Page No.:
Date: / 201

ASSIGNMENT - 1

1 Define File structure. Briefly discuss the evolution of file structure. [10 m]

A File structure is a combination of
i) representations for data in files, and of
ii) Operations for accessing the data
A File structure allows applications to read, write and modify data. It might also support finding the data that matches some search critiria or reading through the data in some particular order.

Evolution of file structure:

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

ii. Emergence of disks and indexes: [pdf]
* As indexes also have a sequential flavour, when they grew too much, they also became difficult to manage.
* Idea of using tree structure to manage the index emerged in early 1960'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.

iii) Emergence of Tree Structure:
* Sequential access was not a good solution for large files.
* Disks allowed for Direct access.
* Indexes made it possible to keep a list of keys and pointers in a small file that could be searched very quickly.
* With the key & pointer, the user has direct access to large primary file.

iv) Balanced Trees:
* In 1963, researchers came up with the idea of AVL trees for data in memory. However, they are not applied to files because they work well only when tree nodes are composed of single records.
* In 1970's, came the idea of B-Trees which require an $O(\log_k N)$ access time where,
  $N \rightarrow$ no. of entries in the file
  $k \rightarrow$ no. of entries indexed in a single block of the B-Tree structure.
* B-Tree can guarantee that one can find one file entry among millions of others with only 3-4 trips to the disk.

v) Hash tables:
* From early on, hashing was a good way to reach the goal of accessing data with a single request with files that do not change size greatly over time, but do not work well with volatile, dynamic files.
* Extendible, dynamic hashing reaches this goal.

2. Discuss about the fundamental file processing operations. [10 marks]

File processing operations:

i. Physical files and Logical files:
Physical files: A file that actually exists on secondary storage. It is the file as known by the computer OS and that appear in its File directory.
(Or) A collection of bytes stored on a disk or tape.

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

ii) Opening files:
* Unix system function open() [API] is used to open an existing file or create a new file.

* Many C++ implementation supports this function.

Syntax:

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

where,

fd → File descriptor. Type: integer (int)
It is the logical file name
If there is an error in the attempt to open the file, this value is -ve

filename → physical filename. Type: char *
This argument can be a pathname.

flags → Controls the operation of the open function. Type: int

The value of flag is set by performing a bitwise OR of the following values.

O_APPEND: Append every write operation to the end of the file.
O_CREATE: Create and open a file for writing. It has no effect if file already exists.
O_EXCL: return an error if O_CREATE is specified and the file exists.
O_RDONLY: Open a file for reading only
O_WRONLY: Open a file for writing only.
O_RDWR: Open a file for read & write
O_TRUNC: if a file exists, truncate it to a length of zero, destroying its contents.

pmode → required if O_CREATE is specified.
TYPE: int. It specifies the protection mode for the file.

In unix, pmode is a three digital octal number that indicates how the file can be accessed by user, group & others.

Eg: pmode → 751 = 111 101 001
owner group others.

Examples:

i) fd = fopen (filename, O_RDWR | O_CREAT, 075);
ii) fd = open (filename, O_RDWR | O_CREAT | O_TRUNC, 075);
Creates new files for reading and writing. If it already exists, its contents are truncated.

iii) Closing files:

* Making the logical filename available for another physical file.
* Ensures that everything has been written to the file.
* Files are usually closed by the OS when a program terminates normally.
* close() is a function or system call that breaks the link b/w a logical and corresponding physical filename.

iv. Reading and writing function:

* These actions make file processing an I/O operation.

Read function: It is an function or system call used to obtain input from a file or device.

Syntax: Read (source_file, dist_addr, size);

where,

source_file → location, the program reads from
dest_addr → First address of memory block where we want to store the data.
size → how much information is being brought in from the file <Byte count>

Write function: It is a function or system call used to provide output capabilities.

* Syntax: Write (dest_file, source_addr, size);
where,
dest_file → logical filename where the data will be written
source_addr → First address of memory block where the data to be written is stored
size → Number of bytes to be written.

V. Seeking:
Seek is a function or system call that sets the read/write pointer to a specified position in the file.
Syntax: seek (source_file, offset);
where
source_file → logical filename in which the seek will occur
offset → number of positions in the file the pointer is to be moved from the start of the file.
Eg: seek (data, 373);

VI. Special characters in files:
* When DOS files are opened in text mode, the internal separator ('\n') is translated to the external separator (<CR><LF>) during read and write.
* while in binary mode, internal separator is not translated to external separator during read and write.

