

Video - 1

Intro to Data Structure

Data Structure

- ⇒ ways to arrange data in main memory for efficient usages.
- ⇒ eg: array, linklist, stack, queue, BST etc.

Algorithm

- ⇒ sequence of steps to solve a given problem.
- ⇒ eg: sorting an array

Data Base

- ⇒ Main Memory : \Rightarrow RAM
- ⇒ Jab koi program chalo hota hai to wo main memory (RAM) $\frac{1}{2}$. Load $\frac{1}{2}$ $\frac{1}{2}$ then something will happen in the program.

↓
whatever's is the programme code
↓
whatever has been written
↓
whatever the sequence of steps
are there to open a new
file , to download a new file
↓
and fiddling with the data will
happen.
↓
make operations are done to
arrangement optimal .
↓
To minimize the usage of RAM
↓
optimally load the program in
main memory
↓
Data Structure

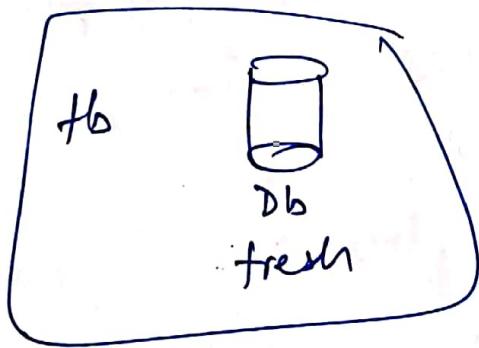
Data - Base

→ Read, Retrieve & Update data from Hard Disk. ⇒ eg: MySQL

Data warehouse

→ legacy data

→ job will data Ram me more w/o, it will get lost after closing the program.



+



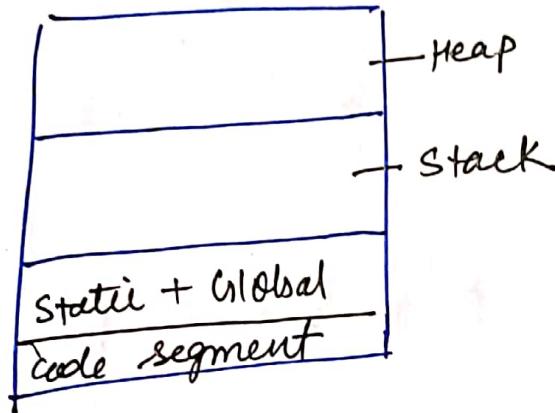
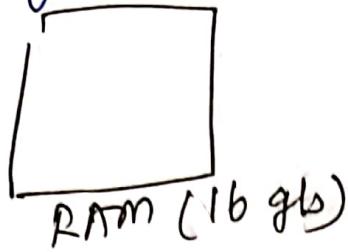
legacy data

Big data

→ Huge data that your normal algorithm may not be able to deal with it

→ diff. analysis & techniques are used.

memory layout in C++

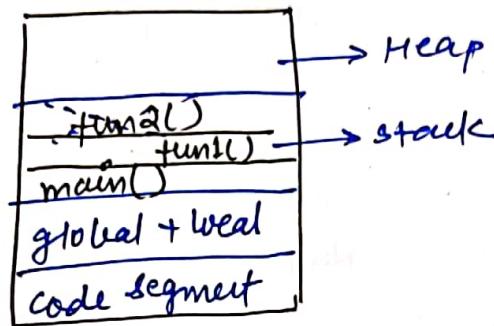


=> program as you open gets loaded
in main memory (RAM)

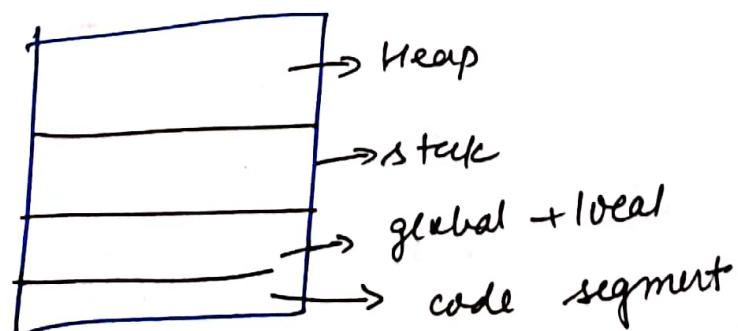
* Stack :- normal memory
eg: int i, char ch

eg:
fun1()
{
 fun2();
}
main()
{
 fun1();
}

→ when main function gets - called
↓
then it will get its place here.



- let's say fun2 returned '3' after final calculation,
now its data in stack will get erased
- similarly fun1() will execute & after returning a value, its data in stack will get erased
- so, after finishing the program all will get erased similarly
- III



→ And finally stack gets emptied.

Heap

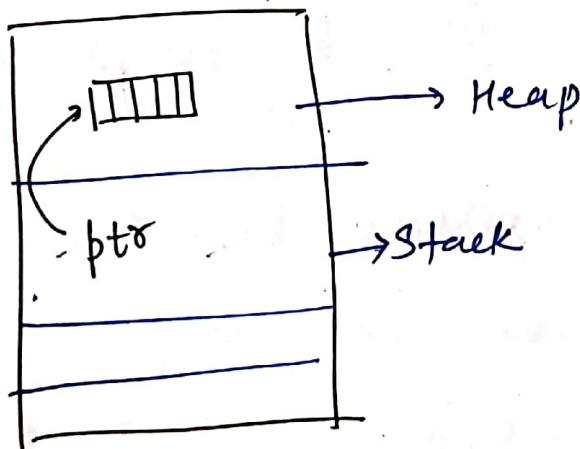
→ program allocates dynamic memory in the heap

→ we request with help of "Pointer".

* Pointer : → variable which stores the address.

we use

malloc → C } to request
new operator → C++ } dynamic memory



* we requested in Heap because
* wanted to create it and then

delete it right there inside that function

→ so, when we want we can request & delete this memory

→ But in start till our program is there, it will be there.

→ with the help of pointer we can allocate our memory from stack to Heap

→ we have to use our memory very efficiently & by running our algorithm in very less time

→ Both space & time is important

Ques → Heap is used instead of Stack?

⇒ Yes, it can be done but what if I had to delete in the same function.

→ Stack of any function ends, when the function is returned

Ex:-

fⁿ are getting called

↓
a new stack is being created

↓
it started with main() here

↓
main():= int i
 fun1()

↓
returned value of fun1() is called
and it(stack) gets paused
the main() function

↓
fun2() then pauses fun1()
then get its returned value

↓
as their values gets returned,
their execution will keep resuming

How C program is run inside over memory

1st : → code will be copied
(code segment)

code that is copied is machine code

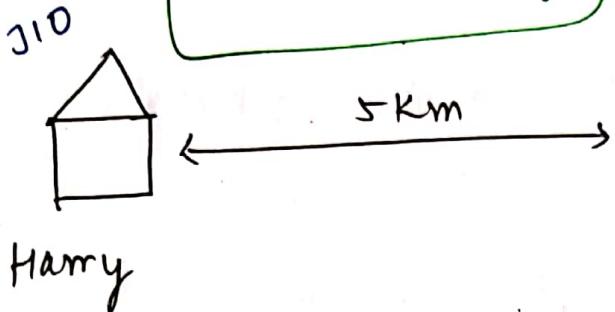
2nd : - stack + heap works

Memory leak

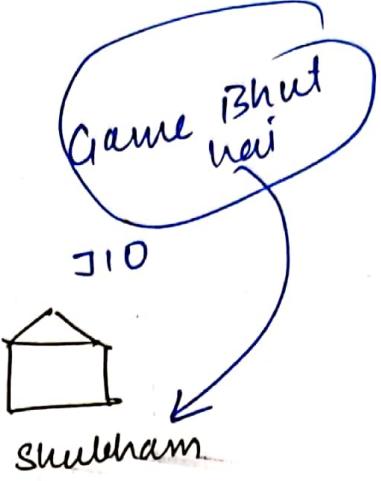
- You have to use the heap responsibly
- if made a mistake while using the heap then you can cause memory leak by using a lot of memory here
- you haven't free up the memory of map & you keep requesting

Video - 2

Time Complexity



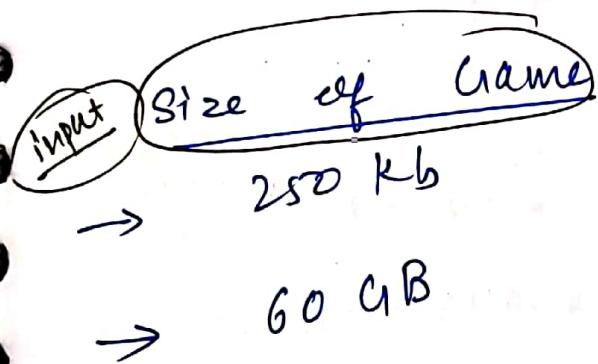
Game izhut nei



1GB/day.



game leue jaa raha



medium → Algo 1
internet

Physical visit
→ Algo 2

As the size of input was increased like that, the run time of the algorithms keep on increasing (changing)

Speed of net → 250 kb/s

250 kb → 1s

1 mb → 4s

Algo 2

→ same hard-disk, same bike,
hard-disk me you bring 250 kb
or
1 Tb
time remains same

ii) Keeping :- Assume ignore the copying time
copying time is negligible

Note:-
→ After increasing your input, run time
no change in your run time

→ we generally check:
As the size of input keeps on
increasing, similarly what is the
effect of algorithm on runtime

↓
asking the answer to this question
ii)
Analysis \Rightarrow Time complexity
analysis

Algo 2

k_1 = to wear clothes (time)

k_2 = time to travel from your house to friend's house

k_3 = time to do snacks at friend's house

k_4 = time to return back to your house

n = size of the input (game size)

$$t^{(n)}_{\text{algo 2}} = k_1 + k_2 + k_3 + k_4$$

$$[n^0 = 1]$$

in terms of n

$$t^{(n)}_{\text{algo 2}} = k_1 n^0 + k_2 + k_3 + k_4$$

Note:-

out these terms, which term can make it big?

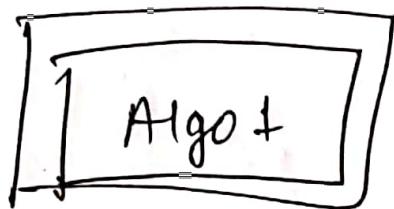
$$\Downarrow \\ = n^0 \quad \text{imperfect terms}$$

$$= O(n^1) \rightarrow \text{Big } O$$

$$= O(1) \rightarrow \text{Big } O \text{ of } 1.$$

→ ∴ This algorithm runs in constant
time

→ Big O(1) ⇒ Constant



→ Upload ~~of~~^{of} send ~~on~~^{to} ~~your~~^{the} system

l_1 = time to travel on your system

l_2 = speed of network (wi-fi)

l_2 (kb/s)

n = size of game

time to transfer game = $\left(\frac{n}{l_2}\right)$

$$t_{\text{algo+}}^n = l_1 + \left(\frac{n}{l_2}\right)$$

$$= l_1 + l_2' n$$

$$= l_1 n^0 + l_2' n^1$$

$$\frac{1}{l_2} = l_2'$$

\therefore Most impetuous term $\Rightarrow \Theta(n)$

$$\approx n^1$$

$$\approx O(n) = O(n)$$



Big O of n



Linear

Note

$$O(n \log n)$$

$$O(n^2)$$

$$O(n^3)$$

$$O(n!)$$

* Time Complexity of Big O

↳ time required to run your algorithm ; it scales according to what

- ↓
- ⇒ If it runs in linear time $\rightarrow O(n)$
 - ⇒ If it runs in const. time $\rightarrow O(1)$
-

Big O

Mathematically

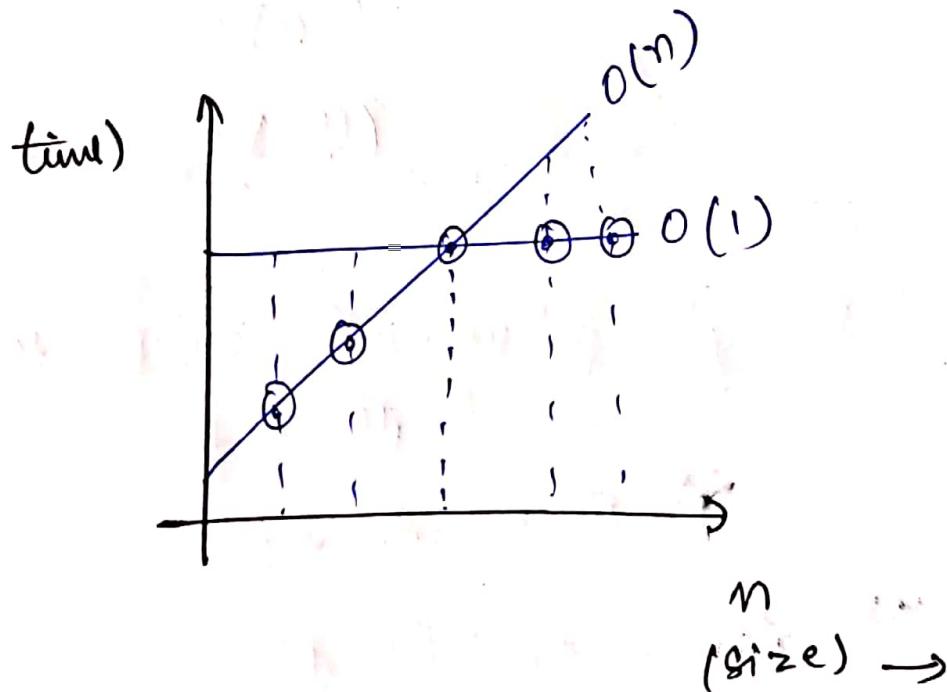
$O(n)$ is also $O(n^2)$
is also $O(n^3)$

Industrial

Order of

→ jitna kaam
hi ek Aana ga

→ Min. of these



Video - 3

[O, Ω , Θ]

→ we use Asymptotic notations to compare 2 algorithm

→ 3 types of Asymptotic notations are there:

Big O, Big Theta, Big Omega
 (O) (Θ) (Ω)

Big Oh (O)

$O \leq f(n) \leq c \cdot g(n) \quad \forall n > n_0$

$f(n)$ → input (size)
time function

Ex:- $f(n) = n^2 + n + 3$

→ find such number, any no. greater than which satisfies the n^2

if that no. is found out

then

$f(n)$ is said to be $O(gn)$

Note

Let: $f(n) = n^3 + 1$

$g(n) = n^4$

$$0 \leq n^3 + 1 \leq n^4 \cdot c$$

$n=2$

$$0 \leq g \leq 16$$

$$\begin{cases} c=1 \\ n_0=2 \end{cases}$$

satisfy this
 $\forall n$

②

$$f(n) = n^3 + 1$$

$$g(n) = n^3$$

$$0 \leq f(n) \leq c \cdot g(n)$$

$$0 \leq (n^3 + 1) \leq c \cdot n^3$$

$$\begin{cases} c=2 \\ n_0=2 \end{cases}$$

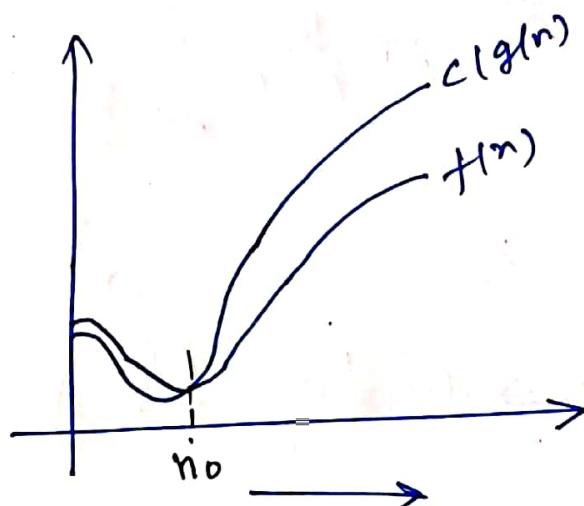
$$\Downarrow O(n^3) = O(n^4) \dots$$

Note:-

* while we are calculating big O,
if any $f(n) ; O(n^3)$ then it is
automatically $\rightarrow O(n^4)$ and also $O(n^5)$

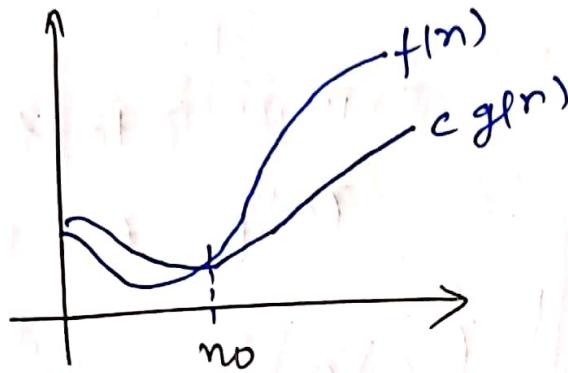
* But best answer is to take
minimum $g(n)$

↓
Repeat $O(n^3)$ here.



Big Omega (Ω)

$$0 \leq c g(n) \leq f(n) \quad \forall n \geq n_0$$



$\Omega(g(n))$

$\Rightarrow f(n)$ is said to be

Ex:-

$$f(n) = n^3 + 1$$

$$0 \leq c g(n) \leq f(n)$$

$$0 \leq 1 \leq n^3 + 1$$

$$\begin{array}{c} g(n) = 1 \\ c = 1 \\ n_0 = 1 \end{array}$$

Big theta (Θ)

→ Always report this to your answers

$$0 \leq f(n) \leq c_1 g(n)$$

—— ①

$$0 \leq c_2 g(n) \leq f(n)$$

—— ②

if $n > n_0$

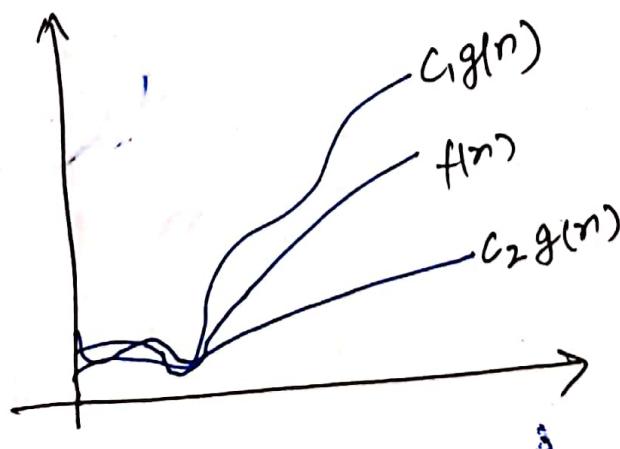
↓↓

$$c_2 g(n) \leq f(n) \leq c_1 g(n)$$

lower bound

upper bound

→ if something is Big O or Big Ω then it is also Big Θ .



Video - 4

Best, Worst, Avg Case

Algo-1

sorted array \rightarrow arr

1	5	7	9	24
---	---	---	---	----

(a)

if $a=8 \rightarrow 0$ (answer if not present)
 $a=9 \rightarrow 1$ (if present)

- This algo basically checks whether the number is present or not.
- ② → g+ will start comparing with each element of the array
- if size of array is ' n ' then this algorithm compares n -times.
(if no. is not got)

for Algo 1

1) Best Case $\rightarrow O(1)$

$$\downarrow \quad \{ T_n = k \}$$

search + getting the result
at p^+ time only.

[if $a=1$]

(2) Worst Case $\rightarrow O(n)$ $[T_n = nk]$



when last element of the array got matched

(3) Average Case or Expected Case \rightarrow

\Rightarrow sum of the run time of for the total no. of possibilities.

$$= O\left(\frac{\sum \text{all possible run time}}{\text{Total no. of poss.}}\right)$$

Example:

1	5	7	9	24
				$(n+1)$

$$\text{Total no. of poss.} = (5+1) = 6$$

\downarrow
if not found in the array

$$\therefore \text{Avg. Case} = \frac{k + 2k + 3k + \dots + nk + nk}{n+1}$$

$$= \frac{k(1+2+3+\dots+n) + n}{n+1}$$

$$= K \left[\frac{\left[\frac{n(n+1)}{2} \right] + n}{n+1} \right]$$

$$= K \left[\frac{n^2 + n + 2n}{2(n+1)} \right] \rightarrow \text{non-dominating team}$$

$$= \frac{K \cdot n(n+1)}{2(n+1)} = \left(\frac{K}{2} \right) n \rightarrow \text{constant team}$$

Avg. $\underline{\underline{= O(n)}}$

\downarrow

avg $O(n)$

Algo - 2

sorted array \Rightarrow arr

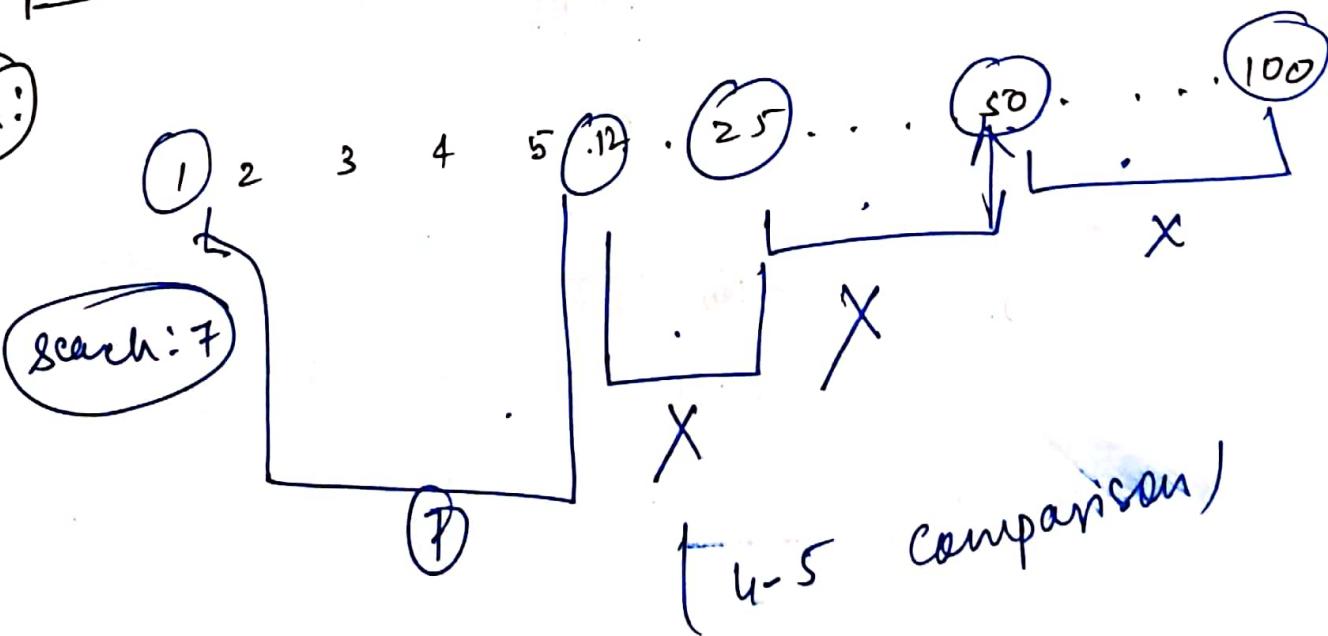
1	5	7	9	24
---	---	---	---	----

\rightarrow it'll take 1st + last element & match it first

~~go~~ \rightarrow then, it'll search for midpoint of that array & then search still if not matched, again mid point of that small array and this process continues.

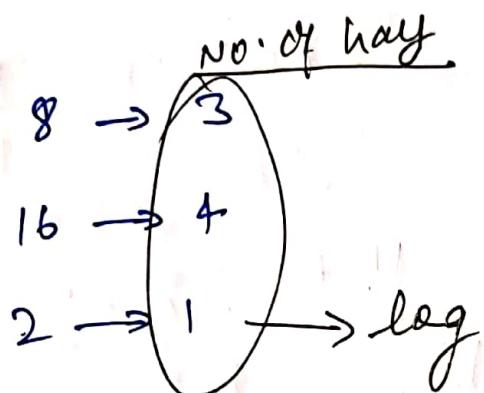
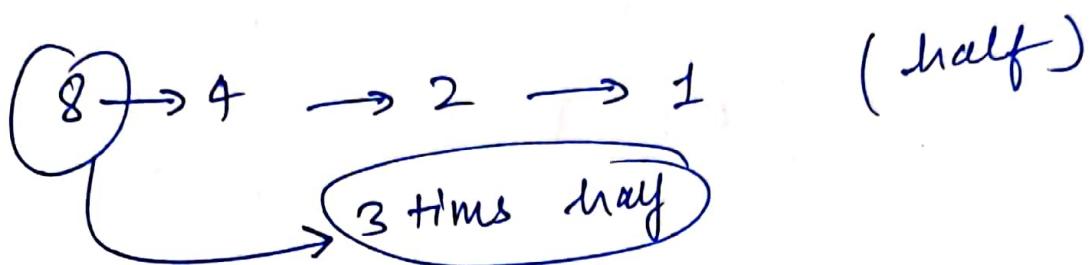
Best Case $\rightarrow O(1)$

Ex:



(4-5 comparison)

Worst Case



$$\log_2 2 = 1$$

$$\log_2 4 = 2$$

∴ Worst Case $\rightarrow \log(n)$

$$n = \text{फलों} \quad 0\overline{12} \quad 3\overline{12} \quad 4\overline{2} \quad 2\overline{4} \quad \frac{7}{21}$$

$\Rightarrow \log(100)$

key $\frac{a}{\sqrt{E}}$ $\frac{a}{\sqrt{E}}$ $\frac{a}{\sqrt{E}}$ $\frac{a}{\sqrt{E}}$ $\frac{a}{\sqrt{E}}$

Worst "1" $\frac{1}{\sqrt{E}}$ $\frac{1}{\sqrt{E}}$

Video - 5

Question Practice

Ticks

- ① Drop the Non-dominant terms

eq: $\text{int } i : \left\{ \begin{array}{l} k_1 + i \\ \text{for } (i=0; i < n; i++) \\ \quad \left\{ \begin{array}{l} R = i; \\ R++; \\ x = y + 1; \end{array} \right. \end{array} \right. \right\}$ k_2 time

② $T_n = k_2 n + k_1$

when n is bigger than k_1 is neglected

$$\therefore T_n = k_2 n$$

- ② Drop the constant term
 $\therefore k_2$ is dropped

$$\therefore T_n = O(n)$$

③ Break the code into fragments

eq: $\text{int } i$ $\rightarrow k_1 \text{ time}$ (constant time $\Theta(1)$)
 $\text{int } K=0$

$\text{for } (i=0; i < n; i++)$
 {
 $K = i$
 $K++$
 $x = y + 1$

Σ
 $\text{for } (j=0; j < n; j++)$
 {
 $K = j + 1$
 $K--$

3

$$T_n = k_1 + k_2 n + k_3 n$$

non-dominant term

$$T_n = k_2 n + k_3 n = n(k_2 + k_3)$$

$$= n k_+$$

\downarrow
 new const. \rightarrow drop the const. term

$$\therefore T_n = O(n)$$

Example :

Nested
for
loop

$\text{int } i$] $\rightarrow k_1 \text{ time}$
 $\text{int } k=0$
 $\text{for } (i=0; i < n; i++)$
 {
 $\text{for } (j=0; j < n; j++)$
 {
 $k=j$
 $k++$
 $x=y+1$] $\rightarrow k_2 \text{ time}$

3

3

$$\therefore T_n = k_1 + k_2$$

i	j	i	j	$\frac{i}{2}$	$\frac{j}{2}$
0	0	1	0	2	1
0	1	1	1	.	.
0	2	1	2	.	.
:	:	:	:	2	n
0	n	1	n		

$\Rightarrow n \text{ time for each } i \text{ (value)}$
 $i = 0 + 1 + (n + n + \dots + n-1) \rightarrow n \text{ time}$
Index

$$= n(1 + 1 + \dots + n-1)$$

$$= n \times n$$

$$= \underline{\underline{n^2}}$$

$$\therefore T_n = n^2 \times K_2$$

so,

$$T_n = O(n^2)$$

Question - Practise

(Ques 1) find time complexity of "fun1" in program

$$1 = ?$$

→ array is passing to "fun1"

fun1 → sum of array

→ product of array

In "fun1" of the program

- * $k_1 \rightarrow$ upper wall initialized term
- * $k_2 n \rightarrow$ 1st for loop. (it will go n times)
- * $k_3 n \rightarrow$ 2nd for loop (n times execute)

$$T_n = \cancel{k_1} + k_2 n + k_3 n$$

$$\Rightarrow (k_2 + k_3) n$$

$$\Rightarrow k_4 n$$

($n = \text{length of array}$)

$$\therefore T_n = O(n)$$

$$\therefore T_n = O(\text{length})$$

(Ques 2)

$K_1 \rightarrow$ upper wall

(sol)

loop \rightarrow 0 to n } nested loop
 \rightarrow 0 to n

$$[n + n + \dots + (n-1)n] K_2$$

$$T_n = K_2 n^2$$

(K_1 is neglected)

$$T_n = O(n^2)$$

recursivex problem

(Ques 3)

Avg. time $T(n)$
then $T(6) = ?$

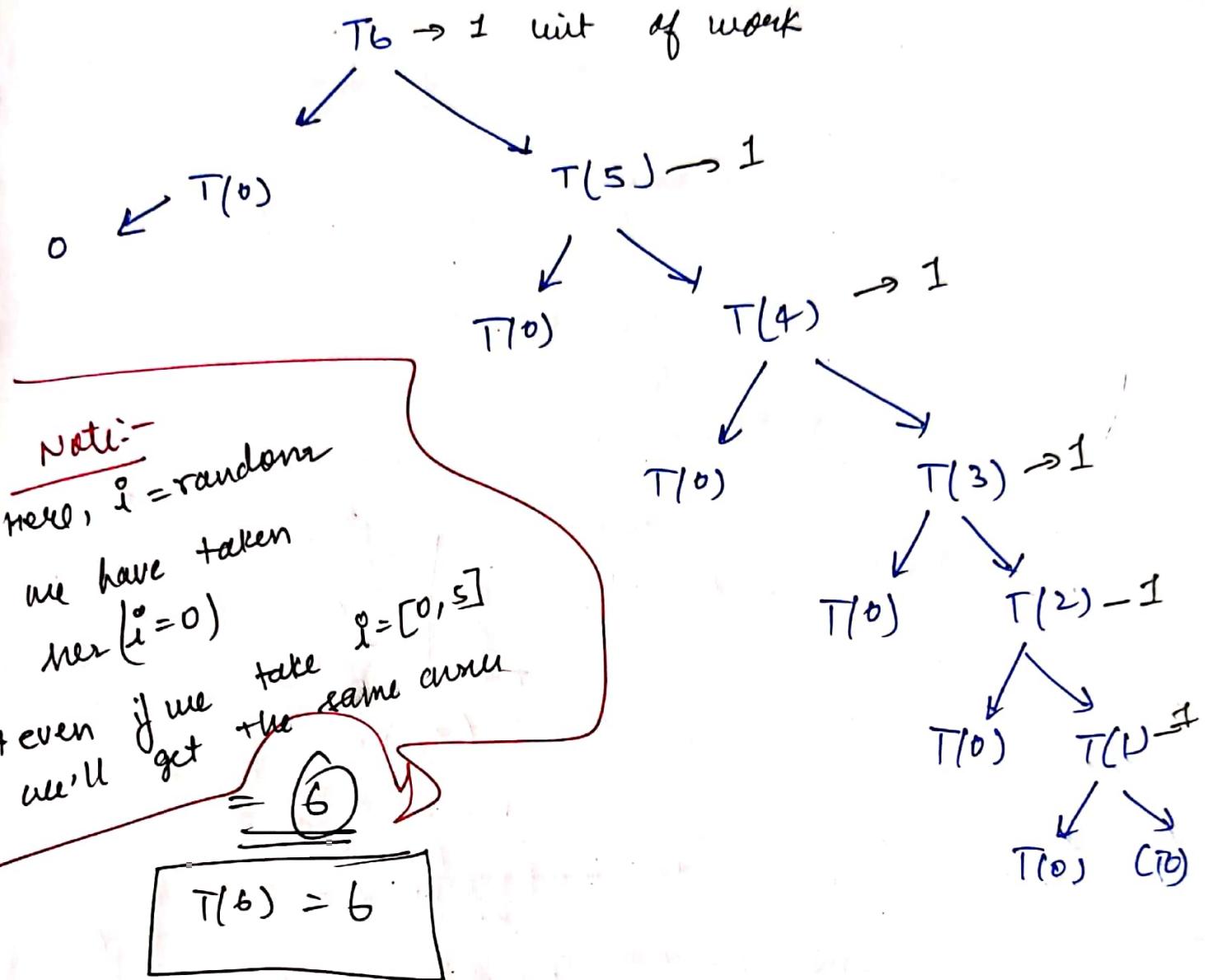
(sol)

$K_1 \rightarrow$ int i

\downarrow it takes very minute time

random(6) \rightarrow returns any integer no.

from $[0, 6]$ \rightarrow interval.



~~Ans 4~~ $\Rightarrow O(N)$

(A) $O(N+P)$ $\because P < N/g$
 $P = \text{non-dominant term}$

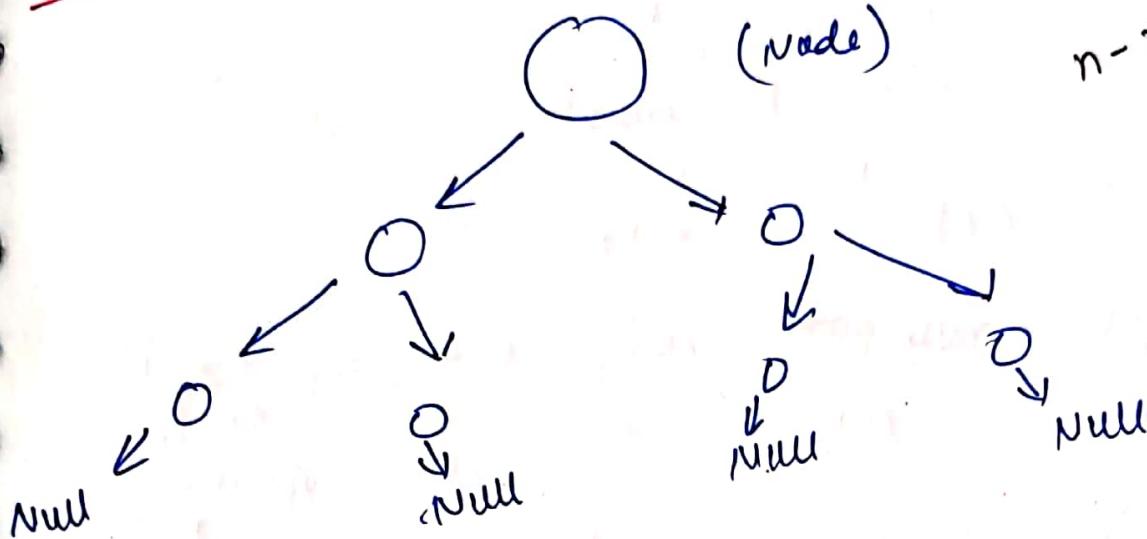
(B) $O(gN - K) = O(N)$

(C) $O(N + 8\sqrt{N}) = O(N)$
non dominant

(D) $O(N + M^2) \neq O(N)$

(Ques 5)

Binary search Tree



$$\Rightarrow n = \text{no. of nodes}$$

$$\boxed{\text{Run time} = O(n)}$$

(Ans 6)

Tests whether a no. is prime or not?

Sol?)

if直达 part $\rightarrow k_1$ time (const.)

loop part $\rightarrow k_2 n$ time

$$(i=2) \text{ to } (\cancel{i^2 = n}) \quad i^2 = n-1$$

$$i=2$$

$$i=3$$

$$\vdots$$

$$i = [\sqrt{n}]$$

approx

$(\sqrt{n} \text{ time})$

root n and greatest integer value.

$$T_n = O(\sqrt{n})$$

approximately

\sqrt{n} \Rightarrow dominant term

~~Ans⁷~~ \rightarrow But the given program does not check for prime NO.

$$\therefore T_n = k_1 \rightarrow O(1)$$

$$\therefore \boxed{T_n = O(1)}$$

Video - 6

⇒ Array &
Abstract Data Type

- ✓ Processor
- ✓ Graphic card
- ✓ Mother board
- ✓ RAM chip
- ✓ SSD

→ PC Build

Blueprint of PC

with the help of blueprint

build a P.C.

* Abstract Data Type *

- Blueprint
- tells us minimum requirements along with some operations
- Model to build a data-structure
- Data structure is the actual implementation.

Abstract Data
Type

Minimal Requirements
(MRF)

operations

data
structure

MRF : Minimal Requirement Functionality

Array as an Abstract Data Type (ADT)

Arrays

MRF

get(i) \Rightarrow can get value at i^{th} index

set(i, num) \Rightarrow can make the value of i as num

methods/operations

- insert
- delete
- add
- sort/resize

arr[2] = 67;
cout < arr[5];
 \uparrow setting value of 5th position
 \uparrow getting value at 5th position

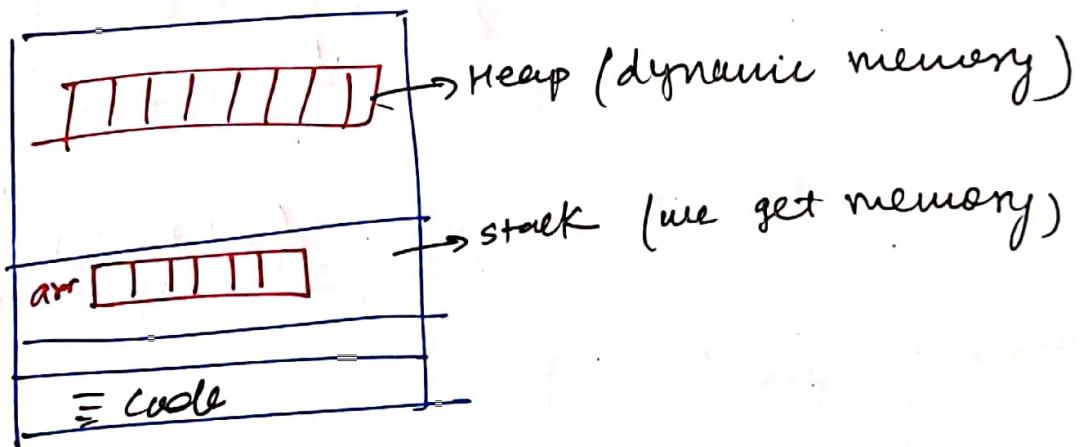
Abstraction \Rightarrow Hidding details

- \Rightarrow don't go for implementation details
- \Rightarrow only go for Usages

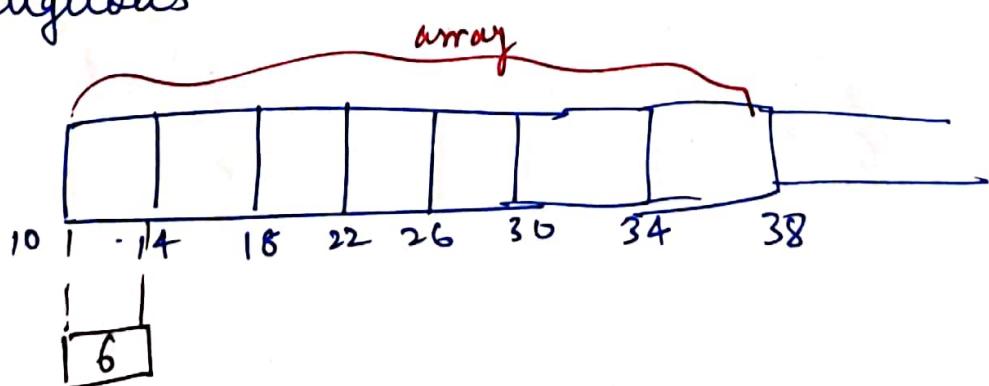
Array

\rightarrow Collection of elements accessible by an index

Memory Layout



\rightarrow contiguous blocks of memory



\rightarrow we cannot resize the array

→ after '38' → it is possible that some other application was given memory by the compiler

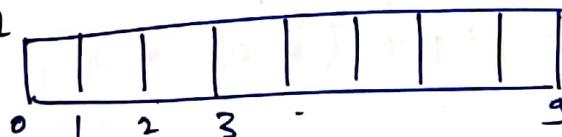
→ so, we cannot make it larger in this array

→ But we can increase the size in linklist

→ If you want to request a larger array, you'll have to copy all this content one by one.

Resizing an Array

can't do here

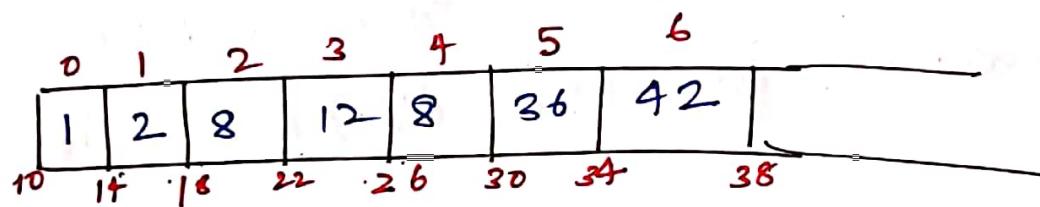


copy this content & increase its size



Why we made Array

- Bcz at one point I have these 10 addresses. Then if we ask for 4th (let say) element of this array.
- so, here I'll count and I'll know that this is at a distance 4
- so, 4th element; I'll try to find in one calculation i.e., $O(1)$ time.



$10 + 4 \times (i)$ → general address for accessing the above array

- array \Rightarrow faster access $\frac{1}{\text{fast}}$ $\frac{1}{\text{slow}}$
- you can access the element of array in $O(1)$ time (constant time)

→ so, this is benefiting a lot
we can access anything faster.

Problem with array

⇒ If I want to insert an element here in b/w array, if sufficient size is not there, then we have to make another one.
After that I'll have to keep the new element that I'm inserting.
⇒ This insertion becomes costly.
⇒ similarly, deletion is also costly in an array.

Video - 7

Array ADT

→ Abstract Data Type can be interlinked with OOPs.

ADTs

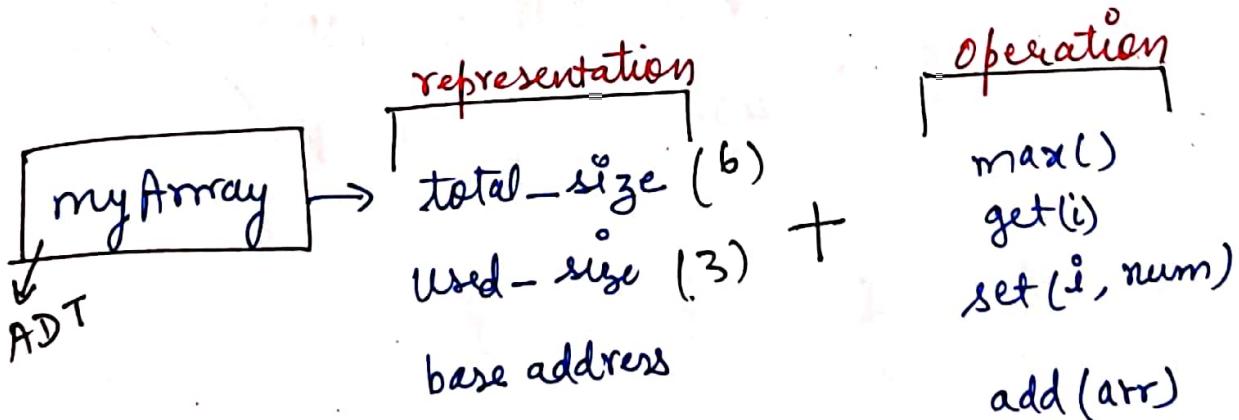
→ set of values + set of operations

int → 9, 10, 12
(integer data type)

$$9 + 12 = 21$$

operator
↳ details of calculation abstraction

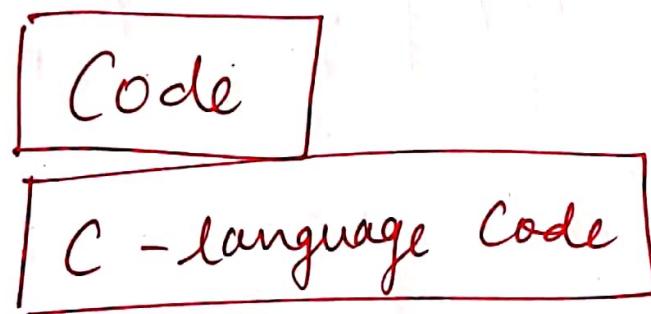
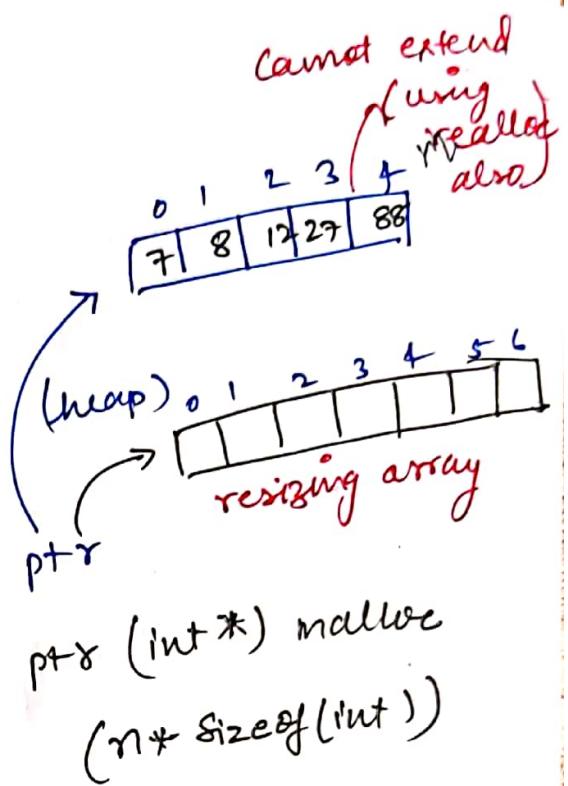
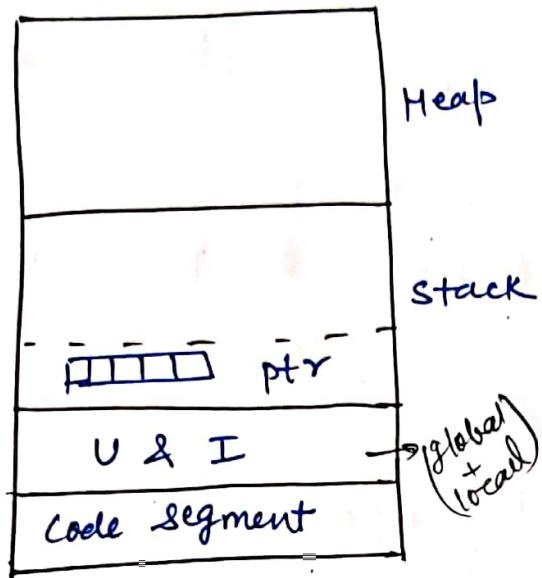
→ Computer performs this operation in bits



Eg:-

0	1	2	3	4	5
17	18	22			

for latter use



* base addresses \rightarrow we can know the upcoming addresses (31131 after)

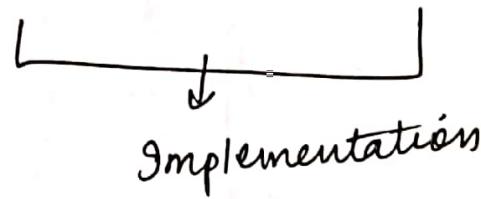
delhi

1	2	3	4	5
190	191	192	193	194 km

\therefore 3 mala \rightarrow 192 km from delhi

struct myArray {

 int total-size; → memory to be reserved
 int used-size; → memory to be used



Video - 8

vs code

coding C language

Video - 9

Array Operations

Primary Operations In Array

- ① Traversal
- ② Insertion
- ③ Deletion
- ④ Searching

Sorting = (others)

(1) Traversal

- ⇒ Visiting every element of an array
- ⇒ printing each element of an array.
- ⇒ Every element of array is visited once.

0	1	2	3	4
7	8	9	12	15

travel index

→ can be done:-

- ✓ printing array element
- ✓ setting array element
- ✓ updating array element

Eg:

int arr[100];

0	1	2	3	-	-	99
6	7	12	15			

→ 100 space memory

→ But here we have filled only 4
rest our memory is empty
we can use them in future

→ Reserved 100 space memory

→ Can used ≤ 100 space

→ array: contiguous block

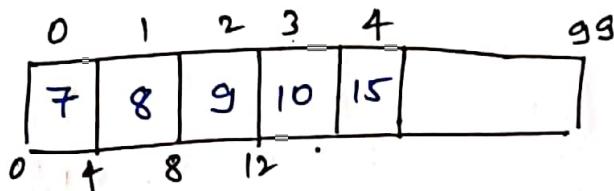


Capacity: Total it can have memory

Size: How much memory we are using

→ we can use by "for loop".

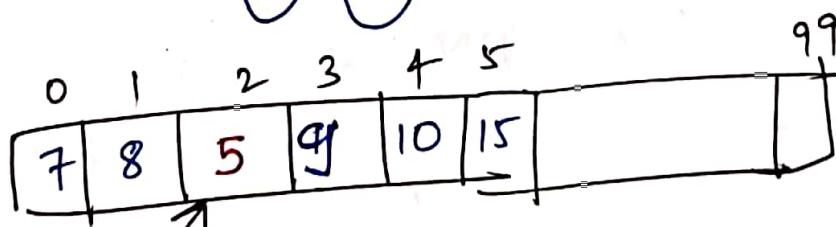
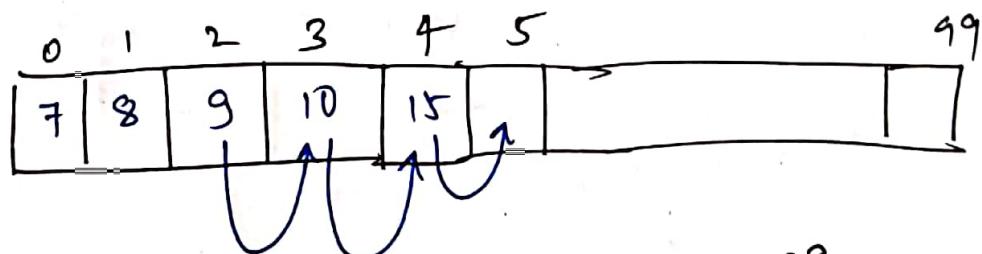
② Insertion



* Index 2 pc insert = 5 *

→ Care - I : → insert while maintaining the order of array
5 is followed by 9, 10, 15

→ we have to create gap by shifting every element



Best Case Insertion

⇒ suppose we have to insert at index 5
and array has element till index 4.
This will be the best case.

$$\Rightarrow \boxed{\text{Run-time} = O(1)}$$

Worst Case Insertion

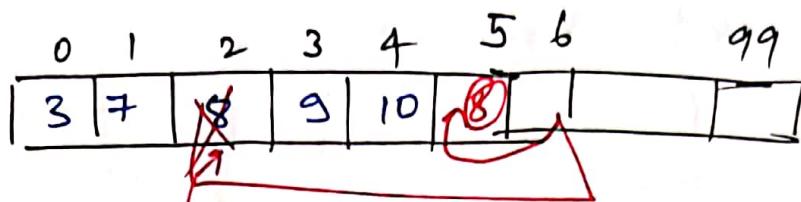
⇒ suppose we have to insert at
1st index of our array; i.e.,
shift all the elements of array

$$\Rightarrow \boxed{\text{Run-time} = O(n)}$$

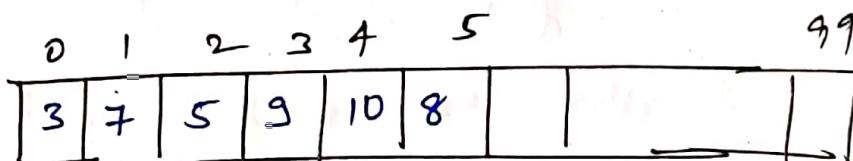
↓
 $n = \text{no. of elements present in}$
 the array

or
 $\text{length of the array}$

Case - 2: \Rightarrow don't have to maintain order



Insert 5 at index 2



* Note *

while performing insertions,
size variable we are using
Remember \downarrow to update that $(+1)$

Run Time = $O(1) = \text{constant}$

③ Deletion

→ Remove an Element from Array

Case-I :- Order of element Matters

0	1	2	3	4	5	99
3	2	5	10	12	18	1

Remove element at index '2'

New Array :-

0	1	2	3	4	99
3	5	10	12	18	1

Case-II :-

Best Case

we have to remove last element of the array

Run time = $O(1)$

worst case

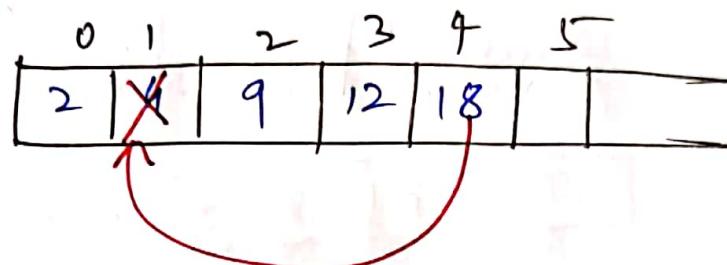
we have to remove 1st element of array

Run time = $O(n)$

n = length of array

Case-2: order of element doesn't matter

→ delete the given index & put the last element of array to its place



Remove '4' & place '18' at its place

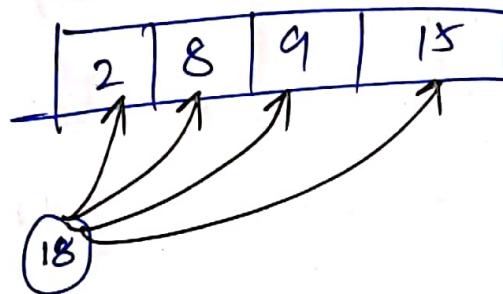
$$\text{size} = -1$$

function = $O(1)$ = constant

④ Searching

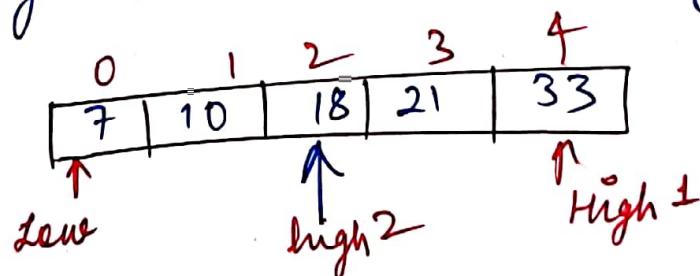
① Linear Search

- Both sorted or unsorted array
- Search & element at each index



② Binary Search

- Only for sorted array



$$\text{mid } 1 = \left[\frac{0+4}{2} \right] = 2$$

$$\text{mid } 2 = \left[\frac{0+2}{2} \right] = 1$$

⇒ converge till you get your result.

