

Content :

- ① Intro to DSA
- ② Recursion + Problems
- ③ Big 'O' + Problems
- ④ Arrays
- ⑤ Lists
- ⑥ Dictionary
- ⑦ Tuple
- ⑧ Linked List
 - Singly L-L
 - Circular · Singly L-L
 - Doubly L-L
 - Circular · Doubly L-L
- ⑨ Stack
- ⑩ Queue
- ⑪ Tree
- ⑫ Binary Search Tree - BST
- ⑬ AVL tree
- ⑭ Binary Heap
- ⑮ Trie
- ⑯ Hashing
- ⑰ Sorting Algorithms
- ⑱ Graph Algorithms
- ⑲ Kruskal's Algorithm
- ⑳ Prim's Algorithm
- ㉑ Greedy Algorithm
- ㉒ Divide and Conquer Algorithm

Definitions

(10-07-2021)

Data structure

- D.S. are different ways of organizing data on our computers, that can be used effectively.

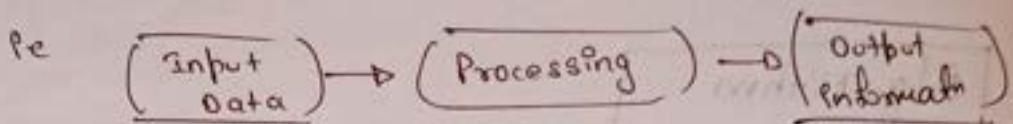
Algorithm

- set of Rules for a computer program to accomplish a task.
- few famous algo. used by
 - ✓ Google & Facebook transmits live video across internet through ?
[Compression algo.]
 - ✓ How to find shortest path on the map ?
[Graph algo.] → Dijkstra
 - ✓ How to arrange solar panels on the International space station ?
[Optimization & scheduling algo.]
- What makes a good algorithm ?
 - ✓ correctness
 - ✓ efficiency.

- Why Data structures & Algo. are imp. ?

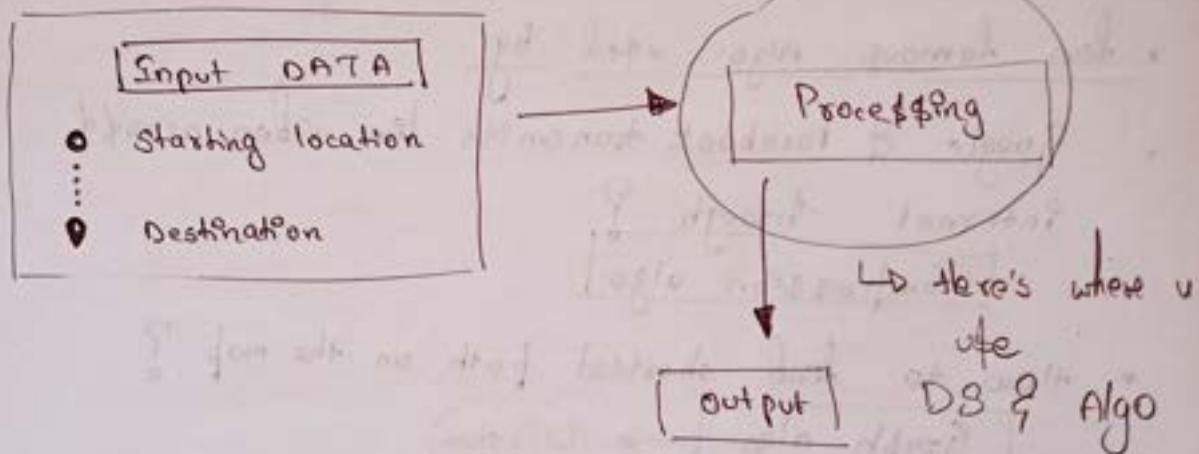
★ See of a Computer scientist

u hv to work with data & e DATA PROCESSING



(Ex)

① (GOOGLE MAPS.)



↳ Here's where u use
DS & Algo

Algo

DATA

Types of

- Based on either
- either

Primitve

PRIMITIVE

- INTEGER
- FLOAT
- CHARACTER
- STRING
- BOOLEAN

LINEAR

the data items in the memory
in a linear, sequential manner.

STATIC

structures associated memory
locations are fixed at compile time

ARRAY

DATA STRUCTURES

predefined or defined by user
in 2 types

User-defined

NON-PRIMITIVE

[Combination of 2 or more data structures]
Primitive DS

[also include tuple, dict, set]

NON-LINEAR

the data items are conn. to several
other items, they are not organised
sequentially.

TREE

GRAPH

DYNAMIC

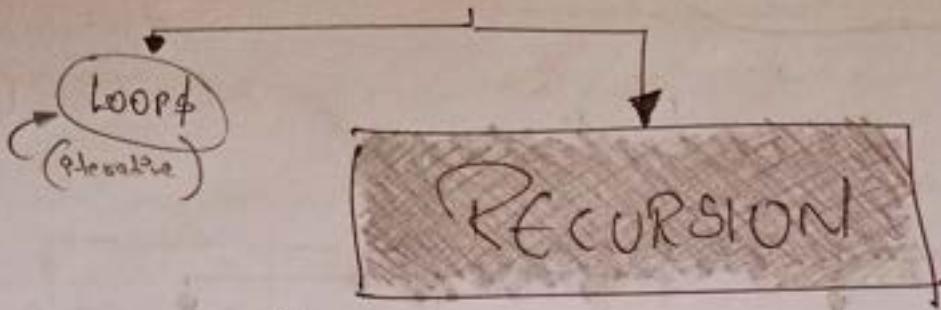
associated memory locations
change

- LINKED LIST
- STACK
- QUEUE

[Repetitive]

~~Staircase~~

problems can be solved in ② ways



- A way of solving a problem by having a function calling itself.

OR

- A data structure relying of the similar structure of its own instances.

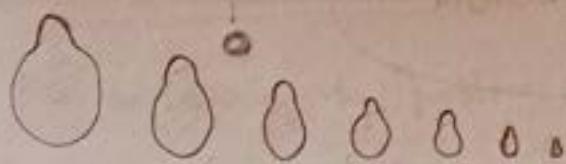
• Properties of Recursion

- ① Performing the same operation multiple times with different inputs.
- ② In every step we try smaller inputs to make the problem smaller.
- ③ Base condition is needed to stop the recursion, otherwise infinite loop will occur.
if the base condition is not specified.

• EX

- A Russian Matryoshka Dolls.

11-07-2021



- So code for that exple example would be like :

```

- def openRussianDoll(doll):
    if doll == 1: ③
        print("All dolls are opened")
    else:
        openRussianDoll(doll - 1) ②
    
```

Types of Recursion

Direct recursion:

A function say "fun" is called D.R. if it calls the same function "fun".

Indirect recursion:

A function say "fun" is called I.D.R. if it calls another function say "fun-new" and "fun-new" calls "fun" directly or indirectly.

Tailed Recursion: if recursive call

If the last thing done by the functn. There's no need to keep record of previous state.

```

def fun(n):
    if (n == 0):
        return
    else:
        print(n)
        return fun(n-1)
    
```

Non-tailed Recursion: if recursive call is not the last thing done by the functn.

After returning back, there's some "something" left to evaluate.

```

def fun(n):
    if (n == 0):
        return
    fun(n-1)
    print(n)
    
```

Why Recursion?

- ① Recursive thinking is really imp. in programming,
it helps you break down big problems into smaller
ones.
↳ easier to use.
- ② Recursive soln can be simpler to type than iterative
 - Recursion is used all the time in nearly every field
in nearly every language.
 - Recursive codes are quite easy to write compared to
iterative ones.
 - Iterative codes are the ones where we use loops.
- ③ But there are situations where iteration performs
faster than recursion.
So it depends on situation.

③ When to choose Recursion ?

Rule

- If u can divide the problems into **similar** problems you can use recursion.
- how would u know that the sub-problems are WIT?
 - Design an algo. to compute nth... } of problems
 - Write code to list the n... } begin with these
 - Implement a method to compute all. } No Recursion.

④ PRACTICE

- The prominent usage of recursion in data structures like trees & graphs.
- Asked in interviews.
- Recursion is used in many algorithms like (divide & conquer, Greedy, dynamic programming)

How Recursion Works?

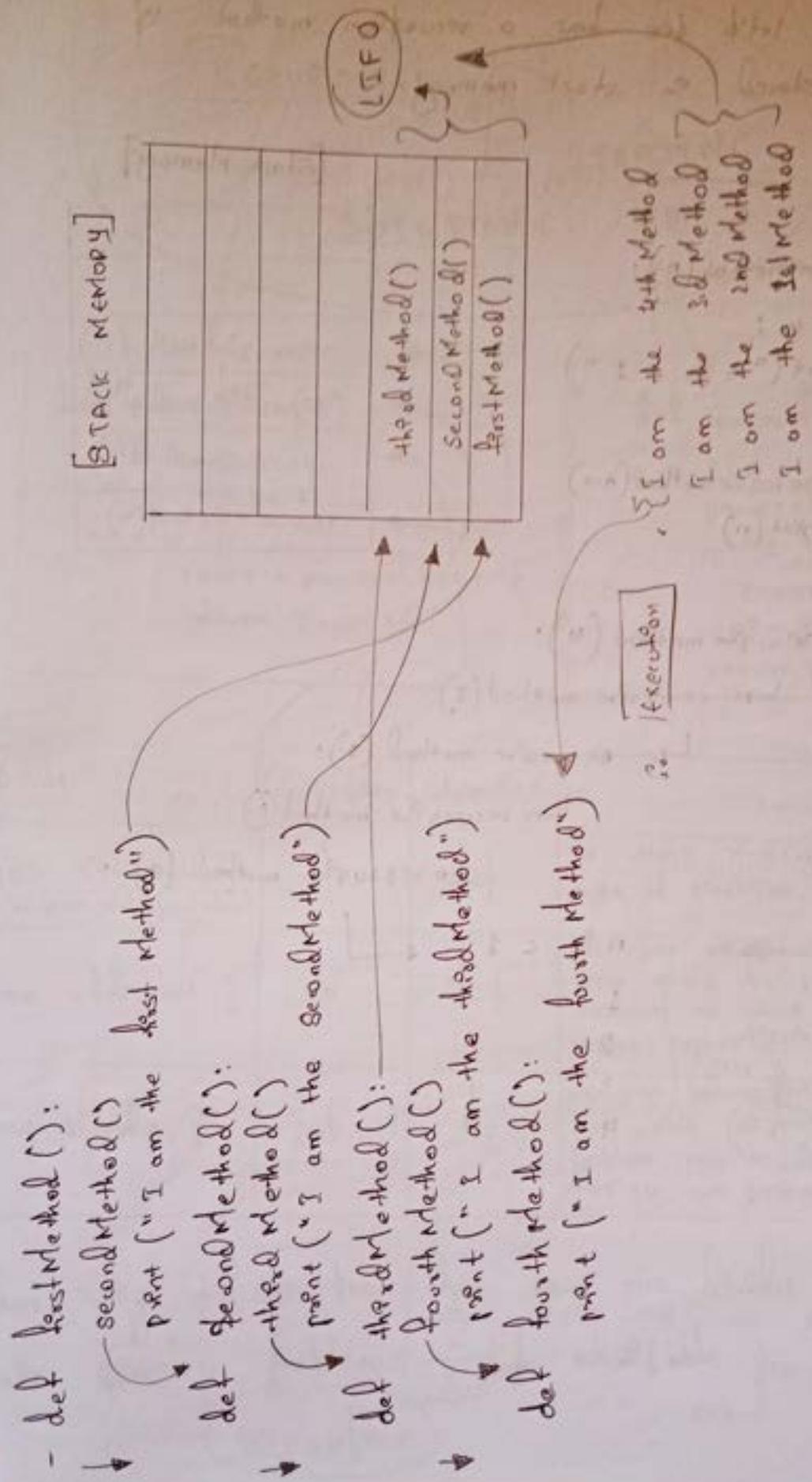
- ① A method / function calls itself
- ② Exit from infinite loop.

③ Syntax of a Recursion

```
- def recursionMethod(parameters):  
    if exit from condition satisfied:  
        return some value  
    else:  
        recursionMethod(modified parameters)
```

④ By learning how Recursion works

Let's understand how a method works with stack memory.



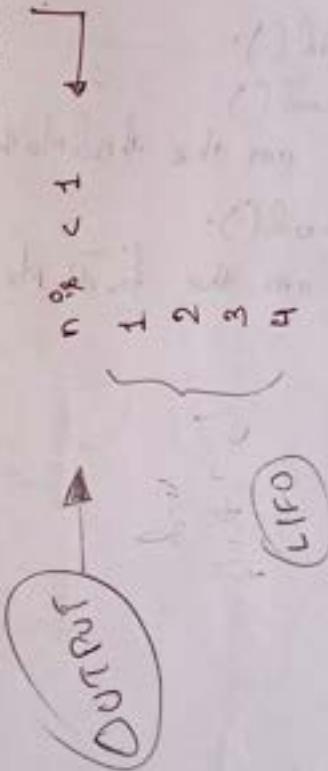
- Now let's see how a recursive method of short on stack memory.

[Stack memory]

```

    - D.P
      ↗
      ↗ recursive method(n):
      ↗ n<1:
        print("n is < 1")
      else:
        ↗ recursive method(n-1)
        print(n)
    
```

→ ~~def~~ recursive method(n):
 ↗ recursive method(5):
 ↗ recursive method(2):
 ↗ recursive method(1):
 ↗ recursive method(0):



Recursive vs Iterating

Solutions

def power of two(n):

if n==0:

return 1

else:

power = power of two(n-1)

return power * 2

def power of two(n):

q=0

power=1

while q<n:

power = power * 2

q = q + 1

return power

Points

Recursion Iteration

Point	Recursion	Iteration
Space efficient?	No <small>(memory not space efficient)</small>	Yes
Time efficient?	No	Yes
Easy to code?	Yes	No

In case of recursion system needs more memory for each & push elements to stack memory which makes recursion less space efficient.

We use recursion especially in the cases we know that a problem can be divided into simple sub problems.

That's why if u know that you can divide a problem into smaller problems, it's preferable to use recursion

BAN
EGO

Use / AVOIDRecursionwhen to use it ?

- when we can easily breakdown a problem into similar subproblem.
- when we are fine with extra overhead [ie both time & space complexity] that comes with it.
- when we need a quick working soln instead of efficient one.
- when we traverse a tree
- when we use Memoization in recursion.

When to avoid it ?

- if time & space complexity matters for us.
- Recursion uses more memory.
if we use embedded memory for ex an applic* that takes more memory in the phone is not efficient.
- Recursion can be slow.

How to write Recursion in

3 steps

- ① Recursive case - the flow
- ② Base case - the stopping condition
- ③ Unintentional case - the constraint.

(fx) factorial of a no.

Step 1

$$\hookrightarrow n! = n(n-1)(n-2)\dots 2 \times 1$$
$$n! = n(n-1)!$$

↳ the flow.

Step 2

$$\begin{aligned} \hookrightarrow 0! &= 1 \\ 1! &= 1 \end{aligned} \left. \begin{array}{l} \text{Base} \\ \text{condition} \end{array} \right\}$$

Step 3

$$\begin{aligned} \hookrightarrow \text{factorial}(-1) \\ \text{factorial}(1.5) \end{aligned} \left. \begin{array}{l} \\ \times \end{array} \right\}$$

using those 3 cases/steps

def factorial(n):

assert n >= 0 and n <= 10, "The no. must be between 0 to 10"

if n == 0 or 1:

return 1

else:

return n * factorial(n-1)

↳ Step 1

(a)

Using fibonacci no.

print the nth term in the fibonacci series,
taking nth term.

Step 1 $\rightarrow f(n) = f(n-1) + f(n-2)$

Step 2 $\rightarrow 0 \& 1$

Step 3 $\rightarrow \{ \begin{matrix} \text{fibonacci}(-1) \\ \text{fibonacci}(1) \end{matrix} \}$

Initial conditions

def fibonacci(n):

assert $n >= 0$ and int(n) == n, 'fibonacci no cannot be non-integer'.

~~if $n <= 0$ or:~~ / if n in [0, 1]:

return n

else:

return fibonacci(n-1) + fibonacci(n-2)

Code

(b)

$\text{fibonacci}(4) = \underbrace{\text{fibonacci}(3)}_2 + \underbrace{\text{fibonacci}(2)}_1$

$\hookrightarrow \underbrace{\text{fibonacci}(2)}_1 + \underbrace{\text{fibonacci}(1)}_1$

$\hookrightarrow \underbrace{\text{fibonacci}(1)}_1 + \underbrace{\text{fibonacci}(0)}_0$

$\hookrightarrow \underbrace{\text{fibonacci}(1)}_1 + \underbrace{\text{fibonacci}(0)}_0$

Verify -
 $\text{fibonacci}(4) \rightarrow 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89$
Index $\rightarrow 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \dots$

$4 = 4 - 3 = 1$

INTERVIEW-Q-1

Q) To find the sum of digits of a positive no. using recursion?

$$\text{Step 1: } \rightarrow \textcircled{1} [10] \rightarrow 1 + 0 = 1$$

" how do u get this ? "

$$\hookrightarrow \frac{10}{10} + 10 \% 10 \Rightarrow 1 + 0 = 1$$

$$[54] \rightarrow 54 / 10 + 54 \% 10$$

$$5 + 4 = \textcircled{9}$$

[54]

$$112 / 10 + 112 \% 10$$

$$11 / 10 + 11 \% 10 + 2$$

$$1 + 1 + 2 = \textcircled{4}$$

(Rec)

Step 2: $\rightarrow n = 0$ (base condition)

Step 3: $\rightarrow \begin{cases} \text{sumofdigits}(.) \\ \text{sumofdig}'(21, 5) \end{cases} = \textcircled{5}$

when num > deno
 $n \% d = a$ lenge
 method
 But if num < deno
 $n \% d = num$

def sumofdig's(n):

assert n >= 0 and n == int(n), "the no. has to be the integer of"

If $n == 0:$

return n

else:
 return sumofdig's(int(n / 10)) + n % 10

Print(sumofdig's(6)))

Interview - Q - 2

Q) Can calculate power of a no. using Recursion?

Step 1 →

$$\left. \begin{aligned} x^n &= x \cdot x \cdot x \cdots x \quad (\text{n times}) \\ 2^4 &= 2 \times 2 \times 2 \times 2 \\ x^0 &\times x^b = x^{a+b} \\ x^3 \times x^4 &= x^{2+4} \end{aligned} \right\} x^n = x \times x^{n-1}$$

Step 2 →

$$n = 0 \\ n = 1$$

Step 3 →

$$\begin{aligned} \text{power}(-1, 2) \\ \text{power}(3, 2) \\ \text{power}(2, 1) \\ \text{power}(2, -1) \end{aligned}$$

```
def power(base, exp):
```

assert exp >= 0 and int(exp) == exp, 'The exponent must be an integer only.'

if exp == 0:

return 1

~~else if~~ exp == 1:

return base

else:
 return base * power(base, exp - 1)

```
print(power(-1, -1))
```



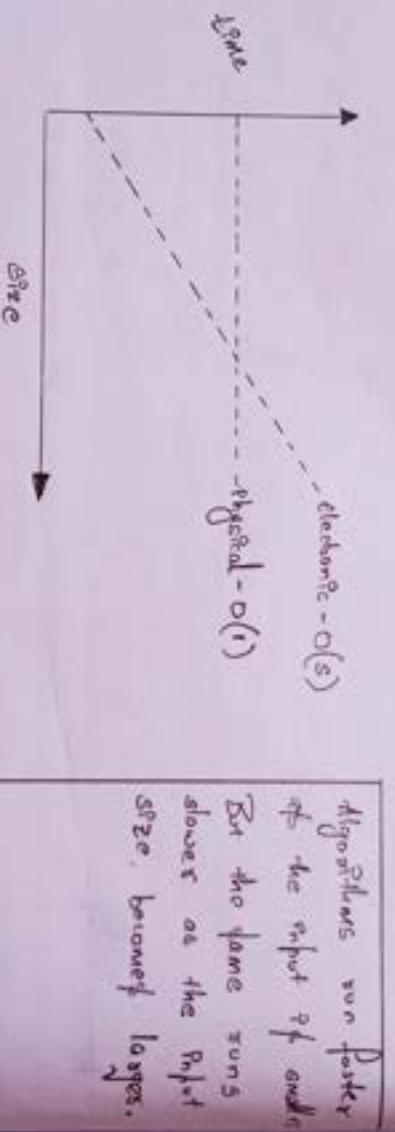
Big 'O' Notation

* Big-O Notation :

↳ + language & metric we use to determine the efficiency of algorithm.

• helps you to describe → how the time it takes to run your function grows at the log of input growth.

(Ex) In case of transfer of a file :
① way → electronic
② way → physical



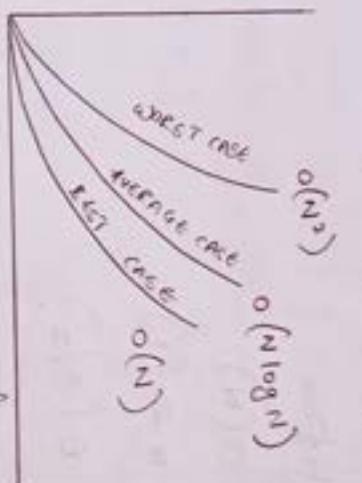
Algorithms run faster if the input is smaller. But the same runs slower as the Input size becomes larger.

(k) Height of [Punckine] —> $O(N)$. $O(N^2)$. $O(2^n)$

(l) To paint a wall of height (h) & width (w)

The runtime is :
Time complexity : $O(wh)$.

- ★ There are 3 denotat. in code of measuring an algorithm
→ Best case
→ Average case
→ Worst case



★ To express diff. phenomena of algo there are 3 diff. Notations

- ① Big-O : It is complexity that is going to be less or equal to the worst case.
- ② Big-Omega : It is complexity that is going to be at least more than the best case.
- ③ Theta : It is a complexity that is between bound of worst and the best case.

④ Algorithm Run-time Notation

(ii) If you have one element in an array & the element had a mark at the end then:

5	4	10	...	2	8	6	...	11	18	20	9
---	---	----	-----	---	---	---	-----	----	----	----	---

- Let's say to traverse through each element the time taken is 1 millisecond.
- Then: the time required to traverse the entire array would be $(n * 1)$ millisecond.

In terms of notations:

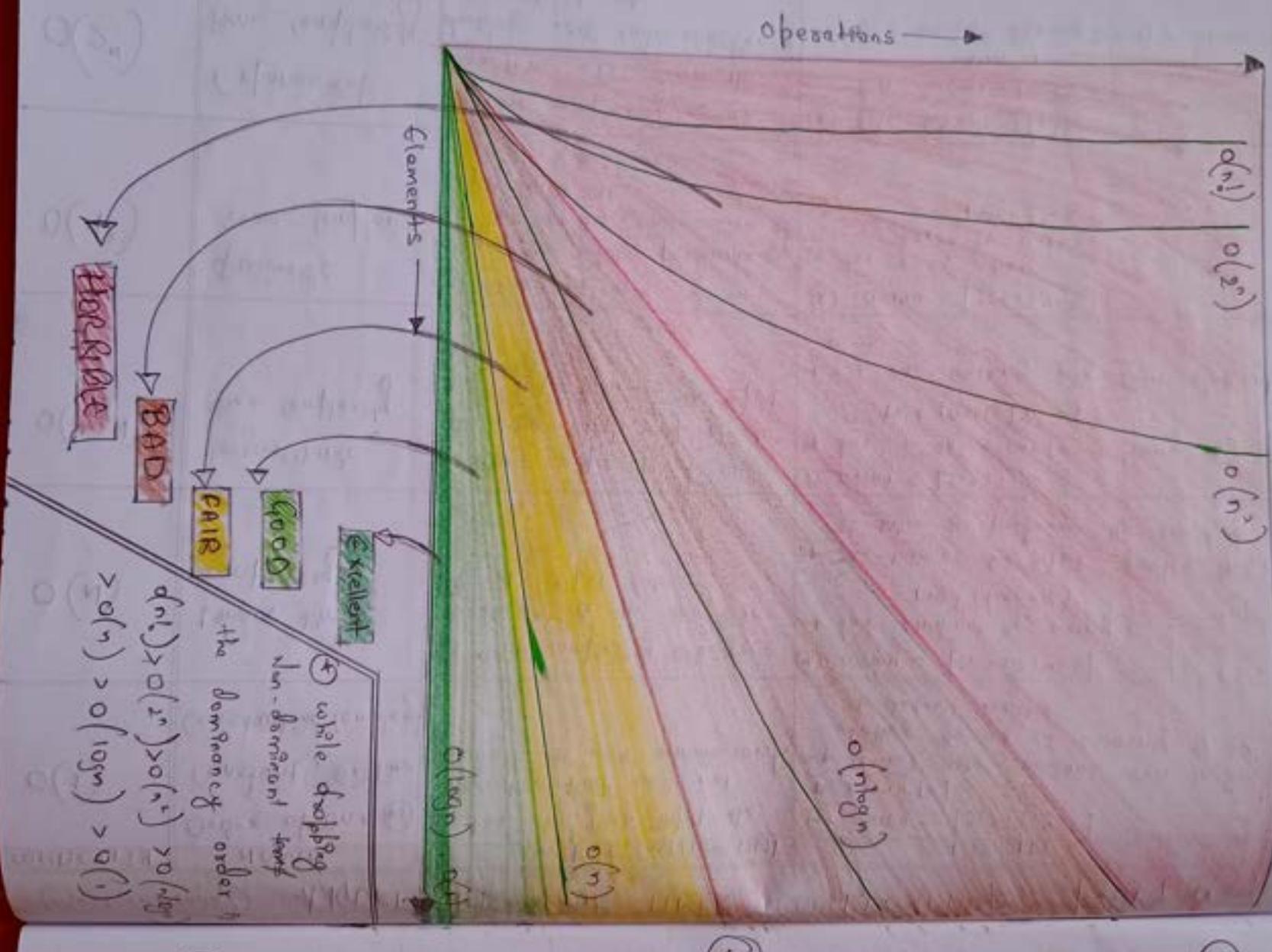
- Big O $\rightarrow O(n)$
- Big O $\rightarrow \Theta(n^1)$
- Big O $\rightarrow O(n^{1/2})$

For your knowledge you should know all 5 (O, Θ, Ω) But in interview the "Big O" of majority mostly asked.

(Most common) ALGORITHM - RUN - TIME - COMPLEXITY [BIG-O'S]

COMPLEXITY	NAME	SAMPLE EXPLANATION	EXAMPLE
$O(1)$	Order of One Constant Time Constant time complexity	★ For any given input the execution time will not change, it will remain const.	★ array = [1, 2, 3, 4, 5] → array[0] → It will take constant time to access 1st element whether the no. of elements is 50 or 10 or 100. It doesn't matter.
$O(N)$	Linear time complexity	★ Time complexity will grow in Directn O to the size of the input data	★ array = [1, 2, 3, 4, 5] → If it was my 3 element then it would be basic constant time complexity → for element in array: print(element) → The larger the input, greater the amount of time it takes to perform the execution.
$O(\log N)$	Logarithmic time complexity	★ Time complexity of an algorithm which runs O to the logarithm of the input size.	★ array = [1, 2, 3, 4, 5] → for i in range(0, len(array), 5): print(array[index]) → It is visiting only some elements, not all. → Not $O(n)$ <div style="border: 1px solid black; padding: 2px; margin-left: 20px;">Best example of Binary search - we find an element in a sorted array elements</div>
$O(N^2)$	Quadratic time complexity	★ Time complexity of an algorithm whose performance is O to the square size of input data set (Generally in nested loops)	★ array = [1, 2, 3, 4, 5] → for i in array: for j in array: print(i, j)
$O(2^N)$	Exponential time complexity	★ Time complexity of an algorithm whose growth doubles with each addition to the input set.	★ def fibonaci(n): def if n == 1: return n return fibonaci(n-1) + fibonaci(n-2)

Big-O - Complexity Chart



Space Complexity

- ① Time is not the only thing that matters in an algorithm.
- A 2nd concept to time complexity is the space complexity
Space Complexity → If a measure of the amount of working storage an algorithm needs.
 - ie → Also much memory in the worst case needed @ any point in the algorithm.
- ② If a is an array of size ' n '
 $a = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ \vdots & \vdots & \vdots \\ a & a & a \end{bmatrix}$ → space complexity $\rightarrow O(n)$
- ③ An array of size ' $n \times n$ '
 $a = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ \vdots & \vdots & \vdots \\ a & a & a \end{bmatrix}$ → s.c. $\rightarrow O(n^2)$
- ④ def sum(n):
if $n <= 0$:
return 0
else:
$$\text{return } n + \text{sum}(n-1)$$

Q) Drop Constant

E) Non-Dominant Term

① Drop Constant

Drop Non-Dominant terms

$$O(2N) \rightarrow O(N)$$

$$O(N + 100N) \rightarrow O(N)$$

$$O(2 * 2^N + 1000N^{100}) \rightarrow O(2^N)$$

② Why should we do that?

- ① It's very possible that $O(N)$ code is faster than $O(1)$ code for specific inputs.
- ② Different computers with diff architectures have diff constant factors.
 - like → fast computers → fast memory access → lower const
 - slow computers → slow → higher const
- ③ of algorithms with the same basic idea and computations complexity might have slightly diff const.
- ④ If in a computer subtraction of shows them multiplication then $a^*(b-c) \oplus \underbrace{a^*b - a^*c}_{\text{would be higher constant}}.$
 \therefore we ignore that.
- ⑤ If $n \rightarrow \infty$; constant factors are not really big deal.

ADD ✓ VS MULTIPLY
of Runtimes

```
for a in arrayA:  
    print(a)  
for b in arrayB:  
    print(b)
```

① add the Runtimes:
 $O(n + b)$

Logic

```
for a in arrayA:  
    for b in arrayB:  
        print(a,b)
```

② Multiply the Runtimes:
 $O(n * b)$

if your algorithm is in the form

"do this, then when you are all done, do that"
then you add the runtimes.

if your algorithm is in the form
"do this to each time you do that"
then you Multiply the runtimes.

How To Measure Time Complexity

using "Big-O"

Rules

No.	Description	Complexity
Rule 1	Any assignment statements and if statements that are executed once regardless of the size of the problem.	$O(1)$
Rule 2	A simple "for" loop from 0 to n (with no internal loops)	$O(n)$
Rule 3	A nested loop of the same type takes quadratic time complexity (i.e., $O(n^2)$)	$O(n^2)$
Rule 4	A loop in which the controlling parameter is divided by 2 at each step	$O(\log n)$
Rule 5	When dealing with multiple statements - just calculate them up	

(E)

sample array = [5, 4, 10, ..., 8, 11, 68, 12, ...]

def findBiggestNumber(sampleArray):

 biggestNumber = sampleArray[0] ————— $O(1)$

 for index in range(1, len(sampleArray)): ————— $O(n)$

 if sampleArray[index] > biggestNumber: ————— $O(1)$

 biggestNumber = sampleArray[index] ————— $O(1)$

print(biggestNumber) ————— $O(1)$

∴ Time complexity → $O(1) + O(n) + O(1) + O(1) + O(1)$

$O(n) + O(1))$

$\boxed{O(n)}$

using step notation:
E.g. non dominant term

How to measure

Recursive algo. that
Makes multiple calls

```
def f(n):
    if n == 1:
        return 1
    else:
        return f(n-1) + f(n-1)
```

Many think (u don't) 

- that if in previous function case the time complexity is $O(n)$; where the function calls itself once
- Here the function is calling itself twice then the time complexity will be $O(n^2)$.

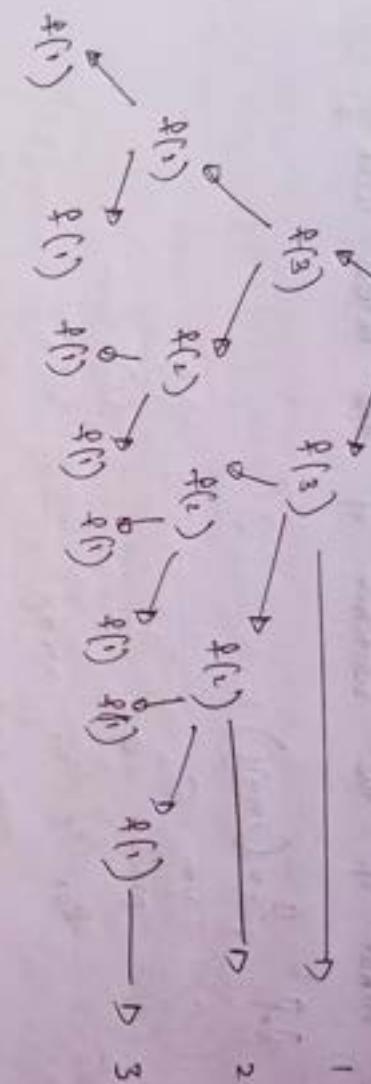
Abusing \rightarrow $n=4$

then:

$O(1)$

$f(n) \rightarrow 0$

1



level	node	can be expressed as
0	$f(0)$	2^0
1	$f(1)$	2^1
2	$f(2)$	2^2
3	$f(3)$	2^3
4	$f(4)$	2^4

$$x_0 = 2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^n$$

$$= 2^{n+1} - 1$$

[Since we go till $n-1$ not n]

$$= 2^n - 1 \rightarrow O(2^n)$$

General $\rightarrow O(b^{depth})$
 In our case
 branches $b = 2$ time to the $depth$ will be n
 $\therefore depth \ i.e. n = 4$
 $\therefore O(2^4) = O(16)$

Interview - Q - 1

Q) What is the runtime of the below code?

Def foo(array):

 sum = 0 O(1)

 product = 1 O(1)

 for i in array: O(n)

 sum += i O(1)

 for i in array: O(n)

 product *= i O(1)

Print ("Sum = " + str(sum) + ", Product = " + str(product))

$$\text{Runtime} = O(1) + O(n) = \boxed{O(n)}$$

($O(n)$)

Interview-Q - 2

② What is the runtime of the below code?

```
def printPoint (array):
    for i in array: _____ O(n)
    for j in array: _____ O(n)
        print(str(i) + ", " + str(j)) —— O(1)
```

Runtime = $O(n^2) + O(n) + O(1)$
= $O(n^2)$

Interview - Q- 3

③ What is the runtime of the below code?

```
def printUnorderedPairs(array):
    for i in range(0, len(array)):
        for j in range(i+1, len(array)):
            print(array[i] + "," + array[j])
```

Runtime could be cal. in ② ways :

① Counting the iterations

for inner loop ($n = \text{len of array}$)
 1st iteration $\rightarrow n-1$
 2nd iteration $\rightarrow n-2$

$$\therefore \text{Runtime} = (n-1) + (n-2) + (n-3) + \dots + 2 + 1$$

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

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

$$= \frac{n^2}{2} + \frac{n}{2}$$

$$= n^2$$

$$\therefore \boxed{\text{Runtime} = O(N^2)}$$

② Average Work

✓ if the outer loop runs b times

for Inner loop :

let's say for the inner loop

- 1st iteration of outer loop \rightarrow 10 times

- 2nd iteration \rightarrow 9

$$\text{Average} = \frac{10}{2} = 5$$

∴ Instead of 10 times if it was been n times the average

$$\text{Average} = \frac{n}{2}$$

$$\therefore \text{Runtime} = n * \frac{n}{2} = O \frac{n^2}{2} = O(N^2)$$

Interview - Q - 4

(Q) What is the Runtime?

```
def printUnorderedPairs(arrayA, arrayB):
    for i in range(len(arrayA)):
        for j in range(len(arrayB)):
            if arrayA[i] < arrayB[j]:
                print(str(arrayA[i]) + " " + str(arrayB[j]))
```

for i in range(len(arrayA)).
for j in range(len(arrayB)).
if arrayA[i] < arrayB[j].
print

for every n of
1st loop the 2nd } is $O(n^2)$
loop runs n times

$O(1)$ Complexity
 $O(n)$ wrong.

1st thing \rightarrow there are 2 diff array input arguments in the loop (i.e. for outer & inner).

so if outer loop runs for say (m) times if $len(arrayA) = m$

& inner loop runs for say (n) times if $len(arrayB) = n$

then for every this do that $\Rightarrow O(mn)$

Interview - 8 - 5

⑤ Runtime of the below code?

```
def printUnderscores(arrayA, arrayB):
    for i in range(len(arrayA)):
        for j in range(len(arrayB)):
            for k in range(0, 10000):
                print(str(arrayA[i]) + "," + str(arrayB[j]))
```

$O(ab)$

$O(1)$

$O(1)$

[\because it take constant arguments]

\therefore Runtime = $O(ab)$

Interview - Q - 6

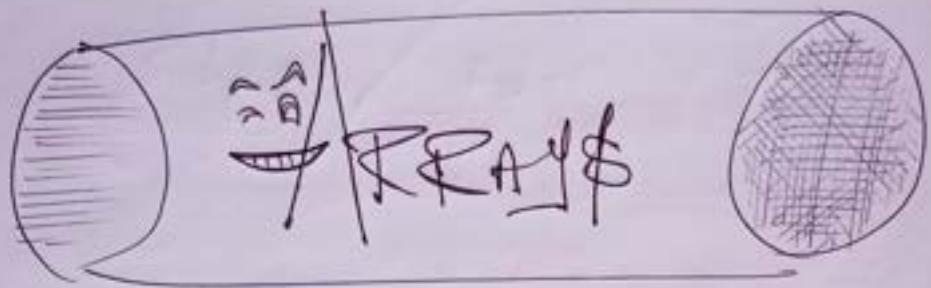
⑥ Runtime of b/l code?

def reverse(array):

```
for i in range(0, int(len(array)/2)):  
    other = len(array) - i - 1  
    temp = array[i]  
    array[i] = array[other]  
    array[other] = temp  
print(array)
```

Runtime → $O(N)$

[1, 2, 3, 4, 5]



are Data structures

An array } collection of elements
} each identified by at least one array index.

- Ex →
• Pens in a stand
• Books in a shelf
• People standing in a queue

Properties

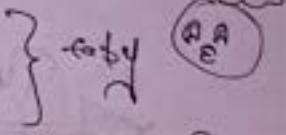
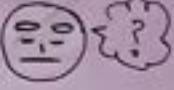
- Can store data of specified type only.
- Element are contiguous i.e. No gap → next to each other
- Each element has unique index.
- Size of array is predefined
- Cannot be MODIFIED.



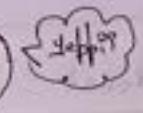
4	8	7	6	5
[0]	[1]	[2]	[3]	[4]

→ [only int type]

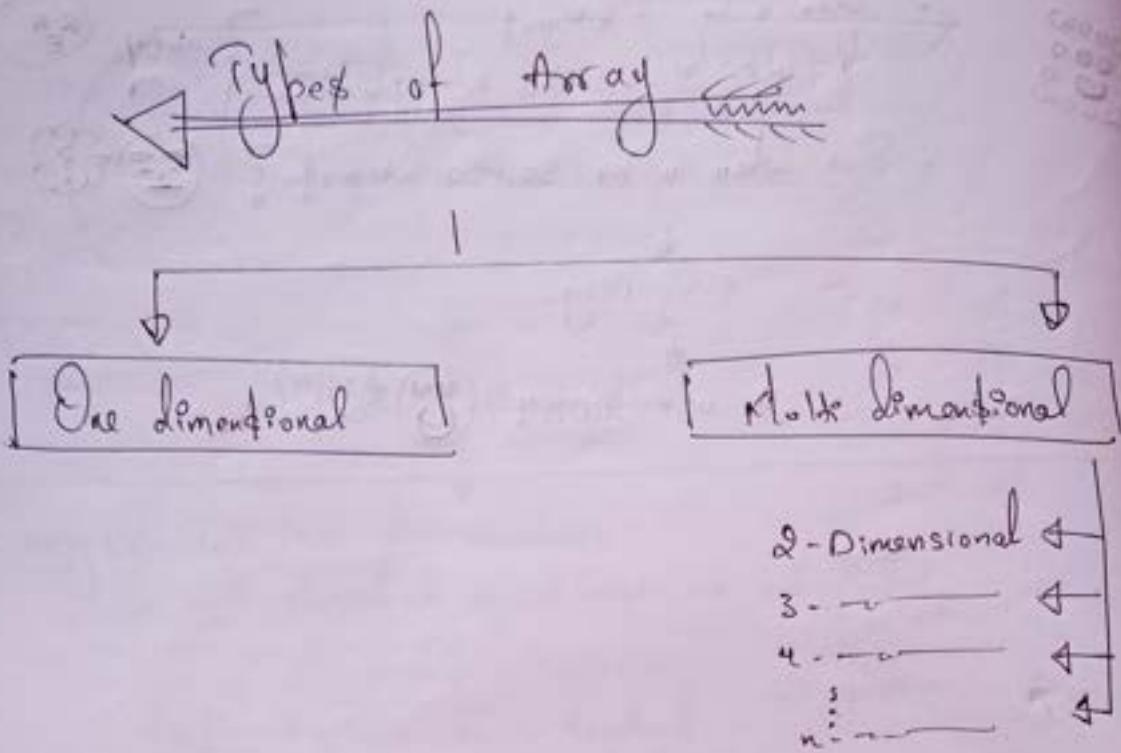
WOOOW
WOOOW

- v When u hv 3 integers
- v use 3 variables to store them } 
- v But when u hv 50,000 integers? 

Goldy-Piggy.

v. use "ARRAY"  

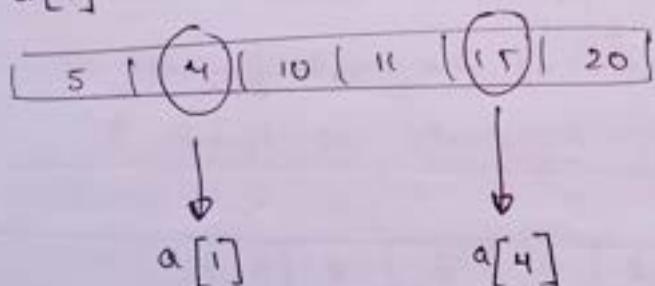




1-Dimensional | Linear Array:

An array with a bunch of values having been declared with a single index.

$a[i]$.



2-Dimensional array

An array with a bunch of values having been declared with double index.

$a[i][j]$

	[0]	[1]	[2]	[3]	[4]	
[0]	8	4	6	2	9	$\rightarrow a[0][4]$
[1]	10	9	8	4	25	
[2]	21	5	3	31	17	

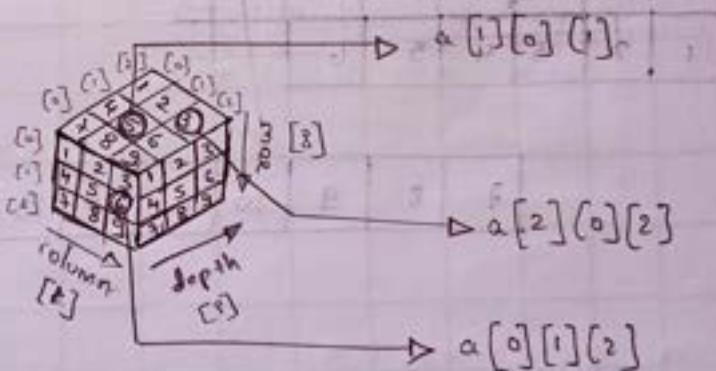
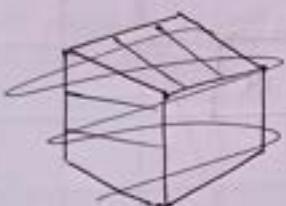
$\downarrow \quad \quad \quad \downarrow$

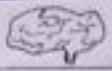
$a[2][1] \quad \quad \quad a[1][3]$

3-Dimensional array

An array with a bunch of values having been declared with triple index.

$a[i][j][k]$





◇ ARRAY in MEMORY ◇

① 1-D Array

[1, 2, 3, 4, 5, 6]

1	2	3	4	5	6
---	---	---	---	---	---

(ptr) [Memory]

② 2-D Array

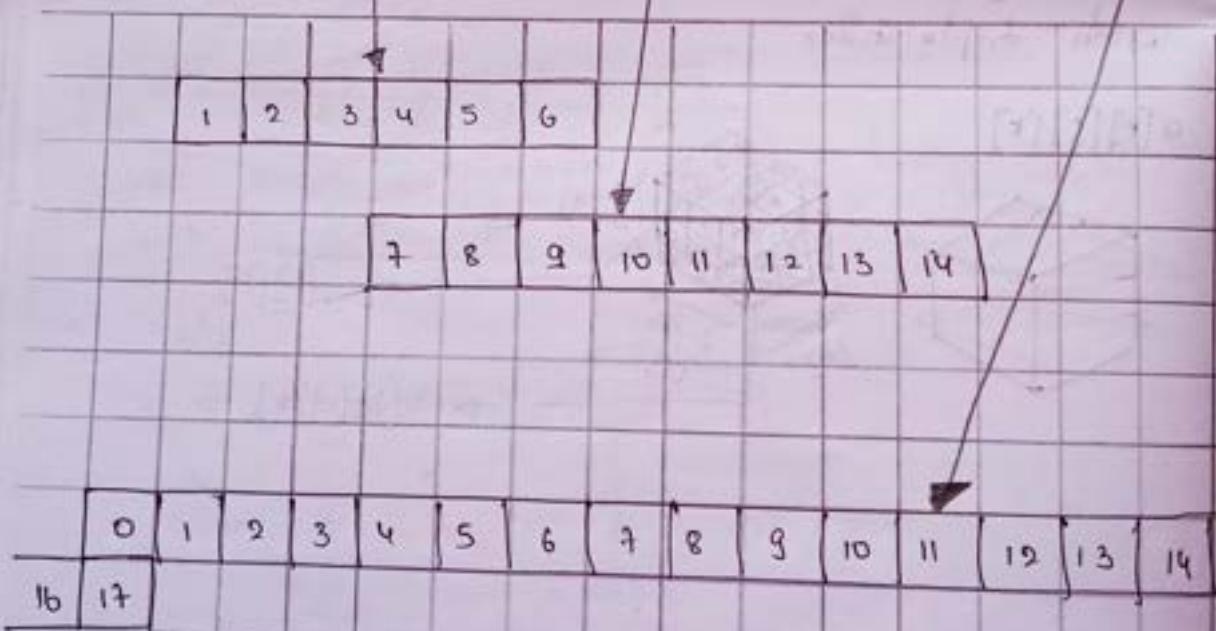
7	8	9	10
11	12	13	14

③ 3-D Array

[[[0, 1, 2],
[3, 4, 5]],

[[6, 7, 8],
[9, 10, 11]],

[[12, 13, 14],
[15, 16, 17]]]



- ④ In the memory the array can be stored honywhere ; it can start from anywhere it want
- But it is always contiguous
 - Even a 2 or 3 - Dimensional Array are arranged in the Memory like the 1-Dimensional Array.

* Contiguous means not
(disjoint), non-contiguous

CREATING AN ARRAY

- Assign it to a variable
- Define the type of elements that it will store
- Define its size. (optional)

(Rem) —>

i → int

f → float

d → Double (in python - float).

```
from array import *
arrayName = array(typecode, [initializers])
```

(Ex):

```
from array import *
```

```
arr arr1 = array('i', [1, 1, 3, 4])
```

```
arr2 = array('f', [1.1, 2.2, 3.3, 4.4])
```



Insertion In Array

How things work (inside):

[0] [1] [2] [3] [4] [5] [6]
 [1 | 2 | 3 | 4 | 5 |]

Rem: All these are done automatically, u don't have to do it.

Say u have an array [1, 2, 3, 4, 5] with 2 empty spaces.
 Q: When u add 2 elements: it's easy
 u just do $a[5] = 6$ & $a[6] = 7$

But what if it's like:

u wanna add element at $a[0]$
 Rem → Array is not mutable - u cannot change the element

But u simply write $a[0] = 10$

Q: It gives [10 | 1 | 2 | 3 | 4 | 5 |]

e.g. [1 | 2 | 3 | 4 | 5 |]

Q: When u say $a[0] = 10$
 all the elements are moved to 1 step index further
 which makes an empty $a[0]$ index.

But what if it's like:

u wanna insert an element at $a[0]$ & the array is full

i.e. [1 | 2 | 3 | 4 | 5 | 6 |]

Then all those elements are copied to a new, bigger array

i.e. [1 | 2 | 3 | 4 | 5 | 6 | 7 |]

the $a[0] = 10 \Rightarrow [10 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |]$

```
import  
from array import *
```

```
arr1 = array('i', [1, 2, 3, 4, 5])
```

after

~~arr1[5] = 10~~

arr1.insert(5, 10) *insert element at index 5*

[1, 2, 3, 4, 5, 10]

arr1.insert(0, 10)

[10, 1, 2, 3, 4, 5]

arr1.insert(3, 10)

[1, 2, 3, 10, 4, 5]

Time Complexity :

① Inserting @ the beginning of the arry $\rightarrow O(n)$
as it has to shift all the 'n' elements to next index

② Inserting @ the final of the arry $\rightarrow O(1)$

③ Inserting an element to a full-arry $\rightarrow O(n)$
as it has to create a new-larger arry & transfer the elements to the old arry

How to Traverse an Array



```
def traverseArray(array):  
    for i in array:  
        print(i)
```

Simple!!

- ★ Time complexity $\longrightarrow O(n)$
- ★ Space complexity $\longrightarrow O(1)$ (\because no additional space is required)

Accessing an element in array

```
a = array ('r', [1, 2, 3, 4, 5])
```

$a[5] =$

error (≥ 5)

$a[2]$

3

- ★ Time complexity $\longrightarrow O(1)$
- ★ Space complexity $\longrightarrow O(1)$

Finding an element

( finding Nemo <<

```
def searchingInarray (array, value):  
    for i in array:  
        if i == value:  
            return array.index(value)  
    return "The element does not exist in Array"
```

① Time complexity $\rightarrow O(n)$

② Space complexity $\rightarrow O(1)$

Deleting an Element

a =

1	2	3	4	5	6
---	---	---	---	---	---

if we delete/remove 4 then \rightarrow

1	2	3	5	6
---	---	---	---	---

a =

1	2	3	5	6
---	---	---	---	---

p.e

from array import *

arr1 = array('i', [1, 2, 3, 4, 5, 6])

arr1.remove(4)

print(arr1)

array('i', [1, 2, 3, 5, 6])

Element Not in index

① Time complexity $\rightarrow O(n)$

② Space complexity $\rightarrow O(1)$

TIME & SPACE

Complexity in 1-Dimensional

ARRAY

SUMMARY

OPERATION	TIME COMPLEXITY	SPACE COMPLEXITY
Creating an empty array	$O(1)$	$O(n)$ if has allocate n space in the memory
Inserting a value in an array	$O(1) \rightarrow @ \text{end}$ $O(n) \rightarrow @ \text{starting or somewhere inside}$	$O(1)$
Traversing a given array	$O(n)$	$O(1)$
Accessing a given cell	$O(1)$	$O(1)$
Searching a given value	$O(n)$	$O(1)$
Deleting a given value	$O(1) \rightarrow \text{last value}$ $O(n) \rightarrow \text{Beginning or Middle}$	$O(1)$

NOTE

- ✓ In searching a value in an ARRAY
 - ✓ If the value == data in ARRAY @ the 1st iteration i.e
the required value is @ 1st index
then time complexity = $O(1)$
- But why do we write the time complexity of —
as $O(n)$
- \because Big-O tells up the time complexity @ worst case
 $\therefore O(n) \leq O(n)$.

H/W

- ① Create an array & traverse it.

```
from array import *
arr = array ('i', [1, 2, 3, 4, 5])
for i in arr:
    print (i)
```

Time.C - O(n)

- ② Access individual elements through index.

```
from array import *
arr2 = array ('f', [1.1, 2.2, 3.3, 4.4, 5.5])
index_value = int (input ("Enter the index value : "))
def index_value (index_value):
    for i in arr2:
        if index[i] == index_value:
            return i
```

Next
Topic
out

2
Rev (2)
□ = 2 X
2 = 2
5

③ Append & Insert:

arr3 = array ('P', [1, 2, 3, 4, 5])

arr3.append(6)

print(arr3)

[1, 2, 3, 4, 5, 6]

arr3.insert(0, 7)

print(arr3)

[11, 1, 2, 3, 4, 5]

METHODS

IN

ARRAY

• append()

• extend()

• insert()

• fromlist()

• pop()

• remove()

• index()

• buffer_info()

• count()

• tolist()

④ Extend:

arr4 = array ('P', [1, 2, 3, 4])

arr5 = array ('P', [5, 6, 7, 8])

arr4.extend(arr5)

print(arr4)

[1, 2, 3, 4, 5, 6, 7, 8]

} merged
the 2 arrays.

⑤ Add list items to an array

templist = [8, 18, 88, 888]

arr6 = [1|2|3|4|5]

arr6.fromlist(templist)

print(arr6)

[1, 2, 3, 4, 5, 8, 18, 88, 888]

fromlist

⑥ Remove :

a = array(1, 2, 3, 4, 5, 6)

a. remove(2)

print(a)

[2, 4, 5, 6]

⑦ Pop:

a = array(1, 2, 3, 4, 5)

a. pop()

print(a)

[2, 3, 4]

⑧ index() : (\rightarrow to find index of an element)

a = array(1, 2, 3, 4, 5, 6)

a. index(4)

2

⑨ Reverse ():

a = array(1, [1, 2, 3, 4, 5])

a.reverse ()

print(a)

[5, 4, 3, 2, 1]

⑩ buffer_info (): → D (Gives a tuple of
(address, no. of elements))

a = array(1, [1, 2, 3, 4, 5])

print(a.buffer_info ())

(104680419029464, 5)

⑪ Count (): → how many times is
occurrence of an element.

a = array(1, [1, 2, 3, 3, 3, 3, 4])

print(a.count(3))

4

(12)

Converting:

★ Array to string \Rightarrow toString()

Array to string {
 $a = \text{array}('!', [1, 2, 3, 4, 5])$
 $b = a.\text{toString}()$

will give u something weird but it has converted to check convert it back

String to array {
 $bac = \text{arr}('!')$ ————— (empty Array)
 $c = \text{fromstring}(b)$
 $\text{print}(c)$
 $[1, 2, 3, 4, 5]$

1114 for list \rightarrow tolist()

(13) Slicing :

$a = \text{array}('!', [1, 2, 3, 4, 5]) \rightarrow$ (111 for list)
 $\text{print}(a[1:4])$
 $[2, 3, 4, 5]$

2-Dimensional ARRAY

When do we use this?

- Book My Show has slots of seats to select this is a 2-D Array.

$a[i][j]$

→ Row

→ column

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Using `numpy` for arrays would make everything easy.

```
import numpy as np
```

```
twoDarray = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
print(twoDarray)
```

```
[[1, 2, 3]
 [4, 5, 6]
 [7, 8, 9]]
```

A
2-D
ARRAY

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

(118) 0 → (1000 999 999) 0

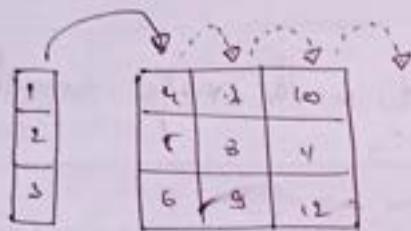
Insertion in 2-D Array

When

if u want to insert a row or column in a 2-D array
what happens if

1	2	3	4
5	6	7	8
9	10	n	12

Adding column:

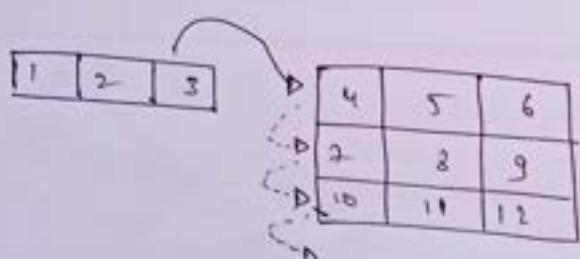


the next column has to move 1 step right

★ TIME COMPLEXITY → $O(mn)$

↑ no. of rows
↓ no. of columns

Adding row:



the next row moves 1 step down

★ TIME COMPLEXITY → $O(mn)$

But this is time consuming



getting
late!!!

how to use?

```
import numpy as np
```

```
arr1 = np.array ([[1,2,3], [4,5,6], [7,8,9]])
```

```
print(arr1)
```

```
newarr1 = np.insert (arr1, 0, [8,88,888], axis=1)
```

```
print(newarr1)
```

↓ ↓ ↓ ↓
where index what u axis=1 → column
want to want to went to axis=0 → row
add add insert

Simply → insert [8,88,888]

@ index [0] of column in
arr1

```
[ [1,2,3]
```

```
[4,5,6]
```

```
[7,8,9]]
```

```
[ [8,1,2,3]
```

```
[88,4,5,6]
```

```
[888,7,8,9]]
```

Now row wise;

```
newarr1 = np.insert (arr1, 1, [[7,77,777]], axis=0)
```

```
[ [1,2,3]
```

```
[7,77,777]
```

```
[4,5,6]
```

```
[7,8,9]]
```

Append

Name as insert but it will add @ the end
⑥ last index so No need to specify index.

newarr1 = np.append (arr1 , [[7, 77, 777]] , axis=0)

print (newarr1)

```
[ [ 1, 2, 3]
  [ 4, 5, 6]
  [ 7, 8, 9]
  [ 7, 77, 777] ]
```

④ Time Complexity $\rightarrow O(1)$

to add @ either last of row or column

Accessing elements

```
def accessElement (array, rowIndex, colIndex) :
    if rowIndex >= len(array) or colIndex >= len(array[0]):
        print ('out of index')
    else:
        print (array [rowIndex] [colIndex])
```

arr1 = np.array ([[1,2,3], [4,5,6], [7,8,9]])

arr1

accessElement (arr1 , 1 , 0)

★ TIME COMPLEXITY $\rightarrow O(1)$
for this code

Traversing through the
Array

`array = np.array ([[1,2,3], [4,5,6], [7,8,9]])`

```
def traversearray (array):
    for i in range (len (array)):
        for j in range (len (array [0])):
            print (array [i] [j])
```

1 {
 2 | len (array) = 3 ^{row}
 3 | len (array [0]) = 3 ^{col}
 4 }
 5
 6
 7
 8
 9

1	2	3
4	5	6
7	8	9

Ⓐ TIME COMPLEXITY $\rightarrow O(mn)$

$m \rightarrow$ no. of rows $n \rightarrow$ no. of columns $O(mn) = O(n^2)$

Ⓑ Space C $\rightarrow O(1)$

Searching for an Element

```
arr1 = np.array ([[1,2,3], [4,5,6], [7,3,9]])  
def searchelement (array, value):  
    for i in range (len (array)):  
        for j in range (len (array [0])):  
            if array [i] [j] == value:  
                print ('Value is located at ' + str (i) + str (j))  
            print  
            else:  
                print ('Element is not found')  
  
print (searchelement (arr1, 6))
```

- ★ if there are more than one '6' then it will return the index value of 1st occurrence

Time complexity $\rightarrow O(mn)$

Space complexity $\rightarrow O(1)$

Deleting A Row/Col IN ARRAY

Time consuming

Deleting column:

④

	1	2	3
4	5	6	
7	8	9	

Deleting row:

	1	2	3
4	5	6	
7	8	9	

★ Time complexity $\rightarrow O(mn)$ row
column
if $m=n \Rightarrow O(n^2)$

`arr1 = np.array ([[1, 2, 3], [4, 5, 6], [7, 8, 9]])`

`print (arr1)`

`newarr1 = np.delete (arr1, 0, axis=1)`

`print (newarr1)`

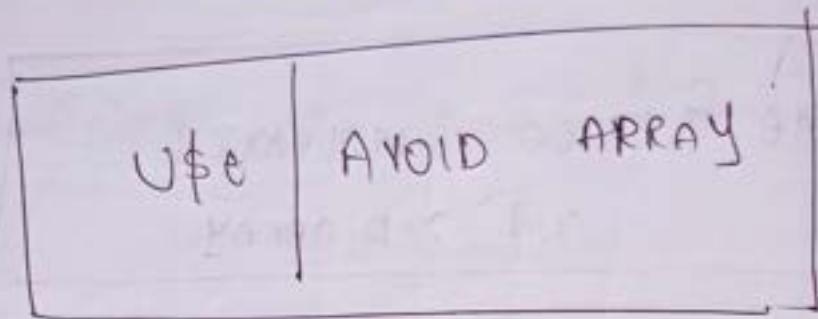
$axis=0 \rightarrow$
 $1 \rightarrow 0$ value

★ Time c.c $\rightarrow O(mn)$ row
column

TIME & SPACE COMPLEXITY

IN 2-D ARRAY

Operation	TIME - STONE	SPACE - STONE
Creating an empty array	$O(1)$	$O(mn)$
Inserting a col / row in an array	$O(mn)$ $O(1)$ @last	$O(1)$
Traversing a given array	$O(mn)$	$O(1)$
Accessing a given cell	$O(1)$	$O(1)$
Searching a given value	$O(mn)$	$O(1)$
Deleting a given value	$O(mn)$ $O(1)$ @last	$O(1)$



when to use :

- ↳ To store multiple variables of same datatype
- ↳ Random access

when to avoid :

- ↳ same datatype elements
- ↳ Reserve memory

Ordered
Mutable
Allow duplicates
Indexed



- ORDERED COLLECTION OF ITEMS

[Milk, eggs, cheese, Butter]

elements / Items

- [10, 20, 30] → int

['10', '20', '30'] → str

[10, '10', 10.0] → (int, str, float)

[10, [10, 11], 12] → nested list → len = 3

Accessing Elements

a = [1, 2, 3, 4]

a[3]

4

a[-2]

3

Checking items in a list

if True — it exist in the list

if False — not it doesn't

a = ['milk', 'curd', 'Butter']

print ('milk' in a)

True

print ('Eggs' in a)

False

Traversing through the list

a = ['Me', 'You', 'We', 'Us']

for i in a:
 print(i)

Updating elements:

a = [0] [1] [2] [3]
[1, 2, 3, 4]

a[2] = 6

a[1] = 7

print(a)

[1, 7, 6, 4]

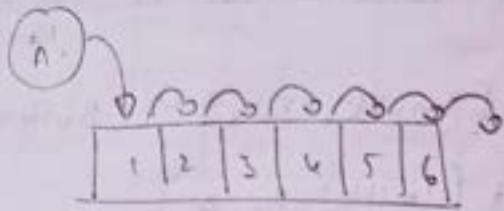
★ Time. C → O(1)

★ Space. C → O(1)

Insert

.insert(index, element)

a = [1, 2, 3, 4, 5]



a.insert(0, 6)

print(a)

[6, 1, 2, 3, 4, 5]

★ T.C $\rightarrow O(n)$

Append

.append(element)

a.append('B')

print(a)

[6, 1, 2, 3, 4, 5, 'B']

★ T.C $\rightarrow O(1)$

★ S.C $\rightarrow O(1)$

Extend

.extend(theList)

~~o.o.~~

b = ['Me', 'You']

a.extend(b)

print(a)

[6, 1, 2, 3, 4, 5, 'B', 'Me', 'You']

★ T.C $\rightarrow O(n)$

n -> Depends on the elements present

★ S.C $\rightarrow O(n)$

Slicing

a = ['a', 'b', 'c', 'd', 'e']

print(a[0:2])
['a', 'b']

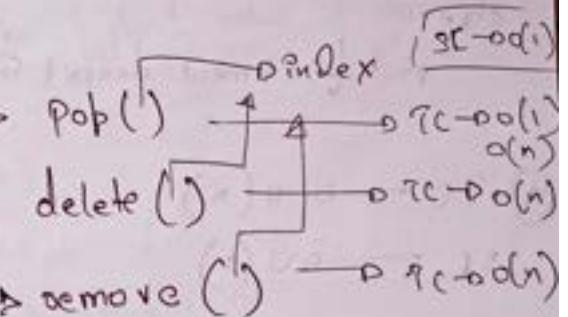
a[0:2] = 'm'

print(a)
['m', 'c', 'd', 'e']

Deleting

④ a = [1, 2, 3, 4]

a.pop(2) a.pop()
print(a) 4



→ will return pop the deleted item

④ a.delete(3) → will not return the deleted item

print(a)
[1, 2, 3] [1]

④ a.remove(4)
print(a)
[1, 2, 3]

Searching an Element

[IN operator]

[Linear search]

a = [10, 20, 30, 40, 50]

```
if 20 in a:  
    print (a.index(20))  
else:  
    print ('Element doesn't exist')
```

TC → O(n)
SC → O(1)

```
for i in a:  
    if i == value:  
        print (a.index(i))  
    else:  
        print ('Element doesn't exist')
```

TC → O(n)
SC → O(1)

[+ operators : concatenation]

a = [1, 2, 3]

b = [4, 5, 6]

c = a + b

print (c)

[1, 2, 3, 4, 5, 6]

`* Operator`

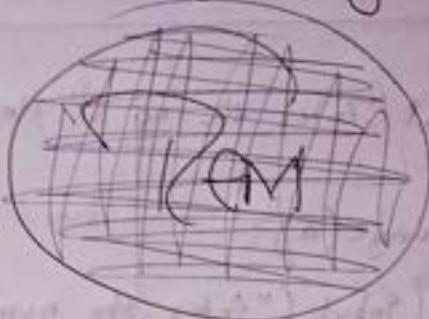
(Repeating an element)

`a = [2]`

`a = a * 4`

`print(a)`

`[2, 2, 2, 2]`



`len()`

→ returns the length (no. of elements)

`max()`

→ returns the highest value in the list

`a = [1, 2, 3, 4]`

`print(a.max())`

`min()`

→ to find minimum no.

`print(a.min())`

`sum()`

→ to find sum of elements.

