

# Disk, Magnetic Disk

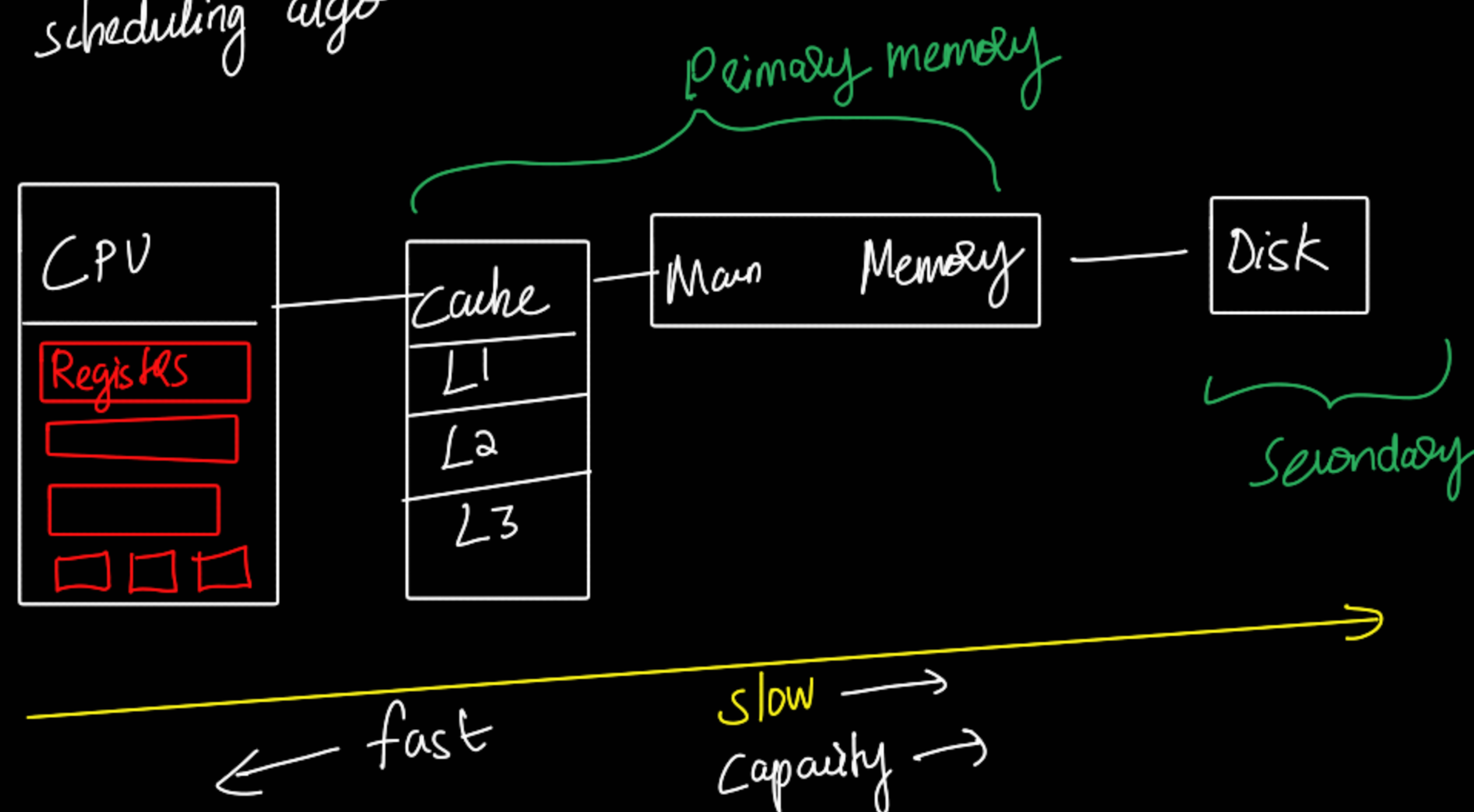
**Database Management and Warehousing:** ER-model, relational model: relational algebra, tuple calculus, SQL, integrity constraints, normal form, file organization, indexing, data types, data transformation such as normalization, discretization, sampling, compression; data warehouse modelling: schema for multidimensional data models, concept hierarchies, measures: categorization and computations.

Database <sup>physically stored</sup> → Hard disk → Structure  
→ Performance  
→ Disk scheduling algo

## GO CLASSES Database Management System GO Classes

Although a database system provides a high-level view of data, ultimately data have to be stored as bits on a storage device. A vast majority of databases today store data on magnetic disk and fetch data into main memory for processing.

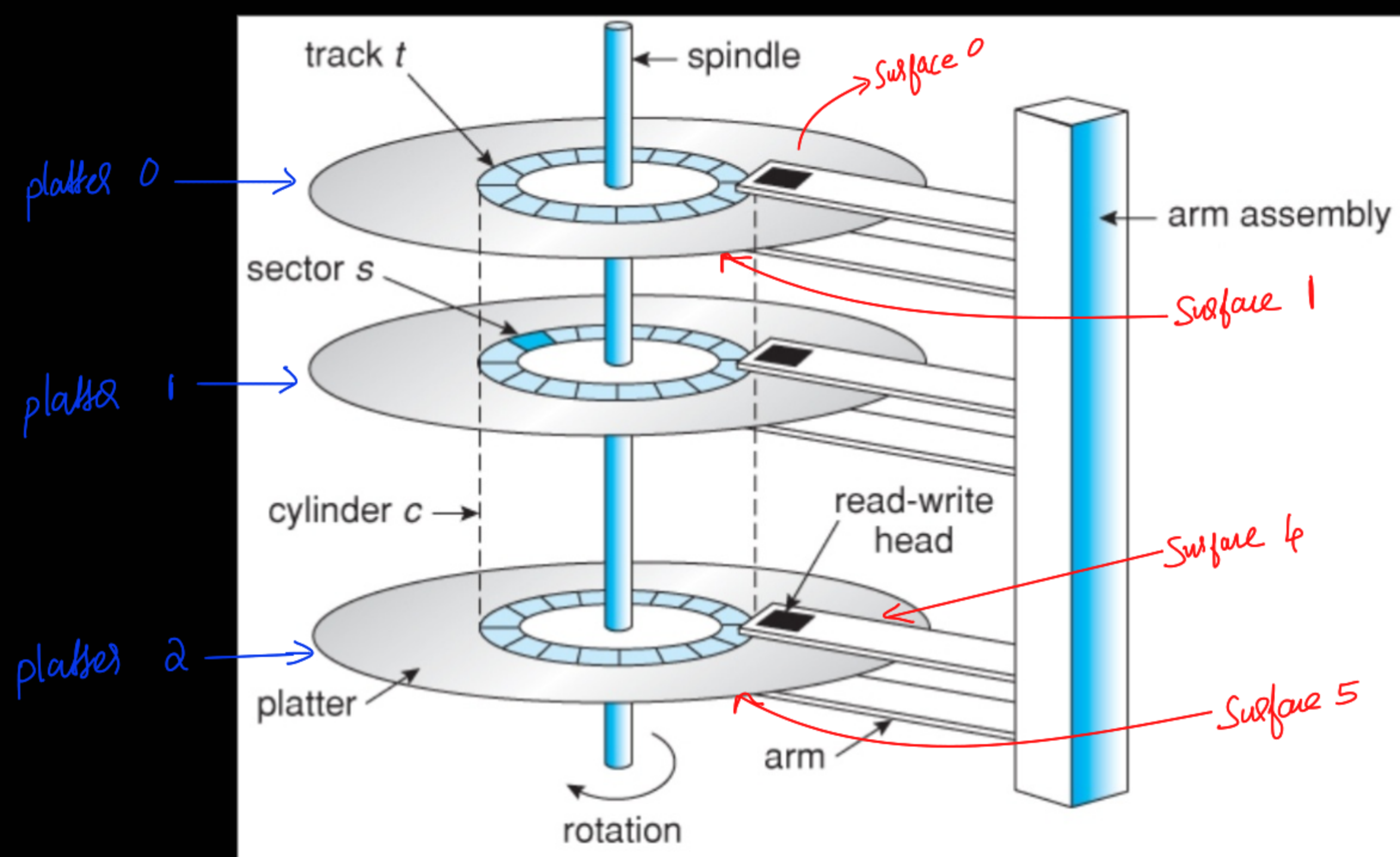
The physical characteristics of storage devices play a major role in the way data are stored, in particular because access to a random piece of data on disk is much slower than memory access: Disk access takes tens of milliseconds, whereas memory access takes a tenth of a microsecond.



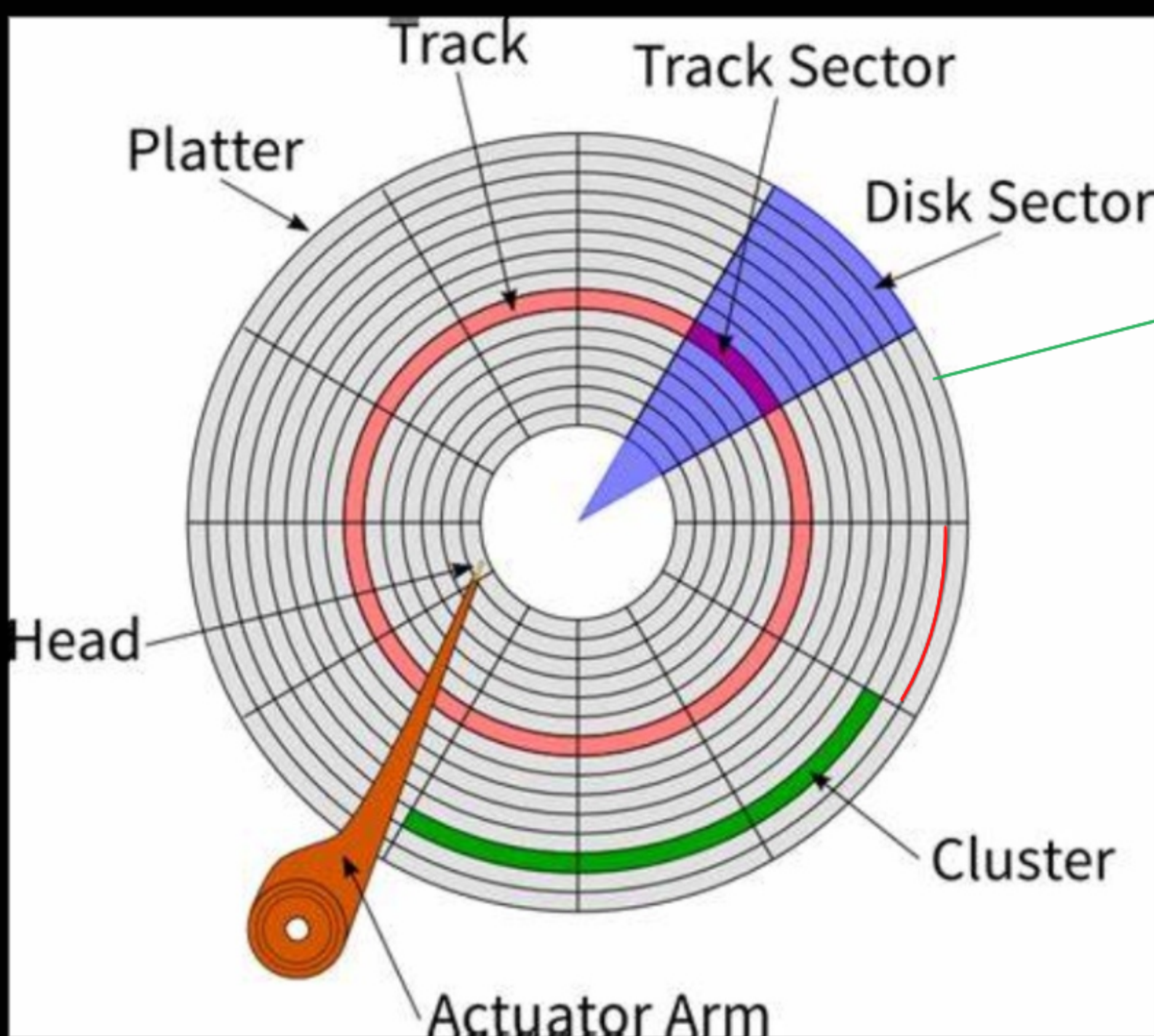
## Secondary Memory Vs primary

- More space
- Slow
- non-volatile
- Random access
- Access time varies

## Structure of Hard Disk





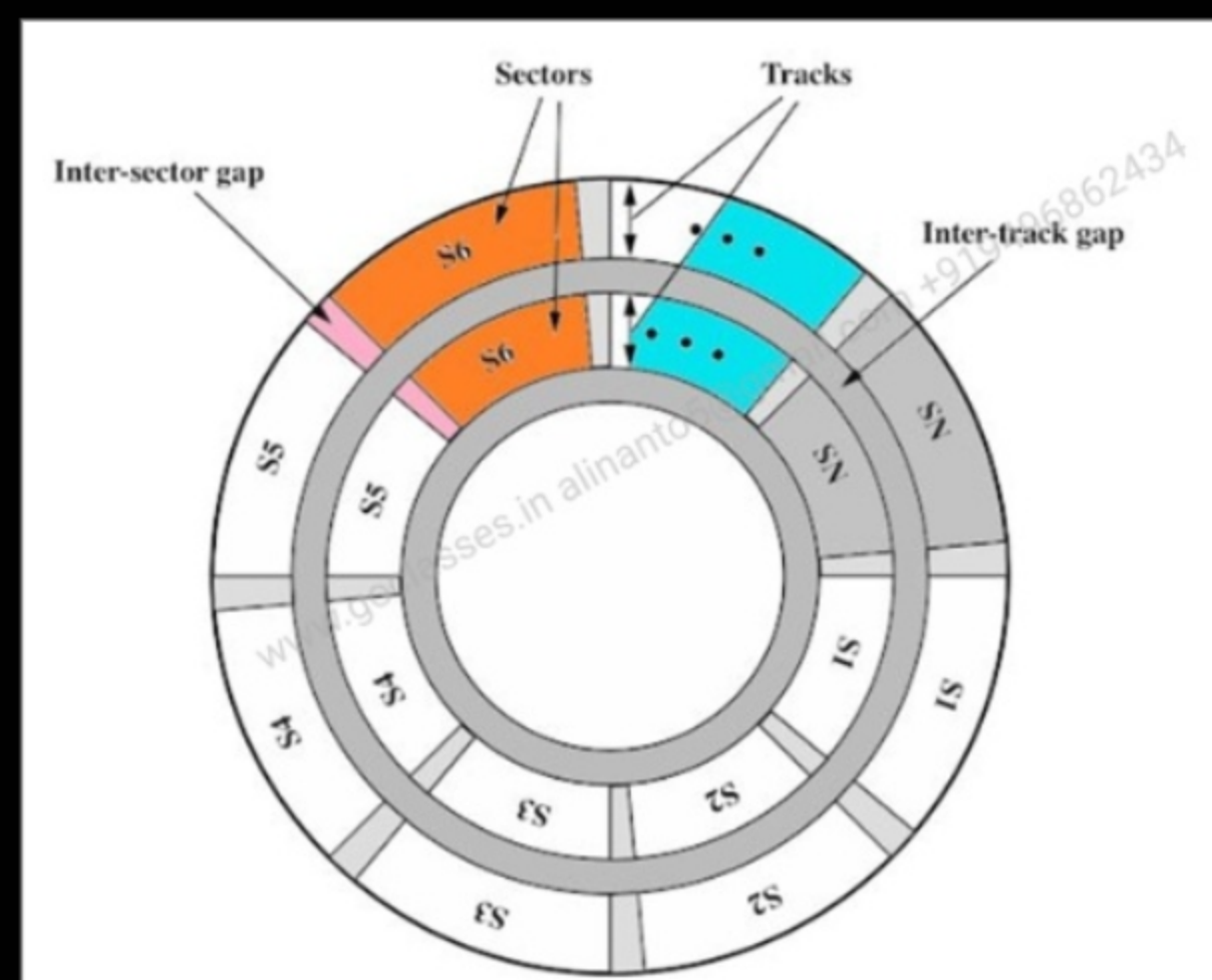


→ Outmost track — track 0

→ Inter sector gap

→ Inter track gap

- **Seek time** → Putting R/W Head on desired track
- **Rotational latency** → Rotate to starting point of desired sector
- **Transfer time** → Rotate complete sector arc
- **Block** → Not a property of disk
  - A property of software → filesystem (OS/DBMS)
  - Consecutive sectors to be considered
- **Sector size** → Property of disk → Cannot change in software



The set of all tracks with the same diameter is called a **cylinder**, because the space occupied by these tracks is shaped like a cylinder; A cylinder contains one track per platter surface.

Each track is divided into arcs called sectors, whose size is a characteristic of the disk and cannot be changed. The size of a **disk block** can be set when the disk is initialized as a multiple of the sector size.

- All R/W Head move together in same cylinder position
- Only one R/W head is active at any time.
- To read one whole track, one rotation is sufficient

#### 10.2.1 Physical Characteristics of Disks

Physically, disks are relatively simple (Figure 10.2). Each disk **platter** has a flat, circular shape. Its two surfaces are covered with a magnetic material, and information is recorded on the surfaces. Platters are made from rigid metal or glass.

When the disk is in use, a drive motor spins it at a constant high speed (usually 60, 90, or 120 revolutions per second, but disks running at 250 revolutions per second are available). There is a read-write head positioned just above the surface of the platter. The disk surface is logically divided into **tracks**, which are subdivided into **sectors**. A **sector** is the smallest unit of information that can be read from or written to the disk. In currently available disks, sector sizes are

- Disk surface is even coat of magnetic material
- However surface is logically divided into tracks & sectors, with sufficient track-to-track & sector-to-sector gap

→ 512 Bytes  
= 1/2 KB

#cylinders = #tracks per surface

#### Example calculation (rough scale)

Let's take a 1 TB HDD (older generation):

| Component          | Approx. Value  | Comment          |
|--------------------|----------------|------------------|
| Sectors per track  | 500 (average)  | varies with zone |
| Tracks per surface | 100,000        | typical          |
| Bytes per sector   | 4096           | modern format    |
| Surfaces           | 8 (4 platters) | common layout    |

Then:

$8 \text{ surfaces} \times 100,000 \text{ tracks/surface} \times 500 \text{ sectors/track} \times 4096 \text{ bytes/sector} \approx 1.6 \times 10^{12} \text{ bytes} \approx 1.6 \text{ TB}$

That's in the right ballpark (after formatting overhead, you get  $\approx 1$  TB usable).



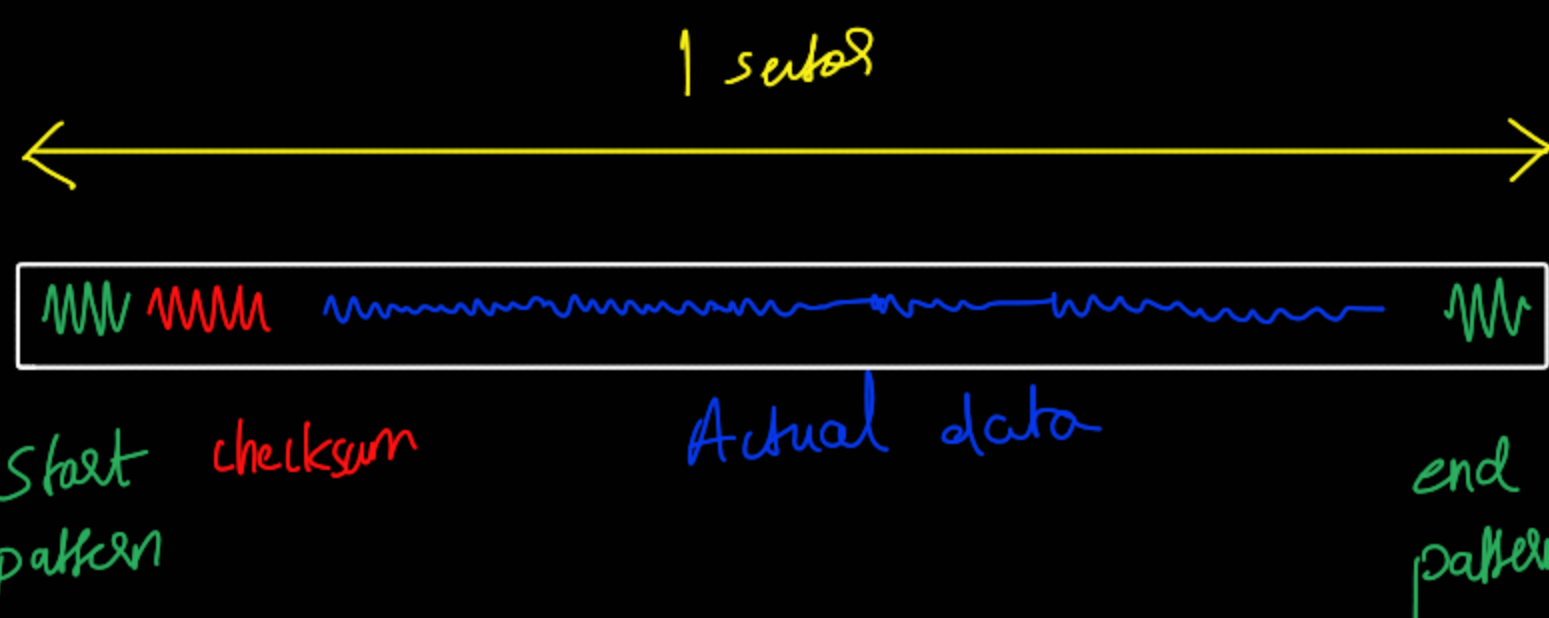
The read-write head stores information on a sector magnetically as reversals of the direction of magnetization of the magnetic material.

Each side of a platter of a disk has a read-write head that moves across the platter to access different tracks. A disk typically contains many platters, and the read-write heads of all the tracks are mounted on a single assembly called a disk arm, and move together.

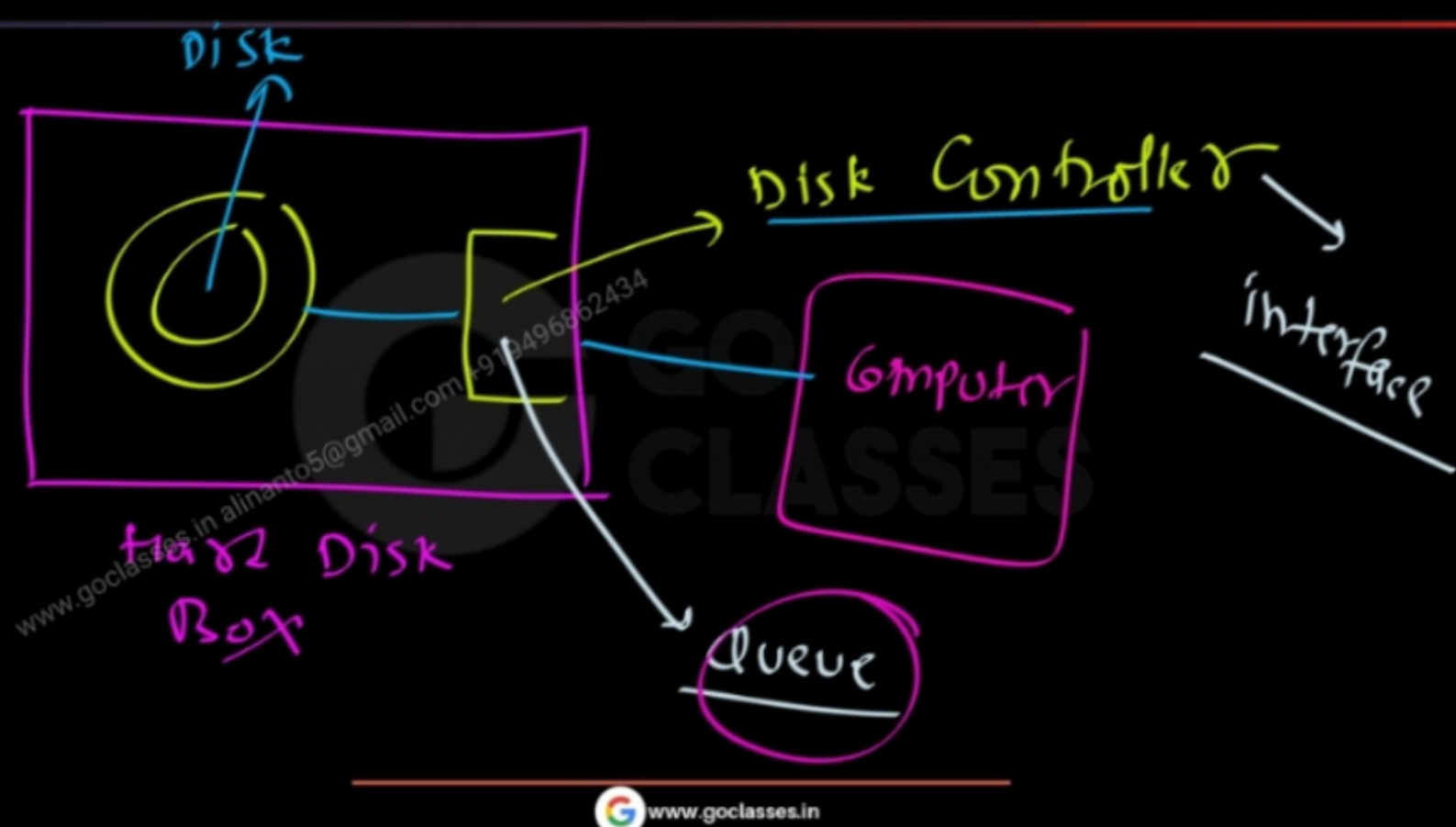
Since the heads on all the platters move together, when the head on one platter is on the  $i_{th}$  track, the heads on all other platters are also on the  $i_{th}$  track of their respective platters. Hence, the  $i_{th}$  tracks of all the platters together are called the  $i_{th}$  cylinder.

www.goclasses.in

## Sector



→ Every extra info in any sector apart from actual data is called disk formatting or formatted data



Some means is needed to locate sector positions within a track. Clearly, there must be some starting point on the track and a way of identifying the start and end of each sector. These requirements are handled by means of control data recorded on the disk. Thus, the disk is formatted with some extra data used only by the disk drive and not accessible to the user.

A **disk controller** interfaces between the computer system and the actual hardware of the disk drive; in modern disk systems, the disk controller is implemented within the disk drive unit. A disk controller accepts high-level commands to read or write a sector, and initiates actions, such as moving the disk arm to the right track and actually reading or writing the data. Disk controllers also attach **checksums** to each sector that is written; the checksum is computed from the data written to the sector. When the sector is read back, the controller computes the checksum again from the retrieved data and compares it with the stored checksum; if the data are corrupted, with a high probability the newly computed checksum will not match the stored checksum. If such an error occurs, the controller will retry the read several times; if the error continues to occur, the controller will signal a read failure.

$$\begin{aligned} \# \text{ Average Rotational Latency} &= \frac{1}{2} \text{ Time for one revolution} \\ &= \text{Time for half revolution} \end{aligned}$$

## Disk Capacity

$$\underbrace{\# \text{ platters} \times 2}_{\# \text{ recording surface}} \times \underbrace{\# \text{ tracks/surface}}_{\# \text{ cylinders}} \times \# \text{ sectors/track} \times \text{Sector size}$$

## Access Time & Transfer time

The sum of the seek time, if any, and the rotational delay equals the access time, which is the time it takes to get into position to read or write.

Once the head is in position, the read or write operation is then performed as the sector moves under the head; this is the data transfer portion of the operation; the time required for the transfer is the transfer time.

### DISK ACCESS TIME QUIZ

- 1000 CYLINDERS, 10 SECTORS/TRACK
- HEAD ASSEMBLY AT CYLINDER 0 INITIALLY
- HEAD MOVES 10ms/CYLINDER
- DISK ROTATES 100 TIMES/SECOND
- CONTROLLER, BUS PERFECT, NO PREVIOUS REQUEST

AVERAGE TIME TO READ A Byte on  
Cylinder 5 \_\_\_\_\_

$$\begin{aligned} 50 \mu s &\rightarrow \text{To move to cylinder 5} \\ 1 \text{ revolution} &\rightarrow 10 \text{ ms} \Rightarrow \text{Average rotational latency} = 5 \text{ ms} \\ \text{Transfer time} &= \frac{10 \text{ ms}}{10} = 1 \text{ ms} \end{aligned} \quad \left. \begin{array}{l} \\ \\ \end{array} \right\} \text{Total } 6.05 \text{ ms}$$



Q In above question  $\rightarrow$  Average time to read a randomly chosen byte is ?

$\rightarrow$  Randomly chosen byte is equally likely to be in any cylinder

$$\rightarrow E(\text{seek time}) = E(\text{X}^{\text{th}} \text{ cylinder} \times 10\mu\text{s}) = 10\mu\text{s} \times E(k) = 10\mu\text{s} \times 500 = \underline{5\text{ms}}$$

$$\rightarrow E(\text{rotational latency}) = 5\text{ms}$$

$$\rightarrow E(\text{transfer time}) = 1\text{ms}$$

$$\text{Total} = \underline{11\text{ms}}$$

Consider a disk with a sector size of 512 bytes, 2,000 tracks per surface, 50 sectors per track, 5 double-sided platters, average seek time of 10 msec.

1. What is the capacity of a track in bytes? What is the capacity of each surface? What is the capacity of the disk?

2. How many cylinders does the disk have?

3. Give examples of valid block sizes. Is 256 bytes a valid block size? 2,048?

4. If the disk platters rotate at 5,400 rpm (revolutions per minute), what is the maximum rotational delay?

5. Assuming that one track of data can be transferred per revolution, what is the transfer rate?

$$5) 25 \text{ KB} \times 90 = 2250 \text{ KBps} \approx 2 \text{ MBps}$$

$$1) 1 \text{ track} = 50 \text{ sectors} = 50 \times 512 \text{ bytes} = 25 \text{ KB}$$

$$2) 2000 \text{ cylinders}$$

$$3) 256 \text{ --- NO}$$

$$2048 \text{ --- Yes --- 4 sectors}$$

$$4) 5400 \text{ rpm} = 90 \text{ rps}$$

$$1 \text{ revolution} = \frac{1}{90} \text{ seconds} \approx \underline{11\text{ms}}$$

The question given below: concern a disk with a sector size of 512 bytes, 2000 tracks per surface, 50 sectors per track, five double-sided platters, and average seek time of 10 milliseconds.

If  $T$  is the capacity of a track in bytes, and  $S$  is the capacity of each surface in bytes, then  $(T, S) = \underline{\hspace{2cm}}$

A. (50K, 50000K)

B. (25K, 25000K)

☒ C. (25K, 50000K)

D. (40K, 36000K)

$$T = 512 \times 50 \text{ bytes} = 25600 \text{ bytes} = 25 \text{ KBytes}$$

$$S = 512 \times 50 \times 2000 = 50000 \text{ KBytes}$$

Q:

Consider a disk, having 6 platters, with an advertised average seek time of 4 ms, rotation speed of 15,000 rpm, and 512-byte sectors with 500 sectors per track. The track-to-track access time is 1 ms. Suppose that we wish to read a file consisting of 2500 sectors for a total of 1.28 Mbytes. We would like to estimate the total time for the transfer.

1. Assume that the file is stored consecutively on a cylinder. (5 surfaces)
2. Assume that the file is stored consecutively on a surface. (5 tracks)
3. Assume that the file is stored randomly in sectors throughout the disk.

www.goclasses.in

$$2) T_{\text{transfer}} = 20\text{ms}$$

additional seek time 4ms for next 4 tracks  
Initial seek time for 1st track = 4ms

$$R.L = 5 \times 2 = 10\text{ms}$$

$$\text{Total} = 20 + 8 + 10 = \underline{38\text{ms}}$$

$$3) 2500 \text{ random sectors} \Rightarrow \frac{2500 \times 4\text{ms}}{500} \text{ seek time}$$

$$2500 \text{ random sectors} \Rightarrow \frac{2500 \times 2\text{ms}}{500} \text{ rotation}$$

$$5 \text{ tracks} \Rightarrow 20\text{ms} \text{ transfer}$$

$$\text{Total} = \underline{15.02 \text{ seconds}}$$

$$1 \text{ track} = 512 \times 500 \text{ bytes} = 256000 \text{ bytes} = 256 \text{ KB}$$

File needs 5 tracks

1) Tracks are in same cylinder

$\rightarrow$  No additional track-track seek time  
 $\rightarrow$  Initial seek time = 4ms

$$\rightarrow 15,000 \text{ rpm} = 250 \text{ rps}$$

$$\rightarrow 1 \text{ revolution completes in } \frac{1}{250} \text{ s} = \underline{4\text{ms}}$$

$$\rightarrow \text{Rotational latency} = 2\text{ms per track}$$

$$\rightarrow \text{transfer} = 5 \times 4\text{ms} = 20\text{ms}$$

$$\rightarrow \text{Total} = \underbrace{20\text{ms}}_{\text{transfer}} + \underbrace{2\text{ms} \times 5}_{\text{Rotation Latency}} + \underbrace{4\text{ms}}_{\text{seek}} = 34\text{ms}$$



For any two sectors, the closest they are in decreasing order of closeness

- ① Two sectors in same track consecutively
- ② Two sectors in same cylinder  $\rightarrow$  simple rotation latency
- ③ Two sectors in adjacent cylinder  $\rightarrow$  simple rotation latency + very less track to track seek time

In decreasing order of closeness:  
 they could be on the same block,  
 they could be on the same track,  
 the same cylinder,  
 an adjacent cylinder.

The data on a disk is ordered in the following way:  
 It is first stored on the first sector of the first surface of the first cylinder. Then in the next sector, and next, until all the sectors on the first track are exhausted. Then it moves on to the first sector of the second surface (remains at the same cylinder), then next sector and so on. It exhausts all available surfaces for the first cylinder in this way. After that, it moves on to repeat the process for the next cylinder.

## Sector Addressing LBA vs $\langle C, h, s \rangle$

$\langle chs \rangle \rightarrow$  cylinder-head-sectors  
 LBA  $\rightarrow$  Logical block addressing  $\rightarrow$  Integer Numbers

$\langle C, h, s \rangle$   
 $\downarrow$  Cylinder no.  
 $\uparrow$  head no. = Surface No.  
 $\uparrow$  Sector no.  
 For all three counting starts at 0.

$\rightarrow$  By default track numbers starts at 0. (outermost track)  
 $\rightarrow$  Highest number track (next to spindle)

$\rightarrow$  In a disk, sectors are the smallest unit that can be read/write  
 $\rightarrow$  This is why sectors are given addresses.

Understand with example

3 platters  $\equiv$  6 surface  
 6 heads

3 cylinders

4 sectors per track

| LBA | $\langle C, h, s \rangle$ |
|-----|---------------------------|
| 0   | $\langle 0, 0, 0 \rangle$ |
| 1   | $\langle 0, 0, 1 \rangle$ |
| 2   | $\langle 0, 0, 2 \rangle$ |
| 3   | $\langle 0, 0, 3 \rangle$ |
| 4   | $\langle 0, 1, 0 \rangle$ |
| 5   | $\langle 0, 1, 1 \rangle$ |
| 6   | $\langle 0, 1, 2 \rangle$ |
| 7   | $\langle 0, 1, 3 \rangle$ |
| 8   | $\langle 0, 2, 0 \rangle$ |

stay in same cylinder  
 switch to different head

Q) What will be  $\langle 1, 0, 0 \rangle$ ?  
 $= 6 \times 1 \times 4 = 24$

Let's derive general formula

$\langle C, h, s \rangle$  to LBA

$$LBA = s + \left( \frac{\# \text{sectors}}{\text{track}} \right) \left[ h + (\# \text{surface}) \cdot C \right]$$

$$LBA = s + \#SPT \left[ h + C \cdot \#heads \right]$$

$$LBA = s + \#SPT \cdot h + \#SPC \cdot C$$

$\#SPT \rightarrow$  Sectors per track  
 $\#SPC \rightarrow$  Sectors per cylinder  
 $\#SPC = \#SPT \times \#heads$   
 $= \#SPT \times \#surfaces$



But how to convert LBA to  $\langle c, h, s \rangle$

$$c = \frac{LBA}{\#SPC} \quad h = \frac{LBA \% \#SPC}{\#SPT} \quad s = (LBA \% \#SPC) \% \#SPT$$

eg.  $\langle c, h, s \rangle = \langle 2, 3, 2 \rangle$

$$LBA = 2 + (4 \times 3) + (2 \times 2)$$

$$= 2 + 12 + 4$$

$$LBA = 18$$

eg.  $LBA = 35$        $\langle c, h, s \rangle = ?$

$$c = \frac{35}{24} = 1 \text{ remainder } 11$$

$$h = \frac{11}{4} = 2 \text{ remainder } 3$$

$$s = 3$$

$c, h, s = \langle 1, 2, 3 \rangle$

$$LBA = s + \#SPT \cdot h + \#SPC \cdot c$$

if counting starts from 1 for all

$$\left. \begin{array}{l} LBA' = LBA - 1 \\ s' = s - 1 \\ h' = h - 1 \\ c' = c - 1 \end{array} \right\} \begin{array}{l} LBA' = s' + \#SPT \cdot h' + \#SPC \cdot c' \\ \text{then } LBA = LBA' + 1 \end{array}$$

A hard disk has with the given specification as 52 sectors per track, 8 platters each with 2 recording surfaces and 500 cylinders. The address of a sector is given as a triple  $(c, p, s)$ , where  $c$  is the cylinder number,  $p$  is the surface number and  $s$  is the sector number. Thus, the 0th sector is addressed as  $(0, 0, 0)$ , the 1st sector as  $(0, 0, 1)$  and so on. The address  $\langle 300, 6, 27 \rangle$  corresponds to sector number

$$\#SPT = 52$$

$$\#SPC = \#SPT \times \#heads$$

$$\#SPC = 52 \times 16 = 832$$

$$LBA = (300 \times 832) + (6 \times 52) + 27$$

$$LBA = 249939^{\text{th}} \text{ sector}$$

$$\begin{array}{r} 832 \times \\ \underline{300} \\ 249600 + \\ \quad 312 + \\ \quad \quad 27 \\ \hline 249939 \end{array}$$



File Organisation & Indexing

- Block size → determined by software, all other parameters are fixed by hardware
- File Organisation → how data is stored (actual data)
- Indexing → Index datatype for speeding up data retrieval (facility of DBMS)

■ **Block** – a contiguous sequence of sectors from a single track

● data is transferred between disk and main memory in blocks

● sizes range from 512 bytes to several kilobytes

- ▶ Smaller blocks: more transfers from disk
- ▶ Larger blocks: more space wasted due to partially filled blocks
- ▶ Typical block sizes today range from 4 to 16 kilobytes

→ Block is the smallest unit of data access w.r.t DBMS software.  
→ Block is also called Page ☆

If that is the case, there is no speed advantage to using separate DBMS blocks.

