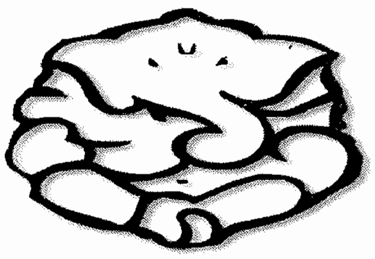


6TH CSE/ISE

FILE STRUCTURES

ASHOK KUMAR K
VIVEKANANDA INSTITUTE OF TECHNOLOGY
MOB: 9742024066
e-MAIL: celestialcluster@gmail.com

S. K. L. N ENTERPRISES
Contact: 9886381393

UNIT 1 **INTRODUCTION**

syllabus.

chapter 1A: Introduction

* Heart of filestructure design.
* short history of filestructure design.
* Conceptual tool kit

chapter 1B: Fundamental file processing operations.

* physical & logical files
* opening files.
* closing files
* reading & writing
* seeking.
* special char. in files.
* unix directory structure.
* physical devices & logical files.
* file related header files.
* unix file system commands.

chapter 1C: secondary storage & system software.

* DISKS
* magnetic Tapes.
* Disk versus Tape.
* Introduction to CD ROM.
* physical organisation of CD Rom.
* CD ROM strengths & weaknesses.
* storage as hierarchy.
* A Journey of a byte.
* Buffer management.
* I/O in unix.

- 7 Hours.

Ashok Kumar K

VIVEKANANDA INSTITUTE OF TECHNOLOGY

1. What are File structures? Why study file structure design?
2. Briefly explain evolution/ history of file structures.
3. Explain
   a) physical files and logical files.
   b.) opening and closing files w.r.t unix OS.
   c) Reading and writing wrt Unix OS.
4. What are streams? Explain seeking with C and C++ stream classes in detail.
5. Explain how data on magnetic disks is organised with relevent sketches.
6. Explain the organization of data on tapes with a neat diagram. Estimate the tape length requirement with a suitable example.
7. Explain briefly physical organization of a CD ROM.
8. Discuss CD ROM strengths and weaknesses.
9. Explain Journey of a byte.
   (or) Explain with neat diagram, what happens when the prg statement write(textfile, ch, 1) is executed?
10. Write Explanatory notes on Buffer management.

CHAPTER 1A:

INTRODUCTION TO DESIGN AND SPECIFICATION OF FILE STRUCTURES.

What are the File Structures?

* 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 criteria or reading through the data in some particular order.

What are the primary issues that characterize file structure design.?
(or) why study File structure design?

i.) Data storage

* Computer Data can be stored in three kinds of Locations
  → Primary storage (computer memory)
  → Secondary storage (online Disk/Tape/CDrom that can be accessed by the computer)
  → Tertiary storage (offline Disk/Tape/CDrom not directly available to the computer).

ii.) computer memory v/s secondary storage

* Secondary storage such as disks can pack thousands of megabytes in a small physical location.
* Computer memory (RAM) is limited.
* However, relative to RAM, access to secondary storage is extremely slow.

iii.) How can secondary storage access time be improved?

* By improving File structures.

What are the general goals of Filestructure Design?

i.) Get the information we need with one access to the disk.

ii.) If that's not possible, then get the information with as few accesses as possible.

iii.) Group information so that we are likely to get everything we need with only one trip to the disk.

Explain Briefly the evolution of File structure?
(or) Short history of filestructure.

i.) Early work.

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

ii.) Emergence of Disks and indexes.

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

iii.) Emergence of Tree Structures.

* As indexes also have a sequential flavour, when they grew too much, they also became difficult to manage.
* Idea of Using Tree structures to manage the index emerged in the 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.

iv.) Balanced Trees

* ~~In~~ 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 rather th

* 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 or 4 Trips to the ~~file~~ disk.

v.) <u>Hash Tables.</u>
* Retrieving entries in 3 or 4 accesses is good, but 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, but do not work well with volatile, dynamic files.
* Extendible, dynamic Hashing reaches this goal.

CHAPTER 1B :

FUNDAMENTAL FILE PROCESSING OPERATIONS

PHYSICAL FILES AND LOGICAL FILES

Physical File

* A File that actually exists on secondary storage. It is the File as known by the computer OS and that appear in its File Directory.

(or) A collection of bytes stored on a disk or Tape.

Logical File

* The File as 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 file's location and physical format to the program.

* This logical file will have logical name which is what is used inside the program.

OPENING FILES

* once we have a logical File identifier hooked up to a physical file or device, we need to declare what we intend to do with the file.
  → open an existing File
  → create an new file

This makes file ready to use by the program. We are positioned at the beginning of the file and are ready to read or write.

* unix system function open ( ) [API] is used to open an existing file or create a new file.
many C++ implementation supports this function.
It is defined in the header file

* 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 values 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_CREAT : Create and open a file for writing. It has no effect if file already exists.
O_EXCL : return an error if O_CREAT 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 reading & writing.
O_TRUNC : If file exists, truncate it to a length of zero, destroying its contents.

pmode → required if O_CREAT is specified. TYPE : int
It specifies the protection mode for the file.
In unix, pmode is a three-digit octal number that indicates how the file can be used by the owner (1st digit), by the members of the group (2nd digit), and by every one else (3rd digit)

$$\text{eg: pmode} = 751 = \underbrace{\overset{r\ w\ e}{1\ 1\ 1}}_{\text{owner}}\quad \underbrace{\overset{r\ w\ e}{1\ 0\ 1}}_{\text{group}}\quad \underbrace{\overset{r\ w\ e}{0\ 0\ 1}}_{\text{world}}$$

* Examples .

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

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

↳ creates new files for reading & writing . If it already exists, its contents are truncated .

iii.) fd = open (filename , O_RDWR | O_CREAT | O_EXCL , 0751);

CLOSING FILES

* Makes the logical filename available for another physical file (it's like hanging up the telephone after a call)
* Ensures that everything has been written to the file . [since data is written to the buffer prior to the file] .
* Files are usually closed automatically by the OS (unless the program is abnormally interrupted) when a program terminates normally .

READING AND WRITING

note : < Definitions >

* open() is a function or system call that makes the file ready for use . It may also bind a logical filename to a physical file .
* close() is a function or system call that breaks the link b/w a logical file name & corresponding physical filename .

READING AND WRITING

* 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 , Destination-addr , size) ;

where ,

source-file → location , the program reads from. ie ~~its~~ logical file name (like phone Line)

Destination-addr → First address of memory block where we want to store the data .

size → how much information is being bought in from the file < Byte count >

Write Function

* It is an function or system call used to provide output capabilities .
* syntax :

Write (Destination-file , source-addr , size)

where,

Destination-file → logical filename where the data will be written .

source-addr → First address of the memory block where the data to be written is stored.

size → number of bytes to be written .

Files with C-streams and C++ streams

* In C or C++ , a file is a stream of data (stream can be a file or some other source or consumer of data)
* There are two sets of I/O operations.
  - C streams in stdio.h
  - C++ stream classes in iostream.h and fstream.h

C streams

* There are three standard streams :
  stdin, stdout, and stderr.

* opening file.

  file = fopen (filename, type);

  where,

  file → Type: FILE * ; A pointer to file descriptor.
  It is set to null, if there is error in opening file.

  filename → Type : ~~FILE *~~ char * ; filename.

  Type → Type : char * , controls the operation of open function. following values are supported.

"r"	: open an existing file for i/p
"w"	: create new file, or truncate an existing one for output.
"a"	: create new file, or append to an existing one for output.
"r+"	: open existing file for i/p & o/p.
"w+"	: create new file or truncate existing one for i/p & o/p.
"a+"	: create new file or append to existing one for i/p + o/p.

* closing file.

  fclose (FILE *fp);

* Reading file

  fread (void *buf, size_t size, size_t num, FILE *fp)

* Writing file

  fwrite (const void *buf, size_t size, size_t num, FILE *fp

  fputc (int ch, FILE *fp);

  fputs (const char *buf, FILE *fp);

  fprintf (FILE *fp, const char *format, ...)
  //write formatted data to fp.

C++ streams.

* opening file
    - → constructor
    - → member function open.
* closing file
    - → Destructor
    - → member function close
* Reading file
    - → overload extracting operator <<
    - → read, get, getline
* writing file
    - → overload inserting operator >>
    - → write, put.

program to Display the contents of a File (~~using C++~~)

File listing program using C streams

```
#include<stdio.h>
main()
{ char ch;
  FILE *fp;
  char filename[20];
  printf("Enter the filename");
  gets(filename);
  fp=fopen(filename,"r");
  while((fread(&ch,1,1,fp))!=0)
      fwrite(&ch,1,1,stdout);
  fclose(fp);
}.
```

The file listing program using c++ stream classes.

```
#include <fstream.h>
main()
{ char ch;
  fstream file;
  char filename[20];                        // force output
  cout<<" Enter filename"<<flush;
  cin>>filename;
  file.open (filename, ios::in);
  file.unsetf (ios::skipws); // include white space
                             // in read

   while (1)
   { file>>ch;
      if(file.fail())
          break;
      cout<<ch;
   }
  file.close();
}
```

Detecting End-of-file

* In unix
  - read returns 0
* using C streams
  - fread returns -1
  - feof returns true.
* using C++ streams classes
  - fail returns true
  - eof returns true.

SEEKING (used to provide random access)

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

* syntax: (in unix)

  seek (source-file , offset)

  where,

  source-file → logical filename in which the seek will occur.

  offset → no. of positions in the file the pointer is to be moved from the start of file

* eg: seek (data , 373)

* **Definition:** seek is a function or system call that sets the read/write pointer to a specified position in the file.

Seeking with C streams

* The cstream seek function, fseek lets 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 in 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 pos

* 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
```

* Eg:

```
long pos;
fseek (FILE *file , long offset, int origin);
FILE *file;
  =
pos = fseek (file , 373L , 0);
```

<u>Seeking with C++ stream classes</u>

* 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.
* note: 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);
```

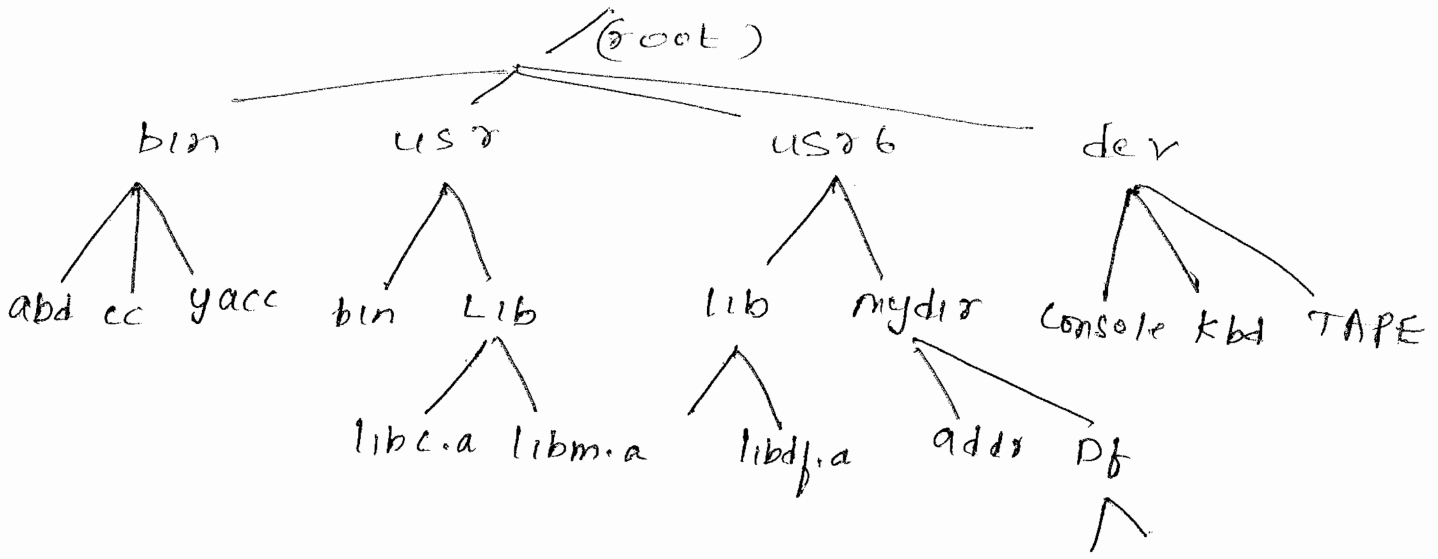
SPECIAL CHARACTERS IN FILES

* Sometimes, the OS ~~makes~~ attempts to make user's life easier by automatically adding or deleting characters for them.
* These modifications, ~~having~~ however, make the life of programmers building sophisticated file structure more complicated.
* Examples:
  i.) Control-Z is added at the end of all files (MS-DOS). This is to signal an end-of-file.
  ii.) <carriage-return> + <line-feed> are added to the end of each line (again MS-DOS)
  iii.) <carriage_return> is removed & replaced by character count on each line of text (VMS)
* programmers building sophisticated file structure must spend a lot of time finding ways to disable this automatic assistance so they can have complete control over what they are building.

THE UNIX DIRECTORY STRUCTURE

* In many computer systems, there are many files (100's or 1000's). To provide convenience access to such large no. of files, these should be organised using some method. In unix this is called file system.
* unix file system is a tree-structured organization of directories, with the root of tree represented by the character / .
* All directory can contain two kinds of files: regular files and directories (refer fig).
* Filename stored in a unix directory corresponds to its physical name.
* Any file can be uniquely identified by giving its

* The directory you are in is called your current directory, where you issue commands to unix system.
* you can refer to a file by the path relative to the current directory.
* . stands for current directory.
  .. stands for parent directory.
* Fig shows sample unix directory structure.



PHYSICAL DEVICES AND LOGICAL FILES.

physical Devices as files

* unix has a very general view of what a file is. it corresponds to a sequence of bytes with no worries about where the bytes are stored or where they originate
* magnetic disks or tapes can be thought of as files and so can the keyboard & console.
  In above fig /dev/kbd & /dev/console.
* no matter what the physical form of a unix file (real file or device), it is represented in the same way in unix; by an integer - the file descriptor. This integer is an index to an array of more complete information about the file.

The console, the keyboard, and standard error

* stdout → console
  eg: fwrite(&ch, 1, 1, stdout);
* stdin → keyboard
  eg: fread (&ch, 1, 1, stdin);
* stderr → standard error (again console)
  when compiler detects error, the error message is written to this file.

I/O Redirection and pipes.

* < Filename [redirect stdin to "filename"]
* > Filename [redirect stdout to "Filename"]

eg: i) a.out < my-input > my-output
  ii) list.exe > myfile.

* pipes
  program1 | program2

  means take any stdout output from program1 and use it in place of any stdin input to program2.
  eg: list | sort

UNIX FILE SYSTEM COMMANDS.

* cat Filenames.
* tail Filename → Print last 10 lines of text file
* cp file1 file2
* mv file1 file2 → (rename)
* rm filenames
* chmod mode filename
* ls
* mkdir name
* rmdir name

CHAPTER 1C :

SECONDARY STORAGE AND SYSTEM SOFTWARE

DISKS

* There are two classes of devices.
  → Direct Access Storage Devices (DASDs)
  → Serial Devices.

  Disks belong to DASDs because they make it possible to access the data directly.
  Serial Devices permits only serial access (eg: mag. Tape)

* Different types of Disks:
  → Hard disk : High capacity + Low cost per bit
  → Floppy disk : Cheap, but slow & holds little data.
  → optical disk (CDROM) : Read only, but holds lot of data and can be reproduced cheaply. However, slow.

organisation of Disks.

* The information stored on a disk is stored on the surface of one or more platters. (refer fig)
  This arrangement is such that the information is stored in successive tracks on the surface of the disk.

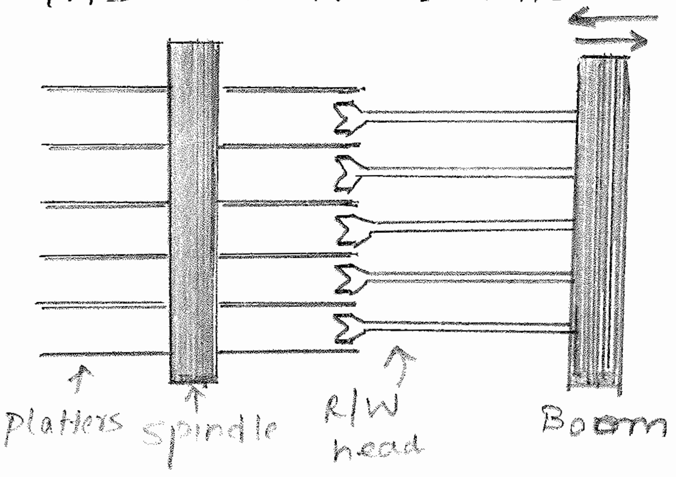
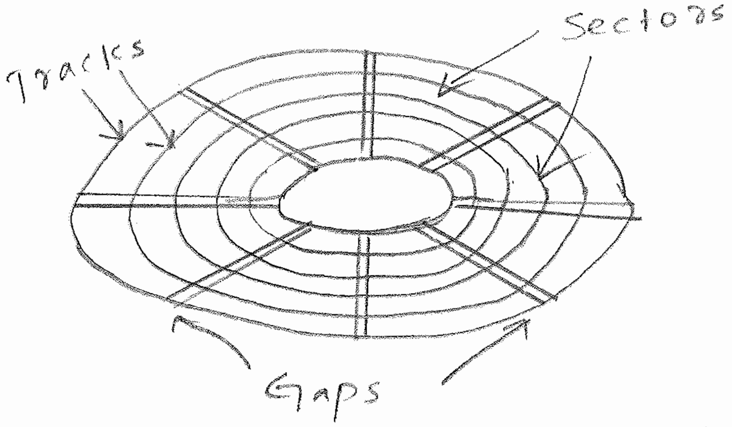


Illustration of disk drive



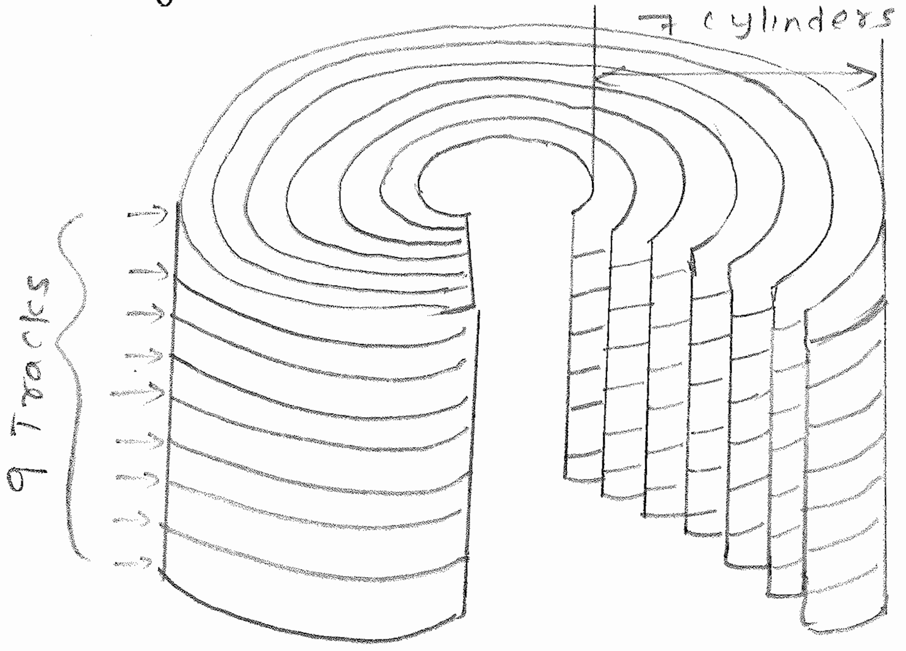
Surface of disk showing track & sectors

* 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 comp. OS finds the correct platter, track & sector, reads entire
  sector into a special area in m/m called 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.



Estimating capacities and space needs.

* Disk ranges in width from 2 to 14 inches. commonly 3.5".
* capacity of disk ranges from several MB to several hundreds of GB.
* In a disk, each platter can store data on both sides, called surfaces.
  - Number of surfaces is twice the no. of platters.
  - no. of cylinders is same as no. of tracks on a single surface
  - Bit density on a track affects the amount of data can be held on the track surface.
  - A low density disk can hold about 4KB on a track and 35 tracks on a surface.
  - A top-of-the line disk can hold more than 1MB on a track and more than 10000 tracks on a surface (cylinders).

* (Track capacity) = (no. of sectors per track) * (Bytes per sector)

(cylinder capacity) = (no. of tracks per cylinder) * (Track capacity)

(Drive capacity) = (no. of cylinders) * (cylinder capacity) problem: Given:

no. of bytes per sector = 512
no. of sectors per track = 63
no. of Tracks per cylinder = 16
no. of cylinders = 4092

How many cylinders does the file require if each data record required 256 bytes.

no. of data records = 50,000 fixed length.

Each sector can hold = 2 records.

Soln: File requires $\frac{50,000}{2} = 25000$ sectors.

one cylinder can hold $16 * 63 = 1008$ sectors.

$\therefore$ no. of cylinders required is $\frac{25000}{10008} = 24.8 = \boxed{25 \text{ cylinders}}$

~~organizing Tracks by sector.~~

~~The physical placement of sectors.~~

There are two basic ways for organizing data on disk.

→ By sector
→ user defined block.

organising Tracks by sector.

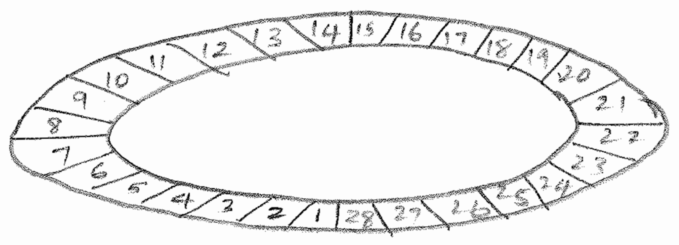
The physical placement of sectors.

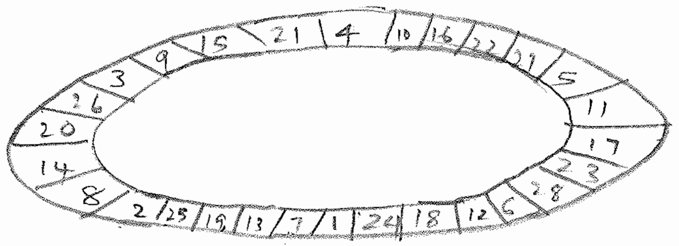
* The most practical logical organisation of sectors on a track is that sectors are adjascent, fixed sized segments of a track that happens to hold a file.
* physically, however, this organisation is not optimal: After reading the data, it takes the disk controller some time to process the received information before it is ready to accept more. If sectors are physically adjascent, we would use the start of the next sector while processing the information just read in.

* We can physically place sectors in two ways:
  → physically adjascent sector (explained above)
  → interleaving sectors.

* Traditional solution for the problem created by physically adjascent sector is to interleave the sector. ie leave an interval of several sectors (interleaving factor) b/w logically adjascent sectors.
* nowadays, (in early 1990's) however, the controller's speed has improved so that no interleaving is necessary

fig: Two views of organisation of sectors on a 28 sector track.


(a) physically adjascent sector


(b) interleaving sector.
interleaving factor : 5.

clusters

* The file can also be viewed as series of clusters of sectors which represent a fixed number of (logically) contiguous sectors. (not physically). The degree of physical contiguity is determined by the interleaving factor.
* Once a cluster has been found on disk, all sectors in that cluster can be accessed without requiring an additional seek.
* The file manager ties logical sectors to the physical clusters they belong to, by using a File allocation Table (FAT)
* The system administrator can determine how many sectors in a cluster.

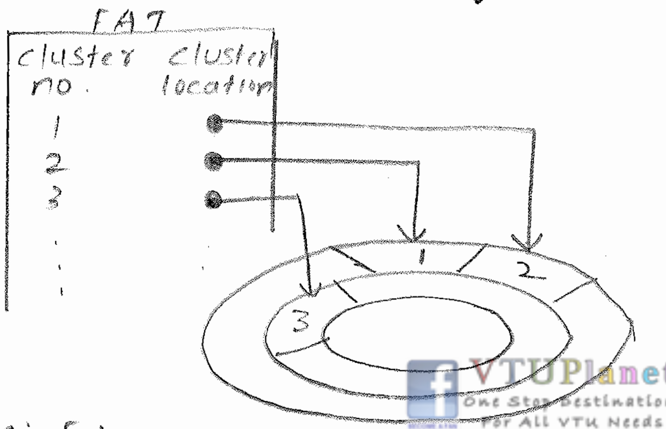


fig: File manager determines which

~~* If there is~~ a free room on a disk.

Extents.

* If there is a lot of free room on a disk, it may be possible to make a file consist entirely of contiguous clusters. Then, we say that the file consists of one extent: all of its sectors, tracks, and (if it is large enough) cylinders form one contiguous whole (refer fig a) Then whole file can be accessed with minimum amount of seeking.

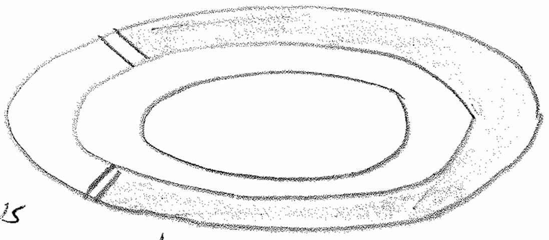


fig (a)

* If there is not enough contiguous space available to contain an entire file, the file is divided into two or more non contiguous parts. Each part is an extent. (refer fig b)
* As the number of extents increases in a file, the file becomes more spread out on the disk, & the amount of seeking necessary increases.

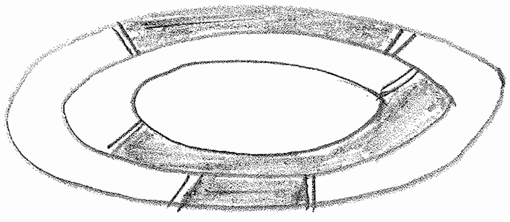


fig (b).

Fragmentation

* Generally, all sectors on a given drive must contain same no. of bytes.
 There are two possible organisations for records (if the records are smaller than the sector size)
 1.) Store 1 record per sector.
 2.) Store records successively (ie one record may span two sectors).

Adv:
* Each record can be retrieved from one sector
~~* No internal~~

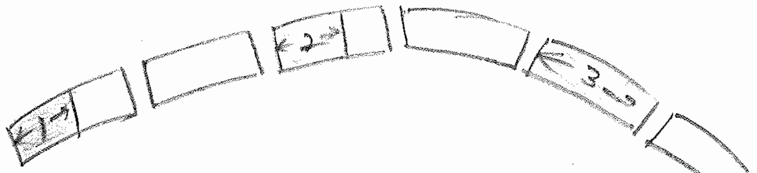
Disadv
* Loss of space with each sector. This is called internal fragmentation.

Adv.
* no internal fragmentation.