`if`

`if` `elif` `else`

(single-line if)

`[condition, condition, condition]`

★ To take a list & return average

```
a = int(input("Enter the no. of elements in the list:"))
b = []
for i in range(a):
    c = int(input("Enter the number:"))
    b.append(c)
average = sum(b) / len(b)
print('average')
print(f'average is : {average}')
```

Convert str to list

```
a = 'ananth'
b = list(a)
print(b)
['a', 'n', 'a', 'n', 't', 'h']
```

Split

```
a = 'spam spam spam'
print(a.split())
['spam', 'spam', 'spam']
```

a = 'spam-spam1-spam2'

thing = '-'

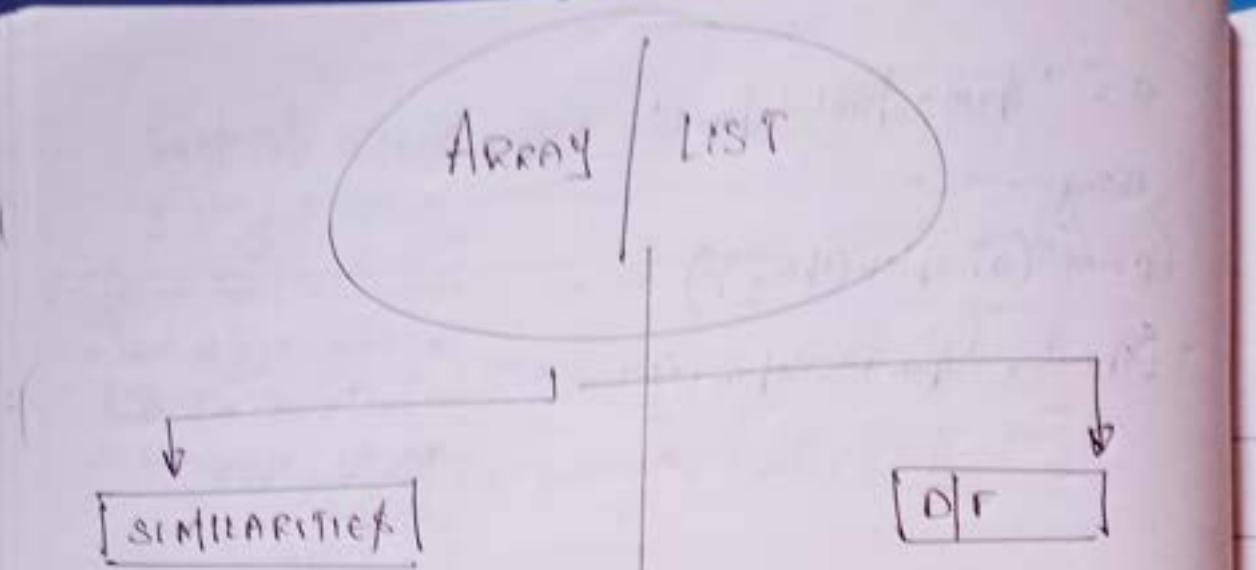
print(a.split(thing))

['spam', 'spam1', 'spam2']

→ there's no space

entire thing is 1

so if u want to split
u have to declare a
variable & put that
as the argument in
.split()



- ① Both data structures are mutable
- ② Both can be indexed & iterated through
- ③ They can be both sliced

④ Arrays are better than list for Arithmetic operations

Ex: arr = np.array([1, 2, 3, 4])

arr[1] = [1, 2, 3, 4]

print(arr[1])

[0.5, 1.0, 1.5, 2.0]

(b.t)

print(tots[1])

ERROR (can't do)

⑤ List can have diff data types but array cannot.

Ex even if u do like

arr = np.array([1, 2, 3, 4, 'a'])

print(arr)

[1, 2, 3, 4, 'a']

it will automatically convert all to str.

Time-space Complexity

of list

operation	<u>Time-space()</u>	<u>Space-space()</u>
Creating a list	$O(1)$	$O(n)$
Inserting a value in a list	$O(1) \mid O(n)$	$O(1)$
Traversing a given list	$O(n)$	$O(1)$
Accepting a given cell	$O(1)$	$O(1)$
Searching a given value	$O(n)$	$O(1)$
Deleting a given value	$O(1) \mid O(n)$	$O(1)$

Q - ①

① Find the Missing number in a list of numbers

say $a = [1, 2, 3, 4, 5, 7, 8, 9, 10]$

Sol:

def missnum(list, n):

sum1 = sum(list)

sum2 = $\frac{(n \star (n+1))}{2}$

print ('the missing number is', {sum2 - sum1})

print (missnum(a, 10))

Don't miss

Q - ②

② Write a program to find all pairs of integers whose sum is equal to a given number.

(Ex) $[2, 6, 3, 9, 11]$
if the no. is 9 then, $\text{ans} \rightarrow [6, 3]$

things that should come in your head !!!

- Does array contain only +ve or -ve numbers?
- what if the same pair repeats twice, should we print it every time?
- If the reverse of the pair is acceptable
Ex: can we print (4,1) & (1,4) if given sum is 5
- Do we need to print only distinct pairs?
does (3,3) is a valid pair for given sum of 6
- How big of the array?

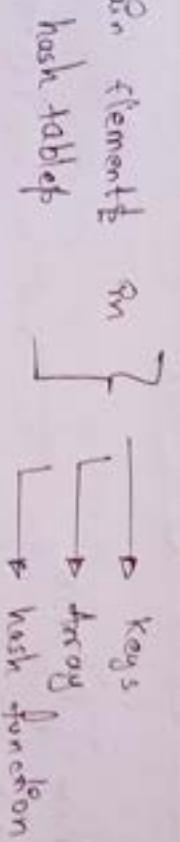
Dictionary

- A collection of unordered, changeable & indexed (or keyed) elements.

★ - mydict1 = { "key" : "value"}
- mydict2 = {} - Empty Dictionary.
- mydict3 = {"flash", "speed", "Ironman", "Rich", "Hulk", "Boo"}
mydict3["flash"]
"speed"

④ Time.C for Accepting Element → o()

Dictionary in Memory

- (*) Dictionary are indexed by keys & key can be taken at associative way
- (*) Python Dictionaries are implemented using Hash Tables
- (*) Hash Table: a way of doing key-value lookups.
 - the values are stored in an array
 - then u use Hash function to find the index of array cell that corresponds to the key-value pair.
- (*) Hash increases the performance of the Dictionary
- (*) 3 main elements in 
 - Keys
 - array
 - hash function
- (*) A good hash function ~~minimizes~~ minimizes the no. of collisions.
- (*) Of keys having same hash value
 - If collisions occur the element would be added to the array under next element having same index using LINKED LIST

Updating a dictionary

```
my_dict = {'name': 'Anand', 'age': 18}
```

```
my_dict['age'] = 20
```

```
print(my_dict)
```

```
{'name': 'Anand', 'age': 20}
```

Addition of new key-value

```
my_dict['place'] = 'India'
```

```
print(my_dict)
```

```
{'name': 'Anand', 'age': 20, 'place': 'India'}
```

SC → O(1)

SC → O(n+k)

Traversing through a dictionary

dict.items() (dict):

for key in dict:

print(key, dict[key])

* TC - O(n), SC - O(1)

for value in dict.values():

for key, value in dict.items():

if key == "key":

dict["key"] = value

* TC - O(n), SC - O(1)

return "the value doesn't exist."

Deleting an element

```
mydict = {'name': 'month', 'age': '20', 'place': 'India'}
```

when key is not in the dictionary

```
mydict.pop('name')  
print(mydict)  
{'age': '20', 'place': 'India'}
```

popitem()

randomly picks & deletes any 1 of the key-value pair

```
mydict.popitem()  
print(mydict)  
{'name': 'month', 'place': 'India'}
```

clear()

will clear all the elements in the dictionary

```
mydict.clear()  
print(mydict)  
{}
```

del

A while

del mydict

it will delete all the elements

But when u write after

print (~~del~~ mydict)

it will give u an error

④ $\text{TC} \rightarrow o(.) \rightarrow \text{Average rate}$

$\rightarrow oo(n) \rightarrow \text{Amortized worst case}$

⑤ $SC \rightarrow o(.)$

SOME MORE METHODS

```
mydict = {'name': 'Edy', 'age': '32', 'place': 'London'}
```

• `copy()`

→ doesn't change the original dictionary

• `copy()`

→ No argument
→ will copy all the elements to the assigned new variable

```
newdict = mydict.copy()
```

`print(newdict)`

```
{'name': 'Edy', 'age': '32', 'place': 'London'}
```

• `fromkeys()`

→ takes 2 arguments → dictionary • `fromkeys(keys, value)`

→ will create a new dictionary from the keys & sequences of values (optional) provided

↳ `newdict = dict.fromkeys(['name', 'age'], 0)`

```
print(newdict)
{'name': 0, 'age': 0}
```

`print(newdict)`

```
{'name': 0, 'age': 0}
```

• `get(key, value)` → `print(mydict.get('age', 88))`

3.2 ~~print~~

↳ if the key doesn't exist it will give u the specified value i.e. `print(mydict.get('city', 33))`

(But) if even the value is not specified then → None
`print(mydict.get('city'))`
None

`.items()`

—> returns every No argument

returns a tuple of key-value pair of all elements in the array. dictionary.

`print(mydict.items())`

`dict_items([('name', 'Edy'), ('age', 26), ('place', 'London')])`

`.keys()`

—> No argument

returns a list of keys in the Dictionary

`print(mydict.keys())`

`dict_keys(['name', 'age', 'place'])`

`.setdefault(key, default_value)`

—> 2 arguments
optional

—> returns the value of the key specified
if the key doesn't exist in the dictionary
it will return the default value given specified.

`print(mydict.setdefault('name', 'Unknown'))`

`Edy`

(But)
`print(mydict.setdefault('name8', 'Unknown'))`

`Unknown`

• values()

→ No arguments

• list of all values in the dictionary

Dictionary

↳ print(mydict.values())

[‘edg’, 26, ‘london’]

• update(other_dictionary)

→ other Dictionary as argument

↳ will merge the newdictionary with the 1 argument

↳ newdict = {‘a’: 1, ‘b’: 2, ‘c’: 3}

mydict.update(newdict)

print(mydict)

{‘name’: ‘edg’, ‘age’: 26, ‘place’: ‘london’, ‘a’: 1, ‘b’: 2, ‘c’: 3}

Dictionary operators

In Built-in functions

```
mydict = { 'name': 'Gdy', 'age': 30, 'place': 'LA'}
```

In - operator

→ where something appears as a key in the given dictionary or not
↳ if there → True
↳ if not → False

→ print('name' in mydict) → this is general/default one will always only check for keys.

→ print('30' in mydict) → will never check for values if u enter it will return false

→ print('30' in mydict.values()) → if u want to check for value hv to use .values()
True

The In-operator uses $O(n)$ algorithm for lists & dictionary

for lists → it uses linear search algo.

→ As the length of list ↑ the time - frame of linear search \rightarrow go also ↑ proportionally.

For Dictionary → it uses hashtable algo.

Efficient
→ the In-operator takes only $O(1)$ time always no matter the size of dictionary

for operator → can visit each key of a dictionary

↳ If key in mydict:
 print(key)
 for key in mydict:
 print(key, mydict[key])

Built-in python functions:

↳ When 1 of the parameters is False; return False

all()

→ will →

cases

return value

All values are True	True
All values are False	False
1 value is True(others False)	False
1 value is False(others True)	False
empty iterable	True

- ↳ mydict = {1: True, 2: False, 3: True}
- ↳ will give (True) when printed → print(all(mydict))
- ★ mydict2 = {1: False, 2: False, 3: False}
 - ↳ → → (False)
 - ★ mydict3 = {1: True, 2: False, 3: False}
 - ↳ → → (False)
 - ★ mydict4 = {1: False, 2: True, 3: True}
 - ↳ → → (False)
 - ★ mydict5 = {}
 - ↳ → → (True)

any ()

→ Returns True
if any 1 of the parameters is true
else False.

Cases	Return value
All values are true	True
All values are false	False
1 value is true (others false)	True
1 value is false (others true)	True
Empty iterable	false

- ④ mydict1 = {1: True, 2: True, 3: True}
 - ↳ print(any(mydict1)) → True
- ⑤ mydict2 = {1: False, 2: False, 3: False}
 - ↳ print(any(mydict2)) → False
- ⑥ mydict3 = {1: True, 2: False, 3: False}
 - ↳ print(any(mydict3)) → True
- ⑦ mydict4 = {1: False, 2: True, 3: True}
 - ↳ print(any(mydict4)) → True
- ⑧ mydict5 = {1: True, 2: True, 3: True}
 - ↳ print(any(mydict5)) → True

`len()`

→ returns the no. of key-value pair

`print(len(mydict))`

3

`sorted()`

→ Returns only the keys

3 arguments (depends on ur need)

→ `sorted(iterable, reverse, key)`

could be any collection will reverse

✓ List will sort the sorted list

✓ Tuple

✓ True

✓ False

✓ None

✓ Anythin else than True

↳

`print(sorted(mydict))`

↳

④ `print(sorted(mydict))`

↳

`print(sorted(mydict, reverse=True))`

↳

`print(sorted(mydict, reverse=True))`

↳

⑤ `print(sorted(mydict, key=len))`

↳

`mydict = {'Hello': 1, 'gao': 2, 'id': 3}`

↳

Dictionary

✓

3

187

Dictionary	List
Unordered	Ordered
Accessed via keys	Accessed via index
Collection of key-value pairs	Collection of elements
Preferred when u hv unique key values	Preferred when u hv ordered data
No duplicate members	Allows duplicate members

Time - space
Complexity of Dict

operation	Time - space	Space - space
Creating a Dictionary	$O(\text{len}(\text{dict}))$	$O(n)$
Inserting a value in a dictionary	$O(1)$	$O(n)$
Traversing a given dictionary	$O(n)$	$O(1)$
Accessing a given cell	$O(1)$	$O(1)$
Searching a given value	$O(n)$ <small>Worst case $\rightarrow O(n)$ if "n" open cells</small>	$O(1)$
Deleting a given value	$O(1)$	$O(1)$

dict $\text{dict} = \{\text{apple}: 1, \text{orange}: 2\}$

Tuple

★ A tuple is an immutable sequence of Python objects

★ Tuples are also comparable & hashable

t = ('a', 'b', 'c', 'd') ➔ 'a', 'b', 'c', 'd'

b

✓ (But they are ^{not} mutable.)

★ This even for a single element it should be ;

t = ('a',) ➔ Mandatory !!

If we write something like t = ('a')

it will consider it as a string enclosed in parentheses.

★ We can also create a tuple, like this :-

t = tuple('abcd')

print(t)

Result

FOR creating a tuple

TC → O(1)

SC → O(n)

Result

('a', 'b', 'c', 'd')

Tuples On Memory

- ↳ elements are located contiguously.
- ↳ they are **immutable** — since declared can't be changed.

Ex Sample-Tuple: `['a', 'b', 'c', 'd', 'e']`

'a'	'b'	'c'	'd'	'e'
-----	-----	-----	-----	-----

↳ assuming what at memory

Read
`>>> tuple_name = ('a', 'b', 'ab', 'ba')`

`>>> tuple_name[0]`

a

But
we do
the
same

`>>> tuple_name = 'a', 'b', 'ab', 'ba'`

`>>> tuple_name[0]`

a

00

Accessing elements on Tuple

\Rightarrow Indexing

1



★ $t = ('a', 'b', 'c', 'd', 'e')$

$t[-1] \rightarrow t[3]$

e

d

Some

Slicing 2

$\star t[0:2] \rightarrow t[:2]$

$('a', 'b')$

But \rightarrow the only diff of

$t[-1] = 'z'$

*

\Rightarrow error on tuple.

immutable

Traversing a tuple

$t = ('a', 'b', 'c', 'd')$

- For i in t :
- $\text{print}(i)$

④

- for i in range($\text{len}(t)$):
 $\text{print}(t[i])$

④ $rc \longrightarrow o(i)$

④ $sc \longrightarrow o(.)$

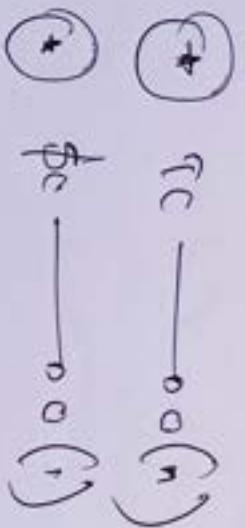
Searching an element

① using in - operator

```
t = ('a', 'b', 'c', 'd')  
print('b' in t)  
True  
print('f' in t)  
False
```

② Looping or Binary search

```
def searching_element(tuple1, element):  
    for r in tuple1:  
        if r == element:  
            print(tuple1.index(r))  
        else:  
            print("not present")
```



Tuple - Operations

tuple1 = (1, 2, 3, 4)
tuple2 = (5, 6, 7, 8)

[+] → Concatenation

```
print(tuple1 + tuple2)  
(1, 2, 3, 4, 5, 6, 7, 8)
```

[★] → Star operator

```
print(tuple1 * 3) → entire tuple would be printed 3 times.  
(1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4)
```

[n → operat]

Methods → methods like add, remove doesn't exist for tuple as they are immutable.

[• count()] → Repetition of elements.

```
t = (1, 2, 3, 4, 4)  
t.count(4))
```

2

[• index(x)] → will return the element of index

```
print(t.index(3))
```

2

Built-in functions

len() → t = (1, 2, 3, 4)

print(len(t))

4

max() → returns max number

print(max(t))

min() → returns minimum number

print(min(t))

④ type-casting

print(tuple([1, 2, 3, 4]))

(1, 2, 3, 4)

Tuple VS List

④ Immutable {
 | Notes - In:
 | + fast & In
 | + efficient

[But]

(we can reassing the entire
tuple, of list)

Ex:
 $t = (1, 2, 3, 4, 5)$

$t[0] = 6$ ✗

$t = (8, 9, 10)$ ✓

★ cannot delete a single
element in a tuple

But can delete the entire
tuple

Functions common for both tuple & list

Ex: len(), max(), min(), sum(), any(), all(), sort

Methods common for both tuple & list

Count(), Index()

Methods only for
List {
 + can't be used for tuple
 + append()
 + insert()
 + remove()
 + sort()

④ Tuples can be stored
in list
Ex: Lists can be stored
in tuple

★ Both tuple & list can be nested.

★ Tuples are used for
heterogeneous data types
(J/F)

④ Lists are used for
homogeneous (H/W) data

Time & space Complexity

Operation	Time-shape	Space-shape
Creating a table	$O(1)$	$O(n)$
Traversing a given table	$O(n)$	$O(1)$
Accessing a given element	$O(1)$	$O(1)$
finding a given element	$O(n)$	$O(1)$

Classification of time complexity based on algorithmic paradigm

- Divide and conquer paradigm
- Greedy paradigm
- Dynamic programming paradigm
- Backtracking paradigm
- Branch and bound paradigm

Greedy paradigm → Greedy algorithm

Divide and conquer paradigm → Divide and conquer algorithm

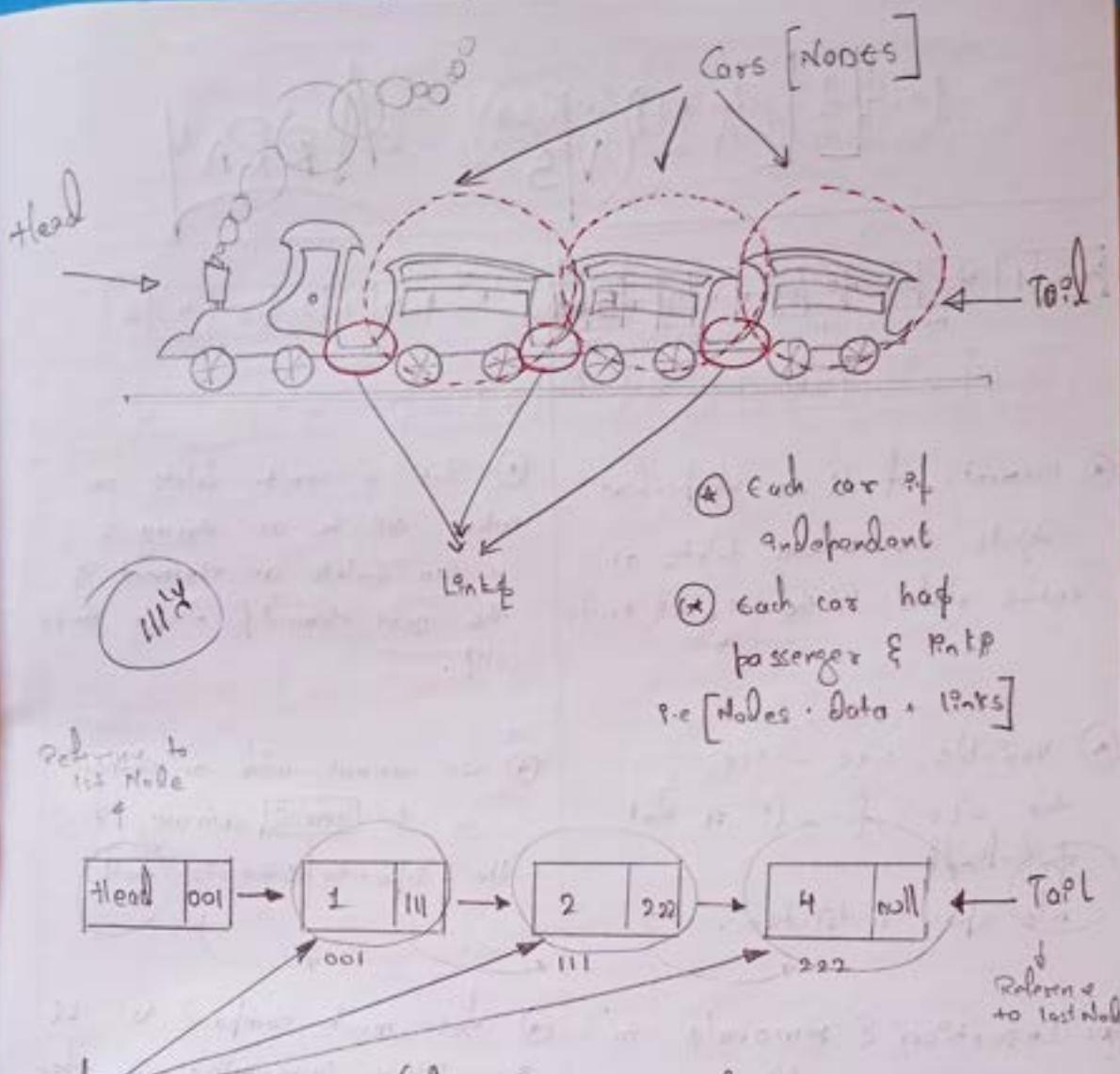
Dynamic programming paradigm → Dynamic programming algorithm

LINKED LIST

- ↳ It's a form of sequential collection of elements where order is not mandatory,
- A LL is made up of independent nodes that may contain any type of data
- & each node has a reference to the next node in the link.

[LL is almost like a tree]





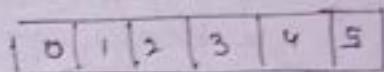
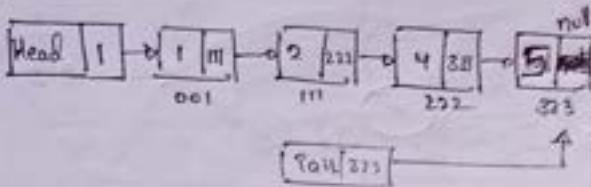
Nodes: data & links (references) to next node

why should u know head? → In order to access the elements u should know where it starts i.e. u should know head.

why Tail? → If u don't know the last nodes' address & when u want to add an element u hv to traverse right from the 1st, which makes $T.C = O(n)$. But if u know the address of tail $T.C = O(1)$ ⇒ Time complexity % ↓.

The only dif of the LL are not Contiguous.

LL VS ARRAY



- ★ Elements of LL are independent objects i.e. u can delete an entire Node & the LL still exists.
- ★ But u can't delete an entire cell in an array. u can delete an element & the next elements occupy those cells.
- ★ Variable size - size of all LL is not predefined. No space restriction.
- ★ we cannot add an element to the same array if the capacity/size is full.
- ★ Insertion & removals in LL are very efficient.
- ★ But if u want to access elements in LL as they are sequential to be to start from the head.
- ★ Not much compared to LL i.e. time consuming : when to add or delete an element the next elements hv to move 1 step (image for 1000 elements).
- ★ Array are indexed so accessing elements are easy TC → O(1).

Types of LL

Singly-LL

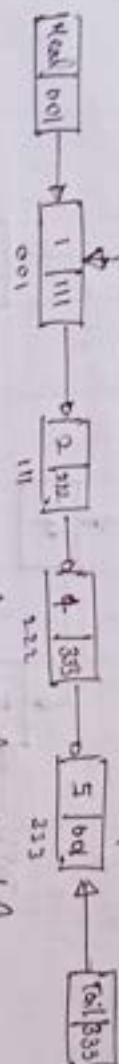
SL



- ① each Node holds a value & a reference to the next node in the list

Circular singly-LL

CSL

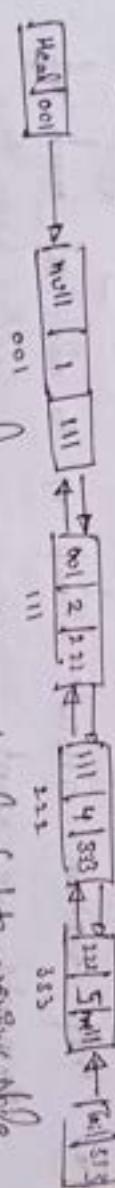


- ② the last node holds no reference of 1st Node.

(Ex) when u play **ludo** after the chance of player 1, 2, & 3 the next chance should be of player 1.
→ the player 3 holds no reference of player 1, 2, & 0

Doubly-LL

DL

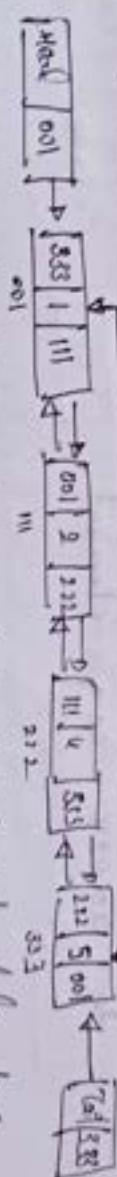


- ③ each Node has 2 reference — 1 to next node & 1 to previous Node

(Ex) → it is a multi-player game ← → we can move both backwards & forward

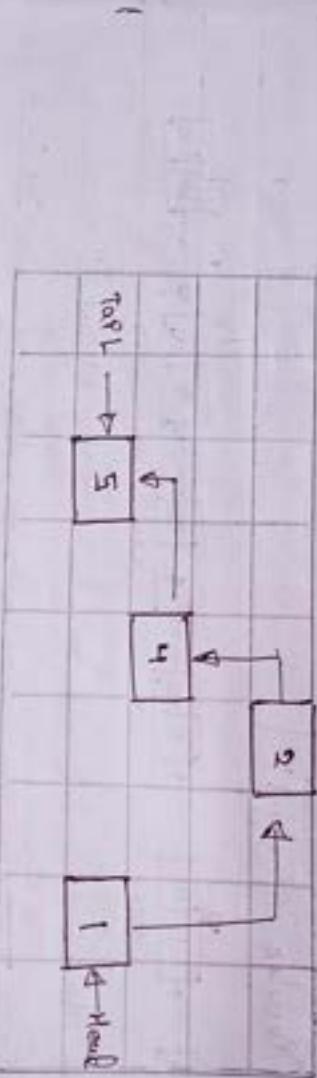
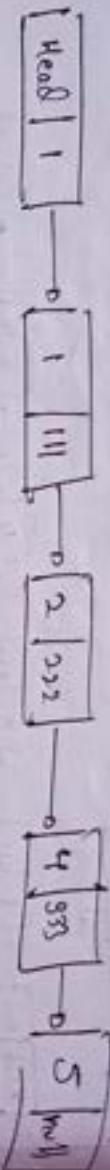
Circular Doubly-LL

CDL
(i) and right side in place



- ④ the only diff b/w DLL & CDLL is in b/w 1st & last Node b/w the last Node & has the reference of previous & 1st node & 1st Node has reference of next node & last Node

L.L in memory



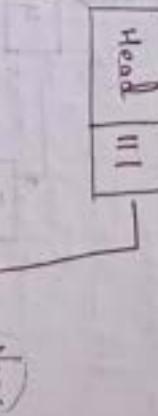
[Memory]

- (*) Elements are not located contiguously
- (*) That random allocate allow up to add as much as elements we want
- (*) At the price of LL is not pre-defined
- (*) But due to that Random allocate we cannot access the element using `int` index
so we have to traverse right from the list re the head
to access any element.

Creation of Singly Linked List

Create head & tail.

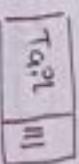
Initialize with null



Create a blank Node &
assign a value to it &
reference to null



Link head & tail with
shape Node



```

class SLL:
    def __init__(self):
        self.head = None
        self.tail = None

    class Node:
        def __init__(self, value=None):
            self.value = value
            self.next = next = None

```

```

Singlylinkedlist = SLL()
node1 = Node(1)
node2 = Node(2)

singlelinkedlist.head = node1
singlelinkedlist.tail = node2
singlelinkedlist.tail = node2

```

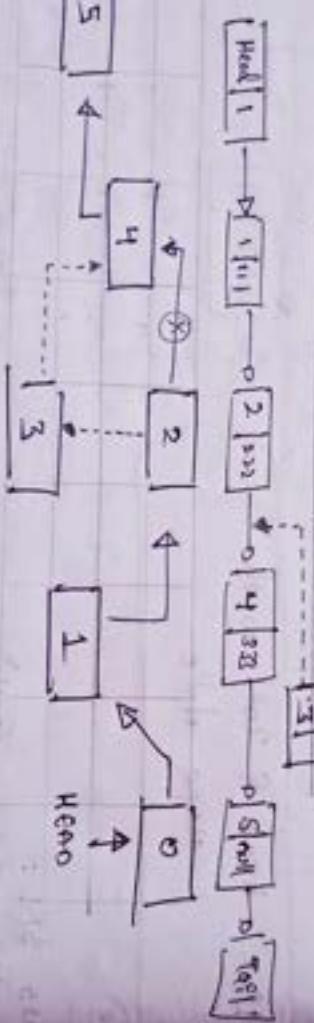
Insertion of to SLL in memory

Insert a new node @ the Beginning

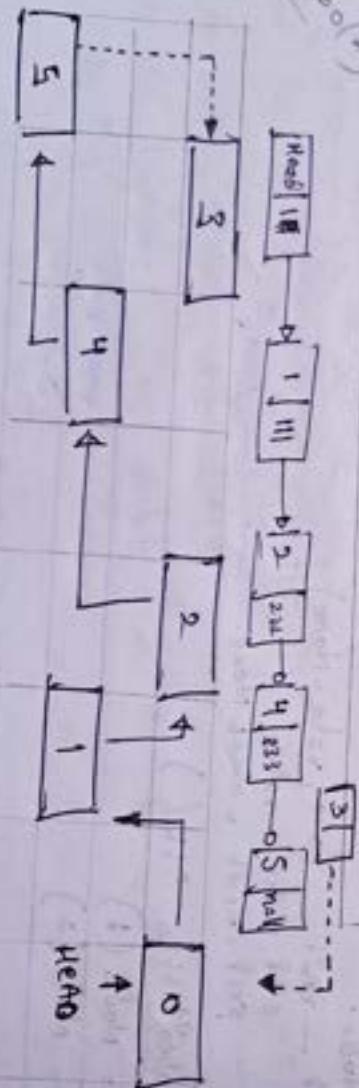


A new node stored in the heap memory by new is the rank $\text{new} \in \boxed{1}$ is broken & $\text{new} \in \boxed{0}$ is $\text{new} \in \boxed{2}$ of $\text{new} \in \boxed{3}$

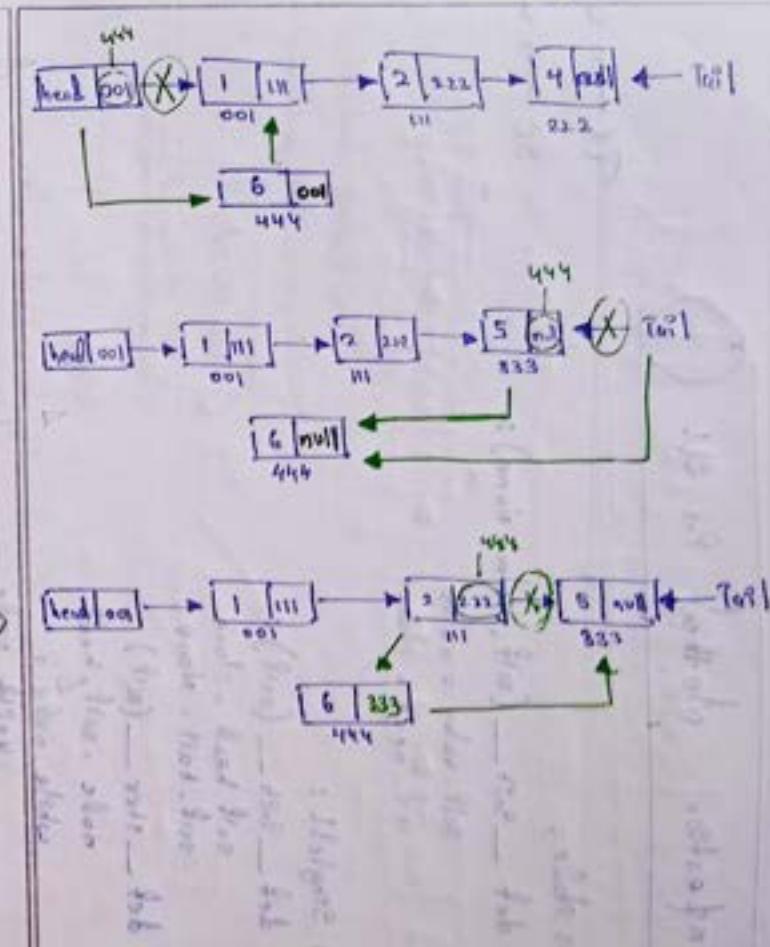
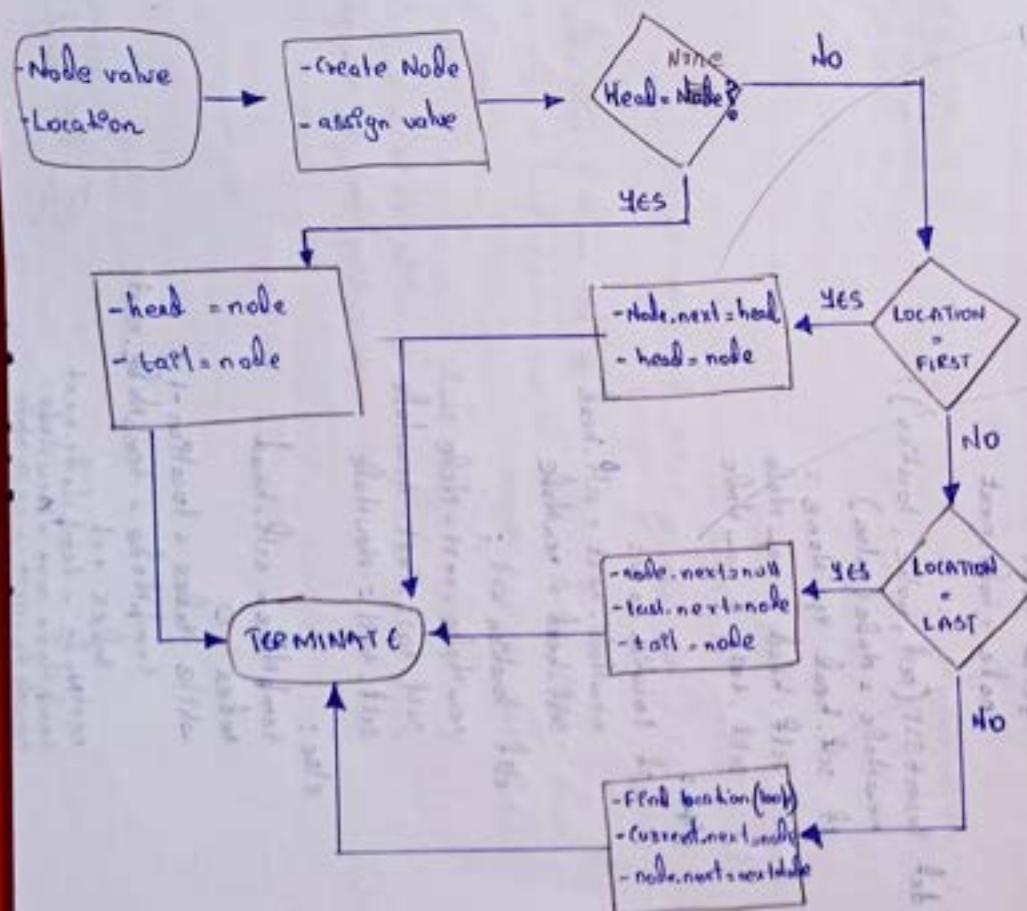
Insert a new node after a node



Insert a new node @ the end of LL



Insertion in SLL - ALGORITHM (All 3 types in 1)



○ Insertion Method in SLL

TC $\rightarrow O(n)$
 SC $\rightarrow O(1)$

Class Node:

```
def __init__(self, value=None):
    self.value = value
    self.next = None
```

Class Singlll:

```
def __init__(self):
    self.head = None
    self.tail = None
```

```
def __iter__(self):
    node = self.head
    while node:
        yield node
        node = node.next
```

```
# testing on SLL
def insertSLL(self, value, location):
    newNode = Node(value)
    if self.head == None:
        self.head = newNode
        self.tail = newNode
    else:
```

```
        if location == 0:
            newNode.next = self.head # set newNode's next reference
            self.head = newNode # set variable head to reference the new node
        elif location == 1:
            newNode.next = None
            self.tail.next = newNode # making old tail node point to new node
            self.tail = newNode # setting tail to new node
```

```
    else:
        tempNode = self.head
        index = 0
        while index < location - 1:
            tempNode = tempNode.next
            index += 1
        nextNode = tempNode.next
        tempNode.next = newNode
        newNode.next = nextNode
```

Sorry
for a little
change
in
color

newNode

current

newNode.next
p.e. tempNode

Next how it works now

Output:

singlyLinkedList = singlyLL()

singlyLinkedList.insertSLL(1, 1)

singlyLinkedList.insertSLL(2, 1)

singlyLinkedList.insertSLL(3, 1)

singlyLinkedList.insertSLL(0, 4)

singlyLinkedList.insertSLL(9, 0)

print([node.value for node in singlyLinkedList])

[0, 1, 2, 3, 0]

(0, 0, 0, 0)

(0, 0) HC cannot

work as last. If we ++

("this has not yet been popped off") then

if node
else

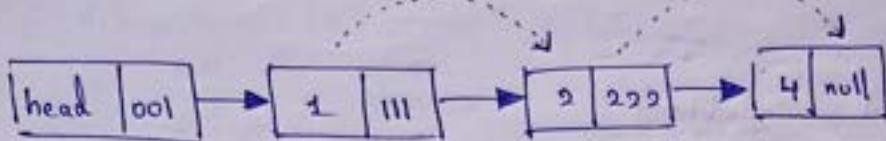
break else + else

work + work + work

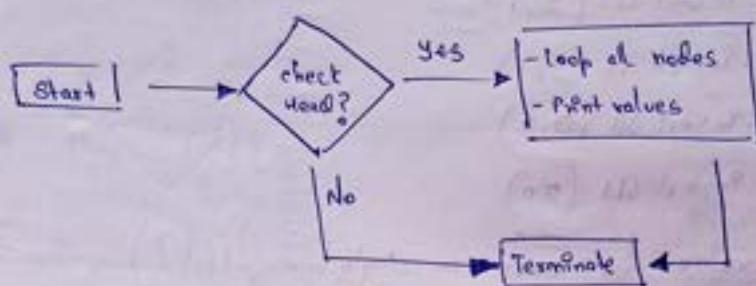
(value,value) thing

true, else = else

Traversing in SLL



Algorithm



Code

#Continuation of previous code (of class SinglyLL)

```

def traverseSLL(self):
    if self.head is None:
        print("The singly linked list does not exist")
    else:
        node = self.head
        while node is not None:
            print(node.value)
            node = node.next
  
```

TC $\rightarrow O(n)$

SC $\rightarrow O(1)$

[0, 1, 2, 3, 0]

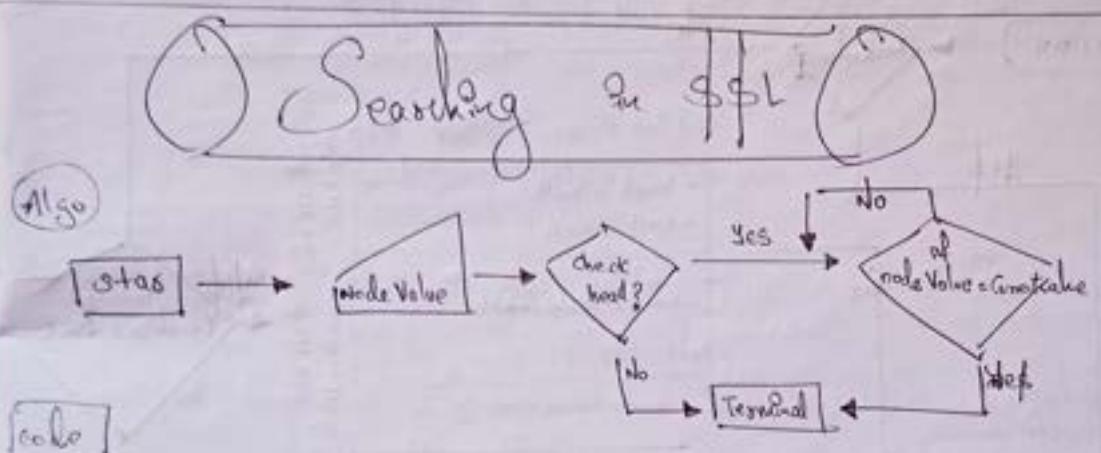
3

The value doesn't exist in the list

// Alter for the same code values.

```
print ([node.value for node in singlylinkedlist])  
singlylinkedlist. traversalSLL() # assuming def traversal() is  
[0, 1, 2, 3, 0] a method of singlylinkedlist()
```

0
1
2
3
0



```
def searchSLL(self, nodeValue): # continuation of previous code  
    if self.head == None:  
        print ("The list does not exist")  
    else:  
        node = self.head  
        while node != None:  
            if node.value == nodeValue:  
                return node.value  
            node = node.next  
        return "The value doesn't exist in this list"
```

TC - O(n)
SC - O(1)

```
print ([node.value for node in singlylinkedlist])  
singlylinkedlist. searchSLL(3)  
singlylinkedlist. searchSLL(66)
```

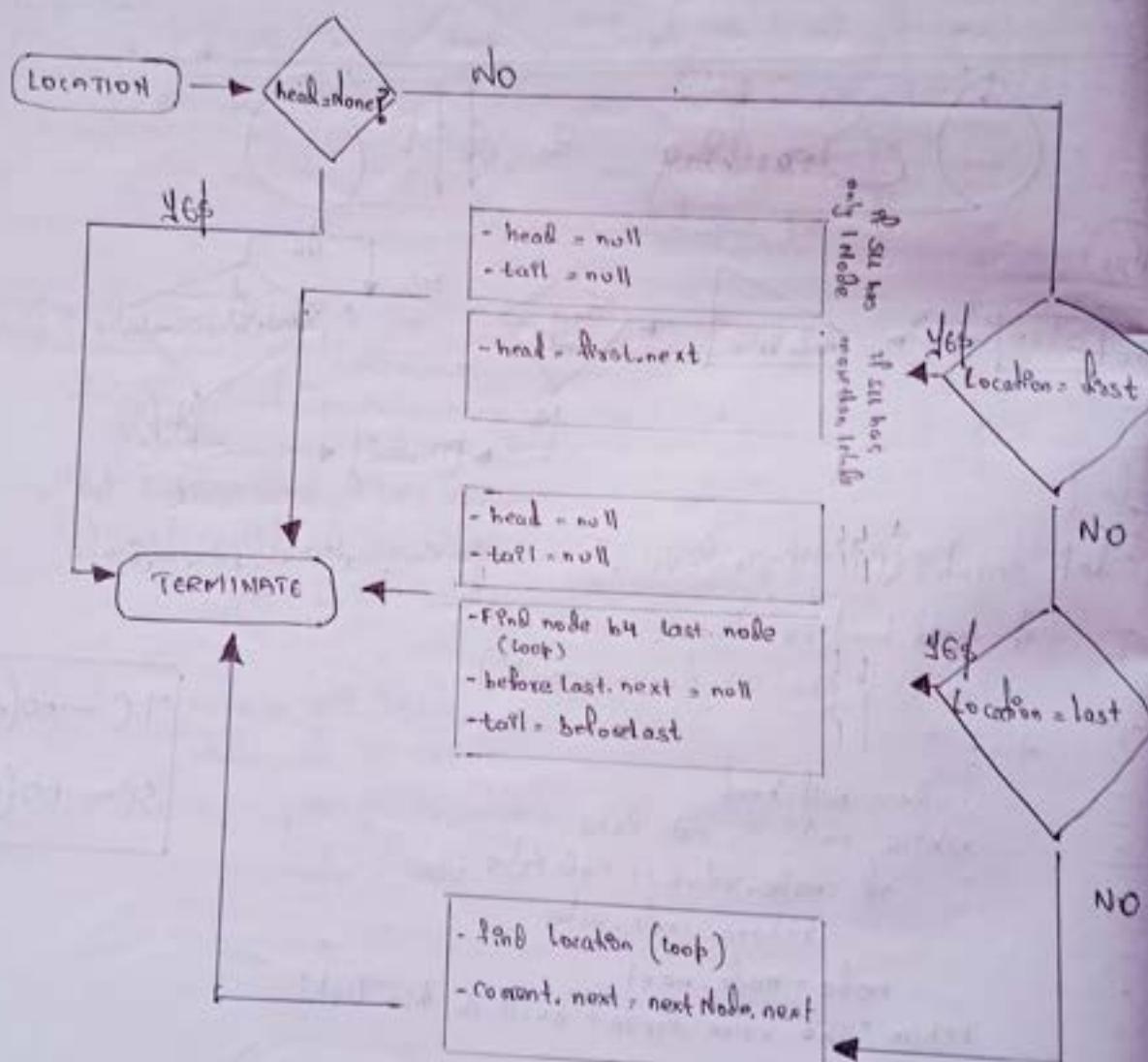
Deletion in SLL

③ cases → - Deleting the 1st node

- Deleting the any given node

- Deleting the last node

[Algorithm]



```
def deleteNode (self, location):  
    if self.head == None:  
        print ("The list doesn't exist")
```

$$\begin{array}{l} \textcircled{\ast} \text{ TC} \rightarrow O(n) \\ \textcircled{\ast} \text{ SC} \rightarrow O(1) \end{array}$$

else:

if location == 0:

```
    if self.head == self.tail:  
        self.head = null None  
        self.tail = None
```

else:

```
    self.head = self.head.next # Reassigning head to node after  
                            # the 1st node
```

if location == 1:

```
    if self.head == self.tail:  
        self.head = None  
        self.tail = None
```

else:

node = self.head

while node is not None:

if node.next == self.tail: # we're reaching the last
 # node of the list
 break

node = node.next

node.next = None # we set the reference of node b4 last

self.tail = node # & tail value to last node (the new 1)

else:

tempNode = self.head

index = 0

while index < location - 1:

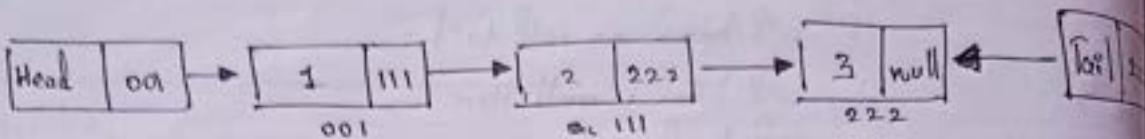
tempNode = tempNode.next

index += 1

nextNode = tempNode.next

tempNode.next = nextNode.next

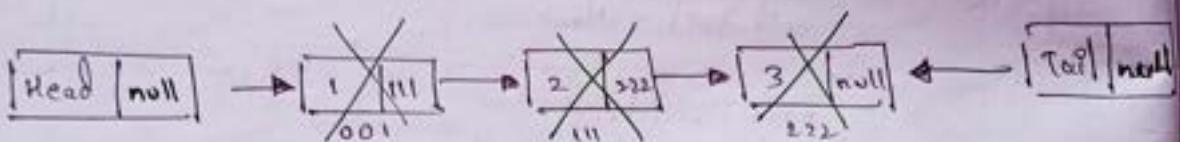
Allow Delete on entire \$LL



if u set the reference of head & tail to null

\Rightarrow all that there are no nodes in SLL

that's how you delete an entire SLL



★ when u set the ref head.next & tail.next to none
then the G.C traversing from the start

★ checks the ; there's nothing referencing the 1st node
 \Rightarrow it deletes the 1st node.

■ it checks that there's ————— 2nd node
 \Rightarrow it ————— 2nd node

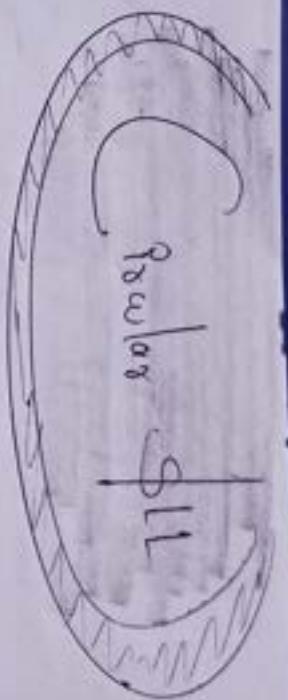
■ ————— . 3rd node
 \Rightarrow it ————— 3rd node

```
def delete_from_ll(self):
    if self.head is None:
        print("The LL doesn't exist")
    else:
        self.head = None
        self.tail = None
```

$$\textcircled{\star} \quad TC \rightarrow O(1) \quad \textcircled{\star} \quad SC \rightarrow O(1)$$

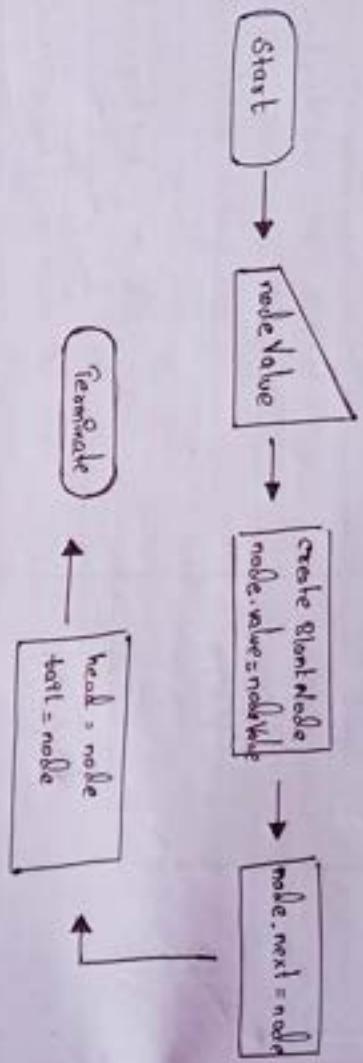
TIME & SPACE Complexity

	TIME-STIME	SPACE-STIME
Creation of SLL	$O(1)$	$O(1)$
Insertion in SLL	$O(n)$	$O(1)$
Searching in SLL	$O(n)$	$O(1)$
Traversing in SLL	$O(n)$	$O(1)$
Deletion of a node in SLL	$O(n)$	$O(1)$
Deletion of linked list (entire)	$O(1)$	$O(1)$

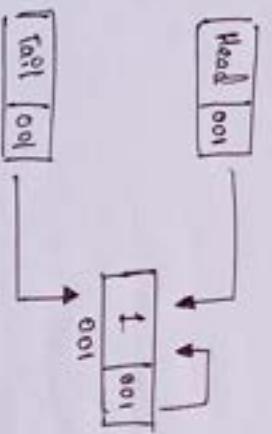


Creation of C.SLL

[Algorithm]



(A C.SLL is like with a single node)



TC → O(1)

SC → O(1)

Class Node:
def __init__(self, value=None):
 self.value = value
 self.next = None

Class CSLL:
def __init__(self):
 self.head = None
 self.tail = None
def __iter__(self):
 node = self.head
 while node:
 yield node
 if node.next == self.head: node = self.tail.next
 break
 node = node.next



creation of CSLL

def createCSLL(self, nodevalue):
 node = Node(nodevalue)
 node.next = node
 self.head = node
 self.tail = node

return "The CSLL has been created"

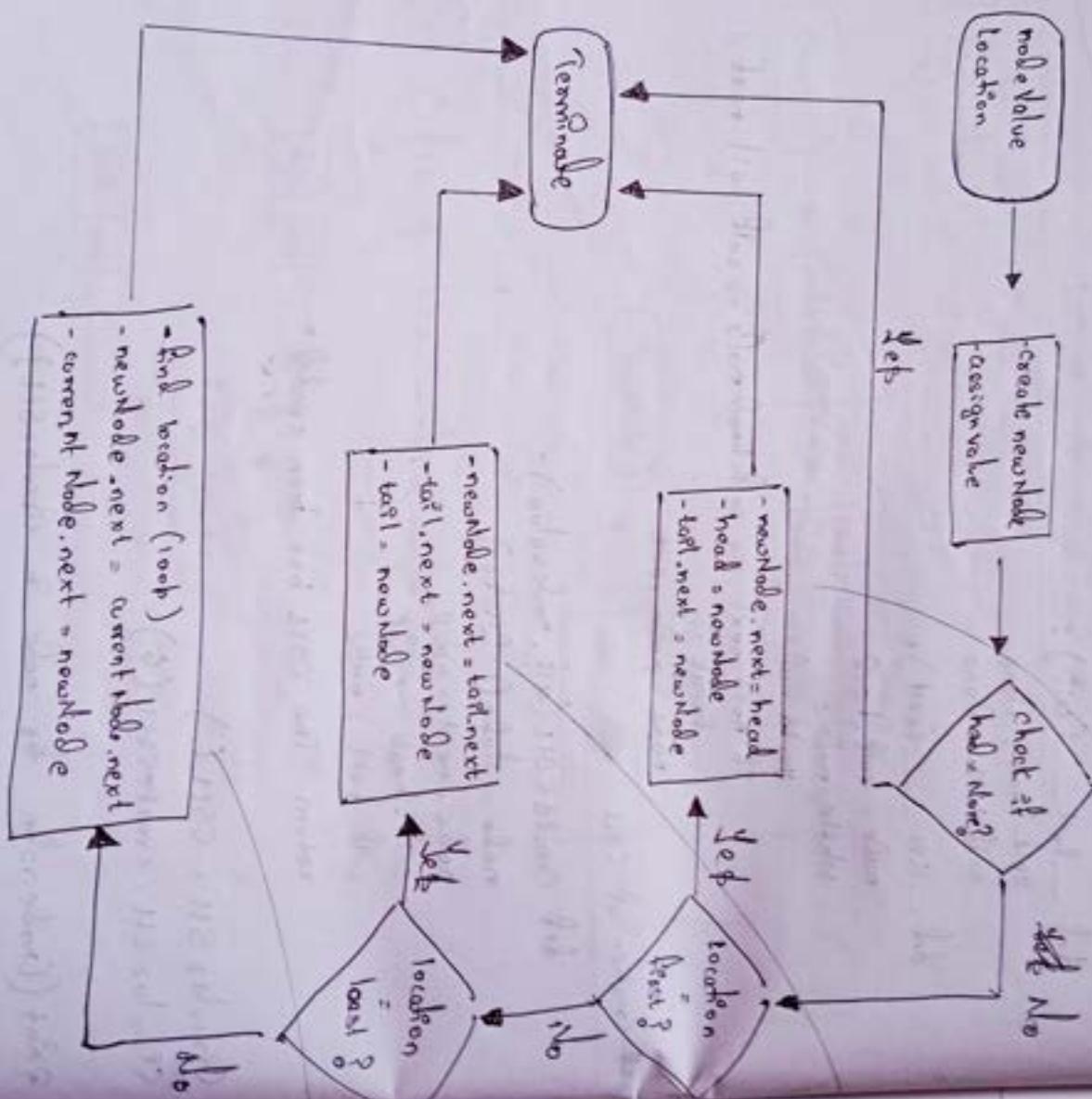
CircularSLL = CSLL()
CircularSLL.createCSLL(6)

print([node.value for node in CircularSLL])

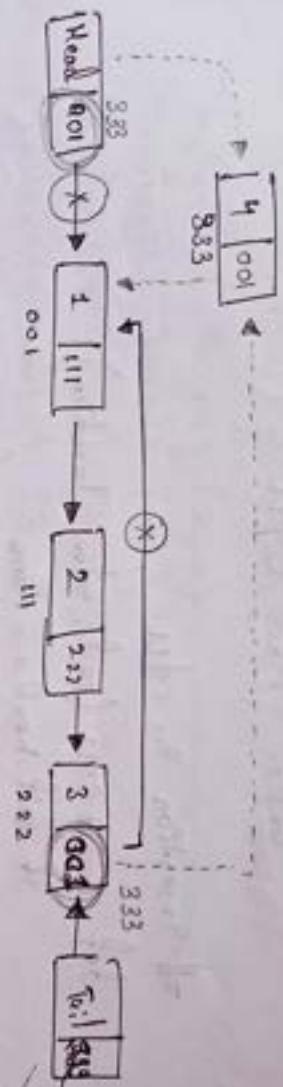
[6]

Insertion in C.que

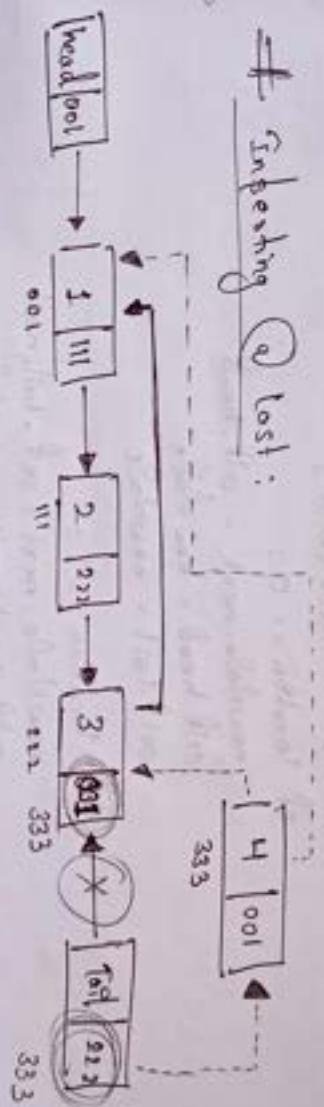
[Algorithm]



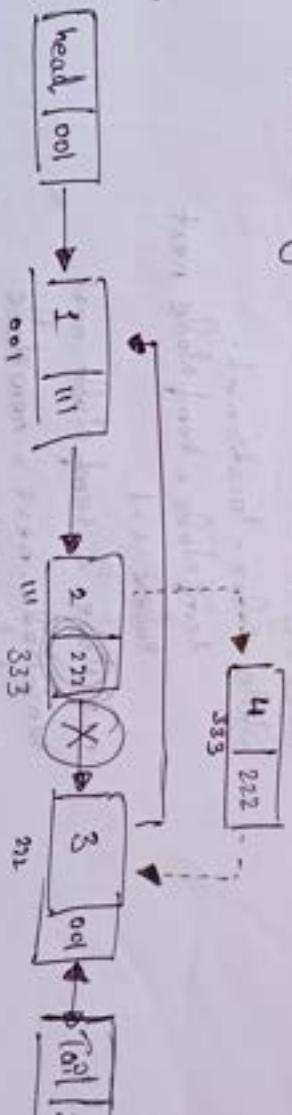
Inserting @ Beginning:



Inserting @ Last:



Inserting @ Specific location:



"inserted node will be inserted before the node with value 333"

Continuing on the from previous code
and class CList

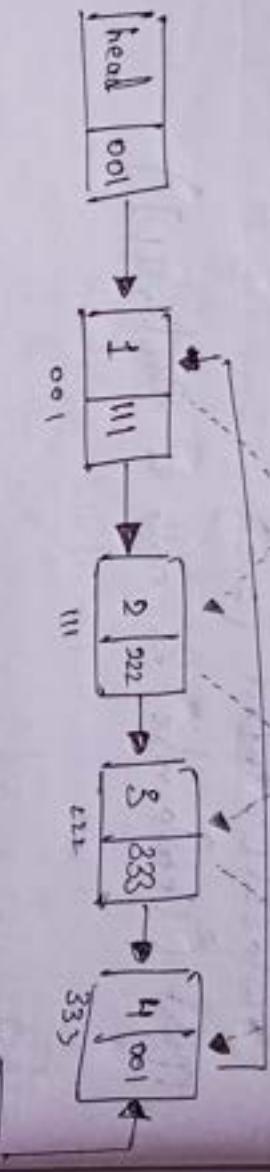
```
# Insertion in cll
def insertCll (self, value, location):
    if self.head == None:
        return "The head reference is None"
    else:
        newNode = Node (value)
        if location == 0:
            newNode.next = self.head
            self.head = newNode
            self.tail = newNode
        elif location == 1:
            newNode.next = self.tail.next
            self.tail.next = newNode
            self.tail = newNode
        else:
            tempNode = self.head
            index = 0
            while index < location - 1:
                tempNode = tempNode.next
                index += 1
            nextNode = tempNode.next
            tempNode.next = newNode
            newNode.next = nextNode
            new_node.next = nextNode
    return "The node has been successfully inserted"
```

```
CircularSLI = CircularSLI::CSLI()
Print(CircularSLI.CreateCSLI(1),
CircularSLI.Insert(0,0)
CircularSLI.Insert(2,1)
CircularSLI.Insert(3,1)
CircularSLI.Insert(2,0)
Print(CircularSLI.nodeValueForNodeInCircularSLI))
```

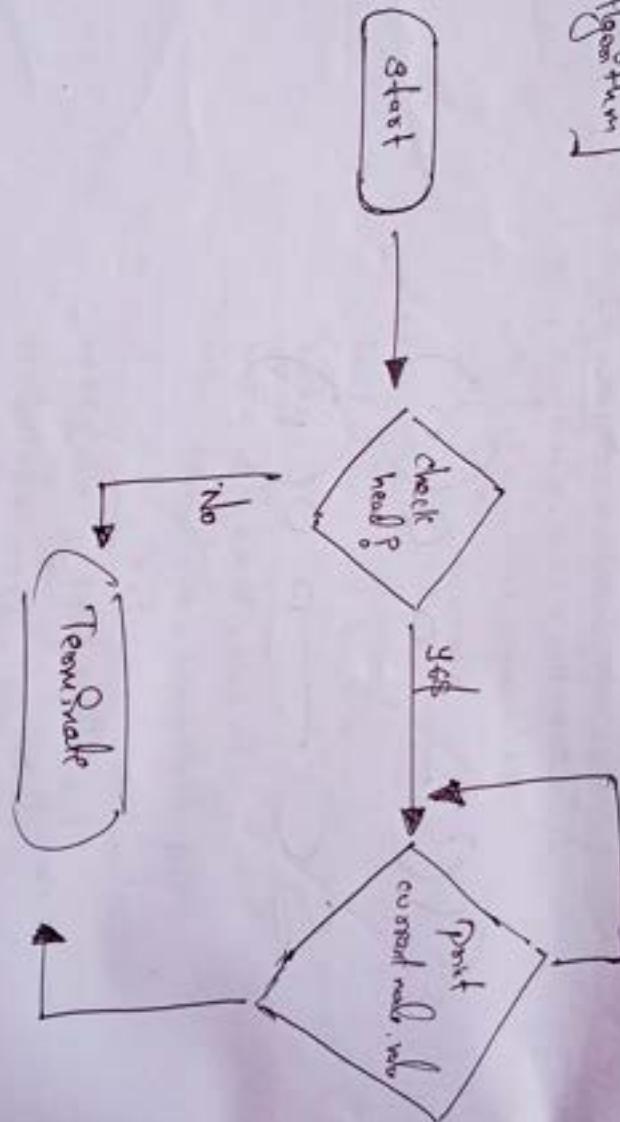
```
[0, 1, 2, 1, 3]
```

```
TC → O(1)
SC → O(1)
```

Traversal in C.S.L



[Algorithm]



```
def traversal_CSLL(self):  
    if self.head == None:  
        return "There is not any element to handle"
```

else:

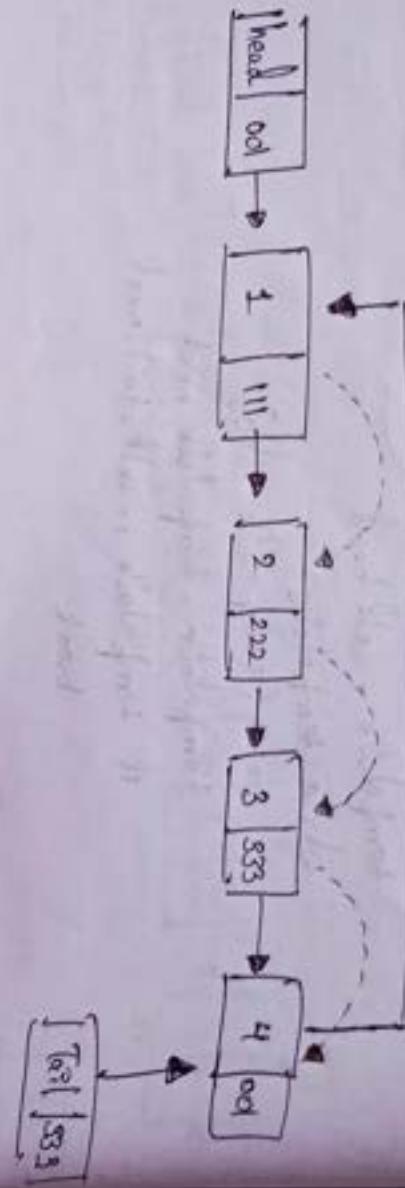
```
tempNode = self.head  
while tempNode:  
    print(tempNode.value)  
    tempNode = tempNode.next  
if tempNode == self.head.next:  
    break
```

```
CircularSLL.traversal_CSLL()
```

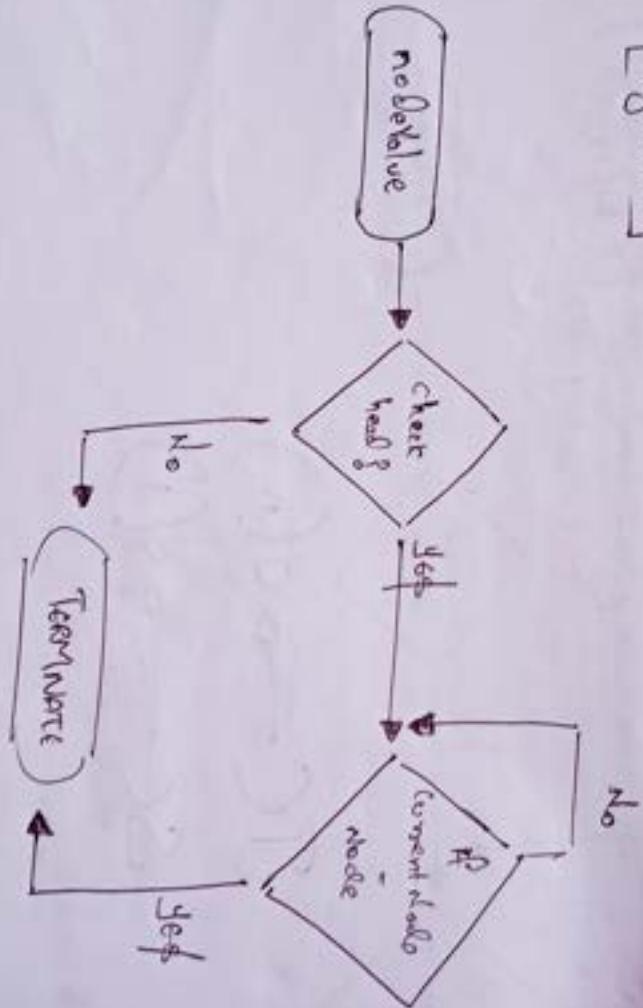
=> Also visit the same
previous node again

TC - O(n)
SC - O(1)

Descending On Cpu



[Algorithm]



```
def searchSU (self, nodeValue):
    if self.head == None:
        return "There is not any node in the list"
    else:
```

```
        tempNode = self.head
        while tempNode:
            if tempNode.value == nodeValue:
                return tempNode.value
            tempNode = tempNode.next
        tempNode = self.tail.next
        if tempNode == self.tail:
            return "This node does not exist in the list"
```

```
print (linkedlist.searchSU(4))
```

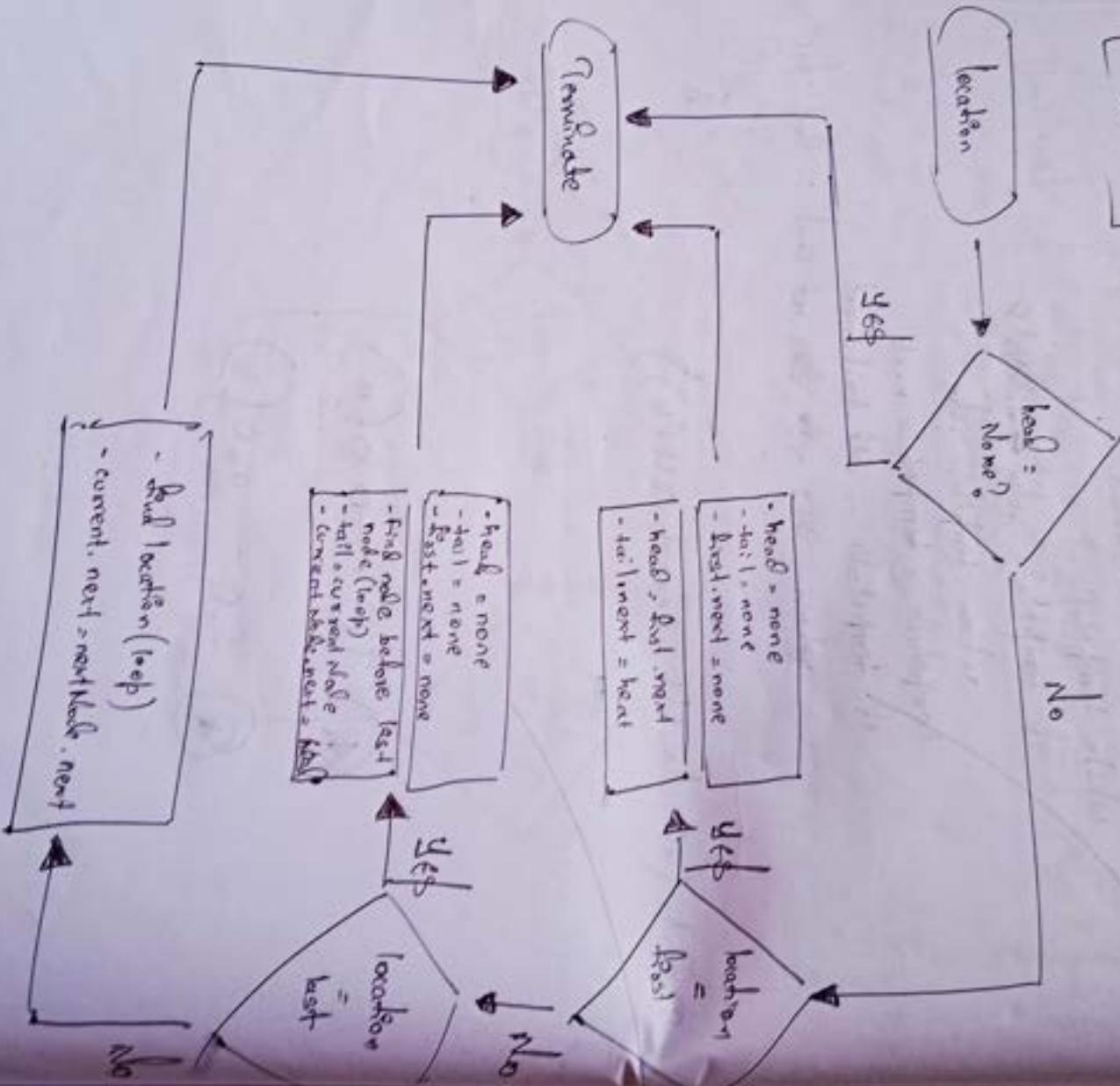
"The node" does not exist in the list

(R) TC → O(n)

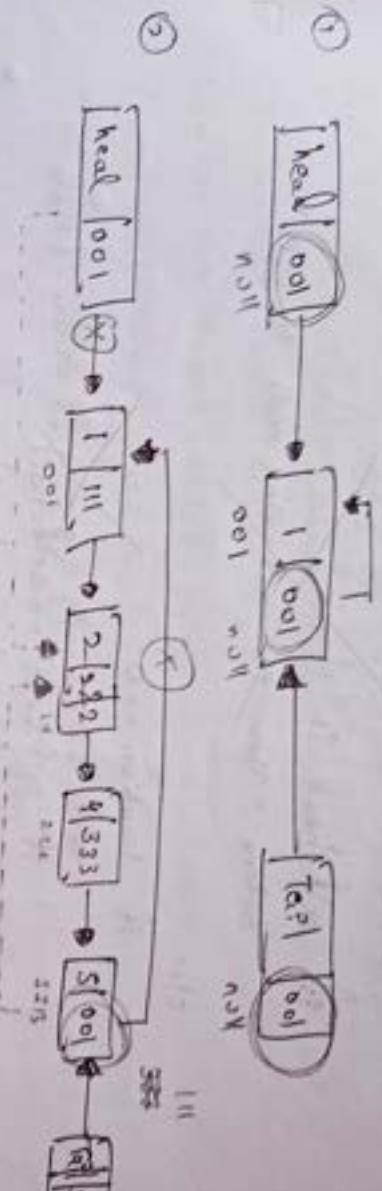
(R) SC → O(1)

○ Deletion in Cpu

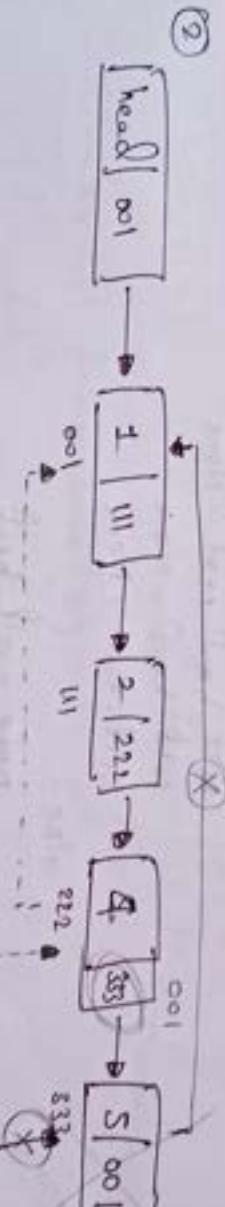
[Algorithm]



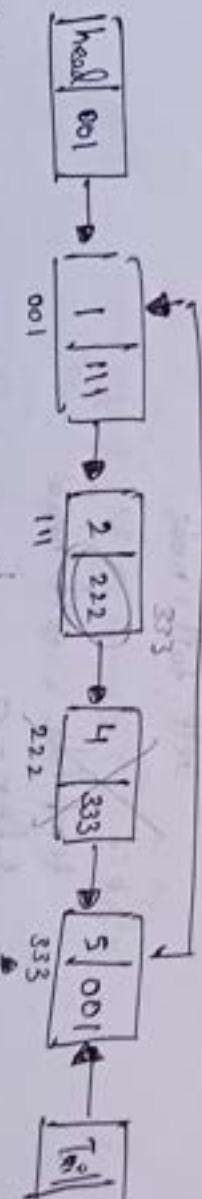
Deletion last Node.



