Atria Institute of Technology

Department of Information Science and Engineering

Bengaluru-560024

ACADEMIC YEAR: 2021-2022
EVEN SEMESTER NOTES

Semester	**:**	**6th Semester**
Subject Name	**:**	**File Structure**
Subject Code	**:**	**18IS61**
Faculty Name	**:**	**Mrs. Ranjitha J**

Name of the Course: File Structure
NOTES

Introduction

The Heart of File Structure Design

Disks are slow compared to other parts of the computer. They are very slow compared to accessing time of a memory on the other hand they provide enormous capacity to hold data at much less cost. They are non-volatile and preserve the information even when the system is turned off. The driving force behind file structure design is the disks relative slow access and the non-volatile capacity.

Secondary storage such as disks can pack thousand of megabytes in a small physical location. Computer memory(RAM) is limited. However, relative to memory, access to secondary storage is extremely slow(for example- getting information from slow RAM takes 120 nanosecond – 10^{-9} seconds. While getting information from disk disk takes 30 milliseconds – 10^{-3} seconds.

Good File Structure design will give us access to all the capacity without making our application spend a lot of time waiting for the disk. A file structure is a combination of representation for data in files and of operation for accessing the data. A file structure allows application to read write and modify the data. It might also support finding the data that matches some search criteria or reading through the data in some particular order.

A Short History of File Structure Design

1. General Goals – The general goals of R&D in file structure can be drawn directly from our analogy.
   - Get the information we need with one access to the disk.
   - If that’s not possible, then get the information with as few accesses as possible.
   - Group information so that we are likely to get everything we need with only one trip to the disk.
2. Early work – Early Work assumed that files were on tape. Access was sequential and the cost of access grew in direct proportion to the size of the file.

3. The emergence of Disks and Indexes – As files grew very large, unaided sequential access was not a good solution. Disks allowed for direct access and Indexes made it possible to keep a list of keys and pointers in a small file that could be searched very quickly. With the key and pointer, the user had direct access to the large, primary file.
4. The emergence of Tree Structures – As indexes also have a sequential flavour, when they grew too much, they also became difficult to manage. The idea of using tree structures to manage the index emerged in the early 60's. However, trees can grow very unevenly as records are added and deleted, resulting in long searches requiring many disk accesses to find a record.
5. Balanced Trees – In 1963, researchers came up with the idea of AVL trees for data in memory. AVL trees, however, did not apply to files because they work well when tree nodes are composed of single records rather than dozens or hundreds of them. In the 1970's came the idea of B-Trees which require an O(logk N) access time where N is the number of entries in the file and k, th number of entries indexed in a single block of the B-Tree structure --> B-Trees can guarantee that one can find one file entry among millions of others with only 3 or 4 trips to the disk.
6. Hash Tables – Retrieving entries in 3 or 4 accesses is good, but it does not reach the goal of accessing data with a single request. From early on, Hashing was a good way to reach this goal with files that do not change size greatly over time. Recently, Extendible Dynamic Hashing guarantees one or at most two disk accesses no matter how big a file becomes.

<u>Conceptual Toolkit: File Structure Literacy</u>

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 and 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.

Fundamental File Processing Operations

Physical Files and Logical Files

Operating System is responsible for associating a logical file in a program to a physical file in disk or tape, writing to or reading from a file in a program is done through the Operating System.

- Physical file: physically exists on secondary storage; known by the operating system; appears in its file directory .
- Logical file, what your program actually uses, a 'pipe' though which information can be extracted, or sent.
- Operating system: get instruction from program or command line; link logical file with physical file or device .

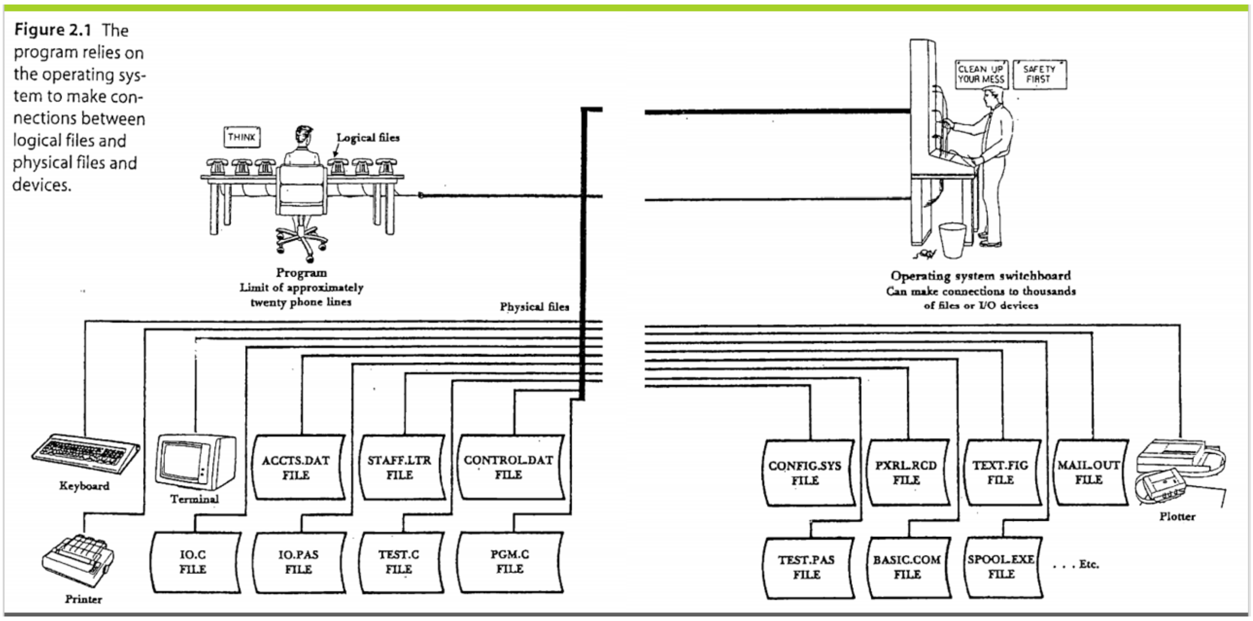


Figure 2.1 The program relies on the operating system to make connections between logical files and physical files and devices.

The figure specifies that the application program relies on the operating system to take care of the details of the telephone switching system. Note that from the program point of view, input devices(keyboard) and output devices(console, pointer, etc) are treated as files. places where bytes come from or sent to. These may be thousand of physical files on a disk, but a program only have a limited number of logical files open at the same time.

To deal with Files

– An object stream should be defined (The logical name)

fstream file_name; // for Input and output.

ifstream file_name; // for Input

ofstream file_name; // for Output

– Logical file should be connected to the physical file and the mode should be specified.

• Filename.open("Physical file name", mod)

Or

• It can be done through the constructor

– E.g., ofstream file_name("test.txt", ios::out);

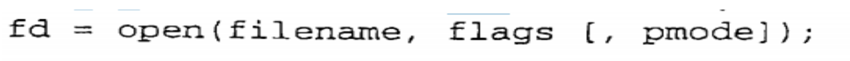
The physical file has a name, for instance "account.txt"

The logical file has a logical name used for referring to the file inside the program. The logical name is a variable inside the program, for instance "infile".

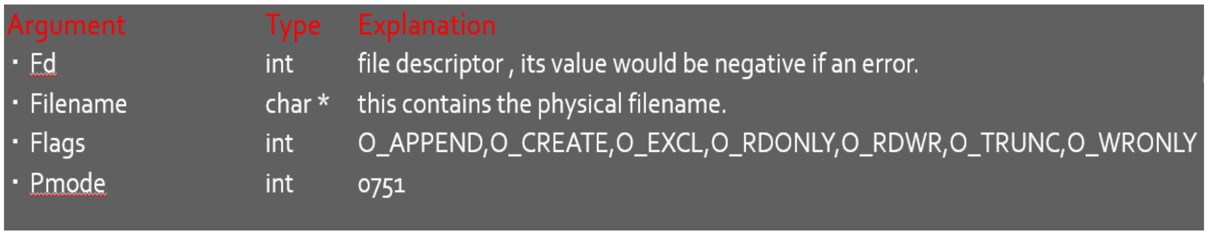
Opening Files

The operating system would hook up the logical identifier with the physical file. Then need to declare what intend to do with the file. In general two options are there open an existing file, create a new file.

The open function called is used to create or open a file. This is present in the header "fcntl.h".



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



Argument	Type	Explanation
• Fd	int	file descriptor , its value would be negative if an error.
• Filename	char *	this contains the physical filename.
• Flags	int	O_APPEND,O_CREATE,O_EXCL,O_RDONLY,O_RDWR,O_TRUNC,O_WRONLY
• Pmode	int	0751

Pmode- protection mode = 0751 =	111	101	001
	rwe	rwe	rwe
	(owner)	(group)	(others)

Example

```
fd = open(filename, O_RDWR | O_CREAT, 0751);

fd = open(filename, O_RDWR | O_CREAT | O_TRUNC, 0751);
```

Closing Files

Files are usually closed automatically by the operating system when a program terminate normally. The execution of close statement within a program is needed only to protect it against data loss in the event that the program is interrupted and to free up logical filenames for reuse.

Reading and writing

Reading and writing are fundamental to file processing; they are the actions that make file processing an input/output(I/O) operation. The form of the read and write statements used in different languages varies.

- Read and Write Functions.
- Files with C streams and C++ stream classes.
- Program in C++ to display the contents of a file.
- Detecting End-of-File.

1. Read and Write Functions

   Reading and writing at a relatively low level. It is useful to have a kind of systems-level understanding of what happens when send and receive information to and from a file.

   A low level read call requires three pieces of information, expressed as arguments to a generic read function.

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

Source_file – the read call must know where it is to read from.(Remember, before we do any reading, we must have already opened the file so the connection between a logical file and a specific physical file or device exists.

Destination_addr – read must know where to place the information it reads from the input file. In this function specify the destination by giving the first address of the memory block where to store the data.

Size – Read must know how much information to bring in from the file. Here the argument is supplied as a byte count.

A write statement is similar; the only difference is that the data moves in the other direction:

```
Write(Destination_file, Source_addr, Size)
```

Destination_file – the logical file name that is used for sending the data.

Source_addr – write must know where to find the information it will send.

Size – the number of bytes to be written must be supplied.

2. Files with C streams and C++ stream classes

   I/O operation in C and C++ are based on the concept of a stream. The first uses the standard C functions defined in header file stdio.h. this is often referred to as C streams or C input/output. The second uses the stream classes of header files iostream.h and fstream.h this is often referred to as c++ streams.

   Other files can be associated with streams through the use of the fopen function:

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

The return value file and the arguments filename and type have the following meanings:

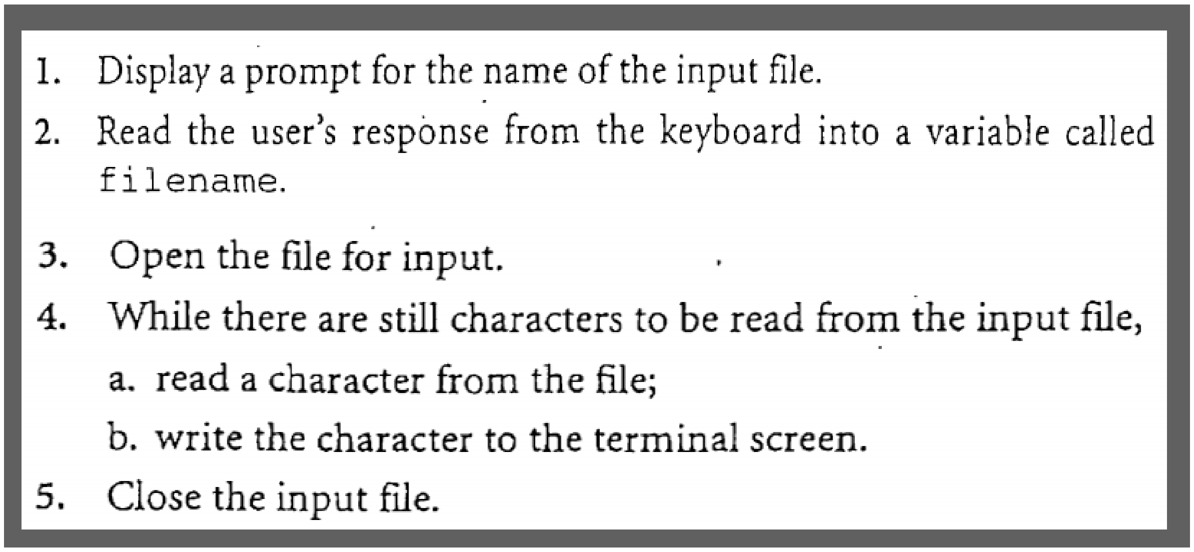
Argument	Type	Explanation	
file	FILE *	A pointer to the file descriptor. Type FILE is another name for struct _iobuf. If there is an error in the attempt to open the file, this value is null, and the variable errno is set with the error number.	
filename	char *	The file name, just as in the Unix open function.	
type	char *	The type argument controls the operation of the open function, much like the flags argument to open. The following values are supported:	
		"r"	Open an existing file for input.
		"w"	Create a new file, or truncate an existing one, for output.
		"a"	Create a new file, or append to an existing one, for output.
		"r+"	Open an existing file for input and output.
		"w+"	Create a new file, or truncate an existing one, for input and output.
		"a+"	Create a new file, or append to an existing one, for input and output.

```
fstream (); // leave the stream unopened
fstream (char * filename, int mode);
int open (char * filename, int mode);
int read (unsigned char * dest_addr, int size);
int write (unsigned char * source_addr, int size);
```

The argument filename of the second constructor and the method open are just as seen before. These two operations attach the fstream to a file. The value of mode controls the way the file is opened, like the flags and type argument. The value is set with a bit-wise or of constants defined in class ios.

3. Programs in C++ to display the contents of a file

   The 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,
   a. read a character from the file;
   b. write the character to the terminal screen.
5. Close the input file.

```
c++ program to read & display all the contents from the specified file

#include<iostream.h>
#include<fstream.h>
void main()
{
        fstream file;
        char fname[10];
        char ch;
        cout << "Enter a filename names.txt";   step1
        cin >> fname;   step2
        file.open(names.txt,ios::in);   step3
        while(!file.fail())
        {
                file >> ch;     step4a
                cout << ch;     step4b
        }
        file.close();   step5
}
```

4. Detecting End-of-File

Each C++ stream has a state that can be queried with function calls.

The function fail, which returns true(1) if the previous operation on the stream failed. A function end_of_file can be used to test for end-of-file. As read from a file, the operating system keeps track of our location in the file with a read/write pointer. This is necessary: when the next byte is read, the system knows where to get it. The end_of_file function queries the system to see whether the read/write pointer has moved past the last element in the file. If it has, end_of_file returns true; otherwise it returns false.

```
if (file.fail()) break;
```

Seeking

The action of moving directly to a certain position in a file is often called seeking. A seek requires at least two pieces of information.

1. Seeking with C streams
2. Seeking with C++ Streams

```
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.

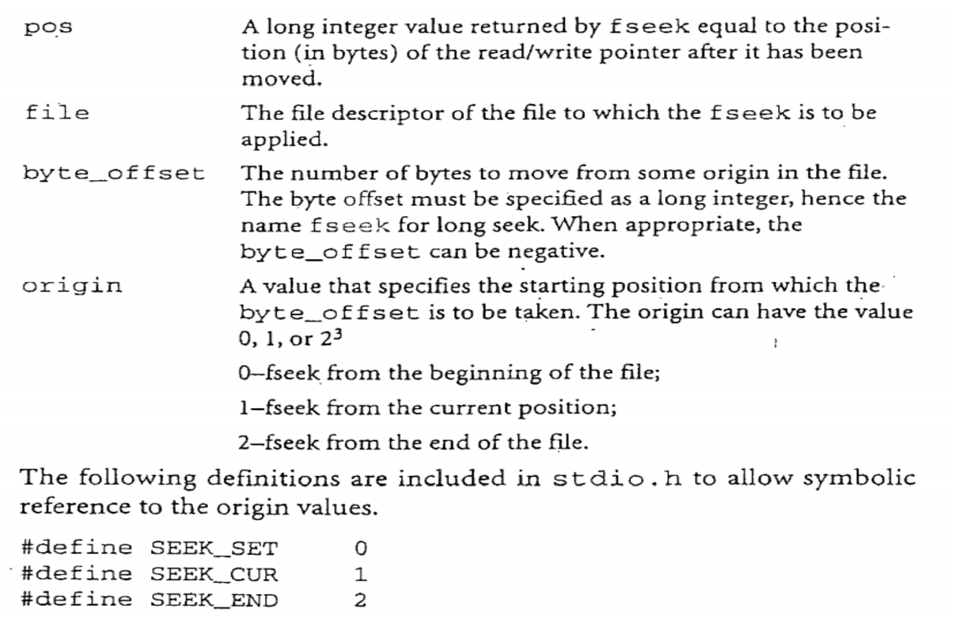
Now, if want to move directly from the origin to the 373d position in a file called data, so no need to move sequentially through the first 372 position. Instead, can say

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

1. Seeking with C streams

   In unix a file can be viewed as very large array of bytes that are stored on secondary memory. In an array of bytes in memory can move to any particular byte using a subscript. The C stream seek function "fseek"provides a similar capability for files. Syntax of fseek is:

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



`pos`	A long integer value returned by `fseek` equal to the position (in bytes) of the read/write pointer after it has been moved.
`file`	The file descriptor of the file to which the `fseek` is to be applied.
`byte_offset`	The number of bytes to move from some origin in the file. The byte offset must be specified as a long integer, hence the name `fseek` for long seek. When appropriate, the `byte_offset` can be negative.
`origin`	A value that specifies the starting position from which the `byte_offset` is to be taken. The origin can have the value 0, 1, or 2[3] 0–fseek from the beginning of the file; 1–fseek from the current position; 2–fseek from the end of the file.

The following definitions are included in `stdio.h` to allow symbolic reference to the origin values.

```
#define SEEK_SET     0
#define SEEK_CUR     1
#define SEEK_END     2
```

2. Seeking with C++ streams

Seeking with C++ streams is almost similar to C streams with following syntactic differences:

- An object of type fstream has two file pointers. A get pointer for input and put pointer for output. Two functions are used to move these pointers. Seekg moves the get pointer and seekp moves the put pointer.
- The seek operations are methods of the stream classes. Hence the syntax is:

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

Here the value of origin comes from ios class. The values are:

ios::beg – from beginning of the file.

ios::cur – from current position.

ios::end – from end of the file.

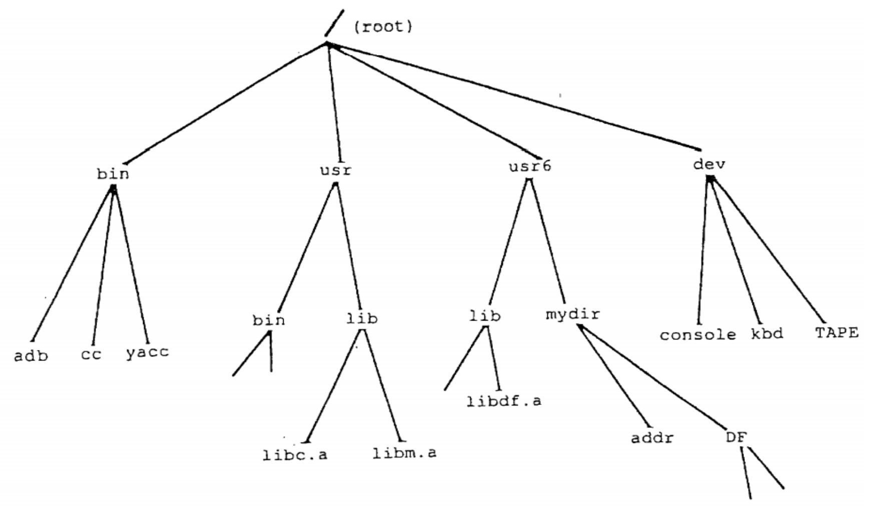
Example: file.seekg(373,ios::beg); file.seekp(373,ios::beg)

Special Characters in Files

You may encounter some difficulty with extra, unexpected characters that turn up in your files with character that disappear and with numeric counts that are inserted into your files. Some example of the kinds of things you might encounter:

- On many computers you may find that a Control-Z (ASCII value of 26) is appended at the end of your files. Some applications use this to indicate end-of-file even if you have not placed it there. This is most likely to happen on MS-DOS systems.
- Some systems adopt a convention of indicating end-of-line in a text file[4] as a pair of characters consisting of a carriage return (CR: ASCII value of 13) and a line feed (LF: ASCII value of 10). Sometimes I/O procedures written for such systems automatically expand single CR characters or LF characters into CR-LF pairs. This unrequested addition of characters can cause a great deal of difficulty. Again, you are most likely to encounter this phenomenon on MS-DOS systems. Using flag "b" in a C file or mode ios::bin in a C++ stream will suppress these changes.
- Users of larger systems, such as VMS, may find that they have just the opposite problem. Certain file formats under VMS *remove* carriage return characters from your file without asking you, replacing them with a *count* of the characters in what the system has perceived as a line of text.

The Unix Directory Structure



A unix file system is a tree structured organization of directories, with the root of the tree signified by the character /. All the directories including the root contains two types of files

Regular files and Directories.

Any file in the unix file system can be uniquely identified by the absolute pathname that begins with root directory. all the pathname that begin with the current directory is called as relative pathname. The special file names '.' Stands for current directory and '..' stands for parent directory.

Physical Devices and Logical Files

1. Physical Devices as Files
2. The Console, the keyboard, and standard error
3. I/O Redirection and pipes

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. 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

3. I/O Redirection and pipes

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

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

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

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

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

Example:

```
list.exe > myfile
```

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

pipe

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

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

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

