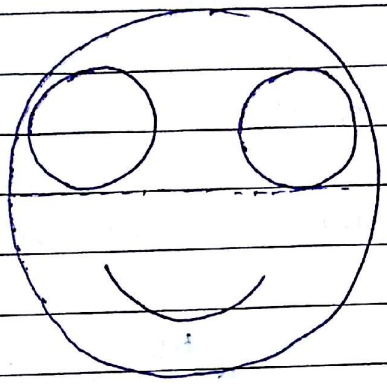
→ Every program is stored in a file, whatever instruction it may we

Camlin | Page
Date 04 / 10 / 17

Ques Draw a circle. Inside

Check (marks ≥ 50)



Revision :

→ understand requirements — System Analysts
↓ there has to be a data model that represents requirements
→ ER Diagram : converts into relational model bcoz DBMS uses th
↓
Relational data model : set of tables → relations (corresponding to entities)

Student

Id	name	grade

when you populate it with data, the data is stored in some file.

• When some device is not recognized/supported by a system, a driver installation is required

→ Cache is primary memory
Secondary : which can't be removed (magnetic disk)
Tertiary memory : which can be removed from system
(pen, flash drive, CD-ROM, DVD, Magnetic tapes)

Among

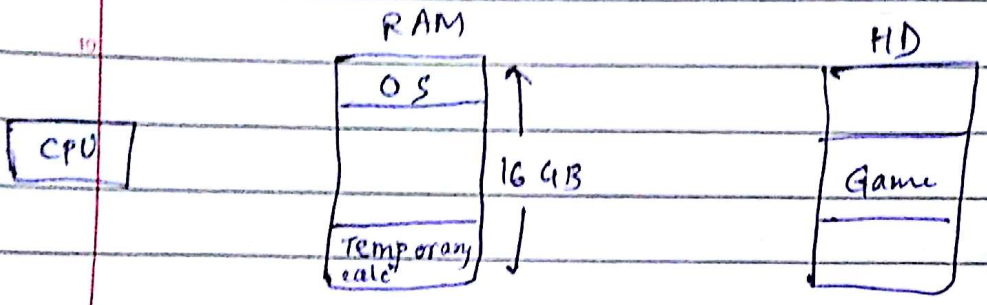
RAM (GB)

Cache (MB)

Secondary - Magnetic disk (TB)

Hard disk is of highest capacity.

→ How you play a game (64GB) on 16GB computer



App. 12 GB left for your program (game here)
when work of this is completed, it goes back & next 12 GB are brought in RAM.

FILE ORGANISATION

Memory Hierarchy :

Cache

↓ ↑

Main memory (RAM)

↓ ↑

Magnetic disk

↓ ↑

Flash memory

↓ ↑

magnetic tapes / DVD / CD

→ Programs are not stored in RAM because it is volatile memory.
→ Anything that we store is stored in the form of file.
→ The OS figures out which program is better for specific types of files.
→ Even data is stored in files.

In DBMS, we store in form of table. The physical level takes care of storing data in files.

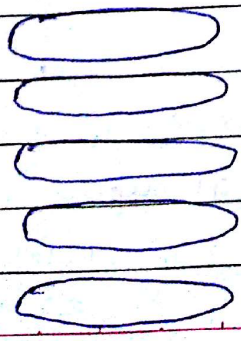
File — name
File — size
File — organization of records → value of some attribute in some entity

→ One structure variable is kind of record.
→ Whenever a file is created, a ~~f~~ it is created with a fixed size.
→ We should find optimum size of file so that neither wastage happens nor shortage.

→ Access to cache memory takes time. Because we have to search in less info. less memory while RAM has ~~g more~~ stored more info.

→ In sequential programs, first few statements are more likely to be executed (10) → brought to RAM. Out of these 10, first 4 are most likely to be executed → brought to cache
Locality of reference.

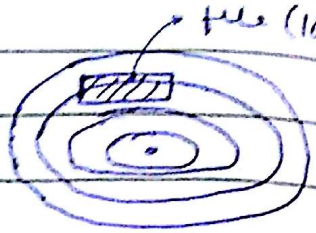
~~The~~ Hard disk : coll^n of disks.



→ Disk is organised in concentric circles on both sides called tracks

Data is stored on these tracks in bits

Only 0s & 1s are stored by magnetization, ing the form of ↻ & ↺



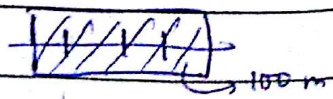
When you ask to open a file OS gives ins's to disk controller (already in drive). D C will first search the file (locating disk → track & fetching data).

Eg: Now, you've 100 pages. Now, page 53 is most likely to have data. So, go to page 53 & search the particular data. It takes lot of time to ~~read~~ search, read & write on a disk.

Eg. You've to fetch info. from page 1, 64, 2.
First you will have to go on page 1, then on page 64, then back to page 2.

What we can do ? : We can have to sort so that time taken would be less.
Disk controller does something like this. Instead of moving head from 1 to 64 to 2, it goes in forward dirⁿ initially.
Suppose it has reached 80 then, & 22 is inserted so, it will start moving backward.

* It takes lot of ~~data~~ time to fetch data from disk. So, optimization is occuring everytime.



→ Not whole 16MB is brought to RAM.
Track is divided into blocks. The block size is determined at time of formatting. It depends on OS. We generally take 1 block = 512 bytes

16 MB $\Rightarrow$ $\frac{2^4 \times 2^{10} \times 2^{10}}{2^9} = \frac{2^{24}}{2^9} = 2^{15}$

$\rightarrow$ 2^{15} blocks are needed

If file starts at block no. 100, then it will start reading. All 512 bytes will be read at same time.

Suppose 4 MB is allocated to this program. Suppose it can have only 10 blocks. Then 10 blocks will be put on RAM. Suppose only 4 blocks out of those 10 b can be stored in cache. Then those 4 blocks will be stored in cache. When executed, next 4 will come & so on...

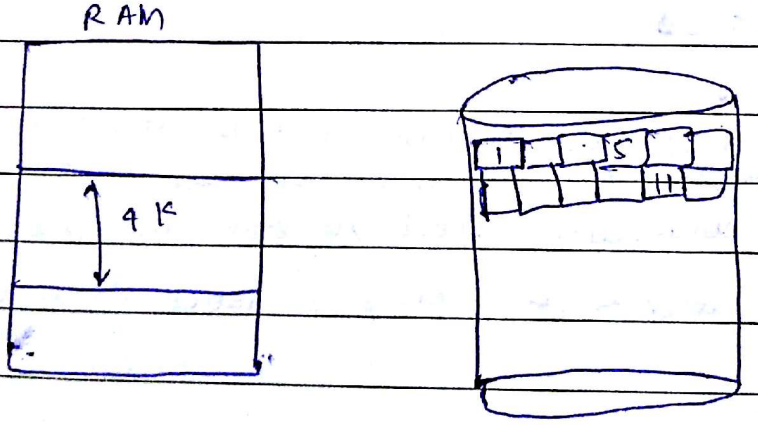
$\rightarrow$ If block is on disk (OS) : physical blocks = 512 B

brought on RAM : logical blocks (DBMS) = 4K (or 8K)

07/10/17

At a time, 4K (or 8K) can be read at RAM, although data can be in TBs.

How many blocks of data can be read in RAM = $\frac{4K}{512} = 8$



blocks 1, 5, 11 contain CSE info.

```
select *
from employee
where dname = 'CSE'
```

first blocks 1, 5, 11 will be read in RAM & then transferred.

→ Search : same as array

~~a[5]=5~~

Fixed Length | Variable Length

Eg. student | student

id char (8) | id char (8)

name char (25) | name varchar (25)

dname char (5) | dname char (5)

↓ | ↓

fixed size record | variable size record

(every record will take 36 bytes). | (may have 1 record with 18 bytes (name's size))

In C :

name (char) | name (char *)

How fixed length can be stored in a Block

Every byte has an address (Assume)

512 bytes ⇒ 512 address

↳ stored in hexadec. form.

We have fixed length : 36 bytes

↳ 0 to 35 : 1 record

36 to 71 : 2nd record

No. of records accomodated in 1 block :

$\frac{512}{36} = 14.22$

↓

0.2 means the record can't be stored as a whole in this block, we need a 2nd consecutive block to store the record.

→ To fetch a 12th record ~~block which is~~, only 1 block access is needed.

But to fetch 15th record, ~~only~~ 2 block access are needed.