Deleting last Node.



@ any given Node :



```
def deleteNode (self, location)
```

```
if self.head == None:
```

```
return "There is not any node in CLL"
```

```
else :
```

```
if location == 0:
```

```
self.
```

```
temp
```

```
= self.head == self.tail: # If 1 node
```

```
self.head.next = None
```

```
self.tail = None
```

```
self.head = None
```

```
else :
```

```
self.head.next = self.head.next
```

```
self.tail.next = self.head
```

```
elif location == 1:
```

```
if self.head == self.tail:
```

```
self.head.next = None
```

```
self.head = None
```

```
self.tail = None
```

```
else :
```

```
node = self.head
```

```
while node != None:
```

```
if node.next == self.tail:
```

```
break
```

```
node = node.next
```

```
node.next = self.head
```

```
self.tail = node
```

```
else :
```

```
tempNode = self.head
```

```
index = 0
```

```
while index < location - 1:
```

```
tempNode = tempNode.next
```

```
index += 1
```

```
nextNode = tempNode.next
```

```
(tempNode.next = nextNode.next
```

CircularSLL = CircularlyLinkedList()

CircularSLL.createEmpty(0)

CircularSLL.insertCSLL(0, 0)

o also insert

CircularSLL.insertCSLL(2, 1)

CircularSLL.insertCSLL(3, 0)

CircularSLL.insertCSLL(4, 1)

Print [node.value for node in CircularSLL]

CircularNodeSLL.deleteNode(2)

Print [node.value for node in CircularSLL]

[0, 0, 1, 3, 4]

[0, 0, 1, 3, 4]

CircularSLL.deleteNode(0)

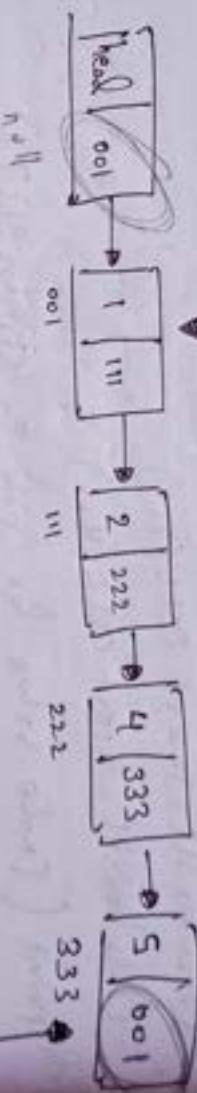
[0, 0, 1, 3, 4]

[0, 1, 2, 3, 4]

④ TC → O(n)

④ SC → O(1)

Deleting entire CSLL



$n = 1$

```
def deleteEntireCSLL(self):
```

```
    self.head = None  
    self.tail = None  
    self.tail.next = None
```

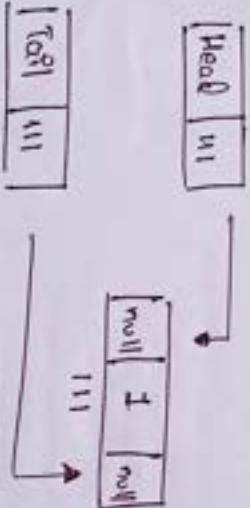
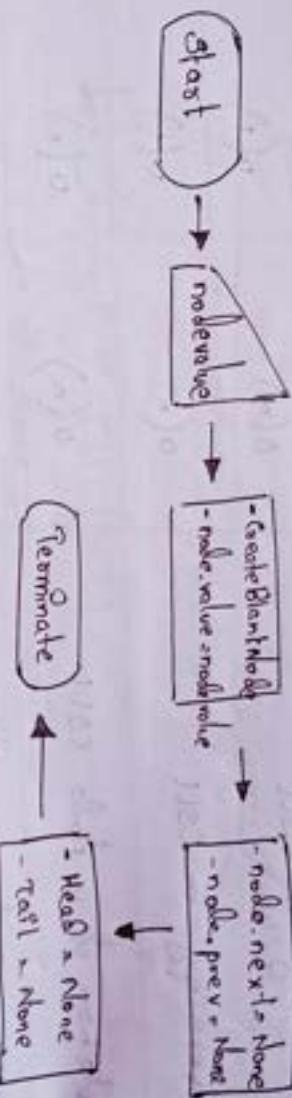
* TC → O(1)
* SC → O(1)

CFQ.C

	Time - space	Space - space
Creation of CSLL	$O(1)$	$O(1)$
Insertion in CSLL	$O(n)$	$O(1)$
Searching in CSLL	$O(n)$	$O(1)$
Traversing in CSLL	$O(n)$	$O(1)$
Deletion of a Node CSLL	$O(n)$	$O(1)$
Deletion of entire CSLL	$O(1)$	$O(1)$

Doubly LL

Creation of DLL

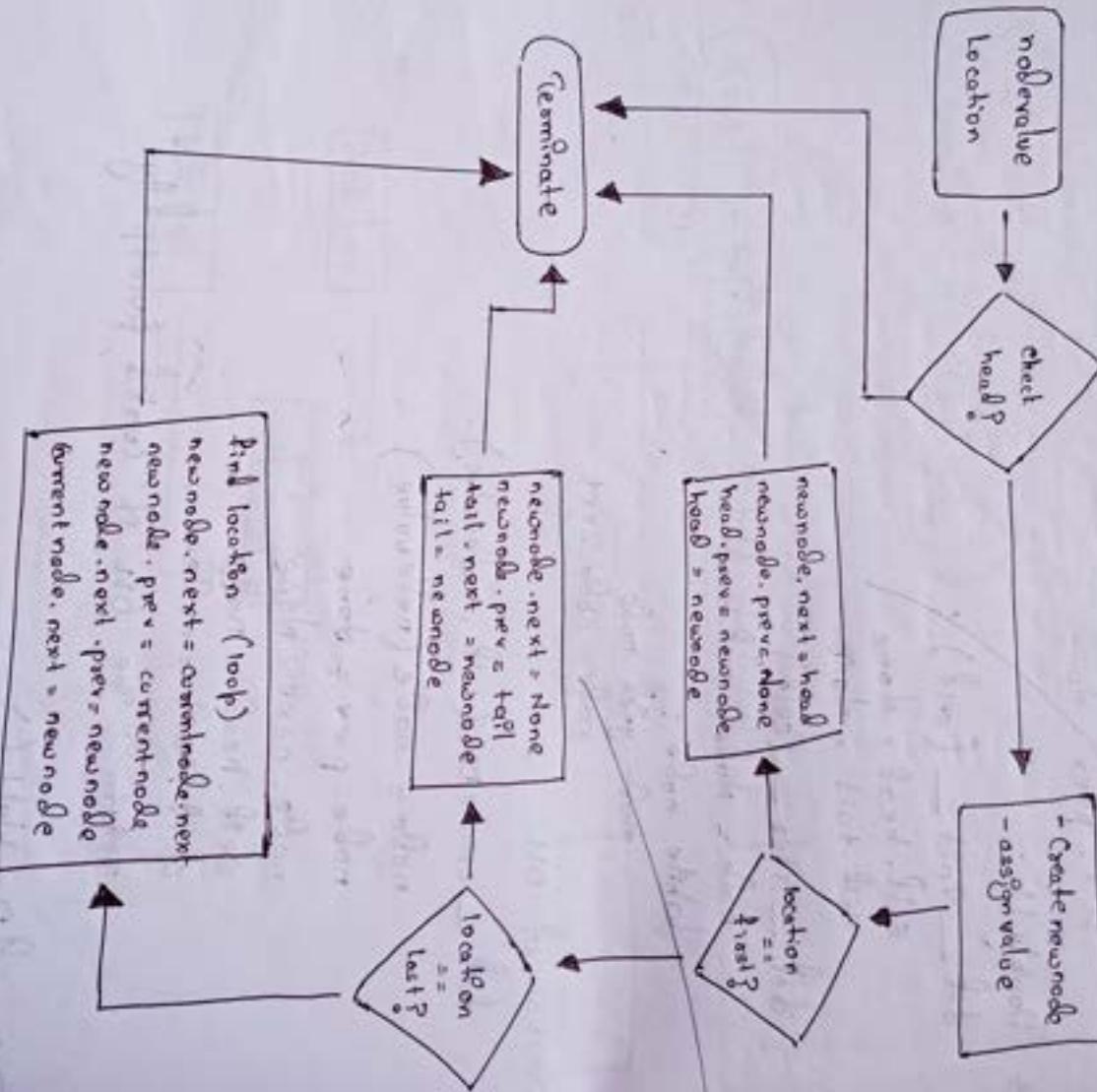


X

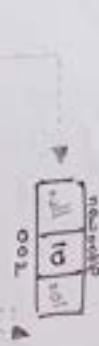
TC → O(1)
SC → O(1)

```
class node:  
    def __init__(self, value = None):  
        self.head = None  
        self.value = value  
        self.next = None  
        self.prev = None  
  
class DoublyLL:  
    def __init__(self):  
        self.head = None  
        self.tail = None  
  
    def __str__(self):  
        node = self.head  
        while node:  
            yield node  
            node = node.next  
  
    # creation of DLL  
    def createDLL(self, nodevalue):  
        node = node(nodevalue)  
        node.prev = None  
        node.next = None  
        self.head = node  
        self.tail = node  
  
    return "The DLL is created successfully."  
  
# DLL = DoublyLL()  
# DLL.createDLL(5)  
# DLL  
# print([node.value for node in DoublyLL])
```

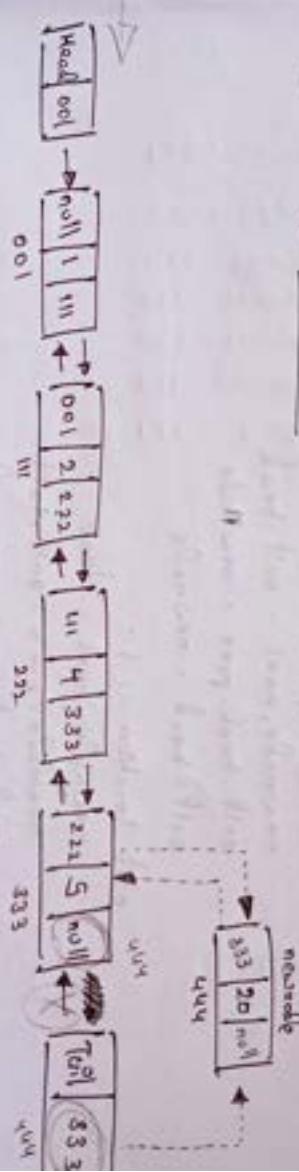
Insertion In DLL



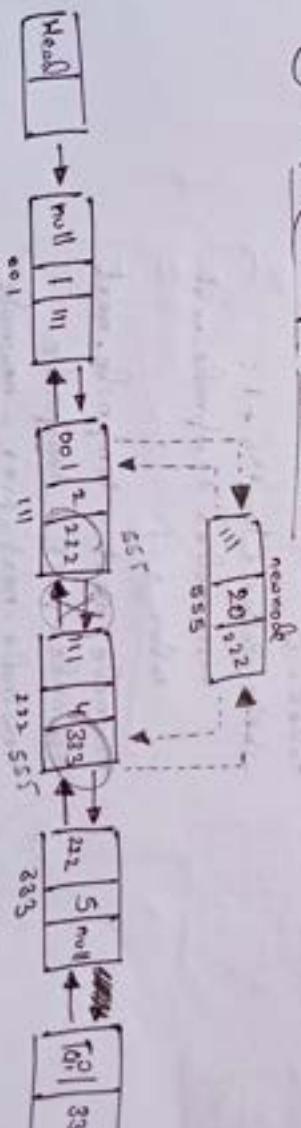
① \neq ② the beginning:



② \neq ② the end:



③ \neq ② specific position:



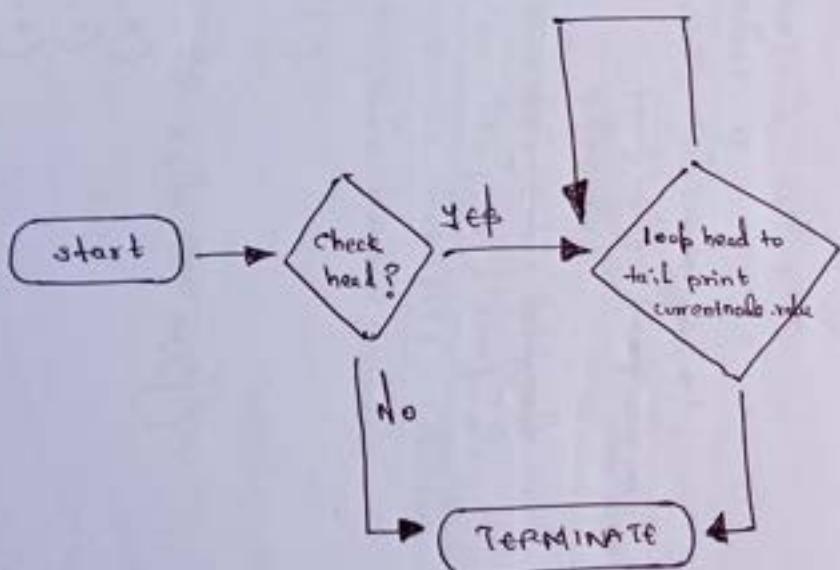
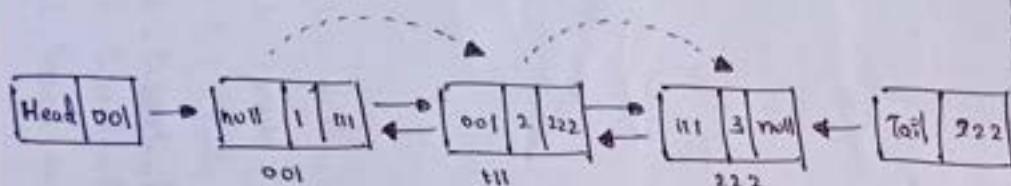
Insertion Method in DLL:

```
def insertnode(self, nodename, location):
    if self.head == None:
        print("The node cannot be inserted")
    else:
        newnode = node(nodename)
        if location == 0:
            newnode.prev = None
            newnode.next = self.head
            self.head.prev = newnode
            self.head = newnode
        elif location == 1:
            newnode.next = None
            newnode.prev = self.tail
            self.tail.next = newnode
            self.tail = newnode
        else:
            tempnode = self.head
            index = 0
            while index < location - 1:
                tempnode = tempnode.next
                index += 1
            newnode.next = tempnode.next
            newnode.prev = tempnode
            newnode.next.prev = newnode
            tempnode.next = newnode
```

```
DL = DoublyLL()
DL.create(5)
print([node.value for node in DL])
DL.insertnode(0, 0)
DL.insertnode(2, 1)
DL.insertnode(6, 2)
```

[5]
[0, 5, 6, 8]

Traversal - DLL



Traversal method in DLL :

```

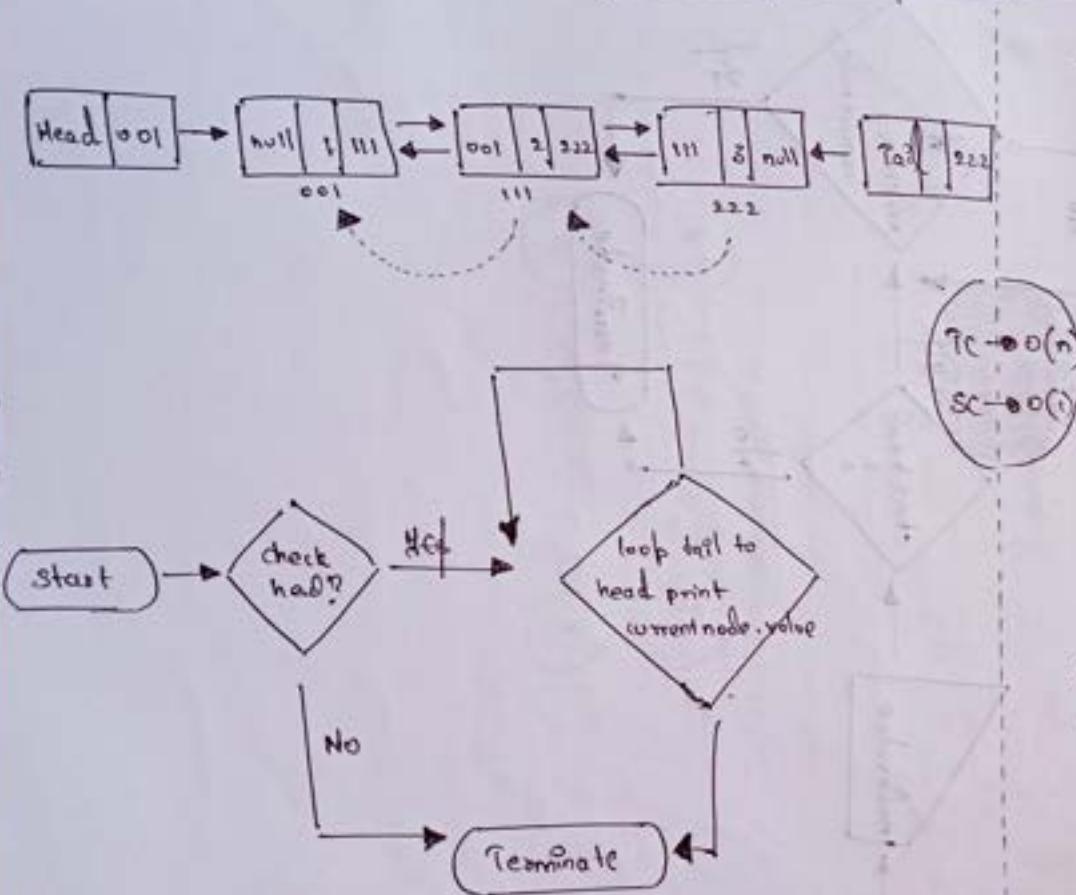
def traversalDLL(self):
    if self.head is None:
        return "There is no element to traverse"
    else:
        tempnode = self.head
        while tempnode:
            tempnode = tempnode.next
            print(tempnode.value)
            tempnode = tempnode.next
  
```

```

d = DLL = DoublyLL()
DLL.createDLL(5)
DLL.insertNode(0,0)
DLL.insertNode(2,1)
DLL.insertNode(6,2)
print([node.value for node in DLL])
DLL.traversalDLL()
  
```

[0, 5, 6, 2]

Reversible Traversal - DLL



Reverse Traversal method in DLL :

```

def reverseTraversal DLL(self):
    if self.head == None:
        return "There is no element to traverse"
    else:
        tempnode = self.tail
        while tempnode:
            tempnode
            print(tempnode.value)
            tempnode = tempnode.prev
  
```

DLL = DoublyLL()

DLL.create DLL(5)

DLL.insertNode(0,0)

DLL.insertNode(1,1)

DLL.insertNode(6,2)

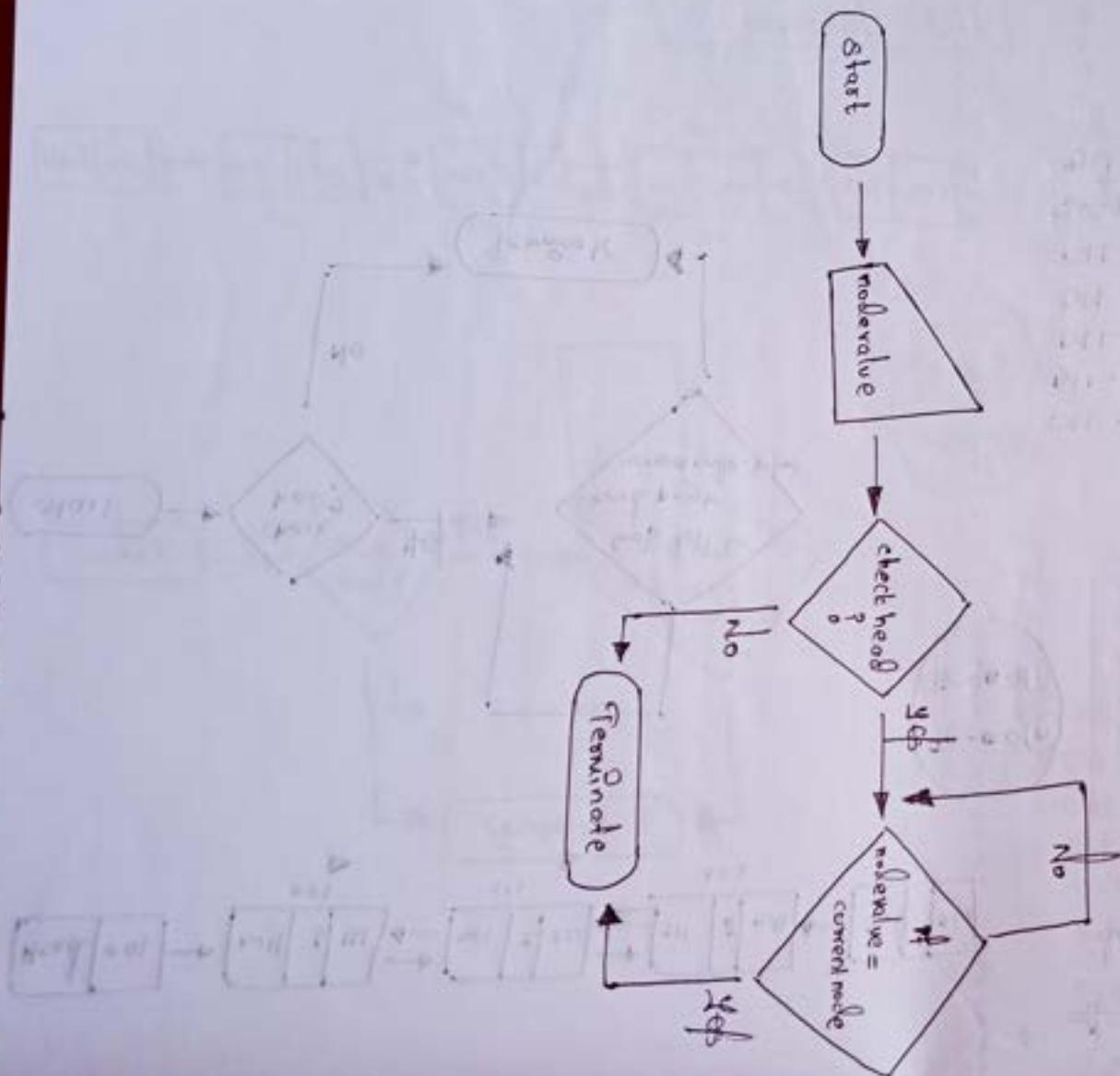
print ([node.value for node in DLL])

DLL.reverseTraversal DLL()

[0, 5, 6, 1]

2
6
5
0

Searching on DL



```
def search DLL (self, nodevalue):
```

```
if self.head == None:
```

```
return "There is no element to the list"
```

```
else:
```

```
tempnode = self.head
```

```
while tempnode:
```

```
if tempnode.value == nodevalue:
```

```
return tempnode.value
```

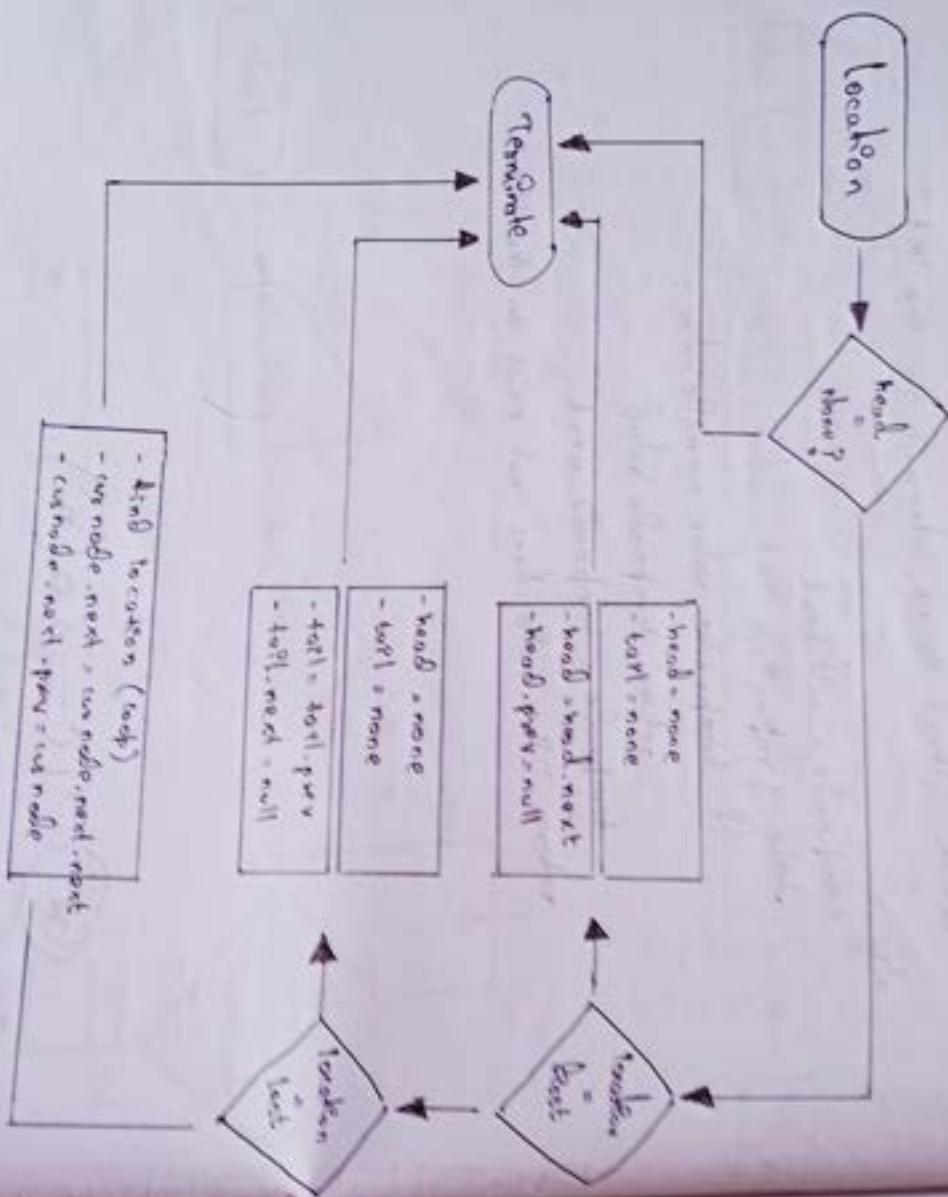
```
tempnode = tempnode.next
```

```
return "The node does not exist in the list"
```

$$\star \quad T C \rightarrow O(n)$$

$$\star \quad SC \rightarrow O(1)$$

○ Deletion In DLL



Deletion In DLL (Method)

TC - $O(n)$
SC $\rightarrow O(1)$

```
def deleteNode(self, location):
    if self.head == None:
        return "There is no element in DLL"
    else:
        if location == 0:
            if self.head == self.tail:
                self.head = None
                self.tail = None
            else:
                self.head = self.head.next
                self.head.prev = None
        elif location == 1:
            if self.head == self.tail:
                self.head = None
                self.tail = None
            else:
                self.tail = self.tail.prev
                self.tail.next = None
        else:
            tempnode = self.head
            index = 0
            while index < location - 1:
                tempnode = tempnode.next
                index += 1
            tempnode.next = tempnode.next.next
            tempnode.next.prev = tempnode
```

return "The node has been successfully deleted"

Diagram illustrating the deletion process:

- Initial state: A linked list with three nodes (represented by rectangles) connected by arrows pointing right.
- Step 1: Node 1 is identified as the head. An arrow points from Node 1 to its "head" label.
- Step 2: Node 1 is removed. Its "head" label is crossed out, and it is labeled "temp".
- Step 3: Node 2 becomes the new head. An arrow points from Node 2 to its "head" label.
- Step 4: Node 2 is modified to point to Node 3 (its original next node).
- Step 5: Node 3 becomes the new tail. An arrow points from Node 3 to its "tail" label.
- Step 6: Node 3 is modified to point to Node 2 (its original previous node).

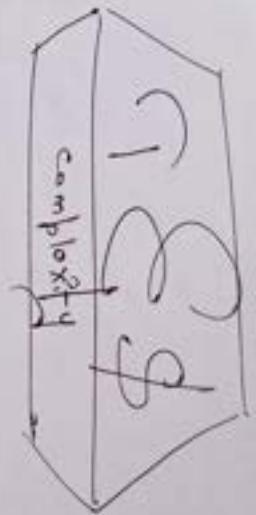
Deleting entire DLL

TC → O(n)
BC → O(1)

Step 1: While traversing through the DLL
set the prev reference of node (del node) to null.

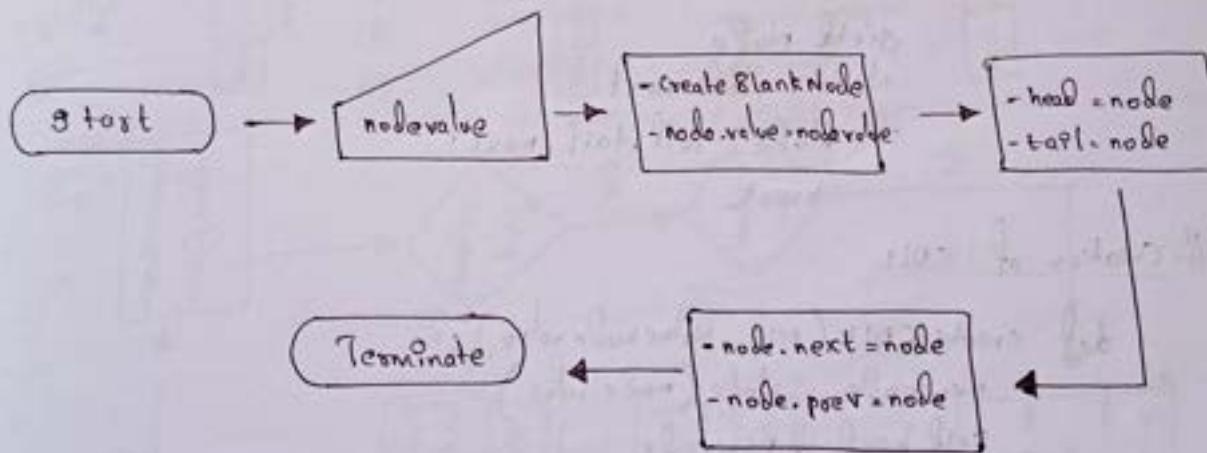
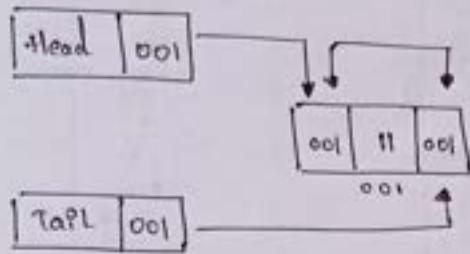
Step 2: Set head and tail to null.

```
def delete_entire_dll(self):
    if self.head == None:
        raise "There is no node in DLL"
    else:
        temp_node = self.head
        while temp_node:
            temp_node.prev = None
            temp_node = temp_node.next
        self.head = None
        self.tail = None
    return "The DLL has been successfully deleted."
```



	Time - space	Space - space
Creation	$O(1)$	$O(1)$
Insertion	$O(n)$	$O(1)$
Searching	$O(n)$	$O(1)$
Traversing (forward, backward)	$O(n)$	$O(1)$
Deletion of a node	$O(n)$	$O(1)$
Deletion of entire LL	$O(n)$	$O(1)$

CDLL



TC $\rightarrow O(1)$
SC $\rightarrow O(1)$

for creation

Class Node:

```
def __init__(self, value=None):  
    self.value = value  
    self.next = None  
    self.prev = None
```

Class CDLL:

```
def __init__(self):  
    self.head = None  
    self.tail = None
```

```
def __iter__(self):  
    node = self.head  
    while node:  
        yield node  
        node = node.next  
        if node == self.tail.next:  
            break
```

Creation of DLL

```
def create_DLL(self, valuenodevalue):
```

→ newnode = Node(node value)

self.head = newnode

self.tail = newnode

newnode.prev = newnode

newnode.next = newnode

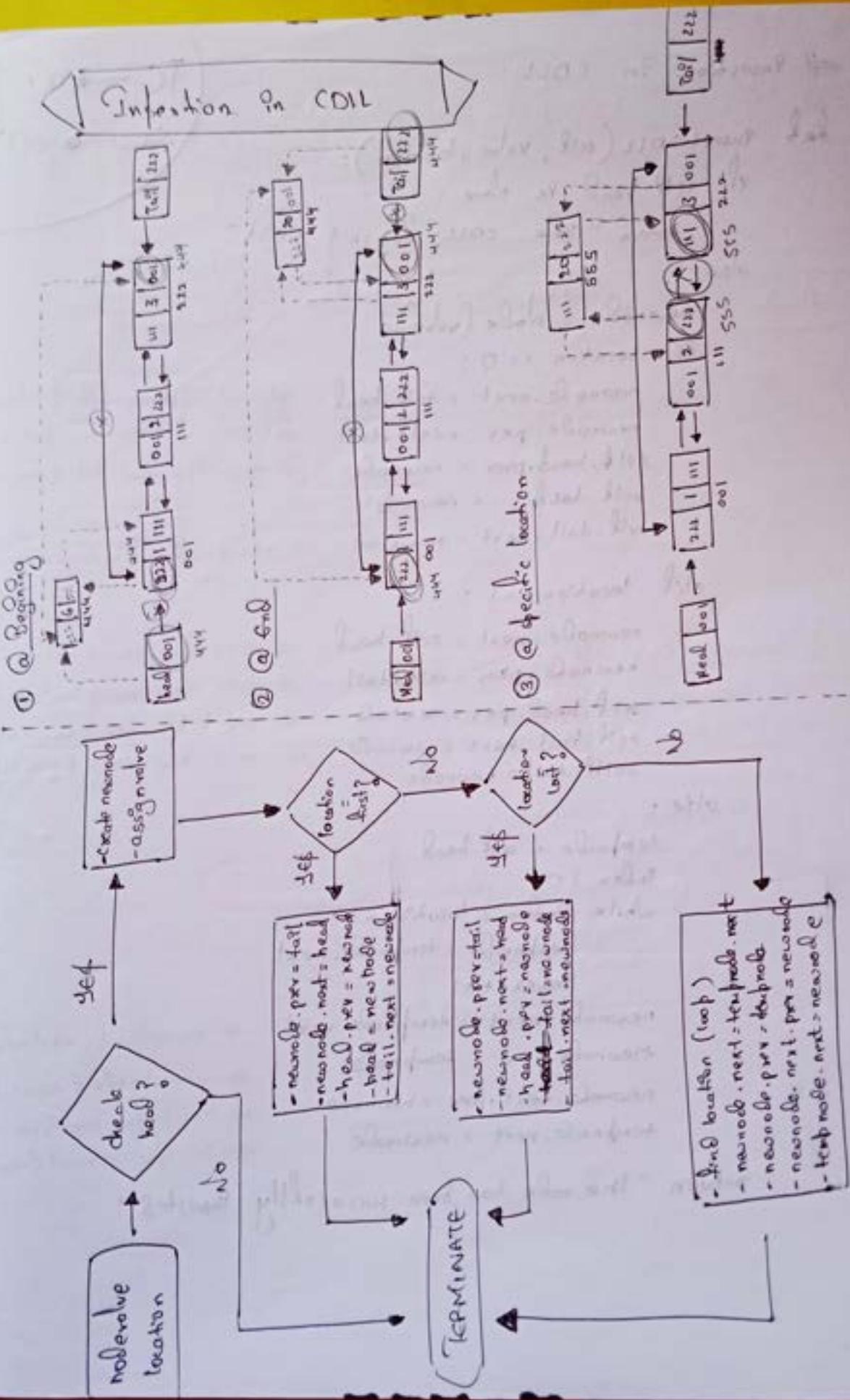
return "The DLL has been created successfully"

```
CircularDLL = DLL()
```

```
print(CircularDLL.create_DLL(5))
```

```
print([node.value for node in CircularDLL])
```

Infection in CDLL



Insertion in DLL:

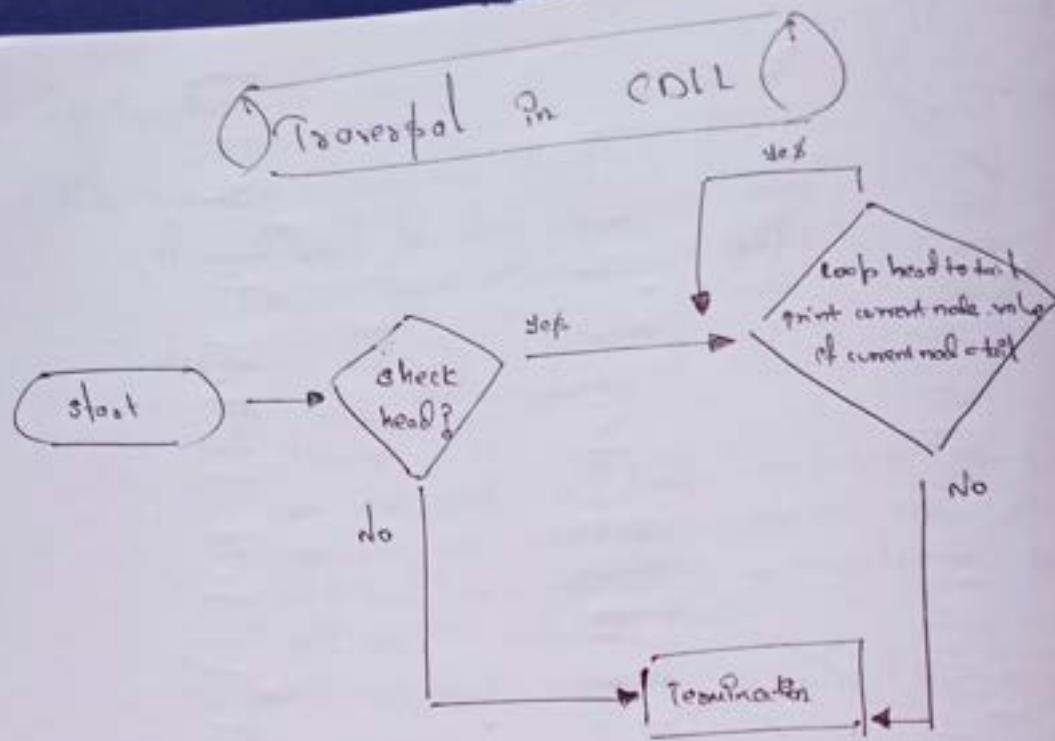
```
def insertDLL(self, value, location):
    if self.head is None:
        return "The DLL doesn't exist"
    else:
        newnode = Node(value)
        if location == 0:
            newnode.next = self.head
            newnode.prev = self.tail
            self.head.prev = newnode
            self.head = newnode
            self.tail.next = newnode
        elif location == 1:
            newnode.next = self.head
            newnode.prev = self.tail
            self.head.prev = newnode
            self.tail.next = newnode
            self.tail = newnode
        else:
            tempnode = self.head
            index = 0
            while index < location - 1:
                tempnode = tempnode.next
                index += 1
            newnode.next = tempnode.next
            newnode.prev = tempnode
            newnode.next.prev = newnode
            tempnode.next = newnode
    return "The node has been successfully inserted"
```

(TC $\rightarrow O(1)$)
SC $\rightarrow O(1)$)

init b/w newnode & 1st node
link b/w 1st & newnode
move list b/w 1st & newnode
move tail b/w last & newnode

init b/w new & last node
link b/w new & last node
rev. l b/w 1st & new node
rev. l b/w last & new node

newnode & nextnode
current node & newnode
Rev. l b/w next & newnode
l.l b/w current & newnode



```

def traverse DLL(self):
    if self.head is None:
        return "There's no node to traverse"
    else:
        tempnode = self.head
        while tempnode:
            print tempnode.value
            if tempnode == self.tail:
                break
            tempnode = tempnode.next
  
```

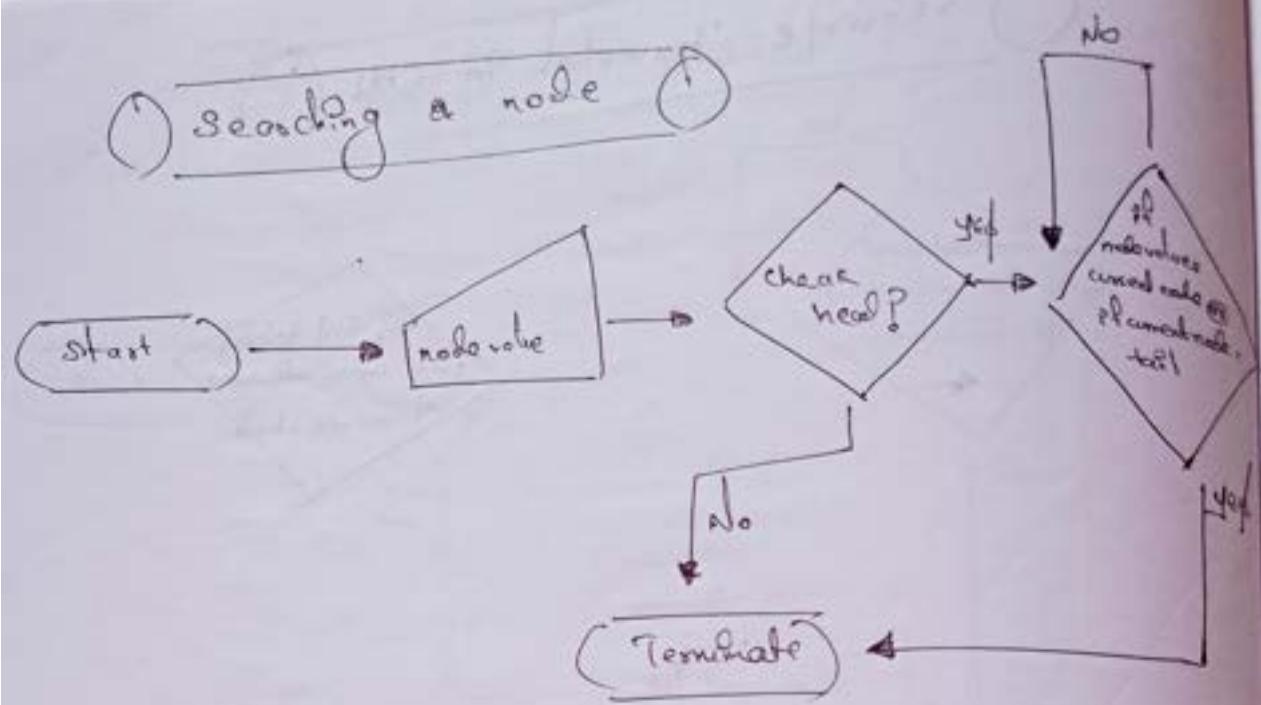
$T C \rightarrow O(n)$
 $S C \rightarrow O(1)$

Reverse Traversal in DLL

```
loop tail to head  
print currentnode.value  
if currentnode == head
```

```
def Reveresetraverse DLL (self):  
    if self.head is None:  
        return "There's no node to Reverse traverse"  
    else:  
        tempnode = self.tail  
        while tempnode:  
            print tempnode.value  
            if tempnode == self.head:  
                break  
            tempnode = tempnode.nextprev
```

REM



```

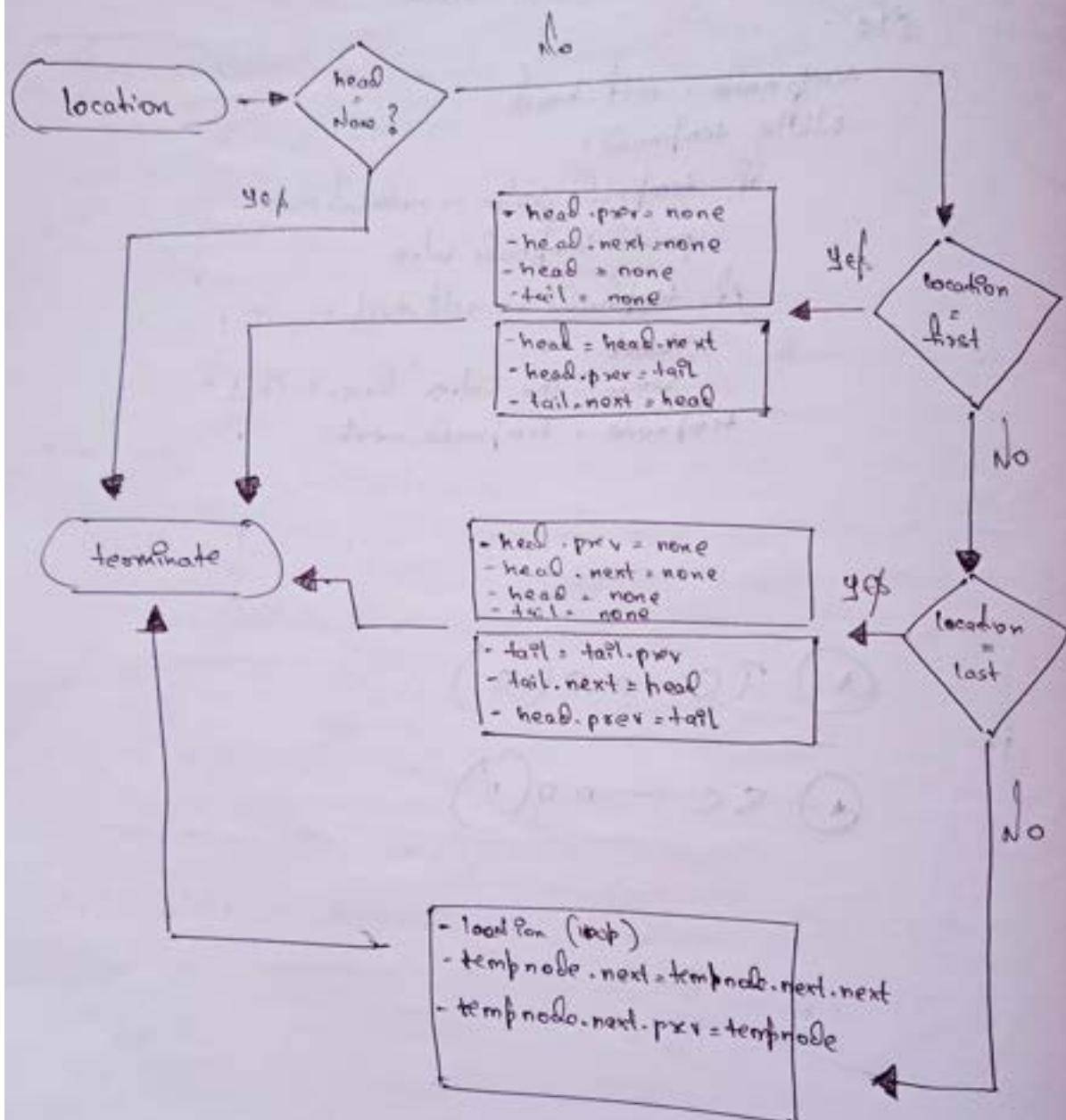
def search CDLL (self , nodevalue):
    if self.head == None:
        return "No node in DLL"
    else:
        tempnode = self.head
        while tempnode:
            if tempnode.value == nodevalue:
                print tempnode.value
            if tempnode == self.tail:
                break
            tempnode = tempnode.next

```

① TC $\rightarrow O(n)$

② SC $\rightarrow O(1)$

Deleting Node in CDLL



```
def deleteCDLL(self, location):  
    if self.head == None:  
        return "there no node to delete"  
    else:
```

if location == 0 : *(Not) Head*

if self.head == self.tail :

self.head = None

self.tail = None

head.prev = None

head =

head.next = None

Del by Ref that
points to self.

else :

self.head = self.head.next # link b/w head & 2nd

self.head.prev = self.tail # Rev l b/w 2nd & tail

self.tail.next = self.head # link b/w last & 2nd

elif location == 1 :

if self.head == self.tail

else :

self.tail = self.tail.prev # link b/w tail & by last node

self.tail.next = self.head # b/w next b/w last & off 1st

self.head.prev = self.tail # RV-L b/w 1st & the rest

else :

tempnode = self.head

index = 0

while index < location - 1:

tempnode = tempnode.next

index += 1

tempnode.next = tempnode.next.next # b/w temp & node after node

tempnode.next.prev = tempnode # b/w b/w cur & node after node

print ("the node has been deleted")

$T C \rightarrow O(n)$
 $S C \rightarrow O(1)$

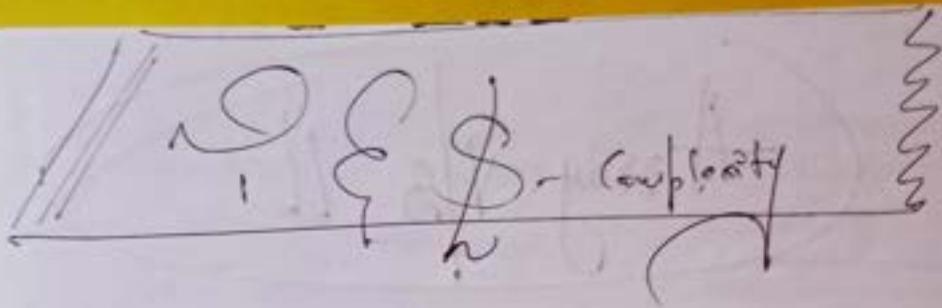
Current & Temp

Deleting entire CDLL

```
def deleteCDLL(self):
    if self.head == None:
        return "There's no node to delete"
    else:
        self.tail.next = None # Del list b/w last & 1st node
        tempnode = self.head
        while tempnode:
            tempnode.prev = None
            tempnode = tempnode.next
        self.head = None
        self.tail = None
    print("The DLL has been Deleted")
```

$$\Theta(n) \rightarrow O(n)$$

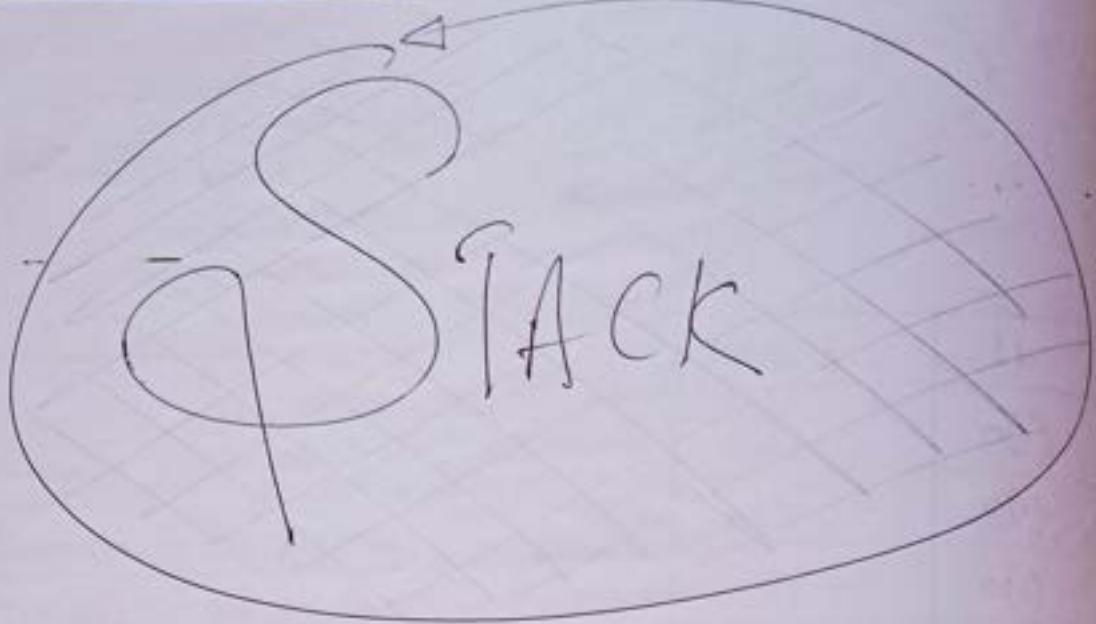
$$\Theta(1) \rightarrow O(1)$$



	(lime-stone)	(sporo-stone)
Creation	$O(1)$	$O(1)$
Insertion	$O(n)$	$O(1)$
Searching	$O(n)$	$O(1)$
Traversing (forward, backward)	$O(n)$	$O(1)$
Deletion of a node	$O(n)$	$O(1)$
Deletion of Entire coll	$O(n)$	$O(1)$

Array VS LL

	ARRA Y	LL
Creation	$O(1)$	$O(1)$
Insertion @ 1st position	$O(1)$	$O(1)$
Insertion @ end position	$O(1)$	$O(1)$
Insertion @ n^{th} position	$O(1)$	$O(n)$
Searching in Unsorted Data	$O(n)$	$O(n)$
Searching in Sorted Data	$O(\log n)$ - binary search	$O(n)$
Increasing	$O(n)$	$O(n)$
Deletion @ 1st position	$O(1)$	$O(1)$
Deletion @ last position	$O(1)$	$O(n) / O(1)$
Deletion @ n^{th} position	$O(1)$	$O(n)$
Deletion of entire Array / LL	$O(1)$	$O(n) / O(1)$
Access n^{th} element	$O(1)$	$O(n)$

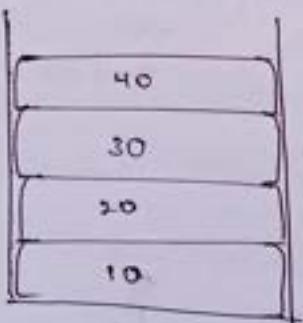


A Data-structure



that stores items in

a L.I.F.O - Last In first Out
manner.



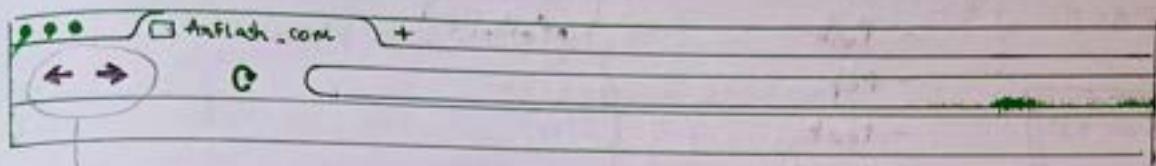
to remove 10

& how to let remove

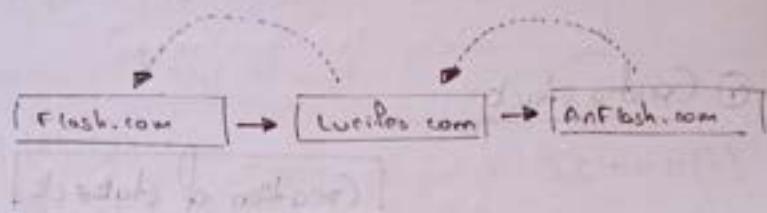
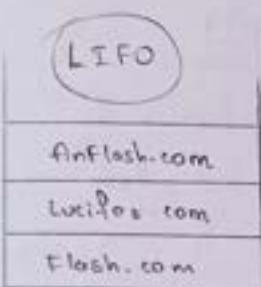
40 - 30 - 20

& then 10

Dept example:



→ Work on stack



Common Stack Operations

- - Create stack
- Push
- Pop
- Peek
- IsEmpty
- IsFull
- DeleteStack

stackops

① Create stack

Creation of stack

Stack using LIST

- Easy to implement
- Space problem when it grows

Stack using L.L

- Fast Performance
- Implementation is not easy

delete

```
def delete(self):  
    self.list = None
```

{
TC → O(1)
SC → O(1)

This is stack with NOT LIMITED space to list

Class Stack :

```
def __init__(self):
    self.list = []
def __str__(self):
    values = self.list.reverse()
    values = [str(x) for x in self.list]
    return '\n'.join(values)
```

will
work to
display like
elements
3
2
1

LIST

TC → O(1)
SC → O(1)

isempty : to check if stack is empty

```
def isempty(self):
    if self.list == []:
        return True
    else:
        return False
```

TC → O(1)
SC → O(1)

push : to add elements in stack

```
def push(self, value):
    self.list.append(value)
    return "The element has been inserted"
```

Anamotized
TC → O(n²)
SC → O(1)

pop : removes & displays last element in stack

```
def pop(self):
    if self.isempty():
        return "There's no element in stack"
    else:
        return self.list.pop()
```

TC → O(1)
SC → O(1)

peek : will return the last element in stack [not remove]

```
def peek(self):
    if self.isempty():
        return "There's no element in stack"
    else:
        return self.list[len(self.list)-1]
```

TC → O(1)
SC → O(1)

Creating Stack with LIMITED size

```
class stack:  
    def __init__(self, maxsize):  
        self.maxsize = maxsize  
        self.list = []  
  
    def __str__(self):  
        values = self.list.reverse()  
        values = [str(x) for x in self.list]  
        return '\n'.join(values)  
  
    # isEmpty  
    def isEmpty(self):  
        if self.list == []:  
            return True  
        else:  
            return False  
  
    # isFull  
    def isfull(self):  
        if len(self.list) == self.maxsize:  
            return True  
        else:  
            return False  
  
    # Push  
    def push(self, value):  
        if self.isfull():  
            return "Stack is full"  
        else:  
            self.list.append(value)  
            return "The element has been inserted"  
TC → amortized  
SC → O(1)
```

pop

```
def pop(self):
    if self.is_empty():
        return "There's no element in stack"
    else:
        return self.list.pop()
```

TC - O(1)

SC - O(1)

peek

```
def peek(self):
    if self.is_empty():
        return "There's no element in stack"
    else:
        return self.list[ten(self.list)-1]
```

TC - O(1)

SC - O(1)

delete

```
def delete(self):
    self.list = None
```

T | S complexity & Stack with LIST

	TIME - STONE	SPACE - STONE
Create stack	$O(1)$	$O(1)$
Push	$O(1) / O(n^2)$	$O(1)$
top	$O(1)$	$O(1)$
Peek	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
Delete entire stack	$O(1)$	$O(1)$

Stack using LL

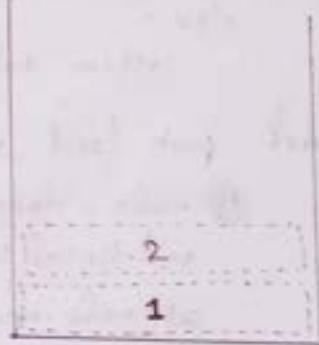
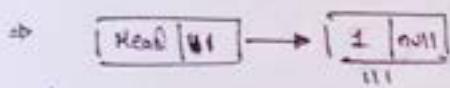
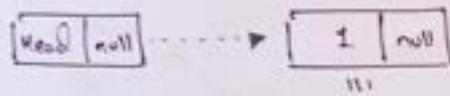


- ① To create a stack using LL [be appending/pushing elements
↓
of LL to stack]

Create an object of SLL class
↳

Create blank nodes, assign values & then
push it to the stack.