Video - 10 & 11

Coding

Deletion

~~for (i = size;~~

~~for (i = index ; i < size - 1 ; i++)~~

{

$a[i] = a[i+1]$;

$$\begin{cases} a_2 = a_3 \\ a_3 = a_4 \end{cases}$$

[index = position from
where deletion
e.g. 311]

Video - 12

Linear & Binary Search

Array	0	1	2	3	4	5	6	7
	4	8	10	12	15	16	2	8

* Search element "2" in this ~~video~~ array

→ search all the indices one by one

→ '2' will be found at index 6

→ But if 2 not found → not present in this array

when we traverse at the end of the array

Linear Search

→ works for both
(sorted + Unsorted) Array

→ done through Array Traversal

(visiting all the elements)

→ find all the elements of array one by one.

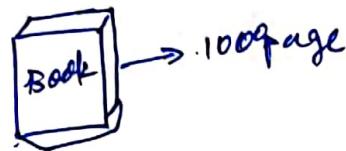
when we got our element → element found

But we don't get at the end of array → element not found

Binary Search

→ slightly smarter algorithm

→ Ex:



open → 2³⁸ page No.

→ condition: Array must be sorted

0	1	2	3	4	5	6	7	8
2	8	14	32	66	100	194	200	400

* search 200 in this array *

low mid new mid high

$$\text{mid} = \left\lceil \frac{\text{low} + \text{high}}{2} \right\rceil = \left\lceil \frac{0+8}{2} \right\rceil = 4$$

greatest integer

low	Mid	high	Element
0	4	8	Not found
4	6	8	Not found
6	7	8	found

200 → Mid & High
7 is set

Video - 13

Linked List

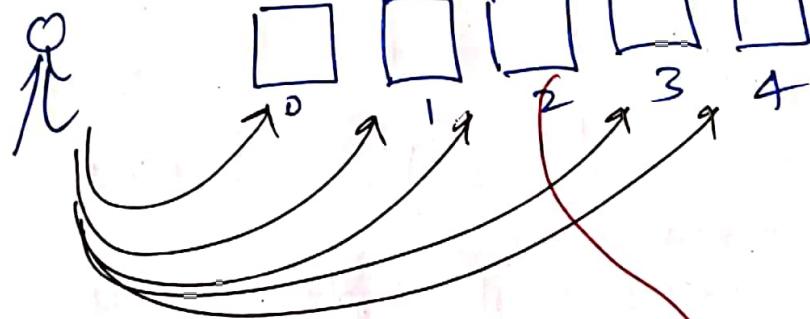
Linked List

(full body c (setup)
F.B.C
Dr.

H₁ - Hospital

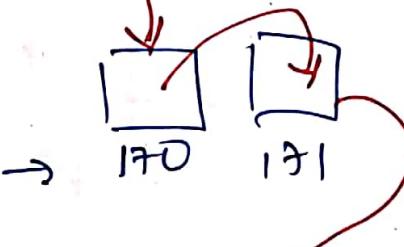
Delhi - hospital

... 100 beds



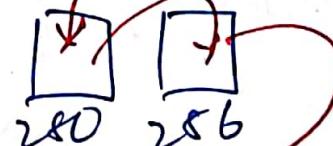
H₂ - Hospital

Bed empty



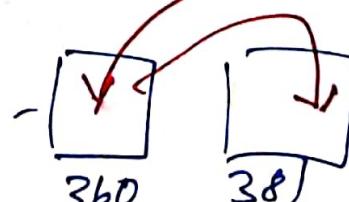
H₃ - hospital

Bed empty



H₄ - hospital

Bed empty



patient chain

(each patient has record of where

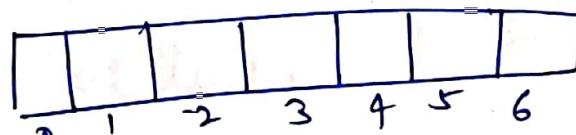
is the next patient)

→ Although beds were not there ^{is} in 1 hospital but we did was ⁱⁿ (same place) every hospital each single bed was utilized.

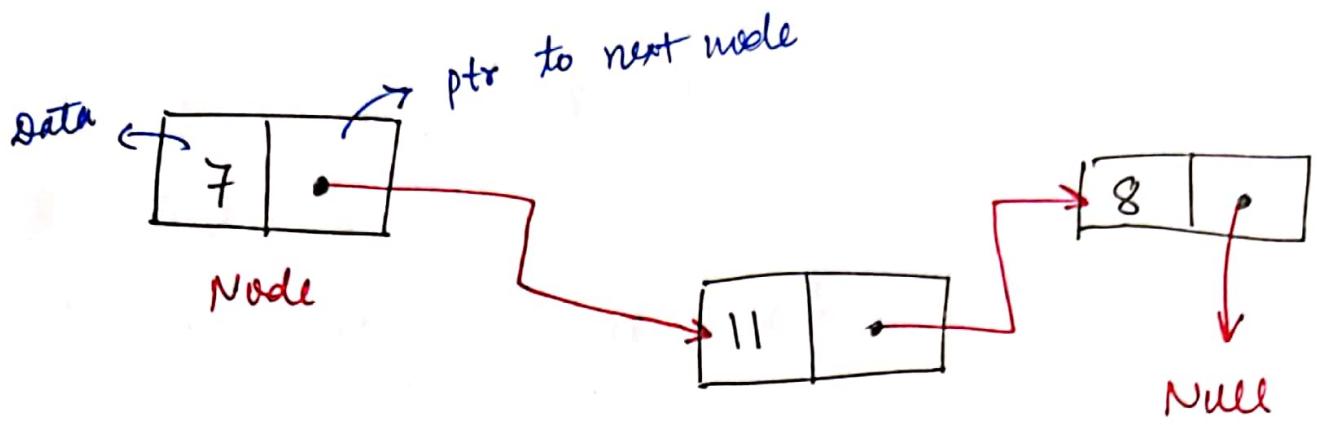
linked list

→ Replacement to array

→ Array has limited capacity

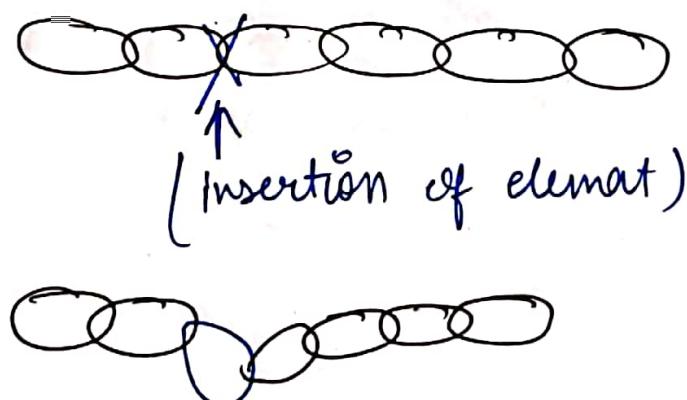


if we want to enlarge this array, we can increase the capacity to elements → we have to create a new array & copy in it even if you want an extra element



→ array element → contiguous
 ↓
 to access its elements

→ In linked-list
 ⇒ Insertion is easy

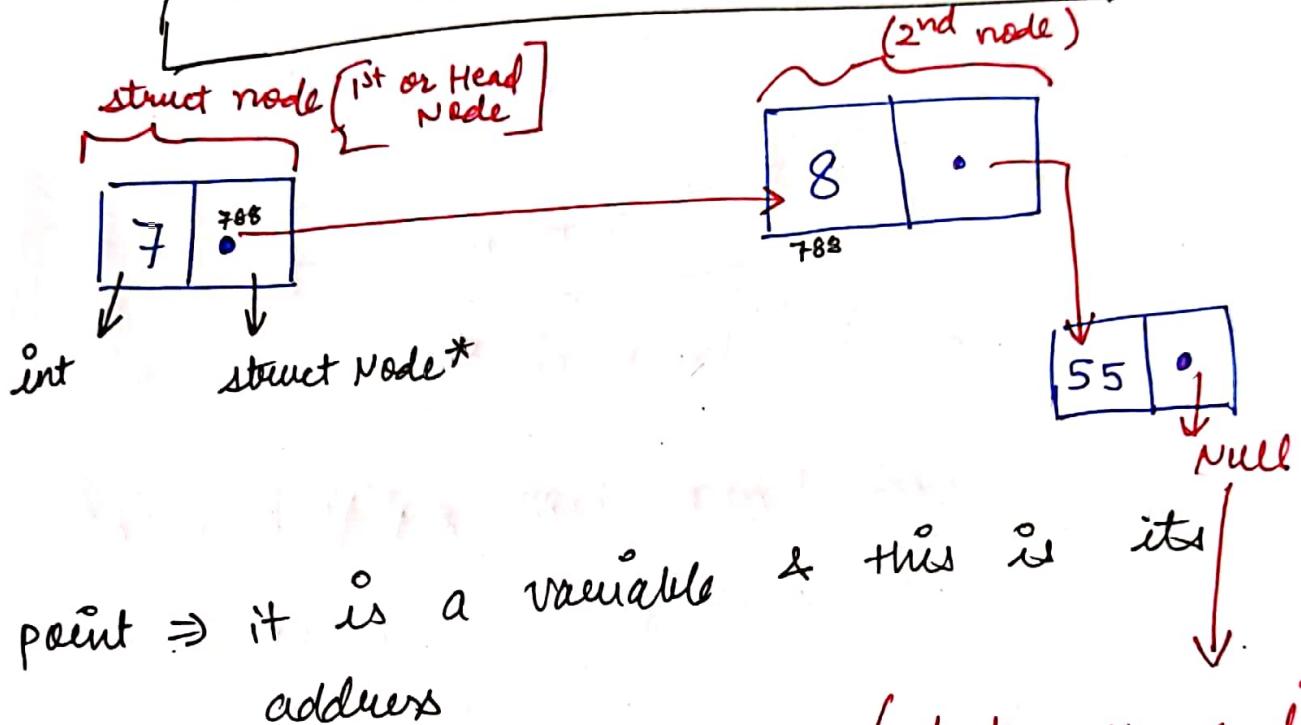


→ Similarly, Deletion is also very easy in linked list.

- Here, we just need to inter-connect two
- But array doesn't allow this facilities
↓
Array Restriction : {elements stored in contiguous memory location}
- In linkedlist, no contains of contiguous memory location.
- Access is not easy in linked-list
- point ~~an~~ : It means in this the address is kept stored

Video - 14

Linked List Creation + Traversal



+ point \Rightarrow it is a variable & this is its address

(declare the end of the list)

head = (struct Node*) malloc (sizeof (struct Node))

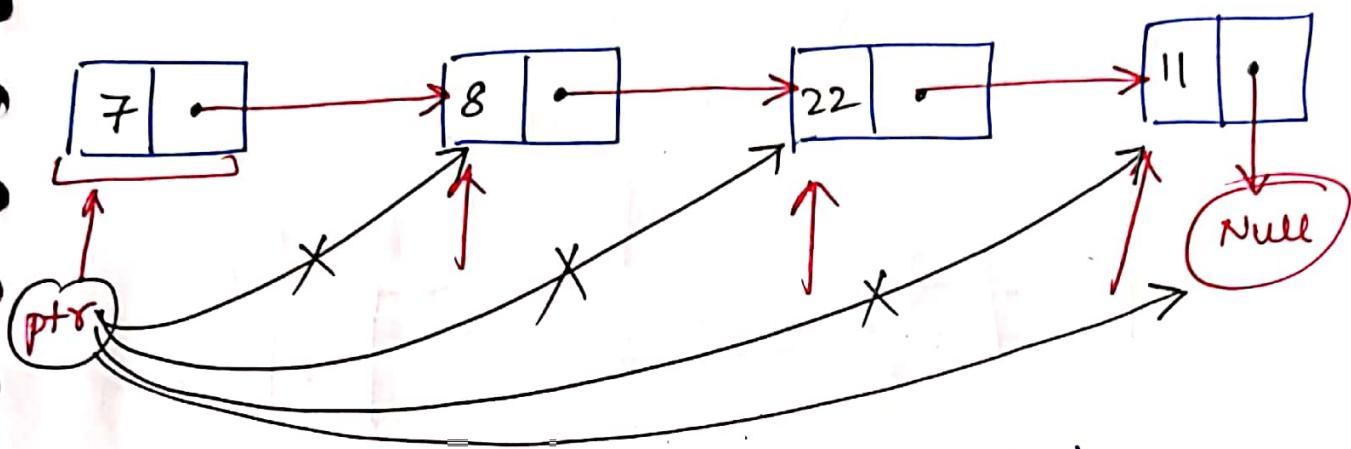
(struct pointer)
to access the data

or

head → data = 7
head → next = second Node

Structural pointer

Traversal

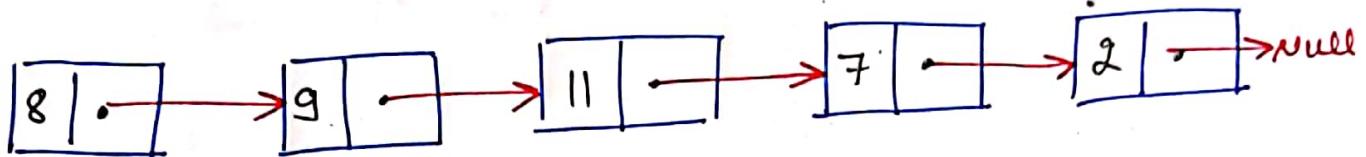


Runtime of Traversal = $O(n)$

[visiting each node one]

Video - 15

Insertion in Linked-list



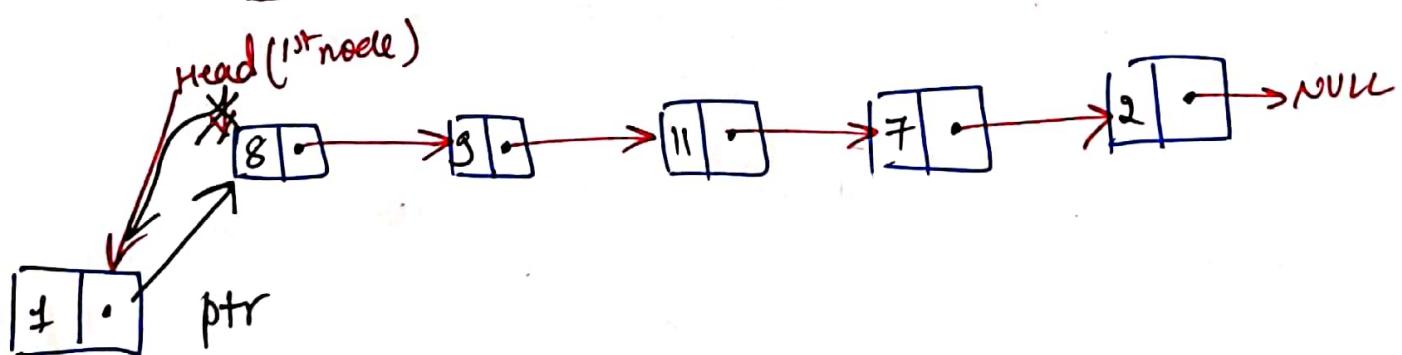
Case I: Insert at the beginning

Case II: Insert in between

Case III: Insert at the end

Case IV: Insert after a node

Case - I

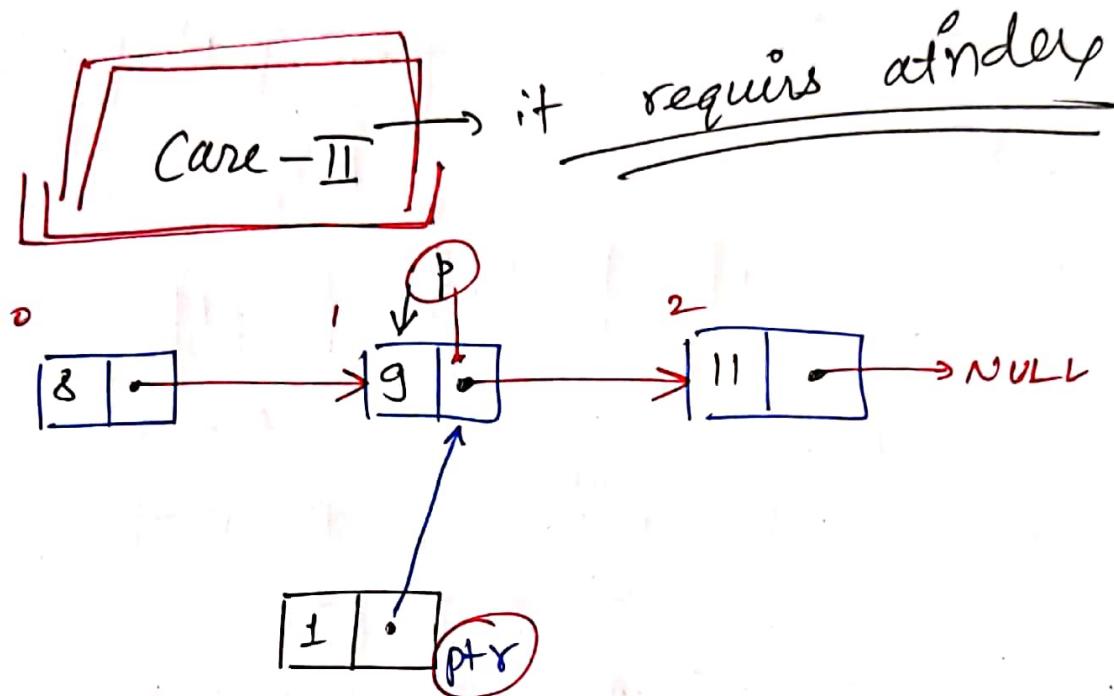


struct node * ptr = (struct node *) malloc (sizeof (struct node))

ptr → next = head; head = ptr;
 return head;

* ptr → stores the address *

Run Time (complexity) = $O(1)$ (constant)



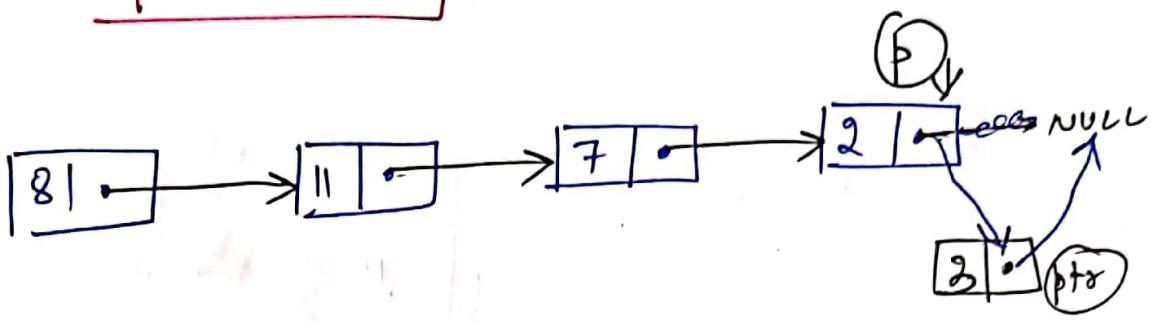
$ptr \rightarrow next = p \rightarrow next;$

$p \rightarrow next = ptr;$

Time Complexity = $O(n)$ → worst case

Best case Time Complexity = constant.

Case - III



$$p \rightarrow \text{next} = \text{ptr}$$

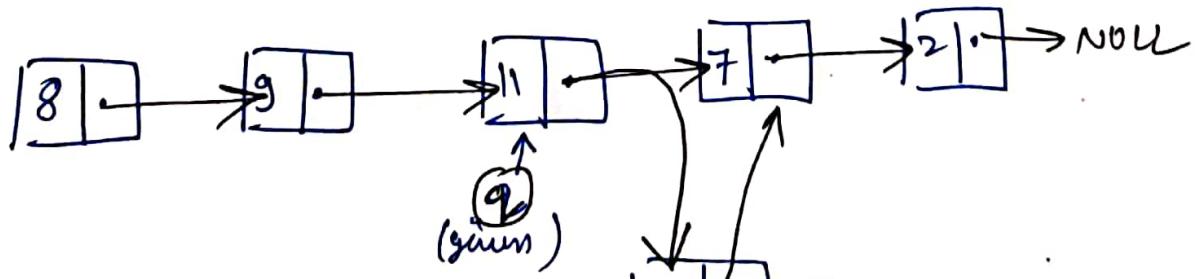
$$\text{ptr} \rightarrow \text{next} = \text{NULL};$$

Time Worst Complexity = $O(n)$

Case - IV

→ Time Complexity = $O(1)$
= Constant time

- * Insert after a Node whose address is known/given in the question



$$\text{ptr} \rightarrow \text{next} = q \rightarrow \text{next}$$

$$q \rightarrow \text{next} = \text{ptr}$$

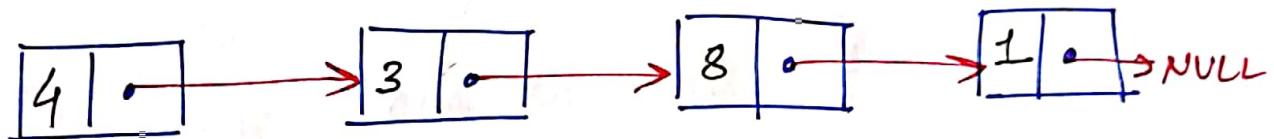
Video - 16



Coding

Video - 17

Deletion in linked list

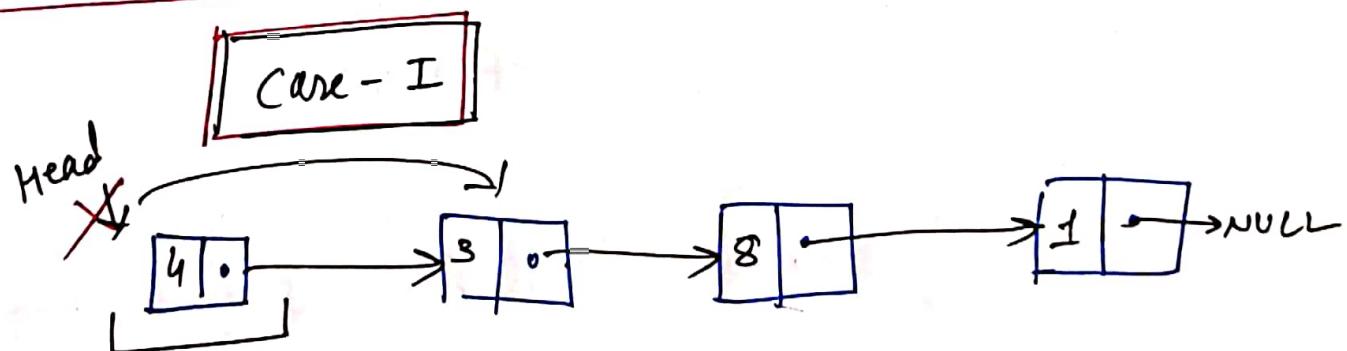


Case 1: Deleting 1st Node

Case 2: Deleting a node in between

Case 3: Deleting the last node

Case 4: Delete a node



→ forgot 1st node and point head to
next node & broke this link

→ memory of 1st node which is stored

in my heap what will happen to
that? \Downarrow

when you delete node of linked list
will you leave this memory on-hang?
 \Downarrow

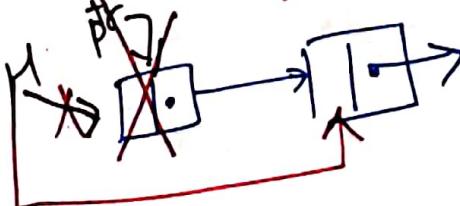
No, we have to free it.
It is a dynamic memory
We have allocate this memory dynamically
→ so, we need to free this.

```
struct * Node * ptr = head ;  
head = head → next ;  
free (ptr) ;
```

→ Run time = $O(1)$ = Constant

Computation

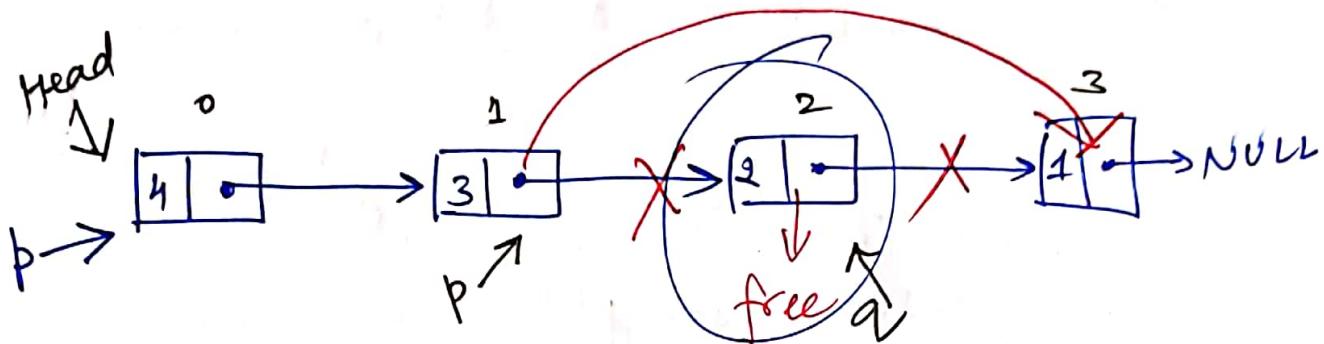
Complexity



Case - II

Let,

→ index is given ($\text{index} = 2$)



$\text{ind} = 2$;

`struct Node* p = head;`

`while (index - 1)`

{

`p = p->next;`

}

`struct Node* q = p->next;`

`p->next = q->next;`

`free(q);`

Note:-

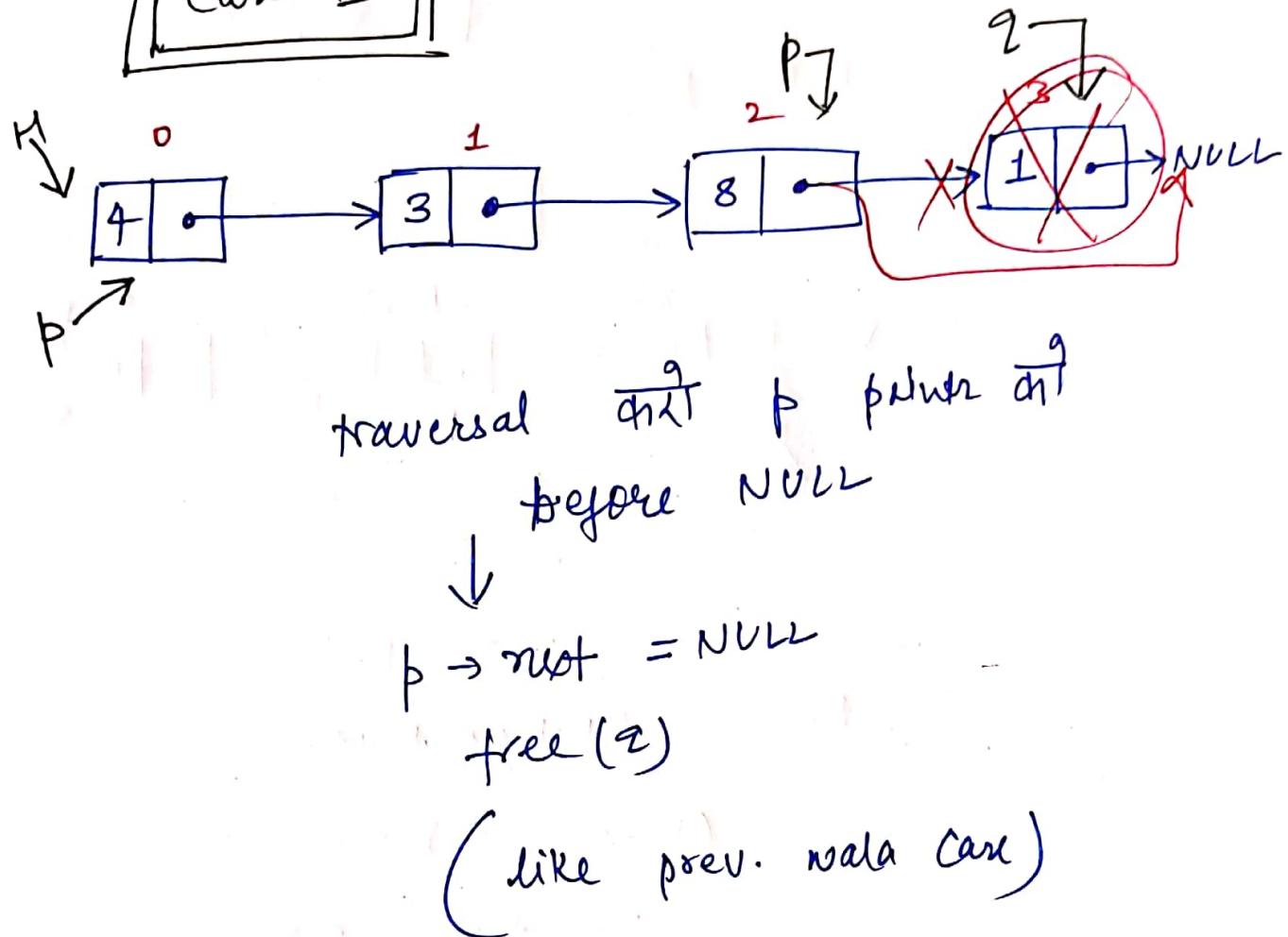
`p->points → to head` to pt. $\overline{\text{q}}$

`p at traversal` $\overline{\text{for i=1}}$ $(\text{index}-1)$ $\overline{\text{N}}$

`p at next = q->next`

`q = p at next wala`
`free(q)`

Case - III



Case - IV

\rightarrow 1st node with that value
of Node
(if any element present 2-times)

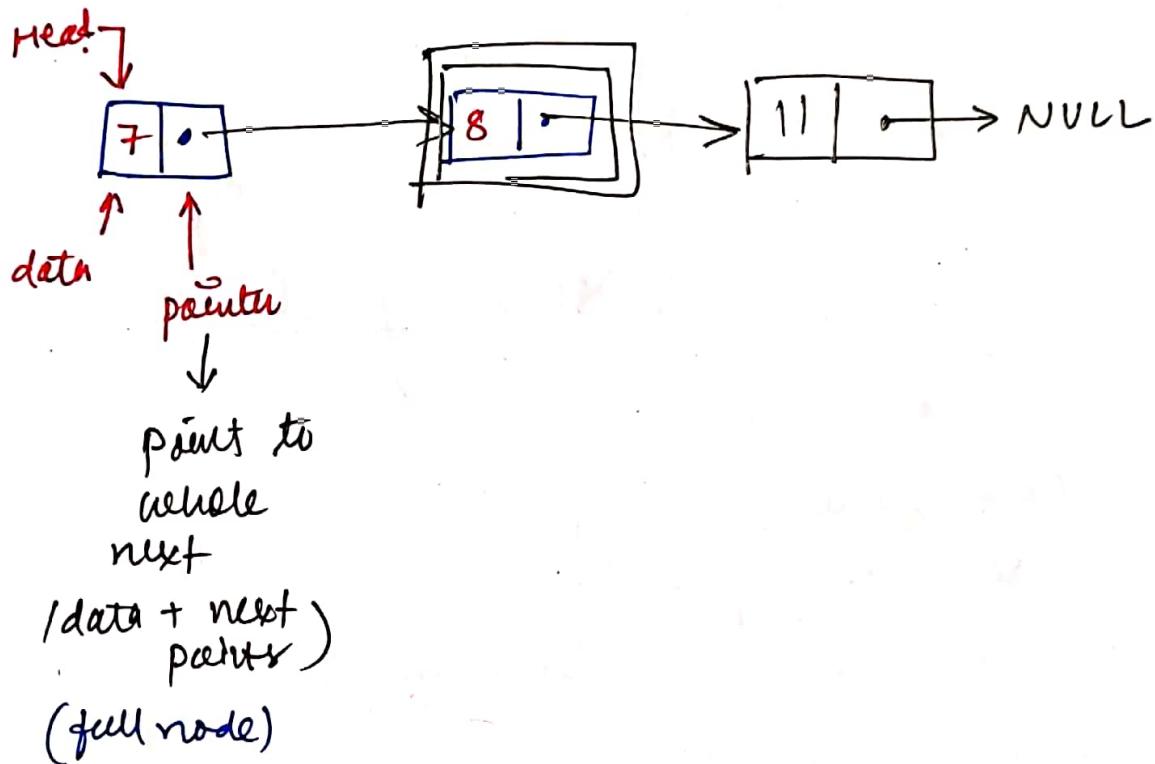
(procedure similar to above)

Video - 18

↳ Coding in V-S. Code

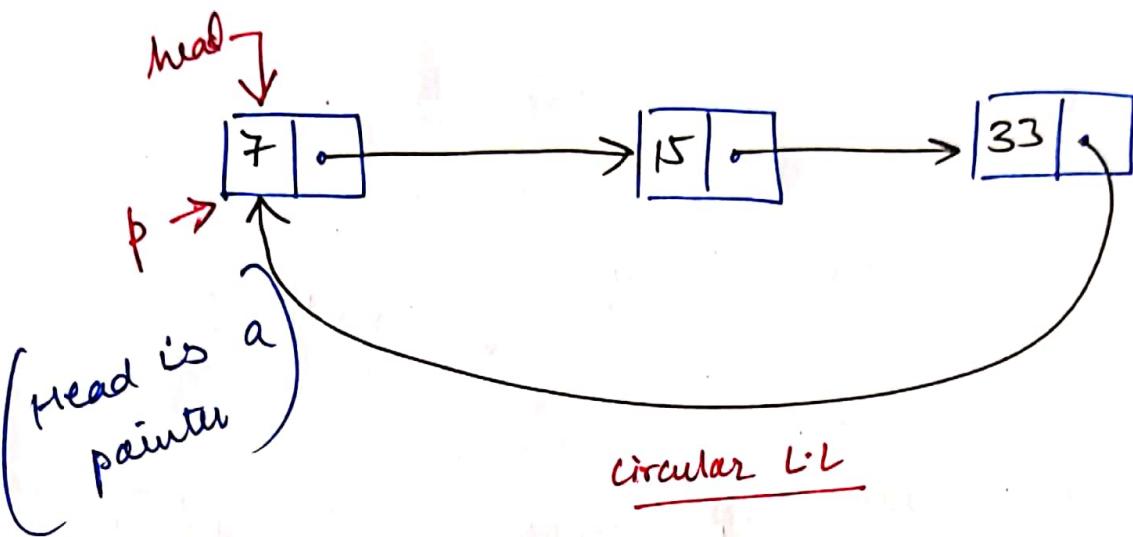
Video - 19

Circular Linked List

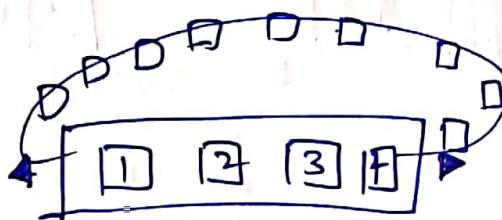


Circular L-L

where LL was pointing to null now what will happen , it will not point to null : it will come & point this 1st one .



Ex:-



Traversal

~~print(head → data)~~

```
while(p->next != head);
    print(p->data);
```

$p = p \rightarrow next$

}

$print(p \rightarrow data)$

or [use do... while]
↓ we have to print even last node
of our L.L.

Note:
we use 'p' as
any notation not
head so that
we should not
lost our original
L.L.

```
do {  
    point(p->data);  
    p = p->next;  
}  
while(p != head)
```

Searching Element in Circular L-L

- # If u want to search an element,
see search is same as traversal
you just return it when you
find it.
- that this is the pointer of that
data & here is the data.
- # search is done through traversal

- ** Insertion & Deletion of an Element
in Circular L-L is similar to
that of L-L (simply L-L)

→ when you do insertion sometimes it doesn't matter where our head is, because this is circular linked list.

It doesn't have any 1st element. It only have 1 head which helps you in getting started.

How to make a Circular L-L which is Empty

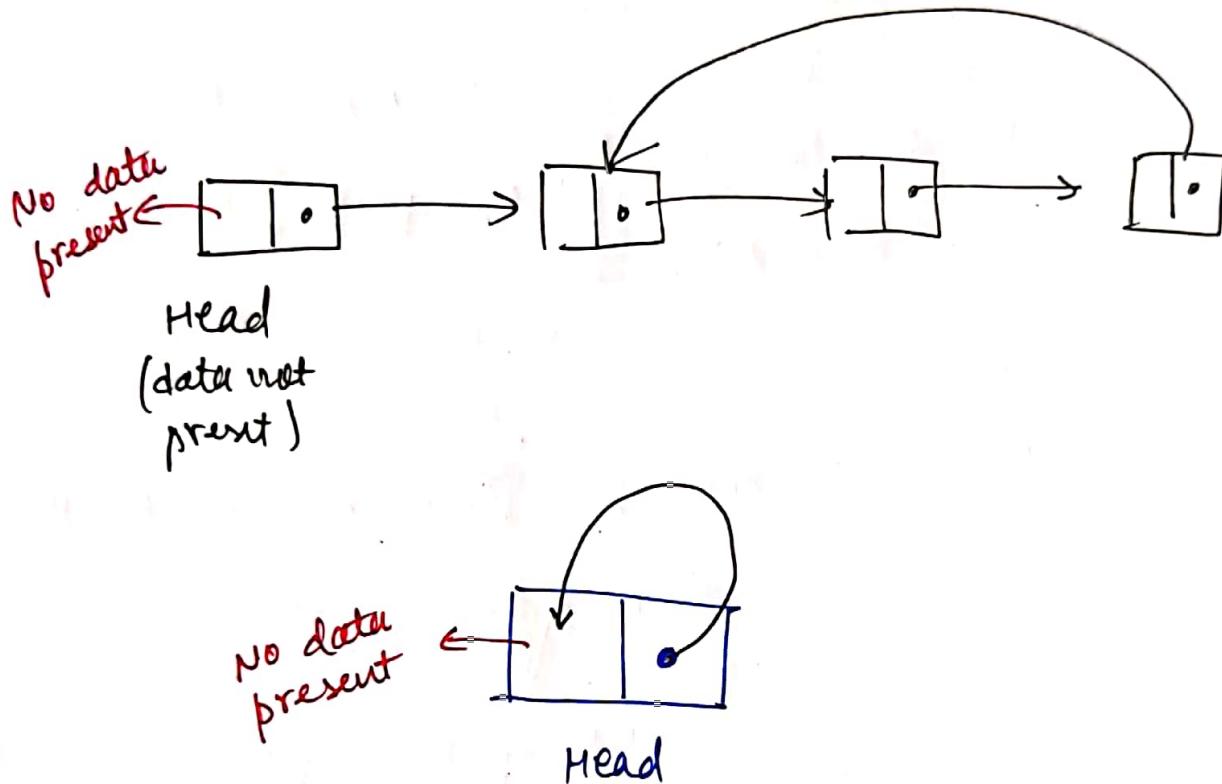
① Empty singly L-L

$p \rightarrow \text{NULL}$

② Circular Empty L-L

⇒ create a node, there is pointer & your real L-L starts from here and this is only 1 head one, where you can not store data in this.

⇒ In this you can point this only with another node & point that with next node and finally connect it to 1st one.

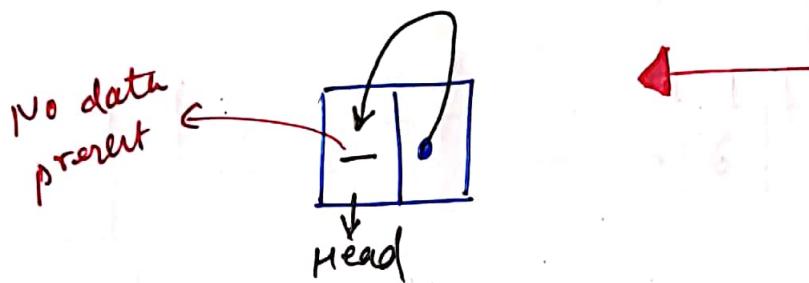


⇒ Data is not stored. You can't add data here you will never access this, bcz this was created to only access a circular L.L.; this is a head node.

Start from here (head node) and do head next you can go move in linked list

⇒ If u want to tell this linked list empty then you point its pointer to itself so this is one circular empty LL will be created.

→ (Alternative representation)

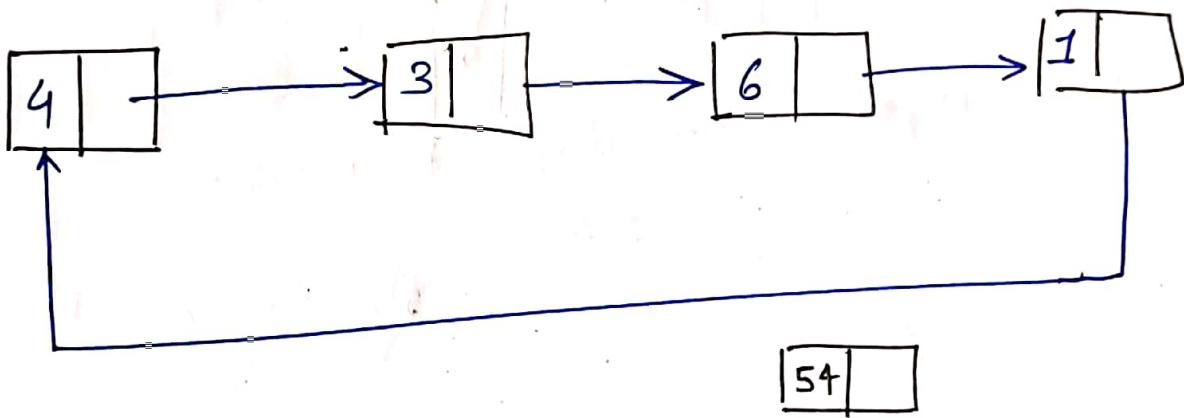


* Head is a node

Video - 20

Coding in V.S. Code

Circular LL



Insertion of an Element in Circular LL

TRAVERSAL

⇒ 1 thing will get changes other as
same as simply LL

⇒ 4th node will point to the head of 1st node

Note:-
when the traversal will happen here,
keep moving forward & forward till
you reach NULL.

But in this case of Circular LL , we can never reach NULL .
NULL can never be found .

=> Construct a pointer (ptr)



ptr → head



move ptr jab tak wasas head
na aa jaye



if we use while loop

while ($\text{ptr} \neq \text{head}$)

it will point nothing



Bcz \rightarrow ptr already = head in the beginning



so, we need to point 1st time

manually

But + will be left

so use do... while loop

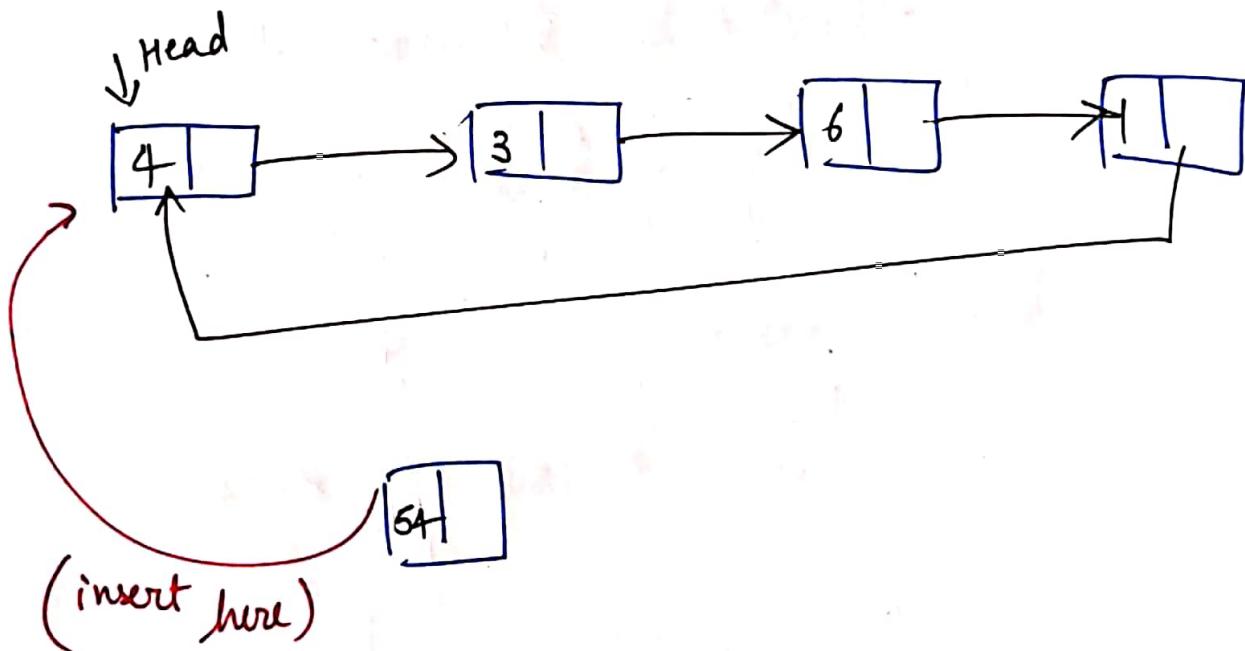
```

point (ptr->data);
ptr = ptr->next;
d0
{
    ptr = ptr->next
}
while (ptr != head);

```

Insertion in Circular linked list

#> Insertion at 1st (head position) node



\Rightarrow move it until p's next = head

$p \rightarrow \text{next} = \text{head}$

\rightarrow dynamically created a ptr & allocated memory, requested memory in the heap for one node

$\text{ptr} \rightarrow \text{points}$ $\frac{\text{data}}{\text{next}}$ $\frac{\text{data}}{\text{next}}$ $\frac{\text{data}}{\text{next}}$ $\frac{\text{data}}{\text{next}}$ insert

$p \rightarrow \text{point}$ $\text{head} \rightarrow \text{next}$

$\text{ptr} \rightarrow \text{data} = \text{data}$

move until: $p \rightarrow \text{next} = \text{head}$

while ($p \rightarrow \text{next} \neq \text{head}$)

{

$p = p \rightarrow \text{next};$

}

// at this pt. p points to the last node of this circular L.L.

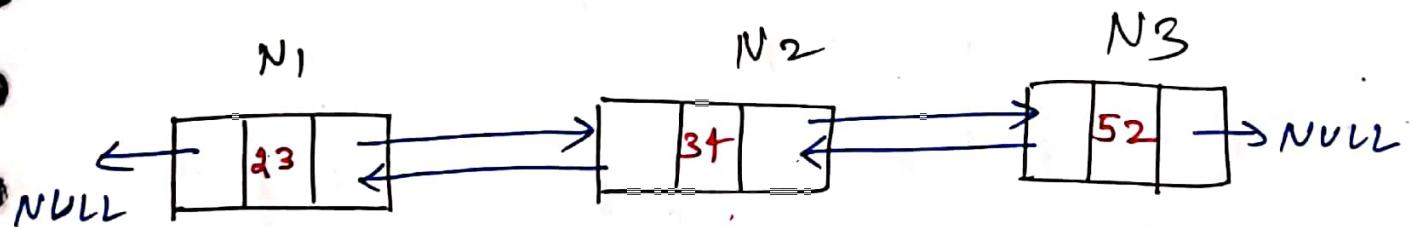
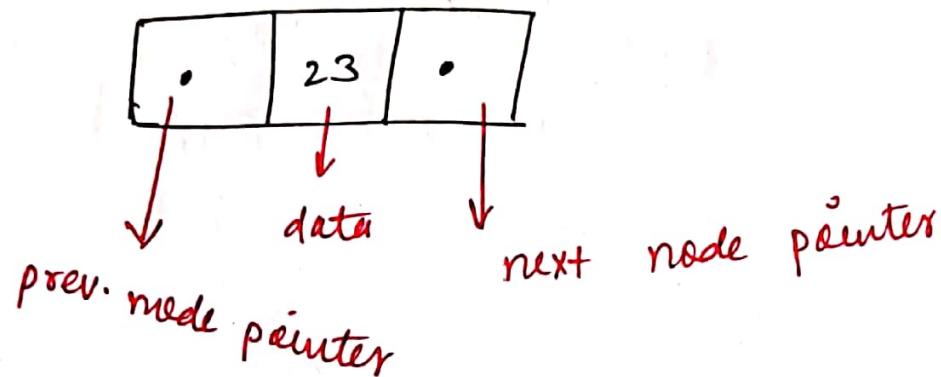
$p \rightarrow \text{next} = \text{ptr};$

$\text{ptr} \rightarrow \text{next} = \text{head};$
 $\text{head} = \text{ptr};$
 $\text{return head};$

↓
Insert 3 at 1st place & head
is also changed

Video - 21 #

Doubly Linked List



if you are anywhere in the node,
then you can ever go backwards
or forward .

Previously, we can only move forward
our pointer

- we can reverse it easily by swapping the pointers
- At the same time, if you have a pointer to the last node then you can walk in both this direction & in this direction.

→ 1st Node (Head node)

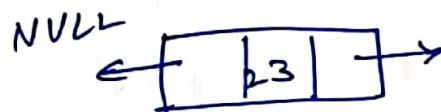
↓
prev. : NULL

If I have a pointer & want to find out whether it is a head point or not. I can find out by accessing its previous directly whether that head node belongs to the linked list or not.

Similarly,
If I get a pointer to a point with the last one then I can find out its next whether it is NULL or not.

* Amicus shows whole Node
(data + pointer)

* Head
pointing to whole node.



Head . data = 23

Head . prev = NULL

Head . next =
(next will full node)

A hand-drawn diagram of a node structure, similar to the one above, containing the value '34'. It has arrows pointing from its left and right edges towards the text '(next will full node)' to its right, indicating that it is the last node in the list.

Code :-

Struct Node

```
{  
    int data ;  
    Struct Node * prev ;  
    Struct Node * next ;  
}
```

* Extra will give benefit :
we can go forward + backward

→ biggest con of doubly LL



I'm taking extra spaces



So, at the cost of this extra space,
I'm getting traversal on both
sides.

⇒ we use doubly LL when we have
to go both ($f+b$)
otherwise use singly LL.

Code

// Creating nodes :-

// Create struct Node * N₁ = (struct Node *) malloc (sizeof {

" " N₂

" " N₂

// Link nodes :

N₁ → next = N₂ ;

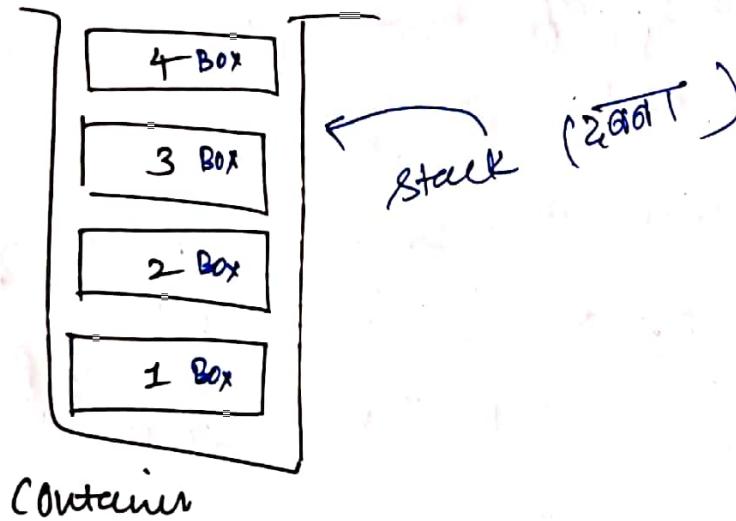
N₁ → prev = NULL ;]

similarly with all

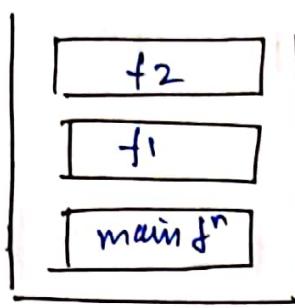
Video - 22

Stack in Data Structure

- Stack is a linear data structure.
- Operations in stack performed in LIFO [Last in First Out] Order.



