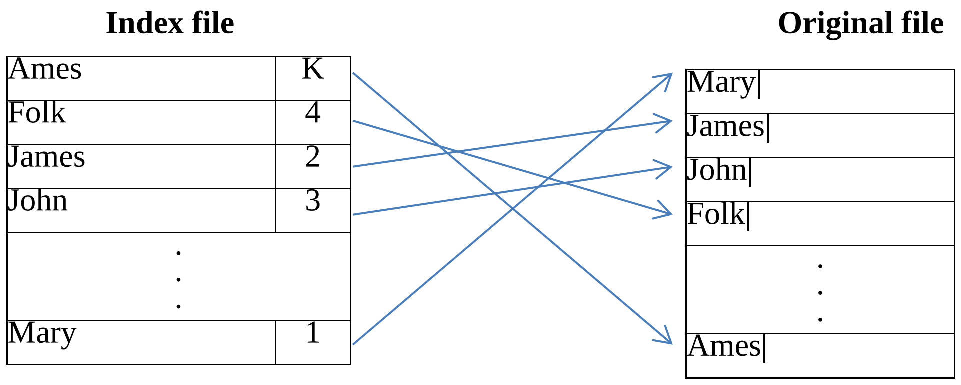


Fig: Relationship between index file and data file

6.4.4 Pinned Records

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

►Remember that in order to support deletions we used **AVAIL LIST**, a list of available records

►The **AVAIL LIST** contains info on the physical information of records. In such a file, a record is said to be **pinned**

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

CHAPTER – 7: 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

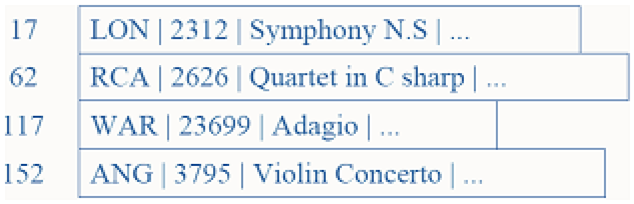
simple index

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.

Records (Variable-length)

Actual data record



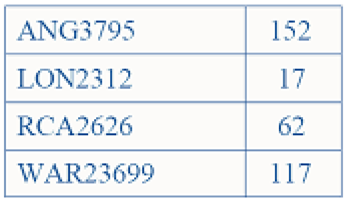
17	LON | 2312 | Symphony N.S | ...
62	RCA | 2626 | Quartet in C sharp | ...
117	WAR | 23699 | Adagio | ...
152	ANG | 3795 | Violin Concerto | ...

address of record

Primary key = company label + record ID

Index:

Key reference field



ANG3795	152
LON2312	17
RCA2626	62
WAR23699	117

7.3 Us in g Te m pl at e Cl

asses in C++ for object I/O:

It is the C++ template feature that supports parameterized function and class definitions, and RecordFile is a template class.

7.4 Object Oriented Support for Indexed, Entry Sequenced Files entry-sequenced file of data objects

7.4.1 Operations required to Maintain an Indexed file

1) Create the original empty index and data files

- A data file to hold the data objects and
- An index file to hold the primary key index.

2) Load index file into memory before using it

- Index is represented as array of records
- The loading into memory can be done sequentially, reading a large number of index records (which are short at once).

3) Rewrite the index file from memory

- When close function is executed, the index file is written back to the disk from memory.

4) Record addition

- Adding a new record to data file requires that we also add an entry to the index file.
- Since the index is kept in sorted order by key, insertion of the new index entry requires some rearrangement of the index.

5) Record deletion

- When we delete a record from the data file, we must also delete the corresponding entry from the index file.
- Deleting the index entry, requires shifting the other entries to remove the empty spaces.

6) Record Updation

Record updating falls into two categories:

1. <u>The update changes the value of the key field:</u> Here, both index and data file may need to be reordered.
2. <u>The update does not affect the key field:</u> Does not require rearrangement of the index file but may well involve in reordering of data file.

7.4.2 Class TextIndexedFile:

It supports files of data objects with primary keys that are strings. There are methods: Create, Open, Close, Read, Append, and Update. Example: fig 7.7.

7.4.3 Enhancements to TextIndexedFile:

Even though class TextIndexedFile is parametrized to support a variety of data object classes, it restricts the key type to string (char *).

7.5 Indexes that are too Large to Hold in Memory

If the index is too large to hold in memory, we come across some disadvantages.

- Binary searching of the index requires several seeks instead of taking place at memory speed'
- Index rearrangement due to record addition or deletion requires shifting or sorting records on secondary storage, which is extremely time consuming.

The solution is using

- ➢ Hashed organization if access speed is of top priority
- ➢ Tree structured, or multilevel index such as a B-tree if we need rhe flexibility of both random access and sequential access.

Advantages of simple indexes on secondary over the use of data file sorted by key are:

- ➢ A simple index allow use of binary search in a variable-length record file.
- ➢ If the index entries are substantially smaller than the data file records, sorting and maintaining the index can be less expensive than the data file.
- ➢ If there are pinned records in the data file, the use of an index lets us rearrange the keys without moving the data records.
- ➢ Provides multiple views of a data file.

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.

Operations on secondary indexes:

1) **<u>Record addition:</u>** When the secondary index is used, adding a record involves updating the data file, the primary index, and the secondary index. Secondary keys are sorted in canonical form.
2) **<u>Record deletion:</u>** Removing a record from a data file means removing the corresponding entry in primary index and all the entries in secondary indexes that reference to this. The problem here is, like the primary index, the secondary index must also be sorted.
3) **<u>Record updating:</u>** There are three possible situations;

   (i) Update changes the secondary key:

   We may have to rearrange the secondary key index so it stays in sorted order.(Relatively expensive operation)

   (ii) Update changes the primary key:

   Has large impact (or changes) on primary key index but often requires that we update only the affected reference field in all secondary index

(iii) Update confined to other fields:

No changes necessary to neither primary nor secondary indexes.

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

- Secondary index structures results in two distinct difficulties;
  - We have to rearrange the index file every time a new record is added to the file, even if the new record is for an existing secondary key.
  - If there are duplicate secondary keys, the secondary key field is **repeated** for each entry.
- There are two solutions for this.

Solution 1:

- Change the secondary index structure so that it associates an array of references with each secondary key.

Example: BEETHOVEN ANG3736 DG13902 DG18806 RCF2524

- Fig below shows secondary key index containing space for multiple references for each secondary key.

Secondary key	**Set of primary key references**			
BEETHOVEN	ANG3736	DG13902	DG18806	RCF2524
NEIL	LON3221			
JOHN	DSE2424			
FOLK	WAR23699			
MARY	ANG0102			

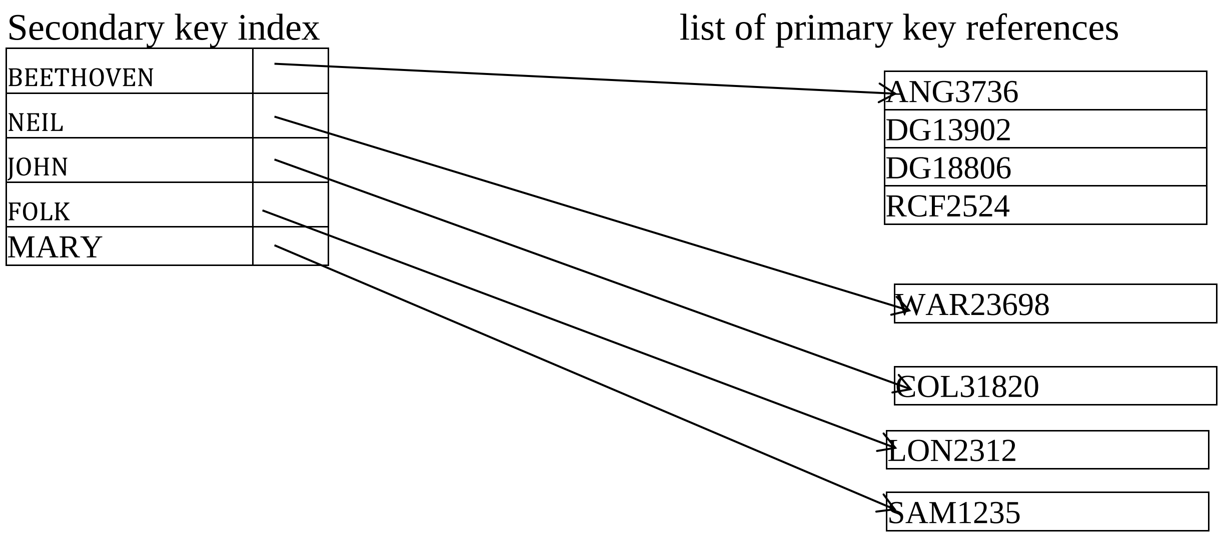
Advantage: Avoids rearrange of file.

Disadvantage: May restrict the number of references that can be associated with each secondary key.

Solution 2: Linking the list of references (Better solution)

Inverted list

- Files such as secondary indexes, in which a secondary key leads to a set of one or more primary keys, are called **inverted lists**.
- **Method**: Each secondary key points to a different list of primary key references. Each of these lists could grow to be as long as it needs to be and no space would be lost to internal fragmentation.
- Fig shows conceptual view of primary key reference lists as a series of lists.



Advantages:

- Secondary index file needs to be rearranged only when new record is added (i.e., when new composer's name is added or existing composer's name is changed.
- Rearranging is faster, since there are fewer records and each record is smaller.
- There is less need for sorting. Therefore we can keep secondary index file on disk.
- Label ID list file is entry sequenced. i.e., primary index never needs to be sorted.
- Space from deleted primary index records can easily be reused.

Disadvantage:

- Label IDs associated with a given composer are no longer guaranteed to be grouped together physically. i.e., locality(togetherness) in the secondary index has been lost.

7.9 Selective Indexes

selective index

An index which contains keys for only part of the records in a data file. Such an index provides the user with a view of a specific subset of the file’s records.

7.10 Binding

binding

The association of a symbol with a value.

- ➢ So far, the binding of our primary keys takes place at construction time.

  Advantage: Faster access

  Disadvantage: Reorganisation of data file must result in modifications to all bound index files.

- ➢ Binding of our secondary keys takes place at the time they are used**.**

  Advantage: Safer

Tradeoff in binding decisions:

- ❖ **Tight binding** (construction time binding i.e., during preparation of data files) is preferable when
  - Data file is static or nearly static, requiring little or no adding, deleting, or updating.
  - Rapid performance during actual retrieval is a high priority.
- ❖ **Postponing binding** as long as possible is simpler and safer when the data file requires a lot of adding, deleting and updating.

Note: Here, the connection between the key and the particular physical record is postponed until the record is retrieved in the course of program execution.

************* End of Module – 2 *************

Module 3
COSEQUENTIAL PROCESSING AND THE SORTING OF LARGE FILES

DEFINITION:

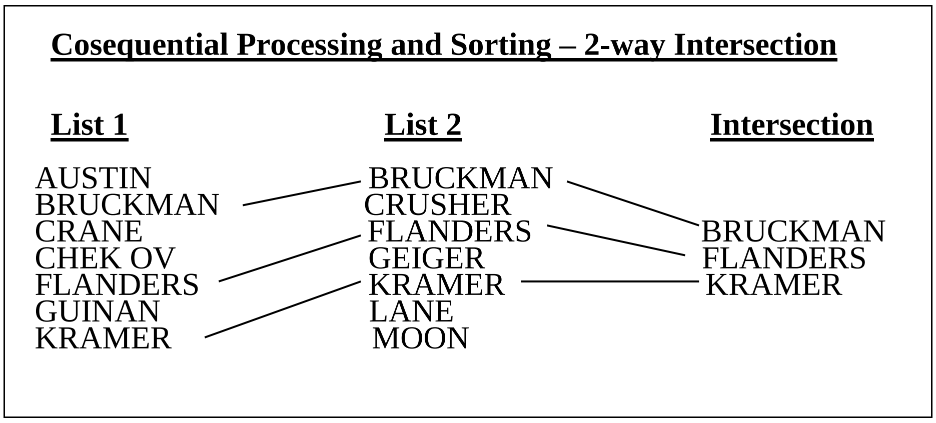
- ***Cosequential operations*** involve the coordinated processing of two or more sequential lists to produce a single output list.
- This is useful for ***merging*** (or taking the ***union***) of the items on the two lists and for ***matching*** (or taking the ***intersection***) of the two lists.
- These kinds of operations are extremely useful in file processing.

8.1 AN OBJECT-ORIENTED MODEL FOR IMPLEMENTING COSEQUENTIAL PROCESSES

We present a Single, Simple model that can be the basis for the construction of any kind of cosequential process.

8.1.1 Matching Names in Two Lists

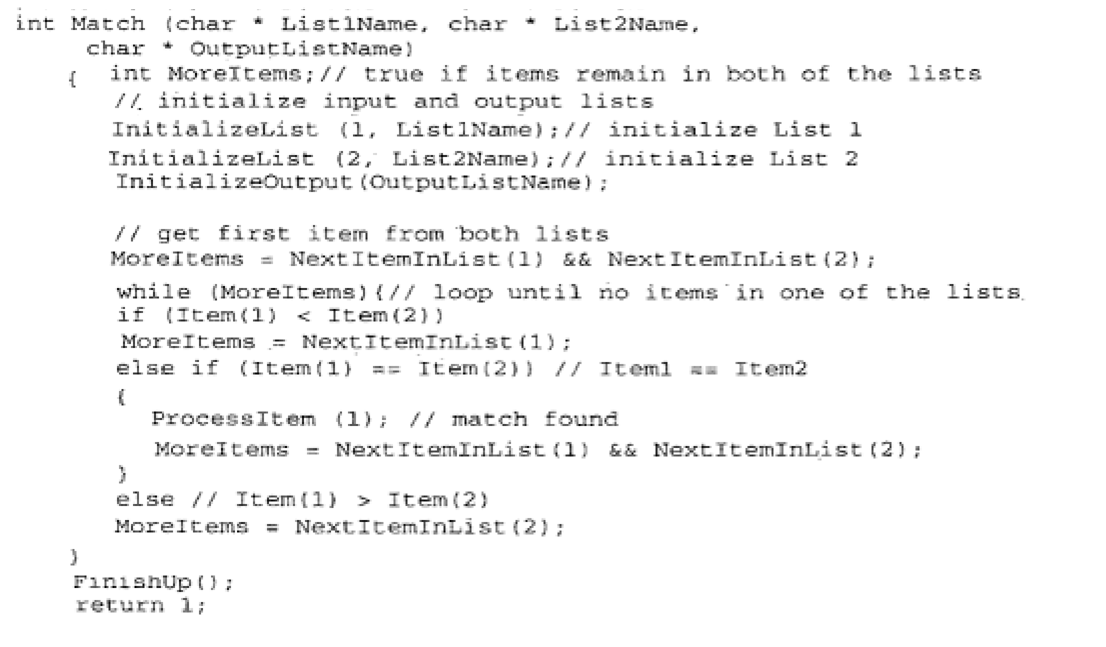
Suppose we want to output the names common to the two lists Shown in Fig. 8.1. This operation is usually called a ***match operation,* or an *intersection***. We assume, for the moment, that we will not allow duplicate names within a list and that the lists are sorted in ascending order.



At each step in the processing of the two lists, we can assume that we have two items to compare: a current item from List 1 and a current item from List 2. Let's call these two current items Item (l) and Item (2). We can compare the two items to determine whether Item (i) is less than, equal to, or greater than Item (2):

- If Item (1) < Item (2), get the next item from List 1;
- If Item (1) > Item (2), get the next item from List 2; and
- If Item (1) = Item (2), output the item and get the next items from the two lists.

It turns out that this can be handled very cleanly with a single loop containing one three-way conditional statement, as illustrated in the algorithm of Fig. 8.2. (Also can refer program 7)



```
int Match (char * List1Name, char * List2Name,
     char * OutputListName)
   {  int MoreItems;// true if items remain in both of the lists
      // initialize input and output lists
      InitializeList (1, List1Name);// initialize List 1
      InitializeList (2, List2Name);// initialize List 2
      InitializeOutput(OutputListName);

      // get first item from both lists
      MoreItems = NextItemInList(1) && NextItemInList(2);
      while (MoreItems){// loop until no items in one of the lists
      if (Item(1) < Item(2))
      MoreItems = NextItemInList(1);
      else if (Item(1) == Item(2)) // Item1 == Item2
      {
         ProcessItem (1); // match found
         MoreItems = NextItemInList(1) && NextItemInList(2);
      }
      else // Item(1) > Item(2)
      MoreItems = NextItemInList(2);
   }
   FinishUp();
   return 1;
```

Figure 8.2 Cosequential match function based on a single loop.

Although the match procedure appears to be quite simple, there are a number of matters that have to be dealt with to make it work reasonably well.

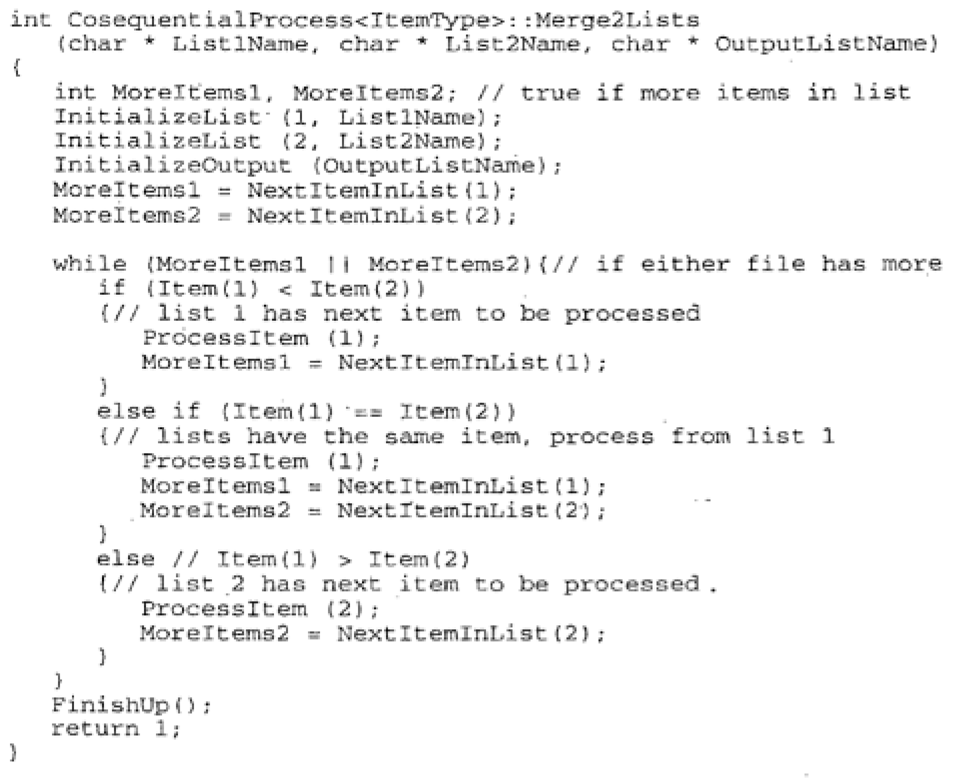
Considerations for Cosequential Algorithms

- **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 Merging Two Lists

The three-way-test, single- loop model for cosequential processing can easily be modified to handle *merging* of lists simply by producing output for every case of the *if-then-else* construction since a merge is a *union* of the list contents. Method Merge2Lists is given in Fig. 8.5. Once again, you should use this logic to work, step by step, through the lists provided in

Fig. 8.1 to see how the resynchronization is handled and how the use of the HighValues forces the procedure to finish both lists before terminating.



```
int CosequentialProcess<ItemType>::Merge2Lists
   (char * List1Name, char * List2Name, char * OutputListName)
{
   int MoreItems1, MoreItems2; // true if more items in list
   InitializeList (1, List1Name);
   InitializeList (2, List2Name);
   InitializeOutput (OutputListName);
   MoreItems1 = NextItemInList(1);
   MoreItems2 = NextItemInList(2);

   while (MoreItems1 || MoreItems2){// if either file has more
      if (Item(1) < Item(2))
      {// list 1 has next item to be processed
         ProcessItem (1);
         MoreItems1 = NextItemInList(1);
      }
      else if (Item(1) == Item(2))
      {// lists have the same item, process from list 1
         ProcessItem (1);
         MoreItems1 = NextItemInList(1);
         MoreItems2 = NextItemInList(2);
      }
      else // Item(1) > Item(2)
      {// list 2 has next item to be processed .
         ProcessItem (2);
         MoreItems2 = NextItemInList(2);
      }
   }
   FinishUp();
   return 1;
}
```

Figure 8.5 Cosequential merge procedure based on a single loop.

List 1	**List 2**	**Union**
AUSTIN	BRUCKMAN	AUSTIN
BRUCKMAN	CRUSHER	BRUCKMAN
CRANE	FLANDERS	CHEK OV
CHEK OV	GEIGER	CRANE
FLANDERS	KRAMER	CRUSHER
GUINAN	LANE	FLANDERS
KRAMER	MOON	GEIGER
		GUINAN
		KRAMER
		LANE
		MOON

8.1.3 Summary of the Cosequential Processing Model

It is important to understand that the model makes certain general assumptions about the nature of the data and type of problem to be solved. For the list of the assumptions, together with clarifying comments refer text.

Given these assumptions, the essential components of the model are:

- Initialization. Previous item values for all files are Set to the low value; then current records for all files are read from the first logical records in the respective files.

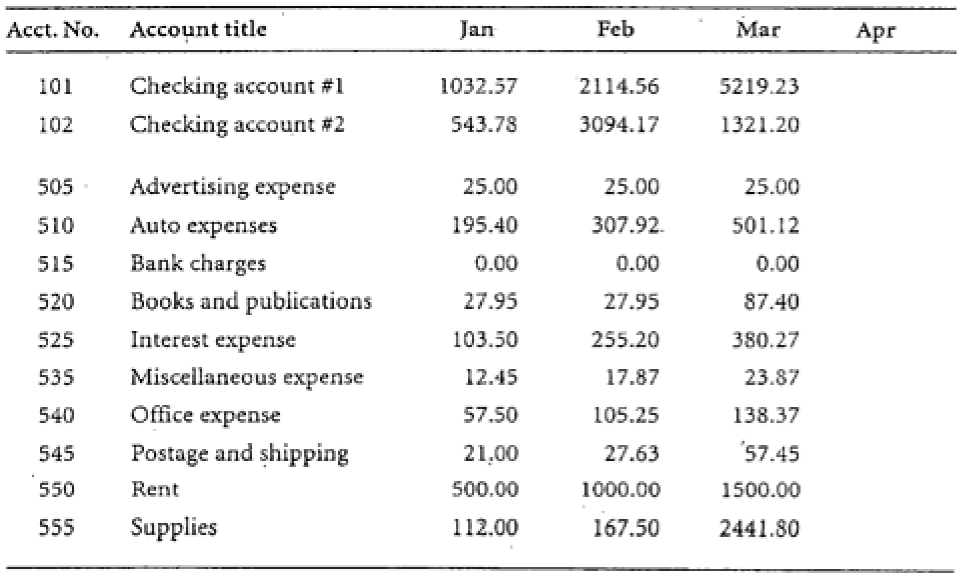
- One main Synchronization loop is used, and the loop continues as long as relevant records remain.
- Within the body of the main synchronization loop is a selection based on comparison of the record keys from respective input file records. If there are two input files, the selection takes the form given in function Match of Fig. 8.2.
- Input files and output files are sequence checked by comparing the previous item value with the new item value when a record is read. After a successful sequence check, the previous item value is set to the new item value to prepare for the next input operation on the corresponding file.
- High values are substituted for actual key values when end-of- file occurs. The main processing loop terminates when high values have occurred for all relevant input files. The use of high values eliminates the need to add special code to deal with each end-of- file condition.
- All possible I/O and error detection activities are to be relegated to supporting methods so the details of these activities do not obscure the principal processing logic.