We need to minimize this

→ Assume length of record < size of block.

⇒ Free list → list of free nodes

We do floor of record. The next record will be stored in next block.

→ Records can also be updation & deletion

↓

If already in RAM, we can update directly

→ Deletion of record 2.

36 to 71 is deleted. We should optimize the use of memory. 1.) We can shift & get free storage at last → O(n)

2) we have to get the location of free space so that whenever a new record comes, we can store over there

↳ We can put location of free space in stack. → 2 block accesses will be needed (1 to have stack)

↳ We can have Free list containing ptr to next free space which contains pointer to next free space (have to access only 1 block)

→ can insert in free space

→ If we have 36 bytes free space, record of only that size can be inserted there.

* It is easy to store fixed length record

Variable length :

→ We can't calculate no. of records that could be accomodated in 1 block.

416 Archie
CCE | 416 Betty
MME | 416
Any CSE

write record as they come.

Only char & numerals have ASCII values.

How to fetch the record ?

1	0	1	1	0	0	1	0

int a → 4 bytes should be reserved.
↳ how strings of 0 & 1 has to be interpreted

int : → convert into decimal & then display
float : → base & exponent
char : → ASCII value.

*) If fixed length : (prev. eg)
first 8 bytes : as ASCII value
next 25 : as ASCII value ...

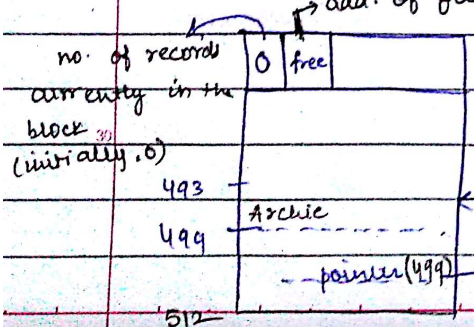
But we can't predict in case of variable-length. This is the problem with storing variable-length record.

char * name :→ needs 2 bytes for address (32-bit machine)
(4 → 64-bit mach.)

prev. eg : 8 + 2 + 25 : bytes needed

Instead of variable length string, we use address of starting point. (pointer)

Now, block will be divided into two parts



Stored in form of stacks.

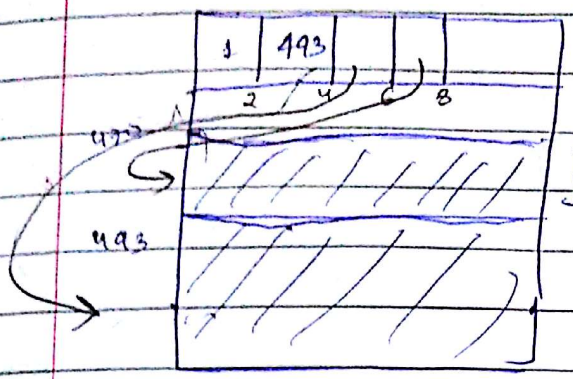
1st record : 8 bytes id 504
2 bytes pointer 502
25 bytes dname 499

Textbook : Korth

→ Insertion : O(1) time : either at free space or lowest

Deletion : O(n) : 1st search (random) ⇒ O(n)

→ 2nd record will start from 493



→ header part

8'

2nd record

next record : b/w 8 & 470

→ Deletion of Records

(Pop) (Bring pointer down)

16/10/17

Tables

↓

Files (collection of records)

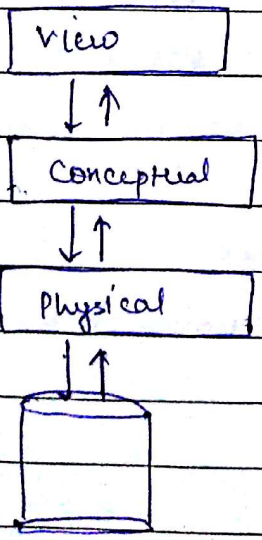
- fixed length
- variable length

Blocks

- Physical
- Logical

File Organisation

- Heaped
- Sequential
- Hashed



Eg. id — int 10 (fixed)

name → varchar (2) pointer takes 2 bytes

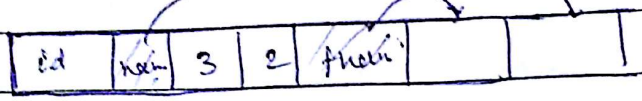
branch — int 3 (fixed)

grade — 2 (fixed)

fname → varchar (2)

fixed part : 10 + 2 + 3 + 2 + 2 = 19

20th byte will contain name



→ File is actually composed of block.
HD consist of platter of disks each having tracks
512 byte → block size (OS sets the block size)

→ CPU can only access logical block
IN Ram, diff blocks can have same add, while in Physical block, a block has a specific add. until it is reformatted.

→ Earlier, whenever we entered a record, we didn't bother about sequence { wherever there is space, it gets inserted }:
Heaped File Organisation either in free space or in the end

→ Sequential File Organisation :
a record stored in particular sequence.
either ascending / descending order of an attribut

1.) Hostel Room No. acc. to
2.) Records acc. to city in which you live (Ajmer → Bhilwara → Bikaner)
3.) Records acc. to ID's ⇒ Sequential
(1, 2, 3, ...)

→ Students are

⇒ Ajmer : for same city, there may be many records

id is key

005	001
002	002
004	004
001	005
↓	↓
Heaped	Sequential

→ Room No : ⟹ Search key

In order of room no. Records kisi bhi order me aaye, if sequence is maintained, it is sequential. Doesn't matter how ~~ord~~ records come.

A01
A04
B02
B02

→ City → : asc order of ~~A~~ city

Ajmer
Bhilwara

Zakir Hussain

: if order by city, it becomes sequential ↳ this key could be PK or not.

All ~~are~~ above are sequential

↳ Our goal is to minimize the searching time.

I want to ~~the~~ search id = 005

Heaped : $O(n)$

Sequential : $O(\log_2 n)$ → binary search

But insertion & deletion are ~~tou~~ harder in sequential

Camlin | Page
Date / /

Array

	Search	Insertion	Deletion
Heaped	$O(n)$	$O(1)$	$O(n)$ → have to search first
Sequential	$O(\log_2 n)$	$O(n)$ ↓ $O(\log_2 n) + O(n)$ ↓ ↓ search shift	$O(\log_2 n)$ ↓ we won't be shifting, we will maintain a free list.

- If records are arranged already : Sequential (take care of insertions)

If dynamic insertions (large no. of insertion) : Heaped
If no. of searches > no. of insertions : Sequential
(student dB . joins once)

Banking : heaped (no. of insertions is more)

Q. Find page with topic "Integrity constraint".

Ans. Access to table of contents → DS making search efficient taking $O(n)$ time
If not listed in table of contents :
we can go in index (already sorted), apply binary search : $O(\log_2 n)$

→ Implement index in a program :- DS
2 contents : { → Name of Topic, → Page No

1) LL : sorted order, but no binary search ✗
2) Stack, Queues : can't come to middle
3) sorted Array } can be used (Array is better because index is fixed)
4) BST

Date 23 / 10 / 17

A

5	CSE
8	CCE
4	CSE
9	CCE
1	MME
15	ECE
3	CSE
2	MME

block

B

ID	dept
8	CCE
9	CCE
3	CSE
4	CSE
5	CSE
15	ECE
1	MME
2	MME

B → sequential organisation.

wrt IDs → both are heaped.

wrt dept-name → B is sequential

search key

- search key can also be comb^n of both.

→ This is decided by database designer how files are organised (heap or sequential)

→ Both files occupy 2 blocks.

A :

To search a record, no. of blocks to be brought into RAM : 4 (worst case)

Let us assume RAM has capacity to store only 1 block at a time.

B :

A file contains 1 million records. Let say 10 records fit in 1 block. How many blocks are needed to stored in file ?