- 4th Box last me enter hua Container me, wo sales ^a 400 T 31/IV/II Container se
- Also called : LIFO & FILO
↓
(first in last out)



→ Inside stack

\downarrow
main f^n (calls f_1)

\downarrow
other f^n (f_1) → (calls f_2)

\downarrow
other f^n (f_2)

⇒ finally as that f^n will be returned,
will remove it from stack.

Similarly, as the f^n will be returning
will keep on destroying their activation
records.

And finally when our program will
end, all the stacks we have will
become empty (will be empty).

stack will be empty.

→ Stack is useful in designing such
system.

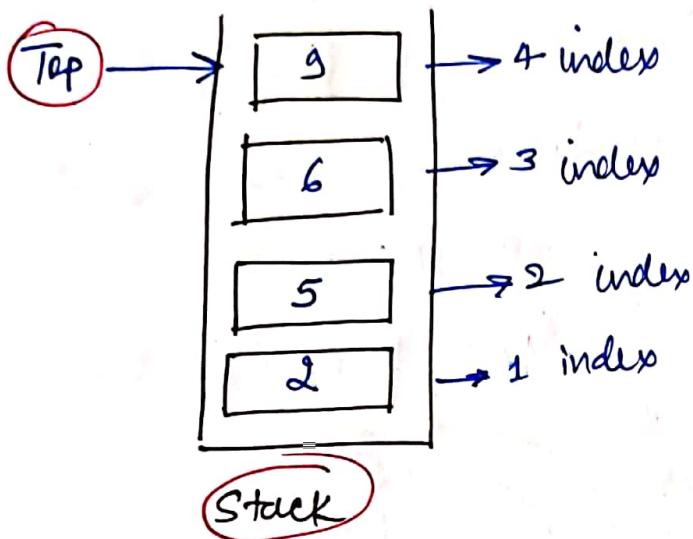
Parenthesis Matching

- we need to match parenthesis in an expression
- If one mathematical exp. is written in which parenthesis are used . so is that mathematical exp valid or not ?
Parenthesis in it are matched properly or not ?

Stack ADT (Abstract Data Type)

- with some methods & some data representation → I want to represent as an ADT .
- I want a pointer on the top most element in stack
- we can implement stack
 - Array
 - or
 - Linked List

→ stack is like a container.



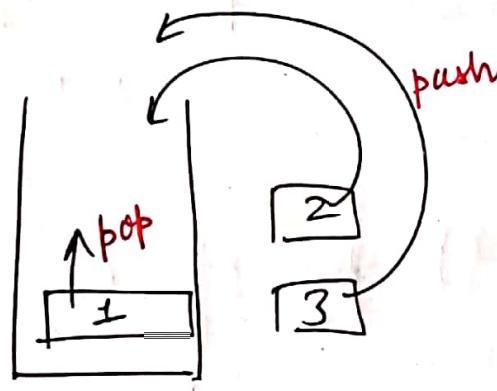
[index starts from
bottom of stack]

→ we have to specify while "push" or
"pop" which element we are doing
or dealing in the stack

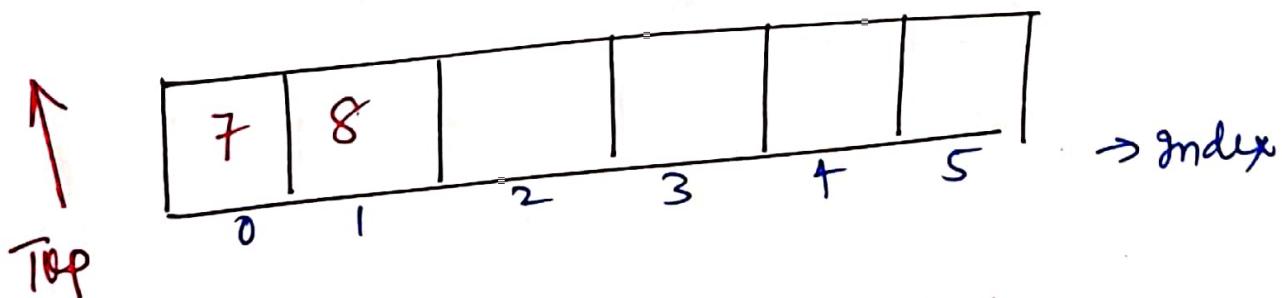
Video-23

Implementing Stack Using Array in DS

→ Stack is collection of elements following LIFO. Elements can be inserted or removed only from 1 end.



→ It is not compulsory that size of my array = no. of elements in stack



- * Stack is not array
- * Topmost element which will point somewhere

* default value of $\text{Top} = -1$
(if there are no elements)

* $\text{Top} = 1$ (if 2 elements)

* $\text{Top} = -1$ (no element in stack)

Implementing Stack Using Array

① fixed array size creation

② Top element (to store top element in stack)

struct stack

{

int size; // max. size of array

int top; // to store top value in stack

int *arr; // creating array in heap

}

struct stack s;

s.size = 80;

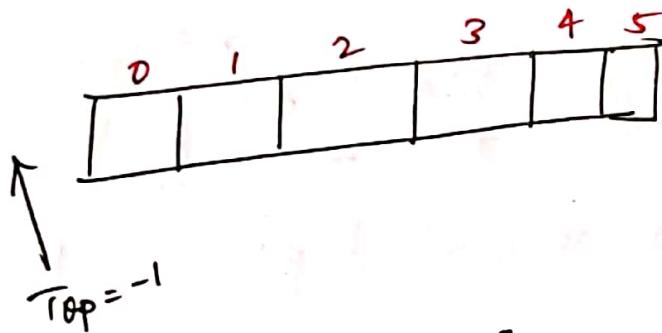
s.top = -1;

~~s.arr = (int *) malloc (s.size * sizeof(int))~~

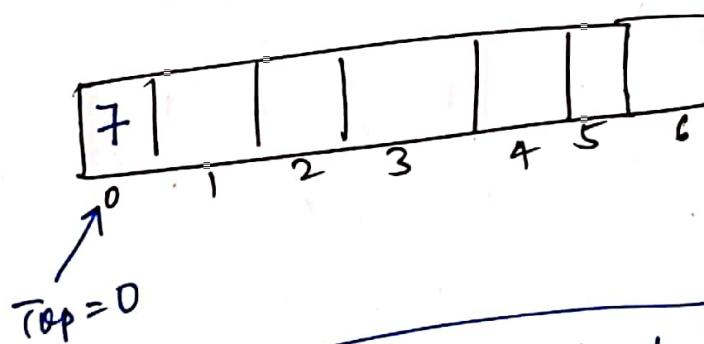
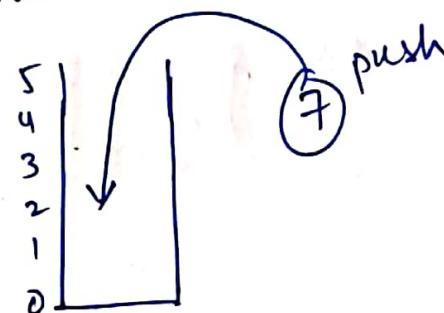
s.arr = (int *) malloc (s.size * sizeof(int));

→ Integer is used as a data type in
above example.

→ we can also create custom data-type



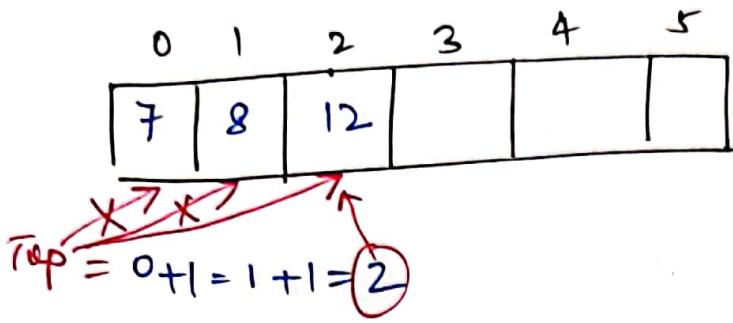
push 7 in the above array?



Run Time = $O(1) = \text{Const.}$

In most of the operations

→ **Top** → is denoting till where your
stack is filled



~~return s. arr [top];~~

~~-top =~~

return s. arr [s. top];

s. top --;

Video - 24

* Coding V.S. Code *

→ for learning purpose, we read
only Stack of Integers

struct stack *s;

I'll make a stack pointer

→ I can pass function to it by this
stack pointer

→ I can do call by reference very
easily.

→ Can Change the structure

→ *s → s is a pointer now

{ before s → stack }
Ab s → pointer }

→ now it became a pointer to
the stack structure

in program

→ arr → size = 80

top = -1 (no element present)

Note :-

* Before you remove an element from stack, check whether your stack is empty or not.

* Before you put an element in your stack, check whether your stack is full or not.

Condition for stack to Empty

→ Check whether stack is Empty or not

top = -1

then

stack is empty

else

stack is not empty

Condition for stack to be full

if ($\text{top} = \text{size} - 1$)

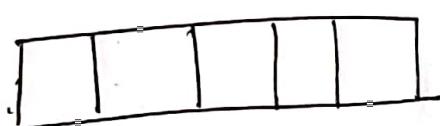


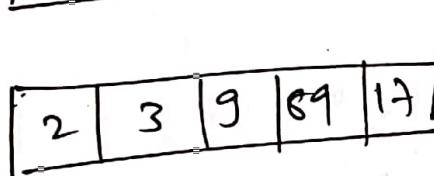
stack is full

use ptr (point until checking condition)

Video - 25

Stack Operations while Implementing Using Arrays

 → cannot Pop if Empty

 → cannot Push if full

struct stack

```
{  
    int size;  
    int top;  
    int *arr;  
}
```

① way
struct stack s

```
{  
    s.size;  
    s.top;  
    s.arr;  
}
```

② way

(by making structure pointer)

```
struct stack *s
```

```
{  
    s->size;  
    s->top;  
    s->arr;  
}
```

Use of Arrow Operator

- do reference then use dot operator
- if do dereferencing then use dot operator.

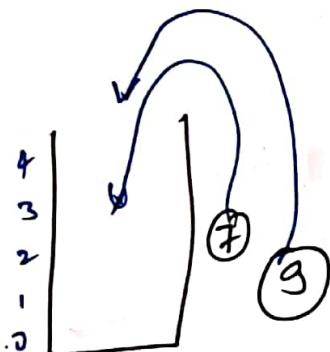
Operation 1: Push

struct stack *s ;
{
 s->size = 8 ;

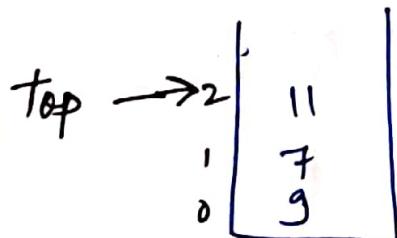
s->top = -1 ;

s->arr = (int *) malloc (s->size * sizeof(int));

Nothing
returned here



when you use malloc, then
it will return void pointer

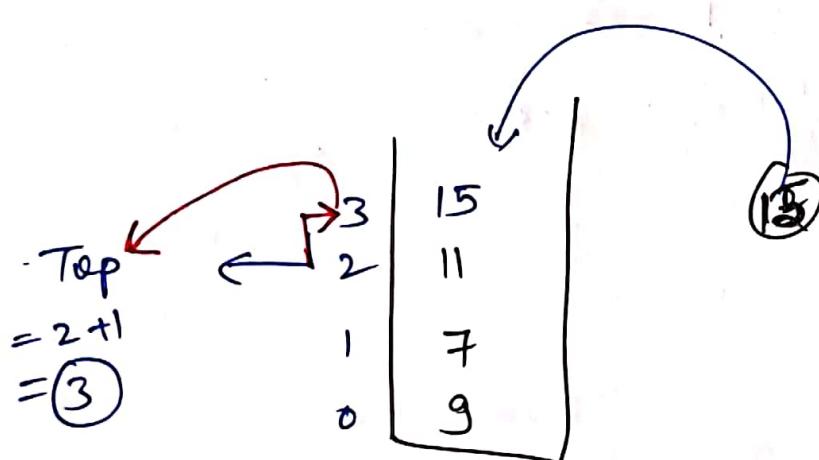


1st check whether the stack is
full or not?

```

Push(value)
{
    if (isFull(s))
    {
        print("stack overflow");
    }
    else
    {
        s-> top++;
        s-> arr[s-> top] = value;
    }
}

```



Operation 2 → Pop

check whether stack is empty or not

```
if (isEmpty (s))  
{  
    printf ("Stack Underflow");  
    return -1;  
}  
else  
{  
    int val = sp → arr [sp → top];  
    s → top = sp → top - 1;  
    return val;  
}
```

3	X	top = 3 X
2	11	top = 2
1	7	
0	9	

Video - 26

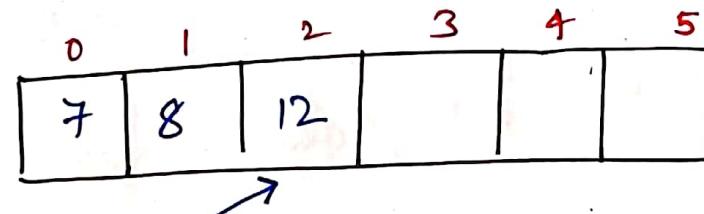
Coding . in V.S. Code

Video - 27

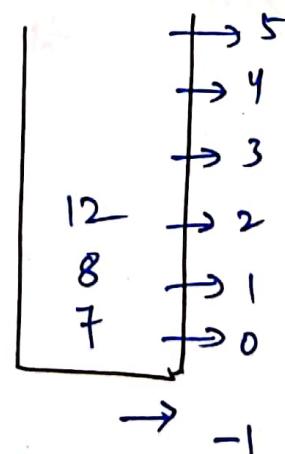
Peak Operation in Stack Using Array

* Peak Operation : *

An operation that will tell you about what value , at which position inside a stack .



Top



Note:-

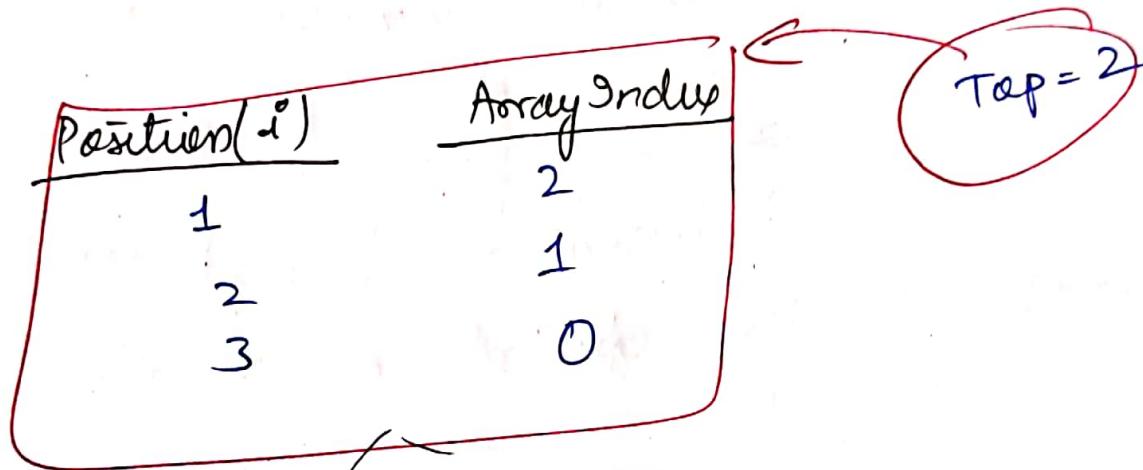
- * when stack is empty we take top as (-1) .
- * what happens when we pop things out of the stack, then we keep going down the top value i.e., from 2 to 1 , becoming 1 to 0 . So, when we pop out the last element from the stack then

$$0 - 1 = -1$$

So, when $\text{top} = -1$, then we cannot get anything out of the stack.
bzc our stack = empty.

$i = \text{index}$

→ When we are using stack, we can't start index from 0(zero)
we'll start our index from "1".



In General :

Position(i)

Array Index
 $(\text{Top} - i + 1)$

Code for Peak

```
int peak (struct Stack *S, int  $i$ )
```

```
{ if ( $S \rightarrow \text{top} - i + 1 < 0$ )
```

```
    { printf("Invalid position");  
      return -1; }
```

```
else
```

```
}
```

```
return S->arr [ $S \rightarrow \text{top} - i + 1$ ];
```

~~⊗~~ in the above Example : ~~⊗~~

peak(5);



matlab 5th position ka
element return karao

→ But in the above eq: it is
not possible or invalid as no
element is present at 5th position.

→ So, check for invalid position

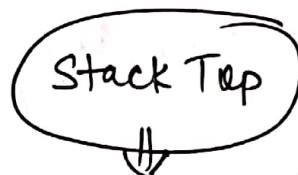
Check:- ~~⊗~~
$$[(Top - i + 1) < 0]$$

Video - 28

Other Operations on Stack

Operation : StackTop

→ gives me top most value of stack



return $s \rightarrow arr[s \rightarrow \text{top}]$;

Time Complexity = $O(1)$ = const. (accessing an array element to constant time)

Operation : Stack Bottom

→ gives bottom most element of our stack

return $s \rightarrow arr[0]$;

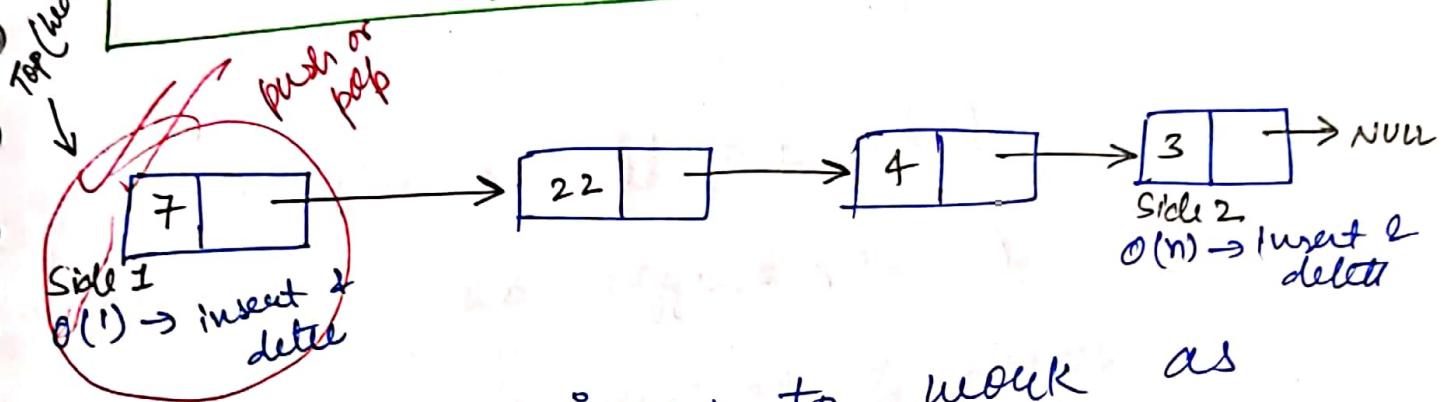
Time Complexity = $O(1)$ = constant

Note

- * Run Time of isFull $\rightarrow O(1)$
- * Run Time of isEmpty $\rightarrow O(1)$
- * Run Time of Push $\rightarrow O(1)$
- * Run Time of Pop $\rightarrow O(1)$
- * Run Time of Peak $\rightarrow O(1)$
- * Run Time of StackTop $\rightarrow O(1)$
- * Run Time of StackBottom $\rightarrow O(1)$

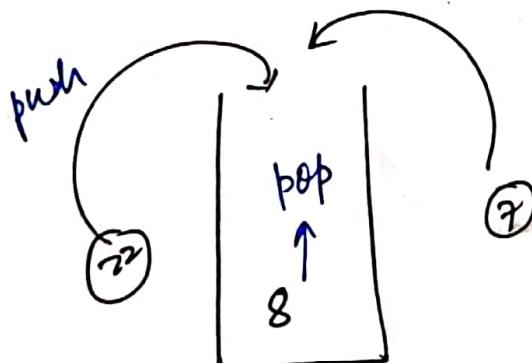
Video - 29

Implementing Stack Using Linked-List



→ we want this LL to work as stack

- Now, we have 2 option
- (i) consider "7" as top
 - (ii) consider "3" as top



Struct Node

{

 int data;

 struct Node * next;

}

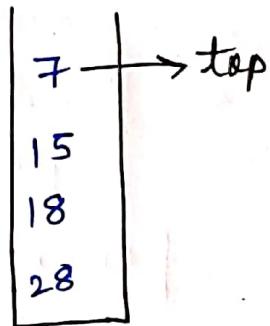
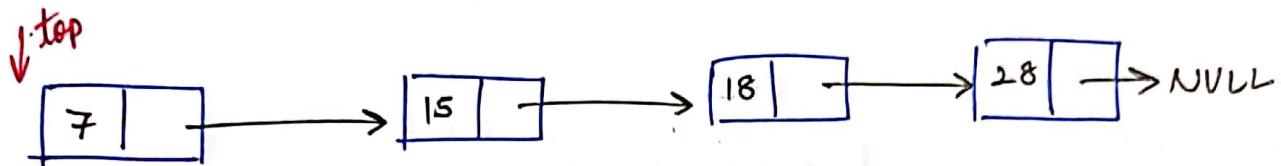


structures for
LL.

- * size + will be used for putting & popping.
- * Head is now referred as top.
- * Stack Empty condition \Rightarrow top == NULL
- * Stack Full condition \Rightarrow when Heap memory is exhausted
 \downarrow
$$[\text{ptr} == \text{NULL}]$$
- * we can make custom conditions for size (full size)
- * You can always set a custom size.

Video - 30

Stack Operations Using Linked Lists



- * "7" is considered as top bcz $O(1)$ for all operations whereas in case of "28" Time complexity = $O(n)$.

① IsEmpty() → Operation

void isEmpty :

```
if (top == NULL)
{
    return 1;
}
else
{
    return 0;
}
```

② IsFull() → Operation

- if we want we can put limit to the size (it depends on you)
- we can also make with unlimited size (heap size)
- # my stack's limit should be unlimited, whatever elements I add to it, it'll accept it.

→ This stack will be full when I am creating node & it couldn't be created.

→ Creating node dynamically

```
struct Node *n = (struct Node *)malloc(  
    sizeof(Node*))
```

```
if (n == NULL)  
{  
    return 1;
```

```
else  
{  
    return 0;
```

* when malloc is returning NULL, that means memory couldn't be returned

③ Push — Operation

Push(x)

→ 1st : Create a new Node

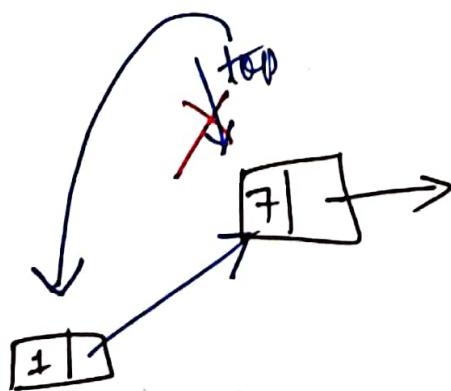
→ 2nd : Check whether stack is full or not

→

Stack Node * $n = (\text{struct Node } *) \text{malloc}(\dots)$

```
if ( $n = \text{NULL}$ )
    {
        print ("Stack Overflow");
    }
else
{
     $n \rightarrow \text{data} = x;$ 
     $n \rightarrow \text{next} = \text{top};$ 
     $\text{top} = n;$ 
}
```

⇒ inserting node @ index 0



④ Pop - Operation

⇒ I need to delete the element at index 0.

```
if (isEmpty)
{
    printf("Stack Overflow");
}
```

```
else
{
```

Struct Node *n = top;

creating pointer

```
top = top->next;
int x = n->data;
```

```
free(n);
```

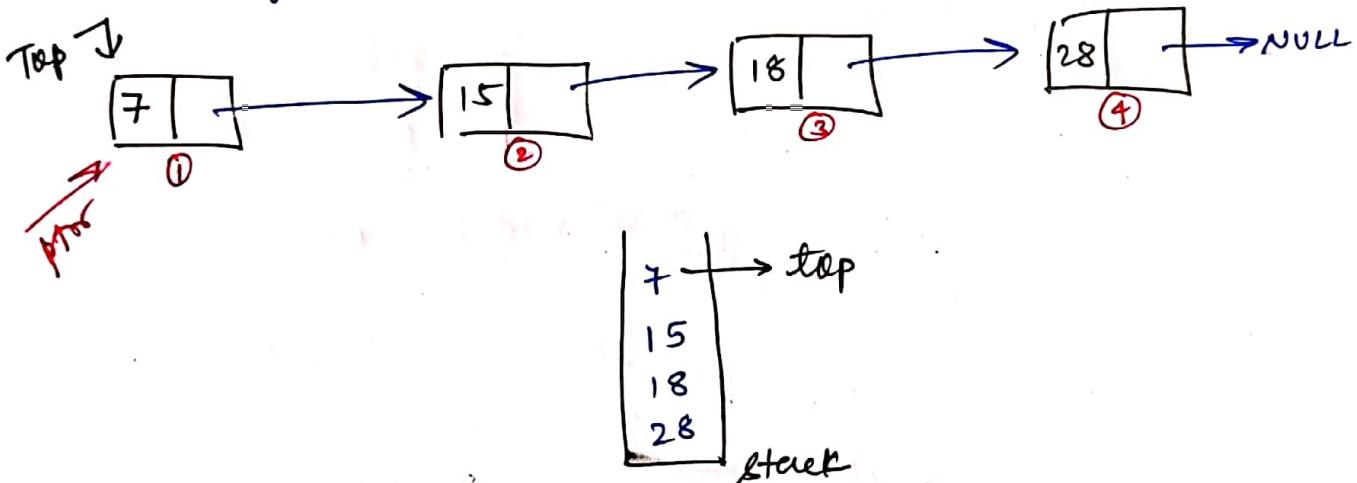
```
return n;
```

Video - 3)

Other Operation on Stack Using L-L

① Peak operation

- it takes an int position
- doing traversal basically (so let a ptr)



Peak (int pos)

{
struct Node *ptr = top;

for (i=0 ; (i<pos-1 && ptr!=NULL) ; i++)

{

 ptr = ptr->next;

}

if (ptr!=NULL)

{
 return ptr->data;

}

```
    else  
    {  
        return -1;  
    }
```

pos	n time movement
1	0
2	1
3	2

(2) Stack Top - Operation

→ return top data

```
int StackTop ()  
{  
    return top->data;  
}
```

(3) Stack Bottom - Operation

→ return bottom-most data

```
int StackBottom()  
{  
    return  
}
```

Video - 32

Parenthesis Matching

$\Rightarrow \underline{(3*2) - + (8-2)}$ → Balanced Parenthesis

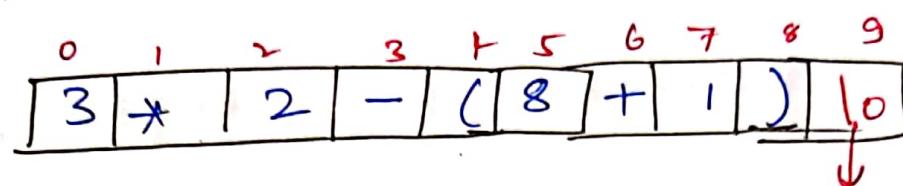
↓
is the parenthesis balanced here
↓
are the brackets used in this
expression balanced

$\Rightarrow a = 1 - 3) * 4 (8$ → Not Balanced Parenthesis

⇒ But if 800 brackets used in any operation
then Parenthesis matching is used.

Example :-

$$\text{Expression} = 3 * 2 - (8 + 1)$$



$$= 3 * 2 - (8 + 1)$$

(? ✓ if yes push into stack
) ? ✓ if yes pop out of stack

* whenever you find an opening parenthesis push it into the stack.

do this until the expression ends.

if anything beside opening & closing parenthesis, just ignore them.

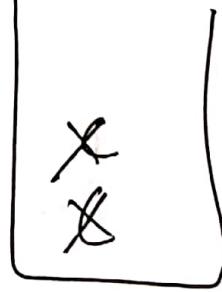
• Opening Parenthesis : - Push into stack

• Closing Parenthesis : - Pop

Condition for a Balanced Expression

- ① while Popping stack should not underflow
⇒ if it happens → unbalanced expression
- ② At end of expression, the stack must be empty ⇒ if it happens → unbalanced exp.
- ③ don't count (Don't do this way)
no. of opening = no. of closing

Example :

$$(7 * 8) - 3(7) \rightarrow \text{unbalanced by } 1 \text{ rule}$$


- It just tells whether parenthesis is balanced or not.
- It doesn't give the idea about validity of an expression.

Video - 33

C-Code for Parenthesis Matching

- 1st : Use a character array (character pointer)
- 2nd : write a function (parenthesis match)
 - it will take expression
- 3rd : → return true : if balanced
return false : if unbalanced
- Any character array which is corresponding to character pointer will scan & tell me if contains parenthesis match or not

Code:-

```
int parenthesisMatch (char *exp)
```

```
{
```

```
Stack *sp;
```

```
// Create the stack (LL or array)
```

```
for (i=0; exp[i]!='\0'; i++)
```

```
{
```

if ($\text{exp}[i] == '('$) \rightarrow opening parenthesis

{
 push (&p, exp[i]);
}

else if ($\text{exp}[i] == ')'$)

{
 if (isEmpty (sp))
 {
 return 0;
 }
 pop (&p);
}

} || for loop

if ($\text{isEmpty} (\text{sp})$) *
{
 return 1;
}

else

{
 return 0;
}

~~selber~~

* Note: \rightarrow Using char array

Video - 34

Multiple Parenthesis Matching Using Stack

$$a = \{ 7 - (3 - 2) + [8 + (99 - 11)] \}$$

op → { ([→ push
close → ?)] → pop



check pop success

when successful when your open & closing parenthesis match

Yes

Keep checking

No

Not Balanced Exp.

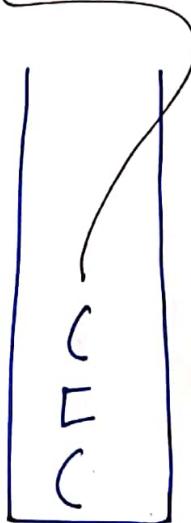
Is stack Empty

Yes
(Balanced)

NO
(Not Balanced)

Ex :-

$$([a * a - (3 + 2)])$$



X not balanced
↓
then don't check
further

Code :-

V.S. Code .

Video - 35

Infix, Prefix & Postfix

⇒ Notations to write an expression

Infix

① $a + b$

② $a - b$

③ p / q

④ $x - 4$

operand <operator> operand

Prefix

<operator> <oprand 1> <oprand 2>

$+ab$

$-xy$

$-pq$

$*pb$

Postfix

<op 1> <op 2> <operator>

$ab +$

$xy -$

$pq *$

Infix

$a * b$

$a - b$

Prefix

$* a b$

$- a b$

Postfix

$a b *$

$a b -$

Note :

Infix : $A * (B + C) * D$

Postfix : $A B C + * D *$

(A 8 * D *)

$7 D *$

↓
7 × D

⇒ Postfix is used by computer/machine

It is easy to evaluate postfix
for the machine hence we convert
infix to postfix

prefix is not much used.

Convert to prefix & postfix

(Q1) $x - y * z$

Prefix \Rightarrow

Step 1: Parenthesize the expression

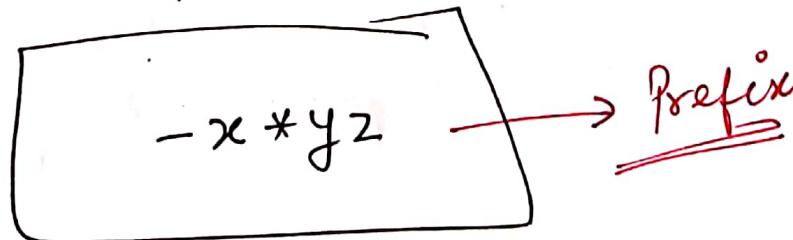
~~.....~~

$$(x - (y * z))$$

Now; once I parenthesized this then
I'll go inside inner most parentheses
will convert this one by one.

$$(x - [*yz])$$

* Square bracket \rightarrow have converted this
parenthesized exp into prefix



Postfix

⇒ operators will come after operands

Step 1: ⇒ Parenthesis the exp.

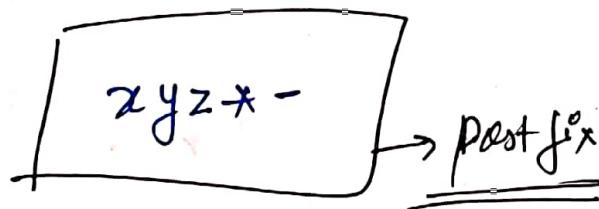
$$(x - (y * z))$$

Step 2: ⇒

$$(x - [yz *])$$

→ Add square bracket to show it's done

Step 3: ⇒



Example:-

Ans. $p - q - r/a$

(1) Prefix:

$$\Rightarrow ((p - q) - (r/a))$$

→ follow operator's precedence and associativity.

$$\Rightarrow [-pq] - [1/ra]$$

$$\Rightarrow \text{---} \rightarrow \begin{matrix} \text{prefix exp} \\ -pq/ra \end{matrix}$$

(2) Postfix:

$$\Rightarrow [pq-] - [ra1]$$

$$\Rightarrow pq-ra/- \rightarrow \begin{matrix} \text{postfix} \\ pq-ra/- \end{matrix}$$

Example:

$$(m-n)*(p+q)$$

→ You can say parenthesis are wrong in this ques. But if parenthesis are already in question you cannot change it.

Step 1: $((m-n)*(p+q))$

Postfix

$$[mn-]*[pq+]$$

$$[mn-][pq+]*$$

$$\Rightarrow mn-pq+*$$

postfix

Prefix

$$[-mn]*[+pq]$$

$$\Rightarrow *-mn+pq$$

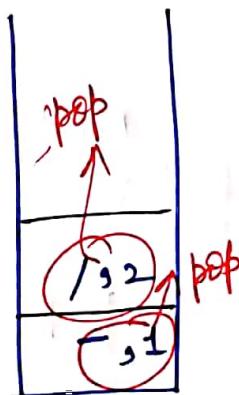
prefix

Video - 36

Infix to Postfix Using Stack

$$\text{Expression} = x - y / z - k * d \quad = \text{Infix}$$

Postfix Expression :



$x y z / - k d$

write the operand and push the operator to stack.

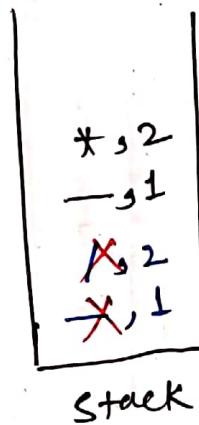
operator Precedence

*	/	2
-		
+		1

- ⇒ If it is bigger than the operator already in stack.
If it is big then it will push bcz its precedence is higher kind of like king.
- ⇒ Always push according to the rule that top one should have higher precedence in stack.
- ⇒ Now, "-" comes. Its precedence is less than "/". so, it can't be push into stack.
So, now, pop "/" and use / place it with the expression.
- ⇒ Now, precedence of 2^{-} is equal; again pop it.
- ⇒ Equal precedence also can't stay together in stack so pop it.
- ⇒ So pop it & place in the expression

- Now push the another " $-$ " in stack.
- "*" can sit over it in stack.
- Now if my whole expression is finished so we will pop "*" and " $-$ " and add them in our expression.

Postfix Exp: $xyz|-kd*$



now solving same ques. manually

$$((x - (y|z)) - (k*d))$$

$$\Rightarrow x - [yz] - [kd*]$$

$$\Rightarrow ([xyz|-] - [kd*])$$

$$\Rightarrow [xyz|-kd* -]$$

Example - 2 :

~~x y z + - k~~

I
Manually

$$x + y * z - k$$

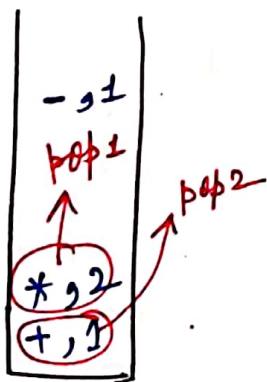
$$((x + (y * z)) - k)$$

$$\Rightarrow ((x + [yz *]) - k)$$

$$\Rightarrow ([xyz * +] - k)$$

$$\Rightarrow \boxed{xyz * + k -} \rightarrow \text{postfix exp.}$$

II
Using Stack



Postfix Exp:

xyz * + k -

+,- $\Rightarrow 1$
1,* $\Rightarrow 2$

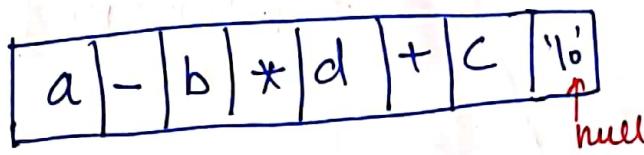
precedence
equal can't stay together in
stack

Video - 37

Infix to Postfix Using Stack Coding

Infix Exp : $(a - b * d + c)$

Character array format



→ input of our program

→ Output of our program will be its
postfix expression.

* Return a character pointer
char * InToPo (char * infix)

{

// declare stack

struct stack * sp;

// initialize the stack

// create empty postfix
size of this = size of infix
char * postfix = (char *) malloc(strlen(infix+1)) *
sizeof(char);

int i=0; → infix scanner

int j=0; // postfix to fill ~~ask for~~

postfix →

--	--	--	--

while (infix[i] != '\0')

functions created

{ if (!isoperator(infix[i]))

postfix[j] = infix[i];

i++;

j++;

}

else

{

if (precedence(infix[i]) > precedence
(Stack Top (St)))

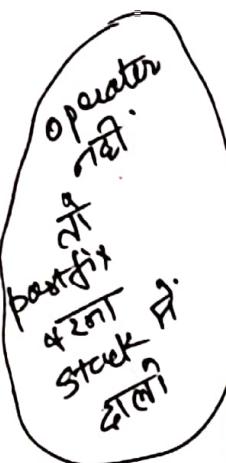
(Stack Top (St))

push(sp, infix[i]);

else

postfix[j] = pop(sp);

j++;



```
while (!isEmpty(sp))  
{  
    postfix[j] = pop(sp);  
    j++;  
  
    if (postfix[j] == 'o')  
        return postfix;  
}
```

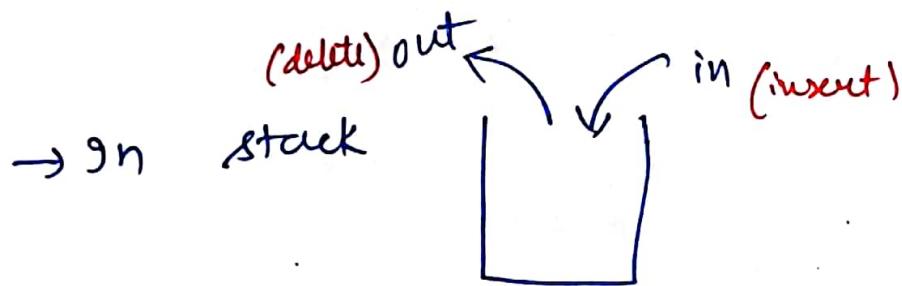
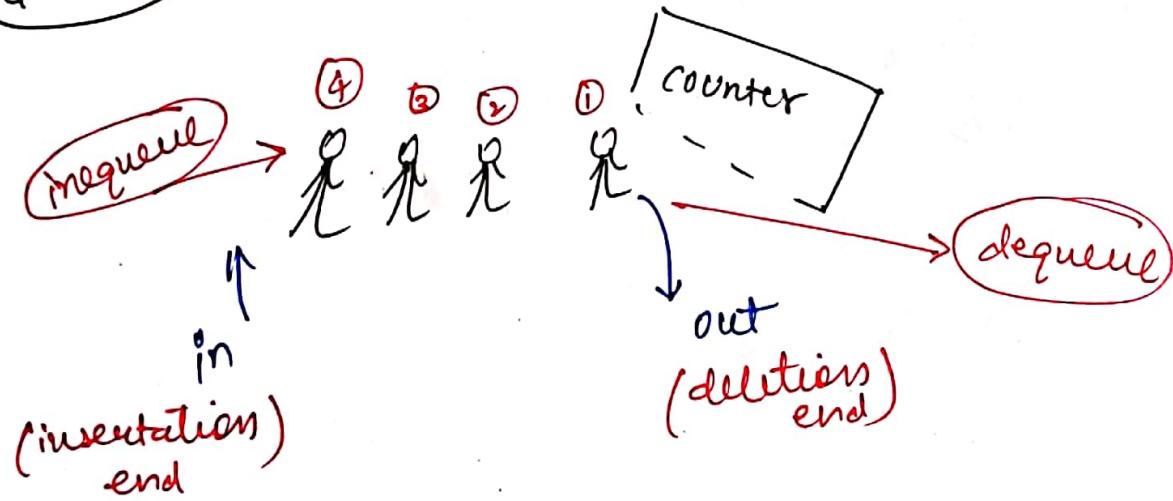
Video - 38

Queue Data Structure

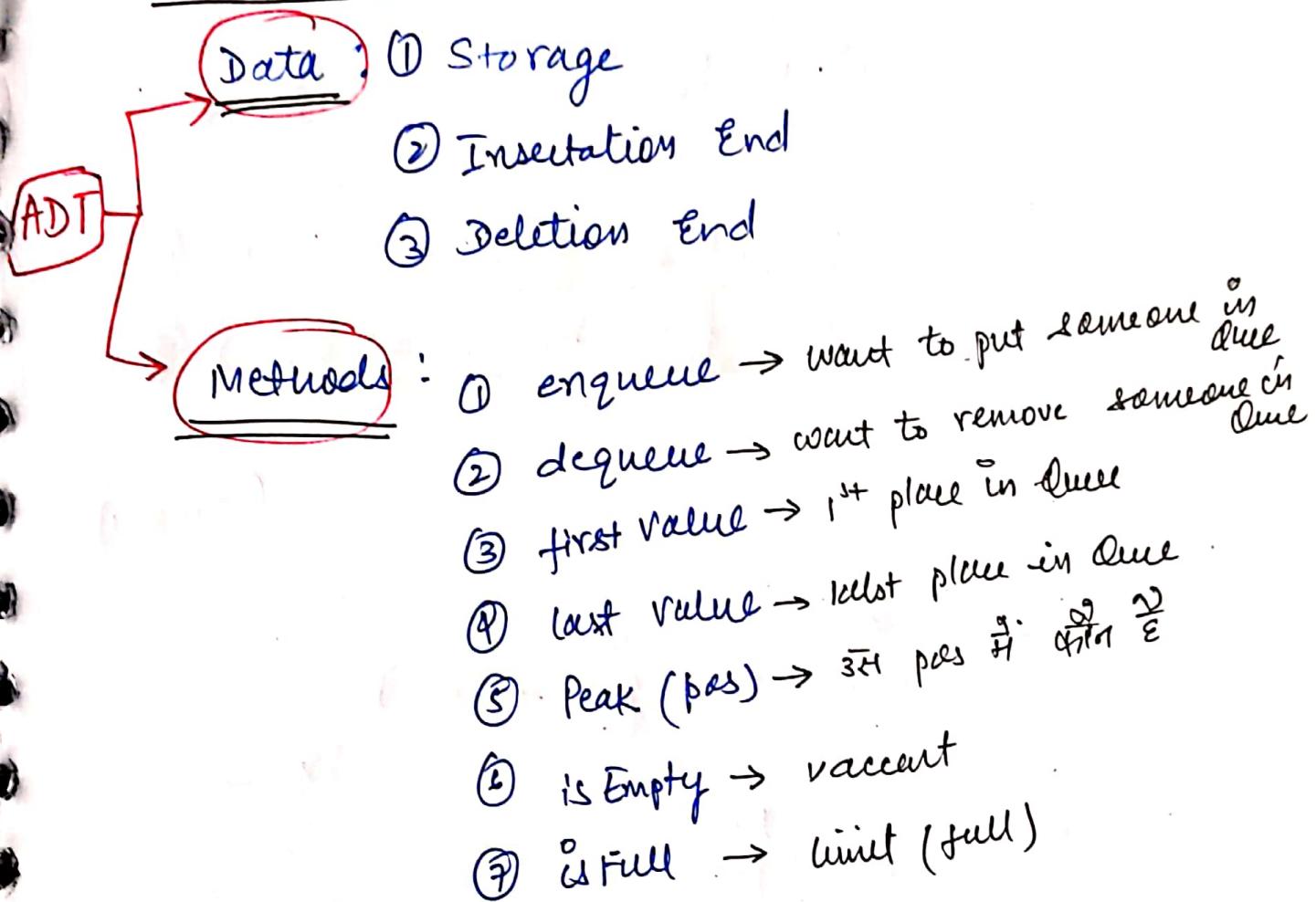
① Stack → LIFO (last in 1st out)
discipline

② Queue → FIFO (first in 1st out)
discipline

Queue



Queue ADT (Abstract Data Type)

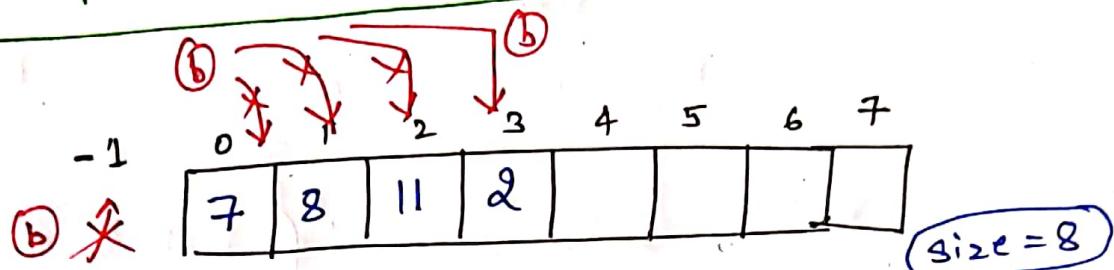


Note:

- # Queue \Rightarrow Not limited to counters
- # Queue can be implemented in various ways
 - ① array
 - ② linked list
 - ③ stack
 - ④ using other ADT

Video - 39

Implementation of Queue Using Array



back pointer
invalid index

→ As we go on inserting the element, value of back pointer changes.

Inserting an Element → Enqueue Operation

- Increment back index
- Insert at back end

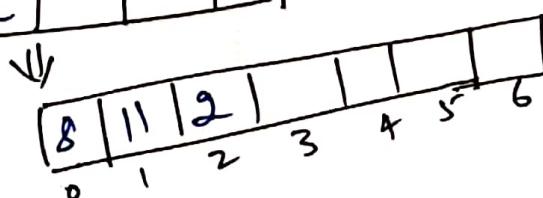
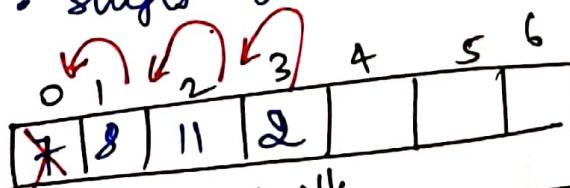
$O(1) = \text{const.}$

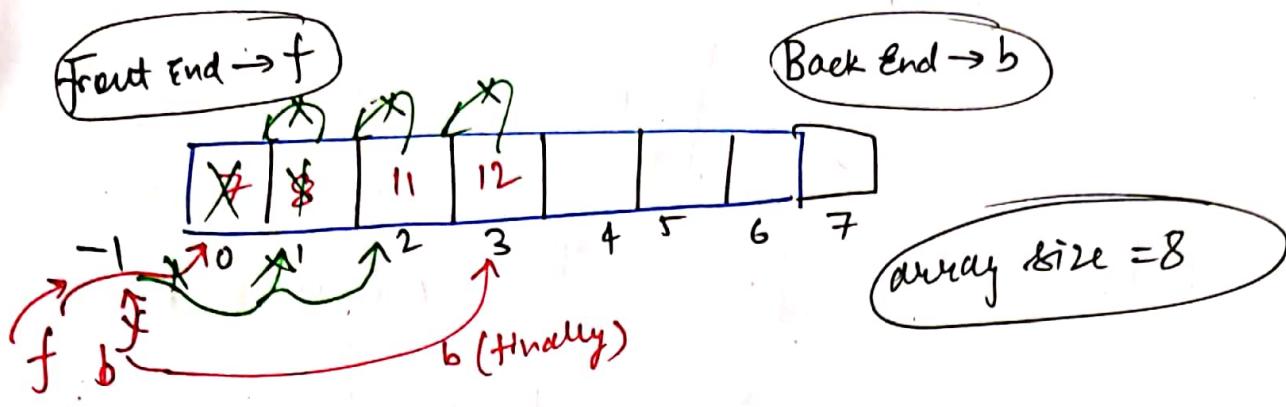
Remove/ Deleting an Element → Dequeue Operation

- Remove at element at index = 0
- Shift all elements

$O(n)$

$\Rightarrow n = \text{no. of elements}$





Insert :-

\rightarrow increment b & insert at b

Remove :-

\rightarrow increment f & remove element at ' f '.

\therefore first element $\rightarrow (\text{front } \text{ind} + 1)$
 Rear element $\rightarrow (\text{Back Ind})$

Queue Empty Condition

front End = Back End

$f = b ;$

Queue Full Condition

$b = (\text{size} - 1) ;$

$\text{size} = \text{array size}$

Note:

~~Stack~~ dequeue Operation $\rightarrow O(n)$

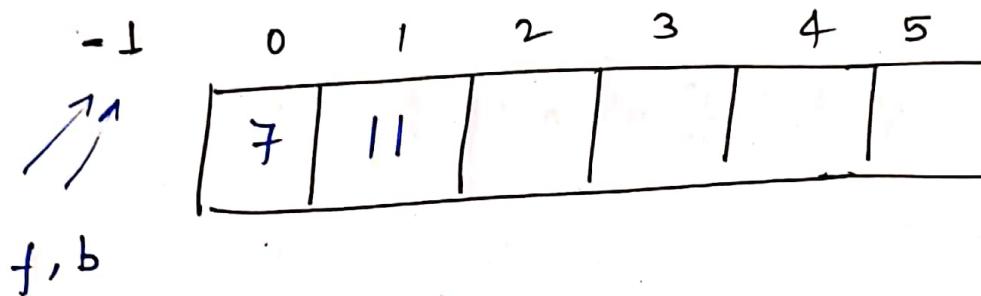
↓
To remove such time complexity
we introduced
back & front end.

∴ maintain front & back end (index)

* Basically, front index not storing index
of front most element.
∴ storing index of = (front element - 1)

Video - 40

Queue Using Arrays



- ⇒ front (f) & back (b) → initially both at -1
when it's empty (not pointer just stores index value)
- ⇒ As I keep on adding elements, then
back (b) will move and add the element.

Code:-

struct Queue

```
{  
    int size;  
    int f;  
    int b;  
    int *arr  
};
```

→ // also dynamically memory allocate for this array

Implementing Queue

```
int main()  
{
```

```
    struct Queue q;
```

```
    q.size = 10;
```

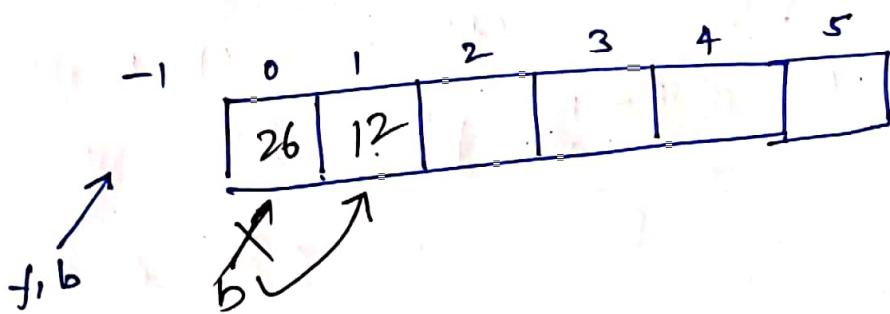
```
    q.f = q.b = -1;
```

```
    q.arr = (int*)malloc(q.size * sizeof(int));
```

```
}
```

we can do 2 way
→ either this way
or
→ dynamically
creating pointer

Enqueue Operation



```
struct Queue {
```

```
    int size;
```

```
    int f;
```

```
    int b;
```

```
    int *arr;
```

```
};
```

```

void enqueue ( struct Queue *q , int val )
{
    if ( isFull ( q ) )
    {
        printf ( " Queue - Overflow " );
    }
    else
    {
        q-> b = q-> b + 1 ;
        q-> arr [ q-> b ] = val ;
    }
}

```

Dequeue Operation

```

struct Queue
{
    int size ;
    int f ;
    int b ;
    int *arr ;
};

;
```

→ elements are removed in queue from
the front end

int deque (struct Queue *q)

{

int a = -1;

if (q->f == q->b) → queue empty condition
isEmpty () {

printf ("No element to dequeue");

}

else {

q->f ++;

a = q->arr [q->f];

}

return a;

} || f^n

(Full f^n)

if (q->b == q->size - 1)

{ return 1; (queue full condition) }

}

Video - 41

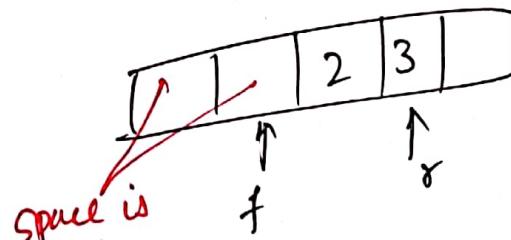
Coding in V.S. Code

Video - 42

Circular Queue

Draw back of Queue Using Array

- space is not used efficiently



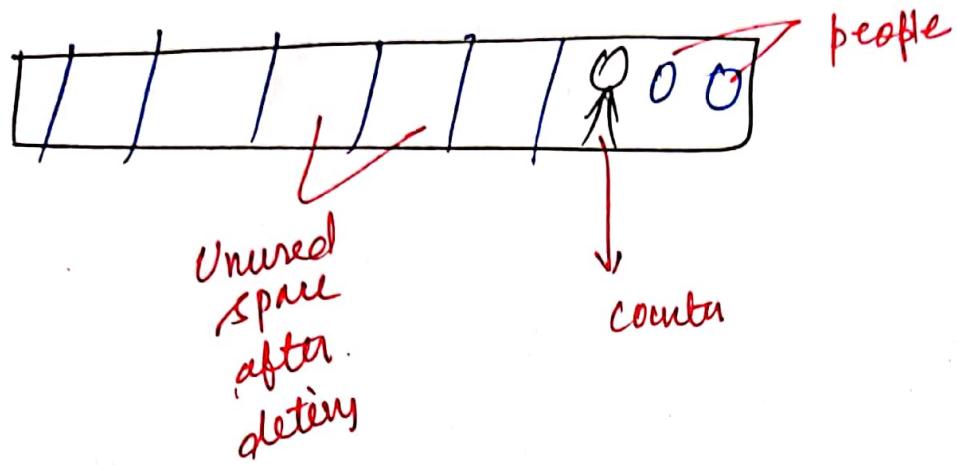
space is
not used after deleting element

Sol ①. $\text{front} = \text{back} \Rightarrow$ Queue ~~full~~ Empty

$$\downarrow \\ f = r = 1$$

↓
it will give us chance to use
space only when $\text{front} = \text{back}$.