8.2 APPLICATION OF THE MODEL TO A GENERAL LEDGER PROGRAM

8.2.1 The Problem

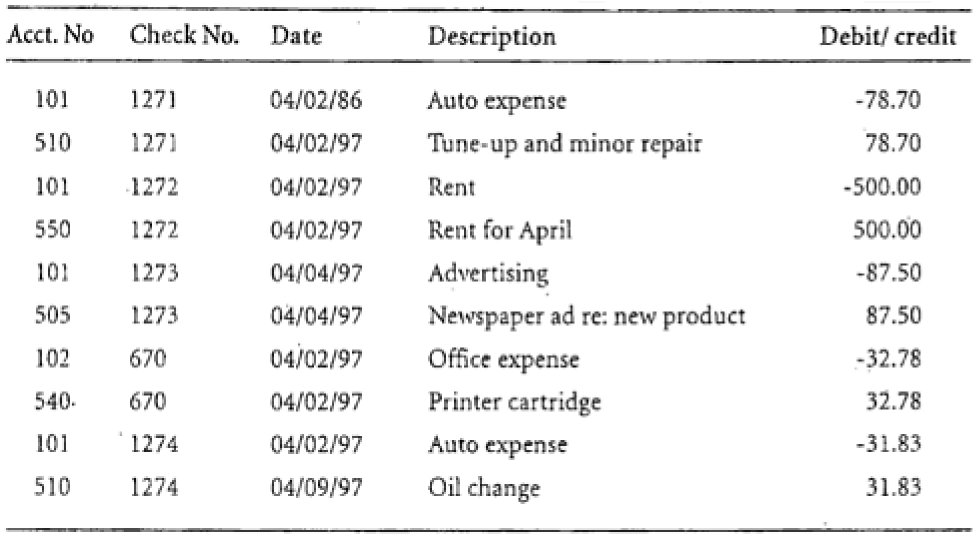
Suppose we are given the problem of designing a general ledger posting program as part of an accounting system. The systern includes a **journal file and a ledger file**. The **ledger** contains month-by- month summaries of the values associated with each of the bookkeeping accounts. A sample portion of the ledger, containing only checking and expense accounts, is illustrated in Fig. 8.6.

The **journal file** contains the monthly transactions that are ultimately to be posted to the ledger file. Figure 8.7 shows these journal transactions.



Acct. No.	Account title	Jan	Feb	Mar	Apr
101	Checking account #1	1032.57	2114.56	5219.23	
102	Checking account #2	543.78	3094.17	1321.20	
505	Advertising expense	25.00	25.00	25.00	
510	Auto expenses	195.40	307.92	501.12	
515	Bank charges	0.00	0.00	0.00	
520	Books and publications	27.95	27.95	87.40	
525	Interest expense	103.50	255.20	380.27	
535	Miscellaneous expense	12.45	17.87	23.87	
540	Office expense	57.50	105.25	138.37	
545	Postage and shipping	21.00	27.63	57.45	
550	Rent	500.00	1000.00	1500.00	
555	Supplies	112.00	167.50	2441.80	

Figure 8.6 Sample ledger fragment containing checking and expense accounts.



Acct. No	Check No.	Date	Description	Debit/ credit
101	1271	04/02/86	Auto expense	-78.70
510	1271	04/02/97	Tune-up and minor repair	78.70
101	1272	04/02/97	Rent	-500.00
550	1272	04/02/97	Rent for April	500.00
101	1273	04/04/97	Advertising	-87.50
505	1273	04/04/97	Newspaper ad re: new product	87.50
102	670	04/02/97	Office expense	-32.78
540	670	04/02/97	Printer cartridge	32.78
101	1274	04/02/97	Auto expense	-31.83
510	1274	04/09/97	Oil change	31.83

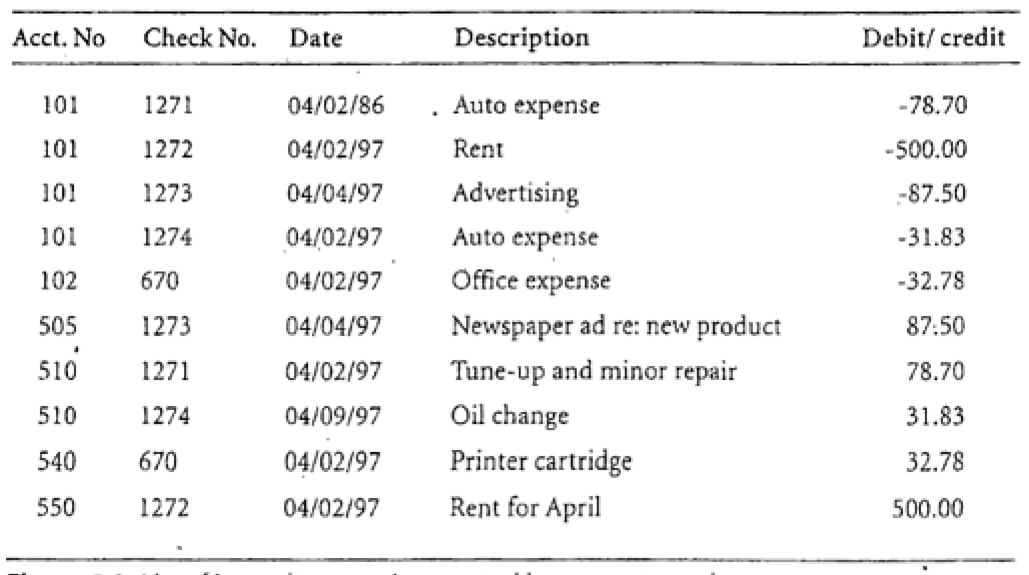
Figure 8.7 Sample journal entries.

Once the journal file is complete for a given month, meaning that it contains all of the transactions for that month, the journal must be posted to the ledger. ***Posting*** involves associating each transaction with its account in the ledger.

How is the posting process implemented?

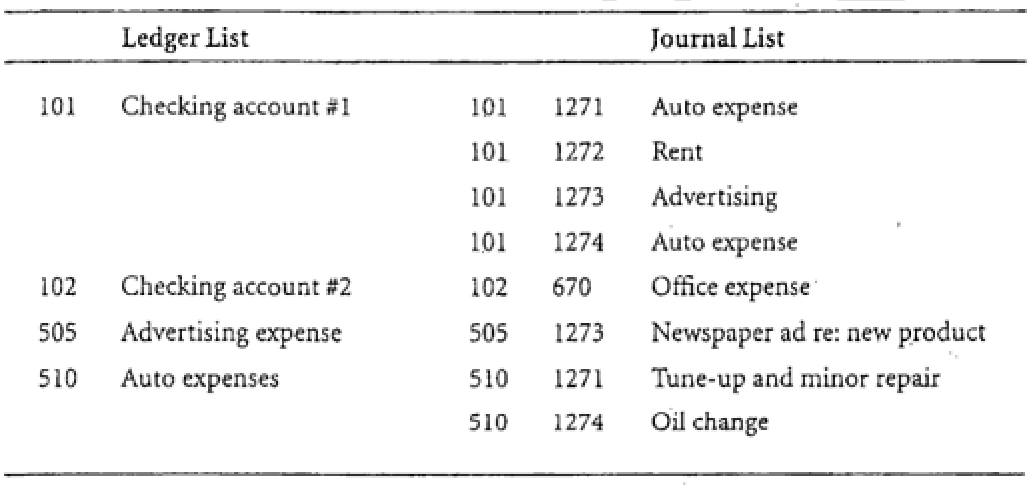
Clearly, it uses the account number as a *key* to relate the Journal transactions to the ledger records.

A better solution is to begin by collecting all the journal transactions that relate to a given account by sorting the journal transactions by account number working through both the ledger and the sorted journal *cosequentially*, meaning that we process the two lists sequentially and in parallel, producing a list ordered as in Fig. 8.9. This concept is illustrated in Fig. 8.10.



Acct. No	Check No.	Date	Description	Debit/ credit
101	1271	04/02/86	Auto expense	-78.70
101	1272	04/02/97	Rent	-500.00
101	1273	04/04/97	Advertising	-87.50
101	1274	04/02/97	Auto expense	-31.83
102	670	04/02/97	Office expense	-32.78
505	1273	04/04/97	Newspaper ad re: new product	87.50
510	1271	04/02/97	Tune-up and minor repair	78.70
510	1274	04/09/97	Oil change	31.83
540	670	04/02/97	Printer cartridge	32.78
550	1272	04/02/97	Rent for April	500.00

Figure 8.9 List of journal transactions sorted by account number.



	Ledger List	Journal List		
101	Checking account #1	101	1271	Auto expense
		101	1272	Rent
		101	1273	Advertising
		101	1274	Auto expense
102	Checking account #2	102	670	Office expense
505	Advertising expense	505	1273	Newspaper ad re: new product
510	Auto expenses	510	1271	Tune-up and minor repair
		510	1274	Oil change

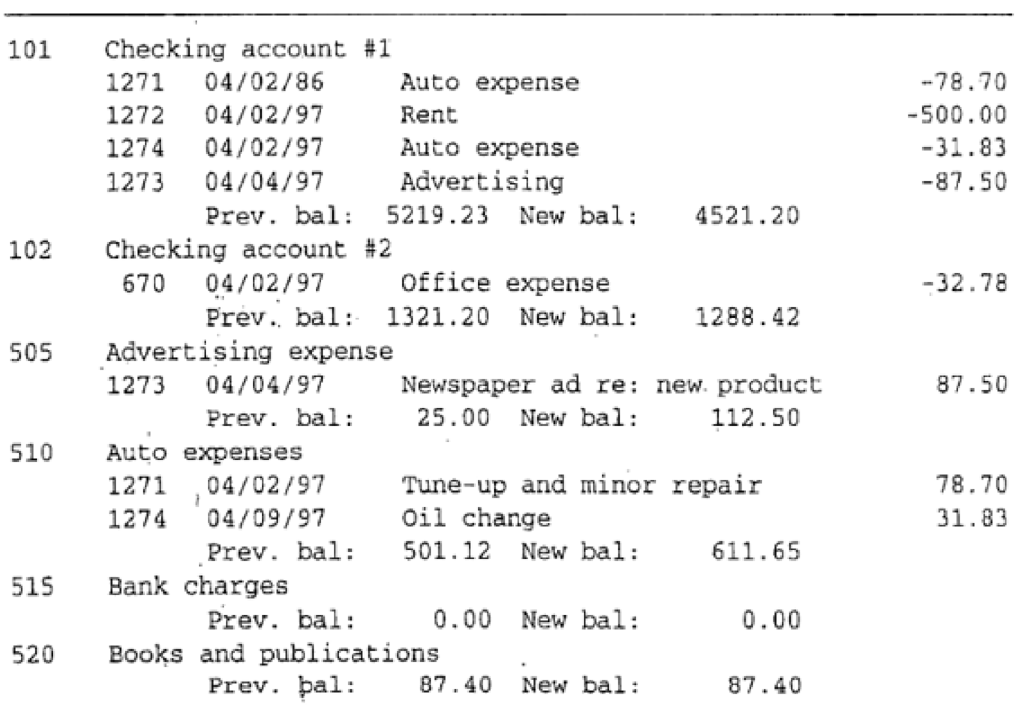
Figure 8.10 Conceptual view of cosequential matching of the ledger and journal files.

8.2.2 Application of the Model to the Ledger Program

The monthly ledger posting program must perform two tasks:

- o It needs to update the ledger file with the, correct balance for each account for the current month.
- π It must produce a printed version of the ledger that not only shows the beginning and current balance for each accounts but also lists all the journal transactions for the month.

As you can see in figure 8.11, the printed output from the monthly ledger posting program shows the balances of all ledger accounts, whether or not there were transactions for the account.



```
101   Checking account #1
      1271  04/02/86    Auto expense                      -78.70
      1272  04/02/97    Rent                             -500.00
      1274  04/02/97    Auto expense                      -31.83
      1273  04/04/97    Advertising                       -87.50
            Prev. bal:  5219.23  New bal:   4521.20
102   Checking account #2
       670  04/02/97    Office expense                    -32.78
            Prev. bal:  1321.20  New bal:   1288.42
505   Advertising expense
      1273  04/04/97    Newspaper ad re: new product       87.50
            Prev. bal:    25.00  New bal:    112.50
510   Auto expenses
      1271  04/02/97    Tune-up and minor repair           78.70
      1274  04/09/97    Oil change                         31.83
            Prev. bal:   501.12  New bal:    611.65
515   Bank charges
            Prev. bal:     0.00  New bal:      0.00
520   Books and publications
            Prev. bal:    87.40  New bal:     87.40
```

Figure 8.11 Sample ledger printout for the first six accounts.

In summary, there are three different steps in processing the ledger *entries:*

o Immediately after reading a new ledger object, we need to print the header line and initialize the balance for the next month from the previous month's balance.

p For each transaction object that matches, we need to update the account balance.

q After the last transaction for the account, the balance line should be printed.

Figure 8.13 has the code for the three-way-test loop of method PostTransactions.

```
While (MoreMasters ||MoreTransacions)

        if(Item(1)<Item(2)) { // finish this master record
        ProcessEndMaster();
        MoreMasters = NextItemInList(1);
        if(MoreMasters) ProcessNewMaster();
}
else if (Item(1)=Item(2)) { // transaction matches master
        ProcessCurrentMaster(); // another transaction for master
        ProcessItem(2); // output transaction record
        More transactions = NextItemInList(2);
}
else { // Item(1)>Item(2)) transaction with no master
        ProcessTransactionError();
        More transactions = NextItemInList(2);
}
```

Figure 8.13 Three-way-test loop for method PostTransactions of class MasterTransactionProcess

The reasoning behind the three-way test is as follows:

- If the ledger (master) account number (Item [1]) is Ièss-than the journal (transaction) account number (Item[2]), then there are no more transactions to add to the ledger account this month (perhaps there were none at all), so we print the. ledger account balances (ProcessEndMaster) and read in the next ledger account (NextItemlnList(I)). If the account exists (MoreMasters is true), we print the title line for the new account (ProcessNewMaster).

- If the account numbers match, then we have a journal transaction that is to be posted to the current ledger account. We add the transaction amount to the account balance for the new month (ProcessCurrentMaster), print the description of the transac tion (ProcessItem(2)), then read the next journal entry (NextItemlnList(1)).

- If the journal account is less- than the ledger account, then it is an unmatched journal account, perhaps due to-an input error. We print an error message (ProcessTransactionerror) and continue with the next transaction.

8.3 EXTENSION OF THE MODEL TO INCLUDE MULTIWAY MERGING

The most common application of cosequential processes requiring more than two input files is a *K-way merge,* in which we want to merge K input lists to create a single, sequentially ordered output list. K is often referred to as the *order* of a K-way merge.

8.3.1 A K-way Merge Algorithm (Also can refer program 8)

Suppose we keep an array of lists and array of the items (or keys) that are being used from each list in the cosequential process:

list [0] , list [I] , list [2] , list [k-I]

Item[O] , Item[I] , Item[3] , Item [k-I]

The main loop for the merge processing requires a call to a Minlndexfunc tion to find the index of item with the minimum collating sequence value and an inner loop that finds all lists that are using that item:

```
int minltem = Minlndex(Item,k); // find an index of minimum item
Processltem(minltem);          // Item(minltem) is the next
 for (i = 0; i<k; i++)            Output // 100`k at each list
```

```
if (Item(minltem) == ItemCi)) // advance list i
   Moreltems[i] = NextltemlnList(i);
```

Clearly, the expensive parts of this procedure are finding the minim um and testing to see in which lists the item occurs and which files therefore need to be read. Note that because the item can occur in several lists, every one of these *if* tests must be executed on every cycle
through the loop. However, it is often possible to guarantee that a Single item, or Key, occurs in only one list. In this case, the procedure becomes simpler and more efficient.

In short for your reference

K--way Merging Algorithm

An array of *K* index values corresponding to the current element in each of the *K* lists, respectively.

Main loop of the K-Way Merge algorithm:

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

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

get next item from List[*minItem*]

Different ways of implementing the method:

Solution 1: when the number of lists is small (say K<= 8)

►Line(1) does a sequential search on item[1], item[2], ..., item[*K*]
Running time: $O(K)$

►Line(5) just replaces item[i] with newly read item
Running time: $O(1)$

Solution 2: when the number of lists is large.

►When the number of lists is large, store current items item[1], item[2], ..., item[*K*] into priority queue (heap).

►Line(1) does a min operation on the heap.
Running time: $O(1)$

►Line(5) performs a **extract-min** operation on the heap:
Running time: $O(\log_2 K)$

►and an **insert** on the heap
Running time: $O(\log_2 K)$

The Detailed Analysis of Both Algorithm is somewhat involved

► Let N = Number of items in output list

M = Number of items summing up all input lists

(Note N≤M because of possible repetitions)

► Solution 1

– Line(1): $K \times N$ steps

– Line(5): counting all executions: $M \times 1$ steps

– Total time: $O(K \times N+M) \subseteq O(K \times M)$

► Solution 2

– Line(1): $1 \times N$ steps

– Line(5): counting all executions: $M \times 2 \times \log_2 K$ steps

– Total time: $O(N+M \times \log_2 K) \subseteq O(M \times \log_2 K)$

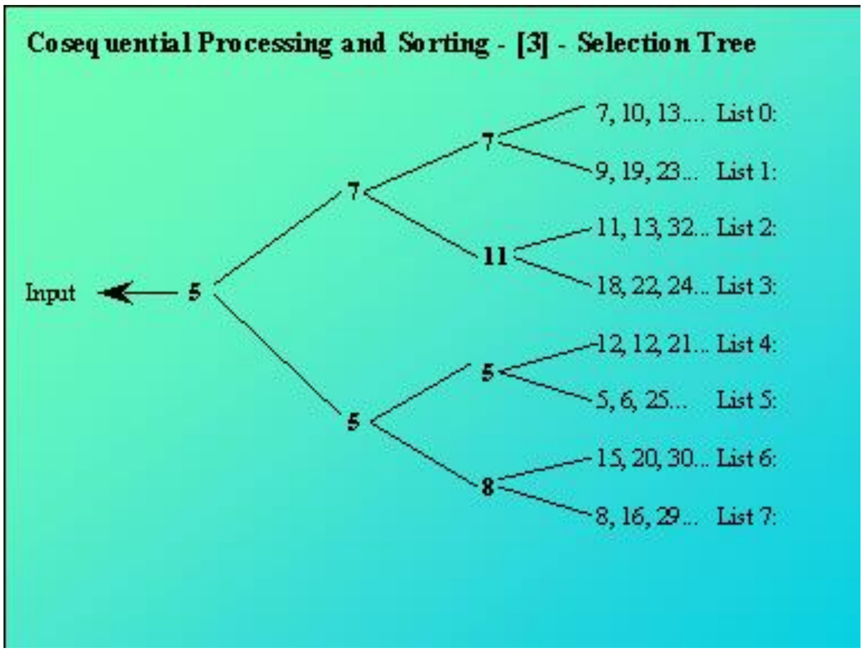
8.3.2 A Selection Tree for Merging Large Numbers of Lists

The K-way merge described earlier works nicely if K is no larger than 8 or so. When we begin merging a larger number of lists, the set of sequential comparisons to find the key with the minimum value becomes noticeably expensive.

The use of a selection tree is an example of the classic time- versus space trade-off we so often encounter in computer science. The concept underlying a selection tree can be readily communicated through a diagram such as that in Fig. 8.15.

The selection tree is a kind of *tournament* tree in which each higher level node represents the "winner" (in this case the *minimum* key value) of the comparison between the two descendent keys. The minimum value is always at the root node of the tree. If each key has an associated reference to the list from which it came, it is a simple matter to take the key at the root, read the next element from the associated list, then run the tournament again. Since the tournament tree is a binary tree, its depth is ceiling function $O(\log_2 K)$

for a merge of K lists. The number of comparisons required to establish a new tournament winner is, of course, related to this depth rather than being a linear function of K.



8.4 **A SECOND LOOK AT SORTING IN MEMORY**

Consider the problem of sorting a disk file that is small enough to fit in memory. The operation we described involves three separate steps:

 Read the entire file from disk into memory.

 Sort the records using a standard sorting procedure, such as shell sort.

 Write the file back to disk.

The total time taken to sort the file is the sum of the times for the three steps. Can we improve on the time that it takes for this memory sort?

8.4.1 Overlapping Processing and I/O: Heap sort

Most of the time when we use an internal sort, we have to wait until we have the whole file in memory, before we can start sorting. Is therein vernal sorting algorithm that is reasonably fast and that can begin sorting numbers immediately as they are read rather than waiting for the whole file to be in memory?

In fact there is it is called *heapsort.*

Heapsort keeps all of the keys in a structure called a *heap.* A heap is a binary tree with the following properties:

1. Each node has a single key, and that key is greater than or equal to the key at its parent node.

 It is a *complete* binary tree, which means that all of its leaves are on no more than two levels and that all of the keys on the lower level are in the leftmost position.

 Because of properties 1 and 2,.storage for the tree can be allocated Sequentially as an array in such a way that the root node is index 1 and the indexes of the left and right children of node i are 2i and 2i + 1, respectively.

Figure 8.16 shows a heap in both its tree form and as it would be Stored in an array.

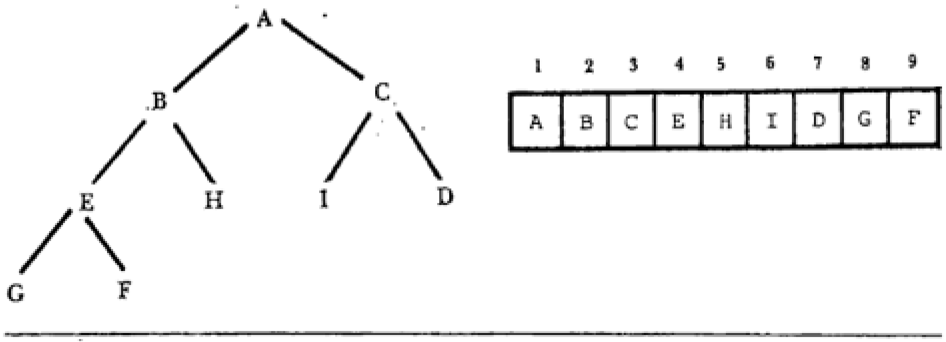


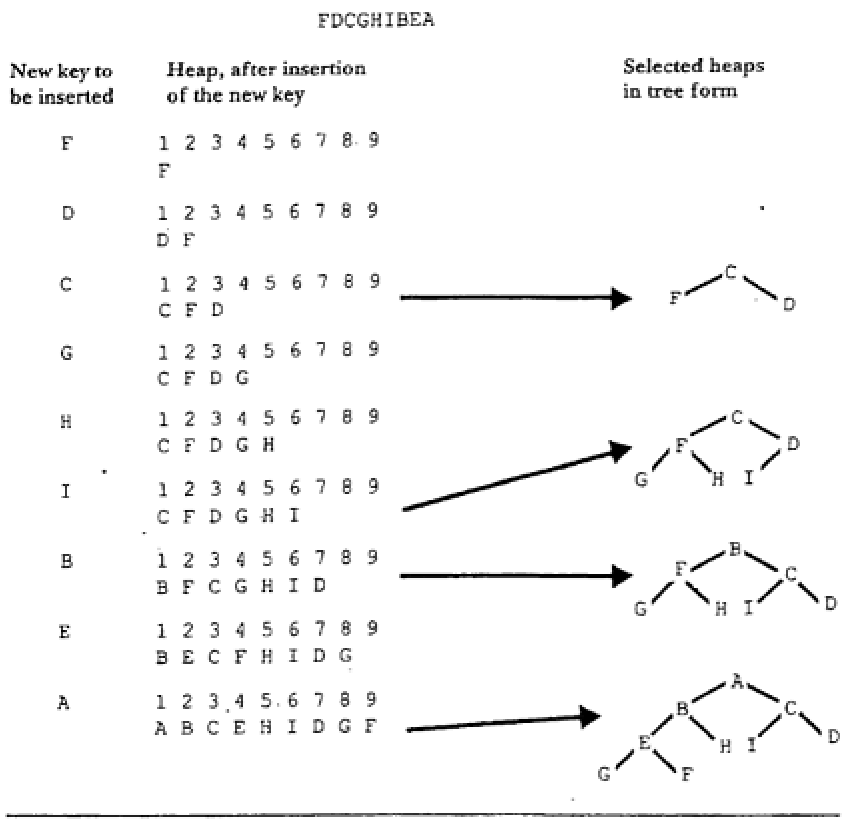
Figure 8.16 A heap in both its tree form and as it would be stored in an array.