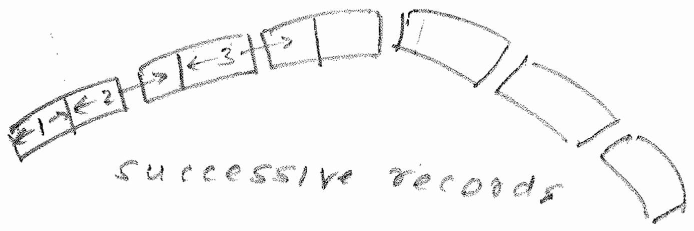
Disadv
* Two sectors may need to be accessed to retrieve a single record.

* Definition : Loss of space within a sector is called internal fragmentation.

* use of clusters also leads to internal fragmentation. If number of bytes in a file is not a multiple of the cluster size, internal fragmentation will occur in the last extent of the file.



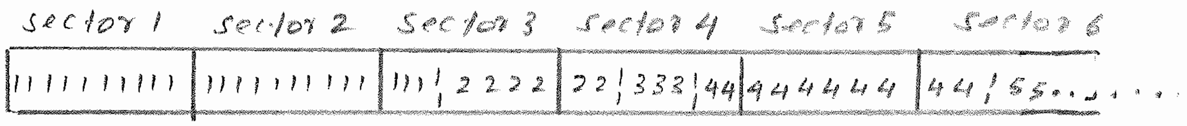
1 record per sector.



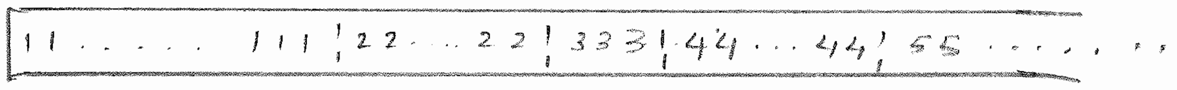
successive records

Organizing Tracks by Blocks.

* Rather than dividing into sectors, the disk tracks can be (or may be) divided into user defined blocks, whose size can vary.
* when the data on a track is organized by block, this usually means that the amount of data transferred in ~~one~~ single I/O operation can vary depending on the needs of s/w designer, not the hardware.
* Blocks can normally be either fixed or variable in length, depending on the requirements of the file designer and the capabilities of OS.



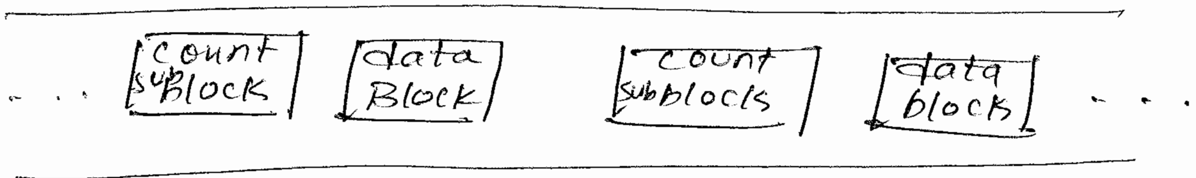
(a) Data stored on sectored track.



(b) Data stored on a blocked track.

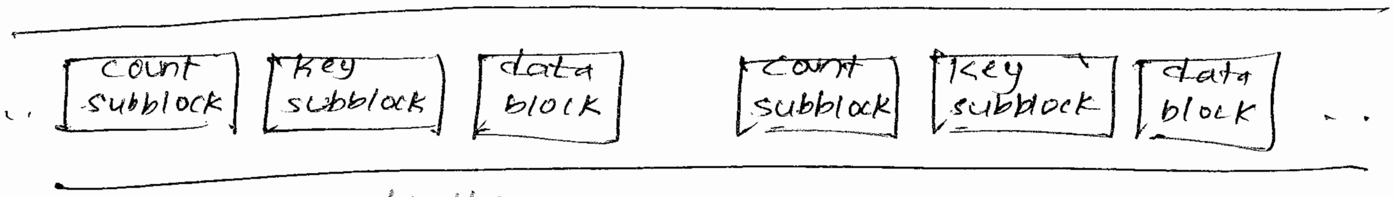
* Blocks dont have sector spanning and fragmentation problem of sectors since they vary in size to fit the logical organisation of data.
* The term Blocking factor indicates the number of records that are to be stored in each block in a file.

* Each block is usually accompanied by
~~a)~~ subblocks containing extra information about the data block.
 i) ~~count~~ count subblock - contains number of bytes in accompanying data block.
 ii) Key subblock - allow the disk controller to search a track for a block or record identified by a given key.
 ↳ contains the key for last record in the data block.



fig(a)

~~coun~~



fig(b).

Non Data Overhead.

* Both blocks and sectors requires that a certain amount of space be taken up on the disk in the form of non data overhead. Some of the overhead consists of information that is stored on the disk during preformatting.
* On sector-addressable disks, preformatting involves storing (at the beginning of each sector)
 → sector address
 → Track address
 → condition (usable or defective)
 → gaps & synchronization marks b/w fields of information to help R/W mechanism distinguish b/w them.
* On block-organised disks,
 → sub blocks
 → inter block gaps.

problem:

If there are ten 100-byte records per blocks, each block holds 1000 bytes of data & uses 300 + 1000, or 1300 bytes of track space when over head is taken into account. Find no. of blocks ~~needed to~~ that can fit on a 20000 byte track.

Soln:- $\frac{20000}{1300} = 15.38 = \boxed{15}$.

so 15 blocks, or 150 records can be stored per track.

problem: If there are sixty 100-byte records per block, each block holds 6000 bytes of data & uses 6300 bytes of track space, Find no. of blocks per track.

Soln:- $\frac{20000}{6300} = \boxed{3}$

So 3 blocks, or 180 records can be stored per track.

Cost of a Disk access.

* Seek time: Time required to move the access arm to the correct cylinder on a disk drive.
* Rotational Delay: Time it takes for the disk to rotate so the desired sector is under read/write head.
* Transfer time:
  once the data ~~we~~ want is under R/W head, it can be transferred.

$$\left(\begin{matrix}\text{Transfer}\\ \text{Time}\end{matrix}\right) = \frac{\text{number of bytes Transferred}}{\text{no. of bytes on a track}} \times \left(\begin{matrix}\text{rotation}\\ \text{time}\end{matrix}\right)$$

problem: Suppose the disk has 256 sectors per track with 10000 rpm (resolutions per minute), average seek time = 10ms, average rotational delay = half resolution = ~~5~~ $\frac{1}{2} \times \frac{1}{10000}$ min = 3ms.

And the file is stored as

i.) random sectors. (ie we can read only 1 sector at a time)
ii.) random clusters. (each cluster has 8 sectors(4KB))
iii.) One extent

Soln:

case 1: Assume the file is read sector by sector in random. Then ~~time t~~

average seek = 10 ms

rotational delay = 3 ms.

Time to read one sector = $\frac{1}{256} \times \frac{1}{10000}$ min = 0.023 ms.

Total = 10 ms + 3 ms + 0.023 ms = 13.023 ms.

$\therefore$ Total time = 250000 * 13.023 ms = $\boxed{54 \text{ minutes}}$

case 2: Assume the file is read cluster by cluster in random.

average seek = 10 ms

rotational delay = 3 ms.

Time to read one cluster = $\frac{8}{256} * \frac{1}{10000}$ min = 0.187 ms.

Total = 13.187 ms.

$\therefore$ Total Time = $\left(\frac{250000}{8}\right)$ * 13.187 ms = $\boxed{6.9 \text{ minutes}}$.

Case 3: sequential access.

average seek = 10 ms * 41 = 410 ms.

rotational delay = 3 ms.

read one extend = $\left(\frac{250000}{256}\right) * \left(\frac{1}{10000}\right)$ min = 5859.4 ms

$\therefore$ Total time = 410 ms + 3 ms + 5859.4 ms = $\boxed{6.3 \text{ seconds}}$

Conclusion:

* Seeking is most expensive operation. Avoid seeking as much as possible.
* Grouping data into larger units (eg clusters) can reduce access time.
* sequential access is much faster than random access

Disk as Bottleneck.

* processes are often Disk bound. ie the network & the CPU often have to wait inordinate lengths of time for the disk to transmit data.
* solution 1: multiprogramming (CPU works on other jobs while waiting for the disk)
* solution 2: stripping
  Disk stripping involves splitting the parts of a file on several different drives, then letting the separate drives deliver part of the file to the n/w simultaneously.
  (It achieves parallelism)
* solution 3: RAID: Redundant array of independent disks.
* solution 4: RAM disks: simulate the behaviour of mechanical disk in m/m. (provides faster access).
* solution 5: Disk cache: large block of m/m configured to contain pages of data from a disk.
  ~~[illegible]~~ working: check cache first, if not there, go to the disk & replace some page in cache with the page from disk containing the data.

MAGNETIC TAPE

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

Organization of Data on nine-track Tapes.

* on a tape, the logical position of a byte within a file corresponds directly to its physical ~~loca~~ position relative to the start of the file.

* The surface of a typical tape can be seen as a set of parallel tracks, each of which is a sequence of bits. If there are nine tracks (see fig), the nine bits that are at corresponding positions in the nine respective tracks are taken to constitute 1 byte, plus a parity bit.
So a byte can be thought of as a one bit wide slice of tape. such a slice is called a frame.

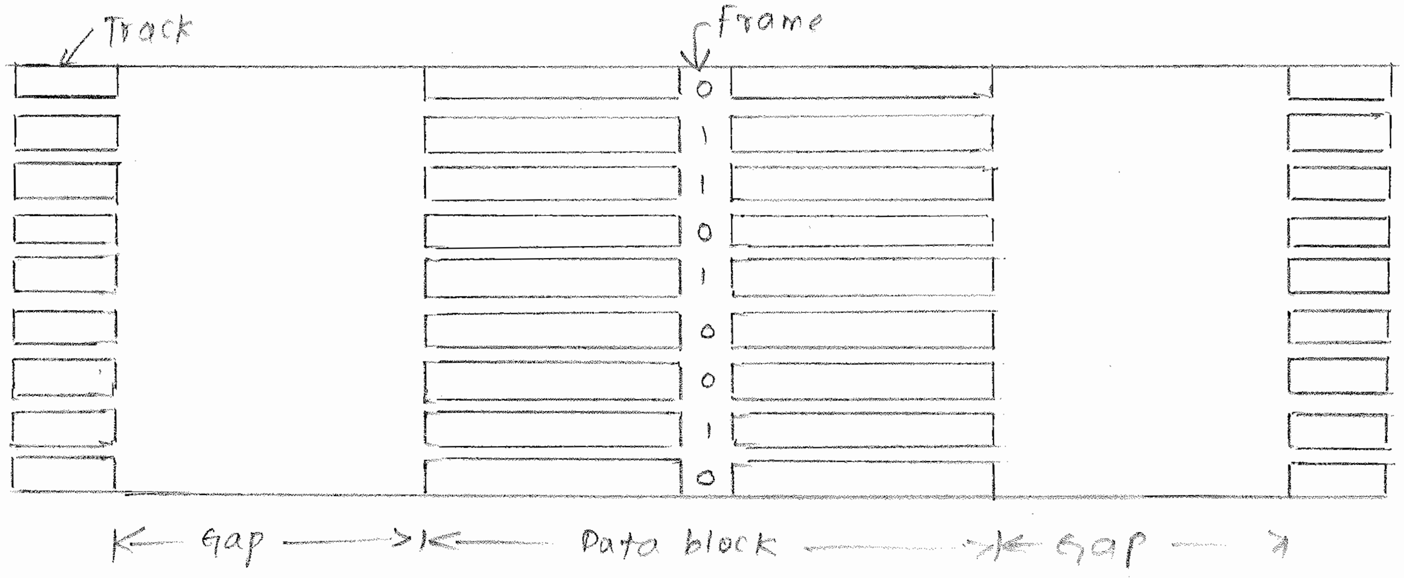


fig: nine-track Tape.

* In odd parity, the bit (frame) is set to make the number of bits in the frame odd. This is done to check the validity of data.

* Frames are organised into data blocks of variable size separated by interblock gaps (long enough to permit stopping and starting).

Estimating Tape length requirements.

* let b = physical length of data block
  g = ~~number of~~ length of an interblock gap, and
  n = number of datablocks.

Then the space requirement s, for storing the file is

$$\boxed{s = n * (b + g)}$$

and,

$$\boxed{b = \frac{\text{blocksize (bytes per block)}}{\text{tape density (bytes per inch)}}}$$

Example: file has ~~10~~ one million 100 byte records.
If we want to store the file on a 6250-bpi tape that has interblock gap of 0.3 inches, how much tape is needed?

solution: $b = \frac{\text{blocksize}}{\text{Tapedensity}} = \frac{100}{6250} = \underline{\underline{0.016 \text{ inch.}}}$

$s = 1000000 * (0.016 + 0.3) \text{ inch}$
$= 1000000 \times 0.316 \text{ inch}$
$= 316000 \text{ inches}$

or $\boxed{s = 26333 \text{ feet}}$

note: no. of records stored in a physical block is called blocking factor.
Here we have chosen blocking factor as 1 because each block has only one record.

Effective measure density : It is the general measure of the effect of choosing different block sizes.

It is defined as

$$= \frac{\text{number of bytes per block}}{\text{number of inches required to store a block}}.$$

In above ex, it is

$$= \frac{100 \text{ bytes}}{0.316 \text{ inches}} = \boxed{316.4 \text{ bpi}}$$

Estimating Data Transmission times.

* normal data transmission rate } = Tapedensity(bpi) * tape speed (ips)

* Interblock gaps, however must be taken into consideration.

∴ Effective transmission rate } = (Effective recording density) * (Tape speed)

DISK VERSUS TAPE

* In past : Both disk and tapes were used for secondary storage. Disks were prefered for random access & tape for better sequential access.

* Now : Disks have taken over much of secondary storage because of decreased cost of disk + memory storage. Tapes are used as tertiary storage.

INTRODUCTION TO CD ROM

* ~~A~~ CDRom is an acronym for compact Disk Read only memory.
* A single disk, can hold more than 600 mega bytes of data (≃ 200,000 printed pages).
* CD Rom is read only. ie it is a publishing medium rather than a data storage and retrieval like magnetic disks.
* CD Rom strengths: High storage capacity, inexpensive price, Durability.
* CDRom weaknesses: Extremely slow seek performance. (b/w 1/2 a second to second). This makes intelligent filestructures difficult.

PHYSICAL ORGANIZATION OF CD ROM

* CDRom is a descendent (child) of CD audios. ie listening to music is sequential and does not require fast random access to data.

Reading pits and lands.

* CDROMs 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 by the beam. The smooth unchanged areas between the pits are called lands.
* when we read the stamped copy of the disk, 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 reflects most 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 b/w transitions, the more 0's we have

* Given this scheme, it is not possible to have two adjascent 1's : 1's are always separated by 0's
  Infact, (due to limits of resolution of the optical pickup) there must be atleast two 0s between any pair of 1's.
  This means that the raw pattern of 1's and 0s has to be translated to get the 8-bit patterns of 1's and 0's that form the bytes of the original data.

* This translation scheme, called <u>**EFM encoding**</u> (eight to fourteen modulation) which is done through a look up table turns the original 8 bits of data into 14 expanded bits that can be represented in the pits and lands on the disc. The reading process reverses this translation

* Fig shows a portion of EFM encoding table

Dec. Value	original bits	Translated bits
0	00000000	01001000100000
1	00000001	10000100000000
2	00000010	10010000100000
3	00000011	10001000100000
4	00000100	01000100000000

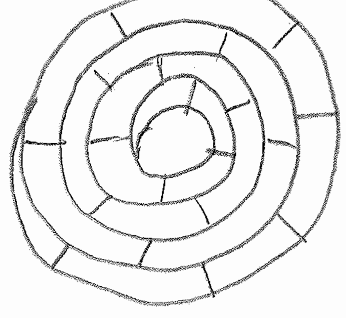
* since 0's are represented by the length of time b/w transition, the disk must be rotated at a precise and constant speed. This effects the CDROM drive's ability.

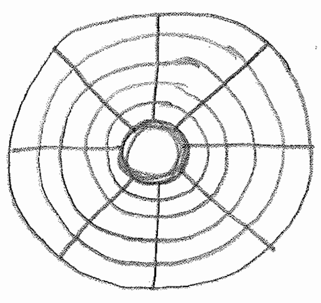
CLV instead of CAV

* Data on a CDROM is stored on a single spiral track. this allows data to be packed as tightly as possible since all the sectors have the same size (whether in the center or at the edge)
* In regular arrangement, the data is packed more densely in the center than in the edge ⇒ space is lost in the edge.
* since reading the data requires that it passes under the optical pickup device at a constant rate, the disc has to spin more slowly when reading the outer edges than when reading towards the center.

  this is why the spiral is a constant linear velocity (CLV) format: as we seek from the center to the edge, we change the rate of rotation of the disc so ~~that~~ the linear speed of the spiral past the pickup device stays the ~~sa~~ same.
* By contrast, the familiar constant angular Velocity (CAV) arrangement (ref fig) with its concentric tracks and pie-shaped sectors, writes data less densely in the outer tracks than in the center tracks.

  we are wasting storage capacity in the outer tracks but have the adv. of being able to spin the disc at the same speed for all positions of the read head.


CLV


CAV

* CLV format is responsible in large part, for the poor seeking performance of CD-ROM drives: There is no straight farword way to jump to a location.
CAV format provides definite track boundaries and timing mark to find the start of a sector.

* On the positive side,
CLV arrangement contributes to the CDROMs large sector storage capacity.
With CAV arrangement, CDROM would have only a little better than ~~haf~~ half its present capacity..

Addressing

* Each second of playing time on a CD is divided into 75 sectors Each sector holds 2KB of data. Each CDROM contains atleast one hour of playing time.
* ie the disc is capable of holding atleast 540,000 KB of data
60 min * 60 sec/min * 75 sectors/sec = 270,000 sectors.
* sectors are addressed by min:sec:sector
eg: 16:22:34.

CDROM strengths and weaknesses.

1. Seek performance

* Very Bad.
* current mag. disk technology has an average random ~~access~~ data access time of about 30msec (combining seek time & rotational delay). But it is about 500msec in case of CDROM.

2. <u>Data Transfer Rate</u>

* not terrible / not great
* A CD ROM drive reads 70 sectors or 150 KB of data per second.
* It is the modest transfer rate, of about 5 times faster than the transfer rate for floppy discs, and and an order of magnitude slower than the rate for good winchester disks.

3. <u>Storage Capacity</u>

* Great.
* Holds more than 600 MB of data.
* Benefit : enables us to build indexes and other support structures that can help overcome some of the limitations associated with CD-ROM's poor performance.

4. <u>Read Only Access.</u>

* CD Rom is a publishing medium, a storage device that cannot be changed after manufacture.
* This provides significant advantages:
  - we never have to worry about updating
  - This not only simplies some of the file structures but also optimizes ~~(of)~~ our index structures and other aspects of file organization.

5. <u>Asymmetric reading and writing.</u>

* For most media, files are written & read using the same comp. system. often reading & writing are both interactive & are therefore constrained by the need to provide ~~a~~ quick response to the user.
* CDROM is different. We create the files to be placed on the disc once ; then we distribute the disc , and is accessed thousands, even millions , of times.

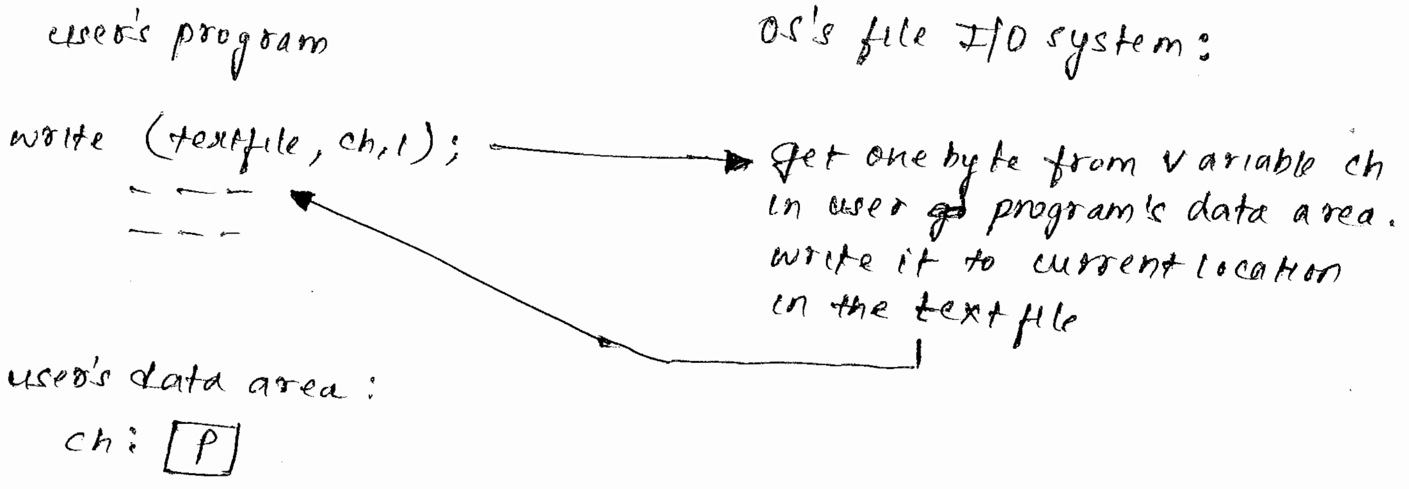
*Conclusion: No need for interaction with the user.

A JOURNEY OF A BYTE

* what happens when the program statement write (textfile, ch, 1) is executed ?

part that takes place in memory.

* The statement calls the OS which overseas the operation
* Filemanager (part of OS that deals with I/O)
  - checks whether the operation is permitted.
  - Locates the physical location where the byte will be stored. (ie drive, cylinder, track, & sector)
  - Finds out whether the sector to locate the 'p' (ie ch) is already in memory. if not, call I/O Buffer.
  - puts 'p' in the I/O buffer.
  - keeps the sector in memory to see if more bytes will be going to the same sector in the file.



user's program

write (textfile, ch, 1);
— — —
— — —

user's data area :
ch: $\boxed{P}$

OS's file I/O system:

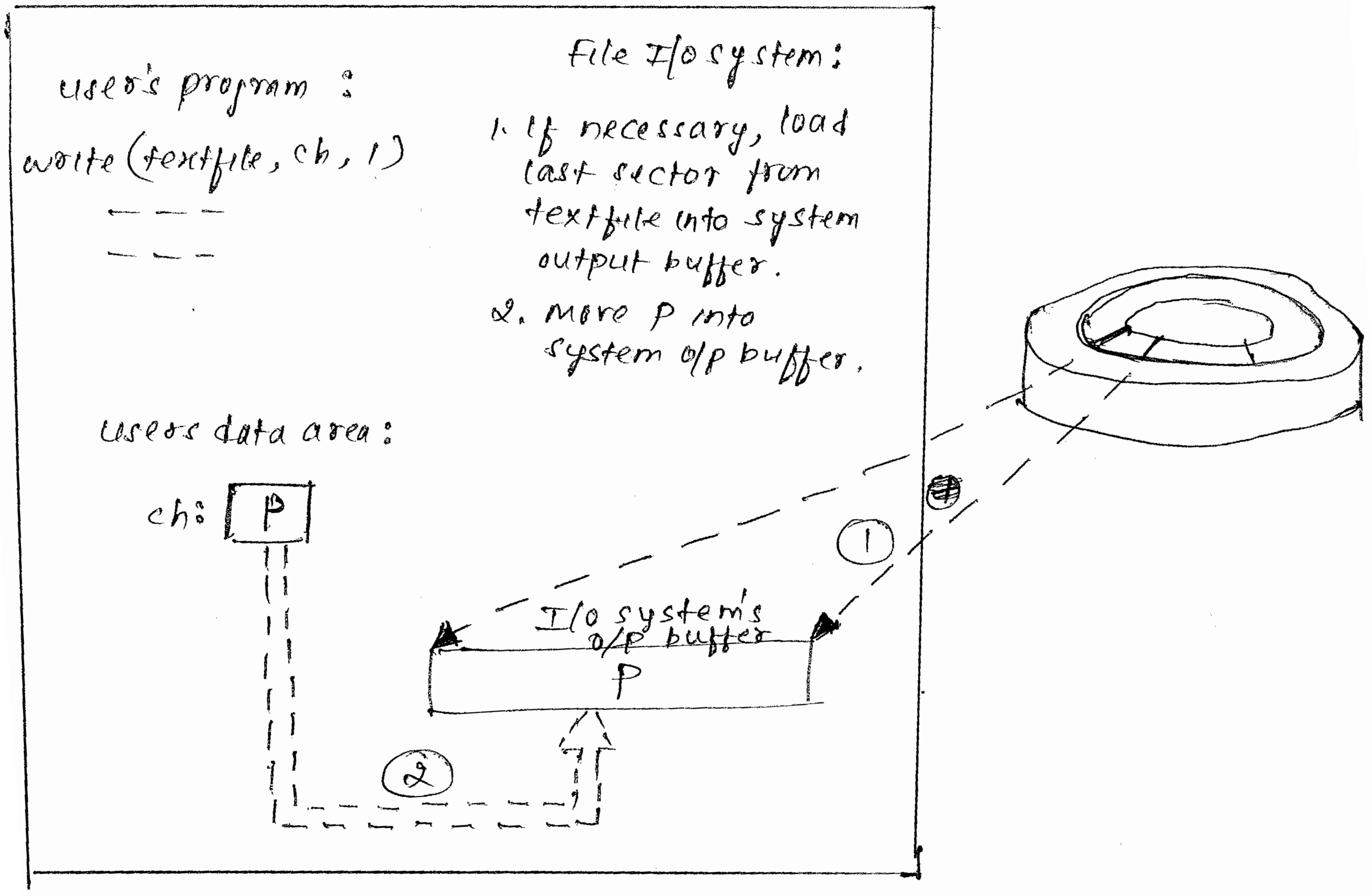
get one byte from variable ch in user ~~go~~ program's data area. write it to current location in the text file

part that takes place outside the memory.

* I/O processor : waits for an external data path to become available (CPU is faster than data-paths ⇒ delays)

* <u>Disk controller</u>

- I/O processor asks the disk controller if the disk drive is available for ~~w~~riting.
- 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.



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

Fig: File manager moves P from the prg's data area to a system o/p 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.