🗨️ That's a very natural conclusion — and you're thinking in the right direction.

At first glance, it looks like if the DBMS ultimately relies on the OS to read/write the same disk sectors, then defining its own block size adds no real speed advantage.

But actually — the DBMS's own block system is not about raw disk alignment speed alone. It exists for architectural control, caching efficiency, and predictable I/O behavior that the OS file system can't provide.

Let's go step by step to see why DBMS blocks still matter a lot.

◆ 1. File system I/O ≠ DBMS I/O needs

The OS and file system are general-purpose: They're optimized for arbitrary files — documents, binaries, logs, etc.

A DBMS, on the other hand:

- accesses data in highly structured ways (tables, indexes, records), and
- needs to control when and how much I/O happens, for correctness and performance.

So even if the OS already uses blocks, the DBMS needs its own “logical I/O unit” tuned for database semantics, not just for storage efficiency.

◆ 2. DBMS blocks give control over caching

The OS already has a page cache — but that's too generic. If the DBMS relied on it directly:

- it wouldn't know what's cached,
- it couldn't evict pages intelligently, and
- it couldn't coordinate cache with transaction consistency.

That's why every major DBMS (Oracle, PostgreSQL, MySQL/InnoDB, SQL Server) manages its own buffer pool (in RAM), organized by DBMS blocks.

Example:

- Suppose the OS block size = 4 KB, DBMS block size = 8 KB.
- DBMS buffer pool caches data in 8 KB chunks — one per DBMS page.
- The DBMS knows which table/index each page belongs to, when it was last used, and whether it's dirty (modified but not yet written to disk).

➡ The DBMS buffer manager can then decide what to write, when to flush, and in what order — critical for transaction performance and recovery.

◆ 3. DBMS blocks enable logical organization of data

Inside the database file, each DBMS block stores:

- a page header (metadata about that page),
- records (tuples) or index entries,
- and maybe free space for new inserts.

So DBMS blocks are not just I/O units — they're also logical containers for data organization.

This allows:

- indexing structures (like B+ trees) to reference page IDs, not byte offsets.
- recovery logs to record changes at the page level.
- buffer management, checkpointing, and WAL (Write-Ahead Logging) to work efficiently.

◆ 4. Performance control beyond OS behavior

Even if alignment is perfect, the OS file system caching strategy is not optimal for databases.

| OS cache behavior                             | DBMS needs                                               |
|-----------------------------------------------|----------------------------------------------------------|
| Prefetches sequentially                       | Needs random access patterns                             |
| Writes asynchronously with its own heuristics | Needs strict write ordering for durability (WAL)         |
| Flushes based on system-wide LRU              | Needs fine-grained eviction (e.g., keep hot index pages) |

That's why DBMSs often use:

- O\_DIRECT (Linux) or direct I/O to bypass OS cache,
- and implement their own I/O scheduler and write-ahead logging.

So DBMS blocks let the database bypass OS-level uncertainty.

◆ 5. Speed advantage — not raw I/O, but smarter I/O

Even though both layers move the same bytes, DBMS blocks enable speed through:

| Mechanism                 | Benefit                                             |
|---------------------------|-----------------------------------------------------|
| Buffer pool caching       | Avoids repeated disk reads for hot data             |
| Prefetch and write-behind | Optimized batch I/O for pages                       |
| Checkpoints and WAL       | Controlled, predictable disk writes                 |
| Index-based access        | Accesses only relevant DBMS blocks, not whole files |

So yes — the raw disk I/O cost per byte is similar — but the total I/O performed is much less, thanks to smart DBMS block management.

◆ 6. Summary

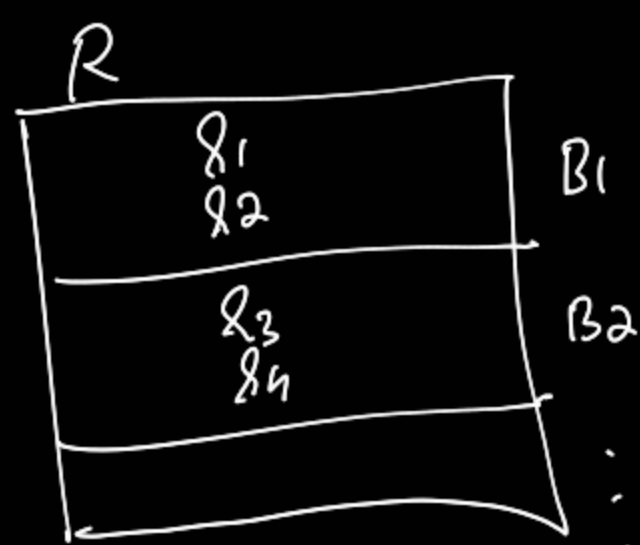
| Question                                       | Answer                                                                                          |
|------------------------------------------------|-------------------------------------------------------------------------------------------------|
| Do DBMS blocks align perfectly with OS blocks? | Not always, but usually multiples for efficiency.                                               |
| If not, does that hurt performance?            | Slightly, but DBMS design mitigates it.                                                         |
| So why define DBMS blocks at all?              | To control caching, I/O scheduling, recovery, and data organization at the database level.      |
| Is there speed advantage?                      | Yes — through reduced total I/O and better predictability, not through faster individual reads. |

✓ In short:

DBMS blocks aren't about reading faster per se — they're about reading smarter.



One relation  $\xrightarrow{\text{stored as}}$  One file  $\rightarrow$  Sequence of Blocks



$\rightarrow$  Block size  $\geq$  Record size  
 $\rightarrow$  Block can contain multiple tuples

Block size — Multiple of Sector size  
 — Cannot exceed size of a track

Record  $\equiv$  tuple  $\rightarrow$  ① Fixed-Length Record  $\rightarrow$  All fields of fixed size — eg. char(30)  
 File  $\equiv$  relation  $\rightarrow$  ② Variable-Length Record  $\rightarrow$  Any of the field is variable size  
 $\rightarrow$  eg. varchar(30)  $\rightarrow$  Atmost 30 bytes may be smaller

## Records

|                                                                                                        |           |  |
|--------------------------------------------------------------------------------------------------------|-----------|--|
| Create Table star (<br>ID Int,<br>Name char(30),<br>Address char(255),<br>Gender char(1),<br>DOB Date) |           |  |
| ID Int,                                                                                                | 4 bytes   |  |
| Name char(30),                                                                                         | 30 bytes  |  |
| Address char(255),                                                                                     | 255 bytes |  |
| Gender char(1),                                                                                        | 1 byte    |  |
| DOB Date)                                                                                              | 10 bytes  |  |

$\rightarrow$  Fixed Length Record

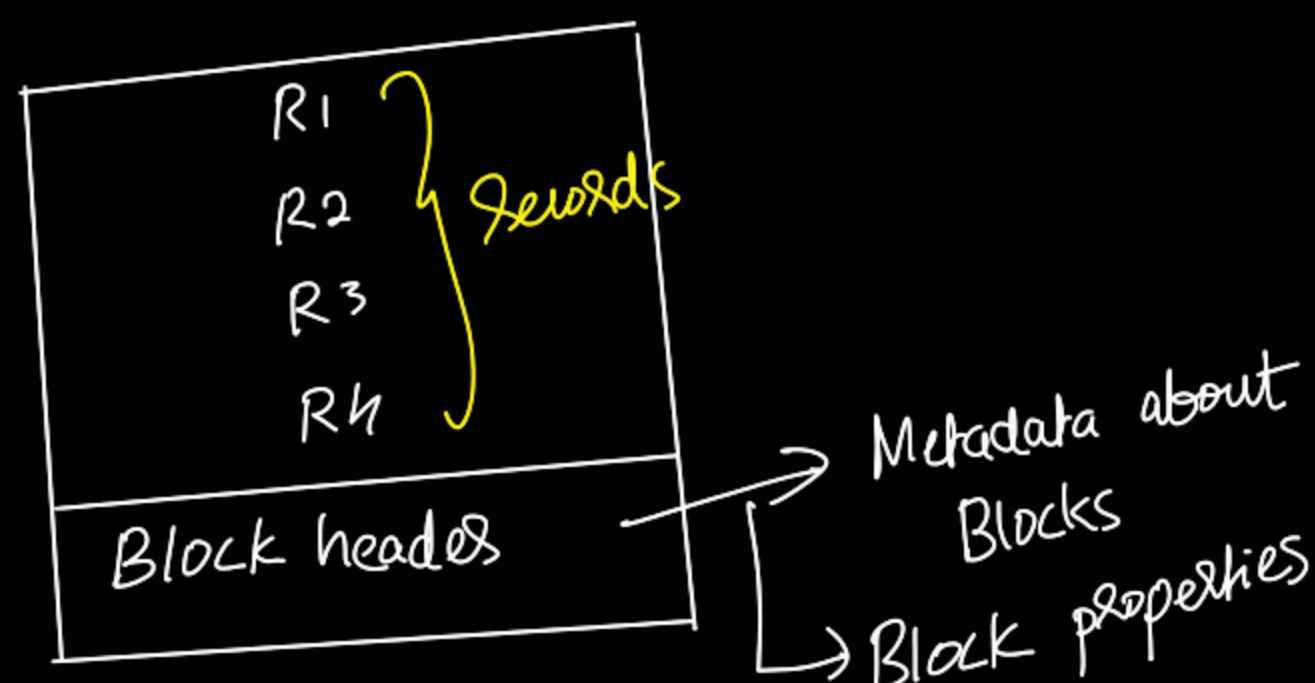
$\rightarrow$  Factors affecting where the record / a field in record is stored.

- How many fields in record
  - What sizes are the fields
  - Recent access time  $\rightarrow$  whether in cache, main memory or disk
- $\rightarrow$  This is a property of record & has to be stored inside the record.

| Record   |               |      |
|----------|---------------|------|
| Metadata | Actual Fields | Data |

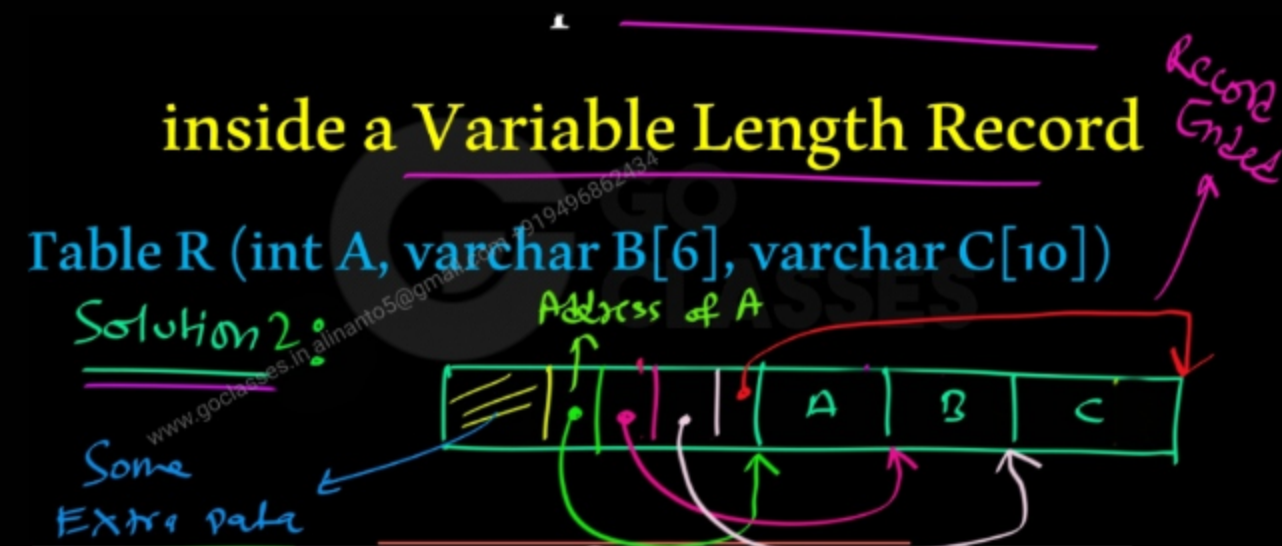
- $\rightarrow$  Extra info
- $\rightarrow$  Last accessed time
- $\rightarrow$  Record header
- $\rightarrow$  other record properties
- $\rightarrow$  Timestamp - creation

Size of every field  $\rightarrow$  Fixed Length Record  $\rightarrow$  stored in Block Header  
 $\rightarrow$  Variable Length Record  $\rightarrow$  Block Header



How to find a particular field inside a variable length record

Solution 1: Use Terminating special character for variable field  
 $\rightarrow$  Not used since cannot access next field directly without first scanning for special character.





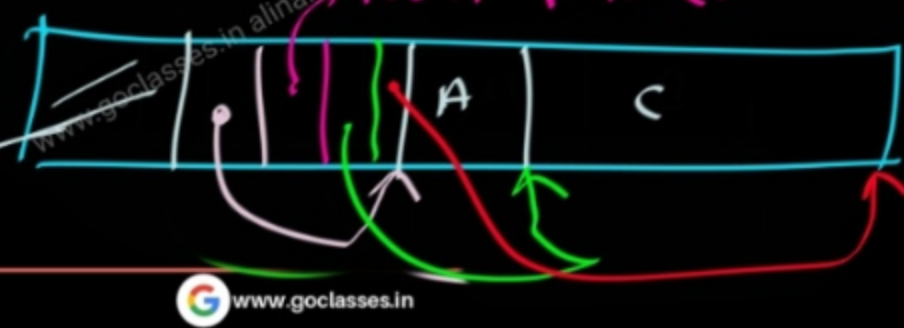
For John:  $B = \text{NULL}$

## inside a Variable Length Record

Table R (int A, varchar B[6], varchar C[10])

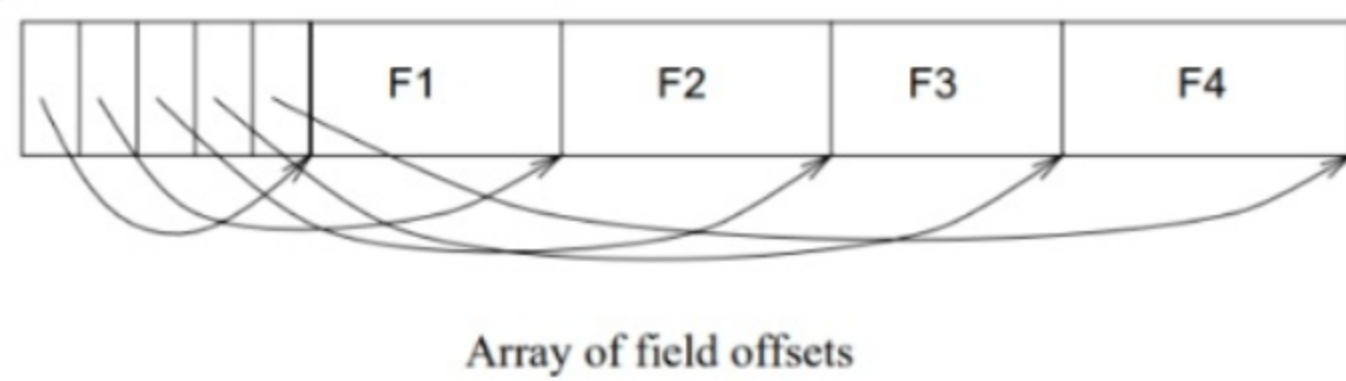
Solution 2:

Some Extra data



→ Solution 2: Store pointers  
→ In case of NULL field, store NULL pointer ☆

An alternative is to reserve some space at the beginning of a record for use as an array of integer offsets—the  $i$ th integer in this array is the starting address of the  $i$ th field value relative to the start of the record. Note that we also store an offset to the end of the record; this offset is needed to recognize where the last field ends. Both alternatives



Array of field offsets

## Spanned Vs Unspanned Record Organization

- Record can exist in two blocks.
- $\frac{B-K}{R} = \text{bfr}$  (Blocking factor)
- No unused space
- Multiple block may be lead for one record
- Variable length record prefer spanned however unspanned can also be used
- For variable length record  $\text{bfr} = \text{average bfr}$

- Records are not allowed to cross block boundaries
- $\# \text{Records} / \text{Block} = \left\lfloor \frac{B-K}{R} \right\rfloor$
- $\# \text{Records} / \text{Block} = \text{Blocking Factor (bfr)}$
- Some Bytes might be wasted in a block
- Unused space =  $(B-K) \% R = B - (\text{bfr} \cdot R)$
- Assume header  $k=0$  unless given
- Usually preferred for fixed length record.

Note: → Unless specified → Unspanned is the default case  
→ Fixed length Record is the default case

$$\# \text{ of Block in file} = \left\lceil \frac{R}{\text{bfr}} \right\rceil \quad \begin{array}{l} R - \# \text{ records in file} \\ \text{bfr} - \text{Blocking factor} \end{array}$$

Side note

$\lceil x \rceil \rightarrow \text{ceiling function}$

$\lfloor x \rfloor \rightarrow \text{Floor function}$

Q:

For variable-length records, different records have different sizes. Let the average record size in a file be  $R$ , and number of records in the file be  $r$ .

Assume that Block size is  $B$  bytes, and  $K$  bytes are used for Block Header. Using spanned organization, find the number of blocks needed to store this file?

$$\begin{aligned} \rightarrow \text{Total file} &= R \cdot r \\ \text{Usable block size} &= B - K \end{aligned}$$

$$\rightarrow \# \text{ blocks} = \left\lceil \frac{R \cdot r}{B - K} \right\rceil$$

$$\rightarrow \text{bfr} = \frac{B - K}{R} \text{ (avg)}$$

## Storing Records on Disk

→ Even though conceptually order of tuples (records) doesn't matter when storing physically on disk, order matters.

### ① Unordered / Heap Files

- Stored in the order in which new records are entered/inserted.
- Advantage: Insertion efficient
- Disadvantage: Search Inefficient  
→ Linear search
- Worst case search —  $b$  blocks
- Average case search —  $b/2$  blocks

Inserting a new record is very efficient. The last disk block of the file is copied into a main memory buffer, the new record is added, and the block is then rewritten back to disk. The address of the last file block is kept in the file header.



## ② Ordered / Sequential File

- Stored in sorted order based on some field
- Binary search on blocks instead of records
- Useful when search is based on ordered field
- No advantage for searching based on unordered field

**Table 16.3** Average Access Times for a File of  $b$  Blocks under Basic File Organizations

| Type of Organization | Access/Search Method            | Average Blocks to Access a Specific Record |
|----------------------|---------------------------------|--------------------------------------------|
| Heap (unordered)     | Sequential scan (linear search) | $b/2$                                      |
| Ordered              | Sequential scan                 | $b/2$                                      |
| Ordered              | Binary search                   | $\log_2 b$                                 |

## Addressing

Block address  $\rightarrow \langle c, h, s \rangle$   
 $\rightarrow$  LBA

Record address  $\rightarrow$  block address + offset of record starting point

eg. Block address 0 } 1st record  
 offset 0 }

**Exercise 13.5.1:** Suppose a record has the following fields in this order: A character string of length 15, an integer of 2 bytes, a SQL date, and a SQL time (no decimal point). How many bytes does the record take if: (SQL dates require 10 bytes, and SQL times require 8 bytes)

- Fields can start at any byte.
- Fields must start at a byte that is a multiple of 4.
- Fields must start at a byte that is a multiple of 8.

a) bytes 0-14 string  
 16-17 integer  
 24-33 SQL date  
 40-47 SQL time  
 Total 48 bytes

a) Record size =  $15 + 2 + 10 + 8 = 35$  bytes

b) bytes 0-14 string  
 1 byte waste  
 bytes 16-17 integer  
 2 bytes waste  
 bytes 20-29 SQL date  
 2 bytes waste  
 bytes 32-39 time  
 Total 40 bytes

**Exercise 13.5.2:** Repeat Exercise 13.5.1 for the list of fields: a real of 8 bytes, a character string of length 17, a single byte, and a SQL date.

- $8 + 17 + 1 + 10 = 36$  bytes
- $8 + 20 + 4 + 12 = 44$  bytes
- $8 + 24 + 8 + 16 = 56$  bytes

☆ Round up each field to next multiple

Consider a disk with the following characteristics (these are not parameters of any particular disk unit): block size  $B=512$  bytes, interblock gap size  $G=128$  bytes, number of blocks per track=20, number of tracks per surface=400. A disk pack consists of 15 double-sided disks.

- What is the total capacity of a track and what is its useful capacity (excluding interblock gaps)?
- How many cylinders are there?
- What is the total capacity and the useful capacity of a cylinder?
- What is the total capacity and the useful capacity of a disk pack?
- Suppose the disk drive rotates the disk pack at a speed of 2400 rpm (revolutions per minute); what are the transfer rate in bytes/msec and the block transfer time  $btt$  in msec? What is the average rotational delay  $rd$  in msec?

e)  $2400 \text{ rpm} = 40 \text{ rps}$   
 1 revolution =  $\frac{1000 \text{ ms}}{40} = 25 \text{ ms}$   
 1 track = 25ms  
 ie  $\frac{10 \text{ KB}}{25 \text{ ms}} = 0.4 \text{ B/ms (usable)}$   
 actual =  $\frac{12.5 \text{ KB}}{25 \text{ ms}} = \frac{20 \times [512 + 128]}{25} = \frac{20 \times 640}{25} = 512 \text{ B}$   
 $btt = 1 \text{ ms}$

a) Total capacity =  $(512 + 128) \times 20 = 12.5 \text{ KB}$   
 Usable =  $512 \times 20 = 10 \text{ KB}$

b) 400

c) Total =  $12.5 \times 30 = 375 \text{ KB}$   
 Usable =  $10 \times 30 = 300 \text{ KB}$

d) Total =  $12.5 \times 400 \times 2 = 10000 \text{ KB}$   
 Usable =  $10 \times 400 \times 2 = 8000 \text{ KB}$

(f) Suppose the average seek time (time to go to the correct track mechanically) is 30 msec. How much time does it take (on the average) in msec to locate and transfer a single block given its block address?

(g) Calculate the average time it would take to transfer 20 random blocks and compare it with the time it would take to transfer 20 consecutive blocks using double buffering to save seek time and rotational delay.

f) Total =  $30 \text{ ms} + 12.5 \text{ ms} + 1 \text{ ms} = 43.5 \text{ ms}$

g) 20 Random =  $20 \times 43.5 \text{ ms}$   
 Double buffering =  $30 + 12.5 + (20 \times 1 \text{ ms})$   
 Once



## Problem 2

A file has  $r=20,000$  STUDENT records of fixed-length. Each record has the following fields: NAME (30 bytes), SSN (9 bytes), ADDRESS (40 bytes), PHONE (9 bytes), BIRTHDATE (8 bytes), SEX (1 byte), MAJORDEPTCODE (4 bytes), MINORDEPTCODE (4 bytes), CLASSCODE (4 bytes, integer), and DEGREEPROGRAM (3 bytes). An additional byte is used as a deletion marker. The file is stored on the disk whose parameters are given in Problem 1.  $\rightarrow$  Block size = 512 B

(a) Calculate the record size  $R$  in bytes.

(b) Calculate the blocking factor  $bfr$  and the number of file blocks  $b$  assuming an unspanned organization.

$$R = 30 + 9 + 40 + 9 + 8 + 1 + 4 + 4 + 4 + 3 + 1 = 113$$

$$R = 20,000$$

$$b) \text{ bfr} = \left\lfloor \frac{B}{R} \right\rfloor = \frac{512}{113} = 4$$

$$\# \text{ file block} = \frac{20,000}{4} = 5000$$

16.36. Suppose that only 80% of the STUDENT records from Exercise 16.28 have a value for Phone, 85% for Major\_dept\_code, 15% for Minor\_dept\_code, and 90% for Degree\_program; and suppose that we use a variable-length record file. Each record has a 1-byte field type for each field in the record, plus the 1-byte deletion marker and a 1-byte end-of-record marker. Suppose that we use a spanned record organization, where each block has a 5-byte pointer to the next block (this space is not used for record storage).

a. Calculate the average record length  $R$  in bytes.

Block size = 512

b. Calculate the number of blocks needed for the file.

A file has  $r = 20,000$  STUDENT records of variable. Each record has the following fields: Name (30 bytes), Ssn (9 bytes), Address (40 bytes), PHONE (10 bytes), Birth\_date (8 bytes), Sex (1 byte), Major\_dept\_code (4 bytes), Minor\_dept\_code (4 bytes), Class\_code (4 bytes, integer), and Degree\_program (3 bytes).

$$b) \# \text{ blocks for file} = \frac{R \cdot r}{B - K} = \frac{20000 \cdot R}{512 - 5}$$

## Free Space Management

- $\rightarrow$  A filesystem does not store file contiguously
- $\rightarrow$  It stores files wherever free space is available
- $\rightarrow$  Files & free space will be fragmented

### Small Block size

- Less wastage
- More fragmented
- Slower read

### Large Block Size

- More space wastage for small files
- Less block per file
- Lesser seeks & delays
- Faster read

## Keeping track of free space

① Bitmap (bit vector)

② Free List Method (Free List) aka Linked List Method

- $\rightarrow$  1 bit per block
- $\rightarrow$  size of bitmap = #blocks (in bits)
- $\rightarrow$  if the block is free, the bit is 1.
- $\rightarrow$  Block is used  $\Rightarrow$  bit is 0.

Bitmap size - fixed & independent of free space

For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bit map would be

001111001111110001100000011100000 ...

$\rightarrow$  highly efficient to find first free block or first  $n$  contiguous free blocks.

Q:

Consider a 1.3 GB disk with 512-byte block. The system uses a bit-vector to allocate disk space.

What be the size of bit map to keep track of free blocks??

$$= \frac{1.3 \times 2^{30}}{2^9 \times 8} \text{ bytes in bitvector}$$

$$= 1.3 \times 2^{18} \text{ bytes}$$

$$= 332.8 \text{ KB}$$



stores block address of next  $n$  free blocks

- This is the last ( $n^{\text{th}}$ ) address in the first free block
- It stores the address of next  $n$  free blocks



Q: With a 1-KB block and a 32-bit disk block number, each block on the free list can hold how many numbers of free blocks?

$$32 \text{ bit} = 4 \text{ byte}$$

$$1 \text{ block} = 1 \text{ KB} = 1024 \text{ bytes}$$

$$\Rightarrow \text{It can store } \frac{1024}{4} = 256 \text{ addresses}$$

☆ → But the last address is not free ⇒ It can hold 255 free Block addresses



→ When the disk is nearly full → The Linked List method will require fewer blocks than bitmap

Generally, Free Blocks are used to hold the free list, so the storage is essentially free.

Basically, **Linked Free List method doesn't eat up our useful space...**

**It takes space, BUT from the Already Free space it takes.**

## Database Management System

To create a file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list.

**When a file is deleted, its disk space is added to the free-space list.**

**The free-space list may be implemented as bitmap or linked list.**

### Practice Question

Free disk space can be kept track of using a free list or a bitmap.

**Disk addresses require D bits.** For a disk with B blocks, F of which are free, state the condition under which the free list uses less space than the bitmap.

B total block

F are free

Disk address = D bits

Total space required by free List = D × F bits

Total space required by bitmap = B bits

$B > DF \Rightarrow$  free List less space

ex. if  $D=16$   $B > 16 \cdot F$

or  $F < \frac{1}{16} \cdot B \Rightarrow F < 6.66\% \text{ of } B$

$\Rightarrow D=16 \Rightarrow$  if free space is less than 6.66% of total space  $\Rightarrow$  free List method saves space.

A bit-map can be used to keep track of which blocks are free in a file-system's partition on disk. Assuming, 1 KB block size and a disk size of 40 GB, what is the size of the bit map?

$$40 \text{ GB} = 40 \times 2^{30} \text{ bytes} = 40 \times 2^{20} \text{ KB}$$

$$1 \text{ block} = 1 \text{ KB}$$

$$\Rightarrow \text{Bitmap size} = \frac{40 \times 2^{20} \text{ KB}}{1 \text{ KB}} \text{ bits} = 40 \times 2^{20} \text{ bits}$$

$$\text{Bitmap size} = 5 \times 2^{20} \text{ bytes} = \underline{5 \text{ MB}}$$

For a Hard disk of Capacity 20 MB. Disk block address of 16 bits and disk block size of 1 KB. The number of disk blocks required to store all the Free Blocks using Bitmap method is

- 1
- 2
- 3 ✓
- None

$$20 \text{ MB} = 20 \times 2^{10} \text{ KB}$$

$$1 \text{ block} = 1 \text{ KB}$$

3 block required

$$\text{Bitmap} = \frac{20 \times 2^{10}}{8 \times 1} \text{ bytes} = \underline{25 \text{ KB}}$$

How much space will be required to store the bit map of a 1.3 GB disk with 512 bytes block size?

- A. 332.8 KB ✓
- B. 83.6 KB
- C. 266.2 KB
- D. 256.6 KB

$$\text{Bitmap size} = \frac{1.3 \times 2^{30} \text{ bytes}}{8 \times 512 \text{ bytes}} \text{ bytes}$$

$$\text{Bitmap size} = \frac{1.3 \times 2^{20}}{2^{12}} \text{ KB} = 1.3 \times 2^8 \text{ KB} = 256 \times 1.3 \text{ KB} = \underline{332.8 \text{ KB}}$$

$$\begin{array}{r} 1.3 \\ \times 256 \\ \hline 768 \\ 3328 \\ 13280 \\ \hline 332.8 \end{array}$$



The beginning of a free-space bitmap looks like this after the disk partition is first formatted: 1000 0000 0000 0000 (the first block is used by the root directory). The system always searches for free blocks starting at the lowest-numbered block, so after writing file A, which uses six blocks, the bitmap looks like this: 1111 1110 0000 0000. Show the bitmap after each of the following additional actions:

- File B is written, using five blocks.
- File A is deleted.
- File C is written, using eight blocks.
- File B is deleted.

Using a larger block size in a fixed block size file system leads to

- ✓ better disk throughput but poorer disk space utilization
- better disk throughput and better disk space utilization
- poorer disk throughput but better disk space utilization
- poorer disk throughput and poorer disk space utilization

# INDEXING

## Index

- ① Ordered Indexes / Static Indexes
- ② Tree Based Indexes / Dynamic Indexes
- ③ Hash Index

Simple Level Indexes  
Multilevel Indexes

B Tree  
B<sup>+</sup> Tree

## Ordered Indexes in DBMS

- A single file
- Sorted w.r.t some field
- pairs of (Index field value, Block pointer)
- <Search key, pointer>
- Index can be created w.r.t to each field in a relation simultaneously.
- Max no. of index from a relation = #fields in relation = degree = arity

## Single Level Ordered Indices

- index entries are sorted by search key value
- Default case → Index file is ordered. → sequential file
- Actual data may or may not be ordered. → heap file
- Ordered Index can be created on data even in heap file

- ① Primary Index
- ② Clustering Index
- ③ Secondary Index

- Data file is stored in sorted order of key then the index on this key is a primary index
- Data stored as heap file ⇒ No primary index possible
- Data file is ordered by some field A. This is called ordering field
- A index of field A is primary index if A is ordering key field. OKF ☆

INDEX FILE VS INDEXED FILE  
data file is aka indexed file  
index is stored in index file

☆ → Almost one primary index on one file.