8.4.2 Building the Heap While Reading the File

The algorithm for heap sort has two parts. First we build the heap; then we output the keys in sorted order. The first stage can occur at virtually the same time that we read the data, so in terms of elapsed time it comes essentially free. Insert method that adds a string to the heap is shown in Fig.8.17. Figure 8.18 contains a sample application of this algorithm.

```
int Heap::Insert(char * newKey)
{
   if (NumElements == MaxElements) return FALSE;
   NumElements++; // add the new key at the last position
   HeapArray[NumElements] = newKey;
   // re-order the heap
   int k = NumElements; int parent;
   while (k > 1) // k has a parent
   {
      parent = k / 2;
      if (Compare(k, parent) >= 0) break;
         // HeapArray[k] is in the right place
      // else exchange k and parent
      Exchange(k, parent);
      k = parent;
   }
   return TRUE;
}
```

Figure 8.17: Insert method to insert key in heap



FDCGHIBEA

New key to be inserted	Heap, after insertion of the new key								
	1	2	3	4	5	6	7	8	9
F	F								
D	D	F							
C	C	F	D						
G	C	F	D	G					
H	C	F	D	G	H				
I	C	F	D	G	H	I			
B	B	F	C	G	H	I	D		
E	B	E	C	F	H	I	D	G	
A	A	B	C	E	H	I	D	G	F

Figure 8.18 Sample application of the heap-building algorithm. The keys F, D, C, G, H, I, B, E, and A are added to the heap in the order shown.

How to make the input overlap with the heap-building procedure? We read a block of records at a time into an input buffer and then operate on all of the records in the block before going on to the next block. Each time we read a new block, we just append it to the end of the heap (that is, the input buffer "moves" as the heap gets larger). The first new record is then at the end of the heap array, as required by the Insert function (Fig. 8.17). Once that record ss absorbed into the heap, the next new record is at the end of the heap array, ready to be absorbed into the heap, and so forth.

Use of an input buffer avoids an excessive number of seeks, but it still doesn't let input occur at the same time that. We build the heap. The way to make processing overlap with I/O is to use more than one buffer. With multiple buffering, as we process the keys in one block from the file, we can simultaneously read later blocks from the file.

Figure 8.19 illustrates the technique that we have just described, in which we append each new block of records to the end of the heap, thereby employing a memory-sized set of input buffers. Now we read new blocks as fast as we can, never having to wait for processing before reading a new block.

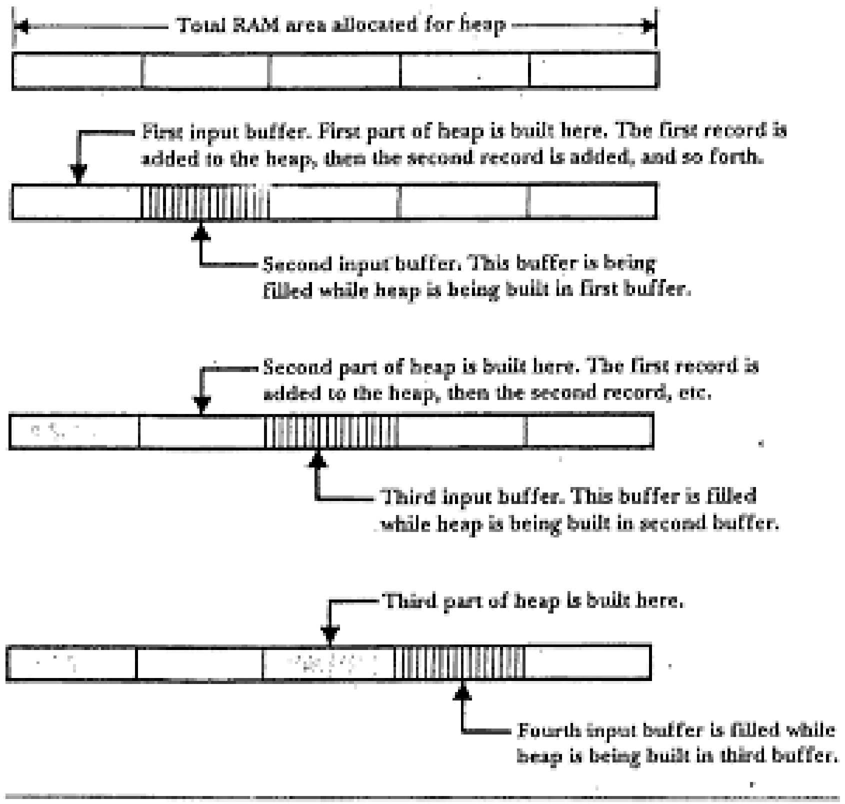

Total RAM area allocated for heap
First input buffer. First part of heap is built here. The first record is added to the heap, then the second record is added, and so forth.
Second input buffer. This buffer is being filled while heap is being built in first buffer.
Second part of heap is built here. The first record is added to the heap, then the second record, etc.
Third input buffer. This buffer is filled while heap is being built in second buffer.
Third part of heap is built here.
Fourth input buffer is filled while heap is being built in third buffer.

Figure 8.19 Illustration of the technique described in the text for overlapping input with heap building in memory.

8.4.3 Sorting While Writing to the File

The second and final step involves writing the heap in sorted order again, it is possible to overlap I/O (in this case writing), with processing. First, let's look at how to output the sorted keys. Retrieving the keys in order is $imply a repetition of the following steps:

 Determine the value of the key in the first position of the heap. This is the smallest value in the heap.

 Move the largest value in the heap into the first position, and decrease the number of elements by one. The heap is now out of order at its root.

 Reorder the heap by exchanging the largest element with the smaller of its children and moving down the tree to the new position of the largest element until the heap is back in order.

Each time these three steps are executed, the smallest value is retrieved and removed from the heap. Figure 8.20 contains the code for method Remove that implements these steps.

```
char * Heap::Remove()
{// remove the smallest element, reorder the heap,
 // and return the smallest element
   // put the smallest value into 'val' for use in return
   char * val = HeapArray[1];

   // put largest value into root
   HeapArray[1] = HeapArray[NumElements];
   // decrease the number of elements
   NumElements--;

   // reorder the heap by exchanging and moving down
      int k = 1; // node of heap that contains the largest value
      int newK; // node to exchange with largest value
      while (2*k <= NumElements)// k has at least one child
      {   // set newK to the index of smallest child of k
          if (Compare(2*k, 2*k+1)<0) newK = 2*k;
          else newK = 2*k+1;
          // done if k and newK are in order
          if (Compare(k, newK) < 0) break; //in order
          Exchange(k, newK); // k and newK out of order
          k = newK; // continue down the tree
      }
      return val;
}
```

Figure 8.20 MethodRemove of class Heap removes the smallest element and reorders the heap.

First we see that we know immediately which record will be written first in the sorted file; next, we know what will come second; and so forth. So as soon as we have identified a block of records, we can write that block, and while we are writing that block, we can identify the next block, and so forth.

8.5 MERGING AS A WAY OF SORTING LARGE FILES ON DISK

The multiway merge algorithm discussed in Section 8.3 provides the beginning of an attractive solution to the problem of sorting large files.

As example consider a file with 8000000 records, each of which is 100 bytes long and contains a key field that is 10 bytes long. we can create a sorted subset of our full file by reading records into memory until the memory work area is almost full, sorting the records in this work area, then writing the sorted records back to disk as a sorted sub file. We call such a sorted sub file a *run.*

A schematic VIEW of this run creation and merging process is provided in Fig. 8.21. This solution to our sorting problem has the following features:

- It can, in fact, sort large files and can be extended to files of any size.
- Reading of the input file during the run-creation step is sequential and hence is much faster than input that requires seeking for every record individually (as in a keysort.)
- Reading through each run during merging and writing the sorted records is also sequential.

Random accesses are required only as we switch from run to run during the merge operation.

 If a heap sort is used for the in- memory part of the merge, as described in Section 8.4, we can overlap these operations with I/O so the in memory part does not add appreciably to the total time for the merge.

 Since I/O is largely sequential, tapes can be used if necessary for both input and output operations.

►Characteristics of the file to be sorted: 8,000,000 records

Size of a record = 100 Bytes

Size of the key = 10 Bytes

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

Total file size = 800 MB

Total number of bytes for all the keys = 80 MB

►So, we cannot do internal sorting

Solution

►Forming runs: bring as many records as possible to main memory, do internal sorting and save it into a small file. Repeat this procedure until we have read all the records from the original file

►Do a multiway merge of the sorted files

In our example, what could be the size of a run?

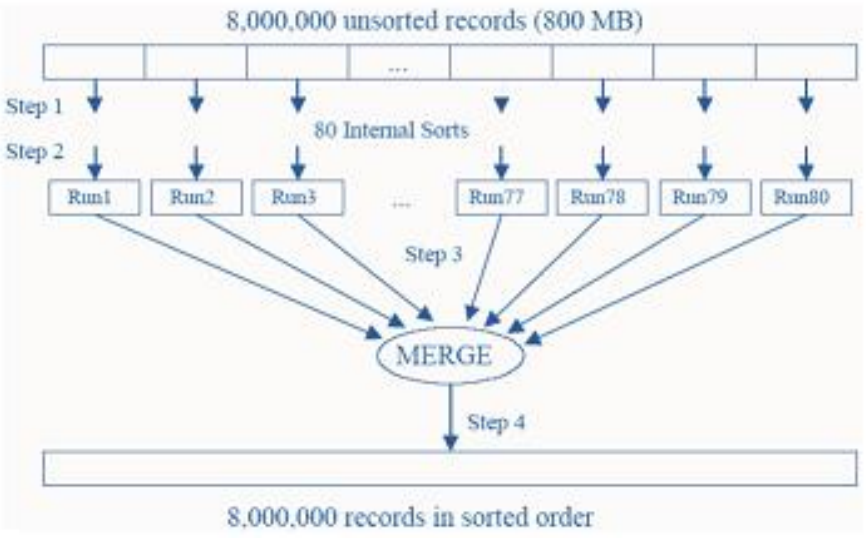
Available memory = 10 MB ≅ 10,000,000 bytes

Record size = 100 bytes

Number of records that can fit into available memory = 100,000 records

Number of runs = 80 runs

80 Internal Sorts



I/O operations are performed in the following times:

1. Reading each record into main memory for sorting and forming the runs
2. Writing sorted runs to disk

The two steps above are done as follows:

– Read a chunk of 10 MB; Write a chunk of 10 MB (Repeat this 80 times)

– In terms of basic disk operations, we spend:

– For reading: 80 seeks + transfer time for 800 MB Same for writing.

3. Reading sorted runs into memory for merging. In order to minimize "seeks" read one chunk of each run, so 80 chunks. Since the memory available is 10 MB each chunk can have 10,000,000/80 bytes = 125,000 bytes = 1,250 records
   – How many chunks to be read for each run?
   – size of a run/size of a chunk = 10,000,000/125,000=80
   – Total number of basic "seeks" = Total number of chunks (counting all the runs) is 80runs × 80 chunks/run = 80^2 chunks

   Reading each chunk involves **basic seeking.**
4. When writing a sorted file to disk, the number of basic seeks depends on the size of the output buffer: bytes in file/ bytes in output buffer.
   – For example, if the output buffer contains 200 K, the number of basic seeks is 200,000,000/200,000 = 4,000

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

The Bottleneck

► There are ways of reducing the time for the bottleneck step 3

1. Allocate more resource (e.g. disk drive, memory)
2. Perform the merge in more than one step – this reduces the order of each merge and increases the run sizes
3. Algorithmically increase the length of each run
4. Find ways to overlap I/O operations

8.5.1 How Much Time Does a Merge Sort Take?

We See in Fig. 8.21 that there are four times when I/O is performed. During the sort phase:

1. Reading all records into memory for sorting and forming runs, and
2. Writing sorted runs to disk.

 Reading sorted runs into memory for merging, and

 Writing sorted file to disk.

Let's look at each of these in order.

Step 1: Reading Records into Memory for Sorting and Forming Runs

Since we sort the file in 10- megabyte chunks, we read 10 megabytes at a time from the file. In a sense, memory is a 10- megabyte input buffer that we fill up eighty times to form the eighty runs. In computing the total time to input each run, we need to include the amount of time it takes to *access* each block (seek time + rotational delay), plus the amount of time it takes to *transfer* each block. Seek and rotational delay times are 8 msec and 3 msec, respectively, so total time per Seek is 11 msec. The transmission rate is approximately 14,500 bytes per msec. Total input time for the sort phase consists of the time required for 80 seeks,

plus the time required to transfer 800 megabytes:

Access:	80 seeks x 11 msec =	1 Sec
Transfer:	800 megabytes @ 14500 bytes/msec=	60 sec
Total:		61 Sec

Step 2: Writing Sorted Runs to Disk

In this case, writing is just the reverse of reading-the same number of seeks and the same amount of data to transfer. So it takes another 61 seconds to write the 80 sorted runs.

Step 3: Reading Sorted Runs Into Memory for Merging -

Since we have 10 megabytes of memory for storing runs, we divide 10 megabytes into 80 parts for buffering the 80 runs. In a sense, we are reallocating our 10 megabytes of memory as 80 input buffers. Each of the 80 buffers then holds 1/80th of a run (125 000 bytes), so we have to access each run 80 times to read all of it. Because there are 80 runs, in order to complete the merge operation (Fig. 8.22) we end up making

80 runs x 80 seeks = 6400 Seeks.

Total seek and rotation time is then 6400 x 11 msec = 70 seconds.

Since 800 megabytes is still transferred, transfer time is still 60 seconds.

Step 4: Writing Sorted File to Disk

To compute the time for writing the file, we need to know how big our output buffers are. To keep matters simple, let us assume that we can allocate two 200 000-byte output buffers. With 200000 bytes buffers, we need to make 4000 seeks.

Total seek and rotation time is then 4000 x 11 msec = 44 seconds. Transfer time is still 60 seconds.

Simplifying assumptions:

- Only one seek is required for any single sequential access.
- Only one rotational delay is required per access.

Expensive steps (i.e. involving I/O) occurring in MergeSort

- During the ***sort phase***:
  - Reading all records into memory for sorting and forming runs.
  - Writing sorted runs to disk
- During the ***merge phase***:
  - Reading sorted runs into memory for merging.
  - Writing sorted file to disk.

8.5.2 Sorting a file that is Ten Times Larger

What kinds of I/O take place during the Sort and the Merge phases?

- Since, during the ***sort phase***, the runs are created using heapsort, I/O is ***sequential***. ***No performance improvement*** can ever be gained in this phase.
- During the ***reading step*** of the ***merge phase***, there are a lot of ***random accesses*** (since the buffers containing the different runs get loaded and reloaded at unpredictable times). The number and size of the memory buffers holding the runs determine the number of random accesses. ***Performance improvements*** can be made in this step.
- The ***write step*** of the ***merge phase***, is ***not influenced*** by the way in which we organize the runs.

8.5.3 ***The Cost of Increasing the File Size***

- In general, for a K-way merge of K runs where each run is as large as the memory space available, the buffer size for each of the runs is: ***(1/K)* size of memory space = (1/K) * size of each run***.
- So K seeks are required to read all of the records in each individual run and since there are K runs altogether, the merge operation requires K^2 seeks.
- Since K is directly proportional to N, the number of records, SortMerge is an $O(N^2)$ operation, measures in terms of seeks.

What can be done to Improve MergeSort Performance?

There are different ways in which MergeSort's efficiency can be improved:

- Allocate more Hardware such as disk drives, memory, and I/O channels.
- Perform the merge in more than one step, reducing the order of each merge and increasing the buffer size for each run.
- Algorithmically increase the lengths of the initial sorted runs.
- Find ways to overlap I/O Operations.

8.5.4 ***Hardware-Based Improvements***

- ***Increasing the amount of memory***: helps make the buffers larger and thus reduce the numbers of seeks.
- ***Increasing the Number of Dedicated Disk Drives***: If we had one separate read/write head for every run, then no time would be wasted seeking.
- ***Increasing the Number of I/O Channels***: With a single I/O Channel, no two transmission can occur at the same time. But if there is a separate I/O Channel for each disk drive, then I/O can overlap completely.
- But what if hardware based improvements are not possible?

8.5.6 ***Increasing Run Lengths Using Replacement Selection***

Replacement Selection Procedure:

- Read a collection of records and sort them using heapsort. The resulting heap is called the *primary heap*.
- Instead of writing the entire primary heap in sorted order, write only the record whose key has the lowest value.
- Bring in a new record and compare the values of its key with that of the key that has just been output.

- If the new key value is higher, insert the new record into its proper place in the primary heap along with the other records that are being selected for output.
- If the new record's key value is lower, place the record in a secondary heap of records with key values smaller than those already written.

- Repeat Step 3 as long as there are records left in the primary heap and there are records to be read. When the primary heap is empty, make the secondary heap into the primary heap and repeat steps 2 and 3.

Analysis of Run Length Selection

- ***Question 1***: Given P locations in memory, how long a run can we expect replacement selection to produce on average?
- ***Answer 1***: On average we can expect a run length of 2P.
- ***Question 2***: What are the costs of using replacement selection?
- ***Answer 2:*** Replacement Selection requires much more seeking in order to form the runs. However, the reduction in the number of seeks required to merge the runs usually more than offsets that extra cost.

8.5.7 *Replacement Selection + MultiStep Merging*

- In practice, Replacement Selection is not used with a one-step merge procedure.
- Instead, it is usually used in a two-step merge process.
- The reduction in total seeks and rotational delay time is most affected by the move from one-step to two-step merges, but the use of Replacement Selection is also somewhat useful.

8.5.8 *Using Two Disk Drives with Replacement Selection*

- Replacement Selection offers an opportunity to save on both transmission and seek times in ways that memory sort methods do not.
- We could use one disk drive to do only input operations and the other one to do only output operations.
- This means that:
  - Input and Output can overlap ==> Transmission time can be decreased by up to 50%.
  - Seeking is virtually eliminated.

8.5.9 More Drives? More Processor?

- We can make the I/O process even faster by using more than two disk drives.
- If I/O becomes faster than processing, then more processors can be used. Different network architectures can be used for that:
  - Mainframe computers
  - Vector and Array processors
  - Massively parallel machines
  - Very fast local area networks and communication software.

8.5.10 Effects of multi programming

In our discussions of external sorting on disk we are, of course, making tacit assumptions about the computing environment in which this merging is taking place. We are assuming, for example, that the merge job is running in a dedicated environment (no multiprogramming). If, in fact, the operating system is multiprogrammed, as it normally is, the total time for the I/O might be longer, as our job waits for other jobs to perform their I/O.

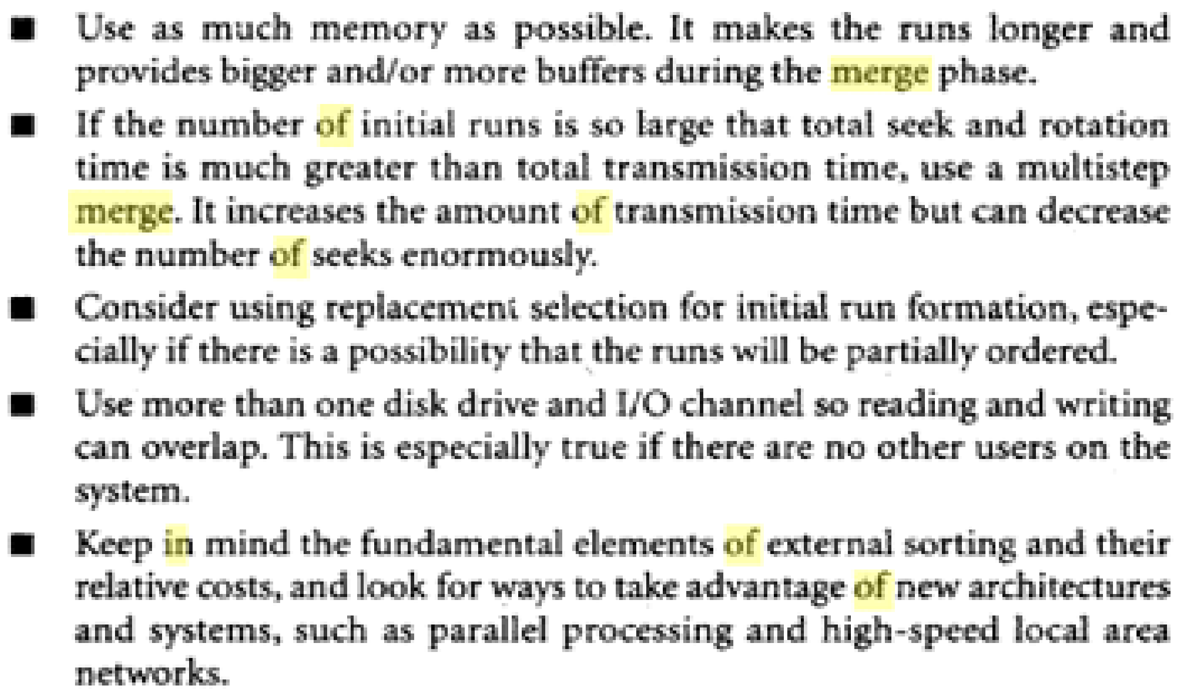
On the other hand, one of the reasons for multiprogramming is to allow the operating system to find ways to increase the efficiency of the overall system by overlapping processing and I/O among different jobs. So the system could be performing I/O for our job while it is doing CPU processing on others, and vice versa, diminishing any delays caused by overlap of I/O and CPU processing within our job.

Effects such as these are hard to predict, even when you have much information about your system. Only experimentation can determine what real performance will be like on a busy, multiuser system.

8.5.11 A Conceptual Toolkit for External Sorting

We can now list many tools that can improve external sorting performance. It should be our goal to add these various tools to our conceptual toolkit for designing external sorts and to pull them out and use them whenever they are appropriate. A full listing of our new set of tools would include the following:

- For in-memory sorting, use heapsort for forming the original list of sorted elements in a run. With it and double buffering, we can overlap input and output with internal processing.