```
program1 | program2
```

Output of program1 is used as input for program2

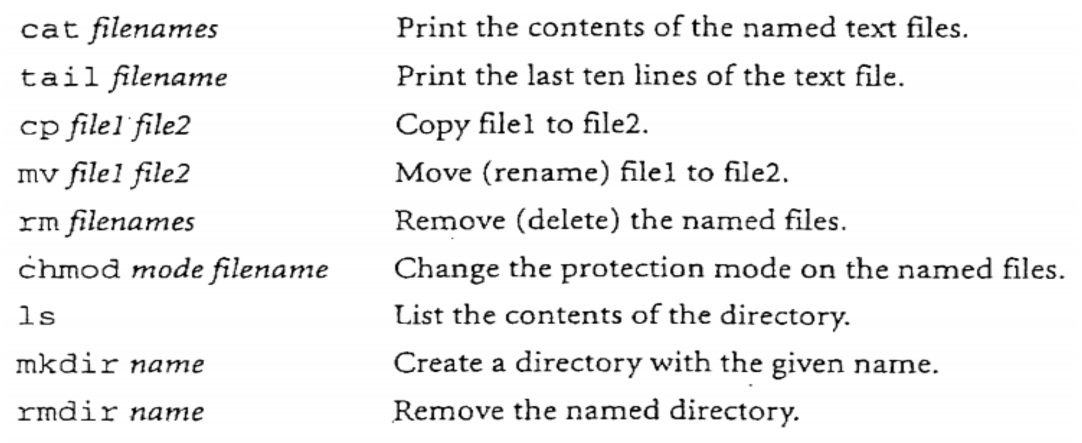
File-Related Header Files

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

Unix File System Commands

Unix provides many commands for manipulating files. Unix manual for more information on how to use them.



`cat` *filenames*	Print the contents of the named text files.
`tail` *filename*	Print the last ten lines of the text file.
`cp` *file1 file2*	Copy file1 to file2.
`mv` *file1 file2*	Move (rename) file1 to file2.
`rm` *filenames*	Remove (delete) the named files.
`chmod` *mode filename*	Change the protection mode on the named files.
`ls`	List the contents of the directory.
`mkdir` *name*	Create a directory with the given name.
`rmdir` *name*	Remove the named directory.

Disks

1. The organization of disks
2. Estimating capacities and space needs
3. Organizing tracks by sector
4. Organizing tracks by block
5. Nondata overhead
6. The cost of a disk access
7. Effect of block size on performance: a unix example
8. Disk as bottleneck

Disk drives belong to a class of devices known as direct access storage devices(DASDs) because they make it possible to access data directly. DASDs are contrasted with serial devices, the other major class of secondary storage devices. Serial devices use media such as magnetic tape that permit only serial access, which means that a particular data item cannot be read or written until all of the data preceding it on the tape have been read or written in order.

1. The organization of disks
   Magnetic disks come in many forms. So called hard disks offer high capacity and low cost per bit. Hard disks are the most common disk used for file processing.
   The information stored on a disk is stored on the surface of one or more platters. The arrangement is such that the information is stored in successive tracks on the surface of the disks. Each track is often divided into number of sectors. A sector is the smallest

addressable portion of a disk. Disk drives have a number of platters. The tracks that are directly above and below one another form a cylinder can be accessed without moving the arm that holds the read/write head. Moving this arm is called seeking. The arm movement is the slowest part of reading information from a disk.

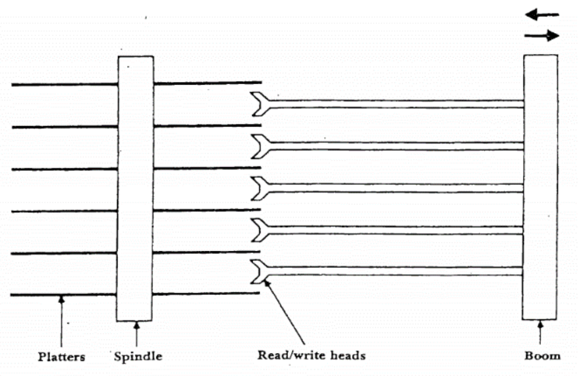


Figure 3.1 Schematic illustration of disk drive.

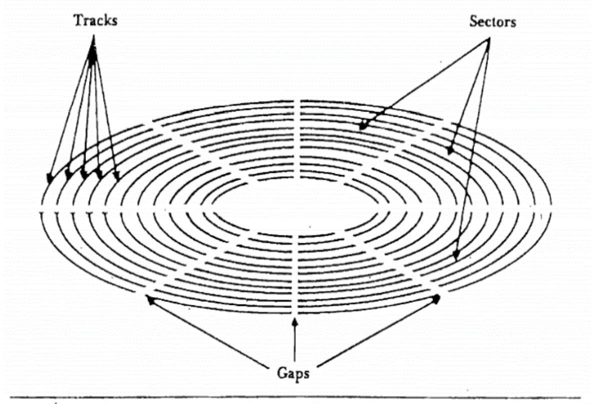


Figure 3.2 Surface of disk showing tracks and sectors.

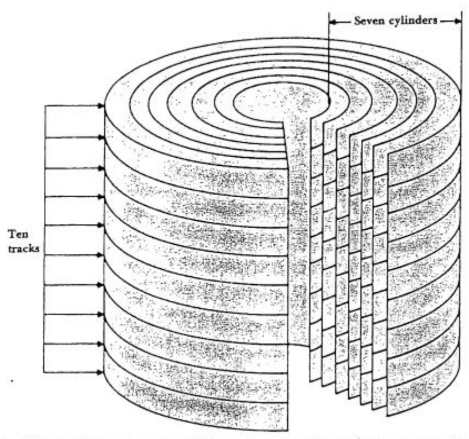


Figure 3.3 Schematic illustration of disk drive viewed as a set of seven cylinders.

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

2. Estimating capacities and space needs

Storage capacity of a disk range from hundreds of millions to billions of bytes. In a disk each platter has two surfaces. So the number of tracks per cylinder is twice the number of platters. The capacity of the disk is a function of the number of cylinders, the number of tracks per cylinder and the capacity of the track.

Track capacity = number of sectors per track × bytes per sector
Cylinder capacity = number of tracks per cylinder × track capacity
Drive capacity = number of cylinders × cylinder capacity.

3. Organizing tracks by sector
   - The physical placement of sector

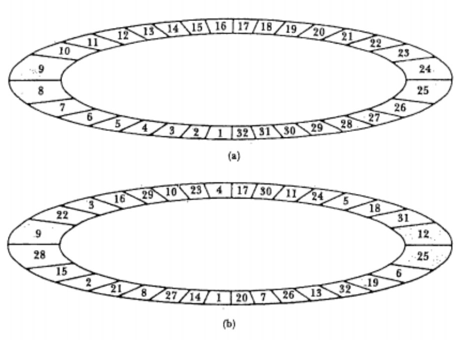


Figure 3.4 Two views of the organization of sectors on a thirty-two-sector track.

The sectors are usually adjacent, fixed sized segments of tracks that happen to hold a file. This is a good way to view the file logically, but it may not be a good way to store sectors physically. If the sectors are placed one right after the other, all in the same track we cannot read adjacent sectors. The disk controller after reading the data, takes some time to process the received information before it is ready to accept the next. If the sectors are placed physical adjacent to each other then we would miss the start of the following sector while we are processing the one we have just read in. Due to this would be able to read only one sector per revolution. This problem can be solved by interleaving the sectors. An interval of several physical sectors are left between logically adjacent sectors. For a disk of interleaving factor of 5 the following figure shows the organization.

   - Cluster

A cluster is a fixed number of contiguous sectors. Once a given cluster has been found on a disk all sector in that cluster can be accessed without requiring an additional seek. The file manager ties logical sectors to the physical location of the cluster they belong to by using a file allocation table(FAT). Each entry in the FAT is an entry giving the physical location of the cluster.

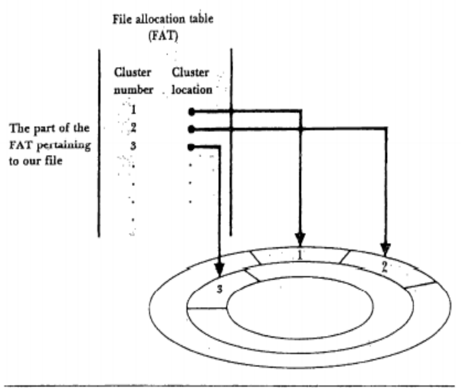

File allocation table (FAT)
Cluster number
Cluster location
The part of the FAT pertaining to our file
1
2
3

Figure 3.5 The file manager determines which cluster in the file has the sector that is to be accessed.

- Extents
  Extents of a file are those parts of the file which are stored in contiguous clusters. It is preferable to store the entire file in one extent. But this may not be possible due to various reasons like non availability of contiguous space errors in allocated sectors etc.

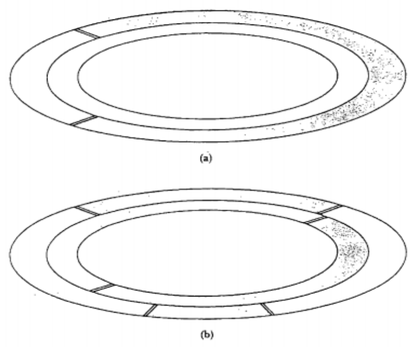


Figure 3.6 File extents (shaded area represents space on disk used by a single file).

- Fragmentation

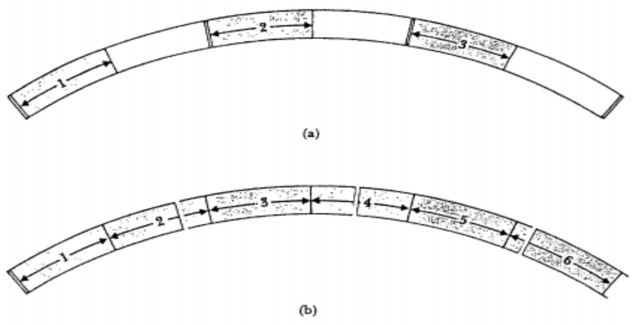


Figure 3.7 Alternate record organization within sectors (shaded areas represent data records, and unshaded areas represent unused space).

Unused space within a file.

- Clusters are also referred to as allocation units (ALUs).
- Space is allocated to files as integral numbers of clusters.
- A file can have a single extent, or be scattered in several extents.
- Access time for a file increases as the number of separate extents increases, because of seeking.
- Defragmentation utilities physically move files on a disk so that each file has a single extent.
- Allocation of space in clusters produces fragmentation.
- A file of one byte is allocated the space of one cluster.
- On average, fragmentation is one-half cluster per file.

4. Organizing tracks by block
   Sometimes disk tracks are not divided into sectors, but into integral numbers of user-defined blocks whose sizes can vary.

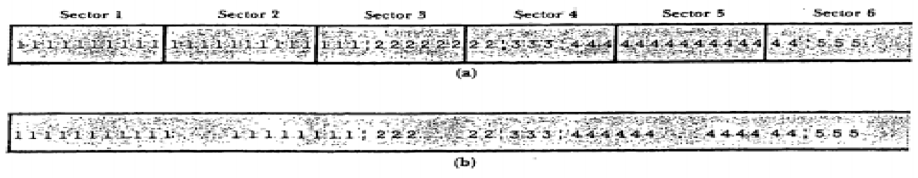


Figure 3.8 Sector organization versus block organization.

In the above figure illustrate the dtfference between one view of data on a sectored track and that on a blocked track. A blocking organization does not present the sector-spanning and fragmentation problems of sectors because blocks can vary in size to fit the logical organization of the data. A block is usually organized to hold an integral number of logical records. The term blocking factor is used to indicate the number of records trhat are to be stored in each block in a file.

In blocking-addressing schemes, each block of data is usually accompanied by one or more subblocks containing extra information about the data block. Typically there is a count subblock that contains the number of bytes in the accompanying data block(figure 3.9a). There may also be a key subblock containing the key for the last record in the data block(figure.3.9b). When key subblocks are used, the disk controller can search a track for a block with desired key. This approach can result in much more efficient searches than normally possible with sector-addressable schemes, in which keys are generally cannot be interpreted without first loading them into primary memory.

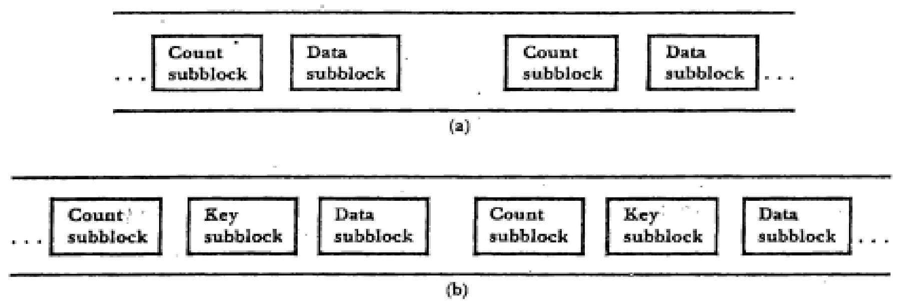


Figure 3.9 Block addressing requires that each physical data block be accompanied by one or more subblocks containing information about its contents.

5. Nondata overhead
   Both blocks and sectors require that a certain amount of space be taken up on the disk in form of nondata overhead. Some of the overhead consists of information that is stored on the disk during performatting which is done before the disk can be used.

   On sector-addressable disks, preformatting involves storing, at the beginning of each sector, information such as sector address, track address, and condition (whether the sector is usable or defective). Preformatting also involves placing gaps and synchronization marks between fields of information to help the read/write mechanism distinguish between them. This nondata overhead usually is of no concern to the programmer. When the sector size is given for a certain drive, the programmer can assume that this is the amount of actual data that can be stored in a sector.

   On a block-organized disk, some of the nondata overhead is invisible to the programmer, but some of it must be accounted for. Since subblocks and interblock gaps have to be provided with every block, there is generally more nondata information provided with blocks than with sectors. Also, since the number and size of blocks can vary from one application to another, the relative amount of space taken up by overhead can vary when block addressing is used.

6. The cost of a disk access
   A disk access can be divided into three distinct physical operations, each with its own cost:
   - Seek Time – the time taken for read/write head move to the required cylinder is called seek time. The amount of time spent seeking during a disk access depends on how far the arm/head has to move. Since it is difficult to know exactly how many track will be traversed in every seek. We determine the average seek time. Most hard disks available today have average seek time of less than 10 milliseconds.

- Rotational Delay – the time taken for a disk to rotate and bring the required sector under read/write head is called rotational delay. This is considered to be half of the time taken for one rotation. If the hard disk spins at the speed of 500rpm thren it takes 12ms to complete the rotation, so the rotational delay is 6ms.

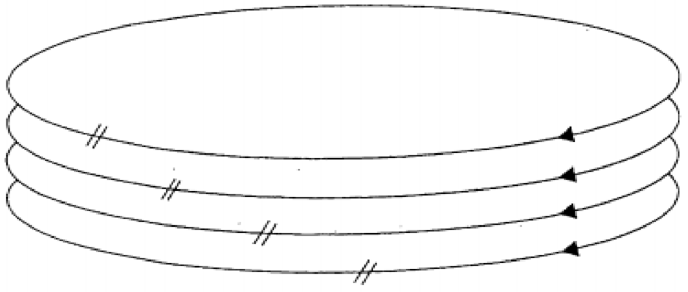


Figure 3.10 When a single file can span several tracks on a cylinder, we can stagger the beginnings of the tracks to avoid rotational delay when moving from track to track during sequential access.

- Transfer Time – the time taken to transfer one byte of data from track to read/write head is called transfer time. Transfer time can be calculated using

$$\text{Transfer time} = \frac{\text{number of bytes transferred}}{\text{number of bytes on a track}} \times \text{rotation time}$$

7. Effect of block size on performance: a unix example
   The results are summarized in table 3.2.

Table 3.2 The amount of wasted space as a function of block size.

Space Used (MB)	Percent Waste	Organization
775.2	0.0	Data only, no separation between files
807.8	4.2	Data only, each file starts on 512-byte boundary
828.7	6.9	Data + inodes, 512-byte block Unix file system
866.5	11.8	Data + inodes, 1024-byte block Unix file system
948.5	22.4	Data + inodes, 2048-byte block Unix file system
1128.3	45.6	Data + inodes, 4096-byte block Unix file system

From *The Design and Implementation of the 4.3BSD Unix Operating System*, Leffler et al., p. 198.

8. Disk as bottleneck
   Disk performance is increasing steadily, even dramatically, but disk speeds still lag far behind local network speeds. A number of techniques are used to solve this problem. One is multiprogramming, in which the CPU works on other jobs while waiting for the data to arrive. But if multiprogramming is not available or if the process simply cannot afford to lose so much time waiting for the disk, methods must be found to speed up disk I/O. One technique now offered on many high-performance systems is called striping. Disk striping involves splitting the parts of a file on several different drives, then letting the separate drives deliver parts of the file to the network simultaneously. Disk striping can be used to put different blocks of the file on different drives or to spread individual blocks onto different drives.

Magnetic Tape

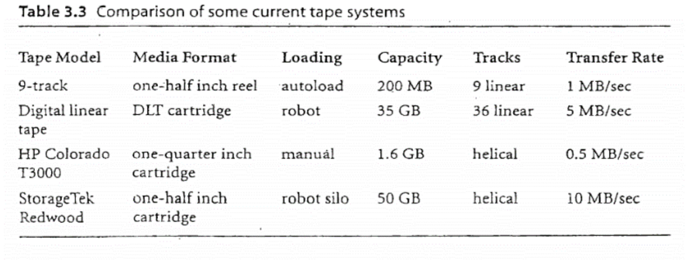
1. Types of Tape System
2. An Example of a High-Performance Tape System
3. Organization of Data on Nine-Track Tapes
4. Estimating Tape Length Requirements
5. Estimating Data Transmission Times

Many years ago tape systems were widely used to store application data. An application that needed data from a specific tape would issue a request for the tape, which would be mounted by an operator onto a tape drive. At present tapes are primarily used as archival storage. That is, data is written to tape to provide low cost storage and then copied to disk whenever it is needed. Tapes are very common as backup devices for PC systems. In high performance and high volume applications, tapes are commonly stored in racks and supported by a robot system that is capable of moving tapes between storage racks and tape drives.

1. Types of Tape System

Table shows a comparison of some current systems. In the past, most computer installations had a number of reel-to-reel tape drives and large numbers of racks or cabinets holding tapes. The primary media was one-half inch magnetic tape on 10.5-inch reels with 3600 feet of tape.

Table 3.3 Comparison of some current tape systems



Tape Model	Media Format	Loading	Capacity	Tracks	Transfer Rate
9-track	one-half inch reel	autoload	200 MB	9 linear	1 MB/sec
Digital linear tape	DLT cartridge	robot	35 GB	36 linear	5 MB/sec
HP Colorado T3000	one-quarter inch cartridge	manual	1.6 GB	helical	0.5 MB/sec
StorageTek Redwood	one-half inch cartridge	robot silo	50 GB	helical	10 MB/sec

2. An Example of a High-Performance Tape System

The StorageTek Redwood SD3 is one of the highest-performance tape systems available in 1997. It is usually configured in a *silo* that contains storage racks, a tape robot, and multiple tape drives. The tapes are 4-by-4-inch cartridges with one-half inch tape. The tapes are formatted with helical tracks. That is, the tracks are at an angle to the linear direction of the tape. The number of individual tracks is related to the length of the tape rather than the width of the tape as in linear tapes. The expected reliable storage time is more than twenty years, and average durability is 1 million head passes.

3. Organization of Data on Nine-Track Tapes

On a tape the logical position a byte within a file corresponds directly to physical position relative to the start of the file. A surface of a typical tape consist of parallel tracks each of which is a sequence of bits. If there are nine tracks the nine bits that are at corresponding positions in the nine respective tracks are taken to constitute 1 byte plus a parity bit which is also called as a frame. Frames are grouped into data block size may vary tapes are often read one block at a time and as they cannot stop or start immediately, blocks are separated by interblock gaps. The performance of the tape is measured by

Tape Density: no. of bits per inch

Tape Speed: no. of inches transmitted per sec

Size of Interblock gap: space b/w any two data block.

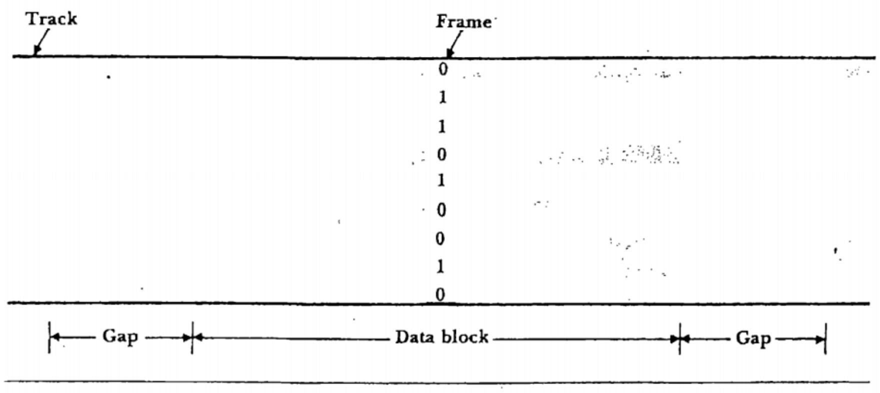


Figure 3.11 Nine-track tape.

4. Estimating Tape Length Requirements
   The length of the tape is calculated using the formula

$$s = n \times (b + g)$$

b = the physical length of a data block,
g = the length of an interblock gap, and
n = the number of data blocks

Suppose it is required to store a backup of one million records on a 6250bpi tape which would has a interblock gap of 0.3 inches and a blocking factor of 50. Calculate the tape length required.

Given – no. of rec = 1000000
Bpi= 6250
g = 0.3 inches
blocking factor = 50

to find – L ?

s= n*(b+g)

b= block size(bytes per block)/tape density(bytes per inch)

b= 50/6250
= 0.008 inch

s = 1000000*(0.008+0.3)
s = 308000 inch
s= 308000/12
s = 25666.6 feet

5. Estimating Data Transmission Times

Transmission rate is affected by tape density and tape speed. Transmission rate can be calculated using

Nominal rate = tape density (bpi) × tape speed (ips)

There are other factors that can influence performance, block size is generally considered to be the one variable with the greatest influence on space utilization and data transmission rate. The other factors have included- gap size, tape speed , and recording density- are often beyond the control of the user. Another factor that can sometimes be important is the time it takes to start and stop the tape.

Disk versus Tape

FOR COMPARISON	MAGNETIC TAPE	MAGNETIC DISK
Basic	Used for backup, and storage of less frequently used information.	Used as a secondary storage.
Physical	Plastic thin, long, narrow strip coated with magnetic material.	Several platters arranged above each other to form a cylinder, each platter has a read-write head.
Use	Idle for sequential access.	Idle for random access.
Access	Slower in data accessing.	Fast in data accessing.
Update	Once data is fed, it can't be updated.	Data can be updated.
Data loss	If the tape is damaged, the data is lost.	In a case of a head crash, the data is lost.
Storage	Typically stores from 20 GB to 200 GB.	From Several hundred GB to Terabytes.

Expense	Magnetic tapes are less expensive.	Magnetic disk is more expensive.

CD-ROM

Introduction to CD-ROM

1. A Short History of CD-ROM
2. CD-ROM as a File Structure Problem

Stands for "Compact Disc Read-Only Memory." A CD-ROM is a CD that can be read by a computer with an optical drive. The "ROM" part of the term means the data on the disc is "read-only," or cannot be altered or erased. Because of this feature and their large capacity, CD-ROMs are a great media format for retail software. The first CD-ROMs could hold about 600 MB of data, but now they can hold up to 700 MB. CD-ROMs share the same technology as audio CDs, but they are formatted differently, allowing them to store many types of data.

1. A Short History of CD-ROM
   CD-ROM is the offspring of videodisc technology developed in the late 1960s and early 1970s, before the advent of the home VCR. The goal was to store movies on disc. The consumer products industry spent a great deal of money developing the different technologies, including several approaches to optical storage, then spent years fighting over which approach should become standard.
   The surviving format is one called LaserVision. The LaserVision format supports recording in both a constant linear velocity (CLV) format that maximizes storage capacity and a constant angular velocity (CAV) format that enables fast seek performance. By using the CAV format to access individual video ‘frames quickly, a number of organizations, including the MIT Media Lab, produced prototype interactive video discs that could be used to teach and entertain.
   In the early 1980s, a number of firms began looking at the possibility of storing digital, textual information on LaserVision discs. LaserVision stores data in an analog form; it is, after all, storing an analog video signal. Different firms came up with different ways of encoding digital information in analog form so it could be stored on the disc.
   This time they worked with other players in the consumer products industry to develop a licensing system that resulted in the emergence of CD audio as a broadly accepted, standard format as soon as the first discs and players were introduced. CD audio appeared in the

United States in early 1984. CD-ROM, which is a digital data format built on top of the CD audio standard, emerged shortly thereafter. The first commercially available CD-ROM drives appeared in 1985.
In 1985 the firms emerging from the videodisc/digital data industry, all of which were ‘relatively small, called together many of the much larger firms moving into the CD-ROM industry to begin work on a standard file system that would be built on top of the CD-ROM format. In a rare display of cooperation, the different firms, large and small, worked out the main features of a file system standard by early summer of 1986; that work has become an official international standard for organizing files on CD-ROM.

2. CD-ROM as a File Structure Problem
CD-ROM presents interesting file structure problems because it is a medium with great strengths and weaknesses. The strengths of CD-ROM include its high storage capacity, its inexpensive price, and its durability. The key weakness is that seek performance on a CD-ROM is very slow, often taking from a half second to a second per seek.

Physical Organization of CD-ROM

1. Reading Pits and Lands
2. CLV instead of CAV
3. Addressing
4. Structure of a Sector

In this Physical organization of CD-ROM having 4 types.\

1. Reading Pits and Lands
CD-ROM is a descendent of CD Audios. i.e., listening to music is sequential and does not require fast random access to data.
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 stamped copy of the disc, we focus a beam of laser light on the track as it moves under the optical pickup. The pits scatter the light, but the lands reflect most of it back to the pickup. This alternating pattern of high- and low-intensity reflected light is the signal used to reconstruct the original digital information.
1’s are represented by the transition from pit to land and back again. 0’s are represented by the amount of time between transitions. The longer between transitions, the more 0s we have. Given this scheme, it is not possible to have 2 adjacent 1s: 1s are always separated

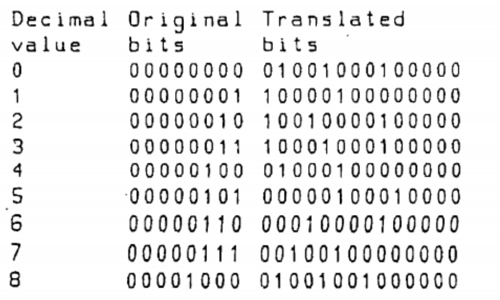
by 0s. As a matter of fact, because of physical limitations, there must be at least two 0s between any pair of 1s.

Raw patterns of 1s and 0s have to be translated to get the 8-bit patterns of 1s and 0s that form the bytes of the original data.

EFM encofing (Eight to Fourteen Modulations) turns the original 8 bits of data into 14 expanded bits that can be represented in the pits and lands on the disk.

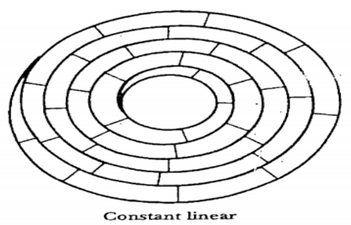
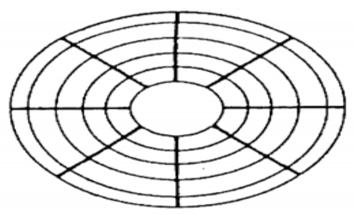
Since 0s are represented by the length of time between transition, the disk must be rotated at a precise and constant speed. This affects the CD-ROM drive's ability . to seek quickly.

Figure 3.12 A portion of the EFM encoding table.



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

2. CLV instead of CAV

Sl.No.	Constant Linear Velocity(CLV)	Constant Angular Velocity(CAV)
1.	Tracks are in spiral format	Tracks are in circular format
2.	Generally used to store audio data (songs)	Used to store normal data(files)
3.	Rate of spinning of disk would change as data is read from outer sector towards inner sector	Rate of spinning remains constant as data is read from outer sector towards inner sector
4.	All the sectors contain equal amount of data so no wasteage of memory	Sector in outer track contain less data compared to inner track so some memory is wasted.
5.	Seek performance is poor	Seek performance is relatively good
6.	 Constant linear velocity	 Constant angular velocity

3. Addressing
   Different from the “regular” disk method. Each second of playing time on a CD is divided into 75 sectors. Each sector holds 2 Kilobytes of data. Each CD-ROM contains at least one hour of playing time.
   ==> The disc is capable of holding at least 60 min * 60 sec/min * 75 sector/sec * 2 Kilobytes/sector = 540, 000 Kbytes. Often, it is actually possible to store over 600, 000 KBytes. Sectors are addressed by min:sec:sector e.g., 16:22:34.

4. Structure of a Sector
   When we want to store sound, we need to convert a wave pattern into digital form. Figure 3.14 shows a wave. At any given point in time, the wave has a specific amplitude.

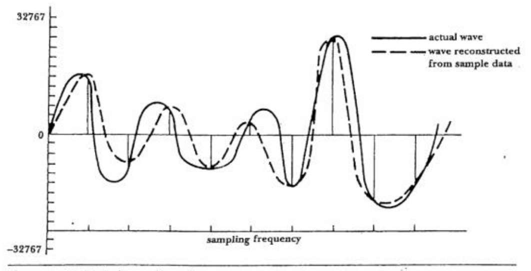


Figure 3.14 Digital sampling of a wave.

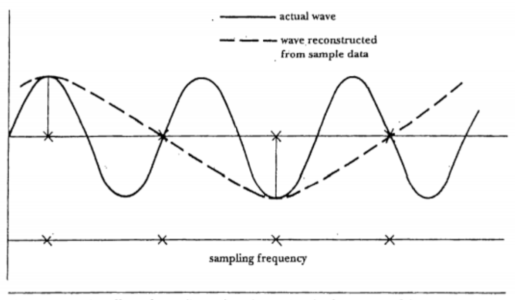
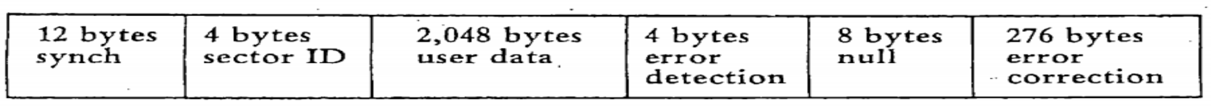


Figure 3.15 The effect of sampling at less than twice the frequency of the wave.

Figure 3.15 The effect of sampling at less than twice the frequency of the wave. CD audio uses 16 bits to store each amplitude measurement; that means that the “ruler” we use to measure the height of the wave has 65,536 different gradations. To approximate a wave accurately through digital sampling, we need to take the samples at a rate that is more than twice as frequent as the highest frequency we want to capture. This makes sense if you look at the wave in Fig. 3.15.



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

Figure 3.16 Structure of a CD-ROM sector.

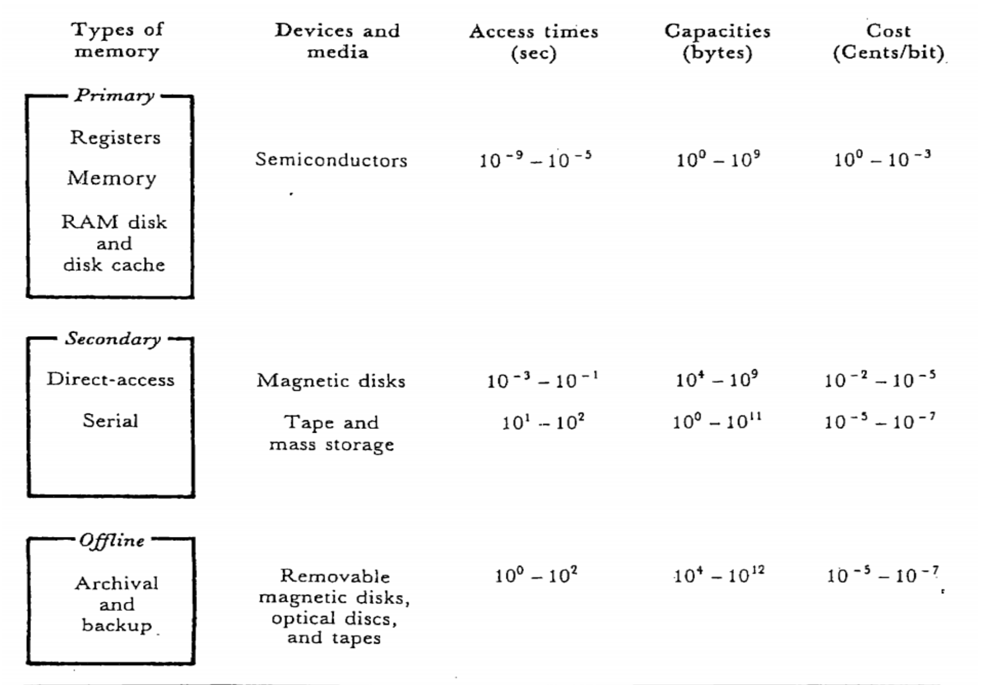
CD-ROM divides up this “raw” sector storage as shown in Fig. 3.16 to provide 2 kilobytes of user data storage, along with addressing information, error detection, and error correction information. The error correction information is necessary because, although CD audio contains redundancy for error correction, it is not adequate to meet computer data storage needs. The audio error correction would result in an average of one incorrect byte for every two discs. The additional error correction information stored within the 2352-byte sector decreases this error rate to 1 uncorrectable byte in every twenty thousand discs.

CD-ROM Strengths & Weaknesses

1. Seek Performance – the chief weakness of CD-ROM is the random access performance. The current magnetic disk technology takes 30 msec to access the data, while CD-ROM would take 500msec.
2. Data Transfer Rate – A CD-ROM reads 70 sectors per sec this data transfer rate definition cannot be changed. It is a modest transfer rate about five times faster than the transfer rate for floppy disks. This transfer rate is fast enough relative to CD-ROM's seek performance.
3. Storage Capacity – A CD-ROM holds more than 600 megabytes of data. This is a huge amount of memory for storage. Many typical text databases and document collections published on CD-ROM use only a fraction of memory with such large capacity it enables us to build indexes and other structures that can help us overcome the disadvantage of seek performance of CD-ROM.
4. Read Only Access – CD-ROM is a publishing medium, a storage device that cannot be changed after manufacture. The major advantage is that we never have to worry about updating. This simplifies the file structure design.
5. Asymmetric Writing & Reading – with CD-ROM we create files that are placed on the disc once , then we distribute the disc, which is accessed millions of times with CD-ROM we make investment in intelligent carefully designed file structures only once, users can enjoy the benefits of this investment again and again.

Storage as a Hierarchy

The best mixture of devices for a computing system depends on the needs of the system's users, we can imagine any computing system as a hierarchy of storage devices of different speed, capacity, and cost. Figure 3.17 summarizes the different types of storage found at different levels in such hierarchies and shows approximately how they compare in terms of access time, capacity, and cost.



Types of memory	Devices and media	Access times (sec)	Capacities (bytes)	Cost (Cents/bit)
Primary				
Registers, Memory, RAM disk and disk cache	Semiconductors	$10^{-9} - 10^{-5}$	$10^{0} - 10^{9}$	$10^{0} - 10^{-3}$
Secondary				
Direct-access	Magnetic disks	$10^{-3} - 10^{-1}$	$10^{4} - 10^{9}$	$10^{-2} - 10^{-5}$
Serial	Tape and mass storage	$10^{1} - 10^{2}$	$10^{0} - 10^{11}$	$10^{-5} - 10^{-7}$
Offline				
Archival and backup	Removable magnetic disks, optical discs, and tapes	$10^{0} - 10^{2}$	$10^{4} - 10^{12}$	$10^{-5} - 10^{-7}$

Figure 3.17 Approximate comparisons of types of storage.

A Journey of a Byte

1. The file Manager
2. The I/O Buffer
3. The byte leaves memory: the I/O processor & disk controller

What happens when the program statement: write(textfile, ch, 1) is executed ?

Part that takes place in memory:

Statement calls the Operating System (OS) which overseas the operation.

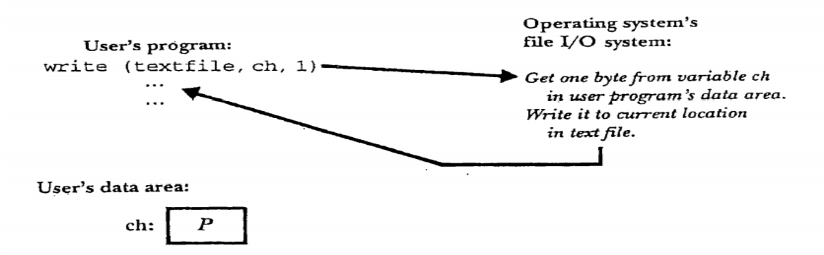
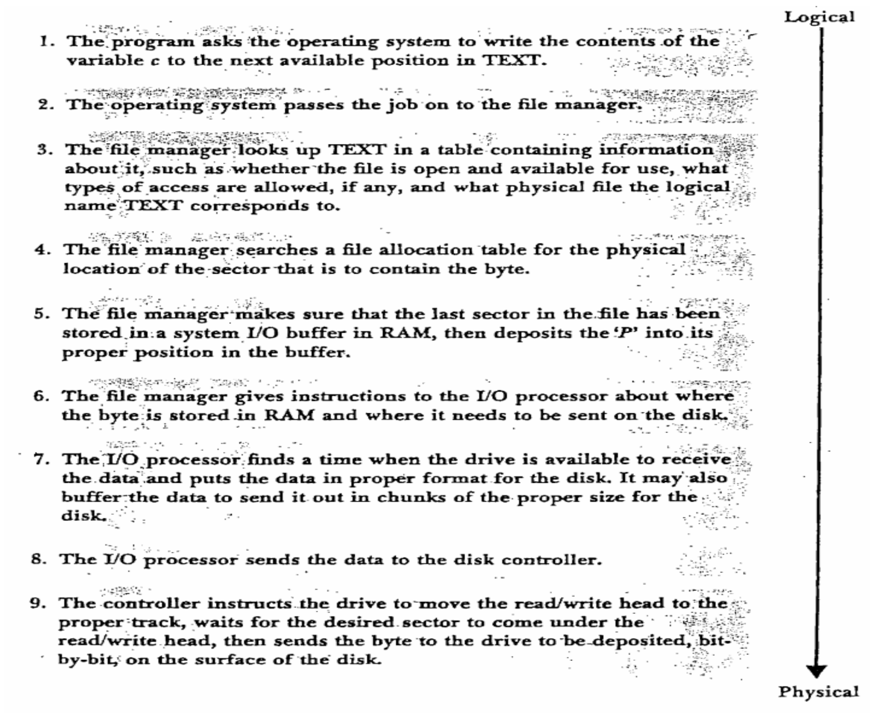

User's program:
write (textfile, ch, 1)
...
...
Operating system's file I/O system:
Get one byte from variable ch in user program's data area. Write it to current location in text file.
User's data area:
ch:
P

Figure 3.18 The write statement tells the operating system to send one character to disk and gives the operating system the location of the character. The operating system takes over the job of writing, and then returns control to the calling program.

In this journey of a byte there will be a 3 types.

1. The file Manager

File manager (Part of the OS that deals with I/O).



1. The program asks the operating system to write the contents of the variable *c* to the next available position in TEXT.
2. The operating system passes the job on to the file manager.
3. The file manager looks up TEXT 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 TEXT 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, then deposits the '*P*' into its proper position in the buffer.
6. The file manager gives instructions to the I/O processor about 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, then sends the byte to the drive to be deposited, bit-by-bit, on the surface of the disk.

2. The I/O Buffer

The file manager determines whether the sector that is to contain the P is already in memory or needs to be loaded into memory. If the sector needs to be loaded, the file manager must find an available system I/O buffer space for it and then read it from the disk. Once it has the sector in a buffer in memory, the file manager can deposit the P into its proper position in the buffer (Fig. 3.20). The system 1/O buffer allows the file manager to read and write data in sector-sized or block-sized units. In other words, it enables the file manager to ensure that the organization of data in memory conforms to the organization it will have on the disk.

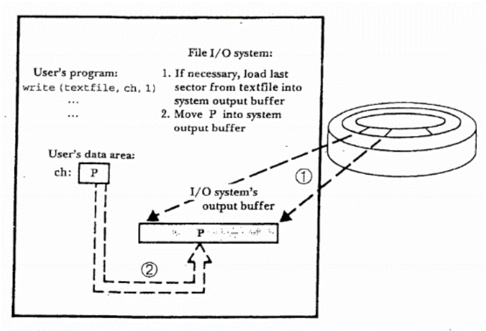

File I/O system:
1. If necessary, load last sector from textfile into system output buffer
2. Move P into system output buffer
User's program:
write (textfile, ch, 1)
User's data area:
ch: P
I/O system's output buffer
P
①
②

Figure 3.20 The file manager moves P from the program's data area to a system output buffer where it may join other bytes headed for the same place on the disk. If necessary, the file manager may have to load the corresponding sector from the disk into the system output buffer.

3. The byte leaves memory: the I/O processor & disk controller
   I/O Processor: Wait for an external data path to become available (CPU is faster than data-paths ==> Delays)
   Disk Controller:
   - I/O Processor asks the disk controller if the disk drive is available for writing.
   - Disk Controller instructs the disk drive to move its read/write head to the right track and sector.
   - Disk spins to right location and byte is written.

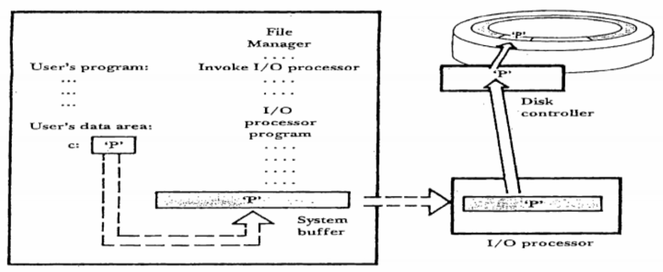


Figure 3.21 The file manager sends the I/O processor instructions in the form of an I/O processor program. The I/O processor gets the data from the system buffer, prepares it for storing on the disk, then sends it to the disk controller, which deposits it on the surface of the disk.

Buffer Management

1. Buffer Bottlenecks.
2. Buffering Strategies.

In this buffer management concept there are two types let's in detail.

1. Buffer Bottlenecks.
   that a file manager allocates I/O buffers that are big enough to hold incoming data, but we have said nothing so far about how many buffers are used. In fact, it is common for file managers to allocate several buffers for performing I/O. To understand the need for several system buffers, consider what happens if a program is performing both input and output on one character at a time and only one I/O buffer is available. When the program asks for its first character, the I/O buffer is loaded with the sector containing the character, and the character is transmitted to the program. If the program then decides to output a -inaracter, the I/O buffer is filled with the sector into which the output character needs to go, destroying its original contents. Then when the next input character is needed, the buffer contents have to be written to disk to make room for the (original) sector containing the second input character, and so on.

2. Buffering Strategies.
   What happens to data travelling between a program's data area and secondary storage?
   The use of Buffers: Buffering involves working with a large chunk of data in memory so the number of accesses to secondary storage can be reduced.
   - Multiple Buffering
     Assume a program that is writing to a disk, writing or reading from a disk is the consuming. To utilize CPU efficiently the following method can be used while writing contents to disk. We can maintain two buffers, the contents from one buffer can be written to the disk while the other buffer can be filled. Once the contents are written the jobs of the buffer can be exchanged. This process of swapping the roles of two buffer after each output or input is called double buffering.

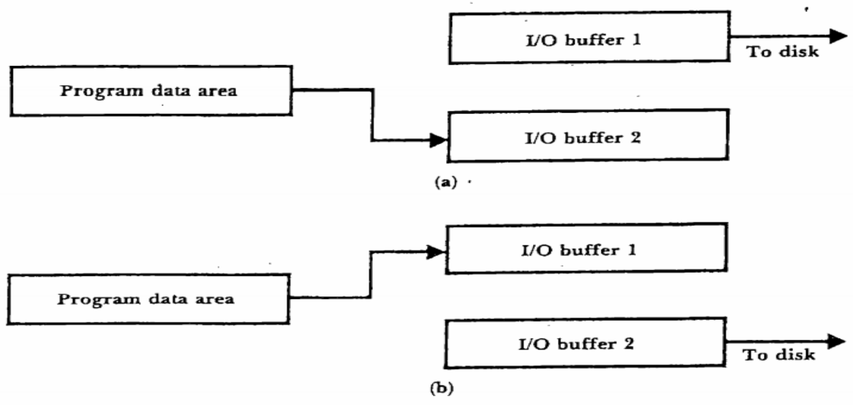


Figure 3.22 Double buffering: (a) the contents of system I/O buffer 1 are sent to disk while I/O buffer 2 is being filled; and (b) the contents of buffer 2 are sent to disk while I/O buffer 1 is being filled.

- Move mode and locate mode
  For the program to use the contents of system buffer, it needs to be copied to the programs data area. The time taken to move the contents would be substantial. This way of handling buffer data is called move mode as it involves moving chunks of data from one place in memory to another before it is accessed. There are two ways that move mode can be avoided:
  The file manager can directly perform I/O operation between secondary storage and program data area, no extra move is required.
  The file manager can provide the location where the system buffers are present by presenting pointers.
  Both the techniques of handling buffer is called locate mode. When locate mode is used program is able to operate directly on data eliminating the need to transfer data between I/O buffer and program buffer.

- Scatter/Gather I/O
  While reading a file with many blocks and each block many consists of a header followed by data. In order to process the data we need to remove header and move it to separate buffer, remove data and move it to separate buffer with scatter input, a single read call identifies not one, but a collection of buffer into which data from a single block is to be scattered. The converse of scatter input is gather output with gather output several buffers can be gathered and written with a single write call; this avoids the need to copy them to a single output buffer.

I/O in Unix

1. The Kernel
   the process of transmitting data from a program to an external device can be described as proceeding through a series of layers. The topmost layer deals with data in logical, structural] terms. This reflects the view that an application has of what goes into a file. The layers that follow collectively carry out the task of turning the logical object.into a collection of bits on a physical device, Likewise, the topmost'I/O layer in Unix deals with data primarily in logical terms. This layer in Unix consists of processes that impose certain logical views on files. Processes are associated with solving some problem, such as counting the words in the file or searching for somebody's address. The components of the kernel that do I/O are illustrated in

Figure 3.23. The kernel views all I/O as operating on a sequence of bytes, so once we pass control to the kernel all assumptions about the logical view of a file are gone. The decision to design Unix this way—to make all operations below the top layer independent of an application's logical view of a file—is unusual. It is also one of the main attractions in choosing Unix as a focus for this text, for Unix lets us make all of the decisions about the logical structure of a file, imposing no restrictions on how we think about the file beyond the fact that it must be built from a sequence of bytes.

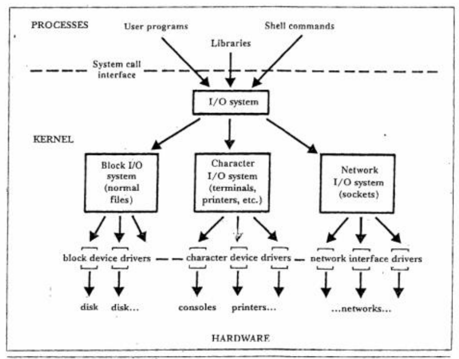


Figure 3.23 Kernel I/O structure.

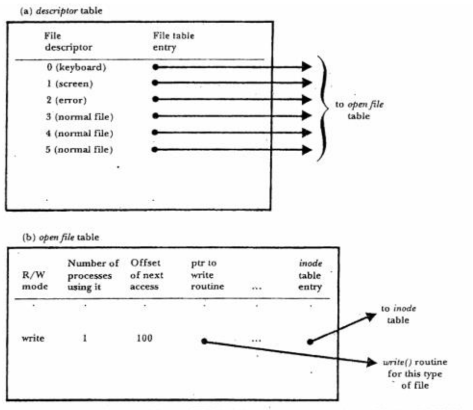


Figure 3.24 Descriptor table and open file table.

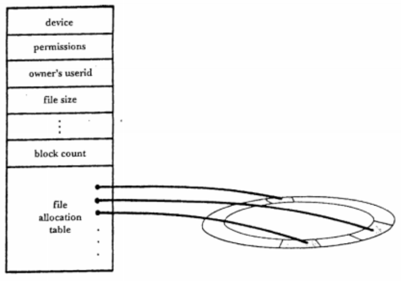


Figure 3.25 An inode. The inode is the data structure used by Unix to describe the file. It includes the device containing the file, permissions, owner and group IDs, and file allocation table, among other things.

When your program executes a system call such as '

write (fd, &ch, 1);

the kernel is invoked immediately. The routines that let processes communicate directly with the kernel make up the system call interface. In this case, the system call instructs the kernel to write a character to a file. The kernel I/O system begins by connecting the file descriptor (£d) in your program to some file or device in the file system. It does this by proceeding through a series of four tables that enable the kernel to find its way from a process to the places on the disk that will hold the file they refer to. The four tables are

- file descriptor table;

- an open file table, with information about open files;
- a file allocation table, which is part of a structure called an index node;
- a table of index nodes, with one entry for each file in use.

Although these tables are managed by the kernel's I/O system, they are, in a sense, "owned" by different parts of the system:

- The file descriptor table is owned by the process (your program).
- The open file table and index node table are owned by the kernel.
- The index node is part of the file system.

The four tables are invoked in turn by the kernel to get the information it needs to write to your file on disk. Let's see how this works by looking at the functions of the tables. The file descriptor table (Fig. 3.24a) is a simple table that associates each of the file descriptors used by a process with an entry in another table, the open file table. Every process has its own descriptor table, which includes entries for all files it has opened, including the "files" stdin, stdout, and stderr. The open file table (Fig. 3.24b) contains entries for every open file. Every time a file is opened or created, a new entry is added to the open file table. These entries are called file structures, and they contain important information about how the corresponding file is to be used, such as the read/write mode used when it was opened. The information is found in an index node, more commonly referred to as an inode (Fig. 3.25). An inode is a more permanent structure than an open file table's file structure. A file structure exists only while a file is open for access, but an inode exists as long as its corresponding file exists. For this reason, a file's inode is kept on disk with the file (though not physically adjacent to the file).

2. Linking File Names to File
   It is instructive to look a little more closely at how a file name is linked to the corresponding file. All references to files begin with a directory, for it is in directories that file names are kept. In fact, a directory is just a small file that contains, for each file, a file name together with a pointer to the file's inode on disk.!2 This pointer from a directory to the inode of a file is called a hard link.
3. Normal Files, Special Files, & Sockets
   The "everything is a file" concept in Unix works only when we recognize that some files are quite a bit different from others. Can observe in previous Fig. 3.23 that the kernel distinguishes among three different types of files. Normal files are the files that this text is about. Special files almost always represent a Stream of characters and control signals that drive some device, such as a line printer or a graphics device. The first three file descriptors

in the descriptor table (Fig. 3.24a) are special files. Sockets are abstractions that serve as endpoints for interprocess communication.

4. Block I/O
   The three different types of files access their respective devices via three different I/O systems: the block I/O system, the character I/O system, and the network I/O system. Henceforth we ignore the second and third categories, since it is normal file I/O that we are most concerned with in this text. The block I/O system is the Unix counterpart of the file manager in the journey of a byte. It concerns itself with how to transmit normal file data, viewed by the user as a sequence of bytes, onto a block-oriented device like a disk or tape.
5. Device Drivers
   For each peripheral device there is a separate set of routines, called a device driver, that performs the I/O between the I/O buffer and the device. A device driver is roughly equivalent to the I/O processor program described in the journey of a byte.
6. The Kernel & File Systems
   The Unix concept of a file systent. A Unix file system is a collection of files, together with secondary information about the files in the system. A file system includes the directory structure, the directories, ordinary files, and the inodes that describe the files.
7. Magnetic Tape & Unix
   Important as it 1s to computing, magnetic tape is somewhat of an orphan in the Unix view of I/O. A magnetic tape unit has characteristics similar to both block I/O devices (block oriented) and character devices (primarily used for sequential access) but does not fit nicely into either category. Character devices read and write streams of data, not blocks, and block devices in general access blocks randomly, not sequentially.

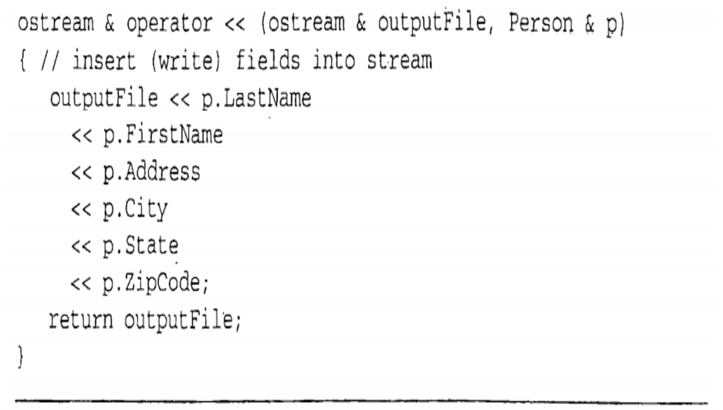
Fundamental File Structure Concepts, Managing Files of Records

Field & Record Organization

1. A Stream File
2. Field Structures
3. Reading a Stream of Fields
4. Record Structures
5. A record structure that uses a length indicator
6. Mixing numbers & characters: use of a file dump

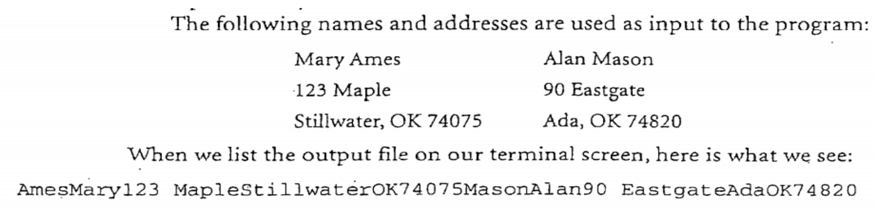
In this field & record organization there are 6 types.

1. A Stream File
   The the objects we wish to store contain name and address information about a collection of people. “Using Objects in C++,” to store information about individuals, Figure 4.1 (and filewritestr.cpp) gives a C++ function (operator <<) to write the fields of a Person toa file as a stream of bytes.



```
ostream & operator << (ostream & outputFile, Person & p)
{ // insert (write) fields into stream
   outputFile << p.LastName
     << p.FirstName
     << p.Address
     << p.City
     << p.State
     << p.ZipCode;
   return outputFile;
}
```

Figure 4.1 Function to write (<<) a Person as a stream of bytes.



The following names and addresses are used as input to the program:

Mary Ames	Alan Mason
123 Maple	90 Eastgate
Stillwater, OK 74075	Ada, OK 74820

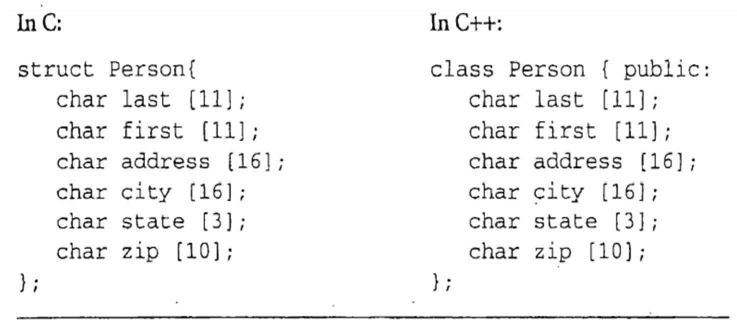
When we list the output file on our terminal screen, here is what we see:

```
AmesMary123 MapleStillwaterOK74075MasonAlan90 EastgateAdaOK74820
```

2. Field Structures

   Various ways of adding structure to files to maintain the identity of fields.

   - Method 1 – Fix the length of fields.
   - Method 2 – Begin each field with a length indicator.
   - Method 3 – Separate the fields with delimiter.
   - Method 4 – Use a "Keyword = Value" expression to identify fields.



In C:

```
struct Person{
   char last [11];
   char first [11];
   char address [16];
   char city [16];
   char state [3];
   char zip [10];
};
```

In C++:

```
class Person { public:
   char last [11];
   char first [11];
   char address [16];
   char city [16];
   char state [3];
   char zip [10];
};
```

Figure 4.2 Definition of record to hold person information.

1. Method 1 – Fix the length of fields

   In this method we organise fields by limiting the maximum size of each field. This is called as fixed length fields. As the fields are of predictable lengths, we can pull them back out of the file simply by counting our way to the file simply by counting our way to the end of the file.

   The disadvantage of the method

   - Padding is required to bring the fields upto a fixed length makes the file much larger.
   - Data might be lost if it is too long to fit into the allocated amount of space.

Due to the above mentioned problems the fixed field approach to structing data is inappropriate for data that contains a large amount of variability in the length of fields. Fixed length field is a very good solution if the fields are already fixed in length or there is very little variation in field lengths.

2. Method 2 – Begin each field with a length indicator

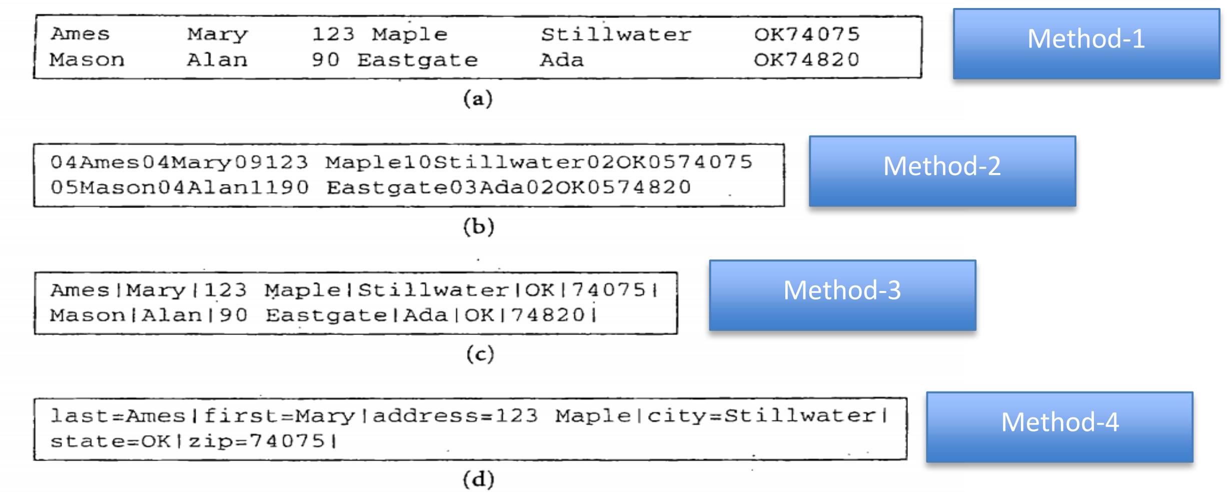
   The end of the field can be reached by knowing the length of the field. The length of each field can be stored at the begin of each field. These fields are called length based fields.

3. Method 3 – separate the fields with delimiters

   The identity of fields can be preserved by separating them with delimiter. We need to choose some special characters that will not appear in a field and insert that delimiter into the file after writing each field.

4. Method 4 – use a "keyword = value" expression to identify fields

   It is the first structure in which a field provides information about itself. Such self describing structures are very useful tools for organizing files in many application Using this structure it is easy to tell what are all fields present in a file. It is easy to identify the missing fields. The main disadvantage of this structure is the 50% or more of the files space could be taken up by the keywords.



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

(a)

```
04Ames04Mary09123 Maple10Stillwater02OK0574075
05Mason04Alan1190 Eastgate03Ada02OK0574820
```

(b)

```
Ames|Mary|123 Maple|Stillwater|OK|74075|
Mason|Alan|90 Eastgate|Ada|OK|74820|
```

(c)

```
last=Ames|first=Mary|address=123 Maple|city=Stillwater|
state=OK|zip=74075|
```

(d)

Method-1
Method-2
Method-3
Method-4

Figure 4.3 Four methods for organizing fields within records. (a) Each field is of fixed length. (b) Each field begins with a length indicator. (c) Each field ends with a delimiter |. (d) Each field is identified by a key word.

3. Reading a Stream of Fields

   It contains the implementation of the extraction operation. Extensive use is made of the istream method get line. The arguments to get line are a character array to hold the string, a maximum length, and a delimiter. Get line reads up to the first occurrence of the delimiter, or the end-of-line, whichever comes first.

```
istream & operator >> (istream & stream, Person & p)
{ // read delimited fields from file
   char delim;
   stream.getline(p.LastName, 30,'|');
   if (strlen(p.LastName)==0) return stream;
   stream.getline(p.FirstName,30,'|');
   stream.getline(p.Address,30,'|');
   stream.getline(p.City, 30,'|');
   stream.getline(p.State,15,'|');
   stream.getline(p.ZipCode,10,'|');
   return stream;
}
```

Figure 4.4 Extraction operator for reading delimited fields into a `Person` object.

```
Last Name   'Ames'
First Name  'Mary'
Address     '123 Maple'
City        'Stillwater'
State       'OK'
Zip Code    '74075'
Last Name   'Mason'
First Name  'Alan'
Address     '90 Eastgate'
City        'Ada'
State       'OK'
Zip Code    '74820'
```

4. Record Structures
   - Method 1 – Make Records a Predictable number of bytes
   - Method 2 – Make records a predictable number of fields
   - Method 3 – Begin each record with a length indicator
   - Method 4 – Use an index to keep track of addresses
   - Method 5 – Place a delimiter at the end of each record

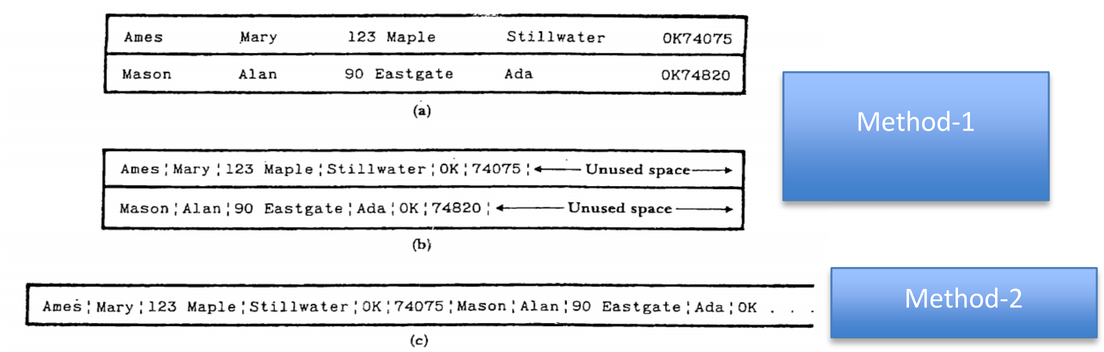

Ames Mary 123 Maple Stillwater OK74075
Mason Alan 90 Eastgate Ada OK74820
(a)
Ames¦Mary¦123 Maple¦Stillwater¦OK¦74075¦ Unused space
Mason¦Alan¦90 Eastgate¦Ada¦OK¦74820¦ Unused space
(b)
Ames¦Mary¦123 Maple¦Stillwater¦OK¦74075¦Mason¦Alan¦90 Eastgate¦Ada¦OK . . .
(c)
Method-1
Method-2

Figure 4.5 Three ways of making the lengths of records constant and predictable. (a) Counting bytes: fixed-length records with fixed-length fields. (b) Counting bytes: fixed-length records with variable-length fields. (c) Counting fields: six fields per record.

Method 1 – Make Records a Predictable number of bytes
A fixed length record file is one in which each records contains the same number of bytes. The size can be determined by adding the maximum space occupied by each field. Though the size of the entire record is fixed, the fields inside the record can be varying size or of fixed size.

Method 2 – Make records a predictable number of fields
In this method the number of fields in each record is fixed. This method is also called as fixed field count structure. Assuming that every record has six fields then each record can be displayed by counting the fields modulo six.

Method 3 – Begin each record with a length indicator
In this method every record would begin with integer. The first field would contain an integer value which would indicate the length of record in bytes. This method is commonly used in handling variable length records.

Method 4 – Use an index to keep track of addresses
In this method we use an index to keep a byte offset for each record in the original file. The byte offset allows us to find the beginning of each successive record and compute the length of each record. We look up the position of the record in the index then seek to the record in the data file.
Method 5 – Place a delimiter at the end of each record
In this method at the end of each record we place a special character which we encountered indicates the end of the record. The delimiter character must not get in the way of processing.

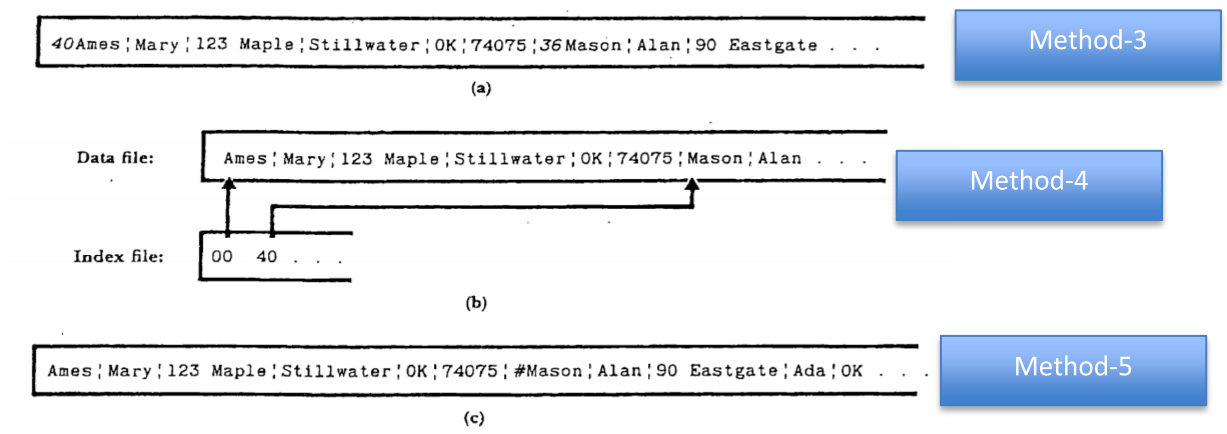

40Ames¦Mary¦123 Maple¦Stillwater¦OK¦74075¦36Mason¦Alan¦90 Eastgate . . .
(a)
Method-3
Data file:
Ames¦Mary¦123 Maple¦Stillwater¦OK¦74075¦Mason¦Alan . . .
Method-4
Index file:
00 40 . . .
(b)
Ames¦Mary¦123 Maple¦Stillwater¦OK¦74075¦#Mason¦Alan¦90 Eastgate¦Ada¦OK . . .
Method-5
(c)

Figure 4.6 Record structures for variable-length records. (a) Beginning each record with a length indicator. (b) Using an index file to keep track of record addresses. (c) Placing the delimiter # at the end of each record.

5. A record structure that uses a length indicator

```
const int MaxBufferSize = 200;
int WritePerson (ostream & stream, Person & p)
{  char buffer [MaxBufferSize]; // create buffer of fixed size
   strcpy(buffer, p.LastName); strcat(buffer,"|");
   strcat(buffer, p.FirstName); strcat(buffer,"|");
   strcat(buffer, p.Address);  strcat(buffer,"|");
   strcat(buffer, p.City);  strcat(buffer,"|");
   strcat(buffer, p.State);  strcat(buffer,"|");
   strcat(buffer, p.ZipCode);  strcat(buffer,"|");
   short length=strlen(buffer);
   stream.write (&length, sizeof(length)); // write length
   stream.write (&buffer, length);
}
```

Figure 4.7 Function `WritePerson` writes a variable-length, delimited buffer to a file.

1- writestr

2

```
40 Ames|Mary|123 Maple|Stillwater|OK|74075|36
Mason|Alan|90 Eastgate|Ada|OK|74820
```

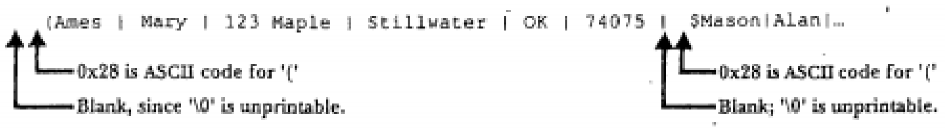
Figure 4.8 Records preceded by record-length fields in character form.

```
int ReadVariablePerson (istream & stream, Person & p)
{ // read a variable sized record from stream and store it in p
   short length;
   stream . read (&length, sizeof(length));
   char * buffer = new char[length+1];// create buffer space
   stream . read (buffer, length);
   buffer [length] = 0; // terminate buffer with null
   istrstream strbuff (buffer); // create a string stream
   strbuff >> p; // use the istream extraction operator
   return 1;
}
```

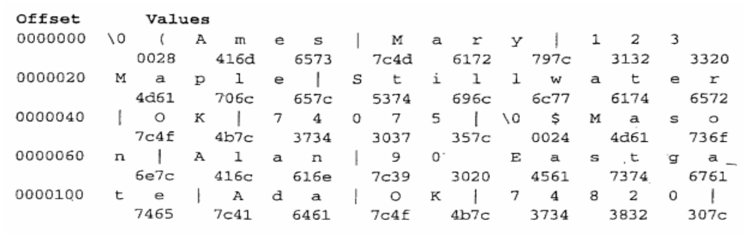
Figure 4.9 Function `ReadVariablePerson` that reads a variable-sized `Person` record.

3- readstr

6. Mixing numbers & characters: use of a file dump

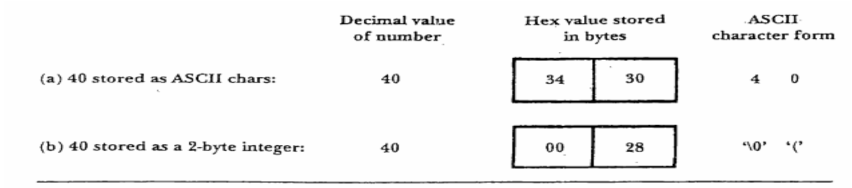

(Ames | Mary | 123 Maple | Stillwater | OK | 74075 | $Mason|Alan|...
0x28 is ASCII code for '('
Blank, since '\0' is unprintable.
0x28 is ASCII code for '('
Blank; '\0' is unprintable.