VII. The UNIX Directory structure:
In UNIX, the directory structure is a single tree for the entire file system.
Separate disks appear as subdirectories of the root (/)
Eg: /usr/bin/pgm

VIII. Physical devices and logical files:
Physical devices as files: In unix, devices like keyboard and console are also files.
The console, keyboard and std. error:
* Stdout → console
* Stdin → Keyboard
* Stderr → standard error.

I/O redirection and pipes:
* > filename [redirect stdout to "filename"]
* < filename [redirect stdin to "filename"]
pgm1 | pgm2
take any stdout output from pgm1 and use it as stdin input on to pgm2
Eg: list | sort

IX. File-Related header files:
Header files might vary,
Stdio.h, iostream.h, fstream.h, fcntl.h and file.h are some of header files.

X. UNIX file system.

cat filename	Type contents of a file
tail filename	Type last ten lines of file
cp f1 f2	Copy f1 to f2
mv f1 f2	move f1 to f2.
rm filename	Delete file
chmode mode f1	change protection mode
ls	List contents of a directory
mkdir	Create directory
rmdir	Remove directory.

3 What are various major strengths and weakness of CD-ROM [10 Marks]

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

Data transfer rate: Not terribly slow nor very fast It has modest transfer rate of 75 sectors or 150 kilo bytes per second. Its about 5 times faster than transfer rate of floppy discs and an order of magnitude slower than rate for good Winchester disks.

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

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

a) We never have to worry about updating.

b) This simplifies the file structure and also optimizes our index structures and other aspects of file organization.

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

4 Differentiate the following:

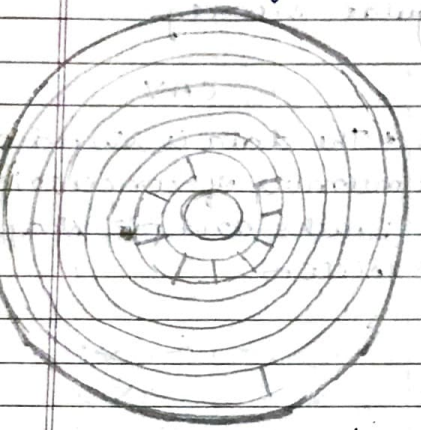
i) CLV and CAV

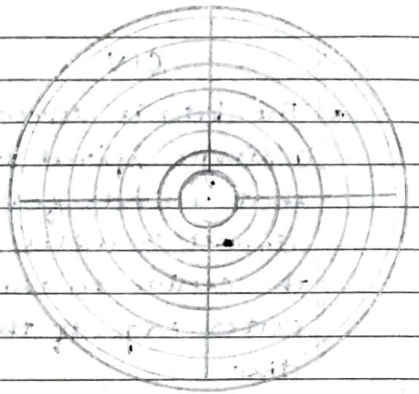
CLV: constant linear velocity

CAV: constant angular velocity.

CLV	CAV
* The data is stored on a single spiral track that winds for almost 3 miles from the centre to the outer edge of the disc	* The data is stored on in number of concentric tracks and pie shaped sectors.
* All the sectors take same amount of space.	* Inner sectors take small amount of space compared to outer sectors
* Storage capacity of all sectors is same.	* Storage capacity of all sectors is same.
* All sectors are written at maximum density. (Constant data density)	* Writes data less densely in outer sectors and more densely at inner sectors (Variable data density)
* Due to constant data density space is not wasted in either inner or outer sectors.	* Due to variable data density space is wasted in outer sectors.

* Constant data density implies that disc has to spin more slowly when reading data at outer sectors compared to reading at the inner sectors	* Variable data density implies that disc rotate at constant speed irrespective of reading from inner sectors or outer sectors.
* Poor seeking performance	* Seeking is fast compared to CLV.





ii) Physical and logical files

Physical files	Logical files.
* It occupies the portion of memory. It contains the original data.	* It does not occupy memory space. It does not contain data
* A physical file contains one record format.	* It can contain upto 32 record formats.
* It can exist without logical file.	* It cannot exist without physical file.
* If there is a logical file for physical file, the physical file cannot be deleted until and unless the logical file is deleted.	* If there is a logical file for a physical file, the logical file can be deleted without deleting the physical file.
* CRTPF command is used to make such object	CRTLF command is used to make such object.