- Use as much memory as possible. It makes the runs longer and provides bigger and/or more buffers during the merge phase.
- If the number of initial runs is so large that total seek and rotation time is much greater than total transmission time, use a multistep merge. It increases the amount of transmission time but can decrease the number of seeks enormously.
- Consider using replacement selection for initial run formation, especially if there is a possibility that the runs will be partially ordered.
- Use more than one disk drive and I/O channel so reading and writing can overlap. This is especially true if there are no other users on the system.
- Keep in mind the fundamental elements of external sorting and their relative costs, and look for ways to take advantage of new architectures and systems, such as parallel processing and high-speed local area networks.

CHAPTER - 9

MULTILEVEL INDEXING AND B-TREES

9.1 Introduction

Limitation of Indexing i.e., if index file itself is so voluminous that only rather small parts of it can be kept in main memory at one time, led to utilization of trees concept to file structures. Evolution of the trees for file structures started with applying Binary search tree (BST) for file structures.

The limitation was unbalanced growth of the BST.

This gave rise to applying of AVL tree concept to File structures. With this the problem of height balance was addressed but to make the tree height balance lot of local rotations was required. This limitation gave rise to development of B- 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 is limited.

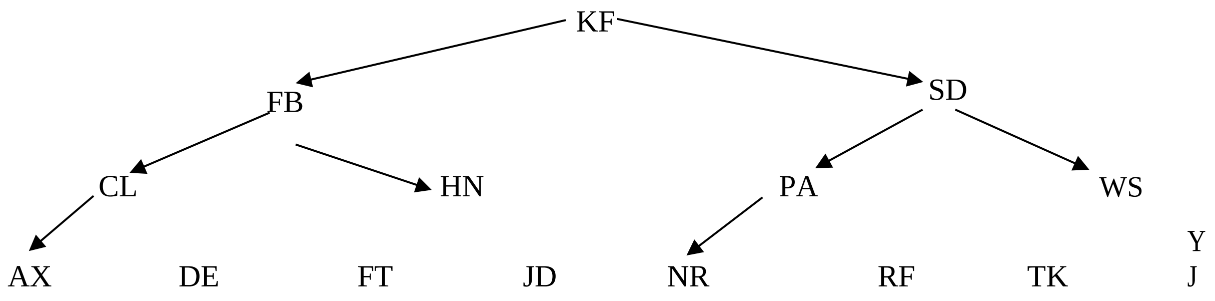
9.2 Statement of the problem

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

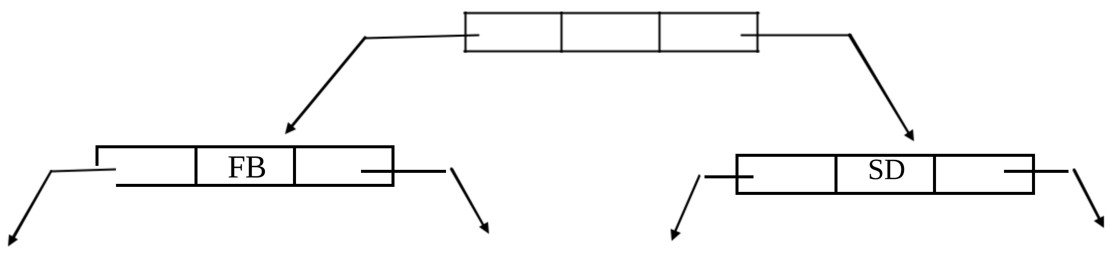
9.3 Indexing with Binary Search Trees

Given the sorted list below we can express a binary search of this list as a binary search tree, as shown in the figure below.

AX, CL, DE, FB, FT, HN, JD, KF, NR, PA, RF, SD, TK, WS, YJ



Using elementary data structure techniques, it is a simple matter to create node that contains right and left link fields so the binary search tree can be constructed as a linked structure.



The figure below shows record contents for a linked representation of the above binary tree.

ROOT

	Key	Left Child	Right Child
0	FB	10	8
1	JD		
2	RF		
3	SD	6	13
4	AX		
5	YJ		
6	PA	11	2
7	FT		

	Key	Left Child	Right Child
8	HN	7	1
9	KF	0	3
10	CL	4	12
11	**NR**	**15**	**--**
12	DE		
13	WS	14	5
14	TK		
15	***LV***		

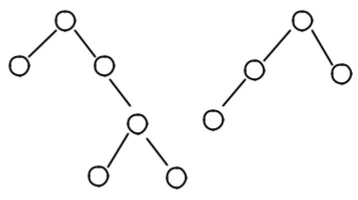
The records in the file illustrated in the figure above appear in random rather than sorted order. The sequence of the records in the file has no necessary relation to the structure of the tree: all the information about the logical structure is carried in the link fields. The very positive consequence that follows from this is that if we add a new key to the file such as LV, we need only link it to the appropriate leaf node to create a tree that provides search performance that is as good as we would get with a binary search on a sorted list. (Showed in Bold)

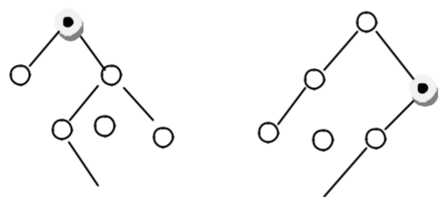
9.3.1 AVL Trees (G. M Adel'son, Velskii, E. M Landis)

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

- π No two subtree's height of any root differs by more than one level.
- θ AVL trees maintain *HB(1)* with local rotations of the nodes.
- ρ AVL trees are not perfectly balanced.
- σ Worst case search with an AVL tree is 1.44 $\log_2$ (N + 2) compares.

Examples of AVL trees: Examples of Non AVL Tress:

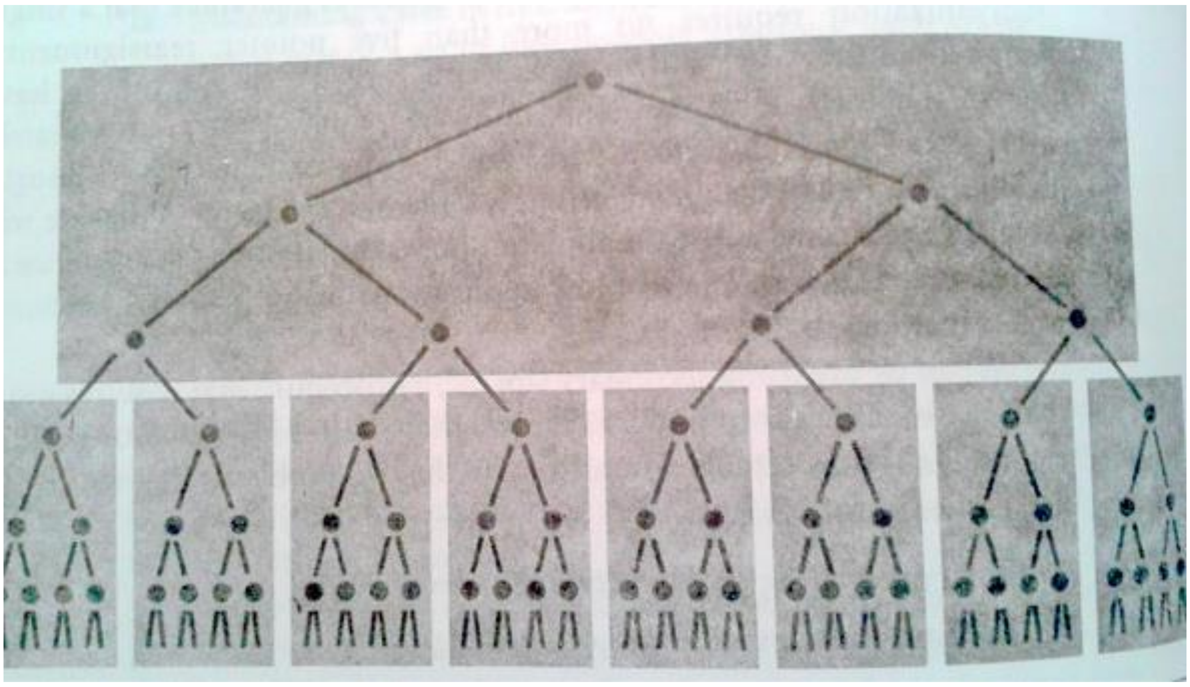




9.3.2 Paged Binary Trees

Disk utilization of a binary search tree is extremely inefficient. That is when a node of binary search tree is read, there are only three useful information. They are the key value, the address of the left and right sub trees. Each disk read produces a minimum of single page. The paged binary tree attempts to address this problem by locating multiple binary nodes on the same disk page. Paging divides a binary tree in to pages and then storing each page in a block of contiguous locations on disk, so that reduces the number of seeks associated with search.

- 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.



The Problem with Paged Binary Trees

- Only valid when we have the entire set of keys in hand before the tree is built.
- Problems due to out of balance.

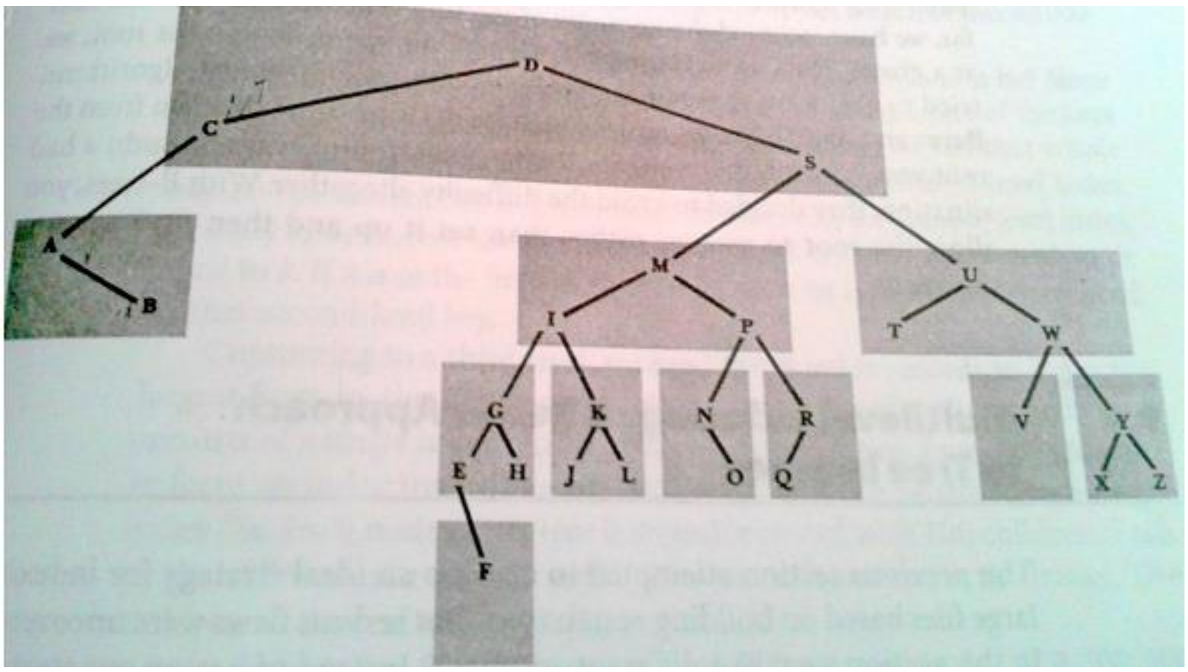


Fig: out of balance tree

9.4 Multilevel Indexing: A Better Approach to Tree Indexes

Consider a 80 MB file which has 80,00,000 (80 lakhs) records. Each record is 100 bytes & each key is 10 – byte keys. An index of this file has 80,00,000 key – reference pairs divided among a sequence of index records. Let's suppose that we can put 100 key – reference pairs in a single index record. Hence there are 80,000 records in the index.

The 100 largest keys are inserted into an indexed record, and that record is written to the index file. The next largest 100 keys go into the next record of the file, and so on. This continues until we have 80,000 index records in the index file. We must find a way to speed up the search of this 80000 record file. So we construct an index to index file. Then an index to index to index file is constructed. This is as shown in the below figures.

Data file (80 MB)

1
2
3
..
8000000

Index file to data file

1 2 3 4 100
101, 102, 103, ...200
201, 202, 203,......300
....
7999901, 7999902,....8000000

Index to Index File

100, 200, 300, 400,....8000

8100, 8200, 8300,....16000

16100, 16200,.........24000

....

.......7999900, ...8000000

Like this we will be having 80k records with 800 records. Each record contains 100 keys.

Index to Index to Index File

We will be having 800 keys with 8 records. Each record contains 100 keys.

9.5 B-Trees: Working up from the Bottom

B – Trees are multileveled indexes that solve the problem of linear cost of insertion and deletion.

In B – trees the overflow record is split into two records, each half full. Deletion takes a similar strategy of merging two records into a single record when necessary. Each node of a B – tree is index record.

Each record has same maximum number of key reference pairs, called the order of the B – tree. The records also have a minimum number of key-reference pairs, typically half of the order.

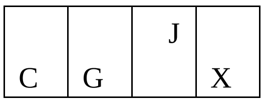
- An attempt to insert a new key into an index record that is not full is simply done by updating the index record.
- If the new key is the new largest key in the index record, it is the new higher level key of that record, and the next higher level of the index must be updated.
- When insertion into an index record causes it to be overflow, it is split into two records, each with half of the keys. Since a new index node has been created at this level, the largest key in this new node must be inserted into the next higher level node. This is called promotion of the key.
- Sometimes promotion of key may cause an overflow at that level. This in turn causes that node to be split, and a key promoted to the next level. This continues as far as necessary. This causes another level to be added to the multilevel index.

9.6 Example of Creating a B – Tree. (Also refer to the example solved in the class).

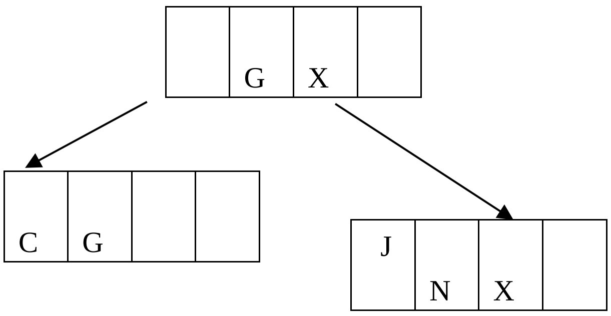
Construct a B – Tree of order 4, for the following set of keys

CGJXNSUOAEBHIFKLQRTV

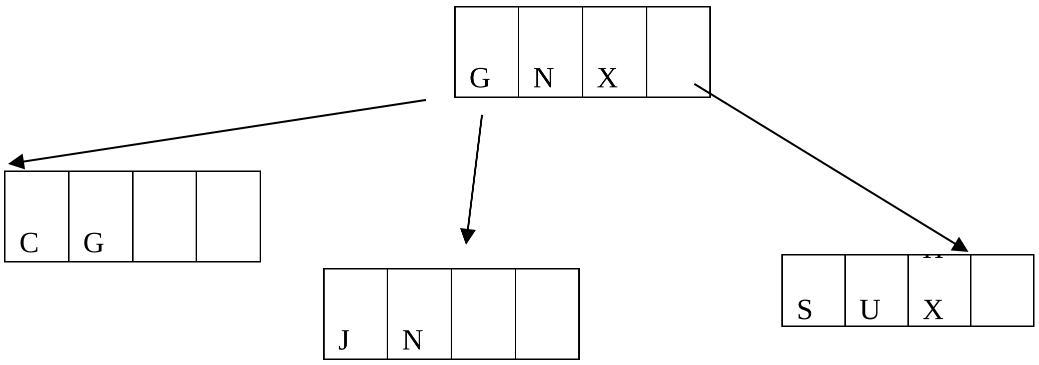
Step 1: After Insertion of C G J X



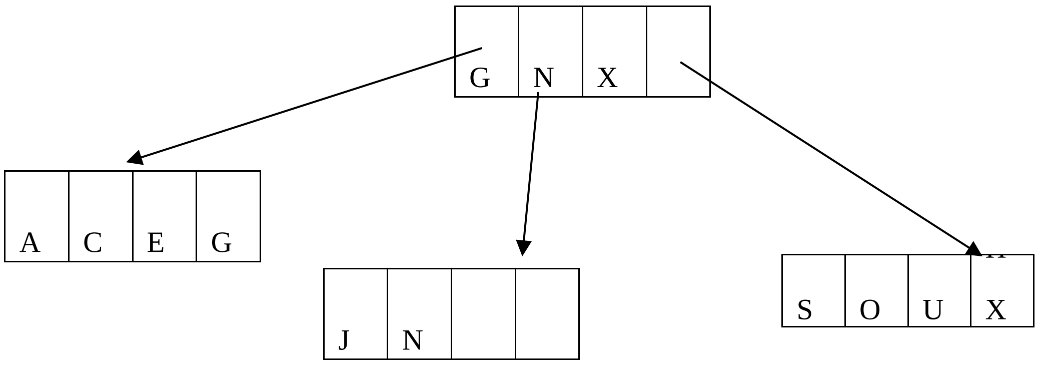
Step 2: Insertion of N causes node to split & the largest key in each leaf node(G & X) to be placed in the node.



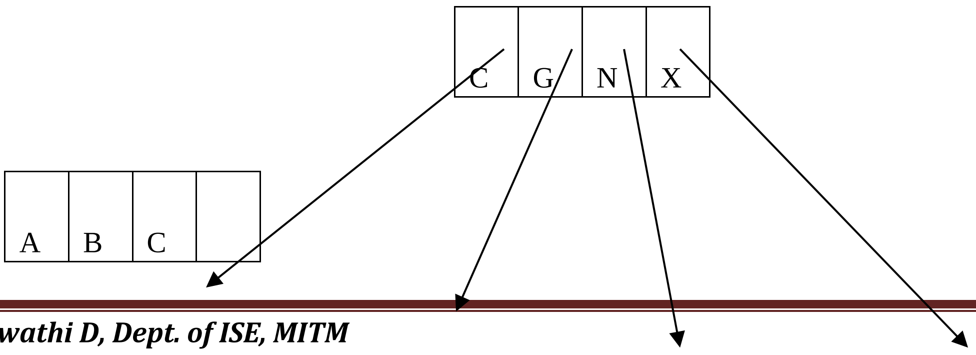
Step 3: 'S' is inserted into rightmost leaf node, and insertion of 'U' causes it to split.

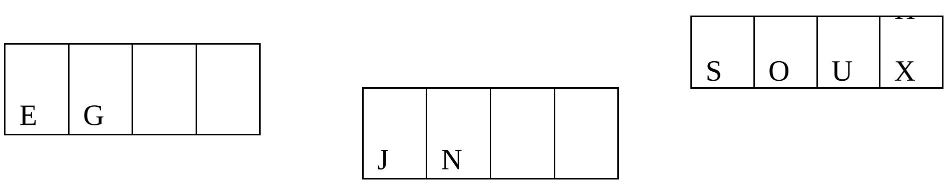


Step 4: 'O', 'A', 'E' are inserted into appropriate leaf nodes

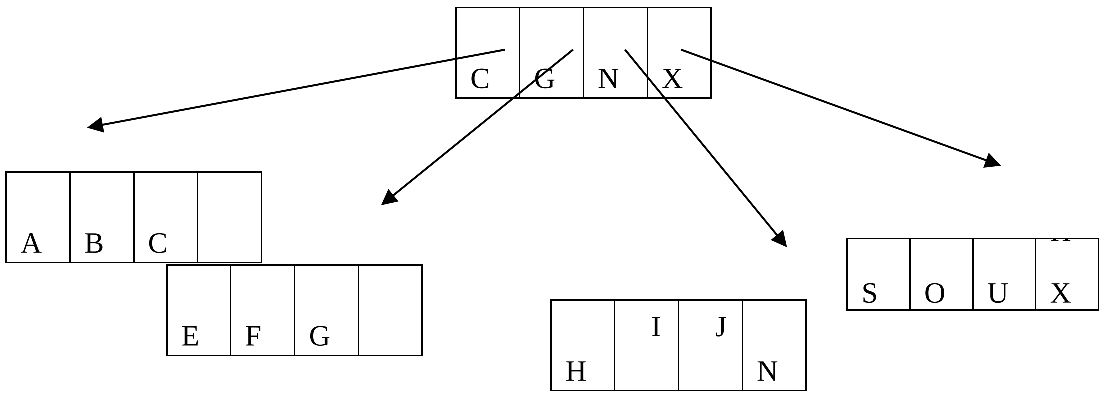


Step 5: Insertion of 'B' causes leftmost leaf node to split.

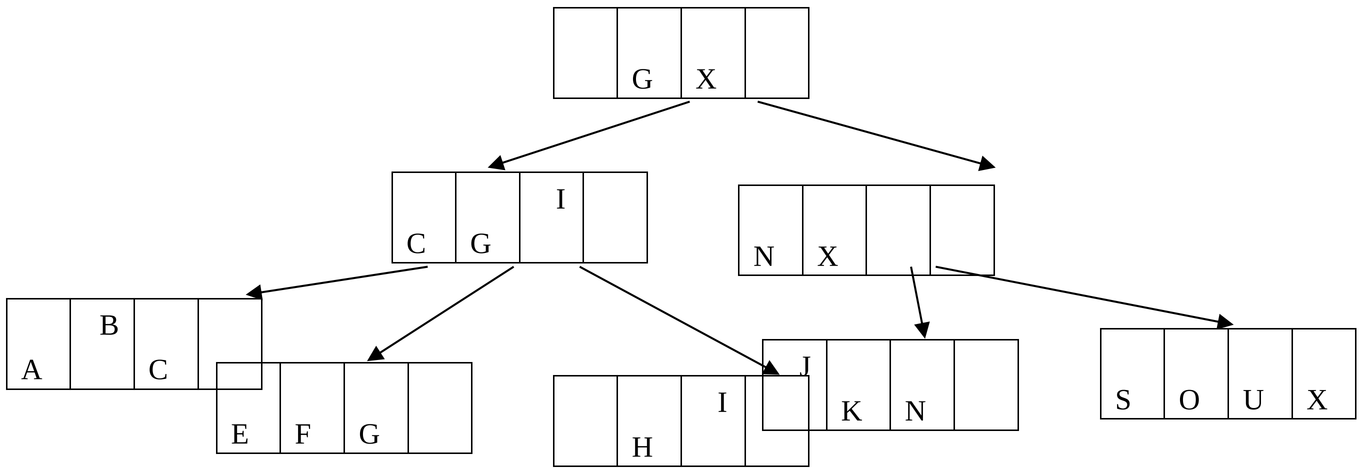




Step 6: 'H', 'I' and 'F' are inserted at appropriate leaf node.



Step 7: Insertion of 'K' causes leftmost leaf node to split, also root to split and the tree grows to level 3.



9.9 B – Tree Nomenclature

- Order of a B – tree:

According to **Bayer, Mc Creight & Come r** order of the B – tree is **minimum** number of **keys** that can be in a page of the tree.

Knuth definition of order of the B- tree is **maximum** number of **descendants** that a page can have.

- LEAF

Bayer and **Mc Creight** refer to the **lowest** level of keys in a B – tree as the leaf level.

Knuth considers the leaf of a B – tree to be **one level below the lowest level** of keys. In other words, he considers the leaves to be the actual data records that might be pointed to by the lowest level of the keys in the tree.

9.10 Formal Definition of B-Tree Properties

For a B-Tree of Order m,

Every page has a maximum of m descendants.

- Every page, except for the root and the leaves, has a minimum of $\lceil m/2 \rceil$ descendants.
- The root has a minimum of 2 descendants (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.

9.11 Worst Case Search Depth

The maximum number of disk accesses required to locate a key in the tree is nothing but worst case search. Every key appears in the leaf level. So finding the depth of the tree is nothing but worst case search. The worst case occurs when every page of the tree has only the minimum number of descendants. In such case the keys are spread over a maximal height for the tree and a minimal breadth.