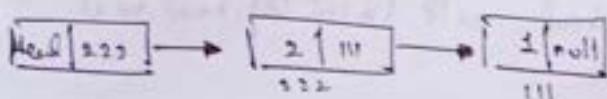
- # After creating a blank node &
assigning a value to it;
create a reference b/w head & the node



- # When u update the pointers/link, the node can be pushed to stack

lets add 1 more node [2]
 |
 ||

Rem: whenever you add a new node to the LL it gets appended @
the beginning & whenever u delete the last element gets deleted



Class Node :

```
def __init__(self, value, None):
    self.value = value
    self.next = next
```

```
def __init__(self):
    tempnode = self.head
    while tempnode:
        yield tempnode
        tempnode = tempnode.next
```

Class Linkedlist :

```
def __init__(self):
    self.head = None
```

Class Stack :

```
def __init__(self):
    self.linkedlist = linkedlist()
```

Creating stack
using LL

(like LIFO or stack)

def isEmpty(self):

if self.linkedlist.head == None:

return True

else:

return False

def push(self, value):

~~node~~ node = Node(value)

self.linkedlist.head = ~~node~~ node

self.node.next = self.linkedlist.head

1. New node
2. No free page
3. No head page

def pop(self):

if ~~is empty~~ self.isEmpty():

return "No element to pop"

else:

~~nodevalue~~ nodevalue = self.linkedlist.head.value

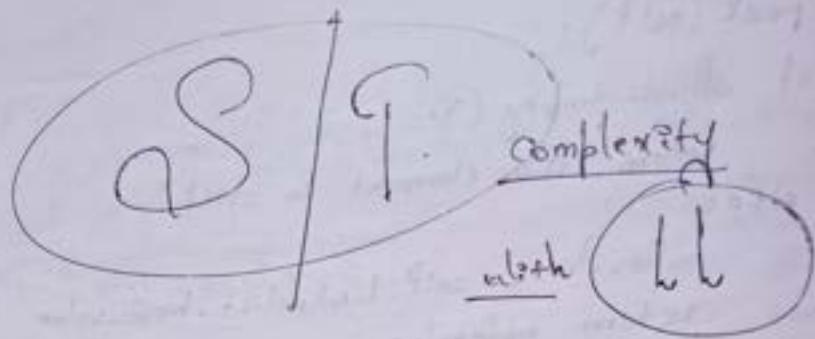
self.linkedlist.head = self.linkedlist.head.next

return nodevalue

last node

```
def peek(self):
    if self.isEmpty():
        return "No element in stack"
    else:
        nodeValue = self.linkedlist.head.value
        return nodeValue

def delete(self):
    self.linkedlist.head = None
```



	<u>Time - stone</u>	<u>Space - stone</u>
Create stack	$O(1)$	$O(1)$
Push	$O(1)$	$O(1)$
Pop	$O(1)$	$O(1)$
Peek	$O(1)$	$O(1)$
IsEmpty	$O(1)$	$O(1)$
Delete Entire stack	$O(1)$	$O(1)$

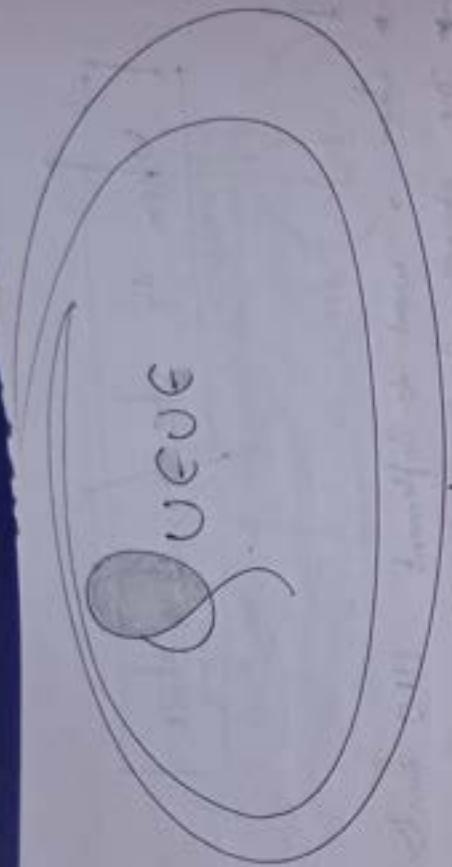
When to use / avoid - Stack

Use

- when u want to implement LIFO functionality
- The chance of Data corruption is minimum
 - e.g. u cannot add data @ the middle of stack.

Avoid

- Random access is Not possible
 - & as the Data size ↑ the accessibility ↓

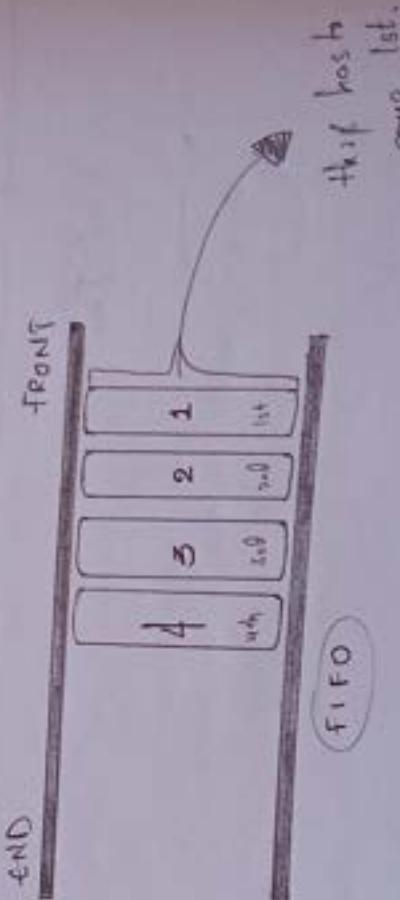


Data structure

The first item in first out manner

FIFO

④ New addition to queue happens @ the end of queue.



First push
comes 1st.

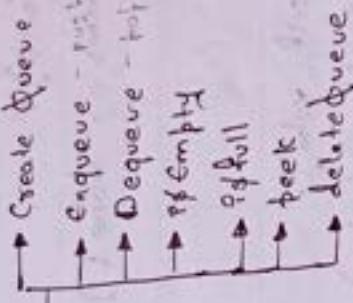
When do we need Queue

↳ When u want to apply FIFO method:

e.g. Utilize first coming data first,
while others wait for their turn.

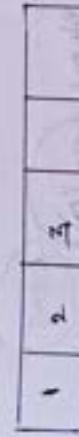
Ex: Ticket booking, Ordering food,
printing sheets, Call center phone system,

* Queue Operations:

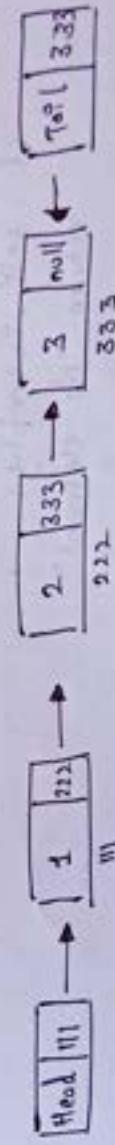


* Implementation:

- ① Python list: ↗ Queue without capacity
- ② Linked list: ↗ Queue with capacity (Circular Queue)
 - ↳ Time efficient.



② Linked list:



Queue without maxsize - list

```
class Queue:  
    def __init__(self):  
        self.items = []  
  
    def size(self):  
        return len(self.items)  
  
    def isEmpty(self):  
        if self.items == []:  
            return True  
        else:  
            return False  
  
    def enqueue(self, value):  
        self.items.append(value)  
        return "The element has been inserted @ the end of queue"  
  
    def dequeue(self):  
        if self.isEmpty():  
            return "No element"  
        else:  
            x self.items.pop(0)  
            return self.items.pop(0)  
  
    def peek(self):  
        if self.isEmpty():  
            return "No element"  
        else:  
            return self.items[0]
```

def delete(self):

self.items = None

	O(1)	O(n)
Create Queue	O(1)	O(1)
Enqueue	O(n)	O(1)
Dequeue	O(n)	O(1)
Peek	O(1)	O(1)
IsEmpty	O(1)	O(1)
Delete entire Queue	O(1)	O(1)

Queue with fixed capacity (Circular Queue)

- # when we create a queue \uparrow (c, q)
- ④ we create a blank list with fixed size q
- 2 variables called start \uparrow & top \uparrow

size = 6

start = -1

top = -1

None	None	None	None	None
------	------	------	------	------

- ⑤ when you enqueue an element !

- ⑥ enqueue (5)

start = 0

top = 0

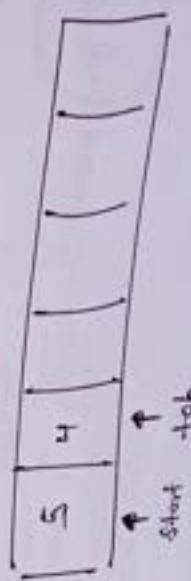


- (start & top will be pointing @ index [0])

- ⑦ enqueue (4)

start = 0

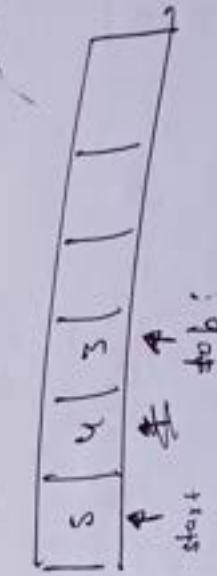
top = 1



- ⑧ enqueue (3)

start = 0

top = 2



② Enqueue (2)

start = 0

top = 2

5	4	3	2	1	
---	---	---	---	---	--

Start

Top

③ Now when you dequeue the element (obvio - let go of it),

④ Dequeue () → 5

start = 1

top = 3

-	4	3	2	
---	---	---	---	--

Start

Top

⑤ Enqueue (6)

Enqueue (2)

start = 1

top = 5

4	3	2	6	7	
---	---	---	---	---	--

Start

Top

⑥ Now see ↓ which is why we call → Circular Queue

⑦ Dequeue () → 4

Enqueue (3)

start = 2

top = 0

9	3	2	6	7	
---	---	---	---	---	--

Start

Top

(above + 10) goes back to 1

↑ 10 goes back to 1

↑ 10 goes back to 1

Class Queue:

```
def __init__(self, maxsize):
    self.maxsize = maxsize
    self.items = maxsize * [None]
    self.start = -1
    self.top = -1

    def __str__(self):
        values = [str(x) for x in self.items]
        return "\n".join(values)

    def isfull(self):
        if self.top + 1 == self.start:
            return True
        elif self.start == 0 and self.top + 1 == self.maxsize:
            return True
        else:
            return False

    def isempty(self):
        if self.top == -1:
            return True
        else:
            return False

    def enqueue(self, value):
        if self.isfull():
            return "The queue is full"
        else:
            self.items[self.top + 1] = value
            self.top += 1
            return "Enqueued successfully"

    def dequeue(self):
        if self.isempty():
            return "The queue is empty"
        else:
            value = self.items[self.start]
            self.items[self.start] = None
            self.start += 1
            return value
```

```

def enqueue (self, value):
    if self.isFull ():
        return "The queue is full."
    else:
        if self.top + 1 == self.maxsize: # top element
            points last elem
            self.top = 0
        else:
            self.top += 1
            if self.start == -1: # no elements
                self.start = 0
            self.items [self.top] = value
        return "The element is inserted."
    
```



```

def dequeue (self):
    if self.isEmpty ():
        return "There's no element in Queue"
    else:
        firstElement = self.items [self.start]
        start = self.start
        if self.start == self.top: # only 1 element
            self.start = -1
            self.top = -1
        elif self.start + 1 == self.maxsize:
            self.start = 0
        else:
            self.start += 1
        self.items [start] = None
        return firstElement
    
```

```

def peek(self):
    if self.isEmpty():
        return "None"
    else:
        return self.items[self.start]

```



```

def delete(self):
    self.items = self.items * (None)
    self.top = -1
    self.start = -1

```

	Time-stone	Space-stone
Create Queue	$O(1)$	$O(n)$
enqueue	$O(1)$	$O(1)$
dequeue	$O(1)$	$O(1)$
Peek	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
isFull	$O(1)$	$O(1)$
Delete entire queue	$O(1)$	$O(1)$

Queue using linked list

```
class Node:  
    def __init__(self, value=None):
```

```
        self.value = value
```

```
        self.next = None
```

```
    def __str__(self):  
        return str(self.value)
```

```
class linkedList:
```

```
    def __init__(self):
```

```
        self.head = None
```

```
        self.tail = None
```

```
    def __str__(self):
```

```
        curNode = self.head
```

```
        while curNode:
```

```
            yield curNode
```

```
            curNode = curNode.next
```

```
class Queue:
```

```
    def __init__(self):
```

```
        self.linkedlist = linkedlist()
```

```
    def __str__(self):
```

```
        values = [str(x) for x in self.linkedlist]  
        return '\n'.join(values)
```

```
def enqueue(self, value):
```

```
    newnode = Node(value)
```

```
    if self.linkedlist.head == None:  
        self.linkedlist.head = newnode  
    else:  
        self.linkedlist.tail.next = newnode  
    self.linkedlist.tail = newnode
```

```
else:
```

```
    self.linkedlist.tail.next = newnode  
    self.linkedlist.tail = newnode
```

In Queue

Nodes get opened
@ last

or FIFO

```
def isEmpty(self):
```

```
    if self.linkedlist.head == None:  
        return True
```

```
else:
```

```
    return False
```

```
def dequeue(self):
```

```
    if self.isEmpty():
```

```
        return "No element in Queue"
```

```
else:
```

```
    tempnode = self.linkedlist.head  
    if self.linkedlist.head != None:  
        self.linkedlist.head = tempnode.next  
        tempnode.next = None  
    else:
```

```
        self.linkedlist.head = self.linkedlist.tail  
        self.linkedlist.tail = None
```

```

def peek(self):
    if self.isEmpty():
        return "No data"
    else:
        return self.linkedList.head

```

```

def delete(self):
    self.linkedList.head = None
    +-----+
    +-----+

```

	Time-complexity	Space-complexity
Create Queue	$O(1)$	$O(1)$
Enqueue	$O(1)$	$O(1)$
Dequeue	$O(1)$	$O(1)$
Peek	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
Delete from Queue	$O(1)$	$O(1)$

	List with no capacity limit		List with capacity (Circular Queue)		Linked list	
	T.C	S.C	T.C	S.C	T.C	S.C
Create Queue	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Enqueue	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Dequeue	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Peek	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
IsEmpty	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
isFull	-	-	$O(1)$	$O(1)$	$O(1)$	-
DeleteQueue	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

Space Complexity

Time Complexity

Space Complexity

Time Complexity

Space Complexity

Time Complexity

FIFO Queue Module

- * In Python we can create Queue using 3 different modules
 - ↳ Collections module
 - ↳ Queue module
 - ↳ Multithreading module.

① Collections module

- * From Collections module we use deque class.

```
from collections import deque
customQueue = deque(maxlen=3)
print(customQueue)
```
- * The deque class implements a double ended queue
 - ↳ which supports adding & removing elements from either end in O(1)
- * Queue objects are implemented as Double-LL → which gives them excellent performance for enqueueing & dequeuing;
so they can serve as queue as stack.
- * Here we are creating FIFO Queue Method →
 - ↳ append() - to create queue with @ with
 - ↳ popleft() - to queue
 - ↳ append() - to queue
 - ↳ popright() - to queue
 - ↳ clear() - to delete queue

```
deque([1,2,3], maxlen=3)
2
deque([3,4], maxlen=3)
None
deque([5], maxlen=3)
```

Queue Module

- ④ Queue module implements

- multi-producer, - multi-consumer queue.

(especially useful in threaded programming where info must be exchanged safely b/w multiple threads)

- ⑤ This module implements 3 types of queue,

which differs only in order in which the entries are retrieved

- FIFO queue - [The list lists added]
- LIFO queue - [The most recently added entry is the last added]
- Priority queue - [The entries are kept sorted & the lowest value entry is returned]

- ⑥ will look @ FIFOqueue

Methods

- `q.size()` - returns current size of queue
- `empty()`
- `full()`
- `put()` - to queue
- `get()` - dequeues
- `last_value()`
- `join()`

import queue as q

customqueue = q.Queue(maxsize=

print(customqueue.put(1))

customqueue.put(2)

customqueue.put(3)

```
print(customqueue.full())
print(customqueue.get())
print(customqueue.qsize())
```

True ... 1, 2, 3

1
2

Multiprocessing queue module

- This is like a queue module & all methods are
- Multiprocessing module → it's meant for sharing data b/w processes & can store any pickleable objects.

from multiprocessing import Queue

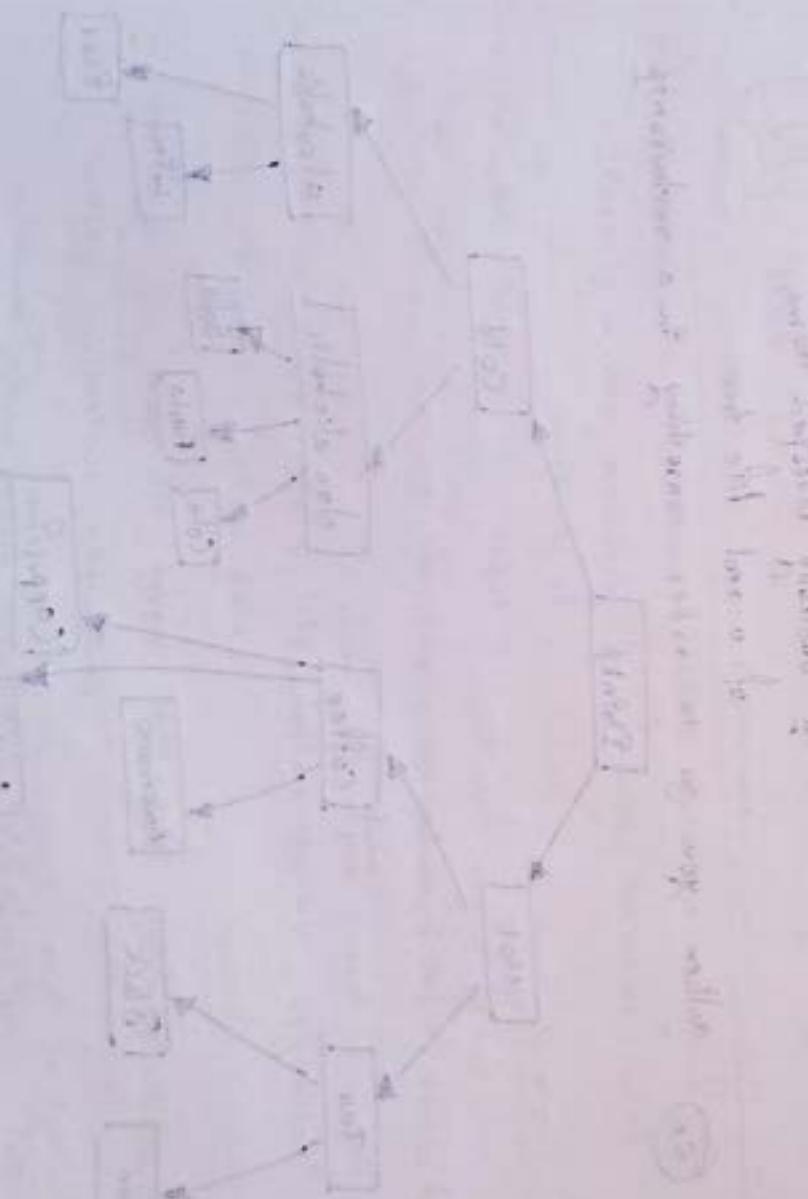
customQueue = Queue(maxsize=3)

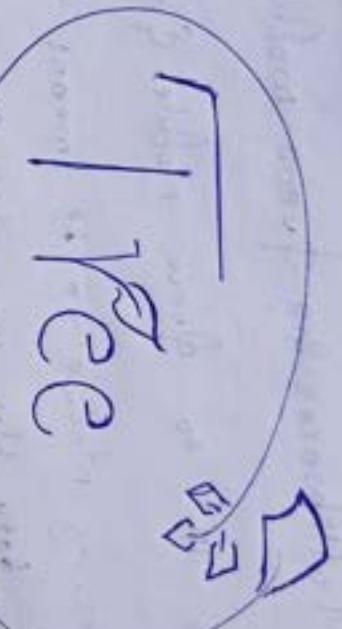
customQueue.put(1)

print(customQueue.get())

front
-task-done()
-join()

1



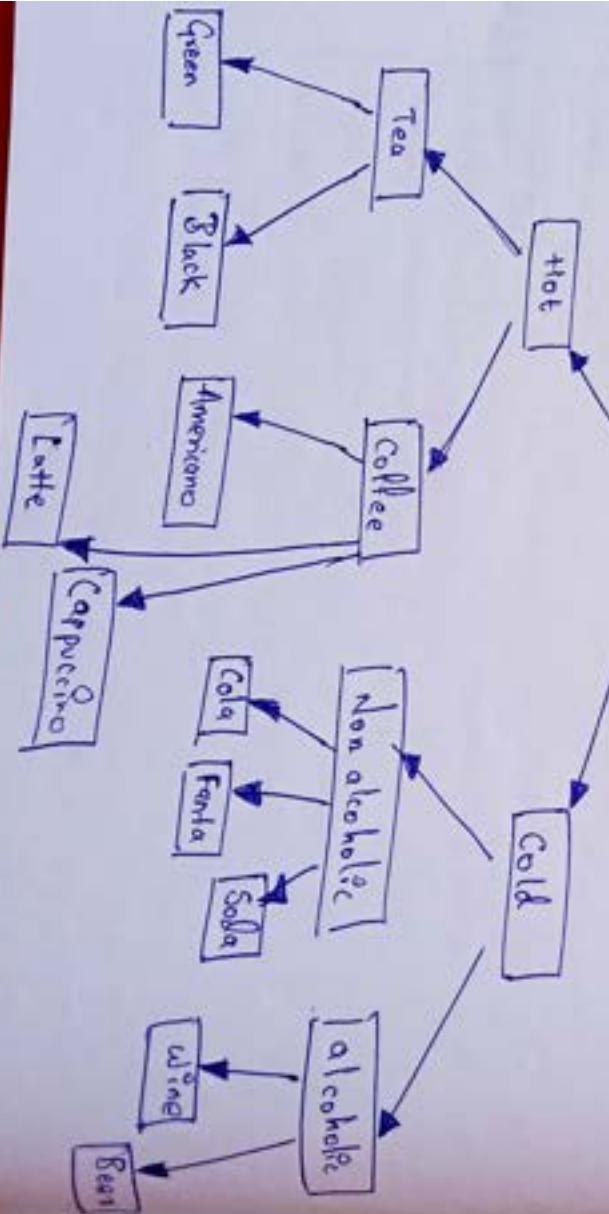


A Non-linear data structure

With hierarchical relationships of all elements,
without having any cycle.

It is basically reversed form
of a real life tree.

(*) When you go to order something in a restaurant



Properties

⇒ Represents hierarchical data

↳ i.e. every step we go down

it represents/becomes more

specialized form of its parent

⇒ each node has 2 components

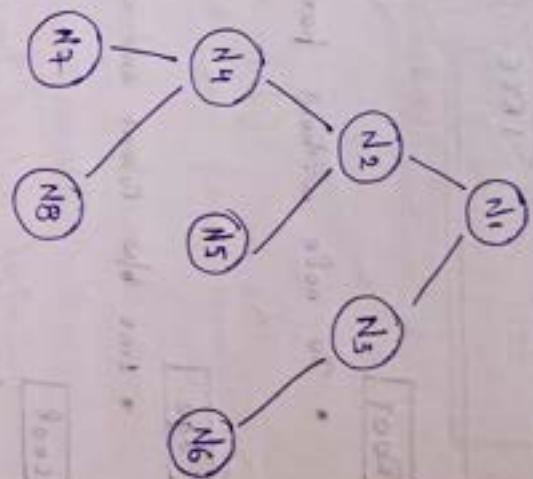
↳ Data

↳ Link to other subcategory

⇒ Basic category and sub categories
under it (i.e. root node & sub nodes)

e.g. Root category [Drinks] has [Dollar-C]
with [Sub-C] under it.

→ Each node stores the
physical location of its
sub nodes.



Balanced Tree

→ Compared to list, array... where data are arranged
linearly, the acceptability of data increases with
complexity of
operations on input size.

But since tree are non-linear ⇒ quicker, easier access to
data.

→ Store hierarchical data like folder structure, organization
structure, XML / HTML data.

→ There are many diff types tree-D.S which performs
better in various situations

↳ Binary search tree, AVL, Red Black tree, Trie

~ IEEE Terminologies

① Root

- * To node which hasn't parent

(1)

② Edge

- * Link b/w parent and child

(2)

③ Leaf

- * Node which doesn't have children

(3)

④ Sibling

- * Children of same parent

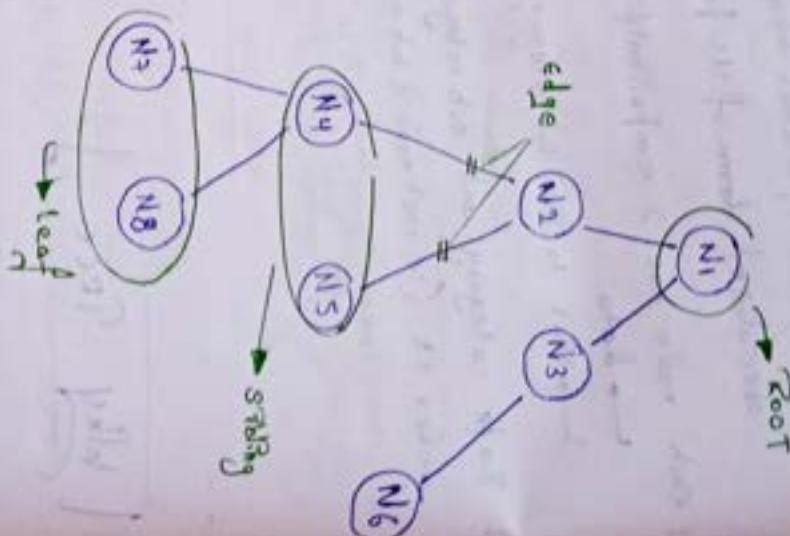
(4)

⑤ Ancestors

- * Parent, grandparent, great grandparent of a node

(5)

- Ex: $N_2 \rightarrow N_4$ — parent
 $N_2 \rightarrow N_5$ — grandparent
 $N_1 \rightarrow N_2$ — great grandparent



⑥ Depth of node :

No. of edges from root to that node

Depth of tree : depth of root node
 ↪ Generally zero

⑦ Height of node:

No. of edges from that node to the deepest node

Height of tree: height of root node

Creation of Tree using Python List

```
class TreeNode:
    def __init__(self, data, children = []):
        self.data = data
        self.children = children

    def __str__(self, level=0):
        ret = " " * level + str(self.data) + "\n"
        for child in self.children:
            ret += child.__str__(level+1)
        return ret

    def addchild(self, treeNode):
        self.children.append(treeNode)

Drinks
ksee = TreeNode('Drinks', [])
cold = TreeNode('Cold', [])
hot = TreeNode('Hot', [])

Drinks.addchild(cold)
Drinks.addchild(hot)

cola = TreeNode('Cola', [])
fanta = TreeNode('Fanta', [])
tea = TreeNode('Tea', [])
coffee = TreeNode('Coffee', [])

cold.addchild(cola)
cold.addchild(fanta)
hot.addchild(tea)
hot.addchild(coffee)
```

print(Drinks)

Output

Drinks

Cold

Cola

Fanta

Hot

Tea

Coffee

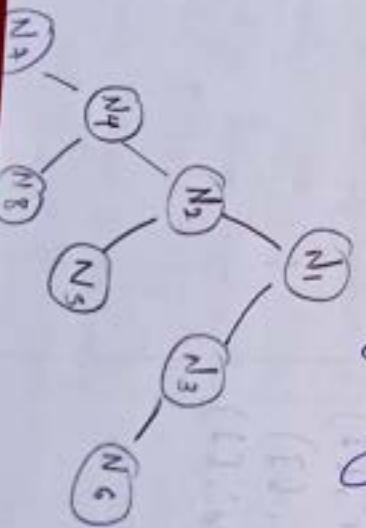
Binary Tree

- ★ Binary tree are tree data structures in which each node has atmost 2 children, often referred as left & right children.

- ★ All the data structures like BST, Heap tree, AVL, red black trees, Syntax tree are all derived from Binary Tree.

Why need?

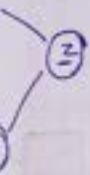
- ★ Binary trees are a prerequisite for more advanced trees like BST, AVL, Red Black Trees.
- ★ And moreover - Huffman coding problem
 - Heap priority problem
 - expression parsing problemcan be solved efficiently using binary tree



Type of Binary Tree

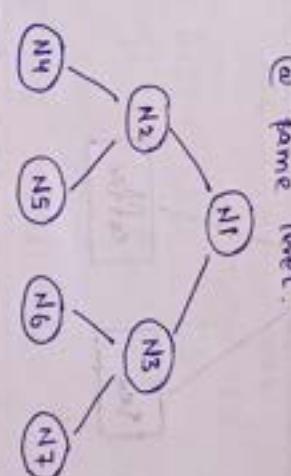
1. Full B.T

- * If each node of a BT has 0 or 2 nodes but not 1 then it's full BT

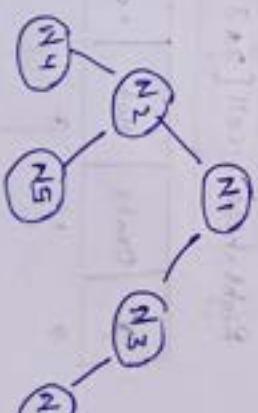


2. Perfect B.T

- * All non-leaf nodes have 2 children and they are at same depth and all leaf nodes are located @ same level.



3. Complete B.T



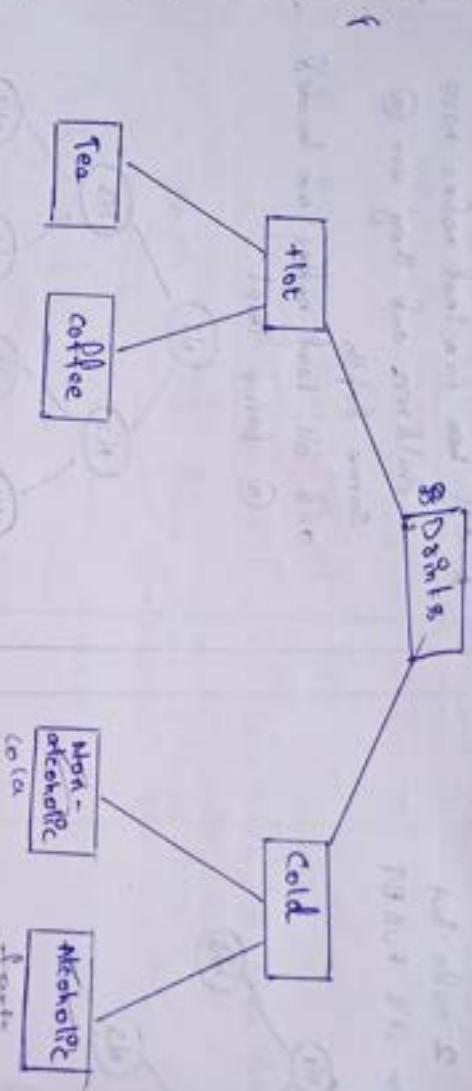
4. Balanced B.T

- * All leaf-nodes are located @ equal distance from the Root node i.e. depth should remain same of leaf-nodes

CREATION OF A B-T

- ↳ Linked list
- ↳ Python list (array)

④ Let's take an example of B-T



for hot

Left child = cell[$2 * 2$] = cell[4] → Tea

Right child = cell[$2 * 2 + 1$] = cell[5] → coffee

$$2 * n + 1$$

X	Doms	Hot	Cold	Tea	coffee			
0	0	1	2	3	4	5	6	7

for cold

Left child = cell[$2 * 3$] = cell[6] → Cola

Right child = cell[$2 * 3 + 1$] = cell[7] → Fanta

X	Doms	Hot	Cold	Tea	coffee	Cola	Fanta
0	0	1	2	3	4	5	6

① Using L.L.

Left child : Data : Right child

222	Drink	333
-----	-------	-----

111

444	flat	555
-----	------	-----

222

null	Tea	null
------	-----	------

444

555

666

777

666

777

null	coffee	null
------	--------	------

666

666

777

null	cola	null
------	------	------

666

666

777

null	fanta	null
------	-------	------

666

666

777

② Using List :

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

X								
---	--	--	--	--	--	--	--	--

- ★ We always leave last block i.e $\text{index}[6]$ - blank
↳ to take calculation.

- ★ The we place the Root node @ index[1] & based on Left child and Right child formula we place L.C & R.C.

$$\text{LEFT CHILD} = \text{cell}[2x] \quad \text{RIGHT CHILD} = \text{cell}[2x+1] \quad \left. \begin{array}{l} x \rightarrow \text{Index value of this} \\ \text{Parent node.} \end{array} \right.$$

0	1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---	---

Left child = cell [2*1] = cell[2] \rightarrow flat

Right child = cell [2*1 + 1] = cell[3] \rightarrow cold

X	Dairy	flat	cold					
---	-------	------	------	--	--	--	--	--

0 1 2 3 4 5 6 7 8

B.T. w/ creation using L.L

L.L

- Creation of tree
- Insertion of a node
- Deletion of a node
- Search for a value
- Traverse all nodes
- Deletion of tree

②

↑↑ creation of tree :

class TreeNode:

```
def __init__(self, data):
```

self.data = data

self.leftchild = None

self.rightchild = None

newBT = TreeNode("prints")

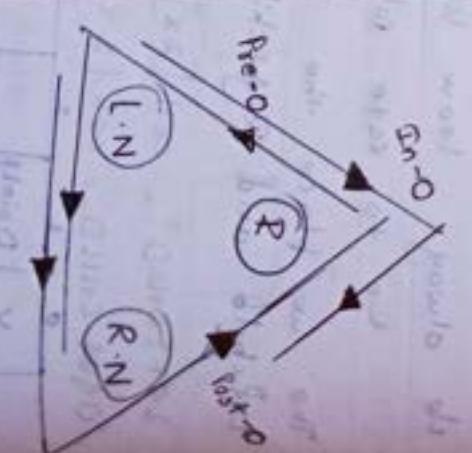
③

Traversal of B.T

Types

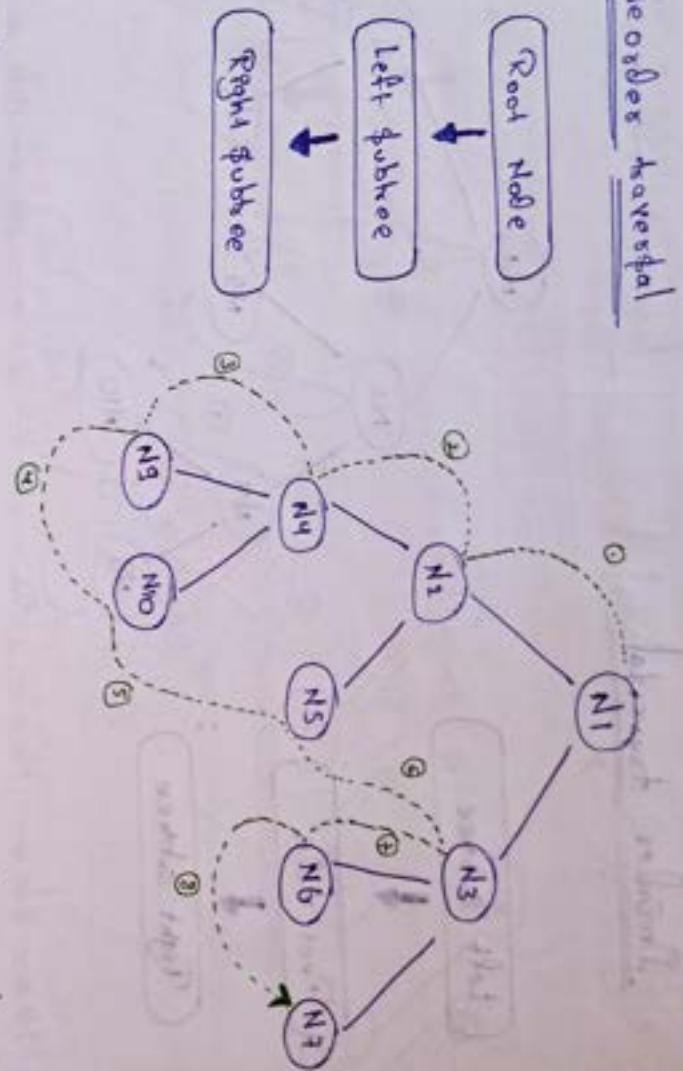
Depth first search

- Pre-order traversal
- In-order traversal
- Post-order traversal
- Breadth first search
- Level order traversal



Traversal	Order	Advantage	Disadvantage
Pre-order	R, L.N, R.N	Easy to implement	Information about right child lost
In-order	L.N, R, R.N	Information about left child lost	Time consuming
Post-order	L.N, R.N, R	Information about both children lost	Time consuming

Preorder traversal



$[O(n)] = \downarrow N_1 \rightarrow N_2 \rightarrow N_4 \rightarrow N_9 \rightarrow N_{10} \rightarrow N_5 \rightarrow N_3 \rightarrow N_6 \rightarrow N_7$

• class TreeNode :

```
newBT = TreeNode("Drinks")
leftchild = TreeNode("Hot")
rightchild = TreeNode("Cold")
```

```
newBT.leftchild = leftchild
newBT.rightchild = rightchild
```

```
def preOrderTraversal(rootNode):
    if not rootNode:
```

```
        return
```

```
    print (rootNode.data)
```

```
    preOrderTraversal(rootNode.leftchild)
```

```
    preOrderTraversal(rootNode.rightchild)
```

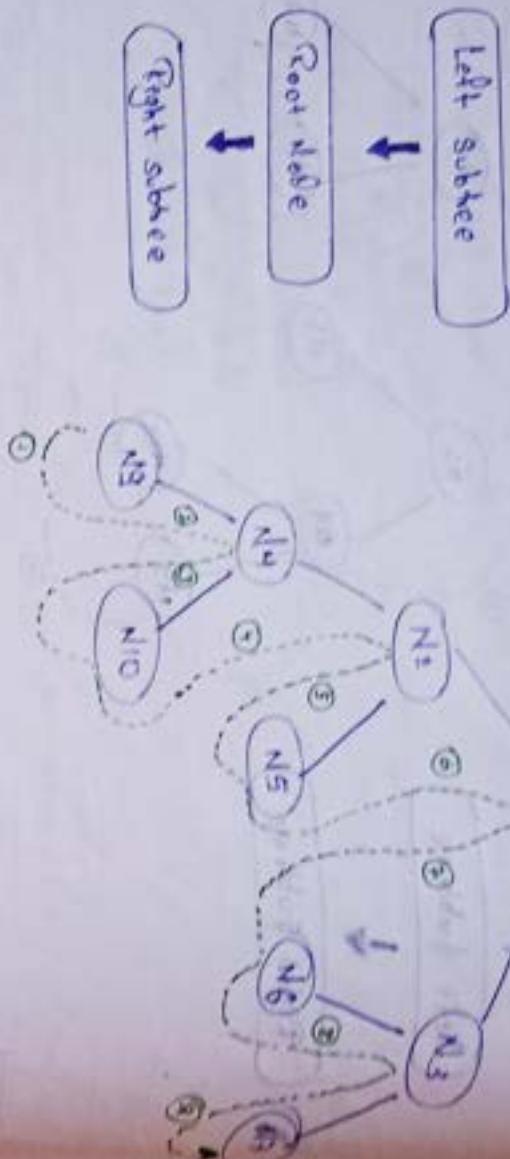
$\star Tc = O(n)$
 $\star Sc = O(n)$

↳ we ja
wir
stark
haben
zu tun

preOrderTraversal($newBT$)

Drinks
Hot
Cold

Inorder traversal



N9 → N4 → N0 → N2 → N5 → N1 → N6 → N3 → N7

```
def inorderTraversal(rootNode):
```

```
    if not rootNode:
```

return

```
    inorderTraversal(rootNode.leftchild) - - - O(n/2)
```

```
    print (rootNode.data) - - - O(1)
```

```
    inorderTraversal(rootNode.rightchild) - - - O(n/2)
```

Inorder traversal (recursion)

not
Dont
old

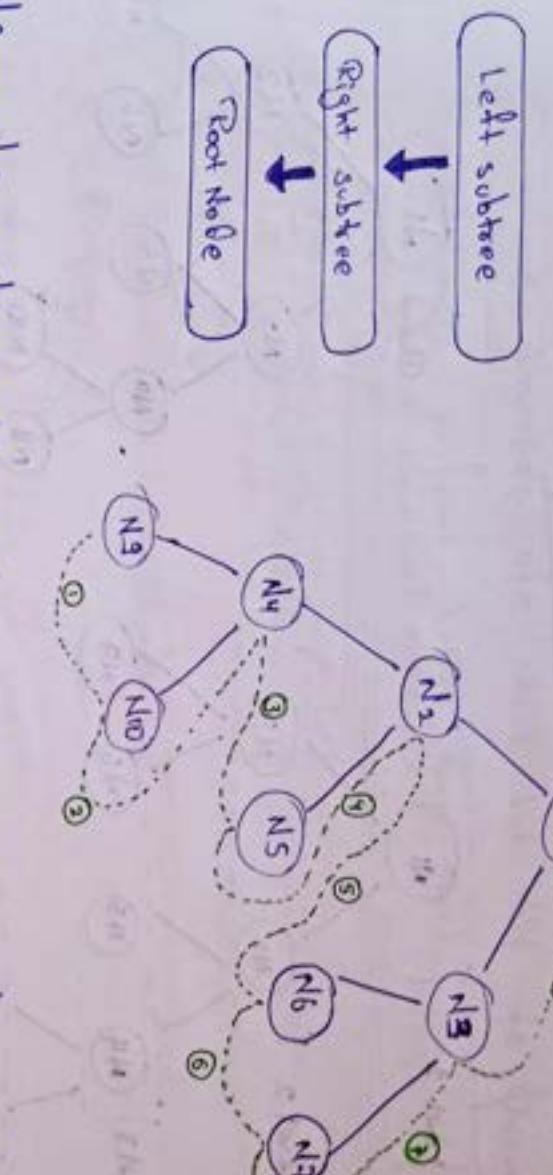
$$\textcircled{4} \quad TC \rightarrow O(n)$$

$$\textcircled{5} \quad SC \rightarrow O(n)$$

↳ Recursion → stack

Post Order traversal

[Inorder Traversal]



$N_9 \rightarrow N_{10} \rightarrow N_4 \rightarrow N_5 \rightarrow N_2 \rightarrow N_6 \rightarrow N_7 \rightarrow N_3 \rightarrow N_1$

new BT = TreeNode ("Dinty")
left child = TreeNode ("Hot")

tea = Tree Node ("Tea")
coffee = Tree Node ("coffee")

left child . left child = tea
left child . right child = coffee

Right child = TreeNode ("cold")

new BT . left child . left child

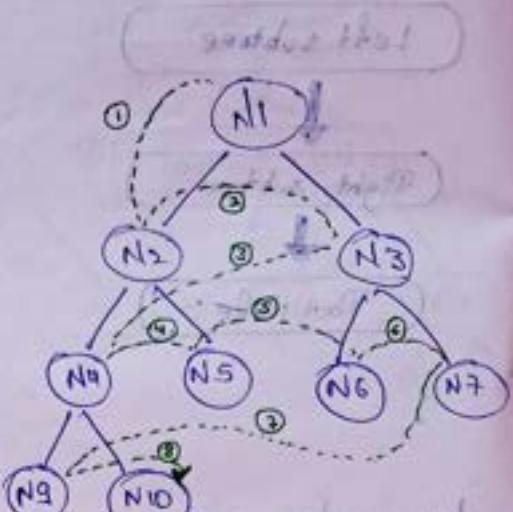
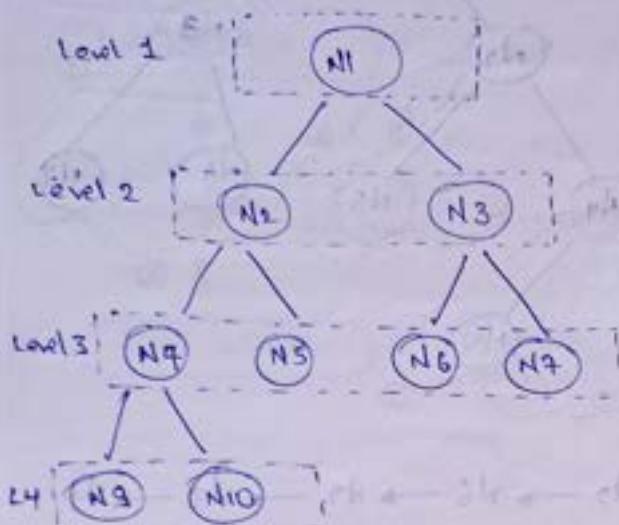
new BT . Right child . Right child

def postOrderTraversal (rootNode):

if not rootNode :
return

postOrderTraversal (rootNode.Rightchild) --> O(n)
postOrderTraversal (rootNode.Leftchild) --> O(n)
print (rootNode.data) --> O(1)

Level Order Traversal



$N_1 \rightarrow N_2 \rightarrow N_3 \rightarrow N_4 \rightarrow N_5 \rightarrow N_6 \rightarrow N_7 \rightarrow N_9 \rightarrow N_{10}$

```

import QueueLinkedList as queue
def levelOrderTraversal(rootNode):
    if not rootNode:
        return
    else:
        customQueue = queue.Queue()
        customQueue.enqueue(rootNode)
        while not (customQueue.isEmpty()):
            root = customQueue.dequeue()
            print(root.value.data)
            if (root.value.leftchild is not None):
                customQueue.enqueue(root.value.leftchild)
            if (root.value.rightchild is not None):
                customQueue.enqueue(root.value.rightchild)
  
```

```

levelOrderTraversal(newBT)
Drinks : coffee, tea, cold, hot
Hot
cold
tea
coffee
  
```

Now there will be using Level-Order traversal
as all other traversal use stack but this use Queue
& Queue always perform better than stack.

SEARCHING

```
def searchBT (rootNode, nodeValue):  
    if not rootNode:  
        return "The BT does not exist"  
    else:  
        customQueue = queue.Queue()  
        customQueue.enqueue(rootNode)  
        while not (customQueue.isEmpty()):  
            root = customQueue.dequeue()  
            if root.value.data ==nodeValue:  
                return "Success"  
            if (root.value.leftchild is not None):  
                customQueue.enqueue(root.value.leftchild)  
            if (root.value.rightchild is not None):  
                customQueue.enqueue(root.value.rightchild)  
    return "Not found"
```

```
print (searchBT (newBT, "Tea"))
```

```
print (searchBT (newBT, "Cola"))
```

Success

Not found

* TC $\rightarrow O(n)$

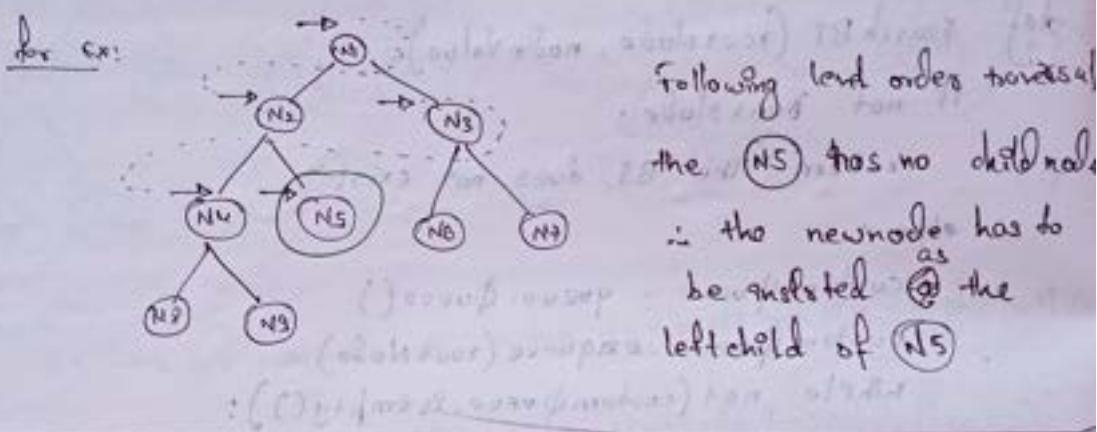
* SC $\rightarrow O(n)$

④ Insert a node in B.T.

if tree is tree with root
empty or null then create new place & insert node into it

2 cases

- If A node Root node is blank.
- The tree exists & we have to look for a first vacant place.



```

def InsertNodeBT (rootNode, newNode):
    if not rootNode:
        rootNode = newNode
    else:
        customQueue = queue.Queue()
        customQueue.enqueue(rootNode)
        while not (customQueue.isEmpty()):
            root = customQueue.dequeue()
            if root.value.leftchild is not None:
                customQueue.enqueue(root.value.leftchild)
            else:
                root.value.leftchild = newNode
                print ("Successfully inserted")
            if root.value.rightchild is not None:
                customQueue.enqueue(root.value.rightchild)
            else:
                root.value.rightchild = newNode
                print ("Successfully inserted")
    
```

```
newNode = TreeNode("cola")
```

```
print(InsertNodeBT(newBT, newNode))
```

```
LevelOrder Traversal(newBT)
```

Drinks

hot

cold

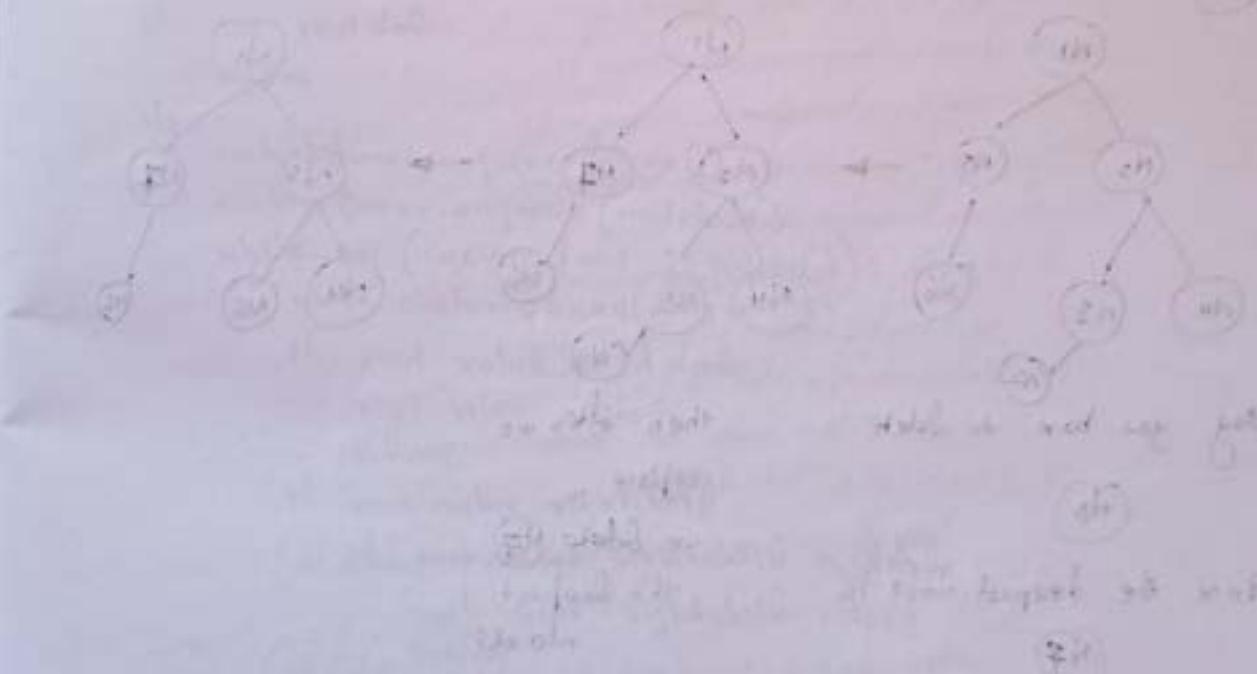
Tea

Coffee

Cola

★ TC $\rightarrow \Theta(n)$

★ SC $\rightarrow \Theta(n)$



() Delete a node from Binary Tree

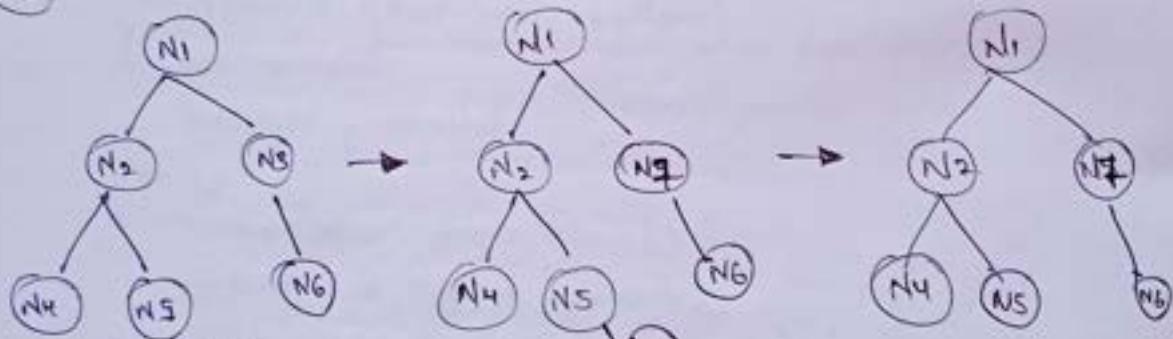
② Cases → the value that we want to delete doesn't exist in B.T
 → the value exist in the B.T

- ① 1st find the deepest node
- ② Then replace the node that you want to delete with the deepest node
- ③ And then delete the deepest node.

*) We cannot directly delete the node

∴ there are other nodes depending on the that node

e.g.



Say you have to delete

N3

Since the deepest node is

N6

then after we delete,

we delete N7
the deepest
node

Get deepest Node:

```
def getDeepestNode(rootNode):
    if not rootNode:
        return
    else:
        customQueue = queue.Queue()
        customQueue.enqueue(rootNode)
        while not (customQueue.isEmpty()):
            root = customQueue.dequeue()
            if (root.value.leftchild is not None):
                customQueue.enqueue(root.value.leftchild)
            if (root.value.rightchild is not None):
                customQueue.enqueue(root.value.rightchild)
        deepestNode = root.value
    return deepestNode
```

To delete deepest Node:

```
def deleteDeepestNode(rootNode, dNode):
    if not rootNode:
        return
    else:
        customQueue = queue.Queue()
        customQueue.enqueue(rootNode)
        while not (customQueue.isEmpty()):
            root = customQueue.dequeue()
            if root.value == dNode:
                root.value = None
                return
            if root.value.leftchild:
                if root.value.leftchild == dNode:
                    root.value.leftchild = None
                else:
                    customQueue.enqueue(root.value.leftchild)
```

```

if root.value.leftchild:
    if root.value.leftchild is DNode:
        root.value.leftchild = None
    return
else:
    customQueue.enqueue(root.value.leftchild)

```

~~# To check whether these 3 method works:~~

```

# newNode = getDeepestNode(newBT)
# deleteDeepestNode(newBT, newNode)
# levelOrderTraversal(newBT)

```

~~# deleting the node:~~

```

def deleteNodeBT(rootNode, node):
    if not rootNode:
        return "The BT does not exist"
    else:
        CustomQueue = queue.Queue()
        CustomQueue.enqueue(rootNode)
        while not (CustomQueue.isEmpty()):
            root = CustomQueue.dequeue()
            if root.value.data == node:
                dNode = getDeepestNode(rootNode)
                root.value.data = dNode.data
                deleteDeepestNode(rootNode, dNode)
                return "The node has been successfully deleted"
            if (root.value.leftchild is not None):
                CustomQueue.enqueue(root.value.leftchild)
            if (root.value.rightchild is not None):
                CustomQueue.enqueue(root.value.rightchild)
        return "Failed to delete"

```

$Tc \rightarrow O(n)$
 $Sc \rightarrow O(n)$

```

deleteNodeBT(newBT, 'flat')
levelOrderTraversal(newBT)

```

Delete entire Binary Tree

```
def deleteBT (rootNode):  
    rootNode.data = None  
    rootNode.leftchild = None  
    rootNode.rightchild = None  
    return "B.T has been successfully deleted"
```

```
deleteBT (newBT)  
levelorderTraversal (newBT)
```

None. (\Rightarrow There's no node in the B.T to traverse)

$$FC \rightarrow O(1)$$
$$SC \rightarrow O(1)$$

BT using LIST

fixed size

Creation of BT using LIST:

Class BinangTree :

```
def __init__(self, size):
    self.customlist = size * [None]
    self.lastusedIndex = 0
    self.maxsize = size
```

```
newBT = BinangTree(8)
```

wouldn't give you anything ↴ you haven't performed any method.

TC - O(1)

SC - O(n)

Insertion of a Node

check in
YouTube
he hasn't done

```
def insertNode(self, value):
    if self.lastusedIndex + 1 == self.mysize:
        return "The Binary Tree is full."
    else:
        self.customlist[self.lastusedIndex + 1] = value
        self.lastusedIndex += 1
        return "The value has been inserted"
```

Searching a Node

```
def searchNode (self, nodeValue):  
    for i in range (self.customList):  
        if self.customList [i] == nodeValue:  
            return "Success"  
    return "Not found"
```

TC $\rightarrow \Theta(n)$
SC $\rightarrow O(1)$

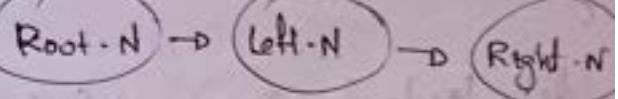
```
newBT = BinaryTree (8)  
print (newBT.insertNode ("Oranges"))  
print (newBT.insertNode ("Mangoes"))  
print (newBT.insertNode ("Golad"))  
print (newBT.searchNode ("Mangoes"))
```

The value has been successfully searched

Success

It doesn't matter that it's Python list or L-L
the algorithm of traversals remains the same.