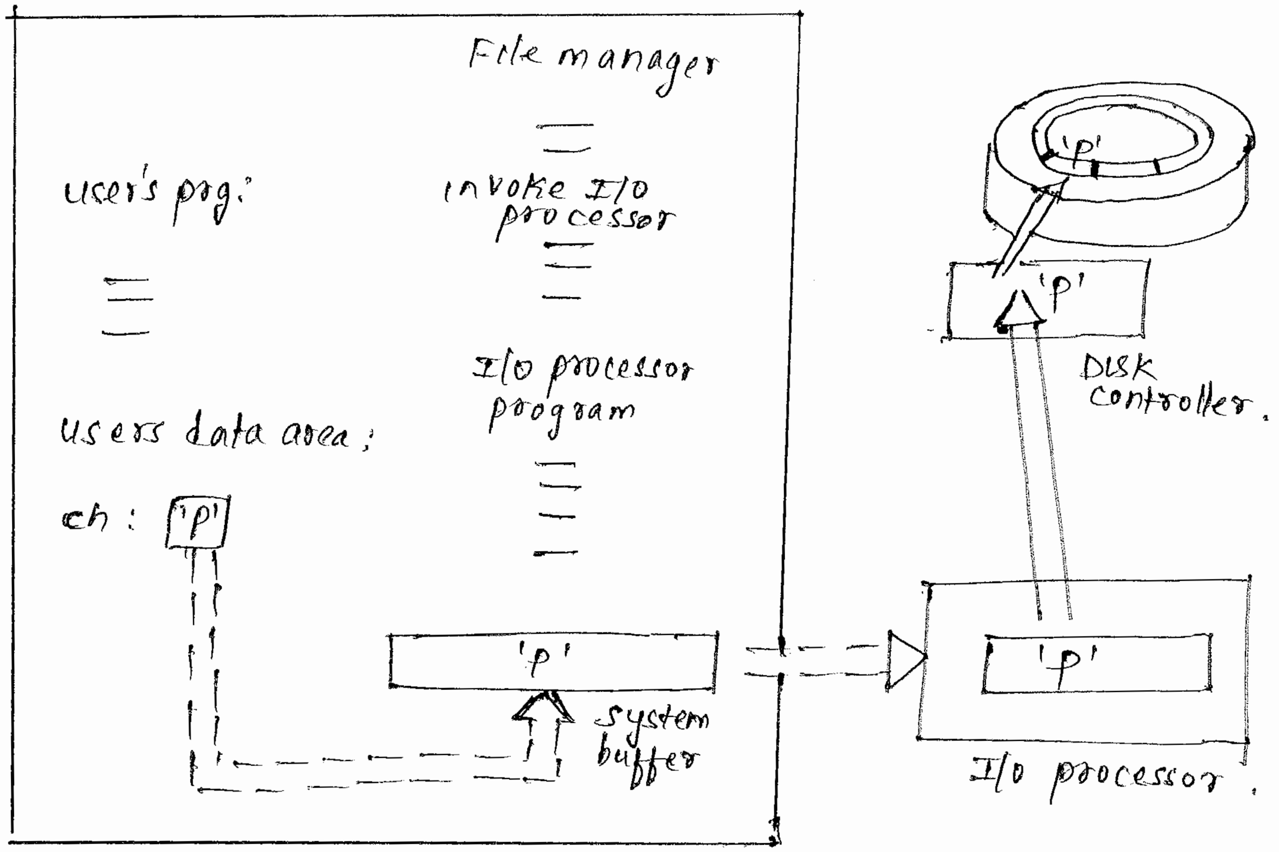
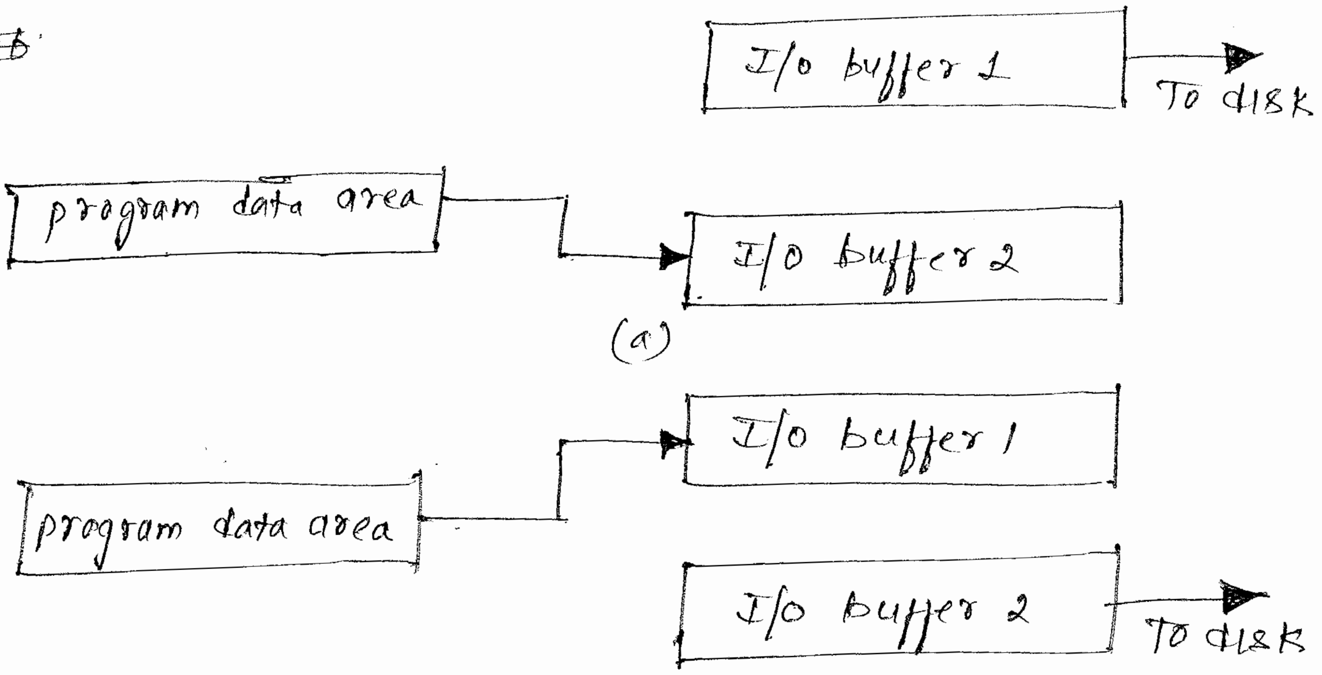


fig: File manager sends the I/O processor instructions in the form of an I/O processor program. the I/O processor gets the data from the system buffer, prepares it for storing on the disk, then sends it to the disk controller, which deposits it on the surface of the disk.

BUFFER MANAGEMENT

* what happens to the data travelling b/w a program's data area and secondary storage?
* use of buffers: Buffering involves working with a large chunk of data in memory so ~~that~~ the number of accesses to secondary storage can be reduced.

* ~~Doub~~



(b)

(a) contents of system I/O buffer 1 are sent to disk while I/O buffer 2 is ~~bef~~ being filled.
(b) contents of buffer 2 are sent to disk while I/O buffer 1 is being filled.

Buffer pooling

* when a system buffer is needed, it is taken from a pool of available buffers and used.
* when the system receives a request to read a certain sectors or block, it looks to see if one of its buffers already contains ~~that~~ sector or block. If no buffer contains it, the system finds from its pool of buffers one that is not currently in use & loads the sector/block into it.

Move mode and Locate mode

* <u>move mode</u> involves moving chunks of data from one place in m/m to another before they can be accessed.
  Data is ~~mo~~ copied from a system buffer to a program buffer & vice versa.
  Disadv: Time taken to perform the move.

Buffer Bottlenecks.

* Assume that the system has a single buffer and is performing both input and output on one ~~characte~~ character at a time alternatively.
* In this case, the sector containing the character to be read is constantly over written by the sector containing the spot where the character will be written, & vice versa.
* In such a case, the system needs more than one buffer. atleast, one for input & other for output.
* moving data to and from disk is very slow and programs may become I/O bound. ∴ we need to find better strategies to avoid this problem.

Buffering strategies

* Some Buffering strategies are
  - multiple Buffering
    - Double Buffering
    - Buffer pooling
  - move mode and locate mode.
  - scatter / gather I/O.

Multiple Buffering.

Double Buffering

* System with Double buffering have two buffers. method of swapping the roles of two buffers after each output (or input) operation is called double buffering.
* Double buffering allows the OS to operate on one buffer while the other buffer is being [illegible]

~~* Two ways~~ .

* Locate mode has two techniques.

- If file manager can perform I/O directly ~~from~~ b/w sec. storage & prog data area, no extra move is necessary

- Alternatively, file manager could use system buffers to handle all I/O but provide the prog with the locations, using pointer variables, of the system buffers.

* locate mode eliminates need to transfer data b/w an I/O buffer & a prog buffer.

Scatter / Gather I/O

* Scatter input : with scatter i/p, a single read call identifies not one, but a collection of buffers into which data from a single block is to be scattered.

* ~~Scatter~~ gather output: converse of scatter input. with gather o/p, several buffers can be gathered & written with a single write call.

Ashok Kumar K
VIVEKANANDA INSTITUTE OF TECHNOLOGY

UNIT 3 | ORGANIZATION OF FILES FOR PERFORMANCE, INDEXING

Syllabus

Chapter 3A: ORGANISING FILES FOR PERFORMANCE

* Data compression
* Reclaiming spaces in files
* Finding Things quickly: An introduction to internal sorting and Binary searching
* Key Sorting.

Chapter 3B: INDEXING

* what is an index?
* A simple index for entry-sequenced files.
* using Template classes in C++ for object I/O.
* object oriented support for indexed, ~~Entity~~ Entry-sequenced files of data objects.
* indexes that are too large to hold in memory.
* indexing to provide access by multiple keys.
* Retrieval using combinations of secondary keys.
* improving the secondary index structure: inverted lists.
* Selective indexes.
* Binding.

– 7 Hours

Ashok Kumar K

VIVEKANANDA INSTITUTE OF TECHNOLOGY

chapter 3A:

ORGANIZING FILES FOR PERFORMANCE

DATA COMPRESSION

* Why do we want to make files smaller?

Answer: smaller files

i.) use less storage, resulting in cost savings.

ii.) can be transmitted faster, decreasing access time, or, alternatively, allowing the same access time but with a lower and cheaper bandwidth.

iii.) can be processed faster sequentially.

* Definition:

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

Few Data compression techniques are discussed here.

Using a Different Notation (Redundancy reduction)

* Fixed length fields are good candidates for compression. In prev. unit, the person file had a fixed length field 'state' which required 2 ASCII bytes. Was that really necessary? How many bits are really needed for this field?

Since there are only 50 states, we could represent all possible states with only 6 bits. Thus saving one byte per state field. (or 50%)

* 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 (Encoding/Decoding modules)

Suppressing Repeating Sequences (redundancy reduction)

* When the data is represented in a sparse array, we can use a type of compression called Run Length encoding.

* Algorithm (or procedure)

  i.) Read through the array in sequence except where the same value occurs more than once in succession.

  ii.) When the same value occurs more than once, substitute the following 3 bytes in order.
  - → Special run length code indicator
  - → Values that is repeated
  - → Number of time the value is repeated.

Example: Encode the following sequence of hexadecimal byte values. chose 0xff as ~~length~~ run-length indicator.

22 23 24 24 24 24 24 24 24 25 26 26 26 26 26 26 25 24

Soln: Resulting sequence is:

22 23 ff 24 07 25 ff 26 06 25 24

* Disadvantage

  i.) No guarantee that space will be saved.

Assigning variable-length codes.

* Principle:

  Assign short codes to the most frequent occuring values and long codes to the least frequent ones.

* The code size can not be fully optimized as one wants codes to occur in succession, without delimiters between them, & still be recognized.

* This is the principle used in morse code. As well, it is used in Huffman coding.

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

Irreversible compression technique

* It is based on the assumption that some information can be sacrificed.
* Ex: shrinking a raster image from 400-by-400 pixels to 100-by-100 pixels. The new image contains 1 pixel for every 16 pixels in the original image, & there is no way to determine what the original pixels were from the one new pixel.
* In data files, irreversible compression is seldom used, however they are used in image and speech processing.

Compression in Unix

* System V unix has routines called pack & unpack, which uses huffman codes on a byte-by-byte basis.
  Typically pack achieves 25 to 40% reduction on text files, but less on binary files that have a more uniform distribution of byte values.
  unpack appends a .z to the end of file it has compressed.
* Berkely unix has routines called compress and uncompress, which uses effective dynamic method called Lempel-Ziv.
  compress appends a .Z to the end of the file it has compressed.

RECLAIMING SPACE IN FILES

* modifications can take any one of 3 forms
  - → Record addition
  - → Record updating
  - → Record deleting.

note: $\begin{pmatrix}\text{record}\\ \text{updating}\end{pmatrix} = \begin{pmatrix}\text{record}\\ \text{deletion}\end{pmatrix} + \begin{pmatrix}\text{record}\\ \text{addition}\end{pmatrix}$

* Here, we focus on record deletion.

Record Deletion and storage compaction

* How to indicate the records as **deleted?**

 simple approach is to place a **special mark** in each deleted record.

eg:

```
Mary|Ames|123 london
John|James|50 USA .....
Folk|michael|75 UK......
```

fig(a): Before the second record is marked as deleted.

```
mary|Ames|123 london
*|hn|James|50 USA......
Folk|michael|75 UK.....
```

fig(b): After the second record is marked as deleted.

* **storage compaction** makes files smaller by looking for places in a file where there is no data at all and recovering this space (or)
 Reusing the space from the record is called storage compaction.

* After deleted records ~~are~~ have accumulated for some time, a special program is used to **reconstruct** the file with all deleted ~~approaches~~ records squeezed out as shown

```
Mary|Ames|123 london
Folk|Michael|75 UK.....
```

* storage compaction can be used with both fixed and

Deleting fixed length records for reclaiming space dynamically

* In some applications, it is necessary to reclaim space immediately. In general,
* To provide a mechanism for record deletion with subsequent reutilization of freed space, we need to be able to guarantee two things
  → That deleted records are marked in some special way.
  → That we can find the space that deleted records once occupied, so we can reuse that space when we add records.
* To make record reuse happen more quickly, we need
  → A way to know immediately if there are empty slots in the file,
  → A way to jump directly to one of those slots if they exist.

Solution: use linked lists in the form of a stack.
RRN plays the role of a pointer.

linked list as stack.

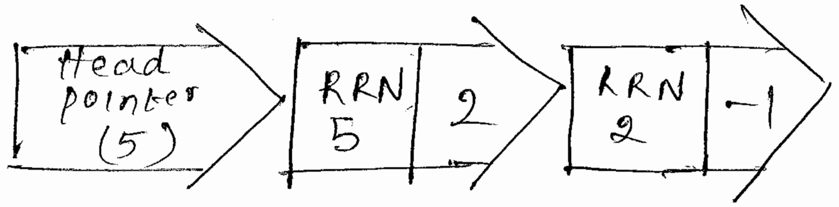


fig: When Record with RRN 5 & 2 are deleted

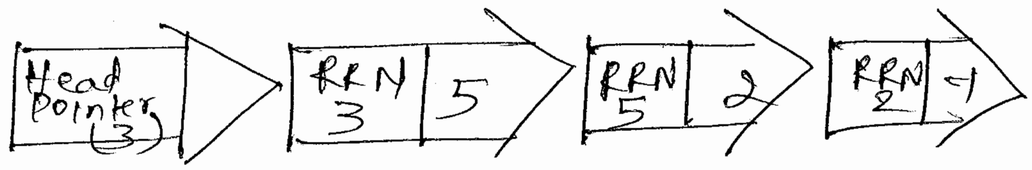


fig: When record with rrn 3 is also deleted

fig: avail list (list of ~~free~~ deleted records).

Linking and Stacking Deleted records.

* Below figure shows sample file showing linked lists of deleted records.
  < first ~~first~~ field of deleted record → marked with asterisk.
  Second field of deleted record → link to next record on the avail list

list head (first available record) →5

0	1	2	3	4	5	6
Edwards....	Bates....	Wills....	* -1	master....	* 3	John...

(a) After deletion of records 3 and 5 in that order.

list head → 1

0	1	2	3	4	5	6
Edwards...	* 5	Wills..	* -1	master.	* 3	John...

(b) After deleting records 3, 5, & 1 in that order.

list head → -1

0	1	2	3	4	5	6
Edwards...	1st new record	Wills...	3rd new record	master...	1st new record	John...

(c) After insertion of three new records.

Deleting Variable length records

* 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 records from the avail list when we are ready to use them.

An avail list of variable length records.

* Same ideas as in fixed length rec. but few modifications.
* 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 for links, we

Illustration.

* fig shows sample file illustrating variable length record deletion.

HEAD.FIRST_AVAIL : -1

~~10~~ '

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

fig(a): original sample file stored in variable length format with byte count

~~HEAD.AVAIL~~ : .

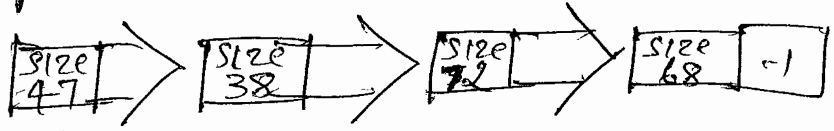
HEAD.FIRST-AVAIL : 21

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

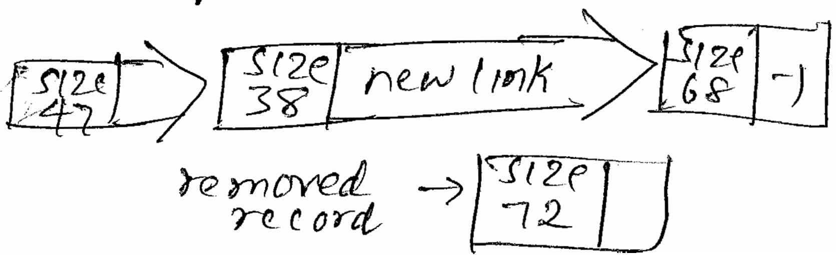
fig(b): Sample file after deletion of second record

Adding and removing records.

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



(a) Before removal



Here the new record to be added is assumed to be of 55 bytes.

Storage Fragmentation

* **Internal fragmentation** - Wasted space within a record is called internal fragmentation.
* Fixed length record structures often result in internal fragmentation.
* Variable-length records do not suffer from internal fragmentation. However external fragmentation is not avoided

  external fragmentation - Form of fragmentation that occurs in a file when there is unused space outside or between individual records.
* Three ways to deal with external fragmentation
  - → storage compaction
  - → Coalescing the holes
  - → use a clever placement strategy

Placement Strategies

* A **placement strategy** is a mechanism for selecting the space on the avail list that is to be used to hold a new record added to the file.
* **First fit placement strategy**: Accept the first available record slot that can accomodate the new record.
  (or large enough to hold)
* **Best fit placement strategy**: Finds the available record slot that is closest in size to what is needed to hold the new record.
* **Worst fit placement strategy**: selects the largest available record slot, regardless of how small the new record is

notes on

* **First fit strategy**

→ least possible amount of work is expended.

→ We develop more orderly approach for placing records on the avail list

* **Best fit strategy**

→ Avail list should ~~be~~ in ascending order in size.

→ We should search through atleast part of the avail list not only when we get records from the list, but also when we put newly deleted records on the list.
(ie extra processing time)

→ Results in external fragmentation.

* **Worst fit strategy**

→ Avail list should be in descending order in size.

→ Decreases the likelihood of external fragmentation.

→ procedure for removing records can be simplified ~~as~~ so it looks only at the first element. If first record slot is not large enough to do the job, none of the others will be.

* **Remarks about placement strategies.**

→ placement strategies only apply to variable length records.

→ If space is lost due to internal fragmentation, the choice is b/w first fit & best fit. A worst fit strategy truly makes internal fragmentation worse.

→ If the space is lost due to external fragmentation, one should give careful consideration to a worst fit strategy.

FINDING THINGS QUICKLY: AN INTRODUCTION TO INTERNAL SORTING AND BINARY SEARCHING

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

note: often sorting is the first step to searching efficiently. We develop approaches for searching & sorting that minimizes no. of disk accesses (or seeks)

Finding things in simple field and record files.

* So far, in case of fixed length records, the only way we have to retrieve or find records quickly is by using their RRN.
* without a RRN or in case of variable length records, only way so far to do it is by using a sequential search which is very inefficient method.
* We are interested in more efficient ways to retrieve records based on their key value.

Search by Guessing: Binary search.

* suppose we are looking for a record for Jane kelly in a file of 1000 fixed length records.
* Assume the file is sorted in ascending order based on the key (name)
* We start by comparing KELLY JANE (canonical form of search key) with the middle key in the file, which is the key whose RRN is 500.
* Result of the comparision tells us which half of the file contains Jane kelly's record.

* next we compare KELLY JANE with the middle key among records in the selected half of the file to find out which quarter of the file jane kelly's record is in.
* This process is repeated until either Jane kelly's record is found or we have narrowed the no. of potential records to zero.
* This kind of searching is called as Binary searching.
* Binary search Algorithm.

```
int BinarySearch (FixedRecordFile &file,
                  RecordType &obj, KeyType & key)

// if key found, obj contains corresponding record, 1 returned
// if key not found, 0 returned.
{ int low=0; int high=file.numRecs() -1;
  while (low<=high)
  { int guess=(high-low)/2;
    file.ReadByRRN (obj, guess);
    if (obj.key() ==key) return 1;
    if (obj.key() <key) high= guess-1 ;
    else low= guess +1 ;
  }
  return 0;
}
```

* classes and methods that must be implemented to support binary search algorithm

```
class KeyType
{ public:
   int operator ==(KeyType &);
   int operator < (KeyType &);
};

class RecType
{ public:
  KeyType Key();
```

```
class FixedRecordFile
{ public:
   int NumRecs();
   int ReadByRRN (RecType &record,
                        int RRN);
};
```

Binary search	Sequential Search
* Takes $O(\log_2 n)$ comparisions $n \rightarrow$ no. of records.	* Takes $O(n)$ comparisions
* when the filesize is doubled, it adds only one more guess to our worst case	* When the no. of records (file size) is doubled, it doubles the no. of comparisions required.
* File must be sorted	* no need of sorting.

Sorting a Disk file in memory

* if the entire contents of the file can be held in memory, we can perform an **internal sort** (sorting in memory) which is **very efficient.**
* But, most often, the file does not hold entirely in memory. In this case, any sorting algorithm will require large number of seeks. Solutions has to be found for this.

Limitations of binary search and internal sorting

problem 1: Binary search requires more than one or two accesses.

* when we access records by RRN rather than by key, we are able to retrieve a record by single access.
* Ideally, we would like to approach RRN retrieval performance while still maintaining the advantages of access by key. (indexing)

problem 2: Keeping a file ~~stored~~ sorted is very expensive.

* In addition to searching for the right location for the insert, once this location is found, we have to shift records to open up the space for insertion.

problem 3: An internal sort works only on small files.

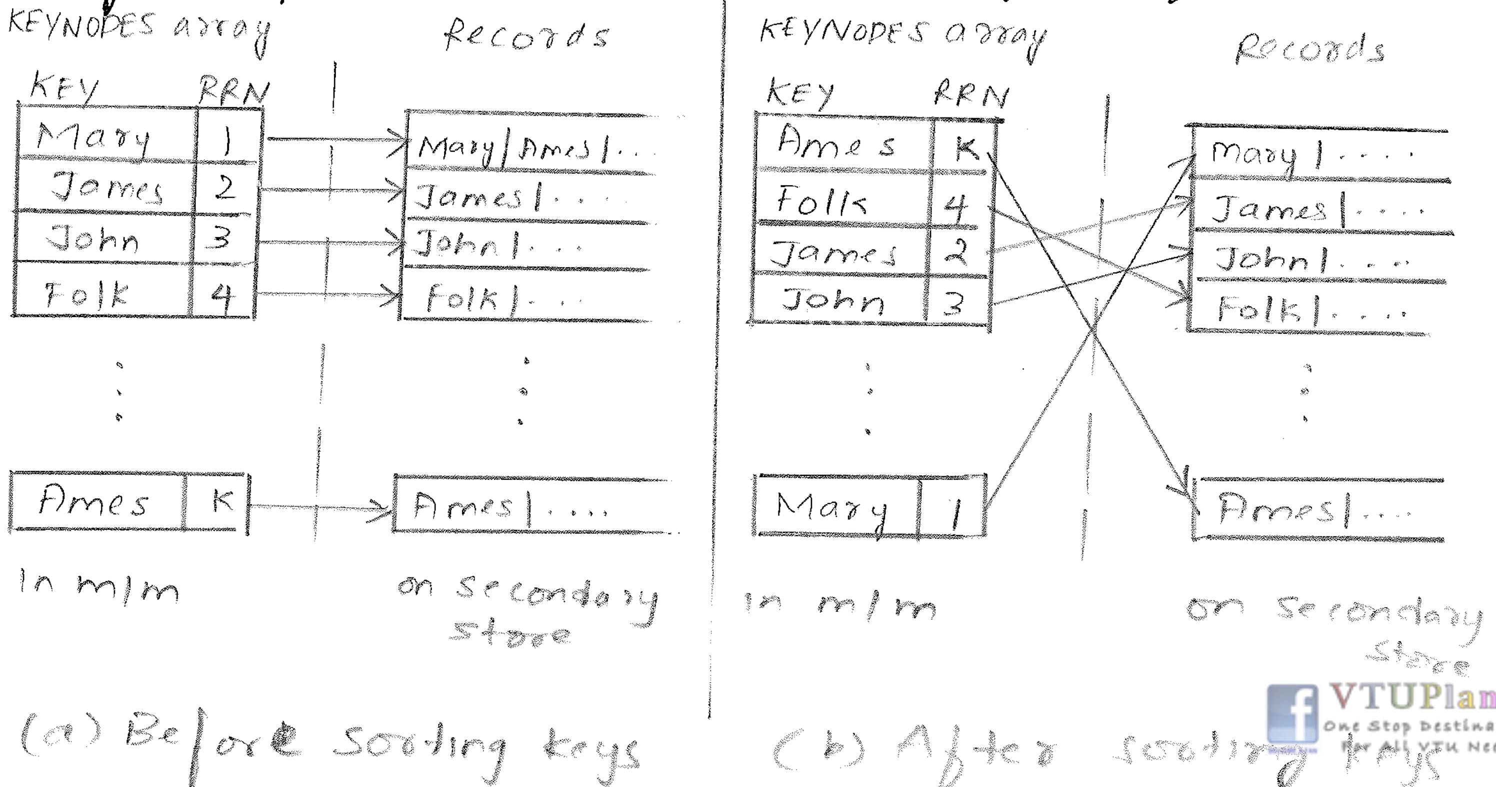
KEY SORTING

* It is a method of sorting a file that does not require holding the entire file in memory.
  Only the **Keys** are held in memory, along with the **pointers** that tie these keys to the records in the file from which they are extracted.
  These keys are sorted, and the sorted list of keys is used to construct a new version of the file that has the records in sorted order.
* **Adv**: requires less m/m than a internal sort (m/m sort)
* **Disadv**: process of constructing a new file requires a lot of seeking for records.
* Two differences in keysorting from internal sorting.
  → Rather than read an entire record into a m/m array, we simply read each record into temporary buffer, extract the key, then discard it.
  → when we are writing the records out in sorted order, we have to read them in a second time, since they are not all stored in m/m.

* Algorithm for Keysort

```
int KeySort ( FixedRecordFile &infile , char * outfilename )
{ RecType obj ;
   KeyRRN *KEYNODES = new KeyRRN [infile.numrecs()];
   for(int i=0 ; i<infile.numrecs() ; i++ )
    { infile . ReadByRRN (obj , i);
      KEYNODES[i] = KeyRRN ( obj.Key() , i);
    }
   Sort ( KEYNODES , infile.Numrecs());
   FixedRecordFile outfile ;
   outfile.create ( outfilename);
   for (int j=0 ; j< infile.numrecs(); j++ )
    {
      infile.ReadByRRN ( obj , KEYNODES[j].RRN);
      outfile.Append (obj);
    }
   return -1;
}
```

* Fig: conceptual view of KEYNODES array and file



Limitations of the keysort method

* writing the records in sorted order requires as many random seeks as there are records.
* Since writing is interspersed with reading, ~~writing~~ reading also requires as many seeks as there are records.