For a B – tree of order 'm' the minimum number of descendants from the root page is two, so the second level of the tree contains only two pages. Each of these pages in turn has at least ceiling function of m/2 given by " $\lceil m/2 \rceil$ " descendants. The general pattern of the relation between depth and the minimum number of descendants takes the following form:

Level	Minimum Number of Descendants
1 (root)	2
2	$2^* \lceil m/2 \rceil$
3	$2^*(\lceil m/2 \rceil)^2$
4	$2^*(\lceil m/2 \rceil)^3$
d	$2^*(\lceil m/2 \rceil)^{d-1}$

So in general for any level d of B – tree, the minimum number of descendants extending from that level is

$$2^*(\lceil m/2 \rceil)^{d-1}$$

For a tree with 'N' keys in its leaves, we express the relationship between keys and the minimum height d as

□ $$\geq 2^* \lceil m/2 \rceil^{d-1}$$

Solving for d we arrive at the expression:

$$d \leq 1 + \log_{\lceil m/2 \rceil} (N/2)$$

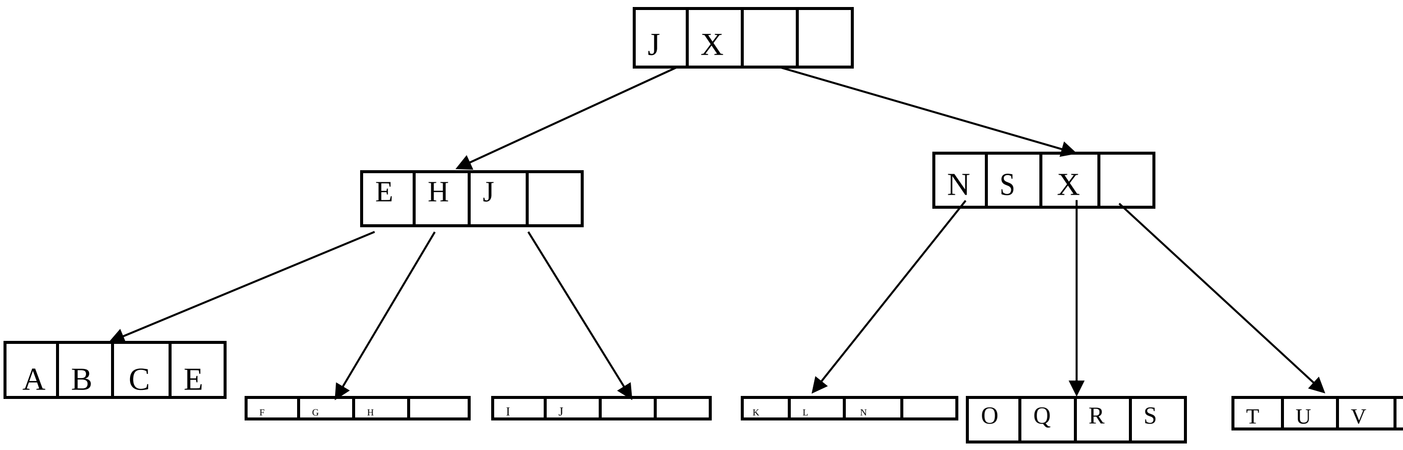
Problem based on the depth calculation refer class notes.

9.12 Deletion, Merging and Redistribution

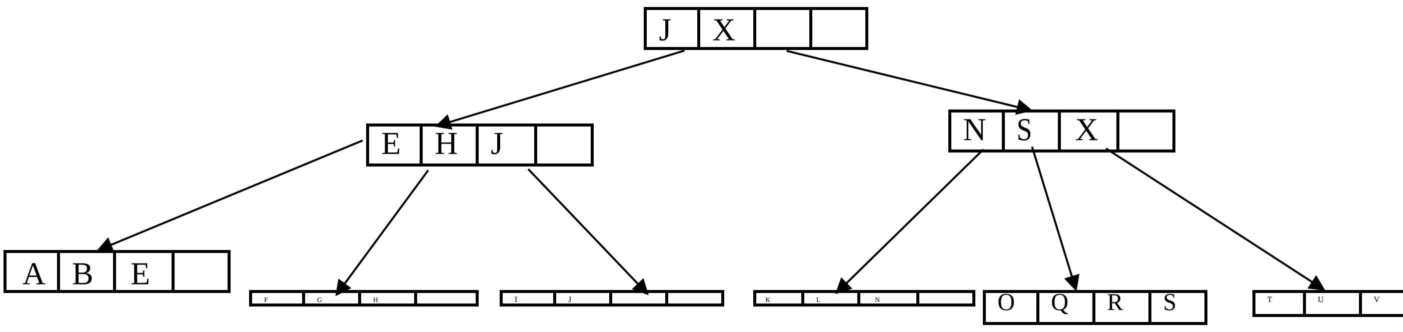
Deletion of a key can result in several different situations. Consider the B – tree below. What

happens when we delete some of its keys.

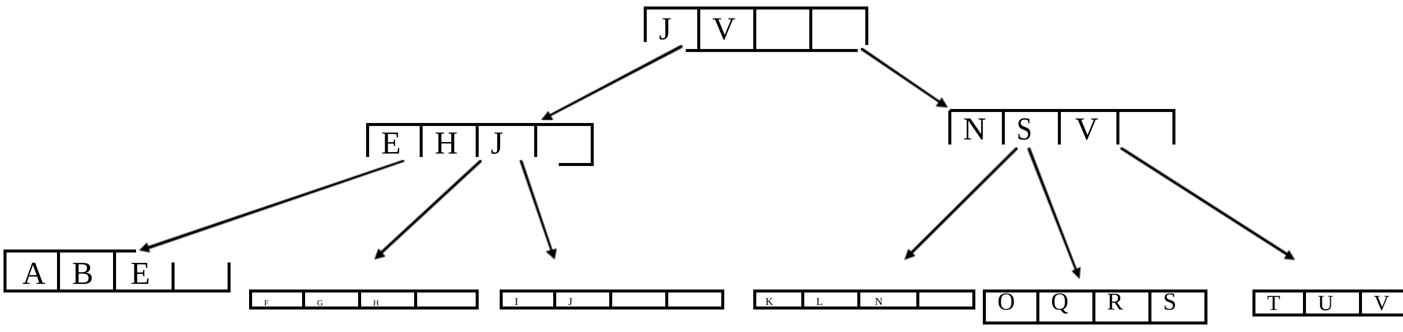
The simplest situation is illustrated by deleting key 'C'



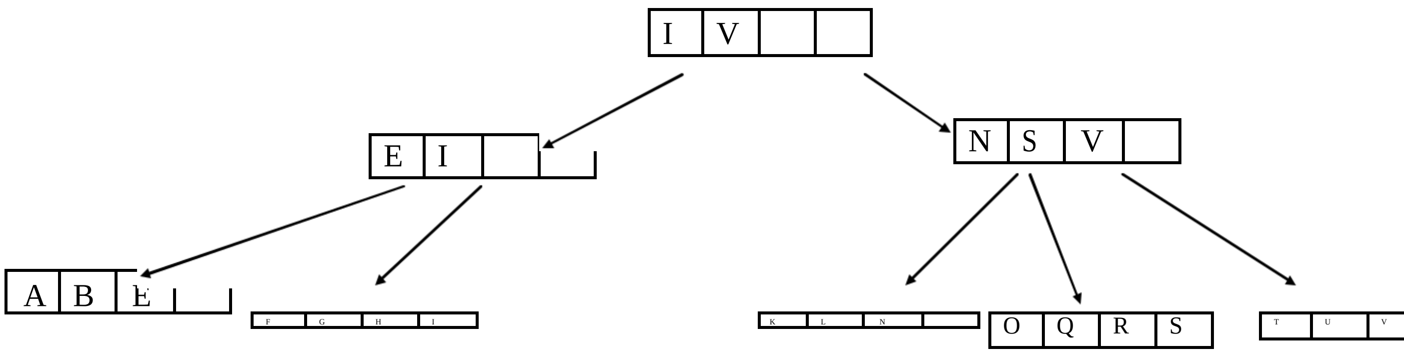
Removal of key 'C', change occurs only in leaf node.



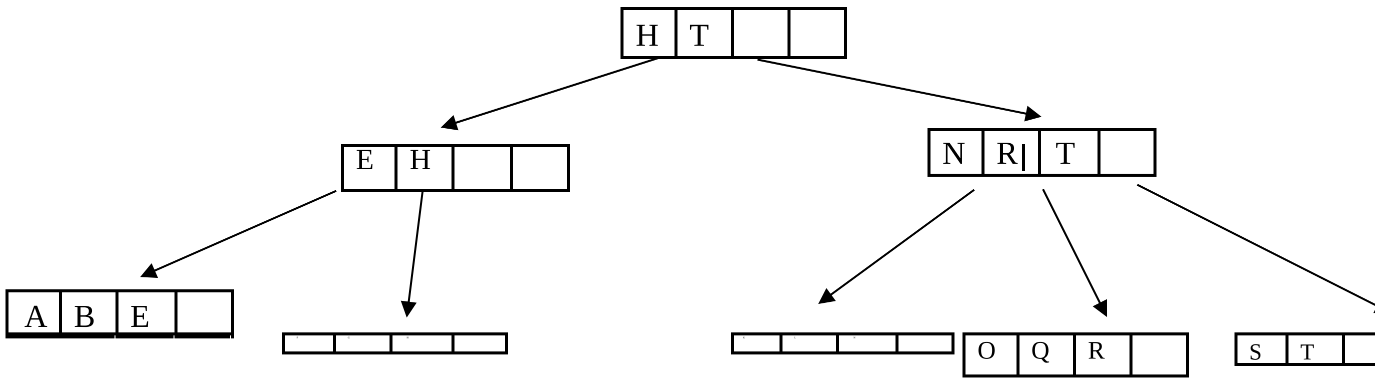
Removal of key 'X' change occurs in level 2 which needs to be updated in level 1 also.



Removal of key 'J' causes underflow of that leaf. So it is merged.

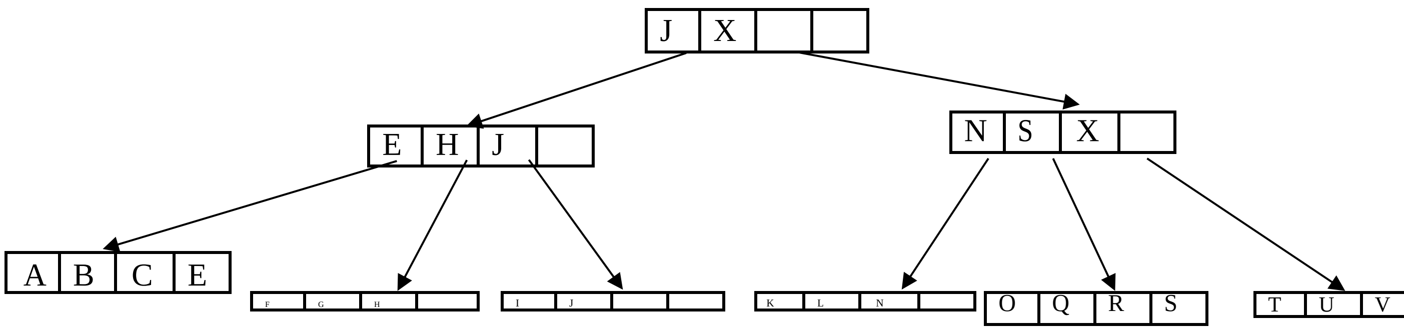


Deletion of keys ‘U’ & ‘V’ causes the right most leaf node to underflow. Here redistribution of keys among the nodes needs to be done. This redistribution leads to be updated in level 2 which in turn needs to be updated in level 1.

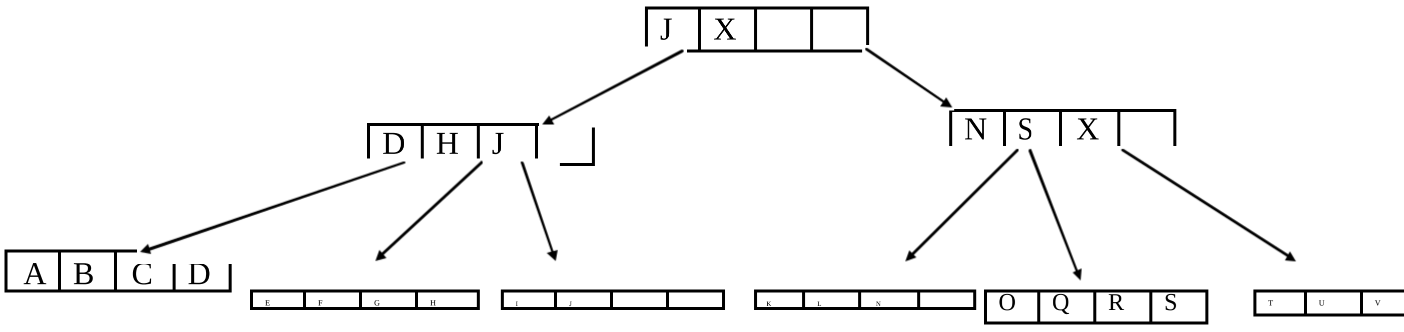


9.13 Redistribution during Insertion: A Way to Improve Storage Utilization

Redistribution during insertion is a way of avoiding, or at least postponing, the creation of new pages. Rather than splitting a full page and creating two approximately half- full pages, redistribution lets us place some of the overflowing keys in to another page. This indeed makes B- Tree more efficient in space utilization. For example consider the B - tree below



Now if key ‘D’ needs to be inserted, the left most page overflow and it needs to be split which results in one more leaf node added to B - Tree. Which results in approximately two half full pages. Instead of splitting the nodes if we can redistribute the keys among the nodes, we can postpone the splitting. The B – Tree below shows after insertion of key 'D' and redistribution.



9.15 Buffering of Pages: Virtual B – Trees

Consider an index file which is 1 MB of records and we have memory of 256 KB. Given a page size of 4 KB, holding around 64 keys per page, the B – tree can be contained in three levels. We may need at the max 2 disk seeks to retrieve any key from the B – Tree. To still improve on this we create a page – buffer to hold some number of B – tree pages, perhaps 5, 10 or more. As we read pages in from the disk in response to user request, we fill up t he buffer. Then when a page is requested, we access it from the memory if we can, thereby avoiding a disk access. If the page is not in the memory, then we read it in to the buffer from secondary storage, replacing one of the pages that were previously there. A B- tree that uses a memory buffer in this way is sometimes referred to as a Virtual B – Tree. The page replacement algorithms are:

- LRU- Least recently used.
- Replacement based on the page height.—Here we always retain the pages that occur at the highest levels of the tree.

*******END OF MODULE- 3********

MODULE – 4
INDEXED SEQUENTIAL FILE ACCESS AND PREFIX B+ TREES

10.1 INDEXED SEQUENTIAL ACCESS

- Indexed sequential file structures provide a choice between two alternatives. views of a file: *Indexed:* the file can be **seen** as a set of records that is *indexed* by key;

or

- *Sequential:* the file can be accessed sequentially (physically contiguous records-no seeking), returning records in order by key.

B tree structure provides excellent *indexed* access to any individual record by key, even as records are added and deleted. Now let's suppose that we also want to use this file as part of a consequential merge. In consequential processing we want to retrieve all the records in order by key. Since the records in this file system are *entry sequenced,* the only way to retrieve them in order by key without sorting is through the index. For a file of N records, following the N pointers from the index into the entry sequenced set requires N essentially random seeks into the record file. This is a *much* less efficient process than the sequential reading of physically adjacent record.

10.2 MAINTAINING A SEQUENCE SET

Firstly the focus is on the problem of keeping a set of records in physical order by key as records are added and deleted. We refer to this ordered set of records as a *sequence set. Once we have a good way of maintaining a sequence set, we will find some way to index it as well.*

10.2.1 The Use of Blocks

One of the best ways to restrict the effects of an insertion or deletion to just a part of the sequence set involves collecting the records into *blocks. When we block records, the block becomes the basic unit of input and output.* We read and write entire blocks at once.

An example illustrates how the use of blocks can help us keep a sequence set in order. Suppose we have records that are keyed on last name and collected together so there are four records in a block we ~so include *link fields* in each block that point to the preceding block and the following block. We need these fields because; consecutive blocks are not necessarily physically adjacent.

As with B-trees, the insertion of new records into a block can cause the block to *overflow*. The overflow condition can be handled by a block-splitting process that is analogous to, but not the same as, the block-splitting process used in a B-tree. For example, Fig. 10.1(a) shows what our blocked sequence set looks like before any insertions or deletions take place. Only the forward links are shown. In Fig. 10.1(b) we have inserted a new record with the key CARTER. These insertion causes block 2 to split. The second half of what was originally block 2 is found in block 4 after the split. We just divide the records between two blocks and rearrange the links so we can still move through the file in order by key, block after block.

Deletion of records can cause a block to be less than half full and therefore to *underflow*.

In Fig. 10.1(c) we show the effects of deleting the record for DAVIS. Block 4 underflows and is then merged with its successor in *logical* sequence, which is block

q The merging process frees up block 3 for reuse.

The costs associated with the avoidance of sorting are:

Once insertions are made, our file takes up more space than an unblocked file of sorted records because of internal fragmentation within a block.

The order of the records is not necessarily *physically* Sequential throughout the file. The maximum guaranteed extent of physical sequentiality is within a block.

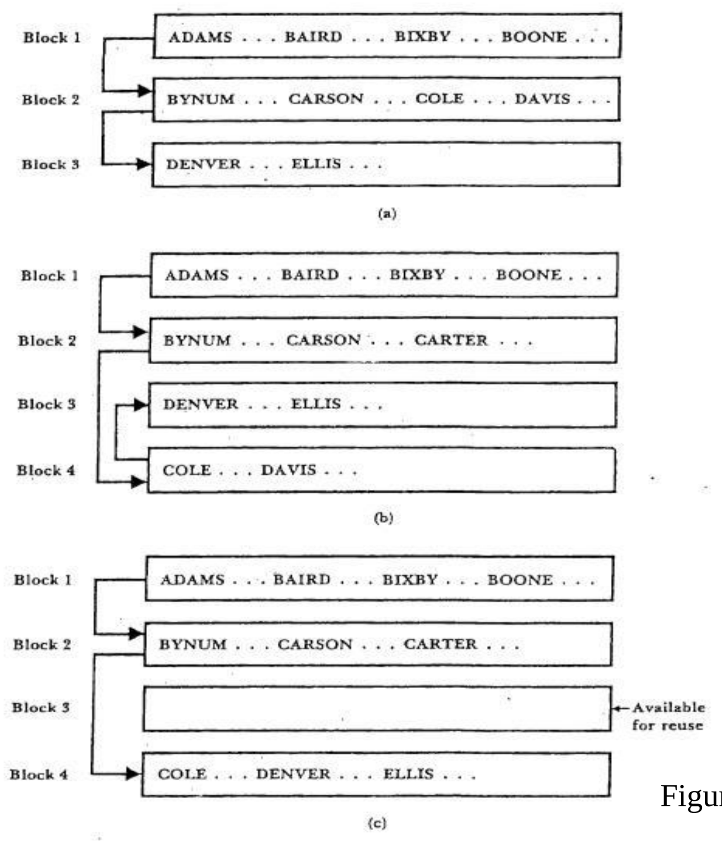

Block 1
ADAMS . . . BAIRD . . . BIXBY . . . BOONE . . .
Block 2
BYNUM . . . CARSON . . . COLE . . . DAVIS . . .
Block 3
DENVER . . . ELLIS . . .
(a)
Block 1
ADAMS . . . BAIRD . . . BIXBY . . . BOONE . . .
Block 2
BYNUM . . . CARSON . . . CARTER . . .
Block 3
DENVER . . . ELLIS . . .
Block 4
COLE . . . DAVIS . . .
(b)
Block 1
ADAMS . . . BAIRD . . . BIXBY . . . BOONE . . .
Block 2
BYNUM . . . CARSON . . . CARTER . . .
Block 3
Available for reuse
Block 4
COLE . . . DENVER . . . ELLIS . . .
(c)

Figure 10.1 Block splitting and merging.

This last point leads us to the important question of selecting a block size.

10.2.2 Choice of Block Size

Our first consideration regarding an upper bound for block size is as follows:

Consideration 1: The block size should be such that we can hold severe blocks in memory at once. For example, in performing a block split or merging, we want to be able to hold at least two blocks in memory at a time. If we are implementing two-to-three splitting to conserve disk space, we need to hold at least three blocks in memory at a time.

Consideration 2: Reading in or writing out a. block should not take very long. Even if we had an unlimited amount of memory, we would want to place an upper limit on the block size so we would not end up reading in the entire file just to get at a single record.

10.3 ADDING A SIMPLE INDEX TO THE SEQUENCE SET

Let's see whether we can find an efficient way to locate some specific block containing a particular record, given the record's key. We can view each of our blocks as containing a *range* of records, as illustrated in Fig. 10.2.

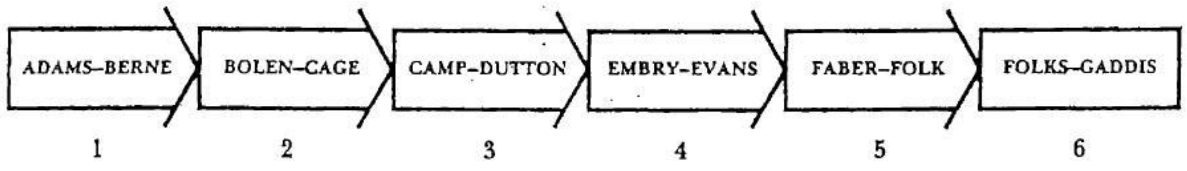
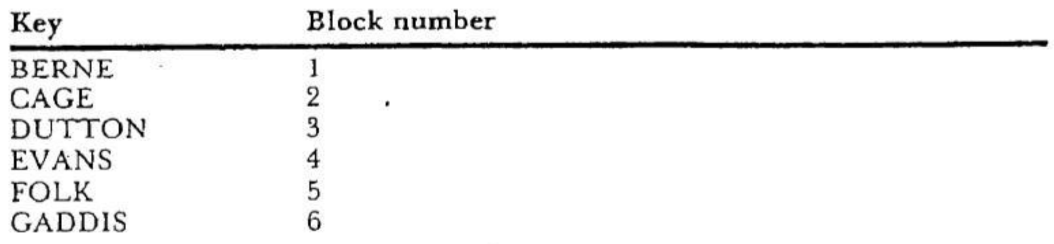


Figure 10.2 Sequence of blocks showing the range of keys in each block.

It is easy to see how we could construct a simple, single- level index for these blocks. We might choose, for example, to build an index of fixed length records that contain the key for the last record in each block, as shown in Fig. 10.3. Note that we are using the largest key in the block as the key of the whole block. The combination of this kind of index with the sequence set of blocks provides complete indexed sequential access. If we need to retrieve a specific record we consult the index and then retrieve the correct block; if we need sequential access we start at the first block and read through the linked list of blocks until we have read them all.



Key	Block number
BERNE	1
CAGE	2
DUTTON	3
EVANS	4
FOLK	5
GADDIS	6

Figure 10.3 Simple index for the sequence set illustrated in Fig. 10.2.

The requirement that the index be held in memory is important for two reasons:

- ρ Since this is a simple index we find specific records by means of a binary Search of the index. Binary searching works well if the searching takes place in memory, but, it requires too many seeks if the file is on a secondary storage device.
- σ As the blocks in the sequence set are changed through splitting, merging, and redistribution, the index has to be updated. Updating a simple, fixed- length record index of this kind works well if the index is relatively small and contained in memory. If however, the updating requires seeking to individual index records on disk, the process can become very expensive.