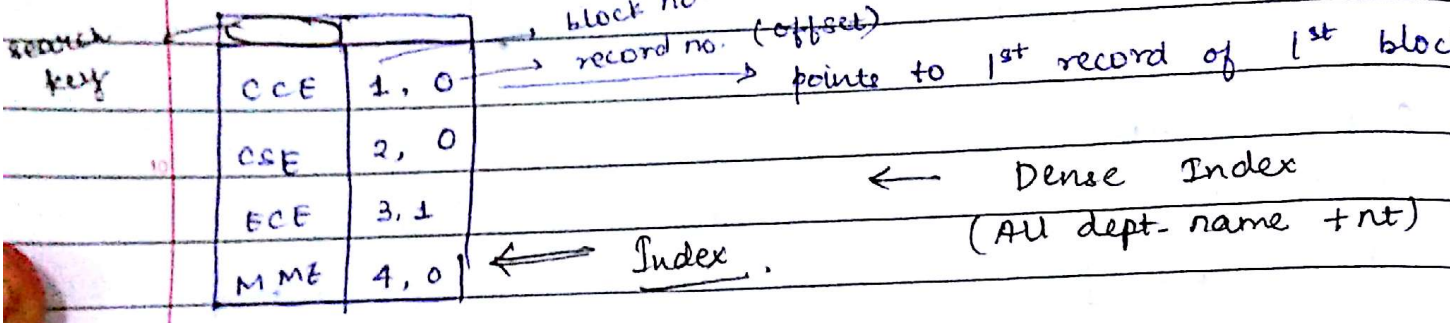
$$\frac{10^6}{10} = 10^5$$

⇒ Can we directly bring the required block into the RAM ?

→ Creating an index is in my control if I am a database designer.

→ CREATE INDEX indexname ON colname

It will give (for B)



Index is so small, it'll always be present in RAM

↓

to minimize amount of time

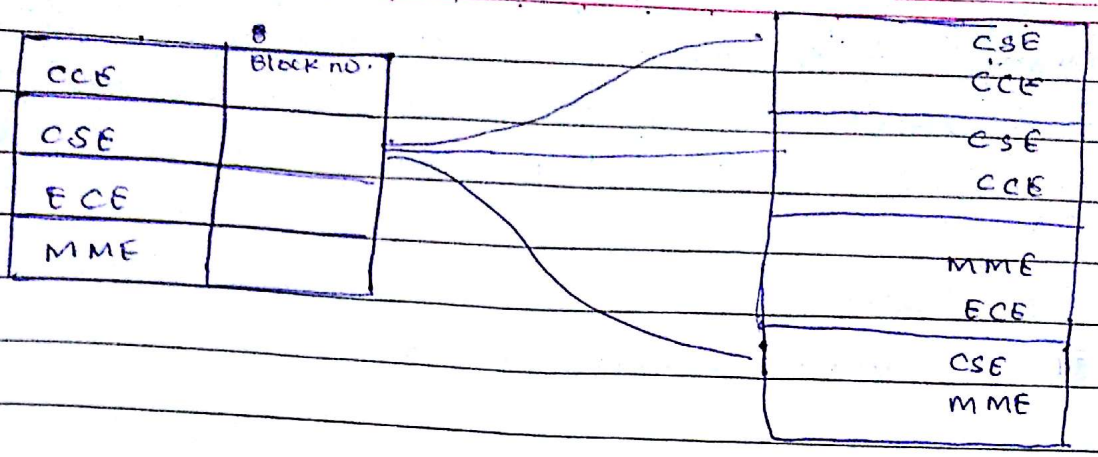
→ Now, if I want to search for a record in MME dept, I can directly bring 4th block to the RAM.
(Efficiency ↑ed)

→ Index is also a file in DBMS.
Whenever app^n starts, index is directly brought to RAM

→ ~~For files A~~ For searching CSE dept, it knows 2nd & 3rd block needs to be brought into RAM.

Index : in heaped structure (A)
Index is always sorted, so dept. name will always be sorted. So, 1st col. will remain the same.
I need to have pointer to every block in which that dept. name is present.

Date / /



→ If there is index structure, the searching complexity is of O(1) because we can directly access the blocks which have dept-name 'CSE'. We are able to get to necessary block directly.

It is not how many blocks, it is can you directly bring the block to RAM.

→ If I've index : B :

CCE	1,0
ECE	3,1

← Sparse Index (only some dept-names)

↓ works in sequential only.

→ Now, to search 4 records of 'CSE'

Since it is sorted file, record can be only b/w CCE & ECE. So, it can be either 1, or 2, or 3.

So, I have 3,1 for ECE, so, I can search only till 3rd block.

I can't use same index for heap because until we don't bring all block, we won't know which block has CSE.

not possible in heaped organization

→ Clustered index :
corresponds to key on which file is sequentially organized. dept. name pe index banaya in B

→ non-clustered index :
ID pe index banaya in B

25/10/17

Let no. of tuples = 100,000,000 → contains all entries on which index is made
∴ no. of entries in dense index = ? — (a)
Let no. of entries that fit in one block = 100
∴ .. blocks for index = ? — (b)
Let one block = 4 KB
∴ Total space for index = ? — (c)

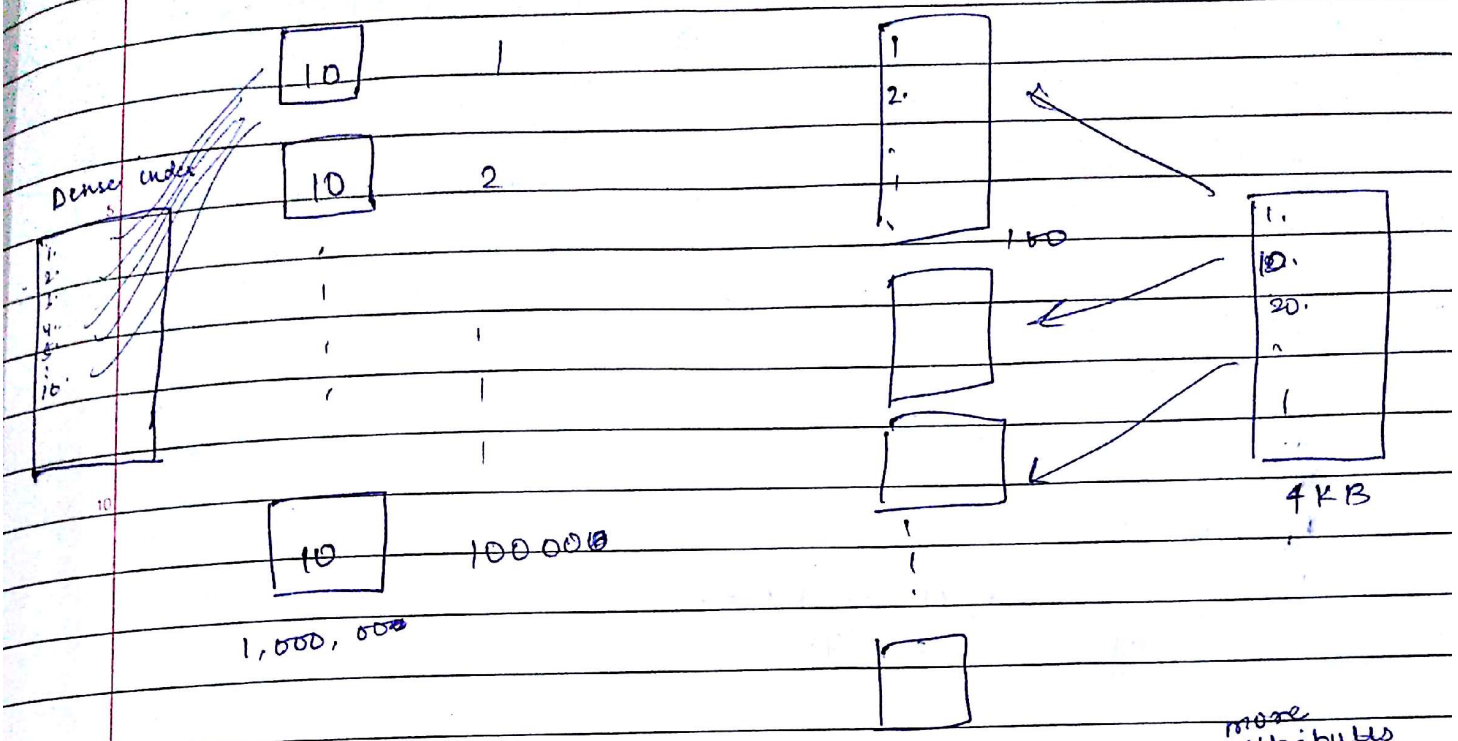
a) no. of entries = 100,000,000
as dense entries contain all entries (all tuples)

b) $\frac{10^8}{10^2} = 10^6$

c) 4×10^6 KB $\qquad 10^6$ KB = 1 GB
= 4 GB

→ If index is kept in RAM, it will take 4 GB.
If some file needs 3 index, —— 12 GB.
It is not possible to given this much memory to index alone.

