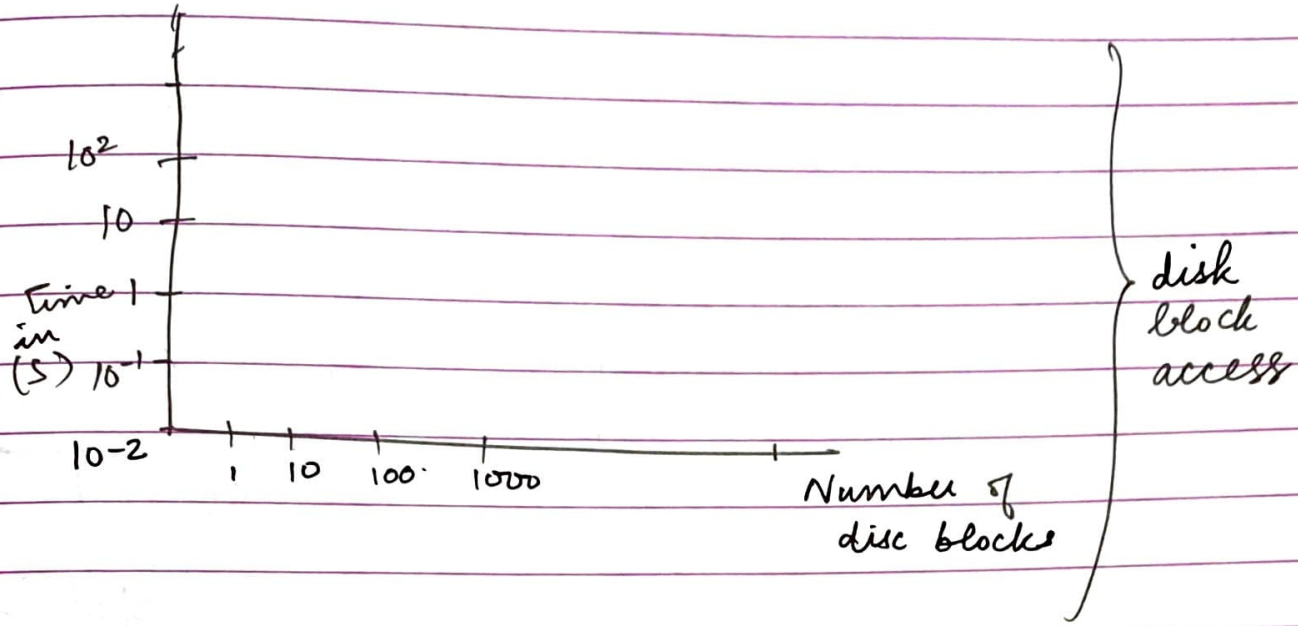
Class 3: 17th Aug' 20

Q1



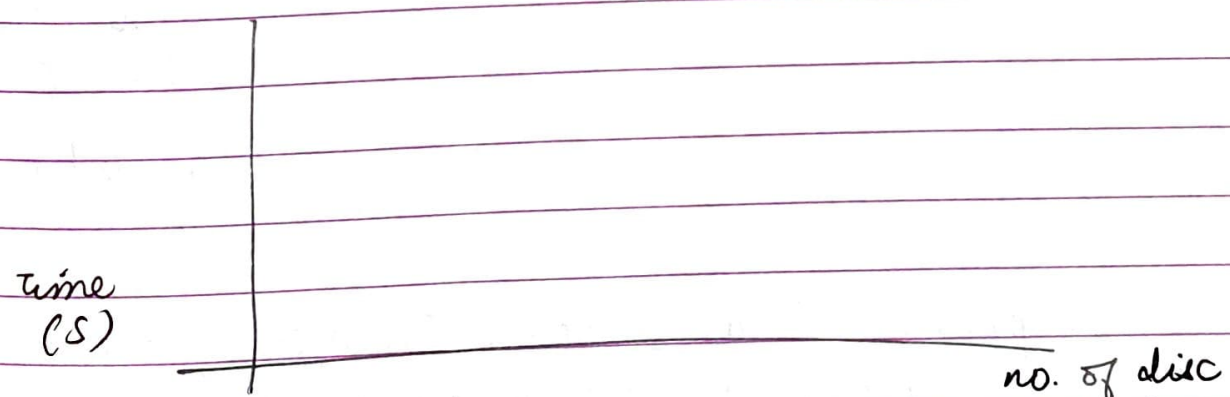
Hard disk → latency time + seek time

SDD

Reference — where you're getting numbers from.

IBM disk etc.

Q2 Given a disk / SDD & bandwidth



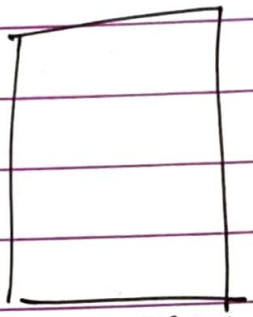
SDD to MM (main memory bandwidth)

Page No

Class 5: 24 Aug

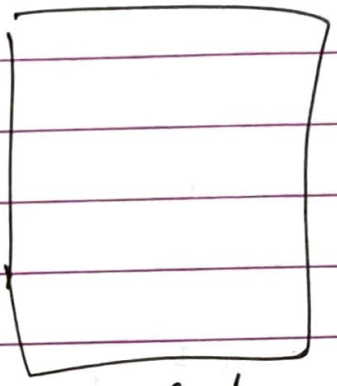
hashfile, datafile → both in external storage

File – set of records



unsorted

$n/2$



sorted

$\log_2(n)$

To get one row:

- disk block access (NOT GOOD SOLN.)
- sorted/ordered on ENo

if $n \uparrow\uparrow$

10^8 disk blocks

$n/2 \rightarrow 2 \times 10^3$ sec

10^8 dis

$\log_2(10^8)$

$= 18.4$

$\simeq 19$ disk block acces

$\times 0.001$

$= 0.019$ sec

→ Suppose ∃ a mechanism to get record given a key (ENO) — anyone block acess 0.001 secs.