10.4 THE CONTENT OF THE INDEX: SEPARATORS INSTEAD OF KEYS

The purpose of the index we are building is to assist us when we are searching for a record with a Specific key. The index must guide us to the block in the sequence set that contains the record, if it exists in the sequence set at all. Given this view of the index set, we can take the very important step of recognizing that *we do not need to have keys in the index set.* Our real need is for *separators.* Figure 10.4 shows one possible set of separators for the sequence set in Fig. 10.2.

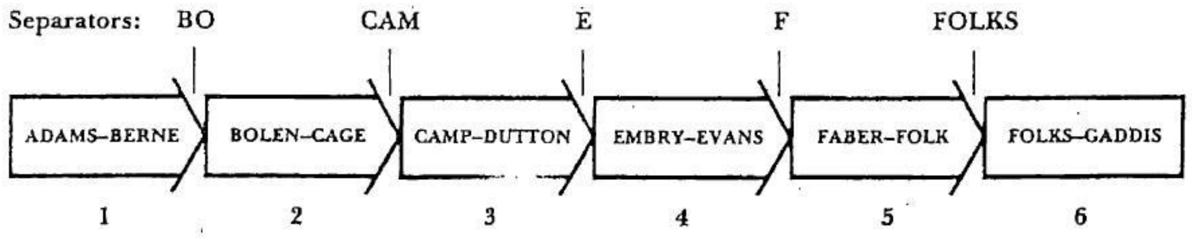


Figure 10.4 Separators between blocks in the sequence set.

Note that there are many potential separators capable of `distinguishing between two blocks. For example, all of the strings shown between blocks 3 and 4 in Fig. 10.5 are capable of guiding us in our choice between the blocks as we search for a particular key.

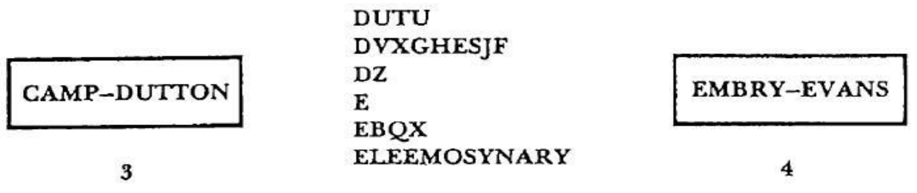


Figure 10.5 A list of potential separators.

If a-string comparison between the key and any of these separators shows that the key precedes the separator, we look for the key in block 3. If the key follows the Separator, we look in block 4.

If we are willing to treat the separators as variable- length entities within our index structure (we talk about how to do this later), we can save space by placing the *shortest separator* in the index structure. Consequently, we use E as the separator to guide our choice between blocks 3 and 4. Note that there is not always a unique shortest separator. For example, BK, BN, and BO are separators that. are all the same length and are equally effective as separators between blocks 1 and 2 in Fig. 10.4. We choose BO and all of the other separators contained in Fig. 10.4 by using the logic embodied in the C++ function shown in Fig. 10.6. We must decide to retrieve the block to the right of the separator or the one to the left of the separator according to the following rule:

Relation ofsearch key and separator	*Decision*
Key < Separator	Go left
Key = separator	Go right
Key > separator	Go right

```
void FindSeparator (char * key1, char * key2, char * sep)
{// key1, key2, and sep point to the beginning of char arrays
   while (1) // loop until break
   {
      *sep = *key2; sep ++; //move the current character into sep
      if (*key2 != *key1) break; // stop when a difference is found
      if (*key2 == 0) break; // stop at end of key2
      key1 ++; key2 ++; // move to the next character of keys
   }
   *sep = 0; // null terminate the separator string
}
```

Figure 10.6 C++ function to find a shortest separator.

10.5 THE SIMPLE PREFIX B+ TREE

Figure 10.7 shows how we can form the separators identified in Fig. 10.4 into a B-tree index of the sequence set blocks. The B-tree index is called the *index set.* Taken together with the sequence set, it forms a file structure called a *simple prefix B+ tree.*

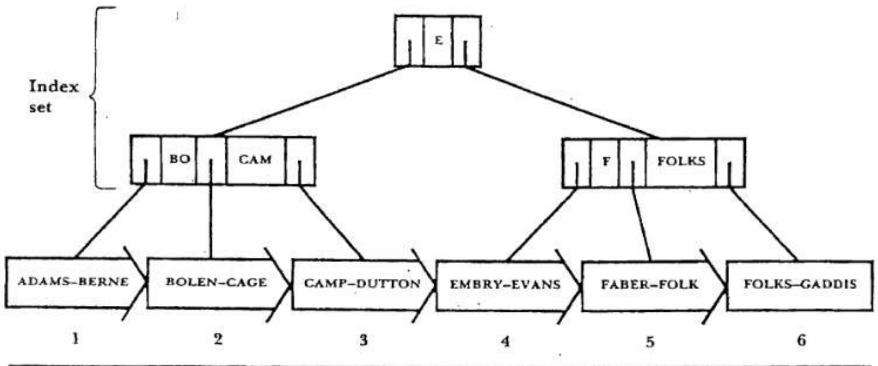


Figure 10.7 A B-tree index set for the sequence set, forming a simple prefix B^+ tree.

Note that the index set is a B-tree, and a node containing N separators branches to N+ 1 children. If we are searching for the record with the key EMERY, we start at the root of the index set, comparing EMBRY with the separator E. Since EMERY comes after E, we branch to the right, retrieving the node containing the separators F and FOLKS. Since EMERY comes before even the first of these separators, we follow the branch that is to the left of the F separator, which leads us to block 4, the correct block in the sequence set.

10.6 SIMPLE PREFIX B+ TREE MAINTENANCE

10.6.1 Changes Localize d to Single Blocks in the Sequence Set

Let's suppose that we want to delete the records for EMBRY and FOLKS and that neither of these deletions results in any merging or redistribution within the sequence set. Since there is no merging or redistribution, the effect of these deletions on the *sequence set* is limited to changes *within* blocks 4 and 6. The record that was formerly the second record in block 4 (let's say that its key is ERVIN) is now the first record. Similarly, the former second record in `block 6 (we assume it has a key of FROST) now starts that block. These changes can be seen in Fig. 10.8.

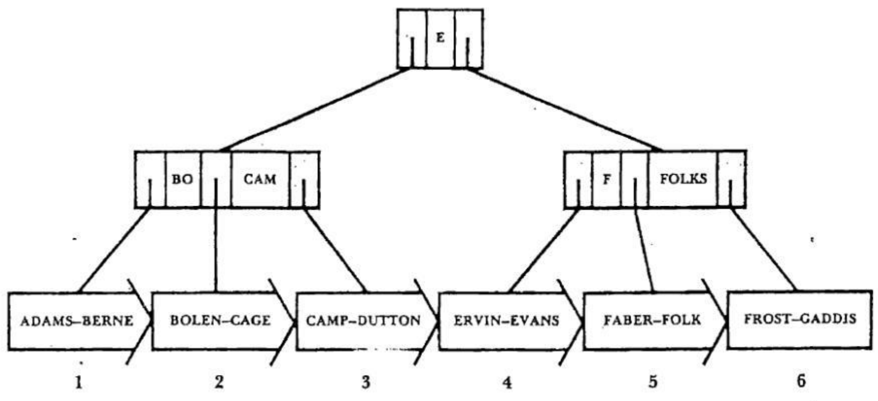


Figure 10.8 The deletion of the EMBRY and FOLKS records from the sequence set leaves the index set unchanged.

Since the number of sequence set blocks is unchanged and since no records are moved between blocks, the index set can also remain unchanged. This is easy to see in the case of the EMBRY deletion: E is still a perfectly good separator for sequence set blocks 3 and 4, so there is no reason to change it in the index set.

The effect of inserting into the sequence set new records that do not cause block splitting is much the same as the effect of these deletions that do not result in merging. The index $ct remains unchanged. Suppose for example, that we insert a record for EATON. Following the path indicated by the separators in the index set, we find that we will insert the new record into block 4 of the sequence set. The new record becomes the first record in block 4, but no change in the index set is necessary.

10.6.2 Changes Involving Multiple Blocks in the Sequence Set

We begin with an insertion into the sequence set shown in Fig. 10.8. Specifically, let's assume that there is an insertion into the first block and that this insertion causes the block to split. A new block (block 7) is brought in to hold the second half of what was originally the first block. This new block is linked into the correct position in the sequence set, following block I and preceding block 2 (these are the *physical* block numbers).These changes to the Sequence set are illustrated in Fig. 10.9.

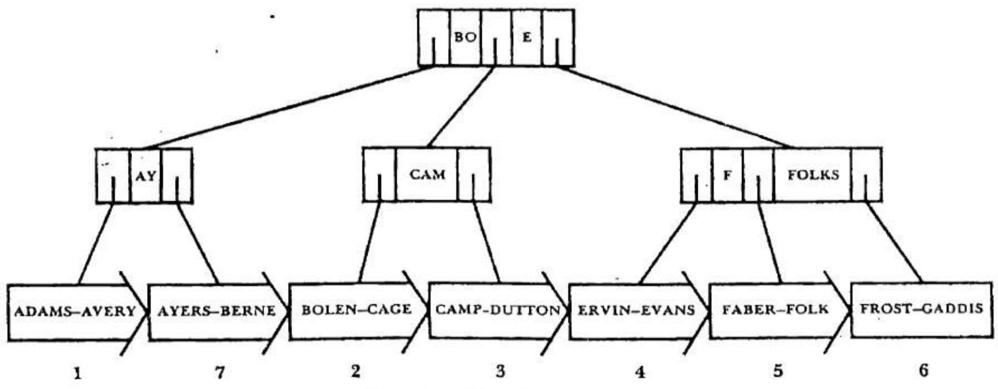


Figure 10.9 An insertion into block 1 causes a split and the consequent addition of block 7. The addition of a block in the sequence set requires a new separator in the index set. Insertion of the AY separator into the node containing BO and CAM causes a node split in the index set B-tree and consequent promotion of BO to the root.

Now let's suppose we delete a record from block 2 of the sequence set and that this, causes an underflow condition and consequent merging of blocks 2 and 3, Once the merging is complete, block 3 is no longer needed in the sequence set, and the separator that once distinguished between blocks 2 and 3 must be removed from the index set. Removing this separator, CAM, causes an underflow in an index set node. Consequently, there is another merging, this time in the index set, that results in the demotion of the BO separator from the root, bringing it back down into a node with the AY separator. Once these changes are complete, the simple prefix B+ tree has. the structure illustrated in Fig. 10.10.

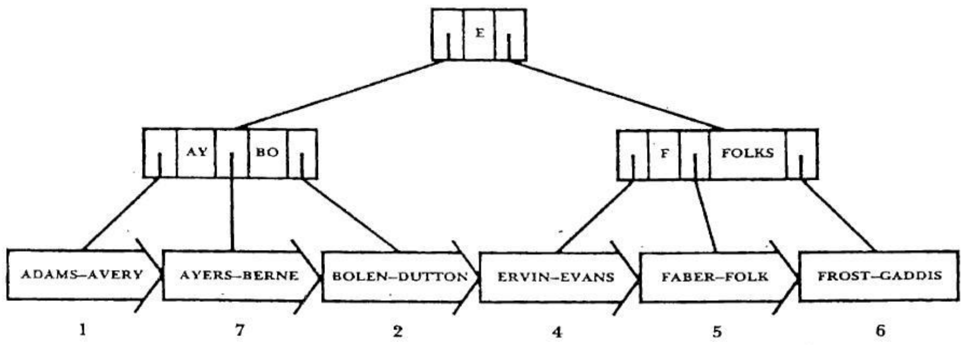


Figure 10.10 A deletion from block 2 causes underflow and the consequent merging of blocks 2 and 3. After the merging, block 3 is no longer needed and can be placed on an avail list. Consequently, the separator CAM is no longer needed. Removing CAM from its node in the index set forces a merging of index set nodes, bringing BO back down from the root.

Record insertion and deletion *always* take place in the sequence set, since that is where the records are. If splitting, merging, or redistribution is necessary, perform the operation just as you would *if there were no index set at all.* Then, after the record operations in the sequence set are complete, make changes as necessary in the index set:

- o If blocks are split in the sequence set, a new separator must be inserted into the index set;
- π If blocks are merged in the sequence set, a separator must be removed from the index set; and
- θ If records arc redistributed between blocks in the sequence set, the value of a separator in the index set must be changed.

10.7 INDEX SET BLOCK SIZE

The physical size of a node for the index Set is usually the same as the physical size of a block in the sequence set. There are a number of reasons for using a common block size for the index and Sequence sets:

- The block size for the sequence set is usually chosen because there is a good fit among this block size, the characteristics of the disk drive, and the amount of memory available the choice of an index set block Size is governed by consideration of the same factors; therefore, the block size that is best for the sequence set is usually best for the index set.
- The index set blocks and sequence set blocks are often mingled within the same file to avoid seeking between two separate files while accessing the simple prefix B+ tree. Use of one file for both kinds of blocks is simpler if the block sizes are the same.

10.8 INTERNAL STRUCTURE OF INDEX SET BLOCKS: A VARIABLE–ORDER B-TREE

We want each index set block to hold variable number of variable length separators. We therefore need to structure the block so it can support a binary search, despite the fact that the separators are of variable length. For example, suppose we are going to place the following set of separators into an index block: As, Ba, Bro, C, Ch, Cra, Pele, Edi, Err, Fa, Fe

We could merge these separators and build an index for them, as shown in Fig. 10.11

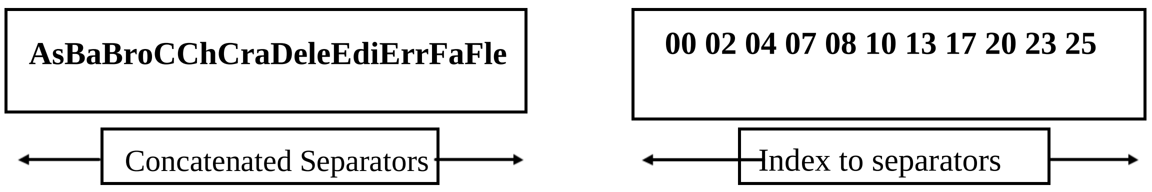


Figure 10.11 Variable length separators and corresponding index.

The index set block needs some way to store references to its children, to the blocks descending from it in the next lower level of the tree. We assume that the references are made in terms of a relative block number (RBN). If there are N separators within a block, the block has N+ 1 children and therefore needs Space to store N+ I RBNs in addition to the separators and the index to the separators.

There are many ways to combine the list of separato rs, the index to separators, and the list of RBNs into a single index set block. One possible approach is illustrated in Fig. 10.12. In addition to the vector of separators, the index to these separators, and the list of associated block numbers, this block structure includes:

Separator count*:* we need this to help us find the middle element in the index to the separators so we can begin our binary search.

Total length of separators*:* the list of merged separators varies in length from block to block. Since the index to the separators begins at the end of this variable- length list, we need to know how long the list is so we can find the beginning of our index.

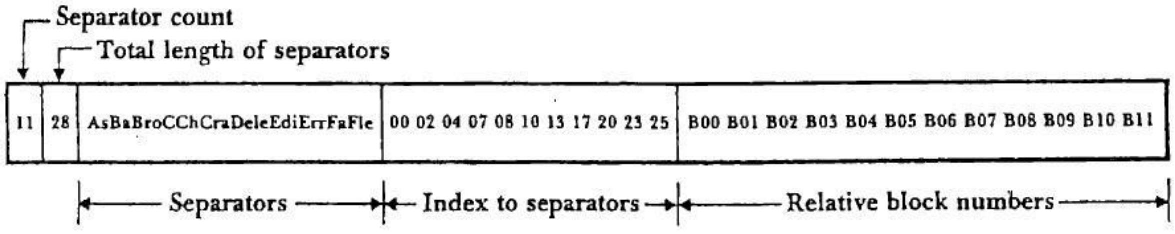


Figure 10.12 Structure of an index set block.

Let's suppose, once again, that we are looking for a record with the key "Beck" and that the search has brought us to the index set block pictured in Fig. 10.12. The total length of the separators and the separator count allow us to find the beginning, the end, and consequently the middle of the index to the separators. We perform a binary Search of the separators through this index, finally concluding that the key "Beck" falls between the separators "Ba" and *" Bro"* . *Conceptually,* the relation between the keys and the RBNs is illustrated in Fig. 10.13.

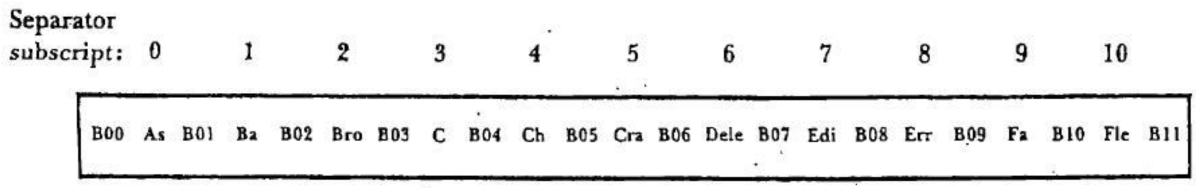

Separator subscript: 0 1 2 3 4 5 6 7 8 9 10
B00 As B01 Ba B02 Bro B03 C B04 Ch B05 Cra B06 Dele B07 Edi B08 Err B09 Fa B10 Fle B11

Figure 10.13 Conceptual relationship of separators and relative block numbers.

As Fig. 10.13 makes clear, discovering that the key falls between." Ba" and " Bro" allow us to decide that the *"next* block we need to retrieve has the RBN stored in the B02 position of the RBN vector. This next block could be another index set block and thus another block of the road map, or it could be the sequence set block that we are looking for. In either case, the quantity and arrangement of information in the current index set block is sufficient to let us conduct our binary search *within* the index block and proceed to the next block in the simple prefix B+ tree.

10.9 LOADING A SIMPLE PREFIX B+ TREE

It is possible to *conceive* of simple prefix B+ tree as a sequence set with an added index, but one can also *build* them the other way as mentioned below.

We can begin by sorting the records that are to be loaded. Then we can guarantee that the next record we encounter is the next record we need to load. Working from a sorted file, we can place the records into sequence set blocks, one by one, starting a new block when the one we are working with fills up. As we make the transition between two sequence set blocks, we can determine the shortest Separator for the blocks we can collect these separators into an index set block that we build and hold in memory until it is full.

To develop an example of how this works let's assume that we have sets of records associated with terms that are being compiled for a book index. The records might consist of a list of the occurrences of each term.

In Fig. 10.14 we show four sequence set blocks that have been written out to the disk and one index set block that has been built in memory from the shortest separators derived from the sequence set block keys. As you can See, the next sequence set block consists of a set of terms ranging from CATCH through CHECK, and therefore the next separator is CAT. Let's suppose that the index set block is now full.-We write it out to disk. Now what do we do with the separator CAT?

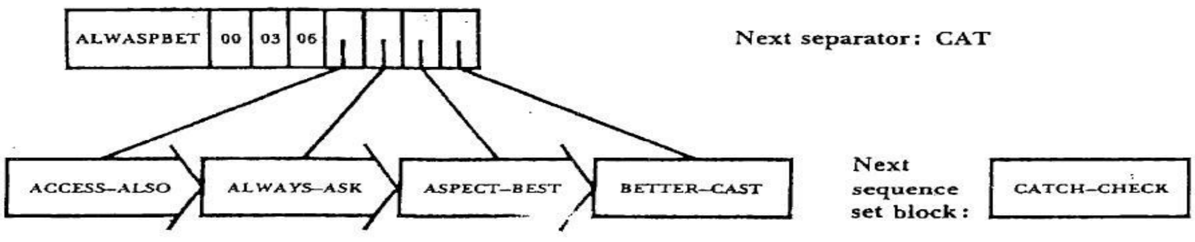


Figure 10.14 Formation of the first index set block as the sequence set is loaded.

Clearly, we need to start a new index block. However, we cannot place CAT into another index block at the same level as the one containing the separators ALW, ASP, and BET because we cannot have two blocks at the same level without having a parent block. Instead, we promote the CAT separator to a higher- level block. However, the` higher- level block cannot point directly to the sequence set, it must point to the lower- level index blocks. This means that we will now be building *two levels* of the index set in memory as we build the sequence set. Figure 10.15 illustrates this working-on two-level phenomenon: the addition of the CAT separator requires us to Start a new, root- level index block as well as a lower-level index block.

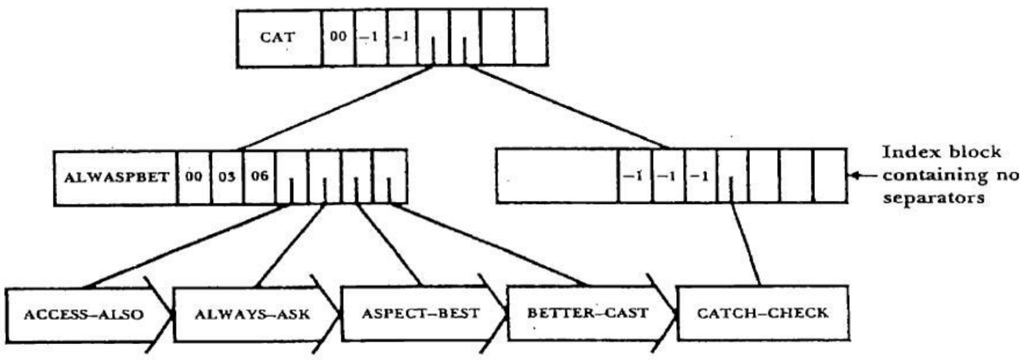


Figure 10.15 Simultaneous building of two index set levels as the sequence set continues to grow.

Figure 10.16 shows what the index looks like after even more sequence set blocks are added. As you can see, the lower- level index block that contained no separators when we added CAT to the root has now filled up.