Q) if Primary Index possible, which field to use as key?  
Ans) Id

| Id    | Name       | Dep        | Salary |
|-------|------------|------------|--------|
| 10101 | Srinivasan | Comp. Sci. | 65000  |
| 12121 | Wu         | Finance    | 90000  |
| 15151 | Mozart     | Music      | 40000  |
| 22222 | Einstein   | Physics    | 95000  |
| 32343 | El Said    | History    | 60000  |
| 33456 | Gold       | Physics    | 87000  |
| 45565 | Katz       | Comp. Sci. | 75000  |
| 58583 | Califieri  | History    | 62000  |
| 76543 | Singh      | Finance    | 80000  |
| 76766 | Crick      | Biology    | 72000  |
| 83821 | Brandt     | Comp. Sci. | 92000  |
| 98345 | Kim        | Elec. Eng. | 80000  |

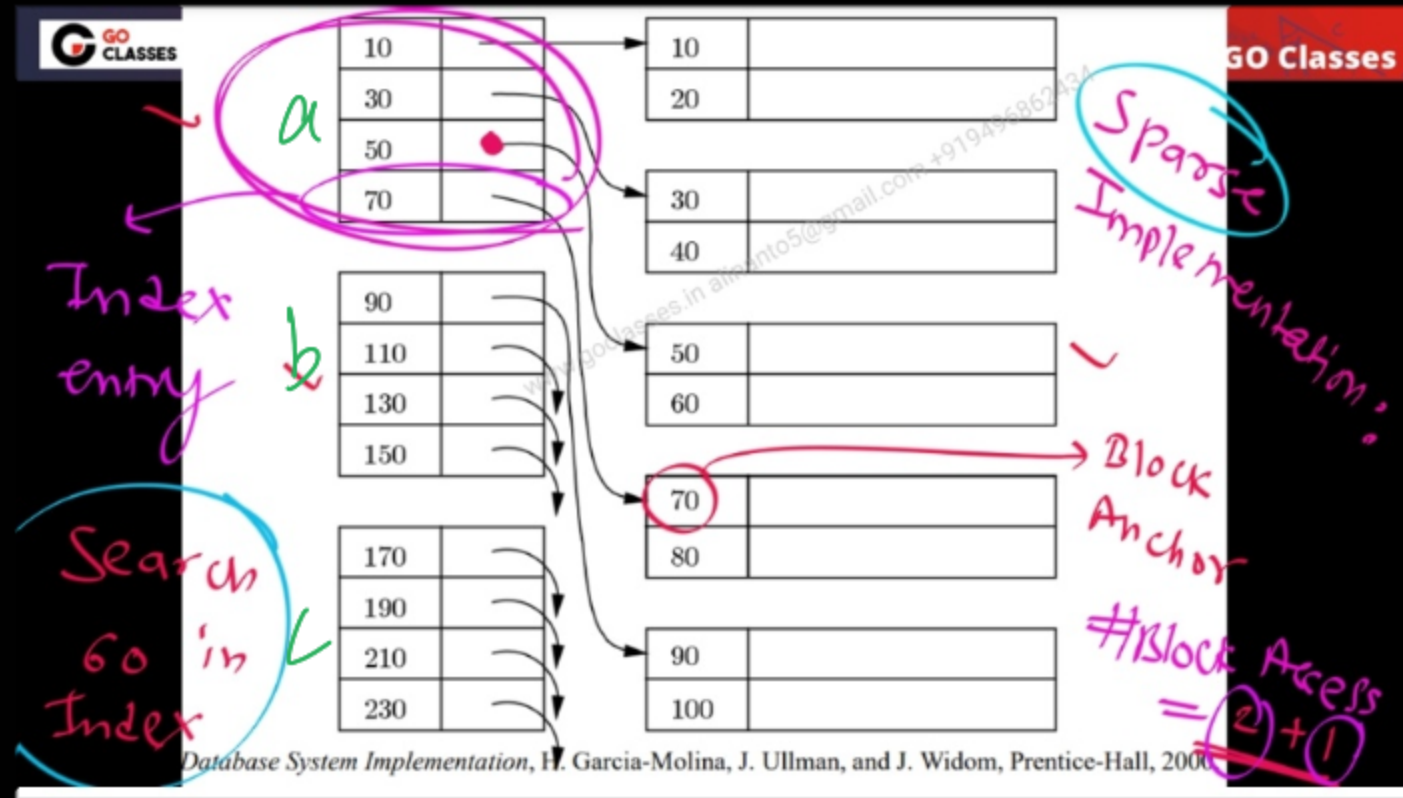


## Implementation of Index

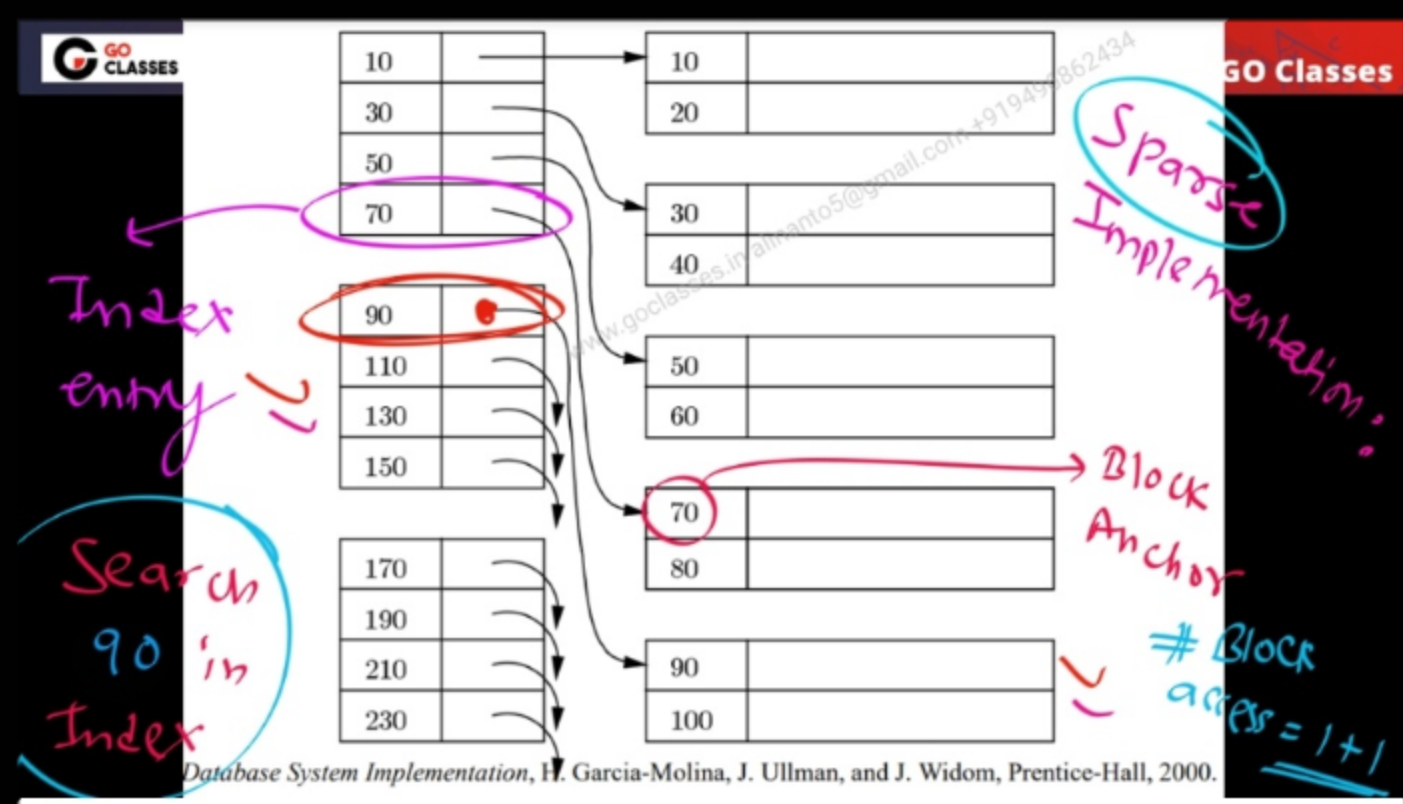
① Dense Implementation → Every data record has one Index record (not used for primary index)

② Sparse Implementation → Every block in file has one index record = primary index

Search Algorithm → Step I: Perform Binary search on Index blocks  
Step II: After search → 1 more fetch for datablock  
→ #Blocks accessed =  $\log_2(I) + 1$



→ Binary search: 1st access block b  
→  $60 < 90 \Rightarrow$  Next access block a  
→  $50 < 60 < 70 \Rightarrow$  Next access data block  
Total block access =  $2+1 = 3$



→ Total blocks accessed  
=  $1+1 = 2$

## Primary Sparse Index

- Search key value of anchor record of data block
- #index entries = #blocks

The first record in each block of the data file is called the **anchor record of the block**, or simply the **block anchor**.

## Access cost

- Without primary Index  
binary search directly on blocks of sequential data file  
→  $\lceil \log_2 B \rceil$  (worst case)

- With primary Index  
binary search on index blocks and one data block access after search  
→  $\lceil \log_2(I) \rceil + 1$

**An Example** ☆☆☆ OKF → Ordered Key Field

Data file:  
No. of blocks  $b = 9500$   
Block size  $B = 4KB$   
OKF length  $V = 15$  bytes  
Block pointer length  $p = 6$  bytes

Index file on OKF:  
No. of records  $r_i = 9500$   
Size of entry  $21$  bytes  
Blocking factor  $BF_i = \frac{4 \times 2^{10} B}{21 B} = \frac{4 \times 1024}{21} = 195$   
No. of blocks  $b_i = \frac{9500}{195} = 49$   
Max No. of block accesses for getting record using the primary index  
Max No. of block accesses for getting record without using primary index

$\lceil \log_2(b) \rceil + 1$   
=  $\lceil \log_2(49) \rceil + 1$   
=  $7 + 1$   
=  $8$   
worst case



Q: Suppose that we have an ordered file with  $r = 300,000$  records stored on a disk with block size  $B = 4,096$  bytes. File records are of fixed size and are unspanned, with record length  $R = 100$  bytes. Assume the file is ordered by a Key field  $V$ , & we make a search query to find the record having  $V=7$  then what is the worst case number of disk blocks we need to access for this query??

In above Question, if we search instead by non key field  
Worst case = 7500 blocks

$$\text{Worst case} = \lceil \log_2 b \rceil = \log_2 (7500) = \underline{13}$$

where  $b = \# \text{blocks}$

$$bfs = \left\lfloor \frac{4096}{100} \right\rfloor = 40$$

$$b = \# \text{blocks} = \frac{300,000}{40} = 7500$$

Now suppose that the ordering key field of the file is  $V = 9$  bytes long, a block pointer is  $P = 6$  bytes long, and we have constructed a primary index for the file.

The number of Blocks needed to store the Primary Index is??

$$\text{Index record} = 9 + 6 = 15 \text{ bytes long}$$

$$\text{indexing } bfs = \left\lfloor \frac{4096}{15} \right\rfloor = 273 \text{ records/block}$$

$$b_i \# \text{blocks for primary Index} = \left\lceil \frac{7500}{273} \right\rceil = 28 \text{ blocks}$$

$$\text{Worst case} = \lceil \log_2 b_i \rceil + 1 = 5 + 1 = \underline{6 \text{ blocks}}$$

## Clustering Index

→ If the datafile is sequentially ordered by a non-key (duplicates allowed) field  $F$ , then the index using search key  $F$  is called a clustering Index

If file records are physically ordered on a nonkey field—which *does not* have a distinct value for each record—that field is called the clustering field and the data file is called a clustered file. We can create a different type of index, called a clustering index, to speed up retrieval of all the records that have the same value for the clustering field. This differs from a primary index, which requires that the ordering field of the data file have a *distinct value* for each record.

- DNKF → Ordered Non Key Field
- We cannot create clustering Index on a heap data file
- At one point almost one clustered index possible in a data file.
- Both primary Index & Clustering index not possible on a single file simultaneously.

Index [Department] → Clustering Index

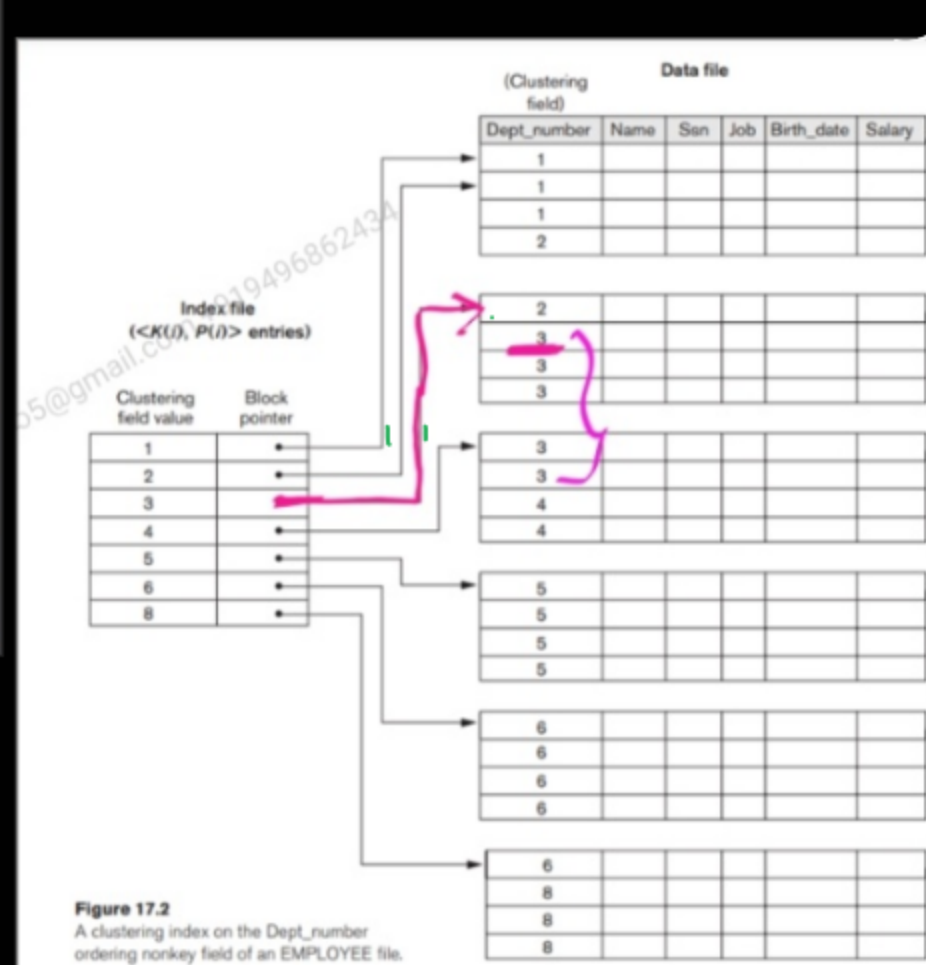
eg

| s-id  | Name       | Department | Salary |
|-------|------------|------------|--------|
| 76766 | Crick      | Biology    | 72000  |
| 10101 | Srinivasan | Comp. Sci. | 65000  |
| 45565 | Katz       | Comp. Sci. | 75000  |
| 83821 | Brandt     | Comp. Sci. | 92000  |
| 98345 | Kim        | Elec. Eng. | 80000  |
| 12121 | Wu         | Finance    | 90000  |
| 76543 | Singh      | Finance    | 80000  |
| 32343 | El Said    | History    | 60000  |
| 58583 | Califieri  | History    | 62000  |
| 15151 | Mozart     | Music      | 40000  |
| 22222 | Einstein   | Physics    | 95000  |
| 33465 | Gold       | Physics    | 87000  |

## Clustered Index Implementation

| Clustering field value | Block pointer |
|------------------------|---------------|
| Biology                | 76766         |
| Comp. Sci.             | 10101         |
| Elec. Eng.             | 45565         |
| Finance                | 83821         |
| History                | 98345         |
| Music                  | 12121         |
| Physics                | 76543         |

# Index entries = # Clusters  
= # Distinct Ordered Field Values



Search key Value, Block pointer →  
→ Point to that block which contains the first search key value entry.  
→ This block may/maynot contain records not relevant to current search key value.

Index entry: Distinct value  $V_i$  of the OF address of the first block that has a record with OF value  $V_i$

Index file: Ordered file (sorted on OF)  
size – no. of distinct values of OF

Clustering Index = Sparse Index



Q: Suppose that we consider the same ordered file with  $r = 300,000$  records stored on a disk with block size  $B = 4,096$  bytes. Imagine that it is ordered by the attribute Zipcode and there are 1,000 zip codes in the file (with an average 300 records per zip code, assuming even distribution across zip codes.) Assume 5-byte Zipcode and 6-byte block pointer.

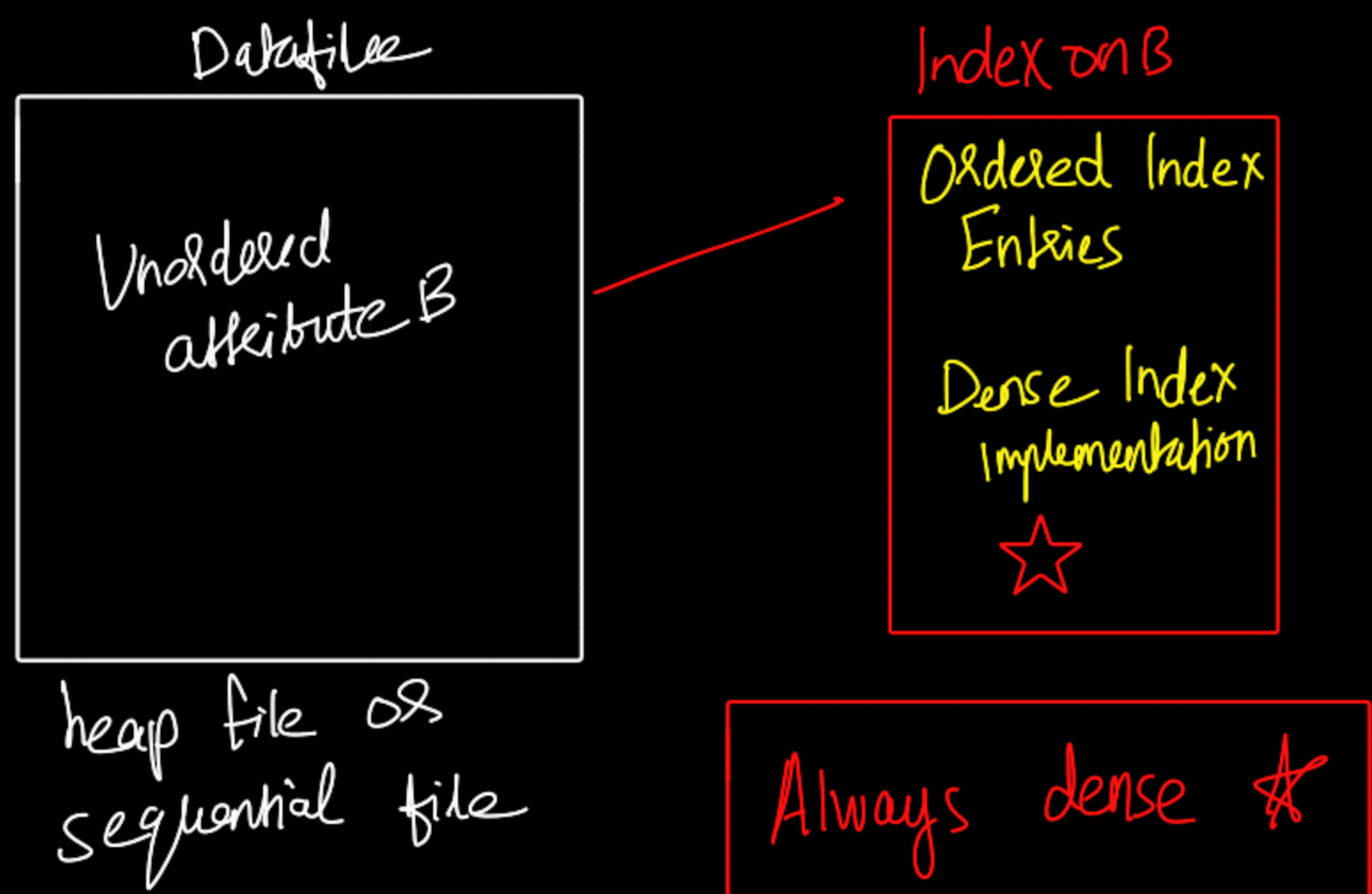
We create Index on Zipcode. Number of Index entries? Blocking Factor of the Index?? Number of Index Blocks?

$\# \text{clusters} = 1000 = \# \text{index entries}$       index entry size =  $5+6 = 11B$   
 Index blocking factor =  $\frac{4096}{11} = 372$   
 $\# \text{Index file blocks} = \lceil \frac{1000}{372} \rceil = 3 \text{ block for index}$

**Example 2.** Suppose that we consider the same ordered file with  $r = 300,000$  records stored on a disk with block size  $B = 4,096$  bytes. Imagine that it is ordered by the attribute Zipcode and there are 1,000 zip codes in the file (with an average 300 records per zip code, assuming even distribution across zip codes.) The index in this case has 1,000 index entries of 11 bytes each (5-byte Zipcode and 6-byte block pointer) with a blocking factor  $bfr_i = \lfloor (B/R_i) \rfloor = \lfloor (4,096/11) \rfloor = 372$  index entries per block. The number of index blocks is hence  $b_i = \lceil (r_i/bfr_i) \rceil = \lceil (1,000/372) \rceil = 3$  blocks. To perform a binary search on the index file would need  $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 3) \rceil = 2$  block accesses. Again, this index would typically be loaded in main memory (occupies 11,000 or 11 Kbytes) and takes negligible time to search in memory. One block access to the data file would lead to the first record with a given zip code.

## Secondary Index

→ Data file is unordered in attribute B → Index file on B is called secondary index



- Attribute B can be primary key or not
- Data file may be heap or sequential but not ordered in B.
- For heap files only secondary index possible
- Index entry can be record pointer or block pointer (default case for secondary index)
- Disk block access equivalent access lost to record

## Implementation of Secondary Index on Non Key

- When Secondary Index is created on non-key attribute B, there is high chance for value of B to repeat in data record
- This means for each search key value, there may be multiple relevant records scattered in the apoint
- Where will the index point

| account_number       | branch_name | balance |
|----------------------|-------------|---------|
| B <sub>1</sub> A-101 | Downtown    | 500     |
| B <sub>1</sub> A-102 | Perryridge  | 400     |
| B <sub>2</sub> A-201 | Brighton    | 900     |
| B <sub>2</sub> A-215 | Mianus      | 700     |
| B <sub>3</sub> A-217 | Brighton    | 750     |
| B <sub>3</sub> A-222 | Redwood     | 700     |
| B <sub>4</sub> A-305 | Round Hill  | 350     |

- Implementation 1**

  - Variable length index Entry
  - Single entry for a distinct search key value
  - Store multiple record pointers in one index entry corresponding for all relevant records
  - $\# \text{index entry} = \# \text{distinct search key values}$
- Implementation 2**

  - Fixed Length index Entry
  - $\# \text{index entry} = \# \text{records in datafile}$
  - Repeat index entry for a search key as much as required so that every relevant record pointer can be stored
- Implementation 3**

  - Fixed length index entry
  - $\# \text{index entry} = \# \text{distinct search key values}$
  - Each entry stores a pointer to a list of block pointers stored separately from the index file

INDEX

| Balance | Block Pointer                   |
|---------|---------------------------------|
| 350     | B <sub>4</sub>                  |
| 400     | B <sub>1</sub>                  |
| 500     | B <sub>1</sub>                  |
| 700     | B <sub>2</sub> , B <sub>3</sub> |
| 750     | B <sub>3</sub>                  |
| 900     | B <sub>2</sub>                  |

INDEX

| Balance | Block Pointer  |
|---------|----------------|
| 350     | B <sub>4</sub> |
| 400     | B <sub>1</sub> |
| 500     | B <sub>1</sub> |
| 700     | B <sub>2</sub> |
| 700     | B <sub>3</sub> |
| 750     | B <sub>3</sub> |
| 900     | B <sub>2</sub> |

INDEX

| Balance | Pointer                             |
|---------|-------------------------------------|
| 350     | a → B <sub>4</sub>                  |
| 400     | b → B <sub>1</sub>                  |
| 500     | c → B <sub>1</sub>                  |
| 700     | d → B <sub>2</sub> , B <sub>3</sub> |
| 750     | e → B <sub>3</sub>                  |
| 900     | f → B <sub>2</sub>                  |

- Implementation 3 is preferred.
- Implementation 3 is default case in secondary index

Option 3, which is more commonly used, is to keep the index entries themselves at a fixed length and have a single entry for each index field value, but to create an extra level of indirection to handle the multiple pointers. In this scheme, the pointer  $P(i)$  in index entry  $\langle K(i), P(i) \rangle$  points to a disk block, which contains a set of record pointers; each record pointer in that disk block points to one of the data file records with value  $K(i)$  for the indexing field. If some value  $K(i)$  occurs in too many records, so that their record pointers cannot fit in a single disk block, a cluster or linked list of blocks is used. This technique is illustrated in Figure 17.5. Retrieval via the



# Implementation of Secondary Index on Unordered Key Attribute

→ # Index Entries = # records in data file  
 → Index Entry :-  $\langle \text{value of search key, pointer to record} \rangle$

$$(b_i) \text{ Index BFR} = \left\lfloor \frac{B}{(V + P_r)} \right\rfloor$$

$$\# \text{ Index Blocks} = \left\lceil \frac{I}{b_i} \right\rceil$$

$$\text{Access cost} = \lceil \log_2 I \rceil + 1$$

$B$  = Block size  
 $V$  = Size of Index attribute  
 $P_r$  = Record pointer  
 $R$  = # records in data file  
 $I$  = # index entries

## An Example

### Data file:

No. of records  $r = 90,000$  Block size  $B = 4KB$   
 Record length  $R = 100$  bytes  $BF = \lceil 4096/100 \rceil = 40$ ,  
 $b = \lceil 90000/40 \rceil = 2250$   
 NOF length  $V = 15$  bytes length of a record pointer  $P_r = 7$  bytes

### Index file:

No. of records  $r_i = 90,000$  record length  $= V + P_r = 22$  bytes  
 $BF_i = \lceil 4096/22 \rceil = 186$  No. of blocks  $b_i = \lceil 90000/186 \rceil = 484$

Max no. of block accesses to get a record using the secondary index  $1 + \lceil \log_2 b_i \rceil = 10$   
 Avg no. of block accesses to get a record without using the secondary index  $b/2 = 1125$   
 A very significant improvement

## PRACTICE

### Database Management System GO Classes

Q 1: Consider relation  $R(A, B, C, D)$   
 Assume  $A$  is the Key. File  $R$  stored in 100 Blocks.

SELECT \* FROM  $R$  WHERE  $a = 10$ ;

No Index. File  $R$  uses Heap Organization.

Number of Disk Block Access needed??

b) File is ordered by  $A$

a) Best case - 1  
 Average case - 50  
 Worst case - 100

b) Worst case  
 $\lceil \log_2(100) \rceil = 7$

Q 12: Consider relation  $R(A, B, C, D)$   
 Assume  $A, B$  are two Keys.  
 File  $R$  is sequentially ordered by  $A$ .

How many Indexes can we create (using single attribute search key) simultaneously??  
 Mention their types.

Primary Index - 1  
 Clustering Index - 0  
 Secondary Index on key - 1  
 Secondary Index on non key - 2  
 Total 4

Q 16: Consider relation  $R$ .

We create an Index  $I$  on  $R$ . The (Physical) Order of search key in  $I$  is same or almost same as order of  $R$ .

Then  $I$  is?

→ For  $PI \rightarrow$  order exactly same. Hence not possible  
 → For  $CI \rightarrow$  order almost same except Index entries don't repeat records may.  
 →  $SI \rightarrow$  No order correlation: Hence not possible

3.7.5 Indexing: GATE CSE 2008 | Question: 16, ISRO2016-60

A clustering index is defined on the fields which are of type

- A. non-key and ordering ✓
- B. non-key and non-ordering
- C. key and ordering
- D. key and non-ordering

## GATE CSE 2013 | Question: 15

asked in Databases Sep 23, 2014

An index is clustered, if

- A. it is on a set of fields that form a candidate key
- B. it is on a set of fields that include the primary key
- C. the data records of the file are organized in the same order as the data entries of the index
- D. the data records of the file are organized not in the same order as the data entries of the index





A file is organized so that the ordering of the data records is the same as or close to the ordering of data entries in some index. Than that index is called

- A. Dense
- B. Sparse
- C. Clustered ✓
- D. Unclustered

Secondary Index  $\equiv$  Unclustered Index



A record id, or rid for short, is a unique identifier for a particular record in a set of records. An rid has the property that we can identify the disk address of the page containing the record by using the rid. The number of I/O's required to read a record, given a rid, is therefore 1 I/O.



Q) True or False?

Processing a range query (e.g., salary BETWEEN 10000 AND 20000) using a clustered index is always faster than using an unclustered index.

→ Results in contiguous block for clustered index  
→ Yes

### Q 31: True/False ??

Consider a relation stored as a randomly ordered file (i.e. Heap Organization) for which the only index is an unclustered (secondary) index on a field called sal.

If you want to retrieve all records with  $sal > 20$ , is using the index always the best alternative?

→ Depends on # Records in result  
→ For small # records in result → look into index  
→ For large # records → directly access datafile  
→ Since after looking into index blocks, we might still need to look into majority of datablocks

Q) True or False

Processing a range query using an unclustered index is always faster than using a file scan.

→ False  
→ Reasoning for above question

**Table 17.1** Types of Indexes Based on the Properties of the Indexing Field

|                          | Index Field Used for Physical Ordering of the File | Index Field Not Used for Physical Ordering of the File |
|--------------------------|----------------------------------------------------|--------------------------------------------------------|
| Indexing field is key    | Primary index                                      | Secondary index (Key)                                  |
| Indexing field is nonkey | Clustering index                                   | Secondary index (NonKey)                               |

**Table 17.2** Properties of Index Types

| Type of Index      | Number of (First-Level) Index Entries                                                | Dense or Nondense (Sparse) | Block Anchoring on the Data File |
|--------------------|--------------------------------------------------------------------------------------|----------------------------|----------------------------------|
| Primary            | Number of blocks in data file                                                        | Nondense                   | Yes                              |
| Clustering         | Number of distinct index field values                                                | Nondense                   | Yes/no <sup>a</sup>              |
| Secondary (key)    | Number of records in data file                                                       | Dense                      | No                               |
| Secondary (nonkey) | Number of records <sup>b</sup> or number of distinct index field values <sup>c</sup> | Dense or Nondense          | No                               |



# Multilevel Indexes & ISAM

For single Index  $\rightarrow$  Binary search access cost =  $\lceil \log_2(b_i) \rceil + 1$

Ways to reduce this further ①  $\rightarrow$  Key entire Index in Main memory  
 $\rightarrow$  Not feasible for multiple large indexes  
 $\rightarrow$

② Use multilevel Indexing

$\rightarrow$  Our Index file is always ordered

$\rightarrow$  Our Index file has unique & distinct values in Search key field

$\rightarrow$  Search key is like primary key when we consider Index file as another data file

$\Rightarrow$  So we can use a primary Index one more level higher & so on until the highest level index only has one block.

4th level (IL4)

| SK | Pointer |
|----|---------|
| ~  | ~       |
| ~  | ~       |

$\rightarrow$  only 1 block  
 $\rightarrow$  PI on IL3  
 $\rightarrow$  Called top level

3rd level (IL3)

| SK | Pointer |
|----|---------|
| ~  | ~       |
| ~  | ~       |
| ~  | ~       |

$\rightarrow$  PI on IL2

2nd level (IL2)

| SK | Pointer |
|----|---------|
| ~  | ~       |
| ~  | ~       |
| ~  | ~       |
| ~  | ~       |

$\rightarrow$  PI on IL1

INDEX FILE (IL1)

| SK | DataBlock Pointer |
|----|-------------------|
| ~  | ~                 |
| ~  | ~                 |
| ~  | ~                 |
| ~  | ~                 |
| ~  | ~                 |

Ordered, distinct & unique

$\rightarrow$  PI, CI or SI on datafile

$\rightarrow$  Always type primary & sparse  
 $\rightarrow$  Index level K is PI (sparse) on level K-1

## Making the Index Multi-level

Index file – itself an ordered file

– another level of index can be built

Multilevel Index –

Successive levels of indices are built till the last level has one block

height – no. of levels

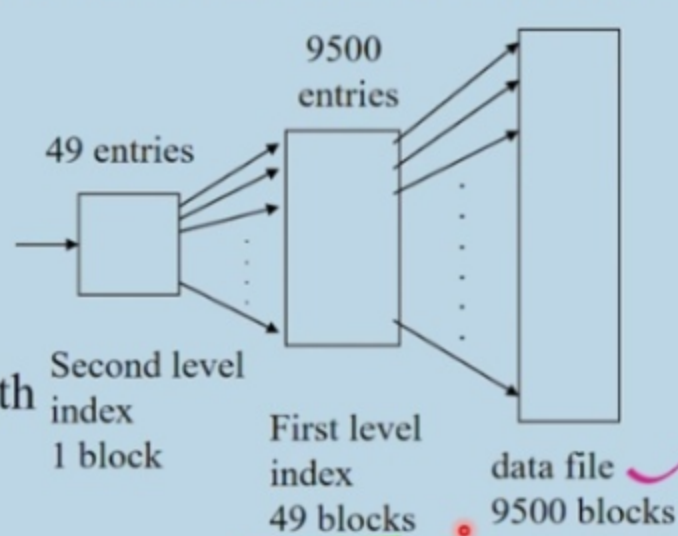
block accesses: height + 1

(no binary search required)

For the example data file:

No of block accesses required with multi-level primary index: 3

without any index: 14



Access cost = height + 1

height = #Index levels

Access cost for MLI is less than Binary search



17.18. Consider a disk with block size  $B = 512$  bytes. A block pointer is  $P = 6$  bytes long, and a record pointer is  $P_R = 7$  bytes long. A file has  $r = 30,000$  EMPLOYEE records of fixed length. Each record has the following fields: Name (30 bytes), Ssn (9 bytes), Department\_code (9 bytes), Address (40 bytes), Phone (10 bytes), Birth\_date (8 bytes), Sex (1 byte), Job\_code (4 bytes), and Salary (4 bytes, real number). An additional byte is used as a deletion marker.

a. Calculate the record size  $R$  in bytes.

b. Calculate the blocking factor  $bfr$  and the number of file blocks  $b$ , assuming an unspanned organization.

$$a) R = 30 + 9 + 9 + 40 + 10 + 8 + 1 + 4 + 4 + 1$$

$$R = 116$$

$$b) bfr = \left\lfloor \frac{512}{116} \right\rfloor = 4$$

$$\# \text{ file blocks} = \frac{30,000}{4} = 7500 \text{ blocks}$$

$$c-i) b_i = \text{Index } bfr = \left\lfloor \frac{512}{9+6} \right\rfloor = \left\lfloor \frac{512}{15} \right\rfloor = 34$$

$$c-ii) \# \text{ First level Index entries} = 7500$$

$$\# \text{ First level Index block} = \left\lceil \frac{7500}{34} \right\rceil = 221$$

$$c-v) \# \text{ access cost} = \text{height} + 1$$

$$= 3 + 1 = 4 \text{ blocks}$$

$$c-iii) \# \text{ Second level entry} = 221$$

$$\# \text{ Second level block} = \left\lceil \frac{221}{34} \right\rceil = 7$$

$$\# \text{ Third level entry} = 7$$

$$\# \text{ Third level blocks} = \left\lceil \frac{7}{34} \right\rceil = 1$$

$$c-iv) \text{ Total } \# \text{ block} = 1 + 7 + 221 = 229 \text{ blocks}$$

d. Suppose that the file is not ordered by the key field Ssn and we want to construct a secondary index on Ssn. Repeat the previous exercise (part c) for the secondary index and compare with the primary index.

$$d-iii) \text{ IL2 index entry size} = 9 + 6 = 15$$

$$\text{IL2 } bfr = \left\lfloor \frac{512}{15} \right\rfloor = 34$$

$$\text{IL2 } \# \text{ entries} = 938$$

$$\text{IL2 } \# \text{ blocks} = \left\lceil \frac{938}{34} \right\rceil = 28$$

$$\text{IL3 } \# \text{ entries} = 28$$

$$\text{IL3 } \# \text{ blocks} = \left\lceil \frac{28}{34} \right\rceil = 1$$

d-i) Index entry size =  $9 + 7 = 16$  *★ Record pointer instead of Block pointer*

$$d-ii) \# \text{ Index entries} = \# \text{ records} = 30,000$$

$$\# \text{ IL1 } bfr = \left\lfloor \frac{512}{16} \right\rfloor = 32$$

$$\# \text{ IL1 index blocks} = \left\lceil \frac{30,000}{32} \right\rceil = 938$$

d-iv

$$\# \text{ Total blocks} = 1 + 28 + 938 = 967 \text{ blocks in MLI}$$

$$d-v) \text{ Access cost} = \text{height} + 1 = 3 + 1 = 4$$

e. Suppose that the file is not ordered by the nonkey field Department\_code and we want to construct a secondary index on Department\_code, using option 3 of Section 17.1.3, with an extra level of indirection that stores record pointers. Assume there are 1,000 distinct values of Department\_code and that the EMPLOYEE records are evenly distributed among these values. Calculate (i) the index blocking factor  $bfr$ , (which is also the index fan-out  $fo$ ); (ii) the number of blocks needed by the level of indirection that stores record pointers; (iii) the number of first-level index entries and the number of first-level index blocks; (iv) the number of levels needed if we make it into a multilevel index; (v) the total number of blocks required by the multilevel index and the blocks used in the extra level of indirection; and (vi) the approximate number of block accesses needed to search for and retrieve all records in the file that have a specific Department\_code value, using the index.

$$i) \text{ IL2 } \# \text{ entries} = 30$$

$$bfr = 34$$

$$\# \text{ blocks} = 1$$

only 2 levels

$$v) \text{ Blocks} = 1 + 30 + 1000 = 1031 \text{ blocks}$$

$$vi) 2 \text{ level} + 1 \text{ indirection} + \frac{1000}{34} = 4$$

Worst case for data match def

IL1

$$i) \text{ entries} = \# \text{ distinct search key value} = 1000$$

$$\text{Index entry size} = 9 + 6$$

for department field value  
Block pointer → points to blocks used for indirection

$$bfr = \left\lfloor \frac{512}{15} \right\rfloor = 34$$

$$ii) \# \text{ Index blocks} = \left\lceil \frac{1000}{34} \right\rceil = 30$$

$$iii) \text{ Indirection entries} = \frac{30000}{1000} = 30 \text{ (assuming evenly distributed)}$$

$$\text{size of indirection pointer} = 7B$$

(given indirection stores record pointer)

$$\Rightarrow \# \text{ bytes required for 1 distinct value for indirection} = 7B \times 30 = 210B$$

$$\text{Block required} = 1 \times 1000 = 1000$$



f) Repeat c if the file is ordered by nonkey field dep-code  $\rightarrow$  clustered index F.W

Let  $x_1$  be the # Index entries in the first level index  
 Let  $bfs_i$  be the blocking factor for all levels assuming it is same for all levels  
 $\rightarrow$  This is true because the index entry is same size

$\rightarrow bfs_i = \text{fan out}(f_0)$  for MLI

$\rightarrow$  The number of levels  $= t$ , then

$$1 \leq x_1 / f_0^t$$

$$t = \lceil \log_{f_0}(x_1) \rceil$$

+ disk block accesses for index search  
 + 1 for data usually.

## ISAM - Indexed sequential access method.

$\rightarrow$  By default  $\rightarrow$  ISAM = Multilevel + PI (Sequential file + Ordered by Key Field)  
 $\rightarrow$  ISAM = MLI + PI(DKF)

Also possible  $\rightarrow$  ISAM = MLI + CI(DNKF)  $\rightarrow$  Clustered Index

$\rightarrow$  More suitable for files that are relatively static. ★★★★★

A common file organization used in business data processing is an ordered file with a multilevel primary index on its ordering key field. Such an organization is called an indexed sequential file and was used in a large number of early IBM systems.

3.7.3 Indexing: GATE CSE 1993 | Question: 14

<https://gateoverflow.in/2311>



An ISAM (indexed sequential) file consists of records of size 64 bytes each, including key field of size 14 bytes. An address of a disk block takes 2 bytes. If the disk block size is 512 bytes and there are 16K records, compute the size of the data and index areas in terms of number blocks. How many levels of tree do you have for the index?

$$R = 64B, \text{ Disk Block} = 512B, x = 16K; bfs = \lfloor \frac{512}{64} \rfloor = 8, \# \text{ data blocks} = \frac{16000}{8} = 2000 \text{ blocks}$$

IL1

$$\text{Size of entry} = 14 + 2 = 16B$$

$$bfs_i = f_0 = \lfloor \frac{512}{16} \rfloor = 32$$

$$\# \text{ entries} = 2000 \text{ (sparse)}$$

$$\# \text{ blocks} = \lceil \frac{2000}{32} \rceil = 63$$

IL2

$$\# \text{ entries} = 500$$

$$\# \text{ blocks} = \lceil \frac{500}{32} \rceil = 2$$

$$\Rightarrow \# \text{ levels}(t) = \lceil \log_{32} 2000 \rceil = 3 \text{ levels}$$

IL3

$$\# \text{ entries} = 2$$

$$\# \text{ blocks} = 1$$

$$\text{Total Index blocks} = 1 + 2 + 63 = 66 \text{ index blocks}$$





One giga bytes of data are to be organized as an indexed-sequential file with a uniform blocking factor 8. Assuming a block size of 1 Kilo bytes and a block referencing pointer size of 32 bits, find out the number of levels of indexing that would be required and the size of the index at each level. Determine also the size of the master index. The referencing capability (fanout ratio) per block of index storage may be considered to be 32.