Solution: Why bother to write the file back?

* Instead of writing out a sorted version of a file, we write out a copy of the array of canonical key nodes. (ie writing out the contents of our KEYNODES[] array) This will be index to the original file. Relationship b/w two files is shown in the figure.

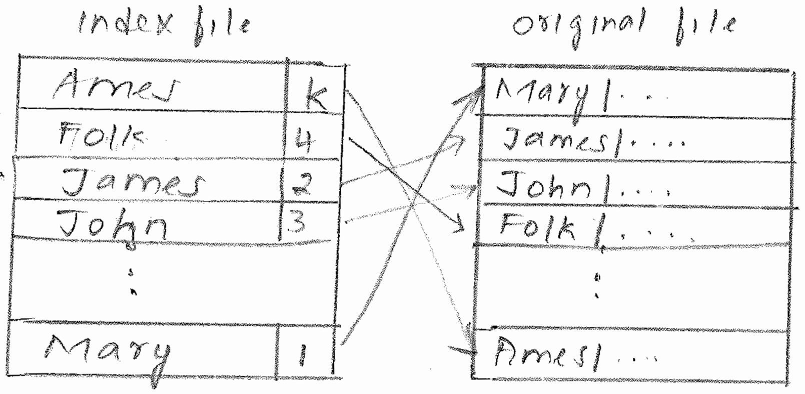


fig: relⁿ b/w index file & data file.

Pinned Records

* A record is **pinned** when there are other records or file structures (in same file or different file) that refer to it by its physical location.
* Pinned records **cannot be moved**, because these references no longer lead to the record; they become dangling pointer.
* Use of pinned records in a file **make sorting more difficult or sometimes impossible.**

Soln: use index file to keep the sorted ~~our~~ order of the records while keeping the data file in its original order.

CHAPTER 3B:

INDEXING

WHAT IS AN INDEX?

* An index is a table containing a list of keys associated with a reference field pointing to the record where the information referenced by the key can be found.

(or) An index is a tool for finding records in a file. It consists of

→ Key field on which the index is searched

→ reference field that tells where to find the data file record associated with particular key.

* An index lets you impose order on a file without rearranging the file.
* You can have different indexes for the same data: multiple access paths.
* Indexing gives us keyed access to variable-length record files

A SIMPLE INDEX FOR ENTRY-SEQUENCED FILES

* Suppose that you are looking at a collection of recordings with the following information about each of them.

  - Identification number
  - ~~File~~ Title
  - Composer or Composers
  - Artist or Artists
  - Label (publisher)

* We chose to organize the file as variable length record with a size field preceeding each record. The fields within each record are also of variable-length but are separated by delimiters.

* We form a **primary key** by concatenating the record company label code and the record's ID number. This should form a unique identifier.

* fig: Contents of sample recording file.

Record address	Label	ID number	Title	Composer(s)	Artist(s)
17	LON	2312	Romeo & Juliet	neil	Ames
62	RCA	2626	Touch stone	John	David
117	WAR	23699	Good news	Folk	Bill
152	ANG	3795	Nebraska	Mary	Wills.
241	DG	18807	Symphony no. 9	Beethoven	Martin

* In order to provide rapid keyed access, we build a simple index with a key field assocciated with a reference field which provides the address of the first byte of the corresponding data record.

* fig: index of the sample recording file.

index.

Key	Ref field
ANG 3795	~~3795~~ 152
LON 2312	~~2312~~ 17
RCA 2626	~~2626~~ 62
WAR 23699	~~23699~~ 117

Recording file

Addr of record	Actual data record
17	LON|2312|Romeo& Juliet|....
62	RCA|2626|Touchstone|...
117	WAR|23699|Good news|....
152	ANG|3795|Nebraska|...

* The **index may be sorted** while the **file does not have to be**. This means that the data file may be entry sequenced ie the record occur in the order they are entered in the file.

notes on index

* The index is easier to use than the data file because,
  - → It uses fixed length records.
  - → likely to be much smaller than the data file.
* By requiring fixed length records in the index file, we impose a limit on size of the primary key field. This could cause problems.
* The index could carry more other information than the key and reference fields. (eg: length of each data record)

OBJECT ORIENTED SUPPORT FOR INDEXED, ENTRY SEQUENCED FILES OF DATA OBJECTS

Operations required to maintain an indexed file

* Assumption: Index is small enough to be held in memory.

  Some operations used to find things by means of the index include the following.

→ create the original empty index and data files.
→ load index file into m/m before using it.
→ Rewrite the index file from m/m after using it.
→ Add data records to data file
→ Delete records from data file
→ update records in data file.
→ update the index to reflect changes in the data files.

Creating the Files

* Two files must be created: a data file to hold the data objects and an index file to hold the primary key index.

Loading the index into memory

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

Rewriting the index file from memory.

* What happens if the index changed but its rewriting does not takes place or takes place incompletely? (ie index on the disk is out of date)

→ use a mechanism for indicating whether or not the index is out of date.

→ Have a procedure that reconstructs the index from the data file in case it is out of date.

Record Addition.

* Adding a new record to data file requires that we also add an entry to the index.
* In data file, record can be added anywhere. However the byte offset of new record should be saved.
* Since the index is kept in sorted order by key, insertion of new index entry probabaly requires some rearrangement of the index.

  We have to shift all the records that belong after the one we are inserting to open up space for the new record.

  However, this operation is not costly (no file access) as it is performed in memory.

Record Deletion.

* Prev chap explained no. of approaches to deleting records. These approaches can be used.
* Index record corresponding to data record ~~bet~~ being deleted must also be deleted. Once again, since this deletion

Record updating

* Record updating falls in two categories:

→ **The update changes the value of the key field**

* Here, both index and data file may need to be reordered.
* conceptually, the easiest way to think of this kind of change is as a deletion followed by an insertion. (but the user needs not know about this)

→ **the update does not affect the key field**

* Does not require rearrangement of the index file but may well involve in reordering of data file.
* If the record size is unchanged or decreased by the update, the record can be written directly into its old space.
* But, if the record size is increased by the update, a new slot for the record will have to be found. Again, the delete/insert approach to maintaining the index can be used.

INDEXES THAT ARE TOO LARGE TO HOLD IN MEMORY

* If the index is too large, ~~to be~~ then index access and maintainence must be done on secondary storage

Disadvantages

→ Binary searching of index requires several seeks rather than being performed at m/m speed.

→ Index rearrangement (due to record addition/deletion) requires shifting or sorting records on secondary storage, which is extremely time consuming.

Solution: using

→ Hashed organisation - if access speed is a top priority.

→ Tree structured, or multilevel index (like B-tree) - if you need the flexibility of both keyed access and

* Adv of simple indexes on secondary storage over the use of datafile sorted by key are:

-> A simple index allow use of binary search in a variable-length record file.

-> If the index entries are substantially smaller than the data file records, sorting and maintaining the index can be less expensive than the data file.

-> If there are pinned records in the datafile, the use of an index lets us rearrange the keys without moving the data records.

-> provides multiple views of a datafile.

INDEXING TO PROVIDE ACCESS BY MULTIPLE KEYS

* So far, our index allows only key access. ie you can retrieve record RCA2626, but you cannot retrieve a recording of John's touchstone.

* We could build catalog for our record collection consisting of entries for album title, composer, and the artist. These fields are secondary key fields.

* Fig shows index file that relates composer to label ID.

composer index

secondary key	primary key
neil	LON3212
John	RCA2626
Folk	WAR23699
mary	ANG3795

fig: secondary key index

* Although it would be possible to relate a secondary key to actual byte offset, this is usually not done. instead we relate the secondary key to a primary key which ~~w~~ then will point to the actual byte offset

~~Note~~ :

Record Addition

* When a secondary index is used, adding a record involves updating the data file, the primary index, and the secondary index.
  Secondary index update is similar to primary index update. (ie either records must be shifted, or a vector of pointers to structures need to be rearranged,
* ||^r to primary indexes, cost of doing this greatly decreases if the secondary indexes can be read into m/m and changed there.
* Secondary keys (or key field in sec. index file) are stored in canonical form (all of composer's name in capitals)

* fig shows sec. key index organized by recording title.

Title index

secondary key	primary key
Romeo Juliet	LON2312
Touchstone	RCA2626
Touchstone	DG18807
Touchstone	COL31809
Good news	WAR23699
Nebraska	ANG3795

* sec. keys are held to a fixed length, ie sometimes they are truncated.
* one imp. difference b/w sec. index & primary index is that a secondary index can contain duplicate keys (grouped together) and the primary index could'nt. (refer fig)

Record Deletion.

* Removing a record from data file means removing the corresponding entry in primary index and all the entries in secondary indexes that refer to this

* **problem**: like primary index, the secondary indexes are maintained in sorted order by key. Deleting an entry would involve rearranging the remaining entries to close up the space left open by deletion.

* This delete-all-references approach is advisable if the secondary index referenced the data file directly. But since secondary keys were made to point at primary ones, we can eliminate modify and rearrange the secondary key index on record deletion
⇒ Searches starting from secondary key index that lead to a deleted record are cought when we consult the primary key index.

* **Disadv**: Deleted records take up space in the secondary index files.
soln: B-tree (allows for deletion without having to rearrange a lot of records)

Record updating

* There are 3 possible situations

i.) **update changes the secondary key:**
We may have to rearrange the secondary key index so it stays in sorted order. <Relatively expensive operation>

ii.) **update changes the primary key:**
Has large impact (or changes) on primary key index but often requires that we update only the affected reference field (label ID) in all sec. indexes

iii.) **update confined to other fields:**
No changes necessary to primary nor secondary indexes.

RETRIEVAL USING COMBINATIONS OF SECONDARY KEYS

* ~~Using~~ one imp applications of secondary keys involves using two or more of them in combination to retrieve special subsets of records from the data file.
* with sec. keys, we can search for things like
  - → recording with label ID COL38358
  - → recordings of John's work
  - → recordings titled 'RomeoJuliet'
* More importantly, we can use combination of secondary keys <eg find all recordings of Beethoven's Symphony no. 9 >
* without the use of secondary indexes, this request requires a very expensive sequential search through the entire file. With the aid of secondary indexes, responding to this request is simple and quick.

find all data records with:
composer = 'BEETHOVEN' and title = 'SYMPHONY no. 9'

composer index yields,	Title index yields	matched list
ANG3795 →	ANG3795 →	ANG3795
DG139201	COL31809	DG18807
DG18807 →	DG18807 →	
RCA2626		

then we can retrieve the records;

ANG|3795|symphony no. 9|Beethoven | Mills
DG |18807| symphony no.9| Beethoven | Martin.

IMPROVING THE SECONDARY INDEX STRUCTURE : INVERTED LISTS.

* Secondary index structures results in two distinct difficulties :

→ We have to rearrange the index file everytime a new record is added to the file, ~~#~~ even if the new record is for an existing secondary key.

→ If there are duplicate secondary keys, the sec. key field is repeated for each entry. ⇒ space is wasted larger index files are less likely to fit in m/m.

* There are two solutions for this.

Solution 1

* change the secondary index structure so it associates an array of references with each secondary key

eg :

BEETHOVEN ANG3795 DG139201 DG18807 RCA2626

* Fig shows secondary key index containing space for multiple references for each sec. key.

Revised composer index

Secondary key	set of primary key references
BEETHOVEN	ANG3795 DG139201 DG18807 RCA2626
NEIL	LON3212
JOHN	RCA 2626
FOLK	WAR23699
MARY	ANG0102

Adv

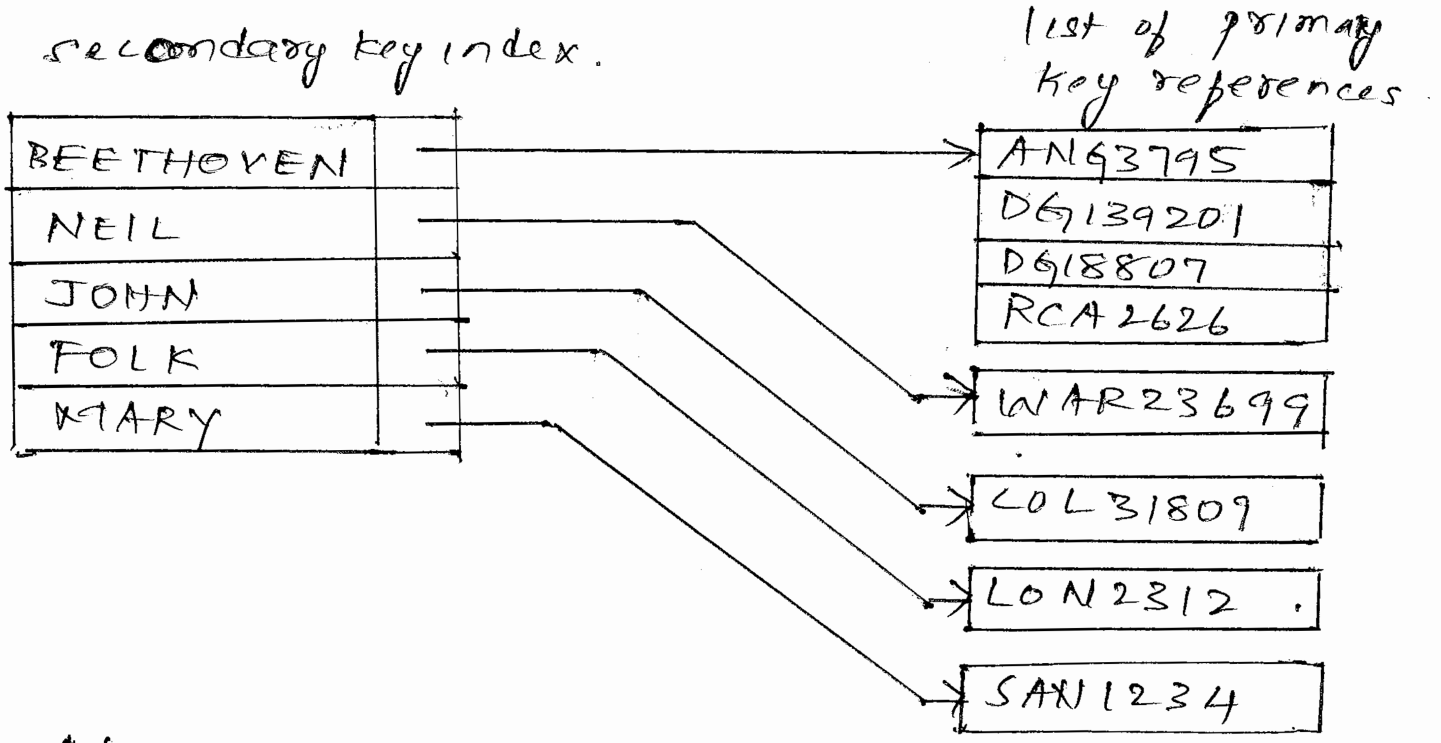
* Avoids the need to rearrange the

Disadv

* may restrict the no. of references that can be associated with each sec. key.

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

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



Advantages

* secondary Index file needs to be rearranged only when new record is added (ie when new composer's name is added or existing composer's name is change)
* Rearranging is faster, since there are fewer records and each record is smaller.

* There is less need for sorting. Therefore we can keep secondary index file on disk.
* Label ID list file is entry sequenced. ie primary index never needs to be sorted.
* Space from deleted primary index records can easily be reused.

Disadvantage

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

SELECTIVE INDEXES

* A selective index contains keys for only a portion of the records in the data file.
  Such an index provides the user with a view of a specific subset of the file's records.

BINDING

* Question: At what time is the key bound to the physical address of its associated record?
* Answer so far:

→ The Binding of our primary keys takes place at construction time.
Adv: faster Access.
Disadv: Reorganisation of data files must result in modifications to all bound index files.

→ Binding of our secondary keys ~~pla~~ takes place at the time ~~time~~ they are used.
Adv: safer.

* Tradeoff in binding decisions :

→ Tight binding (construction time binding ie during preparation of data file) is preferable when

- Data file is static or nearly static, requiring little or no, adding, deleting, or updating
- Rapid performance during actual retrieval is a high priority.

note: In tight binding, indexes contains explicit references to the associated physical data record.

→ postponing binding as long as possible is simpler and safer when the data file requires a lot of adding, deleting, and updating.

note: Here the connection b/w a key and a particular physical record is postponed until the record is retrieved in the course of program execution.

Ashok Kumar K

VIVEKANANDA INSTITUTE OF TECHNOLOGY

UNIT 5

MULTILEVEL INDEXING, AND B-TREES

Syllabus

* Invention of B-tree
* Statement of the problem
* Indexing with binary search trees
* Multilevel indexing
* B-trees
* Example of creating a B-tree
* An object Oriented representation of B-trees
* B-tree methods
* Nomenclature.
* Formal definition of B-tree properties
* Worst case search depth.
* Deletion, merging, and Redistribution.
* Redistribution during insertion.
* B* Trees
* Buffering of pages; Virtual B-Trees
* Variable length records & keys

7 Hours

Ashok Kumar K
VIVEKANANDA INSTITUTE OF TECHNOLOGY

* R. Bayer and ~~E. Mcg~~. E. McCreight invented B-trees - standard organisation for indexes in a database system.
It provides rapid access to the data with minimal overhead cost.

Statement of the problem.

* When indexes grow too large, they have to be sorted on secondary storage.
* However, there are two fundamental problems associated with keeping an ~~indexes~~ index on secondary storage:
→ searching the index must be faster than binary searching
→ insertion & deletion must be as fast as search.

~~Indexing with Binary Sea~~

INDEXING WITH BINARY SEARCH TREES

Negative Aspects.

* Given a sorted list (fig a), it can be expressed in a Binary search Tree representation. (fig b)
Binary search tree can be constructed as a linked structure (fig c) using elementary datastructure techniques

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

fig (a).

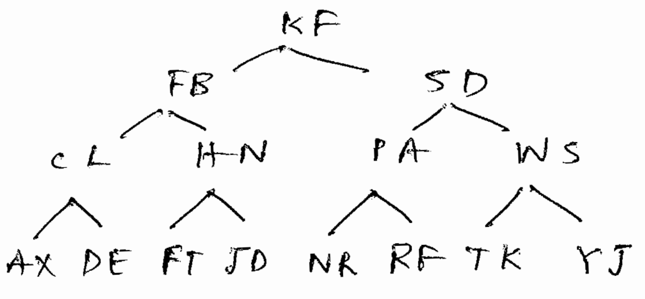


fig (b).

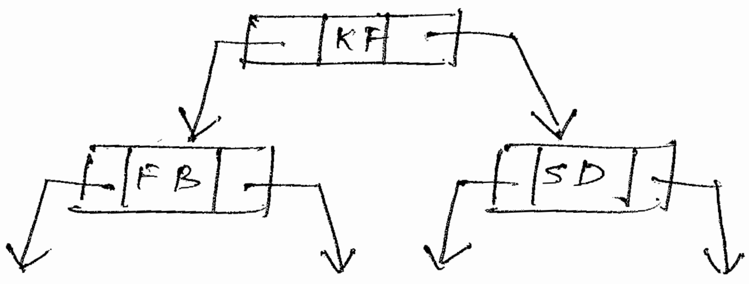


fig (c)

* However there are two problems with binary search trees:
→ They are not fast enough for disk resident indexing.
→ There is no effective strategy of balancing the tree.
* We will look at 2 solutions:
  1. AVL Trees.
  2. Paged Binary Trees.

Positive Aspects.

* This tree structure gives us an important new capability: we no longer have to sort the file to perform a binary search.

~~Illustration.~~
* To add a new key, we simply link it to the appropriate ~~new node~~ leaf node.

illustration.
* Note that records in the file shown (fig a) appear in random rather than sorted order.
* ~~TO~~ To add a new key LV, we need only link it to the appropriate leaf node to create a tree that provides search performance that is as good as we would get with a binary search on a sorted list. The tree with LV added is shown in fig (b).

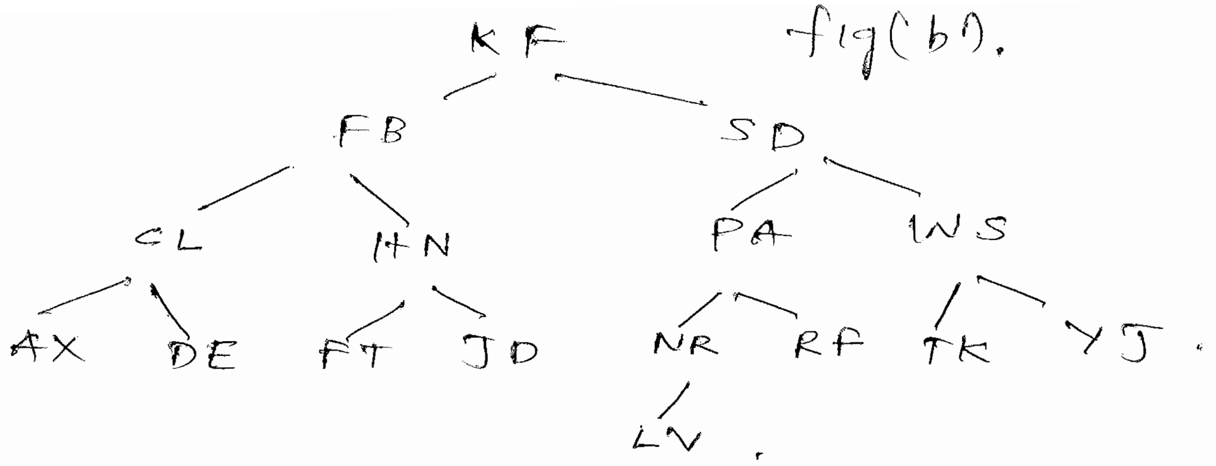
$\boxed{\text{ROOT}} \rightarrow 9$

	Key	left child	right child
0	FB	10	8
1	JD		
2	RF		
3	SD	6	13
4	AX		

	Key	left child	right child
6	PA	11	2
7	FT		
8	HN	7	1
9	KF	0	3
10	CL	4	12
11	NP		

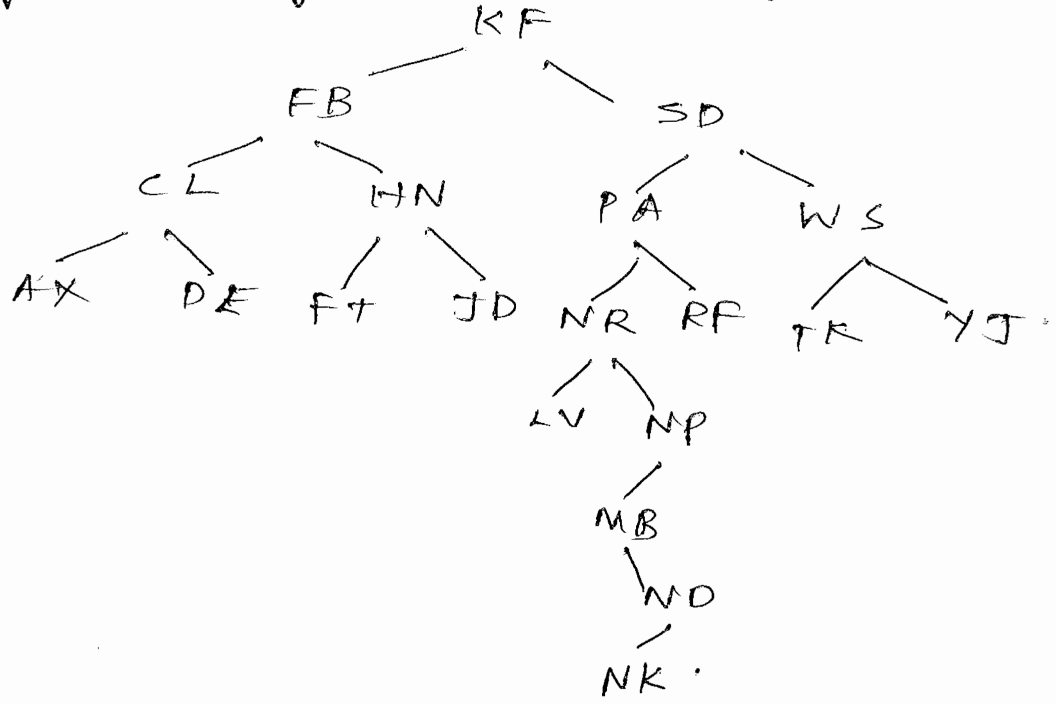
	Key	left child	right child
13	WS	14	5
14	TK		

fig(b).



* problems occur when the tree gets unbalanced.

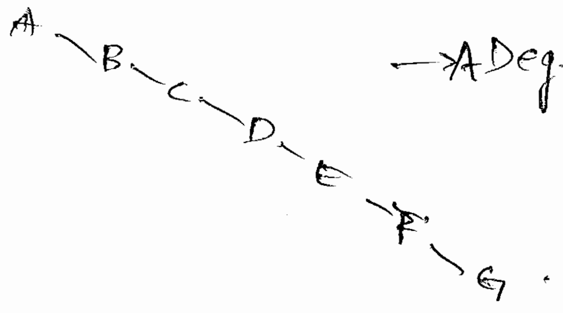
fig (c): Binary-search tree showing the effect of added keys.



* ∴ We look the schemes that allow trees to remain balanced.

AVL Trees

note:



→ A Degenerate tree.

* AVL trees allows us to reorganise the nodes of the tree as we receive new keys, maintaining a near optimal tree structure.

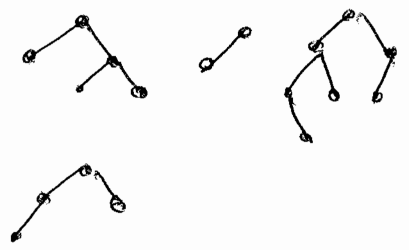
* AVL tree is named after pair of russian mathematician G M Adelson-Velskii and E.M Landis

* An AVL tree is an height balanced tree : ie, a tree that ~~produces~~ places a limit on the amount of difference allowed b/w the heights of any two subtrees sharing a common root.

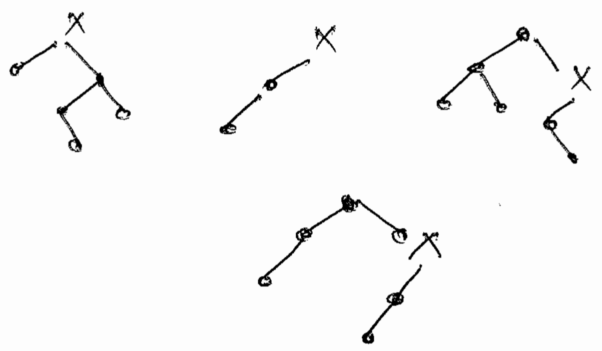
* In AVL tree (or HB-1 tree), maximum allowable difference is one.
generally, HB-K trees are permitted to be k-levels out of balance.