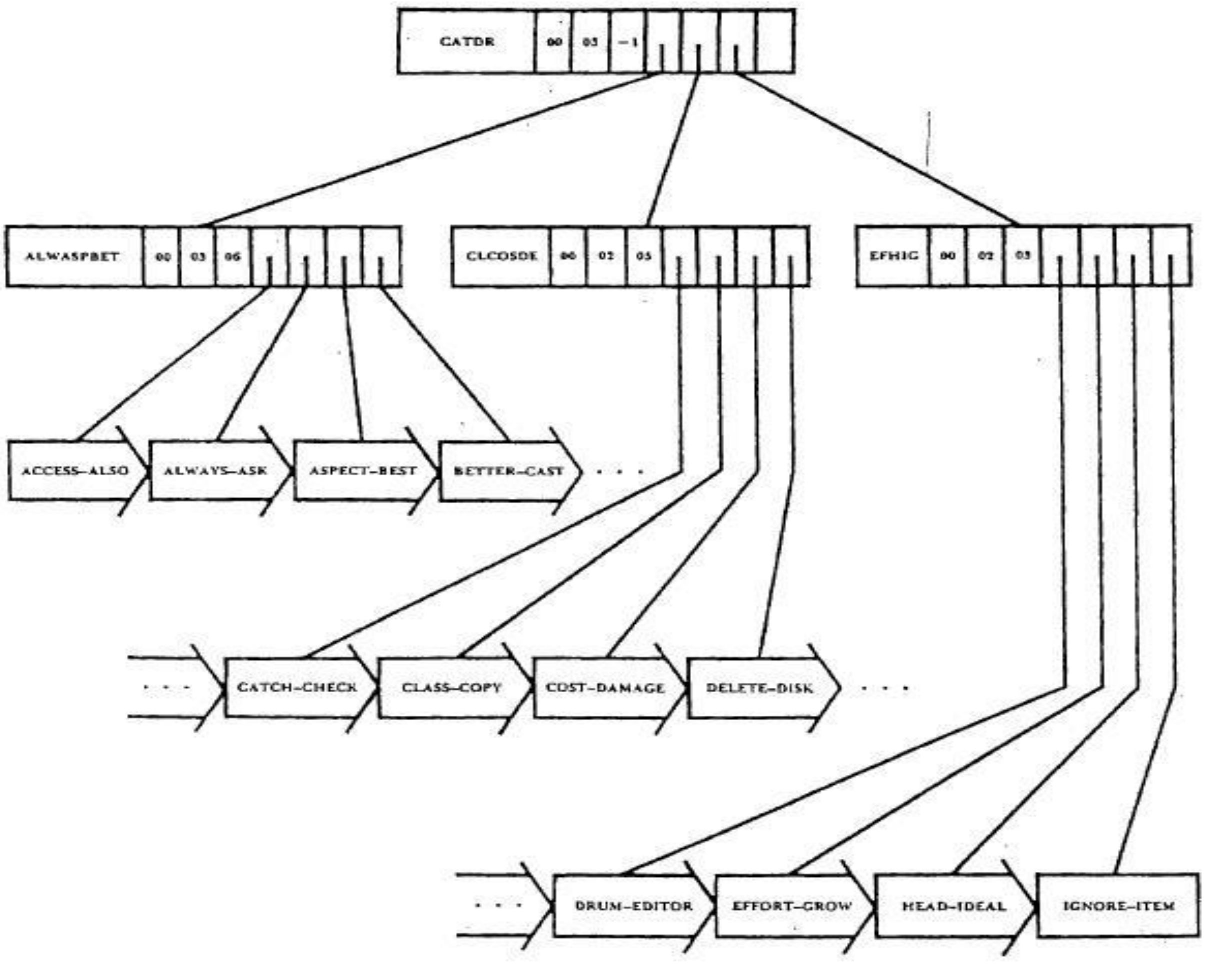


Figure 10.16 Continued growth of index set built up from the sequence set.

The principal advantage is that the loading process goes more quickly because

- The output can be written sequentially;
- We make only one pass over the data, rather than the many passes associated with random order insertions; and
- No blocks need to be reorganized as we proceed.

10.10 B+ TREES

The difference between a simple prefix B+ tree and a plain B+ tree is that the latter structure does not involve the use of prefixes as separators. Instead, the separators in the index set are simply copies of the actual keys. Contrast the index set block shown in Fig.10.17, which illustrates the initial loading steps for a B+ tree, with the index block that is illustrated in Fig. 10.14, where we are building a Simple prefix B+ tree.

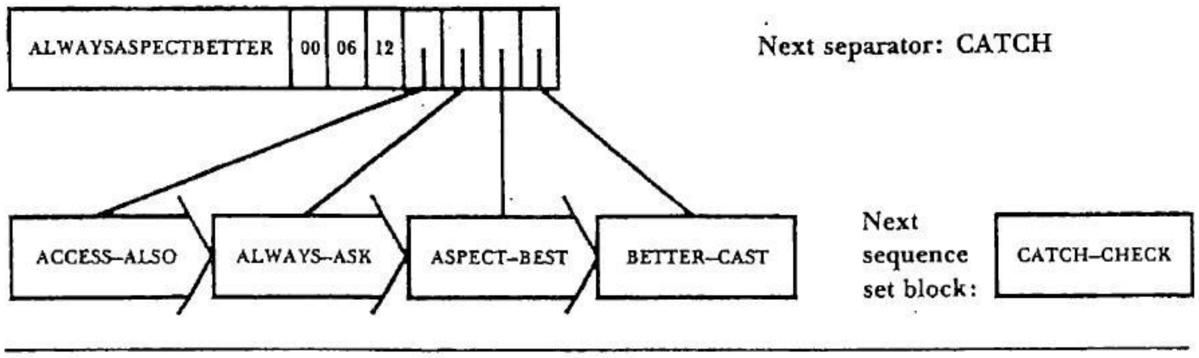


Figure 10.17 Formation of the first index set block in a B⁺ tree without the use of shortest separators.

There are, however, at least two factors that might give favour to using a B+ tree that uses full copies of keys as separators.

- The reason for using shortest separators is to pack more of them into an index set block. This imp lies, ineluctably, the use of variable- length fields within the index set blocks. For some applications the cost of the extra overhead required to maintain and use this variable- length structure outweighs the benefits of shorter separators. In these cases one might choose to build a Straightforward B+ tree using fixed- length copies of .the keys fro m the sequence set as separators.
- Some key sets do not show much compression when the Simple prefix method is used to produce separators. For example, suppose the keys consist of large, consecutive alphanumeric Sequences such as 34C18K756,.34C18K757, 34C18K758, and so on. In this case, to enjoy appreciable compression, we need to use compression techniques that remove redundancy from the *front* of the key.

10.11 B-TREES, B+ TREES, AND SIMPLE PREFIX B+ TREES IN PRESPECTIVE

B- trees, B+ trees, and simple prefix B+ trees or, not a panacea. However, they do have broad applicability, particularly for situations that require the ability to access a large file in order by key, and through an index all three of these tools share the following characteristics:

- They are all paged index structures, which means that they bring entire blocks of information into memory at once. As a consequence, it is possible to choose between a great many alternatives (for example, the keys for hundreds of thousands of records) with just a few seek out to disk storage the shape of these trees tends to be broad and shallow.
- All three approaches maintain height- balanced trees. The trees do not grow in an uneven way, which would result in some potentially long searches for certain keys.
- In all cases the trees grow from the bottom up. Balance is maintained through block splitting, merging, and redistribution.
- With all three structures it is possible to obtain greater storage efficiency through the use of two-to-three splitting and of redistribution in place of block splitting when possible.
- All three approaches can be implemented as virtual tree structures in which the most recently used blocks are held in memory.
- Any of these approaches can be adapted for use with variable- length records using structures inside a block similar to those outlined in this chapter.
- For all of this similarity, there are some important differences. These differences are brought into focus through a review of the strengths and unique characteristics of each of these file structures.

B Trees

The B- trees are multilevel indexes to data files that are entry sequenced this is the simplest type of B-tree to implement and is a very efficient representation for most cases. The strengths of this approach are the simplicity of implementation, the inherent efficiency of indexing, and a maximization of the breadth of the B- tree. The major weakness of this strategy1s the lack of organization of the data file, resulting in an excessive amount of seeking for sequential access.

B+ Trees

The primary difference between the B+ tree and the B- tree is that in the B+ tree all the key and record information is contained in a linked set of blocks known as the *sequence set.* The key and record information is *not* in the upper- level, tree like portion of the B+ tree. Indexed access to this sequence set is provided through a conceptually separate structure called the *index set.*

There are three significant advantages that the B+ tree structure provides over the B- tree:

τ The sequence set can be processed in a truly linear, sequential way, providing efficient access to records in order by key.

υ The index is built with a single key or separator per block of data records instead of one key per data record.

ϖ The size of the lowest- level index is reduced by the blocking factor of the data file. Since there are fewer keys, the index is smaller and hence shallower.

Simple Prefix B+ Trees

The Simple prefix B+ tree builds on the advantage of B+ tree by making the separators in the index set *smaller* than the keys in the sequence set, rather than just using copies of these keys. If the separators are smaller, we can fit more of them into a block to obtain a higher branching factor out of the block. In a sense, the simple prefix B+ tree takes one of the strongest features of the B+ tree one step farther.

The price we have to pay to obtain this separator compression and consequent increase in branching factor is that we must use an index block structure that supports variable-length fields.

******END OF MODULE- 4********

MODULE – 5

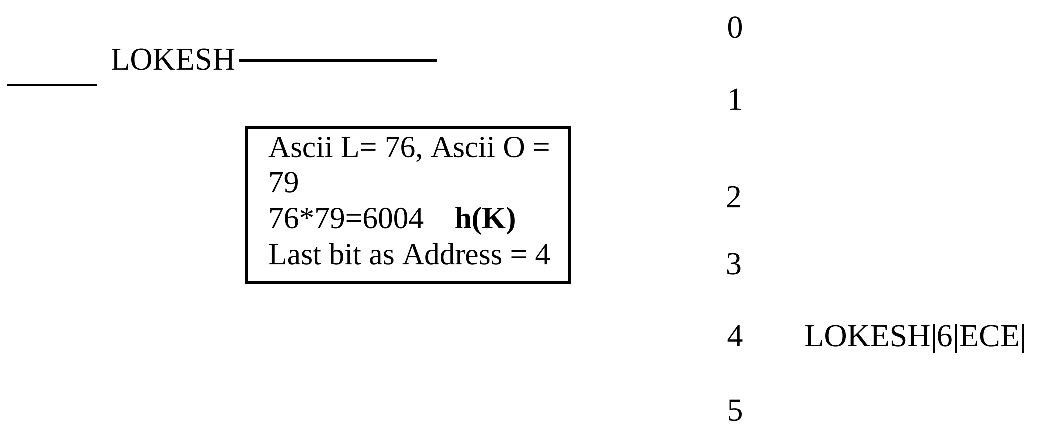
CHAPTER - 11: HASHING

11.1 Introduction:

Sequential access efficiency is O (N). B trees/ B+ trees efficiency is O(log$_K$ N). Both are dependent on 'N' where N is the number of records in the file i.e file size. So there was a need for a technique whose efficiency is independent of the size of the file. This lead to Hashing which provides an efficiency of O(1).

A Hash function is like a black box that produces an address every time a key is dropped. The process of producing an address when a key is dropped to hash function is called Hashing. Formally hash function is given by h(K) that transforms a key 'K' into an address. The resulting address is used to store and retrieve the record.

Example: Consider a key LOKESH which is dropped in to a hash function which does the following computation to transform the key to its address. Assume the address space is 10. i.e in the range 0 to 9.



In indexing we associate a key with relative record number (RRN). Hashing is also similar to indexing in this way. i.e associating a key to a address (relative record number). But hashing differs from indexing in two major ways. They are:

- With hashing the address generated appear to be random. In indexing association of key with RRN is sequential whereas in hashing this association is random.
- In hashing two different keys may be associated (Hashed) to same address. This is called collision. Whereas in indexing there is no collision. i.e there is no chance of two different keys being associated to same RRN.

The ideal solution is to find or devise a transformation algorithm that avoids collision altogether. Such an algorithm is called a perfect hashing algorithm. It is near to impossible to find a perfect hashing algorithm. So a more practical solution is to reduce the number of collision to an acceptable level. Some of the ways to reduce the collusion to acceptable level are:

- Spread out the records.
- Use extra memory.
- Put more than one record at a single address.

Spread out the records: Collision occur when two or more records compete for the same address. If we could find hashing algorithm that distributes the records fairly randomly among the available addresses, then we would not have large number of records clustering around certain address.

Use extra me mory: It is easier to find a hash algorithm that avoids collisions if we have only a few records to distribute among many addresses than if we have about the same number of records as addresses. Example chances of collisions are less when 10 records needs to be distributed among 100 address space. Whereas chances of collision is more when 10 records needs to be distributed among 10 address space. The obvious disadvantage is wastage of space.

Put more than one record at a single address: Create a file in such a way that it can store several records at every address. Example, if each record is 80 bytes long and we create a file with 512 byte physical records, we can store up to six records at each address. This is called as bucket technique.

11.2 A Simple Hashing Algorithm

This algorithm has three steps:

- Represent the key in numerical form.
- Fold and add.
- Divide by a prime number and use the reminder as the address.

Step 1: Put more than one record at a single address.

If the key is string of characters take ASCII code of each character and use it to form a number. Otherwise no need to do this step.

Example if the key is LOWELL we will have to take ASCII code for each charaters to form a number. If the key is 550 then no need to do this step. We will consider LOWELL as the key.

LOWELL – 76 79 87 69 76 76 32 32 32 32 32 32

32 is ASCII code of blank space. We are padding the key with blank space to make it 12 characters long in size.

Step 2: Fold and add.

In this step Folding and adding means chopping off pieces of the number and add them together. Here we chop off pieces with two ASCII numbers each as shown below:

7679 | 8769 | 7676 | 3232 | 3232 | 3232 -------------- Folding is done

On some microcomputers, for example, integer values that exceed 32767 causes overflow errors or become negative. So addition of the above integer values results in 33820 causing an overflow error. Consequently we need to make sure that each successive sum is less than 32767. This is done by first identifying the largest single value we will ever add in summation and making sure after each addition the intermediate result differs from 32767 by that amount.

Assuming that keys contain only blanks and upper case alphabets, so the largest addend is 9090 corresponding to ZZ. Suppose we choose 19937 as our largest allowable intermediate result. This differs from 32767 by much more than 9090, so no new addition will cause overflow.

Ensuring of intermediate result not exceeding 19937 is done by using mod operator, which returns the reminder when one integer is divided by another.

7679 + 8769 = 16448 => 16448 mod 19937 => 16448
16448 + 7676 = 24124 => 24124 mod 19937 => 4187
4187 + 3232 = 7419 => 7419 mod 19937 => 7419
7419 + 3232 = 10651 => 10651 mod 19937 => 10651
10651 + 3232 = 13883 => 13883 mod 19937 => 13883

The number 13883 is the result of the fold and add operation.

Step 3: Divide by the Size of the Address Space.

In this step the number produced by the previous step is cut down so that it falls in the within the range of addresses of records in the file. This is done by dividing the number by address size of the file. The remainder will be home address of record. It is given as shown below

a = s mod n

If we decide to have 100 addresses i.e 0-99 than a= 13883 mod 100, which is equal to 83. So hash address of LOWELL will be 83.

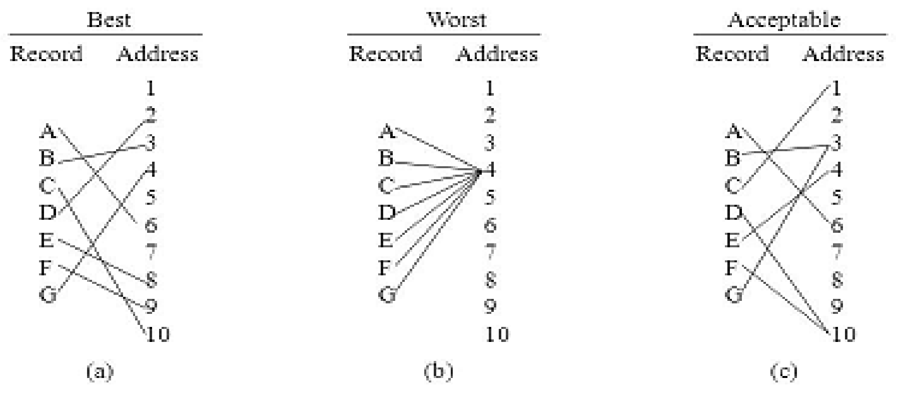
11.3 Hashing Functions & Record Distribution:

The figure below shows three different distributions of seven records among nine addresses.

If the hash function distributes seven records in ten addresses without any collision then it is called as **uniform distribution** as shown in figure '**a**'.

If all the seven records are hashed to the same home address then it is called as **worst distribution** as shown in the figure '**b'**

Figure '**c**' illustrates a distribution in which the records are somewhat spread out, but with a few collisions. This is called as **Acceptable distribution**.



<Figure 11.3> Different distributions. (a) Uniform distribution(Best) (b) Worst case (c) Randomly distribution (Acceptable)

11.3.1 Some Other Hashing Methods

Some methods those are potentially better than random. The simple hashing algorithm explained in section 7.2 which has three steps are designed to take advantage of natural ordering among the keys. The next two methods can be tried when for some reasons, the better than random methods do not work.

- **Square the key and take the mid (Mid Square method):** This method involves treating the key as single large number, squaring the number and extracting whatever number of digits are required from the middle of the result.
  - For example: consider the key 453, its square is $(453)^2$ = 205209. Extracting the middle two digits yields a number 52 which is between 0 – 99.
- **Radix Transformation:** This method involves converting the key from one base system to another base system, then dividing the result with maximum address and taking the reminder**.**

  For example: If the hash address range is 0 – 99 and key is $(453)_{10.}$. Converting this number to base 11 system results in $(382)_{11}$. Then 382 mod 99 = 85. So 85 is the hash address.

11.3.2 Predicting the Distribution of Records.

7.3.3 Predicting Collision for a Full File.

7.4 How Much Extra Memory should be Used.

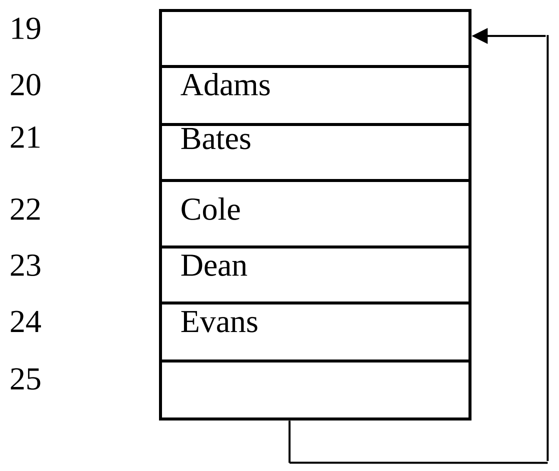
7.5 Collision Resolution by Progressive Overflow

- **Progressive overflow** is a collision resolution technique which places overflow records at the first empty address after the home address; if the file is full the search comes back to where it began. Only then it is clear that the file is full and no empty address is available.
- With progressive overflow, a sequential search is performed beginning at the home address and if end of the address is reached, then wrap around is done and the search continuous from the beginning address in the file.
- The search is continued until the desired key or a blank record is found.
- If the file is full the search comes back to where it began. Only then it is clear that the record is not in the file.
- Progressive overflow is also referred to as *linear probing.*

Consider the table given below:

Key	**Home address**
Adams	20
Bates	21
Cole	21
Dean	22
Evans	20

Table 1.



Key	Home address	**Actual Address**	**Search Length**
Adams	20	**20**	**1**
Bates	21	**21**	**1**
Cole	21	**22**	**2**
Dean	22	**23**	**2**
Evans	20	**24**	**5**

Average search length is $\frac{1+1+2+2+5}{5} = 2.2$

Making Deletions

Now suppose record Dean is deleted which makes the address 23 empty. When the key Evans is searched it starts from home address of the i.e 20 and when it reaches address 23, which is empty the search stops and displays that key not present. But actually the key is present in the file. So deletion of records leads to this type of problems. Using the concept of Tombstone technique this limitation can be overcome. The figure below depicts the above mentioned scenario.

	19
Adams	20
Bates	21
Cole	22
######	23
Evans	24
	25

In tombstone technique whenever a record is deleted that address space is marked by a special character. So whenever a special character is encountered the search continuous to next address without stopping at that point only. Further these marked addresses can be reused or reclaimed to store the new record.

Implications of Tombstone for Insertions.

Now suppose a new record with key Evans needs to be stored. When the key Evans is dropped to hash function it produces an address of 20. Since address 20 is already occupied by Adams according to progressive overflow technique the search for empty address slot starts and when it reaches address 23 the New Evans is inserted. But the problem is already there is a record for Evans i.e duplicate records are inserted which is certainly not acceptable. So to avoid we have to search the entire address space if already a key with same name is present or not. Only then the new record can be inserted. This in turn slows down the insertion operation.

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.

Address	Counter	Records		
19	0			
20	2	Adams	Evans	
21	2	Bates	Cole	

22	1	Dean		
	0			

- There should be a counter to keep track of how many records are already stored in any given address. The figure above represents storage of records with bucket size of 3.

11.8 Other Collision Resolution Techniques:

11.8.1 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.)
- If there is collision at the hash address produced by $h_1(k)$ = Say **'X',** then the key is dropped to second hash function to produce address **'C'**.
- The new record is stored at the address **'X+C'.**
- A collision resolution scheme which applies a second hash function to keys which collide, to determine a probing distance 'C'.
- The use of double hashing will reduce the average number of probes required to find a record.

11.8.2 Chained Progressive Overflow

- Chained progressive overflow forms a linked list, or chain, of synonyms.
- Each home address contains a number indicating the location of the next record with the same home address.
- The next record in turn contains a pointer to the other record with the same home address.
- This is shown in the figure below: In the figure below Admans contain the pointer to Cole which is synonym. Then Bates contain pointer to Dean which are again synonym. (Consider the below given Table)

Key	**Home Address**
Adams	20
Bates	21
Cole	20
Dean	21
Evans	24
Flint	20

The figure below represents the chained progressive overflow technique.

Home Address	Actual Address	Records	Address of next Synonym	Search Length
	19	.	.	.
20	20	Adams	**22**	**1**
21	21	Bates	**23**	**1**
20	22	Cole	**25**	**2**
21	23	Dean	**-1**	**2**
24	24	Evans	**-1**	**1**
20	25	Flint	**-1**	**3**
	26		.	.
			.	.

Now suppose if the Dean's home address was 22 instead of 21. By the time Dean is loaded, address 22 is already occupied by Cole, so Dean ends up at address 23. Does this mean that Cole's pointer should point to 23 (Dean's actual address) or to 25 (the address of Cole's synonym Flint)? If the pointer is 25, the linked list joining Adams, Cole, Flint is kept intact, but Dean is lost. If the pointer is 23 Flint is lost.

The problem here is that a certain address (22) that should be occupied by a home record (Dean is occupied by a different record. Solution to this type of problem are:

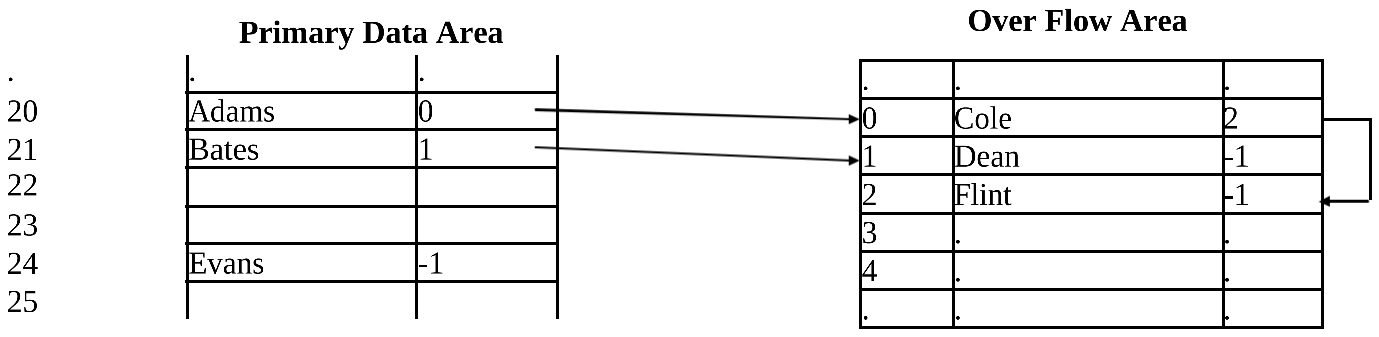
- Two pass loading.
- Chaining with a separate overflow area.

11.8.2 Two Pass Loading

- As the name implies, involves loading a hash file in two passes. On the first pass, only home records are loaded.
- All records that are not home records are kept in separate file.
- On the second pass, each overflow record is loaded and stored in one of the free addresses according to whatever collision resolution technique is being used.