CLASSMATE

Find out techniques to directly get disk block where record with key is stored.

① Technique #1 : Hashing

SEH: Static External Hashing

data is stored in disc

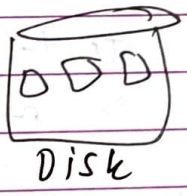
data - disc
algo - MM

unlike → Data & Program in MM

Vitter Survey — techniques when data not in MM.

---

2 aspects to this



some ID for disk blocks

1, 2, ---- m-1

Somepl, Instead of disk block there is a bucket of disk blocks.
So buck > disk blocks.

In a record,

key	

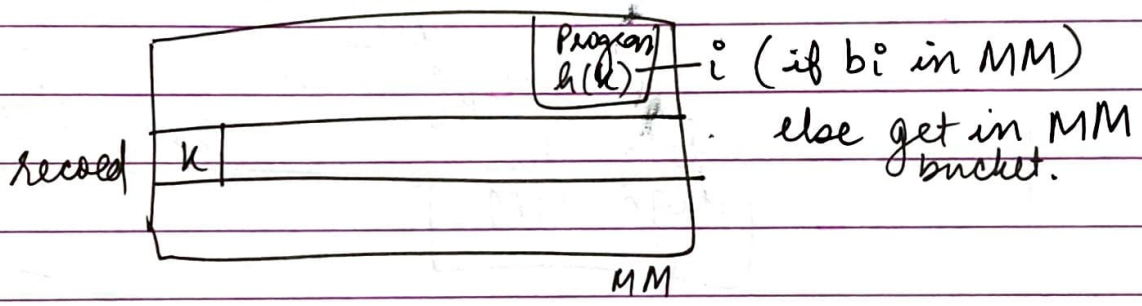
Hash(key) → will give bucket no.

Put that record in that bucket no.

→ ~~h(k)~~ So, $h(k) = i$;

→ if $h(k_2) = i$ i.e.
If bucket i is full — COLLISION

Hash funct^n



→ Computat^n of $h(k)$ is also an issue & the way it runs. (if high rate of data is coming)

→
Modern Data Systems — Twitter/Facebook
└ high speed data retrival & coming in.

→ Speed at which hashing becomes imp.

SEH — 2 components:

① Collisions

$h(key1) = i$
$h(key2) = i$ → overflow blocks (linked list)

check h(key1) & h(key2)
if not there check in overflow blocks.

worst case – search entire LL. → O(1+l)
best case – O(1)

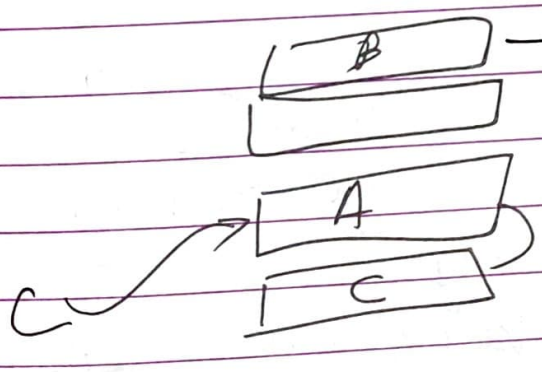
Disda (2)

(1) Ordered access is inefficient because hash table doesn't store in order

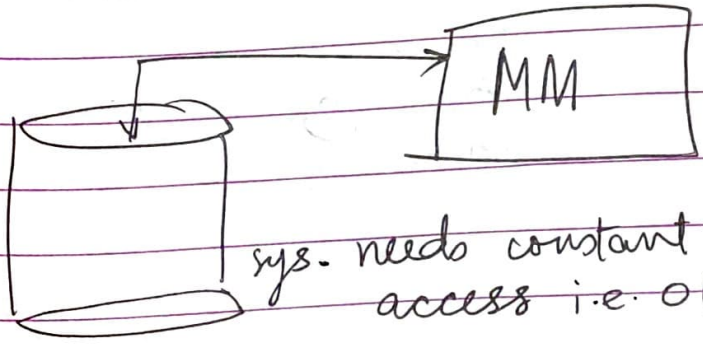
(2)

→ ~~Chaining~~ chaining of overflow buckets

#2 Linear Probing



→ each slot is a page can be



sys. needs constant no. of disk block access i.e. O(1)

and do not want variation (high)
that for one access - O(1)
& for another - O(10) - etc
O(100 - -

Handling delete

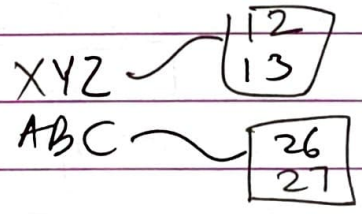
└ Place "delete" marker instead of simply deleting

#1 Tombstone marker
#2 Move all of them up (all blocks below) — how many to move & when to move becomes challenging
↳ we might move block where there were no collision - so move only ones with collision

→ Circular search is used in LP.