* eg:

fig(a): AVL trees



fig(b): Trees that are not AVL trees.



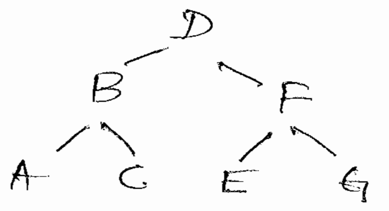
* Two features that make AVL trees important are

→ By setting a maximum allowable difference in the height of any ~~sub~~ two subtrees, AVL trees guarantees a minimum level of performance in searching; and

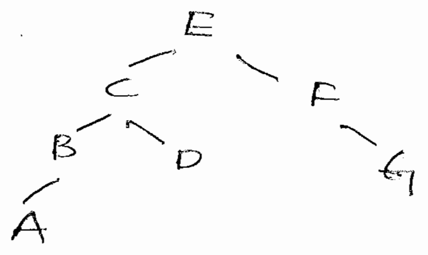
→ Maintaining a tree in AVL form as new nodes are inserted involves the use of a set of four possible rotations. Each of the rotation is confined to a single, local area of the tree. The most complex of the rotations requires only five pointer reassignments.

* AVL trees are not, themselves, directly applicable to most file structures because like all strictly binary trees, they have two many levels - they are too deep.
* AVL trees, however are important because they suggest that it is possible to define procedures that maintain height balance.
* search performance of an AVL tree approximates that of a completely binary tree.

eg: i/p keys - B C G E F D A



(a) completely balanced searchTree



(b) AVL tree.

for a completely balanced tree, the worst case search to find a key. (given N possible keys) is

$\boxed{\log_2 (N+1)}$

ie it looks at this no. of levels of the tree.

For an AVL tree, the worst case search could look at

$\boxed{1.44 \log_2 (N+2) \text{ levels.}}$

* Two problems we identified earlier are
→ Binary searching requires too many seeks
→ keeping an index in sorted order is expensive.

* soln to second problem is AVL tree
—"— first —"— is paged binary tree

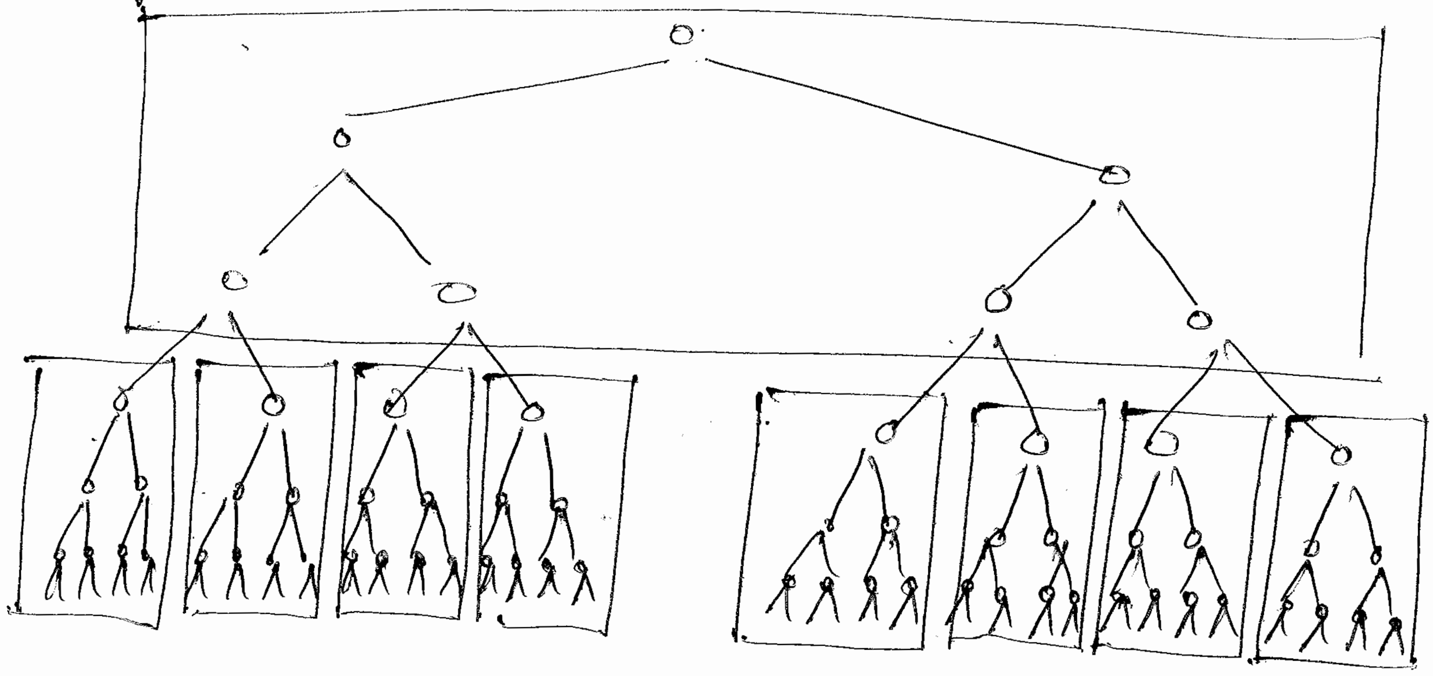
Paged Binary Trees

Binary searching requires too many seeks.

* Paged Binary Tree attempts to address the problem( ) (~~keeping an index in sorted order is expensive~~) by locating multiple binary nodes on the same disk page

* In a paged system you do not incur the cost of a disk seek just to get a few bytes. Instead, once you have taken the time to seek to an area of the disk, you read in an entire ~~file~~ page from the file.
This page might consist of many individual records; if the next bit of information you need from the disk is in this page; you have saved the cost of a disk access.

* Fig below illustrates a paged binary tree.



* When searching a binary tree, the no. of seeks necessary is $\boxed{\log_2(N+1)}$. It is $\boxed{\log_{K+1}(N+1)}$ in the paged version, where K→ no of keys held in single page.

note: second formula is generalization of the first, since no. of keys held in a page of a purely binary tree is 1;

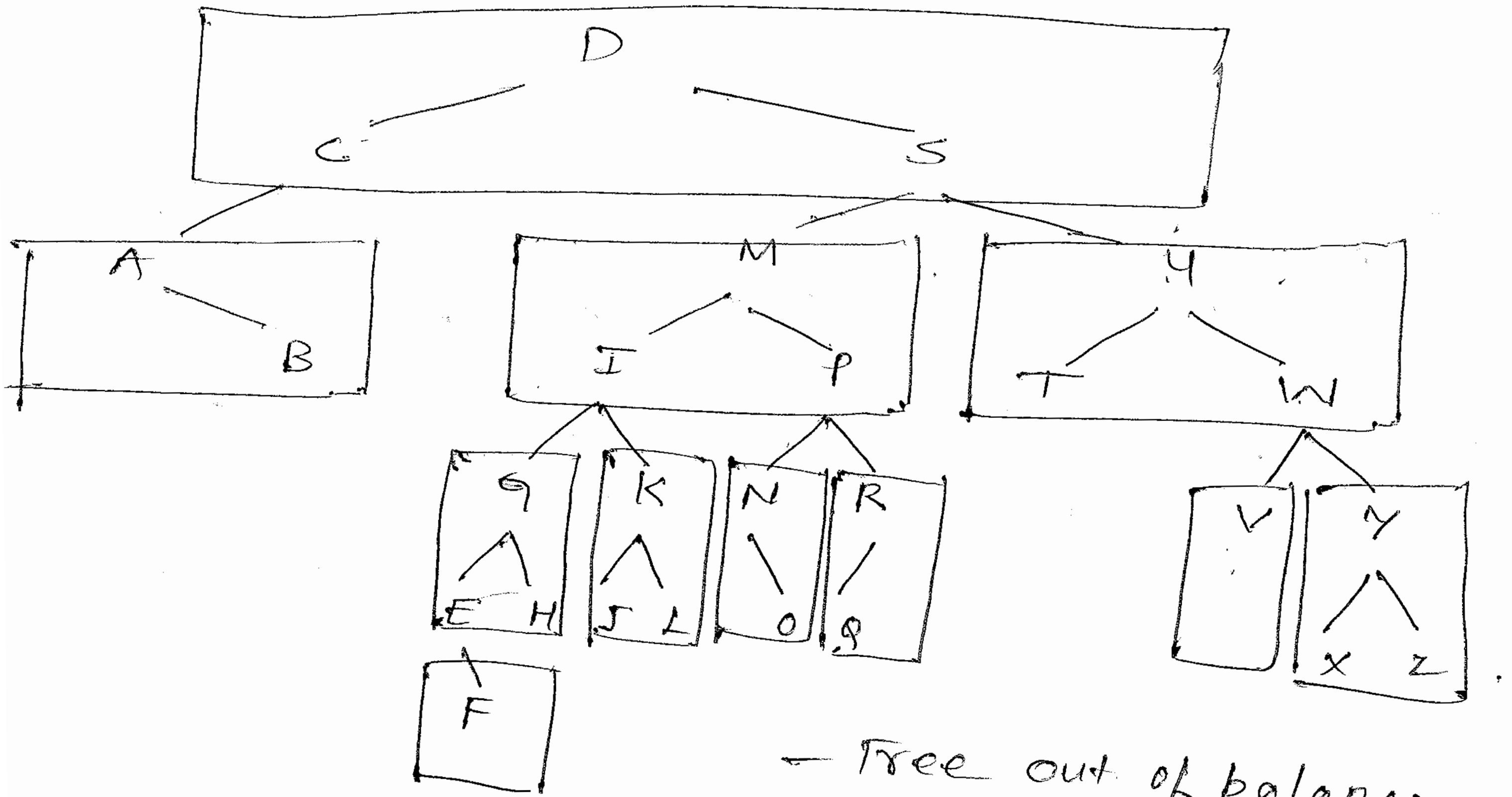
problems with paged Binary Trees.

* inefficient disk usage - major problem.
* How should we build a paged tree?
  - Easy if we know what the keys are and their order before starting to build a tree.
  - Much more difficult if we receive keys in random order and insert them as soon as we receive them. The problem is that the wrong keys may be placed at the root of the trees & cause an imbalance

*eg: Assume that we must build a paged tree as we receive the following sequence of single ~~letter~~ letter keys

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

resulting tree is shown below.



- Tree out of balance.

* Three problems ~~arised~~ arised with paged trees;

- How do we ensure that the keys in the root page turn out to be good separator keys, dividing up the set of other keys more or less evenly.
- How do we avoid grouping keys that should't share a page? (eg C,D,S in our ex)
- How can we guarantee that each of the pages contains atleast some minimum number of keys?

MULTILEVEL INDEXING: A BETTER APPROACH TO TREE INDEXES.

* upto this, we have seen indexing a file based on building a search tree. Also we have seen some problems associated with this.
* Instead, we get back to the notion of simple indexes we saw earlier (unit 3) but we extend this notion to that of multi record indexes, and then multilevel indexes.
* Multi Record Indexes.
  - A multi record index consists of a sequence of simple index records.
  - The keys in one record in the list are all smaller than the keys of the next record.
* Multi-level-indexes
  - It is nothing but the index of the index file.
  - Since index records form a sorted list of keys, we can choose one of the keys (for ex: largest) in each index record as the key of that whole record.

* multi record indexes and multi level indexes help reduce the no. of disk accesses and their overhead ~~costs~~ space costs are minimal.
But here, inserting a new key or deleting the old one is very costly.

B-TREES: WORKING UP FROM THE BOTTOM

* B-Trees
- Have advantages as that of multilevel indexes, and does not suffer from its disadvantages.
- B-Trees are built upward from the bottom rather than downward from top, thus addressing the problems of paged trees.
- B-Trees are multilevel indexes that solve the problem of linear cost of insertion & deletion. They are now standard way to represent indexes.
- B-Trees are balanced, shallow (requiring few seeks) and guarantee atleast 50% storage utilization.

Definition.

B-tree of order m is a multilevel index tree with these properties
- Every node has a maximum of m descendents.
- Every node except the root has atleast $\lceil m/2 \rceil$ descendents.
- The root has atleast two descendents (unless it is a leaf)
- All of the leaves appear on the same level.

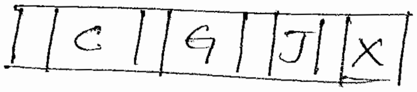
EXAMPLE OF CREATING A B-TREE

* Assume order = 4.

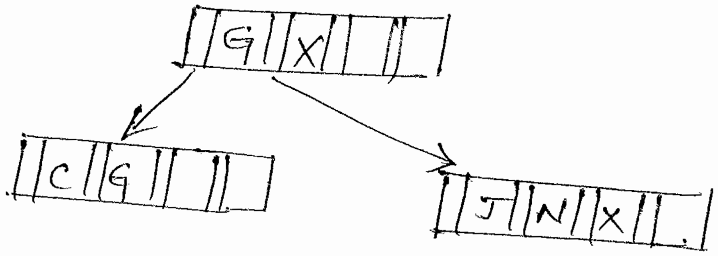
key sequence =

C G J X N S U O A E B H I F K

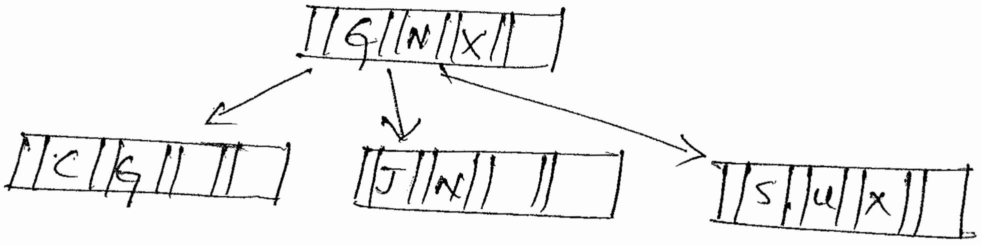
a.) Insertion of C, G, J, X to the initial node



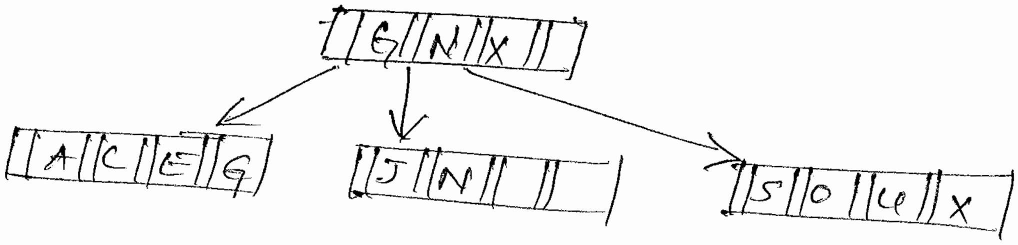
b.) Insertion of N causes node to split & the largest key in each leaf node (G & X) to be placed in the root node.



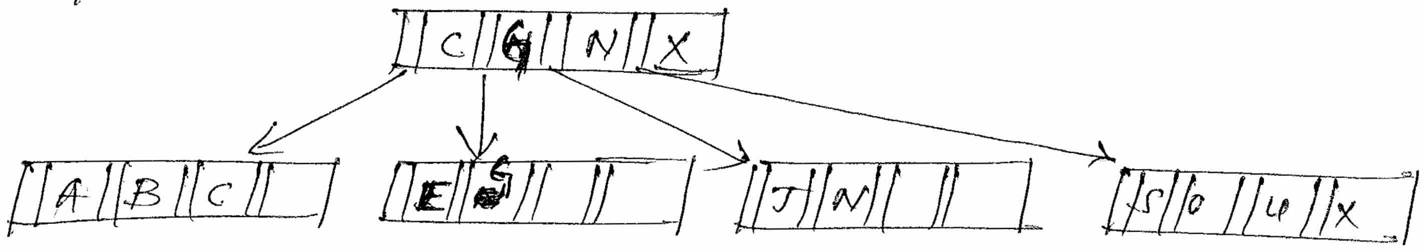
c.) 'S' is inserted into rightmost leaf node, and insertion of 'U' causes it to split



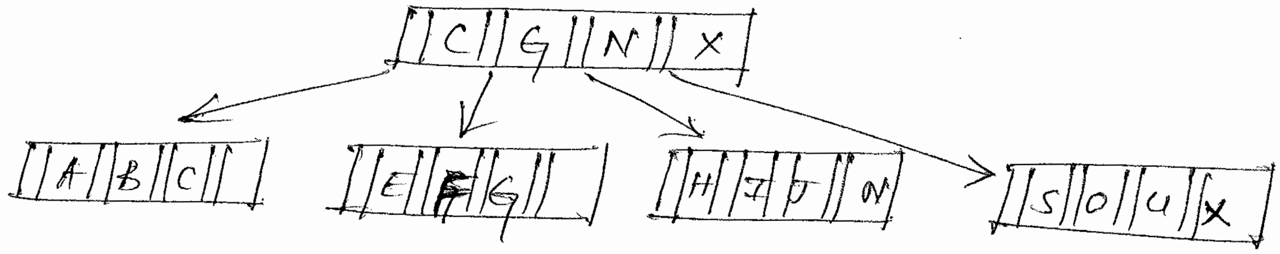
d.) 'O', 'A', 'E' are inserted into appropriate leaf nodes



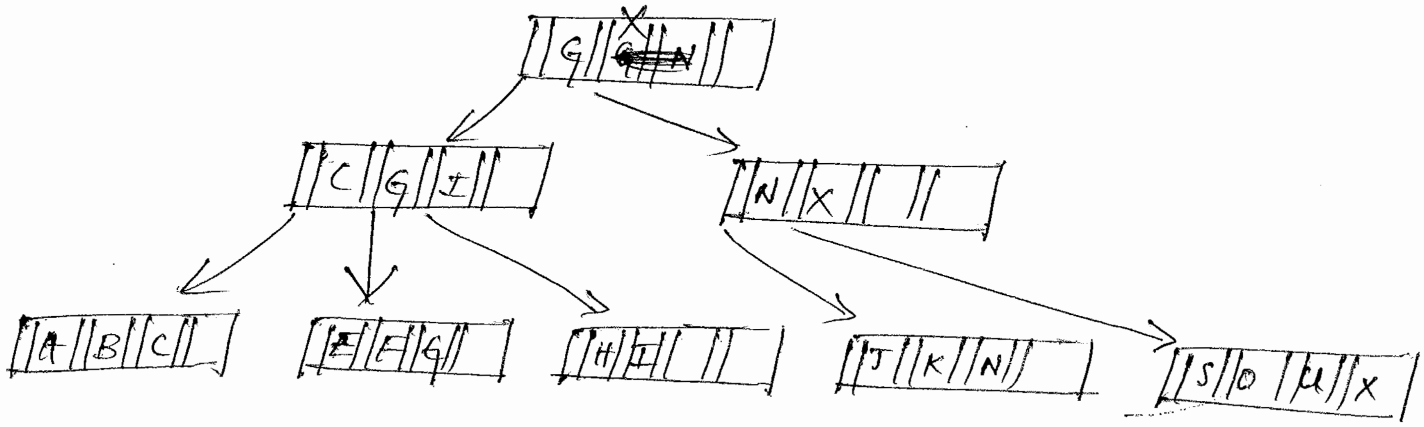
e). insertion of 'B' causes leftmost leafnode to split



f.) 'H' & I', F are inserted at appropriate leaf nodes.



(g) insertion of 'K' causes the leafnode to split, also root to split and the tree grows to level 3.



Note: references to actual record only occur in the leaf nodes. The interior nodes are only higher level indexes (this is why there are duplication in the tree).

B-TREE METHODS : SEARCH, INSERT, AND OTHERS

Searching

```
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 )
 { int recAddr , level;
   for (level=1 ; level < Height ; level ++ )
    {
      recAddr = Nodes[level-1] -> search (key, -1, 0);
      Nodes[level] = Fetch( recAddr );
    }
   return Nodes[level-1];
 }
```

Insertion

* Iterative procedure have 3 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 new root node ; if the current root was split

FORMAL DEFINITION OF B-TREE PROPERTIES.

* In a B-Tree of order m
- Every page has a maximum of m descendents.
- Every page, except for the root and leaves, has atleast ⌈m/2⌉ descendents.
- The root has atleast two descendents (unless it is a leaf)
- 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

⟨Relationship b/w the pagesize of a B-tree, the no. of keys to be stored in the tree, and the no. of levels that the tree can extend⟩

* Given 1,000,000 keys and a B-Tree of order 512. what is the maximum no. of disk accesses necessary to locate a key in the tree? In other words, how ~~long~~ deep will the tree be?

* WKT every key appears in the leaf level. Hence we need to calculate the max. height of the tree with 1,000,000 keys in the leaves.

* The maximum ~~no. of~~ height will be reached if all pages (or nodes) in the tree has the minimum allowed number of descendents - worst case.
(max height, min breadth)

* For a B-Tree of order m, the minimum no of descendents from the root page is 2. (So the second level of the tree contains only 2 pages). Each of these pages in turn has atleast ⌈m/2⌉ descendents.

* The general pattern of the relation b/w depth and the min. no. of descendents takes following form.

level	min. no. of descendents
1 (root)	2
2	$2 * \lceil m/2 \rceil$
3	$2 * \lceil m/2 \rceil * \lceil m/2 \rceil$ or $2 * \lceil m/2 \rceil^2$
4	$2 * \lceil m/2 \rceil^3$
⋮	⋮
d	$2 * \lceil m/2 \rceil^{d-1}$

ie for any level d of a B-tree the min. no. of descendents extending from that level is

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

* For a tree with N keys in its leaves, we have

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

Solving for d,

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

This exp. gives upper bound for depth of Btree with N keys.

* For $m = 512$, $N = 1,000,000$
we get $\boxed{d \leq 3.37}$

ie given 1,000,000 keys, a B-Tree of order 512 has a depth of no more than 3 levels.

DELETION, MERGING, AND REDISTRIBUTION.

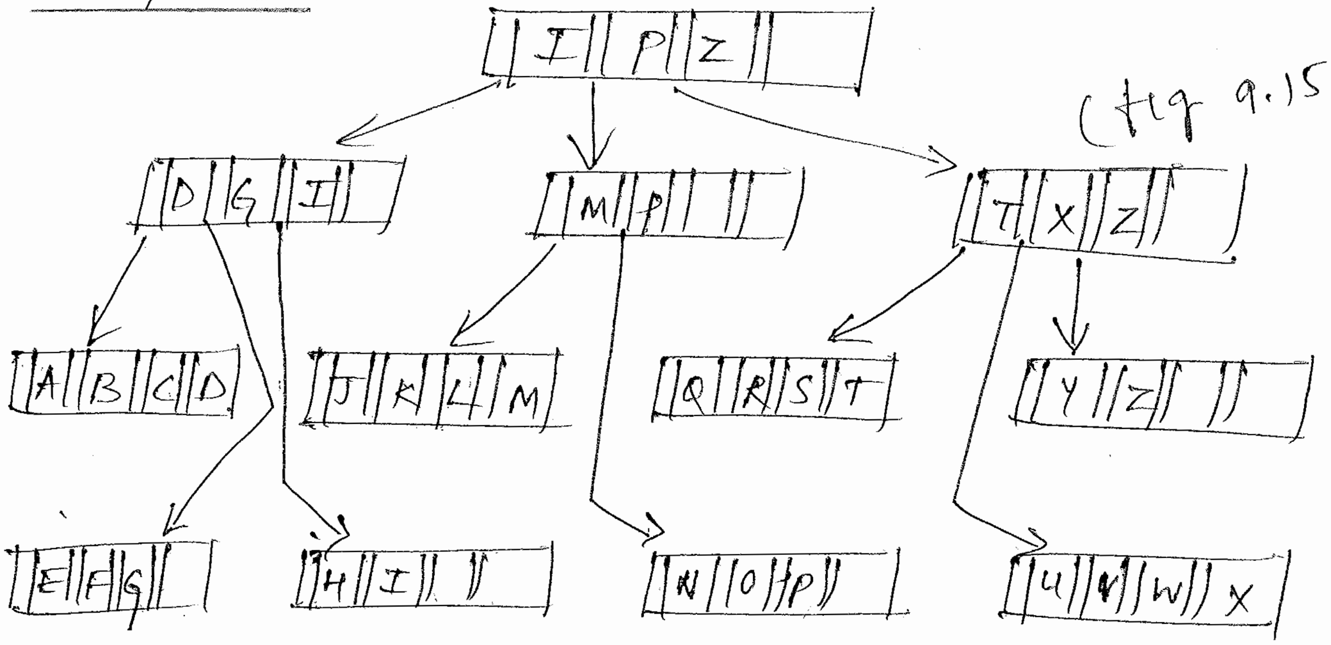
Rules for deleting a key k from a node n in a B-tree

1. If n has more than the min. no. of keys and k is not the largest in n, simply delete k from n.
2. If n has more than the min. no. of keys & the k is the largest in n, delete k and modify the higher level

3. If n has exactly the min. no. of keys and one of the siblings of n has few enough keys, merge n with its sibling and delete a key from the parent node

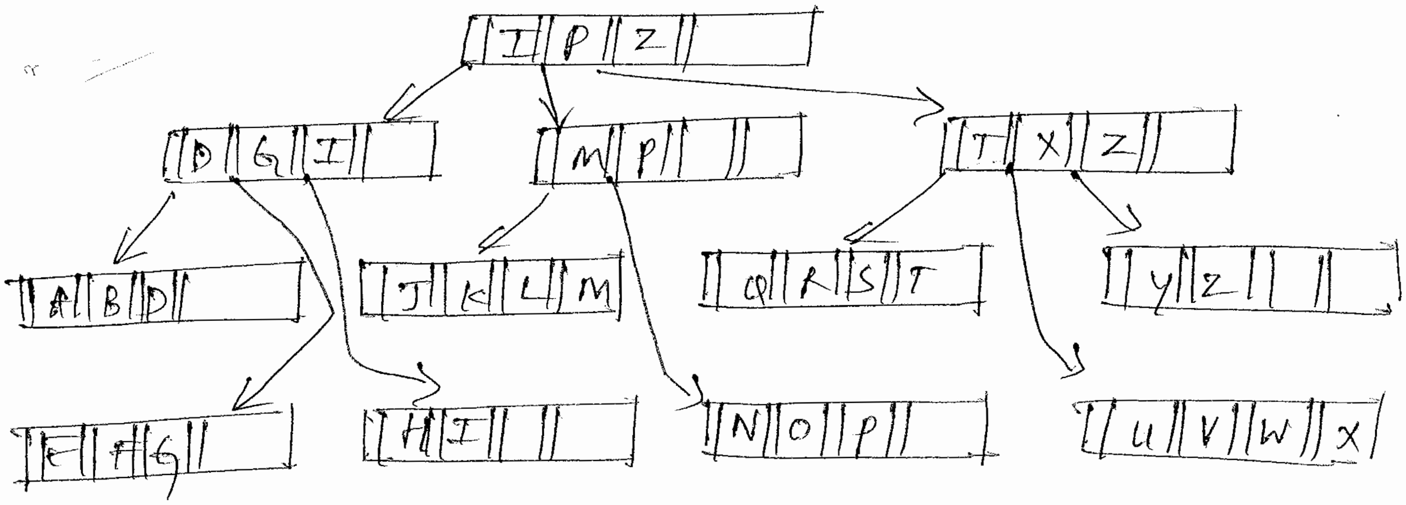
4. If n has exactly the min. no of keys & one of the siblings of n has extra keys, redistribute by moving some keys from a sibling to n, and modify the higher level indexes to reflect the new largest keys in the affected nodes.