11.8.3 Chaining with a separate overflow area.

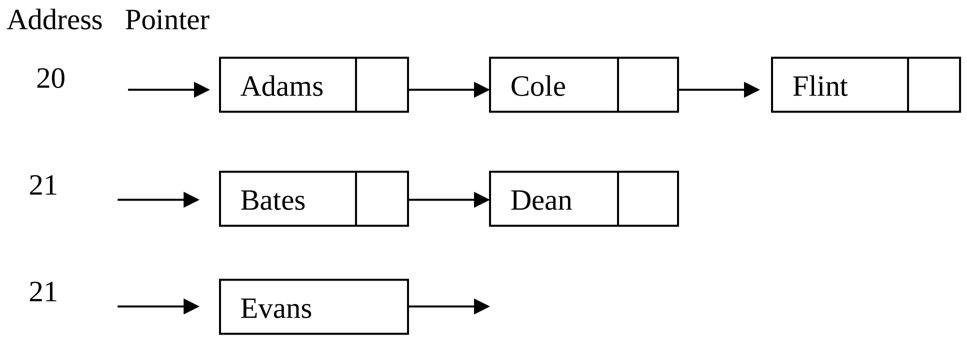
- One way to keep overflow records from occupying home addresses where they should not be is to move them all to a separate overflow area.
- The set of home addresses is called prime data area, and the set of overflow addresses is called the overflow area.
- With respect to the previous example the records for Cole, Dean and Flint are stored in a separate overflow area as shown in the figure below.
- Whenever a new record is added if its home address is empty, it is stored in primary storage area. Otherwise it is moved to overflow area, where it is added to a linked list that starts at home address.



11.8.4 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*.
- In scatter table records are not stored at the hash address.
- Only pointer to the record is stored at hash address.
- Scatter tables are similar to index but they are searched by hashing rather than some other method which is used in case of indexing.
- Main advantage of scatter tables is they support use of variable length records.

Key	**Home Address**
Adams	20
Bates	21
Cole	20
Dean	21
Evans	22
Flint	20



11.9 PATTERN OF RECORDS

If we have some information about what records get accessed most often, we can optimize their location so that these records will have short search length. By doing this, we try to decrease the effective average search length even if the nominal average search length remains the same. This principle is related to the one used in Huffman encoding.

CHAPTER – 12: EXTENDIBLE HASHING

12.1 Introduction

12.2 How Extendible Hashing Works

Tries

The key idea behind extendible hashing is to combine conventional hashing with another retrieval approach called the trie. Tries are also sometimes referred to as *radix searching* because the branching factor of the search tree is equal to the number of alternative symbols (the radix of the alphabet) that can occur in each position of the key.

Suppose we want to build a trie that stores the keys *able*, *abrahms, Adams, anderson, andrews,* and *Baird.* A schematic form of the trie is shown in Fig. 12.1. As you can see, the searching proceeds letter by letter through the key. Because there are twenty-six symbols in the alphabet, the potential branching factor at every node of the search is twenty-Six.

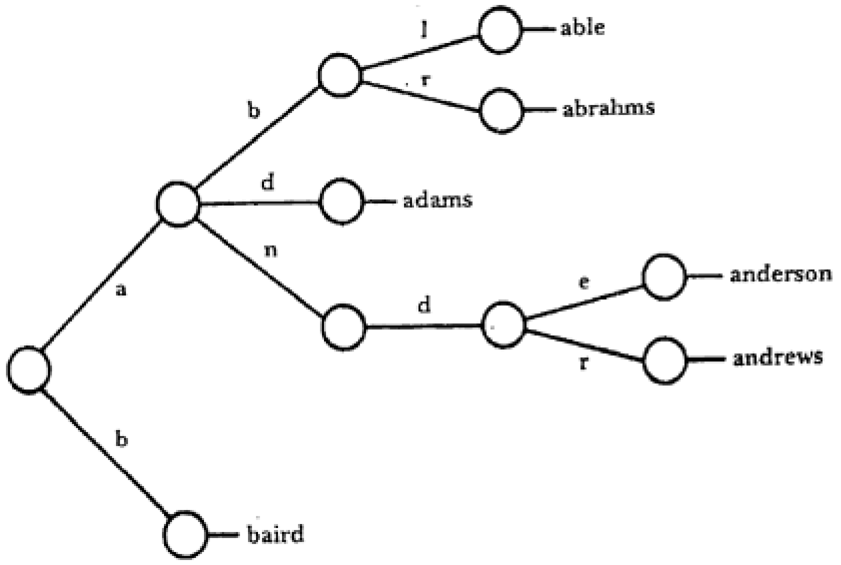


Figure 12.1

Notice that in searching a trie we sometimes use only a portion of the key. We use more of the key as we need more information to complete the search. This use-more-as-we-need- more capability is fundamental to the structure of extendible hashing.

12.2.2 Turning the Trie into a Directory

We use tries with a radix of 2 in our approach to extendible hashing: search decisions are made on a bit-by-bit basis. We will not work in terms of individual keys but in terms of *buckets* containing keys.

Suppose we have bucket A containing keys that, when hashed, have hash addresses that begin with the bits *01*. Bucket B contains keys with hash addresses beginning with *10* and bucket C contains keys with addresses that start with 11. Figure 12.3 shows a trie that allows us to retrieve these buckets.

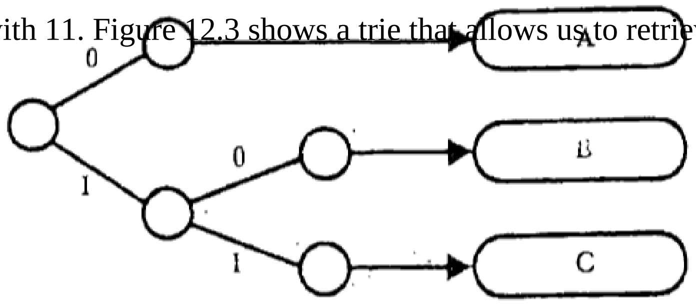


Figure 12.3

How should we represent the trie?

Rather than representing the trie as a tree, we flatten it into an array of contiguous records, forming a directory of hash addresses and pointers to the corresponding buckets.

Step 1: The first Step in turning a tree into an array involves extending it so it is a complete binary tree with all of its leaves at the Same level as shown in Fig. 12.4 (a). Even though the initial 01s enough to select bucket A, the new form of the tree also uses the second address bit so both alternatives lead to the same bucket.

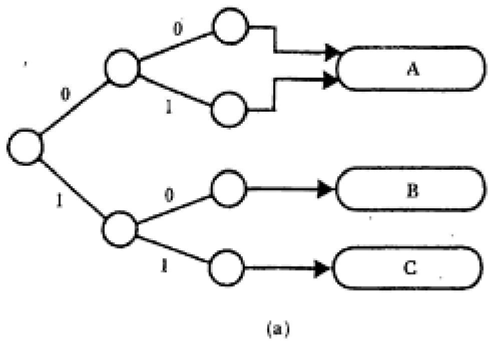


Figure 12.4 (a)

Step 2: Once we have extended the tree this way, we can collapse it into the d.irectory structure shown in Fig. 12.4(b). Now we have a Structure that provides the kind of direct access associated with hashing: given an address beginning with the bits *10,* the 1O2th directory entry gives us a pointer to the associated bucket.

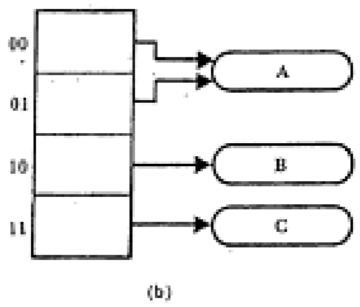


Figure 12.4(b)

12.2.3 Splitting to Handle Overflow

The goal in an *extendible* hashing system is to find a way to increase the address space in response to overflow.

Suppose we insert records that cause bucket A in fig. 12.4(b) to overflow. In this case the solution is Simple: since addresses beginning with *00* and *01* are mixed together in bucket A, we, can Split bucket A by putting all the *01* addresses in a new bucket D, while keeping only the *00* addresses in A.

We must use the full 2 bits to divide the addresses between two buckets. We do not need to extend the address $pace; we simply make full use of the address information that we already have. Figure 12.5 shows the directory and buckets after the split.

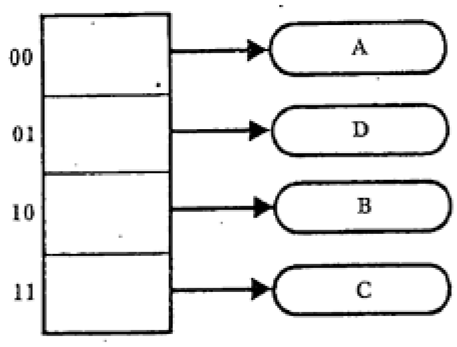


Figure 12.5

Let's consider a more complex case, starting once again with the directory and buckets in Fig. l2.4 (b), suppose that bucket B overflows. How do we split bucket B and where do we attach the new bucket after the split? Unlike our previous example, we do not have additional, unused bits of address space that we can press into duty as we split the bucket we now need to use 3 bits of the hash address in order to divide up the records that hash to bucket B. The trie illustrated in Fig. 12.6

 makes the distinctions required to complete the split. Figure 12.6 (b) shows what this trie looks like once it is extended into a completely full binary tree with all leaved at the same level, and Fig. 12.6 (c) shows the collapsed, directory form of the trie.

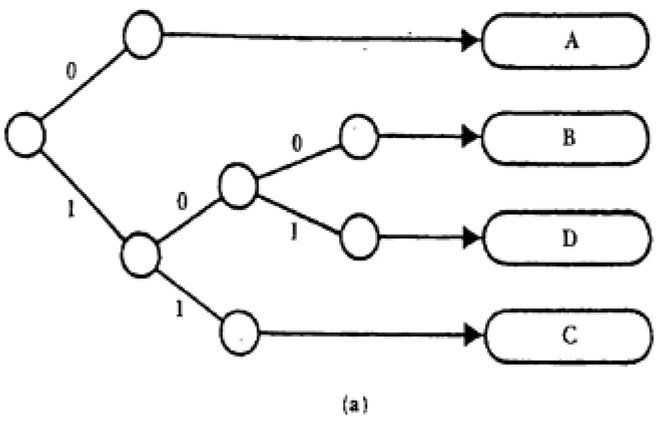
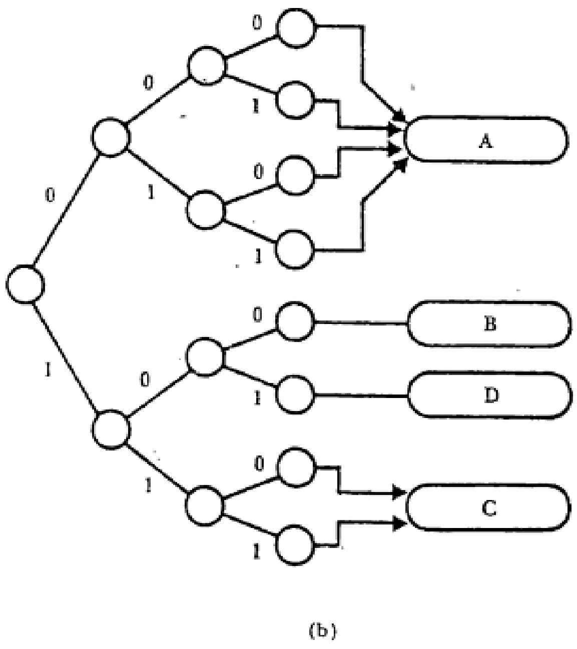


Figure 12.6 (a)



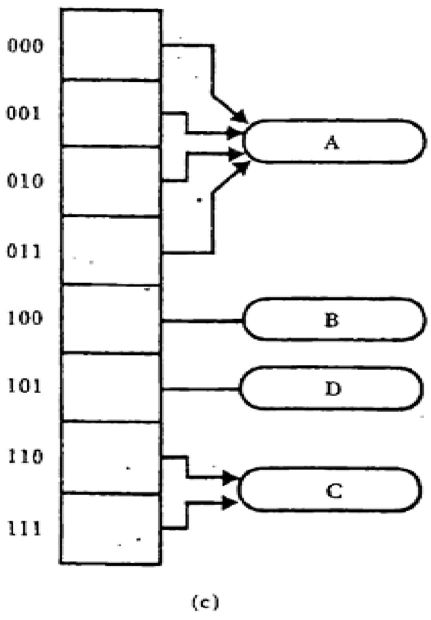


Figure 12.6 (b) Figure 12.6 (c)

12.3 Implementation

Creating the address

```
int Hash (char* key)
{
        int sum = 0;
        int len = strlen(key);
        if (len % 2 == 1) len ++; // make len even
        for an odd length, use the trailing '\0' as part of key for
        (int j = 0; j < Jen; j+=2)
```

```
            sum = (sum + 100 * key[j] + key[j+I]) % 19937; return
    sum;
}
```

The function Hash is a simple variation on the fold-and-add hashing algorithm. Because extendible hashing uses more bits of the hashed address as they are needed to distinguish between buckets, we need a function MakeAddress that extracts a portion of the full hashed address. We also use MakeAddress to reverse the order of the bits in the hashed address, making the lowest-order bit of the hash address the highest-order bit of the value used in extendible hashing. By reversing the bit order, working from right to left, we take advantage of the greater variability of low-order bit values.

```
int MakeAddress (char * key, int depth)
{
   int retval = 0;
   int hashVal = Hash(key);
   // reverse the bits
   for (int j = 0; j < depth; j++)
   {
      retval = retval << 1;
      int lowbit = hashVal & 1;
      retval = retval | lowbit;
      hashVal = hashVal >> 1;
   }
   return retval;
}
```

12.4 Deletion

Overview of the Deletion Process

If extendible hashing is to be a truly *dynamic* system like B-trees or AVL trees, it must be able to *shrink* files gracefully as well as grow them.

When do we combine buckets? This question, in turn, leads us to ask which buckets can be combined? In extendible hashing we use a similar concept: buckets that are *buddy* buckets. Look again at the trie in Fig. 12.6 (b). Which buckets could be combined? Trying to combine anything with bucket A would mean collapsing everything else in the trie first. Similarly, there is no single bucket that could be combined with bucket C. But buckets B and D are in the same configuration as buckets that have just split.` They are ready to be combined: they are buddy buckets.

After combining buckets, we examine the directory to see if we can make changes there. Looking at the directory form of the trie in Fig. 12.6 (c), we see that once we combine buckets B and D, directory cells

100 and *101* both point to the same bucket. In fact, each of the buckets has at least a pair of directory cells pointing to it. In other words, none of the buckets requires the depth of address information that is currently available in the directory. That means that we can shrink the directory and reduce the address space to half its size.

Reducing the size of the address space restores the directory and buck et structure to the arrangement shown in Fig. 12.4, before the additions and splits that produced the structure in Fig.12.6 (c). Reduction consists of collapsing each adjacent pair of directory cells into a single cell. This is easy, because both cells in each pair point to the same bucket. Note that this is nothing more than a reversal of the directory splitting procedure that we use when we need to add new directory cells.

12.5 Extendible hashing performance

A Procedure for Finding Buddy Buckets

The method works by checking to see exclusive or with low bit whether it is possible for there to be a buddy bucket. Clearly, if the directory depth is 0, meaning that there is only a single bucket, there cannot be a buddy. The next test compares the number of bits used by the bucket with the number of bits used in the directory address space. A pair of buddy buckets is a set of buckets that are immediate descendants of the same node in the trie. It is only when a bucket is at the outer edge of the trie that it can have a single parent and a single buddy.

Once we determine that there is a buddy bucket, we need to find its address. First we find the address used to find the bucket we have at hand ;

```
int Bucket::FindBuddy ()
{// find the bucket that is paired with this
if (Dir.Depth == O) return -I; // no buddy, empty directory

        unless bucket depth == directory depth, there ìs no single
        bucket to pair with
     if (Depth < Dir.Depth) return -1;
     int sharedaddress = MakeAddress(Keys[O], Depth);
            address of any key
     return sharedaddress ^ I; //
}
```

Since we know that the buddy bucket is the other bucket that was formed from a split, we k now that the buddy has the same address in all regards except for the last bit. So, to get the buddy address, we flip the last bit with an exclusive or.

Collapsing the Directory

The other important support function used to implement deletion is the function that handles collapsing the directory. Collapsing directory begins by making sure that we are not at the lower limit of directory size. The test.to see if the directory can be collapsed consists of examining each pair of directory cells to see if they point to different buckets. As soon as we find Such a pair, we know that- we *cannot* collapse the directory. If we get all the way through the directory without encountering such a pair, then we can collapse the directory.

The collapsing operation consists of` allocating space for a new array of bucket addresses that is half the size of the original and then copying the bucket references shared by each cell pair to a single cell in the new directory.

12.6 Alternative Approaches

Dynamic Hashing

Functionally, dynamic hashing and extendible hashing are very similar. Both use a directory to track the addresses of the buckets, and both-extend the directory through the use of tries.

The key difference between the approaches is that dynamic hashing, like conventional, static hashing, starts with a hash function that covers an address space of a fixed Size.

As buckets within that fixed address space overflow, they split, forming the leaves of a trie that grows down from the original address node.

Eventually, after enough additions and splitting, the buckets are addressed through a forest of tries that have been seeded out of the original static address space.

Let's look at an exa m ple. Figure 12.23 (a) Shows an initial address space of four and four buckets descending from the four addresses in the director y.

In Fig. 12.23 (b) we have split the bucket at address 4. We address the two buckets resulting from the split as 40 and *41* . We cha nge the sha pe of the directory node at address 4 from a square to a circle because it has changed from an external node.

In Fig. 12.23 (c) we split the bucket addressed by node 2, creating the new external nodes *20* and *21* . We also split the bucket addressed by *41* , extending the trie downward to include *410* and *41 1*. Finding a key in a dynamic hashing scheme can involve the use of two hash functions rather than just one. First, there is the hash function that covers the original address space. If you find that the directory node is an external node and therefore points to a bucket, the search is complete. However, if the directory node is an internal node, then you need additional address information to guide you through the ls and 0s that form the trie.

The primary difference between the two approaches is that dynamic hashing allows for slower, more gradual growth of the directory, whereas extendible hashing extends the directory by doubling it.

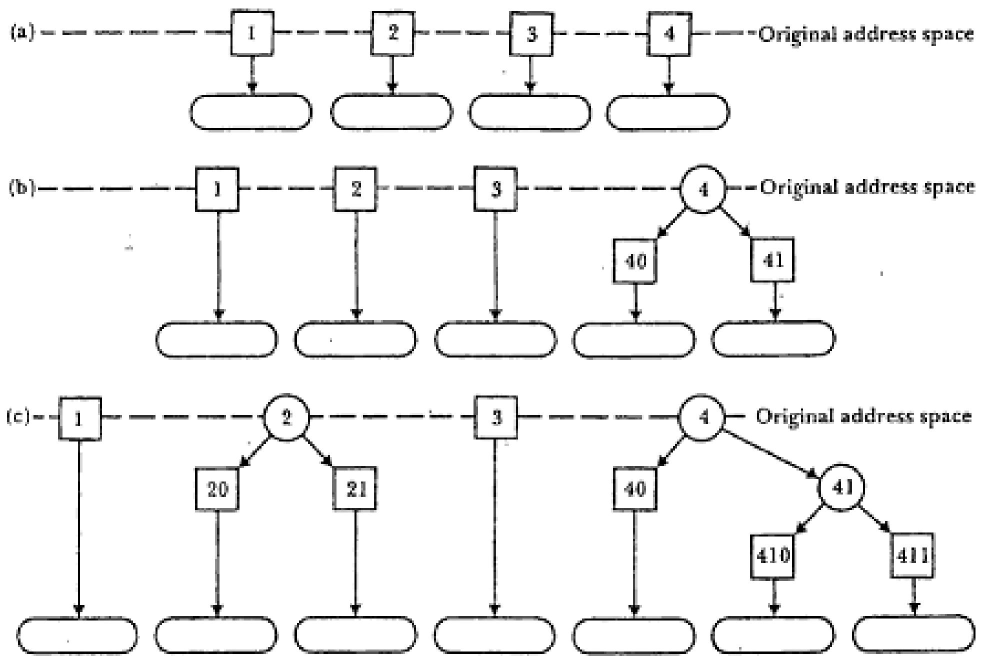


Figure 12.23

Linear Hashing

Linear hashing, introduced by Litwin in 1980, does away with the directory. Linear hashing, like extendible hashing, uses more bits of hashed value as the address space grows. Note that the address space consists of four *buckets* rather than four directory nodes that can point to buckets.

As we add records, bucket b overflows. The overflow forces a split. However, as Fig. 12.24(b) Shows, it is not bucket b that splits, but bucket a. The reason for this is that we are extending the address space *linearly,* and bucket a is the next bucket that must split to create the next linear exten-sion, which we call bucket A. A 3-bit hash function, h3(k), is applied to buckets a‘and A to divide

the records between them. Since bucket b was not the bucket that we split, the overflowing record is placed into an overflow bucket w.

We add more records, and bucket d overflows. Bucket b is the next one to split and extend the address space, so we use the h3(k) address function to divide the records from bucket b and its overflow bucket between b and the new bucket B. The record overflowing bucket d is placed In an overflow bucket x. The resulting arrangement is illustrated in Fig. 12.24(c).

Figure 12.24 (d) shows what happens when, as we add more records, bucket d overflows beyond the capacity of the overflow bucket w. Bucket c is the next in the extension sequence, so we use the h3(k)address function to divide the records between c and C.

Finally, assume that bucket B overflows. The overflow record is placed in the overflow bucket z. The overflow also triggers the extension to bucket D, dividing the contents of d, x, and y between buckets d and D. At this point all of the buckets use the h3(k) address function, and we have finished the expansion cycle. The pointer for the next bucket to be split returns, to bucket a to get ready for a new cycle that will use an h4(k) address function to reach new buckets.

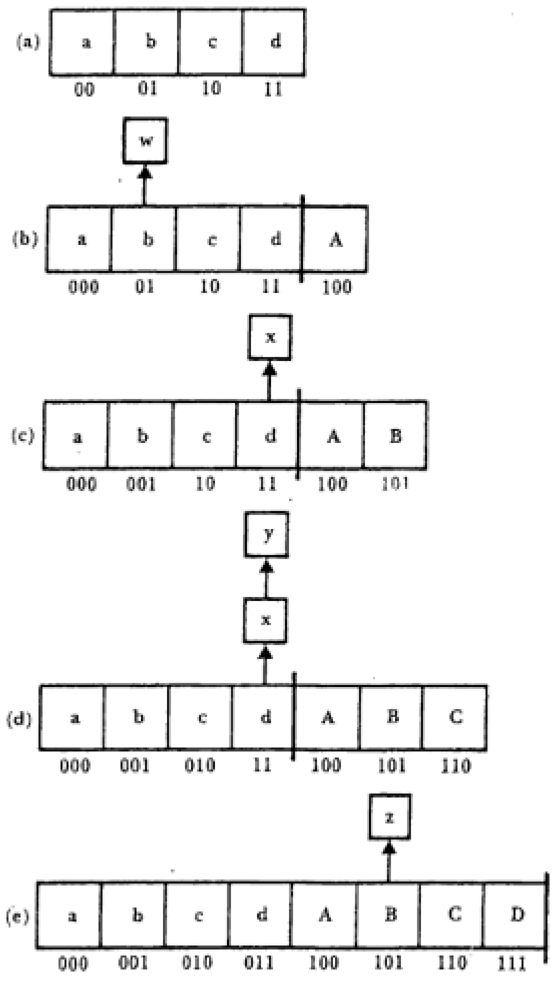


Figure 12.24

**********END OF MODULE – 5**********