5. Briefly explain the different basic ways to organize the data on disk. [10 marks]

* The information stored on a disk is stored on the surface of one or more platters. This arrangment is such that the information is stored in successive tracks on the surface of the disk.

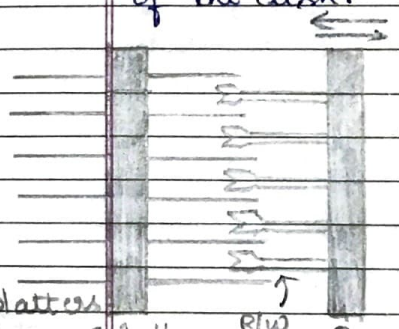
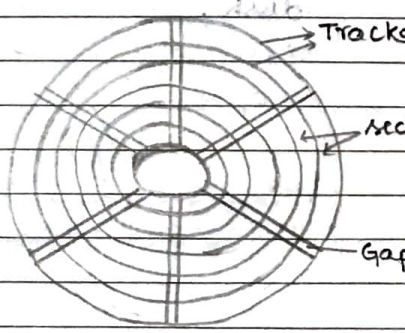


Illustration of disk drive

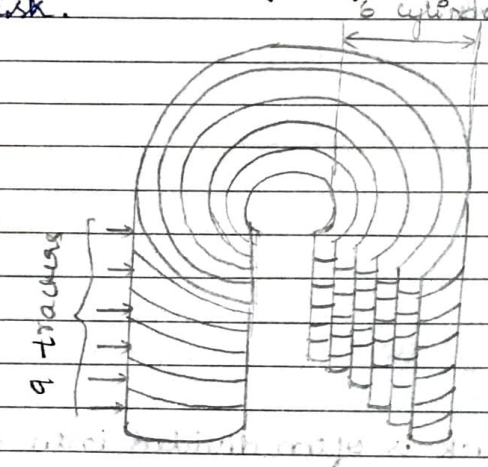


Surface of disk showing tracks and sectors

* Each track is often divided into number of sectors. A sector is smallest addressable portion of a disk.
* When a read statement calls for a particular byte from a disk file, the

computer operating system finds the correct surface, track, and sector, reads the entire sector into a special area in memory called a buffer, and then finds the requested byte within that buffer.

* Disk drives typically have a number of platters. The tracks that are directly above and below one another form a cylinder. Significance of the cylinder is that all of the information on a single cylinder can be accessed without moving the arm that holds the read/write heads.
* Moving this arm is called seeking. The arm movement is usually the slowest part of reading information from a disk.



Write a short note on:

i) Cost of disk access

Seek Time: It is the time required to move the access arm to the correct cylinder. If we are alternately accessing sectors from two files that are stored at the opposite extremes on a disk (one on the innermost cylinder, one on the outermost cylinder), seeking is very expensive.

- Most hard disks available today have average seek time of less than 10 ms and high performance hard disks have average seek time as low as 7.5 ms

Rotational Delay: Refers to the time it takes for the disk to rotate so the sector we want is under the read/write head. Hard disk with rotation speed of 5000 rpm takes 12 ms for one rotation.

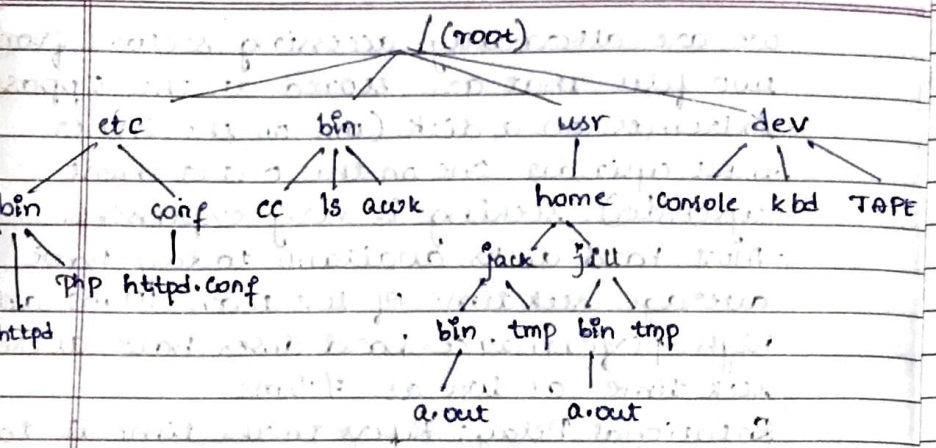
- On average, the rotation delay is half a revolution, or 6 ms.