Example.

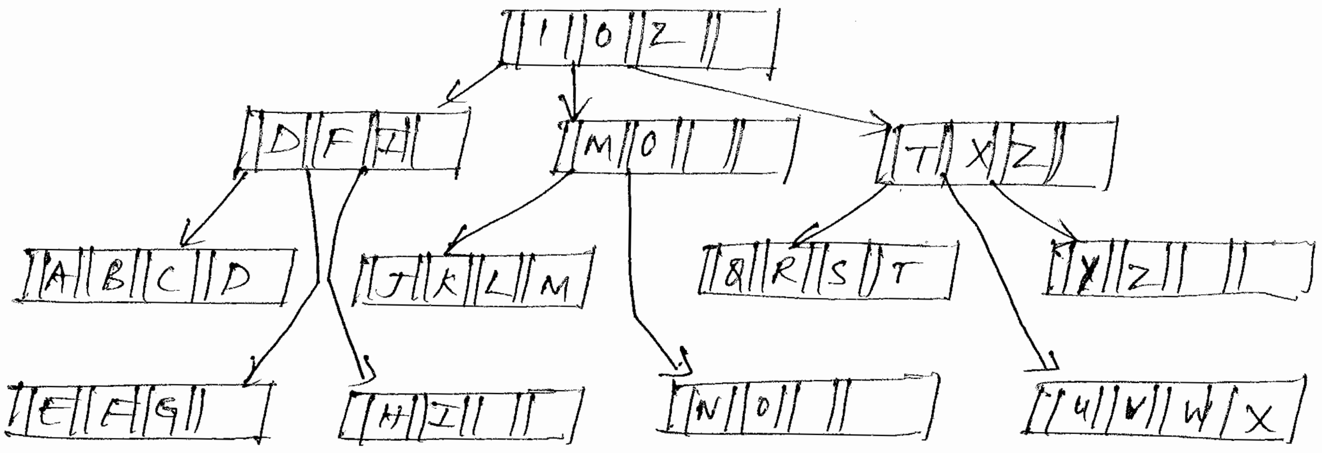


(fig 9.15).

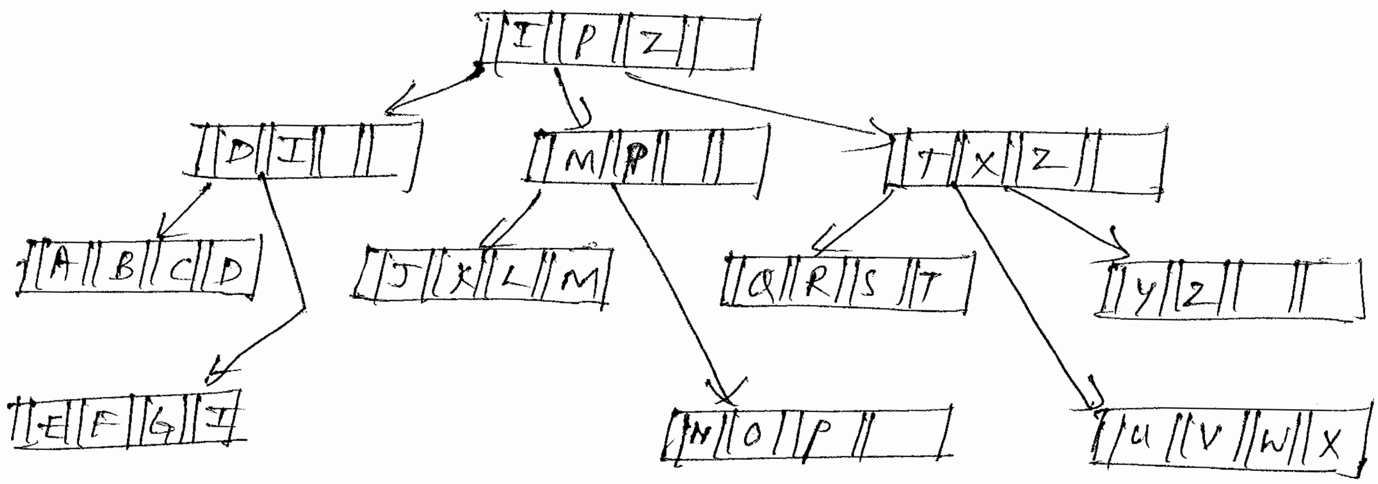
a i) Removal of c : change occurs only in ~~root~~ leaf node.



b:) Result of deleting P from fig 9.15
P changes to O in the second level & the root



c:) Result of deleting H from fig 9.15.
Removal of H caused an underflow, & two leaf nodes were merged.



Left Topics
- merging, Redistribution.
- B* trees, Virtual B-Trees.

Redistribution during insertion.

* Its a way to avoid / atleast postpone the creation of new pages
* Redistribution allows us to place some of the overflowing keys into another page instead of splitting an overflowing page.
* B* Trees formalizes this idea.

B* Trees properties

* Every page has max of m descendents.
* Every page except the root has atleast $\lceil \frac{2m-1}{3} \rceil$ descendents.
* Root has atleast two descendents (unless it is a leaf)
* All leaves appear on the same level.

The main difference b/w B-Tree and B* Tree is the 2nd rule.

NIT 6 : INDEXED SEQUENTIAL FILE ACCESS AND PREFIX B+ TREES

Syllabus

* Indexed sequential access
* Maintaining a sequence set
* Adding a simple index to sequence set
* The content of the index : separators instead of keys.
* The simple prefix B+ Tree.
* Simple prefix B+ tree maintainence
* Index set Block size
* Internal structure of index set blocks : A variable order B-tree
* Loading a simple prefix B+ tree.
* B+ trees.
* B-Trees, B+ Trees, and simple prefix B+ Trees in perspective

6 Hours.

Ashok kumar K

VIVEKANANDA INSTITUTE OF TECHNOLOGY

∝ —

INDEXED SEQUENTIAL ACCESS

* Indexed sequential file structures provide a choice between two alternative views of a file:

- Indexed : The file can be seen as a set of records that is indexed by key, or
- sequential : The file can be accessed sequentially (physically contiguous records - No seeking). returning records in order by key.

Definition:

Indexed sequential access is not a single-access method but rather a term used to describe situations in which a user wants both sequential access to records, ordered by keys, and indexed access to those same records.

B+ Trees are just one method for providing indexed sequential access.

MAINTAINING A SEQUENCE SET

* A sequence set is a set of records in physical key order which is such that it stays ordered as records are added and deleted.

The use of Blocks

* We can rule out sorting and resorting the entire sequence set as records are added and deleted, since we know that sorting an entire file is expensive process. Instead, we need to find a way to localize the changes.

* The idea is to use blocks that can be read into m/m & rearranged there quickly.
Like in B-Trees, blocks can be split, merged, or their

* using blocks, we can thus keep a sequence set in order by key without ever having to sort the entire set of records.

<u>Illustration: How the use of blocks can help us keep a sequence set in order.</u>

- Suppose we have records that are keyed on last name and collected together so there are four records in a block.
- We also include <u>link fields</u> in each block that points to the preceding block and the following block.

(link fields are needed because, consequetive blocks are not necessarily physically adjascent)

- Fig (a) shows initial blocked sequence set.

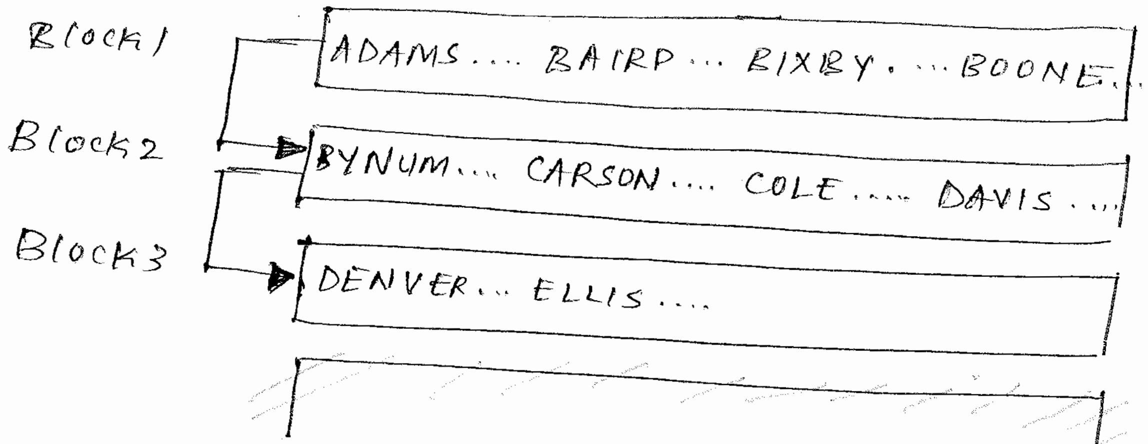

Block 1
ADAMS.... BAIRD... BIXBY.... BOONE..
Block 2
BYNUM.... CARSON.... COLE.... DAVIS....
Block 3
DENVER... ELLIS....

fig (a)

- insertion of new records into a block can cause the block to <u>overflow</u>. This condition can be handled by block splitting process (analagous, not same to that in case of B-Trees)
- Fig(b) shows sequence set after insertion of CARTER record

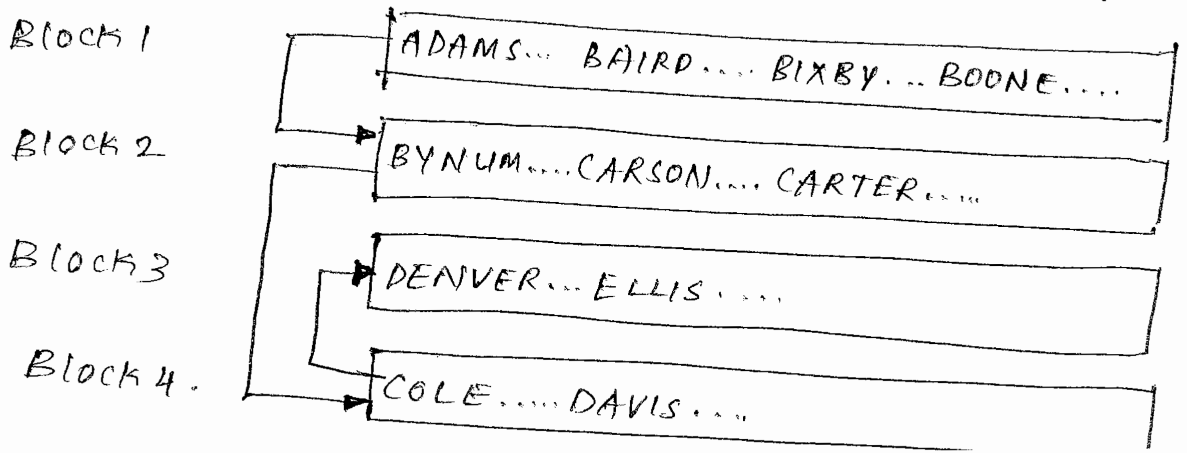
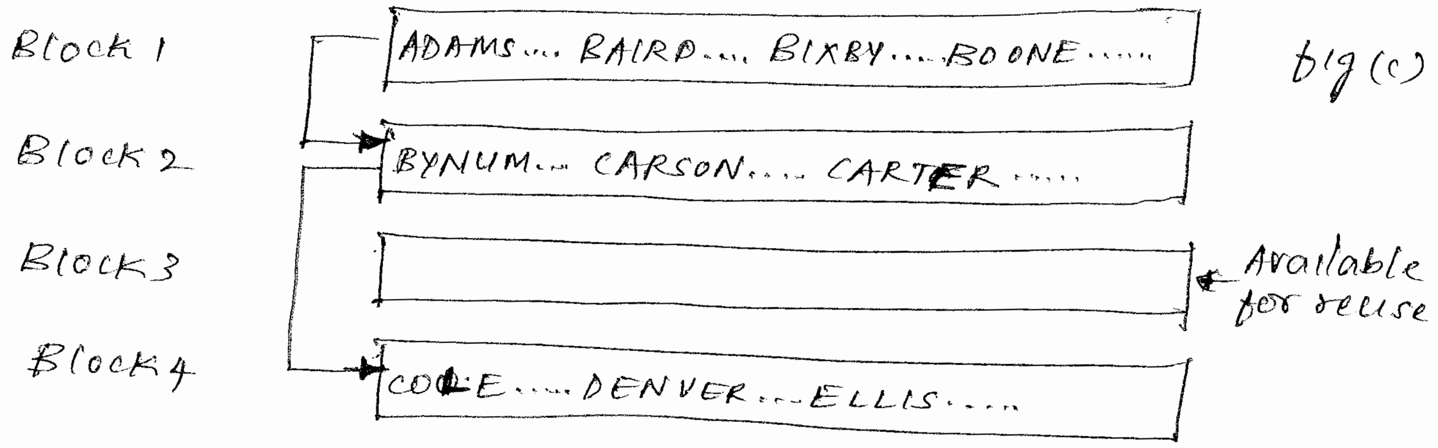

Block 1
ADAMS... BAIRD.... BIXBY... BOONE....
Block 2
BYNUM....CARSON.... CARTER.....
Block 3
DENVER... ELLIS....
Block 4.
COLE..... DAVIS....

fig (b).

This insertion causes block 2 to split. The second half of block 2 is found on block 4 after split.

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

fig (c) shows sequence set after deletion of DAVIS record



Block 1
ADAMS.... BAIRD.... BIXBY.... BOONE......
Block 2
BYNUM.... CARSON.... CARTER......
Block 3
Available for reuse
Block 4
COLE.... DENVER.... ELLIS....

fig (c).

After deleting DAVIS, block 4 underflows & is then merged with its successor in logical sequence, which is block 3. This merging process frees up block 3 for reuse

* Costs associated with this approach (Disadvantages of using blocks)

- Blocked file takes up more space than unblocked file because of internal fragmentation within a block.
- The order of the records is not necessarily physically sequential throughout the file. The maximum guaranteed extent of physical sequentiality is within a block.

Choice of Block size

* There are two consideration to keep in mind when choosing a block size.

Consideration 1:

The Block size should be such that we can hold several blocks in m/m at once. For ex: In performing a block split or merging, we want to be able to hold at least two blocks

Consideration 2: (← imprecise)

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

Consideration 2 (redefined)

The block size should be such that we can access a block without having to bear the cost of a disk seek within the block read or block write operation.

ADDING A SIMPLE INDEX TO THE SEQUENCE SET

* Each of blocks we created for our sequence set contains a range of records as shown.

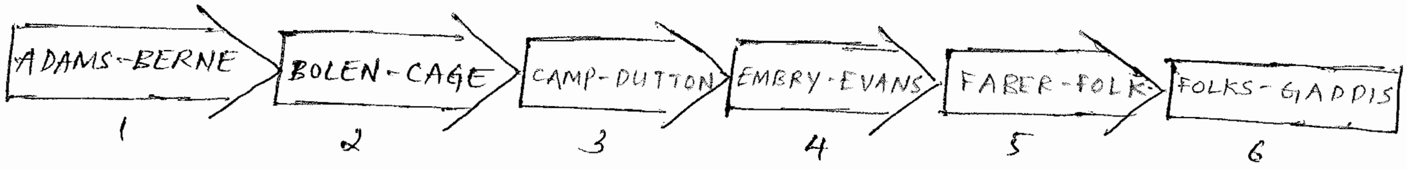


fig: sequence of blocks showing range of keys in each block.

* We can construct a simple, single level index for these blocks. We might choose, for ex, to build an index of fixed length records that contain the key for the last record in each block as shown:

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

fig: simple index for sequence set shown in above fig.

* The combination of this kind of index with the sequence set of blocks provides complete indexed sequential access.
* If we need to retrieve a specific record we consult the index and then retrieve the correct block.
* If we need sequential access, we start at the first block & read through the linked list of blocks until we have read them all.

* This method works well as long as the entire index can be held in m/m.

Why the entire index must be held in m/m?

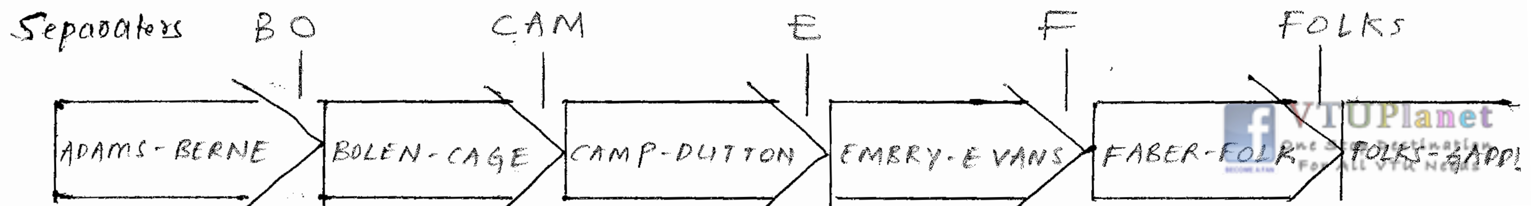
- We find specific records by means of binary search of the index, which takes too many seeks if the file is stored on sec. storage device.
- As blocks in the sequence set are changed (through splitting, merging, redistribution), the index has to be updated. Updating a simple, fixed length record index of this kind works well if the index is relatively small & contained in m/m.

* If entire index could not be held in m/m, then we can use a B+ Tree which is a B-tree index plus a sequence set that holds the records.

(B+ tree) = (B-tree) + (sequence set).

CONTENT OF THE INDEX : SEPARATORS INSTEAD OF KEYS

* purpose of the index we are building is to assist us when we are searching for a record with a specific key. It should guide us to the block in the sequence set that contains the record.
* What if, we do not ~~ha~~ need to have keys in the index set? What we really need are separators capable of distinguishing b/w blocks.
* Below fig shows one possible set of separators for sequence set shown before.



* We can save space by using variable-length separators and placing the shortest separator in the index structure

Note: There are many potential separators capable of distinguishing b/w two blocks
ex: fig shows a list of potential separators b/w blocks 3 & 4.

CAMP-DUTTON		EMBRY-EVANS
	DUTU	
	DVXGHESJF	
	DZ	
	E	
	EBQZ	
	ELEEMOSYNARY.	

* As we use the separators as a road map to the sequence set, we must decide to retrieve the block to the right of the separator or the one to the left of the separator acc. to following rules:

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

* C++ function to find a shortest separator.

```
void FindSeparator (char *key1, char *key2, char * sep)
{   while (1)
    {
        *sep = *key2 ;
        sep++;
        if (*key2 != *key1)
            break;
        if (*key2 == 0)
            break;
        key1++;
        key2++;
    }
    *sep = 0;
}
```

THE SIMPLE PREFIX B+ TREE

* Below fig shows how we can form the separators ~~and~~ identified in prev. section into a B-tree index of sequence set blocks.

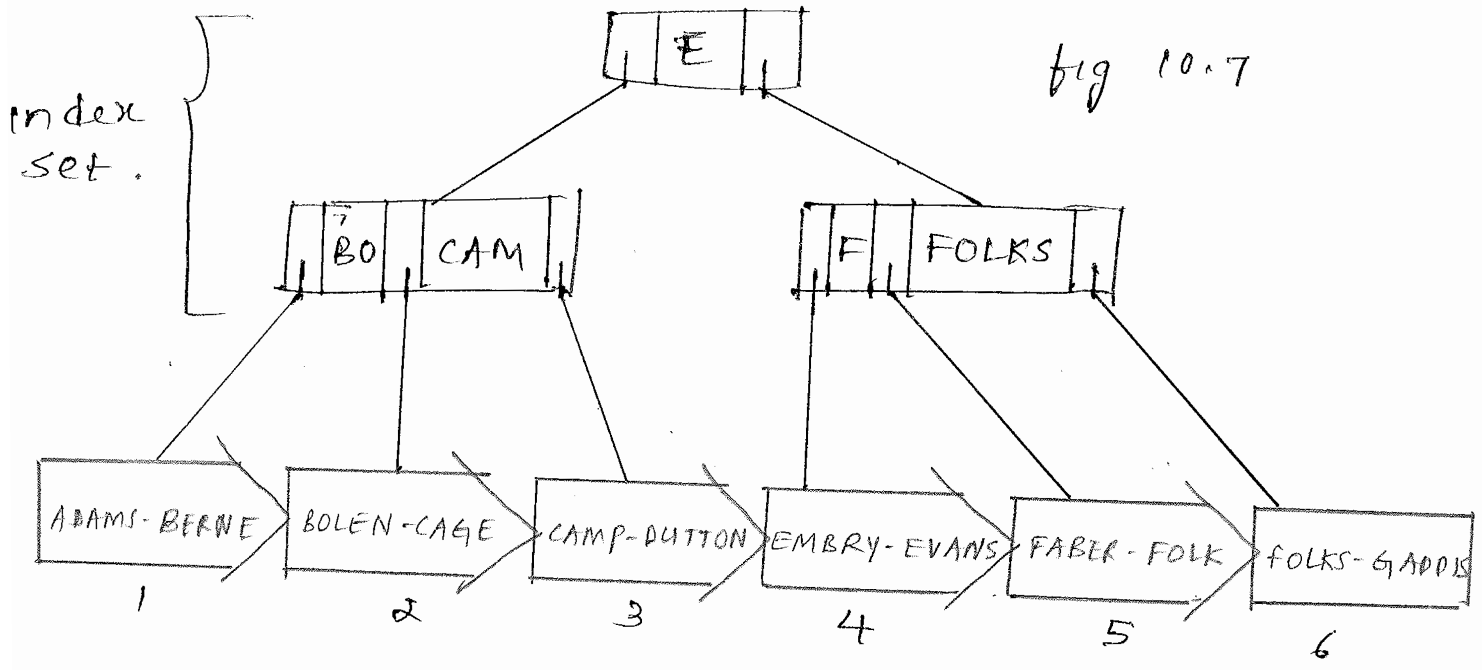


fig 10.7

* The B-Tree index is called the index set. Taken together with the sequence set, it forms a file structure called a simple prefix B+ tree.
The modifier "simple prefix" indicates that the index set contains shortest separators, or prefixes of the keys rather than the copies of actual keys.

SIMPLE PREFIX B+ TREE MAINTAINANCE

- changes localized to single blocks in the sequence set.

- changes involving multiple blocks in the sequence set.

changes localised to single blocks in the sequence set

* suppose that we want to delete any number of records from the simple prefix B+ tree, and if these deletions does not results in any merging or redistribution within the sequence set, then
  - The effect of these deletions on sequence set is limited to changes within particular blocks (from which records are deleted)
  - since the number of sequence set blocks is unchanged and since no records are moved b/w blocks, the index set can also remain unchanged (no need of changing the separators).
* Example: Below fig shows simple prefix B+ tree ~~res~~ after deleting EMBRY & FOLKS record from fig ~~7.1~~ 10.7

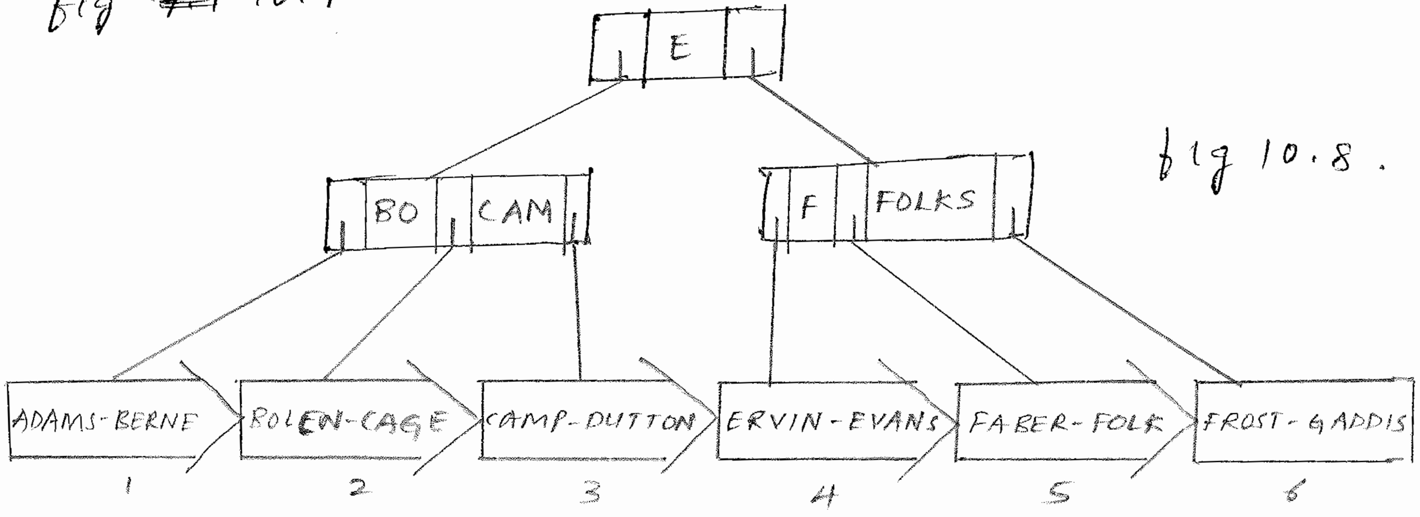


fig 10.8.

* The effect of inserting into the sequence set new records that do not cause block splitting is much the same ~~effect~~ as the effect of these deletions that do not result in merging - index set remains unchanged.

changes involving multiple blocks in the sequence set

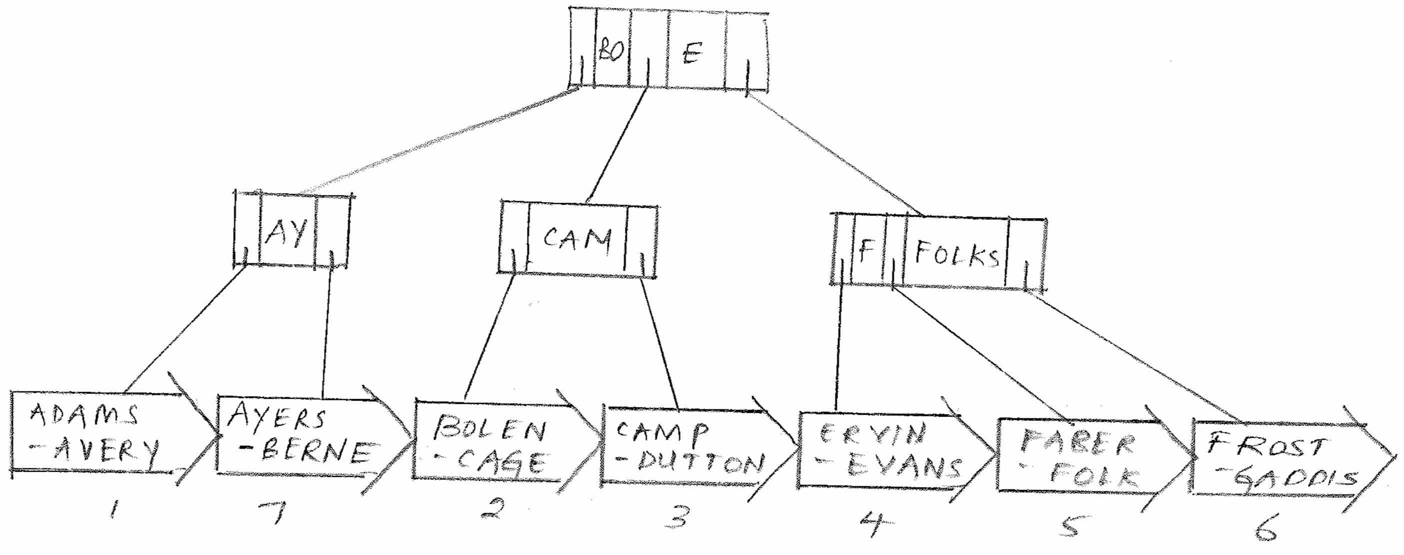
* changes takes place from the bottom up.
* If splitting, merging, or redistribution is necessary, perform the operations just as you would if there were

* Then, after the record operations in the sequence set are complete, make changes as necessary in the index set.