```
od -xc filename
```



```
Offset      Values
0000000  \0   (   A   m   e   s   |   M   a   r   y   |   1   2   3
             0028    416d    6573    7c4d    6172    797c    3132    3320
0000020   M   a   p   l   e   |   S   t   i   l   l   w   a   t   e   r
             4d61    706c    657c    5374    696c    6c77    6174    6572
0000040   |   O   K   |   7   4   0   7   5   |  \0   $   M   a   s   o
             7c4f    4b7c    3734    3037    357c    0024    4d61    736f
0000060   n   |   A   l   a   n   |   9   0       E   a   s   t   g   a
             6e7c    416c    616e    7c39    3020    4561    7374    6761
0000100   t   e   |   A   d   a   |   O   K   |   7   4   8   2   0   |
             7465    7c41    6461    7c4f    4b7c    3734    3832    307c
```

http://www.csgnetwork.com/asciiset.html



	Decimal value of number	Hex value stored in bytes		ASCII character form	
(a) 40 stored as ASCII chars:	40	34	30	4	0
(b) 40 stored as a 2-byte integer:	40	00	28	'\0'	'('

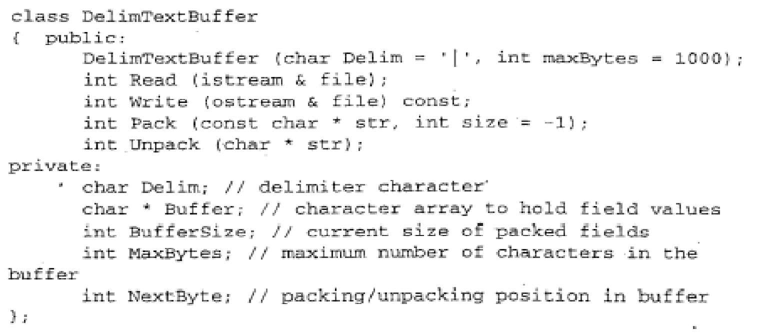
Figure 4.10 The number 40, stored as ASCII characters and as a short integer.

Using Classes to Manipulate Buffers

1. Buffer Class for Delimited Text Fields
2. Extending Class Person with Buffer Operations
3. Buffer Classes for Length-Based and Fixed-Length Fields

In this concept there are 3 types,

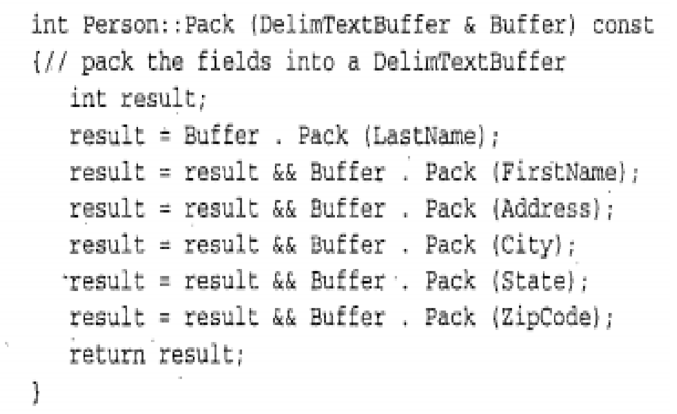
1. Buffer Class for Delimited Text Fields
   The first buffer class, DelimTextBuffer, supports variable-length buffers whose fields are represented as delimited text. A part of the class definition is given as Fig. 4.11. The full definition is in filedeltext.h The following code segment declares objects of class Person and class DelimTextBuf fer, packs the person into the buffer, and writes the buffer to a file:



```
class DelimTextBuffer
{  public:
     DelimTextBuffer (char Delim = '|', int maxBytes = 1000);
     int Read (istream & file);
     int Write (ostream & file) const;
     int Pack (const char * str, int size = -1);
     int Unpack (char * str);
private:
     char Delim; // delimiter character
     char * Buffer; // character array to hold field values
     int BufferSize; // current size of packed fields
     int MaxBytes; // maximum number of characters in the
buffer
     int NextByte; // packing/unpacking position in buffer
};
```

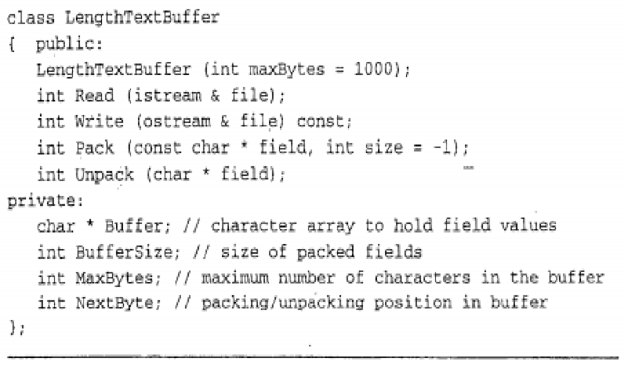
Figure 4.11 Main methods and members of class DelimTextBuffer.

2. Extending Class Person with Buffer Operations
   The buffer classes have the capability of packing any number and type of values, but they do not record how these values are combined to make objects. In order to pack and unpack a buffer for a Person object, for instance, we have to specify the order in which the members of Person are packed and unpacked. It included operations for packing and unpacking the members of Person objects in insertion (<<) and extraction (>>) operators.



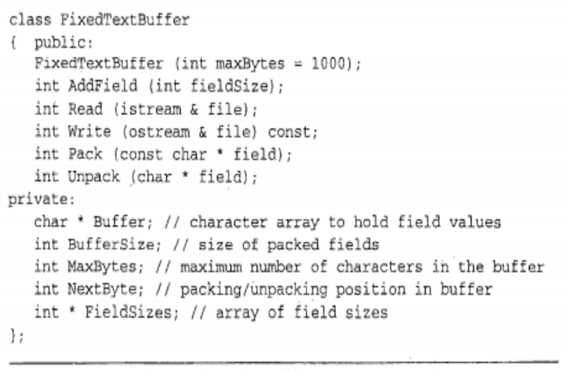
```
int Person::Pack (DelimTextBuffer & Buffer) const
{// pack the fields into a DelimTextBuffer
   int result;
   result = Buffer . Pack (LastName);
   result = result && Buffer . Pack (FirstName);
   result = result && Buffer . Pack (Address);
   result = result && Buffer . Pack (City);
   result = result && Buffer . Pack (State);
   result = result && Buffer . Pack (ZipCode);
   return result;
}
```

3. Buffer Classes for Length-Based and Fixed-Length Fields
   Representing records of length-based fields and records of fixed-length fields requires a change in the implementations of the Pack and Unpack methods of the delimited field class, but the class definitions are almost exactly the same. The main members and methods of class LengthTextBuf fer are given in Fig. 4.12. The full class definition and method implementation are given in lentext.h and lentext.cpp



```
class LengthTextBuffer
{  public:
   LengthTextBuffer (int maxBytes = 1000);
   int Read (istream & file);
   int Write (ostream & file) const;
   int Pack (const char * field, int size = -1);
   int Unpack (char * field);
private:
   char * Buffer; // character array to hold field values
   int BufferSize; // size of packed fields
   int MaxBytes; // maximum number of characters in the buffer
   int NextByte; // packing/unpacking position in buffer
};
```

Figure 4.12 Main methods and members of class LengthTextBuffer.



```
class FixedTextBuffer
{  public:
   FixedTextBuffer (int maxBytes = 1000);
   int AddField (int fieldSize);
   int Read (istream & file);
   int Write (ostream & file) const;
   int Pack (const char * field);
   int Unpack (char * field);
private:
   char * Buffer; // character array to hold field values
   int BufferSize; // size of packed fields
   int MaxBytes; // maximum number of characters in the buffer
   int NextByte; // packing/unpacking position in buffer
   int * FieldSizes; // array of field sizes
};
```

Figure 4.13 Main methods and members of class FixedTextBuffer.

Using Inheritance in the C++ Stream Classes

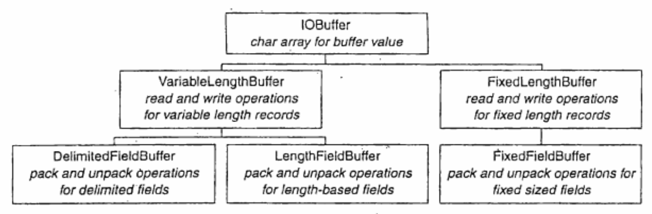
1. Inheritance in the C++ Stream Classes
2. A Class Hierarchy for Record Buffer Objects

In this concept there are two types,

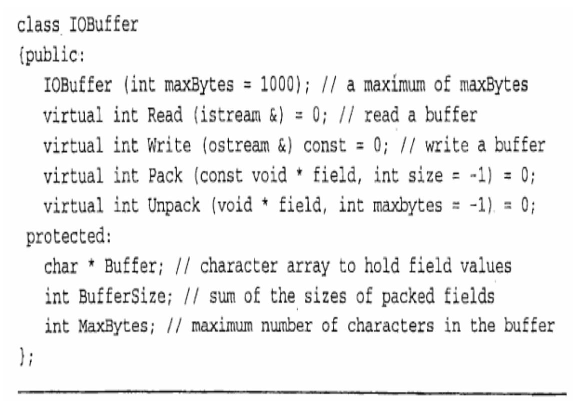
1. Inheritance in the C++ Stream Classes
   The main advantage of inheritance is that it helps us to reuse the code. C++ incorporates inheritance to allow multiple classes to share members and method. One or more base class define member and methods which are then used by subclasses. Actually the fstream class is defined in the same manner fstream inherits properties of iostream. Iostream inherits properties from istream and ostream. Fstream also inherits properties from fstream base. Fstream class uses multiple inheritance i.e a classcan have more than one base class. In case of multiple inheritance, both the parent class can have common methods. When inherited both the copies would be present in the child class. To avoid this we can use the virtual keyword in the parent class.

```
class istream: virtual public ios { . . .
class ostream: virtual public ios { . . .
class iostream: public istream, public ostream { . . .
class ifstream: public fstreambase, public istream { . . .
class ofstream: public fstreambase, public ostream { .   .
class fstream: public fstreambase, public iostream { . . .
```

2. A Class Hierarchy for Record Buffer Objects

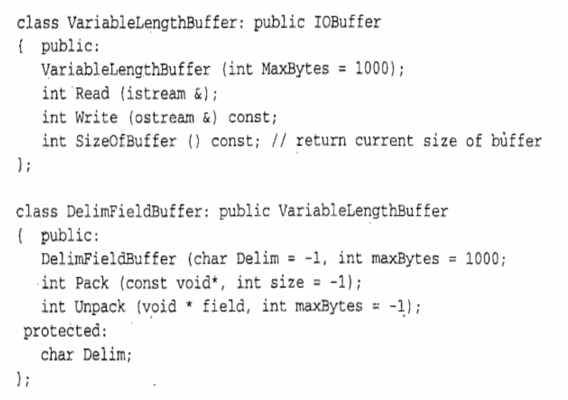

IOBuffer
char array for buffer value
VariableLengthBuffer
read and write operations
for variable length records
FixedLengthBuffer
read and write operations
for fixed length records
DelimitedFieldBuffer
pack and unpack operations
for delimited fields
LengthFieldBuffer
pack and unpack operations
for length-based fields
FixedFieldBuffer
pack and unpack operations for
fixed sized fields
Figure 4.14 Buffer class hierarchy

In buffer class hierarchy the IOBuffer class is at the top most level. The common methods required for fixed length buffer and variable length buffer is present. The variable length buffer class and fixed length buffer class would have their own implementation of various pack() and unpack() methods depending upon the field structure used.



```
class IOBuffer
{public:
   IOBuffer (int maxBytes = 1000); // a maximum of maxBytes
   virtual int Read (istream &) = 0; // read a buffer
   virtual int Write (ostream &) const = 0; // write a buffer
   virtual int Pack (const void * field, int size = -1) = 0;
   virtual int Unpack (void * field, int maxbytes = -1) = 0;
 protected:
   char * Buffer; // character array to hold field values
   int BufferSize; // sum of the sizes of packed fields
   int MaxBytes; // maximum number of characters in the buffer
};
```

Figure 4.15 Main members and methods of class IOBuffer.



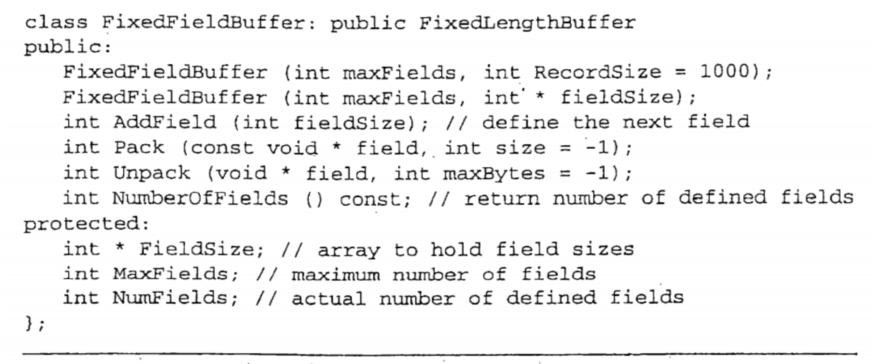
```
class VariableLengthBuffer: public IOBuffer
{  public:
   VariableLengthBuffer (int MaxBytes = 1000);
   int Read (istream &);
   int Write (ostream &) const;
   int SizeOfBuffer () const; // return current size of buffer
};

class DelimFieldBuffer: public VariableLengthBuffer
{  public:
   DelimFieldBuffer (char Delim = -1, int maxBytes = 1000;
   int Pack (const void*, int size = -1);
   int Unpack (void * field, int maxBytes = -1);
 protected:
   char Delim;
};
```

Figure 4.16 Classes VariableLengthBuffer and DelimFieldBuffer.

Managing Fixed-Length, Fixed-Field Buffers

Class FixedLengthBuffer .is the subclass of IOBuffer that supports read and write of fixed-length records. For this class, each record is of the same size. Instead of storing the record size explicitly in the file. Along with the record, the write method just writes the fixed-size record. The read method must know the size in order to read the record correctly. Each FixedLengthBuf fer object has a protected field that records the record size, Class FixedFieldBuffer, as shown in Fig. 4.17 and files fixfld.hand fixfld.cpp, supports a fixed set of fixed-length fields.

class FixedFieldBuffer keeps track of the field sizes. The protected member FieldSize holds the field sizes in an integer array. The AddField method Is used to specify field sizes.



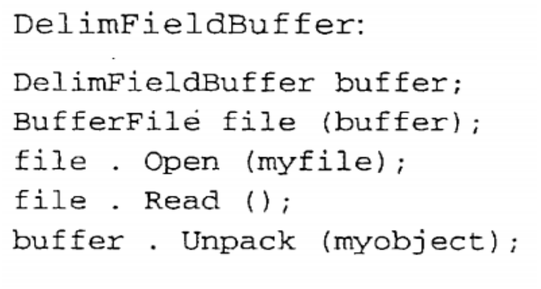
```
class FixedFieldBuffer: public FixedLengthBuffer
public:
   FixedFieldBuffer (int maxFields, int RecordSize = 1000);
   FixedFieldBuffer (int maxFields, int * fieldSize);
   int AddField (int fieldSize); // define the next field
   int Pack (const void * field, int size = -1);
   int Unpack (void * field, int maxBytes = -1);
   int NumberOfFields () const; // return number of defined fields
protected:
   int * FieldSize; // array to hold field sizes
   int MaxFields; // maximum number of fields
   int NumFields; // actual number of defined fields
};
```

Figure 4.17 Class FixedFieldBuffer.

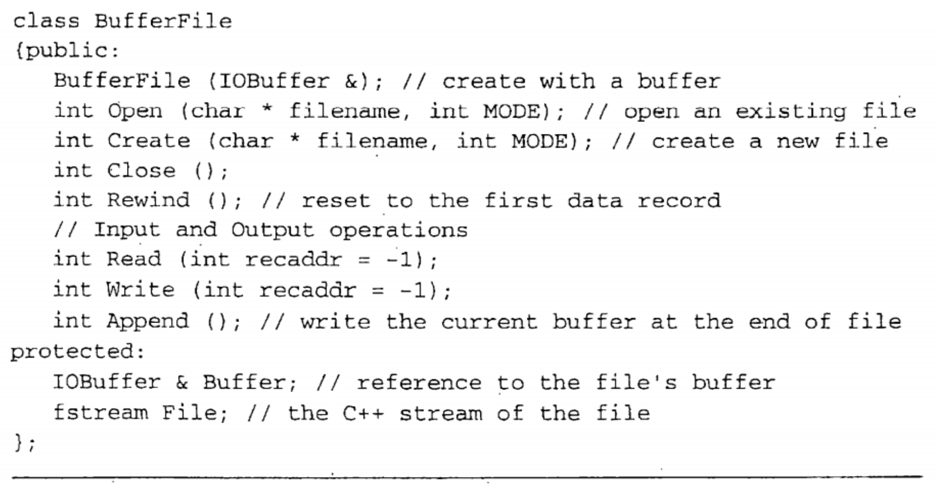
An Object-Oriented Class for Record Files

An object of class Buf ferFile is created from a specific buffer object and can be used to open and create files and to read and write records. Figure 4.18 has the main data methods and members of BufferFile. Once a BufferFile object has been created and attached to an operating system file, each read or write is performed using the same buffer. Hence, each record is guaranteed to be of the same basic type. The following code sample shows how a file can be created and used with a

DelimFieldBuffer



```
DelimFieldBuffer:

DelimFieldBuffer buffer;
BufferFile file (buffer);
file . Open (myfile);
file . Read ();
buffer . Unpack (myobject);
```



```
class BufferFile
{public:
   BufferFile (IOBuffer &); // create with a buffer
   int Open (char * filename, int MODE); // open an existing file
   int Create (char * filename, int MODE); // create a new file
   int Close ();
   int Rewind (); // reset to the first data record
   // Input and Output operations
   int Read (int recaddr = -1);
   int Write (int recaddr = -1);
   int Append (); // write the current buffer at the end of file
protected:
   IOBuffer & Buffer; // reference to the file's buffer
   fstream File; // the C++ stream of the file
};
```

Figure 4.18 Main data members and methods of class BufferFile.

A buffer is created, and the BufferFile object file is attached to it. Then Open and Read methods are called for file. After the Read, buffer contains the packed record, and buffer.Unpack puts the record into myobject.

Record Access

1. Record Keys
2. A Sequential Search
3. Unix tools for sequential processing

4. Direct access

In this Record Access there are four types,

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.
   Foe 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.
2. A 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
3. Unix tools for sequential processing
   The most common file structure used in Unix is ASCII file with newline character as record delimiter. All files created on Unix would have the same structure, such files are easy to process.
   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 words lines 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
```

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.

<u>More about Record Structures</u>

1. Choosing a record structure and record length.
   Once we decide to fix the length of our records so we can use the RRN to give us direct access to a record, we have to decide on a record length. Clearly, this decision is related to the size of the fields we want to store in the record. Sometimes the decision is easy. Suppose we are building a file of sales transactions that contain the following information about each transaction:
   •A six-digit account number of the purchaser,
   •Six digits for the date field,
   •A five-character stock number for the item purchased,
   •A three-digit field for quantity, and

•A ten-position field for total cost.

These are all fixed-length fields; the sur of the field lengths is 30 bytes.

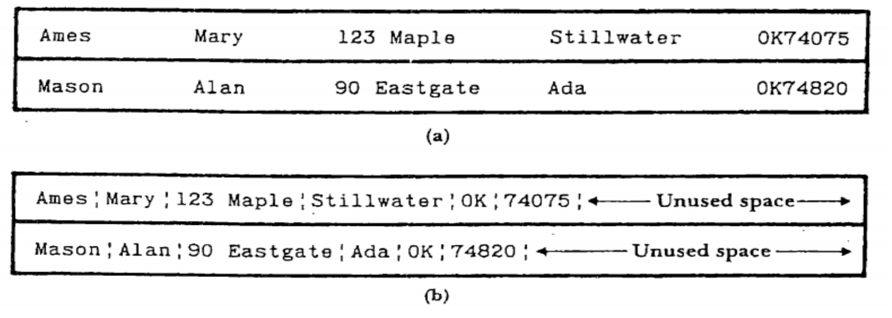

Ames Mary 123 Maple Stillwater OK74075
Mason Alan 90 Eastgate Ada OK74820
(a)
Ames¦Mary¦123 Maple¦Stillwater¦OK¦74075¦ Unused space
Mason¦Alan¦90 Eastgate¦Ada¦OK¦74820¦ Unused space
(b)

Figure 5.1 Two fundamental approaches to field structure within a fixed-length record. (a) Fixed-length records with fixed-length fields. (b) Fixed-length records with variable-length fields.

there are two general approaches we can take toward organizing fields within a fixed-length record. The first, illustrated in Fig. 5.1(a) and implemented in class FixedFieldBuffer, uses fixed-length fields inside the fixed-length record. This is the approach we took for the sales transaction file previously described. The second approach, illustrated in Fig. 5.1(b), uses the fixed-length record as a kind of standard-sized container for holding something that looks like a variable-length record. The first approach has the virtue of simplicity: it is very easy to "break out" the fixed-length fields from within a fixed-length record. The second approach lets us take advantage of an averaging-out effect that usually occurs: the longest names are not likely to appear in the same record as the longest address field.

2. 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.

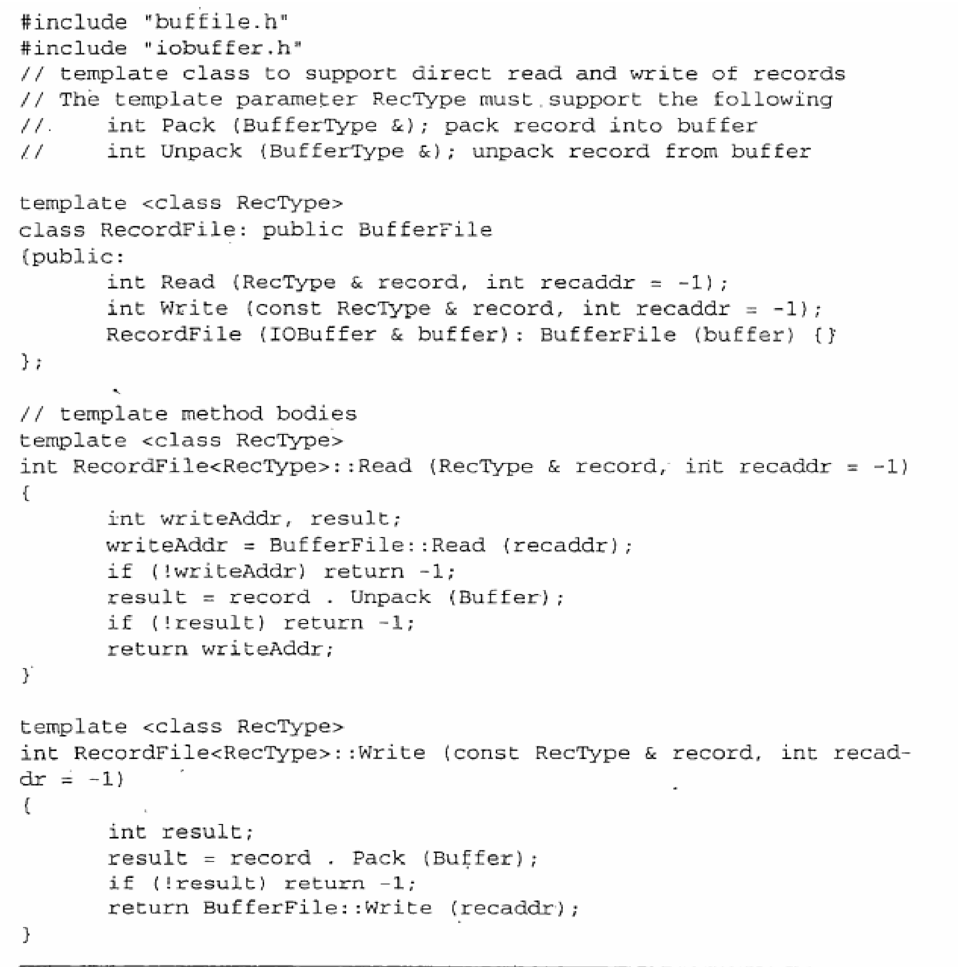
3. Adding header to c++ buffer classes

   The full definition of our buffer class hierarchy, has been extended to include methods that support header records. Class TOBuf fer includes the following methods:

```
virtual int ReadHeader ();
virtual int WriteHeader ();
```

Encapsulating Record Operations in a Single Class

A good object-oriented design for making objects persistent should provide operations to read and write objects directly. So far, the write operation requires two separate operations: pack into a buffer and write the buffer to a file.



```
#include "buffile.h"
#include "iobuffer.h"
// template class to support direct read and write of records
// The template parameter RecType must support the following
//      int Pack (BufferType &); pack record into buffer
//      int Unpack (BufferType &); unpack record from buffer

template <class RecType>
class RecordFile: public BufferFile
{public:
      int Read (RecType & record, int recaddr = -1);
      int Write (const RecType & record, int recaddr = -1);
      RecordFile (IOBuffer & buffer): BufferFile (buffer) {}
};

// template method bodies
template <class RecType>
int RecordFile<RecType>::Read (RecType & record, int recaddr = -1)
{
      int writeAddr, result;
      writeAddr = BufferFile::Read (recaddr);
      if (!writeAddr) return -1;
      result = record . Unpack (Buffer);
      if (!result) return -1;
      return writeAddr;
}

template <class RecType>
int RecordFile<RecType>::Write (const RecType & record, int recad-
dr = -1)
{
      int result;
      result = record . Pack (Buffer);
      if (!result) return -1;
      return BufferFile::Write (recaddr);
}
```

Figure 5.3 Template class RecordFile.

File Access & File Operation

In our discussions we have looked at

- Variable-length records,
- Fixed-length records,
- Sequential access, and
- Direct access.

The first two of these relate to aspects of file organization; the last two have to do with file access. The interaction between file organization and file access is a useful one; we need to look at it more closely before continuing. Most of what we have considered so far falls into the category of file organization:

- Can the file be divided into fields?
- Is there a higher level of organization to the file that combines the fields into records?
- Doall the records have the same number of bytes or fields?
- How do we distinguish one record from another?
- How do we organize the internal structure of a fixed-length record so we can distinguish between data and extra space?

We have seen that there are many possible answers to these questions and that the choice of a particular file organization depends on many things, including the file-handling facilities of the language you are using and the use you want to make of the file.

Prepared By,
Ranjitha J
Assistant Professor
Dept, of ISE
Atria Institute of Technology

Name of the Course: File Structure
NOTES

Organization of Files for Performance, Indexing

Data Compression

Data compression involves encoding the information in a file in such a way that it takes up less space. Various techniques are available for compressing data. Few of them are very general and some are designed for specific kinds of data, such as speech, pictures, text or instrument data. The main advantage of data compressions is:

- Use less storage, resulting in cost saving
- Can be transmitted faster, decreasing the access time
- Can be processed faster sequentially

Various Techniques of Data Compression

1. Using a different notation
   Fixed length fields are good candidates for compression of this type. For example, if a class person contains a field by name state which would occupy 2 ASCII bytes i.e., 16 bits compression can be done in the following manner. As there are only fifty states, we could represent all possible states in only 6 bits, i.e. we can encode all state names in a single 1 byte field, resulting in a space saving of 1 byte or 50%, per occurrence of the state field.

2. Supressing repeating sequences (Run Length Encoding)
   Sparse arrays are good candidates for run length encoding. It is an example of redundancy reduction. Run length encoding algorithm is a simple one whose associated costs rarely affect performance appreciably. The algorithm works in the following manner
   - We choose one special unused byte value to indicate that run length encoding starts
   - Copy the contents to file in a sequence, except where the same data value occurs more than once in succession
   - Where the same value occurs more than once in succession, we substitute the following 3 bytes in order
     - The special run length code indicator(ff)

- The value that is repeated
- The number of times that the value is repeated

Example: 22 23 24 24 24 24 24 25 26 28 28 28 27 29 29 29 29 29 24

Run length indicator is ff

22 23 ff 24 05 25 26 ff 28 03 27 ff 29 05 24

3. Assigning variable length codes
Variable length codes are based on the principle that some values occur more frequently than others, so the codes for those values should take the least amount of space. Variable length codes are another form of redundancy reduction. This scheme is the oldest and most common of the variable length codes, the morse code. The problem with morse code is that the look up table is static and the codes generated would not change depending upon the occurrence of the characters in the current input. Inorder to overcome this problem need dynamically generate the look up table.
Huffman code determines the probabilities of each values occurring in the data set and then builds a binary tree in which the search path for each value represents the code for that value. More frequently occurring value are given shorter search paths in the tree. This tree is turned into a table that can be used to encode and decode the data.

letter	a	b	c	d	e	f	g
probability	0.4	0.1	0.1	0.1	0.1	0.1	0.1
code	1	010	011	0000	0001	0010	0011
abef	101000010010						

4. Irreversible compression techniques
It is based on the assumption that some information can be sacrificed. This compression is less common in data files. This is applied only those files where at times when the information that is lost is of little or no value. An example of irreversible compression would be shrinking a raster image from say 400-by-400 pixels to 100-by-100 pixels. The new image contains 1 pixel for every 16 pixels in the origin image.
5. Compression in Unix
Berkeley and system unix provide various compression routines. System has routines called pack and unpack which uses Huffman code on a byte-by-byte basis. Pack routines used for compression achieves 25 to 40 percent reduction on text files. The pack routine during compression automatically appends .z at the end signaling that the file has compressed using the standard compression algorithm. Berkely unix uses routines called

compress and uncompress which uses an effective dynamic method called -ziv compress method appends .z at the end of the compressed file.

Reclaiming Space in Files

1. Record deletion & storage compaction
2. Deleting fixed-length records for reclaiming space dynamically
3. Deleting variable-length records
4. Storage fragmentation
5. Placement strategies

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

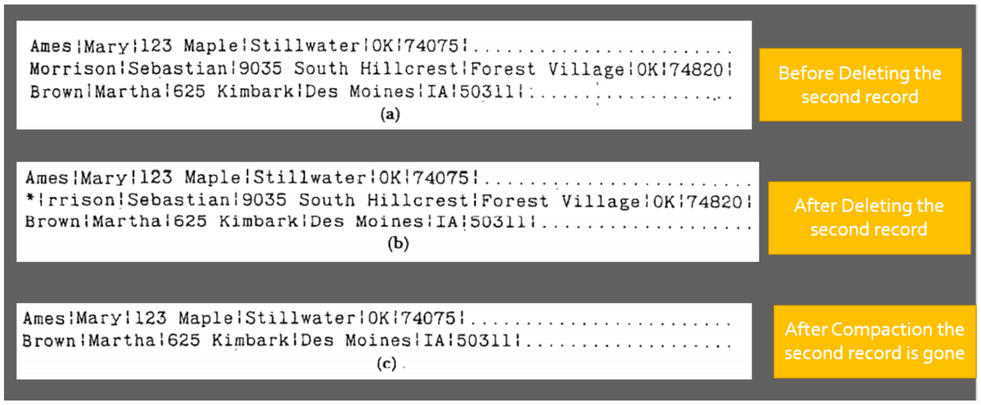
►We know how to create a file, how to add records to a file, modify the content of a record. These actions can be performed physically by using the various basic file operations we have seen (fopen, fclose, fseek, fread, fwrite)

►What happens if records need to be deleted?

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

In this there are 5 types

1. Record deletion & storage compaction
   Storage compaction makes files smaller by looking for places in a file where there is no data at all and recovering this space. Any record-deletion strategy must provide some way for us to recognize records as deleted. A simple and usually workable approach is to place a special mark in each deleted record.



```
Ames|Mary|123 Maple|Stillwater|OK|74075|........................
Morrison|Sebastian|9035 South Hillcrest|Forest Village|OK|74820|
Brown|Martha|625 Kimbark|Des Moines|IA|50311|...................
                              (a)
```

Before Deleting the second record

```
Ames|Mary|123 Maple|Stillwater|OK|74075|........................
*|rrison|Sebastian|9035 South Hillcrest|Forest Village|OK|74820|
Brown|Martha|625 Kimbark|Des Moines|IA|50311|...................
                              (b)
```

After Deleting the second record

```
Ames|Mary|123 Maple|Stillwater|OK|74075|........................
Brown|Martha|625 Kimbark|Des Moines|IA|50311|...................
                              (c)
```

   Figures 6.3(a) and 6.3(b) show a name and address file. Here the second record is marked as deleted. Once we are able to recognize a record as deleted, the next question is

how to reuse the space from the record. Approaches to this problem that rely on storage compaction do not reuse the space for a while. The records are simply marked as deleted and left in the file for a period of time. Programs rising the file must include logic that causes them to ignore records that are marked as deleted. One benefit to this approach is that it is usually possible to allow the user to undelete a record with very little effort. This is particularly easy if you keep the deleted mark in a special field rather than destroy some of the original data, as in our example.
The reclamation of space from the deleted records happens all at once. After deleted records have accumulated for some time, a special program is used to reconstruct the file with all the deleted records squeezed out as shown in Fig. 6.3(c). If there is enough space, the simplest way to do this compaction is through a file copy prog ram that ships over the deleted records. It is also possible, though more complicated and time-consuming, to do the compaction in place. Either of these approaches can be used with both fixed- and variable-length records. The decision about how often to run the storage compaction program can be based on either the number of deleted records.

2. Deleting fixed-length records for reclaiming space dynamically
   Storage compaction is the simplest and most widely used of the storage reclamation methods we discuss. There are some applications, however, that are too volatile and interactive for storage compaction to be useful. In these situations, need to reuse the space from deleted records as soon as possible. So discussion of such dynamic storage reclamation with a second look at fixed-length record deletion, since fixed-length records make the reclamation problem much simpler.
   In general, to provide a mechanism for record deletion with subsequent reutilization of the freed space, need to be able to ensure 2 things:
   - That deleted records are marked in some special way, and
   - That we can find the space that deleted records once occupied so we can reuse that space when we add records.

   Space reiitilization can take the form of looking through the file, record by record, until deleted record is found.

   - Linked List
     The use of a linked list for stringing together all of the available records can meet both of these needs. A linked list is a data structure in which each element or node contains some kind of reference to its successor in the list. In Fig. 6.4 we use a -1 in the point- er field to mark the end of the list. When a list is made up of deleted

records that have become available space within the file, the list is usually called an avail list.

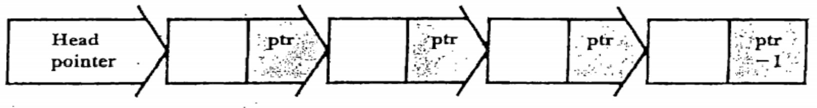
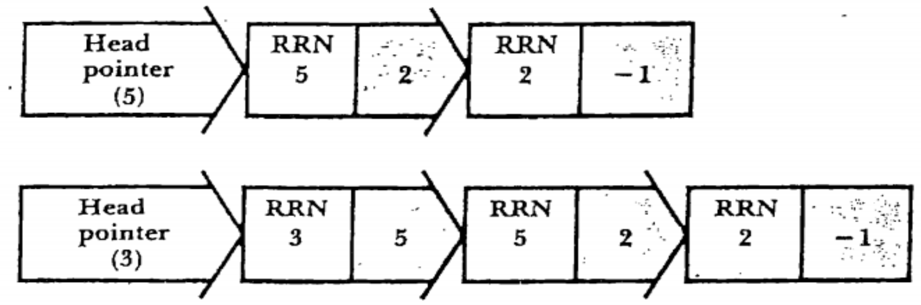


Figure 6.4 A linked list.

- Stacks

  The simplest way to handle a list is as a stack. A stack is a list in which all insertions and removals of nodes take place at one end of the list. So, if we have an avail list managed as a stack that contain5 relative record numbers (RRN) 5 and 2, and then add RRN 3, it looks like this before and after the addition of the new node:



  When a new node is added to the top or front of a stack, we 5ay that it is pushed onto the stack. If the next thing that happens is a request for some available space, the request is filled by taking RRN 3 from the avail list. This is called popping the stack. The list returns to a state in which it contains only records 5 and 2.

- Linking and stacking deleted records

  Stack implementation using linked list. The two criteria for rapid access to reusable space from deleted records.

  1. A way to know immediately if there are empty slots in the file, and
  2. A way to jump directly to one of those slots if it costs.

  Record 5 is the first record on the avail list (top of the stack) as it is the record that is most recently deleted. Following the linked list, we see that record 5 points to record 3. Since the link field for record 3 contains -1, which is our end-of-list marker, we know that record 3 is the last slot avail- able for reuse.

  Figure 6.5(b) shows the same file after record 1 is also deleted. Note that the contents of all the other records on the avail list remain unchanged. Treating the list as a stack results in a minimal amount of list reorganization when we push and pop records to and from the list. If we now add a new name to the file, it is placed in record 1, since RRN 1 is the first available record. The avail list would return to the configuration shown in Fig. 6.5(a). Since there are still two record slots on the avail list, we could add two more names to the file without increasing the size of the file.

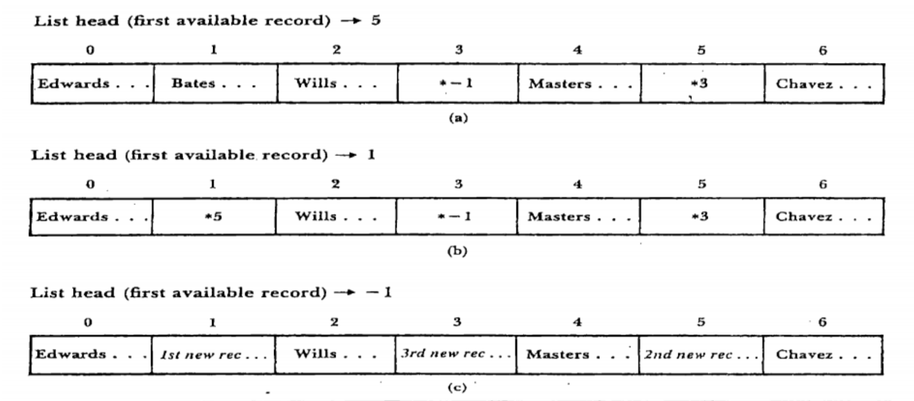
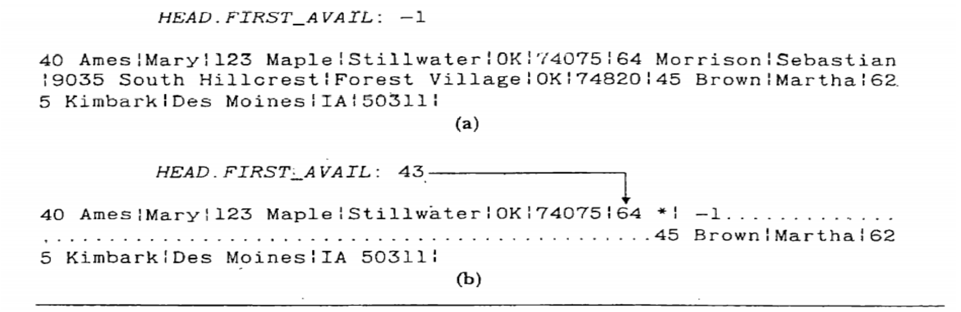


Figure 6.5 Sample file showing linked lists of deleted records. (a) After deletion of records 3 and 5, in that order. (b) After deletion of records 3, 5, and 1, in that order. (c) After insertion of three new records.

- Implementing fixed-length record deletion
  Implementing mechanisms that place deleted records on a linked avail list and that treat the avail list as a stack is relatively straightforward. We need a suitable place to keep the RRN of the first available record on the avail list. Since this is information that is specific to the data file, it can be carried in a header record at the start of the file.
  When we delete a record, we must be able to mark the record as delet- ed and then place it on the avail list. A simple way to do this is to place an
  * (or some other special mark) at the beginning of the record as a deletion mark, followed by the RRN of the next record on the avail list.
  Once we have a list of available records within a file, we can reuse the space previously occupied by deleted records. For this we would write a single function that returns either
    - the RRN of a reusable record slot or
    - the RRN of the next record to be appended if no reusable slots are available.

3. Deleting variable-length records
   We have seen that to support record reuse through an avail list, we need
   - A way to link the deleted records together into a list (that is, a place to put a link field);
   - An algorithm for adding newly deleted records to the avail list; and
   - An algorithm for finding and removing records from the avail list when we are ready to use them.
   - An Avail List of Variable-Length Records
     Begin with a variable-length record file containing the three records for Ames, Morrison, and Brown introduced earlier. Figure 6.6(a) shows that the file looks like (miv us the header) before any deletions, and Fig. 6.6(b) shows what it looks like after the deletion of the second record. The periods in the deleted record signing discarded characters.



```
HEAD.FIRST_AVAIL: -1

40 Ames|Mary|123 Maple|Stillwater|OK|74075|64 Morrison|Sebastian
|9035 South Hillcrest|Forest Village|OK|74820|45 Brown|Martha|62
5 Kimbark|Des Moines|IA|50311|
                              (a)

HEAD.FIRST_AVAIL: 43

40 Ames|Mary|123 Maple|Stillwater|OK|74075|64 *| -1.............
.............................................45 Brown|Martha|62
5 Kimbark|Des Moines|IA 50311|
                              (b)
```

Figure 6.6 A sample file for illustrating variable-length record deletion. (a) Original sample file stored in variable-length format with byte count (header record not included). (b) Sample file after deletion of the second record (periods show discarded characters).

   - Adding and Removing Records
     It is possible, even likely, that we need to search through the avail list for a record slot that is the right size. We can't just pop the stack and expect the first available record to be big enough. Finding a proper slot on the avail 1ist now means traversing the list until a record slot that is big enough to hold the new record is found.
     For example, suppose the avail list contains the deleted record slots shown in Fig. 6.7(a), and a record that requires 55 bytes is to be added. 3ince the avail list i5 not empty, we traverse the records whose sizes are 47 (too small), 38 (too small), and 72 (big enough). Having found a slot big enough to hold our record, we remove it from the avail list by creating new link that jumps over the record as shown in Fig. 6.7(b). If we had reached the end of the avail list before finding a record that was large enough, we would have appended the new record at the end of the file.

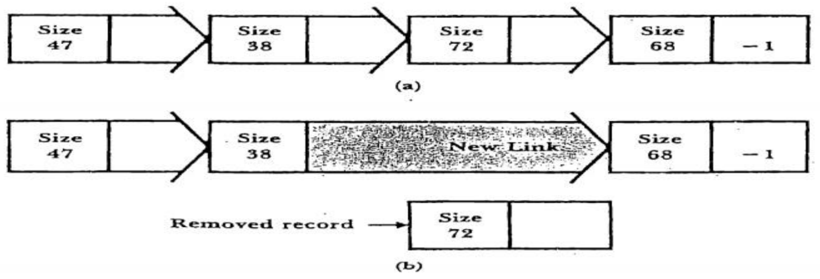
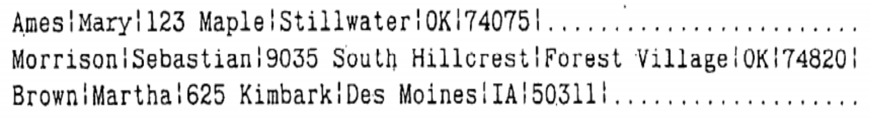


Figure 6.7 Removal of a record from an avail list with variable-length records. (a) Before removal. (b) After removal.

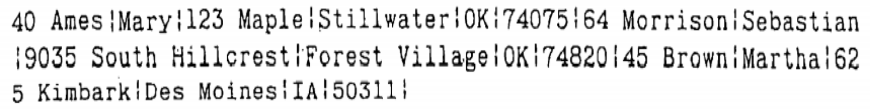
4. Storage fragmentation

   the fixed-length record version of our three-record file (Fig. 6.8). The dots at the ends of the records represent characters we use as padding between the last field and the end of the records. The padding is wasted space; it is part of the cost of using fixed-length records. Wasted space within a record is called internal fragmentation.



```
Ames|Mary|123 Maple|Stillwater|OK|74075|........................
Morrison|Sebastian|9035 South Hillcrest|Forest Village|OK|74820|
Brown|Martha|625 Kimbark|Des Moines|IA|50311|...................
```

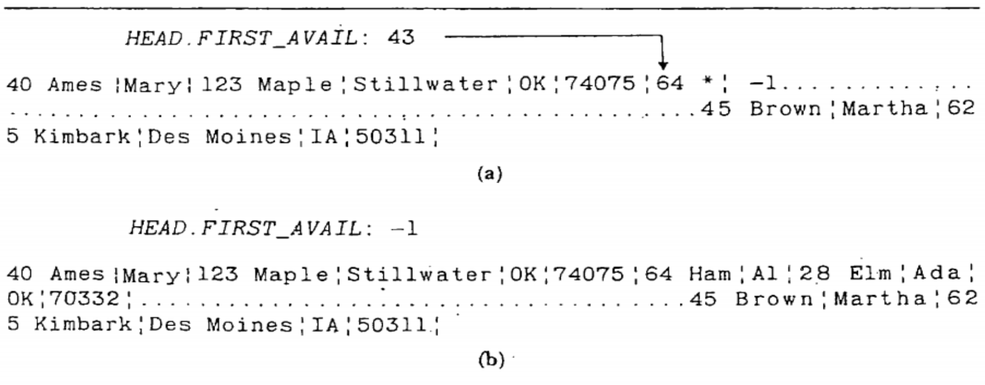
Figure 6.8 Storage requirements of sample file using 64-byte fixed-length records.



```
40 Ames|Mary|123 Maple|Stillwater|OK|74075|64 Morrison|Sebastian
|9035 South Hillcrest|Forest Village|OK|74820|45 Brown|Martha|62
5 Kimbark|Des Moines|IA|50311|
```

Figure 6.9 Storage requirements of sample file using variable-length records with a count field.

Compare the fixed- length example with the one in Fig. 6.9, which uses the variable length record structure—a byte count followed by delimited data fields. The only space (other than the delimiter5) that is not used for holding data in each record is the count field. If we assume that this field uses 2 bytes, this amounts to only 6 bytes for the three-record file. The fixed-length record file wastes 24 bytes in the very first record.



```
        HEAD.FIRST_AVAIL: 43

40 Ames|Mary|123 Maple|Stillwater|OK|74075|64 *| -1.............
.........................................45 Brown|Martha|62
5 Kimbark|Des Moines|IA|50311|
                             (a)

        HEAD.FIRST_AVAIL: -1

40 Ames|Mary|123 Maple|Stillwater|OK|74075|64 Ham|Al|28 Elm|Ada|
OK|70332|.....................................45 Brown|Martha|62
5 Kimbark|Des Moines|IA|50311|
                             (b)
```

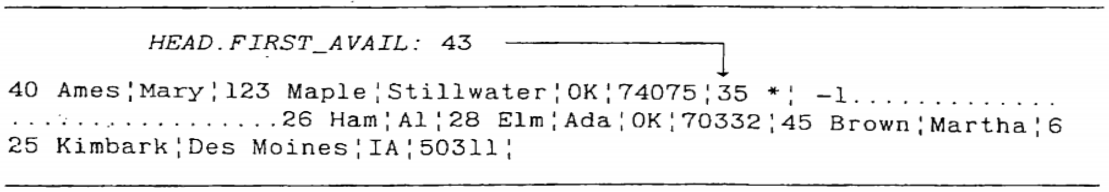
Figure 6.10 Illustration of fragmentation with variable-length records. (a) After deletion of the second record (unused characters in the deleted record are replaced by periods). (b) After the subsequent addition of the record for Al Ham.