Block size = 1024B  
bfr = 8  
(fo) fanout = 32  
Data size = 1GB  
Block pointers 32B

Question about index only, not about data, so # records in data is not required since we only need # blocks in data assuming ISAM (given) (MLI + PK (DKF))

# data block =  $\frac{1GB}{1KB} = \frac{1KB \times 2^{20}}{1KB} = 2^{20}$  data

# IL1 entry =  $2^{20}$   
# levels for MLI (t) =  $\lceil \log_{32}(2^{20}) \rceil = \log_{32}(2^5)^4 = \log_{32}(32)^4 = 4 \text{ levels}$

IL1  
#entry =  $2^{20}$   
#blocks =  $\frac{2^{20}}{32} = 2^{15}$  blocks

IL2  
#entries =  $2^{15}$   
#blocks =  $\frac{2^{15}}{32} = 2^{10}$  blocks

IL3  
#entries =  $2^{10}$   
#blocks =  $2^5$

IL4  
#entries =  $2^5$   
#blocks = 1

Total index size =  $2^{15} + 2^{10} + 2^5 + 1$  blocks  
=  $32768 + 1024 + 32 + 1 = 33825$  blocks

→ data file bfr & size of block pointers are redundant info in question to create confusion



Consider a file of 16384 records. Each record is 32 bytes long and its key field is of size 6 bytes. The file is ordered on a non-key field, and the file organization is unspanned. The file is stored in a file system with block size 1024 bytes, and the size of a block pointer is 10 bytes. If the secondary index is built on the key field of the file, and a multi-level index scheme is used to store the secondary index, the number of first-level and second-level blocks in the multi-level index are respectively

- ✓ A. 8 and 0  
B. 128 and 6  
C. 256 and 4  
D. 512 and 5

SI (VDKF) ⇒ # IL1 entries = # data records = 16384

Record pointers not given in Q

IL1  
entry size = 6 + 10 = 16B  
bfr =  $\frac{1024}{16} = 64$

IL2  
#entries = 256  
#blocks =  $\lceil \frac{256}{64} \rceil = 4$

IL3  
#blocks = 1

t =  $\lceil \log_{64}(16384) \rceil$   
t =  $\lceil \log_{2^6}(2^{14}) \rceil$

#blocks =  $\lceil \frac{16384}{64} \rceil = 256$

t =  $\frac{14}{6} = 2.33 = 3$

### Disadvantage of Ordered (Static) Index

- Costly to maintain for dynamic files
- Preferable only when data is mostly static
- insertion, deletion, modify
  - Changes anchored record
  - need to update all subsequent blocks
  - Almost entire index needs to be updated
  - if MLI even more costly



# Dynamic Indexes — B tree, B+ tree

→ Dynamic — multilevel tree based Indexes  
→ Modification friendly

BST — Binary search tree  
AVL Tree  
Heap  
2-3-4 Tree

Main memory data structure not suitable for disk  
→ We cannot use a block for every new Node  
→ We cannot keep multiple nodes in one block → pointer issue

The in-memory Data Structures are NOT suitable for Disk Data Structures... WHY?

Ask yourself:

What should be a NODE in the Tree??

In main memory, a Node was a Structure, having few Bytes only, So, suitable for Memory as we access main memory in Words...

BUT Disk Access happens Block by Block & Block size is VERY Large than the main memory word size...

So, If we use in-memory data structures for Disk, A LOT of space per block will be Wasted...

## 1. Binary Search Tree (BST)

A **BST** is a binary tree with ordering rules:

- Left child < Parent
- Right child > Parent
- No duplicate keys (usually)

### Properties

- Search: average  $O(\log n)$ , worst  $O(n)$  (when unbalanced)
- Insert/Delete: same as above

### Issue

BST can become **skewed** (like a linked list) if elements are inserted in sorted order.

### Use cases

- Implementing sets, maps
- Searching sorted data
- In-order traversal gives sorted order

## 2. AVL Tree (Balanced BST)

An **AVL Tree** is a **self-balancing BST**.

It maintains the rule:

**Balance Factor** =  $\text{height}(\text{left}) - \text{height}(\text{right})$  must be -1, 0 or +1

If violated, the tree performs **rotations** (LL, RR, LR, RL).

### Properties

- Search:  $O(\log n)$  worst case
- Insert/Delete:  $O(\log n)$  (rotations may occur)
- Always height-balanced

### Use cases

- When fast lookup is MOST important
- Databases, memory indexing (rare now; often replaced by B-trees)

## Summary Table

| Structure  | Balanced?   | Search                         | Insert/Delete           | Typical Use               |
|------------|-------------|--------------------------------|-------------------------|---------------------------|
| BST        | ✗ No        | Avg $O(\log n)$ , Worst $O(n)$ | Same                    | Basic searching, learning |
| AVL Tree   | ✓ Strict    | $O(\log n)$                    | $O(\log n)$ + rotations | Fast searches             |
| Heap       | ✗ Not a BST | $O(n)$ search                  | $O(\log n)$             | Priority queues           |
| 2-3-4 Tree | ✓ Yes       | $O(\log n)$                    | $O(\log n)$             | Databases, file systems   |

## 3. Heap (Binary Heap)

A **heap** is a **complete binary tree** with **heap property**:

- **Min-heap**: parent  $\leq$  children
- **Max-heap**: parent  $\geq$  children

Not a BST → no ordered left/right property.

### Properties

- Insert:  $O(\log n)$
- Delete-min / delete-max:  $O(\log n)$
- Search arbitrary element:  $O(n)$  (not sorted)

### Implemented using

- Arrays (very efficient)

### Use cases

- Priority Queues
- Dijkstra's algorithm
- Heap sort
- Event simulation systems

## 4. 2-3-4 Tree (B-tree of order 4)

A **2-3-4 tree** is a **multi-way self-balancing search tree**.

Each node can have:

- **2-node**: 1 key, 2 children
- **3-node**: 2 keys, 3 children
- **4-node**: 3 keys, 4 children

### Properties

- Always perfectly balanced
- Height is  $O(\log n)$  but very small (shallower than AVL)
- Insert/Delete:  $O(\log n)$

### Special feature

Nodes can split when full.

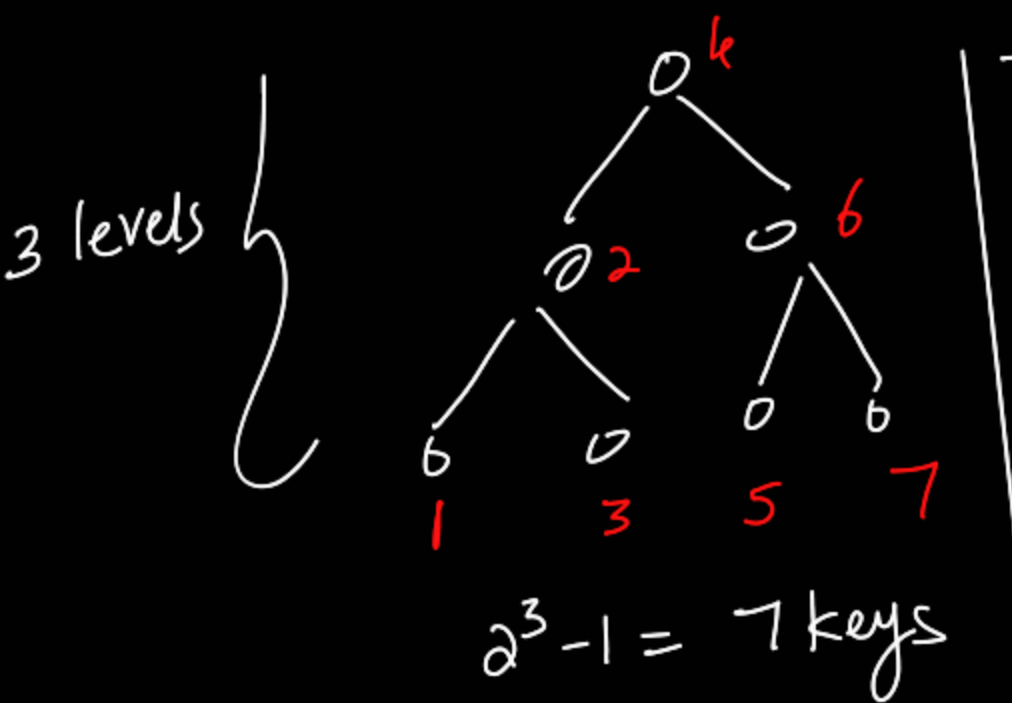
### Use cases

- Basis for **Red-Black Trees**
- Used in **file systems, databases**, where disk access matters
- Preferred when data must stay balanced with minimal height



Q) Assume 1M records on disk.

Balanced BST or AVL Tree  $\rightarrow$  How many disk block access



$\rightarrow$  each node stores one value of distinct search key value  
 $\rightarrow \# \text{ keys} = 2^{\text{level}} - 1$

$\Rightarrow$  for 1M

$2^{20} > 1M$   
 $2^{19} < 1M$  } Assuming dense

$\Rightarrow$  At least 20 levels required for Balanced BST or AVL Tree

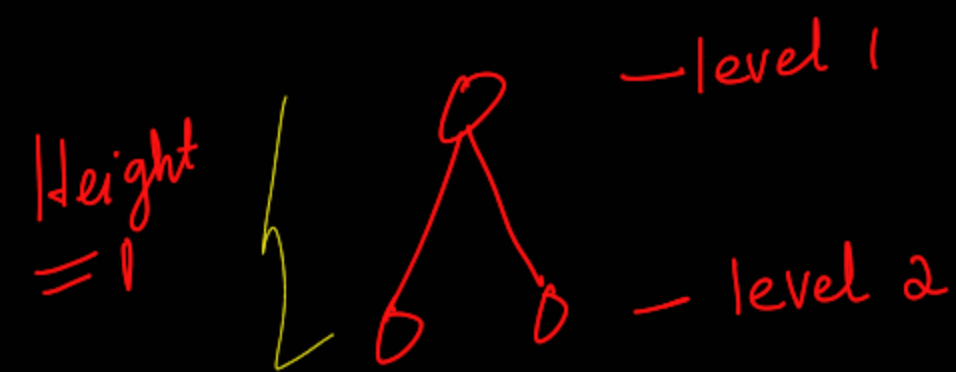
$\Rightarrow$  20 disk block access for search + 1 disk block access for data

B tree } idea  
 B+ tree }

- Put a lot of index entries in a single node
- 1 node  $\equiv$  1 block
- Since multiple entries per block  $\Rightarrow$  no wastage

Def Convention

• Root  $\rightarrow$  level 1

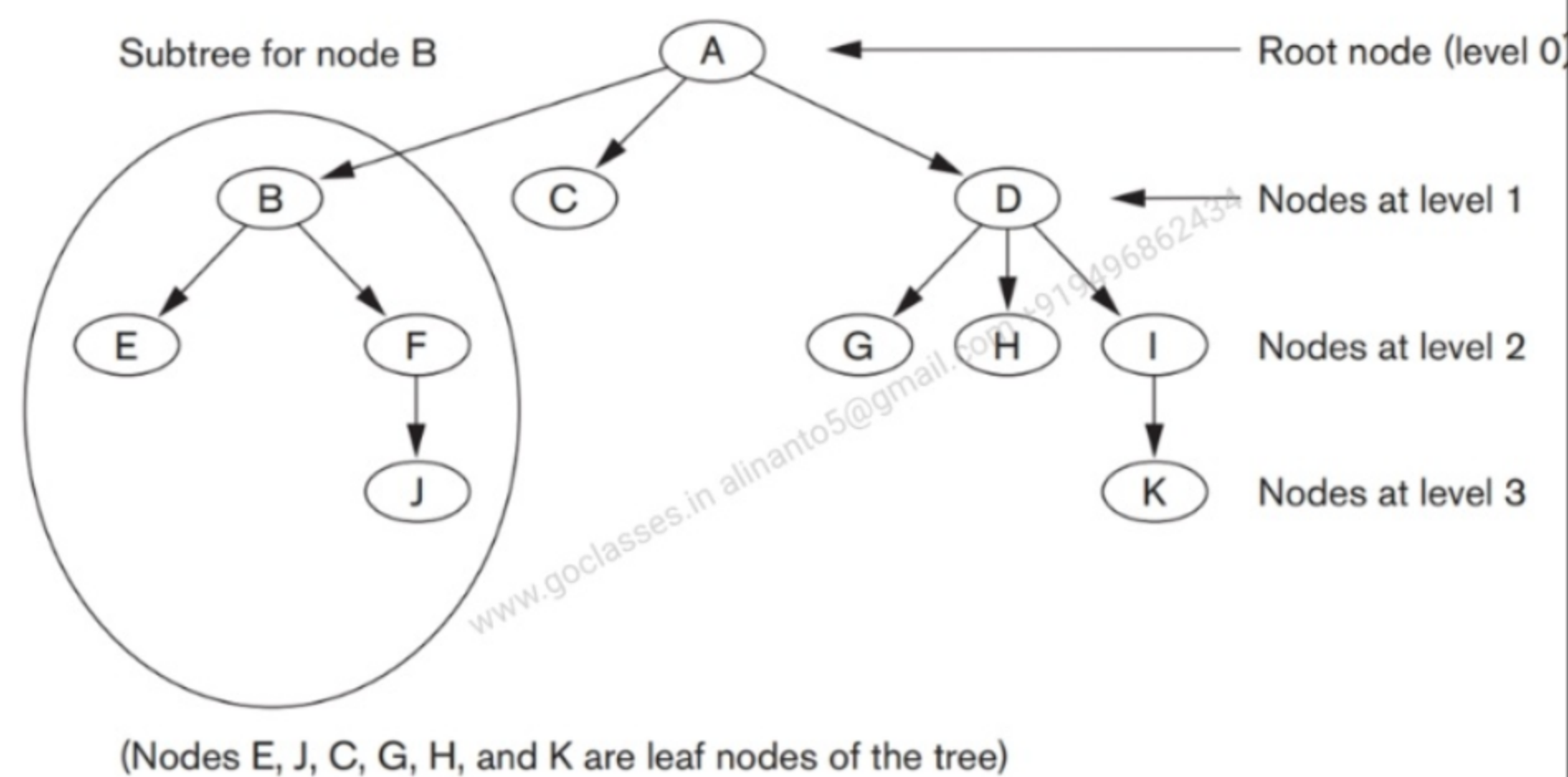


• Height is #edges from root to leaf

B-trees and B<sup>+</sup>-trees are special cases of the well-known search data structure known as a **tree**. We briefly introduce the terminology used in discussing tree data structures. A **tree** is formed of **nodes**. Each node in the tree, except for a special node called the **root**, has one **parent** node and zero or more **child** nodes. The root node has no parent. A node that does not have any child nodes is called a **leaf** node; a nonleaf node is called an **internal** node. The **level** of a node is always one more than the level of its parent, with the level of the root node being zero.<sup>7</sup> A **subtree** of a node consists of that node and all its **descendant** nodes—its child nodes, the child nodes of its child nodes, and so on. A precise recursive definition of a subtree is that it consists of a node  $n$  and the subtrees of all the child nodes of  $n$ . Figure 17.7 illustrates a tree data structure. In this figure the root node is A, and its child nodes are B, C, and D. Nodes E, J, C, G, H, and K are leaf nodes. Since the leaf nodes are at different levels of the tree, this tree is called **unbalanced**.

Figure 17.7

A tree data structure that shows an unbalanced tree.

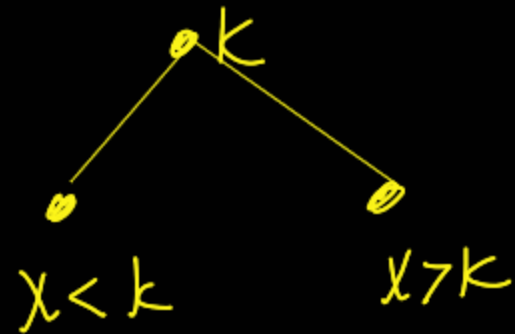


Search Trees

$\Rightarrow$  left child  $\leq$  parent  
 right child  $\geq$  parent

$\rightarrow$  in main memory  $\rightarrow$  BST, AVL

$\rightarrow$  in disk  $\rightarrow$  BTree, B<sup>+</sup> Tree



A node in a search tree  $\rightarrow$

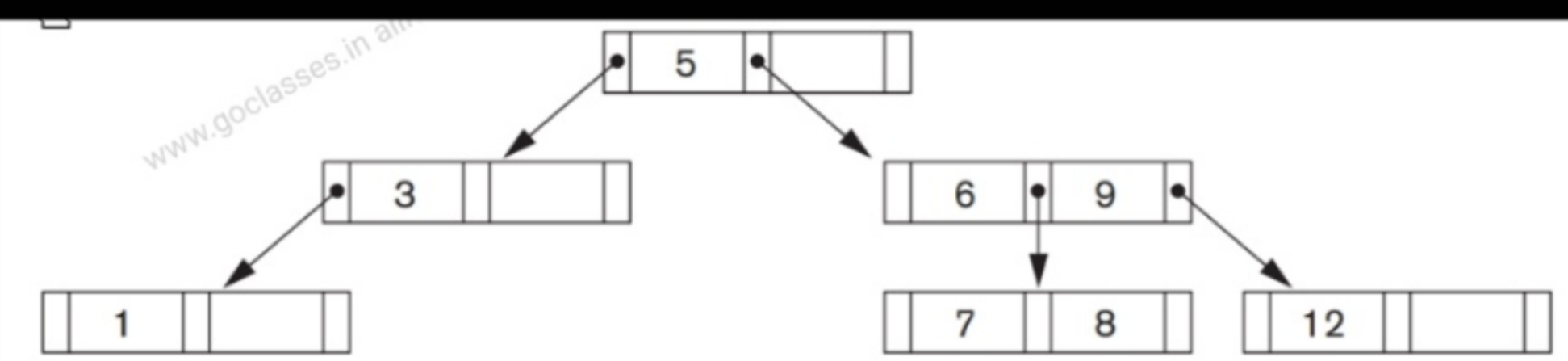


child  $x < K_1$

child  $x \mid K_1 < x < K_2$

child  $x > K_{j-1}$

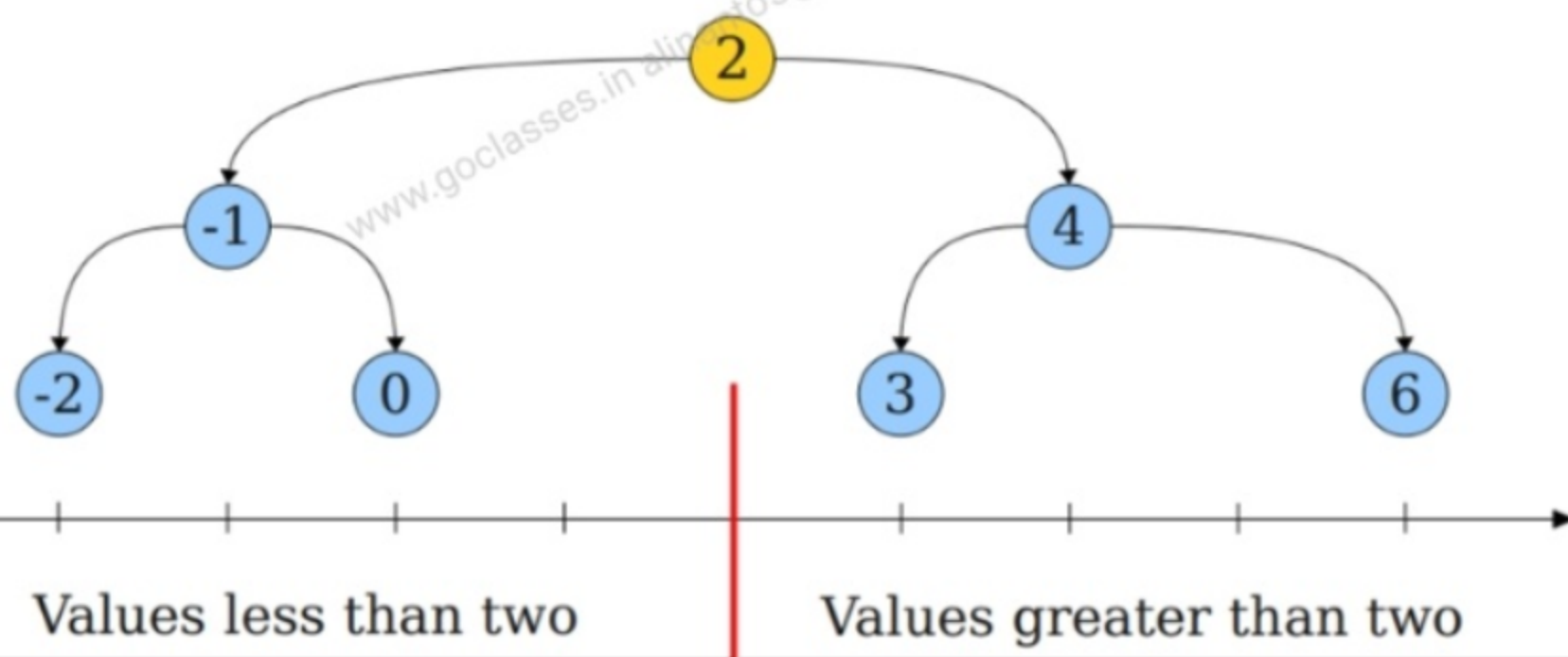
$P_i \rightarrow$  points for child  $i$  with value  $x$   
 such that  $K_{i-1} < x < K_i$



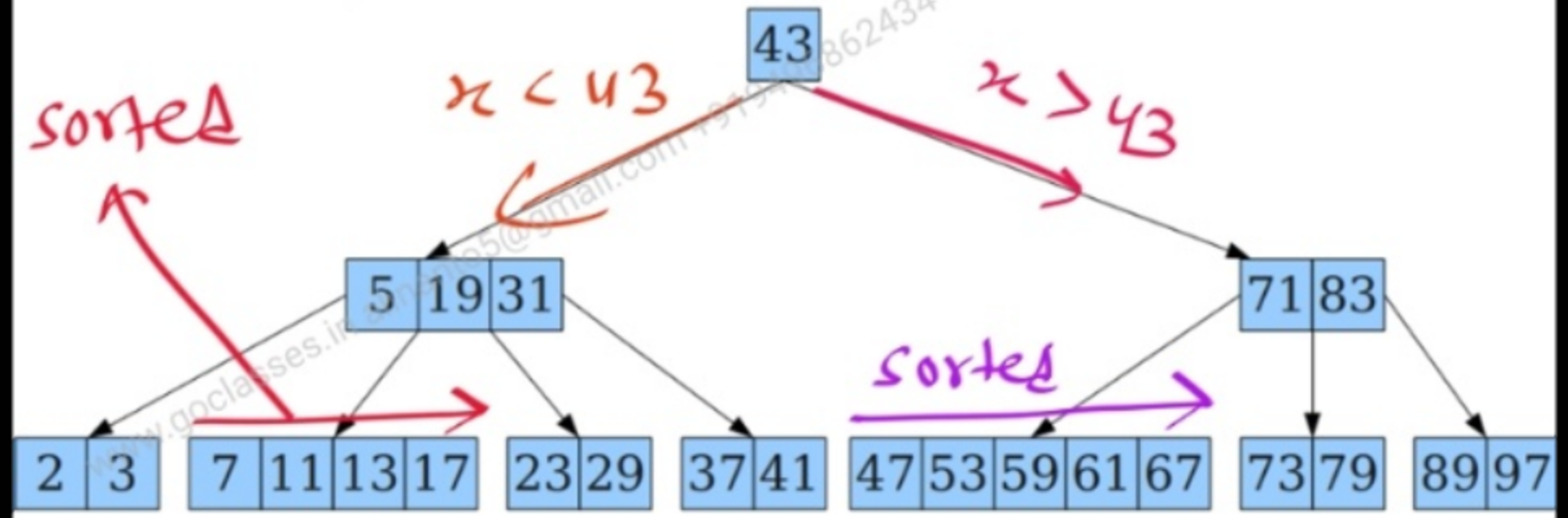


# Generalizing BSTs

- In a binary search tree, each node stores a single key.
- That key splits the "key space" into two pieces, and each subtree stores the keys in those halves.



- In a **multiway search tree**, each node stores an arbitrary number of keys in sorted order.



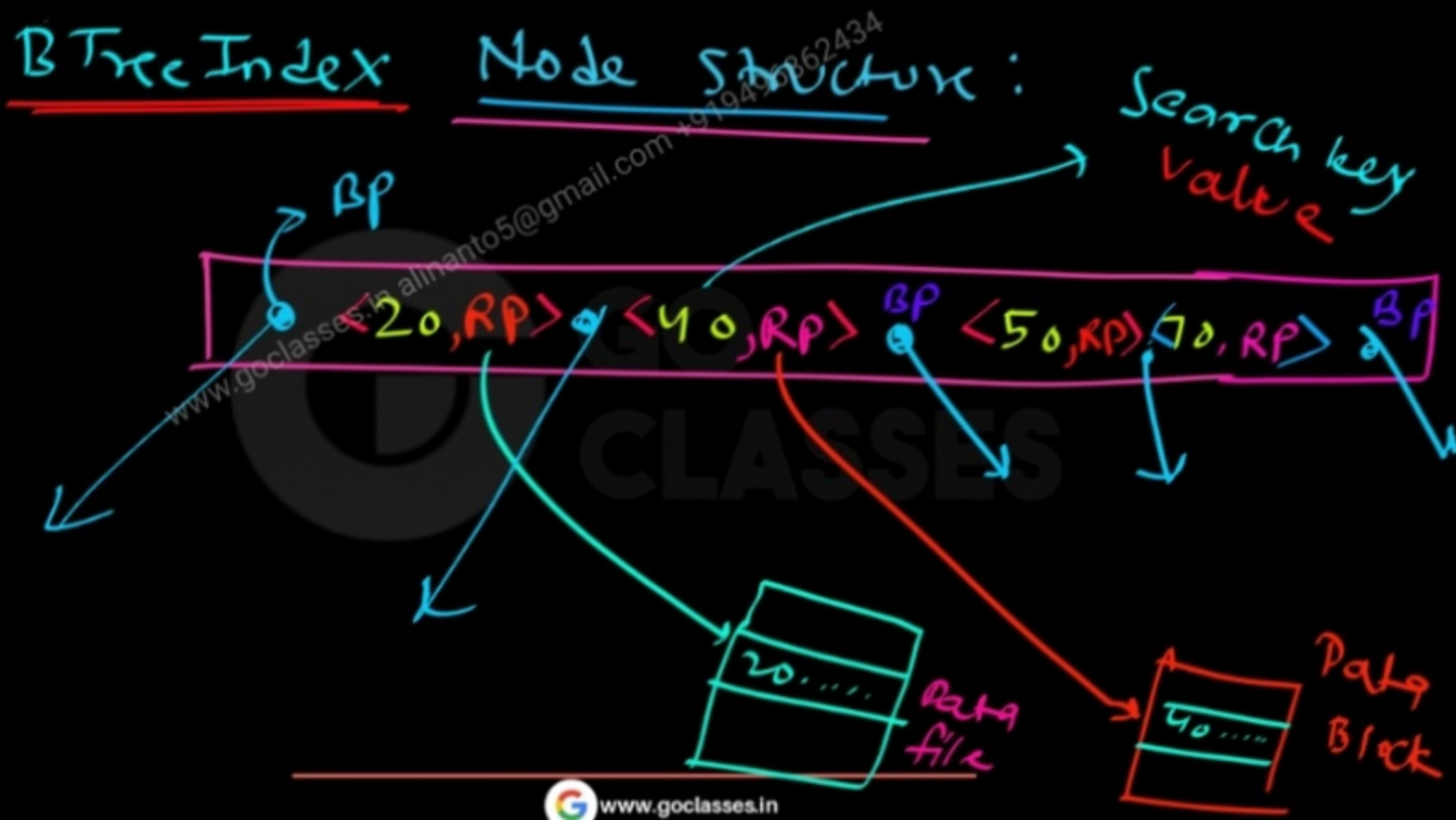
- In a node with  $k$  keys splits the "key space" into  $k + 1$  pieces, and each subtree stores the keys in those pieces.

→ Our goal

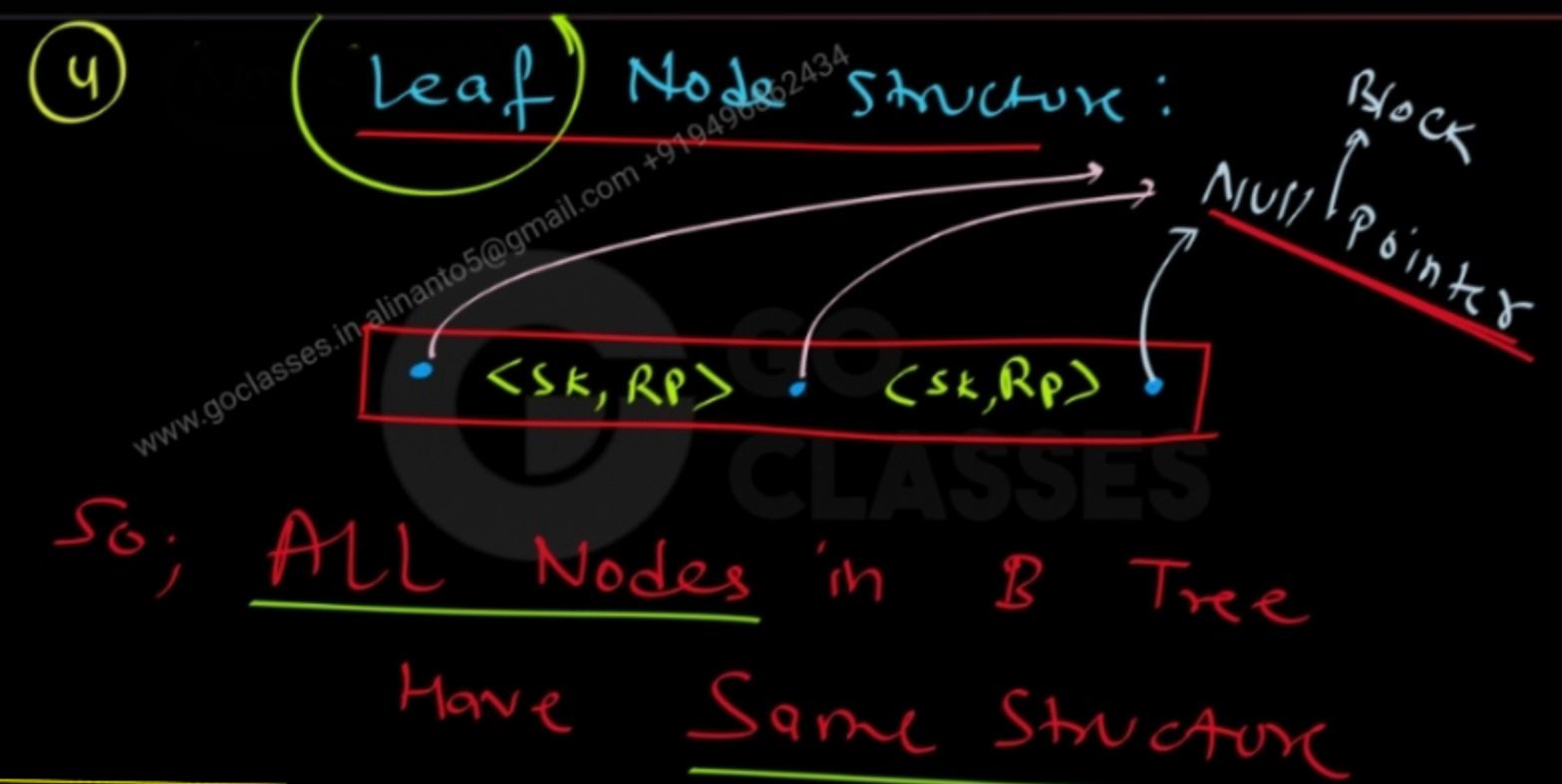
- Minimise #levels  $\Rightarrow$  reduced access cost for search
- Maximise Block space utilisation  $\Rightarrow$  Nodes as full as possible

Solution — B Tree

- Order of B Tree  $\rightarrow$  Maximum #child pointers ( $P$ )
- There are two types of pointers in DBMS file organisation
  - Block pointers
  - Child pointers
 But in BTree, Node pointers are block pointers only. Data pointers are Record pointers by default.
- Additional Constraints  $\Rightarrow$ 
  - All leaf nodes must be on same level (balanced)
  - Space utilisation  $> 50\%$   $\rightarrow$  Exception for root node.



|            | #children allowed                  | Exception               |
|------------|------------------------------------|-------------------------|
| Root node: | 2 to $P$                           |                         |
| non root : | $\lceil \frac{P}{2} \rceil$ to $P$ | ( $> 50\%$ utilisation) |



Key points :- All leaf nodes at same level

Maximum no. of search key values per node =  $p-1$

Maximum no. of child nodes (node pointers) per node =  $p$   $\rightarrow$  Order of B tree

If a node has  $x$  BP, then #keys in this node =  $x-1$

Minimum 50% utilization of all nodes except for root node

Root node  $\rightarrow 2$  to  $P$  pointers | Root node  $\rightarrow 2-1$  to  $P-1$  keys  
 non root node  $\lceil \frac{P}{2} \rceil$  to  $P$  | non root node  $\lceil \frac{P}{2} \rceil - 1$  to  $P-1$  keys

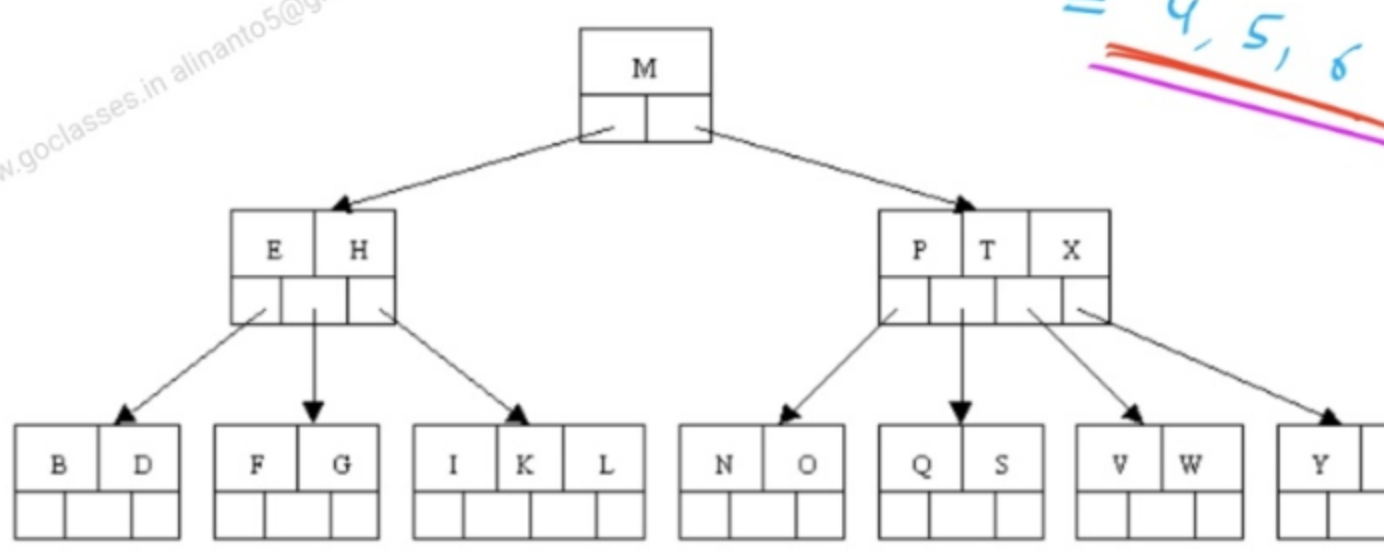
Q: for the following B Tree: possible orders = 4, 5, 6

☆  $\rightarrow$  Order of B tree can never be 1

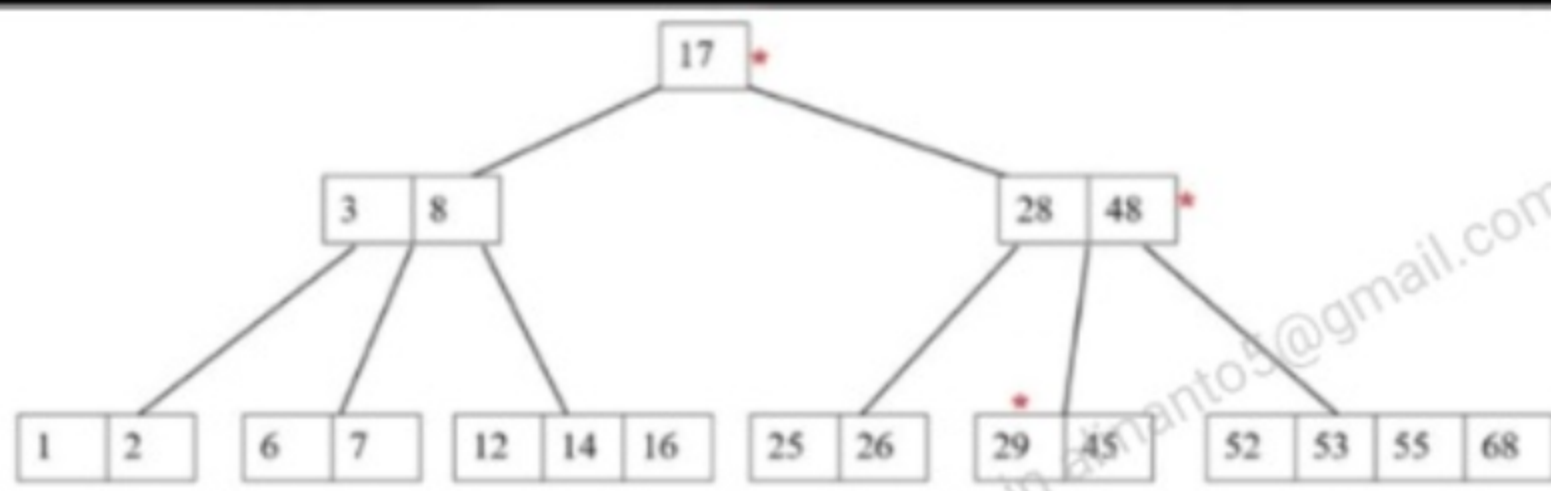
$\rightarrow$  In such questions strategy is to look at largest node  $L$  & smallest node  $S$ .

$$\left. \begin{array}{l} L = \text{\#keys in largest node} \\ S = \text{\#keys in smallest node} \end{array} \right\} \Rightarrow L+1 \leq p \leq 2(S+1)$$

here  $4 \leq p \leq 6$







$$L = 4$$

$$S = 2$$

$$L+1 \leq p \leq 2(S+1)$$

$$5 \leq p \leq 6$$

1. Each internal node in the B-tree (Figure 17.10(a)) is of the form

$$\langle P_1, \langle K_1, Pr_1 \rangle, P_2, \langle K_2, Pr_2 \rangle, \dots, \langle K_{q-1}, Pr_{q-1} \rangle, P_q \rangle$$

where  $q \leq p$ . Each  $P_i$  is a **tree pointer**—a pointer to another node in the B-tree. Each  $Pr_i$  is a **data pointer**<sup>10</sup>—a pointer to the record whose search key field value is equal to  $K_i$  (or to the data file block containing that record).