→ Let there be 10^5 records & 1 ~~record~~ block has 10 records



more attributes ↓ more memory

Student relation : id, name, fname, add, Phone, city

But index contain only 2 fields ⇒ it requires less memory

1 block can have more records

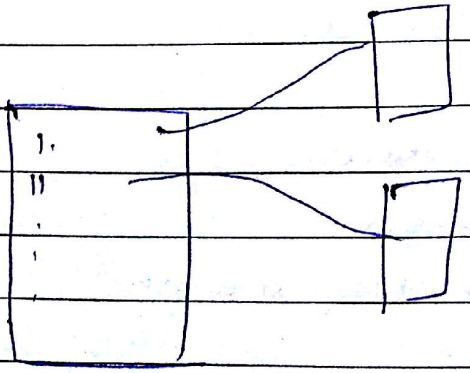
Dense index : 1st 10 entries → 1st block

Next 10 → 2nd block

:

Next 10 → 10th block

Sparse index : 1 entry for 1 block



To find id=5, find largest id <5 , ie, 1, search that block (other blocks won't contain it)

But for this : 100 entries in 1 block corresponding to 1 index → 100 blocks

→ we need 1000 blocks for all entries

→ to search id = 5, we'll go by binary search (sorted)

50
25
13
:

find largest id < 5

$\lceil \log_2 100 \rceil$ ← no. of blocks = (7)

→ 7 blocks will be brought into the memory

→ If no. of blocks = 10^6

$\lceil \log_2 10^6 \rceil$ blocks will be brought into the memory

→ If it takes 10 ms to bring 1 block into memory

→ 10 ms — 7 block

→ 10 ms → 1 search

→ 1 sec → $\lceil \frac{1000}{70} \rceil$ = 15 searches

→ It is able to search 13 records

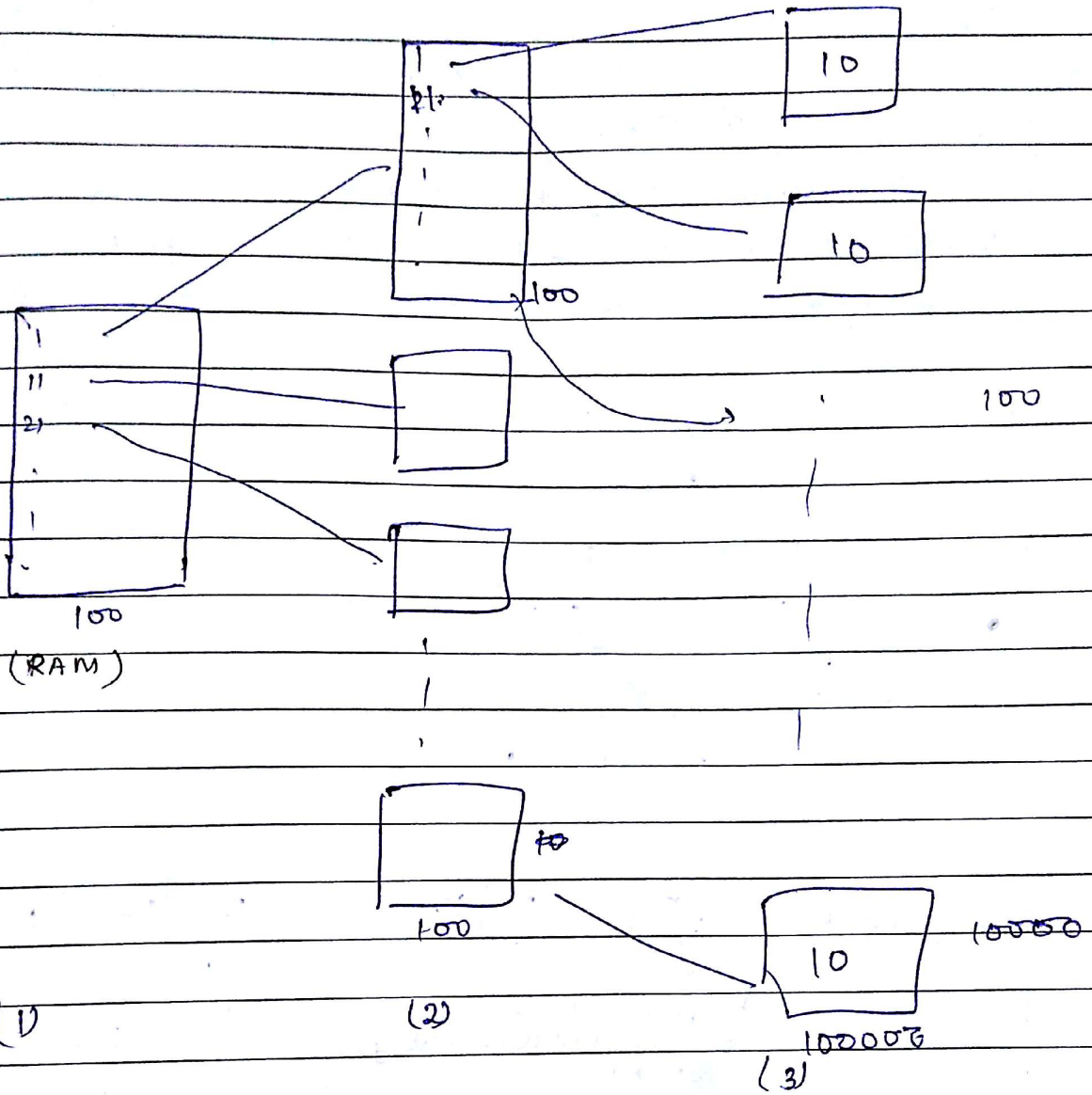
To ↑ efficiency upto 100 searches / sec, create 1 more level of index.

(We have made a tree data structure)

* Multi-level indexes → Only for sparse indexes → Only for sequential file

Camlin Page / / Date / /

Now to search id=5, id ≥ 5 ⇒ 1, largest



largest id < 5 ⇒ 1 ⇒ go to 1st in (2) ⇒ go to 1st in (3) ⇒ directly bring that block to memory

→ Only 10 ms required for 1 search.

→ This approach is good only when file is sequential

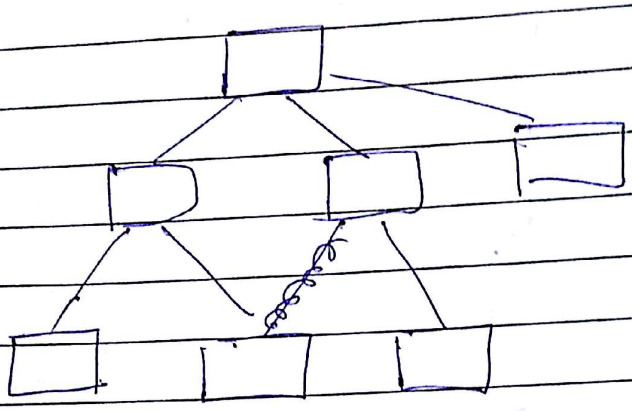
→ It works only for sparse index.

→ Can work only in case of small db or embedded systems.

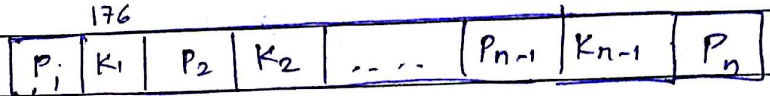
What happens if we have a heap ??