(when queue is not empty & still
spaces are there, we can't
make use of them spaces)



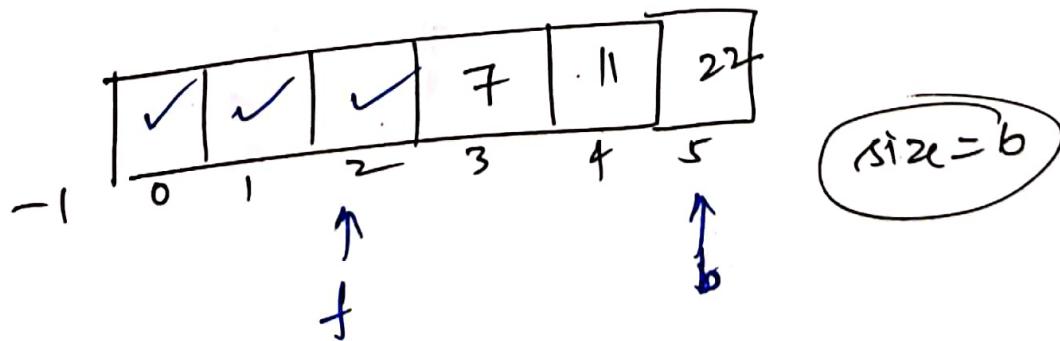
⇒ The outcome for this :

Circular Queue

Circular Increment :

$i = i + 1; \rightarrow$ linear increment
use it by mod operator

$$\Rightarrow i = (i + 1) \% \text{size};$$



~~m*~~ $(i+1) \% \text{size}$ → Circular Increment
 size of above array = 6

$$i=0 \quad (0+1) \% 6 = 1$$

$$i=1 \quad (1+1) \% 6 = 2$$

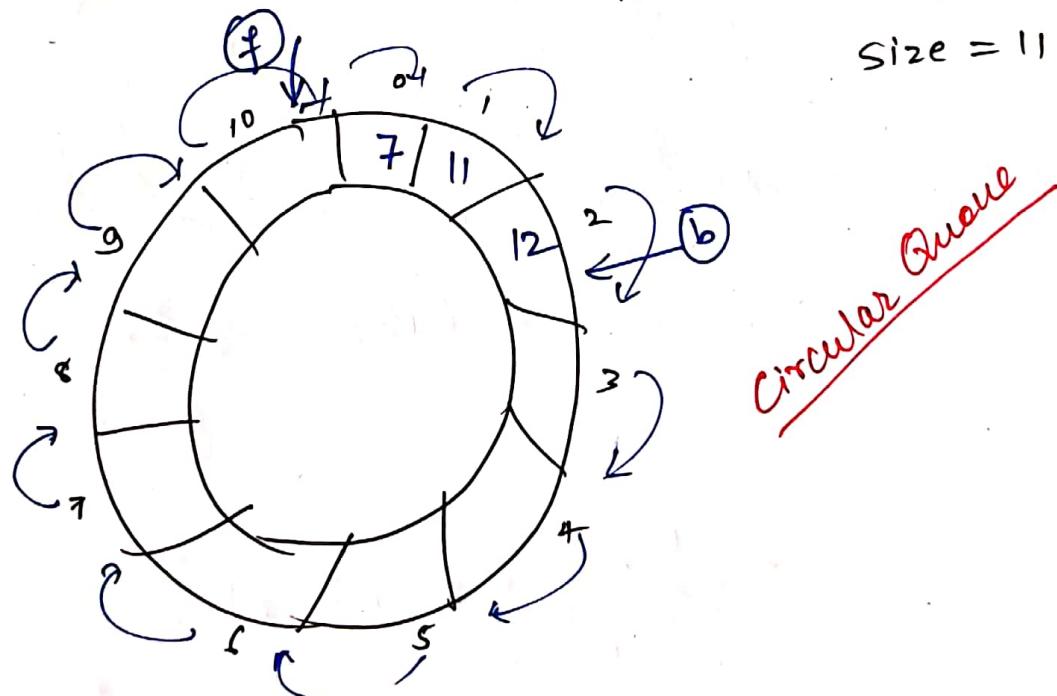
$$i=2 \quad (2+1) \% 6 = 3$$

$$\vdots$$

$$i=4 \quad (4+1) \% 6 = 5$$

$$i=5 \quad (5+1) \% 6 = 0$$

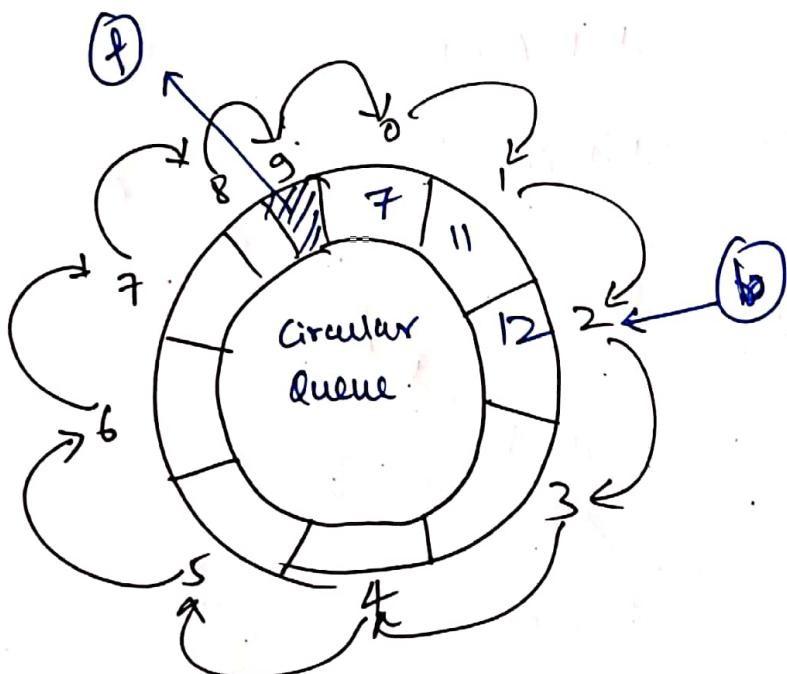
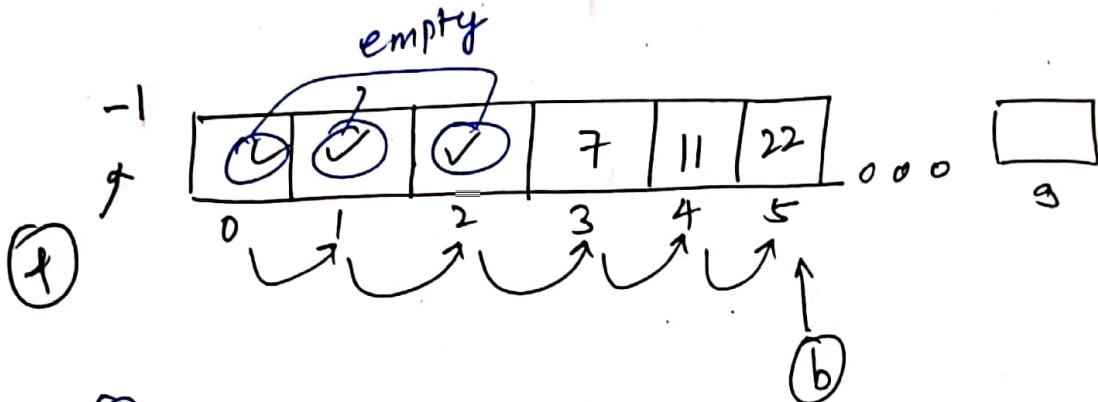
remainder
mod operator



Visualizing array this way
(in circular form/pattern)

Video - 43

Coding Circular Queue



Enqueue Operation \rightarrow Insert

```
void enqueue ( struct Queue *q , int val )
```

{

```
{ if ((q->b+1) % q->size) == q->f )
```

{

```
    printf (" Queue Overflow " );
```

}

```
else
```

{

Circular
movement

```
    q->b = ((q->b+1) % (q->size));
```

```
    q->arr [q->b] = val ;
```

}

```
3 / \ fn
```

Φ

Dequeue Operation \rightarrow delete

```
int dequeue(struct Queue *q)
```

```
{
```

```
    int val = -1;
```

// check whether Queue or not

if Empty

```
if ( $q \rightarrow b == q \rightarrow f$ )
```

```
{
```

```
    printf("Empty Queue");
```

```
}
```

```
else
```

```
{
```

```
     $q \rightarrow f = ((q \rightarrow f + 1) \% (q \rightarrow size))$ ;
```

```
    val = q->arr[q->f];
```

```
}
```

```
return val;
```

```
} // f^n
```

Note:-

* front

~~exit~~

khali rehta hai Queue \Rightarrow

Video - 44

Coding in VLSI Code

Note:-

* In Circular Queue, initially \Rightarrow
front = back = 0 (not -1)

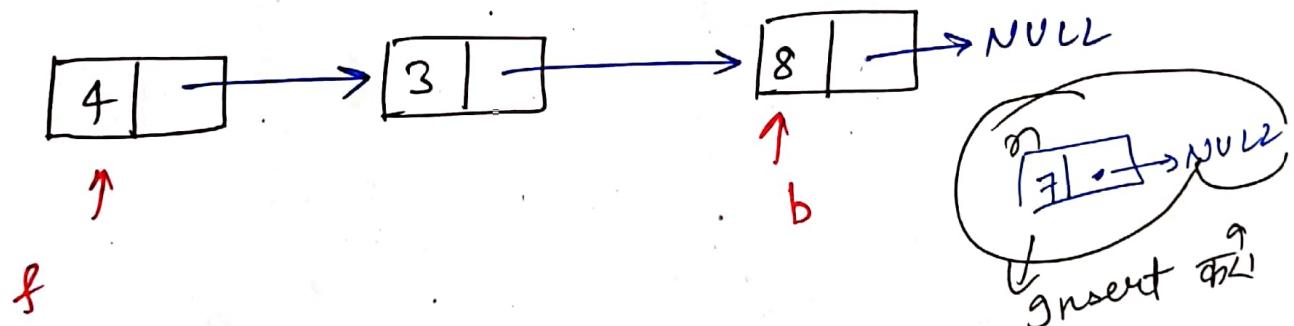
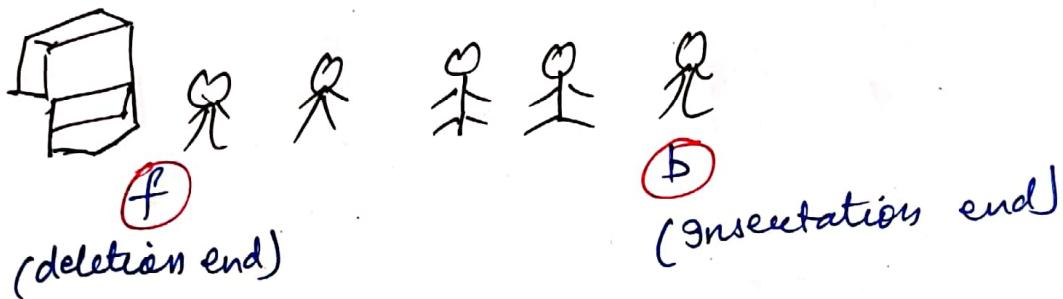
* But in linear Queue,
initially front = back = -1 (not 0)

it will overwrite its element
to fix 1 position in circular loop
i.e., Circular Queue
so that it doesn't overwrite its
content.
(1 blank space are left here.)

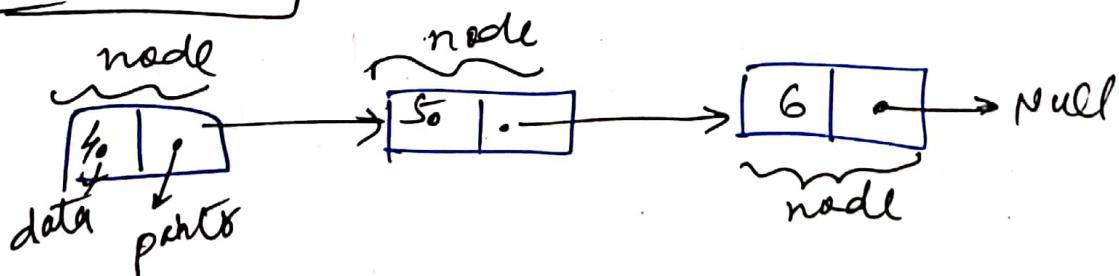
Video-45

Queue Using Linked List

queue



Linked List



Enqueue Operation (Insert)

```
void enqueue (struct Node* f, int val)
{
    struct Node* n = (struct Node*) malloc(sizeof(Node));
    n->data = val;
    n->next = f;
}
```

$n = (\text{struct Node}^*) \text{malloc}(\text{sizeof}(N))$;

$\text{if } (n == \text{NULL})$

{
 printf("Queue full");
}

else

{
 n->data = val;
 n->next = f;
 }
~~return 0;~~

* when Queue Empty $\Rightarrow f = b = \text{NULL}$

Empty Queue

Special Case

{
 if ($f == \text{NULL}$)
 {
 f = b = n;
 }
}

else

{
 b->next = n;
 b = n;
}

sign

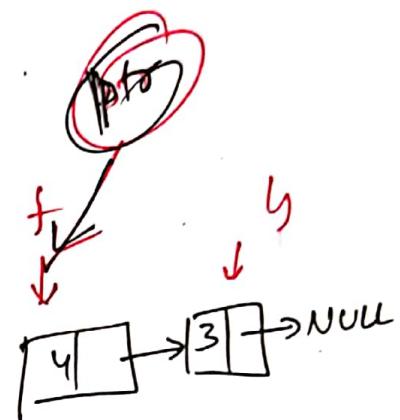
Queue Empty Condition

① $f(\text{front}) == \text{NULL}$

Condition for Queue full

① $n == \text{NULL}$

(3rd node ~~is null~~)



Dequeue Operations

$\rightarrow \text{front} \leftarrow \text{hoga}$

~~int dequeue (struct Node *f) {~~

 int val = -1;

 Node *ptr = f;

 // check for empty queue

 if ($f == f \rightarrow \text{next}$);

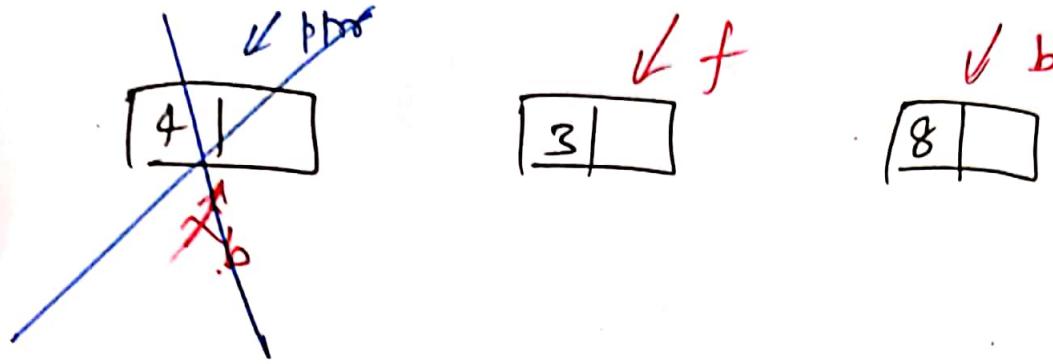
 val = ptr \rightarrow data;

 free (ptr);

 return val;

 else block
(if not Q empty)

}



Video - 46

Coding in V.S. Code

Video - 47

Double Ended Queue

(DEQueue)

→ Not to be confused with deque

→ It's DEQueue

→ Queue → FIFO principle (first in first out)

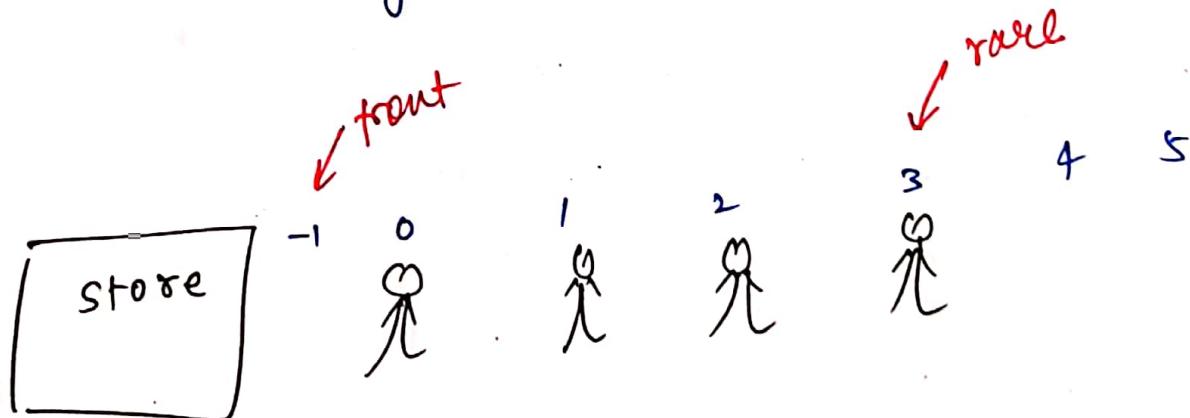
→ Queue → Not FIFO X

Queue

• Insertion from Rear

• Deletion from front

• front initially at "-1" position



D E Queue

- Insertion from (Rare + front) \rightarrow शुरू होना एवं रखना
- Deletion from (Rare + front) \rightarrow छोड़ना एवं रखना
- Don't follow FIFO principle

Two Types of D E Queue

- ① Restricted Input D E Queue
- ② Restricted Output D E Queue

Restricted Input D E Queue

- Insertions from front not allowed

Restricted Output D E Queue

- Deletion from rare is not allowed

DE Queue ADT

- ① Data \rightarrow same as Queue
 - ② Operation \rightarrow ~~is Empty()~~
 - 1) ~~is Full()~~
 - 2) ~~Initialize()~~
 - 3) ~~point()~~
 - 4) ~~enqueue F()~~ \rightarrow front = F
 - 5) ~~enqueue R()~~ \rightarrow rear = R
 - 6) ~~dequeue F()~~ \rightarrow front = F
 - 7) ~~dequeue R()~~ \rightarrow rear = R
- only this
2 done to
be left*

Video - 48

Introduction to Sorting Algorithm

Sorting

→ Method to arrange a set of elements in either increasing or decreasing order according to some basic or relationship among the elements.

1 9 8 2 7 . . .  → sorting

Two way of sorting

① Ascending order
→ 1 2 7 8 9

② Descending order
→ 9 8 7 2 1

Ques: Why Sorting ?

swiggy/zomato → Restaurant

→ sort by Rating
→ sort by Price

① School Assembly

↓
we stood at height-line during morning assembly.

The basic of sorting \Rightarrow height

② Binary Search

↓
Searching in sorted array takes at most $O(\log N)$ time. if not sorted it takes $O(n)$ time.

∴ Retrieval becomes much faster.

③ Dictionary

↓
words are sorted for you to find it easily (lexicographically \rightarrow sorted)

④ Swiggy, zomato, flipkart, Amazon

↓
sort
→ prices
offer
new launch

⑤ social media application \rightarrow emails, outlooks

Videos - 49

Analysis Criteria For Sorting Algorithm

① Time Complexity

→ which algorithm works efficiently for larger data sets and which algorithm works faster with smaller data sets.

→ Eg: 1 sorting algo sorts only 4 elements efficiently & fails to sort 1000 elements

⇒ Algo 1 $\rightarrow O(n^2)$

Algo 2 $\rightarrow O(n \log n)$

∴ Algo 2 better than Algo 1

$O(n \log n)$ better $O(n^2)$

⇒ Lesser the time complexity, the better is the algorithm.

② Space Complexity

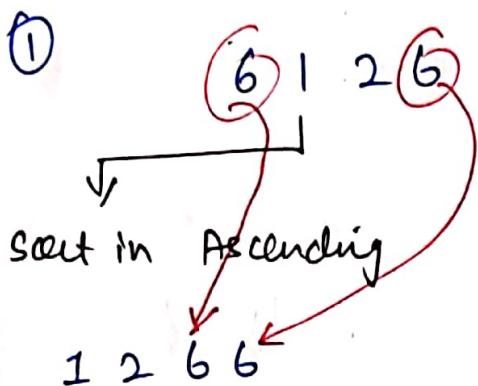
- help us to compare the space the algorithm uses to sort any data set.
- If any algorithm consumes a lot of space for larger inputs \Rightarrow it's considered a poor algorithm for sorting large data sets.

In-place sorting algorithm

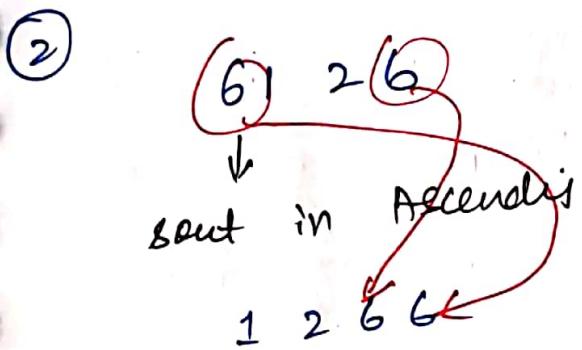
- ⇒ Algorithm which results in constant space complexity.
- ⇒ Mostly use swapping and rearrange technique to sort a data set.
eg: Bubble sort

③ Stability

- ⇒ whether the order of the elements having equal status on some basis is preserved or not.



→ stable sorting algo



→ not stable sorting algo

Example:-

store

1
A

3
B

8
C

7
E

2
F

rank of officer
Name of officer

Sort
↓
A E F B D C } → stable sorting algorithm

⇒ if 2 person. with same Rank,
then whoever comes 1st; he/she

will be served first.

⇒ maintain the order

⇒ same value to order to maintain
that is

④ Internal / External Sorting Algorithm

Internal SA

⇒ All the data is loaded into the memory

External SA

⇒ All the data is not loaded into the memory
memory (RAM)

⑤ Adaptive

⇒ adapt the fact that if the data
are already sorted and it must take
less time

⑥ Recursive / Non-Recursive SA

⇒ Algorithm uses recursion to sort
a data set ⇒ Recursive Algorithm

otherwise ⇒ Non-Recursive Algorithm

Video - 50

Bubble Sort Algorithm

0	1	2	3	4	5
7	11	9	2	17	4

→ input array

0	1	2	3	4	5
2	4	7	9	11	17

→ output array

→ sorting in Ascending Order

1st Pass

→ 5 comparison → 5 possible swap

→ whole array → unsorted
→ start by comparing each adjacent pair
→ ∵ array size = n
then pair to compare = $(n-1)$

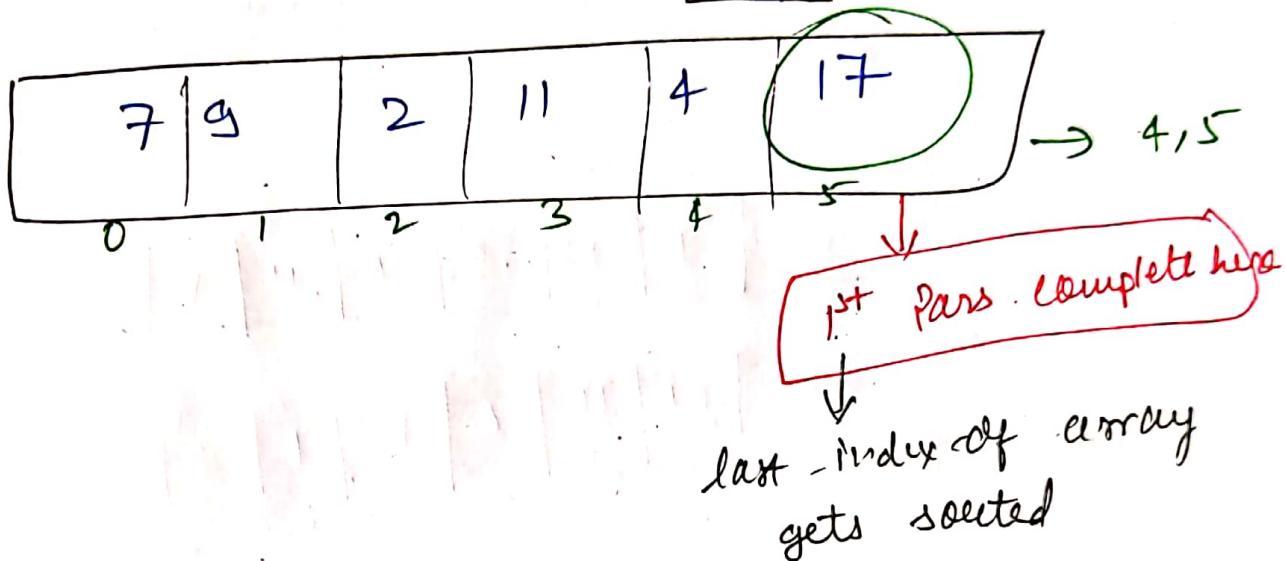
0	1	2	3	4	5
7	11	9	2	17	4

→ 0, 1

✓ ✓ swap
7 9 11 2 17 4 → 112
swap

7 9 2 11 17 4 → 2, 3

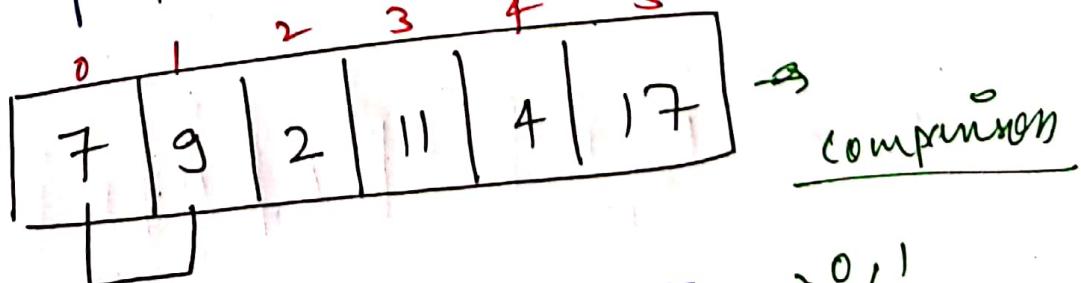
7 9 2 11 17 4 → 3, 4
swap



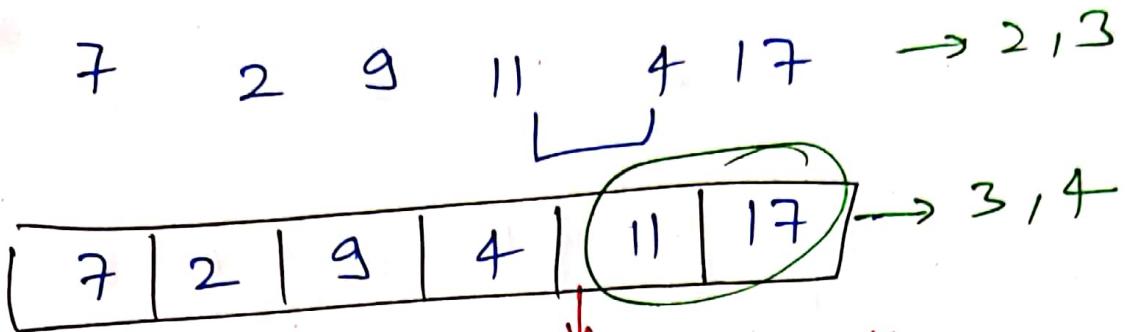
→ now we have to sort only upto
4th index of array

2nd Pass → 4 comparison → 4 possible & swap

→ beginning se start karو again swapping
& no. of pair to compare = 4

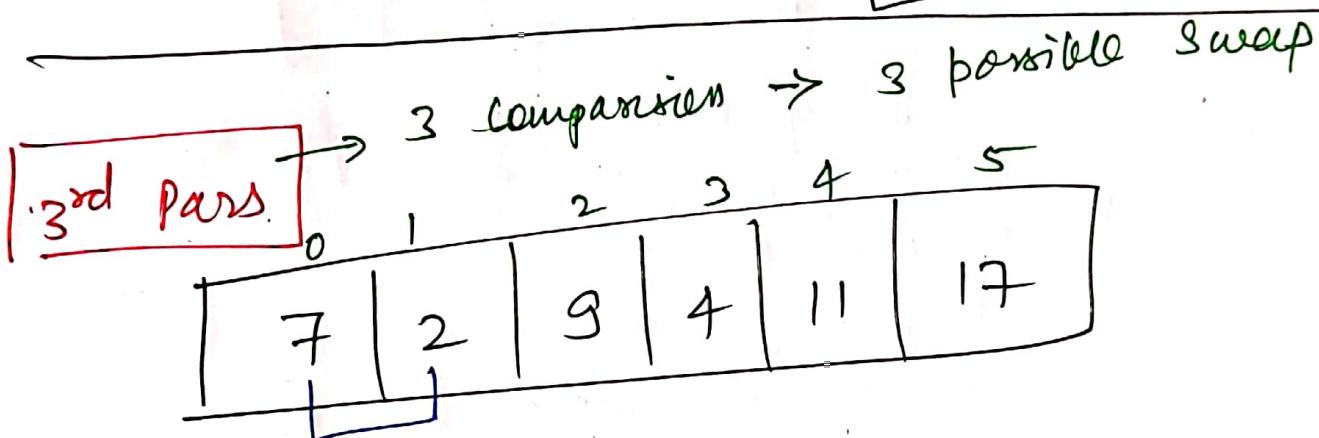


7 9 2 11 4 17 → 0, 1
7 2 9 11 4 17 → 1, 2



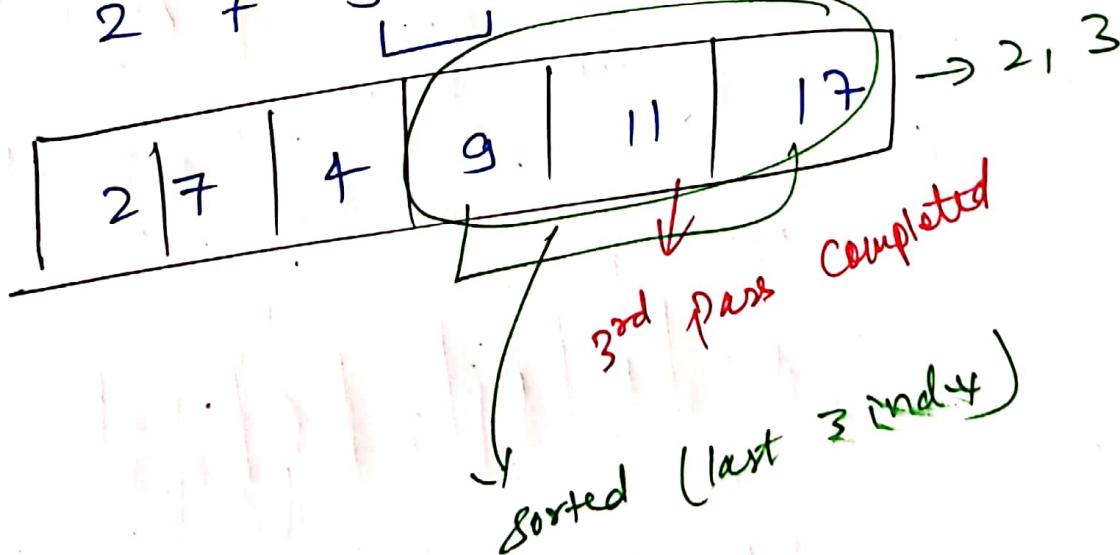
2nd pass complete

↓
last 2 index sorted
[11 17]

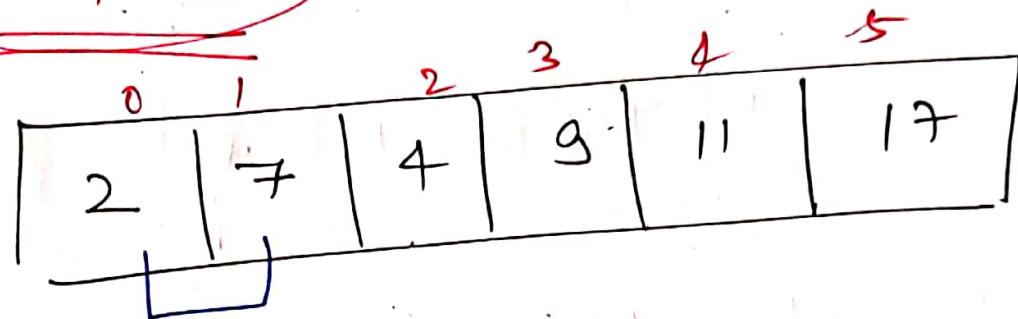


2 7 9 4 11 17 → 0, 1

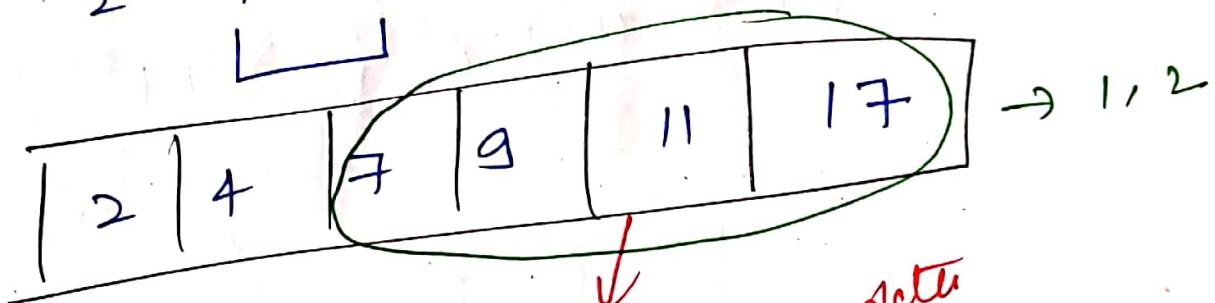
2 7 9 4 11 17 → 1, 2



4th Pass :- → 2 Comparison → 2 pos. swap

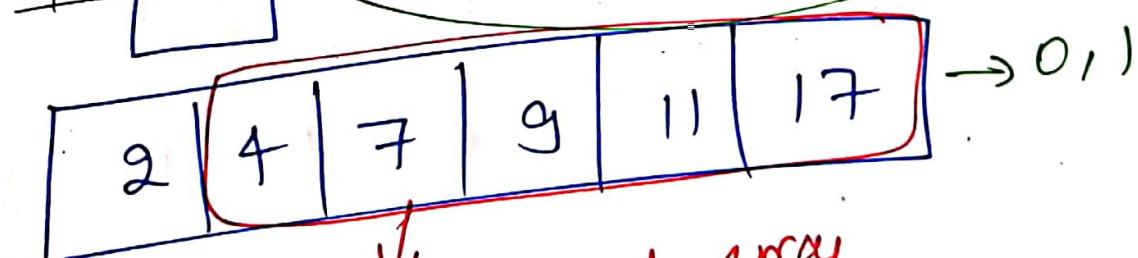
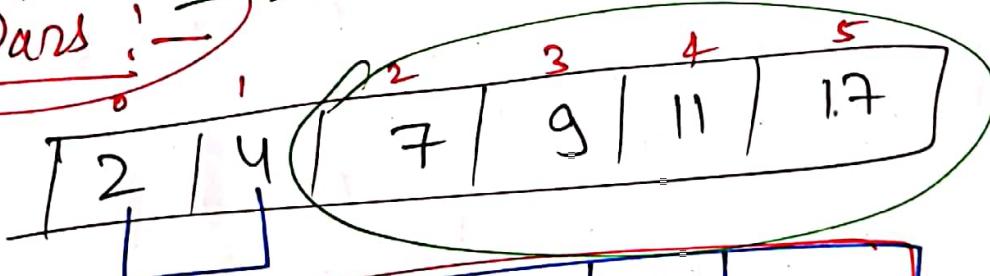


2 7 4 9 11 17 → 0, 1



4th pass complete

5th Pass :- → 1 comparison → 1 possible swap



got sorted array

5th pass gives ⇒ Sorted Array

Note:-

Length of array = 6 (n)
No. of Passes = 5 $(n-1)$

Note:- ① Comparison Number (Total)

Total no. of comparison for array

$$= (5+4+3+2+1)$$

$$= 15 \text{ comparison}$$

for General:

$$= 1 + 2 + 3 + 4 + \dots + (n-1)$$

$$= \frac{n(n-1)}{2} = O(n^2) \text{ (time complexity)}$$

$$\downarrow (n-1) + (n-2) + (n-3) + \dots + 1$$

$$= \frac{n(n-1)}{2} = O(n^2)$$

② Stability

✓	✓		
7	8	7	7

0 1 2 3

→ Unsorted array

↙

✓	✓	✓	
7	7	7	8

0 1 2 3

→ sorted array

⇒ order is same as the order in input array
 ⇒ Hence, stable Bubble sort is stable

⇒ Bubble sort is not adaptive by default, But we can make

it adaptive

0	1	2	3	4	5
7	11	9	2	17	4

↓

lighter element ←

heavier element →

Time Complexity

$O(n)$ → adaptive

Bubble Sort → if already sorted array is given

$O(n^2)$ → otherwise

Video - 51

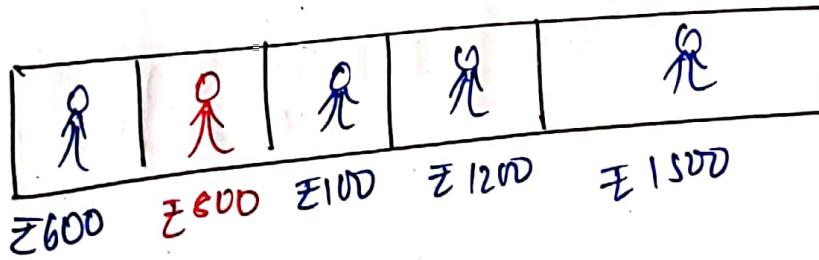
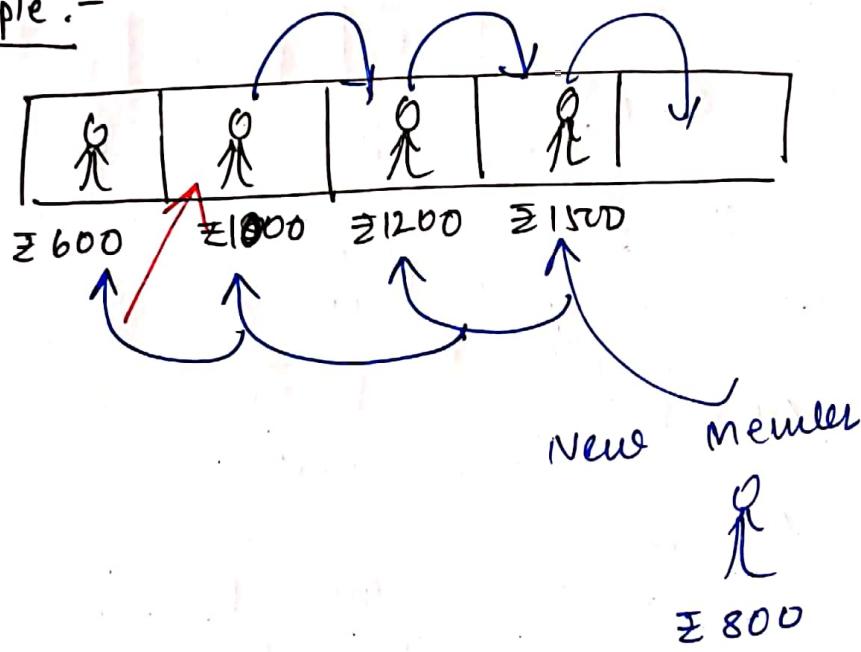
	Bubble Sort	Coding	
$i=0$ 1st Pass	: 5 comp.	; $(n-1)$ comp	$(n-1-i)$
$i=1$ 2nd Pass	: 4 comp.	; $(n-2)$ "	
$i=2$ 3rd Pass	: 3 comp.	; $(n-3)$ "	
$i=3$ 4th Pass	: 2 comp.	; $(n-4)$ "	
$i=4$ 5th Pass	: 1 comp.	; $(n-5)$ "	

$$\text{Size} = 6 = n$$

Video - 52

Insertion Sort Algorithm

Example :-



⇒ insert new member +
sorting of array is conserved.

7	11	20	21	
---	----	----	----	--



(13)

7	11	20		21
---	----	----	--	----



(13)

7	11		20	21
---	----	--	----	----



(13)

7	11	13	20	21
---	----	----	----	----

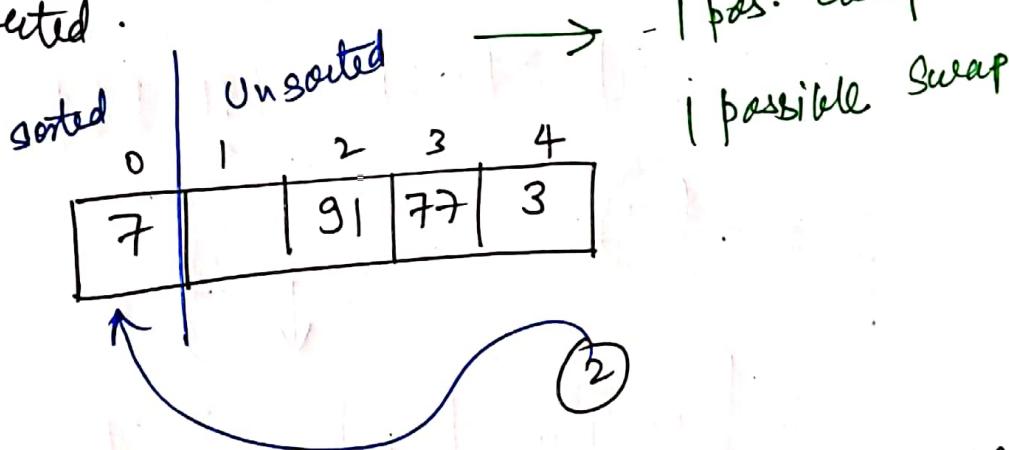
* Inserting an element in
a sorted array *

⇒ Use the insertion sort algo to sort its elements in increasing order.

0	1	2	3	4
7	2	31	77	3

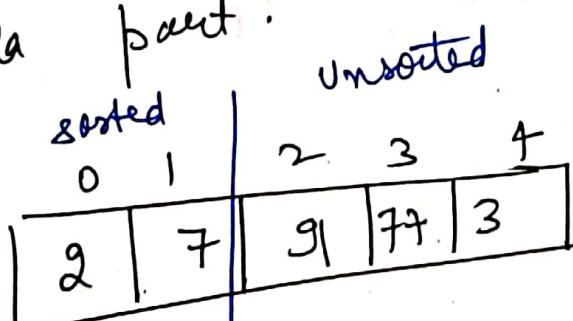
* Array of single element is always sorted.

1st Pass

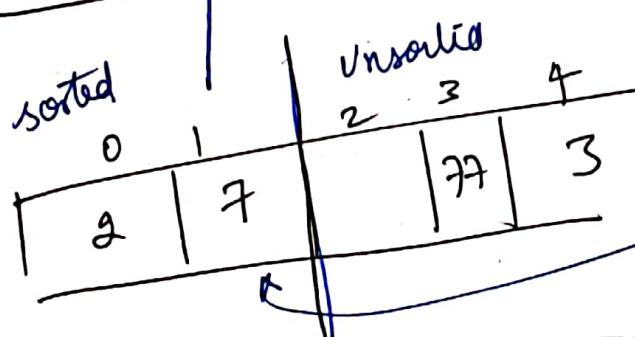


* Pluck 1st element from beginning of unsorted part and insert it in sorted part.

2nd Pass



2 possible comparisons
2 possible swap



0	1	2	3	4
2	7	91	77	3

sorted unsorted

Pass - 3

0	1	2	3	4
2	7	91		3

3 possible compare
3 possible swap

0	1	2	3	4
2	7	77	91	3

4 unsorted

Pass - 4

0	1	2	3	4
2	7	77	91	

4 poss. comp
4 poss. swap

0	1	2	3	4
2	3	7	77	91

sorted array

Time Complexity

length of array = 5

(n)

Total no. of Passes = 4

(n-1)

Total Pass. Comp./Sweep = $1 + 2 + \dots + (n-1)$

$$= \frac{n(n-1)}{2}$$

$$= \mathbb{O}(n^2)$$

↓ worst case complexity

Best Case Complexity

↓ Already Sorted Array

Total Comp. = (n-1)

$$= \mathbb{O}(n)$$

↓ best case =

Analysis

① Total Parses = $(n-1)$

② Total Comp. = $1+2+\dots+(n-1)$
= $\frac{n(n-1)}{2}$

= $O(n^2) \rightarrow \underline{\text{worst case}}$

③ Best Case Total Comp = $O(n)$

④ Stability → Yes, it's stable

⑤ Adaptive → Yes, it is adaptive
when the array is sorted
by nature
this is an adaptive nature

* No Intermediate Result

* But Bubble sort → intermediate result
↓
last index sorted out.

Video - 53

Insertion Sort in V.S. Code

Video - 54

Selection Sort Algorithm

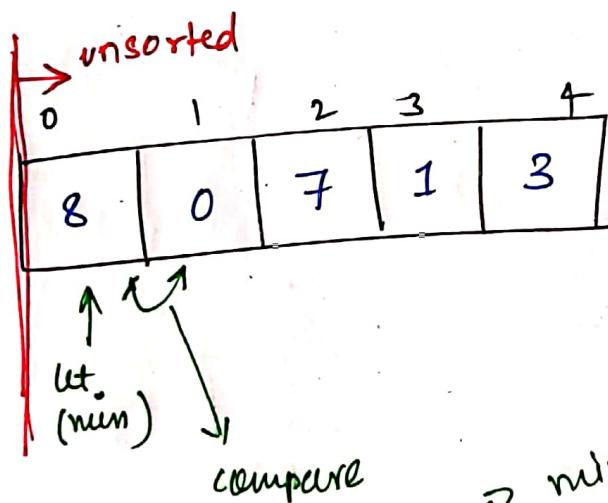
0	1	2	3	4
8	0	7	1	3

length of array = 5

Unsorted Array

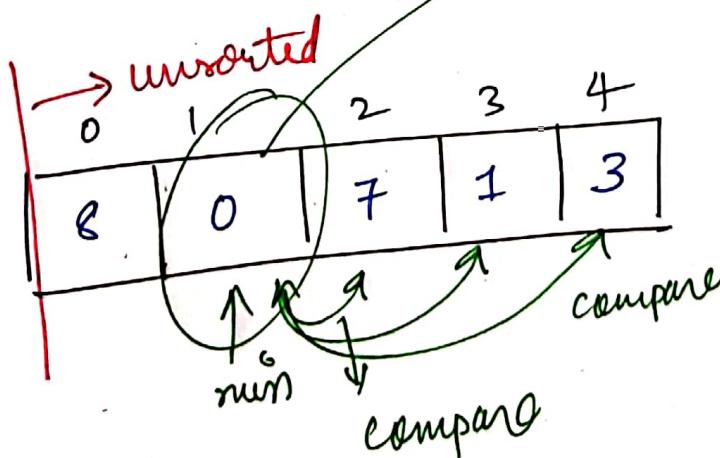
Now

①



Pass - 1

②



exchange
min with
"8"

1st position
element

mini \Rightarrow "0"

⇒

sorted

0	1	2	3	4
0	8	7	1	3

(2)

Pass

Pass - 2

↓
wt
(min)
compare

unsorted

~~0 8 7 1 3~~

0	8	7	1	3

↑
min
compare

~~0 8 7 1 3~~

0	8	7	1	3
0	1	2	3	4

min
compare

exchange
min with 1
(2)

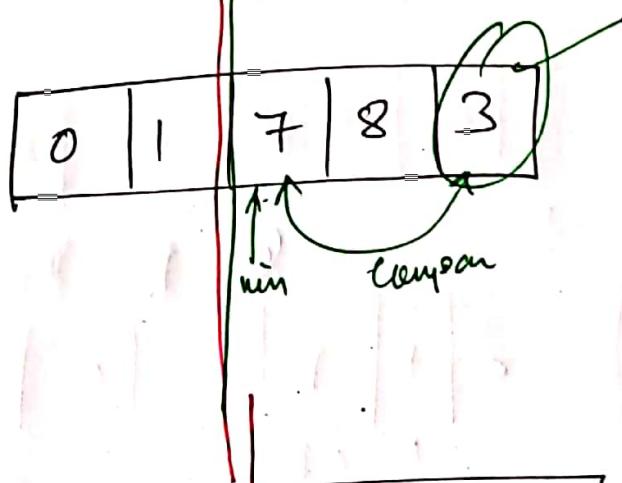
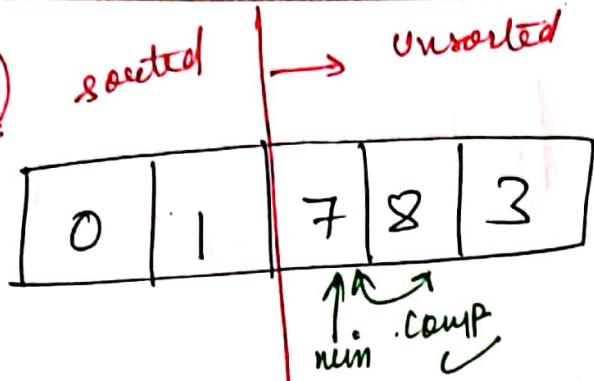
→
new position
element
(Index = 1)

→ 1st element
after red
line

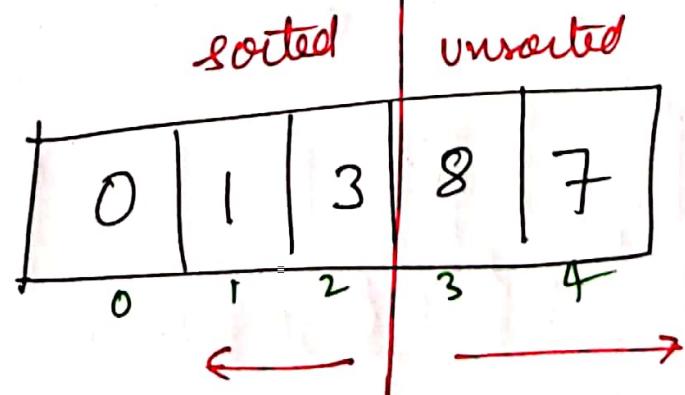
~~0 8 7 1 3~~

0	1	7	8	3
0	1	2	3	4

Pass 3



↓
exchange
min
with
1st element after
red line



Pass 4

sorted	unsorted
0 1 3 8	7

↑ min
compare

0	1	3	8	7
---	---	---	---	---

↑ min

↓ exchange with
pt element
after red line

sorted	unsorted
0 1 3 7	8

0 1 2 3 4

⇒ ∵ a single element is always sorted,
∴ we ignore unsorted part & make it
sorted too.

Detailed Explanation

0	1	2	3	4
8	0	7	1	3

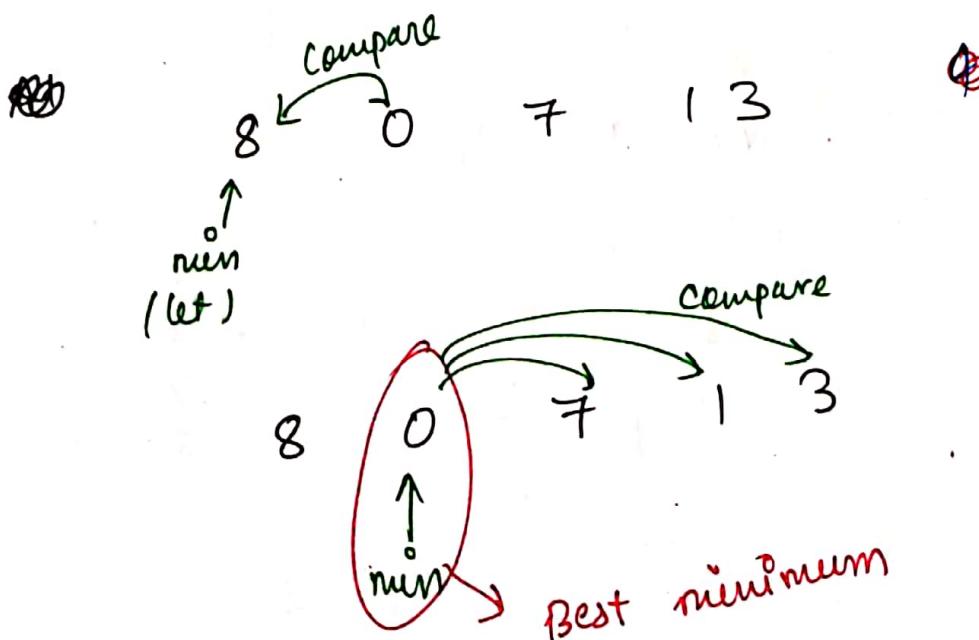
unsorted array

1st Pass:- (1 loop)

Consider element at $\text{index} = 0$ as minimum and then compare it with each rest element of array ; as you proceed and get other value less than that at $\text{index} = 0$; make it as "min" and again compare "new min" with rest element.

check the Best minimum.