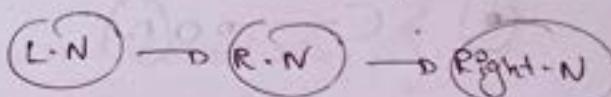
PreOrder traversal



```
def preOrderTraversal(self, index):  
    if index > self.lastUsedIndex: --- O(1)  
        return  
    print(self.customList[index])  
    self.preOrderTraversal(index * 2) --- O(n/2)  
    self.preOrderTraversal(index * 2 + 1) --- O(n/2)
```

TC → O(n)
SC → O(n)

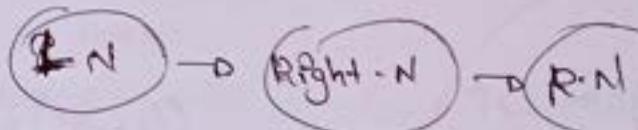
InOrder traversal



```
def InOrderTraversal(self, index):  
    if index > self.lastUsedIndex:  
        return  
    InOrderTraversal(index * 2)  
    print(self.customList[index])  
    InOrderTraversal(index * 2 + 1)
```

TC → O(n)
SC → O(n)

Post Order traversal



```
def PostOrderTraversal(self, index):  
    if index > self.lastUsedIndex:  
        return  
    PostOrderTraversal(index * 2 + 1)  
    print(self.customList[index])  
    PostOrderTraversal(index * 2)  
    print(self.customList[index])
```

TC → O(n)
SC → O(n)

LevelOrder Traversal

```
def levelorder traversal(self, index):
```

```
    for i in range(index, self.lastUsedIndex+1):  
        print(self.customList[i])
```

TC - $O(n)$

SC - $O(n)$

Delete a node from BT

- ① Find the deepest Node = lastUsedIndex
- ② Replace the node that you want to delete by deepest node.
- ③ Delete the deepest Node

```
def deleteNode(self, value):  
    if self.lastUsedIndex == 0:  
        return "There is no node to delete"  
    for i in range(1, self.lastUsedIndex + 1):  
        if self.customList[i] == self.customList[self.lastUsedIndex]:  
            value :  
                self.customList[i] = self.customList[self.lastUsedIndex]  
                self.customList[self.lastUsedIndex] = None  
                self.lastUsedIndex -= 1  
    return "The node is deleted"
```

```
print(newBT.deleteNode("Kot"))  
newBT.levelOrderTraversal()
```

TC $\rightarrow O(n)$
SC $\rightarrow O(1)$

Delete entire BT

TC and SC o(n)

```
def deleteBT(self):  
    self.customlist = None  
    return "The BT has deleted"
```

```
print(newBT.deleteBT())
```

```
newBT.levelorderTraversal()
```

SC → O(1)
TC → O(1)

The BT has deleted

Create (\because there's nothing to handle)

T & S complexity

	Python list with capacity		L.L	
	Time-store	S.S	T.S	S.S
Create B.T	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Insert node in B.T	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Delete node in B.T	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Search node in B.T	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Traverse B.T	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Delete B.T	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Space efficient?		No		Yes

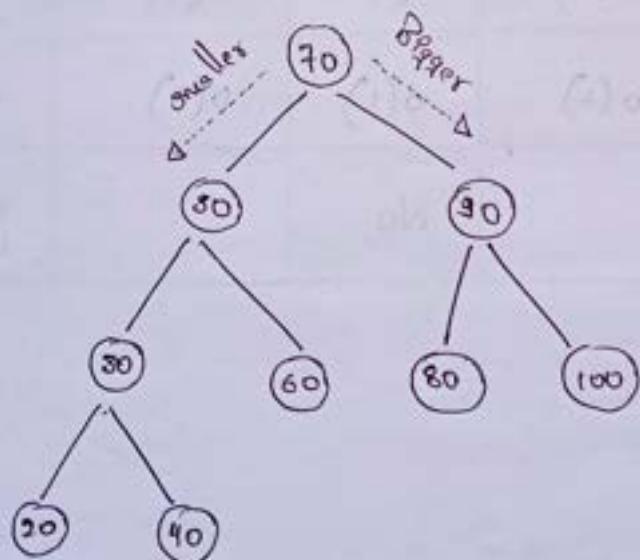


BINARY SEARCH TREE

[Here will use only LL, if u want to implement using left
u can try again]

Basic Property

- ① In the left subtree : the value of node is less than or equal to its parent node's value
- ② In the right subtree : the value of a node is greater than its parent node's value



BST is faster than BT? ← Read out

Methods that can be performed on a B.S.T

- * Creation of Tree
- * Insertion of a Node
- * Deletion of a Node
- * search for a value
- * Traverse all Nodes
- * Deletion of tree

Creation of a BST

class BSTNode:

```
def __init__(self, data):  
    self.data = data  
    self.leftchild = None  
    self.rightchild = None
```

```
newBST = BSTNode(None)
```

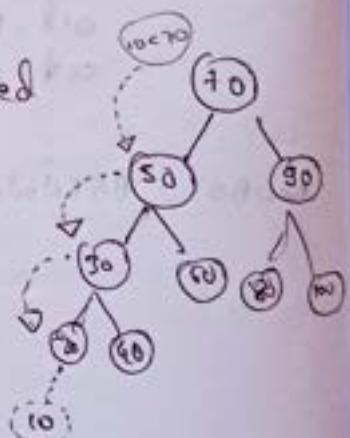
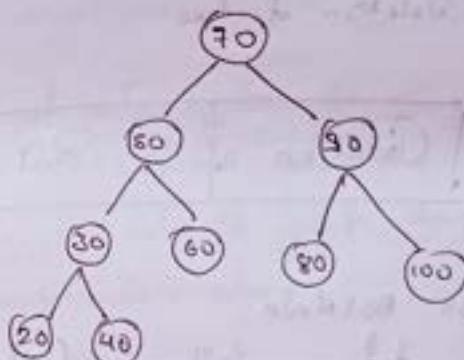
- ★ TC → O(1)
- ★ SC → O(1)

Insert a Node to BST

- ② cases:
- The rootnode is blank
i.e. the tree doesn't have any nodes in it.
 - BST has nodes in it

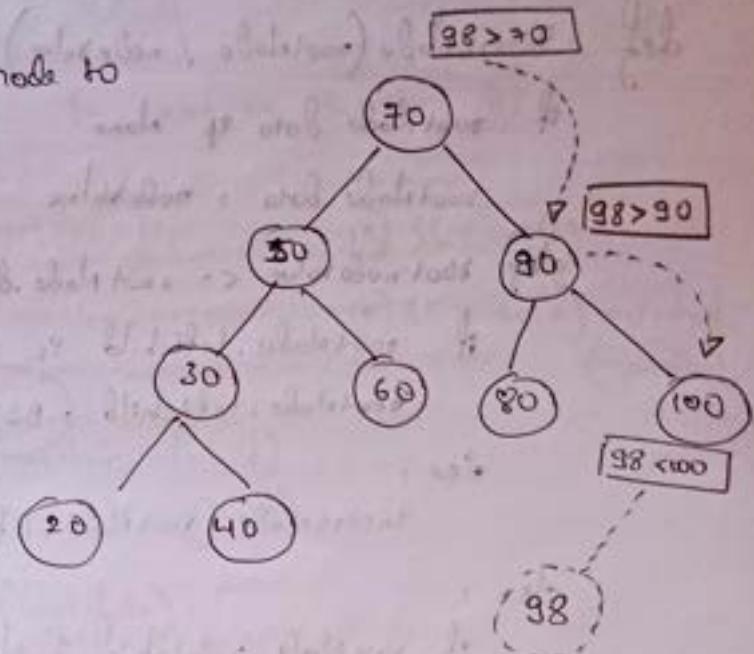
In the 2nd case:

- ① let's say 10 is the node to be inserted, to the tree →
- ② In BST while inserting a node only 1 half is traversed
→ this makes it faster than BT
- ③ since $10 < 70$ ∴ left subtree is searched
- ④ Now 10 is again < 50 ∴ left subtree of node 50 is searched
- ⑤ Now $10 < 20$
- ⑥ $10 < 20$ ∴ it is inserted \oplus as the left subnode of RootNode 20



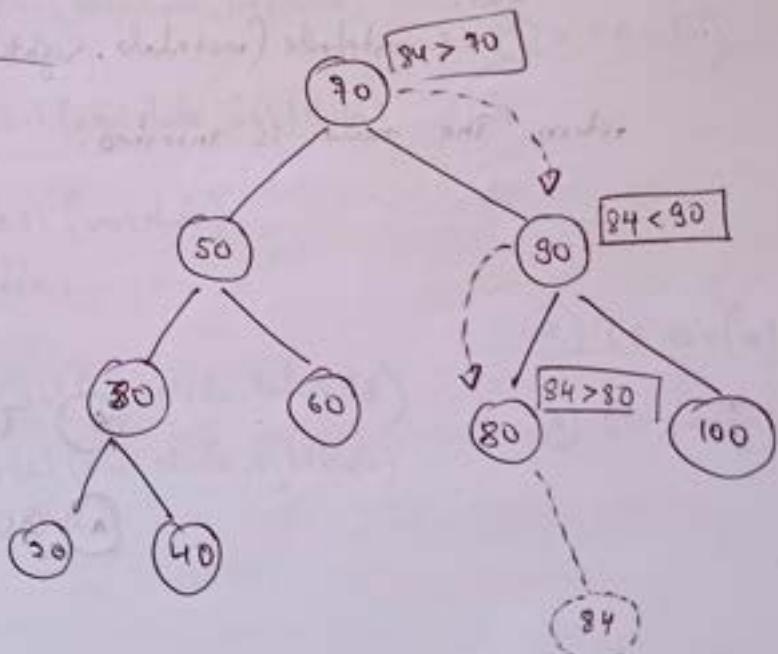
Example 2

- Say 98 is the node to be inserted



Example 3

- Say 84



[6 / 14]

```

def insertNode(rootNode, nodeValue):
    if rootNode.data == None:
        rootNode.data = nodeValue
    elif rootNode.data <= nodeValue:
        if rootNode.leftchild == None:
            rootNode.leftchild = BSTNode(nodeValue)
        else:
            insertNode(rootNode.leftchild, nodeValue)
    else:
        if rootNode.rightchild == None:
            rootNode.rightchild = BSTNode(nodeValue)
        else:
            insertNode(rootNode.rightchild, nodeValue)
    return "The node is inserted"

```

$O(\log n)$

* TC $\rightarrow O(\log n)$

* SC $\rightarrow O(\log n)$

Traversal in BST

The traversal remains the same as in the BT

```
def PreorderTraversal (rootNode):    TC -> O(n)
    if not rootNode:
        return
    print (rootNode.data)           SC -> O(n) - O stack
    preOrderTraversal (rootNode.leftchild)      (Execution)
    preOrderTraversal (rootNode.rightchild)
```



```
def InorderTraversal (rootNode):       TC -> O(n)
    if not rootNode:
        return
    InorderTraversal (rootNode.leftchild)      SC -> O(n)
    print (rootNode.data)
    InorderTraversal (rootNode.rightchild)
```



```
def PostorderTraversal (rootNode):      TC -> O(n)
    if not rootNode:
        return
    PostorderTraversal (rootNode.leftchild)     SC -> O(n)
    PostorderTraversal (rootNode.rightchild)
    print (rootNode.data)
```

```
def levelorderTraversal (rootNode):  
    if not rootNode:  
        return  
  
    else:  
        customQueue = queue.Queue()  
        customQueue.enqueue (rootNode)  
        while not (customQueue.isEmpty()):  
            root = customQueue.dequeue()  
            print (root.value.data)  
            if root.value.leftchild is not None:  
                customQueue.enqueue (root.value.leftchild)  
            if root.value.rightchild is not None:  
                customQueue.enqueue (root.value.rightchild)
```

★ LC → O(n)

★ SC → O(n)

Search for a Node

```

def searchNode(rootNode, nodeValue):
    if rootNode.data == nodeValue:
        print("The value is found")
    elif nodeValue < rootNode.data:
        if rootNode.leftchild.data == nodeValue:
            print("The value is found")
        else:
            searchNode(rootNode.leftchild, nodeValue)
    else:
        if rootNode.rightchild.data == nodeValue:
            print("The value is found")
        else:
            searchNode(rootNode.rightchild, nodeValue)

```

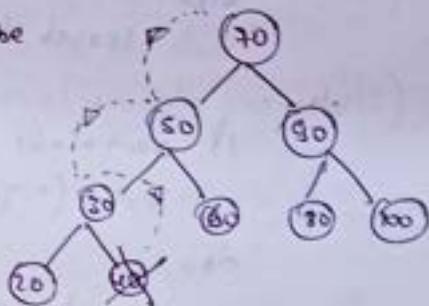
$$\star \text{ TC} \longrightarrow O(\log N)$$

$$\textcircled{F} \text{ SC} \longrightarrow O(\log N)$$

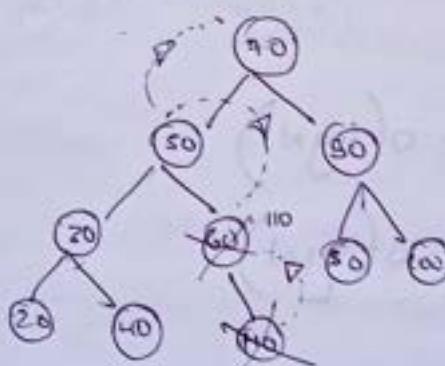
Delete a node from BST

- Case ③
- The node to be deleted is a leaf node
 - The node has 1 child
 - The node has 2 children

Case 1 say 40 is the leaf node to be deleted

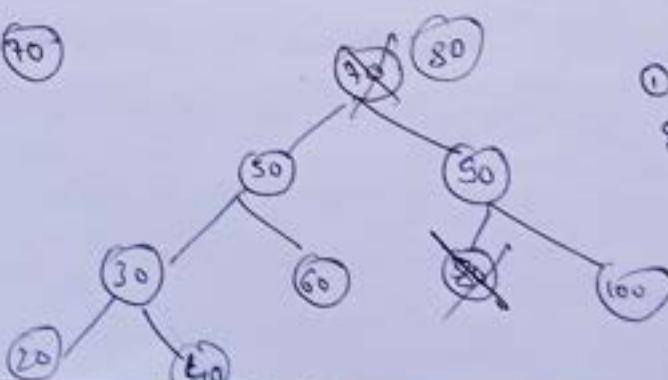


Case 2 say 60

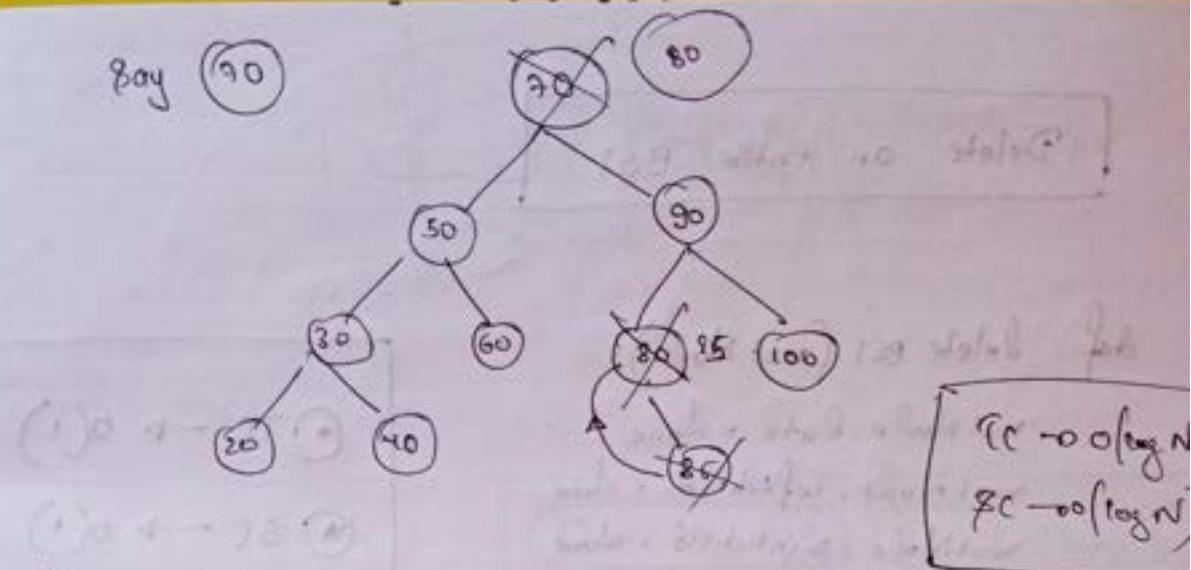


a lineage till
60 is then replace
by 110

Case 3 say 70



① is how to find the
successor of 70
i.e. the smallest node
in Right subtree
& replace it.



```

def minValueNode(bstNode):
    current = bstNode
    while (current.leftchild is not None):
        current = current.leftchild
    return current

def deleteNode(rootNode, nodeValue):
    if rootNode is None:
        return rootNode
    if nodeValue < rootNode.data:
        rootNode.leftchild = deleteNode(rootNode.leftchild, nodeValue)
    elif nodeValue > rootNode.data:
        rootNode.rightchild = deleteNode(rootNode.rightchild, nodeValue)
    else:
        if rootNode.leftchild is None:
            temp = rootNode.rightchild
            rootNode = None
            return temp
        if rootNode.rightchild is None:
            temp = rootNode.leftchild
            rootNode = None
            return temp
        temp = minValueNode(rootNode.rightchild)
        rootNode.data = temp.data
        rootNode.rightchild = deleteNode(rootNode.rightchild, temp.data)
    return rootNode

```

Delete an entire BST

```
def delete_BST(rootNode):  
    rootNode.data = None  
    rootNode.leftchild = None  
    rootNode.rightchild = None  
    return "Successfully Deleted"
```

TC $\rightarrow O(1)$

SC $\rightarrow O(1)$

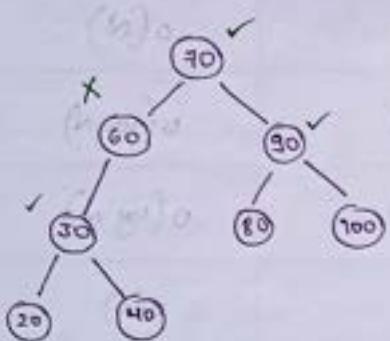
T & S - Complexity

	TIME-STONE	SPACE-STONE
Create BST	$O(1)$	$O(1)$
Insert a node in BST	$O(\log N)$	$O(\log N)$
Traverse BST	$O(N)$	$O(N)$
Search for a node in BST	$O(\log N)$	$O(\log N)$
Delete node from BST	$O(\log N)$	$O(\log N)$
Delete Entire BST	$O(1)$	$O(1)$



- An AVL tree is a SELF-BALANCING BST where the difference in heights of left & right subtrees cannot be more than 1, i.e., at nodes (except leaf nodes).

(Illustration)



$$\begin{aligned} \checkmark 70 &\rightarrow \text{Height of LST} = 3 \\ &\quad \text{Height of RST} = 2 \end{aligned} \left\{ \begin{array}{l} 3 - 2 = 1 \\ \text{H of BST} = 3 \end{array} \right. \quad ①$$

$$\begin{aligned} \checkmark 60 &\rightarrow \text{H of LST} = 2 \\ &\quad \text{H of RST} = 0 \end{aligned} \left\{ \begin{array}{l} 2 - 0 = 2 \\ \text{H of BST} = 2 \end{array} \right. \quad ②$$

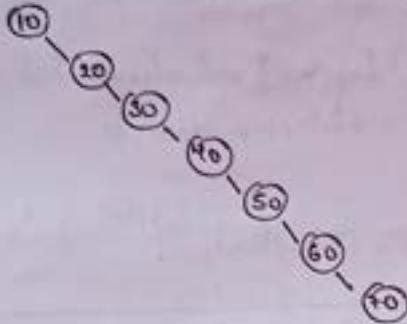
$$\begin{aligned} \checkmark 30 &\rightarrow \text{H of LST} = 1 \\ &\quad \text{H of RST} = 1 \end{aligned} \left\{ \begin{array}{l} 1 - 1 = 0 \\ \text{H of BST} = 1 \end{array} \right. \quad ③$$

$$\begin{aligned} \checkmark 80 &\rightarrow \text{H of LST} = 1 \\ &\quad \text{H of RST} = 1 \end{aligned} \left\{ \begin{array}{l} 1 - 1 = 0 \\ \text{H of BST} = 1 \end{array} \right. \quad ④$$

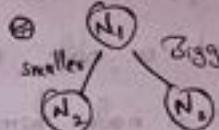
- If at any time heights of left & right subtrees differ by more than 1, then REBALANCING is done to restore AVL PROPERTY & the concept is called ROTATION

Why do we need AVL Tree?

Say u want to insert these no. to a BST - 10, 20, 30, 40, 50, 60, 70

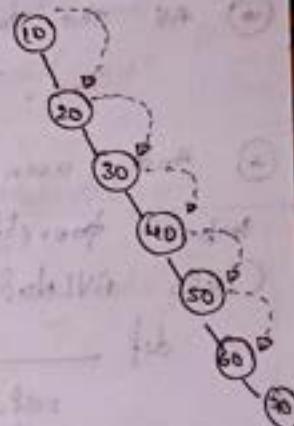


∴ In BST



Now if u want to search 60 in this, then u hv to traverse right from the top

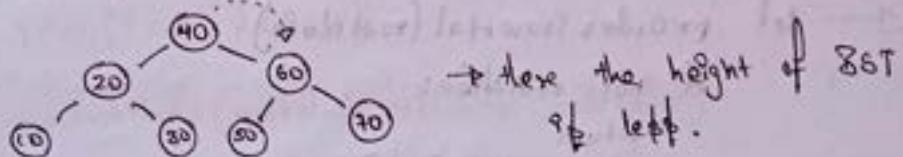
$$\Rightarrow \text{TC} \rightarrow O(n)$$



But the TC of a BST $\rightarrow O(\log n)$

This forces if the height of BST is too much.

If the same BT is balanced then it will look like



If u want to search 60 then it's just 1 step.

$$\{\text{TC} \rightarrow O(\log n)\}$$

- ★ AVL tree makes sure that we never end up with this kind of unbalanced tree, AVL tree enforces some rules when we are inserting or deleting nodes from BST to make sure that it would never be unbalanced tree. [P.e uses ROTATION]

Common operations on AVL Trees

- Creation of AVL Trees
- Search for a node in AVL Tree
- Traverse all nodes in AVL Tree
- Insert a node in AVL tree
- Delete a node from AVL tree
- Delete the entire AVL tree

① All the traversal are as same of as the BST:

No. Post, In & level orders traversal.

② And even creation & searching

Roots queue-like list as queue

Class AVLNode:

```
def __init__(self, data):
```

```
    self.data = data
    self.leftchild = None
    self.rightchild = None
    self.height = 1
```

```
def preOrderTraversal(rootNode):
```

```
    if not rootNode:
```

```
        return
```

```
    print(rootNode.data)
```

```
    preOrderTraversal(rootNode.leftchild)
```

```
    preOrderTraversal(rootNode.rightchild)
```

```
def inOrderTraversal(rootNode):
```

```
    if not rootNode:
```

```
        return
```

```
    inOrderTraversal(rootNode.leftchild)
```

```
    print(rootNode.data)
```

```
    inOrderTraversal(rootNode.rightchild)
```

```
def postOrderTraversal (rootNode):
```

```
if not rootNode:
```

```
return
```

```
postOrderTraversal (rootNode.leftchild)
```

```
postOrderTraversal (rootNode.rightchild)
```

```
print (rootNode.data)
```

```
def levelOrderTraversal (rootNode):
```

```
if not rootNode:
```

```
return
```

```
else:
```

```
customqueue = queue.Queue()
```

```
customqueue.enqueue (rootNode)
```

```
while not (customqueue.isempty ()): O(n)
```

```
root = customqueue.dequeue()
```

```
print (root.value.data)
```

```
if root.value.leftchild is not None:
```

```
customqueue.enqueue (root.value.leftchild)
```

```
if root.value.rightchild is not None:
```

```
customqueue.enqueue (root.value.rightchild)
```

```
def searchNode (rootNode, nodeValue):
```

```
if rootNode.data == nodeValue:
```

```
print ("Value found")
```

```
if nodeValue < rootNode.data:
```

```
if rootNode.leftchild.data == nodeValue:
```

```
print ("Value found")
```

```
else:
```

```
searchNode (rootNode.leftchild, nodeValue)
```

```
else:
```

```
if rootNode.rightchild.data == nodeValue:
```

```
print ("Value found")
```

```
else:
```

```
searchNode (rootNode.rightchild, nodeValue)
```

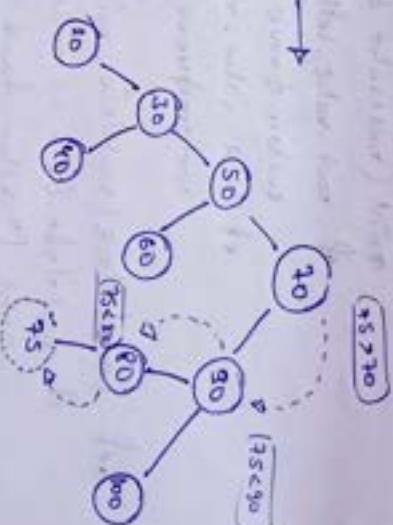
Implementation on AVL

- ② Cases
 → Rotation is not required
 → Rotation is required

- ★ Rotation is required only when a BST is disbalanced.

- ★ When the rotation is not required
CASE 1
insertion is same as that in BST

say u want to insert 73 →



- ★ When Rotation is Required

CASE 2

There are 4 condition where rotation is required

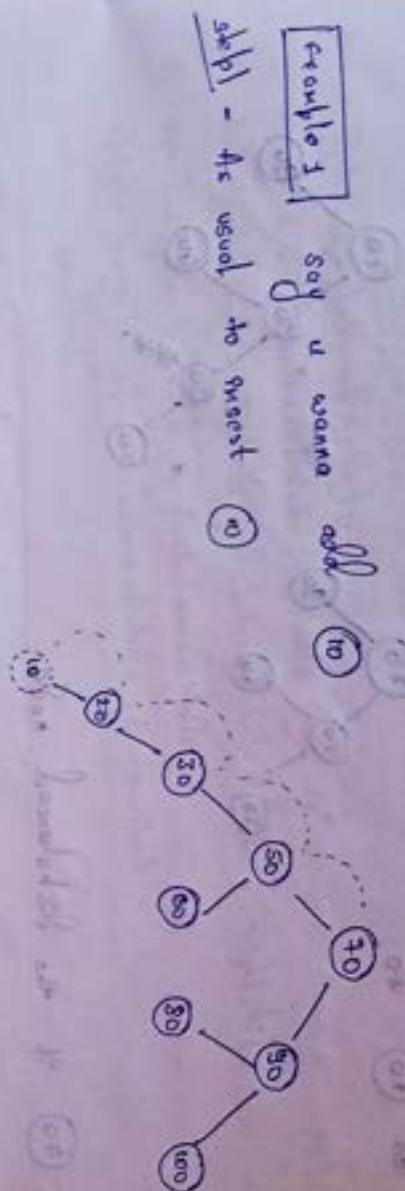
- ① LL - left left condition
- ② LR - left right condition
- ③ RR - right right condition
- ④ RL - Right left condition

① LL-condition

Example 1:

say u wanna add 10

step 1: - As usual do insert 10



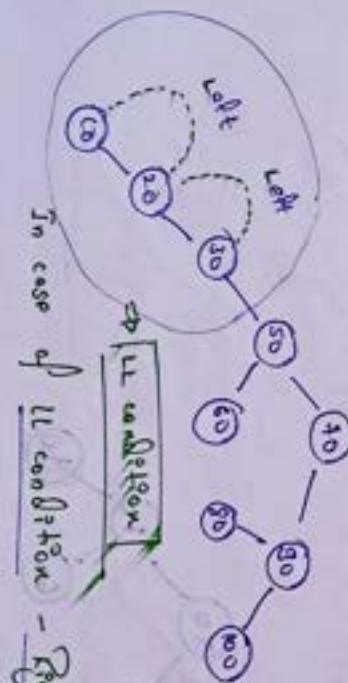
step 2: - Find the node causing imbalance

- start from leaf node

- Here 10 is causing imbalance.

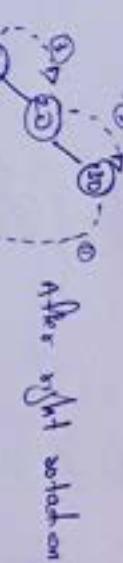
step 3: - Find the grand child of imbalanced node

- The path from imbalanced node and its grandchild which causes the imbalance - gives u the condition.

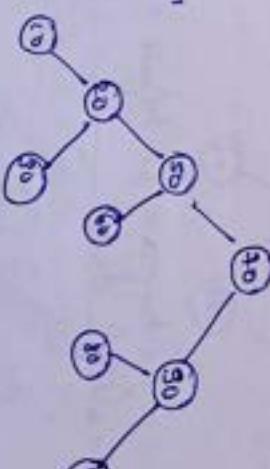


In case of LL condition - Right rotation is required

step 4: -



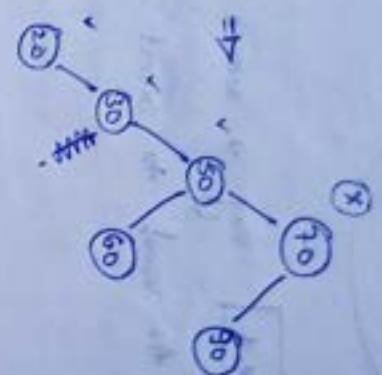
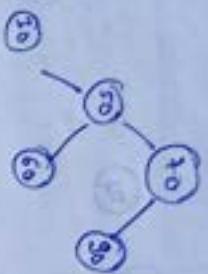
After right rotation



{ Now skip 8 & All tree }

Example - 2

all $\{10, 40\}$



$\{50\}$ is the balanced node

$\{40\}$ has 2 grand child $\{50\} \notin \{60\}$

- but the 1 hop greater height is to be chosen Pre $\{10\}$

\therefore
 $\begin{array}{c} 10 \\ | \\ 50 \\ / \backslash \\ 20 \quad 60 \\ | \quad | \\ 50 \quad 40 \\ | \quad | \\ 60 \quad 70 \end{array}$ \Rightarrow Right rotation.

Algorithm of LL rotation:

rotateRight (disbalanceNode):

- ① newRoot = disbalanceNode . leftchild
- ② disbalanceNode . leftchild . disbalanceNode . leftchild . rightchild
- ③ newRoot . rightchild = disbalanceNode
- ④ update height of disbalanceNode & newRoot
- return newRoot.

how the algo. works?

say this is the
disbalanced node

① newRoot = newRoot

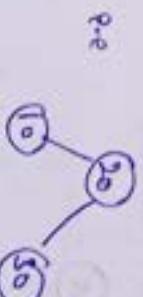
disbalanceNode . LC . RC

② None = disbalanceNode

③ disbalanceNode = ②

④

newRoot . right child = ③



height of newRoot = 1

disbalanceNode

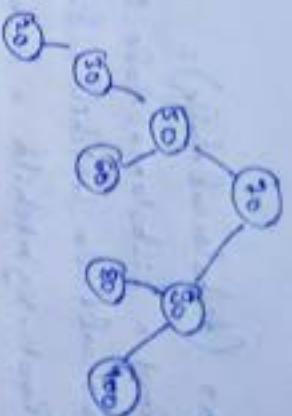
① LR → ② O(1) ② SC → O(1)

(2) LR-rotation

① try a swap to insert 25 to

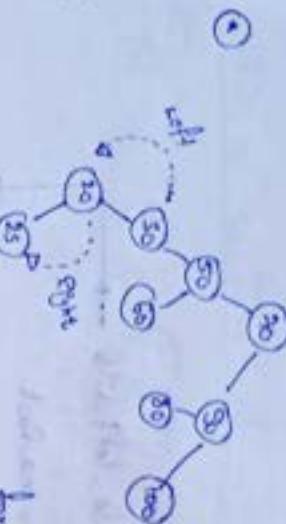
middle-right child of 10
left child of 10 has right child

bottom 3 nodes are moved to bottom right



new tree 10 8 40

disbalanced node



now for LC condition w. 80

left → left rotation
then → Right rotation

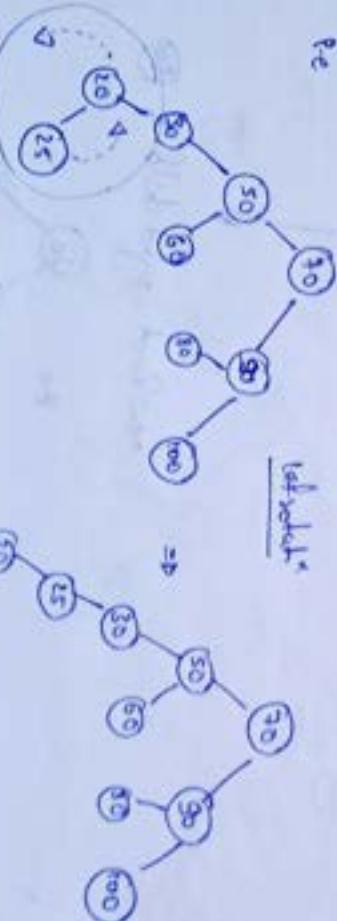
if the left rotation is done to the left child of the
disbalance node (90 here)
→



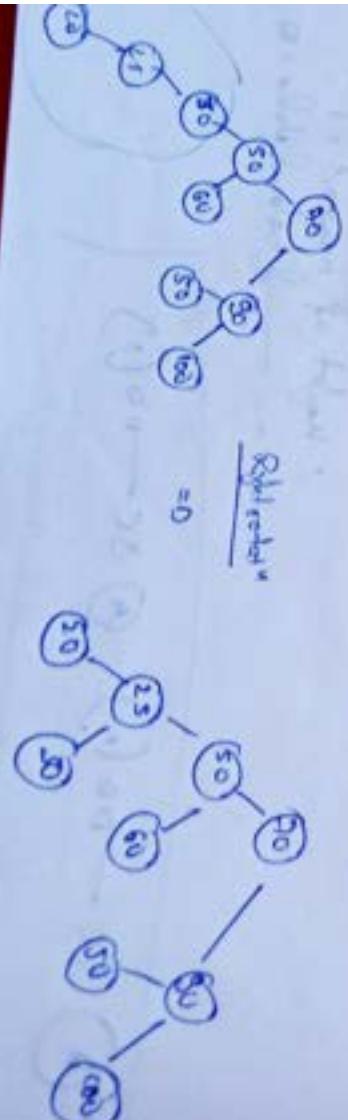
left

right

left



Right rotation



= 0

Left rotation

Algorithm of LF condition:

step1 : isolate left the balanced node. leftchild

step2 : isolate right balanced node

① rotateLeft (balancedNode):

newRoot = balancedNode . rightchild
balancedNode . right child = balancedNode . rightchild . leftchild
newRoot . leftchild = balancedNode
update height of balancedNode & newRoot
return newRoot

② rotateRight (balancedNode):

newRoot = balancedNode . leftchild
balancedNode . left child = balancedNode . leftchild . rightchild
newRoot . right child = balancedNode
update height of balanced Node & newRoot
return newRoot.

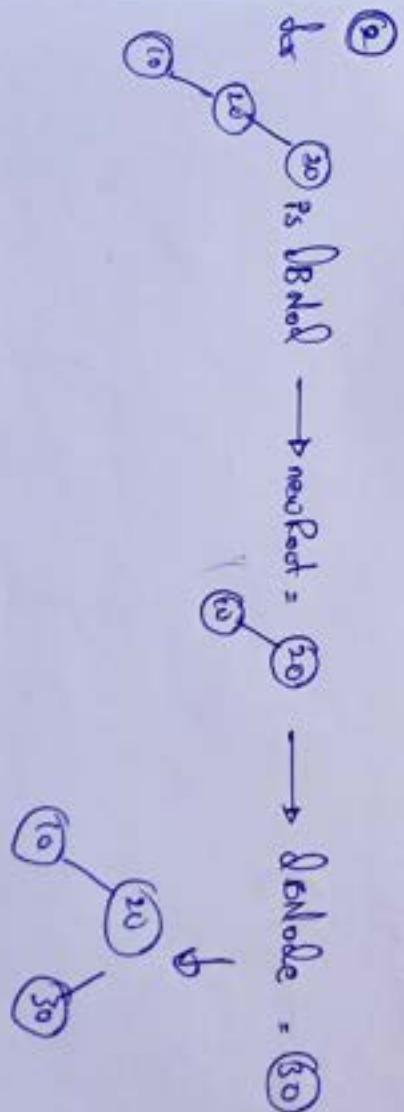
③ if shift 30 q, the balancedNode → newRoot = 20 →



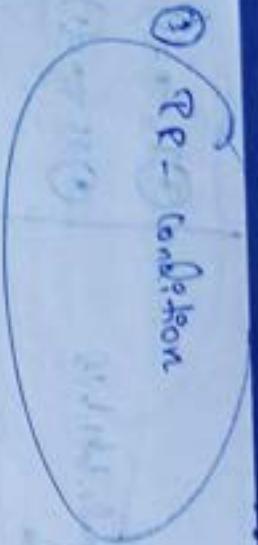
bbNode . RC = None

20 = None

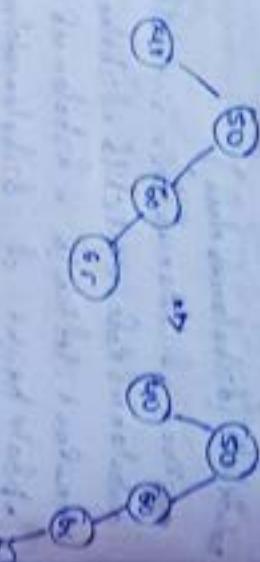
→ bbNode = 10



③ RP - condition

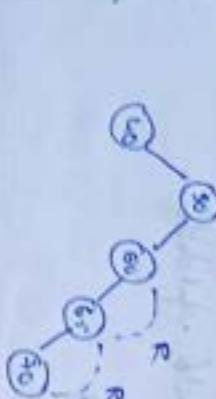


④ Play a song to want ⑤ to do

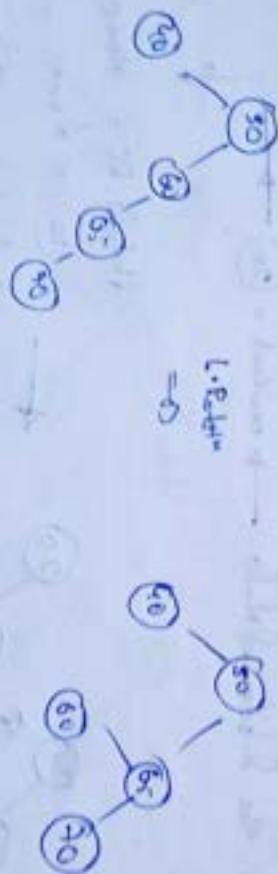


⑥ Here ④ is possible (start from last node)
to clarify the condition - if we pass top song first

for RP-condition → left rotation



★



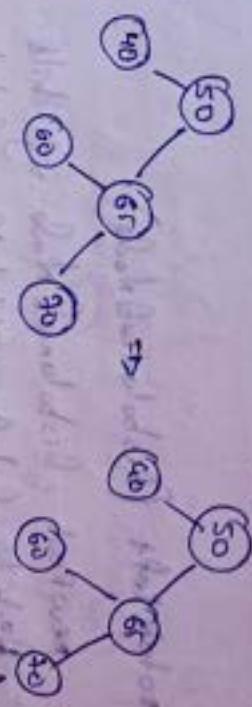
→
left rotation



→
left rotation

Example - 2

④ want to insert ⑤ to ④



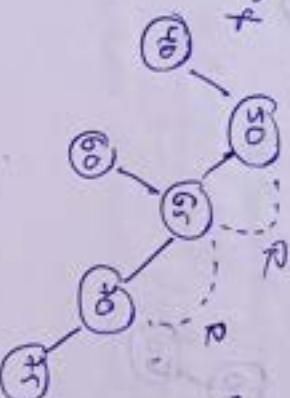
Left child, right child, grandchild, etc., are available.

⑤ or Denode (from the leaf-node 57)

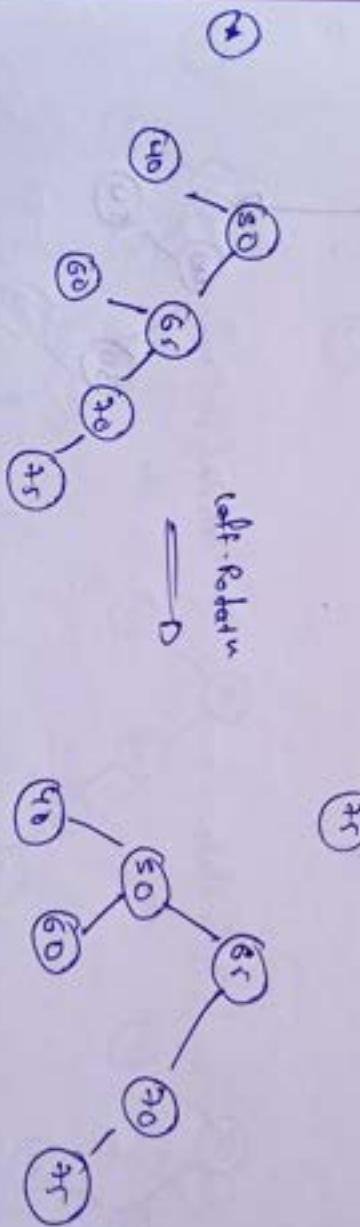
⑥ find the grand child

- there are 2 grandchild 60 & 60
- put 60 to choose ④ if it's height is greater

⑦ the Path to ③ of ④



→ pre-config: tree



→ left-right

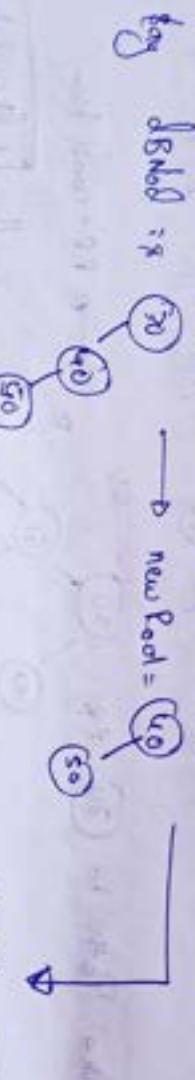
Algorithm

rotateLeft (disbalancedNode):

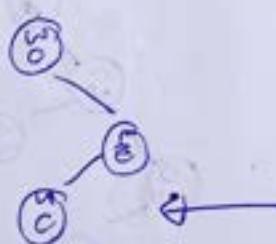
newRoot = Disbalanced Node . right child
Disbalanced Node . right child = Disbalanced Node . right child . left child

newRoot . leftchild = Disbalanced Node
update height of Disbalanced Node & newRoot

return \oplus newRoot



DBNdo = 30



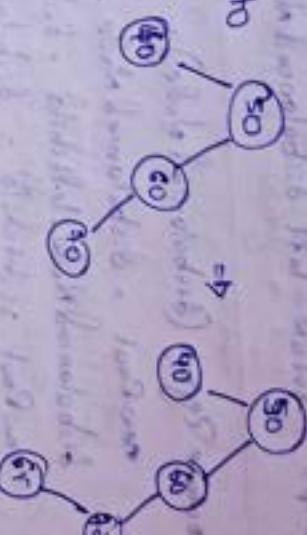
④ TC → O(1)

④ SC → O(1)

(4) RL-condition

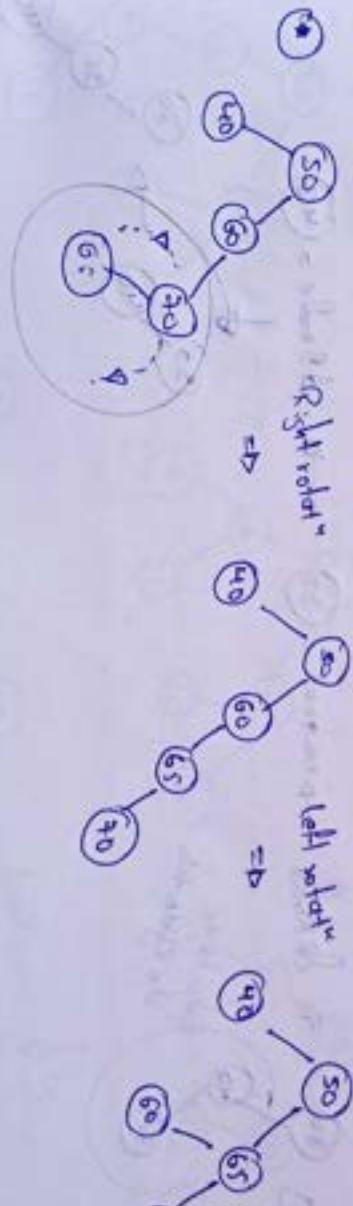
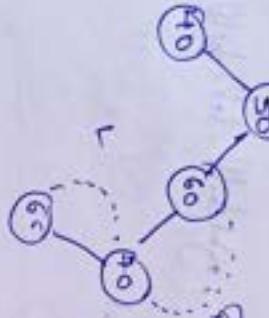
right child

- * try to insert 65 to



- * 65 is the RB node
- * and the path to its grand child i.e. 65

RL condition \rightarrow Right rotate the right child of a Ba
then \rightarrow left rotation.



Algorithm

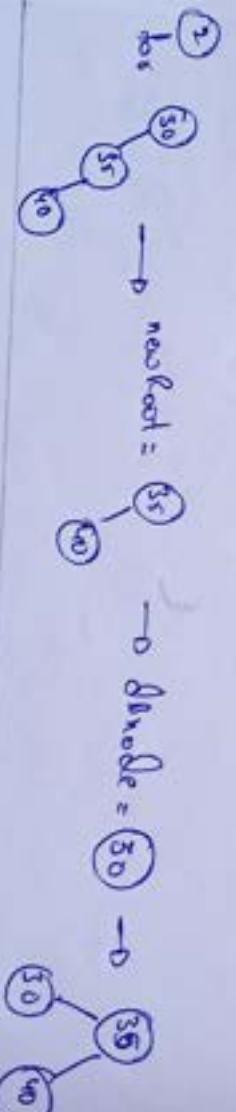
step1 : rotate Right *disbalancedNode*.rightchild
 step2 : rotate left *disbalancedNode*

rotateRight (disbalancedNode):

newRoot = *disbalancedNode*.leftchild
disbalancedNode.leftchild = *disbalancedNode*.leftchild.leftchild
newRoot.rightchild = *disbalancedNode*
 update height of *disbalancedNode* & *newRoot*,
 return *newRoot*

rotateLeft (disbalancedNode):

newRoot = *disbalancedNode*.rightchild
disbalancedNode.rightchild = *disbalancedNode*.rightchild.rightchild
newRoot.leftchild = *disbalancedNode*.leftchild
 update height of *disbalancedNode* & *newRoot*
 return *newRoot*.



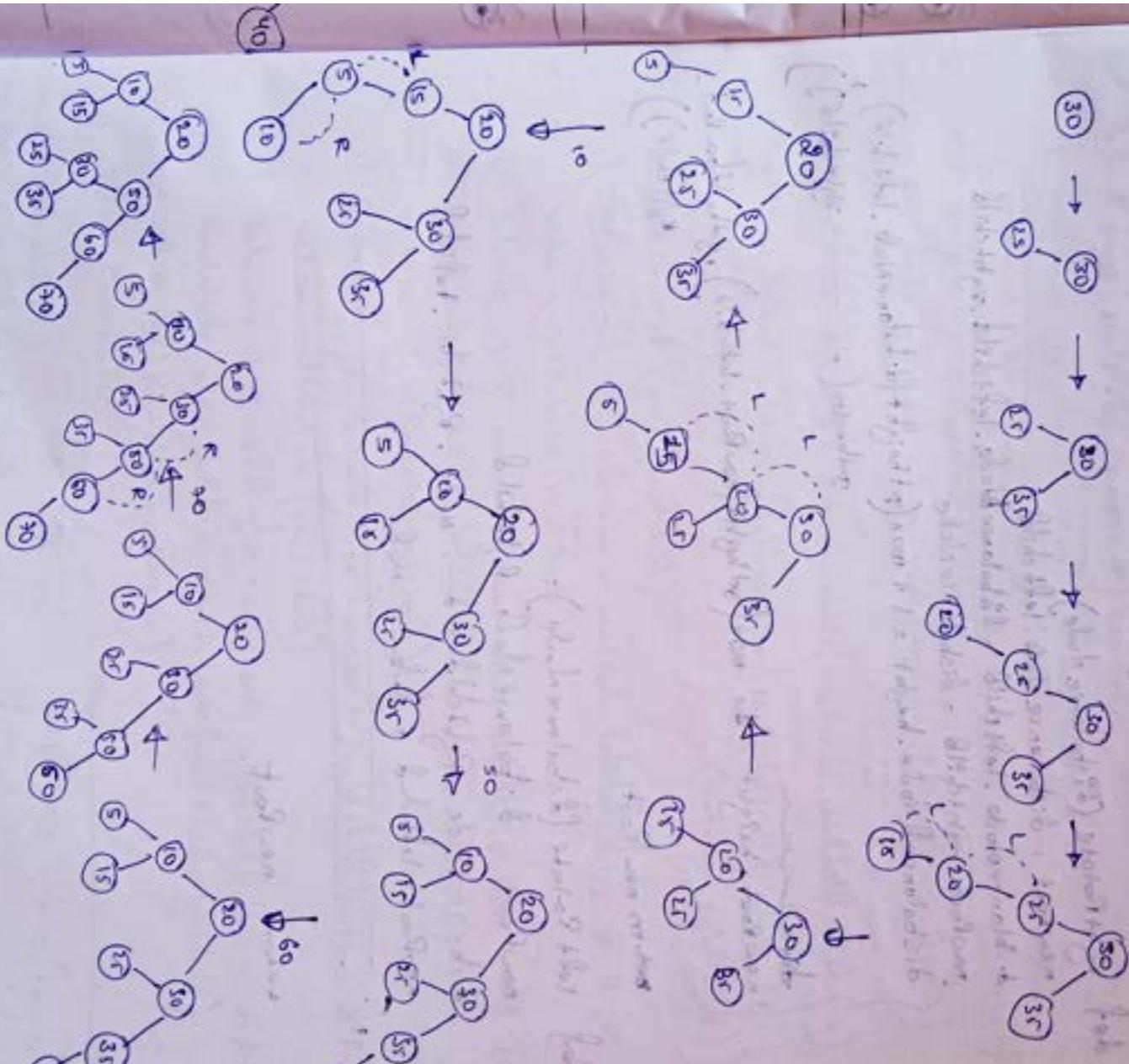
$$\textcircled{1} \quad TC \rightarrow O(1) \quad \textcircled{2} \quad SC \rightarrow O(1)$$

Inversion of node in AVL tree

(all together)

say u wanna add these & using
insertion & deletion

30, 25, 35, 10, 15, 5, 10, 50, 60, 70, 40



Method

```
def getHeight (rootNode):    # helper function  
    if not rootNode:  
        return 0  
    return rootNode.height  
  
def rightRotate (disbalanceNode):  
    newRoot = disbalanceNode.leftChild  
    disbalanceNode.leftChild = disbalanceNode.leftChild.rightChild  
    newRoot.rightChild = disbalanceNode  
  
    disbalanceNode.height = 1 + max(getHeight(disbalanceNode.leftChild),  
                                    getHeight(disbalanceNode.rightChild))  
  
    newRoot.height = 1 + max(getHeight(newRoot.leftChild),  
                            getHeight(newRoot.rightChild))  
  
    return newRoot  
  
def leftRotate (disbalanceNode):  
    newRoot = disbalanceNode.RightChild  
    disbalanceNode.RightChild = disbalanceNode.RightChild.LeftChild  
    newRoot.LeftChild = disbalanceNode  
  
    return newRoot
```

def getBalance (rootNode): # to check if its balanced or not
 if not rootNode:
 return 0

 return getheight (rootNode.leftchild) - getheight (rootNode.rightchild)

def insertNode (rootNode, nodeValue):
 if rootNode is None: # or if not rootNode:
 return AVLNode (nodeValue)
 elif nodeValue < rootNode.data:
 rootNode.leftchild = insertNode (rootNode.leftchild, nodeValue)
 else:
 rootNode.rightchild = insertNode (rootNode.rightchild, nodeValue)
 rootNode.height = 1 + max (getheight (rootNode.leftchild), getheight (rootNode.rightchild))
 if balance = getBalance (rootNode)
 if Balance > 1 & nodeValue < rootNode.leftchild.data: # LL
 return rightRotate (rootNode)
 if Balance > 1 & nodeValue > rootNode.leftchild.data: # LR
 rootNode.leftchild = leftRotate (rootNode.leftchild)
 return rightRotate (rootNode)
 if Balance < -1 & nodeValue > rootNode.rightchild.data: # RR
 return leftRotate (rootNode)
 if Balance < -1 & nodeValue < rootNode.rightchild.data: # RL
 rootNode.rightchild = rightRotate (rootNode.rightchild)
 return leftRotate (rootNode)
 return rootNode

newAVL = AVLNode (5)
newAVL = insertNode (newAVL, 10)
newAVL = insertNode (newAVL, 15)
newAVL = insertNode (newAVL, 20)

TC $\rightarrow O(\log n)$
SC $\rightarrow O(\log n)$

Delete a node from AVL Tree

3 Case

① The tree does not exist

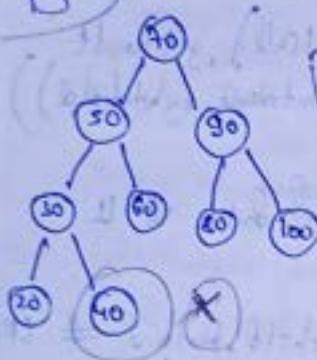
② Rotation is not required

③ Rotation is required (LL, LR, RR, RL)

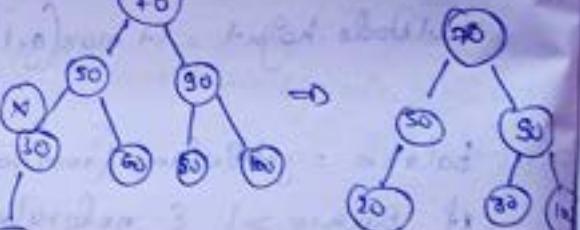
Case 1

The node to be deleted is a leaf Node

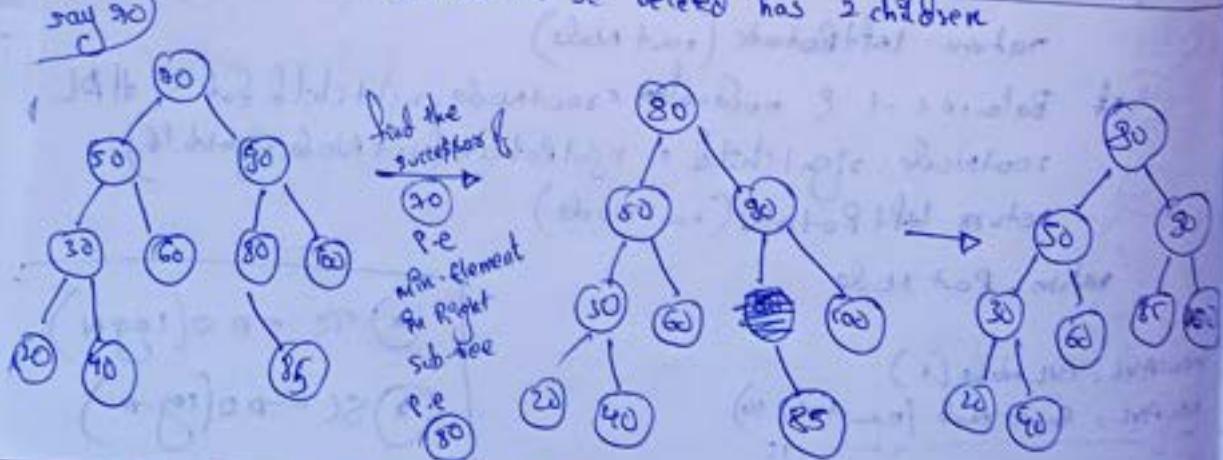
say 40

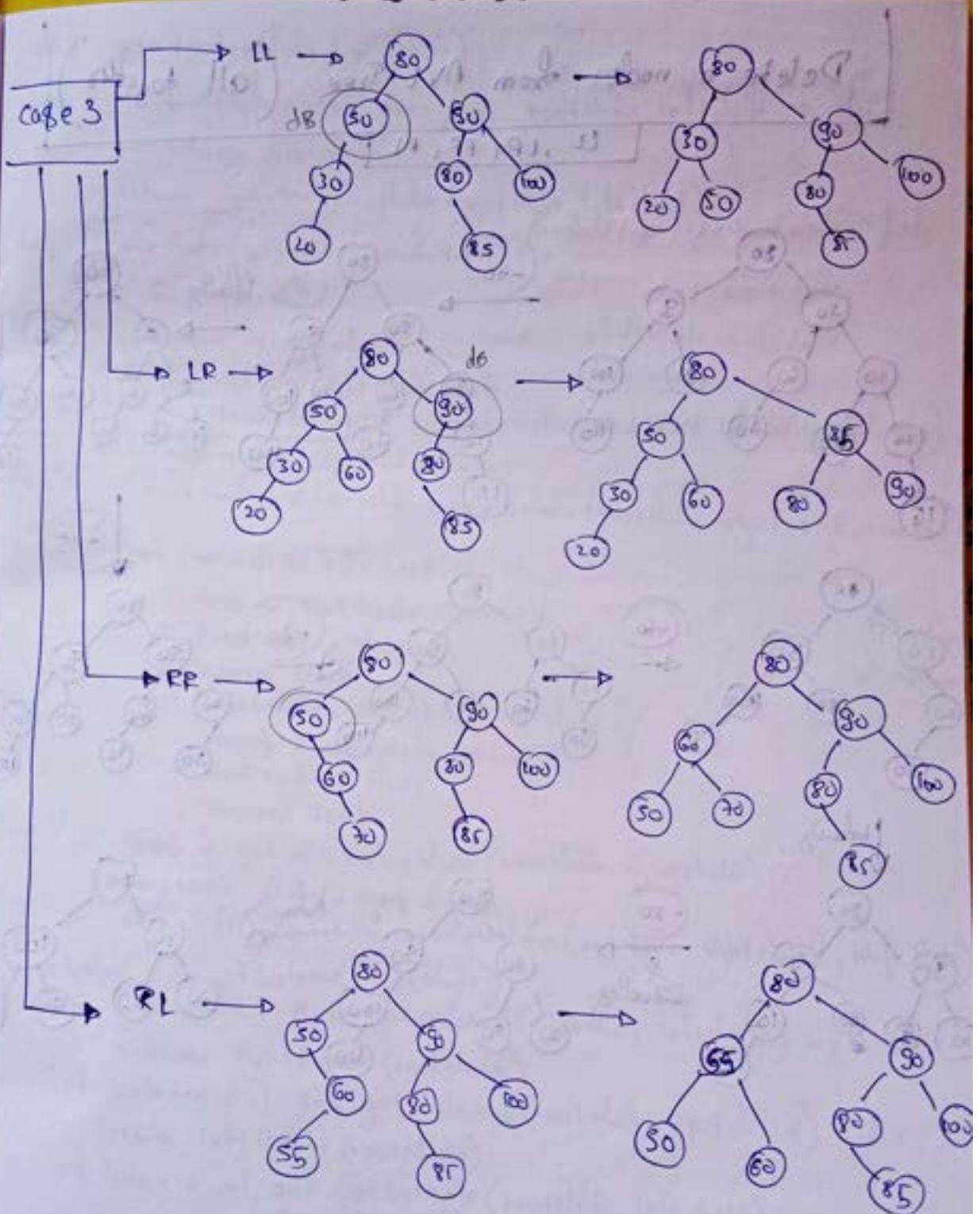


The node to be deleted had a child node



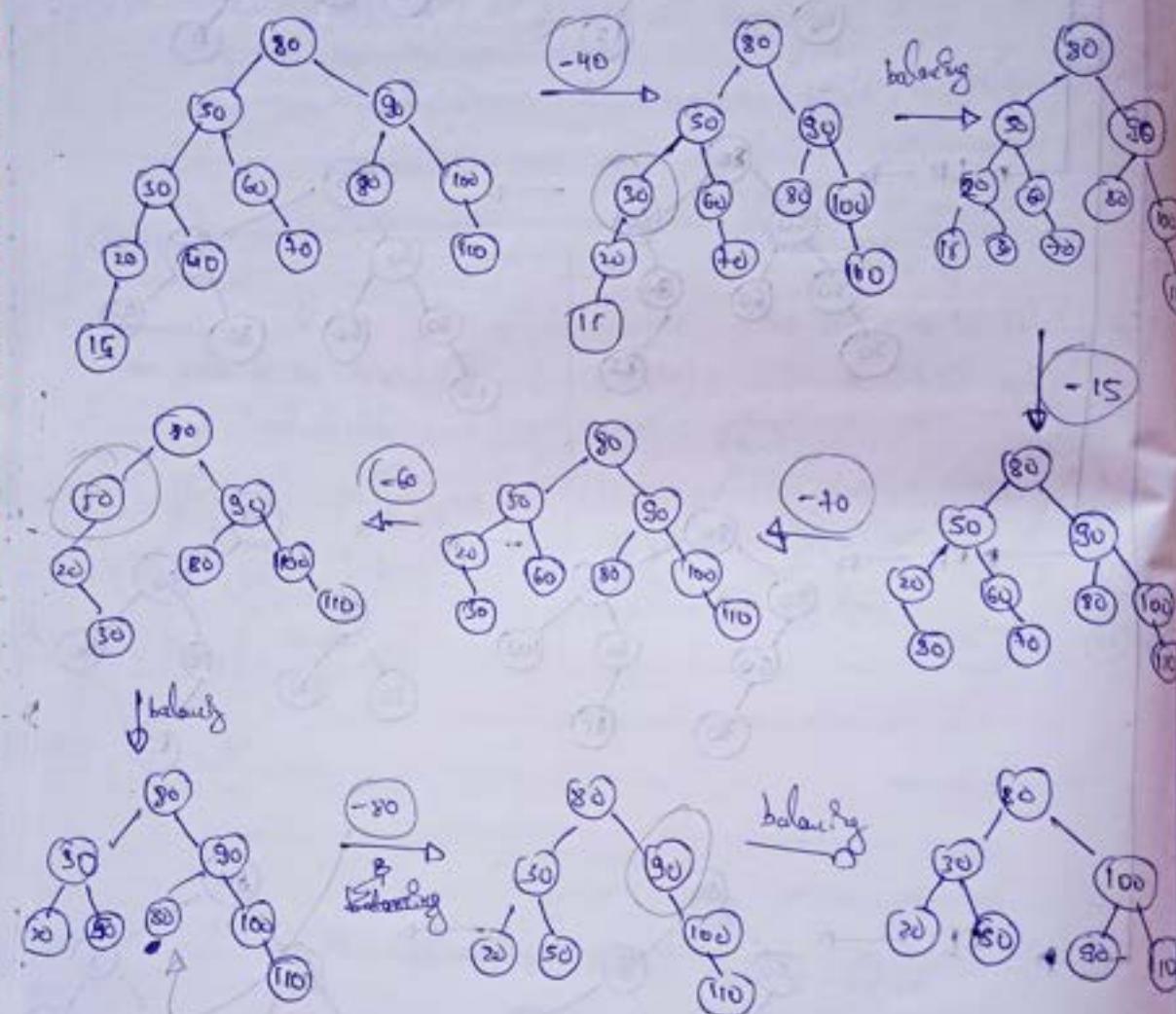
The node to be deleted has 2 children





Delete a node - from AVL tree (all together)

LL, LP, PF, RL



```

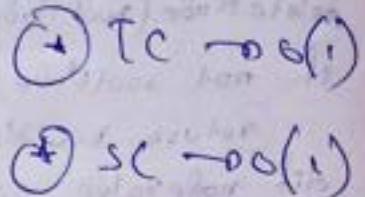
def minValueNode (rootNode):
    if rootNode is None or rootNode.leftchild is None:
        return rootNode
    return minValueNode (rootNode.leftchild)

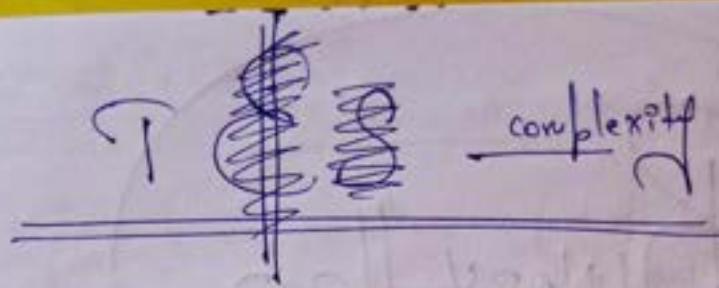
def deleteNode (rootNode, nodeValue):
    if not rootNode:
        return rootNode
    if nodeValue < rootNode.data:
        rootNode.leftchild = deleteNode (rootNode.leftchild, nodeValue)
    elif nodeValue > rootNode.data:
        rootNode.rightchild = deleteNode (rootNode.rightchild, nodeValue)
    else:
        if rootNode.leftchild is None:
            temp = rootNode.rightchild
            rootNode = None
            return temp
        elif rootNode.rightchild is None:
            temp = rootNode.leftchild
            rootNode = None
            return temp
        temp = minValueNode (rootNode.rightchild)
        rootNode.data = temp.data
        rootNode.rightchild = deleteNode (rootNode.rightchild, temp.data)
    balance = getBalance (rootNode)
    if balance > 1 and getBalance (rootNode.leftchild) >= 0:
        return rightRotate (rootNode)
    if balance < -1 and getBalance (rootNode.rightchild) <= 0:
        return leftRotate (rootNode)
    if balance > 1 and getBalance (rootNode.leftchild) < 0:
        rootNode.leftchild = leftRotate (rootNode.leftchild)
        return rightRotate (rootNode)
    if balance < -1 and getBalance (rootNode.rightchild) > 0:
        rootNode.rightchild = rightLeftRotate (rootNode.rightchild)
        return leftRotate (rootNode)
    return rootNode

```

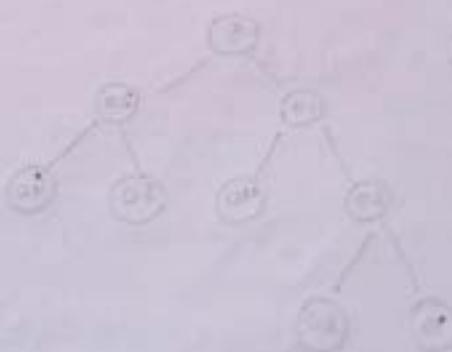
Delete entire AVL Tree

```
def deleteAVL(rootNode):  
    rootNode.data = None  
    rootNode.leftchild = None  
    _____rightchild_____  
    return "Deleted!!"
```





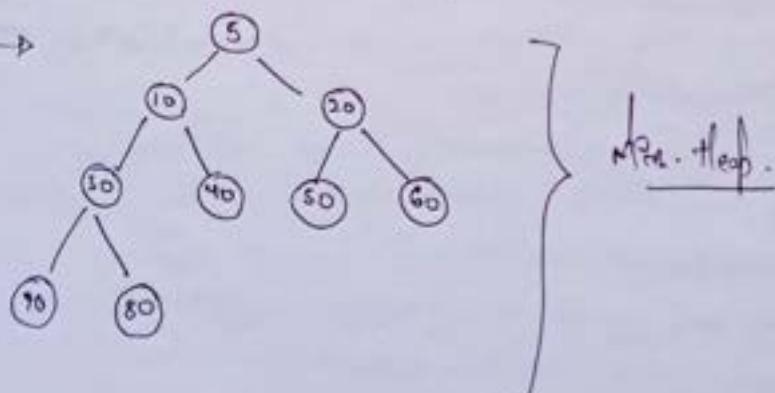
	BST	AVL
	Time / Space	Space / Time
Create AVL	$O(1)$	$O(1)$
Insert a node AVL	$O(\log N)$	$O(\log N)$
Traverse AVL	$O(N)$	$O(N)$
Search for a node AVL	$O(\log N)$	$O(\log N)$
Delete node from AVL	$O(\log N)$	$O(\log N)$
Delete entire AVL	$O(1)$	$O(1)$



BINARY HEAP

- ① A Binary heap is a Binary Tree with following properties:
- Each node should have atmost 2 subnodes.
- If B-H is either Min-heap or Max-heap
 - Min-H → the root node must be minimum among all others present in B-H & nodes in B-T
 - Max-H → maximum
- It's a complete tree [All levels are completely filled except possibly the last level should hv all nodes/keys asf to lf as possible]
- This property of B-H makes them suitable to be stored in an array.

(Ex) →



Why we need a B-H ?

Q) Find the min. & max. no. among a set of numbers in log_n time. And also we want to make sure that inserting additional numbers does not take more than O(log n) time.

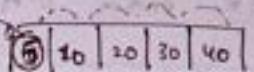
Possible solutions:

- Store the nos. in sorted Array

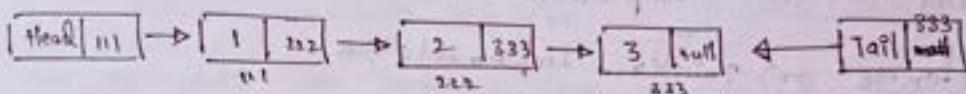
10	20	30	40	
----	----	----	----	--

o find minimum $\rightarrow O(1)$

But

Insertion \rightarrow  $\Rightarrow O(n)$

- Store the numbers in L-L in sorted manner.



Same here; if you want to insert an element then you would have to traverse right from the start.

So the Only way is Binary Heap !!

Practical Use :

- Prim's Algorithm.
- Heap sort
- Priority Queue.

Common operations on B-H

- Creation of B-H
- Peek top of B-H
- Extract Min. / Extract Max.
- Traversal of B-H
- Size of Binary H
- Insert value in B-H
- Delete entire B-H

Implementation options

- Array (or) List Implementation
- Reference (or) pointers Implementation.

Creation of B-H !