B^+ Trees

- Balanced / satisfy BST properties
- Index structure for non-sequential files
- Every node can have more than 2 children

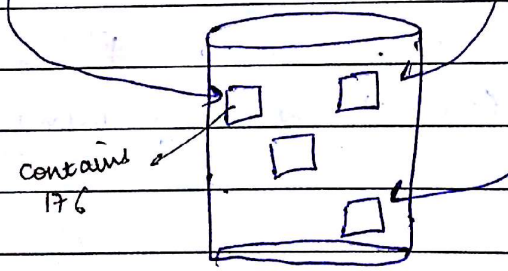


leaf node :

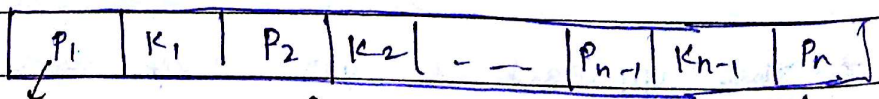


P_1 points to a block which contains entries for K_1

P_1 & P_2 can point to same ~~node~~ block also.



non-leaf node :

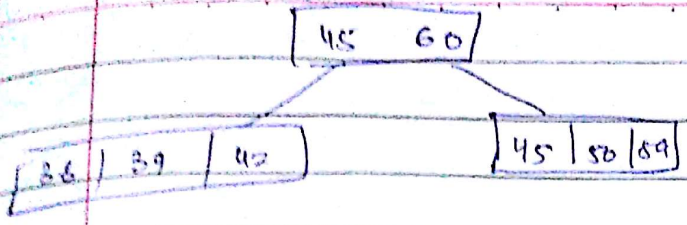


P_1 points to a node which contains values less than K_1

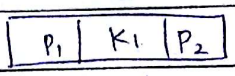
P_2 points to node ~~contains value~~ containing value less than K_2 & greater than equal to K_1

P_n points to value greater than equal to K_{n-1}

Search key (K1, K2 ...) : we are assuming to be integers.



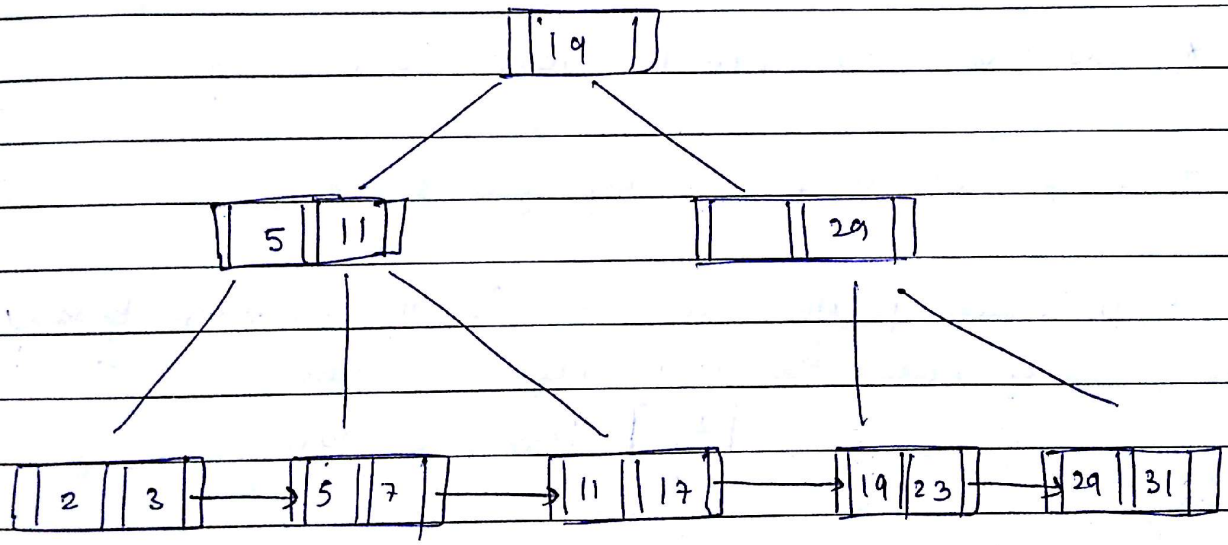
* Each node can contain (n-1) keys & n pointers ⇒ Order of tree = n

* BST is special case of B+ Trees. : 

27/10/17

. Query : SELECT x
from Student
where id = 176

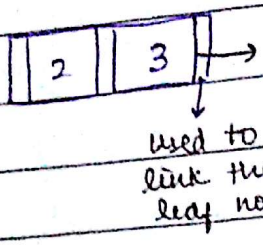
→ 2, 3, 5, 7, 11, 17, 19, 23, 29, 31



At leaves : we have all values present in file
→ dense index

At non-leaf : sparse index.

⇒ Used in case of very large files



→ To search for id = 17 → Exact match queries
search like BST, go to leaf node containing 17 & go to block where it is pointing to.

→ They are in sorted order. Because index is always in sorted order

→ If search for all ids b/w 7 & 23 → Range queries
search for 7 & using → ptr in leaf node, get all the values till 23 (as leaf nodes are dense index)

* Indexing : not good for range queries. Why?

→ Every node is pointed by just 1 ptr.

→ If Order of the tree = n (To maintain balance)
leaf nodes : (n-1) keys : max
$\lceil \frac{n-1}{2} \rceil$ keys : min.

at non-leaf nodes : We worry only about pointers
max : n pointers
min : $\lceil \frac{n}{2} \rceil$ pointers

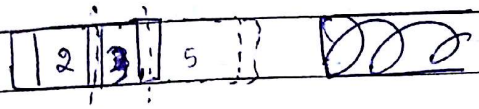
→ At root node : min : 2 → only this constraint ~~max : n~~

→ 2, 3, 5, 7, 11, 17, 19, 23, 29, 31

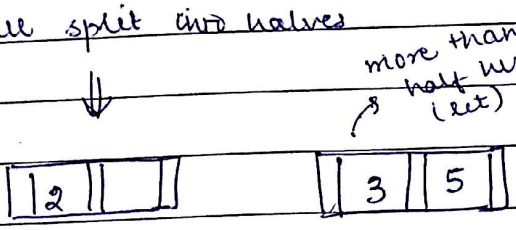
Assume $n = 3$

⇒ leaf node : max : 2 keys
min : 1 key

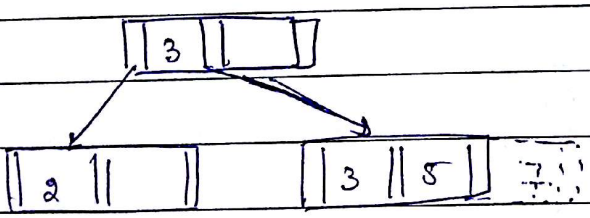
non-leaf node : max ptr : 3
min : $\lceil \frac{3}{2} \rceil : 2$