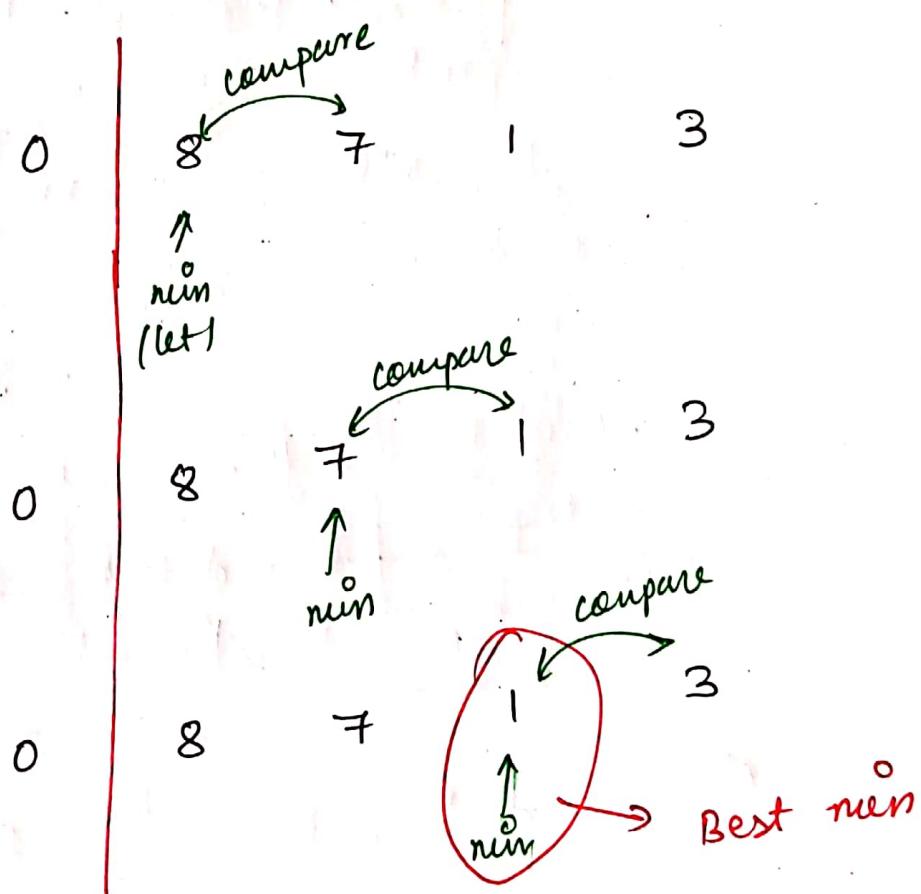
Then, swap the Best minimum with element at $\text{index} = 0$.



<i>sorted</i>	0	1	2	3	4	<i>unsorted</i>
	0	8	7	1	3	

2nd pass. → (3 comp.)

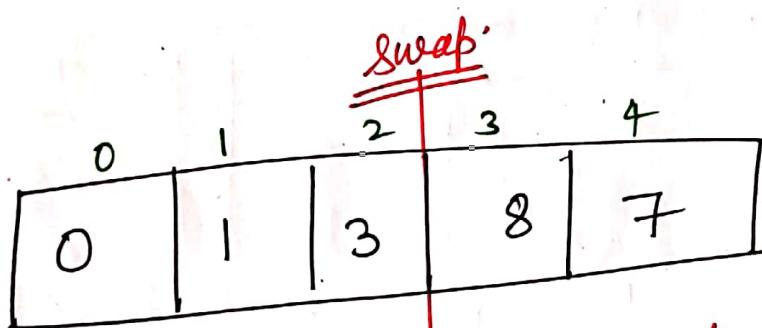
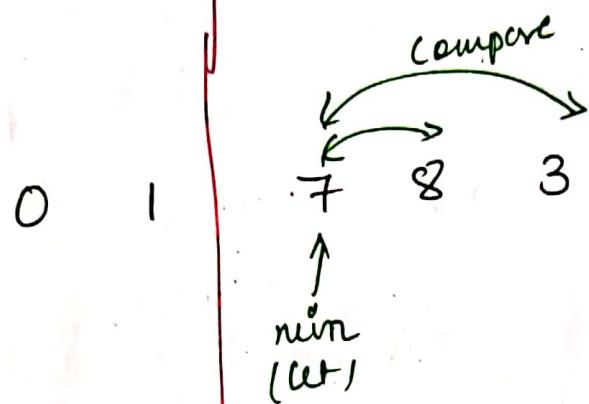
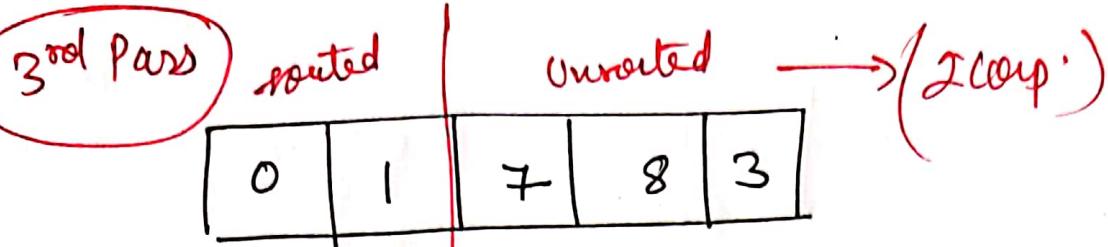
Now repeat pass 1 from beginning
of unsorted array i.e., from index = 1



swap.

0	1	7	8	3
0	1	2	3	4

sorted *unsorted*



sorted unsorted

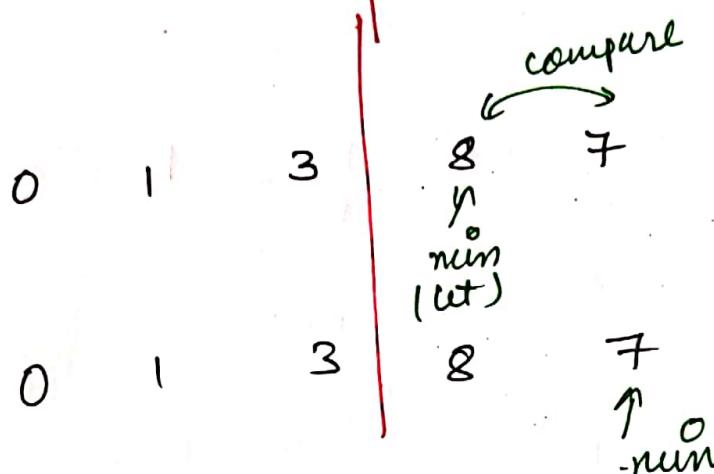
9th Pass:

0	1	3	8	7
---	---	---	---	---

sorted

unsorted

(1 Comp.)



swap

0	1	3	7	8
---	---	---	---	---

sorted

unsorted

Note:-

A single element is always sorted
we ignore unsorted part & make it
sorted.

0	1	3	7	8
---	---	---	---	---

sorted array.

Note:-

- ① Length of array = $\frac{n}{2}$ (n)
- ② Total no. of passes = $\frac{1}{2} (n-1)$
- ③ Possible Comparisons = $1 + 2 + \dots + (n-1)$
= $\frac{n(n-1)}{2}$
Time complexity $= O(n^2)$

- ④ Max. Swap = $n-1$
⑤ Min. Swap = 0 [sorted for array]

⑥ Stability & Adaptive

7	8	8	1	8
---	---	---	---	---

Unsorted array

after sorting

1	7	8	8	8
---	---	---	---	---

sorted array



⇒ Selection Sort is not stable.

⇒ Selection Sort is not Adaptive

↓
Comparison
 \leq \geq \neq $=$

⇒ sorting in minimum no. of swaps.

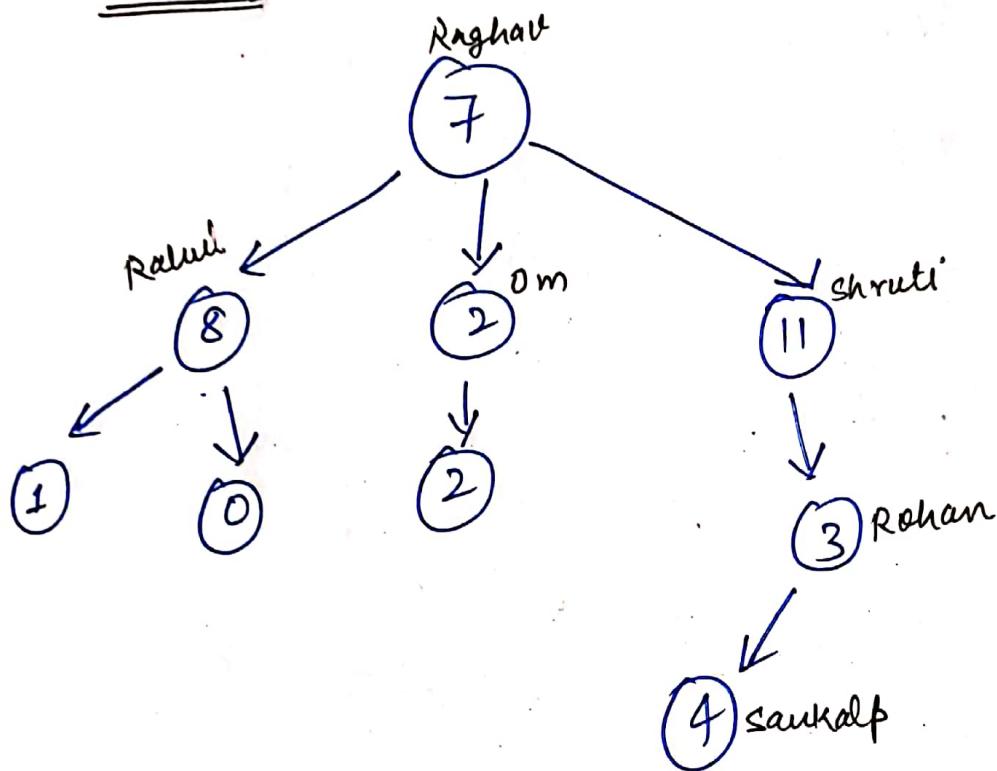
Video - 5-5

Selection Sect in VSS Code

Video-61

Introduction to Trees

Trees



Terminology

- ① Root → Topmost Node
- ② Parent → node which connects to the child
- ③ Child → node which is connected by another node as its child

④ Leaf / External node → Nodes with no children

⑤ Internal node → Node with atleast 1 child

⑥ Depth → No. of edges from root to the node

⑦ Height → No. of edge from node to the deepest leaf.

⑧ Sibling → nodes belonging to the same parent

⑨ Ancestors / descendants

Defination

Tree

→ represent the hierarchy of elements and depicts the relationship b/w the elements.

→ Trees are considered as one of the largely used facets of data structure.

→ made up of nodes & edges.

Terminology

① Root

- Topmost node of a tree
- There is no edge pointing to it, but one or more edge originating from it.

② Parent

- Any node which connects to the child.
- Node which has an edge pointing to some other node.

③ Child

- Any node which is connected to a parent node.
- Node which has an edge pointing to it from some other node.

④ Siblings

- Nodes belonging to the same parent

⑤ Ancestors

- Nodes accessible by following up the edges from a child node upwards.

⑥ Descendants

→ Nodes accessible by following up the edges from a parent node downwards.

⑦ Leaf / External Node

→ Nodes which have no edge originating from it, and have no child attached to it. These nodes cannot be a parent.

⑧ Internal Node

→ Nodes with atleast one child.

⑨ Depth

→ No. of edges from root to that node.

⑩ Height

→ No. of edges from that node to the deepest leaf.

~~(Node)~~

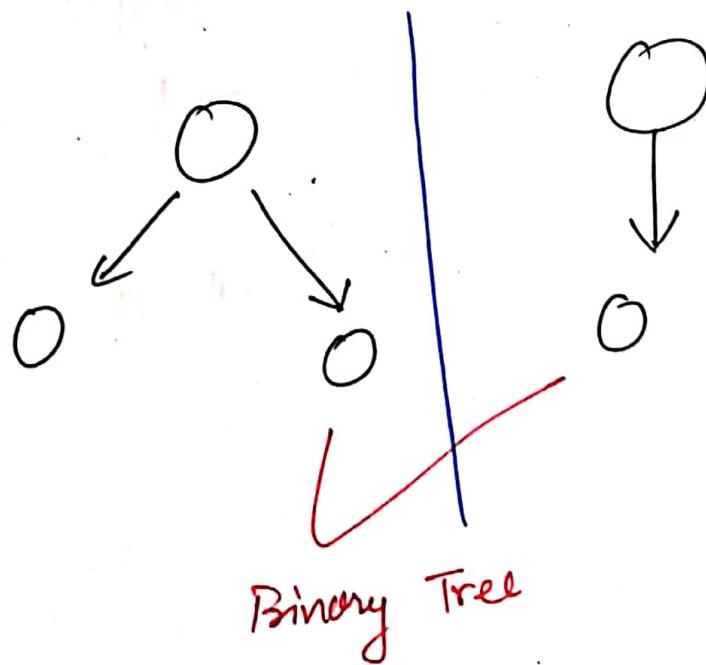
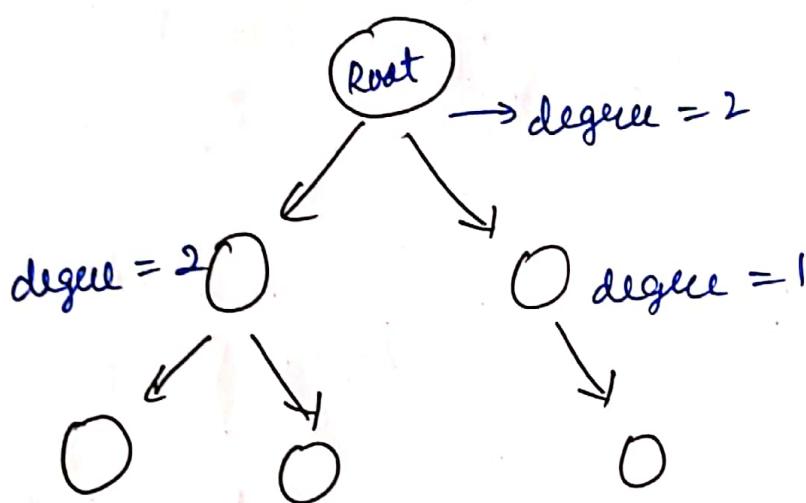
→ degree of tree = highest degree of a node among all the nodes present in the tree.

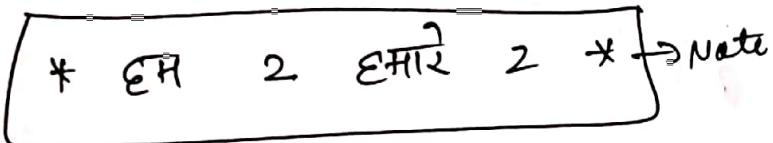
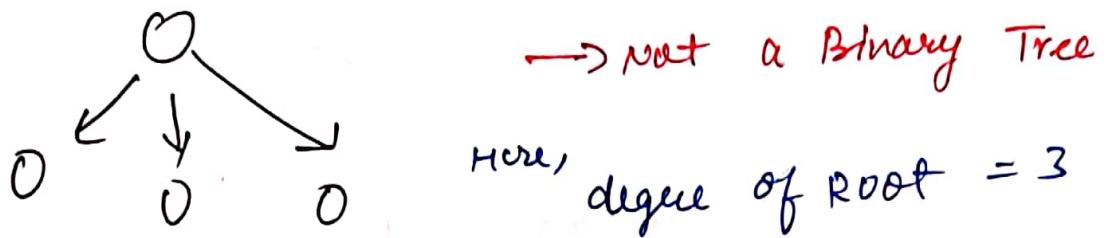
→ degree of node = no. of children of a node.

Video - 62

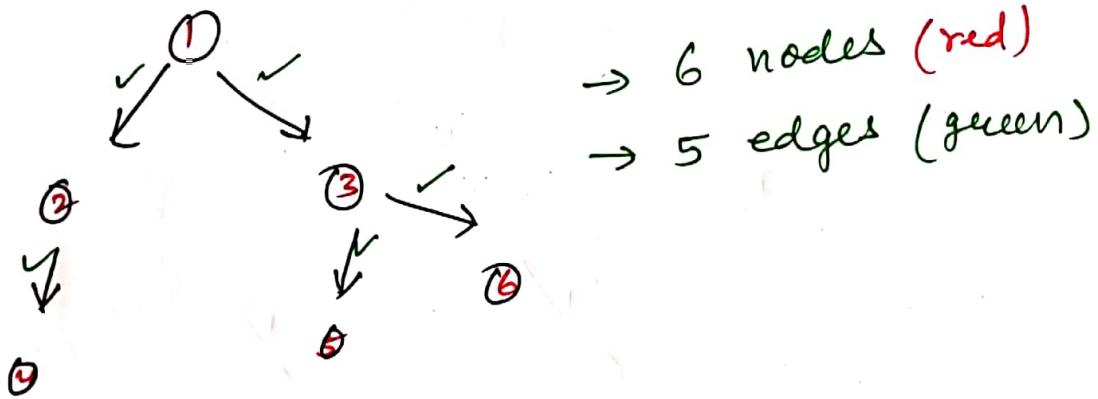
Binary Tree

→ special type of tree where each node has a degree equal to or less than two which means each node should have at most 2 children.





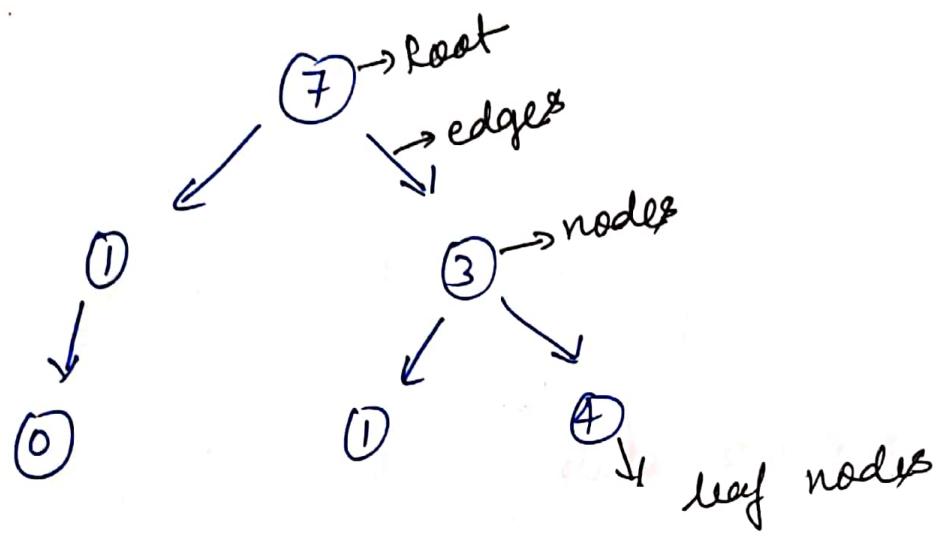
- ① Tree is made up of nodes & edges
- ② n nodes $\Rightarrow (n-1)$ edges



- ③ Degree of Tree \Rightarrow NO. of direct children (for a node)

- ④ Degree of Tree $\xrightarrow{\text{whole}}$ highest degree of a node among all the nodes present in the tree.

- ⑤ Binary Tree \Rightarrow Tree of degree 2
Nodes can have 0 or 2 children



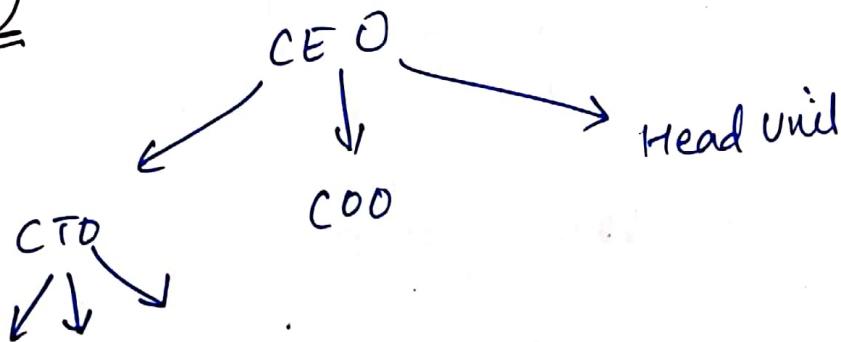
Binary Tree

Video - 63

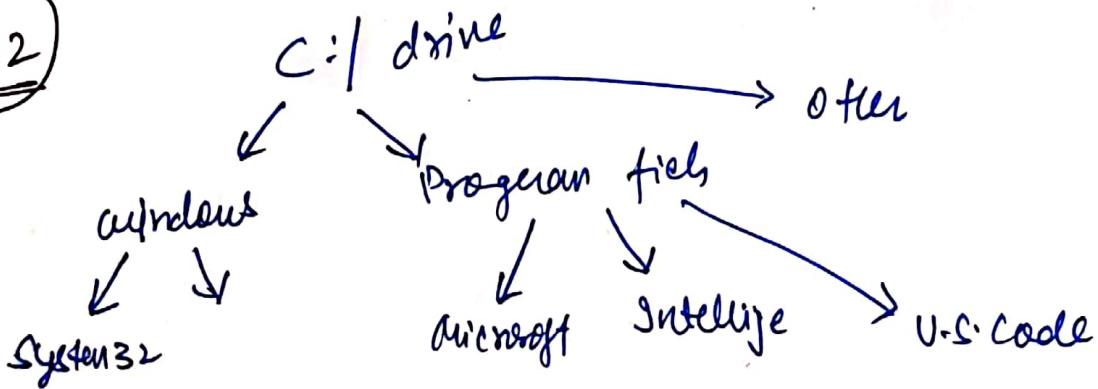
Types of Binary Trees

- **Tree**: Non linear data structure
- **Array, Stack, Queue, Linked List**: Linear data structures
- ideal for representing hierarchical data
- in a tree with n nodes $\rightarrow (n-1)$ edges

Ex: 1



Ex: 2



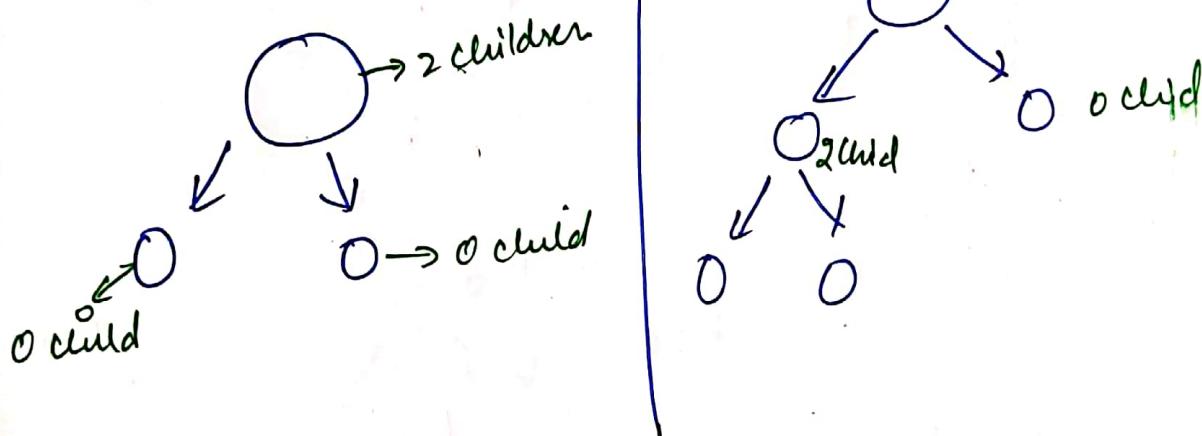
→ We use tree to represent this type of data.

Types of Binary Tree

① Full or Strict Binary Tree

→ All nodes have either 0 or 2 children

→ Eg:



⇒ Strict/Full Binary Tree have all of its nodes with a degree "0" or "2".
⇒ Each of its nodes either have 2 children or is a leaf node.

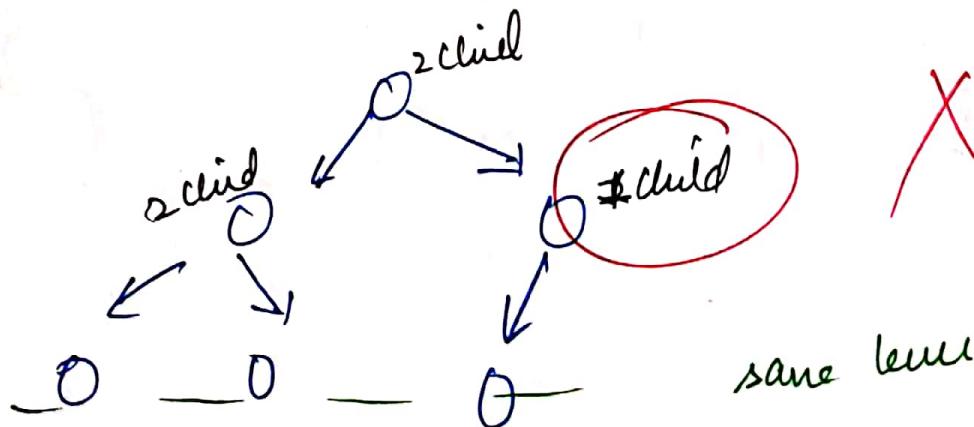
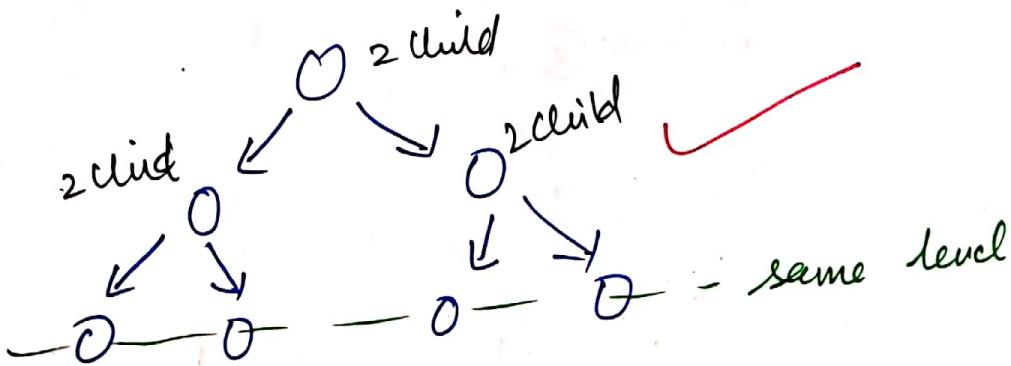
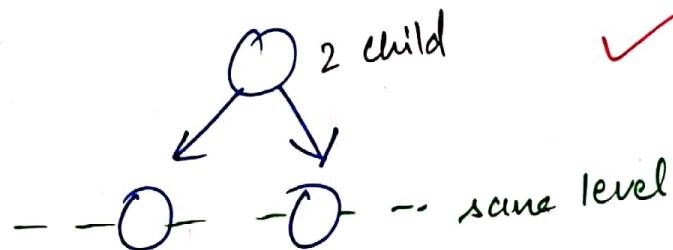
② Perfect Binary Tree

→ It has all its internal nodes with degree strictly "2" and has all its leaf nodes on the same level.

→ Internal nodes have 2 children

BOTH conditions should satisfy

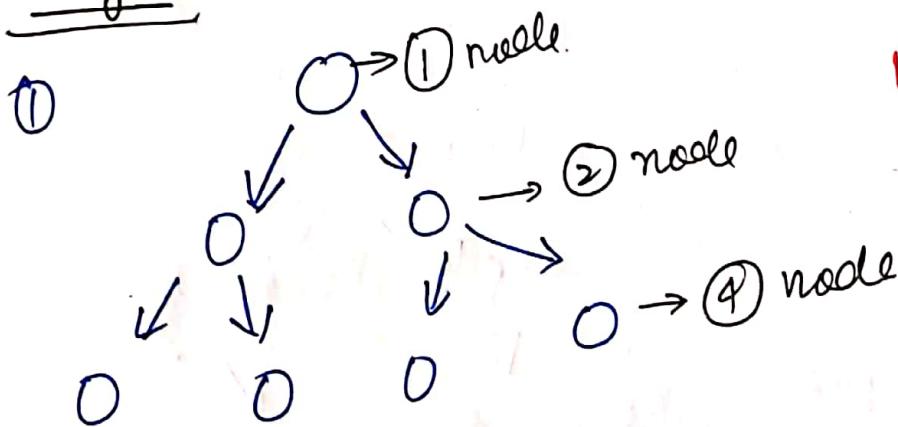
eg:



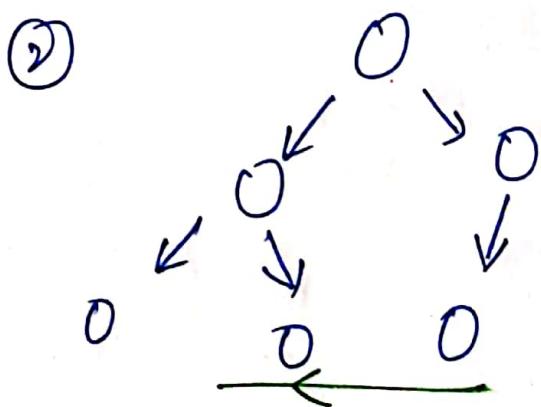
③ Complete Binary Tree

- It has all its levels completely filled except possibly the last level. ①
- if the last level is not completely filled then the last level's keys must be all left-aligned.
- It must have rounded tough to figure out.

→ Eg:-

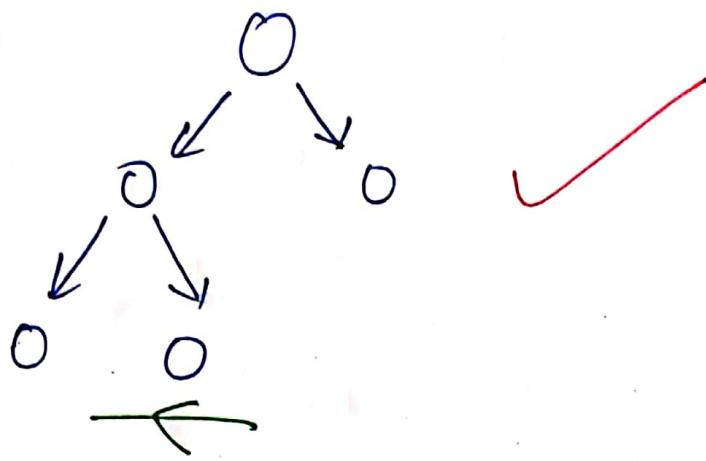


✓
all level
completely
filled



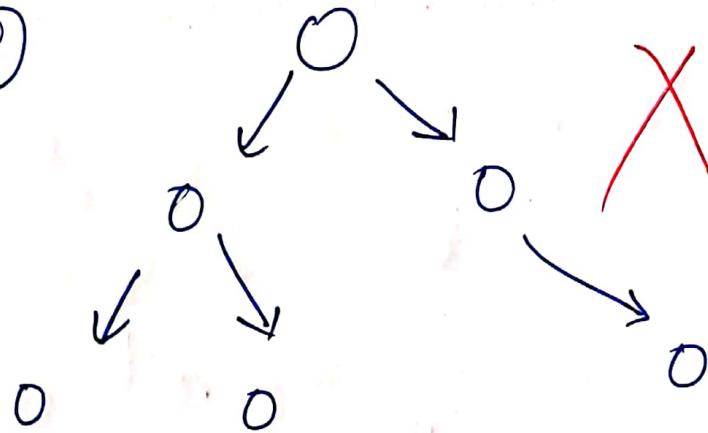
✓
last level node
aligned in left

③



(left aligned)

④



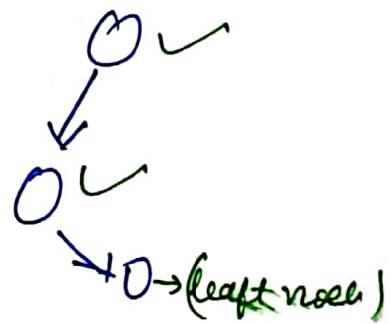
{ not left aligned
+
not all levels complete }

⑤

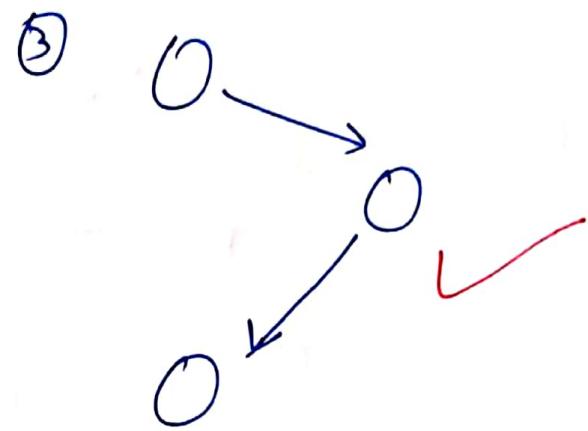
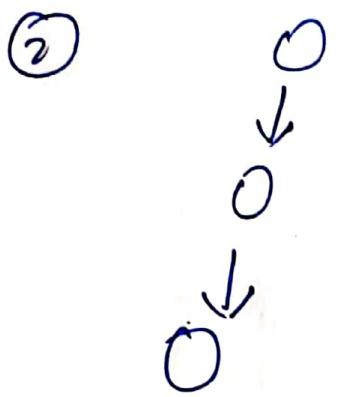
Degenerate Tree

→ Every parent node has just one child and that can either be to its left or right.

eg:



→ O → (left node)

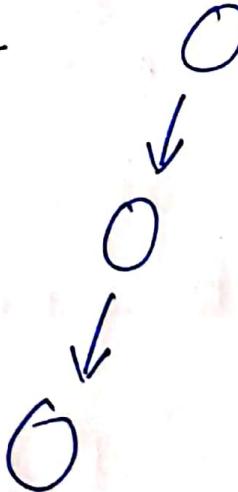


⑤ Skewed Tree:

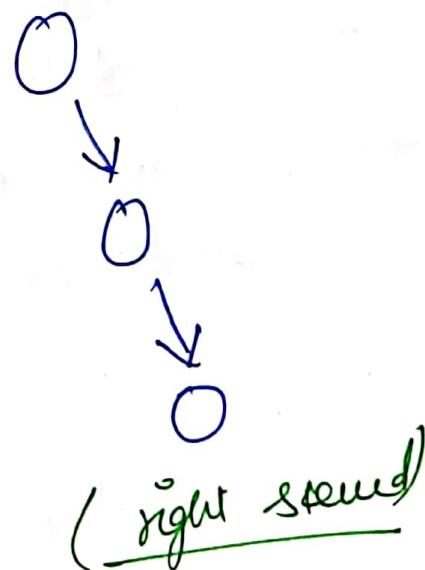
⇒ Every parent node has just a single child & that child should be strict to the left aligned. ⇒ left skewed tree

⇒ if right aligned ⇒ right skewed tree

Ex:-



(left skewed)



(right skewed)

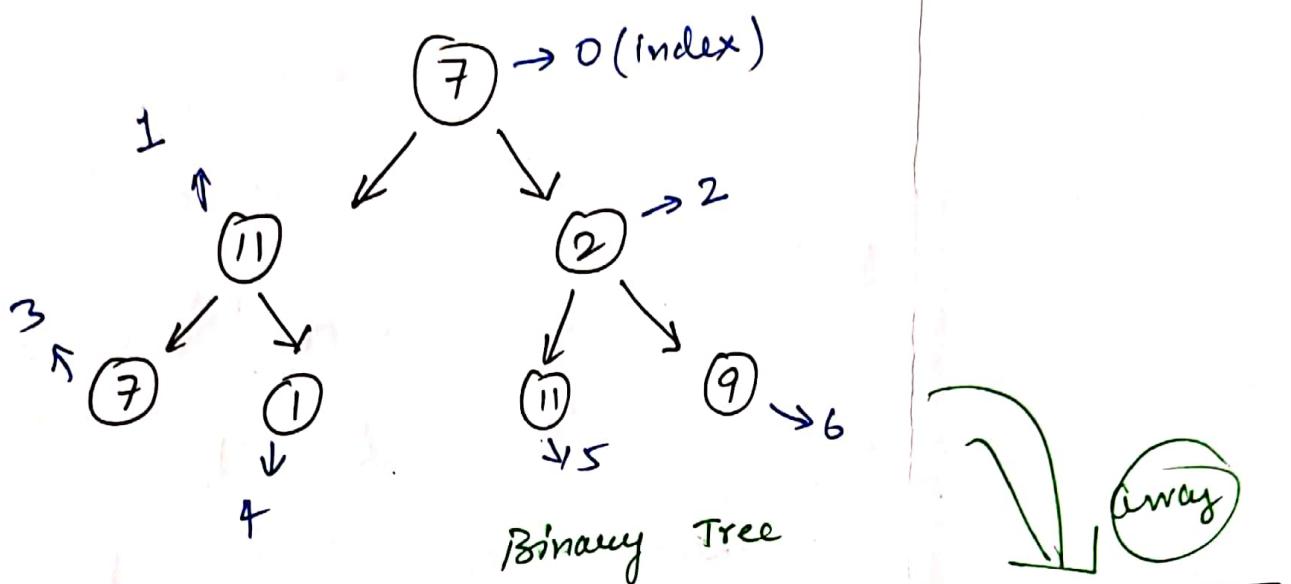
Video - 64

Representation of Binary Tree

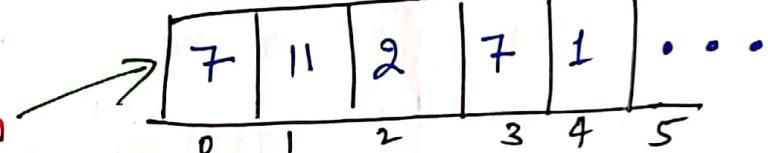
① Array Representation

- Array are linear data structures
- for array to function, their size must be specified before elements are inserted into them. And this counts as the biggest element of representing binary trees using arrays.
- suppose you declare an array of size 100 and after storing 100 nodes in it, you can't even insert a single element further, regardless of the spaces left in the memory and you have to copy the whole thing again to new array of bigger size.

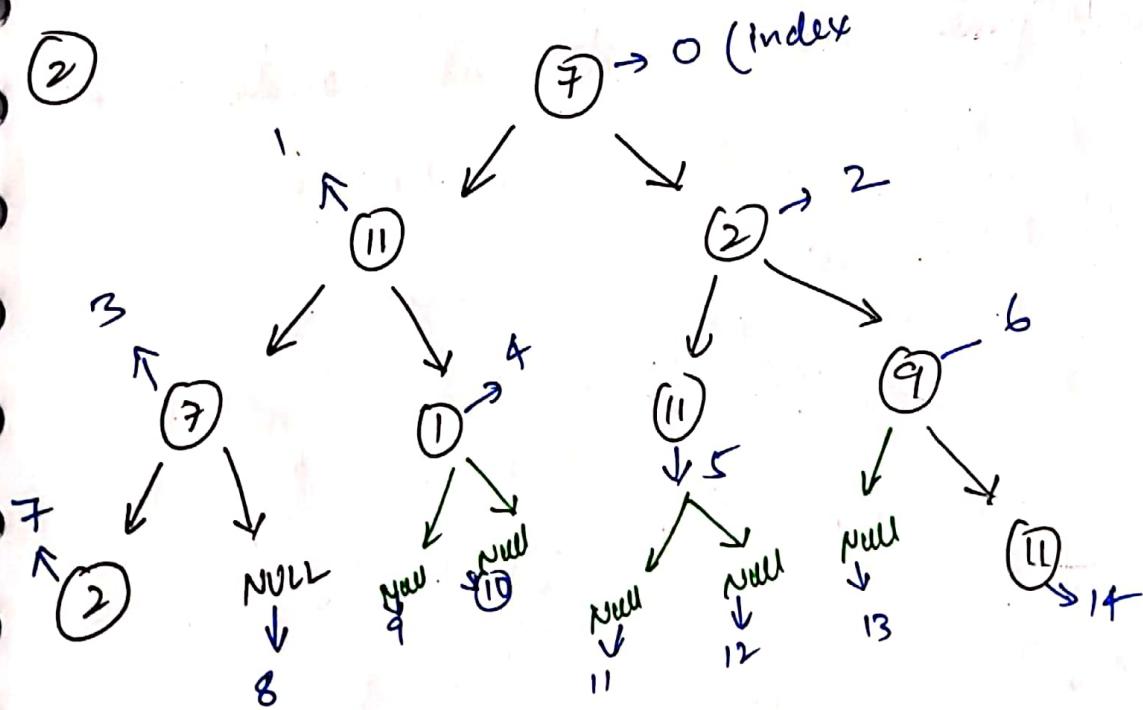
①



② Array Representation



②



⇒ if anything is not present ; make it
as "null" → null

Size of array = 15

Array is needed to store this tree!

* We use binary tree →

to make efficiently modifying tree

↳ search very fast

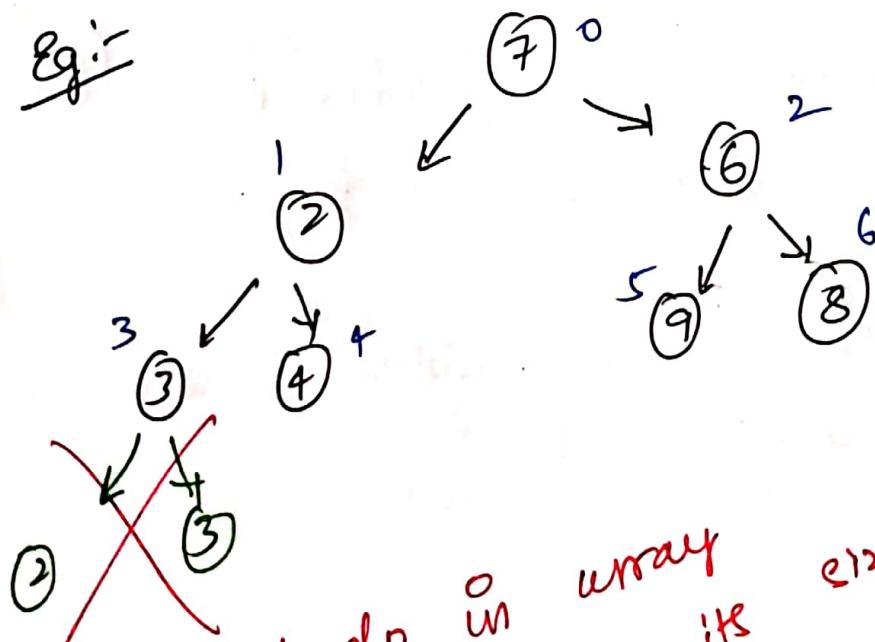
↳ easily remove/add element to it

↳ easily extendable

* Operation of binary tree in Array is very costlier.

↓
to extent this tree, we have to make a new array for that.

Eg:-



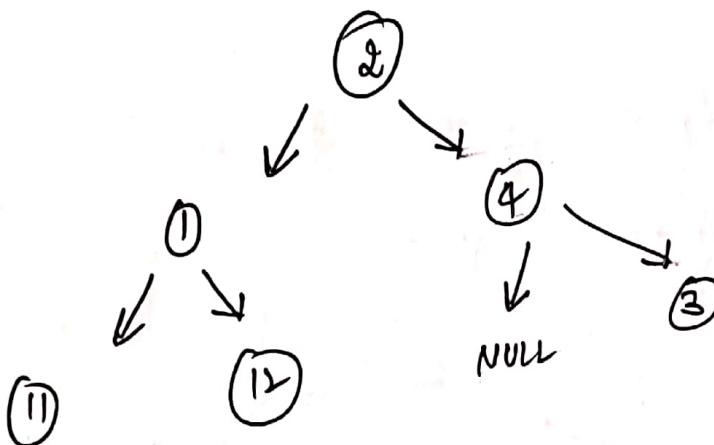
array \Rightarrow length = 7

can't do in array as its size = fixed!

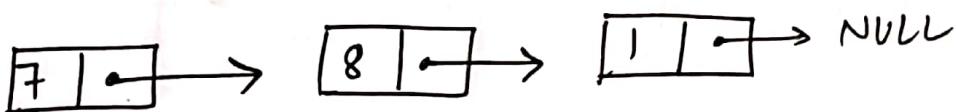
② Linked or ~~linked~~ List Representation

⇒ Most efficient method of representation of Binary Tree.

⇒ Null means → no element present at that node.



* Linked List → linear representation of data structure

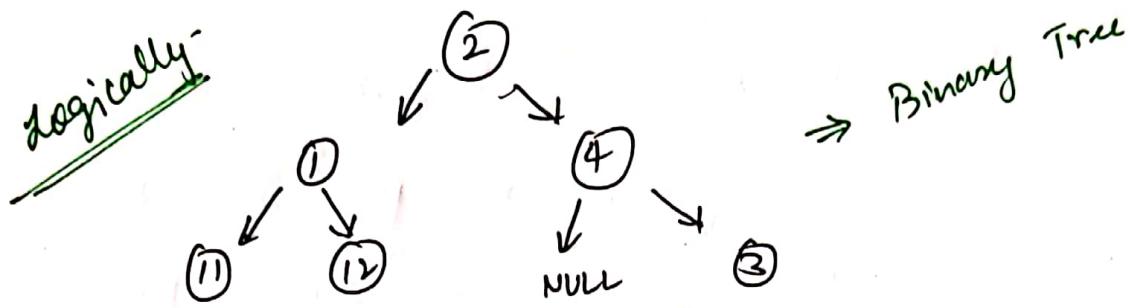


⇒ Not Linked List Representation

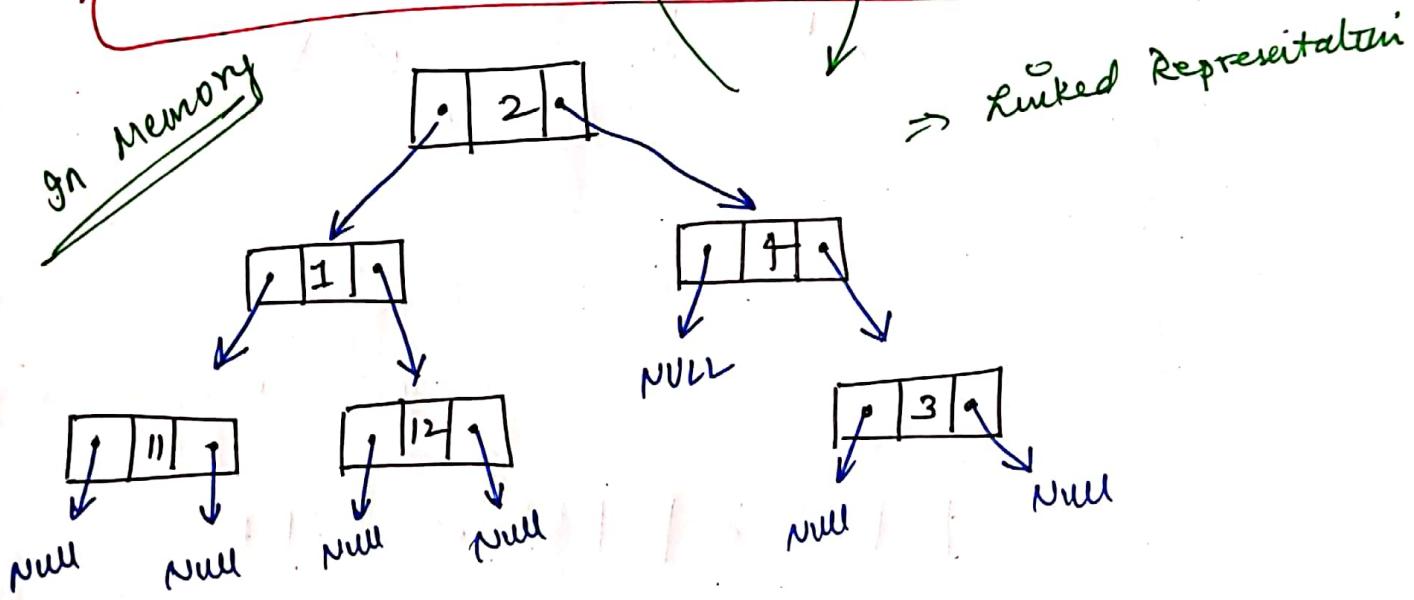
⇒ Only Linked Representation

⇒ Use Doubly linked list

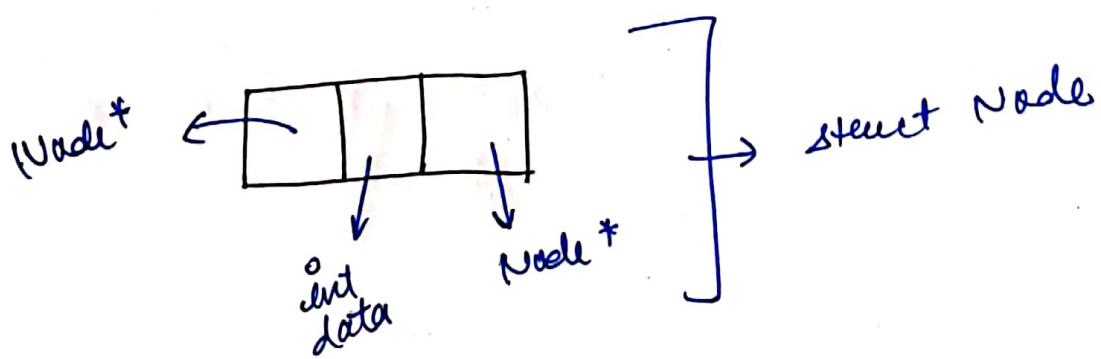
Linked Representation



Representation of Binary Tree Using Linked Representation



Note



Struct node

{

int data;

Struct node * left;

Struct node * right;

};

Structure of a node in C language

Video - 65

Linked Representation of Binary
Tree in C-language

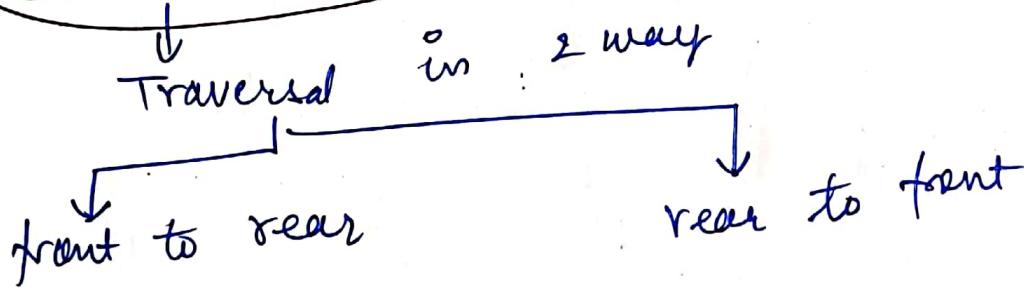
In V.S. Code

Video - 66

Traversal in Binary Tree

- ⇒ Traversal
- ⇒ visiting every node in the list at least once.

Linear Data Structure



Non-linear Data Structure

Traversal depends on type of data structure

3 Types of Traversal In

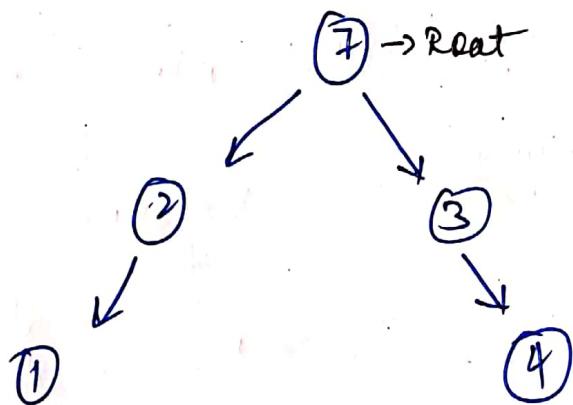
non-linear data structure

① Preorder
Traversal

② Postorder
Traversal

③ In-order
Traversal

Sample Tree →



① Pre-order Traversal

⇒ Here, 1st node you start with is the root node.

And, thereafter you visit the left sub-tree then the right sub-tree.