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

$$\text{transfer time} = \frac{\text{no. of bytes transferred}}{\text{no. of bytes on a track}} \times \text{rotation time}$$

ii) UNIX file directory structure

* The UNIX file system is a tree-structured organization of directories, with the root of the tree signified by character /.
* In UNIX, the directory structure is a single tree for the entire file system.
* In UNIX, separate disks appear as subdirectories of the root (/).



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

iii) Seeking with C and C++:

Seeking with C:

* The C stream seek function, fseek let us set the read/write pointer to any byte in the file.
* Syntax: pos = fseek (file, byte_offset, origin); where

pos → A long integer value returned by fseek equal to the position of read/write pointer (in bytes) after it has been moved.
file → File descriptor of the file. TYPE: FILE*
byte_offset → no. of bytes to move from some origin to the file. Type: long.

This variable may have -ve values.
origin → Value that specifies starting position from which the byte_offset is to be taken. Type: int. It can have following values.

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.

Eg: long pos;
fseek (FILE * file, long offset, int origin);
FILE * file;
...
pos = fseek (file, 3732, 0);

Seeking with C++

* An object of type fstream has two file pointers
- A get pointer for input
- A put pointer for output.
* Two functions are supplied for seeking
- seekg: moves the get pointer
- seekp: moves the put pointer
* We often call both functions together since it is not guaranteed that the pointers move separately.
* Syntax:
file.seekg (byte_offset, origin);
file.seekp (byte_offset, origin);
* The value of origin comes from class ios; They are

ios::beg (beginning of file)

ios::cur (current position)
ios::end (end of file)
Eg: file.seekg (373, ios::beg);
file.seekp (373, ios::beg);

7 Discuss the steps involved in displaying the contents of file in C and C++ streams with a program? : [10 marks]

The file processing programs opens a file for input and reads it, character by character, sending each character to the screen after it is read from the 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.

* Steps 1 and 2 of program involve writing and reading, but in each of the implementations this is accomplished through the usual functions for handling the screen and keyboard.
* Step 4a, in which we read from the input file, is the first instance of actual file I/O.
* The fread function's first argument gives the address of a character variable used as the distination for the data, the 2nd & 3rd arguments are the element size and the number of elements; and the 4th argument gives a pointer to the file descriptor as the source for the input.

Using C streams.

```
// Program using c streams to read characters
// from a file and write them to the
// terminal screen
#include <stdio.h>
main () {
  char ch;
  File * file; //pointer to file descriptor
  Char filename[20];
  printf ("Enter the name of file: "); // Ste
  gets (filename);                     //step
  file = fopen (filename, "r");        //step
  while (fread (&ch, 1, 1, file) != 0)  //step4
    fwrite (&ch, 1, 1, stdout);         //step4
  fclose (file);                        //step 5
}
```

Using C++ streams:

```
// list contents of file using c++ streams
#include <fstream.h>
main () {
  char ch;
```

```
fstream file; // declare unattached fstream
char filename[20];
cout << "Enter name of file: "; // 1
      << flush;                 // force output
cin >> filename;                // 2
file.open(filename, ios::in);   // 3
file.unsetf(ios::skipws)        // include white space
while (1)
{
   file >> ch;                  // 4a
   if (file.fail())
      break;
   cout << ch;                  // 4b
}
file.close();                   // 5
}
```

8. Explain the different Record structure used in the organization of file. [10 m]

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

Method 1: Make Records a Predictable Number of Bytes: [Fixed-length Records]

* A fixed-length record file is one in which each record contains the same number of bytes.
* Fixed-length record structures are among the most commonly used methods for organizing files.
* We have a fixed number of fields, each with a predetermined length, that combine to make a fixed-length record. This kind of field and record structure is shown in below figure.

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

* However, the fixing of number of bytes in a record does not imply that the size or no. of fields in record must be fixed.
* Fixed-length records are frequently used as containers to hold variable numbers of variable-length fields.
* It is also possible to mix fixed and variable-length fields within a record as shown below

Ames | Mary | 123 Maple | Stillwater | OK | 74075 ← unused
Mason | Allen | 90 Eastgate | Ada | OK | 74820 ← unused

Method 2: Make records a predictable Number of fields.