2. Within each node,  $K_1 < K_2 < \dots < K_{q-1}$ .

3. For all search key field values  $X$  in the subtree pointed at by  $P_i$  (the  $i$ th subtree, see Figure 17.10(a)), we have:

$$K_{i-1} < X < K_i \text{ for } 1 < i < q; X < K_i \text{ for } i = 1; \text{ and } K_{i-1} < X \text{ for } i = q$$

<sup>10</sup>A data pointer is either a block address or a record address; the latter is essentially a block address

4. Each node has at most  $p$  tree pointers.

5. Each node, except the root node, has at least  $\lceil p/2 \rceil$  tree pointers. The root node has at least two tree pointers unless it is the only node in the tree.

6. A node with  $q$  tree pointers,  $q \leq p$ , has  $q - 1$  search key field values (and hence has  $q - 1$  data pointers).

7. All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their **tree pointers**  $P_i$  are NULL.

### Q 14.10: (Ullman DBMS Book 2nd Ed)

Suppose our blocks are 4096 bytes. Also let keys be integers of 4 bytes and let pointers be 8 bytes. If there is no header information kept on the blocks, then **find the Order of B Tree** (where order is defined as maximum number of tree pointers per B tree node)

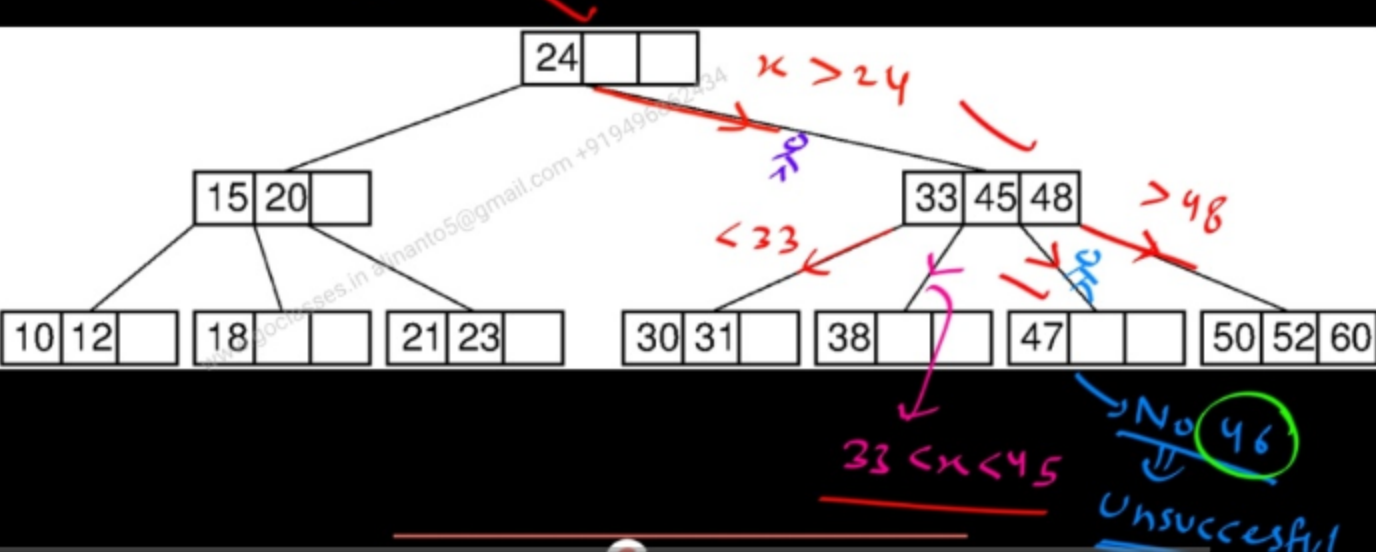
$$\text{Block} = 4096 \text{ B}$$

$$\begin{aligned} \text{in a block of order } p & \begin{cases} p \text{ tree pointers} \Rightarrow p \cdot 8 \text{ B} \\ p-1 \text{ keys} \Rightarrow (p-1) \cdot 4 \text{ B} \\ p-1 \text{ data pointers} \Rightarrow (p-1) \cdot 8 \text{ B} \end{cases} \\ \text{Total} &= (20p - 12) \text{ B} \end{aligned}$$

$$\begin{aligned} \text{Total} &\leq 4096 \\ 20p - 12 &\leq 4096 \\ 20p &\leq 4108 \\ p &\leq \frac{4108}{20} \end{aligned} \Rightarrow p = \underline{\underline{205}}$$

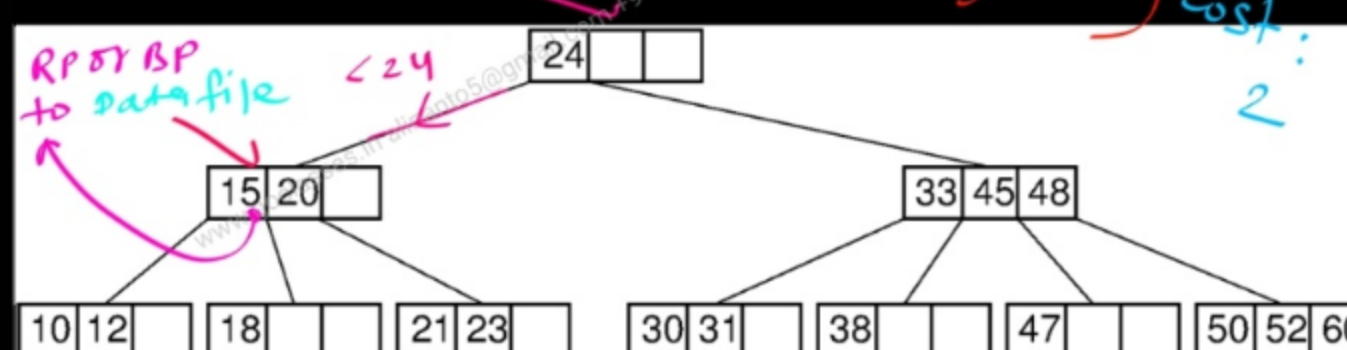
### B Tree Search algorithm

Search (46) → # Disk Block Access = 3



### Database Management System

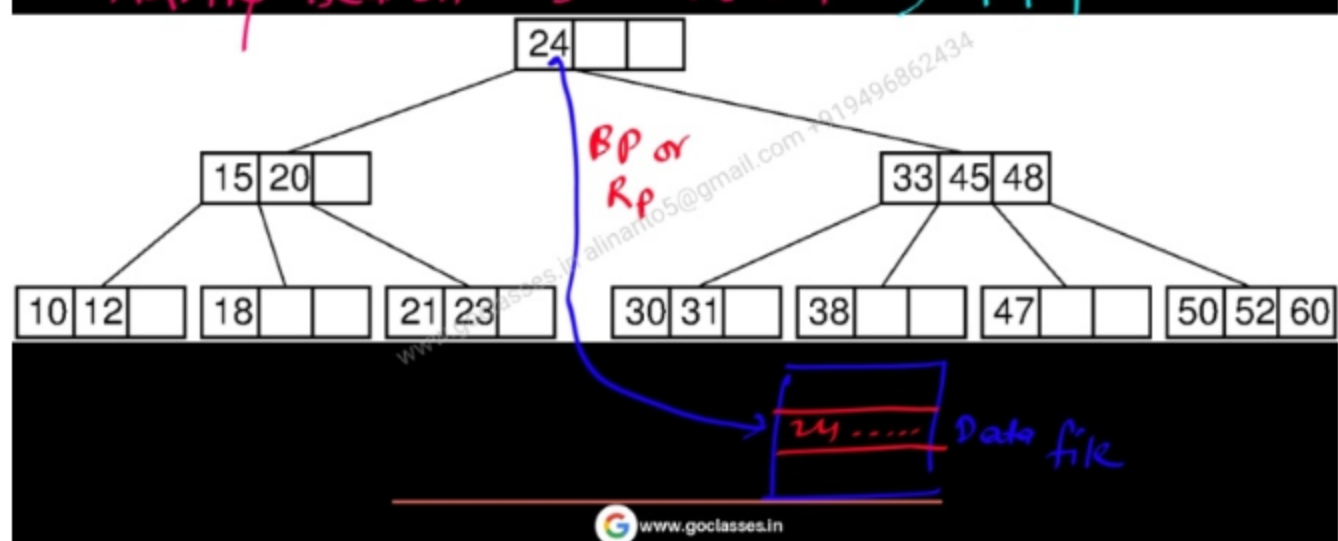
Search B Tree Index for Search key = 15



I/O cost for Data Record having Search key value = 15 ⇒ 2 + 1

### Database Management System

I/O cost to find Record (Data Record) having Search key value 24 ⇒ 1 + 1



→ Every unsuccessful search goes till last leaf

→ Best case is 1

→ Worst case is level of tree



### Searching in a B-Tree

- Doing a search in a B-tree involves
  - searching the root node for the key, and
  - if it's not found, recursively exploring the correct child.
- Using binary search within a given node, can find the key or the correct child in time  $O(\log \text{number-of-keys})$ .
- Repeat this process  $O(\text{tree-height})$  times.
- Time complexity is  $O(\log \text{number-of-keys} \cdot \text{tree-height})$

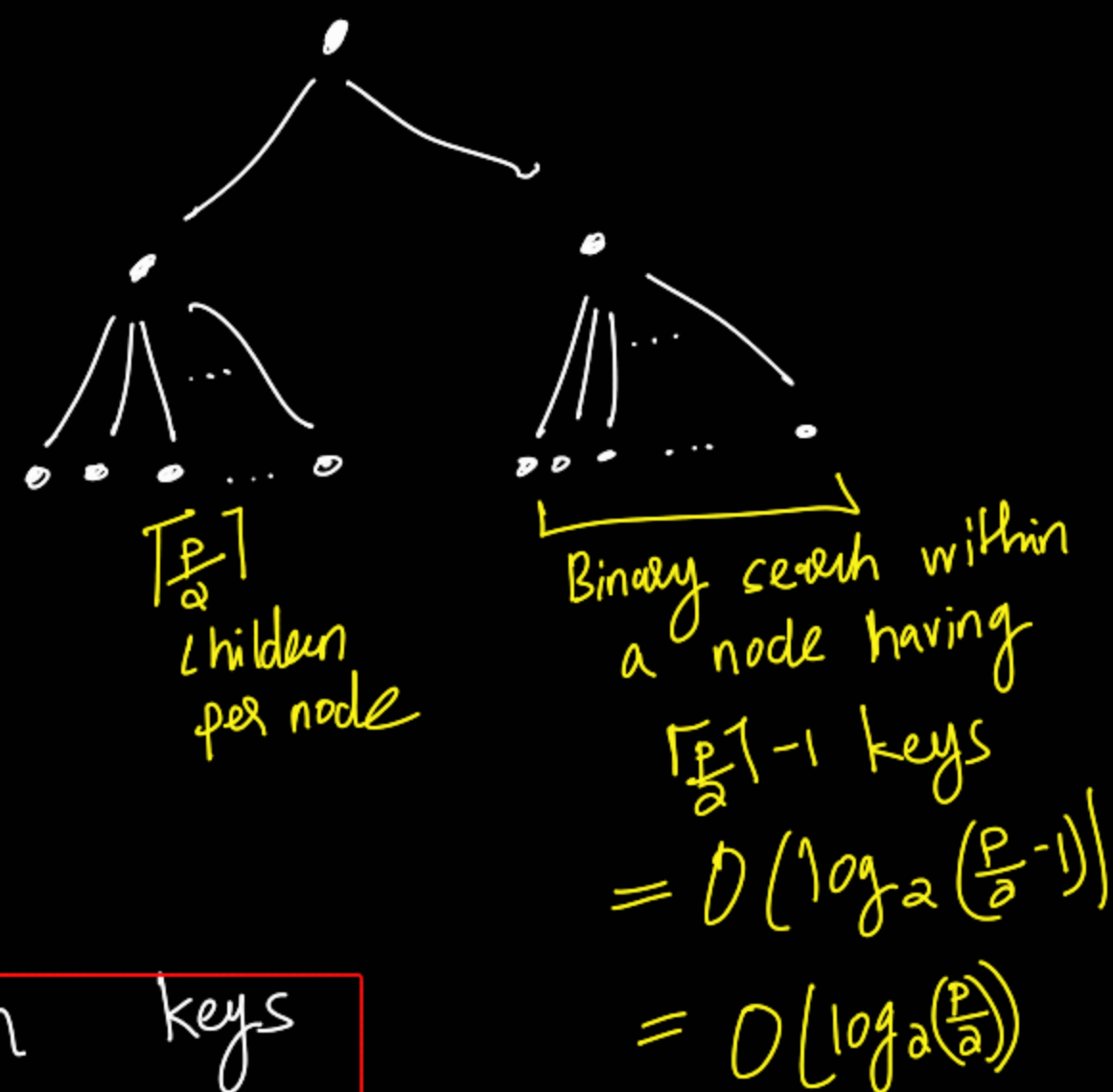
in a node  
Max = p-1

$$\begin{aligned} \text{Time complexity} &= O(\log_a(p-1) \cdot \text{height}) \\ &\approx O(\log_2(p) \cdot \text{height}) \end{aligned}$$



## Worst Case Height

- if the total no of valid search keys are  $n$ , then what is the worst case height of B tree (maximum height)
- Worst case happens when all nodes have min utilisation
  - $\Rightarrow$  all nodes have  $\lceil \frac{P}{2} \rceil - 1$  keys only.
  - $\Rightarrow$  all nodes (except root) has  $\lceil \frac{P}{2} \rceil$  children only
  - $\Rightarrow$  root node has 1 key & 2 children only
- $\Rightarrow$  Maximum height =  $\log_{P/2}(n)$



Time complexity of searching a B tree having  $n$  keys

Use given  $n$  &  $P$

$$= O\left[\underbrace{\log_{P/2}(n)}_{\text{Worst case height}} \cdot \underbrace{\log_2\left(\frac{P}{2}\right)}_{\text{Cost for within node binary search}}\right]$$



$$= O[\log_2(n)]$$

Side quest

$$\log_a b = \frac{\ln_z b}{\ln_z a}$$

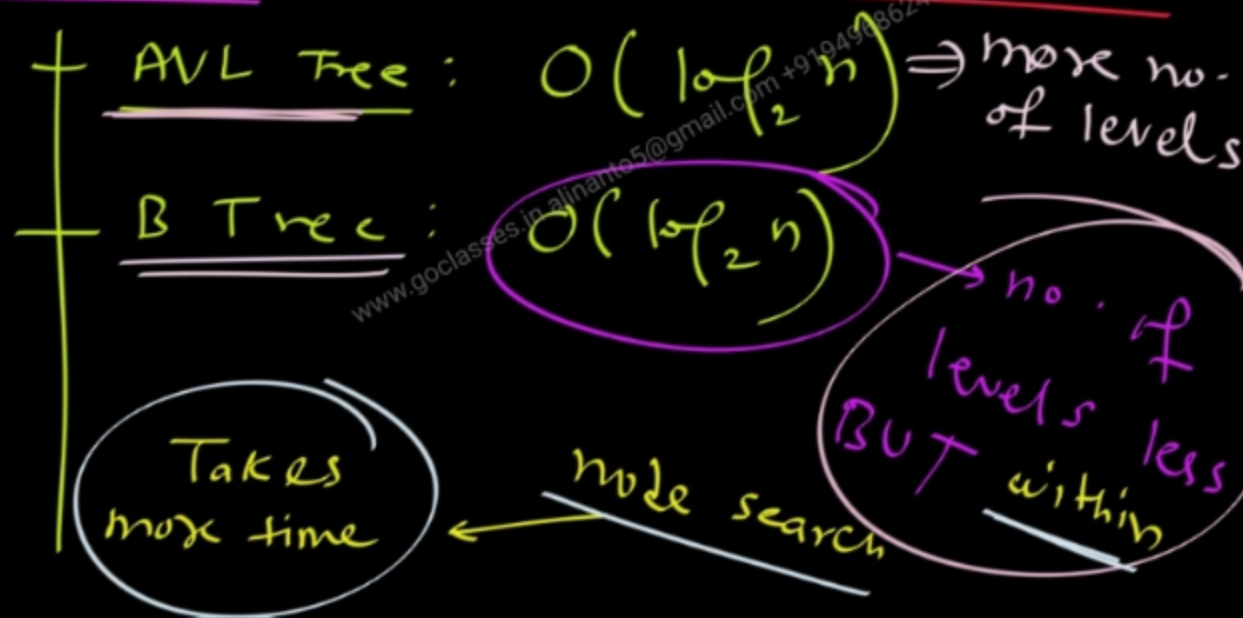
$$\Rightarrow \log_{P/2}(n) = \frac{\log_2(n)}{\log_2(P/2)}$$

$$\Rightarrow \log_{P/2}(n) \cdot \log_2(P/2) = \log_2(n)$$

## AVL vs B tree

$n$  keys:

Search time Comp:



## Insertion Algorithm

Database Management System

The algorithm for insertion takes an entry, finds the leaf node where it belongs, and inserts it there.

Insertion in B Tree **ALWAYS** happens at leaf Node.

B Tree of order 5:

Insertion Case 1:

**Always at leaf**

leaf is NOT full

then insert in sorted way.

leaf node  
2 6 8  
NOT full

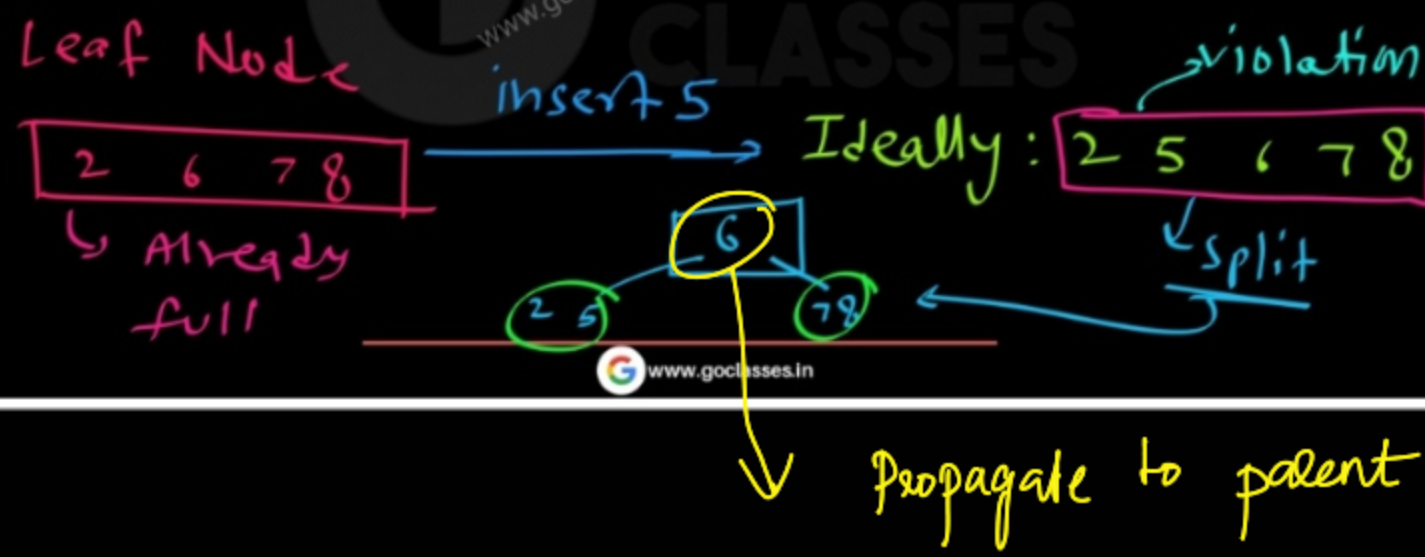
insert 7

2 6 7 8



B Tree of order 5:

Insertion 2: Leaf Node is full: **SPLIT**



## Insertion in B-Trees

### OVERFLOW CONDITION:

A root-node or a non-root node of a B-tree of order  $m$  overflows if, after a key insertion, it contains  $m$  keys. (i.e. the node, before insertion was full and had  $m-1$  keys)

### Insertion algorithm:

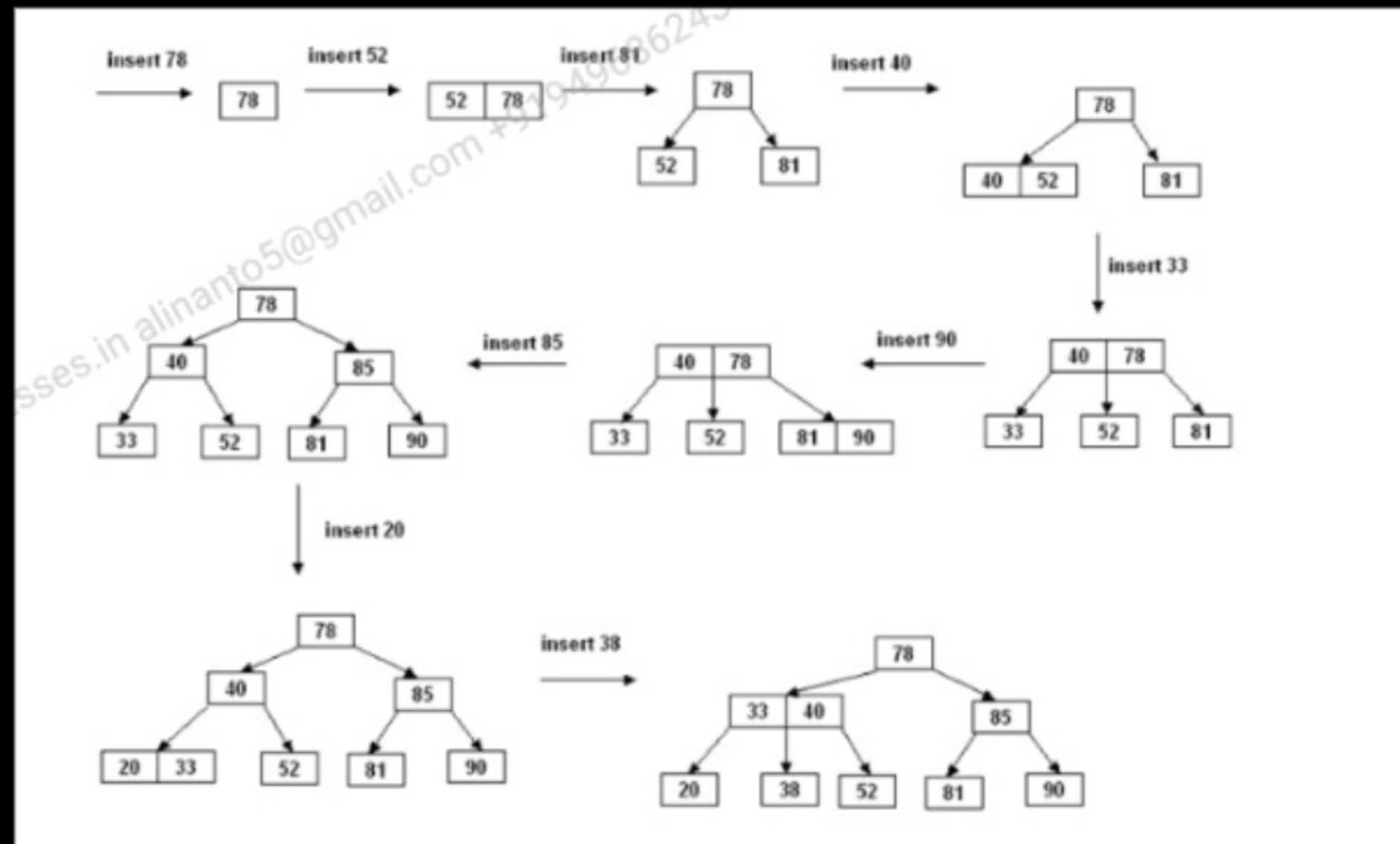
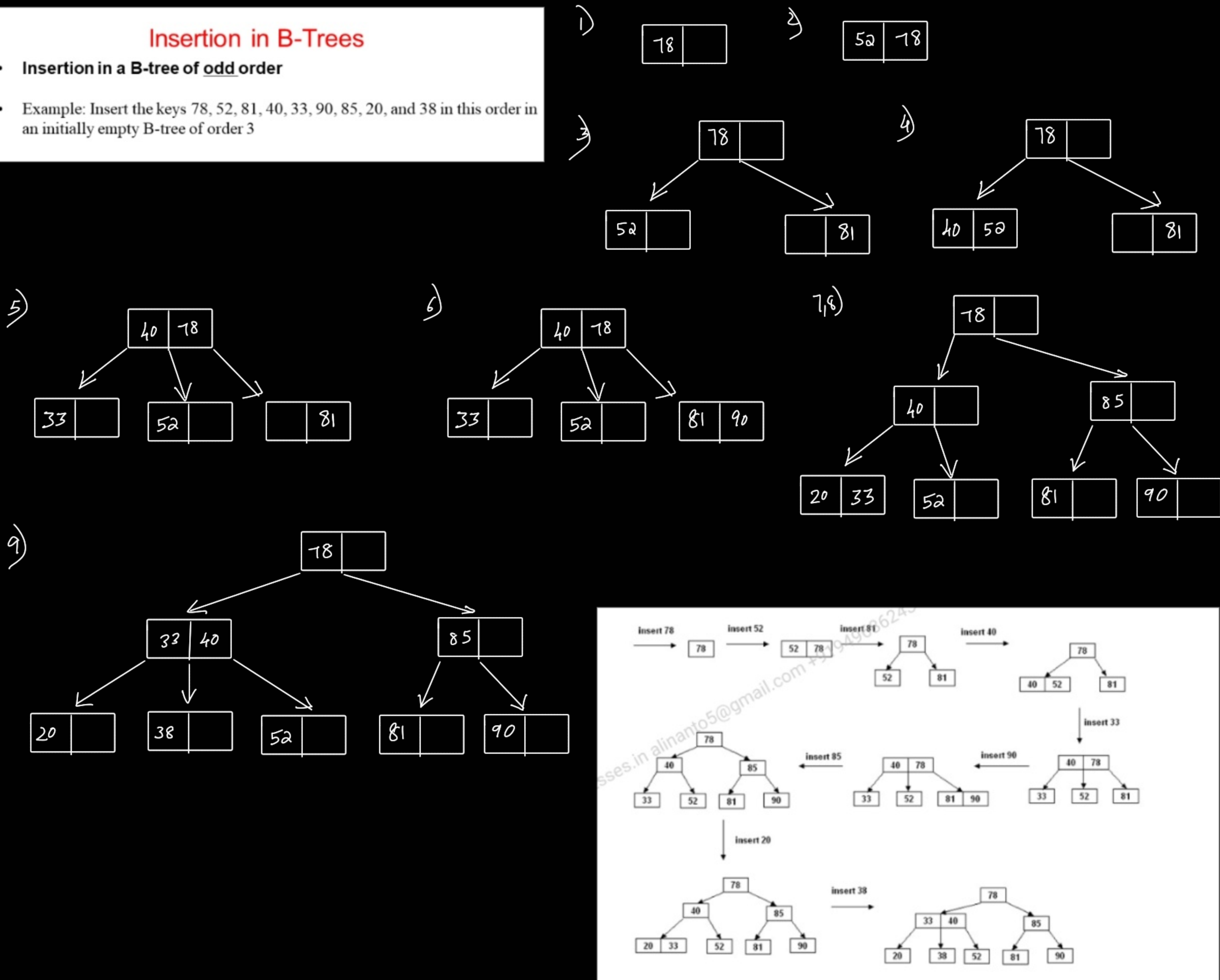
If a node overflows, split it into two, propagate the "middle" key to the parent of the node. If the parent overflows the process propagates upward. If the node has no parent, create a new root node.

Note: Insertion of a key always starts at a leaf node.

## Insertion in B-Trees

### Insertion in a B-tree of odd order

Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3

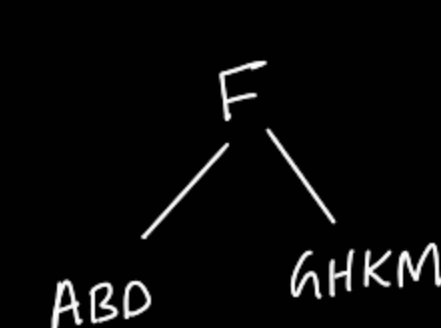
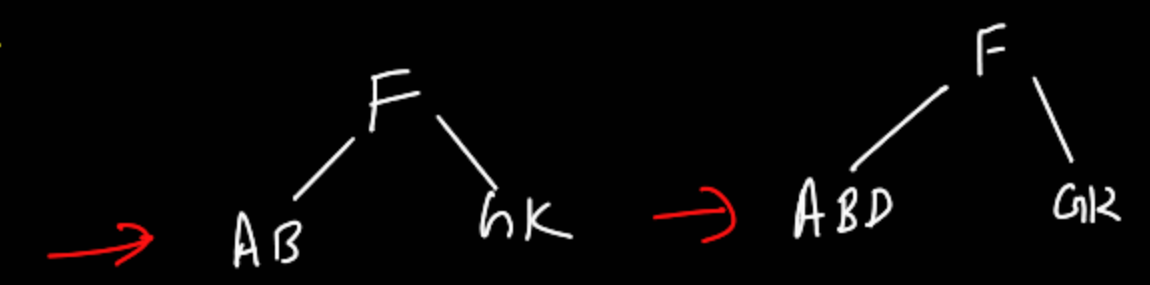
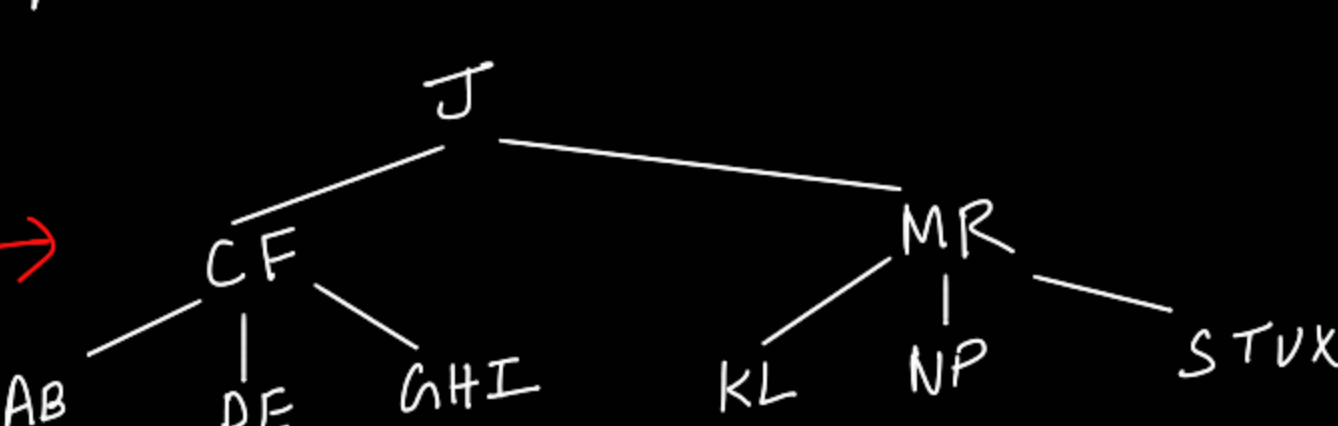
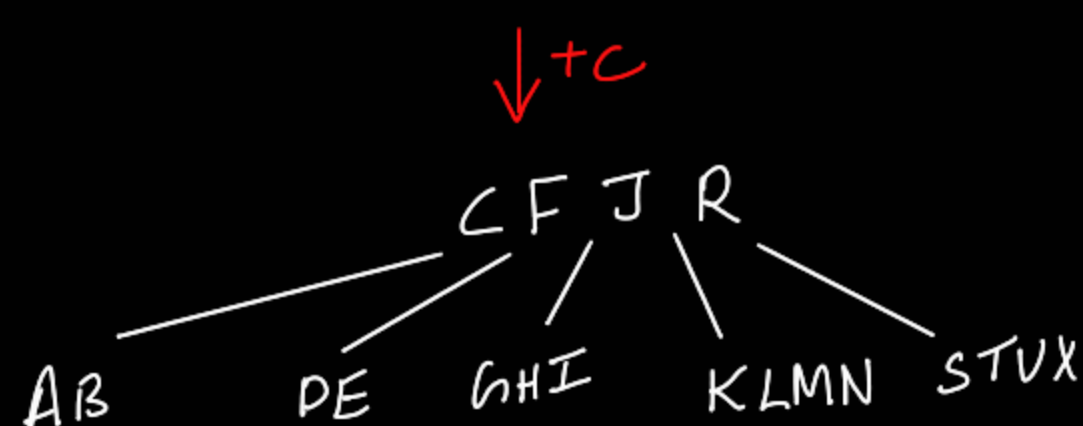


Q: Creating a B tree of order 5.