⇒ Root → left subtree → right subtree

② Post-order Traversal

⇒ first visit the left subtree, and then
the right sub-tree.
so, last node you'll visit is root node

⇒ left subtree → Right subtree → Root

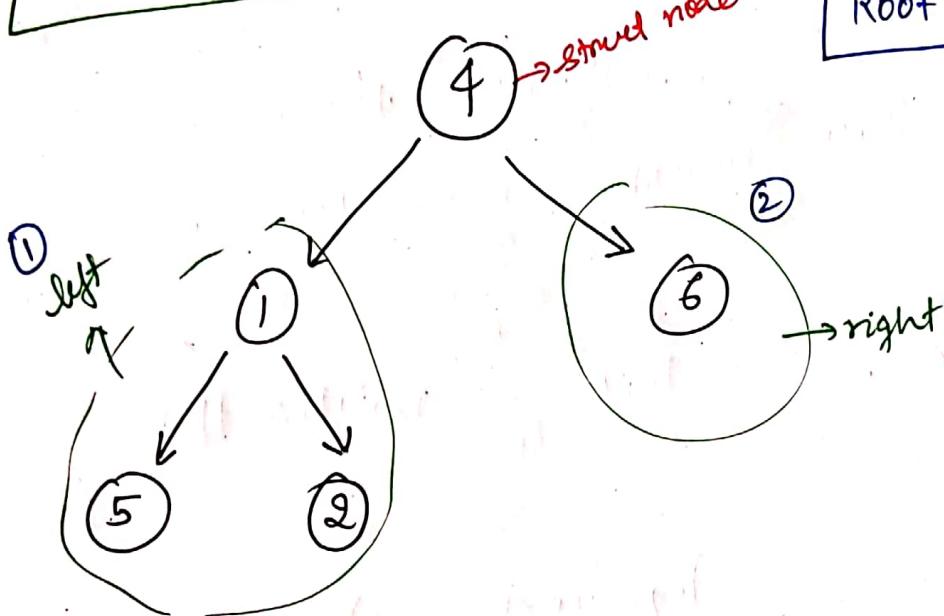
③ In-order Traversal

⇒ left subtree → root → Right subtree

Video - 67

Pre-Order Traversal in Binary Tree

Root → left → Right



4 → [] → 6

4 → [1 5 2] → 6

4 1 5 2 6

pre-order traversal
for above
binary tree

Code :

```
void preorder(struct node* root)
```

```
{
```

```
    if (root != NULL)
```

```
{
```

```
        printf(root->data);
```

```
        preorder(root->left);
```

```
        preorder(root->right);
```

```
}
```

```
else
```

```
{
```

```
    printf(" anything as per choice");
```

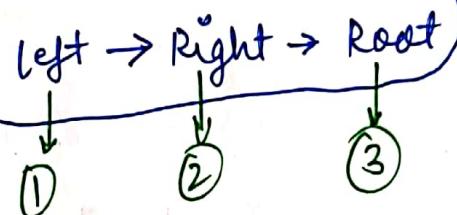
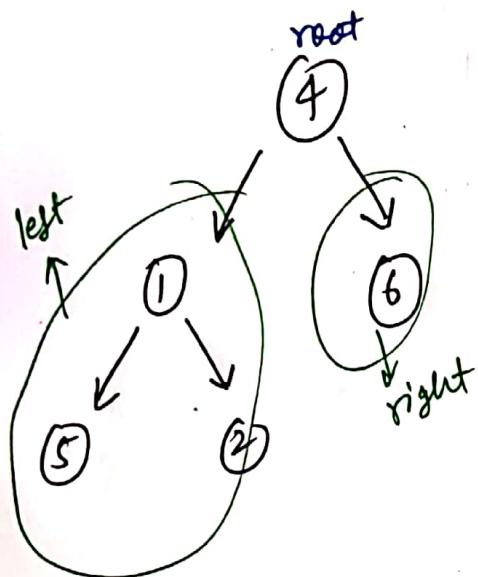
```
}
```

```
} // end of f
```

recursion
method

Video - 68

Post-Order Traversal in Binary Tree



[] → 6 → 4

5 2 7 → 6 3 4

5 2 1 6 4 → post order traversal

5 2 1 [] → 6 → 4
left right root

5 2 1 6 4 → post order traversal

```
void postorder(struct node* root)
```

```
{ if (root != NULL)
```

```
{ postorder (root->left);
```

```
postorder (root->right);
```

```
printf("%d", root->data);
```

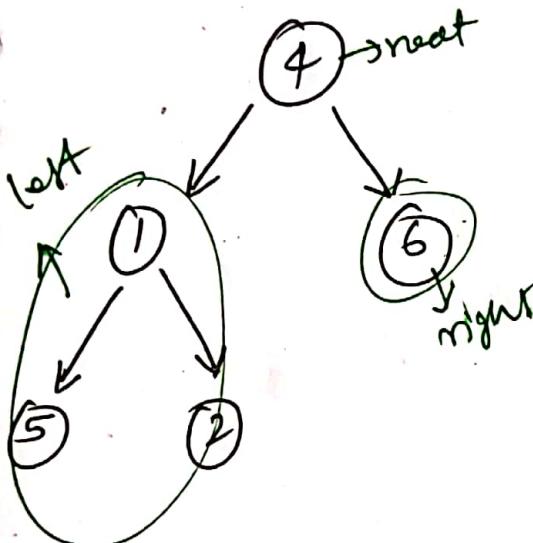
```
}
```

```
{}
```

Video - 69

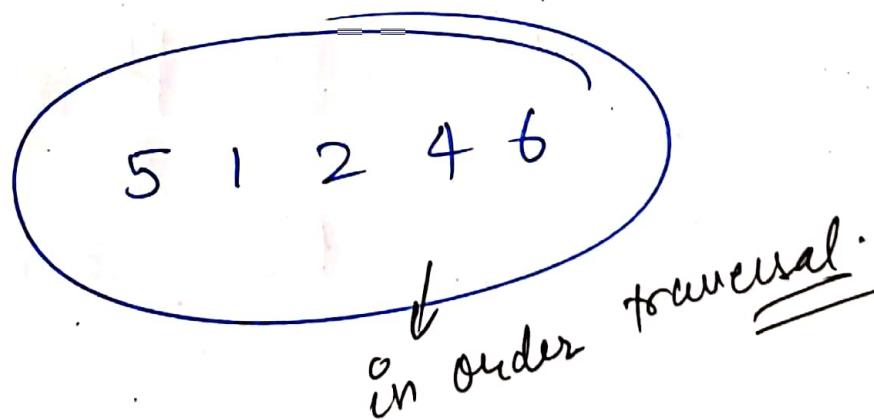
In-Order Traversal in Binary Tree

left \rightarrow Root \rightarrow Right



[] \rightarrow 4 \rightarrow 6

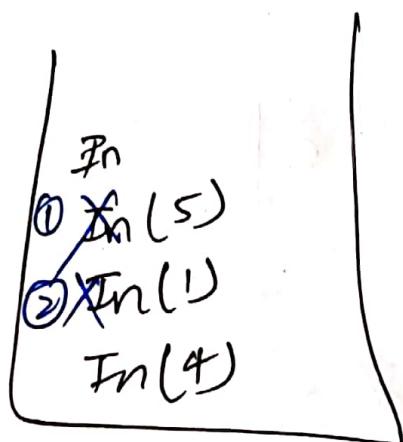
[5 | 2] \rightarrow 4 \rightarrow 6



void inorder (struct node* root)

```
{\n    if (root != NULL)\n        {\n            inorder (root->left);\n            cout << root->data;\n            inorder (root->right);\n        }\n}
```

Dry Run :-

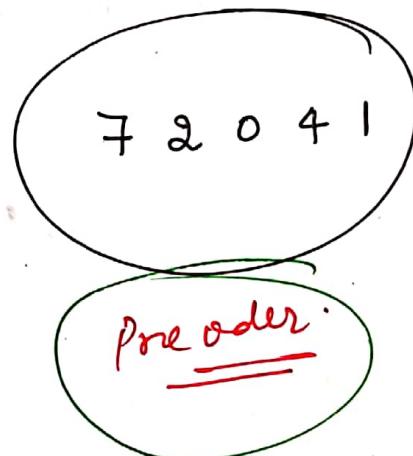
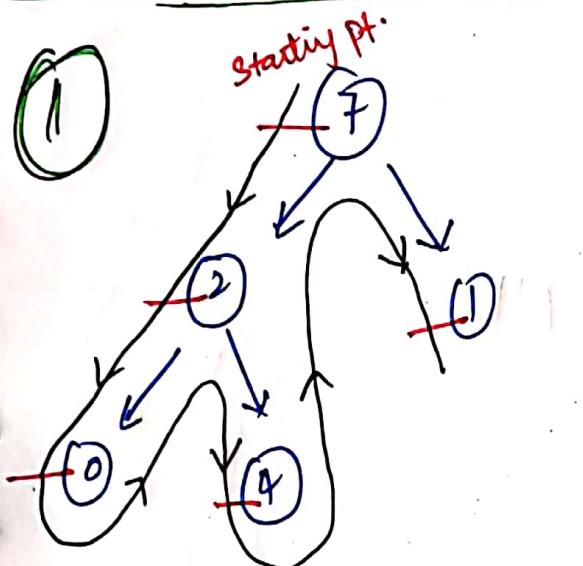


Std Out :-
[5 1 2] 4 6

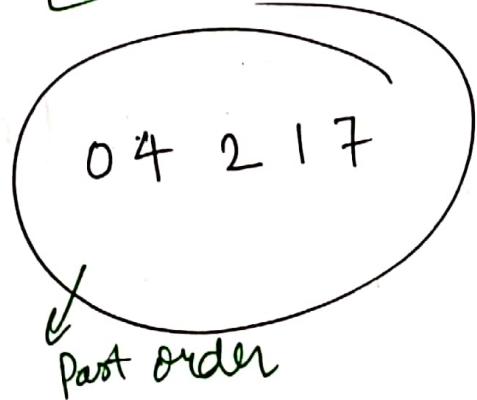
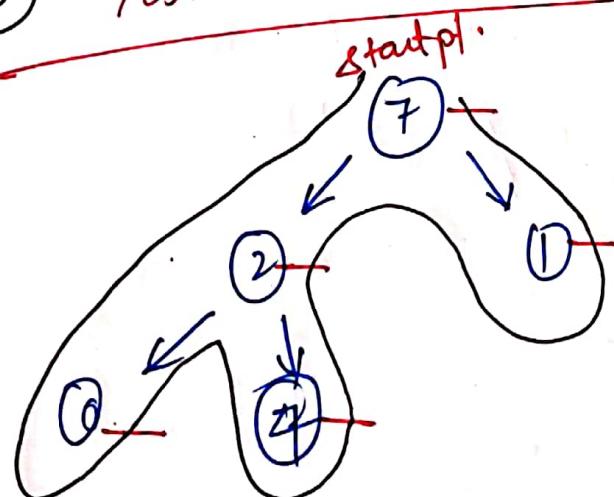
Video - 70

Trick to find :

Inorder, Preorder, Post Order Traversal

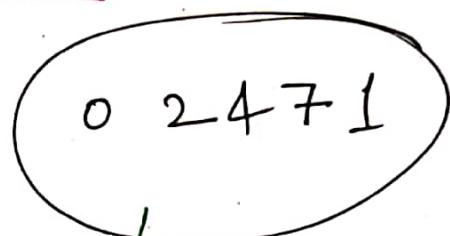
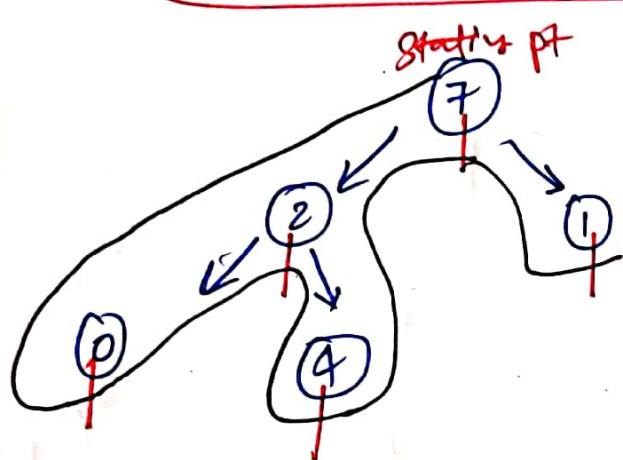


② Post Order Traversal



red ko cett kamine
pe uska data like
warna skip the data
at that point

(3) In Order Traversal



inorder traversal

Video - 71

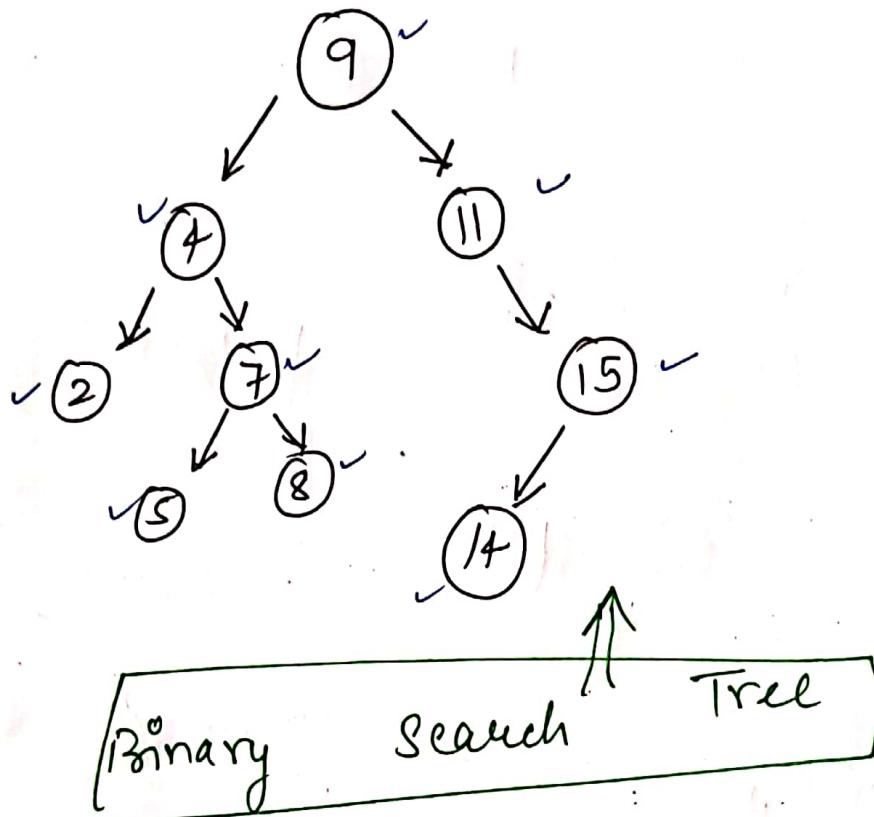
Binary Search Trees (B.S.T)

→ type of Binary trees

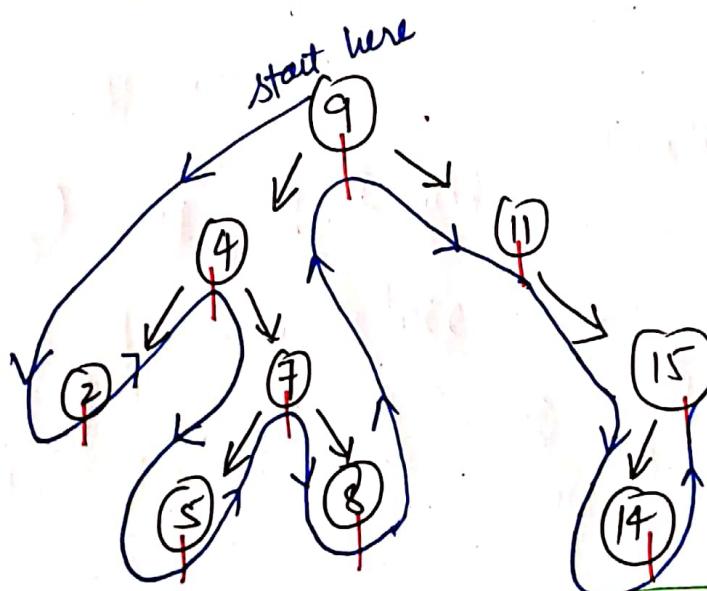
→ Properties

- ① All nodes of the left subtree are lesser.
- ② All nodes of the right subtree are greater.
- ③ Left & Right subtree are also binary tree.
- ④ There are no duplicate nodes.
(Same value as of node 78, 87)

- ⑤ Inorder Traversal of a BST gives an Ascending Sorted Array



Now, calculating In-order traversal for above Tree :-



2, 4, 5, 7, 8, 9, 11, 14, 15

∴ it is BST

if
Ascending Order.
sorted array.

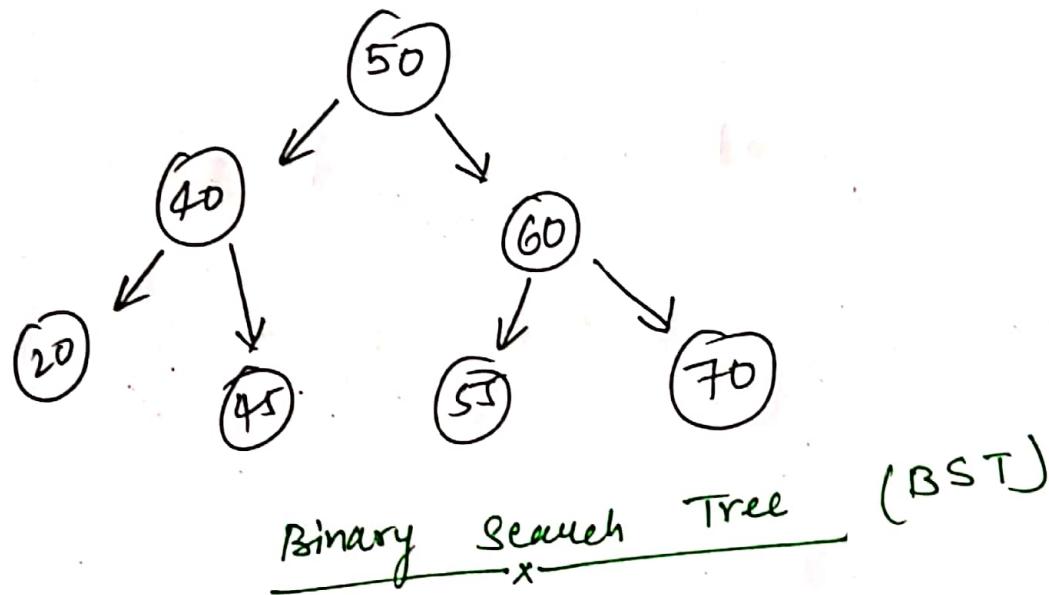
Video - 72

Checking if a binary tree is a
BST or Not !

Coding in V.S. Code

Video - 73

Searching in a Binary S. Tree



search $55 \rightarrow$ in above 'BST'

\Rightarrow BST helps in better searching, insertion & deletion of any element.

Comparisons in worst case \propto height of the tree

$$T(n) = h \times t$$

$$\log n \leq h \leq n$$

Best Case $\rightarrow O(\log n)$

worst case $\rightarrow O(n) \rightarrow$ (skewed tree)

C- Code

between type \rightarrow pointer

(75)

ptr

Node* Search (Node * root, int key)

{
if (root == NULL)

{
return NULL;
}

if (root->data == key)

{
return root;
}

elseif (root->data > key)

{
return Search (root->left);
}

else

{
return Search (root->right);
}

* Range of height of tree

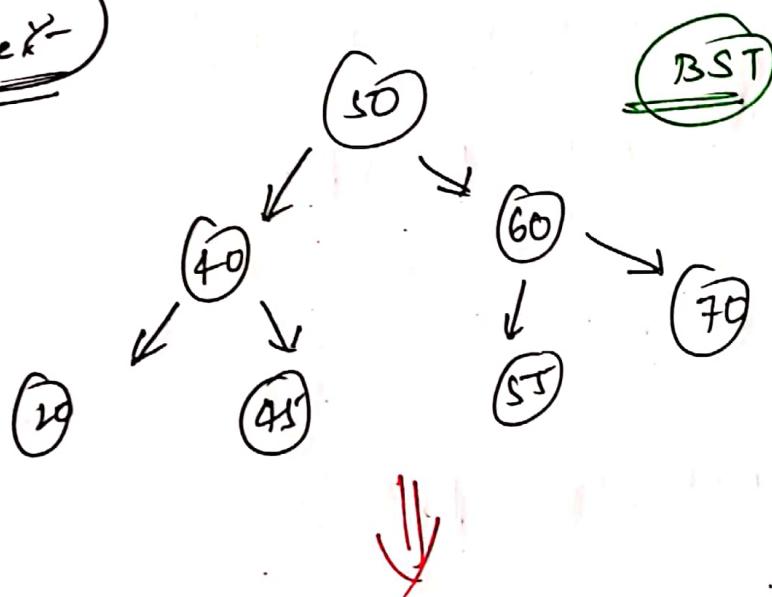
$\log n$

$$(\log n) \leq h \leq n$$

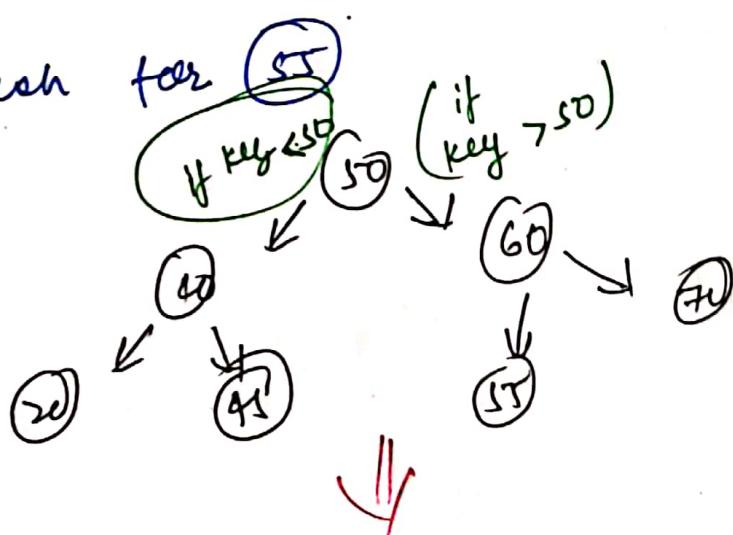
Best $T_n = O(\log n)$

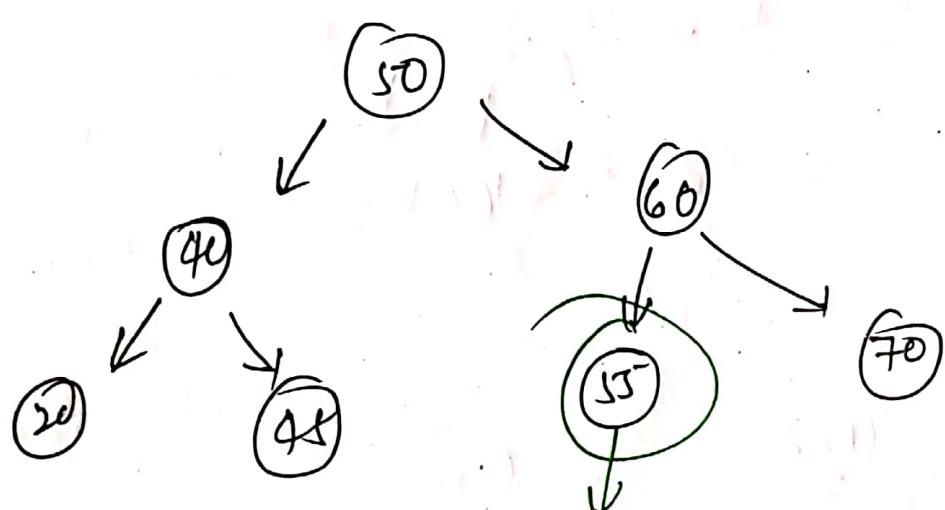
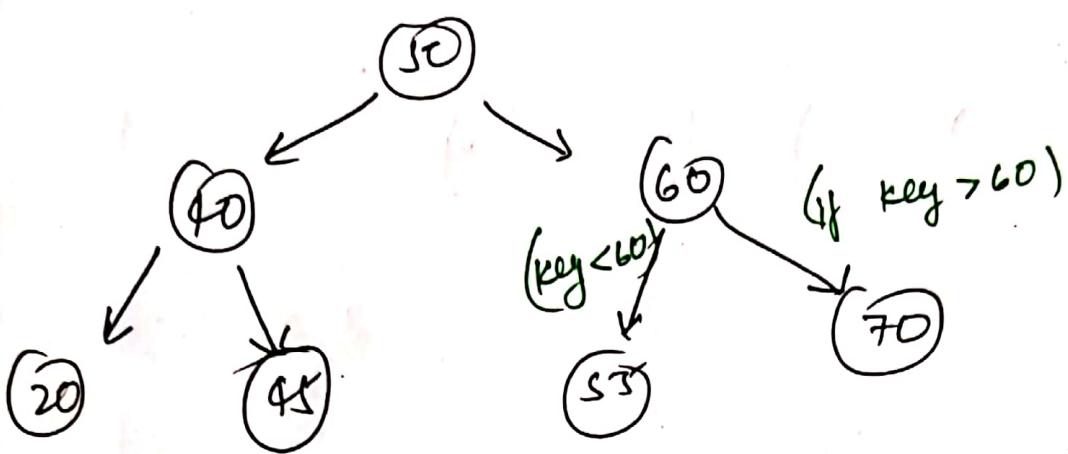
worst $T_n = O(n)$

Example :-



① search for 55





found the searched
element

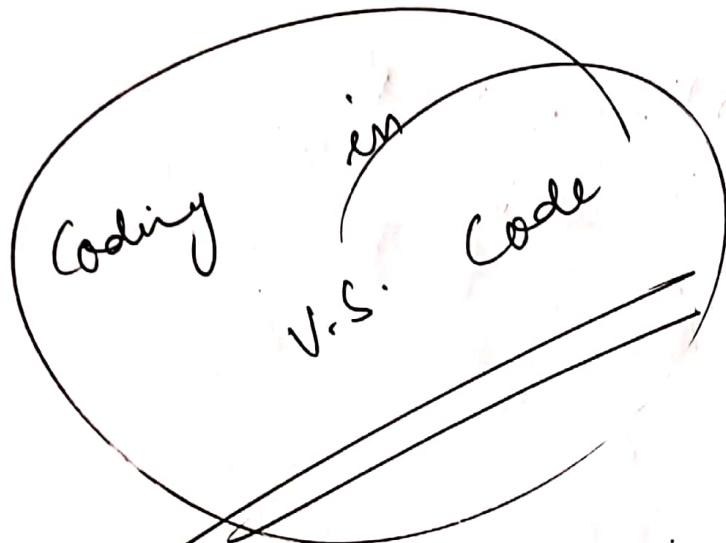
Video - 74

C - Code for Searching in B.S.T

Coding in V.S. Code

Video - 75

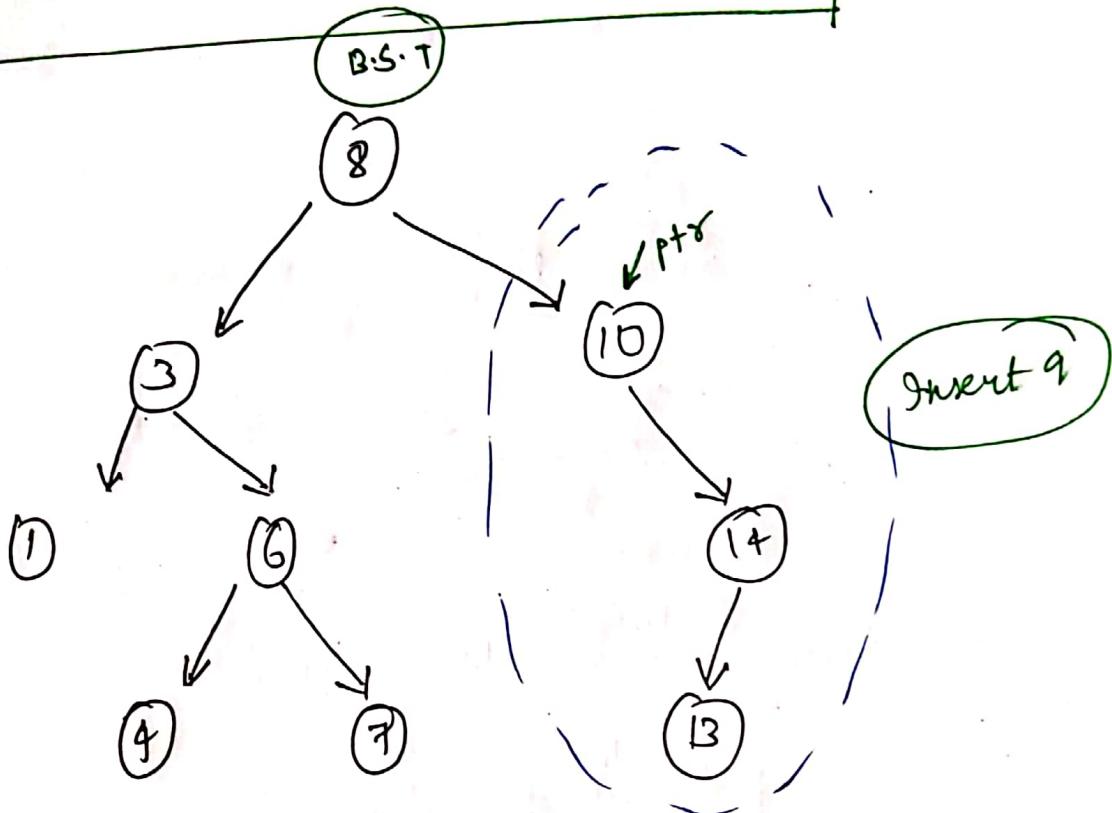
Iterative Search in BST



```
while (root != null)
{
    if ((root->data) == key)
    {
        return root;
    }
    else if (key < root->data)
    {
        root = root->left;
    }
    else
    {
        root = root->right;
    }
}
// while
return null;
```

Video - 76

Insertion in BST



(*) BST \rightarrow NO duplicates Allowed ! (*)

You simply start from the root node , and see if the element you want to insert is greater than or less than . And since $9 > 8$, we move to the right of the root . And the root element is the element = 10 .

and since this time $g < 10$, we move to the left. And since there are no elements to its left, we simply insert element 9 there.

* dynamically allocate the memory

node *
node
int
initialize

```
no = malloc (for a node)
no -> data = Key;
no -> left = NULL;
no -> right = NULL;
```

jiso insert $\frac{11}{2}$

Now, we have to do comparison

void Insert(Node *root, int key)

{
 Node * prev = NULL;

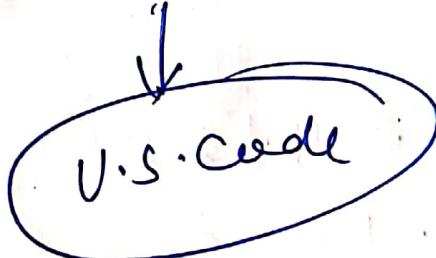
 Node * ptr =);

 while (root != NULL)

 created
 g
 pointer

```
prev = root ;  
if (key == root->data)  
    return ;  
elseif (key < root->data)  
{  
    root = root->left ;  
}  
else  
{  
    root = root->right ;  
}  
};
```

// Now link the nodes



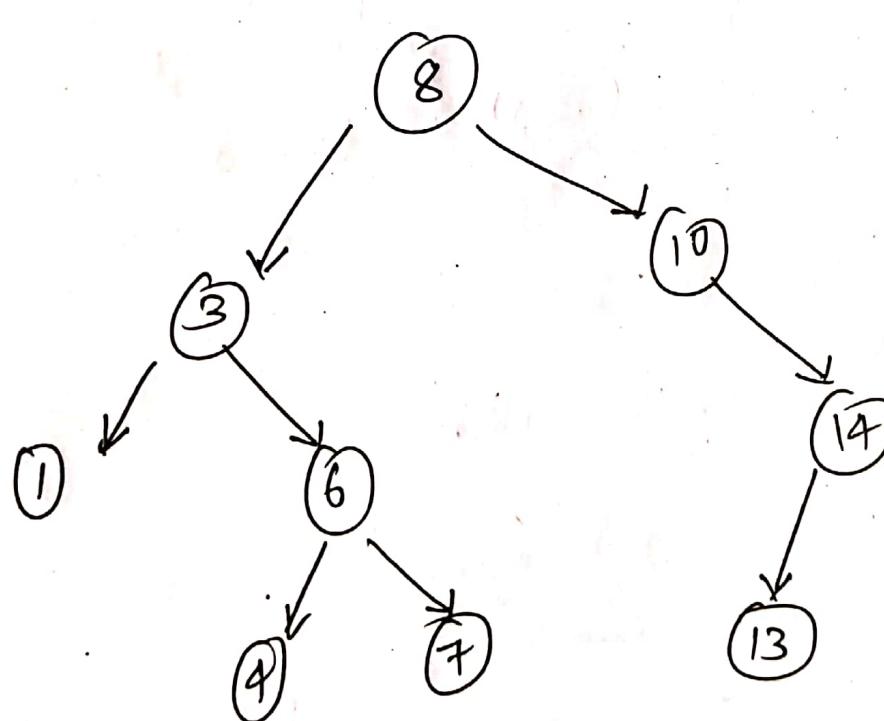
Video - 77

Deletion in BST

CASE-I :- The node is a leaf node

CASE-II :- The node is a non-leaf node

CASE-III :- The node is the root node.



Case-I : The node is a leaf node

Step 1 : Search the Node

Step 2 : Delete the Node

Done !!

delete $\rightarrow (1, +, 7 \text{ or } 13)$ \rightarrow there are leaf node

Case-II : The node is a non-leaf node

delete $\rightarrow (6)$ in BST (prev.)



node is not a leaf node, so you cannot just make its parent point to NULL, and get away with it. You have to even deal with the children of this node.

+ 1st thing : \rightarrow search for element "6"

+ 2nd thing : \rightarrow now dilemma is which node will replace the position of node = 6.

Care - I

Deleting a leaf node is the simplest care in binary search trees where the only thing you have to do is to search the element in the tree & remove it from the tree, and make its parent node point to NULL.



It's says, when you delete a node that it is not a leaf node, you replace its position with its Inorder predecessor or Inorder successor.

↓
write the Inorder traversal of the above tree, the node coming immediately before or after node = 6, will be the one replacing it.

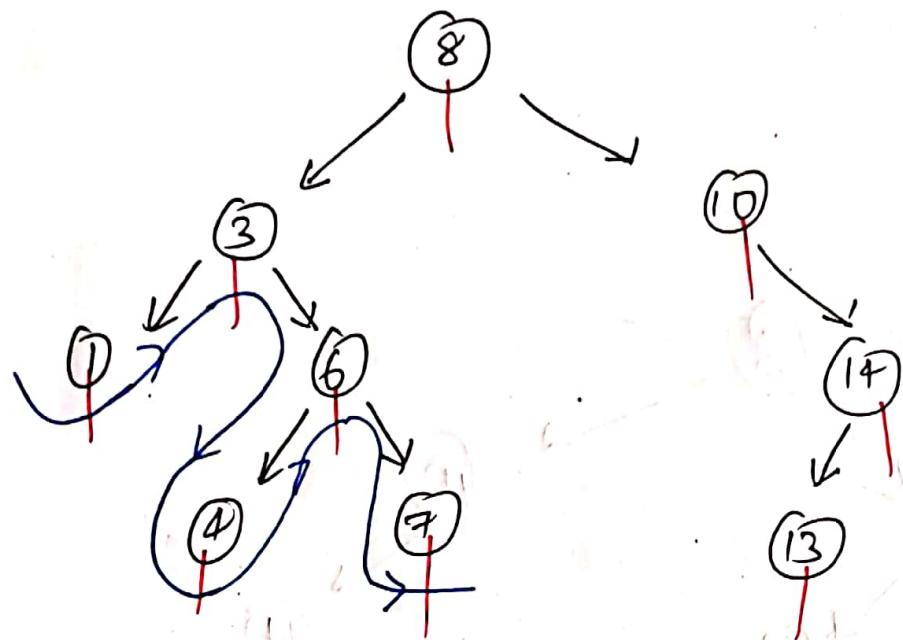
Inorder Traversal

1 → 3 → 4 → 6 → 7 → 8 → 10 → 13 → 14

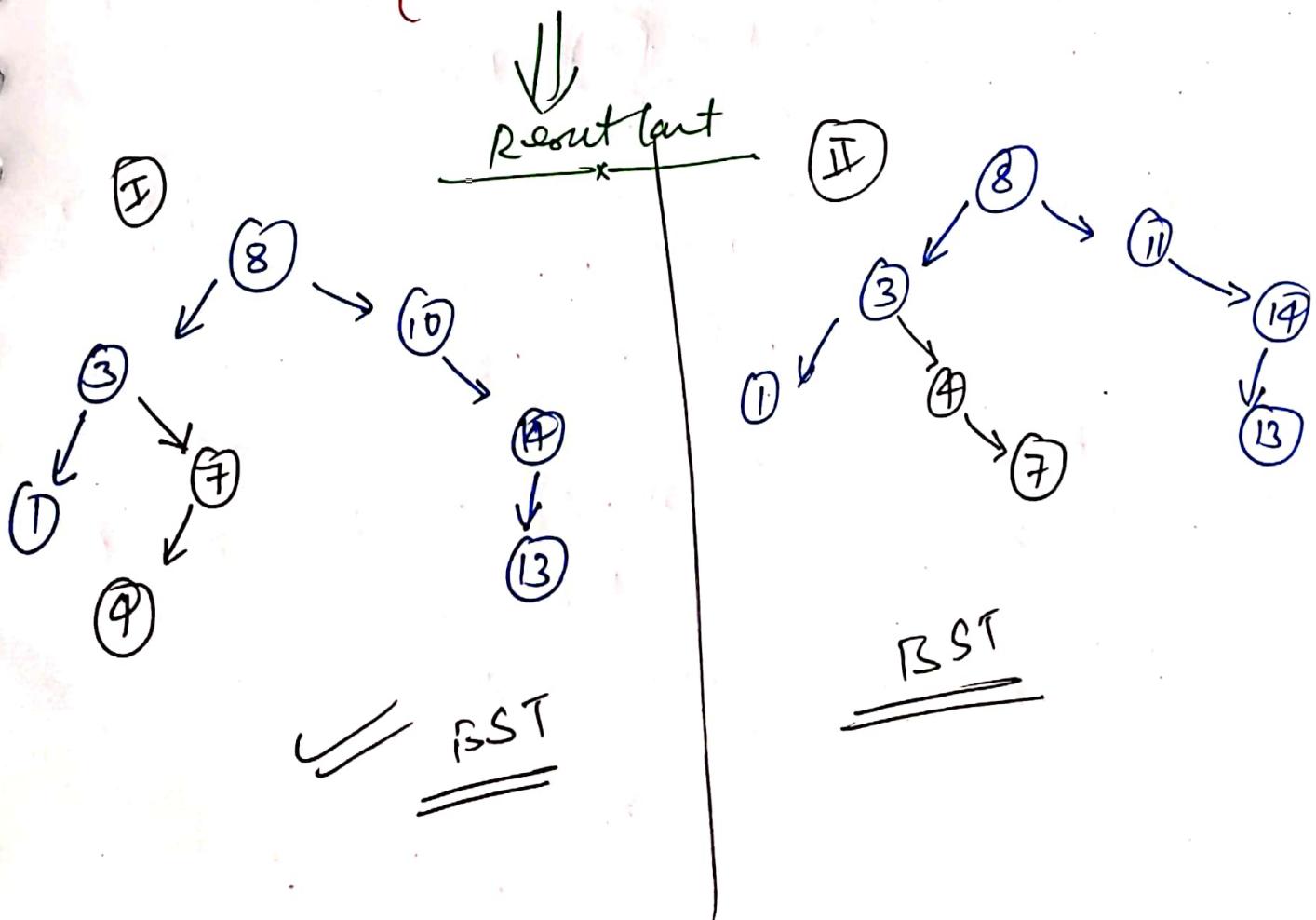
Now;

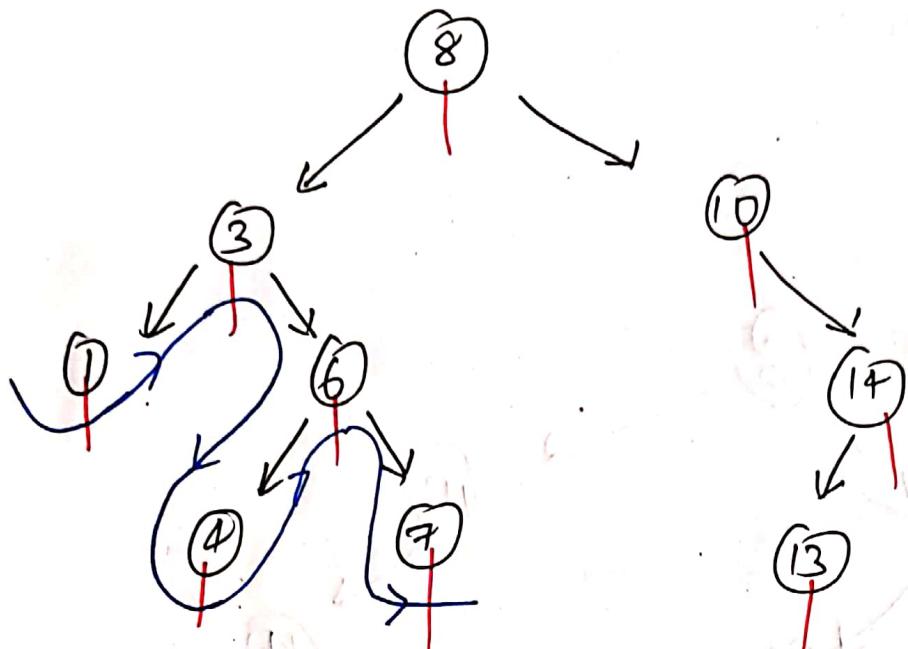
its predecessor = 4
successor = 7.

Now, substitute node = 6 with any of these nodes and the tree will still be a valid binary search tree.

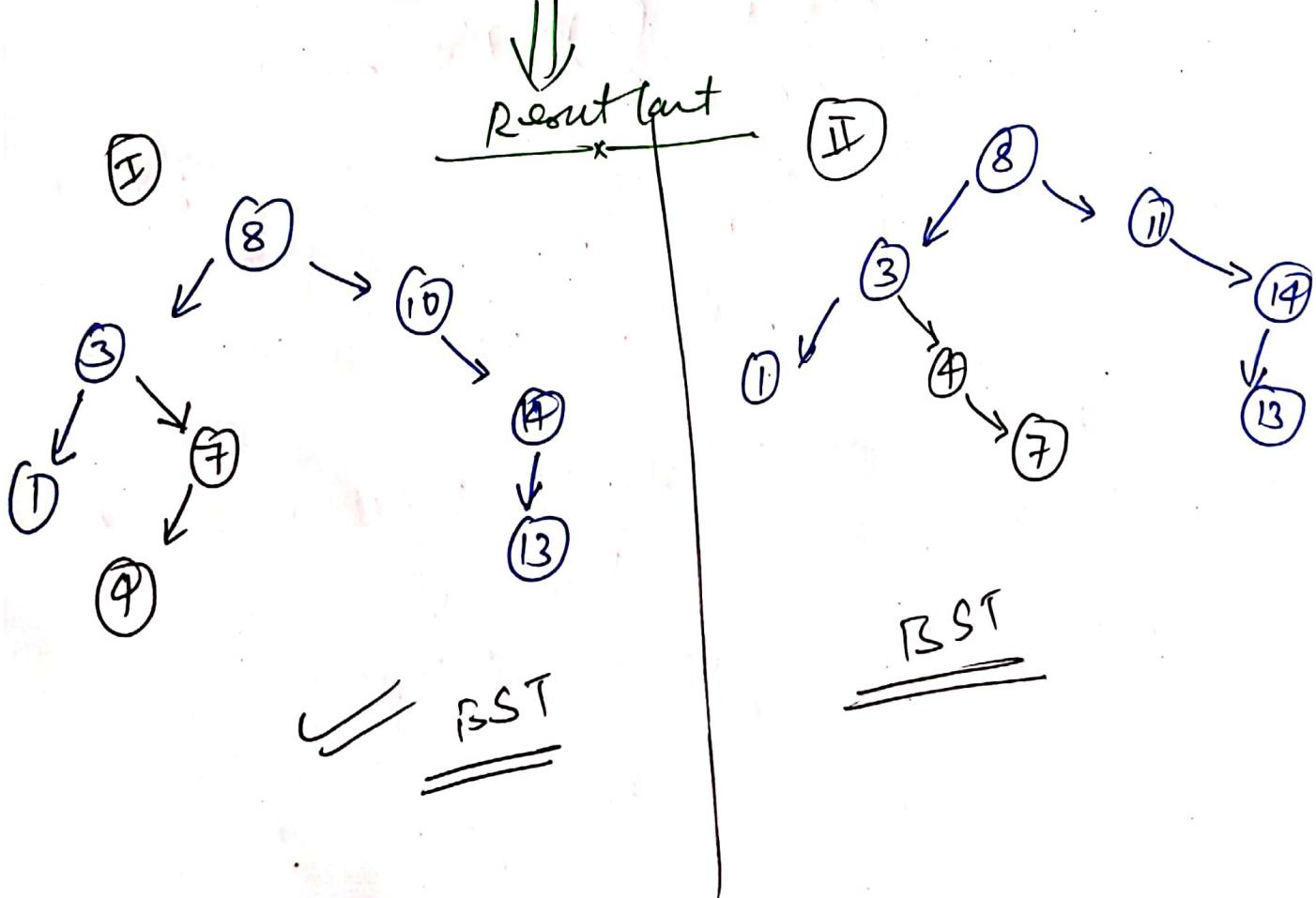


~~(1 2 3 4 5 6 7 8 9 10 11 12 13 14)~~
 (1 3 4 6 7)
 pre
 (inorder pre)
 post
 (inorder post)



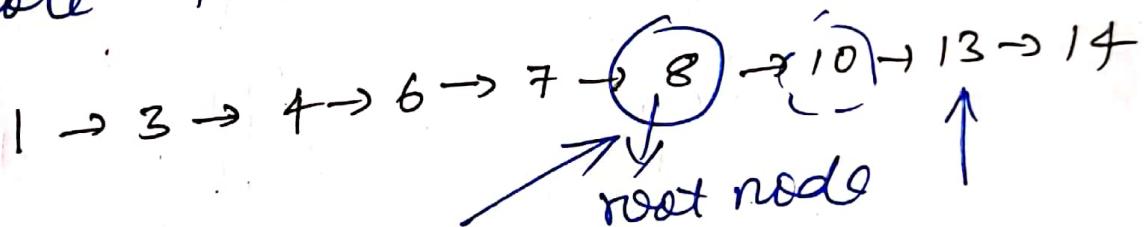


~~(12)~~ (1 3 4 6 7)
 pre (inorder pre) post (inorder post)



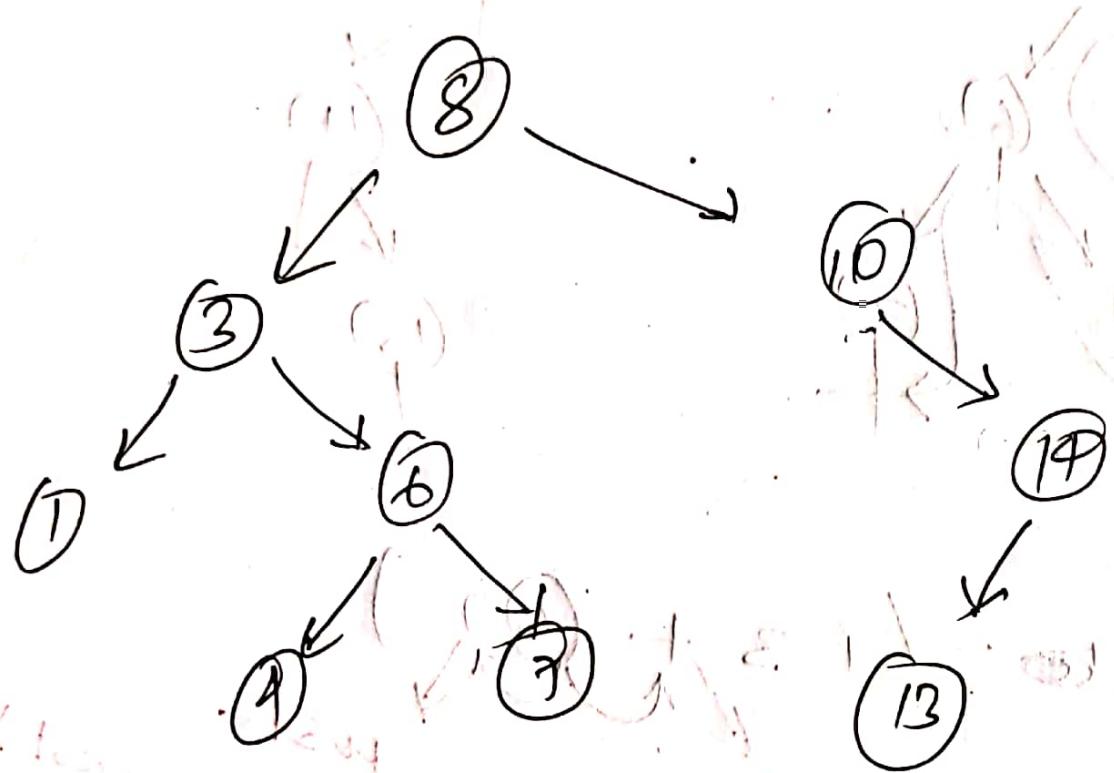
Case - III :- The Node is the root Node

- ① Write Inorder traversal of the whole tree.



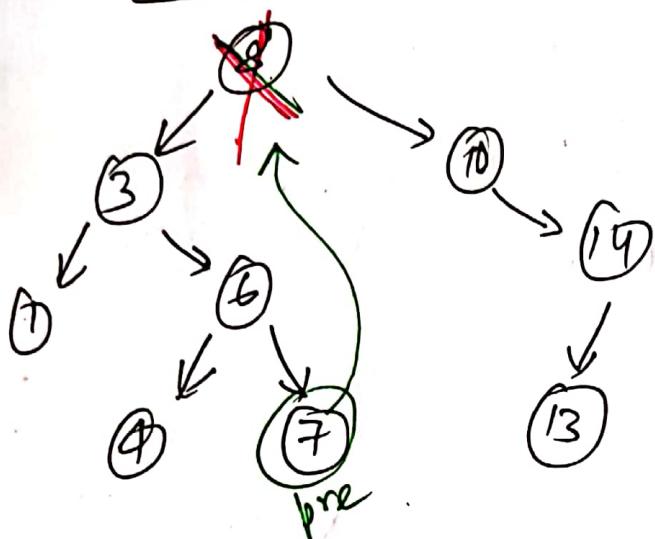
- ② in order pre : ⑦
in order post : ⑩

- ③ Now, replace '8' with 7 or 10 as per ur choice.
Make sure no empty nodes are there.

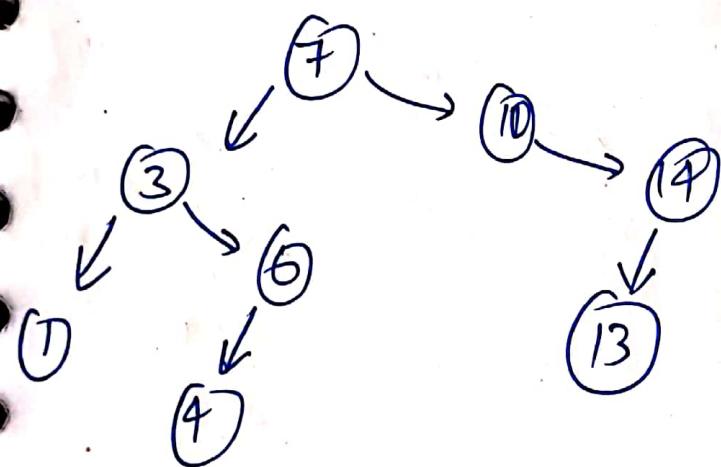


Case - I

Pre:



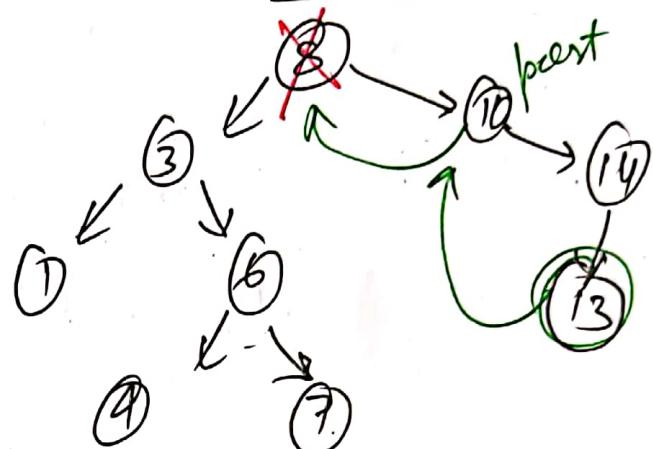
↓



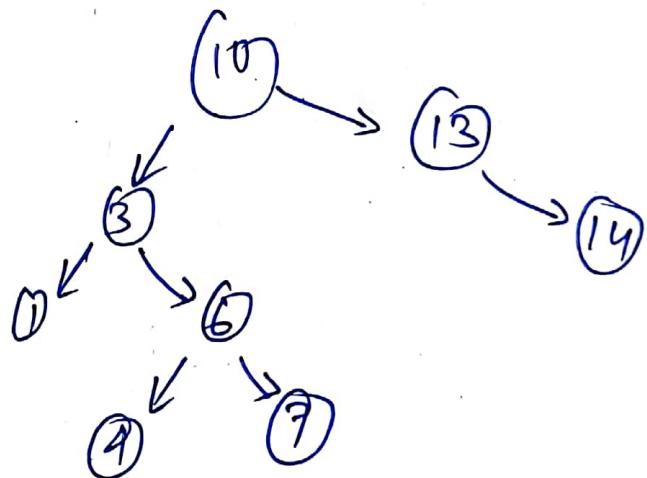
BST

Case - II

Post:



↓



BST

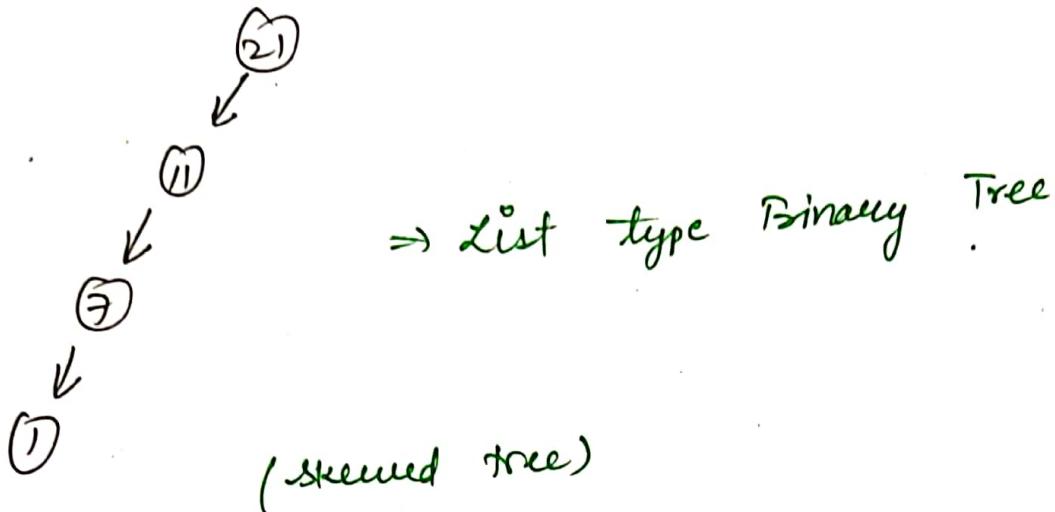
Video - 78

Code for deletion in B.S.T

Video - 79

AVL - Trees

- Q) Why do we need an AVL Trees?
- ① Almost all the operations in a BST are of order $O(h)$
 $h = \text{height of the tree}$
 - ② If we don't plan our tree properly, this height can get as high as n .
 $n = \text{no. of nodes in BST}$
 \Rightarrow Skewed tree
 - ③ To guarantee an upper bound of $O(\log n)$ for all tree operations, we use balanced trees.