Figure 6.10 shows how the problem could occur with our sample file when the second record in the file is deleted and the following record is added:

Ham|All28 ElmlAda|OK|70332I

It appears that escaping internal fragmentation is not so easy. The slot vacated by the deleted record is 37 byres larger than is needed for the new record. Since we treat the extra 37 bytes as part of the new record, they are not on the avail list and are therefore unusable, But instead of keeping the 64-byte record slot intact, suppose we break it into two parts: one part to hold the new Ham record, and the other to be placed back on the avail list. Since we would take only as much space as necessary for the Ham record, there would be no internal fragmentation.

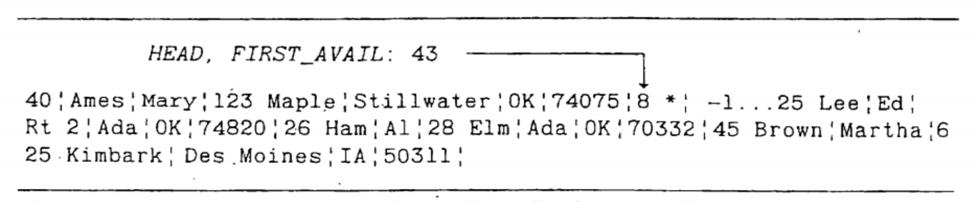
Figure 6.11 shows what our file looks like if we use this approach to insert the record for Al Ham. We steal the space for the Ham record from the end of the 64-byte slot and leave the first 35 bytes of the slot on the avail list. (The available space is 35 rather than 37 bytes because we need 2 bytes to form a new size field for the Ham record.) The 35 bytes still on the avail list can be used to hold another record.



```
        HEAD.FIRST_AVAIL: 43

40 Ames|Mary|123 Maple|Stillwater|OK|74075|35 *| -1.............
.................26 Ham|Al|28 Elm|Ada|OK|70332|45 Brown|Martha|6
25 Kimbark|Des Moines|IA|50311|
```

Figure 6.11 Combating internal fragmentation by putting the unused part of the deleted slot back on the avail list.

Figure 6.12 shows the effect of inserting the following 25-byte record:
`Lee|Ed|Rt 2|Ada|OK|74820|`



```
          HEAD, FIRST_AVAIL: 43
40¦Ames¦Mary¦123 Maple¦Stillwater¦OK¦74075¦8 *¦ -1...25 Lee¦Ed¦
Rt 2¦Ada¦OK¦74820¦26 Ham¦Al¦28 Elm¦Ada¦OK¦70332¦45 Brown¦Martha¦6
25 Kimbark¦ Des Moines¦IA¦50311¦
```

Figure 6.12 Addition of the second record into the slot originally occupied by a single deleted record.

As we would expect, the new record is carved out of the 35-byte record that is on the avail list. The data portion of the new record requires 25 bytes, and we need 2 more bytes for another size field. This leaves 8 bytes in the record still on the avail list.
These 8 bytes are not usable, even though they are not trapped inside any other record. This is an example of external fragmentation.

5. Placement strategies
There are several strategies for selecting a record from AVAIL LIST when adding a new record:
First-Fit Strategy –
AVAIL LIST is not sorted by size. – First record large enough to hold new record is chosen.
► Example: – AVAIL LIST: size=10,size=50,size=22,size=60 – record to be added: size=20 – Which record from AVAIL LIST is used for the new record?
Best-Fit Strategy –
AVAIL LIST is sorted by size. – Smallest record large enough to hold new record is chosen.
► Example: – AVAIL LIST: size=10,size=22,size=50,size=60 – record to be added: size=20 – Which record from AVAIL LIST is used for the new record?
Worst-Fit Strategy –
AVAIL LIST is sorted by decreasing order of size. – Largest record is used for holding new record; unused space is placed again in AVAIL LIST.
► Example: – AVAIL LIST: size=60,size=50,size=22,size=10 – record to be added: size=20 – Which record from AVAIL LIST is used for the new record?

Internal Sorting and Binary Searching

1. Finding Things in Simple Field and Record Files
   Sequential search is one of the simplest forms of file searching. The file is searched one record at a time, until a record is found with a particular key Sequential search is slow:
   • If there are n records in the file, you may have to look at all of them before you find the one you want
   • If the key you are looking for is in the file, on average you will need to look through n/2 records before finding it Sequential search is said to be O(n), because the time it takes is proportional to n.
   Although sequential search is slow, it is not appalling Sequential search always looks at the adjacent record in the file next
      - Therefore, it makes good use of the fact that every read of a file does not result in a disk access
      - A big chunk of the file is read into a buffer in main memory
      - So most reads of the file will not actually result in disk accesses
2. Binary Search
   Let's take an example:
   • Suppose we're looking for a student with id number 76634 in a file of 10,000 fixed length records
   • Assume further than the file has been sorted into ascending order of student numbers
   • We start by comparing 76634 with the student number of the record in the middle of the file, that is record 5,000
   • If record 5000's student number if greater than 76634, we know that 76634 will be found in the first half of the file
   • If it is less than 76634, then we know 76634 can be found in the
   second half of the file
   • Assuming it we know that 76634 is in the first half, we now compare this with the student number of the record at position 2,500 to find out which quarter of the file 76634 is in
   • The process is repeated until either 76634 is found or we have narrowed the number of potential records to zero
   • This is called binary search

```
low = 0
high = number of records -1
while ( low <= high )
          guess = (low + high) / 2
          key_found = read_key_number(guess)
          if ( key_sought > key_found )
                    low = guess + 1
          else if (key_sought < key_found ) high = guess - 1
                    else
                              we've found it
endwhile
```

3. Binary Search Versus Sequential Search
   The difference becomes dramatic if there are a lot of records in the file
   • When we double the number of records, we double the number of comparisons for sequential search
   • When we double the number of records, we add one to the number of comparisons for binary search
   • But, even though it might take sequential search 5,000 comparisons, and binary search only 14 comparisons, does not mean that binary search is 5,000 / 14 = 357 times faster than sequential search

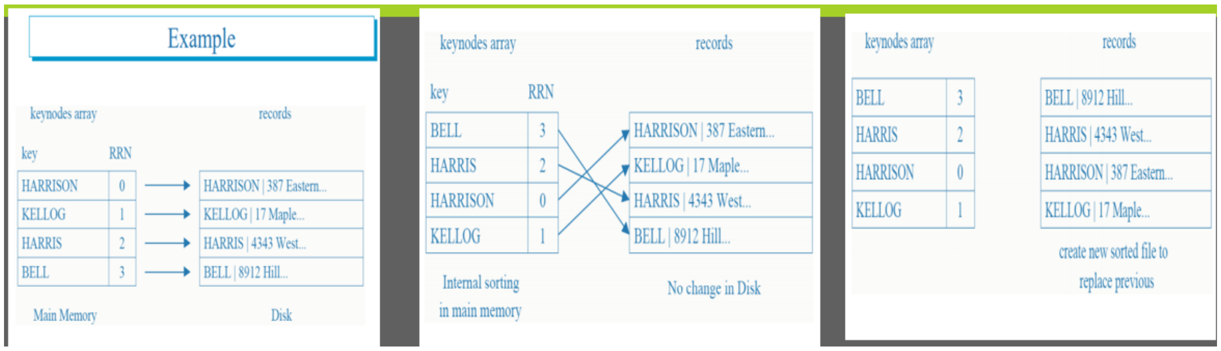
4. Sorting a Disk File in Memory
   Consider the operation of any internal sorting algorithm with which you are familiar. The algorithm requires multiple passes over the list that is to be sorted, comparing and reorganizing the elements. Some of the items in the list are moved a long distance from their original positions in the list. If such an algorithm were applied directly to data stored on a disk, it is clear that there would be a lot of jumping around, seeking, and rereading of data. This would be a very slow operation—unthinkably slow.
5. The Limitations of Binary Searching and Internal Sorting
   If we have a sorted file we can find a record quickly with binary search.
   But binary search is still not ideal:
   Problem 1: binary search requires several disk accesses:
   • Although binary search is a tremendous improvement over sequential search, those disk accesses are still expensive
   • Ideally we would be able to find the data in just one or two accesses
   • Ideally, we would be able to work out at which record number the data is stored from the key.
   Problem 2: Keeping a file sorted can be very expensive

• When we add records to the file, we need to resort the file. This can be very, very expensive. If we add records as often as we search for records, we will spend most of our time sorting the file. Even if we can find the position to put the new record into cheaply, we need to move records to make space for the new record
Better solutions will have at least one of the following features:
• They will not involve re–ordering the file when a new record is added
• They will use data structures that allow rapid, efficient re–ordering of the file
Problem 3: An Internal sort works only on small files
An internal sort works only if we can read the entire contents of a file into the computer's electronic memory. If the file is so large that we cannot do that, we need a different kind of sort.

Keysorting

Keysort, sometimes referred to as tag Sort, is based on the idea that when we sort a file in memory the only things that we really need to sort are the record keys; therefore, we do not need to read the whole file into memory during the sorting process. Instead, we read the keys from the file into memory, sort them, and then rearrange the records in the file according to the new ordering of the keys.

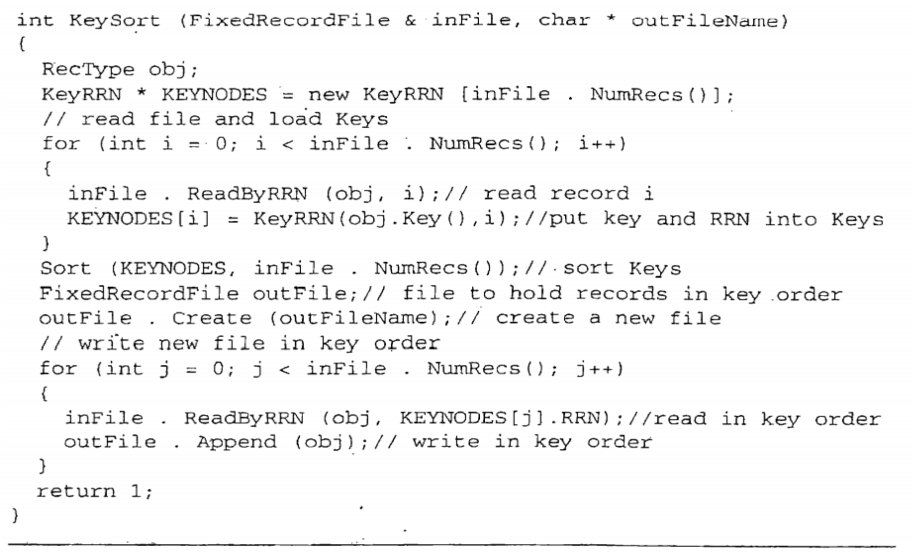
1. Description of the Method
   Suppose a file needs to be sorted, but it is too big to fit into main memory. To sort the file, we only need the keys. Suppose that all the keys fit into main memory
   ►Idea
   – Bring the keys to main memory plus corresponding RRN
   – Do internal sorting of keys
   – Rewrite the file in sorted order



How much effort we must do?

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

Keysort Algorithm



```
int KeySort (FixedRecordFile & inFile, char * outFileName)
{
  RecType obj;
  KeyRRN * KEYNODES = new KeyRRN [inFile . NumRecs()];
  // read file and load Keys
  for (int i = 0; i < inFile . NumRecs(); i++)
  {
    inFile . ReadByRRN (obj, i);// read record i
    KEYNODES[i] = KeyRRN(obj.Key(),i);//put key and RRN into Keys
  }
  Sort (KEYNODES, inFile . NumRecs());// sort Keys
  FixedRecordFile outFile;// file to hold records in key order
  outFile . Create (outFileName);// create a new file
  // write new file in key order
  for (int j = 0; j < inFile . NumRecs(); j++)
  {
    inFile . ReadByRRN (obj, KEYNODES[j].RRN);//read in key order
    outFile . Append (obj);// write in key order
  }
  return 1;
}
```

Figure 6.18 Algorithm for keysort

2. Limitations of the keysort method
   Limitations - Can be more expensive than first appears
   - Applies only to files on disk
   - Reads each record twice
   - Second read involves reading records in sorted order which may require a random seek on disk
   - Writes are sequential but interleaved with random seeks
   - Size of file that can be sorted is limited by the number of key/pointer pairs that can be contained in RAM
3. Another solution: why bother to write the file back?
   This is an instance of one of our favorite categories of solutions to computer science problems: if some part of a process begins to look like a bottleneck, consider skipping it altogether. Ask if you can do without it. Instead of creating a new, sorted copy of the file to use for searching, we have created a second kind of file, an index file, that is to be used in conjunction with the original file. If we are looking for a particular record, we do our binary search on the index file and then use the RRN stored in the index file record to find the corresponding record in the original file.

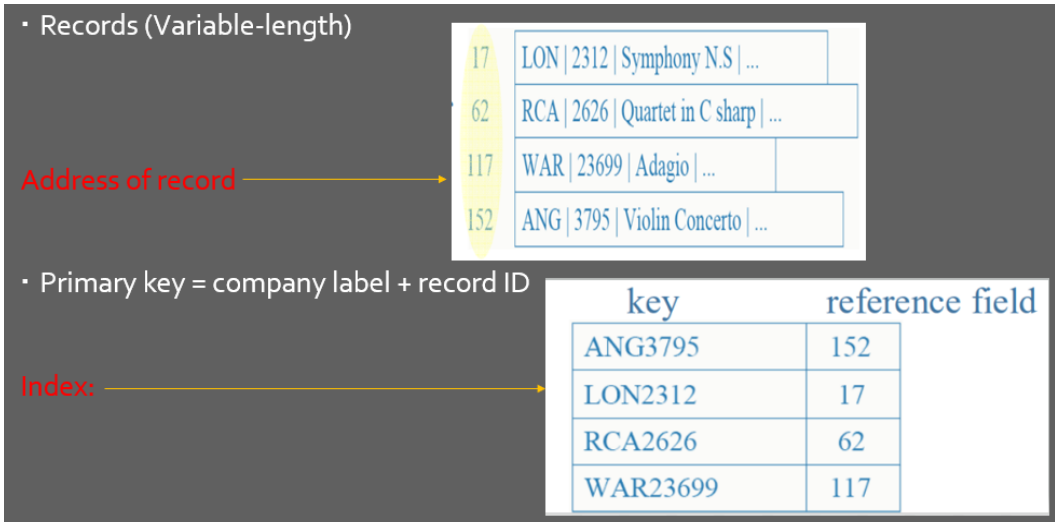
4. Pinned Records
   Remember that in order to support deletions we used AVAIL LIST AVAIL LIST, a list of available records. The AVAIL LIST AVAIL LIST contains info on the physical information of records. In such a file, a record is said to be pinned. If we use an index file index file for sorting, the AVAIL LIST AVAIL LIST and positions of records remain unchanged. This is a good news.

What is an Index?

An index is a table containing a list of keys and corresponding reference fields. All indexes are based on the concept of keys and references. Even though indexes are very simple they still provide a powerful tool for file processing. It is an alternative tool to scan a file sequentially to find a record. An index let us impose order on a file without rearranging the file. Indexing gives us keyed access to variable length record file.

A Simple Index for Entry-Sequenced File

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


- Records (Variable-length)

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

- Primary key = company label + record ID

key	reference field
ANG3795	152
LON2312	17
RCA2626	62
WAR23699	117

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

How to search for a recording with given LABEL ID?

– Binary search (in main memory) in the index: find LABEL ID, which leads us to the referenced field.

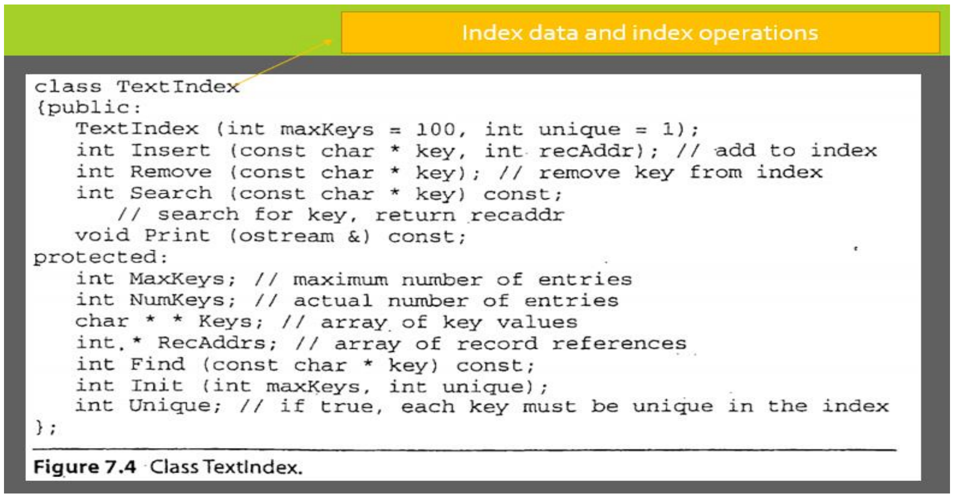
– Seek for record in position given by the reference field.

How to make a persistent index

– i.e. how to store the index into a file when it is not in main memory.

How to guarantee that the index is an accurate reflection of the contents of the file

– This is tricky when there are lots of additions, deletions and update.


Index data and index operations

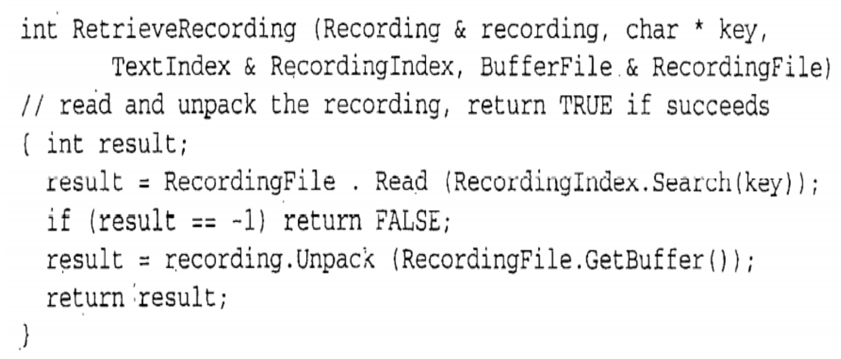
```
class TextIndex
{public:
   TextIndex (int maxKeys = 100, int unique = 1);
   int Insert (const char * key, int recAddr); // add to index
   int Remove (const char * key); // remove key from index
   int Search (const char * key) const;
      // search for key, return recaddr
   void Print (ostream &) const;
protected:
   int MaxKeys; // maximum number of entries
   int NumKeys; // actual number of entries
   char * * Keys; // array of key values
   int.* RecAddrs; // array of record references
   int Find (const char * key) const;
   int Init (int maxKeys, int unique);
   int Unique; // if true, each key must be unique in the index
};
```

Figure 7.4 Class TextIndex.

Class Text Index of Fig. 7.4 encapsulates the index data and index operations. The full implementation of class Text Index- An index is implemented with arrays to hold the keys and record references. Each object is declared with a maximum number of entries and can be used for unique kéys (no duplicates) and for nonunique keys (duplicates allowed). The methods Insert and Search do most of the work of indexing. The protected method Find locates the element key and returns its index. If the key is not in the index, Find returns -1. This method is used by Insert, Remove, and Search. In the case of class Text Index, the protected members Keys and Recaddr s are created dynamically by the constructor and should be deleted by the destructor to avoid an obvious memory leak:

TextIndex::-TextIndex ()(delete Keys; delete RecAddrs;}

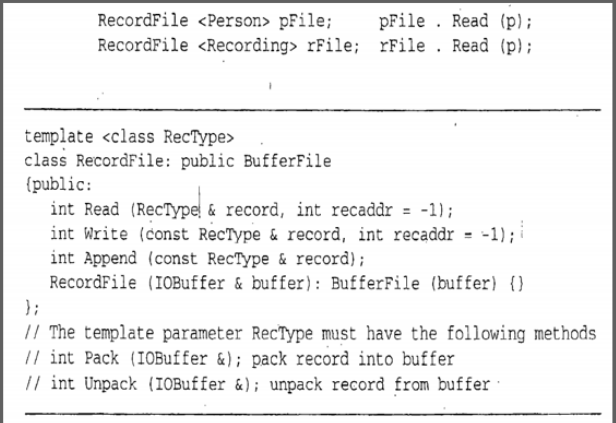
Using the index to provide access to the data file by Label ID is a simple matter. The code to use our classes to retrieve a single record by key from a recording file is shown in the function RetrieveRecording:



```
int RetrieveRecording (Recording & recording, char * key,
       TextIndex & RecordingIndex, BufferFile & RecordingFile)
// read and unpack the recording, return TRUE if succeeds
{ int result;
  result = RecordingFile . Read (RecordingIndex.Search(key));
  if (result == -1) return FALSE;
  result = recording.Unpack (RecordingFile.GetBuffer());
  return result;
}
```

with an open file and an index to the file in memory, RetrieveRecording puts together the index search, file read, and buffer unpack operations into a single function. Keeping the index in memory as the program runs also lets us find records by key more quickly with an indexed file than with a sorted one since the binary searching can be performed entirely in memory.

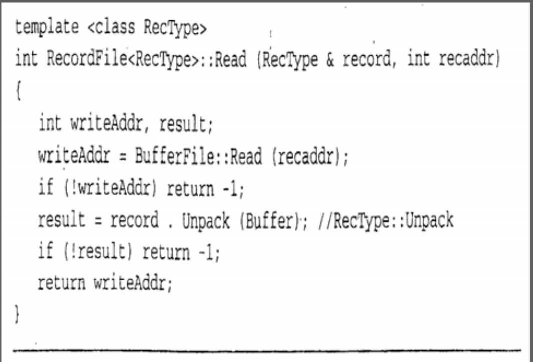
<u>Using Template Classes in C++ for Object I/O</u>



```
        RecordFile <Person> pFile;     pFile . Read (p);
        RecordFile <Recording> rFile;  rFile . Read (p);
```

```
template <class RecType>
class RecordFile: public BufferFile
{public:
   int Read (RecType & record, int recaddr = -1);
   int Write (const RecType & record, int recaddr = -1);
   int Append (const RecType & record);
   RecordFile (IOBuffer & buffer): BufferFile (buffer) {}
};
// The template parameter RecType must have the following methods
// int Pack (IOBuffer &); pack record into buffer
// int Unpack (IOBuffer &); unpack record from buffer
```

Figure 7.5 Template Class RecordFile.



```
template <class RecType>
int RecordFile<RecType>::Read (RecType & record, int recaddr)
{
   int writeAddr, result;
   writeAddr = BufferFile::Read (recaddr);
   if (!writeAddr) return -1;
   result = record . Unpack (Buffer); //RecType::Unpack
   if (!result) return -1;
   return writeAddr;
}
```

Figure 7.6 Implementation of RecordFile::Read.

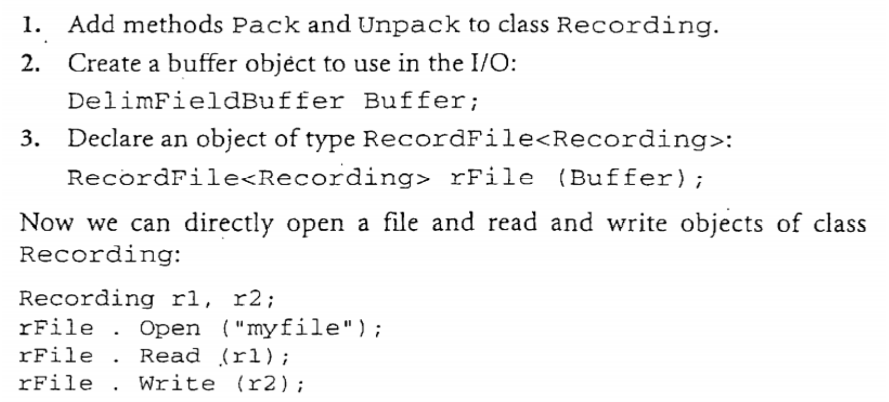
It is the C++ template feature that supports parameterized function and class definitions, and RecordFile is a template class. As shown in Fig. 7.5, class RecordFile includes the parameter RecType, which is used as the argument type for the read and write methods of the class. Class RecordFile is derived from Bufferfile, which provides most of the functionality. The constructor for RecordFile is given inline and simply calls the Bufferfile constructor.

The definitions of pFile and rFile just given are not consistent With use of a template class. The actual declarations and calls are:

RecordFile <Person> pFile; pFile . Read (p);

RecordFile <Recording> rFile; rFile . Read (p);

Class RecordFile accomplishes the goal of providing object- oriented 1/O for data. Adding I/O to an existing class (class Recording, for example) requires three steps:



1. Add methods `Pack` and `Unpack` to class `Recording`.
2. Create a buffer object to use in the I/O:
   `DelimFieldBuffer Buffer;`
3. Declare an object of type `RecordFile<Recording>`:
   `RecordFile<Recording> rFile (Buffer);`

Now we can directly open a file and read and write objects of class `Recording`:

```
Recording r1, r2;
rFile . Open ("myfile");
rFile . Read (r1);
rFile . Write (r2);
```

<u>Object Oriented Support for Indexed, Entry-Sequenced Files of Data Objects</u>

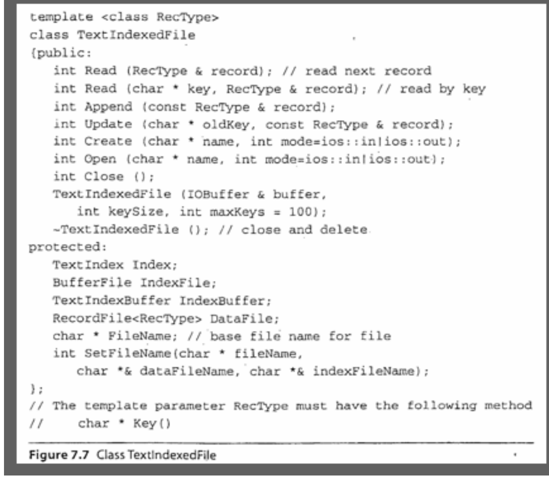
1. Operations Required to Maintain an Indexed File
   The support and malntenance of an entry-sequenced file coupled with a simple index requires the operations to handle a number of different tasks. Besides the RetrieveRecording function described previously, other operations used to find things by means of the index include the following:
   - Creating the files – two files must be created. A data file to hold the data objects and index file to hold the primary key index. Both index file and the data file are created as empty files with header records.
   - Loading the index into memory – only the index file would be loaded into the memory using the index file corresponding record in the file is read through the reference in the index file.
   - Rewriting the index file from memory – the index file that was loaded on to memory needs to be rewritten back to the index file on the close operation. The action of rewriting the index is very important to guard against power failure, the operator turning the machine off at the wrong time and other such disasters. One of the dangers associated with copy of an index on disk might be out of date and we should prevent our programs from using such out dated index copy using some safeguard measures. The following two mechanism can be implemented in the program to avoid such problem.

- – there should be mechanism that permits the program to know when the index is out of date. This can be done by setting a status flag as soon as the copy of the index in memory is changed.
- – if a program detects that an index is out of date, the program must have access to a procedure that reconstructs the index from the data file and this must be done automatically by the program.

- Record Addition – adding a new record to the data file requires that we also add an entry to the index file. Index is always in a sorted order, so adding a new entry means that rearrangement is required to place the new entry at the correct location. All the required rearrangement can be done quickly as the entire is in the memory and no file access is required.
- Record Deletion – the main advantage of indexed file organisation is that the record in file need not be moved around to maintain an ordering on the file inorder to delete a record we just remove the corresponding entry from index and the space created can be filled up shifting the other entries to close up the space. This operation may not be expensive as the entire index is the memory and no file operation is done. At end a new file can be created for only those entire that are present in the index file.
- Record Updating –
    - the update changes the value of the key field,
    - the update does not affect the key field.

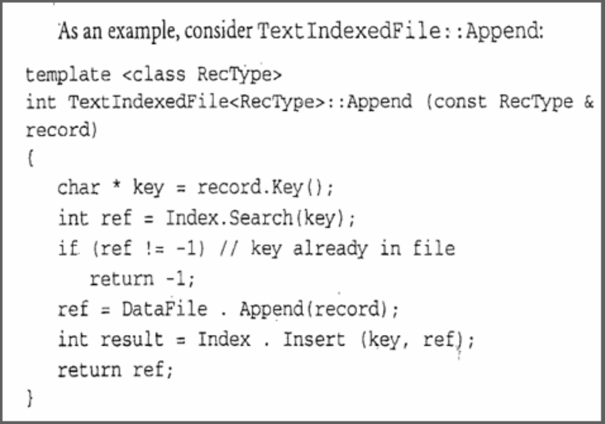
  In the first type of update an recording in both index as well as the data file is required. In the second type reordering is required only if the size of the new record is more than the available space.

2. Class TextIndexedFile



```
template <class RecType>
class TextIndexedFile
{public:
   int Read (RecType & record); // read next record
   int Read (char * key, RecType & record); // read by key
   int Append (const RecType & record);
   int Update (char * oldKey, const RecType & record);
   int Create (char * name, int mode=ios::in|ios::out);
   int Open (char * name, int mode=ios::in|ios::out);
   int Close ();
   TextIndexedFile (IOBuffer & buffer,
      int keySize, int maxKeys = 100);
   ~TextIndexedFile (); // close and delete
protected:
   TextIndex Index;
   BufferFile IndexFile;
   TextIndexBuffer IndexBuffer;
   RecordFile<RecType> DataFile;
   char * FileName; // base file name for file
   int SetFileName(char * fileName,
      char *& dataFileName, char *& indexFileName);
};
// The template parameter RecType must have the following method
//    char * Key()
```

Figure 7.7 Class TextIndexedFile



As an example, consider TextIndexedFile::Append:

```
template <class RecType>
int TextIndexedFile<RecType>::Append (const RecType &
record)
{
   char * key = record.Key();
   int ref = Index.Search(key);
   if (ref != -1) // key already in file
      return -1;
   ref = DataFile . Append(record);
   int result = Index . Insert (key, ref);
   return ref;
}
```

Class TextIndexedFile is defined in Fig. 7.7. It supports files of data objects with primary keys that are strings. As expected, there are methods: Create, Open, Close, Read (sequential and indexed), Append, and Update. In order to ensure the correlation between the index and the data file, the members that represent the index in memory (Index), the index file (IndexFile), and the data file (DataFile) are protected members. The only access to these member is for the user is through the methods.
The Key method is used to extract the hey value from the record. A search of the index is used to determine if the key is already in the file. If not, the record is appended to the data file, and the resulting address is inserted into the index along with the key.

3. Enhancement to Class TextIndexedFile
   - **Other types of keys**
     TextIndexedFile
     - support a variety of data object classes, but restrict the keytype to string (char *)

     => change a class to a template class
     (1) add a template parameter
     - replace char * by keyType
     (2) replace a class name with the parameter name
     - produce a template class SimpleIndex with a parameter for the key type
   - **Data object class hierarchies**
     TextIndexedFile
     - every object stored in a RecordFile must be of the same type
     - Can the I/O classes support objects that are of a variety of types but all from the same type hierarchy ?
       - if the type hierarchy supports virtual pack methods, the Append and Update will correctly add records to indexed files
       - if BaseClass supports Pack, Unpack, and Key (virtual functions), the class TextIndexedFile<BaseClass> will correctly output objects derived from BaseClass
     - What about Read ?
     - in a virtual function call, the type of the calling object determines which method to call

BaseClass * obj = new subclass1;

obj->Pack(Buffer); //virtual function calls

obj->Unpack(Buffer);

- the type of the object referenced by obj (*obj) determines which Pack and Unpack are called
- In the case of Pack
    - information from *obj, of type subclass1, is transferred to Buffer
- In the case of Unpack
    - information from Buffer is transferred to *obj
    - if Buffer has been filled from an object of class subclass2 or BaseClass, the unpacking cannot be done correctly
    - the source of information (contents of buffer) determines the type of the object in the Unpack, not the memory object

=> Read must be able to determine reliably the type of the target object

- **Multirecord index files**
  TextIndexedFile
    - the entire index fits in a single record

      TextIndex Index;
      BufferFile IndexFile;
    - the maximum # of records in the file is fixed when the file is created

  => Modify class TextIndexedFile to allow the index to be an array of TextIndex objects
  => Add protected methods Insert, Delete, and Search to manipulate the arrays of index object
- **Optimization of operation**
  Use binary search in the Find method, which is used by Search, Insert, and Remove.
    - Avoid writing the index record back to the index file when it has not been changed
    - add a flag (where ?) to the index object to signal when it has been changed
        - Set to false when the record is initially loaded into memory and set to true whenever the index record is modified by the Insert and Remove methods
        - Close method can check this flag and write the record only when necessary
    - useful for manipulating multirecord index files

Indexes That are Too Large to Hold in Memory

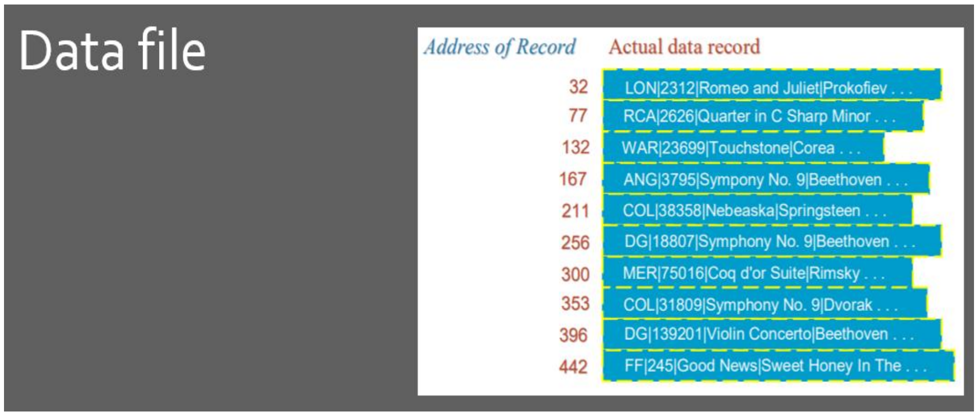
- Disadvantages of simple indexes, which is too large to hold in memory
  - binary searching of the index requires several seeks
  - binary searching of an index on secondary storage is not fast
  - index rearrangement due to record addition or deletion requires shifting or sorting records on secondary storage

  these problems are no worse than those associated with any file that is sorted by key

- Advantages over the use of a data file sorted by key, even if the index cannot be held in memory
  - can use a binary search to access to a record in a variable-length record file
  - sorting and maintaining the index can be less expensive than the data file
  - can rearrange the keys without moving the data records, when pinned records are involved in the data files
  - can use multiple indexes to provide multiple views of a data file

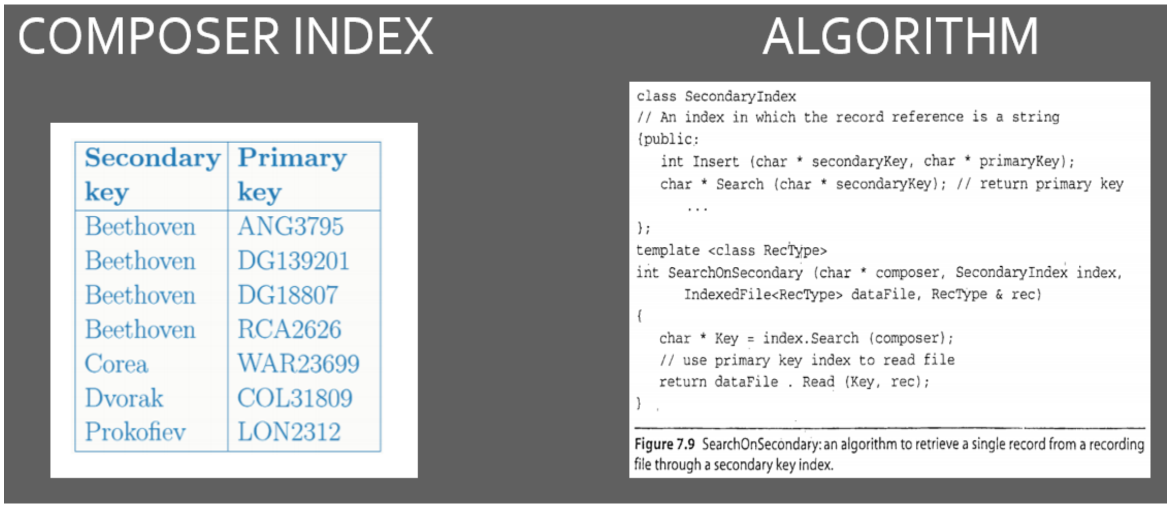
Indexing to Provide Access by Multiple Keys

Suppose that you are looking at a collection of recordings with the following information about each of them: Identification Number, Title..



Data file

Address of Record	Actual data record
32	LON|2312|Romeo and Juliet|Prokofiev . . .
77	RCA|2626|Quarter in C Sharp Minor . . .
132	WAR|23699|Touchstone|Corea . . .
167	ANG|3795|Sympony No. 9|Beethoven . . .
211	COL|38358|Nebeaska|Springsteen . . .
256	DG|18807|Symphony No. 9|Beethoven . . .
300	MER|75016|Coq d'or Suite|Rimsky . . .
353	COL|31809|Symphony No. 9|Dvorak . . .
396	DG|139201|Violin Concerto|Beethoven . . .
442	FF|245|Good News|Sweet Honey In The . . .



COMPOSER INDEX

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

ALGORITHM

```
class SecondaryIndex
// An index in which the record reference is a string
{public:
   int Insert (char * secondaryKey, char * primaryKey);
   char * Search (char * secondaryKey); // return primary key
      ...
};
template <class RecType>
int SearchOnSecondary (char * composer, SecondaryIndex index,
      IndexedFile<RecType> dataFile, RecType & rec)
{
   char * Key = index.Search (composer);
   // use primary key index to read file
   return dataFile . Read (Key, rec);
}
```

Figure 7.9 SearchOnSecondary: an algorithm to retrieve a single record from a recording file through a secondary key index.

Secondary key fields - relate a secondary key to a primary key and after consulting the secondary key index, consult the primary key index. In the above Fig 7.9 it's a Class definition for secondary key index and a read function. Two binding methods

(1) Direct addressing - binding of a secondary key to a specific address the secondary key references(Beethoven) is directly related to a byte offset(256)

(2) Indirect addressing - binding of a secondary key to a specific primary key the secondary key references(Beethoven) is related to a primary key(DG18807)

Record Addition - when a secondary index is present, adding a record to the file means adding and entry to the secondary index. The cost of doing this is very similar to the cost of adding an entry to the primary index: either records must be shifted, or a vector of pointers to structures needs to be rearranged. As with primary indexes, the cost of doing this decreases greatly if the secondary index can be read into memory and changed there. Note that the key field in the secondary index file is stored in canonical form (all of the composers' names are capitalized), since this is the form we want to use when we are consulting the secondary index.

Record Deletion - Deleting a record usually implies removing all references to that record in the file system. So removing a record from the data file would mean removing not only the corresponding entry in the primary index but also all of the entries in the secondary indexes that refer to this primary index entry. The problem with this is that secondary indexes, like the primary index, are maintained in sorted order by key. Consequently, deleting an entry would involve rearranging the remaining entries to close up the space left open by deletion.

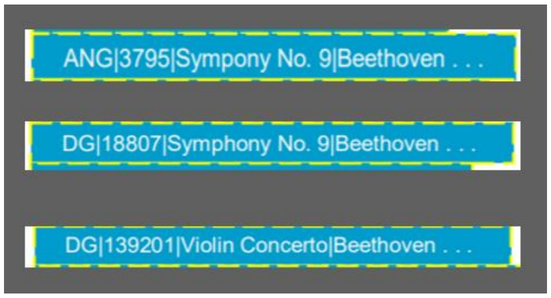
Record Updating - That the primary key index serves as a kind of protective buffer, insulating the secondary indexes from changes in the data file. This insulation extends to record updating as well. If our secondary indexes contain references directly to byte offsets in the data file, then updates to the data file that result in changing a record's physical location in the file also require updating the secondary indexes. But, since we are confining such detailed information to the primary index, data file updates affect the secondary index only when they change either the primary or the secondary key. There are three possible situations:

- Update changes the secondary key: if the secondary key is changed, we may have to rearrange the secondary key index so it stays in sorted order. This can be a relatively expensive operation.
- Update changes the primary key: this kind of change has a large impact on the primary hey index but often requires that we update only the affected reference field (Label ID in our example) in all the secondary indexes. This involves searching the secondary indexes (on the unchanged secondary keys) and rewriting the affected fixed-length field. It does not require reordering of the secondary indexes unless the corresponding secondary key occurs more than once in the index. If a secondary key does occur more than once, there may be some local reordering, since records having the same secondary key are ordered by the reference field (primary key).
- Update confined to other fields: all updates that do not affect either the primary or secondary key fields do not affect the secondary keys index, even if the update is substantial. Note that if there are several secondary key indexes associated with a file, updates to records often affect only a subset of the secondary indexes.

Retrieval Using Combinations of Secondary Keys

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

- Find all records with composer “BEETHOVEN”
- Find all records with composer “BEETHOVEN” and title “Symphony No.9”


ANG|3795|Sympony No. 9|Beethoven . . .
DG|18807|Symphony No. 9|Beethoven . . .
DG|139201|Violin Concerto|Beethoven . . .

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

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

Improving the Secondary Index Structure: Inverted lists

Two difficulties found in the proposed secondary index structures:

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

Solution 1

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

Problems:

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

Solution 2: Inverted Lists

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

Secondary Key Index File

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

LABEL ID List File

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

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

Advantages:

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

Disadvantages:

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

Selective Indexex

We can build selective indexes, such as:

Recordings released prior to 1970, recordings since 1970.

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

Binding

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

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

Advantages of postponing binding (as in our example):

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

Disadvantages:

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

When to use a tight binding?

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

When to use the bind-at-retrieval system?

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

Prepared By,
Ranjitha J
Assistant Professor
Dept, of ISE
Atria Institute of Technology

Name of the Course: File Structure
NOTES

Consequential Processing and the Sorting of Large Files, Multi-Level Indexing and B-Trees

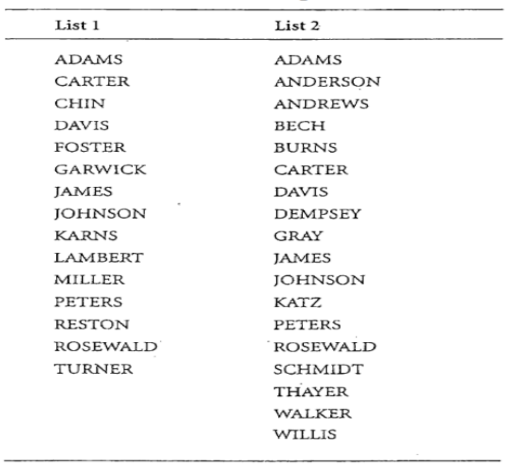
A Model for Implementing Consequential Processes

Consequential operations involve the coordinated processing of two or more sequential lists to produce a single output list. Sometimes the processing results in a merging, or union, of the items in the input lists; sometimes the goal is a matching, or intersection, of the items in the lists; and other times the operation is a combination of matching and merging.

1. Matching Names in Two Lists
2. Merging Two Lists
3. Summary of the Consequential Processing Model

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

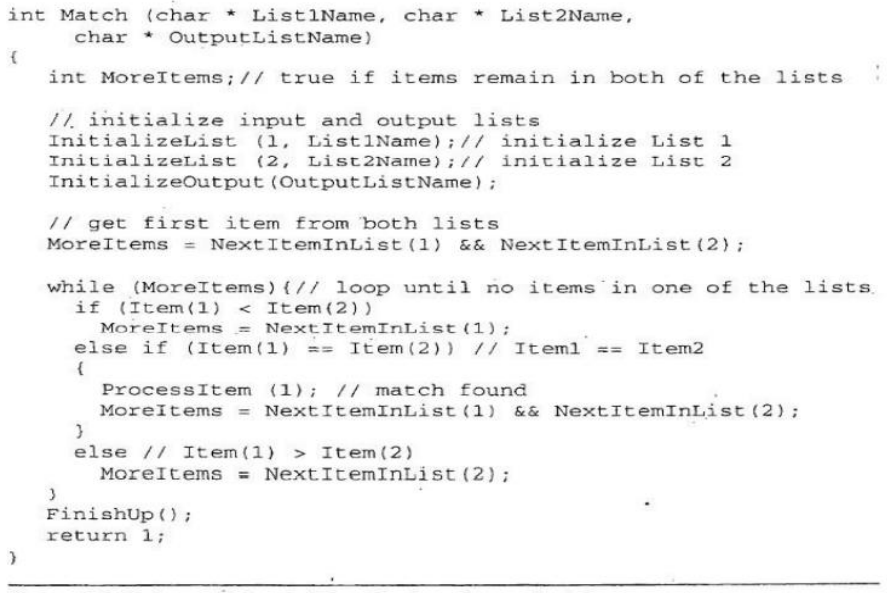
1. Matching Names in two lists
   Suppose we want to output the names common to the two lists Shown in Fig. 8.1. This operation is usually called a match operation, or an intersection. We assume, for the moment, that we will not allow duplicate names within a list and that the lists are sorted in ascending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



List 1	List 2
ADAMS	ADAMS
CARTER	ANDERSON
CHIN	ANDREWS
DAVIS	BECH
FOSTER	BURNS
GARWICK	CARTER
JAMES	DAVIS
JOHNSON	DEMPSEY
KARNS	GRAY
LAMBERT	JAMES
MILLER	JOHNSON
PETERS	KATZ
RESTON	PETERS
ROSEWALD	ROSEWALD
TURNER	SCHMIDT
	THAYER
	WALKER
	WILLIS

Figure 8.1 Sample input lists for cosequential operations.