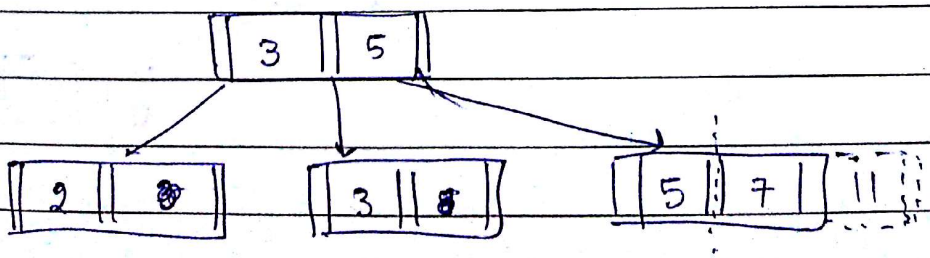
will split into halves



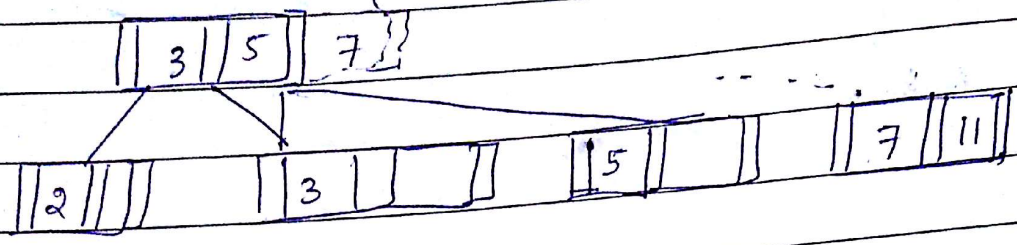
2 nodes ⇒ There has to be root node

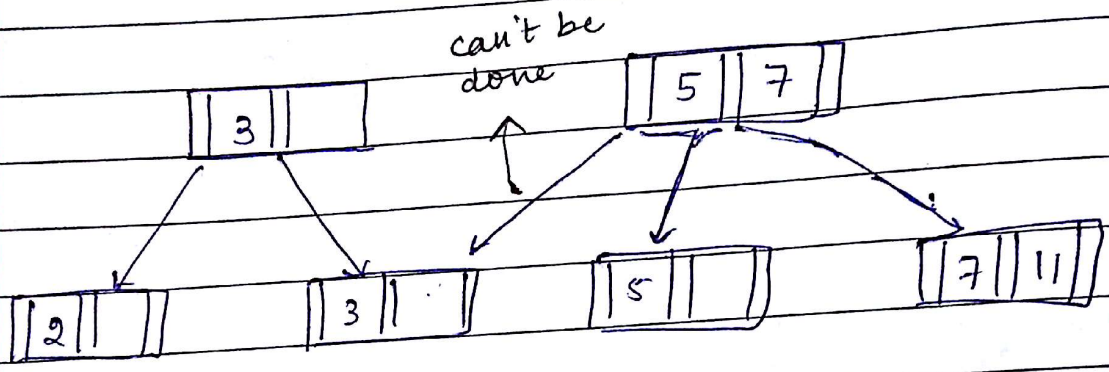


Splitted

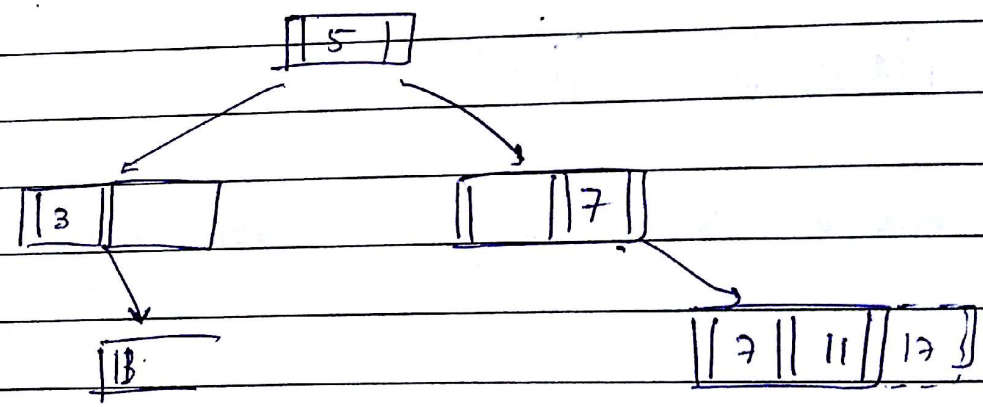


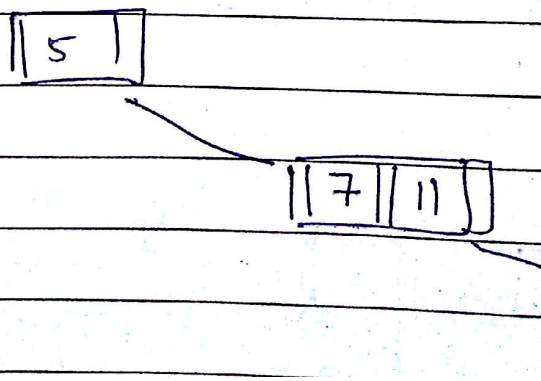
split (otherwise, 4 ptrs)





If we remove 5 from non-leaf node, it will be ok (move that 5 to up)





Eg. 2, 31, 3, 29, 5, 23, 7, 19, 11, 17

Construct B^+ Tree. Keep $n = 4$.

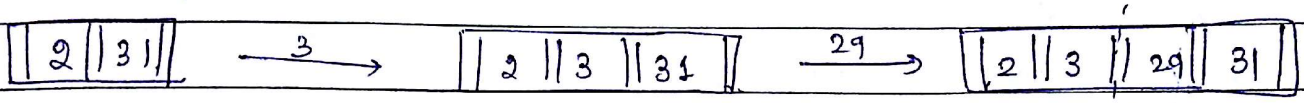
leaf node : max : 3 keys

min : 2 keys

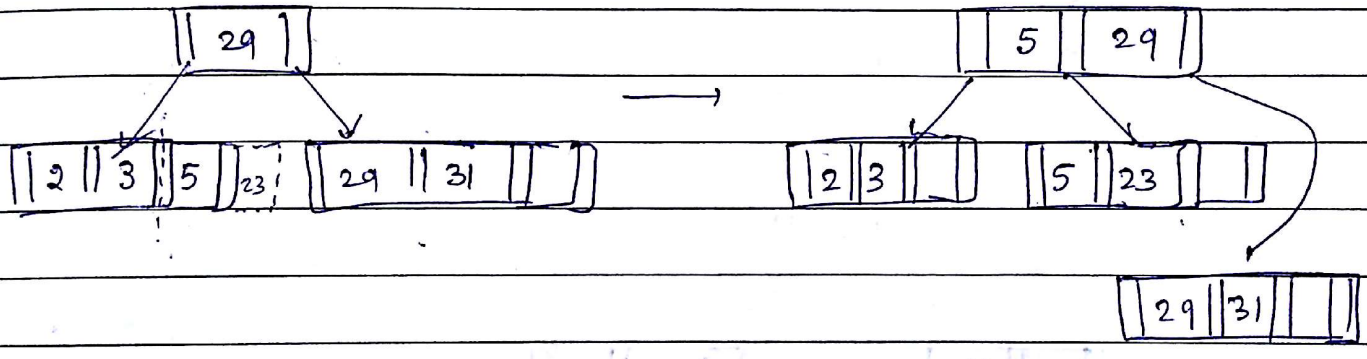
non-leaf node : max : 4 pointers

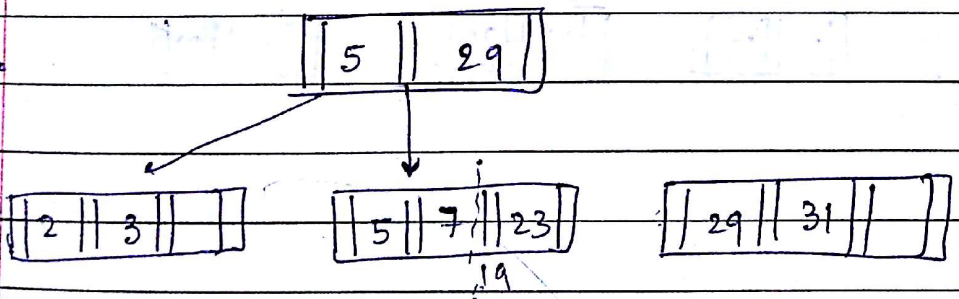
min : 2 pointer

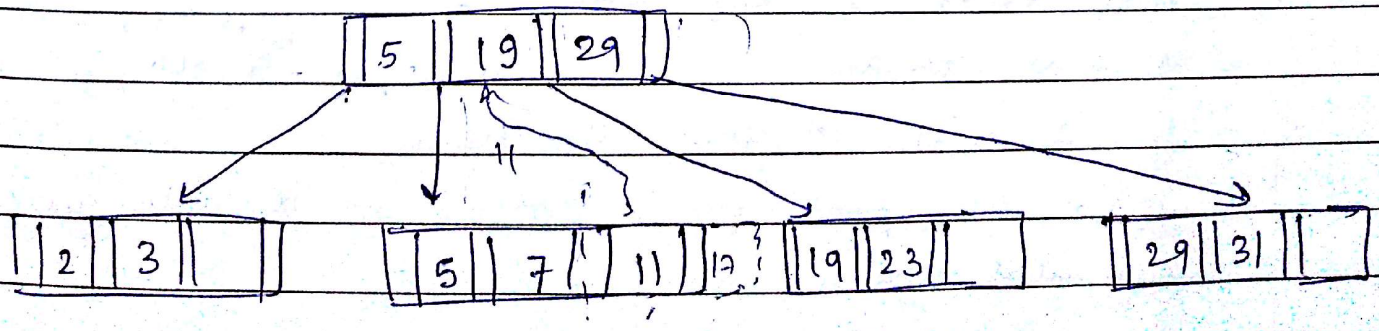
①

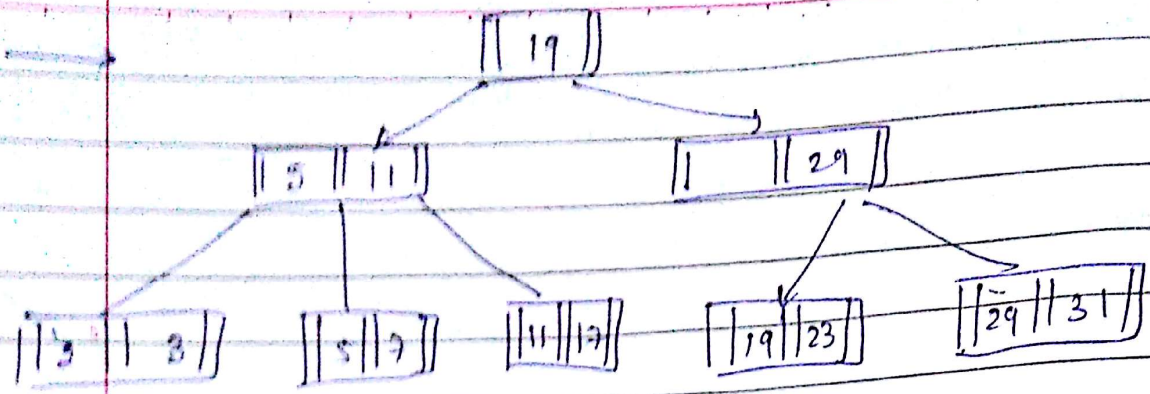


②









Deletion "delete from ---"