Ques: What is an AVL Tree?

- ① Height balanced BST
- ② Height diff. b/w height of left & right subtree for every node is less than or equal to 1.
- ③ Balanced factor = $(\text{height of right subtree}) - (\text{height of left subtree})$
- ④ can be $-1, 0$ or 1 for a node to be balanced in a BST.
- ⑤ can be $-1, 0$ or 1 for all nodes of AVL Tree.

$$|BF| \leq 1 \quad BF = \{-1, 0, 1\}$$

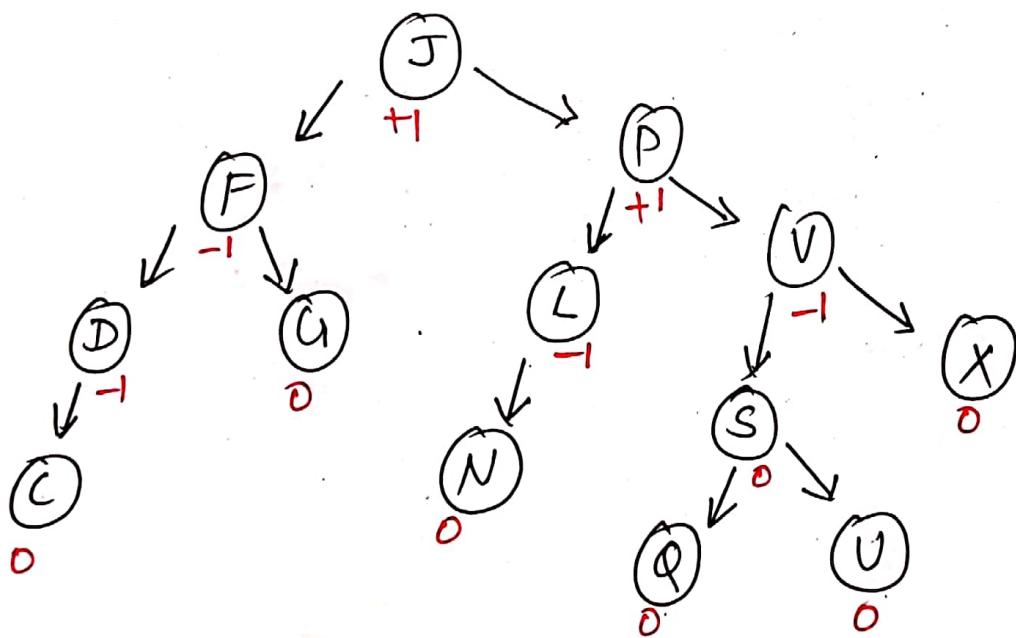
$$-1 \leq BF \leq 1$$

* Use BST - when you are doing lot of Insertion

* To balance a tree

↓
we perform "Rotations".

AVL Tree



* No. represent \Rightarrow Balanced factor

Video - 8D

Insertion & Rotation in AVL Tree

LL Rotation in AVL Tree

⇒ we do rotation in AVL Tree
b/c to make $|BF| \leq 1$



①

7

5

$|BF| \leq 1$

⇒ AVL Tree → Yes

Now, 7 wants to reduce its BF

rotation - clockwise

②

7

5

7

rotation done

5

7

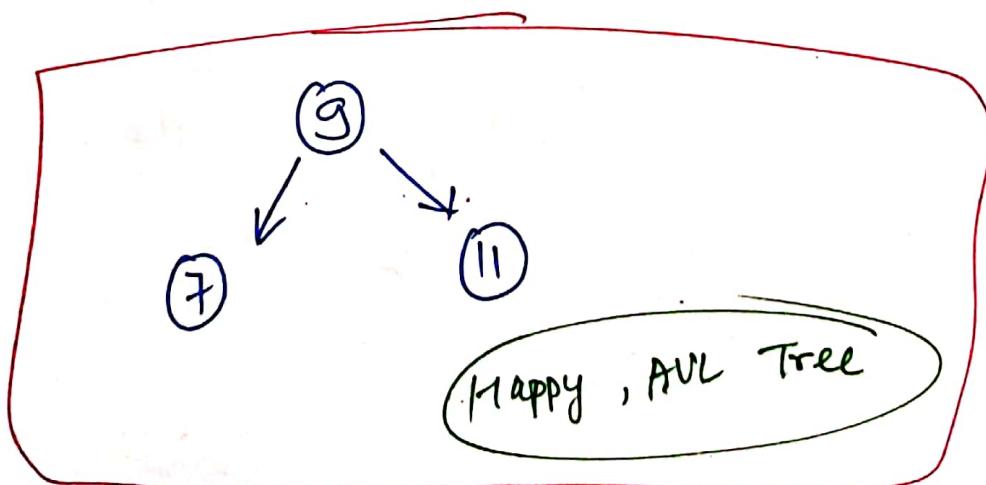
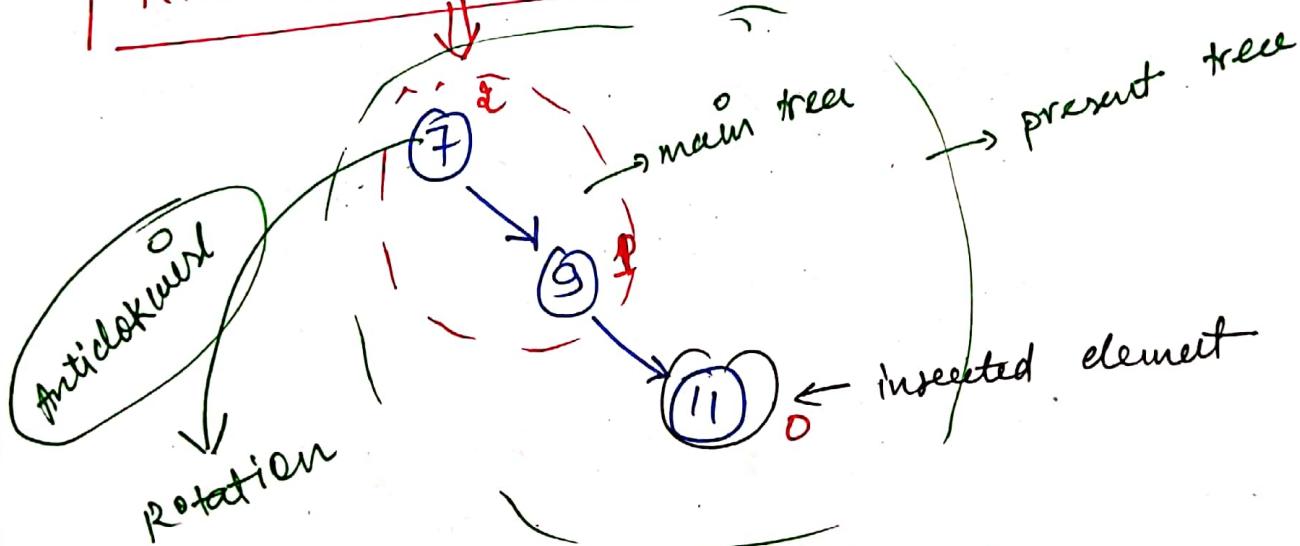
(Happy → AVL Tree)

LL Rotation

→ We inserted the new element to the left subtree of the Root. In this rotation technique, you just simply rotate your tree 1-time in the clockwise direction.

Now, Our tree got Balanced Again !!

RR Rotation in AVL Tree

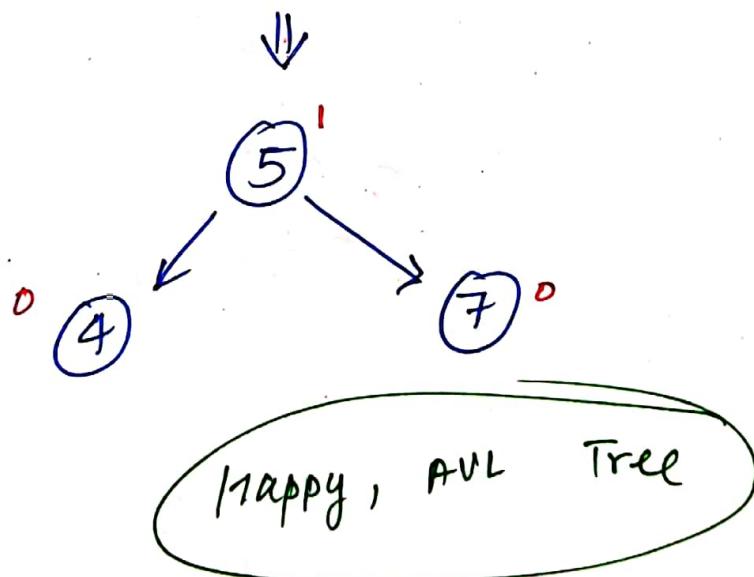
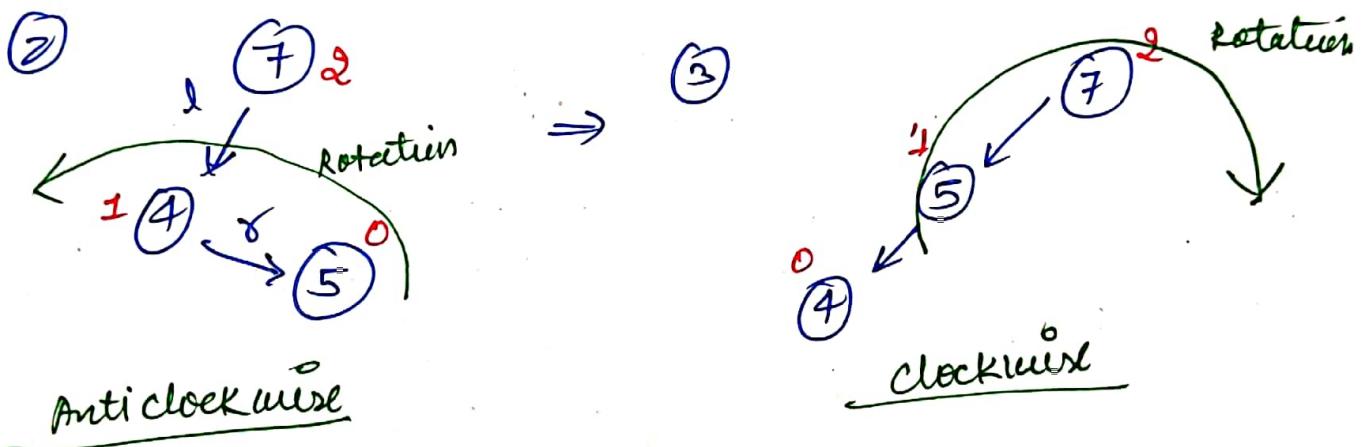
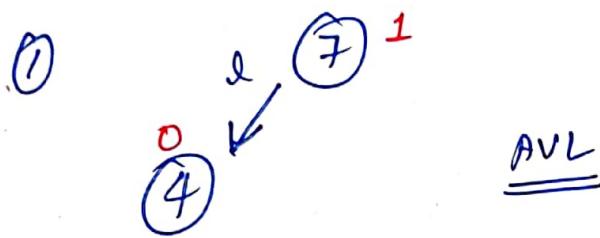


RR

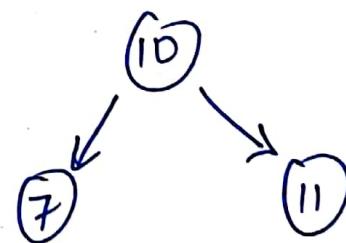
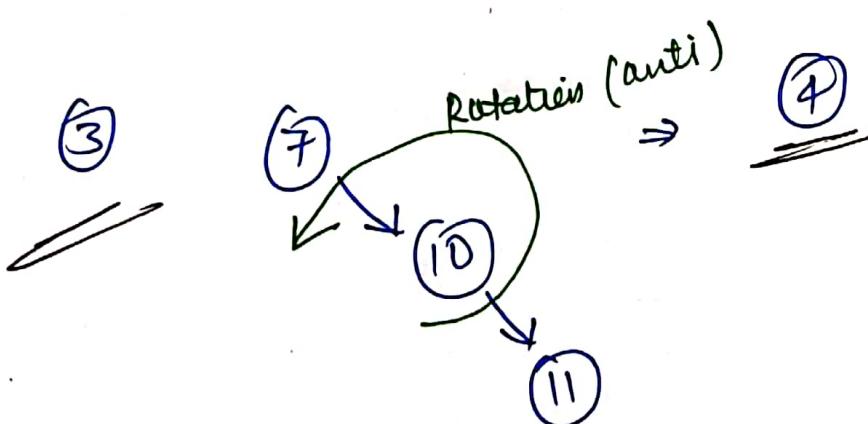
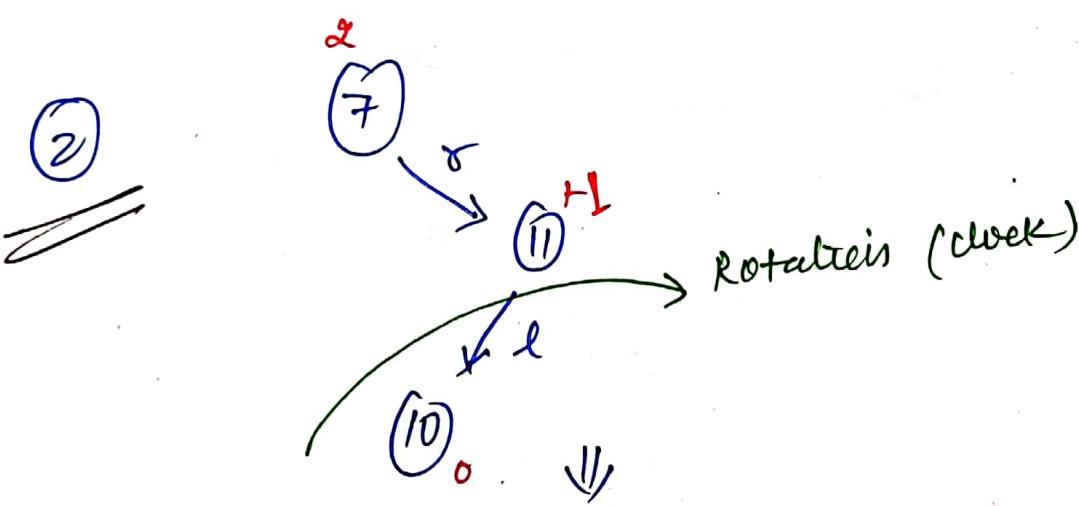
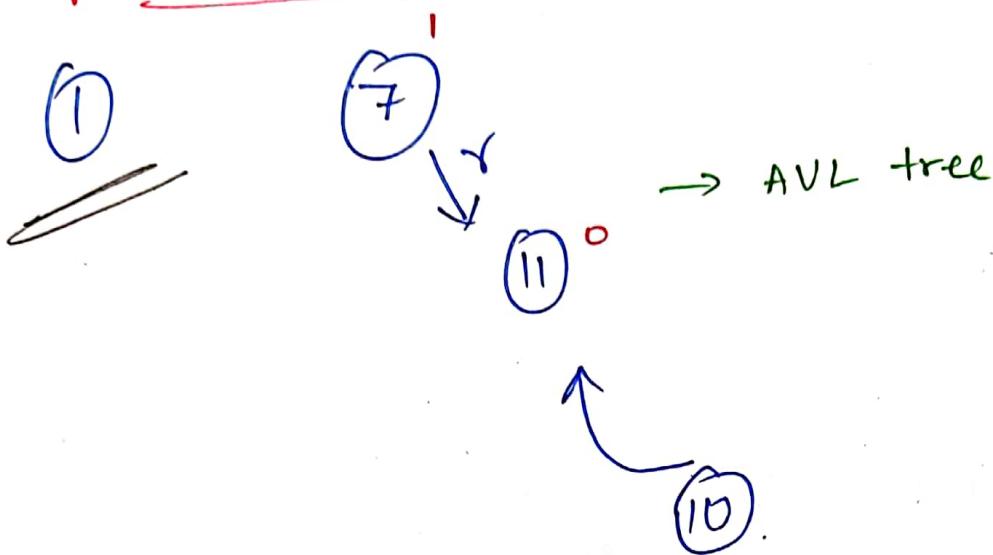
→ (+) Inserted new element to the right sub-tree of the root.

In this, you simply rotate your tree one time in an anti-clockwise direction #

LR Rotation in an AVL Tree



RL rotation in an AVL Tree



Video - 81

* Rotation in AVL Trees

with Multiple Nodes

Rotate Operations

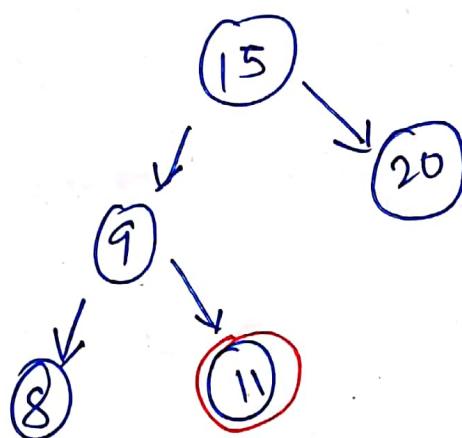
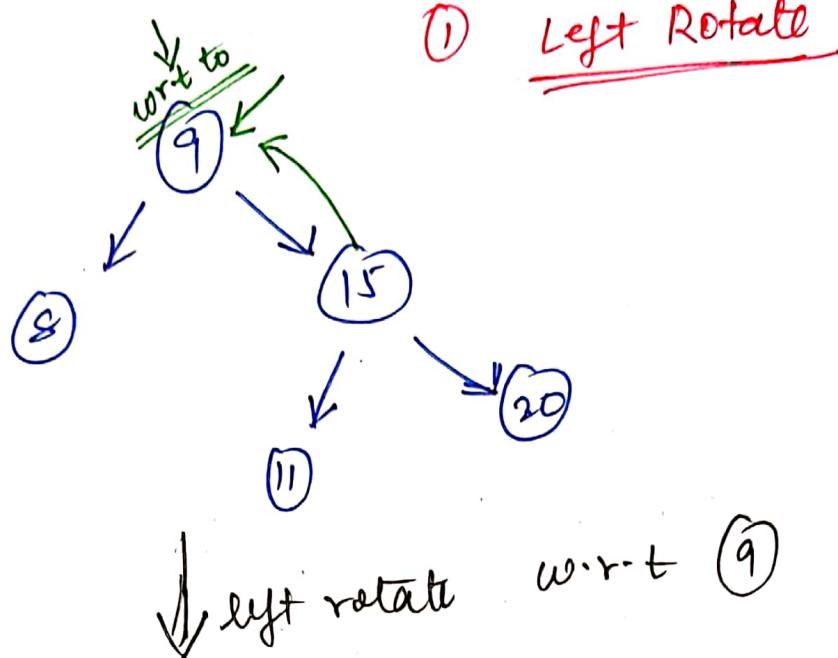
→ we can perform rotate operations to balance a BST such that the newly formed tree satisfies all the properties of a BST.

2 Basic rotate operations ⇒

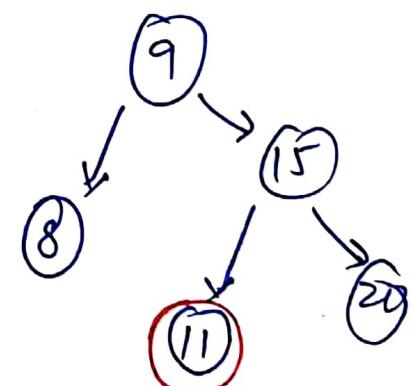
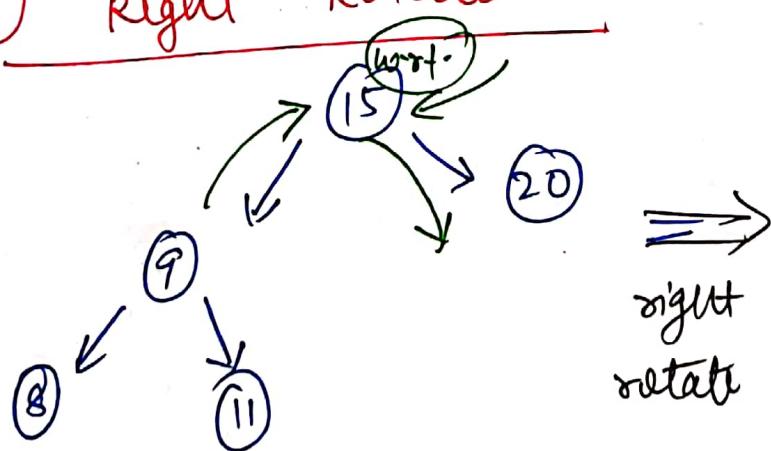
(1) Left Rotate wrt a node
• Node is moved towards the left.

(2) Right Rotate wrt a node
• Node is moved towards the right.

Eg:-



② Right Rotate



Tree 1

left rotate

Tree 2

right rotate

Balancing AVL Tree after Insertion

① For a left-left insertion

Right rotate once w.r.t the 1st imbalanced node

② For a Right-Right insertion

Left rotate once w.r.t the 1st imbalanced node

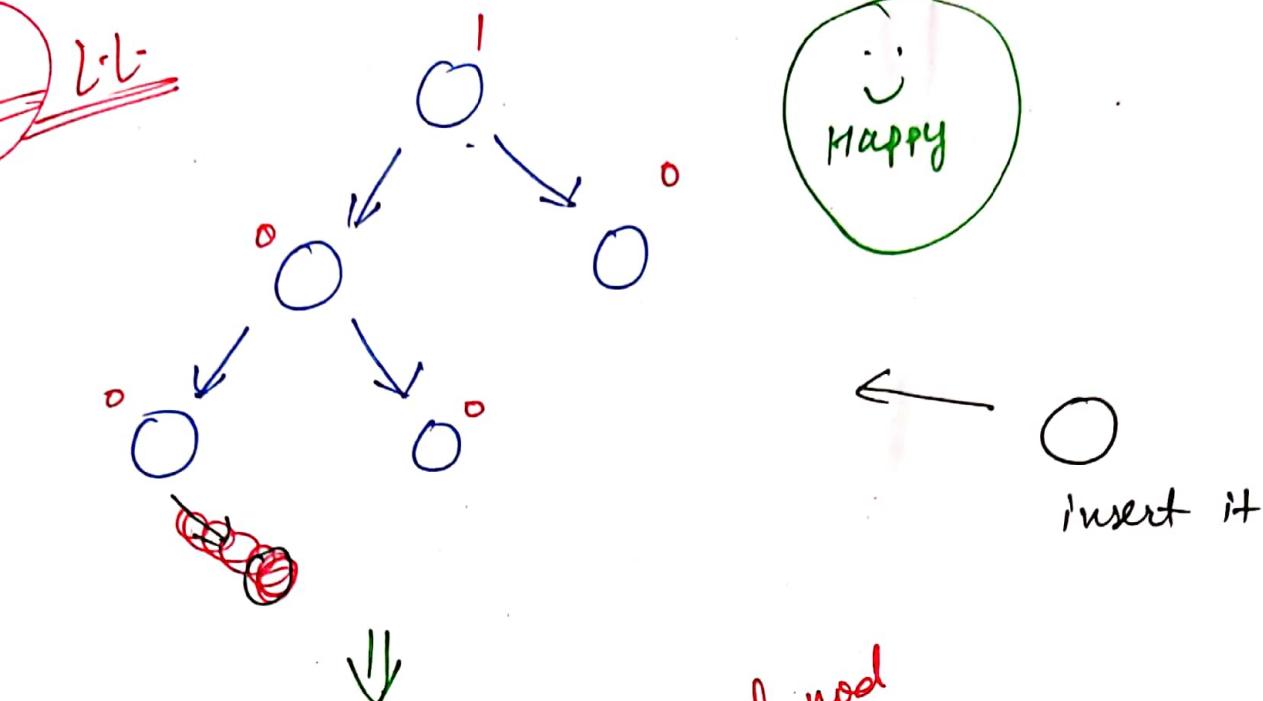
③ For a left-right insertion

left rotate once & then right rotate once

④ For a Right-Left insertion

right rotate once & then left rotate once

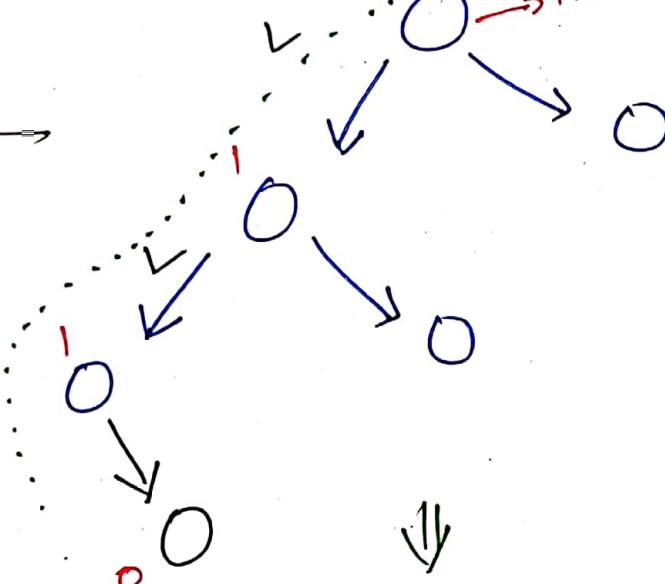
~~1~~ L.L



↓

imbalanced nod

Unbalanced
tree



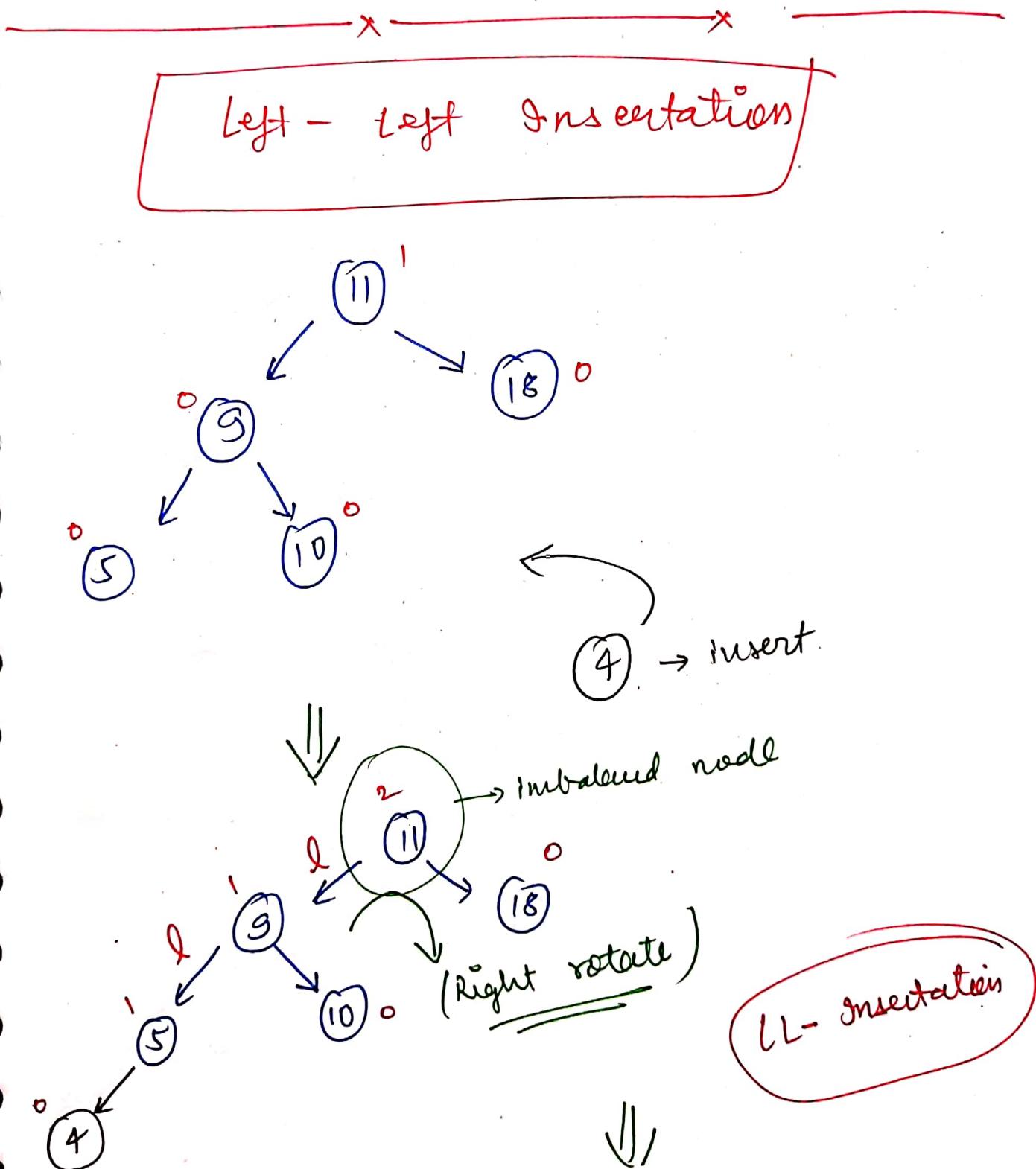
Search for 1st imbalanced node

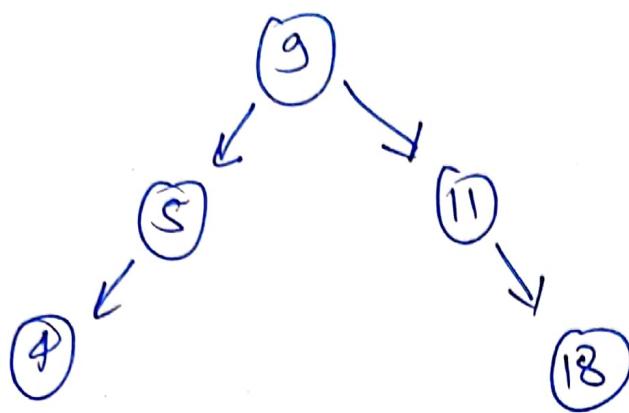
↓
jaha se insertion kiya wala
se upper move ~~out~~ & check

for imbalanced node

The diagram illustrates a linked list insertion operation. It shows a sequence of nodes connected by arrows labeled "left". The first node is a circle containing the number "0". An arrow labeled "left" points from this node to another circle containing "0". From this second node, an arrow labeled "left" points to a third node, which is a circle containing "1". A fourth node, also a circle containing "1", is shown further along the chain. To the right of the third node, the text "element is link" is written. A large circle, representing a temporary variable or a new node being inserted, is positioned below the third node. Inside this circle, the number "1" is written above a checkmark "✓". A diagonal line through the circle contains the word "insertation".

To balance it, 1 time right rotation
w.r.t imbalanced node

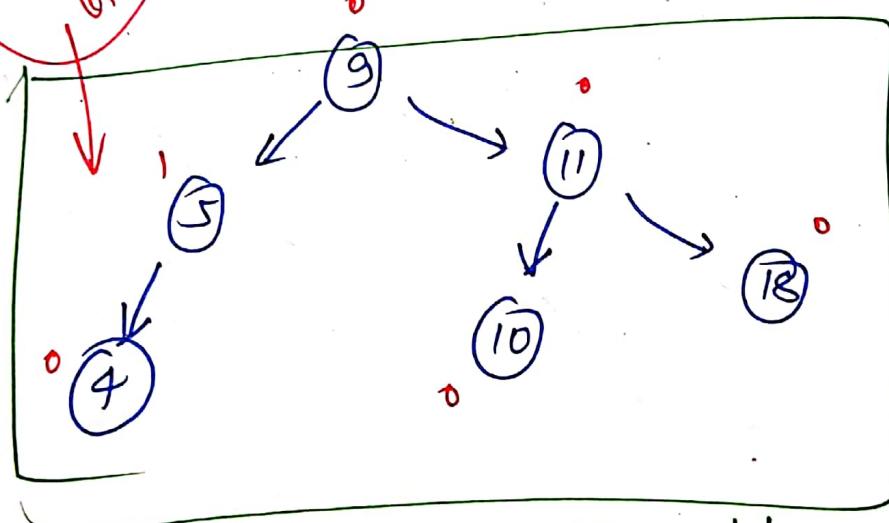




(10) gets out of
this tree

Right
Rotate
at 9

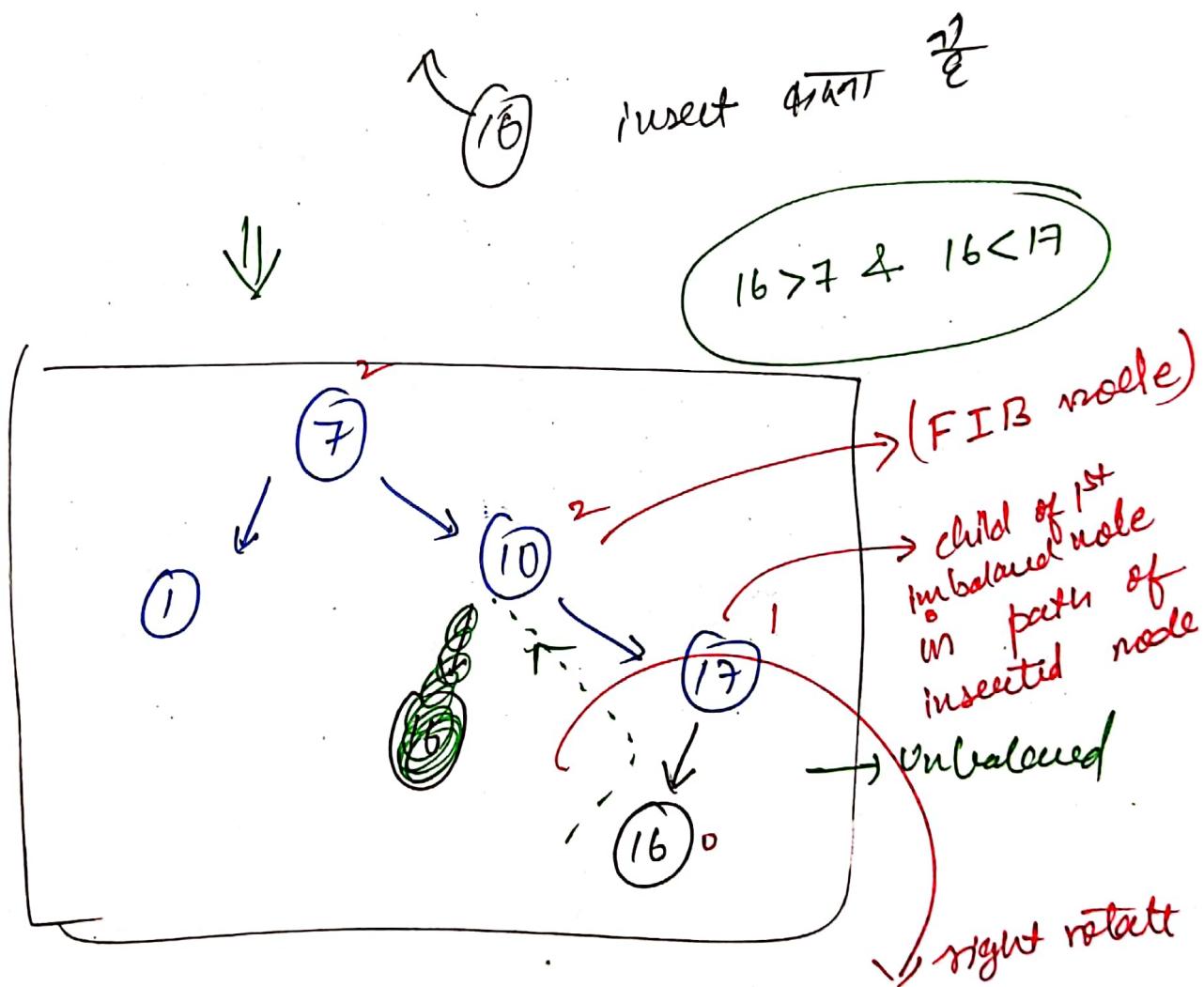
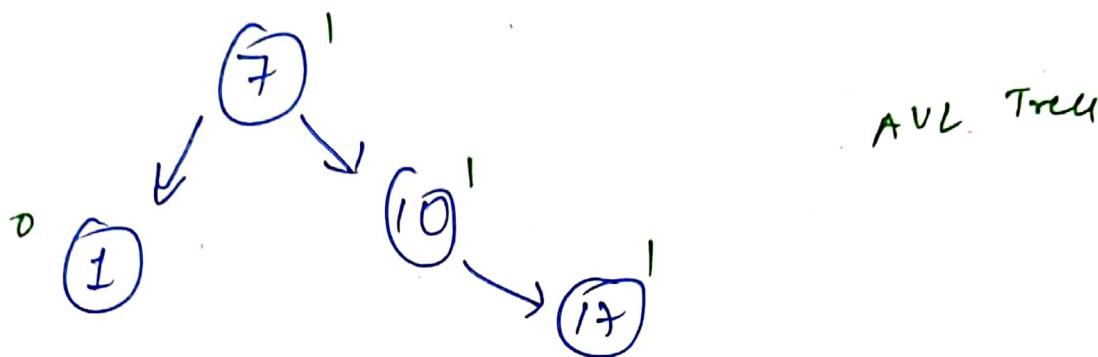
$10 > 9 \quad \& \quad 10 < 11$



AVL Tree !!

* 1st unbalanced node $\xrightarrow{\text{rotate}}$,
not necessary from root node!

② Right - Left Insertion

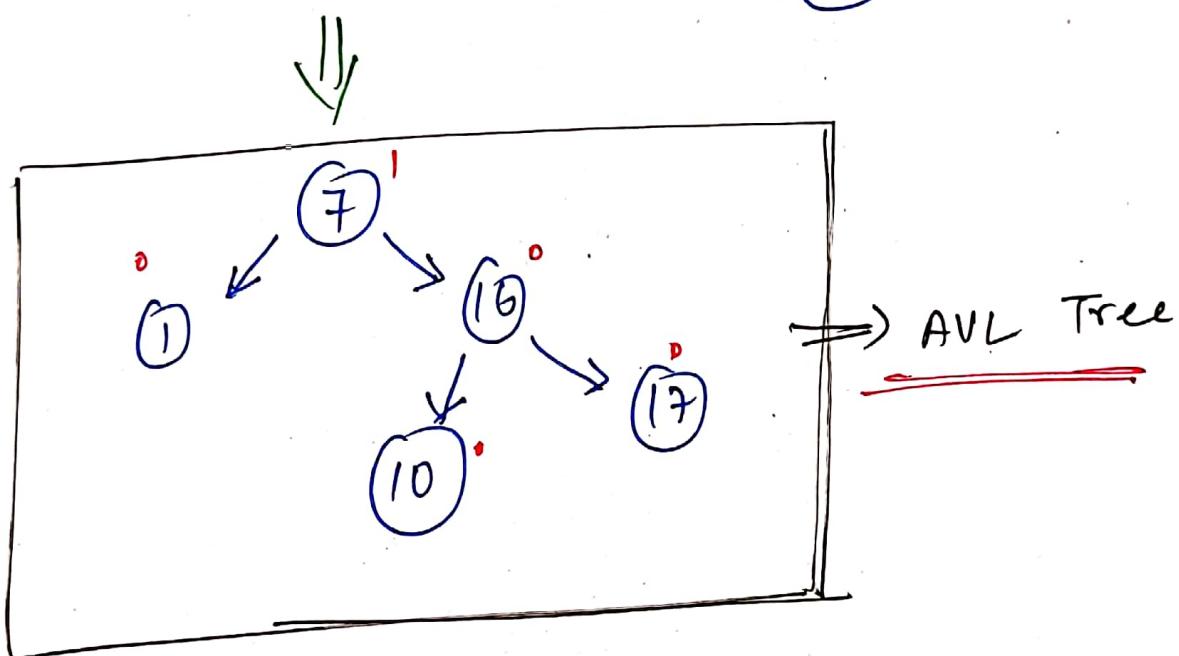
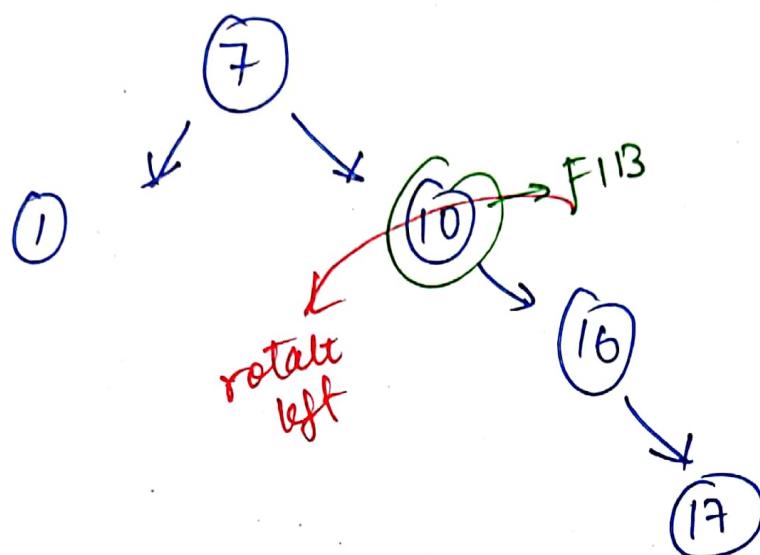


Right - Left insertion

↓
1st right rotate once \Rightarrow child of 1st imbalance node

2nd left rotate once \Rightarrow wrt FIB

↓ right rotation (child of FIB in path of inserted node)



Video - 82

C-Code for AVL Tree Insertion

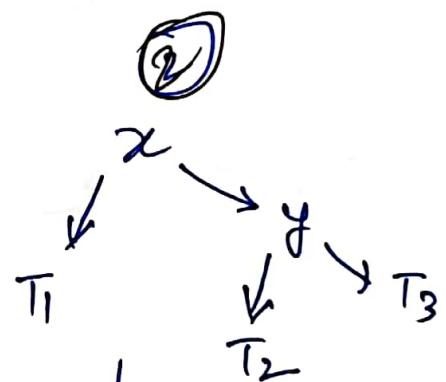
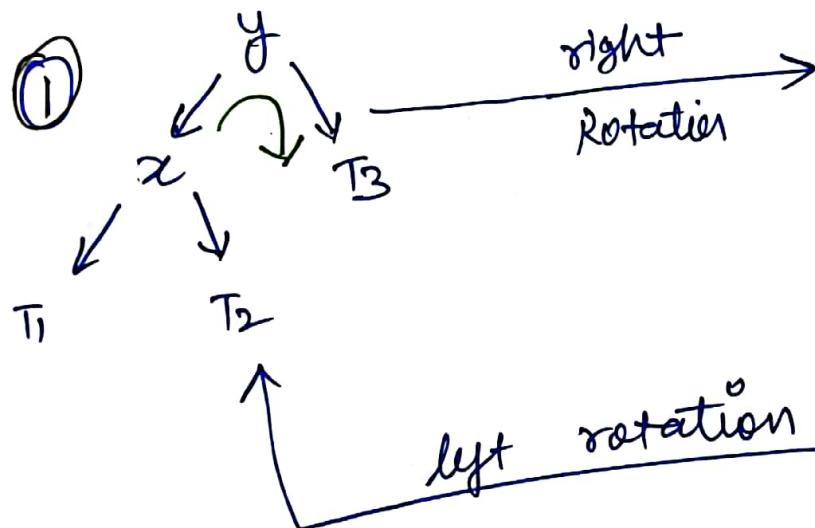
A

Rotation



(V.S. Code)

~~Right Left Rotation~~

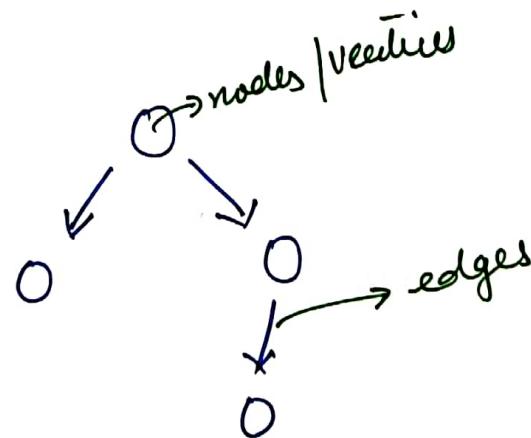


Video - 83

Introduction to Graphs

Graph

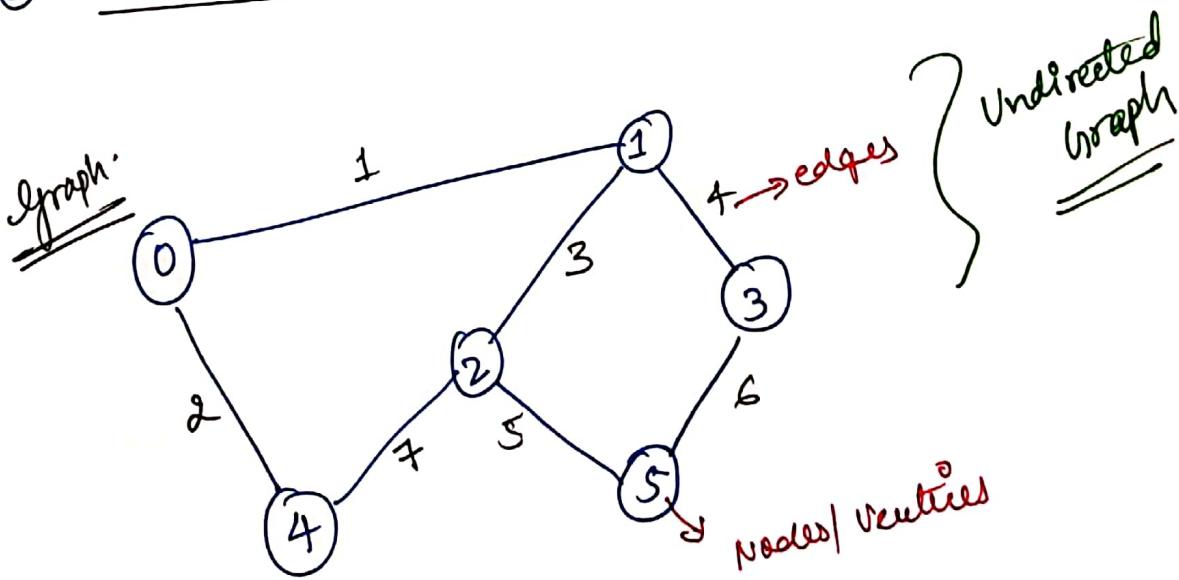
- ① Array, linked list & stack : \Rightarrow linear Data Structure
- ② BST & AVL Tree : \Rightarrow non-linear Hierarchical Data structure
- ③ Graph is an example of non-linear data structure
- ④ A Graph is a collection of nodes connected through edges.



- * Circle \Rightarrow nodes/ vertices
- * connecting dash b/w 2 nodes \Rightarrow edges

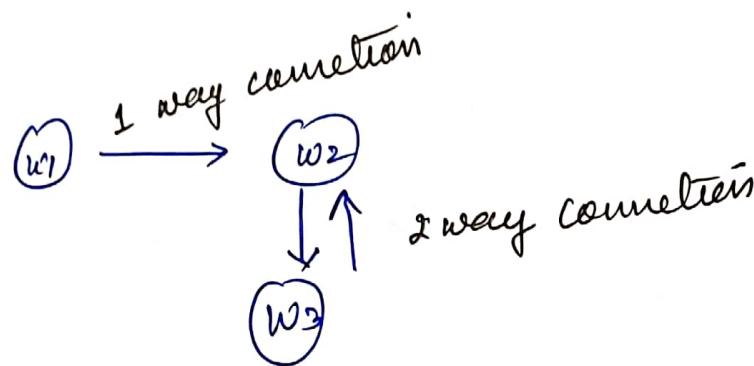
Formal Definition of Graph

- ① A graph $G = (V, E)$ is a collection of vertices & edges connecting these vertices.
- ② Used to model paths in a city, social network, website backlinks, internal employee network etc.
- ③ A vertex or node is one fundamental unit/entity of which graphs are formed.
- ④ An edge is uniquely defined by its 2 endpoints.
- ⑤ Directed Edge : \rightarrow one way connection
- ⑥ Undirected Edge : \rightarrow two way connection
- ⑦ Directed Graph : \rightarrow All directed edges
- ⑧ Undirected Graph : \rightarrow All undirected edges

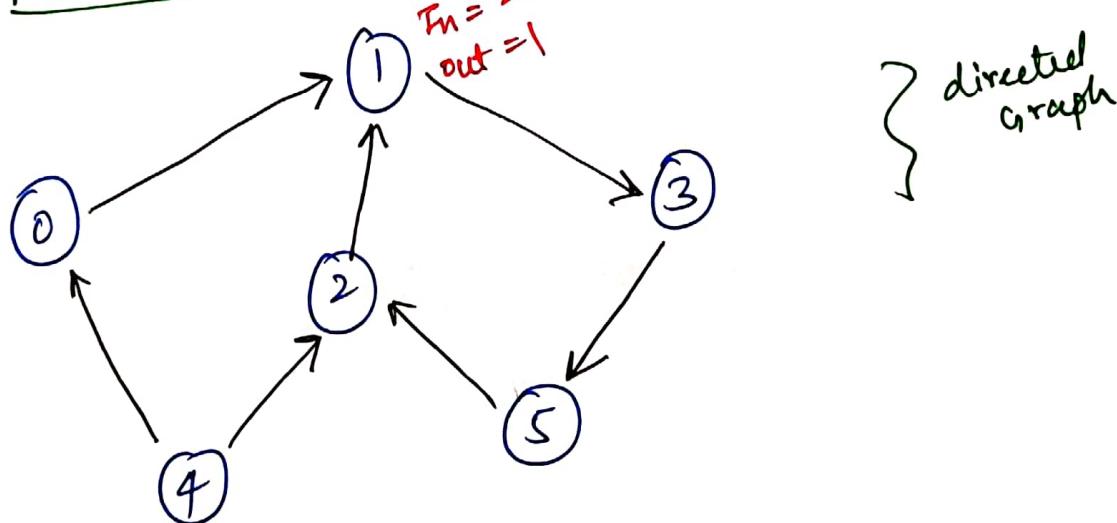


$V = \text{Vertices} = \{0, 1, 2, 3, 4, 5\}$

$E = \text{Edges} = \{\{0, 1\}, \{1, 3\}, \{1, 2\}, \{0, 4\}, \{2, 4\}, \{2, 5\}, \{3, 5\}\}$



Indegree & Outdegree of Node



Indegree : No. of edges going out of the Node

Outdegree : No. of edges coming into the Node

FaceBook — A graph of Users

- ① Graphs are used to model relationship b/w nodes.
- ② we can apply graph algorithm to suggest friends to people, calculate no. of mutual friends etc.
- ③ other eg: includes result of a web crawl for a website or for the entire world wide web, city routes etc.
↓
suggesting mutual friend.

Video - 84

Representation of Graphs

- ① Graph = Nodes & edges
- ② Used to model real world problems like managing a social network, website links etc.
- ③ solve prob. like is there a path b/w Delhi & California by road . If yes, which is the shortest route.