AGFBKDHMJESIRXCLNTUP

→ Don't draw box in exam

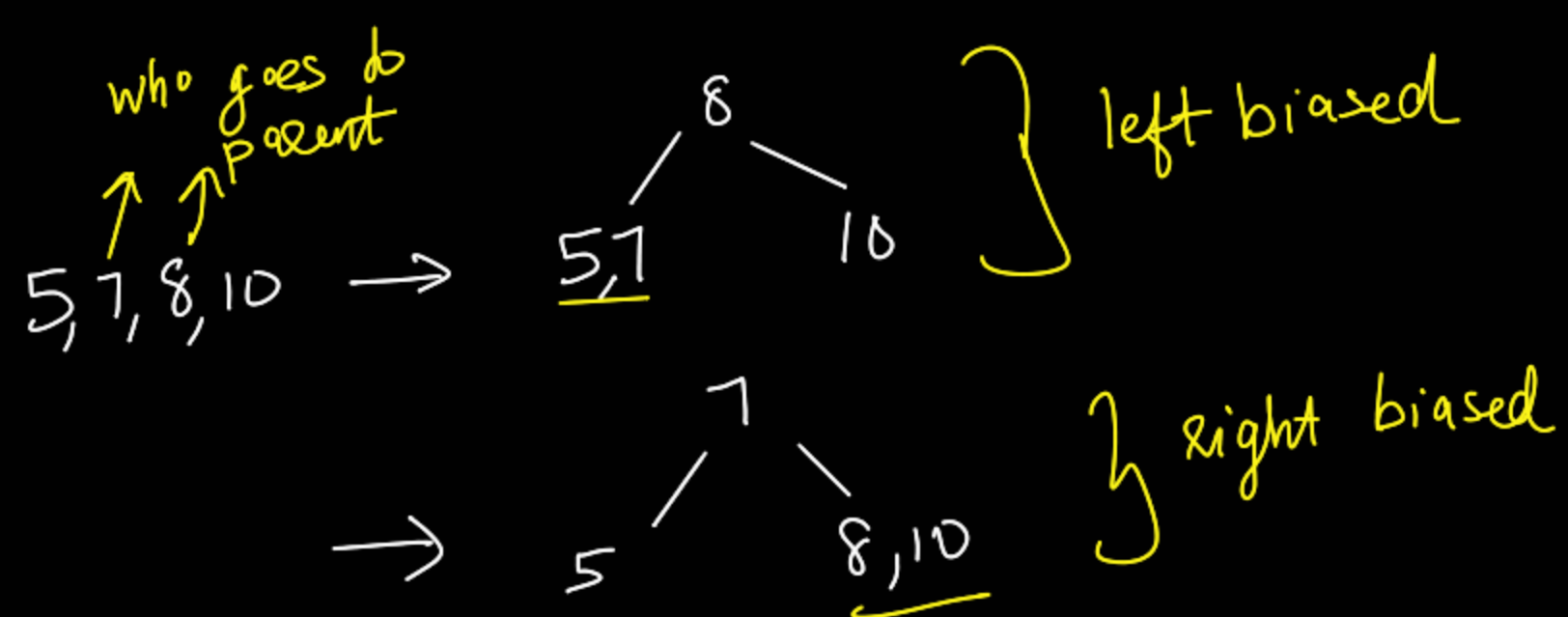
→ Prev Question for understanding & clarity.





Special case  $p = \text{even} \Rightarrow \text{Max \# keys} = p-1 = \text{odd}$

eg.  $p=4$  insert 5, 7, 10, 8



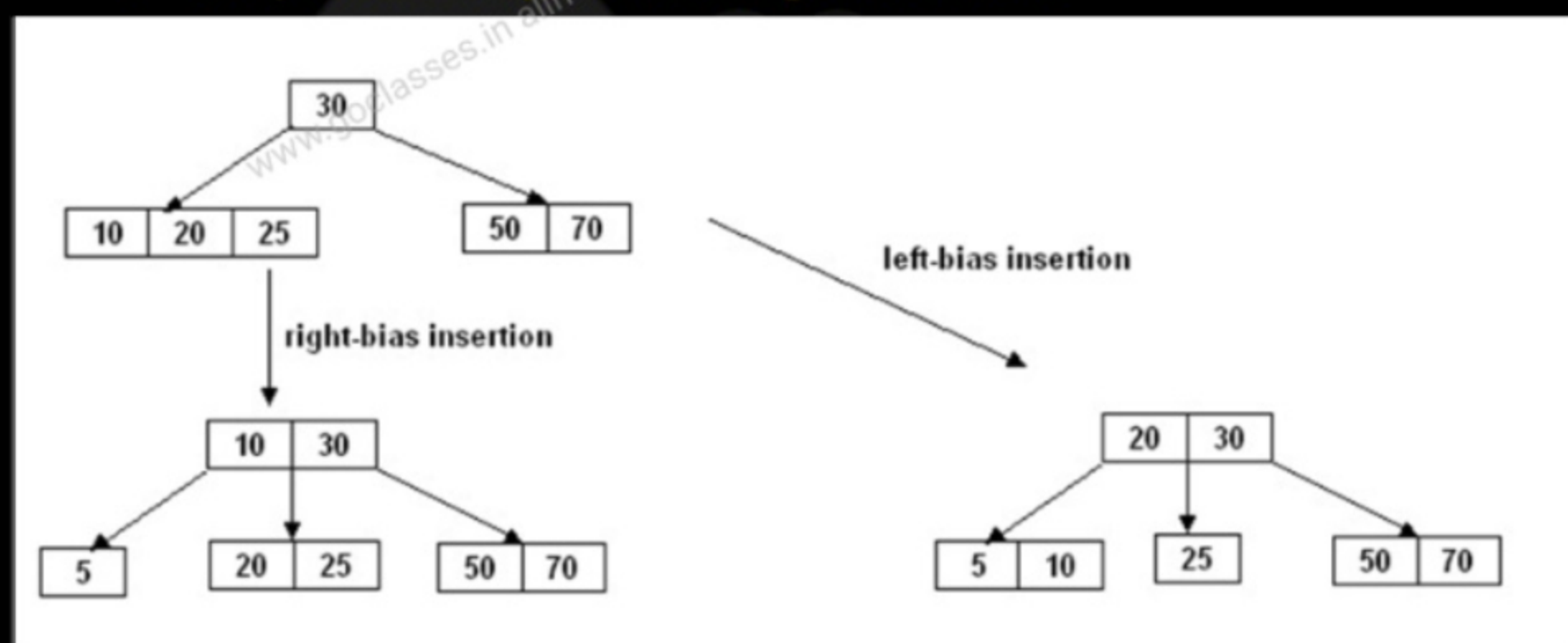
Always stick to one method consistently throughout  $\rightarrow$  Do not mix both methods

### Insertion in a B-tree of even order

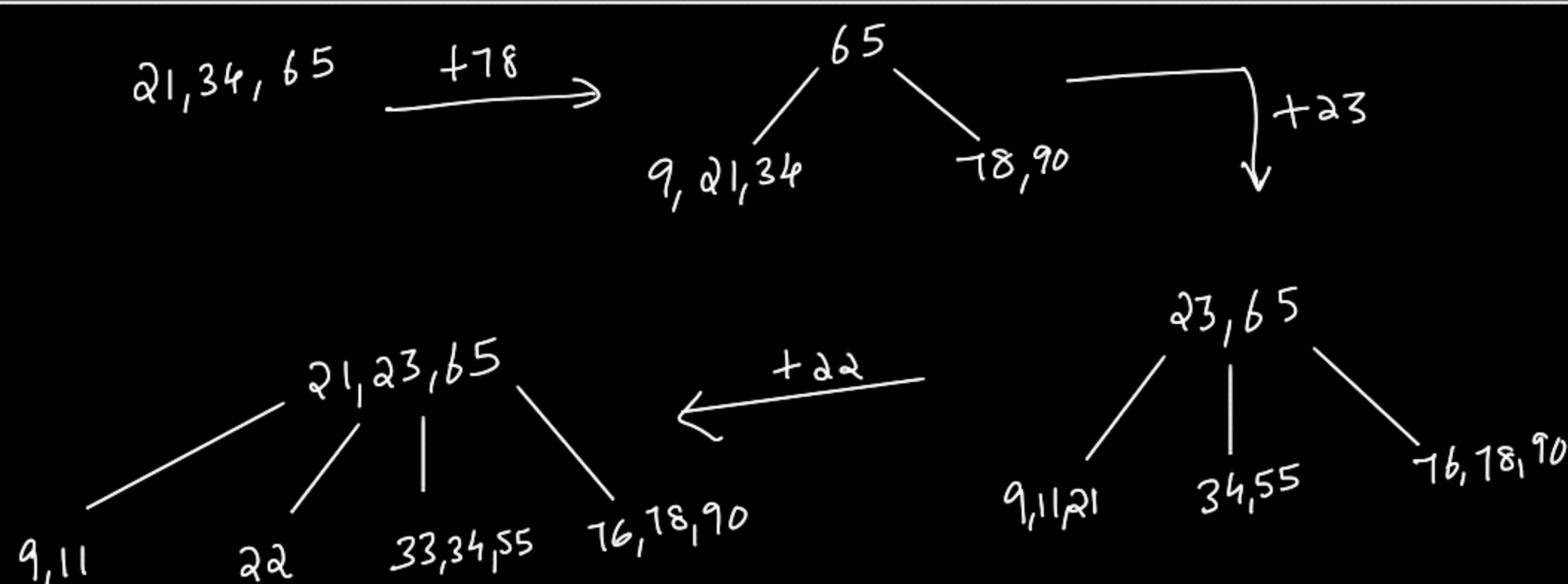
At each node the insertion can be done in two different ways:

- **right-bias:** The node is split such that its right subtree has more keys than the left subtree.
- **left-bias:** The node is split such that its left subtree has more keys than the right subtree.

• **Example:** Insert the key 5 in the following B-tree of order 4:



Define B Tree. Create a B Tree of order 4 using the keys: 21, 34, 65, 78, 90, 09, 23, 11, 55, 76, 22, 33 and make the tree a "left-biased".



**Database Management System** GO Classes

Q: Does Order in which keys are inserted in B Tree Matter?? Can we get Different B trees, of Different heights for the same set of keys, But using different order of insertion of keys in B tree?

Ans: YES Consider a B Tree of Order 3, and the following set of keys: {1, 2, 3, 4, 5, 6, 7}

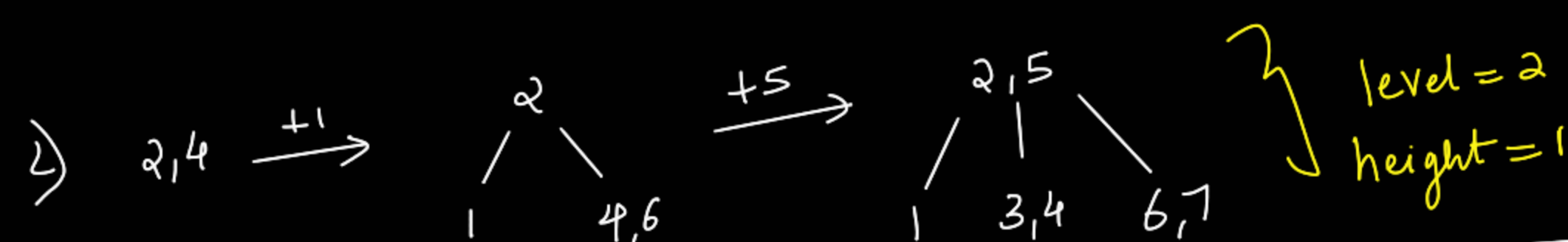
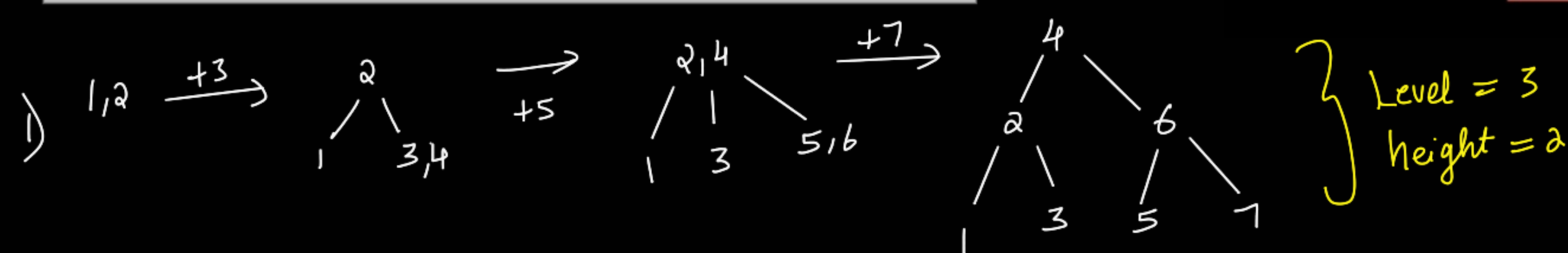
Insertion Order 1: 1, 2, 3, 4, 5, 6, 7

Insertion Order 2: 2, 4, 1, 6, 5, 3, 7

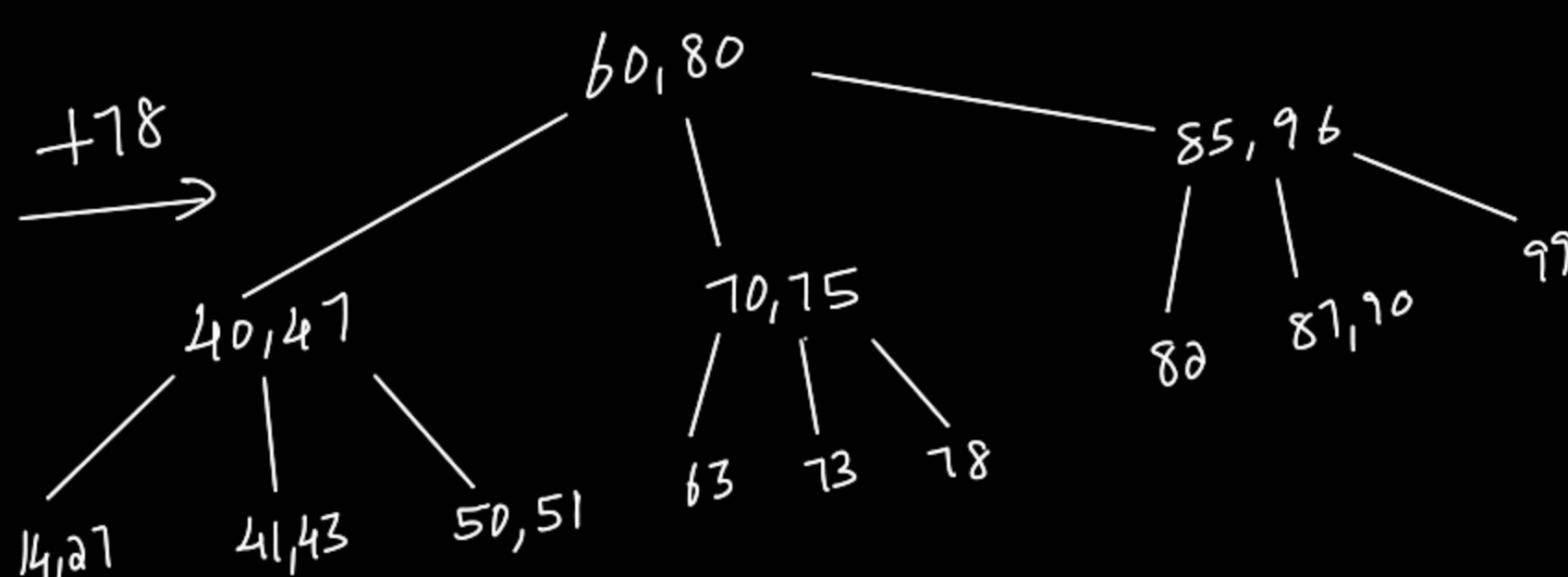
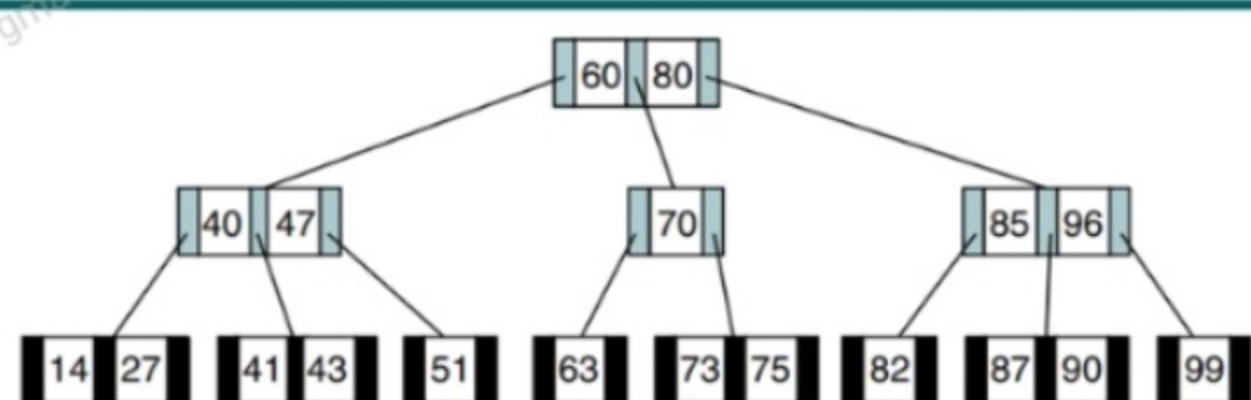
**Database Management System** GO Classes

B-Tree: Insertion  $\phi$  frequent

B+ Tree: finding order  $\phi$  are frequent

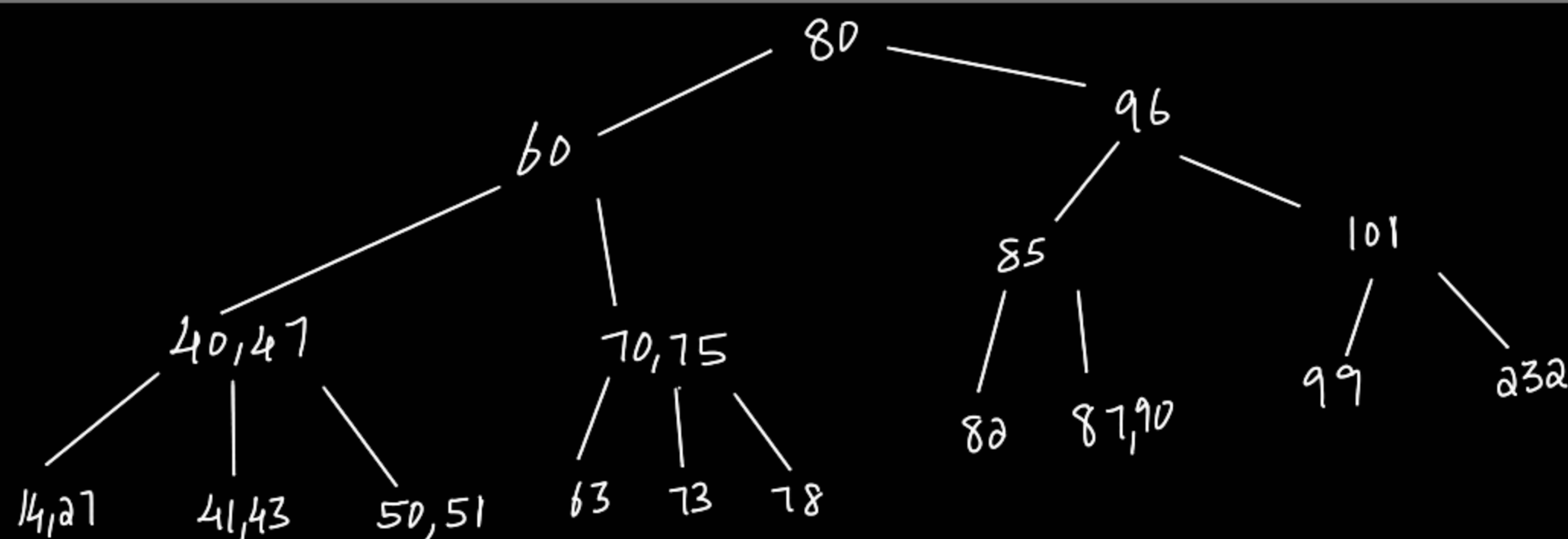


10. Using the B-tree of order 3 shown in Figure 10-24, add 50, 78, 101, and 232.





+232



A B-tree starts with a single root node (which is also a leaf node) at level 0 (zero). Once the root node is full with  $p - 1$  search key values and we attempt to insert another entry in the tree, the root node splits into two nodes at level 1. Only the middle value is kept in the root node, and the rest of the values are split evenly between the other two nodes. When a nonroot node is full and a new entry is inserted into it, that node is split into two nodes at the same level, and the middle entry is moved to the parent node along with two pointers to the new split nodes. If the parent node is full, it is also split. Splitting can propagate all the way to the root node, creating a new level if the root is split.

→ Root Node split ⇒ Level increases

$b$  is predecessor of  $a$  if  $\forall x [x < a \rightarrow x < b]$

The largest node in the tree that is smaller than a given node is its predecessor

$b$  is successor of  $a$  if  $\forall x [x > a \rightarrow x > b]$

The smallest node in the tree that is larger than a given node is its successor

#### Inorder Predecessor of a node X

- If X has a left subtree → predecessor = rightmost node in X.left
- Else → predecessor = nearest ancestor for which X lies in its right subtree

#### Inorder Successor of a node X

- If X has a right subtree → successor = leftmost node in X.right
- Else → successor = nearest ancestor for which X lies in its left subtree

Q T/F → The immediate predecessor of every key of a nonleaf node is always in a leaf node. **True.**

Case 1 →

49, 50, 55

40, 49

50, 55

always stays in leaf node even if gets split

Case 2 →

50, 72

31, 49

always stays in leaf node even if gets split

Contra positive

$$\forall x [\text{leaf}(x) \rightarrow \text{leaf}(x_{\text{predecessor}})]$$

$$\equiv \forall x [\text{leaf}(x) \rightarrow \text{leaf}(x_{\text{successor}})]$$

$$\neq \forall x [\text{leaf}(x) \rightarrow \text{leaf}(x_{\text{predecessor}})]$$

$$\neq \forall x [\text{leaf}(x) \rightarrow \text{leaf}(x_{\text{successor}})]$$

• if the immediate predecessor to a node  $x$  is not a leaf node then  $x$  is a leaf node } ⇒ For a non-leaf node, the immediate successor is always a leaf

Q T/F) The immediate predecessor of every key of a nonleaf node is always in a leaf node } **True**

The immediate successor of every key of a nonleaf node is always in a leaf node } **True**

The immediate predecessor of every key of a leaf node is always in a nonleaf node } **False**

The immediate successor of every key of a leaf node is always in a nonleaf node } **False**

# Leaf Nodes & # Keys in Non leaf Node

From above result → For every Non Leaf Key there is a leaf immediate predecessor

→ Immediate Predecessors will be unique

→ No two keys can have same immediate predecessor

→ This predecessor will be always be right most value in its node



→ This implies that every non leaf key has a corresponding unique leaf node in which its immediate predecessor resides as the right most value.

→ # Non leaf Keys + 1 = # leaf nodes  
 ↳ Corresponds to the right most leaf node which does not correspond to any non leaf key

$$\# \text{Keys in Non-leaf Nodes} + 1 = \# \text{Leaf nodes}$$

### Conclusion/Results:

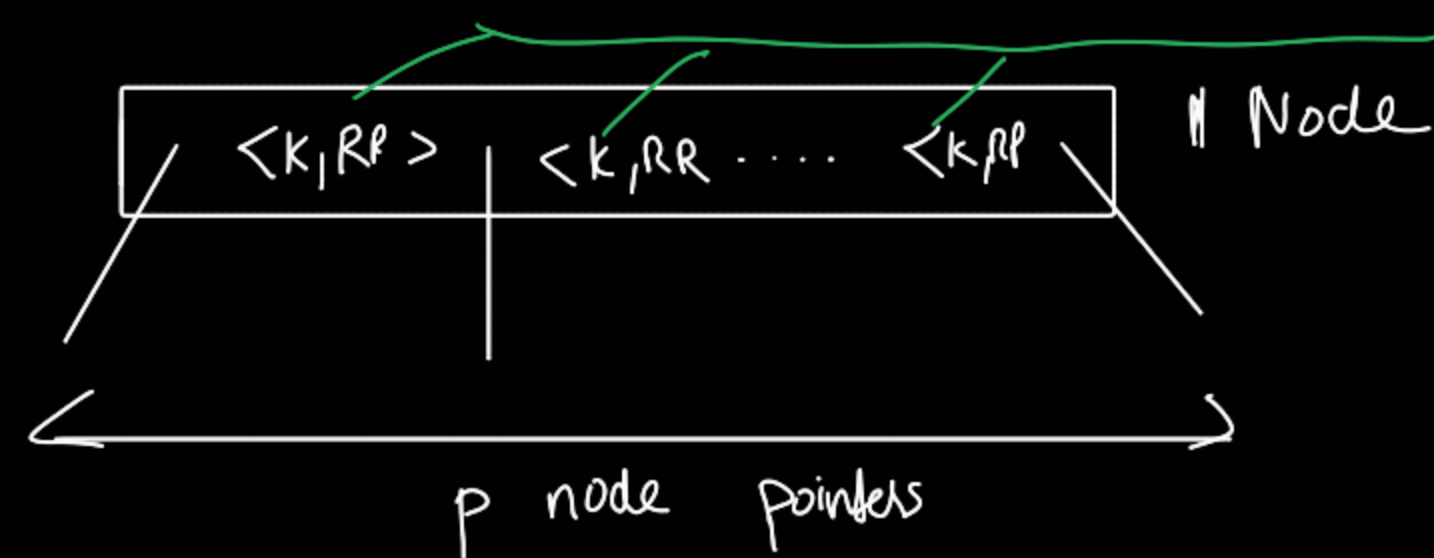
1. Immediate successor of every key in non leaf is always in a leaf.
2. Immediate predecessor of every key in non leaf is always in a leaf.

3. For a B Tree of at least two levels:

$$\text{Number of Leaf Nodes} = 1 + \text{Number of Keys in Non-leaf nodes}$$

### Order of B Tree is p:

Each B-tree node can have at most p tree/Block/Node pointers, p - 1 data(Record or Block) pointers, and p - 1 search key field values.



→ Every Node is 1 block

$$\Rightarrow p(\text{BP size}) + p-1 \cdot (\text{Search key size} + \text{Datapointer}) \leq \text{Block size}$$

↳ can be ① Record pointer  
② Block pointer

EXAMPLE 4: Suppose the search field is V = 9 bytes long, the disk block size is B = 512 bytes, a record (data) pointer is Pr = 7 bytes, and a block pointer is P = 6 bytes. Each B-tree node can have at most p tree pointers, p - 1 data pointers, and p - 1 search key field values. These must fit into a single disk block if each B-tree node is to correspond to a disk block.

find p?

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

$$6p + 16p - 16 \leq 512$$

$$22p \leq 528$$

$$p = \left\lfloor \frac{528}{22} \right\rfloor = 24$$

\$ → size of

General formula

$$p \cdot \$BP + p-1(\$SK + \$DP) \leq \$Block$$

$$p[\$BP + \$SK + \$DP] \leq \$Block + \$SK + \$DP$$

$$p = \left\lfloor \frac{\text{Block size} + \text{Search key size} + \text{Datapointer}}{\text{Block pointer} + \text{Search key} + \text{Datapointer}} \right\rfloor$$

If Some other Definition of ORDER of a B Tree Node is given then somehow get back to OUR default definition of Order:

Eg: Order of B Tree is p:

Each B-tree node can have at most p keys.

find p?

$$p(K+DP) + (p+1)BP \leq \text{Block size}$$

→ Got lemon

→ Made Lemonade



Eg: Order of B Tree is (p):

Each B-tree node, except root can have at least p, and at most 2p tree pointers. If tree is non-empty, root can have minimum 2 pointers.

$$\Rightarrow 2p \cdot BP + (2p-1)(SK+DP) \leq \text{Block size}$$

find p?

The order p of B tree node is defined as:

Each B-tree node can have at most p keys.

Given that the block size is 1K bytes, data record pointer is 7 bytes long, the value field is 9 bytes long and a block pointer is 6 bytes long, what is the order of the B tree node?

$$p(7+9) + (p+1)6 \leq 1024$$

$$22p \leq 1024 - 6$$

$$p = \left\lfloor \frac{1018}{22} \right\rfloor = 46$$

138

→ Remember → if nothing is given as order definition then  $p = \text{Max children per node}$

The order of a leaf node in a B tree is the maximum number of (value, data record pointer) pairs it can hold. Given that the block size is 1K bytes, data record pointer is 7 bytes long, the value field is 9 bytes long and a block pointer is 6 bytes long, what is the order of the leaf node? find p?

$$p(7+9) + (p+1)6 = 1024$$

$$p = \left\lfloor \frac{1018}{22} \right\rfloor = 46$$

→ Space of BP has to be still allocated even if NULL  
→ Since a leaf node can split anytime  
→ All nodes have same structure

3.1.24 B Tree: GATE IT 2004 | Question: 79

Consider a table T in a relational database with a key field K. A B-tree of order p is used as an access structure on K, where p denotes the maximum number of tree pointers in a B-tree index node. Assume that K is 10 bytes long; disk block size is 512 bytes; each data pointer P<sub>D</sub> is 8 bytes long and each block pointer P<sub>B</sub> is 5 bytes long. In order for each B-tree node to fit in a single disk block, the maximum value of p is

- A. 20
- B. 22
- C. 23
- D. 32

$$p \cdot 5 + (p-1)(10+8) \leq 512$$

$$23p \leq 530$$

$$p = \left\lfloor \frac{530}{23} \right\rfloor = 23$$

NOTE that in B tree, ALL the nodes have the SAME structure...

In the leaf nodes, ALL tree pointers are NULL  
(But tree pointers are present in the leaf node, even though they all are NULL).

GO CLASSES

Database Management System

GO Classes

The Variation:

Q: Since leaf nodes require no pointers to children, they could conceivably store a different (larger) number of keys than internal nodes for the same disk page size.

Considering this Variation of B tree, Assume order of a leaf node of B tree is the maximum number of keys that can be stored in the leaf node.

Given that the block size is 1K bytes, data record pointer is 7 bytes long, the value field is 9 bytes long and a block pointer is 6 bytes long, what is the order of the leaf node of B tree?

$$p(7+9) \leq 1024$$

$$p = \left\lfloor \frac{1024}{16} \right\rfloor = 64$$

Problem 2: B-tree (20 points)

Consider a DBMS that has the following characteristics:

- 2KB fixed-size blocks
- 12-byte pointers
- 56-byte block headers

We want to build an index on a search key that is 8 bytes long. Calculate the maximum number of records we can index with

a 3-level B tree

$$12p + p-1(12+8) \leq 2048-56$$

$$32p \leq 2048-56+20$$

$$p = \left\lfloor \frac{2014}{32} \right\rfloor = 62$$

$$\# \text{ keys in tree} = p^2 - 1 = 62^2 - 1 = 238327 \text{ keys}$$

$$\begin{array}{r} 62 \\ \times 62 \\ \hline 124 \\ 372 \times \\ \hline 3844 \\ 62 \\ \hline 7688 \\ 23064 \times \\ \hline 238328 \end{array}$$



Max #keys in a tree of degree  $p$  & level  $n$  / depth  $n-1$

$$\#Nodes = 1 + p + p^2 + \dots + p^{n-1} = \frac{p^n - 1}{p - 1}$$

each node can have max of  $p-1$  keys

$$\Rightarrow \text{Max \#keys} = (p-1) \cdot \frac{p^n - 1}{p - 1} = p^n - 1$$

→ Same above question, Min keys?



$$\begin{aligned} &= 1 + 30 \times 2 + 30 \times 2 \times 31 \\ &= 1 + 60 + 1860 \\ &= 1921 \end{aligned}$$

62 nodes

$$\begin{array}{r} 930 \times 31 \\ 2790 \times \\ 28830 + \\ 931 \\ \hline 29761 \end{array}$$

## Database Management System

### The Variation:

Q: Since leaf nodes require no pointers to children, they could conceivably store a different (larger) number of keys than internal nodes for the same disk page size.

Considering this Variation of B tree, What will be the answer??

Consider a DBMS that has the following characteristics:

- 2KB fixed-size blocks
- 12-byte pointers
- 56-byte block headers

max no. of BP per internal node = 62

" " of keys per leaf node = 99

We want to build an index on a search key that is 8 bytes long. Calculate the maximum number of records we can index with

max " " key per internal node = 61

3 level tree

$$\begin{aligned} \text{Max \#keys} &= 1 \times 61 + 62 \times 61 + 62 \times 62 \times 99 \\ &= 61 + 3782 + 386556 \\ &= 384399 \end{aligned}$$

$p$  = degree of non leaf node

$$12p + p - 1 (12+8) \leq 2048 - 56$$

$$32p \leq 2014$$

$$p = \left\lfloor \frac{2014}{32} \right\rfloor = 62$$

$\Rightarrow$  61 keys in non leaf nodes

$$2(12+8) \leq 2048 - 56$$

$$2 = \left\lfloor \frac{1994}{20} \right\rfloor = 99 \text{ keys in leaf node}$$

$$\begin{array}{r} 62 \times 62 \\ 124 \times \\ 372 \times \\ 384 \times \\ 99 \\ 34596 \\ 34596 \times \\ 380556 \end{array}$$

Q1: Height  $h$ ; max #keys in B Tree?

$$= (p-1) + (p-1)p + (p-1)p^2 + \dots + (p-1)p^h$$

$$= (p-1) [1 + p + p^2 + \dots + p^h]$$

$$= (p-1) \left[ \frac{p^{h+1} - 1}{p - 1} \right] = \frac{p^{h+1} - 1}{p - 1}$$

Q2: Height  $h$ ; max #Nodes in B Tree?

$$= 1 + p + p^2 + \dots + p^h$$

$$= \frac{p^{h+1} - 1}{p - 1}$$



Given height  $h$  & order  $p \rightarrow$  min #keys?

$$= 1 + \left( \left\lceil \frac{p}{2} \right\rceil - 1 \right) \cdot 2 + \left( \left\lceil \frac{p}{2} \right\rceil - 1 \right) \cdot \left\lceil \frac{p}{2} \right\rceil \cdot 2 + \dots$$

$$= 1 + (t-1) \cdot 2 + (t-1) \cdot 2t + (t-1) \cdot 2t^2 + \dots + (t-1) \cdot 2t^{h-1}$$

$$= 1 + (t-1) [2 + 2t + 2t^2 + \dots + 2t^{h-1}]$$

$$= 1 + t-1 \left[ \frac{2(t^h - 1)}{t - 1} \right] = 1 + 2(t^h - 1)$$

Min #keys given height  $h$ , order  $p$

$$t = \left\lceil \frac{p}{2} \right\rceil$$

$t-1 \rightarrow$  min keys per node

let  $\left\lceil \frac{p}{2} \right\rceil = t$  ( $> 50\%$  utilisation of #children)

| Height | Order | min #keys?                                                                            |
|--------|-------|---------------------------------------------------------------------------------------|
| $L_0$  | Root  | #nodes: 1, #Bp: 2, #keys: 1                                                           |
| $L_1$  |       | #nodes: 2, #Bp: $2+t$ , #keys: $2 \times (t-1)$                                       |
| $L_2$  |       | #nodes: $2+t$ , #Bp: $2t+t$ , #keys: $2t \times (t-1)$                                |
| $L_3$  |       | #nodes: $t^2$ , #Bp: $2t^2+t$ , #keys: $2t^2 \times (t-1)$                            |
| $L_h$  |       | #nodes: $2 \cdot t^{h-1}$ , #Bp: $2t^{h-1} \times t$ , #keys: $2t^{h-1} \times (t-1)$ |



Given height  $h$ , order  $p \rightarrow$  Min # Nodes

let  $\lceil \frac{p}{2} \rceil = t$

$$= 1 + 2 + 2 \times t + 2t^2 + \dots + 2t^{h-1}$$

$$= \frac{1 + 2(t^h - 1)}{t - 1}$$

$\rightarrow$  Rather than remembering the formula, remember the idea of expansion

$\rightarrow$  Questions of type  $\rightarrow$  Given # Search Keys  $\rightarrow$  Max & Min Height of Tree

① Maximum height for  $n$  keys & order  $p$

$\rightarrow 1 + 2(t^h - 1) = \text{Min \#keys} \leq n$

where  $t = \lceil \frac{p}{2} \rceil$

$\Rightarrow t^h \leq \frac{n-1}{2} + 1$

$\downarrow$   
#keys should be atleast  
= Min #keys possible

Max height  $\Rightarrow h = \left\lceil \log_t \left( \frac{n+1}{2} \right) \right\rceil$

$\rightarrow$  ie if value is 48.3 then 48 ✓  
49 ✗

② Minimum height for  $n$  keys & order  $p$

$\rightarrow p^{h+1} - 1 \rightarrow \text{Max \#keys} \geq n$

$\Rightarrow p^{h+1} \geq n+1$

Min Height  $\Rightarrow h = \left\lceil \log_p (n+1) \right\rceil - 1$

Let  $h$  be the height of the classic B-tree  
Let  $n \geq 0$  be the number of entries in the tree. Let  $m$  be the maximum number of children a node can have. Each node can have at most  $m-1$  keys.  
It can be shown (by induction for example) that a B-tree of height  $h$  with all its nodes completely filled has  $n = m^{h+1} - 1$  entries. Hence, the best case height (i.e. the minimum height) of a B-tree is:  
$$h_{\min} = \lceil \log_m (n+1) \rceil - 1$$
  
Let  $d$  be the minimum number of children an internal (non-root) node must have. For an ordinary B-tree,  $d = \lceil m/2 \rceil$ .  
Cormen (1979) and Cormen et al. (2001) give the worst case height (the maximum height) of a B-tree as:  
$$h_{\max} = \left\lceil \log_d \frac{n+1}{2} \right\rceil$$

Consider a B-tree with Order  $P$ , where Order  $P$  is defined as usual (maximum number of child pointers a node can have).

The maximum levels of index required to store 400 distinct keys in order  $P=5$  B-tree index  $h+1=5$ ?