non-leaf

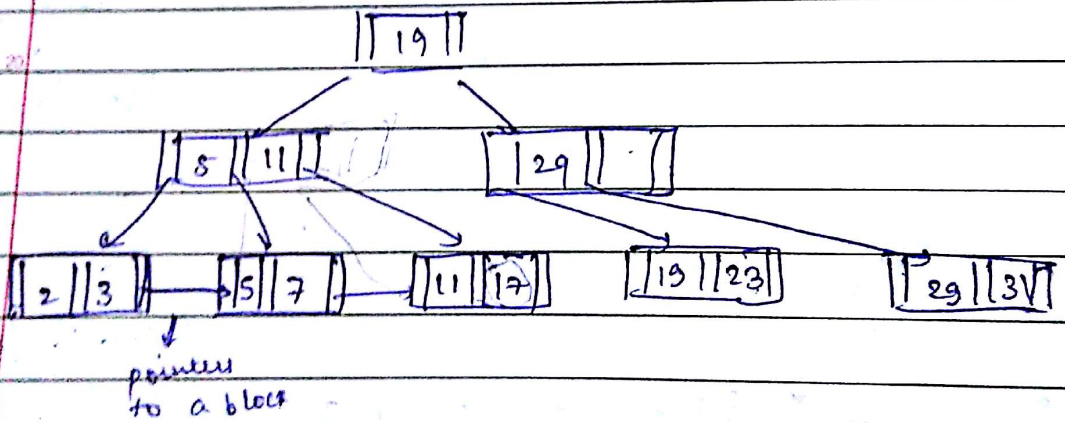
max : $n = 4$

min : $\lceil n/2 \rceil = 2$

leaf - searching key

max : $n-1 = 3$

min : $\left\lceil \frac{n-1}{2} \right\rceil = 2$



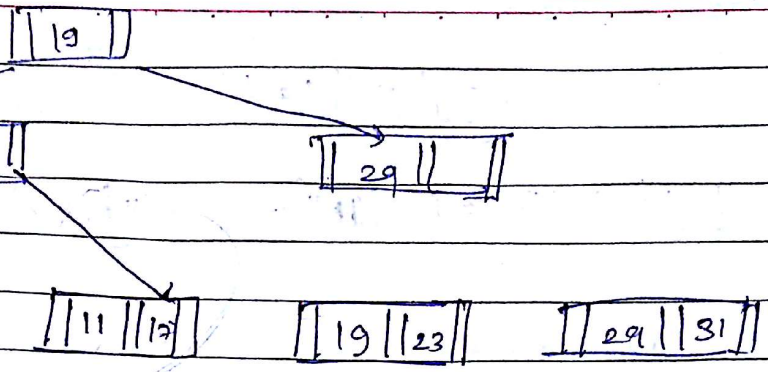
→ delete 2

min. value of leaf node = 2

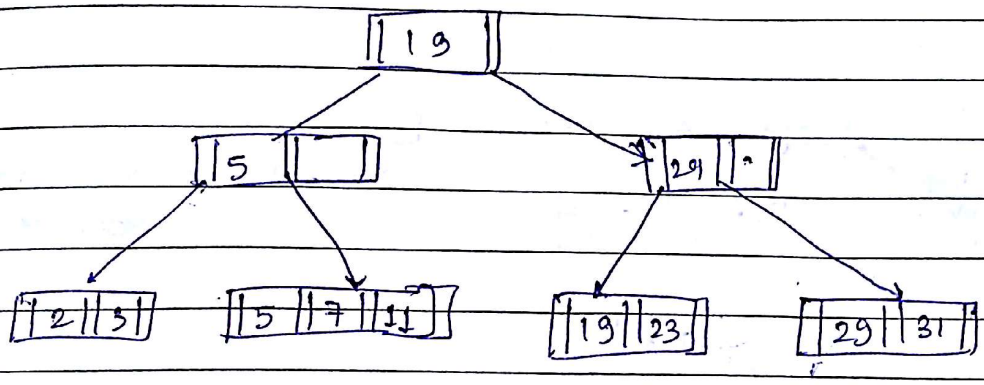
It will violate

so, it will 1st check its siblings (with less no. of nodes). It will see if ∃ a space for 5 there. So, it will get adjusted there.

When we delete the node, ~~where~~ the corresponding ptr will also be deleted



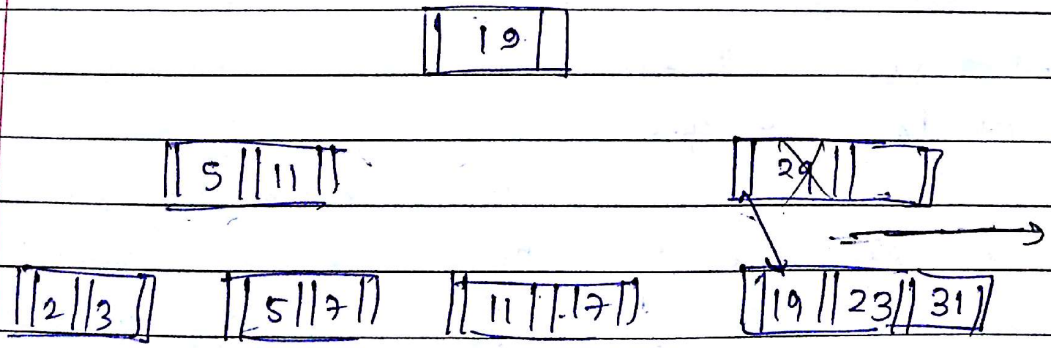
↳ delete 17 in original tree



→ if we shift to the right ⇒ changes will occur on more than 1 level

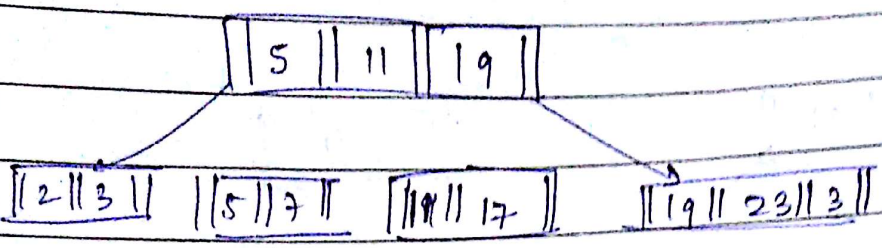
⇒ always do left shift,

↳ Now, delete ~~root~~ 29 from original

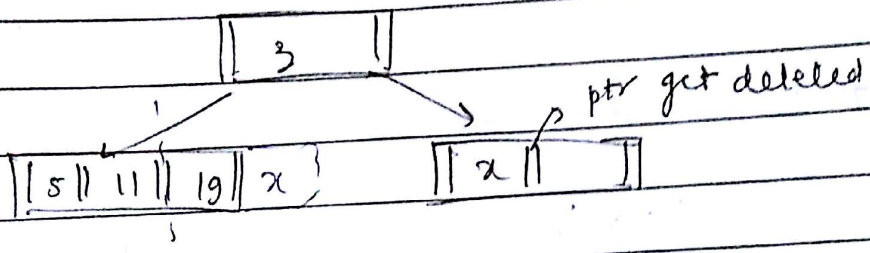


only 1 ptr. so had to delete & merge.

value at top node will come down



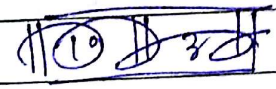
→ If we have

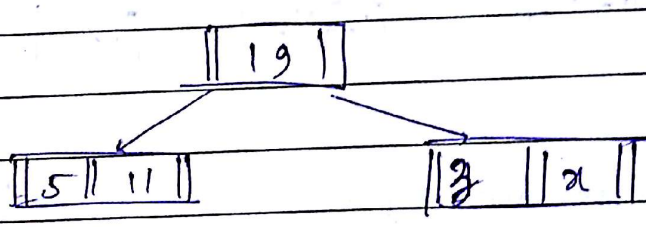


SO 1st merge & then split

↳ split के बाद वाला will go up

on top will come down





How value of n is decided in practical life ??