* Rather than specifying that each record in a file contain some fixed number of bytes, we can specify that it will contain a fixed no. of fields.
* This is a good way to reorganize the records in the name and address file we have been looking at.
* The program asks for six pieces of information for every person, so there are six contigous fields on the file for each record.
* We could modify readdel to reorganize fields simply by counting the fields modulo

six, outputting record boundary information to the screen every time the count starts over.

es|Mary|123 Maple|Stillwater|OK|74075|Mason|Alan|....

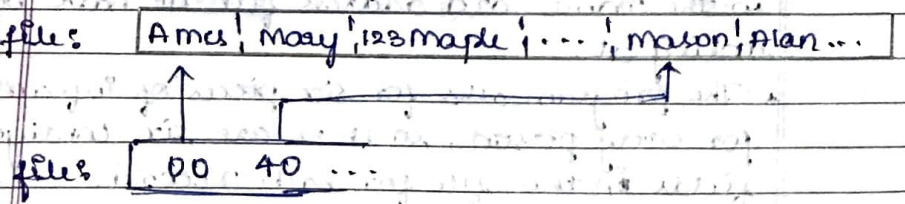
Method 3: Begin each record with a length indicator.

* We can communicate the length of records by beginning each record with a field containing an integer that indicates how many bytes there are in the rest of the record.
* This is commonly used for handling variable-length records.

Ames|Mary|123 Maple|Stillwater|OK|74075|82 Mason|.....

Method 4: Use an index to keep track of addresses

* We can use an index to keep a byte offset for each record in original file.
* The byte offset allows us to find beginning of each successive record and compute length of each record.
* We look up the position of a record in index then seek to the record in data file.



9

Explain the different ways of adding structure to files to maintain the identity of fields. [10 Marks]

Four of the most common methods follow:

Method 1: Fix the length of fields

* The fields in our sample file vary in length.
* If we force the fields into predictable lengths, we can pull them back out of the file simply by counting our way to the end of the field.
* We can define a struct in C or a class in C++ to hold these fixed-length fields.