Ways to Represent a Graph

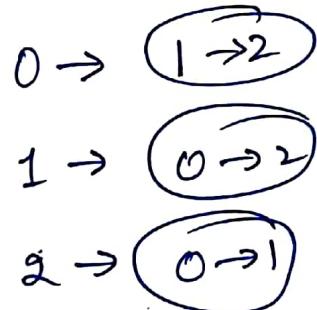
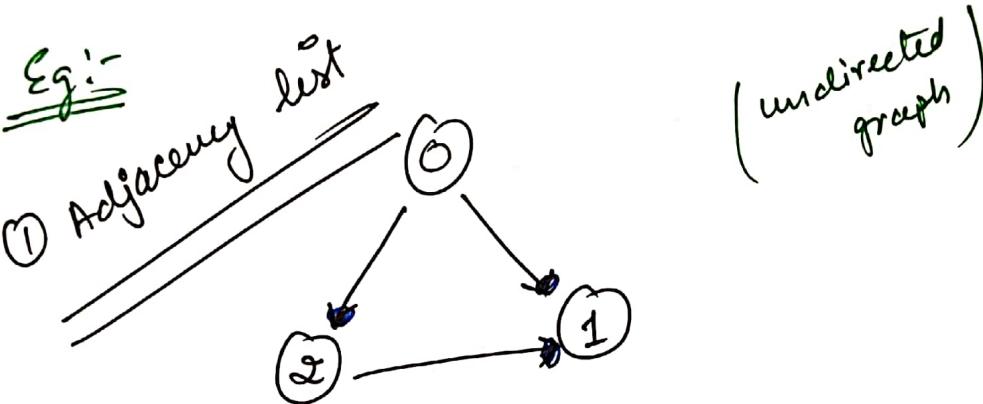
- ① Adjacency list
→ mark the nodes with the list of its neighbors
- ② Adjacency Matrix (A_{ij})
 $A_{ij} = 1$ for an edge b/w i & j ,
0 otherwise .

③ Edge Set

→ Store the pair of nodes / vertices connected with an edge.

④ Other Implementation :

Cost adjacency list, Cost adjacency matrix,
Compact list Representation



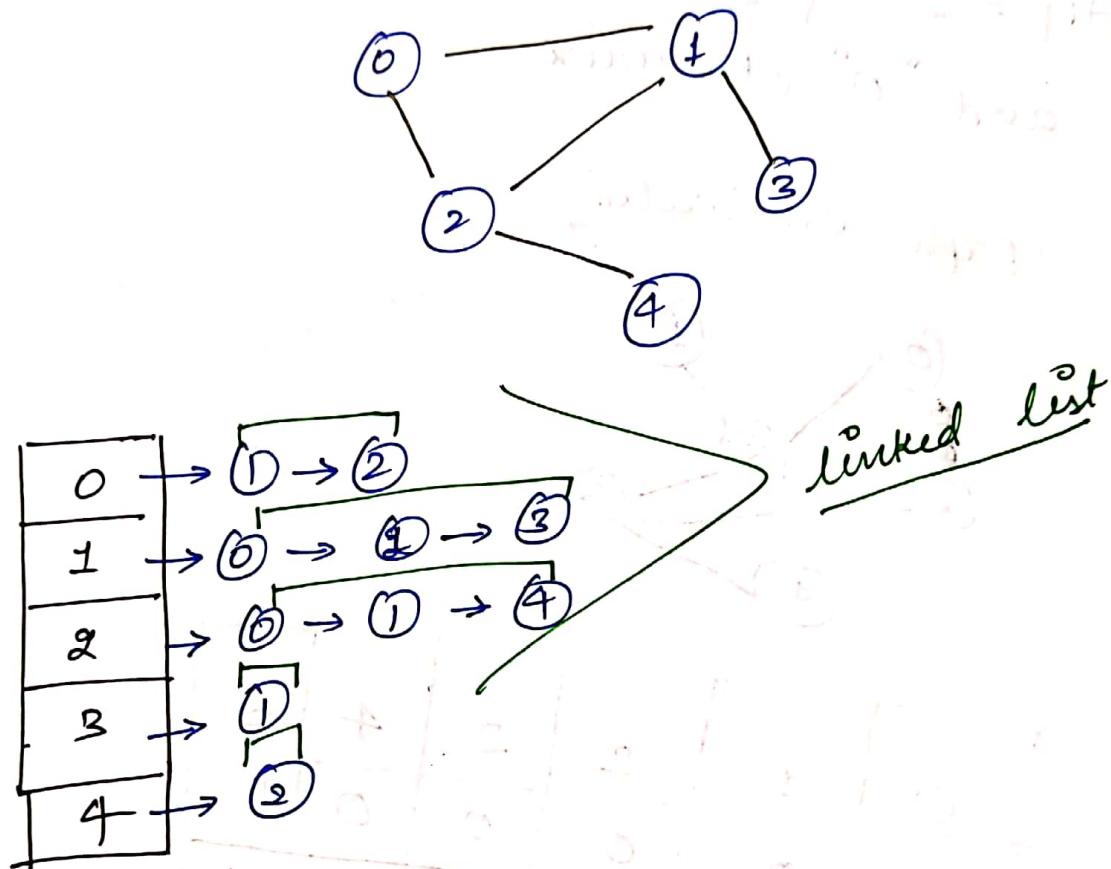
② Adjacency Matrix

	0	1	2
0	0	1	1
1	1	0	1
2	1	1	0

→ mark
 1
 (connected) → 0
 (not connected)

Adjacency List

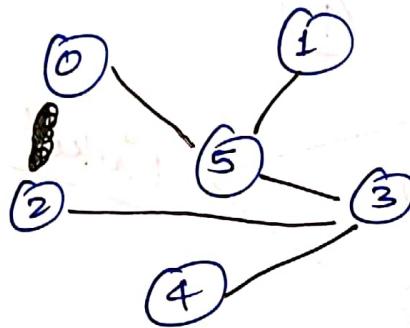
⇒ Map the nodes with the list of its neighbors.



Adjacency Matrix

$\Rightarrow A_{ij}^o = "1"$ for an edge b/w i and j
and "0" otherwise.

Graph Undirected



	0	1	2	3	4	5
0	0	1	0	0	0	1
1	0	0	0	0	0	0
2	0	0	0	1	0	1
3	0	0	1	0	1	0
4	0	0	0	0	1	0
5	1	1	0	1	0	0

Cost - Adjacency Matrix

- A_{ij}^o = cost for an edge b/w i & j ;
'0' otherwise !

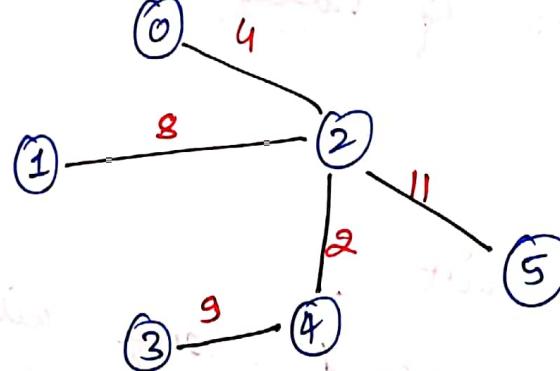
- if the cost can be 0 :

A_{ij} = cost for an edge b/w i & j :

'-1' otherwise

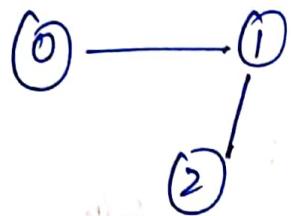
when
 $\text{cost} = 0$

graph



	0	1	2	3	4	5
0	-1	-1	4	-1	-1	-1
1	-1	-1	8	-1	-1	-1
2	4	8	-1	-1	2	11
3	-1	-1	-1	-1	9	-1
4	-1	-1	2	9	-1	-1
5	-1	-1	11	-1	-1	-1

Edge Set

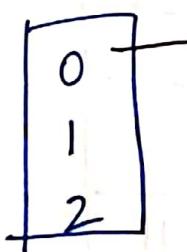


$\{(0,1), (1,2)\}$

- not used commonly
- store the pair of nodes connected with an edge.

Cost - Adjacency List

- ⇒ Cost is also stored along with the links.



Compact List Representation

- ⇒ Entire graph is stored in 1-d array.

Video - 85

* Graph - Traversal

- ① It refers to process of visiting (checking or updating) each vertex in a graph.
- ② Sequence of steps known as "Graph traversal algorithm" can be used to traverse a graph.
- ③ Two Algorithms of graph Traversal:
 - a) Breadth First Search (BFS) → (queue data structure)
 - b) Depth First Search (DFS) →
 (stack data structure)

Exploring a Vertex Node

- ① we have already looked into tree traversal algorithm in our section of on trees.
- ② In a typical Graph Traverse Algorithm, we traverse through (or visit) all the nodes of a graph and add it to the collection of visited nodes.
- ③ Exploring a vertex in a graph means visiting all the connected vertices.

Video - 86

Breadth First Search (B.F.S)

- Queue data structures follow
- In BFS, we start with node & start exploring its connected nodes. The same process is repeated with all the connecting nodes until all the nodes are visited.

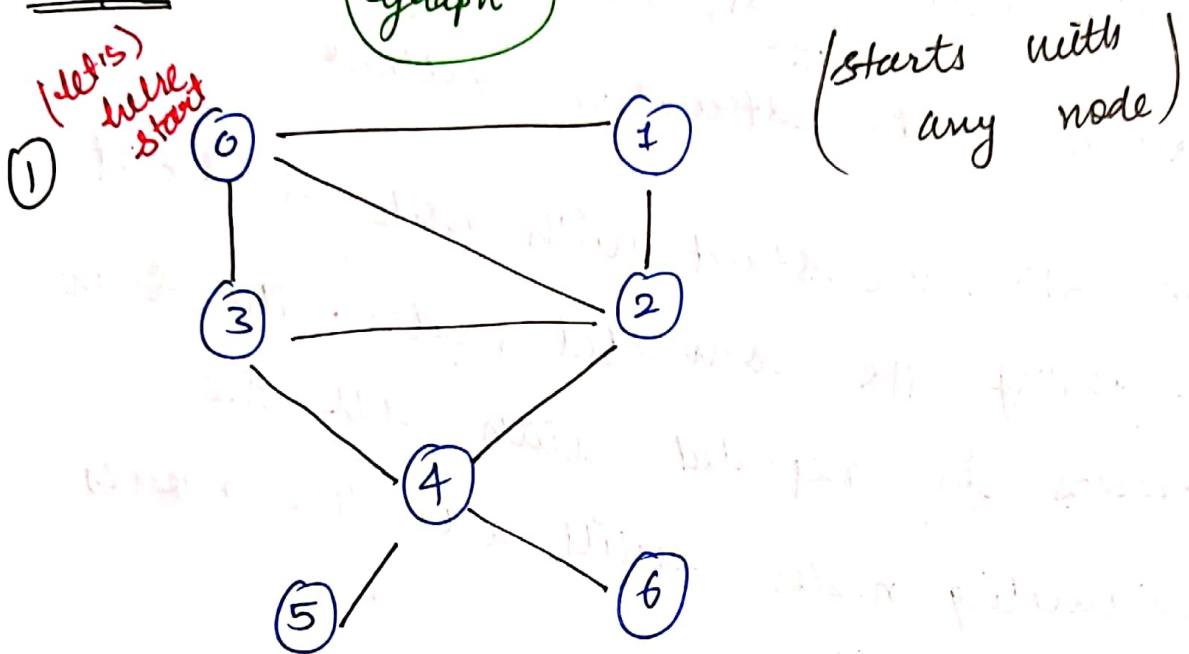
BSF spanning tree

- we can start with any source node
- let's start with O
- Try to construct a tree with O as the root
- mark all the sideways or duplicate edge as dashed
- This constructed tree \Rightarrow BSF spanning tree
- level order traversal of a BSF spanning tree is a valid BSF traversal

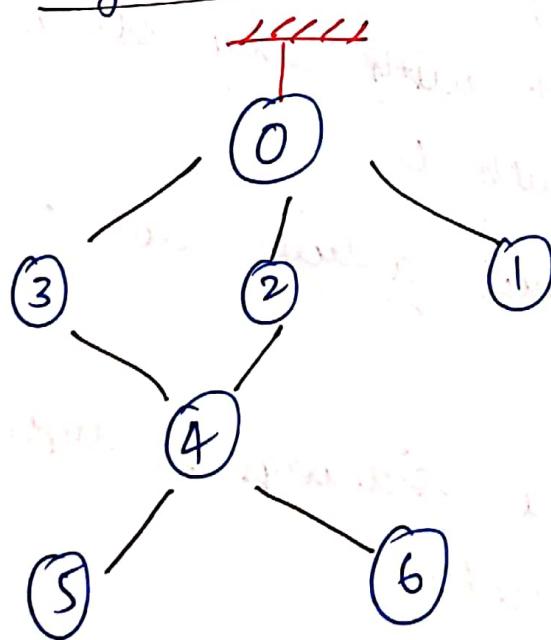
of a graph!

Method - I

Eq:-



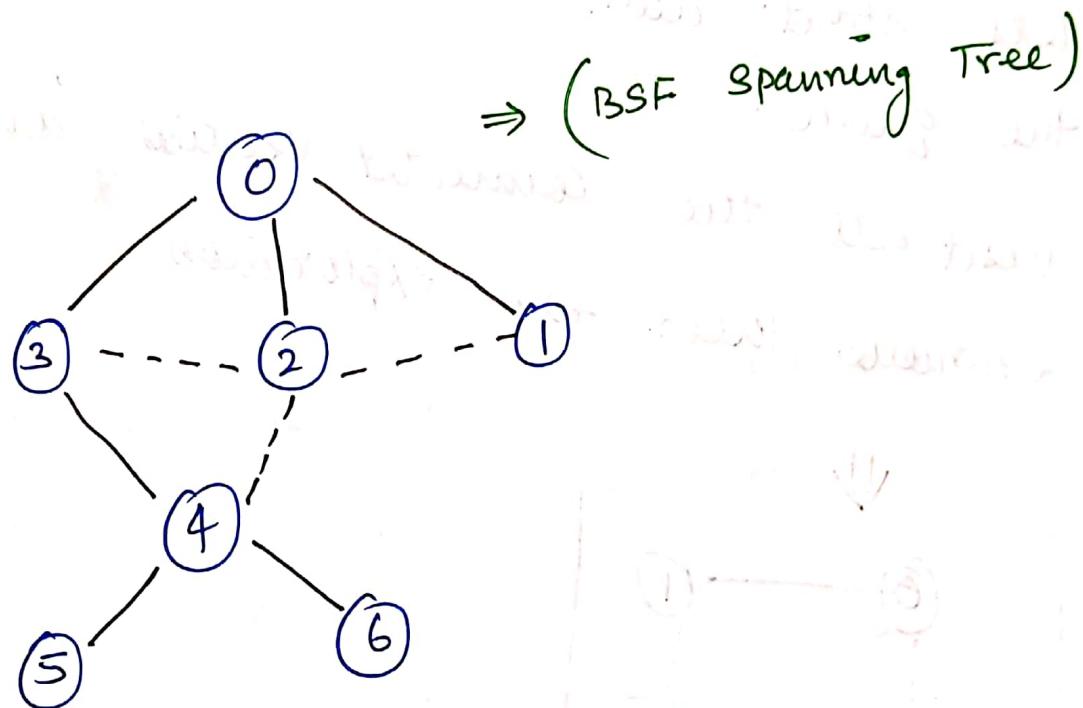
- (2) Hang ① node with the ceiling



③ Now mark all duplicate / sideways
as dashed (---)

→ we are doing dashed to make it
a valid tree

→ Tree hai, But not Binary Tree



* Level Order Traversal (LOT) ⇒ simply write the nodes
in the same level from left to right

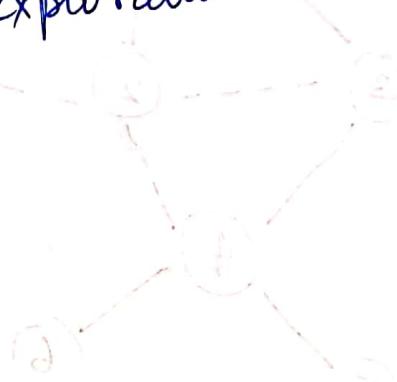
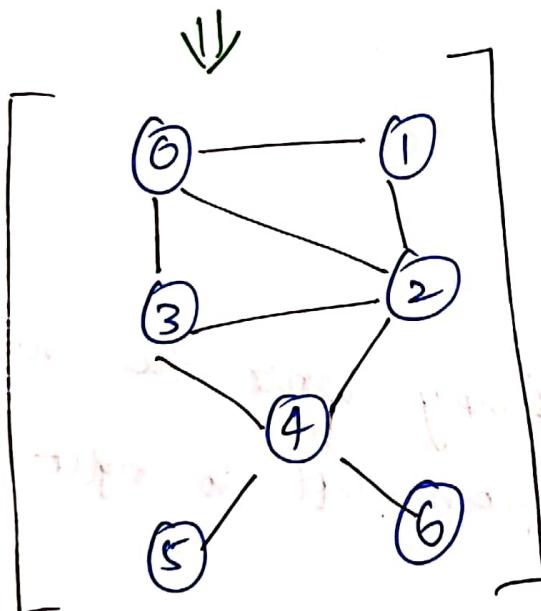
∴ BSF traversal of graph

⇒ LOT ⇒ [0 3 2 1 4 5 6]

Method - 2

BSF Traversal

- we can start with any source node.
- lets start with 0 and insert it in the queue.
- Visit all the connected vertices and enqueue them for exploration.



⇒ Exploration Queue :- (Ex)

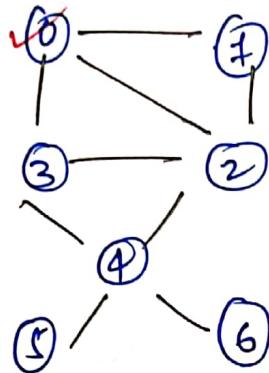
⇒ holds the nodes we'll be exploring one by one.

⇒ Visited Queue :- (V) ⇒ array holding the

status of whether a node is already visited or not.

Step: 1

(Let's start with 0_{zero})



$$V = 0$$

$$Ex = 0$$

Step-2

Start visiting all nodes connected to node 0 from Ex Queue.

$$V = 0 \pm 2 3$$

$$Ex = \cancel{0} \pm 2 3$$

Step-3

we have node - 1 at the top in Ex-Queue.
So, we'll take it out & visit all unvisited nodes connected to it.

$$V = 0 \ 1 \ 2 \ 3$$

$$Ex = \cancel{0} \ \cancel{1} \ 2 \ 3$$

Step - 4

Now, we have node - 2 & only unvisited node connected to it = 4. So, we'll mark it visited and will also enqueue it in our exploration queue.

$$V: 0 \ 1 \ 2 \ 3 \ 4$$

$$Ex: \cancel{X} \ \cancel{X} \ \cancel{X} \ 3 \ 4$$

Step - 5

Do similar for node - 3.

$$V \Rightarrow 0 \ 1 \ 2 \ 3 \ 4$$

$$Ex \Rightarrow \cancel{X} \ \cancel{X} \ \cancel{X} \ \cancel{X} \ 4$$

Step - 6

for node = 4

$$V \Rightarrow 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6$$

$$Ex \Rightarrow \cancel{X} \ 5 \ 6$$

Step - 7

for node = 5 & 6

$$V \Rightarrow 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6$$

$$Ex \Rightarrow \cancel{X} \ \cancel{X} \ \cancel{X}$$

Breadth 1st traversal.

Important Points

- ① we can start with any vertex.
- ② There can be multiple BFS results for a given graph.
- ③ The order of visiting the vertices can be anything.

Video-87

BFS Implementation in C-Code

V.S. Code

Coding

int u = 0;
int i = 0;
int visited [7] = {7 nodes array};

empty array
let a [7][7]
create
Adjacency Matrix

	0	1	2	3	4	5	6
0	0	1	1	1	0	0	0
1	1	0	1	0	0	0	0
2	1	1	0	1	1	0	0
3	1	0	1	0	1	0	0
4	0	0	1	1	0	1	1
5	0	0	0	0	1	0	0
6	0	0	0	0	1	0	0

Video - 88

Depth First Search (DFS)

Graph Traversal

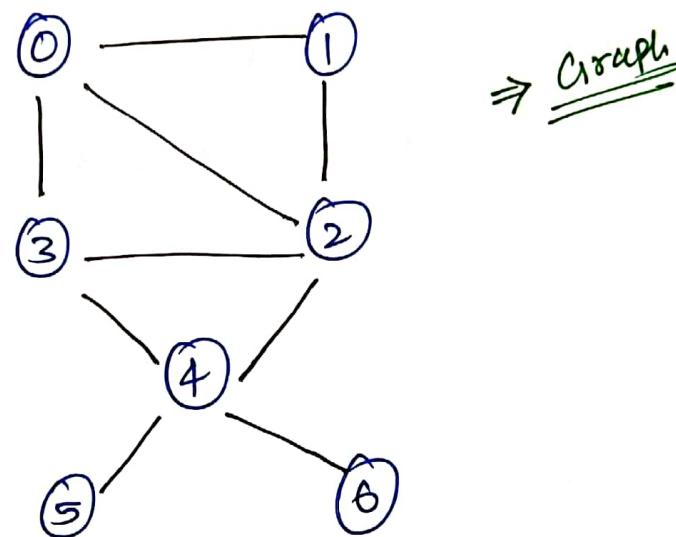
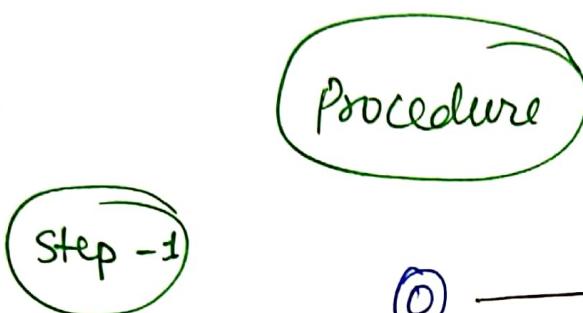
⇒ In DFS, we start with a node & start exploring its connected nodes keeping on suspending the exploration of forev. nodes.

Procedure

- ① Start by putting any one of the graph's vertices on top of a stack +
- ② Take the top item of the stack + add it to the visited list .
- ③ Create a list of that vertex's adjacent nodes . Add the ones which aren't in the visited list to the top of the stack .

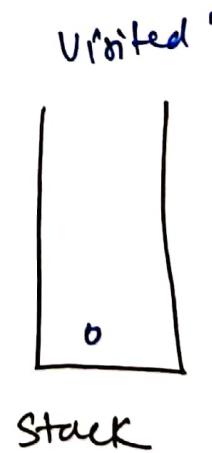
④ keep repeating steps 2 & 3 until the stack is empty.

⇒ Use stack data structure,



Let's start with (0).

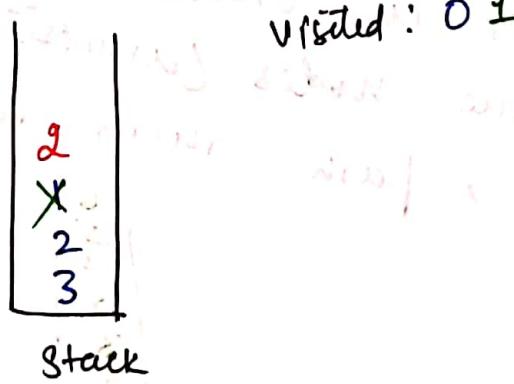
Push node = 0 in stack.



Step-2

Pop out node = 0 from stack & mark it visited. Start visiting all the nodes connected to node = 0 which are not visited before. And push them into the stack i.e., node connecting to 0 (zero) and the order in which you pushes doesn't matter at all.

∴ Push \rightarrow 3, 2, 1 node in our stack



Step-3

Now, pop top element from stack which = 1 node, & mark it visited.

Only node 1 connected = 0 + 2.
Since only unvisited = 2 (0 already in visited list)

⇒ so push 2 into the stack again.

Step-4

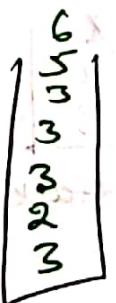
→ we have node = 2 at the top. → 2-pop out
mark 2 as visited & unvisited node
connected to it, put them in the
stack.



Visited = 0 1 2

Step-5

→ node = 4 at the top. pop it out &
mark it in visited list.
all the nodes (unvisited node) connected
to 4, push them in stack



Visited = 0 1 2 4

↓
Repeat this procedure until
your stack is empty.

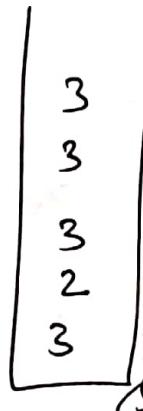
Step-6

Visited : 0 1 2 4 6



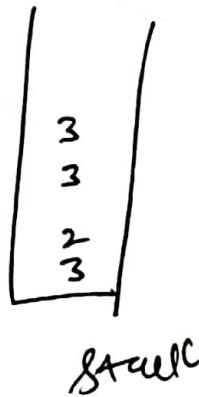
Step-7

Visited = 0 1 2 4 6 5



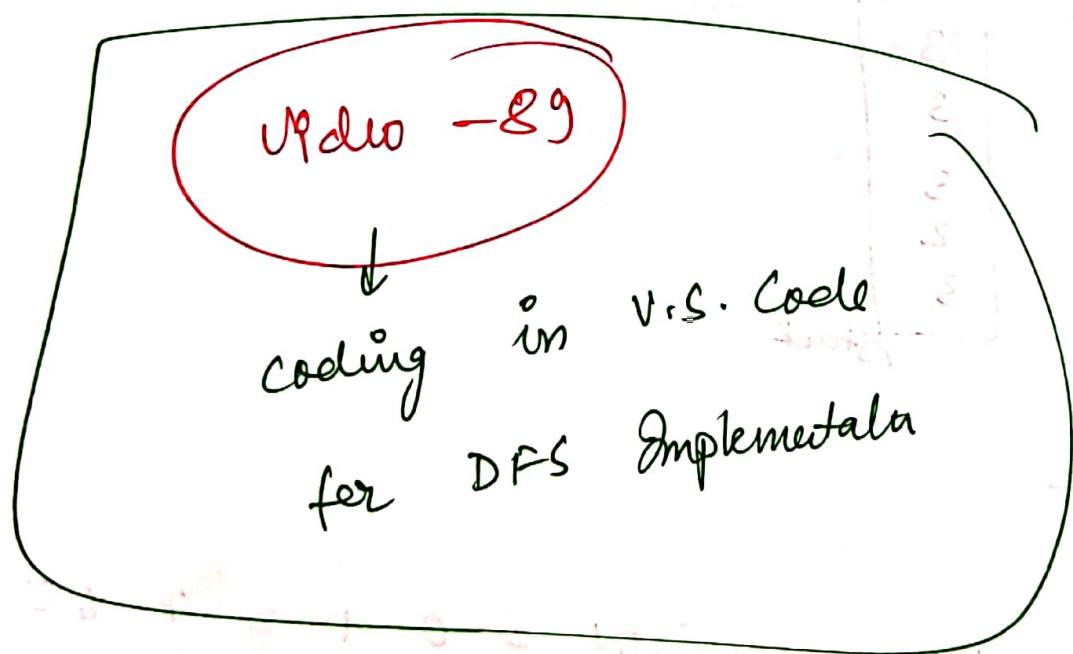
Step-8

Visited = 0 1 2 4 6 5 3



⇒ Now, no nodes left to be visited.
Although there are elements in the stack to be explored. So just pop them one by one & ignore finding them already visited. And this gets our stack empty.

Note f^n uses stack by default.

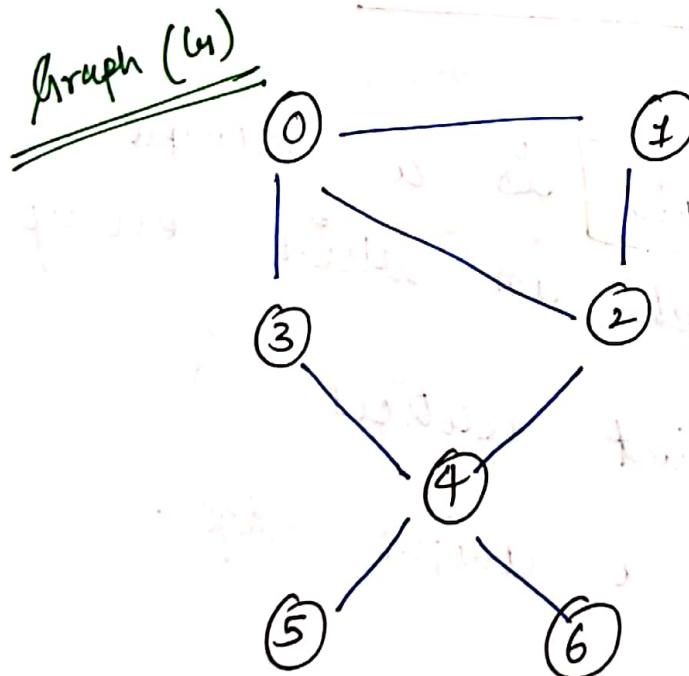


Video - 90

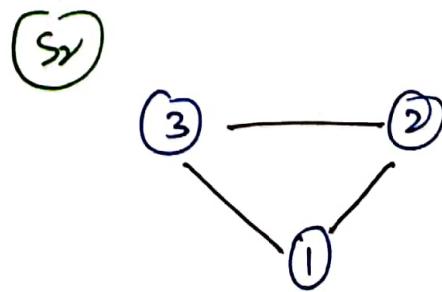
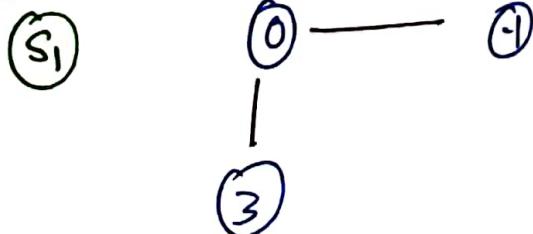
Spanning Trees & Maximum no. of possible spanning trees

Subgraph

- A Subgraph of a graph ' G_1 ' is a graph whose vertices & edges are subsets of the original graph ' G_1 '.

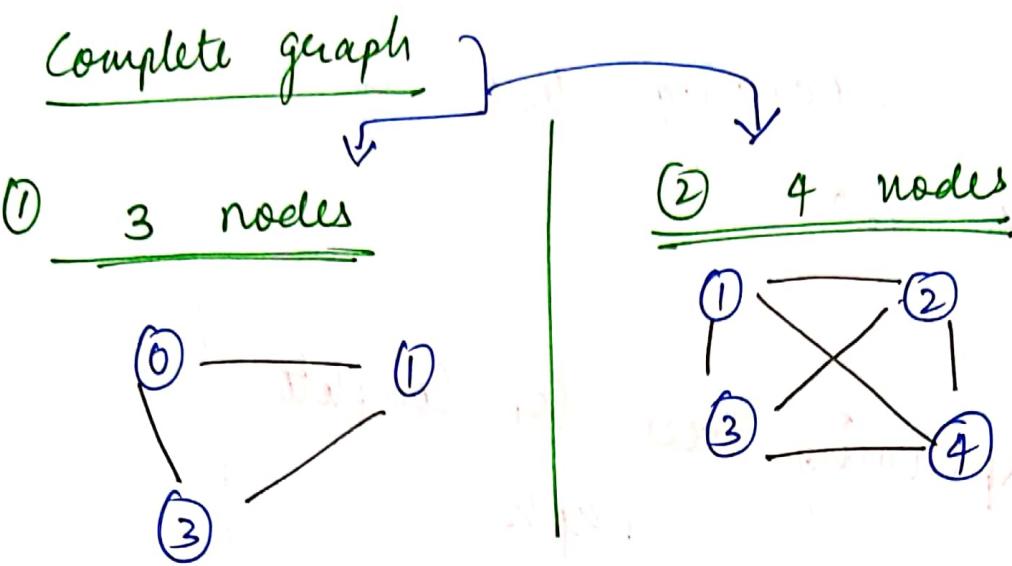


Subgraph



Connected & Complete Graphs

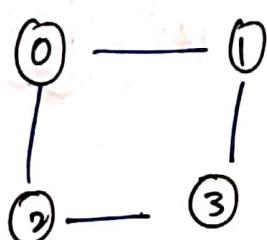
- * A **connected graph** is a graph that is connected in the sense of a topological space i.e., there is a path from any point to any other point in the graph.
- * A graph that is not connected is said to be **disconnected graph**.
- * A **complete graph** is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.



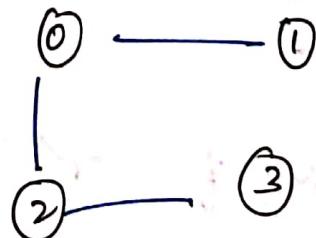
Spanning Trees

- * A connected subgraph 'S' of graph $G(V, E)$ is said to be a spanning tree of graph G iff (if & only if) :-
 - (i) All vertices of G must be present in S
 - (ii) No. of edges in S should be $V-1$.
(vertices - 1)

G (graph)



S (spanning tree)

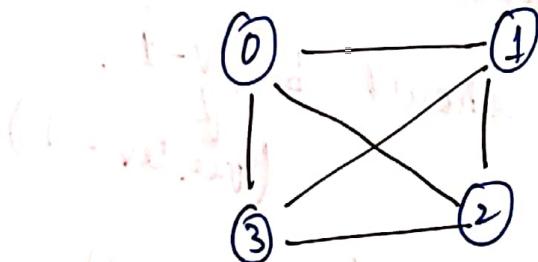


$\therefore S$ is a spanning tree of graph ' G_1 '.

No. of spanning Trees for Complete Graph

* A complete graph has $\Rightarrow (n^{n-2})$ spanning trees where,
 $n = \text{no. of vertices in the graph}$

Graph given

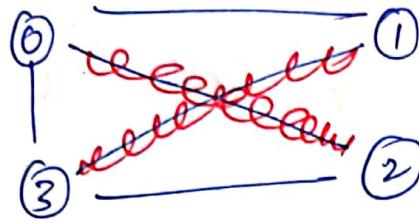


vertices = 4

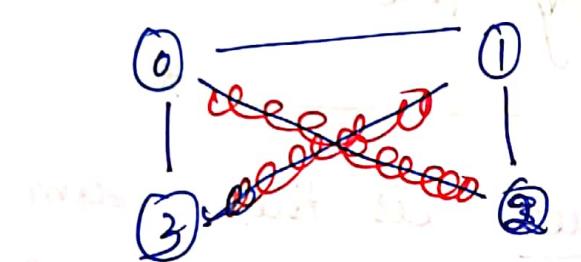
$$\text{No. of spanning tree} = (4)^{4-2} = 16$$

Spanning Tree for above Graph

(1)

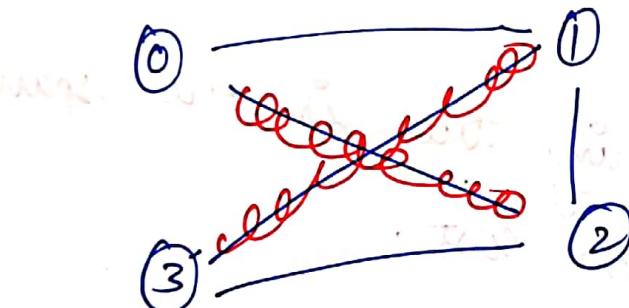


(2)



3 spanning tree.

(3)

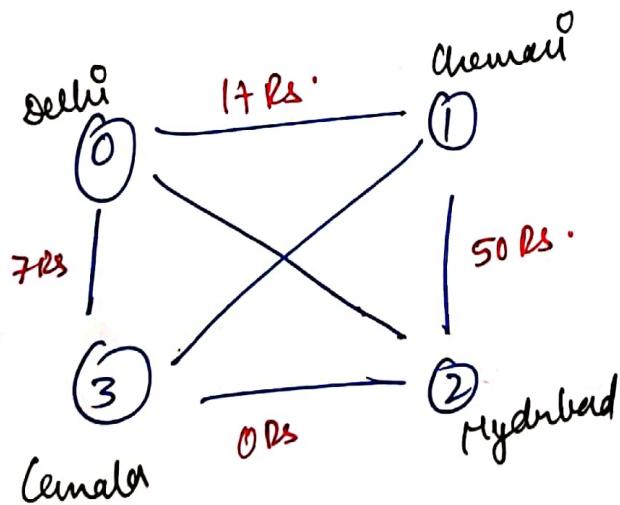


Video - 91

* Calculating Spanning Tree Cost & Minimum Spanning Tree

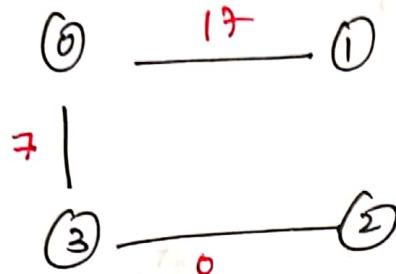
* Cost of Spanning tree is the sum of the weights of all the edges in the tree.

* A minimum spanning tree is the spanning tree with minimum cost.



Spanning Tree

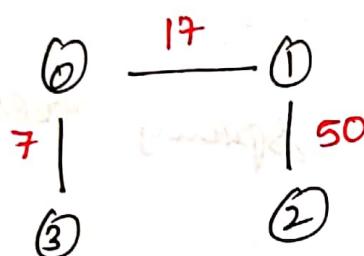
①



(A)

$$\text{Cost}_1 = 17 + 7 + 0 = 24$$

②



(B)

$$\text{Cost}_2 = 7 + 17 + 50$$

$$= 74$$

$$\text{Cost}_1 < \text{Cost}_2$$

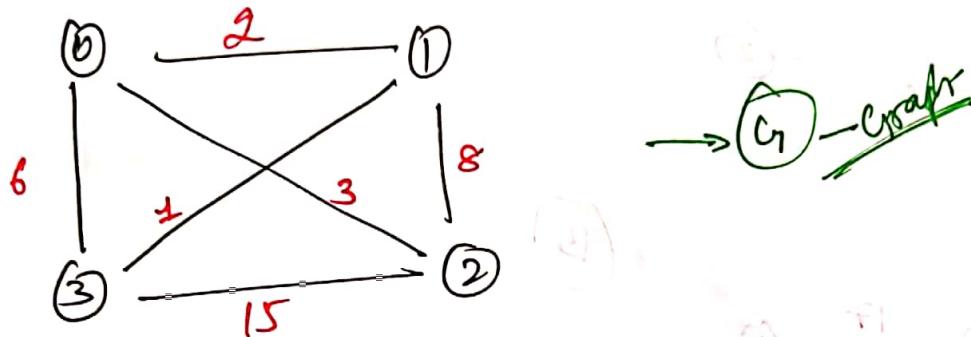
\therefore 1st spanning tree will be preferred

from above

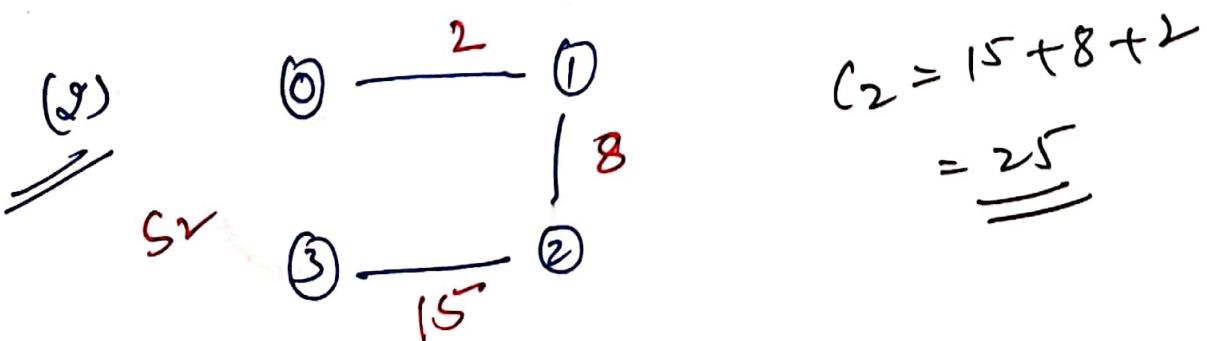
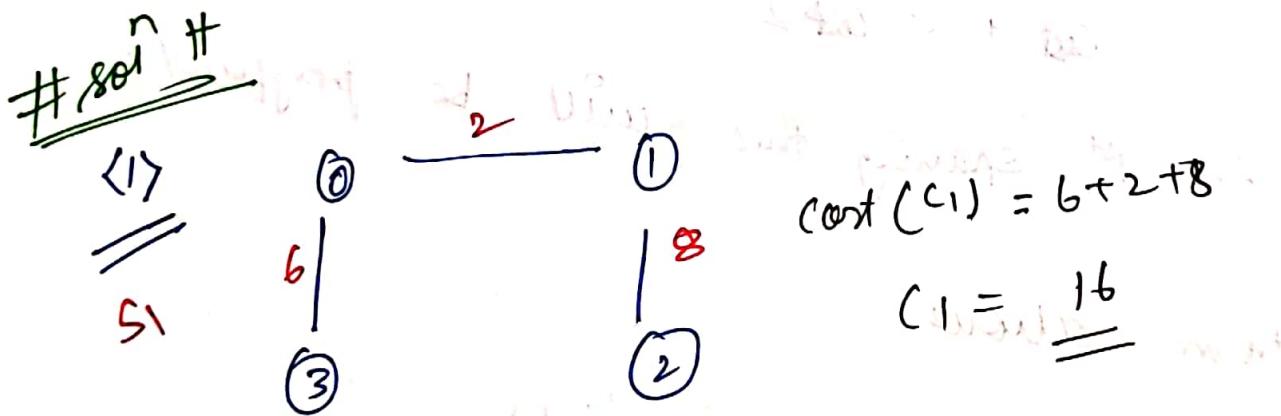
$B = \min \cdot \text{Spanning trees}$
(MST)

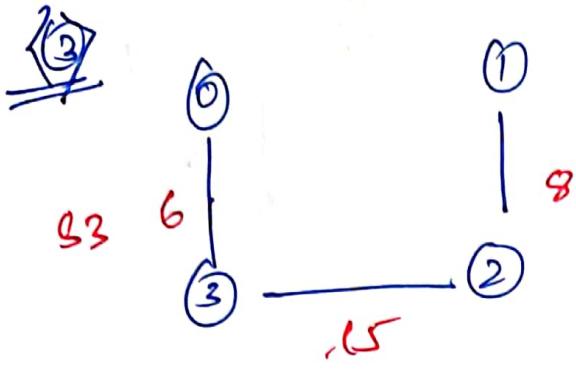
Exercise :

Cost of Spanning Tree

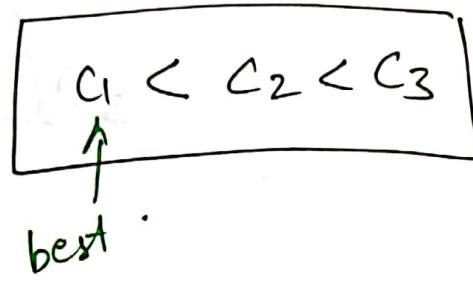


find the cost of any spanning tree of the graph 'G1'?



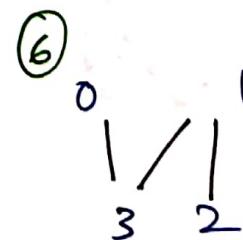
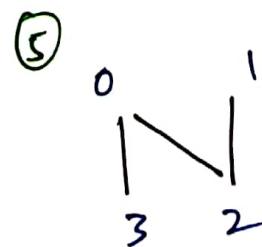
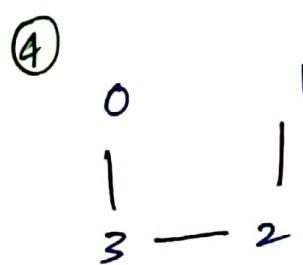
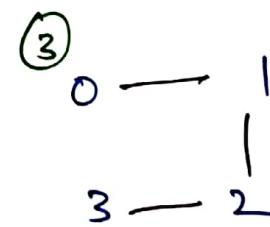
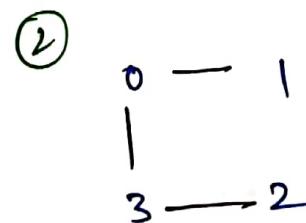
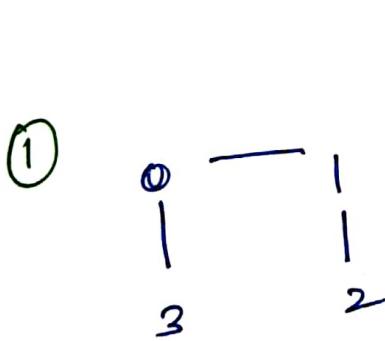


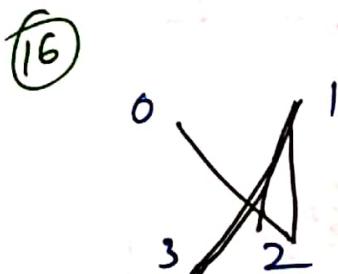
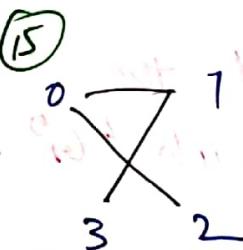
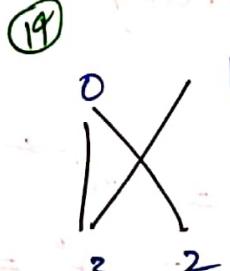
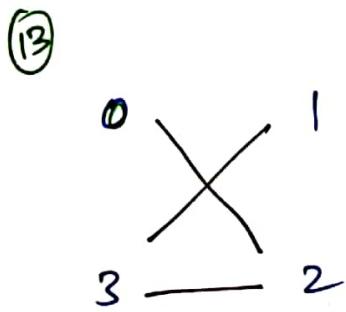
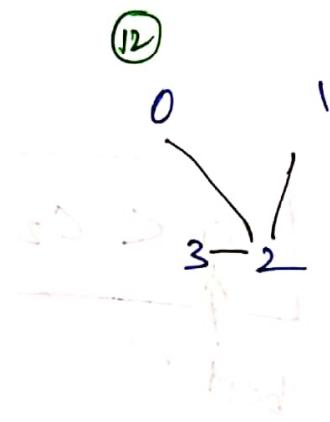
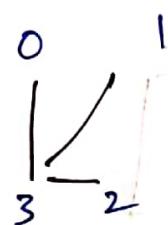
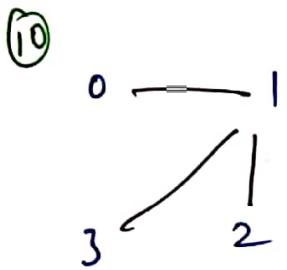
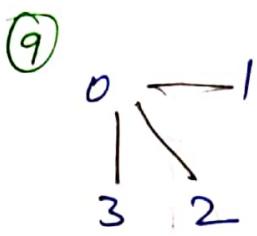
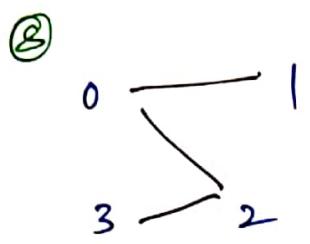
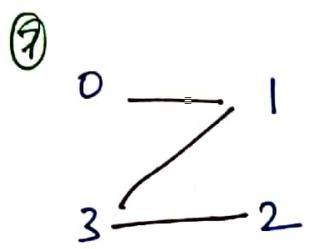
$$C_3 = 6 + 8 + 15 \\ = \underline{\underline{29}}$$



Ques. find the min. spanning tree of the graph 'G1'.

no of spanning tree = ~~n^{n-2}~~
 $= 4^2 = \underline{\underline{16}}$





Video - 92

Prims Algorithm

Prims Algorithm

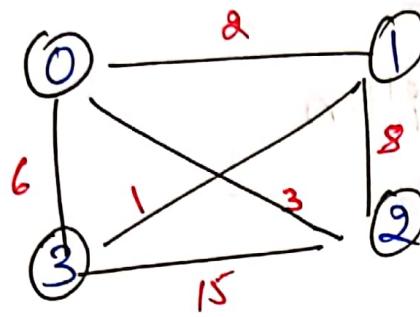
→ Uses Greedy approach to find the minimum spanning tree (MST)

→ we start with any node & start creating a MST

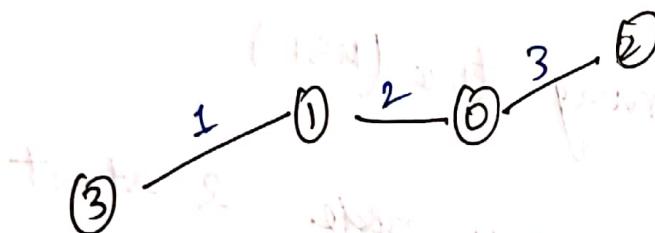
→ In Prim's Algo ; we grow spanning tree from a starting point until

$(n-1)$ edges are formed.
(n = nodes number)

Ex: 1

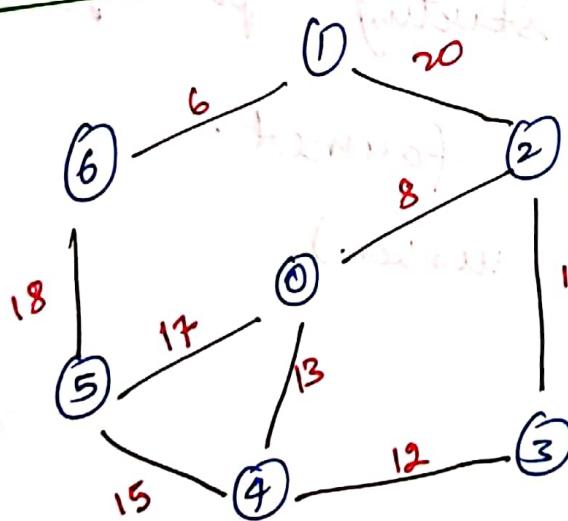


Ex: 1



$$\text{cost of MST} = 1 + 2 + 3 \\ = \underline{\underline{6}}$$

Ex: 2



Graph (6)

Colⁿ

Step 1: Choose any arbitrary node
(let's say = ①)

Not included in MST

$$V = \{0, 2, 3, 4, 5, 6\}$$

$$V = \{0, 2, 3, 4, 5\}$$

$$V = \{0, 2, 3, 4\}$$

$$V = \{0, 2, 3\}$$

$$V = \{0, 2\}$$

$$V = \{0\}$$

do this at each level.

$$0 \rightarrow 5 = 17 X$$

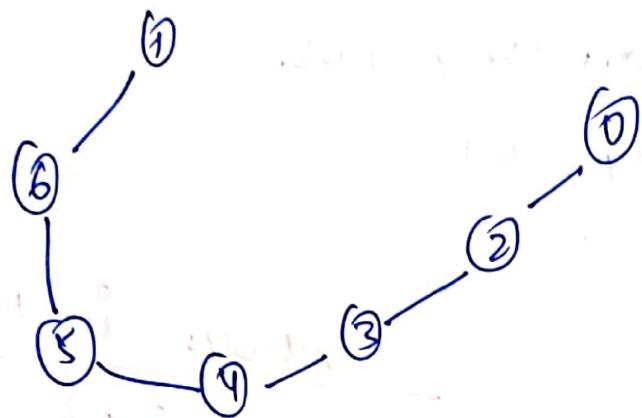
$$0 \rightarrow 4 = 13 \checkmark$$

$$0 \rightarrow 2 = 8 \checkmark$$

$$\begin{aligned} A &= \{1, 4, 5, 6\} \\ A &= \{1, 3, 4, 5, 6\} \\ A &= \{1, 2, 3, 4, 5, 6\} \end{aligned}$$

$$V = \{\}$$

$$A = \{0, 1, 2, 3, 4, 5, 6\}$$



\therefore Points \rightarrow $6 + 18 + 15 + 12 + 11 + 8 = 70$

$$= 6 + 18 + 15 + 12 + 11 + 8 = 70$$

$$\text{Sum} = \underline{\underline{70}}$$

Note

options:

$$\begin{aligned} \textcircled{1} & \quad 1 \rightarrow 2 = 20 \times \\ & \quad 1 \rightarrow 6 = 6 \checkmark \end{aligned}$$

$$\begin{aligned} \textcircled{2} & \quad 1 \rightarrow 2 = 20 \times \\ & \quad 6 \rightarrow 5 = 18 \checkmark \end{aligned}$$

$$\begin{aligned} \textcircled{3} & \quad 1 \rightarrow 2 = 20 \times \\ & \quad 5 \rightarrow 0 = 17 \times \\ & \quad 5 \rightarrow 4 = 15 \checkmark \end{aligned}$$

(4)

$$\begin{aligned}1 \rightarrow 2 &= 20 \times \\4 \rightarrow 3 &= 12 \checkmark \\4 \rightarrow 0 &= 13 \times \\5 \rightarrow 0 &= 17 \times\end{aligned}$$

(5)

$$\begin{aligned}1 \rightarrow 2 &= 20 \times \\3 \rightarrow 2 &= 11 \checkmark \\4 \rightarrow 0 &= 13 \times \\5 \rightarrow 0 &= 17 \times\end{aligned}$$

(6)

$$\begin{aligned}0 \rightarrow 5 &= 17 \times \\0 \rightarrow 4 &= 13 \times \\0 \rightarrow 2 &= 8 \checkmark\end{aligned}$$