Non unique key

- Separate LL



→ Redundant key
└ store key & value both
ABC : 12
ABC : 13
XYZ : 26

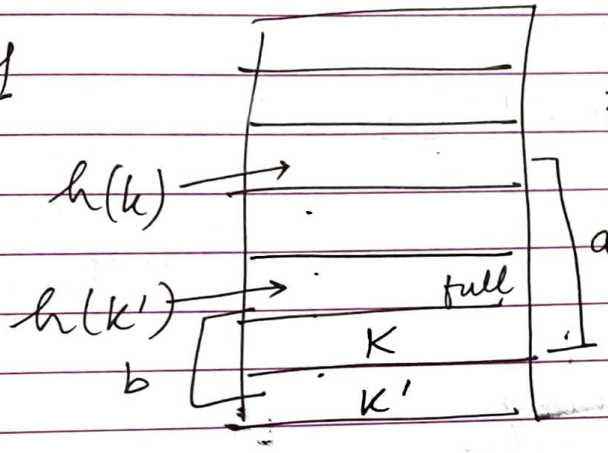
Disadv:

① Suffer for some key values to find empty slots — penalisation

side angle to LP : RHH

~~ROBIN~~

#3 ROBIN HOOD HASHING



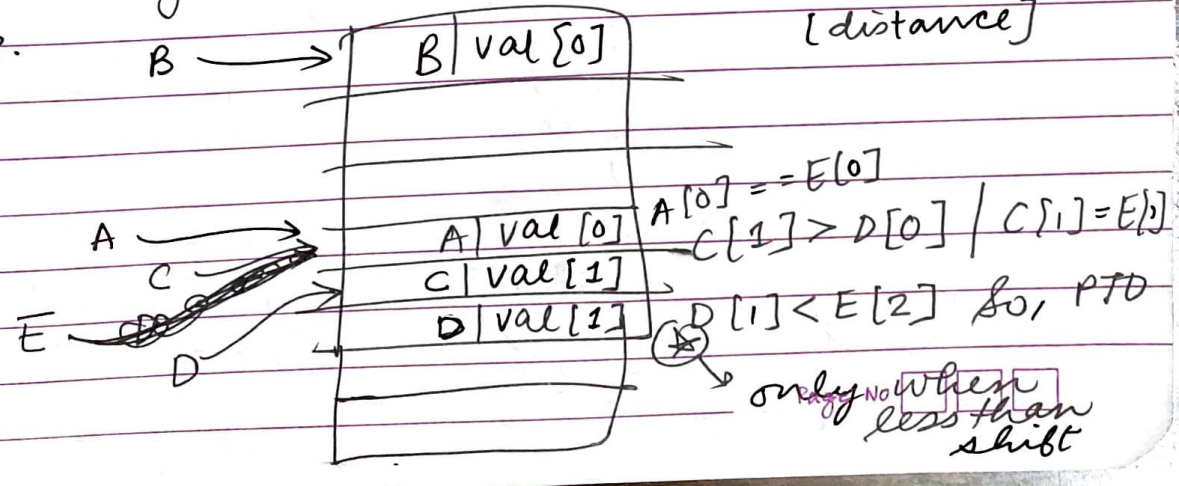
a > b

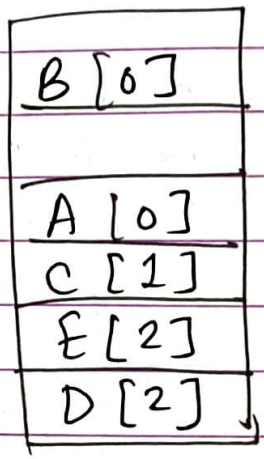
k & k' hashed to diff slots — k travelled greater dist than k' to find empty slot.

So, is there a way to ~~xx~~ normalise → max(dist) − min(dist)

So that penalise ~~more~~ the smaller dist. (Rich keys) & give them to poor keys.

eg.