Eg: 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];
};
```

Definition of record to hold person info.

* In this example, each field is a character array that can hold a string value of some maximum size.
* This kind of fixed-length structure changes our output so it looks like as shown below

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

* Disadvantage: Adding all the padding required to bring the fields up to a fixed length makes the file much larger. We can also encounter

problems with the data that is too long to fit into the allocated amount of space. We could solve this second problem by fixing all the fields at lengths that are large enough but this would make first problem even worse.

Method 2: Begin each field with a length indicator

* Another way to make it possible to count to the end of a field is to store the field length just ahead of the field.
* If the fields are not too long, it is possible to store the length in a single byte at the start of each field.
* We refer to these fields as length-based.

04Ames04Mary03123Maple10Stillwater020K0574075
05Mason04Alan11190Eastgate03Ada02OK05·74820

Method 3: Seperate the fields with Delimiters.

* We can also preserve the identity of fields by seperating them with delimiters
* All we need to do is choose some special character or sequence of characters that will not appear within a field and then insert that delimiter into the file after writing each field.
* White space would be a poor choice for our file since blanks often occur as legitimate characters within an address field. Hence, we use vertical bar instead of white-space characters. (tab, blank, new line).

Ames|Mary|123 Maple|Stillwater|OK|74075|

Method 4: Use a 'Keyword = Value' exp. to identify fields.

* This option has an advantage that the others do not. It is first structure in which a field provides info. about itself.
* Self-describing structures can be very useful tools for organizing files in many applications.
* It is easy to tell which fields the files contain. It is also a good format for dealing with missing fields.
* Unfortunately, for the address file this format also wastes a lot of space: 50% or more of files space could be taken up by the keywords.

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

10 What is data compression. Explain different techniques available for data compression [10M]

Data compression involves encoding the information in a file such that it takes up less space.

Data compression techniques:

Using a different Notation: (Redundancy notatn)

* Fixed length fields are good candidates for compression.
* The 'state' fied in 'person' record, as used earlier, is an example of such a field. There are 676 (26 x 26) possible two letter abbrevations, but there are only 50 states. By assigning an ordinal number to each state, and storing the code as

a one-byte binary number, the field size is reduced by 50%.
* No information is has been lost. The compression can be completely reversed, replacing the numeric code with two letter abbreviations when file is read.
Disadvantages:
i) By using pure binary encoding, we have made the file unreadable by humans
ii) Cost of encoding/decoding time
iii) Increased software complexity

Suppressing repeating sequences: (redundancy reduction).
[Run length encoding]
An encoding scheme which replaces the long runs of a single symbol with the symbol and a repetition factor.
- Run-length encoding is useful only when the text contains long runs of single value
- Run-length encoding is useful for images which contain solid color areas.
- Run-length encoding may be useful for text which contains strings of blanks.

Algorithm:
i) Read through the array in sequence except where same value occurs more than once in succession
ii) when same value occurs more than once, substitute the following 3 bytes in order
- Special run length code indicator
- Value that is repeated
- No. of times the value is repeated

Ex: Uncompressed text (hexadecimal format):
40 40 40 40 40 40 43 43 41 41 41 41 42
Compressed text (hexadecimal format):
FE 06 40 43 43 FE 05 41 42
Where FE is compression escape code, followed by a length byte and byte to be repeated.
Disadvantage: No Guarantee that space will be saved.

Assigning variable length code:
Principle: Assign short codes to most frequently occuring values and long codes to least frequent once.
* The code size cannot be fully optimized as one wants code to occur in succission, without delimiters between them & still be recognized.
* This is the principle used in morse code and Huffman coding as well.
Example showing Huffman encoding for a set of seven letters, assuming certain probabilities.

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

Irreversable compression technique:
* It is based on assumption that some info. can be sacrificed.
* Example: Shrinking a raster image from 400 by 400 px to 100 by 100 px. There is no way to determine what the original pixels were from the one new pixel.
* In data files, irreversable compression is

seldom used; however they are used in image and speech processing.

11 Explain how space can be reclaimed in fixed length and variable length records. [10 m]

Deleting fixed length Records.

Linked list

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

Avail list: A list of unused spaces in a file.

Stack:

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

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

Record 1	Record 2	*	Record 3	Record 4

* To reuse the empty space, there must be a mechanism for finding it quickly.
* One way of managing the empty space within a file is to organize as a linked list, known as avail list.
* The location of first space on the avail list, the head pointer of linked list, is placed in the header record of the file.
* Each empty space contains the location of the next space on the avail list, except for the last space on the list.

Page No.
Date: / 201

* The last space contains a number which is not valid as a file location, like -1.

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

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

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

* When a record is added, it is placed in the space which is at the head of the avail list.
* The push process is reversed; the empty space is popped from the stack by moving the pointer in the first space to the header record as new avail list head pointer.
* With fixed length records, the relative record numbers (RRNs) can be used as location pointers in the avail list.

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

Deleting Variable-length Records

* If the files uses variable length records, the spaces are not interchangable; a new record will not just fit into any unused space.
* For record reuse, we need
- A way to link the deleted records together in a list.
- An algorithm for adding newly deleted records to the avail list
- An algorithm for finding and removing record from the avail list when we are ready to use them.

* An avail list of variable length records
- we place single asterisk in first field of deleted record followed by a binary link field pointing to next deleted record on avail list. we cannot use RRN links

Illustration:

Figure shows sample file illustrating variable length record deletion.

HEAD. FIRST_AVAIL: -1

18 Ames | Mary | 123 USA | 17 James | John | 45 UK | 24 FOLK | Michael | 150 London

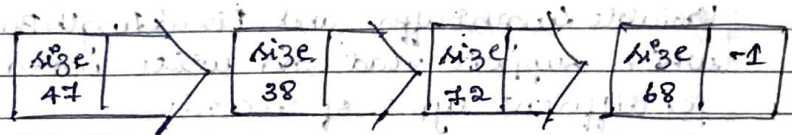
(A) Original sample file stored in variable length format with byte count.

HEAD. FIRST AVAIL: 21

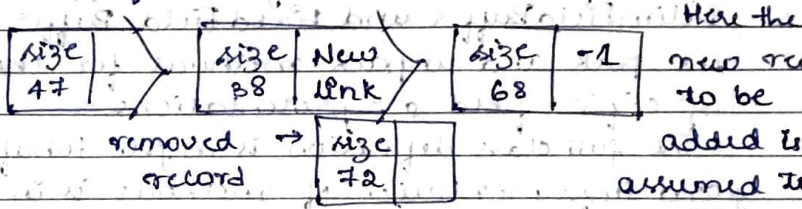
18 Ames | Mary | 123 USA | 17 * | -1 .... | 24 FOLK | Michael | 150 London

(B) Sample file after deletion of 2nd record.

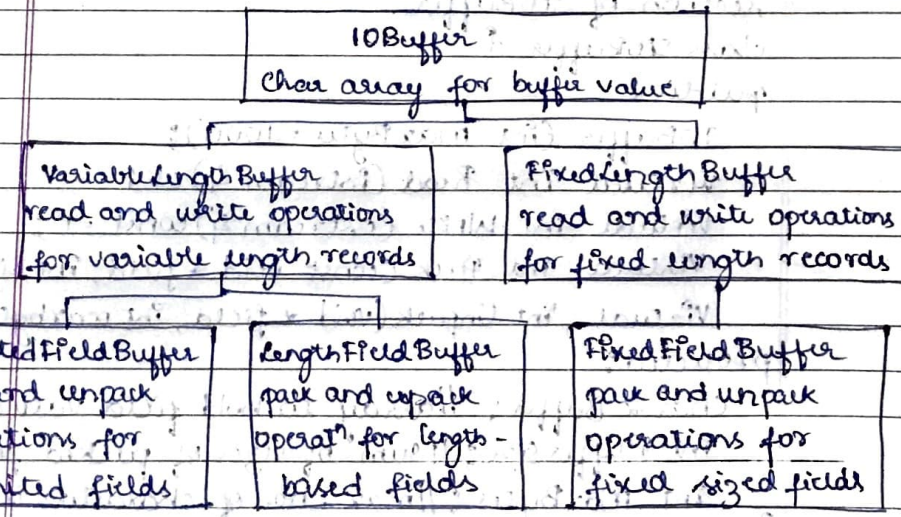
* Adding and removing records
- Here, we cannot access the avail list as a stack, since the avail list differ in size We search through the avail list for a record slot that is the right size.
- Figure shows removal of a record from avail list



(a) Before removal



12 Explain Buffer class hierarchy with neat diagram. [10 marks]



The characteristics of the three buffer classes can be combined into a single class hierarchy.

* The members and methods that are common to all of the three buffer classes are included in the base class IOBuffer.
* Other methods are in classes VariableLengthBuffer and FixedLengthBuffer, which support read and write operations for different types of records.
* Finally the classes LengthFieldBuffer, DelimFieldBuffer and FixedFieldBuffer have the pack and unpack methods for the specific field representations.
* The full class definition is in file iobuffer.h, and implementation of methods is in file iobuffer.cpp. The common members of all of the buffer classes, BufferSize, MaxBytes, NextByte and Buffer, are declared in class IOBuffer. These members are in the protected section of IOBuffer.