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). Compare the two items to determine whether Item (1) is less than, equal to, or greater than Item (2):
If Item (1) is less than Item (2), we get the next item from List I;
If Item (1) is greater than Item (2), we get the next item from List 2; and
If the items are the same we 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.



```
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.

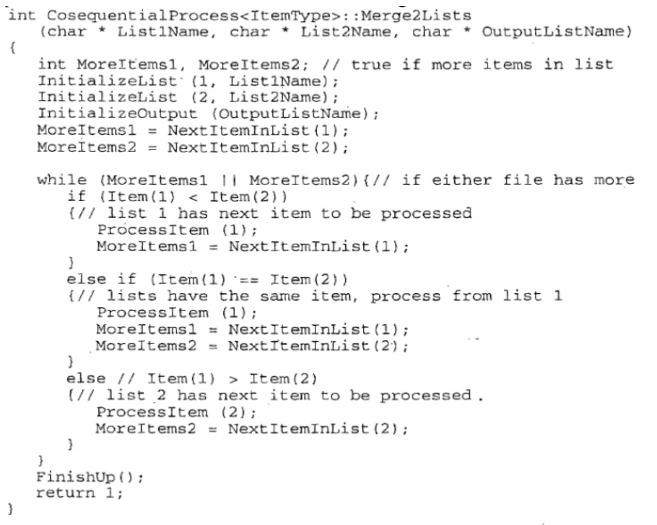
Although the match procedure appears to be quite simple, there are a number of matters that have to be dealt with to make it work reasonably well.

- Initializing: We need to arrange things in such a way that the procedure gets going properly.
- Getting and accessing the next list item: we need simple methods that support getting the next list element and accessing it.
- Synchronizing: we have to make sure that the current item from one list is never so far ahead of the current item on the other list that a match will be missed. Sometimes this means getting the next item from List 1, sometimes from List 2, sometimes from both lists.
- Handling end-of-file conditions: when we get to the end of either List I or List 2, we need to halt the program.

- Recognizing errors: when an error occurs in the data (for example, duplicate items or items out of sequence), we want to detect it and take some action.

2. Merging Two Lists

The three-way-test, single- loop model for consequential 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 high values 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.

An important difference b/w matching & merging is that with merging must read completely through each of the lists.

3. Summary of the Consequential Processing Model

The essential components of the model are:

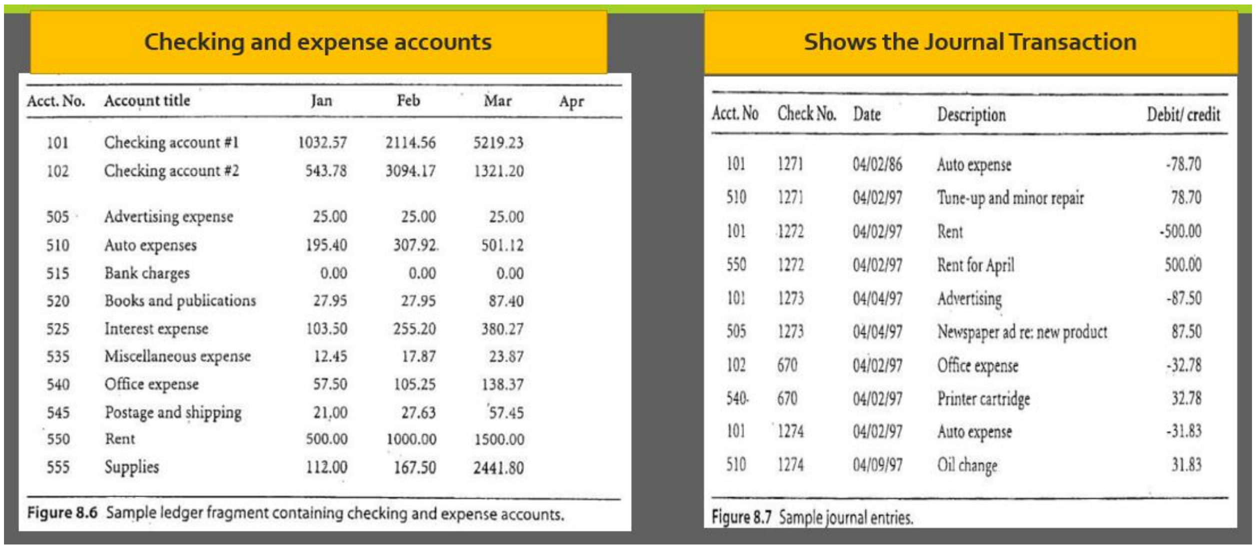
- Initializing: We need to arrange things in such a way that the procedure gets going properly.
- Getting and accessing the next list item: we need simple methods that support getting the next list element and accessing it.
- Synchronizing: we have to make sure that the current item from one list is never so far ahead of the current item on the other list that a match will be missed. Sometimes this means getting the next item from List 1, sometimes from List 2, sometimes from both lists.

- Input and output files are sequence checked by comparing the previous item value with the new item value when a record is read.
- Handling end-of-file conditions: when we get to the end of either List I or List 2, we need to halt the program.
- Recognizing errors: when an error occurs in the data (for example, duplicate items or items out of sequence), we want to detect it and take some action.

Application of the Model to a General Ledger Program

1. The Problem
2. Application of the Model to the Ledger Program

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.



Checking and expense accounts

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	

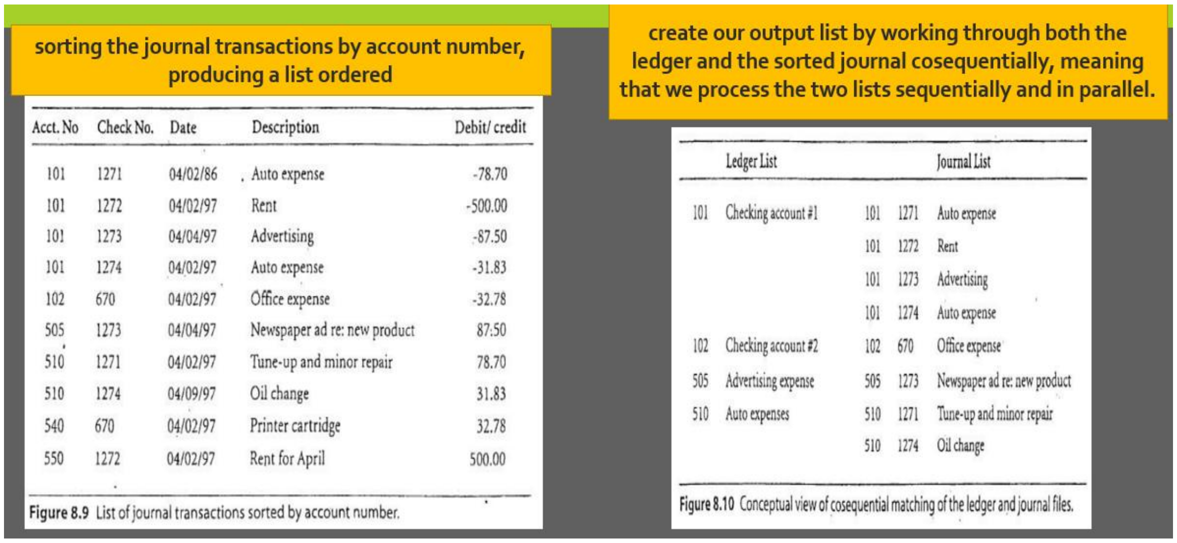
Figure 8.6 Sample ledger fragment containing checking and expense accounts.

Shows the Journal Transaction

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. This involves sorting the journal transactions by account number, producing a list ordered as in Fig. 8.9. Now we can create our output list by working through both the ledger and the sorted journal consequentially, meaning that we process the two lists sequentially and in parallel. This concept is illustrated in Fig. 8.10.


sorting the journal transactions by account number, producing a list ordered

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.

create our output list by working through both the ledger and the sorted journal cosequentially, meaning that we process the two lists sequentially and in parallel.

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

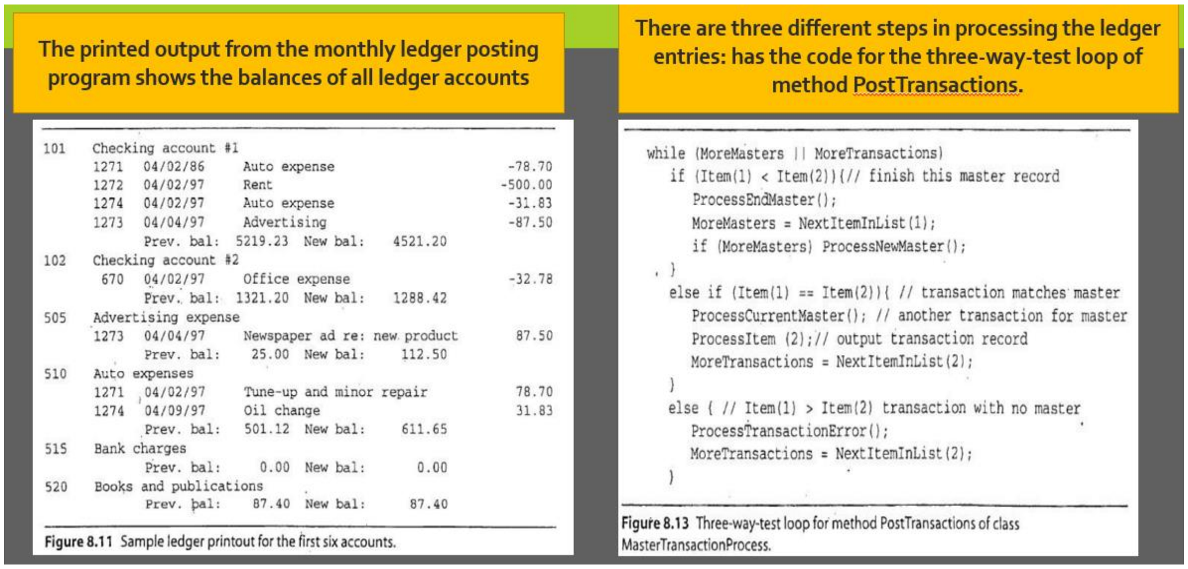
Figure 8.10 Conceptual view of cosequential matching of the ledger and journal files.

2. Application of the Model to the Ledger Program
   The monthly ledger posting program must perform two tasks:
   - It needs to update the ledger file with the correct balance for each account for the current month.
   - It must produce a printed version of the ledger that not only shows the beginning and current balance for each account 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. In summary, there are three different steps in processing the ledger entries:

- 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.
- For each transaction object that matches, we need to update the account balance.
- After the last transaction for the account, the balance line should be printed. This is the place where a new ledger record could be written to create a new ledger file.

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



The printed output from the monthly ledger posting program shows the balances of all ledger accounts

```
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.

There are three different steps in processing the ledger entries: has the code for the three-way-test loop of method PostTransactions.

```
while (MoreMasters || MoreTransactions)
   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
      MoreTransactions = NextItemInList(2);
   }
   else { // Item(1) > Item(2) transaction with no master
      ProcessTransactionError();
      MoreTransactions = 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 less-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 transaction (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.

<u>Extension of the Model to include Multiway Merging</u>

The most common application of consequential processes requiring more than two input files is a K-Way merge, in which 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.

1. A K-way Merge Algorithm
2. A Selection Tree for Merging Large Numbers of Lists

1. A K-way Merge Algorithm
   This merging operation can be viewed as a process of deciding which of two input items has the minimum value, outputting that item, then moving ahead in the list from which that item is taken. In the event of duplicate input items, move ahead in each list.
   Suppose an array of lists and array of the items (or keys) that are being used from each list in the consequential process:

```
list[0], list[1], list[2],... list[k-1]
Item[0], Item[1], Item[3],... Item[k-1]
```

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

```
int minItem = MinIndex(Item,k); // find an index of minimum item
ProcessItem(minItem); // Item(minItem) is the next output
for (i = 0; i<k; i++) // look at each list
   if (Item(minItem) == Item(i)) // advance list i
      MoreItems[i] = NextItemInList(i);
```

   If that single item(key) occurs in only one list in this case the procedure becomes simpler.

```
int minI = minIndex(Item,k); // find index of minimum item
ProcessItem(minI); // Item[minI] is the next output
MoreItems[minI]=NextItemInList(minI);
```

   The resulting merge procedure clearly differs in many ways from our initial three-way-test, single-loop merge for two lists. Here we determine which lists have the key with the lowest value, output that key, move ahead one key in each of those lists, and loop again. The procedure is as simple as it is powerful.

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 $\log_2 K$.
   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.

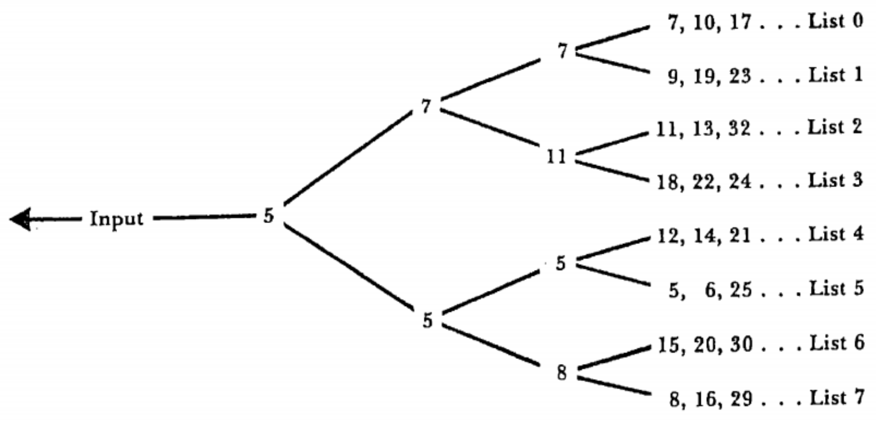


Figure 8.15 Use of a selection tree to assist in the selection of a key with minimum value in a *K*-way merge.

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.
- Write the file back to disk.

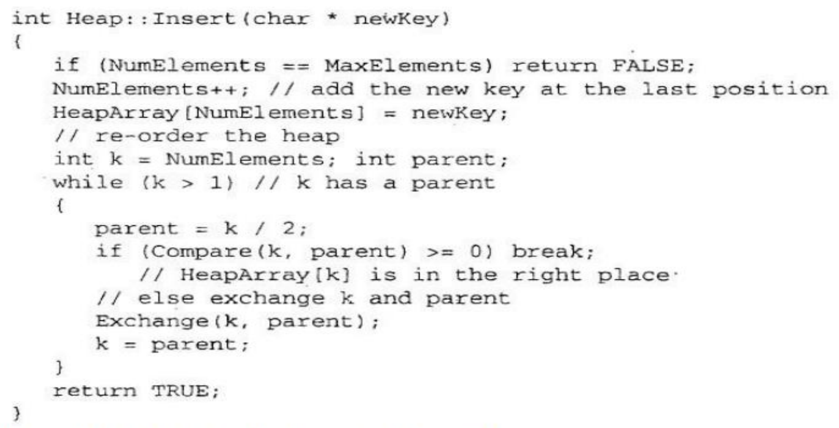
The total time taken to sort the file is the sum of the times for the three steps. This procedure is much faster than sorting the file in place, on the disk, because both reading and writing are sequential and each record is read once and written once.

The three operations involved in sorting a file that is small enough to fit into memory, is there any way to perform some of them in parallel? If we have only one disk drive, clearly can't overlap the reading and writing operations, but how about doing either the reading or writing (or both) at the same time that can sort the file?

1. Overlapping Processing and I/O: Heapsort
   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 there an 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 solves the problem by keeping all of the keys in a structure called a heap.
   A heap is a binary tree with the following properties:
   - 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. Conversely, the index of the parent of node j is[j/2].

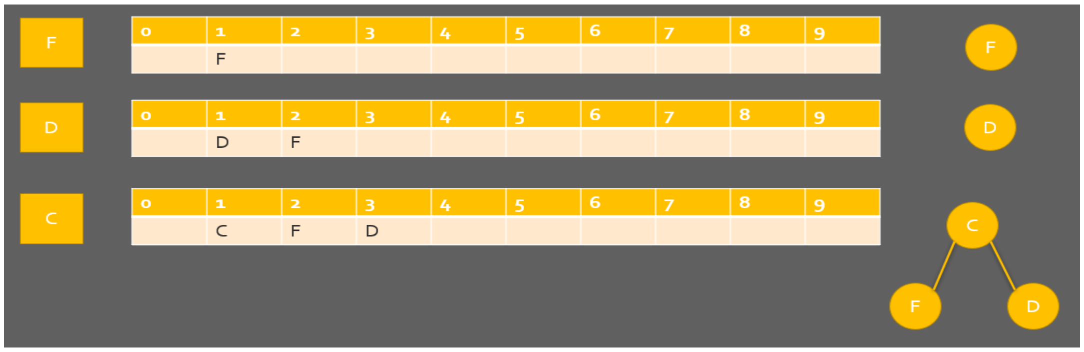
2. Building the heap while reading the file
   Heap sort algorithm for sorting in decreasing order.
   - Build a min heap from the i/p data.
   - At this point, the smallest item is stored at the root of the heap by 1. finally, heapify the root of the tree.
   - Repeat step2 while the size of the heap is less.

   The main members of s simple class Heap and its Insert methods that adds a string to the heap.



```
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;
}
```

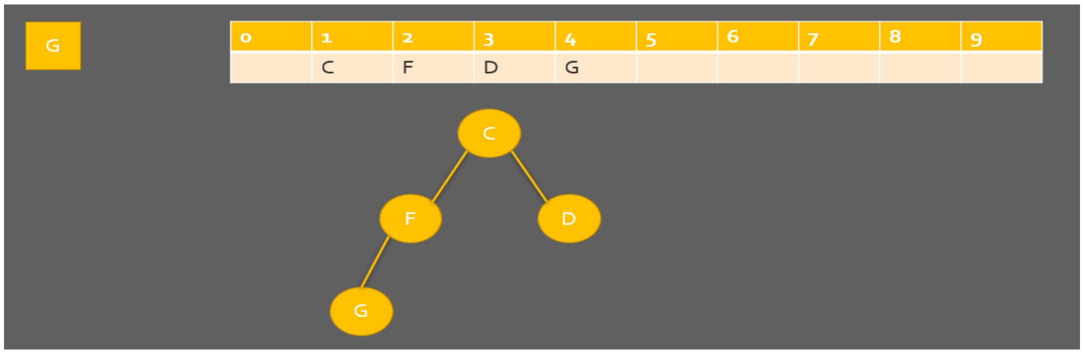
Figure 8.17: Insert method to insert key in heap



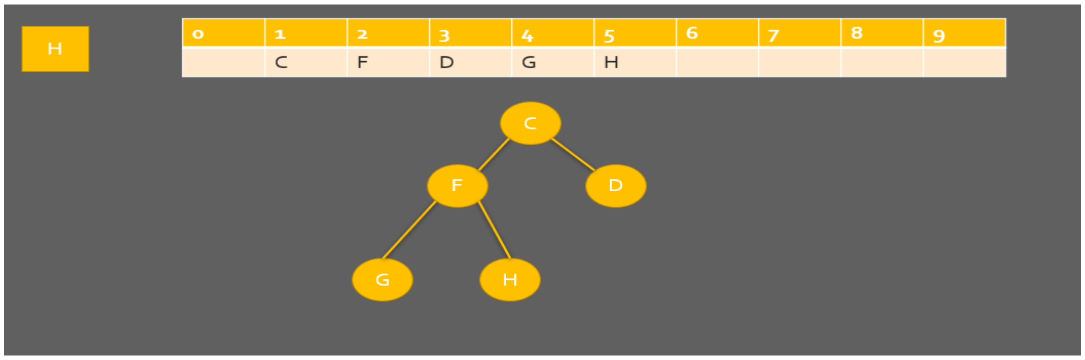
0	1	2	3	4	5	6	7	8	9
	F								

0	1	2	3	4	5	6	7	8	9
	D	F							

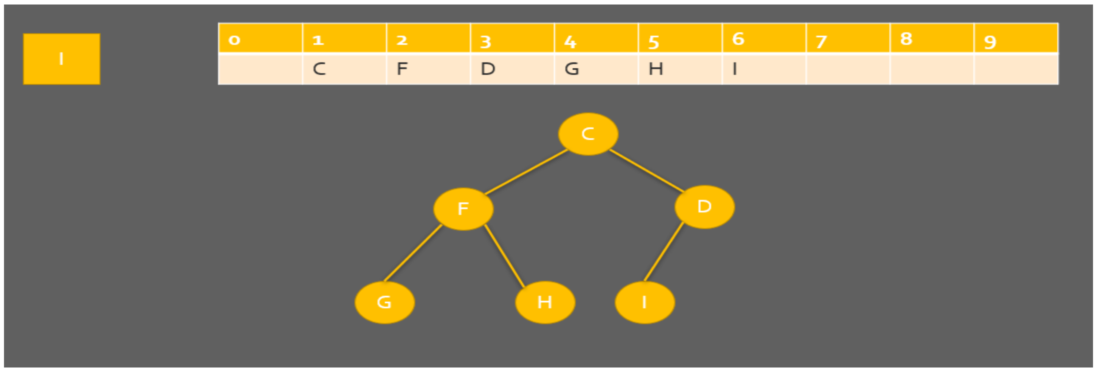
0	1	2	3	4	5	6	7	8	9
	C	F	D						



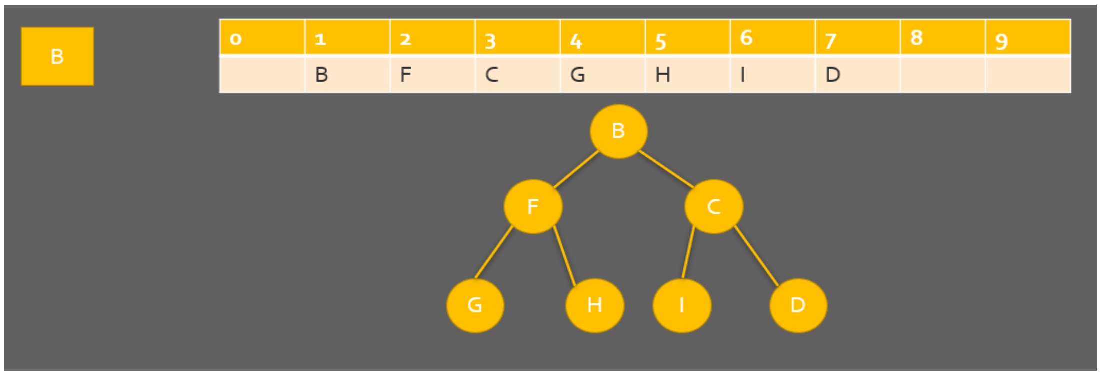
0	1	2	3	4	5	6	7	8	9
	C	F	D	G					



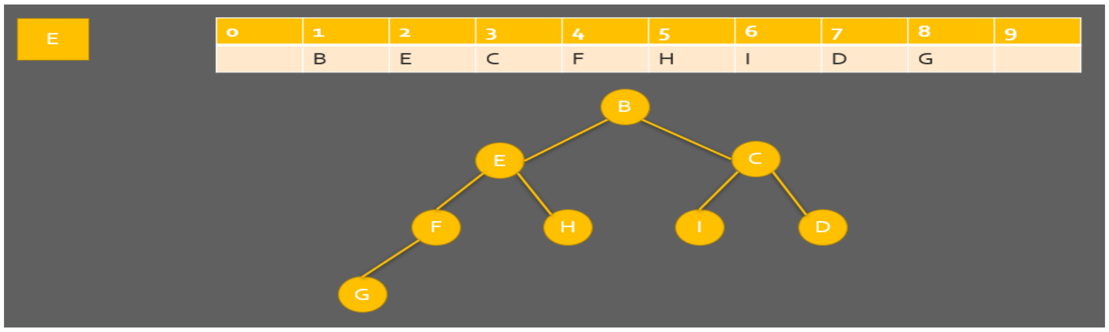
0	1	2	3	4	5	6	7	8	9
	C	F	D	G	H				



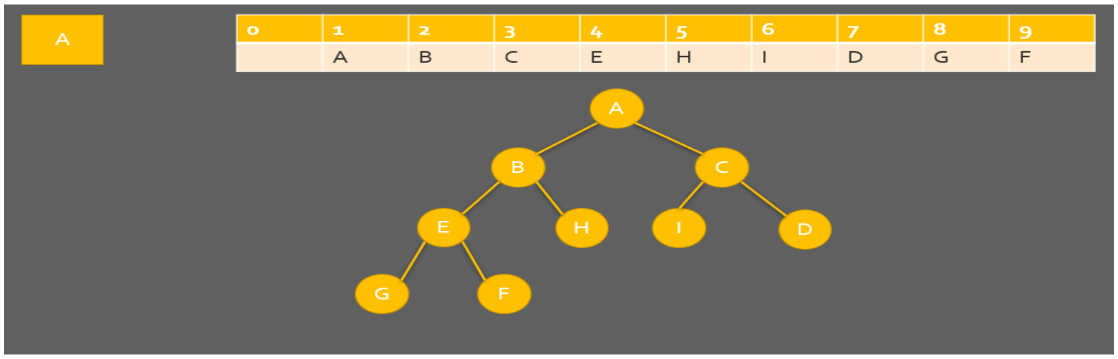
0	1	2	3	4	5	6	7	8	9
	C	F	D	G	H	I			



0	1	2	3	4	5	6	7	8	9
	B	F	C	G	H	I	D		



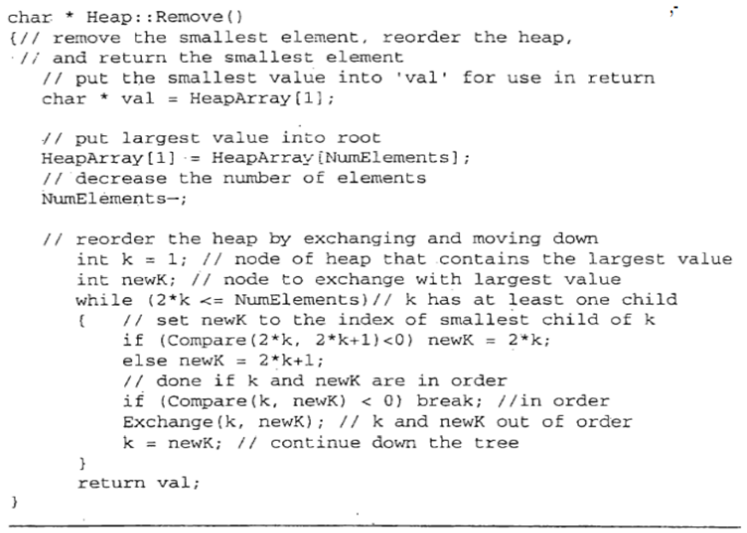
0	1	2	3	4	5	6	7	8	9
	B	E	C	F	H	I	D	G	



0	1	2	3	4	5	6	7	8	9
	A	B	C	E	H	I	D	G	F

If you want to learn in detail – Refer this link
https://drive.google.com/file/d/1RnKDX0VTgXrAyaecmcfxtMMuZLUHs-Tt/view?usp=sharing

3. Sorting while writing to the file
   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.



```
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.

<u>Merging as a way of sorting large files on disk</u>

Characteristics of the file to be sorted:

80,00,000 records
Size of a record = 100 bytes
Size of the key = 10 bytes

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

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

Idea:

- 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 = 100000 records
  Number of runs = 80 runs

1. How much time does a merge sort take?

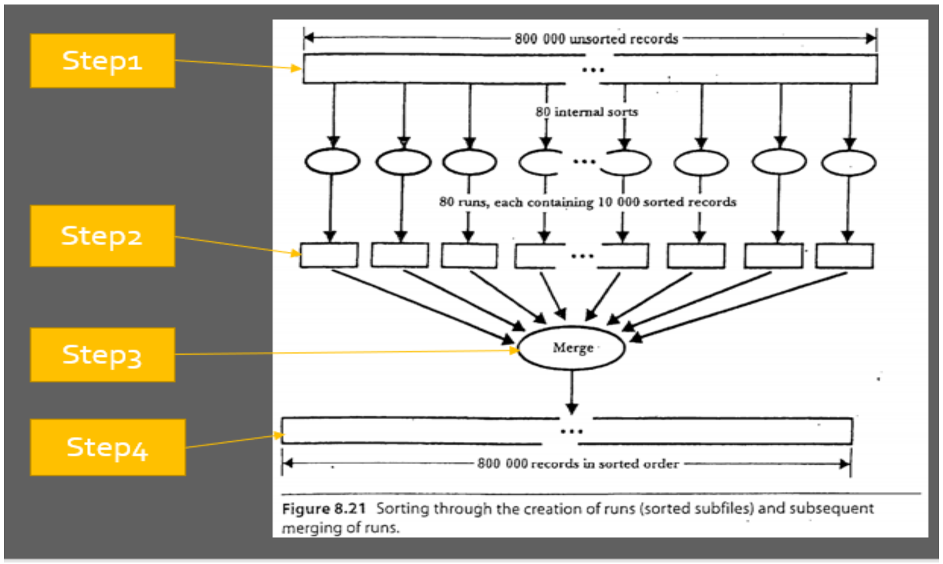

Step1
Step2
Step3
Step4
800 000 unsorted records
80 internal sorts
80 runs, each containing 10 000 sorted records
Merge
800 000 records in sorted order

Figure 8.21 Sorting through the creation of runs (sorted subfiles) and subsequent merging of runs.

- Reading records into memory for sorting and forming runs
  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. We keep these two times separate because, as we see later in our calculations, the role that each plays can vary significantly depending on the approach used.
- Writing sorted runs to disk
  The two steps above are done as follows:
  Read a chunk of 10 MEGS; write a chunk of 10 MEGS (repeat this 80 times)

In terms of basic disk operations, we spend:
For reading: 80 seeks[1] + transfer time for 800 MB
Same for writing.

- Reading sorted runs into memory for merging
  In order to minimize "seek" read one chunk of each run, so 80 chunks. Since the memory available is 10 MB each chunk can have 10,000,000/80 bytes = 125,000 bytes = 1,250 records.
  How many chunks to be read for each run?
  Size of a run/size of a chunk = 10,000,000/125,000 = 80
  Total number of basic "seeks" = Total number of chunks (counting all the runs) is 80 runs * 80 chunks/run = 80^2 chunks.
  Reading each chunk involves basic seeking.
- Writing 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: 200000000/200000=4,000.

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

2. Sorting a file that is ten times larger
   In sorting phase, chunks of data small enough to fit in main memory are read stored, and written out to a temporary file. In merge phase, the sorted sub-files are combined into a single large file. Finally merge the resulting runs together into successively bigger runs, until the file is sorted.

3. The cost of increasing the file size
   The principle that as files grow large, so can except the time required for our merge sort to increase rapidly. It would be nice if could find some ways to reduce this time. There are several ways:
   - Allocate more hardware (e.g., disk drives, memory), and I/O channel.
   - Perform the merge in more than one step – this reduces the order of each merge and increases the run sizes.
   - Algorithmically increase the length of each run.
   - Find ways to overlap I/O operations.

4. Hardware-based improvements
    - Increasing the amount of memory
      Increasing memory space ought to have a substantial effect on total sorting time. A larger memory size means longer and fewer initial runs during the sort phase, and it means fewer seeks per run during the merge phase. The product of fewer runs and fewer seeks per run means a substantial reduction in total seeks.
    - Increasing the number of disk drives
      If a separate read/write head for every run and no other users contending for use of the same read/write heads, there would be no delay due to seek time after the original runs are generated. The primary source of delay would now be rotational delays and transfers, which would occur every time a new block had to be read.
    - Increasing the amount of memory.
      If there is only one I/O channel, no two transmissions can occur at the same time, and the total transmission time is the one we have computed. But if there is a separate I/O channel for each disk drive, I/O can overlap completely.

5. Decreasing the number of seeks using multiple step merges
   The merge pattern that would minimize the number of comparisons for our sample problem, in which we want to merge 800 runs, would be the 800-way merge considered. Looked at from a point of view that ignores the cost of seeking.

6. Increasing run length using replacement selection
   Replacement selection can be implemented as follows:

   1. Read a collection of records and sort them using heapsort. This creates a heap of sorted values. Call this heap the primary heap.

   2. Instead of writing the entire primary heap in sorted order (as we do in a normal heapsort), write only the record whose key has the lowest value.

   3. Bring in a new record and compare the value of its Key with that of the key that has just been output.
      a. 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.

b. If the new record's key value is lower, place the record in a secondary heap of records with key values lower than those already written. (It cannot be put into the primary heap because it cannot be included in the run that is being created.)

4. 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.

7. Replacement selection plus multiple steps merging
   While these comparisons highlight the advantages of replacement selection over memory sorting, we would probably not in reality choose the one-step merge patterns. And we have seen that two-step merges can result in much better performance than one-step merges.

8. Using two disk drives with replacement selection
   Two disk drives to which we can assign the separate dedicated tasks of reading and writing during replacement selection. One drive, which contains the original file, does only input, and the other does only output. This has two very nice results: (1) it means that input and output can overlap, reducing transmission time by as much as 50 percent; and (2) seeking is virtually eliminated.

9. More drives? More processors?
   Possibilities include the following:
   - Mainframe computers, many of which spend a great deal of their time sorting, commonly come with two or more processors that can simultaneously work on different parts of the same problem.
   - Vector and array processors can be programmed to execute certain kinds of algorithm orders of magnitude faster than scalar processors.
   - Massively parallel machines provide thousands, even millions, of processors that can operate independently and at the same time communicate in complex ways with one another.
   - Very fast local area networks and communication software make it relatively easy to parcel out different parts of the same process to several different machines.

10. Effects of multiprogramming
    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.

11. A conceptual toolkit for external sorting
    - 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.

Multilevel Indexing and B-Trees

Invention of the B-Tree

Invented in 1969, B-trees are still the prevailing data structure for indexes in relational databases and many file systems (Comer 1979), (Weikum and Vossen 2002). Large means that the index is too large for main memory and must be stored on a secondary store like a hard disk (HD).

Statement of the Problem

The main problem with keeping an index on secondary storage is, accessing secondary storage is slow. This can be broken into two more specific problems.

1. Searching the index must be faster than binary searching

   Searching for a key on a disk often involves seeking to different disk tracks. A search that has to look in more than three or four location before finding the key often requires more time than is desirable as seeking is very expensive.

2. Insertion and deletion must be as fast as search

   Inserting a key into an index often involves moving a large number of the other keys in the index, index maintenance becomes nearly impossible on secondary storage for indexes. Thus, insertion and deletion must have only local effect in the index rather than requiring massive reorganization.

Indexing with Binary Search Trees

The major problem with binary search tree was that the file had to be sorted order. We could sort the file once, but maintaining the sorted file was difficult. Every time a record was added we need to find the exact location and move all the records such that we create the space to insert the new record in order to overcome the problem of maintaining the files in sorted order we have the tree structure, using the concept of indexing we can implement binary keys into the files we no longer have to sort the file on addition of every new key to the file we need only link it to the appropriate leaf node to the tree. The following figure shows the binary search tree representation of the list of keys.

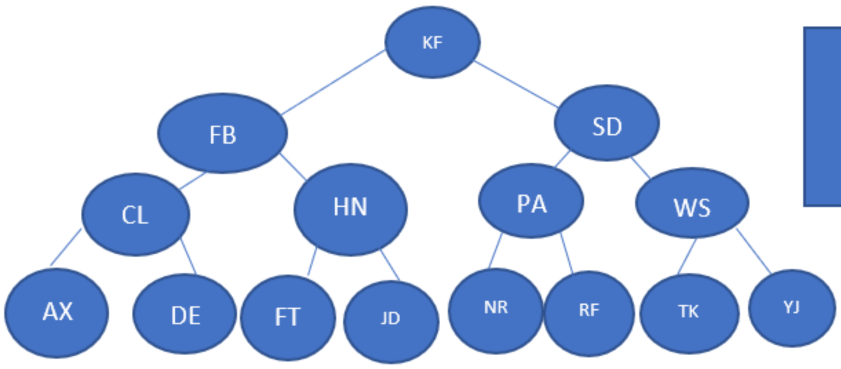
1. AVL Trees
2. Paged Binary Trees
3. Problems with Paged Trees

BS Tree from sorted linked list. Constructing from sorted array in O(n) time is simpler as we can get the middle element in O(1) time. Following is a simple algorithm

1. Get the middle of the array & make it root =15/2=7~~.5~~
2. Recursively do same for left half & right half
   a) Get the middle of left half & make it left child of the root created in step1
   b) Get the middle of right half & make it right child of the root created in step1.

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

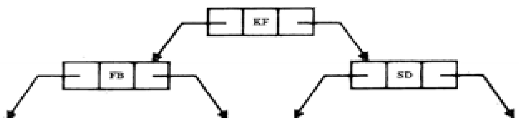
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
AX	CL	DE	FB	FT	HN	JD	KF	NR	PA	RF	SD	TK	WS	YJ



=15/2=7~~.5~~

=7/2=3

=7/2=3



left	info	right

ROOT = 4

Index Node	Key	Left Child	Right Child
0	HN		
1	SD	6	5
2	CL	7	8
3	FB	2	0
4	KF	3	1
5	WS		
6	PA		
7	AX		
8	DE		
9	NR		
10	KF		
11	JD		
12	FT		
13	TK		
14			

1. AVL(Adelson, Velski,Landis) Trees

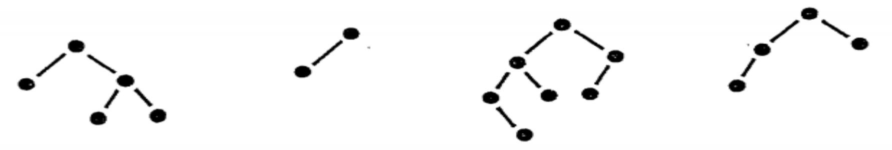


Figure 9.8 AVL trees.

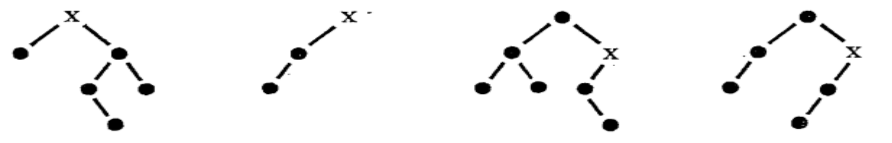
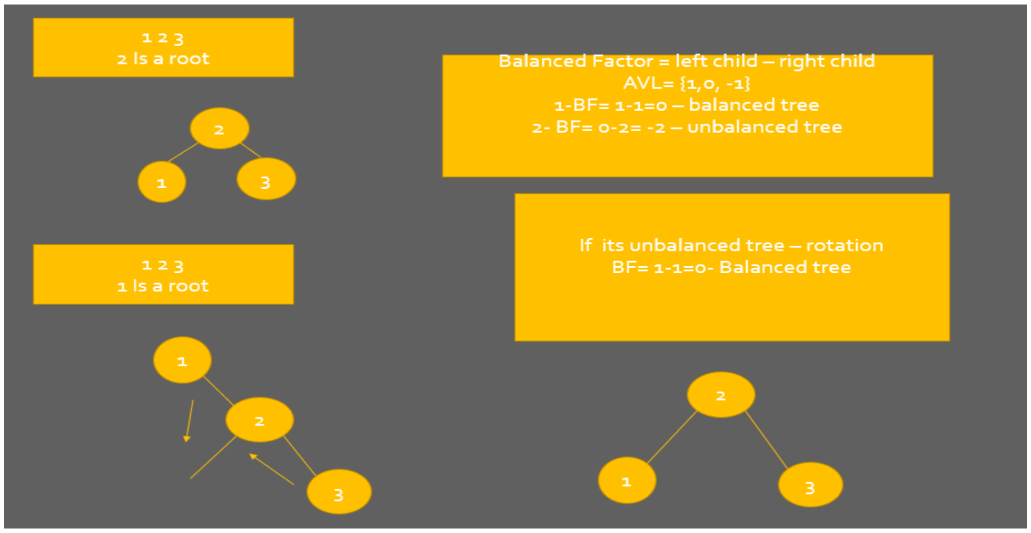


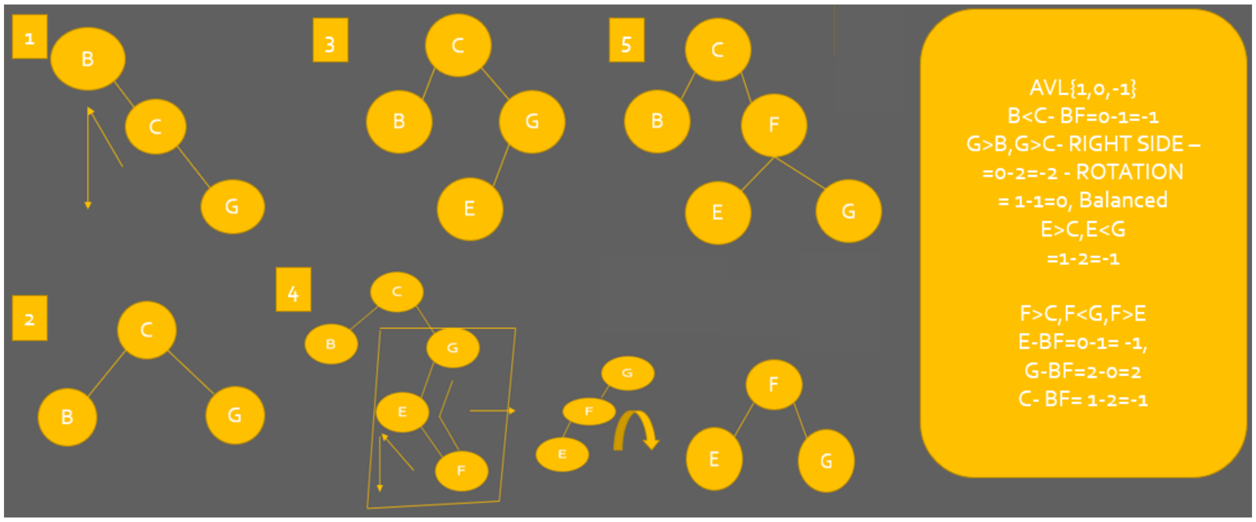
Figure 9.9 Trees that are not AVL trees.

Example -


1 2 3
2 Is a root
2
1
3
Balanced Factor = left child – right child
AVL= {1,0, -1}
1-BF= 1-1=0 – balanced tree
2- BF= 0-2= -2 – unbalanced tree
If its unbalanced tree – rotation
BF= 1-1=0- Balanced tree
1 2 3
1 Is a root
1
2
3
2
1
3

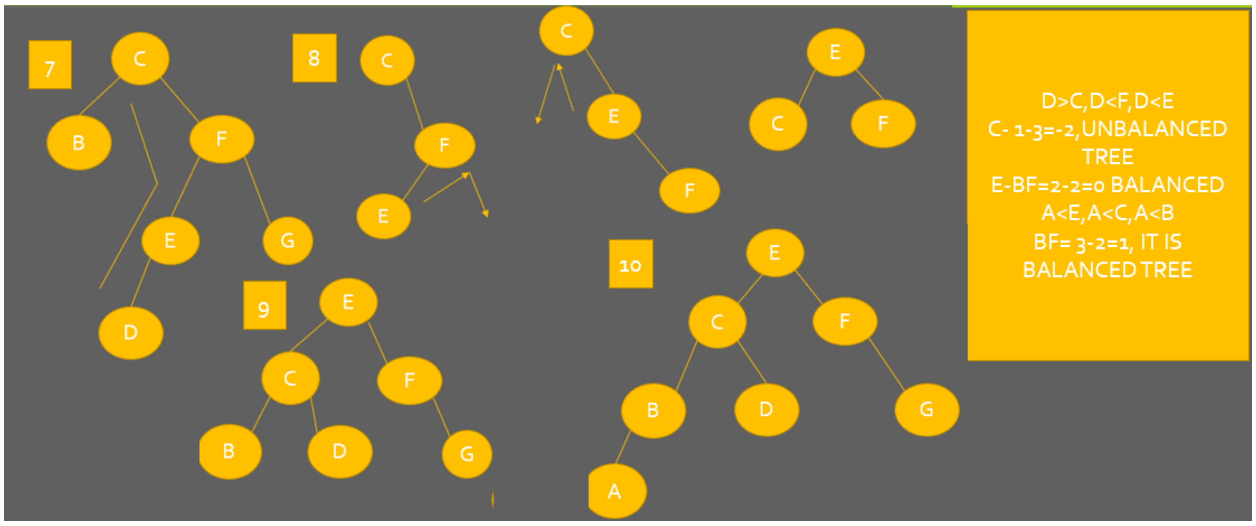
AVL tree provides different reorganization techniques or rotation in order to balance the tree constructed during the binary search tree in order to make them a Optimal Solution. The property of the AVL tree is that the height of the left subtree, the height of the right subtree range the balanced factor {0,1,-1} if the value process other than this range (0,1,-1). One of the **rotation** is applied to make it into a balanced tree.

B C G E F D A – B WILL BE A ROOT NODE



AVL{1,0,-1}
B<C- BF=0-1=-1
G>B,G>C- RIGHT SIDE –
=0-2=-2 - ROTATION
= 1-1=0, Balanced
E>C,E<G
=1-2=-1

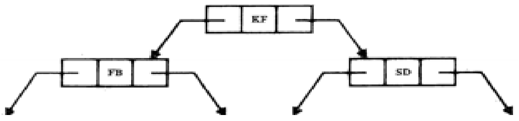
F>C,F<G,F>E
E-BF=0-1= -1,
G-BF=2-0=2
C- BF= 1-2=-1



D>C,D<F,D<E
C- 1-3=-2,UNBALANCED TREE
E-BF=2-2=0 BALANCED
A<E,A<C,A<B
BF= 3-2=1, IT IS BALANCED TREE

2. Paged Binary Trees

Disk utilization of a binary search tree is extremely inefficient that is when we read the node of a binary search tree, there are only three useful pieces of information the key value and the left subtree and the right subtree. A disk as a capacity to read more amount of data and more amount of bytes but seen we are getting a single node a lot of disk created is wasted here



So, to overcome that

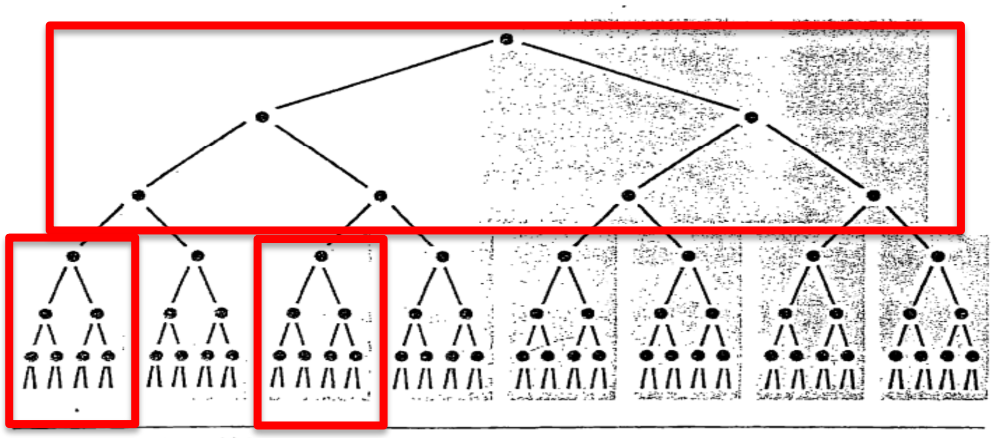


Figure 9.12 Paged binary tree.

we can used different tree structure we can call that as paged binary tree where in the binary tree is divided into number of pages. So each pages can have multiple nodes so each page will have a root node and its corresponding child node. Here we have divided node and the pages containing the 7 nodes each when a binarytree is on secondary storage if you want to search for an element which is in the last position so for that only 2 access of disk is enough.