#levels = height + 1

$h_{\max} = \left\lceil \log_t \left( \frac{n+1}{2} \right) \right\rceil = \left\lceil \log_3 (200.5) \right\rceil = 4$

$t = \lceil \frac{5}{2} \rceil = 3$

$h_{\min} = \left\lceil \log_p (n+1) \right\rceil - 1$

Solving conceptually  
Fill minimally

| Level | height | #nodes    | #keys             |
|-------|--------|-----------|-------------------|
| 1     | 0      | 1         | 1x1 = 1           |
| 2     | 1      | 2         | 2x2 = 4           |
| 3     | 2      | 2x3       | 2x2x3 = 12        |
| 4     | 3      | 2x3x3     | 2x2x3x3 = 36      |
| 5     | 4      | 2x3x3x3   | 2x2x3x3x3 = 108   |
| 6     | 5      | 2x3x3x3x3 | 2x2x3x3x3x3 = 324 |

For level upto 5  
i need to store min 161 keys  
For level upto 6  
i need to store min 485 keys  
 $\Rightarrow$  Level 5 is max



**Example 5.** Suppose that the search field is a nonordering key field, and we construct a B-tree on this field with  $p = 23$ . Assume that each node of the B-tree is 69% full. (at least 69% full)

Find the number of keys stored in 4 Levels (Height 3)?

B-Tree: 

$$\# \text{children} = \left\lceil \frac{23 \times 69}{100} \right\rceil = 16$$

⇒ 15 keys per node

| Level | height | #Nodes   | #Keys       |
|-------|--------|----------|-------------|
| 1     | 0      | 1        | 15x1        |
| 2     | 1      | 16       | 15x16       |
| 3     | 2      | 16x16    | 15x16x16    |
| 4     | 3      | 16x16x16 | 15x16x16x16 |

$$15(1 + 16 + 16^2 + 16^3) = 15 \left( \frac{16^4 - 1}{16 - 1} \right) = 16^4 - 1 = 65535$$

Which of the following items is FALSE about B-trees?

- ☐ A binary search can be used to search within a node in a B-tree.
- ☐ A B-tree is a multi-way tree with many data items per node.
- ☐ A B-tree is a binary tree.
- ☐ A B-tree can be used for external storage.

True

True

False

True



# B+ Tree

There are two differences between the B-tree and the B+tree:

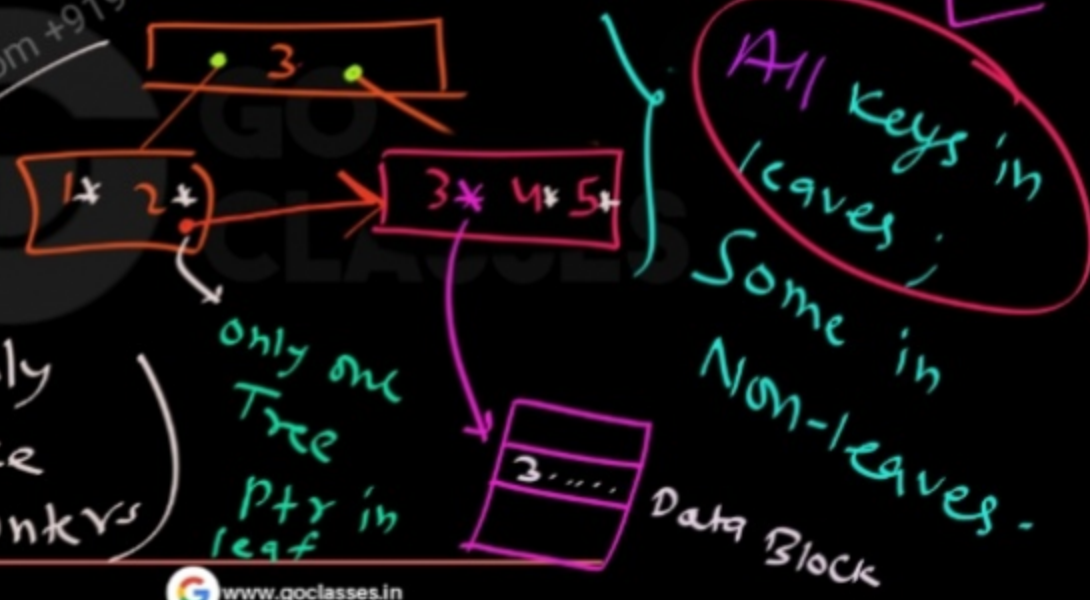
1. Each data entry must be represented at the leaf level, even though there may be internal nodes with the same keys. Because the internal nodes are used only for searching, they generally do not contain data.
2. Each leaf node has one additional pointer, which is used to move to the next leaf node in sequence. This structure is shown in Figure 10-21.

Search keys:

1, 2, 3, 4, 5

order: 5

B+Tree:

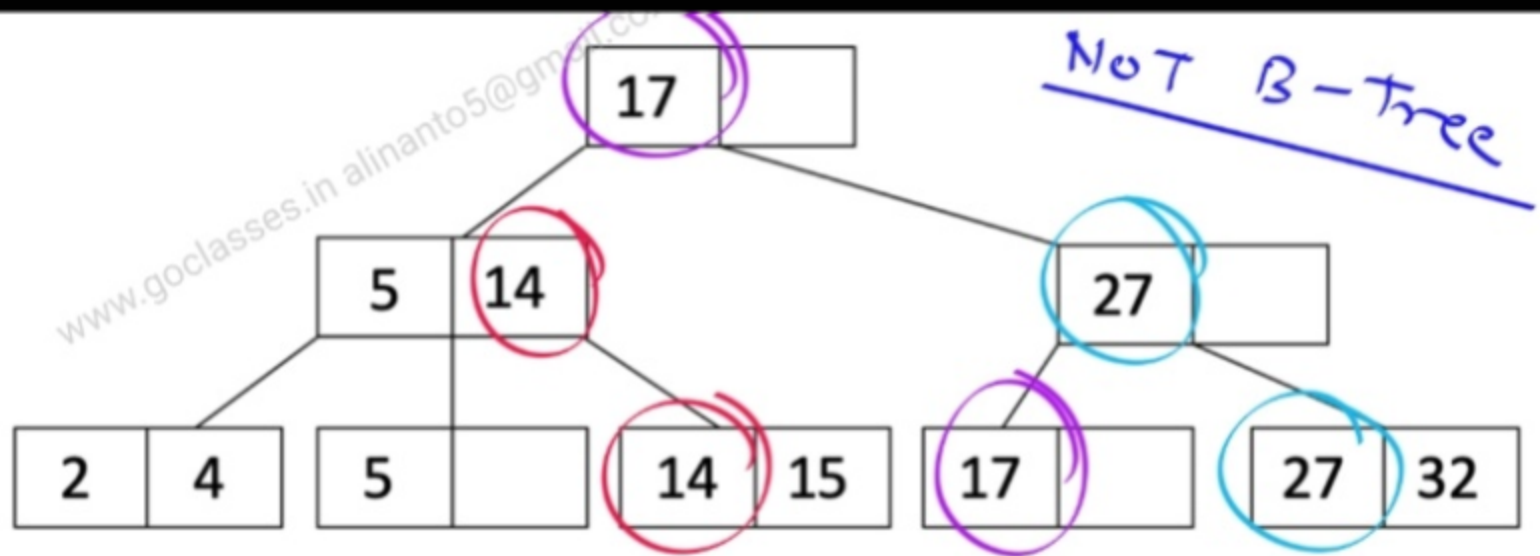


Differences of B Tree & B+ Tree:

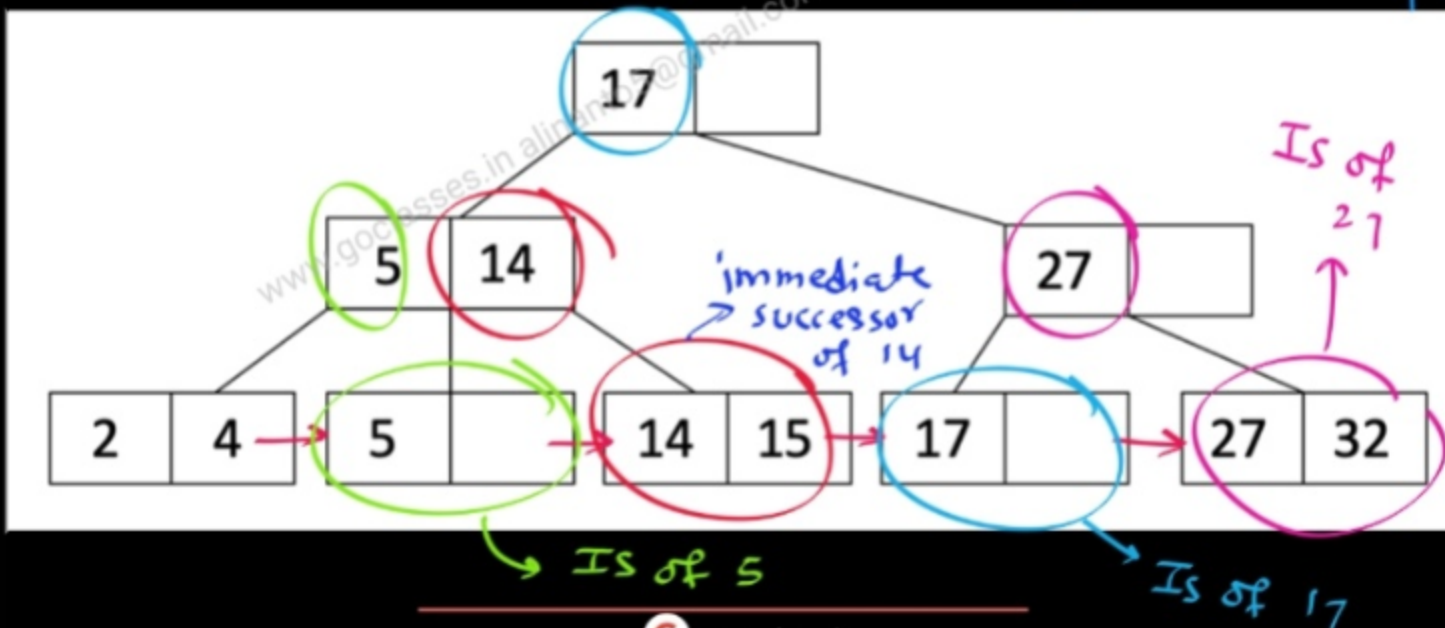
- ③ In Non-leaf Nodes, No Duplication happens.
- ④ Every leaf Node Points to Next Leaf Node (on Right)

Differences of B Tree & B+ Tree:

- ⑤ All Data pointers  $\Rightarrow$  in Leaves  
No " " in Non-leaf.  
Only Tree pointers in Non-leaf.  
One " " in every leaf to Next leaf



All values in Leaves. Some values in Non-leaf.



$\rightarrow \# \text{ Non leaf keys } + 1 = \# \text{ Leaf Node} \rightarrow \text{True here also}$

Order of B+ Tree

Maximum number of Total Pointers per Node

Order of B Tree

Max number of Node Points per Node

Rules for making a valid B+ tree

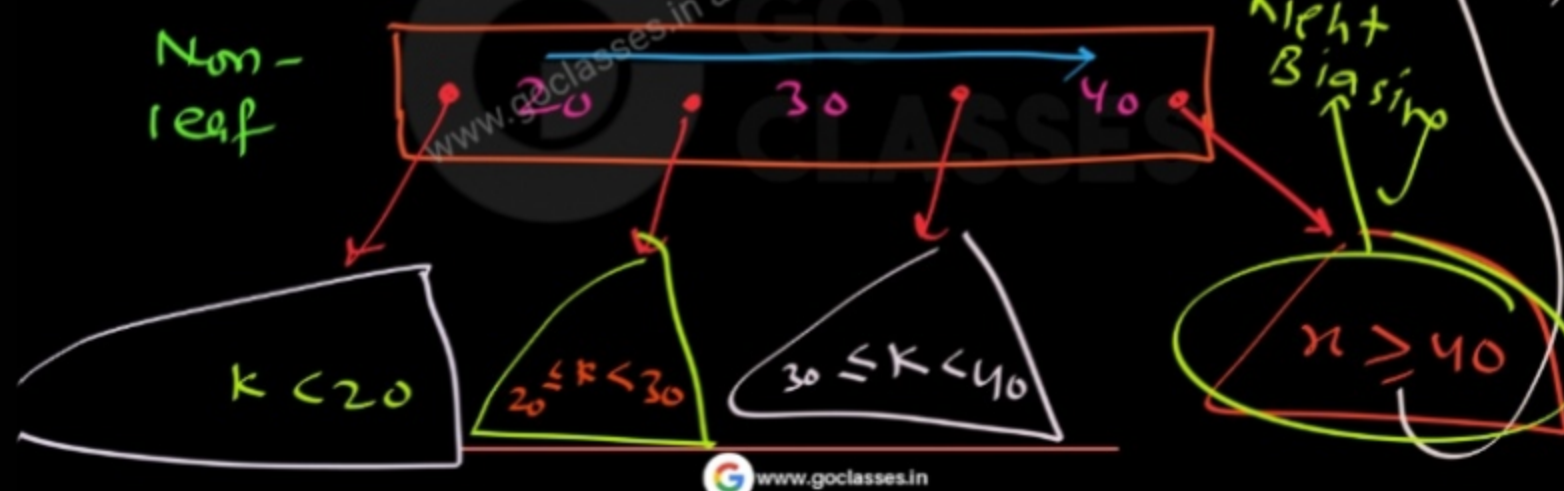
- ① Obvious Search tree rules  $\rightarrow$   
 $\rightarrow$  B+ Trees are Right

biasing.

Meaning key values in non leaf nodes can be repeated in the right subtree as immediate successor.

B+ Tree Rules:

- ① Obvious Search Tree Rules:





- ② All leaves on same level
- ③ Order  $p \rightarrow$  Max # Total Pointers in a Node

**Database Management System** GO Classes

B+ Tree Order  $p$

Root: 2 BP to  $p$  BP

Leaf:  $\lceil \frac{p-1}{2} \rceil + 1$  to  $p$  Total pointers

Remaining Internal nodes:  $\lceil \frac{p}{2} \rceil$  BP to  $p$  BP

Tree Pointer

Root: 2 BP to  $p$  BP

Nonroot }  $\lceil \frac{p}{2} \rceil$  to  $p$  BP

Non Leaf }

Leaf Node: 1

$\lceil \frac{p-1}{2} \rceil$  to  $p-1$  tree pointer to next leaf

$1 + \lceil \frac{p-1}{2} \rceil$  to  $p$  <key, data pointer> pairs

Total pointers in leaf

#data pointer =  
# Keys in leaf Node  
=  $p-1$

**Database Management System** GO Classes

B+ Tree Order  $p$

Root: 1 keys to  $p-1$  keys

Leaf:  $\lceil \frac{p-1}{2} \rceil$  keys to  $p-1$  keys

Remaining Internal nodes:  $\lceil \frac{p}{2} \rceil - 1$  keys to  $p-1$  keys

**Database Management System** GO Classes

Leaf Node Structure:

$\langle k_1, dp_1 \rangle \langle k_2, dp_2 \rangle \langle k_3, dp_3 \rangle$

Tree pointer (BP)

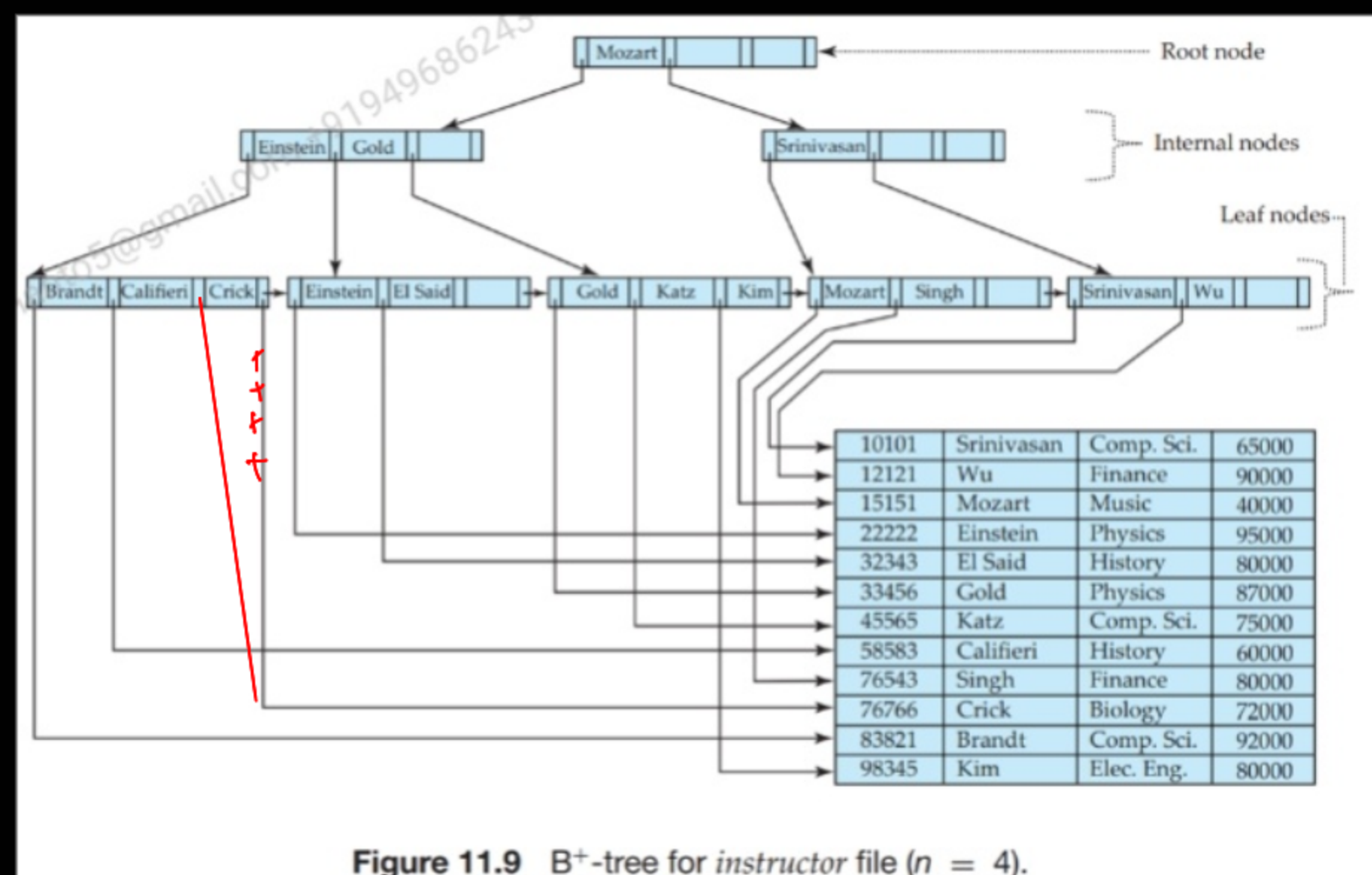
Next leaf

1 BP; If 2 Keys then 2 Data pointers

**Database Management System** GO Classes

Internal Node Structure:  $\Rightarrow$  { No Data Pointers }

BP<sub>1</sub>  $k_1$  BP<sub>2</sub>  $k_2$  BP<sub>3</sub>  $k_3$  BP<sub>4</sub>  $k_4$  BP<sub>5</sub>



Remember data pointers are  
Removed pointers by default ☆

**Database Management System** GO Classes

Q: Order  $p$  of B+ Tree = Maximum number of Pointers per Node. For what values of  $p$  is the tree of below Figure a legal B+ tree?

Violation

17

5 14 27

2 4 5 14 15 17 27 32

$p=3$  ✓

$p=4$  ✗

Leaf:  $\lceil \frac{4-1}{2} \rceil = 2$  keys at leaf

**Database Management System** GO Classes

Q: Suppose our blocks are 4096 bytes. Also let keys be integers of 4 bytes and let pointers be 8 bytes. If there is no header information kept on the blocks, then find the Order of B+ Tree (where order is defined as maximum number of pointers per B+ tree node)

$4 + 8 = 12B$  for  $\langle \text{key}, dp \rangle$

leaf node

$(p-1) \cdot 12 + 8 \leq 4096$

$\Rightarrow 12p - 12 + 8 \leq 4096$

$\Rightarrow 12p \leq 4100$

$p = \lfloor \frac{4100}{12} \rfloor = 341$

Internal node

$p$  — #Block pointers

$p-1$  — #keys

$\Rightarrow 12(p-1) + 8p \leq 4096$

$\Rightarrow 12p \leq 4100 \Rightarrow p = \lfloor \frac{4100}{12} \rfloor$

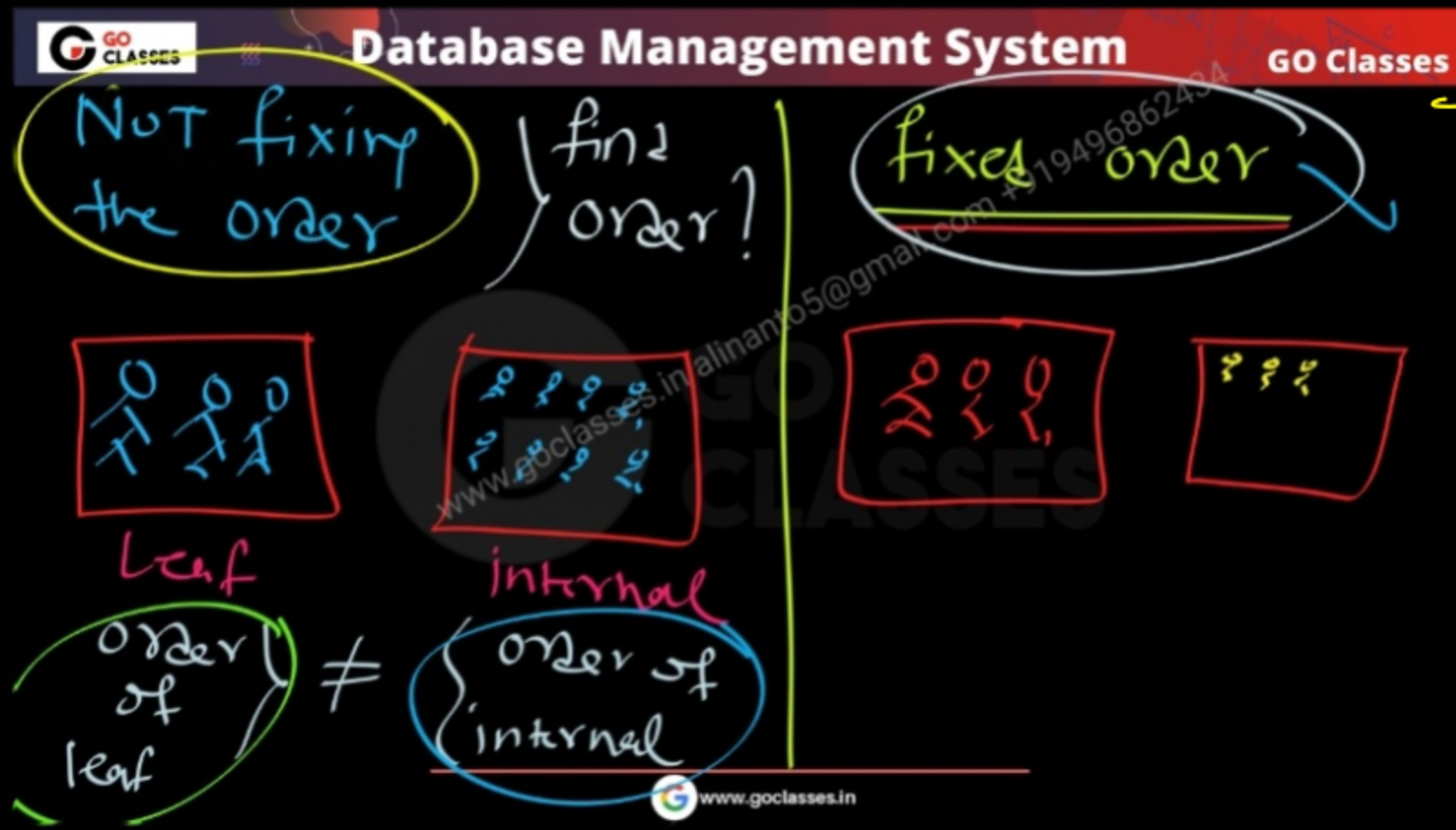
$p = 341$

①  $\rightarrow$  Both have  $p-1$  keys &  $p$  BPs

☆ Note if Data pointer = Block pointer then  
Order of Leaf Node = Order of Internal Node



★ → ② if Data pointer = Record pointer ⇒ Generally #Record pointers > #Block pointers  
 ⇒ Max order of Leaf Node < Max order of Non Leaf  
 ⇒ But order of key = order of Leaf Node → Limiting factors in this case  
 ⇒ Internal nodes not fully utilised



→ In questions asked, if not fixed order given, the this rule can be followed

eg:- Q: Record pointer 9 Bytes  
 Suppose our blocks are 4096 bytes. Also let keys be integers of 4 bytes and let pointers be 8 bytes. If there is no header information kept on the blocks, then find the Order of Leaf Node; order of internal node:

Leaf Node

$$p-1(9+4) + 8 \leq 4096$$

$$13p - 5 \leq 4096$$

$$p = \left\lfloor \frac{4101}{13} \right\rfloor = 315$$

Internal

$$p \cdot 8 + (p-1) \cdot 4 \leq 4096$$

$$12p \leq 4100$$

$$p = \left\lfloor \frac{4100}{12} \right\rfloor = 341$$

Search

In B Tree:

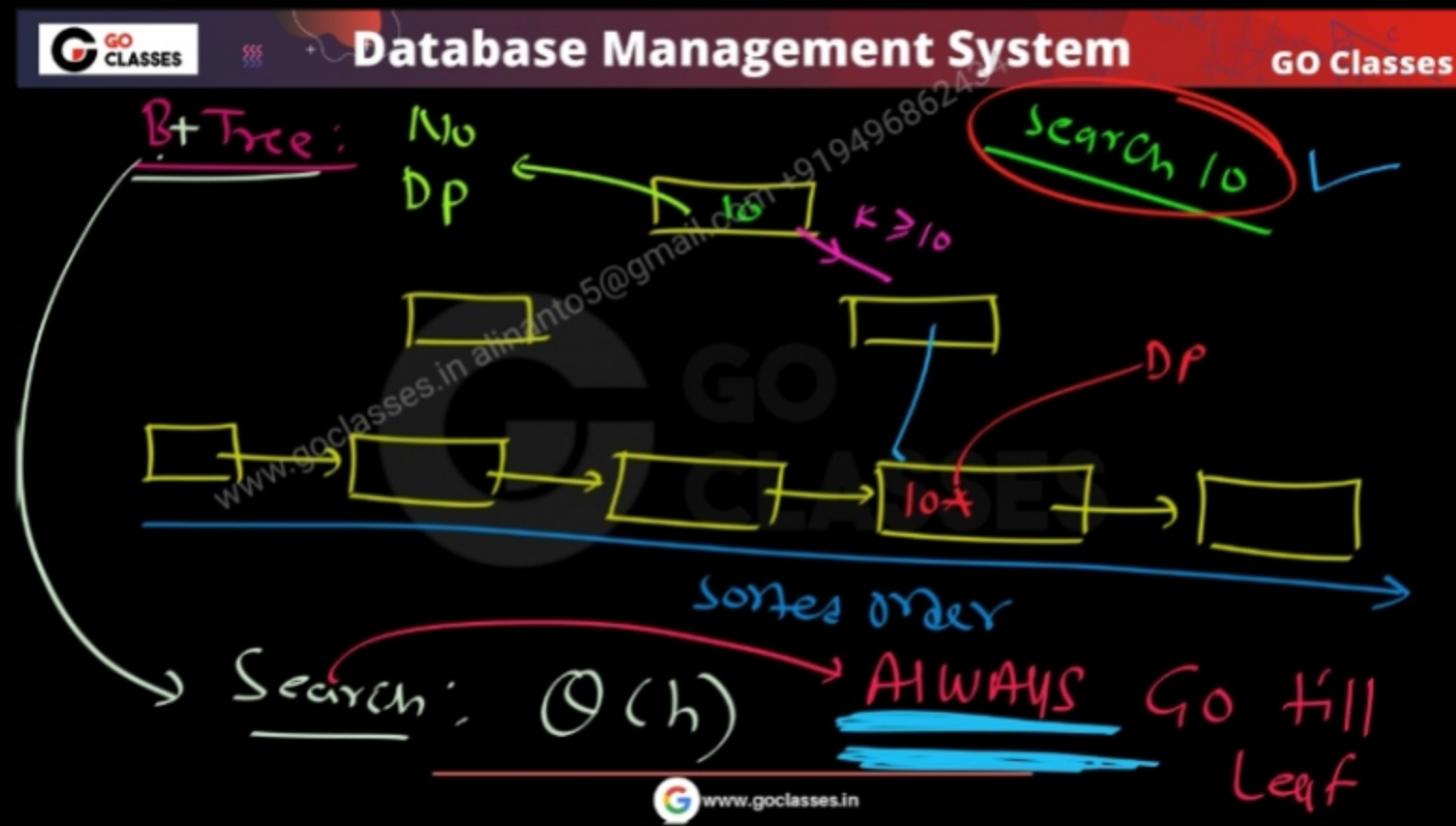
Search stops anywhere from Root to leaf.

$O(h)$  ✓  
 $\Theta(h)$  ✗

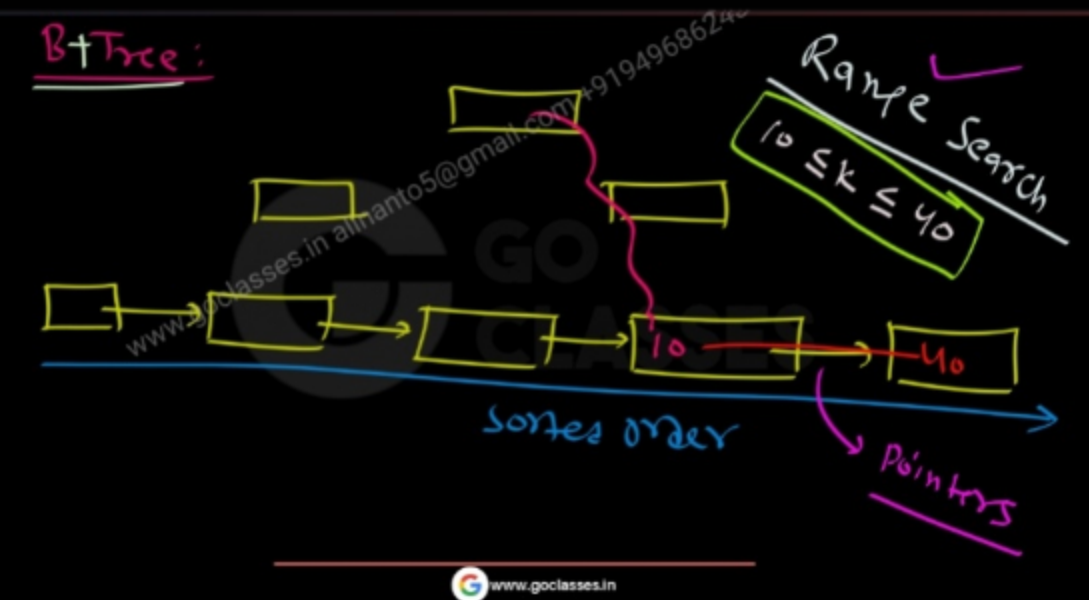
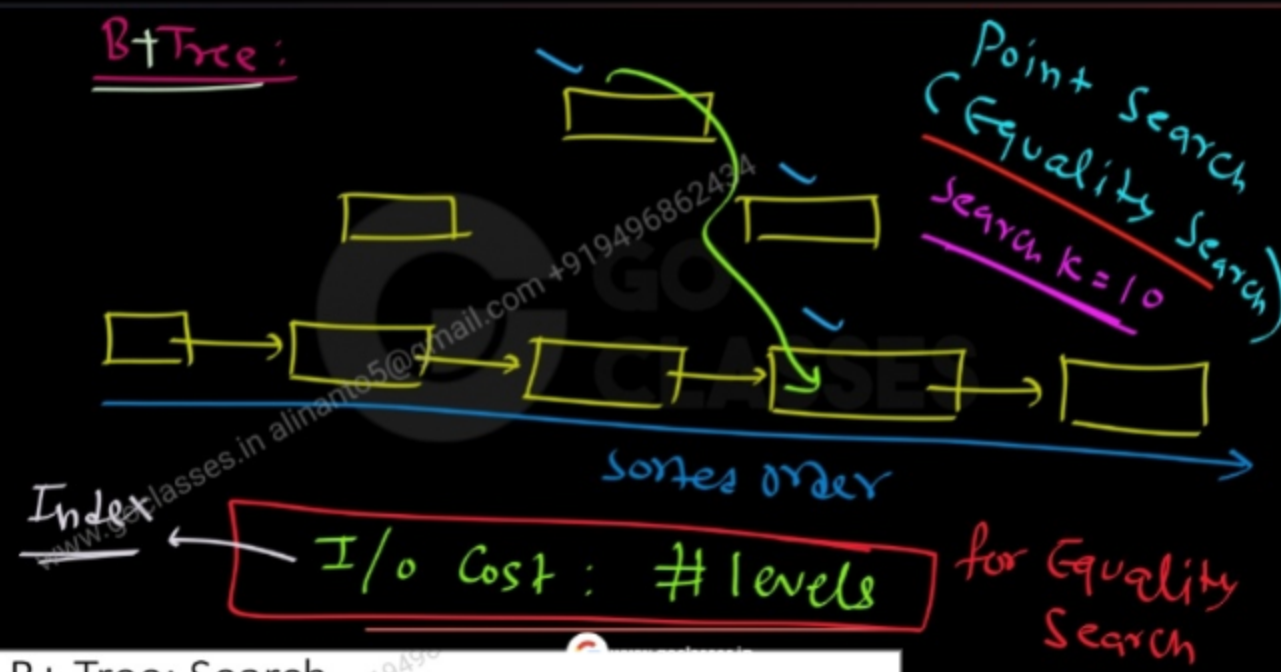
If Search 10 then we will go till Leaf. No T

DP Search 10

Data Block



$O$  Big O → worst case  
 $\Theta$  Theta → Avg case  
 $\Omega$  omega → Best case



### Searching a B+ Tree

- For exact key values:
  - Start at the root
  - Proceed down, to the leaf

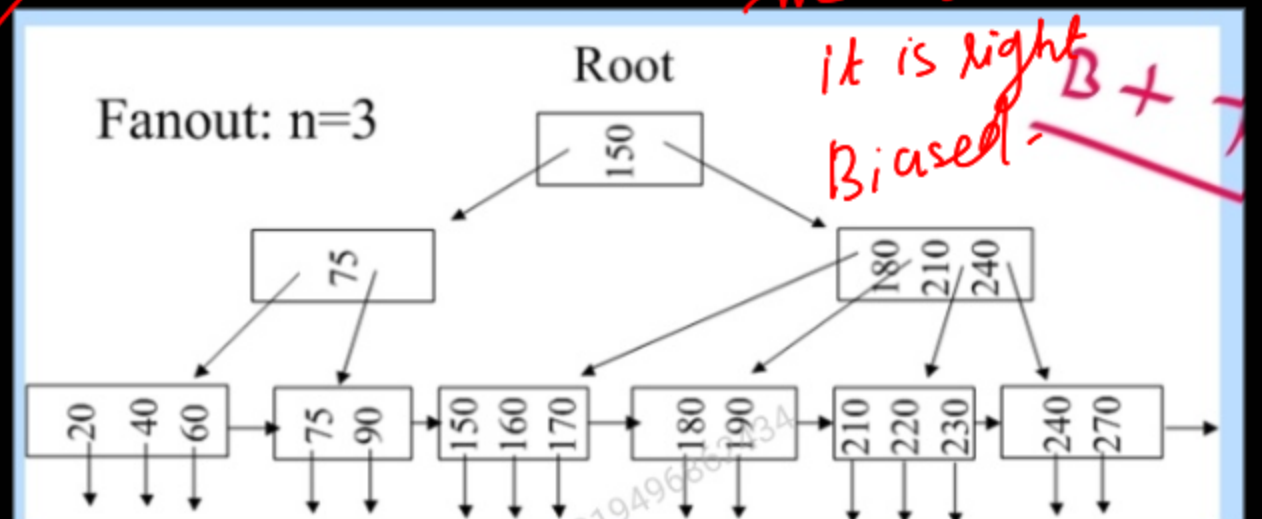
SELECT name  
FROM people  
WHERE age = 25

- For range queries:
  - As above
  - Then sequential traversal

SELECT name  
FROM people  
WHERE 20 <= age  
AND age <= 30

### B+ Tree: Search

- start from root
- examine index entries in non-leaf nodes to find the correct child
- traverse down the tree until a leaf node is reached
- non-leaf nodes can be searched using a binary or a linear search



For each of the following queries, compute the minimum number of disk I/Os to retrieve the records (not including the disk I/Os to read the records). We assume we need to follow the sequence pointers of the leaf nodes to traverse the leaf nodes (if necessary).