```
class IOBuffer {
public:
   IOBuffer (int maxBytes = 1000);
   virtual int Read (istream &) = 0;
   virtual int Write (ostream &) const = 0;
   virtual int Pack (const void * field, int size = -1);
   virtual int Unpack (void * field, int maxbytes = -1);
protected:
   char * Buffer; // array to hold field values
   int BufferSize; // sum of sizes of packed fields
   int MaxBytes; // max. no. of characters in buffer
};
```

* The protected members of IOBuffer can be used by methods in all of the classes in this hierarchy. Protected members of VariableLengthBuffer can be used in its subclasses but not in classes IOBuffer and FixedLengthBuffer.
* The full implementation of read, write, pack & unpack operations for delimited text records is supported by two more classes.
* The reading and writing of variable-length records are included in the class VariableLengthBuffer, and files Varlen.h and Varlen.cpp. Packing and Unpacking delimited fields is in class DelimitedFieldBuffer and in files delim.h and delim.cpp.

```
class VariableLengthBuffer : public IOBuffer {
public:
   VariableLengthBuffer (int MaxBytes = 1000);
   int Read (istream &);
   int Write (ostream &) const;
   int SizeOfBuffer () const;
};
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;
};
```

* The full implementation of I/O Buffer classes includes class LengthFieldBuffer,

which supports field packing with length plus values representation. This class is like DelimFieldBuffer is that it is implemented by specifying only the pack and unpack methods.

* The read and write operations are supported by base class, VariableLengthBuffer.