Class Heap :

```
def __init__(self, size):  
    self.customlist = (size+1)*[None] → Initialization size of  
    self.heapsize = 0 → List/Array to store  
    self.maxsize = size+1
```

newBh = Heap(5)

0	1	1	1	1
---	---	---	---	---

∴ list, dict & array is not used

① TC → O(1)

② SC → O(n)

We are symbolizing the
size of B-H as for the
array's entry -

Peek of B-H \Rightarrow Root value of a B-H

```
def peekofHeap (rootNode):  
    if not rootNode:  
        return  
    else:  
        return rootNode.customList[1]
```

① TC \rightarrow O(1)
② SC \rightarrow O(1)

Size of B-H \Rightarrow No. of elements

```
def sizeofHeap (rootNode):  
    if not rootNode:  
        return  
    else:  
        return rootNode.heapSize
```

① TC \rightarrow O(1)
② SC \rightarrow O(1)

All the traversal's are same : here we do only level order traversal

L.O Traversal in B-H

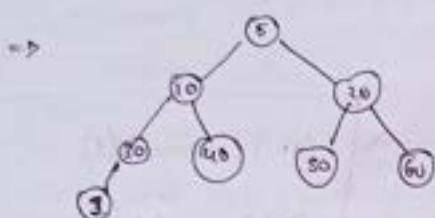
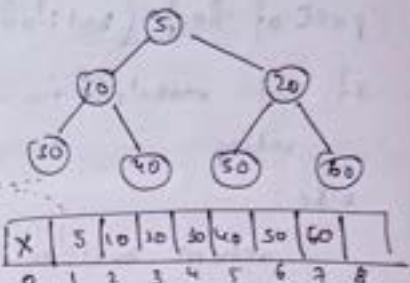
```
def levelOrderTraversal (rootNode):  
    if not rootNode:  
        return  
    else:  
        for i in range(1, rootNode.heapSize + 1):  
            print (rootNode.customList[i])
```

* TC \rightarrow O(n)
* SC \rightarrow O(n)

Insertion in B-T:

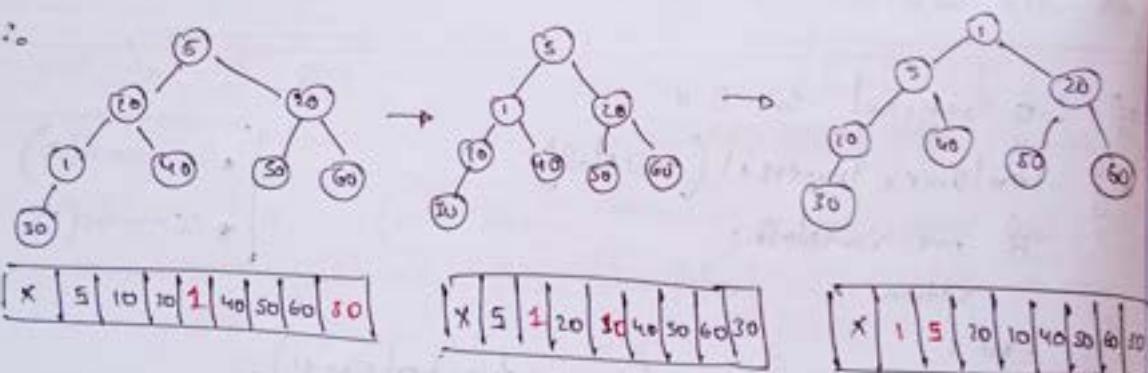
Concept

* say u wanna insert ① to



But this is not a minimum B-T

X	5	10	20	30	40	50	60	1
0	1	2	3	4	5	6	7	8



newHeap = Heap(5)

insertNode(newHeap, 4, "Max")

5
2
1

levelorderTraversal(newHeap)



~~heapsy tree Insert~~ method will do adjustments to make this
B-H proper S-H.
→ will be min/max.

def heapsyTreeInsert (rootNode, index, heapType)

parentIndex = $\text{int}(\text{index}/2)$ ← Lo of the node to which we want to
make adjustments.
if index <= 1:
 return

if heapType == "Min":
 if rootNode.customList[index] < rootNode.customList[parentIndex]:
 temp = rootNode.customList[index]
 rootNode.customList[index] = rootNode.customList[parentIndex]
 rootNode.customList[parentIndex] = temp
 heapsyTreeInsert (rootNode, parentIndex, heapType)

elif heapType == "Max":

if rootNode.customList[index] > rootNode.customList[parentIndex]:

def InsertNode (rootNode, nodeValue, heapType):

if rootNode.heapSize + 1 == rootNode.maxSize:

return "The BinaryHeap is full"

else:

rootNode.customList[rootNode.heapSize + 1] = nodeValue

rootNode.heapSize += 1

heapsyTreeInsert (rootNode, rootNode.heapSize, heapType)

return "The value is inserted"

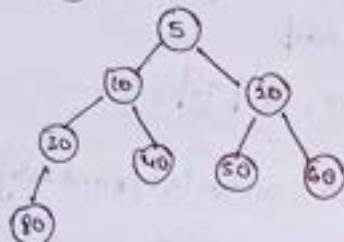
R
④ TC - O(logn)
④ SC - O(logn)

Extracting a Node from B-H

- Only the root node can be deleted from a B-H.
if after extraction its place has to be replaced by other node.

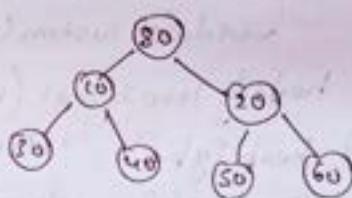
logic → for minimum B-H:

Say we want extract 5



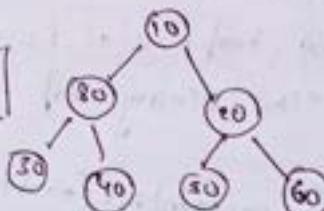
X	5	10	10	15	40	50	60	80
0	1	2	3	4	5	6	7	8

Replace 5 by last node i.e. 10



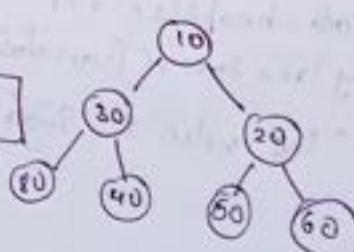
X	20	10	20	30	40	50	60	
0	1	2	3	4	5	6	7	8

↓
Make it a min. B-H again.



Since its min.
a replace 7 with 10

X	10	20	20	40	50	60	
0	1	2	3	4	5	6	7



```

def heapifyTreeExtract(rootNode, index, heapType):
    leftIndex = index * 2
    rightIndex = index * 2 + 1
    swapchild = 0
    if rootNode.headsize < leftIndex:
        return
    oldRootNode.heapSize == leftIndex:
        if heapType == "Min":
            if rootNode.customList[index] > rootNode.customList[leftIndex]:
                temp = rootNode.customList[index]
                rootNode.customList[index] = rootNode.customList[leftIndex]
                rootNode.customList[leftIndex] = temp
            return
        else:
            if rootNode.customList[index] < rootNode.customList[rightIndex]:
                return
    else:
        if heapType == "Max":
            if rootNode.customList[leftIndex] < rootNode.customList[rightIndex]:
                swapchild = leftIndex
            else:
                swapchild = rightIndex
            if rootNode.customList[index] > rootNode.customList[swapchild]:
                return
            else:
                if rootNode.customList[leftIndex] >

```

heapifyTreeExtract(rootNode, swapchild, heapType)

```

def extractNode (rootNode, heapType):
    if rootNode.heapSize == 0:
        return
    else:
        extractedNode = rootNode.customList[0]
        last node
        rootNode.customList[0] = rootNode.customList[rootNode.heapSize]
        rootNode.customList[rootNode.heapSize] = None
        rootNode.heapSize -= 1
        heapifyTreeExtract (rootNode, 0, heapType)
        return extractedNode

```

$$\begin{cases}
 \star TC \rightarrow O(\log N) \\
 \star SC \rightarrow O(\log N)
 \end{cases}$$

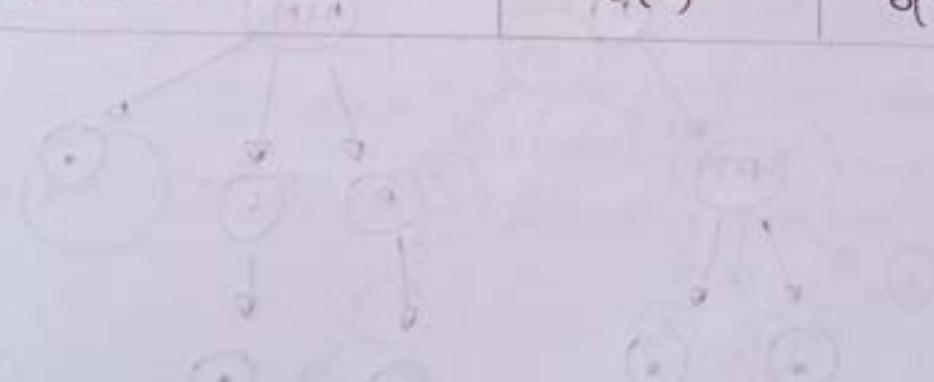
Delete entire B-H

```
def deleteEntireB_H(rootNode):
    rootNode.customList = None
```

$$\begin{array}{l} \textcircled{\star} \text{ TC} \rightarrow O(1) \\ \textcircled{\star} \text{ SC} \rightarrow O(1) \end{array}$$

ES Complexity

	<u>TIME-STONE</u>	<u>SPACE-STONE</u>
Create B-H	$O(1)$	$O(n)$
Peek of heap	$O(1)$	$O(1)$
Size of heap	$O(1)$	$O(1)$
Traversal of heap	$O(n)$	$O(1)$
Insert a node to B-H	$O(\log n)$	$O(\log n)$
Extract a node from B-H	$O(\log n)$	$O(\log n)$
Delete Entire B-H	$O(1)$	$O(1)$



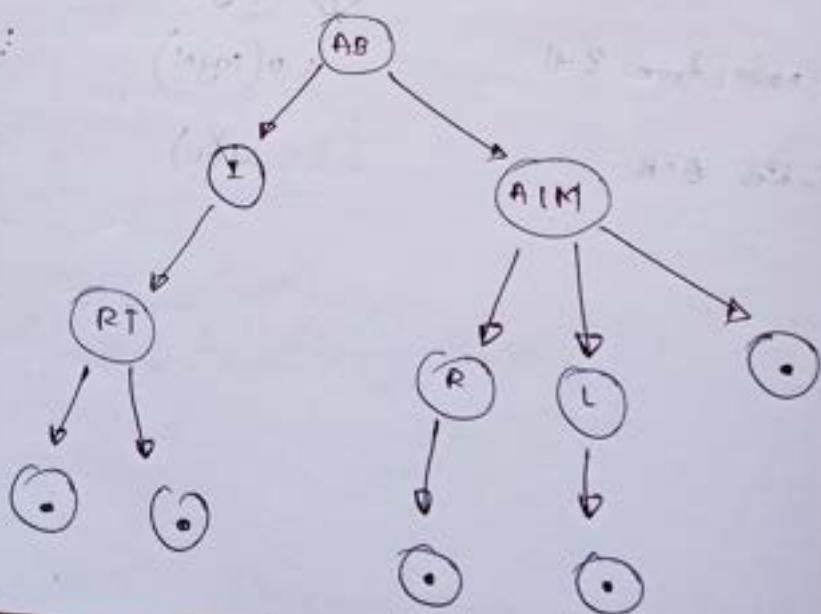
TRIE

It's a tree-based data structure that organizes information in hierarchy.

Properties:

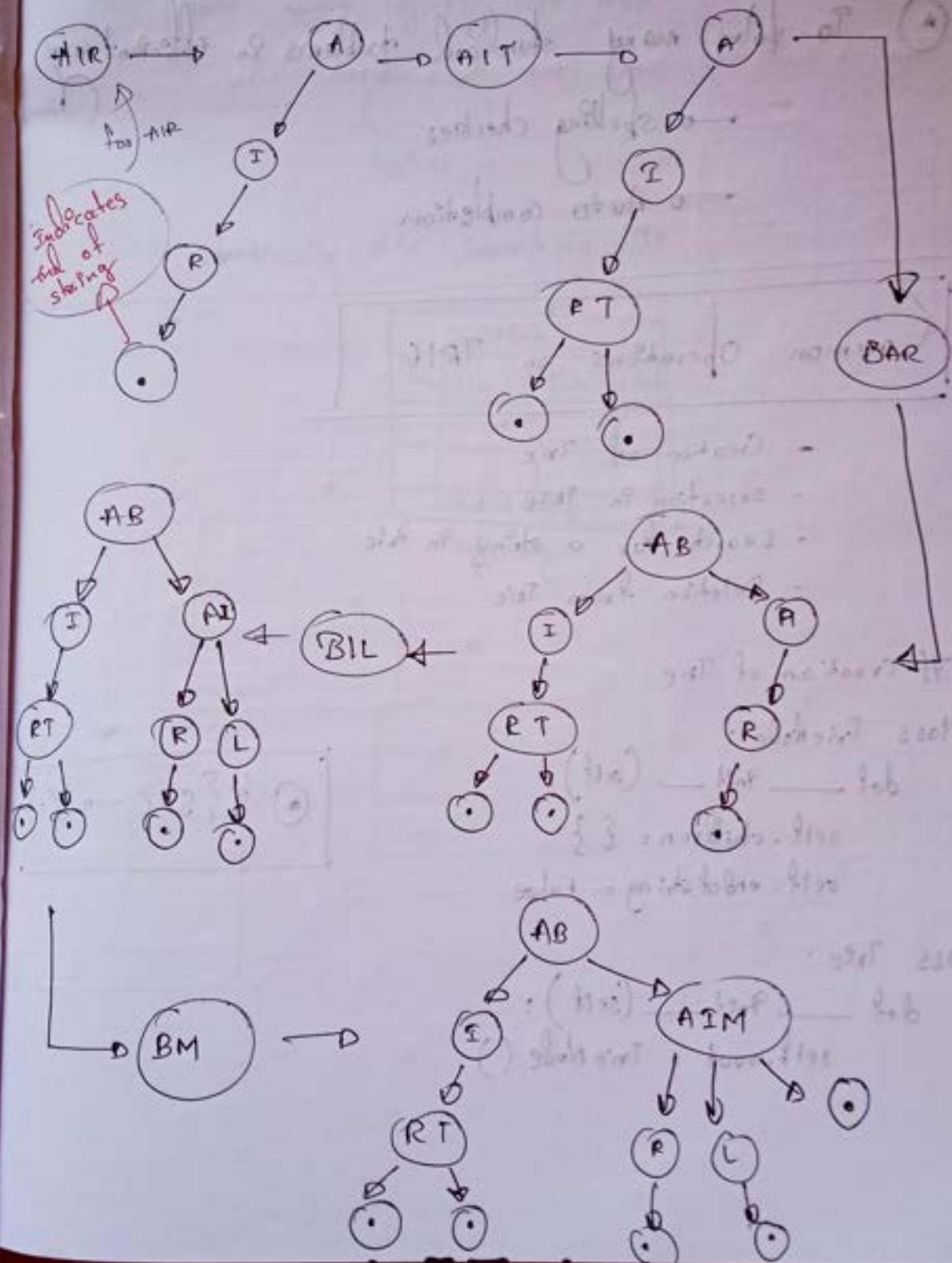
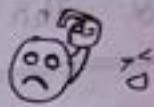
- It is typically used to store or search strings in a space and time efficient way
- Any node in trie can store Non-Repetitive multiple characters
- Every node stores link of the next character of the string
- Every node keeps track of "end of string"

Example:



the words stored here are AIR, AIT, BAF, BIL, BM

How is it happening ???



why we need this ???

- ★ To solve many standard problems in efficient way
 - spelling checker
 - Auto completion

Common Operations on Trie

- Creation of Trie
- Insertion in Trie
- Search for a string in Trie
- Deletion from Trie

Creation of Trie

Class TrieNode:

```
def __init__(self):  
    self.children = {}  
    self.endofstring = False
```

Class Trie:

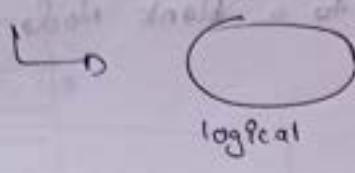
```
def __init__(self):  
    self.root = TrieNode()
```

★ T{S-C → o(i)}



Turn the page Dude ☺☺

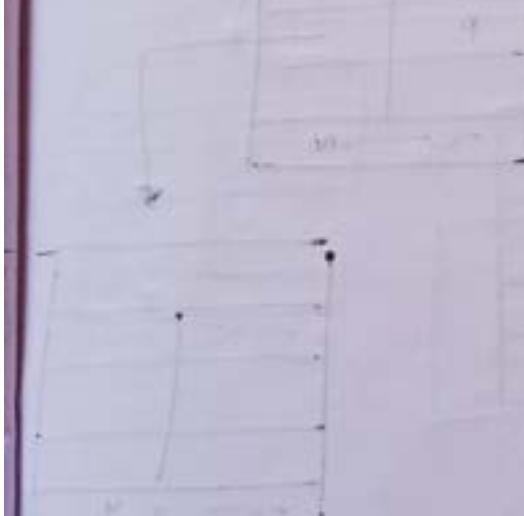
A Node might logically look like



But physically it's something like

stab	
Characters	Point to Tail Node
end of string	

Physical



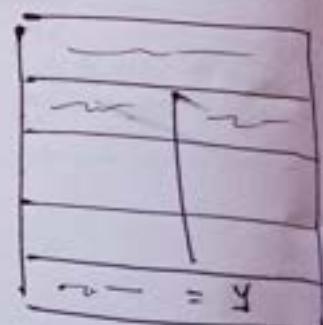
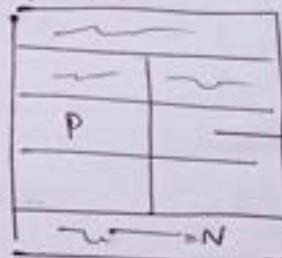
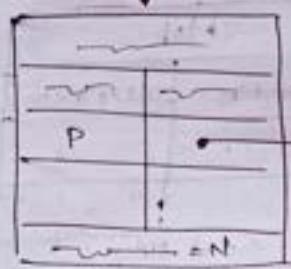
Insertion of a string in Trie

4 cases

Case 1, A Trie is Blank

Say u want to insert APP to a blank node

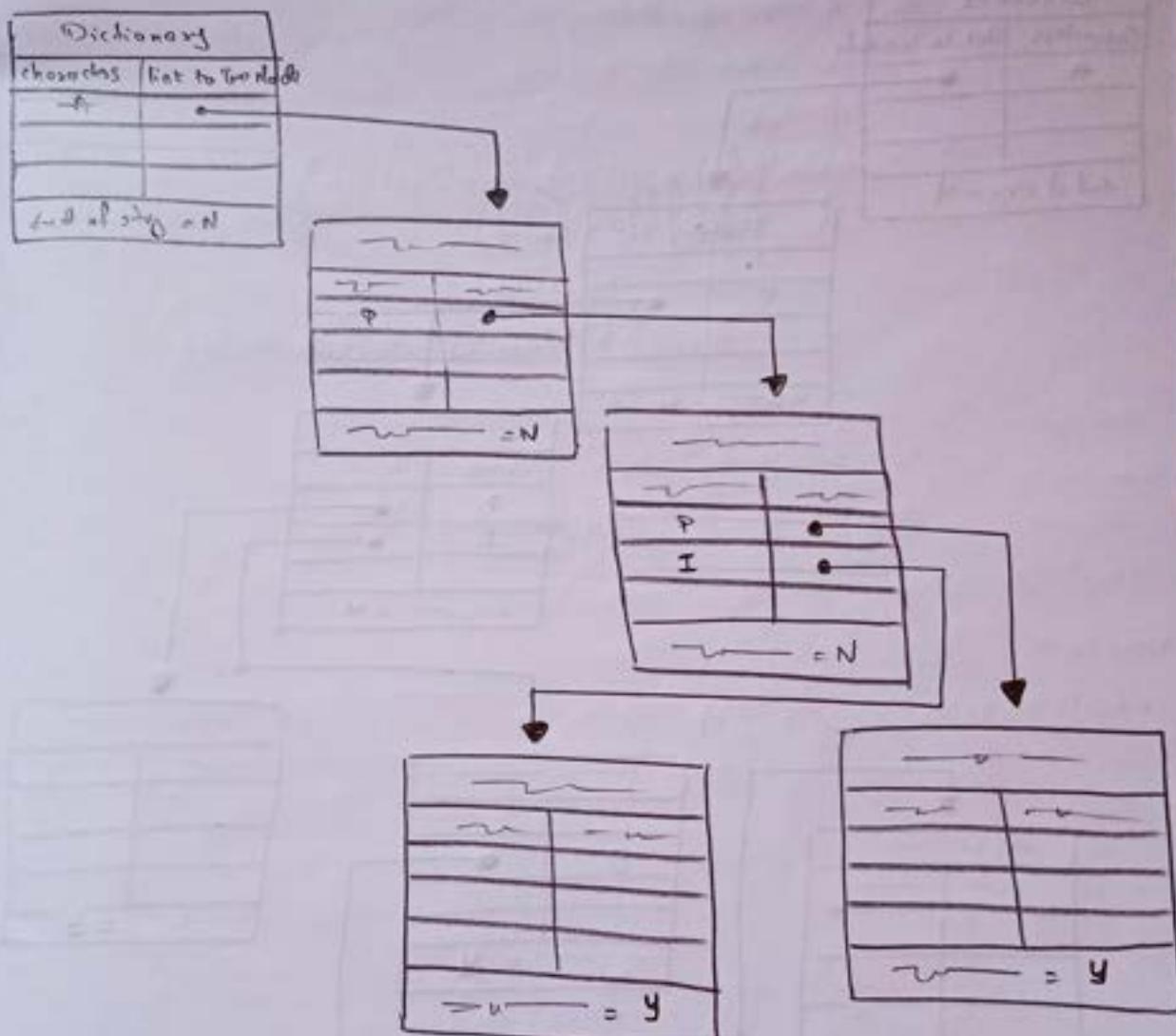
Dictionary	
Characters	Link to Triesell
A	•
End of string = N	



Case 2 : New string's prefix is common to another string's prefix.

API after APP

APP



Case 3: New string's prefix is already present as complete string.

APIs

Dictionary	
Operations	Get to Two Node
A	*
add string = N	

-	
-	-
P	*
-L--	= N

-	
-	-
P	*
I	*
-L--	= N

-	
-	-
-L--	= Y

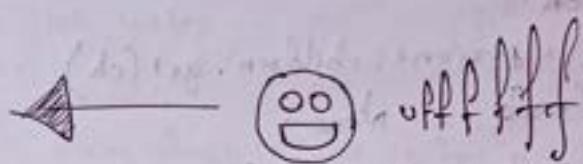
-	
-	-
S	*
-L--	= Y

-	
-	-
-L--	= Y

Case 4

: string to be inserted is already present in Trie

APIS



(*) 0 → ST

(*) 0 → ZZ

inside the class Trie

```
def insertString(self, word):
    current = self.root
    for c in word:
        ch = c
        if ch:
            node = current.children.get(ch)
            if node == None:
                node = TrieNode()
            current.children.update({ch: node})
        current = node
    current.endOfString = True
    print("Successfully inserted")
```

newTrie = Trie()

newTrie.insertString("App")
newTrie.insertString("Abp")

$$\textcircled{*} \quad TC \longrightarrow O(n)$$

$$\textcircled{*} \quad SC \longrightarrow O(n)$$

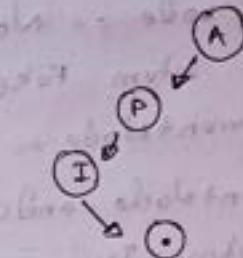
Search for a string in Trie

Case 1 : String does not exist in Trie

Say we have to search **BED** in

- * Check the 1st letter with the Root Node

* return "The string doesn't exist in Trie"



Case 2 : String exists in Trie

Say **API** in

- * Check the 1st, 2nd and 3rd letter with the corresponding node

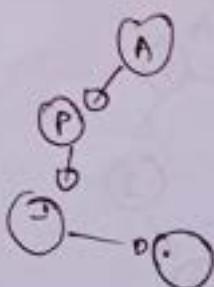


- * And it should be terminated with 0
then return "The string exists"

Case 3 : String is a prefix of another string, but doesn't exist

Say **AP** in

- * 1st & 2nd



- * But the end letter should (not) terminate with 0

```

def searchString(self, word):
    currentNode = self.root
    for i in word:
        node = currentNode.children.get(i)
        if node == None:
            return False
        currentNode = node
    if currentNode.endOfString == True:
        return True
    else:
        return False

```

```

newTrie = Trie()
newTrie.insertString("App")
print(newTrie.searchString("App"))
print(newTrie.searchString("Abp"))
print(newTrie.searchString("Dcb"))

```

True

False

False

★ TC $\rightarrow O(m)$

★ SC $\rightarrow O(1)$

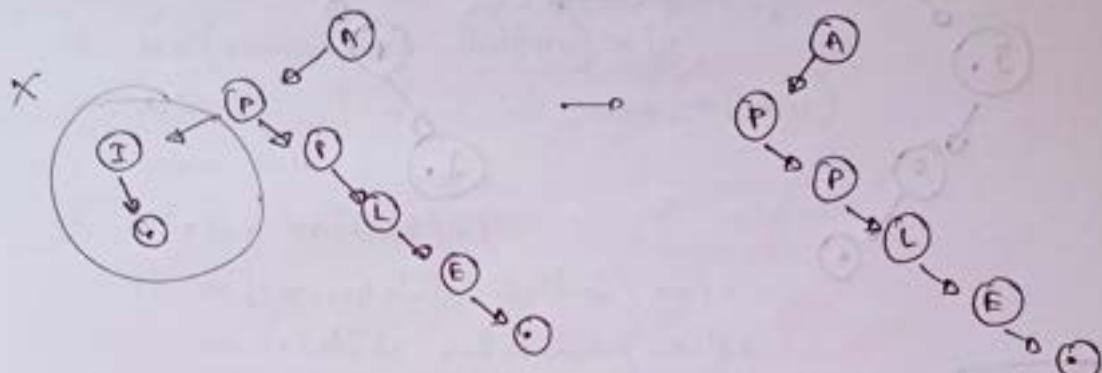
m = length of string

Deletion of a string from Trie

[Case 1]

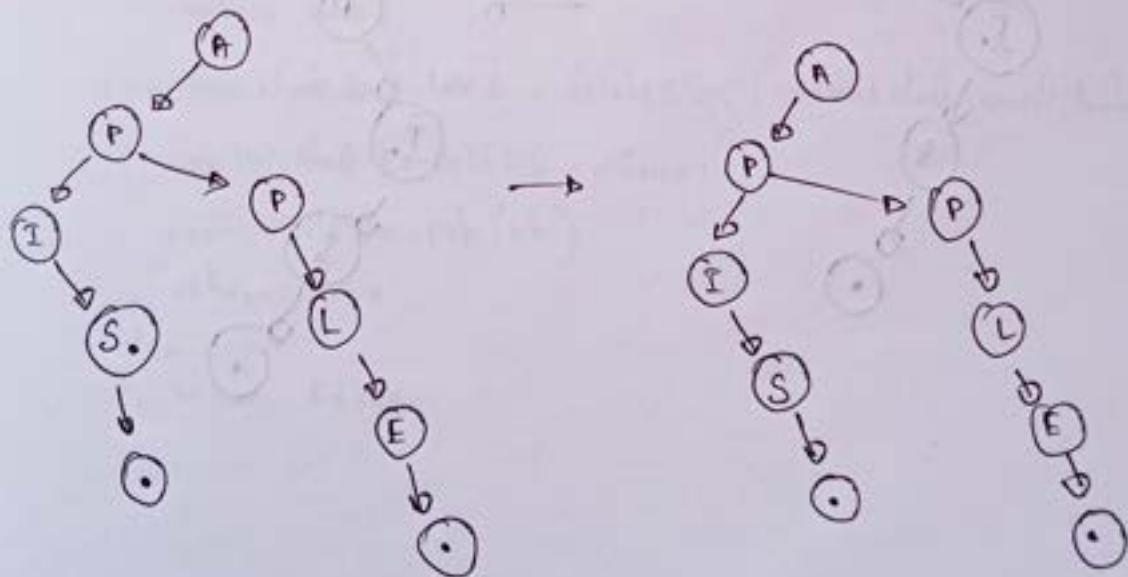
: Some other prefix of a string is same as the one that we want to delete (API, APPle)

? u wanna delete APPle



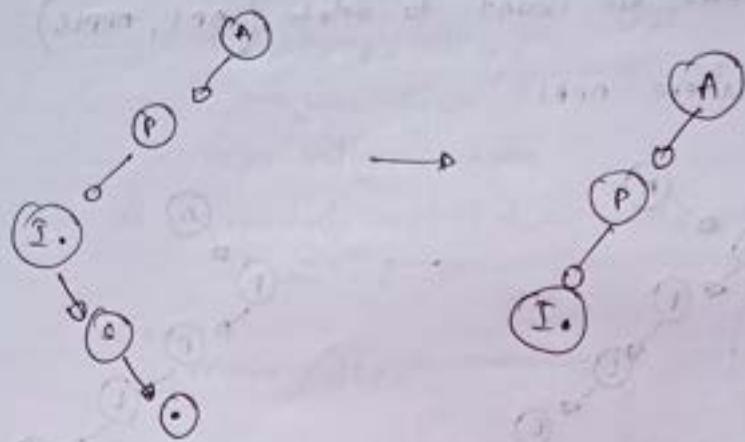
[Case 2] : The string is a prefix of another string
(API, APIS)

? u wanna delete API

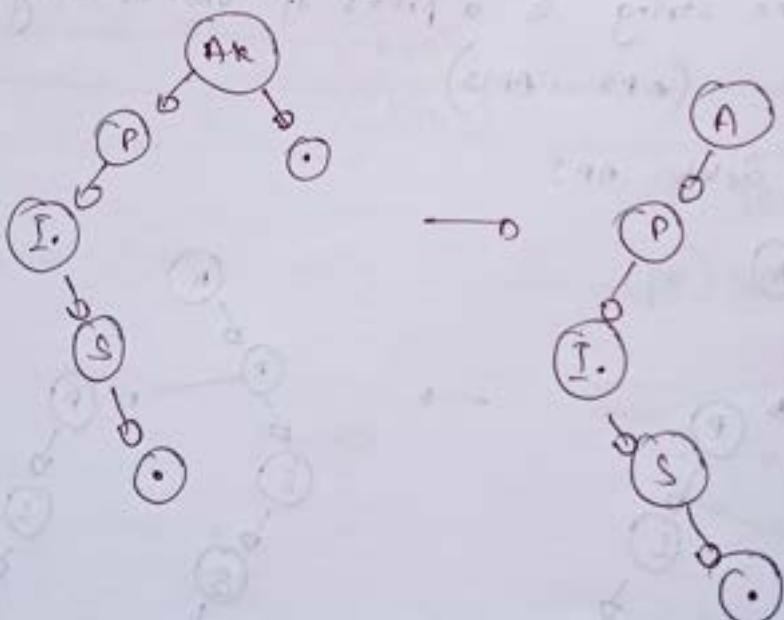


Case 3 : other string is a part of this string (apis, ...)

{ u wanna delete APIs



Case 4 : Not any node depends on this string (ϵ)



Outside Trie class:

```
def deleteString(root, word, index):
    ch = word[index]
    currentNode = root.children.get(ch)
    canThisNodeBeDeleted = False

    if len(currentNode.children) > 1:
        deleteString(currentNode, word, index+1)
        return False

    if index == len(word) - 1:
        if len(currentNode.children) >= 1:
            currentNode.endOfString = False
            return False
        else:
            root.children.pop(ch)
            return True

    if currentNode.endOfString == True:
        deleteString(currentNode, word, index+1)
        return False

    canThisNodeBeDeleted = deleteString(currentNode, word, index+1)

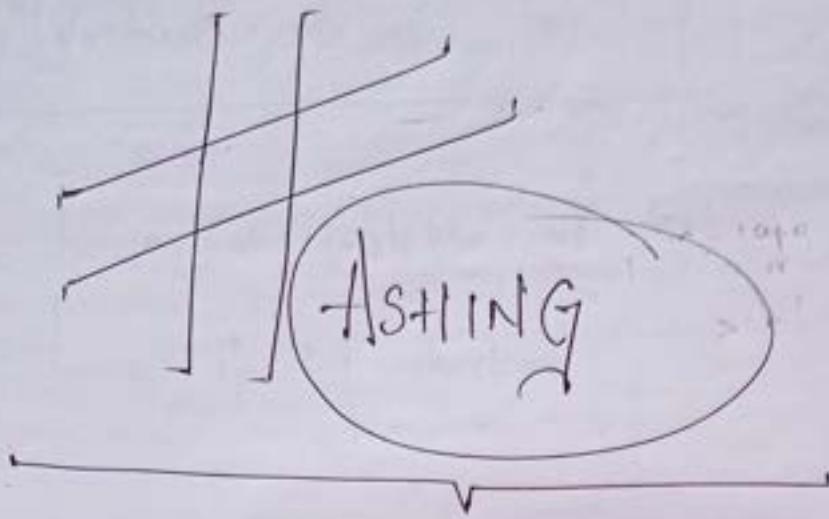
    if canThisNodeBeDeleted == True:
        root.children.pop(ch)
        return True

    else:
        return False
```

Practical use of Trie

↳ Autocompletion of text

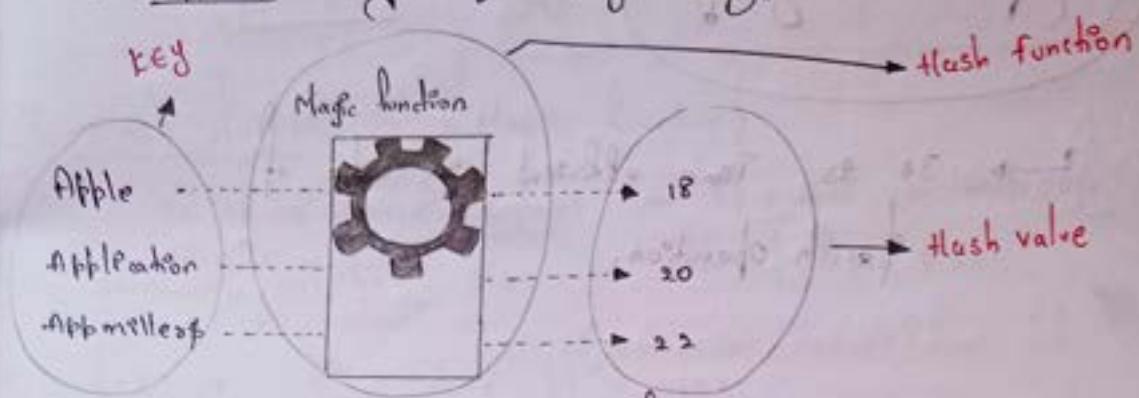
↳ spelling checker



Hashing is a method of
Sorting & indexing data

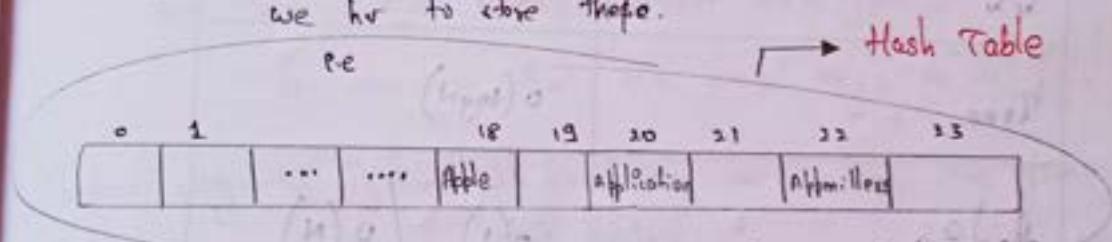
The idea behind hashing is to
allow large amounts of data to be indexed
using keys
commonly created by formulas..

- Let's say we hv 3 string & we hv to store it in an efficient way ; [i.e using hashing]



Now, we use this Magic function to convert these strings to numbers by using some formula.

Now by using some data structure - like list or arrays etc we hv to store those.



Now u can easily access by using [index value]

which takes $O(1)$ time complexity.

Why Hashing?

→ It is time efficient in case of search operation.

Data Structure	T.C for search
Array / Python list	$O(\log n)$
L.L	$O(n)$
Tree	$O(\log n)$
Hashing	$O(1)$ / $O(n)$

When the collisions are less
i.e @ ideal case

The T.C increases to $O(n)$
as the no. of collisions
increases.

Hashing Terminology:

- ① hash function : {Magic function}
It is a function that can be used to map data of arbitrary size to data of fixed size.
- ② key : Input data by a user.
- ③ hash value : The value that is returned by hash funct.
- ④ hash Table : It is a datastructure which implements an associative array abstract data type, a structure that can map keys to values.
[Basically in Python, dictionaries are using the same logic as hash tables.]
- ⑤ Collision : A collision occurs, when 2 diff keys to a hash function produce the same output.

Hash functions

%

- ① for integers e.g. [user input is an "int"]
 > total no. of cells
 > no. that you want to insert

```

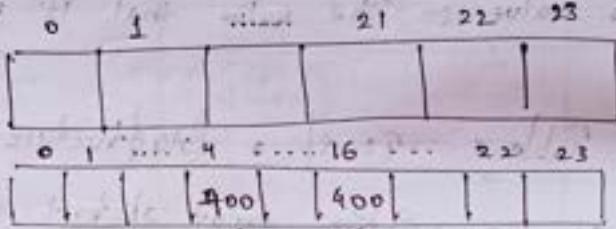
def mod(number, cellnumber):
    return number % cellnumber
    
```

Ex) $\text{mod}(900, 24)$

→ 16

$\text{mod}(900, 24)$

→ 4



- ② for string → ASCII function

```

def modASCII(string, cellnumber):
    total = 0
        
```

for s in string:

total += ord(s)

return total % cellnumber

> total no. of cells

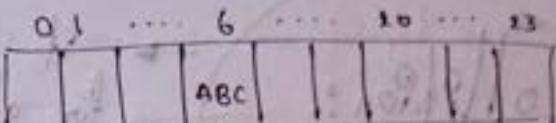
> string that you want to insert

> help to convert char to ASCII value.

Ex.

modASCII("ABC", 24)

$$\begin{array}{l} A \rightarrow 65 \\ B \rightarrow 66 \\ C \rightarrow 67 \end{array} \left. \begin{array}{l} \\ \\ \end{array} \right\} 65, 66, 67 = 198 \% 24 = 6$$



Properties of good hash function:

- It distributes hash values uniformly across hash table.
otherwise we face collision.
- It has to use all the input data

↳ say ABCD, ABCDEF are the 2 "keys" to
want to insert

? if u don't use all the input dat & e

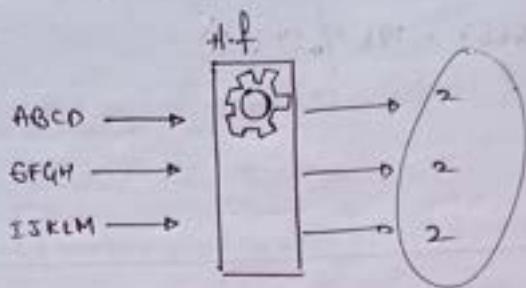
say - u use [ABC]D, [ABC]EF

both would get same hash value
which would result in Collision.

"A good hash function is the 1 which avoids
Collision."

Collision:

→ If u obtain same hash value for after hash functioning for diff key values.



Collision Resolution Techniques

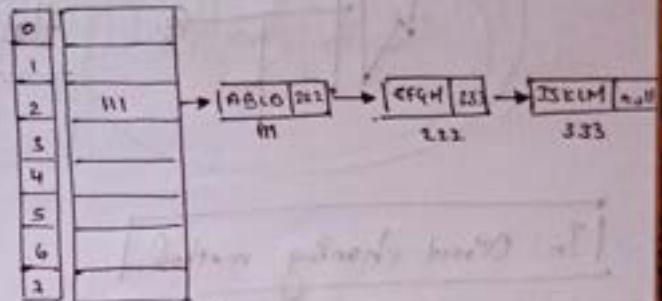
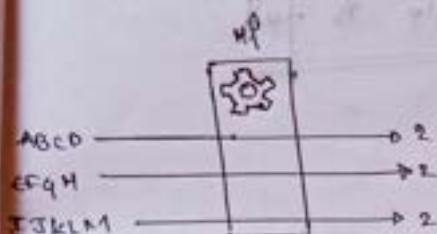
[Direct Chaining]

[Open Addressing]

- Linear Probing
- Quadratic Probing
- Double Hashing

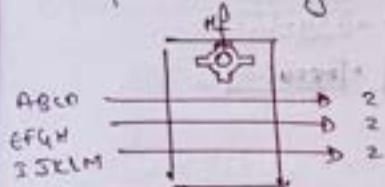
Direct Chaining

→ Implements the buckets as L.L.
Colliding elements are stored in this L.L.



Linear Probing

↳ It places new key into
closest following next empty cell

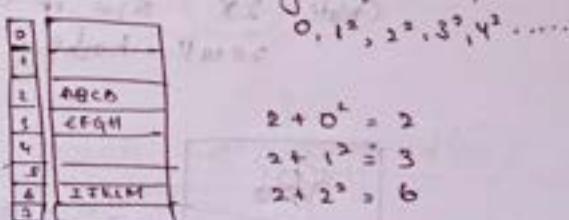
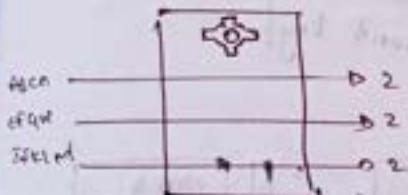


→ Colliding elements are stored in other vacant buckets.
During storage & lookup these are found using probing.

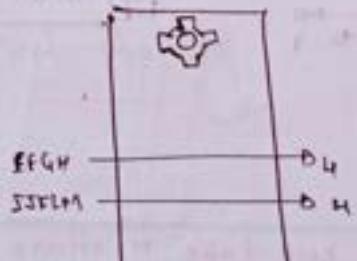
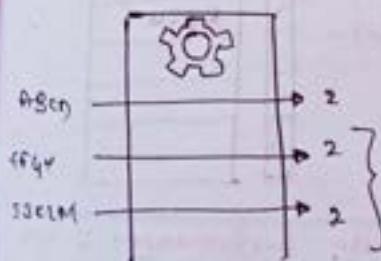


Quadratic Probing

→ Adding arbitrary quadratic polynomial to
the index until an empty cell is found.



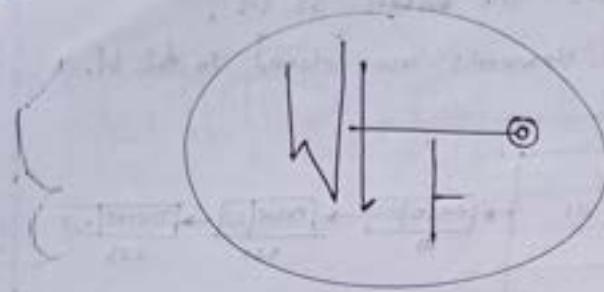
Double Hashing



$$2+4=6$$

$$(2+4=6) \Rightarrow 2(2+4)=8$$



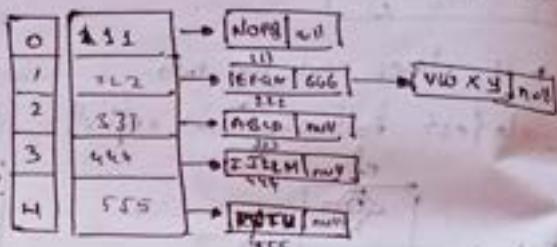


hash Table is full.

In Direct chaining method

This situation will never arise

A B C M	→ D 2
E F G H	→ D 1
I J K L M	→ D 3
N O P Q	→ D 0
R S T U	→ D 4
V W X Y Z	→ D 1



Open addressing

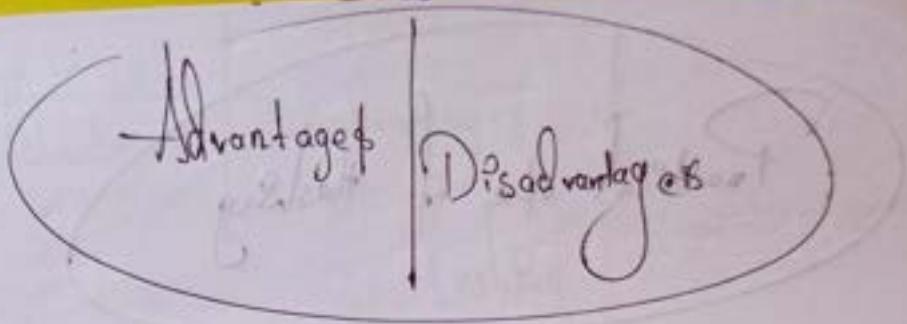
Create $(2 \times)$ size of current hash table & do linear hashing for current keys.

A B D	→ D 2
E F G H	→ D 1
I J K L M	→ D 3
N O P Q	→ D 0
R S T U	→ D 1

0	NOPQ
1	EFGH
2	ABCD
3	IJKL

0	NOPQ
1	EFGH
2	ABCD
3	IJKL
4	RSTU
5	
6	
7	

Now when you create new hash table it affects the performance : we have to call hashfunction for all these strings & more time to insert it to hash table → if the no. of strings is 'n' then we have to call 'n' times hash function which will take $O(n)$ P.C. i.e. Time consuming.



Direct Chaining

- Hash table never gets full
- Huge I.I caused performance leak.
(T.C for search operation becomes $O(n)$)

Open addressing

- easy implementation
- When Hash Table is full,
creation of new Hash tables affects performance
(T.C for search operation becomes $O(n)$).

- If the input size is known we always use "Open addressing"
- If we perform deletion operation frequently we use "Direct chaining"

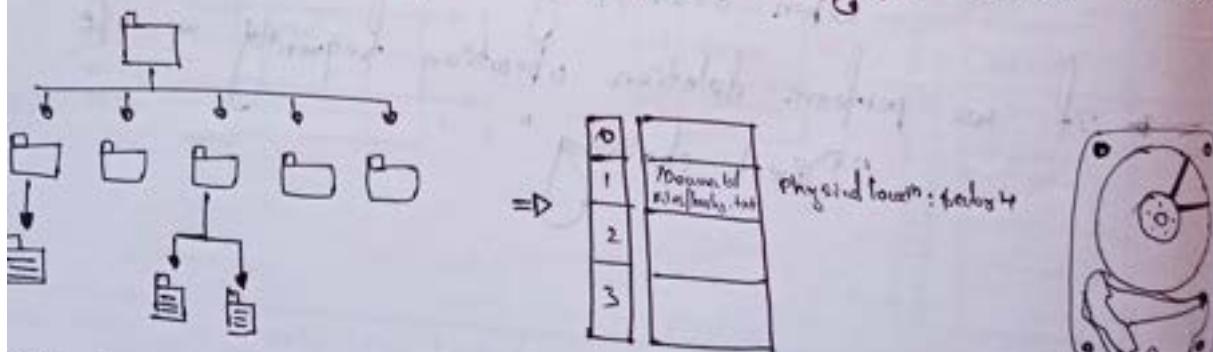
Practical use of Hashing

① Password Verification:

- * When u enter ur password for the 1st time it gets stored say in Google servers, in Hash.
- * Now when you want to login again it converts & again checks if its correct.
- * If the Google servers get hacked, the hacker won't be able to get the passwords. It would be in hash values: *271283*a12
- & moreover it's a 1 way traffic.
u cannot convert back from hash values to keys

② File system:

file path is mapped to physical location on disk



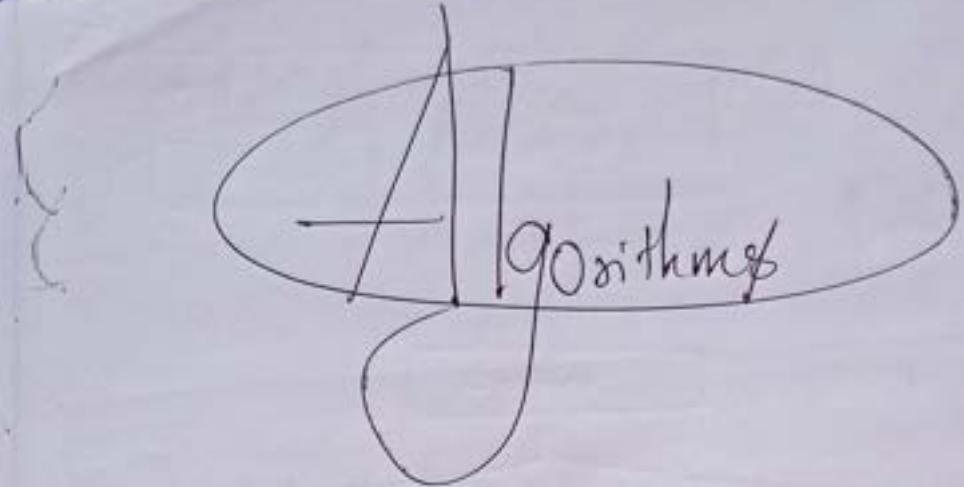
Path : /Documents/Files/hashing.txt

[It's more complex than it looks here.]

Advantages | Disadvantages of Hashing

- ✓ On an average Insertion | Deletion | Search operations take $O(1)$ T.C
- ✗ When hash funcn isn't good enough Insertion | Deletion | Search operations takes $O(n)$ T.C

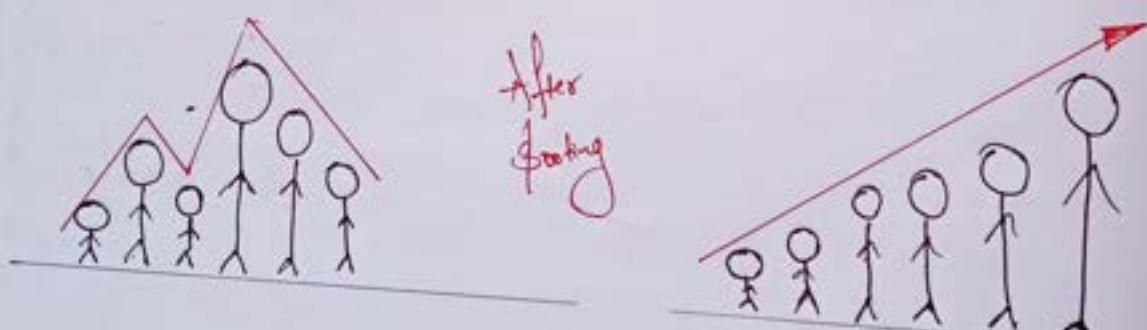
Operations	Array Python List	L.L	Tree	Hashing
Insertion	$O(n)$	$O(n)$	$O(\log n)$	$O(1) O(n)$
Deletion	$O(n)$	$O(n)$	$O(\log n)$	$O(1) O(n)$
Search	$O(n)$	$O(n)$	$O(\log n)$	$O(1) O(n)$



Sorting - Algorithm

→ Arranging data in a particular format : either ascending (or) descending.

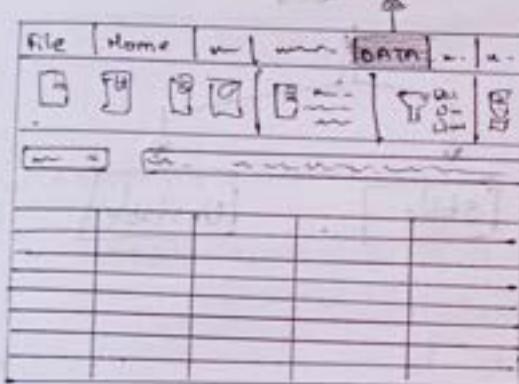
(Ex) :



Practical use of sorting

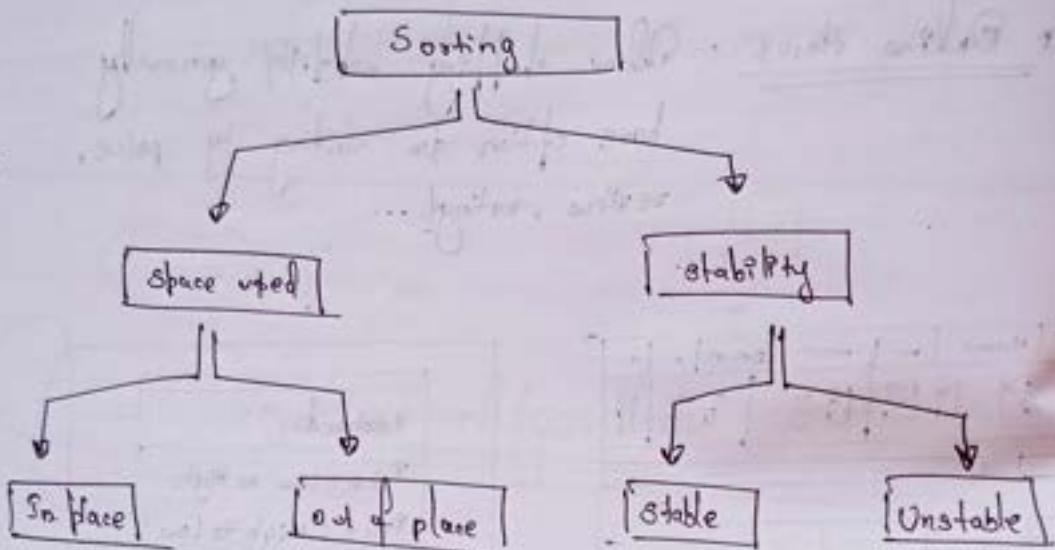
↳ Microsoft Excel : Built in functionality to sort data

↳ Online stores : Online shopping websites generally have option for sorting by price, reviews, ratings...



Featured
Price : low to High
Price : High to Low
Avg. Customer Review
Newest Arrival

Type of S. Algo



Space used

Inplace sortig : sorting algorithms which does not require any extra space for sorting.

(Ex) Bubble sort.

20 20 30 10 40 60	→	10 20 30 40 60 70
-----------------------------	---	-----------------------------

outplace sortig : Sorting algorithms which requires an extra space for sorting.

(Ex) Merge sort.

10 20 30 10 40 60	→	10 20 30 40 60 70
-----------------------------	---	-----------------------------

Stability

stable sorting : If a sorting algorithm after sorting the contents do not change the sequence of similar content in which they appear.

(Ex) Insertion sort

10 20 30 10 40 30 60 50	→	10 20 30 10 40 30 60 50
---------------------------------------	---	---------------------------------------

Unstable sorting : If a sorting algorithm after sorting the contents changes the sequence of like content in which they appear.

(Ex) Quick sort

10 20 30 10 40 30 60 50	→	10 20 30 10 40 30 60 50
---------------------------------------	---	---------------------------------------

10 20 30 30 40 50 60 70

Practical use of sorting (unstable sorting)

↳ Database mein Group by concept

useful

Sorting Terminologies

Increasing Order

- if successive element is \neq than previous one.

(Ex) : 1, 2, 3, 4, 5, 6

Decreasing Order

- if successive element is \neq than previous one.

(Ex) : 6, 5, 4, 3, 2, 1

Non-increasing Order

- if successive element is less than or equal to its previous element in the sequence.

(Ex) : 6, 5, 4, 3, 2, 1

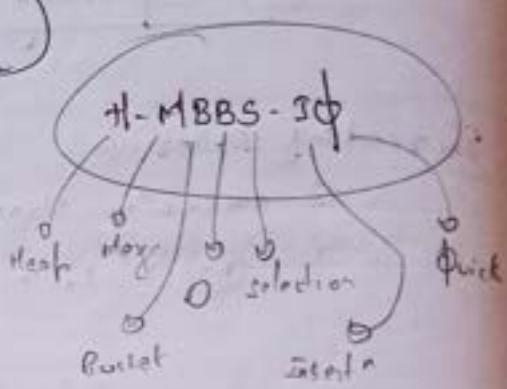
Non-decreasing Order

- if successive element is greater than or equal to its previous element in the sequence.

(Ex) : 1, 2, 3, 4, 5, 6

Sorting Algorithms

- * Bubble sort
- * Selection sort
- * Insertion sort
- * Bucket sort
- * Merge sort
- * Quick sort
- * Heap sort



Which 1 to select ?

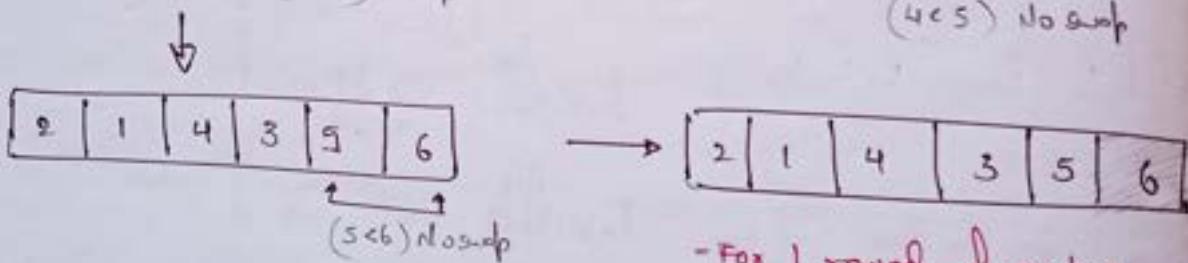
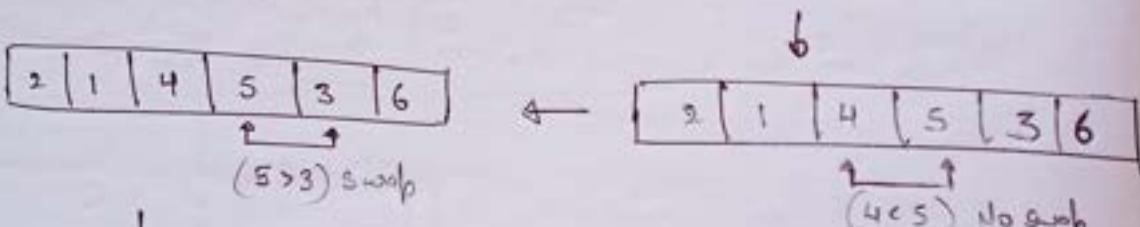
- ↳ Stability
- ↳ Space efficient
- ↳ Time efficient

Bubble Sort

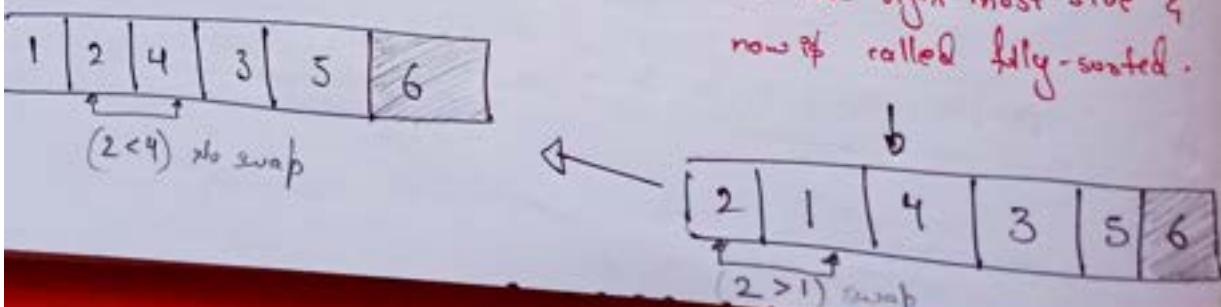
- Bubble sort is also referred as swapping sort
- we repeatedly compare each pair of adjacent items and swap them if they are in the wrong order.

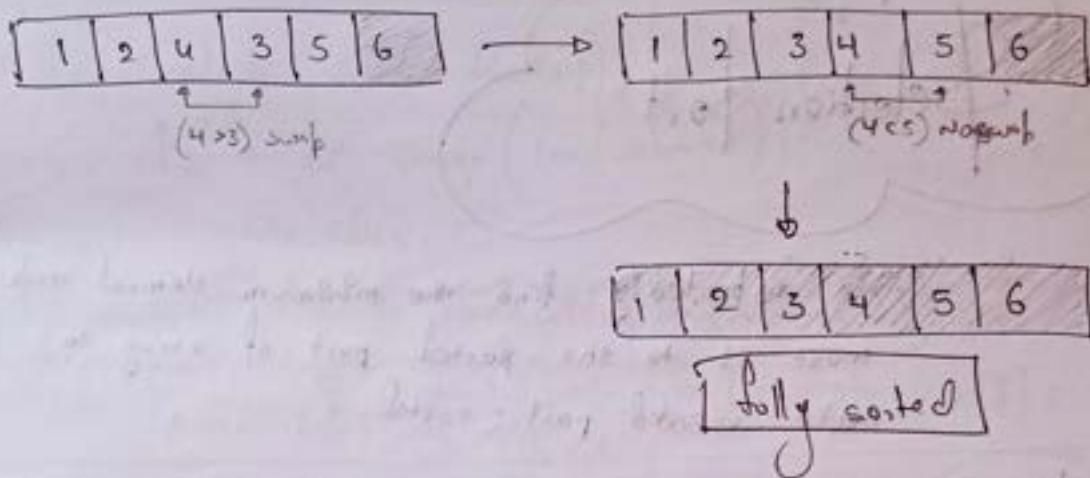
Let's understand (behind the screen) logic !!

An unsorted array/list: (for ascending)



- For 1 round of sorting
the largest element has moved
to the right most side &
now it's called fully-sorted.





```

def bubbleSort(customlist):
    for i in range(len(customlist)-1):
        for j in range(len(customlist)-i-1):
            if customlist[j] > customlist[j+1]:
                customlist[j], customlist[j+1] = customlist[j+1],
                                            customlist[j]
    print(customlist)

```

clist = [2, 1, 7, 6, 5, 3, 4, 9, 8]

bubbleSort (clist)

[1, 2, 3, 4, 5, 6, 7, 8, 9, 9]

★ T.C $\rightarrow \Theta(n^2)$
★ S.C $\rightarrow \Theta(1)$

When to use Bubble sort?

- ↳ When the input is already sorted
- ↳ Space is a concern
- ↳ Easy to implement

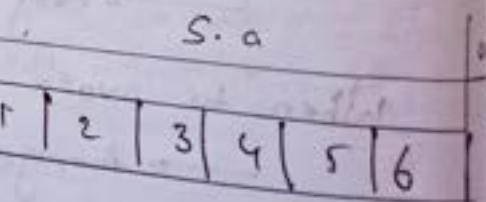
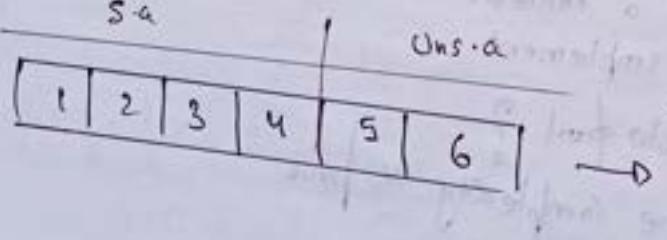
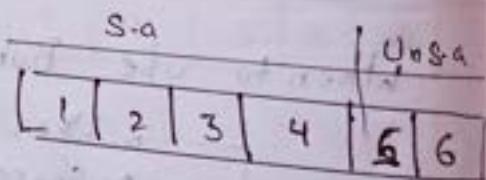
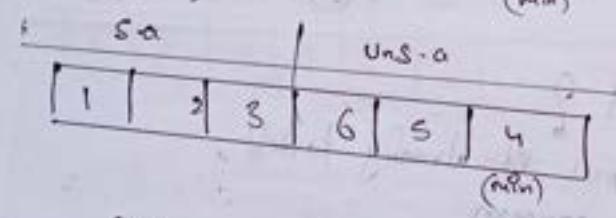
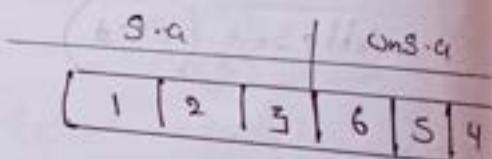
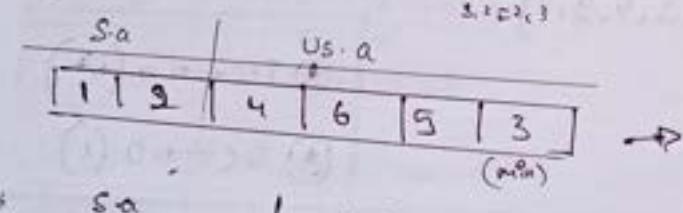
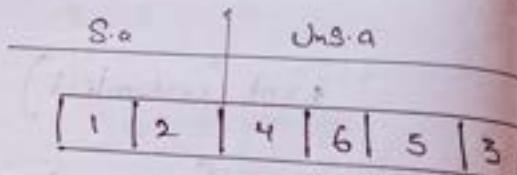
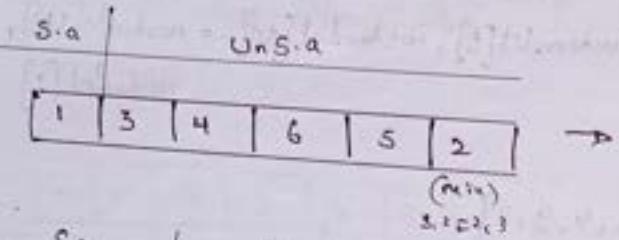
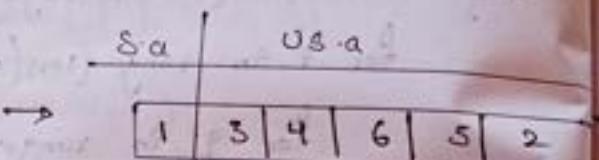
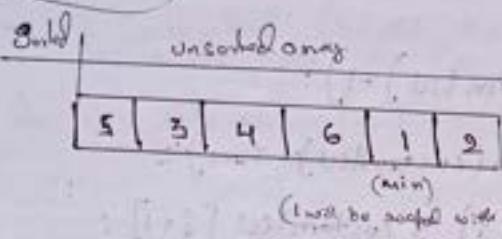
When to avoid Bubble sort?

- ↳ Average time complexity is poor

Selection Sort

- We repeatedly find the minimum element and move it to the sorted part of array to make unsorted part - sorted

logic?



```

def selectionSort(customList):
    for i in range(len(customList)):
        min_index = i
        for j in range(i+1, len(customList)):
            if customList[min_index] > customList[j]:
                min_index = j
        customList[i], customList[min_index] = customList[min_index],
                                                customList[i]
    print(customList)

```

★ TC $\rightarrow O(n^2)$, ★ SC $\rightarrow O(1)$

When to use?

- When we have insufficient memory
- Easy to implement

When to avoid?

- When time is a concern.

$i = 0$
 $j \rightarrow 1, 6$
 $5 > 3 \checkmark \rightarrow m^2 = 3$
 $3 > 4 X$
 $3 > 6 X$
 $5 > 1 \rightarrow m^2 = 1$
 $1 > 2 X$

Inception Sort

- Divide the array into 2 part. (s, uns)
- Take 1st element from unsorted array & find its correct position in sorted array.
- Repeat until unsorted array is empty

```
def insertionSort(customlist):  
    for i in range(1, len(customlist)):  
        tempKey = customlist[i]  
        j = i - 1  
        while j >= 0 and tempKey < customlist[j]:  
            customlist[j + 1] = customlist[j]  
            j -= 1  
        customlist[j + 1] = tempKey  
    print(customlist)
```

★ IC $\rightarrow O(n^2)$
★ SC $\rightarrow O(1)$

When to use?