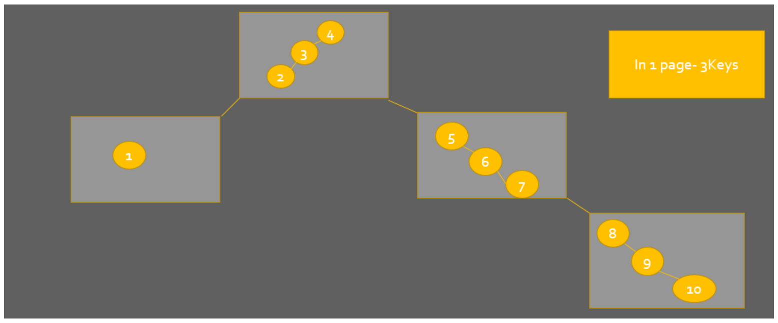
The number of seeks required for a worst-case search of a completely full, balanced binary tree is $\log_2(N+1)$.

Where N is the number of keys in the tree, the number of seeks required for the paged versions of a completely full, balanced tree is $\log_{k+1}(N+1)$

3. Problems with Paged Trees

In the construction of a paged binary tree the first element would always become the root for the successive inputs depending upon the values they are either placed to the left or to the right. If a wrong element is placed at the root node than the tree would not be balanced. In an unbalanced binary search tree we would require more number of read operations to fetch the leaf node or in other words advantage of page binary tree used only if the tree is balanced whether the tree is balanced or not is always decided by the first element placed in the tree. To overcome this problem, we have B-Tree in which the root node is automatically decided depending upon the input provided.

Example – 4 3 2 1 5 6 7 8 9 10



Multilevel Indexing: A Better Approach to Tree Indexes

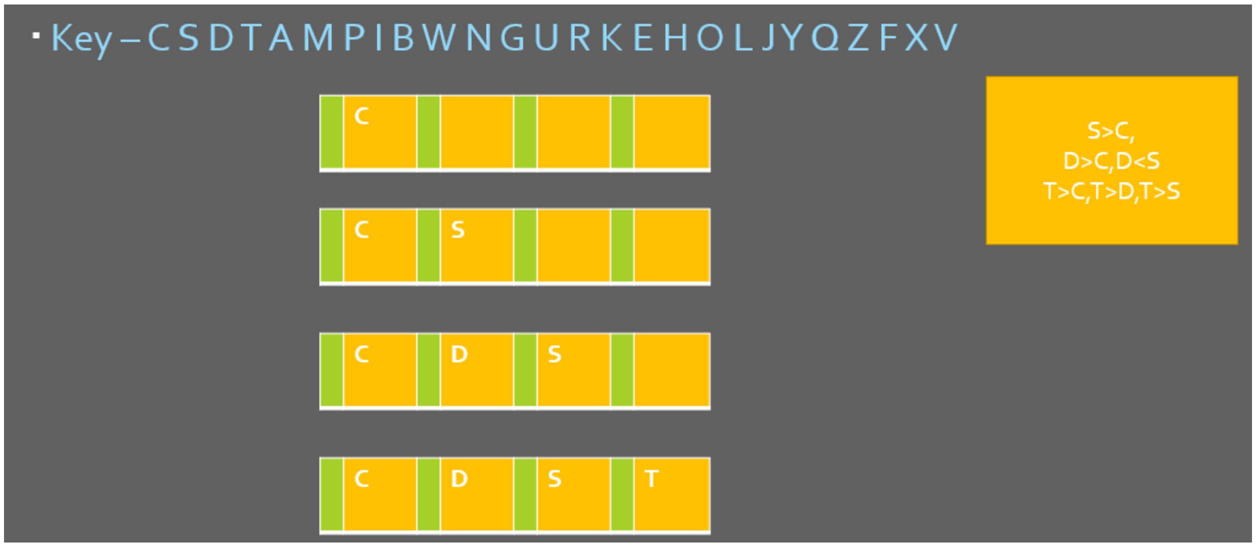
A single record index puts a limit on the number of keys allowed and for a very large file we need multi record indexes. A multi record index consists of a sequence of simple index record. The keys in one record in the list are all smaller than the keys of the next record. A binary search is possible on a file that consists of an ordered sequence. Consider a large file of 80 megabyte with 80,00000 records, 100 bytes each and 10-byte key. An index of this file has 80,00000 key reference pairs divided among a sequence of index records. If we put 100 key reference pairs in a single index record, we would have an index 80,000 records in index. To this index file we can build an index this would consist of 800 records this is the second level of index so this can be continued to third level, the index would contain only 8 records, finally to the fourth level, the index consists of just one record with 8 keys.

B-Tree: Working up from the Bottom

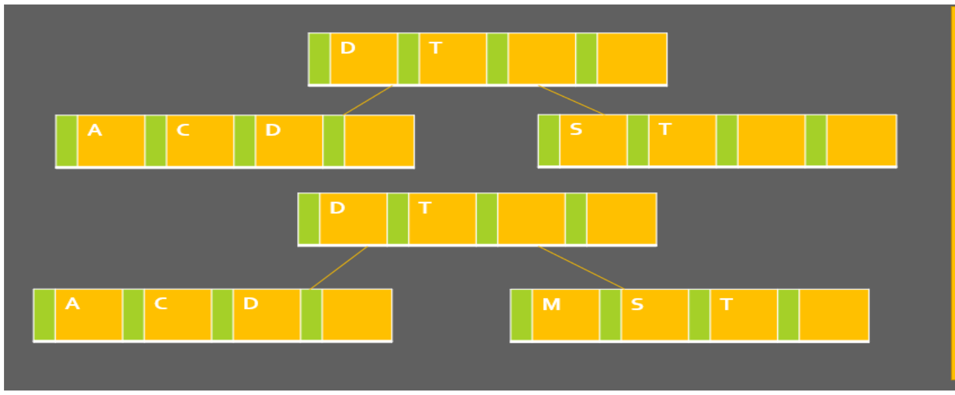
B-trees are multilevel indexes that solve the problem of linear cost of insertion and deletion. This is what makes B-trees so good, and why they are now the standard way to represent indexes. The solution is twofold. First, don’t require that the index records be full. Second, don’t shift the overflow keys to the next record; instead split an overfull record 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 an index record. Each of these records has the same maximum number of key-reference pairs, called the order of the B-tree.

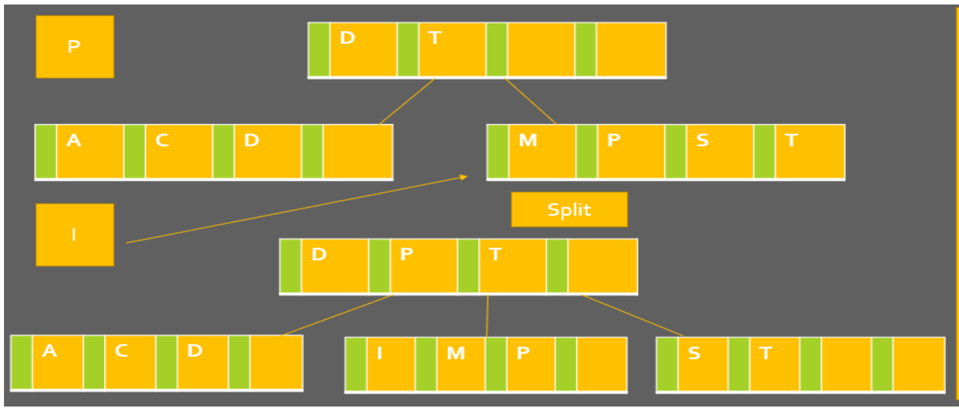
Example of Creating a B-Tree



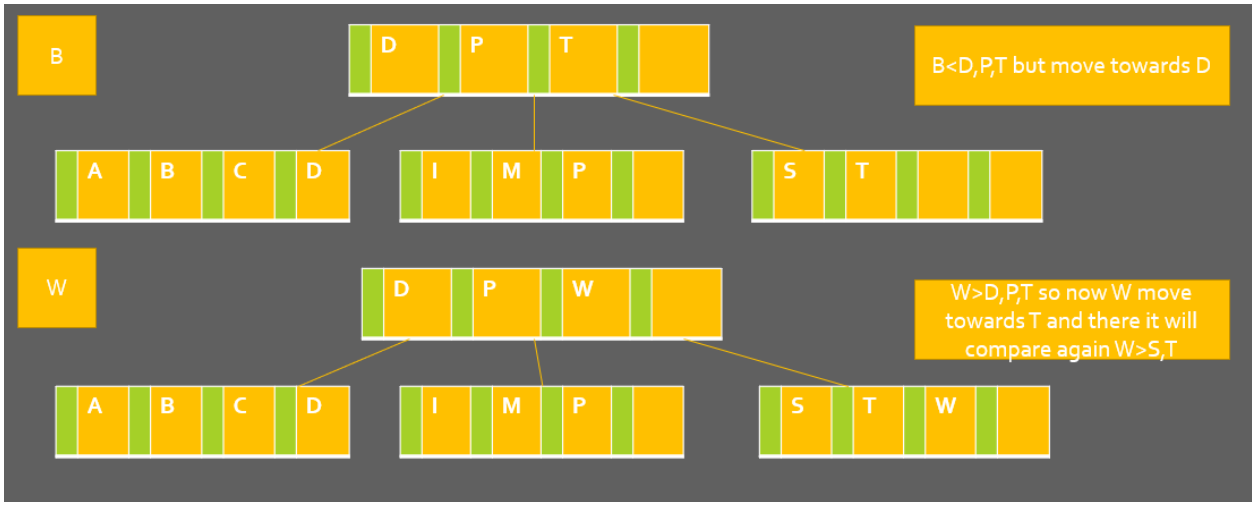
S>C,
D>C,D<S
T>C,T>D,T>S

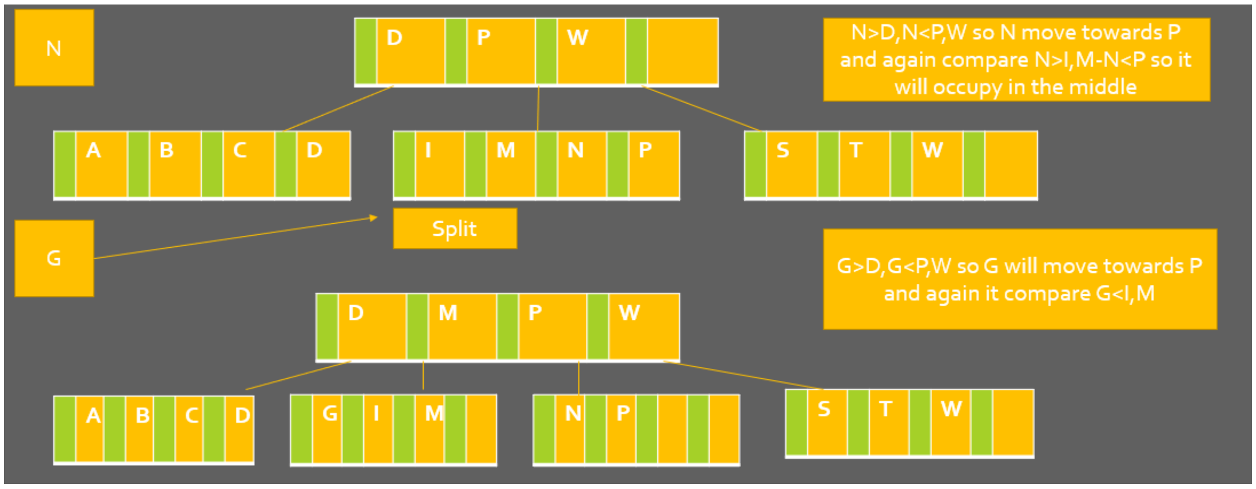


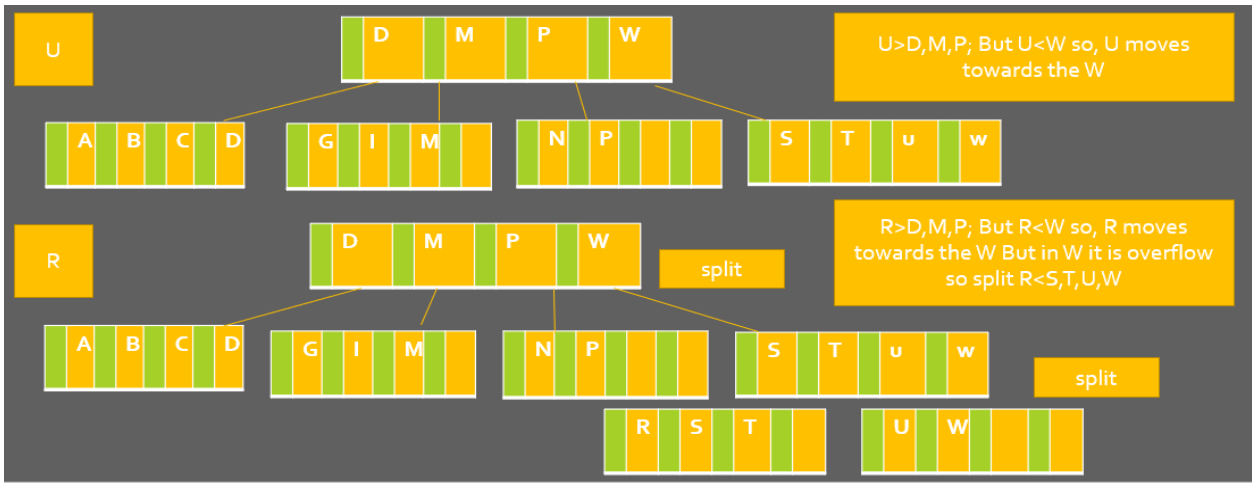
Now 'A' is lesser than all even node is full so we need to do operation in order to accomidate the data. So we need to do split-bit operation.
A<C,D,S,T but A should be inserted front.
M>D,M<T- so M will move towards T again it will compare M<S,M<T so it will place in the middle.

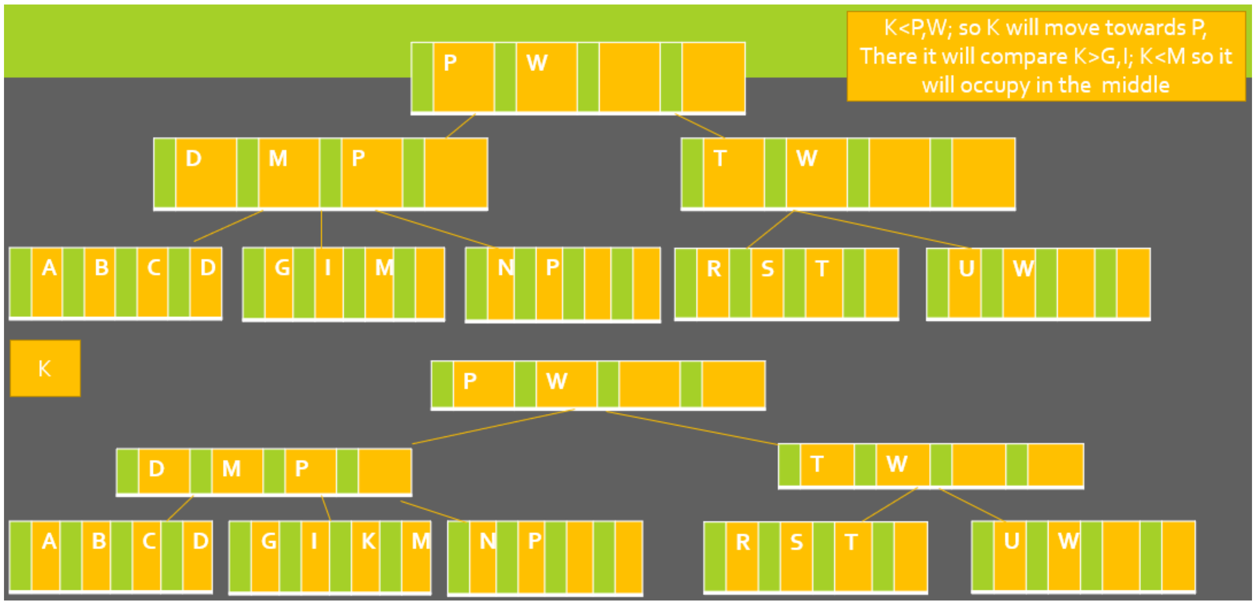


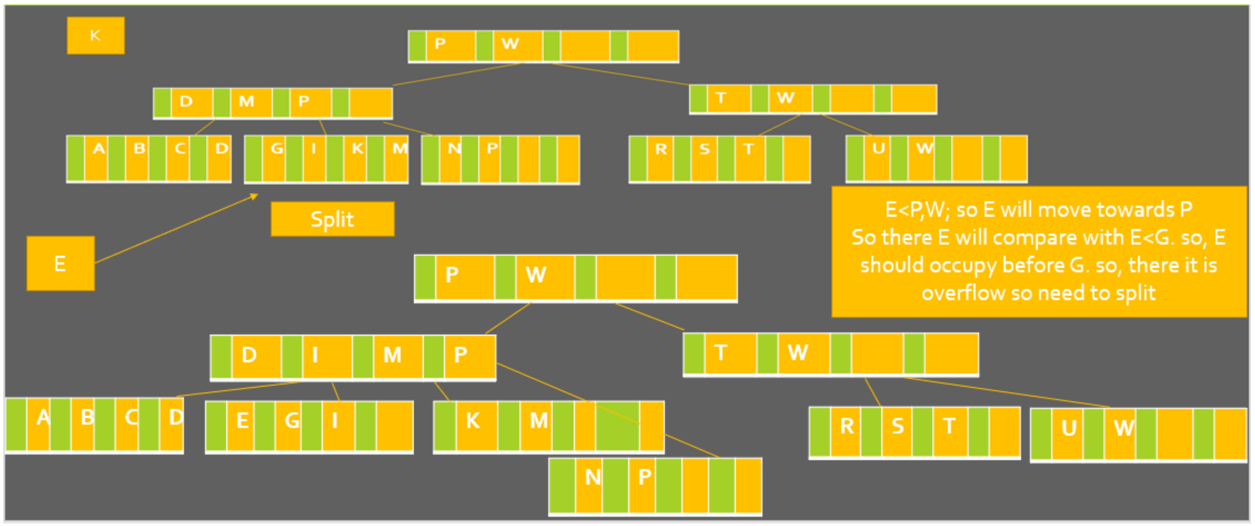
Now insert another key 'P'
P>D,P<T so P will move to a T side again it will compare P>M,P<S,T so it will place in the between the M & S.
Same steps follows it will compare I>D,I<T so it will move to a T so there again we can observe there is a overflow so again we need to split. I<M,P,S,T so I should place in before the M.

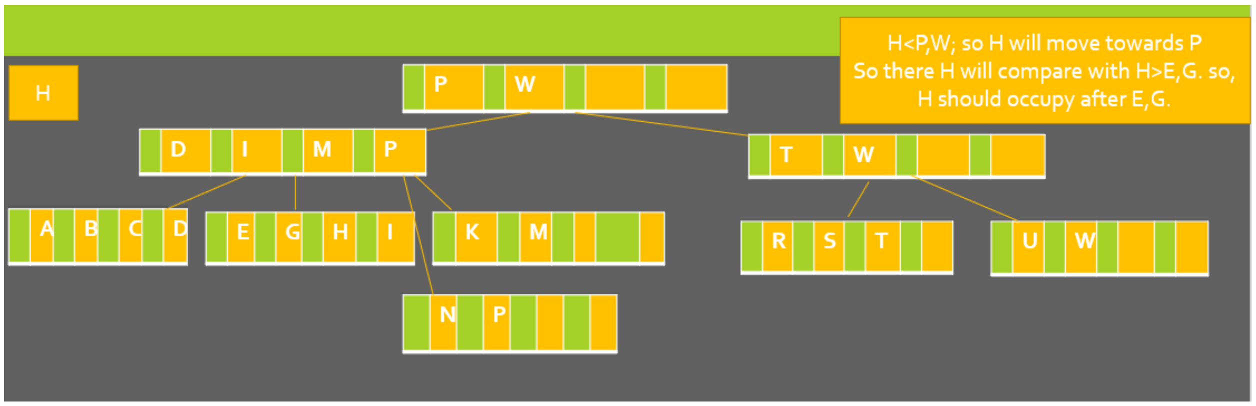

B
D P T
B<D,P,T but move towards D
A B C D
I M P
S T
W
D P W
W>D,P,T so now W move towards T and there it will compare again W>S,T
A B C D
I M P
S T W

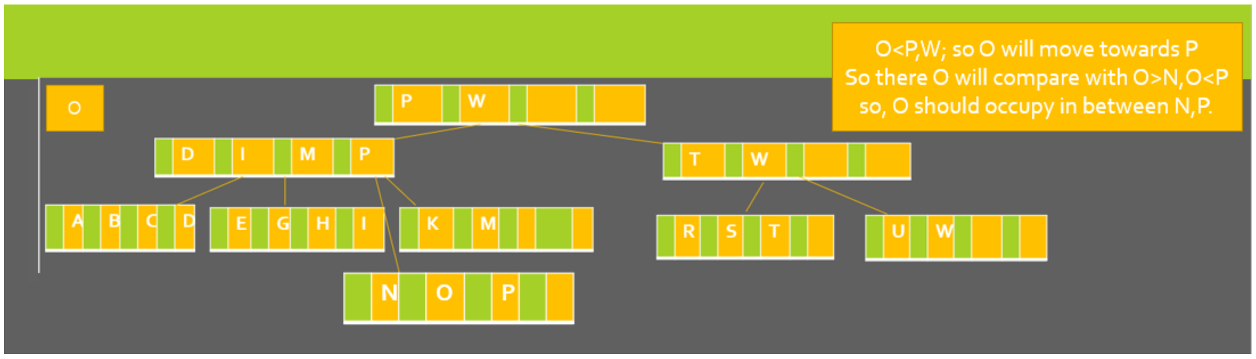

N
D P W
N>D,N<P,W so N move towards P and again compare N>I,M-N<P so it will occupy in the middle
A B C D
I M N P
S T W
G
Split
G>D,G<P,W so G will move towards P and again it compare G<I,M
D M P W
A B C D
G I M
N P
S T W


U
D M P W
U>D,M,P; But U<W so, U moves towards the W
A B C D
G I M
N P
S T u w
R
D M P W
split
R>D,M,P; But R<W so, R moves towards the W But in W it is overflow so split R<S,T,U,W
A B C D
G I M
N P
S T u w
split
R S T
U W

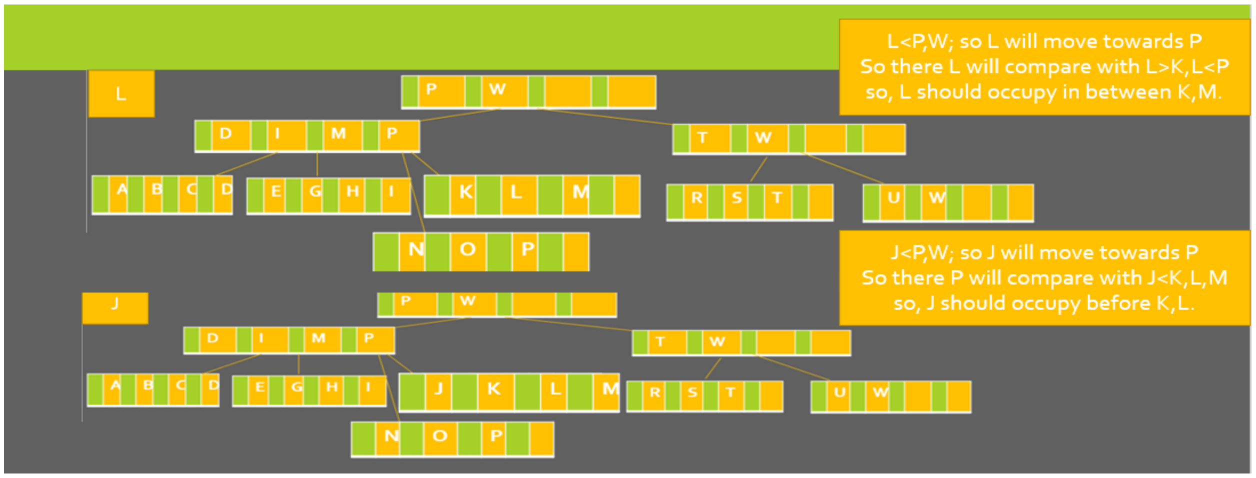

K<P,W; so K will move towards P, There it will compare K>G,I; K<M so it will occupy in the middle
P W
D M P
T W
A B C D
G I M
N P
R S T
U W
K
A B C D
G I K M
N P
R S T
U W


K
Split
E
E<P,W; so E will move towards P So there E will compare with E<G. so, E should occupy before G. so, there it is overflow so need to split
P W
D I M P
T W
A B C D
E G I
K M
N P
R S T
U W


H<P,W; so H will move towards P So there H will compare with H>E,G. so, H should occupy after E,G.
H
P W
D I M P
T W
A B C D
E G H I
K M
N P
R S T
U W

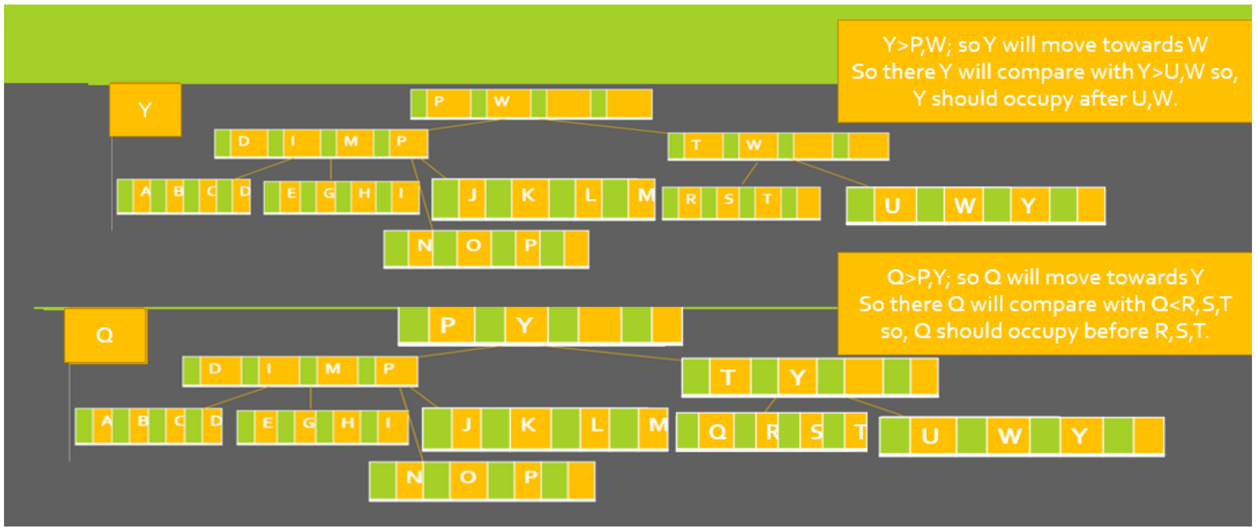


O<P,W; so O will move towards P
So there O will compare with O>N,O<P
so, O should occupy in between N,P.



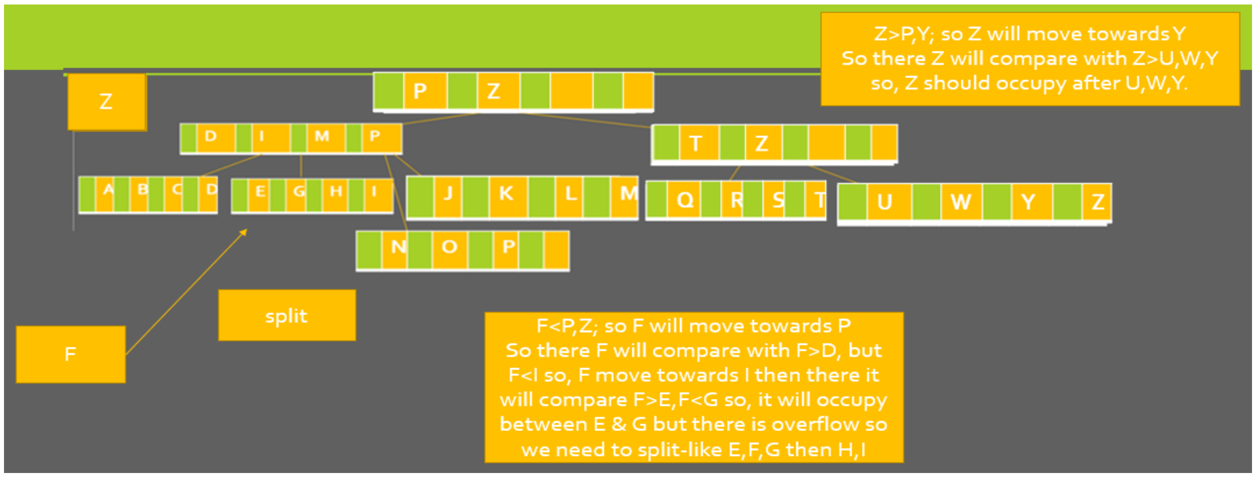
L<P,W; so L will move towards P
So there L will compare with L>K,L<P
so, L should occupy in between K,M.

J<P,W; so J will move towards P
So there P will compare with J<K,L,M
so, J should occupy before K,L.



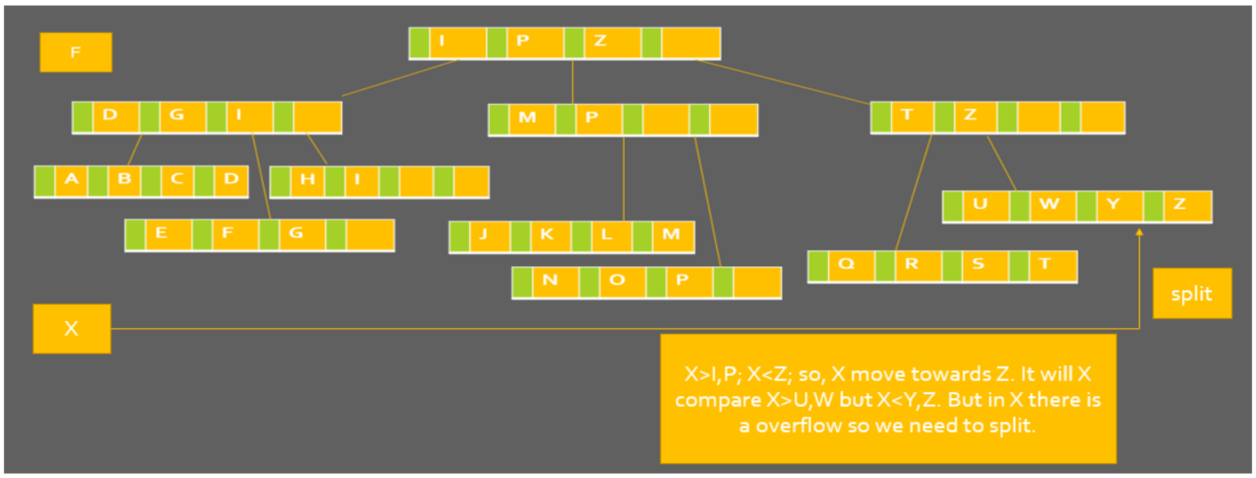
Y>P,W; so Y will move towards W
So there Y will compare with Y>U,W so,
Y should occupy after U,W.

Q>P,Y; so Q will move towards Y
So there Q will compare with Q<R,S,T
so, Q should occupy before R,S,T.

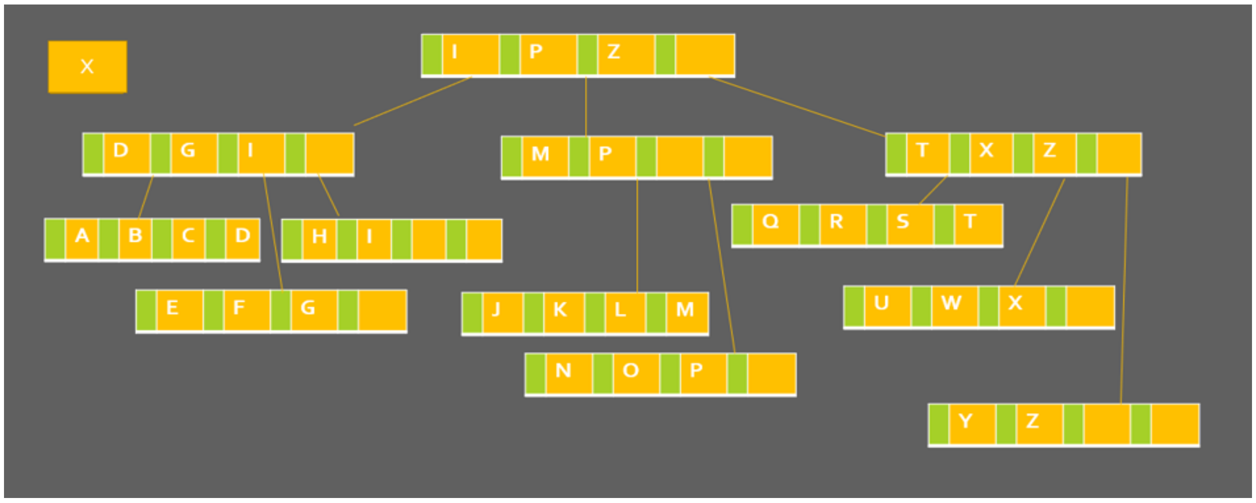


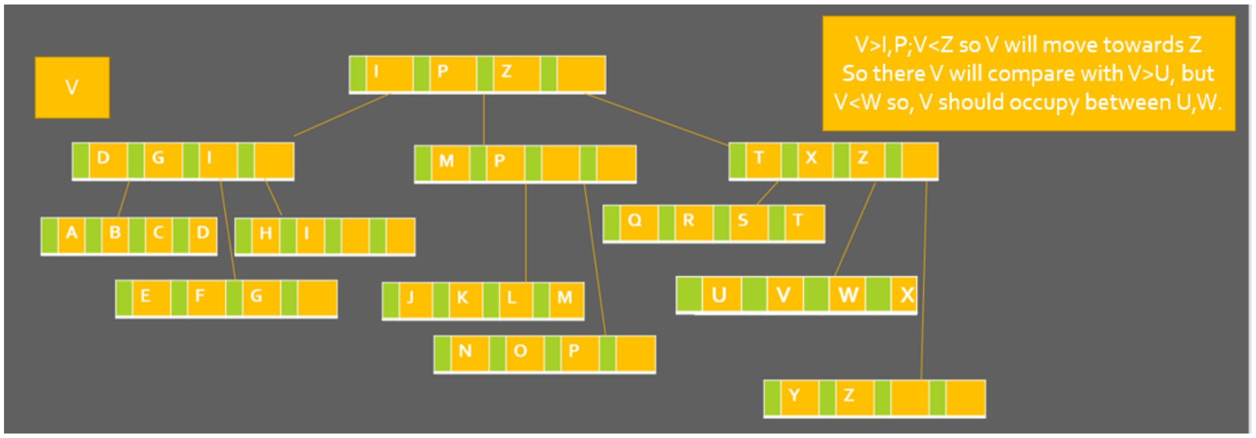
Z>P,Y; so Z will move towards Y
So there Z will compare with Z>U,W,Y
so, Z should occupy after U,W,Y.

F<P,Z; so F will move towards P
So there F will compare with F>D, but
F<I so, F move towards I then there it
will compare F>E,F<G so, it will occupy
between E & G but there is overflow so
we need to split-like E,F,G then H,I



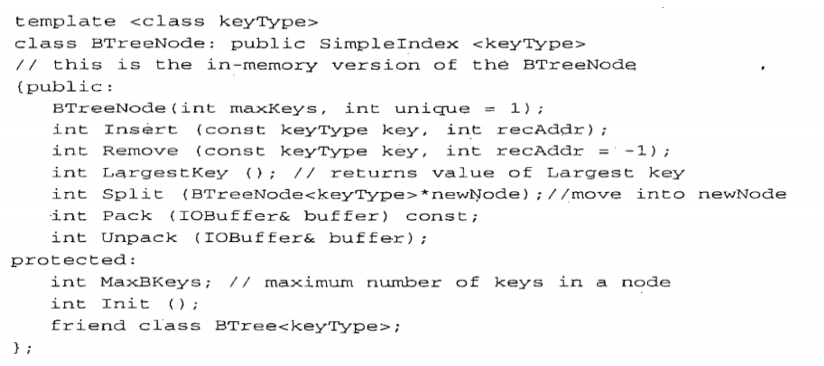
X>I,P; X<Z; so, X move towards Z. It will X compare X>U,W but X<Y,Z. But in X there is a overflow so we need to split.




V
I P Z
V>I,P;V<Z so V will move towards Z So there V will compare with V>U, but V<W so, V should occupy between U,W.
D G I
M P
T X Z
A B C D
H I
Q R S T
E F G
J K L M
U V W X
N O P
Y Z

An Object-Oriented Representation B-Tree Nodes in Memory

1. Class BTreeNode: Representing B-Tree Nodes in Memory
   A B-tree is an index file associated with a data file. Most of the operations on B-trees, including insertion and deletion, are applied to the B-tree nodes in memory. The B-tree file simply stores the nodes when they are not in memory. Hence, we need a class to represent the memory resident B-tree nodes. Note that a BTreeNode object has methods to insert and remove a key and to split and merge nodes. There are also protected members that store the file address of the node and the minimum and maximum number of keys. You may notice that there is no search method defined in the class. The search method of the base class SimpleIndex works perfectly well.
   It is important to note that not every data member of a BTreeNode has to be stored when the object is not in memory. The difference between the memory and the disk representations of BTreeNode objects is managed by the pack and unpack operations. The call to the SimpleIndex constructor creates an index record.

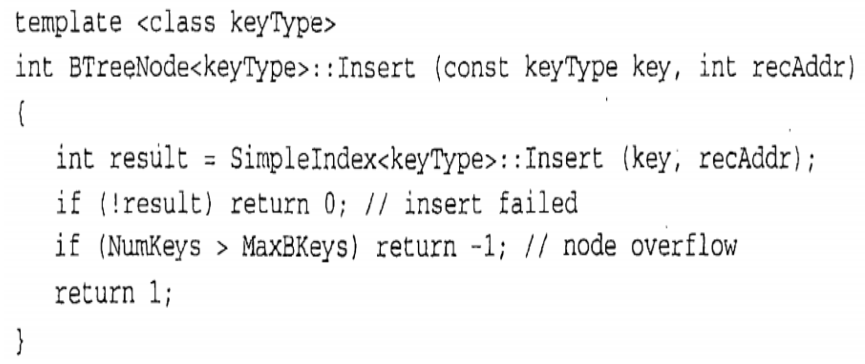


```
template <class keyType>
class BTreeNode: public SimpleIndex <keyType>
// this is the in-memory version of the BTreeNode
{public:
   BTreeNode(int maxKeys, int unique = 1);
   int Insert (const keyType key, int recAddr);
   int Remove (const keyType key, int recAddr = -1);
   int LargestKey (); // returns value of Largest key
   int Split (BTreeNode<keyType>*newNode);//move into newNode
   int Pack (IOBuffer& buffer) const;
   int Unpack (IOBuffer& buffer);
protected:
   int MaxBKeys; // maximum number of keys in a node
   int Init ();
   friend class BTree<keyType>;
};
```

Figure 9.16 The main members and methods of class BTreeNode: template class for B-tree node in memory.

For this class, the order of the B-tree node (member MaxBKeys) is one less than the value of MaxKeys, which is a member of the base class SimpleIndex. Making the index record larger allows the Insert method to create an overfull node. The caller of BTreeNode: : Insert needs to respond to the overflow in an appropriate fashion. Similarly, the Remove method can create an underfull node, Method Insert simply calls SimpleIndex: : Insert and then checks for overflow. The value returned is 1 for success, 0 for failure, and —1 for overflow:

2. Class BTree: Supporting Files of B-Tree Nodes
   class BTreé which uses in-memory BTreeNode objects, adds the file access portion, and enforces the consistent size of the nodes. This contain the definition of class BTree. Here are methods to create, open, and close a B-tree and to search, insert, and remove key-reference pairs. In the protected area of the Class, we find methods to transfer nodes from disk to memory (Fetch) and back to disk (Store). There are members that hold the root node in memory and represent the height of the tree and the file of index records. Member Nodes is used to keep a collection of tree nodes in memory and reduce disk accesses.



```
template <class keyType>
int BTreeNode<keyType>::Insert (const keyType key, int recAddr)
{
   int result = SimpleIndex<keyType>::Insert (key, recAddr);
   if (!result) return 0; // insert failed
   if (NumKeys > MaxBKeys) return -1; // node overflow
   return 1;
}
```

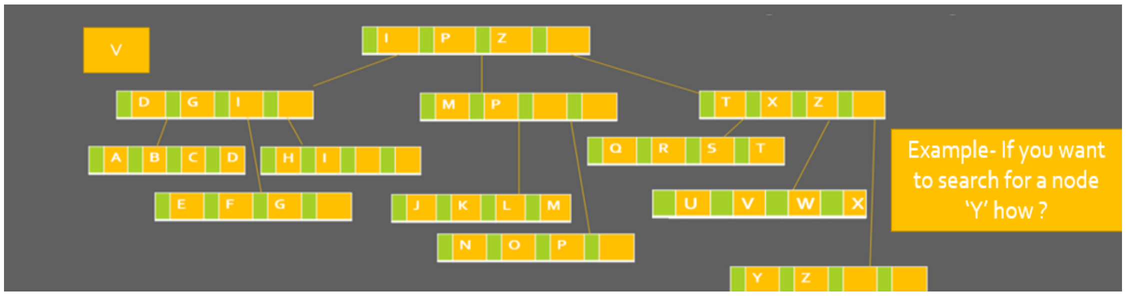
B-Tree Methods Search, Insert, and Others

1. Searching - Searching a node in a B-Tree
   - Perform a binary search on the records in the current node.
   - If a record with the search key is found, then return that record.
   - If the current node is a leaf node and the key is not found, then report an unsuccessful search.
   - Otherwise, follow the proper branch and repeat the process.

```
template <class keyType>
int BTree<keyType>::Search (const keyType key, const int recAddr)
{
    BTreeNode<keyType> * leafNode;
    leafNode = FindLeaf (key);
    return leafNode -> Search (key, recAddr);
}

template <class keyType>
BTreeNode<keyType> * BTree<keyType>::FindLeaf (const keyType key)
// load a branch into memory down to the leaf with key
{
    int recAddr, level;
    for (level = 1; level < Height; level++)
    {
       recAddr = Nodes[level-1]->Search(key,-1,0);//inexact search
       Nodes[level]=Fetch(recAddr);
    }
    return Nodes[level-1];
}
```

Figure 9.18 Method BTree::Search and BTree::FindLeaf.

Example –



2. Insertion
   There are two important observations we can make about the insertion, splitting, and promotion process:
   - It begins with a search that proceeds all the way down to the leaf level, and
   - After finding the insertion location at the léaf level, the work of insertion, overflow detection, and splitting proceeds upward from the bottom.

   Consequently, we can conceive of our iterative procedure as having three phases:

   1. Search to the leaf level, using method FindLeaf, before the iteration;

   2. Insertion, overflow detection, and splitting on the upward path;

3. Creation of a new root node, if the current root was split.

Example refer the creation of B-Tree

4. Create, Open, and Close
   Method Create has to write the empty root node into the file BTreeFile so that its first record is reserved for that root node. Method Open has to open BTreeFile and load the root node into memory from the first record in the file. Method Close simply stores the root node into BTreeFile and closes it.

5. Testing the B-Tree

```
const char * keys="CSDTAMPIBWNGURKEHOLJYQZFXV";
const int BTreeSize = 4;
main (int argc, char * argv)
{
    int result, i;
    BTree <char> bt (BTreeSize);
    result = bt.Create ("testbt.dat",ios::in|ios::out);
    for (i = 0; i<26; i++)
    {
        cout<<"Inserting "<<keys[i]<<endl;
        result = bt.Insert(keys[i],i);
        bt.Print(cout); // print after each insert
    }
    return 1;
}
```

B-Tree Nomenclature

The literature on B-trees is not uniform in its terminology

Bayer and McCreight (1972), Comer (1979), and others define the order of B-tree as the minimum number of keys that can be in a page of a tree.

Knuth (1998) avoids the problem by defining the order to be the maximum number of children (which is one more than the maximum number of keys).

The term leaf is also inconsistent. Bayer and McCreight (1972) considered the leaf level to be the lowest level of keys, but Knuth considered the leaf level to be one level below the lowest keys. There are many possible implementation choices. In some designs, the leaves may hold the entire data record; in other designs, the leaves may only hold pointers to the data record.

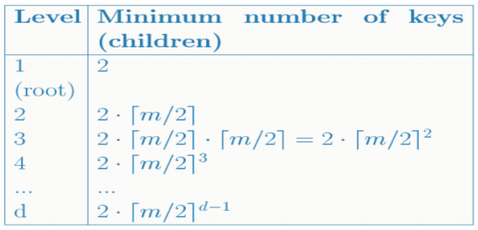
Formal Definition of B-Tree Properties

- Every page has a maximum of m descendants.
- Every page, except the root and the leaves has at least m/2 descendants.
- The root has at least two descendants
- All the leaves appear on the same level
- The leaf level forms a complete, ordered index of the associated data file.

Worst-Case Search Depth

It is important to have a quantitative understanding of the relationship between the page size of a B-tree, the number of keys to be stored in the tree, and the number of levels that the tree can extend. For example, you might know that you need to store 1 000 000 keys and that, given the nature of your storage hardware and the size of your keys, it is reasonable to consider using a B-tree of order 512 (maximum of 511 keys per page). Given these two facts, you need to be able to answer the question: In the worst case, what will be the maximum number of disk accesses required to locate a key in the tree?

For a B-tree of order m, the minimum number of descendants from the root page is 2, so the second level of the tree contains only 2 pages. Each of these pages, in turn, has at least [m/2 | descendants. The third level, then, contains 2 x [m/2] pages. Since each of these pages, once again, has a minimum of [m/2] descendants, the general pattern of the relation between depth and the minimum number of descendants takes the following form:



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

Assume that we have N keys in the leaves.