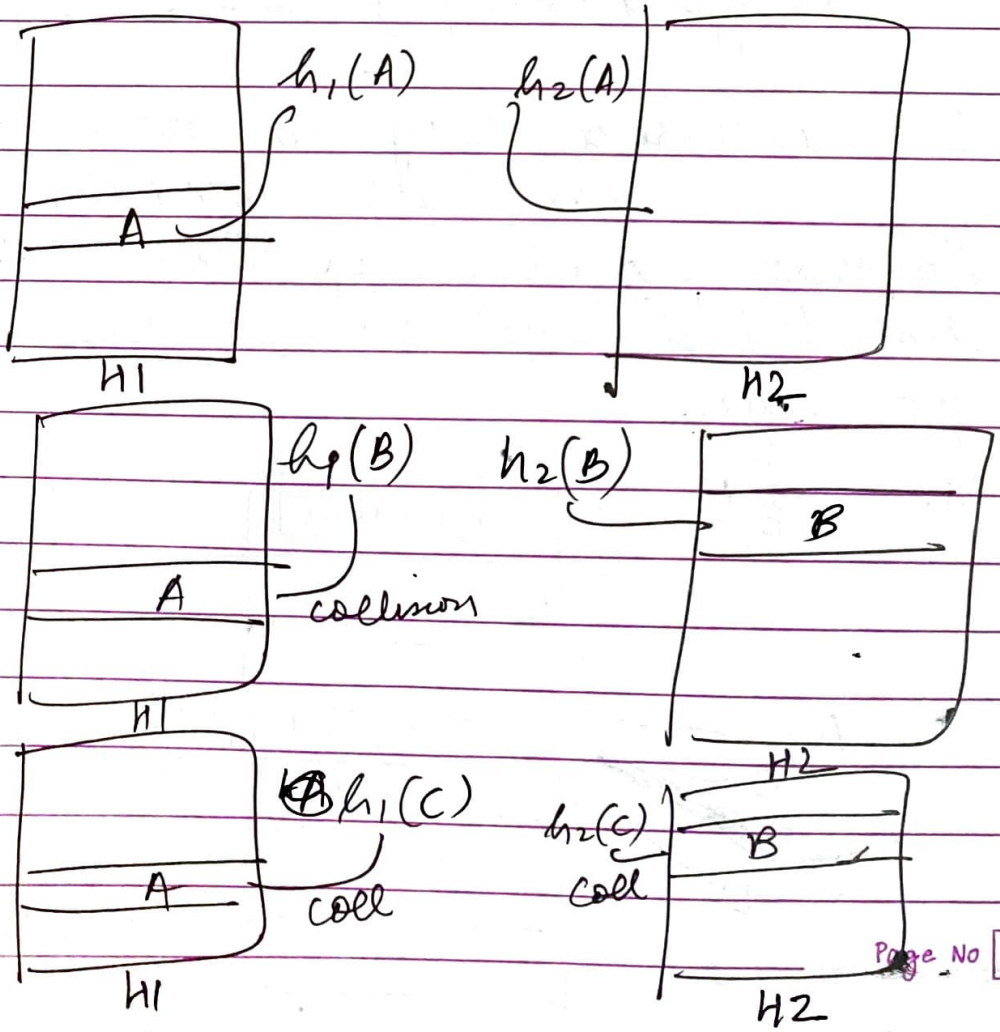


now max (gap) = 2.
- min (gap)

other wise
map(-min = 3
if E[3].

#4 Cuckoo Hashing

→ Look-ups & detetions : O(1)
→ 2 Hash Tables & 2 Hash func.



Now, put C at B (h_2)

use $h_1(B)$ → replace A & B

use $h_2(A)$ → now here put A.

disadv

① May end up ∈ cycle — same slots

↓

then use LP or overflow block.

SEH summary

① Overflow Blocks / Chaining

② LProbe Hashing

└ 2(i) Robin Hood

2.(ii) Cuckoo

→ Powerful hash fn:

XX Hash — 300 collisions for 10^8 data

Static Hashing — no. of blocks (m) are given/ required

Problem

└ wasting memory

└ Too many reorganisatn of hash table

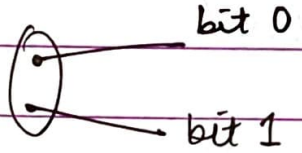
Dynamic & Extensible Hashing

└ works ε bit vector

Bucket can take two records

	Bit vector
5	0101
8	1000
15	1111
87	0111
10	1010
2	0010
3	0011
1	0001

① Start ε header block

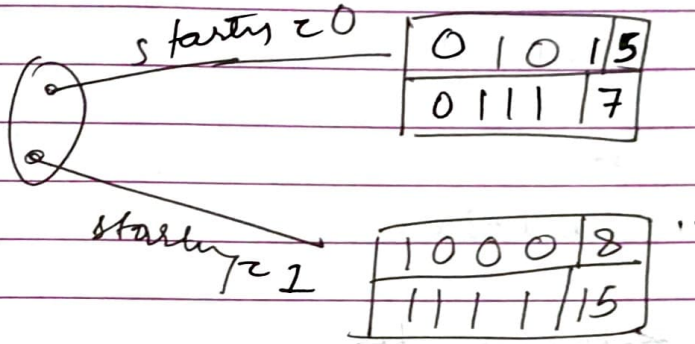


Initially one data block

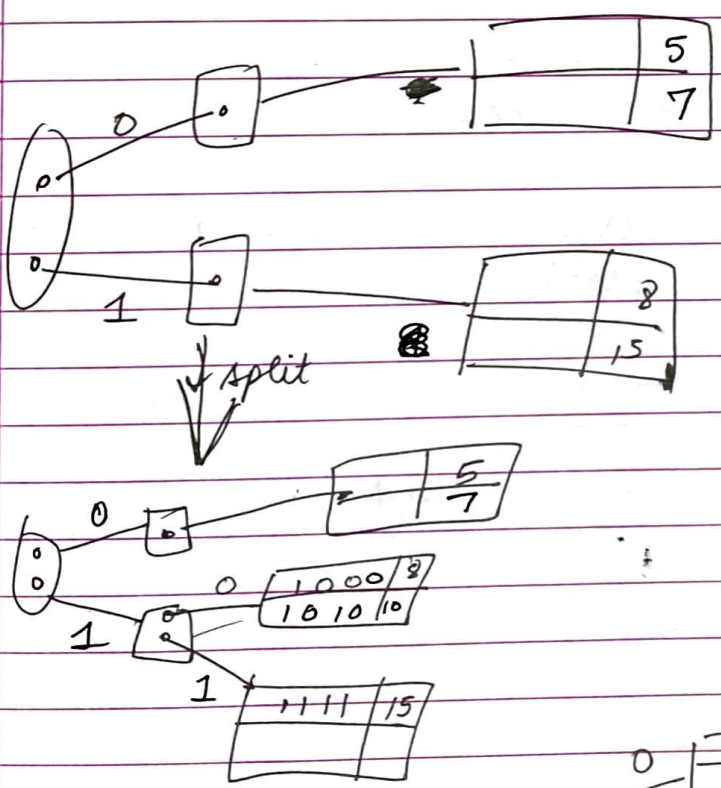
0101	5
1000	8

Then comes 15

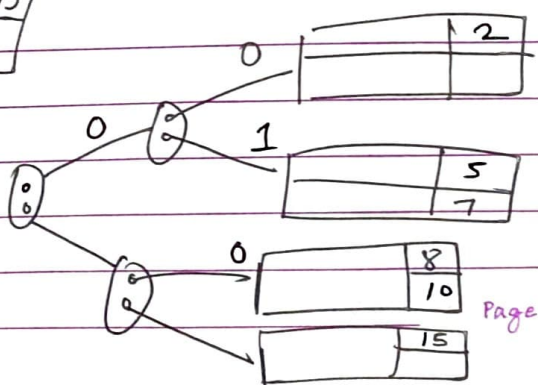
now need a header block



Now comes 10.