- When we have insufficient memory
- easy to implement
- When we have continuous inflow of nos & want to keep them sorted

When to Avoid?

- When time is a concern

logic!!!

(c)

8	4	1	5	9	2
---	---	---	---	---	---

Since the 1st element is always a sorted 1 :-)

S	vs				
8	4	1	5	9	2

q	up				
8	4	1	5	9	2

so will take a temp variable & using a loop starting from index 1, assign the 1st element to temp & then compare it with the element in S.L & shift there.

for i=0 to l-1

temp = L[i]

g = p-1

while g>0 & L[g]>temp

L[g+1] = L[g]

L[g+1] = temp

i : 1	→	b
8		1 5 9 2

temp = 4

else since 4 < 8

4	8	1	5	9	2
---	---	---	---	---	---

temp = 1

1 < 8 & 1 < 4

1	4	8	1	5	9	2
---	---	---	---	---	---	---

1	4	8	1	9	2
---	---	---	---	---	---

temp = 5

5 < 8 & 5 > 4

1	4	5	8	1	9	2
---	---	---	---	---	---	---

1	4	5	8	2	
---	---	---	---	---	--

temp = 9

9 > 8

1	4	5	8	9	2
---	---	---	---	---	---

1	4	5	8	9	
---	---	---	---	---	--

temp = 2

2 < 9 < 8 < 5 < 4 &

1	2	4	5	8	9
---	---	---	---	---	---

Bucket sort

④ The T.C of Bucket Sort depends on the T.C of the sorting method used.
 [ex: T.C of Insertion Sort is $O(n^2)$:: T.C of $O(n^2)$ is $O(n^2)$]

⑤ for the best cost use Quick Sort will give T.C $\rightarrow O(n \log n)$ (S)

logic?

→ Create buckets & distribute elements of array into buckets

→ Sort buckets individually

→ Merge buckets after sorting.

Formula

$$\textcircled{1} \text{ No. of Buckets} = \text{round}(\sqrt{n} \text{ (no. of elements)})$$

$$\textcircled{2} \text{ Appropriate Bucket Num} = \text{ceil}(\text{Value} * \text{No. of Buckets}) / \text{no. of elements}$$

What's happening

5	3	4	7	2	8	6	9	1
---	---	---	---	---	---	---	---	---

$$- \text{No. of Buckets} = \text{round}(\sqrt{9}) = 3$$

$$- \text{App Bucket Num} = \text{ceil}(5 * 3 / 9) = \text{ceil}(1.6) = 2$$

$$- \text{App Bucket Num} = \text{ceil}(3 * 3 / 9) = \text{ceil}(1) = 1$$

$$= \text{ceil}(4 * 3 / 9) = \text{ceil}(1.3) = 2$$

$$= \text{ceil}(7 * 3 / 9) = \text{ceil}(2.3) = 3$$

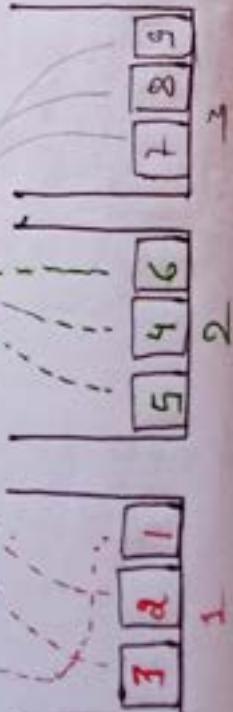
$$= \text{ceil}(2 * 3 / 9) = \text{ceil}(0.6) = 1$$

$$= \text{ceil}(8 * 3 / 9) = \text{ceil}(2.6) = 3$$

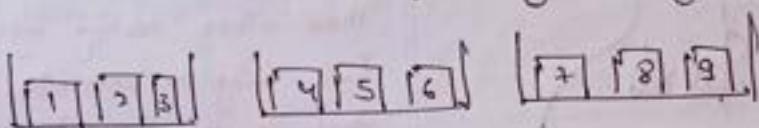
$$= \text{ceil}(6 * 3 / 9) = \text{ceil}(2) = 2$$

$$= \text{ceil}(9 * 3 / 9) = \text{ceil}(3) = 3$$

$$= \text{ceil}(1 * 3 / 9) = \text{ceil}(0.3) = 1$$



- Sort individual buckets (use any sorting algorithm)



- Merge all the buckets : [1|2|3|4|5|6|7|8|9]

Import math

```
def bucketSort(customList):
    numofBuckets = round(math.sqrt(len(customList)))
    maxValue = max(customList)
    arr = [] # temp array for buckets

    for i in range(numofBuckets):
        arr.append([])

    for j in customList:
        b_index = math.ceil(j * numofBuckets / maxValue)
        arr[b_index - 1].append(j) # -1 as array index starts from 0

    for i in range(numofBuckets):
        arr[i] = insertionSort(arr[i]) # notice a change in this sorting method i.e., instead of printing, return the customlist@list

    k = 0
    for i in range(numofBuckets):
        for j in range(len(arr[i])):
            customList[k] = arr[i][j]
            k += 1

    return customList
```

TC $\rightarrow O(n^2)$
SC $\rightarrow O(n)$ (\because we're creating a temp array)

When to use?

↳ When input is uniformly distributed over range:
P.e. if b/w element is uniform

[1|2|3|4|6|8|9] (v) [1|2|4|9|4|9|10] (x)

When to avoid?

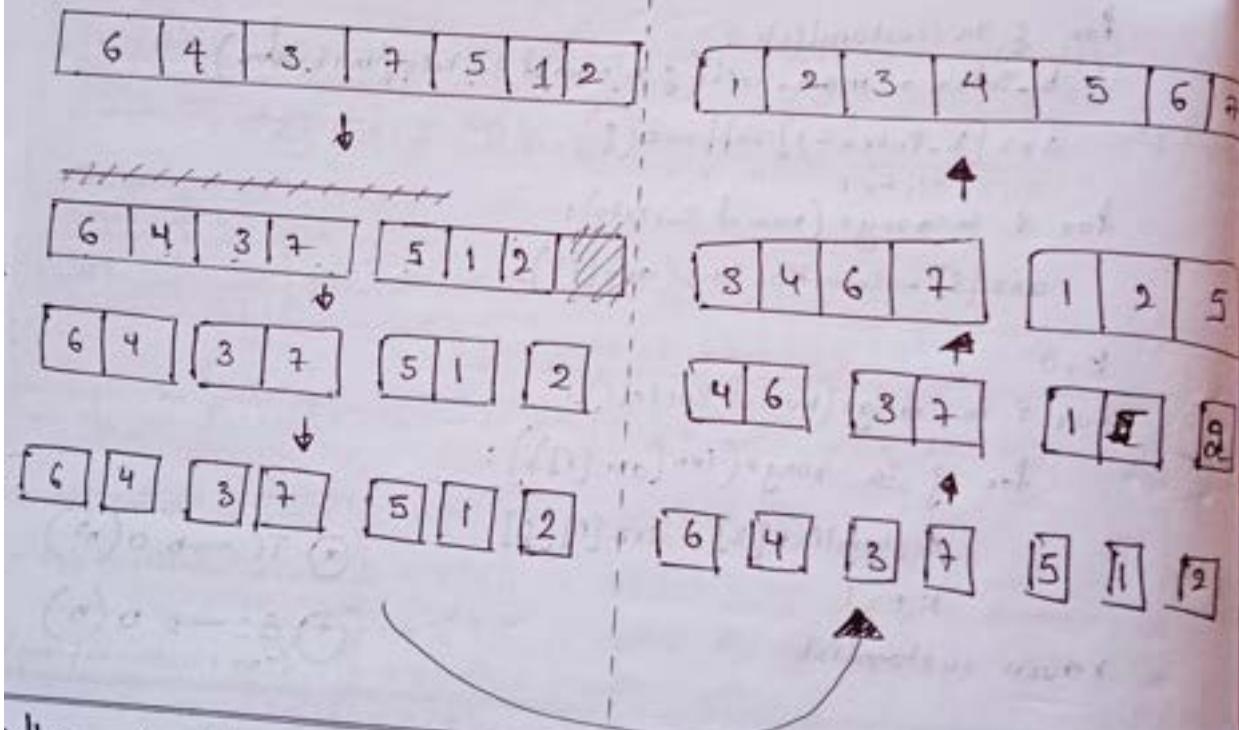
↳ When space is a concern.

Merge Sort

★ Merge Sort is faster than other sorting algorithm but thing to remember is that it takes $O(n)$ Space complexity.

- Merge sort is a "Divide & Conquer ALGORITHM"
- Divide the input array in 2 halves & we keep halving recursively until they become too small that cannot be broken further.
- Merge halves by sorting them.

LOGIC !!!



When to use :

- when you need stable sort
- when average expected time is $O(n \log n)$

When to avoid :

- when space is a concern.

10/11

helper function for mergeSort

```

def merge(customlist, l, m, r):
    # Parameters - clist, left[], middle
    # right[], & last()
    n1 = m - l + 1
    # no. of elements in 1st subarray
    n2 = r - m
    # no. of elements in 2nd subarray
    L = [0] * (n1)
    R = [0] * (n2)
    # 2 temp arrays for 1st & 2nd subarray

    for i in range(0, n1):
        L[i] = customlist[l+i]
    # copying elements from customlist to these 2 arrays

    for j in range(0, n2):
        R[j] = customlist[m+1+j]

    i = 0
    # initial [] of 1st subarray
    j = 0
    # 2nd
    k = l
    # merged

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            customlist[k] = L[i]
            i += 1
        else:
            customlist[k] = R[j]
            j += 1
        k += 1

    while i < n1:
        customlist[k] = L[i]
        i += 1
        k += 1

    while j < n2:
        customlist[k] = R[j]
        j += 1
        k += 1

```

def mergeSort(customlist, l, r):

If $l < r$:

$$m = (l + (r - 1)) // 2 \quad \# mid element$$

mergeSort(customlist, l, m) \rightarrow for 1st subarray

mergeSort(customlist, m+1, r) \rightarrow for 2nd subarray

merge(customlist, l, m, r)

return customlist

★ TC $\rightarrow O(n \log n)$
 ★ SC $\rightarrow O(n)$

Quick Sort

- Quick sort is a divide & conquer algo.
- Select pivot no. & make sure smaller numbers are located @ the left of pivot & bigger no. to right of pivot.
- Unlike merge sort extra space is not required.

LOGIC !!

* Consider an array

3	5	8	1	2	9	4	7	6
---	---	---	---	---	---	---	---	---

left = 0

3	5	8	1	2	9	4	7	6
①	②	③	④	⑤	⑥	⑦	⑧	⑨

all nos.
Right Pivot no.

* Since the smaller no. should be on left & larger to right of pivot no. will start from ①

3	5	8	1	2	9	4	7	6
①	②	③	④	⑤	⑥	⑦	⑧	⑨

$3 < 6 \checkmark$

3	5	8	1	2	9	4	7	6
①	②	③	④	⑤	⑥	⑦	⑧	⑨

$5 < 6 \checkmark$

$8 > 6 \therefore$ the ① stops @ ⑧
then ① moves ...

3	5	8	1	2	9	4	7	6
①	②	③	④	⑤	⑥	⑦	⑧	⑨

$7 > 6 \checkmark$

{ Since $4 < 6 \therefore$ the ② stops

{ then ④ & ⑥ replaces their position

3	5	4	1	2	9	8	7	6
①	②	④	⑤	⑥	⑦	⑧	⑨	⑩

{ then the same ..

3	5	4	1	2	9	8	7	6
①	②	④	⑤	⑥	⑦	⑧	⑨	⑩

$4 < 6 \checkmark$
 $1 < 6 \checkmark$
 $2 < 6 \checkmark$
 $(9 > 6)$

3	5	4	1	2	9	8	7	6
①	②	④	⑤	⑥	⑦	⑧	⑨	⑩

$8 > 6 \checkmark$

Now when both ① & ⑩ coincide to some value ; Replace that value with the ⑨ & then that ⑨ value after replacing it sorted.

3	5	4	1	2	6	8	7	9
Left part	Right part							

same concept individually

3	5	4	1	2	6	8	7	9
①	⑥	④	②	③	⑦	⑧	⑨	⑤

1	5	4	3	2	6	8	7	9
④	⑥	③	②	①	⑦	⑧	⑨	⑤

$3 < 1 \oplus$
 $1 > 2 \otimes$
swap

1	5	4	3	2	6	8	7	9
④	⑥	③	②	①	⑦	⑧	⑨	⑤

1	2	4	3	5	6	8	7	9
④	⑥	③	②	①	⑦	⑧	⑨	⑤

1	2	4	3	5	6	8	7	9
④	⑥	③	②	①	⑦	⑧	⑨	⑤

1	2	4	3	5	6	8	7	9
④	⑥	③	②	①	⑦	⑧	⑨	⑤

when ④ coincides with ⑥ that element is sorted.

```
def partition(customlist, low, high):
    i = low - 1
    pivot = customlist[high]
    for j in range(low, high):
        if customlist[j] <= pivot:
            i += 1
            customlist[i], customlist[j] = customlist[j], customlist[i]
    customlist[i+1], customlist[high] = customlist[high], customlist[i+1]
    return (i+1)
```

def quicksort (customlist, low, high):

```
if low < high:
    pi = partition(customlist, low, high)
    quicksort(customlist, low, pi-1)
    quicksort(customlist, pi+1, high)
```

When to use :

- When average expected time is $O(n \log n)$

1	2	4	3	5	6	8	7	9
④	⑥	③	②	①	⑦	⑧	⑨	⑤

1	2	3	4	5	6	8	7	9
④	⑥	③	②	①	⑦	⑧	⑨	⑤

1	2	3	4	5	6	8	7	9
④	⑥	③	②	①	⑦	⑧	⑨	⑤

1	2	3	4	5	6	8	7	9
④	⑥	③	②	①	⑦	⑧	⑨	⑤

1	2	3	4	5	6	7	8	9
④	⑥	③	②	①	⑦	⑧	⑨	⑤

completely sorted.

```
def partition (clist, low, high):
    i = low - 1
    j = high - 1
    pivot = clist [high]
    while i < j:
        while i < high and clist[i] < pivot:
            i += 1
        while j > low and clist[j] > pivot:
            j -= 1
        if i < j:
            clist[i], clist[j] = clist[j], clist[i]
    if clist[i] > pivot:
        clist[i], clist[high] = clist[high], clist[i]
    return i
```

TC $\rightarrow O(n \log n)$

SC $\rightarrow O(n)$

When to avoid :

- When space is a concern
- When you need stable sort

Heap sort

This is best suited with array.
It doesn't work best with list.

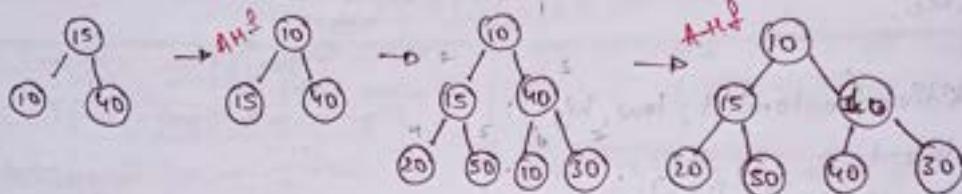
- Insert data to Binary Heap Tree
- Extract data from Binary Heap

how it goes!!

After applying

15	10	40	20	50	10	50	45	5
----	----	----	----	----	----	----	----	---

↳ Inserting to B-Tree



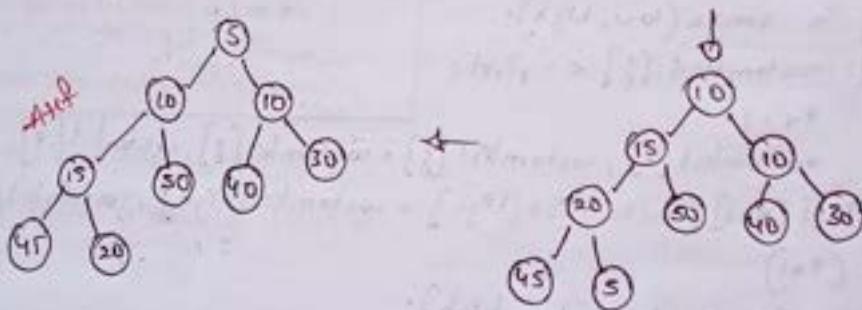
Min-Heap

No. of leaf nodes ($n - \text{end index}$)

$$(\frac{n}{2} + 1) \rightarrow n - n$$

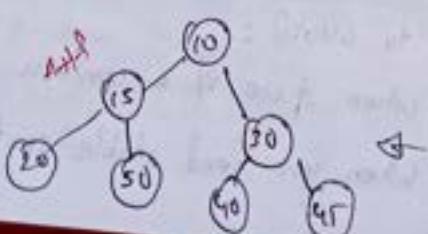
$$O(\frac{n}{2} + 1) = O(n)$$

def



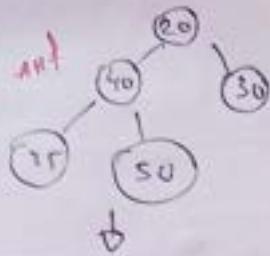
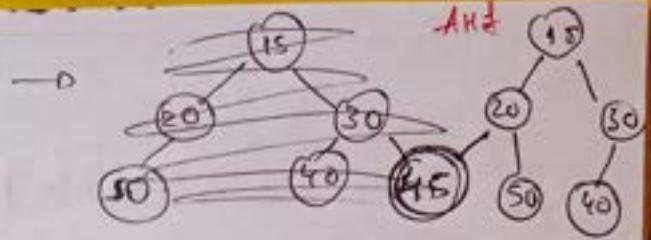
↳ Extracting data

5						
---	--	--	--	--	--	--



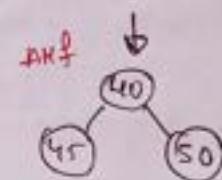
5	10	50				
---	----	----	--	--	--	--

5	10	10					
---	----	----	--	--	--	--	--



5	10	10	15				
---	----	----	----	--	--	--	--

5	10	10	15	20			
---	----	----	----	----	--	--	--



def heapify(customlist, n, i):

smallest = i

l = 2 * i + 1

r = 2 * i + 2

if l < n and customlist[l] < customlist[smallest]:

smallest = l

if r < n and customlist[r] < customlist[smallest]:

smallest = r

if smallest != i:

customlist[i], customlist[smallest] = customlist[smallest], customlist[i]
heapify(customlist, n, smallest)

def heapSort(customlist):

n = len(customlist)

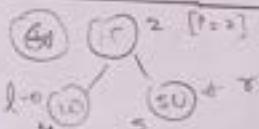
for i in range(n - 1, -1, -1):

heapify(customlist, n, i)

for i in range(n - 1, 0, -1):

customlist[i], customlist[0] = customlist[0], customlist[i]
heapify(customlist, n, 0)

customlist.reverse()



you can do the same
for max-heap

finding the smallest
node

① TC → O(nlogn) ② SC → O(1)

Searching Algorithms

Linear search
(sequential search)

Binary search

① Linear search

```
n = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
for i in range(n):  
    if i == user_input:  
        print("found")  
print("Not found")
```

TC → O(n)

SC → O(1)

② Binary search (Do after)

Recursive way

```
def binarysearch(arr, low, high, value):
    if high >= low:
        mid = (high + low) // 2
        if value == arr[mid]:
            return mid
        elif value < arr[mid]:
            return binarysearch(arr,
                                low, mid - 1, value)
        else:
            return binarysearch(arr,
                                mid + 1, high, value)
    else:
        return "not found"
```

```
def binarysearch(arr, value):
    low = 0
    high = len(arr) - 1
    mid = 0
    while low <= high:
        mid = (high + low) // 2
        if arr[mid] < value:
            low = mid + 1
        elif arr[mid] > value:
            high = mid - 1
        else:
            return mid
    return "not found"
```

GRAPH ALGORITHM

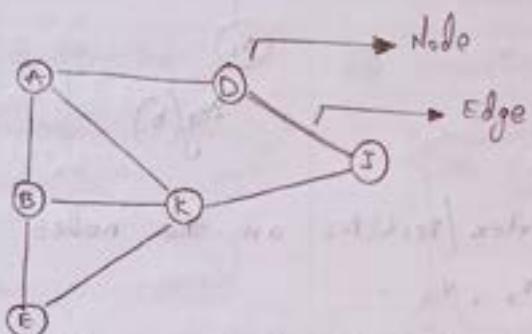
Content :-

- * What is Graph ? Why do we need it ?
- * Graph Terminologies
- * Types of Graphs . Graph Representation.
- * Traversal of Graphs . (BFS and DFS)
- * Topological sorting
- * Single source shortest path (BFS , Dijkstra & Bellman Ford)
- * All pair shortest path (BFS , Dijkstra, Bellman Ford. & Floyd Warshall algorithms)
- * Minimum spanning Tree (Kruskal and Prim algorithms)

Graph ?

→ A non-linear datastructure consisting of nodes and edges
 ↓
 (vertices)

- * A Graph consists of a finite set of vertices (or nodes) and a set of edges which connect a pair of nodes.



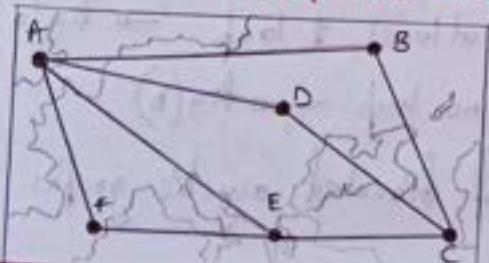
Why Graphs ?

→ Graphs are used to represent networks

Telephone network ↗
 ↗ a circuit network

- ↳ Graphs are also used in social media networks like LinkedIn, Facebook
- ↳ Graphs are also used to solve problems on shortest path.

↳ Trees are not used ∵ there's no cycle in trees



→ To go from A to C we have many paths, u don't use tree D-E ∵ cycle like it can start from A go to B to C then D & again to A.
 So u use graphs to find shortest path b/w 2 c.

GRAPH TERMINOLOGY

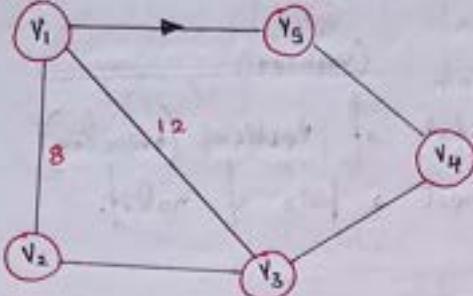


fig (a)



fig (b)

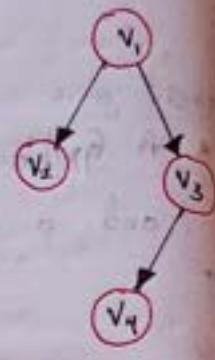


fig (c)

① **Vertices [Vertex]** : Vertex / Vertices are the nodes of the Graph

Ex : $v_1, v_2, v_3 \dots$

② **Edge** : The edge is the line that connects pairs of vertices

③ **Unweighted graph** : A graph which doesn't have a weight associated with (@) any edge.

Ex : In fig (a) : $v_1 \xrightarrow{8} v_2$

④ **1st Unweighted graph** : A graph which doesn't have a weight associated with any edge.

⑤ **Undirected graph** : In case the edges of the Graph do not have a direction associated with them.

⑥ **Directed graph** : If the edges in a graph hv a direction associated with them.

Ex : $v_1 \rightarrow v_5$ in fig (a)

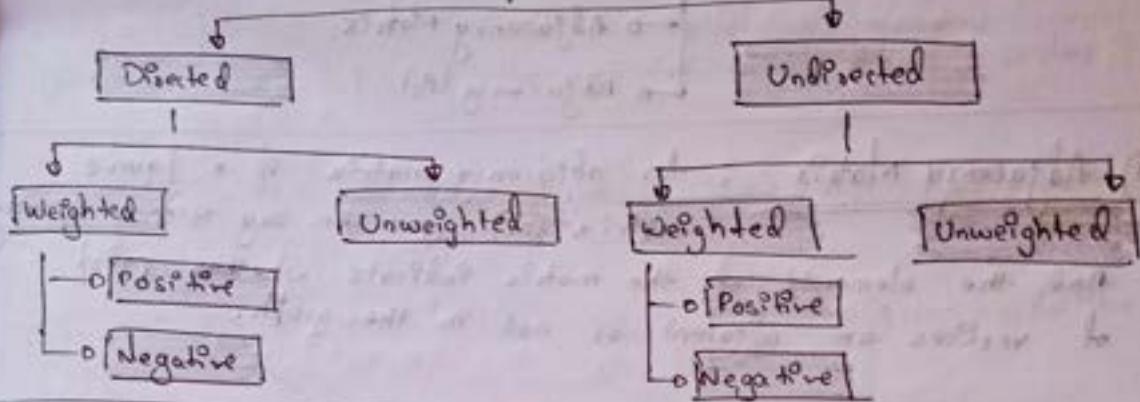
⑦ **Cyclic graph** : A graph which has atleast 1 loop \rightarrow fig (a)

⑧ **Acyclic graph** : A graph with no loop \rightarrow fig (b)

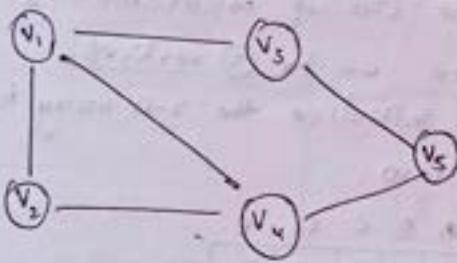
⑨ **Tree** : It is a special case of directed acyclic graphs \rightarrow fig (c)

Graph Types

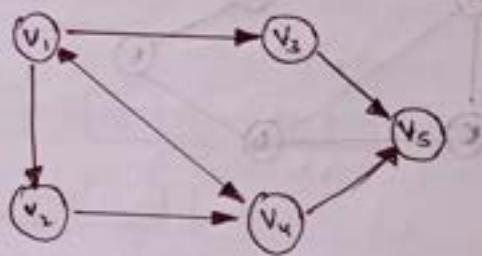
Graph



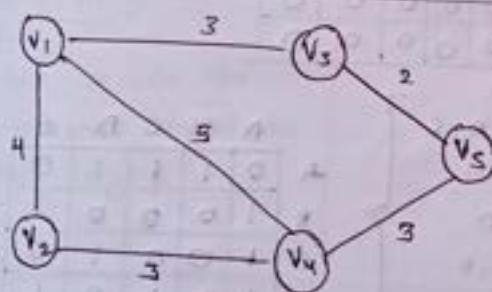
① Unweighted - Undirected



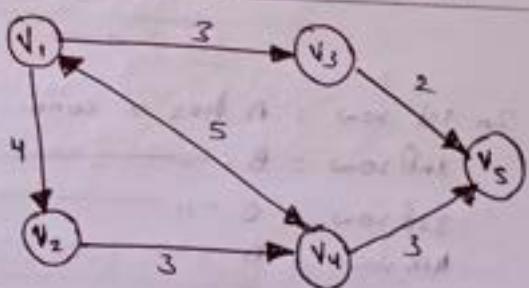
② Unweighted - Directed



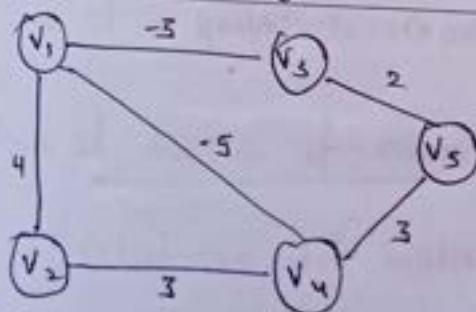
③ Positive - weighted - Undirected



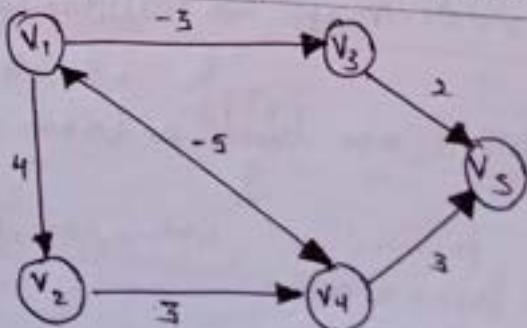
④ Positive - weighted - Directed



⑤ Negative - weighted - Undirected



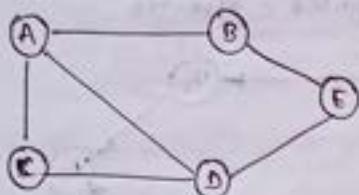
⑥ Negative - weighted - Directed



Graph Representation

↳ Adjacency Matrix
↳ Adjacency List

① Adjacency Matrix : An adjacency matrix is a square matrix (as) you can say it's a 2D array and the elements of the matrix indicate whether pair of vertices are adjacent or not in the graph.



1st - We create a 2-D array with the size of no. of vertices here we have 5 vertices. & initialize the 2-D array to zero.

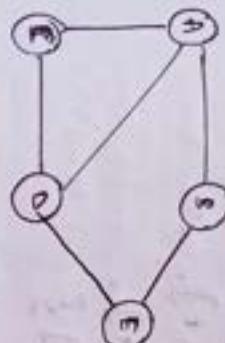
	A	B	C	D	E
A	0	0	0	0	0
B	0	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	0	0	0

In 1st row : A has a conn. with B, D, C
2nd row : B -> A, E
3rd row : C -> A, D
4th row : D -> C, A, E
5th row : E -> B, D

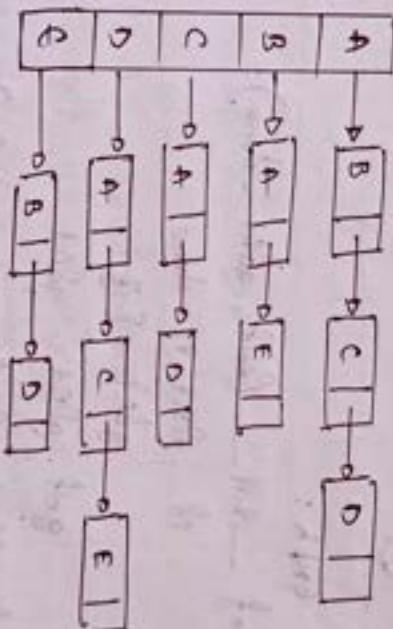
	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	0	1
C	1	0	0	1	0
D	1	0	1	0	1
E	0	1	0	1	0

② Adjacency List : An adjacency list is a collection of unordered list used to represent a graph.

- Each list describes the set of neighbors of a vertex in the graph.
- We use L.L to have all these link b/w the vertices.



[list] - we create a 1-D array with the size of the no. of vertices to the array & no repeat edges use L.L



Rem :
Here we use the array to store vertices
L.L to store edges

Rem :

Here we use the 2-D array instead to store edges.

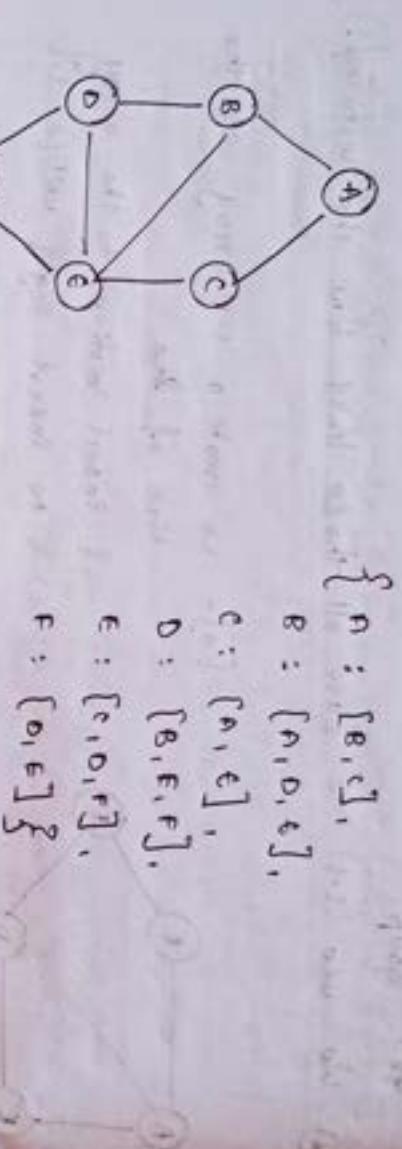
[Which is to use ?]

→ Maximum usage of edges

- If a graph is complete / almost complete we use A. matrix
- If the no. of edges are few then we should use A. list
- For A. list we used Dictionary in Python where key = vertices value = edges

Dictionary Implementation of Graph

would look like



```
Class Graph:  
def __init__(self, gdict = None):  
    if gdict is None:  
        self.gdict = {}  
    else:  
        self.gdict = gdict  
  
def add_edge(self, vertex, edge):  
    self.gdict[vertex].append(edge)
```

OR

```
customdict = { "a" : ["b", "c"],  
              "b" : ["a", "d", "e"],  
              "c" : ["a", "e"],  
              "d" : ["b", "e", "f"],  
              "e" : ["d", "f"],  
              "f" : ["a", "e"] }
```

```
graph = Graph(customdict)  
graph.add_edge("e", "c")  
graph.get()
```

```

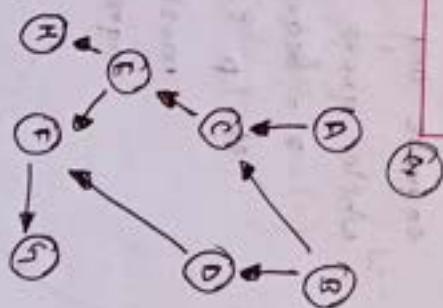
from collections import defaultdict

class Graph:
    def __init__(self, number_of_vertices):
        self.graph = defaultdict(list)
        self.number_of_vertices = number_of_vertices

    def addEdge(self, vertex, edge):
        self.graph[vertex].append(edge)

customGraph = Graph(6)
customGraph.addEdge('A', 'B')
customGraph.addEdge('A', 'C')
customGraph.addEdge('A', 'D')
customGraph.addEdge('B', 'E')
customGraph.addEdge('C', 'E')
customGraph.addEdge('C', 'F')
customGraph.addEdge('D', 'F')
customGraph.addEdge('E', 'G')
customGraph.addEdge('F', 'G')

```



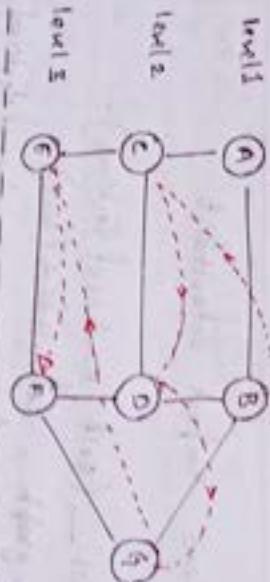
Graph Traversals

- Visiting all vertices in a given graph
- Breadth First search [BFS] Depth First search [DFS]

BFS

- Work with all 6 diff types of graphs

- BFS is an algorithm for traversing graph data structure
- It starts @ some arbitrary node of the graph and explores the neighbor nodes (which are @ current level) 1st, by moving to the next level neighbors.
- It basically uses the concept of level order traversal.



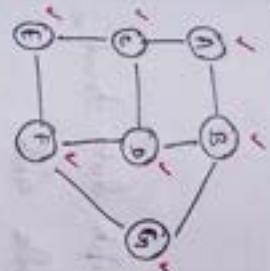
Algorithm

BFS

```
enqueue any starting vertex  
while queue is not empty  
    p = dequeue()  
    if p is unvisited  
        mark p visited  
        enqueue all adjacent unvisited vertices of p
```

how it works according to the algorithm:

(ii) - Queue



P = A

Queue : ~~A B C~~

P = A B

Queue : ~~A B C D~~

P = A B C

Queue : ~~A B C D E~~

P = A B C D

Queue : ~~A B C D E F~~

P = A B C D E

Queue : ~~A B C D E F G~~

P = A B C D E F

Queue : ~~A B C D E F G H~~

P = A B C D E F G

Queue : ~~A B C D E F G H I~~

P = A B C D E F G H

Queue : ~~A B C D E F G H I J~~

P = A B C D E F G H I

Queue : ~~A B C D E F G H I J K~~

P = A B C D E F G H I J

Queue : ~~A B C D E F G H I J K L~~

```
graph = Graph(customDict)
graph.bfs("a")
```

∴ All the vertices are now visited.

```
def bfs(self, vertex):
    visited = [vertex]
    queue = [vertex]
    while queue:
        deVertex = queue.pop(0)
        print(deVertex) FIFO
        for adjVertex in self.gdict[deVertex]:
            if adjVertex not in visited:
                visited.append(adjVertex)
                queue.append(adjVertex)
```

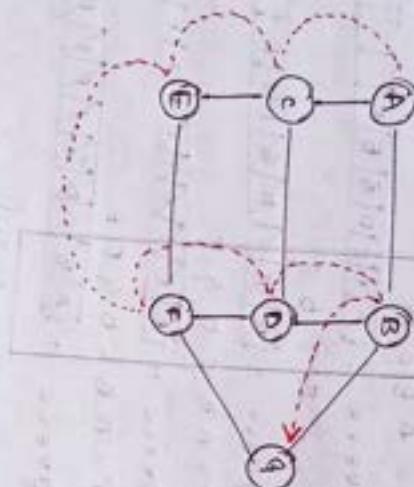
★ TC - O(m+n)

★ SC - O(m+n)

(m is then n)

Df

- Dfs is an algorithm for traversing a graph data structure
- It starts selecting some arbitrary node and explores all form of possible along each edge by backtracking

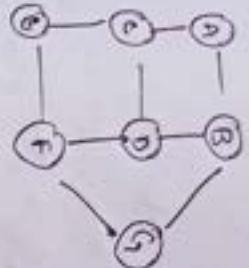


Algorithm

Dfs

```
push any starting vertex
while stack is not empty
    p = pop()
    if p is unvisited
        mark p visited
        push all adjacent unvisited vertices of p
```

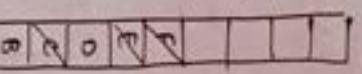
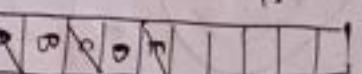
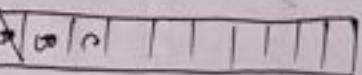
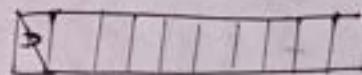
how it works according to the algorithm!



P = n

P, A, C

P, A, C, E



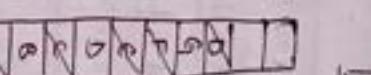
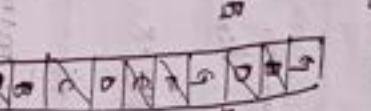
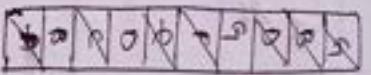
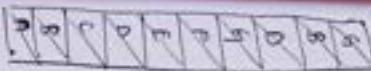
R = ACEFDOBq

R = ACEFDOB

R = ACEFD

R = ACEF

R = ACE



def dfs(self, vertex):

visited = [vertex]

stack = [vertex]

while stack:

popVertex = stack.pop()

print(popVertex)

for adjacentVertex in self.graph[popVertex]:

if adjacentVertex not in visited:

visited.append(adjacentVertex)

stack.append(adjacentVertex)

① TC - O(n+m)

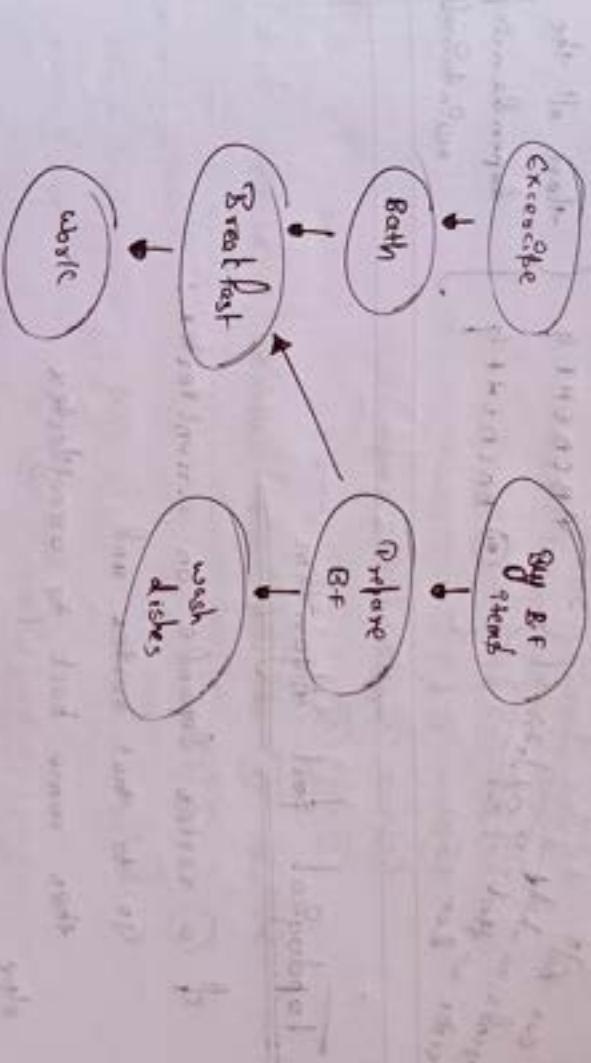
② SC - O(m+n)

BFS vs DFS

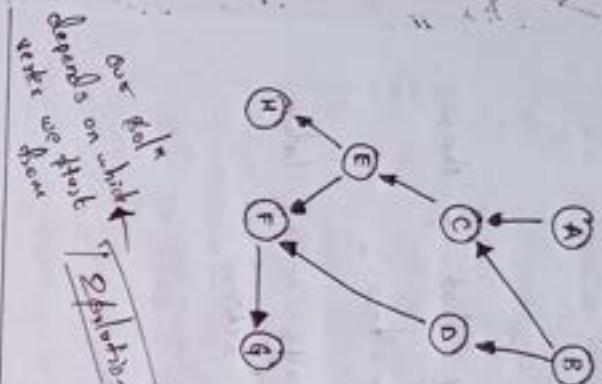
	BFS	DFS
How does it work internally?	It goes in breadth first	It goes in depth first
Which DS does it use internally?	queue	stack
Time complexity	$O(mn)$	$O(m+n)$
Space complexity	$O(n)$	$O(m+n)$
when to use?	If we know that the target is close to the starting point If we know that the target never is burned very deep.	If we already know that the target is far away from the starting point

Topological sort

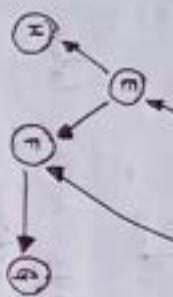
for a given actions in such a way that if there is a dependency of 1 action on another, then the dependent action always comes later than its parent action.



Example



if you want to do something here you should respect the dependencies of u e.g. F is dependent on A and B after C and D.



The sort could be something like:

the following depends on all the dependencies of one maintained.

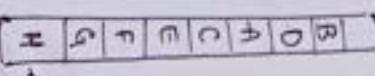
Topological Sort Algorithm

- if a vertex depends on current vertex:
 - ① go to that vertex and
 - ② then come back to current vertex
- else
 - ③ push current vertex to stack

How the algo. works !!

Let's start from \textcircled{A}

- ① Now since \textcircled{C} depends on unvisited pre \textcircled{A} , then acc to algo. $\textcircled{3}$ go to \textcircled{C} , now then \textcircled{E} then \textcircled{H} ; now \textcircled{H} is not dependent by other vertex). else condition ②



- # Now acc to $\textcircled{3}$ go to \textcircled{E} then $\textcircled{4}$ also dependent by \textcircled{F} then \textcircled{G} then $\textcircled{2}$
- then $\textcircled{G} \rightarrow$ pre $\textcircled{F} \rightarrow \textcircled{2}$
- then $\textcircled{3} \rightarrow$ pre $\textcircled{E} \rightarrow \textcircled{2}$
- then $\textcircled{3} \rightarrow$ pre $\textcircled{C} \rightarrow \textcircled{2}$
- then $\textcircled{3} \rightarrow$ pre $\textcircled{B} \rightarrow \textcircled{2}$

- # Now will move to \textcircled{B} then $\textcircled{8}$ —
 $\textcircled{B} \rightarrow \textcircled{1} \rightarrow \textcircled{D} \rightarrow \textcircled{1} \rightarrow \textcircled{F}$
 but \textcircled{F} is already visited

∴ $\textcircled{1} \rightarrow \textcircled{2}$
 then $\textcircled{3} \rightarrow \textcircled{1} \rightarrow \textcircled{2}$

Now the result of $\boxed{\text{BOA CEFGH}}$ & all dependencies are respected.

There are multiple results

The no. of results — depends on — no. of vertices

(d)

and our sol's will be diff based on the start-vertex

using stack



```
def topologicalSort(self, v, visited, stack):
    visited.append(v)
    for i in self.graph[v]:
        if i not in visited:
            self.topologicalSort(i, visited, stack)

stack = []
stack.append(0)
topologicalSort(0, stack)
```

(i) stack.append(0)

```
def topologicalSort(self):
```

```
visited = []
stack = []
```

for i in rest(self.graph):

```
if i not in visited:
    self.topologicalSort(i, visited, stack)
```

print(stack)

```
customGraph = Graph()
```

```
customGraph.addEdge("A", "C")
```

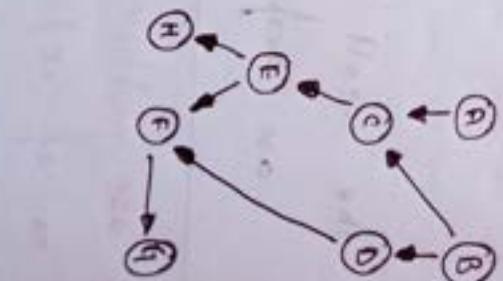
```
customGraph.addEdge("C", "E")
```

```
customGraph.addEdge("C", "D")
```

```
customGraph.addEdge("D", "B")
```

```
customGraph.addEdge("D", "F")
```

```
customGraph.addEdge("F", "G")
```



```
customGraph.topologicalSort()
```

graph

S_{ingle} S_{ource} S_{hortest} P_{ath} P_{roblem}

- A **problem** of about finding a path b/w a given vertex (called source vertex) to all other vertices in a graph such that the total distances b/w them (source & destination) is minimum.

ways of solving

BFS
Breadth First Search

Dijkstra's
Algorithm

Bellman
Ford

BFS for Graph

- ★ The only additional point is that we have to keep a track of the parent vertex.
- Everything else is same as that of BFS of tree problem.

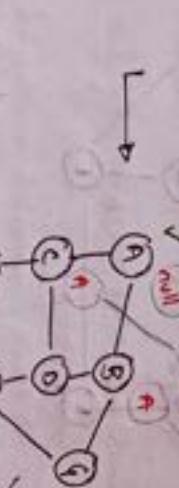
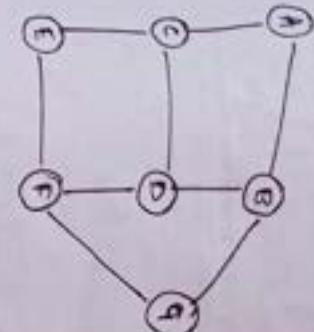
BFS for Graph - Algorithm

BFS

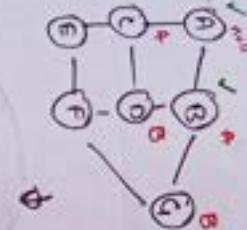
- enqueue any starting vertex
- while queue is not empty
 - p = dequeue()
 - if p is unvisited
 - mark p as visited
 - enqueue all adjacent unvisited vertices of p
 - update parent of adjacent vertices to current x

How the algorithm works !!

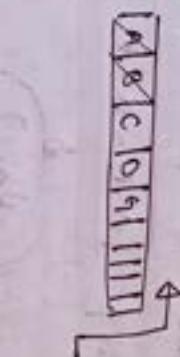
Let's start from A



$$P = A B$$



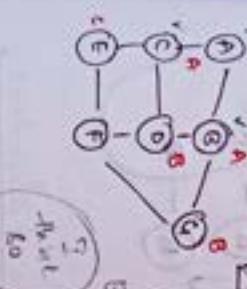
$$P = A B C$$



$$P = A B C D$$

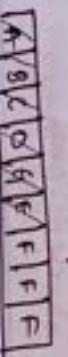


$$\emptyset$$

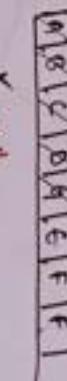


(No. of
nodes = 6)

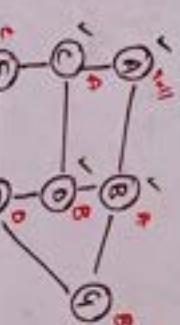
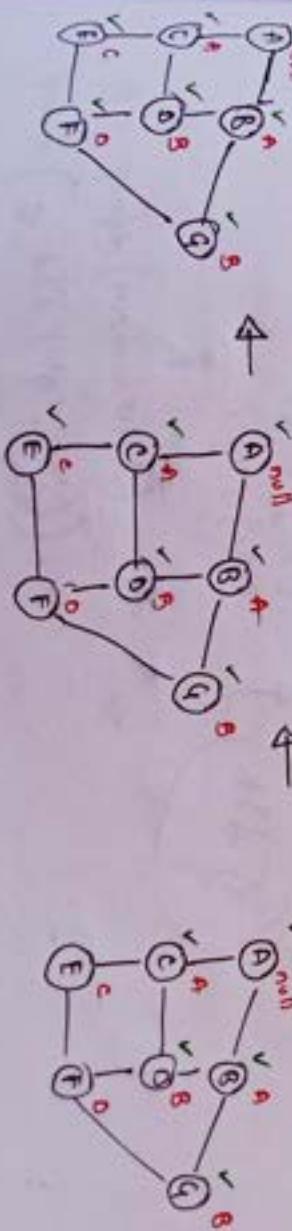
$$P = A B C D E F$$



$$P = A B C D E$$



$$P = A B C D E$$

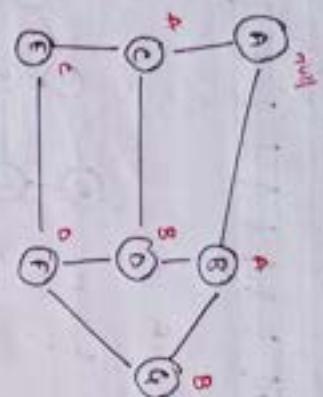


$$\emptyset$$

No wrong
order
of C, u write consider
only E not D
previously u hv mentioned that
② parent & ④)

No wrong
order
of C, u write consider
only E not D
previously u hv mentioned that
② parent & ④)

So on the end we got the graph from they take



Note if u want to find the shortest path by u

$$A \rightarrow F \rightarrow D \rightarrow B$$

How?

look @ the parent of F

now look @ the parent of D

is it A or F



Here if u hv considered A as the source vertex
u can't short the path from any other vertex u
won't get the shortest path.

5 work

o In case of SSSP
we'll only visit
the conn. vertices,

```
cListt Graph:  
def __init__(self, newdict = None):  
    if newdict == None:  
        newdict = {}  
    self. newdict = newdict
```

```
def bfs(self, start, end):
```

```
queue = []
```

```
queue.append([start])
```

```
while queue:
```

```
path = queue.pop(0)
```

```
node = path[-1]
```

```
if node == end:
```

```
return path
```

```
for adjacent in self. newdict.get(node, []):
```

```
newPath = list(path)
```

```
newPath.append(adjacent)
```

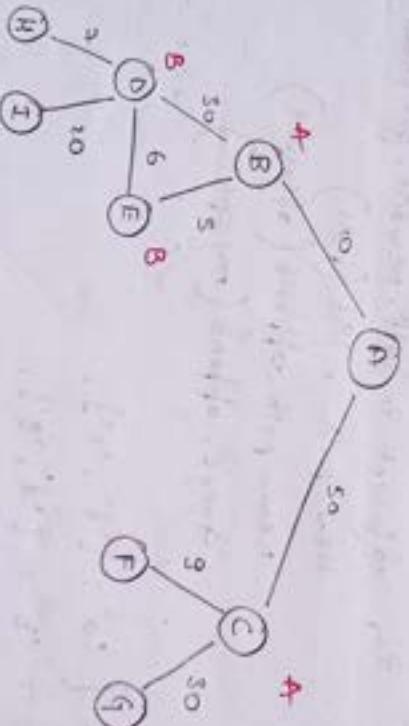
```
queue.append(newPath)
```

```
customdict = { 'a': ['b', 'c'],  
              'b': ['d', 'e'],  
              'c': ['d', 'e'],  
              'd': ['f'],  
              'e': ['f'],  
              'f': ['g'] }
```

```
g = Graph(customdict)  
g.bfs('a', 'e')
```

Why BFS is not Working for ~~graph~~

Graph Type	BFS
Unweighted - undirected	OK
Unweighted - directed	OK
Positive - weighted - undirected	X
Positive - weighted - directed	X
Negative - weighted - directed	X
- - - undirected	X



say ~~as~~ if u want to go from $\textcircled{A} \rightarrow \textcircled{D}$
 the path is $\boxed{\textcircled{A}\textcircled{B}\textcircled{D}}$ but it will take $30 + 10 = 40$ units
 but if u take the path $\boxed{\textcircled{A}\textcircled{B}\textcircled{E}\textcircled{D}}$ will take 21 units
 so BF doesn't work for weighted one's.

Why does Depth First Search not work with graphs

DFS has the tendency to go "as far as possible" from source, hence it can never find "shortest path"



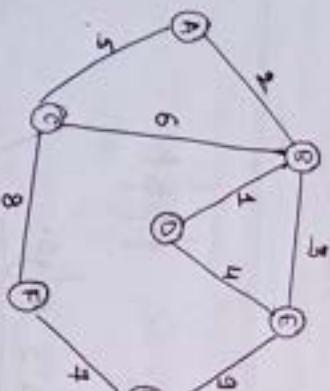
to solve weighted graph or if we go for Dijkstra's algo.

Dijkstra's Algorithm for Graph

An algorithm to find shortest path
between 2 vertices.

Used for [Weighted] graph.

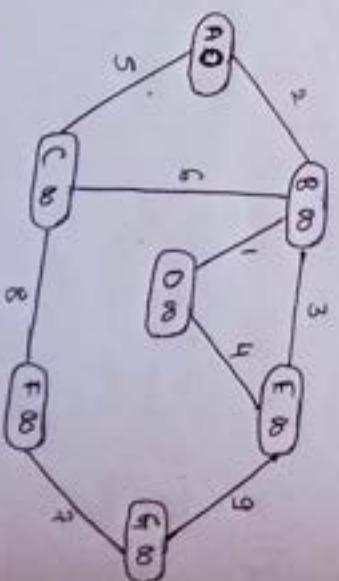
$\log n$



- The 1st step is to set initial values at each point.
- The starting point is always zero.

- The other points are set to infinity.

If we start from A →



- + Once we start from A, we have B & C as our next move.
 - + The method of calculating our next move/vertex is
- ~~calculated cost = The current vertex/points cost + cost of moving to that candidate vertex.~~

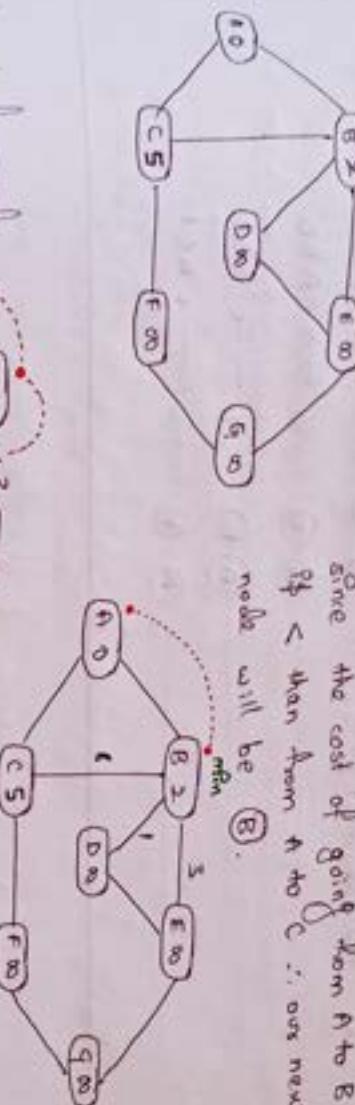
$$\begin{array}{l} \text{for } B: \quad \text{②} \quad \text{⑦} \\ \text{for } C: \quad \text{③} \quad \text{④} \end{array}$$

$$\begin{array}{l} \text{②} = 0 + 2 = 2 \\ \text{③} = 0 + 5 = 5 \end{array}$$

$\left\{ \begin{array}{l} \text{if the calculated cost of} \\ \text{less than the current} \\ \text{node's cost then will} \\ \text{update the current node} \\ \text{cost also will not update} \end{array} \right.$

∴ to calculate ② \rightarrow $2 < 8$ so for B & 5 cost for C is will update them

Since the cost of going from A to B is less than from A to C ∴ our next node will be B.

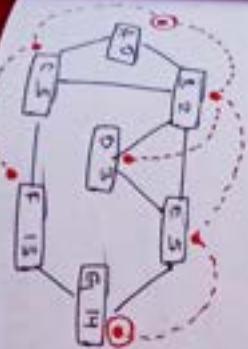
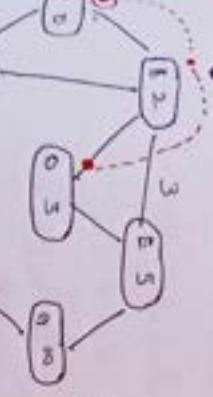


The cost of going from
C → ⑤ is 7
Since we need to reach
all objective nodes
∴ choose ⑤ → C

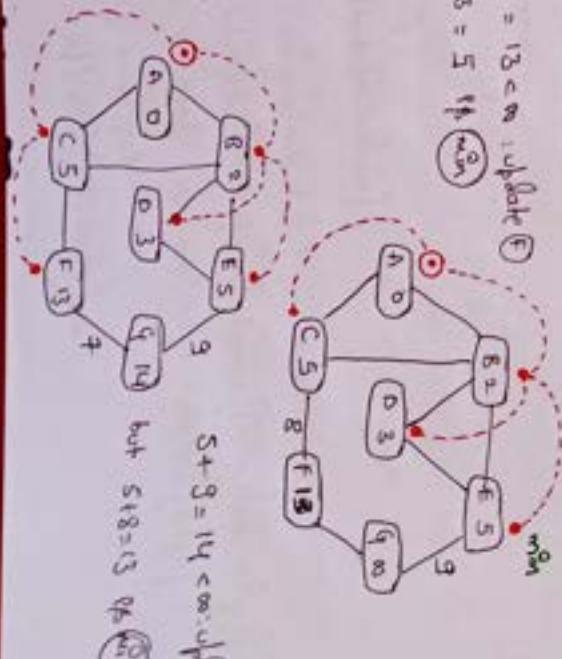
$2 + 6 = 8 < 5$: don't update C
 $2 + 1 = 3 < 8$: update D
 $2 + 3 = 5 < 8$: update E
and the min. path of ⑤ → D

$$5 + 3 = 13 < 8 \text{ : update } \textcircled{5}$$

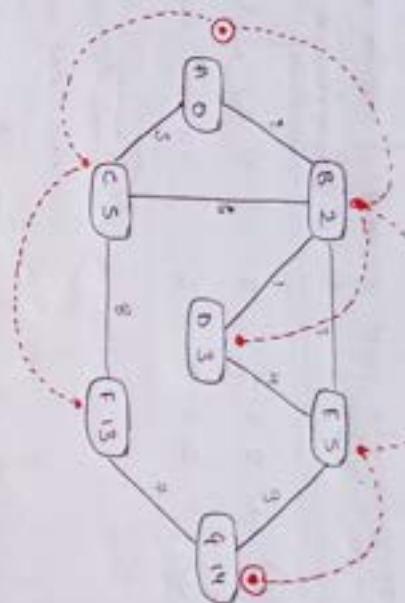
$$\text{but } 2 + 3 = 5 < 14 \text{ (NO)}$$



$$\begin{array}{l} 5 + 3 = 14 < 8 \text{ : update } \textcircled{5} \\ \text{but } 2 + 3 = 5 < 10 \text{ (NO)} \\ 13 + 4 = 17 \end{array}$$



The final graph for part 1:



Also shortest path from A to G is → A B E G

- (A) → A B D
- (B) → A B E
- (C) → A C F

from collections import defaultdict

graph = defaultdict(list)

def __init__(self):

self.nodes = set()

self.edges = defaultdict(dict)

self.distances = {}

def addNode(self, value):

self.nodes.add(value)

def addEdge(self, fromNode, toNode, distance):

self.edges[fromNode].append(toNode)

self.distances[(fromNode, toNode)] = distance

def dijkstra(graph, init):

visited = {initial: 0}

path = defaultdict(list)

nodes = set(graph.nodes)

while nodes:

minNode = None

for node in nodes:

if node in visited:

if minNode is None:

minNode = node

elif visited[node] < visited[minNode]:

minNode = node

if minNode is None:

break

nodes.remove(minNode)

currentWeight = visited[minNode]

for edge in graph.edges[minNode]:

weight = currentWeight + graph.distance[(minNode, edge)]

if edge not in visited or weight < visited[edge]:

visited[edge] = weight

path[edge] = append(minNode)

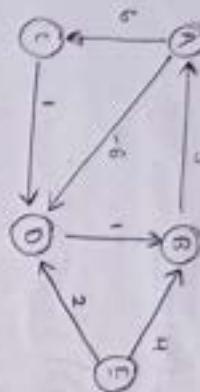
return visited, path

class Dis
out again

Dijkstra's algorithm with Negative edge

We cannot find a -ve cycle on a graph using Dijkstra.

(1)



Path from A to B = $6 + 1 + -5$

$$= -5 + 3 + (-6) + 1 = -9$$

1. If kept 4 on
2. not 11 if on a loop
and with an initial
loop.

To solve negative weighted cycle graph we
use Bellman-Ford Algorithm

An Overview On Graphs

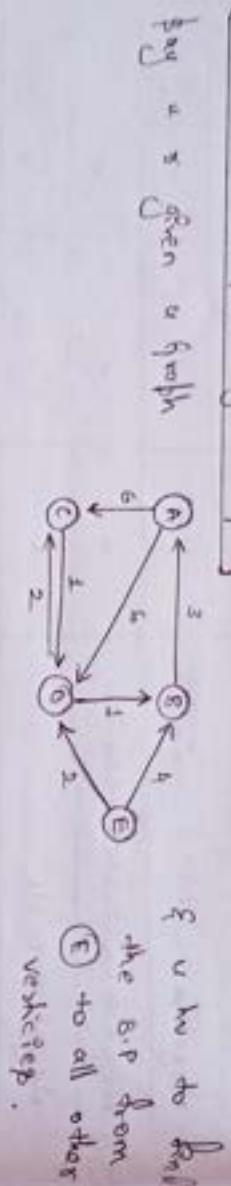
Graph Type	BFS	Dijkstra's	Bellman Ford
Unweighted - undirected	✓	✓	✓
Unweighted - Directed	✓	✓	✓
Weighted - undirected	✗	✓	✓
Weighted - directed	✗	✓	✓
Negative - weighted - undirected	✗	✓	✓
Negative - weighted - Directed	✗	✓	✓
Negative cycle	✗	✗	✓

Bellman Ford Algorithm

Bellman Ford algorithm is used to find TSP.

- If there is a -ve cycle it catches
- & report its existence.

How Bellman Ford Algo. works



• u is to find
the B.P from
E to all other
vertices.

Q. Let's see how B.F. algo. works for the cycle then
will see for -ve cycle.

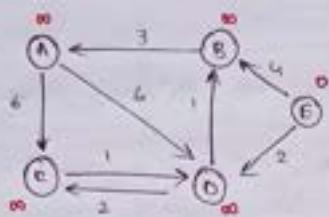
We have to run n iteration [Vertex - 1] times.

B.F. algo. works the same way as Dijkshtra's algo. works
but with a small difference.

If the distance of destination vertex > (distance of source vertex +
weight b/w source & destination vertex):
update distance of destination vertex to (distance of source
vertex + weight b/w source & destination vertex)

1st step: let the source vertex be $E = 0$
 $\&$ all others = ∞

check the (condition) (as we did in DjAlg.) to go in the order for
 the following edges given b/l.
 Above we Don't sort w.r.t. the concept of min. (as did in DjAlg.)
 we go in the order mentioned b/l.