N ≥ 2 × [m/2]d-1

So, d ≤1+ log[m/2] (N/2).

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

$d \leq 1 + \log[256]\ (N/2).$

$d \leq 3.37$

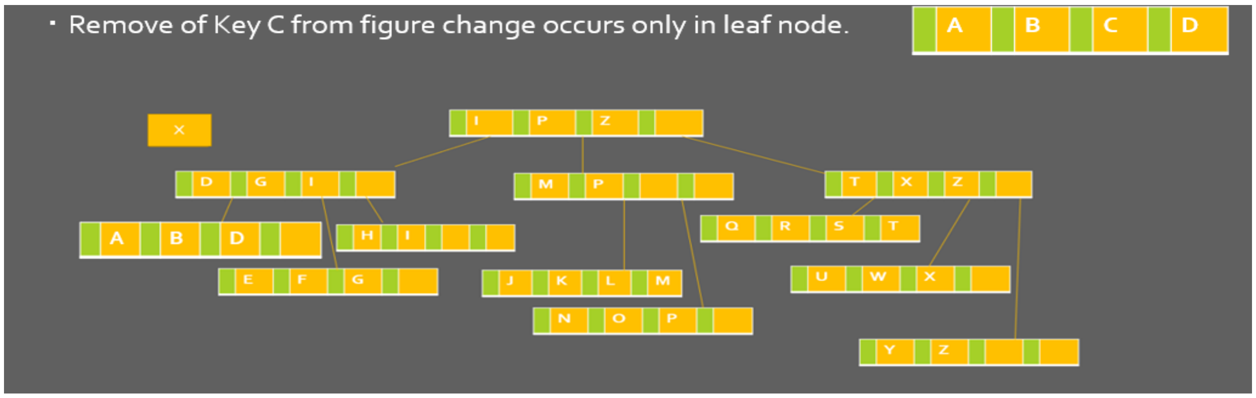
So we can say that given 1000000keys, a Btree of order 512 has a depth of no more than three levels.

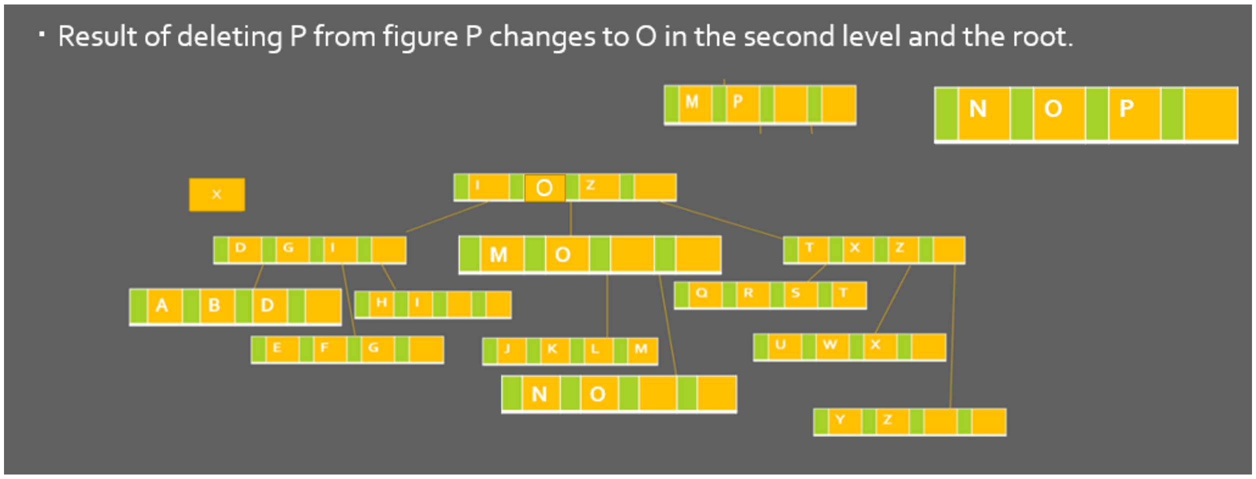
Deletion, Merging, and Redistribution

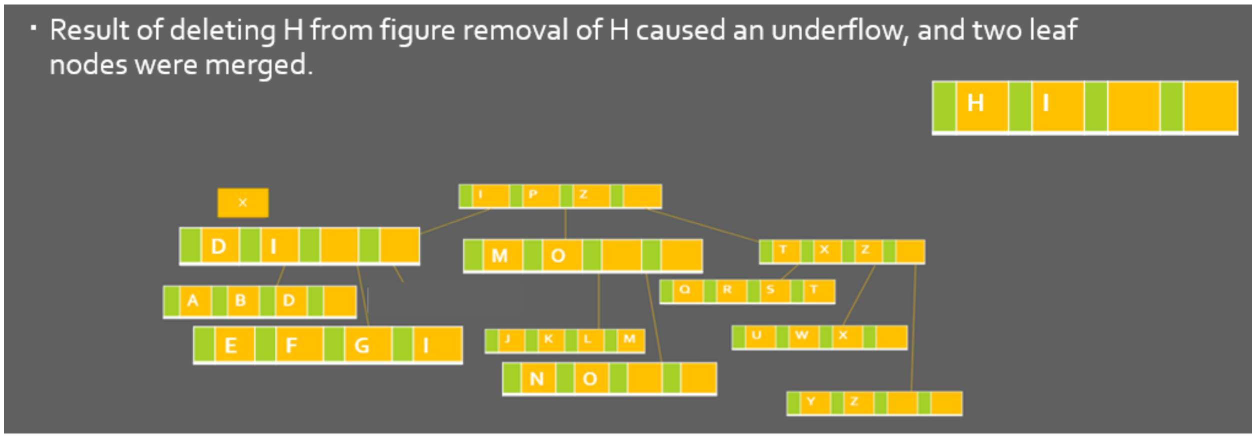
The rules that state the following:

Every page except for the root and the leaves has at least[m/2] descendants.

A page contains at least [m/2] keys and no more than m keys.


- Remove of Key C from figure change occurs only in leaf node.


- Result of deleting P from figure P changes to O in the second level and the root.



- Result of deleting H from figure removal of H caused an underflow, and two leaf nodes were merged.

The rules for deleting a key

The rules for deleting a key K from a node n :

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

Redistribution

Underflow in the leaf node:

∗ Try to redistribute (balancing) the entries from the left sibling if possible.

∗ Try to redistribute (balancing) the entries from the right sibling if possible.

∗ If the redistribution is not possible, then the three nodes are merged into two leaf nodes. In this case, underflow may propagate to internal nodes because one fewer tree pointer and search value are needed. This can propagate and reduce the tree levels.

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, the redistribution place some of the overflowing keys into another page. It is possible to quantify this efficiency of space usage by viewing the amount of space used to store information as a percentage of the total amount of space required to hold the B-tree.

B*Trees

B-trees in 1973, Knuth (1998) extends the notion of redistribution during insertion to include new rules for splitting. He calls the resulting variation on the fundamental B-tree form a B* tree.

B* Trees are variation of B-Tree. These trees have the following properties.

- Every page has a maximum of m descendants.
- Every page except for the root has at least [(2m-1)/3] descendants.
- The root has at least two descendants.
- All leaves appear on the same level.

Buffering of Pages: Virtual B-Trees

B-tree size >> main memory (in practice)

- Need buffering pages of B-tree
- Better to keep the root page in the main memory

Buffer replacement algorithm:

LRU Replacement – Least Recently Used the process of accessing the disk to bring in a page that is not already in the buffer is called a page fault, There are two causes of page faults:

1. We have never used the page.
2. It was once in the buffer but has since been replaced with a new page.

The first cause of page faults is unavoidable: if we have not yet read in and used a page, there is no way it can already be in the buffer. But the second cause is one we can try to minimize through buffer management.

Replacement based on page height – There is another, more direct way to use the hierarchical nature of the. B- tree to guide decisions about page replacement in the buffers. Our simple, keep-the-root strategy exemplifies this alternative: always retain the pages that occur at the highest levels of the tree.

Importance of virtual B-Trees – As we have emphasized, to fall into that trap is to lose sight of the original problem: to find a way to reduce the amount of memory required to handle large indexes. We did not, however, need to reduce the amount of memory to the amount required for-a single index page. It is usually possible to find enough memory to hold a number of pages. Doing so can dramatically increase system performance.

Variable-Length Records and Keys

In many applications the information associated with a key varies in length. Secondary indexes that reference inverted lists are an excellent example of this. One way to handle this variability is to place the associated information in a separate, variable-length record file; the B-tree would contain a reference to the information in this other file. Another approach is to allow a variable number of keys and records in a B-tree page. Up to this point we have regarded B-trees as being of some order m. Each page has a fixed maximum and minimum number of keys that it can legally hold. The notion of a variable-length record and, therefore, a variable number of keys per page is a significant departure from the point of view we have developed so far. A B-tree with a variable number of keys per page clearly has no single, fixed order.

Prepared by,
Ranjitha J
Dept. of ISE
AIT, Bangalore

Name of the Course: File Structure
NOTES

Indexed Sequential File Access and Prefix B^+ Tree

Indexed Sequential Access

Single organization

Indexed: seen as a set of records that is indexed by key

B-tree: for excellent indexed access

Sequential: returning records in order by key (physically contiguous records -- no seeking)

Usage of B-tree for a cosequential processing retrieve all the records in order by key B-tree file of N records, following the N pointers from the index requires N random seeks into the record file. => inefficient process. Indexed sequential access method for both interactive random access and cosequential batch processing (example- student record systems at **universities**, - it requires keyed access to individual records while also requiring a large amount of batch processing, as when grades are posted or when fees are paid during registration).

Maintaining a Sequence Set

1. The Use of Blocks
2. Choice of Block Size

First, they focus 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 if find a good way of maintaining a sequence set, then will find some way to index it as well.

1. The Use of Blocks
   The Grouping of the records into a set of sequence is called block.
   Sorting and resorting the entire sequent set as records are added and deleted very expensive process. Block the basic unit of input and output the size of buffers: can hold an entire block. Blocks for collecting the records a way to localize the changes restrict the effects of an insertion or deletion to just a part of the sequence set.

   Blocks of sequence set in order **insertion** of new records into a block overflow can be handled by a block-splitting process divide (split) the records between two blocks and

rearrange the links analogous to, but not the same that of a B-tree just divide two blocks between two blocks and rearrange the links no promotion of key.
Deletion of records from a block underflow: less than half full ,
a neighbouring node: also, half full => merge, neighbouring nodes: more than half full => redistribution analogous to, but not the same that of a B-tree.

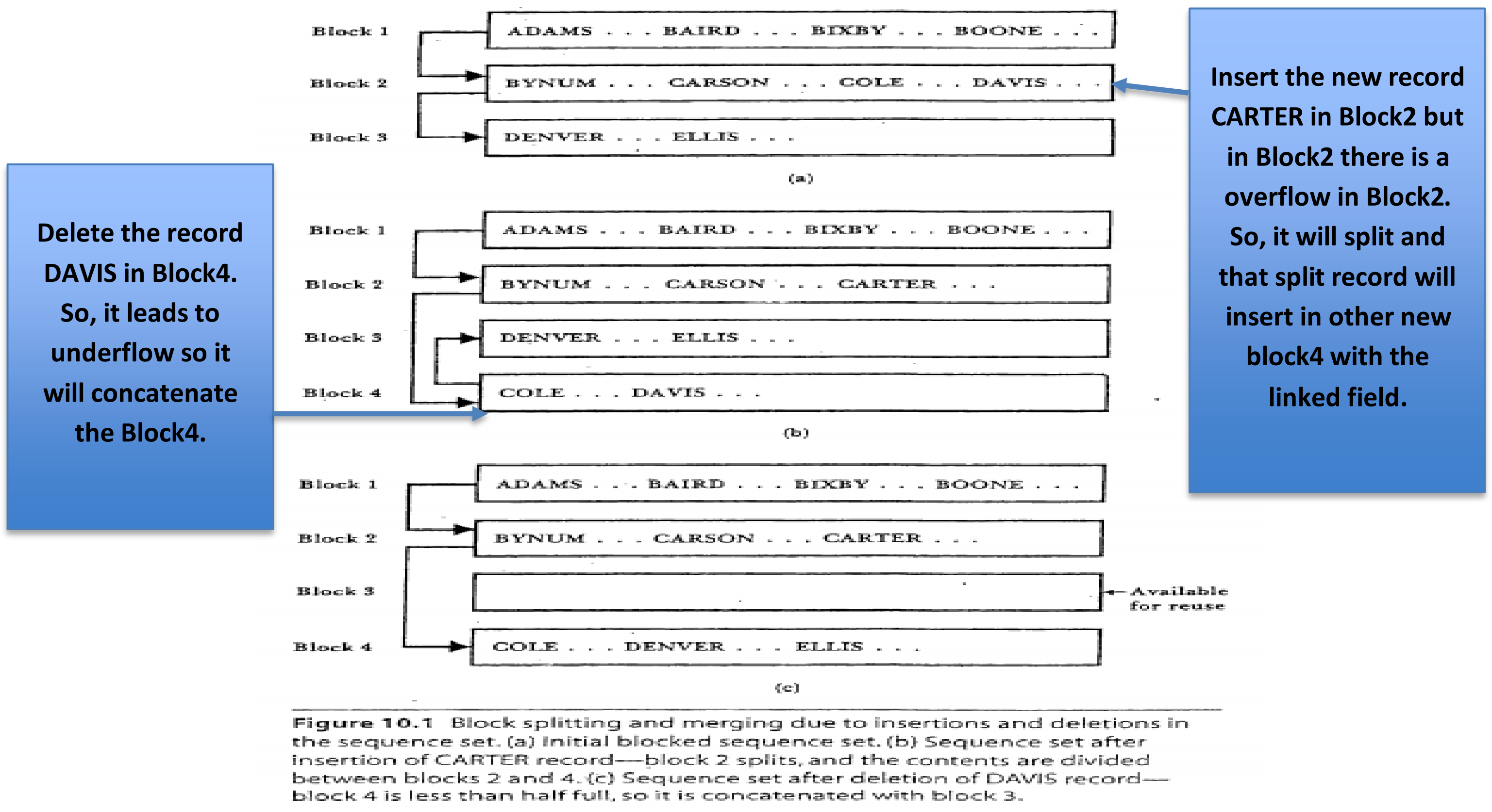

Block 1
ADAMS . . . BAIRD . . . BIXBY . . . BOONE . . .
Block 2
BYNUM . . . CARSON . . . COLE . . . DAVIS . . .
Block 3
DENVER . . . ELLIS . . .
(a)
Block 1
ADAMS . . . BAIRD . . . BIXBY . . . BOONE . . .
Block 2
BYNUM . . . CARSON . . . CARTER . . .
Block 3
DENVER . . . ELLIS . . .
Block 4
COLE . . . DAVIS . . .
(b)
Block 1
ADAMS . . . BAIRD . . . BIXBY . . . BOONE . . .
Block 2
BYNUM . . . CARSON . . . CARTER . . .
Block 3
Available for reuse
Block 4
COLE . . . DENVER . . . ELLIS . . .
(c)
Insert the new record CARTER in Block2 but in Block2 there is a overflow in Block2. So, it will split and that split record will insert in other new block4 with the linked field.
Delete the record DAVIS in Block4. So, it leads to underflow so it will concatenate the Block4.

Figure 10.1 Block splitting and merging due to insertions and deletions in the sequence set. (a) Initial blocked sequence set. (b) Sequence set after insertion of CARTER record—block 2 splits, and the contents are divided between blocks 2 and 4. (c) Sequence set after deletion of DAVIS record—block 4 is less than half full, so it is concatenated with block 3.

2. Choice of Block Size
   Consideration 1 – the block size should be such that can hold several blocks in memory at once.
   Consideration 2 – the block size should be such that can access a block without having to bear the cost of a disk seek within the block read or block write operation.
   Block the maximum guaranteed extent of physical sequentiality make each block equal to the size of a cluster (i.e., the minimum number of sectors allocated at a time).

Adding a Simple Index to the Sequence Set

Created a mechanism for maintaining a set of records so we can access them sequentially in order by key. It is based on the idea of grouping the records into blocks then maintaining the blocks, as records are added and deleted, through splitting, merging, and redistribution. Now let's see whether can find an efficient way to locate some specific block containing a particular record, given the record's key.

Index of fixed-length records contain the **key** for the **last record** in each block.

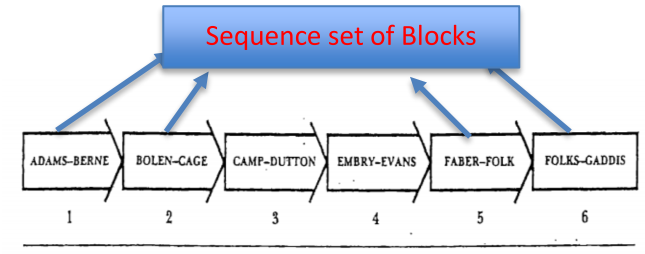
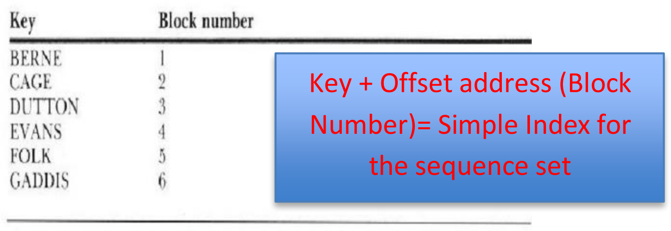


Figure 10.2 Sequence of blocks showing the range of keys in each block.


Key + Offset address (Block Number)= Simple Index for the sequence set

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:

In find of specific records by means of a binary search of the index. Binary searching works well if the searching takes place in memory, but, as saw in the B-trees, it requires too many seeks if the file is on a secondary storage device.

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.

B+-tree the use of a B-tree index for sequence set of blocks.

The Content of the Index: Separators Instead of Keys

The purpose of the index are building is to assist us when the 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. The index serves as a kind of road map for the sequence set. Given this view of the index set as a road map, can take the very important step of recognizing in the index set. Our real need is for separators. **If a string comparison between the key and any of these separators shows that the key precedes the separator.**

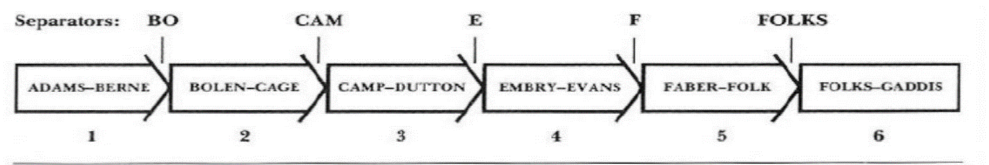
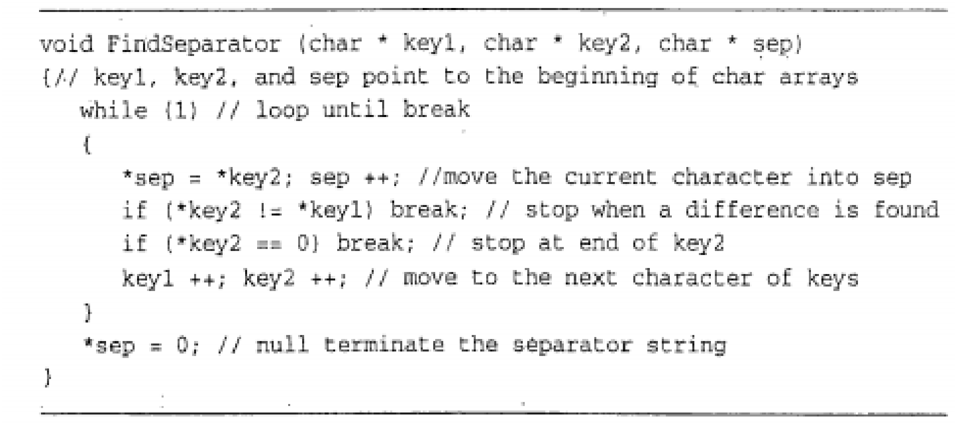


Figure 10.4 Separators between blocks in the sequence set.

Relation of search key and separator	Decision
Key < separator	Go left
Key = separator	Go right
Key > separator	Go right