Now, 2



→ Start ∈ 0 bucket ✗ no header
then use 2 buckets & header using significant bits.

→ Now you have a directory (tree struct) of disk blocks.

~~So & make a dir~~

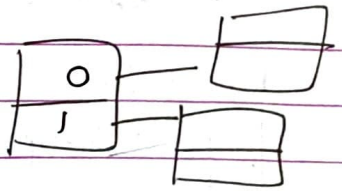
→ So if directory in MM — easier

→ any reorganisatn will happen only on affected disk blocks & hence reorganisatn lessens.

(?) → The whole directory is stored in one or more disk blocks

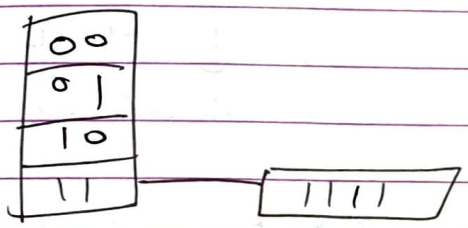
Cost : { get bit string (if key is char/string) → used for hashing
performance of Hash }

Extendible Hashing



how many bits actually being used to store

local depth
global depth
→ max bits using to store records



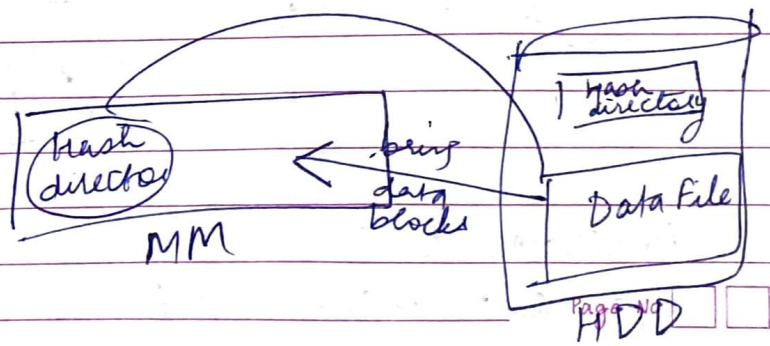
→ helps in large file storing which expand & shrink

→ D & E Hashing do not require overflow area

Linear Hashing (listen to last 15-20 min again)

27th August

How many data blocks to a file?
if file ↑↑ — inefficient
otherwise a lot of unused memory being wasted.



has the look-up tables

Hash directory

h(attribute value) = i (block i)

↓ look-up table

logical block ID

↓ look-up table

physical block

In case of Dynamic & Extendible there is a Hash directory (in MM) for shrinking/expanding the data files (where data is located).

Is there a mechanism to improve upon this shrinking/expanding?

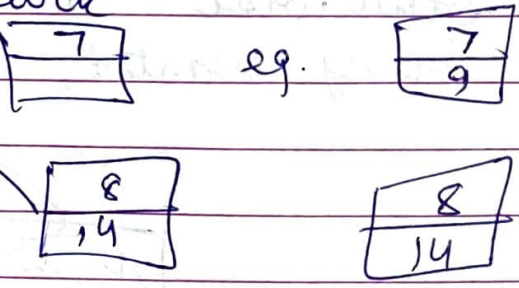
↓

Linear Hashing — by withold Litwin

Read from book ?

Intial data block | 7 / 8 | (2^0)

then comes 14 — there is no space so create another block & use (mod 2^1) to decide h_1



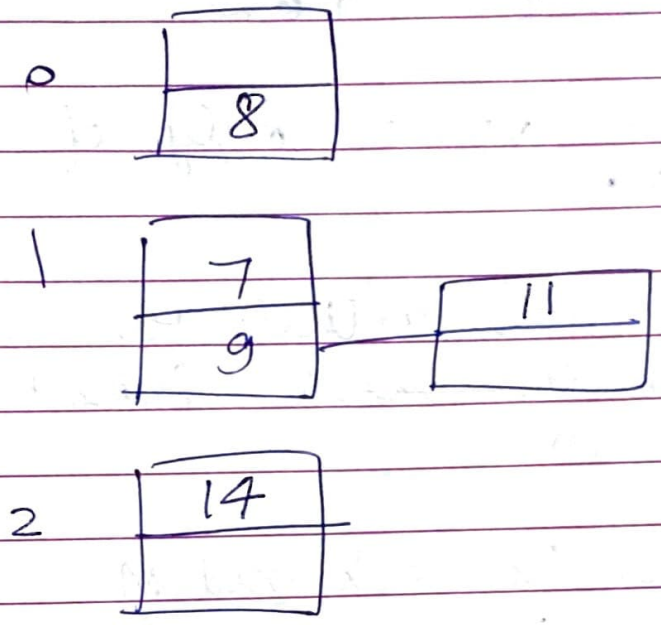
Next comes 11

└ new block needed

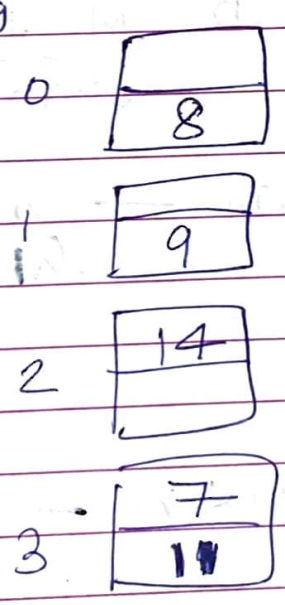
Page No

CLASSMATE

Create h_2 has $(k \bmod 2^2)$.