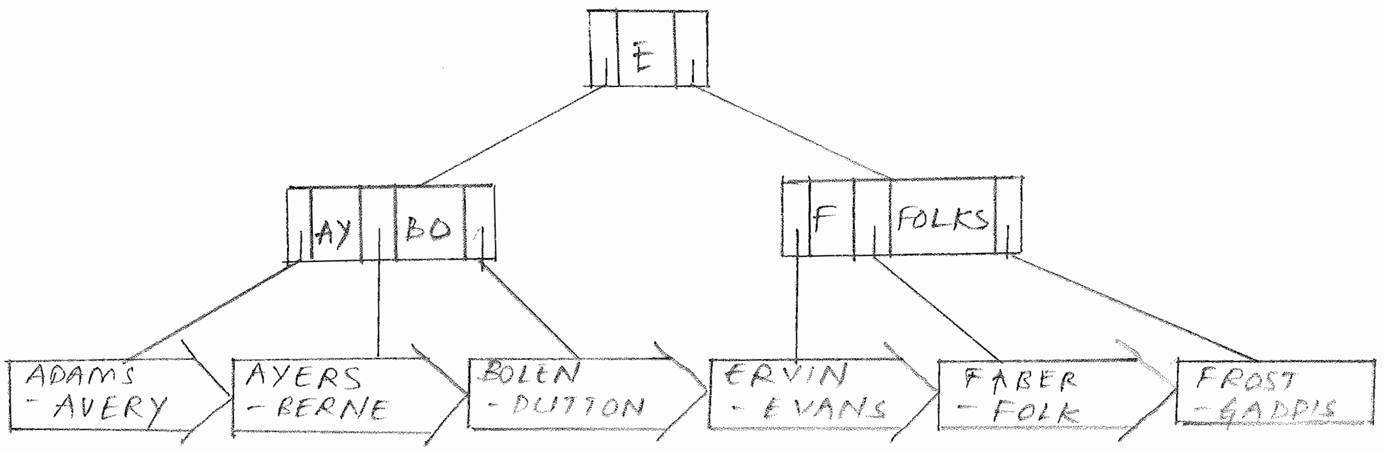
- if blocks are split in the sequence set, a new separator must be inserted into index set
- if blocks are merged in the sequence set, a separator must be removed from the index set
- if records are redistributed b/w blocks in the sequence set, the value of a separator in the index set must be changed.

Examples.

1. An insertion into block 1 (in fig 10.8) causes a split and the consequent addition of block 7.
   The addition of block in the sequence set requires a new separator in the index set.
   Insertion of AY separator into the node containing BO and CAM causes a node ~~to~~ split in the index set B-tree and consequent promotion of BO to the root



2. A deletion from block 2 causes underflow and the consequent merging of blocks 2 & 3.
   After merging, the block 3 is placed on 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



INDEX SET BLOCK SIZE

* The physical size of a node for the index set is usually the same as the physical size of a block in the sequence set. when this is the case, we speak of index set blocks, rather than nodes.

* Reasons for using common block size for index & sequence sets.

- The blocksize for sequence set is usually chosen because there is a good fit among this block size, the characteristic of the disk drive, & amount of m/m available. The choice of an index set blocksize is governed by consideration of some factors; therefore, 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~~ simple prefix B+ tree.

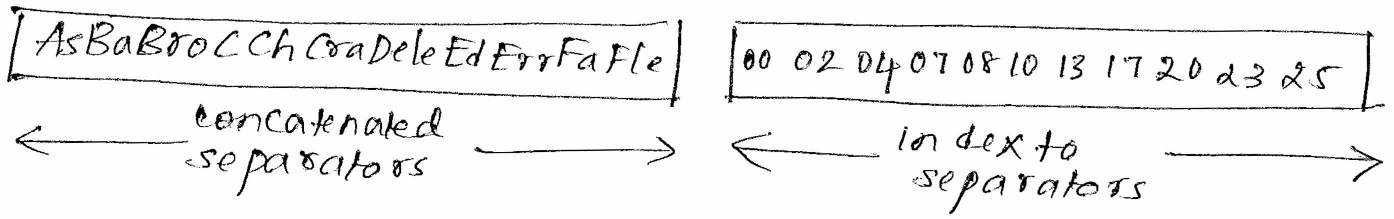
- The index set blocks and sequence set blocks are often mingled within the same file to avoid seeking ~~within~~ b/w two separate files while accessing the simple prefix B+ tree. Use of one file for both kinds of blocks is simpler if the block sizes are same.

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

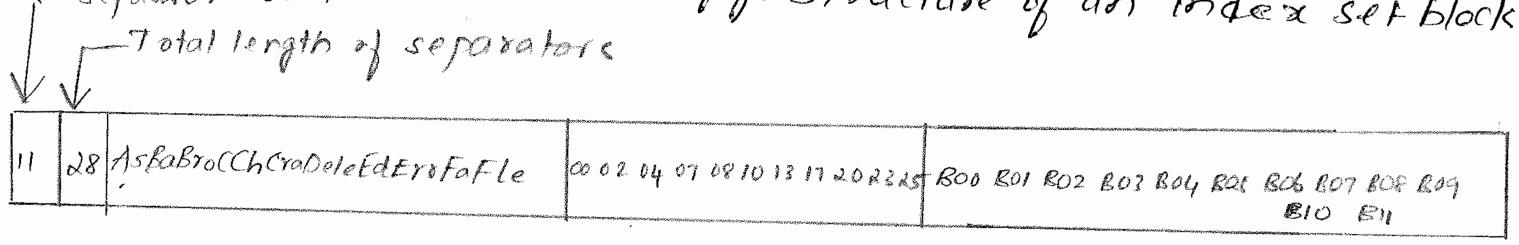
* Given a large, fixed size block for the index set, how do we store the separators within it?

~~* There are many ways to combine the list of separators, the index~~

* For ex: suppose we are going to place the following set of separators into an index block.
  As, Ba, Bro, C, ch, Cra, Dele, Edi, Err, Fa, Fle.
  We could merge these separators and build an index for them as shown



* There are many ways to combine the list of separators, the index to separators, and relative Block nos (RBNs) into a single index set block.
* One possible approach is illustrated in below fig.



Separator count
Total length of separators
11
28
AsBaBroCChCraDeleEdErrFaFle
00 02 04 07 08 10 13 17 20 23 25
B00 B01 B02 B03 B04 B05 B06 B07 B08 B09 B10 B11

fig: structure of an index set block

* In addition to the vector of separators, the index to these separators, & the list of associated block numbers, this block structure includes;

- Separator count:
  Helps us to find the middle element in the index to the separtors so we can begin our binary search.

- <u>Total length of separators</u>:

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

* Fig below shows the conceptual relationship of separators & the RBNs.

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

Suppose we are looking for a record with the key "Beck". We perform binary search & conclude that the key "Beck" falls b/w the separators "Ba" & "Bro". This allows us to decide that the next block we need to retreive has the RBN stored in the B02 position of the RBN vector.

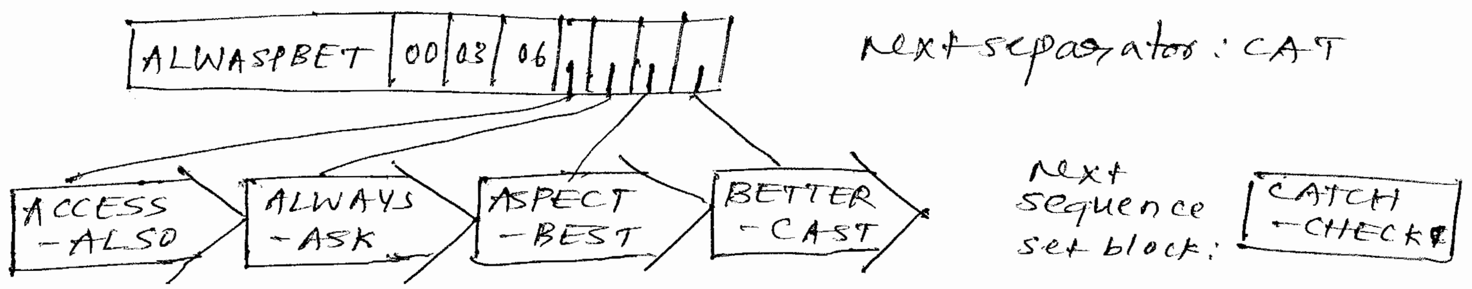
* <u>This kind of index block structure illustrates two imp. points</u>
1. A block can have a <u>sophisticated internal structure</u> all its own, including its own internal index, a collection of variable-length records, separate sets of fixed-length records, & so forth.
2. Node within the B-tree index set ~~of~~ of our simple prefix B^+ tree is of <u>variable order</u> (since each index set block contains variable no. of separators). <u>This variability has interesting implications</u>
- No. of separators in a block is <u>directly limited by block size</u> rather than by some <u>predetermined order</u> (as in an order m B-tree).
- Since tree is of variable order, operations like determining when a block is full, or half full, are <u>very complicated</u>. Decisions about when to split, merge, or redistribute become <u>more complicated</u>.

LOADING A SIMPLE PREFIX B+ TREE

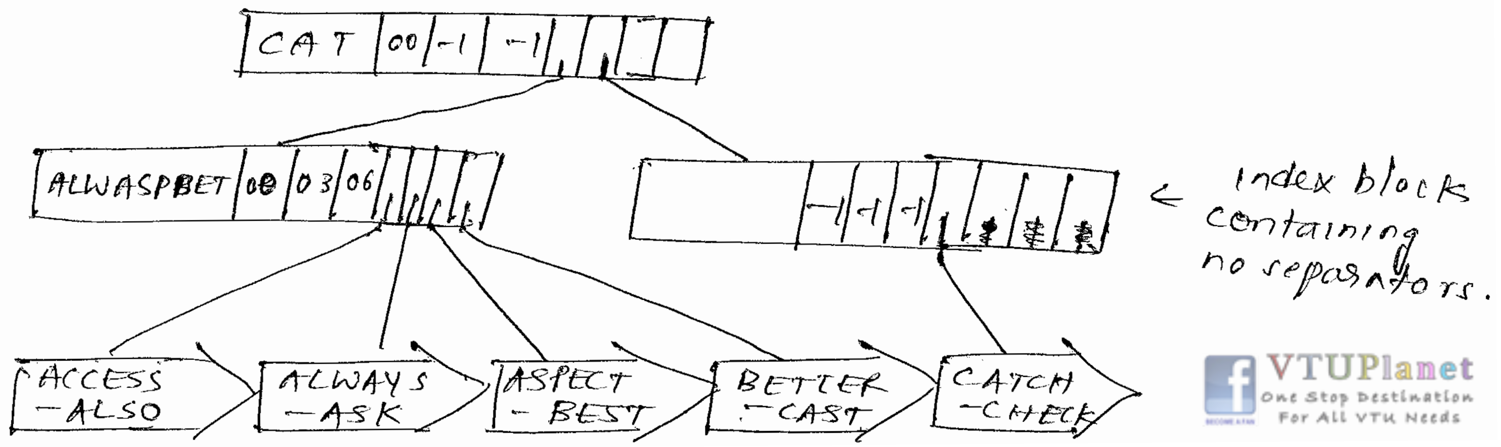
* one way of building a simple prefix B+ tree is through a series of successive insertions. this method is not good because splitting and redistribution are relatively expensive.

* Working from a sorted file, we can place the records into sequence set blocks, one by one, starting a new block when the one we are working with fills up. As we make transitions b/w two sequence set blocks, we can determine the shortest separator for the blocks. We can collect these separators into an index set block that we build and hold in m/m until it is full.

ex: fig shows four sequence set blocks that have been written out to the disk & one index set block that has been built in m/m from the shortest separators derived from the sequence set block keys.



* simultaneous building of two index set levels as the sequence set continues to grow.



Advantages

* Advantages of loading a simple prefix B+ tree almost always outweigh the disadvantages associated with the possibility of creating blocks that contain ~~few~~ too few records or too few separators.

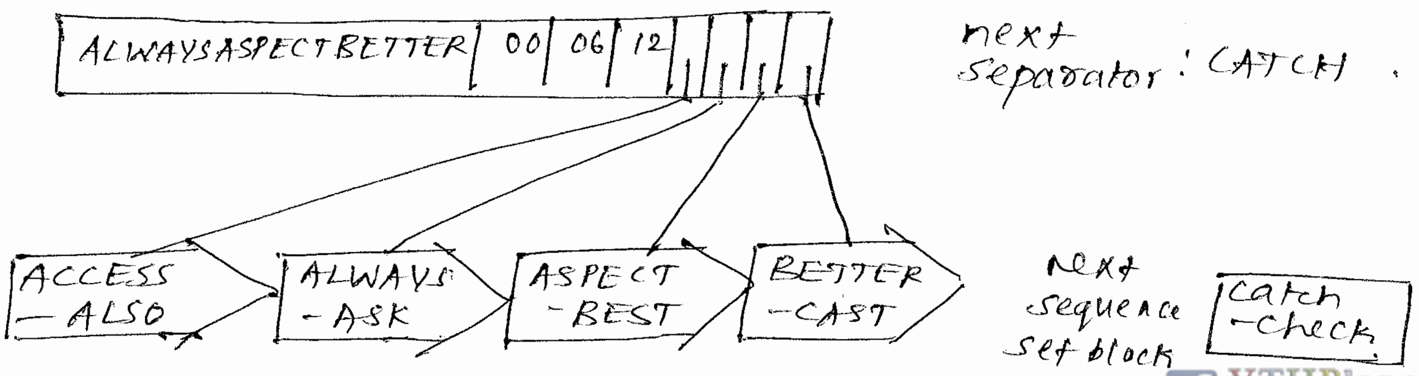
* particular advantage is that the loading process goes more quickly because
  - o/p can be written sequentially
  - We make only one pass over the data
  - No blocks needs to be reorganised as we proceed.

* Advantages related to performance after the tree is loaded
  - The blocks are 100% full by using sequential loading process
  - Sequential loading creates a degree of spatial locality within our file. ⇒ Seeking can be minimized.

B^+ Trees

* The difference b/w a simple prefix B^+ tree and a plain B^+ tree is that the latter structure does not involve the use of prefixes as separators. Instead, the separators in the index set are simply copies of the actual keys.

* ex:



* Simple prefix B^+ trees are often more desirable than plain B^+ trees because the prefix separators take up less space than the full keys.

* B^+ trees, however, are sometimes more desirable since
- They do not need variable-length separator fields. (Cost of extra overhead required to maintain & use var-length structure is eliminated)
- Some key sets do not show much compression when the simple prefix method is used to produce separators. for eg: keys - 34C18K756, 34C18K757, 34C18K758 ..... are diff. to compress.

B-TREES, B^+ TREES, AND SIMPLE PREFIX B^+ TREES IN PERSPECTIVE

* B and B^+ Trees are not the only tools used for file structure design. Simple indexes are useful when they can be held fully into memory, and hashing can provide much faster access than B & B^+ trees.
* common char. of B and B^+ and simple prefix B^+ trees.
- They are all paged index structures, ie they bring entire blocks of information into m/m at once. ⇒ Broad & shallow trees.
- All three maintain height-Balanced trees.
- In all cases, the tree grows from bottom up. Balance is maintained through block splitting, merging & redistribution
- with all three structures, it is possible to obtain greater storage efficiency through the use of two-to-three splitting & of redistribution in case of block splitting when possible.
- All three approaches can be implemented as virtual tree structures in which the most recently used blocks are held in m/m.
- Any of these ~~structures~~ approaches can be adapted for use with variable length records

Differences b/w various structures (through a review of strengths & unique char. of each of these file structures)

* B-Trees

- B-Trees are multilevel indexes to data files that are entry sequenced.
- strengths: simplicity of implementation, inherent efficiency of indexing, maximization of breadth of B-tree
- Weaknesses: excessive seeking necessary for sequential access (∵ lack of organisation of data file)

* B-Trees with associated information

- These are B-Trees that contain record contents at every level of the B-Tree
- strengths: can save up space
- Weaknesses: Works only when the record information is located within the B-Tree. Otherwise, too much seeking is involved in retrieving the record information.

* B+ Trees

- The primary difference b/w the B+ Tree & B-tree is that in the B+ tree, all the key & record information is contained in a linked set of blocks known as the sequence set.
- Indexed access to this sequence set is provided through a conceptually (not necessarily physically) separate structure called index set.
- Advantages.
  - Sequence set can be processed in a truly linear, sequential way, providing efficient access to records in order by key.
  - Index is built with a single key or separator per block of data records instead of one key per data record ⇒ index is smaller & hence shallower.

* <u>simple prefix B⁺ trees</u>
- the separators in the index set are smaller than the keys in the sequence set
  ⇒ <u>Adv:</u> Tree is <u>even</u> smaller.

Ashok Kumar. K
VIVEKANANDA INSTITUTE OF TECHNOLOGY

UNIT 7:

HASHING

Syllabus

* Introduction
* A simple Hashing Algorithm.
* Hashing Functions and Record Distribution.
* How much Extra m/m should be used?
* Collision Resolution by progressive overflow.
* Buckets
* Making Deletions.
* Other Collision Resolution Techniques
* Patterns of Record Access

– 7 Hours.

Ashok Kumar K
VIVEKANANDA INSTITUTE OF TECHNOLOGY

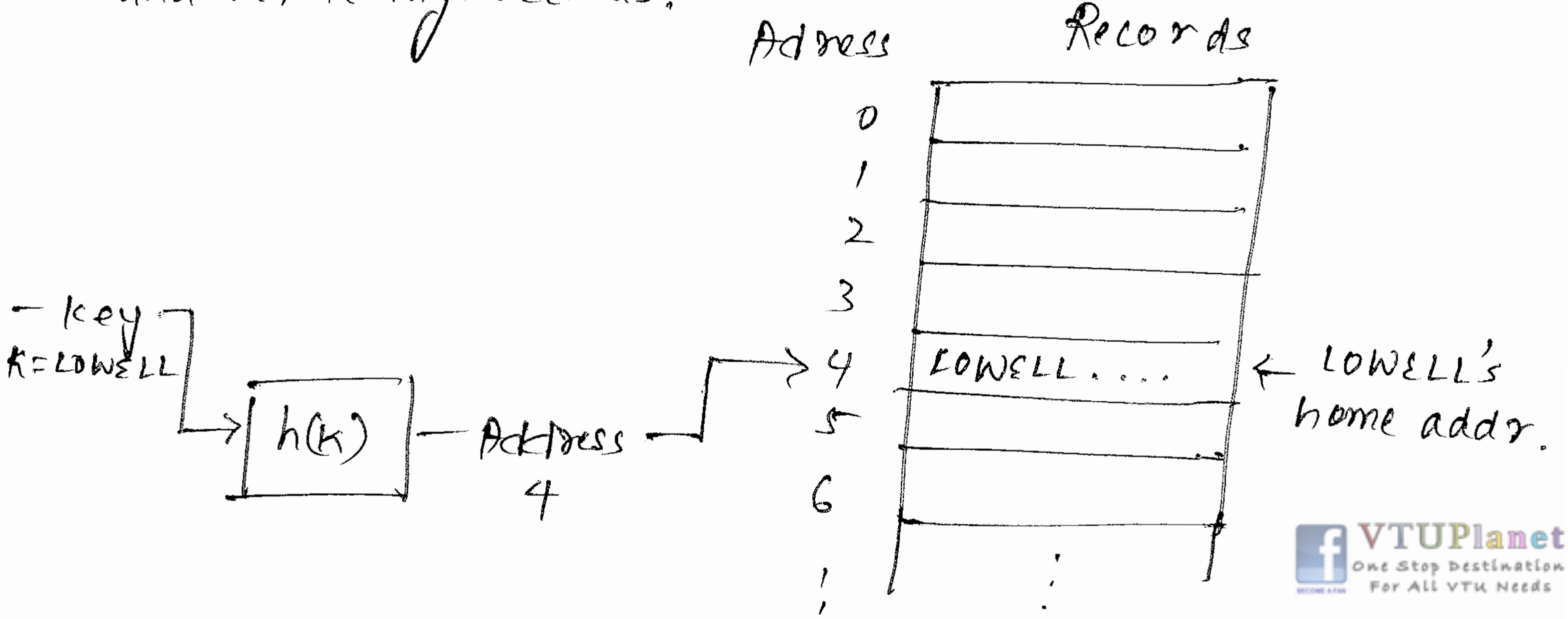
INTRODUCTION.

* Sequential searching can be done in $O(N)$ access time. ie no. of seeks grows in proportion to the size of the file.
* B-Trees improves on this greatly, providing $O(\log_k N)$ access where $K \rightarrow$ measure of the leaf size (ie number of records that can be stored in a leaf)
* What we would like to achieve is an $O(1)$ access. ie no matter how big a file grows, access to a record always takes same small no. of seeks.
* Static Hashing Techniques can achieve such performance provided that the file does not increase in time.

What is Hashing ??

* Hash function $h(K)$ is like black box that transforms a key K into an address.

The resulting address is used as the basis for storing and retrieving records.



* Hashing is like indexing in that it involves associating a key with a relative record address.
  Hower Hashing differs from indexing in two important ways:-

  1. With hashing, there is no obvious connection between the key and the location.
  2. With hashing, two different keys may be transformed to the same address.

Collisions

* When two different keys produce the same address, there is a collision. The keys involved are called synonyms.
* Avoiding collisions is extremely difficult. So we find ways to deal with them.
* possible solutions:
  1. spread out the records.
  2. use extra memory.
  3. put more than one record at a single address.

A SIMPLE HASHING ALGORITHM.

step 1: Represent the key in Numerical Form.

eg : 76 79 87 69 76 76 32 32 32 32 32 32

L O W E L L ← BLANK →

ASCII code.

step 2: Fold and Add

It means chopping off pieces of the number & adding them together. Here we chop off pieces with two ASCII nos each:

7679 | 8769 | 7676 | 3232 | 3232 | 3232

$\therefore$ 7679 + 8769 + 7676 + 3232 + 3232 = 30588 ($\because$ Adding one more 3232 results in overflow. ie > 32767)

choose one number that is largest allowable intermediate result for ex 19937.

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

13883 is the result of fold and add operation.

Step 3: Divide by a prime number and use the remainder as the address. ← size of address space.

formula: $a = s \bmod n$ ← no of addresses in a file

↑ address we are trying to ↑ sum produced eg1: a = 13883 mod 100

= 83

↳ is called the home address of the record

eg2: a = 13883 mod 101

= 46.

ie record whose key is LOWELL is assigned to record no 46 in a file.

```
int Hash (char key[12], int maxAddr)
{  int sum=0;
   for(int j=0; j<12; j+=2)
      sum= (sum*100 + key[j] + key[j+1]) %
                                     19937;
   return sum % maxAddr;
}
```

HASHING FUNCTIONS AND RECORD DISTRIBUTIONS.

Distributing Records Among Addresses

* Records can be distributed among addresses in different ways-

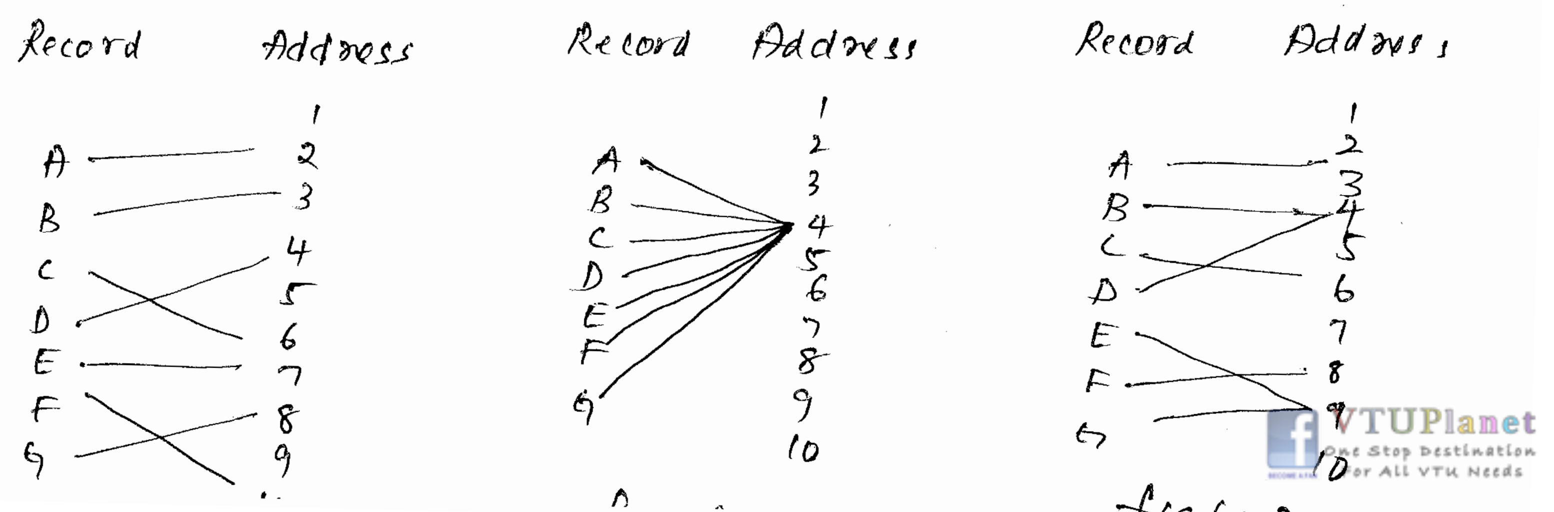


fig (a) : Best , no synonyms (uniform distribution)
fig (b) : Worst , All synonyms
fig (c) : Acceptable , A few synonyms (Random distribution)

* purely uniform distributions are difficult to obtain, and may not be worth searching for.
* Random distributions can be easily derived, but they are not perfect since they may generate a fair number of synonyms.
* So we look at better Hashing methods.

Some Other Hashing Methods.