Separator can be treated as variable-length entities within index structure save space by placing the shortest separator in the index structure.



```
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.

<u>The Simple Prefix B$^+$ Tree</u>

The B-tree index (index set) plus the sequence set it forms the file structure called a simple prefix B$^+$ Tree. The Simple prefix indicates that the index set contains shortest separators, or prefixes of the keys. Let’s see the Figure 10.7 and search for the record with the key EMBRY, start at the root of the index set, comparing EMBRY with the separator E, Since EMBRY comes after E, So branch to the right, retrieving the node containing the separators F and FOLKS. Since EMBRY 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.

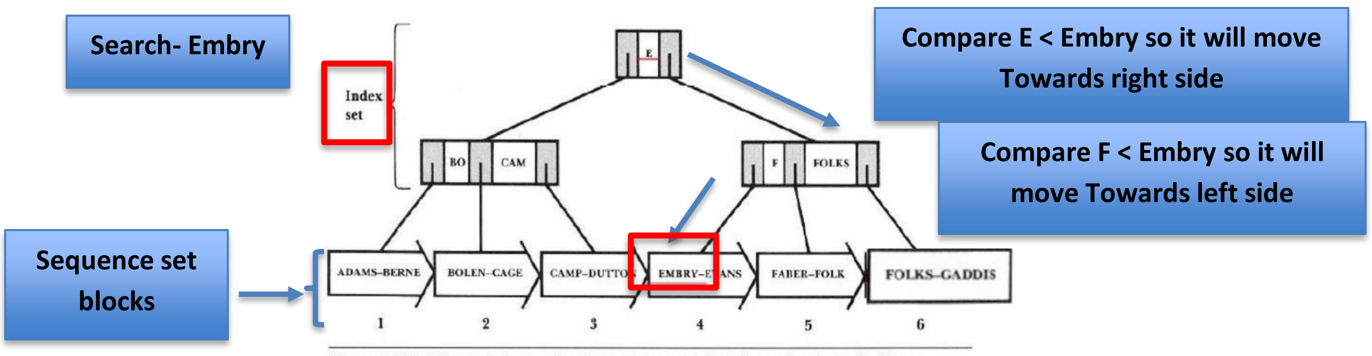


Figure 10.7 A B-tree index set for the sequence set, forming a simple prefix B+ tree.

The Simple Prefix B$^+$ Tree and its Maintenance

1. Changes Localized to **Single Blocks** in the Sequence Set
   Deletions on the sequence set delete the records for **EMBRY** and **FOLKS** limited to changes within blocks. 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.

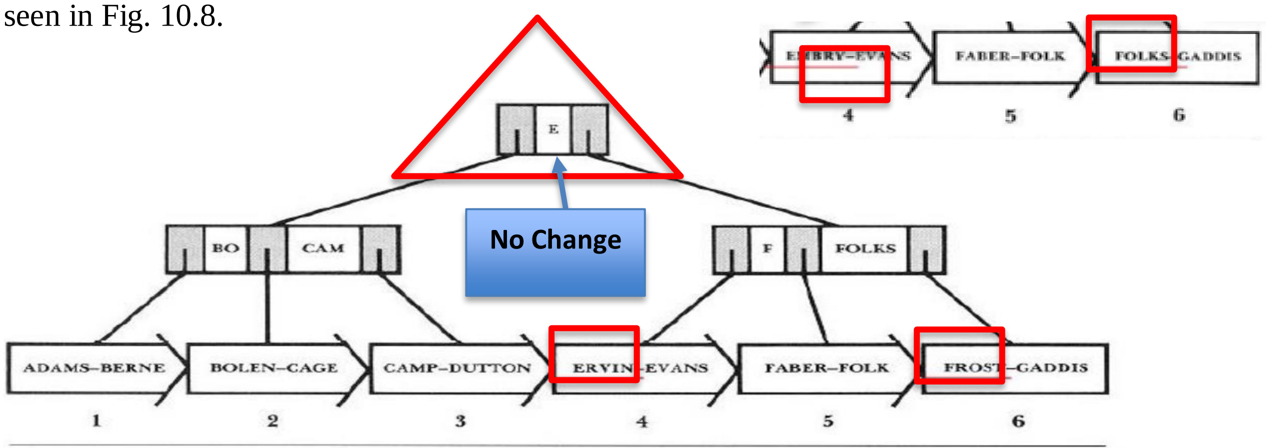


Figure 10.8 The deletion of the EMBRY and FOLKS records from the sequence set leaves the index set unchanged.

   The new record becomes the first record in block 4, but no change in the index set is necessary. This is not surprising: this decided to insert the record into block 4 on the basis of the existing information in the index set. It follows that the existing information in the index set is sufficient to allow us to find the record again.

2. Changes Involving **Multiple Blocks** in the Sequence Set
   Addition of records to the sequence set can change the number of blocks in the sequence set => need additional separators in the index set.

Deletion of records from the sequence set if we have fewer blocks, we need fewer separators Changes to the index set handled by B-tree insertion and deletion. Example of a B-tree of order three the maximum number of separators is two

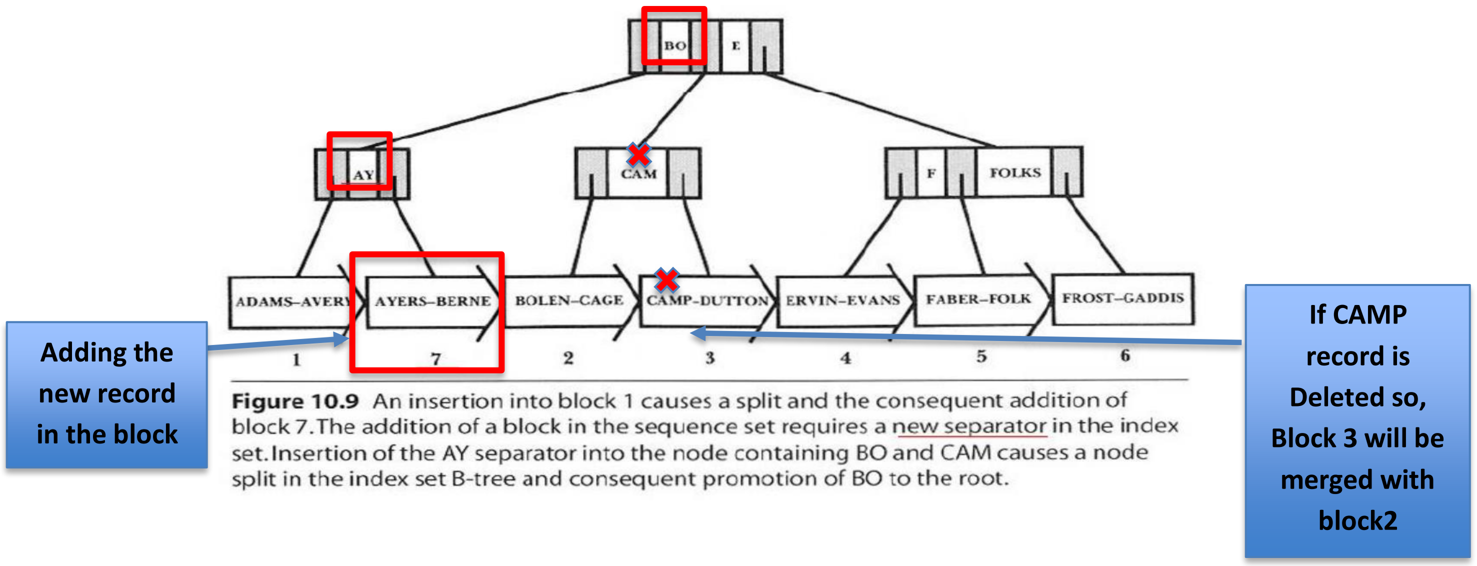

BO
E
AY
CAM
F
FOLKS
ADAMS–AVERY
AYERS–BERNE
BOLEN–CAGE
CAMP–DUTTON
ERVIN–EVANS
FABER–FOLK
FROST–GADDIS
1
7
2
3
4
5
6
Adding the new record in the block
If CAMP record is Deleted so, Block 3 will be merged with block2

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.

Example of a B-tree of order three (Cont'd) insertion into block1 in the sequence set split block 1 => block 1 and block 7 change to the sequence set addition of a block in the sequence set need a new separator, with a value of AY cause a split and promotion of BO to the root. Example of a B-tree of order three (Cont'd)

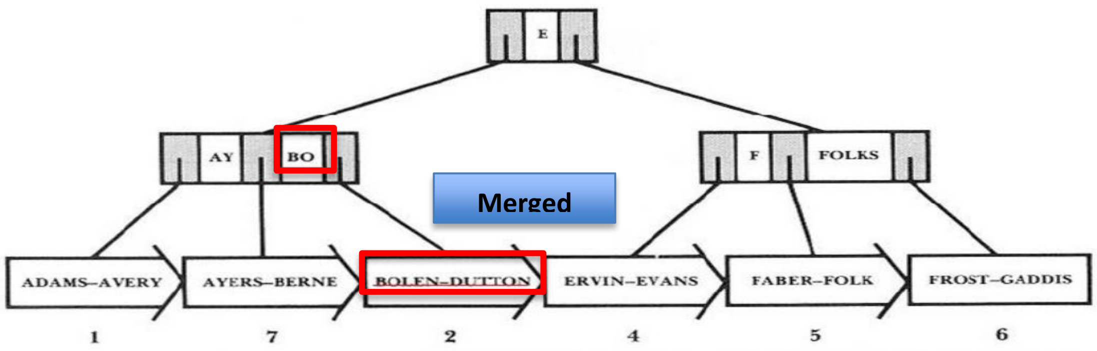


Figure 10.10 A deletion from block 2 causes underflow and the consequent merging of blocks 2 and 3. After the merging, block 3 is no longer needed and can be placed on an avail list. Consequently, the separator CAM is no longer needed. Removing CAM from its node in the index set forces a merging of index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

Example of a B-tree of order three (Cont'd) deletion of a record from block 2 of the sequence set cause an underflow condition in the sequence set need a concatenation of blocks 2 and 3 (assumption) remove block 3 remove the separator CAM that distinguish between blocks 2 and 3 cause an underflow in the index set node => merging need a concatenation of index nodes bring BO back down from the root. Block split in the sequence set results in a node split in the index set. Block merge(concatenation) in the sequence set results in a node merge(concatenation) in the index set.

Insertions and deletions in the index set are handled as B-tree operations Record insertion and deletion always take place in the sequence set

Index Set Block Size

The size and structure of an index node for the index set is usually the same as the physical size of a block in the sequence set. There is a reason for using a common block size for the index and sequence sets:

Common block size for the index and sequence sets the **block size that is best** for the sequence set is usually best for the index set a **common block** size makes it easier to implement a **buffering scheme to create a virtual simple prefix B^+ tree**. The index set blocks and sequence set block are often mingled with in the same file to avoid seeking between two separate files while accessing the simple prefix B^+. Use of one file for both kinds of blocks is simpler if the block sizes are the same.

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

Separators are chosen instead of keys so that can hold more separators in a single node but if the order of B-tree is fixed then the number of separators stored inside a node would also be fixed. The main motivation for using separators would not be effectively used. Inorder to use it efficiently need to store more separator for which the order of the B-tree should be variable.
Large, fixed-size block for the index set for a variable # of variable-length separators a variable-order B-tree how to search through variable-length separators a block can hold a large number of separators once we read a block into memory, binary search rather than sequential search is needed.

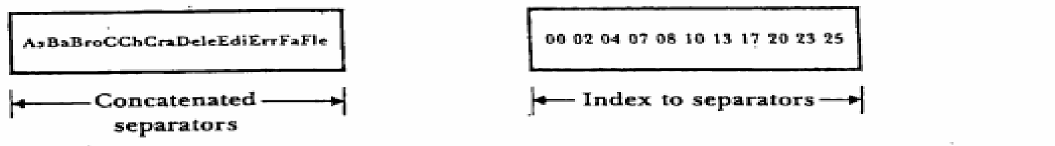


Figure 10.11 Variable-length separators and corresponding index.

The binary search is applied on the block, corresponding key is fetched and corresponding block containing the key is fetched. To fetch the block, need the block number. Thus make the node as variable order then each node should contain **separator count, total length of separator, relative block numbers**.

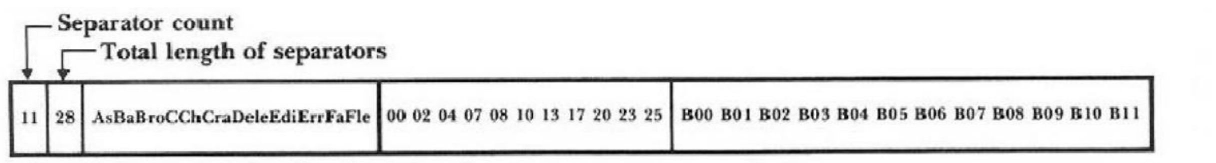


Figure 10.12 Structure of an index set block.

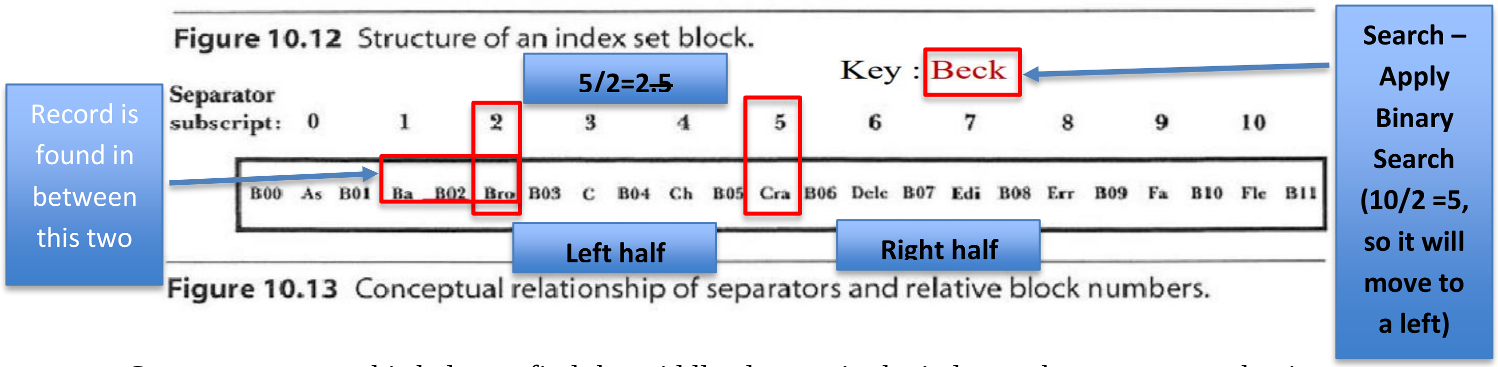

Record is found in between this two
5/2=2.5
Key : Beck
Search – Apply Binary Search (10/2 =5, so it will move to a left)
Separator subscript: 0 1 2 3 4 5 6 7 8 9 10
B00 As B01 Ba B02 Bro B03 C B04 Ch B05 Cra B06 Dele B07 Edi B08 Err B09 Fa B10 Fle B11
Left half
Right half

Figure 10.13 Conceptual relationship of separators and relative block numbers.

Separator count – this helps us find the middle element in the index to the separator so begin our binary search.

Total length of separators – the index block would begin after the separator block so need to know the offset at which the index would begin.

Relative Block Number – this is used to fetch the block containing the record.

Loading a Simple Prefix B^+ Trees

Split or redistribution of blocks in the sequence set and, in the index, set in order to maintain B+ tree splitting and redistribution are relatively expensive involve searching down through the tree for each insertion then reorganization the tree.

When we are loading the B+ tree, begin by sorting the records that are to be loaded place the records into sequence set blocks, one by one, starting a new block make the transition between two sequence set blocks determine the shortest separator for the blocks collect separators into an index set block.

Four sequence set blocks and one index set block

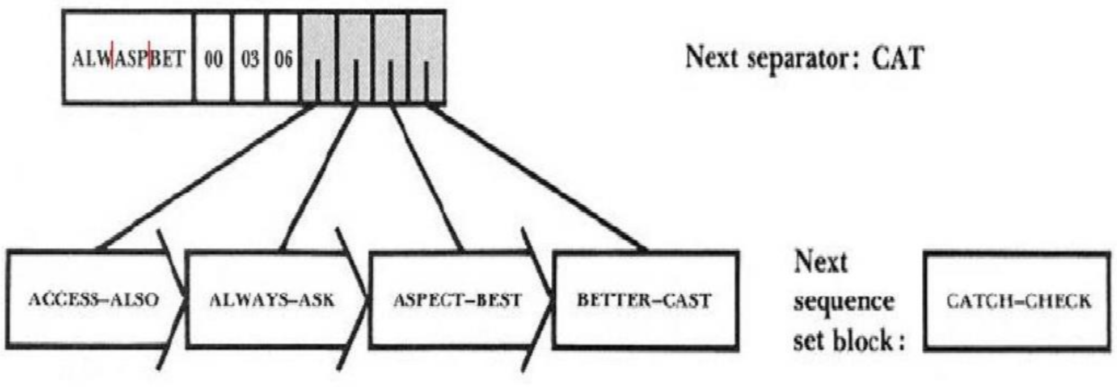


Figure 10.14 Formation of the first index set block as the sequence set is loaded.

Building of two index set levels

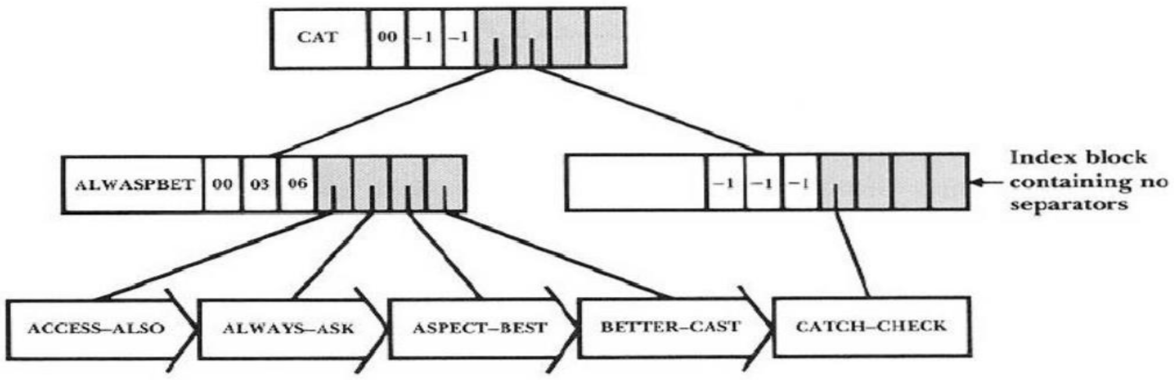


Figure 10.15 Simultaneous building of two index set levels as the sequence set continues to grow.

Growth of index set built up from the sequence set.

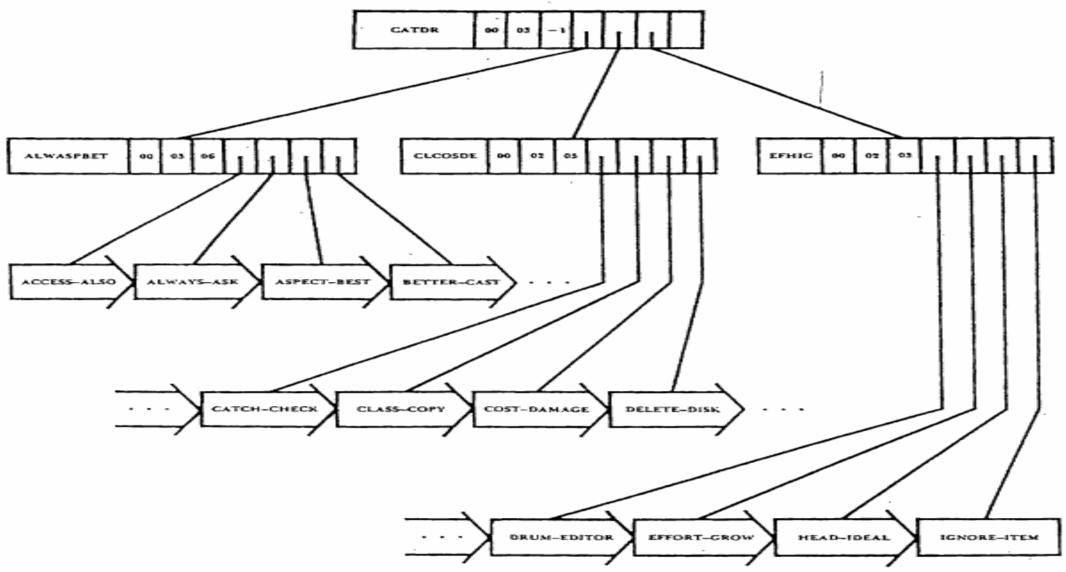


Figure 10.16 Continued growth of index set built up from the sequence set.

<u>B^+ Trees</u>

B+ tree does not involve the use of prefixes as separators separators in the index set are copies of the actual keys Fig. 10.17: initial loading steps for a B+ tree

Difference between a B+ tree and a simple prefix B+ tree distinguish between the role of the separators in the index set and keys in the sequence set difficult to make distinction when the separators are exact copies of the keys. B+ tree without the use of shortest separators.

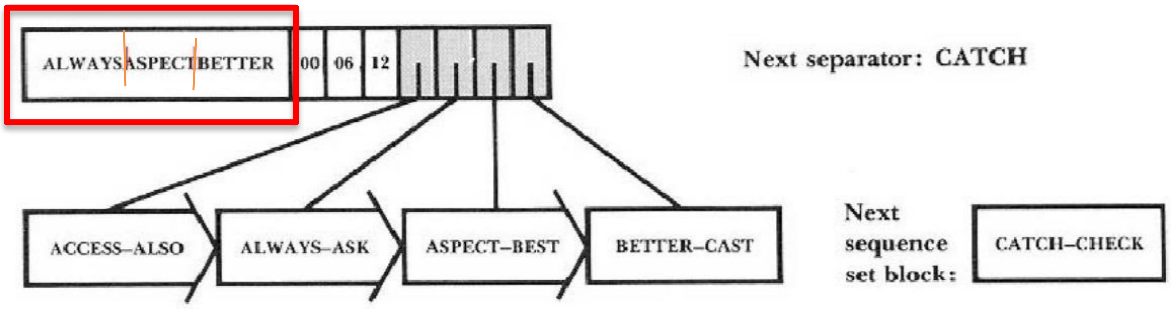


Figure 10.17 Formation of the first index set block in a B^+ tree without the use of shortest separators.

<u>B-Tree, B$^+$ Tree and Simple Prefix B$^+$ Trees in Perspective</u>

B-Trees

Information can be found at any level of the B-trees can take up less space than does a B+ trees because there is no need for additional storage to hold separators sequential access obtained through an in-order traversal of the tree not workable because of all the seeking required to retrieve the actual record information if the B-trees merely contains pointers to records.

B+ Trees

Separation of the index set and the sequence set separators: copies of the keys advantages of the B+ trees over B-trees sequence set can be processed in a truly linear, sequential way, providing efficient access to records in order by key B+ tree approach can often result in a shallower tree than would B-tree approach.

Simple Prefix B+ Trees

separation of the index set and the sequence set (same as the B+ trees) sequence set can be processed in a truly linear, sequential way (the same as the B+ trees) separators : smaller than the actual keys allow us to build a shallower tree than B+ trees separator compression and consequent increase in branching factor -> use an index set block structure that support variable length fields

Prepared by,
Ranjitha J
Assistant Professor
Dept. of ISE
Atria Institute of Technology

Name of the Course: File Structure
NOTES

Hashing, Extendible Hashing

Introduction

Sequential access efficiency is O (N). B trees/ B+ trees efficiency is O(logK 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).

What is Hash?

Hash function is a function h(k) that transforms a key into an address.
a = h(K)
h (hash function), K (key), a (home address)
Example
K=BASS
h = (first char * second char) mod 1000
a = h(K) = (66 * 65) mod 1000 = 4,290 mod 1000 = 290

Collision – two different keys may be sent to the same address generating a collision.
Example
key-LOWELL=>a=(76*79)mod1000=6,004mod1000=4
OLIVIER => a = (79 * 76) mod 1000 = 6,004 mod 1000 = 4
Several ways to reduce the number of collisions

1. Spread out the records
   Collision occurs 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.
2. Use extra memory
   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 need to be distributed among 100 address space. Whereas chances of collision are more when 10 records need to be distributed among 10 address space. The obvious disadvantage is wastage of space.

3. 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.

A Simple Hashing Algorithm

3 Steps

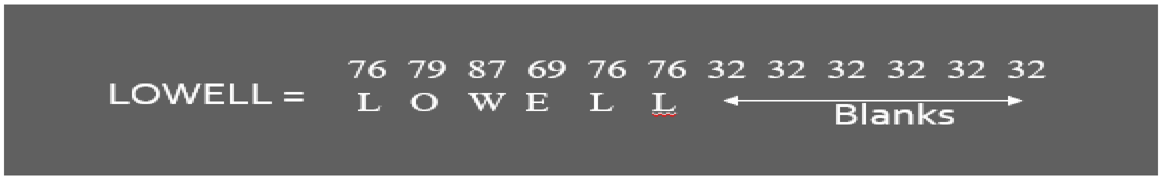
1. Represent the key in numerical form

2. Fold and add

3. Divide by a prime number and use the remainder as the address

Example

Step 1. Represent the Key in Numerical Form



Step 2. Fold and Add

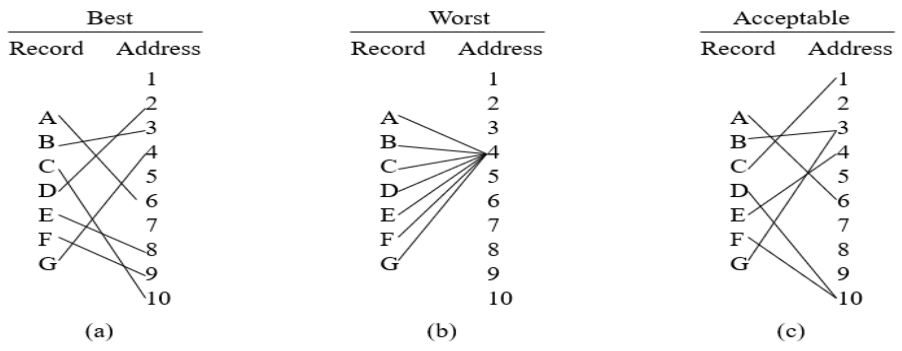
76 79 | 87 69 | 76 76 | 32 32 | 32 32 | 32 32
7679 + 8769 + 7676 + 3232 + 3232 = 30588
(30588+3232 = 33820)

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

Step 3. Divide by the Size of the Address Space
a = s mod n (n: # of address in file)
a = 13883 mod 100 = 83
a = 13883 mod 101 = 46

Hashing and Record Distribution

1. Distributing Records among Addresses
   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)

2. 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). Converting this number to base 11 system results in (382)11. Then 382 mod 99 = 85. So, 85 is the hash address.

3. Predicting the distribution of records

The Poisson Distribution

When single key is hashed, there are two possible outcomes with respect to the given address:

A—The address is not chosen; or

B—The address is chosen.

If there are 10 addresses (N=10)

$$p(B) = b = \frac{1}{N}$$

b=1/10

=0.1

$$p(A) = a = \frac{N-1}{N} = 1 - \frac{1}{N}$$

a= 1-0.1

=0.9

If suppose two keys are hashed. So, the two applications of the hashing functions are independent of one another, the probability that both will produce the given address is a product:

$$p(BB) = b \times b = \frac{1}{N} \times \frac{1}{N} \quad \text{for } N = 10 : b \;\; b = 0.1 \times 0.1 = 0.01$$

Even other outcomes are also possible when two keys are hashed. For example, the second key could hash to an address other than the given address. The probability of this is the product.

$$p(BA) = b \times a = \frac{1}{N} \times \left(1 - \frac{1}{N}\right) \quad \text{for } N = 10 : b \times a = 0.1 \times 0.9 = 0.09$$

P(BABBA) = 0.81*0.001 = 0.00081

In general, when want to know the probability of a certain sequence of outcomes, such as BABBA, can replace each A & B

Outcome	Probability	For $N = 10$
$BBAA$	$bbaa = b^2a^2$	$(0.1)^2(0.9)^2 = 0.0036$
$BABA$	$baba = b^2a^2$	$(0.1)^2(0.9)^2 = 0.0036$
$BAAB$	$baab = b^2a^2$	$(0.1)^2(0.9)^2 = 0.0036$
$ABBA$	$abba = b^2a^2$	$(0.1)^2(0.9)^2 = 0.0036$
$ABAB$	$abab = b^2a^2$	$(0.1)^2(0.9)^2 = 0.0036$
$AABB$	$aabb = b^2a^2$	$(0.1)^2(0.9)^2 = 0.0036$

$$p(BBAA) + p(BABA) + \ldots + p(AABB) = 6b^2a^2 = 6 \times 0.0036 = 0.0216.$$

The 6 in the expression 6b2a2 represents the number of ways two Bs and two As can be distributed among four places. So, that “r trials result in r-x As and x Bs”. The probability of each such way is

$$a^{r-x}b^x$$

and the number of such ways is given by the formula

$$C = \frac{r!}{(r-x)!\,x!}$$

This is the well-known formula for the number of ways of selecting x items out of a set of r items. It follows that when r keys are hashed, the probability that an address will be chosen x

$$p(x) = Ca^{r-x}b^x$$

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

$$p(0) = C\left(1 - \frac{1}{N}\right)^{r-0}\left(\frac{1}{N}\right)^0$$

$$p(1) = C\left(1 - \frac{1}{N}\right)^{r-1}\left(\frac{1}{N}\right)^1$$

This expression has the disadvantage that it is awkward to compute.

(Try it for 1000 addresses and 1000 records: N=r=1000.) Fortunately, for large values of N and r, there is a function that is a very good approximation for p(x) and is much easier to compute. It is called the poisson function.

The poisson Function applied to hashing, p(x) is given by

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

N = the number of available addresses;
r = the number of records to be stored; and
x = the number of records assigned to a given address,

Here 1000 addresses (N=1000) and 1000 records whose keys are hashed to the addresses (r=1000).

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

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

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

In general, if there are N addresses, then the expected number of addresses with x records assigned to them is Np(x).

4. Predicting collisions for a full file
Suppose you have a hashing function that you will believe will distribute records randomly and you want to store 10000 records in 10000 addresses. How many addresses do you expect to have no records assigned to them?
Since r= 10 000 and N= 10 000, r/N = 1. Hence the proportion of addresses with 0 records assigned should be

$$p(0) = \frac{1^0 e^{-1}}{0!} \doteq 0.3679$$

The number of addresses with no records assigned is

$$10\,000 \times p(0) = 3679$$

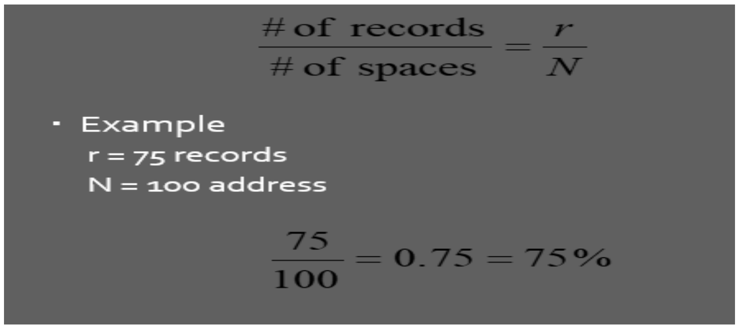
How many addresses should have one, two, and three records assigned, respectively?

$$10\,000 \times p(1) = 0.3679 \times 10\,000 = 3679$$
$$10\,000 \times p(2) = 0.1839 \times 10\,000 = 1839$$
$$10\,000 \times p(3) = 0.0613 \times 10\,000 = 613$$

Since the 3679 addresses corresponding to x = 1 have exactly one record assigned to them, their records have no synonyms. The 1839 addresses with two records apiece, however, represent potential trouble. If each such address has space only for one record, and two records are assigned to them, there is a collision, this means that 1839 records: will fit into the addresses, but another 1839 will not fit. There will be 1839 overflow records. This is a bad situation that thousands of records that do not fit into the addresses assigned by the hashing function. need to develop a method for handling these overflows records. But first, let's try to reduce the number of overflow records.

How much Extra Memory should be used?

1. Packing Density



$$\frac{\#\text{ of records}}{\#\text{ of spaces}} = \frac{r}{N}$$

- Example
  r = 75 records
  N = 100 address

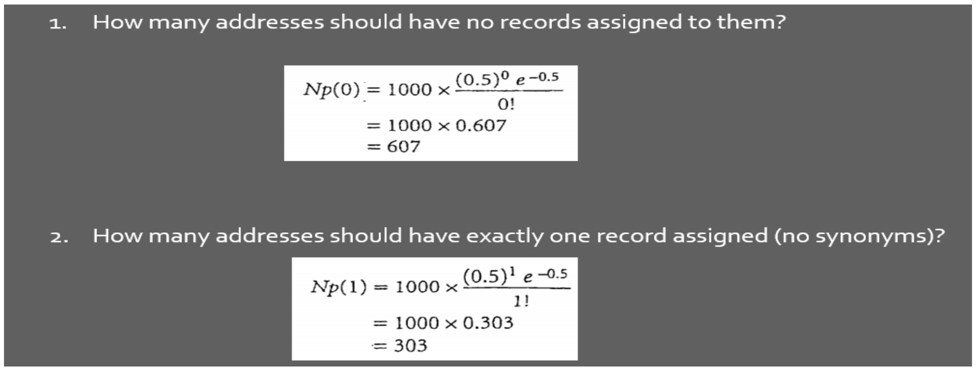
$$\frac{75}{100} = 0.75 = 75\%$$

2. Predicting Collisions for Different Packing Densities

The formula for packing density (r/N) occurs twice in the poisson formula

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

Here the same behavior is exhibited by 500 records distributed among 1000 addresses as by 500000 records distributed among 1000000 addresses.
Suppose that 1000 addresses are allocated to hold 500 records in a randomly hashed file, and that each address can hold one record. The packing density for the file is r/N = 500/1000 = 0.5.
Lets us answer the following questions about the distribution of records among the available addresses in the file:

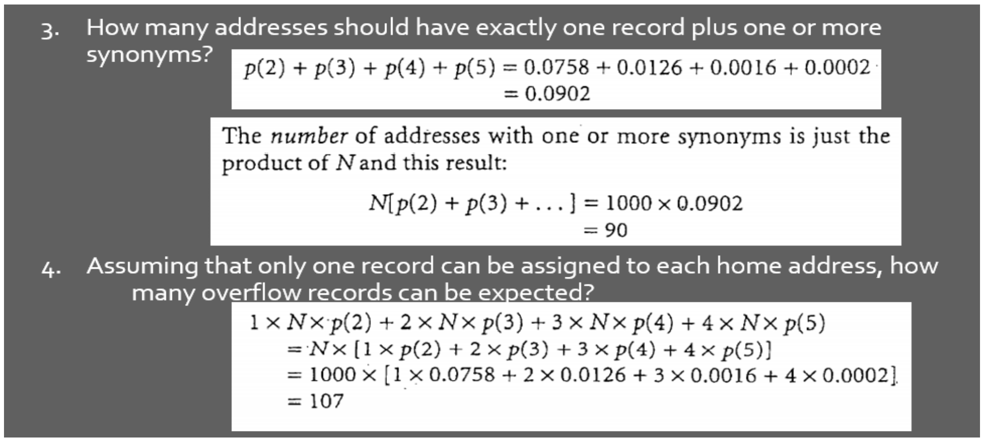


1. How many addresses should have no records assigned to them?

$$\begin{aligned} Np(0) &= 1000 \times \frac{(0.5)^0\, e^{-0.5}}{0!} \\ &= 1000 \times 0.607 \\ &= 607 \end{aligned}$$

2. How many addresses should have exactly one record assigned (no synonyms)?

$$\begin{aligned} Np(1) &= 1000 \times \frac{(0.5)^1\, e^{-0.5}}{1!} \\ &= 1000 \times 0.303 \\ &= 303 \end{aligned}$$



3. How many addresses should have exactly one record plus one or more synonyms?

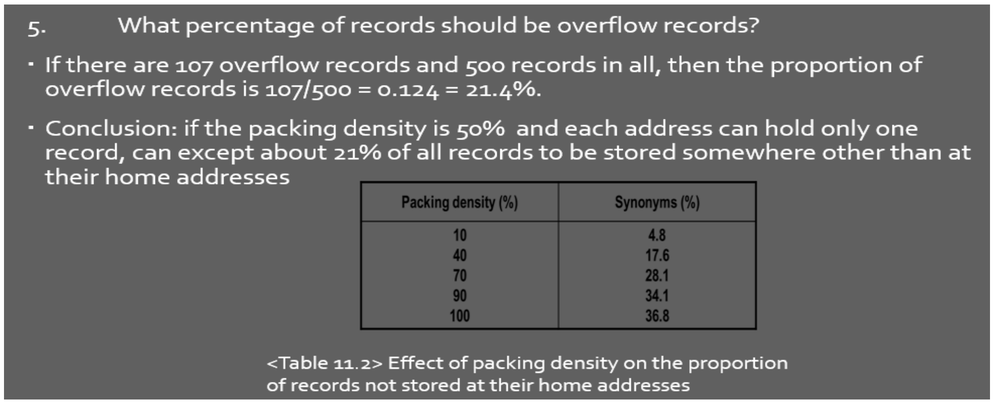
$$\begin{aligned} p(2) + p(3) + p(4) + p(5) &= 0.0758 + 0.0126 + 0.0016 + 0.0002 \\ &= 0.0902 \end{aligned}$$

The *number* of addresses with one or more synonyms is just the product of N and this result:

$$\begin{aligned} N[p(2) + p(3) + \ldots] &= 1000 \times 0.0902 \\ &= 90 \end{aligned}$$

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

$$\begin{aligned} & 1 \times N \times p(2) + 2 \times N \times p(3) + 3 \times N \times p(4) + 4 \times N \times p(5) \\ &= N \times [1 \times p(2) + 2 \times p(3) + 3 \times p(4) + 4 \times p(5)] \\ &= 1000 \times [1 \times 0.0758 + 2 \times 0.0126 + 3 \times 0.0016 + 4 \times 0.0002] \\ &= 107 \end{aligned}$$



5. What percentage of records should be overflow records?

- If there are 107 overflow records and 500 records in all, then the proportion of overflow records is 107/500 = 0.124 = 21.4%.
- Conclusion: if the packing density is 50% and each address can hold only one record, can except about 21% of all records to be stored somewhere other than at their home addresses

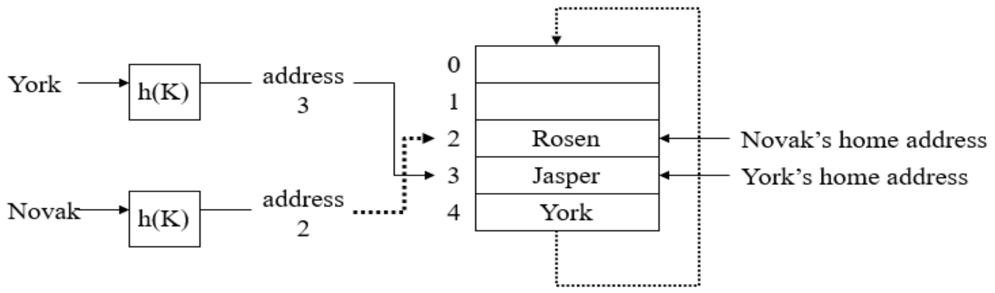
Packing density (%)	Synonyms (%)
10	4.8
40	17.6
70	28.1
90	34.1
100	36.8

<Table 11.2> Effect of packing density on the proportion of records not stored at their home addresses

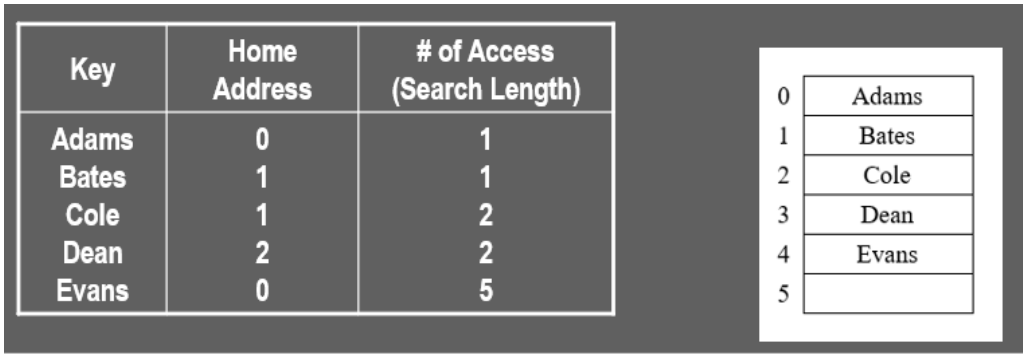
Collision resolution by progressive overflow

1. Progressive Overflow / linear probing works as follows:
   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.

   Example



2. Search Length



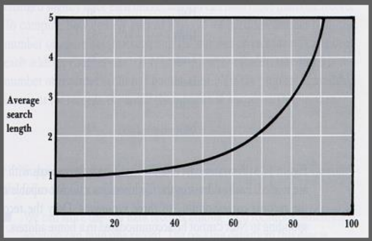
Key	Home Address	# of Access (Search Length)
Adams	0	1
Bates	1	1
Cole	1	2
Dean	2	2
Evans	0	5

0	Adams
1	Bates
2	Cole
3	Dean
4	Evans
5	

$$\text{Average Search Length} = \frac{\text{total search length}}{\text{total number of records}}$$

Example

$$\text{Average Search Length} = \frac{1+1+2+2+5}{5} = 2.2$$



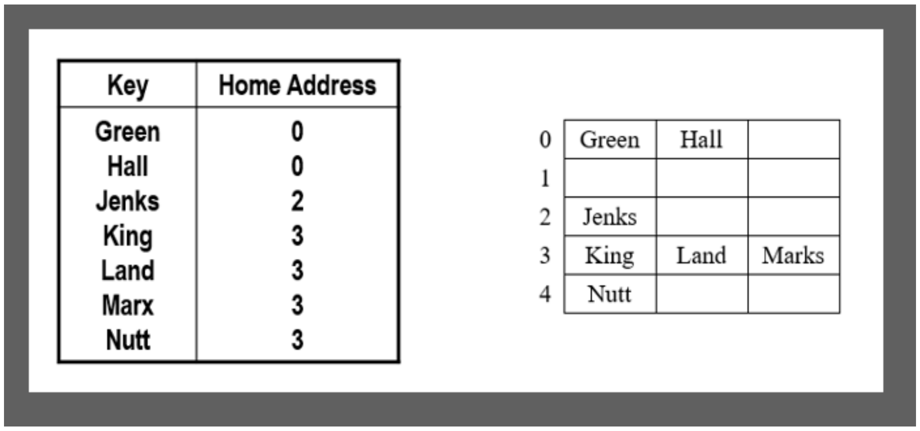
<Figure 11.7> Average search length versus packing density in a hashed file

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.



Key	Home Address
Green	0
Hall	0
Jenks	2
King	3
Land	3
Marx	3
Nutt	3

0	Green	Hall	
1			
2	Jenks		
3	King	Land	Marks
4	Nutt		

1. Effects of Buckets on Performance
   We will compare the following two alternatives

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

   1. Storing 750 data records into a hashed file with 1000 addresses, each holding 1 record.

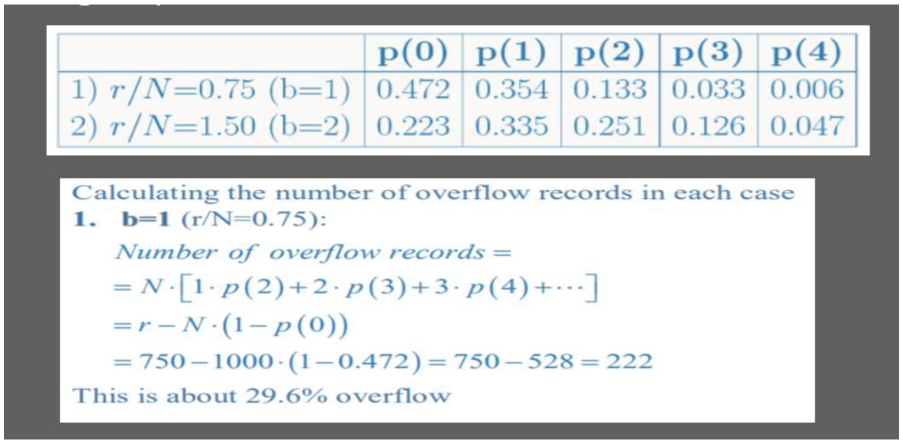
2. Storing 750 data records into a hashed file with 500 bucket addresses, each bucket holding 2 records

► In both cases the packing density is 0.75 or 75%.

► In the first case r/N=0.75.

► In the second case r/N=1.50

Estimating the probabilities as defined before:



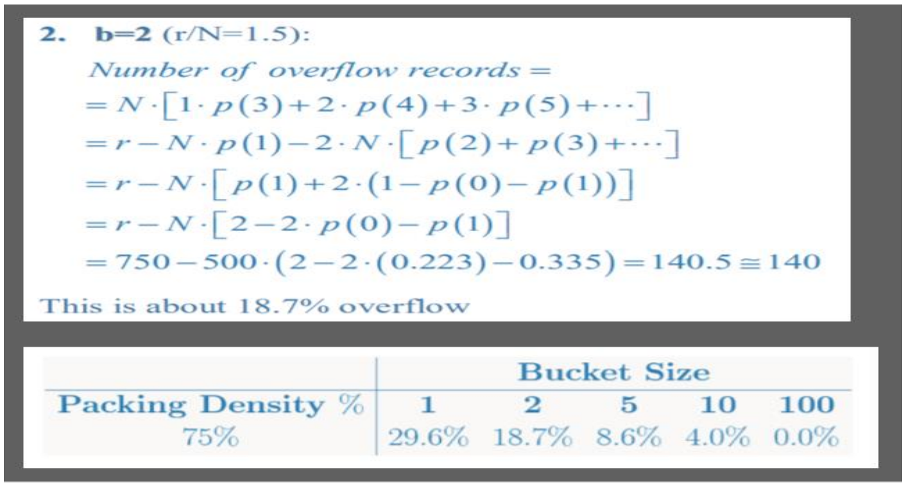
	p(0)	p(1)	p(2)	p(3)	p(4)
1) r/N=0.75 (b=1)	0.472	0.354	0.133	0.033	0.006
2) r/N=1.50 (b=2)	0.223	0.335	0.251	0.126	0.047

Calculating the number of overflow records in each case

1. b=1 (r/N=0.75):

$$\begin{aligned}&\textit{Number of overflow records} =\\&= N\cdot[1\cdot p(2)+2\cdot p(3)+3\cdot p(4)+\cdots]\\&= r-N\cdot(1-p(0))\\&= 750-1000\cdot(1-0.472)=750-528=222\end{aligned}$$

This is about 29.6% overflow



2. b=2 (r/N=1.5):

$$\begin{aligned}&\textit{Number of overflow records} =\\&= N\cdot[1\cdot p(3)+2\cdot p(4)+3\cdot p(5)+\cdots]\\&= r-N\cdot p(1)-2\cdot N\cdot[p(2)+p(3)+\cdots]\\&= r-N\cdot[p(1)+2\cdot(1-p(0)-p(1))]\\&= r-N\cdot[2-2\cdot p(0)-p(1)]\\&= 750-500\cdot(2-2\cdot(0.223)-0.335)=140.5\cong 140\end{aligned}$$

This is about 18.7% overflow

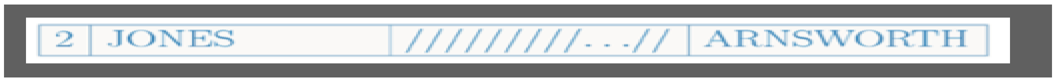
	Bucket Size				
Packing Density %	**1**	**2**	**5**	**10**	**100**
75%	29.6%	18.7%	8.6%	4.0%	0.0%

2. Implementation Issues

1. Bucket Structure

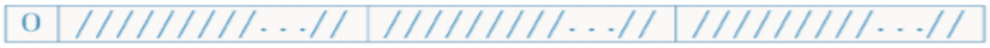
A Bucket should contain a counter that keeps track of the number of records stored in it.

- Empty slots in a bucket may be marked '//.../'
- Example: Bucket of size 3 holding 2 records



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

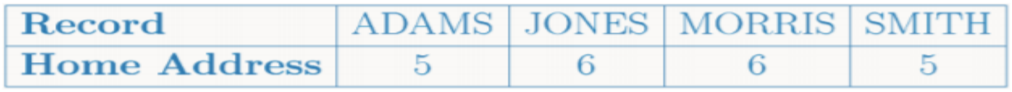
2. Initializing a file for hashing
    - Decide on the Logical Size (number of available addresses) and on the number of buckets per address.
    - Create a file of empty buckets before storing records. An empty bucket will look like



3. Loading a hash file
    - When inserting a key, remember to:
    - Be careful with infinite loops when hash file is full

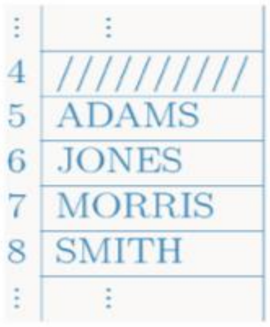
Making Deletions

Deletions in a hashed file have to be made with care



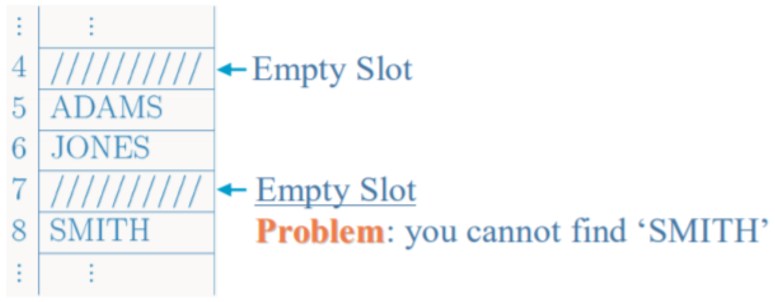
Record	ADAMS	JONES	MORRIS	SMITH
Home Address	5	6	6	5

Hashed File using Progressive Overflow



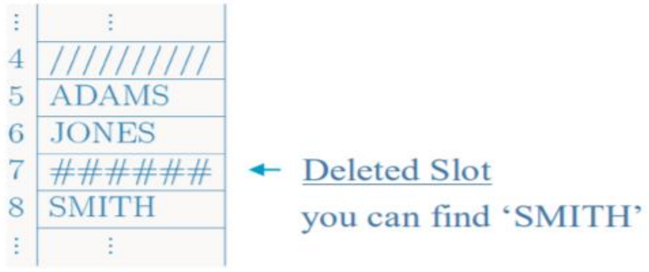
⋮	⋮
4	//////////
5	ADAMS
6	JONES
7	MORRIS
8	SMITH
⋮	⋮

1. **Tombstones** for handling deletions
   Making Deletions: Delete ‘MORRIS’ Making Deletions: Delete ‘MORRIS’
   If ‘MORRIS’ is simply erased, a search for ‘SMITH’ would be unsuccessful



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

Solution- Replace deleted records with a marker indicating that a record once lived there



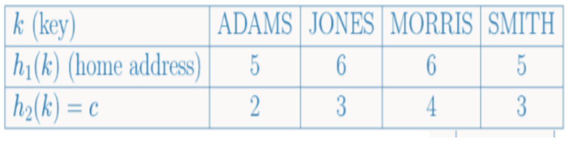
A search must continue when it finds a tombstone, but can stop whenever an empty slot is found.

2. Implications of Tombstones for Insertion
   Only insert a tombstone when the next record is occupied or is a tombstone.
   Insertions should be modified to work with tombstones: if either an empty slot or a tombstone is reached, place the new record there.
3. Effects of Deletions and Additions on Performance
   The presence of too many tombstones increases search length.
   Solutions to the problem of deteriorating average search lengths:
   Deletion algorithm may try to move records that follow a tombstone backwards towards its home address. Complete reorganization: re-hashing Use a different type of collision resolution technique.

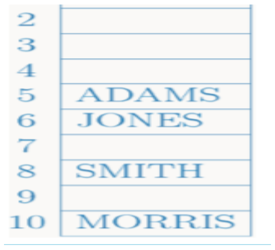
Other Collision Resolution Techniques

1. Double Hashing
   The first hash function determines the home address
   - If the home address is occupied, apply a second hash function to get a number c (c relatively prime to N)

- c is added to the home address to produce an overflow addresses: if occupied, proceed by adding c to the overflow address, until an empty spot is found.



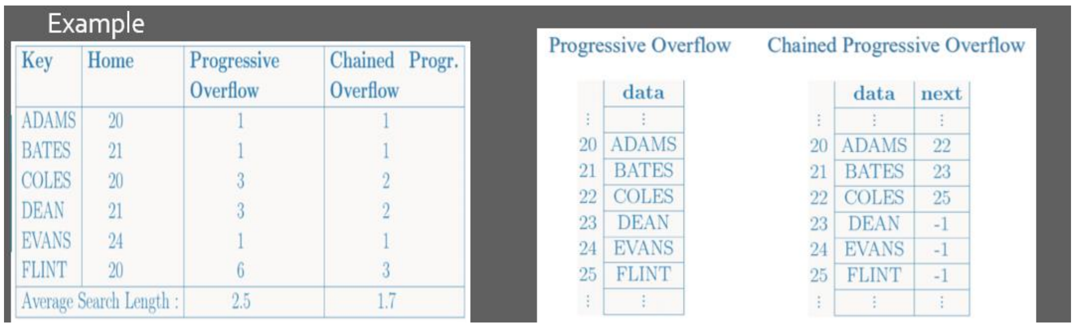
k (key)	ADAMS	JONES	MORRIS	SMITH
$h_1(k)$ (home address)	5	6	6	5
$h_2(k) = c$	2	3	4	3



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

Hashed file using double hashing.

2. Chained Progressive Overflow
   Similar to progressive overflow, except that synonym are linked together with pointers. The objective is to reduce the search length for records within clusters.



Example

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

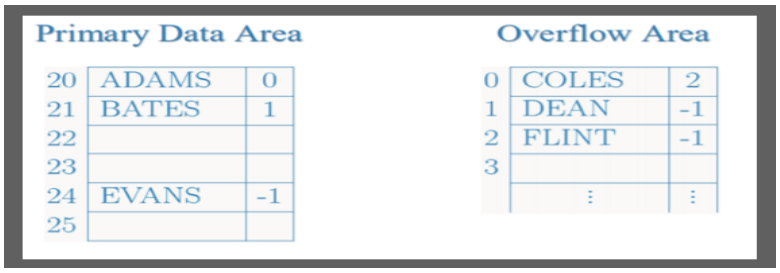
Progressive Overflow

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

Chained Progressive Overflow

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

3. Chained with a Separate Overflow Area
   Move overflow records to a Separate Overflow Area
   A linked list of synonyms starts at their home address in the Primary data area, continuing in the separate overflow area.
   When the packing density is higher than 1 an overflow area is required.



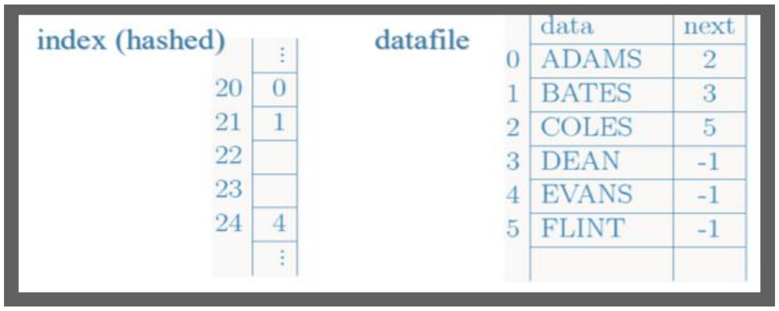
Primary Data Area

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

Overflow Area

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

4. Scatter Tables: Indexing Revisited
   Similar to chaining with separate overflow, but the hashed file contains no records, but only pointers to data records.



Patterns of Record Access

The information about what records get accessed most often, can optimize their location so that these records will have short search lengths.

By doing this, try to decrease the effective average search length even if the nominal average search length remains the same.

A small percentage of the records in a file account for a large percentage of the accesses :

80 / 20 Rule

80% of the accesses are performed on 20% of the records.

Extendible Hashing

How Extendible Hashing Works

1. Tries
   The key idea behind extendible hashing is to combine conventional hashing with another retrieval approach called the tries. 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 want to build a tries that stores the keys able, abrahms, Adams, anderson, andrews, and Baird. A schematic form of the tries 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.

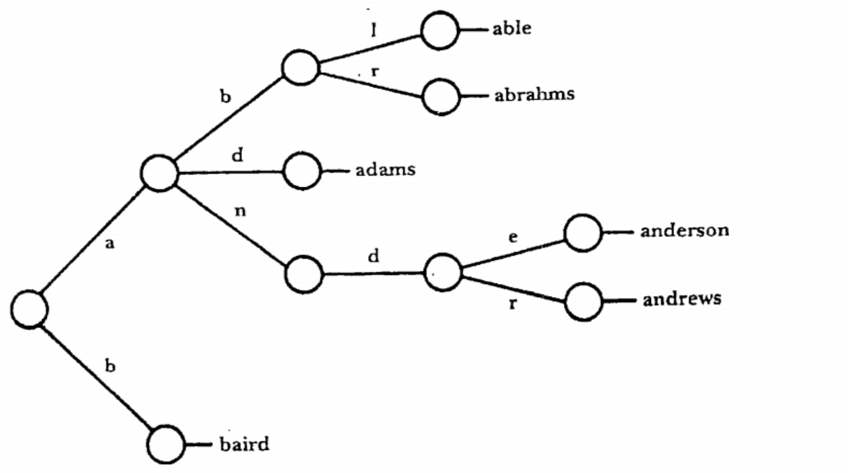


Figure 12.1 Radix 26 trie that indexes names according to the letters of the alphabet.

   Notice that in searching a tries 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.

2. Turning the Tries into a directory
   The 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 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.

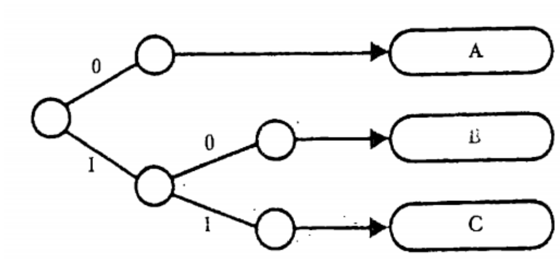


Figure 12.3 Radix 2 trie that provides an index to buckets.

How should we represent the tries?
Rather than representing the tries as a tree, 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.
Step 2: Once extended the tree this way, can collapse it into the directory structure shown in Fig. 12.4(b). Now have a Structure that provides the kind of direct access associated with hashing.

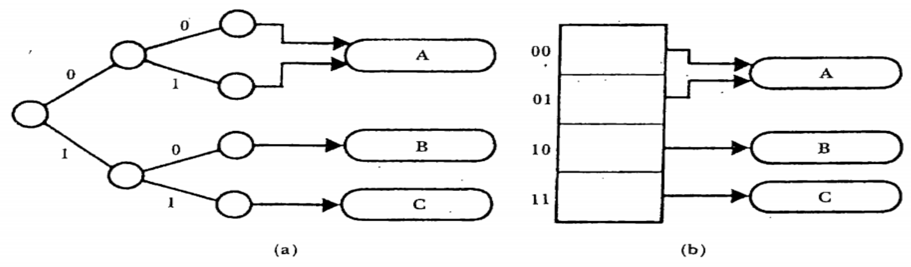


Figure 12.4 The trie from Fig. 12.3 transformed first into a complete binary tree, then flattened into a directory to the buckets.

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

So, use the full 2 bits to divide the addresses between two buckets. Don't need to extend the address space; simply make full use of the address information that already have. The figure shows the directory and buckets after the split.

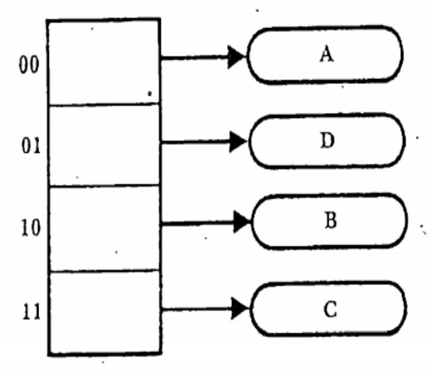


Figure 12.5 The directory from Fig. 12.4(b) after bucket A overflows.

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 to split bucket B and where to attach the new bucket after the split? Unlike our previous example, do not have additional, unused bits of address space that can press into duty as split the bucket now need to use 3 bits of the hash address in order to divide up the records that hash to bucket B. The tries illustrated in Fig. 12.6

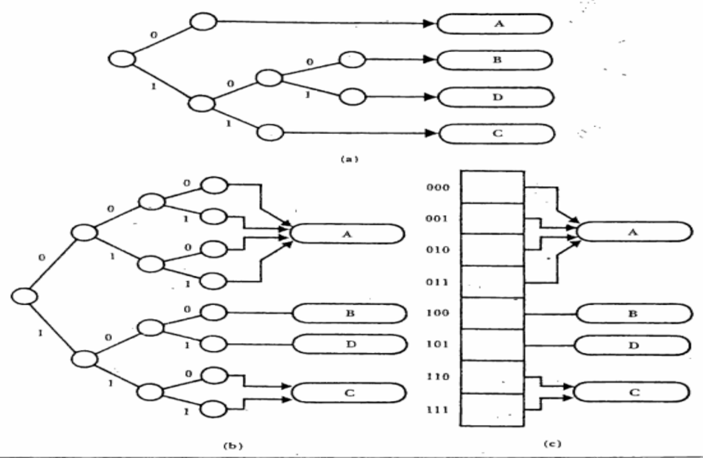


Figure 12.6 The results of an overflow of bucket B in Fig. 12.4(b), represented first as a trie, then as a complete binary tree, and finally as a directory.

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

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.

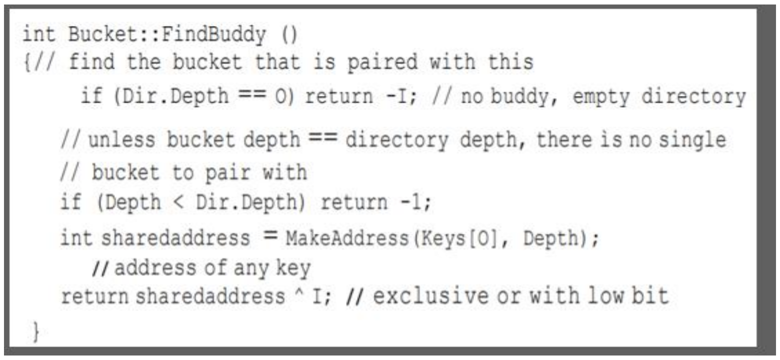
```
int MakeAddress (char * key, int depth)
{
   int retval = 0;
   int hashVal = Hash(key);
   // reverse the bits
   for (int j = 0; j < depth; j++)
   {
      retval = retval << 1;
      int lowbit = hashVal & 1;
      retval = retval | lowbit;
      hashVal = hashVal >> 1;
   }
   return retval;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

Deletion

1. 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.

Reducing the size of the address space restores the directory and bucket structure to the arrangement, before the additions and splits that produced the structure. 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 use when need to add new directory cells.

2. A Procedure for Finding Buddy Buckets
   The method works by checking to see 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.



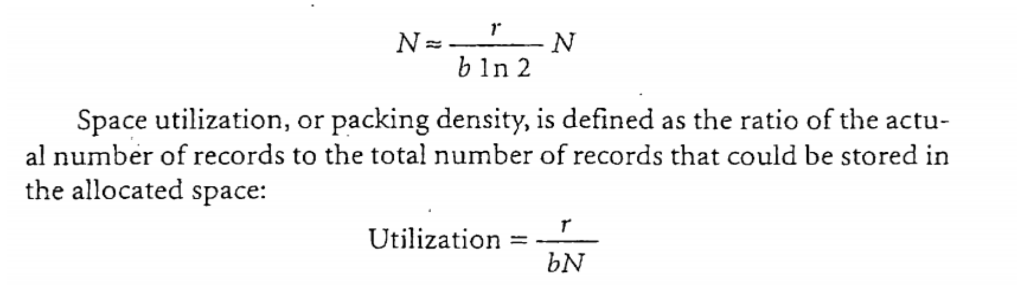
```
int Bucket::FindBuddy ()
{// find the bucket that is paired with this
     if (Dir.Depth == 0) return -I; // no buddy, empty directory
   // unless bucket depth == directory depth, there is no single
   // bucket to pair with
   if (Depth < Dir.Depth) return -1;
   int sharedaddress = MakeAddress(Keys[0], Depth);
      //address of any key
   return sharedaddress ^ I; // exclusive or with low bit
 }
```

3. 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 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 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.

<u>Extendible Hashing Performance</u>

1. Space Utilization for buckets

   Fagin, Nievergelt, Pippenger, and Strong include analysis and simulation of extendible hashing performance. Both the analysis and simulation show that the space utilization is strongly periodic, fluctuating between values of 0.53 and 0.94. The analysis portion of their paper suggests that for a given number of records r and a block size of b, the average number of blocks N is approximated by the formula

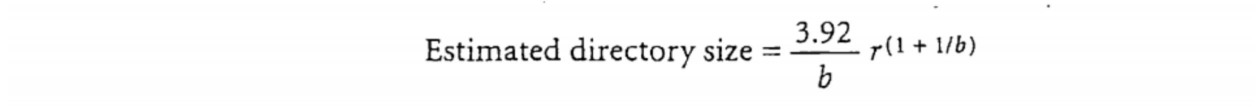


$$N \approx \frac{r}{b \ln 2} N$$

Space utilization, or packing density, is defined as the ratio of the actual number of records to the total number of records that could be stored in the allocated space:

$$\text{Utilization} = \frac{r}{bN}$$

2. Space Utilization for the directory
   Flajolet also provides the following formula for making rough estimates directory size



$$\text{Estimated directory size} = \frac{3.92}{b} r^{(1 + 1/b)}$$

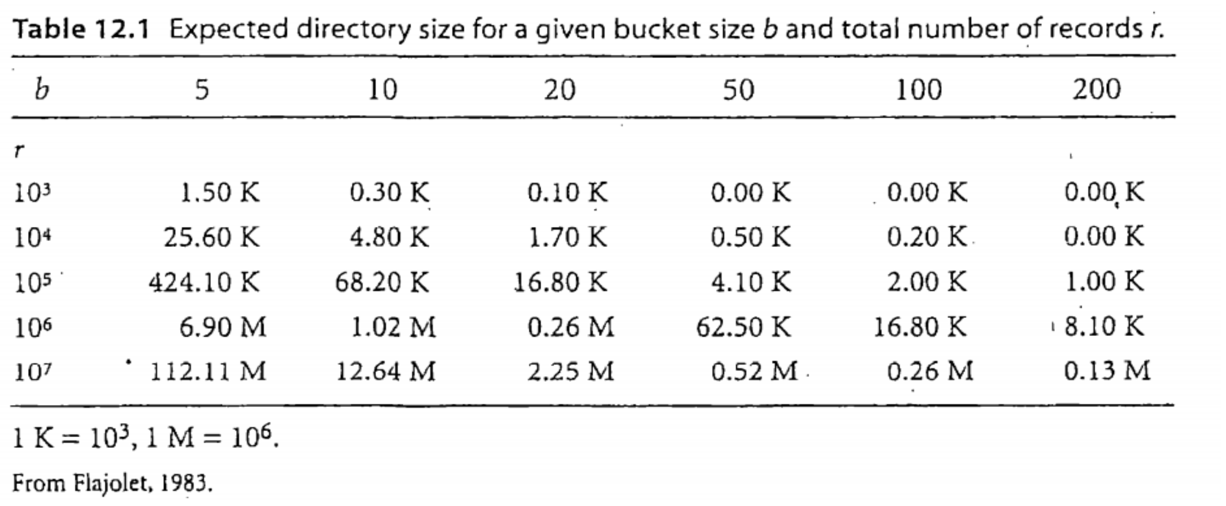


Table 12.1 Expected directory size for a given bucket size *b* and total number of records *r*.

b	5	10	20	50	100	200
r						
10^3	1.50 K	0.30 K	0.10 K	0.00 K	0.00 K	0.00 K
10^4	25.60 K	4.80 K	1.70 K	0.50 K	0.20 K	0.00 K
10^5	424.10 K	68.20 K	16.80 K	4.10 K	2.00 K	1.00 K
10^6	6.90 M	1.02 M	0.26 M	62.50 K	16.80 K	8.10 K
10^7	112.11 M	12.64 M	2.25 M	0.52 M	0.26 M	0.13 M

1 K = 10^3, 1 M = 10^6.

From Flajolet, 1983.

Alternative Approaches

1. Dynamic Hashing
   Shows an initial address space of four and four buckets descending from the four addresses in the directory.
   split the bucket at address 4. We address the two buckets resulting from the split as 40 and 41 . So, change the shape of the directory node at address 4 from a square to a circle because it has changed from an external node

split the bucket addressed by node 2, creating the new external nodes 20 and 21 .So, also split the bucket addressed by 41 , extending the tries 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.

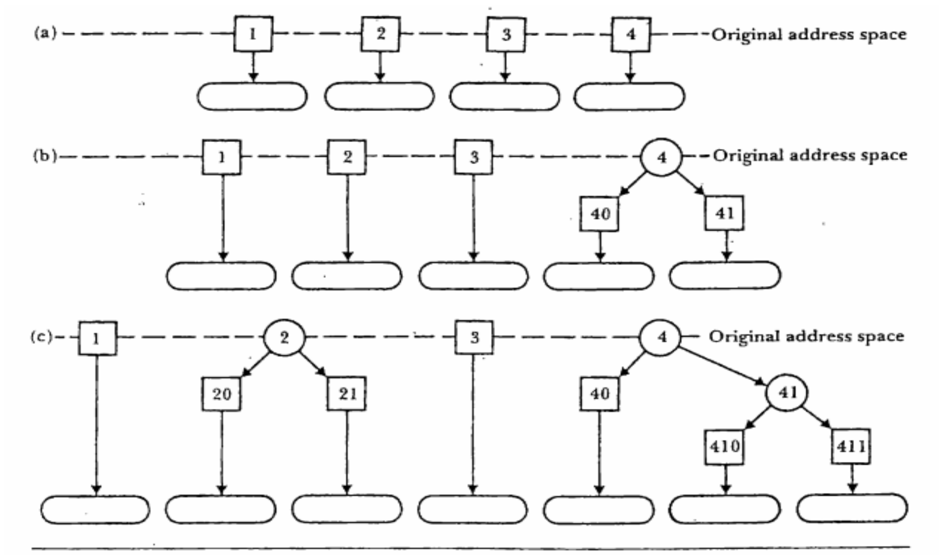


Figure 12.23 The growth of index in dynamic hashing.

2. Linear Hashing
   Linear hashing, like extendible hashing, uses more bits of hashed value as the address space grows.

(a) Note that the address space consists of four buckets rather than four directory nodes that can point to buckets.

(b) add records, bucket b overflows. The overflow forces a split. However, as 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 extension, which 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 split, the overflowing record is placed into an overflow bucket w.

(c) 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.

At this point all of the buckets use the h3(k) address function, and have finished the expansion cycle.

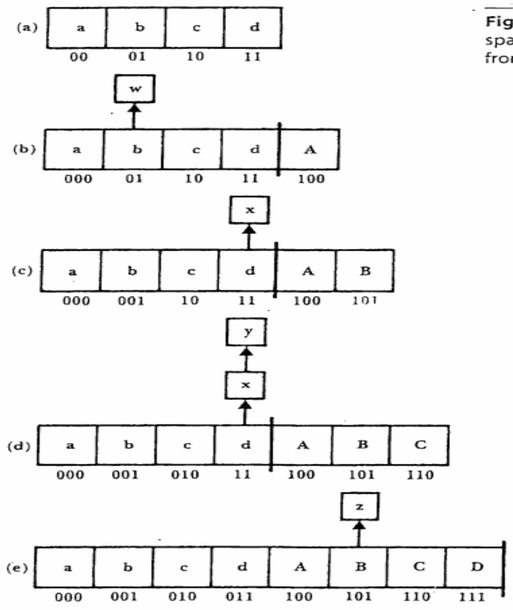


Figure 12.24 The growth of address space in linear hashing. Adapted from Enbody and Du (1988).

3. Approaches to controlling splitting

   Postpone splitting for extensible hashing

   1. Use chaining overflow bucket
   2. Avoid doubling directory space
   3. 1.1 seek, 76% ~ 81% storage utilization

Ranjitha J
Assistant Professor
Dept. of ISE
Atria Institute of Technology Bangalore