Next, 19 (Rearrange using $k \bmod 2^3$)



$k \bmod 2^3$ — rearrange

Current no. of blocks(m) = 2^k

if $m > 2^k \Longrightarrow$ overflow blocks

Page No

i.e. for some

blocks we use $k \bmod 2^k$ for some k $\bmod 2^{k+1}$

→ Rehash values only of the block ∝ collision

→ when you doublele the hash — only then change ~~the hash~~ forget the original hash fn

① $h_0(k) = k \bmod M$

② Collision ⟶ split into two bucks.

$n = \underbrace{M_1 - M}_{\text{original}}$ & $\underbrace{M_1 + M}_{\text{new}}$

③ if 2M buckets ⟶ new hash

$h_1 = k \bmod (2M)$

To retrieve

① $h_0(k)$

if $h_0(k) < n$ ⟶ $h_1(k)$

Buckets ∝ load factor < threshold

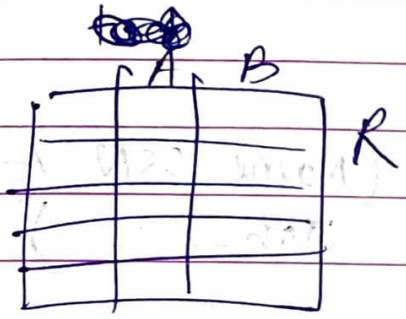
can be combined

INDEX STRUCTURES

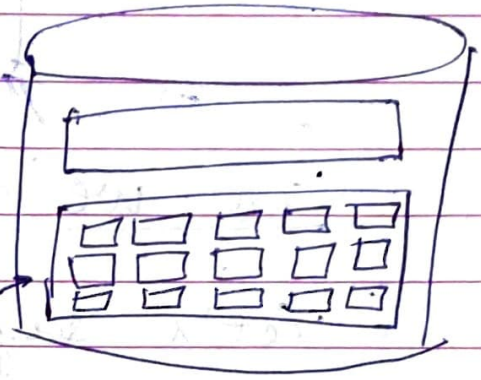
File —has→ records / tuples / rows

↓ requires

n blocks (to get stored)



Relation R



File is unordered heap

if file is ordered

$\lceil \frac{n_R}{2} \rceil$ disk block accesses

reasonable to do this

(A = V) > 1 row

get any one row

but if we want

all rows ∈ (A = V)

n_R = disk block access

→ if file is ordered on A

↳ Binary search $\lceil \log n_R \rceil$ disk block access

$\frac{\log_{10} 18}{\log_{10} 2}$

CLASSMATE

but if $B = V$ (when file is not ordered on B)

$\left\lceil \frac{n_R}{2} \right\rceil$ block access

Can we do better?

Let's say Employee (Name, SSN, Address, Job ----).

Record length $l = 150$ bytes
disk block size $B = 512$ bytes
N = number of records = 30,000 records

How many blocks do we need to store R.

given B, l

bfr = records per disk block $= \left\lfloor \frac{B}{l} \right\rfloor$ {unspanned record organisat^n}

$= \left\lfloor \frac{512}{150} \right\rfloor = 3$

each block can store 3 records

n_R = so, no. of blocks = 10,000

$n_R = 10,000$ blocks

$$n_R = \left\lceil \frac{n}{bfr} \right\rceil = \left\lceil \frac{n}{\left\lfloor \frac{B}{l} \right\rfloor} \right\rceil$$

So now if (SSN = val) — $\frac{n_R}{2}$ disk block access

— 5,000 disk block access (DBA)

if each dBA takes 0.0001 sec = 5 sec. for access

So, if unoeder file ⇒ 5 secs
5000 DBA. — (a)

So, to get one record
Job = Progeamer] — 5000 DBA
5 secs

but if all records — 10,000 DBA
i.Job ≠ Progeamer 10 secs

Now if 3×10^6 records/employees.
⇒ 1000 secs.

So, its not efficient.

#2 Lets say file oeder on SSN

to get (SSN = val)

$\log_2 10000 = 14$ disk blocks Accesses

$= 0.014$ secs

(b)

from (a) to (b) → we got it down (improvement by just having an ordered file)

#3 if file ordered on Job

to get Job = Programmer &

get one record — 0.014

get all records — 0.014 + (getting first one) seq. time on blocks

$= 0.014 + \left\lceil \frac{n_p}{bfr} \right\rceil * 0.001$ sec

if $\left\lceil \frac{n_p}{bfr} \right\rceil$ programmer blocks

i.e. { select Employee from emp
where Job = Progr }

→ this will give (n_p) — no. of prog blocks

Accesses

if $\left\lceil \frac{n_p}{bfr} \right\rceil = k$

so blocks need could be $k \mid k-1 \mid k+1$

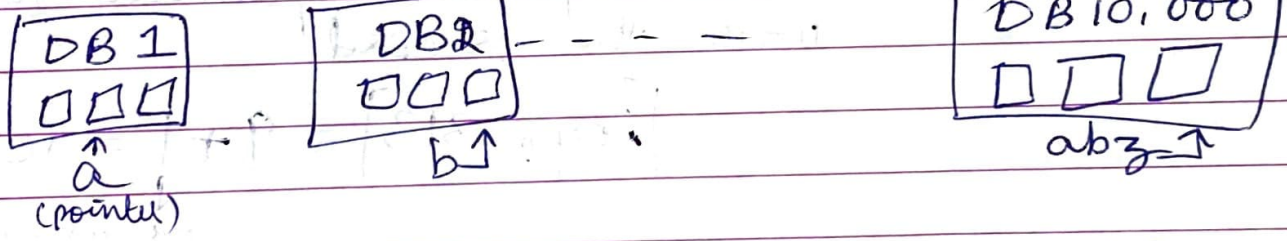
Key — SSN = value (length of key VS = 9 bytes) (length of Record Pointer $RP = 7_b$)

Lets say #1 file is ordered

key 1 to key 30,000

no. of disk blocks = $n_R = 10,000$

for getting any record $\log_2 n_R = 14$ DBA



Create a file F ← this file has VS & RP

File F → (Indexed file)

VS	RP
1	a
4	b
7	c
9998	abz

← Alw . . .