* Though there is no hash function that guarantees better than random distributions in all cases, by taking into consideration the keys that are being hashed, certain improvements are possible.
* There are some methods that are potentially better than random -

1. Examine Keys for a pattern:
   Sometimes keys fall in pattern that naturally spread themselves out (eg employee id (key) may be ordered). This leads to no synonyms.
2. Fold parts of a key.
   Involves extracting digits from part of a key and adding the extracted parts together. This method may spread the keys naturally in some circumstances.

3. Divide the key by a number:

Division preserves consequetive key sequences, so you can take advantage of sequences that effectively spread out keys. Researches have shown that dividing by a number with no divisors less than 19 avoid collisions.

4. Square the key & take the middle:

often called mid-square method,

eg: key = 453, address space = 0 to 99

↳ its square is 205209.

extracting middle two digits yields a no. b/w 0 to 99

Here it is 52.

5. Radix Transformation:

involves converting the key to some other base & then taking the result modulo max addr.

eg: key = 453, addr space = 0 to 99

↳ its Base 11 ~~rep~~ equivalent is 382.

382 mod 99 = 85

predicting the Distribution of Records

* when using a random distribution, we can use a number of mathematical tools to obtain conservative estimates of how our hashing function is likely to behave.

the poisson Distribution.

used to approximate the distribution of records among addresses if the distribution is random.

formula

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

where $N \rightarrow$ no. of addresses available

$r \rightarrow$ no. of keys.(ie set of items)

$\hookrightarrow$ ie no. of ~~at~~ records to be stored.

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

eg: if $x=0$, we can compute the probability that a given address will have 0 records assigned to it by the hashing func. using the formula,

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

The poision function applied to Hashing

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

$x \rightarrow$ no. of records assigned to a given address.

In general,

If there are N addresses, then the expected no. of addresses with x records assigned to them is

$$\boxed{N\,p(x)}$$

predicting collisions for a full file

* suppose you have a hashing function that you believe will distribute records randomly and you want to store 10000 records in 10000 addresses.

1. How many addresses do you expect to have no records assigned to them?

$r = 10000 \quad N = 10000 \Rightarrow r/N = 1$

Hence,

(the proportion of addresses with 0 records assigned) $p(0) = \frac{1^0 e^{-1}}{0!} = 0.3679$

∴ the no. of addresses with no records assigned is

$10000 \times 0.3679 = \boxed{3679}$

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

$10000 \times P(1) = 10000 \times 0.3679 = \boxed{3679}$

$10000 \times P(2) = 10000 \times 0.1839 = \boxed{1839}$

$10000 \times P(3) = 10000 \times 0.0613 = \boxed{0613}$

→ Here there are 1839 overflow records

→ Here there are 613 × 2 1226 overflow records.

lets try to reduce the no. of overflow records.

HOW MUCH EXTRA MEMORY SHOULD BE USED ?

* Reducing collisions can be done by choosing good hashing function or using extra m/m.
* Q: How much extra m/m should be used to obtain a given rate of collision reduction ??

packing Density.

* Definition: packing Density refers to ratio of the number of records to be stored (r) to the number of available spaces (N)

$$\text{packing density} = \frac{\text{no. of records}}{\text{no. of spaces}} = \frac{r}{N}$$

It gives a measure of amount of space in a file that is used.

eg : $n = 100, r = 75$

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

predicting collisions for Different packing Densities.

* The poissons distribution allows us to predict the number of collisions that are likely to occur given a certain packing density (r/N)

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

* we use poisson's distribution to answer the following questions.

consider $N = 1000$, $r = 500$ ie $\frac{r}{N} = 0.5$

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

$\Rightarrow \quad Np(0) = 1000 \times \dfrac{0.5^0 e^{-0.5}}{0!} = 607$

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

$\Rightarrow \quad Np(1) = 303$

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

$\Rightarrow \quad p(2) + p(3) + p(4) + \ldots$

$= 0.0758 + 0.0126 + 0.0016 + 0.0002$

$= 0.0902$

$N(p(2) + p(3) \ldots) = 1000 \times 0.0902$

$= 90$.

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

$\Rightarrow 1 \times N \times p(2) + 2 \times N \times p(3) + 3 \times N \times p(4) + 4 \times N \times p(5)$

$= 107$

5. What percentage of records should be overflow records?

$\Rightarrow$ if there are 107 overflow records & 500 records in all, then the proportion of overflow record is -

$\dfrac{107}{500} = 21.4\%$

note: As packing density ↑↑, synonyms also ↑↑.

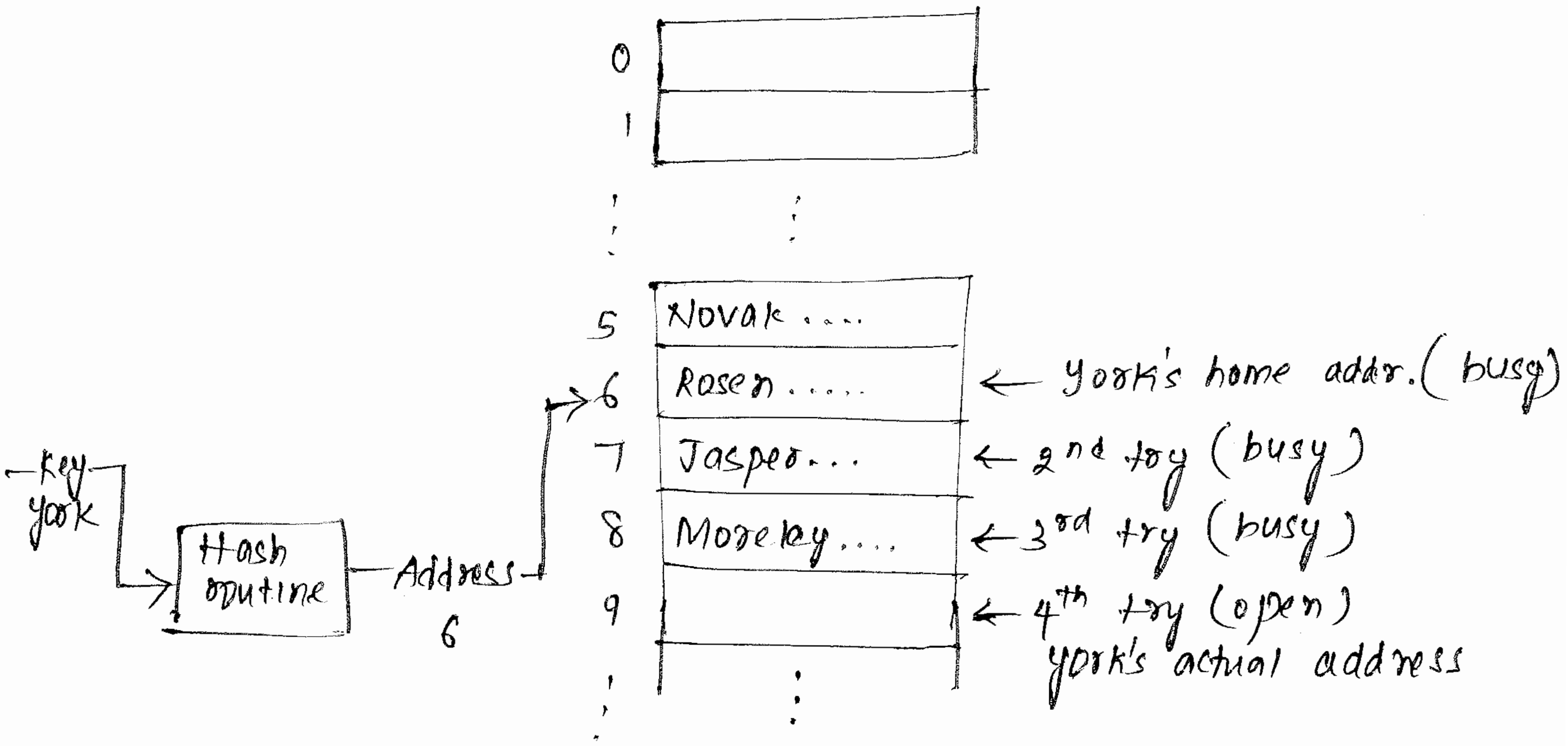
COLLISION RESOLUTION BY PROGRESSIVE OVERFLOW.

* How do we deal with records that cannot fit into their home address?

A simple approach: progressive overflow or linear probing.

* progressive overflow is a overflow handling technique in which collisions are resolved by storing a record in the next available ~~at~~ address after its home address.

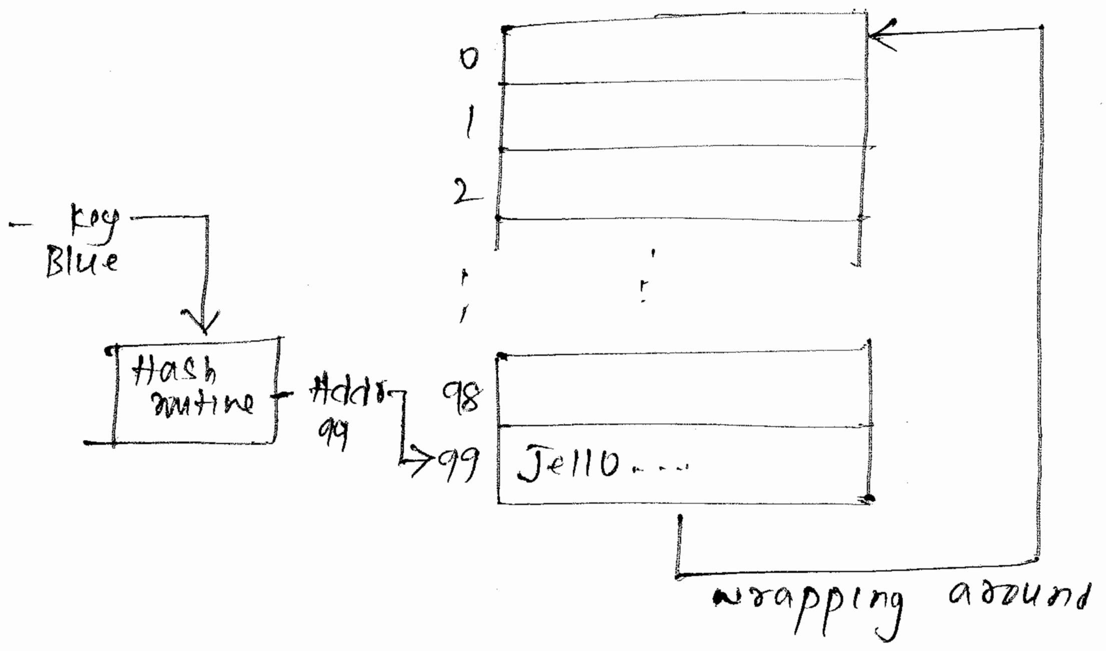
Example:



* The name york hashes to the same address as the name Rosen, whose record is already stored there. Since york cannot fit in its home addr, it is an overflow record

* If progressive overflow is used, the next several addresses are searched in sequence until an empty one is found. (If end of addr. space is reached, then wrap around ~~it~~ the addr space) The first free address becomes the address of the ~~record~~ record

* In this ex, addr 9 is the first rec. found empty. so the record pertaining to york is stored in addr. 9.



Blue is hashed to record 99, which is already occupied by Jello. Since file holds only 100 records, it is not possible to use 100 as next address.

<u>soln</u>: wrap around the address space of the file by choosing address 0 as next addr. Blue gets stored in addr 0.

* what happens if there is a search for a record but the record was never placed in a file.? Search begins from home address, then two things can happen

- if open addr. is encountered, the searching routine might assume this means that the rec. is not in the file, or
- if the file is full, the search comes back to where it began. only then it is clear that the rec. is not in the file.

Search length

* progressive overflow causes extra searches & thus extra ~~seeks~~ disk ~~seeks~~. accesses.
  If there are many collisions, then many records will be far from home.

* Definition:
  Search length refers to the number of accesses required to retrieve a record from secondary memory.
  Average search length is the average number of times you can expect to have to access the disk to retrieve a record.

$$\left.\begin{array}{l}\text{Average search}\\ \text{length}\end{array}\right\} = \frac{\text{Total search length}}{\text{Total number of records}}.$$

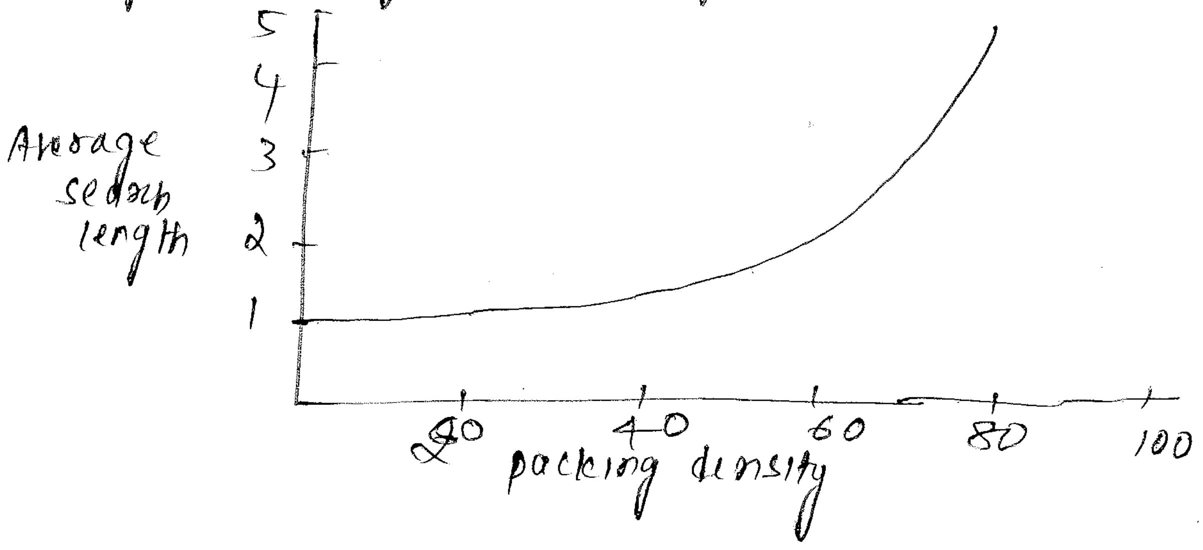
eg:

		Home addr	no. of accesses needed to retrieve
0			
⋮	⋮		
20	Adams	20	1
21	Bates	21	1
22	Cole	21	2
23	Dean.....	22	2
24	Evans	20	5
25			
⋮	⋮		

$$\left.\begin{array}{l}\text{Average}\\ \text{search}\\ \text{length}\end{array}\right\} = \frac{1+1+2+2+5}{5} = \underline{\underline{2.2}}$$

note: Average search length greater than 2.0 are generally considered unacceptable.

Average search length v/s packing density



STORING MORE THAN ONE RECORD PER ADDRESS : BUCKETS.

* Definition: A bucket describes a block of records ~~that is~~ sharing the same address that is ~~described by~~ retrieved ~~by~~ in one disk access.

* when a ~~bucket~~ record is stored or retrieved, its home bucket address is determined by Hashing. when a bucket is filled, we still have to worry about the record overflow problem, but this occurs much less often than when each address can hold only one record.

eg :-

key	Home addr
Green	30
Hall	30
Jenks	32
King	33
land	33
marx	33

Bucket contents

30	Green...	Hall...	
31			
32	Jenks...		
33	King....	land...	Marx...

(Nutt is an overflow record)

fig: each bucket can hold upto 3 records. Only one synonym here.

Effects of Buckets on performance

+ To compute how densely packed a file is -

$$\boxed{\text{packing density}) = \frac{r}{bN}}$$

$N \rightarrow$ no of addresses Buckets.
$r \rightarrow$ no of records in a file
$b \rightarrow$ no. of records that fit in a bucket.

(bN is the no. of available locations for records)

+ suppose we have 750 records to be stored in a file

1. we can store 750 records among 1000 locations, where each locn can hold one record.

$$\text{packing density}) = \frac{750}{1000} = \underline{\underline{75\%}}$$

2. we can store 750 records among 500 locations, where each locn has a bucket size of 2.

$$\text{packing density}) = \frac{r}{bN} = \underline{\underline{75\%}}$$

MAKING DELETIONS

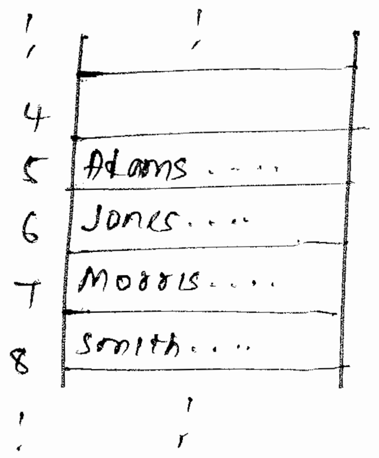
# Deleting a record from a Hashed File is more complicated than adding a record for two reasons -

1. The slot freed by the deletion must not be allowed to hinder later searches
2. It should be possible to reuse the freed slot for later additions.

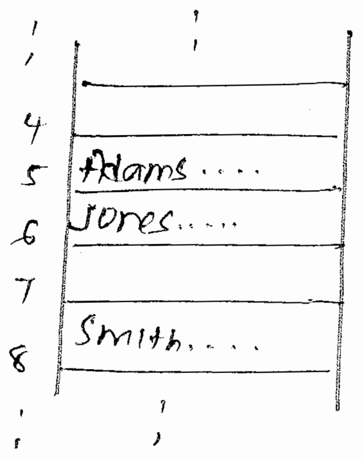
* In order to deal with deletions, we use tombstones ie a marker indicating that a record once lived there but no longer does.

placed in the key field of a record

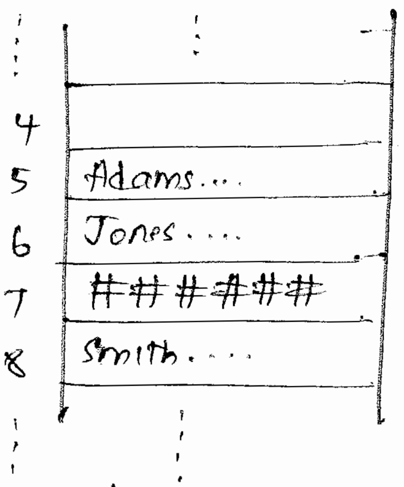
Tombstones solve both the problems caused by deletion.



(a) File organisation before deletions.



(b) File orgⁿ with Morris deleted.



(c) File orgⁿ after insertion of a tombstone for Morris.

note: It is not necessary to insert a tombstone every time a deletion occurs
for eg: The slot next to smith is empty, thus if we delete smith, there is no need to insert a tombstone

~~Impt~~

* Insertion of records is slightly different (difficult) when using tombstones -

- If you want to add smith rec. to the file shown in fig(c). Assume Home addr. of smith is 5.
- If the prog simply searches until it encounters #####, it never notices that smith is already in the file. It results in duplication which we don't need.
- To prevent this, the program must examine entire cluster of contiguous keys & tombstones to ensure that no ~~duplication~~ duplicate key exists, then go back & insert the record in the first available tombstone, if there is one.

Effects of Deletions and Additions on performance.

* After a large number of deletions and additions have taken places, one can expect to find many tombstones occupying places that could be occupied by records whose home address proceeds them but that are stored after them. This deteriorates average search lengths.
* There are 3 types of solutions for dealing with this problem -
  1. Doing a bit of local reorganizing every time a deletion occurs.
  2. Completely reorganising the file after the average search length reaches an unacceptable value.
  3. Use a different collision resolution algorithm.

OTHER COLLISION RESOLUTION TECHNIQUES

# There are a few variations on random Hashing that may improve performance -

1. Double Hashing:

when a collision occurs, a second hash function is applied to the key to produce a number c that is relatively prime to the number of addresses.

The value c is added to the home address to produce the overflow address. If the overflow address is already occupied, c is added to it to produce another overflow address. process continues until a free overflow addr is found.

2. chained progressive overflow.

# It works in the same manner as progressive overflow, except that synonyms are linked together with pointers. ie each home address contains a number indicating the location of the next record with the same home address, This next record intum contains a pointer to the following record with the same home address, and so forth.

Key	Home address	Actual Address	Search length
Adams	20	20	1
Bates	21	21	1
cole	20	22	3
Dean	21	23	3
Evans	24	24	1
Flint	20	25	6

Average search length ~~####~~ $= \frac{1+1+3+3+1+}{6}$

$= \underline{\underline{2.5}}$

fig: Hashing with progressive overflow.

Home addr.	Actual addr.	Data.	Addr of next synonym	search length
20	20	Adams	22	1
21	21	Bates	23	1
20	22	Cole	25	2
21	23	Dean	-1	2
24	24	Evans	-1	1
20	25	Flint	-1	3

fig: Hashing with chained progressive overflow.

$$\text{Average Search length} = \frac{1+1+2+2+1+3}{6} = \underline{\underline{1.7}}$$

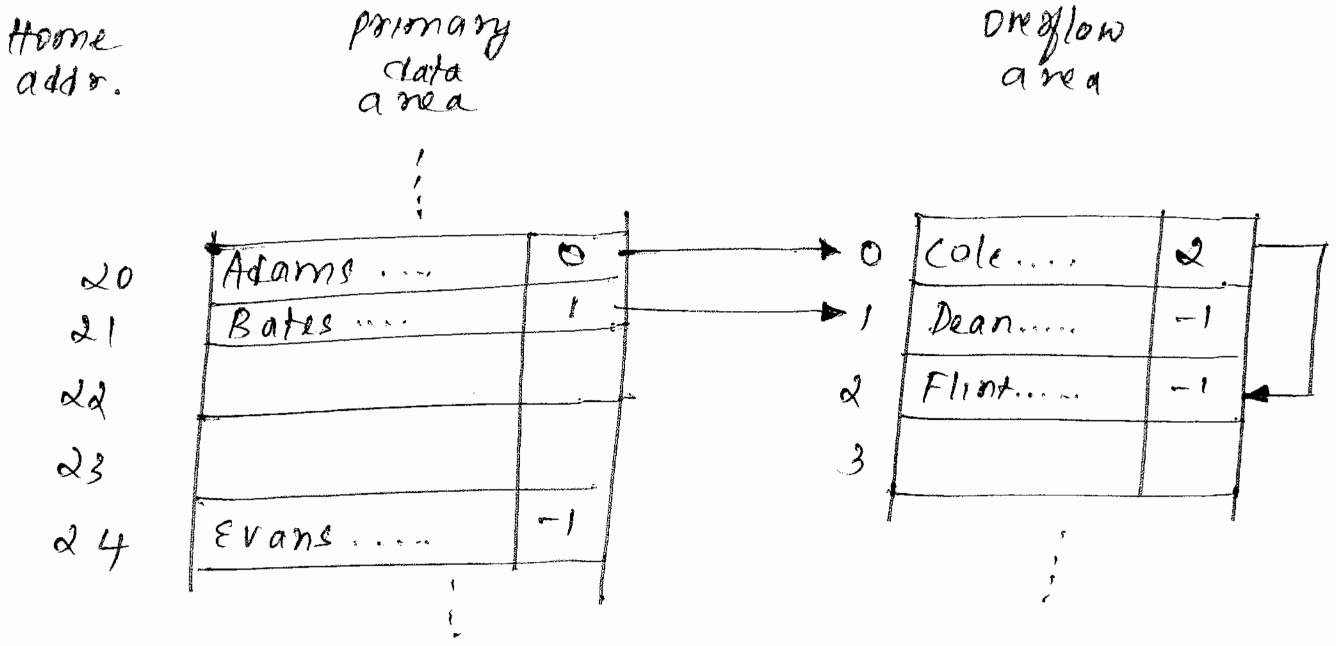
Suppose in the above ex Dean's home addr is 22.
The problem here is certain address (22) that should be occupied by a home record (Dean) is occupied by a different record.

Soln:- Load the file using a technique called Two-pass loading.
It involves loading a hashfile in two passes.
On the first pass, only home records are loaded. All other records are kept in separate file. This ~~guan~~ guarantees that no potential home addr are occupied by overflow records.
On the second pass, each overflow record is loaded & stored in one of the free addresses according to whatever collision resolution technique is being used.

3. Chaining with a separate overflow Area.

* The set of home addresses are called primary data area. The set of overflow addresses is called the overflow area.

* This technique is similar to chained progressive overflow except that overflow addresses do not occupy home addresses. ie overflow records are stored in separate overflow area rather than in potential home addresses for later-arriving records (refer fig)

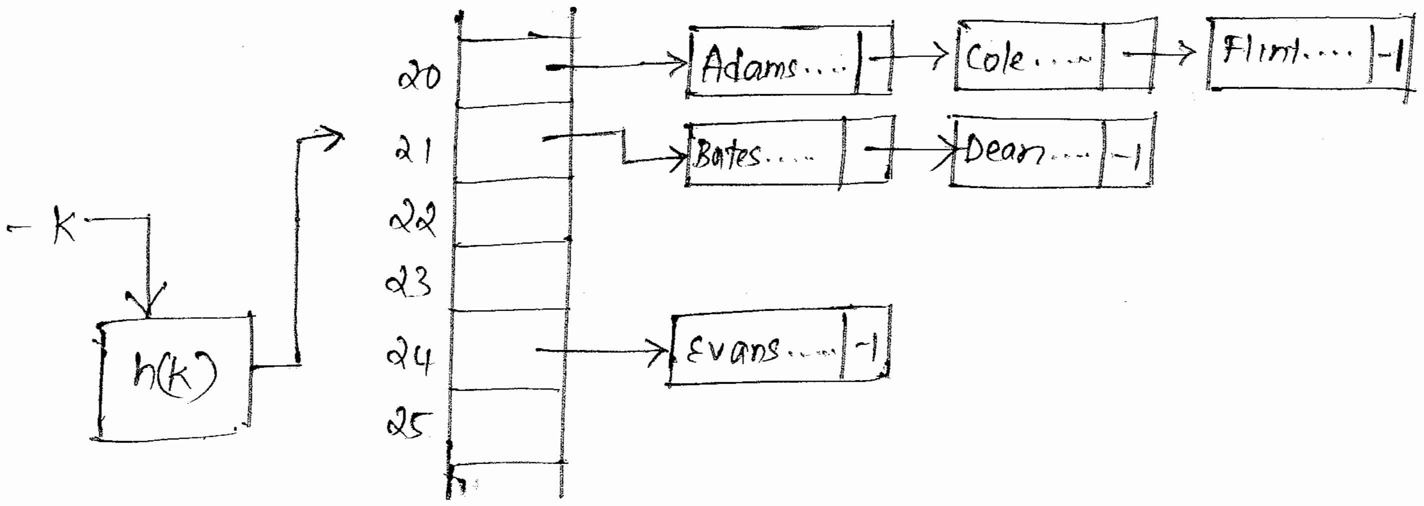


4. Scatter Tables: (Indexing Revisited)

* Here the Hash file contains no records, only pointers to records

* ie Here the file is obviously just an index that is searched by Hashing rather than by some other method.

* The term scatter table (severance, 1974) is often applied to this approach to file organisation. (refer fig)



* The Data file can be implemented in many ways; like -
  - It can be a set of linked lists of synonyms (refer fig)
  - sorted file
  - Entry sequenced file etc

PATTERN OF RECORD ACCESS

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

Ashok Kumar.K
VIVEKANANDA INSTITUTE OF TECHNOLOGY