Q. search key (K) = 8 bytes

Block size (B) = 512 bytes

(Pointer to record) P_L = 7 bytes

(Pointer to block) P_B = 6 bytes

Let the order of tree = n

every node :

non-leaf nodes : n ptr for blocks

& search keys = n-1

$\Rightarrow n * 6 + (n-1)\,8 \leq 512$ → 1 node is 1 block

$\Rightarrow 14n \leq 520$

$\Rightarrow n \leq 37.14$

n = 37 ?

leaf node : (n-1) ptrs,
(n-1) search keys
1 ptr to next block

$(n-1) * 7 + (n-1) * 9 + 6 \leq 512$

$16n - 10 \leq 512$

$16n \leq 522$

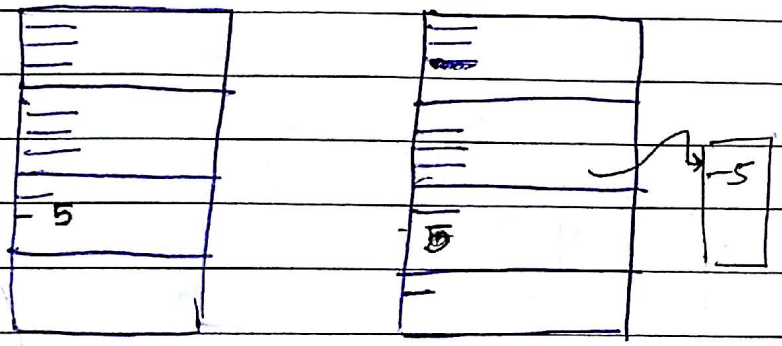
$n \leq \frac{48}{32}$

⇒ we would take $n = 32$.

1/11/17

Hashed File Org

heap ~~sequential~~ sequential ~~heap~~ hashed



→ can give rights of a particular view to a particular person.

→ let say id = 5 has to be inserted

→ each is 4 block long
↙
3 records / ~~block~~ (7 already been inserted)

→ sequential if 5 has to be inserted in 2nd block (full), new block is created & linked to 2nd block
↓
Overflow block

heap : 3 block access needed
sequential : O(n) + linked list

→ B+ Tree can also be created for sequential

Hashing

$h(k) \rightarrow b$

∴ no. of blocks = 4, we generally do a % 4

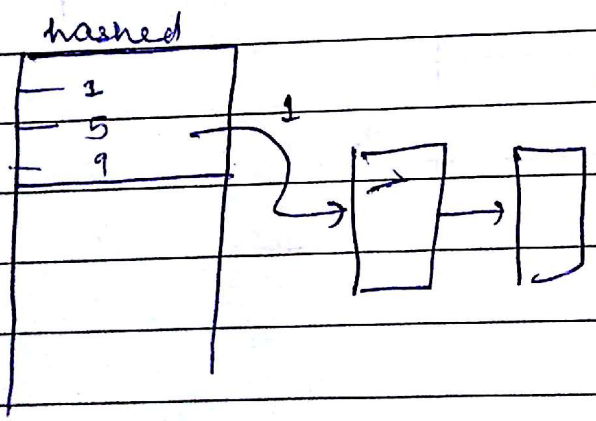
id % 4 = 5 % 4 = 1 ⇒ insert in block 1.

Since we can directly get block, so no indexing is needed (using hash func")

→ Properties of hash func" ~~should~~ be uniform & random.

⇓

should point to blocks uniformly



hashing

→ linear probing → used in next free slot

→ Chaining or open addressing (not create new block & link it to old block)

→ In DBMS, we use chaining only (open addressing)

⇓

needs more space than original hash table.

Hashing in DBMS

disadvantages of Open-addressing

a) as soon as 80% of block is filled, new block is created
→ large memory

b) if records are inserted dynamically, we can't estimate size of file (% n, n = ??)

In DBMS, we will talk about Dynamic Hashing (not static) generally we do % ~ if we take array of size 10 → %10 then we don't know size of block

	K		K mod 64	% 3 (6 bit pattern)
Eg.	288	——	32	100000
	8	——	8	000008
	1064	——	40	101000
	120	——	56	111000
	148	——	20	010100
	204	——	12	001100
	641	——	1	000001
	700	——	60	111100
	258	——	2	000010
	1586	——	50	110010
	44	——	44	101010

64 | 700 | 64 | 60

bucket size = 4

(Whatever we have done using blocks, we will use bucket)

we will convert it into corresponding bit strings

→ 1 bucket contains 4 records

11 records → 3 buckets needed → 2 bits needed to represent the block

00	01	10	11
8	148	288	120
204		1064	700
641		44	1586
258			

→ In static, we took % 64, so we would have taken 64 buckets

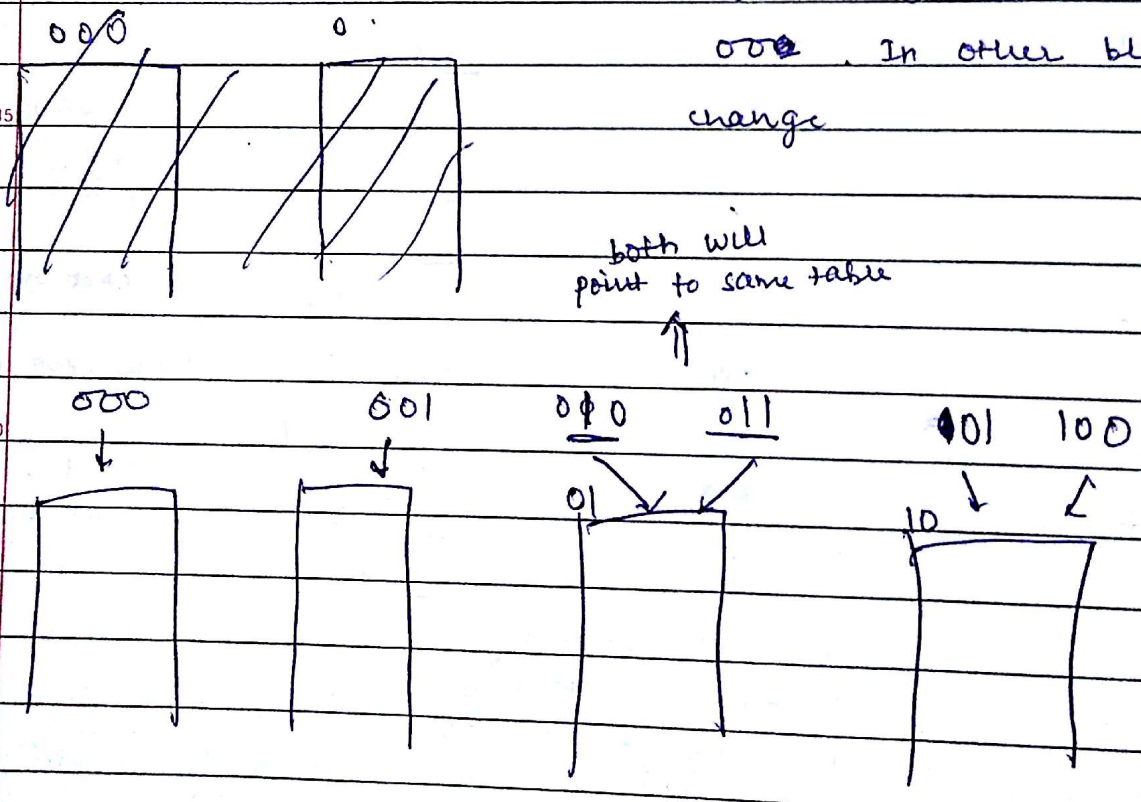
→ Look 1st 2 bits & add in corresponding buckets (either take LSB or MSB)

New record : 68 $68 \% 64 = 4$ → 000100

it should be added in '00' bucket (already 4 records)
Now, we need 1 more block → 3 bits will be needed now

That doesn't mean we need all 8 blocks

* look at 1st 3 bits starting with 000. In other blocks no change



If 000 is also full, it will again be splitted into 4 bits