→ for this file how many blocks are needed?

PTO

Record size = 9+7 = 16 bytes

Block size = 512 bytes

How many records of F in B

$$= \left\lfloor \frac{512}{16} \right\rfloor$$

$$= 32 \ (bfr_F)$$

How many blocks $= \left\lceil \frac{10000}{32} \right\rceil = 313$ blocks

Now to get SSN = V

① Binary Search on file F

It will give a pointer to block with the record.

$$\lceil \log_2 313 \rceil = 9 + 1 = 10 \text{ DBA}$$

(1 ↓ for getting data record)

Time = 10 * 0.014 = 0.14 sec

∵ we cannot sort file in multiple columns — we create a index — for each value contains a pointer of block & records.

& hence

what if the data file is ordered & field values aren't consecutive

CLASSMATE

#2 if file is unordered

So in F file we will have 30,000 records — ∴ each SSN can be in a diff disk block

$$\left\lceil \frac{30,000}{32} \right\rceil = 938$$

$$\lceil \log_2 938 \rceil + 1 = 11 \text{ DBA.}$$

Advantage — is to create multiple indices for multiple attributes — so even if file is sorted on one column — we will get improvement of no. of DBA for all attribute

from main file

→ All index files are ordered files
→ Index file occupies less disk blocks since only 2 columns — Field Value & Record Pointer

→ Creating index file will require scan of all disc blocks

→ Index file can be stored as a hash file

Primary Index
- ordered file
- insertion / deletion hard
- non-dense index
  - all values are distinct ~~and~~ indices

block anchoring ✓

Clustering Index
- Clustered on a non-key field
- insertn / deletion straightforward
- non-dense index
  - all

block anchoring ✓

Secondary Index — ordered file (index file)
- defined on unordered data
- dense index — each entry is for each record of data file (not distinct index)
- more space, longer search time but better search time for ARBITARARY

no block anchoring

$r = 30,000$ $\quad l = 100$

$B = 1024$

b ~~bfr = $\frac{r}{l}$ = 3000 blocks~~

$$bfr = \left\lfloor \frac{1024}{100} \right\rfloor = 10$$

$$n_R = \left\lceil \frac{n}{bfr} \right\rceil = \frac{30,000}{10} = 3000$$

linear search = 1500 DBA

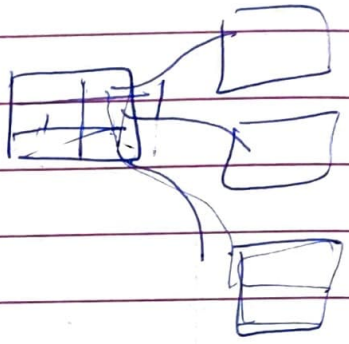
l_1

~~Primary Index~~ Blocks

→ $\left\lceil \frac{n_R}{2} \right\rceil$ DBA - unor

ord - $\log_2 n_R$ = DBA

→ $n_R = \left\lceil \frac{n}{\left\lfloor \frac{B}{l} \right\rfloor} \right\rceil$



~~Secondary Index~~

Primary Index

blocking factor (data) $= \left\lfloor \frac{B}{l} \right\rfloor$

$n_R = \left\lceil \frac{n}{bfr} \right\rceil$

bfr_1 (index) $= \left\lfloor \frac{B}{R_1} \right\rfloor$

b_2 (non_dense) $= \left\lceil \frac{n_R}{bfr_1} \right\rceil$

$DBA = \lceil \log_2 (b_2) \rceil + 1$

CLASSMATE

~~Secondary~~
Clustered Index

Secondary Index

R_i = secondary index key field

$$bfr_i = \left\lfloor \frac{B}{R_i} \right\rfloor$$

$$b_i \text{ (index blocks)} = \left\lceil \frac{n}{bfr_i} \right\rceil = \left\lceil \frac{\text{no. of records (dense)}}{\left\lfloor \frac{B}{R_i} \right\rfloor} \right\rceil$$

$$DBA = \lceil \log_2 b_i \rceil + 1$$

Page No

CLASSMATE

Multi-level indexes

also fanout

bfr_1 (index block) $= \left\lfloor \frac{B}{R_i} \right\rfloor$

b_1 (no. of blocks) $= \left\lceil \frac{n}{bfr_1} \right\rceil$ } first level blocks

x

no. of x-indexd blocks

second level blocks } $\frac{b_1}{bfr}$

third $= \frac{b_2}{bfr}$

DBA $= t+1$

where t = levels

B-Trees

fanout = p * % full

p = order of tree

$(p * BP) + ((p-1) * (DP + \text{Search field})) \leq B$

p-1 search key fields

B+

$p * BP + (p-1) * V \leq B$

$P_e * (RP + V)) + P \leq B$

DP = % * Pleaf

P = p * %

Quiz-2 (Indexing)

CLASSMATE

PRIMARY INDEX (Physical ordering)
- on ordering key field of ordered file
  - PRIMARY KEY — POINTER
  - Total entries = no. of DB in ordered file
  - sparse / non-dense
  - ✓ block anchoring
  - insert^n / deletion hard

Clustered Index (Physical ordering)
- when file is ordered on a non-key (data) field
  - index file is ordered
  - non-dense (can be dense)

Sec. Index — (LOGICAL ORDERING)
- data file: unordered on sec. index key
- on candidate or ~~compo~~ non-key attr
  - index file is ordered
  - dense

(a) when index entry (is key) — eg. Pan card
- so data file is ordered on Roll No. so unordered on Pan card no.