- Find record(s) with the key value 160. 3-3
- Find record(s) with the key value 280. 3
- Find records with the key values in the range 30 to 50 inclusive. 3
- Find records with the key values in the range 30 to 60 inclusive. 3
- Find records with the key values in the range 30 to 75 inclusive. 4
- Find records with the key values in the range 30 to 90 inclusive. 4
- Find records with the key values in the range 30 to 220 inclusive. 7
- Find records with the key values in the range 30 to 240 inclusive. 8

- Find records with the key values in the range 30 to 90 inclusive: 3 I/Os. ✗
- Find records with the key values in the range 30 to 75 inclusive: 3 I/Os. ✗
- Find records with the key values in the range 30 to 75 inclusive: 5 I/Os. ✗
- Find records with the key values in the range 30 to 220 inclusive: 7 I/Os. ✓



# Insertion

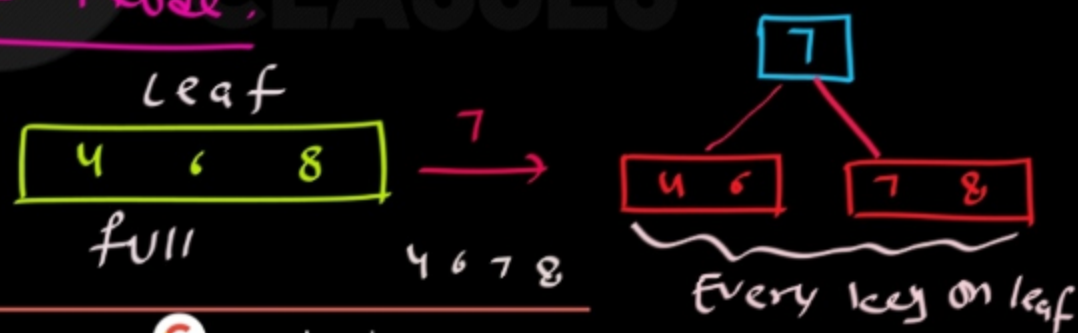
Similar to B Tree Insertion, Except when we split Leaf node.

Split of Internal Node Same as B Tree

Split of leaf Node:

Order: 4

max #keys=3

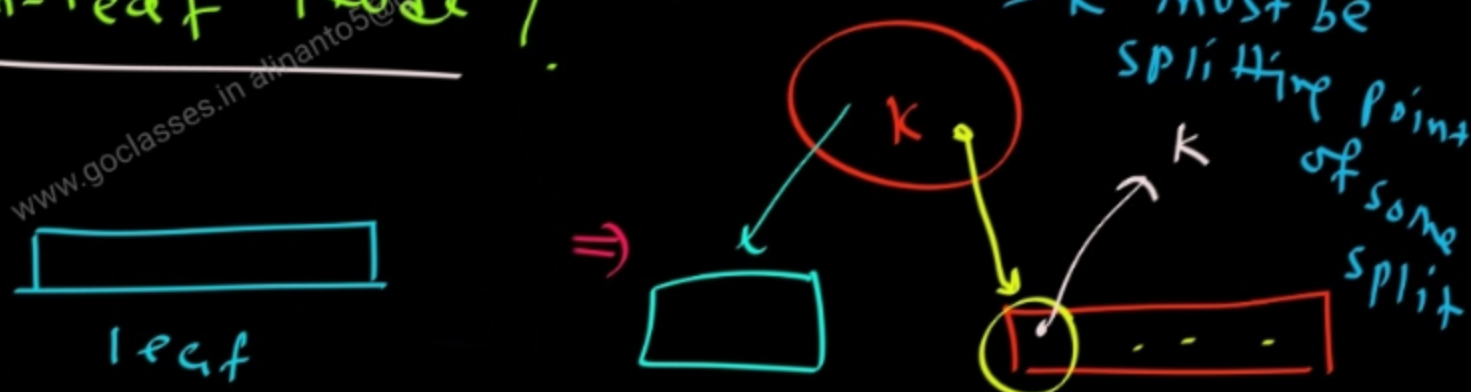


| Leaf page full? | Index page full? | Action                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|-----------------|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| No              | No               | Place the record in sorted position in the appropriate leaf page                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| Yes             | No               | <ol style="list-style-type: none"> <li>1. Split the leaf page including the inserted record.</li> <li>2. Place Middle Key in the index page in sorted order.</li> <li>3. Left leaf page contains records with keys below the middle key.</li> <li>4. Right leaf page contains records with keys equal to or greater than the middle key.</li> </ol>                                                                                                                                                                                                                     |
| Yes             | Yes              | <ol style="list-style-type: none"> <li>1. Split the leaf page including the inserted record.</li> <li>2. Records with keys &lt; middle key go to the left leaf page.</li> <li>3. Records with keys &gt;= middle key go to the right leaf page.</li> <li>4. Split the index page.</li> <li>5. Keys &lt; middle key go to the left index page.</li> <li>6. Keys &gt; middle key go to the right index page.</li> <li>7. The middle key goes to the next (higher level) index.</li> </ol> <p>If the next level index page is full, continue splitting the index pages.</p> |

Observation ①

B+ Tree insertion: Always Happen at leaf Node.

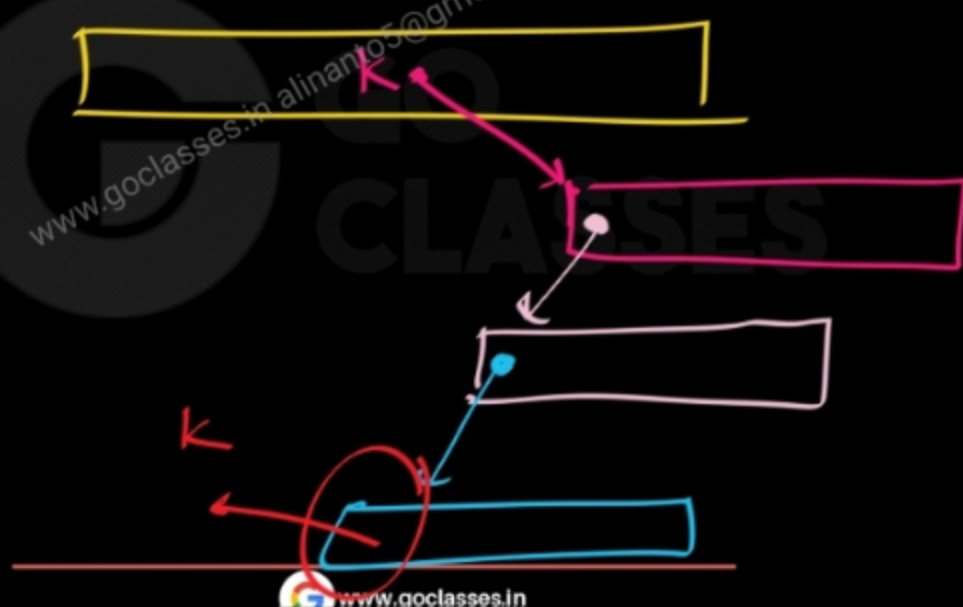
In which Condition Key k can go to Non-leaf Node?



→ Any key k in a nonleaf Node is also the leftmost key in the immediate Successor leaf node.

Observations:

① Right Biased B+ Tree:



☆ → A alone root node is a leaf node & not a intermediate node

Q 2. True/False??

Every value of leaf nodes is also present in non-leaf nodes in B+ tree.

→ No, only left most values in leaf nodes are present.  
→ left most value of left most leaf not present

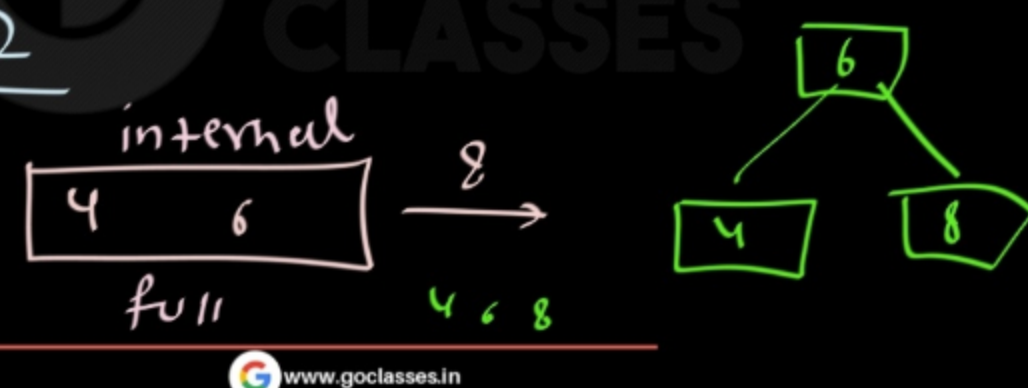
Q 4. True/False??

Every value of non-leaf nodes is also present in the leaf nodes in B+ tree.

Similar to B Tree Insertion, Except when we split Leaf node.

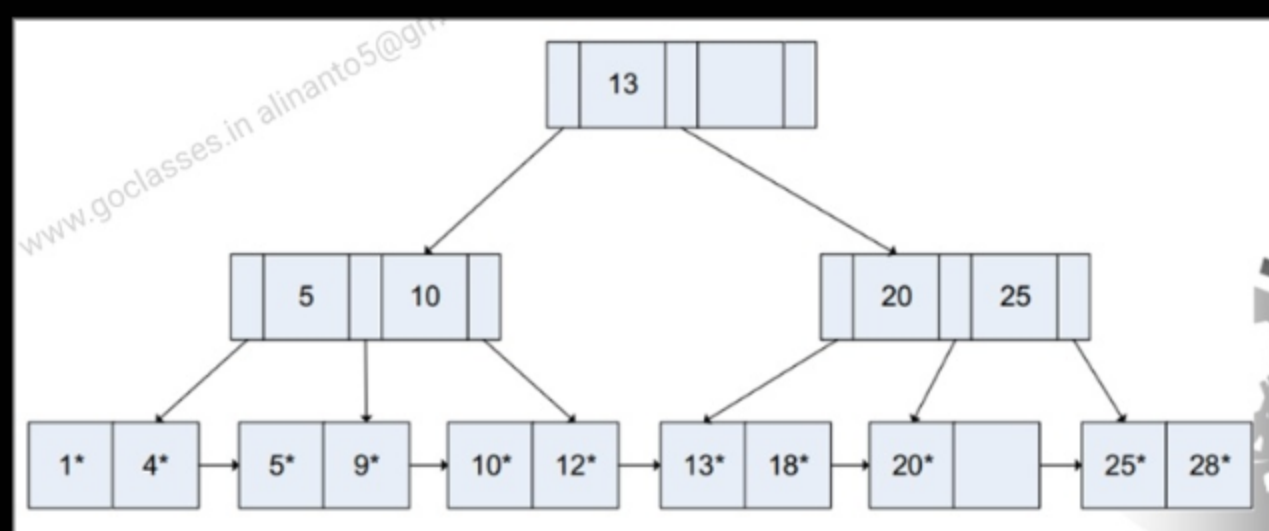
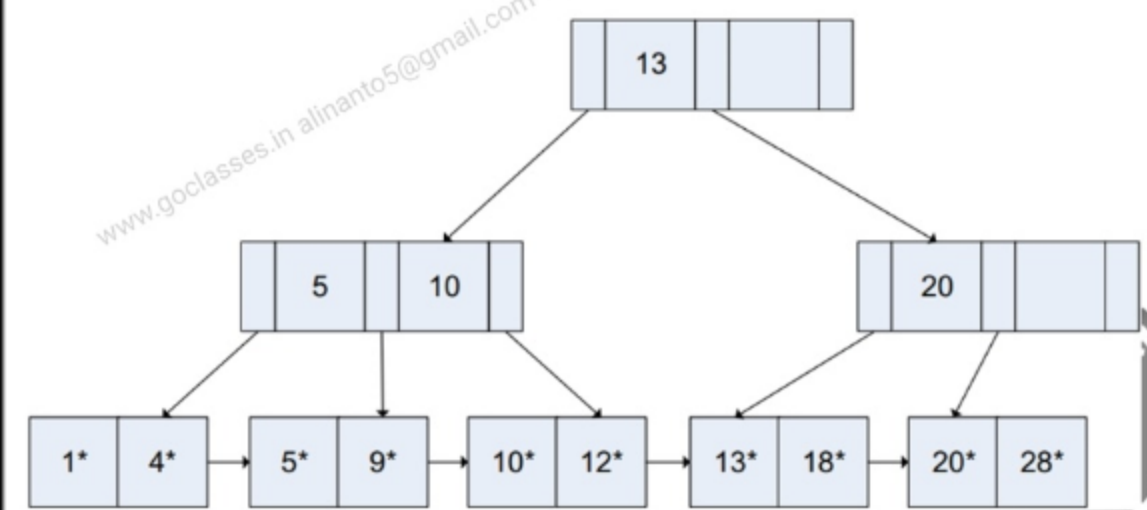
Internal Node:

max keys: 2



Insertion always starts at leaf Node

Insert 25 in the below tree:



⇒ #leaf Node = 1 + #Keys in Non-leaf Node

True for both B tree & B+ Trees

→ Observation ② → Order of Insertion Matters



Q 5. True/False??

Values in non-leaf nodes are all unique in B+ tree.

→ Value in a non leaf node, don't appear again, in another non leaf node

Q 6. True/False??

ALL searches (successful OR unsuccessful) in B+ tree take Exactly same amount of time(Disk block access or I/O cost).

→ In B+ Tree True  
In B Tree False

Order Questions

**Example 6.** To calculate the order  $p$  of a B<sup>+</sup>-tree, suppose that the search key field is  $V = 9$  bytes long, the block size is  $B = 512$  bytes, a record pointer is  $Pr = 7$  bytes, and a block pointer/tree pointer is  $P = 6$  bytes. An internal node of the B<sup>+</sup>-tree can have up to  $p$  tree pointers and  $p - 1$  search field values; these must fit into a single block. Hence, we have:

Order of Internal Node:  $(p)$

$$P(6) + (p-1)(9) \leq 512$$
$$p \leq \frac{512}{15} = 34.13 \dots$$

$p = 34$

Order of Leaf Node = (order  $y$ )

$y-1$  keys  
 $y-1$  RP  
1 BP

$$(y-1)(9+7) + 8 \leq 512$$
$$y \leq \frac{522}{16} = 32.625$$

$y = 32$

**Question 2. [20 marks]**

Suppose that the size of a search key field is  $V=16$  bytes, the size of a data (record) pointer is  $Pr=6$  bytes, the size of a block pointer/tree pointer is  $P=8$  bytes, the block size  $B$  is 1024 bytes. If we want to construct a B+ tree, what are the maximum values of the order of internal nodes, and the order of leaf nodes?

Order of internal node:  $\rightarrow p$

$$P(8) + (p-1)(16) \leq 1024$$
$$p \leq \frac{1024}{12} = 85.33 \dots$$

$p = 85$

Order of leaf node:  $\rightarrow y > p$

$$(y-1)(16+6) + 8 \leq 1024$$
$$y \leq \frac{1016}{22} = 46.18 \dots$$

$y = 46$

**GATE CSE 2002 | Question: 2.23, UGCNET-June2012-II: 26**

asked in Databases Sep 16, 2014 • edited Jan 31, 2018 by dj\_1 10,927 views

A B<sup>+</sup>-tree index is to be built on the Name attribute of the relation STUDENT. Assume that all the student names are of length 8 bytes, disk blocks are of size 512 bytes, and index pointers are of size 4 bytes. Given the scenario, what would be the best choice of the degree (i.e. number of pointers per node) of the B<sup>+</sup>-tree?

A. 16  
B. 42  
C. 43  
D. 44

leaf

$$(p-1)(8+4) + 4 \leq 512$$
$$12p \leq 520$$
$$\Rightarrow p = \left\lfloor \frac{520}{12} \right\rfloor = 43$$

Non leaf

$$(p-1)8 + p4 \leq 512$$
$$= 12p \leq 520$$
$$p = 43$$

The order of an internal node in a B<sup>+</sup> tree index is the maximum number of children it can have. Suppose that a child pointer takes 6 bytes, the search field value takes 14 bytes, and the block size is 512 bytes. What is the order of the internal node?

A. 24  
B. 25  
C. 26  
D. 27

$$p6 + (p-1)14 \leq 512$$
$$20p \leq 526$$
$$p = \left\lfloor \frac{526}{20} \right\rfloor = 26$$

The order of a leaf node in a B<sup>+</sup>-tree is the maximum number of (value, data record pointer) pairs it can hold. Given that the block size is 1K bytes, data record pointer is 7 bytes long, the value field is 9 bytes long and a block pointer is 6 bytes long what is the order of the leaf node?

A. 63  
B. 64  
C. 67  
D. 68

got lemon making lemonade

$$p(7+9) + 6 \leq 1024$$
$$16p \leq 1018$$
$$p = \left\lfloor \frac{1018}{16} \right\rfloor = 63$$

**3.1.20 B Tree: GATE CSE 2015 Set 3 | Question: 46**

Consider a B+ tree in which the search key is 12 byte long, block size is 1024 byte, recorder pointer is 10 byte long and the block pointer is 8 byte long. The maximum number of keys that can be accommodated in each non-leaf node of the tree is 50.

$$k12 + (k+1)8 \leq 1024$$
$$20k \leq 1016$$
$$k = \left\lfloor \frac{1016}{20} \right\rfloor = 50$$

Order  $p$ : max no. of Total pointers in a node.

Leaf:  $\left\lfloor \frac{p-1}{2} \right\rfloor$

to  $p-1$  keys

why in Leaf only??

Gives more Balanced Structure



# Question on Max, Min, Structure

Consider a DBMS that has the following characteristics:

- 2KB fixed-size blocks
- 12-byte pointers
- 56-byte block headers

We want to build an index on a search key that is 8 bytes long. Calculate the maximum number of records we can index with

- a 3-level B+ tree (2 levels plus the root)
- 

leaf Node/Internal — Since BP ⇒ DP

$$56 + p(12) + (p-1)8 \leq 2048$$

$$\Rightarrow 20p \leq 2048 + 8 - 56$$

$$20p \leq 2000$$

$$p = 100$$

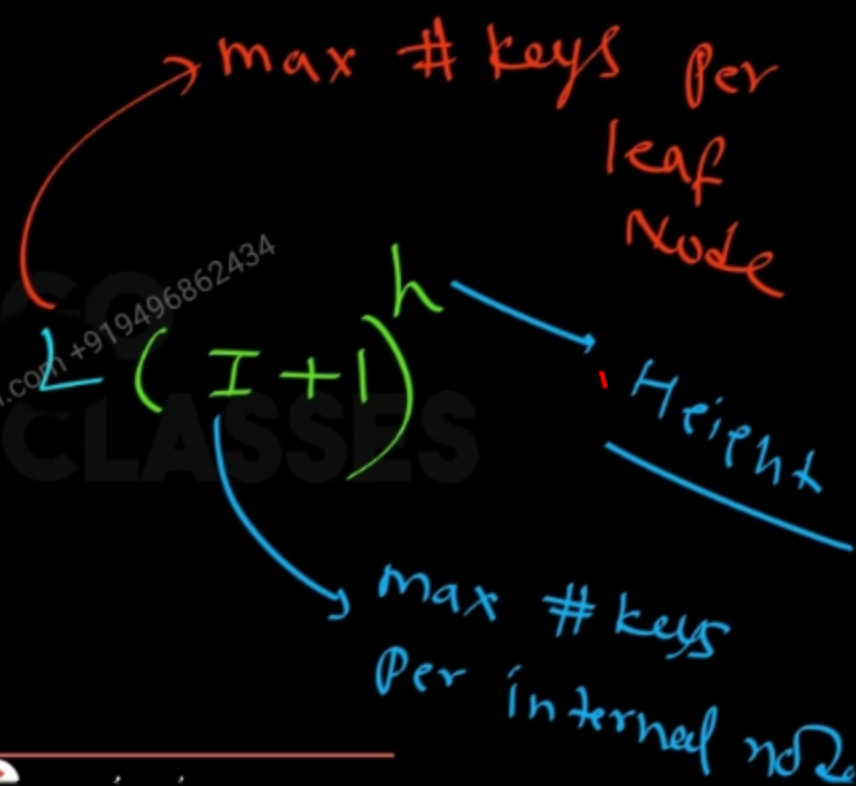
Root level 1 → 99 keys  
 level 2 → 100 × 99 keys  
 level 3 → 100 × 100 × 99 keys = 990000 keys

} not usefull

B+ Tree:

Height h:

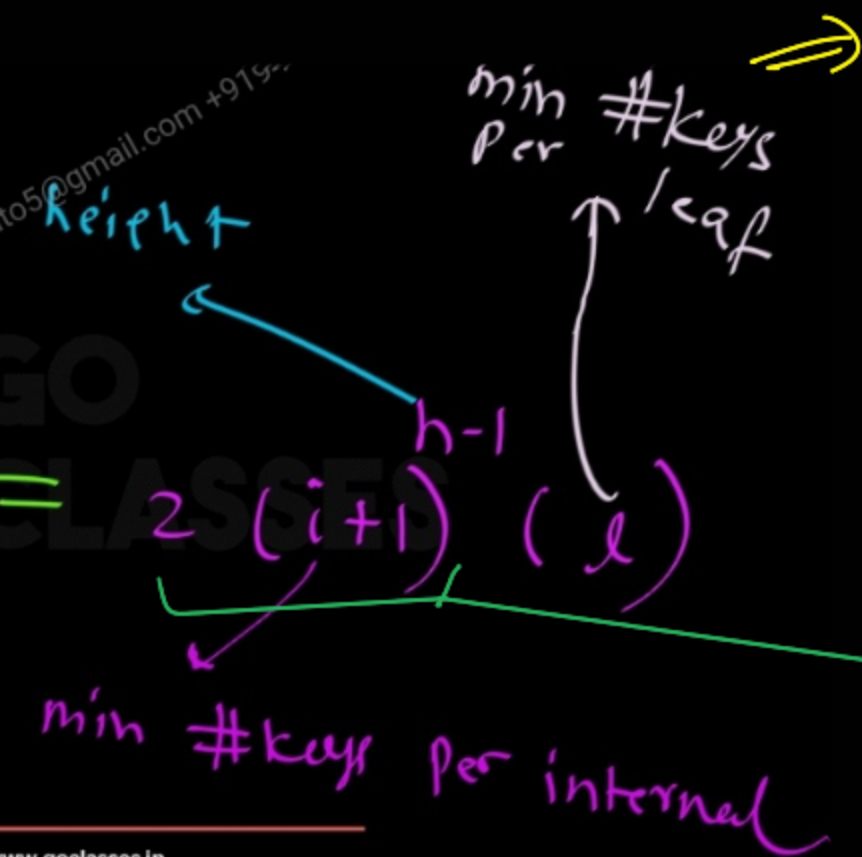
max #keys:



B+ Tree:

Height h:

$$\text{min \#keys} = 2(i+1)(l)$$

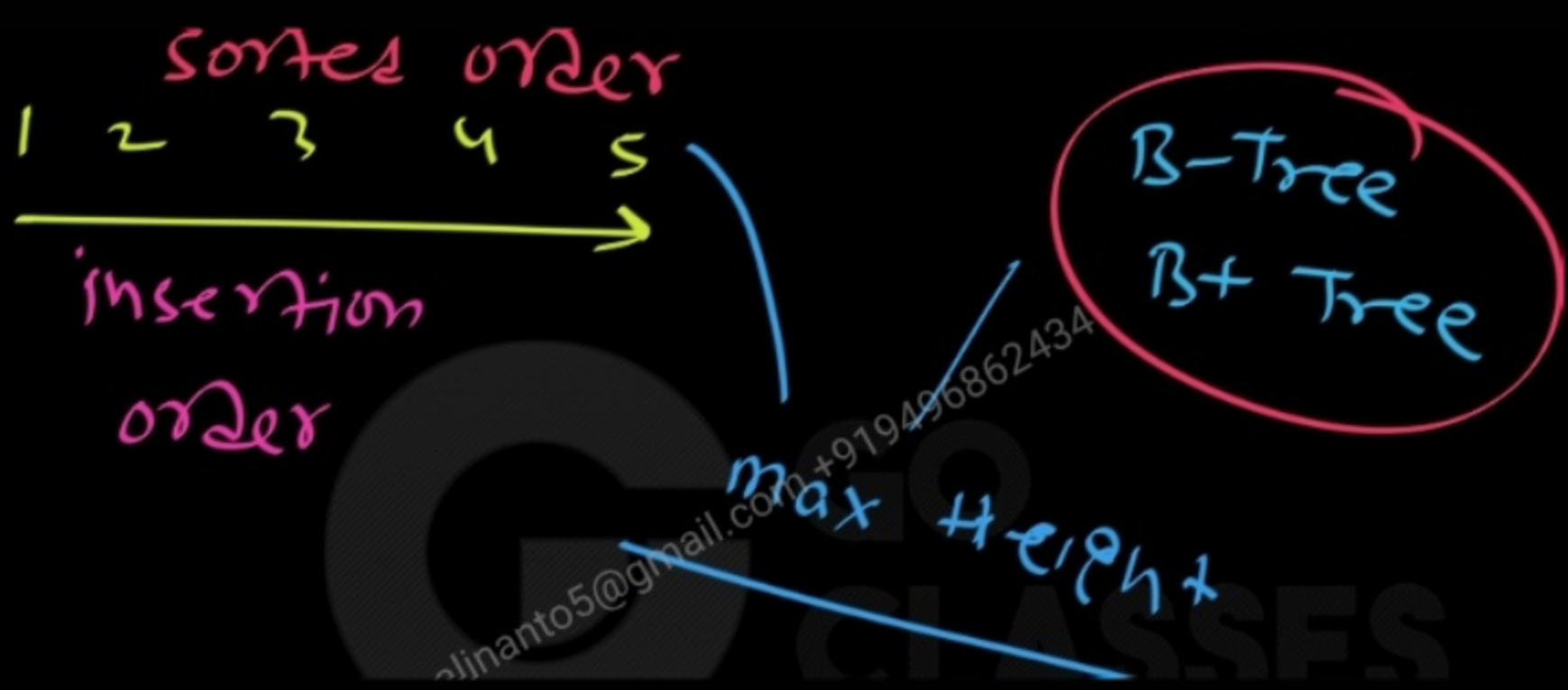


| Min height | Nodes          |
|------------|----------------|
| 0          | 1              |
| 1          | 2              |
| 2          | $2(i+1)$       |
| 3          | $2(i+1)^2$     |
| h          | $2(i+1)^{h-1}$ |

Questions of Type → Given # Keys → Max & Min Height of B+ Tree

- 2 Methods →
- ① Reverse above formula
  - ② Top-down conceptually.

Bulk loading (Bottom up Insertion)  
 (Simultaneously Insertion)  
 B+ Tree  
 Heap (In-memory Data Structure)



Q: Given a Set of keys

- ✓ min. Height possible?
  - ✓ max " " ?
- Bulk loading  
 Concept



Create a B+tree of order 5 for the following data arriving in sequence:

92 24 6 7 11 8 22 4 5 16 19 20 78

8. Create a B+tree of order 5 for the following set of keys with

92 24 6 7 11 8 22 4 5 16 19 20 78

#keys: 13 keys

Bulk loading

Leaf: 2 to 4 keys

minimum Height!

Bottom-up: (minimum height  $\Rightarrow$  max keys per Node)

1; #levels = 2  
#Nodes = 5

4 BP

Level 1

Root

4 BP

Lo

leaf level  $\Rightarrow$  13 keys

#Leaf Nodes:  $\lceil \frac{13}{4} \rceil = 4$  Nodes

Min height  $\begin{cases} \#nodes = 5 \\ \#levels = 2 \end{cases}$

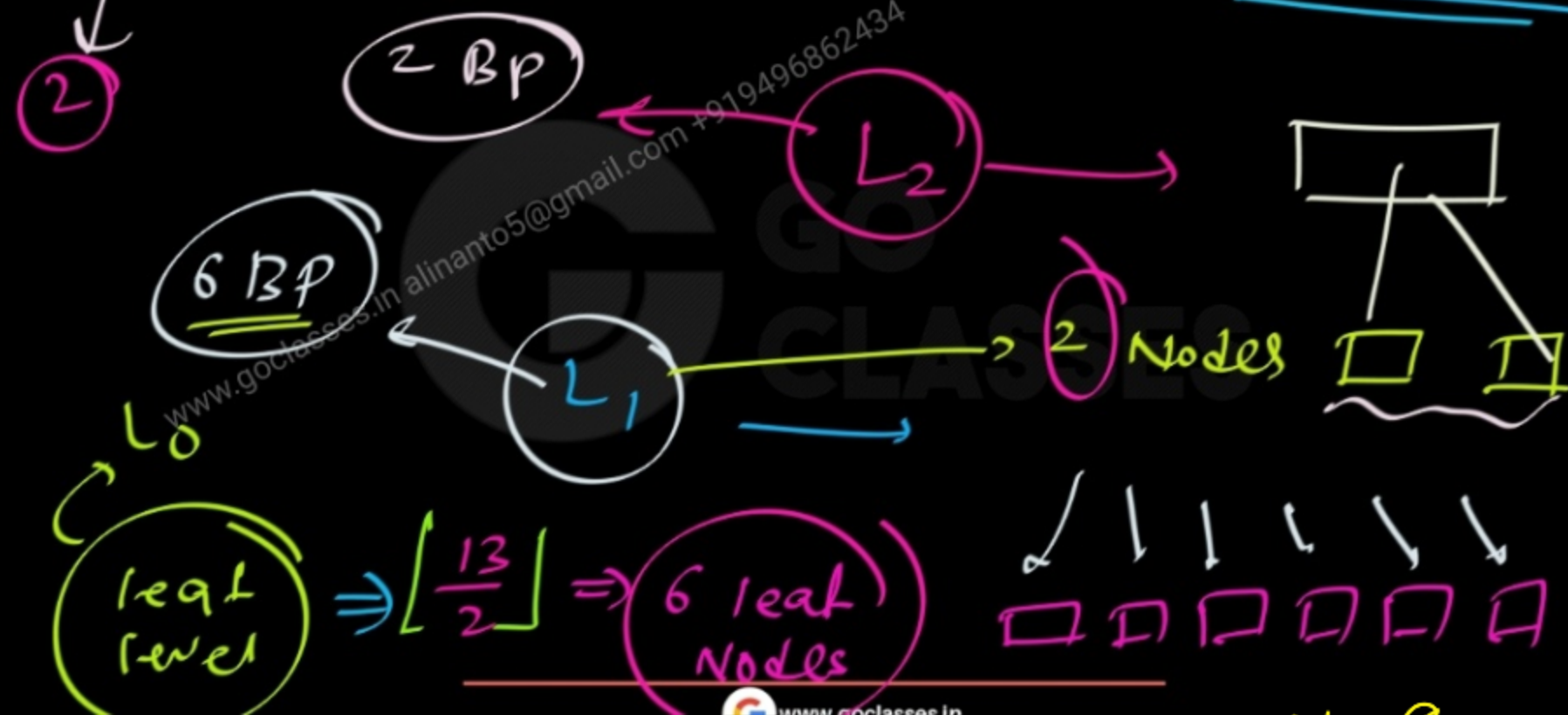
for minimum Height:

Put max #keys per Node

Internal node: 3 to 5 pointers

Root node: 2 to 5 pointers

max. Height: fill every Node as less as possible



Max Height  $\begin{cases} \#Node = 6 + 2 + 1 = 9 \\ \#levels = 3 \end{cases}$

For Same system  $\rightarrow$  same file, Block size, #Keys  $\rightarrow$  B+tree has less levels & less #Nodes

why?  
why?



Internal node

B+ Tree

we can put more child pointers because we save space of Record pointer.

So B+ Tree will have less height

HW B Tree  $\rightarrow$  n keys

B+ Tree  $\rightarrow$  in internal node we can put more #keys, more #child pointers

3.1.20 B Tree: GATE CSE 2015 Set 3 | Question: Consider a B+ tree in which the search key is 12 byte long, block size is 1024 byte, recorder pointer is 10 byte long and the block pointer is 8 byte long. The maximum number of keys that can be accommodated in each non-leaf node of the tree is \_\_\_\_\_.

#levels B Tree  $\geq$  #levels B+ Tree

Same System (Same Block Size)



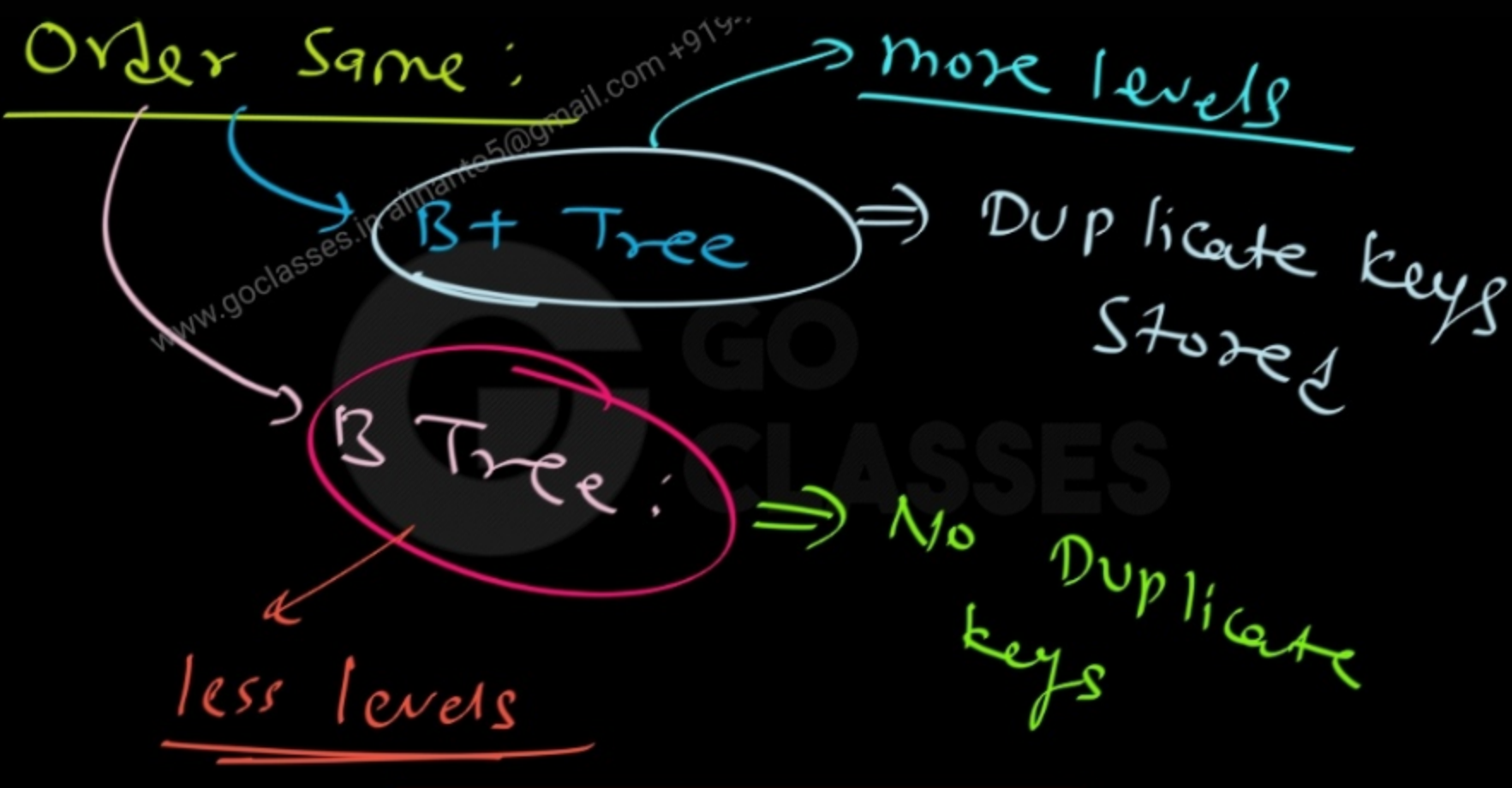
$\rightarrow$  Intuition  $\rightarrow$  B+Tree has higher fanout  $\Rightarrow$  Less height



# Comparison Type:

Order of B, B+ tree is Same:

For n Keys, which index, B or B+ tree will take Less number of levels & less number of nodes..



## Comparison Type:

Range Query: B+ Tree  $\rightarrow$  Leaf level = linked List

Which index, B or B+ tree should be preferred??

When we process the data randomly, we modify the tree search to find the target data only at a leaf. The search time is thus increased slightly. However, to process the data sequentially, we simply locate the far-left entry and then process the data as though we were processing a linked list in which each node is an array.

1. In B+ tree data is ordered in sequential linked list and the data is stored in leaf nodes but in B tree the leaf nodes cannot be stored in linked list due to lack of the sibling pointers.
2. In B+ tree the data is stored in leaf node so searching of any data requires scanning only of leaf node alone, whereas in B tree data is stored in non-leaf nodes as well and to search any data value the complete tree needs to be traversed.

5. As the data is not there in non-leaf node so the fan out is higher in B+ tree, tree depth is kept shorter and thus there are less reads done from secondary storage. [11]

B tree  $\rightarrow$

- Random access more costly on average
- sequential access  $\rightarrow$  less costly