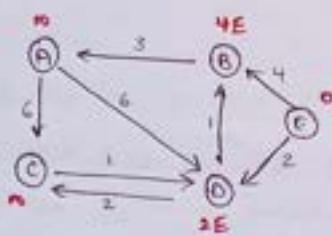


edge	weight
A-C	6
A-D	6
B-E	4
C-D	1
D-C	2
D-B	1
E-B	4
E-D	2

Distance Matrix	
Vertex	Distance
A	∞
B	∞
C	∞
D	∞
E	0

so using the condition

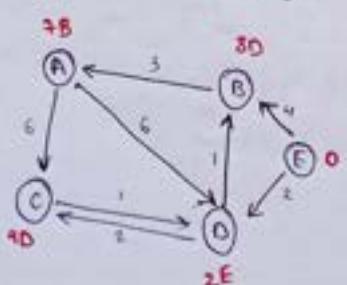
edge
 A-C —> $0 + 6 = 6 < \infty$ no update
 A-D —>
 B-A —> $0 + 2 = 2 < \infty$ no update
 C-D —>
 D-C —>
 D-B —>
 E-B —> $0 + 4 = 4 < \infty$ update B with parent
 F-D —> $0 + 2 = 2 < \infty$ update D with parent



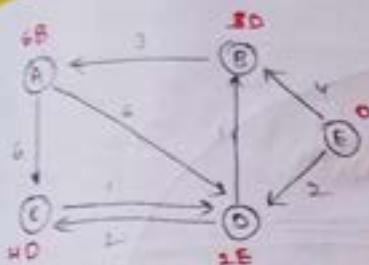
edge	weight
A-C	6
A-D	6
B-E	4
B-D	1
C-D	1
D-C	2
D-B	1
E-B	4
E-D	2

Distance Matrix			
Vertex	Distance	Iteration 1	
		Distance	Parent
A	0	00	-
B	0	4	E
C	0	00	-
D	0	2	E
E	0	0	-

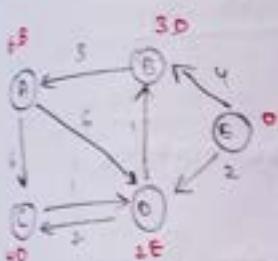
Again applying the same condition ; for edges in same order



Vertex	Distance	Iteration 1		Iteration 2	
		Distance	Parent	Distance	Parent
A	0	00	-	4+3=7	B
B	0	4	E	2+1=3	D
C	0	00	-	2+1=4	D
D	0	2	E	2	E
E	0	0	-	0	-



Vertex	Distance	Distance Matrix					
		Iteration 1 Distance	Iteration 1 Parent	Iteration 2	Iteration 3		
A	00	00	-	$4+3=7$	B	$3+3=6$	E
B	00	4	E	$2+1=3$	D	3	D
C	00	00	-	$2+1=3$	D	4	D
D	00	2	E	2	E	2	E
E	0	0	-	0	-	0	-



Distance Matrix

Vertx	Dist	Distance Matrix			
		I-1	I-2	I-3	I-4
A	00	00	00	00	00
B	00	4	2	B	6
C	00	10	-	3	3
D	00	2	E	0	4
E	0	0	-	0	0

Final Solution

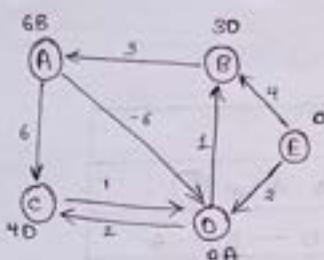
Vertx	Distance from E	Path from E
A	6	E - D - B - A
B	3	E - D - B
C	4	E - D - C
D	2	E - D
E	0	0

Bellman Ford Algorithm with negative cycle

The concept remaining the same :

If the distance of destination vertex > (distance of source vertex + weight between source and destination vertex) :

update distance of destination vertex to (distance of source vertex + weight between source and destination vertex).



EDGE	WEIGHT
A-C	6
A-D	-5
B-A	3
C-D	1
D-C	2
D-B	1
E-B	4
E-D	2

Distance Matrix

Vertex	Distance	Iteration 1		2 2		3 3		3 4		3 5	
		Distance	Parent	D	P	D	P	D	P	D	P
A	68	00	-	$4+3=7$	B	$3+3=6$	B	0	P	D	P
B	30	4	E	$2+1=3$	D	3	D	6	B	4	B
C	40	00	-	$2+2=4$	D	4	D	3	D	0	D
D	00	2	E	2	E	$7+(-6)=1$	A	$6+(-6)=0$	A	0	D
E	0	0	-	0	-	0	-	0	-	-	-

it actually make go till 4 iterations only. \rightarrow (vertex-1)

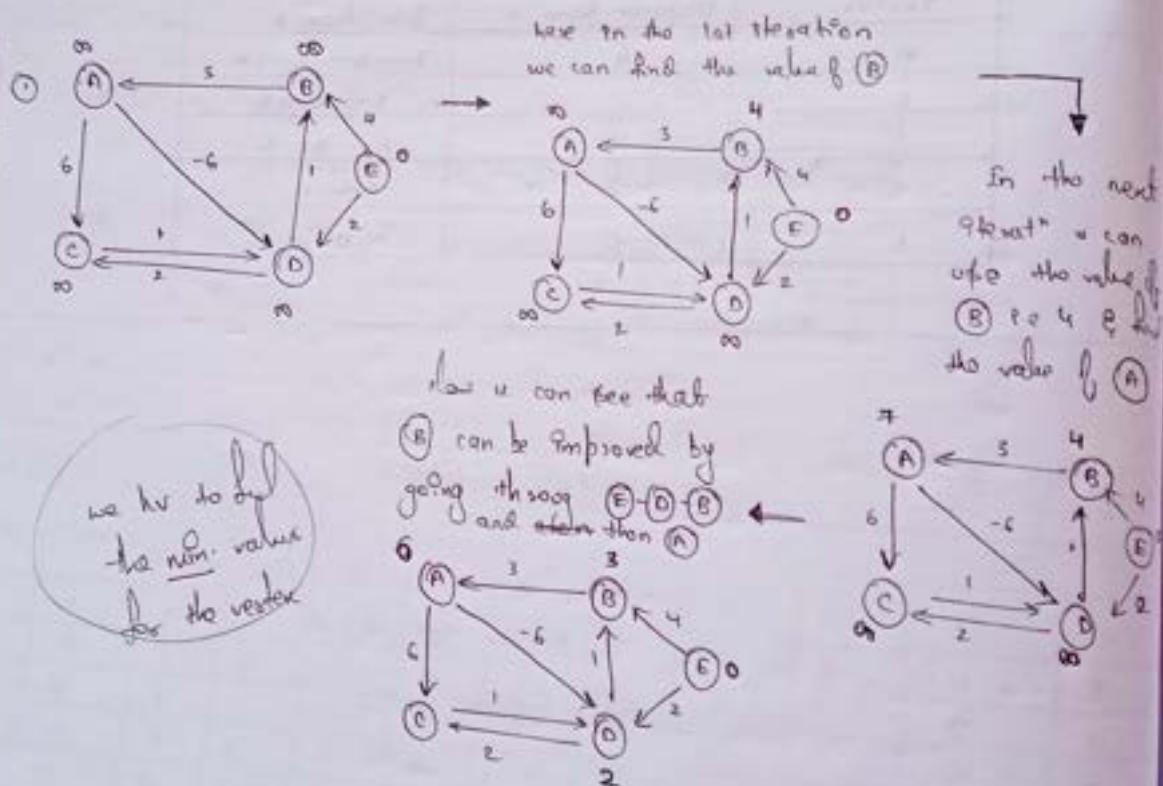
But if we go for the 5th iteration and if any vertex weight is changed then we know that we have a negative iteration.

so our final form will look like this
as we are not going till 5th iteration.

FINAL SOLUTION		
Vertex	Distance from E	Path from E
A	6	E—D—B—A
B	3	E—D—B
C	4	E—D—C
D	2	E—D
E	0	0

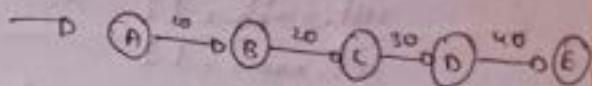
Why Bellman Ford algo. runs $V-1$ times?

- ① If any node gets achieved better distance in previous iteration, then that better distance is used to improve distance of other vertices.
- ② Identify worst case graph that can be given to us.
- ③ Processing of edge might happen in unfavourable way.



Q) worst case of graph if the max. distance b/w the source and the destination

If we change the Graph in a simpler format



be the max. of 4 edges.

$$\leq \boxed{V-1}$$

- Q) say u hv a graph like
-
- ```
graph LR; A((A)) -- "10" --> B((B)); B -- "20" --> C((C)); C -- "30" --> D((D))
```
- if u wanna find the shortest Path by Bellman Ford Algo.
- o let will set source = 0 & others =  $\infty$
  - o then u can see that C can be updated only after the 1st iteration.
  - o & B & A cannot be updated unless we update # C @ 1st iterations.
  - o so the no. of iterations = 3 =  $\boxed{V-1}$

=o in  $[V-1]$  iteration we can find the fastest solution.

=o that in  $V$ th iterat<sup>n</sup> nothing should change in terms of distances and vertices.

But if the distances changes then the Bellford Algo. understand that it's a negative condition.

### class Graph:

```
 def __init__(self, vertices):
 self.V = vertices
 self.graph = []
 self.nodes = []

 def add_edge(self, s, d, w):
 self.graph.append([s, d, w])

 def add_node(self, value):
 self.nodes.append(value)

 def print_solution(self, dist):
 print("Vertex Distance from Source")
 for i in self.nodes:
 print(i, ":", dist[i])

 def bellman_ford(self, src):
 dist[src] = 0
 for i in range(len(self.nodes) - 1):
 for v in range(self.V):
 for s, d, w in self.graph:
 if dist[s] != float('inf') and dist[s] + w < dist[d]:
 dist[d] = dist[s] + w

 for v in range(self.V):
 for s, d, w in self.graph:
 if dist[s] != float('inf') and dist[s] + w < dist[d]:
 print("Graph contains negative cycle")
 return

 self.print_solution(dist)
```

• TC —  $O(V^2)$   
• SC —  $O(V)$

g = Graph()

g.add\_node("n")

A

B

C

D

E

F

g.add\_edge("n", "c", c)

|  |   |   |   |
|--|---|---|---|
|  | A | 0 | 6 |
|  | B | 0 | 3 |
|  | C | 0 | 1 |
|  | D | 0 | 2 |
|  | E | 0 | 1 |
|  | F | 0 | 4 |
|  |   |   |   |
|  | E | 0 | 2 |

g.bellmanFord("E")

Vertex Distance from source

|   |   |    |
|---|---|----|
| A | : | 6  |
| B | : | 3  |
| C | : | 4  |
| D | : | 10 |
| E | : | 0  |

list[ ]

[ ]:

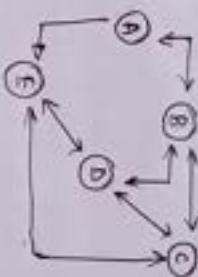
# Diff v/s Depth Bellman Ford

|                  | SP                                                                | DJ                                                                  | B.Ford                                                       |
|------------------|-------------------------------------------------------------------|---------------------------------------------------------------------|--------------------------------------------------------------|
| T.C.             | $O(n^2)$                                                          | $O(n^2)$                                                            | $O(nE)$                                                      |
| D.C.             | $O(\epsilon)$                                                     | $O(n)$                                                              | $O(n)$                                                       |
| Implementation   | Easy                                                              | Moderate                                                            | Moderate                                                     |
| Limitation       | Not work for weighted graph<br>Not work for negative weight graph | Not work for negative weight graph                                  | Not work for negative weight graph                           |
| Unweighted Graph | OK                                                                | OK                                                                  | OK                                                           |
| Weighted Graph   | Use D.J. as it's more efficient than SP and easy to implement     | Not up to date implementation<br>Not up to date as T.C. is not good | Not up to date as T.C. is not good as T.C. is not up to date |
| Weighted Graph   | X                                                                 | Not up to date                                                      | OK                                                           |
| Negative Cycle   | Not supported                                                     | Not supported<br>Use D.J. as other                                  | Not supported<br>Not supported                               |

# All pair shortest path problem

M.P.S.P.P. is about finding a path b/w every vertex to all other vertices in a graph, such that the total distance b/w them [source and destination] is minimum.

Ex - In a graph like this



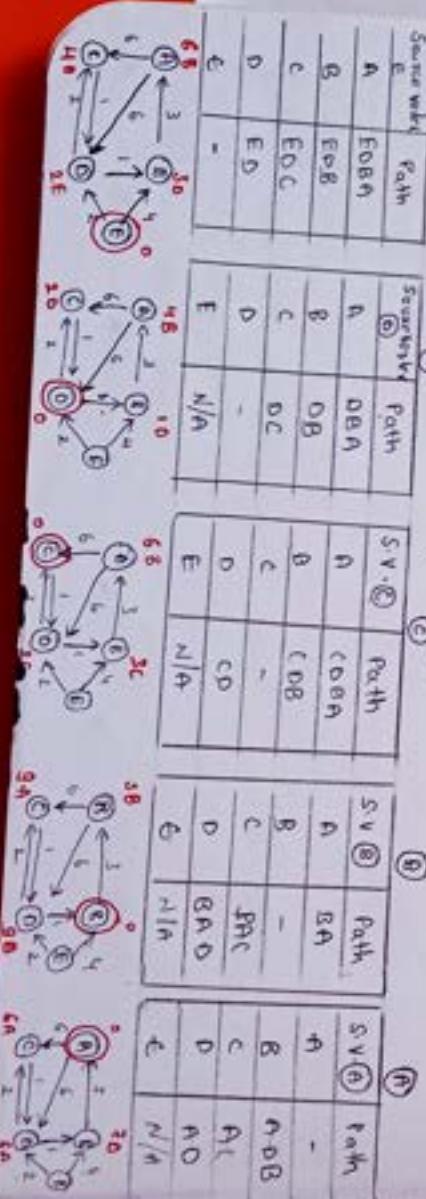
Just we to find the short single source shortest path, P for each vertex individually [i.e. (A), (B), (C), (D), (E)], dubbing them would give to A.P.S.P.P.

In - short

Dry run for all pair shortest path problem:

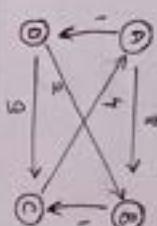
By using any 1 of the 3 algo. i.e BFS, Dijkstra's or Bellman Ford algo. compute single source shortest path - Problem for each vertex , considering each vertex as source vertex.

( $\rightarrow$  no - need application's algo here)



# Floyd - Warshall Algo

→ U can use this algo. to find all pairs shortest path  
for u we given a graph like →



### Algorithm

$$s.t. d[u][v] \geq d[u][v_{\text{max}}] + d[v_{\text{max}}][v] :$$

$$d[u][v] = d[u][v_{\text{max}}] + d[v_{\text{max}}][v]$$

$d_{ij}$  denotes dist. b/w vertex i & j.  
Some iteration value  
and  $d_{ij} \leftarrow \infty$  from  $a-a-c$   
the  $d_{ij}$  [via x] were

Let u create a matrix like this →

|   | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 8 | 9 | 1 |
| B | 0 | 0 | 1 | 8 |
| C | 4 | 0 | 0 | 8 |
| D | 8 | 1 | 9 | 0 |

Now u perform  $(V \times V)$  where x  $\in$  each vertex for n iterations.

Iteration - 1

| VIA A | A | B | C | D |
|-------|---|---|---|---|
| A     | 0 | 8 | 9 | 1 |
| B     | 0 | 0 | 1 | 8 |
| C     | 4 | 0 | 0 | 8 |
| D     | 8 | 1 | 9 | 0 |

Iteration - 2

| VIA B | A | B  | C  | D |
|-------|---|----|----|---|
| A     | 0 | 8  | 9  | 1 |
| B     | 0 | 0  | 1  | 8 |
| C     | 4 | 12 | 0  | 5 |
| D     | 8 | 2  | 13 | 0 |

Iteration - 3

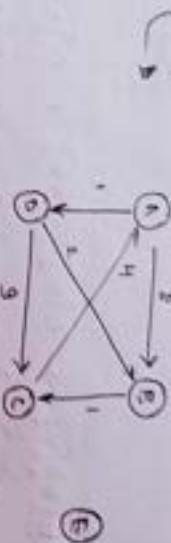
| VIA C | A | B  | C  | D |
|-------|---|----|----|---|
| A     | 0 | 8  | 9  | 1 |
| B     | 0 | 14 | 13 | 4 |
| C     | 4 | 12 | 0  | 5 |
| D     | 8 | 2  | 13 | 0 |

Iteration - 4

|   | A | B | C | D |
|---|---|---|---|---|
| A | 0 | 3 | 4 | 1 |
| B | 5 | 0 | 1 | 6 |
| C | 4 | 1 | 0 | 5 |
| D | 7 | 2 | 3 | 0 |

## Why Floyd Warshall Algorithm ?

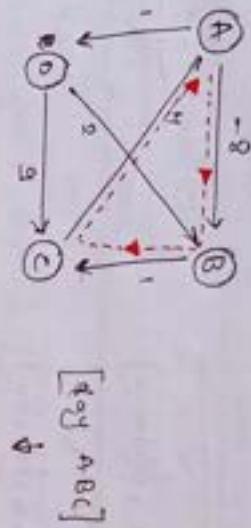
- If there's a vertex  $\oplus$   $\ominus$  which is not reachable then u cannot directly apply floyd warshall algo. for that



- 2 vertex vertices are directly connected then:
  - ↳ that is the best solution
  - ↳ but it can be improved via other vertex

- 2 vertices are connected via other vertex.

+ Floyd Warshall Algo. with Negative cycle:



- To go through cycle ; we need to go via negative cycle's participating vertex at least twice.  
Ex -  $\Delta$  -  $B$  -  $C$  -  $\Delta$  considering  $\Delta$  as the source vertex.
- Floyd Warshall runs NEVER until loop since via same vertex.
- Hence : Floyd Warshall can never detect a negative cycle

## Floyd Warshall in Python:

[Implementation details]

★ TC → O( $V^3$ )

★ SC → O( $V^2$ )

```
INF = 999
def printSolution(numVertices , distance):
 for i in range(numVertices):
 for j in range(numVertices):
 if distance[i][j] == INF:
 print("INF" , end = " ")
 else:
 print(" ")
```

```
def FloydWarshall(numVertices , graph):
 distance = graph
 for k in range(numVertices):
 for i in range(numVertices):
 for j in range(numVertices):
 if j in range(numVertices):
 distance[i][j] = min(distance[i][j] , distance[i][k] + distance[k][j])
 printSolution(numVertices , distance)
```

```
G = [[0, 2, INF, 1],
 [INF, 0, 1, 2INF],
 [4, INF, 0, 1INF],
 [INF, 0, 3, 1]]
```

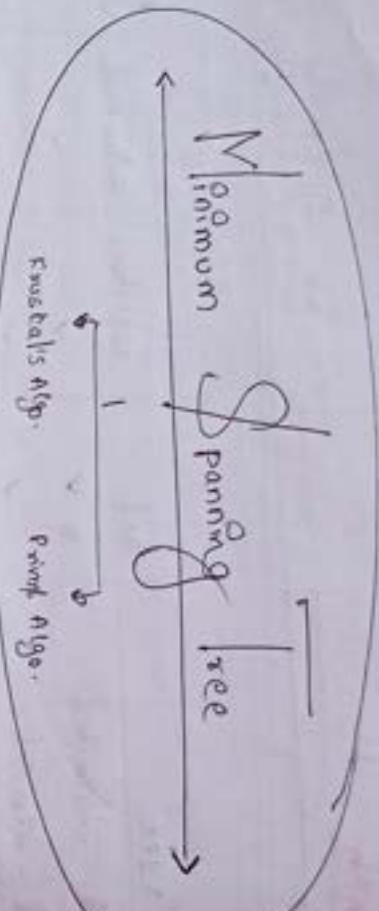
```
floydWarshall(4, G)
```

|   |   |   |   |
|---|---|---|---|
| 0 | 3 | 4 | 1 |
| 5 | 0 | - | 6 |
| 4 | 4 | 0 | 5 |
| 2 | 3 | 5 | 1 |

## Which Algorithm to use for App?

| GRAPH TYPE                     | BFS  | Dijkstra's | Bellman Ford | Floyd Warshall |
|--------------------------------|------|------------|--------------|----------------|
| Unweighted - undirected        | OK ✓ | ✓          | ✓            | ✓              |
| Unweighted - directed          | ✓    | ✓          | ✓            | ✓              |
| Positive-weighted - undirected | ✗    | ✓          | ✓            | ✓              |
| Positive-weighted - directed   | ✗    | ✓          | ✓            | ✓              |
| Negative-weighted - undirected | ✗    | ✓          | ✓            | ✓              |
| Negative-weighted - directed   | ✗    | ✓          | ✓            | ✓              |
| Negative cycle                 | ✗    | ✗          | ✓            | ✗              |

|                       | BFS                                   | Dijkstra                     | Bellman Ford          | Floyd Warshall                         |
|-----------------------|---------------------------------------|------------------------------|-----------------------|----------------------------------------|
| G.C                   | $O(V^2)$                              | $O(V^2)$                     | $O(VE)$               | $O(V^2)$                               |
| f.c                   | $O(EN)$                               | $O(EV)$                      | $O(V^2)$              | $O(V^2)$                               |
| Implementation        | easy                                  | Moderate                     | Moderate              | Moderate                               |
| (k)(f) Implementation | Not worst for weighted graph          | Not worst for negative cycle | N/A                   | Not worst for negative cycle           |
| Unweighted graph      | OK                                    | OK                           | OK                    | OK                                     |
| Weighted graph        | One that is TC good and can implement | Not use, as TC is bad        | Not use, as TC is bad | can be used                            |
| Weighted graph        | ✗                                     | OK                           | OK                    | OK                                     |
| Negative cycle        | Not supported                         | can be used                  | not use, as TC is bad | can be removed if implementation of TC |
| Negative cycle        | ✗                                     | ✗                            | OK                    | ✗                                      |



Fundamentals Alg. Point Alg.

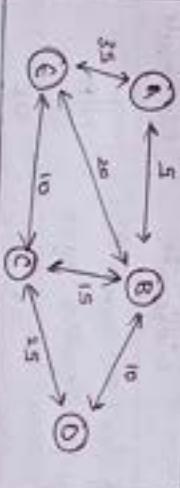
A MST is a subset of the edges of connected, weighted and undirected graph which;

- connects all vertices together
- No cycle
- Minimum total edge.

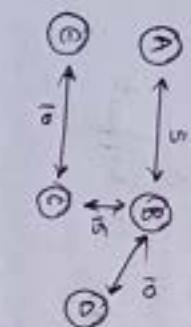
There's a diff b/w MST & Graph

Don't complete b/w them

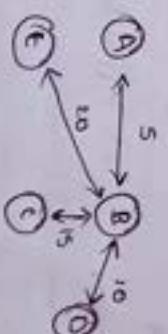
Say u have a problem like that



Solving this using



MST



Graph

Here in MST, unlike Dijk we don't have a source node and find the min. distance to all other vertex.

We connect all vertices together in such a way that the total edge weight minimum & there's no cycle.

In graph the total weight of graph is  $\rightarrow$  50  
The same in MST is  $\rightarrow$  40.

tot wt

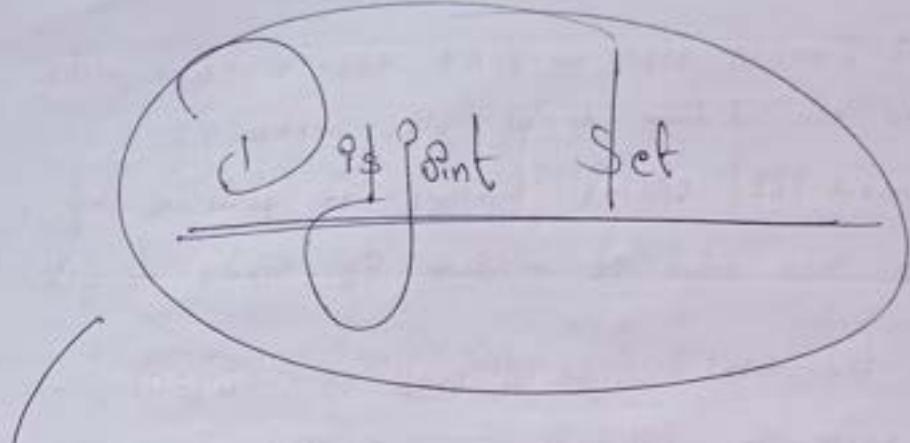
Dijk tot

least cost

all

se the

o. o.



It is a datastructure, that keeps track of set of elements which are partitioned into a number of disjoint & non-overlapping sets and each set has representative which helps in identifying that set.

3 standard operations → Make set  
Union set  
Find set

Say we're given 5 elements A,B,C,D,E & we're performing Disjoint set operations on these;

① Make set → using this we can make individual sets for the elements;

(A) (B) (C) (D) (E)

② Union set → using this we can merge 2 given sets.

union(A,B) → (AB) (C) (D) (E)

union(A,E) → (ABE) (C) (D)

③ Find set → Returns the set name in which this element is located.

Findset(B) → will return (AB)

Findset(E) → → (ABE)

class DisjointSet:

def \_\_init\_\_(self, vertices):

self.vertices = vertices

self.parent = {}

for v in vertices:

self.parent[v] = v

self.rank = dict.fromkeys(vertices, 0)

def find(self, item):

if self.parent[item] == item:

return item

else:

return self.find(self.parent[item])

def union(self, x, y):

xroot = self.find(x)

yroot = self.find(y)

if self.rank[xroot] < self.rank[yroot]:

self.parent[xroot] = yroot

elif self.rank[xroot] > self.rank[yroot]:

self.parent[yroot] = xroot

else:

self.parent[yroot] = xroot

self.rank[xroot] += 1

vertices = ["A", "B", "C", "D", "E"]

print(ds.find("A"))

ds.union("A", "B")

print(ds.find("B"))

ds.union("A", "C")

print(ds.find("A"))

★ TC → O(n)

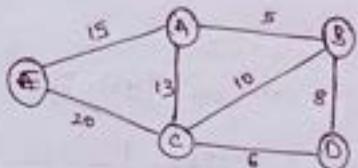
★ SC → O(n)

# KRUSKAL'S ALGORITHM

- ↳ It is a Greedy algorithm
- ↳ Used to solve MST
- ↳ It helps you to find Minimum Spanning Tree for weighted undirected graph in 2 ways:
  - ↳ Add increasing cost edges at each step
  - ↳ Avoid any cycle at each step.

logic?

Say you are given a graph something like



1st → we create individual sets

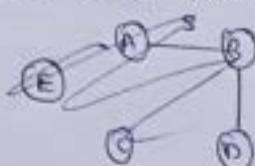
A  
B  
E  
C  
D

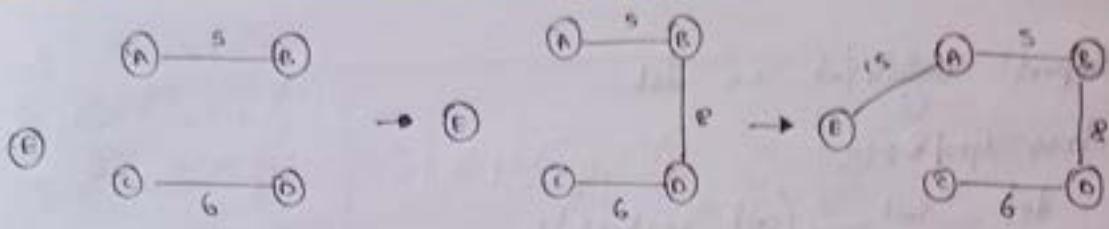
then we take the min. edge i.e. 5 here & check who,

A --- B  
E

C  
D

then we continuously link the min. edge along side check whether it's forming a loop; if it's then discard it





Discarding 10, 20, 15 ... it forms loop.

pseudocode:

Kruskal(G):

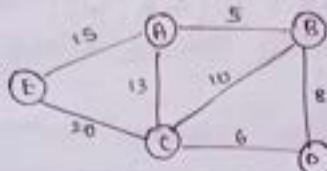
```

for each vertex: O(v)
 makeSet(v)

sort each edge in nondecreasing order by weight - O(ElogE)

for each edge (u, v): O(E)
 if findSet(u) ≠ findSet(v): O(1)
 union(u, v) O(v)
 cost = cost + edge(u, v) O(1)

```



TC → O(ElogE)

SC → O(V+E)

|                        |              |                                                |
|------------------------|--------------|------------------------------------------------|
| cost = 0               | (5 vertices) | (5 vertices)                                   |
| cost = 0+5             | (4 sets)     | (2 edges)                                      |
| cost = 0+5+6           | (3 sets)     | (2 edges)                                      |
| cost = 0+5+6+8         | (1 set)      | (3 edges)                                      |
| cost = 0+5+6+8+15 = 34 |              | (5 edges highlighted with a dashed red circle) |

This is the MST for the given graph.

import DisjointSet as dst

class Graph:

def \_\_init\_\_(self, vertices):

self.V = vertices

self.graph = []

self.nodes = []

self.MST = []

def addEdge(self, s, d, w):

self.graph.append([s, d, w])

def addNode(self, value):

self.nodes.append(value)

def printSolution(self, s, d, w):

for s, d, w in self.MST:

print("%s - %s : %d (%s, %d, %d)"

def kruskalAlgo(self):

q, e = 0, 0

ds, d

ds = dst.DisjointSet(self.nodes)

self.graph = sorted(self.graph, key=lambda item: item[2])

while e < self.V - 1:

s, d, w = self.graph[q]

q += 1

x = ds.find(s)

y = ds.find(d)

if x != y:

e += 1

self.MST.append([s, d, w])

ds.union(x, y)

self.printSolution(s, d, w)

$g = \text{Graph}(5)$

$g.\text{addNode}("A")$

$g.\text{addNode}("B")$

$g.\text{addNode}("C")$

$\rightarrow A$

$\rightarrow B$

$\rightarrow C$

$\rightarrow D$

$\rightarrow E$

$g.\text{addEdge}("A", "B", 5)$

$\rightarrow A \quad C \quad 13$

$\rightarrow A \quad E \quad 15$

$\rightarrow B \quad A \quad 5$

$\rightarrow B \quad C \quad 10$

$\rightarrow B \quad D \quad 8$

$\rightarrow C \quad A \quad 13$

$\rightarrow C \quad B \quad 10$

$\rightarrow C \quad E \quad 20$

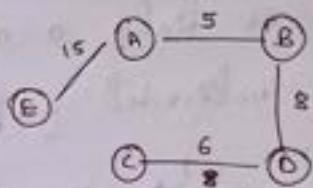
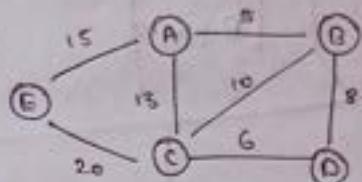
$\rightarrow C \quad D \quad 6$

$\rightarrow D \quad B \quad 8$

$\rightarrow D \quad C \quad 6$

$\rightarrow E \quad A \quad 15$

$\rightarrow E \quad C \quad 20$



$g.\text{bfsalgo}$

$A - B : 5$

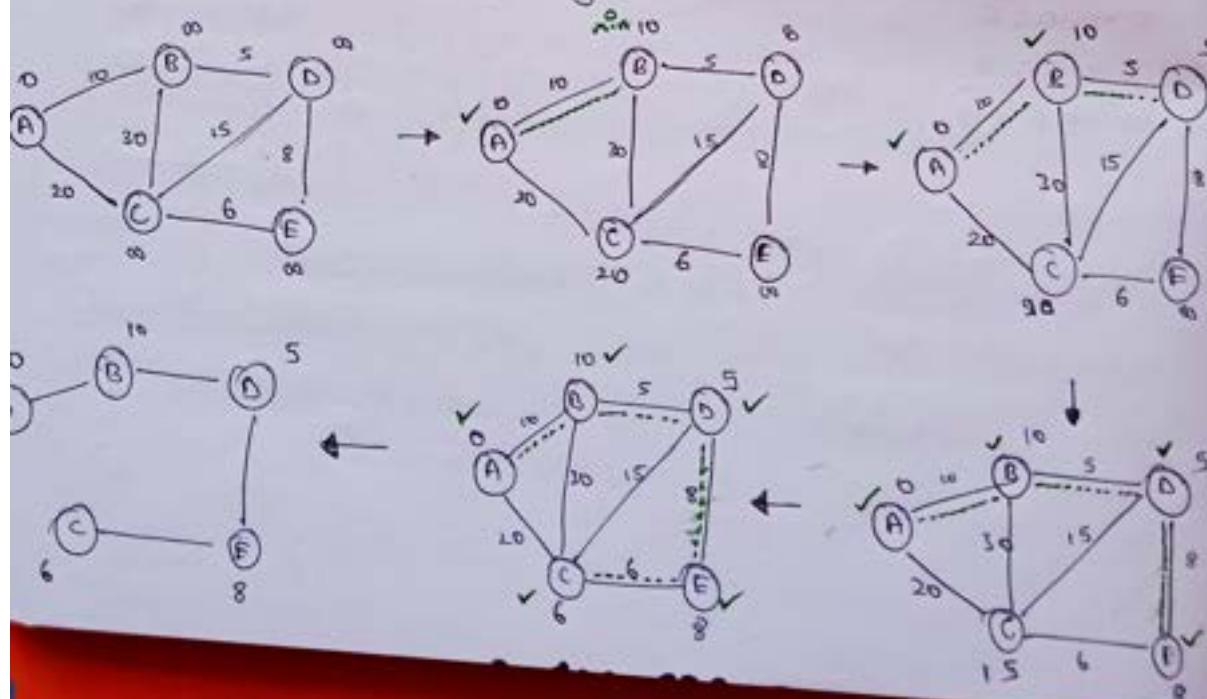
$C - D : 6$

$B - D : 8$

$A - E : 15$

# PRIMS ALGORITHM

- It is a Greedy Algorithm.
- It finds a minimum spanning tree for weighted undirected graphs in following way:
  - ↳ Take any vertex of a choice set off weight to 0 & all other vertices' weight to  $\infty$
  - ↳ For every adjacent vertex if the current weight is more than current edge then we set it to current edge, & mark visited
  - ↳ After marking visited Mark current vertex visited
  - ↳ After marking current vertex visited, among the many adjacent edges/vertices choose the min. one for the n of the next path.
  - ↳ Repeat these steps for all vertices in increasing order of weight.



```
import sys
```

```
class Graph:
```

```
 def __init__(self, vertexNum, edges, nodes):
```

```
 self.edges = edges
```

```
 self.nodes = nodes
```

```
 self.vertexNum = vertexNum
```

```
 self.MST = []
```

```
 def printSolution(self):
```

```
 print("Edge : Weight")
```

```
 for s,d,w in self.MST:
```

```
 print("%s -> %s : %d" % (s,d,w))
```

```
 def primsAlgo(self):
```

```
 visited = [0] * self.vertexNum
```

```
 edgeNum = 0
```

```
 visited[0] = True
```

```
 while edgeNum < self.vertexNum - 1: ----- o(v)
```

```
 min = sys.maxsize
```

```
 for i in range(self.vertexNum): ----- o(v)
```

```
 if visited[i]:
```

```
 for j in range(self.vertexNum): ----- o(v)
```

```
 if ((not visited[j]) and self.edges[i][j]):
```

```
 if min > self.edges[i][j]:
```

```
 min = self.edges[i][j]
```

```
 s = i
```

```
 d = j
```

```
 self.MST.append([self.nodes[s], self.nodes[d], self.edges[s][d]])
```

```
 visited[d] = True
```

```
 edgeNum += 1
```

```
edges = [[0, 10, 20, 0, 0],
```

```
 [10, 0, 30, 5, 0],
```

```
 [20, 30, 0, 15, 6],
```

```
 [0, 5, 15, 0, 8],
```

```
 [0, 0, 6, 8, 0]]
```

```
nodes = ["A", "B", "C", "D", "E"]
```

$g = \text{Graph}(5, \text{edges}, \text{nodes})$

$g.\text{primsAlgo}()$

edge : weight

A → B : 0

C → D : 5

B → E : 2

E → C : 6

① TC → O(v<sup>2</sup>)

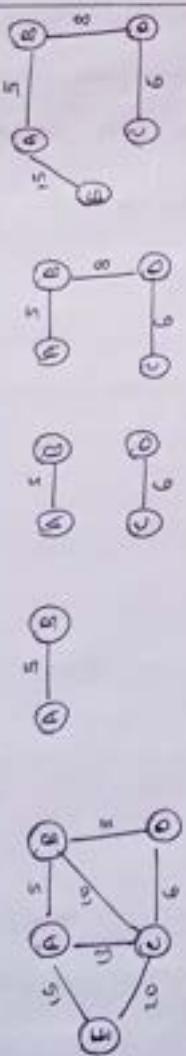
② SC → O(v)

# KRUSKAL      v/s      PRIM

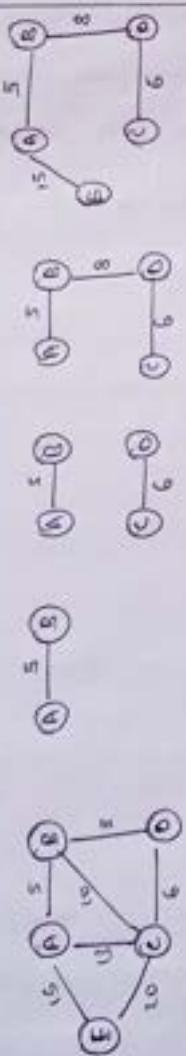
## Applications:

- concentrates on edges
- finalize edge in each iteration

Kruskal's  
Algo.

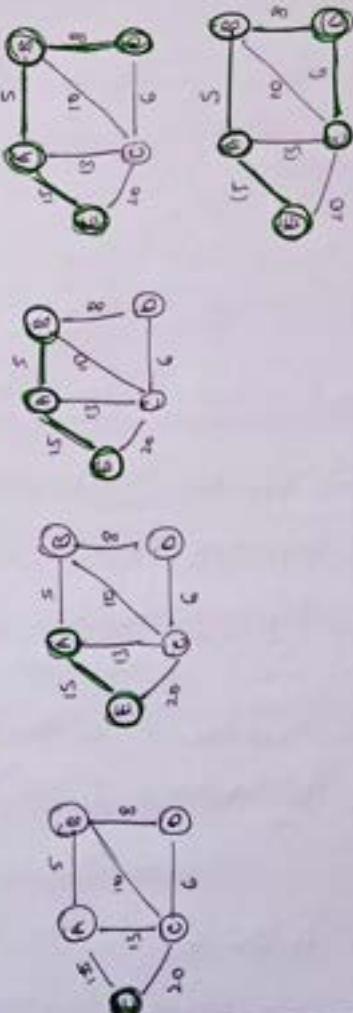


- landing cables
- TV network
- Train operation
- LAN Network
- A network of pipes for drinking water or natural gas
- An electric grid
- Single - link cluster



- concentrates on vertices
- finalize vertex in each iteration

Prim's  
Algo.



- network of roads & rail track connecting all the cities
- Telecommunication channels & placing microwave tower
- designing a bus-optic grid @ set
- Traveling Salesman Problem
- Cluster analysis
- Path finding algorithms used in mobile robotics

# Greedy Algorithm

- It is an algorithmic paradigm that builds the solution piece by piece
- In each step it chooses the piece that offers most obvious and immediate benefit.
- It fits perfectly for those problems in which local optimal points lead to global solutions.

(Ex) -> Brick-wall building.

Problems that can be solved using Greedy Algo.

- Insertion sort
  - Selection sort
  - Topological sort
  - Prim's Algorithm
  - Kruskal's Algorithm
  - Dijkstra's Algorithm
- 
- Activity Selection Problem
  - Coin change Problem
  - Fractional knapsack Problem.

| Inception point | Selection point | Topological point                                                                                                                                                                                                                                                                                                                                                                 | Kruskal Algo.                                                                                                                                                                                                                                                                                                                                                                      |
|-----------------|-----------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| S.A             | W.S.A           | <p>Here again we <math>\frac{G}{2}</math> into 2 parts</p> <p>@ each step we find the min. element &amp; put it in the perfect location in S.A.</p> <ul style="list-style-type: none"> <li>- w.l.o.g we do it into 2 parts i.e sorted &amp; unsorted</li> <li>- In the unsorted array we take the last element &amp; put it in the perfect location in the sorted part</li> </ul> | <ul style="list-style-type: none"> <li>- Here we are given a weighted undirected graph &amp; we need to find the min. spanning tree.</li> <li>- At each step we find the min. path among the many paths.</li> <li>- Here, @ each node we check whether the node again leads to a local optimum (L.O.) not. if not we push it to stack else we go to the next local opt.</li> </ul> |
| S.A             | W.S.A           | <p>2   3   4   5   7   8   6   9   1</p> <p>@ the middle of right look like this</p> <ul style="list-style-type: none"> <li>- This follows S.A as it aims to find the local optimum point @ each step which lead to global optimum</li> <li>- the find</li> </ul>                                                                                                                 | <p>5   3   4   7   2   8   6   9   1</p> <p>5   3   4   7   2   8   6   1   9   12</p> <p>By combining all local opt. we can obtain a global opt.</p> <p>So if there local opt. leads to a global opt.</p>                                                                                                                                                                         |

## Activity selection problem :

Given 'N' numbers of activities with their start and end times. we need to select the max. number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

| Activity | A <sub>1</sub> | A <sub>2</sub> | A <sub>3</sub> | A <sub>4</sub> | A <sub>5</sub> | A <sub>6</sub> |
|----------|----------------|----------------|----------------|----------------|----------------|----------------|
| Start    | 0              | 3              | 1              | 5              | 5              | 8              |
| Finish   | 6              | 4              | 2              | 8              | 7              | 9              |

2 Activities

But After a sort based on finish,

| Activity | A <sub>3</sub> | A <sub>2</sub> | A <sub>5</sub> | A <sub>4</sub> | A <sub>6</sub> |
|----------|----------------|----------------|----------------|----------------|----------------|
| Start    | 1              | 3              | 0              | 5              | 5              |
| Finish   | 2              | 4              | 6              | 7              | 8              |

4 Activities

### Algorithm

- sort activities based on finish time
- select first activity from sorted array & print it
- do all remain activities:

If the start time of this activity is greater than or equal to the finish time of previously selected activity the select this activity & print it

```

activities = [
 ["A1", 0, 6],
 ["A2", 3, 4],
 ["A3", 1, 2],
 ["A4", 5, 8],
 ["A5", 5, 9],
 ["A6", 3, 9]
]

```

★ TC  $\rightarrow \Theta(n \log n)$   
★ SC  $\rightarrow O(1)$

```

def print_max_activities(activities):
 activities.sort(key=lambda x: x[2]) # sorting the 2nd
 # list @ index 2
 i = 0
 first_A = activities[i][0] # setting 1st activity to firstAvailable
 print(first_A)
 for j in range(len(activities)):
 if activities[j][1] > activities[i][2]:
 print(activities[j][0])
 i = j
print_max_activities(activities)

```

A3

A2

A5

A6

## Coin Change Problem

- You are given coins of diff denominations & total amount of money.  
U have to find the minimum no. of coins that you need to make up the given amount.
- You can use the element infinite no. of times.
- Ex.  $\rightarrow \{1, 2, 5, 10, 20, 50, 100, 1000\}$

① Total amount : 40

Ans: 2 :  $50 + 20 = 40$

② Total amount : 122

Ans: 3 :  $100 + 20 + 2 = 122$

How's it happening :

③ Total amount : 2035

max num Bp 1000

$\therefore 2035 - 1000 = 1035$

$1035 - 1000 = 35$

$35 - 20 = 15$

$15 - 10 = 5$

$5 - 5 = 0$

result = 1000, 1000, 20,

10, 5

~~Result = 5~~  
Ans:

### Algorithm:

- Find the biggest coin that is less than given total number.
- Add coin to the result & subtract coin from total no.
- If V is equal to zero:  
Then print result  
else:  
Repeat step 2 & 3

```
def coinchange (totalnumber, coins):
 N = totalnumber
 coins.sort()
 index = len(coins) - 1
 while True:
 coinValue = coins[index]
 if N >= coinValue:
 print(coinValue)
 N = N - coinValue
 if N < coinValue:
 index -= 1
 if N == 0:
 break
```

$$\begin{array}{l} \textcircled{\ast} \text{ TC} \rightarrow O(n) \\ \textcircled{\ast} \text{ SC} \rightarrow O(1) \end{array}$$

coins = [1, 2, 5, 20, 50, 100]  
 coinchange (201, coins)

100  
200  
1

## Fractional knapsack Problem :

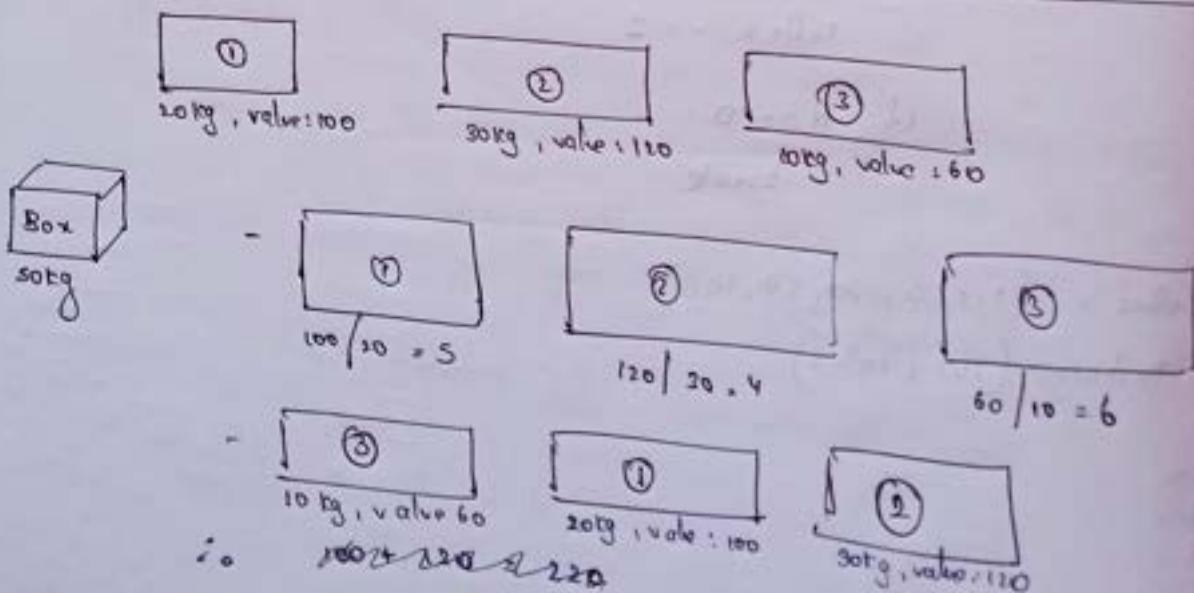
- Given a set of items,  
each with a weight & a value

We have to determine the no. of each item to include in a collection,  
so that the total weight is less than or equal to a  
given limit &

the total value is as large as possible.

### Algorithm:

- Calculate the density (or) ratio for each item
- sort items based on this ratio
- Take items with the highest ratios sequentially until weight allowed.
- Add the next item as much (fractional) as we can.



$$\begin{array}{r}
 60 + 100 + \\
 \underbrace{120 * 2/3}_{20\text{kg}} = 240
 \end{array}$$

⚡ for 30 kg → 120  
 $20\text{kg} \rightarrow 120 * \frac{20}{30}$

Clapp Item:

def init (self, weight, value):

$$\text{self}.weight = \text{weight}$$

$$\text{self}.value = \text{value}$$

$$\text{self}.ratio = \text{value} / \text{weight}$$

★ TC ~ O(nlg n)

★ SC ~ O(1)

def knapsackMethod (items, capacity):

items.sort (key = lambda x: x.ratio, reverse = True)

$$\text{usedCapacity} = 0 \quad \dots \quad O(n \log n)$$

$$\text{totalValue} = 0$$

for i in items: - - - O(n)

if usedCapacity + i.weight <= capacity:

$$\text{usedCapacity} += i.weight$$

$$\text{totalValue} += i.value$$

else :

$$\text{unusedWeight} = capacity - usedCapacity$$

$$\text{value} = i.ratio * unusedWeight$$

$$\text{usedCapacity} += unusedWeight$$

$$\text{totalValue} += value$$

if usedCapacity == capacity:

break

print ("Total value obtained : " + str(totalValue))

item1 = Item(10, 100)

item2 = -- 20, 120

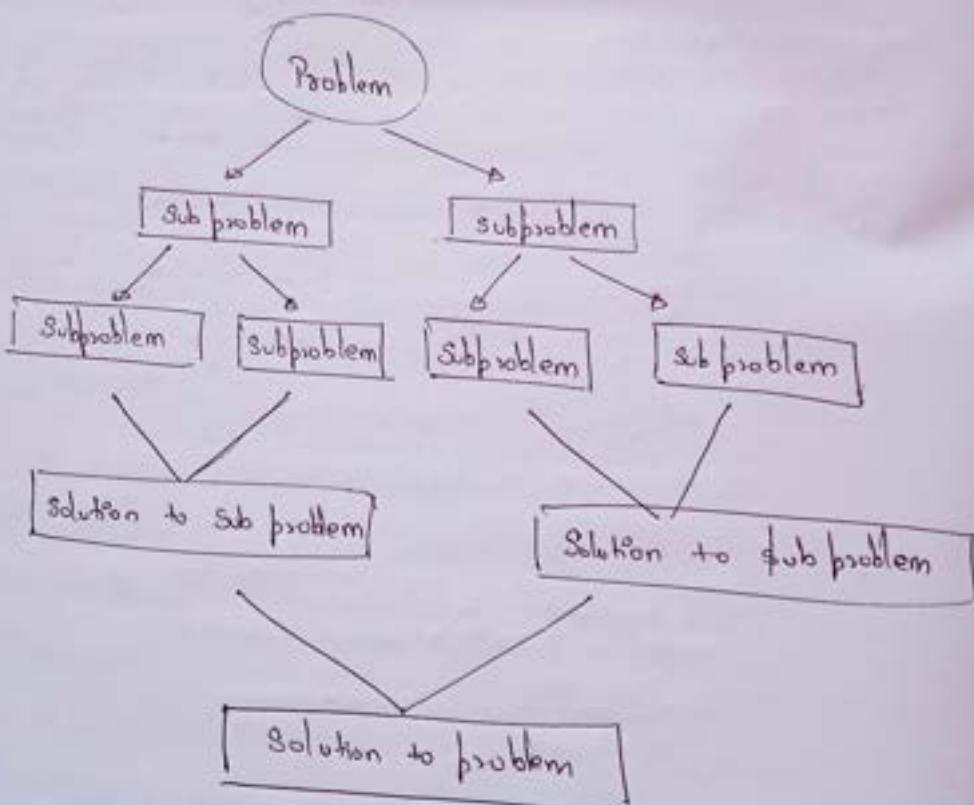
item3 = -- 10, 60

clist = [item1, item2, item3]

knapsackMethod(clist, 50)

# Divide & Conquer Algorithm

Divide and conquer is an algorithm, design which works by recursively breaking down a problem into subproblems of similar type, until these become simple enough to be solved directly. The solutions to the subproblems are then combined to give a solution to the original problem.



## Property of Divide and Conquer Algo.

### Optimal Substructure:

→ If any problem's overall optimal soln can be constructed from the optimal solns of its subproblems then that problem has optimal substructure.

$$\text{Ex} - \text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$$

### Why we need it?

→ It is very effective when the problem has optimal substructure property.

## Common Divide & Conquer Algo. been till now:

- Merge sort
- Quick sort
- Binary search algorithm.

## Fibonacci series - using OGC algo.

↳ A series of numbers in which each number is the sum of the 2 preceding numbers.  
→ If 1st 2 numbers are 0 and 1

e.g. → 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 ...

$$\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$$

```
def fibonacci(n):
 assert n >= 0 and type(n) == int
 if n < 0 or type(n) != int(n):
 print("The number should be a positive integer")
 if n == 0 or n == 1:
 return n
 else:
 return fibonacci(n-1) + fibonacci(n-2)
```

## (1) Number factors using OGC Algo.

Problem statement:

Given 'n', find the number of ways to express 'n' as a sum of 1, 3 and 4.

$$(Ex 1) \rightarrow n = 4$$

no. of ways = 4

Explanation = There are 4 ways we can express N

$$\{4\}, \{1,3\}, \{3,1\}, \{1,1,1,1\}$$

$$(Ex 2) \rightarrow n = 5$$

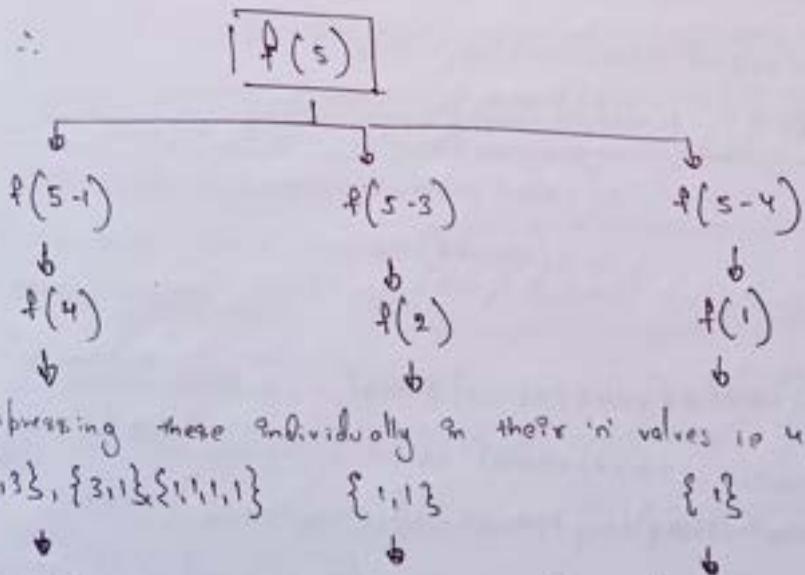
no. of ways = 6

Explanation = There are 6 ~

$$\rightarrow \{4,1\}, \{1,4\}, \{3,2\}, \{2,3\}, \{1,1,1,1,1\}, \{1,3\}, \{1,1,3\}$$

[ How's it happening? ]

Let's say  $n=5$  since we have to express  $n$  in 1, 3 and 4.



Expressing these individually in their 'n' values is 4, 2 & 1 respectively  
 $\{4\}, \{1,3\}, \{3,1\}, \{1,1,1,1\}$        $\{1,1,3\}$        $\{1,3\}$

Now expressing these in terms of  $f(n)$

$$f(4) + 1$$

$$f(2) + 3$$

$$f(1) + 4$$

$$\{4,1\}, \{1,3\}, \{3,1\}, \{1,1,1,1\}$$

$$\{1,1,3\}$$

$$\{1,3\}$$

```
def numberfactor(n):
```

```
if n in (0, 1, 2):
```

```
 return 1
```

```
elif n == 3:
```

```
 return 2
```

```
else:
```

```
 subP1 = numberfactor(n-1)
```

```
 subP2 = numberfactor(n-3)
```

```
 subP3 = numberfactor(n-4)
```

```
 return subP1 + subP2 + subP3
```

```
print(numberfactor(5))
```

6

# House Robber using DP Alg.

## Problem Statement:

- ↳ Given 'n' number of houses along the street with some amount of money.
- Adjacent houses cannot be stolen.
- Find the maximum amount that can be stolen.

Example 1:



Answer:

- Max. amount = 41
- Houses that are stolen = 1, 30, 4

Algo maxValueHouse(houses, currentHouse):

```
if currentHouse > length of houses
 return 0
```

else :

```
stealFirstHouse = currentHouse + maxValueHouse(houses,
 currentHouse + 2)
```

```
skipFirstHouse = maxValueHouse(houses, currentHouse + 1)
```

```
return max (stealFirstHouse, skipFirstHouse)
```

def houseRobber (houses, currentIndex):

```
if currentIndex >= len(houses):
 return 0
```

else :

```
stealFirstHouse = houses[currentIndex] + houseRobber(houses, currentIndex + 2)
```

```
skipFirstHouse = houseRobber(houses, currentIndex + 1)
```

```
return max (stealFirstHouse, skipFirstHouse)
```

houses = [6, 7, 1, 30, 8, 2, 14]

houseRobber(houses, 0))

$$6 + 8 + 4 = 18$$

1 + 30 + 4 = 35

# ( ) Convert string using D&C Algo. ( )

Problem statement :

↳  $s_1$  and  $s_2$  are given strings

↳ Convert  $s_2$  to  $s_1$  using Delete, Insert or replace operations

↳ find the minimum count of edit operations.

Example:

$s_1 = "table"$       } Delete  
 $s_2 = "tgable"$

$s_1 = "table"$       } Insert  
 $s_2 = "tble"$

$s_1 = "table"$       } Replace  
 $s_2 = "tble"$

findMinOperation ( $s_1, s_2, index_1, index_2$ ):

If  $index_1 == \text{len}(s_1)$

return  $\text{len}(s_2) - index_2$

If  $index_2 == \text{len}(s_2)$

return  $\text{len}(s_1) - index_1$

If  $s_1[index_1] == s_2[index_2]$

return findMinOperation ( $s_1, s_2, index_1+1, index_2+1$ )

Else:

deleteOp = 1 + findMinOperation ( $s_1, s_2, index_1+1, index_2+1$ )

insertOp = 1 + \_\_\_\_\_ ( $s_1, s_2, index_1+1, index_2$ )

replaceOp = 1 + \_\_\_\_\_ ( $s_1, s_2, index_1+1, index_2+1$ )

return min (deleteOp, insertOp, replaceOp)

```
def findMinOperation(s1, s2, index1, index2)
 if index1 == len(s1):
 return len(s2) - index2
 if index2 == len(s2):
 return len(s1) - index1
 if s1[index1] == s2[index2]:
 return findMinOperation(s1, s2, index1 + 1, index2 + 1)
 else:
 deleteOp = 1 + findMinOperation(s1, s2, index1, index2 + 1)
 insertOp = 1 + _____(s1, s2, index1 + 1, index2)
 replaceOp = 1 + _____(s1, s2, index1 + 1, index2 + 1)
 return min(deleteOp, insertOp, replaceOp)
```

```
print(findMinOperation("rat", "cat", 0, 0))
```

4

```
print(findMinOperation("table", "tbolt", 0, 0))
```

3

## Zero-One Knapsack Problem

Problem statement:

- Given the weights and profits of N items.
- Find the maximum profit within given capacity of C
- Items cannot be broken.

Example:

| Mango       | Apple       | Orange      | Banana      |
|-------------|-------------|-------------|-------------|
| Weight : 3  | Weight : 1  | Weight : 2  | Weight : 5  |
| Profit : 31 | Profit : 26 | Profit : 17 | Profit : 72 |

Knapsack Capacity : 7

Answer Combinations:

- Mango( $w: 3, p: 31$ ) + Apple ( $w: 1, p: 26$ ) + Orange( $w: 2, p: 17$ )  
 $= w: 6, \text{ Profit: } 74$
- Orange( $w: 2, p: 17$ ) + Banana ( $w: 5, p: 72$ )  $= w: 7, \text{ Profit: } 89$
- Apple ( $w: 1, p: 26$ ) + Banana ( $w: 5, p: 72$ )  $= w: 6, \text{ Profit: } 98$

zotknapsack (items, capacity, currentIndex):

If capacity  $c=0$  or currentIndex  $< 0$  or currentIndex  $> \text{len}(\text{profs})$   
return 0

Elif currentItemWeight  $\leq$  capacity

Profit1 = currentItemProf + zotknapsack(items, capacity -  
currentItemWeight, nextItem)

Profit2 = zotknapsack(items, capacity, nextItem)

return max(Profit1, Profit2)

Else

return 0

Class Item:

```
def __init__(self, profit, weight):
 self.profit = profit
 self.weight = weight
```

```
def zotknapsack(items, capacity, current_index):
 if capacity <= 0 or current_index < 0 or current_index >= len(items):
 return 0
 elif items[current_index].weight <= capacity:
 profit1 = items[current_index].profit +
 zotknapsack(items, capacity - items[current_index].
 weight, current_index + 1)
 profit2 = zotknapsack(items, capacity, current_index + 1)
 return max(profit1, profit2)
 else:
 return 0
```

mango = Item(31, 3)

apple = Item(26, 1)

orange = Item(17, 2)

banana = Item(72, 5)

items = [mango, apple, orange, banana]

print(zotknapsack(items, 7, 0))

# Longest Common Subsequence (LCS) using D&C Algo.

Problem Statement :

→  $s_1$  and  $s_2$  are given strings

→ find the length of the longest subsequence which is common to both strings.

→ Subsequence : a sequence that can be derived from another sequence by deleting some elements without changing the order of them.

Example : -  $s_1 = \text{"elephant"}$

-  $s_2 = \text{"eepat"}$

- output = 5

- longest string : "eepat"

Algorithm

findLCS( $s_1, s_2, \text{index}_1, \text{index}_2$ ):

if  $\text{index}_1 > \text{len}(s_1)$  or  $\text{index}_2 > \text{len}(s_2)$ :

return 0

if  $s_1[\text{index}_1] == s_2[\text{index}_2]$ :

return 1 + findLCS( $s_1, s_2, \text{index}_1 + 1, \text{index}_2 + 1$ )

else :

opt1 = findLCS( $s_1, s_2, \text{index}_1, \text{index}_2 + 1$ )

opt2 = findLCS( $s_1, s_2, \text{index}_1 + 1, \text{index}_2$ )

return max(opt1, opt2)

```
def findLCS(s1, s2, index1, index2):
 if index1 == len(s1) or index2 == len(s2):
 return 0
 if s1[index1] == s2[index2]:
 return 1 + findLCS(s1, s2, index1+1, index2+1)
 else:
 op1 = findLCS(s1, s2, index1, index2+1)
 op2 = findLCS(s1, s2, index1+1, index2)
 return max(op1, op2)
```

```
print(findLCS("elephant", "elephant", 0, 0))
```

5

# ○ Longest Palindromic Subsequence (LPS) using D&C

Problem Statement :

↳ S :- given string.

↳ find the longest palindromic subsequence (LPS)

↳ Subsequence : a sequence that can be derived from another sequence by deleting some elements without changing the order of them.

↳ Palindrome is a string that reads the same backward as well as forward.

Example : 1 :

- S = "ELPMENMET"

- Output = 5

- LPS : "EMEME"

findLPS (s, startIndex, endIndex):

If startIndex > endIndex :  
return 0

If s1[startIndex] == s2[endIndex]  
return 2 + findLPS (s, startIndex+1, endIndex+1)

else :

o<sub>1</sub> = findLPS (s, startIndex, endIndex-1)

o<sub>2</sub> = findLPS (s, startIndex+1, endIndex)

return max(o<sub>1</sub>, o<sub>2</sub>)

```
def findLPS(s, startIndex, endIndex):
 if startIndex > endIndex:
 return 0
 elif startIndex == endIndex:
 return 1
 elif s[startIndex] == s[endIndex]:
 return 2 + findLPS(s, startIndex+1, endIndex-1)
 else:
 op1 = findLPS(s, startIndex, endIndex-1)
 op2 = findLPS(s, startIndex+1, endIndex)
 return max(op1, op2)

print(findLPS("ELRMENNTER", 0, 8))
```

5

Min. Cost to reach the last cell in a 2D array using D&C Algo.

Problem Statement:

- 2D Matrix is given
- Each cell has a cost associated with it for accepting.
- We need to start from  $(0,0)$  cell and go till  $(n-1) \times (n-1)$  cell
- We can go only to right or down cell from current cell.
- find the way in which the cost is minimum.

Example:

|   |   |   |   |   |
|---|---|---|---|---|
| 4 | 7 | 8 | 6 | 4 |
| 6 | 7 | 3 | 9 | 2 |
| 3 | 8 | 1 | 2 | 4 |
| 7 | 1 | 7 | 3 | 7 |
| 2 | 9 | 8 | 9 | 3 |



|   |   |   |   |   |
|---|---|---|---|---|
| 4 | 7 | 8 | 6 | 4 |
| 6 | 7 | 3 | 9 | 2 |
| 3 | 8 | 1 | 2 | 4 |
| 7 | 1 | 7 | 3 | 7 |
| 2 | 9 | 8 | 9 | 3 |

~~start~~

Min. cost = 36

look @ 3 : Since u can either go up or left

since  $7 < 9 \therefore$  up

then from 7 either up or left

since  $3 < 4 \therefore$  right

:

u will reach to 4.

```

> def findMinCost (twoDArray , row , col):
 if row == -1 or col == -1:
 return float ("inf")
 elif row == 0 and col == 0: # we have reached 1st cell
 return twoDArray [0] [0]
 else:
 opt1 = findMinCost (twoDArray , row - 1 , col) # up
 opt2 = findMinCost (twoDArray , row , col - 1) # left
 return min (opt1 , opt2)
 twoDArray [row] [col] + min (opt1 , opt2)

```

$\text{twoDList} = \begin{bmatrix} [4, 7, 8, 6, 4], \\ [6, 7, 3, 9, 2], \\ [3, 8, 1, 2, 4], \\ [2, 1, 7, 3, 7], \\ [2, 3, 8, 9, 3] \end{bmatrix}$

$\text{print} (\text{findMinCost} (\text{twoDList} , 4, 4))$

# Number of paths to reach the cell with given Cost using D&C Algo.

Problem Statement:

- 2D Matrix is given
- Each cell has a cost associated with it for accepting
- we need to start from  $(0,0)$  cell and go till  $(n-1, n-1)$  cell.
- We can go only to right or down cell from current cell.
- we are given total cost to reach the last cell
- Find the no. of ways to reach end of matrix with given "total cost"

Example:

Total cost = 25

|   |   |   |   |   |
|---|---|---|---|---|
| 4 | 7 | 1 | 6 | / |
| 5 | 2 | 3 | 3 | / |
| 3 | 2 | 1 | 2 | / |
| 7 | 1 | 6 | 3 | / |
| / | / | / | / | / |

|   |   |   |   |
|---|---|---|---|
| 4 | 2 | 1 | 6 |
| 5 | 2 | 3 | 3 |
| 3 | 2 | 1 | 2 |
| 7 | 1 | 6 | 3 |

|   |   |   |   |
|---|---|---|---|
| 4 | 7 | 1 | 6 |
| 5 | 2 | 3 | 3 |
| 3 | 2 | 1 | 2 |
| 7 | 1 | 6 | 3 |

```

def number_of_paths(twoDArray, row, col, cost):
 if cost < 0:
 return 0
 elif row == 0 and col == 0:
 if twoDArray[0][0] - cost == 0:
 return 1
 else:
 return 0
 elif row == 0: # @ the 1st row & can go only to left
 return number_of_paths(twoDArray, 0, col-1,
 cost - twoDArray[0][col])
 elif col == 0: # can only go up
 return number_of_paths(twoDArray, row-1, 0,
 cost - twoDArray[row][0])
 else:
 op1 = number_of_paths(twoDArray, row-1, col, cost - twoDArray[row-1][col])
 op2 = number_of_paths(twoDArray, row, col-1, cost - twoDArray[row][col-1])
 return op1 + op2

```

TwoDList = [[4, 7, 1, 6],  
 [5, 2, 3, 9],  
 [3, 2, 1, 2],  
 [4, 1, 6, 3]]

int (number\_of\_paths (TwoDList, 3, 3, 25))