Sec Index

Pan card (Ordered)	Pointers to record

Index entries = Records in data

Page No.

$\log(n_R) + 1$

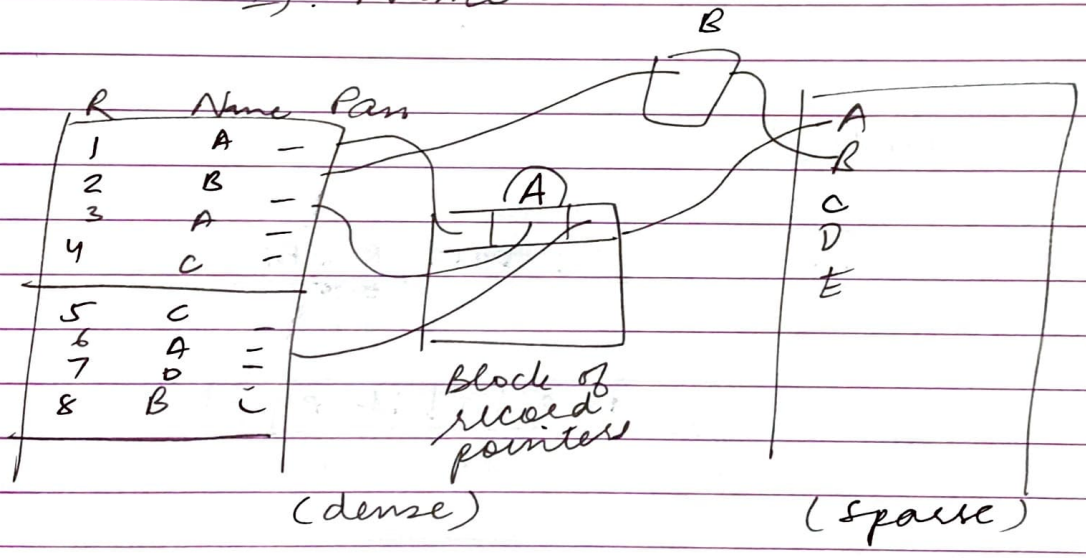
↳ number of blocks in sec index

(SEE PREV)

(b) when index entry (non-key)

i.e. non-key secondary attribute

Eg. Name



dense + sparse = dense.

IC.

x + $\log_2 N$ 1

↓ sparse

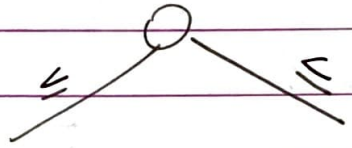
no. of record pointers in block

Multi-level Indexes

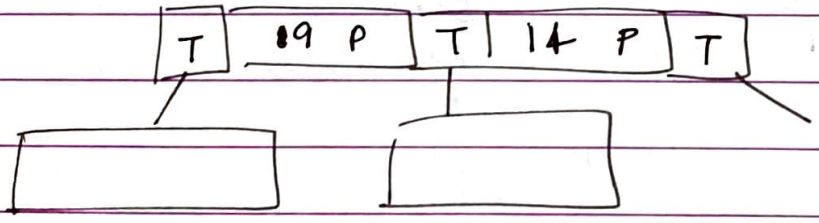
See PREV.

Dynamic Multilevel

B-Tree (balanced



Node :



Each node : at most p pointers
at least $\lceil p/2 \rceil$ pointers

if q pointers
⇒ q-1 values

Nodes are 69% full

eg. Order of tree p = 23 = $\left\lfloor \frac{B}{D+T+V} \right\rfloor$

69% full

$\Rightarrow$ p = .69 × 23 = 16

$\Rightarrow$ p-1 values = 15

So average fanout = 16.

bfr = $\left\lfloor \frac{B}{Ro} \right\rfloor$ = 16

	Pointers		Values	
Root	16	×16	15	
Level 1	256	×16	(240) 15 × 16	(values × pointers)
Level 2	4096		3840	16

B+ Tree

Internal node | P1 V1 | P2 V2 |

Leaf | V_1 P_1 | V_2 P_2 | → | V_3 P_3 | V_4 P_4 | →

next leaf pointers

order of B+ Tree

Internal node

└ p Tree pointers & p-1 values

So,

$$(p \times T) + ((p-1) \times V) \leq B$$

p = order & fanout.

Leaf node

└ Pleaf Block pointers & p leaf values

& L leaf pointers where L = P(TP)

So, $(P_{leaf} * (BP + V)) + \frac{L}{or\ T} \leq B$

~~In one leaf~~

Eg. p = 34, Pleaf = 31

0.69 × p = 23 Treep & 22 val

0.69 × pleaf = 21 DataPoi in one leaf node

		Val	Pointers
Root :	1 node	22	23
Level 1 :	23	22 × 23	5 29
Level 2 :	5 29	22 × 23 × 23	5 29 × 23
leaf :	5 29 × 23		

pointers in leaf = 21 × 23 × 529 leaf node